COMP 4735: Operating Systems Concepts

Lesson 9: Scheduling

Rob Neilson

rneilson@bcit.ca

Administrivia

- As per the schedule posted on webct ...

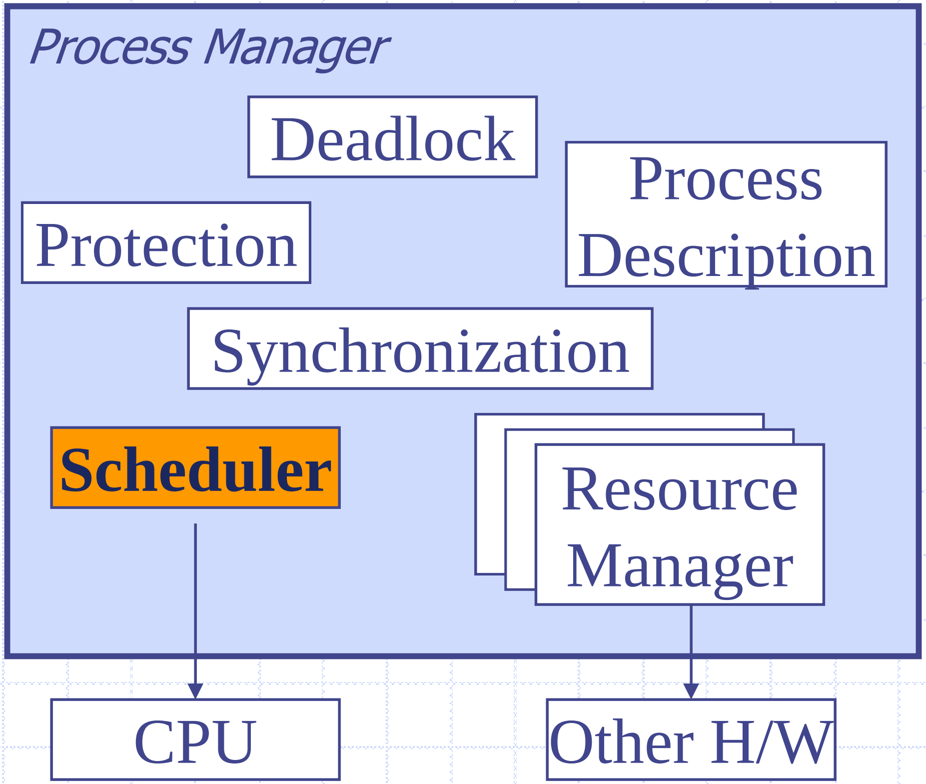
Week	Date	Lesson and Lab Focus	Textbook	Weekly Topic
. . .				
7	Lab 6:	Scheduling Questions from Textbook	2.4	Scheduling
	Feb 16	Quiz 5: Chapters 2.3	2.3	
	Feb 18	Scheduling (Introduction / Principles / Batch Algorithms)	2.4	
	Feb 19	Scheduling Algorithms (part 1)	2.4	
8	Lab 7	*No Labs This Week (except Set F)*		Scheduling
	Feb 23	Scheduling Algorithms (part 2)	2.4	
	Feb 25	*No Classes - Staff PD Day*		
	Feb 26	*Classes cancelled this day*		
9	Lab 8	Memory Management Lab (general questions)	3.1, 3.2	Memory Management
	Mar 2	Quiz 6: Chapters 2.4	2.4	
	Mar 4	Memory (swapping)	3.1, 3.2	
	Mar 5	Memory (virtual memory)	3.3	
. . .				

