Exam
Operating Systems - OS

Ludovic Apvrille
ludovic.apvrille@telecom-paristech.fr

February, 10th, 2012

Authorized documents: Nothing! The grading do take into account 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.

1 Understanding of the course (10 points, ∼60 minutes)

1. Consider a user-level process executing on a CPU. Give several reasons for this process to be blocked, and so to be preempted from the CPU by the OS? What are the conditions for that process to be granted again the CPU? [2.5 points]
   Here are the most common reasons for a process to become blocked when it is executing:

   - *The process has made a blocking system call, for example, reading a file from disk.*
   - *The process is trying to access an empty communication facility, e.g., trying to read a message from an empty message queue.*
   - *The process is trying to access a critical section that is already occupied by another process (e.g., performing a mutex_lock() call).*
   - *A segment or a page of the process has been swapped out, and the process now needs that segment or page.*
   - *The process wants to wait for a given period of time, or for the expiration of a timer.*

   To be granted the CPU again, a blocked process must first be "unblocked" and then be selected by the CPU scheduler. Unblocking reasons are respectively:

- *The system call unblocks, for example because requested data is now available*
- *The communication facility is not empty anymore*
- *The critical section can now be entered*
- *The segment or page was swapped in*
- *The time period has elapsed*

Note: When a process voluntarily relinquishes the CPU (e.g., by calling yield()), or is preempted because another process must be executed, the process is NOT blocked, but it is simply considered as runnable (but not running).

2. Recall what is the Shortest Job First scheduling policy, and explain its main advantages and drawbacks. [2.5 points]
   The SJF scheduling policy selects the process whose next execution time is the shortest. Therefore, this policy relies on the fact to know exactly those next execution times. If we assume that this information is known, this policy is very easy to implement. It is also quite fair - giving the CPU to the less greedy process - , and is optimal when all processes are ready at the same time.
   Unfortunately, this next execution time is generally not known. Therefore, it has to be predicted, using e.g. the exponential average approach. This is clearly a drawback of this policy.

3. What is the purpose of a page replacement algorithm i.e., in which conditions is it used by an OS? Do you remember or can you imagine a few algorithms? If so, explain them briefly. [3 points]
   Memory pages of processes can be swapped out, that is, moved to larger (but slower) memory, e.g. a hard disk drive. Whenever a process tries to access to a swapped-out page, the page must be first swapped in. If the main memory is full, another page must be swapped out before the swap in operation can be performed. A page replacement algorithm intends to select the "best" page to be swapped out.
   A few approaches: Least Recently Used intends to swap out pages that have not been used for the longest period of time. Not Recently Used works a bit the same way, except that usage on pages is considered only for a given period of time. Another policy is also to randomly select a page. As usual, a trade-off has to be found between complexity and efficiency, and according to the kind of processes being executed in the system (computations, interactive, etc.).

4. Provide the reasons why it is not a good idea to enter a critical section CS2 when already being in a critical section CS1. Give concrete examples to illustrate your response. [2 points]
   Nested critical sections can quite easily lead to deadlock situations, i.e. to situations where processes will never get access to the critical sections they are trying to enter in.
   Let's consider the following example with two processes P1 and P2: Process P1 enters

CS1. It is preempted, and Process P2 enters CS2. Then, P1 wants to enter CS2 and P2 CS1: they block each other forever.

2 Programming kernels (10 points, ∼60 minutes)

The Linux code provided at the end of this exercise is taken from an IBM webpage whose purpose is to illustrate the use of Linked Lists at kernel level. You goal is to answer the following questions regarding that code.

Note: all explanations for the code are provided here:
http://www.ibm.com/developerworks/linux/library/l-timers-list/index.html
I was obviously not expecting the whole content of this website as answers ;-)

1. Cite different reasons why linked lists are commonly used in kernels. In particular, could you cite modules of kernel that make use of them? [1.5 points]
   Linked lists(ll) are used by almost all kernel facilities. For example:

   - *ll are used by the OS process handler to store Process Control Blocks*
   - *ll are used by the OS memory handles to keep track of allocated and non allocated memory blocks*
   - *ll are used by OS Input/Ouput handlers to store output and input data (i.e., buffers)*
   - *ll are used by OS communication objects to store data elements (e.g., message queues)*

2. Apart from the list manipulation, give all the elements of the code that are specific to a (Linux) kernel code, and explain them. Also, explain **WHY** they are specific to kernel code. [1.5 points]
   First, the general structure of the code is specific to a Linux Kernel module. Indeed, it has no "main" function just like in a regular program, but it has a startup function (called init_module) and an exit function (called cleanup_module). Included files also generally differ from the ones of a regular program, for example, inclusion of kernel.h Then, the start and exit functions make use of kernel specific primitives. For example, kmalloc to allocate memory and printk to print a message in the system console. Those functions are specific of the kernel because they are implemented in a more efficient way (for example, kmalloc allocates only given size of memory), but with less protections (e.g., an error may result in a kernel panic).

