22. Procesy v Unixu

Obsah

- Vznik a zánik procesu
- Vlastnosti a prostředí procesu
- Vztah mezi procesem, úlohou a vláknem
- Synchronizace procesů
- Signály
- Plánování úloh

Vznik a zánik procesu

Definice procesu

Proces je instance běžícího programu. Na rozdíl od statického programu (který je jen souborem instrukcí uloženým na disku) je proces aktivní entita s vyhrazeným prostorem paměti, která je plánována operačním systémem ke spuštění na procesoru.

Vznik procesu

V Unixových systémech vznikají procesy pomocí dvojice systémových volání:

1. **fork()** - vytváří nový proces jako kopii rodičovského procesu
   - Nový proces (potomek) je téměř identickou kopií procesu, který zavolal fork() (rodič)
   - Potomek má vlastní PID, ale sdílí zdrojový kód s rodičem
   - Po fork() oba procesy pokračují od stejného bodu v kódu
   - Hlavní rozdíly: PID, PPID (PID rodiče), návratová hodnota fork()
2. **exec()** - nahrazuje aktuální proces novým programem
   - Nahraje nový program do paměti procesu a začne ho vykonávat
   - Původní program je zcela nahrazen
   - Existuje několik variant: execl(), execv(), execle(), execve(), execlp(), execvp()

Typická sekvence při spouštění nového programu v Unixu:

```
1. Rodičovský proces zavolá fork()
2. Vznikne nový proces (potomek)
3. Potomek zavolá exec() k nahrání a spuštění nového programu
4. Rodičovský proces typicky čeká na dokončení potomka pomocí wait() nebo waitpid()
```

Zánik procesu

Proces může zaniknout několika způsoby:

1. **Normální ukončení**
   - Proces dokončí svou činnost a vrátí návratovou hodnotu pomocí exit() nebo return z main()
   - Návratový kód 0 typicky značí úspěšné provedení, nenulové hodnoty označují chybu
2. **Ukončení chybou**
   - Neošetřená výjimka nebo signál, který způsobí pád procesu
   - Typické signály: SIGSEGV (Segmentation fault), SIGBUS, SIGILL, SIGFPE
3. **Ukončení signálem**
   - Proces je ukončen externím signálem (např. SIGKILL, SIGTERM)
   - Může být vyvolán uživatelem (příkaz kill) nebo operačním systémem
4. **Zombie procesy**
   - Když proces skončí, stále zůstává záznam v tabulce procesů
   - Rodiče jsou zodpovědní za "úklid" svých potomků pomocí wait() nebo waitpid()
   - Pokud rodič nevykoná wait(), potomek zůstane jako zombie

Průběh zániku procesu

1. Proces zavolá exit() nebo return z main()

2. Systém provede úklid prostředků (uzavře soubory, uvolní paměť)
3. Proces se stane zombie procesem (udržuje PID a návratový kód pro rodiče)
4. Rodiči je zaslán signál SIGCHLD informující o ukončení potomka
5. Rodič zavolá wait() nebo waitpid() pro získání návratového kódu a úplné odstranění procesu
6. Pokud rodič zemře před potomkem, proces init (PID 1) "adoptuje" potomky a vykoná wait()

