3. Procesy a vlákna

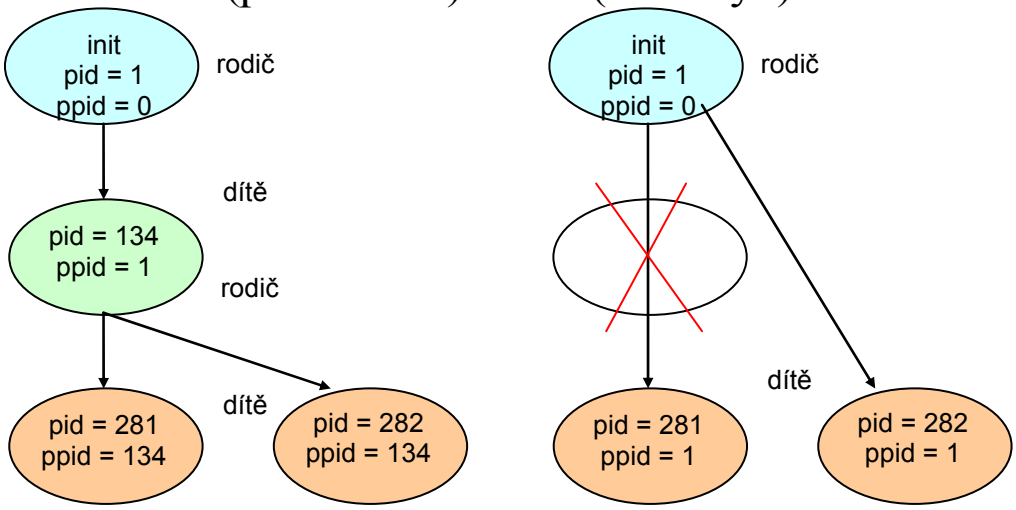
Proces = samostatně prováděný program ve vlastním adresovém prostoru.
Program - statický kód, počáteční data (program *vi*)
Proces - dynamický, běžící program (několik spuštěných *vi*), rozlišení instancí = *identifikace procesu*
Stav procesu – registry procesoru, data, zásobník, systémové prostředky.

Systémy POSIX:
Spuštění procesu: *fork()*
Ukončení procesu: *exit()*, návrat z *main()*, chyba, signál
Čekání na ukončení spuštěného procesu: *waitpid(), wait()*
Identifikace procesu: celé číslo v rozsahu 1..*max*, typ `pid_t`
Získání identifikace procesu: *getpid()*
Atributy procesu: informace o procesu uložené v jádře, lze je získat, některé i nastavit:

Atribut	**typ**	**získání**	**nastavení**
číslo procesu	pid_t	*getpid()*	
číslo procesu otce	pid_t	*getppid()*	
číslo skupiny procesů	pid_t	*getpgrp()*	*setpgid()*
číslo skutečného majitele	uid_t	*getuid()*	*setuid()*
číslo efektivního majitele	uid_t	*geteuid()*	*setuid()*
číslo skutečné skupiny	gid_t	*getgid()*	*setgid()*
číslo efektivní skupiny	gid_t	*getegid()*	*setgid()*
čísla doplňkových skupin	gid_t[]	*getgroups()*	*setgroups()*
pracovní adresář	char *	*getcwd()*	*chdir()*
maska vytváření souborů	mode_t	*umask()*	*umask()*
maska signálů	sigset_t	*sigprocmask()*	*sigprocmask()*
množina čekajících signálů	sigset_t	*sigpending()*	
časy procesu	struct tms	*times()*	
časovač	int	*alarm()*	*alarm()*
sezení			*setsid()*

Hierarchie procesů:

vztah rodič (parent/otec) – dítě (child/syn)



