ELEC 377 – Operating Systems

Week 4 – Class 1

Last Class

- Finished Semaphores
- Classic Problems
- Critical Regions
- Monitors

Today

- Regions & Monitors
- Java Synchronization
- Scheduling

Critical Regions

- shared variable is used

```
shared T v;
```

example:

```
shared int v1;

struct xyzzy {
     char * a;
     int b;
 }
shared struct xyzzy v2;
```

Critical Regions

- special language construct to access shared variable

```
region v when B do S;
```

◊ B is a boolean condition
◊ S is one or more statements

example:

```
region v1 when (true) do v1++;

region v2 when (v2.b > 0) {
    printf("%s\n",v2.a);
    b--;
}
```

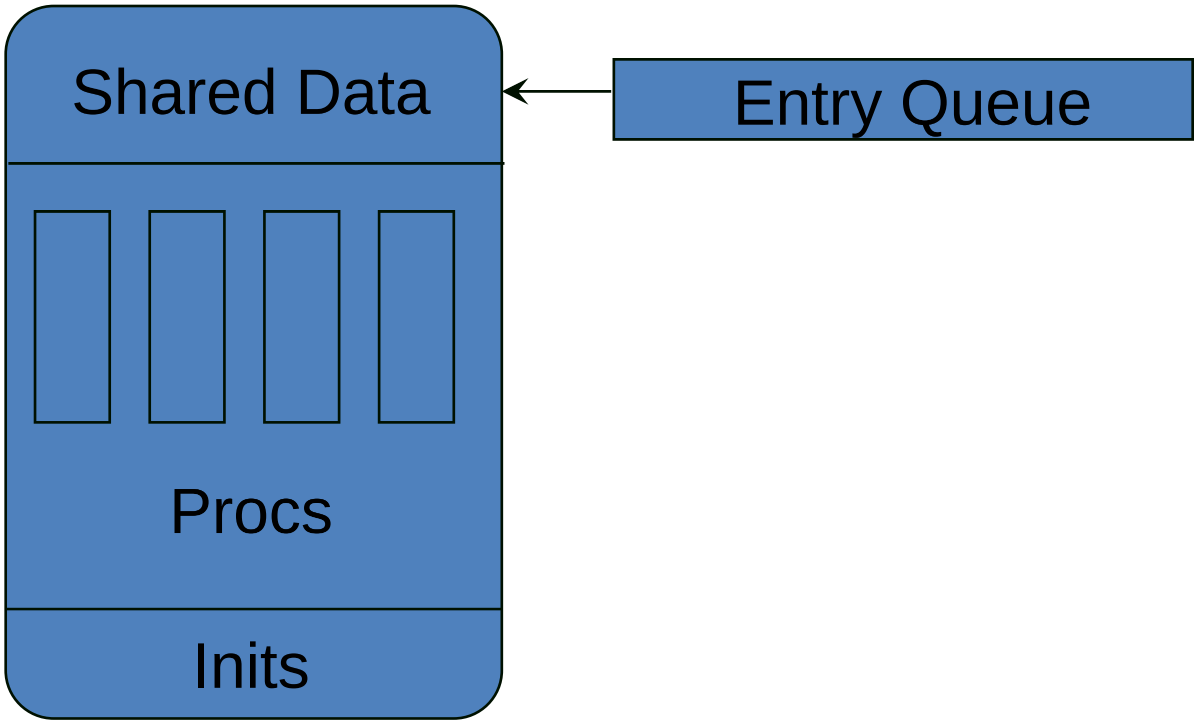
Monitors

- high level synchronization construct

◊ allows safe sharing of an abstract data type

```
monitor Accout {
  float balance;
  procedure deposit(float amt){
    balace += amt;
  }
  procedure withdrawl(float amt){
    balance -= amt;
  }
  {
    balance = 0.0;
  }
}
```

