

Datorteknik och Realtidssystem

Fö4

Monitors and Message passing

Agenda

- Reflektion Lab1
 - Kort om Lab2, moduler
 - Händelsevariabler
 - Structar och dynamisk minnesallokering
 - Abstraktion, Monitor
 - Barriär med händelsevariabler
 - Trådar vs Processer
 - Message Passing (meddelandekommunikation)
 - Lab3 och Lab4
-

Reflektion Lab 1

Lab1 : odd_even

```
int main(void)
{
    ...
    pthread_create(&disp_thread_handle, NULL, display_thread,0);
    pthread_create(&change_thread_handle, NULL, change_thread,0);
    pthread_create(&exit_thread_handle, NULL, exit_thread,0);

    pthread_join(disp_thread_handle, NULL);
    pthread_join(change_thread_handle, NULL);
    pthread_join(exit_thread_handle, NULL);

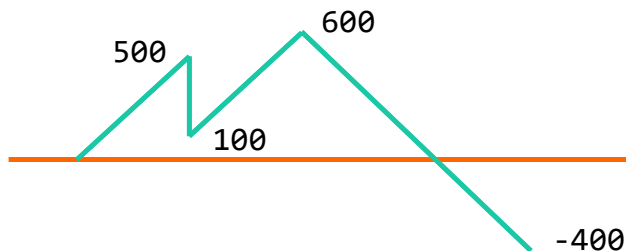
    return 0;
}
```

display_thread

```
...
set_number(100);
sem_post(&Sem);
while(1)
{
    ...
}
```

change_thread

```
...
sem_wait(&Sem);
while(1)
{
    for (i=0;i<1000;i++)
        increment_number();
    for (i=0;i<1000;i++)
        decrement_number();
}
```



Även om trådarna startar i en viss ordning, så garanterar inte det att de hinner utföra allt före while-satsen innan nästa tråd börjar arbeta.

Lösningen är **asymmetrisk synkronisering**.

Lab 2, moduler

Lab2 : 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"
```

Händelsevariabler

Condition variables - händelsevariabler

Kan användas för att implementera villkorliga kritiska regioner

Tre operationer:

- *Initiering*(C, S) – interna data för villkorsvariabeln C initieras, och denna associeras till semaforen S
- *Await*(C) – anropande process blir väntande (placeras i väntekö) på C , och en Signal-operation görs på S
- *Cause* – processer som väntar på C , flyttas till att istället vänta på S

Initiering och användning av semafor med händelsevariabel

Initiering:

```
pthread_mutex_t mutex;  
pthread_cond_t change;  
...  
// I main():  
pthread_mutex_init(&mutex, NULL);  
pthread_cond_init(&change, NULL);  
...
```

I Simple-OS associeras händelsevariabeln med semaforen vid initieringen: `si_cv_init(&change, &Sem)`

Användning:

```
// I process:  
pthread_mutex_lock(&mutex);  
...  
pthread_cond_wait(&change, &mutex);  
...  
pthread_cond_broadcast(&change);  
...  
pthread_mutex_unlock(&mutex);
```

I Linux associeras händelsevariabeln med mutexen vid användningen: `pthread_cond_wait(&change, &Mutex)`

Initiering och användning av semafor med händelsevariabel

Hur associeras händelsevariabeln med semaforen?

- I Linux:
`pthread_cond_init(&change, NULL);`
`pthread_cond_wait(&change, &mutex);`
- I simple OS:
`si_ev_init(&change, &mutex);`
`si_ev_await(&change);`

Structar och dynamisk minnesallokering

Struct

```
struct
{
    int first;
    int second;
} variable;
```

```
void main(void)
{
    variable.first = 1;
    variable.second = 2;
    printf("The struct variables are %d and %d\n",
        variable.first, variable.second);
}
```

Struct type

