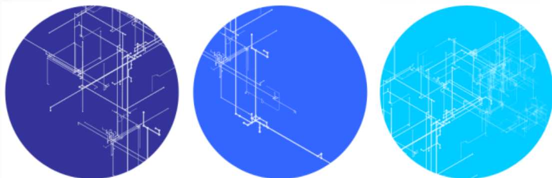
PB153
OPERAČNÍ SYSTÉMY A JEJICH ROZHRANÍ



Synchronizace procesů

08

PARALELNÍ BĚH PROCESŮ

- Synchronizace běhu procesů
  - jeden proces čeká na událost z druhého procesu
- Komunikace mezi procesy
  - komunikace – způsob synchronizace, koordinace různých aktivit
  - může dojít k uváznutí
    - každý proces čeká na zprávu od nějakého jiného procesu
  - může dojít ke stárnutí
    - dva procesy si opakovaně posílají zprávy zatímco třetí proces čeká na zprávu nekonečně dlouho
- Sdílení prostředků
  - procesy používají a modifikují sdílená data
  - operace zápisu musí být vzájemně výlučné
  - operace zápisu musí být vzájemně výlučné s operacemi čtení
    - operace čtení mohou být realizovány souběžně
  - pro zabezpečení integrity dat se používají kritické sekce

NEKONZISTENCE

- Paralelní přístup ke sdíleným údajům může být příčinou nekonzistence dat
- Udržování konzistence dat vyžaduje používání mechanismů, které zajistí patřičné provádění spolupracujících procesů
- Problém komunikace procesů v úloze typu Producent-Konzument přes vyrovnávací paměť s omezenou kapacitou

PRODUCENT-KONZUMENT (1)

- Sdílená data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

PRODUCENT-KONZUMENT (2)

- Producent

```
item nextProduced;

while (1) {
	while (counter == BUFFER_SIZE)
		; /* do nothing */
	buffer[in] = nextProduced;
	in = (in + 1) % BUFFER_SIZE;
	counter++;
}
```

PRODUCENT-KONZUMENT (3)

- Konzument

```
item nextConsumed;

while (1) {
    while (counter == 0)
            ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

PRODUCENT-KONZUMENT (4)

- Atomická operace je taková operace, která vždy proběhne bez přerušení
- Následující příkazy musí být atomické
  - **counter++;**
  - **counter--;**
- count++ v assembleru může vypadat
  - **register1 = counter**
  - **register1 = register1 + 1**
  - **counter = register1**
- count-- v assembleru může vypadat
  - **register2 = counter**
  - **register2 = register2 – 1**
  - **counter = register2**

PRODUCENT-KONZUMENT (5)

- Protože takto implementované operace count++ a count-- nejsou atomické, můžeme se dostat do problémů s konzistencí
- Nechť je hodnota **counter** nastavena na 5. Může nastat:
  - producer: **register1 = counter** (*register1 = 5*)
  - producer: **register1 = register1 + 1** (*register1 = 6*)
  - consumer: **register2 = counter** (*register2 = 5*)
  - consumer: **register2 = register2 – 1** (*register2 = 4*)
  - producer: **counter = register1** (*counter = 6*)
  - consumer: **counter = register2** (*counter = 4*)
- Výsledná hodnota proměnné **counter** bude buďto 4 nebo 6, zatímco správný výsledek má být 5.

RACE CONDITION

- Race condition (podmínka soupeření):
  - více procesů současně přistupuje ke sdíleným zdrojům a manipulují s nimi
  - konečnou hodnotu zdroje určuje poslední z procesů, který zdroj po manipulaci opustí
- Ochrana procesů před negativními dopady race condition
  - je potřeba procesy synchronizovat

PROBLÉM KRITICKÉ SEKCE

- N procesů soupeří o právo používat jistá sdílená data
- V každém procesu se nachází segment kódu programu nazývaný *kritická sekce*, ve kterém proces přistupuje ke sdíleným zdrojům
- Problém:
  - je potřeba zajistit, že v kritické sekci, sdružené s jistým zdrojem, se bude nacházet nejvýše jeden proces

PODMÍNKY ŘEŠENÍ PROBLÉMU KRITICKÉ SEKCE

- Podmínka vzájemného vyloučení (mutual exclusion), podmínka bezpečnosti, „safety“
  - jestliže proces P1 provádí svoji kritickou sekci, žádný jiný proces nemůže provádět svoji kritickou sekci sdruženou se stejným zdrojem
- Podmínka trvalosti postupu (progress), podmínka živosti, „liveliness“
  - jestliže žádný proces neprovádí svoji sekci sdruženou s jistým zdrojem a existuje alespoň jeden proces, který si přeje vstoupit do kritické sekce sdružené s tímto zdroje, pak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat nekonečně dlouho
- Podmínka konečnosti doby čekání (bounded waiting), podmínka spravedlivosti, „fairness“
  - musí existovat horní mez počtu, kolikrát může být povolen vstup do kritické sekce sdružené s jistým zdrojem jiným procesům než procesu, který vydal žádost o vstup do kritické sekce sdružené s tímto zdrojem, po vydání takové žádosti a před tím, než je takový požadavek uspokojen
  - předpokládáme, že každý proces běží nenulovou rychlostí
  - o relativní rychlosti procesů nic nevíme

POČÁTEČNÍ NÁVRHY ŘEŠENÍ

- Máme pouze 2 procesy, P_0 a P_1
- Generická struktura procesu P_i

