

# **Seminarie 2**

## **Operativsystem**

Henrik Borg

January 8, 2020

**Or...**

No problem - On one condition

No shared resources

---

## 2.4 A small test

In this task two or more threads are supposed to issue a yield and by that take turn in performing their respective tasks. For that we need a ready queue to keep track of the threads that are ready to run, which in this case are all threads.

### How do we create the queue of threads?

The thread structure contains links to the next thread to run (struct green\_t \*next;). The first element in the list is the one currently running, there is a local static variable for this (static green\_t \*running;).

### How to add a thread to the end of the queue?

We can either traverse the list//

```
while(NULL != object->next)
    object = object->next;
object->next = last_thread;
```

Or we can keep track of the end of the list and directly add a thread to the end of the list. In this example we also take care of the corner case then the list is empty.

```
if(NULL == ready_queue_end) {
    ready_queue_end = new;
    ready_queue_end->next = new;
    running->next = new;
} else {
    ready_queue_end->next = new;
    ready_queue_end = new;
}
```

Do we need to keep track of the end of the queue?

The simple answer here is no, and yes. It depends on if we need to do it in a constant and very well defined length of time? If we are developing an application for a real time system with hard dead lines, then we have to perform calculation in a formal way to prove that the system will survive for ever, or at least till it is put out of use for other reasons than bad development. In this case we want to do it in a constant and very well define length of time, in clock cycles. Or if we are developing a system with a huge amount of threads so the time it takes to traverse huge queue of threads will have a bad enough impact on the over all system, then we also want to do it in constant time. We can do it in constant time by keeping track of the end of the list.

## 3 Suspending on a condition

In this task the threads take turn by suspending themselves with a wait-command, thereby it is put in a waiting queue. At a signal-command the first thread in the waiting queue will be moved back to the ready queue.

### Is it safe to run multi-threaded programs without locks?

---

The answer to this question is easier than one can think. It is YES, but on one condition: NO shared resources!

Can we reuse the previous type of queue? In the way the ready queue were implemented meen that than we jump issue a Not without a lot of work. `typedef struct green_thread_queue struct green_t *next; struct green_t *end;`

## 4 Adding a timer interrupt

In this task we will on top of the wait- and signal-commands add a periodic timer that triggers a timer interrupt handler who is a scheduler of type Round Robin that put the running task back into the ready queue and activate the next task in the ready queue.

**Can this lead to problems?** Yes, if we do not understand the implications of this. But it is often quite easily handled. Every time we will work on a shared resource we must somehow take care the timer interrupt problme. In this task the shared recourses are the queues and our global variable 'running'. Before we touch the shared resource we must first turn off the timer interrupt and then turn the timer interrupt back on again then these actions are done.

**Can these handling of the timer interrupt lead to any problem?** Yes it can. If the time while the timer interrupt is turned off streches over two timer interrupts will lead to a missed interrupt, possibly leading to problems. A similar problem occurs then the time while the timer interrupt is turn off streches over one timer interrpt and is long enough so the task or tasks that should be handled in the next time period can not finish in time before the next timer interrupt, possibly leading to problems. These are two common problem while developing real time systems with firm or hard deadlines, for soft deadlines it is no problem since breaking soft deadlines can not affect a system negatively.

**How to handle this problem in a good way?** It depends on the application. One example is if we read some shard resources, do some calculations and then write to some othere shared resources. Then we should do all the reading in the beginning of the task, and all the writing at the end of the task while only disable the interrupt during the reading respectively the writing. In or case then we issue a read modify write of te shared resources, we must disable the interrupt from the first reading to the last writing, and we should do this in as short time as possible, meaning doing all the precalculations on beforehand and all postcaclulations after.

## 5 A mutex lock

## 6 The final touch

## 7 Summary