Vlastnosti a prostředí procesu

Základní atributy procesu

1. **PID (Process ID)**
   - Unikátní identifikátor procesu (obvykle 5ciferné číslo)
   - První proces (init nebo systemd) má PID 1
2. **PPID (Parent Process ID)**
   - PID rodičovského procesu
   - Pokud rodič zemře, PPID se změní na 1 (init)
3. **UID/EUID (User ID/Effective User ID)**
   - ID uživatele, který proces vlastní nebo pod jehož právy proces běží
   - EUID může být jiné než UID (např. při použití setuid programů)
4. **GID/EGID (Group ID/Effective Group ID)**
   - ID primární skupiny procesu
   - Podobně jako u UID/EUID
5. **Stav procesu**
   - **Running**: proces běží nebo je připraven k běhu
   - **Blocked/Waiting**: proces čeká na událost (I/O, signál)
   - **Stopped**: proces byl pozastaven signálem
   - **Zombie**: proces dokončil běh, ale jeho rodič nevykonal wait()
6. **Priorita**
   - Určuje, kdy bude proces spuštěn ve srovnání s ostatními procesy
   - Může být upravena příkazy nice a renice
7. **Pracovní adresář**
   - Aktuální adresář procesu, kde proces hledá soubory při použití relativních cest
8. **Kořenový adresář**
   - Pomocí chroot může být změněn kořenový adresář procesu (pro omezení přístupu)
9. **Otevřené soubory a popisovače souborů**
   - Seznam všech otevřených souborů používaných procesem
   - Standardní vstupy/výstupy: stdin (0), stdout (1), stderr (2)
10. **Maska tvorby souborů (umask)**
    - Určuje výchozí práva pro nově vytvořené soubory

Virtuální paměť procesu

Každý proces má vlastní virtuální adresový prostor, který obsahuje:

1. **Text (kód)** - spustitelné instrukce programu (read-only)
2. **Data** - inicializované globální a statické proměnné
3. **BSS (Block Started by Symbol)** - neinicializované globální a statické proměnné
4. **Heap** - dynamicky alokovaná paměť (malloc, new)
5. **Stack** - zásobník pro lokální proměnné, návratové adresy funkcí
6. **Sdílené knihovny** - kód a data sdílených knihoven

Prostředí procesu

Prostředí procesu obsahuje proměnné prostředí, které proces zdědí od svého rodiče a které mohou ovlivnit jeho chování:

1. **Proměnné prostředí**
   - PATH - cesta pro hledání spustitelných souborů
   - HOME - domovský adresář uživatele

    - USER - jméno uživatele
    - SHELL - aktuální shell
    - TERM - typ terminálu
    - LANG, LC_* - lokalizační nastavení
2. **Přístup k prostředí v programech**
    - V C/C++:
        - extern char **environ
        - getenv() a setenv() funkce
    - V shellu:
        - export VARIABLE=value
        - echo $VARIABLE

Běh na popředí a pozadí

Procesy v Unixu mohou běžet v popředí nebo na pozadí:

1. **Běh v popředí**
    - Proces získává vstupy z klávesnice a vypisuje data na obrazovku
    - Shell čeká na dokončení procesu před zobrazením nového promptu
    - Výchozí způsob spuštění příkazů
2. **Běh na pozadí**
    - Proces běží nezávisle na terminálu
    - Shell ihned zobrazí nový prompt a uživatel může pokračovat v práci
    - Spouští se přidáním znaku `&` na konec příkazu nebo pomocí příkazu `bg`
    - Přepnutí z popředí na pozadí: Ctrl+Z (pozastavení) a pak `bg`
    - Přepnutí z pozadí na popředí: `fg`

Vztah mezi procesem, úlohou a vláknem

Proces

- Nejzákladnější jednotka běžícího programu
- Má vlastní adresový prostor, PID a prostředky
- Procesy jsou izolované navzájem (jeden proces nemůže přímo přistupovat k paměti jiného)
- Komunikace mezi procesy vyžaduje speciální mechanismy (IPC - Inter-Process Communication)

Úloha (Job)

- V kontextu shellu je úloha nadmnožina procesů
- Může představovat samostatný příkaz, skript nebo řetězec příkazů spojených pomocí pipe (|)
- Příkazy pro správu úloh:
    - `jobs` - výpis běžících úloh
    - `fg %n` - přesunutí úlohy n do popředí
    - `bg %n` - spuštění pozastavené úlohy n na pozadí
    - `kill %n` - ukončení úlohy n
    - `stop %n` - pozastavení úlohy n

Vlákno (Thread)

