INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

OPERAČNÍ SYSTÉMY

MANAGEMENT PROCESŮ

doc. Dr. Ing. Oldřich Kodym

Ostrava 2013

© doc. Dr. Ing. Oldřich Kodym

© Vysoká škola báňská – Technická univerzita Ostrava

ISBN 978-80-248-3053-7

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

3. MANAGEMENT PROCESŮ 3

1. Procesy 4

1.1 Stavy procesu 4

1.2 Process Control Block 5

2. Plánování procesů 5

2.1 Plánovací fronty 5

2.2 Přepínání kontextu 8

3. Operace s procesy 8

3.1 Vytvoření procesu 8

3.2 Zrušení procesu 9

4. Spolupráce procesů 10

5. Komunikace procesů 11

5.1 Základní struktura 11

5.2 Přímá komunikace 12

5.3 Nepřímá komunikace 13

5.4 Buffery 14

5.5 Výjimečné situace 15

5.6 Komunikace v OS UNIX 16

3. MANAGEMENT PROCESŮ

OBSAH KAPITOLY:

Stavový diagram procesů.

Plánování procesů, plánovací fronty.

Operace s procesy.

Spolupráce procesů.

Vlákna.

MOTIVACE:

Chod operačního systému využívá mnoha standardních mechanizmů známých z jiných oblastí řízení i běžného života. Proces jako takový není pouze kód (instrukce) programu, ale zahrnuje v sobě i všechny jeho "aktivity", které jsou reprezentovány hodnotami ukazatele programu (program counter) a obsahem registrů procesoru. Životní cyklus procesu sestává z přesunu mezi jednotlivými plánovacími frontami OS. To, který proces je třeba zařadit do které fronty, rozhoduje OS pomoci tzv. plánovačů. Operační systém musí provádět mechanismy vedoucí k vytvoření a zrušení procesu. Spolupráci procesů si lze představit jako spolupráci výrobce a příjemce. Producent produkuje informace, které přijímá příjemce.

CÍL:

Stavy procesu, Process Control Block, přepínání kontextu

Plánování procesů, plánovací fronty

Operace s procesy – vytvoření, zrušení

Komunikace procesů – základní principy, přímá a nepřímá komunikace, buffery

1. PROCESY

Starší počítače umožňovaly spouštět pouze jeden program. Tento program plně využíval OS i všechny systémové zdroje. Současné počítače umožnují běh více programu současně. Instancí běžícího programu v takovém systému je *proces*. *Proces* je základní jednotka práce moderního OS se sdílením času.

Takový systém obsahuje velké množství souběžně spuštěných procesů:

- procesů operačního systému,
- uživatelských procesů.

Všechny tyto procesy běží zdánlivě současně a CPU je střídavě obsluhuje. OS se tím stává produktivnějším.

Proces jako takový není pouze kód (instrukce) programu, ale zahrnuje v sobě i všechny jeho "aktivity", které jsou reprezentovány hodnotami *ukazatele programu (program counter)* a obsahem registrů procesoru. V globálu proces v sobě zahrnuje také *zásobník (stack)* obsahující dočasná data (jako např. parametry volaných podprogramů, návratové adresy a dočasné proměnné) a sekci dat pro globální proměnné (code segment, stack segment, data segment viz Segmentace paměti).

Program sám o sobě není proces! Je to pasivní entita existující např. jako soubor dat na disku. Proces je aktivní entita s ukazatelem programu, který specifikuje následující instrukci, která se má vykonat a dalších zdrojů (viz výše).

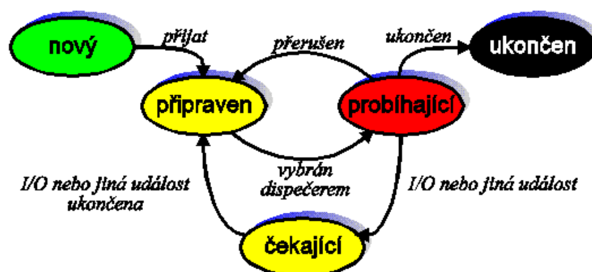
Více procesů může byt asociováno s týmž programem. Několik uživatelů najednou může spustit např. mailového klienta nebo textový editor. Každý z těchto uživatelů vytvoří svůj proces, který má identickou kódovou část s ostatními, ale datovou má každý proces jinou.

1.1 Stavy procesu

Každý proces se může vzhledem ke své momentální aktivitě vyskytovat v některém z následujících stavů:

- **Nový (New)** – proces byl právě vytvořen.
- **Probíhající (Running)** – instrukce procesu začala být vykonávána.
- **Čekající (Waiting)** – proces čeká na nějakou událost (např. na dokončení I/O operace nebo přijetí signálu).
- **Připraven (Ready)** – proces má k dispozici všechny zdroje, čeká jen na procesor.
- **Ukončen (Terminated)** – proces dokončil svou činnost.

Názvy stavů procesu se někdy liší mezi jednotlivými OS, ale všechny tyto stavy všechny systémy obsahují. Je vhodné zaznamenat, že v každém systému může být v danou chvíli pouze jeden proces ve stavu *probíhající* na jednom procesoru a mnoho procesů může být ve stavu *čekající* nebo *připravený*. Stavový diagram procesů je na obr. 1.



Obrázek 1 - Stavový diagram procesu

1.2 Process Control Block

Každý proces je v OS reprezentován záznamem, který se jmenuje *Process Control Block*. Struktura PCB je na obr. 2. Obsahuje množství informací o procesu, jako např.:

- **Status procesu (Process state)** - {Nový, Probíhající, Čekající, Připravený, Ukončen}.
- **Ukazatel programu (Program counter)** - obsahuje adresu nesledující instrukce v průběhu vykonávání procesu.
- **CPU registry (CPU registers)** - počet registrů je různý v závislosti na architektuře procesoru. Jsou tam akumulátory (registr výsledků aritmetickologické jednotky), indexové registry, ukazatele do zásobníků aj. V případě přerušení běhu procesu musí být aktuální obsah registrů, stejně jako ukazatel programu uložen do PCB, aby při dalším přidělení procesoru mohl proces korektně pokračovat.
- **CPU plánovací informace (CPU scheduling information)** - priorita procesu, ukazatele do plánovacích front aj.
- **Informace správy paměti (Memory management information)** - počet base a limit registrů, tabulky stránek nebo segmentů.
- **Účtovací informace (Accounting information)** - informace o čase přidělení procesoru, časový limit, číslo procesu aj.
- **I/O stavové informace (I/O status information)** - seznam I/O zařízení alokovaných pro tento proces, seznam otevřených souborů apod.

