Scheduling

Prof. K. Gopinath
IISc

Scheduling problematic!

- eg: Sol 2.2. Expt: Mix of jobs with
  - typing (text editor using X: interactive)
  - video (RT video player: captures data from digitizer, dithers to 8b & then displays thru X: continuous media)
  - compute (make appl: batch)
- 1st Expt: Make all jobs timesharing
  - input events (mouse/kbd) not accepted, video freezes, sh does not run!
  - batch class spawns and parents wait for children=> "I/O" intensive => repeated priority boosts for sleeping
  - window server identified as "CPU-intensive" and priority decreases
  - typing appl suffer as X does not run!

- 2nd Expt: assign RT to video:
  - input not accepted, video degrades badly
  - video active all the time: so TS tasks (shell, X) not run!
- 3rd Expt: assign X to RT:
  - mouse OK but batch hogs the CPU
- 4th Expt: assign X+video to RT:
  - typing and batch suffer, sh does not run
  - flushing dirty pages to disk, process swapping do not happen!
- 5th Expt: X+typing+video in RT with P(X)>P(typing)>P(video):
  - typing does not run as it needs STREAMS!

How to model?

- Scheduling CPUs
  - QoS for multimedia
- Scheduling I/Os
  - Linux2.6: elevator, deadline, anticipatory, complete fair queuing, noop
- Scheduling network packets
  - QoS for packets
- Scheduling groups of processes across dispersed nodes
  - Grid computing

Stochastic models

Operational models

Test 1. Writes-Starving-Reads

In the background, perform a streaming write, such as:

```
while true
do
        dd if=/dev/zero of=file bs=1M
done
```

Meanwhile, time how long a simple read of a 200MB file takes:

```
time cat 200mb-file > /dev/null
```

Test 2. Effects of High Read Latency

Start a streaming read in the background:

```
while true
do
        cat big-file > /dev/null
done
```

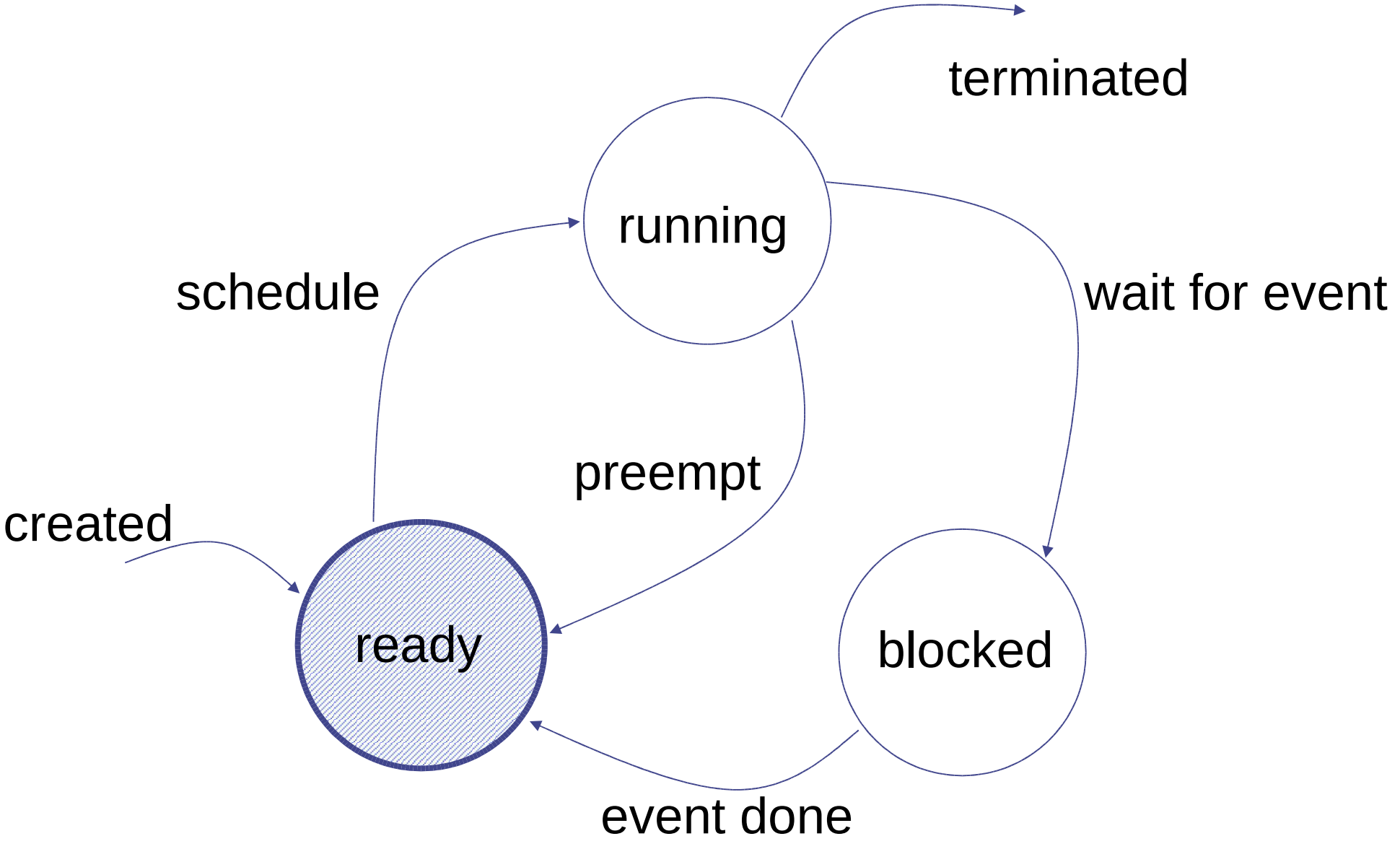
Meanwhile, measure how long it takes for a read of every file in the kernel source tree to complete:

```
time find . -type f -exec cat '{}' ';' > /dev/null
```

The Results

I/O Scheduler and Kernel	Test 1	Test 2
Linus Elevator on 2.4	45 secs	30 mins, 28 secs
Deadline I/O Scheduler on 2.6	40 secs	3 mins, 30 secs
Anticipatory I/O Scheduler on 2.6	4.6 secs	15 secs

