

Paralelní programování

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 10

BAB36PRGA – Programování v C

Přehled témat

- Část 1 – Úvod do paralelního programování

Úvod

Paralelní výpočet/zpracování

Semaforey

Sdílená paměť

Zprávy

Část I

Část 1 – Úvod do paralelního programování

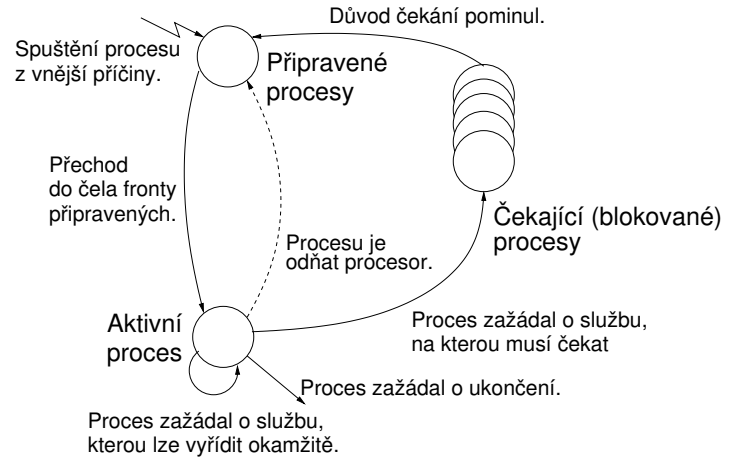
Paralelní programování

- Myšlenka paralelního programování pochází z 60. let 20. století, kdy vznikly první víceprogramové a pseudoparalelní systémy.
- Paralelismus může být hardwarový nebo softwarový.
 - Hardwarový – skutečný hardwarový paralelismus víceprocesorových systémů.
 - Softwarový – pseudoparalelismus.
- Pseudoparalelismus – program s paralelními konstrukcemi může běžet v pseudoparalelním prostředí na jedno- nebo víceprocesorových systémech.

Motivace, proč se zabývat paralelním programováním

- Zvýšení výpočetního výkonu.
 - S víceprocesorovým systémem můžeme řešit výpočetní problém rychleji.
- Efektivní využití výpočetního výkonu.
 - I běžící program může čekat na data.
 - Např. běžný program s interakcí s uživatelem obvykle čeká na vstup uživatele.
- Současné zpracování mnoha požadavků.
 - Zpracování požadavků od jednotlivých klientů v architektuře klient/server.

Stavy procesu



Proces – Spuštěný program

- Proces je vykonávaný program běžící ve vyhrazeném paměťovém prostoru.
- Proces je entita operačního systému (OS), která je rozvrhována pro nezávislé provádění.
- Proces se obvykle nachází v jednom ze tří základních stavů:
 - Provádí se – právě běží na procesoru (CPU);
 - Blocked – čeká na periférii;
 - Waiting – čeká na procesor .
- Proces je v operačním systému identifikován svým identifikátorem, např. *Process IDentifier* – PID.
- Plánovač OS spravuje běžící procesy, které mají být přiděleny dostupným procesorům.

Víceprocesorové systémy

- Víceprocesorové systémy umožňují skutečný paralelismus.
- Je nutné synchronizovat procesory a podporovat datovou komunikaci.
 - Zdroje pro synchronizaci činností.
 - Prostředky pro komunikaci mezi procesory (procesy).

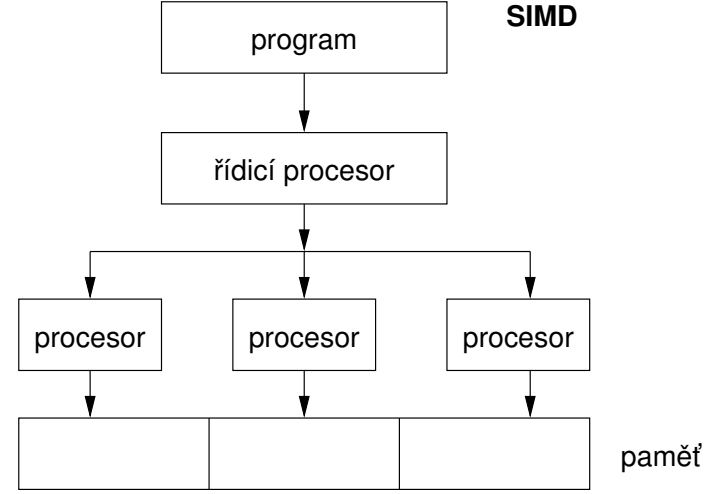
Možné architektury paralelní provádění programů (výpočtů)

- Řízení jednotlivých instrukcí.
 - **SIMD** – Single-Instruction, Multiple-Data – stejné instrukce jsou prováděny současně na různých datech.
 - "Procesory" jsou identické a běží synchronně.
 - Např. "vektorizace", jako MMX, SSE, 3Dnow! a AVX, AVX2 atd.
 - **MIMD** – Multiple-Instruction, Multiple-Data – procesory pracují nezávisle a asynchronně.
- Řízení přístupu do paměti.
 - Systémy se sdílenou pamětí – centrální sdílená paměť.

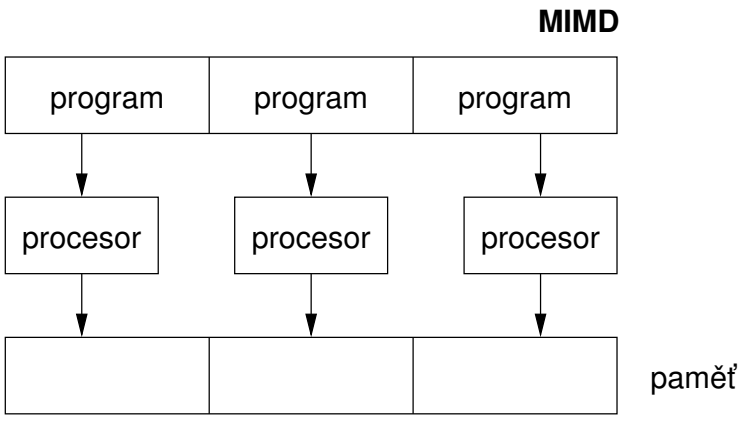
Např. vícejádrové procesory.
 - Systémy s distribuovanou pamětí – každý procesor má vlastní paměť.