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
. . .	

Obrázek 2 - Process Control Block

2. PLÁNOVÁNÍ PROCESŮ

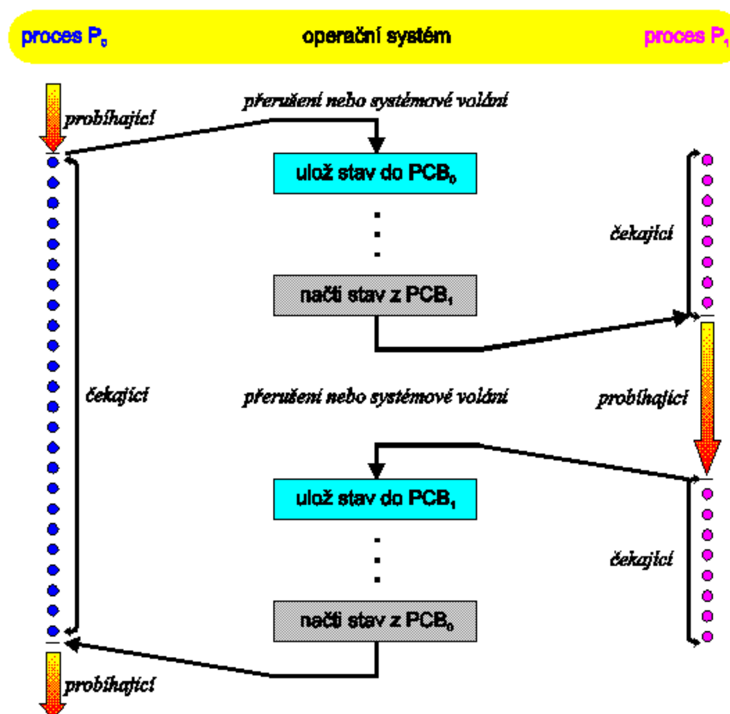
Systém s jedním (jednojádrovým) procesorem může v danou chvíli obsluhovat pouze jeden proces. Přepínání mezi procesy ukazuje obr. 3. Procesy jsou pro přidělování procesoru řazeny do tzv. plánovacích front.

2.1 Plánovací fronty

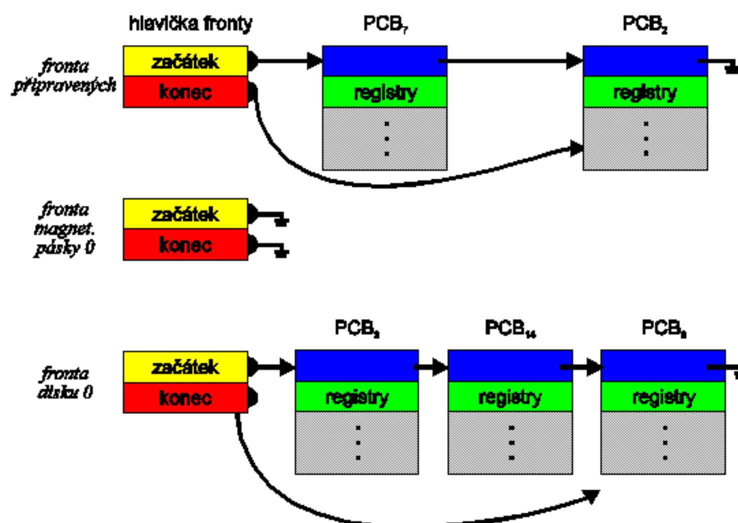
V okamžiku, kdy je proces vytvořen, je jeho záznam vložen do *fronty procesu (job queue)*. Tato fronta obsahuje všechny procesy v systému.

Procesy, které jsou ve fyzické paměti a čekají na přidělení procesoru, jsou ve *frontě připravených (ready queue).* Tato fronta je fronta ukazatelů. Hlavička *fronty připravených* obsahuje ukazatel na PCB bloky prvního a posledního procesu ve frontě. Kazdy PCB blok potom obsahuje položku ukazatele na PCB blok následujícího procesu ve *frontě připravených.*

Sejde-li se najednou více požadavků na určité I/O zařízení, OS tyto požadavky spravuje pomocí *fronty zařízení (device queue)*. Každé I/O zařízení má svou vlastní *frontu zařízení.* (viz obr 4).



Obrázek 3 - Přepínání CPU mezi procesy



Obrázek 4 - Plánovací fronty OS

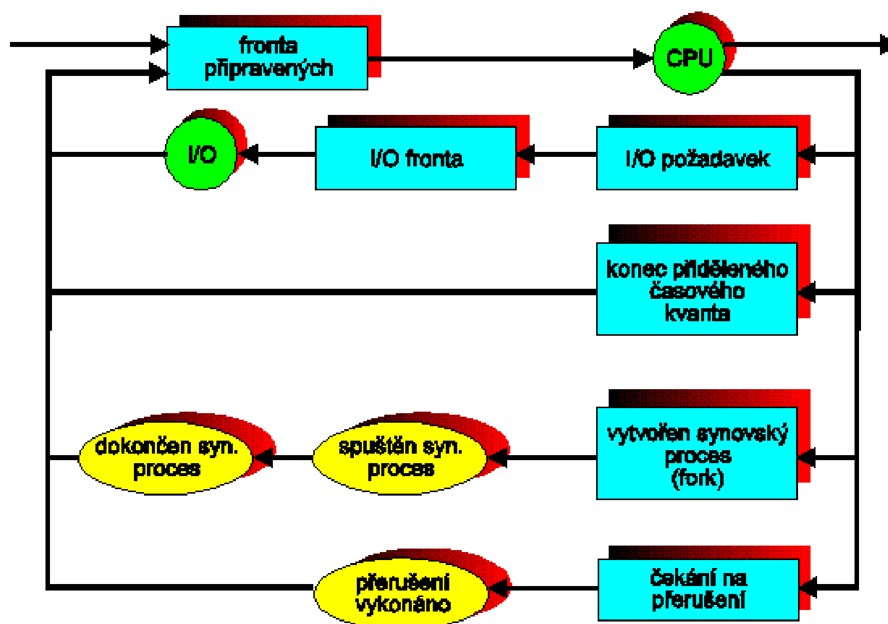
Na obr. 5 je diagram front OS. Každý obdélník reprezentuje jednu frontu. Je na něm možno nalézt 2 základní typy front: (1) *frontu připravených*; (2) *fronty I/O zařízení*. Každý kruh reprezentuje zdroj, který obsluhuje tu kterou frontu, a šipky určují toky požadavků v systému.

