EURECOM
Sophia Antipolis

Exam
Operating Systems - OS
Solution

Ludovic Apvrille
ludovic.apvrille@telecom-paristech.fr

February, 2017

Authorized documents: Nothing! The grading takes into account the fact that you don't have any documents with you.
A grade is provided for each question (beware: be sure to organize your time with regards to the grading policy). 1 additional point is given for general appreciation, including writing skills and readability.

1 Course knowledge (5 points, ∼20 minutes)

a. If you were to design and program a complex application (e.g., Firefox), would you rather decide to use multiple processes, multiple threads, or both? Discuss the pros and cons of these three alternatives. [1.5 points]

If we assume the OS supports kernel threads, then using processes and threads offer the same degree of parallelism. The main difference between the two is the memory isolation between processes, which could be of interest for isolating the different "tabs" of a browser. Yet, this isolation makes it more difficult to share memory between processes than between threads. Therefore, all elements of Firefox that would need to easily share memory (e.g., the network connections for getting the elements of a tab) should be implemented as threads.

b. What is a deadlock? What are the techniques that are used in general-purpose Operating Systems (Windows, MacOS, Linux) to handle processes that are in a deadlock situation? [1.5 points]

A process is in a deadlock situation whenever it is blocked waiting for a resource that will never be available. A typical deadlock situation for a process P is to wait

for a mutex that will never be released because the process that holds the mutex is in an infinite loop, or is itself waiting for a resource that P holds.
Usually, OS do nothing to detect deadlock situations nor to recover from these situations. OS for embedded systems may use watchdogs to detect those situations, and usually enter a fail-safe mode from which the faulty processes will be killed while ensuring minimal system operations.

c. First, explain what are swap-out and swap-in operations, and in which case a swap-in operation must be performed. Then, explain why quite frequently that before a swap-in operation can be performed, a swap-out operation must be performed first. [2 points]

Swap-in / Swap-out operations consist in moving data from disk to RAM, and from RAM to disk, respectively. Whenever a process that is running needs data which has been swapped out, a swap-in operation must be performed before the process can access to this data. If a swap-out operation was performed in the system, this probably means that the RAM is full, and therefore, the swap-in operation is likely to fail if a page/segment of the same size is not swapped-out before the swap-in operation.

2 Page fault (11 points, ∼60 minutes)

For the following questions, using a sample code could facilitate your answer, especially for the last two questions.

a. What is a page fault? Give the main two reasons why it can occur. [1 point]

A page fault corresponds to a trap that is generated by the MMU whenever the CPU makes a request to an invalid address. The two reasons for an address to be invalid are:

(a) A process has tried to access to an address outside its process address space
(b) The corresponding page/segment has been swapped out

b. What are the actions taken by an Operating System when a page fault occurs? [2 points]

The OS catches the trap. Then, it determines if page fault is due to an out-of-address-space request — in that case, the process is killed by the OS — or because of a swap-out operation. In the latter case, the process that has provoked the fault is blocked. Then, the OS performs a swap-in operation (it may also have to perform a swap-out operation first). Once the swap-in has been completed, the process is put back to runnable mode.

c. We now assume a system in which the swapping facility has been disabled. Give a C code that will always provoke a page fault, and give another code that may provoke a page fault, but that will not always provoke a page fault. [2 points]

To provoke a page fault, we can access decremental addresses until the process is killed. Pseudo code:

```
int cpt;

int main() {
  int a[1];
  while (1) {
    a[cpt--] = 1;
  }
}
```

Note that this code could erase the stack/heap. Also, if it were to erase the value of cpt, then the code could fail to provoke a page fault. Also, a simple access to address 0x0 would work.
To provoke a page fault sometimes, we can pick up a random address, or use the size of a page (e.g., 4096)

```
int a[1];
int x = random(0, MAX_INT);
a[-x] = 1;
```

d. Assume two processes sharing code with shared memory. Explain the actions that are executed in the Operating System when:

(a) the shared memory area is created [1 point]
The OS creates a new IPC object that represents shared memory area, and allocates this area in the physical memory.

(b) the shared memory area is attached to one of the two processes [1 point]
When a process makes e.g. a mmap of the shared area to its address space, the OS updates the MMU so that the virtual addresses of the process are mapped to the corresponding physical memory.

(c) one of the two processes accesses the shared memory area. [1 point]
There, the OS does nothing! (and this is better that way, otherwise memory accesses would be verrryyyy slooooow). Indeed, the memory access of the process is verified as valid by the MMU.

e. Assume two threads of two different processes share part of their memory. In which cases do we have to use synchronization mechanisms e.g. *mutex_lock* / *mutex_unlock*? [1.5 points]

We need to use synchronization mechanisms if we expect read operations to be synchronized with write operations. For example, assume we have two processes:

- *P generates temperature values (e.g., P could be the driver of a sensor)*
- *Q reads the temperature values*

synchronizing P and Q could guarantee that Q always reads a fresh value. If we don't want to force Q to read fresh values, then synchronization techniques are not necessary. With synchronization techniques, we could program P and Q that way:

P

```
while 1
   lock
   temp = newValue()
   fresh = true
   unlock
```

Q

```
while 1
    lock
    if (fresh)
       print(temp)
       fresh = false;
    unlock
```

f. Continuing the previous question, in which synchronization cases do we have to combine the use of condition variables with *mutex_lock* / *mutex_unlock* ? [1.5 points]

Condition variables are useful to improve performance whenever in a synchronization scheme, readers/writers may not always be able to work on a shared elements once they have entered into a critical section. For example, the performance of the code of the previous question can be enhanced with conditions variables. Indeed, Q may enter in the locking facility when "fresh" is true, and so, it perform useless lock/unlock operations . Ideally, that would be a better programming approach if P could inform Q that the value of "fresh" has changed: conditions variables are made for this.

