

CSE 120, Answers to HW1

Chapter 1

Q 1 (See Tanenbaum 1.1)

One function of the OS is to abstract out the hardware, i.e. provide an easier to program interface to the user by hiding all the superfluous details of the inner-workings of the machine.

The OS also functions as a resource manager, i.e. allows for an orderly allocation of resources (e.g. processor, I/O devices) among the various processors competing for them.

Q 2 (See Tanenbaum 1.2.3)

Basically multiprogramming is the ability of a computer to hold multiple jobs (e.g. processes) at once by partitioning memory appropriately, with the purpose of reducing processor idle time: whenever the current job releases the processor (e.g. waiting for I/O) another job can be scheduled.

Remember that earlier computers held one job at the time and ran it till completion before another could be loaded.

Q 7 (See Tanenbaum 1.2.3)

- The screen can display $25 * 80$ characters. If each character requires 8 bits then the total memory needed is $25*80*8 = 16000$ bits. Here 8 bits is sufficient because no color encoding is required on a *monochrome* text screen.
- Each pixel requires 24 bits to be encoded, so the total memory needed is $1024*768*24/8 = 2359296$ bytes or 2304 KB.
- $2304*5 = \$11520$ would be the cost of the required RAM back then. You can now get 256 Mb for \$30 which is about \$0.25 for 2 Mb.

Q 8

(a) and (d)

Q 14 (See Tanenbaum 1.4.3 & 1.6)

While both traps and interrupts are used to switch the system to kernel mode, a trap is an instruction used by a process to make a system call, whereas an interrupt is generated by hardware (an I/O device or timer). Interrupts can happen at any time.

Q 18 (See Tanenbaum 1.6.1 & 1.6.3)

- `fork` would fail for lack of memory to allocate a new process space, in fact it would fail for lack of any resources required by the OS to create a new process.

- `exec` would fail if the file to be executed did not exist, or if the process does not have the right permissions to execute it.
- `unlink` would fail if the process does not have the right permissions to unlink the file, or if the file does not exist.

Q 20 (See Tanenbaum 1.6.2)

4, 1, 5, 9.

Chapter 2

Q 4 (See Tanenbaum 2.1.6)

In most cases the interrupted process is not likely to be given back the CPU when the OS finishes handling the interrupt of system call trap. The process information (logical PC, stack pointer, ...) would therefore need to be saved to the process table for a new process to be ran.

The work of saving this information would need to be done by loading appropriate OS instructions, and therefore the physical PC and possibly stack pointer need to be reset. Since these registers originally contain information about the interrupted process, these values need to be stored temporarily before the OS instructions are loaded. In general it is appropriate to save the temporary information on the corresponding process's stack, which then calls for using a separate stack area for the OS instructions (any other solution would be cumbersome).

Q 8

Kernel level threads must be used. This way when a thread is busy waiting for file I/O, the whole process won't be blocked, i.e. other threads from the Web server can keep running.

Q 9 (See Tanenbaum 2.2.3)

On a system with only kernel level threads there may never be a need to call `thread_yield`. On the other hand, when user level threads are being used, an implementation can force a thread to explicitly yield the processor whenever it has to wait for a local process resource. The user level scheduler would then ensure that this thread is scheduled again whenever it become ready.

Q 18 (See Tanenbaum 2.3.1)

Please read this section to gain a good understanding of the concept. In summary, a race condition is a situation where two or more threads are reading and writing the same data and the final result depends on who runs precisely when.

Q 22 (See Tanenbaum 2.3.3)

The importance of the TSL instruction is that the variable `LOCK` can be read and updated in one indivisible operation. This way if two processes called `enter-region` at the same time, only one would return immediately (the first to set lock to 1) and the other would then wait for the lock. If the operation was divisible (first `READ LOCK`, then `UPDATE` to 1), both processes could read `LOCK` to be 0 (one reads it first but before it can update `LOCK` it is pre-empted and the other one gets to read the same value of 0) in which case they both would enter the critical region simultaneously.

An implementation using the alternative `SWAP` operation is given below:

enter_region:

```
MOVE REGISTER, 1      //set the register value to 1
SWAP REGISTER, LOCK    //swap the values of register and lock in an indivisible operation
CMP REGISTER, #0
JNE enter_region      //jump back to the beginning if lock was not zero (loop)
RET                   //return to caller.
```

leave_region remains the same as in Fig 2-22.

Q 23 (See Tanenbaum 2.3.5)

The key idea is to implement up and down to be atomic operations:

Queue Waiting; //store id's of process waiting on the semaphore

```
down (int process_id, int& semaphore) {
    disable_interrupts();

    if (semaphore != 0)
        semaphore = semaphore - 1; //just decrease and we're done
    else {
        Waiting.add(process_id);
        yield(process_id);          //force the calling process to go to sleep
    }

    enable_interrupts;
}

up (int process_id, int& semaphore) {
    disable_interrupts();

    if (Waiting.isEmpty())
        wake_up(Waiting.dequeue()); //wake up the first process waiting, done.
    else
        semaphore = semaphore + 1;   //just increment

    enable_interrupts();
}
```

Note:

We assume here that up and down are system calls. This way the instructions would be executed in kernel space so the call to yield won't prevent the procedure to complete and re-enable interrupts.

Q 24 (See Tanenbaum 2.3.5)

Just replace disable_interrupts() and enable_interrupts() in the above code with, respectively, mutex.lock() and mutex.unlock() (assuming some global mutex variable).