Microkernels
(continued)

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

Scheduling

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

Timely Response

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

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

Sharing

- All active threads share processor time equally

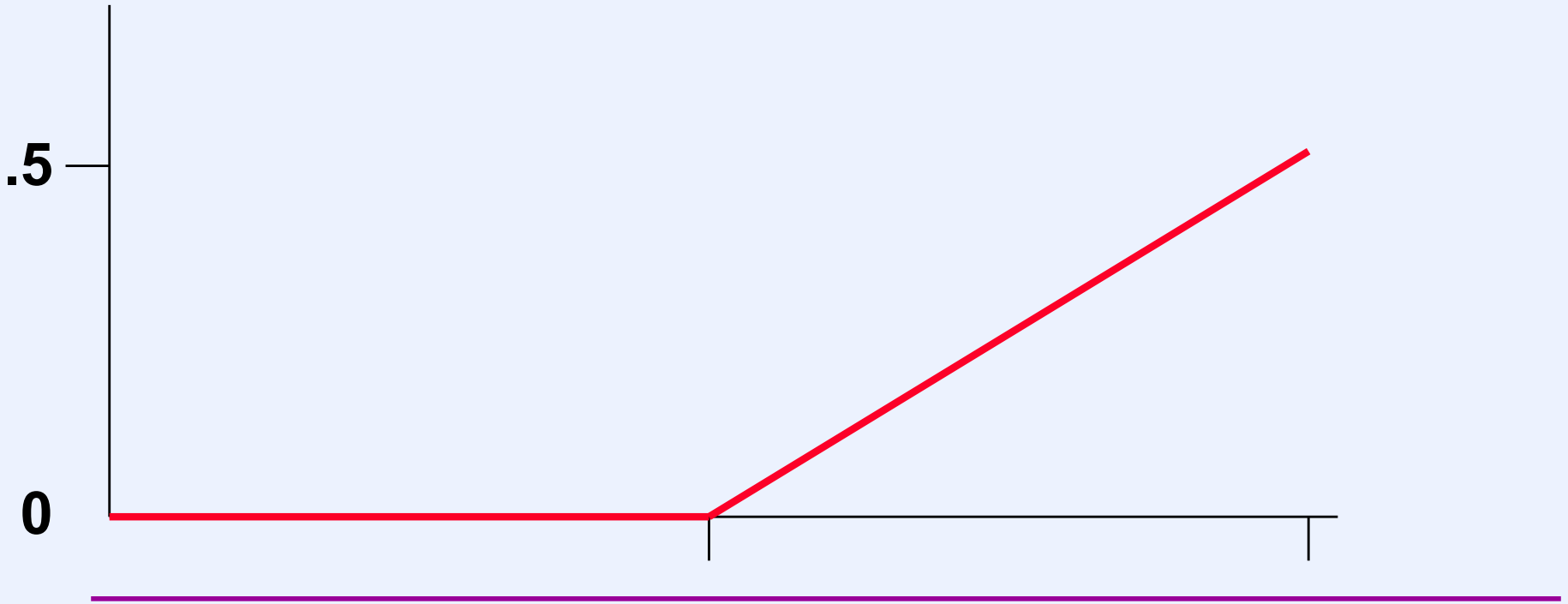
Scenario

- 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



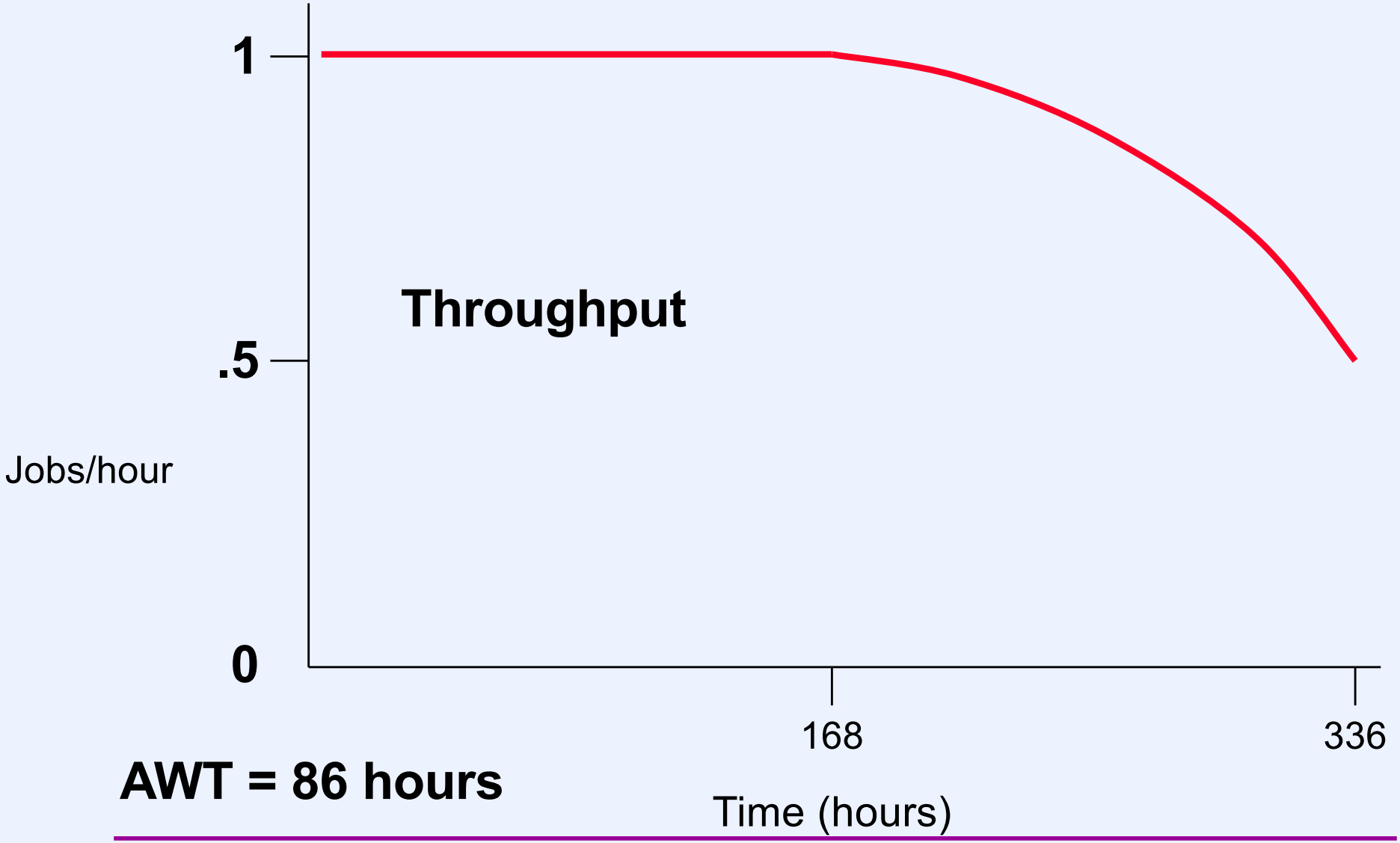
Average Wait Time

- 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*