```
do {
        entry section
            critical section
        exit section
            reminder section
    } while (1);
```

- Procesy mohou za účelem dosažení synchronizace svých akcí sdílet společné proměnné
- Činné čekání na splnění podmínky v „entry section“ – „busy waiting“

ŘEŠENÍ PROBLÉMU KS

- Softwarová řešení
  - algoritmy, jejichž správnost se nespoléhá na žádné další předpoklady
  - s aktivním čekáním „busy waiting“
- Hardwarová řešení
  - vyžadují speciální instrukce procesoru
  - s aktivním čekáním
- Řešení zprostředkované operačním systémem
  - potřebné funkce a datové struktury poskytuje OS
  - s pasivním čekáním
  - podpora v programovacím systému/jazyku
    - semafory, monitory, zasílání zpráv

ALGORITMUS 1

- Sdílené proměnné
  - **int turn**;
    počátečně **turn = 0**
  - **turn = i** ⇒ P_i může vstoupit do KS
- Proces P_i

```
do {
        while (turn != i) ;
                critical section
        turn = j;
                reminder section
} while (1);
```

- Splňuje vzájemné vyloučení, ale ne trvalost postupu

ALGORITMUS 2

- Sdílené proměnné
  - **boolean flag[2]**;
    počátečně **flag [0] = flag [1] = false.**
  - **flag [i] = true** ⇒ P_i může vstoupit do své KS
- Process P_i

```
do {
        flag[i] := true;
        while (flag[j]) ;
    critical section
        flag [i] = false;
            remainder section
} while (1);
```

- Splňuje vzájemné vyloučení, ale ne trvalost postupu

ALGORITMUS 3 (PETERSONŮV)

- Kombinuje sdílené proměnné algoritmů 1 a 2
- Proces P_i

```
do {
        flag [i]:= true;
        turn = j;
        while (flag [j] and turn == j) ;
                critical section
        flag [i] = false;
                remainder section
} while (1);
```

SYNCHRONIZAČNÍ HARDWARE

- Speciální instrukce strojového jazyka
  - test_and_set, exchange / swap, …
- Stále zachována idea používání „busy waiting“
- Test_and_set
  - testování a modifikace hodnoty proměnné – atomicky

```
boolean TestAndSet (boolean &target) {
        boolean rv = target;
        target = true;
        return rv;
    }
```

- Swap
  - Atomická výměna dvou proměnných

