EURECOM
Sophia Antipolis

Exam: Solutions
Operating Systems - OS

Ludovic Apvrille
ludovic.apvrille@telecom-paristech.fr

February, 8th, 2013

Authorized documents: Nothing! The grading takes into account the fact that you don't have any document with you.
A grade is provided for every question (beware: do organize your time). 1 additional point is given as a general appreciation, including written skills and readability.

This is only an example of one possible solution to get the maximum grade. So, you can get the maximum grade with a different solution. Also, contact me if you think my solution can be improved

1 Understanding of the course (5 points, ∼30 minutes)

1. What is the main purpose of swapping? Can a process be run by an Operating System if some of its pages are swapped out? [2.5 points]

   The main purpose of swapping is to have a larger storage facility to store data allocated by processes. That larger storage is commonly a hard disk, which is much slower than the main memory.
   If we assume a virtual memory managed by the Operating System, and supported by a MMU, then, all pages under manipulation by a running process must be swapped in before they can be used. Thus, if a process does not need some of its pages, yes, it can be run with some of its currently-not-used-pages swapped out.

2. What are the two techniques that are commonly used by device drivers to exchange information with devices? Explain the two, and explain in which situations they are efficient, or not. [2.5 points]

   The two main techniques are polling and interrupts.

Polling consists in periodically reading a given register (e.g., status register) of a device to know whether the latter has completed its operation, or not.
In the interrupt technique, the device asserts a hardware signal - called interrupt - to the microprocessor. The corresponding interrupt service routine set up by the Operating System at boot up will then be called.
Polling can be used when the delay between two operations is short and known: it avoids interrupting the processor. When that delay is likely to be long, or is unknown, interrupts give a chance to the Operating System to schedule other processes while waiting for the interrupt to occur.

2 Memory allocation (6 points, ∼40 minutes)

Memory allocated by Operating Systems is usually a multiple of a given memory page size. Operating Systems commonly store references to allocated pages in linked lists. Yet, programmers like to allocate a memory chunk whose size is not necessarily a multiple of memory pages handled by the Operating System. To do so, user-level libraries manages more fine-grained chunks of memory. For example, *malloc()* is a user-level library function which handles random sizes of memory allocations. That is, that library function allocates necessary pages using system calls (e.g., *mmap()*, *brk()*), and manages allocations within pages using its own data structures (e.g., linked lists).

1. Why isn't it the Operating System directly handling fine-grained allocations? [1 point]

   For efficiency reasons, the Memory Management Unit should manage lookup tables of reasonable size. Therefore, an Operating system can only handle pages and segments whose size is at least a few kbytes. Finer-grained allocations are thus managed at user-level e.g., by the libC (malloc, etc.).

2. Let's now consider the following code:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
  char *str;

  str = (char *)malloc(sizeof(char)*4);
  strcpy(str, "12345678\n");
  printf("str=%s", str);
  free(str);
}
```

   a. At execution, an error may occur when executing *strcpy*: could you explain it precisely, i.e., what is the cause of the error, and how it is detected at run time? In particular, you should explain why sometimes no error occurs. [2 points]

      The program first allocates 4 characters at address str. Then, 10 characters are

copied at address str, including the NULL character (end of the string). Either the address str+10 is in a page already allocated by the process, and in that case, no error occurs. Or, str+10 is outside the process address space, so when the process tries to access to that address, the MMU generates a page fault exception, which result in the process to be killed by the Operating System.

b. Again at execution, an error may also occur when executing *free*: could you explain it precisely, i.e., what is the cause of the error, and how it is detected at run time? [3 points]

The process was able to execute the strcpy instruction without being killed by the operating system, i.e. str+10 is part of the process address space. Thus, if an error occurs when calling free, it means that the function free could not be correctly executed, which probably means that the data structure managing the memory at user-level was corrupted by the 5 extra characters written at address str.

3 POSIX programming (8 points, ∼50 minutes)

Let's consider the following code.
(Memo: pthread_yield() causes the calling thread to relinquish the CPU.)

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t m;
pthread_t a, b;

void *f(void *param) {
  while (1)
    {
      pthread_mutex_lock(&m);
      printf("%s", param);
      pthread_mutex_unlock(&m);
      pthread_yield();
    }
}

int main() {
  pthread_mutex_init(&m, NULL);
  pthread_create(&a, NULL, f, "Hello ");
  pthread_create(&b, NULL, f, "World\n");

  pthread_join(a, NULL);
  pthread_join(b, NULL);
}
```

1. Give two possible traces of execution. [1 point]

First trace: the World process executes first. We assume that the yield operation provokes a switch to Hello, and then to World, etc.

World
Hello World
Hello World
etc.

