SOLUTION NOTES

Operating System Functions 2001 Paper 3 Question 7 (SMH)

Issues with scheduling for multiprocessors

When scheduling for multiprocessors one generally has the dual goals of trying to keep all CPUs busy, while also trying to allow applications to make use of parallelism. The additional problems which must be faced are cache thrashing and thread interaction. Cache thrashing occurs when a thread consecutively executes on different processors, in each case requiring that all of its cache-lines are re-fetched while simultaneously polluting its previous cache for other threads.

Cache thrashing can be overcome by running threads on appropriate processors. Thread interactions are more difficult to characterise: if they are infrequent and small (i.e. involving a small amount of memory), then it may be best to schedule the interacting threads simultaneously. If the interactions are infrequent and large, then scheduling the interacting threads successively on the same processor may be best. And if the interactions are frequent, then it is difficult to get good performance no matter what (cache-line "ping-pong" if simultaneously scheduled, high context-switch overhead if successively scheduled).

Marks will also be given for any plausible alternatives.

Description of multiprocessor techniques

2 marks for each description.

Processor affinity has a shared central ready queue, but associates with each thread with a particular processor, and attempts to schedule that thread in that processor whenever possible.

Take scheduling has a per-processor queue, but arranges that an idle processor can "steal" a thread from a more heavily loaded processor.

Gang scheduling schedules all processors (or at least a subset of them) *together*. In essence, it chooses a given application or task, and schedules its threads on each of the processors in the system.

Problems with processor affinity

The main problem is that if processor affinity is relatively static, then one can end up with load-balancing problems; however if it becomes too dynamic, then little benefit is

obtained from its use. This problem can be overcome by having a suitable dynamic affinity adjustment scheme.

Alternatively, one could argue that the main challenge with processor affinity is determining with which processor a given thread should be associated, particularly if trying to do this dynamically ("cache affinity"). This argument is substantially equivalent to the above.

Applications of gang scheduling

Gang scheduling is most usefully employed when the workload comprises a set of multi-threaded applications, where each application expects its threads to interact somewhat, but not extremely frequently. This sort of model is most commonly observed among scientific applications where the data can be partitioned between simultaneously executing threads (e.g. matrix multiplication, ray tracing, etc.). The widest multiprocessors machines around have traditionally been used for such tasks.

Virtual memory management on multiprocessors

This is a somewhat open-ended question. Marks will be awarded for anything plausible.

For UMA machines, one of the most obvious things the VM system has to deal with is handling (and synchronising) multiple copies of various data structures. In particular per-process page tables will need to be updated synchronously if e.g. threads from the same application are executing concurrently. Similarly TLBs must be flushed on multiple processors when a permission change occurs (although note that if the change is to "more permissive access rights", flushes can be handled lazily).

As well as having to synchronise updates to page tables, one must also aggregate statistics in order to aid e.g. page replacement. It also becomes necessary to consider whether or not replacement should consider process-level or thread-level locality.

Introspecting for a moment, it is also necessary that the VM system itself be multiprocessor friendly; i.e. ensure that the layout and use of its data structures don't encourage poor cache-line sharing. It may also wish to include parallel versions of various tasks like page scanning and replacement.

It is also necessary to rethink how to share physical memory between the processors – this is a more difficult version of the frame allocation problem in uniprocessors since the working set of any given process may comprise $> 1/n^{th}$ of total memory.

Furthermore, a reasonable argument can be made for taking account of memory management information (such as working set sizes and actual members) when scheduling – i.e. have some sort of "holistic" scheduling scheme.

For NUMA machines care must be taken about deciding the "home memory" for any

given page. This issue is strongly analogous with cache coherence in UMA machines, except the policies used and the actions taken are performed in software. Some of the considerations here are: to which memory should a given page first be assigned; under which circumstances should a page be moved ("migrated") to another memory; in which circumstances does it make sense to have copies of pages in multiple memories; what form of consistency is really required; is partial page locking (or "object based" locking) worth while; and so on.

If the NUMA machine extends to having more than two levels of memory access time in its hierarchy (e.g. either by being non-CC NUMA, or by explicitly adding an additional level), then things become even more complicated. In particular, migration costs become variable depending on where something is migrating too. Since such architectures also tend to include more processors, the VM system must also deal with real k-way sharing. At this stage, DSVM coherence techniques begin to seem tempting.