Dept. Comp. Sc. **Operating Systems**
Vrije Universiteit **01.07.2005**

Part I

This part covers the same material as the midterm exam.

1a Explain the difference between user mode and kernel mode. *5pt*

In user mode, execution is restricted in that not all instructions can be executed, nor can all registers or memory locations be accessed. In contrast, in kernel mode, execution can make use of all facilities offered by the hardware.

1b MINIX is organized as a client-server operating system. What does this mean? *5pt*

It means that MINIX consists of a (relatively small) kernel whose main task is to establish communication between client applications and specific service programs. These service programs run as normal processes (i.e., in user mode), but there is no way that a client application can directly communicate with them: the kernel always sits in between.

2a If a CPU scheduler is implemented as a separate process, when will it be executed? *5pt*

In principle, every time an interrupt occurs, or when the current process executes a blocking statement, context is switched from the current process to the scheduler.

2b Name one important advantage of letting an OS kernel manage threads. *5pt*

Perhaps the most important reason is that when a thread does a blocking system call, the operating system can schedule another thread from the same process to continue. In other words, there is no need to block the entire process.

2c Processes are generally subject to preemptive scheduling, whereas threads are not. Explain why. *5pt*

Considering that preemptive scheduling is mainly deployed to establish fairness, it doesn't make much sense to apply it to threads as these belong to the same process, and thus user. Different processes, on the other hand, are often owned by different users, for whom it is important that they all get an "equal" share of the available processing power.

3a Show how the following concurrent programming can come to a deadlock. Make your assumptions explicit. *5pt*

```
(1) process producer(){
(2)   while(true){
(3)     produce_item();
(4)     if (count==N) sleep();
(5)     enter_item();
(6)     count = count + 1;
(7)     if (count==1) wakeup(consumer);
(8)   }
(9) }
```

```
(1) process consumer(){
(2)   while(true){
(3)     if (count==0) sleep();
(4)     remove_item();
(5)     count = count - 1;
(6)     if (count==N-1) wakeup(producer);
(7)     consume_item();
(8)   }
(9) }
```

To simplify matters, assume the producer and consumer are executed on a single processor. We assume that waking up a process that has not called sleep has no effect: the wakeup is "lost." Let `count==N` *and assume the producer has just completed the test on line (4). Before executing the call to* `sleep()`*, the consumer is scheduled, empties the buffer and calls* `wakeup()` *in line (6). That wakeup is lost, so when the consumer eventually calls* `sleep()` *in line (3), and the producer is eventually rescheduled to complete its call to* `sleep()`*, both processes will be sleeping.*

3b A binary semaphore has two *atomic* operations `lock()` and `unlock()` as specified below. Give a deadlock-free solution for the producer-consumer problem using only binary semaphores. *5pt*

```
atomic lock(s){
  if (s.val == 1 )
    s.val = s.val - 1;
  else{
    enter_queue(s.q)
    sleep();
  }
}
```

```
atomic unlock(s){
  if (length(s.q) > 0){
    p = remove_head(s.q);
    wakeup(p);
  }
  else
    s.val = s.val + 1;
}
```

```
process producer(){
  while(true){
    produce_item();
    lock(buffer);
    if (count==N){
      unlock(buffer);
      lock(prod);
      lock(buffer);
    }
    enter_item();
    count = count + 1;
    if (count==1) unlock(cons);
    unlock(buffer);
  }
}
```

```
process consumer(){
  while(true){
    lock(buffer);
    if (count==0) {
      unlock(buffer);
      lock(cons);
      lock(buffer);
    }
    remove_item();
    count = count - 1;
    if (count==N-1) unlock(prod);
    unlock(buffer);
  }
}
```

4a Give the general flow of control after a device has sent an electrical signal to the interrupt controller, until the point that interrupt handling is completed. *5pt*

(1) The interrupt controller sends a signal to the CPU. (2) The CPU acknowledges the interrupt. (3) The interrupt controller passes an ID of the interrupting device. (4) The hardware saves the current program counter and possibly other register values. (5) The hardware loads the address of the interrupt handler from the interrupt vector into the program counter. (6) Other registers are saved and the interrupt handler is executed. (7) Call the scheduler to restart a process.

4b Can an interrupt handler be implemented as a process? Explain your answer. *5pt*

Yes, although it may not be very efficient. The basic idea in this case is to have only very small interrupt handlers that immediately schedule the associated handler process, then call the scheduler, who subsequently selects the handler. This scheme does require that handlers get top priority. When a handler is done, it simply calls the scheduler again and goes to sleep, unless there was more work to do.

Part II

4a How many entries does a single-level page table have in a n-bit address space with a page size of 2^p bytes? *5pt*

This should be an easy one. There are 2^n memory locations grouped into units of 2^p bytes. Therefore, we need a total of $2^n/2^p = 2^{n-p}$ entries.

4b How would you compute the physical size (i.e., in bytes) of a single-level page table? *5pt*

In this case, we need to know the maximum number of page frames. For simplicity, assume that the page frame size and logical page size are the same, i.e. 2^p bytes. If the physical memory is limited to 2^m bytes, there would a total of 2^{m-p} page frames, each requiring a $(m-p)$-bit identifier. Furthermore, some extra bits are needed for administrative purposes, like a present/absent bit, protection bits, etc. Say this adds up to another k bits. This leads to a total of $2^{n-p} \times \lceil (m-p+l)/8 \rceil$ bytes.

