Chapter 2

Processes and Threads

2.5 Scheduling

Scheduling

- When more than one process is ready to run, but only one CPU is available, a choice is to make
- Part of OS that does it is scheduler
- The algorithm it uses is scheduling algorithm

Scheduling

- Efficiency is needed as process switching is costly:
  - Switch from user mode to kernel mode
  - State of current process need to be saved
  - Memory map may be saved
  - A process is selected
  - MMU to be reloaded with memory map of new process
  - New process is started

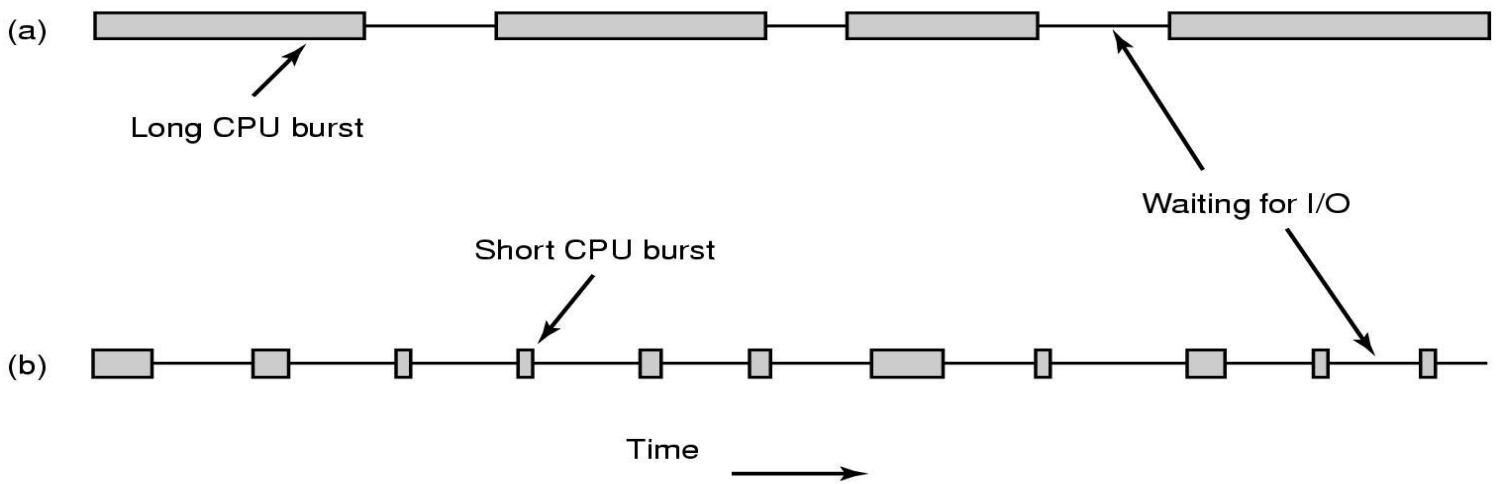
Importance of Scheduling

- Good scheduling algorithms can make a big difference
  - Resource utilization
  - Perceived performance & User satisfaction
  - Meeting other system goals (e.g., important tasks being taken care of immediately)

Process Behavior

- Processes usually alternate bursts of *computing* with *I/O requests.*
- *CPU burst: the amount of time the process uses the processor before it is no longer ready*
- I/O in this sense is when a process enters the blocked state waiting for an external device to complete its work

Process: Compute- and I/O-bound



- a CPU-bound process (data encryption/decryption, multimedia encoding)
  - Spend most of the time computing
  - Long CPU bursts, infrequent I/O waits
- an I/O bound process (shell waiting for user commands)
  - Spend most of the time waiting for I/O
  - Short CPU bursts, frequent I/O waits
- Key factor is the **length of CPU burst** not the length of the I/O burst

Process: Compute- and I/O-bound

- As the CPUs get faster, processes tend to get more I/O bound: WHY?
- If a I/O bound process is ready, it should get a chance quickly.
  - Increase resource utilization

When to Schedule: Issues(1)