Např. výpočetní síť.

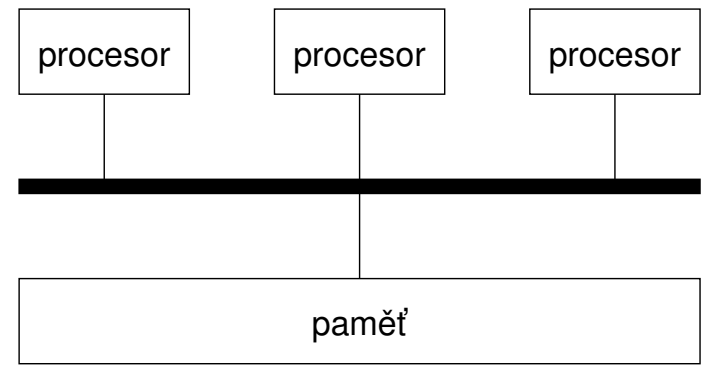
SIMD – Single-Instruction, Multiple-Data



MIMD – Multiple-Instruction, Multiple-Data

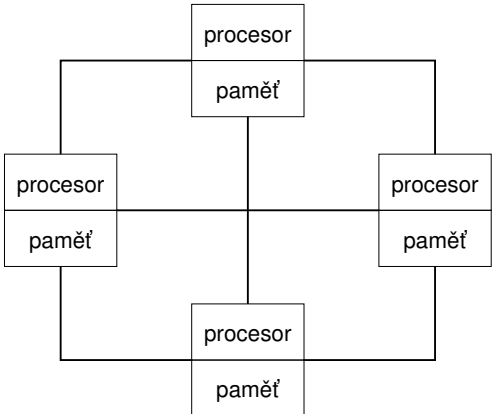


Systémy se sdílenou pamětí



- Procesory komunikují prostřednictvím sdílené paměti.
- Procesory mohou také synchronizovat své činnosti, tj. poskytovat exkluzivní přístup k paměti.

Systémy s distributivní pamětí



- S výhradním přístupem do paměti není problém.
- Je však nutné řešit komunikaci mezi procesory.

Paralelní zpracování a programovací jazyky

- Pokud jde o programovací jazyky s podporou paralelního zpracování, lze je rozdělit na jazyky bez explicitní podpory paralelismu a s explicitní podporou paralelismu.
 - Bez explicitní podpory paralelismu – možné mechanismy paralelního zpracování.
 1. Paralelní zpracování je realizováno kompilátorem a operačním systémem.
 2. Paralelní konstrukce jsou explicitně označeny pro kompilátor.
 3. Paralelní zpracování je realizováno systémovými voláními operačního systému.
 - S explicitní podporou paralelismu.

Úloha operačního systému (OS)

- OS poskytuje hardwarovou abstrakční vrstvu – zapouzdřuje HW a odděluje uživatele od konkrétní hardwarové architektury (pravý/pseudoparalelismus).
- OS je zodpovědný za synchronizaci běžících procesů.
- OS poskytuje uživatelská rozhraní (systémová volání).
 - K vytváření a ukončování procesů.
 - Ke správě procesů a procesorů.
 - Plánovat procesory na dostupných procesorech.
 - Řízení přístupu ke sdílené paměti.
 - Mechanismy pro meziprocsovou komunikaci (IPC).
 - Mechanismy pro synchronizaci procesů.

Příklad paralelního výpočtu realizovaného překladačem 1/2

Příklad – násobení pole

```

1 #include "my_malloc.h"
2 #define SIZE 30000000
3 int main(int argc, char *argv[])
4 {
5     int i;
6     int *in1 = (int*)myMalloc(SIZE * sizeof(int));
7     int *in2 = (int*)myMalloc(SIZE * sizeof(int));
8     int *out = (int*)myMalloc(SIZE * sizeof(int));
9     for (i = 0; i < SIZE; ++i) {
10        in1[i] = i;
11        in2[i] = 2 * i;
12    }
13    for (i = 0; i < SIZE; ++i) {
14        out[i] = in1[i] * in2[i];
15        out[i] = out[i] - (in1[i] + in2[i]);
16    }
17    return 0;
18 }

```

Příklad paralelního výpočtu realizovaného překladačem 2/2

Příklad 1 Příklad 2

```

1 icc compute.c
2 time ./a.out
4 real 0m0.562s
5 user 0m0.180s
6 sys 0m0.384s

1 icc -m SSE compute.c; time ./a.out
2 compute.c(8) : (col. 2) remark: LOOP WAS VECTORIZED.
4 real 0m0.542s
5 user 0m0.136s
6 sys 0m0.408s

```

Příklad 3

```

1 icc -parallel compute.c; time ./a.out
2 compute.c(12) : (col. 2) remark: LOOP WAS AUTO-PARALLELIZED.
4 real 0m0.702s
5 user 0m0.484s
6 sys 0m0.396s

```

Příklad – OpenMP – Násobení matic 1/2

- Open Multi-Processing (OpenMP) - aplikační programové rozhraní pro multiplatformní *multiprocessing* se sdílenou pamětí. <http://www.openmp.org>
- Makry můžeme kompilátor instruovat v vytvoření paralelní konstrukce, například paralelizaci přes vnější smyčku přes proměnnou *i*.

```

1 void multiply(int n, int a[n][n], int b[n][n], int c[n][n])
2 {
3     int i;
4     #pragma omp parallel private(i)
5     #pragma omp for schedule (dynamic, 1)
6     for (i = 0; i < n; ++i) {
7         for (int j = 0; j < n; ++j) {
8             c[i][j] = 0;
9             for (int k = 0; k < n; ++k) {
10                c[i][j] += a[i][k] * b[k][j];
11            }
12        }
13    }
14 }

```

lec10/demo-omp-matrix.c
Pro zjednodušení jsou použity čtvercové matice stejných rozměrů.

