“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)

- *Protection mode in the CPU. Some instructions need to be privileged instructions i.e. instructions meant to be performed only by the OS. Examples of such instructions are the one to configure interrupts, to configure the timer, and to configure the MMU.*
- *Memory Management unit: a memory management unit offers the possibility to forbid the access to unauthorized addresses (RAM, I/O devices).*

(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)

- *First case: files are not cached in main memory. In that case, the operating system may first copy the provided data to an intermediate buffer. Then, a DMA transfer is performed from that buffer (or from the initial buffer) to the disk. During that transfer, the task is put on I/O wait. Then, once the transfer is completed, the task is moved back to a "runnable" state.*
- *In the second case, data are just directly copied to a memory location that corresponds to the cached file. Then, the task may continue its execution, it is not necessary to put it on I/O wait. The file will be written back to disk when the file is closed, or when the operating system needs more memory, or when the operating system halts. Or also, simply when the system is not loaded, the operating system may decide to write back the file to disk.*

(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)

The most important technique to reduce the time between a timer expiration and its notification to related user programs is to speedup the preemption of other currently running tasks, and also to reduce the computation performed in interrupt service routine, and more precisely in the one handling the timer. To speedup preemption, a well-known technique is to introduce preemptions points every n instructions in the kernel. That n is commonly close to 1000 to 2000 instructions. Note that reducing that number may imply a too important overhead on the kernel.

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)

The shell first forks to create a new process (fork, vfork(), etc.). Then, the new process environment is replaced by a new one corresponding to the program that the user wishes to start: the code and data of the corresponding program file are loaded into memory, the environment variables are loaded into the program, the heap and stack are initialized. Then, the C execution environment is started (initialization routines, loading of librairies). At last, the first instruction of the corresponding C file may be executed. Note

that dynamic libraries may also be loaded, either during the initialization of the program, or at execution step.

(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)

Since kmalloc() may allocate only power-of-2 blocks of memory, malloc() has to do its own memory management - relying only on kmalloc() for allocation purpose-. Malloc() must keep track of allocated blocks. To do so, we may use a linked list of all memory blocks – allocated or not – with a special tag saying for each block whether it is allocated or not. Then, when a call to malloc() is performed, there are two cases. (We assume that a memory request with malloc() cannot be more than 128KB):

- *First, an algorithm is applied to search for a continuous block of the right size, in the linked list of memory blocks. If such a block can be found, then, it is marked as allocated and then returned to the user. A first fit policy may be applied on the linked list to find that block.*
- *Second, if a block cannot be found, a 128KB block is allocated with kmalloc(), and added to the linked list. The first fit algorithm may be applied again, and is sure to return a memory block of the right size.*

Note that we have to manage blocks – at malloc level - of a minimal size to avoid a too big overhead (memory and computation overhead). There might be also other optimizations: allocate with kmalloc a block close to the required one when no other already allocated block is available, etc.

(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)

When a call is made to free, the corresponding malloced-block is marked as free, and may be merged with previous or next blocks of the linked list. Only blocks of the same kmalloced 128KB blocks may be merged together. Then, if a kmalloced block is totally free of any allocation (i.e. a 128KB block that was previously allocated with kmalloc()), that block may be removed from the linked list, and a kfree() may be performed on that block.

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)

We need two semaphores to do this, so as to make a signal between the client and the server, and between the server and the client. Let's call them S1 and S2, respectively. We denote by share1 and share2 the two shared memory locations.
We note T(S1)the function to try to get S1(i.e. decrement S1) et R(S1) the function to release S1 i.e. to increment S1.

Server	*Client*
S1 = 0 *S2 = 0*	
while(1) { *T(S1)*	
	share1 = x *R(S1)* *T(S2)*
y = share1 *…* *share2 = z* *R(S2)*	
}	
	y = share2 *…*

Thus,
Event-Pair::Wait().is:
Event-Pair::Wait(S) {
T(S)
}

EventPair::Handoff(S, share, data) {
share = data;

R(S)
}

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

In that case, we use one mutex (called mut), and two condition variables: wisml1 (written in memory location one) and wishl2.

Server	*Client*
while(1) {	
mutex_lock(m)	
wait(&wisml1, &m)	
	mutex_lock(m)
	share1 = x
	signal(&wisml1)
	wait(&wisml2, &m)
y = share1	
…	
share2 = z	
signal(&wisml2)	
mutex_unlock(m)	*y = share2*
}	*…*

We may deduce from that code the behavior of Event-Pair::Wait() and EventPair::Handoff, as done in question (a).