Principper for Samtidighed og Styresystemer

Synkronisering og Deadlocks

René Rydhof Hansen

Februar 2008

Skemaændringer

- Forelæsning og øvelser 4. marts (næste uge) er flyttet til 8. maj.
- Forelæsning og øvelser 18. marts er flyttet til 15. maj.

Mål

- At kunne definere begrebet gensidig udelukkelse
- At kunne definere og bruge forskellige metoder til opnåelse af gensidig udelukkelse
- At kunne redegøre for begrebet relativ tid og dets relation til synkronisering
- At kunne forklare de grundlæggende træk ved procesoprettelse under UNIX
- At kunne definere og bruge grundlæggende systemkald til procesoprettelse under UNIX

Opgaver

- Opgave 1: Tråde og gensidighed i Java
- Opgave 2: Tråde og gensidighed i Java
- Opgave 3: Implementation af gensidig udelukkelse
- Opgave 4: Memory mapping

Gensidig udelukkelse — Vigtige begreber

- Race Condition
  - Tilstand hvor resultatet afhænger af den relative hastighed af de enkelte tråde
  - Det vil sige af den faktiske ordning (interleaving) af hændelserne i trådene
  - Vanskeligt at fejlfinde
- Kritisk region
  - Programfragment der giver anledning til race conditions
  - Engelsk: "Critical sections" eller "critical regions"
- Gensidig udelukkelse
  - En tilstand hvor der blandt en mængde tråde kun er en enkelt tråd der kan tilgå en bestemt ressource eller udføre en bestemt del af programteksten
  - Engelsk: "mutual exclusion"
- Atomisk
  - Hændelse eller sekvens af hændelser der sker uafbrydeligt

Kritiske regioner skal udføres under gensidig udelukkelse!

Mutex: lock variabel

```
bool flag[2] = { false, false };

thread P0                          thread P1
{                                  {
  while flag[1]                      while flag[0]
  {                                  {
  }                                  }
  flag[0] = true;                    flag[1] = true;

  /* kritisk region */               /* kritisk region */

  flag[0] = false;                   flag[1] = false;
}                                  }
```

Algoritmen er forkert!

Decker's mutex-algoritme

```
bool flag[2] = {false, false};
int turn = 0;

process P0 {
  flag[0] = true;
  while flag[1] {
    if turn == 1 {
      flag[0] = false;
      while turn == 1 { }
      flag[0] = true;
    }
  }
  /* kritisk region */
  turn = 1;
  flag[0] = false;
}
```

```
process P1 {
   flag[1] = true;
  while flag[0] {
    if turn == 0 {
       flag[1] = false;
     while turn == 0 { }
       flag[1] = true;
    }
  }
   /* kritisk region */
   turn = 0;
   flag[1] = false;
}
```

- Ikke særlig effektiv
- Problem: compiler kan allokere visse variable til registre (dermed ikke delt mellem trådene)

Test and Set

```
int mutex = 0;

thread P0
{
 while(test_and_set(mutex)==1)
 {
 }
 /* kritisk region */
 mutex = 0;
}
```

```
thread P1
{
  while(test_and_set(mutex)==1)
 {
 }
  /* kritisk region */
  mutex = 0;
}
```

- Sæt variabel til 1 og returner tidligere værdi
- `test_and_set` findes ofte som en atomisk processorinstruktion
- Busy waiting

Exchange

- `xchange(a,b)` bytter `a` og `b`
- Hvis `xchange` er atomisk, hvordan kan den så bruges til at sikre gensidig udelukkelse?

Exchange

- `xchange(a,b)` bytter `a` og `b`
- Hvis `xchange` er atomisk, hvordan kan den så bruges til at sikre gensidig udelukkelse?

```
int mutex = 0; /* 0: unlocked; 1: locked */

thread P0 {
  int flag = 1;

  do {
    swap(mutex,flag);
  } while(flag == 1);

  /* Kritisk region */

  mutex = 0;
}
```

Busy Waiting

```
thread P0
{
  while test_and_set(mutex) == 1
  {

  }
  /* Kritisk region */
  mutex = 0;
}
```

- `test_and_set` og `xchange` kræver busy waiting
- En løkke hvor processen aktivt venter på adgang til den kritiske region
- Bruges ofte i kernen til at implementere andre mutex mekanismer
- Problem?
-

Busy Waiting

```
thread P0
{
  while test_and_set(mutex) == 1
  {
    yield;
  }
  /* Kritisk region */
  mutex = 0;
}
```

- `test_and_set` og `xchange` kræver busy waiting
- En løkke hvor processen aktivt venter på adgang til den kritiske region
- Bruges ofte i kernen til at implementere andre mutex mekanismer
- Problem? Potentielt spild af CPU-tid
- Kan forbedres ved at overlade CPU'en til andre tråde
  - Linux: `sched_yield`
  - Grimt!

