Agenda

- Från förra föreläsningen
  - C-programmering
  - Labmiljö
- Task synchronization
- Mutex, Semafor
- Asymmetrisk och Symmetrisk synkronisering, Tidssynkronisering
- Lab 2
- Villkorsvariabler/Händelsevariabler

Kort repetition

C-programmering, Makefile

- Syntax för en Makefile:

```
Target [target] : dependency dependency ...
<-TAB!->command
```

- Exempel

```
proj : file-a.c file-b.c file-a.h file-b.h
   gcc –Wall file-a.c file-b.c -o proj
clean:
   rm -rf proj *~ *.o
```

- Target proj bygger projektet. Lägg till nya filer som ska ingå.
- Target clean rensar filer. Tar bort programmet proj samt backupfiler och objektfiler (*.o).

C-programmering, struct

- Struct grupperar sådant som hör ihop till en abstrakt enhet
- Clock är här statiskt allokerad
  - Finns allokerat plats för denna struktur i minnet från början

```
struct
{
   int hours;
   int minutes;
   char alarm_text[20];
} clock;

clock.hours = 11;
clock.minutes = 30;
clock.alarm_text[] = ”lunch time”;
```

C-programmering, struct och pekare

```
typedef struct
{
   int hours;
   int minutes;
   char alarm_text[20];
} clock_data_type;

typedef clock_data_type* clock_type;

clock_type clock;

clock = (clock_type)

malloc(sizeof(clock_data_type));

clock_set(clock);
```

```
void clock_set(clock_type c)
{
   c->hours = 11;
   c->minutes = 30;
   c->alarm_text[] =
          ”lunch time”;
};
```

- Fält i struct refereras nu till via operator ->
- Clock är dynamiskt allokerad
  - Plats skapas vid anrop till malloc

C programmering, static variabler

```
typedef struct
{
   int hours;
   int minutes;
   char alarm_text[20];
} clock_data_type;

static clock_data_type clock;

void clock_set(void) {
   clock.hours = 11;
   clock.minutes = 30;
   clock.alarm_text[] =
        ”lunch time”;
};
```

- Här blir clock av type static, vilket medför att den är endast tillgänglig/synlig inom samma fil
- Man tvingar därmed kod som direkt ska påverka variabeln clock att vara i samma fil, och får då bra modularisering.

C-programmering resurser

- Generell info
  http://www.isy.liu.se/edu/kurs/TSEA81/c_resources.html
- Pekare och struktar
  https://www.tutorialspoint.com/cprogramming/index.htm

Gitlab: Repositorie för git

- https://gitlab.liu.se
- Git är bra för att:
  - Hantera versioner av program/dokument
  - Arbeta med samma program/dokument på flera system
  - Samarbeta med andra i samma projekt
- Använd inte github.com eller liknande publika repositories
- Kort enkel GIT-manual:
  http://rogerdudler.github.io/git-guide/

Att arbeta på distans, bra linux-kommandon

Att arbeta hemifrån/från annan lokal på LiU

- Mer information om hur man kan logga in hemifrån på liunet student
  - https://liuonline.sharepoint.com/sites/student-under-studietiden/SitePages/Fjarrinloggning.aspx
  - VPN krävs om datorn inte befinner sig på universitetet
- Thinlinc rekommenderas framför direkt ssh från egen maskin
- Måste ansluta vidare till en av maskinerna i MUX1 eller MUX2 labben från thinlinc
  - Logga in till MUX-labb utifrån (från thinlinc eller maskin i annat lab):

    ssh -XC liuid@muxenX.-0YY.ad.liu.se X={1,2}, YY={01..16}
    - Observera! Stort X och stort C för komprimering (mindre mängd data)