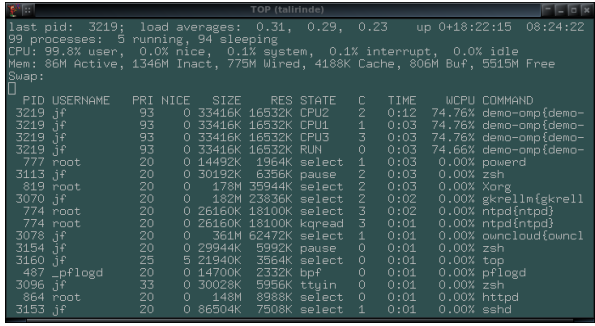
Příklad – OpenMP – Násobení matic 2/2

- Porovnání násobení 1000x 1000 matic s OpenMP na iCore5 (2 jádra s HT).

```

1 gcc -std=c99 -O2 -o demo-omp demo-omp-matrix.c -fopenmp
2 ./demo-omp 1000
3 Size of matrices 1000 x 1000 naive
4 multiplication with O(n^3)
5 c1 == c2: 1
6 Multiplication single core 9.33 sec
7 Multiplication multi-core 4.73 sec
8 export OMP_NUM_THREADS=2
9 ./demo-omp 1000
10 Size of matrices 1000 x 1000 naive
11 multiplication with O(n^3)
12 c1 == c2: 1
13 Multiplication single core 9.48 sec
14 Multiplication multi-core 6.23 sec

```



Použijte např. program top pro zobrazení běžících procesů/vláken.
lec10/demo-omp-matrix.c

Jazyky s explicitní podporou paralelismu

- Má podporu pro vytváření nových procesů.
 - Spuštěný proces vytvoří kopii sebe sama.
 - Oba procesy provádějí totožný kód (zkopírovaný).
 - Proces **parent** (rodič) a proces **child** (dítě) jsou rozlišeny identifikátorem procesu (PID).
 - Segment kódu (paměť s instr. programu) je explicitně spojen s novým procesem.
- Bez ohledu na to, jak je nový proces vytvořen, nejdůležitější je vztah k provádění nadřazeného procesu a přístupu do paměti.
 - Zastaví nadřazený proces své provádění až do konce podřízeného procesu?
 - Je paměť sdílená podřízeným a rodičovským procesem?
- Granularita procesů – paralelismus od úrovně instrukcí až po paralelismus programů.

Paralelismus – úroveň příkazů

Příklad – blok parbegin-parend

```
parbegin
  S1;
  S2;
  ...
  Sn
parend
```

- Příkazy S1 jsou Sn prováděny paralelně.
- Provádění hlavního programu je přerušeno, dokud nejsou ukončeny všechny příkazy S1 až Sn.
- Příkazy S1 jsou Sn prováděny paralelně.

Příklad – doparalelní

```
1 for i = 1 to n doparalel {
2   for j = 1 to n do {
3     c[i,j] = 0;
4     for k = 1 to n do {
5       c[i,j] = c[i,j] + a[i,k]*b[k,j];
6     } } }
```

Paralelní provádění vnější smyčky nad všemi i.

Například OpenMP v C.

Paralelismus – úroveň procedur

- Procedura je spojena s procesem provádění.

```
...
procedure P;
...
PID xpid = newprocess(P);
...
killprocess(xpid);
```

- P je procedura a xpid je identifikátor procesu.
- Přřazení procedury/funkce k procesu v deklaraci PID xpid process(P).
 - Proces je vytvořen při vytvoření proměnné x.
 - Proces je ukončen na konci x nebo dříve.

Např. vlákna (pthreads) v jazyce C.

Paralelismus – úroveň programu (procesu)

- Nový proces může být pouze celý program.
- Nový program je vytvořen systémovým voláním, které v okamžiku volání vytvoří kompletní kopii sebe sama včetně všech proměnných a dat.

Příklad - Vytvoření kopie procesu systémovým voláním fork

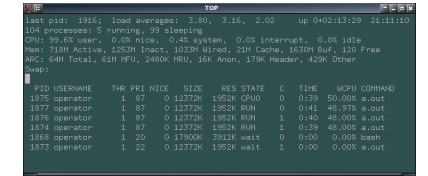
```
1 if (fork() == 0) {
2   /* code executed by the child process */
3 } else {
4   /* code executed by the parent process */
5 }
```

Např. fork() v jazyce C

Příklad – fork()

```
1 #define NUMPROCS 4
2 for (int i = 0; i < NUMPROCS; ++i) {
3   pid_t pid = fork();
4   if (pid == 0) {
5     compute(i, n);
6     exit(0);
7   } else {
8     printf("Child %d created\n", pid);
9   }
10 }
11 printf("All processes created\n");
12 for (int i = 0; i < NUMPROCS; ++i) {
13   pid_t pid = wait(&r);
14   printf("Wait for pid %d return: %d\n", pid, r);
15 }
16 void compute(int myid, int n)
17 {
18   printf("Process myid %d start computing\n", myid);
19   ...
20   printf("Process myid %d finished\n", myid);
21 }
```

lec10/demo-fork.c



```
clang demo-fork.c && ./a.out
Child 2049 created
Process myid 0 start computing
Child 2050 created
Process myid 1 start computing
Process myid 2 start computing
Child 2051 created
Child 2052 created
Process myid 3 start computing
All processes created
Process myid 1 finished
Process myid 0 finished
Wait for pid 2050 return: 0
Process myid 3 finished
Process myid 2 finished
Wait for pid 2049 return: 0
Wait for pid 2051 return: 0
Wait for pid 2052 return: 0
```