Short-Term Scheduling



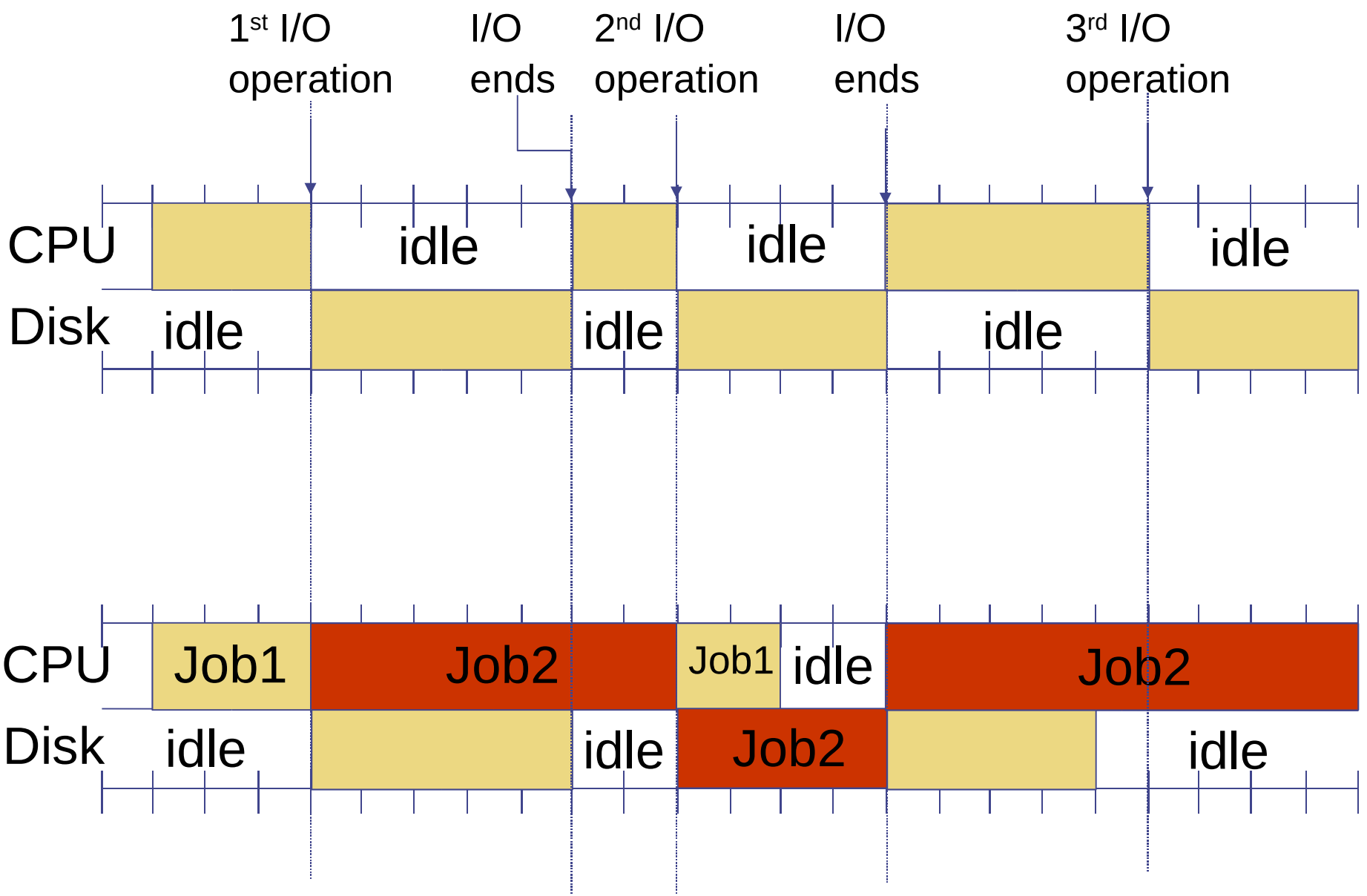
Short-Term Scheduling (STS)

- Process execution pattern consists of alternating CPU cycle and I/O wait
  - CPU burst – I/O burst – CPU burst – I/O burst…
- Processes ready for execution held in a ready (run) queue
- STS schedules process from the ready queue once CPU becomes idle