```
struct struct_type
{
    int first;
    int second;
};
```

```
void print_struct_type_var(struct struct_type var)
{
    printf("The struct variables are %d and %d\n",
        var.first, var.second);
}

void print_struct_type_var_pointer(struct struct_type *var)
{
    printf("The struct variables are %d and %d\n",
        var->first, var->second);
}

void main(void)
{
    struct struct_type variable;
    variable.first = 3;
    variable.second = 4;
    print_struct_type_var(variable); // call by value
    print_struct_type_var_pointer(&variable); // call by reference
}
```

Struct typedef

```
typedef struct
{
    int first;
    int second;
} typedef_struct_type;
```

```
void print_typedef_struct_type_var(typedef_struct_type var)
{
    printf("The struct variables are %d and %d\n",
        var.first, var.second);
}

void print_typedef_struct_type_var_pointer(
    typedef_struct_type *var) {
    printf("The struct variables are %d and %d\n",
        var->first, var->second);
}

void main(void)
{
    typedef_struct_type variable;
    variable.first = 5;
    variable.second = 6;
    print_typedef_struct_type_var(variable);
    print_typedef_struct_type_var_pointer(&variable);
}
```

Struct typedef med dynamisk allokering

```
typedef struct
{
    int first;
    int second;
} typedef_struct_type;
```

```
typedef typedef_struct_type* str_p_type;

str_p_type init(void) {
    str_p_type var =
        (str_p_type)malloc(sizeof(typedef_struct_type));
    var->first = 7;
    var->second = 8;
    return var;
}

