"Operating Systems" Course

Final Examination – Fall 2008

February, 2009

Duration: 2h

ludovic.apvrille@telecom-paristech.fr

Documents regarding Operating Systems (OS) or RTOS are not 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 that were given to you during the two following lecture sessions: January 22th and February 1st. Having other documents (or communicating devices) with you shall be considered no less than 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 is the difference between the supervisor mode of a microprocessor and the administrator / root rights provided by an operating system? Are those two modes related? If yes, closely explain in what they are related. (2 points)

The supervisor mode of a microprocessor offers (hardware) instructions that cannot be executed in user mode. Those instructions are generally dedicated to the communication and configuration of hardware devices, including the MMU.
The administrator / root right is a right at operating system level i.e. it provides a user with privileges for executing specific commands (e.g. poweroff) and configuring the operating system.
The two modes are somehow connected in the fact that many operating system services, running in supervisor mode, are executed with the root rights. Also, if a user can execute its own code in supervisor mode, that user could for example manipulate the memory of the system so as to obtain the root right.

(b) Explain in 5 lines at most the role of a driver. Then, explain why drivers frequently need to rely on buffers for managing devices. At last, explain why,

when removing a USB key from a computer running Linux (or Solaris, Windows, etc.), one must first "detach the device". (3 points)

A driver is a piece of software code running in supervisor mode. Its role is to simplify the access to hardware elements external to the CPU (devices). To do so, drivers offer simple interface to users of devices (e.g. read, write functions) and transform those basic calls (e.g. read, write) to a set of low-level instructions (setting and reading the value of registers, etc.). Drivers also ensure device error management.
Since user application wanting to read / write from / to a device may need to write data of a given size to a speed which may be different from the one of the underlying device, buffers are necessary.
When removing a USB key from a computer, data to be written to the key may still be pending: detaching the device makes it possible to first flush those data before removing the key.

(c)What are preemptions points used for? What happens if they are put too frequently in the operating system? On the contrary, what happens if they are not frequently put in the operating system? Why are preemption points more particularly at stake in real-time operating systems? (2 points)

Preemptions points are entry points in the code of operating systems in which the kernel can be preempted i.e. kernel data structures can safely be accessed from those points.
If those points are put too often in the code, then, the kernel must often be sure that all data structures can be accessed, leading to a great overhead because complex algorithms implemented by the kernel must be cut into small sections of codes.
If those points are not frequently put in the code, this leverages the kernel overhead, but the kernel itself cannot be preempted as often: it is less reactive. And so, when a urgent task is released – i.e. a task which deadline is close – it may have to wait longer for the kernel to reach a preemption point: it may miss its deadline.

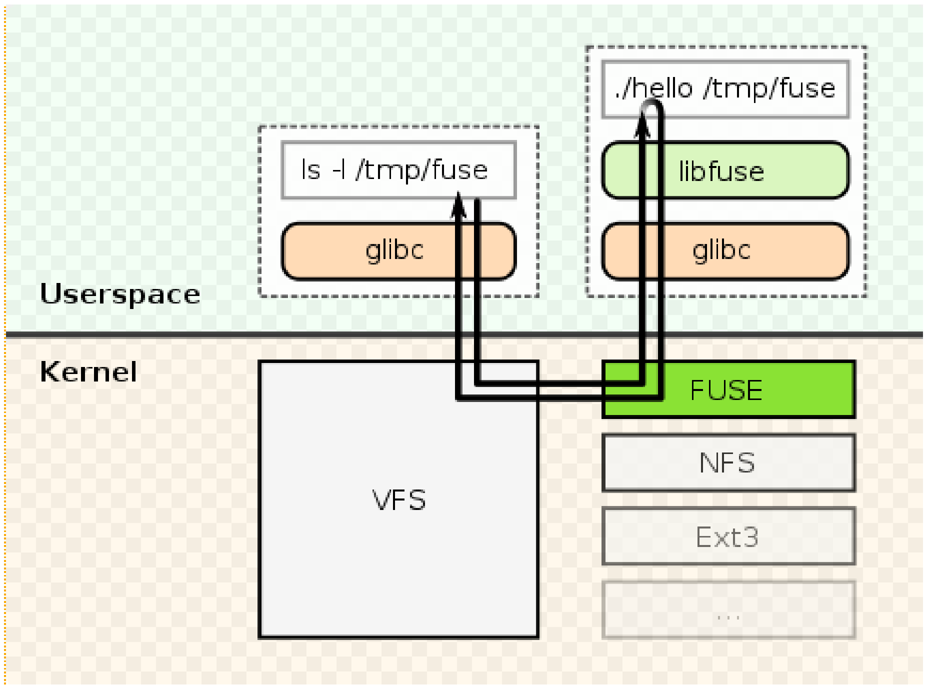
II. File systems in user space (7 points)