- When a new process is created: which one to run? Parent or child? Both are Ready.
- When a process exits: One of the ready processes should be run
  - If none is ready a system-supplied idle process is normally run
- When a process blocks: Another process has to be selected to run
  - Sometimes the reason for blocking may play a role in the choice (A is waiting for B and B is in CR)
  - Blocking may occur for:
    - I/O
    - Semaphore

When to Schedule: Issues(2)

- When an I/O interrupt occurs: a scheduling decision may be made. In case of an interrupt of an I/O device having completed its work, some blocked process may now be ready.
- If a clock provides periodic interrupt: A scheduling decision can be made at each (or k-th) clock interrupt.

Preemptive & Non-preemptive

Classification of Scheduling Algorithm depending on dealing with clock interrupt

- Non-preemptive: Picks a process to run and lets it run until it blocks or voluntarily releases the CPU. In effect at each clock interrupt, no scheduling is done.
- Preemptive: Picks a process and lets it run for a maximum of some fixed time. If still running, it is suspended and another is picked.
- Preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler.

Different Systems, Different Focuses

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Batch Systems

- Users submit their job to the batch system
- Batch system starts user job when appropriate
- User gets notification that job is done. No interaction in between
- No users impatiently waiting at terminals for a quick response to a short request
- Used in business world such as Profit calculation at banks, claims processing at insurance companies…

Batch Systems

- Common performance metrics
  - Throughput: number of jobs completed per hour
  - Turnaround time: average time between the submission and completion of a job
- Maximizing Throughput may not necessarily minimize Turnaround time

Batch Systems

Algorithms used:

- non-preemptive algorithms
- preemptive algorithms with long time periods are often acceptable
    - Reduces process switches and improves performance

Representative algorithms:

2. First Come First Serve (FCFS)
3. Shortest Job First
4. Shortest Remaining Time First.

First Come First Serve (FCFS)

- Process that requests the CPU FIRST is allocated the CPU FIRST.
- Also called FIFO
- non-preemptive
- Used in Batch Systems
- Real life analogy?
  - Fast food restaurant
- Implementation
  - FIFO queues
  - A new process enters the tail of the queue
  - The schedule selects from the head of the queue.

FCFS Example

Process	Duration	Order	Arrival Time
P1	24	1	0
P2	3	2	0
P3	4	3	0

The final schedule:

P1 turnaround: 24
P2 turnaround: 27
P3 turnaround: 31

The average turnaround:
(24+27+31)/3 = 27.33

FCFS Example 2

Process	Duration	Order	Arrival Time
P1	24	1	0
P2	3	2	2
P3	4	3	4

The final schedule:

P1 turnaround: 24
P2 turnaround: 25
P3 turnaround: 27

The average turnaround:
(24+25+27)/3 = 25.33

Advantage

- Easy to understand and implement
- Fair

Problems with FCFS

- Non-preemptive
- Non optimal turnaround
- Cannot utilize resources in parallel:
  - Assume 1 process CPU bounded and many I/O bound processes
  - result: Convoy effect, low CPU and I/O Device utilization
  - Why?

Convoy effect

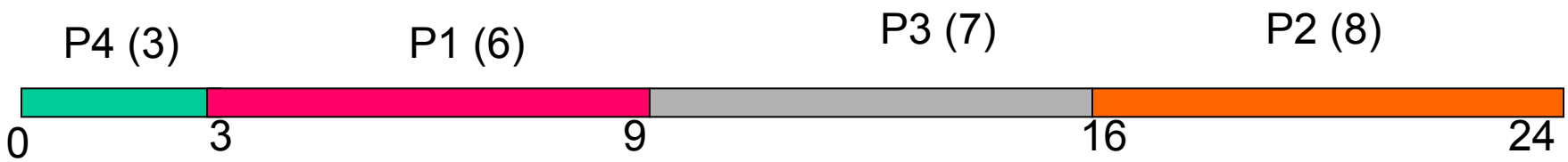
- The convoy effect occurs when a long-running process blocks others from getting CPU time, creating a “traffic jam” of waiting processes — reducing system efficiency.
- When the CBP uses the CPU, The IBPs are waiting behind it, leaving the I/O devices idle.
- When the CBP finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O,
- then the cycle repeats itself when the CBP gets back to the ready queue.