- Lehčí jednotka provádění v rámci procesu
- Všechna vlákna v rámci procesu sdílejí stejný adresový prostor, popisovače souborů a další prostředky
- Každé vlákno má však vlastní:
    - Zásobník (stack)
    - Registry procesoru včetně programového čítače
    - Stav vlákna
    - Thread ID (TID)
- Výhody vláken oproti procesům:
    - Menší režie při vytváření a přepínání

    - Jednodušší sdílení dat (všechna vlákna vidí stejnou paměť)
    - Efektivnější komunikace (není potřeba IPC)
  - Implementace vláken v Unixu:
    - POSIX Threads (pthreads) - standardní API pro práci s vlákny
    - Jádro spravuje vlákna samostatně (každé vlákno dostává čas CPU)

```c
// Příklad vytvoření vlákna v C pomocí pthreads
#include <pthread.h>

void* thread_function(void *arg) {
    // Kód vlákna
    return NULL;
}

int main() {
    pthread_t thread_id;
    int result = pthread_create(&thread_id, NULL, thread_function, NULL);

    // Čekání na dokončení vlákna
    pthread_join(thread_id, NULL);

    return 0;
}
```

Druhy procesů

1. **Normální procesy**
   - Standardní procesy spouštěné uživatelem
   - Mají proces předka (často shell)
   - Když doběhnou, předek je informován o jejich ukončení
2. **Sirotci (Orphan)**
   - Proces, jehož rodič skončil dříve než on sám
   - Takový proces je "adoptován" procesem init (PID 1), který se stane jeho novým rodičem
   - Init automaticky vykonává wait() pro všechny své potomky při jejich ukončení
3. **Zombie procesy**
   - Proces, který ukončil svůj běh, ale jeho rodič nevykonal wait()
   - Zabírá místo v tabulce procesů, ale žádné jiné prostředky
   - Zombie proces nelze ukončit ani pomocí SIGKILL
   - Problém nastává, když rodičovský proces vytváří mnoho zombie procesů a nečistí je
4. **Démoni (Daemon)**
   - Procesy běžící na pozadí nezávisle na terminálu
   - Typicky systémové služby (webový server, databáze, sshd)
   - Obvykle se spouští při startu systému
   - Názvy démonů často končí "d" (sshd, httpd, crond)
   - Vytvoření démona typicky zahrnuje:
     - Odpojení od terminálu (setsid())
     - Změnu pracovního adresáře (chdir())
     - Uzavření standardních vstupů/výstupů nebo jejich přesměrování

Sledování procesů

Pro zobrazení a sledování procesů slouží několik nástrojů:

1. **ps** - zobrazí aktuální procesy

```
ps -ef    # zobrazí všechny procesy ve formátu full
ps aux    # alternativní formát s využitím CPU a paměti
ps -eLf   # zobrazí i vlákna
```

2. **top** - interaktivní nástroj pro sledování běžících procesů

- Zobrazuje procesy seřazené podle využití CPU
- Umožňuje interaktivní zabíjení a změnu priority procesů

3. **htop** - vylepšená verze top
    - Barevné rozhraní
    - Pokročilá interakce pomocí myši
    - Stromové zobrazení procesů

Synchronizace procesů

Synchronizace je potřebná, když více procesů nebo vláken přistupuje ke sdíleným prostředkům. Cílem je zabránit: - Race condition (závody) - výsledek závisí na pořadí vykonávání - Deadlock (uváznutí) - procesy čekají na prostředky držené jinými procesy - Starvation (vyhladovění) - proces nikdy nedostane přístup k prostředku

Základní synchronizační mechanismy

1. **Semafory**
    - Celočíselná proměnná s atomickými operacemi P (down/wait) a V (up/signal)
    - Binární semafor (hodnoty 0 a 1) nebo počítací semafor (libovolná nezáporná hodnota)
    - Používá se pro řízení přístupu ke sdíleným prostředkům

```c
// Příklad použití semaforu v C (POSIX)
#include <semaphore.h>

sem_t mutex;
sem_init(&mutex, 0, 1);  // Inicializace binárního semaforu

// Proces 1
sem_wait(&mutex);  // Zamknutí (P operace)
// Kritická sekce - přístup ke sdílenému prostředku
sem_post(&mutex);  // Odemknutí (V operace)
```

