Process Scheduling and Implementation

Scheduling

Last lecture we discussed half of process scheduling, when a process gives up the CPU. Today we start with the other half, which process is scheduled to take its place. This is our first introduction to scheduling algorithms which will be a repeating topic in the course. Operating systems schedule pages of memory, disk blocks, and several other things. The algorithms discussed today and variations on them tuned for specific other applications are important tools for your bag of OS degisn tricks.

Why not just pick a process at random? Congratulations, that's a scheduling discipline - *random scheduling.* It has the advantages that it's easy to implement, but gives somewhat unpredicatable results. **N. B.** if you have a homogeneous set of jobs, it may be an effective scheduling mechanism!

All scheduling mechanisms involve design tradeoffs. The relevant parameters to trade off in process scheduling include:

Response Time	Time for processes to complete. the OS may want to favor certain types of processes or to minimize a statistical property like average time
Implementation Time	This includes the complexity of the algorithm and the maintenance
Overhead	Time to decide which process to schedule and to collect the data needed to make that selection
Fairness	To what extent are different users' processes treated differently

Some Scheduling Disciplines

First-In-First-Out (FIFO) and Round Robin

The ready queue is a single FIFO queue where the next process to be run is the one at the front of the queue. Processes are added to the back of the ready queue. This is a simple discipline to implement, and with equal sized quanta on a preemptive scheduling system results in each process getting roughly an equal time on the processor.[1] In the limit, i.e., a preemptive system with a quanta the size of one machine instruction and no context switch overhead, the discipline is called *processor sharing,* and each of *n* processes gets *1/n* of the CPU time.

As quanta get larger, FIFO tends to discriminate against short jobs that give up the CPU quickly for I/O while long CPU-bound jobs hold it for their full quantum.

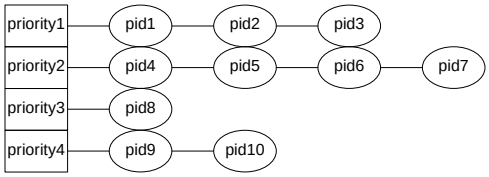
Priority Scheduling

FIFO is eqalitarian - all processes are treated equally. It is often reasonable to discriminate between processes based on their relative importance. (The payroll calculations may be more important than my video game.)

One method of handling this is to assign each process a priority and run the highest priority process. (What to do at on a tie puts us back to square 1 - we pick a scheduling policy).

---

[1] Tannenbaum considers Round Robin only in this context, but FIFO process scheduling is commonly used in batch systems or non-preemptive systems as well.



This solves FIFOs problem with interactive jobs in a mixed workload. Interactive jobs are given high priority and run when there are some. Lower-priority CPU-bound jobs share what's left. Particularly aggressive priority schedulers reschedule jobs whenever a job moves on any queue, so interactive jobs would be able to run immediately after their I/O completes.

The CTSS system in Tannenbaum uses a different quantum at each priority scheduling level. More complex systems have rules about moving processes between priority levels. (Systems that move processes between mutliply priorities based on their behavior are sometimes called *mulit-level feedback queues*.)

Priority Problems - Starvation and Inversion

When processes cooperate in a priority scheduling system, there can be interactions between the processes that confust the priority system. Consider three processes, A, B, C; A has the highest priority (runs first) and C the lowest with B having a priority between them. A blocks waiting for C to do something. B will run to completion even though A, a higher priority process, could continue if C would run. This is sometimes referred to as a *priority inversion.* This happens in real systems - the Mars Rover a couple years ago suffered a failure due to a priority inversion.

Starvation is simpler to understand. Imagine our 2-level priority system above with an endless, fast stream of interactive jobs. Any CPU-bound jobs will be never run.

The final problem with priority systems is how to determine priorities. The can be statically allocated to each program (ls always runs with priority 3) or each user (root always runs with priority 3), or computed on the fly (process aging). All of these have their problems. For every scheduling strategy, there is a counter-strategy.

Shortest Job First (SJF)

An important metric of interactive job performance is the response time of the process (the amount of time that the process is in the sysytem, i.e., on some queue). SJF minimizes the average repsonse time for the system. Processes are labelled with their expected processing time, and the shortest one is scheduled first.

$$\text{response time} = \frac{\sum_{i=0}^{n}(n-i)t_i}{n}$$

The problem, of course, is determingin those response times. For batch processes that run frequently, guesses are easy to come by. Other programs have run times that vary widely (e.g. a prime tester runs quickly on even numbers and slowly on primes). In general, the problem of determining run times *a priori* is impossible.

There is hope, however, in the form of heuristics (that is, algorithms that provide good guesses). The simplest is to use a moving average. An average run time is kept for each program, and after each run of that program it is recomputed as:

$$\text{estimate} = \alpha\ (\text{old estimate}) + (1-\alpha)\ \text{measurement}\quad (\text{for } 0 < \alpha \leq 1 \text{ and constant})$$

Moving averages are another powerful tool for your design toolkit.

Other Scheduling Systems

There are other specialized scheduling mechanisms discussed in the text and other places. And it is easy to derive scheduling systems from the ones discussed today (LIFO, LJF). In all cases they have to be

evaluated on how well they influence their figure of merit (e.g., system response time) and the overhead of implementing them (e.g., estimating run time).

For every scheduling strategy, there is a counter-strategy.

Process Implementation

The operating system represents a process primarily in a data structure called a Process Control Block (PCB). You'll see Tack Control Block (TCB) and other variants. When a process is created, it is allocated a PCB that includes

- CPU Registers
- Pointer to Text (program code)
- Pointer to uninitialized data
- Stack Pointer
- Program Counter
- Pointer to Data
- Root directory
- Default File Permissions
- Working directory
- Process State
- Exit Status
- File Descriptors
- Process Identifier (pid)
- User Identifier (uid)
- Pending Signals
- Signal Maps
- Other OS-dependent information

These are some of the major elements that make up the process context. Although not all of them are directly manipulated on a context switch.

Context Switching

The act of switching from one process to another is somewhat machine-dependent. A general outline is :

- The OS gets control (either because of a timer interrupt or because the process made a system call.
- Operating system processing info is updated (pointer to the current PCB, etc.)
- Processor state is saved (registers, memory map and floating point state, etc)
- This process is replaced on the ready queue and the next process selected by the scheduling algorithm
- The new process's operating system and processor state is restored
- The new process continues (to this process it looks like a block call has just returned, or as if an interrupt service routine (not a signal handler) has just returned

Context switches must be made as safe and fast as possible. Safe because isolation must be maintained and fast because any time spent doing them is stolen from processes doing useful work. Linux's well-tuned context switch code runs in about 5 microseconds on a high-end Pentium.