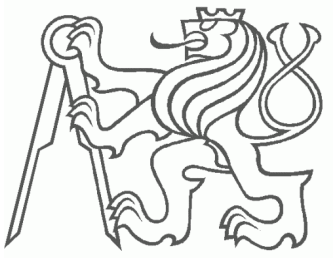
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA JADERNÁ A FYZIKÁLNĚ INŽENÝRSKÁ
Katedra matematiky

Bc. Jan Hofta

Úvod do mainframe



Název práce:
Úvod do mainframe

Autor: Bc. Jan Hofta

Abstrakt: Tento spis vznikl rozšířením mé bakalářské práce. Předtím v češtině neexistoval žádný soubornější přehled problematiky mainframů. Práce se pokouší přinést ucelené shrnutí základních informací o mainframech, podpořené vyzkoušením v praxi. Je rozdělena na pět částí. První část přináší úvod do oblasti mainframů. Ve druhé si můžeme přečíst o operačním systému z/OS. Třetí část shrnuje základní informace o JCL, jazyku pro řízení úloh. Čtvrtá část se věnuje programování v jazyce C/C++ na mainframech a konečně pátá část nám představí základy programovacího jazyka REXX. Celkově práce popisuje základy problematiky mainframů a může sloužit jako základní úvodní přehled pro všechny zájemce o tuto oblast informačních technologií.

Klíčová slova: mainframe, z/OS, JCL, C/C++, REXX

Title:
Introduction to Mainframe

Author: Bc. Jan Hofta

Abstract: This publication was created by extending my bachelor thesis. Before there did not existed any publication of the area of mainframes in Czech. This document tries to bring the coherent summary of the basic information of mainframes, supported by testing it in practice. It is divided into five parts. The first one covers the introduction to mainframes. In the second one is written about operating system z/OS. Basic information of JCL (job control language) is contented in the third one. The fourth part brings facts about programming in C/C++ on mainframes and, finally, programming language REXX is introduced in the fifth part. In conclusion, the thesis describes the basics of the area of mainframes and could serve as the basic introduction for all the people interested in this area of information technologies.

Key words: mainframe, z/OS, JCL, C/C++, REXX

Obsah

Úvod

Tento spis vznikl jako rozšíření mojí bakalářské práce. Jeho úkolem je shrnout a prakticky vyzkoušet základní poznatky o mainframech. Je určen všem zájemcům o získání základního přehledu v oblasti mainframů.

Mým cílem bylo napsat práci česky, a to včetně odborných termínů. Proto se snažím vyhýbat používání anglických slov, pokud pro daný objekt existuje české označení. V případech, kdy toto pojmenování čeština nemá, zkouším navrhnout vlastní české názvosloví. Pouze u některých věcí, kdy se v praxi běžně používá pouze jméno anglické (např. data set), ho uvádím i já.

Práce sestává z pěti částí. V první kapitole se čtenář seznámí se základními informacemi o mainframech, s jejich historií a použitím. Druhá část se věnuje operačnímu systému z/OS, který mainframy používají. Nalezneme tu nejen jeho teoretické vlastnosti, ale i popis ovládání uživatelského rozhraní či pravidla pro práci se soubory. Ve třetí kapitole se dozvíme, jak spouštět na mainframech úkoly pomocí jazyka JCL. Nachází se tu i několik příkladů úkolů napsaných pomocí JCL. Čtvrtá část popisuje úskalí programování v jazyce C/C++ na mainframech. Kromě podrobných popisů průběhu vytváření aplikace zde opět můžeme najít vyzkoušené příklady. Konečně v páté sekci, která nebyla původně součástí bakalářské práce, si představíme úplné základy programovacího jazyka REXX

Při práci mi velmi pomáhal pan Ing. Tomáš Oberhuber. Díky jeho přehledu o problematice jsem vždy věděl, ve kterém manuálu najít informace o konkrétních věcech. Cenné byly i jeho připomínky k napsanému textu, pomoc při spolupráci s externími odborníky a zajištění mainframu pro praktické zkoušení poznatků. Za to všechno mu velice děkuji.

Mé díky patří i panu Ing. Milanu Svátkovi z české pobočky společnosti CA za jeho cenné připomínky k podobě práce. Jako člověk, který již mnoho let s mainframy aktivně pracuje, mi pomohl odhalit chyby vzniklé nesprávnou interpretací manuálů a zpřesnit tak práci.

1. Úvod do problematiky

1.1. Co je to mainframe

1.1.1. Úvod do oblasti mainframů

Každý z vás jistě někdy použil počítač. Většinou však klasický osobní počítač, jaký se objevil až v osmdesátých letech minulého století. Společným úsilím firem IBM a Microsoft úplně zaplavily svět. Jenže už před nimi tu byly jiné stroje, a to sálové počítače. Mnozí lidé si myslí, že vývoj byl lineární a že sálové počítače byly těmi osobními úplně nahrazeny. Tak to ale není a důkazem tohoto tvrzení jsou právě dnešní mainframy. Ty přímo navazují na staré sálové počítače. Vyvíjely se *vedle* PC a plynule fungují už více než 40 let.

Z jakého důvodu tomu tak je? Vždyť dnešní PC jsou tak výkonná, že nějaké sálové obludy určitě nejsou potřeba a jistě se jimi dají nahradit.

Máte pravdu. Dají. Ale jen velmi těžko. Osobní počítače totiž nejsou vždy úplně spolehlivé, nedokáží rozumně pracovat s opravdu velkými objemy dat a třeba dlouhodobé využití procesoru na 95% pro ně představuje přinejmenším problém.

Oproti tomu mainframy jsou právě pro tyto účely vyráběny. V dnešní době už mainframe neznamená pouze krabici plnou procesorů, ale je to systém, který velké společnosti používají jako základ pro umístění svých databází, transakčních serverů a jiných aplikací, které vyžadují vyšší stupeň bezpečnosti a dostupnosti, než mohou nabídnout menší počítače. Není to jen hardware, ale celý směr informačních technologií. Mainframy mají vlastní operační systémy, na míru šité aplikace, které kladou důraz na bezpečnost a funkčnost a dokáží zpracovávat tisíce současných přístupů k datům. Vzhledem k tomu, že veškerý hardware je zdvojený, fungují tyto systémy 99,999% času, což si můžeme představit tak, že se průměrně zastaví pouze na pět minut v roce.

Přesto jejich existence není nikterak příliš známá. Dokonce i lidé, kteří se považují za dosti vzdělané v informačních technologiích, o nich často vůbec neví. To je dáno asi tím, že mainframů je na celém světě pouhých 10 000. Jejich vysoká spolehlivost totiž zvyšuje jejich cenu a proto je vlastní pouze asi tisíc největších světových společností (v České republice to jsou například Česká spořitelna nebo donedávna Škoda Auto). Na druhé straně ale investice do mainframů dosahují desítek miliard dolarů ročně a proto se určitě nejedná o nějakou okrajovou oblast informačních technologií.

1.1.2. Základní vlastnosti

Tři nejdůležitější vlastnosti mainframů shrnuje zkratka RAS, které odkazuje na anglické názvy pro:

- Spolehlivost (Reliability)
- Dostupnost (Availability)
- Provozuschopnost (Serviceability)

Tyto atributy by jistě měl mít každý počítač, ale u mainframů je na ně kladen nejvyšší důraz. Proto má mnoho součástí systému schopnost kontrolovat sebe sama a případně se i opravit (spolehlivost), v případě poruchy neponičit zbytek systému a nechat se nahradit paralelní součástí (dostupnost) a aniž by se musel běh operačního systému měnit, mohou být součásti nahrazeny novými (provozuschopnost). Takovýto systém potom většinou nemusí být vůbec odstavován z provozu kvůli vylepšením či opravám a když, tak pouze na velmi krátký čas. Tyto charakteristiky platí jak pro software, tak pro hardware.

Dalšími důležitými vlastnostmi jsou vysoká bezpečnost dat, přizpůsobivost systému novým vlastnostem (nové procesory, paměť apod.) a zpětná kompatibilita. Je téměř neuvěřitelné, že i na nejnovějších systémech fungují programy napsané před čtyřiceti lety.

1.2. Historie

1.2.1. Starší modelové řady

Stačí si připomenout jména jako ENIAC, von Neumann a podobně a dostaneme se k prvním počítačům v užším slova smyslu, které se objevily za druhé světové války. Jak vývoj pokračoval, tyto stroje se stále zdokonalovaly. Pro nás je důležitá tzv. třetí generace počítačů. Pro ni je charakterické, že dřívější tranzistory jsou nahrazovány integrovanými obvody, které byly vynalezeny v roce 1957. 7. dubna 1964 pak společnost IBM představila rodinu pěti vysoce výkonných strojů System/360, které představovaly počátek mainframových počítačů. Tehdy to byly jediné počítače, které jste si mohli pořídit. První typ System/360 (S/360) dokázal obsluhovat 44 periferních zařízení. V roce 1968 pro něj IBM vyvinula transakční systém zvaný CICS (Customer Information Control System, čteme [kiks]). Jedná se o aplikaci, která dovoluje on-line práci s daty. Až do dnešní doby zůstává CICS jedním z nejoblíbenějších prostředků pro sledování probíhajících transakcí. Není bez zajímavosti, že právě S/360 pomáhaly Apollu 11 s přistáním na Měsíci.

Obrázek 1: S/360 Model 40

IBM poté představila zhruba každých deset let nový model. V roce 1970 to byla rodina mainframů System/370 (S/370). Hlavní rozdíl od předchozích byl v tom, že S/370 mohly používat více procesorů (zpočátku obvykle dva), které se spolu dělily o paměť. Oproti S/360 byly tyto stroje větší a výkonnější. Prvně se v nich objevila plně integrovaná monolitická paměť a technologie virtuální paměti. Na počátku osmdesátých let přišly S/370XA, které přinesly 31-bitové adresování (dříve 24 bitů). Pro S/370XA byla v roce 1988 vyvinuta aplikace DB2, která má na starosti správu databází i na dnešních mainframech.

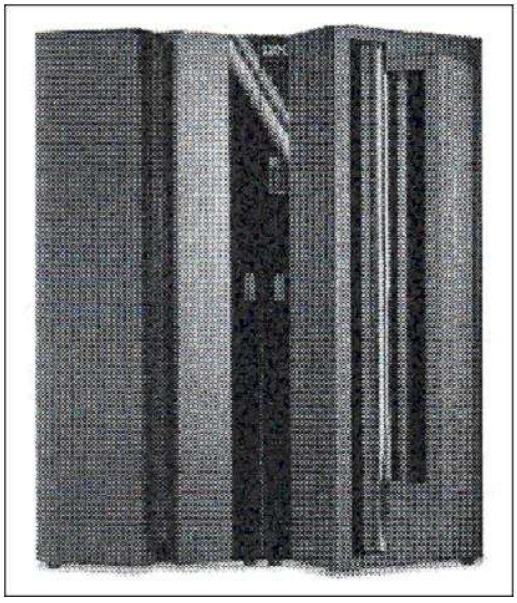
Devadesátá léta minulého století znamenala obrovský nástup osobních počítačů. Právě tehdy asi prožily mainframy svou největší krizi. Osobní počítače byly malé a stále výkonnější a mnohým se zdálo, že mainframy jim už nemůžou konkurovat. („Předpokládám, že poslední mainframe bude odpojen 15.dubna 1995.“ Stewart Alsop, Infoworld, duben 1991). Jenže IBM se s tímto stavem nemínila smířit. V roce 1990 přinesla rodinu System/390 (S/390). Tyto počítače si zachovaly spolehlivost a bezpečnost svých předchůdců, ale uvnitř byly úplnou novinkou. Jejich velikost se podstatně zmenšila, už nezabíraly samostatné sály, ale měly velikost větší ledničky. Také klesla jejich cena. Dalším důležitým vylepšením bylo zavedení

systému tzv. paralelního sysplexu. Tato unikátní technologie, zahrnující specielní software, hardware a komunikační kód, umožňuje mimo jiné velmi efektivně sdílet data a díky zdvojení všech zařízení (krom diskového pole) také zajišťuje vysokou dostupnost systému. V S/390 nechyběly ani nové typy procesorů. V roce 1999 se na těchto strojích poprvé objevila podpora Linuxu a také technologie „kapacita na požádání“ (Capacity on Demand). To znamená, že pokud potřebuje společnost větší výkon nebo více paměti na svém mainframu, stačí získat od IBM speciální kód. Ten dočasně zprovozní procesory či paměť, které jsou ve stroji fyzicky přítomné, ale neaktivní. Hardware se tedy vůbec nemusí měnit.

Od října 2000 nastupuje zatím poslední rodina mainframů, tzv. z/Series. Ze změn je asi nejdůležitější 64-bitové adresování a plná spolupráce s Linuxem.

1.2.2. Aktuální model

Aktuální model se jmenuje IBM z9 –109. Tento stroj dokáže pracovat až se šedesáti logickými oddíly (logical partition, LPAR), které se vzájemně neovlivňují. To znamená, že na něm funguje až šedesát nezávislých kopií operačního systému. V jednom mainframu můžeme nalézt až 54 procesorů. Všechny jsou fyzicky stejně postavené, ale díky různému mikrokódu mají různé funkce. Např. tzv. centrální procesor je používán pouze operačním systémem, svůj procesor tu mají i vstupy a výstupy, Linux či Java. Uvnitř je vše propojeno až 1024 fyzickými kanály. Například vnitřní síť v rámci jednoho počítače se nazývá HiperSocket a její rychlost je 20 GB/sec. Jako zajímavost si můžeme uvést, že IBM z9-109 obsahuje i šifrovací architekturu, která jako jediná na světě splňuje 4.úroveň amerického bezpečnostního standardu FIPS 140-1.



Obrázek 2 : IBM z9-109

1.3. Využití mainframů

1.3.1. Příklady využití

I když o nich obvykle vůbec nevíme, většina z nás mainframy využívá. Obyčejný automat, který se nám ozve, když zavoláme na naši bankovní linku, abychom zadali trvalý příkaz, může být napojený na nějaký mainframe. Ten by v tomto případě spravoval databázi klientů dané banky. Kromě finančnictví a bankovnictví bychom mohli vysledovat mainframy také v lékařství, pojišťovnictví a všude jinde, kde je třeba stovek či tisíců současných přístupů

do obřích databází. Pro jeden mainframe není problém zpracovávat terabajty dat. Sám o sobě má výkon stovek unixovských serverů.

1.3.2. Typy práce mainframů

Mainframy obvykle plní dva typy úkolů. Prvním z nich jsou tzv. dávkové úlohy (batch jobs). Je to typ práce, kdy na začátku mainframe dostane vstupní data (třeba několik terabajtů), nějakým způsobem s nimi naloží a vytvoří použitelný výstup. Například v bance má mainframe jako vstup databázi klientů a jako výstup to bude pravidelné měsíční vyúčtování pro každého z nich, statistická zpráva pro vedení podniku, zpráva pro společnost spravující kreditní karty a vytvoření záložní kopie dat. Obecně jsou dávkové úlohy charakterizovány velkým objemem vstupních dat, delší dobou běhu (minuty, hodiny), sestávají z většího počtu dílčích úkolů a vytvářejí informace pro velké množství uživatelů.

Druhou skupinu úkolů tvoří transakce v reálném čase. Jedná se o práci, kdy uživatel komunikuje prostřednictvím terminálu s mainframem a ten mu poskytuje žádané služby. Jako příklad můžeme uvést výběr peněz z bankomatu nebo webový systém rezervace vstupenek na koncerty. O tom, zda pro tuto činnost nasadit mainframe, nebo zda postačí obyčejný server, rozhodují převážně tři kritéria: počet klientů, kteří přistupují k systému v libovolném daném čase, počet transakcí za vteřinu a to, kdy má být daná služba k dispozici (24 hodin denně, 7 dní v týdnu). Obecně jsou transakce v reálném čase na mainframech charakterizovány malým objemem vstupních dat, velmi krátkou dobou běhu (méně než 1 sekunda), vysokým počtem transakcí prováděných naráz mnoha uživateli, vysokou bezpečností a časovou dostupností.

1.3.3. Personál kolem mainframů

Jak už jsme se mohli přesvědčit, mainframy jsou skutečně velmi komplexními systémy. Proto, aby vše fungovalo, jak má, je potřeba se o ně náležitě starat. K tomu je potřeba mnoho odborníků, z nichž každý má určitou funkci. Rozdělit bychom je mohli takto:

- Systémoví programátoři – péče se o operační systém jako celek (instalace, změny v nastavení, plánování výkonu)
- Systémoví administrátoři – každodenní údržba systému, někdy splývá se systémovými programátory
- Návrháři aplikací a programátoři – příprava nových programů pro mainframy
- Systémoví operátoři – správa velkých podsystémů, péče o správnou spolupráci mezi softwarem a hardwarem
- Analytici kontrolující výstupy – kontrola správnosti průběhu úkolů

1.4. Výhody a nevýhody

1.4.1. Výhody

Již v kapitole 1.1. jsme se zmínili o základních vlastnostech mainframů, o jejich vysoké spolehlivosti, dostupnosti a provozuschopnosti. To jsou beze sporu i jejich hlavní výhody. Především dostupnost je u těchto strojů skutečně obdivuhodná, běží 99,999% času, což si lze představit tak, že průměrná doba jejich odstávky za rok je pouhých 5 minut. Naproti tomu odpovídající doba pro unixovské servery je 23,6 hodin za rok. Výpadky a odstávky serverů jsou způsobeny jak plánovanými změnami, tak různými neplánovanými chybami a živelnými pohromami. U mainframů probíhají plánované změny přímo za běhu, aniž by je uživatel poznal. Při neplánovaných výpadcích se činnost systému díky zdvojení všech částí okamžitě převede na paralelní systém. Ten si může díky technologii „kapacita na požádání“ zvýšit své možnosti a vše funguje dál, většinou beze ztráty dat.

K přepnutí na paralelní systém a udržování aktuálnosti dat na obou systémech se využívá technologie GDPS. Díky ní se mainframe z havárie zcela „vzpamatuje“ za méně než

hodinu. GDPS je kombinací technologie paralelního sysplexu (sdílení dat a zdvojení součástí), synchronizovaného vzdáleného kopírování a automatického opravování chyb. Jedinou podmínkou pro fungování je, aby vzdálenost primárního a sekundárního systému nepřekročila 40 km.

Výhod je ale samozřejmě mnohem víc. Například dlouhodobé využití procesorů na více než 90% je pro mainframe, narozdíl od počítačů s Windows nebo Unixem, normální stav. Další výhody byly zmíněny v předchozích kapitolách.

1.4.2. Nevýhody

Nevýhodami mainframů se IBM už tolik nechlubí. Jedinou zmiňovanou je jejich cena, která je opravdu vysoká. Podle mého názoru jich je ale víc.

V kapitole 1.1. jsem psal o zpětné kompatibilitě mainframů až do roku 1964. Jistě je skvělé pouštět si na nejnovějších počítačích čtyřicet let staré programy, ale díky této filozofii je stále 60% programů napsaných v assembleru. V poslední době sice byla přidána podpora Javy a Unixu, ale jejich využití je stále velmi nízké. Stejně tak uživatelská rozhraní pro práci s nejrozšířenějším operačním systémem z/OS (budeme o něm mluvit v příští kapitole) jsou velmi zastaralá, pohybuje se v nich pomocí šipek (v lepším případě pomocí tabelátoru) a na různých místech tu přepisujeme obrazovku. To je podle mého názoru velmi nepraktické, snižuje to rychlost vývoje nových aplikací a naopak zvyšuje nároky na vývojáře a ostatní lidi, pohybující se kolem mainframů. Jistě to také podkopává snahu IBM získat do svého týmu mladé vývojáře.

Jako příklad místa, kde nevýhody převážily nad výhodami, můžeme uvést automobilku Škoda. Mainframe tam fungoval až do loňska, poté byl při reorganizaci koncernu Volkswagen vyřazen z provozu. Škoda místo něj pořídila několik stovek unixových systémů. Hlavními důvody odstavení mainframu byly:

- Přílišné náklady na provoz
- Málo programů běžících na mainframu, vysoké ceny vyžadované od IBM za nové programy
- Málo kvalifikovaných odborníků

I tak ale v koncernu Volkswagen ještě osm mainframů běží.

To byly konkrétní příčiny odstavení jednoho z mainframů v ČR. Ovšem jak moc se mainframy budou zdát výhodné nebo nevýhodné každému z vás se musíte rozhodnout na základě vaší práce s nimi sami.

2. Operační systém z/OS

2.1. Základní charakteristiky

2.1.1. Hardware

Každý lepší mobilní telefon má v dnešní době svůj operační systém (OS). Tím spíše ho mají i počítače. Operační systém je kolekce programů, díky kterým vůbec počítač funguje a která zprostředkovává spojení mezi jednotlivými částmi systému (hardwarem, aplikacemi).

Abyste si mohli spustit mainframovský operační systém, potřebujete k tomu příslušný hardware. Především je to mainframový počítač s procesory a takzvanou centrální pamětí[1]. Dnes používané OS bez problémů zvládají práci s více procesory, které se dělí o ostatní hardwarové prostředky systému. K těm patří paměťové disky (tzv. paměťová zařízení s přímým přístupem – direct access storage device, DASD), páskové paměťové mechaniky, děrné štítky (stále ještě) a uživatelské konzole, což bývají osobní počítače. Dříve se OS ovládal pomocí terminálu 3270, ale v dnešní době se již většinou používají jeho emulátory, běžící na osobních počítačích.

2.1.2. Různé operační systémy

Pokud máme připravený potřebný hardware, můžeme si představit samotné operační systémy. Už modely rodiny S/370 měly svůj OS. Jmenoval se MVS/370 a používal 24-bitové adresování, to znamená, že měl přístup k 2^{24} bajtů (tj. 16 MB). Adresování bude podrobněji popsáno v kapitole 2.2. Teď je důležité, že OS mainframů se stejně jako tyto stroje postupně vyvíjely, až v roce 2000 se společně s rodinou z/Series objevil i operační systém z/OS, který je v současné době naprosto dominantní.

Z/OS používá 64-bitové adresování paměti (viz dále) a mezi jeho úkoly patří především:

- Správa dat, jejich skladování, ukládání a načítání
- Péče o bezpečnost systému, kontrola přístupů k datům
- Péče o maximální využití možností systému, přidělování prostředků aplikacím a správa současného běhu více programů
- Síťová komunikace
- Poskytování služeb pro vývoj nových aplikací
- Podpora Unixu
- Poskytování služeb e-businessu
- Poskytování tiskových služeb

Mainframový počítač může fungovat ve dvou módech. V základním běží na jednom stroji pouze jeden z/OS. Můžeme ale také stroj logicky rozdělit na nezávislé logické části (tzv. LPAR) a na každé z nich provozovat samostatný operační systém. Tento způsob práce se nazývá LPAR mód.

2.1.3. Součásti z/OSu

Páteří tohoto operačního systému je aplikace nazvaná základní řídící program (base control program, BCP), která zprostředkovává nejzákladnější služby, například přesun aplikace k vyhodnocení na procesor prostřednictvím správce zátěže (workload manager,

[1] V angličtině se tato paměť nazývá různě, např. processor storage, central storage, real storage, real memory či main storage. Význam těchto názvů je zaměnitelný.

WLM), který je její částí. z/OS ale obsahuje daleko více prvků, jen základních elementů je přes třicet.

Celý z/OS je poskládaný z instrukcí pro různé činnosti, jako je přijetí práce, její zobrazení nebo tvorba výstupu. Soubor příbuzných instrukcí se nazývá rutina nebo modul. Příbuzné moduly pak tvoří dohromady systémovou komponentu, jako příklad si můžeme uvést již zmiňovaného správce zátěže.

Aby byl celý počítačový systém přehledně uspořádaný pro programy a aby se v rámci z/OSu dalo komunikovat, jsou všechny jeho části reprezentovány tzv. řídícími bloky. Každý z nich má určitou strukturu, která je programům známá. Rozeznáváme tři typy – řídící bloky spojené s operačním systémem, s prostředky a s požadavky. Každý řídící blok spojený s operačním systémem představuje jeden z/OS a obsahuje informace o systému, jako například kolik procesorů v současnosti používá. Řídící blok spojený s prostředkem podávává informace o daném nástroji (např. procesoru nebo paměťovém zařízení). Konečně řídící blok spojený s požadavkem reprezentuje jednotku práce.

2.1.4. Programy na mainframech

Než se pustíme do popisu toho, jak z/OS pracuje s programy a spravuje paměť, seznámíme se ještě s programy, které se na mainframech používají, ale nejsou součástí z/OSu. Jsou to hlavně:

- Bezpečnostní systémy (např. RACF)
- Překladače (z/OS sám o sobě obsahuje překladač jazyků C a assembler)
- Relační databáze (např. DB2)
- Nástroje pro sledování transakcí (např. CICS)
- Programy pro třídění velkého množství dat
- Další pomocné programy (např. SDSF, program který zobrazuje výsledky úloh, podrobněji bude popsán později)

Kromě těchto programů se na mainframech ještě často vyskytuje tzv. middleware, což jsou aplikace, které fungují mezi operačním systémem a koncovými programy. Přinášejí programům nové funkce, které OS poskytovat neumí. Jako příklad můžeme jmenovat databázové systémy, webové servery nebo virtuální stroje pro Javu.

2.2. Práce z/OSu s pamětí

2.2.1. Fyzická paměť

Jak už jsme si mohli všimnout, má z/OS k dispozici dvě místa, kam může fyzicky ukládat data. Zaprvé je to centrální paměť. Procesor do ní přistupuje přímo, současně se svým fungováním, je s ním de facto spojená. Na druhé straně tzv. vnější paměť se nalézá mimo mainframe na paměťových discích DASD, popřípadě na páskových paměťových mechanikách. Pokud procesor potřebuje něco z vnější paměti, musí vyslat požadavek a než mu požadovaná data přijdou, může se věnovat jiné práci. Tato paměť je tedy podstatně pomalejší, ale také levnější. Obě společně tvoří tzv. fyzickou paměť.

2.2.2. Virtuální paměť

Uživatel pracující se z/OSem do fyzické paměti přistupovat nemůže. Toto právo má pouze operační systém. Ten pro uživatele vytváří jinou paměť, tzv. virtuální. Virtuální paměť je iluze, kterou operační systém „přesvědčí“ každý program, že může přistupovat do celé paměti systému. Ve skutečnosti tomu tak samozřejmě není, programů běží pod operačním systémem celá řádka. I když je pro tento postup potřeba poměrně velká centrální paměť a

skutečně obrovská paměť vnější, ukazuje se, že je velmi výhodný a právě na něm je založena schopnost mainframů být používaný mnoha stovkami uživatelů najednou.

Řekli jsme, že program si myslí, že může přistupovat do celé paměti systému. Jak velká tato paměť je? Operační systém z/OS podporuje tzv. 64-bitové adresování. To znamená, že program může teoreticky využívat až 2^{64} bajtů. Toto číslo je skutečně obrovské. Připomeňme si proto nepříliš často používané předpony pro velikost fyzikálních jednotek. Ve světě z/OSu neznačí mocniny desítky, ale co nejbližší mocniny dvojky. Takže 2^{10} jsou kilobajty (kB), 2^{20} megabajty (MB), 2^{30} gigabajty (GB), 2^{40} terabajty (TB), 2^{50} petabajty (PT) a konečně 2^{60} exabajty (EB). V desítkové soustavě je u exabajtu za jedničkou 18 dalších míst. A právě 16 exabajtu dokáže z/OS opatřit adresami a tak je zpřístupnit programům. Je to zároveň i velikost virtuální paměti. Centrální paměť je ovšem daleko menší. Aby z/OS mohl vytvořit iluzi šestnácti exabajtů pro každý program, do centrální paměti načte pouze jeho část, která je zrovna aktivní. Zbytek schová ve vnější paměti. Když program potřebuje další kousek, OS ho pouze přesune, aniž by program cokoli poznal.

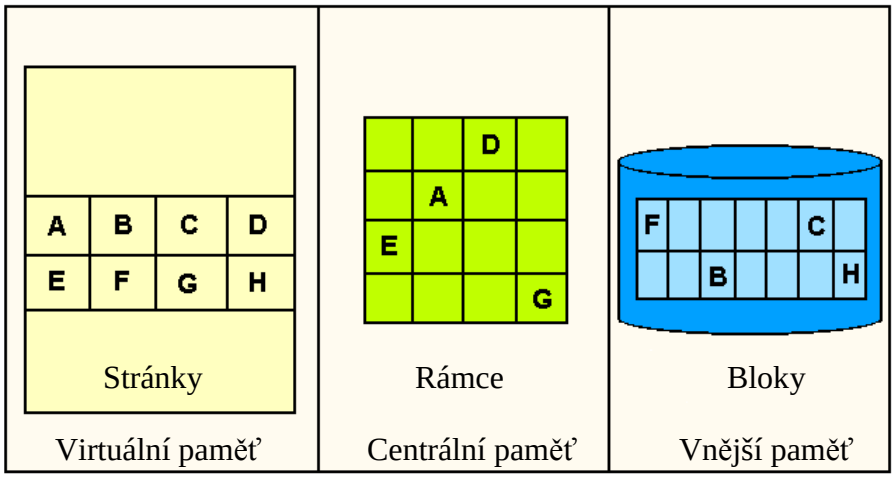
2.2.3. Stránky, rámce a bloky

Tyto kousky programu vznikají hned při jeho startu. Mají velikost 4 kB, nazývají se stránky (pages) a každý je vybaven unikátní virtuální adresou. Virtuální adresa slouží jako identifikátor dané stránky, nic neříká o tom, kde se zrovna stránka nalézá. Po celou dobu běhu programu se nemění. Stránka může být fyzicky uložená na dvou místech – buď v rámci (frame) nebo v bloku (slot). Jak centrální, tak vnější paměť jsou totiž také rozdělené na čtyřkilobajtové části. V centrální paměti je nazýváme rámce, ve vnější bloky.

Stránky se skládají z bajtů, které mají svou vlastní virtuální adresu.

Přesun stránek mezi rámci a bloky se nazývá stránkování a je plně pod kontrolou z/OSu. Ten do vnější paměti posílá stránky, které procesor už dlouho nepoužil a naopak do centrální přináší ty požadované. K tomu si udržuje zásobu použitelných rámců. Když je zásoba moc malá, nějakému uživateli jednoduše odešle stránku do vnější paměti a přivlastní si takto uvolněný rámec. Tento postup se nazývá kradení stránek (page stealing). Když program neběží (např. čeká na zadání dat), může se mu stát, že budou všechny jeho stránky přesunuté do vnější paměti. Tomuto procesu se říká swapování a z/OS je k němu vybízen svou již několikrát zmiňovanou komponentou správce zátěže (WLM).

Každému programu se pořád zdá, že je celý načtený v centrální paměti. Přehledně to je vidět na obrázku 3:



Obrázek 3: Stránky, rámce a bloky

Kde je která stránka opravdu umístěná, lze zjistit procesem dynamického překladu adresy (dynamic address translation, DAT). Ten se zavolá, pokud je nějaká stránka potřeba.

Když DAT zjistí, že stránka není v centrální paměti, upozorní z/OS a ten ji tam přemístí. DAT sestává z kombinace různých tabulek a vyrovnávací paměti.

2.2.4. Adresové prostory

Jak už jsme si několikrát řekli, na jednom mainframu může najednou pracovat velmi mnoho uživatelů. Každý z nich dostane od z/OSu při přihlášení tzv. adresový prostor, v rámci něhož může běžet jeden i více programů. Adresový prostor je sadou virtuálních adres, které mohou programy využívat, je to virtuální paměť daného uživatele. Každý adresový prostor používá stejné virtuální adresy, ale jejich přiřazení k fyzickým adresám v centrální paměti je samozřejmě různé. Virtuální adrese „10254000“ z adresového prostoru uživatele A odpovídá třeba adresa v centrální paměti „00971000“, stránku se stejnou virtuální adresou „10254000“ z adresového prostoru uživatele B pak z/OS umístí do rámce s adresou např. „0014A000“.

Koncept adresových prostorů je podobný unixovským identifikačním číslům procesů. Nicméně adresové prostory mají navíc několik výhod. Tou nejvýznamnější jsou tzv. služby skrz paměť (cross-memory services). Pomocí nich může uživatel přistupovat do některých (veřejných) oblastí adresového prostoru jiných uživatelů. Tento přístup je naprosto bezpečný a umožňuje rychlou komunikaci se službami, jako jsou například transakční a databázoví správci.

Adresový prostor se dělí na:

- Stránky o velikosti 4 kB
- Segmenty o velikosti 1 MB
- Regiony o velikosti 2 až 8 GB

Adresový prostor může obsahovat jeden region o velikosti 2 GB až osm miliard takovýchto regionů. Od jeho struktury je také odvozen tvar virtuální adresy stránek. Ta je rozdělená na šest částí. Bity 0 až 10 se nazývají první index regionu (region first index, RFX), bity 11 až 21 druhý index regionu (region second index, RSX), bity 22 až 32 třetí index regionu (region third index, RTX), bity 33 až 43 index segmentu (segment index, SX), bity 44 až 51 index stránky (page index, PX) a konečně bity 52 až 63 index bajtu (byte index, BX).