SJF and Our Example



AWT = 86 hours

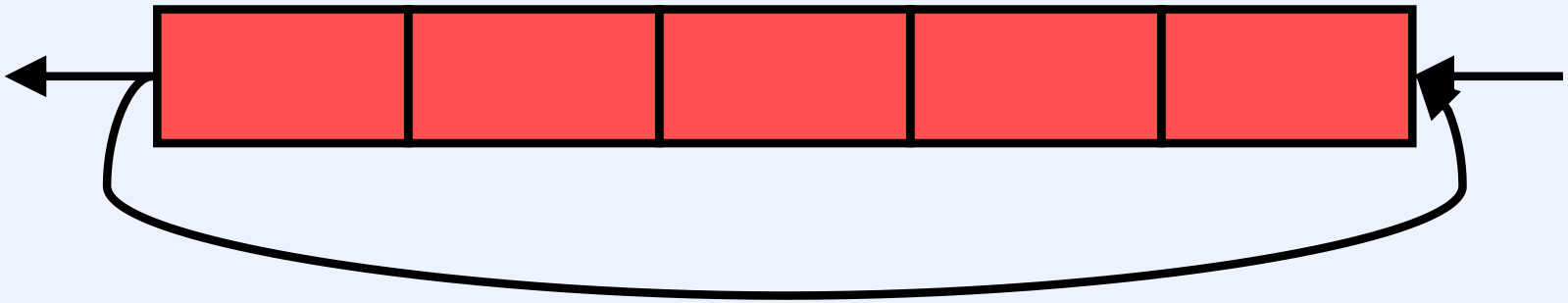
Preemption

- 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



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

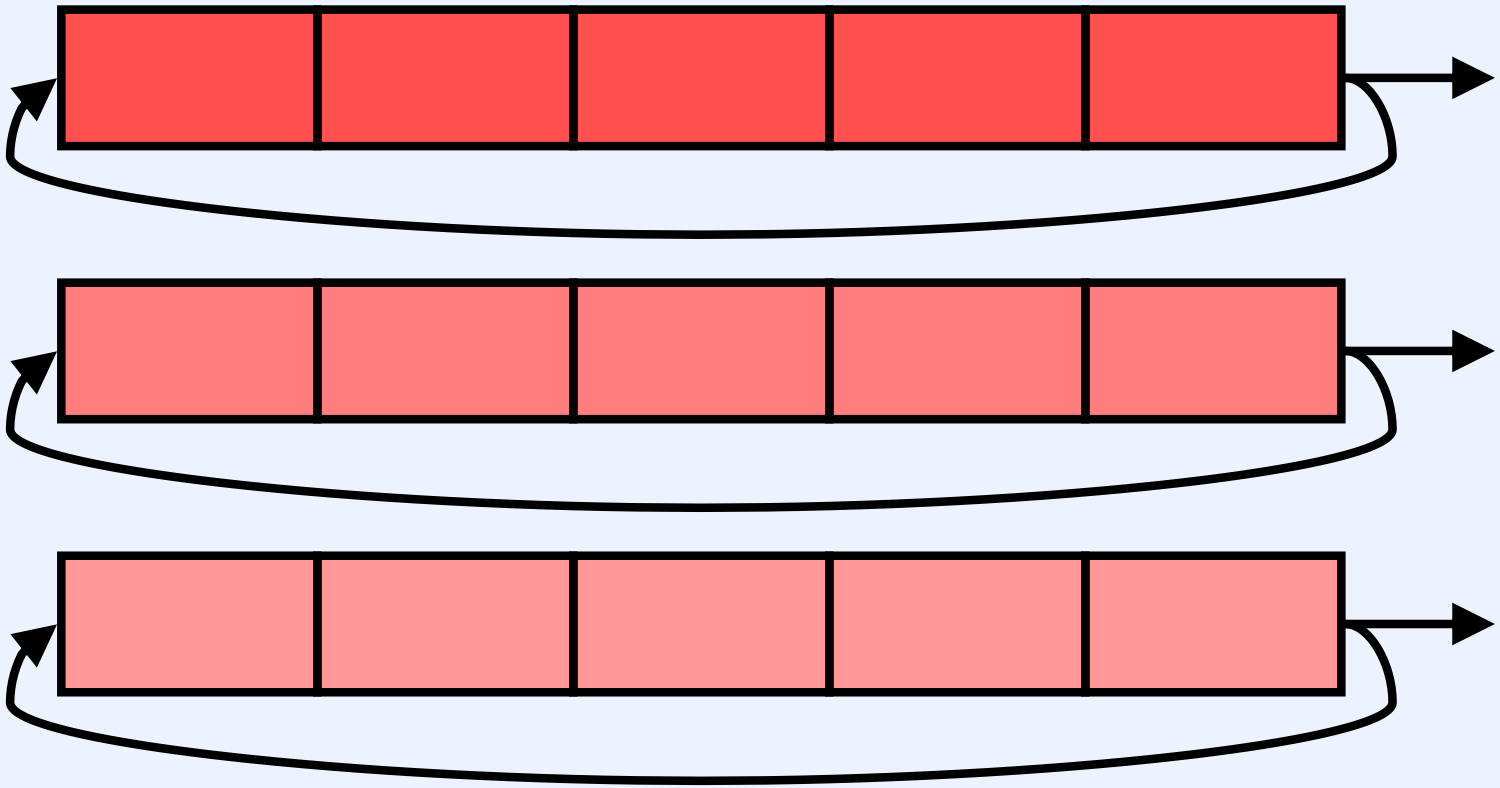