- Proces může čekat pouze na své dětské procesy – *waitpid()*.
- Pokud rodič skončí dříve než dítě, stane se dítě sirotkem => jeho rodičem se stane proces s pid=1 (proces *init*).
- Pokud dětský proces skončí a rodič **čeká** na jeho ukončení (*wait*, *waitpid*), zanikne dětský proces okamžitě.
- Pokud dětský proces skončí a rodič **nečeká** na dokončení (nezajímá se o jeho stav), stane se z dětského procesu **zombie** – mrtvý proces obsahující pouze informaci o stavu a způsobu ukončení. Rodič musí přebírat stav ukončených dětských procesů – pokud tak nečiní, brzy narazí na limit počtu procesů. Pokud rodič skončí bez přebrání stavu, stane se rodičem pid 1 a ten si stav okamžitě převezme (dělá trvale *wait()*), čímž zombie zanikne.

stav procesů – viz příkaz ps(1), top(1)

Příklad:

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

```c
#include <stdio.h>

int main(void)
{
   int status;
   pid_t id = fork();
   if (id == 0) { /* child */
      sleep(10);
      return 0;
   } else
   if (id != -1) {   /* parent */
      signal(SIGINT, SIG_IGN);
      waitpid(id, &status, 0);
      if (WIFEXITED(status)) {
         printf("child %d exit %d\n",
               (int)id, WEXITSTATUS(status));
      } else
      if (WIFSIGNALED(status)) {
         printf("child %d signal %d\n",
               (int)id, WTERMSIG(status));
      }
      return 0;
   } else {
      perror("fork");
      return 1;
   }
}
```

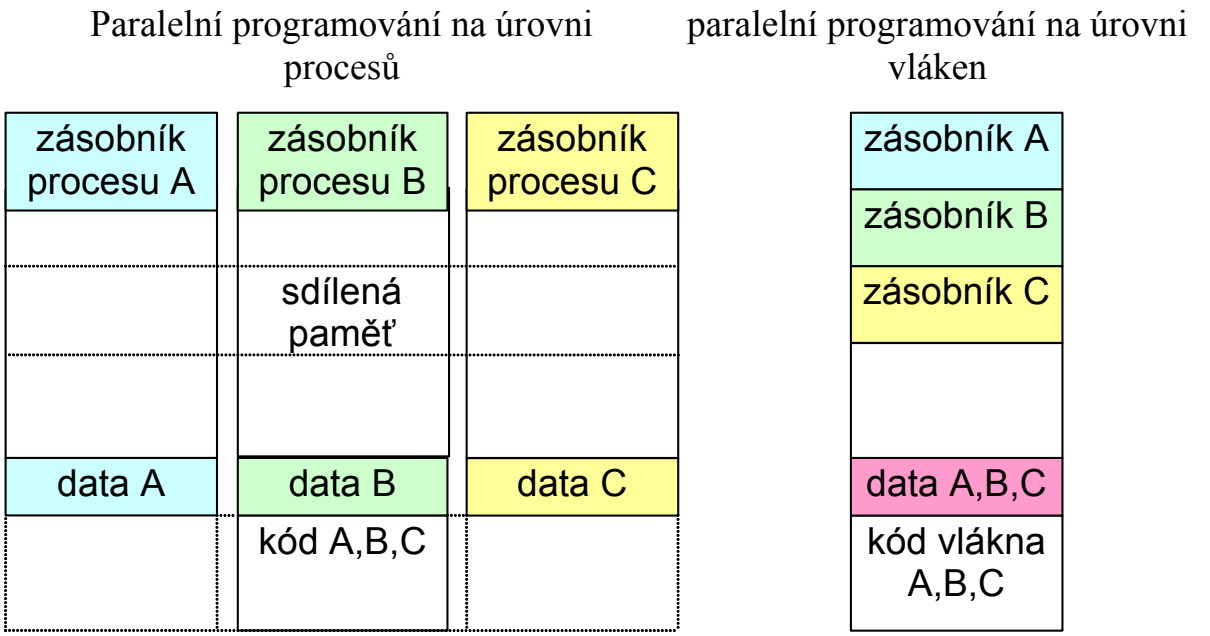
Použití - spustit a počkat, dětský proces doběhne s exit 0, spustit a Ctrl-C, dětský proces bude ukončen signálem.