void print_typedef_struct_type_var_pointer(str_p_type var) {
    printf("The struct variables are %d and %d\n",
        var->first, var->second);
}

void main(void) {
    str_p_type variable;
    variable = init();
    print_typedef_struct_type_var_pointer(variable);
}
```

Statisk kontra dynamisk minnesallokering

Statisk	Dynamisk
Alla datastrukturer behöver allokeras som globala variabler	Datastrukturer allokeras med new och/eller malloc
Fördel: Total minnesåtgång är lättberäknad	Fördel: Mer flexibel

Annat värt att beakta:

- Anropsstacken kräver också minnesutrymme
- Dvs rekursiva processer är inte att rekommendera

Abstraktion, Monitor

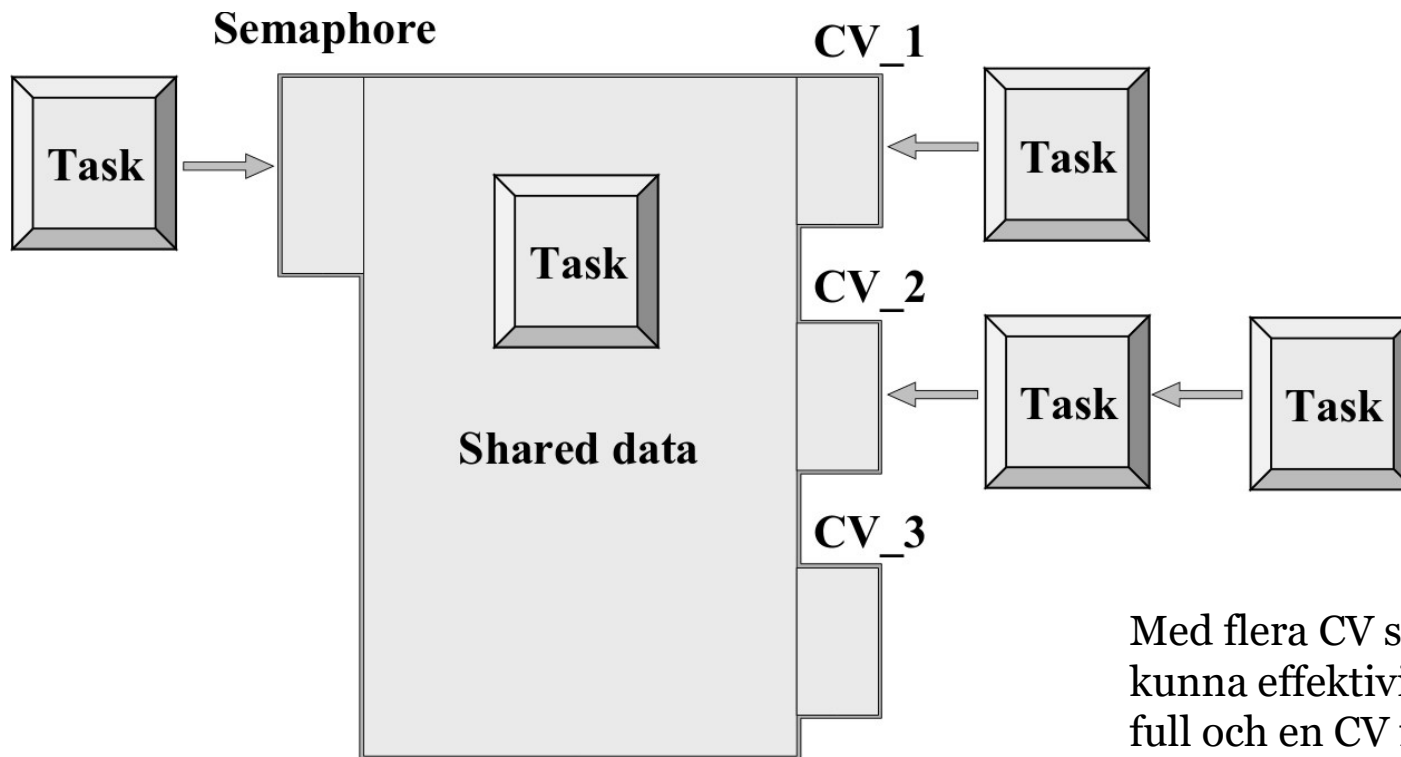
Data abstraction

- Vi kan skapa strukturerade datatyper genom att använda en *struct*
 - Vi kan skapa *abstrakta* datatyper, genom att kräva att all access till datatypen sker via ett urval av funktioner, t ex pekare, punktnotation
-
- I förra föreläsningen skapade vi *nästan* en sk monitor
 - Enligt gängse terminologi krävs en högre abstraktionsnivå för att det ska kallas en monitor

Monitor

- En *monitor* är en abstrakt datatyp med datafält för delade resurser till ett program med processer, och semaforer för att skydda dessa resurser samt noll eller flera händelsevariabler för att kontrollera accessen av resurser i villkorliga kritiska regioner.

Visualisering av en Monitor



Med flera CV skulle `put_item()` / `get_item()` kunna effektiviseras, t ex en CV för buffer full och en CV för buffer tom

Producent/konsument-exemplet

