

BRNO UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF CONTROL AND INSTRUMENTATION

Kolejní 4, 616 00 Brno

tel.: +420 5 4114 1113 fax: +420 5 4114 1123

E-mail: kucera@feec.vutbr.cz <http://taceo.eu>



Operační Systémy Reálného Času

II.

Pavel Kučera

CAK, E112

kucera@feec.vutbr.cz



Obsah

1. Víceprocesorové architektury
2. Plánovací algoritmy
3. Správa paměti
4. Race Condition
5. Synchronizace
6. Spinlock
7. Mutex
8. Events
9. Deadlock
10. Reentrantní funkce
11. Inverze priorit



Scheduling (plánování)

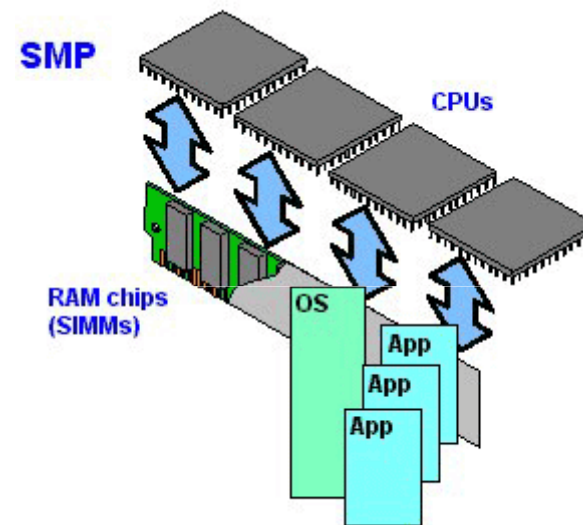
Víceúlohové OS vyžadují paralelní běh několika úloh. Paralelního běhu lze v zásadě docílit dvěma způsoby:

- na jednoprocesorovém systému pseudokonkurencí,
- na víceprocesorovém systému skutečným paralelním během (pravá konkurence). Víceprocesorové architektury se pak dle organizace CPU a paměti dělí na:
 - Symetric multiprocessing (SMP)
 - Asymetric multiprocessing (ASMP)
 - Non-Uniform Memory Access (NUMA)



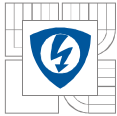
Architektura SMP

- Nutná podpora v OS, popřípadě v compileru
- Zrychlují běh i procesů navržených pro uniproceorové platformy
- Rychlá výměna dat mezi procesy
- Pomalejší přístup do dedikované



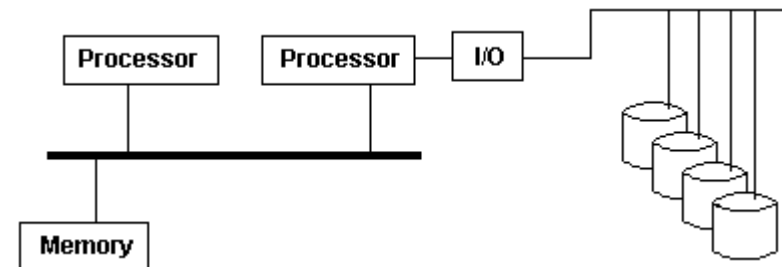
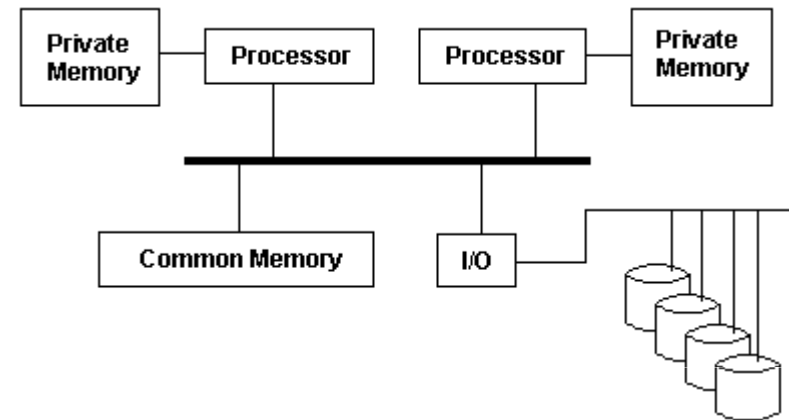
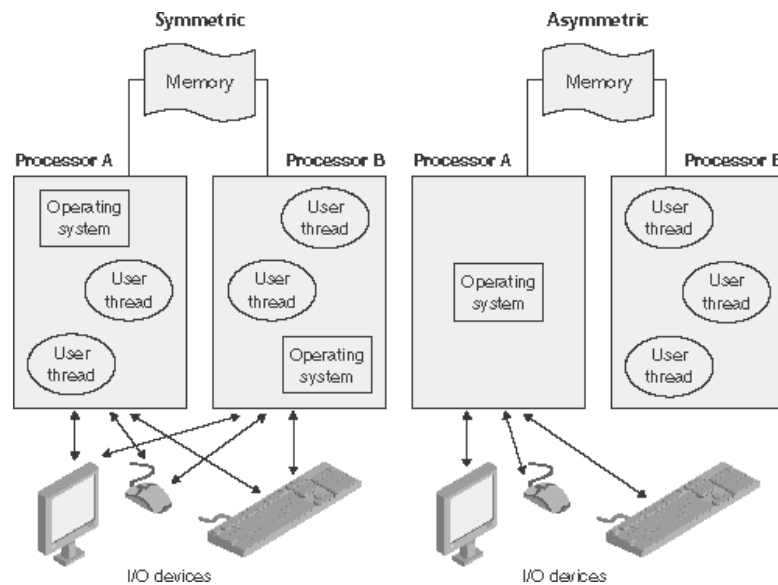
Zdroj: www.zdnet.co.kr

QNX
Mac OS
OS/2
VxWorks
Windows NT (2k, XP, 2003)
Unix
Linux
Solaris



Architektura ASMP

- Řešení je často na uživateli, popřípadě je šité na míru
- Pomalá výměna dat mezi procesy
- Rychlý přístup do dedikované paměti



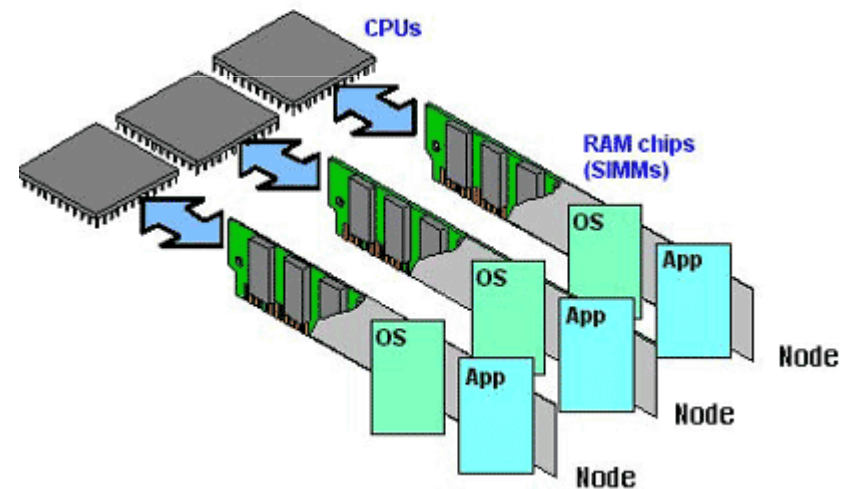
Zdroj: ohlandl.ipv7.net



Architektura NUMA

- Speciální (ale častý) případ ASMP
- Extrémně rychlý přístup do přidělených pamětí (často i fyzicky)
- Problémy se synchronizací a sdílenými daty – pomalý přenos dat mezi procesy