2. **Mutexy**
    - Jednodušší verze binárního semaforu
    - Může být zamknut pouze procesem, který ho zamknul
    - Používá se především pro synchronizaci vláken

```c
// Příklad použití mutexu v C (POSIX)
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mutex);
// Kritická sekce
pthread_mutex_unlock(&mutex);
```

3. **Condition Variables (Podmínkové proměnné)**
    - Umožňují procesům/vláknům čekat na určitou podmínku
    - Vždy se používají s mutexem

```c
// Příklad použití condition variable v C (POSIX)
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&mutex);
while (!condition)
    pthread_cond_wait(&cond, &mutex);  // Automaticky odemyká mutex a čeká
// Kritická sekce - podmínka je splněna
pthread_mutex_unlock(&mutex);

// V jiném vlákně:
pthread_mutex_lock(&mutex);
```

```
condition = true;
pthread_cond_signal(&cond);  // Probudí jedno čekající vlákno
pthread_mutex_unlock(&mutex);
```

4. **Message Queues (Fronty zpráv)**
   - Umožňují procesům posílat a přijímat zprávy
   - Asynchronní komunikace

```
// Příklad použití POSIX message queue
#include <mqueue.h>

mqd_t queue;
queue = mq_open("/myqueue", O_CREAT | O_RDWR, 0644, NULL);

// Odesílání zprávy
mq_send(queue, message, strlen(message), 0);

// Příjem zprávy
mq_receive(queue, buffer, sizeof(buffer), NULL);
```

5. **Shared Memory (Sdílená paměť)**
   - Nejrychlejší způsob komunikace mezi procesy
   - Procesy mohou přímo přistupovat ke stejné paměti
   - Vyžaduje explicitní synchronizaci (semafory, mutexy)

```
// Příklad použití POSIX sdílené paměti
#include <sys/mman.h>
#include <fcntl.h>

int fd = shm_open("/myshm", O_CREAT | O_RDWR, 0644);
ftruncate(fd, 4096);  // Nastavení velikosti

void *shared_memory = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

6. **File Locks (Zámky souborů)**
   - Používané pro synchronizaci přístupu k souborům
   - Exkluzivní (write) nebo sdílené (read) zámky

```
// Příklad použití file lock v C
#include <fcntl.h>

struct flock lock;
lock.l_type = F_WRLCK;  // Exkluzivní zámek
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = 0;  // Uzamknout celý soubor

fcntl(fd, F_SETLKW, &lock);  // Zamknutí (čeká, pokud je zamčeno)
// Práce se souborem
lock.l_type = F_UNLCK;
fcntl(fd, F_SETLK, &lock);  // Odemknutí
```

Signály

Signály jsou softwarová přerušení zasílaná procesům, která slouží k informování procesu o určité události nebo k vyžádání určité akce.

Základní charakteristiky signálů

- Asynchronní (mohou přijít kdykoliv)
- Nižší úroveň komunikace než IPC (Inter-Process Communication)
- Omezené množství informací (jen typ signálu)
- Proces může signál ignorovat, obsloužit vlastní funkcí nebo nechat výchozí akci

Nejčastější signály

Signál	Číslo	Popis	Výchozí akce	Ignorovatelný
SIGHUP	1	Zavěšení terminálu	Ukončení	Ano
SIGINT	2	Přerušení z klávesnice (Ctrl+C)	Ukončení	Ano
SIGQUIT	3	Quit z klávesnice (Ctrl+\)	Core dump	Ano
SIGKILL	9	Okamžité ukončení	Ukončení	Ne
SIGSEGV	11	Segmentation fault	Core dump	Ne
SIGTERM	15	Ukončovací signál	Ukončení	Ano
SIGSTOP	17,19,23	Zastavení procesu	Zastavení	Ne
SIGTSTP	20	Stop z klávesnice (Ctrl+Z)	Zastavení	Ano
SIGCONT	18,25	Pokračování po zastavení	Pokračování	Ano
SIGCHLD	20,17	Změna stavu potomka	Ignorování	Ano
SIGUSR1	30,10	Uživatelský signál 1	Ukončení	Ano
SIGUSR2	31,12	Uživatelský signál 2	Ukončení	Ano

Odesílání signálů

Signály lze posílat několika způsoby:

1. **Příkaz kill**