Monitors

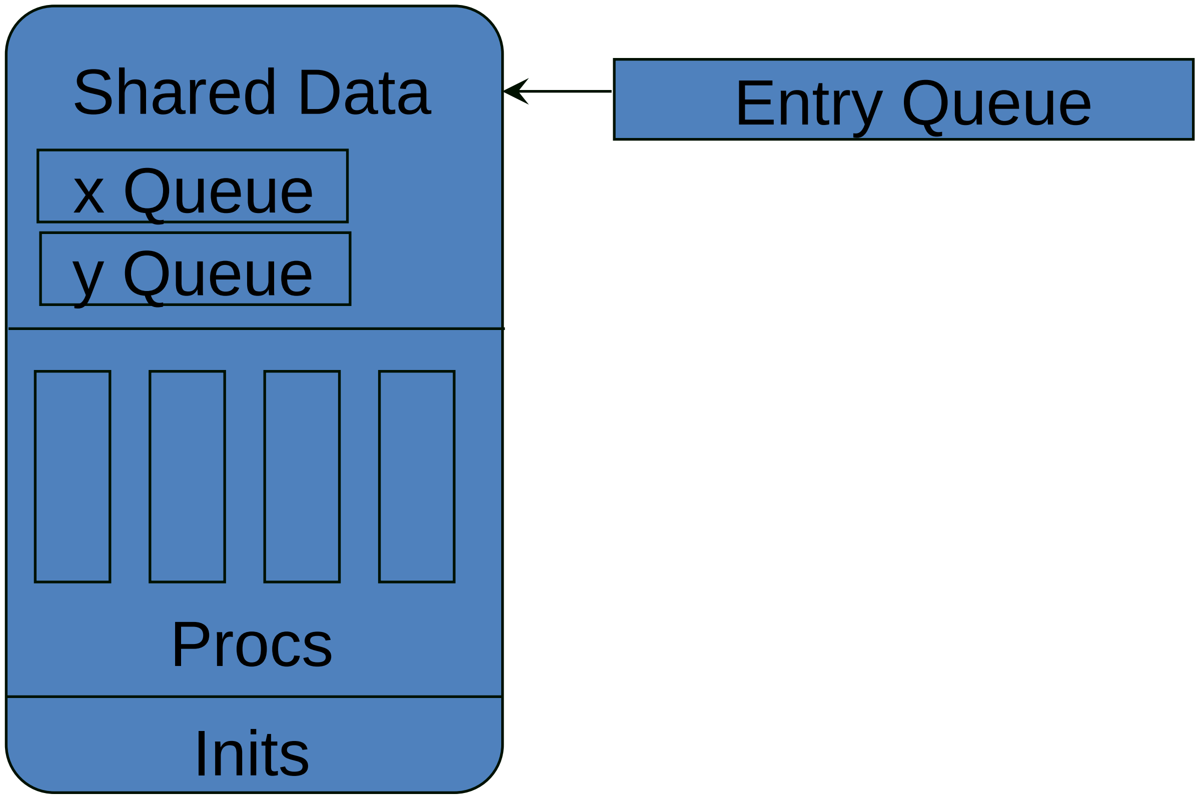


Monitors

- Processes may want to wait for another process
  - ◊ e.g. a buffer might be full!
- Condition variable
  - ◊ declared in shared variable section, private to monitor

    ```
    condition x,y;
    ```
  - ◊ Two operations, wait and signal
  - ◊ x.wait() means go to sleep and yield lock on monitor
  - ◊ x.signal() means wake up one process if there is a process that did an x.wait(). A process that did a y.wait() is not affected. If no process are waiting on condition x, then no effect.

Monitors



Signal()

- When a process executes x.signal() and another process is waiting on condition X, what happens?
- Several cases

◊ First process (signaler) goes to sleep until second process exits (releases lock) or waits on another condition

◊ First process continues until it leaves or waits on a condition and then signaled process continues

Producer Consumer

```
monitor buffer {
  condition full, empty;
  procedure addX(char X){
    if (buffer is full) full.wait();
    …
    empty.signal();
  }
  procedure getX(char &X){
    if (buffer is empty) empty.wait();
    …
    full.signal();
  }
  …
}
```

Monitors

- Prioritized waiting
  - ◊ x.wait(c) – c is an integer expressions
  - ◊ gives priority on queue for X
- System correctness
  - ◊ easier than semaphores
  - ◊ use monitors to guard shared resources, but not put shared resources inside monitor (may be more than one)
  - ◊ must still make sure that process makes correct monitor calls
  - ◊ concurrent processing is tricky!!

The two meanings of wait and signal

- Two versions of wait and signal
- Semaphores

  Semaphore mutex = 1

  wait (mutex)

  ...

  signal(mutex)
- Monitors

  Condition x

  x.wait()

  ...

  x.signal()

The two meanings of wait and signal

- Semaphores
  integer variable
  user visible value influences operation
- Monitor Condition
  Queue variable
  No user visible value

– wait in a semaphore may go right through
  (value > 0)
– wait in a monitor always means stop

Today

- Monitors
- Java Synchronization <<<<<<
- Scheduling

Java Synchronized

- The java synchronized keyword provides high level synchronization

- Two cases:

◊ synchronized methods – similar to monitors

◊ synchronized blocks – similar to critical regions

Java Synchronized Methods

- synchronized keyword is applied to methods

```
        class buffer{
 …
 public synchronized boolean putX(int x)
 …
}
```

- the instance of the class is the shared entity

  buffer a = new buffer();
  buffer b = new buffer();
  two processes may not call a.putX() at the same time
  they may call a.putX() and b.putX() at the same time