Introduction

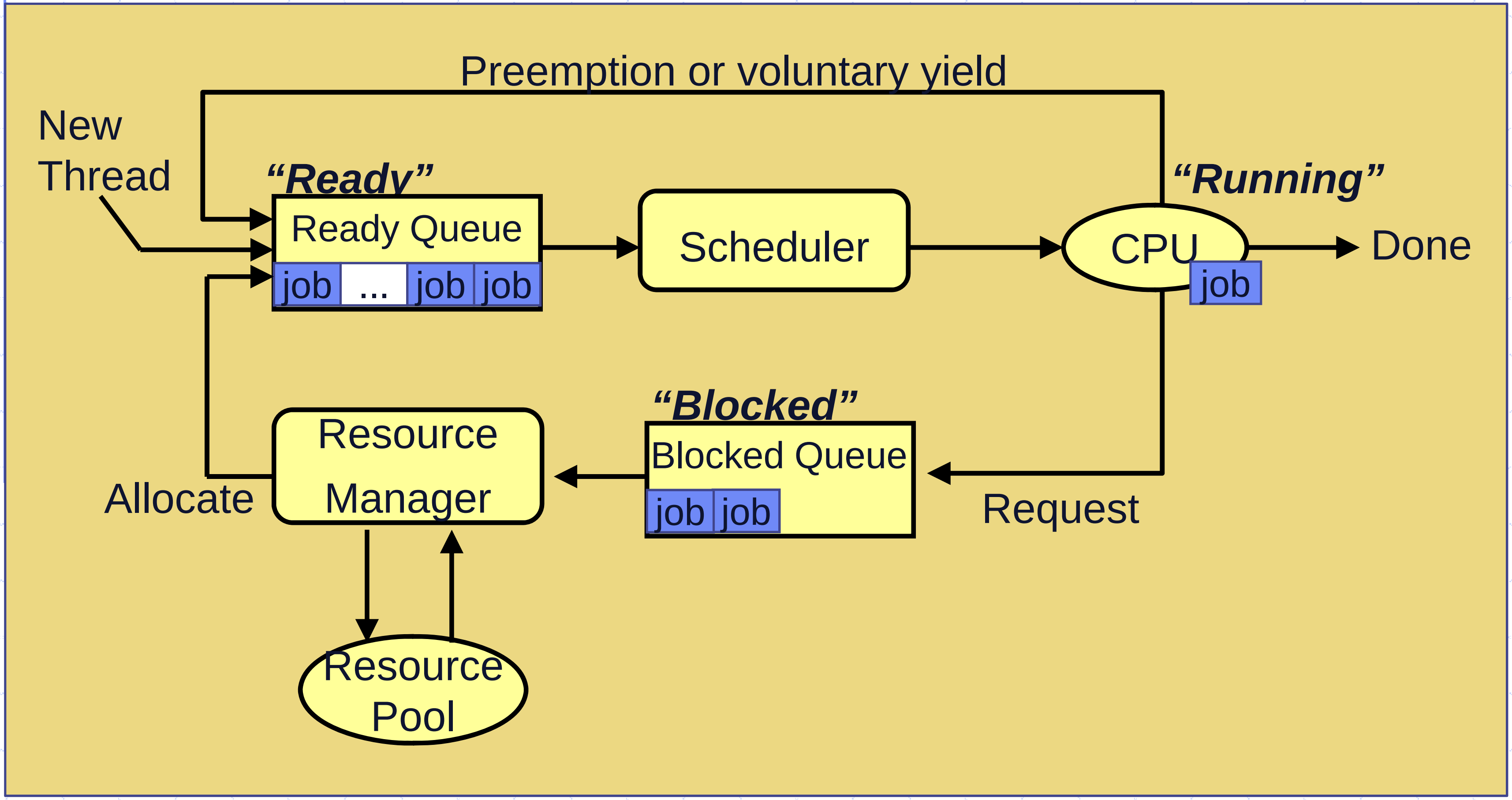
- so far we have discussed processes, threads, synchronization, context switching
- all of this depends on some way to schedule the threads or processes

- scheduler is the part of the OS that determines which process or thread gets the cpu next