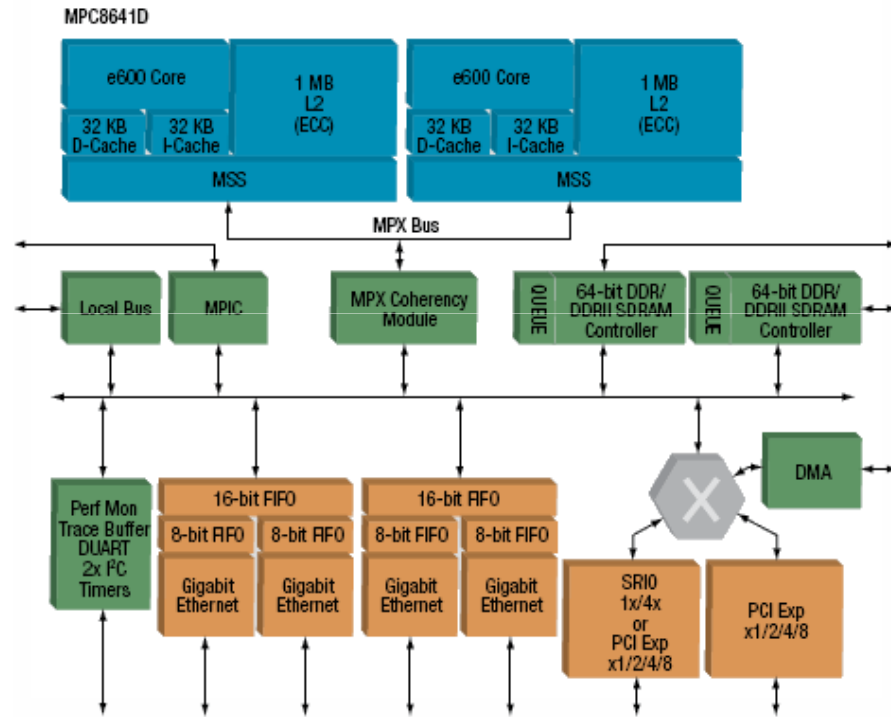
Intel Itanium
AMD Opetron
SGI
DEC
Alpha EV7
NUMA for Linux
NUMA for Windows
OpenSolaris NUMA



Zdroj: www.zdnet.co.kr

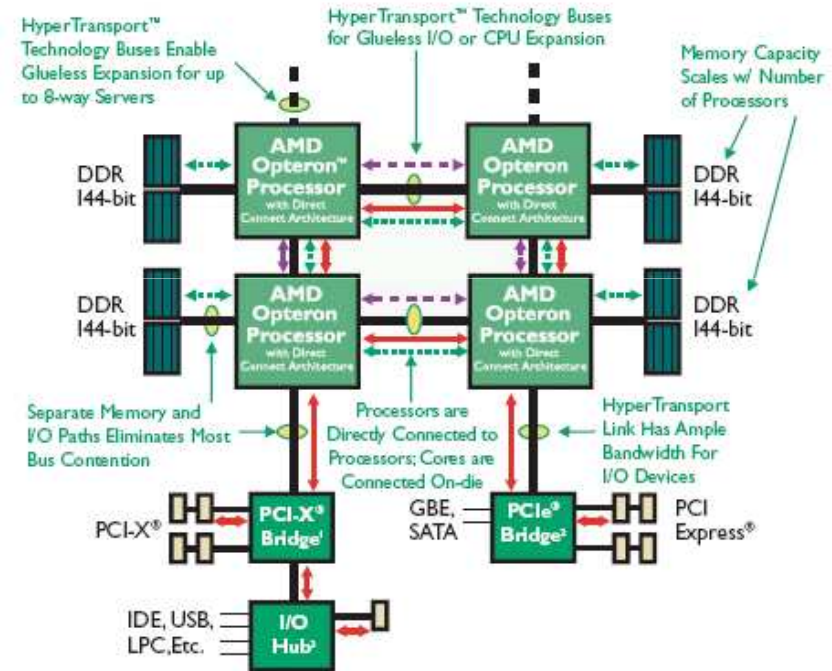


SMP vs NUMA



Zdroj: www.rtc magazine.com

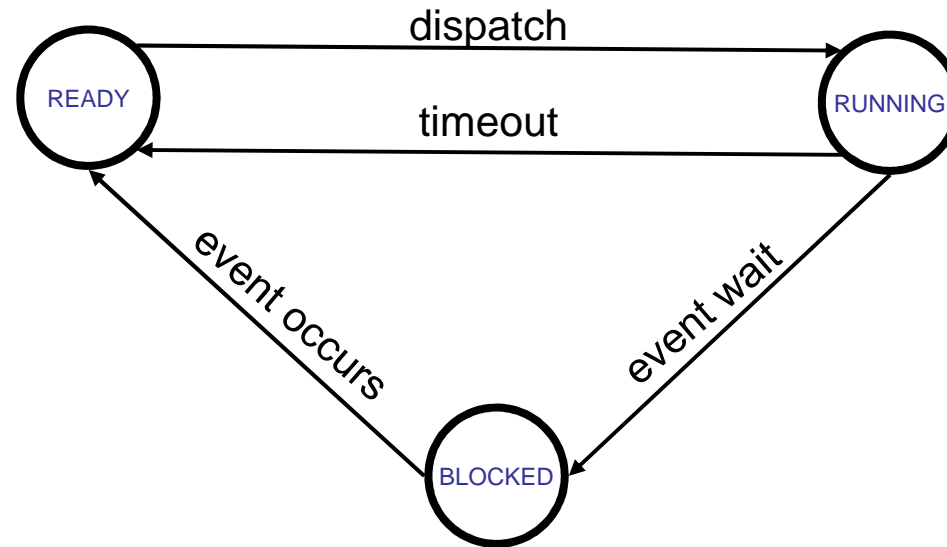
AMD OPTERON



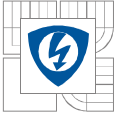
Zdroj: developer.amd.com



Stavový automat úlohy

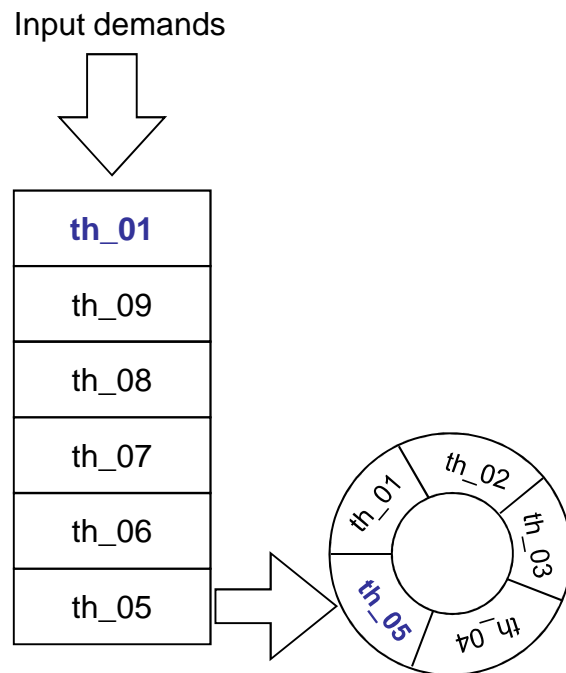


- READY** úloha je připravena ke spuštění
- RUNNING** úloha je vykonávána
- BLOCKED** úloha se nevykonává, protože čeká na událost



FIFO - FCFS

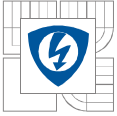
FCFS First Come, First Served



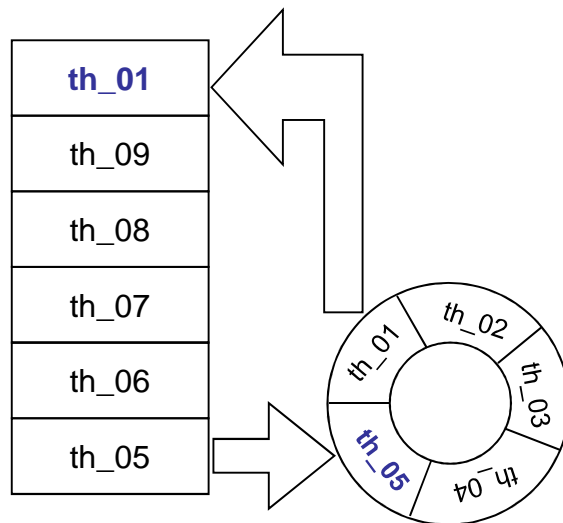
Obvykle implementováno jako nezávislá množina obsluh přerušení.

