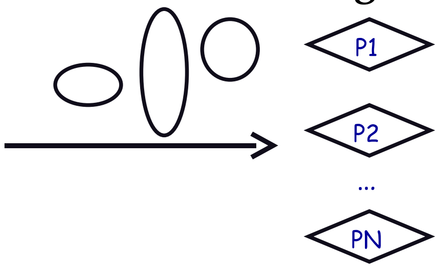
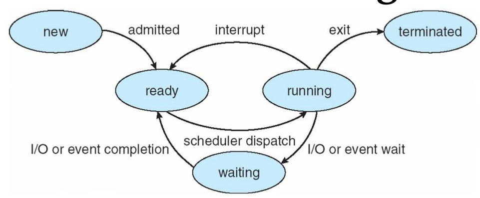
CPU Scheduling



- **The scheduling problem:**
  - Have K jobs ready to run
  - Have $N \geq 1$ CPUs
  - Which jobs to assign to which CPU(s)
- **When do we make decision?**

CPU Scheduling



- **Scheduling decisions may take place when a process:**
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Exits
- **Non-preemptive schedules use 1 & 4 only**
- **Preemptive schedulers run at all four points**

Scheduling criteria

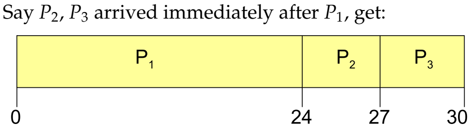
- **Why do we care?**
  - What goals should we have for a scheduling algorithm?

Scheduling criteria

- **Why do we care?**
  - What goals should we have for a scheduling algorithm?
- ***Throughput* – # of procs that complete per unit time**
  - Higher is better
- ***Turnaround time* – time for each proc to complete**
  - Lower is better
- ***Response time* – time from request to first response (e.g., key press to character echo, not launch to exit)**
  - Lower is better
- **Above criteria are affected by secondary criteria**
  - *CPU utilization* – keep the CPU as busy as possible
  - *Waiting time* – time each proc waits in ready queue

Example: FCFS Scheduling

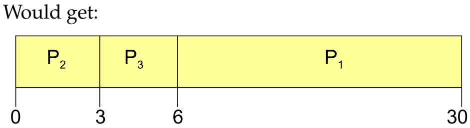
- **Run jobs in order that they arrive**
  - Called "*First-come first-served*" (FCFS)
  - E.g.., Say P_1 needs 24 sec, while P_2 and P_3 need 3.
  - Say P_2, P_3 arrived immediately after P_1, get:



- **Dirt simple to implement—how good is it?**
- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround Time:** $P_1 : 24$, $P_2 : 27$, $P_3 : 30$
  - Average TT: $(24 + 27 + 30)/3 = 27$
- **Can we do better?**

FCFS continued

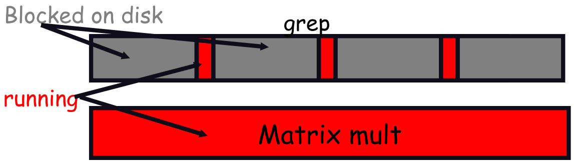
- **Suppose we scheduled** P_2, P_3, **then** P_1
  - Would get:



- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround time:** $P_1 : 30$, $P_2 : 3$, $P_3 : 6$
  - Average TT: $(30 + 3 + 6)/3 = 13$ – much less than 27
- **Lesson: scheduling algorithm can reduce TT**
  - Minimize waiting time to minimize TT
- **What about throughput?**

I/O devices just special CPUs

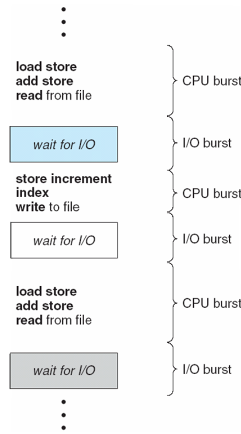
- **An I/O device is like a special purpose CPU**
  - "special purpose" = disk drive can only run a disk job, tape drive a tape job, …
- **Implication: 1-CPU system with n I/O devices is like an $n + 1$-CPU multiprocessor**
  - Result: all I/O devices + CPU busy $\Longrightarrow$ n+1 fold speedup!



