Microkernels
(continued)

Details about Mach can be obtained from “Mach 3 Kernel Principles” by Keith Loepere, available at http://www.shakthimaan.com/downloads/hurd/kernel_principles.pdf.

Successful Microkernel Systems

-
-
- …

Attempts

- Windows NT 3.1
  - graphics subsystem ran as user-level process
  - moved to kernel in 4.0 for performance reasons
- Apple OS X
  - based on Mach
  - all services in kernel for performance reasons
- HURD
  - based on Mach
  - services implemented as user processes
  - no one used it, for performance reasons

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.

Scheduling

- **Aims**
  - **provide timely response**
  - **provide quick response**
  - **use resources equitably**

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

- "Soft" real time
  - the longer it takes, the less useful a chore's result becomes
    - responding to user input
    - playing streaming audio or video

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

FIFO

Throughput

- "Goodness" criterion is jobs/hour
- One 168-hour job
- Followed by 168 one-hour jobs

The right half of the graph is actually a "tilted staircase."

Average Wait Time

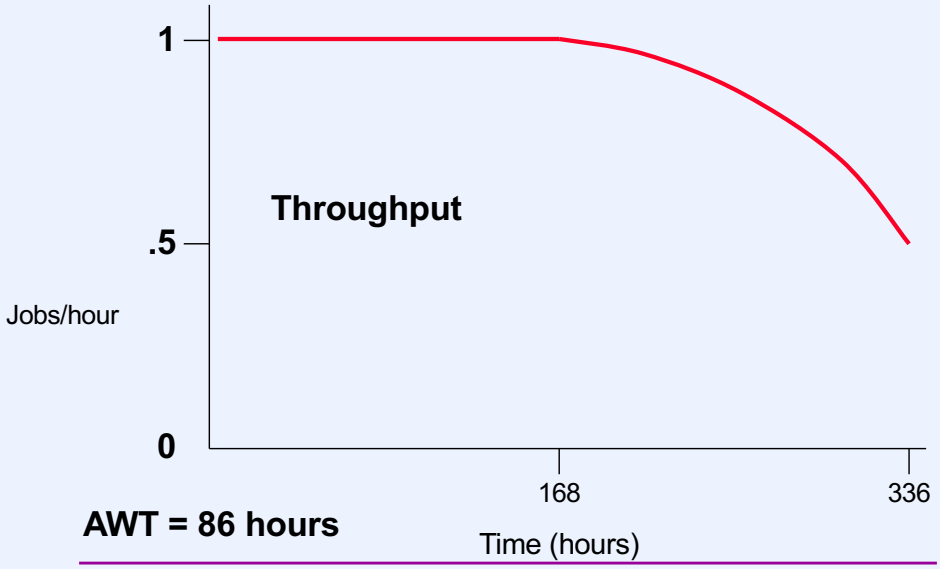
- Jobs J_i with processing times T_i
- *Average wait time* (*AWT*)
  - J_i started at time t_i
  - $AWT = sum(t_i + T_i)/n$
    - $t_i = sum_{j=0 \text{ to } i-1}(T_j)$
- For our example
  - AWT = 252 hours

Shortest Job First

- $AWT = sum(t_i + T_i)/n$
  - $t_i = sum_{j=0 \text{ to } i-1}(T_j)$
- $AWT = (nT_{i_0} + (n-1)T_{i_1} + \ldots + 2T_{i_{n-2}} + T_{i_{n-1}})/n$
- Minimized when i_j chosen so that
  - $T_{i_j} <= T_{i_{j+1}}$
  - 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



AWT = 86 hours

Preemption

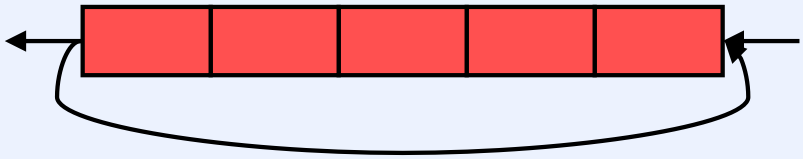
- Current job may be preempted by others
  - shortest remaining time next (SRTN)
    - optimized throughput

This handles situations in which some jobs enter the system after it has started. Thus, a really short job may enter the system late, but should be immediately scheduled so as to minimize average wait time.