Java Synchronized Blocks

- closer to Critical Regions

```
…
synchronized (x) {
  …
}
…
```

- x must be an object pointer (not integral type)
  - ◊ lock is on object given by x
  - ◊ other threads with similar synchronized blocks may have different variables, but bound to same object
  - ◊ Java only provides a single lock on an object, so a synchronized block

1) with a given object and
2) a synchronized method in class of the object

are mutually exclusive

Java Wait() and Notify()

- Yet a third wait, but only two signals
- like wait() and signal() in monitors, but only one (implicit) condition variable.
- Producer/Consumer problem as given uses two condition variables. Can be done with one condition variable.

Java Produce Consumer

```
class buffer{
  vars for buffer
  public synchronized void putX(int x){
    while(buffer is full){wait();}
    …add x to buffer…
    notify();
  }
  public synchronized int getX(){
    int retval;
    while(buffer is empty){wait();}
    …remove from buffer into retval…
    notify();
    return retval;
  }
}
```

Today

- Monitors
- Java Synchronization
- Scheduling <<<<<<

Scheduling – Basic Concepts

- Goal: Maximum CPU utilization

◊ give CPU to another process while other is waiting I/O

- Processes proceed in bursts

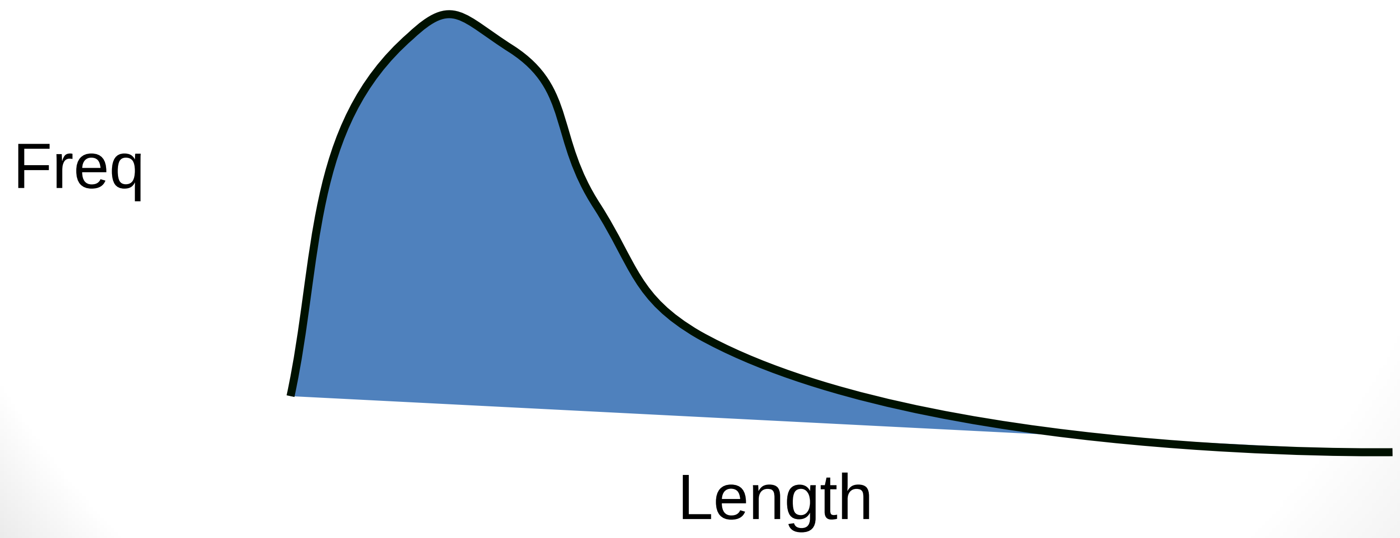
◊ Do some work

◊ Do some I/O

◊ repeat

Processing Bursts

- Most CPU Bursts are short
- ◊ Extensively studied
- ◊ The longer the burst, the less likely it is to occur
- ◊ Hyper exponential distribution
- ◊ Parameters of curve depend on OS, Applications



CPU Scheduler

- Selects processes from ready queue and allocates to CPU
- When?