2.2.5. Vývoj velikosti adres

Operační systémy předcházející z/OSu nepoužívali 64-bitové adresování. MVS/370 z roku 1970 ho měl pouze 24-bitové, tím pádem měl adresový prostor velikost pouhých 16 MB. V roce 1981 se objevil OS MVS/XA, který přišel s 31-bitovou adresací paměti. Tím se virtuální paměť jednotlivých uživatelů zvětšila na přijatelné 2 GB. Jenže co se starými programy napsanými pro OS s 24-bitovými adresami? IBM, věrná své strategii absolutní zpětné kompatibility, to vyřešila velmi jednoduše. Poslední, třicátý druhý, bit v adrese MVS/XA se nepoužíval pro tvorbu adres, ale jako příznak. Jednička znamená adresování 31-bitové, nula staré 24-bitové.

Dnešní z/OS zvládá kromě 64-bitového i obě stará 24- a 31-bitová adresování.

2.2.6. Obsah adresových prostorů

Struktura obsahu adresového prostoru je této kompatibilitě přizpůsobená. Na prvních adresách do 16MB (tato hranice se nazývá „čára“, anglicky „line“) můžeme nalézt jak veřejné, tak soukromé oblasti. Neboť adresový prostor představuje celou paměť systému, jsou v obou oblastech některé adresy vyhrazené pro systémový kód a systémová data. Na ostatních se nalézá kód a data uživatele. Stejná struktura se opakuje mezi 16 MB a 2 GB (této hranici se říká „pruh“, anglicky „bar“) a následně mezi pruhem a maximální hranicí dvou exabajtů. Operační systém má tedy své části rozházené po celém adresovém prostoru. Kde se která část

nachází, můžete vidět v tabulce 1. Nevyšrafované části tabulky znamenají oblasti adresového prostoru, které jsou ve všech adresových prostorech stejné.

Vysoký uživatelský region	16 EB 512 TB
Adresy standardní sdílené paměti	2 TB
Nízký uživatelský region	4 GB
Rezervovaný prostor	2 GB, pruh
Rozšířené LSQA/SWA/229/230	
Rozšířený uživatelský region	
Rozšířené CSA	
Rozšířené PLPA/FLPA/MLPA	
Rozšířené SQA	
Rozšířený nucleus	16 MB, čára
Nucleus	
SQA	
PLPA/FLPA/MLPA	
CSA	
LSQA/SWA/228/230	
Uživatelský region	24 kB
Systémový region	8 kB
PSA	0

Tabulka 1: Obsah adresového prostoru

Popišme si nyní alespoň některé části adresového prostoru se součástmi operačního systému trochu podrobněji.

Oddíl nucleus představuje jádro operačního systému. Načítá se ihned při startu z/OS. Moduly, které nucleus obsahuje, se nalézají v data setu SYS1.NUCLEUS (data set představuje soubor na mainframech, věnuje se jim kapitola 2.5.). Po celou dobu běhu z/OS se nucleus nalézá v centrální paměti.

V SQA (system queue area) můžeme nalézt tabulky a fronty, spojené s celým systémem. Co přesně se tu nachází, se výrazně liší u jednotlivých instalací. Tato oblast musí být dostatečně velká, jinak systém začne přidělovat paměť v okolí, což vede k jeho spadnutí.

LSQA je to samé, ale pouze pro ten který konkrétní adresový prostor.

Další částí je CSA (common service area). Zde najdeme data, adresovatelná všemi aktivními adresovými prostory. To se hodí pro jejich vzájemnou komunikaci.

V LPA (link pack area) jsou uskladněny všechny moduly a systémové programy, určené pouze ke čtení. LPA dělíme na:

- PLPA (pagable, stránkovatelné), kde se nalézají moduly, které mohou být odsunuty pomocí swapování z centrální paměti
- FLPA (fixed, pevné), kde nalezneme naopak moduly, které centrální paměť opustit nesmí
- MLPA (modified, upravené), kam ukládáme moduly, které mohou být při startu systému použité pro úpravy těch z PLPA

Adresových prostorů systém vytváří mnoho. Jak už bylo řečeno, každý uživatel dostane od z/OSu svůj vlastní. Kromě toho existuje adresový prostor *MASTER*, který se vytváří při startu z/OSu a ve kterém běží základní služby OS. Své vlastní adresové prostory mají také některé podsystémy (např. JES - viz dále), middleware (CICS, DB2) i jednotlivé dávkové úlohy běžící na mainframech. Každý adresový prostor má své identifikační číslo (ASID), podle kterého ho z/OS může jednoznačně identifikovat.

2.3. TSO a ISPF

2.3.1. Uživatelská rozhraní

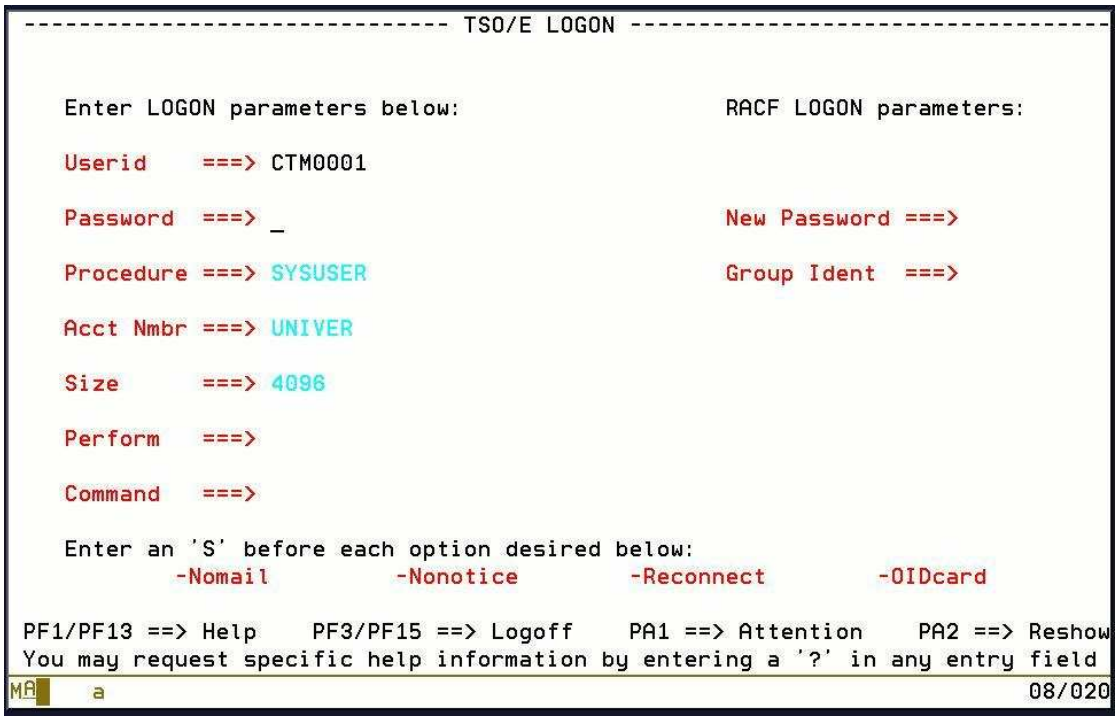
V předchozích dvou kapitolách jsme se seznámili se z/OSem teoreticky, v následujících třech nás čeká seznámení praktické.

Již jsme si řekli, že z/OS se typicky ovládá pomocí terminálu 3270, respektive jeho emulátorů pro osobní počítače. Když uživatel spustí terminál, bude první částí operačního systému, se kterou se setká, uživatelské rozhraní z/OSu. O komunikaci s uživatelem se zde stará součást zvaná Time Sharing Option/Extensions, pro kterou se používá především zkratka TSO/E nebo jen TSO.

S ní se pojí dvě základní uživatelská rozhraní[1]. Prvním je tzv. původní režim TSO. Je to řádkové rozhraní a mohli bychom ho přirovnat k příkazové řádce DOSu u osobních počítačů. Druhé uživatelské rozhraní je o něco sofistikovanější a jmenuje se Interactive System Productivity Facility (ISPF). V něm uživatel používá různá menu a panely. Panelem rozumíme předdefinované schéma, které zabírá celou obrazovku a na některých místech do něj lze vpisovat pokyny. ISPF se obvykle ovládá pouze pomocí klávesnice a s trochou fantazie bychom ho mohli přirovnat k Norton Commanderu v DOSu, jen místo souborů tu máme příkazy. TSO a ISPF používáme k instalaci nových programů, interaktivní komunikaci se systémem, zadávání úloh systému, vývoji aplikací, komunikaci s ostatními uživateli a zkrátka k veškeré naší interakci s operačním systémem.

2.3.2. Přihlášení do systému

Než začneme pracovat, musíme se nejprve do systému přihlásit. K tomu slouží úvodní obrazovka (přihlašovací panel), kterou můžete vidět na následujícím obrázku. Vzhledem k tomu, že každá instalace z/OSu je „originál“, mohou se nabídky částečně lišit.



Obrázek 4: Přihlašovací obrazovka

[1] z/OS ještě někdy nabízí třetí důležité rozhraní, které s ním umožňuje pracovat jako s unixovským systémem. Vzhledem k tomu, že jsem pracoval na mainframu, kde tohle uživatelské rozhraní nebylo, jsem neměl možnost ho vyzkoušet a proto ho ve své práci nezmiňuji. Případné zájemce mohu pouze odkázat na použitou literaturu.

Tento panel poskytuje přímo TSO. Důležité je na něm správně vyplnit vaše přihlašovací jméno (Userid) a heslo (Password). Po obrazovce se pohybujete šipkami a na správná místa píšete potřebné informace. Mezi těmito místy můžete také přímo přeskakovat pomocí tabelátoru. Pokud data náhodou napíšete vedle, může se terminál zastavit a vy ho musíte v lepším případě odblokovat klávesou Reset, která se na klávesnici zobrazuje obvykle na Esc (není to tedy tlačítko Reset pro restart počítače!), nebo v horším případě spustit znovu. Když zadáte špatné heslo, TSO vám to ohlásí a buď znovu zobrazí úvodní obrazovku nebo vy napíšete příkaz LOGON, který ji vyvolá.

Největším problémem při přihlašování (kromě zapomenutého hesla) je chvíle, kdy jste přihlášení a spadne vám spojení (vypnutá elektřina, vytažený síťový kabel apod.). Tehdy vás totiž z/OS neodhlásí, takže když se znovu pokoušíte přihlásit, vehementně vám tvrdí, že uživatel vašeho jména již přihlášen je a tudíž vás dál nepustí. Tento problém by asi správně měla řešit volba Reconnect na přihlašovacím panelu, ale na mainframu, kde jsem to zkoušel, vedlo napsání požadovaného „S“ před ní pouze k úplnému zamrznutí terminálu. Jak jsem se dozvěděl později, volba Reconnect funguje pouze výjimečně. Takže jediné, co mi zbývalo, bylo čekat pět hodin, po kterých mě konečně z/OS odhlásil automaticky.

Tento problém jako uživatel můžete vyřešit pouze používáním zvláštního softwaru, systémoví administrátoři však mohou mainframe nastavit tak, aby tento stav poznal už třeba za přijatelné dvě minuty – stačí se jim tedy ozvat.

2.3.3. Původní režim TSO

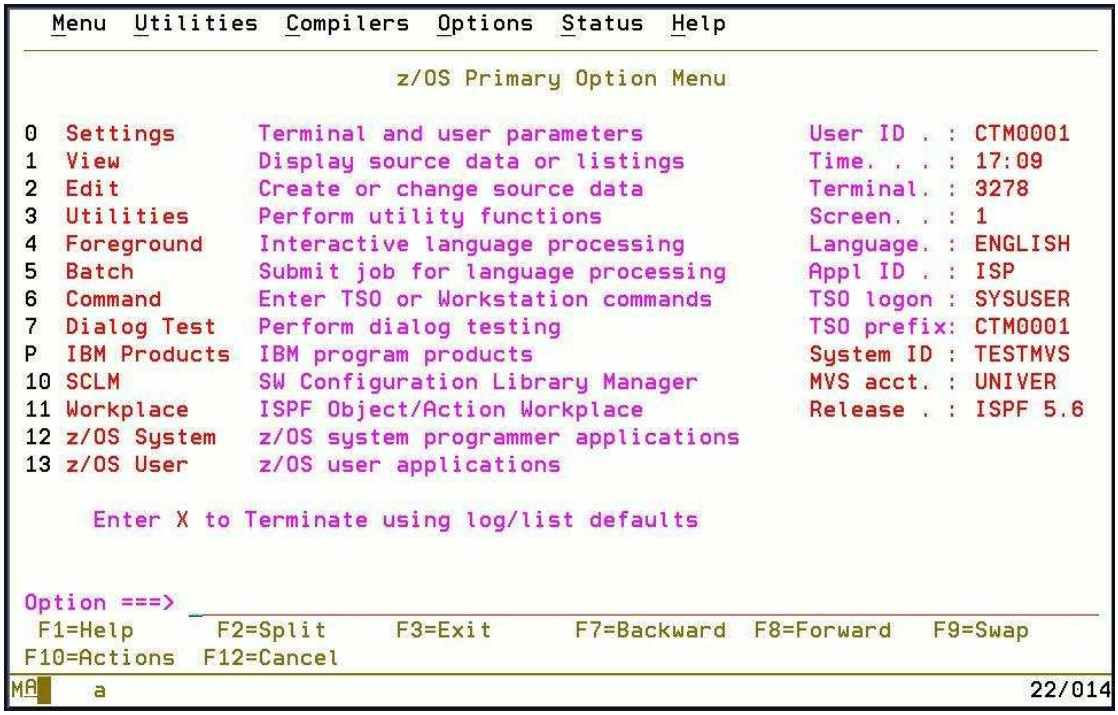
Po úspěšném přihlášení vypíše TSO informace o tom, kdy jste se přihlašovali naposledy a jestli má pro vás nějaké nové zprávy. Pak následuje buď automatický start ISPF (o něm si povíme za chvilku) nebo se objeví pouze nápis READY a pod ním kurzor. V druhém případě jste právě v původním režimu TSO.

Původní režim TSO zvládá základní příkazy, ale prakticky je lepší se hned přepnout do ISPF. Docílí se toho příkazem ISPF nebo PDF. Nicméně pracovat můžete i v původním režimu. Na začátku je na obrazovce napsané READY a pod ním kurzor. Na jeho místo vy zadáte příkaz a odešlete ho. TSO provede požadovanou činnost a vypíše o tom informace. Celý výpis zakončí hlášením READY a vy můžete psát další příkaz. Pomocí TSO lze například příkazem ALLOCATE vytvořit soubor (přesněji data set, řeč o tom bude v následující kapitole), příkazem RENAME ho můžete přejmenovat, HELP zobrazí nápovědu a TIME aktuální čas. Příkazy lze také spojit do skupin v rámci souboru, tzv. CLISTu. To se hodí pro spouštění pravidelně se opakujících činností. Jediný zádrhel je, že na psaní CLISTů potřebujete znát specielní příkazový jazyk CLIST, který používají pouze mainframy.

2.3.4. ISPF, menu základních voleb

Teď už ale přejdeme ke slíbenému ISPF. ISPF se poprvé objevilo v roce 1975 a od té doby pomáhá programátorům vyvíjet nové aplikace pro mainframy, uživatelům s nimi pracovat a správcům komunikovat s operačním systémem. Sestává ze čtyř hlavních komponent. Správce dialogu (Dialog Manager, DM) má na starosti komunikaci mezi počítačem a člověkem, přijímá požadavky uživatele, posílá mu zprávy a spravuje panely. Dalšími třemi jsou prostředek pro vývoj programů (Program Development Facility, PDF), správce softwarové konfigurace knihoven (Software Configuration Library Manager, SCLM) a komponenta agent pracovní stanice (Workstation Agent), která umožňuje fungování ISPF na terminálech a s využitím jejich operačního systému zobrazuje panely ISPF.

První věcí, kterou z ISPF uživatel spatří, bude menu základních voleb ISPF. Jak vypadá, máte možnost vidět na obrázku 5. Opět platí, že se může u jednotlivých instalací lišit.



Obrázek 5: Menu základních voleb ISPF

Hned po přihlášení se na tomto panelu navíc vlevo dole zobrazují informace o licencích, které zmizí stisknutím enteru.

Popišme si menu základních voleb trochu důkladněji, neboť ostatní menu a panely mu jsou většinou podobné. V dolní části se nalézá místo, kde je napsáno Option = = = >. Za tento nápis uživatel zadává příkazy. Část pod ním nám ukazuje, jak jsou přiřazeny funkční klávesy. Její zobrazení lze zapnout nebo vypnout příkazy PFSHOW ON, resp. PFSHOW OFF. Z funkčních kláves jsou asi nejdůležitější klávesa pro vyskakování z různých menu (typicky F3) a klávesy pro posunování textu dopředu a dozadu (F8, F7) a hodí se i klávesa pro rozdělení okna na dvě (F2). Vždy, když lze použít klávesy F7 a F8, nalezneme někde na obrazovce nápis Scroll = = = > a za ním úsek, o který se po jednom stisku klávesy posuneme. PAGE značí posun o stránku (obrazovku), HALF o půl strany a při volbě CSR se řádka, na níž je kurzor, přesune na první místo. Změnit to lze přepsáním a následným odesláním enterem (zadáme P pro PAGE, H pro HALF a C pro CSR).

Zcela nahoře menu základních voleb najdeme lištu s nástroji, kde například můžeme přepínat barevné zvýraznění textu a podobně upravovat panely. Pod některými názvy se skrývají stejné volby, jako v centrální nabídce (např. volba 3 Utilities a nástroj Utilities nabídnou totožné možnosti). Nejdůležitější částí je nabídka voleb uprostřed. Pro vybrání příslušného nástroje z lišty, respektive nabízené volby z menu, máme dvě možnosti. Buď najedeme kurzorem na jejich název a zmáčkneme enter nebo napíšeme do prostoru pro příkazy číslo naší volby a opět ho odešleme enterem.

Jako výsledek se u nástrojů z lišty obvykle objeví okno, které nezabírá celou obrazovku. V něm jsou uvedené další možné volby. Nějakou z nich zvolíme zase buď najetím na její název nebo napsáním jejího čísla do levého horního rohu okna na podtržené místo a následným odentrováním.

2.3.5. Nejdůležitější volby ISPF

Pokud vybereme volbu z prostřední nabídky, obvykle se dostaneme na další panel s podobným menu. Pokud s ISPF už nějakou chvilku pracujeme, můžeme si čísla našich oblíbených voleb zapamatovat a urychlit si tak práci. Pokud totiž do příkazového místa menu

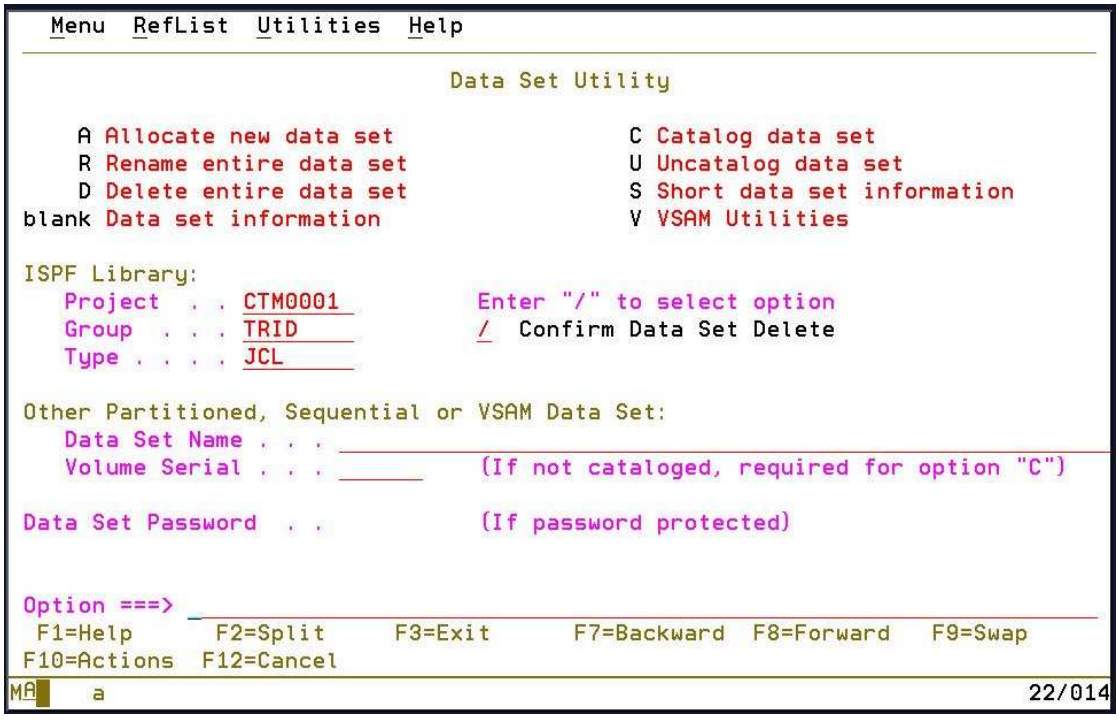
základních voleb zadáme např. 3.4, ISPF automaticky zobrazí volbu číslo 4 v menu, které by se zobrazilo volbou číslo 3 v menu základních voleb.

Nyní bychom se už měli po panelech ISPF umět pohybovat a proto si ukážeme, co tu lze najít. Samozřejmě je toho celá řádka a tak zmíníme jen několik základních voleb.

Volba 2 (Edit) otevře vstupní panel pro úpravy data setů. Data set je název pro soubor na mainframech a je jim věnována příští kapitola. Zatím nám stačí vědět, že jejich název se skládá obvykle ze tří částí, u typu PDS ze čtyř. Pokud chcete nějaký data set upravovat, zadáte části jeho jména postupně na místa Project, Group a Type (pro data sety PDS i Member, takto lze vytvářet také nové členy, ale to předbíháme). Může se stát, že bude mít jeho název více částí. V tom případě ho celý napíšeme na místo za Data Set Name v části panelu Other Partitioned, Sequential or VSAM Data Set. Po odentrování se otevře editor, ve kterém můžeme existující data set modifikovat. Jak se editor ovládá, popíšeme v kapitole 3.1., v následujících se také dozvíte, co do data setů psát, aby obsahovaly něco užitečného.

Volbou 3 (Utilities) se dostaneme do menu s nabídkou pomocných operací. Z nich jsou nejdůležitější volba číslo 2 (Data Set Utility) a číslo 4 (DSLIST).

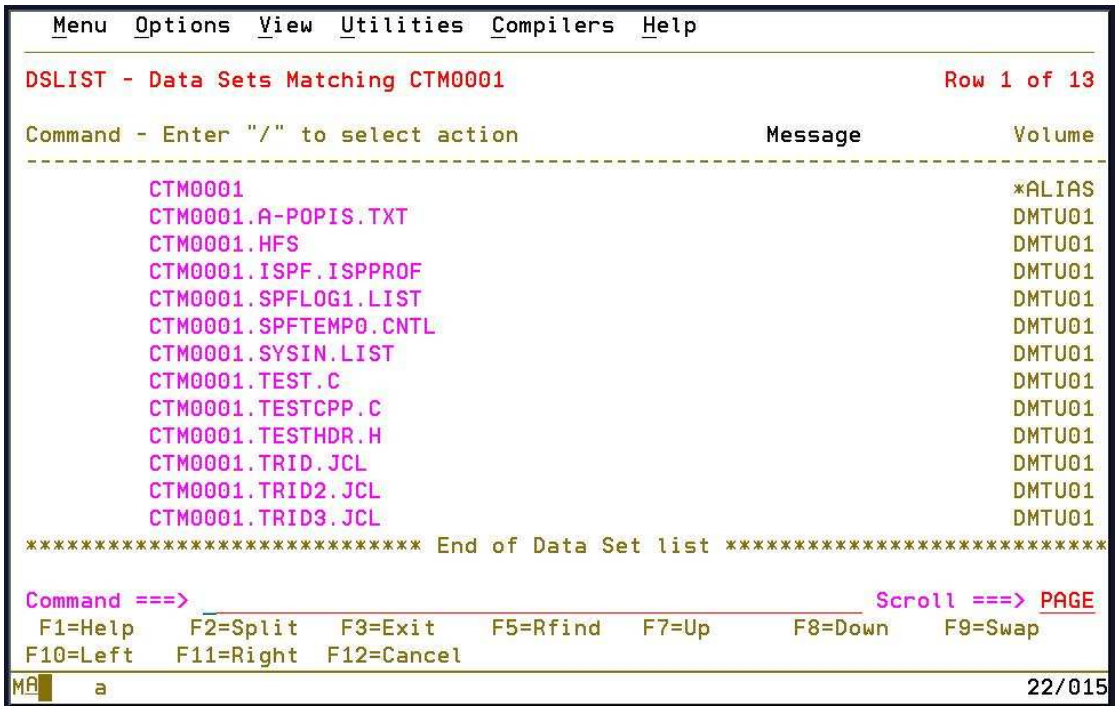
Data Set Utility je nástroj pro správu data setů, umožňuje např. tvorbu nových a přejmenovávání a mazání starých data setů. Jak vypadá jeho panel, můžeme vidět na obrázku 6. Především tvorba nových data setů je velmi důležitá a bude podrobně popsaná v příští kapitole. Data Set Utility se ovládá tak, že na místa Project, Group a Type postupně napíšeme části jména požadovaného data setu (pokud má název více částí, tak za Data Set Name) a poté za nápis Option = = = > písmeno z nabídky nahoře. Vše odešleme jako obvykle enterem.



Obrázek 6: Panel Data Set Utility

Pod volbou 3.4 nalezneme DSLIST. Je to nástroj, který dokáže vypsat seznam data setů a potom s nimi dál pracovat. Na panelu, který se po jeho spuštění objeví, např. stačí za nápis Dsname Level zadat první část jména data setu, která obvykle značí vlastníka, a DSLIST zobrazí všechny naše data sety. Může také zobrazit jen některé, pokud zadáme i druhou část jména (lze použít *, ** a % pro nahrazení znaků a jejich posloupností). Ve výsledném seznamu (obrázek 7) lze s data sety dále pracovat a to pomocí příkazů psaných

vlevo vedle jejich názvu. Například E zobrazí členy data setu typu PDS, S je vybírá a D maže. Pokud napíšete /, otevře se okno, ve kterém je kompletní nabídka operací.



Obrázek 7: Výpis data setů pomocí DSLIST

Další důležitou aplikací, kterou lze pomocí ISPF spustit je SDSF, o kterém se více zmíníme v souvislosti s úlohami. Pod kterou volbou ji uživatel nalezne, hodně závisí na konkrétní instalaci z/OSu.

2.4. Data sety

2.4.1. Co je data set

Data set je kolekce logicky souvisejících dat, tedy něco jako soubor na osobním počítači. Může obsahovat zdrojový kód k programu, knihovnu maker nebo třeba textové záznamy, které je třeba setřídit.

Oproti osobním počítačům nejsou data sety uložené ve stromové struktuře[1], kterou ale můžeme vhodným pojmenováváním simulovat (viz dále). Jeden data set se může nalézat i na více DASD discích. Uživatel, který chce data set používat, potřebuje znát pouze jeho jméno, nikoli umístění.

2.4.2. Pojmenovávání data setů

Pojmenovávání data setů má určitá pravidla. Jméno může být nejvýše 44 znaků dlouhé. Skládá se z několika částí (obvykle tří či čtyř, ale lze až z dvaceti dvou), tzv. kvalifikátorů. První kvalifikátor se nazývá kvalifikátor nejvyšší úrovně, poslední je kvalifikátorem nejnižší úrovně. Kvalifikátory jsou oddělené tečkami a každý sestává z maximálně osmi znaků, z nichž první musí být písmeno (velké) nebo symbol @, # či $. Následující znaky mohou být navíc i číslice či pomlčka.

Jak konkrétně se bude váš data set jmenovat, záleží pouze na vás, ale i tady se vžily určité konvence. Kvalifikátor nejvyšší úrovně je obvykle tvořen uživatelským jménem

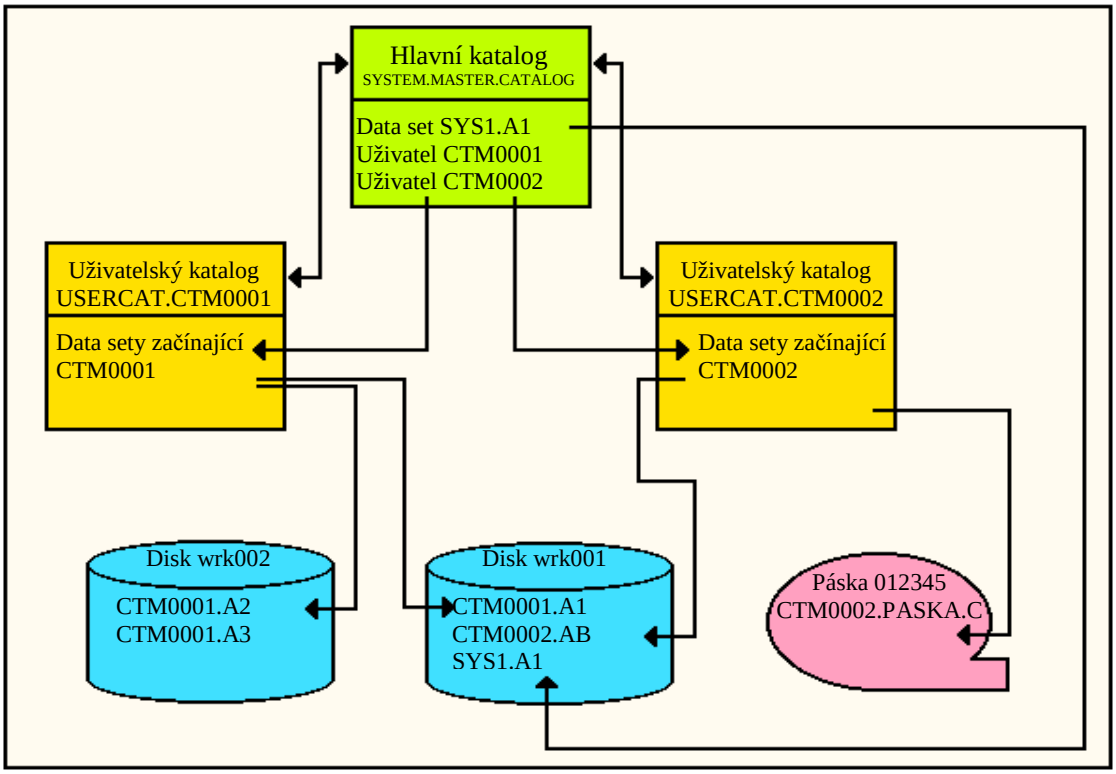
[1] Opět nemluvíme o systému souborů z/OS Unixu.

uživatele, který data set vytvořil. Pokud se v některém kvalifikátoru objeví písmena LIB, znamená to, že data set je knihovnou. Příkazy JCL (viz další kapitoly) v data setu oznamují zkratky JCL, JOB nebo CNTL, procedury z příkazů JCL jsou indikovány písmeny PROC, PRC či PROCLIB. Také zdrojové kódy v různých programovacích jazycích uložené v data setu se projeví na jejím názvu (např. písmeno C jakožto kvalifikátor nejnižší úrovně bude pravděpodobně značit zdrojový kód jazyka C/C++). Příklady platných a použitelných názvů data setů jsou CTM0001.POKUS-1.JCL nebo ZPROF.ZSCHOLAR.LIB.SOURCE a mnohé další.

2.4.3. Ukládání data setů, katalogy

Data sety se ukládají na paměťové disky DASD. Samozřejmě si můžeme sami ručně nastavit, kam data sety uložíme, mnohem praktičtější je však nechat to za nás udělat operační systém, respektive jeho komponentu DFSMS (Data Facility Storage Management Subsystem). Ta nejen zajistí potřebné místo na discích, ale také náš data set zapíše do tzv. katalogu.

