Operating Systems 2001

Hard-links

True. UNIX does not allow hard-links to span mount points because each path name inside a file-system simply maps to an inode within that file-system. Hence a hard-link across file-systems would require that (a) all relevant file-systems were mounted every time the file were accessed, and that (b) synchronized updates were performed to each copy of the file's inode.

DMA

False. DMA takes some load of the processor (CPU), and can hence improve overall performance. However it does nothing whatsoever to the speed to the device itself.

Paper 1 Question 11 (20 Marks)

Describe how the CPU is allocated to processes if static priority scheduling is used. Be sure to consider the various possibilities available in the case of a tie. [4 marks]

"All scheduling algorithms are essentially priority scheduling algorithms."

Discuss this statement with reference to the first-come first-served (FCFS), shortest job first (SJF), shortest remaining time first (SRTF) and round-robin (RR) scheduling algorithms. [4 marks]

What is the major problem with static priority scheduling and how may it be addressed? [4 marks]

Why do many CPU scheduling algorithms try to favour I/O intensive jobs? [2 marks]

Describe how this is achieved in the (a) UNIX and (b) Windows NT operating systems. [3 marks in each case]

Answers to Paper 1 Question 11

Description of Static Priority Scheduling

In static priority scheduling, each process/job/task (wlog: process) is assigned a priority value, usually an integer within some pre-defined range. The scheduler then chooses to schedule the runnable/ready process with the highest priority.

In the case of a tie, two main options are available[2] preemptive or non-preemptive round-robin. In the former case each of the highest-priority processes is allowed to run for up to Q time units, where Q is called the *quantum* . If a process is still running after its quantum is up, it is preempted and placed on the back of the queue for this priority level. Preemption typically also occurs if a higher priority process becomes runnable once more.

Non-preemptive round-robin allows the highest-priority process to run to completion (or, more likely, "to blocking"). This is seldom used.

"All scheduling is priority scheduling..."

1 mark each for each algorithm. Note that since the RR answer is pushing it a bit I'll be happy to award a mark for an answer which says that RR isn't really a priority algorithm.

FCFS can be considered non-preemptive static priority scheduling where the time of arrival into the system is the priority (earlier values corresponding to higher priorities).

SJF can be considered a non-preemptive dynamic priority scheduling algorithm where the (estimated) job completion time is the priority, and with shorter times corresponding to higher priorities.

SRTF an be considered a preemptive dynamic priority scheduling algorithm in which the remaining time is the priority (shortest first).

It is more difficult to shoe-horn RR into this taxonomy, but just about possible: RR is a preemptive dynamic priority scheduling algorithm in which priority is the amount of time one has been ready waiting for the CPU (with longer waits corresponding to higher priorities), and where a process's "wait time" is reset to '0' (and hence its priority is set to the lowest possible value) once it has been current for an amount of time Q.

Problem with Static Priority Scheduling

2 marks for stating problem, 2 marks for a solution.

The major problem with static priority scheduling is that processes may be starved. This may be solved by making the priorities dynamic, e.g. by aging processes, or by using process accounting information to explicitly adjust priorities.

[2]Marks will be awarded for any other plausible scheme here, provided it is argued.

Why scheduling algorithms favour I/O intensive jobs

Operating systems achieve good throughput by multiplexing together a set of processes/jobs/tasks (wlog: processes). When e.g. a process blocks shortly after being run, this allows a new process to be scheduled while the first process makes progress in its own way (i.e. waiting for I/O completion). As such, I/O intensive processes are "good value" for the scheduler.

Favouring I/O intensive jobs in Unix and NT

In UNIX, the priority of process j at the beginning of time interval i is given by:

$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{4} + 2 \times nice_j$$

where $CPU_j i$ is a measure of how much CPU time process j has been using.

Since higher integer values in UNIX correspond to lower priorities, this means that processes which get a lot of CPU time (i.e. CPU-bound processes) will have their priority dropped over time. This will allow I/O intensive processes to have a go.

Note: the equation is not required; an answer which just explains that UNIX *reduces the priority of a CPU-intensive process is fine.*

In Windows NT/2K, an explicit priority boost is given to I/O bound processes when they are unblocked after the I/O they were waiting for completes. The amount of the boost is device (or device-driver) dependent, and decays over time as and when the process completes a quantum.

Paper 1 Question 12 (20 Marks)

From the point of view of the device driver, data may be read from an I/O device using *polling*, *inerrupt-driven programmed I/O*, or *direct memory access* (DMA). Briefly explain each of these terms, and in each case outline using pseudo-code (or a flow chart) the flow of control in the device driver when reading data from the device. [14 marks]

From the point of view of the application programmer, data may be read from a device in a *blocking*, *non-blocking* or *asynchronous* fashion. Using a keyboard as an example device, describe the expected behaviour in each case. [6 marks]