4c Consider a process using a lot of virtual memory. Would you prefer a single-level or two-level page table? Explain your answer. *5pt*

The issue here is that multi-level page tables are mainly useful for processes that don't need a lot of memory, for in that case you would just be wasting space by keeping the complete table in memory. For a process using a lot of memory, you will need to access lots of page references in memory anyway, for which reason it may not make much difference if you use a single-level or multi-level table. The drawback of the multi-level page tables is that lookups can be more expensive.

4d Explain what an inverted page table is, and how a memory location is looked up. *5pt*

An inverted page table simply keeps a mapping of a page frame to a (process, logical page)-pair. As a consequence, when looking up a (virtual) memory location, you need to go through entire table to find a matching entry, if any. (This lookup can be improved by caching previously looked up entries in a TLB.)

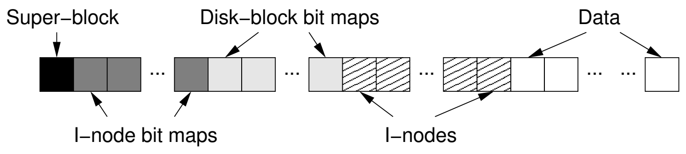
5a MINIX uses a separate process called the *system task*. What does this process do and why is it needed? *5pt*

The system task acts as an intermediate between a number of user-level system processes such as the memory manager and the file system manager, and the operating system kernel. The system task is allowed to modify kernel-level data structures, such as the memory map. However, thses structures are actually maintained by user-level processes. The latter cannot directly modify kernel-level data, and will therefore request the system task to do so on their behalf.

5b Give at least one compelling argument why the MINIX file manager needs to be aware of processes. *5pt*

One good argument is that files can be shared between processes. This means that the file manager will have to set up an administration concerning processes pointing to inode descriptors. As an example, when a process creates a child, the latter will inherit all open files from its parent.

6a A UNIX-like file system generally has a disk layout as shown below. Explain what is typically found in the super-block. *5pt*

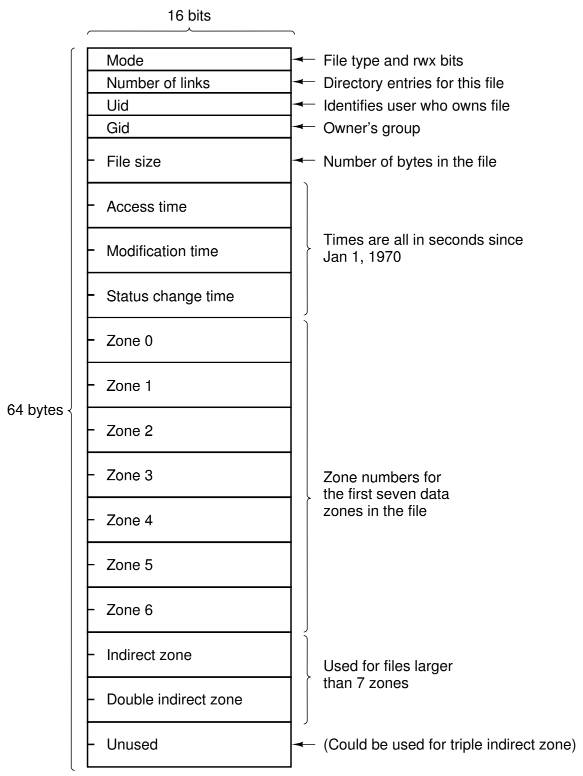


Your answer should at least include the number of i-node bit map blocks and the number of disk-block bit maps, along with information on where the first data block can be found. Other useful information includes the size of disk blocks, maximum file size, and size of the file system.

6b A MINIX i-node is 64 bytes. With a disk-block size of 1 Kbyte, how many disk blocks need to be reserved for storing the i-node bit maps and the i-nodes, given that you want to support a maximum of *N* files? *5pt*

You need a total of $\lceil N/16 \rceil$ *blocks to store i-nodes. Furthermore, a single block can store a 8192-sized bit map, meaning that another* $\lceil N/8192 \rceil$ *blocks are needed for managing the i-nodes.*

6c Assuming that zones and disk blocks have a size of 1 Kbyte, compute the largest size of a MINIX file, given that disk blocks are addressed by means of 32-bit pointers. *5pt*



16 bits
64 bytes
Mode
Number of links
Uid
Gid
File size
Access time
Modification time
Status change time
Zone 0
Zone 1
Zone 2
Zone 3
Zone 4
Zone 5
Zone 6
Indirect zone
Double indirect zone
Unused
File type and rwx bits
Directory entries for this file
Identifies user who owns file
Owner's group
Number of bytes in the file
Times are all in seconds since Jan 1, 1970
Zone numbers for the first seven data zones in the file
Used for files larger than 7 zones
(Could be used for triple indirect zone)

Maximum file size = $7 \times 1K + 256 \times 1K + 256 \times 256 \times 1K = 65.799 Kbyte$

Final grade: *(1) Add, per part, the total points. (2) Let T denote the total points for the midterm exam $(0 \leq T \leq 45)$; $D1$ the total points for part I; $D2$ the total points for part II. The final number of points E is equal to* $\max\{T, D1\} + D2 + 10$.