Katalog je obyčejný data set, ve kterém jsou uložené ukazatele na příslušné zapsané data sety. U každého data setu je tu napsáno, na kterém disku DASD se nachází (pomocí jména disku, tzv. Label, což je šest znaků v úplně prvním záznamu na disku). Díky katalogu uživatel při přístupu do požadovaného data setu nepotřebuje vědět, kde je uložen, ale stačí mu znát pouze jeho jméno. Katalogů bývá obvykle víc. Umístění konkrétních data setů na discích se zapisuje do uživatelských katalogů. V každém uživatelském katalogu jsou zapsané informace o data setech jednoho nebo obvykle více uživatelů (podle kvalifikátoru nejvyšší úrovně). Hlavní katalog potom obsahuje dva typy záznamů – s ukazateli na některé data sety a s ukazateli na příslušné uživatelské katalogy. Příklad této struktury můžeme vidět na obrázku 8:



Obrázek 8: Struktura katalogů

2.4.4. Struktura disků, formáty záznamů data setů

Při vytváření nového data setu je důležité vědět, kolik prostoru by měl zabírat. Disky DASD jsou rozdělené na cylindry (cylinder), ty dále na stopy (track) a ty na záznamy (records). Pomocí všech těchto jednotek (stejně jako pomocí bajtů) můžeme vybrat požadovanou velikost našeho data setu (která není neměnná, viz dále). Nejdůležitějšími z nich jsou záznamy. Jejich velikost je různá, záleží jak ji vybereme.

Záznamy rozdělujeme na dva typy - logické záznamy a bloky. Logické záznamy jsou jednotlivé informace o daném problému, které lze samostatně zpracovávat (např. číslo účtu zákazníka banky nebo řádek v textovém souboru). Logické záznamy se dají spojit do skupin a fyzicky uložit a právě těmto uloženým skupinám logických záznamů říkáme bloky.

Podle vztahů mezi bloky a logickými záznamy rozdělujeme záznamy data setů na mnoho různých formátů. Pět nejdůležitějších přehledně zachycuje následující tabulka.

Formát	Zkratka	Počet logických záznamů v bloku (N)	Velikost záznamů (LRECL)	Velikost bloků (BLKSIZE)
Pevný	F	Jeden	Stejná	LRECL
Pevný blokový	FB	Několik	Stejná	N x LRECL
Proměnný	V	Jeden	Různá	Více než LRECL
Proměnný blokový	VB	Několik	Různá	Více než 4+N x LRECL
Nedefinovaný	U	Jeden	Různá	Nemá strukturu

Tabulka 2: Formáty záznamů data setů

Nejpoužívanějšími formáty jsou FB a VB. U formátů V a VB je před každým logickým záznamem uložen ještě čtyřbajtový identifikátor, ve kterém je zapsána délka logického záznamu. U formátu VB je na začátku každého bloku podobný identifikátor s jeho délkou.

2.4.5. Typy data setů

Data sety mohou být mnoha typů. Tři nejpoužívanější typy jsou sekvenční data sety, knihovny (partitioned data set, PDS) a data sety VSAM.

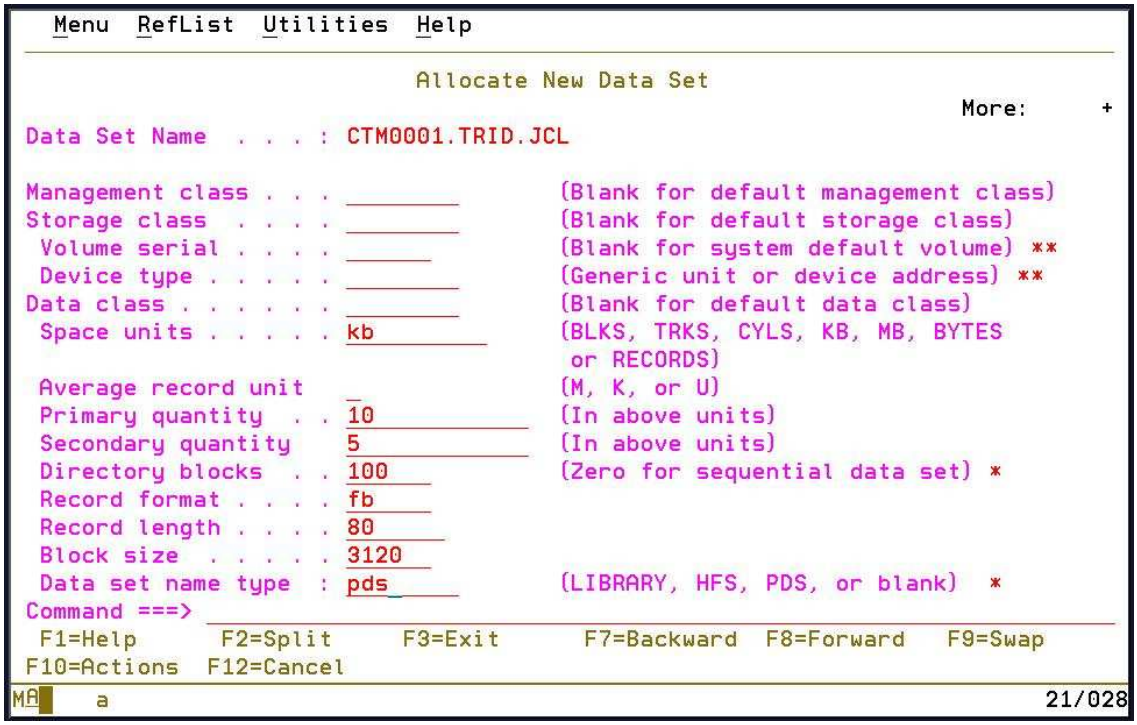
Sekvenční data sety jsou nejjednodušší. Sestávají ze záznamů, které se fyzicky ukládají postupně za sebou. Pouze tento typ data setů lze uložit na magnetické pásky.

Knihovny nebo též data sety PDS se skládají z několika sekvenčních data setů, které v tomto případě nazýváme členy. Každý člen má své jméno, které musí splňovat stejné požadavky, jako libovolný kvalifikátor v názvu data setu. Nové členy můžeme vytvářet například pomocí volby 2 v ISPF. Výhodou data setů PDS je, že uživatel může naráz pracovat se všemi členy i s každým zvlášť. Představit si je můžeme jako adresář, obsahující sekvenční data sety. PDS je velmi snadné vytvořit a pomocí ISPF se s nimi dá jednoduše pracovat. Hodí se také jako komprimační nástroj – pomocí nich můžeme uložit mnoho malých sekvenčních data setů do jedné stopy na disku (nemusí mít každá svou vlastní, jako by to bylo bez nadřazené PDS). Mezi jejich nevýhody patří neuvolňování místa na disku po smazání člena, pomalost při práci s velkými PDS a omezenost maximální velikosti data setu. Vylepšené rozšířené knihovny se nazývají PDSE a odstraňují většinu nevýhod PDS.

Data sety VSAM jsou úzce spojeny s metodou přístupu k datům. Dělí se na několik typů, obvykle má každý záznam u těchto data setů svůj identifikátor, díky němuž lze k záznamu náhodně přistupovat. Podobají se tak struktuře pole v programovacích jazycích. Data sety VSAM slouží pouze pro aplikace a nedá se s nimi pracovat prostřednictvím ISPF. Podrobněji se s nimi seznámíme v následující kapitole 2.5..

2.4.6. Tvorba nových data setů

Nyní již o data setech něco víme a proto můžeme zkusit nějaký vytvořit. Existuje víc způsobů, např. pomocí příkazu ALLOCATE v původním módu TSO nebo pomocí příkazů JCL, které budou vysvětleny později. Nejjednodušší je to ale pomocí volby 3.2 v ISPF. Po jejím zadání se objeví panel Data Set Utility. Za nápis Project napíšeme požadovaný kvalifikátor nejvyšší úrovně, tedy první část názvu našeho data setu, obvykle uživatelské jméno. Následují další kvalifikátory z požadovaného jména napsané za slova Group a Type. Vyplňování tohoto panelu zakončíme napsáním A za Option = = = > a jeho odesláním enterem. Pokud jsme zadali platné jméno nového data setu, objeví se panel Allocate New Data Set, který můžete vidět na obrázku 9. Zde je už vyplněný údaji, které by měly vést k vytvoření data setu, který se jmenuje CTM0001.TRID.JCL. CTM0001 je v tomto případě naše uživatelské jméno a již z názvu je vidět, že v tomto data setu budou uloženy příkazy JCL.



Obrázek 9: Tvorba nového data setu

Popišme si panel trochu podrobněji. Prvních pět kolonek zůstalo nevyplněných, tam by se dalo specifikovat, kam chceme náš data set uložit. To za nás obstará DFSMS. Jako jednotky velikosti jsme se rozhodli pro kilobajty, ale mohli bychom volit také záznamy či cokoli jiného z nabídky. Na řádku Primary quantity zadáváme, kolik má mít data set velikost na začátku, Secondary quantity nám říká, kolik paměti se přidá, pokud nebude Primary quantity stačit. Přidávat se může u starších typů data setů až šestnáctkrát, u novějších až 128 či 255krát. Může se to zdát zbytečně složité, ale zvyšuje to výkonnost aplikace. Systém totiž přesně ví, kde jsou části dat a nemusí hledat jejich konce. Také to snižuje fragmentaci disků. Řádek Directory blocks je důležitý pro data sety PDS, neboť na něm specifikujeme, jak velká je oblast, popisující jednotlivé členy. Je lepší zadat sem větší číslo, neboť tento parametr nelze později měnit. Řádek Record format udává formát záznamů (viz tabulka 2), následuje velikost logického záznamu, bloku a typ vytvářeného data setu. Hodnoty zadané do panelu na obrázku 9 by měly vést k vytvoření nového použitelného data setu.

2.4.7. Generační soubory

Pokud používáme stále stejná data a jen je během času aktualizujeme, můžeme jednotlivé verze nebo též generace data setů spojovat do generačních souborů (Generation data group, GDG). GDG je kolekce historicky spojených data setů (nikoli VSAM), uložených v chronologickém pořádku. Výhodou tohoto postupu je možnost volat všechny data sety pouze jedním jménem nebo možnost přikázat systému, aby staré generace automaticky mazal.

Technicky tento servis zabezpečuje katalog. Ke jménu data setu se při zápisu do katalogu připojí číslo, které udává, jak moc aktuální verze dat to je. Aktuální generace má číslo nula a čím starší verze, tím nižší celé záporné číslo.

2.5. Data sety VSAM

2.5.1. Základní charakteristika

V podkapitole 2.4.5. jsme si řekli, že existují zvláštní data sety VSAM. Nyní se s nimi seznámíme blíže. Zkratka VSAM značí metodu pro přístup k virtuální paměti (Virtual Storage Access Method, VSAM). Odkazuje tím na vlastnost charakterizující data sety VSAM a to sice na jejich specielní strukturu. Oproti běžným data setům, které slouží pouze jako úložiště dat, mají totiž data sety VSAM zvláštní stavbu, vzdáleně podobnou datové struktuře pole, známé z běžných programovacích jazyků. Stejně jako pole obsahují záznamy, ke kterým můžeme náhodně přistupovat pomocí jejich identifikátorů.

Vnitřní logická stavba data setů VSAM je také zajímavá. Záznamy jsou logicky uloženy v datových oblastech zvaných kontrolní intervaly (Control Interval, CI), které mají velikost 4 až 32 kB a které navíc obsahují popis jednotlivých záznamů (Record Descriptor Fields), popis celého kontrolního intervalu (CI Descriptor Field) a prázdné místo. Fyzické uložení dat z jednotlivých kontrolních intervalů jejich strukturu však nekopíruje. Několik kontrolních intervalů tvoří kontrolní oblast (Control Area, CA) a ty pak dále společně se záznamy indexů celý data set VSAM.

Do data setů VSAM nemůžeme vstupovat prostřednictvím běžných nástrojů ISPF. Pro jejich ovládání existuje zvláštní nástroj jménem IDCAMS, který dokáže spravovat data uložená v těchto data setech. Volat ho lze například pomocí různých programů nebo pomocí úlohy, což je nejsnazší. Několik jeho základních příkazů si popíšeme v podkapitole 2.5.4..

2.5.2. Typy data setů VSAM

Data sety VSAM se dělí podle své struktury na několik typů. Nejběžnějšími typy jsou následující čtyři:

- KSDS soubory (Key-Sequenced Data Set, KSDS)
- ESDS soubory (Entry-Sequenced Data Set, ESDS)
- RRDS soubory (Relative Record Data Set, RRDS)
- LDS soubory (Linear Data Set, LDS)

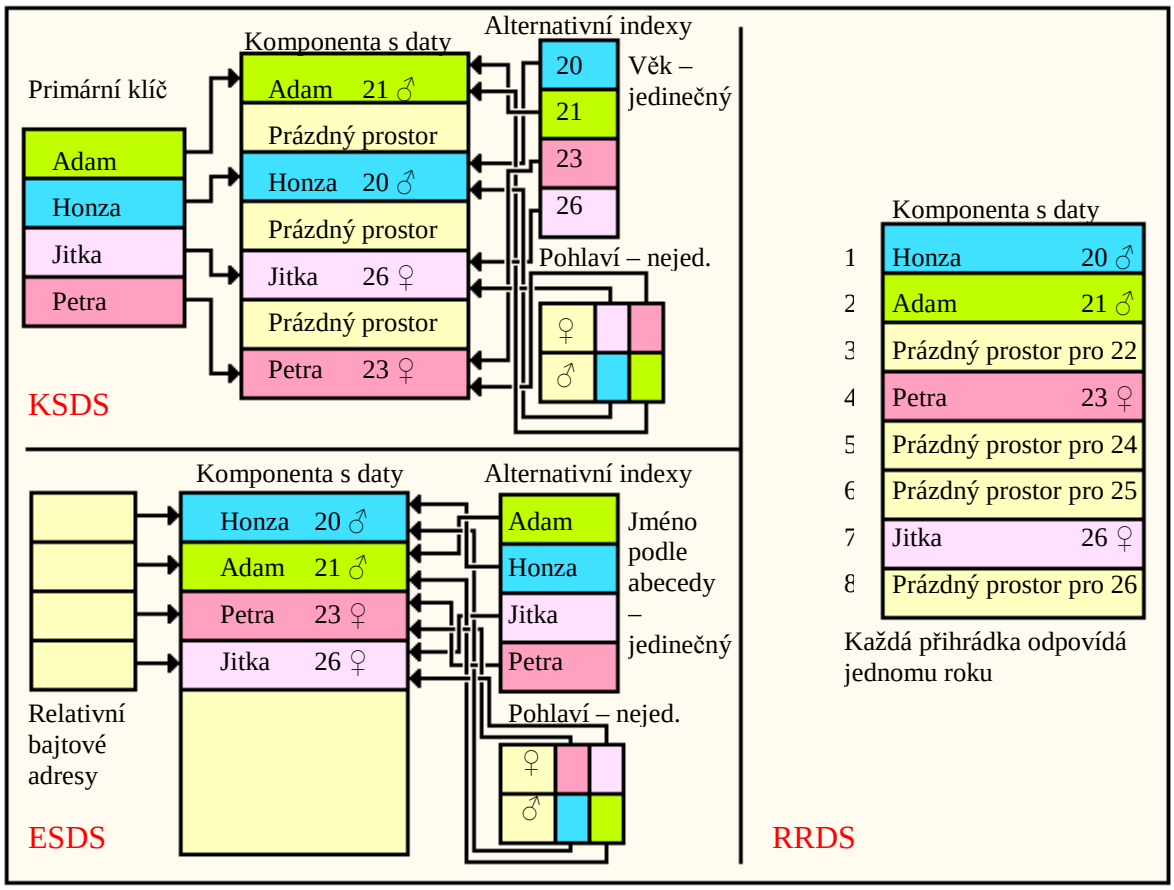
Nejpoužívanějšími data sety VSAM jsou KSDS soubory. K záznamům v těchto data setech přistupujeme pomocí jedinečného primárního klíče, kterým může být například uživatelské jméno uživatele počítačové učebny, nebo postupně, jak jsou uloženy za sebou. Jednotlivé záznamy mohou mít rozdílnou délku.

ESDS soubory se používají především pro data, ke kterým přistupujeme postupně v pořadí, ve kterém byla vytvořena. K datům lze též přistupovat náhodně pomocí přístupu prováděného přes relativní bajtové adresy (Relative Byte Address, RBA), které jsou přidělené jednotlivým záznamům.

Nejpodobnější zmiňovaným polím jsou RRDS soubory. Každý záznam tu má totiž přirozené číslo, pomocí kterého k němu přistupujeme. Stejně jako u předchozích data setů lze přistupovat k záznamům postupně.

LDS soubory obsahují bajtové řetězce dat. V převážné většině případů je používají pouze systémové funkce z/OSu. Na rozdíl od předchozích typů dokonce ani nejsou podporovány programovacím jazykem C/C++.

Jak první tři typy data setů vypadají, můžete přehledně vidět na obrázku 10. Zachycuje uložení záznamů o čtyřech lidech (jméno, věk a pohlaví). U typů KSDS a ESDS zobrazuje též přístupové alternativní indexy, o kterých budeme mluvit v příští podkapitole.



Obrázek 10: Typy data setů VSAM

2.5.3. Alternativní indexy

Soubory KSDS a ESDS nám dovolují kromě obvyklého přístupu také přístup pomocí tzv. alternativního indexu. Alternativní index je další klíč, pomocí kterého lze přistupovat k záznamům. Například je to znázorněné na obrázku 10. Pokud máme jako primární klíč pro KSDS jméno, alternativním indexem může být věk nebo pohlaví.

Také tu vidíme, že alternativních indexů existují dva typy – jedinečné a nejedinečné. Jaký je mezi nimi rozdíl, je nasnadě – zatímco jedinečný alternativní indexech ukazuje pouze na jeden záznam, nejedinečný může ukazovat na více záznamů.

Alternativní indexy se skladují v KSDS. Jsou tu uložené jako záznamy s různou délkou, které obsahují hodnotu klíče a ukazatel nebo ukazatele na záznamy v jiné KSDS nebo ESDS.

Stejně jako většinu věcí spojených s data sety VSAM také je vytváří IDCAMS.

2.5.4. Základní příkazy IDCAMS

Jak už jsme si řekli, IDCAMS je prostředek pro správu data setů VSAM. Volat ho můžeme pomocí vlastních programů nebo pomocí úlohy, napsané v JCL. O JCL pojednává kapitola 3., kterou je vhodné mít pro pochopení následujícího přečtenou.

Nejsnazší je spuštění IDCAMS úlohou. Uděláme to tak, že do parametru PGM příkazu EXEC zadáme IDCAMS. Následuje příkaz DD pro umístění výstupu. Pak už pomocí dalšího DD můžeme zadat vlastní příkazy IDCAMS. Jasně je to vidět na následujícím příkladu:

```
//STEP1  EXEC PGM=IDCAMS
//SYSPRINT  DD SYSOUT=*
//SYSIN  DD *
   PRIKAZY_IDCAMS
/*
```

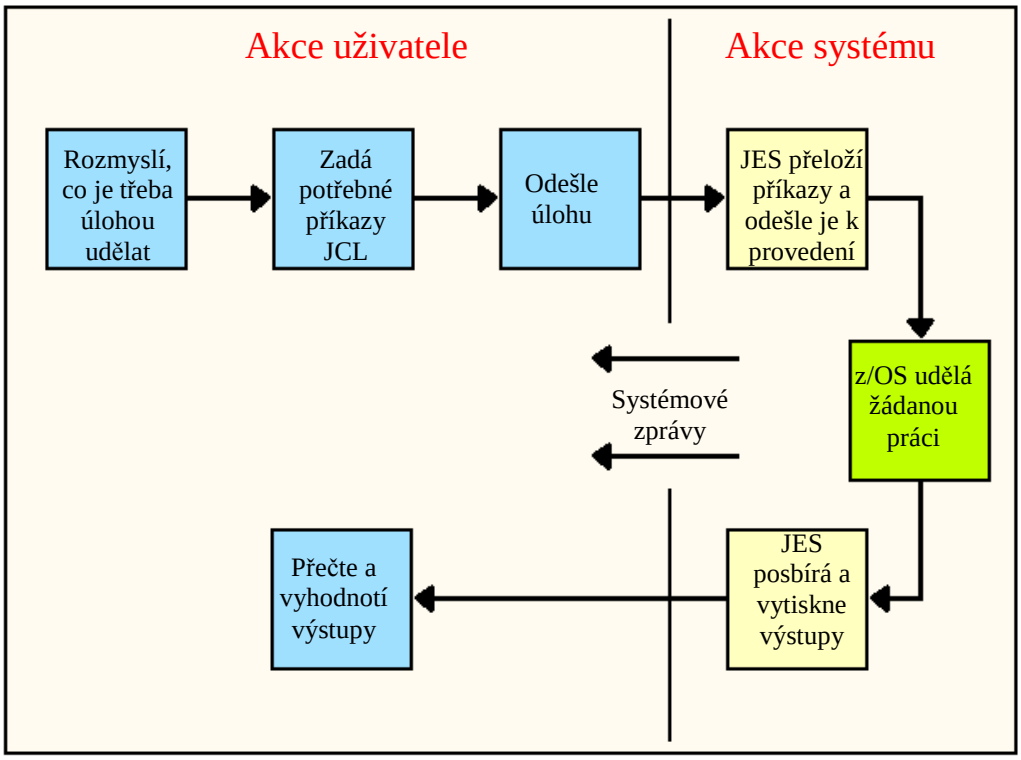
Mezi nejpoužívanější příkazy IDCAMS patří DELETE pro mazání záznamů, DEFINE CLUSTER pro tvorbu nového záznamu v existujícím data setu VSAM, REPRO pro uložení dat z jiného místa do záznamu, DEFINE AIX pro tvorbu alternativního indexu a DEFINE PATH pro vytvoření cesty. Každý z těchto příkazů má množství parametrů, které oddělujeme pomlčkou a pro přehlednost píšeme vždy na nový řádek. Jejich praktické použití můžete nalézt v příloze 1 v příkladu převzatém z literatury.

2.6. JES

2.6.1. Kontrola nad úlohami

V kapitole 1.3. jsme si řekli, že mainframy obvykle provádí dva typy prací. Prvními jsou dávkové úlohy a druhými transakce v reálném čase. V této a následujících kapitolách se blíže seznámíme s dávkovými úlohami nebo též jen úlohami.

Jak už víme, úlohy jsou typ práce, kdy na začátku mainframe dostane vstupní data, nějakým způsobem s nimi naloží a vytvoří použitelný výstup. Do tohoto procesu uživatel zasahuje jen minimálně nebo vůbec. Co má systém provádět, mu uživatel oznámí na začátku pomocí jazyka řízení úloh JCL, o kterém bude řeč v následujících kapitolách. Jakmile tyto příkazy odešle, ztrácí nad úlohou kontrolu a opět se s ním dostane do kontaktu až po jeho skončení, když mu operační systém vydá výstupy. Mezitím přebírá dohled část z/OSu jménem podsystém vstupu úloh (job entry subsystem, JES), o kterém si budeme v této kapitole povídat. JES ztrácí dozor nad úlohou pouze při provádění požadované práce. Přehledně to můžete vidět na obrázku 11:



Obrázek 11: Kontrola nad úlohou

2.6.2. Účel JESu, JES2 a JES3

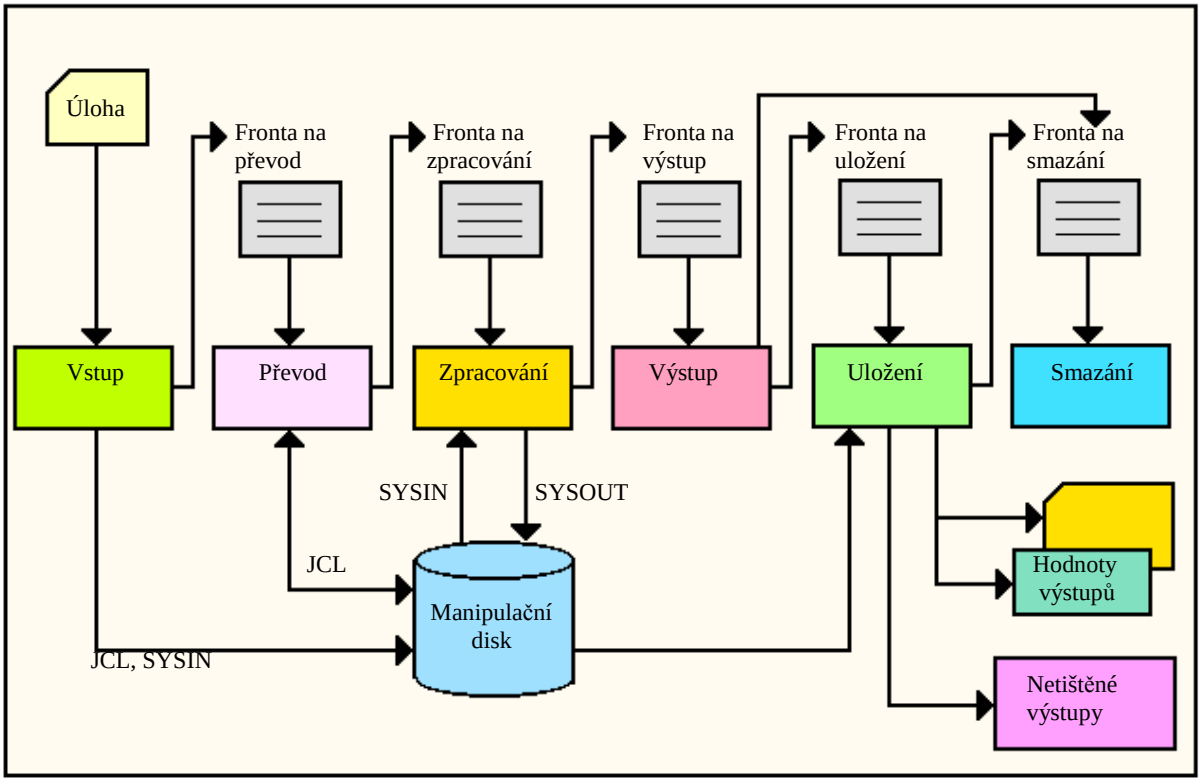
Jak je vidět, JES se o úlohu stará před a po průběhu programu. Jeho účelem je:

- přijetí úloh ke zpracování
- jejich převedení do strojově čitelné formy
- postupné odesílání úloh k vyhodnocení
- kontrola a zpracování výstupů
- odstranění výstupů ze systému

Existují dva typy JESu – JES2 a JES3. Pokud obsahuje mainframe pouze jeden procesor, fungují oba téměř stejně. JES3 navíc zvládá některé funkce, k jejichž provádění potřebuje JES2 jiné prostředky. Výrazné odlišnosti se projeví u systémů s více procesory. JES2 v tomto případě funguje pro každý procesor nezávisle na ostatních. Oproti tomu JES3 vytvoří na jednom procesoru tzv. globální kopii, která pak má kontrolu nad všemi ostatními, tzv. lokálními. Globální JES3 spravuje příjem všech úloh a posílá je ke zpracování. Tím pádem je JES3 efektivnější, nicméně stále je podstatně méně rozšířený než JES2.

2.6.3. Cesta úlohy systémem

Podívejme se nyní podrobněji na cestu úlohy systémem používajícím JES2. Během ní projde šesti fázemi – vstupem (input), převodem (conversion), zpracováním (processing), výstupem (output), uložením (hard-copy) a smazáním (purge). Průběh cesty úlohy je zobrazen na obrázku 12.



Obrázek 12: Cesta úlohy systémem

Úlohu do systému mohou posílat jak vstupní zařízení, tak různé programy. Ve vstupní fázi ji převezme JES2 a každému příkazu JCL přiřadí identifikátor. Potom kód JCL společně s hodnotami vstupních parametrů a svými kontrolními příkazy uloží do manipulačních data setů na manipulační disk. Manipulační disk je oblast fyzické paměti, ve které jsou uložená data ve speciálním formátu a jsou odtud rychle přístupná. Hodnoty vstupních parametrů se ukládají do data setu SYSIN, která je plně pod kontrolou JES2.

Při převodní fázi JES2 použije převaděč, který přidá do kódu příkazy z volaných knihoven, poté celý kód převede na strojově čitelný text a ten JES2 opět uloží na manipulační disk. Pokud JES2 objeví v příkazech JCL nějaké chyby, pošle úlohu do výstupní fronty, v opačném případě do fronty na zpracování.

Na začátku fáze zpracování JES2 předá úlohu tzv. iniciátoru. Iniciátor je systémový program, který nejprve kontroluje, zda provádění úloh nevede k špatnému použití uložených data setů (jedna úloha něco čte a jiná jí to maže) a zda je připraven potřebný hardware. Pokud je vše v pořádku, nalezne pro úlohu požadovaný prováděcí program. Iniciátorů je víc druhů, každý obsluhuje jinou skupinu úloh. Při zpracování dokáží komunikovat s JES2, které jim může posílat potřebné data sety a jiné věci. Na konci provádění úlohy systém JES2 upozorní, že si opět může úlohu převzít.

JES2 má plnou kontrolu nad data setem SYSOUT. Ten je uložen na manipulačním disku a ukládají se do něj veškeré výstupy po zpracování. JES2 je během výstupní fáze roztřídí na zprávy systému a data, která chtěl uživatel úlohou získat a pošle je na správná místa k vytištění či uložení.

Při fázi uložení zpracuje JES2 podle mnoha kritérií získané výstupy a poté pošle úlohu do fronty na smazání.

Během mazání úlohy uvolní JES2 veškerý prostor na manipulačním disku, který byl úloze přidělen, a pošle zprávu, že úloha byla ukončena.

3. JCL

3.1. Editor data setů

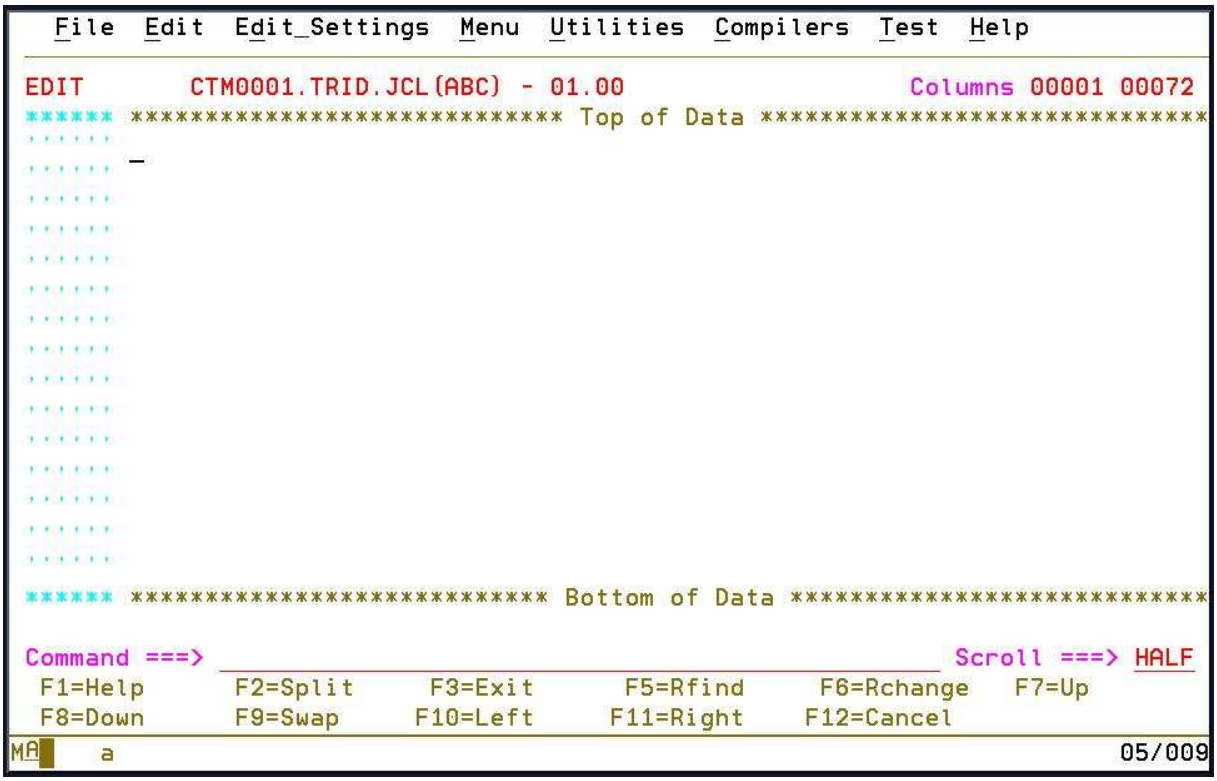
3.1.1. Úvod

V minulé kapitole jsme se seznámili s dávkovými úlohami teoreticky. Už víme, jak úloha putuje systémem a kdo se o ni kdy stará. Pojďme nyní vyzkoušet praktickou stránku věci. Na následujících několika stranách si přiblížíme kde vytvořit úlohu, jak ji vytvořit, jak ji odeslat systému ke zpracování a kde najít výstupy úlohy. V kapitole 3.4. pak naleznete konkrétní příklady fungujících úloh.