```
Void Swap (boolean &ra, boolean &rb] {
        boolean temp = a;
        a = b;
        b = temp;
    }
```

VYUŽITÍ TESTANDSET

- Sdílená data (inicializováno na false):

  boolean lock:

- Proces Pi

```
do {
        while (TestAndSet(lock)) ;
                critical section
        lock = false;
                remainder section
   }
```

VYUŽITÍ SWAP

- Sdílená data (inicializováno na false):

  boolean lock;

  boolean waiting[n];

- Proces Pi

```
do {
        key = true;
        while (key == true)
                Swap(lock,key);
            critical section
        lock = false;
            remainder section
    }
```

SITUACE BEZ PODPORY OS

- Nedostatek softwarového řešení
  - procesy, které žádají o vstup do svých KS to dělají metodou „busy waiting“
    - po nezanedbatelnou dobu spotřebovávají čas procesoru
- Speciální instrukce
  - výhody
    - vhodné i pro multiprocesory (na rozdíl od prostého maskování/povolení přerušení)
  - nevýhody
    - opět „busy waiting“
    - možnost stárnutí – náhodnost řešení konfliktu
    - možnost uváznutí v prioritním prostředí (proces v KS nedostává čas CPU)

SEMAFORY

- Synchronizační nástroj, který lze implementovat i bez „busy waiting“
  - proces je (operačním systémem) „uspán“ a „probuzen“
  - tj. řešení na úrovni OS
- Definice

  Semaphore S : integer

- Lze ho zpřístupnit pouze pomocí dvou atomických operací

wait (S):

while S ≤ 0 do no-op;

S --;

signal(S):

S ++;

KRITICKÁ SEKCE

- Sdílená data:

  semaphore mutex;   // počátečně mutex = 1

- Proces P_i:

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (1);
```

SYNCHRONIZACE SEMAFOREM

- Má se provést akce B v P_j pouze po té, co se provede akce A v P_i
- Použije se semafor flag inicializovaný na 0
- Program:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

.   .

UVÁZNUTÍ A STÁRNUTÍ

- Uváznutí
  - dva nebo více procesů neomezeně dlouho čekají na událost, kterou může generovat pouze jeden z čekajících procesů
  - Nechť S a Q jsou dva semafory inicializované na 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- Stárnutí
  - neomezené blokování, proces nemusí být odstraněný z fonty na semafor nikdy (předbíhání vyššími prioritami, …)

DVA TYPY SEMAFORŮ

- Obecný semafor S
  - celočíselná hodnota z neomezovaného intervalu
- Binární semafor
  - celočíselná hodnota z intervalu <0,1>
- Implementovatelnost
  - binární semafor lze snadněji implementovat
  - obecný semafor lze implementovat semaforem binárním

PROBLÉMY SE SEMAFORY

- Semafory jsou mocný nástroj pro dosažení vzájemného vyloučení a koordinaci procesů
- Operace wait(S) a signal (S) jsou prováděny více procesy a jejich účinek nemusí být vždy explicitně zřejmý
  - semafor s explicitním ovládáním wait/signal je nástroj nízké úrovně
- Chybné použití semaforu v jednom procesu hroutí souhru všech spolupracujících procesů
- Patologické případy použití semaforů:

wait(x)	wait(x)
KS	KS
wait(x)	signal(y)

KRITICKÉ OBLASTI

- Konstrukt programovacího jazyka vysoké úrovně
- Sdílená proměnná v typu T, je deklarována jako:

  v: shared T

- Proměnná v je dostupná pouze v příkazu

  region v when B do S

  kde B je booleovský výraz

- Po dobu, po kterou se provádí příkaz S, je proměnná v pro jiné procesy nedostupná
- Oblasti referující stejnou sídlenou proměnnou se v čase vzájemně vylučují

KRITICKÉ OBLASTI (2)

