Dept. Computer Science **Operating Systems**
Vrije Universiteit **28.03.2007**

This exam consists of two pages

1a MINIX3 has adopted the **client-server model** to structure itself. Explain what this model entails, and notably what the role of the kernel is. *5pt*

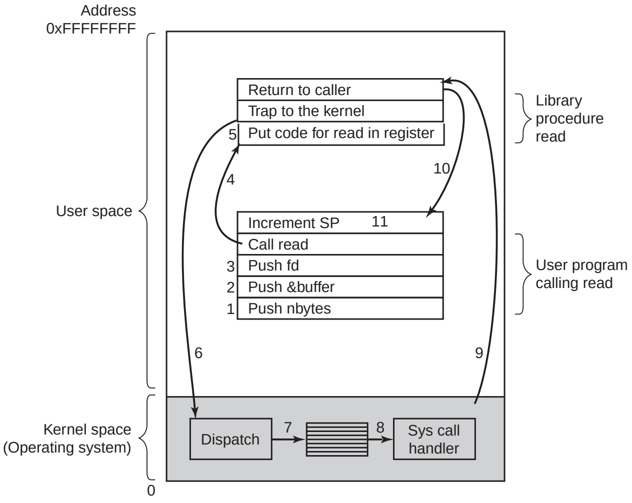
In the CS model, OS services are largely incorporated into separate processes, such as a file service, a process server, etc., also low-level I/O drivers are run as ordinary user-space processes. The role of the kernel is mostly restricted to handling requests from applications, which are dubbed clients, to OS services, by sending request messages and handling the reply messages. Also, in order to facilitate direct communication with HW controllers, the kernel has a separate task that communicates with authorized process to handle such I/O.

1b Drivers in MINIX3 run as ordinary user-space processes. In what sense does this impose restrictions for communicating with hardware controllers, and how is that solved? *5pt*

The problem is that user-space processes cannot directly send commands to controllers for doing low-level I/O. Drivers generally need this ability. The solution in MINIX3 is to have a separate system task that executes as a kernel-level task, and which accepts requests for low-level I/O from device drivers. The system task will communicate with the controllers, sending the results back to the drivers.

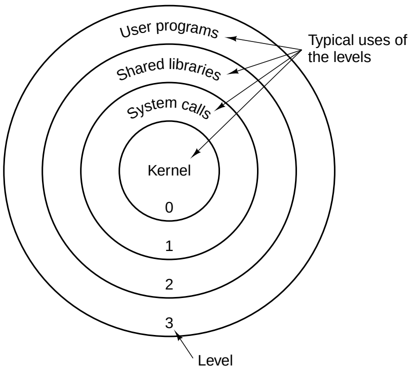
1c Sketch the flow of control when a user-space process calls the library routine read(fd,buffer,bytes) in a monolithic operating system. Hint: use a diagram. *5pt*

In essence, you should provide the following figure:



1d In many cases, the hardware offers support for multiple rings of protection for programs. How can we make use of this support when organizing an operating system? *5pt*

The crux here is that processors such as the Pentium offer different protection modes that allow calls from layer $k+1$ to layer k, but not to other lower level layers. For MINIX3, we could place user programs in layer 3, shared libraries in layer 2, client-side system call implementations in layer 1, and the kernel in layer 0, as shown in the following figure:



2a The semantics of the *atomic* **swap** machine instruction is defined as follows. Show how this instruction can be used to protect a critical section. *5pt*

swap(**inout** boolean a, **inout** boolean b){ temp = a; a = b; b = temp;}

A more or less obvious solution is the following. Crucial to your answer is that you see that swap effectively does almost the same thing as the TSL instruction.

```
bool lock; /* shared variable, initially set to FALSE */
void process some_process(){
   bool key;
   while(TRUE){
      key = TRUE;
      while( key == TRUE ) swap( &lock, &key );
      criticial_section();
      lock = FALSE;
      non_criticial_section();
   }
}
```

2b Consider the program on page 5, which is to be executed as a separate MINIX3 user-space **lock manager** process. The core of the program is formed by the functions do_down() and do_up() which are standard operations on counting semaphores. Given lock_manager(), give a pseudo-code implementation of the function do_down(sema). *5pt*

An important aspect is that in this case you cannot simply let the caller block inside do_down(): you would be blocking the lock manager. Instead, simple registration of the process needs to take place if the value of the associated semaphore is less or equal than zero. This leads to the following:

```
int do_down(sema){
   if( value[sema] <= 0 ){
      append( queue[sema], who );
      return( SUSPEND );
   }else{
      value[sema] = value[sema] - 1;
      return( OK );
   }
}
```

2c Also give a sketch of the implementation of do_up(). *5pt*

Crucial in this case is that you need to prepare a reply message when “unblocking” a previously blocked process:

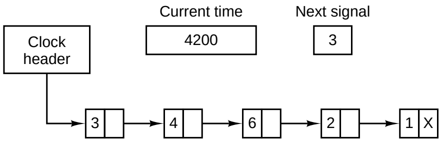
```
int do_up(sema){
   if( length(queue[sema] > 0) ){
      remove_head( queue[sema], &proc );
      setreply( proc, OK );
   }else{
      value[sema] = value[sema] + 1;
   }
   return(OK);
}
```

2d Returning SUSPEND by do_down() has the result of suspending a process. Explain which process that is, and how this blocking is actually effectuated. *5pt*

The process that initially called the system calls associated with do_down() and do_up(). In MINIX3, these calls have been transformed into messages that are sent to the lock manager. Because message-passing in MINIX3 is synchronous, the caller will remain blocked by the kernel until a reply is sent back. The lock manager will do so only when the down() operation succeeded, or later when the up() operation lead to unblocking the previously blocked caller.

3a Explain how MINIX3 (and many other operating systems) simulate multiple timers using a single clock. Draw a figure to explain your answer. *5pt*

Your answer should more or less explain the following figure:



3b Explain the difference between **character devices** and **block devices**, and why making this distinction can be helpful for improving I/O. *Hint: think of writing a stream of bytes to disk.* *5pt*

The difference lies in the unit of transferring data: character devices do this in terms of bytes, whereas block devices operate on whole blocks of data. As a consequence, it is much easier to optimize performance with block devices, as you can easily build buffer caches as in most UNIX systems. Although this is also possible for character devices, you still need to process input on a per-character basis (meaning that, in principle, an interrupt will have to be generated at each incoming character. Another characteristic difference is that block devices can support an lseek operation, which is less obvious for character devices.

4a Explain the principle working of the fork() system call. *5pt*

Your answer should include mentioning that fork() places a copy of the caller's memory image, effectively copying the calling process to a new one. The caller is returned the child's PID, while the child is returned the value 0.

4b **Copy-on-write** is a technique by which a block of memory is filled with data from a specific source only when first written to. How can this technique help in optimizing the implementation of fork()? *Be precise!* *5pt*

The trick is that copy-on-write only allocates memory to the child process, but it will not copy the parent's memory segments until necessary. To this end, note that a complete map of the child memory will have to be installed; the only thing that is not being done is the expensive copying. The technique is useful because in many cases the child will want to execute a different program than its parent, effectively wasting a complete copy anyway.

5a Explain what the mount() system call does by means of an example. *Explain your example!* *5pt*

Mount() establishes that a complete file system as stored on, e.g., disk, becomes accessible through another file system. Your example can be simple, but make sure you mention the notion of mount point and that the root of the file system is taken as mounted point.

5b Mount() changes fields in inodes and in-memory copies of superblocks. Explain these changes. *5pt*

First, the inode that is mounted on will have a boolean set that it is now a mount point. This bit is needed when parsing pathnames. Second, the superblock of the mounted file system will (1) have a pointer to the mount point, and (2) a pointer to the inode of its root node.

5c Consider the following operations that are carried out on a formatted, but otherwise empty USB stick. Explain what the result will be when listing the directory contents (by means of the last operation ls). *5pt*

```
mount /dev/sdb1 /usbstick   Mount the USB stick
cd /usbstick                Enter the directory
mkdir test                  Create a subdirectory ''test.''
touch test/x                Create a file ''x'' in ''test.''
mount /dev/sdb1 test        Mount the USB stick again
ls test                     List the directory contents
```

It's not that difficult: you will see that the directory "test" is listed – the device is simply mounted twice.

5d Explain precisely what happened with the superblock table and inode table after the two mount operations from the previous example have been carried out. *5pt*

The first mount operation creates an entry in the superblock table to which the superblock of the USB file system will be loaded. At the same time, an inode entry in the inode table is created for the inode of the root node of that file system. Pointers and booleans will be set as explained for question (b). This procedure is repeated as is, *except that for the second mount we have a different inode to consider.*

6a What is a protection domain? *5pt*

"A [protection] domain is a set of (object, rights) pairs. Each pair specifies an object and some subset of the operations that can be performed on it. A right in this context means permission to perform one of the operations."

6b Give a practical example of how to switch from one protection domain to another, and explain how such a switch could be implemented by an operating system. *5pt*

When excuting the passwd *command, you may need to change the otherwise protected password file. This can be done safely by switching to the domain of the super user. Such a switch can be implemented by keeping track of effecitive UIDs, and letting the file server control the transition from real UID to effective UID. When accessing files, only the effective UID is checked when checking authorisation.*

```
PUBLIC int lock_manager(){
  int result, s, proc_nr;
  struct mproc *rmp;
  while (TRUE) {
    receive(ANY, &msg_in);
    who = msg_in.m_source;          /* who sent the message          */
    sema = msg_in.m5_l1;            /* which semaphore is this?      */
    call_request = msg_in.m5_i1;    /* which operation is requested? */
    mp = &mproc[who];
    switch(call_request){
      DOWN: result = do_down(sema); break;
      UP:   result = do_up(sema); break;
    }

    /* Send the results back to the user to indicate completion. */
    if (result != SUSPEND) setreply(who, result); /* Prepare  reply message */
    /* Send out all pending reply messages, including the answer to
     * the call just made above.
     */
    for (proc_nr = 0, rmp = mproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
      if ((rmp->mp_flags & REPLY) == REPLY ){
        send(proc_nr, &rmp->mp_reply);
        rmp->mp_flags &= ˜REPLY;
      }
    }
  }
  return(OK);
}
```

Grading: *The final grade is calculated by adding the scores per question (maximum: 90 points), and adding 10 bonus points. The maximum total is therefore 100 points.*