# Examen d'Architecture et Système

16 janvier 2015

Duration: 2 hours. Answers can be given in either English or French. Justify all your answers. The final mark for the course will be either the average of the marks for the midterm and the final exam, or the mark of this exam, whichever is better.

**Notes on the correction:** There were 50 points available in total, 23 for the first section, 12 for the second, and 15 for the third. The marks were calculated on a basis of 45 points. (I.e. 22.5 points to obtain a mark of 10, 45 to obtain a mark of 20.)

## 1 Processes, threads and locks

In this part, we will review processes and threads, their similarities and differences, and approaches to synchronization. Consider the programs P1 and P2 presented in Figure 1.

(a) *In the following, you will be given a series of potential outputs of the programs P1 and P2. For every output, discuss whether this output could be obtained from executing P1 or P2, respectively. If the output is possible, describe an execution leading to this output. Otherwise, justify why the output cannot be obtained.*

(i) `Two Three One`

(ii) `Two One Two`

(iii) `Two Two Two`

(iv) `Three Three`

**Solution:** Recall that spawned processes and threads can be executed in any *arbitrary* order; and that one main difference between processes and threads is that each process possesses its own address space, whereas global variables are shared between threads.

(i) This output can be obtained from both P1 and P2. For P1, the main process P1 first spawns a child process C1 in line 7 and sleeps; C1 runs, calls `f("One")` and then outputs `One`; P1 wakes up, spawns a child process C2 in line 11 and sleeps; C2 executes and outputs `Two`, etc. The execution pattern works analogously for P2.

```
// Program P1

1:  char* s;

2:  void* f(void* data) {
3:   s = (char*) data;
4:   printf("%s ", s);
5:  }

6:  int main() {
7:   int pid = fork();
8:   if (pid == 0) {
9:     f("One");
10:  } else {
11:    pid = fork();
12:    if (pid == 0) {
13:      f("Two");
14:    } else {
15:      pid = fork();
16:      if (pid == 0)
17:        { f("Three"); }
18:    }
19:  }
20: }
```

```
// Program P2

1:  char* s;

2:  void* f(void* data) {
3:   s = (char*) data;
4:   printf("%s ", s);
5:  }

6:  int main() {
7:    pthread_t t1,t2,t3;
8:    pthread_create(&t1,NULL,
                          f,"One");
9:    pthread_create(&t2,NULL,
                          f,"Two");
10:   pthread_create(&t3,NULL,
                        f,"Three");
11:   pthread_join(t1, NULL);
12:   pthread_join(t2, NULL);
13:   pthread_join(t3, NULL);
14: }
```

Figure 1: Program P1 (left) and P2 (right).

(ii) This output cannot be obtained by P1. In P1, each time `f` is called with a different argument, and since child processes run in their own address space, every output `One`, `Two` and `Three` has to occur exactly once.

This output also cannot be obtained by P2. The shared variable `s` can only be set to `"Two"` once. However, after having been set to `"Two"` for the first time, `s` gets set to `"One"`, and thus the second output `"Two"` is not possible.

(iii) The same argument as above shows that this cannot be obtained as an output of P1.

It is, however, an output of P2. First, P2 spawns three threads T1, T2 and T3 which immediately sleep after being spawned. Then, T1 awakes and sets `s` to `"One"`, then T3 awakes and sets `s` to `"Three"`, and then T2 awakes and sets `s` to `"Two"`. Now all three threads progress and call the `printf` function, which outputs the value of the shared variable `s`, which now is `"Two"` in all three threads.

(iv) This output cannot be obtained from either P1 or P2. For P1, the reasoning is

```
// Thread T1                    // Thread T2

a1();                           b1();
sem_wait(&semB);                sem_wait(&semA);
sem_post(&semA);                sem_post(&semB);
a2();                           b2();
```

Figure 2: The two threads T1 (left) and T2 (right).

as above. For P2, since all three threads produce some output, and P2 waits for all three threads to finish, there have to be at least three, not necessarily different, outputs.

(b) *In P2, there is a line that can be moved to another place so that the program behaves like P1. Which line is it, and where should it be moved?*

**Solution:** When line 1 is moved to become the first line of the `f` function, `s` becomes a local variable of each thread, and thus its value cannot be changed by other threads. Consequently, any execution that exists in P1 has a corresponding execution in P2, and vice versa.

(c) *Which of the outputs given in (a) would additionally be possible if the last three lines (11-13) of P2 were to be removed?*

**Solution:** Output (iv) would additionally be possible. The last three lines wait for each thread to terminate. If they are removed, as soon as the main process P2 terminates, all previously spawned threads are terminated as well. An execution similar as in case (iii) could thus lead to output (iv) when one thread is terminated before it can call the `printf` function.