Když jsme si povídali o ISPF, zmínili jsme, že pod volbou 2 v menu základních voleb se nachází editor data setů. Jak název napovídá, můžeme s jeho pomocí upravovat existující data sety. To se hodí při mnoha příležitostech, například při vytváření nových úloh pomocí jazyka řízení úloh JCL. Než se do toho pustíme, seznámíme se blíže s ovládáním editoru data setů.

3.1.2. Spuštění a první pohled

Abychom mohli editor spustit, potřebujeme mít připravený existující data set. Jak se data sety vytvářejí, bylo popsáno v podkapitole 2.4.5. Název data setu zadáme na příslušná místa (jednotlivé kvalifikátory za slova Project, Group, Type a Member, pokud má název data setu více kvalifikátorů, vyplníme prostor za Data Set Name v části Other Partitioned, Sequential or VSAM Data Set) na vstupním panelu úprav. Poté stačí odeslat enterem a otevře se panel, který můžete vidět na obrázku 13.



Obrázek 13: Panel, kde lze upravovat data sety

Zcela nahoře nalezneme lištu s nástroji. Některé nástroje mají funkce příkazů editoru (viz dále), jiné zobrazují volby z různých menu ISPF. Pod nimi je napsané jméno

upravovaného data setu. V našem případě to je CTM0001.TRID.JCL(ABC), což znamená, že upravujeme člen ABC v data setu typu PDS. Vedle názvu máme napsáno, které sloupce se zobrazují, momentálně to jsou sloupce 1 až 72. Úplně dole se nachází přiřazení funkčních kláves (zejména ty pro posouvání textu se nám budou nyní opravdu hodit) a nad ním nápis Command = = = > pro zadávání příkazů editoru a Scroll = = = > pro množství posouvaného textu (úpravy viz 2.3.4). Uprostřed mezi nápisy ** Top of Data ** a ** Bottom of Data ** leží prostor pro zadávání našeho textu. V tuto chvíli je na začátku každého řádku šest teček. Pokud zmáčkneme enter, tečky zmizí a oba nápisy s hvězdičkami jsou těsně nad sebou.

3.1.3. Psaní textu, identifikace řádků

V editoru můžeme dělat tři základní úkony – psát text, zadávat řádkové příkazy a zadávat příkazy editoru.

Text lze psát do sloupců, které leží pod sedmou hvězdičkou a více vpravo, a do řádků, které mají na začátku čísla nebo tečky. Takovéto řádky získáme pomocí řádkového příkazu (viz dále). Každý řádek představuje jeden záznam v data setu a proto je jeho délka udána parametrem Record lenght při vytváření data setů. Typicky to bývá (a pro data sety definující úlohy přímo musí být) osmdesát znaků.

Každý popsaný řádek má vlastní identifikační číslo, které se zobrazuje na jeho začátku pod prvními šesti hvězdičkami. Číslování nových řádků probíhá po stovkách, první řádek má tedy číslo 000100, druhý 000200 atd. Pokud se rozhodneme mezi existující popsané řádky vložit nějaký nový, starým čísla zůstanou a nové řádky získají čísla s desítkami, tedy např. pokud vložíme dva řádky mezi šestý (číslo 000600) a sedmý (číslo 000700), budou mít nové řádky čísla 000610 a 000620. Obdobný proces s jednotkami proběhne, pokud se později rozhodnu mezi tyto řádky vložit další – získal by číslo 000611. Pokud tento systém nejde použít (mezi dva vkládám jedenáct nových nebo chci přidat řádek mezi dva s čísly končícími jednotkou) dochází k přečíslování starých řádků.

3.1.4. Řádkové příkazy

Řádkové příkazy zadáváme přepisováním prvních šesti hvězdiček na řádku s nápisem ** Top of Data ** a přepisováním teček, respektive čísel pod nimi. Přehled těch nejdůležitějších a toho, jak působí, si můžete prohlédnout v tabulce 3. Jako obvykle je odesíláme enterem.

Příkaz	**Popis**	
	Zapsání	**Výsledek**
I	Vloží jeden řádek	

	`I****** **** Top of Data ****`
	`000100 ABC`
	`000200 DEF`
	`****** *** Bottom of Data **`
		`****** **** Top of Data ****`
		`......`
		`000100 ABC`
		`000200 DEF`
		`****** *** Bottom of Data **`

Ixx	Vloží xx řádků	

	`****** **** Top of Data ****`
	`I30100 ABC`
	`000200 DEF`
	`****** *** Bottom of Data **`
		`****** **** Top of Data ****`
		`000100 ABC`
		`......`
		`......`
		`......`
		`000200 DEF`
		`****** *** Bottom of Data **`

Příkaz	Popis	
	Zapsání	**Výsledek**
D	Smaže jeden řádek	

	`****** **** Top of Data ****`
	`000100 ABC`
	`D00200 DEF`
	`****** *** Bottom of Data **`
		`****** **** Top of Data ****`
		`000100 ABC`
		`****** *** Bottom of Data **`

Dxx	Smaže xx řádků	
DD	Smaže blok řádků, uvozený mezi DD a DD	

	`****** **** Top of Data ****`
	`000100 ABC`
	`DD0200 DEF`
	`000300 GHI`
	`DD0400 JKL`
	`000500 MNO`
	`****** *** Bottom of Data **`
		`****** **** Top of Data ****`
		`000100 ABC`
		`000500 MNO`
		`****** *** Bottom of Data **`

R	Zopakuje řádek jednou	

	`****** **** Top of Data ****`
	`R00100 ABC`
	`000200 DEF`
	`****** *** Bottom of Data **`
		`****** **** Top of Data ****`
		`000100 ABC`
		`000110 ABC`
		`000200 DEF`
		`****** *** Bottom of Data **`

Rxx	Zopakuje řádek xx-krát	
RR	Zopakuje jednou blok řádků, uvozený mezi RR a RR	
C, A	Zkopíruje řádek, označený C za řádek označený A	

	`****** **** Top of Data ****`
	`C00100 ABC`
	`A00200 DEF`
	`000300 GHI`
	`****** *** Bottom of Data **`
		`****** **** Top of Data ****`
		`000100 ABC`
		`000200 DEF`
		`000210 ABC`
		`000300 GHI`
		`****** *** Bottom of Data **`

C, B	Zkopíruje řádek, označený C před řádek označený B	
Cxx, A či B	Zkopíruje xx řádků, počínaje označeným C za řádek označený A či před řádek označený B	
M, A	Přesune řádek, označený M za řádek označený A	
M, B	Přesune řádek, označený M před řádek označený B	
Mxx, A či B	Přesune xx řádků, počínaje označeným C za řádek označený A nebo před řádek označený B	

Tabulka 3: Řádkové příkazy editoru data setů

3.1.5. Příkazy editoru

Příkazy editoru zadáváme za nápis Command = = = >. Na rozdíl od řádkových příkazů mají vliv na celý data set. S jejich pomocí můžeme například hledat konkrétní řádek či řetězec znaků, ukládat data nebo spojovat či rozdělovat členy data setů PDS.

K nejdůležitějším patří příkaz SAVE. Slouží k uložení provedených změn v data setu. Ty se uloží také pokud editor opouštíme vyskakovací klávesou (F3), ovšem pouze tehdy, když máme v profilu nastaveno AUTOSAVE ON (obvykle to nastaveno je).

Zda tomu tak je, se lze přesvědčit přímo v editoru zadáním příkazu PROFILE. Po jeho napsání se na začátku data setu vypíší všechny parametry profilu. Schováme je příkazem RESET nebo jen RES.

Příkaz UNDO slouží k odvolání posledních změn data setu, ke kterým došlo před posledním odentrováním. Funguje pouze, pokud je v profilu nastaveno RECOVERY ON

(obvykle to nastaveno není). Silnější je příkaz CANCEL, který nám dovolí vrátit se k verzi data setu vzniklé posledním uložením pomocí příkazu SAVE nebo opuštěním editoru.

Pro odesílání úloh napsaných pomocí JCL se používá příkaz SUBMIT nebo jen SUB, který budeme již za chvilku potřebovat.

Příkazů je samozřejmě celá řada a jejich přehled lze nalézt v nápovědě editoru.

3.2. Příkazy jazyka řízení úloh JCL

3.2.1. Vlastnosti data setů s JCL

Teď už víme jak a kde psát úlohy. Otázkou, na kterou odpovídá tato kapitola, je, co do data setů psát, aby nějaká úloha vznikla.

Na začátek důležité upozornění. Data sety, do kterých chceme psát příkazy JCL, musí splňovat jisté podmínky. Vždy totiž musí mít formát záznamů pevný blokový (Record format FB) a velikost logického záznamu 80 (Record length 80). Obvykle jsou také typu PDS a mají v názvu na pozici kvalifikátoru nejnižší úrovně písmena JCL (např. se jmenují CTM0001.TRIDENI.JCL), to už ale není pravidlem.

3.2.2. Co je JCL

Jazyk řízení úloh (Job Control Language, JCL) představuje prostředek, jak systému říci, co chceme dělat. V každé úloze prostřednictvím JCL zadáváme systému požadavky, který program nebo programy si přejeme použít a popisujeme, kde nalézt a jak zpracovat jejich vstupy a výstupy. Děje se tak skrze příkazy pro kontrolu úloh a jejich argumenty, které dohromady tvoří JCL. Příkazů existuje mnoho, nicméně většina úloh si vystačí pouze s několika.

V rámci každé úlohy jsou příkazy uspořádány do kroků úlohy. Každý krok sestává z příkazů nutných pro běh jednoho programu. Úloha může těchto kroků zahrnovat nejvýše 255, vyšší množství vyvolává chybu.

Základní příkazy JCL jsou tři. Prvním je příkaz JOB. Každá úloha ho musí obsahovat právě jednou. Slouží k vyznačení začátku úlohy a odeslání důležitých informací bezpečnostního, administrativního a identifikačního rázu. Druhým je příkaz EXEC. Tímto příkazem se označuje začátek každého kroku úlohy a v jeho argumentech specifikujeme, který program si přejeme spustit. Každá úloha obsahuje stejný počet těchto příkazů jako má kroků, tedy minimálně jeden. Třetím je příkaz DD. Slouží k popisu vstupů a výstupů, použitých v konkrétních krocích úlohy a k práci s data sety.

Všechny tři si za chvilku charakterizujeme podrobněji. Než se do toho pustíme, rád bych ještě zmínil některá základní pravidla pro psaní úloh:

- Příkazy na každém řádku začínají dvěma lomítky //. Za nimi hned a bez mezery následuje identifikátor příkazu. Na to je třeba dát pozor, i mezera je tu nositelem informace.
- Řádek s komentářem začíná dvěma lomítky a hvězdičkou //*.
- Všechno píšeme pouze velkými písmeny.
- Pokud má příkaz více argumentů, oddělujeme je čárkami. Další argument může následovat ihned za čárkou (BEZ mezery) nebo na následujícím řádku, který pak musí mít strukturu: dvě lomítka, jedna až třináct mezer, argument.
- Konec úlohy značí dvě lomítka // na řádku a za nimi už nic.

3.2.3. Parametry příkazu JOB

Příkaz JOB má následující strukturu:

```
//CTM0001A JOB(UNIVER),'CTM0001',CLASS=A,REGION=4096K,
//           MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
```

Takto vypadá začátek každé úlohy. Nápis CTM0001A před příkazem JOB je jméno úlohy. Bývá zvykem pojmenovávat úlohy uživatelským jménem uživatele (a někdy je to nutné), který je vytvořil a nějakým písmenkem napsaným za něj. Následuje příkaz JOB a jeho parametry:

- (UNIVER) značí příslušnou bezpečnostní klasifikaci a identifikační informaci o úloze. U každého systému je jiná, uživatel si ji musí zjistit od informovanějších kolegů.
- ‘CTM0001‘ je uživatelské jméno tvůrce úlohy.
- CLASS=A specifikuje požadavky úlohy na systémové zdroje. Příslušnou hodnotu je opět třeba zjistit, zde je to A.
- REGION=4096K označuje paměť, která má být přidělena úloze. Pro běžné úlohy stačí uvedených 4096K, ale někdy je potřeba i víc.
- MSGLEVEL=(1,1) říká systému, že má napsat zadané příkazy JCL i na výstupu a přidat zprávy o přidělení paměti.
- MSGCLASS=H sděluje systému, co má dělat se zprávami, vznikajícími při zpracování úlohy. Hodnota H znamená, že je má schovat pro pozdější zobrazení, hodnota T by říkala, že se mají objevit v pozastavené výstupní frontě (viz 3.3.3).
- NOTIFY=&SYSUID oznamuje, kam má systém poslat informaci, že úloha skončila. Hodnota &SYSUID znamená, že na tohle místo má být vloženo vaše uživatelské jméno a tedy, že zpráva má být poslána vám.

Zadané hodnoty parametrů v tomto příkladu jsou vyzkoušené praxí a měly by fungovat. Záleží ovšem na konkrétní instalaci. Existují i další parametry, které jsem tu nezmínil, ale ty nejsou pro zdárné spuštění jednoduchého úlohy potřeba.

3.2.4. Parametry příkazu EXEC

Příkaz EXEC má strukturu stejnou jako příkaz JOB. Po dvou lomítkách tady následuje jméno kroku úlohy, pro které už žádné konvence nejsou. Dále mezera, vlastní příkaz EXEC a jeho parametry.

Parametrů má příkaz EXEC méně než JOB, obvykle pouze jeden, kterým je PGM=. Jeho hodnotou je název spouštěného programu. Druhá možnost je parametr PROC=, do jehož hodnoty zadáme jméno procedury, kterou si přejeme spustit. O procedurách se zmíníme později. Pokud potřebuje program nebo procedura nějaká speciální data, zadávají se za jejich jméno, jak můžeme vidět na následujícím příkladu:

```
//STEP1 EXEC PROC=TRIDENI,VSTUP=CTM0001.TRID2.JCL(DATA3),
//        VYSTUP=CTM0001.TRID2.JCL(VYSTUP)
```

Dalšími možnými parametry jsou například TIME=, který zadává časový limit pro zpracování, nebo COND= pro podmíněné zpracování.

3.2.5. Parametry příkazu DD

Struktura příkazu DD se opět neliší od předchozích dvou, po dvou lomítkách, názvu příslušného data setu, mezeře a DD následují parametry. Příkaz DD má možných parametrů velké množství. Přehled několika nejdůležitějších je v následující tabulce 4.

Parametr	Význam
DSN=jméno_data_setu, DSNAME=jméno_data_setu	Slouží k identifikaci používaného data setu
DISP=(status, normální konec, nesprávný konec), DISP=(status, normální konec), DISP=status kde • status může mít hodnoty: o NEW – data set ještě neexistuje a bude se vytvářet, úloha má do něj přístup jako jediná o OLD – data set existuje a úloha má do něj přístup jako jediná o SHR – data set existuje a přistupovat do něj může naráz více úloh o MOD – pokud data set existuje, úloha má do něj přístup jako jediný; když do něj zapisuje výstupní data, přidávají se na konec; pokud data set neexistuje, může být vytvořen • normální a nesprávný konec mohou mít hodnoty: o DELETE – data set se smaže po konci příslušného kroku úlohy o KEEP, UNCATLG – data set se ponechá po konci příslušného kroku úlohy (neukládá se do katalogu) o CATLG – data set se ponechá po konci příslušného kroku úlohy (ukládá se do katalogu) o PASS – co se bude s data setem dělat se upřesní až v dalším kroku úlohy	Popis vlastností data setu na začátku běhu úlohy, při normálním ukončení úlohy a při nesprávném ukončení úlohy
* následovaná textovým řetězcem ukončeným /* na zvláštním řádku	Textový řetězec představuje vstupní data
DLM=@@ následované textovým řetězcem ukončeným @@ na zvláštním řádku	Textový řetězec představuje vstupní data
SYSOUT=*, SYSOUT= jméno_data_setu	Kam se umisťuje výstup úlohy
SPACE=(co,(prvně,přidat,kolik bloků)), SPACE=(co,(prvně,přidat)), SPACE=(co, prvně), kde • „co“ značí jednotky paměti (TRK, CYL, REC, KB, MB, jako řádek Space units na panelu tvorby nového data setu ISPF, podkapitola 2.4.5.) • „prvně“ značí velikost data setu na začátku (jako řádek Primary quantity) • „přidat“ značí kolik se má přidávat, když velikost nestačí (jako řádek Secondary quantity) • „kolik bloků“ značí velikost oblasti pro popis data setů PDS (jako řádek Directory blocks)	Kolik paměti požadujeme při tvorbě nového data setu

DCB=(LRECL=číslo,BLKSIZE=číslo,RECFM=formát, DSORG=typ)	Stejný význam podparametrů jako řádků Record lenght, Block size, Record format a Data set name type na panelu tvorby nového data setu ISPF, podkapitola 2.4.5.

Tabulka 4: Parametry příkazu DD

Parametrů je samozřejmě ještě mnohem víc (např. pro všechny ostatní řádky na panelu tvorby nového data setu ISPF) , toto jsou pouze ty nejdůležitější.

3.2.6. Další příkazy JCL – knihovny a procedury

Kromě základních tří příkazů má JCL ještě speciální příkazy pro otevírání knihoven a pro vytváření procedur.

Knihovny můžeme otevírat na dvou místech. První možností je hned za příkazem JOB. Zde lze zadat příkaz JCLLIB ve tvaru:

```
//MYLIB JCLLIB ORDER=('jméno_knihovny')
```

Do argumentu můžeme zadat i více jmen knihoven, oddělených čárkami. Procedury z takto otevřených knihoven budou k dispozici pro celou úlohu. Pokud místo tohoto příkazu použijeme příkaz JOBLIB DD[1], můžeme pro potřeby úlohy spustit nějaký program. Příkaz STEPLIB DD[1] umístěný za příkaz EXEC spustí program pouze pro daný krok úlohy.

Procedur rozeznáváme dva typy. Prvními jsou tzv. včleněné procedury (In-stream procedures). Jak již jejich název napovídá, jsou umístěné přímo v úloze. Jedná se o kolekci příkazů, které se v dané úloze často opakují. Pro přehlednost je proto můžeme umístit do nějaké takovéto včleněné procedury. Struktura včleněné procedury je následující:

```
//jméno PROC
//krok EXEC …
…
//    PEND
```

Takto napsanou proceduru si můžeme připravit na začátek úlohy. Poté, až ji budeme potřebovat, zadáme pouze do parametru PROC od příkazu EXEC její jméno. V rámci jedné úlohy můžeme vytvořit nejvýše patnáct včleněných procedur.

Druhým typem procedur jsou procedury katalogizované. Na rozdíl od včleněných se nacházejí v jiných data setech (knihovnách) než úloha a proto je může používat více různých úloh. Data set s katalogizovanými procedurami obsahuje pouze procedury se strukturou, jakou mají včleněné procedury, tedy žádný příkaz JOB. Pokud chceme nějakou použít, nejprve musíme otevřít příslušnou knihovnu a poté už používáme katalogizovanou proceduru stejně jako včleněnou. Do jednoho členu knihovny lze uložit pouze jednu proceduru a bývá zvykem pojmenovat člen a proceduru stejně.

3.2.7. Odesílání úloh

Jakmile máme naši úlohu napsanou, můžeme ji odeslat ke zpracování. Možnosti, jak to udělat, jsou tři. První a nejjednodušší je napsání příkazu SUB nebo SUBMIT přímo v editoru data setů za nápis Command = = = > a jeho odeslání enterem. Druhá možnost je napsat v původním režimu TSO příkaz SUBMIT 'jméno_data_setu_s_úlohou' a odeslat

[1] Přesněji řečeno se nejedná o příkaz JCL, ale o vyhrazené jméno pro příkaz DD.

enterem a třetí je napsat tento příkaz na příkazovou řádku TSO. Po odeslání libovolného z nich se objeví hlášení JOB jméno_úlohy(číslo úlohy) SUBMITTED, po čase následované zprávou o úspěšném ukončení úlohy (job successful) nebo chybě JCL či nesprávném ukončení programu. Co přesně se stalo, můžeme zjistit pomocí nástroje SDSF, o kterém pojednává následující kapitola.

3.3. SDSF

3.3.1. Co je SDSF

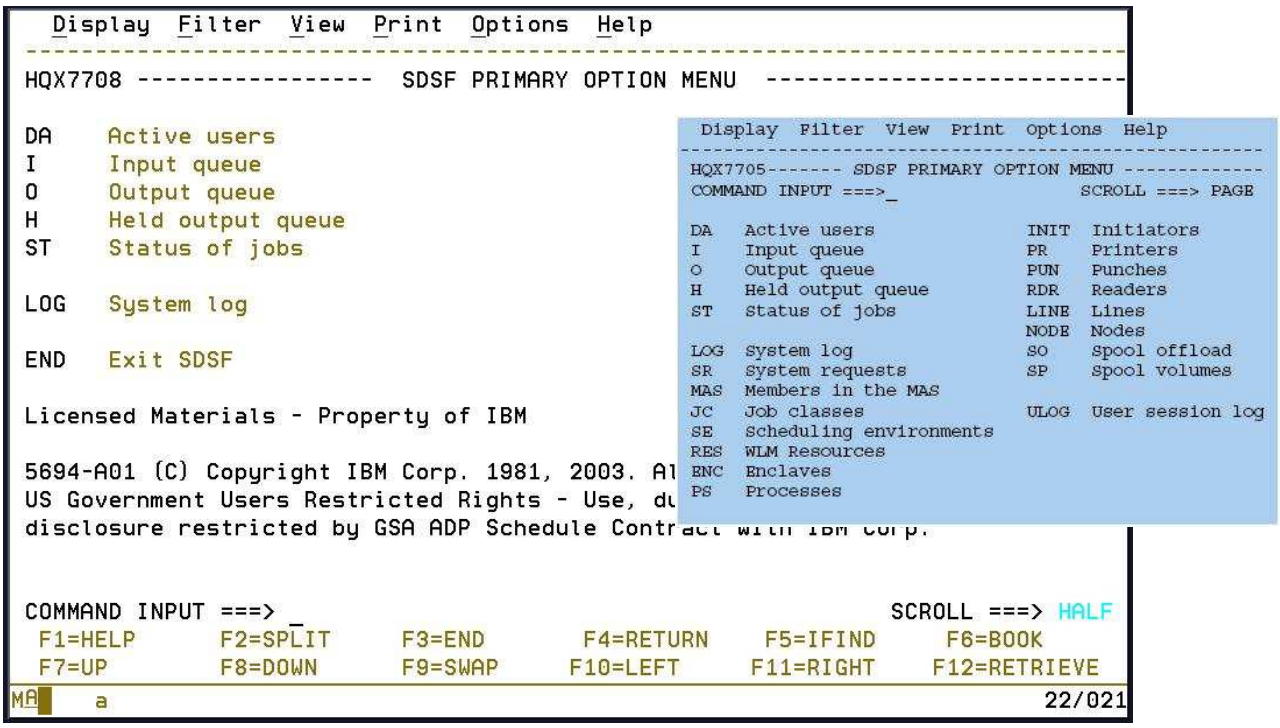
SDSF, neboli Prostředek pro zobrazování a prohledávání manipulačního prostoru (Spool Display and Search Facility), může být velmi mocným nástrojem při práci se z/OS. Nejen, že jeho prostřednictvím můžeme sledovat a kontrolovat všechny úlohy procházející systémem včetně jejich výstupů a front, ale navíc lze skrze SDSF zadávat také příkazy JES2 a celému systému (pokud máme příslušná práva). SDSF zvládá i dohled na všechny tiskárny systému a spoustu dalších věcí.

SDSF bývá nainstalováno pouze na systémech, využívajících JES2. Pokud se používá JES3, pro zobrazování výstupů úloh slouží nástroj (E)JES.

Umístění příkazu pro spuštění SDSF na panelech ISPF se výrazně liší v každé instalaci a uživatel ho musí chvíli hledat. Druhou možností, jak začít práci s tímto nástrojem, je příkaz SDSF zadaný v původním režimu TSO.

3.3.2. Panely a nabídky SDSF

Po spuštění SDSF se jako první objeví panel menu základních voleb SDSF. Co se na panelu nalézá, se výrazně liší jednak podle instalace a jednak podle vašich přístupových práv. Na hlavní části obrázku 14 můžete vidět obsah menu na instalaci, se kterou jsem pracoval já. Tato verze SDSF je velmi okleštěná a využít se dá pouze ke sledování úloh a jejich výstupů. Na jiných instalacích mohou být varianty s daleko větším množstvím voleb pro správu systému, tiskáren a ostatních zdrojů. Příklad můžete vidět na obrázku 14 ve výřezu.



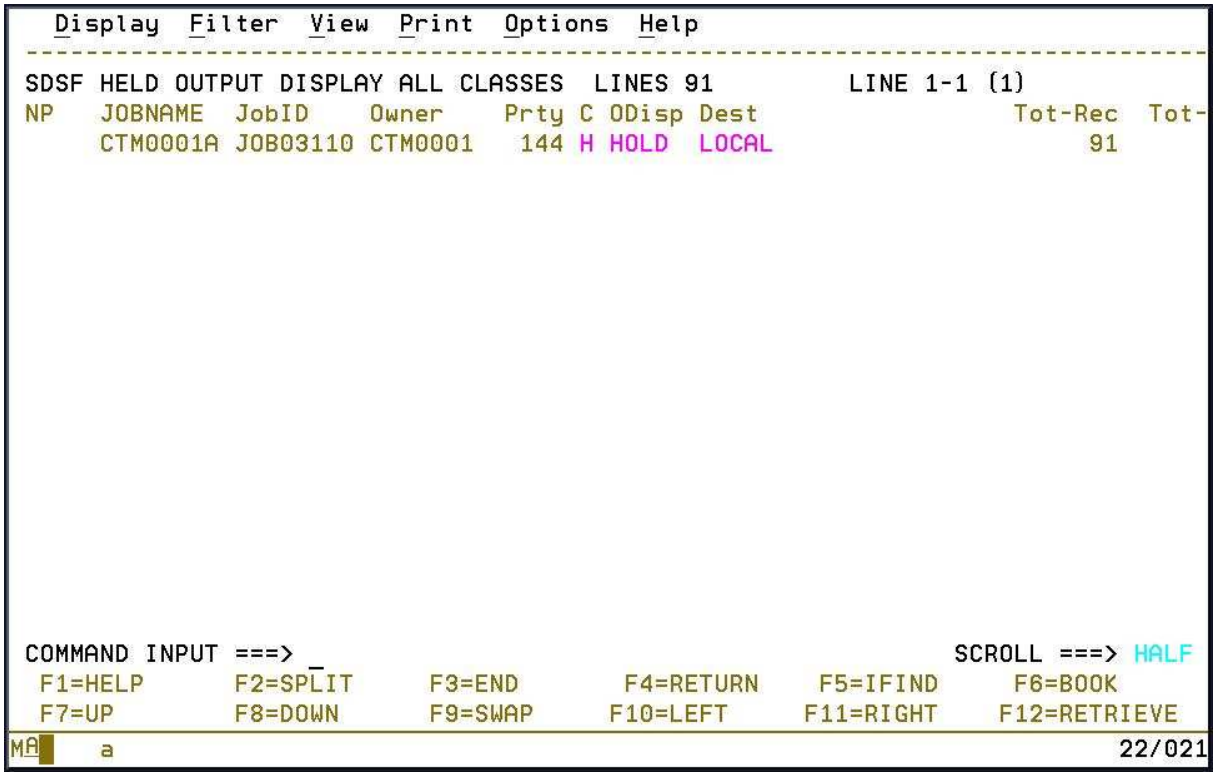
Obrázek 14: Menu základních voleb SDSF

Popišme si nyní pořádně menu základních voleb SDSF. Zcela nahoře je lišta s nástroji, která se nemění ani na dalších panelech. V nabídce Display můžeme zvolit, kterou frontu úloh chceme vidět (stejně jako příkazem vybraným z centrální části menu a napsaným za Command Input = = =>). O jednotlivých frontách si budeme povídat za chvilku. Nabídka Filter umožňuje na panelech se seznamy zobrazovat pouze některé položky (např. pouze úlohy, jejichž název začíná nějakým prefixem), pomocí volby View můžeme rozhodnout, jak se budou položky řadit. Volba Print slouží k tištění výstupů. Jelikož jsem asi pro něj neměl dostatečná oprávnění, pokus o zvolení libovolné položky této nabídky vedl k zamrznutí terminálu. Nabídka Options přináší mnoho různých pokročilejších funkcí. Hodí se funkce číslo 5 (Set display ON/OFF), díky které se na obrazovce objeví aktuální hodnoty filtrů a řazení položek seznamů (funguje stejně, jako příkaz SET DISPLAY ON či OFF napsaný za Command Input = = =>). Nabídka Help skrývá překvapivě nápovědu.

Ostatní části panelu už známe od jinud. Za zmínku snad ještě stojí, že za nápis Command Input = = => mohou lidé s dostatečným oprávněním zadávat systémové příkazy a příkazy pro JES2.

Nyní přejdeme na další panely. Zadáním příkazu DA se ocitneme na panelu, informujícím o současné zátěži systému. Jsou zde vypsané všechny požadavky na systém, uživatelé systému a úlohy a informace o nich (např. využití procesoru). Když panel otevřeme pomocí příkazu DA OTSU, zobrazí se pouze uživatelé, skrze DA OJOB se dostaneme pouze k seznamu právě probíhajících úloh a že chceme vidět pouze požadavky vyjádříme příkazem DA OSTC.

Dalšími příkazy jsou I, O a H pro vypsání úloh, požadavků a uživatelů, nalézajících se momentálně ve frontě na zpracování nebo které jsou zpracovávány (příkaz I jako input), respektive těch ve frontě na výstup (příkaz O jako output), respektive v pozastavené frontě na výstup (příkaz H jako held). Všechny tři panely mají podobnou strukturu, opět je to seznam úloh s mnoha atributy. Jak vypadá panel s úlohami z pozastavené výstupní fronty můžete vidět na obrázku 15.



Obrázek 15: Pozastavená výstupní fronta

Pomocí příkazu ST získáme seznam všech úloh, požadavků a uživatelů systému, které jsou v libovolné frontě JES2. Další příkazy tu rozebírat nebudu, neboť jsem neměl možnost se s nimi lépe seznámit.

3.3.3. Výstup úloh

Nyní už se dokážeme v SDSF trochu zorientovat a proto je vhodná doba podívat se na výstup úlohy, kterou jsme před časem odeslali do systému ke zpracování a informace o jejím provádění. Zda a kde je nalezneme, jsme sami určili hodnotou parametru MSGCLASS u příkazu JOB. Pokud jsme zadali například MSGCLASS=H, nalezneme obvykle naši úlohu v pozastavené výstupní frontě (když se jmenuje, jak má, tedy s uživatelským jménem na začátku) nebo v seznamu na panelu ST (ve všech případech). Pro její identifikaci si potřebujeme pamatovat jeho číslo, které nám systém sdělil po odeslání úlohy.

Podle toho, jaké údaje nás zajímají, máme dvě možnosti, co udělat. Pokud chceme vidět celý výstup, napíšeme vlevo na řádku s naší úlohou písmeno S, když nás zajímá jen část, napíšeme tam otazník. To se hodí, když má naše úloha více kroků a my chceme vidět informace pouze o jednom. Po napsání otazníku se totiž objeví panel s nabídkou přístupu k jednotlivým sekcím zpráv, uložených v různých data setech. Data set JESMSGLG obsahuje systémové zprávy, v JESJCL nalezneme příkazy JCL, do kterých jsou dosazené všechny procedury z knihoven a JESYSMSG zahrnuje další systémové zprávy. Pak následují další data sety se zprávami od programů z jednotlivých kroků a s jejich výstupními daty. Libovolný data set zobrazíme napsáním písmene S vlevo od jeho jména.