Úvod	Paralelní výpočet/zpracování	Semafor	Sdílená paměť	Zprávy
<h2>Semafor</h2> <ul style="list-style-type: none"> E. W. Dijkstra – Semafor je mechanismus pro synchronizaci paralelních procesů se sdílenou pamětí. Semafor je celočíselná proměnná s následujícími operacemi. <ul style="list-style-type: none"> <i>InitSem</i> - inicializace. <i>Wait</i> <ul style="list-style-type: none"> Pokud $S > 0$, pak $S \leftarrow S - 1$ (zdroje jsou k dispozici, v tomto případě získáme jeden). V opačném případě pozastaví provádění volajícího procesu (počkejte, až se S stane $S > 0$). <i>Signal</i> <ul style="list-style-type: none"> Jestliže existuje čekající proces, probudte ho (nechte proces získat jeden prostředek).. V opačném případě zvýšíme hodnotu S o jedničku, tj. na $S \leftarrow S + 1$. (uvolnit jeden prostředek). Semafor lze použít k řízení přístupu ke sdílenému prostředku. <ul style="list-style-type: none"> $S < 0$ – sdílený prostředek je používán. Proces požádá o přístup ke zdroji a čeká na jeho uvolnění. $S > 0$ - sdílený prostředek je k dispozici. Proces prostředek uvolní. <p><i>Hodnota semaforu může představovat počet dostupných zdrojů. Pak můžeme získat (nebo čekat na) k zdrojů – wait(k): $S \leftarrow S - k$ pro $S > k$ a také uvolnit k zdrojů – signal(k): $S \leftarrow S + k$.</i></p> 				
Jan Faigl, 2024	BAB36PRGA – Přednáška 10: Paralelní programování			28 / 52

Úvod	Paralelní výpočet/zpracování	Semafor	Sdílená paměť	Zprávy
<h2>Použití semaforů</h2> <ul style="list-style-type: none"> Semafor lze využít pro definování kritických sekcí. Kritická sekce je část programu, kde musí být zaručen exkluzivní přístup ke sdílené paměti (zdrojům). <p>Příklad kritické sekce chráněné semaforem</p> <pre>InitSem(S,1); Wait(S); /* Kód kritické sekce */ Signal(S);</pre> <ul style="list-style-type: none"> Synchronizace procesů semaforem. <p>Příklad synchronizace procesů.</p> <pre>/* process p */ ... InitSem(S,0) Wait(S); ... exit(); /* process q */ ... Signal(S); exit();</pre>				
Jan Faigl, 2024	BAB36PRGA – Přednáška 10: Paralelní programování			30 / 52

Úvod	Paralelní výpočet/zpracování	Semafor	Sdílená paměť	Zprávy
<h2>Implementace semaforu</h2> <ul style="list-style-type: none"> Operace se semaforem musí být atomické. <p><i>Processor nemůže být během provádění operace přerušen.</i></p> Strojová instrukce <i>TestAndSet</i> načte a uloží obsah adresovaného paměťového prostoru a nastaví paměť na nenulovou hodnotu. Během provádění instrukce <i>TestAndSet</i> drží procesor systémovou sběrnici a přístup do paměti není povolen žádnému jinému procesoru. 				
Jan Faigl, 2024	BAB36PRGA – Přednáška 10: Paralelní programování			29 / 52

Úvod	Paralelní výpočet/zpracování	Semafor	Sdílená paměť	Zprávy
<h2>Příklad – Semafor 1/4 (systémové volání)</h2> <ul style="list-style-type: none"> Semafor je entita operačního systému (OS). <pre>1 #include <sys/types.h> 2 #include <sys/ipc.h> 3 #include <sys/sem.h> /* create or get existing set of semphores */ 6 int semget(key_t key, int nsems, int flag); /* atomic array of operations on a set of semphores */ 9 int semop(int semid, struct sembuf *array, size_t nops); /* control operations on a st of semaphores */ 12 int semctl(int semid, int semnum, int cmd, ...);</pre>				
Jan Faigl, 2024	BAB36PRGA – Přednáška 10: Paralelní programování			31 / 52

Příklad – Semafor 2/4 (synchronizační protokol)

- Příklad, kdy hlavní (primární) proces čeká, až budou připraveny dva další procesy (sekundární).
 1. Primární proces pozastaví provádění a čeká, až budou připraveny dva další sekundární procesy.
 2. Sekundární procesy pak čekají na uvolnění primárním procesem.
- Navrhovaný synchronizační "protokol".
 - Definuje náš způsob synchronizace procesů s využitím semaforů OS.
 - Sekundární proces zvýší semafor o 1.
 - Sekundární proces čeká, až semafor získá hodnotu 0, a pak je ukončen.
 - Primární proces čeká na dva sekundární procesy a sníží semafor o 2.
 - Musí také zajistit, aby hodnota semaforu nebyla 0; jinak by sekundární procesy byly předčasně ukončeny.
 - Musíme použít atomické operace se semaforem.

```
lec10/sem-primary.c lec10/sem-secondary.c
```

Příklad – Semafor 4/4 (sekundární proces)

```

1 int main(int argc, char* argv[])
2 {
3     struct sembuf sem;
4     int id = semget(1000, 1, 0);
5     int r;
6     if (id != -1) {
7         sem.sem_num = 0; // add the secondary process
8         sem.sem_op = 1; // to the "pool" of resources
9         sem.sem_flg = 0;
10        printf("Increase semaphore value (add resource)\n");
11        r = semop(id, &sem, 1);
12        sem.sem_op = 0;
13        printf("Semaphore value is %d\n", semctl(id, 0, GETVAL, 0));
14        printf("Wait for semaphore value 0\n");
15        r = semop(id, &sem, 1);
16        printf("Done\n");
17    }
18    return 0;
19 }

```

- Entity IPC lze zobrazit nástrojem `ipcs`.


```
clang sem-primary.c -o sem-primary
clang sem-secondary.c -o sem-secondary
```