Novy proces je inicializován, zařazen do fronty připravených a čeká na přidělení CPU. V okamžiku, kdy se mu dostane je spuštěn a může nastat některá z následujících možností:

- Proces požaduje I/O operaci a je zařazen do příslušné I/O fronty.
- Proces vytvořil podproces a čeká na jeho dokončení.

- Procesu je násilně odebrána CPU v důsledku přerušení a je uložen zpět do fronty připravených.

V prvních dvou případech se proces dostává do stavu čekající a po dokončení I/O operace je navrácen zpět do fronty připravených a tedy do stavu připraven. Proces probíhá tímto cyklem, dokud není ukončen. Potom je vymazán ze všech front a jeho PCB je uvolněn.



Obrázek 5 - Diagram front OS

Životní cyklus procesu sestává z přesunu mezi jednotlivými plánovacími frontami OS. To, který proces je třeba zařadit do které fronty, rozhoduje OS pomoci tzv. *plánovačů*.

V dávkových systémech bývá často mnoho procesů odloženo v záložní paměti, než budou spuštěny. Tyto procesy jsou spoolovany (nejčastěji na disku) pro pozdější spuštění. *Plánovač úloh (job scheduler)* vybírá procesy z tohoto spoolu a zavádí je do fyzické paměti ke spuštění. *Plánovač procesů (CPU scheduler)* vybírá z připravených procesů ten, kterému bude přidělen procesor.

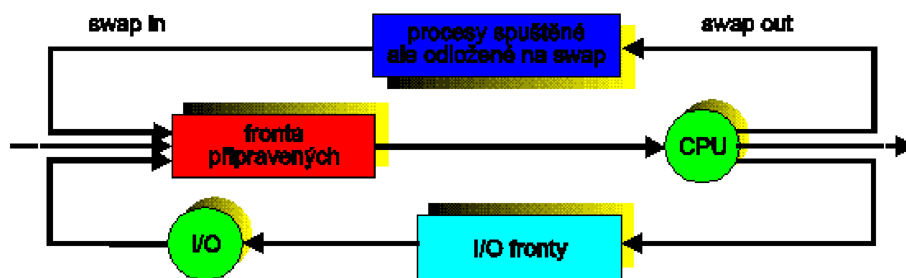
Tito dva plánovači se od sebe výrazně liší frekvencemi, s jakými jsou spouštěni. *Plánovač procesů* musí vybírat proces pro přidělení CPU mnohem častěji. Proces může být spuštěn jen na pár milisekund, než je nucen např. čekat na nějakou I/O operaci. Často je plánovač procesu spouštěn i každých 100 ms. Během přidělování CPU musí tedy být *Plánovač procesů* **velmi** rychlý. Pokud by přepřidělení procesoru trvalo jen 10 ms, potom 10/(10 + 100) = 9 % z casu CPU je zabráno jen pro jeho přidělování.

Plánovač úloh je naopak spouštěn s mnohem menší frekvencí. Mezi vstupem nového procesu do systému mohou uběhnout řádově minuty. *Plánovač úloh* sleduje počet procesů v paměti (degree of multiprograming). Delší dobu mezi vstupy jednotlivých procesů do systému může *plánovač úloh* věnovat rozhodování, který další proces do systému zařadit.

Plánovač úloh musí vpouštět úlohy do systému. Procesy se v podstatě dají rozdělit na ty, které převážně využívají I/O zařízení (I/O-bound process) nebo CPU (CPU-bound process). Je vhodné, aby *plánovač procesů* zatahoval do systému rovnoměrné oba tyto typy procesů. Pokud by všechny procesy využívaly převážně I/O zařízení, potom by fronta připravených byla často prázdná a plánovač procesů by měl málo co na práci. Pokud by všechny procesy využívaly převážně CPU, potom by byly prázdné I/O fronty, zařízení by byla málo využita a systém by byl nevyvážený.

V některých systémech není *plánovač úloh* implementovaný, nebo je minimalizován. V systémech se sdílením času velmi často plánovač není a všechny procesy jsou zaváděny přímo do paměti a předány plánovači procesů. Stabilita takových systémů je limitována jednak hardwarovými aspekty (např. počtem terminálů v závislosti na velikosti paměti) a také psychickými vlastnostmi uživatelů (pokud výkon nebo odezva klesne pod přijatelnou úroveň, rozumní odcházejí dělat něco jiného).

Zejména systémy se sdílením času mají navíc interaktivní úroveň přidělování. Myšlenka spočívá v tom, že někdy může být pro OS výhodné odstranit aktivní proces na čas z paměti od CPU, a snížit tak počet úloh v paměti. Později může být proces opět do paměti zaveden a spuštěn tak, že pokračuje ve výpočtu od místa, kde byl přerušen – *swapping*. Swapování řídí *plánovač střední doby (medium-term scheduler).* Swapování je vhodné pro předělovaní CPU nebo pokud nároky změn v paměti přesahují její současnou volnou část apod.



Obrázek 6 - Swapovací mechanismus v diagramu front

2.2 Přepínání kontextu

Přidělení procesoru jiné úloze vyžaduje uložit stav procesu, který procesor opouští a načíst stav procesu, který k procesoru přichází. Tato výměna se nazývá *přepínání kontextu (context switch)* viz obr. 3.

Přepínání kontextu představuje čistě režii systému, protože ten během přepínání kontextu nemůže dělat nic jiného. Rychlost přepínání je různá mezi jednotlivými stroji. Je podmíněna vybavovací rychlostí paměti, počtem registrů, které musí být překopírovány apod. Typická doba je mezi 1 až 1000 ms.

Přepínání kontextu zahrnuje nejdříve změnu ukazatele do paměti na aktuální kontext. Ten je pak z paměti načten (při vstupu procesu) nebo je do paměti uložen aktuální stav (pokud proces procesor opouští).