Na následujících řádcích můžete vidět, jak vypadá příklad úplného výstupu od jednoduché úlohy:

```
****************************************** TOP OF DATA ************************************************************
1                J E S 2  J O B  L O G  --  S Y S T E M  M V S 1  --  N O D E  T S T M V S 0 1
0
 18.34.37 JOB03121 ---- TUESDAY,   14 MAR 2006 ----
 18.34.37 JOB03121  IRR010I  USERID CTM0001  IS ASSIGNED TO THIS JOB.
 18.34.37 JOB03121  ICH70001I CTM0001  LAST ACCESS AT 18:33:08 ON TUESDAY, MARCH 14, 2006
 18.34.37 JOB03121  $HASP373 CTM0001A STARTED - INIT 1    - CLASS A - SYS MVS1
 18.34.37 JOB03121  IEF403I CTM0001A - STARTED - TIME=18.34.37
 18.34.37 JOB03121  -                                  --TIMINGS (MINS.)--            ----PAGING COUNTS---
 18.34.37 JOB03121  -JOBNAME  STEPNAME PROCSTEP    RC   EXCP    CPU    SRB  CLOCK   SERV  PG   PAGE   SWAP    VIO SWAPS
 18.34.37 JOB03121  -CTM0001A          STEP1      00     44    .00    .00    .00    173   0      0      0      0     0
 18.34.37 JOB03121  IEF404I CTM0001A - ENDED - TIME=18.34.37
 18.34.37 JOB03121  -CTM0001A ENDED.  NAME-CTM0001             TOTAL CPU TIME=   .00  TOTAL ELAPSED TIME=   .00
 18.34.37 JOB03121  $HASP395 CTM0001A ENDED
0------ JES2 JOB STATISTICS ------
-  14 MAR 2006 JOB EXECUTION DATE
-           18 CARDS READ
-           83 SYSOUT PRINT RECORDS
-            0 SYSOUT PUNCH RECORDS
-            6 SYSOUT SPOOL KBYTES
-         0.00 MINUTES EXECUTION TIME
         1 //CTM0001A  JOB (UNIVER),'CTM0001',CLASS=A,REGION=4096K,                JOB03121
           //         MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID                     00020000
           IEFC653I SUBSTITUTION JCL - (UNIVER),'CTM0001',CLASS=A,REGION=4096K,MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=CTM0001
         2 //STEP1   EXEC PGM=SORT                                               00030009
         3 //SYSIN   DD *                                                        00031009
         4 //SYSOUT  DD SYSOUT=*                                                 00034009
         5 //SORTIN  DD *                                                        00040002
         6 //SORTOUT  DD SYSOUT=*                                               00150000
           /*                                                                   00160006
 ICH70001I CTM0001  LAST ACCESS AT 18:33:08 ON TUESDAY, MARCH 14, 2006
 IEF236I ALLOC. FOR CTM0001A STEP1
 IEF237I JES2 ALLOCATED TO SYSIN
 IEF237I JES2 ALLOCATED TO SYSOUT
 IEF237I JES2 ALLOCATED TO SORTIN
 IEF237I JES2 ALLOCATED TO SORTOUT
IEF142I CTM0001A STEP1 - STEP WAS EXECUTED - COND CODE 0000
 IEF285I   CTM0001.CTM0001A.JOB03121.D0000101.?         SYSIN
 IEF285I   CTM0001.CTM0001A.JOB03121.D0000103.?         SYSOUT
 IEF285I   CTM0001.CTM0001A.JOB03121.D0000102.?         SYSIN
 IEF285I   CTM0001.CTM0001A.JOB03121.D0000104.?         SYSOUT
 IEF373I STEP/STEP1   /START 2006073.1834
 IEF374I STEP/STEP1   /STOP  2006073.1834 CPU    0MIN 00.01SEC SRB    0MIN 00.00SEC VIRT  1068K SYS   332K EXT    6148K SYS   10632K
 IEF375I  JOB/CTM0001A/START 2006073.1834
 IEF376I  JOB/CTM0001A/STOP  2006073.1834 CPU    0MIN 00.01SEC SRB    0MIN 00.00SEC
1ICE143I 0 BLOCKSET     SORT  TECHNIQUE SELECTED
 ICE250I 0 VISIT http://www.ibm.com/storage/dfsort FOR DFSORT PAPERS, EXAMPLES AND MORE
 ICE000I 1 - CONTROL STATEMENTS FOR 5694-A01, Z/OS DFSORT V1R5 - 18:34 ON TUE MAR 14, 2006 -
0             SORT    FIELDS=(1,75,CH,A)                                        00032009
 ICE201I 0 RECORD TYPE IS F - DATA STARTS IN POSITION 1
 ICE751I 0 C5-BASE   C6-BASE   C7-BASE   C8-Q83041 E4-BASE   C9-BASE   E5-Q90312 E7-Q91626
 ICE193I 0 ICEAM1 ENVIRONMENT IN EFFECT - ICEAM1 INSTALLATION MODULE SELECTED
 ICE088I 1 CTM0001A.STEP1   .        , INPUT LRECL = 80, BLKSIZE = 80, TYPE = FB
 ICE093I 0 MAIN STORAGE = (MAX,6291456,6278238)
 ICE156I 0 MAIN STORAGE ABOVE 16MB = (6217712,6217712)
 ICE127I 0 OPTIONS: OVFLO=RC0 ,PAD=RC0 ,TRUNC=RC0 ,SPANINC=RC16,VLSCMP=N,SZERO=Y,RESET=Y,VSAMEMT=Y,DYNSPC=256
 ICE128I 0 OPTIONS: SIZE=6291456,MAXLIM=1048576,MINLIM=450560,EQUALS=N,LIST=Y,ERET=RC16 ,MSGDDN=SYSOUT
 ICE129I 0 OPTIONS: VIO=N,RESDNT=ALL ,SMF=NO   ,WRKSEC=Y,OUTSEC=Y,VERIFY=N,CHALT=N,DYNALOC=N             ,ABCODE=MSG
 ICE130I 0 OPTIONS: RESALL=4096,RESINV=0,SVC=109 ,CHECK=Y,WRKREL=Y,OUTREL=Y,CKPT=N,STIMER=Y,COBEXIT=COB2
 ICE131I 0 OPTIONS: TMAXLIM=6291456,ARESALL=0,ARESINV=0,OVERRGN=65536,CINV=Y,CFW=Y,DSA=0
 ICE132I 0 OPTIONS: VLSHRT=N,ZDPRINT=Y,IEXIT=N,TEXIT=N,LISTX=N,EFS=NONE    ,EXITCK=S,PARMDDN=DFSPARM ,FSZEST=N
 ICE133I 0 OPTIONS: HIPRMAX=OPTIMAL,DSPSIZE=MAX ,ODMAXBF=0,SOLRF=Y,VLLONG=N,VSAMIO=N,MOSIZE=MAX
 ICE235I 0 OPTIONS: NULLOUT=RC0
 ICE084I 0 BSAM ACCESS METHOD USED FOR SORTOUT
 ICE084I 0 BSAM ACCESS METHOD USED FOR SORTIN
```

```
ICE750I 0 DC 500000 TC 0 CS DSVVV KSZ 75 VSZ 75
ICE752I 0 FSZ=6250 RC  IGN=0 E  AVG=80 0  WSP=650 C  DYN=0 0
ICE751I 1 DE-Q82816 D5-Q84357 D9-Q91626 E8-Q91626
ICE090I 0 OUTPUT LRECL = 80, BLKSIZE = 80, TYPE = FB
ICE080I 0 IN MAIN STORAGE SORT
ICE055I 0 INSERT 0, DELETE 0
ICE054I 0 RECORDS - IN: 7, OUT: 7
ICE134I 0 NUMBER OF BYTES SORTED: 560
ICE199I 0 MEMORY OBJECT STORAGE USED = 0M BYTES
ICE180I 0 HIPERSPACE STORAGE USED = 0K BYTES
ICE188I 0 DATA SPACE STORAGE USED = 0K BYTES
ICE052I 0 END OF DFSORT
JUPITER                                                         00110000
MARS                                                            00100000
NEPTUN                                                          00050000
PLUTO                                                           00060000
SATURN                                                          00120000
URAN                                                            00130000
ZEME                                                            00070000
******************************** BOTTOM OF DATA ********************************************************
```

V části 1 jsou systémové informace, které se liší u jednotlivých instalací. Část 2 je tvořena zprávami JESu o úloze, v části 3 následuje kód jazyka JCL, kterým jsme úlohu popisovali. Část 4 zahrnuje systémové zprávy, vzniklé v průběhu provádění úlohy. Velmi důležité jsou řádky, obsahující text COND CODE 0000 (červeně zvýrazněno). Pokud je hodnota 0000, proběhl daný krok úlohy v pořádku, jiná hodnota značí chybu. V části 5 nalezneme výstupní data námi požadovaného programu.

3.4. Příklady fungujících úloh

3.4.1. Jednoduchý příklad

Na závěr kapitoly o JCL si ještě ukážeme několik příkladů konkrétních úloh. Jména používaných data setů vždy začínají mým uživatelským jménem CTM0001 a po jeho nahrazení vaším by měly úlohy fungovat. Všechny úlohy jsou modifikacemi následujícího jednoduchého příkladu:

```
EDIT       CTM0001.TRID2.JCL(PRIKAZY) - 01.10
****** ***************************** Top of Data ************************
000100 //CTM0001A   JOB (UNIVER),'CTM0001',CLASS=A,REGION=4096K,
000200 //           MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
000300 //STEP1  EXEC PGM=SORT
000400 //SYSIN  DD *
000500   SORT  FIELDS=(1,75,CH,A)
000510 /*
000600 //SYSOUT DD SYSOUT=*
000700 //SORTIN DD *
000800 PLUTO
000900 MARS
001000 MERKUR
001100 ZEME
001200 URAN
001300 VENUSE
001400 JUPITER
001500 SATURN
001600 NEPTUN
001700 /*
001800 //SORTOUT  DD SYSOUT=*
****** **************************** Bottom of Data **********************
```

Celá úloha se nalézá v jednom data setu. Jejím posláním je setřídit podle abecedy názvy planet, zadané na řádcích 000800 až 0001600. Na prvních dvou řádcích zadáváme povinné parametry příkazu JOB. Na třetím řádku začíná jediný krok této úlohy, který se jmenuje STEP1 a využívá program SORT. Na následujícím řádku zadáváme do data setu SYSIN parametry, jak si třídění představujeme. V našem případě říkáme, že chceme třídit

sloupce 1 až 75 podle napsaných znaků (CH) vzestupně (A). Řádek 000600 udává, kam se má odesílat systémový výstup. Hvězdička znamená, že má zůstat k dispozici tam, kde určuje parametr MSGCLASS příkazu JOB a nemá se ukládat do žádného zvláštního data setu. Následuje načtení tříděných dat a konečně specifikace toho, kam uložit výstup třídícího programu, tedy seřazené planety. Hvězdička znamená, že výstup zůstane opět k dispozici v místě určeném parametrem MSGCLASS příkazu JOB.

3.4.2. Přesměrování vstupních a výstupních dat

Nyní příklad změníme. Vstup bude prováděn z existujících dat mimo data set s úlohou. Ukážeme si, že data můžeme získávat i z více míst najednou. Část tříděných dat uložíme do členu data setu, který se jmenuje CTM0001.TRID2.JCL(DATA), část do členu DATA2 stejného data setu. Ukládáme je tam bez jakýchkoli formalismů, stejně jako na řádky 000800 až 001600 v minulém příkladu. Upravený příklad potom bude vypadat takto:

```
EDIT       CTM0001.TRID2.JCL(PRIKAZY) - 01.11
****** ***************************** Top of Data ******************
000100 //CTM0001A   JOB (UNIVER),'CTM0001',CLASS=A,REGION=4096K,
000200 //           MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
000300 //STEP1  EXEC PGM=SORT
000400 //SYSIN  DD *
000500   SORT  FIELDS=(1,75,CH,A)
000510 /*
000600 //SYSOUT DD SYSOUT=*
000700 //SORTIN DD DSN=CTM0001.TRID2.JCL(DATA),DISP=OLD
000800 //       DD DSN=CTM0001.TRID2.JCL(DATA2),DISP=OLD
001800 //SORTOUT  DD SYSOUT=*
****** **************************** Bottom of Data ***************
```

Všimněme si řádků 000700 a 000800. Parametr DISP je tady použit ve své nejjednodušší podobě, kdy pouze specifikuje vlastnosti data setu, ale po ukončení běhu úlohy s ním už nijak nezachází.

Podobně můžeme přesměrovat i výstup, např. nahrazením řádku 001800 takovýmto:

```
001800 //SORTOUT  DD DSN=CTM0001.TRID2.JCL(VYSTUP),DISP=OLD
```

Při takovémto nahrazení musí člen data setu VYSTUP na začátku běhu úlohy už existovat, jinak vznikne chyba. Při jeho vytváření je dobré mít v něm uložený nějaký text (který se pak přepíše výstupem úlohy), protože pokud ho necháme prázdný, může se stát, že ho systém v rámci optimalizace po zavření smaže.

Může vás také napadnout zkusit přesměrovat systémové výstupy na řádku 000600 do nějakého data setu. Nedělejte to! Když jsem to zkoušel, povedlo se mi docílit toho, že nešlo otevřít nejen ten data set, kam jsem data posílal, ale ani data set s úlohou, v níž byl příkaz k přesměrování. Systém pouze dokola hlásil „I/O READING ERROR“ a já musel přepsat úlohu zpaměti do jiného data setu. Způsobeno to mohlo být tím, že výstup nerespektoval strukturu data setu PDS a zapsal se nejen do „svého“ členu, ale i přes příkazy (formát bloků výstupu nedefinovaný, viz podkapitola 2.4.4.).

3.4.3. Včleněná procedura

Další modifikací je přepsání části úlohy do včleněné procedury TRIDENI:

```
EDIT       CTM0001.TRID2.JCL(PRIKAZY2) - 01.13
****** ***************************** Top of Data ****************
000001 //CTM0001A   JOB (UNIVER),'CTM0001',CLASS=A,REGION=4096K,
000002 //           MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
```

```
000003 //*-------------------------------------------*
000004 //TRIDENI  PROC
000005 //TRID     EXEC PGM=SORT
000006 //SORTIN   DD DISP=SHR,DSN=&VSTUP
000007 //SORTOUT  DD DISP=SHR,DSN=&VYSTUP
000008 //SYSOUT   DD SYSOUT=*
000009 //         PEND
000010 //*-------------------------------------------*
000011 //STEP1  EXEC TRIDENI,VSTUP=CTM0001.TRID2.JCL(DATA3),
000012 //       VYSTUP=CTM0001.TRID2.JCL(VYSTUP)
000013 //SYSIN  DD *
000014    SORT  FIELDS=(1,75,CH,A)
000015 /*
****** **************************** Bottom of Data ***************
```

Na řádcích 000003 a 000010 si všimněte použití komentáře, uvozeného //*. Na řádcích 000011 a 000012 je potom vidět, jak se volá procedura s parametry.

3.4.4. Katalogizovaná procedura

Na závěr ještě zkusme přemístit proceduru do jiného data setu a vytvořit tak knihovnu s katalogizovanou procedurou.

Data set s procedurou:

```
EDIT       CTM0001.TRID3.JCL(TRIDENI) - 01.02
****** **************************** Top of Data ***************
000100 //TRIDENI  PROC
000200 //TRID     EXEC PGM=SORT
000300 //SORTIN   DD DISP=SHR,DSN=CTM0001.TRID2.JCL(DATA3)
000400 //SORTOUT  DD DISP=SHR,DSN=CTM0001.TRID3.JCL(VYSTUP)
000500 //SYSOUT   DD SYSOUT=*
000600 //         PEND
****** **************************** Bottom of Data *************
```

Data set s úlohou volající proceduru:

```
EDIT       CTM0001.TRID3.JCL(PRIKAZY) - 01.26
****** **************************** Top of Data ****************
000100 //CTM0001A   JOB (UNIVER),'CTM0001',CLASS=A,REGION=4096K,
000200 //           MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
000300 //MYLIB    JCLLIB ORDER=CTM0001.TRID3.JCL
000410 //STEP1    EXEC PROC=TRIDENI
000600 //SYSIN    DD *
000700    SORT FIELDS=(1,75,CH,A)
000800 /*
****** **************************** Bottom of Data ***************
```

Novinkou oproti předchozímu je tu především řádek 000300 v data setu s příkazy, kterým se otevírá knihovna.

4. Programování v C/C++

4.1. Programovací jazyky na mainframech

4.1.1. Stručný přehled

Obsahem celé této části bude programování jednoduchých aplikací na mainframech. Vzhledem k velkému rozšíření znalosti jazyka C/C++ jsme se rozhodli ilustrovat proces tvorby nové aplikace právě na něm. Nicméně programovací jazyk C/C++ není jediným využívaným programovacím jazykem ve světě mainframů a dokonce ani není nejrozšířenějším. Více než 60% programů se stále píše v assembleru, používají se i jazyky COBOL, PL/I, Java, CLIST, REXX i jiné. Který z nich je vhodné použít pro psaní konkrétního programu, záleží na mnoha okolnostech. V této kapitole si alespoň stručně zmíněné jazyky představíme a následujících se již budeme plně věnovat programování v C/C++.

4.1.2. Assembler

Assembler je na rozdíl od ostatních zmíněných programovacích jazyků velmi jednoduchý. Jeho příkazy jsou jen symbolickým přepisem strojových instrukcí daného počítače. Proto se jeho příkazy na různých strojích liší. Nicméně právě díky této vlastnosti lze pomocí assembleru přistupovat až k jádru problému a ovládat naši aplikaci doslova bit za bitem. Hodí se také pro tvorbu podprogramů pro ostatní programovací jazyky. Svoje assemblery mají kromě mainframů i osobní počítače a třeba také herní konzole.

4.1.3. COBOL

Programovací jazyk COBOL (Common Business-Oriented Language) se poprvé objevil v USA v roce 1959. Od té doby se stále vyvíjí až dodnes, aktuální standard COBOL 2002 podporuje i objektově orientované programování. V COBOLu se dodnes píše velmi mnoho programů, dokonce více než polovina aplikací pro kritické úlohy. Na mainframech se v současnosti používá verze COBOLu jménem IBM Enterprise COBOL for z/OS and OS/390. Ta zvládá spolupráci s jazykem Java a lze ji využít také pro internetové obchodování a zpracování dat zapsaných v Unicode či XML. Příkazy COBOLu jsou více podobné angličtině než bývá u programovacích jazyků běžné a celý jazyk je zaměřený na tvorbu aplikací pro obchodování.

4.1.4. PL/I

PL/I (Programming Language/I, zkratka se čte jako „P-L one“) byl vyvinut přímo společností IBM pro použití na prvních mainframech z rodiny S/360. Jazyk PL/I měl být univerzálním jazykem, spojujícím programování pro komerční účely (do té doby COBOL), pro účely vědecké (FORTRAN) a strukturované programování (tehdy jazyk ALGOL). Představen byl roku 1964 a v sedmdesátých letech byl poměrně úspěšný, i když ne tak, jak jeho tvůrci doufali. Používal se nejen na mainframech, ale např. společnost Intel s jeho pomocí vyvíjela software pro její procesory 8080 a 8085 a dnes existují jeho verze i pro Windows a Unix.

Mezi jeho hlavní klady patřila univerzálnost a jednoduchost použití, ze záporů to je přílišná složitost překladu a snaha vyjít vstříc všem vedoucí k tomu, že vědci PL/I považovali za příliš obchodnický a naopak. Standard jazyka byl vydán v roce 1987. Ve své době to byl jistě krok vpřed (nově ukazatele na všechny datové typy atd.), ale dnes už se hodí jen pro případné úpravy programů ze sedmdesátých let.

4.1.5. Java

V současnosti velmi populární jazyk Java pochází až z počátku devadesátých let minulého století. Je to moderní objektově orientovaný programovací jazyk, odvozený z C/C++. Jeho použití na mainframech je spjaté s internetovými aplikacemi i s vývojem nových programů pro komerční sféru. Rozhraní pro práci s Javou jsou vytvořena pomocí jazyka PL/I. Pomocí javovských apletů lze například rozběhnout emulátor ovládacího terminálu 3270 pro z/OS na osobním počítači s Windows. Bohužel programování v Javě zatím nedosahuje na mainframech takové popularity jako na jiných platformách. Možná je to dáno její moderností, nesplňující oblíbenou strategii zpětné kompatibility.

4.1.6. CLIST

O CLISTu jsme se již zmínili v podkapitole 2.3.3. Je to speciální programovací jazyk, používaný pouze na mainframech pro snazší práci s příkazy původního režimu TSO/E. Na rozdíl od většiny programovacích jazyků se program v CLISTu nemusí překládat ani spojovat, pouze se spouští a případné chyby se opravují tak dlouho, dokud aplikace neběží, jak má. Je tu trochu nejednota v názvosloví, neboť název CLIST se používá jak pro programovací jazyk, tak pro sérii příkazů TSO. Rozeznáváme tři hlavní oblasti, kde se CLIST používá:

- Pro psaní rutin
- Pro psaní specielních programů (např. panely ISPF)
- Pro spravování aplikací napsaných v jiných programech

4.1.7. REXX

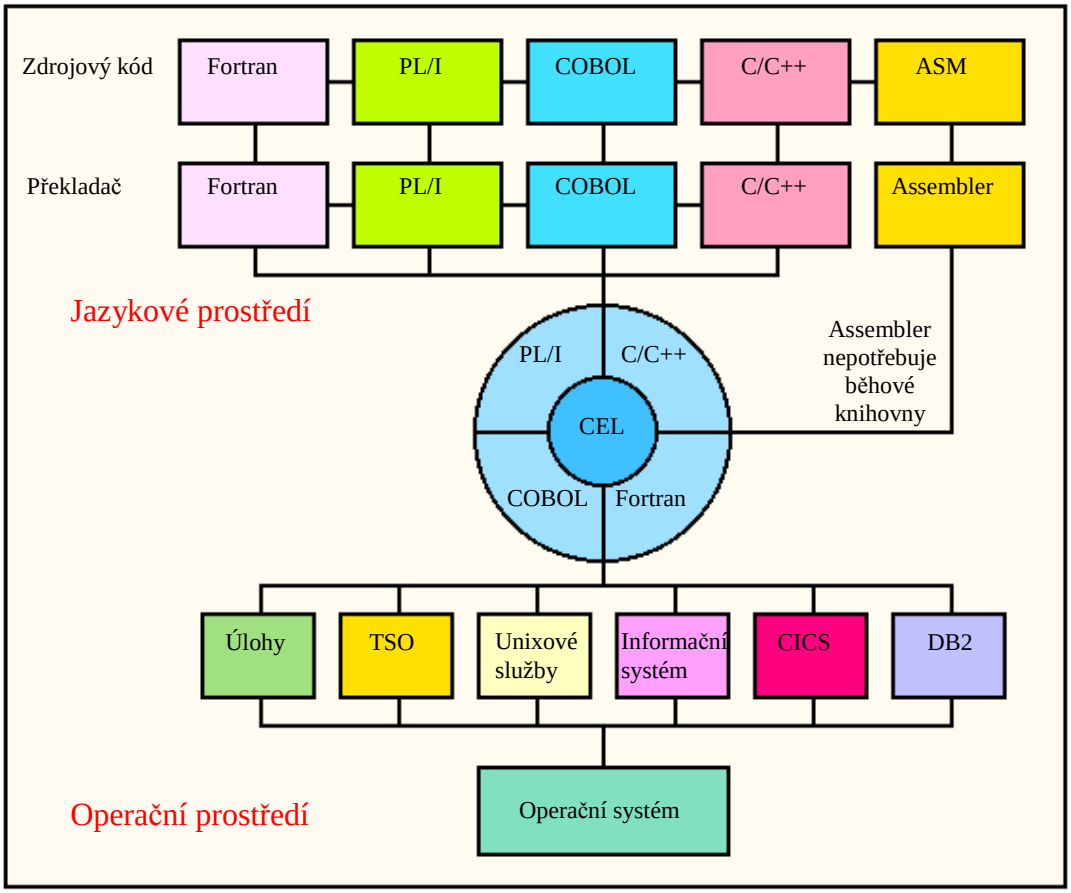
S programovacím jazykem REXX se podrobněji seznámíme v páté kapitole. Nyní si ho jen několika slovy představíme.

REXX byl vyvinut v letech 1979 až 1982 v IBM. Původně to měl být skriptovací jazyk pro libovolný systém (jako např. dnešní Python). Má mnoho verzí, od těch pro mainframy až po varianty pro Windows, Amigy, Linux či Solaris. Od poloviny devadesátých let dvacátého století se objevily dvě verze tohoto jazyka – NetRexx a ObjectRexx. NetRexx silně spolupracuje s Javou a dokonce požívá javovské knihovny. Z tohoto důvodu není se starším „klasickým“ REXXem kompatibilní. Naopak ObjectRexx je vlastně objektově orientovanou verzí starších REXXů.

REXX se vyznačuje například absencí klíčových slov, nedeklarováním proměnných, decimální aritmetikou či bohatým výběrem zabudovaných funkcí. Jak již bylo řečeno, podrobněji se s ním seznámíme v páté kapitole.

4.1.8. Jazykové prostředí

Jazykové prostředí (Language Environment) je nástroj na mainframech, který dokáže spojovat jednotlivé programy napsané v různých vyšších programovacích jazycích do jednotné aplikace. Jazykové prostředí má v sobě zabudovanou podporu jazyků C, C++, COBOL, Fortran, PL/I a Java. Skládá se ze základních rutin, které dokážou spouštět a zastavovat programy, alokovat paměť, komunikovat s programy napsanými v různých jazycích a obsluhovat chyby, ze společných knihoven, poskytujících běžné služby (datum a čas, matematika a podobně), a ze specielních knihoven jednotlivých programovacích jazyků. Obvyklé prostředí vytvářené Jazykovým prostředím můžete vidět na obrázku 16:



Obrázek 16: Obvyklé běhové prostředí Jazykového prostředí

4.2. Programovací jazyk C/C++, průběh vytváření aplikace

4.2.1. Charakteristika jazyka C/C++

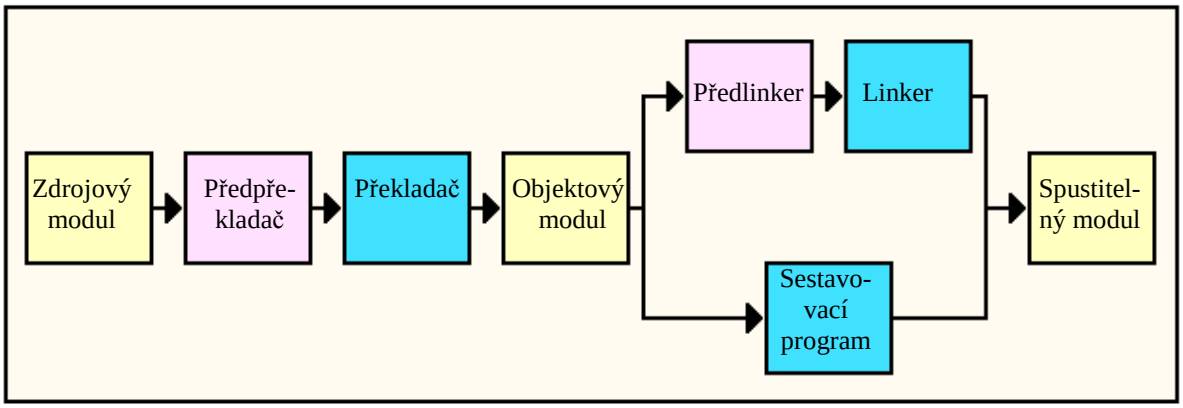
Původní jazyk C vznikl počátkem 70. let 20. století jako programovací jazyk pro operační systém Unix. Od té doby se rozšířil i na většinu dalších operačních systémů a dnes patří společně se svými pozdějšími modifikacemi k nejrozšířenějším programovacím jazykům vůbec. Jeho hlavním kladem je vysoká efektivita kódu, který se jeho prostřednictvím vytváří. Jazyk C je využíván především k psaní systémového softwaru, méně pak k vytváření koncových aplikací. Nejedná se totiž o úplně typický vyšší programovací jazyk (i když se mezi ně počítá), neboť jeho základní příkazy jsou interpretovány pouze několika málo strojovými instrukcemi. Proto jazyk C poskytuje i mnohé výhody nižších programovacích jazyků (assembler).

Vylepšená verze jazyka C zvaná C++ byla vyvinuta v roce 1983 v Bellových laboratořích v USA. Oproti klasickému C podporuje C++ navíc např. objektově orientované programování, dědění, třídy, šablony či přetěžování operátorů.

Jelikož mnoho programátorů používá směs příkazů z C a C++, mluví se obvykle o programování v C/C++. Základní princip vytváření programu v obou jazycích je na mainframech stejný, proto i v této práci mluvíme o programovacím jazyku C/C++. V místech, kde se programování v C a v C++ liší, bude na tuto skutečnost čtenář upozorněn.

4.2.2. Průběh vytváření aplikace

Idea tvorby nového softwaru je ve všech programovacích jazycích na mainframech stejná. Přehledně ji zachycuje obrázek 16. Jednotlivé části procesu si stručně představíme v této kapitole, v následujících kapitolách pak bude následovat podrobnější popis s důrazem na jejich konkrétní podobu při práci s programovacím jazykem C/C++.



Obrázek 17: Tvorba aplikace[1]

Programování na mainframech se obvykle drží modulárního stylu. To znamená, že každý program je rozdělený na jednotlivé moduly, které jsou dále tvořeny jednou nebo několika spřízněnými funkcemi. Výhodou tohoto stylu je, že při vývoji jedné aplikace můžeme používat na různé moduly více odlišných programovacích jazyků.

Na začátku připravíme tzv. zdrojový modul, tedy zdrojový text napsaný v nějakém programovacím jazyce. Pokud neobsahuje chyby, uloží se text v data setu PDS, který se nazývá zdrojová knihovna a který je umístěný na discích DASD. Tam také můžeme nalézt tzv. kopírovací knihu (copybook). Obvykle je to sdílená knihovna, ve které se nalézá často používaný text, který můžeme vkládat do našeho kódu. V tomto případě se nejedná o podprogramy, ale pouze o text, který se přidá do našeho.

První věcí, se kterou se náš zdrojový text může setkat, je předpřekladač. Bývá součástí některých překladačů a jeho úkolem je eliminovat ze zdrojového kódu příkazy, které nejsou součástí programovacího jazyka (např. EXEC CICS nebo EXEC SQL), a nahradit je jeho příslušnými příkazy.

V další fázi se náš zdrojový modul překládá pomocí překladače na objektový modul. Ten obsahuje nespustitelný tzv. objektový kód zpracovatelný dále linkerem nebo spojovacím programem. Při překladu zdrojového kódu získají všechna data a instrukce relativní adresy, aby mohl být program libovolně přemisťován. Odkazy na vnější programy nebo podprogramy zůstávají v této chvíli nevyřešené, dojde na ně až při spojování nebo při běhu programu. Jak odešleme modul s příkazy v C/C++ k překladu, se dozvíte v následující kapitole 4.3.. Objektové moduly se skladují v data setu jménem objektová knihovna (object library).

Následuje zpracování objektových modulů na spustitelné moduly, které může proběhnout dvěma způsoby.

[1] K názvosloví: V duchu filozofie celé práce jsem se i tady snažil nalézt vhodné české ekvivalenty pro anglické termíny. Slovo „compiler“ tedy překládám jednoduše jako „překladač“, logicky pak ze slova „precompiler“ vzniká „předpřekladeč“. Nechtěl jsem však zabřednout do anglických nuancí slov „link“ a „bind“, která se obě do češtiny překládají jako „spojovat“, ale jejichž význam je různý. Proto sloveso „link“ překládám počeštělým „linkovat“ a sloveso „bind“ českým „spojovat“. Analogicky pak vznikají názvy zařízení pro tyto činnosti – ponechaný anglický „linker“ a přeložený „sestavovací program“ z anglického slova „binder“. Název „předlinker“ pro anglický „prelinker“ plyne z jednotnosti s „předpřekladačem“.

Prvním z nich je předlinkování a linkování. Je to starší a méně účinný postup. Na rozdíl od spojování nezvládá např. zpracovávat objektové moduly používající dlouhá jména, 64-bitové moduly a ty, které jsou dynamickými knihovnami DLL či obsahují kód C++. Předlinkování, které se používalo pro odstranění některých zmíněných problémů, nelze upotřebit vždy. Sama IBM doporučuje, kdykoli je to možné, používat druhý způsob, tedy místo linkeru sestavovací program.

Sestavovací program umí totéž co linker a ještě něco navíc a tudíž celý proces linkování lze nahradit spojováním. Při spojování přijímá sestavovací program na vstupu nejen objektové, ale i spustitelné moduly. Jak už jeho název napovídá, jeho úkolem je potom všechny jednotlivé moduly pospojovat dohromady. Tak vznikne jediný výsledný spustitelný modul, uložený v data setu PDS a připravený ke spuštění. Podrobněji si o tom povíme v kapitole 4.4..

Nyní je naše aplikace hotová a připravená ke spuštění. Jak přesně jednotlivé kroky probíhají, bude obsahem příštích kapitol.

4.3. Překlad programů v C/C++

4.3.1. Vstupy a výstupy překladače

Jak už jsme si v minulé kapitole naznačili, první věcí, na kterou zdrojový kód narazí, je překladač. Ten jeho instrukce převede do objektového kódu. Překladač pro jazyk C/C++ má více než padesát různých voleb, které jeho funkci nějak ovlivňují. My si obvykle vystačíme s jejich implicitními hodnotami, zájemci mohou nalézt povídání o každé z nich v použité literatuře. Zmíníme se pouze o některých volbách na konkrétních místech jejich výskytu.

