CS 4/53201 # Homework #1 **Operating Systems**

Due in class on Wednesday 23 September 1998

1. **(Exercise 1.9 from OSC) Under what circumstances would a user be better off using a time-sharing system, rather than a personal computer or single-user workstation?**

   When there are few other users, the task is large, and the hardware is fast, time-sharing makes sense since the mainframe has more computing power than a personal computer and that full computing power can be brought to bear on the user's problem. Using a time-sharing system would also be appropriate if that system has expensive resources (e.g., a color laser printer) that are not available on the personal computer. Finally, if the user needs to "run" multiple tasks simultaneously, the time-sharing system would also be most appropriate, as most personal computer operating systems (with the exception of Linux) don't support multiprocessing very well. (A personal computer, in contrast, would be better when the job is small enough to be reasonably executed on it, and when the machine's performance is sufficient to execute the program to the user's satisfaction.)

   Note that this question asks why "a" user (i.e., a single user) would be better off… It does not ask about multiple users.

2. **(Exercise 2.6 from OSC (2.7 in 4th Edition)) Some computer systems do not provide a privileged mode of operation in hardware. Consider whether it is possible to construct a secure operating system for these computers. Give arguments both that it is and that it is not possible.**

   It is possible, but the solutions aren't very appealing. An operating system that can not rely on a privileged mode of operation would need to remain in control (or kernel mode) at all times. This could be done through software interpretation of all user programs (like Lisp, for example), with the software providing what the hardware does not. Another possibility would be to require that all programs be written in high-level languages, and modify the compilers to generate protection checks that are missing in hardware.

3. **(Exercise 3.5 from OSC) What is the purpose of the command interpreter? Why is it usually separate from the kernel?**

   It reads commands typed by the user (or from a file) and executes those commands, usually by turning them into system calls. It is usually not part of the kernel, since multiple command interpreters (shells, in Unix terminology) may be supported by an operating system, and they don't really need to run in kernel mode.

4. **(Exercise 3.9 from OSC) What is the main advantage of the layered approach to system design?**

   The system is easier to debug and modify, because changes affect only limited portions of the code, and the programmer doesn't have to know the details of the other layers.. Information is also kept only where it is needed and is accessible only in certain ways, so bugs affecting that data are limited to a specific module or layer.

5. **(Exercise 4.5 from OSC) What resources are used when a thread is created? How do they differ from those used when a process is created?**

   A thread control block must be created, including space for storing registers, the PC, and the SP. Stack space must also be allocated for the thread within the process.

6. **(Exercise 4.9(a) from OSC) Consider the interprocess-communication scheme where mailboxes are used. Suppose a process P wants to wait for two messages, one from mailbox A and one from mailbox B. What sequence of send and receive should it execute?**

   Sends aren't involved in waiting for messages. One possibility would be to execute Receive(A, message) followed by Receive(B, message), but this would mean that the messages must be received in the order A,B (since the first Receive will block until it receives a message). It would be nice to somehow say Receive(A or B, message), but that sort of operation isn't usually supported. So, one answer to this question is: there isn't any good way to do this.

   An alternative answer would be for process P to fork two threads, one of which executes Receive(A, message), and the other of which executes Receive(B, message). This way each will block if no message is available, but whichever receives its message first will wake up and process the message.