Vlákna

Problém: máme víceprocesorový (vícejádrový) systém - jak v jednom programu využít více (jader) procesorů?
Lze spustit více procesů, ale ty nemají nic společného (jedině explicitně zřízenou sdílenou paměť).
Krize Moorova zákona – další nárust výkonu lze získat pouze větší paralelizací, růstem počtu jader procesorů, dnes běžně 2-8.
Doporučený studijní text: *Herb Sutter: The Free Lunch Is Over, A Fundamental Turn Toward Concurrency in Software* (viz odkazy u slajdů).

Vlákno = samostatně prováděná část programu v rámci jednoho „procesu“ (proces přestává být jednotkou provádění)
„Proces“ v systémech s vlákny:

- sada souběžně prováděných vláken v jednom adresovém prostoru,
- přestává být jednotkou přidělování procesoru,
- zůstává obálkou vláken pro přidělování systémových prostředků a správu adresového prostoru.



Důležitý princip: Přepínání kontextu mezi vlákny, spouštění a synchronizace vláken by měly mít menší režii než u procesů (jinak by neměla vlákna smysl). Spuštění vláken je typicky o 1 až 2 řády rychlejší než u procesů (nemusí se vytvářet a kopírovat adresový prostor), stejně tak synchronizace.

Typické použití vláken:
1. Urychlení běhu - paralelní programování (multiprocessing)
2. Proložení V/V a běhu - zálohování, vypalování CD, multimédia
3. Síťové servery (Web, FTP)
4. Zpřístupnění sdílených dat více klientům (DB, IRC, MUD)
5. Grafické uživatelské rozhraní
6. Systémy reálného času

„Proces" a vlákna:
- Všechna vlákna v rámci jednoho „procesu" sdílí společný adresový prostor a systémové prostředky
- všechna vlákna sdílí stejný kód a data (není zde ochrana!),
- každé vlákno má své registry, zásobník, stav provádění.

Termín *proces* v praxi (POSIX) reprezentuje vlákna běžící v jednom adresovém prostoru, která sdílí:
- identifikaci procesu (*pid = getpid()*)
- majitelství (uživatel - *uid = getuid()*, skupina - *gid = getgid()*)
- nastavení zpracování signálů (*sigaction()*)
- deskriptory souborů (*fd = open(), pipe(), socket()*)

Proces - jednotka pro přidělování systémových prostředků
Vlákno - jednotka pro přidělování procesoru (čili proces v terminologii teorie OS)

Pozor na marketing:
Intel HyperThreading (P4, Core i5/7, Atom) - více kontextů provádění na jednom fyzickém procesoru/jádře (více sad registrů, PC, SP, atd.), z hlediska uživatelského nerozpoznatelné od systému s odpovídajícím násobným počtem jader/procesorů, nemá spojitost s vlákny (v jádře systému se rozlišují fyzické procesory/jádra a logické, jádro musí znát mapování logických procesorů na fyzické pro efektivní využití fyzických procesorů). Poměrně jednoduchým doplněním HW lze získat výkon větší až o 50% (za cenu zvětšení počtu prvků o cca. 5%). Původní idea *Simultaneous Multithreading Project* (SMT), viz odkaz u slajdů.

Přidělování procesoru vláknům:
- **globální** - pro každé vlákno v systému nezávisle na procesech
- **lokální** - na úrovni procesů, čas procesoru dostává proces

Podpora vláken:
- Open Software Foundation Distributed Computing Environment Threads (OSF DCE Threads) – historicky nejstarší (1990), draft POSIX 1003.4 (prehistorický, chyby)
- IEEE POSIX 1003.1c - pthreads (POSIX threads)
- OS/2
- Win32/NT
- Java, Perl, Python, C11, C++11, ...

IEEE/ISO POSIX 1003.1c-1995 (1003.1-1996 a vyšší)

Vlákno = sekvenčně prováděná funkce jazyka C:

```
int global;              /* sdílená proměnná */

void *vlakno(void *arg)
{
    int local;           /* privátní prom.vlákna */
    static global;       /* sdílená proměnná  */
    ...                  /* kód vlákna */
    return stat;         /* ukončení vlákna */
/*  pthread_exit(void *stat); /* ekvivalentní*/
}
```

- Lokální (privátní pro vlákno) jsou hodnotou předávané parametry funkcí a auto proměnné – každé vlákno má svůj zásobník a tudíž i instance těchto proměnných.
- Globální (tedy implicitně sdílené) jsou všechny ostatní proměnné (extern, static, heap=malloc).