Aby překladač mohl konat svou činnost, potřebuje k tomu kromě našeho zdrojového kódu také naše hlavičkové soubory a standardní hlavičkové soubory. Ty dohromady tvoří jeho vstupní data. Zdrojový kód můžeme ukládat do:

- Sekvenčních data setů
- Jednotlivých členů data setů PDS
- Všech členů data setů PDS

Hlavičkové soubory, které používá náš program, můžeme umisťovat do stejných typů data setů jako zdrojový kód. Překladač je nalezne pomocí hodnoty volby LSEARCH nebo SEARCH v úloze, kterou překlad spouštíme. Jak to vypadá konkrétně, se můžete podívat do kapitoly 4.8. s příklady.

Výstupní data mohou být opět ukládána do sekvenčních i PDS data setů. Záleží ovšem na tom, kde byl uložen vstupní kód. Různé kombinace můžete vidět v tabulce 5. Použité zkratky znamenají: DS – data set, SDS – sekvenční data set, PDS – data set PDS.

Typ DS se zdrojovým kódem	Výstupní DS určená bez jména členu, např. A.B.C	Výstupní DS určená se jménem členu, např. A.B.C(D)
SDS, např. A.B	1. pokud soubor existuje jako SDS, přepíše se 2. pokud soubor neexistuje, vytvoří se jakožto SDS 3. jinak se překlad nezdaří	1. pokud PDS neexistuje, vytvoří se společně se členem 2. pokud PDS existuje a neexistuje člen, vytvoří se člen 3. pokud existuje PDS i člen, pak je člen přepsán

Člen PDS, např. A.B(C)	1. pokud soubor existuje jako SDS, přepíše se 2. pokud soubor existuje jako PDS, vytvoří nebo přepíše se člen 3. pokud soubor neexistuje, vytvoří se PDS a člen	1. pokud PDS neexistuje, vytvoří se společně se členem 2. pokud PDS existuje a neexistuje člen, vytvoří se člen 3. pokud existuje PDS i člen, pak je člen přepsán
Všechny členy PDS, např. A.B	1. pokud soubor existuje jako PDS, vytvoří nebo přepíší se členy 2. pokud soubor neexistuje, vytvoří se PDS a členy 3. jinak se překlad nezdaří	Nelze

Tabulka 5: Kombinace typů vstupních a výstupních data setů

Překladač může produkovat různé typy výstupu. Obvykle je to objektový modul, který lze použít jako vstup pro sestavovací program (kam se ukládá, lze specifikovat pomocí volby překladače OBJECT(soubor)). Dalšími možnostmi je vytvoření výpisového souboru (SOURCE(soubor), LIST(soubor) a INLRPT(soubor), umístění u všech se musí shodovat), výstupu předpřekladače (PPONLY(soubor)), souboru s událostmi (EVENTS(soubor)), šablonového výstupu (TEMPINC(umístění)) a šablonových registrů (TEMPLATEREGISTRY(soubor)). Pokud příkaz zadáme bez určení data setu do něhož se má výstup uložit, vytvoří si překladač data set s vlastním názvem, odvozeným ze jména vstupu.

K čemu jednotlivé výstupy slouží, plyne z jejich názvů. Například ve výpisovém souboru nalezneme zprávy o překladu a kompletní přepis programu do assembleru (část LIST).

4.3.2. Jak odeslat zdrojový kód k přeložení

Jako u spousty dalších činností na mainframech existuje i pro odeslání zdrojového kódu k překladu více možností. Můžeme například použít sérii příkazů pro původní režim TSO či něco z jazyka REXX. Zdaleka nejjednodušší se však jeví překlad pomocí úlohy. Můžete si ji napsat celou sami, ale je zde připraveno i mnoho procedur, které překlad provedou za vás. Přehled procedur pro překlad programu napsaného v jazyce C je v první části v následující tabulce 6, ve druhé části naleznete procedury pro překlad jazyka C++. Za názvem procedury následují činnosti, které procedura zvládá spustit (nejen překlad, ale i linkování, spojování či spuštění), poté informace o adresování paměti programem a o IPA a XPLINK. To jsou optimalizační nástroje, blíže o nich budeme mluvit v další podkapitole.

Název procedury	Překlad	Spojení	Předlinkování	Linkování	Spuštění	31-bitů	64-bitů	XPLINK bez IPA	IPA bez XPLINK	IPA s XPLINK	Poznámka
Jazyk C											
EDCC	x					x	x				
EDCCB	x	x				x					
EDCXCB	x	x				x		x			
EDCQCB	x	x					x				
EDCCL	x			x		x					
EDCCBG	x	x			x	x					

Název procedury	Překlad	Spojení	Předlinkování	Linkování	Rozběhnutí	31-bitů	64-bitů	XPLINK bez IPA	IPA bez XPLINK	IPA s XPLINK	Poznámka
EDCXCBG	x	x			x	x		x			
EDCQCBG	x	x			x		x				
EDCCLG	x			x	x	x					
EDCCPLG	x		x	x	x	x					
EDCCLIB	x					x	x				Pro tvorbu obj. knihovny
EDCI		x				x			x		Linkovací krok IPA
EDCXI		x					x			x	Linkovací krok IPA
CCNPD1B		x				x			x		
CCNXPD1B		x				x				x	
CCNQPD1B		x					x		x		
EDCQB		x					x				
EDCQBG		x			x		x				
Jazyk C++											
CBCC	x					x	x				
CBCCB	x	x				x					
CBCXCB	x	x				x		x			
CBCQCB	x	x					x				
CBCCL	x		x	x		x					
CBCCBG	x	x			x	x					
CBCXCBG	x	x			x	x		x			
CBCQCBG	x	x			x		x				
CBCCLG	x		x	x	x	x					
CBCI		x				x			x		Linkovací krok IPA
CBCXI		x				x	x			x	Linkovací krok IPA
CCNPD1B		x							x		
CCNQPD1B		x					x		x		
CCNXPD1B		x								x	
CBCB		x				x					
CBCXB		x				x		x			
CBCQB		x					x				
CBCXBG		x			x	x		x			
CBCQBG		x			x		x				
CBCLG			x	x	x	x					
CBCG					x	x	x				
CBCXG					x	x	x				

Tabulka 6: Procedury pro zpracování zdrojového kódu v jazyce C/C++

Jak vidíme, některé připravené procedury umí zdrojový text nejen přeložit, ale i spojit či slinkovat objektové moduly dohromady a spustit program. Jméno požadované procedury zadáváme do úlohy do hodnoty parametru PROC příkazu EXEC. Ještě před tím nesmíme zapomenout otevřít knihovnu s příslušnou procedurou. Typicky jsou procedury uskladněné

v data setech CEE.SCEERUN, CEE.SCEERUN2 a CBC.SCCNCMP, ale například na instalaci, se kterou jsem pracoval já, se nalézaly v data setech CEE.SCEEPROC a CBC.SCBCPRC. Nejsnazší se asi bude na to zeptat vzdělanějšího kolegy.

Další věc, kterou uvádíme do úlohy spouštějící překlad, je umístění zdrojového kódu a to prostřednictvím parametru INFILE='jméno_data_setu' příkazu EXEC (pro procedury) nebo příkazem SYSIN DD DSNAME=jméno_data_setu,DISP=SHR (vlastní překlad). Nesmí chybět ani umístění hlavičkových souborů, zadané do parametru CPARM='LSEARCH(''''jméno_data_setu'''')' příkazu EXEC.

Zde si můžeme všimnout čtyř jednoduchých uvozovek kolem jména data setu. Skutečně se tam píší, neboť dáváme do uvozovek něco v uvozovkách a vzhledem k tomu, že v hodnotách parametrů se každé uvozovky píší dvakrát, vychází konečná čtyřka.

Jak taková překládací úloha celkově vypadá, se můžete podívat do kapitoly 4.8 s příklady.

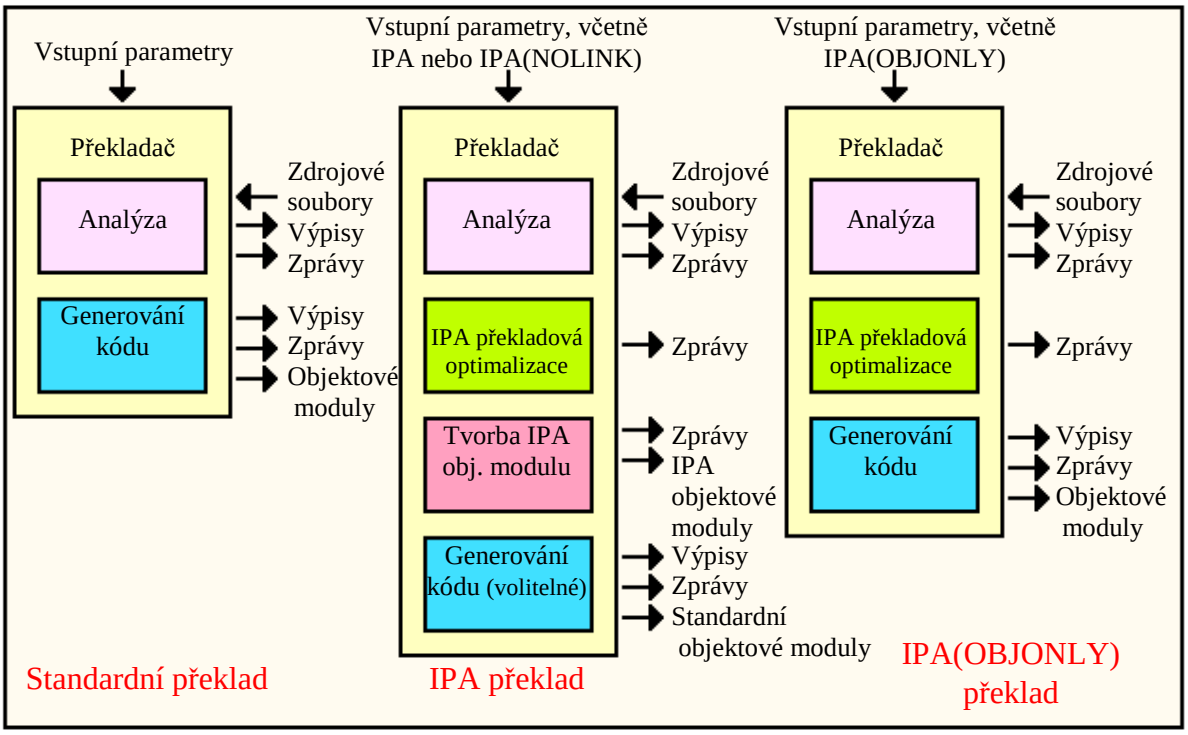
4.3.3. Interprocedurální analýza

Nejsilnější nástrojem pro optimalizaci překladu je tzv. interprocedurální analýza (Interprocedural Analysis, IPA). IPA se při překladu programu úlohou spouští parametrem IPA parametru CPARM příkazu EXEC. Standardní překladače dokáží kód optimalizovat pouze v rámci jednotlivých funkcí, nejlépe pak v rámci jedné překládané jednotky (zdrojový soubor a jeho hlavičkové soubory). Oproti tomu interprocedurální analýza dokáže optimalizovat kód jako celek, napříč jednotlivými zdrojovými soubory a funkcemi. Jednoduše řečeno, IPA sleduje aplikaci globálně jako celek.

Proces interprocedurální analýzy má dva kroky. Prvním z nich je IPA překlad, který se spouští parametrem IPA(NOLINK). Je velmi podobný standardnímu překladu, navíc IPA přidá do objektového modulu informace o překládané jednotce a své speciální údaje. Ty se následně využijí při druhém kroku, zvaném IPA linkovací krok a spustitelném IPA(LINK). Jeho funkce je obdobná jako u standardního spojování. IPA linkovací krok dokáže pospojovat nejen objektové moduly přeložené za asistence interprocedurální analýzy, ale i standardní objektové moduly a spustitelné moduly. Ty samozřejmě nedokáže optimalizovat, ale použít je umí.

Provádění kompletní interprocedurální analýzy dokáže velmi urychlit kód, ale bohužel je dost náročné na paměť a čas. Proto existuje ještě volba IPA(OBJONLY). V tomto případě se jedná o kompromis mezi standardním překladem a klasickou interprocedurální analýzou. Jde o upravený proces překladu. Použijí se při něm stejné optimalizace, jako při IPA překladu, ale negenerují se žádné specielní údaje pro spojování. Takto vzniklý objektový modul může být použitý jako vstup pro linker, sestavovací program i IPA linkovací krok. Ten už ale nemůže tyto moduly optimalizovat, neboť, jak bylo řečeno, neobsahují informace pro interprocedurální analýzu.

Srovnání klasického, IPA a IPA(OBJONLY) překladu můžete vidět na obrázku 18.



Obrázek 18: Různé typy překladu

4.3.4. Další možnosti optimalizace

Na závěr kapitoly o překladu zmíníme ještě dva další způsoby optimalizace kódu.

Prvním z nich je použití parametru překladače XPLINK. Pokud ho použijeme, říkáme tím překladači, že má spojit jednotlivé podprogramy v překládaném zdrojovém souboru tzv. vysoce výkonnými vazbami (extra performance linkage). To znamená snižování vnějších spojení podprogramů a ukládání parametrů volaných funkcí přímo do registrů. Tím se samozřejmě zvýší výkon naší aplikace. Sám XPLINK má ještě několik vlastních podparametrů a pokud se o něm chcete dozvědět více, doporučuji použitou literaturu.

Druhým způsobem je nástroj zvaný Tvarem řízená zpětná vazba (Profile-Directed Feedback, PDF). Tento nástroj dokáže sledovat, jak často jsou jednotlivé části kódu používány. Například obsluha výjimek bude využita méně často než funkce main. PDF pak překladači oznámí, které části kódu se obvykle neupotřebí, a ten se příliš nemusí zdržovat s jejich optimalizací. Postup je takový, že nejprve přeložíme program standardně, pak ho s PDF chvíli sledujeme a následně ho necháme přeložit znovu se zřetelem na informace PDF. Hodnoty jednotlivých použitých parametrů překladače nalezne čtenář opět v literatuře.

4.4. Spojování programů v C/C++

4.4.1. Kdy lze použít sestavovací program

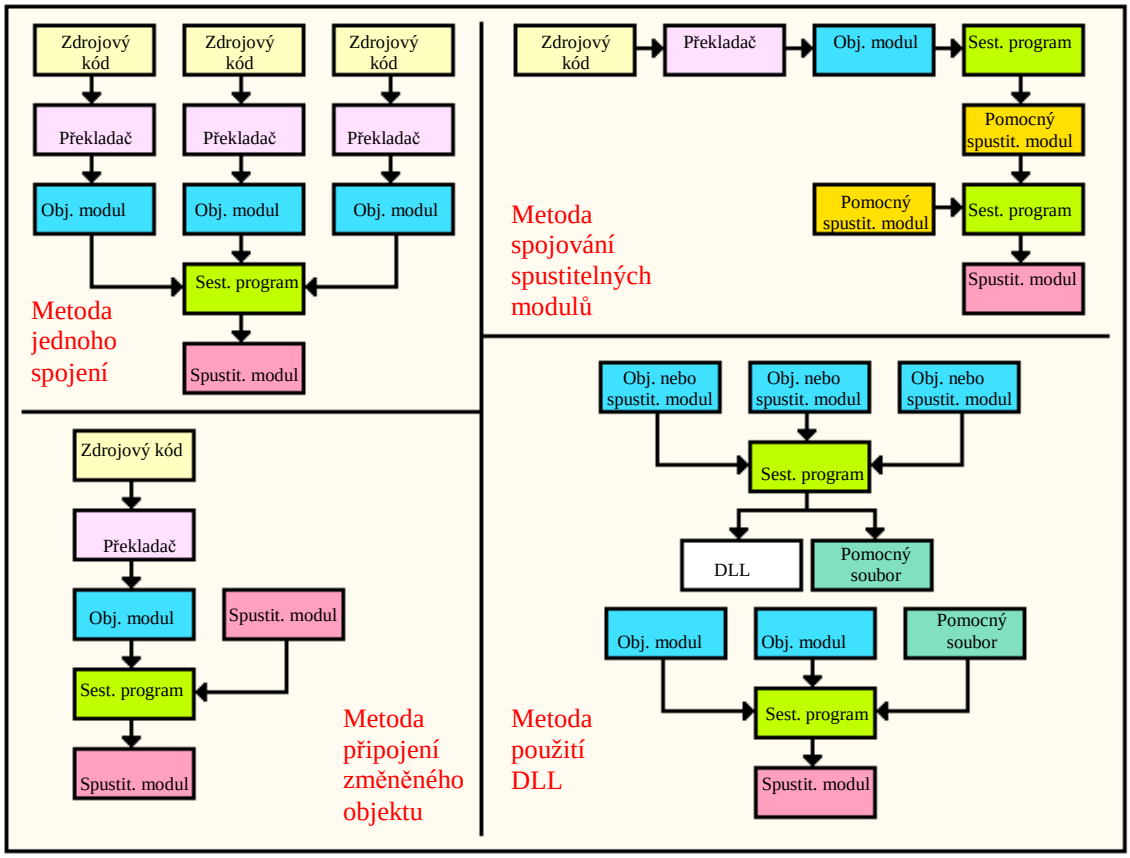
Pokud máme program přeložený, můžeme jednotlivé části spojit dohromady. Jak už jsem se zmínil, jde to dvěma způsoby – pomocí sestavovacího programu nebo pomocí linkeru. První možnost je daleko šířeji použitelná, proto ji tu popíšeme trochu detailněji, druhá se liší především v technickém zpracování a její podrobný popis lze najít v literatuře.

Sestavovací program lze použít vždy, když jeho výstup ukládáme do vylepšené rozšířené knihovny PDSE. Tyto data sety používáme ve většině případů, i když jim říkáme jen PDS. Pokud si přejeme mít výstup uložený v obyčejném data setu PDS, musíme upotřebit

předlinker v případech, kdy spojujeme kód v jazyce C++ nebo v jazyce C přeložený s volbou XPLINK. Sestavovací program také nelze aplikovat při práci s transakčním serverem CICS starším než verze 1.3 a pro zpracování objektového souboru přeloženého pomocí IPA(NOLINK,NOOBJECT). Ten obsahuje pouze informace interprocedurální analýzy a žádný objektový modul. Naopak objektové moduly IPA vzniklé IPA překladem sestavovací program propojit umí – prostě ignoruje IPA informace a spojí je jako klasické.

4.4.2. Metody spojování

Sestavovací program umí na vstupu přijímat nejen objektové, ale i spustitelné moduly. Z toho plyne, že existuje více cest k získání požadovaného programu. Na následujícím obrázku 19 můžete vidět srovnání jednotlivých metod, jejich slovní popis následuje poté.



Obrázek 19: Různé metody spojování

První z nich je metoda jednoho spojení. Každý zdrojový soubor samostatně přeložíme do objektového modulu a ty pak naráz všechny spojíme dohromady. Hodí se pro první vytvoření aplikace. Výhodou tohoto postupu je jeho jednoduchost a shoda s existujícími metodami stavby aplikací (např. unixovské makefile).

Druhá cesta je metoda spojování spustitelných modulů. Při ní každý zdrojový modul přeložíme a protáhneme sestavovacím programem zvlášť. Tak vzniknou pomocné spustitelné moduly, které nemusí mít vyhodnoceny všechny odkazy na procedury mimo modul (tudíž vlastně nejsou spustitelné). Tyto pomocné moduly poté všechny pošleme sestavovacímu programu a ten z nich vytvoří jeden konečný spustitelný modul (který už spustit lze). Tato metoda se hodí, pokud měníme část zdrojového kódu pouze v některém zdrojovém objektu. Její hlavní výhodou je vyšší rychlost oproti metodě jednoho spojení.

Pokud si přejeme použít metodu vytvoření a použití DLL, zvolíme třetí cestu. Aby mohla DLL vzniknout, musí zdrojový kód obsahovat symboly pro export. Lze použít volbu překladače EXPORTALL nebo direktivu #pragma pro označení symbolů C/C++, které mají být exportovány. Při spojování pak sestavovací program vygeneruje DLL a vedlejší soubor obsahující příkazy IMPORT a exportované symboly (pro každý symbol jeden IMPORT). Takto vytvořenou dynamickou knihovnu pak můžeme použít pro tvorbu naší aplikace. Ta musí být napsaná v C++ nebo v C přeloženém s parametrem překladače DLL. U každého symbolu, který chceme importovat z DLL, musí být uveden příkaz IMPORT. Při běhu aplikace se pak podle pomocného souboru přiřazují správné symboly. Výhodou tohoto postupu je, že při změně DLL se nemusí znovu přestavovat aplikace, které ji používají.

Poslední cestou, jak používat sestavovací program, je metoda připojení změněného objektu. Pokud totiž změníme pouze jeden ze zdrojových objektů, můžeme ho přeložit a následně poslat sestavovacímu programu společně s původním spustitelným modulem. Sestavovací program dokáže ve spustitelném modulu nahradit části odpovídající změněnému objektovému modulu pomocí nového. Využití se nabízí samo – při upgradu programů stačí uživateli dodat změněný modul a on si ho jen připojí ke své aplikaci. Jako hlavní klad tohoto postupu můžeme jmenovat jeho rychlost.

4.4.3. Jak spojovat

Jako obvykle vede k cíli více cest. Nejsnazší je opět použít úlohu s připravenou procedurou. Použitelné procedury naleznete v tabulce 6 v podkapitole 4.3.2.. Pokud si přejete vytvářet dynamickou knihovnu DLL, je třeba přidat volbu DYNAM(DLL) do parametrů zvolené procedury. Složitější cesty jsou příkazy pro původní režim TSO nebo vlastní úloha se svými příkazy jazyka JCL pro ovládání sestavovacího programu.

4.5. Spouštění programů

4.5.1. Přidělení paměti

Ve chvíli, kdy máme aplikaci přeloženou, ji obvykle chceme spustit. Aby to šlo, potřebujeme si pro náš program vyžádat přiměřené množství paměti. Standardní předdefinovaná hodnota je 48 MB. Změnit ji můžeme pomocí parametru REGION=xM. Místo x napíšeme požadovanou hodnotu. Ovšem pozor! Náš program nedostane tolik paměti, kolik zadáme do tohoto parametru, ale tolik, kolik ukazuje následující tabulka:

Hodnota parametru REGION	Přidělená paměť	Celková paměť pro program
0M nebo 0K	Všechna volná do a nad hodnotu 16 MB	Neodhadnutelné
(0M, 16M)	Pokud je volná paměť požadovaného množství v oblasti do 16 MB, tak se přidělí, jinak nestandardní ukončení. Navíc dostane program 32 MB nad hranicí 16 MB.	REGION + 32 MB
<16M, 32M)	Všechna volná do 16 MB, navíc 32 MB nad hranicí 16MB.	Něco (8 až 10 MB) + 32 MB
<32M, 2047M)	Všechna volná do 16 MB, navíc požadované množství v oblasti nad 16 MB, pokud tam není dostatečné množství, nastane nestandardní ukončení	Něco (8 až 10 MB) + REGION

Tabulka 7: Přidělování paměti

4.5.2. Jak spouštět aplikaci

Abychom mohli spouštět naše aplikace pomocí úloh, potřebujeme mít knihovny SCEERUN, SCEERUN2, SCLBDLL a SCLBDLL2. Ty ovšem potřebujeme i pro překlad a spojování, takže to není nic nového.

Pokud je máme, můžeme si napsat stejnou úlohu jako pro překlad či spojování, jen použijeme jinou proceduru. Kromě procedur z tabulky 6 v kapitole 4.3.2. můžeme ještě využít třeba proceduru CBCG, která pouze spustí program napsaný v C++ (který byl už dříve přeložený a spojený).

Z/OS nám také dovoluje přizpůsobovat běhové volby (run-time options) našich programů, jako je úprava jejich výkonu nebo obsluha výjimek. Aby to bylo možné, musíme aplikaci přeložit s parametrem překladače EXECOPS. Implicitně překladač tento parametr používá, takže ho ani zdůrazňovat nemusíme. Běhové volby potom ovládáme hodnotami, zadanými do parametru GPARM použitých procedur pro spuštění programu. Když nepoužíváme připravené procedury, ale vlastní JCL, zadáváme tyto hodnoty do parametru PARM příkazu EXEC. Ve zdrojovém kódu tato místa v obou případech označuje příkaz #pragma runopts.

Jestliže chceme zadávat našemu programu nějaké parametry, lze toho docílit umístěním lomítka za zvolené běhové volby do parametru GPARM, resp. PARM. Hodnoty před lomítkem pak budou běhovými volbami a za lomítkem se odešlou jako parametry funkci main(). Jak to vypadá, můžeme vidět na následujícím řádku:

```
GPARM='běhové_volby/první_parametr_main,druhý_parametr_main'
```

Jako příklad možné běhové volby lze zmínit RPTOPTS(ON), pomocí kterého říkáme, že si přejeme vygenerovat zprávu o paměti a zprávu o běhových volbách.

4.6. Vstupy a výstupy programů

4.6.1. Typy vstupních a výstupních dat

Nyní bychom měli alespoň teoreticky být schopni napsat, přeložit, spojit a spustit jednoduchý program v C/C++. V této kapitole se dozvíme, jak náš program zpracovává potřebná vstupní a výstupní data. Abychom nemuseli stále psát „vstupy a výstupy“, budeme občas používat i obecně známou anglickou zkratku „I/O“ (Input/Output).

Budeme uvažovat tři typy dat a jejich zpracování: textové řetězce, binární řetězce a záznamy I/O[1]. Řetězcem v tomto případě rozumíme posloupnost znaků a záznamem I/O kolekci dat, tvářící se jako nedělitelná jednotka.

Textové řetězce mohou obsahovat tisknutelné a kontrolní znaky. Jsou uspořádané v řádcích a každý řádek končí kontrolním znakem, který na to upozorňuje. Obvykle to bývá symbol pro novou řádku „\n“ nebo pro návrat vozíku „\r“. Existují i další kontrolní znaky, které však v našem povídání zmiňovat nebudeme. Kromě standardních textových řetězců ještě existují textové řetězce ASA (American Standards Association). Mají zvláštní formát (v prvním sloupci kontrolní znak) a z/OS C/C++ je zpracovává jinak než obvyklé textové řetězce. Musí být uloženy ve zvláštních data setech, které zmíníme za chvilku. Celkově se však textovým řetězcům ASA příliš věnovat nebudeme.

Binární řetězce obsahují posloupnost bajtů. Jejich hodnoty se při použití na I/O nepřekládají a celý řetězec se nijak nedělí.

[1] Musíme rozlišovat záznamy I/O, které jsou formátem vstupních a výstupních dat a logické záznamy, které budeme zmiňovat v následující podkapitole a které odkazují k systému ukládání dat do data setů.

Pokud máme soubor obsahující záznamy I/O, můžeme vždy pracovat právě s jedním záznamem I/O. Pro přístup můžeme používat pouze funkce fread() a fwrite(), nikoli jiné jako např. fprintf() nebo fscanf(). Při přepisování dat v záznamech I/O se velikost záznamů nemění. Pokud bychom tedy do něj chtěli uložit více dat, než je jeho velikost, budou naše data patřičně zkrácena. Když má mít nový záznam naopak méně znaků než původní, přepíší se pouze znaky na začátku záznamu I/O a zbytek se nemění. Záznamy I/O jsou uložené v binární podobě, proto na ně kontrolní znaky textových řetězců nemají vliv.

4.6.2. Modely ukládání dat

Data programu mohou přicházet z různých míst a mohou být odesílána na odlišná umístění. Podle toho, odkud a kam to je, jsou údaje uloženy odlišným způsobem. Mluvíme buď o tzv. bajtovém nebo o tzv. záznamovém modelu ukládání dat.

Bajtový model ukládání dat znamená, že informace jsou uložené v binárním souboru bez vnitřní struktury. Tento model ukládání se využívá pouze při práci se soubory z/OS UNIXu a s paměťovými soubory. To jsou dočasně vytvořené soubory, které nemají strukturu data setů a užívají se pro rychlý přístup k datům.

Záznamový model ukládání dat upotřebíme ve všech ostatních případech. Pro umisťování údajů se používá struktura bloků a záznamů, která odpovídá struktuře bloků a logických záznamů data setů, jak jsme se s nimi seznámili v podkapitole 2.4.4.. Záznamem v tomto případě nazýváme jednotku informace odesílanou a přijímanou programem, blokem pak údaje přenášené z a do zařízení. Jeden blok může obsahovat i více záznamů. Abychom předešli záměně těchto záznamů a záznamů I/O, budeme tyto záznamy dále nazývat logické záznamy.

4.6.3. Ukládání různých typů dat v bajtovém modelu

Vzhledem k tomu, že údaje uložené pomocí bajtového modelu nemají žádnou vnitřní strukturu, je ukládání všech tří typů velmi jednoduché.

Pokud ukládáme textový řetězec, jeho kontrolní znaky se uloží tak, jak jsou, a při opětovném znovuotevření jsou na vstupu normálně načteny. Tento model dokonce ani nerozlišuje mezi standardním a ASA textovým řetězcem.

Stejně tak se ukládají binární řetězce – prostě tak, jak jsou.

Trochu složitější je pouze umisťování záznamů I/O. Jelikož soubor, do kterého ukládáme, nemá žádnou strukturu, musíme ji vhodně nasimulovat. Dělá se to tak, že na konec dat z každého záznamu I/O se připojí kontrolní znak „\n“. To má i svou nevýhodu – pokud se tento znak v záznamu již nachází, při následném načtení dat ze souboru se záznamy I/O bude tento záznam načten jako dva. Proto se doporučuje tyto znaky do záznamů vůbec nepsat. Paměťové soubory dokonce ani ukládání záznamů I/O nepodporují.

4.6.4. Různé druhy záznamového modelu ukládání dat

Jak si pamatujeme z podkapitoly 2.4.4., u data setů můžeme určovat formáty jejich logických záznamů, jejich velikost a velikost bloků. Stejně tak je tomu u vstupních a výstupních souborů našeho programu. Jak se to dělá konkrétně, uvidíme v kapitole 4.7.. Nyní si jen připomeneme a rozšíříme naše znalosti o formátech logických záznamů. Podle nich totiž můžeme rozdělit záznamový model ukládání dat na několik druhů a u každého druhu si říci, jak se do něj umisťují všechny tři typy vstupních a výstupních dat.

Formát logických záznamů určujeme pomocí parametru RECFM. Základní hodnoty jsou tři: „F“ pro pevnou (stejná délka všech), „V“ pro proměnnou (různá délka) a „U“ pro nedefinovanou délku (různá délka, bez identifikátorů popisujících délku) logických záznamů. Zkráceně budeme mluvit o pevném, proměnném a nedefinovaném formátu. Podle těchto

hodnot také budeme rozdělovat záznamový model na tři druhy. Tyto základní volby můžeme dále kombinovat s následujícími hodnotami:

- „A“ – soubor obsahuje textový řetězec ASA
- „B“ – soubor je blokový, tedy v každém bloku může být více logických záznamů (bez této volby jeden logický záznam = jeden blok)
- „M“ – soubor obsahuje znaky pro kontrolu stroje
- „S“ – soubor je ve standardním nebo rozšířeném formátu

Volba „S“ znamená různé věci v závislosti na ostatních parametrech. Pokud je navíc formát pevný (F), značí „S“ standardní formát. To znamená, že každý blok se nejprve zaplní záznamy a pak teprve vzniká další. Všechny bloky až na poslední tedy obsahují stejný počet logických záznamů, což je praktické a zvyšuje to výkon aplikace. Když je navíc formát proměnný (V), označuje „S“ rozšířený formát. V něm smí mít logický záznam větší délku než blok. V tomto případě se části logického záznamu ukládají do několika po sobě následujících bloků.

Stejně jako u data setů obsahují logické záznamy v proměnném formátu na začátku čtyřbajtový identifikátor, který popisuje jejich délku. Když je formát proměnný blokový, navíc se nalézá podobný identifikátor i na začátku bloků. Pokud máme formát proměnný rozšířený, identifikátor se umisťujeme též na začátky jednotlivých částí rozděleného logického záznamu. To je hlavní rozdíl oproti logickým záznamům v nedefinovaném formátu.

Hodnoty formátu logických záznamů nelze sdružovat zcela libovolně. Povolené kombinace můžete vidět v následující tabulce:

Základní formát	Povolené kombinace
Pevný (F)	F, FA, FB, FM, FS, FBA, FBM, FBS, FSA, FSM, FBSA, FBSM
Proměnný (V)	V, VA, VB, VM, VS, VBA, VBM, VBS, VSA, VSM, VBSA, VBSM
Nedefinovaný (U)	U, UA, UM

Tabulka 8: Povolené hodnoty RECFM

4.6.5. Ukládání dat v záznamovém modelu s pevnou délkou logických záznamů

Pokud umisťujeme textové řetězce do souboru s pevnou délkou logických záznamů, kontrolní symboly značící konec řádky se do logických záznamů neukládají. Namísto nich se zapisují prázdné symboly a to počínaje pozicí kontrolního znaku a konče posledním volným místem v logickém záznamu (hodnota LRECL). Takže např. pokud zapisujeme řetězec „AHOJ\n“ do logického záznamu délky 7 (LRECL=7), bude výsledkem: „AHOJ_ _ _“. Při načtení takovéhoto vstupu se hned za platné znaky automaticky doplní symbol „\n“. V důsledku toho postupu ztrácíme při uložení a následném načtení všechny prázdné znaky umístěné před kontrolním symbolem značícím konec řádky.