Příklad – Semafor 3/4 (primární proces)

```

1 int main(int argc, char* argv[])
2 {
3     struct sembuf sem[2]; // structure for semaphore atomic operations
4     int id = semget(1000, 1, IPC_CREAT | 0666); // create semaphore
5     if (id != -1) {
6         int r = semctl(id, 0, SETVAL, 0) == 0;
7         sem[0].sem_num = 0; // operation to acquire semaphore
8         sem[0].sem_op = -2; // once its value will be >= 2
9         sem[0].sem_flg = 0; // representing two secondary processes are ready
10        sem[1].sem_num = 0; // the next operation in the atomic set
11        sem[1].sem_op = 2; // of operations increases the value of
12        sem[1].sem_flg = 0; // the semaphore about 2
13        printf("Wait for semvalue >= 2\n");
14        r = semop(id, sem, 2); // perform all (two) operations (in sem array) atomically
15        printf("Press ENTER to set semaphore to 0\n");
16        getchar();
17        r = semctl(id, 0, SETVAL, 0) == 0; // set the value of semaphore
18        r = semctl(id, 0, IPC_RMID, 0) == 0; // remove the semaphore
19    }
20    return 0;
21 }

```

```
lec10/sem-primary.c
```

Problémy se semaforů

- Hlavní problémy vyplývají z nesprávného použití.
- Typické chyby jsou následující.
 - Špatně identifikovaná kritická sekce.
 - Proces může být zablokovaný vícenásobným voláním `Wait(S)`.
 - Např. **uváznutí (deadlock)** může vzniknout, např. v následujících situacích.

Příklad – Deadlock

```

/* Process 1*/
...
Wait(S1);
Wait(S2);
...
Signal(S2);
Signal(S1);
...

/* Process 2*/
...
Wait(S2);
Wait(S1);
...
Signal(S1);
Signal(S2);
...

```


Sdílená paměť

- Označená část paměti přístupná z různých procesů.
- Služba operačního systému poskytovaná systémovými voláními.

Příklad systémových volání

```

1 /* obtain a shared memory identifier */
2 int shmget(key_t key, size_t size, int flag);
   /* attach shared memory */
5 void* shmat(int shmid, const void *addr, int flag);
   /* detach shared memory */
8 int shmdt(const void *addr);
   /* shared memory control */
11 int shmctl(int shmid, int cmd, struct shmid_ds *buf);

```

- OS spravuje informace o využití sdílené paměti.
- OS také spravuje oprávnění a přístupová práva.

Příklad – Sdílená paměť 1/4 (zápis)

- Zápis načteného řádku z `stdin` do sdílené paměti.

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <stdio.h>
   #define SIZE 512
   int main(int argc, char *argv[])
9 {
10     char *buf;
11     int id;
12     if ((id = shmget(1000, SIZE, IPC_CREAT | 0666)) != -1) {
13         if ( (buf = (char*)shmat(id, 0, 0)) ) {
14             fgets(buf, SIZE, stdin);
15             shmdt(buf);
16         }
17     }
18     return 0;
19 }

```

lec10/shm-write.c

Příklad – Sdílená paměť 2/4 (čtení)

- Načtení řádku ze sdílené paměti a vytisknutí na `stdout`.

```

1 #include <sys/types.h>
2 #include <sys/shm.h>
3 #include <stdio.h>
   #define SIZE 512
   int main(int argc, char *argv[])
8 {
9     int id;
10    char *buf;
11    if ((id = shmget(1000, 512, 0)) != -1) {
12        if ((buf = (char*)shmat(id, 0, 0)) ) {
13            printf("mem:%s\n", buf);
14        }
15        shmdt(buf);
16    } else {
17        fprintf(stderr, "Cannot access to shared memory!\n");
18    }
19    return 0;
20 }

```

lec10/shm-read.c

Příklad – Sdílená paměť 3/4 (ukázka)

1. Použijeme `shm-write` k zápisu textového řetězce do sdílené paměti.
2. Použijeme `shm-read` k načtení dat (řetězce) ze sdílené paměti.
3. Odstranění segmentu sdílené paměti.

`ipcrm -M 1000`

4. Pokus o čtení dat ze sdílené paměti.

```

1 % clang -o shm-write shm-write.c
2 % ./shm-write
3 Hello! I like programming in C!

```

```

1 % clang -o shm-read shm-read.c
2 % ./shm-read
3 mem:Hello! I like programming in C!
4 % ./shm-read
5 mem:Hello! I like programming in C!
6 % ipcrm -M 1000
7 % ./shm-read
8 Cannot access to shared memory!

```

lec10/shm-write.c lec10/shm-read.c

Příklad – Sdílená paměť 4/4 (stav)

- Seznam přístupů do sdílené paměti nástrojem `ipcs`.

```

1 after creating shared memory segment and before writing the text
2 m      65539      1000 --rw-rw-rw-   jf   jf   jf   jf
   1      512      1239      1239 22:18:48 no-entry 22:18:48
4 after writing the text to the shared memory
5 m      65539      1000 --rw-rw-rw-   jf   jf   jf   jf
   0      512      1239      1239 22:18:48 22:19:37 22:18:48
7 after reading the text
8 m      65539      1000 --rw-rw-rw-   jf   jf   jf   jf
   0      512      1239      1260 22:20:07 22:20:07 22:18:48
    
```

Příklad – Předávání zpráv 1/4 (synchronizace, primární)

- Dva procesy jsou synchronizovány námi definovaným protokolem zpráv.
 - Proces `primary` čeká na zprávu od procesu `secondary`.
 - Primární proces informuje sekundární proces o řešení úlohy.
 - Sekundární proces informuje primární proces o řešení.
 - Primární proces odešle zprávu o ukončení.

Příklad Primary process 1/2

```

1 struct msgbuf {
2     long mtype;
3     char mtext[SIZE];
4 };
5 int main(int argc, char *argv[])
6 {
7     struct msgbuf msg;
8     int id = msgget(KEY, IPC_CREAT | 0666);
9     int r;
10    if (id != -1) {
    
```

Zprávy a fronty zpráv

- Procesy mohou komunikovat prostřednictvím zpráv odesílaných/přijímaných do/z front(y) systémových zpráv.
- Fronty jsou entity operačního systému s definovanými systémovými voláními.

Příklad systémových volání

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/msg.h>
4 /* Create a new message queue */
5 int msgget(key_t key, int msgflg);
6 /* Send a message to the queue -- block/non-block (IPC_NOWAIT) */
7 int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
8
9 /* Receive message from the queue -- block/non-block (IPC_NOWAIT) */
10 int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
11
12 /* Control operations (e.g., destroy) the message queue */
13 int msgctl(int msqid, int cmd, struct msqid_ds *buf);
    
```

Jiný systém předávání zpráv může být implementován uživatelskou knihovnou, např. síťovou komunikací.