- Když se proces pokusí provést příkaz region, vyhodnotí se Booleovský výraz B
- Je-li B pravdivý, příkaz S se provede
- Je-li B nepravdivý, provedení příkazu S se oddálí do doby až bude B pravdivý a v oblasti (region) spojené s V se nenachází žádný proces

MONITORY

- Synchronizační nástroj vysoké úrovně
- Umožňuje bezpečné sdílení abstraktního datového typu souběžnými procesy
- Provádění P_1, P_2, … se implicitně vzájemně vylučují

```
monitor monitor-name
{
  shared variable declarations
  procedure body P1 (…) {
  . . .
  }
  procedure body P2 (…) {
  . . .
  }
  procedure body Pn (…) {
   . . .
  }
  {
  initialization code
  }
}
```

PŘÍKLAD: LINUX

- Synchronizace
  - signály (asynchronní události)
  - wait
  - semafory (semaphores)
  - mutexy v pthreads
- Komunikace
  - nepojmenované roury (ls|pr|lpr) fce popen/pclose, …
  - pojmenované roury (mkfifo)
  - zprávy (messages, message queues)
  - sdílená paměť (shared memory)

LINUX: Signály dle POSIX

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

LINUX: Signály

```
1)  SIGHUP       2)  SIGINT       3)  SIGQUIT      4)  SIGILL
5)  SIGTRAP      6)  SIGABRT      7)  SIGEMT       8)  SIGFPE
9)  SIGKILL     10)  SIGBUS      11)  SIGSEGV     12)  SIGSYS
13)  SIGPIPE    14)  SIGALRM     15)  SIGTERM     16)  SIGURG
17)  SIGSTOP    18)  SIGTSTP     19)  SIGCONT     20)  SIGCHLD
21)  SIGTTIN    22)  SIGTTOU     23)  SIGIO       24)  SIGXCPU
25)  SIGXFSZ    26)  SIGVTALRM 27)  SIGPROF     28)  SIGWINCH
29)  SIGINFO    30)  SIGUSR1     31)  SIGUSR2
```

LINUX: posílání signálů

```
Sending a Signal
    The following system calls and library functions allow  the  caller  to
    send a signal:

    raise(3)        Sends a signal to the calling thread.

    kill(2)         Sends  a  signal to a specified process, to all members
                    of a specified process group, or to  all  processes  on
                    the system.

    killpg(2)       Sends  a  signal  to  all of the members of a specified
                    process group.

    pthread_kill(3) Sends a signal to a specified POSIX thread in the  same
                    process as the caller.

    tgkill(2)       Sends  a signal to a specified thread within a specific
                    process.  (This is the system call  used  to  implement
                    pthread_kill(3).)

    sigqueue(2)     Sends  a  real-time  signal with accompanying data to a
                    specified process.
```

LINUX: real-time signály

```
Real-time Signals
    Linux  supports real-time signals as originally defined in the POSIX.1b
    real-time extensions (and now included in POSIX.1-2001).  The range  of
    supported  real-time  signals  is  defined  by  the macros SIGRTMIN and
    SIGRTMAX.  POSIX.1-2001 requires  that  an  implementation  support  at
    least _POSIX_RTSIG_MAX (8) real-time signals.

    The  Linux  kernel  supports a range of 32 different real-time signals,
    numbered 33 to 64.  However, the  glibc  POSIX  threads  implementation
    internally  uses  two  (for NPTL) or three (for LinuxThreads) real-time
    signals (see pthreads(7)), and adjusts the value of  SIGRTMIN  suitably
    (to 34 or 35).  Because the range of available real-time signals varies
    according to the glibc threading implementation (and this variation can
    occur  at  run  time  according to the available kernel and glibc), and
    indeed the range of real-time signals varies across Unix systems,  pro-
    grams should never refer to real-time signals using hard-coded numbers,
    but instead should always refer to real-time signals using the notation
    SIGRTMIN+n, and include suitable (run-time) checks that SIGRTMIN+n does
    not exceed SIGRTMAX.

    Unlike standard signals, real-time signals have no predefined meanings:
    the entire set of real-time signals can be used for application-defined
    purposes.  (Note, however, that the  LinuxThreads  implementation  uses
    the first three real-time signals.)

    The  default  action  for an unhandled real-time signal is to terminate
    the receiving process.
```