Shortest Job First (SJF)

- Non-preemptive scheduling algorithm in batch systems
- Schedule the job with the shortest run time first
- Requirement: the run time needs to be known in advance
- SJF is optimal in terms of turnaround, if all jobs arrive at same time

SJF: Example

Process	Duration	Order	Arrival Time
P1	6	1	0
P2	8	2	0
P3	7	3	0
P4	3	4	0



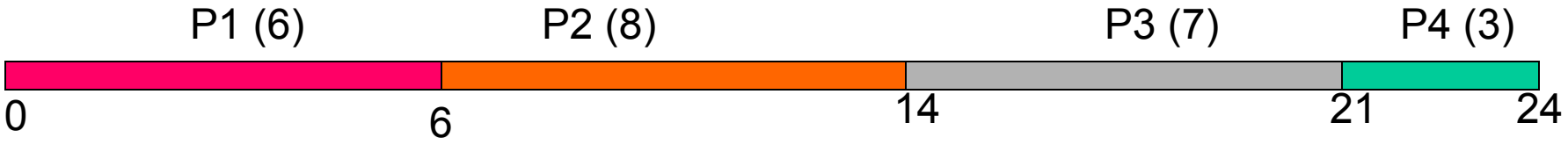
Do it yourself

P4 turnaround: 3
P1 turnaround: 9
P3 turnaround: 16
P2 turnaround: 24

Total execution time: 24
The average turnaround:
(3+9+16+24)/4 = 13

Comparing to FCFS

Process	Duration	Order	Arrival Time
P1	6	1	0
P2	8	2	0
P3	7	3	0
P4	3	4	0



P1 turnaround: 6
P2 turnaround: 14
P3 turnaround: 21
P4 turnaround: 24

The total time is the same.
The average turnaround:
(6+14+21+24)/4 = 16.25
(comparing to 13)

SJF is not always optimal

- SJF optimal only if all jobs have arrived at scheduling time

Process	Duration	Order	Arrival Time
P1	10	1	0
P2	2	2	2

P1 turnaround: 10
P2 turnaround: 10

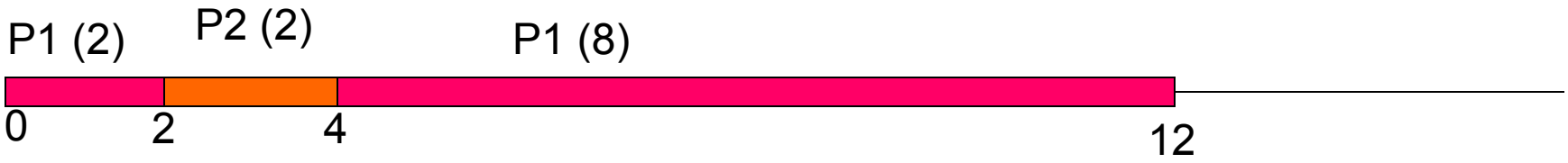
The average turnaround (AWT):
(10+10)/2 = 10

Preemptive SJF

- Also called Shortest Remaining Time Next
  - Schedule the job with the shortest remaining time required to complete
  - When new job arrives, compare its total time with the remaining time of the running job
  - If the new job needs less time the current job is suspended and the new job started
- Requirement: the run time needs to be known in advance

Preemptive SJF: Same Example

Process	Duration	Order	Arrival Time
P1	10	1	0
P2	2	2	2



P1 turnaround: 12
P2 turnaround: 2

The average turnaround:
(2+12)/2 = 7

Problem with Preemptive SJF?

- Starvation
  - In some condition, a job is waiting for ever
  - Example: Preemptive SJF
    - Process A with run time of 1 hour arrives at time 0
    - But every 1 minute, a short process with run time of 1 minute arrives
    - Result of Preemptive SJF: A never gets to run
- What's the difference between starvation and a dead lock?