Klade nízké nebo vůbec žádné požadavky na plánovač jádra.

To však neúměrně stěžuje návrh systému.



Cyclic executive



Velmi podobný přístupu “supersmyčka“

Jednoduchá implementace i ověření

Složité časování při modifikaci úlohy

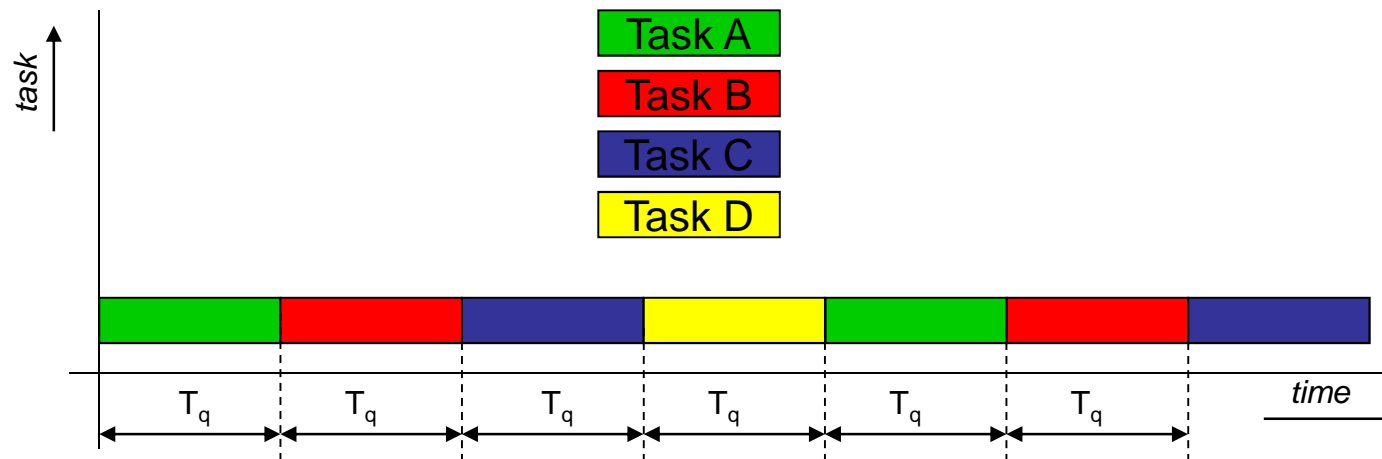
Kvůli jednoduchosti používáno v safety-critical a fail-safe aplikacích (NASA)

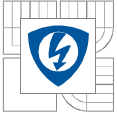


Round Robin (time-slicing)

Všechny úlohy mají k CPU přístup po stejné časové kvantum T_q .

Nevýhodou je, že úloha s velkou prioritou, která potřebuje CPU se k němu dostane až za dlouhý interval.

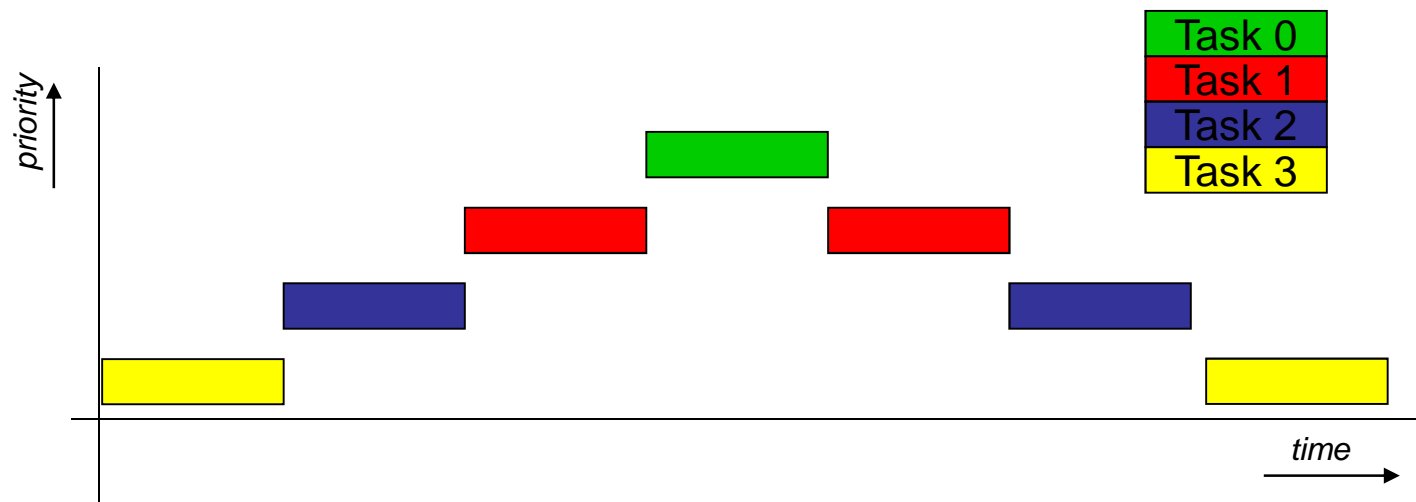




Fixed Priority Scheduling

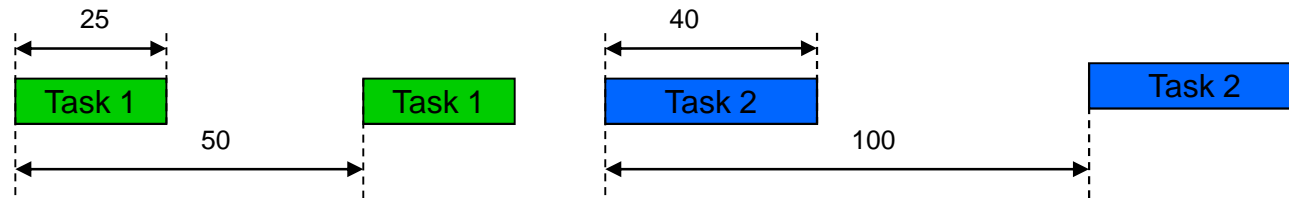
Každá úloha má nastavenou prioritu (celé číslo, **obvykle** nižší hodnota značí vyšší prioritu).

Pokud dvě úlohy mají stejnou prioritu, tak ta která přijde dřív dostane CPU a nebude přerušena.

