A two-part lecture.
First: Some Monitor Examples
Then: Discussion of Scheduling

Ken Birman

Review: Monitors

- Why review something we just covered?
  - People find it hard to get into the "concurrency mindset"
  - It takes time to get used to this style of coding, and to thinking about the correctness of concurrent objects
- So: review a little, then discuss a related topic, and then (next time) tackle a new concurrency issue
  - Yes, this isn't "linear"...
  - ... but avoids "too much, too fast" ...

Producer Consumer using Monitors

```
public class Producer_Consumer {
   int N;
   Object[]  buf;
   int n = 0, tail = 0, head = 0;
   Object not_empty = new Object();
   Object not_full = new Object();

   public Producer_Consumer(int len) {
      buf = new object[len];
      N = len;
   }

   public void put(Object obj) {
      synchronized(not_full) {
         while(n == N)
             not_full.wait();
         buf[head%N] = obj;
         head++;
         synchronized(this) { n++; }
      }
      synchronized(not_empty) { not_empty.notify(); }
   }
```

```
   public Object get()  {
      Object obj;
      synchronized(not_empty) {
         while(n == 0)
            not_empty.wait();
         object obj = buf[tail%N];
         tail++;
         synchronized(this) { n--; }
      }
      synchronized(not_full) { not_full.notify(); }
      return obj;
   }
}
```

Subtle aspects

- When updating a variable shared by producer and consumer, the code needs to lock it
  - Hence synchronized(this) { n++; }
- But we can *read* the variable without interlocks:

```
while(n == N)
        not_full.wait();
```

Subtle aspects

- To call wait or notify, must first call synchronized on the object that you will wait or notify on, hence
  - synchronized(xyz) { xyz.wait(); }
  - synchronized(xyz) { xyz.notify(); }
  - ... if our code had been synchronized on something else, you get a "thread isn't the owner of this object" exception
- When you do call wait, do it inside a while loop and recheck the condition when you wake up
  - while(n == N) NotFull.wait();
  - This way, if someone wakes up the thread (because n < N) but some *other* thread sneaks in and now n==N again, your code will be safe.

Subtle aspects

- Notice that when a thread calls wait(), if it blocks it also automatically releases the lock on the object on which wait was done
- This is an elegant solution to an issue seen with semaphores
  - Caller did mutex.acquire().... But now needs to call not_empty.aquire()... and this second call might block
  - So we need to call mutex.release()...
    - Danger is that some other thread "slips in" between the two
    - A race condition!

Subtle aspects

- But…. Java has a bug…
- … nested synchronized() calls are "risky"
  - If the inner block calls wait, the outer lock won't be automatically released
  - Can easily result in deadlocks
- This is why our bounded buffer code synchronized on Not_Full…
  - … although sometimes we can even take advantage of this annoying behavior

Readers and Writers

```
public class ReadersNWriters {
   int NReaders = 0, NWriters = 0;
   Object CanBegin = new Object();

   public synchronized void BeginWrite()
   {
      synchronized(CanBegin) {


         NWriters = 1;
      }
   }
   public void EndWrite()
   {
      synchronized(CanBegin) {
         NWriters = 0;

      }
   }
```

```
public synchronized void BeginRead()
 {
      synchronized(CanBegin) {


         ++NReaders;
      }
   }
   public void EndRead()
   {
      synchronized(CanBegin) {
            --NReaders

      }
   }
}
```

Readers and Writers

```
public class ReadersNWriters {
   int NReaders = 0, NWriters = 0;
   Object CanBegin = new Object();

   public synchronized void BeginWrite()
   {
      synchronized(CanBegin) {
         if(NWriters == 1 || NReaders > 0)
            CanBegin.Wait();
         NWriters = 1;
      }
   }
   public void EndWrite()
   {
      synchronized(CanBegin) {
         NWriters = 0;

      }
   }
```

```
public synchronized void BeginRead()
 {
      synchronized(CanBegin) {
         if(NWriters == 1)
            CanBegin.Wait();
         ++NReaders;
      }
   }
   public void EndRead()
   {
      synchronized(CanBegin) {
            --NReaders

      }
   }
}
```

