Seminar 2: Processes and Threads
Sistemes Operatius

1 Introduction

The objective of this seminar is to understand Processes, Threads and their differences. We are going then to look at the basic programming schemes Concurrent and Sequential for both processes and threads. Here you have helper signatures that you will need:

```
int fork();
int wait(int* status);
int waitpid(int pid, int* status);
execvp(char *name, char* argv[]);

pthread_create(pthread_t *tid, NULL, fthread, void* p);
int pthread_join(pthread_t thread, NULL);
pthread_mutex_lock(pthread_mutex_t* lock);
pthread_mutex_unlock(pthread_mutex_t* lock);
```

```
int open(char* name, O_CREAT | O_RDWR, 0644);
int read(int fd, void *buf, int nbyte);
int read_split( int fd, char* buff, int maxlen, char* ch_end )
sscanf(buff,"%d", &int_var);
sprintf(buff,"%d", int_var);
void close(int fd);
```

1.1 Processes

1. Create 100 processes which all wait 1 second (using the function sleep provided in unistd.h) and the father waits for them to finish at the end after their creation.
2. Create 100 processes that execute concurrently the shell command "sleep 1" (which creates a process that lasts for 1 second) and wait for them to finish.
3. Create 100 processes that execute sequentially the shell command "sleep 1" (which creates a process that lasts for 1 second) and wait for them to finish.
4. Create a number of processes indicated by a command line argument. Each process executes the command line "ps" and waits for them to finish at the end.
5. Assume we have a python file simulation.py that can be run from the command line using the python interpreter as below including and argument option "–seed" for the random number generator. Launch 100 concurrent simulations with seeds from 0 to 99.

   ```
   python simulation.py --seed 23
   ```

6. Read shell commands from standard input (using read_split and while there is input) and allows for the execution in the background using the character ampersand &. Assuming you have a text file (cmds.txt) with shell commands written on it, how can you call this program in the shell to execute all commands in the file?

1.2 Threads

1. Create 100 threads such that all wait 1 second (using the function sleep provided in unistd.h) and the main thread waits for them to finish at the end after their creation.
2. Set a global array $A[100]$ to the square of its index $A[i] = i * i$, using a thread per cell.
3. Set a global matrix $A[100][100]$ with each position being $A(i, j) = i * j$ using a thread per cell. To pass the parameters to the thread function you can use an array of two positions to indicate the cell to each thread or a struct.

1.3 Delivery Exercises

1. Set each entry of an array $A[100]$ to the square of its index $A[i] = i*i$ in the following way: (1) initialize it to 1 (for all i, $A[i] = 1$), (2) create 2x100 threads that each multiply its given index i by i such that each cell is multiplied twice, resulting $A[i] = i * i$ as in thread ex. 2. Discuss the need of locks; do you need one or many?

2. Set each entry of an array $A[100]$ to the square of its index $A[i] = i*i$ in the following way: (1) initialize it to 1 (for all i, $A[i] = 1$), (2) create 2 that each multiply its given index i by i one starting from 0 and the other from the end such that each cell is multiplied twice, resulting $A[i] = i * i$ as in thread ex. 2. Discuss the need of locks; do you need one or many?