Identifikace vlákna = typ **pthread_t**
Konkrétní typ záleží na implementaci (většinou int), je to černá skříňka, lze pouze předávat jako parametr nebo porovnávat dvě hodnoty pomocí *pthread_equal(pth1, pth2)*.

Vytvoření a spuštění vlákna – *pthread_create()*:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
     const pthread_attr_t *attr, /* atributy */
     void *(*func)(void *),      /* vlákno */
     void *arg);                 /* parametr */
```

Spustí nový kontext provádění, který začne paralelně provádět nové vlákno řízené funkcí *func*. Na adresu zadanou prvním parametrem je uložena identifikace spuštěného vlákna. Atributy vlákna mohou být NULL (implicitní). Parametr *arg* je předán spuštěné funkci *func*, lze použít pro rozlišení vícenásobného

spuštění vlákna řízeného stejnou funkcí. **Pozor na životnost** předávané hodnoty, pokud jedno vklákno spouští jiné, je chybou předávat ukazatel na lokální proměnnou prvního vlákna (může zaniknout dříve než se spuštěné rozběhne). Korektní je pouze alokovat hodnotu pomocí *malloc()*, předat ukazatel a spuštěné vlákno po zpracování uvolnit hodnotu pomocí *free()*.

Příklad:

```
pthread_t pt;       /* identifikace vlákna */
...
while (pthread_create(&pt, NULL, vlakno, NULL)) {
/* chyba, pokud EAGAIN, lze zkusit znovu */
    printf("pthread_create() error %d\n", errno);
    exit(1);
}
/* vlákno vlakno() nyní běží paralelně s následujícím  kódem */
```

Získání identifikace běžícího vlákna:

```
pthread_t pthread_self();
```

Čekání na ukončení vlákna a převzetí stavu ukončení:

```
int pthread_join(pthread_t thread, void **st);
```

Čekat lze pouze, pokud není vlákno osamostatněno (*detached*). Čekat může kterékoli vlákno, není zde omezení jako u procesů podle vztahu rodič/dítě. Pokud je vlákno ukončeno, ale není převzat stav, počítá se do limitu počtu spuštěných vláken. Pro návratovou hodnotu platí stejná omezení jako pro argument vlákna.

Atributy vlákna

Atributy vytvářeného vlákna jsou zadávány typem *pthread_attr_t*
Proměnná obsahující atributy musí být inicializována funkcí:

```
int pthread_attr_init(pthread_attr_t *attr);
```

Po inicializaci jsou atributy nastaveny na implicitní hodnotu ekvivalentní NULL. Pro nastavení (získání) jednotlivých atributů jsou definovány funkce:

*int pthread_attr_setdetachstate(pthread_attr_t *attr, int state);*

- samostatné vlákno - PTHREAD_CREATE_DETACHED, lze na něj čekat - PTHREAD_CREATE_JOINABLE

*int pthread_attr_setscope(pthread_attr_t *attr, int scope);*

PTHREAD_SCOPE_PROCESS - plánování lokální
PTHREAD_SCOPE_SYSTEM - plánování globální

*int pthread_attr_setstacksize(pthread_attr_t *attr, size_t size);*

- velikost zásobníku pro vlákno – je třeba počítat s tím, že zásobník obsahuje nejen aktivace funkcí (parametry, návratovou adresu), ale také lokální proměnné funkcí, min. velikost je PTHREAD_STACK_MIN

*int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);*

- plánovací algoritmus *(*SCHED_FIFO, SCHED_RR, SCHED_OTHER)

*int pthread_attr_setchedparam(pthread_attr_t *attr,*
*struct sched_param *sp);*

- parametry plánovacího algoritmu

Po použití atributů v *pthread_create()* nejsou dále k ničemu a je třeba je uvolnit:

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Příklad:

```
int main(int argc, char *argv[])
{
pthread_attr_t attr;
pthread_t pt;
void *result;
int res;
/* vytvoření implicitních atributů */
if ((res = pthread_attr_init(&attr)) != 0) {
    printf("pthread_attr_init() err %d\n", res);
    return 1;
}
/* nastavení typu vlákna v atributech */
if ((res = pthread_attr_setdetachstate(&attr,
     PTHREAD_CREATE_JOINABLE)) != 0) {
    printf("pthread_attr_setdetachstate() err
%d\n", res);
    return 1;
}
/* vytvoření a spuštění vlákna */
res = pthread_create(&pt, &attr, vlakno, NULL);
if (res) {
    printf("pthread_create() error %d\n", res);
    return 1;
}
/* čekání na dokončení a převzetí stavu */
if ((res = pthread_join(pt, &result)) != 0) {
    printf("pthread_attr_init() err %d\n",res);
    return 1;
}
printf("vlákno skončilo se stavem %d\n",
(int)result);
return 0;
}
```

Stav vlákna je typu *void* *, pokud se vrací celé číslo, je třeba přetypovat (pthread_exit((void *)(1));

Problémy implementace vláken - statické proměnné standardních funkcí:

- **errno** musí obsahovat chybový kód specifický pro vlákno, které volalo funkci standardní knihovny C/POSIX.
- Mnohé funkce standardní knihovny C si ukládají něco do statických proměnných (*strtok*, *asctime*, *ctime*, *gmtime*, *localtime*, *tmpnam*, *rand*) – nejsou pak reentrantní.
- Stejně tak funkce klasického rozhraní Unix/POSIX: *getlogin*, *ttyname*, *readdir*, *getgrgid*, *getgrnam*, *getpwuid*, *getpwnam*, *ctermid*.

Řešení:

- Zdrojový soubor – přidat před prvním `#include`:

```
#define _POSIX_C_SOURCE 199506L       a
#define _REENTRANT                    (starší)
```

- upravené verze hlavičkových souborů

```
/* hlavičkový soubor stdio.h */
...
#if __POSIX_VISIBLE >= 199506
#define getc_unlocked(fp)        __sgetc(fp)
#define putc_unlocked(x, fp)     __sputc(x, fp)
...
#endif
```

- per-vlákno verze *errno* (musí se získat vždy pomocí #include <errno.h>, nelze deklarovat *extern int errno!*):

```
/* hlavičkový soubor errno.h */
extern int *__errno_location(void);
#define errno (*__errno_location())
```

- proměnné specifické pro vlákno (univerzální obdoba **errno**)

```
int pthread_key_create(pthread_key_t *,
        void (*) (void *));
int pthread_setspecific(pthread_key_t,
        const void *);
void *pthread_getspecific(pthread_key_t);
```

- Reentrantní verze standardních funkcí: *localtime_r()*, *readdir_r()*, *strtok_r(),* atd. – parametr navíc pro uložení kontextu, stavu.

- Co když se přepne kontext mezi vlákny uprostřed funkce:
  ```
  printf("hodnota A=%d, B=%d", a, b);
  ```
  Byl by proložen výstup z různých vláken!

Standardní V/V je na celou operaci automaticky interně zamčen (platí pro všechny V/V operace *printf*, *fprintf*, *fputs*, *scanf*, atd.). Explicitně lze zamknout daný soubor pro více operací pomocí:

```
void flockfile(FILE *f);
void funlockfile(FILE *f);
int ftrylockfile(FILE *f);
```

V zamykání je skryta režie, proto jsou navíc doplněny operace bez zamykání:

```
int getchar_unlocked(void);
int getc_unlocked(FILE *f);
int putchar_unlocked(int c);
int putc_unlocked(int c, FILE *f);
```

Jak překládat a sestavovat program?

Obecné pravidlo není, vždy je třeba znát konkrétní překladač a sestavovací program:

```
cc -mt -O -o prog prog.c -lpthread    (Solaris)
cc -pthread -O -o prog prog.c      (clang FreeBSD)
icc -pthread -O -o prog prog.c     (Intel C)
gcc -pthread -O -o prog prog.c     (GCC Linux)
```

Připojí často další knihovny (libpthread, viz *ldd*)

Kombinace *fork()* a *pthread_create()*

Pokud systém podporuje vlákna, jak vlastně funguje *fork()*?

- Pokud proces nespustil žádné vlákno, žádná změna. Jeden kontext provádění, další *fork()* kopíruje tento celý proces.
- Pokud proces spustil alespoň jedno vlákno, pak běží více kontextů provádění = vlákna. Kterékoli běžící vlákno může kdykoli zavolat *fork()*:
  1. *fork()* kopíruje celý adresový prostor procesu, včetně všech interních datových struktur vláken, tj. semaforů a dalších synchronizačních prostředků (jsou obvykle v paměti).
  2. v nově spuštěném dětském procesu bude pouze jeden kontext provádění = jedno vlákno, které bude odlišné od všech vláken v rodičovském procesu.

Problém:
Co se zámky na úrovni vláken, které jsou v okamžiku *fork()* zamčené? V nově vzniklém procesu nemají zámky majitele, jsou zamčené, ale nikdo je neodemkne.

Řešení:

```
int pthread_atfork(void (*prepare)(void), void
    (*parent)(void), void (*child)(void));
```

Registrace funkcí, které se volají:

1. Před provedením *fork()* v rodičovském procesu (*prepare*)
2. Před návratem z *fork()* v rodičovském procesu (*parent*)
3. Před návratem z *fork()* v dětském procesu (*child*)

Registraci lze volat opakovaně, nepotřebné mohou být NULL.
Správné řešení spočívá v zamčení všech zámků v kroku *prepare* (tím je nemůže mít zamčené nějaké jiné vlákno) a jejich odemčení v krocích *parent* i *child*.
Problém: co když je těch zámků hodně? Použít *fork()* jen pokud je následován *exec()* a v dětském procesu nic nezamykat!

ISO C11 (ISO/IEC 9899:2011)

Doplněn paměťový model, zarovnání, atomické operace, vlákna a synchronizace. Volitelná implementace, test:

```
#if __STDC_VERSION__ >= 201112L /* C-2011 */
#if __STDC_NO_ATOMICS__
/* není <stdatomic.h> a typ _Atomic */
#endif
#if __STDC_NO_THREADS__
/* není <threads.h>, vlákna, synchronizace */
#endif
#endif
```

Vlákno = sekvenčně prováděná funkce typu **thrd_start_t**:

```
int global;              /* sdílená proměnná */
thread_local var1;       /* lokální prom.vlákna */

int vlakno(void *arg)
{
    int local;           /* privátní prom.vlákna */
    static global;       /* sdílená proměnná  */
    thread_local var2; /* lokální prom.vlákna */
    ...                  /* kód vlákna */
    return res;          /* ukončení vlákna */
/*  thrd_exit(int res); /* ekvivalentní*/
}
```

- Rozdíl mezi *auto* a *thread_local* je v alokaci, *auto* se alokuje při každém volání funkce (rekurzivním), *thread_local* pouze při spuštění vlákna.

Identifikace vlákna = typ **thrd_t**
Porovnání identifikací pomocí *thrd_equal(thrd_t th1, thrd_t th2).*

Vytvoření a spuštění vlákna:

```
#include <threads.h>
int thrd_create(thrd_t *thr,
     thrd_start_t func,          /* vlákno */
     void *arg);                 /* parametr */
```

Vytvoří a spustí nové vlákno řízené funkcí *func*. Výsledkem je *thrd_success, thrd_nomem* nebo *thrd_error*.

Získání identifikace běžícího vlákna:

```
thrd_t thrd_current();
```

Ukončení vlákna:

```
void thrd_exit(int res);
```

Pokud skončí poslední vlákno procesu, je proces ukončen stejně jako voláním exit(EXIT_SUCCESS);

Čekání na ukončení vlákna a převzetí stavu ukončení:

```
int thrd_join(thrd_t thr, int *res);
```

Vrací *thrd_success* nebo *thrd_error*.

Osamostatnění vlákna (nelze pak na něj čekat):