Readers and Writers

```
public class ReadersNWriters {
   int NReaders = 0, NWriters = 0;
   Object CanBegin = new Object();

   public synchronized void BeginWrite()
   {
      synchronized(CanBegin) {
         if(NWriters == 1 || NReaders > 0)
            CanBegin.Wait();
         NWriters = 1;
      }
   }
   public void EndWrite()
   {
      synchronized(CanBegin) {
         NWriters = 0;
         CanBegin.Notify();
      }
   }
```

```
public synchronized void BeginRead()
 {
      synchronized(CanBegin) {
         if(NWriters == 1)
            CanBegin.Wait();
         ++NReaders;
      }
   }
   public void EndRead()
   {
      synchronized(CanBegin) {
         if(--NReaders == 0)
            CanBegin.Notify();
      }
   }
}
```

Understanding the Solution

- If any thread is waiting, no other thread gets into BeginRead or BeginWrite
  - This is because of the "bug" mentioned before
  - When we nest synchronization blocks, a wait will only release the lock on the object we wait on… not the outer synchronization lock

Readers and Writers

```
public class ReadersNWriters {
   int NReaders = 0, NWriters = 0;
   Object CanBegin = new Object();

   public synchronized void BeginWrite()
   {
      synchronized(CanBegin) {
         if(NWriters == 1 || NReaders > 0)
            CanBegin.Wait();
         NWriters = 1;
      }
   }
   public void EndWrite()
   {
      synchronized(CanBegin) {
         NWriters = 0;
         CanBegin.Notify();
      }
   }
```

```
public synchronized void BeginRead()
 {
      synchronized(CanBegin) {
         if(NWriters == 1)
            CanBegin.Wait();
         ++NReaders;
      }
   }
   public void EndRead()
   {
      synchronized(CanBegin) {
         if(--NReaders == 0)
            CanBegin.Notify();
      }
   }
}
```

Understanding the Solution

- A writer can enter if there are no other active writers and no readers are waiting

Readers and Writers

```
public class ReadersNWriters {
   int NReaders = 0, NWriters = 0;
   Object CanBegin = new Object();

   public synchronized void BeginWrite()
   {
      synchronized(CanBegin) {
         if(NWriters == 1 || NReaders > 0)
             CanBegin.Wait();
         NWriters = 1;
      }
   }
   public void EndWrite()
   {
      synchronized(CanBegin) {
         NWriters = 0;
         CanBegin.Notify();
      }
   }

public synchronized void BeginRead()
 {
      synchronized(CanBegin) {
         if(NWriters == 1)
            CanBegin.Wait();
         ++NReaders;
      }
   }
   public void EndRead()
   {
      synchronized(CanBegin) {
         if(--NReaders == 0)
            CanBegin.Notify();
      }
   }
}
```

Understanding the Solution

- A reader can enter if
  - There are no writers active or waiting
- So we can have many readers active all at once
- Otherwise, a reader waits (maybe many do)

Readers and Writers

```
public class ReadersNWriters {
   int NReaders = 0, NWriters = 0;
   Object CanBegin = new Object();

   public synchronized void BeginWrite()
   {
      synchronized(CanBegin) {
         if(NWriters == 1 || NReaders > 0)
             CanBegin.Wait();
         NWriters = 1;
      }
   }
   public void EndWrite()
   {
      synchronized(CanBegin) {
         NWriters = 0;
         CanBegin.Notify();
      }
   }

public synchronized void BeginRead()
 {
      synchronized(CanBegin) {
         if(NWriters == 1)
            CanBegin.Wait();
         ++NReaders;
      }
   }
   public void EndRead()
   {
      synchronized(CanBegin) {
         if(--NReaders == 0)
            CanBegin.Notify();
      }
   }
}
```

Understanding the Solution

- When a writer finishes, it lets one thread in, if someone is waiting
  - Doesn’t matter if this is one writer, or one reader
  - If a reader gets in, and then another shows up, it will get in too… once it gets past the “entry” synchronization
- Similarly, when a reader finishes, if it was the last reader, it lets a writer in (if any is there)

Readers and Writers

```
public class ReadersNWriters {
   int NReaders = 0, NWriters = 0;
   Object CanBegin = new Object();

   public synchronized void BeginWrite()
   {
      synchronized(CanBegin) {
         if(NWriters == 1 || NReaders > 0)
             CanBegin.Wait();
         NWriters = 1;
      }
   }
   public void EndWrite()
   {
      synchronized(CanBegin) {
         NWriters = 0;
         CanBegin.Notify();
      }
   }

public synchronized void BeginRead()
 {
      synchronized(CanBegin) {
         if(NWriters == 1)
            CanBegin.Wait();
         ++NReaders;
      }
   }
   public void EndRead()
   {
      synchronized(CanBegin) {
         if(--NReaders == 0)
            CanBegin.Notify();
      }
   }
}
```

Understanding the Solution

- It wants to be fair
  - If a writer is waiting, readers queue up “outside”
  - If a reader (or another writer) is active or waiting, writers queue up

Comparison with Semaphores

- With semaphores we never had a “fair” solution
  - In fact it can be done, in much the same way
- The monitor is relatively simple… and it works!
  - In general, monitors are relatively less error prone
  - Still, this particular one isn’t so easy to derive or understand, because it makes subtle use of the synchronization lock
- Our advice: teams should agree to use monitor style

Slight topic shift…

Scheduling

Why discuss?