1 Processes goes to wait state (I/O, event wait, etc.)
2 Process is interrupted
3 Process goes from wait to ready (I/O completes)
4 Process Terminates

◊ 1 and 4 are nonpreemptive
◊ Others are preemptive

Dispatcher

- *Dispatcher* is the part of the scheduler responsible for performing the context switch and resuming the process

- *Dispatch Latency*

◊ time for dispatcher to run

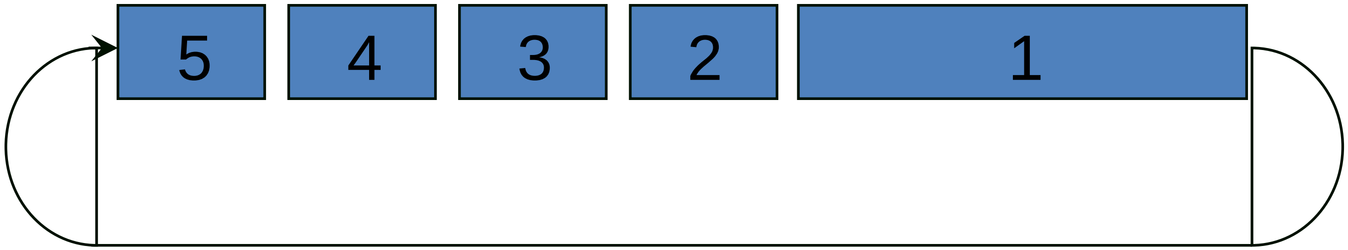
Scheduling Criteria

- CPU utilization – keep CPU as busy as possible
- Throughput – # of jobs done per time unit
- Turnaround Time – Time of submission to Time of Completion
- Waiting Time – amount of time in ready queue
- Response Time – submit time to time of first output request

First Come First Served (FCFS)

- Simple, easy to implement
- ◊ ready queue is a first-in-first-out (FIFO) queue
- Average Waiting time may be long
- ◊ Short Bursty I/O do not have priority over CPU intensive jobs
- ◊ variance in wait time/throughput is large, depends on order of jobs
- ◊ *Convoy effect*, all I/O jobs end up behind CPU jobs which hog the CPU
- Tends to make poor response in interactive systems
- ◊ used only in simple OS or in systems with very little variance in CPU burst times

Convoy Effect



- Tends to make poor response in interactive systems

◊ used only in simple OS or in systems with very little variance in CPU burst times

- How do we handle high variance in CPU Burst times?

Shortest Job First (SJR)

- scheduling ordered on the length of the next CPU burst
- Nonpreemptive - process gets entire slot
- Preemptive - after interrupt, if a new process with a CPU burst time less than remaining time, then current process looses CPU (Shortest Remaining Time First [SRTF])
- SJF is optimal for minimal waiting time

Estimating CPU Burst Times

- Use length of last CPU burst – exponential average

t_n = current CPU burst time

$|_0$ = initial estimate

$|_n$ = predicted for current burst

$|_{n+1}$ = prediction for next CPU burst

$\langle$ = weighting parameter

$$|_{n+1} = \langle\ t_n + (1 - \langle)\ |_n$$

$\langle = 0$ □ $|_{n+1} = |_0$ (initial estimate) never changes

$\langle = 1$ □ $|_{n+1} = t_n$ (last time slice) only used

Estimating CPU Burst Times

- Use length of last CPU burst – exponential average

$|_0 = 10$

$t_0 = 5, t_1 = 5, t_2 = 5, t_3 = 8, t_4 = 8$

$\langle = 0.3$

$|_1 = 0.3 * 5 + (0.7) * 10 = 8.5$

$|_2 = 0.3 * 5 + (0.7) * 8.5 = 7.45$

$|_3 = 0.3 * 5 + (0.7) * 7.45 = 6.715$

$|_2 = 0.3 * 8 + (0.7) * 6.715 = 7.1005$

$|_2 = 0.3 * 8 + (0.7) * 7.1005 = 7.37035$

Estimating CPU Burst Times

- predicted time always lags real time
- If process spends a reasonable period of time at a constant burst range then estimate approaches current burst time
- what is reasonable? how to tune?

◊ 〈 is the tuning parameter

◊ 〈 is low, then past behaviour has heavier weight, estimate is slower to change
  - ignore transient behaviour

◊ 〈 is high, then last time slice has heavier weight, estimate is faster to change
  - faster to adapt to changes

Priority Scheduling

- Each process has a priority

- CPU goes to process with highest priority

◊ ready queue is sorted based on priority
◊ same priority is handled FCFS
◊ preemptive / nonpreemptive
◊ internal/external

- SJF is priority scheduling with priority based on length of next CPU burst.

