OS – Procesy a vlákna

Tomáš Hudec

Tomas.Hudec@upce.cz

http://fei-as.upceucebny.cz/usr/hudec/vyuka/os/

Osnova

- procesy
  - příkazy pro procesy
  - procesy – systémová volání
- signály
  - signály – systémová volání
- vlákna
  - vlákna – posixová volání

Proces

- instance programu v paměti systému
  - program = recept (na dort)
  - proces = pečení (příprava dortu dle tohoto receptu)
- provádění: sekvenční × multiprogramming
- tabulka procesů, PCB (Process Control Block)
  - identifikace procesu, adresový prostor, stav, přidělené prostředky, práva, čas běhu, …
- hierarchie procesů
  - vztah rodič–potomek, strom

Vznik procesu

- původcem je jádro
  - při inicializaci systému
    - první proces (v posixových systémech obvykle **init**)
    - služby jádra (mikrojádrový OS)
- původcem je proces
  - systémové volání
    - uživatel zadá příkaz, který interpret (shell) zpracuje tak, že systémovým voláním vytvoří další proces
    - proces může být aktivován také stiskem tlačítka (dávkové a vestavěné systémy)

Zánik procesu

- dobrovolné ukončení (systémovým voláním)
  - normální – úloha byla dokončena
  - při detekování chyby – např. chybné vstupy
- nedobrovolné ukončení
  - fatální (neošetřená) chyba
    - př.: neplatná adresa, nepovolená instrukce, dělení nulou
    - jádro proces ukončí
  - zabití jiným procesem (uživatelem)