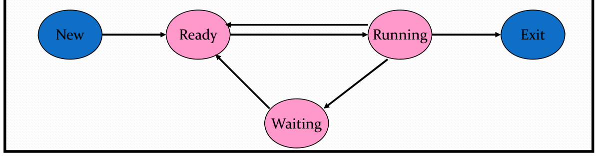
- Hidden under the surface in our work up to now is the role of the scheduler
- It watches the pool of threads, or processes
- And when the current thread/process waits, or has been running for too long
  - If necessary, “preempts” the current thread/process
  - Selects something else to run
  - Context switches to it

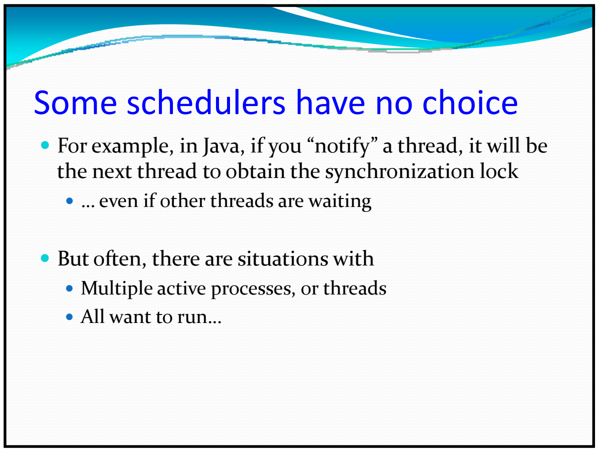
OK… so who cares?

- We’ve worried about the fairness of our solutions
- But implicit in this is the assumption that the scheduler itself is reasonably fair
  - At a minimum, that every runnable thread/process gets plenty of opportunities to run
- Let’s look more closely at how schedulers really work

Process Scheduling

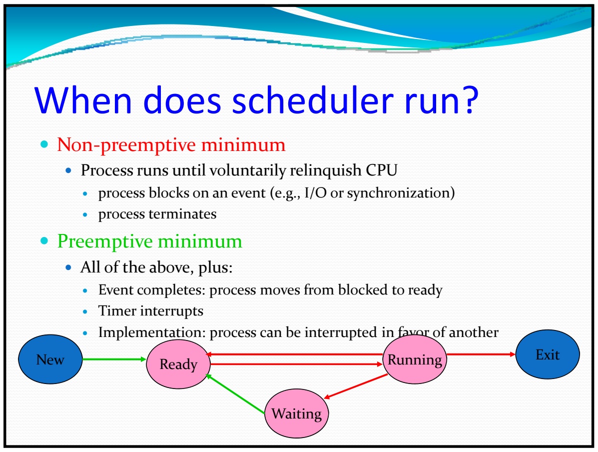
- Rest of lecture: “process” and “thread” used interchangeably
- Many processes in “ready” state
- Which ready process to pick to run on the CPU?
  - 0 ready processes: run idle loop
  - 1 ready process: easy! But if > 1 ready process: what to do?





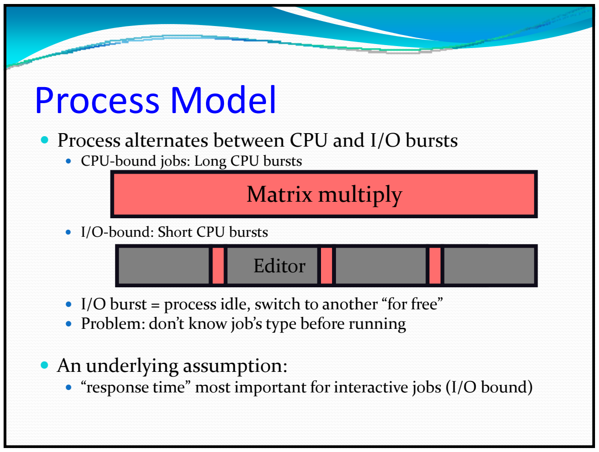
Some schedulers have no choice

- For example, in Java, if you "notify" a thread, it will be the next thread to obtain the synchronization lock
  - … even if other threads are waiting
- But often, there are situations with
  - Multiple active processes, or threads
  - All want to run…



When does scheduler run?

