Operativsystem

Kommunikation mellan processer

Inter Process Communication (IPC)

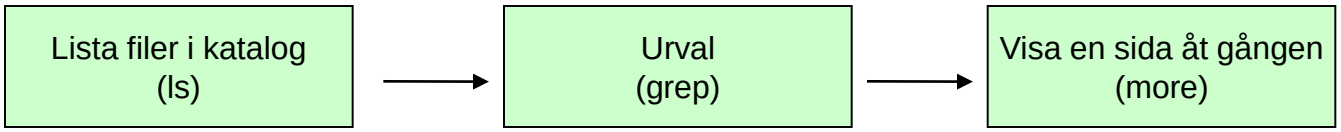
(2.3 i boken)

Kommunikation mellan processer

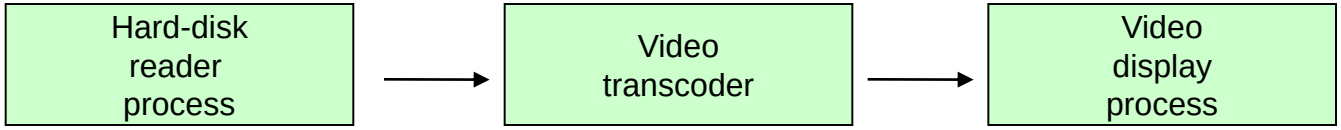
- Vi har lyckats skapa processer – program under exekvering som lever i sin egen ”skyddade värld”
  - det en process gör påverkar INTE en annan
- Men – processer måste även kommunicera med varandra – hur gör vi det när de är skyddade från varandra?
  - t.ex. den data som en process genererar fungerar som input för en annan process
- Nu vill vi med andra ord komma över de gränser vi själv har byggt in i operativsystemet....

Exempel på IPC

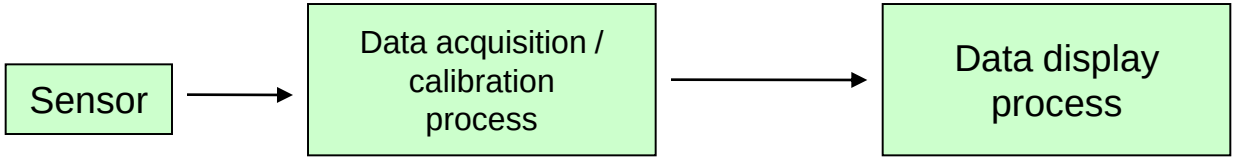
- Visa urval ur filkatalog på skärmen



- Digital video recorder



- Data acquisition



Mekanismer för IPC

- Delad data mellan processer
  - Pipes
  - FIFO:s
  - Delat minne
  - Sockets
- Meddelanden
  - Meddelandköer
- Händelsenotifieringar
  - signaler
- (Synkronisering)
  - Semaforer, mutex, aktivt väntande

Pipes

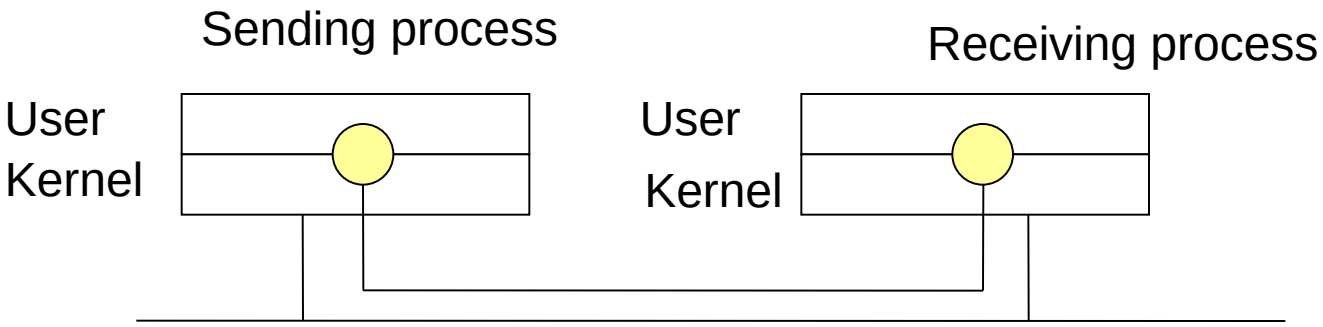
- Envägs dataflöde mellan processer
  - All data som skrivs av en process ruttas av kernelen till en annan process
- UNIX shell: vanligen ” |” kommandot
  - `$ ls | more`
- Fungerar som en fil, men inget behov av att radera temporära filer
- Skapas med hjälp av anropet `pipe()`, vilket returnerar ett par av fil-deskiptorer