```
kill -SIGTERM 1234  # Pošle signál SIGTERM procesu s PID 1234
kill -15 1234       # Stejné jako výše (použití čísla signálu)
kill -9 1234        # Pošle SIGKILL (nelze ignorovat)
killall firefox     # Pošle SIGTERM všem procesům s názvem firefox
```

2. **Klávesové zkratky v terminálu**
   - Ctrl+C - odešle SIGINT aktuálnímu procesu v popředí
   - Ctrl+Z - odešle SIGTSTP aktuálnímu procesu v popředí
   - Ctrl+ - odešle SIGQUIT aktuálnímu procesu v popředí

3. **Systémové volání kill()**

```
#include <signal.h>

kill(pid, SIGTERM);  // Pošle SIGTERM procesu s daným PID
```

4. **Funkce raise()**

```
#include <signal.h>

raise(SIGUSR1);  // Pošle signál aktuálnímu procesu
```

Zpracování signálů v programu

Proces může určit, jak se má zachovat při příchodu signálu:

1. **Ignorování signálu**

```
#include <signal.h>

signal(SIGINT, SIG_IGN);  // Ignorování Ctrl+C
```

2. **Použití výchozí akce**

```
signal(SIGINT, SIG_DFL);  // Obnovení výchozí akce
```

3. **Vlastní obsluha signálu**

```
#include <signal.h>

void signal_handler(int signum) {
```

```c
    printf("Přijat signál %d\n", signum);
}

int main() {
    // Registrace obslužné funkce pro SIGINT
    signal(SIGINT, signal_handler);

    // Moderní způsob s více možnostmi
    struct sigaction sa;
    sa.sa_handler = signal_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);

    // Hlavní smyčka programu
    while(1) {
        sleep(1);
    }

    return 0;
}
```

Plánování úloh

Plánování úloh v Unixových systémech umožňuje spouštět příkazy v předem definovaných časech nebo v pravidelných intervalech.

Jednorázové úlohy: at a batch

at Příkaz `at` umožňuje naplánovat jednorázové spuštění příkazu v určený čas v budoucnosti.

```
# Spuštění příkazu v konkrétním čase
at 14:30
at> echo "Je 14:30" > /tmp/at_test
at> ^D

# Spuštění příkazu za určitou dobu
at now + 5 minutes
at> banner "Čas vypršel!"
at> ^D

# Spuštění příkazu v konkrétní datum a čas
at 10:00 Dec 25
at> mail -s "Veselé Vánoce" user@example.com < /tmp/vanoce.txt
at> ^D
```

Další příkazy související s `at`: - `atq` - zobrazení naplánovaných úloh (nebo `at -l`) - `atrm` - odstranění naplánované úlohy (nebo `at -r`)

batch Příkaz `batch` je podobný `at`, ale spouští úlohy, když je zatížení systému dostatečně nízké.

```
batch
batch> find / -name "*.tmp" -delete
batch> ^D
```

Pravidelné úlohy: cron

Démon `cron` je systémová služba, která umožňuje spouštění příkazů v pravidelných intervalech.

crontab Soubor crontab obsahuje seznam příkazů, které mají být spuštěny v určených časech:

```
# Zobrazení aktuálního crontabu
crontab -l

# Editace crontabu
crontab -e

# Odstranění crontabu
crontab -r
```

Formát záznamu v crontabu:

```
# m h dom m dow command
# | | |   | |   |
# | | |   | |   +-- Příkaz k vykonání
# | | |   | +------ Den v týdnu (0-7) (neděle = 0 nebo 7)
# | | |   +-------- Měsíc (1-12)
# | | +------------ Den v měsíci (1-31)
# | +-------------- Hodina (0-23)
# +---------------- Minuta (0-59)
```

Příklady záznamů v crontabu:

```
# Spuštění každou minutu
* * * * * echo "Minuta uplynula" >> /tmp/minuta.log

# Spuštění v 8:15 každý den
15 8 * * * echo "Je 8:15" | mail -s "Čas" user@example.com

# Spuštění každý pracovní den v 18:00
0 18 * * 1-5 /usr/local/bin/backup.sh

# Spuštění první den v měsíci o půlnoci
0 0 1 * * /usr/local/bin/monthly-report.sh

# Spuštění každou neděli ve 23:30
30 23 * * 0 /usr/local/bin/weekly-cleanup.sh

# Spuštění každé dvě hodiny
0 */2 * * * /usr/local/bin/check-services.sh
```

Speciální řetězce pro crontab:

```
@reboot     # Spuštění při startu systému
@yearly     # Spuštění jednou ročně (stejné jako "0 0 1 1 *")
@annually   # Stejné jako @yearly
@monthly    # Spuštění jednou měsíčně (stejné jako "0 0 1 * *")
@weekly     # Spuštění jednou týdně (stejné jako "0 0 * * 0")
@daily      # Spuštění jednou denně (stejné jako "0 0 * * *")
@midnight   # Stejné jako @daily
@hourly     # Spuštění jednou za hodinu (stejné jako "0 * * * *")
```

Systémové crontaby

Kromě uživatelských crontabů existují i systémové crontaby:

- `/etc/crontab` - systémový crontab
- `/etc/cron.d/` - adresář s dalšími crontaby
- `/etc/cron.hourly/`, `/etc/cron.daily/`, `/etc/cron.weekly/`, `/etc/cron.monthly/` - adresáře s skripty, které mají být spouštěny v daných intervalech

Příklad praktického skriptu

Skript pro ukončení procesu:

```
#!/bin/bash
# ukonci.sh [cislo_procesu]

if [ $# -ne 1 ]; then
    echo "Špatný počet parametrů" >&2
    exit 2
fi

if kill -9 "$1" 2>/dev/null; then
    echo "OK" >&2
else
    echo "Proces se nepodařilo ukončit" >&2
    exit 1
fi
```

Použití:

```
./ukonci.sh 1234  # Ukončí proces s PID 1234
```

Příklady praktických příkazů

Manipulace s procesy

```
# Spuštění procesu na pozadí
command &

# Pozastavení běžícího procesu
# Stisknutí Ctrl+Z v terminálu

# Zobrazení úloh
jobs

# Přesun úlohy do popředí
fg %1  # Přesune úlohu č. 1 do popředí

# Spuštění pozastavené úlohy na pozadí
bg %1  # Spustí úlohu č. 1 na pozadí

# Ukončení procesu
kill 1234        # Pošle SIGTERM procesu s PID 1234
kill -9 1234     # Pošle SIGKILL (nelze ignorovat)
killall firefox  # Ukončí všechny procesy s názvem firefox

# Změna priority procesu
nice -n 10 command     # Spustí příkaz s nižší prioritou
renice +10 -p 1234     # Změní prioritu běžícího procesu
```

Monitorování procesů

```
# Základní informace o procesech
ps -ef           # Všechny procesy, formát full
ps aux           # Všechny procesy, BSD formát
ps -eLf          # Včetně vláken

# Zobrazení stromové struktury procesů
pstree
```

```
ps -ejH          # Stromová struktura ve formátu
ps axjf          # Alternativní stromové zobrazení

# Interaktivní monitorování procesů
top              # Základní interaktivní monitor
htop             # Vylepšený interaktivní monitor (je třeba nainstalovat)
atop             # Pokročilý monitor systémových a procesových zdrojů