- Non-preemptive minimum
  - Process runs until voluntarily relinquish CPU
    - process blocks on an event (e.g., I/O or synchronization)
    - process terminates
- Preemptive minimum
  - All of the above, plus:
    - Event completes: process moves from blocked to ready
    - Timer interrupts
    - Implementation: process can be interrupted in favor of another

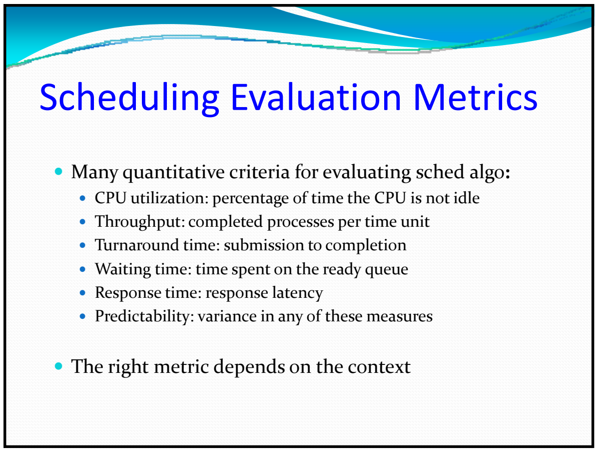


Process Model

- Process alternates between CPU and I/O bursts
  - CPU-bound jobs: Long CPU bursts

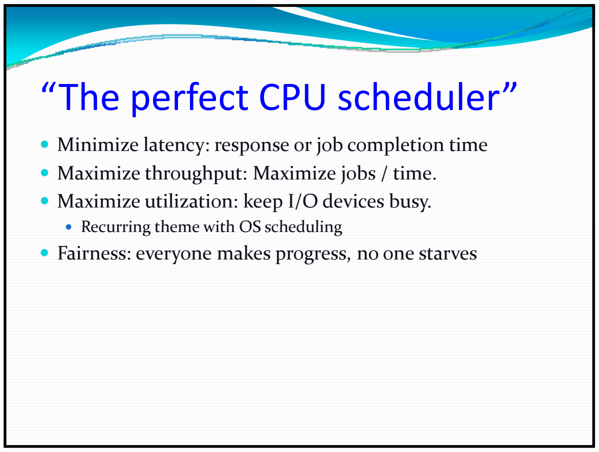
  - I/O-bound: Short CPU bursts

  - I/O burst = process idle, switch to another "for free"
  - Problem: don't know job's type before running
- An underlying assumption:
  - "response time" most important for interactive jobs (I/O bound)



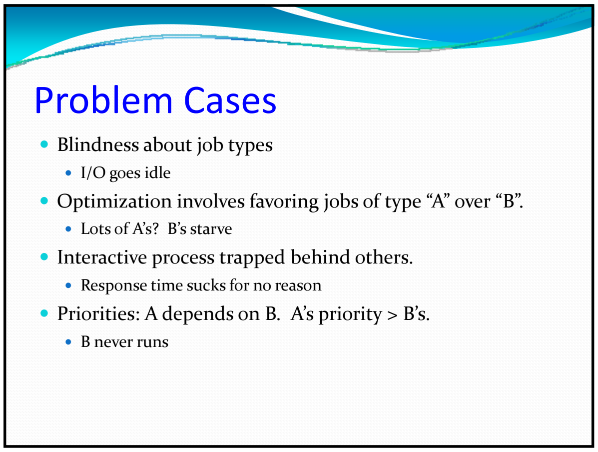
Scheduling Evaluation Metrics

- Many quantitative criteria for evaluating sched algo**:**
  - CPU utilization: percentage of time the CPU is not idle
  - Throughput: completed processes per time unit
  - Turnaround time: submission to completion
  - Waiting time: time spent on the ready queue
  - Response time: response latency
  - Predictability: variance in any of these measures
- The right metric depends on the context



