Recitation 1-
Process management and Synchronization Solutions

Ειρήνη Δέγκλερη (degleri@csd.uoc.gr)
Παναγιώτης-Ηλίας Παπαδόπουλος (ppapadop@csd.uoc.gr)
PROBLEM 3

On all current computers, at least part of the interrupt handlers are written in assembly language. Why?
PROBLEM 3

On all current computers, at least part of the interrupt handlers are written in assembly language. Why?

All interrupts start by saving the registers, often in the process table entry for the current process. Then the information pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the process handler. Actions such as saving the registers and setting the stack pointer cannot even be expressed in high-level languages such as C, so they are performed by a small assembly-language routine, usually the same one for all interrupts since the work of saving the registers is identical, no matter what the cause of the interrupt is.
PROBLEM 4

When an interrupt or a system call transfers control to the operating system, a kernel stack area separate from the stack of the interrupted process is generally used. Why?
PROBLEM 4

When an interrupt or a system call transfers control to the operating system, a kernel stack area separate from the stack of the interrupted process is generally used. Why?

There are several reasons for using a separate stack for the kernel. Two of them are as follows. First, you do not want the operating system to crash because a poorly written user program does not allow for enough stack space. Second, if the kernel leaves stack data in a user program’s memory space upon return from a system call, a malicious user might be able to use this data to find out information about other processes.
A process is just an instance of an executing program, including the current values of the program counter, registers, and variables.

- Each process has an address space and a single thread of control.
- Conceptually, each process has its own virtual CPU.

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
Thread

Threads are like mini processes. The main reason for having threads is that in many applications, multiple activities are going on at once. Some of these may block from time to time.

Why threads?

- ability for the parallel entities to share an address space and all of its data among themselves
- lighter weight than processes, they are easier (i.e., faster) to create and destroy than processes
- allow many activities to overlap, thus speeding up the application.

Like a process, a thread can be in any one of several states: running, blocked, ready, or terminated.
PROBLEM 10

The register set is listed as a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.
PROBLEM 10

The register set is listed as a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.

When a thread is stopped, it has values in the Registers, which must be saved.

Multiprogramming threads is no different than multiprogramming processes, so each thread needs its own register save area.
PROBLEM 14

What is the biggest advantage of implementing threads in user space?

What is the biggest disadvantage?
PROBLEM 14

What is the biggest advantage of implementing threads in user space?

The biggest advantage is the efficiency. No traps to the kernel are needed to switch threads.

What is the biggest disadvantage?

The biggest disadvantage is that if one thread blocks, the entire process blocks.
Implementing Threads in User Space

This method puts the threads package entirely in user space.

The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads, so the system is more efficient. With this approach, threads are implemented by a library.

Despite their better performance, user-level threads packages have some major problems. First among these is the problem of how blocking system calls are implemented. Suppose that a thread reads from the keyboard before any keys have been hit. Letting the thread actually make the system call is unacceptable, since this will stop all the threads. Somewhat analogous to the problem of blocking system calls is the problem of page faults.
PROBLEM 21

In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used? Explain.
PROBLEM 21

In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used? Explain.

Each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses, and so on. This is equally true for user-level threads as for kernel-level threads.
PROBLEM 26

Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions.
PROBLEM 26

Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions.

Associated with each counting semaphore are two binary semaphores, M, used for mutual exclusion, and B, used for blocking. Also associated with each counting semaphore is a counter that holds the number of ups minus the number of downs, and a list of processes blocked on that semaphore.

- To implement down, a process first gains exclusive access to the semaphores, counter, and list by doing a down on M. It then decrements the counter. If it is zero or more, it just does an up on M and exits. If M is negative, the process is put on the list of blocked processes. Then an up is done on M and a down is done on B to block the process.

- To implement up, first M is downed to get mutual exclusion, and then the counter is incremented. If it is more than zero, no one was blocked, so all that needs to be done is to up M. If, however, the counter is now negative or zero, some process must be removed from the list. Finally, an up is done on B and M in that order.
Interprocess communication

Processes that are working together may share some common storage that each one can read and write. Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.

To preventing trouble in situations involving shared memory, we need is mutual exclusion, (or some other synchronization primitive) that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.
PROBLEM 29

Synchronization within monitors uses condition variables and two special operations, wait and signal. A more general form of synchronization would be to have a single primitive, waituntil, that had an arbitrary Boolean predicate as parameter. Thus, one could say, for example,

\[ \text{waituntil } x < 0 \text{ or } y + z < n \]

The signal primitive would no longer be needed. This scheme is clearly more general than that of Hoare or Brinch Hansen, but it is not used. Why not?

*Hint: Think about the implementation.*
PROBLEM 29

Synchronization within monitors uses condition variables and two special operations, wait and signal. A more general form of synchronization would be to have a single primitive, waituntil, that had an arbitrary Boolean predicate as parameter. Thus, one could say, for example,

\[
\text{waituntil } x < 0 \text{ or } y + z < n
\]

The signal primitive would no longer be needed. This scheme is clearly more general than that of Hoare or Brinch Hansen, but it is not used. Why not?

It is very expensive to implement. Each time any variable that appears in a predicate on which some process is waiting changes, the run-time system must re-evaluate the predicate to see if the process can be unblocked. With the Hoare and Brinch Hansen monitors, processes can only be awakened on a signal primitive.
Scheduling

A single processor may be shared among several processes, with some scheduling algorithm being accustomed to determine when to stop work on one process and service a different one.

Algorithms:
- First-Come, First-Served
- Shortest Job First
- Shortest Remaining Time Next
PROBLEM 38

A process running on CTSS (Compatible Time Sharing System) needs 30 quanta to complete. How many times must it be swapped in, including the very first time (before it has run at all)?
A process running on CTSS (Compatible Time Sharing System) needs 30 quanta to complete. How many times must it be swapped in, including the very first time (before it has run at all)?

The first time it gets 1 quantum.

On succeeding runs it gets 2, 4, 8, and 15, so it must be swapped in 5 times.

$$2^n \text{ quanta of time} \rightarrow 1+2+4+8+15 = 30$$