Do existujícího logického záznamu můžeme uložit textový řetězec, který má nejvýše tolik znaků, kolik je délka (LRECL) logického záznamu plus kontrolní znak „\n“. Při znovuzapisování se starý obsah přepíše novým a prázdnými symboly. Např. pokud bychom uložili do logického záznamu z minulého příkladu řetězec „BAF\n“, výsledkem by bylo „BAF_ _ _ _“. Když bychom do logického záznamu chtěli zapsat více znaků, než je jeho velikost, budou přebytečné znaky useknuty a ztraceny. Útěchou nám může být, že nás na to z/OS upozorní pomocí hlášení „errno“ a chybovým příznakem.

Když tu chceme uložit binární řetězec, je to jednodušší. Symboly volně pokračují ze záznamu do záznamu. Pokud se nějaký logický záznam nenaplní celý, doplní se při zavírání souboru nulami („\0“). Tyto nuly tu bohužel zůstávají nastálo. Když příště otevřeme náš soubor, zobrazí se jako součást jeho dat. Jako příklad si uložme binární řetězec délky 10 do

logických záznamů s délkou čtyři (LRECL=4). Při tomto zapisování se dva logické záznamy naplní daty a ve třetím se za 2 symboly ze řetězce uloží nuly.

Úplně nejjednodušší je ukládat do souboru s pevnou délkou logických záznamů záznamy I/O. Každému logickému záznamu tu odpovídá jeden I/O záznam. Jestliže je jeho délka větší než LRECL, data se useknou a ztratí, pokud je menší, doplní se nulami.

4.6.6. Ukládání dat v záznamovém modelu s proměnnou délkou logických záznamů

Textové řetězce se do souboru s proměnnou délkou logických záznamů ukládají snáze než do těch s pevnou. Hranice záznamů tu totiž reprezentují pozici kontrolního znaku konce řádky (stejně „\n“ i „\r“). Z toho plyne, že tyto znaky se do souboru nezapisují. Při čtení souboru se konec logického záznamu převede na symbol „\n“. Pokud přepisujeme takovýto soubor, velikost jednotlivých logických záznamů se už nemění a platí stejná pravidla jako pro soubory s pevnou délkou logických záznamů (doplňování nulami, resp. ztráta dat). Když ukládáme pouze řetězec „\n“, uloží se jako záznam, obsahující jeden prázdný symbol, tedy stejně, jako „ \n“.

Binární řetězce se zde ukládají tak, že pokud jejich délka nepřekročí LRECL mínus 4, uloží se do jednoho logického záznamu potřebné délky. V opačném případě pokračují znaky z řetězce do dalšího logického záznamu.

Záznamy I/O se tady ukládají stejně jako do souborů s pevnou délkou logických záznamů, jen pokud je délka záznamu I/O menší než LRECL, nedoplňuje se nulami, ale vytvoří se kratší logický záznam.

4.6.7. Ukládání dat v záznamovém modelu s nedefinovanou délkou logických záznamů

Pro ukládání textových řetězců do souboru s nedefinovanou délkou logických záznamů platí stejná pravidla jako pro jejich ukládání do souboru s proměnnou délkou logických záznamů.

Binární řetězce při ukládání postupně zaplňují jednotlivé bloky.

Záznamy I/O se ukládají do logických záznamů (totožných s bloky), vždy jeden záznam I/O do jednoho bloku. Délka záznamu I/O nesmí přesáhnout délku bloku (BLKSIZE), jinak dochází k useknutí a ztrátě dat.

4.7. Otevírání souborů v C/C++

4.7.1. Typy vstupů a výstupů

V předchozí kapitole jsem se seznámili s technikou zpracování vstupních a výstupních dat. V této kapitole si povíme několik základních informací o otevírání konkrétních souborů v našich programech. Věnovat se budeme především tomu, jak otevřít nám dobře známé data sety, protože popis otevírání všech možných typů vstupů a výstupů by vydal na samostatnou práci.

Na začátek se podívejme, jaké typy souborů dokážeme pomocí našich aplikací používat. Zachycuje to tabulka 9:

Typ souboru	Model ukládání dat	Poznámka
Vstupy a výstupy OS	Záznamový	Různé data sety (např. sekvenční, PDS), vstupy a výstupy tiskáren, pásky, děrné štítky apod.
Soubory z/OS UNIXu	Bajtový	
Data sety VSAM	Záznamový	

Terminálové I/O	Záznamový	Pro interaktivní vstupní a výstupní operace, prováděné prostřednictvím terminálu, funguje pouze při spouštění programu přes TSO
Paměťové soubory	Bajtový	Dočasné soubory pro rychlý přístup k datům
Paměťové soubory Hiperspace	Bajtový	Dočasné soubory pro rychlý přístup k datům o velikosti 2 GB
Datové fronty CICS	Záznamový	Pro práci se CICS
Soubory se zprávami Jazykového prostředí	Záznamový	

Tabulka 9: Typy vstupů a výstupů

K otevírání všech těchto souborů na vstupu naší aplikace můžeme použít funkce jazyka C fopen() nebo freopen(), popřípadě konstruktory tříd jazyka C++ ifstream, ofstream a fstream nebo funkci open(), která je členem tříd filebuf, ifstream, ofstream a fstream. V dalším textu však budeme pracovat většinou s funkcí fopen().

Jaký typ vstupu otevíráme, pozná z/OS C/C++ podle parametrů otevírající funkce. Například pokud má fopen() otevírat paměťový soubor, musí obsahovat parametr „type=memory“, nebo pro paměťový soubor Hiperspace to je podobný parametr „type=memory(hiperspace)“.

4.7.2. Otevírání vstupů a výstupů OS

Vstupy a výstupy OS jsou většinou různé data sety. Z toho plyne, že program vždy musí znát jejich hlavní vlastnosti, kterými jsou formát logických záznamů (vlastnost RECFM), velikost logických záznamů (LRECL) a velikost bloků (BLKSIZE). Při otevírání žádaného souboru je hledá postupně na následujících místech:

- v parametrech otevírající funkce fopen() přímo v programu napsaném v C/C++
- v příkazu DD jazyka JCL, který se zadává před vlastním spuštěním programu v C/C++
- v informacích o existujícím souboru
- v předdefinovaných hodnotách

Všechna tato místa si nyní podrobněji popíšeme.

Funkce fopen() má strukturu fopen(“jméno_souboru“, “parametry“). Za jméno_souboru dosadíme název požadovaného data setu v jednoduchých uvozovkách, před které ještě napíšeme dvě lomítka. Jaké parametry má tato funkce, je shrnuto v tabulce 10:

Parametr	Hodnoty	Poznámka
recfm=	Libovolné z tabulky 8 nebo *	Formát logických záznamů data setu; * značí, že se použije formát existujícího souboru
lrecl=	0 až 32760, X	Velikost logických záznamů; X značí libovolnou velikost
blksize=	0 až 32760	Velikost bloků
space=	Viz kap. 3.2.5.	Funguje pouze při otevírání neexistujícího souboru, hodnoty stejné, jako parametr SPACE příkazu DD JCL
type=	record	Pro otevření záznamů I/O
byteseek		Pro binární soubory pro určení, že se má používat relativní posun pro prohledávací funkce
noseek		Zákaz používání funkcí ftell(), fseek() apod.

Tabulka 10: Parametry funkce fopen()

Pokud otevíráme již existující soubor, musí mít přirozeně parametry takové hodnoty, jako má otevíraný soubor, nebo je ani nemusíme uvádět. Pokud bychom se totiž snažili otevřít například data set s pevným formátem jako data set s proměnným, došlo by k chybě a neotevřelo by se nic. Stejně tak musíme zadávat hodnoty všech parametrů tak, aby to dávalo smysl.

Pomocí příkazu DD jazyka JCL a jeho parametrů můžeme také specifikovat parametry později otevíraného data setu. Napsat ho lze například do úlohy, kterou program spouštíme. Navíc tím data setu přiřadíme identifikátor zvaný „ddname“. Ten se hodí pro relativní adresování data setů – v programu nepoužíváme plné jméno data setu, ale pouze jeho ddname. Když se pak jeho jméno změní, nemusíme předělávat program, ale stačí upravit přiřazení.

Informace o existujícím souboru dodá funkci fopen() operační systém.

Pokud se ani na jednom z míst funkce fopen() požadované hodnoty nedozví, sáhne pro předdefinované hodnoty. Když není určen formát logických záznamů, určí se následovně:

- potřebujeme binární soubor
  - soubor pro tisk → recfm=VB
  - jinak → recfm=FB
- potřebujeme textový soubor
  - parametr _EDC_ANSI_OPEN_DEFAULT má hodnotu Y a ani RECL ani BLKSIZE nejsou určeny → recfm=F
  - soubor pro tisk → recfm=VBA
  - terminálový soubor → recfm=U
  - je určeno LRECL nebo BLKSIZE → recfm=V
  - jinak → recfm=VB

Dále se určují velikost logických záznamů a bloků. Jakým způsobem se tak děje, můžete vidět v tabulce 11. Názvy LRECL, BLKSIZE a RECFM značí velikost logických záznamů, velikost bloků a formát logických záznamů. Rozdělení nezávisí na volbách S, A a M u formátu logických záznamů. Hodnota max záleží na zařízení, na kterém se má soubor vytvořit (např. 6144 pro disky DASD, 132 pro tiskárny, 80 pro čtečky).

Známe LRECL?	Známe BLKSIZE?	RECFM	Výsledné LRECL	Výsledné BLKSIZE
ne	ne	F	80	80
		FB	80	80 * [max/80]
		V, VB	min {1028, max-4}	max
		U	0	max
ano	ne	F	LRECL	LRECL
		FB	LRECL	LRECL * [max/LRECL]
		V	LRECL	LRECL+4
		U	0	LRECL
ne	ano	F, FB	BLKSIZE	BLKSIZE
		V, VB	min {1028, BLKSIZE-4}	BLKSIZE
		U	0	BLKSIZE

Tabulka 11: Předdefinované hodnoty LRECL a BLKSIZE

4.8. Příklady programů v C/C++

4.8.1. Hello world

Na úplný závěr si ukažme, jak vypadají některé jednoduché programy napsané v jazyce C/C++ a úlohy pro jejich zpracování.

Jak bývá zvykem při práci s novými nebo neznámými programovacími jazyky, začneme příkladem programu, který pouze dokáže napsat „Hello world.“ Zdrojový kód je umístěný v data setu s názvem CTMMSTR.INTRO.C(HELLO), úlohy pro zpracování jsou dva a nalézají se v data setu CTMMSTR.INTRO.C ve členech HELLOCMP a HELLOBND.

Samotný zdrojový kód je jednoduchý, není třeba ho nijak komentovat. Zajímavější jsou úlohy pro jeho zpracování. Úloha s názvem HELLOCMP slouží pouze k překladu programu. Používá k tomu proceduru CCNDRVR. Výstup se ukládá do data setu CTMMSTR.INTRO.OBJ(HELLO). Všimněme si ještě parametru REGION u příkazu JOB – místo obvyklé hodnoty 4096K zde píšeme 48M. Tuto hodnotu používá i druhá úloha HELLOBND. Jejím účelem je poslat výstup minulé úlohy sestavovacímu programu, k čemuž využívá procedury IEWL.

Úloha pro překlad:

```
EDIT       CTMMSTR.INTRO.JCL(HELLOCMP) - 01.14
****** ***************************** Top of Data *************
000100 //CTMMSTRA JOB  (UNIVER),'CTMMSTR',CLASS=A,REGION=48M,
000200 // MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
000210 //*JCLLIB ORDER=(CEE.SCEEPROC,CBC.SCBCPRC)
000220 //COMPILE EXEC PGM=CCNDRVR,
000230 // PARM='/SEARCH(''CEE.SCEEH.+'') NOOPT SO OBJ LIST'
000231 //STEPLIB DD DSNAME=CEE.SCEERUN,DISP=SHR
000232 //        DD DSNAME=CEE.SCEERUN2,DISP=SHR
000233 //        DD DSNAME=CBC.SCCNCMP,DISP=SHR
000234 //SYSLIN  DD DSNAME=CTMMSTR.INTRO.OBJ(HELLO),DISP=SHR
000235 //SYSPRINT DD SYSOUT=*
000236 //SYSIN    DD DSNAME=CTMMSTR.INTRO.C(HELLO),DISP=SHR
****** **************************** Bottom of Data ***********
```

Úloha pro spojování:

```
EDIT       CTMMSTR.INTRO.JCL(HELLOBND) - 01.13
****** ***************************** Top of Data *************
000100 //CTMMSTRA JOB  (UNIVER),'CTMMSTR',CLASS=A,REGION=48M,
000200 // MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
000220 //BIND EXEC PGM=IEWL,PARM='OPTIONS=OPTS'
000230 //OPTS DD *
000240 AMODE=31
000250 /*
000260 //SYSLIB  DD DISP=SHR,DSN=CEE.SCEELKEX
000270 //        DD DISP=SHR,DSN=CEE.SCEELKED
000280 //        DD DISP=SHR,DSN=CEE.SCEECPP
000290 //SYSLIN  DD DISP=SHR,DSN=CTMMSTR.INTRO.OBJ(HELLO)
000300 //SYSLMOD DD DISP=SHR,DSN=CTMMSTR.INTRO.LOAD(HELLO)
000400 //SYSPRINT DD SYSOUT=*
****** **************************** Bottom of Data ***********
```

Vlastní program:

```
EDIT       CTMMSTR.INTRO.C(HELLO) - 01.00
****** ***************************** Top of Data *************
```

```
000100 #include <stdio.h>
000200 int main( int argc, char* argv[] )
000300 {
000310    printf( "Hello world." );
000320    return 1;
000400 }
****** **************************** Bottom of Data ************
```

4.8.2. Jednoduchý příklad v C s hlavičkovým souborem

Druhým příkladem je krátký program napsaný v jazyce C. Úloha i zdrojový kód jsou umístěny ve členech jednoho data setu jménem CTM0001.TEST.C. V úloze používám předdefinovanou proceduru EDCCBG, která zajistí překlad, spojení i spuštění programu. Program sám pak používá jeden hlavičkový soubor, uložený v samostatném členu. Účelem programu je převod Celsiovy teploty na teplotu Fahrenheitovu. Hodnota pro převod je zadaná na řádku 000400 v úloze.

Úloha pro zpracování:

```
EDIT       CTM0001.TEST.C(SPUSTIT) - 01.06
****** ***************************** Top of Data ***************
000010 //CTM0001A JOB (UNIVER),'CTM0001',CLASS=A,REGION=4096K,
000020 //         MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
000030 //MYLIB JCLLIB ORDER=('CEE.SCEEPROC','CBC.SCBCPRC')
000100 //DOCLG EXEC PROC=EDCCBG,INFILE='CTM0001.TEST.C(CTOF)',
000200 // CPARM='LSEARCH(''''CTM0001.TESTHDR.+'''')'
000300 //GO.SYSIN DD DATA,DLM=@@
000400 19
000500 @@
****** **************************** Bottom of Data *************
```

Vlastní program:

```
EDIT       CTM0001.TEST.C(CTOF) - 01.04
****** ***************************** Top of Data ***********************
000100 #include <stdio.h>
000600 #include "ccnuaan.h"
000700 void convert(double);
000800 int main (int argc, char **argv)
000900 {
001000     double c_temp;
001100     if (argc ==1) {
001200        printf("Zadejte Celsiovu teplotu: \n");
001300        if (scanf("%f", &c_temp) != 1) {
001400            printf("Zadejte to spravne\n");
001500            }
001600             else {
001700                 convert(c_temp);
001800              }
001900            }
002000        else {
002100           int i;
002200           for (i = 1; i < argc; ++i) {
002300              if (sscanf(argv[i], "%f", &c_temp) != 1)
002400                 printf("%s neni spravne\n",argv[i]);
002500              else
002600                 convert(c_temp);
002700           }
002800        }
002900        return 0;
```

```
003000 }
003100 void convert(double c_temp) {
003200    double f_temp = (c_temp * CONV + OFFSET);
003300    printf("%5.2f Celsius je %5.2f Fahrenheit\n",c_temp, f_temp);
003400 }
****** **************************** Bottom of Data ********************
```

Hlavičkový soubor:

```
EDIT       CTM0001.TESTHDR.H(CCNUAAN) - 01.01
****** ***************************** Top of Data ***********************
000100 /*************************************************************
000200 * User include file: ccnuaan.h *
000300 *************************************************************/
000400
000500 #define CONV (9./5.)
000600 #define OFFSET 32
****** **************************** Bottom of Data ********************
```

4.8.3. Program pro výpis argumentů

Následující program necháme čtenáře analyzovat samotného, neboť na něm není nic zvláštního. V data setu CTMMSTR.INTRO.C(ARGS) se nalézá zdrojový kód, úloha z data setu CTMMSTR.INTRO.JCL(ARGS) program přeloží a spojí a ta z CTMMSTR.INTRO.JCL(ARGSJOB) ho spustí.

Úloha pro překlad a spojení:

```
EDIT       CTMMSTR.INTRO.JCL(ARGS) - 01.17
****** ***************************** Top of Data **************
000100 //CTMMSTRA JOB  (UNIVER),'CTMMSTR',CLASS=A,REGION=48M,
000200 // MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
000210 //MYLIB JCLLIB ORDER=('CEE.SCEEPROC','CBC.SCBCPRC')
000220 //COMPPRC EXEC PROC=EDCCB,
000230 // CPARM='SO LIST',
000240 // INFILE='CTMMSTR.INTRO.C(ARGS)',
000250 // OUTFILE='CTMMSTR.INTRO.LOAD(ARGS),DISP=SHR'
****** **************************** Bottom of Data ************
```

Úloha pro spuštění:

```
EDIT       CTMMSTR.INTRO.JCL(ARGSJOB) - 01.15
****** ***************************** Top of Data ***************
000100 //CTMMSTRA JOB  (UNIVER),'CTMMSTR',CLASS=A,REGION=4096K,
000200 // MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
000210 //JOBLIB DD DSN=CTMMSTR.INTRO.LOAD,DISP=SHR
000300 //MYARGS EXEC PGM=ARGS,
000310 //       PARM='ARG1 ARG2 ARG3 ARG4'
000600 //SYSOUT DD SYSOUT=*
000900 /*
****** **************************** Bottom of Data *************
```

Zdrojový kód:

```
EDIT       CTMMSTR.INTRO.C(ARGS) - 01.05
****** ***************************** Top of Data **************
000100 #include <stdio.h>
000200 int main( int argc, char* argv[] )
```

```
000300 {
000301    int i;
000302    for( i = 0; i < argc; i ++ )
000310       printf( "Argument %d -> %s\n", i, argv[i]);
000320    return 1;
000400 }
****** **************************** Bottom of Data ************
```

4.8.4. Program pro práci s data sety KSDS

Na závěr bych zde rád napsal několik komentářů k funkcím z programu, který se nalézá v příloze 2. Tento program slouží pro ilustraci práce s data sety VSAM (kapitola 2.5.) typu KSDS. Vzhledem k tomu, že v tomto programu nejde až tak o vlastní programování v C/C++, rozhodl jsem se ho umístit do přílohy a ne sem. Jelikož je poněkud delší, nebudu ho jako celek popisovat, ale pouze zmíním význam funkcí, které se v něm používají.

První z nich je funkce flocate(). Slouží k nalezení konkrétního záznamu v data setu VSAM. Vrací primární klíč nebo relativní bajtovou adresu nebo číslo záznamu, v závislosti na tom, v jakém typu data setu VSAM se pohybujeme. V parametrech udáváme postupně kde hledat, co hledat, velikost klíčových slov a přístupový směr ke klíči. Poslední parametr má zvláštní sadu hodnot, typ ostatních určíme podle okolností.

Další používanou funkcí je fwrite(), která se používá pro zapisování nových záznamů. Za určitých okolností vrací počet úspěšně zapsaných bajtů. Do parametrů zapisujeme postupně co chceme uložit, velikost, velikost záznamu a kam chceme ukládat.

Funkce fread() slouží pro čtení záznamů a má opět čtyři parametry – co chceme číst, velikost, velikost, velikost záznamů a odkud chceme číst. Vrací (za jistých okolností) počet úspěšně přečtených bajtů.

Poslední funkcí, kterou bych chtěl zmínit, je funkce fdelrec(). Jejím úkolem je mazání záznamů. Nevrací nic a parametr má jeden –ukazatel na mazaný záznam.

Uznávám, že popis funkcí nebyl vyčerpávající, ale to ani není smyslem této práce. Zájemci o jejich praktické použití ho naleznou v příloze 2 a ti, kteří se zajímají o jejich teoretické možnosti, se mohou podívat do publikace z/OS C/C++ Run-Time Library Reference, kterou doporučuje použitá literatura.

5. Programovací jazyk REXX

5.1. Úvod

5.1.1. Historie

Něco málo o programovacím jazyce REXX (REstructured eXtended eXecutor, používá se též Rexx) jsme si již řekli v kapitolce o programovacích jazycích na mainframech. Připomeňme si, že REXX byl vyvinut v letech 1979 až 1982 v IBM. Standard REXXu byl vydán až v roce 1996. Původně to měl být skriptovací jazyk pro libovolný systém (jako např. dnešní Python). Má mnoho verzí, od těch pro mainframy až po varianty pro Windows, Amigy, Linux či Solaris. Od poloviny devadesátých let dvacátého století se objevily dvě verze tohoto jazyka – NetRexx a ObjectRexx. NetRexx silně spolupracuje s Javou a dokonce požívá javovské knihovny. Z tohoto důvodu není se starším „klasickým" REXXem kompatibilní. Naopak ObjectRexx je vlastně objektově orientovanou verzí starších REXXů.

Na tomto místě se budeme věnovat implementaci REXXu fungujícího v rozhraní TSO/E na mainframech.

5.1.2. Charakteristika

Hlavními rysy REXXu jsou především:

- intuitivnost – jako příkazy používá běžná anglická slova (SAY, DO, END …)
- volnost formátování – příkazy není nutné psát do určitých sloupců, lze mezi ně vkládat mezery, lze je psát malými či velkými písmeny apod.
- mnoho zabudovaných funkcí
- existence chybových hlášek při překladu
- interpretační jazyk – před spuštěním není třeba programy překládat
- pokročilá analýza vstupů – REXX dovede rozlišovat na vstupu znaky a čísla a oddělit je
- nedeklarují se proměnné

Programovací jazyk REXX sestává z instrukcí, zabudovaných funkcí, vnějších funkcí TSO/E a funkcí pracujících s daty uloženými na zásobníku. Instrukcemi mohou být klíčová slova, přiřazení, labely, null a příkazy.

Jak už bylo řečeno, programy napsané v REXXu není třeba překládat. Nicméně pokud program necháme projít překladačem, získáme mnoho výhod plynoucích z optimalizace kódu. Mezi ně patří třeba zlepšení výkonnosti programu, snížení zátěže systému, ochrana kódu a další.

Programy napsané v programovacím jazyce REXX se nazývají exeky (anglicky execs). Zdrojové kódy exeků zapisujeme standardně do data setů. Každý data set obsahující nějaký exek by měl podle doporučení IBM začínat řádkem s následujícím komentářem:

```
/*************** REXX **************/
```

Tento komentář se uvádí proto, aby se rexxovské exeky odlišily od CLISTů.

5.1.3. Spouštění exeků

Pokud si přejeme spustit námi napsaný exek bez překládání, lze to udělat pomocí příkazu TSO/E EXEC napsaného buď za hlášku READY v původním režimu TSO nebo za COMMAND = = = > v ISPF následujícím způsobem:

`EXEC 'jmeno.data.setu(clen)'` (původní režim TSO)
`TSO EXEC 'jmeno.data.setu(clen)'` (libovolný panel ISPF)

Tento systém spouštění se nazývá explicitní. Exeky lze spouštět také implicitně pouze pomocí zadání jejich jména. K tomu je ovšem nejprve třeba zaregistrovat náš exek mezi systémové procedury, což už je poněkud složitější a lze to dohledat v literatuře.

Pokud exek obsahuje nějakou chybu, objeví se při jeho běhu chybové hlášení. Skládá se ze dvou částí – v první se objeví číslo řádku s chybou, tři plusy a obsah instrukce, ve druhé pak pojmenování chyby.

5.2. Instrukce

5.2.1. Formátování instrukcí

Jak již bylo řečeno, jednou z charakteristik REXXu je volné formátování instrukcí.

Instrukce můžeme psát malými písmeny, velkými písmeny nebo kombinací obojího. Pokud ovšem znaky neuzavřeme mezi jednoduché či dvojité uvozovky, změní se na velká písmena. Série znaků uzavřená do uvozovek se nazývá řetězec literálů (literal string). Je jedno, zda použijeme jednoduché nebo dvojité uvozovky, musíme však napsat na začátek i konec řetězce stejné. Pokud nenapíšeme žádné, zobrazí se požadovaný text přepsaný do velkých písmen. Přehledně to shrnuje tabulka 12:

Zadaný řetězec	Výsledek
SAY 'Zdravi Vas REXX.' SAY "Zdravi Vas REXX."	Zdravi Vas REXX.
SAY "Zdravi Vas REXX.'	Chyba, různé uvozovky
SAY Zdravi Vas REXX.	ZDRAVI VAS REXX.

Tabulka 12: Vliv uvozovek na řetězce

Pokud řetězec náhodou obsahuje apostrof, lze to vyřešit dvěma způsoby. Buď řetězec uzavřeme do dvojitých uvozovek nebo napíšeme apostrof dvakrát.

Další vlastností programů psaných v REXXu je, že příkazy nemusí začínat v nějakém určitém sloupci a mezi příkazy můžeme psát libovolný počet volných mezer i řádků. Zkrátka mezera zde není nositelem informace.

Čárka na konci řádku s instrukcí znamená, že instrukce pokračuje na následujícím řádku. V případě řetězce pokračujícího mezi řádky REXX automaticky vkládá mezi jeho části mezeru. Pokud tomu chceme zabránit, neukončujeme horní řádek čárkou, ale dvěma svislými pruhy || .

Instrukce se běžně ukončuje s koncem řádku. Pokud si přejeme ukončit instrukci dříve (více instrukcí na řádek), uděláme to pomocí středníku.

5.2.2. Typy instrukcí

Instrukce může být pěti různých typů.

Prvním typem jsou klíčová slova. Stejně jako u jiných programovacích jazyků říkají, co se má udělat. Patří mezi ně například zmiňované SAY pro výpis řetězce na obrazovku.

Druhý typ instrukce je přiřazení. V exeku je realizováno pomocí jednoho symbolu rovnítka. Přiřazuje nějakému řetězci hodnotu.

Třetí druh jsou tzv. labely. Formálně to jsou řetězce následované dvojtečkou. Označují část exeku, na kterou se lze pak odvolávat při psaní funkcí a podprogramů. Label by neměl mít více než osm znaků, protože jinak to může vést k nepředpokladatelným důsledkům.

Čtvrtý typ instrukce se nazývá null. Null instrukce znamená prázdný řádek nebo komentář. Nijak se tedy neprojevuje na výstupu programu. Komentáře se uzavírají mezi /* a */. Jak již bylo řečeno, na první řádek každého exeku by se měl uvádět komentář obsahující hvězdičky a slovo REXX. Tento komentář se nazývá identifikátor exeku REXX (REXX exec identifier) a je nutné ho uvádět, aby nedošlo k záměně REXXových exeků a CLISTů.

Konečně pátý typ instrukcí představují příkazy zpracovávané vnějším prostředím.

5.3. Proměnné a operátory

5.3.1. Označování proměnných

Proměnnou rozumíme znak nebo skupinu znaků s přiřazenou hodnotou. Jak už bylo řečeno, u proměnných v REXXu se nedeklaruje jejich typ. Proměnná se vytvoří prvním přiřazením hodnoty. Názvy proměnných mohou sestávat z velkých a malých písmen, čísel a zvláštních znaků @ # $ ¢ ? ! . _ . Tyto se kombinují libovolně, pouze s omezeními:

- první znak nesmí být číslo nebo tečka
- délka názvu nesmí převýšit 250 bajtů
- neměly bychom používat názvy RC, SIGL a RESULT, které REXX používá pro zvláštní účely

5.3.2. Hodnoty proměnných

Proměnným můžeme přiřazovat jako hodnotu celá čísla (se znaménkem i bez), desetinná čísla, čísla s plovoucí desetinou čárkou ve tvaru 1.27E10, textové řetězce (v uvozovkách i bez), hodnoty jiných proměnných nebo hodnoty výrazů (např. a = 3 + 78).

Celá čísla mohou mít nejvýše devět číslic. Tento stav lze upravit instrukcí NUMERIC DIGITS. Desetinná čísla mohou mít také nejvýše devět číslic dohromady před a za desetinnou čárkou. Výjimku tvoří čísla mezi nulou a jedničkou, kde se nula před desetinnou čárkou do součtu nepočítá.

5.3.3. Aritmetické operátory

Přehled aritmetických operátorů nalezneme v následující tabulce 13. Porovnávací operátory vrací nulu (false) nebo jedničku (true).

Symbol	Význam	Symbol	Význam
+	Sčítání	= =	Zcela identické
–	Odčítání	=	Identické
*	Násobení	\ = =	Ne zcela identické
/	Dělení	\ = nebo ><	Neidentické
%	Dělení se zbytkem (vrátí podíl)	> resp. <	Větší resp. menší
//	Dělení se zbytkem (vrátí zbytek)	>=	Větší nebo rovno
**	Umocnění na celé číslo (2 ** 3 = 8)	\ >	Ne větší

Tabulka 13: Aritmetické operátory

Analogicky se vytvářejí symboly „menší nebo rovno“ a „ne menší“. Rozdíl mezi symboly „zcela identické“ a „identické“ je následující. Aby byly dva výrazy zcela identické, musí mít všechno včetně mezer a velikosti písmen naprosto stejné. Naproti tomu identické

jsou takové dva výrazy, které se vyhodnotí stejně. Proto např. porovnání 4E2 = 400 vrátí true, ale 4E2 = = 400 vrátí false. Stejně tak Ahoj = ‘ ahoj‘ je true, ale totéž se dvěma rovnítky false.

5.3.4. Logické operátory

Pokud budeme chtít tvořit složitější podmínky, budou k tomu potřeba logické operátory. Ty jsou celkem čtyři. Symbol & značí konjunkci (pokud platí podmínky vlevo i vpravo, tak true, jinak false). Symbol | označuje disjunkci (pokud platí aspoň jedna z podmínek, tak true, jinak false). Symbolem && popisujeme, že pokud platí právě jedna z podmínek, chceme true, jinak false. Pokud před podmínku přidáme zpětné lomítko, vyhodnotí se podmínka pravdivostně opačně. Tedy např. \ (7 < 4) je true.

5.4. Klíčová slova

Přehled základních klíčových slov obsahuje následující tabulka 14:

Klíčové slovo	Význam
SAY řetězec	Vypíše řetězec na obrazovku
PULL proměnná1 proměnná2	Do proměnné se uloží vstup zadaný z terminálu uživatelem, proměnných lze takto načíst i víc
ARG proměnná1 proměnná2	Do proměnné se uloží hodnota zadaná při spuštění exeku mezi jeho jméno a slovo exec do uvozovek např.: EXEC ‘ABC.REXX(POKUS)‘ ‘42‘ exec nastaví hodnotu proměnné1 uvedené v exeku za slovem ARG na 42
PARSE	Při přidání před PULL nebo ARG zabrání, aby se načtený vstup převedl na velká písmena
TRACE I	Při spuštění exeku vypisuje hodnoty parametrů
IF podmínka THEN instrukce ELSE instrukce	Obvyklá podmínková konstrukce
SELECT WHEN podm1 THEN instrukce WHEN podm2 THEN instrukce … OTHERWISE instrukce END	Výběr z více alternativ
DO počet instrukce END	Cyklus s pevným počtem opakování počet, viz dále
EXIT	Ukončí exek
LEAVE	Ukončí cyklus a pokračuje další instrukcí po END
ITERATE	Ukončí aktuální běh cyklu a začne nový běh od DO
DO WHILE podmínka instrukce END	Cyklus, který probíhá, dokud podmínka platí, nemusí proběhnout vůbec
DO UNTIL podmínka instrukce END	Cyklus, který probíhá, dokud podmínka neplatí, proběhne vždy aspoň jednou

CALL/RETURN	Volání/vracení k vnějšímu či vnitřnímu podprogramu
SIGNAL label	Přeskočí na instrukce za daným labelem
PROCEDURE	Odstíní lokální proměnné

Tabulka 14: Základní klíčová slova

Některá klíčová slova si zaslouží zvláštní komentář.