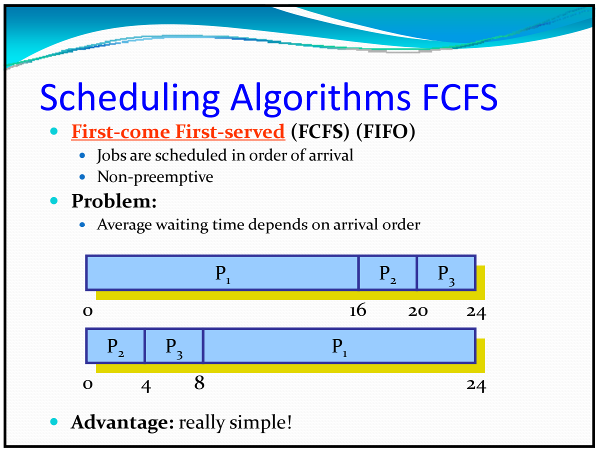
"The perfect CPU scheduler"

- Minimize latency: response or job completion time
- Maximize throughput: Maximize jobs / time.
- Maximize utilization: keep I/O devices busy.
  - Recurring theme with OS scheduling
- Fairness: everyone makes progress, no one starves



Problem Cases

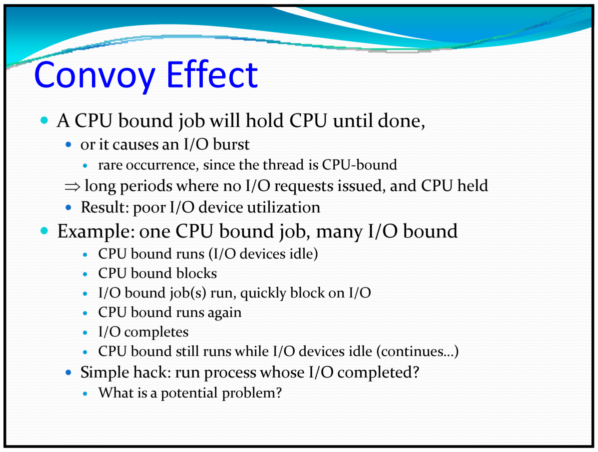
- Blindness about job types
  - I/O goes idle
- Optimization involves favoring jobs of type "A" over "B".
  - Lots of A's? B's starve
- Interactive process trapped behind others.
  - Response time sucks for no reason
- Priorities: A depends on B. A's priority > B's.
  - B never runs



Scheduling Algorithms FCFS

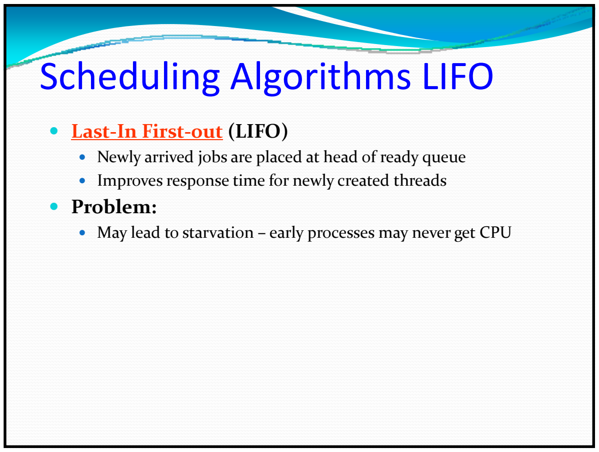
- First-come First-served **(FCFS) (FIFO)**
  - Jobs are scheduled in order of arrival
  - Non-preemptive
- **Problem:**
  - Average waiting time depends on arrival order

- **Advantage:** really simple!



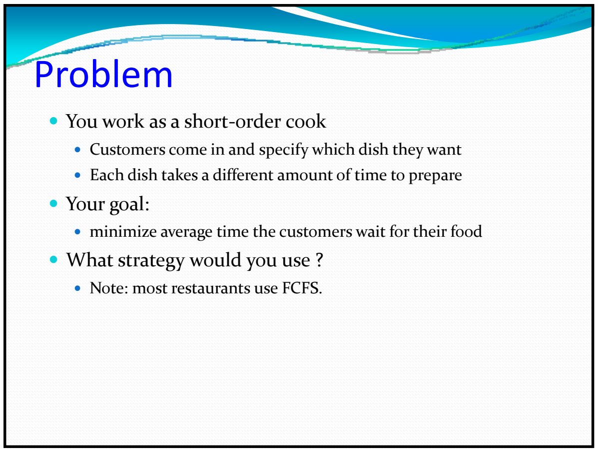
Convoy Effect

- A CPU bound job will hold CPU until done,
  - or it causes an I/O burst
    - rare occurrence, since the thread is CPU-bound
  - ⇒ 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?