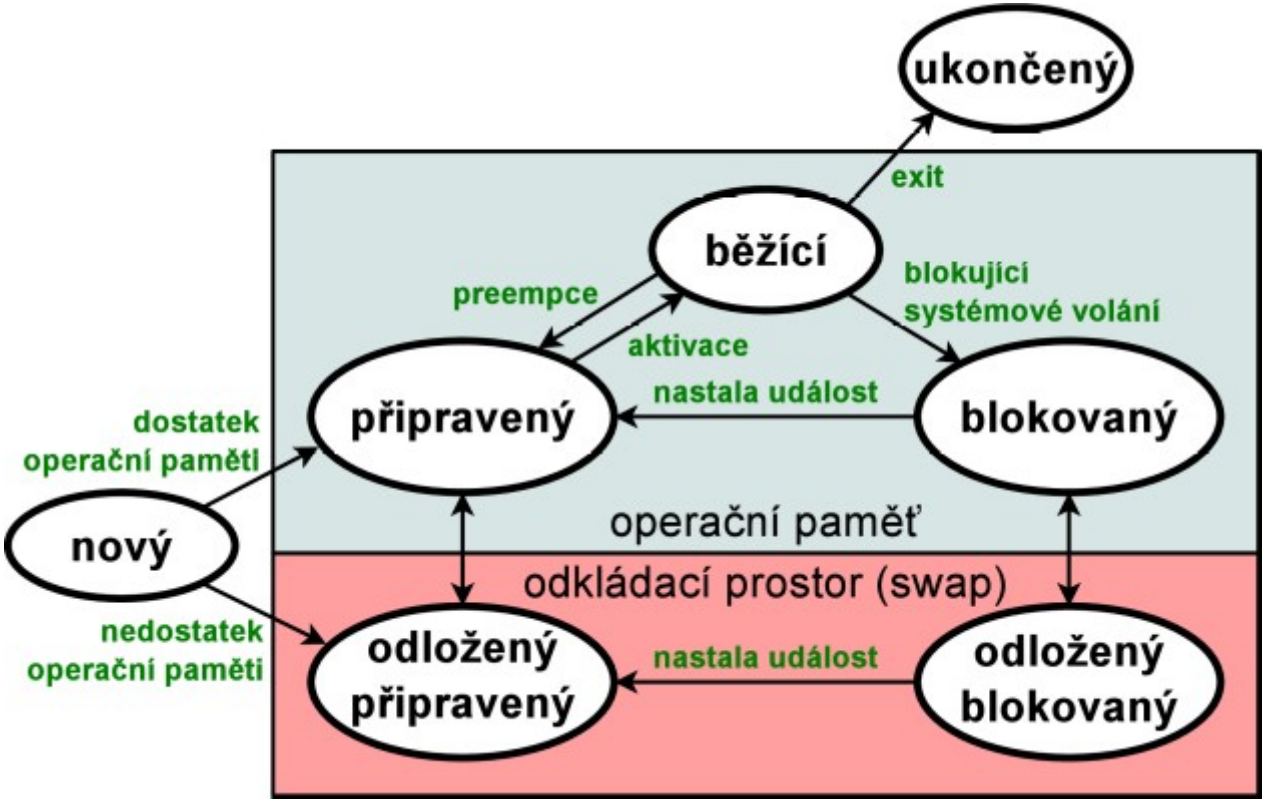
Stavy procesů

- třístavový model:
  - běžící (running) – používá CPU
  - připravený (ready) – pozastaven jádrem OS
  - blokovaný (blocked) – čekající na vnější událost
- scheduler – plánovač
  - vybírá připravený proces pro běh – přiděluje CPU
  - odebírá CPU běžícím procesům – preempce
  - může se aktivovat také při systémových voláních
  - řídí se plánovacím algoritmem

Rozšířené stavy procesů

- základní stavy lze rozšířit – 7stavový model
  - nový (new)
    - nelze zatím spustit (nemá ještě všechny prostředky)
  - ukončený (exit)
    - již se nemůže spustit, ale je třeba ještě držet v paměti jeho informace, např. kvůli účtování (accounting)
  - odložený blokovaný (blocked, suspended)
    - blokovaný proces zabírá paměť, více takových procesů pak ubírá paměť běžícím, proto se proces z paměti odloží na disk (swap)
  - odložený připravený (ready, suspended)
    - nastala již událost, na niž blokovaný proces čekal, ale proces je stále ještě na disku

Stavy procesů (obrázek)



sedmistavový model

Implementace procesů

- tabulka procesů nebo též PCB
  - adresový prostor: kód (text), stack, data, heap
  - přidělené prostředky: otevřené soubory, semafory, …
  - kontext (stav): registry CPU, mapování paměti
  - atributy: id, údaje plánovače, práva, časy, účtování, …
- při přerušení (např. při V/V) – přepnutí kontextu
  - uložení kontextu (stavu) procesu
  - obsluha ovladačem v jádře
  - plánovač rozhodne, který proces poběží poté

Manuálové stránky

- rozděleny do sekcí, uloženy v /usr/share/man

sekce	popis
1	základní uživatelské příkazy
2	služby jádra – systémová volání
3	knihovní funkce
5	formáty souborů, protokoly, struktury v C
7	různé (protokoly, normy apod.)
8	příkazy pro správu systému

- příkaz man(1) – číslo v závorce udává sekci
  - např. pro příkaz kill(1) a systémové volání kill(2):
  - **`man kill`**
  - **`man 2 kill`**

Příkazy pro procesy (UNIX) (1)

- seznam procesů: ps(1), pstree(1)
  - všechny procesy: **`ps -e [-f|-l]`**
- sledování zátěže: top(1), prstat(1)
- sledování délky fronty procesů: xload(1)
  - nebo uptime(1), také **`cat /proc/loadavg`**
- sledování využití procesoru: mpstat(1)
  - např. 10 měření po 1 sekundě: **`mpstat 1 10`**

Příkazy pro procesy (UNIX) (2)

- získání PID podle jména: pidof(1)
- nalezení podle kritérií: pgrep(1)
- poslání signálu: kill(1)
- poslání signálu podle jména: killall(1)
- poslání signálu dle kritérií: pkill(1)
- nastavení priority: nice(1), renice(1)
  - nice: od −20 (maximální priorita) do 19 (min.)