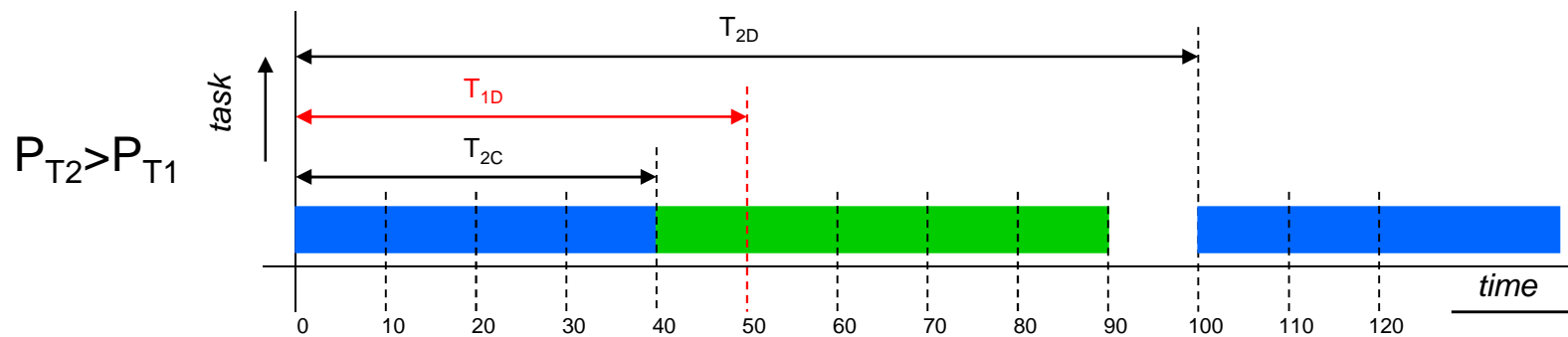
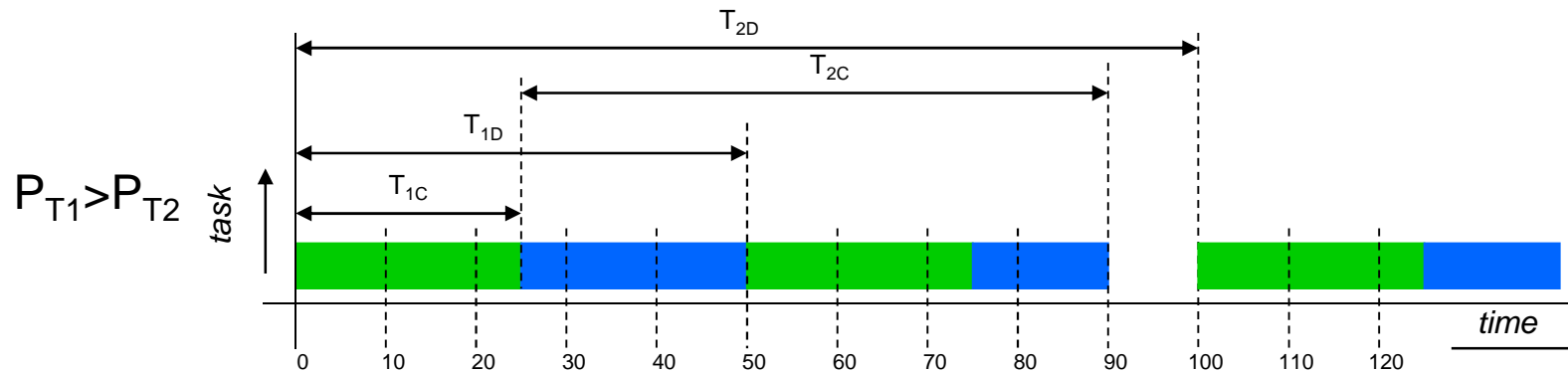




Fixed Priority Scheduling Problem



$$U = \sum_{i=1}^n \frac{C_i}{T_i}$$





Rate Monothonic Algorithm RMA

Algoritmus, který umožní nastavit prioritu úlohám s ohledem na splnění jejich deadlines.

Každé úloze je přiřazena priorita dle periody jejího vykonávání. Kratší perioda->vyšší priorita.

RMA je optimálním statickým plánovacím algoritmem; selže-li, pak selže jakýkoliv jiný statický plánovací algoritmus.

$$U \leq n(\sqrt[n]{2} - 1)$$

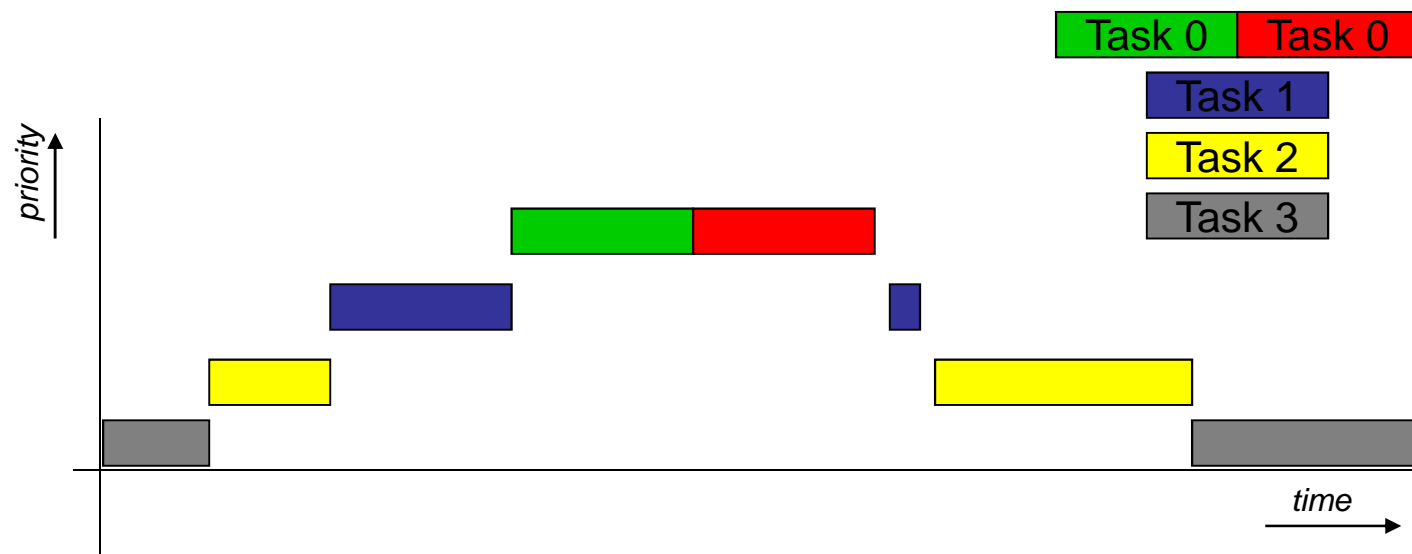
Pro $n \rightarrow \infty$ je $U \leq 69,3\%$. Zbývajících 30,7% lze tedy využít pro ostatní (non-realtime) úlohy.

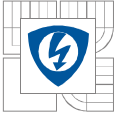
Nevýhodou je, že často periody a délky exekuce neznáme.