LINUX: Semafory

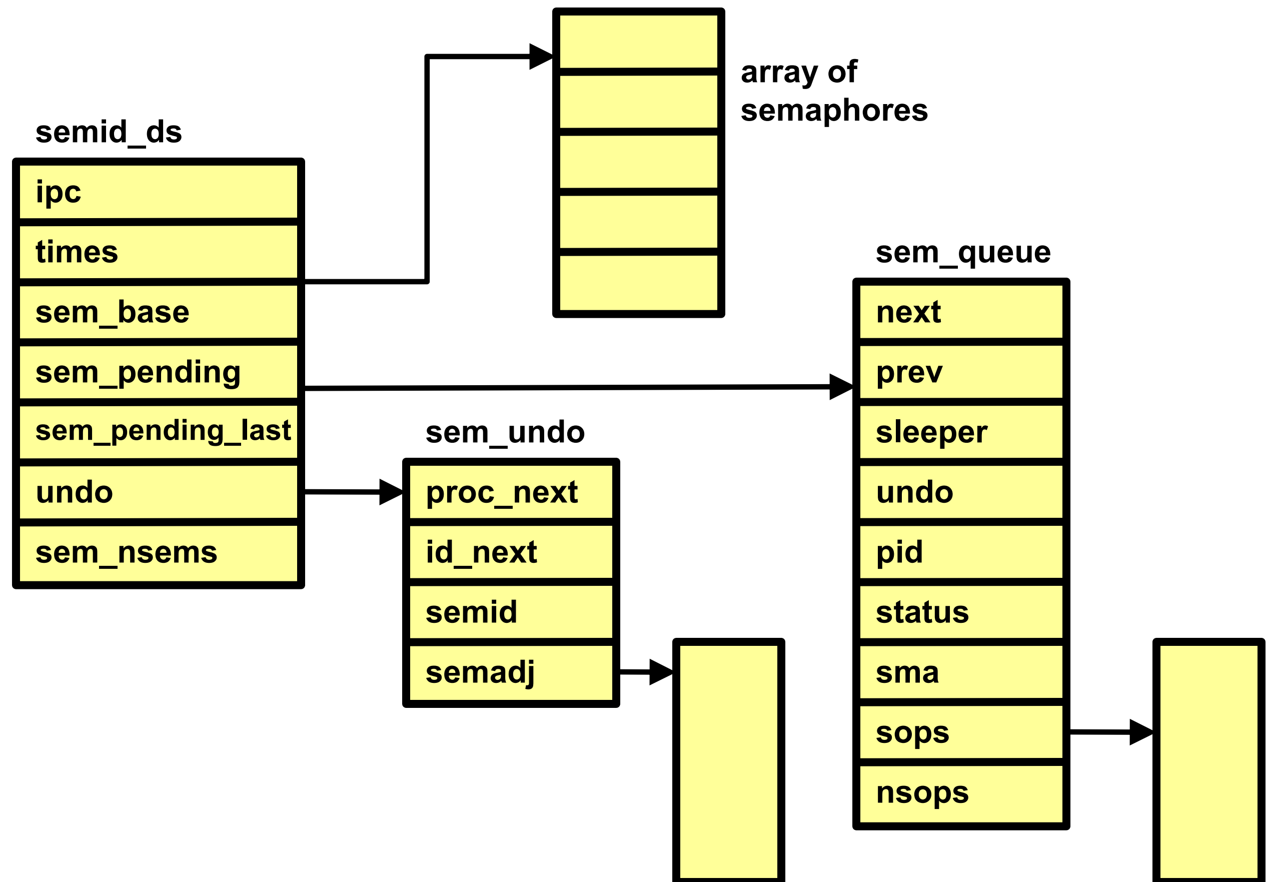
- POSIX
  - #include <semaphore.h>
  - klasické semafory dle POSIXu
- System V
  - #include <sys/sem.h>
  - modernější verze