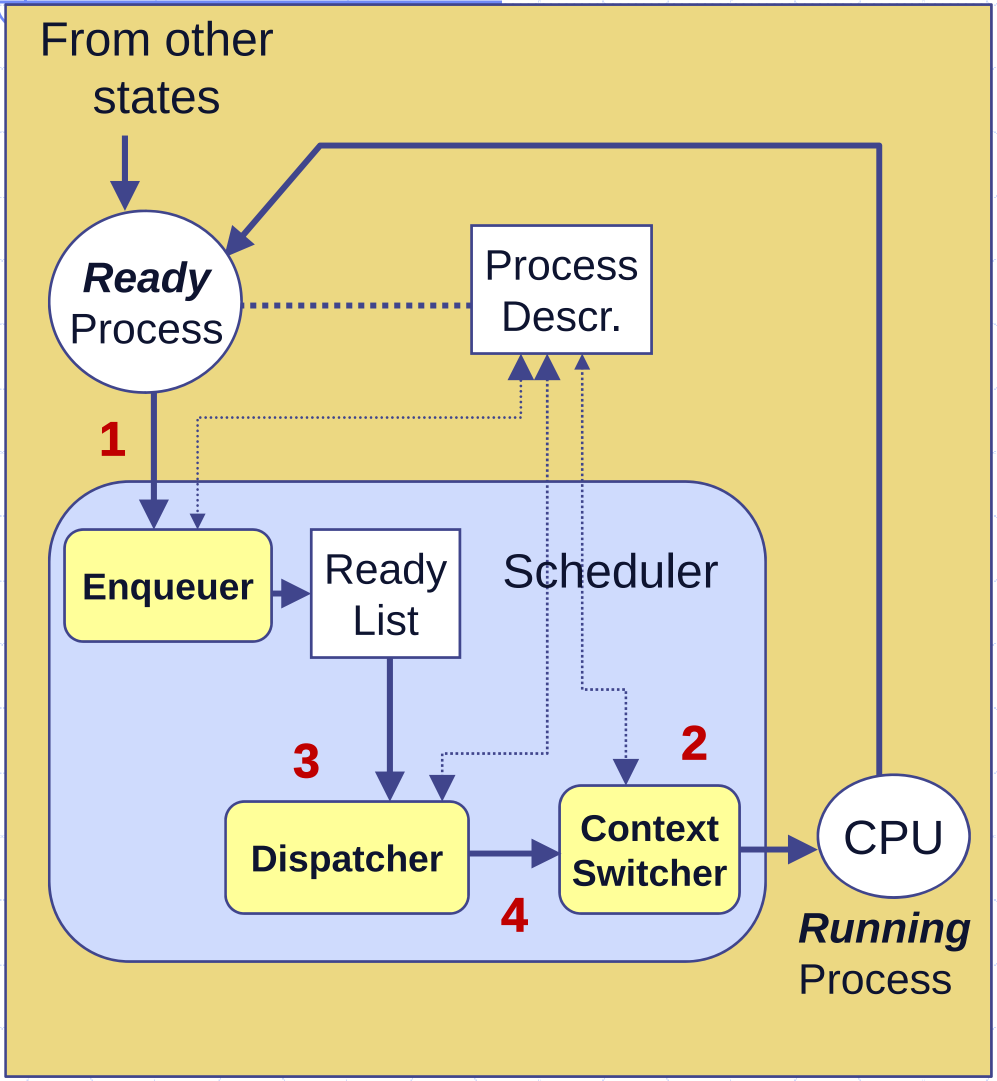
- scheduler is also responsible for doing the context switch



Model of Thread Execution



Scheduler Organization (logical)



1. when state changes to ready, enQ puts ptr to the descriptor in the ready list
2. context switcher runs when we need to remove running process from the CPU
   - saves registers (PC, IR etc) for the process that is being removed)
3. dispatcher selects next process from ready queue
   - requires dispatcher context to be loaded on CPU
4. lastly, dispatcher performs a context switch from itself to the newly selected process

Note: all changes to a processes state are saved in descriptor

Context Switch: How expensive is it?

Very expensive! For example ...

- assume ...
  - a `mov reg,mem` instruction takes k=50ns
  - n=64 general purpose registers, m=8 status registers
    - $(n + m) \times k =$ ***3.6 microseconds***
- need to do this *four times* to start a new process
  1. save the running process registers
  2. load dispatcher registers
     - now dispatcher runs and selects next process to b loaded
  3. save dispatcher registers
  4. load new process registers

Invoking the Scheduler

- Need a *mechanism* to call the scheduler ...

1. Voluntary call (non-preemptive)
   - Process gives up CPU itself (yield)
   - Process blocks on IO or waits on a resource (eg: semaphore)
   - Process terminates

2. Involuntary call (preemptive)
   - OS takes the CPU away from process (interrupt / preemption)
   - Another process forces the active process to terminate / wait

Voluntary CPU Sharing

- use the yield system call to give up CPU
- yield:
  - save the address of the next instruction (PC)
  - branch to a different location in memory (scheduler entry point)
- note:
  - scheduler entry point is stored at a predetermined place in memory
  - current PC will be stored at a predetermined place in memory
  - these memory locations (r, s) are determined relative to the process descriptors of the processes being saved/loaded

```
yield(r, s) {
    memory[r] = PC;    /* save PC to r */
    PC = memory[s];    /* load new PC from s */
}
```

Yield – Cooperating Processes

- when a process p1 yields to another process p2, the new process (p2) knows the p1's next instruction, so it can return control to p1
  - for example:

```
yield(*, p2.id);   /* p1 yields to p2   */

  . . .            /* p2 does stuff... */
yield(*, p1.id);   /* p2 yields to p1   */

  . . .            /* p1 does stuff... */
yield(*, p2.id);   /* p1 yields to p2   */

  . . .            /* p2 does stuff... */
```

- *how does a process know the PC of the process that yielded to it?*
- *what happens to the other registers, stack, etc of the yielding process?*