Round Robin + FIFO

- 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

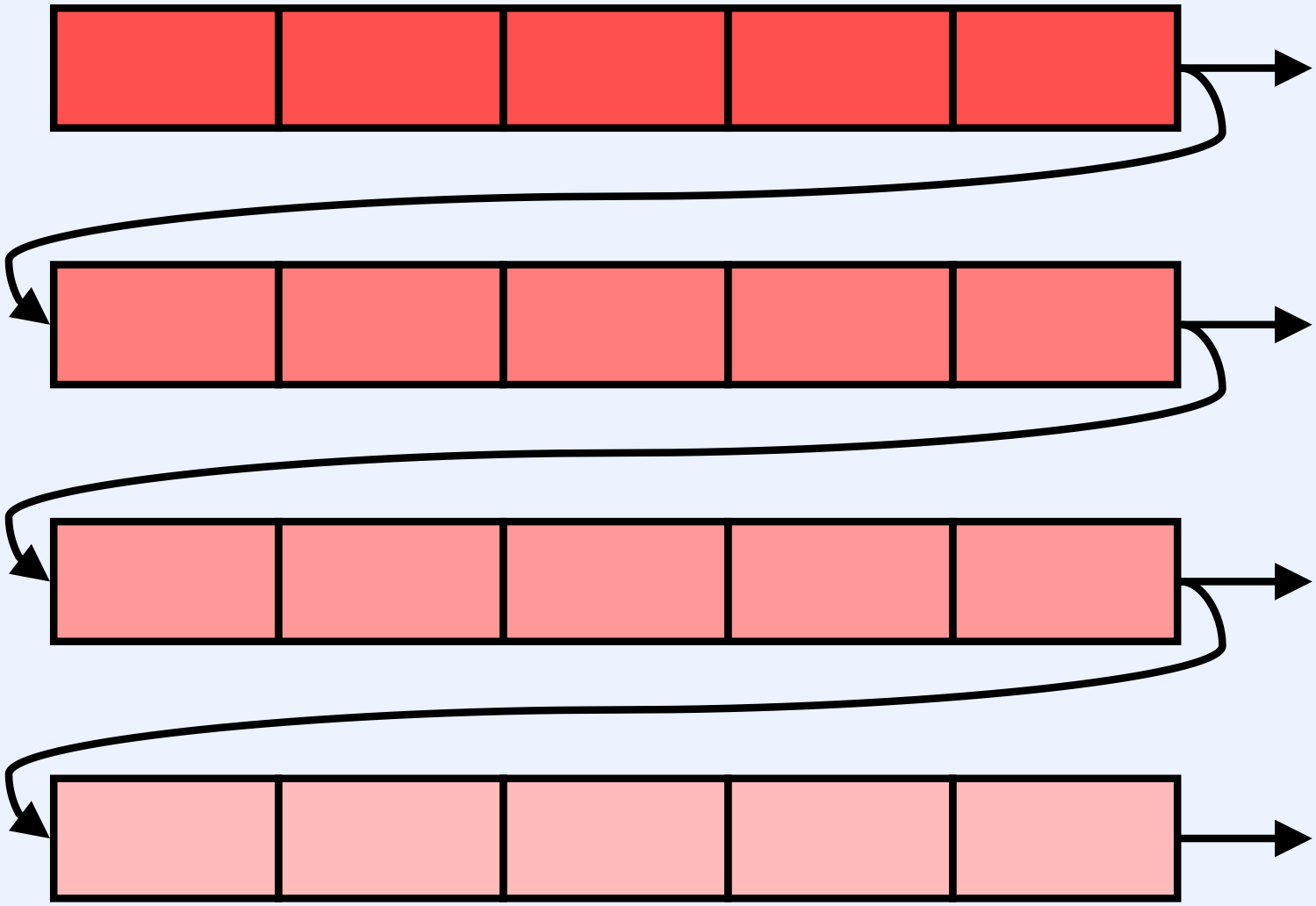
Interactive Systems

- Length of “jobs” not known
- Jobs don’t run to completion
  - run till they block for user input
- Would like to favor interactive jobs

