KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

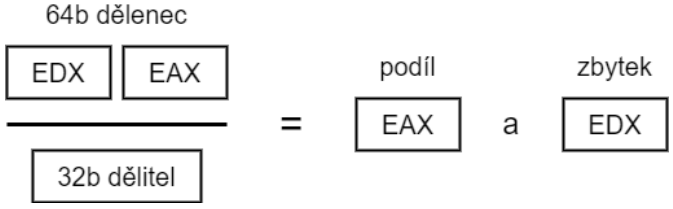
Základní práce s procesy v unixech

KMI/OS1 Operační systémy I

Mgr. Markéta Trnečková, Ph.D.
www.marketa-trneckova.cz

Písemka – časté chyby

- Dělení `eax = edx:eax / op1; edx = edx:eax % op1`:
  `idiv r/m`



- K čemu v .h souboru slouží
  `#ifndef KOD_H`
  `#define KOD_H`

Písemka – časté chyby

- Zjednodušte

```
foo:
    ;kod
    sub esi, 1
    cmp esi 0
    jne foo
```

Příklad

Jak vynulovat registr? Pomocí `sub`?

Písemka – časté chyby

- `zamena()`, `exchange()`
  - zapomenuté `global` a `section .text`
  - Použití registru `rbx` a nezajištění, že na konci má stejnou hodnotu jako na začátku
  - Použití špatné velikosti registru
    `char` = 1 byte – `al, cl`
    `short` = 2 byte – `ax, cx`

Základní práce s procesy v unixech

Učební text ke cvičení:
`https://phoenix.inf.upol.cz/~krajcap/courses/2025LS/OS1/tutorial09.pdf`

Základní práce s procesy v unixech

- Vznik unix 60. - 70. léta minulého století
- = omezený výkon počítačů
- výrazně jednodušší práce s procesy než ve Windows
- **Proces** = základní entita vykonávající program
- proces = instance běžícího programu
- programy v unixech = malé, plnící jeden účel
- do větších celků se skládají pomocí skriptů v *shellu*

Shell

- Shell slouží jako rozhraní pro interaktivní práci uživatelů formou příkazového řádku
- jedná se o plnohodnotný programovací jazyk == v shellu je možné sestavit program, který spouští a propojuje menší programy do větších programů
- **Podrobná dokumentace k jednotlivým programům/nástrojům unixu** – program `man`

Příklad

Vyzkoušejte `man ls`.
Vypíše se manuál k příkazu `ls`.

Příklad

Vyzkoušejte `man fork`.
`fork()` je funkce standardní knihovny.

Identifikace a organizace procesů

- procesy tvoří stromovou hierarchii
- v kořeni je proces `init` – první proces po spuštění OS
- procesy mají id – `pid`
- **Id aktuálního procesu** – funkce `pid_t getpid()` z knihovny `unistd.h`
- **Informace o běžících procesech** – příkaz `ps`
  - bez parametrů vypíše seznam procesů spojených s aktuálním terminálem
  - `ps -u` – vypíše všechny procesy daného uživatele
  - `ps aux` – vypíše podrobné informace o všech procesech

Příklad

Vyzkoušejte si příkaz `ps`.

- **Strom procesů**: příkaz `pstree`

Identifikace a organizace procesů

Příklady

1. Napište program, který po svém spuštění vypíše své pid a bude provádět nějakou činnost, nedojde k jeho ukončení.
2. Identifikujte program pomocí nástroje `ps`.
3. Identifikujte program pomocí nástroje `pstree`.
4. Program ukončete pomocí kombinace kláves `ctrl+c`.

Vytvoření nového procesu

- **Vytvoření nového procesu**: systémové volání`fork`
- Funkce v C: `pid_t fork()` z knihovny `unistd.h`
- potomek procesu, který zavolal `fork` – vznikne klon rodičovského procesu
- rodič i potomek vykonávají stejný kód a každý má vlastní kopii dat
- potomek má přiřazené nové unikátní id a rodič i potomek mají svůj oddělený paměťový prostor
- po zavolání `fork()` existují v paměti dva procesy vykonávající stejný kód, je potřeba je nějakým způsobem rozlišit
- tomu slouží návratová hodnota této funkce
  Pokud se vykonávaný kód nachází v rodiči, funkce `fork` vrací pid potomka, pokud se vykonávaný kód nachází v potomkovi, vrací funkce `fork()` hodnotu 0
- Pokud `fork()` vrátí zápornou hodnotu, došlo k chybě

Příklad

Vytvořte program, který zavolá `fork()`, a ověřte výše popsané chování.

