Extended model answer

Operating Systems II 2005 – Paper 3 Question 5 (SMH)

This question is mostly bookwork, based on material from the first part of the course (pages 6–14) on threads and multiprocessor scheduling. The last part is an open-ended bit which requires them to think a bit.

Processes versus threads

1 mark for each part

Basic difference is that processes have separate address spaces while threads do not.

We have both since typically threads are useful for intra-application (cooperative) concurrency when we want to share lots of data at low cost (and potentially minimize context switch overhead, etc) while processes are useful for inter-application (and potentially multi-user) concurrency where isolation rather than sharing is out goal.

Supporting arbitrary number of processes

This is really a programming question; a limit of 32 processes is almost certainly due to using a fixed length process table. To get past this you just need to dynamically allocate process strctures and maintain them in one or more linked lists.

Eliminating starvation

In a nutshell, replace the static priority scheduling algorithm with a dyanmic one. The new scheduling scheme will need to penalize CPU-bound processes (e.g. by reducing their effective priority) and/or reward IO-bound ones (e.g. by introducing a boost to priority). Both of these strategies require an 'aging' process to ensure the penalty/reward does not persist too long.

Effciently schedule on SMP

Basically need to ensure that we (a) don't lose performance due to cache pollution and (b) keep all CPUs busy. The main trick for the former is to have some kinda of processor affinity, either static (i.e. pin processes to individual CPUs) or dynamic (prefer CPU process last ran on but be prepared to move from time to time). For the latter we need to either have a central queue with contended access (which causes some locking overhead) or use something like "take scheduling". (see notes).

Support threads

Two basic variants: pure kernel thread model (so threads are just 'processes' which may share an address space) or a hybrid model like solaris's LWPs, scheduler activations, etc. Key point is that a standard user-level thread package is no good since it will require all threads of a program to reside on one CPU.

Boot faster

This is a fairly free form question. Some answers I'd expect to give might relate to optimizing disk layout so that set of initially loaded binaries + data are sequentially laid out, caching device layout and configuration information or, more aggressively, eliminating "boot" per se and using suspend/resume instead.