Pokud pomocí klíčového slova ARG přeurčíme nebo nedourčíme parametry exeku (při spuštění zadáme moc nebo málo parametrů), může se stát chyba. Přesnější popis nalezneme v literatuře.

Popis přesného chování TRACE je v literatuře.

Pokud v konstrukci IF/THEN/ELSE nechceme za ELSE psát žádnou instrukci, napíšeme tam slovo NOP. Není tomu tedy tak, že by se ELSE nepsalo jako v jiných programovacích jazycích. Chceme-li naopak psát více instrukcí, vložíme je mezi DO a END. Vidět to můžeme na příkladu:

```
IF cas = konec THEN
   DO
      SAY ‘Vypni pocitac.‘
      SAY ‘Jdi domu.‘
   END
ELSE NOP
```

Počet opakování v cyklu DO/END může být:

- celé číslo
- proměnná s přiřazenou celočíselnou proměnnou
- posloupnost celých čísel, pak máme tvar např. DO i = 1 TO 10; instrukce; END
- FOREVER, tedy cyklus se bude opakovat nekonečněkrát, ukončit ho můžeme zadáním klíčového slova EXIT nebo LEAVE

Ve všech cyklech můžeme zadat mezi klíčová slova jednu nebo více instrukcí.

Klíčová slova CALL, RETURN a PROCEDURE souvisí s psaním funkcí a podprogramů, které bude popsáno v následujících kapitolách.

5.5. Funkce a podprogramy

5.5.1. Co jsou funkce a podprogramy

REXX používá tři typy funkcí. Jsou to zabudované funkce, funkce napsané uživatelem (vnitřní – v rámci exeku, vnější – mimo exek) a balíky funkcí. Libovolnou z nich voláme tak, že napíšeme její jméno a do závorek za něj požadované argumenty funkce. Mezi jménem funkce a levou závorkou nesmí být mezera. Argumentů smí být nejvýše dvacet a oddělují se čárkami. Funkce musí vracet hodnotu a ta se symbolicky objeví v místě volání funkce.

Kromě funkcí také můžeme používat podprogramy (subroutines). Stejně jako funkce jsou i podprogramy vnitřní (v rámci daného exeku) a vnější (mimo). Funkce a podprogramy se liší ve dvou hlavních aspektech – v jejich volání a ve vracení údajů. Podprogram voláme příkazem CALL a následuje jeho jméno, mezera a parametry oddělené čárkami. Podprogram nemusí vracet hodnotu, pokud ji vrací, tak pomocí příkazu RETURN. Do volajícího exeku se pak objeví hodnota ve speciální proměnné RESULT.

5.5.2. Zabudované funkce

V následující tabulce 15 nalezneme příklady několika základních zabudovaných funkcí. Celkem jich je přes padesát a úplný přehled lze nalézt v literatuře.

Funkce	Popis
ABS	Vrací absolutní hodnotu čísla
MIN/MAX	Vrací nejmenší/největší číslo ze zadaných argumentů
RANDOM	Vrací pseudonáhodné číslo ze zadaného rozsahu
TRUNC	Vrací celou část desetinného čísla, volitelně určitý počet desetinných míst
COMPARE	Vrací nulu pokud jsou dva řetězce stejné, jinak vrací číslo prvního znaku, ve kterém se liší
a2b	a, b = B (binární), C (znaky), H (hexadecimální), D (desetinné) převede daný typ řetězce na jiný
WORD	Vrátí slovo s požadovaným číslem ze řetězce
LENGHT	Vrátí délku řetězce
DATE	Vrátí datum ve formátu dd mm rrrr
TIME	Vrátí čas ve formátu hh:mm:ss

Tabulka 15: Příklady zabudovaných funkcí

5.5.3. Jak psát vlastní podprogramy

Vlastní vnitřní podprogramy se píší za hlavní část exeku. Začínají labelem. Aby exek poznal, kde končí hlavní část a začínají podprogramy, je nutné na konec hlavní části napsat příkaz EXIT.

Zavolané vnitřní podprogramy pracují a mění proměnné hlavní části exeku. Navíc hlavní část exeku může volně pracovat s „lokálními“ proměnnými vnitřního podprogramu. To může být někdy problém. Proto lze použít příkaz PROCEDURE umístěný hned za label daného podprogramu. Tento příkaz odstíní vlastní proměnné podprogramu od hlavní části exeku, takže jsou potom skutečně lokální. Pokud takto chceme ochránit všechny proměnné podprogramu kromě některých speciálních, napíšeme toto:

```
Label: PROCEDURE EXPOSE neodstíněné proměnné
```

Vnější podprogramy se zpravidla ukládají do stejného data setu PDS jako exek do samostatného členu, který se jmenuje jako ony.

5.5.4. Jak psát vlastní funkce

Vlastní funkce píšeme naprosto stejně jako podprogramy, jen je vždy musíme ukončit příkazem RETURN, aby vrátily nějakou hodnotu. Naprosto stejně také pracují s proměnnými.

5.6. Pokročilejší datové struktury

5.6.1. Složené proměnné

Stejně jako běžné programovací jazyky umí i REXX vytvářet obdobu struktury jednorozměrného pole či záznamu. V REXXu se toto nazývá složená proměnná. Tvoří se tak, že spojíme dohromady více jednoduchých proměnných. Nejsnáze to bude vidět z následujících dvou příkladů:

Záznam:

```
jmeno = ‘Petr‘
prijmeni = ‘Novak‘
student = jmeno.prijmeni
SAY student.prijmeni  /* zobrazi student.Novak */
```

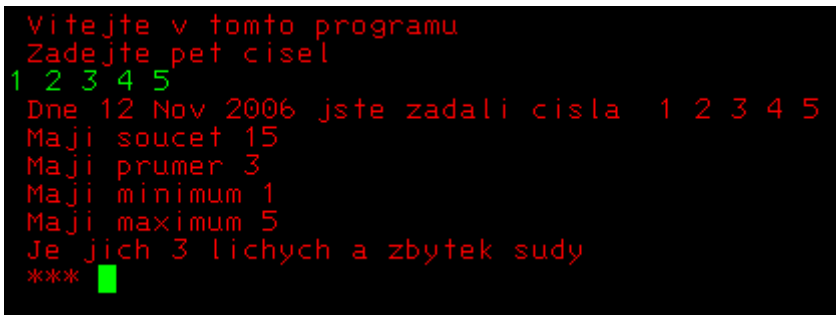
Pole:

```
DO i = 1 TO 6
   SAY ‘Zadejte jmeno.‘
   PARSE PULL student.i   /* pro studenta 4 zadame ‘Jan Novy‘ */
END
SAY student.4   /* zobrazi Jan Novy */
```

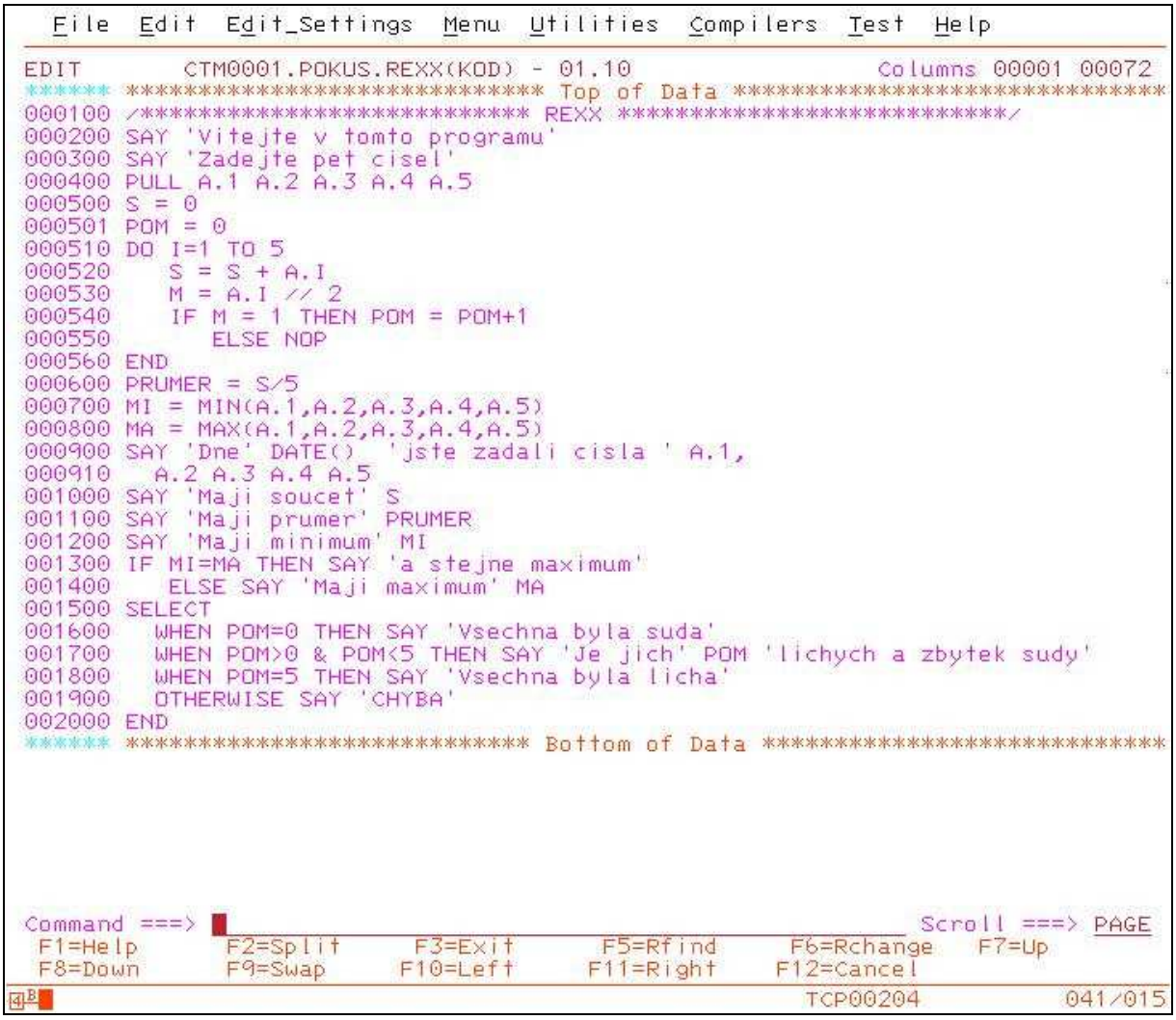
Pokud si přejeme inicializovat všechny položky případné složené proměnné na nějakou hodnotu, uděláme to tak, že přiřadíme tuto hodnotu tzv. kmeni proměnné (anglicky stam). Kmen proměnné z minulého příkladu by byl `student.` s tečkou na konci. Pokud bychom tedy provedli přiřazení `student. = prazdne` , tak by se po zadání `student.cokoli` vždy volala hodnota `prazdne`.

5.7. Příklad

Jednoduchý příklad s výsledkem můžete vidět na následujících dvou obrázcích:



Obrázek 20: Výsledek programu, zeleně napsaná čísla zadával uživatel



Obrázek 21: Zdrojový kód

Příloha 1:

Vztahuje se ke kapitole 2.5. s použitím JCL z kapitoly 3..

```
EDIT       CTM0001.VSAM.JCL(CCNGVS3) - 01.12
****** ***************************** Top of Data ***************************
000001 //CTM0001A JOB  (UNIVER),'CTM0001',CLASS=A,REGION=4096K,
000002 // MSGLEVEL=(1,1),MSGCLASS=H,NOTIFY=&SYSUID
000003 //JOBLIB  DD DSN=CTM0001.VSAM.LOAD,DISP=SHR
000004 //         DD DSN=CEE.SCEERUN,DISP=SHR
000005 //*--------------------------------------------------------------------
000006 //* Delete cluster, and AIX and PATH
000007 //*--------------------------------------------------------------------
000008 //DELETEC EXEC PGM=IDCAMS
000009 //SYSPRINT  DD SYSOUT=*
000010 //SYSIN     DD *
000011    DELETE -
000012      CTM0001.KSDS.CLUSTER -
000013      CLUSTER -
000014      PURGE -
000015      ERASE
000016 /*
000017 //*--------------------------------------------------------------------
000018 //* Define KSDS
000019 //*--------------------------------------------------------------------
000020 //*        FILE(VOLUME) -
000021 //*        VOL(XXXXXX) -
000022 //DEFINE  EXEC PGM=IDCAMS
000023 //*VOLUME    DD UNIT=POOL,DISP=SHR,VOL=SER=(XXXXXX)
000024 //SYSPRINT  DD SYSOUT=*
000025 //SYSIN     DD *
000026     DEFINE CLUSTER -
000027         (NAME(CTM0001.KSDS.CLUSTER) -
000028         TRK(4 4)  -
000029         RECSZ(69 100) -
000030         INDEXED -
000031         NOREUSE -
000032         KEYS(4 0) -
000033         OWNER(CTM0001) ) -
000034       DATA -
000035         (NAME(CTM0001.KSDS.DA)) -
000036       INDEX -
000037         (NAME(CTM0001.KSDS.IX))
000038 /*
000039 //*--------------------------------------------------------------------
000040 //* Repro data into KSDS
000041 //*--------------------------------------------------------------------
000042 //REPRO   EXEC PGM=IDCAMS
000043 //SYSPRINT  DD SYSOUT=*
000044 //SYSIN     DD *
000045    REPRO INDATASET(CTM0001.VSAM.VSAMDATA) -
000046       OUTDATASET(CTM0001.KSDS.CLUSTER)
000047 /*
000048 //*--------------------------------------------------------------------
000049 //* Define unique AIX, define and build PATH
000050 //*--------------------------------------------------------------------
000051 //DEFAIX  EXEC PGM=IDCAMS
000052 //SYSPRINT  DD SYSOUT=*
000053 //SYSIN     DD *
000054     DEFINE AIX -
000055         (NAME(CTM0001.KSDS.UAIX) -
000056         RECORDS(25)  -
000057         KEYS(8,4)    -
000058         UNIQUEKEY -
000059         RELATE(CTM0001.KSDS.CLUSTER)) -
```

```
000060       DATA -
000061         (NAME(CTM0001.KSDS.UAIXDA)) -
000062       INDEX -
000063         (NAME(CTM0001.KSDS.UAIXIX))
000064     DEFINE PATH -
000065         (NAME(CTM0001.KSDS.UPATH) -
000066         PATHENTRY(CTM0001.KSDS.UAIX))
000067     BLDINDEX -
000068         INDATASET(CTM0001.KSDS.CLUSTER) -
000069         OUTDATASET(CTM0001.KSDS.UAIX)
000070 /*
000071 //*-------------------------------------------------------------
000072 //* Define nonunique AIX, define and build PATH
000073 //*-------------------------------------------------------------
000074 //DEFAIX  EXEC PGM=IDCAMS
000075 //SYSPRINT  DD SYSOUT=*
000076 //SYSIN     DD *
000077     DEFINE AIX -
000078         (NAME(CTM0001.KSDS.NUAIX) -
000079         RECORDS(25)  -
000080         KEYS(20, 12)    -
000081        NONUNIQUEKEY -
000082         RELATE(CTM0001.KSDS.CLUSTER)) -
000083       DATA -
000084         (NAME(CTM0001.KSDS.NUAIXDA)) -
000085       INDEX -
000086         (NAME(CTM0001.KSDS.NUAIXIX))
000087     DEFINE PATH -
000088         (NAME(CTM0001.KSDS.NUPATH) -
000089         PATHENTRY(CTM0001.KSDS.NUAIX))
000090     BLDINDEX -
000091         INDATASET(CTM0001.KSDS.CLUSTER) -
000092         OUTDATASET(CTM0001.KSDS.NUAIX)
000093 /*
000094 //*-------------------------------------------------------------
000095 //* Run the testcase
000096 //*-------------------------------------------------------------
000097 //GO        EXEC PGM=CCNGVS2,REGION=5M
000098 //SYSPRINT  DD SYSOUT=*
000099 //SYSTERM   DD SYSOUT=*
000100 //SYSOUT    DD SYSOUT=*
000101 //PLIDUMP   DD SYSOUT=*
000102 //SYSABEND  DD SYSOUT=*
000103 //SYSUDUMP  DD SYSOUT=*
000104 //CLUSTER   DD DSN='CTM0001.KSDS.CLUSTER',DISP=SHR
000105 //AIXUNIQ   DD DSN='CTM0001.KSDS.UPATH',DISP=SHR
000106 //AIXNUNIQ  DD DSN='CTM0001.KSDS.NUPATH',DISP=SHR
000107 /*
000108 //*-------------------------------------------------------------
000109 //* Print out the cluster
000110 //*-------------------------------------------------------------
000111 //PRINTF    EXEC PGM=IDCAMS
000112 //SYSPRINT   DD SYSOUT=*
000113 //SYSIN      DD *
000114    PRINT -
000115        INDATASET(CTM0001.KSDS.CLUSTER) CHAR
000116 /*
****** **************************** Bottom of Data **********************
```

Příloha 2:

Vztahuje se ke kapitole 2.5. s použitím C/C++ z kapitoly 4, komentář v podkapitole 4.8.4..

```
EDIT       CTM0001.VSAM.C(CCNGVS2) - 01.01
****** ***************************** Top of Data ******************************
000001 /* this example demonstrates the use of a KSDS file */
000002 /* part 1 of 2-other file is CCNGVS3 */
000003
000004 #include <stdio.h>
000005 #include <string.h>
000006
000007 /* global definitions
000008
000009 struct data_struct {
000010               char   emp_number[4];
000011               char   user_id[8];
000012               char   name[20];
000013               char   pers_info[37];
000014 };
000015
000016 #define  REC_SIZE                69
000017 #define  CLUS_KEY_SIZE            4
000018 #define  AIX_UNIQUE_KEY_SIZE      8
000019 #define  AIX_NONUNIQUE_KEY_SIZE  20
000020
000021 static void print_amrc() {
000022     __amrc_type currErr = *__amrc; /* copy contents of __amrc     */
000023                                    /* structure so that values    */
000024                                    /* don't get jumbled by printf */
000025     printf("R15 value   = %d\n", currErr.__code.__feedback.__rc);
000026     printf("Reason code = %d\n", currErr.__code.__feedback.__fdbk);
000027     printf("RBA         = %d\n", currErr.__RBA);
000028     printf("Last op     = %d\n", currErr.__last_op);
000029     return;
000030 }
000031 /* update_emp_rec() function definition
000032
000033 int update_emp_rec (struct data_struct *data_ptr,
000034                     struct data_struct *orig_data_ptr,
000035                     FILE   *fp)
000036 {
000037     int          rc;
000038     char         buffer[REC_SIZE+1];
000039
000040     /*  Check to see if update will change primary key (emp_number)   */
000041     if (memcmp(data_ptr->emp_number,orig_data_ptr->emp_number,4) != 0) {
000042        /* Check to see if changed primary key exists                    */
000043        rc = flocate(fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,__KEY_EQ);
000044        if (rc == 0) {
000045           print_amrc();
000046           printf("Error: new employee number already exists\n");
000047           return 10;
000048        }
000049
000050        clearerr(fp);
000051
000052        /* Write out new record                                          */
000053        rc = fwrite(data_ptr,1,REC_SIZE,fp);
000054        if (rc != REC_SIZE || ferror(fp)) {
000055           print_amrc();
000056           printf("Error: write with new employee number failed\n");
000057           return 20;
000058        }
000059
```

```
000060        /* Locate to old employee record so it can be deleted          */
000061        rc = flocate(fp,&(orig_data_ptr->emp_number),CLUS_KEY_SIZE,
000062                     __KEY_EQ);
000063        if (rc != 0) {
000064           print_amrc();
000065           printf("Error: flocate to original employee number failed\n");
000066           return 30;
000067        }
000068
000069        rc = fread(buffer,1,REC_SIZE,fp);
000070        if (rc != REC_SIZE || ferror(fp)) {
000071           print_amrc();
000072           printf("Error: reading old employee record failed\n");
000073           return 40;
000074        }
000075
000076        rc = fdelrec(fp);
000077        if (rc != 0) {
000078           print_amrc();
000079           printf("Error: deleting old employee record failed\n");
000080           return 50;
000081        }
000082
000083     } /* end of checking for change in primary key                */
000084      else { /* Locate to current employee record                  */
000085        rc = flocate(fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,__KEY_EQ);
000086        if (rc == 0) {
000087           /* record exists, so update it                           */
000088           rc = fread(buffer,1,REC_SIZE,fp);
000089           if (rc != REC_SIZE || ferror(fp)) {
000090              print_amrc();
000091              printf("Error: reading old employee record failed\n");
000092              return 60;
000093           }
000094
000095           rc = fupdate(data_ptr,REC_SIZE,fp);
000096           if (rc == 0) {
000097              print_amrc();
000098              printf("Error: updating new employee record failed\n");
000099              return 70;
000100           }
000101        }
000102        else { /* record doesn't exist so write out new record  */
000103           clearerr(fp);
000104           printf("Warning: record previously displayed no longer\n");
000105           printf("       : exists, new record being created\n");
000106           rc = fwrite(data_ptr,1,REC_SIZE,fp);
000107           if (rc != REC_SIZE || ferror(fp)) {
000108              print_amrc();
000109              printf("Error: write with new employee number failed\n");
000110              return 80;
000111           }
000112        }
000113     }
000114     return 0;
000115 }
000116
000117 /* display_emp_rec() function definition                           */
000118
000119 int display_emp_rec (struct data_struct *data_ptr,
000120                      struct data_struct *orig_data_ptr,
000121                      FILE *clus_fp, FILE *aix_unique_fp,
000122                      FILE *aix_non_unique_fp)
000123 {
000124     int     rc = 0;
000125     char    buffer[REC_SIZE+1];
000126
000127     /* Primary Key Search                                          */
```

```
000128     if (memcmp(data_ptr->emp_number, "\0\0\0\0", 4) != 0) {
000129        rc = flocate(clus_fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,
000130                     __KEY_EQ);
000131        if (rc != 0) {
000132           printf("Error: flocate with primary key failed\n");
000133           return 10;
000134        }
000135
000136        /* Read record for display                                    */
000137        rc = fread(orig_data_ptr,1,REC_SIZE,clus_fp);
000138        if (rc != REC_SIZE || ferror(clus_fp)) {
000139           printf("Error: reading employee record failed\n");
000140           return 15;
000141        }
000142     }
000143     /* Unique Alternate Index Search                                 */
000144     else if (data_ptr->user_id[0] != '\0') {
000145        rc = flocate(aix_unique_fp,data_ptr->user_id,AIX_UNIQUE_KEY_SIZE,
000146                     __KEY_EQ);
000147        if (rc != 0) {
000148           printf("Error: flocate with user id failed\n");
000149           return 20;
000150        }
000151
000152        /* Read record for display                                    */
000153        rc = fread(orig_data_ptr,1,REC_SIZE,aix_unique_fp);
000154        if (rc != REC_SIZE || ferror(aix_unique_fp)) {
000155           printf("Error: reading employee record failed\n");
000156           return 25;
000157        }
000158     }
000159     /* Non-unique Alternate Index Search                             */
000160     else if (data_ptr->name[0] != '\0') {
000161        rc = flocate(aix_non_unique_fp,data_ptr->name,
000162                     AIX_NONUNIQUE_KEY_SIZE,__KEY_GE);
000163        if (rc != 0) {
000164           printf("Error: flocate with name failed\n");
000165           return 30;
000166        }
000167
000168        /* Read record for display                                    */
000169        rc = fread(orig_data_ptr,1,REC_SIZE,aix_non_unique_fp);
000170        if (rc != REC_SIZE || ferror(aix_non_unique_fp)) {
000171           printf("Error: reading employee record failed\n");
000172           return 35;
000173        }
000174     }
000175     else {
000176        printf("Error: invalid search argument; valid search arguments\n"
000177               "     : are either employee number, user id, or name\n");
000178        return 40;
000179     }
000180     /* display record data                                           */
000181     printf("Employee Number: %.4s\n", orig_data_ptr->emp_number);
000182     printf("Employee Userid: %.8s\n", orig_data_ptr->user_id);
000183     printf("Employee Name:   %.20s\n", orig_data_ptr->name);
000184     printf("Employee Info:   %.37s\n", orig_data_ptr->pers_info);
000185     return 0;
000186 }
000187
000188 /* main() function definition                                       */
000189
000190 int main() {
000191     FILE*              clus_fp;
000192     FILE*              aix_ufp;
000193     FILE*              aix_nufp;
000194     int                i;
000195     struct data_struct  buf1, buf2;
```

```
000196
000197     char data[3][REC_SIZE+1] = {
000198 "   1LARRY   LARRY               HI, I'M LARRY,                       ",
000199 "   2DARRYL1 DARRYL              AND THIS IS MY BROTHER DARRYL,       ",
000200 "   3DARRYL2 DARRYL                                                   "
000201     };
000202
000203     /* open file three ways                                           */
000204     clus_fp = fopen("dd:cluster", "rb+,type=record");
000205     if (clus_fp == NULL) {
000206        print_amrc();
000207        printf("Error: fopen(\"dd:cluster\"...) failed\n");
000208        return 5;
000209     }
000210     /* assume base cluster was loaded with at least one dummy record  */
000211     /* so aix could be defined                                        */
000212     aix_ufp = fopen("dd:aixuniq", "rb,type=record");
000213     if (aix_ufp == NULL) {
000214        print_amrc();
000215        printf("Error: fopen(\"dd:aixuniq\"...) failed\n");
000216        return 10;
000217     }
000218     /* assume base cluster was loaded with at least one dummy record  */
000219     /* so aix could be defined                                        */
000220     aix_nufp = fopen("dd:aixnuniq", "rb,type=record");
000221     if (aix_nufp == NULL) {
000222        print_amrc();
000223        printf("Error: fopen(\"dd:aixnuniq\"...) failed\n");
000224        return 15;
000225     }
000226
000227     /* load sample records                                            */
000228     for (i = 0; i < 3; ++i) {
000229        if (fwrite(data[i],1,REC_SIZE,clus_fp) != REC_SIZE) {
000230           print_amrc();
000231           printf("Error: fwrite(data[%d]...) failed\n", i);
000232           return 66+i;
000233        }
000234     }
000236     /* display sample record by primary key                          */
000237     memcpy(buf1.emp_number, "   1", 4);
000238     if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
000239        return 69;
000240
000241     /* display sample record by nonunique aix key                    */
000242     memset(buf1.emp_number, '\0', 4);
000243     buf1.user_id[0] = '\0';
000244     memcpy(buf1.name, "DARRYL                    ", 20);
000245     if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
000246        return 70;
000247
000248     /* display sample record by unique aix key                       */
000249     memcpy(buf1.user_id, "DARRYL2 ", 8);
000250     if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
000251        return 71;
000252
000253     /* update record just read with new personal info                */
000254     memcpy(&buf1, &buf2, REC_SIZE);
000255     memcpy(buf1.pers_info, "AND THIS IS MY OTHER BROTHER DARRYL. ", 37);
000256     if (update_emp_rec(&buf1, &buf2, clus_fp) != 0) return 72;
000257
000258     /* display sample record by unique aix key                       */
000259     if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
000260        return 73;
000261
000262     return 0;
000263 }
****** ************************** Bottom of Data **************************
```

Závěr

Práce v současném rozsahu pokrývá základní úvod do problematiky mainframů. To z ní činí vhodnou studijní pomůcku pro všechny zájemce o tuto oblast. Informace a příklady vyčtené z literatury a obsažené v této práci jsem většinou experimentálně ověřil přímo na mainframu, takže jsou prakticky použitelné bez dalších úprav. Navíc jsem občas přidal i nějaké vlastní tipy z praxe, které v literatuře obsažené nejsou, ale které mohou být pro práci s mainframy důležité.

Hodnotu práce zvyšuje také to, že byla konzultována s odborníky z praxe, kteří přichází s mainframy denně do styku. Díky připomínkám pana Ing. Milana Svátka ze společnosti CA jsem mohl upravit teoreticky nabyté informace i navržené české názvosloví tak, aby odpovídaly opravdu používaným. Pan Ing. Svátek také odhalil některé nepřesné formulace, které by mohly být pro neodborníka zavádějící.

Vzhledem k rozsáhlosti problematiky mainframů neobsahuje práce mnohé zajímavé kapitoly, které by do ní nepochybně patřily. Jistě by bylo vhodné více se věnovat programování v assembleru, zjistit jak fungují databáze a programy pro jejich obsluhu či naučit se pracovat s transakčním podsystémem CICS. Bohužel na to již nevyšel čas, ale doufám, že v budoucnu se přinejmenším několika z těchto témat budu moci věnovat.

I tak věřím, že práce svůj úkol splnila a že v příštích letech pomůže alespoň některým českým zájemcům v počátcích seznamování se s problematikou mainframů.

Seznam zkratek a slovníček

Zkratka	Anglicky	Česky
--	Batch job	Dávková úloha
BCP	Base control program	Základní řídící program
BX	Byte index	Index bajtu
CICS [kiks]	Customer Information Control System	
--	Cross-memory services	Služby skrz paměť
CSA	Common service area	
DASD	Direct access storage device	Paměťové zařízení s přímým přístupem
DAT	Dynamic address translation	Dynamický překlad adresy
DFSMS	Data Facility Storage Management Subsystem	
DM	Dialog manager	Správce dialogu
ESDS	Entry-Sequenced Data Set	ESDS soubor
GDG	Generation data group	Generační soubor
GDPS	Geographically Dispersed Parallel Sysplex	
	In-stream procedure	Včleněná procedura
ISPF	Interactive System Productivity Facility	
IPA	Interprocedural Analysis	Interprocedurální analýza
JCL	Job control language	Jazyk řízení úloh
JES	Job entry subsystem	Podsystém vstupu úloh
KSDS	Key-Sequenced Data Set	KSDS soubor
	Language Environment	Jazykové prostředí
LDS	Linear Data Set	LDS soubor
LPA	Link pack area	
LPAR	logical partition	Logický oddíl
OS	operating system	Operační systém
PDF	Program Development Facility	Prostředek pro vývoj programů
PDF	Profile-Directed Feedback	Tvarem řízená zpětná vazba
PDS	Partitioned data set	Knihovna
PX	Page index	Index stránky
RBA	Relative Byte Address	Relativní bajtová adresa
REXX	REstructured eXtended eXecutor	Programovací jazyk REXX
RRDS	Relative Record Data Set	RRDS soubor
RFX, RSX, RTX	Region first, second, third index	První, druhý, třetí index regionu
SCLM	Software Configuration Library Manager	Správce softwarové konfigurace knihoven
SDSF	System Display and Search Facility	Prostředek pro zobrazování a prohledávání manipulačního prostoru
SQA	System queue area	

SX	Segment index	Index segmentu
TSO, TSO/E	Time Sharing Option/Extensions	
VSAM	Virtual Storage Access Method	Metoda pro přístup k virtuální paměti
WLM	Workload manager	Správce zátěže

Seznam použité literatury

1. Ebbers M., O'Brien W., Ogden B.:
   Introduction to the New Mainframe: z/OS Basics, IBM, First Edition, 2005
2. Rogers P., Gelinsky M., Oliveira J. N., Sokal V.:
   ABCs of z/OS System Programming Volume 1, Redbook IBM, 2003
3. Rogers P., Gelinsky M., Oliveira J. N.:
   ABCs of z/OS System Programming Volume 2, Redbook IBM, 2003
4. Kolektiv IBM:
   Interactive System Productivity Facility (ISPF) User's Guide Volume I, IBM, Fourth Edition, 2004
5. Kolektiv IBM:
   MVS JCL User's Guide, IBM, Third Edition, 2003
6. Kolektiv IBM:
   C/C++ Programming Guide, IBM, Sixth Edition, 2004
7. Kolektiv IBM:
   C/C++ User's Guide, IBM, Fourth Edition, 2004
8. Kolektiv IBM:
   DFSMS Access Method Services for Catalogs, IBM, Fourth Edition, 2004
9. Kolektiv IBM:
   Language Environment Programming Guide, IBM, Sixth Edition, 2004
10. Kolektiv IBM:
    TSO/E REXX User's Guide, IBM, Second Edition, 2001
    (Part 1, Learning the REXX)
11. Kolektiv CA:
    IBM Mainframes (Technical advantages, Availability), firemní prezentace, 2005
12. www.wikipedia.org
13. www.rexxinfo.org

Jednotlivá díla byla použita především v následujících kapitolách:

[1]:	1., 2., 3.2., 3.3., 4.1., 4.2.	[8]:	2.5.
[2]:	2.1. až 2.4., 3.	[9]:	4.1.
[3]:	2.6.	[10]:	5.
[4]:	2.3.	[11]:	1.
[5]:	3.2.	[12]:	4.1., 5.
[6]:	4.6., 4.7.	[13]:	5.
[7]:	4.2. až 4.5., 4.8.		