```
typedef struct {  
    char buffer_data[BUFFER_SIZE];  
    int in_pos; // skrivposition  
    int out_pos; // läsposition  
    int count; // antal element i buffern  
  
    pthread_mutex_t mutex; // semafor  
    pthread_cond_t cv_1; // händelsevariabel 1  
    pthread_cond_t cv_2; // händelsevariabel 2  
} buffer_data_type;  
  
typedef buffer_data_type *buffer_type;
```

Producent/konsument-exemplet

```
// skapa buffer och initiera parametrar
buffer_type create_buffer(void) {
    // skapa buffer
    buffer_type buffer;
    buffer = (buffer_type) malloc(sizeof(buffer_data_type));
    // initiera buffervariabler
    buffer->in_pos = 0;
    buffer->out_pos = 0;
    buffer->count = 0;
    // initiera mutex och händelsevariabler
    pthread_mutex_init(&buffer->mutex, NULL);
    pthread_cond_init(&buffer->cv_1, NULL);
    pthread_cond_init(&buffer->cv_2, NULL);

    return buffer;
}
```

Producent/konsument-exemplet

```
// buffern!  
buffer_type b;  
  
b = create_buffer();
```

put_item() och get_item() modifieras på motsvarande sätt

Producent/konsument-exemplet

```
void put_item(char item) {  
    pthread_mutex_lock(&b->mutex);  
    while (b->count == BUFFER_SIZE) {  
        pthread_cond_wait(&b->cv_1, &b->mutex);  
    }  
    b->buffer_data[b->in_pos] = item;  
    b->in_pos++;  
    if (b->in_pos == BUFFER_SIZE)  
        b->in_pos = 0;  
    b->count++;  
    pthread_cond_broadcast(&b->cv_2);  
    pthread_mutex_unlock(&b->mutex);  
}
```

cv_1 : "icke full"

cv_2 : "icke tom"

Producent/konsument-exemplet

```
char get_item(void) {  
    char item;  
    pthread_mutex_lock(&b->mutex);  
    while (b->count == 0) {  
        pthread_cond_wait(&b->cv_2, &b->mutex);  
    }  
    item = b->buffer_data[b->out_pos];  
    b->out_pos++;  
    if (b->out_pos == BUFFER_SIZE)  
        b->out_pos = 0;  
    b->count--;  
    pthread_cond_broadcast(&b->cv_1);  
    pthread_mutex_unlock(&b->mutex);  
    return item;  
}
```

cv_2 : "icke tom"

cv_1 : "icke full"

Apropos pthread_create

Varför void*?

Hur skickar man argument till en tråd vid uppstart?

Apropos pthread_create()

```
void *do_work(void *arg) {
    int *id = (int *) arg;
    printf("Hello from thread %d\n", *id);
}

int main(void) {
    int i;
    pthread_t handle[10];
    for (i=0; i<10; i++) {
        pthread_create(&handle[i], NULL, do_work, (void *) &i);
    }
    for (i=0; i<10; i++) {
        pthread_join(handle[i], NULL);
    }
}
```

[void* ?]

Void* är en pekare till en obestämd typ, dvs "mottagaren" kan välja att tolka på det sätt den vill.

Kommer trådarna att få unika id:n?

Apropos pthread_create()

Problem: Loopen hinner köra vidare innan den skapade tråden läser av variabeln i

Lösning 1:

```
int ids[10];
pthread_t handle[10];
for (i=0; i<10; i++) {
    ids[i] = i;
    pthread_create(&handle[i], NULL, do_work, (void *) &ids[i]);
}
```

Denna lösning fungerar bra om man inte har många trådar och argumentet till tråden inte är minneskrävande

Apropos pthread_create()

Lösning 2: Assymetrisk synkronisering

```
sem_t id_read;      // semafor för assymetrisk synkronisering

Void *do_work(void *arg) {
    int *id = (int *) arg;
    printf("Hello from thread %d\n", *id);
    sem_post(&id_read);
}

sem_init(&id_read, 0, 0);
int i;
pthread_t handle[10];
for (i=0; i<10; i++) {
    pthread_create(&handle[i], NULL, do_work, (void *) &i);
    sem_wait(&id_read);
}
```

Fungerar bra för minneskrävande argument till tråden, men man får vänta

Apropos pthread_create()

Lösning 3: Meddelanden

```
#include "messages.h" // meddelandefunktioner

Void *do_work(void *arg) {
    int *id = (int *) arg;
    printf("Hello from thread %d\n", *id);
    int len = message_receive(msgbuf, 4096, PORT_WORK);
}

int i;
pthread_t handle[10];
for (i=0; i<10; i++) {
    pthread_create(&handle[i], NULL, do_work, (void *) &i);
    message_send((char *) &m, sizeof(m), PORT_WORK[i], 0);
}
```

Fungerar bra för minneskrävande argument till tråden, men man får vänta

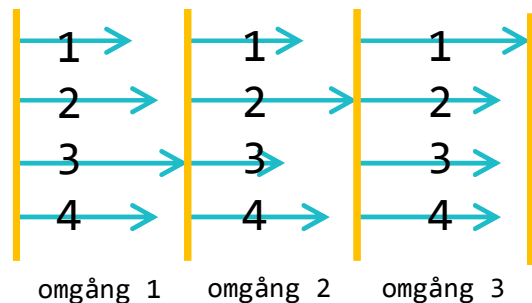
Barriär

Med händelsevariabler

Barrier med CV

- En barrier kan användas vid parallellprogrammering för att snabba upp beräkningar genom att köra dessa på flera processorer, och som en synkroniseringsmekanism för att invänta resultat vid en gemensam punkt.

Exempel: dela upp ett problem i flera omgångar



Omgång 2 kan inte starta förrän alla är klara i omgång 1, osv.

Barrier används för att se till att alla delproblem i första gruppen är klara innan nästa delproblemgrupp kan startas.

Barrier med CV

Ansats:

Huvudloop för en task (det startas alltså 4 st tasks, t ex i main)