The following text and figure are taken from Wikipedia.
Filesystem in Userspace (FUSE) is a loadable kernel module for Unix-like computer operating systems, that allows non-privileged users to create their own file systems without editing the kernel code. This is achieved by running the file system code in user space, while the FUSE module only provides a "bridge" to the actual kernel interfaces. FUSE was officially merged into the mainstream Linux kernel tree in kernel version 2.6.14.

FUSE is particularly useful for writing virtual file systems. Unlike traditional filesystems, which essentially save data to and retrieve data from disk, virtual filesystems do not actually store data themselves. They act as a view or translation of an existing filesystem or storage device. In principle, any resource available to FUSE implementation can be exported as a file system. See Examples for some of the possible applications.

Released under the terms of the GNU General Public License and the GNU Lesser General Public License, FUSE is free software. The FUSE system was originally part of A Virtual Filesystem (AVFS), but has since split off into its own project on SourceForge.net.

FUSE is available for Linux, FreeBSD, NetBSD (as PUFFS), OpenSolaris and Mac OS X.



You may consider, in the following questions, that a file system is a tree whose nodes are either files or directories. A file cannot have a subtree. On the contrary, a directory node has a subtree.

(a) Recall what is the *Virtual File System*, and what are the main system calls to use it. (1 point)

The virtual file system of an operating system is meant to offer a uniform access to files, or data, stored on very various devices, such as USB keys, CDs, hard disks, NFS, etc..
Main system calls: open, read, write, close.

(b) Explain, when you want to implement a new file system based on FUSE, what has to be implemented in the kernel, and what has to be implemented in the userspace? (1 point).

Nothing has to be implemented at kernel-level! Indeed, FUSE has been introduced so that user can develop their own file systems as a user-level application, i.e. no modification must be made to the operating system.
So, everything must be implemented in user-space, including the interface with FUSE.

The following text has been taken from Linux Gazette, January 2007:

curlftpfs: Mount FTP servers

This is something that I really love! Accessing a FTP server as though it's contents were on directories on your own computer! Just get `curlftpfs` *from the curlftpfs page, install it using the standard* `./configure; make; make install`*, or install the package available for you distribution, and just do something like this:*

```
[kumar@debian ~] mkdir IITM_Mirror
[kumar@debian ~] curlftpfs ftp.iitm.ac.in IITM_Mirror/
[kumar@debian ~] cd IITM_Mirror/
[kumar@debian ~/IITM_Mirror] ls
...
README          debian...
```

That's it! I have used `IITM_Mirror` *as the mount point for the mirror. You can now mount FTP servers, even with password login, so that you can do uploads as well. Do* `curlftpfs -h` *for learning how to mount servers with login for write access and using proxies. To unmount, use* `fusermount -u ~/IITM_Mirror`*.*

In the next questions, we consider that ftp offers the following commands:

- **open**, to open a connection with a remote computer [the robot]
- **quit**, to close the connection with the remote computer
- **ls** to list remote files
- **put**, to put a file on the remote filesystem
- **get**, to get a file from the remote file system to the local filesystem
- **mkdir** to create a directory
- **rmdir** to remove an empty directory
- **cd** to change of directory on the remote filesystem

Also, we consider that a user may do the following six system calls on the virtual file system: **ls**, **write**, **read**, **mkdir**, **rmdir**, **cd**.

