Operating Systems 2 - Supervision 1

Matthew Johnson
Trinity Hall
mjj29@cam.ac.uk

February 24, 2003

1 A

1.1

Kernel threads are scheduled within the kernel itself. It can pre-empt threads, and assign them between processors in SMP machines. Applications need not be aware of the threading or scheduling. User threads, however, are maintained by the application programmer. The kernel schedules one user-space application, and within that the application maintains its own scheduler that will run different threads. These threads tend to have to be cooperative, not preemptive, because the user cannot easily preempt them. User threads are lighter, and switch faster, because it doesn't require a full context switch, or a jump to kernel mode, to change between them. If the kernel is unaware of the threads' existence, that is because it merely schedules the single application, and as far as it is concerned that is the unit of scheduling. It is not involved in thread-switching within the application.

1.2

A thread is a unit of processing time, whereas a process is also a unit of storage allocation and so on. Typically a process would spawn multiple threads. Processes have their own address space, which threads share, whereas threads just have a stack pointer and a program counter.

1.3

¡see below¿

2 B - y2001p3q7

2.1

When sheduling for multi-processor machines the key issues are keeping the load well-distributed between processors - so that one processor isn't left idle with processes only running on the others. Secondly, you have to allow user programs to exploit the parallellism of the system. Finally, you have to avoid saturating the memory, so that every context switch pagefaults and requires swapping things into memory, and refreshing the caches.

2.2 Scheduling algorithms

2.2.1 Processor Affinnity

Processes are labelled with which processor they would like to run on. This makes it more likely that data will still be in the cache of the processor it gets cscheduled on. However, if some processes are running for a long time then other processors may fall idle, while several processors are still running on the remaining processor. It tries to overcome the cache thrashing issue.

2.2.2 Take Scheduling

Processes are scheduled in a per-cpu queue. When utilisation of a processor drops, it takes a process from a different queue. to run on this. This is designed to make the load more even over CPUs. It will also have quite good cache affinity since threads will generally spend a while on the same CPU.

2.2.3 Gang scheduling

Related threads are grouped together, and all scheduled at once on different processors. This allows programs to do more parallellisms. This works will if the threads are relatively independent, and aren't needing to share data a lot.

2.3

When dealing with shared-memory multiprocessor machines, the VM system has to consider the coherency of the data in the caches and the main memory. Bus snooping and other techniques can help to ensure that if data is written to memory, all the caches can be updated if necessary with the new data. With write-caching you have to be more careful, to make sure that an update happens before any reads are made to that location. Page locking can be used to stop other caches becoming out of sync, but then processors will block on trying to access that page.

3 C - y200p3q7

The scheduling algorithms used in mainstream operating systems make no regard to strict deadlines on the scheduling of specific processes. They aim to get the highest total throughput, and the least time spent waiting on external devices. Real-time systems, on the other hand, need to be able to guarantee that a process will get a certain amount of processor time before a fixed deadline. This also means limiting the number of processes able to be scheduled in total - a facility which most mainline operating systems also don't have.

3.1 Real Time Schedulars

Rate Monotonic (RM) scheduling assigns static priorities to each process, and gives tasks with earlier deadlines higher priorities. Earliest Dealine First (EDF) is an alternative scheme.

If the total utilisation is less than $\ln 2$ then it is schedulable under RM. (total utilasation $x = \sum_{i=1}^{n} \frac{C_i}{T_i}$. C_i is the amount of CPU time / period, and T_i is the period.). if $0 < x < \ln 2$ it is schedulable under RM, if $0 < x < 1$ it is scedulable under EDF, otherwise it is not schedulable.

4 D - y1997p4q7

4.1

Thrashing is a state which a system enters when it has not enough real memory for the currently running processes. The VM swaps something out of memory onto disk to schedule a process, and then when scheduling the next process immediately has to swap back out that data for the next one. This reduces the system to a completely IO-bound state, and brings the machine to a halt.

4.2

The working set of a process is the minimum number of pages that it needs to make some progress when scheduled. If this is not in memory, then the process will have to swap pages in when it is scheduled. It can be computed by marking pages when they are used, and periodically clearing them.

5 D - y2001p4q7

5.1

5.1.1

Demand paging is when the VM system only pages a file into memory when it is requested. It is implemented by letting a process page fault when it tries to access an area of memory, then the interrupt handler will call the VM system to load that page in from disk, swapping out other pages as necessary. The process is then restarted at the instruction that cause the page fault - and this time the page fault will not happen. This means you are guarenteed 1 page fault per page. When starting programs, however, you will always need some data - and a predictable set of data. Demand paging makes it easy to implement hardware access as a feature of the VM (cf Windows NT)

5.1.2

Temporal locality of reference means that the memory locations are usually accessed relatively soon after each other.

5.1.3

Least-recently used, and not-recently used schemes can be used when choosing pages to swap to disk.

5.1.4

Spatial locality of reference implies that areas on disk that are close together are likely to be accessed together too

5.1.5

Assuming spatial locality allows you to take advantage of read-aheads, and burst reads - as well as cache lines consisting of several blocks, to read other data which is near the requested data. Although it has not been requested yet, it is likely to be needed soon.

5.2

In terms of the program-level view of memory, dividing it into objects and procedures makes sense. When running a program it will typically require all of one object at once, and execute the whole of one procedure. This means that you can load in only that which is exactly required. However, these items would be variable sized, which makes allocation and management of them quite difficult. It is also the case that it is more efficient to move larger block around, and most programs will make use of the larger block - so internal fragmentation in pages is not much of a problem, whereas with the suggested system external fragmentation would be. (cf segmentation)

6 Notes

6.1 Monolithic Operating Systems

For example, DOS - where all of the system has the same priviledges, including user programs.

6.2 Kernel-based Operating Systems

Example: Unix. These introduced kernel & user modes of the processor. There is protection between user programs and the kernel. If user programs want to do priviledged operations they make calls to the kernel - which contain all of the functions that require priviledged access. These have the disadvantage of there being a lot of code in the kernel, which may be prone to break - all of which is running in priviledged mode.

6.3 Micro-kernels

A lot of the functionality of the kernel is moved to user-level 'servers' which in turn call very stripped down stubs in the kernel itself. This reduces system performance due to the amount of IPC, but increases stability and security. The original NT kernel, and the HURD are examples of this.

6.4 Vertically Structured Operating Systems

The operating system code is implemented in user code itself, via shared libraries. Again there is a very small supervisor that manages protection. This has no IPC for operating system calls and is a lot faster than micro-kernels, and has all the stability & security features. There is, however, more context switching overhead, and you can't keep the kernel state so well. The Nemesis project was an example of this.