Flera användare samtidigt på samma maskin

- Kommunikation mellan GUI (SimpleOS) och era program sker via nätverksport
  - Om annan användare redan använder porten kommer kommunikationen ske med den användarens GUI
    - T ex om flera loggar in på samma maskin i mux-labbet och startar GUISimpleOS
  - Behöver då byta port som används
    - Förslag: använd port nummer 2000+liuidnr (t ex linea123 använder port 2123)
    - Sätt miljövariabeln SIMPLE_OS_PORT
      - Se även https://www.isy.liu.se/edu/kurs/TSEA81/Simple_OS/https://www.isy.liu.se/edu/kurs/TSEA81/Simple_OS/

Bra linuxkommandon

man man	manualsida över manualsidor
man printf	kommandot printf
man 3 printf	C-funktionen printf
man –k sökord	hitta relevant info för sökord
apropos sökord	samma sak som man –k
/	sök i man-sida
n	nästa förekomst av sökord
Shift-n	föregående förekomst av sökord
q	avsluta (gå ur man-sida)
ls	lista kataloginnehåll
grep	hitta innehåll, i tex filer
grep –r ”^.*//” *.c	hitta alla rader med kommentar, rekursivt
man 7 regex	beskriver reguljära uttyck

Spinlock och atomiska operationer

Task Synchronization: Test and set (TAS)

- Exempel på assemblerinstruktioner (68000) för synkronisering
- TAS:
  - Läs given minnesadress
  - Sätt flaggor enl. innehållet
  - Skriv värdet 1 till minnesadress

  } Sker atomiskt

- Användning

```
Loop:
   TAS (a0)  ; Test and Set, update Z flag
   BNE loop  ; wait if set at TAS
   CRITICAL REGION
   CLR.B (a0)   ; release lock
```

} spinlock

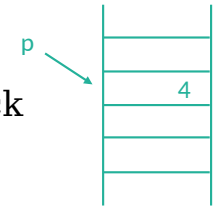
Task Synchronization: Compare-and-Exchange (CMPXCHG) på x86

```
function cas(p:pointer, old:int, new:int) {
  if *p ≠ old {    ; if value has changed
    return false}  ; return false
  *p = new         ; otherwise set value to new
    return true    ; return true
}

function add(p:pointer, a:int) {
  done=false
  while not done {
    value=*p
    done = cas(p, value, value+a)
  }
return value+a
```

Pseudokod
Antas vara atomisk

spinlock



Exempel:
Försök öka värdet på en variabel

Task Synchronization

- Angående spinlocks:
  - En vanlig variant är att loopa endast om den task som håller låset faktiskt kör för tillfället (i en annan processor/kärna). Annars ber vi operativsystemet väcka oss när denna task släpper låset.
- Angående atomiska operationer:
  - Även C++ har stöd för atomiska operationer (std::atomic), dvs man måste inte programmera på assemblernivå

Mutex, Semafor

Mutex i Linux

- En mutex är bara låst/upplåst.

```
// deklarera en mutex
pthread_mutex_t Mutex;

// initiera med attribut, vanligen NULL (standardattribut)
pthread_mutex_init(&Mutex, mutex_attr);

// lock mutex
pthread_mutex_lock(&Mutex);      // count==1 => count=0
                                 // count==0 => vänta på unlock
// unlock mutex
pthread_mutex_unlock(&Mutex);    // ingen väntar => count=1
                 // minst en väntar => väck enn task som väntar
```

Semaforer i Linux

- En semafor är en generaliserad Mutex
- En semafor har ett värde, istället för bara låst/upplåst

```
// deklarera semaforen
sem_t Semaphore;

// initiera till ett visst startvärde
sem_init(&Semaphore, 0, init_value);

//wait
sem_wait(&Semaphore);    // count>0  => count--
                         // count==0 => vänta tills nån gör signal

// signal
sem_post(&Semaphore);    // ingen väntar => count++
                // minst en väntar => väck enn task som väntar
```


Mutual vs Semafor

Mutex	Semafor
Endast tråden som kört lock får köra unlock	Olika trådar får köra wait respektive post

- En semafor *kan* användas till mutual exclusion, men är tänkt för att skicka information, om att en viss händelse har inträffat, från en tråd (eller process) till en annan
- Dvs en semafor används huvudsakligen till asymmetrisk eller symmetrisk synkronisering, vilket inte går att göra med en mutex
- I Simple-OS finns dock enbart semaforer