```
void *task(void *unused) {  
    int i = 0;  
    while (1) {  
        printf("Before do work (%d)\n", i++);  
        do_work();  
        barrier1(4);  
    }  
}
```

Barrier med CV

Första försöket:

```
pthread_mutex_t mutex;  
pthread_cond_t cv;  
int num = 0; // antal tasks i barriären  
  
void barrier1(int N) {  
    pthread_mutex_lock(&mutex);  
    num++;  
    pthread_cond_broadcast(&cv);           // ny task har anlänt  
    while(num < N) {  
        pthread_cond_wait(&cv, &mutex); // "väntrum"  
    }  
    num = 0; // förbered för nästa körning av barrier1()  
    pthread_cond_broadcast(&cv);           // task lämnar barriär  
    pthread_mutex_unlock(&mutex);  
}
```

Vad kommer att hända?

Barrier med CV

Svar:

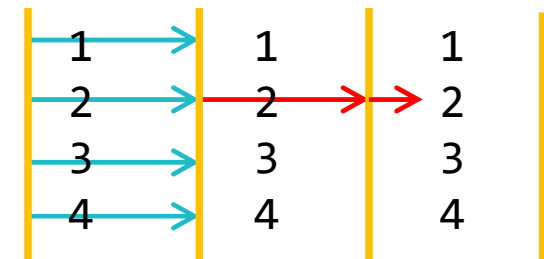
Fungerar inte alls, då den första tråden direkt kommer att sätta $\text{num}=0$ så snart den går ut ur loopen.

Dvs, de övriga trådarna kommer att fastna i barriären eftersom villkoret `while (num<N)` åter då är uppfyllt.

Barrier med CV

```
void barrier2(int N) {  
    pthread_mutex_lock(&mutex);  
    num++;  
    pthread_cond_broadcast(&cv);           // ny task  
  
    while(num < N && (num != 0)) {  
        pthread_cond_wait(&cv, &mutex);    // "väntrum"  
    }  
  
    num = 0; // förbered nästa körning av barrier2()  
    pthread_cond_broadcast(&cv);           // task lämnar  
    pthread_mutex_unlock(&mutex);  
}
```

← Nytt villkor



Ser ut att fungera bra, men vad händer om en "snabb" tråd hinner anropa `barrier2()` igen, innan övriga trådar lämnat barriären?

```
typedef enum {ENTERING, EXITING} barrier_state_t;  
barrier_state_t state;
```

Barrier med CV

```
Void barrier3(int N) {  
    pthread_mutex_lock(&mutex);  
    while(state == EXITING) {  
        pthread_cond_wait(&cv, &mutex);    // "vänta i kapprummet"  
    }  
    num++;  
    pthread_cond_broadcast(&cv);            // ny task  
    while(state == ENTERING) {  
        pthread_cond_wait(&cv, &mutex);    // "väntrum"  
        if (num == N) {  
            state = EXITING;  
            pthread_cond_broadcast(&cv);    // nytt state (behövs egentligen inte)  
        }  
    }  
    num--;    // räkna ner räknaren och ändra till ENTERING-tillståndet  
              // när alla är ute från barrier  
    if (num == 0) {  
        state = ENTERING;  
    }  
    pthread_cond_broadcast(&cv);            // task lämnar  
    pthread_mutex_unlock(&mutex);  
}
```

Processer (dvs inte trådar)

Trådar vs Processer

- Trådbaserad applikation: Alla trådar i ett program har samma minnesutrymme. Trådar skapas startas i Linux med funktionen `pthread_create()`.
- Processbaserad applikation: Alla processer har var sitt minnesutrymme. Processer skapas/startas i Linux med funktionen `fork()`.

Hur fungerar fork?

Fork skapar en kopia av originalprocessen (både minnesutrymme och öppna filer) och gör även den nya processen körklar.

I den gamla processen (parent) returnerar fork process-id:t för den nya processen. I den nya processen returnerar fork värdet 0.

Följande är för övrigt en dålig ide:

```
while(1) {  
    fork();  
}
```

Hur skapas en process som kör en viss funktion?

// parent process (new_pid!=0)

```
int main(...) {  
    pid_t new_pid = fork();  
    if (!new_pid) {  
        // child process  
        sender();  
    } else {  
        // parent process  
        receiver();  
    }  
    return 0;  
}
```

// child process (new_pid==0)

```
int main(...) {  
    pid_t new_pid = fork();  
    if (!new_pid) {  
        // child process  
        sender();  
    } else {  
        // parent process  
        receiver();  
    }  
    return 0;  
}
```

Kommunikation mellan processer

Problem: Hur kan processer kommunicera när de inte kan titta i samma datastruktur?

Exempelvis

- Filer
- Signaler (t ex Ctrl-C skickar SIGINT-meddelande)
- Semaforer (kan skapas i variant som använder gemensamt minne)
- Meddelanden
 - Sockets (Exempelvis TCP/IP)
 - Pipes
 - Posix Message Queues (används i lab 4)

Message Passing

Message passing / Meddelandekommunikation

- Det är ofta vettigt att abstrahera kommunikation mellan tasks med hjälp av meddelanden istället för att direkt läsa/skriva i delade variabler

Message passing (MP) kontra delat minne (DM):

DM: Kommunikation mellan trådar sker via delade variabler

* Data läses/skrives på plats av trådar. Accessmönstret påverkas av:

- Operativsystemet
- Andra aktiviteter på systemet
- Annan hårdvara på systemet (som använder bussen)
- Karakteristik hos processorn (cachemissar, branch prediction m m)

• MP: All kommunikation mellan tasks sker via ett fåtal funktioner

* Data måste kopieras mellan tasks (mycket overhead)

* Underlättar dock ofta felsökning (man kan ofta skriva en process så att beteendet enbart beror på inkommande meddelanden)

Message passing

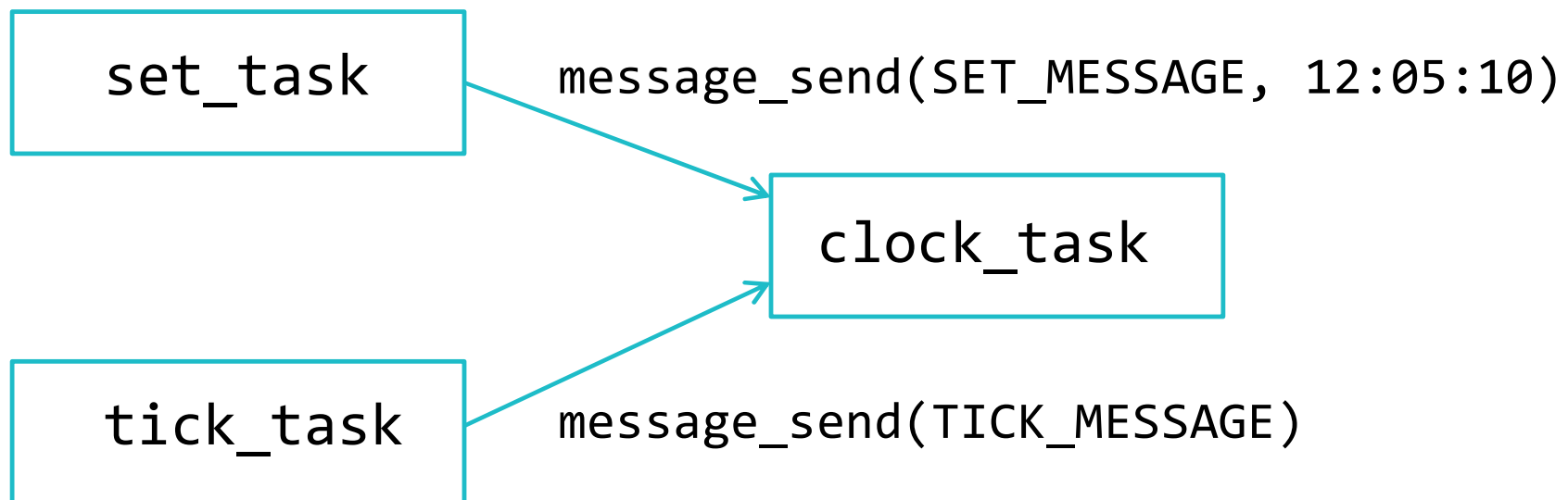
Funktioner: (ingår i `messages.c` / `messages.h` i uppgift 4)