FIFOs

- Pipes är simpla och effektiva, men två processer kan inte använda sig av samma pipe
  - T.ex. Databas-motor
- FIFO:
  - Inod-struktur, vilken som helst process kan accessera FIFO:n
  - FIFO:n är tvåvägs
  - skapas med unix-kommandot `mknod()`

Sockets

- Används såsom FIFO:s, men över ett nätverk
- Efter att två sockets har skapats kan man ihopkoppla dessa



Delat minne (shared memory)

- Två eller flere processer kan accessera gemensamma datastrukturer, genom att använda sig av IPC delade minnesområden
- Varje process måste addera det delade minnet till sin adressrymd
- `shmget()` - för att erhålla en ID för IPC delat minne
- `shmat()` - för att lägga till ett IPC delat minnesområde till processens minnesrymd

System V IPC meddelanden

- Meddelande skickas till en IPC meddelandekö, meddelandet köas tills en annan process läser meddelandet
- msgsnd() används för att sätta meddelandet på kön
- msgrcv() används för att hämta meddelande från kön

Signaler

- Processer kan skicka signaler till andra processer
- Processer kan antingen “fånga” (catch) eller ignorera (ignore) signaler som skickas till denna
- Signaler kan också används i andra sammanhang, t.ex. SIGFPE (division med 0)
- Vissa signaler kan inte processen “fånga”, utan kernelen tar hand om den (SIGKILL)

Signaler

- Skicka en signal

```
int kill(pid_t pid, int sig);
```

- Installera en signalhanterare

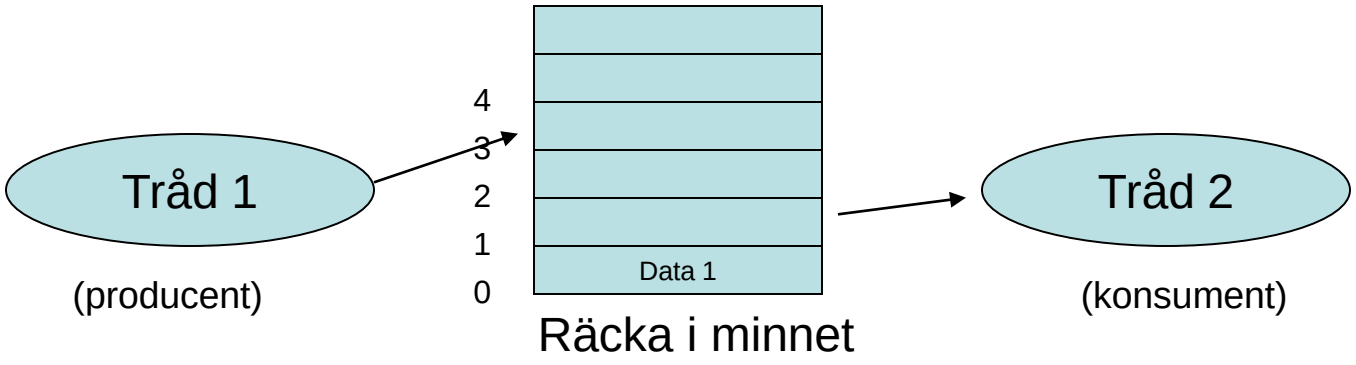
```
void (*signal(int sig, void (*func)(int)))(int);
```

- Exempel på signalhanterare

```
int g_ctrls_pressed = 0; // If Ctrl-c pressed
void sigint_handler(int sig) {
        // Ctrl-c press caused normally a SIGINT
        g_ctrlc_pressed = 1;
}
```

