

Vícevláknové programování

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 11

BAB36PRGA – Programování v C

Přehled témat

- Část 1 – Vícevláknové programování

Úvod

Vícevláknové aplikace a operační systém

Modely vícevláknových aplikací

Mechanismy synchronizace

Vlákna POSIX

C11 Vlákna

Debugging

Část I

Část 1 – Vícevláknové programování

Terminologie – Vlákna

- Vlákno je nezávislé provádění posloupnosti instrukcí.
 - Je to samostatně prováděný **výpočetní tok**.

Typický malý program, který je zaměřen na určitou část.
- Vlákno je spuštěno v rámci procesu.
 - Sdílí stejný paměťový prostor jako proces.
 - Vlákno běží ve stejném paměťovém prostoru procesu.
- Vlákno **runtime environment** – každé vlákno má svůj samostatný prostor proměnných.
 - Identifikátor vlákna a prostor synchronizačních proměnných.
 - Čítač programu (*Program Counter* – PC) nebo ukazatel instrukce (*Instruction Pointer* – IP) – adresa prováděné instrukce.

Udává, kde se vlákno nachází ve své programové sekvenci.
 - Paměťový prostor lokálních proměnných **stack**.

Kde lze použít vlákna?

- Vlákna jsou odlehčené varianty procesů, které sdílejí paměťový prostor.
- Existuje několik případů, kdy je užitečné použít vlákna, nejtypičtější situace jsou následující.
 - **Efektivnější využití dostupných výpočetních zdrojů.**
 - Když proces čeká na zdroje (např. čte z periferie), je zablokovan a řízení je předáno jinému procesu.
 - Vlákno také čeká, ale jiné vlákno v rámci téhož procesu může využít vyhrazený čas pro provádění procesu.
 - Máme-li vícejádrové procesory, můžeme urychlit výpočet využitím více jader současně **paralelními algoritmy**.
 - **Pracování s asynchronními událostmi.**
 - Během blokované i/o operace může být procesor využit pro jiné výpočty.
 - Jedno vlákno může být vyhrazeno pro i/o operace, např. pro komunikační kanál, další vlákno pro výpočty.

Vlákna a procesy

Proces

- Výpočetní tok.
- Má vlastní paměťový prostor.
- Entita (objekt) operačního systému.
- Synchronizace s využitím OS (IPC).
- CPU přidělený plánovačem OS.
- Čas pro vytvoření procesu.

Vlákna procesu

- Výpočetní tok.
- Běží ve stejném paměťovém prostoru procesu.
- Uživatelská entita nebo entita operačního systému.
- Synchronizace prostřednictvím výhradního přístupu k proměnným.
- Procesor přidělený v rámci vyhrazeného času procesu.
- + Vytvoření je rychlejší než vytvoření procesu.

Příklady použití vláken

- **Vstupní/výstupní operace**
 - Vstupní operace mohou zabrat značnou část času běhu, což může být většinou nějaké čekání, např. na vstup uživatele.
 - Během komunikace může být vyhrazený čas procesoru využit pro výpočetně náročné operace.
- **Interakce s grafickým uživatelským rozhraním (GUI)**
 - Grafické rozhraní vyžaduje okamžitou odezvu pro příjemnou interakci uživatele s naší aplikací.
 - Interakce uživatele generuje události, které ovlivňují aplikaci.
 - Výpočetně náročné úlohy by neměly snižovat interaktivitu aplikace.
Zajistit příjemný uživatelský zážitek s naší aplikací.

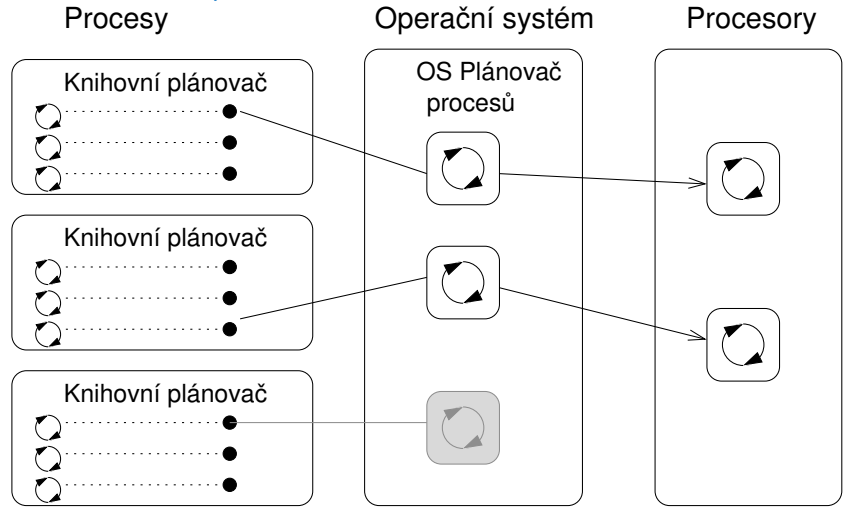
Vícevláknové a víceprocesové aplikace

- **Vícevláknová aplikace.**
 - + Aplikace může využívat vyšší stupeň interaktivity.
 - + Snadnější a rychlejší komunikace mezi vlákny využívajícími stejný paměťový prostor.
 - Nepodporuje přímo škálování paralelního výpočtu do distribuovaného výpočetního prostředí s různými výpočetními systémy (počítači).
- I na jednojádrových jednoprocessorových systémech může vícevláknová aplikace lépe využít procesor.