Mutex vs Semafor

```
// task 1
  for(i=0; i <= 9; i++) {
    array[i] = i;
  }
  sem_post(&Sem);
  // Roterar array[]
  while(1){
    pthread_mutex_lock(&Mutex);
    tmp=array[0];

    for(i=0; i <= 8 ; i++){
      array[i]=array[i+1];
    }
    array[9] = tmp;
    pthread_mutex_unlock(&Mutex);
  }
```

```
// task 2
   sem_wait(&Sem);
   // Summera array[]
   while(1){
     pthread_mutex_lock(&Mutex);
     sum=0;
     for(i=0; i <= 9; i++){
       sum += array[i];
     }

     pthread_mutex_unlock(&Mutex);
     printf("Summa: %d\n", sum);
     usleep(1000000);
   }
```

- Semafor kan användas mellan tasks. Mutex används inom tasks.

Asymmetrisk och Symmetrisk synkronisering, Tidssynkronisering

Asymmetrisk synkronisering

- En en-vägs synkronisering
- En task P1 kan informera en annan task P2 om att den (P2) kan fortsätta med sitt arbete
- Kan implementeras med en semafor, där P2 gör en wait-operation och P1 gör en signal-operation

Signal flera ggr -> signallerar flera händelser (count++)

Semaforen initieras till 0 vid asymmetrisk synkronisering

Asymmetrisk synkronisering

```
while(1) {                  // Vill sampla io varje ms
  sample_io_pins();
  work_sample_data();
  usleep(1000);
}
```

- Två tänkbara problem:
  - work_sample_data tar för lång tid att utföra, dvs sample_io_pins körs inte tillräckligt ofta (dvs varje millisekund)
  - work_sample_data tar olika lång tid var gång, dvs sample_io_pins körs inte med jämn sampel-takt

Asymmetrisk synkronisering

- Möjlig lösning: två trådar
- Hög prio: sample_io_pins

```
// sample_thread, high prio
while(1) {
  sample_io_pins();
  sem_post(&Sample_sem);
  usleep(1000);
}
```

- Låg prio: work_sample_data

```
// work_thread, low prio
while(1) {
  sem_wait(&Sample_sem);
  work_sample_data();
}
```

- Är problemet löst?
  - Ja, om körtiden för sample_io_pins och sem_post är försumbar.
    Annars använd funktioner för att sova till en viss tidpunkt.

Asymmetrisk synkronisering och avbrott

- Ibland är det nödvändigt för en task att vänta på en extern händelse.
- T ex kan en task vänta på att en viss tid ska förlöpa, genom att ett tidsavbrott inträffat ett visst antal gånger (”timer interrupt occurred N times”). När det inträffar ska task:en göras redo att köra igen.
- Asymmetrisk synkronisering kan användas mellan en avbrottshanterare och en task. Tasken:en gör wait och avbrottshanteraren gör signal.

```
ISR : {  //avbrottsrutin
  n++;
  if (n == 1000) {
    sem_post(&data_ready);
    n=0;
  }
}
```

```
void process_data {
  while(1) {
    sem_wait(&data_ready);
    // data processing
    …
  }
}
```

Asymmetrisk synkronisering och avbrott

- OBSERVERA: Man måste vara försiktig med att använda semaforer i avbrott. Avbrott kan komma när som helst och det är ju inte säkert att den task som avbryts har något med semaforen att göra. Dvs den task som avbryts kan komma att flyttas till väntelistan (vid wait).

```
ISR : {
  …
  sem_wait(&data_ready);
  …
}
```

```
task1 : {
  …
  sem_post(&data_ready);
  …
}
```

```
task2 : {
  …
  // not related to
  // semaphore
  // data_ready
  …
}
```

- Observera! ISR är INTE en task! Med otur riskerar task2 att hamna i väntelistan för semaforen data_ready.