Round Robin (RR)

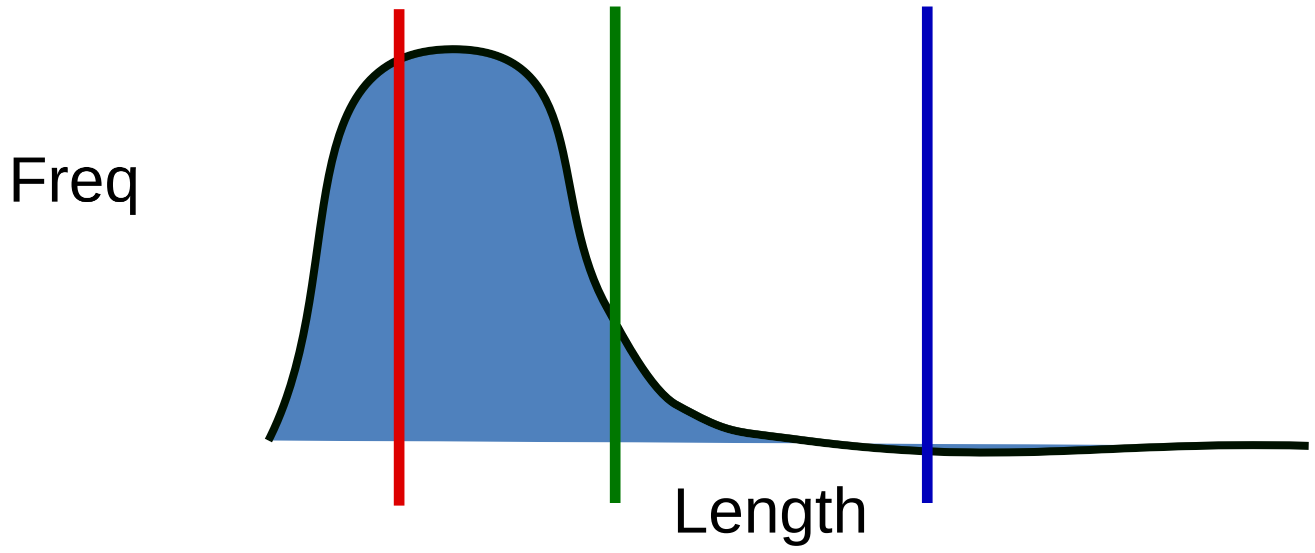
- Similar to FCFS, but add preemption.
- Designed for Time Sharing Systems
- Time slice (*quantum*) –> maximum time process gets to run
- if the quantum (q) is large –> FCFS
- if q is small, then appears to be multiple slower CPU’s (processor sharing).
- context switching is not free

◊ shorter q, more context switches to complete a single CPU burst for a given process

◊ q must be large with respect to context switch time

◊ 80% of CPU bursts should be shorter than q.

Round Robin – Quantum Length



multilevel Queues

highest priority

System Processes
Interactive Processes
Interactive Editing Processes
Batch Processes
Experimental Processes

lowest priority

Multi Level Queues

- Each queue has it’s own scheduling algorithm
- Interactive (foreground) - Round Robin
- Scheduling must be done between the queues

◊ usually fixed priority preemptive scheduling (starvation)

◊ time slice between queues (portion time between queues)

- In simplest form, processes are assigned a queue and remain there until completion
- Higher priority queues may require more money, or more status

Multi Level Feedback

- processes move between queues
- when doing I/O, processes move to higher priority queues
- When CPU intensive, processes move to lower priority queues
- Give higher priority queues smaller quanta (preemptive)
- Processes that use entire quanta are too high priority, bump down to lower priority queue
- Processes that don’t use entire quanta are too low priority and moved up to a higher priority queue

Multi Level Feedback

- parameters

◊ number of queues

◊ the scheduling algorithm for each queue

◊ when to upgrade a process

◊ when to downgrade a process

◊ how to choose the initial queue

- most complex algorithm, is approximated using priorities

Scheduling Algorithms

- FIFO - non preemptive
- SJF - non-preemptive (exponential average)
- SRTF - preemptive
- priority - preemptive/non-preemptive

◊ aging

- Round Robin - preemptive (quantum)
- Multiple queues

◊ multiple scheduling algorithms

◊ mutli level feedback

Multiple Processors

- Scheduling is more complex

◊ usually a common queue for all processors (load sharing)

◊ sometimes hardware limitations (I/O)

◊ actual parallel system, have to watch access to kernel data structures such as PCBs and Queues.

- Homogenous/memory sharing processors

- Symmetric / Asymmetric

Real Time Scheduling

- Hard Real Time
- ◊ guaranteed completion times
- ◊ resource reservation
- ◊ dedicated hardware
- Soft Real Time
- ◊ performance concerns
- ◊ multimedia
- ◊ priority scheduling required
- ◊ low dispatch latency required!!
- ◊ kernel preemption points
- ◊ kernel preempt able

Algorithm Evaluation

- Earlier, we talked about criteria

◊ decide on relative importance of each criteria

- CPU utilization – keep CPU as busy as possible
- Throughput – # of jobs done per time unit
- Turnaround Time – Time of submission to Time of Completion
- Waiting Time – amount of time in ready queue
- Response Time – submit time to time of first output request

Algorithm Evaluation

- Deterministic Modeling

◊ take an example representative workload
  - a set of cpu burst times, usually more than one burst time for each process

◊ calculate each of the criteria for each of the algorithms (wait time, turn around time, etc.)