In the course, you learned that semaphores allow to synchronize threads. Figure 2 shows two threads, T1 and T2, that run in parallel. The threads share two semaphores, `semA` and `semB`, which have been initialized with `sem_init(&semA,0,0)` and `sem_init(&semB,0,0)`, i.e., capacity zero. The goal of the programmer was to make sure that `a2` and `b2` can only be executed after both `a1` and `b1` have finished.

(d) *What happens when you execute both threads in parallel? How does one call this type of problem, and under which circumstances does it happen in general?* (A short explanation is sufficient.)

**Solution:** Both threads hang, because T1 waits for `semB` to be released, and T2 waits for `semA` to be released. In general, this is called a deadlock, which occurs when there are cyclic dependencies on shared ressources between threads.

(e) *Fix the implementation such that the original goal is achieved. Note that the order in which* `a1` *and* `b1` *are executed should not be pre-determined.*

**Solution:** Swapping the `sem_wait` and `sem_post` lines in either T1 or T2 fixes the problem. Suppose we swap those lines in T1. Then `a1()` and `b1()` can be executed in any order. As soon as T1 finishes `a1()`, it releases `semA` and waits for `semB` to be released, i.e., for T2 to finish `b1()`. Likewise, after executing `b1()`, T2 waits for `semA` to be released, i.e., for T1 to finish the execution of `a1()`, and then releases `semB`.

Finally, we will consider a classical mutual exlcusion problem. Suppose there are a car and $n$ passengers that want to take a ride with the car. The car has capacity $c < n$ and will only drive once it is full and only once (i.e., some passengers are left behind). We wish to model the car and the passengers as threads.

The thread modeling the car can invoke the functions `load`, `drive` and `unload`. The thread modelling the passengers can invoke the functions `board` and `unboard`. (We assume that these five functions are given and do not take any arguments.)

The use of these functions is subject to the following conditions:

- Passengers cannot invoke `board` until the car has invoked `load`.
- The car cannot invoke `drive` until $c$ passengers have boarded.
- No more than $c$ passengers can board the car.
- Passengers cannot unboard until the car has invoked `unload`.

(f) *Write code that is executed in a separate thread by each passenger and for the car that enforces these constraints. You may use local variables and operations on them, semaphore functions, and the above mentioned functions* `board`, `unboard`, `load`, `drive`, *and* `unload`.

**Solution:** We will use four shared semaphores in order to synchronize between cars and passengers, `semBoard`, `semAllBoarded`, `semUnload` and `semAllUnboarded` which are all initialized with capacity zero. A possible solution for the car could be as follows

```
load();                        // initiate loading of the car
int i;
for (i = 0; i < c; i++) {
   sem_post(&semBoard);        // signal car can be boarded c times
}
```

```
sem_wait(&semAllBoarded);     // wait for full loading
drive();                      // drive
unload();                     // initiate unloading of the car
for (i = 0; i < c; i++) {
   sem_post(&semUnload);      // signal car can be unboarded c times
}
sem_wait(&semAllUnboarded);   // wait for all passengers to be unboarded
                                 (optional)
```

In addition, passengers use two additional semaphores in order to coordinate entering the car, `sem1` and `sem2`, both initialized with capacity 1. They also share integer variables `boarders` and `unboarders` initialized with value 0. A possible solution for the passengers could be as follows

```
sem_wait(&semBoard);           // wait for boarding signal
board();

sem_wait(&sem1);               // lock to ensure atomicity
  boarders++;                  // increment number of boarded passengers
  if (boarders == c) {         // if c passengers have boarded...
    sem_post(&semAllBoarded);  // ...give a signal to the car
  }
sem_post(&sem1);               // unlock

sem_wait(&semUnload);          // wait for unloading signal
unboard();

sem_wait(&sem2);               // lock to ensure atomicity
  unboarders++;                // increment number of unboarded pass.
  if (unboarders == c) {       // if c passengers have unboarded...
    sem_post(&semAllUnboarded);// ...give a signal to the car
  }
sem_post(&sem2);               // unlock
```

**Notes on the correction:** There were 8+2+2+3+2+6=23 points available in this section.

# 2 Input/output

Pipes are used to facilitate communication between processes. Recall that the system call `pipe(p)` returns two file descriptors, where `p[0]` is used for reading and `p[1]` for writing. Also recall that file descriptor 1 is the standard output, usually the console.

(a) Consider the program P3 on the left-hand side of Figure 3. The parent sends the words `Please`, `fix`, and `me!!` to the child, with small pauses in between. What is the output of the program?

**Solution:** An almost identical program was shown in class. Recall that `read` will block until some input is available. If some input is available, it will return as many characters as it can get, but at most the number given in the call (5 in this example).