```
int message_receive(char *msg, int length, int queueid);  
// Tar emot meddelanden på angiven meddelandekö, returnerar längden.  
// Meddelanden skrivs till buffern msg och får max vara längden length.  
// Blocking (väntar på ett meddelande innan den returnerar)  
  
void message_send(char *msg, int length, int queueid, int priority);  
// Skickar meddelandet i buffern msg av längden length till angivet  
// queue-id med angiven prioritet.  
// Non-blocking (om det finns plats i kön, annars blocking)
```

Implementationen av messages liknar `put_item/get_item`

Message passing

Exempel: Klocka med message passing (jfr med Lab 1)



Message passing

```
struct message {  
    int type;  
    int hours;  
    int minutes;  
    int seconds;  
};  
  
#define TICK_MESSAGE 0  
#define SET_MESSAGE 1  
  
#define QUEUE_CLOCK 0
```

Message passing

```
// clock_task loop:
while(1) {
    char buf[1024];
    int length = message_receive((char *) &buf, 1024, QUEUE_CLOCK);
    struct message *msg;
    msg = (struct message *) buf;
    if (length != sizeof(struct message)) {
        panic("Internal error: Wrong message size");
    }
    // Notera: inga lås krävs nedan
    switch(msg->type) {
        case TICK_MESSAGE:
            increment_time(&time);
            display_time(&time);
            break;
        case SET_MESSAGE:
            time.hours = msg->hours;
            time.minutes = msg->minutes;
            time.seconds = msg->seconds;
            break;
        default:
            error("Unknown message");
    }
}
```

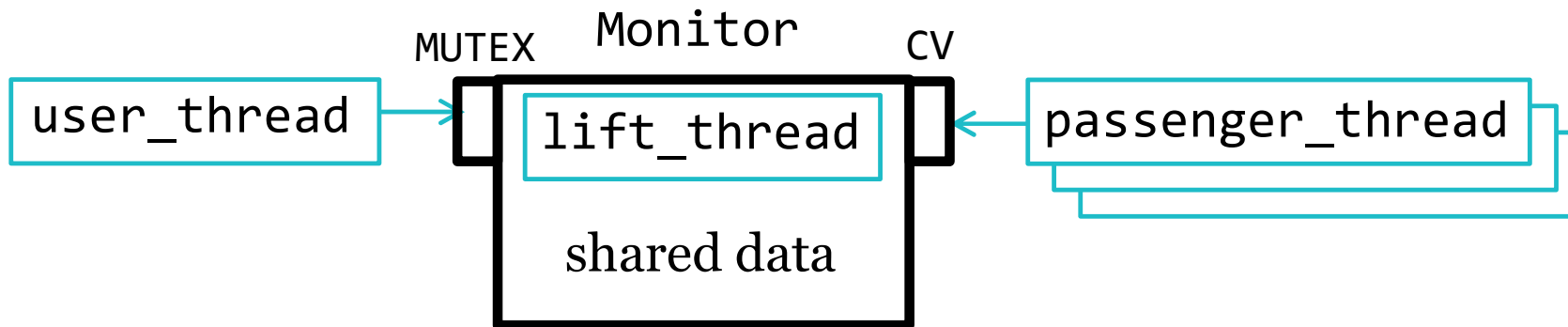
Message passing