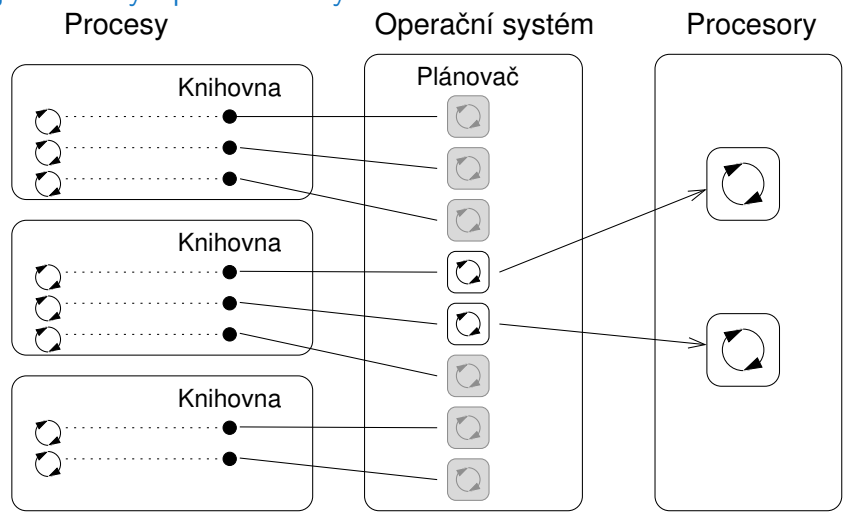
Vlákna v operačním systému

- Vlákna běží v rámci procesu, ale pokud jde o implementaci, vlákna mohou být v uživatelském prostoru nebo jako entity operačního systému.
 - **Uživatelský prostor procesu** – vlákna jsou implementována knihovnou určenou uživatelem.
 - Vlákna nepotřebují zvláštní podporu operačního systému.
 - Vlákna jsou plánována místním plánovačem poskytovaným knihovnou.
 - Vlákna obvykle nemohou využívat více procesorů (více jader).
 - **OS entity**, které jsou plánovány systémovým plánovačem.
 - Může využívat vícejádrové nebo víceprocesorové výpočetní zdroje.

Vlákna v uživatelském prostoru



Vlákna jako entity operačního systému



Vlákna uživatele vs. vlákna operačního systému

Uživatelská vlákna

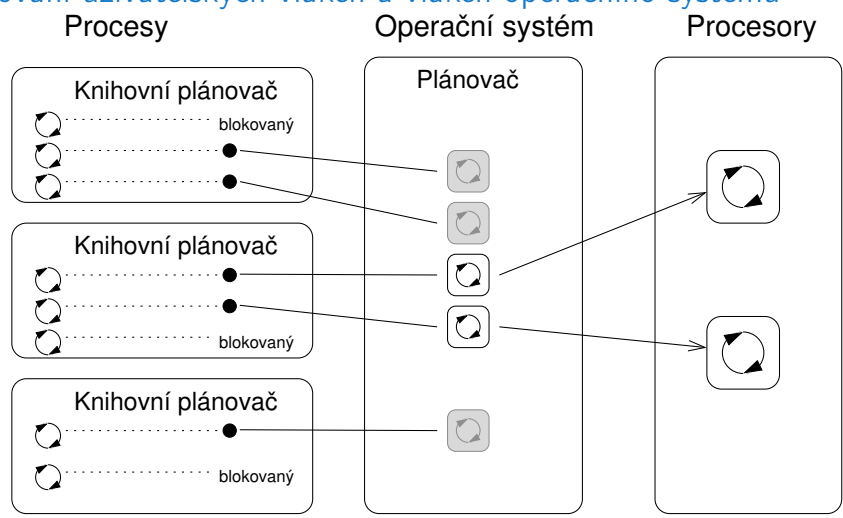
- + Nepotřebují podporu operačního systému.
- + Vytvoření nepotřebuje (drahé) systémové volání.
 - Drahé je relativní k nákladům na vytvoření vlákna, systémového vlákna a procesu.
- Priorita provádění vláken je řízena v rámci přiděleného času procesu.
- Vlákna nemohou běžet současně (pseudoparalelismus).

Vlákna operačního systému

- + Vlákna mohou být naplánována v konkurenci všech vláken v systému.
- + Vlákna mohou běžet současně (na vícejádrovém nebo víceprocesorovém systému – skutečný paralelismus).
- Vytváření vláken je trochu složitější (systémové volání).

Vysoký počet vláken naplánovaných operačním systémem může zvýšit režii. Moderní operační systémy však používají plánovače O(1) - plánování procesu nezávisí na počtu procesů. Plánovací algoritmy jsou založeny na komplexních heuristikách.