3. OPERACE S PROCESY

V první řadě musí operační systém provádět mechanismy vedoucí k vytvoření a zrušení procesu.

3.1 Vytvoření procesu

Libovolný proces může vytvořit nový proces prostřednictvím volání vytvoření procesu. Takový proces se potom nazývá *rodičovským procesem (parent process)* vzhledem k *synovskému procesu (children process)*, který vytvořil. Každý ze synovských procesů může vytvářet další své synovské procesy, čímž vzniká *strom procesů* (viz obr. 7).

Proces pro své naplnění potřebuje zdroje: čas CPU, paměť, soubory, periferie. Nově vzniklý proces může zdroje požadovat buď od OS, nebo může využívat část rodičovských zdrojů (např. soubory nebo paměť). Zúžení zdrojů synovského procesu na zdroje rodiče umožňuje

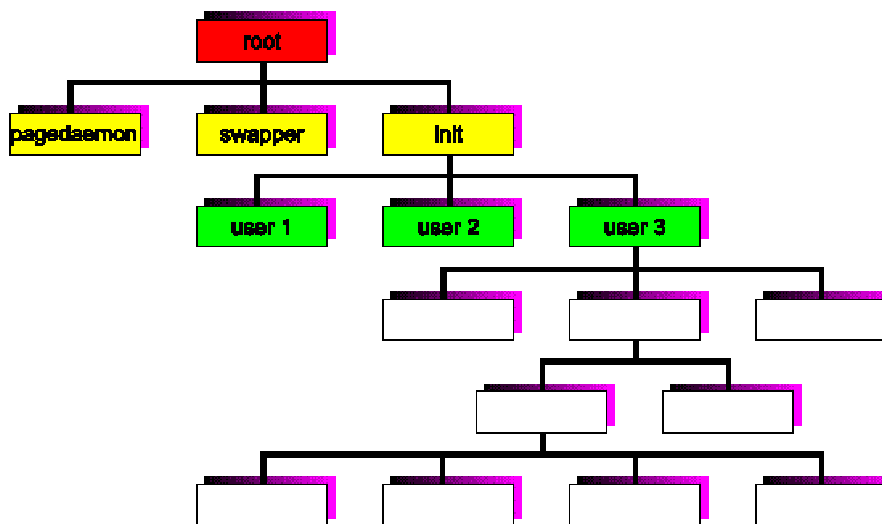
předejít zahlcení systému v důsledku vytvoření příliš mnoha podprocesů takto omezeným procesem rodičovským.

V okamžiku, kdy proces vytvoří další proces, jsou 2 možnosti jejich další existence:

- Rodičovský proces běží dále souběžně se synovským.
- Rodičovský proces čeká na to než některý nebo všechny synovské procesy budou dokončeny.

Vzhledem k adresovému prostoru obou procesů nastávají také 2 možné situace:

- Synovský proces je duplikátem rodičovského.
- Synovský proces má vlastní program a ten je zaveden do paměti.



Obrázek 7 - Strom procesů v OS Unix

Vytváření synovských procesů je v různých OS implementováno různé. V OS UNIX má každý proces jedinečné identifikační číslo (PID). Nový proces je vytvářen voláním jádra **fork**. Vytvořený proces potom obsahuje kopii adresového prostoru procesu rodičovského. To zjednodušuje komunikaci rodiče se synem. Oba procesy pokračují dále za voláním **fork** s jedním rozdílem – návratový kód instrukce **fork** je 0 pro synovský proces a různý od 0 (obsahuje PID synovského procesu) pro rodičovský.

Po užití volání **fork** je často provedeno volání jádra **execve** pro přepsání adresového prostoru procesu novým programem. Voláni **execve** načte do paměti binární soubor a zruší její obsah, který byl před ním a tento soubor spustí. V tomto případě jsou oba procesy schopny komunikovat a potom jde každý svou cestou. Jestliže rodičovský proces čeká na dokončení synovského, užije volání **wait**, které ho odstraní z fronty připravených, dokud nebude dokončen synovský proces.

DEC VMS OS naopak když vytvoří nový proces, natáhne jeho program do paměti a spustí ho. MS Windows NT podporuji obě možnosti vytvoření synovského procesu.

3.2 Zrušení procesu

Ke zrušení procesu dochází poté, co jsou vykonány všechny jeho instrukce a OS ho zajišťuje voláním jádra **exit**. Tehdy může proces vrátit data (výsledek) rodičovskému procesu.

Všechny zdroje alokované pro proces (fyzická paměť, virtuální paměť, otevřené soubory a I/O buffery) jsou uvolněny operačním systémem.

Zrušení procesu může vyžádat i jiný proces prostřednictvím volání **abort**. Většinou smí být tato žádost vyplněna pouze rodičovskému procesu, jinak by libovolný uživatel mohl zrušit libovolný proces. Proto, aby rodič mohl případně zrušit své synovské procesy, musí si o nich uchovávat informace, které získá při vytváření těchto procesů.

Rodič může žádat zrušení synovských procesů z různých důvodů, např.:

- syn překročil možnosti, které mu byly poskytnuty na nějakém systémovém zdroji;
- úkol, který měl syn splnit, již není požadován;
- rodičovský proces byl ukončen a OS ukončuje všechny jeho potomky.

V každém případě musí mít rodičovské procesy k dispozici mechanismy pro kontrolu svých potomků. Mnoho OS (např. VMS) nedovolují existenci synovského procesu po ukončení procesu rodičovského. Je-li rodič ukončen, jsou ukončeny i všichni jeho potomci – *kaskádové ukončení (cascading termination).*

V UNIXu může být proces ukončen voláním jádra **exit**. Rodičovský proces může čekat na výsledky potomka užitím volání **wait**. Volání **wait** vrací PID potomka, na jehož ukončení rodič čeká. Při zrušení rodiče jsou běžně zrušeni i jeho potomci, protože OS neví, kam posílat zprávy o jejich činnosti.

4. SPOLUPRÁCE PROCESŮ

Procesy souběžně spuštěné v systému mohou byt buď *autonomní*, nebo *kooperující*. Proces je *autonomní* pokud nemůže ovlivňovat ani být ovlivněn žádným jiným souběžně běžícím procesem v systému (nemůže sdílet data – dočasná nebo trvalá – s žádným jiným procesem). Proces je *kooperující* jestliže může ovlivňovat nebo být ovlivněn jiným procesem spuštěným v systému (proces sdílí data s jiným/jinými).