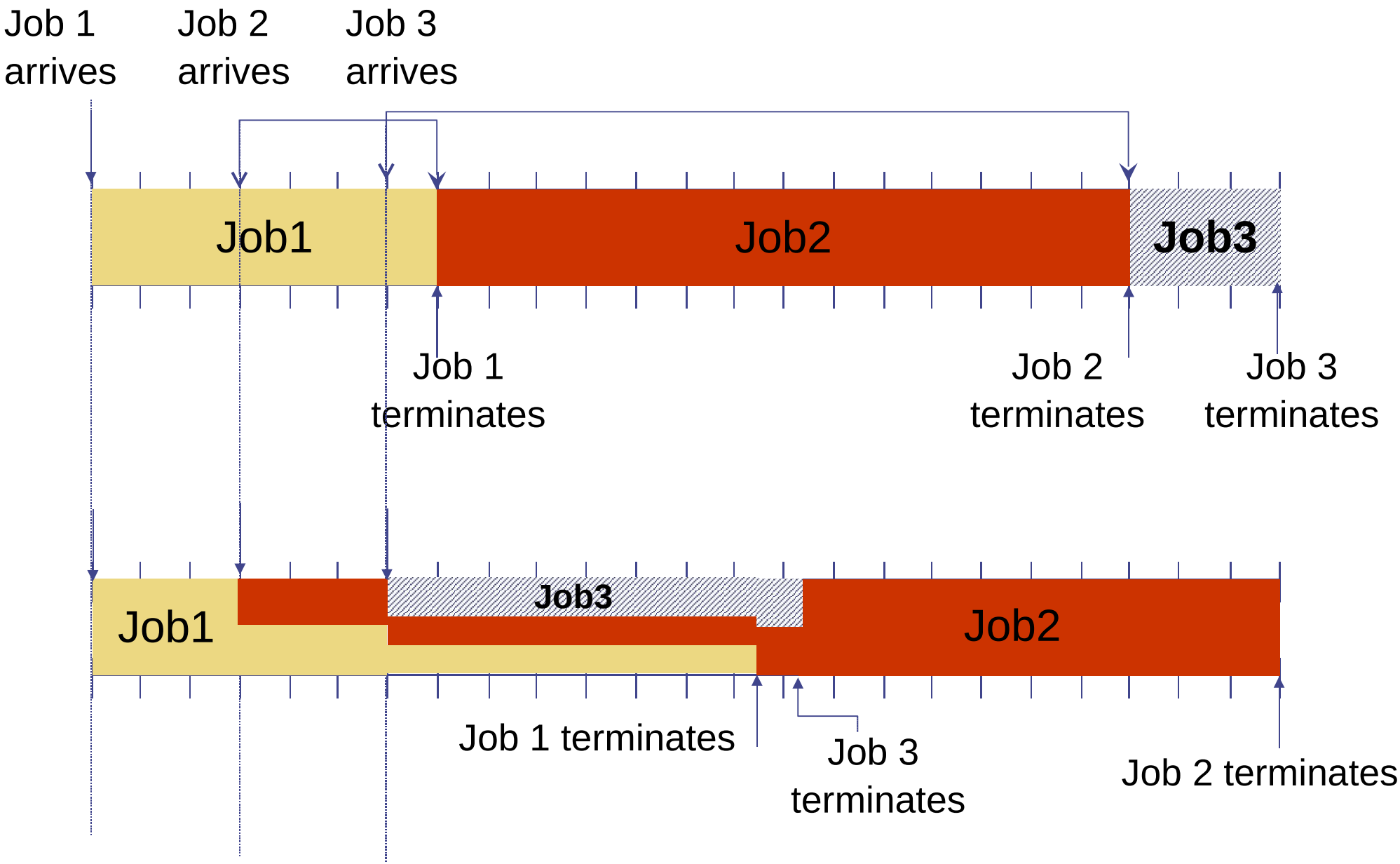
Other:

- Medium-Term: swap out
- Long-term: admission control

Utilization



Responsiveness



Scheduling Jobs

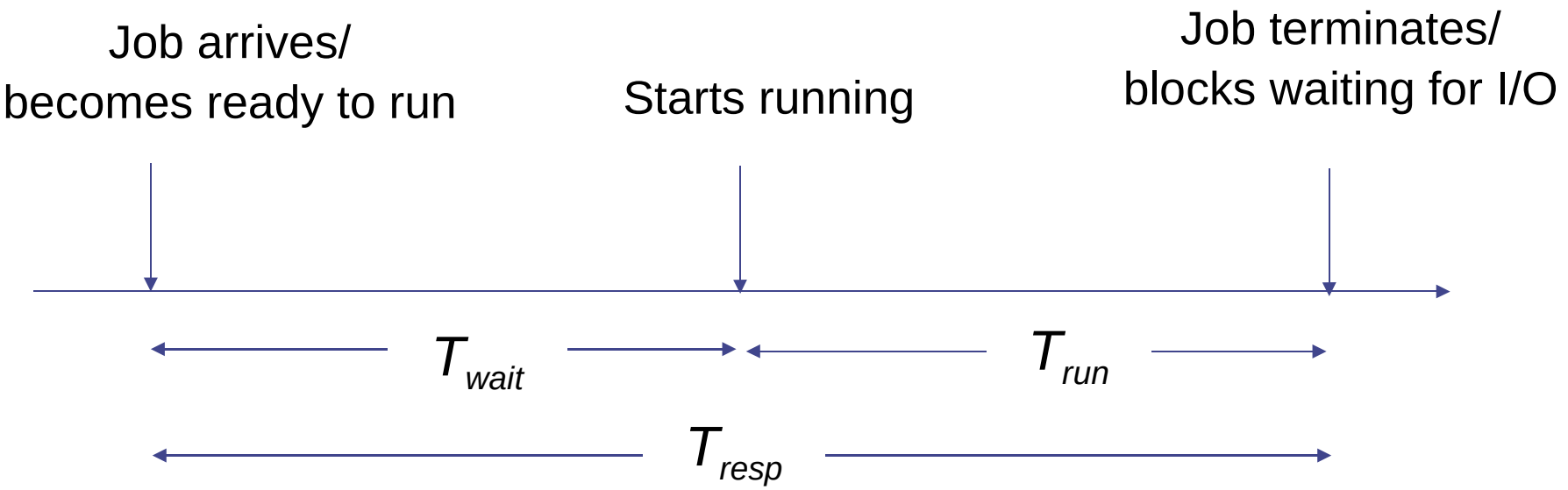
Would CPU sharing improve responsiveness if all jobs take the same time?

No. It makes it worse!

- For a given workload, the answer depends on the value of **coefficient of variation** (CV) of the distribution of job runtimes
  - CV=stand. dev. / mean
  - CV < 1 => CPU sharing does not help
  - CV > 1 => CPU sharing does help
- If all jobs are CPU bound (I/O bound), multiprogramming does not help to improve utilization
- A suitable job mix is created by a long-term scheduling
  - Jobs are classified on-line to be CPU (I/O) bound according to the job's history

Metrics: Response time

- Response time (turnaround time) is the average over the jobs' T_{resp}



$T_{resp} = T_{wait} + T_{run}$

Other Metrics

- Wait time: average of T_{wait}
  - This parameter is under the system control
- Response ratio or slowdown

  $slowdown = T_{resp} / T_{run}$
- Throughput, utilization depend on user imposed workload=>
  - Less useful

Note about running time (T_{run})

- Length of the CPU burst
  - When a process requests I/O it is still "running" in the system
  - But it is not a part of the STS workload
- STS view: I/O bound processes are short processes
  - Although text editor session may last hours!

