Chapter 3: Operating Systems Organization

Exercises

1. There is no security risk in the user program knowing the index of a system function in the index table – it equivalent to knowing the file index in the open file table. However, if the OS rearranges the table, then there is no mechanism to change all the programs that were previously compiled to use a particular table index. By using stub functions, it is easy for the OS to rebuild all the stub functions with the right trap table index whenever the trap table is rearranged.
2. A normal procedure call requires that the contents of general registers and a return address be save. However, it does not require the system to multiplex processes, so all of the effort related to process context switching can be ignored for procedure call.
3. A network of workstations can only communicate with one another by sending and receiving information via the network. If two or more processes are executing on different workstations in the network and they assume the existence of a logical shared memory environment, then the system must provide facilities by which at least some processes exchange messages over the network on memory references. There are several different approaches. For example, assume that processes p_i and p_j are executing on machines M_i and M_j, respectively. Further, assume the variable X has been declared to be in the shared memory, *and is allocated on M_i in p_i 's address space*. Whenever p_i reads X, no special action needs to be taken. Whenever p_j reads X, the system must send a message to system code on M_j to read the value of X and to return it to p_j. (The value could be copied for efficiency, but this introduces new problems not described in this solution.) Whenever p_i writes X, no special action needs to be taken. Whenever p_j writes X, the system must send a message to system code on M_i to cause X to be written with the new value. The logical shared memory is an asset to the application programmer, since s/he need not learn about network messages, using only ordinary memory reads and writes (perhaps using a system call). The logically shared memory is easy to use.
4. **Writing to the stack segment register**. When the assembly language program writes to the stack segment register, the process's entire stack contents are potentially lost. This follows since the part of memory holding the stack is no longer referenced by the stack segment register. In particular, when the assembly language program returns to the C program (presuming it somehow saved a return address prior to clobbering the stack), the C program's automatic variables and call stack will all be lost.
   Writing to the code segment register. Writing to the code segment register immediately causes an unusual form of a branch statement. When the instruction finishes executing, the control unit will add the contents of the PC to the new value of the code segment register, which will now point to a different 64KB block in memory than it did before the instruction was executed. The next instruction will be fetched from the PC displaced address in the new code block.
5. This problem will require that you provide extensive outside help to your students, or that you use the materials on the book's Addison Wesley web site (there is a full lab exercise on the web site, taken from Kernel Projects for Linux. You can obtain the solution to the lab from Addison Wesley).
6. This problem will require that you provide extensive outside help to your students, or that you use the materials on the book's Addison Wesley web site (there is a full lab exercise on the web site, taken from Operating System Projects Using Windows NT. You can obtain the solution to the lab from Addison Wesley).