Asymmetrisk synkronisering och avbrott

- Asymmetrisk synkronisering + Avbrott : En mycket vanlig teknik.
- De flesta program väntar på att ett avbrott ska inträffa, ibland i flera led.
  - T ex, kommandoprompten väntar indata från terminalprogrammet, som väntar på indata från X-servern, som väntar indata från tangentbordet.
- Intressant variant, Linux NAPI (New API) interrupt mitigation:

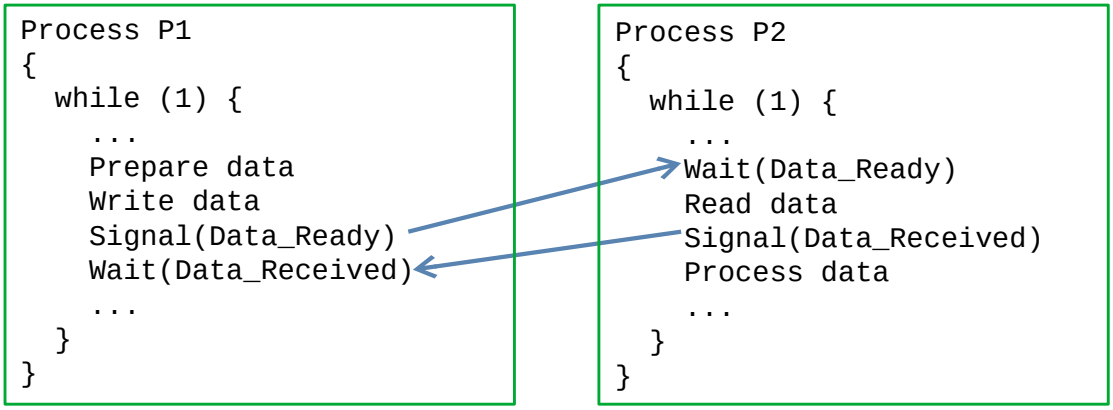
  Interrupt vs polling, beroende på belastning/vikt.

  Uppsala universitet först med NAPI-baserat OS.

  Högfrekventa nätverksavbrott (ett avbrott per paket) kan bli kostsamt och sänka systemet (Jfr DOS-attack). Genom att istället polla (kolla när man har tid) kan flera buffrade paket hanteras samtidigt.

Symmetrisk synkronisering

- Processer väntar på varandra, t ex vid dataöverföring



Tidssynkronisering

- I de två första laborationerna finns implementationer av en klocka.
- Observera följande:
  - Ett anrop av usleep() får en anropande task att vänta en tid relativ till tidpunkten för anropet. Den verkliga klock-perioden blir längre än argumentet till usleep(), då även övriga saker i loopen tar tid.
  - Timingen kan förbättras genom att istället använda clock_nanosleep() med flaggan TIMER_ABSTIME vilket får anropande task att vänta till en absolut tidpunkt. [Nästa slide]
  - Timing kan också förbättras genom att använda en högprioriterad task (tick_task) vars uppgift bara är att räkna upp ett tidskvanta och via assymmetrisk synkronisering tala om för en annan task (clock_task), som har fler uppgifter, att arbeta igen.

Tidssynkronisering : clock_nanosleep

```
struct timespec {
  time_t tv_sec;  /* seconds */
  long tv_nsec;   /* nanoseconds */
};
```

```
#include <time.h>
struct timespec ts; // time struct

clock_gettime(CLOCK_MONOTONIC, &ts); // get current time

while(1) {

  ts.tv_nsec += delay; // add delay to point in time

  if (ts.tv_nsec >= 1000*1000*1000) {
      ts.tv_nsec -= 1000*1000*1000;
      ts.tv_sec++;
  }

  // wait until absolute point in time
  clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &ts, NULL);
  . . .
```


Laborationsuppgift 2

Lab 2 : Alarm Clock

- Bygg en klocka med alarm, innehållande tre processer/trådar:

  Clock : Sköter själva klocktiden, med bästa möjliga precision
  Alarm : Ger alarm (vid rätt tidpunkt) tills användaren bekräftar
  Set : Sköter inställning av klocktid och alarmtid