Kommunikation mellan trådar

- Mellan trådar (som hör till samma process) är kommunikation enklare
  - definitionsmässigt använder de samma minnesrymd
    - dvs då en tråd skriver till minnet kommer den andra trådan att kunna läsa denna information



”Race condition”

```
#define BUFFERSIZE 100

MYDATA Array[BUFFERSIZE];

int Consumer() {
  int i;
  for (i=0; i<BUFFERSIZE; i++) {
   PrintData(Array[i]);
  }
}

int Producer() {
  int i;
  for (i=0; i<BUFFERSIZE; i++) {
   GetDataFromWorld(&Array[i]);
  }
}

main() {
   StartThread(Producer);
   StartThread(Consumer);
}
```

Problem:

Hur skall vi se till att konsumenten inte försöker konsumera sådant som ännu inte är producerat??

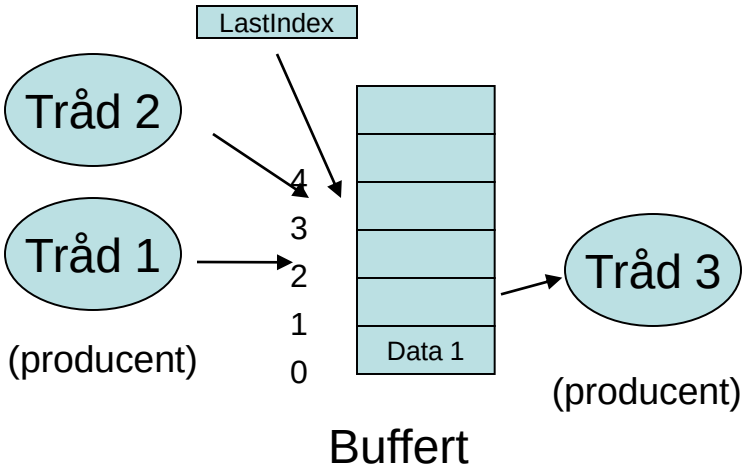
Problematiken kallas på engelska för

”*RACE CONDITION*”

Allmänt: Den problematik som karakteriseras av att utkomsten av en beräkning är beroende av i vilken ordning saker råkar utföras.

Race condition (2)

Lösning: Vi inför en variabel som säger var i bufferten producenten senast har skrivit




```
int Consumer() {
  int i;
  for (i=0; i<BUFFERSIZE; i++) {
   while (i>=LastIndex) ;
   PrintData(Array[i]);
  }
}

int Producer() {
  int i;
  ++LastIndex;
  GetDataFromWorld(
   &Array[LastIndex]);
}
```

Nytt problem: Vad om följande händer:

1. Producent 1 uppdaterar LastIndex
2. Skeduleraren ger över till producent 2
3. Producent 2 uppdaterar LastIndex
4. Producent 2 skriver till bufferten
5. Producent 1 skriver till bufferten, MEN TILL SAMMA PLATS I BUFFERTEN SOM PRODUCENT 2!!!

Kritiska sektioner

- Löser problemet med dubbel uppdatering genom att
  - se till att endast en exekveringstig* åt gången utför programavsnitt som kan leda till ”race condition”
  - vi kallar ett sådant avsnitt i koden för en **KRITISK SEKTION**

```
int Producer() {
  ++LastIndex;
  GetDataFromWorld(
&Array[LastIndex]);
}
```

KRITISK SEKTION

**Exekveringsstig = tråd / process / avbrotts-kod eller liknande som karakteriseras av denna följer sin egen programstig, oberoende av andra*

Kritisk sektion (2)

- För en kritisk sektion vill vi att följande gäller
  - Endast en exekveringsstig kan åt gången vara i samma kritiska sektion
- Om en exekveringsstig är i sin kritiska sektion, måste övriga exekveringsstigar som är på väg in i denna samma kritiska sektion temporärt blockeras (exekveringsstigarna måste synkroniseras)
  - MEN får inte blockaras för evigt!!

Hur blockera exekveringsstigar?