Yield – With a Scheduler

Suppose p_{sched} is the scheduler:

```
. . .                       /* p_i is running          */
yield(*, psched.pc);        /* p_i yields to scheduler */

. . .                       /* scheduler chooses p_k   */
yield(*, pk.pc);            /* scheduler yields to p_k */

. . .                       /* p_k is running          */
yield(*, psched.pc);        /* p_k yields to scheduler */
```

- So in this case we can implement a scheduler as a daemon or service, and processes can just yield to it at appropriate times
- Alternatively, this could be a thread's run-time system that we are yielding to

Involuntary CPU Sharing

- This is the typical "preemptive scheduling"
- Makes use of an *interval timer*
  - Device to produce a periodic interrupt; has a programmable period
  - CPUs typically come with support for a number of timers

- *logical* behaviour of hw timer
- after K ticks an IRQ is set
  - interrupt handler runs
  - loads device handler for interval timer
  - device handler calls the scheduler
  - scheduler does the dispatch / context switch

```
IntervalTimer() {
    InterruptCount--;
    if(InterruptCount <= 0) {
        InterruptRequest = TRUE;
        InterruptCount = K;
    }
}
SetInterval(programmableValue) {
    K = programmableValue:
    InterruptCount = K;
    }
}
```

Involuntary CPU Sharing (cont)

- Interval timer device handler
  - Keeps an in-memory clock up-to-date
  - Invokes the scheduler

```
IntervalTimerHandler() {
    Time++;               /* update the clock */
    TimeToSchedule--;
    if(TimeToSchedule <= 0) {

        <invoke scheduler>;

        TimeToSchedule = TimeSlice;
    }
}
```

Review - what have we covered so far …

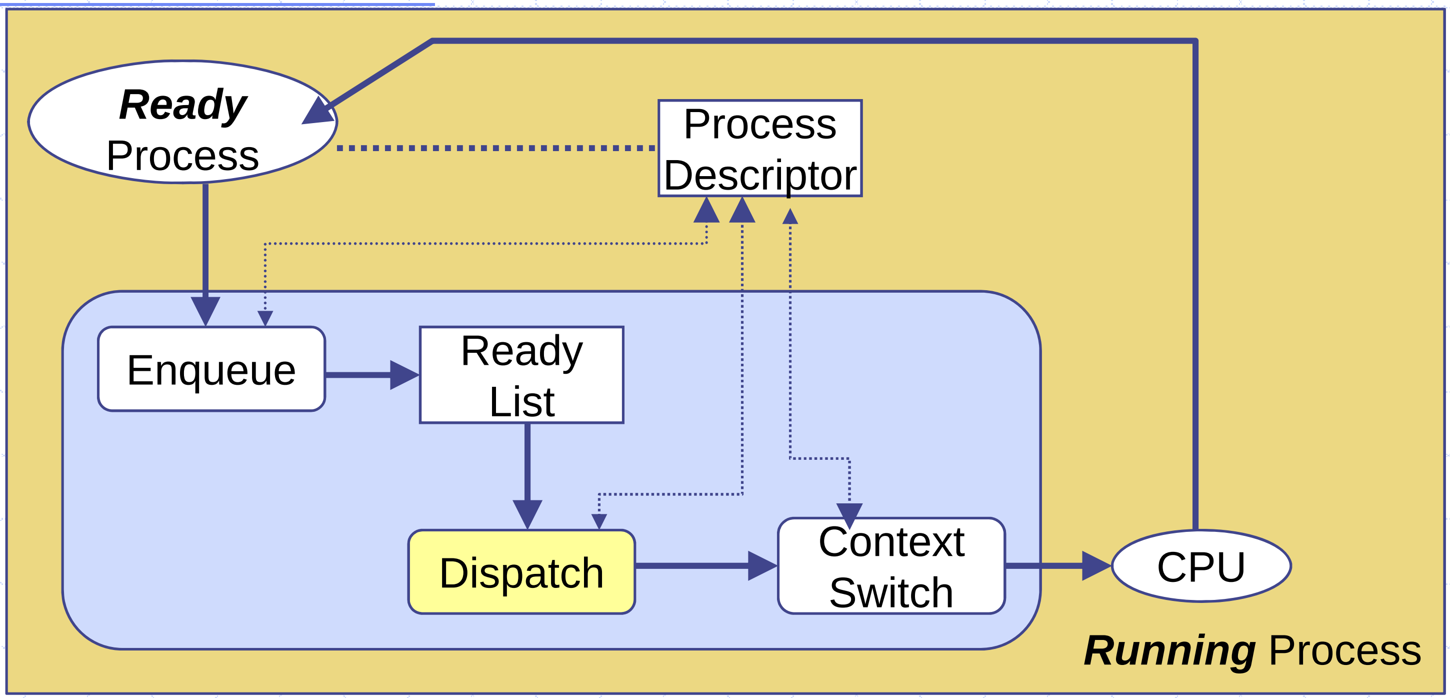
Key Points:

- scheduler is made up (logically) of an enqueuer, dispatcher, context switcher, and a ready list

- the main goals of scheduler design are to minimize context switch time (wasted cycles), and to ensure fair allocation of the CPU

- there are two methods of sharing the CPU:
  - voluntary (non-preemptive)
    - processes/threads use a yield system call to give up CPU, or
    - processes run until complete or they block
  - involuntary (preemptive)
    - processes/threads are "preempted" by device handler code that runs at regular intervals (when a system timer fires)

Mechanism vs Policy



- *Mechanism* never changes
  - the above diagram shows the scheduling mechanism
- *Policy* used by dispatcher to select a process from the ready list
  - Different policies for different requirements
    - eg: batch vs interactive vs real-time
- Policy is implemented as the *Scheduling Algorithm*

Policy Considerations

- Scheduling Policy (which process runs next) can control/influence:
  - CPU utilization
  - Average time a process waits for service
  - Average amount of time to complete a job
- Could strive for any of:
  - Equitability
  - Favor very short or long jobs
  - Meet priority requirements
  - Meet deadlines

A Model to Study Scheduling

Define:

- $P = \{p_i \mid 0 \le i < n\}$ // set of all jobs
- $S(p_i) \in \{\text{running, ready, blocked}\}$ // state of a job

$\tau(p_{ij})$: amount of time a thread is in running state before completes

- called the ***Service Time***; ie: total CPU time used

$W(p_{ij})$: amount of time a thread waits before it first starts running

- called the ***Wait Time***; ie: delay before the thread gets the CPU

$T_{TRnd}(p_{ij})$: amount of time between first ready state and last running

- called the ***Turnaround Time*** : ie: total time from start to finish

- these are base measurements that can be made
- these *measurements* are used to compute and compare various performance *metrics* (eg: response time, throughput rate etc)

Optimal Schedule

- If we know the service time for all Pij, and we have a scheduling goal (criteria for optimality), we can devise an optimal schedule
  - an optimal schedule is one that minimizes (or maximizes) the optimality criteria
- To find an *optimal schedule*:
  - Have a finite, fixed # of p_i
  - Know $\tau(p_i)$ for each p_i
  - Enumerate all possible schedules, then choose the best
- *Do you foresee any difficulties in doing this?*

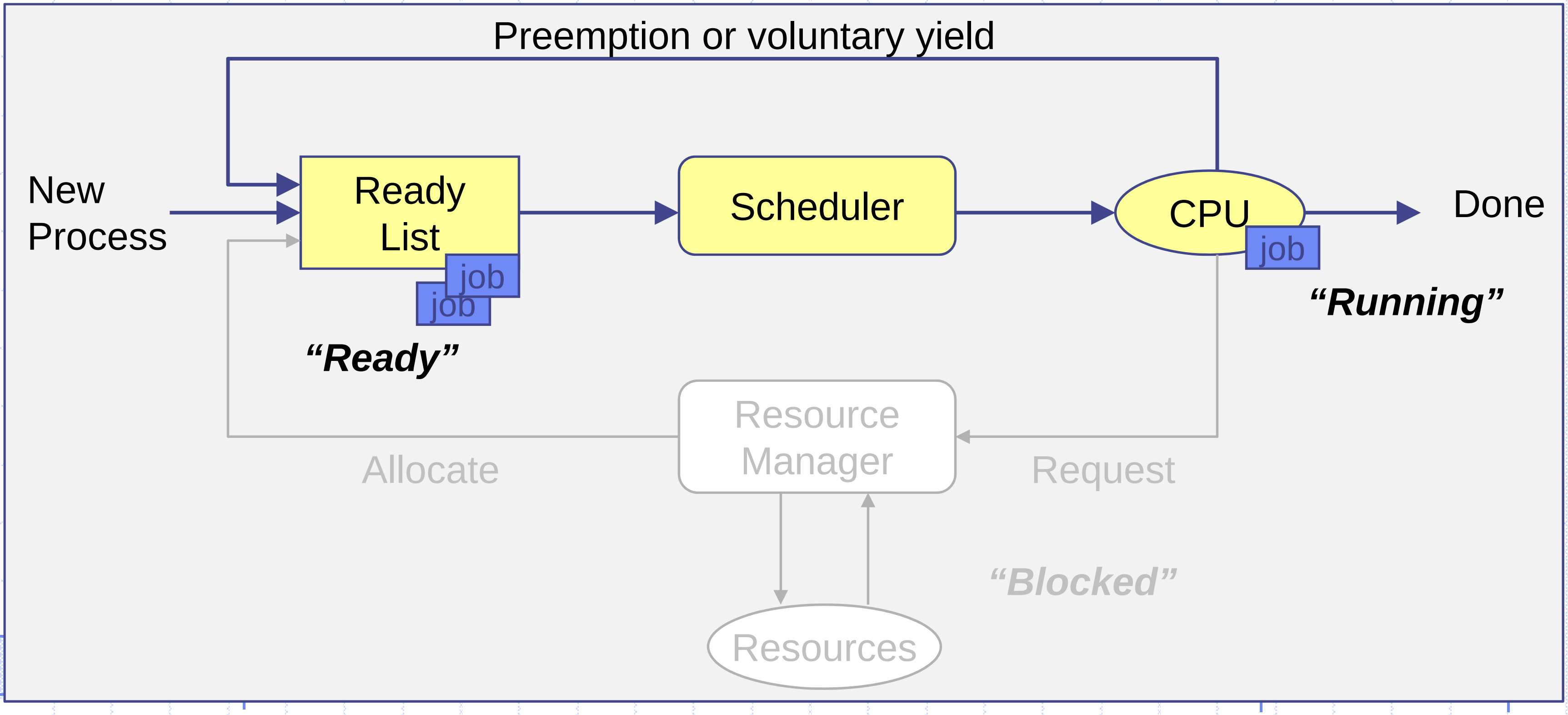
Challenges in determining optimal scheduling ...