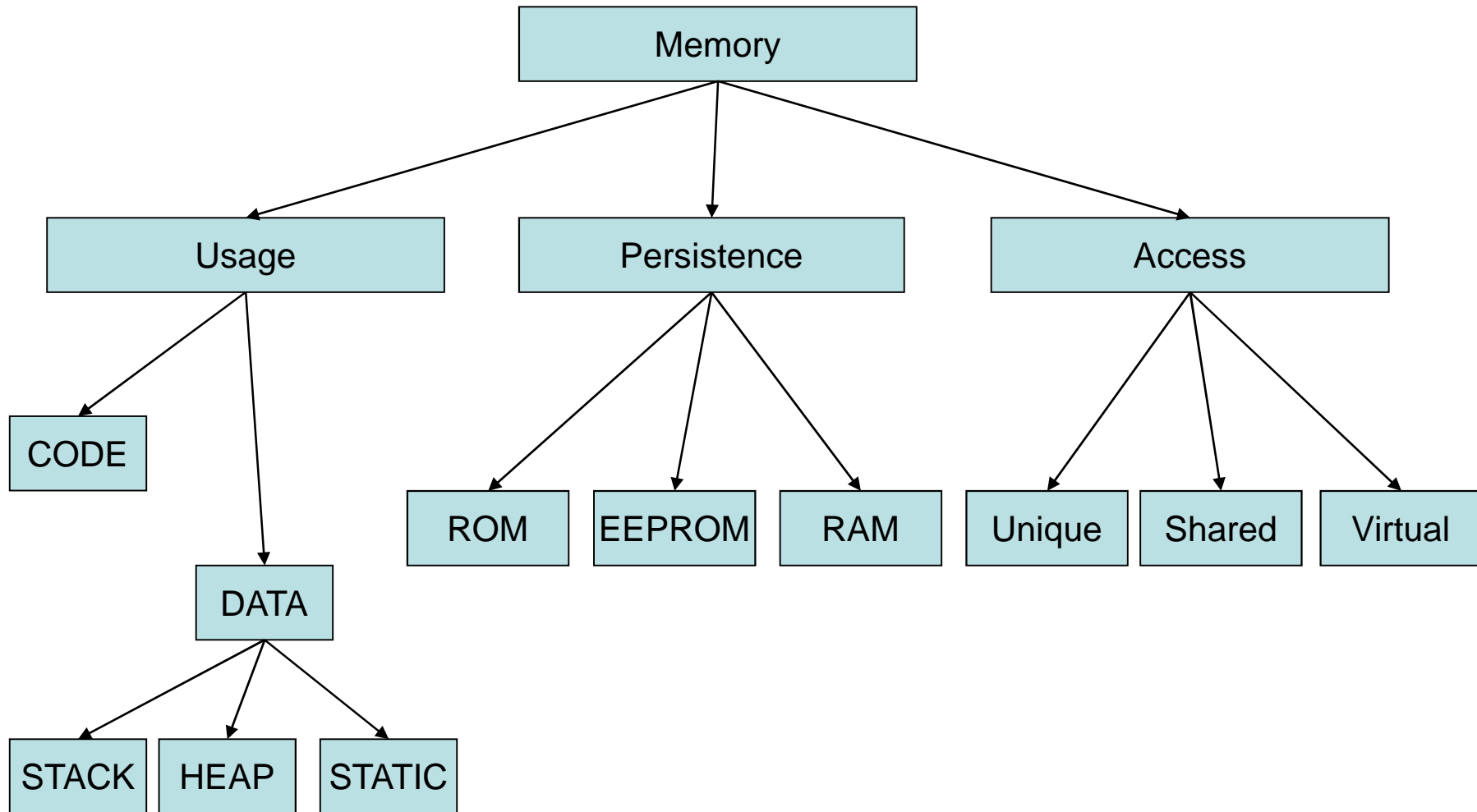
Fixed-Priority Preemptive Round-Robin Scheduling

Pokud je naplánováno (ve stavu READY) více úloh se stejnou prioritou, pak se tyto plánují podle round-robin algoritmu.





Správa paměti v RTOS





Alokace paměti

- Stack** real-time odezva, nutno dát pozor na přetečení (LIFO struktura), **znalost překladače, reentrantnost funkcí**
- Static** není dostatek, často kvůli segmentaci, real-time odezva
- Heap** alokace/dealokace, problém s real-time odezvou

Strategie statické alokace je dobře známá a osvědčená (není problémem v RTOS).

Při dynamické alokaci vzniká problém s velikostí dostupné paměti a deterministickou odezvou při žádosti o paměť. Řešení:

- dynamická alokace paměti při startu (boot phase) – extrémní plýtvání, nemožné v malých RTOS
- dynamická alokace v haldě – problém s fragmentací a real-time odezvou



Heap

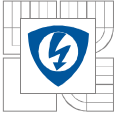
Během chodu aplikace dochází ke ztrátě paměti (v haldě je alokován blok, k němuž neexistuje z aplikace pointer – paměť zůstane neuvolněná). Typickým příkladem jsou smart-pointery v exception handling.

Alokace je často prováděna po blocích (pool). Bloky by měly mít takovou velikost, aby uspokojily libovolný požadavek o paměť (plýtvání).

Některé RTOS nabízí různé haldy s různě velkými memory-pool.

Částečné řešení je pomocí *garbage collector* mechanismu. V RTOS však velmi problematické. Nutno použít zdvojených pointerů, což zvyšuje výpočetní náročnost jak aplikace, tak čištění paměti.

Mnoho RTOS pak paměť v haldě nenabízí vůbec.



Race Condition

```
int sum = 0;

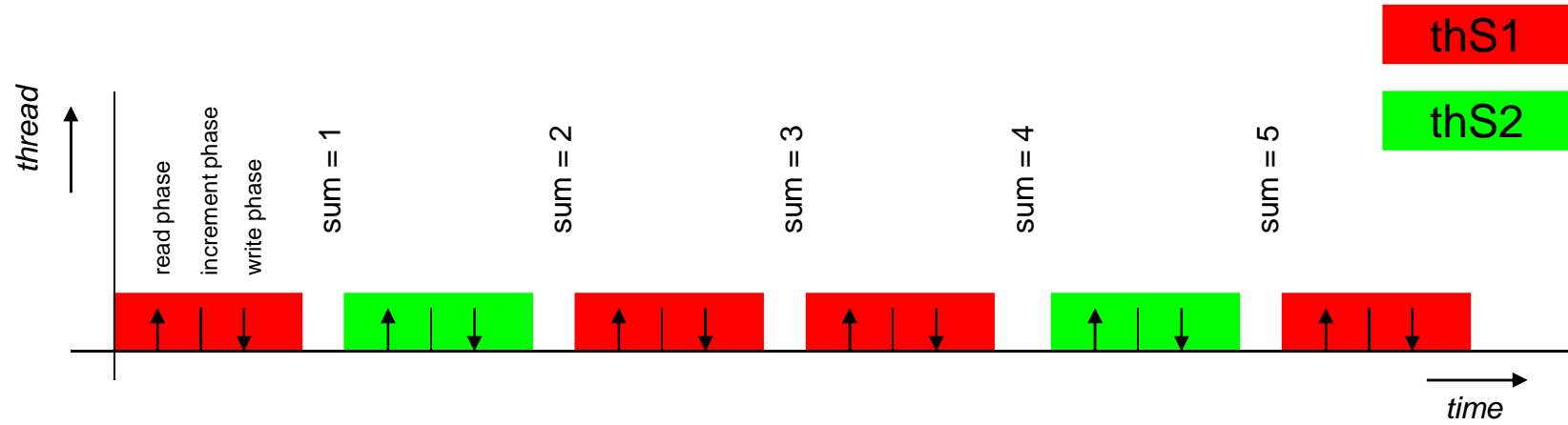
ULONG RTFCNDCL thS1( void *nContext ) {
int is;
while(TRUE);
    is = sum;
    is ++;
    sum = is;
}
return 0;
}

ULONG RTFCNDCL thS2( void *nContext ) {
int is;
while(TRUE);
    is = sum;
    is ++;
    sum = is;
}
return 0;
}
```