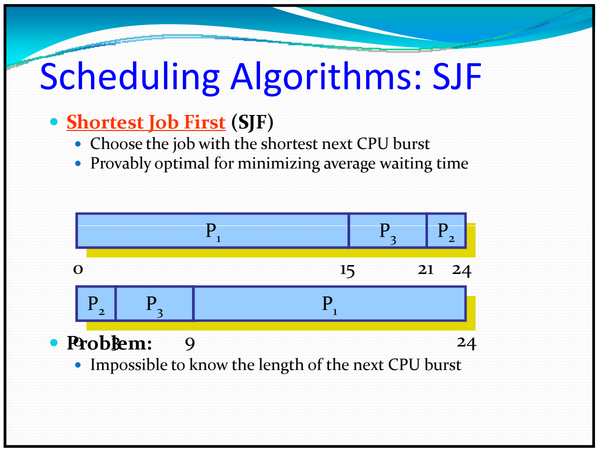
Scheduling Algorithms LIFO

- Last-In First-out **(LIFO)**
  - Newly arrived jobs are placed at head of ready queue
  - Improves response time for newly created threads
- **Problem:**
  - May lead to starvation – early processes may never get CPU



Problem

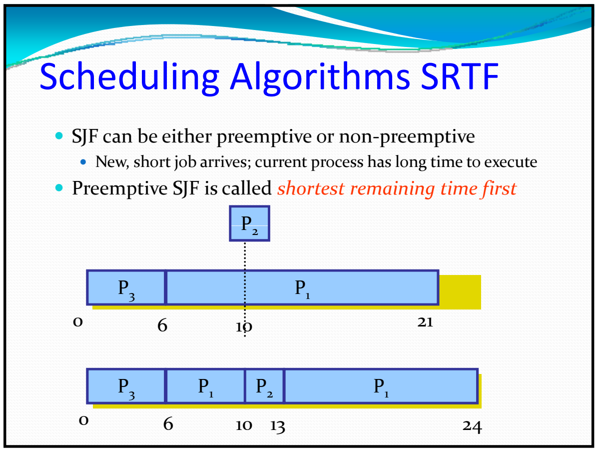
- You work as a short-order cook
  - Customers come in and specify which dish they want
  - Each dish takes a different amount of time to prepare
- Your goal:
  - minimize average time the customers wait for their food
- What strategy would you use ?
  - Note: most restaurants use FCFS.



Scheduling Algorithms: SJF

- Shortest Job First **(SJF)**
  - Choose the job with the shortest next CPU burst
  - Provably optimal for minimizing average waiting time

- **Problem:**
  - Impossible to know the length of the next CPU burst



Scheduling Algorithms SRTF

- SJF can be either preemptive or non-preemptive
  - New, short job arrives; current process has long time to execute
- Preemptive SJF is called *shortest remaining time first*

Shortest Job First Prediction

- Approximate next CPU-burst duration
  - from the durations of the previous bursts
    - The past can be a good predictor of the future
- No need to remember entire past history
- Use exponential average:

  t_n duration of the n^{th} CPU burst

  τ_{n+1} predicted duration of the $(n+1)^{st}$ CPU burst

  $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\, \tau_n$

  where $0 \le \alpha \le 1$

  α determines the weight placed on past behavior

Priority Scheduling

- **Priority Scheduling**
  - Choose next job based on priority
  - For SJF, priority = expected CPU burst
  - Can be either preemptive or non-preemptive
- **Problem:**
  - Starvation: jobs can wait indefinitely
- **Solution to starvation**
  - Age processes: increase priority as a function of waiting time

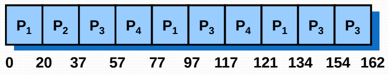
Round Robin

- **Round Robin** **(RR)**
  - Often used for timesharing
  - Ready queue is treated as a circular queue (FIFO)
  - 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.
  - If average queue length is n, each participant gets 1/n

RR with Time Quantum = 20

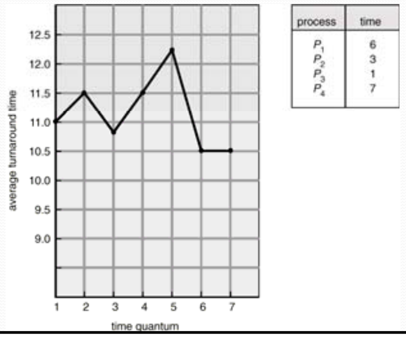
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