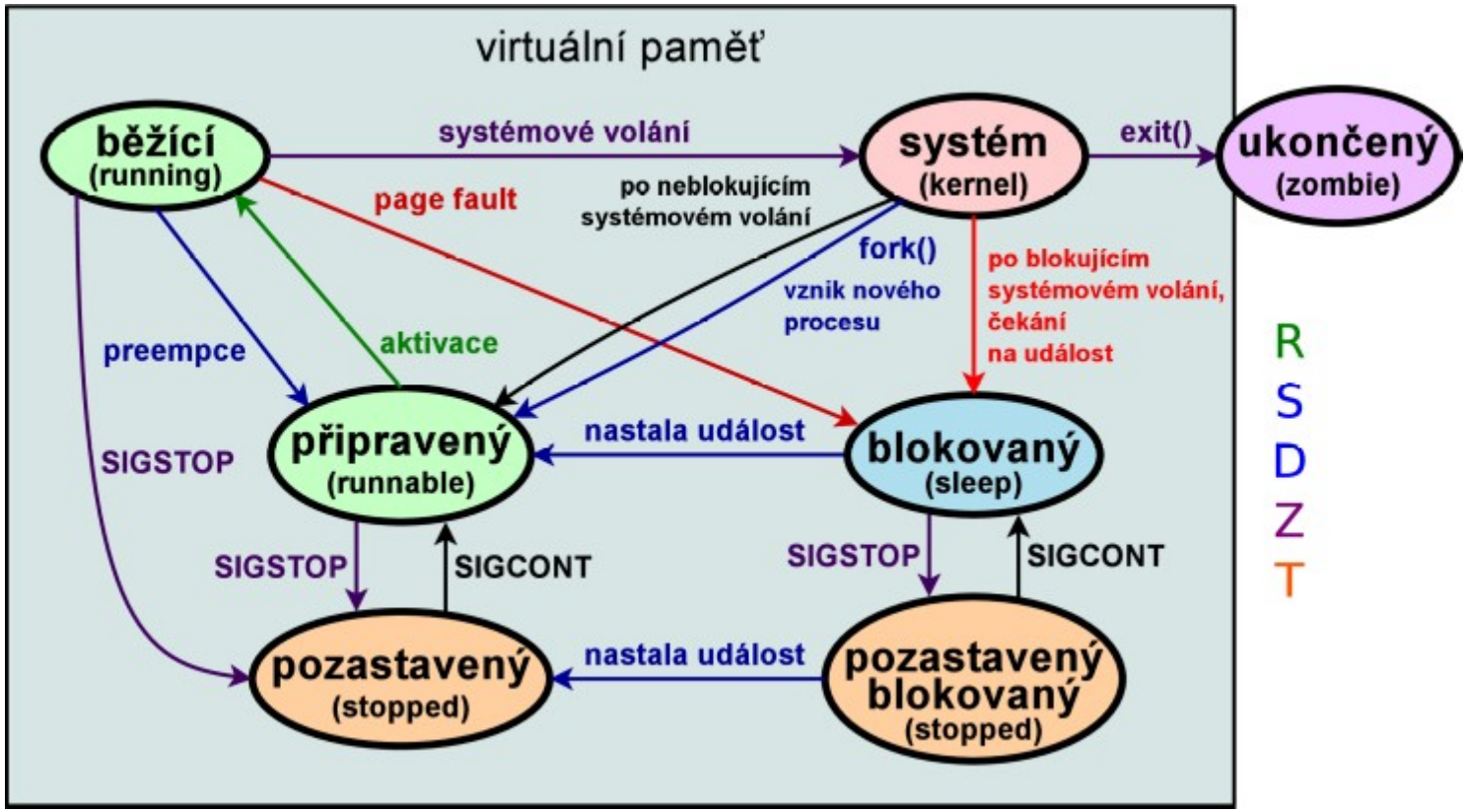
Atributy procesů podle ps (UNIX)