Off-line vs. On-linescheduling

- Off-line algorithms
  - Get **all** the information about **all** the jobs to schedule as their input
  - Outputs the scheduling sequence
  - Preemption is never needed
- On-line algorithms
  - Jobs arrive at unpredictable times
  - Very little info is available in advance
  - Preemption compensates for lack of knowledge

Tradeoffs

- Efficiency: spend as much time in user processes as possible
- Fairness: avoid starvation, give each process a fair share
- Priority handling: allow more important processes better service
- Real-time constraints: a guaranteed level of service
- Hardware constraints: how much does it cost to switch processes?

Issues…

- What is the Application Profile? A program alternates between CPU usage and I/O.
  - Relevant question for scheduling: is a program compute-bound (mostly CPU usage) or I/O-bound (mostly I/O wait)?
- When scheduling occurs:
  - When a process is created
  - When a process terminates
  - When a process issues a blocking call (I/O, semaphores)
  - On a clock interrupt
  - On I/O interrupt (e.g., disk xfer done, mouse click)
  - System calls for IPC (e.g., up on semaphore, signal, etc.)
- Multi-level scheduling (e.g., 2-level in Unix)
  - Swapper decides which processes should reside in memory
  - Scheduler decides which ready process gets the CPU next

- Can preemption occur?
  - Preemptive schedulers can take control from a process at interrupt
  - Non-preemptive scheduler does not
- What are we trying to optimize?
  - CPU utilization: Fraction of time CPU is in use
  - Throughput: average# of jobs completed per time unit
  - Turnaround Time: average time between job submission and completion
  - Waiting Time: average amount of time a process is ready but waiting
  - Response Time: time until system responds to a cmd
  - Response Ratio: (Turnaround Time)/(Execution Time) -- long jobs should wait longer

- Different applications require optimizing different things
  - Batch systems (throughput, turnaround time)
  - Interactive system (response time, fairness, user expectation)
  - Real-time systems (meeting deadlines)
- Overhead of scheduling
  - Context switching expensive (minimize context switches)
  - Data structures & book-keeping used by scheduler
- What is being scheduled?
  - Basic abstraction: Jobs
  - Jobs might be processes, might be threads " processes in Unix, threads in Linux or Solaris

Real workloads

- Exp. Dist: CV=1; Heavy Tailed Dist: CV>1
- Dist. of job runtimes in real systems is heavy tailed
  - CV ranges from 3 to 70
- CPU sharing does improve responsiveness
  - CPU sharing is approximated by time slicing: interleaved execution

First-Come-First-Serve (FCFS)

- Schedules the jobs in the order in which they arrive
  - Off-line FCFS schedules in the order the jobs appear in the input
- Runs each job to completion
- Both on-line and off-line
- Simple, a base case for analysis
- Poor response time

Shortest Job First (SJF)

- Best response time
- Inherently off-line
  - All the jobs and their run-times must be available in advance

Using preemption

- On-line short-term scheduling algorithms
  - Adapting to changing conditions
    - e.g., new jobs arrive
  - Compensating for lack of knowledge
    - e.g., job run-time
- Periodic preemption keeps system in control
- Improves fairness
  - Gives I/O bound processes chance to run

Shortest Remaining Time first (SRTF)

- Job run-times ("CPU burst") are known or “predict”
- Job arrival times are not known
- When a new job arrives:

  if its run-time is shorter than the remaining time of the currently executing job:

  preempt the currently executing job and schedule the newly arrived job

  else continue the current job and insert the new job into a sorted queue
- When a job terminates, select the job at the queue head for execution

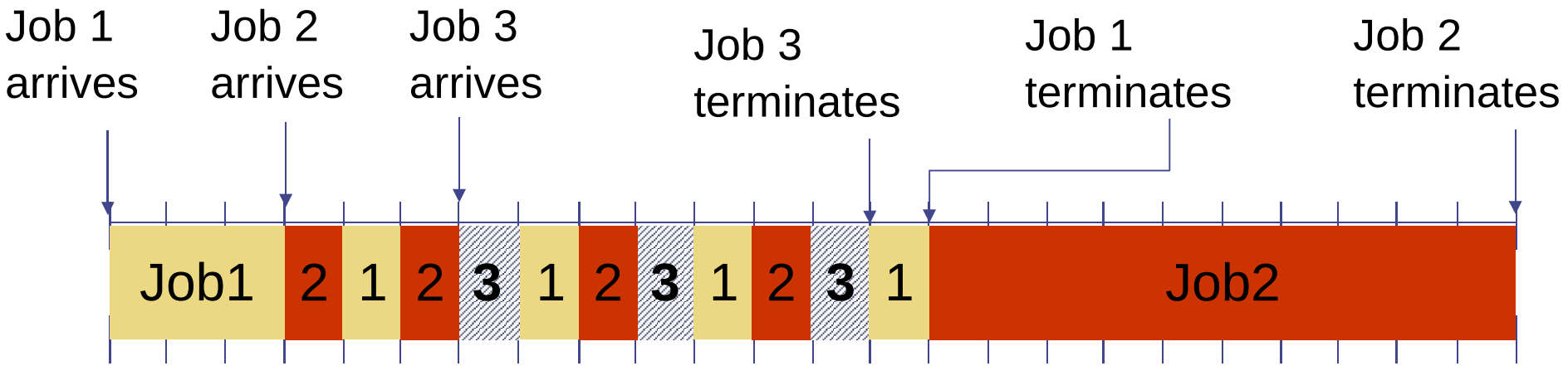
Non-preemptive version also (STF)!