Interactive System

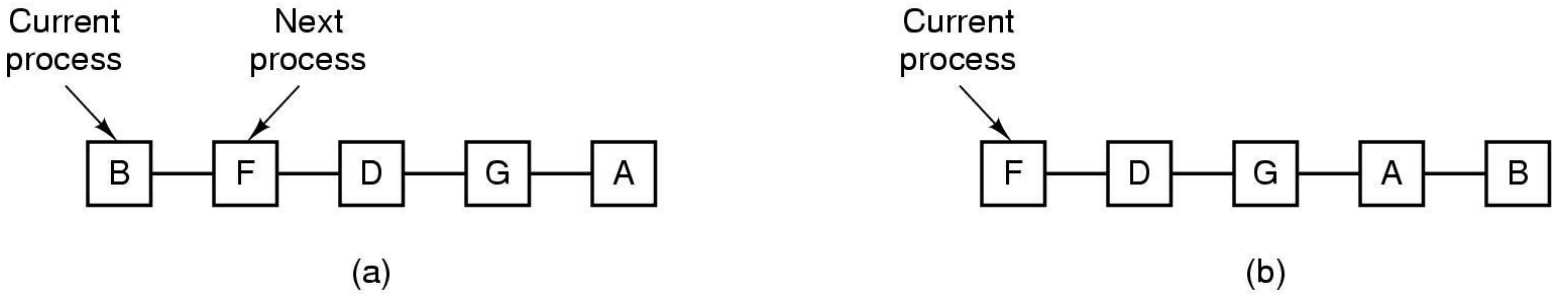
- Example: Servers
  - Serve multiple remote users all of whom are in a big hurry
- Performance Criteria
  - Min response time: respond to requests quickly
  - Best proportionality: meet users' expectation
- Representative algorithms:
  - Round-robin
  - Priority-based
  - Shortest process time
  - Guaranteed Scheduling
  - Lottery Scheduling
  - Fair Sharing Scheduling

Interactive System

- Algorithms used here usually preemptive
  - Time is **sliced** into quantum (time intervals)
  - Scheduling decision is also made at the beginning of each quantum
- Representative algorithms:
  - Round-robin
  - Priority-based
  - Shortest process time
  - Guaranteed Scheduling
  - Lottery Scheduling
  - Fair Sharing Scheduling

Round Robin

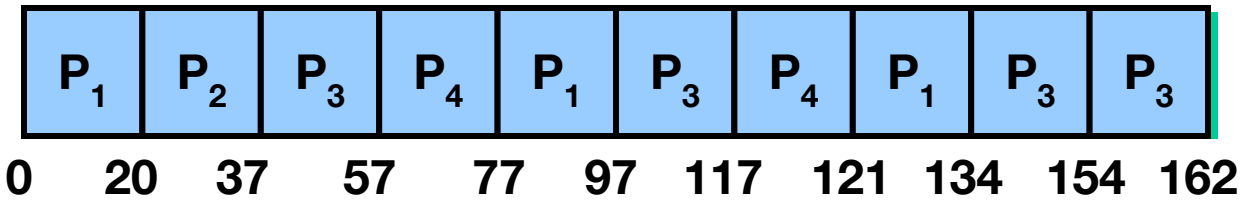
- **Round Robin (RR)**
  - Often used for timesharing
  - Ready queue is treated as a circular queue
  - Each process is given a time slice called a ***quantum***
  - It is run for the quantum or until it blocks
  - RR allocates the CPU uniformly (fairly) across participants from ready queue.
- Problem:
  - Do not consider priority
  - Context switch overhead



RR with Time Quantum = 20

Process	Run Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is



- Higher average turnaround than SJF,
- But better response time

RR: Choice of Time Quantum

- Performance depends on length of the timeslice
  - Context switching isn't a free operation.
  - If timeslice time is set too high
    - attempting to amortize context switch cost, you get FCFS.
      - i.e. processes will finish or block before their slice is up anyway
      - Poor response time
  - If it's set too low
    - you're spending all of your time context switching between threads.
  - A quantum around 20-50 msec is often a resonable compromise

Priority Scheduling

- Each job is assigned a priority.
- Select highest priority job to run next.
- Rational: higher priority jobs are more important
  - Example: DVD movie player vs. send email
- Problems:
  - Low priority process may starve
- Solution:
  - Priority need to be adjusted depending on the situation

Assign Priority

- Two approaches
  - Static (for system with well known and regular application behaviors)
  - Dynamic (otherwise)
- Priority may be based on:
  - Cost to user.
  - Importance of user.
  - Percentage of CPU time used in last X hours.