LINUX: POSIX semafory

- `int sem_init ((sem_t *sem, int pshared, unsigned int value));`
  - Nepojmenovaný semafor
- `sem_t *sem_open (char *name, int oflag, [mode_t mode, int init_value]);`
  - Pojmenovaný semafor
- `int sem_wait ((sem_t *sem));`
- `int sem_post ((sem_t *sem));`
- `int sem_close (sem_t *sem);`
- `int sem_unlink(char *name);`
- `int sem_getvalue (sem_t *sem, int *val);`

LINUX: Semafory dle System V

- Volání jádra:
  - semget
  - semctl
  - semop



LINUX: Mutexy v pthreads

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

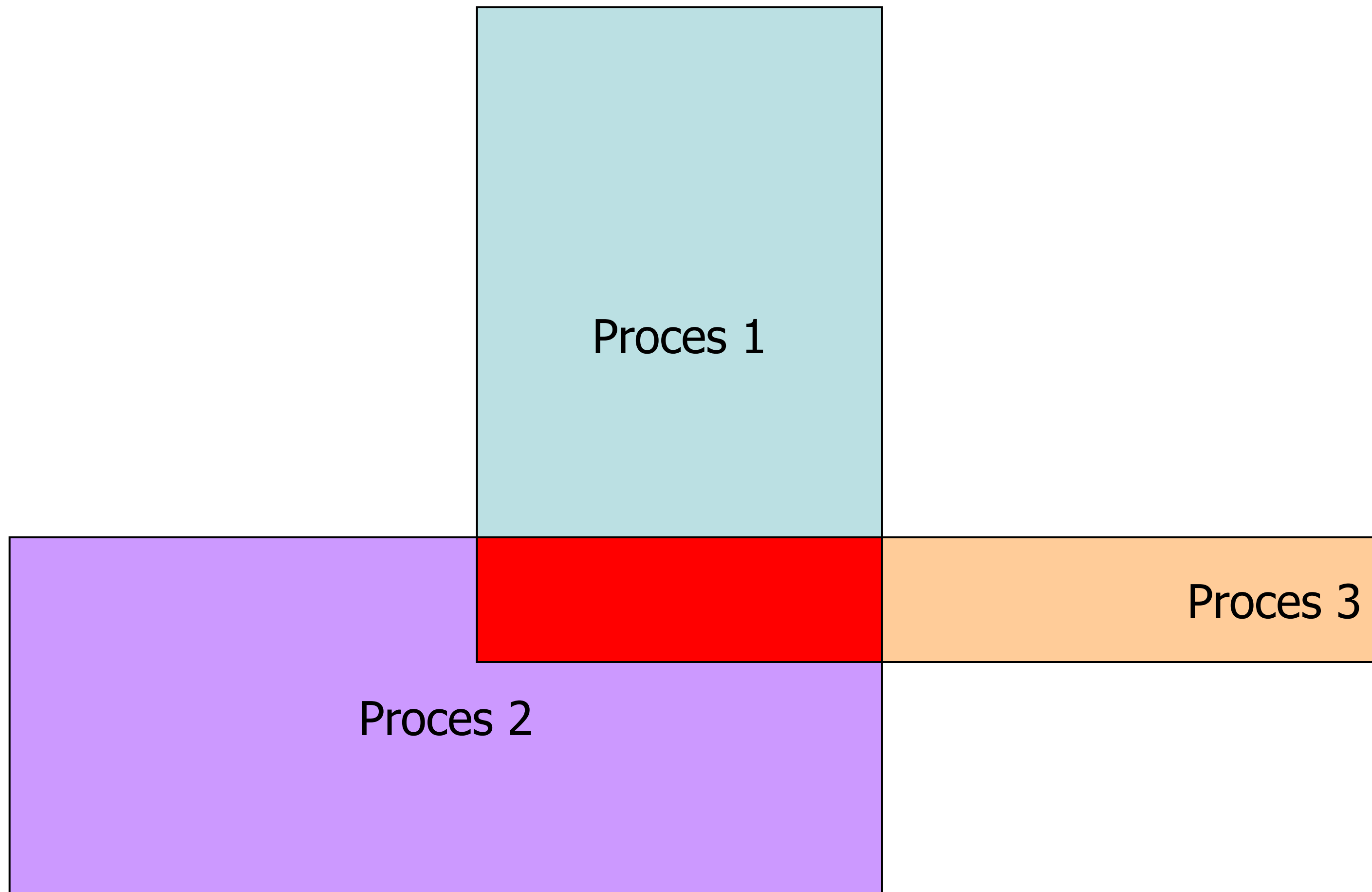
LINUX: Mutexy v pthreads

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

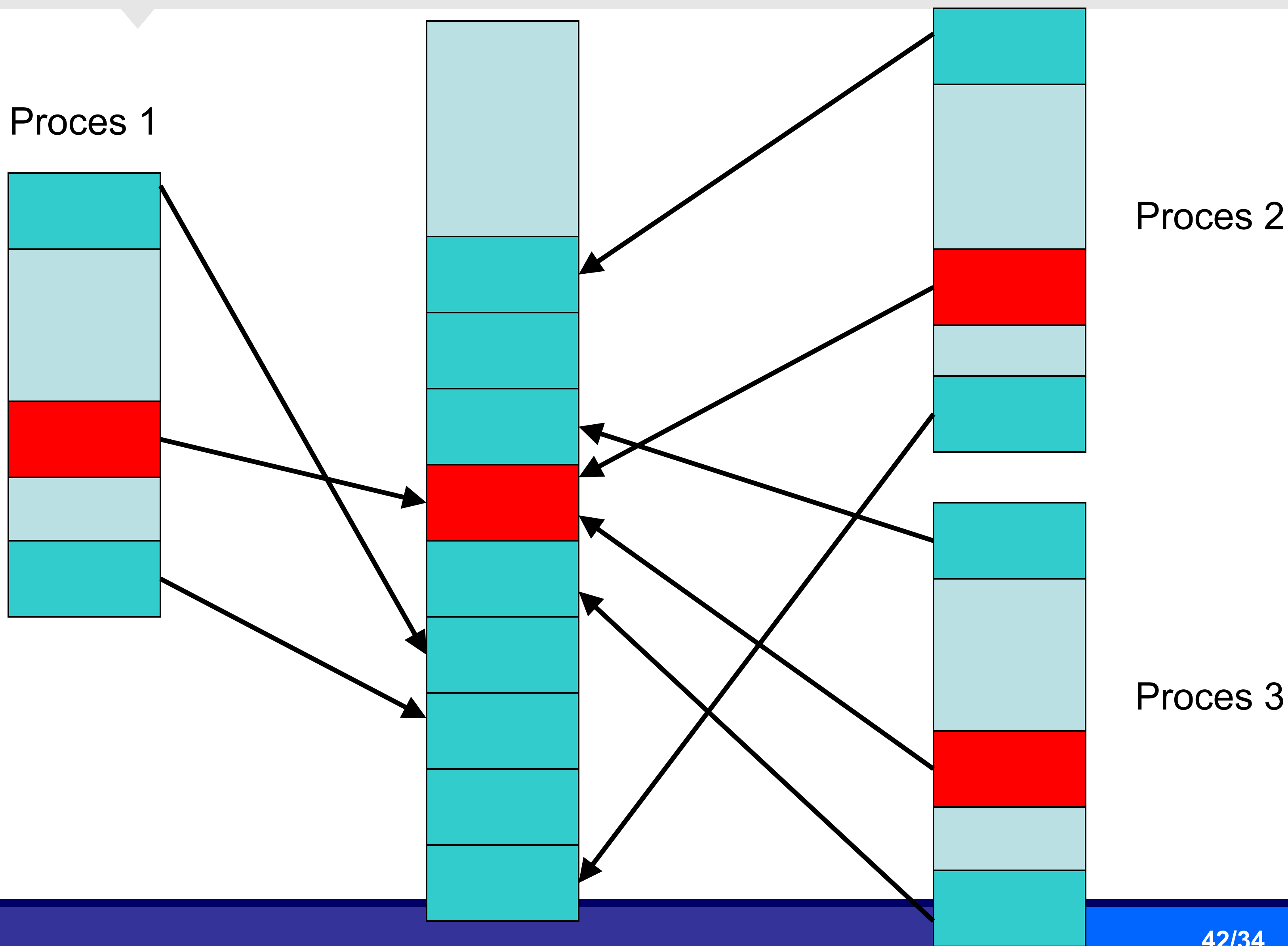
LINUX: Zprávy

- Kapacita zpráv
  - Administrátor může změnit defaultní hodnoty
  - Max msgs: 10 /queue
  - Max Msg Size: 8192
  - Max # of Queues: 256
- Virtuální message queue file system (mqueue)