- Implementationskrav finns på kurshemsidan:
  http://www.isy.liu.se/edu/kurs/TSEA81/assignment_alarm_clock.html

Lab 2

- Att tänka på vid design och implementation:
  - Tänk på vad som är huvuduppgiften för varje task, dvs vad som ska hända inne i while-loopen. [Nästa slide]
  - Tänk på vad som är gemensamma resurser, hur varje task använder dessa och hur de kan skyddas på ett effektivt minimalt sätt.
  - Det är en skillnad mellan alarm enabled (alarmtiden är satt) och alarm activated (”klockan ringer”).
  - Den process som har till uppgift att repetera alarmet (”klockan ringer”) får **INTE** konsumera någon processorkraft när larmet inte är aktiverat. T ex genom att kontinuerligt jämföra klocktid och alarmtid. Här **SKA** istället assymmetrisk synkronisering användas.
  - Klockan **SKA** räknas upp med bästa möjliga precision (clock_nanosleep)

Bygga och starta trådar

```
#include <pthread.h>


void *task1_thread(void *unused) {
  // local initialization
  …

  while(1) {
    // task main loop
    …

  }
}


void *task2_thread(…)

…
```

```
int main(void) {
  pthread_t task1_thread_handle;
  pthread_t task2_thread_handle;
  …

  pthread_create(&task1_thread_handle,
                 NULL,
                 task1_thread,
                 0);
  pthread_create(…);
  …

  pthread_join(task1_thread_handle,

               NULL);
  pthread_join(…);
  …

  while(1);
  // will never be here
}
```


Måste kompileras och länkas med biblioteket pthread i GCC : ”gcc –lpthread” [apropos pthread]

Lab 2: set_clock med moduler

set_clock program

```
//set_clock.c
#include ”clock.h”
#include ”display.h”
#include ”si_ui.h”
...
void main(
{
  clock_set(12,
  ...
```

clock module

```
//clock.h
//clock.c
#include ”clock.h”
...
static Clock

void clock_set(int hours,
{
  pthread_mutex_lock(&M,
  Clock.hours = hours;
  ...
```

display module

```
//display.h
//display.c
#include ”display.h”
```

Makefile

```
#Makefile
all: set_clock
set_clock: set_clock.c ...
     gcc set_clock.c ...
Clean:
     rm –f set_clock ...
```

ui module

```
//si_ui.h
//si_ui.c
#include ”si_ui.h”
```

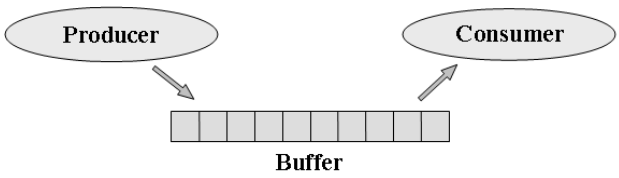
comm module

```
//si_comm.h
//si_comm.c
#include ”si_comm.h”
```

Villkorsvariabler/Händelsevariabler

Producer and Consumer

- En producent och en konsument samt en buffer kan användas för kommunikation:



Typiska krav:

- Producenten ska vänta om buffern är full
- Konsumenten ska vänta om buffern är tom

Producer and Consumer

- Notering: Accessen till den kritiska regionen styrs av ett villkor, och bildar en s k villkorlig kritisk region.
- Notering: Använder en cirkulär buffer. Cirkulära buffrar används i alla typer av FIFOs, sampling, tidsdiskret faltning m m

```
/* store item in buffer */
void put_item(char item) {
  pthread_mutex_lock(&Mutex);
  while (buffer is full) {
    wait until buffer is not full
  }
  Buffer_Data[In_Pos] = item;
  In_Pos++;
  if (In_Pos == BUFFER_SIZE) {
    In_Pos = 0;
  }
  count++;
  wake up processes that are
  waiting for the buffer to become
  non-empty/non-full
  pthread_mutex_unlock(&Mutex);
}
```