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

Turnaround Time w/ Time Quanta

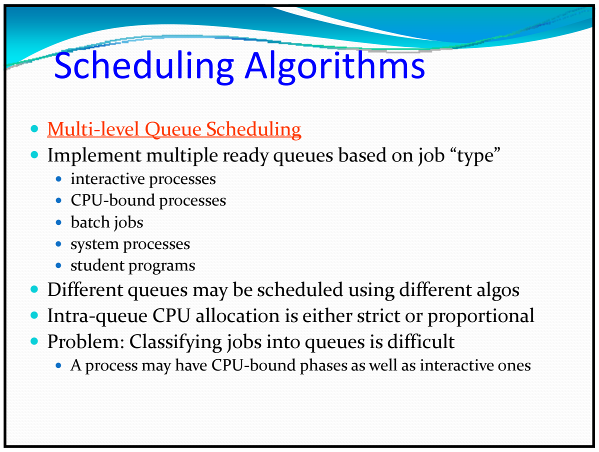


process	time
P_1	6
P_2	3
P_3	1
P_4	7

RR: Choice of Time Quantum

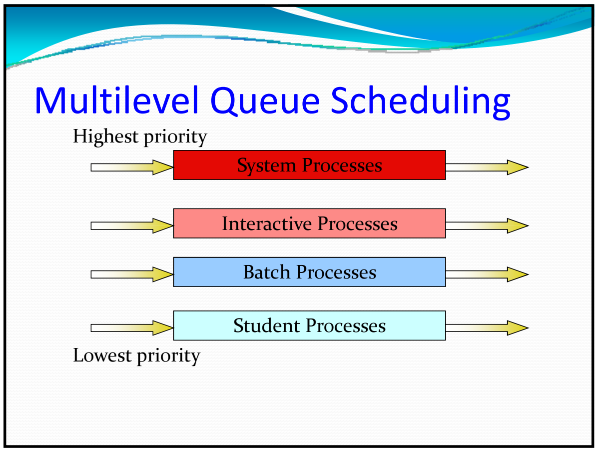
- Performance depends on length of the timeslice
  - Context switching isn't a free operation.
  - If timeslice time is set too high
    - attempting to amortize context switch cost, you get FCFS.
    - i.e. processes will finish or block before their slice is up anyway
  - If it's set too low
    - you're spending all of your time context switching between threads.
  - Timeslice frequently set to ~100 milliseconds
  - Context switches typically cost < 1 millisecond

Moral:
Context switch is usually negligible (< 1% per timeslice) unless you context switch too frequently and lose all productivity



Scheduling Algorithms

- Multi-level Queue Scheduling
- Implement multiple ready queues based on job "type"
  - interactive processes
  - CPU-bound processes
  - batch jobs
  - system processes
  - student programs
- Different queues may be scheduled using different algos
- Intra-queue CPU allocation is either strict or proportional
- Problem: Classifying jobs into queues is difficult
  - A process may have CPU-bound phases as well as interactive ones



