Course Operating Systems

Final Examination – Fall 2005

February, 2006

Duration: 2h

ludovic.apvrille@enst.fr

Authorized documents are limited to course's slides and to the code you produced during lab sessions. Basic calculators are authorized but quite useless. Answers should be as concise as possible. Also, you are free to answer either in **English** or in **French,** but please do not mix them!

Note: There are several possible solutions, this is no more than a sample. Commonly, you spend too much time on first questions, and so, you lack times for last questions.

I. Understanding of the course (7 points)

(a) What are the differences between the polling technique and the interrupt technique, when communicating with devices? For which kind of transfers / devices is the polling technique more efficient? Less efficient? (2 points)

Polling technique: a device is periodically scanned by its device driver to known whether it is busy, or not. If it not busy, the device shall accept new command, and data might be read from that device.

Interrupt technique: a device informs the computer system about the completion of a command, or the availability of data.

Obviously, the polling technique is efficient only when the device has a regular way of performing actions, for example, a parallel port. But for devices with unpredictable behaviors, just like mice, the interrupt technique is more efficient, because no time is being lost scanning the device.

(b) Explain what happened in the computer system when you get a "segmentation fault – core dumped" when executing an application. (2 points)

A segmentation fault is a trap due to the fact that a process has tried to access to a forbidden memory address. The operating system intercepts this trap, and stops the faulty process. The launcher of the process is informed about this killing, and a message "segmentation fault – core dumped" is generated. A core file is also produced by the operating system. This file represents memory space of the process before the faulty operation was performed. It is generated for debugging purpose, and might be analyzed with a tool like gdb.

(c) The course has illustrated the use of condition variables over a producer / consumer example. First, explain what are "mutex" objects, and condition variables. Then, imagine a quite different programming example in which condition variables may be useful. Explain this example, and provide only most relevant parts of its code. (3 points)

A mutex is an object managed by operating systems, and that offers a way to provide mutual exclusion. A condition variable is also an object managed by operating systems. A variable condition makes it possible to sleep, or to send a signal. For both operations, a mutex must have been acquired first.

Such objects may be used for synchronizing threads. For example, suppose we have two threads that must "synchronize" at a given point of their computation i.e. we want to be sure that thread1 has finished algo11() before thread2 executes algo22() (we assume go is a shared variable initialized to 1)

Thread 1 (id = 1)	*Thread 2 (id = 2)*

algo11(); *pthread_mutex_lock(&my_mutex));* *go ++;* *pthread_cond_signal(&go_next);* *pthread_mutex_lock(&my_mutex));* *algo12();*	*algo21();* *pthread_mutex_lock(&my_mutex));* *while(go != 2) {* *pthread_cond_wait(&go_next, &my_mutex);* *}* *algo22();*

II. Memory management in Linux Kernel (6 points)

In annex is provided a *HowTo* Linux document. Most customizations of Linux kernels are provided in that kind of documents. The provided document addresses the problem of transferring data from and to memory. Read it carefully before answering to next questions.

(a) Provide a short summary of this *HowTo* (a few lines).

This how-to proposes addresses the bounce buffers issue under Linux. Indeed, with no specific patch, devices of computer system running Linux 2.4 may not directly copy data to high memory leading to performance drawback and memory starvations. A patch is proposed to solve this.

(b) Explain, in general, what is a DMA transfer.

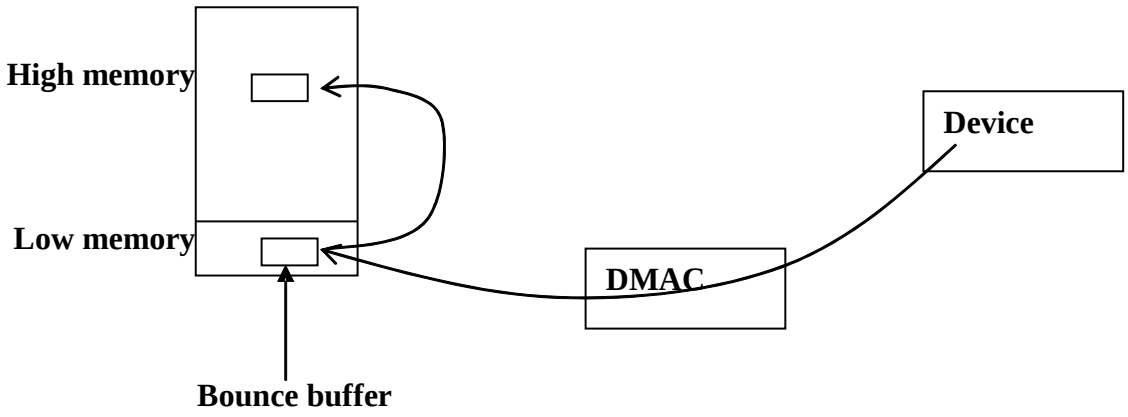
A DMA transfer is a transfer from a device to memory, or reciprocally, which involves the main CPU of the system only to program the transfer, and not to realize it. A DMA transfer is achieved by a DMAC, a DMA controller. A typical DMS transfer is done as follows:

1. *The OS (or a device driver running in kernel mode) programs the DMAC with the source and destination devices, and the number of bytes to transfer*
2. *The DMAC performs the transfer, and decrement a byte counter each time a byte has been transferred*
3. *When the transfer is finished, the DMAC informs the OS (or the device driver) using an interrupt.*

(c) Explain what is the main issue stated in this *HowTo*? Using figures might help you to explain it.

This how-to presents the limitations of using bounce buffers when transferring data from a device to the high memory (over 1GB). Indeed, the low addresses are reserved for the operating system, whereas upper addresses are reserved for the address spaces of processes.