Možnosti spolupráce procesů:

- **Sdílení informací (Information sharing):** Různí uživatelé systému mohou chtít ve stejném čase získat stejné informace (např. sdílený soubor) a OS musí zajistit tyto přístupy uživatelů ke sdíleným zdrojům.
- **Urychlení výpočtu (Computation speedup):** V případě, že výpočetní systém má více zdrojů typu CPU nebo I/O kanál, je možné dílčí úlohy procesu od něj oddělit a spustit je paralelně s ním.
- **Modularita (Modularity):** Možnost vytvářet program jako modulární systém a pro každý modul vytvořit vlastní proces.
- **Zvýšení pohodlí (Convenience):** Uživatel v systému může mít paralelně spuštěno množství programů (např. může editovat, tisknout i překládat zároveň).

Spolupráce procesů vyžaduje mechanismy povolující popř. zakazující komunikaci mezi procesy a mechanismy pro synchronizaci procesů.

Spolupráci procesů si lze představit jako spolupráci *výrobce* a *příjemce*. Producent produkuje informace, které přijímá příjemce. Např. program pro tisk produkuje znaky, které přijímá ovladač tiskárny. Překladač produkuje kód, který dále zpracovává assembler, který produkuje moduly zpracovávané linkerem atd.

Pro spolupracující procesy je třeba vytvořit buffer, který bude naplňován producentem a vyprazdňován příjemcem. Producent může vytvořit další položku v případě, že příjemce jinou odebral – příjemce a producent musí být synchronizováni.

Neomezený buffer (unbounded buffer) – velikost bufferu není omezena – příjemce musí čekat, je-li buffer prázdný, producent může stále produkovat. *Omezený buffer (bounded buffer)* – buffer má pevnou velikost – příjemce musí čekat, je-li buffer prázdný, producent musí čekat, je-li buffer plný. Buffer může obhospodařovat buď OS skrze IPC (*interprocess communication facility*, viz dále) nebo vlastní uživatelský program ve sdílené paměti. Ilustrujme komunikaci pomocí sdílené paměti a omezeného bufferu. Producent a konzument sdílejí tyto proměnné:

```
var n;
type polozka = ....;
var buffer: array (0..n-1) of polozka;
in, out: 0..n-1
```

In, out mají počáteční hodnotu 0. Sdílený buffer je implementován pomoci rotačního pole se 2 logickými ukazateli *in* a *out*. *In* ukazuje na následující volnou pozici v bufferu, *out* ukazuje na první plnou pozici v bufferu => buffer je plný, jestliže *in* = *out*, a je prázdný, jestlize (*in* + 1) **mod** n = *out*.

Kód producenta i příjemce využívá operaci *nic_nedelej*, která je užita v případě, že jeden z nich musí čekat (z výše uvedených důvodů). Producent navíc využívá proměnnou *nova_hodnota*, kam v každém kroku uloží položku, která se v něm má předat do bufferu. Kód producenta:

```
repeat
    {vytvor dalsi polozku do nova_hodnota}
    while in + 1 mod n = out do nic_nedelej;
    buffer(in) := nova_hodnota;
    in := in + 1 mod n;
until false;
```

Konzument navíc obsahuje lokální proměnnou *prijata_hodnota,* do které se uloží nová položka přijatá z bufferu. Kód příjemce:

```
repeat
    while in = out do nic_nedelej;
    prijata_hodnota := buffer(out);
    out := out + 1 mod n;
    {zpracuj prijatou polozku v prijata_hodnota}
until false;
```

Uvedený algoritmus dovoluje současně v bufferu n – 1 položek. Později probereme mechanismy synchronizace a sdílení paměti.

5. KOMUNIKACE PROCESŮ

V odstavci 4 bylo ukázáno, jak procesy mohou komunikovat prostřednictvím sdílené paměti. Tento postup vyžaduje existenci společného bufferu, který je explicitně vytvořen uživatelským programem. Jiná cesta k dosažení téhož je v podpoře komunikace procesů přímo OS přes tzv. *interprocess communication (IPC) facility*.

IPC umožňuje procesům komunikovat a synchronizovat své akce. IPC vytváří systém zpráv. Jak sdílená paměť, tak i systém zpráv mohou být užity současně.

5.1 Základní struktura

Systém zpráv musí umožňovat minimálně 2 operace: **send**(zprava) a **receive**(zprava).

Zprávy doručované systémem mohou byt proměnné nebo pevné délky. Implementace přenosu pouze zpráv pevné délky je náročnější pro programátora, na druhé straně však implementace přenosu zpráv proměnné délky vyžaduje větší fyzickou podporu.

Pokud chtějí procesy *P* a *Q* komunikovat, musejí být navzájem schopny přijímat a vysílat své zprávy. Musí mezi nimi existovat *komunikační spojeni (communication link).* Toto spojení je implementováno různě. Hardwarovou implementaci (sdílená paměť, sběrnice, síť) ponechme zatím stranou a věnujme se softwarové implementaci.

Spojení může být navázáno i mezi více než dvěma procesy. Spojení je *nesměrové (undirectional)*, jestliže každý proces do něj zapojený může buď jen vysílat, nebo jen přijímat, ale ne obojí a každé spojení obsahuje minimálně jednoho vysílajícího a jednoho příjemce.

Následují různé logické implementace spojení:

- Přímá nebo nepřímá komunikace.
- Symetrická nebo nesymetrická komunikace.
- Automatické nebo explicitní bufferování.
- Posílání kopie nebo posílání odkazu.
- Zprávy pevné nebo proměnné délky.

5.2 Přímá komunikace

V tomto typu komunikace musí každý proces, který chce komunikovat znát jméno příjemce. V tomto schématu je definice funkcí **send** a **receive** následující:

- **send***(P, zprava)* – Pošli *zpravu* procesu *P.*
- **receive***(Q, zprava)* – Přijmi *zpravu* od procesu *Q.*

Komunikační spojení v tomto schématu má následující vlastnosti:

- Spojení je vytvořeno mezi právě dvěma procesy.
- Spojení je automaticky povoleno mezi každými dvěma procesy, které chtějí komunikovat. Pro komunikaci musí procesy pouze navzájem znát svou identifikaci.
- Mezi každými dvěma procesy může existovat maximálně jedno spojení.
- Spojení může být nesměrové, ale většinou je dvou směrové.