Multilevel Queue Scheduling

Highest priority

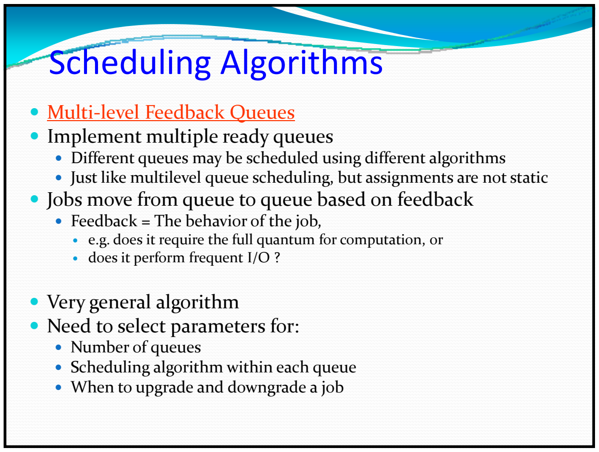
System Processes

Interactive Processes

Batch Processes

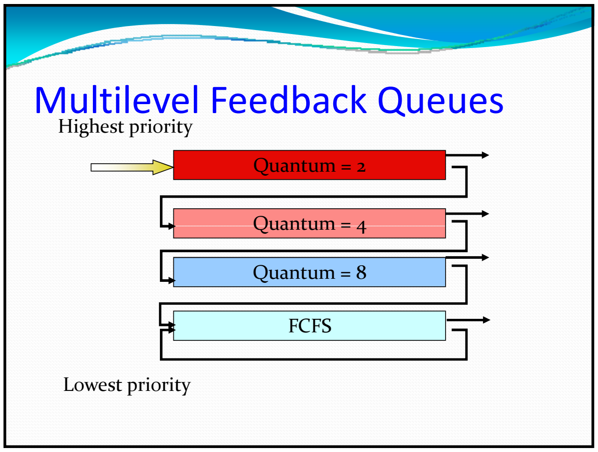
Student Processes

Lowest priority



Scheduling Algorithms

- Multi-level Feedback Queues
- Implement multiple ready queues
  - Different queues may be scheduled using different algorithms
  - Just like multilevel queue scheduling, but assignments are not static
- Jobs move from queue to queue based on feedback
  - Feedback = The behavior of the job,
    - e.g. does it require the full quantum for computation, or
    - does it perform frequent I/O ?
- Very general algorithm
- Need to select parameters for:
  - Number of queues
  - Scheduling algorithm within each queue
  - When to upgrade and downgrade a job



Multilevel Feedback Queues

Highest priority

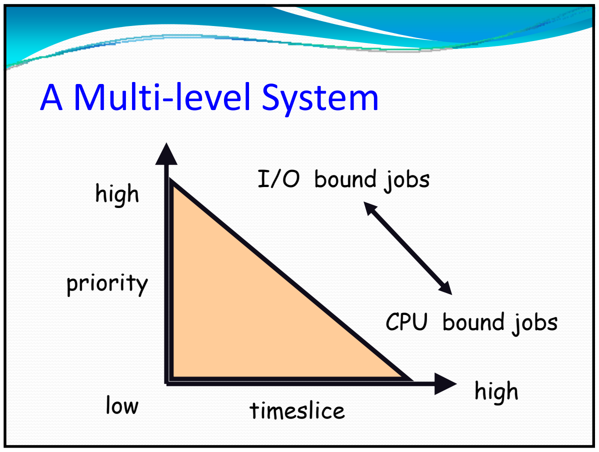
Quantum = 2

Quantum = 4

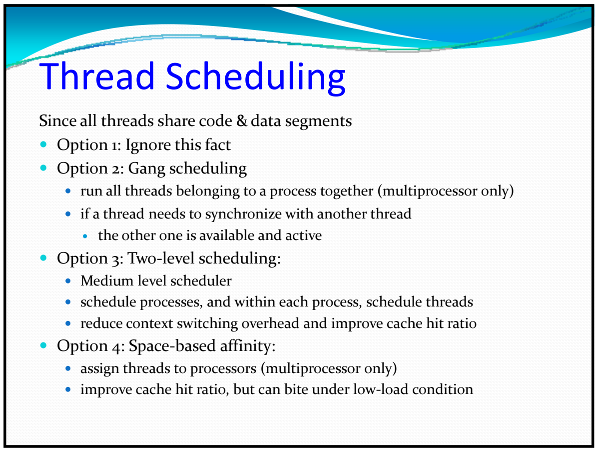
Quantum = 8

FCFS

Lowest priority



A Multi-level System



Thread Scheduling

Since all threads share code & data segments

- Option 1: Ignore this fact
- Option 2: Gang scheduling
  - run all threads belonging to a process together (multiprocessor only)
  - if a thread needs to synchronize with another thread
    - the other one is available and active
- Option 3: Two-level scheduling:
  - Medium level scheduler
  - schedule processes, and within each process, schedule threads
  - reduce context switching overhead and improve cache hit ratio
- Option 4: Space-based affinity:
  - assign threads to processors (multiprocessor only)
  - improve cache hit ratio, but can bite under low-load condition

Real-time Scheduling

- Real-time processes have timing constraints
  - Expressed as deadlines or rate requirements
- Common RT scheduling policies
  - Rate monotonic
    - Just one scalar priority related to the periodicity of the job
    - Priority = 1/rate
    - Static
  - Earliest deadline first (EDF)
    - Dynamic but more complex
    - Priority = deadline
- Both require admission control to provide guarantees

Postscript

- The best schemes are adaptive.
- To do absolutely best we'd have to predict the future.
  - Most current algorithms give highest priority to those that need the least!
- Scheduling become increasingly ad hoc over the years.
  - 1960s papers very math heavy, now mostly "tweak and see"

Problem

- What are metrics that schedulers should optimize for ?
  - There are many, the right choice depends on the context
- Suppose:
  - You own an airline, have one expensive plane, and passengers waiting all around the globe
  - You own a sweatshop, and need to evaluate workers
  - You are at a restaurant, and are waiting for food
  - You are an efficiency expert, and are evaluating government procedures at the DMV
  - You are trying to find a project partner or a co-author
  - You own a software company that would like to deliver a product by a deadline