označení	id	popis
PID	pid	identifikace procesu
%CPU	%cpu	využití CPU
%MEM	%mem	využití paměti
CMD	args	příkaz s argumenty
START	bsdstart	čas vzniku procesu
TIME	bsdtime	celkový čas využití CPU – MMM:SS
TIME	cputime	celkový čas využití CPU – [DD-]HH:MM:SS
ELAPSED	etime	celkový čas běhu procesu – [DD-]HH:MM:SS
USER	user	efektivní uživatel (UID), aliasy euser, uname
GROUP	group	efektivní skupina (GID), alias egroup
RUSER	ruser	reálný uživatel (UID)
RGROUP	rgroup	reálná skupina (GID)
NI	nice	hodnota priority nice
STAT	stat	stavy: D, R, S, T, Z, X; atributy: <, N, L, s, l, +
SZ	sz	velikost ve fyzických stránkách (text, data, stack)
VSZ	vsize	velikost virtuální paměti v KiB

Stavy procesů (Linux)

- stavy procesů – jak je vypisuje ps(1):
  - R – připravený nebo běžící (runnable / running)
  - S – blokovaný (sleep)
  - Z – ukončený (defunct, zombie)
  - T – pozastavený nebo krokovaný (stopped, traced)
    - „preempce uživatelem“
  - D – nepřerušitelný spánek (uninterruptible sleep)
    - proces zavolal systémové volání (blokující, proto sleep), které nelze přerušit (např. signálem)

Stavy procesů (Linux) (obrázek)



stavový model procesu v Linuxu (a UNIXu)

Procesy – systémová volání

- fork(2) – vytvoření procesu
- exec(3), execve(2) – nahrazení kódu procesu
- exit(3), _exit(2) – ukončení procesu
- wait(2), waitpid(2) – čekání na změnu potomka
  - na ukončení, případně pozastavení, obnovení běhu
- getpid(2), getppid(2) – zjištění (P)PID
- kill(2), raise(3) – zaslání signálu / zabití procesu
- signal(2), sigaction(2) – obsluha signálu

Procesy – Win32 API

- CreateProcess() – vytvoření procesu
- ExitProcess() – ukončení procesu
- WaitForSingleObject(), WaitForMultipleObjects(), GetExitCodeProcess() – čekání na událost (ukončení procesu), zjištění návratového kódu
- GetCurrentProcessID() – zjištění PID
- TerminateProcess() – zabití procesu

Procesy – vytvoření, nahrazení

- vytvoření procesu – fork(2)

```
#include <unistd.h>
pid_t fork(void);
```

  - vrací: 0 = potomek, 0 < PID = rodič, −1 = chyba

- nahrazení procesu – execve(2)

```
#include <unistd.h>
int execve(const char *filename,
  char *const argv[], char *const envp[]);
```

  - vrací: úspěch = bez návratu, −1 = chyba

- viz též knihovní funkce system(3), exec(3)

Procesy – ukončení

- ukončení – _exit(2)

```
#include <unistd.h>

void _exit(int status);
```

  - ukončí proces okamžitě, raději tedy: exit(3)

```
#include <stdlib.h>

void exit(int status);
```

  - návratovou hodnotu může rodič získat voláním wait(2) nebo waitpid(2)

Procesy – čekání na ukončení

- čekání na změnu stavu – wait(2)
  - změny: ukončení, signál pozastavení, signál pokračování

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

    - ekvivalentní: `waitpid(-1, &status, 0);`

```
pid_t waitpid(pid_t pid, int *status,
  int options);
```

  - Příklad – kompletní je např. v manuálové stránce wait(2):

```
w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
if (w == -1) { perror("waitpid"); exit(EXIT_FAILURE); }
if (WIFEXITED(status))
  printf("exited, status=%d\n", WEXITSTATUS(status));
```