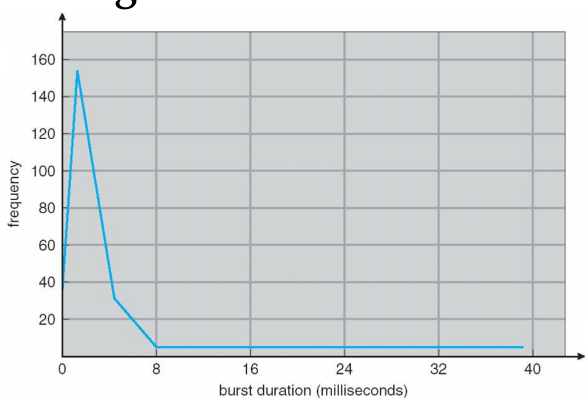
  - Overlap them just right? throughput will be doubled

Bursts of computation & I/O

- **Jobs contain I/O and computation**
  - Bursts of computation
  - Then must wait for I/O
- **To Maximize throughput**
  - Must maximize CPU utilization
  - Also maximize I/O device utilization
- **How to do?**
  - Overlap I/O & computation from multiple jobs
  - Means *response time* very important for I/O-intensive jobs: I/O device will be idle until job gets small amount of CPU to issue next I/O request



Histogram of CPU-burst times



- **What does this mean for FCFS?**

FCFS Convoy effect

- **CPU bound jobs will hold CPU until exit or I/O (but I/O rare for CPU-bound thread)**
  - long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- **Example: one CPU-bound job, many I/O bound**
  - CPU bound runs (I/O devices idle)
  - CPU bound blocks
  - I/O bound job(s) run, quickly block on I/O
  - CPU bound runs again
  - I/O completes
  - CPU bound still runs while I/O devices idle (continues?)
- **Simple hack: run process whose I/O completed?**
  - What is a potential problem?

SJF Scheduling

- ***Shortest-job first* (SJF) attempts to minimize TT**
- **Two schemes:**
  - *nonpreemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Know as the *Shortest-Remaining-Time-First* or SRTF)
- **What does SJF optimize?**

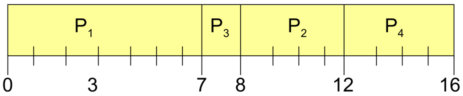
SJF Scheduling

- ***Shortest-job first* (SJF) attempts to minimize TT**
- **Two schemes:**
  - *nonpreemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Know as the *Shortest-Remaining-Time-First* or SRTF)
- **What does SJF optimize?**
  - gives minimum average *waiting time* for a given set of processes

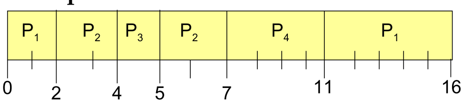
Examples

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- **Non-preemptive**



- **Preemptive**

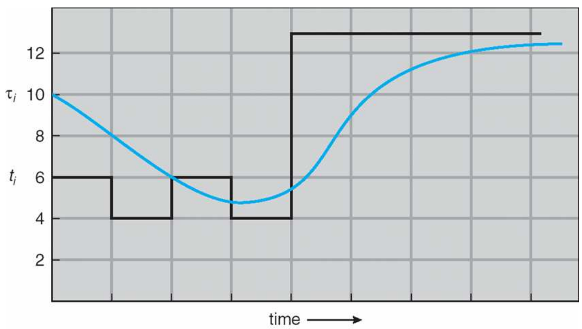


- **Drawbacks?**

SJF limitations

- **Doesn't always minimize average turnaround time**
  - Only minimizes waiting time
  - Example where turnaround time might be suboptimal?
- **Can lead to unfairness or starvation**
- **In practice, can't actually predict the future**
- **But can estimate CPU burst length based on past**
  - Exponentially weighted average a good idea
  - t_n actual length of proc's n^{th} CPU burst
  - τ_{n+1} estimated length of proc's $n+1^{\text{st}}$
  - Choose parameter α where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

Exp. weighted average example



CPU burst (t_i)		6	4	6	4	13	13	13 …
"guess" (τ_i)	10	8	6	6	5	9	11	12 …

Round robin (RR) scheduling

- **Solution to fairness and starvation**
  - Preempt job after some time slice or *quantum*
  - When preempted, move to back of FIFO queue
  - (Most systems do some flavor of this)
- **Advantages:**
  - Fair allocation of CPU across jobs
  - Low average waiting time when job lengths vary
  - Good for responsiveness if small number of jobs
- **Disadvantages?**

RR disadvantages

- **Varying sized jobs are good…**
- **but what about same-sized jobs?**
- **Assume 2 jobs of time=100 each:**

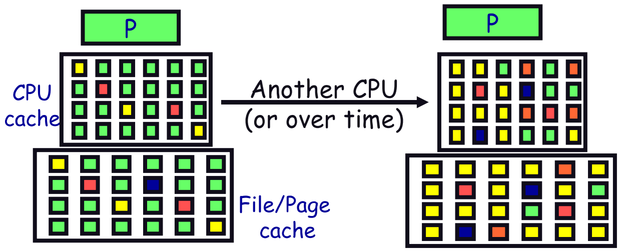
  - What is average completion time?
  - How does that compare to FCFS?