1. The $\tau(p_i)$ are almost certainly just estimates
2. General algorithm to enumerate schedules is $O(n^2)$
3. Other processes may arrive while these processes are being serviced (ie: it is an online problem)

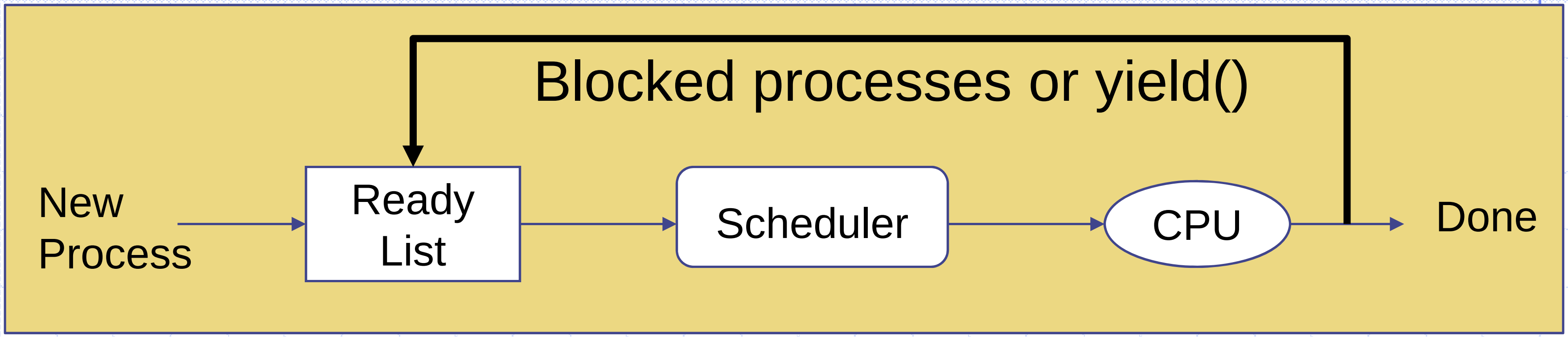
- *Usually, the optimal schedule is only a <u>theoretical benchmark</u> – scheduling policies try to <u>approximate</u> an optimal schedule*

Model of Process Execution (revisited)

- generally, scheduling strategy *ignores the time a thread spends blocked on IO*
- the model will assume "no IO"
- note: some strategies inadvertently consider IO time, for example:
  - a strategy that seeks to optimize turnaround time will favor threads that have spent a lot of time blocked on IO



Non-preemptive Schedulers



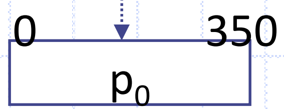
- Typically used for Batch systems
- Try to use the simplified scheduling model
- Only consider running and ready states
- Ignore time in blocked state:
  - "New process created when it enters ready state"
  - "Process is destroyed when it enters blocked state"
  - Really just looking at "small phases" of a process

First-Come-First-Served (1)

Idea: just process them in the order they are placed in ready queue.