Optimal response time: STF among non-preemptive and SRTF among preemptive

Unfair to long jobs and requires knowledge of future

Round Robin (RR)

- Both job arrival times and job run-times are not known
- Run each job cyclically for a short time quantum
  - Approximates CPU sharing
- Choose quantum so that each cpu burst finishes most of the time



Priority Scheduling

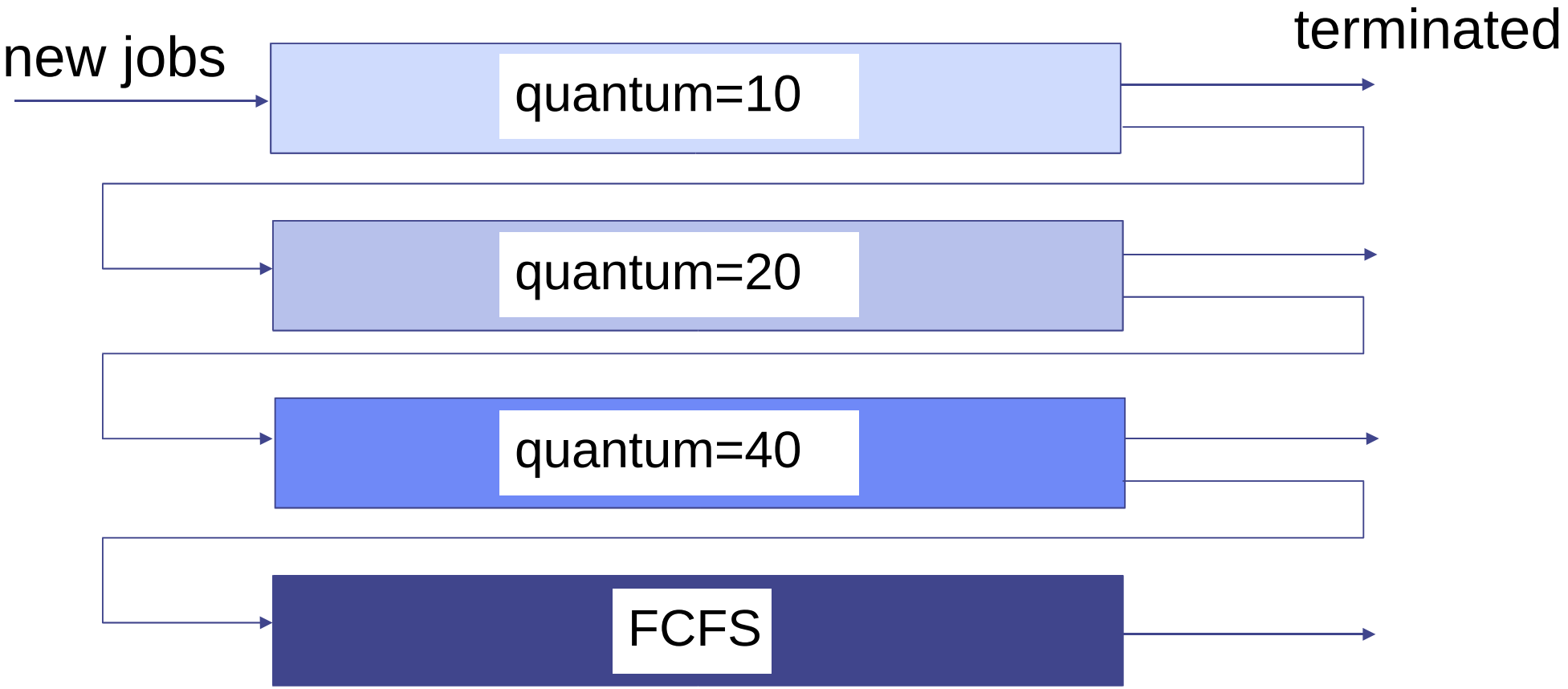
- RR is oblivious to the process past
  - I/O bound processes are treated equally with the CPU bound processes
- Solution: prioritize processes according to their past CPU usage

$$E_{n+1} = \alpha T_n + (1-\alpha)E_n, 0 \le \alpha \le 1$$

$$\alpha = \frac{1}{2} : E_{n+1} = \frac{1}{2}T_n + \frac{1}{4}T_{n-1} + \frac{1}{8}T_{n-2} + \frac{1}{16}T_{n-3} + \ldots$$

- T_n is the duration of the n-th CPU burst
- E_{n+1} is the estimate of the next CPU burst

Multilevel feedback queues



Multilevel feedback queues

- Priorities are implicit in this scheme
- Very flexible
- Starvation is possible: Short jobs keep arriving => long jobs get starved
- Solutions:
  - Let it be
  - Aging

Priority scheduling in UNIX

- Multilevel feedback queues
  - The same quantum at each queue
  - A queue per priority
- Priority is based on past CPU usage

  pri=cpu_use+base+nice
- cpu_use is dynamically adjusted
  - Incremented each clock interrupt: 100 sec^{-1}
  - Halved for **all** processes: 1 sec^{-1}

Problem with Unix: on overload, priority of all jobs increase as little CPU for each: interactive jobs suffer

No Guaranteed Scheduling for RT (soft/hard) sys

Fair Share scheduling algorithms

- Given a set of processes with associated weights, a fair share scheduler should allocate CPU to each process in proportion to its respective weight
  - Achieving pre-defined goals
    - Administrative considerations
      - Paying for machine usage, importance of project, personal importance, etc.
    - Quality-of-service, soft real-time: Video, audio
- Weighted Round Robin
  - Shares are not uniformly spread in time
- Lottery scheduling:
  - Each process gets a number of lottery tickets proportional to its CPU allocation
  - The scheduler picks a ticket at random and gives it to the winning client
  - Only statistically fair, high complexity

Linux 2.4

- The policy field of process descriptor (struct task struct, include/ linux/sched.h) contains:
  - SCHED FIFO: First-In First-Out real-time
  - SCHED RR: Round-Robin real-time
  - SCHED OTHER: non real-time
- The scheduler divides the CPU time in epochs
- When starting a new epoch, the scheduler assigns a new quantum to every process
- When a process exhausts its time quantum, it cannot run anymore until epoch terminates
- The epoch terminates when all runnable processes have exhausted their time quantum

- Linux's tq ranges between 10 ms and 300 ms
- The Base Time Quantum is the default time quantum assigned to a new process
  - On all architectures, it is roughly equal to 50 ms
  - The nice() system call can raise or lower process base tq
  - On a Intel-based arch, base tq is: 6 – nice/4 ticks
    - where 1 ticks is about 10 ms and -20<=nice<=19
- The counter field of process descriptor contains # of ticks left to process before its time quantum expires
  - A periodic timer interrupt decrements current->counter once every tick
  - When current->counter becomes 0, the scheduler is invoked
  - When counter of all runnable processes is 0, a new epoch starts