Context switch costs

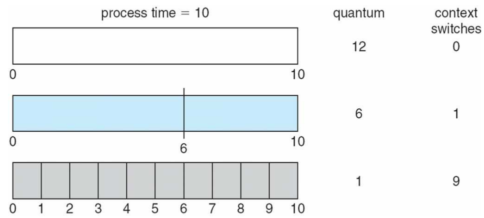
- **What is the cost of a context switch?**

Context switch costs

- What is the cost of a context switch?
- Brute CPU time cost in kernel
  - Save and restore resisters, etc.
  - Switch address spaces (expensive instructions)
- Indirect costs: cache, buffer cache, & TLB misses

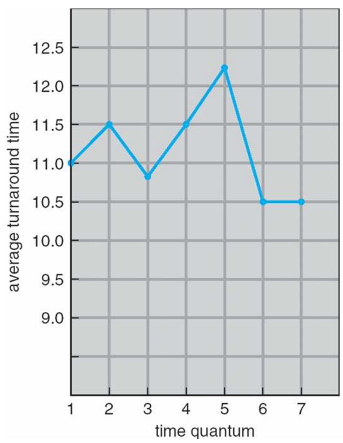


Time quantum



- How to pick quantum?
  - Want much larger than context switch cost
  - But not so large system reverts to FCFS
- Typical values: 10–100 msec

Turnaround time vs. quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Two-level scheduling

- Switching to swapped out process very expensive
  - Swapped out process has most pages on disk
  - Will have to fault them all in while running
  - One disk access costs 10ms. On 1GHz machine, 10ms = 10 million cycles!
- Context-switch-cost aware scheduling
  - Run in core subset for "a while"
  - Then move some between disk and memory
  - How to pick subset? Hot to define "a while"?

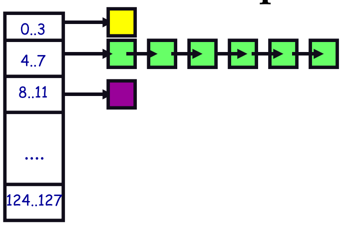
Priority scheduling

- A priority number (integer) is associated with each process
  - E.g., smaller priority number means higher priority
- Give CPU to the process with highest priority
  - Can be done preemptively or non-preemptively
- Note SJF is a priority scheduling where priority is the predicted next CPU burst time
- Starvation – low priority processes may never execute
- Solution?

Priority scheduling

- A priority number (integer) is associated with each process
  - E.g., smaller priority number means higher priority
- Give CPU to the process with highest priority
  - Can be done preemptively or non-preemptively
- Note SJF is a priority scheduling where priority is the predicted next CPU burst time
- Starvation – low priority processes may never execute
- Solution?
  - Aging - increase a process's priority as it waits

Multilevel feeedback queues (BSD)



- **Every runnable proc. on one of 32 run queues**
  - Kernel runs proc. on highest-priority non-empty queue
  - Round-robins among processes on same queue
- **Process priorities dynamically computed**
  - Processes moved between queues to reflect priority changes
  - If a proc. gets higher priority than running proc., run it
- **Idea: Favor interactive jobs that use less CPU**

Process priority

- `p_nice` **– user-settable weighting factor**
- `p_estcpu` **– per-process estimated CPU usage**
  - Incremented whenever timer interrupt found proc. running
  - Decayed every second while process runnable

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right) \texttt{p\_estcpu} + \texttt{p\_nice}$$

- **Run queue determined by** `p_usrpri`/4

$$\texttt{p\_usrpri} \leftarrow 50 + \left(\frac{\texttt{p\_estcpu}}{4}\right) + 2 \cdot \texttt{p\_nice}$$

(value clipped if over 127)

Sleeping process increases priority

- `p_estcpu` **not updated while asleep**
  - Instead `p_slptime` keeps count of sleep time
- **When process becomes runnable**

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right)^{\texttt{p\_slptime}} \times \texttt{p\_estcpu}$$