Blokerende handlinger

- Blokeret tråd
  - Tråd der ikke kører indtil den får et signal fra en anden tråd
  - Speciel tilstand: blokeret
  - Tildeles ikke CPU tid
  - Styresystemet holder regnskab med blokerede tråde og hvorfor de er blokeret
- Blokerende handling
  - En handling der blokerer tråden indtil handlingen er gennemført
  - Systemkald er blokerende handlinger
- Alternativ til busy waiting
  - Bloker tråden indtil der kan opnås adgang til ressourcen
  - Ved brug til mutex: handlingen "frigør ressource/forlad kritisk region" skal medføre mindst een blokeret tråd vækkes

```
thread P0
{
  block_until_access
  /* Kritisk region */
  release_ressource
}
```

Semaforer

- Mutex
  - To tilstande (binær): locked og unlocked
  - `lock`: låser en ulåst mutex; blokerer hvis mutex allerede er låst
  - `unlock`: åbner en ulåst mutex; hvad hvis mutex er ulåst?
- Semafor
  - Ikke-negativ heltalsvariabel med atomare operationer til inkrementering og dekrementering
  - `init(sem,n)`: Initialiserer semaforens værdi til `n`
  - `wait(sem)`: Tæller semaforens værdi ned med en. Blokerer hvis allerede nul.
  - `signal(sem)`: Tæller semaforens værdi op med en. Evt. blokeret tråd vækkes (hvad sker der så med semaforens værdi?)
  - Semafor skal holde styr på `wait`-blokerede tråde
  - I hvilken rækkefølge skal tråde vækkes?
- Mutex: binær semafor
- Semafor: generalisering af mutex

Semaforer i Java

- Initialisering (`n` = initialværdi)

```
sem = New Semaphore(n)
```

- Wait

```
sem.wait();
```

- Signal

```
sem.signal();
```

Mutex med semaforer

```
sem = new Semaphore(1)
thread P0 {
  sem.wait();
  /* Kritisk region */
  sem.signal();
}
thread P1 {
  sem.wait();
  /* Kritisk region */
  sem.signal();
}
thread P2 {
  sem.wait();
  /* Kritisk region */
  sem.signal();
}
```

- Rækkefølgen er udefineret
- Svage semaforer: ingen garanti for at `wait` nogensinde returnerer
- Stærke semaforer garanterer at `wait` før eller siden returnerer (medmindre ingen kalder `signal`).

Synkronisering

```
float T;

thread P0
{
  T = read_sensor();
  ready.signal();
}
```

```
thread P1
{
  ready.wait();
  massive_computation(T);
  output();
}
```

- Hvad kan/skal semaforer bruges til?
-

Synkronisering

```
float T;

thread P0
{
  T = read_sensor();
  ready.signal();
}
```

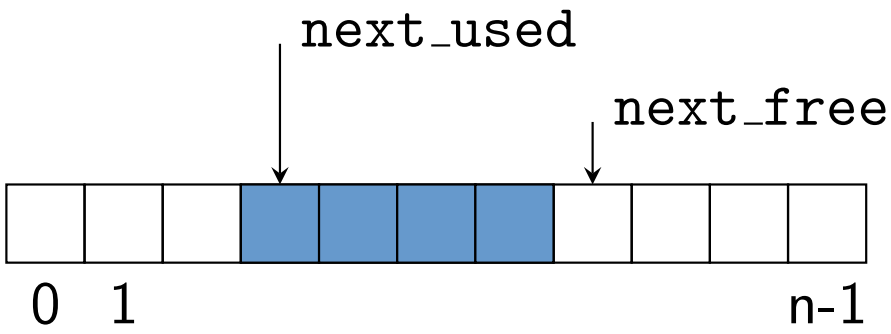
```
thread P1
{
  ready.wait();
  massive_computation(T);
  output();
}
```