The parent first writes seven characters to the pipe. The child will consume the first five in its first read, leading to the output `Pleas`. The second read will obtain only two characters; thus the first two bytes in the buffer are overwritten but all five written to the console. The output from the second read is therefore `e eas`.

After the first `sleep` in the parent, the child obtains four characters and prints `fix s`, then `me!!s`. Altogether, the output is therefore

```
Please easfix sme!!s
```

**Notes on the correction:** Four points were available, I gave one for each write. (So `Pleas` gave at least one point.) Unfortunately, only three persons got the output right.

(b) Apart from the output, what else is wrong with P3? Sketch briefly how to fix it (no actual code necessary).

**Solution:** The parent terminates, hence the shell resumes operation. However, the child has not terminated. While the parent's access to the writing end of the pipe is closed when it terminates, the child itself has not closed it, thus `read` in the child fails to detect end-of-file. The child is therefore stuck in the fifth call to `read` (without consuming any CPU power, but remaining in memory).

The fix is for the child to close `p[1]` and test for the return value of `read`, which is 0 when the end of the file has been reached.

**Notes on the correction:** Four points were available. Explaining what is wrong (and why!) gave two points, explaining the fix (close pipe and test the result of read) gave two points. For other methods of fixing the program, up to two points were given depending on details.

```
// Program P3

1: int main ()
2: {
3:    int p[2];
4:    pipe(p);
5:
6:    if (fork()) {
7:      close(p[0]);
8:      write(p[1],"Please ",7);
9:      sleep(1);
10:     write(p[1],"fix ",4);
11:     sleep(1);
12:     write(p[1],"me!!",4);
13:   } else {
14:     char c[5];
15:     while (1) {
16:       read(p[0],c,5);
17:       write(1,c,5);
18:     }
19:   }
20: }
```

```
// Program P4

1: int main ()
2: {
3:    int p[2];
4:    pipe(p);
5:    close(p[0]);
6:
7:    if (fork()) {
8:      while (1) {
9:        sleep(1);
10:       write(p[1],"a",1);
11:     }
12:   } else {
13:     char c;
14:     while (1) {
15:       read(p[0],&c,1);
16:     }
17:   }
18: }
```

Figure 3: Programs P3 (left) and P4 (right), both using pipes.

(c) Consider the program P4 on the right-hand side of Figure 3. Here, the parent ought to send a stream of as, one per second. When you run P4, you make the following observations instead: (i) the program returns to the shell almost immediately; (ii) after a while your computer feels hot, and you see that the system load is at 100%. Explain both phenomena. Sketch how to fix the program.

**Solution:** Again, an almost identical program was shown in class. The fact that the program returns to the shell tells us that the parent process has terminated (and not the child!). The fact that CPU power is consumed tells us that the child is still active and continually executing. (Even if it is was the parent that survived, it would not consume 100% of the CPU, but sleep most of the time.)

The primary reason for these effects is the `close` statement before `fork`. This has two effects: (i) The parent writes onto a pipe whose reading end has been closed by all parties, therefore receives an uncaught SIGPIPE signal, which terminates the process. (ii) The child reads from a closed filehandle, which fails immediately, returning -1 and an error code (which is not evaluated). This call is repeated without any pause.

Eliminating the offending `close` statement fixes the error. (The specification of the program's intended behaviour does not require it to terminate.)

**Notes on the correction:** Four points were available, distributed as follows:

(i) Deducing that the parent (and not the child) has died gave one point.

(ii) Explaining the reason why the parent dies gave another point (but no one got it).

(iii) Explaining correctly why the child consumes 100% CPU power gave another point.

(iv) Fixing the program gave again one point.

Detecting that the `close` statement was at the origin of the problem but drawing the wrong conclusions from it gave at least 0.5 points for (i) to (iii).

# 3 Memory allocation strategies

In this part of the exam, we will look at strategies for memory allocation.

A memory allocator allows for reserving isolated blocks of memory inside a larger block of memory. For this exercise, we consider a memory allocator that is simplified with respect to the actual memory allocator found in Unix:

- `int malloc(int size)` allocates `size` many blocks and returns a unique identifier of this block. If there is no block of free memory of the requested size available, `malloc` will fail.
- `void free(int identifier)` allows for freeing the block previously allocated by a call to `malloc`.

For instance, suppose we are given a chunk of 12 blocks of memory, illustrated below, where `X` indicates free space.

```
+----------------------------+
|XXXXXXXXXXXXXXXXXXXXXXXXXXXX|
+----------------------------+
```

A call `malloc(5)` reserves 5 blocks of memory and returns a unique identifier, e.g., `id1`.

```
+----------------------------+
|     5    |XXXXXXXXXXXXXXXXX|
+----------------------------+
    id1
```

Two more calls to malloc may partition the memory as follows.

```
+----------------------------+
|     5    |    4    |  2  |XX|
+----------------------------+
    id1         id2      id3
```

Now a call to `free(id2)` will lead to the following partition of memory.

```
+----------------------------+
|     5    |XXXXXXXXX|  2  |XX|
+----------------------------+
    id1                 id3
```

Note that even though there are 5 blocks of free memory available in total, at most 4 blocks could be allocated, since a memory allocator is not allowed to move memory around. This problem is called fragmentation.