P

```
while 1
   lock
   temp = newValue()
   fresh = true
   notify(freshC)
   unlock
```

Q

```
while 1
    lock
    while (!fresh)
       cond(lock, freshC)
    print(temp)
    fresh = false;
    unlock
```

3 Linux kernel (7 points, ∼40 minutes)

a. What is the interest of DMA facilities? In which cases are they used by an Operating System? [1 point]

A DMA is a hardware facility that can handle memory transfers from the RAM to the memory of devices, and from the memory of devices to the RAM. Since its configuration takes several clock cycles, DMA transfers are usually reserved for large memory transfers. While DMA transfers are under execution, the CPU can continue to execute instructions.
OS are responsible for programming DMA transfers (or allowing devices to program DMA transfers), and to get the notification of completed DMA transfers e.g., to unblock a process.

b. The following code is taken from "dma.c" in the Linux kernel sources. Explain in which context it could be used. Also, explain this code. What are the comments you would add to this code? [1.5 points]

This function is called whenever there is a request for a new DMA transfer for a given DMA channel (dmanr) and for a given device (device_id). Other comments were added to the code with a "//". Comments with "/" are original comments. Note that telling me that comments are useless for this function is fine, except I guess for the header which is before the function.*

```
1  // Allocation of a DMA channel for a given device
2  int request_dma(unsigned int dmanr, const char * device_id)
3  {
4       // Checking if the DMA channel is valid
5          if (dmanr >= MAX_DMA_CHANNELS)
6                  return -EINVAL;
7
8       // Locking the DMA channel. Returns an error if it was already busy.
9          if (xchg(&dma_chan_busy[dmanr].lock, 1) != 0)
10                 return -EBUSY;
11
12        // The locked DMA channel can now be reserved for the device
13         dma_chan_busy[dmanr].device_id = device_id;
14
15         /* old flag was 0, now contains 1 to indicate busy */
16         return 0;
17 } /* request_dma */
```

c. Similarly, comment the following function. [1.5 points]

This function frees a DMA channel that was previously requested and locked

```
1  void free_dma(unsigned int dmanr)
2  {
3        //Checking if the DMA channel is valid. If not, issue a kernel warning
4          if (dmanr >= MAX_DMA_CHANNELS) {
5                  printk(KERN_WARNING "Trying to free DMA%d\n", dmanr);
6                  return;
7          }
8
9       // unlock the DMA channel. If the channel was not locked, issue a kernel warning.
10         if (xchg(&dma_chan_busy[dmanr].lock, 0) == 0) {
11                 printk(KERN_WARNING "Trying to free free DMA%d\n", dmanr);
12                 return;
13         }
14
15 } /* free_dma */
```

d. The following function creates a new message queue in the kernel for IPCs (file msg.c). Comment the main steps of this function. For your information, "RCU" stands for "Read Copy Update" which is a facility of the Linux Kernel that allows concurrent reads and modifications on the same data structure. [3 points]
Note: This question can be considered as a bonus since the exam is graded out of 23.

The function creates a new message queue, it allocates the necessary memory, configures the parameters — including the sub-queues of messages / senders / receivers, and finally add the object to the system. More comments below:

```
1  static int newque(struct ipc_namespace *ns, struct ipc_params *params)
2  {
```

```
3           struct msg_queue *msq;
4           int retval;
5           key_t key = params->key;
6           int msgflg = params->flg;
7
8         // Allocating a new message queue within the OS memory. If it fails, return an error
9           msq = kvmalloc(sizeof(*msq), GFP_KERNEL);
10          if (unlikely(!msq))
11                  return -ENOMEM;
12
13          msq->q_perm.mode = msgflg & S_IRWXUGO;
14          msq->q_perm.key = key;
15
16          msq->q_perm.security = NULL;
17
18        // This function probably checks for security aspects
19        // If the check fails, the queue is disallocated
20          retval = security_msg_queue_alloc(msq);
21          if (retval) {
22                  kvfree(msq);
23                  return retval;
24          }
25
26          // Initializing the queue parameters to default value
27        // That includes the configuration of the HEAD of the sub-queues
28       //  (messages, receivers, senders)
29          msq->q_stime = msq->q_rtime = 0;
30          msq->q_ctime = ktime_get_real_seconds();
31          msq->q_cbytes = msq->q_qnum = 0;
32          msq->q_qbytes = ns->msg_ctlmnb;
33          msq->q_lspid = msq->q_lrpid = 0;
34          INIT_LIST_HEAD(&msq->q_messages);
35          INIT_LIST_HEAD(&msq->q_receivers);
36          INIT_LIST_HEAD(&msq->q_senders);
37
38          /* ipc_addid() locks msq upon success. */
39          retval = ipc_addid(&msg_ids(ns), &msq->q_perm, ns->msg_ctlmni);
40          if (retval < 0) {
41                  call_rcu(&msq->q_perm.rcu, msg_rcu_free);
42                  return retval;
43          }
44
45          // The object is unlocked once it has been added
46        // Indeed, addid locked the queue
47          ipc_unlock_object(&msq->q_perm);
48          rcu_read_unlock();
49
50          return msq->q_perm.id;
51  }
```