# Sledování procesu podle PID
top -p 1234      # Sledování konkrétního procesu
watch -n 1 "ps -p 1234 -o pid,%cpu,%mem,cmd"  # Aktualizace každou sekundu

# Vyhledávání procesů
pgrep firefox    # Najde PID procesů odpovídajících vzoru
pidof firefox    # Podobné jako pgrep, ale přesnější shoda

# Informace o procesech
lsof -p 1234     # Výpis otevřených souborů procesu
ls -l /proc/1234/ # Podrobné informace o procesu v pseudo souborovém systému /proc
cat /proc/1234/cmdline # Zobrazení příkazové řádky, kterou byl proces spuštěn
cat /proc/1234/environ # Zobrazení proměnných prostředí procesu
cat /proc/1234/status  # Zobrazení aktuálního stavu procesu

### Příklady pokročilých skriptů pro práci s procesy

#### Monitorování a protokolování zátěže CPU

```bash
#!/bin/bash
# monitor_cpu.sh - monitoruje a zapisuje zátěž CPU
# Použití: ./monitor_cpu.sh [interval_v_sekundách] [počet_měření]

interval=${1:-5}  # Výchozí interval 5 sekund
count=${2:-12}    # Výchozí počet 12 měření
log_file="/tmp/cpu_log_$(date +%Y%m%d_%H%M%S).txt"

echo "Čas,CPU,Paměť,Běžící procesy" > "$log_file"

for ((i=1; i<=count; i++)); do
    timestamp=$(date +"%Y-%m-%d %H:%M:%S")
    cpu_usage=$(top -bn1 | grep "Cpu(s)" | awk '{print $2 + $4}')
    mem_usage=$(free | grep Mem | awk '{print $3/$2 * 100.0}')
    running_procs=$(ps -eo stat | grep -c "R")

    echo "$timestamp,$cpu_usage%,$mem_usage%,$running_procs" >> "$log_file"
    echo "Měření $i/$count: CPU: $cpu_usage%, Paměť: $mem_usage%, Běžící: $running_procs"

    sleep "$interval"
done

echo "Monitoring dokončen. Výsledky uloženy v $log_file"
```

Automatické ukončení procesů podle využití paměti

```
#!/bin/bash
# kill_high_mem.sh - ukončí procesy, které využívají příliš mnoho paměti
# Použití: ./kill_high_mem.sh [limit_v_procentech]
```
```

```
# Výchozí limit 80% využití paměti
MEM_LIMIT=${1:-80}

echo "Vyhledávání procesů využívajících více než $MEM_LIMIT% paměti..."

ps aux | awk -v limit="$MEM_LIMIT" '
BEGIN {
    printf "%-10s %-10s %-8s %-s\\n", "PID", "UŽIVATEL", "PAMĚŤ%", "PŘÍKAZ"
}
{
    if (NR > 1 && $4 > limit) {
        printf "%-10s %-10s %-8s %-s\\n", $2, $1, $4, $11
        system("kill -15 " $2)
        print "Proces " $2 " byl ukončen signálem SIGTERM"
    }
}'

echo "Hotovo."
```

Skript pro sledování potomků procesu

```
#!/bin/bash
# process_children.sh - sleduje potomky zadaného procesu
# Použití: ./process_children.sh [PID]

if [ $# -ne 1 ]; then
    echo "Použití: $0 PID" >&2
    exit 1
fi

if ! [ -e "/proc/$1" ]; then
    echo "Proces s PID $1 neexistuje" >&2
    exit 2
fi

echo "Potomci procesu $1 ($(cat /proc/$1/comm)):"
echo "PID     PPID    CMD"

pgrep -P "$1" | while read -r child_pid; do
    cmd=$(cat /proc/"$child_pid"/cmdline | tr '\0' ' ' | head -c 50)
    printf "%-8s %-8s %s\\n" "$child_pid" "$1" "$cmd"

    # Rekurzivně vyhledat potomky tohoto potomka
    "$0" "$child_pid" | sed 's/^/  /'
done
```