There exist three popular memory allocation strategies to cope with fragmentation:

- *best fit*: the allocator places newly requested memory into the smallest possible free block of memory. In the example above, `malloc(1)` would return the block to the right of the block identified by `id3`.
- *worst fit*: the allocator places newly requested memory into the largest possible free block of memory. In the example above, `malloc(1)` would return one block to the right of the block identified by `id1`.
- *first fit*: the allocator places newly requested memory into the first possible free block of memory, starting from the left. In the example above, `malloc(1)` would return one block to the right of the block identified by `id1`.

In the following questions, you are free to choose the size of the memory and the size of the blocks to be allocated.

(a) Develop a sequence of `malloc` and `free` operations under the *best-fit* strategy such that a subsequent call to `malloc` will fail, however it would succeed if either the worst-fit or first-fit strategy would have been used.

**Solution:** We choose a chunk of 6 blocks of memory and the following sequence of operations:

```
1:  id1 = malloc(3);
2:  malloc(1);
3:  free(id1);
4:  malloc(1)
5:  malloc(2);
6:  malloc(2); // fails
```

This leads to the following sequence of memory configurations when executing lines 1 to 5:

```
+---------------------+
|    3    | XXXXXXXXX |
+---------------------+
+---------------------+
|    3    | 1 | XXXXX |
+---------------------+
+---------------------+
| XXXXXXX | 1 | XXXXX |
+---------------------+
+---------------------+
| XXXXXXX | 1 | 1 | X |
```

```
+--------------------+
+--------------------+
|  2  | X | 1 | 1 | X |
+--------------------+
```

Even though there are 2 free blocks available, `malloc(2)` fails due to fragmentation. However, with the worst- and first-fit strategies, at line 5 we would reach the following configuration, in which line 6 would not fail:

```
+--------------------+
| 1 |  2  | 1 | XXXXX |
+--------------------+
```

(b) Develop a sequence of `malloc` and `free` operations under the *worst-fit* strategy such that a subsequent call to `malloc` will fail, however it would succeed if either the best-fit or first-fit strategy would have been used.

**Solution:** We choose a chunk of 6 blocks of memory and the following sequence of operations:

```
1:  id1 = malloc(2);
2:  malloc(1);
3:  free(id1);
4:  malloc(1)
5:  malloc(3); // fails
```

This leads to the following memory configuration at line 4:

```
+--------------------+
| XXXX | 1 | 1 | XXXX |
+--------------------+
```

However, using the other two strategies we would have obtained the following memry configuration, in which `malloc(3)` does not fail:

```
+--------------------+
| 1 | X | 1 | XXXXXXX |
+--------------------+
```

(c) Develop a sequence of `malloc` and `free` operations under the *first-fit* strategy such that a subsequent call to `malloc` will fail, however it would succeed if either the best-fit or worst-fit strategy would have been used.

**Solution:** We choose a chunk of 8 blocks of memory and the following sequence of operations:

```
1:  id1 = malloc(2);
2:  malloc(1);
3:  id2 = malloc(1);
4:  malloc(1)
5:  free(id1);
6:  free(id2);
7:  malloc(1);
8:  malloc(2);
9:  malloc(2); // fails
```

This leads to the following memory configuration at line 8:

```
+-----------------------------+
| 1 | X | 1 | X | 1 |  2  | X |
+-----------------------------+
```

However, using the other two strategies, `malloc(2)` in line 9 does not fail. For best-fit, we obtain the following configuration at line 8:

```
+-----------------------------+
|  2  | 1 | 1 | 1 | XXXXXXXXX |
+-----------------------------+
```

For worst-fit, we obtain the following configuration at line 8:

```
+-----------------------------+
|  2  | 1 | X | 1 | 1 | XXXXX |
+-----------------------------+
```

(d) Briefly (i.e., in a few lines) discuss advantages and disadvantages you see with any of the above strategies.

**Solution:** As we have seen, no strategy is perfected in the sense that there always exist memory allocation patterns which lead to fragmentation patterns such that even though sufficiently many free blocks are available, they are not consecutively available. From a computational perspective, the first-fit strategy has the slight advantage that it does not have to traverse all free blocks in order to determine where to allocate memory.

**Notes on the correction:** There were 4+4+4+3=15 points available in this section.