Kombinování uživatelských vláken a vláken operačního systému



Typické vícevláknové aplikace

- **Servery** – obsluhují více klientů současně. Může vyžadovat přístup ke sdíleným zdrojům a mnoho i/o operací.
- **Výpočetní aplikace** – s vícejádrovým nebo víceprocesorovým systémem lze zkrátit dobu běhu aplikace současným použitím více procesorů.
- **Aplikace v reálném čase** – pro splnění požadavků na reálný čas můžeme využít specifické plánovače.

Vícevláknová aplikace může být efektivnější než složité asynchronní programování; vlákno čeká na událost oproti explicitnímu přerušení a přepínání kontextu.

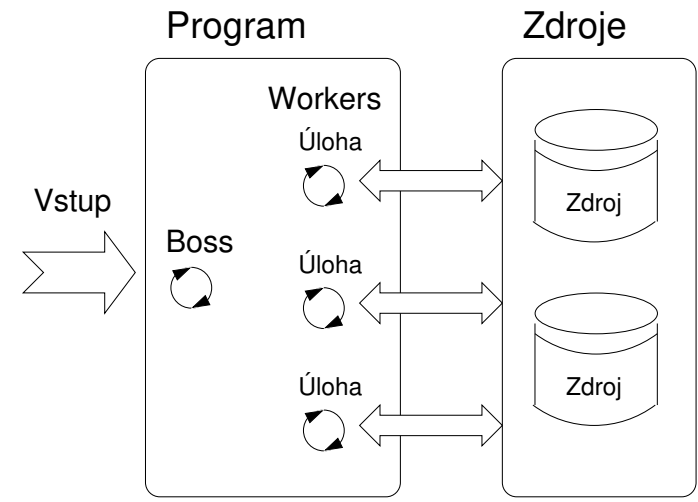
Kdy používat vlákna?

- Vlákna jsou výhodná vždy, když aplikace splňuje některé z následujících kritérií.
- Skládá se z několika nezávislých úloh.
- Může být blokována po určitou dobu.
- Obsahuje výpočetně náročnou část (a zároveň je žádoucí zachovat interaktivitu).
- Musí pohotově reagovat na asynchronní události.
- Obsahuje úlohy s nižší a vyšší prioritou než zbytek aplikace.
- Hlavní výpočetní část lze urychlit paralelním algoritmem s využitím vícejádrových procesorů.

Modely vícevláknových aplikací

- Modely se zabývají vytvářením a rozdělováním práce jednotlivým vláknům.
 - **Boss/Worker** – hlavní vlákno řídí rozdělení práce na ostatní vlákna.
 - **Peer** – vlákna běží paralelně bez určeného správce (šéfa).
 - **Pipeline** – zpracování dat v posloupnosti operací.
Předpokládá dlouhý tok vstupních dat a jednotlivá vlákna pracují paralelně na různých částech toku

Model Boss/Worker



Příklad – Boss/Worker

```

1 // Boss
2 while(1) {
3     switch(getRequest()) {
4         case taskX:
5             create_thread(taskX);
6             break;
7         case taskY:
8             create_thread(taskY);
9             break;
10    }
11 }

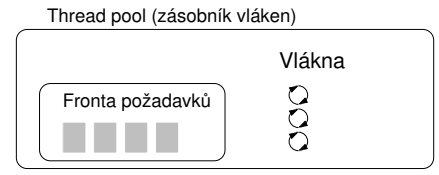
1 // Task solvers
2 taskX()
3 {
4     solve the task // synchronized
5     usage of shared resources
6     done;
7 }
8 taskY()
9 {
10    solve the task // synchronized
11    usage of shared resources
12    done;
13 }
    
```

Model Boss/Worker– role

- Hlavní vlákno je zodpovědná za správu požadavků. Pracuje v cyklu.
 1. Přijme nový požadavek.
 2. Vytvoří vlákno pro obsluhu daného požadavku.
 - Nebo předá požadavek existujícímu vláknku.*
 3. Čekat na nový požadavek.
- Výstup/výsledky přiřazeného požadavku může řídit konkrétní pracovní vlákno nebo hlavní vlákno.
 - Konkrétní vlákno (worker) řešící požadavek.
 - Hlavní vlákno používající synchronizační mechanismy (např. frontu událostí).

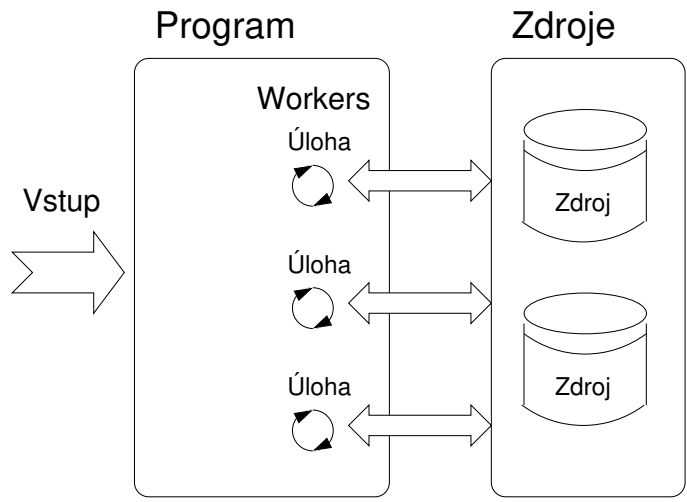
Rezervoár/zásobník vláken – Thread Pool

- Hlavní vlákno vytváří vlákna po přijetí nového požadavku.
- Režie s vytvářením nových vláken může být snížena využitím zásobníku vláken (**Thread Pool**) s již vytvořenými vlákny.
- Vytvořená vlákna čekají na nové úlohy.