Příklad – Předávání zpráv 2/4 (primární)

Příklad Primary process 2/2

```

1     msg.mtype = 3; //type must be > 0
2     printf("Wait for other process \n");
3     r = msgrcv(id, &msg, SIZE, 3, 0);
4     printf("Press ENTER to send work\n");
5     getchar();
6     strcpy(msg.mtext, "Do work");
7     msg.mtype = 4; //work msg is type 4
8     r = msgsnd(id, &msg, sizeof(msg.mtext), 0);
9     fprintf(stderr, "msgsnd r: %d\n", r);
10    printf("Wait for receive work results\n", r);
11    msg.mtype = 5;
12    r = msgrcv(id, &msg, sizeof(msg.mtext), 5, 0);
13    printf("Received message: %s\n", msg.mtext);
14    printf("Press ENTER to send exit msg\n");
15    getchar();
16    msg.mtype = EXIT_MSG; //I choose type 10 as exit msg
17    r = msgsnd(id, &msg, 0, 0);
18 }
19 return 0;
20 }
    
```

lec10/msg-primary.c

Příklad – Předávání zpráv 3/4 (sekundární)

```

1 int main(int argc, char *argv[])
2 {
3     ...
4     msg.mtype = 3;
5     printf("Inform main process\n");
6     strcpy(msg.mtext, "I'm here, ready to work");
7     r = msgsnd(id, &msg, sizeof(msg.mtext), 0);
8     printf("Wait for work\n");
9     r = msgrcv(id, &msg, sizeof(msg.mtext), 4, 0);
10    printf("Received message: %s\n", msg.mtext);
11    for (i = 0; i < 4; i++) {
12        sleep(1);
13        printf(".");
14        fflush(stdout);
15    } //do something useful
16    printf("Work done, send wait for exit\n");
17    strcpy(msg.mtext, "Work done, wait for exit");
18    msg.mtype = 5;
19    r = msgsnd(id, &msg, sizeof(msg.mtext), 0);
20    msg.mtype = 10;
21    printf("Wait for exit msg\n");
22    r = msgrcv(id, &msg, SIZE, EXIT_MSG, 0);
23    printf("Exit message has been received\n");

```

lec10/msg-secondary.c

Komunikace standardním vstupem/výstupem

- Z aplikace můžeme spouštět jiné procesy.
- U spuštěného procesu přesměrujeme standardní vstupy a výstupy (*stdin*, *stdout*, *stderr*).
- Proces běží:
 - V rámci vykonávání naší aplikace.
 - Jako nový paralelní proces, obdoba volání `fork()`.
- Tento mechanismus umožňuje propojování kódu na úrovni programů (binárních).
- Binární kód může být vytvořen v jiném prog. jazyce.

Znovu použitelnost na úrovni samostatných programů je jednou z hlavních motivačních oddělení jádra (výpočetního) programu a grafického rozhraní.

Příklad – Předávání zpráv 4/4 (ukázka)

1. Spustit primární proces.
2. Spustit sekundární proces.
3. Provedení výpočtu.
4. Odstranění vytvořené fronty zpráv `msgid`.

```

                                #define KEY 1000
                                ipcrm -Q 1000
1 % clang msg-primary.c -o primary
2 % ./primary
3 Wait for other process
4 Worker msg received, press ENTER to send work msg
5 msgsnd r: 0
6
7 Wait for receive work results
8 Received message: I am going to wait for exit msg
9 Press ENTER to send exit msg
11 %ipcrm -Q 1000
12 %ipcrm -Q 1000
13 ipcrm: msqs(1000): : No such file or directory
14 %

```

```

1 % clang msg-secondary.c -o secondary
2 % ./secondary
3 Inform main process
4 Wait for work
5 Received message:Do work
6 ...done
7 Work done, send wait for exit
8 Wait for exit msg
9 Exit message has been received
10 %ipcs -q
11 Message Queues:
12 T ID KEY MODE OWNER GROUP
13 q 65536 1000 -rw-rw- jf jf
14 %
                                lec10/msg-primary.c lec10/msg-secondary.c

```

Shrnutí přednášky

Diskutovaná témata

- Úvod do paralelního programování
 - Principy a hlavní architektury
 - Program a proces v OS
- Paralelní výpočet
- Synchronizace a mezi-procesová komunikace (Inter-Process Communication – IPC)
 - Semaforey
 - Sdílená paměť
 - Zprávy
- **Příště: Vícevláknové programování**

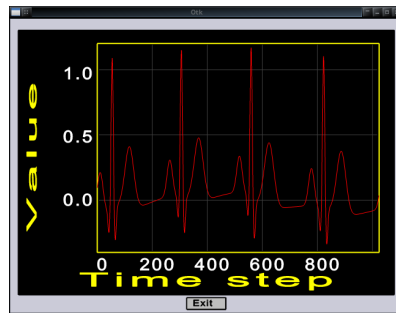
Generátor signálů a vizualizace

- Mějme generátor signálů `sngen`, což je program, který generuje posloupnost hodnot na `stdout`.
- Vizualizace může být realizována v jiné aplikaci `tsignal_viewer`, která čte hodnoty ze `stdin`.
- Tyto dvě aplikace můžeme propojit rourou `./sngen | ./tsignal_viewer`.

```
$ ./sngen
0.000000
1.075095
1.031029
0.916540
0.748307
0.549199
0.342897
0.149521
-0.016643
-0.147374
-0.239553
-0.293963
-0.314364
-0.306833
-0.279048
-0.239311
-0.195373
```

```
$ ./sngen | wc
65772 65772 618529
$ ./sngen | ./tsignal_viewer
```

- HW9B je rozšířením konceptu uživatelem definovaného komunikačního protokolu mezi generátorem signálů a řídicí aplikací s vizualizací a **vícevláknovou aplikací**.