```
// tick_task loop

while(1) {
    msg.type = TICK_MESSAGE;
    // resterande fält i msg används ej
    message_send((char *) &msg,
                 sizeof(struct message, QUEUE_CLOCK, 0);
    usleep(1000000); // wait until är bättre
}
```

Lab3, Lab4

Implementera ett hiss-system med flera trådar som använder en monitor för att skydda gemensamma resurser.



Tips till uppgift 3: valgrind och gdb

- Använd valgrind, vilket kan upptäcka ”vissa” kapplöpningsproblem (race conditions):

```
valgrind --tool=helgrind ./dittprogram
```

- Om du får segmentation fault:

```
gdb ./dittprogram  
r  
(vänta på krasch)  
bt
```

Nu kan man se en så kallad backtrace för kraschen.

Apropos Spurious Wakeup

Följande kod är **ej** garanterad att fungera:

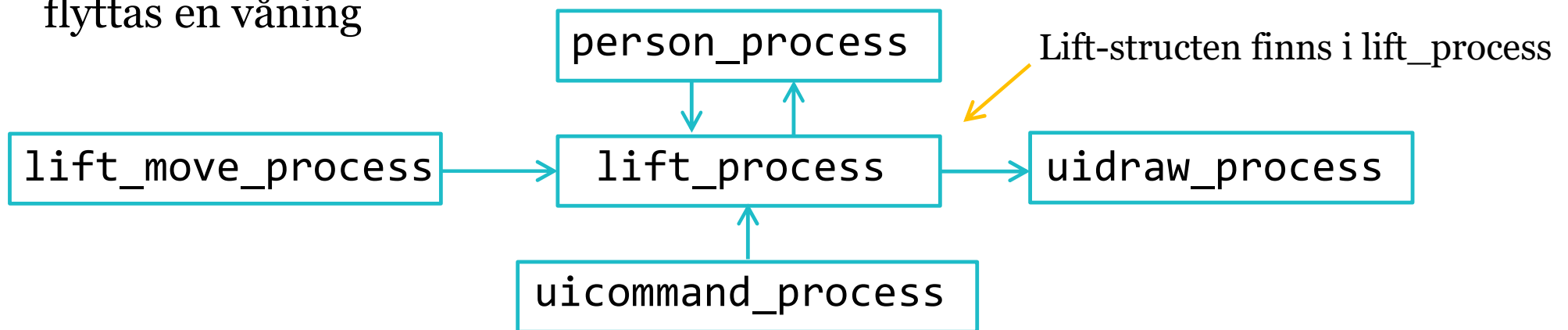
```
if (villkor()) {  
    pthread_cond_wait(&cv, &mutex);  
}
```

- Enligt pthreads-standarden kan pthread_cond_wait i sällsynta fall returnera trots att ingen kört pthread_cond_broadcast()!
Så kallade Spurious Wakeups.
- Spurious Wakeups möjliggör optimeringar (speciellt på system med många kärnor).
- **VIKTIGT: Använd alltid while(villkor())** med händelsevariabler!
- Notering: Man kan inte göra asymmetrisk synkronisering med pthread_cond_wait resp pthread_cond_broadcast.

Lab4 : Message passing

person_process skickar ett TRAVEL-meddelande till lift_process:

- * Skapa en person på våning X och skicka ett TRAVEL_DONE till mig när personen nått sin våning Y.
- * lift_move_process skickar MOVE-meddelanden varje gång hissen ska flyttas en våning



Notering: draw_lift får bara anropas från en process. Detta sker ifrån uidraw_process. (Lift-strukturen skickas till uidraw-processen som ett meddelande.)

För många föreläsningar i schemat

- 22/11 ingen föreläsning
- 13/12 ingen föreläsning

Anders Nilsson

www.liu.se