```
int thrd_detach(thrd_t thr);
```

Proměnné specifické pro vlákno (thread-specific storage):

```
int tss_create(tss_t *key, tss_dtor_t dtor);
void *tss_get(tss_t key);
int tss_set(tss_t key, void *val);
void tss_delete(tss_t key);
```

Ekvivalentní POSIX.

ISO C++11 (ISO/IEC 14882:2011)

Vlákno zabaleno do standardní třídy **std::thread**:

```
#include <thread>
class thread {
public:
    class id {
        public: id();
        bool operator==(id x, id y);
    };
    template< class Function, class ...Args >
    thread(Function&& f, Args&&... args);
    ~thread(); /* destruktor, ukončí a zruší */
    thread::id get_id() const;
    bool joinable() const;
    static unsigned hardware_concurrency();
    void join();
    void detach();
...
};
```

Vlákno je reprezentováno instancí třídy *thread* s parametrem typu *Callable* (funkce nebo třída implementující *operator()*). Do funkce vlákna lze předat libovolný počet parametrů (implicitně hodnotou, pomocí *std::ref()* i odkazem).

Příklad:

```
#include <thread>
void func(int n) { ... }

int main()
{
    std::thread thr(func, 5);
    thr.join();
}
```

překlad na serveru eva (gcc8, pro c++ stačí -pthread):

```
g++ -pthread -std=c++11 -Wl,-rpath=/usr/local/lib/gcc8 \
        -O2 -o thr thr.cc
```

Vlákna rozhraní WIN32 (Windows NT – Win 10)

Vytvoření vlákna:

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES attr,   atributy přístupu
    DWORD size,                   počáteční velikost zásobníku
    LPTHREAD_START_ROUTINE func,  funkce implem. vlákno
    LPVOID lpParameter,           parametr funkce
    DWORD flags,                  příznaky
    LPDWORD threadid              identifikace vlákna
);
```

výsledkem je deskriptor - lze použít ve všech funkcích WIN32 s parametrem HANDLE, např. *DuplicateHandle*()
atributy přístupu - dědění a práva ve spuštěných procesech
příznaky - CREATE_SUSPENDED vytvoří vlákno bez spuštění

Spuštění vlákna:

```
ResumeThread(hThread)
```

Tělo vlákna:

```
DWORD func(LPVOID arg)
{
    ...
    return value;   /* nebo ExitThread(value) */
}
```

Čekání na ukončení vlákna - čekání na objekt (viz kap. 7.10)
Převzetí stavu:

```
BOOL GetExitCodeThread(HANDLE ht, LPDWORD code);
```

Při použití z jazyka C/C++:

```
_beginthreadNT(),_endthread()
```

lokální proměnná vlákna:

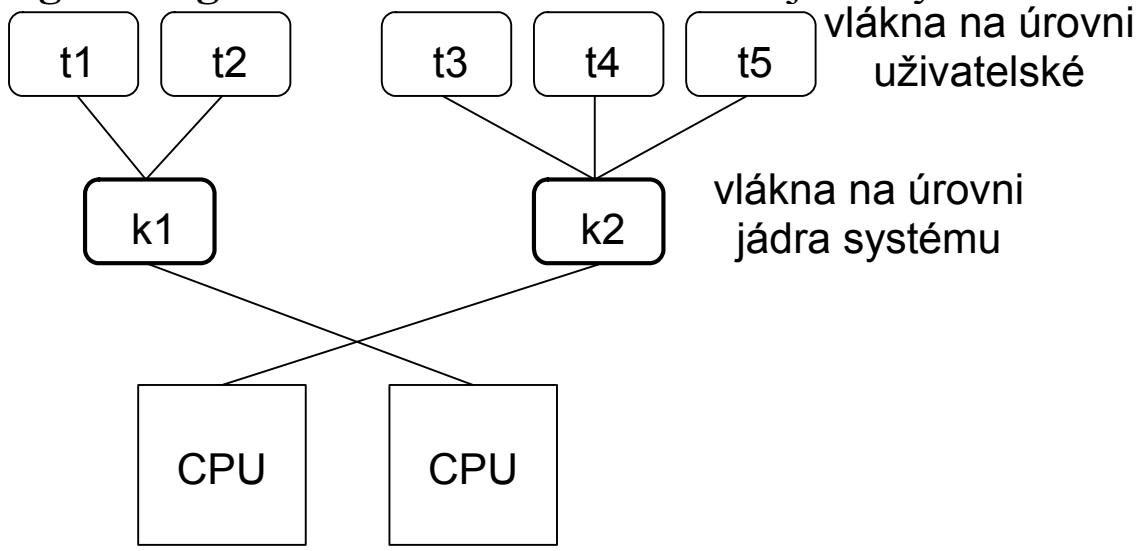
```
__declspec(thread) int local;
```

Implementace vláken

- N:N (1:1) - na úrovni jádra systému, vlákna na úrovni uživatelské jsou reprezentována v jádře (OS/2, AIX < 4.2, Windows/NT, LinuxThreads, NPTL – *clone()*):
  - – režie přepínání kontextu
  - – jádro musí evidovat všechna vlákna (datové struktury v jádře)
  - + plné využití více procesorů v jednom programu
  - + volání jádra přímá (nemusí být zapouzdřena)
- N:1 - na úrovni knihoven, vlákna jsou plně implementována v rámci uživatelského procesu, jádro o nich nic neví (DCE, FreeBSD < 5.x)
  - + nízká režie jádra, plná kontrola nad plánováním
  - – všechna blokující volání jádra musí být zapouzdřena
  - – nelze využít více procesorů v jednom programu, jednotkou přidělování času procesoru je proces
- N:M (N≥M≥1) - kombinovaný přístup, důvody:
  - prováděná vlákna musí být reprezentována v jádře pro správu procesorů, nicméně nemá smysl reprezentovat všechna běžící, stačí tolik, kolik je procesorů,
  - čekající (pozastavená) a připravená vlákna nemusí být reprezentována datovými strukturami v jádře, jádro o nich vůbec nemusí vědět - menší režie
  - přepínání kontextu přes jádro má větší režii než v rámci uživatelského procesu – dokud lze využít přidělený procesor, probíhá běh v režimu sdílení času pro všechna aktivní vlákna v daném procesu.
  - lze simulovat N:1 až N:N nastavením max. počtu vláken na úrovní jádra (*thr_setconcurrency()*)

SOLARIS (LWP), AIX, Irix 6.5, FreeBSD 5.x (KSE) - vlákna na úrovni uživatelské lze vázat dynamicky nebo pevně na vlákna jádra systému

Light Weight Process - vlákno na úrovni jádra systému



Implementace modelu M:N je značně složitá, LWP mají v jádře obdobnou režii jako procesy (bez adresového prostoru). LWP jsou z hlediska jádra jednotkami přidělování procesorů. Klasický proces pak běží jako 1 LWP vlákno.

Problém modelů N:N a N:M – podpora vláken na úrovni uživatelské nemá dostatečné informace o akcích na straně jádra, plánování je problém.

Implementace vláken na uživatelské úrovni (N:1):

- blokující volání jádra musí být nahrazeny neblokujícími:

```
read(fd, buffer, n)→
    fcntl(fd, F_SETFL, O_NONBLOCK);
    if (sys_read(fd, buffer, n) == -1 &&
        errno == EAGAIN) {
        přidej fd do rfd pro select()
        vyjmi current z fronty připravených
        vlož current do fronty čekajících
        plánovač vláken
    }
```

- plánovač (spuštěn explicitně, signálem časovače, aj.)

```
ulož stav vlákna do current (registry, PC, SP)
if (select(nfd, &rfd, &wfd, &efd, &t0) > 0) {
    vyjmi fd z rfd (wfd, efd)
    přesuň odblokované vlákna z fronty čekajících
    do fronty připravených
}    /* signál časovače */
if (current->run_time > time_slice) {
    přesuň current na konec fronty připravených
}
vezmi první vlákno (pt) z fronty připravených
current = pt;
obnov stav vlákna z current
```

Princip je jednoduchý, implementace složitá, je třeba nahradit všechny blokující funkce, nereentrantní a standardní V/V