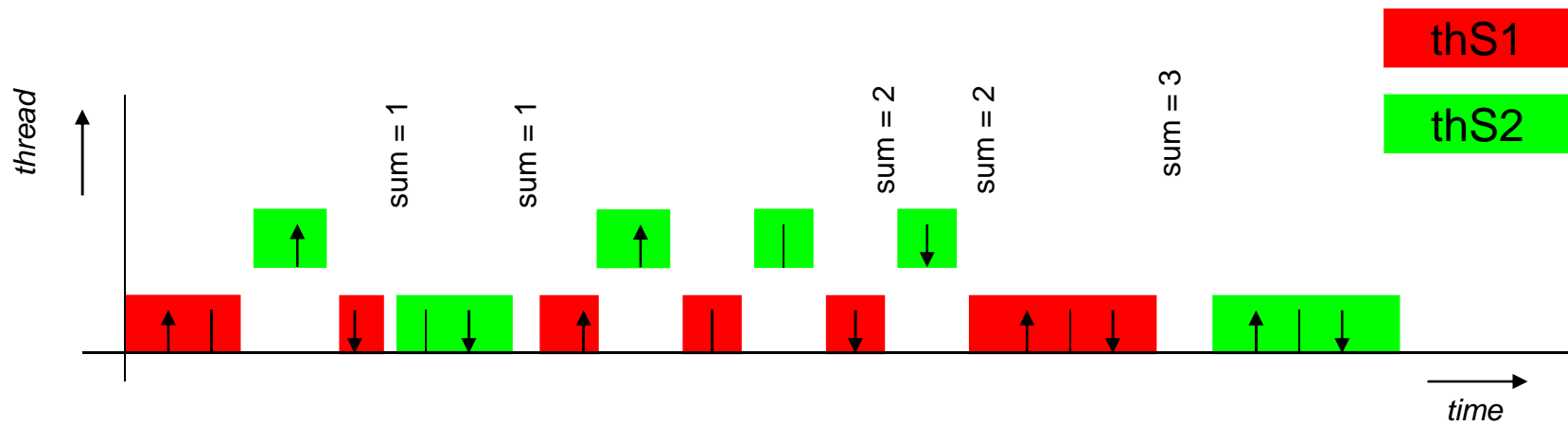


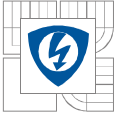
Race Condition

Scenario 1



Scenario 2





Race Condition

```
int sum = 0;

ULONG RTFCNDCL thS1( void *nContext ) {
int is;
while(TRUE);
    is = sum;
    is ++;
    sum = is;
}
return 0;
}

ULONG RTFCNDCL thS2( void *nContext ) {
int is;
while(TRUE);
    is = sum;
    is ++;
    sum = is;
}
return 0;
}
```



Synchronizace mezi procesy

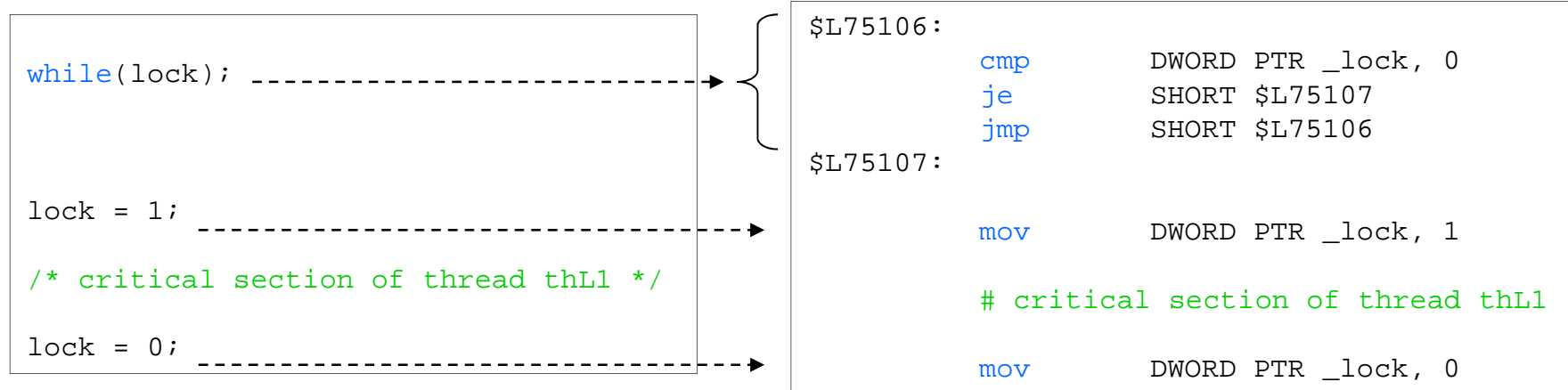
```
int lock = 0;

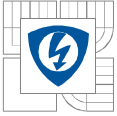
ULONG RTFCNDCL thL1( void *nContext ) {
    while(lock);
    lock = 1;
    /* critical section of thread thL1 */
    lock = 0;
    return 0;
}

ULONG RTFCNDCL thL2( void *nContext ) {
    while(lock);
    lock = 1;
    /* critical section of thread thL2 */
    lock = 0;
    return 0;
}
```



Synchronizace mezi procesy





Synchronizace mezi procesy

```
int lock = 0;

ULONG RTFCNDCL thL1( void *nContext ) {
    while(lock);
    lock = 1;
    /* critical section of thread thL1 */
    lock = 0;
    return 0;
}

ULONG RTFCNDCL thL2( void *nContext ) {
    while(lock);
    lock = 1;
    /* critical section of thread thL2 */
    lock = 0;
    return 0;
}
```



Kritická sekce

Úsek kódu, který přistupuje ke sdílenému zdroji a který nesmí být v jednom okamžiku využíván více než jedním vláknem.

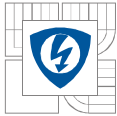
Je tedy nutné zabránit procesoru opuštění kritické sekce, jakmile do ní jednou vstoupí.

- zakázat přerušení
- použít semaforey
- použít mutexy
- použít zámky



Spinlocks

- Atomická struktura umožňující synchronizaci mezi procesy.
- V podstatě nelze realizovat ve vyšších programovacích jazycích.
- Ve strojovém kódu lze realizovat při použití atomických instrukcí (tj. takových, které nemohou být přerušeny) typu test-and-set, fetch-and-add, compare-and-swap.
- Nelze v jednoúlohových RTOS.
- Jednoduchá implementace, časté použití v kernelech a driverech.



Spinlocks

```
lock:                # The lock variable. 1 = locked, 0 = unlocked.
    dd                0

spin_lock:
    mov               eax, 1                # Set the EAX register to 1.

loop:
    xchg              eax, [lock]          # Atomically swap the EAX register with
                                           # the lock variable.
                                           # This will always store 1 to the lock, leaving
                                           # previous value in the EAX register.

    test              eax, eax             # Test EAX with itself. Among other things, this will
                                           # set the processor's Zero Flag if EAX is 0.
                                           # If EAX is 0, then the lock was unlocked and
                                           # we just locked it.
                                           # Otherwise, EAX is 1 and we didn't acquire the lock.

    jnz               loop                 # Jump back to the XCHG instruction if the Zero Flag is
                                           # not set, the lock was locked, and we need to spin.

    ret

spin_unlock:
    mov               eax, 0                # Set the EAX register to 0.

    xchg              eax, [lock]          # Atomically swap the EAX register with
                                           # the lock variable.

    ret                                  # The lock has been released.
```



Mutex (mutual exclusion)

Mutual exclusion – v jednom okamžiku zdroj patří pouze jednomu tasku.

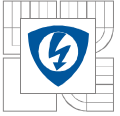
Pokud task čeká na zdroj chráněný mutexem, čeká tak dlouho, dokud zdroj nebude uvolněn. **Ovšem jen pokud použijeme k přístupu mutex!**

Pokud je task, který mutex vlastní, předčasně ukončen (čímž sdílený zdroj řádně neuvolní), jsou o tom ostatní úlohy v procesu informovány (WAIT_ABANDONED); stav sdíleného prostředku je pak nekonzistentní!

Pokud na vlastnění mutexu čeká více tasků, získá ho ten s nejvyšší prioritou. Pokud mají tasky stejnou prioritu, získá mutex ten task, který požádal o mutex nejdříve (tj. čeká nejdéle), t.j. round-robin policy.

Problémy s mutexy:

- Vytváření mutexu a okamžité vlastnění mutexu (rat race).
- Mutex není uvolněn a task skončil -> deadlock.
- Blokují části kódu (režije).