- When starting a new epoch, the scheduler (schedule(), kernel/sched.c) updates the time quantum of all processes:
  - for_each_task(p)

    p->counter = (p->counter / 2) + (6 - p->nice/4);
  - If a process exhausted its time quantum in the previous epoch, it gets a fresh base time quantum
  - A suspended process gets a larger time quantum than before (half of the number of ticks left plus a base time quantum): the “I/O premium”
  - similarly, actual code gives bonus for preserving cache+TLB state

Probs

- The scheduler scans the whole list of runnable processes every time it must perform a process switching
- Starting a new epoch is expensive
- I/O-bound processes are not boosted when the number of runnable processes is high (any epoch is quite long)
- No distinction between interactive processes and batch I/O bound processes

Linux 2.6

- Runs in constant time
- Explicitly recognizes processes as being I/O-bound or CPU-bound
- Any CPU has its own runqueue of runnable processes
- Runnable processes migrate from a runqueue to another when the runqueue lengths are unbalanced

- Any runqueue consists of several round-robin lists including processes having the same priority
- At any timer tick, each CPU decrements the number of tick lefts to the current process before the time quantum expires
- The scheduler is invoked whenever the process has exhausted its time quantum
- The scheduler always selects the first process in the highest-priority list of the runqueue

- The process priority does not depend on the number of ticks left in the time quantum
- If a process goes to sleep, it is rewarded by increasing its priority
- Any process whose priority is higher than a given threshold is recognized as I/O bound
- If a CPU-bound process has exhausted its time quantum, it is inserted in a expired list, and it is never executed again until the epoch terminates
- If a I/O-bound process has exhausted its time quantum, it receives a fresh time quantum and it is inserted in the last position of the list associated with its priority

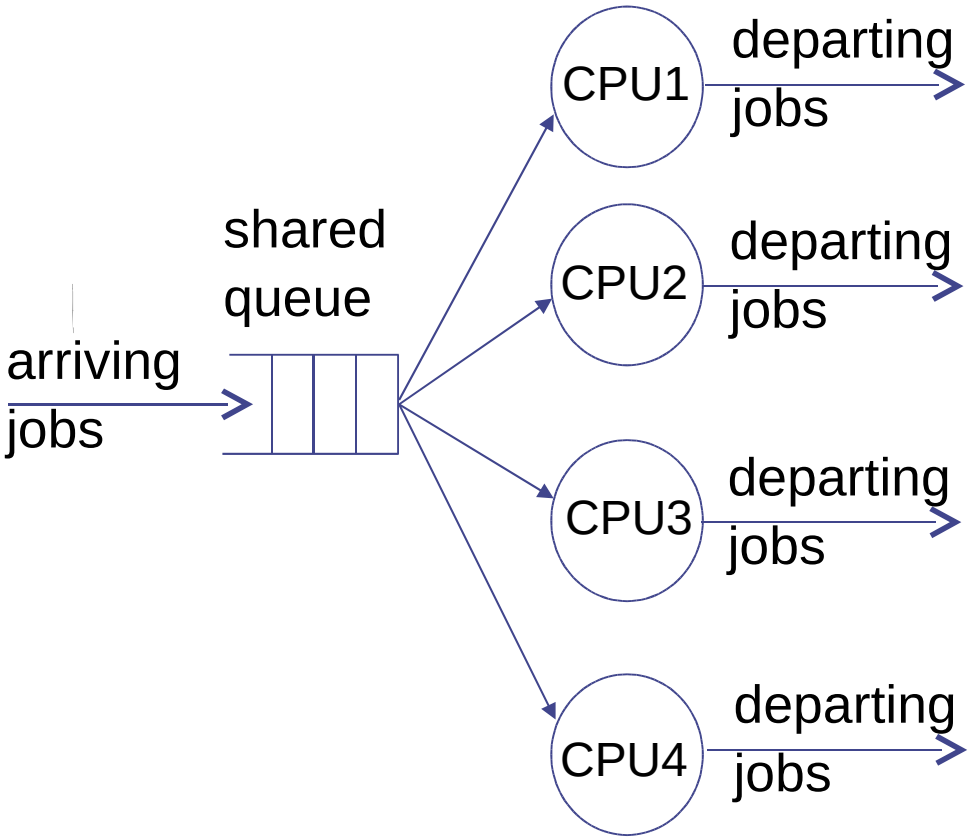
Multiprocessor Scheduling

- Homogeneous vs. heterogeneous
- Homogeneity allows for load sharing
  - Separate ready queue for each processor or common ready queue?
- Scheduling
  - Symmetric or Master/slave
- Assume Markov model: M/M/k
  - In general: M/M/k/B/K/SD: arrival/departure process exponential; k servers, B max system capacity, K population size, SD service discipline (FIFO, LIFO, random, priority, general)
  - M/M/1: mean # in system j = util/(1-util) $=\rho/(1-\rho)$
    ρ= mean arrival rate /mean service rate = λ/μ
    - response time (f arrival to exit)= $j/\lambda = \rho/\lambda(1-\rho)$ Little's law