- Vlastnosti fondu vláken je třeba vzít v úvahu.
 - Počet předem vytvořených vláken.
 - Maximální počet požadavků ve frontě požadavků.
 - Definice chování, pokud je fronta plná a žádné z vláken není k dispozici.
 - Např. zablokovat příchozí požadavky.*

Peer Model



Vlastnosti a příklad Peer modelu

- Neobsahuje hlavní vlákno; první vlákno vytvoří všechna ostatní vlákna a pak:
 - Stane se jedním z ostatních vláken (ekvivalentní).
 - Pozastaví své provádění a čeká na ostatní vlákna.
- Každé vlákno je zodpovědné za svůj vstup a výstup.
- Příklad

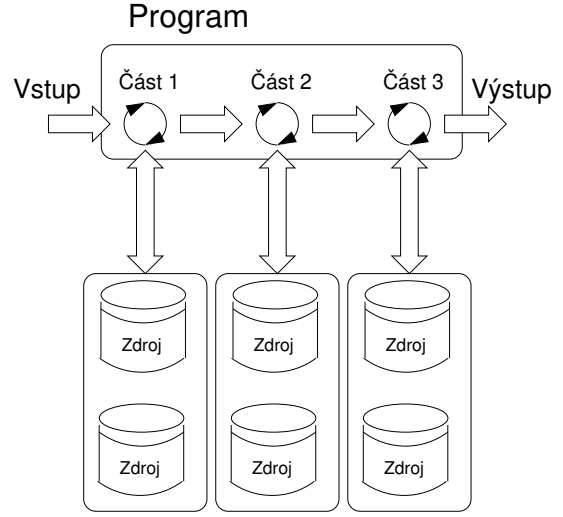
```

1 // Boss
2 {
3   create_thread(task1);
4   create_thread(task2);
5   .
6   .
7   start all threads;
8   wait to all threads;
9 }

1 // Task solvers
2 task1()
3 {
4   wait to be executed
5   solve the task // synchronized usage of shared resources
6   done;
7 }
8 task2()
9 {
10  wait to be executed
11  solve the task // synchronized usage of shared resources
12  done;
13 }
14 }

```

Zpracování datového toku – Pipeline model



Pipeline model – vlastnosti a příklad

- Dlouhý vstupní tok dat s **sekvencí operací** (část zpracování) – každá vstupní datová jednotka musí být zpracována všemi částmi operací zpracování.
- V určitém čase jsou různé vstupní datové jednotky zpracovány jednotlivými částmi zpracování – vstupní jednotky musí být nezávislé.

```

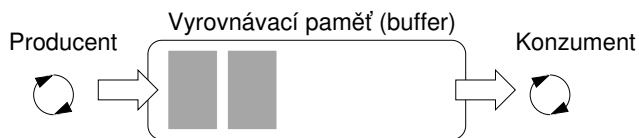
1 main()
2 {
3   create_thread(stage1);
4   create_thread(stage2);
5   ...
6   ...
7   wait // for all pipeline;
8 }
9 stage1()
10 {
11  while(input) {
12    get next program input;
13    process input;
14    pass result to next the stage
15  };
16 }
17 }

1 stage2()
2 {
3   while(input) {
4     get next input from thread;
5     process input;
6     pass result to the next stage;
7   }
8 }
9 ...
10 stageN()
11 {
12  while(input) {
13    get next input from thread;
14    process input;
15    pass result to output;
16  }
17 }

```

Model Producent/konzument (Producer/Consumer)

- Předávání dat mezi jednotkami lze realizovat vyrovnávací pamětí.
 - *Nebo jen vyrovnávací paměť odkazů (ukazatelů) na jednotlivé datové jednotky.*
 - Producent – vlákno, které předává data jinému vláknu.
 - Konzument – vlákno, které přijímá data od jiného vlákna.
- Přístup k vyrovnávací paměti musí být synchronizovaný (výhradní přístup).



Použití vyrovnávací paměti nemusí nutně znamenat zkopírování všech dat (např. stačí pouze ukazatel).

Mutex – Zámek kritické sekce

- Mutex je sdílená proměnná přístupná z jednotlivých vláken.
- Základní operace, které mohou vlákna s mutexem provádět.
 - Zamknout mutex (získá mutex volající vlákno).
 - Pokud vlákno nemůže mutex získat (protože ho drží jiné vlákno), vlákno je **blokováno a čeká na uvolnění mutexu**.
 - Odemkne již získaný mutex.
 - Pokud se o získání mutexu pokouší jedno nebo více vláken (voláním zámku na mutexu), je jedno z vláken vybráno pro získání mutexu.