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



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? Assume the time quantum is close to 0.

a) yes

b) no

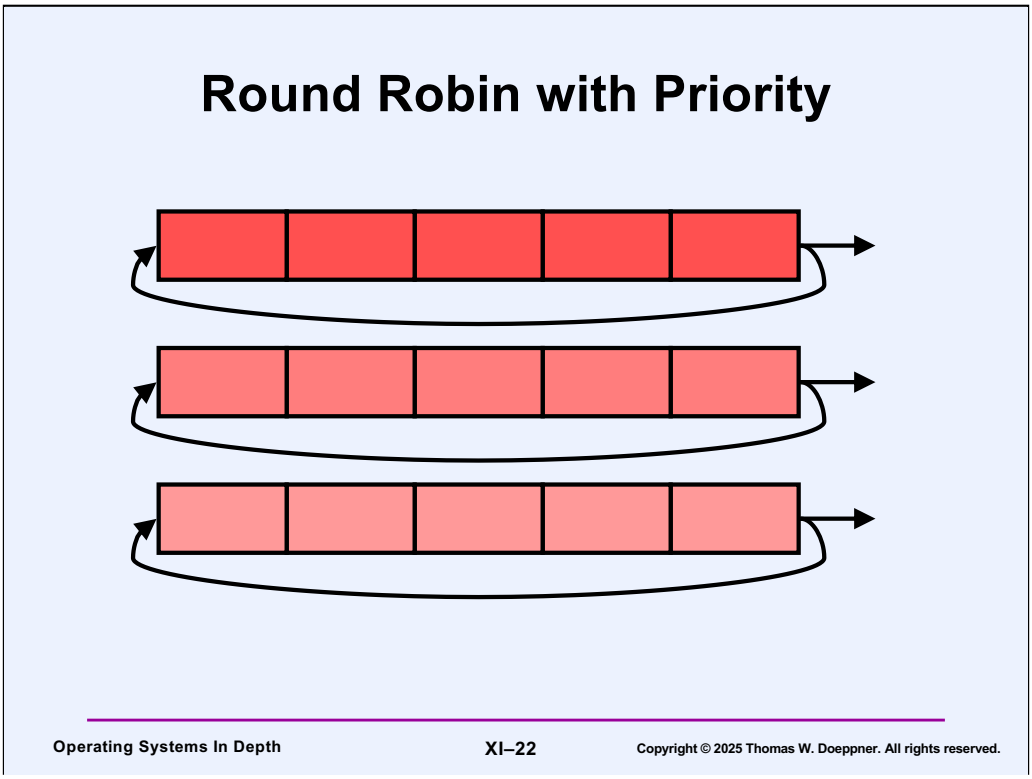
Round Robin + FIFO

- 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

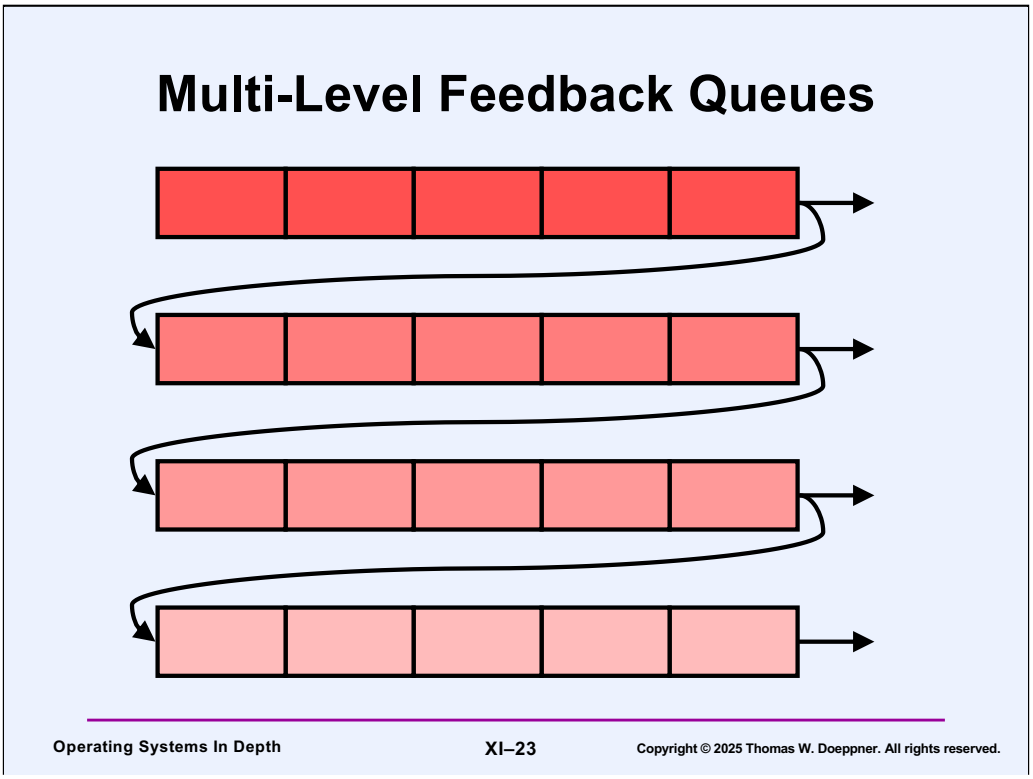
Recall that with SJF, AWT was 86 hours.