- Hvad kan/skal semaforer bruges til?
- Sikre at `read_sensor` sker før `massive_computation`

Bounded buffer and producer/consumer

```
thread INIT
{
 used = new Semaphore(0);
 free = new Semaphore(n);

 buffer = new int[n];
 next_used = 0;
 next_free = 0;
}
```




```
thread PROD
{
 while(true)
 {
  free.wait();

  buffer[next_free] = data;
  next_free = (next_free + 1) % n;

  used.signal();
 }
}
```

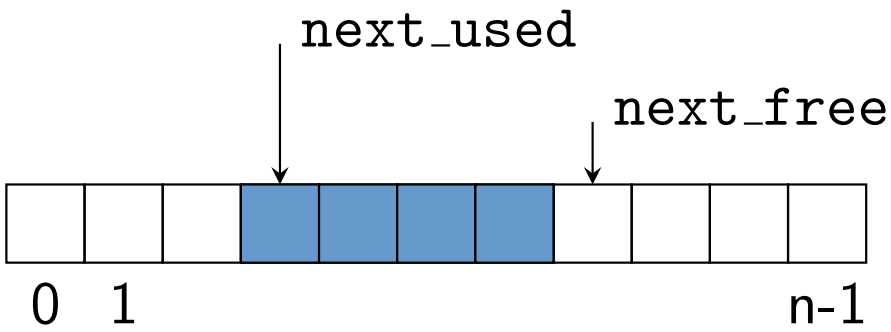
```
thread CONSUMER
{
 while(true)
 {
  used.wait();

  data = buffer[next_used];
  next_used = (next_used +1 ) % n;

  free.signal();
 }
}
```

Bounded buffer and producer/consumer

```
thread INIT
{
 used = new Semaphore(0);
 free = new Semaphore(n);
 mutex = new Semaphore(1);
 buffer = new int[n];
 next_used = 0;
 next_free = 0;
}
```




```
thread PROD
{
 while(true)
 {
  free.wait();
  mutex.wait();
  buffer[next_free] = data;
  next_free = (next_free + 1) % n;
  mutex.signal();
  used.signal();
 }
}
```

```
thread CONSUMER
{
 while(true)
 {
  used.wait();
  mutex.wait();
  data = buffer[next_used];
  next_used = (next_used +1 ) % n;
  mutex.signal();
  free.signal();
 }
}
```

Monitorer

- Semaforer: lav-niveau, let at lave fejl
- Monitor: *sprogkonstruktion* der sikrer gensidig udelukkelse ved at indkapsle
  - Variable (kun tilgås via tilgangsmetoder)
  - Tilgangsmetoder (udføres under gensidig udelukkelse)
  - Initialisering

```
monitor Counter
{
 int i = 0;

 void increment()
 {
  i++;
 }

 ...
}
```

```
void thread()
{
 for(int j = 0; j < 10000;j++)
 {
  Counter.increment();
 }
}

t1 = run(thread);
t2 = run(thread);
wait(t1);
wait(t2);
Counter.print();
```

Monitorer

- Compileren sikrer gensidig udelukkelse på hele monitoren

```
monitor Counter
{
 int i = 0;

 void increment()
 {
  i++;
 }

 void decrement()
 {
  i--;
 }

 ...
}
```

```
void thread()
{
 for(int j = 0; j < 10000;j++)
 {
  Counter.increment();
  Counter.decrement();
 }
}

t1 = run(thread);
t2 = run(thread);
wait(t1);
wait(t2);
Counter.print();
```

Monitors i Java

```
class BoundedBuffer {
 int buffer[], free, used, n;

 BoundedBuffer(int elements) {
  n = elements;
  buffer = new int[n];
  next_used = 0;
  next_free = 0;
  free = n;
  used = 0;
 }
synchronized int get() {
 int data;
 while(used == 0) wait();
 data = buffer[next_used];
 next_used = (next_used + 1) % n;
 free = free + 1;
 used = used - 1;
 notifyAll();
 return data;
}
```

```
synchronized void put(int data) {
 while(free == 0) wait();
 buffer[next_free] = data;
 next_free = (next_free + 1) % n;
 free = free - 1;
 used = used + 1;
 notifyAll();
 }
}
```

