"Operating Systems" Course

Final Examination – Fall 2007

February, 2008

Duration: 2h

ludovic.apvrille@telecom-paristech.fr

No document regarding Operating Systems (OS) or RTOS is allowed. Questions on OS or RTOS do take into account the fact that you don't have any document on that topic. The only documents allowed are the three slide sets and the paper that was given to you during the two following lecture sessions: January 14th and 28th. Having other documents (or communicating devices) with you shall be considered as a regular cheating procedure.
Answers should be as concise as possible. Also, you are free to answer either in **English** or in **French,** but please do not mix both!

I. Understanding of the course (7 points)

(a) What are the minimum hardware mechanisms on which an Operating System must rely to ensure protection between user processes? Clearly explain those mechanisms (just mentioning them is not enough) and which kind of protection they provide. (2 points)

(b) What are the actions taken by the Operating System when a user program makes a call to the *write()* system call? We assume that this call is performed to write data in a file. Provide an answer for the two following cases:
   a. Files are not cached in main memory. (1.5 points)
   b. Files are cached in main memory. (1.5 points)

(c) What are the techniques used in Real-Time Operating Systems to reduce the time between a timer's expiration and the notification of that expiration to a user program? (2 points)

II. Kernel-level programming (8 points)

When programming at kernel level, you may have noticed that it is impossible to use the *malloc()* C-library function, but only *kmalloc()*. Unfortunately, *kmalloc()* can only return blocks of memory which size is a power of 2 (equal or greater than 16B, and less or equal than 128kB). This means that when you allocate some memory with *kmalloc()*, you may have unused blocks since *kmalloc()* allocates more memory than required when you request sizes are not a power of 2. On the contrary, *malloc()* returns a block of memory of the exact required size.

(a) Explain the main steps that happen on UNIX Operating Systems when you start a program i.e. main actions that are executed (by the Shell from which the program is started, by the Operating System) when you type the name of an executable file in a shell until your first C instruction is executed. (2 points)

(b) We now assume that *kmalloc()* is a system call. Suppose that the implementation of *malloc()* C library can only rely on *kmalloc()* (remember: kmalloc can return only blocks of memory that are power of 2, equal or greater than 16B, and less or equal than 128kB). How could *malloc()* be implemented with limited waste of memory? I definitely don't ask you to provide the full C code of your *malloc()*, I just ask you to provide main algorithms and data structures that you may need to implement *malloc()* in a efficient way. (4 points)

(c) Now, you want to program the *free()* C-library function. Propose a short implementation of it that can work conjointly with your *malloc()* function. This function is only allowed to make calls to *kfree()* (we assume that this is a system call). (2 points)

III. Synchronizing tasks (4 points)

Microsoft Windows Vista provides an *EventPair* object to help support fast request/response message passing between *client* and *server* threads. *EventPair* synchronizes a pair of client / server threads. The server thread waits for a request by calling *Event-Pair::Wait()*. The client issues a request by first placing a request message in a shared memory location, and then by calling *EventPair::Handoff()*. *Handoff* wakes up the server thread and simultaneously blocks the client to wait for the reply. The server thread eventually responds by placing the reply message in another shared memory location and then it calls *Handoff*: this wakes up the client and simultaneously blocks the server which waits for the next request. Then, the client may consult the response.

(a) Show how to implement *EventPair* using semaphores. (2 points)

(b) Show how to implement *EventPair* using mutex and condition variables (2 points)