Procesy – spuštění příkazu, získání PID a PPID

- spuštění příkazu v shellu – system(3)

```
#include <stdlib.h>
int system(const char *command);
```

  - spustí příkaz v shellu, čeká na dokončení
  - vrací: 127 = příkaz nenalezen, jinak EC příkazu

- zjištění PID, PPID (parent PID) – getpid(2)

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Signály

- jednoduché zprávy – signal(7)
  - SW obdoba HW přerušení
- pouze posixové systémy
- implicitně je posílá OS při určitých událostech
- explicitně se posílají příkazem kill(1)
- zpracování:
  - ukončení (s eventuálním coredump), pozastavení, pokračování procesu, ignorování

Seznam (některých) signálů

č.	signál	akce	popis
1	SIGHUP	exit	zavěšení, ukončení session
2	SIGINT	exit	ukončení CTRL+C
3	SIGQUIT	core	ukončení CTRL+\
6	SIGABRT	core	ukončení
15	SIGTERM	exit	ukončení
9	SIGKILL	exit	ukončení, nelze zachytit ani ignorovat
	SIGTSTP	stop	pozastavení CTRL+Z
	SIGSTOP	stop	pozastavení, nelze zachytit ani ignorovat
	SIGCONT	resume	pokračování pozastaveného procesu
11	SIGSEGV	core	porušení ochrany paměti
	SIGCHLD	ignore	potomek skončil
14	SIGALRM	exit	alarm
4	SIGILL	core	neplatná instrukce
5	SIGTRAP	core	krokování, ladění
8	SIGFPE	core	výjimka plovoucí řádové čárky

Signály – poslání signálu, obsluha

- poslání signálu procesu – kill(2)

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

  - vrací: 0 = úspěch, −1 = chyba

- nastavení obsluhy signálu (zastaralé, zavrženo)

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum,
  sighandler_t handler);
```

Signály – nedostatky nastavení obsluhy voláním signal(2)

- signal(2) je zastaralé nastavení obslužné rutiny pro zpracování signálu a trpí nedostatky
  - nastavení obsluhy je obvykle pouze jednorázové
    - nelze 100% zaručit opětovné zavolání obslužné rutiny při přijetí dvou signálů rychle po sobě
  - neimplementuje automatické maskování signálů
  - nedostatečně standardizované
    - nedostatečně použitelné chování ve vláknech
    - existují rozdílné implementace

Signály – nastavení obsluhy

- změna obsluhy signálu – sigaction(2)

```
#include <signal.h>

int sigaction(int signum, const struct
  sigaction *act, struct sigaction *oldact);
```

  - vrací: 0 = úspěch, −1 = chyba

```
struct sigaction {
  void     (*sa_handler)(int);    // obsluha, SIG_DFL, SIG_IGN
  void     (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;    // maska blokovaných signálů při obsluze
  int      sa_flags;   // nastavení chování
  void     (*sa_restorer)(void); // zastaralé, nepoužívat
};
```

Signály – maskování, množiny

- blokování signálů, nastavení množin signálů

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set,
  sigset_t *oldset);
```

  - **how**: **SIG_BLOCK**, **SIG_UNBLOCK**, **SIG_SETMASK**

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

  - vrací: 0 = úspěch, −1 = chyba

```
int sigismember(const sigset_t *set, int signum);
```