The problem, as described, is summarizes by the following figure:



1. *First, data are transferred, by DMA, from a device to low memory*
2. *Then, data are copied from low memory to high memory.*

This results in a performance drawback, to a waste of memory, and potentially to an impossibility to make the transfer if not enough low memory is free.

(d) Suppose you have a PC on which you want to solve these issues. What are all different steps you have to take on your system? Also, try to think about other steps not provided in this document.

Various steps are provided in the how-to:

1. Download the patch from the provided website, and install it.
2. Change the kernel configuration as explained in the how-to. This obviously include to download and install kernel sources, and to launch a command such as "make menuconfig" from the place where sources have been uncompressed.
3. Compile the new kernel, install it and reboot.

Other steps are very likely to be performed:

1. For all device drivers no listed in the how-to, check whether they can work efficiently with the patch.
2. Thus, install new device drivers if necessary. To do so, go to vendors' website to have more information
3. Reprogram device drivers from sources if vendors no do provide efficient ones i.e. drivers that can use features provided by the patch.

III. Tasks sharing several resources (6 points)

Consider the following 5 tasks: P1, P2, P3, P4. Assume that Prio(P1)>Prio(P2)>Prio(P3)>Prio(P4)>Prio(P5) where Prio(P) is the priority of task P.

Task	Priority	Execution Sequence	Release time
P1	**High: 7**	**EE+S-E**	**7**
P2	**5**	**E+QQ-E**	**5**
P3	**3**	**E**	**4**
P4	2	**E+S+Q-S-E**	**2**
P5	**Low: 1**	**E+QQQQ-E**	**0**

These tasks share two resources S and Q as indicated in the table above.

We consider the following algorithm (named "stack based protocol"):

- The system is maintaining a variable V = current ceiling (as defined in the Priority Ceiling Protocol)
- When a task P is released, it is blocked from starting execution if Prio(P) is less or equal to V.
- When a task P requests a resource S, it is allocated.

(a) Explain why a deadlock cannot occur

(b) What are the values of the Priority Ceiling of Q and S?

(c) Draw the evolution of the value of V over the time interval [0, 21]

(d) Draw the execution of the above tasks over the time interval [0, 21]

(e) What would be the execution if we had used Highest Lock instead of the stack based protocol?

(f) For each task give the blocking factor under Highest Lock assumptions.

(g) How would you compute the blocking factors if we use the stack based protocol?

ANNEX: Excerpt of the *HowTo Linux* on "Patch to avoid bounce buffers"

Avoiding Bounce Buffers

This section provides information on applying and using the bounce buffer patch on the Linux 2.4 kernel. The bounce buffer patch, written by Jens Axboe, enables device drivers that support direct memory access (DMA) I/O to high-address physical memory to avoid bounce buffers.

This document provides a brief overview on memory and addressing in the Linux kernel, followed by information on why and how to make use of the bounce buffer patch.

Memory and Addressing in the Linux 2.4 Kernel

The Linux 2.4 kernel includes configuration options for specifying the amount of physical memory in the target computer. By default, the configuration is limited to the amount of memory that can be directly mapped into the kernel's virtual address space starting at PAGE_OFFSET. On i386 systems the default mapping scheme limits kernel-mode addressability to the first gigabyte (GB) of physical memory, also known as low memory. Conversely, high memory is normally the memory above 1 GB. High memory is not directly accessible or permanently mapped by the kernel. Support for high memory is an option that is enabled during configuration of the Linux kernel.

The Problem with Bounce Buffers

When DMA I/O is performed to or from high memory, an area is allocated in low memory known as a bounce buffer. When data travels between a device and high memory, it is first copied through the bounce buffer.

Systems with a large amount of high memory and intense I/O activity can create a large number of bounce buffers that can cause memory shortage problems. In addition, the excessive number of bounce buffer data copies can lead to performance degradation.

Peripheral component interface (PCI) devices normally address up to 4 GB of physical memory. When a bounce buffer is used for high memory that is below 4 GB, time and memory are wasted because the peripheral has the ability to address that memory directly. Using the bounce buffer patch can decrease, and possibly eliminate, the use of bounce buffers.

Locating the Patch

The latest version of the bounce buffer patch is *block-highmem-all-18b.bz2*, and it is available from Andrea Arcangeli's -aa series kernels at http://kernel.org/pub/linux/kernel/people/andrea/kernels/v2.4/.

Configuring the Linux Kernel to Avoid Bounce Buffers

This section includes information on configuring the Linux kernel to avoid bounce buffers. The Linux Kernel-HOWTO at http://www.linuxdoc.org/HOWTO/Kernel-HOWTO.html explains the process of re-compiling the Linux kernel.

The following kernel configuration options are required to enable the bounce buffer patch:

Development Code - To enable the configurator to display the `High I/O Support` option, select the `Code maturity level options` category and specify "y" to `Prompt for development and/or incomplete code/drivers`.

High Memory Support - To enable support for physical memory that is greater than 1 GB, select the `Processor type and features` category, and select a value from the `High Memory Support` option.

High Memory I/O Support - To enable DMA I/O to physical addresses greater than 1 GB, select the `Processor type and features` category, and enter "y" to the `HIGHMEM I/O`

`support` option. This configuration option is a new option introduced by the bounce buffer patch.

Enabled Device Drivers

The bounce buffer patch provides the kernel infrastructure, as well as the SCSI and IDE mid-level driver modifications to support DMA I/O to high memory. Updates for several device drivers to make use of the added support are also included with the patch.

If the bounce buffer patch is applied and you configure the kernel to support high memory I/O, many IDE configurations and the device drivers listed below perform DMA I/O without the use of bounce buffers:

aic7xxx_drv.o, aic7xxx_old.o, cciss.o, cpqarray.o, megaraid.o, qlogicfc.o, sym53c8xx.o