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75

Assume all 5 jobs arrived at time 0
Added to ready queue in order listed



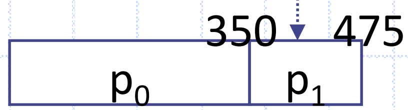
$T_{TRnd}(p_0) = t(p_0) = 350$

$W(p_0) = 0$

First-Come-First-Served (2)

Assume all 5 jobs arrived at time 0
Added to ready queue in order listed

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_0) = t(p_0) = 350$

$T_{TRnd}(p_1) = (t(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$

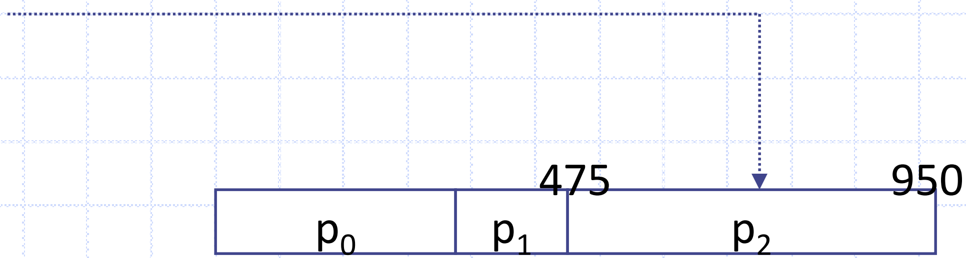
$W(p_0) = 0$

$W(p_1) = T_{TRnd}(p_0) = 350$

First-Come-First-Served (3)

Assume all 5 jobs arrived at time 0
Added to ready queue in order listed

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_0) = t(p_0) = 350$

$T_{TRnd}(p_1) = (t(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$

$T_{TRnd}(p_2) = (t(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$

$W(p_0) = 0$

$W(p_1) = T_{TRnd}(p_0) = 350$

$W(p_2) = T_{TRnd}(p_1) = 475$

First-Come-First-Served (4)

Assume all 5 jobs arrived at time 0
Added to ready queue in order listed

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75

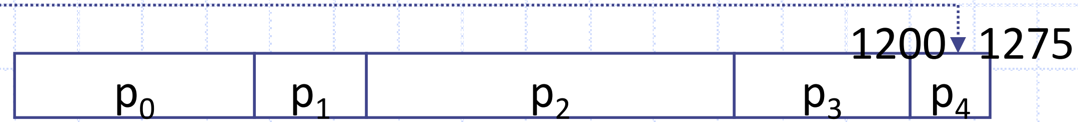
$T_{TRnd}(p_0) = t(p_0) = 350$
$T_{TRnd}(p_1) = (t(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (t(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$
$T_{TRnd}(p_3) = (t(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$
$W(p_3) = T_{TRnd}(p_2) = 950$

First-Come-First-Served (5)

Assume all 5 jobs arrived at time 0
Added to ready queue in order listed

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75

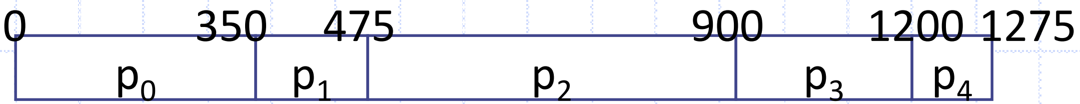


$T_{TRnd}(p_0) = t(p_0) = 350$
$T_{TRnd}(p_1) = (t(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (t(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$
$T_{TRnd}(p_3) = (t(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$
$T_{TRnd}(p_4) = (t(p_4) + T_{TRnd}(p_3)) = 75+1200 = 1275$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$
$W(p_3) = T_{TRnd}(p_2) = 950$
$W(p_4) = T_{TRnd}(p_3) = 1200$

FCFS Average Turnaround Time

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_0) = t(p_0) = 350$
$T_{TRnd}(p_1) = (t(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (t(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$
$T_{TRnd}(p_3) = (t(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$
$T_{TRnd}(p_4) = (t(p_4) + T_{TRnd}(p_3)) = 75+1200 = 1275$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$
$W(p_3) = T_{TRnd}(p_2) = 950$
$W(p_4) = T_{TRnd}(p_3) = 1200$