Spuštění jiného programu

- Vytvoření identické kopie procesu má pouze omezené použití a v praxi často potřebujeme spustit jiný program
- K tomu slouží systémové volání `exec` – do paměti nahraje kód programu a začne jej vykonávat
- V jazyce C funkce z knihovny `unistd.h`

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *filename, char *const argv [], char *const envp[]);
```

Spuštění jiného programu

- Funkce `execl` a `execv` se liší ve formě argumentů
- První 3 jsou fce s proměnlivým počtem argumentů
- Poslední 3 přebírají argumenty v poli
- V obou případech jsou argumenty ukončeny hodnotou NULL
- S výjimkou `execlp` a `execvp` se uvádí plná cesta k souboru s programem
- Funkce `execlp` a `execvp` umí pracovat jen se jménem programu a vyhledávají program v systémové cestě
- Funkce `execle` a `execve` předávají spouštěnému programu proměnné prostředí ve tvaru `promenna=hodnota`
- Ve všech případech, pokud došlo k selhání funkce, je vrácena záporná hodnota.
- Pozor, jako první prvek pole argumentů, je předávaný název programu.

Spuštění jiného programu
Příklady

5. S použitím `exec` spusťte program `uname --all`.
6. Ověřte, že pokud volání `exec` neselže, je nahrazen aktuální kód programu.
7. Spojte volání `fork` a `exec` tak, aby rodič vykonával nějakou činnost, zatímco potomek zavolá `uname --all` a ukončí se.

Ukončení procesu

- Ukončení procesu systémové volání `exit`
- `void exit(int)`, argument odpovídá návratové hodnotě, která je předána rodiči
- Používat by se měly jen hodnoty 0 až 127, další hodnoty mají svůj vyčleněný význam
- def. v `stdlib.h`
- funkce `abort()` ukončí aktuálně běžící program a uloží na disk obraz paměti (tzv. core dump) – to můžeme využít k další analýze programu
- **Ukončení programu jiným procesem** `kill`
  `kill <pid>`
- Zašle signál procesu, aby se ukončil

Ukončení procesu
Příklady

8. Upravte program(y) z předchozích úkolů tak, aby na neúspěšné volání `fork` nebo `exec` zareagovaly voláním `exit` nebo `abort`
9. Ukončete nějaký běžící (ideálně nějaký méně důležitý) proces s pomocí nástroje `kill`. Případně spusťte na pozadí nějaký proces a ten ukončete.

Čekání

- Při souběžné práci s procesy nemáme garantováno, že budou prováděny v určitém pořadí, ani jak bude jejich souběh řešen
- Pro testování je výhodné proces na nějakou dobu uspat
- `unsigned sleep(unsigned seconds)`
- Uspí aktuální proces na několik sekund

Já jsem používala v předchozích příkladech.

Čekání

Čekání na potomka

- Někdy v rodiči potřebujeme počkat, než doběhne kód potomka
- `pid_t wait(int *status)` – čeká než skončí potomek
- `pid_t waitpid(pid_t pid, int *status, int options)` – čeká než skončí potomek s daným pid
- Obě z knihovny `sys/wait.h`
- Obě vrací pid potomka, a pokud ukazatel status není `NULL`, jsou vráceny informace o ukončení potomka
- S těmi se pracuje pomocí maker:
  - `WIFEXITED(status)` – vrací nenulové číslo, pokud potomek skončil normálně
  - `WEXITSTATUS(status)` – vrací návratový kód potomka (dá se použít jen pokud je `WIFEXITED(status)` nenulová hodnota)
  - `WIFSIGNALED(status)` – vrací nenulové číslo, pokud byl potomek ukončen signálem
  - `WTERMSIG(status)` – vrací číslo signálu, který proces ukončil (dá se použít jen pokud je `WIFSIGNALED(status)` nenulová hodnota)

Zombie procesy

- Pokud potomek skončí a rodič na něj nečeká pomocí `wait`, vznikne **zombie proces**
- zombie proces – proces který skončil, ale ještě v systému existuje, dokud si rodičovský proces nevyzvedne informace o jeho ukončení pomocí `wait`
- Pokud si rodič tyto informace nikdy nevyzvedne (ani po svém ukončení), zombie proces je adoptován procesem `init`, který jej odstraní ze systému

Zombie procesy

Příklad

10 Upravte předchozí úkol(y) tak, aby čekaly na dokončení potomka.