Round Robin with Priority



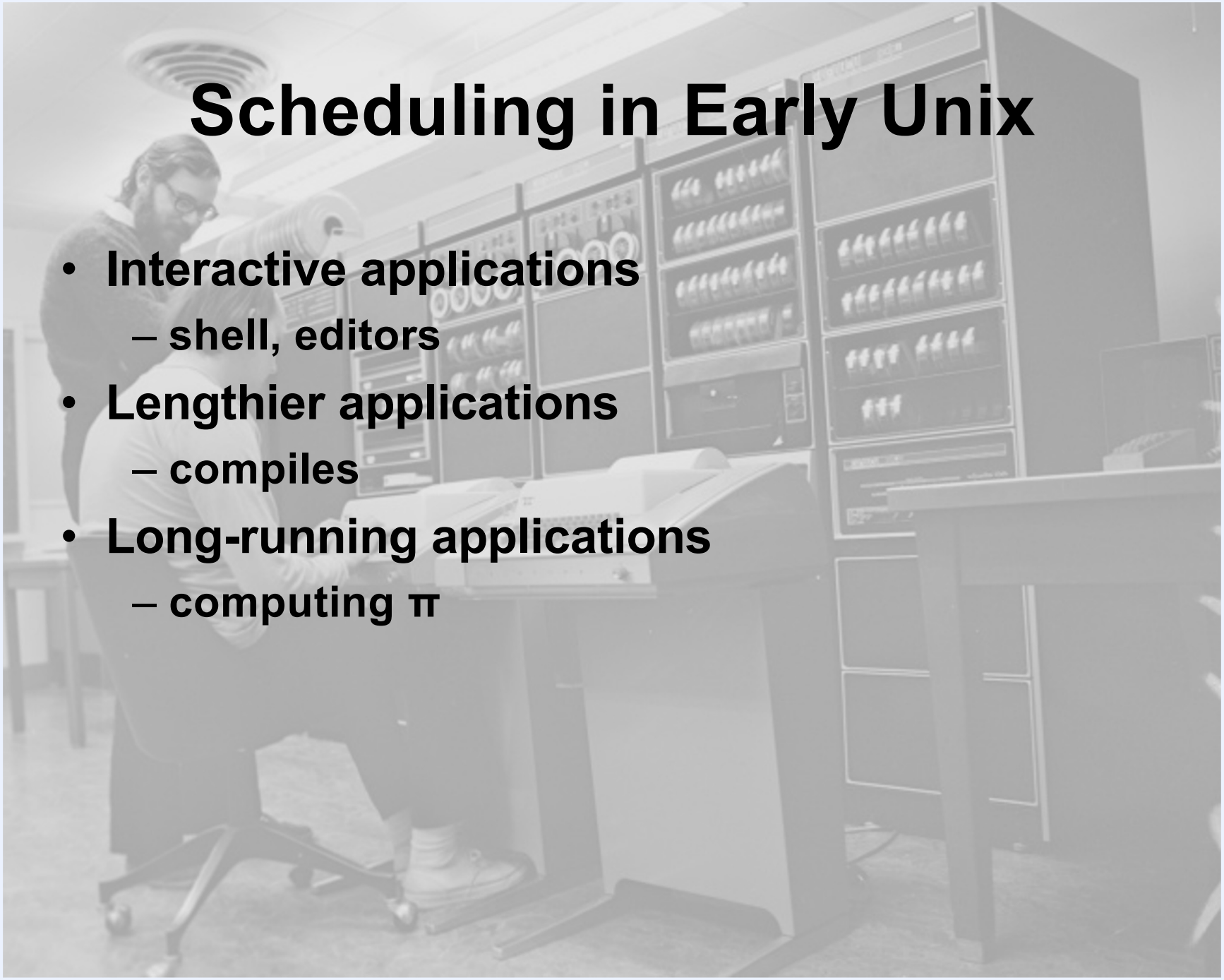
Multi-Level Feedback Queues



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

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

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

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

