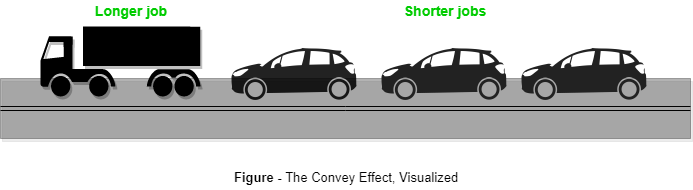
**Convoy Effect in Operating Systems**

**Pre-requisites:** Basics of FCFS Scheduling

Convoy Effect is phenomenon associated with the First Come First Serve (FCFS) algorithm, in which the whole Operating System slows down due to few slow processes.



FCFS algorithm is non-preemptive in nature, that is, once CPU time has been allocated to a process, other processes can get CPU time only after the current process has finished. This property of FCFS scheduling leads to the situation called Convoy Effect.

Suppose there is one CPU intensive (large burst time) process in the ready queue, and several other processes with relatively less burst times but are Input/Output (I/O) bound (Need I/O operations frequently).

**Steps are as following below:**

* The I/O bound processes are first allocated CPU time. As they are less CPU intensive, they quickly get executed and goto I/O queues.
* Now, the CPU intensive process is allocated CPU time. As its burst time is high, it takes time to complete.
* While the CPU intensive process is being executed, the I/O bound processes complete their I/O operations and are moved back to ready queue.
* However, the I/O bound processes are made to wait as the CPU intensive process still hasn’t finished. **This leads to I/O devices being idle.**
* When the CPU intensive process gets over, it is sent to the I/O queue so that it can access an I/O device.
* Meanwhile, the I/O bound processes get their required CPU time and move back to I/O queue.
* However, they are made to wait because the CPU intensive process is still accessing an I/O device. As a result, **the CPU is sitting idle now**.

Hence in Convoy Effect, one slow process slows down the performance of the entire set of processes, and leads to wastage of CPU time and other devices.

**To avoid Convoy Effect, pre-emptive scheduling algorithms like Round Robin Scheduling can be used – as the smaller processes don’t have to wait much for CPU time – making their execution faster and leading to less resources sitting idle.**

1. Give two reasons why caches are useful. What problems do they solve?

**Answer:**

1. Caches are relatively fast than main memory, so getting data from the cache is faster than getting it from the main or secondary memory

2. Caches are also useful when two components need to exchange data, transfers are faster when using cache and the transfer can also be done without having to access the main memory

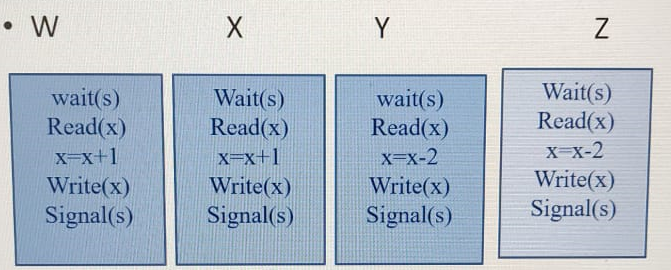
1. What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?

**Answer:**

The main difficulty is keeping the operating system within the fixed time constraints of a real-time system. If the system does not complete a task in a certain time frame, it may cause a breakdown of the entire system it is running. Therefore, when writing an operating system for a real-time system, the writer must be sure that his scheduling schemes don’t allow response time to exceed the time constraint.

**Question:** A shared variable x, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the processes W and X reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes Y and Z reads x from memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes the V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all processes complete execution?

**Semaphore S=2; integer x =0;**

****

**For Maximum Value:**

* Maximum two process can enter in CS
* W enters in CS read x=0 and perform x=x+1 and pre-empted
* Y and Z executed x=-4 ( x= x-4)
* W execute again and write x i.e. 1 and and process X executes x=2

**For Minimum Value:**

* Maximum two process can enter in CS
* Y enters in CS read x=0 and perform x=x-2 and pre-empted
* W and X executed x = 2;
* Y execute again and write x i.e. -2
* Z executes and x=-4

Question: Consider a uniprocessor system executing three tasks T1, T2 and T3, each of which is composed of an infinite sequence of jobs (or instances) which arrive periodically at intervals of 3, 7 and 20 milliseconds, respectively. The priority of each task is the inverse of its period and the available tasks are scheduled in order of priority, with the highest priority task scheduled first. Each instance of T1, T2 and T3 requires an execution time of 1, 2 and 4 milliseconds, respectively. Given that all tasks initially arrive at the beginning of the 1st milliseconds and task preemptions are allowed, In how many milliseconds the first instance of T3 completes its execution?