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.

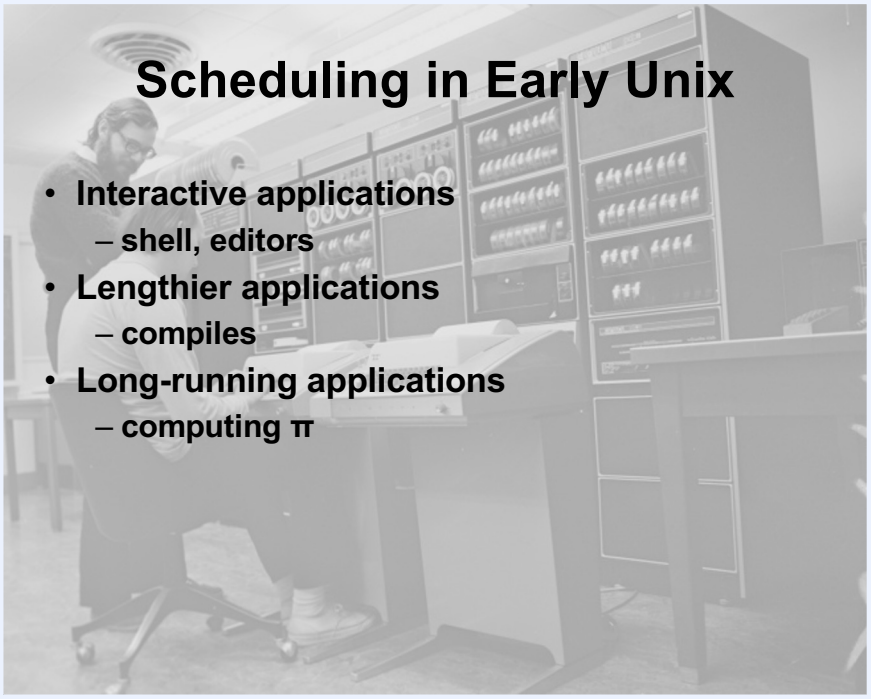


Another variant of round-robin scheduling is what's known as *multi-level feedback queues*. This is similar to round robin with priorities, except that all threads start at the same priority. When a thread's time slice expires, it is moved to the end of the queue for the next-lower priority level. The 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 not 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

6th-Edition Unix Scheduling

- Process priority computation
  - p = (pp->p_cpu & 0377)/16;
  - p =+ PUSER + pp->p_nice;
  - (numerically low priorities are better than numerically high priorities)
- Every "clock tick"
  - current process: p_cpu++
- Every second
  - all processes: p_cpu = max(0, p_cpu − 10)
- Every four seconds
  - force rescheduling
    - time quantum

The scheduler of 6th-edition Unix was remarkably simple. The variable **p** holds a process's priority; the scheduler chooses to run the process with the lowest value of **p**. The variable **p_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_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.

Early BSD Unix Scheduler

- priority = c_1 + (cpuAvg/4) + c_2·nice
  - thread priority, computed periodically
  - lowest-priority thread runs
- cpuAvg++
  - every .01 second, while thread is running
- cpuAvg = (2/3)·cpuAvg
  - computed once/second for each thread
- time quantum is .1 second

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. Each thread has its own **cpuAvg** variable. The scheduler chooses to run the runnable thread with the lowest value of **priority**.

Note that a thread's **cpuAvg** increases while it is running and thus effectively increases the thread's numeric priority value (making it less likely to continue to run). Every second the **cpuAvg**'s for all threads are reduced (decayed), so that waiting threads get better priorities. Thus a thread's priority gets worse while it is running and gets better while it is waiting to run.

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 incorrectly implement the above principle

c) These schedulers don't work well under heavy load because they correctly implement the above principle

By "work fine" the scheduler should continue to favor interactive processing over non-interactive processing.

