Scheduling Part 1
Sample Sorts of Systems

- Simple batch
- Multiprogrammed batch
- Time sharing
- Partitioned servers
- Real time

*Simple batch systems*. These probably don’t exist anymore, but were common into the 1960s. Programs (jobs) were submitted and ran without any interaction with humans, except for possible instructions to the operator to mount tapes and disks. Only one job ran at a time. The basic model is a queue of jobs waiting to be run on the processor. The responsibility of the scheduler was to decide which job should run next when the current one finished. There were two concerns: the system throughput, i.e., the number of jobs per unit time, and the average wait time — how long did it take from when a job was submitted to the system until it completed.

*Multiprogrammed batch systems*. These are identical to simple batch systems, except that multiple jobs are run at once. Two sorts of scheduling decisions have to be made: how many and which jobs should be running, as well as how the processor is apportioned among the running jobs.

*Time-sharing systems*. Here we get away from the problem of how many and which jobs should be running and think more in terms of apportioning the processor to the threads that are ready to execute. The primary concern is wait time, referred to here as response time — the time from when a command is given to when it is completed.

*Partitioned servers*. In some situations we might have a single large computer that’s to be treated as if it were a number of independent ones. For example, a large data-processing computer might be running a number of different on-line systems each of which must be guaranteed a certain capacity or performance level — we might want to guarantee each at least 10% of available processing time. Another strategy, perhaps due to marketing, might be to provide someone with, say, exactly 10% of machine capacity. This might be desired if we are providing web-hosting services.

*Real-time systems*. These are a range of requirements, ranging from what’s known as “soft” real-time to “hard” real-time. An example of the former is a system that plays back streaming audio or video. It’s really important that most of the data be processed in a timely fashion, but it’s not a disaster if occasionally some data isn’t process on time (or at all). An example of the latter is a system controlling a nuclear reactor. It’s not good enough for it to handle most of the data in a timely fashion; it must handle all the data
in a timely fashion or there will be a disaster.
The slide lists three possible aims of an operating system’s scheduler. Note that “timely response” and “quick response” are two very different things!
Timely Response

- “Hard” real time
  - chores must be completed on time
    - controlling a nuclear power plant
    - landing (softly) on Mars

Providing timely response is the realm of what’s known as hard real-time systems, where there are various chores that simply must be completed on time (if not, disaster occurs). What’s important in such systems is predictable behavior.
Fast response is what’s required of soft real-time systems, where timely response is important, but disaster doesn’t occur if time requirements are not met. However, it is important that the requirements are met “most of the time.”
Sharing

- All active threads share processor time equally
Scenario

- Scheduling “jobs”
- Run one at a time
- Running time is known
The right half of the graph is actually a “tilted staircase.”
Average Wait Time

- Jobs $J_i$ with processing times $T_i$
- **Average wait time (AWT)**
  - $J_i$ started at time $t_i$
  - $AWT = \frac{\text{sum}(t_i + T_i)}{n}$
  - $t_i = \text{sum}_{j=0}^{i-1}(T_j)$
- For our example
  - $AWT = 252$ hours
Shortest Job First

- \( AWT = \frac{\text{sum}(t_i + T_i)}{n} \)
  - \( t_i = \text{sum } j=0 \text{ to } i-1(T_j) \)
- \( AWT = (nT_i0 + (n-1)T_{i1} + \ldots + 2T_{in-2} + T_{in-1})/n \)
- Minimized when \( i_j \) chosen so that
  - \( T_{ij} \leq T_{ij+1} \)
  - which is *shortest job first*

*Shortest-job-first* is another fairly straightforward algorithm, in this case one that minimizes *average wait time* for a set of jobs.
SJF and Our Example

Throughput

AWT = 86 hours

Time (hours)

Jobs/hour

Operating Systems In Depth XI-12
Preemption

- Current job may be preempted by others
  - shortest remaining time next (SRTN)
    - optimized throughput
Fairness

- FIFO
  - each job eventually gets processed
- SJF and SRTN
  - a long job might have to wait indefinitely
- What’s a good measure?
Round-robin is a scheduling algorithm used in conjunction with time slicing: each thread runs for up to a certain length of time (known as the quantum), then is preempted by the next thread and goes to the end of the line.
Quiz 1

We implement a round-robin scheduler. Jobs are served from the queue in FIFO order with a fixed time quantum. After a job has executed for its quantum, it's preempted and goes to the end of the queue.

Does this scheduler improve the average wait time (compared to SJF) if applied to our example?

a) yes
b) no
Round Robin + FIFO

- AWT?
  - let quantum approach 0:
    - 169 jobs sharing the processor
    - run at 1/169th speed for first week
    - short jobs receive one hour of processor time in 169 hours
    - long job completes in 336 hours
    - AWT = 169.99 hours
    - average deviation = 1.96 hours
    - for FIFO, average deviation = 42.25 hours
Interactive Systems

- Length of “jobs” not known
- Jobs don’t run to completion
  - run till they block for user input
- Would like to favor interactive jobs
A variant of round robin is to add priority: we have a number of queues, one for each priority level. A thread at a lower priority level cannot run if there are any threads at a higher priority level. Within priority levels, time slicing is used to obtain round-robin scheduling.
Another variant of round-robin scheduling is what’s known as *multi-level feedback queues*. This is similar to round robin with priorities, except that all threads start at the same priority. When a thread’s time slice expires, it is moved to the end of the queue for the next-lower priority level. The time quanta get progressively longer as the priority decreases, culminating in infinite at the lowest level (i.e., there is no time slicing at the lowest priority level). The advantage of this scheme is that it gives favored treatment to short jobs, at the expense of longer ones. However, it does take into account the fact that a thread that was running for a long time might sleep for a bit, then run for short bursts.
Interactive Scheduling

- Time-sliced, priority-based, preemptive
- Priority depends on expected time to block
  - interactive threads should have high priority
  - compute threads should have low priority
- Determine priority using past history
  - processor usage causes decrease
  - sleeping causes increase
Scheduling in Early Unix

- Interactive applications
  - shell, editors
- Lengthier applications
  - compiles
- Long-running applications
  - computing π
The scheduler of 6th-edition Unix was remarkably simple. First of all, the lower the numerical priority, the better. The variable \( p_{\text{cpu}} \) was maintained for each process and was incremented at each clock interrupt (either 50 Hz or 60 Hz, depending on the electric current). PUSER was a scaling factor to allow certain system activities to have a better priority than user processes. \( p_{\text{nice}} \) allowed one to force a process to run in the “background” — running only when no other process was running.

The scheduler would run the thread whose priority had the lowest numeric value. The basic idea was that a thread’s priority grew steadily worse while it was executing and steadily better while it was not executing.

Note that the “=+” in the third bullet is not a typo — this was the correct syntax at the time.
The scheduler of early versions of BSD Unix used a dynamically computed priority that was computed in a similar fashion as that of the earlier 6th-edition Unix.
Quiz 2

The UNIX schedulers seen so far work on the following principle:

- priority steadily gets worse while a thread is running
- priority steadily gets better while a thread is not running

a) These schedulers work fine under heavy load
b) These schedulers don't work well under heavy load because they use the above principle
c) These schedulers don't work well under heavy load because they improperly implement the above principle
The early scheduler did poorly on heavily loaded systems. Since there were so many threads competing for the processor, all threads were waiting such a long time to run that all had a good priority and thus interactive threads had same priority as compute-bound threads.