**The Process Scheduling Shootout**

**Johan Montelius**

**HT2016**

# 1 Introduction

This is an assignment that requires some skill in both C and Erlang (you can use another language such as Go that provide so called *green threads*, check with me before you start). Your task is to set up a ring of processes, pass around a token in the ring and measure how fast this can be done. You should show a performance study and propose a plausible reason for why the results look like they do.

Before you start with this assignment you need a good understanding of how to use *pthreads* in C.

# 2 The Ring

You should write, in both C and Erlang, a program that creates a *ring* of processes. Each process has the possibility to send a *token* to the next process in the ring, so when we start the program the token will be sent around in the ring. When the token has been sent around some number of rounds the program should stop and all processes should terminate. The program should also produce some measurement that tells us how fast it could send the token around in the ring.

A process that simply waits for a message and forward this to the next process in line, will as you might expect be almost trivial to implement (four lines of Erlang). The tricky part will probably be to set up the ring and be sure that all processes have actually been created before you start to send the token around.

You of course also need to keep track of how many rounds the token has been sent in order to know when to stop. This can be solved in several ways, either all process know that they should pass the token around for a certain number of rounds or you have one controller that that does the counting and then terminate all processes (by sending a special message) when the last round has been completed.

## 2.1 in C

In C you can choose to implement the processes using either threads or regular processes. If you choose threads then the communication might be easier to solve and it will probably be easier to set up the ring. If you choose to implement the ring using regular processes you could for example set up

pipes between them. If you have time you can do both and see if there is any difference.

You could choose to implement your own locking mechanism but it will be so much easier to use the mutex-locks in the pthread library. What follows is not a complete implementation but things you get you started going in the right direction.

You will need some data structure that serves as a placeholder of the token and you will have one such structure between any to processes in the ring. In order for one process to add the token and the next to pick it up you need mutex-locks to protect it. It could look as follows:

```
typedef struct synch_t {
  int token;                  // the token
  pthread_mutex_t mtx;        // protected by the lock
  pthread_cond_t cnd;         // the conditional variable
} synch_t;
```

You will also need a data structure to pass the arguments to the threads that we will create. Each thread should be given two placeholders, information on how many rounds, a barrier etc.

```
typedef struct proc_arg_t {
  :
  :
} proc_arg_t;
```

Each process in the ring will run a procedure that will try to take token from one side and place the token on the other side. To do this it must of course take the mutex-locks. You need to understand mutex-locks and how to suspend on conditional variables.

```
void *init(void *arg) {

  // unpack the arguments and get hold of the two placeholders
   :

  // wait for everyone to sign in
  pthread_barrier_wait(barrier);

  // let's go
  while(rounds > 0) {
    // grab lock of previous
    pthread_mutex_lock(prev_mtxp);

    // check if token is there if not suspend on condition
      :

```

```
    // remove the token and unlock previous
      :

    // take mutex-lock and place token in next placeholder
      :

    // unlock and signal on conditional
      :

    rounds--;
  }
  // wait for everyone to sign in
  pthread_barrier_wait(brp);
}
```

The benchmark procedure will first create an array of thread arguments. It will initialize the structures so that the

```
int benc(int p, int r) {
    :
  // set up the barrier
    :
  // allocate some space for a table of thread arguments
    :
  // initialize the thread arguments
    :
  // take lock of one placeholder
    :
  // create all threads
    :
  // take time
    :
  // let's wait for all to sign in
    :
  // start timer
    :
  // set token, release lock and signal condition
    :
  // wait until all done
    :
  // stop timer
    :
  // return result
}
```

Hmm, that was not a lot of code but I think you can figure out how to

do it. The only tricky part is when you initialize the tread arguments and make sure that the first thread will have access to the last placeholder.

### 2.2 in Erlang

If you remember some Erlang you should have this up and running in no-time. The tricky part is of course to make sure that all processes have been created and maybe how to connect the last process to the first to close the ring.

This could be the implementation of a process. In this solution all *slave processes* simply pass a token to the process in front of them untill they receive a `stop` message. A *controller process* is responsible for counting the number of rounds.

```
slave(Lst) ->
    receive
        token ->
            Lst ! token,
            slave(Lst);
        stop ->
            Lst ! stop,
            ok
    end.
```

Since Erlang has message passing as a core part of the language the solution will be quite simple.

### 2.3 measuring time

Make sure that you measure time in a appropriate way. If you measure process time you might not measure the right time, since process will not be running if the toke is not around. You need to use so called *wall time*. Explain how and why you measure time in the way you do.

## 3 The Experiments

How many processes can we have in a ring? How long time does it take to send a message around? Is the time to pass a message from one process to another depending on how large the ring is? Is there any difference between your Erlang implementation and your C implementation? If so, why is there a difference?

Present your benchmark numbers in one or two diagrams and write up a nice six page report that describes the benchmarks and your findings. Use the template in LaTeXthat is provided.