  - Může být připojen pro zobrazení (a manipulaci) se zprávami
  - mkdir /dev/mqueue
  - mount –t mqueue none /dev/mqueue

Linux: sdílená paměť



Linux: sdílená paměť



LINUX: sdílená paměť

- #include <sys/ipc.h> + #include<sys/shm.h>
- int shmget (key_t, int size, int shmflg);
- void * shmat (shmid, shmaddr, shmflg);
- int shmctl (shmid, cmd, struct shmid_ds *buf);
  - cmd:
    - IPC_STAT – return status information about the shared memory in buf.
    - IPC_SET – modify the shared memory based on parameters in buf (can only change UID and mode bits)
    - IPC_RMID – Remove (deallocate) the shared memory segment specified in shmid.
    - IPC_LOCK – lock the shared memory segment in memory (don't swap out).
    - IPC_UNLOCK – release the lock on shared memory

PŘÍKLAD: WIN32 (1)

IPC Mechanism	Win2000	WinNT	Win9x	Win32s(1)	Win16(2)	MS-DOS(2)	POSIX	OS/2
DDE	YES	YES	YES	YES	YES	NO	NO	NO
OLE 1.0	YES	YES	YES	YES	YES	NO	NO	NO
OLE 2.0	YES	YES	YES	YES	YES	NO	NO	NO