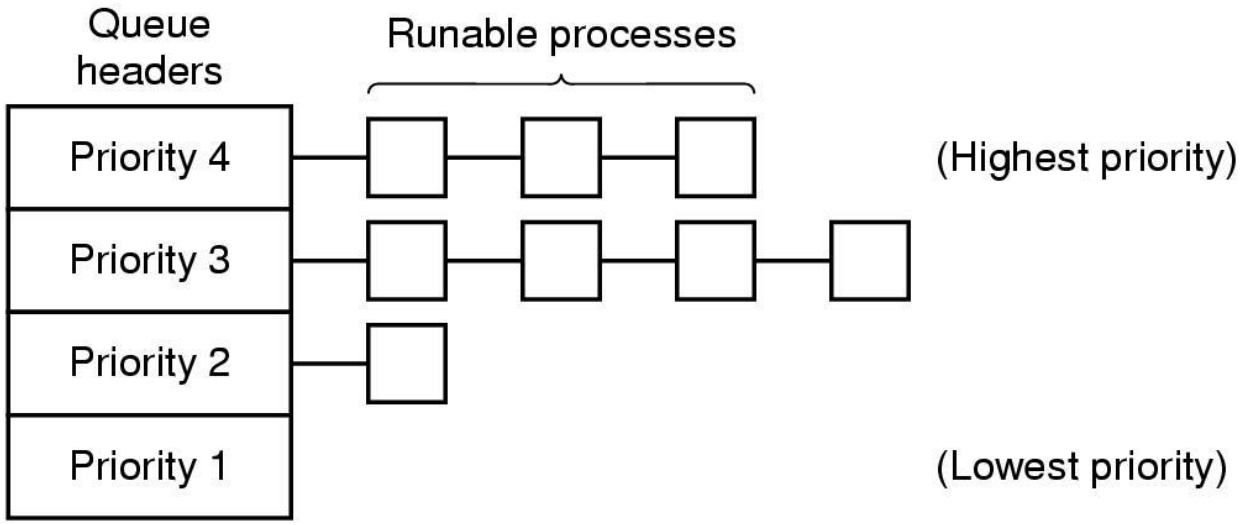
Example: Dynamic Priority Assignment

- Whenever highly I/O bound processes wants the CPU it should be given the CPU immediately.
- Why? 
- A simple algorithm for giving priority to I/O bound processes is to set the priority to $1/f$
  - f is the fraction of the last quantum used by a process
  - A process that used only 1 msec of its 50 msec quantum would get priority 50
  - A process that used 25 msec of its 50 msec quantum would get priority 2

CT 2 Syllabus ends here

Priority class

- It is often convenient to group processes into priority classes and use priority scheduling among the classes but RR within each class



- If priorities are not adjusted occasionally, lower priority classes may all starve to death

Shortest Process Next

- Let's apply SJF for interactive processes
- General pattern of a interactive process: wait for command, execute command, …
- Let's regard the execution of each command as a separate "job"
- Now we can minimize overall response time by running the process with shortest "job" first.

Shortest Process Next

- How to know which of the currently runnable processes has the the shortest job?
- A possible answer: Aging
- Make estimate based on past behavior and run the process with the shortest estimated time
  - Let the estimated time per command for some terminal is T_0
  - its next run is measured to be T_1
  - We could update our estimate by $aT_0 + (1-a)T_1$
  - For $a=2$, the successive estimates are

$$T_0,\quad T_0/2 + T_1/2,\quad T_0/4 + T_1/4 + T_2/2,\quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Guaranteed Scheduling

- Make promises to users about performance & then meet those promises.
- With n processes running, each one should get l/n of the CPU cycles.
- Calculate ratio for each process.

$$\frac{\text{Amount of CPU time process has had since its creation}}{\text{Amount of CPU time process should have since creation}}$$

- Run the process with the **lowest** ratio until its ratio has moved above its closest competitor
- Problem:
  - Implementation is difficult.