Synchronizační mechanismy

- Synchronizace vláken používá stejné principy jako synchronizace procesů.
 - Protože vlákna sdílejí paměťový prostor s procesem, probíhá hlavní komunikace mezi vlákny prostřednictvím paměti a (globálních) proměnných.
 - Rozhodující je řízení přístupu do stejné paměti.
 - **Exkluzivní (výhradní) přístup ke kritické sekci.**
- Základní synchronizační primitiva jsou **mutexy** a **podmíněné proměnné (Condition Variables)**.
 - **Mutex/Locker** pro exkluzivní přístup ke kritické sekci (mutexy nebo spinlocky).
 - **Podmíněná proměnná** synchronizace vláken podle hodnoty sdílené proměnné.

Spící vlákno může být probuzeno další signalizací od jiného vlákna.

Příklad – Mutex a kritická sekce

- Zamknout/odemknout přístup ke kritické sekci mutexem `drawingMtx`

```

1 void add_drawing_event(void)
2 {
3     Tcl_MutexLock(&drawingMtx);
4     Tcl_Event * ptr = (Tcl_Event*)Tcl_Alloc(sizeof(Tcl_Event));
5     ptr->proc = MyEventProc;
6     Tcl_ThreadQueueEvent(guiThread, ptr, TCL_QUEUE_TAIL);
7     Tcl_ThreadAlert(guiThread);
8     Tcl_MutexUnlock(&drawingMtx);
9 }
    
```

Příklad použití podpory vláken z knihovny TCL.

- Příklad použití konceptu `ScopedLock`

```

1 void CCanvasContainer::draw(cairo_t *cr)
2 {
3     ScopedLock lk(mtx);
4     if (drawer == 0) {
5         drawer = new CCanvasDrawer(cr);
6     } else {
7         drawer->setCairo(cr);
8     }
9     manager.execute(drawer);
10 }
    
```

ScopedLock uvolní (odemkne) mutex, jakmile je lokální proměnná lk na konci volání funkce zničena.

Zobecněné modely mutexu

- Rekurzivní – mutex může být stejným vláknem uzamčen vícekrát.
- Try – operace uzamčení se okamžitě vrátí, pokud mutex nelze získat.
- Timed – omezuje dobu získání mutexu.
- *Spinlock* – vlákno opakovaně kontroluje, zda je zámek k dispozici pro získání.
Vlákno není nastaveno do blokováného režimu, pokud zámek nelze získat.

Podmíněná proměnná

- **Podmíněná proměnná** umožňuje signalizaci vlákna z jiného vlákna.
- Koncept **podmíněné proměnné** umožňuje následující synchronizační operace.
 - Čekat – proměnná byla změněna/oznámena.
 - Časované čekání na signál z jiného vlákna.
 - Signalizace jiného vlákna čekajícího na proměnnou podmínky.
 - Signalizace všech vláken čekajících na proměnnou podmínky.
Všechna vlákna jsou probuzena, ale přístup k podmíněné proměnné je chráněn mutexem, který je třeba získat, pouze jedno vlákno může mutex zamknout.

Spinlock

- Za určitých okolností může být výhodné neblokovat vlákno během získávání mutexu (zámku), např.,
 - Provedení jednoduché operace se sdílenými daty/proměnnými v systému se skutečným paralelismem (s použitím vícejádrového procesoru).
 - Zablokování vlákna, pozastavení jeho provádění a předání přiděleného času CPU jinému vláknou může vést ke značné režii.
 - Jiná vlákna rychle provedou jinou operaci s daty, sdílený prostředek by tak byl rychle přístupný.
- Během zamykání vlákno aktivně testuje, zda je zámek volný.
Plýtvá časem procesoru, který lze využít k produktivním výpočtům jinde.
- Podobně jako u semaforu musí být takový test proveden instrukcí TestAndSet na úrovni CPU.
- **Adaptivní mutex** kombinuje oba přístupy a používá **spinlocks** pro přístup ke zdrojům uzamčeným právě běžícím vláknem a blokuje/uspává, pokud takové vlákno neběží.
Na jednoprocessorových systémech s pseudoparalelismem nemá smysl používat spinlocky.

Příklad – Použití podmíněné proměnné

- Příklad použití podmíněné proměnné se zámkem (mutexem), který umožňuje exkluzivní přístup k podmíněné proměnné z různých vláken.

```

Mutex mtx; // shared variable for both threads
CondVariable cond; // shared condition variable

// Thread 1
Lock(mtx);
// Before code, wait for Thread 2
CondWait(cond, mtx); // wait for cond
... // Critical section
UnLock(mtx);

// Thread 2
Lock(mtx);
... // Critical section
// Signal on cond
CondSignal(cond, mtx);
UnLock(mtx);

```


Paralelismus a funkce

- V paralelním prostředí lze funkce volat vícekrát.
- Pokud jde o paralelní provádění, funkce mohou být **reentrantní** nebo **thread-safe**.
 - **Reentrant** – v jednom okamžiku může být stejná funkce provedena vícekrát současně.
 - **Thread-Safe** – funkce může být volána více vlákny současně.
- Pro dosažení těchto vlastností je třeba splnit následující podmínky.
 - **Reentrantní funkce** nezapíše do statických dat a nepracuje s globálními daty.
 - **Vláknově bezpečná funkce** (thread-safe) přistupuje ke globálním datům s využitím synchronizačních primitiv.