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

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

Issues

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

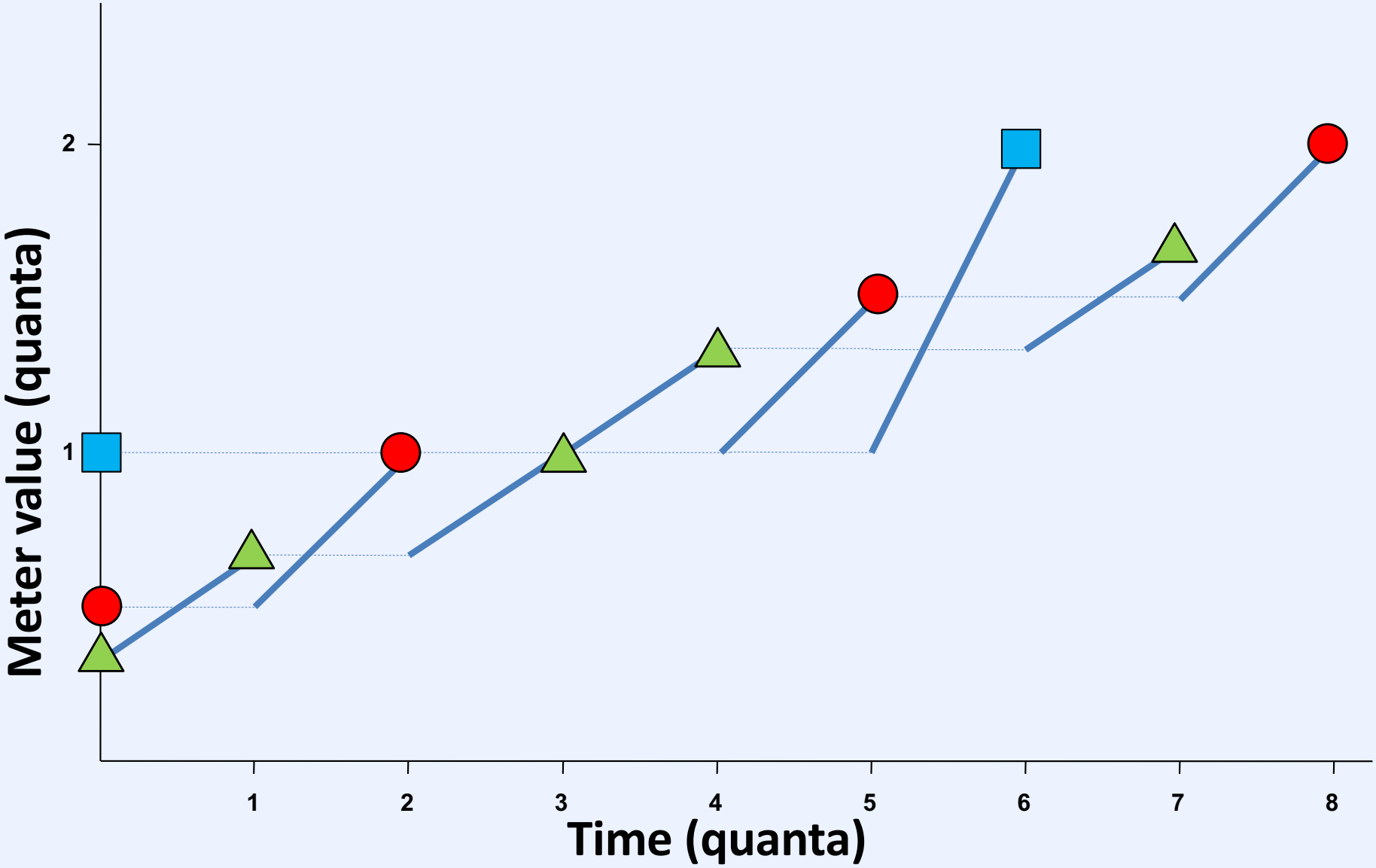
Details …

- 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

Example



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