Lottery Scheduling

- Give processes lottery tickets for CPU time.
- Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the CPU.
- More important processes can be given extra tickets, to increase their chances of winning.
- If there are 100 tickets and one process holds 20 of them, it will have a 20% chance of winning each lottery.
- In the long run, it will get about 20% of the CPU.
- Highly Responsive:
  - if a new process shows up and is granted some tickets,
  - at the very next lottery it will have a chance of winning in proportion to the number of tickets it holds.

Lottery Scheduling

- Cooperating processes may exchange tickets if they wish.
  - When a client **process** sends a message to a server **process** and then blocks, it may give all of its tickets to the server, to increase the chance of the server running next.
  - After finishing, it returns the tickets so that the client can run again.
- Can solve problems that are difficult to handle with other methods
  - In a video server several processes are feeding video streams to their clients, but at different frame rates.
  - Let the processes need frames at 10, 20, and 25 frames/sec.
  - By allocating these processes 10, 20, and 25 tickets, respectively, they will automatically divide the CPU in approximately the correct proportion, that is, 10:20:25.

Fair Share Scheduling

- Schedule considering the process **owner**.
- Each **user** is allocated some fractions of the CPU and scheduler picks processes in such a way to enforce it.
  - Consider a system with **two users**, each of which has been promised 50% of the CPU.
  - User 1 has 4 processes, A, B, C, and D, and user 2 has only 1 process, E.
  - If *round-robin scheduling* is used, a possible scheduling sequence:
    A E B E C E D E A E B E C E D E ...
  - if user 1 is entitled to twice as much CPU time as user 2
    A B E C D E A B E C D E ...

Real-Time Systems

- **Time** plays an essential role
- Usually the computer must react appropriately to **events** generated by external devices within a fixed amount of time
  - Computer in disc player
    - Must convert data from the drive into sound within a very short time interval otherwise **!!?!?**
  - Patient monitoring, autopilot, robot control…
- Getting right answer but too late == Getting **nothing** at all

Real-Time Systems

- 2 types
  - Hard real time
  - Soft real time
- Real time behavior is achieved by
  - Divide the program into a number of predictable, short lived processes
  - When an external event is detected the scheduler schedules the processes properly to met all deadlines

Real-Time Systems

- 2 types of event
  - periodic
  - aperiodic
- A system may have to respond to multiple periodic event streams
- It is not always possible to handle all events

Scheduling in Real-Time Systems

- Given
  - *m* periodic events
  - event *i* occurs with period P_i and requires C_i seconds CPU time to handle
- Then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- A real-time system that meets this criterion is said to be **Schedulable**

Scheduling in Real-Time Systems

- Scheduling algorithm can be
  - Static: make Scheduling decisions **before** the system starts running
    - Need to know about the work to be done and the deadlines to meet
  - Dynamic: make Scheduling decisions at run time
- **BAD NEWS**: Perhaps we would not learn the specific algorithms in this course!!! Please be patient.