Mutex - Dekker's algorithm

```
bool flag_0 = false; //proces 0 chce vstoupit do kritické sekce
bool flag_1 = false; //proces 1 chce vstoupit do kritické sekce
int turn = 0;        //id procesu
//int turn = 1;
```

process_0

```
flag_0 = true;
while(flag_1) {
    if(turn != 0) {
        flag_0 = false;
        while(turn != 0) {
        }
        flag_0 = true;
    }
}
// critical section
turn = 1;
flag_0 = false;
```

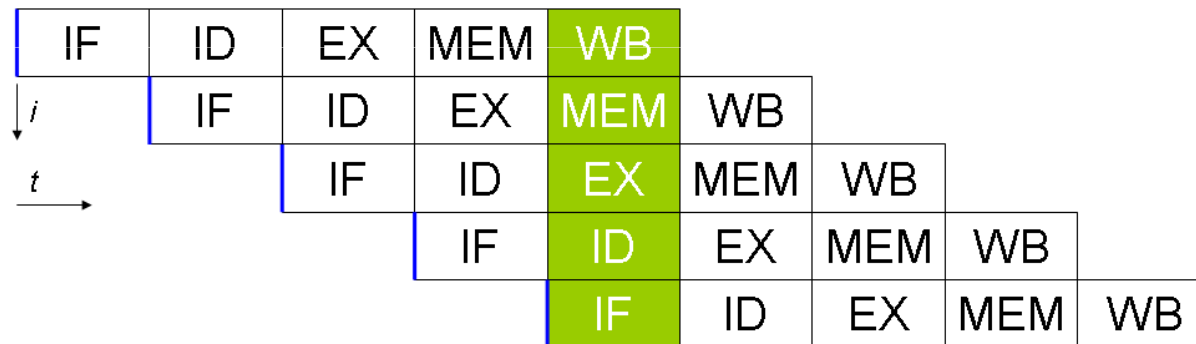
process_1

```
flag_1 = true;
while(flag_0) {
    if(turn != 1) {
        flag_1 = false;
        while(turn != 1) {
        }
        flag_1 = true;
    }
}
// critical section
turn = 0;
flag_1 = false;
```



Mutex – Problémy

- out-of-order řetězení
- pipelining



zdroj: en.wikipedia.org



Event (událost)

Události jsou synchronizačním prostředkem mezi tasky IPC.
Neblokují části kódu jako mutexy – jsou bodové.

RTOS obvykle poskytují operace pro vytvoření události, čekání na událost a signalizace události.

RTOS obvykle umožňují nastavit event jako manual-reset nebo auto-reset a pulse nebo set.

Problémy s eventy:

- Správný výběr eventu (manual-reset/auto-reset). První uvolněný task nastaví při auto-reset event na non-signaled).
- Správná signalizace eventu (pulse/set). Pulse nastaví stav event do signaled, uvolní všechny čekající tasky a poté se sám zruší.



Semafor

Semafore jsou počítané události.

Počítadlo semaforu je nastaveno při jeho vytvoření na určitou úroveň a je sníženo o 1 pokaždé, když je task uvolněn z čekání a poté zvýšeno o definovaný počet při uvolnění semaforu.

Pokud počítadlo semaforu dosáhne hodnoty 0, pak všechny tasky čekají na jeho zvýšení (typické využití při inicializaci aplikace).

Problémy se semaforey:

Čekání na semafor nikdy nevrátí `WAIT_ABANDONED` (na rozdíl od mutexu), takže násilné ukončení tasku, který neuvolnil semafor vede ke ztrátě informace o tom, kolikrát byl daný prostředek použit.

Čeká-li více tasků na semafor, budou volány v pořadí odpovídajícím jejich prioritám. V případě shodných priorit se používá round-robin algoritmus.



Deadlock (uváznutí)

Úloha čeká na podmínku, která **nemůže** být splněna.

V FSA je to stav, který nemá odchozí přechod nebo tento nemůže být nikdy splněn.

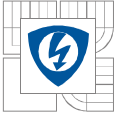
V RTOS systémech kritický a často velmi nebezpečný jev.

Vzniká špatným návrhem systému, špatným použitím synchronizačních primitiv, popřípadě HW chybou.

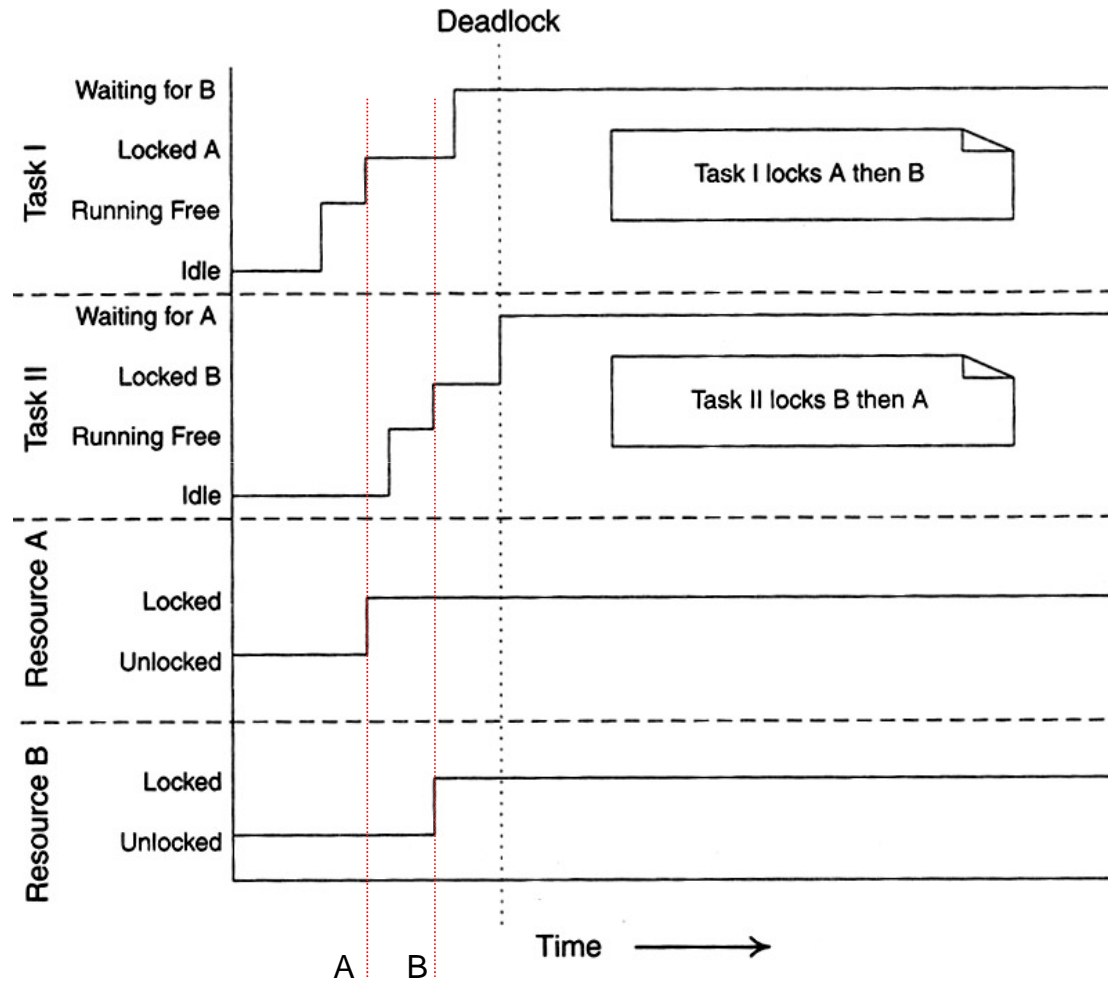
Aby deadlock vznikl musí být splněny všechny tyto podmínky:

1. Úloha během svého chodu musí vyžadovat exklusivní přístup ke sdílenému zdroji.
2. Úloha která již vlastní sdílené zdroje, musí žádat o další.
3. Úlohu nelze přinutit (např. pomocí OS) aby zdroj vrátila.
4. Musí existovat dvě nebo více úloh, které v kruhu čekají na stejný prostředek.

Obecně deadlock **NELZE** předvídat! Je to ekvivalentní úloha s problémem zastavení programu pro libovolná vstupní data (A. Turing).