- Bemærk, at `while(...) wait()` ikke er busy-waiting: `wait()` returnerer først ved kald af `notify()`/`notifyAll()`
- Monitors i Java er bare en udvidelse af objektbegrebet
- Hvert objekt i Java har en indbygget mutex-lås
- Kald af `synchronized` metoder kræver at tråden opnår denne lås

Monitors i Java

```
void init()
{
 buffer = new BoundedBuffer(n);
}
```

```
void producer()
{
 while(true)
 {
  buffer.put(data);
 }
}
```

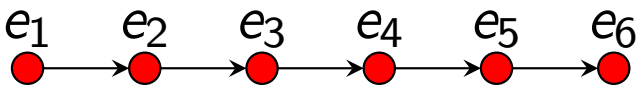
```
void consumer()
{
 while(true)
 {
  data = buffer.get();
 }
}
```

Monitor vs. semafor

- Monitor
  - Sproglig konstruktion til gensidig udelukkelse
  - Indkapser kritiske regioner
  - Garanteres af compileren
- Semafor
  - Kan bruges af programmør til at sikre gensidig udelukkelse
  - Trådprogrammeringens goto
- Hvis strukturerede mekanismer findes bør de foretrækkes (selvom de ikke nødvendigvis er lettere at bruge)

Tidsbegrebet

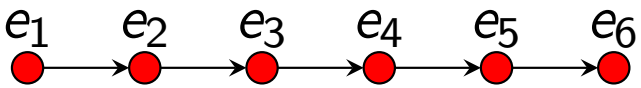
- Hændelser
  - Et punkt i tiden som kan identificeres fordi verdens tilstand har ændret sig
  - Per definition atomiske
  - Eksempel: maskinkodeinstruktioner(?)
- En tråd består af en sekvens af hændelser



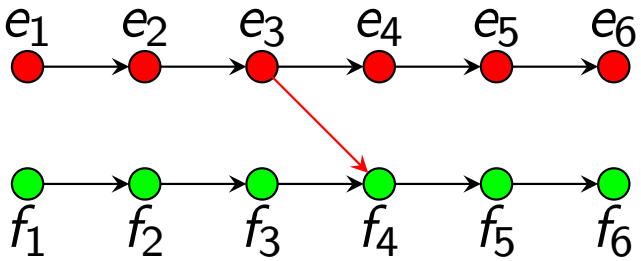
- Lad $T(e)$ være tidspunktet for hændelsen e

Relativ tid: "sker-før"

- $T(e_i) - T(e_j)$ for $i \neq j$ er uforudsigeligt
- Per definition: $T(e_i) < T(e_j)$ for $i < j$
- "Sker-før" relationen
  - Relation mellem hændelser
  - Der gælder at: e_i sker-før e_j medfører $T(e_i) < T(e_j)$
- Relationen er transitiv