Second trace: Hello starts first, but World takes longer to start, and yield does not work as we would like to, i.e., the same thread is given several times the processor again. One possible trace of this would be: Hello Hello Hello World
World
Hello Hello World
etc.

Of course, the second trace is more likely to occur than the first one.

2. Modify this code so as to have "Hello World" printed on each line. [2 points]

We need to alternate between the two threads, and yield is not the right way to do: we should rather use a global variable to do so, and two conditions variables in order to avoid threads to busy wait. To do so, we have split the code of threads in two different functions ("hello", "world"), and declared two condition variables ("toWorld", "toHello") and the global "alternate" variable. Finally, the code is as follows:

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t m;
pthread_cond_t toWorld, toHello;
pthread_t a, b;

int alternate = 0;

void *hello(void *param)
{
  while (1)
    {
      pthread_mutex_lock(&m);
      while (alternate != 0) {
        pthread_cond_wait(&toHello, &m);
      }
      printf("%s", param);
      alternate = 1;
      pthread_cond_signal(&toWorld);
      pthread_mutex_unlock(&m);
    }
}

void *world(void *param)
{
  while (1)
    {
      pthread_mutex_lock(&m);
      while (alternate != 1) {
        pthread_cond_wait(&toWorld, &m);
      }
```

```
        printf("%s", param);
        alternate = 0;
        pthread_cond_signal(&toHello);
        pthread_mutex_unlock(&m);
      }
}

int main()
{
  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&toHello, NULL);
  pthread_cond_init(&toWorld, NULL);
  pthread_create(&a, NULL, hello, "Hello ");
  pthread_create(&b, NULL, world, "World\n");

  pthread_join(a, NULL);
}
```

3. Enhance the code so as to print "Hello World" exactly 10 times. [1 point]

   We need a counter to do so. We have added a "cpt" counter to each thread, and we have defined "MAX_HELLO" to 10. Also, when the two threads terminate after 10 iterations, the main thread can do the join on the two threads. (I provide only part of the code)

```
...
#define MAX_HELLO 10

...

void *hello(void *param)
{
  int cpt = 0;
  while (cpt < MAX_HELLO)
    {
      pthread_mutex_lock(&m);
      ...
      pthread_mutex_unlock(&m);
      cpt ++;
    }
}

// similarly for world
...

int main()
{
  ...

  pthread_join(a, NULL);
  pthread_join(b, NULL);
}
```

4. Now, we would like to have two threads being able to print "Hello" and two printing "World". Synchronize those 4 threads so as to print exactly 10 times "Hello World". [4 points]

We need to start four threads. We also need also to ensure the synchronization between the four threads (for printing hello world correctly), and inside of each group of two, for not printing more that 10 times hello world. A first non working solution would be to simply execute the same code as previously with two variables nbOfWorld and nbOfHello (instead of the cpt variables) being global. Actually, this solution does not work because, when the last "hello" is being written, the other hello thread might already be waiting in the condition variable, and will thus be woken up later on by the last world thread, thus printing a 11th hello: we thus need to retest the number of hellos and worlds printed when waking up from waiting on the condition variable. finally, the solution is as follows.

```
#include <pthread.h>
#include <stdio.h>

#define MAX_HELLO 10

pthread_mutex_t m;
pthread_cond_t toWorld, toHello;
pthread_t a1, b1, a2, b2;

int alternate = 0;

int nbOfWorld = 0;
int nbOfHello = 0;

void *hello(void *param)
{
  while (nbOfHello < MAX_HELLO)
    {
      pthread_mutex_lock(&m);
      while (alternate != 0) {
        pthread_cond_wait(&toHello, &m);
      }
      if (nbOfHello < MAX_HELLO) {
        printf("%s", param);
        nbOfHello ++;
      }
      alternate = 1;
      pthread_cond_signal(&toWorld);
      pthread_mutex_unlock(&m);
    }
}

void *world(void *param)
{
  int cpt = 0;
  while (nbOfWorld < MAX_HELLO)
    {
      pthread_mutex_lock(&m);
      while (alternate != 1) {
        pthread_cond_wait(&toWorld, &m);
      }
      if (nbOfWorld < MAX_HELLO) {
        printf("%s", param);
        nbOfWorld ++;
      }
      alternate = 0;
```

```
      pthread_cond_signal(&toHello);
      pthread_mutex_unlock(&m);
    }
}

int main()
{
  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&toHello, NULL);
  pthread_cond_init(&toWorld, NULL);
  pthread_create(&a1, NULL, hello, "Hello ");
  pthread_create(&a2, NULL, hello, "Hello ");
  pthread_create(&b1, NULL, world, "World\n");
  pthread_create(&b2, NULL, world, "World\n");

  pthread_join(a1, NULL);
  pthread_join(b1, NULL);
  pthread_join(a2, NULL);
  pthread_join(b2, NULL);
}
```