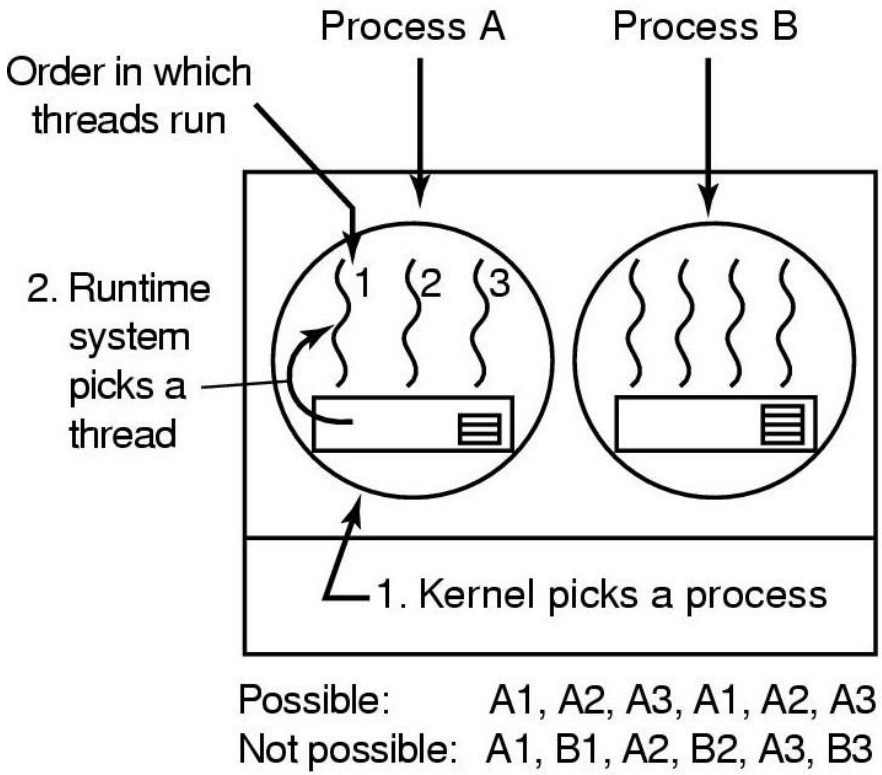
Policy versus Mechanism

- Separate what is allowed to be done with how it is done to make the best decision
  - a process may know which of its children processes are important and need priority
  - A DBMS process may have many children working on different things such as query processing, disk access etc
- How?
  - Scheduling algorithm is parameterized
    - mechanism in the kernel
  - Parameters filled in by user processes
    - policy set by user process

User-level Thread Scheduling

- Kernel picks a process and thread scheduler inside the process decides which thread to run
- Fast Thread switching
- Application specific thread scheduler can be used to maximize output
  - The run time system knows each thread under it

User-level Thread Scheduling



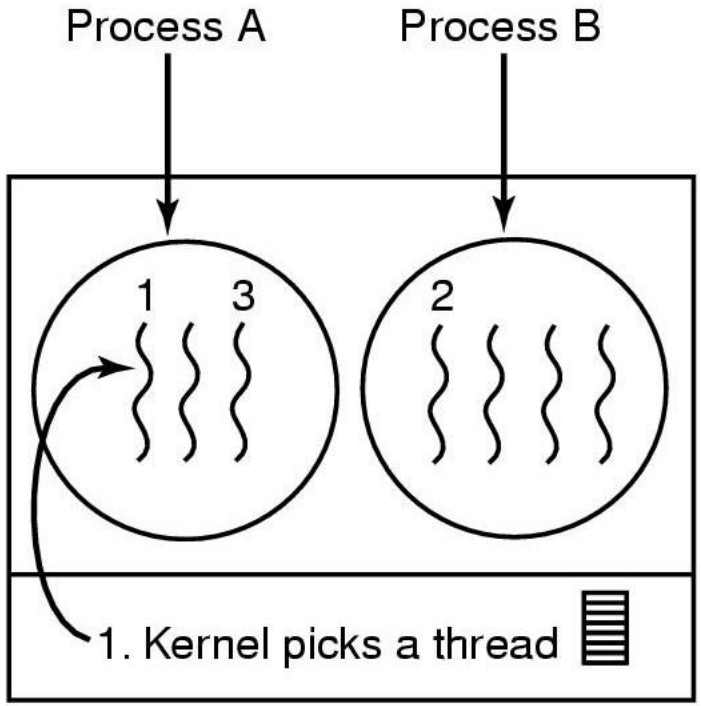
Possible scheduling of user-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

Kernel-level Thread Scheduling

- Kernel picks a particular thread to run
- Having a thread block on I/O does not suspend the entire process
- Expensive thread switching

Kernel-level Thread Scheduling



Possible scheduling of kernel-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst