

Tutorial 1 - Introduction

1. The issue of resource allocation shows up in different forms in different types of operating systems. List the most important resources that must be managed by an operating system in the following settings;

(a) Supercomputer

Since this is most likely used for computation, processor time as well as memory should be carefully managed.

(b) Workstations connected to servers via a network

Network access and bandwidth.

(c) Smartphone

Energy, since it is a portable device, as well as access to hardware such as the camera, GPS, as well as connectivity (Bluetooth, mobile network, etc).

2. What is the kernel of an operating system?

The kernel of the operating system is part of the operating system that remains in memory, executing in the privileged part of the CPU.

3. Why is the separation into a user mode and a kernel mode considered good operating system design? Give an example in which the execution of a user process switches from user mode to kernel mode, and then back to user mode again.

Any bugs executing in user space should not cause the entire system to crash, since the kernel allows for recovery. If all programs were to run in kernel mode, a failure would bring down the entire system. An example of this switch would be writing to disk (or any system call in general).

4. Which of the following instructions should only be allowed in kernel mode, and why?

(a) Disable all interrupts

kernel only

If something in user were to disable interrupts, the kernel would have no way of regaining control,

(b) Read the time of day clock

user

All user processes should be able to access the time if needed.

(c) Change the memory map

kernel only

Managing memory should be restricted to the kernel.

(d) Set the time of day

kernel only

Processes running in user space should not be able to change the time, as it can cause issues for other processes.

5. A portable operating system is one that can be ported from one system architecture to another with little modification. Explain why it is infeasible to build an operating system that is portable without any modification. Describe two general parts that you can find in an operating system that has been designed to be highly portable.

Since the operating system must interact with the hardware, it's not feasible to build an OS that can interact with every hardware configuration. Device drivers (**platform specific**) allow for the operating system to interact with hardware, and this can be provided by the hardware manufacturer. Another part would be an API the OS provides to programs (**platform independent**), allowing them to interact with hardware via this abstraction.

Tutorial 2 - Processes + Threads

1. If a multithreaded process forks, a problem occurs if the child gets copies of all the parent's threads. Suppose that one of the original threads was waiting for keyboard input. Now two threads are waiting for keyboard input, one in each process. Does this problem ever occur in single-threaded processes?

No, this does not happen in single-threaded processes as the entire process would be blocked when waiting for input, and therefore cannot fork.

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

The biggest advantage is that it allows the thread scheduling to be managed by the programmer, and also avoids the overhead of context switching. On the other hand, the main disadvantage is that if any of the user space threads were to block (to wait for input), it would context switch to another thread.

3. If in a multithreaded web server the only way to read from a file is the normal blocking `read()` system call, do you think user-level threads or kernel-level threads are being used?

Kernel-level threads are being used, as the entire web server would block if it was done with user space threads.

4. Why would a thread ever voluntarily give up the CPU by calling `thread_yield()`? After all, since there is no periodic clock interrupts, it may never get the CPU back.

It is done to lower the priority of the thread in the scheduler, allowing another thread to run. This allows threads to cooperate.

5. The register set is 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 its own contents in a register which must be saved (and then restored transparently to the thread).

6. 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 threads are used? Explain.

Since each thread can call procedures, it must have a stack per thread in order to store local variables and calls. The same can be said for kernel threads.

7. In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server, running on a single CPU-machine. It takes 15 ms to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the block cache. If a disk operation is needed, as is the case one-third of the time, an additional 75 ms is required, during which time the thread sleeps. For this problem, assume that thread switching time is negligible. How many requests/sec can the server handle if it is single-threaded? If it is multithreaded?

Let the average time for an operation be $15 + \frac{1}{3} \cdot 75 = 40$ ms. Therefore, with a single thread, it can process 25 requests/s.

We want to calculate the probability that all threads are waiting for I/O. The average blocking time per thread is 25 ms, as seen above. This means that the probability that all threads are blocked is $1 - (\frac{25}{40})^n = 1 - (\frac{5}{8})^n$. As such, we can calculate the number of requests per second as;

$$\left(1 - \left(\frac{5}{8}\right)^n\right) \cdot \frac{1000}{15}$$

Note $\frac{1000}{15}$ requests/sec is at full efficiency.

8. Would an algorithm that performs several independent CPU-intensive calculations concurrently (e.g. matrix multiplication) be more efficient if it used threads, or if it did not use threads? Why is this a hard question to answer?

It would be more efficient as long as the overhead for threads is negligible compared to the performance increase, and the CPU is actually able to perform the computation in parallel (multiple cores), otherwise the overhead of threads will cause it to be less efficient. This is difficult to answer as it depends on the problem itself, how it is divided, as well as the system specifications.

9. IPC mechanisms

- (a) What happens when a signal is received by a process?

Other than **SIGKILL** and **SIGSTOP**, the receiving process is able to choose what it does with the signal. It can either ignore it, or manually handle it. Otherwise, it generally terminates the process.

- (b) When two processes communicate through a pipe, the kernel allocates a buffer (of size 65536 bytes in Linux) for the pipe. What happens when the process at the write-end of the pipe attempts to send additional bytes on a full pipe?

It cannot write, and will block until the process at the read-end reads from it, thus freeing up space.

- (c) What happens when the process at the write-end of the pipe attempts to send additional bytes and the process at the read-end has already closed the file descriptor associated with the read-end of the pipe?

The writing process will have an error returned to it.

- (d) The process at the write-end of the pipe wants to transmit a linked list data structure (with one integer field, an a "next" pointer) over a pipe. How can it do this?

Since they do not share address spaces, it must be serialised in some form by the sending process, which can then be converted back into a linked list by the receiving process.

- (e) When would it be better for two processes to communicate via shared memory instead of pipes? What about the other way around?

It would be better for two processes to communicate via shared memory as it is faster due to the lack of kernel intervention - it also allows for bi-directional communication. On the other hand, pipes are handled by the kernel, thus synchronisation does not have to be implemented by the programmer.

Tutorial 3 - Scheduling

1. State which of the following are true and which false.

- (a) Interactive systems generally use non-preemptive processor scheduling.

False. An interactive system uses preemptive scheduling - if there was a background process that was taking up too much time, the user interface would appear unresponsive. A preemptive scheduler allows this process to be preempted, giving control to the process running the user interface.

- (b) Turnaround times are more predictable in preemptive than in non-preemptive systems.

False. In a non-preemptive system, a process will run to completion (or until it blocks).

- (c) One weakness of priority scheduling is that the system will faithfully honour the priorities, but the priorities themselves may not be meaningful.

True. The actual priority of a job can depend on what other processes are running at the same time.

2. (a) Give an example showing why FCFS is not an appropriate scheduling scheme for interactive users.

Assume that there are two tasks on a system. The first task, is a large computation that happens in the background, and takes a notable amount of time (and is purely CPU bound, hence no blocking). The second task is a browser. Since the first task would be run first, the browser will be unresponsive to the end user, since the first task occupies the resources until it finishes.

- (b) Using the previous example, show why round-robin is a better scheme for interactive users.

Round-robin switches between the two tasks, thus ensuring both tasks get the processor periodically. As such, the browser will be able to respond (since the large task can be interrupted), and the computation can continue after the browser uses up its time quantum.

3. Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, X . In what order should they be run to minimise average turnaround time?

It should be run with the shortest job first, and the longest job last (X should be placed in the list 3, 5, 6, 9 accordingly).

4. Five batch jobs, A through E , arrive at a computer centre at essentially the same time. Their estimated running times, and priorities (lower value is higher priority) are as follows;

job	running time (minutes)	priority
A	15	6
B	9	3
C	3	7
D	6	9
E	12	4

For each of the following scheduling algorithms, determine the turnaround time for each job, and the average turnaround time for all jobs. Ignore process switching overhead and assume all jobs are completely CPU bound.

- (a) non-preemptive priority scheduling

job	turnaround time (minutes)
B	9
E	$9 + 12 = 21$
A	$21 + 15 = 36$
C	$36 + 3 = 39$
D	$39 + 6 = 45$

Hence the average turnaround time is 30 minutes.

- (b) FCFS (in order A, B, C, D, E)

job	turnaround time (minutes)
A	15
B	$15 + 9 = 24$
C	$24 + 3 = 27$
D	$27 + 6 = 33$
E	$33 + 12 = 45$

Hence the average turnaround time is 28.8 minutes.

- (c) shortest job first

job	turnaround time (minutes)
<i>C</i>	3
<i>D</i>	$3 + 6 = 9$
<i>B</i>	$9 + 9 = 18$
<i>E</i>	$18 + 12 = 30$
<i>A</i>	$30 + 15 = 45$

Hence the average turnaround time is 21 minutes.

(d) round robin with a time quantum of 1 minute

The order is as follows; note that a teal quantum is the final quantum for that job.

ABCDEABCDEAB~~C~~DEABDEABDEABDEABEABEABEAEAEAEAAA

job	turnaround time (minutes)
<i>A</i>	45
<i>B</i>	35
<i>C</i>	13
<i>D</i>	26
<i>E</i>	42

Hence the average turnaround time is 32.2 minutes.

Tutorial 4 - Synchronisation

1. Explain why the following statement is false; "when several threads access shared information in main memory, mutual exclusion must be enforced to prevent the production of indeterminate results."

If multiple threads are **reading** the same shared information, then mutual exclusion is not needed. Mutual exclusion is needed if the shared information can be modified.

2. Discuss the pros and cons of busy waiting.

Busy waiting is useful when the wait times are short, as we therefore do not need to invoke the kernel for context switching. Additionally, they are used within the kernel, since it cannot use a blocking abstraction. On the other hand, busy waiting wastes CPU time as it is doing nothing, but occupying the processor.

3. One requirement in the implementation of the semaphore operations **up** and **down** is that each of these operations must be executed atomically; once started, each operation runs to completion without interruption. Give an example of a simple situation in which, if these operations are not executed atomically, mutual exclusion may not be properly enforced.

Assume we are using a binary semaphore, (where the counter is initialised to 1). Two threads are attempting to access a critical region, and therefore both **down** the semaphore. If this wasn't atomic, it's possible that both threads fall into the branch where the counter is greater than 0, and then both attempt to decrement it. This means that neither threads are blocked, and both continue to execute, thus both being in the critical region.

4. Can two threads in the same process synchronise using a kernel semaphore if the threads are implemented by the kernel? What if they are implemented in user space? Assume no threads in other processes have access to the semaphore.

Yes, as the kernel handles the synchronisation by providing the semaphores. On the other hand, user space threads cannot use semaphores, since blocking would block all threads, and therefore user space threads must have another method of synchronisation implemented.

5. Does the strict alternation solution work the same way when process scheduling is preemptive?

Yes, it was designed with preemption. If the scheduling isn't preemptive, and the process that's waiting runs first, it may run indefinitely, since it hasn't completed execution.

6. Give a sketch of how a uni-processor operating system that can disable interrupts could implement semaphores.

Any operations involving a semaphore should disable interrupts right at the start. This ensures atomicity of the semaphores.

When a semaphore is initialised with `init(s, i)`, the **counter** should be set to `i`, and an empty queue is associated with the semaphore. When a semaphore is downed with `down(s)`, it should first check if the counter is greater than 0 - if it is, it decrements the counter, otherwise it adds the calling thread to the queue and blocks the thread (this is atomic). Similarly, if a thread calls `up(s)`, it should disable interrupts, then check the semaphore's queue. If the queue isn't empty, a process in the queue is resumed, otherwise the counter is incremented.

7. Consider the following three threads;

T1:

```
1 a = 1;
2 b = 2;
```

T2:

```
1 b = 1;
```

T3:

```
1 a = 2;
```

- (a) Show all possible thread interleavings.

a = 1; b = 2; b = 1; a = 2;	a = 1; b = 2; a = 2; b = 1;	a = 1; a = 2; b = 2; b = 1;	a = 2; a = 1; b = 2; b = 1;	a = 1; b = 1; b = 2; a = 2;	a = 1; b = 1; a = 2; b = 2;	a = 1; a = 2; b = 1; b = 2;	a = 2; a = 1; b = 1; b = 2;	b = 1; a = 1; b = 2; a = 2;	b = 1; a = 1; a = 2; b = 2;	b = 1; a = 1; a = 2; b = 2;	a = 2; b = 1; a = 1; b = 2;
(2, 1)	(2, 1)	(2, 1)	(1, 1)	(2, 2)	(2, 2)	(2, 2)	(1, 2)	(2, 2)	(2, 2)	(1, 2)	(1, 2)

- (b) If all thread interleavings are as likely to occur, what is the probability to have `a = 1` and `b = 1` after all threads complete execution?

$$\frac{1}{12}$$

- (c) What about `a = 2` and `b = 2`?

$$\frac{5}{12}$$

8. Synchronisation within monitors uses condition variables and two special operations, **wait** and **signal**. A more general form of synchronisation would be to have a single primitive, **waituntil**, that had an arbitrary boolean predicate as a parameter. Thus, one could say;

```
waituntil x < 0 or x + z < n
```

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

This is more expensive, the entire predicate must be re-evaluated every time any of the variables in the predicate change. On the other hand, processes can only be awakened with a **signal** primitive, when using a Hoare monitor.

Tutorial 5 - Deadlocks

- 1.

Tutorial 6 - Memory Management

- 1.

Tutorial 7 - Device Management

1.

Tutorial 8 - Disk Management

1.

Tutorial 9 - File Systems

1.

Tutorial 10 - Security

1.