(c) Explain what probably happens in the operating system and in the curlftpfs application for each commands of the example (mkdir, curlftpfs, etc.). (1 point).

mkdir IITM_Mirror
It creates a new directory on the local filesystem

curlftpfs ftp.iitm.ac.in IITM_Mirror/
This command relies on curlftpfs - implemented in user-space - to mount a remote file system, using ftp connections, to a local directory (*IITM_Mirror*)

cd IITM_Mirror/
***Changes on the local shell of directory so as to go in IITM_Mirror. A ftp connection may be opened to* ftp.iitm.ac.in.**

ls
If the ftp connection to **ftp.iitm.ac.in.** *was not previously opened, curlftps opens a new ftp connection. Then, a "ls" command is performed, by ftp, on the remote site. At last, result of the remote "ls" command is printed on the local shell (or more precisely, to the default output of the local shell).*

(d) For the 6 system calls commands listed above, explain how they could be implemented in C at the FUSE application level (i.e. in *culrftpfs*). I do not ask you to provide the full code, but rather a sketch of what could be done. You may obviously, in your C files, make calls to ftp commands (get, put, etc.). Also, in your implementation, do take into account the fact that several commands may be sent at the same time to the *curlftpfs* FUSE application (explain how you handle this). Also, for commands with similar implementation, you may just provide only one implementation and precise a list of commands to which this implementation applies. At last, to propose an implementation, do assumptions on interfaces provided by *libfuse* (4 points).

ls	*Performs a "ls", then, prints the result of that command to the default output.*
cd	*Opens an ftp connection when entering the mount point of a curlftpfs to the corresponding ftp site. Switch to binary mode. When making a cd that leaves the mountpoint, the ftp connection might be closed, and files stored locally (see write command) may be put on the remote ftp site.*
rmdir	*Relies on the "rmdir" ftp command.*
mkdir	*Relies on the "mkdir" ftp command.*

write	*Several approaches are possibl, including:*
	** buffer all data in a local file, then, when a close is performed on the file, make a put on the remote file system.*
	** Or: each time a write is performed on a remote file, read the content of the file (get), append that content to the file by doing a put.*
	
	Let's give a sketch of the code for the first solution (more complex one):
	write(file, data, size):
	fd = open(localfile)
	fseek(fd, SEEK_END);
	write(fd, data, size);
	close(fd);
	
	When a close occurs:
	close(file)
	put(localfile)

read	*Much easier than write: it simply relies on a get to that file. Then,*

	the file is stored on local memory (hard disk, etc.), and then, each read command can be handled simply by remembering the last read command on that file.

III. Scheduling tasks (5 points)

Question 1:

Consider a set of 3 independent tasks (Task P1, Task P2, Task P3) running on a processor. These tasks are defined by their periods (T) their worst case execution times (called capacity) (C)
P1: T1=4, C1=1
P2: T2=6, C2=2
P3: T3=10, C3 = 3
Priorities are assigned according to Rate Monotonic.

(a) **Compute the Hyperbolic bound of this task set. Can you conclude by this method on the schedulability?**
(b) **Verify the schedulability of this task set**

Question 2:

Consider the following task set:
Task P1 : T1 = 10, C1 = 4,
Task P2 : T2 = 15, C2 = 3,
Task P3 : T3 = 20, C3 = 4,

P1, P2 and P3 are using the same resource S in the following manner:

P1 : +S-	(1 unit of time using S)
P2 : +SSSSS-	(5 units of time using S)
P3: +SSS-	(3 units of time using S)

We assume S is taken according to PIP (Priority Inheritance Protocol) rules

(a) **Compute the blocking factor Bi of each task**
(b) **Verify the schedulability of this task set under Rate Monotonic**