Operating Systems II 2004
Paper 3 Question 5 (SMH)
Extended Model Answer

Problem RT Sched Solves

2 marks for anything plausible; key thing is to mention the predictability.

RT scheduling solves the problem of providing predictable (deterministic) amounts of CPU time to a set of tasks with precise requirements (typically periodic: e.g. *s*ms per *p*ms). It is not about scheduling things "fast".

RT Sched Algorithms

4 marks for each: both RM and EDF are the examples covered in notes, and hence used for the answer here. However any other static or dynamic RT algs are ok too (e.g. LSTF). The level of detail given below is sufficient for full marks.

A static priority algorithm is rate monotonic (RM). With RM, tasks are assigned priorities dependent on their periods, with higher frequency (smaller period) tasks receiving higher priorities. Admission control must be applied to ensure that the current set of tasks is schedulable; for RM this is guaranteed when the total utilization is $\leq ln2$ (ignoring context switch overhead); if e.g. harmonic periods are enforced, then higher utilizations can be allowed. To implement RM, you would simply require a single priority queue (e.g. impl using a heap) for all runnable tasks, and another time-ordered queue for waiting tasks (i.e. tasks which have received their allocation in their current period). Assuming a periodic timer of sufficiently small granularity, the scheduler would proceed every tick by: (a) decrementing the allocation for this period of the current task (aka head of the runq) and, if zero, moving the current task to the waitq, (b) inserting to the runq any waiting tasks which are at the start of a new period and hence are now runnable again and (c) scheduling the head of the runq.

A dynamic priority algorithm is earliest deadline first (EDF). With EDF, every task has a deadline (aka the end of its period) by which it must receive its allocation; the task with the earliest deadline is scheduled (ties broken arbitrarily). Admission control must be applied to ensure utilization is less than 100overhead). To implement EDF, you require a runq ordered by deadline, and a time ordered waitq as per RM. On entering the scheduler, you first need to scan the waitq from the head and add any tasks with a new allocation (i.e. ones for which a new period has begun) to the runq. Assuming a programable timer, you then schedule the head of the runq for MIN(now + remaining allocation, earliest deadline on waitq if less than deadline of current task). You can also implement EDF with a periodic timer, much as for RM.

Priority Inversion

2 marks for identifying priority inversion, 2 marks for a solution. A detailed description of priority inversion (as below) is not required for the marks, but is included for clarity. There is no need to mention priority ceiling.

The scheduling problem which could occur here is priority inversion; For example, consider a RM system in which there are three tasks T_h, T_m and T_l with high, medium and low priorities respectively (or, equivalently, short, medium and long periods respectively). If T_l gets hold of the lock and is then preempted by T_h (e.g. since it starts a new period and becomes runnable) which attempts to get the lock and then blocks. T_m now runs, meaning T_l cannot run and release the lock. Hence T_h misses its deadline.

This problem can be solved by using priority inheritence. In this scheme, the task holding a lock is temporarily 'boosted' to the priority of the highest priority waiting task. Once it releases the lock, its priority is reduced to normal again.

RT Virtual Memory

2 marks for identifying the problem. 4 marks for a suggestion(s) as to how to overcome it.

The key problem which could occur here is that a task can take a page fault at any time, and this may take sufficiently much time to resolve that the task misses its deadline. One simple technique to avoid this is to ensure that pages are pinned for the duration of execution, but this really removes any benefits of demand paging. A modification (used in some RT OSes) is to only allow page faults which have no disk access (e.g. growing a stack or heap when physical memory is available) – this requires some admission control for memory to ensure a 'bad' situation doesn't occur (at least with high probability). A final solution is to use scheduler activations or something similar where the activation handler is pinned but where threads running over it do not have to be. In this case the user-level thread scheduler can choose to run another thread while asynchronously resolving the page fault. Of course if all threads are waiting fault resolution, the same problem returns, but this is probably the best you can do.