3. Basically explain how the code works. To do so, give the comments you would put in that code to help readers/users of that code to understand it. [1.5 points]
   Header of the code: the system allocates two global lists: one for all objects, one for objects with an odd value. This code also illustrates the the use of list reference (full_list and odd_list fields of the data stucture).

Just before the startup function: The function allocates 10 data structure and stores all of them in the full list, and stores objects with an odd value in the odd list.
Just before the cleanup function: The value of objects stored in both lists are printed in the console, first, for the full list, and then for the odd list. Then, allocated objects are all disallocated, and entry from the full list are removed.

4. To facilitate the use of linked lists for kernel programmers, Linux defines a set of macros. For the following ones, your job is to try to guess their role. Justify your answer, otherwise no point is given. [2.5 points]

   a. LIST_HEAD
   b. list_for_each
   c. list_entry
   d. list_for_each_entry
   e. list_for_each_safe

   - *a: Defines the head of the list. For example, my_full_list is defined to be the head of a list.*
   - *b: It is a macro that goes through all elements of a list. It takes as argument a pointer to a list head and a pointer to the general list. From that code, it is clear that an object can be identified from the list head its data structure contains. This facilitates the fact to insert the object in multiple lists.*
   - *c: List entry returns the master object from a list reference (e.g., full_list).*
   - *d: Another macro to list each objects of a list. It takes as argument the list, the list reference of the data structure, and a pointer to the master object (my_obj).*
   - *e: Another macro to go through a list. This macro allows modifications on the list at the same time, for example, to remove elements of the list.*

5. Now, just assume that you have to program the same application in user mode. Explain the general approach, and give the main parts of the C code of that application. [3 points]

   - *Basic linked lists in user mode may be manipulated the same way: their head may be declared as global variables. Add and remove elements from a list are quite easy to implement. So, nothing really different for basic manipulations. Macros can probably be redefined to work the same way in user mode. For the "list_for_each_safe macro", we need to keep track of all elements to allow manipulations of the list while being parsed. To do so, we could simply first copy all elements of the list into another temporary list.*

- *Functions specific to the kernel must be redefined. For kmalloc: malloc, for kfree:free, and for prink: printf*
- *The most important issue consists in transforming the kernel module structure into a user application. Quite obviously, we need an entry point for the program (a "main") and a function corresponding to the clean_up, that can be called from an external process. We could for example use a software signal (SIG_USR1): when received, the signal handler disallocates elements of the to lists and the application terminates.*

Finally, the program may look like this: [Another answer for this question would also be to show a clear implementation of the lists macros]

```
#include <signal.h>

// Initialization of struct and list (similar)
...

void cleanup(int signo) {
// Same code as in cleanup module with kfree replaced with free,
// and printk replaced with printf
...
// Quit the application
exit(0);
}

int main(void) {
signal(SIGUSR1, cleanup);
//same code to initialize the two lists,
// apart from the use of malloc that replaces kmalloc
...

// Waiting for the signal
while(1) {
sleep(10000000);
}
}
```

(The code is on next page)

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/list.h>

MODULE_LICENSE("GPL");

struct my_data_struct {
  int value;
  struct list_head full_list;
  struct list_head odd_list;
};

LIST_HEAD( my_full_list );
LIST_HEAD( my_odd_list );

int init_module( void ) {
  int count;
  struct my_data_struct *obj;

  for (count = 1 ; count < 11 ; count++) {

    obj = (struct my_data_struct *)
            kmalloc( sizeof(struct my_data_struct), GFP_KERNEL );

    obj->value = count;

    list_add( &obj->full_list, &my_full_list );

    if (obj->value & 0x1) {
      list_add( &obj->odd_list, &my_odd_list );
    }

  }
  return 0;
}

void cleanup_module( void ) {
  struct list_head *pos, *q;
  struct my_data_struct *my_obj;

  printk("Emit full list\n");
  list_for_each( pos, &my_full_list ) {
    my_obj = list_entry( pos, struct my_data_struct, full_list );
    printk( "%d\n", my_obj->value );
  }

  printk("Emit odd list\n");
  list_for_each_entry( my_obj, &my_odd_list, odd_list ) {
    printk( "%d\n", my_obj->value );
  }

  printk("Cleaning up\n");
  list_for_each_safe( pos, q, &my_full_list ) {
    struct my_data_struct *tmp;
    tmp = list_entry( pos, struct my_data_struct, full_list );
    list_del( pos );
    kfree( tmp );
  }

  return;
}
```