Deadlock





Deadlock

1. Úloha během svého chodu musí vyžadovat exklusivní přístup ke sdílenému zdroji.
 2. Úloha která již vlastní sdílené zdroje, musí žádat o další.
 3. Úlohu nelze přinutit (např. pomocí OS) aby zdroj vrátila.
 4. Musí existovat dvě nebo více úloh, které v kruhu čekají na stejný prostředek.
-
1. Vyvarovat se exkluzivnímu přístupu ke zdroji (např. pomocí front). V RTOS lze jen zřídka a jen pro určité zdroje. Lze také zajistit při velmi pečlivém návrhu systému nebo pomocí mutexů - RTOS.
 2. Žádat o požadované zdroje naráz a úlohu spustit jen když jsou všechny k dispozici. V RTOS jen pro speciální úlohy.
 3. Přinutit úloh, aby vrátila zdroj je v podstatě nemožné a používá se jen ve fail-stop nebo fail-safe systémech.
 4. Vyvarovat se nehierarchickému přidělování zdrojů (pořadí, ve kterém se o zdroje žádá). V RTOS lze zajistit blokování všech ostatních úloh při žádosti o sdílený zdroj (kritické sekce).



Reentrantní funkce

Reentrantní funkce je taková, které může být **bezpečně** volána v paralelním výpočetním prostředí. Tj. musí ustát opětovné zavolání i přes to, že její kód je již někde vykonáván.

Aby byla funkce reentrantní, musí splňovat tyto podmínky:

1. Funkce nesmí používat statické lokální proměnné (vyjma konstant).
2. Nesmí vracet hodnoty do statických proměnných.
3. Musí pracovat pouze s daty, které jsou jí poskytnuty při zavolání (tedy přes zásobník).
4. Nesmí exklusivně využívat sdílené zdroje.
5. Nesmí volat jiné ne-reentrantní funkce.

Často je nutná dobrá znalost překladače!

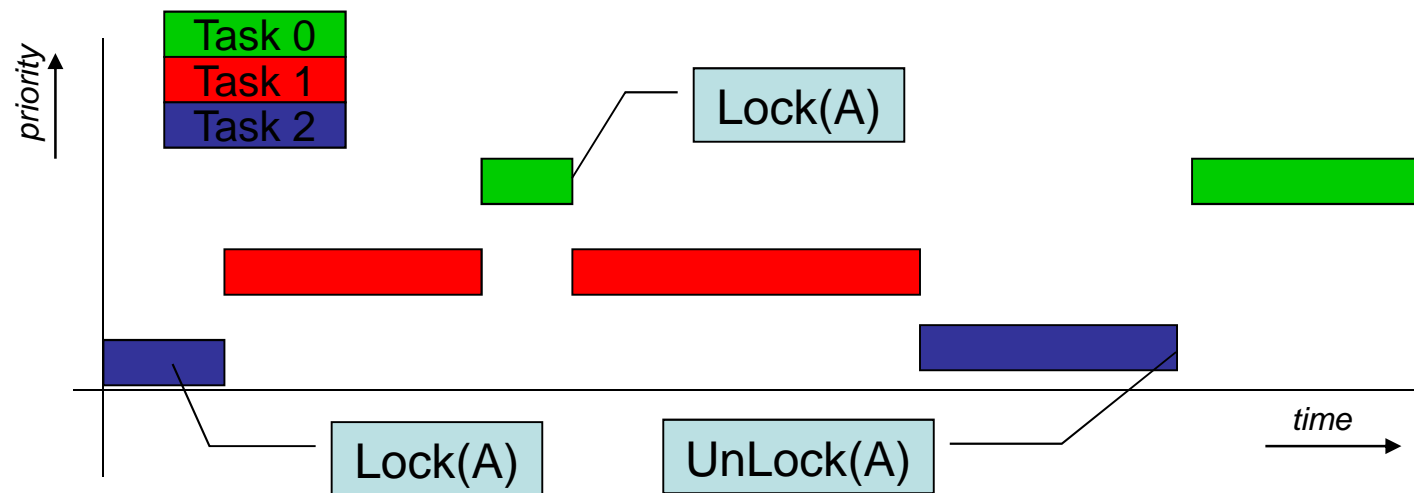


Inverze priorit

Jev, kdy úloha s nižší prioritou (zde Task 2) zablokuje sdílený zdroj, který je posléze vyžadován úlohou s vyšší prioritou (zde Task 0).

Vykonávání úlohy s vyšší prioritou je pak blokováno, dokud úloha s nižší prioritou sdílený zdroj neuvolní.

Situace se často zhoršuje kvůli úloze běžící s prostřední prioritou (zde Task 1), která pak dostane více prostoru k běhu; více než Task 2, protože má vyšší prioritu a také více než Task 0, protože tento Task je ve stavu BLOCKED).





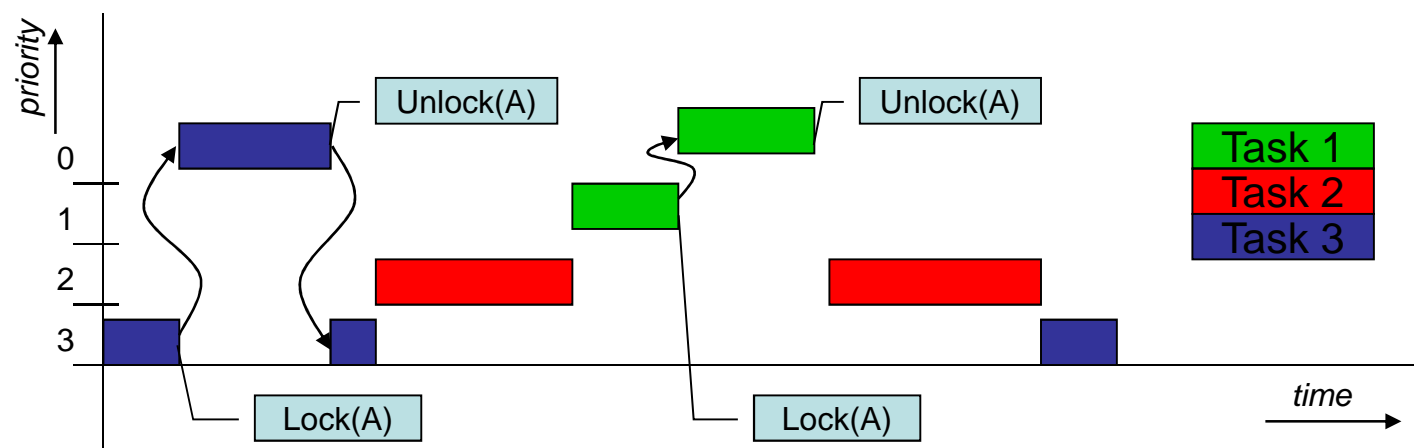
Inverze priorit – priority ceiling

Jeden ze způsobů, jak eliminovat inverzi priorit.

Pracuje tak, že každému sdílenému zdroji je přidělena priorita tak, že je vyšší než-li dosahuje libovolná úloha v systému (přínejmenším o 1).

Pokud nějaká úloha žádá o takový zdroj, obdrží dočasně prioritu tohoto zdroje a po jeho uvolnění se priorita úlohy vrátí na původní úroveň.

Vyžaduje podporu RTOS.





Inverze priorit – priority inheritance

Způsob, jak eliminovat inverzi priorit.

Pracuje tak, že úloha s nižší prioritou zdědí prioritu úlohy s vyšší prioritou, které žádá o sdílený zdroj. Tato změna priority se uskuteční v okamžiku, kdy úloha s vyšší prioritou začne žádat o přístup ke sdílenému zdroji. Úloze je navracena její původní priorita po navrácení zdroje.

Vyžaduje spolupráci RTOS.