- Stäng av avbrott
  - Repetition: Byte av process sker ofrivilligt endast vid avbrott (t.ex. klock-avbrott)

```
int Producer() {

  DisableInterrupts();  //assembler cli;
  ++LastIndex;
  GetDataFromWorld(
&Array[LastIndex]);
  EnableInterrupts();  //assembler sti;
}
```

• Normalt endast ett alternativ i kernel-kod (och där har det använts ganska flitigt). Att stänga av avbrott är dock ett ganska säkert sätt att få maskinen att låsa sig, om man klåpat i koden.
• Dvs. inte ett alternativ för användarprocesser
• Hårdvarubaserad metod!

Blockering genom aktivt väntande – ”busy waiting”

- Vi inför en lås-variabel

```
int Lock = 1; // Default Lock=1: Not locked
int Producer() {
CheckForLock:
  Lock--;  // Must be atomic
  if (Lock!=0) { // Was I the unlucky one??
    while (Lock!=1) ;  // Wait on the lock being
                released, then start over
    goto CheckForLock;
  }
  ++LastIndex;
  GetDataFromWorld(
&Array[LastIndex]);
  Lock=1; // Must be atomic
}
```

- Problem: Konsumerar klockcykler!!
- Bra för korta låsningar

Linux: ”Spin-lock”

```
spin_lock()
1: lock; decb slp
   jns 3f
2: cmpb $0, slp
   pause
   jle 2b
   jmp 1b
3:

spin_unlock()
locki; movb $1, slp
```

Strikt alternering – aktivt väntande

```
int turn=0;
int Producer1() {
while (turn!=0) ;
  ++LastIndex;
  GetDataFromWorld(
&Array[LastIndex]);
  turn = 1;
}
```

```
int Producer2() {
while (turn!=1) ;
  ++LastIndex;
  GetDataFromWorld(
&Array[LastIndex]);
  turn = 0;
}
```

- **Problem 1**: Om någondera processen är betydligt snabbare än den andra, begränsas den snabbare processen ändå av den långsammare p.g.a. den strikta alternering (de MÅSTE utföras turvis)
  Lösning: Petersons (1981) – ömsesidig uteslutning
- **Problem 2**: Aktivt väntande
- **Problem 3**: Endast för 2 processer

Sleep / wakeup

- Aktivt väntande konsumerar klock-cykler, behövs även andra synkroniseringsmetoder
  - sleep() / wakeup()
  - sleep() – exekveringsstigen blockeras och sätts på en väntekö
  - wakeup() – väcker en exekveringsstig som tidigare blockerats

```
#define BUFFERSIZE 100

int count = 0;
int Producer() {

  while (TRUE) {
    if (count == BUFFERSIZE) sleep();
   GetDataFromWorld(
&Array[++count]);
  if (count==1) wakeup (consumer);
 }
}
```

```
int Consumer() {
 while (TRUE) {
  if (count==0) sleep();
  PrintData(Array[--count]);
  if (count == BUFFERSIZE-1)
    wakeup(producer);
 }
}
```

I praktiken implementeras sleep() / wakeup() mer sällan på användarnivå
Linux kernel använder sig av interna sleep_on() och wake_up()

Sleep / wakeup

- Problem
  - Vad händer om den väckande processen använder sig av wake_up före den blockerande processen använder sleep() ?
    - wake_up() går förlorat och processerna förlorar synkronisering

Semaforer

- E.W. Dijkstra föreslog att använda sig av en heltalsvariabel för att räkna hur många wakeups som finns ”i lager” för framtida bruk
- Ursprungsoperationer *down* och *up*
- *down*: Minskar antalet wake-up, om värdet går till 0, gå och sova
- *up*: Om någon process sover i väntan på semaforen, väck processen, annars öka antalet väntande wake-up
- If any processes sleeping on the semaphore, issue a wake-up, else increse the number of saved wake-ups
- Linux erbjuder semaforer på två nivåer
  - Kernel-semaforer, som används internt av Linux
  - System V semaforer, som används av användarprocesser