Later BSD Unix Scheduler

- priority = c_1 + (cpuAvg/4) + c_2·nice
  - thread priority, computed periodically
- cpuAvg++
  - every .01 second, while thread is running
- cpuAvg = ((2·load)/(2·load+1))·cpuAvg
  - *load* is the short-term average of the sum of the run-queue size (including current thread) and the number of "short-term" sleeping threads
  - computed once/second for each thread
- time quantum still .1 second

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 many cpu-bound threads had low values of **cpuAvg** (due to the decay described by the third bullet). This meant that when an interactive thread became runnable, its priority was no better than a lot of cpu-bound threads, and thus it had to wait a fair amount of time before being scheduled.

The modification shown in the slide affects how **cpuAvg** is reduced (decayed) for threads that are waiting to run – the higher the load, the slower **cpuAvg** is reduced. Thus, on a busy system, **cpuAvg** for long-waiting cpu-bound threads is decayed much more slowly when the system is busy, and thus they will tend to have worse priorities than interactive threads.

Shared Servers

- You and four friends each contribute $1000 towards a server
  - you, rightfully, feel you own 20% of it
- Your friends are into threads, you're not
  - they run 5-threaded programs
  - you run a 1-threaded program
- Their programs each get 5/21 of the processor
- Your programs get 1/21 of the processor

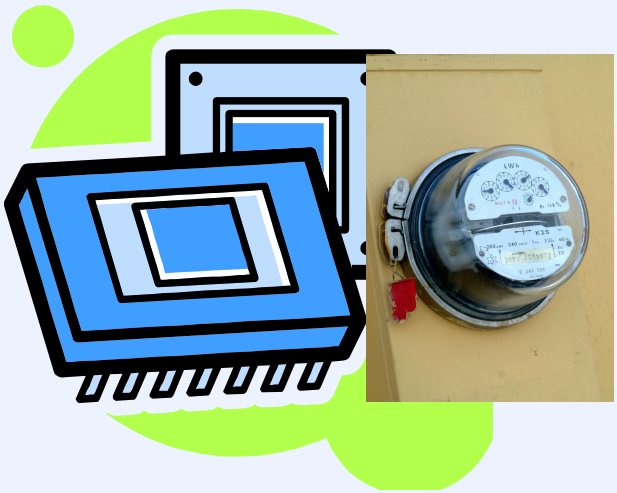
Lottery Scheduling

- 25 lottery tickets are distributed equally to you and your four friends
  - you give 5 tickets to your one thread
  - they give one ticket each to their threads
- A lottery is held for every scheduling decision
  - your thread is 5 times more likely to win than the others

Proportional-Share Scheduling

- Stride scheduling
  - 1995 paper by Waldspurger and Weihl
- Completely fair scheduling (CFS)
  - added to Linux in 2007

Metered Processors



To measure the usage of a processor, let's assume the existence of a meter.

Algorithm

- Each thread has a meter, which runs only when the thread is running on the processor
- At every clock tick
  - give processor to thread that's had the least processor time as shown on its meter
  - in case of tie, thread with lowest ID wins

Assuming all threads are equal, all started at the same time, and all run forever, the intent is to share the processor equitably. Note that has the time between clock ticks approaches zero, each thread gets 1/n of total processor time, where n is the number of threads.