NetBIOS	YES	YES	YES	YES	YES	YES	NO	YES
Named pipes	YES	YES	YES(3)	YES(3)	YES(3)	YES(3)	YES(4)	YES
Windows sockets	YES(5)	YES(5)	YES	YES	YES(5)	NO	NO(6)	NO
Mailslots	YES	YES	YES	YES(3)	NO	NO	NO	YES
Semaphores	YES	YES	YES	NO	NO	NO	YES	YES
RPC	YES	YES	YES(7)	YES(8)	YES	YES	NO	NO
Mem-Mapped File	YES	YES	YES	YES	NO	NO	NO	NO
WM_COPYDATA	YES	YES	YES	YES(9)	YES	NO	NO	NO

PŘÍKLAD: WIN32 (2)

- Semafory (obecné semafory), funkce rozhraní:
  - CreateSemaphore
  - OpenSemaphore
  - ReleaseSemaphore
  - Wait
    - SignalObjectAndWait
    - WaitForSingleObject
    - WaitForSingleObjectEx
    - WaitForMultipleObjects
    - WaitForMultipleObjectsEx
    - MsgWaitForMultipleObjects
    - MsgWaitForMultipleObjectsEx

Výukovou pomůcku zpracovalo

Servisní středisko pro e-learning na MU

CZ.1.07/2.2.00/28.0041
Centrum interaktivních a multimediálních studijních opor pro inovaci výuky a efektivní učení

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