Mutex

- Då man inte behöver räkne-egenskapen hos en semafor, används en något enklare konstruktion, kallad mutex
- Mutex kan vara antingen olåst eller låst, endast en process kan låsa mutexen, andra processer/trådar blockerar om de försöker låsa mutex
  - Då den låsande processen frigör mutex, kommer en av de väntande processerna att erhålla mutex

Producent / konsument med semaforer

```
#define BUFFERSIZE 100

typedef int semaphore;

semaphore mutex=1;

semaphore empty = BUFFERSIZE;

semaphore full = 0;
int Producer() {

  while (TRUE) {
    down(&empty);
    down(&mutex);
    GetDataFromWorld(
&Array[full]);
    up(&mutex);
    up(&full);
  }
}
```

```
int Consumer() {
 while (TRUE) {
  down(&full);
  down(&mutex);
  PrintData(Array[full]);
  up(&mutex);
  up(&empty);
  }
}
```

Monitorer

- Monitorer samlar procedurer och datastrukturer i en modul eller paket
  - Jämför med en klass i ett objektorienterat språk
- Endast en process kan vara aktiv i en monitor åt gången
  - java implementerar monitorer genom:
    - syncronized methods + wait / notify / notifyAll

Linux semaforer

include/asm-i386/semaphore.h

Speciell lock-Instruktion på i386, Gör att nästa instruktion ej avbryts

```
struct semaphore {
        atomic_t count;
        int sleepers;
        wait_queue_head_t wait;
};
```

```
static inline void up(struct semaphore *
sem)
{
        __asm__ __volatile__(
                "# atomic up
operation\n\t"
                LOCK_PREFIX "incl
%0\n\t"     /* ++sem->count */
                "jg 1f\n\t"
                "lea %0,%%eax\n\t"
                "call __up_wakeup\n"
                "1:"
                :"+m" (sem->count)
                :
                :"memory","ax");
}
```

```
static inline void down(struct semaphore * sem)
{
        might_sleep();
        __asm__ __volatile__(
                "# atomic down operation\n\t"
                LOCK_PREFIX "decl %0\n\t"
     /* --sem->count */
                "jns 2f\n"
                "\tlea %0,%%eax\n\t"
                "call __down_failed\n"
                "2:"
                :"+m" (sem->count)
                :
                :"memory","ax");
}
```

Kernelsemaforer i Linux

```
up:
  movl $sem, %ecx
  lock; incl (%ecx)
  jg 1f
  pushl %eax
  pushl %edx
  pushl ecx
  call __up
  popl %ecx
  popl %edx
  popl %eax
```

```
down:
  movl $sem, %ecx
  lock; decl (%ecx)
  jg 1f
  pushl %eax
  pushl %edx
  pushl ecx
  call __down
  popl %ecx
  popl %edx
  popl %eax
```

```
void __up(struct semaphore *sem)
{
        wake_up(&sem->wait);
}

void __down(struct semaphore * sem)
{
        struct task_struct *tsk = current;
        DECLARE_WAITQUEUE(wait, tsk);
        tsk->state = TASK_UNINTERRUPTIBLE;
        add_wait_queue_exclusive(&sem->wait, &wait);

        spin_lock_irq(&semaphore_lock);
        sem->sleepers++;
        for (;;) {
                int sleepers = sem->sleepers;

                /*
                 * Add "everybody else" into it. They aren't
                 * playing, because we own the spinlock.
                 */
                if (!atomic_add_negative(sleepers - 1, &sem->count)) {
                        sem->sleepers = 0;
                        break;
                }
                sem->sleepers = 1;      /* us - see -1 above */
                spin_unlock_irq(&semaphore_lock);

                schedule();
                tsk->state = TASK_UNINTERRUPTIBLE;
                spin_lock_irq(&semaphore_lock);
        }
        spin_unlock_irq(&semaphore_lock);
        remove_wait_queue(&sem->wait, &wait);
        tsk->state = TASK_RUNNING;
        wake_up(&sem->wait);
}
```