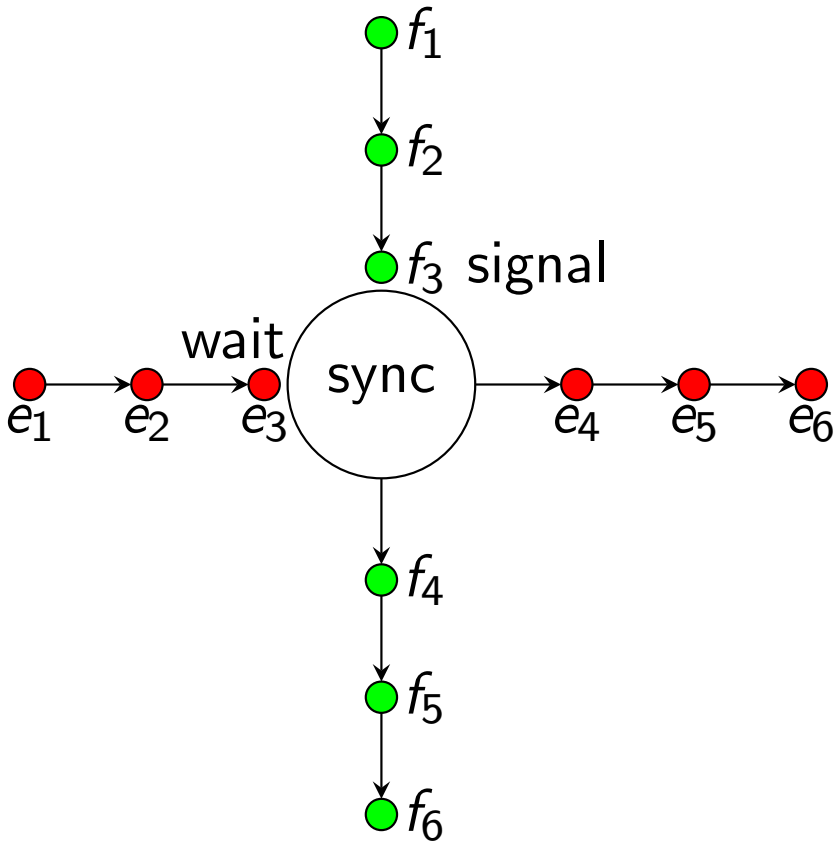


- Der gælder ikke nødvendigvis at $T(e_i) < T(e_j)$ medfører e_i sker-før e_j



Synkronisering

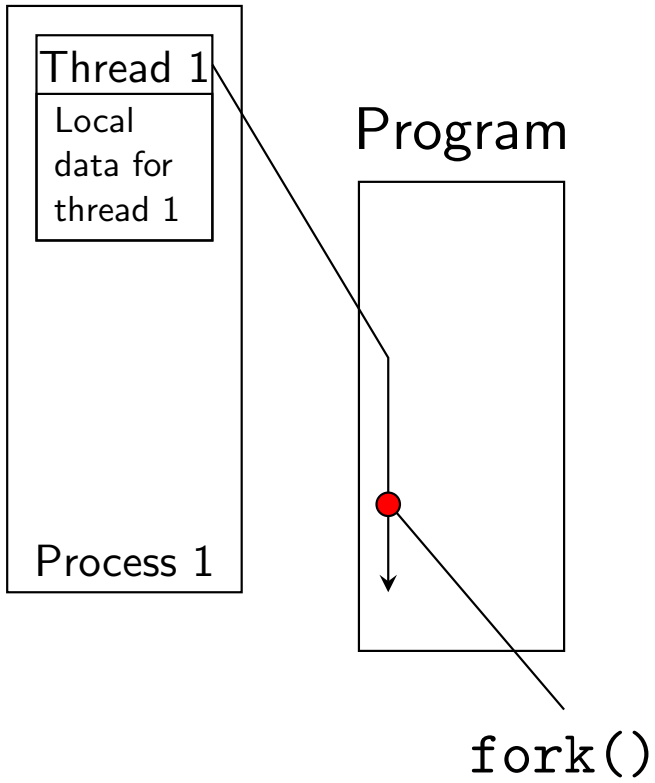
- For at relatere tiden i to tråde skal de synkronisere



Processoprettelse

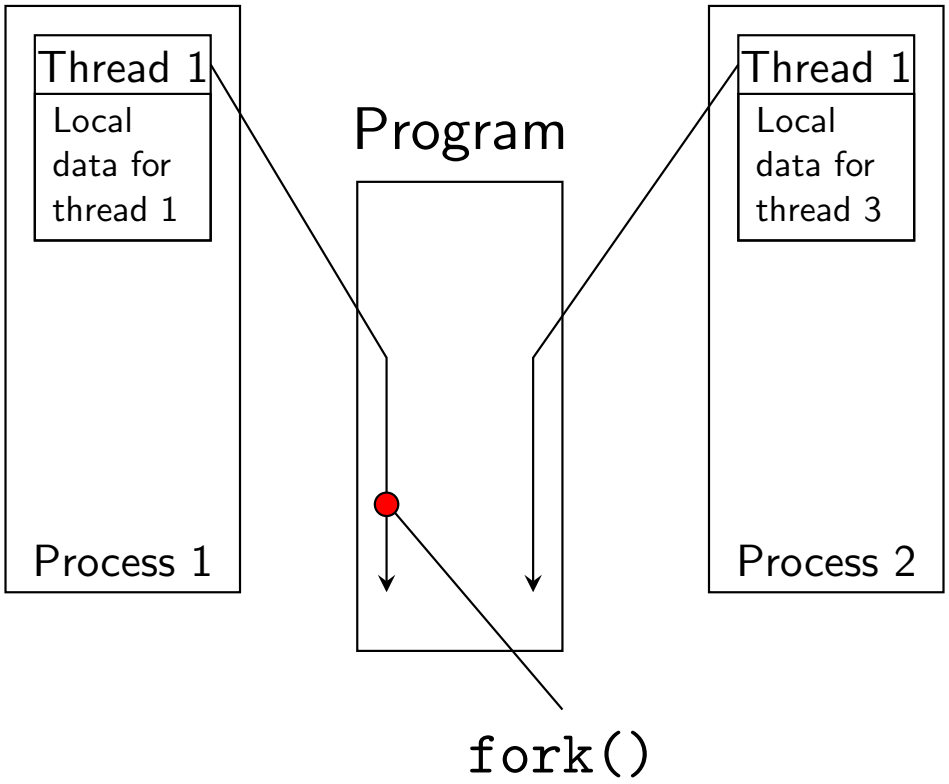
- Processer skaber andre processer
  - Nye processer kaldes børn af den skabende process
  - Første proces skabes ved boot af maskinen (init-processen)
- Forælder-børn relationen giver anledning til at træ
- Når en proces dør bliver den til en zombi
- Forælder processen er ansvalig for at rydde zombier af vejen
- Proces 1 adopterer en død proces' børn
- Init-processen rydder automatisk sine zombier af vejen
- Zombier bruger meget få ressourcer