  - vrací: 1 nebo 0 = úspěch, −1 = chyba

Signály – nahrazení zastaralého volání signal(2) voláním sigaction(2)

obsluha signálu pomocí zastaralého volání signal(2)

```
void handler(int signo) {              // obslužná rutina
   sigset_t mask;                      // maska pro signály
   signal(signo, handler);             // znovunastavení obslužné rutiny
   sigfillset(&mask);                  // naplnění množiny signálů
   sigprocmask(SIG_SETMASK, &mask, NULL);  // nastavení masky
   // obsluha signálu
}
signal(SIGINT, handler);               // (jednorázové) nastavení obslužné rutiny
```

obsluha signálu pomocí sigaction(2)

```
void handler(int signo) {              // obslužná rutina
   // obsluha signálu
}
struct sigaction sa, old_sa;           // struktura pro obsluhu signálu:
sa.sa_handler = handler;               //    název obslužné rutiny
sa.sa_flags = SA_RESTART;              //    příznaky: restart přerušeného syst. volání
sigfillset(&sa.sa_mask);               //    maska
sigaction(SIGINT, &sa, &old_sa);       // (trvalé) nastavení obslužné rutiny
```

Signály – alarm, čekání na signál

- nastavení poslání upozornění – alarm(2)

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

  - vrací: počet sekund zbývajících do alarmu, který byl nastaven předchozím voláním (0 = nebyl)

- čekání na konkrétní signály – sigsuspend(2)

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

  - vrací: vždy −1 (s chybou **EINTR**)

Signály – nastavení časovače

- nastavení upozornění po čase – setitimer(2)

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *val);
int setitimer(int which, const struct itimerval
  *value, struct itimerval *ovalue);
```

  - tři časovače (hodnota which):
    - ITIMER_REAL – SIGALRM, měří reálný čas
    - ITIMER_VIRTUAL – SIGVTALRM, měří čas CPU procesu
    - ITIMER_PROF – SIGPROF, čas CPU procesu + jádra
  - vrací: 0 = úspěch, −1 = chyba

Vlákna

- proces – související prostředky jako celek
  - adresní prostor, environment, pracovní adresář, otevřené soubory, obsluha signálů, nástroje IPC (semafory, sockety), účtování (accounting), …
- vlákno – „odlehčený proces“ – samostatně pouze:
  - stack (lokální data podprogramů) a
  - plánovací položky, tj. stav (připraveno / blokováno / běží), kontext (uložené registry CPU), priorita apod.
  - ostatní je sdíleno s ostatními vlákny procesu

Vlákna – motivace

- kvaziparalelismus – stejný motiv jako proces
- jednodušší a rychlejší správa (vznik, …)
  - pthread_create(3) až 50× rychlejší než fork(2)
- výkon – záleží na aplikaci (CPU × V/V)
- určitě užitečné na SMP
- příklady využití:
  - textový procesor (vstup, V/V, formátování)
  - web-server (síťové spojení, předání stránky)

Vlákna × procesy

- blokující systémová volání (SV) → snadnější programování
- paralelismus, nižší režie → zvýšení výkonu
- příklad web-serveru

	typ SV	snadné programování	paralelismus	režie
jeden proces	blokující	ano	**ne**	nízká
jeden proces	neblokující	spíše ne	ano, 1 CPU	nízká
skupina procesů	blokující	ano	ano	vysoká
vlákna	blokující	ano	ano	nízká

Implementace vláken

- implementace vláken bez podpory OS
  - pomocí knihovních funkcí – problémy:
    - blokující volání převést na neblokující
    - page-fault – stránka není v operační paměti
    - je třeba plánovač vláken – obvykle pracné
- implementace v jádře OS
  - vzdálená volání – více režie
  - není potřeba neblokujících volání
- hybridní implementace (např. Solaris)

Výhody a nevýhody implementací

- implementace bez podpory jádra OS
  - + lepší režie – rychlejší vznik, přepnutí kontextu
  - + nevyžaduje se přechod do režimu jádra
  - + strategie plánovače se dá přizpůsobit aplikaci
  - − složitá implementace (neblokující volání, plánovač)
  - − page-fault zastaví všechna vlákna

Výhody a nevýhody implementací

- implementace v jádře OS
  - + lze provádět i vlákno procesu, jehož jiné vlákno způsobilo page-fault
  - + není třeba neblokujících volání
  - − horší režie – přechod do režimu jádra
  - − pevná strategie plánovače vláken

Problémy při používání vláken

- globální proměnné
  - nutné samostatné alokacé pro každé vlákno (**`errno`**)
  - přístup ke sdíleným proměnným (kritické sekce)
- nereentrantní volání některých knih. funkcí
  - např. alokace paměti (**`malloc`**)
- znalost implementace signálů a jejich obsluhy
  - které vlákno má dostat signál, které se má přerušit
- stack (automatické zvětšení při přetečení?)

Posixová vlákna

- posixová knihovna pthread.h(7)