◊ in general, makes too many assumptions

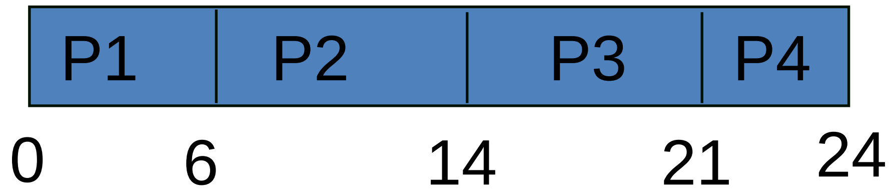
Algorithm Evaluation

- Deterministic Modeling

◊ Gantt charts

P1 - 6ms, P2 - 8ms, P3 - 7 ms, P4 - 3 ms

What is total & average waiting time with FIFO scheduling.



- Waiting time:

  - P1 - 0ms, P2 - 6ms, P3 - 14 ms, P4 - 21 ms = 41ms

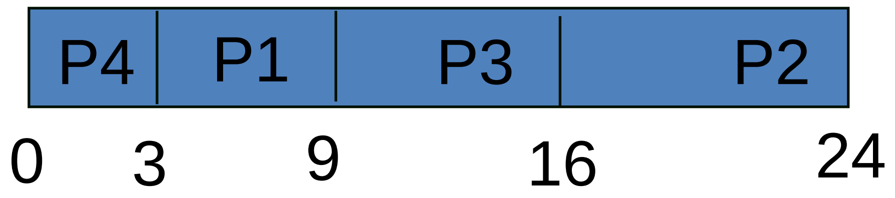
- average = 10.25ms

Algorithm Evaluation

- Same processes

P1 - 6ms, P2 - 8ms, P3 - 7 ms, P4 - 3 ms

What is total & average waiting time with SJF scheduling.



- Waiting time:

  - P1 - 3ms, P2 - 16 ms, P3 - 9 ms, P4 - 0 ms = 28ms
- average = 7ms

Algorithm Evaluation

- Simulation

◊ simulate all of the relevant parts of the system
◊ difficult to link various parts of the model
◊ trace tapes (generated from real systems)

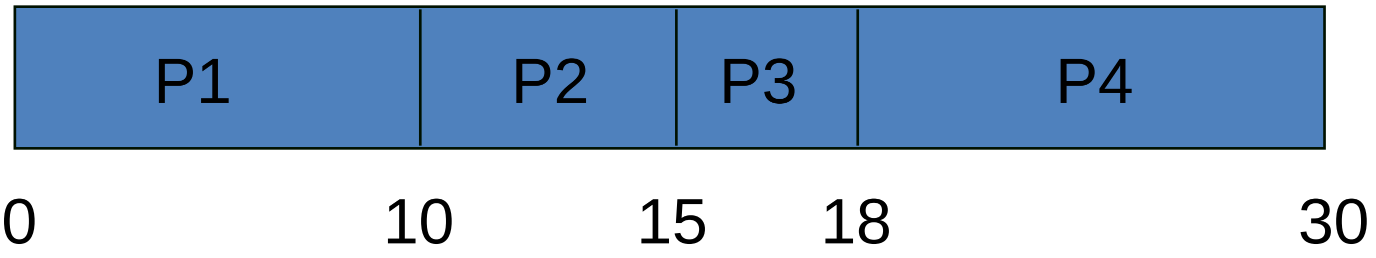
- Implementation

◊ try it and find out.
◊ expensive

Scheduling Examples

P1 - 10 ms, P2 - 5 ms, P3 - 3 ms, P4 - 12 ms

FIFO



Wait Times:

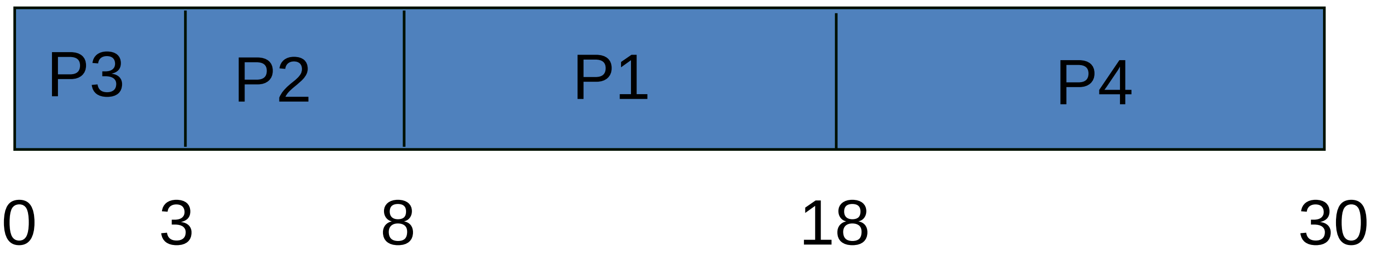
P1: 0 P2: 10 P3: 15 P4: 18

Total: 43 Average: 10.75

Scheduling Examples

P1 - 10 ms, P2 - 5 ms, P3 - 3 ms, P4 - 12 ms

SJF



Wait Times:

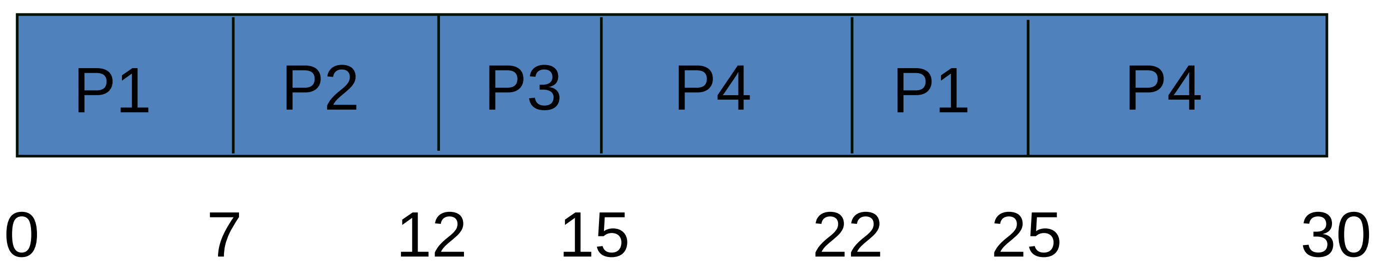
P1: 8 P2: 3 P3: 0 P4: 18

Total: 29 Average: 7.25

Scheduling Examples

P1 - 10 ms, P2 - 5 ms, P3 - 3 ms, P4 - 12 ms

RR, q=7ms, no context overhead



Wait Times:

P1: (22-7) = 15 P2: 7 P3: 12 P4: 15+(25-22)= 18

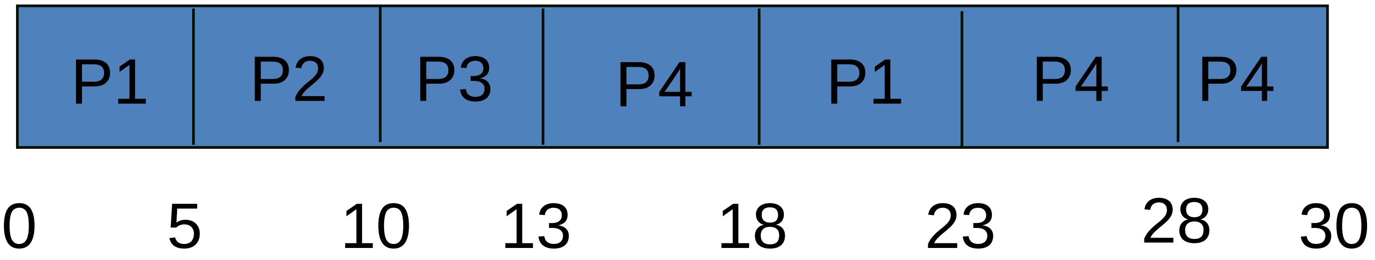
Total: 52 Average: 13

Turnaround for P2: 12 P3: 15

Scheduling Examples

P1 - 10 ms, P2 - 5 ms, P3 - 3 ms, P4 - 12 ms

RR, q=5ms, no context overhead



Wait Times:

P1: 13 P2: 5 P3: 10 P4: 18

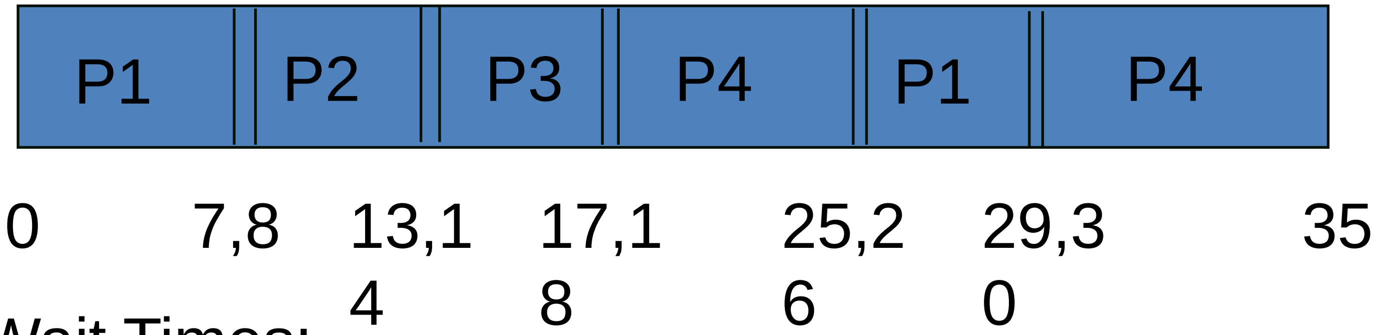
Total: 46 Average: 11.5

Turnaround for P2: 10 P3: 13

Scheduling Examples

P1 - 10 ms, P2 - 5 ms, P3 - 3 ms, P4 - 12 ms

RR, q=7ms, 1 ms context overhead



Wait Times:

P1: (26-7)=19 P2: 8 P3: 14 P4: 18+(30-25)=23

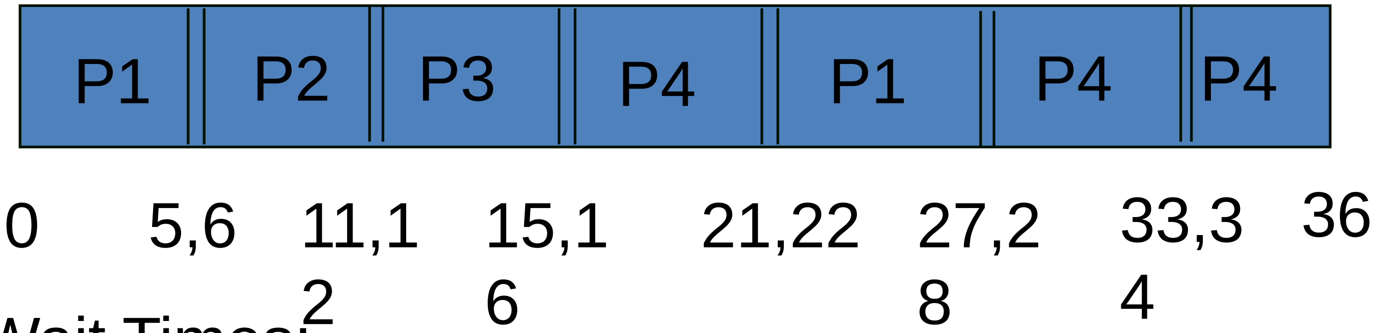
Total: 64 Average: 16

Turnaround for P2: 13 P3: 17

Scheduling Examples

P1 - 10 ms, P2 - 5 ms, P3 - 3 ms, P4 - 12 ms

RR, q=5ms, 1 ms context overhead



Wait Times:

P1: 16 P2: 6 P3: 12 P4: 24

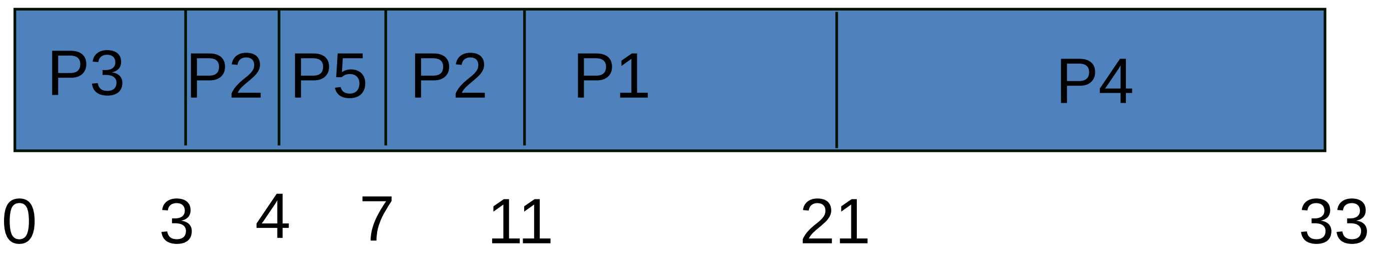
Total: 58 Average: 14.5

Turnaround for P2: 11 P3: 15

Scheduling Examples

P1 - 10 ms, P2 - 5 ms, P3 - 3 ms, P4 - 12 ms

SRTF, interrupt at time 4, P5 - 3 ms



Wait Times:

P1: 11 P2: 6 P3: 0 P4: 21

P5: 0

Total: 38 Average: 7.6