Skabelse af processer på UNIX



- Nye processer er kloner af deres forælder
- En kopi af forælderens hukommelse
  - Programtekst, stakke, heap
- Process 2 er et barn af process 1
- Eneste forskel: returværdi af `fork()`

Skabelse af processer på UNIX



- Nye processer er kloner af deres forælder
- En kopi af forælderens hukommelse
  - Programtekst, stakke, heap
- Process 2 er et barn af process 1
- Eneste forskel: returværdi af `fork()`

Skabelse af processer på UNIX

- Et barn skabt ved kloning får en kopi af fil-descriptor-tabellen men samme fil-descriptors
- I POSIX genskabes kun den kaldende tråd i barnet
- Det er muligt at indlæse et nyt processbillede oveni den eksisterende process
  - Programteskt, stakke og heap erstattes
  - Alle tråde termineres. Under POSIX skabes en ny.
  - Fil-descriptor-tabellen bevares

Skabelse af processer på Windows

- Nye processer og første tråd skabes med `CreateProcess()`
- Skaber et nyt procesbillede fra en eksekverbar fil

```
BOOL WINAPI CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Systemkald: `fork`

- Skaber en ny process ved at klone den kaldende process
- Returnerer 0 i barnet
- Returnerer process ID i forælder processen
- -1 indikerer fejl

```
void run() {
 pid_t pid = fork();
 switch(pid) {
  case 0:   /* the kid */
   for(int i = 0; i < 1000; i++) {
    printf("%d",i);
   }
   exit(0);
  case -1:
   exit(1);
  default:
   break;
 }
}
run(); run();
wait(NULL); wait(NULL);
```

Systemkald: `wait`

- Venter på at et barn terminerer
  - Faktisk fjerner `wait` et zombie-barn
  - Hvis der ikke er et zombie-barn venter den
- Kan returnere statusinformationer om barnet
- `waitpid()` er mere fleksibel

```
void run() {
 pid_t pid = fork();
 switch(pid) {
  case 0:   /* the kid */
   for(int i = 0; i < 1000; i++) {
    printf("%d",i);
   }
   exit(0);
  case -1:
   exit(1);
  default:
   break;
 }
}
int status;
run();
wait(&status);
```

Systemkald: exec

- Familie af kald der erstatter procesbilledet
- Kombinationen af textttfork() og exec() kan simulere Windows' CreateProcess()

```
void run() {
 pid_t pid = fork();
 switch(pid) {
  case 0:   /* the kid */
   execlp("lp","ls",NULL);
   exit(1);
  case -1:
   exit(1);
  default:
   break;
 }
}
int status;
run();
wait(&status);
```

Systemkald: `system`

- Udfører en shell-kommando
- Ikke et systemkald, men implementeret i systembiblioteket ved hjælp af `fork()` og `exec()`

```
system("ls -l");
```

Opsummering og næste gang

- Gensidig udelukkelse
  - Mutex (lock variable)
  - Semaforer
  - Monitorer
- Processer og tråde
  - Skabelse af processer under UNIX (Windows)
  - Systemkald: `fork()`, `wait()`, `exec()`
  - Systembibliotek: `system()`
- Næste gang: deadlocks og deadlockhåndtering