POSIX Funkce knihovny pthread

- Knihovna POSIXových vláken (<pthread.h> a -lpthread) je sada funkcí pro podporu vícevláknového programování.
- Základní typy pro vlákna, mutexy a podmíněné proměnné jsou
 - **pthread_t** – typ pro reprezentaci vlákna;
 - **pthread_mutex_t** – typ pro mutex;
 - **pthread_cond_t** – typ pro proměnnou podmínky.
- Vlákno je vytvořeno voláním funkce **pthread_create()**, která okamžitě spustí nové vlákno jako funkci předanou jako ukazatel na funkci.

Vlákno volající vytvoření pokračuje ve vykonávání.
- Vlákno může čekat na jiné vlákno funkcí **pthread_join()**.
- Partikulární mutex a podmíněné proměnné musí být inicializovány voláním knihovnických funkcí.

Poznámka, inicializované sdílené proměnné před vytvořením vlákna.

 - **pthread_mutex_init()** – inicializace proměnné mutex.
 - **pthread_cond_init()** – inicializace podmíněné proměnné.

Lze nastavit další atributy, viz dokumentace.

Hlavní problémy vícevláknových aplikací

- Hlavní problémy s vícevláknovými aplikacemi se týkají synchronizace.
 - **Uváznutí (Deadlock)** – vlákno čeká na prostředek (mutex), který je právě uzamčen jiným vláknem, které čeká na prostředek (vlákno) již uzamčený prvním vláknem.
 - **Souběh (Race condition)** – přístup několika vláken ke sdíleným prostředkům (paměti/proměnným) a alespoň jedno z vláken nepoužívá synchronizační mechanismy (např. kritickou sekci).

Vlákno čte hodnotu, zatímco jiné vlákno ji zapisuje. Pokud operace čtení/zápisu nejsou atomické, data nejsou platná.

Vláčna POSIX – Příklad 1/10

- Vytvoření aplikace se třemi aktivními vlákny.
 - Obsluha uživatelského vstupu – funkce **input_thread()**.
 - Uživatel zadá periodu výstupu obnovení stisknutím vyhrazených kláves.
 - Zobrazení výstupu – funkce **output_thread()**.
 - Aktualizce výstupu pouze tehdy, když uživatel interaguje s aplikací nebo když alarm signalizuje, že uplynula perioda.
 - Alarm s periodou definovanou uživatelem – funkce **alarm_thread()**.
 - Obnovení výstupu nebo provedení jiné akce.
- Pro zjednodušení program používá **stdin** a **stdout** s hlášením aktivity vlákna do **stderr**.
- Synchronizační mechanismy jsou demonstrují použití mutexu a podmíněné proměnné.
 - **pthread_mutex_t mtx** – výhradní přístup k **data_t data**;
 - **pthread_cond_t cond** – signalizace vláken.

*Sdílená data se skládají z aktuální periody alarmu (**alarm_period**), požadavku na ukončení aplikace (**quit**) a počtu vyvolání alarmu (**alarm_counter**).*

Vlákna POSIX – Příklad 2/10

- Včetně hlavičkových souborů, definice datových typů, deklarace globálních proměnných.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <termios.h>
5 #include <unistd.h> // for STDIN_FILENO
6 #include <pthread.h>
7 #define PERIOD_STEP 10
9 #define PERIOD_MAX 2000
10 #define PERIOD_MIN 10
11 typedef struct {
13     int alarm_period;
14     int alarm_counter;
15     bool quit;
16     pthread_mutex_t *mtx; // avoid global variables for mutex and
17     pthread_cond_t *cond; // conditional variable
18 } data_t; // data structure shared among the threads

```

Vlákna POSIX – Příklad 4/10

- Vytvoření vláken a čekání na ukončení všech vláken.

```

43 call_termios(0); // switch terminal to raw mode
44 for (int i = 0; i < NUM_THREADS; ++i) {
45     int r = pthread_create(&threads[i], NULL, thr_functions[i], &data);
46     printf("Create thread '%s' %s\r\n", threads_names[i], ( r == 0 ? "OK" : "FAIL" ) );
47 }
48 int *ex;
49 for (int i = 0; i < NUM_THREADS; ++i) {
50     printf("Call join to the thread %s\r\n", threads_names[i]);
51     int r = pthread_join(threads[i], (void*)&ex);
52     printf("Joining the thread %s has been %s - exit value %i\r\n", threads_names[i],
53           (r == 0 ? "OK" : "FAIL"), *ex);
54 }
55 call_termios(1); // restore terminal settings
56 return EXIT_SUCCESS;
57 }

```

Vlákna POSIX – Příklad 3/10

- Funkce prototypů a inicializace proměnných a struktur.