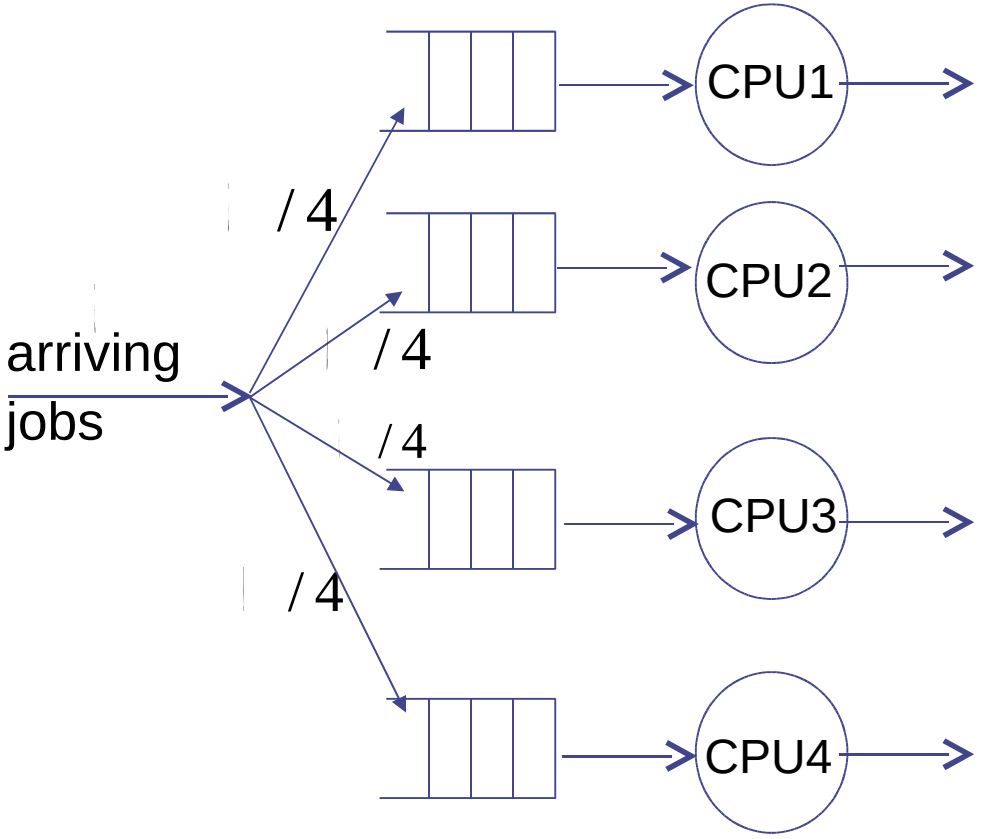
Poisson Arrivals

- Assumes that in a small interval δ
  - # of arrivals: $\lambda*\delta$
  - Prob of more than 1 arrival in δ: negligible
  - Arrivals in nonoverlapping intervals statistically indep
- Expected arrival time = $1/\lambda$
- Probability of an arrival in time t = $1- \exp(-\lambda t)$
  - Probability of no arrivals in time t = $\exp(-\lambda t)$
- Probability of k arrivals in time t= $\exp(-\lambda t)(\lambda t)^k/k!$
- Similarly, Poisson departures

Which is better?

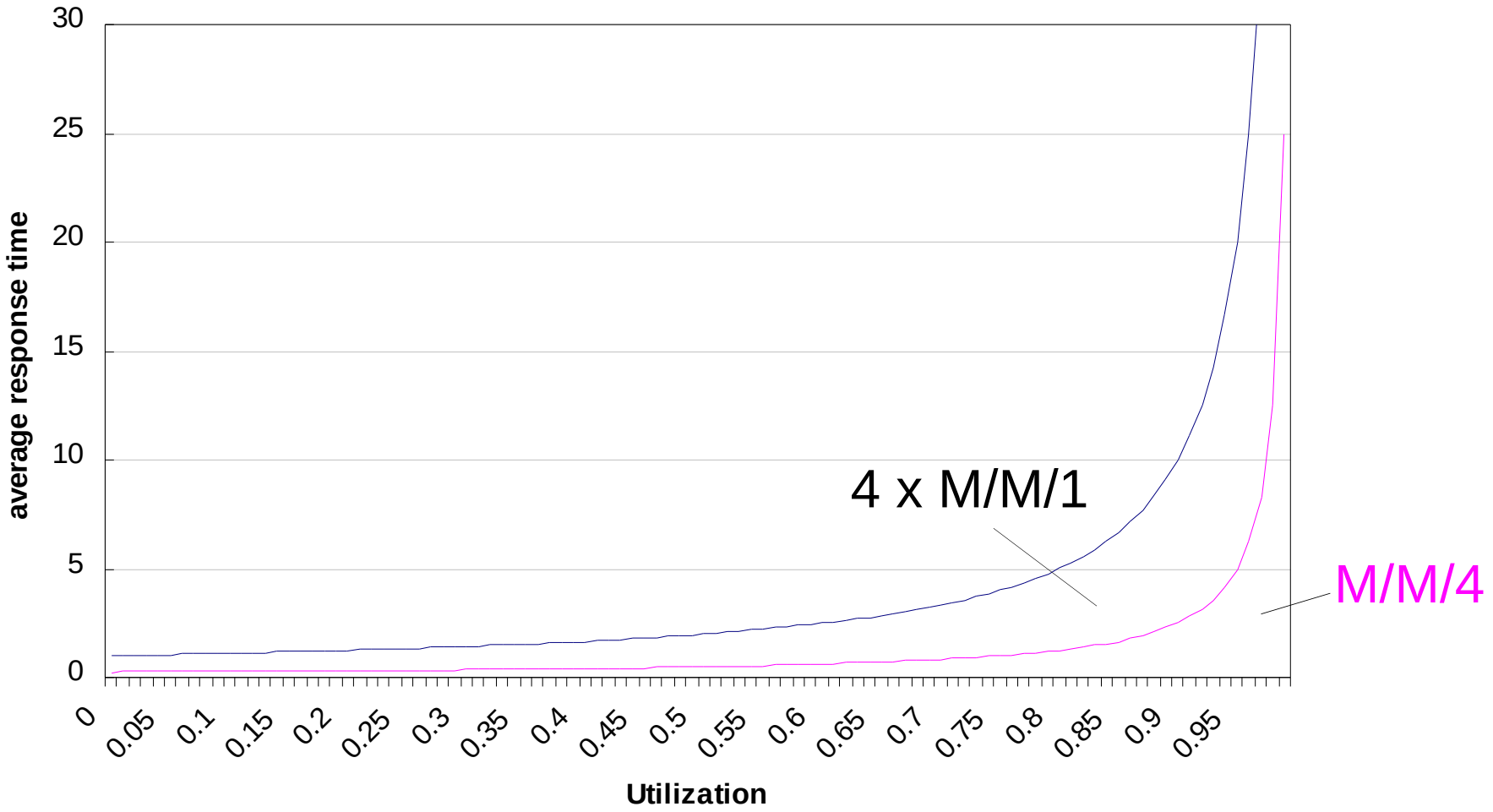


M/M/4



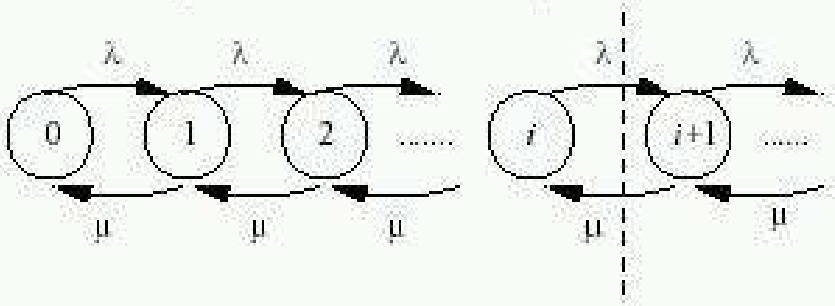
4 x M/M/1

M/M/4!