  - Approximates decay ignoring nice and past loads

Limitations of BSD scheduler

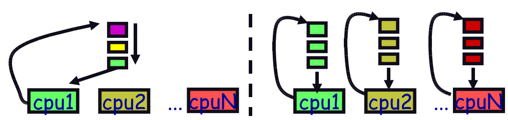
- **Hard to have isolation / prevent interference**
  - Priorities are absolute
- **Can't transfer priority (e.g., to server on RPC)**
- **No flexible control**
  - E.g., In monte carlo simulations, error is 1/sqrt(N) after N trials
  - Want to get quick estimate from new computation
  - Leave a bunch running for a while to get more accurate results
- **Multimedia applications**
  - Often fall back to degraded quality levels depending on resources
  - Want to control quality of different streams

Real-time scheduling

- **Two categories:**
  - *Soft real time*—miss deadline and CD will sound funny
  - *Hard real time*—miss deadline and plane will crash
- **System must handle periodic and aperiodic events**
  - E.g., procs A, B, C must be scheduled every 100, 200, 500 msec, require 50, 30, 100 msec respectively
  - *Schedulable* if $\sum \frac{CPU}{\text{period}} \leq 1$ (not counting switch time)
- **Variety of scheduling strategies**
  - E.g., first deadline first (works if schedulable)

Multiprocessor scheduling issues

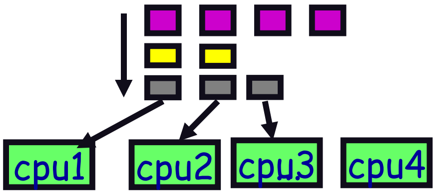
- **Must decide more than which process to run**
  - Must decide on which CPU to run it
- **Moving between CPUs has costs**
  - More cache misses, depending on arch more TLB misses too
- ***Affinity scheduling*—try to keep threads on same CPU**



  - But also prevent load imbalances
  - Do *cost-benefit* analysis when deciding to migrate

Multiprocessor scheduling (cont)

- **Want related processes scheduled together**
  - Good if threads access same resources (e.g., cached files)
  - Even more important if threads communicate often, otherwise must context switch to communicate
- ***Gang scheduling*—schedule all CPUs synchronously**
  - With synchronized quanta, easier to schedule related processes/threads together



Thread scheduling

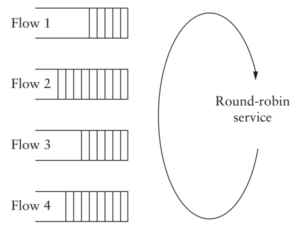
- **With thread library, have two scheduling decisions:**
  - *Local Scheduling* – Threads library decides which user thread to put onto an available kernel thread
  - *Global Scheduling* – Kernel decides which kernel thread to run next
- **Can expose to the user**
  - E.g., `pthread_attr_setscope` allows two choices
  - `PTHREAD_SCOPE_SYSTEM` – thread scheduled like a process (effectively one kernel thread bound to user thread – Will return ENOTSUP in user-level pthreads implementation)
  - `PTHREAD_SCOPE_PROCESS` – thread scheduled within the current process (may have multiple user threads multiplexed onto kernel threads)

Thread dependencies

- **Priority inversion e.g., T_1 at high priority, T_2 at low**
  - T_2 acquires lock L.
  - Scene 1: T_1 tries to acquire L, fails, spins. T_2 never gets to run.
  - Scene 2: T_1 tries to acquire L, fails, blocks. T_3 enters system at medium priority. T_2 never gets to run.
- **Scheduling = deciding who should make progress**
  - Obvious: a thread's importance should increase with the importance of those that depend on it.
  - Naïve priority schemes violate this
- **"Priority donation"**
  - Thread's priority scales w. priority of dependent threads

Fair Queuing (FQ)

- **Digression: packet scheduling problem**
  - Which network packet to send next over a link?
  - Problem inspired some algorithms we will see next time
- **For ideal fairness, would send one bit from each flow**
  - In weighted fair queuing (WFQ), more bits from some flows



- **Complication: must send whole packets**

FQ Algorithm

- **Suppose clock ticks each time a bit is transmitted**
- **Let P_i denote the length of packet i**
- **Let S_i denote the time when start to transmit packet i**
- **Let F_i denote the time when finish transmitting packet i**
- $F_i = S_i + P_i$
- **When does router start transmitting packet i?**
  - If arrived before router finished packet $i-1$ from this flow, then immediately after last bit of $i-1$ (F_{i-1})
  - If no current packets for this flow, then start transmitting when arrives (call this A_i)
- **Thus:** $F_i = \max(F_{i-1}, A_i) + P_i$

FQ Algorithm (cont)

- **For multiple flows**
  - Calculate F_i for each packet that arrives on each flow
  - Treat all F_is as timestamps
  - Next packet to transmit is one with lowest timestamp
- **Not perfect: can't preempt current packet**
- **Example:**