System V IPC semaforer

- Motsvarar kernelsemaforer, MEN
  - Varje IPC-semafor är en mängd av ett eller flere semaforvärden
  - System V IPC -semaforer erbjuder en fail-safe mekanik i situationer då en process dör utan att slutföra sina semafor-operationer
  - Operationer
    - `int semid = semget(key_t key, int nsems, int semflg)`
    - `int res = semop(int semid, struct sembuf *sops, unsigned nsops)`
    - `int res = semctl(int semid, int semnum, int cmd, union semun arg)`

Sys V IPC semaforer - exempel

```
#define SEMKEY 73833
int semid;
  struct sembuf ops;

  // Get the database semaphore
  semid = semget(SEMKEY, 1, 0);
  printf("Got semid: %i\n", semid);

  while (1) {
    usleep(rand() % READSLEEP);
    // Simulating write
    //Locking database
    ops.sem_num = 0;
    ops.sem_op = -1; // Decrement, same as down
    ops.sem_flg = 0;
    printf("Thread %i waiting for database \n", threadnum);
    semop(semid, &ops, 1);
    printf("Thread %i reading from database \n", threadnum);
    usleep(READSLEEP);
    ops.sem_op = 1;
    semop(semid, &ops,1);
    printf("Thread %i finished reading from database \n", threadnum);
  }
}
```

POSIX semaforer

sem_open() -- Connects to, and optionally creates, a named semaphore

sem_init() -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

sem_close() -- Ends the connection to an open semaphore.

sem_unlink() -- Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it.

sem_destroy() -- Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

sem_getvalue() -- Copies the value of the semaphore into the specified integer.

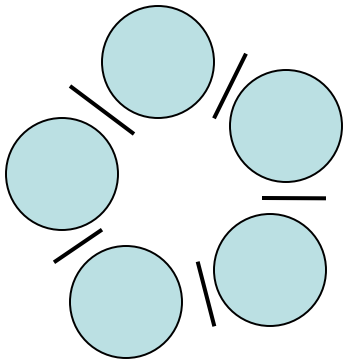
sem_wait(), sem_trywait() -- Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process.

sem_post() -- Increments the count of the semaphore.

Klassiska IPC-problem

The Dining Philosophers Problem — Ett måste för varje OS-kurs

- **Ätande filosofer**: *n* filosofer kring ett runt bord: varje filosof har en tallrik, det finns en gaffel mellan två tallrikar (t.ex. 5 tallrikar = 5 gafflar)
  - en filosof antingen tänker eller äter
  - för att äta behöver han två gafflar – när han blir hungrig försöker ha ta två gafflar och börja äta



- Problemet: Skriv ett program som ser till att varje filosof får äta

Dining philosophers - lösning

```
#define N 5
#define LEFT(i) (i+N-1) %N
#define RIGHT(i) (i+1) %N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i) {
  while(TRUE) {
    think();
    take_forks(i)
    eat();
    put_forks(i);
  }
}
void take_forks(int i) {
  down(&mutex);
  state[i] = HUNGRY;
  test(i);
  up(&mutex);
  down(&s[i]);
}
```

```
void put_forks(int i) {
  down(&mutex);
  state[i] = THINKING;
  test(LEFT(i));
  test(RIGHT(i));
  up(&mutex);
}

void test(int i) {
  if (state[i} == HUNGRY &&
    state[LEFT(i)] != EATING &&
    state[RIGHT(i)] != EATING)
    {
      state[i] = EATING;
      up(&s[i]);
    }
}
```

Readers / writers problem

- Processer/trådar i en databas fungerar som
  - skrivare (exklusiv access till databasen)
  - läsare (många samtidiga läsare OK)
- Hur se till att villkoren ovan uppfylls?

Readers / writers - lösning

```
typedef int semaphore;
semaphore mutex = 1; //access to rc
semaphore db = 1;
int rc = 0; // readers count

void reader(void) {
  while (TRUE) {
    down(&mutex);
    rc ++;
    if (rc==1) down(&db);
    up(&mutex);
    read_db();
    down(&mutex);
    rc--;
    if (rc==0) up (&db);
    up(&mutex);
    use_data();
  }
}
```

```
void writer(void) {
    while(TRUE) {
      prod_data();
      down(&db);
      write_db();
      up(&db);
    }
}
```