CS414 SP 2007 Assignment 1

Due Feb. 07 at 11:59pm Submit your assignment using CMS

1. Which of the following should NOT be allowed in user mode? Briefly explain.

a) Disable all interrupts.
b) Read the time-of-day clock
c) Set the time-of-day clock
d) Perform a trap
e) TSL (test-and-set instruction used for synchronization)

Answer: (a), (c) should not be allowed.

Which of the following components of a program state are shared across threads in a multi-threaded process ? Briefly explain.

a) Register Values
b) Heap memory
c) Global variables
d) Stack memory

Answer: (b) and (c) are shared

2. After an interrupt occurs, hardware needs to save its current state (content of registers etc.) before starting the interrupt service routine. One issue is where to save this information. Here are two options:

a) Put them in some special purpose internal registers which are exclusively used by interrupt service routine.
b) Put them on the stack of the interrupted process.

Briefly discuss the problems with above two options.

Answer:

(a) The problem is that second interrupt might occur while OS is in the middle of handling the first one. OS must prevent the second interrupt from overwriting the internal registers. This strategy leads to long dead times when interrupts are disabled and possibly lost interrupts and lost data.

(b) The stack of the interrupted process might be a user stack. The stack pointer may not be legal which would cause a fatal error when the hardware tried to write some word at it. Also, OS has to be careful to clean up the stack later, to avoid leaking information.

3. You are asked to implement a online computer chess game server. The idea is that people can use the client-side GUI to connect to, and play against the game server. The server should be designed to support a large number of concurrent players. Suppose you can only use blocking I/O to read from network, would you choose user-level threads or kernel-level threads for the implementation? Briefly explain.

Answer:

Kernel-level threads. Otherwise, doing socket I/O to handle one user will block the entire process.

4. Consider a system running ten 1/O-bounds tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and all processes are long-running tasks. What is the CPU utilization for a round-robin scheduler when:

a) The time quantum is 1 millisecond
b) The time quantum is 10 millisecond

Answer:

(a) The time quantum is 1 millisecond: Irrespective of which process is scheduled, the scheduler incurs a 0.1 millisecond context-switching cost for every context-switch. This results in a CPU utilization of 1/1.1 * 100 = 91%.

(b) The time quantum is 10 milliseconds: The I/O-bound tasks incur a context switch after using up only 1 millisecond of the time quantum. The time required to cycle through all the processes is therefore 10*1.1 + 10.1 (as each I/O-bound task executes for 1 millisecond and then incur the context switch task, whereas the CPU-bound task executes for 10 milliseconds before incurring a context switch). The CPU utilization is therefore 20/21.1 * 100 = 94%.

5. Consider a system implementing multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process? What if the system implements round-robin?

Answer:

The program could maximize the CPU time allocated to it by not fully utilizing its time quantums. It could use a large fraction of its assigned quantum, but relinquish the CPU before the end of the quantum, thereby increasing the priority associated with the process.

For round-robin, it's okey to answer that there is no strategy exist. Or, another answer would be to break a process up into many smaller processes and dispatch each to be started and scheduled by the operating system.

6. You are asked to consult on the design of a simple operating system called Lunix. Lunix is designed to support multiprogramming on a single-processor machine. However, the designer have observed that under various conditions the machine hangs because some of the concurrently executing processes either stop making forward progress, or make progress so slowly they might as well have stopped. Your job is to debug their kernel.

The Lunix kernel job queue consists of an array of process control blocks with the following structure:

```
struct PCB {
        int process_id;
        enum { running, ready, waiting } status;
        int priority;
        struct PCB *next_ready;
        /* "next ready" points to the next process on the ready */
        /* queue, or is NULL if this is the last process */
        void *PC;
        /* program counter at which to restart execution */
        ...more stuff...
}
```

The scheduler also has a variable `ready_queue` of type `struct PCB*` that is a pointer to the first process on the ready queue. The ready queue is a linked list formed by the next ready pointers, in which the status of every process is ready. The linked list is kept sorted in order of descending process priority.

a) To impress the Lunix designers with your fitness for the consulting job, name two additional kinds of information you would expect to find in the section of the PCB labeled "more stuff."

Answer:

Memory-management information: page tables, etc.
I/O status information: list of open files and so on.

b) The Lunix kernel implements preemptive priority scheduling, yet the system designers are finding that often processes are not preempted. They have determined that the problem is in the scheduler itself. The scheduler selects and dequeues a new process to run using the following code:

```
struct PCB *choose_process() {
        struct PCB *ret = ready_queue;
        ready_queue = ret->next_ready;
        return ret;
}
```

Because the scheduler is preemptive, sometimes running processes must be placed back on the ready queue when preempted; this is implemented by the following function which ensures the ready queue is kept sorted on priority:

```
struct PCB *reenqueue_running(PCB *current_proc) {
        current_proc->status = ready;
        struct PCB *p = ready_queue;
        struct PCB **prev = &ready_queue;
        while (p != NULL && p->priority > current_proc->priority) {
                prev = &p->next_ready;
                p = p->next_ready;
        }
        *prev = current_proc;
        current_proc->next_ready = p; // insert into linked list
}
```

Why isn't preemption working? Suggest a one-line fix.

Answer:

Imagine if all processes have the same priority, then the newly preempted process will be inserted in the front of the queue, hence will immediately start running again. This will make it look like that process was never preempted. The fix is: in the while statement of `reenqueue_running function, change the ">" to ">="`

c) Even after you fix preemptive scheduling, it is discovered that some processes are not scheduled for execution for long periods of time. When can this happen? What "term" is commonly used to describe this phenomenon?

Answer: Starvation

d) Lunix provides shared-memory segments to which multiple processes can read and write. To allow control of concurrent access to these segments, it provides mutexes with the following operations:

```
void lock(Mutex m); /* wait until mutex is available and then acquire it */
void unlock(Mutex m); /* release a mutex that is held */
```

The Lunix printer server is implemented using shared memory; processes submit a print request by locking a mutex m, putting the request in a print-queue data structure, and releasing the mutex. This involves no system calls and all memory accessed is physically resident – there are no page faults. Note that a mutex is like a lock and is only released through an explicit call to unlock.

When high-priority and low-priority processes attempt to share access to the printer, in the presence of other medium-priority processes, it is found that the high-priority processes often block indefinitely on the initial call to lock(m), while medium-priority processes continue to be scheduled and run. What is happening, and how might the scheduler be modified to address the problem?

Answer: The high-priority process is waiting for the lock which is held by the low-priority process, and since there are presence of the medium-priority processes, the low-priority never get scheduled, hence, the high-priority ones will block. (This problem is known as priority inversion). One possible solution is the following: all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priority revert to their original values.