Část III Appendix

Masivní paralelismus s využitím grafických karet

- Vykreslování obrázků prováděné pixel po pixelu lze relativně <F10>snadno paralelizovat.
- Grafické procesory (GPU) mají podobný (nebo dokonce vyšší) stupeň integrace s hlavními procesory (CPU).
- Mají obrovský počet paralelních procesorů.
- Výpočetní výkon lze využít i v jiných aplikacích.
 - Zpracování proudu dat (instrukce SIMD - procesory).
 - GPGPU - výpočet pro všeobecné účely na GPU.
 - OpenCL (Open Computing Language) – abstraktní rozhraní GPGPU.
 - CUDA - Paralelní programovací rozhraní pro grafické karty Nvidia.

Například GeForce GTX 1060 ~ 1280 jader.

<http://www.gpgpu.org>
http://www.nvidia.com/object/cuda_home.html

Výpočetní výkon (2008)

- Jaký je uváděný výpočetní výkon procesoru?
- Grafické (proudové/stream) procesory.

CSX700	96 GigaFLOPs	
Cell	102 GigaFLOPs	
GeForce 8800 GTX	518 GigaFLOPs	(včetně texturovacích jednotek)
Radeon HD 4670	480 GigaFLOPs	
GeForce RTX 4060	15 110 GigaFLOPs	(2023) Špičkové katalogové hodnoty.

Procesory

Phenom X4 9950 (@2,6 GHz)	21 GigaFLOPs
Core 2 Duo E8600 (@3,3 GHz) a 22 GigaFLOPs	
Cure 2 Quad QX9650 (@3,3 GHz)	35 GigaFLOPs
Cure 2 Quad QX9650 (@3,3 GHz) a 35 GigaFLOPs	
Core i7 970 (@3,2 GHz) a 42 GigaFLOPs	
Core i9-13900 (@2,00–5,60 GHz)	846 GigaFLOPs (2023)

Test linpack 32-bit.

- Je uváděný výkon skutečně dosažitelný?
- Jaké jsou další charakteristiky?
 - CSX700 má typickou spotřebu energie kolem 9W.

(float vs double)
Např. výpočetní výkon / spotřeba energie.

CUDA – výpočetní model

- Jádro (výpočet) je rozděleno do bloků.
- Každý blok představuje paralelní výpočet části výsledku.
Např. část násobení matic.
- Každý blok se skládá z výpočetních vláken.
- Paralelní výpočty jsou synchronizovány v rámci bloku.
- Bloky jsou uspořádány do **mřížky**.
- Škálovatelnost je realizována rozdělením výpočtu do bloků.

Bloky nemusí být nutně počítány paralelně. Podle dostupného počtu paralelních jednotek mohou být určité bloky počítány postupně.

CUDA

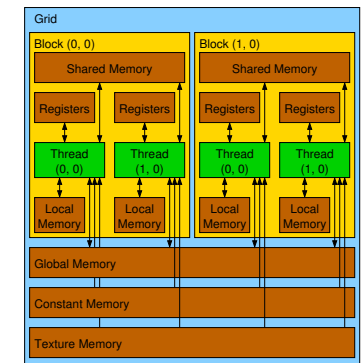
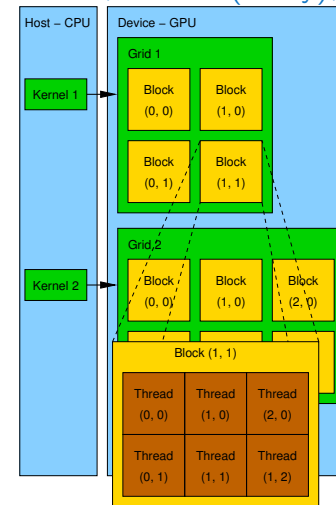
- NVIDIA Compute Unified Device Architecture.
- Rozšíření jazyka C pro přístup k paralelním výpočetním jednotkám GPU.
- Výpočet (**jádro**/keykernel) provádí GPU.
- Jádro se provádí paralelně s využitím dostupných výpočetních jednotek.
- **Host** - Hlavní procesor (proces).
- **Device** - GPU.
- Data musí být v paměti přístupné GPU.

Host paměť → Device paměť

- Výsledek (výpočtu) je uložen v paměti GPU.

Host memory ← Device memory

CUDA – Grid, Blocks (bloky), Threads (vlákna) a přístup do paměti



- Doba přístupu do paměti.
- Kolize při současném přístupu několika vláken.

CUDA – Příklad – Násobení matic 1/8

- NVIDIA CUDA SDK - verze 2.0, matrixMul.
- Jednoduché násobení matic.
 - $C = A \cdot B$,
 - Matice mají stejné rozměry $n \times n$,
 - kde n je násobek velikosti bloku.
- Porovnání
 - naivní implementace v jazyce C ($3 \times$ for cyklus),
 - naivní implementace v C s transpozicí matice.
 - CUDA implementace.
- Hardware
 - CPU - Intel Core 2 Duo @ 3 GHz, 4 GB RAM,
 - GPU - NVIDIA G84 (GeForce 8600 GT), 512 MB RAM.