```

21 void call_termios(int reset); // switch terminal to raw mode
22 void* input_thread(void*);
23 void* output_thread(void*);
24 void* alarm_thread(void*);
25 // - main function -----
26 int main(int argc, char *argv[])
27 {
28     data_t data = { .alarm_period = 100, .alarm_counter = 0, .quit = false };
29     enum { INPUT, OUTPUT, ALARM, NUM_THREADS }; // named ints for the threads
30     const char *threads_names[] = { "Input", "Output", "Alarm" };
31     void* (*thr_functions[])(void*) = {
32         input_thread, output_thread, alarm_thread // array of thread functions
33     };
34     pthread_t threads[NUM_THREADS]; // array for references to created threads
35     pthread_mutex_t mtx;
36     pthread_cond_t cond;
37     pthread_mutex_init(&mtx, NULL); // initialize mutex with default attributes
38     pthread_cond_init(&cond, NULL); // initialize condition variable with default attributes
39     data.mtx = &mtx; // make the mutex accessible from the shared data structure
40     data.cond = &cond; // make the cond accessible from the shared data structure

```

Vlákna POSIX – Příklad 5/10 (Přepnutí terminálu)

- Přepnutí terminálu do režimu raw.

```

59 void call_termios(int reset)
60 {
61     static struct termios tio, tioOld; // use static to preserve the initial
62     settings
63     tcgetattr(STDIN_FILENO, &tio);
64     if (reset) {
65         tcsetattr(STDIN_FILENO, TCSANOW, &tioOld);
66     } else {
67         tioOld = tio; //backup
68         cfmakeraw(&tio);
69         tcsetattr(STDIN_FILENO, TCSANOW, &tio);
70     }

```

Volající je zodpovědný za vhodné volání funkce, např. pro zachování původního nastavení musí být funkce volána s argumentem 0 pouze jednou.

Vlákna POSIX – Příklad 6/10 (Vstupní vlákno 1/2)

```

72 void* input_thread(void* d)
73 {
74     data_t *data = (data_t*)d;
75     static int r = 0;
76     int c;
77     while ((c = getchar()) != 'q') {
78         pthread_mutex_lock(data->mtx);
79         int period = data->alarm_period; // save the current period
80         // handle the pressed key detailed in the next slide
81     }
82     ...
83     if (data->alarm_period != period) { // the period has been changed
84         pthread_cond_signal(data->cond); // signal the output thread to refresh
85     }
86     data->alarm_period = period;
87     pthread_mutex_unlock(data->mtx);
88 }
89 r = 1;
90 pthread_mutex_lock(data->mtx);
91 data->quit = true;
92 pthread_cond_broadcast(data->cond);
93 pthread_mutex_unlock(data->mtx);
94 fprintf(stderr, "Exit input thread %lu\r\n", pthread_self());
95 return &r;
96 }

```

Vlákna POSIX – Příklad 8/10 (výstupní vlákno)

```

96 void* output_thread(void* d)
97 {
98     data_t *data = (data_t*)d;
99     static int r = 0;
100     bool q = false;
101     pthread_mutex_lock(data->mtx);
102     while (!q) {
103         pthread_cond_wait(data->cond, data->mtx); // wait for next event
104         q = data->quit;
105         printf("\rAlarm time: %10i Alarm counter: %10i", data->alarm_period,
106             data->alarm_counter);
107         fflush(stdout);
108     }
109     pthread_mutex_unlock(data->mtx);
110     fprintf(stderr, "Exit output thread %lu\r\n", (unsigned long)pthread_self());
111     return &r;
112 }

```

Vlákna POSIX – Příklad 7/10 (Vstupní vlákno 2/2)

- input_thread() – zpracuje požadavek uživatele na změnu periody.

```

81 switch(c) {
82     case 'r':
83         period -= PERIOD_STEP;
84         if (period < PERIOD_MIN) {
85             period = PERIOD_MIN;
86         }
87         break;
88     case 'p':
89         period += PERIOD_STEP;
90         if (period > PERIOD_MAX) {
91             period = PERIOD_MAX;
92         }
93         break;
94 }

```

Vlákna POSIX – Příklad 9/10 (Alarm vlákno)

```

113 void* alarm_thread(void* d)
114 {
115     data_t *data = (data_t*)d;
116     static int r = 0;
117     pthread_mutex_lock(data->mtx);
118     bool q = data->quit;
119     useconds_t period = data->alarm_period * 1000; // alarm_period is in ms
120     pthread_mutex_unlock(data->mtx);
121     while (!q) {
122         usleep(period);
123         pthread_mutex_lock(data->mtx);
124         q = data->quit;
125         data->alarm_counter += 1;
126         period = data->alarm_period * 1000; // update the period is it has been changed
127         pthread_cond_broadcast(data->cond);
128         pthread_mutex_unlock(data->mtx);
129     }
130 }
131 fprintf(stderr, "Exit alarm thread %lu\r\n", pthread_self());
132 return &r;
133 }

```

Vlákna POSIX – Příklad 10/10

- Příkladový program `lec11/threads.c` lze zkompileovat a spustit.

```
clang -c threads.c -std=gnu99 -O2 -pedantic -Wall -o threads.o
clang threads.o -pthread -o threads
```

- Periodu lze změnit klávesami 'r' a 'p'.

- Aplikace je ukončena po stisknutí 'q'.