Problém producenta a příjemce realizovaný přímou komunikací by vypadal následovně: Oba procesy jsou spuštěny současně a konzument zpracovává položku, zatímco producent již vytváří jinou. V okamžiku kdy ji vytvoří, pošle ji pomocí funkce **send** příjemci. Ten ji přijme prostřednictvím funkce **receive**. Není-li vyprodukována žádná zpráva, je producent ve stavu čekající.

Kostra algoritmu procesu producenta:

```
repeat
    {vytvor dalsi polozku do promenne nova_hodnota}
    send(prijemce, nova_hodnota)
until false
```

Kostra algoritmu příjemce:

```
repeat
    receive (producent, prijata_hodnota)
    {zpracuj polozku v promenne prijata_hodnota}
until false
```

Výše uvedené algoritmy poukazují na symetrii příjemce a producenta - oba musejí znát navzájem své identifikace, aby mohli komunikovat. Druhou variantou je asymetrické adresování tak, že pouze producent zná adresu příjemce, příjemce vsak adresu producenta znát nemusí. V tomto případě jsou záhlaví funkcí **send** a **receive** následující:

- **send***(P, zprava)* – Pošli *zpravu* procesu *P*.
- **receive***(id, zprava)* – Přijmi *zpravu* od procesu. Do proměnné *id* je vložena identifikace procesu, od kterého byla zprava přijata.

Nevýhodou obou těchto schémat komunikace (symetrické i asymetrické) je limitovaná modularita konečné definice procesů. Změna jména procesu si vynutí prohlídku definic všech ostatních procesů. Všechny odkazy na staré jméno musí být nalezeny a zaměněny. Tato situace je nežádoucí.

5.3 Nepřímá komunikace

V této formě komunikace si producent a příjemce vyměňují zprávy pomocí *schránky (mailbox)* také nazývané *port*. Schránka může být chápána jako objekt, do kterého procesy vkládají zprávy a odkud mohou být tyto zprávy procesy vyzvedávány.

Každá schránka má jedinečnou identifikaci. V tomto případě může proces komunikovat s jinými prostřednictvím množství různých schránek. Dva procesy mohou komunikovat pouze, sdílí-li společný mailbox. Funkce **send** a **receive** mají následující hlavičku:

- **send***(A, zprava)* – Pošli *zpravu* do schránky *A*.
- **receive***(A, zprava)* – Přijmi *zpravu* ze schránky *A*.

V tomto případě má komunikační spojení následující charakteristiky:

- Spojení je navázáno mezi dvěma procesy pouze tehdy, mají-li sdílenou schránku.
- Spojení může být navázáno mezi více než dvěma procesy.
- Mezi každým párem komunikujících procesů může být navázáno více spojení; každé spojení vyžaduje jednu schránku.
- Spojení může být nesměrové nebo dvou směrové.

Uvažujme, že procesy P1, P2 a P3 sdílí společnou schránku A. P1 pošle zprávu do schránky A, zatímco procesy P2 a P3 mají spuštěnou funkci **receive** ze schránky A. Který proces přijme zprávu od P1? Otázka může být řešena více způsoby:

- Povolit spojení mezi více než dvěma procesy.
- Povolit nejvýše jedno spuštěni operace **receive** v čase.
- Povolit systému, aby vybíral, který proces zprávu obdrží.

Vlastníkem schránky může být buď OS, nebo proces. Je-li schránka ve vlastnictví procesu (je jím definována), rozlišuje se mezi vlastníkem (tím, kdo do schránky může zapisovat) a uživatelem schránky (tím, kdo z ní může jenom číst).

Každá schránka má jedinečného vlastníka. Je-li proces, který vlastní schránku ukončen je schránka zrušena. Pokud by jiné procesy chtěly nadále komunikovat s touto schránkou, jsou upozorněny, že schránka již neexistuje (prostřednictvím výměny námitek viz dále).

Je více cest, jak stanovit vlastníka a uživatele schránky. Jedna možnost je zavést typ proměnné schránka. Proces, který má deklarovanou proměnnou typu schránka je jejím vlastníkem. Ostatní procesy, které znají jméno této schránky, ji mohou užívat.

Druhým typem jsou schránky ve vlastnictví operačního systému. OS provádí mechanismy vedoucí k:

- Vytvoření nové schránky.
- Poslání a přijetí zprávy prostřednictvím schránky.
- Zrušení schránky.

Proces, který schránku vytvořil, je implicitně jejím vlastníkem a jedině on může do schránky zasílat zprávy. Vlastnictví, stejné jako právo zápisu do schránky může být uděleno jinému procesu prostřednictvím přerušení.

Procesy mohou sdílet schránku prostřednictvím techniky tvorby procesu. Pokud proces *P* vytvoří schránku *A* a posléze vytvoří novy proces *Q*, potom oba procesy *P* a *Q* sdílí schránku *A*.

5.4 Buffery

Kapacita spojení je dána množstvím zpráv, které mohou čekat v záloze, než si je příjemce odebere. V tomto směru je možno chápat spojení jako frontu zpráv. Pro implementaci fronty jsou možné 3 základní způsoby:

- **Fronta s nulovou kapacitou**: Maximální délka fronty je 0; ve spojení nemohou tedy být žádné čekající zprávy v záloze. V tomto případě musí odesílatel čekat, dokud příjemce zprávu nepřevezme. Oba procesy musí být synchronizovány pro přenos zpráv. Tato synchronizace se nazývá *randevous*.
- **Fronta s omezenou kapacitou**: Fronta má konečnou délku *n*; maximálně *n* zpráv může být do ní vloženo. Jestliže fronta v okamžiku zaslání nové zprávy není plná, zpráva je do ní zařazena (je do fronty nakopírována, nebo je do fronty zařazen ukazatel na zprávu) a odesílatel může pokračovat v práci bez čekání. Fronta má ovšem konečnou délku a je-li plná, musí odesílatel čekat s další zprávou, než se ve frontě uvolní místo.
- **Fronta s neomezenou kapacitou**: Fronta má teoreticky neomezenou délku a může v ní čekat teoreticky neomezený počet zpráv. Odesílatel teoreticky nikdy nečeká.