CUDA – Příklad – Násobení matic 3/8

Naivní implementace s transpozicí

```

1 void simple_multiply_trans(const int n, const float *a, const float *b, float *c)
2 {
3     float * bT = create_matrix(n);
4     for (int i = 0; i < n; ++i) {
5         bT[i*n + i] = b[i*n + i];
6         for (int j = i + 1; j < n; ++j) {
7             bT[i*n + j] = b[j*n + i];
8             bT[j*n + i] = b[i*n + j];
9         }
10    }
11    for (int i = 0; i < n; ++i) {
12        for (int j = 0; j < n; ++j) {
13            float tmp = 0;
14            for (int k = 0; k < n; ++k) {
15                tmp += a[i*n + k] * bT[j*n + k];
16            }
17            c[i*n + j] = tmp;
18        }
19    }
20    free(bT);
21 }

```

CUDA – Příklad – Násobení matic 2/8

Naivní implementace

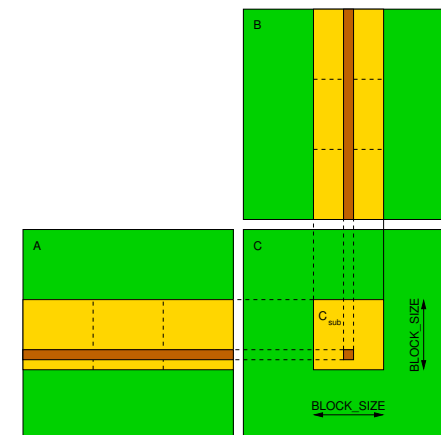
```

1 void simple_multiply(const int n,
2     const float *A, const float *B, float *C)
3 {
4     for (int i = 0; i < n; ++i) {
5         for (int j = 0; j < n; ++j) {
6             float prod = 0;
7             for (int k = 0; k < n; ++k) {
8                 prod += A[i * n + k] * B[k * n + j];
9             }
10            C[i * n + j] = prod;
11        }
12    }
13 }

```

CUDA – Příklad – Násobení matic 4/8

- CUDA – výpočetní strategie
 - Rozdělíme matice do bloků.
 - Každý blok vypočítá jednu dílčí matici C_{sub} .
 - Každé vlákno jednotlivých bloků vypočítá jeden prvek C_{sub} .



CUDA – Příklad – Násobení matic 5/8

CUDA – Implementace – hlavní funkce

```

1 void cuda_multiply(const int n, const float *hostA, const float *hostB, float *hostC)
2 {
3     const int size = n * n * sizeof(float);
4     float *devA, *devB, *devC;
5     cudaMalloc((void**)&devA, size);
6     cudaMalloc((void**)&devB, size);
7     cudaMalloc((void**)&devC, size);
8     cudaMemcpy(devA, hostA, size, cudaMemcpyHostToDevice);
9     cudaMemcpy(devB, hostB, size, cudaMemcpyHostToDevice);
10    dim3 threads(BLOCK_SIZE, BLOCK_SIZE); // BLOCK_SIZE == 16
11    dim3 grid(n / threads.x, n / threads.y);
12    // Call kernel function matrixMul
13    matrixMul<<<grid, threads>>>(n, devA, devB, devC);
14    cudaMemcpy(hostC, devC, size, cudaMemcpyDeviceToHost);
15    cudaFree(devA);
16    cudaFree(devB);
17    cudaFree(devC);
18 }

```

CUDA – Příklad – Násobení matic 7/8

- CUDA zdrojové kódy.

Příklad – Vyhrazený zdrojový soubor cuda_func.cu

1. Deklarace externí funkce.

```

1 extern "C" { // declaration of the external function (cuda kernel)
2     void cuda_multiply(const int n, const float *A, const float *B, float *C);
3 }

```

2. Zkompilujeme kód CUDA do kódu C++.

```
1 nvcc --cuda cuda_func.cu -o cuda_func.cc
```

3. Kompilace souboru `cuda_func.cu.cc` pomocí standardního kompilátoru.

CUDA – Příklad – Násobení matic 6/8

Implementace CUDA – kernel funkce

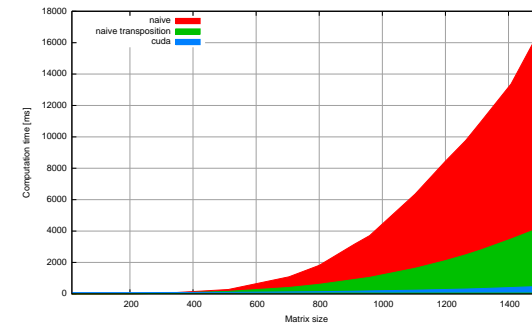
```

1 __global__ void matrixMul(int n, float* A, float* B, float* C) {
2     int bx = blockIdx.x; int by = blockIdx.y;
3     int tx = threadIdx.x; int ty = threadIdx.y;
4     int aBegin = n * BLOCK_SIZE * by; //beginning of sub-matrix in the block
5     int aEnd = aBegin + n - 1; //end of sub-matrix in the block
6     float Csub = 0;
7     for (
8         int a = aBegin, b = BLOCK_SIZE * bx;
9         a <= aEnd;
10        a += BLOCK_SIZE, b += BLOCK_SIZE * n
11    ) {
12        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE]; // shared memory within
13        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE]; // the block
14        As[ty][tx] = A[a + n * ty + tx]; // each thread reads a single element
15        Bs[ty][tx] = B[b + n * ty + tx]; // of the matrix to the memory
16        __syncthreads(); // synchronization, sub-matrix in the shared memory
17        for (int k = 0; k < BLOCK_SIZE; ++k) { // each thread computes
18            Csub += As[ty][k] * Bs[k][tx]; // the element in the sub-matrix
19        }
20        __syncthreads();
21    }
22    int c = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
23    C[c + n * ty + tx] = Csub; // write the results to memory
24 }

```

CUDA – Příklad – Násobení matic 8/8

Výpočetní čas (v milisekundách)



N	Naive	Transp.	CUDA
112	2	1	81
208	11	11	82
304	35	33	84
704	1083	405	122
1104	6360	1628	235
1264	9763	2485	308

- Matlab 7.6.0 (R2008a):
n=1104; A=rand(n,n); B=rand(n,n); tic; C=A*B; toc
Uplynulý čas je 0,224183 sekundy.