Producer and Consumer

- Observera att while används, INTE if. Villkoret måste kontrolleras på nytt när processen blir körande igen.
- Skälet till att processen blir väckt kan ju vara både att buffern är icke-tom eller icke-full.
- En process kan dessutom väckas utan att signal skett (spurious wakeups).

```
/* read item from buffer */
void get_item(void) {
  char item;
  pthread_mutex_lock(&Mutex);
  while (buffer is empty) {
    wait until buffer is not empty}
  item = Buffer_Data[Out_Pos];
  Out_Pos++;
  if (Out_Pos == BUFFER_SIZE) {
    Out_Pos = 0;
  }
  count--;
  wake up processes that are waiting for
  the buffer to become non-empty/non-full
  pthread_mutex_unlock(&Mutex);
  return item;
}
```

Conditional critical regions

- Villkorligt kritiska regioner är kritiska regioner associerade med villkor.
- Krav för villkorligt kritiska regioner:
  - Det måste vara möjligt för en task att vänta (wait) för ett givet villkor
  - Det måste finnas en mekanism för att aktivera denna väntan, på så sätt task:en på nytt kan utvärdera villkoret för att avgöra om den får gå in i sin kritiska region.

Conditional variables (CV)

- Händelsevariabler (condition variables):
- Kan användas tillsammans med en Mutex för att implementera villkorligt kritiska regioner (I Simple-OS tillsammans med semafor)
- Tre operationer:
  - Init : initiering, associering med Mutex/Semafor
  - Await : vänta på händelsevariabeln, anropas med semaforen låst låser upp semaforen, placerar task:en i CV:s väntelista, när Await returnerar kommer semaforen åter att vara låst
  - Cause : Signalera till alla som väntar på händelsevariabeln (alla task:s i CV:s väntelista), när en av dessa task:s aktiveras låses semaforen automatiskt igen
- En händelsevariabel ska alltid vara *associerad* med en Mutex/Semafor

Producer and Consumer + CV

Await : pthread_cond_wait

Cause : pthread_cond_broadcast

```
/* store item in buffer */
void put_item(char item) {
  pthread_mutex_lock(&Mutex);
  while (count == BUFFER_SIZE) {
    pthread_cond_wait(&Change, &Mutex);
  }
  Buffer_Data[In_Pos] = item;
  In_Pos++;
  if (In_Pos == BUFFER_SIZE) {
    In_Pos = 0;
  }
  count++;
  pthread_cond_broadcast(&Change);
  pthread_mutex_unlock(&Mutex);
}
```

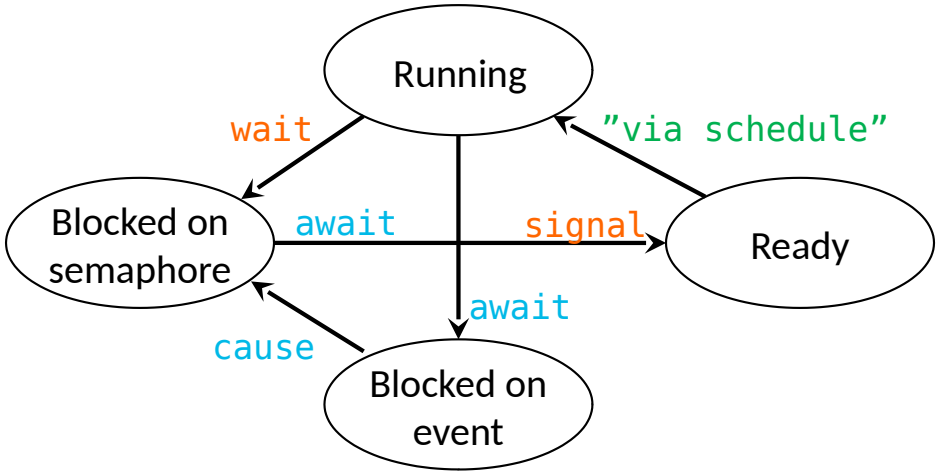

Producer and Consumer + CV

Await : pthread_cond_wait

Cause : pthread_cond_broadcast

```
/* read item from buffer */
void get_item(void) {
  char item;
  pthread_mutex_lock(&Mutex);
  while (count == 0) {
    pthread_cond_wait(&Change, &Mutex);
  }
  item = Buffer_Data[Out_Pos];
  Out_Pos++;
  if (Out_Pos == BUFFER_SIZE) {
    Out_Pos = 0;
  }
  count--;
  pthread_cond_broadcast(&Change);
  pthread_mutex_unlock(&Mutex);
  return item;
}
```