30
25
20
15
10
5
0
average response time
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0.55
0.6
0.65
0.7
0.75
0.8
0.85
0.9
0.95
Utilization
4 x M/M/1
M/M/4

$M/M/1$ Queueing System

- arrivals - Poisson process with rate λ
- service times - exponentially distributed with rate μ (mean $1/\mu$)
- $\rho \equiv \lambda/\mu$
- model as BD process; state $N(t)$ equals number of customers in system at time t
- $\lambda < \mu$ for steady state solution; let $N = \lim_{t\to\infty} N(t)$



$\pi_i = P(N = i)$ satisfies

$$\mu\pi_i = \lambda\pi_{i-1}, \quad i = 1, \ldots$$

or

$$\pi_i = \rho\pi_{i-1}, \quad i = 1, \ldots$$

Solution is

$$\pi_i = \rho^i \pi_0, \quad i = 0, 1, \ldots$$

Coupled with the relation $\sum_{i=0}^{\infty} \pi_i = 1$ yields

$$\boxed{\pi_i = (1-\rho)\rho^i, \; i = 0, 1, \ldots}$$

with mean $E[N]$

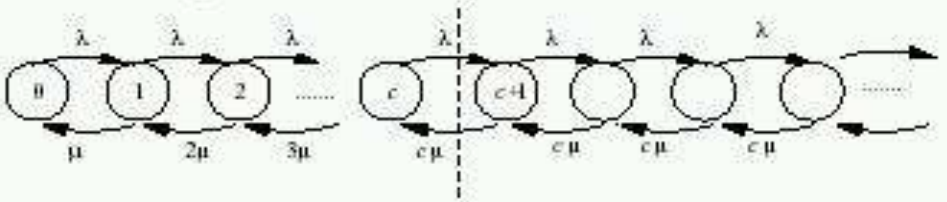
$$\boxed{E[N] = \rho/(1-\rho)}$$

$$E[W] = E[N]/\mu = \frac{1}{\mu} \times \frac{\rho}{1-\rho}$$

$$\begin{aligned} E[T] = E[W] + 1/\mu &= \frac{1/\mu}{1-\rho} \\ &= \frac{1}{\mu - \lambda} \end{aligned}$$

$M/M/c$ System

- arrivals - Poisson process with rate λ
- service times - exponentially distributed with rate μ (mean $1/\mu$)
- c identical servers
- model as BD process; state $N(t)$ equals number of customers in system at time t



Balance equations:

$$\begin{aligned} \pi_{i-1}\lambda &= \pi_i i\mu, \quad i \leq c, \\ \pi_{i-1}\lambda &= \pi_i c\mu, \quad i > c \end{aligned}$$

Solution to balance equations

$$\pi_i = \begin{cases} \frac{\lambda^i}{i!\mu^i}\pi_0, & 0 \leq i \leq c, \\ \frac{\lambda^i}{c!c^{i-c}\mu^i}\pi_0, & c < i \end{cases}$$

$M/M/c$ Metrics

- prob. a customer has to wait

$$\begin{aligned} P(Wait) &= \sum_{n=c}^{\infty} \pi_n, \\ &= \pi_0 \frac{\rho^c}{c!(1-\rho/c)} \\ &= \pi_c/(1-\rho/c), \\ &= \frac{\rho^c/(c-1)!}{(c-\rho)[\sum_{n=0}^{c-1} \rho^n/n! + \rho^c/((c-1)!(c-\rho))]} \end{aligned}$$

  this last expression is called Erlang's C Formula, $C(c,\rho)$

- waiting time statistics

$$P(W \leq t) = 1 - C(c,\rho)e^{-(c\mu-\lambda)t}, \quad t \geq 0$$

$$E[W] = \frac{C(c,\rho)/\mu}{c-\rho}$$

M/M/1 vs M/M/2

- Response time for separate Qs (M/M/1) = $1/\mu(1-\rho)$ with $\rho=(\lambda/2)/\mu$

  $= 2/(2\mu-\lambda) = (4\mu+2\lambda)/(4\mu^2-\lambda^2)$

- Response time for combined Qs (M/M/2):
  - $E[N] = 2\rho/(1-\rho^2)$ where $\rho = \lambda/2\mu$
  - $E[R] = E[N]/\lambda$ (Little's Law)

    $= 4\mu/(4\mu^2-\lambda^2)$
  - Less than that for separate Qs

Video on Thin clients

- 1.2Mbps for video and 300kb for audio
  - I frames: 100kb (1 in 12)
  - P frames: 50kb (3 in 12)
  - B frames: 20kb (8 in 12)
  - IBBPBBPBBPBBI
  - 35kb per frame avg
- 1.5Mbps:
  - ~30 ms disk latency (5400 rpm: rot latency: 5.5 ms, seek time 8 ms; 5 ms xfer) ~50ms netw?
- Double buffering: decoding from MPEG into fb0 and xfer from fb1 into netw

Layered view of the problem

- hw card support: hw scaling &YUV acceleration
  - X11 acceleration support
- driver support?
- kernel support: firm timers (+soft timers), preemptable kernel, adaptive send-buffer tuning, proportion-period or real-rate scheduler
- TCP: nodelay option; send buffer >64KB typ.
- lib support?
- X11: how much does xlib buffer reqs to Xserver? How long does it wait?
  - XAA, DGA, XVideo extension?DirectGraphicsAccess?
- KDE/GNOME/fvwm/...
- VNC: deferUpdate (40ms default)