Issues

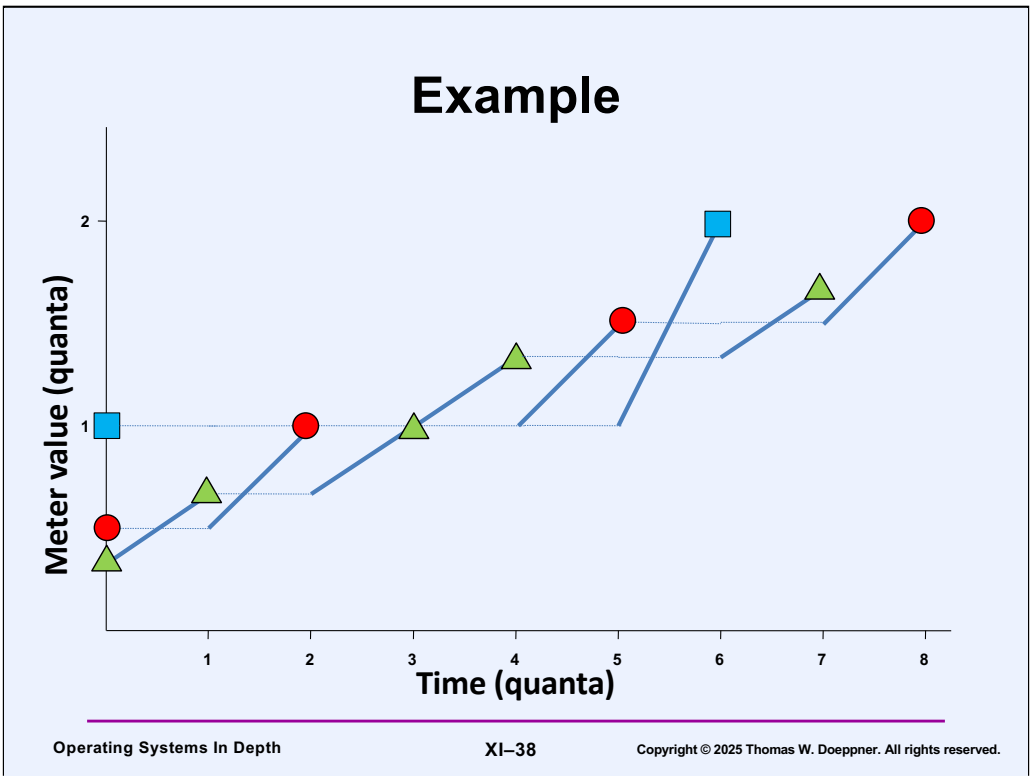
- Some threads may be more important than others
- What if new threads enter system?
- What if threads block for I/O or synchronization?

Details …

- Each thread pays a bribe
  - the greater the bribe, the slower the meter runs
  - to simplify bribing, you buy "tickets"
    - one ticket is required to get a fair meter
    - two tickets get a meter running at half speed
    - three tickets get a meter running at 1/3 speed
    - etc.

New Algorithm

- Each thread has a *(possibly crooked)* meter that runs only when the thread is running on the processor
- Each thread's meter is initialized as 1/bribe
- At every clock tick
  - give processor to thread that's had the least processor time as shown on its meter
  - in case of tie, thread with lowest ID wins



The slide illustrates the execution of three threads using stride scheduling. Thread 1 (labeled with a triangle) has a paid a bribe of three tickets. Thread 2 (labeled with a circle) has paid a bribe of two tickets, and thread three (labeled with a square) had paid only one ticket. The thicker lines indicate when a thread is running. Their slopes are proportional to the meter rates (and inversely proportional to the bribe). Note that meter values on the y axis are twice as far apart as ticks on the x axis.

In this example, a total bribe of six tickets has been paid. After six clock ticks, each thread's meter has been increased by 1.

In general, if the clock ticks once per second and the total bribe is B, then after B seconds, each thread's meter has increased by exactly 1. To see this, assume that each thread t_i starts with a meter reading of the reciprocal of its bribe b_i. To make this easier, let's assume that each thread has paid a different bribe. Suppose thread t_1 paid the largest bribe, b_1. After some period of time its meter will have increased by 1, requiring b_1 seconds of actual execution. Since it's the thread that paid the largest bribe (and thus its meter's initial value is the smallest), its meter will be increased by 1 before that of any other thread. It of course won't run again until its meter has the lowest value. Thread t_2, which paid the second largest bribe, will be the second thread to have its meter increased by 1, requiring b_2 seconds of actual execution. It also won't run again until its meter has the lowest value. Similar arguments can be made for the remaining threads, through t_n. Once t_n's meter has been increased by 1, t_1 again has the lowest meter value and the cycle starts again. The total amount of time required to get to this point is $b_1 + b_2 + \ldots + b_n$, i.e., the total bribe.

Quiz 3

Suppose n threads are being scheduled; assume thread i payed bribe B_i. After X clock ticks, each thread's meter will be incremented by 1. What is X?

a) n

b) $\sum_{i=0}^{n-1} B_i$

c) $n \cdot \sum_{i=0}^{n-1} B_i$

d) none of the above

Example

- Three threads
  - T_1 has one ticket: meter_rate = 1
  - T_2 has two tickets: meter_rate = 1/2
  - T_3 has three tickets: meter_rate = 1/3
  - six total tickets
- After 6 clock ticks
  - T_1's meter incremented by 1 once
  - T_2's meter incremented by 1/2 twice
  - T_3's meter incremented by 1/3 three times
- Each meter shows increase of 1
  - what "1" means depends on the bribe

Assuming six total tickets, after six clock ticks each thread sees its meter being increased by 1. Thus, after each cycle, all threads have the same values shown on their meters, but what these values mean in terms of time is proportional to the number of tickets they've purchased.