Případ (1) je někdy označován jako systém zpráv bez bufferu, (2) a (3) potom jako systém s automatickým bufferováním.

Poznamenejme, že u systémů s automatickým bufferováním odesílatel explicitně neví, kdo jeho zprávu po uložení do bufferu přijme. Jestliže je tato informace pro další výpočet potřebná, musí odesílatel s příjemcem přímo komunikovat. Uvažujme proces *P*, který zasílá zprávu procesu *Q* a může pokračovat ve výpočtu pouze v případě, že proces *Q* zprávu přijal.

Algoritmus procesu *P* potom obsahuje sekvenci příkazů:

```
send(Q,zprava);
receive(Q,zprava);
```

Algoritmus procesu *Q* obsahuje sekvenci příkazů:

```
receive(P,zprava);
send(P,"potvrzeni");
```

Výše uvedené procesy komunikují *asynchronně*.

Existují další výjimečné formy komunikace procesů:

- Proces, který produkuje zprávy, **nikdy** nečeká. Tzn., pokud příjemce nepřijímá, fronta je plná a producent vyslal další zprávu, je první zpráva ve frontě zničena. (problematické naprogramování, nutná přesná synchronizace)
- Proces, který vyslal zprávu, čeká, dokud nedostane odpověď. Některé operační systémy (např. *Thoth*) mají tento systém zpráv a mají implementovánu funkci **reply**(*P,zprava)*, kterou provádí potvrzení o přijetí. Rozdíl mezi funkcemi **send** a **reply** je ten, že po funkci

send je proces blokován, dokud nedostane **reply**, zatímco po provedení funkce **reply** pokračuje ve výpočtu.

Synchronní komunikace může být velmi jednoduše rozšířena na systém **volání vzdálené procedury** (*remote procedure call* – RPC). Systém RPC je konstruován tak, že volání podprogramu nebo procedur v jednoprocesorovém systému je na bázi systému zpráv, kdy odesílatel je blokován, dokud nedostane reply. Zpráva je volání procedury a reply vrací vypočtenou hodnotu. Dalším krokem v rozšiřování RPC je umožnit souběžně běžícím procesům, aby se mohly navzájem volat a vrcholem pyramidy je potom spouštění procesů na různých procesorech různých počítačů. Na principu RPC pracuje např. **Network File System** (NFS) umožňující sdílet souborové systémy nebo jejich části v síti a další aplikace.

5.5 Výjimečné situace

Systém zpráv je užitečný pro budování distribuovaného prostředí, kde jednotlivé procesy mohou být rozloženy na různých místech (různých výpočetních systémech). Je však zřejmé, že problémy vzniklé v souvislosti s komunikací procesů budou mnohem větší než na jednom jednoprocesorovém stroji.

U jednoho stroje je systém zpráv implementován ponejvíce ve sdílené paměti. V tom případě chyba ve spojení znamená chybu v systému. V distribuovaném prostředí je tomu jinak – zprávy jsou přenášeny po počítačové síti a chyba během přenosu v žádném případě nemusí nutně znamenat chybu OS.

Pokud dojde k chybě dat v centralizovaném nebo distribuovaném systému, musí být spuštěny opravné prostředky (prostředky ošetření výjimečné situace), které nějakým způsobem chybu odstraní. Následují některé výjimečné situace, které musí systém umět ošetřit v případě užití RPC apod.

5.5.1 Proces byl ukončen

Odesílatel nebo příjemce může být ukončen dříve, než byla zpracována zpráva. Díky takové události by mohly vzniknout zprávy, které nikdy nebudou přijaty, nebo zablokované procesy čekající na zprávu, která nikdy nebude odeslána. Dvě možné situace:

- Proces *P* čeká na zprávu od procesu *Q*, který byl zrušen. Pokud by nebyly uplatněny prostředky ošetření výjimečné situace, proces *P* by byl blokován navěky. OS musí v tomto případě proces *P* ukončit, nebo mu sdělit, že na zprávu od procesu *Q* už čekat nemusí.
- Proces *P* odeslal zprávu procesu *Q*, který byl zrušen. V systému s automatickým bufferováním se nic neděje a proces *P* klidně pokračuje dál. Pokud ovšem proces *P* potřebuje vědět, že *Q* zprávu přijal, čeká na potvrzení přijetí. V systému bez bufferování je proces *Q* blokován vždy. Opravné mechanismy jsou totožné s bodem (1).

5.5.2 Ztracená zpráva

Zpráva odeslaná procesem *P* procesu *Q* se ztratila kdesi v síti – chybou hardwaru nebo komunikačního spojení. Základní metody ošetření této události:

- OS je schopen detekovat tuto událost a zprávu poslat znovu.
- Odesílající proces je schopen detekovat tuto událost a zprávu odeslat znovu, je-li to zapotřebí.
- OS je schopen detekovat tuto událost a upozorní odesílající proces, že zpráva byla ztracena. Odesílající proces potom může udělat, co uzná za vhodné.

Při použití některých síťových protokolů není nutné detekovat ztracené zprávy OSem, síťový protokol to dělá sám a uživatelský program získává informace přímo od protokolu.

Pro detekci ztracené zprávy je nejpoužívanější metodou metoda detekce pomocí *timeoutu*. Po přijetí každé zprávy je **vždy** odesláno potvrzení o přijetí. OS nebo uživatelský program mohou specifikovat dobu, během které očekávají potvrzení o přijetí odeslané zprávy. Pokud tato doba uplyne, aniž by potvrzení dorazilo, zpráva je považována za ztracenou a je opětovně odeslána. Samozřejmě se může stát, že ani v tomto případě nedošlo ke ztracení zprávy, ta pouze procházela sítí „déle než obvykle". Za této situace je v síti více kopií téže zprávy a musí existovat mechanismy i pro ošetření tohoto stavu.

5.5.3 *Porušená zpráva*

Zpráva byla doručena příjemci, ale cestou došlo k jejímu poškození (např. v důsledku ruchu v síti). Důsledky této události jsou totožné se ztracením zprávy a většinou OS znovu posílá originál porušené zprávy.

Mechanismus detekce poškození spočívá v kontrolním součtu zprávy (parita nebo CRC).