```
#include <pthread.h>
```

  - POSIX.1c – redefinice globální proměnné **errno**

```
#include <errno.h>
```

    - **špatně:** ~~`extern int errno;`~~
  - kompilace se symbolem **_REENTRANT**

```
gcc -D_REENTRANT prog.c -lpthread -o prog
```

    - zajistí reentrantnost funkcí
    - zajistí přenositelnost

Vlákna POSIX – vytvoření

- vytvoření vlákna – pthread_create(3)

```
int pthread_create(pthread_t *restrict thread,
  const pthread_attr_t *restrict attr,
  void *(*start_routine)(void*),
  void *restrict arg);
```

  - spustí nové vlákno – funkci **start_routine(arg)**
  - **thread** – identifikace vlákna
  - **attr** – nastavení atributů, může být **NULL**
  - vrací: 0 = úspěch, jinak číslo chyby

Vlákna POSIX – ukončení

- ukončení vlákna – pthread_exit(3)

```
void pthread_exit(void *value_ptr);
```

  - volá se implicitně při ukončení funkce (`return`)

- čekání na ukončení vlákna – pthread_join(3)

```
void pthread_join(pthread_t thread,
  void **value_ptr);
```

  - `value_ptr` – návratová hodnota z pthread_exit
    - může být `NULL`, pokud ji nepotřebujeme

Vlákna POSIX – odpojení

- odpojení vlákna – pthread_detach(3)

```
int pthread_detach(pthread_t thread);
```

  - nastaví automatické uvolnění zdrojů vlákna
  - na vlákno pak nelze čekat pomocí **pthread_join**
    - nelze tedy získat návratovou hodnotu
    - informace lze ale předat pomocí globální proměnné
  - lze též nastavit atributem
  - vrací: 0 = úspěch, jinak číslo chyby

Vlákna POSIX – zrušení

- zrušení vlákna – pthread_cancel(3)

```
int pthread_cancel(pthread_t thread);
```

  - ukončí vlákno **thread**
  - vlákno může registrovat ukončovací funkce
    - pthread_cleanup_push(3)
    - zavolají se před ukončením vlákna
  - typ ukončení: asynchronní (ihned) nebo odložené
    - pthread_setcanceltype(3)
  - vrací: 0 = úspěch, jinak číslo chyby

Vlákna POSIX – id, atributy

- získání identifikace vlákna – pthread_self(3)

```
pthread_t pthread_self(void);
```

- porovnání vláken – pthread_equal(3)

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

  - vrací: 0 = t1 a t2 jsou různá vlákna, 0 ≠ stejná

- manipulace s atributy vláken

  pthread_attr_destroy(3), pthread_attr_getdetachstate(3), pthread_attr_getstackaddr(3), pthread_attr_getstack(3), pthread_attr_getstacksize(3), pthread_attr_getschedpolicy(3), pthread_attr_getschedparam(3), pthread_attr_getscope(3), pthread_attr_getinheritsched(3)

Vlákna – Win32 API

- CreateThread() – vytvoření vlákna
- ThreadExit() – ukončení vlákna
- WaitForSingleObject(), WaitForMultipleObjects() – čekání na událost (ukončení vlákna)
- SetPriorityClass(), SetThreadPriority() – manipulace s plánovacími atributy