Producer and Consumer + CV



await (”pthread_cond_wait”) i Simple-OS

```
void si_ev_await(si_event *ev)                // await operation on ev
{
  int pid;
  DISABLE_INTERRUPTS;                         // atomic section begins
  if (!wait_list_is_empty(                       // check if processes
    ev->mutex->wait_list, WAIT_LIST_SIZE))    //   are waiting on the mutex
  {
    pid = wait_list_remove_highest_prio(      // get pid with
      ev->mutex->wait_list, WAIT_LIST_SIZE);  //   highest priority
    ready_list_insert(pid);                   // make this process ready to run
  } else {
    ev->mutex->counter++;                     // increment counter
  }
  pid = process_get_pid_running();            // get pid of running process
  ready_list_remove(pid);                     // remove it from ready list
  wait_list_insert(                           // insert it into the
    ev->wait_list, WAIT_LIST_SIZE, pid);      //   event waiting list
  schedule();                                 // call schedule
  ENABLE_INTERRUPTS;                          // atomic section ends
}
```

cause (”pthread_cond_signal”) i Simple-OS

```
void si_ev_cause(si_event *ev)                    // cause operation on ev
{
  int done;
  int pid;
  DISABLE_INTERRUPTS;                             // atomic section begins
  done = wait_list_is_empty(                      // we are done if the
    ev->wait_list, WAIT_LIST_SIZE);               //   wait list is empty
  while (!done)
  {
    pid = wait_list_remove_one(                   // remove one process from the
      ev->wait_list, WAIT_LIST_SIZE);             //   list of waiting processes
    wait_list_insert(                             // insert it into the
      ev->mutex->wait_list, WAIT_LIST_SIZE, pid); //   mutex waiting list
    done = wait_list_is_empty(                    // check if we
      ev->wait_list, WAIT_LIST_SIZE);             //   are done
  }
  ENABLE_INTERRUPTS;                              // atomic section ends
}
```


Semafor: wait (”lock”) i Simple-OS

```
void si_sem_wait(si_semaphore *sem)               // wait operation on semaphore sem
{
  int pid;                                        // process id
  DISABLE_INTERRUPTS;                             // atomic section begins

  if (sem->counter > 0) {                         // check counter
    sem->counter--;                               // decrement
  } else {
    pid = process_get_pid_running();              // get pid of running process
    ready_list_remove(pid);                       // remove it from ready list
    wait_list_insert(                             // insert it into the
      sem->wait_list, WAIT_LIST_SIZE, pid);       //   semaphore waiting list
    schedule();                                   // call schedule
  }
  ENABLE_INTERRUPTS;                              // atomic section ends
}
```

Semafor: signal (”unlock”) i Simple-OS

```
void si_sem_signal(si_semaphore *sem)       // signal operation on semaphore sem
{
  int pid;                                  // process id
  DISABLE_INTERRUPTS;                       // atomic section begins
  if (!wait_list_is_empty(                  // check if processes
    sem->wait_list, WAIT_LIST_SIZE))        //   are waiting on semaphore
  {
    pid = wait_list_remove_highest_prio(    // get pid with
      sem->wait_list, WAIT_LIST_SIZE);      //   highest priority
    ready_list_insert(pid);                 // make this process ready to run
    schedule();                             // call schedule
  } else {
    sem->counter++;                         // increment counter
  }
  ENABLE_INTERRUPTS;                        // atomic section ends
}
```


LINKÖPINGS UNIVERSITET