5.6 Komunikace v OS UNIX

5.6.1 *Sockety*

Nejvýznamnější IPC mechanismus v OS UNIX je *pipe*. Pipe (roura) provádí spolehlivý jednosměrný bytový proud dat (stream). Většinou je implementován jako běžný soubor s malými odlišnostmi. Nemá jméno v systému souborů a je vytvořen během volání jádra **pipe**. Má pevnou délku, a pokud ho producent zaplní, je pozastaven, dokud příjemce všechna data ze souboru nezpracuje.

Výhoda malého rozsahu (většinou 4kB) souboru typu roura je, že je zřídka ukládán na disk a většinou je v cache paměti.

Ve 4.3BSD UNIXU je pipe implementován jako speciální *socket*. Sockety provádí hlavní interface nejen pro mechanismy jako je např. pipe, ale i pro síťové prostředky.

Socket je konečný bod komunikace. Používaný socket má přidělenou adresu. Povahu adresy podmiňuje *komunikační doména* socketu. Procesy komunikující v té samé komunikační doméně užívají stejný formát adresy.

Ve 4.3 BSD UNIXU jsou implementovány 3 komunikační domény:

- doména UNIX (AF_UNIX),
- doména Internet (AF_INET),
- doména Xerox Network services (AF_NS).

Adresa v doméně UNIX je plná (absolutní) cesta k souboru */alfa/beta/gama*. Procesy komunikující v doméně INTERNET užívají protokoly TCP/IP a IP adresu tj. 32bitové číslo coby adresu stroje a 32bitové číslo portu (bod spojení na hostu).

Existují různé *typy socketu*, které definují třídy služeb. Každý typ může nebo nemusí být implementován v komunikační doméně. Pokud daný typ implementován je, může být implementován množstvím komunikačních protokolů:

- **Stream socket:** Provádí spolehlivý duplexní postupný tok dat. Žádná data během tohoto spojení nejsou nikdy ztracena nebo duplikována. Tento typ komunikace je podporován v doméně Internet protokolem TCP, v doméně UNIX jako pipe.
- **Sequenced packet socket:** Obdoba stream socketu v doméně XEROX.
- **Datagram socket:** Tento socket zajišťuje přenos zpráv libovolné délky v libovolném směru. Není zaručeno, že zpráva bude vždy doručena ve stejném pořádku, jako byla odeslána nebo že nebude duplikována či doručena celá. Velikost původní zprávy je

doručena příjemci v jednom datagramu. Tento socket je podporován v doméně Internet protokolem UDP.

- **Reliable delivered message socket (Socket pro spolehlivě doručované zprávy):** Tento socket by zajištoval doručení zprávy při všech ostatních charakteristikách, jaké má datagram socket. Dosud není nikde podporován.
- **Raw socket:** Tento socket umožňuje procesu přímý přístup k jinému typu socketu. Přímý přístup není jen k protokolům výše uvedeným, ale je podporován přímý přístup ke všem protokolům nízké úrovně. Např. v doméně Internet dovoluje protokolu TCP přístup k IP protokolu apod. Užitečné pro vývoj nového protokolu.

Sockety jsou podporovány množinou volání jádra. Volání **socket** vytvoří novy socket. Při specifikaci volání je třeba uvést komunikační doménu, typ socketu a protokol, který bude tento socket využívat. Návratová hodnota volání je celé číslo *socket descriptor*, shodné s příslušným descriptorem souboru. Socket descriptor je index v poli otevřených "souborů" na příslušnou položku.

Pro adresování socketu jiným procesem musí mít socket jméno. To mu přidělí volání jádra **bind**. Obsah a délka jména socketu závisí na jeho typu. Inicializace spojení se provádí voláním jádra **connect**.

Mnoho procesů komunikuje pomocí IPC modelem *klient – server*. V tomto modelu provádí server služby pro klienta. Je-li služba k dispozici, server naslouchá na její veřejné adrese a klient užívá k navázání spojení volání **connect**, jak již bylo uvedeno.

Proces server pak užije volání **socket** pro vytvoření nového socketu a **bind** k napojeni adresy k novému socketu. Volání **listen** říká jádru, že server proces je schopen akceptovat připojení klienta a definuje počet nevyřízených volání, které má jádro uchovávat, než bude server schopen je vyřídit. Nakonec dojde k volání **accept**, kterým server dává vědět, že je schopen akceptovat přímé spojení s klientem.

Jak volání **listen**, tak i **accept** mají za argument descriptor socketu. **Accept** vrací nový descriptor socketu, který provádí nové spojení. Původní socket je stále otevřen pro možná budoucí další spojení. Většinou po akceptování požadavku klienta voláním **accept** server provede **fork** a vytvoří nový proces, který bude klienta obsluhovat a sám dále naslouchá dalším požadavkům.

Když je spojení na příslušném typu socketu ustanoveno, jsou známy adresy obou komunikujících procesů, které jsou třeba pro výměnu dat mezi nimi. K tomu slouží volání **read** a **write**.

Nejjednodušší cesta ke zrušení spojení a asociovaného socketu je v užití volání **close** na patřičný descriptor socketu. Je-li třeba uzavřít pouze jeden směr v obousměrné komunikaci procesů, je možno užít volání **shutdown**.

Některé typy socketů, jako např. datagram socket nepodporují přímé spojení. V tom případě musí být jednotlivé datagramy adresovány individuálně a k tomu slouží volání **sendto** a **recvfrom**. Obě požadují jako argument descriptor socketu, odkaz na buffer, délku zprávy a ukazatel do bufferu adres, který obsahuje adresy pro zaslání datagramu prostřednictvím **sendto**. Buffer je naplňován návratovými adresami z přijatých datagramů prostřednictvím volání **recvfrom**.

Volání jádra **select** je užito při multiplexování přenášených zpráv na různé descriptory souborů nebo socketů. Užívá se, pokud jeden server proces naslouchá více klientům a je voláním **fork** vytvořen zvláštní proces pro každého klienta po navázání spojení. Při každém navázání spojení volá server **socket**, **bind** a **listen** a poté **select** pro multiplexování zpráv na všechny jeho sockety. Jestliže **select** zaznamená aktivitu na nějakém descriptoru, provede

server **accept** a pomocí **fork** vytvoří proces pro vyřízení požadavku na descriptoru, který vrací **accept**. Rodičovský server proces pak opět provede **listen**.