```
./threads
Create thread 'Input' OK
Create thread 'Output' OK
Create thread 'Alarm' OK
Call join to the thread Input
Alarm time:      110 Alarm counter:      20Exit input thread 750871808
Alarm time:      110 Alarm counter:      20Exit output thread 750873088
Joining the thread Input has been OK - exit value 1
Call join to the thread Output
Joining the thread Output has been OK - exit value 0
Call join to the thread Alarm
Exit alarm thread 750874368
Joining the thread Alarm has been OK - exit value 0
```

`lec11/threads.c`

Příklad vláken C11

- Předchozí příklad `lec11/threads.c` implementovaný s vlákny C11 je v `lec11/threads-c11.c`.

```
clang -std=c11 threads-c11.c -stdthreads -o threads-c11
./threads-c11
```

- Volání funkcí je v podstatě podobné, jen se liší názvy a drobnými úpravami.

- `pthread_mutex_*`() → `mxt_*`().
- `pthread_cond_*`() → `cnd_*`().
- `pthread_*`() → `thrd_*`().
- Funkce těla vlákna vrací hodnotu `int`.
- Neexistuje ekvivalent `pthread_self`().
- `thrd_t` závisí na implementaci.
- Vlákna, mutexy a podmíněné proměnné se vytvářejí/inicializují bez specifikace konkrétních atributů. *Zjednodušené rozhraní.*
- Program je spojen s knihovnou `-stdthreads`.

`lec11/threads-c11.c`

Vlákna v C11

- C11 poskytuje „obal“ pro POSIXová vlákna.

Např. viz <http://en.cppreference.com/w/c/thread>.

- Knihovna je `<threads.h>` a `-stdthreads`.

- Základní typy

- `thrd_t` – typ pro reprezentaci vlákna;
- `mtx_t` – typ pro mutex;
- `cnd_t` – typ pro podmíněné proměnné.

- Vytvoření vlákna je `thrd_create()` a funkce těla vlákna musí vrátit hodnotu `int`.

- `thrd_join()` se používá k čekání na ukončení vlákna.

- Mutex a podmíněná proměnná jsou inicializovány (bez atributů).

- `mtx_init()` – inicializuje proměnnou mutex;
- `cnd_init()` – inicializace podmíněnou proměnnou.

Jak ladit vícevláknové aplikace

- Nejlepším nástrojem pro ladění vícevláknových aplikací je `abyste ji nemuseli ladit`.
- Toho lze dosáhnout disciplínou a obezřetným přístupem ke sdíleným proměnným.
- V opačném případě lze využít ladicí program s minimální sadou funkcí.

Podpora ladění

- Požadované funkce ladicího programu.
 - Seznam běžících vláken.
 - Stav synchronizačních primitiv.
 - Přístup k proměnným vláken.
 - Body přerušení (*break points*) v jednotlivých vláknech.
- `lldb` – <http://lldb.linux.org>; `gdb` – <https://www.sourceware.org/gdb.cgdg, ddd, kgdb, Code::Blocks nebo Eclipse, Kdevelop, Netbeans, CLion>.
- `SlickEdit` – <https://www.slickedit.com>; `TotalView` – <http://www.roguewave.com/products-services/totalview>
- **Logování (logging)** může být pro ladění programu efektivnější než ruční ladění s ručně nastavenými body přerušení.
 - Slepá ulička většinou souvisí s pořadím zamykání.
 - Zaznamenávání a analýza přístupu k zámkům (mutexům) může pomoci najít nesprávné pořadí synchronizačních operací vláken.

Komentáře – Uvážnutí (*Deadlock*)

- Uvážnutí souvisejí s mechanismy synchronizace.
 - Uvážnutí je typicky mnohem snazší odladit než souběh.
 - Uvážnutí je často *mutexový deadlock* způsobený pořadím vícenásobného zamykání mutexu.
 - **Mutexový deadlock nemůže nastat**, pokud v každém okamžiku má (nebo se snaží získat) každé vlákno **nejvýše jeden mutex**.
 - Nedoporučuje se volat funkce se zamčeným mutexem, zejména pokud se funkce pokouší zamknout jiný mutex.
 - **Doporučuje se uzamknout mutex na co nejkratší dobu.**

Komentáře – Souběh (*Race Condition*)

- Souběh je obvykle způsoben nedostatečnou synchronizací.
- Je vhodné si zapamatovat následující.
 - **Vlákna jsou asynchronní!**
Nepředpokládejte, že provádění kódu je synchronní na jednoprosesorovém systému.
 - Při psaní vícevláknových aplikací předpokládejte, že vlákno může být kdykoli přerušeno nebo spuštěno!
Části kódu, které vyžadují určité pořadí provádění vláken, potřebují synchronizaci.
 - **Nikdy nepředpokládejte, že vlákno čeká po svém vytvoření!**
Může být spuštěno velmi brzy a obvykle mnohem dříve, než očekáváte.
 - **Pokud nespecifikujete pořadí provádění vlákna, žádné takové pořadí neexistuje!**
“Vlákna běží v nejhorším možném pořadí”. Bill Gallmeister”

Shrnutí přednášky

Diskutovaná témata

- Vícevláknové programování
 - Terminologie, koncepty a motivace vícevláknového programování
 - Modely vícevláknových aplikací
 - Synchronizační mechanismy
 - Knihovny vláken POSIX a C11

Příklad aplikace

- Komentáře k ladění a problematice vícevláknových aplikací – souběhu a uváznutí