$Trnd_{avg} = (350+475+950+1200+1275)/5 = 4250/5 = 850$

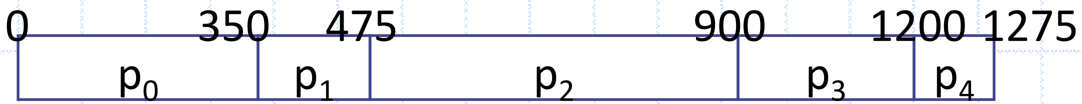
FCFS Average Wait Time

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75

Easy to implement

Ignores service time, etc

Not a great performer



$T_{TRnd}(p_0) = t(p_0) = 350$

$T_{TRnd}(p_1) = (t(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$

$T_{TRnd}(p_2) = (t(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$

$T_{TRnd}(p_3) = (t(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$

$T_{TRnd}(p_4) = (t(p_4) + T_{TRnd}(p_3)) = 75+1200 = 1275$

$W(p_0) = 0$

$W(p_1) = T_{TRnd}(p_0) = 350$

$W(p_2) = T_{TRnd}(p_1) = 475$

$W(p_3) = T_{TRnd}(p_2) = 950$

$W(p_4) = T_{TRnd}(p_3) = 1200$

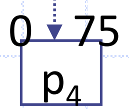
$\mathbf{W_{avg} = (0+350+475+950+1200)/5 = 2975/5 = 595}$

Shortest Job Next (1)

Idea: always choose the job that has shortest service time to run next.

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75

Assume all 5 jobs arrived at time 0
Added to ready queue in order listed

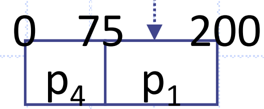


$T_{TRnd}(p_4) = t(p_4) = 75$

$W(p_4) = 0$

Shortest Job Next (2)

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_1) = t(p_1)+t(p_4) = 125+75 = 200$

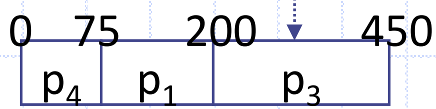
$W(p_1) = 75$

$T_{TRnd}(p_4) = t(p_4) = 75$

$W(p_4) = 0$

Shortest Job Next (3)

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75



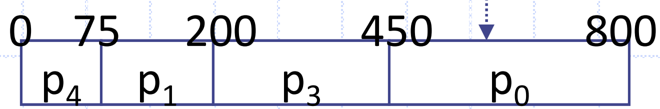
$T_{TRnd}(p_1) = t(p_1)+t(p_4) = 125+75 = 200$ $\quad W(p_1) = 75$

$T_{TRnd}(p_3) = t(p_3)+t(p_1)+t(p_4) = 250+125+75 = 450$ $\quad W(p_3) = 200$

$T_{TRnd}(p_4) = t(p_4) = 75$ $\quad W(p_4) = 0$

Shortest Job Next (4)

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_0) = t(p_0)+t(p_3)+t(p_1)+t(p_4) = 350+250+125+75 = 800$ $W(p_0) = 450$

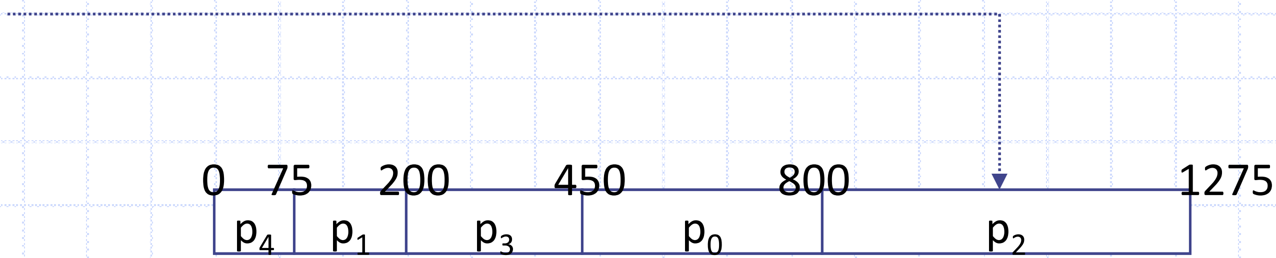
$T_{TRnd}(p_1) = t(p_1)+t(p_4) = 125+75 = 200$ $W(p_1) = 75$

$T_{TRnd}(p_3) = t(p_3)+t(p_1)+t(p_4) = 250+125+75 = 450$ $W(p_3) = 200$

$T_{TRnd}(p_4) = t(p_4) = 75$ $W(p_4) = 0$

Shortest Job Next (5)

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_0) = t(p_0)+t(p_3)+t(p_1)+t(p_4) = 350+250+125+75 = 800$ $W(p_0) = 450$

$T_{TRnd}(p_1) = t(p_1)+t(p_4) = 125+75 = 200$ $W(p_1) = 75$

$T_{TRnd}(p_2) = t(p_2)+t(p_0)+t(p_3)+t(p_1)+t(p_4) = 475+350+250+125+75$ $W(p_2) = 800$
$= 1275$

$T_{TRnd}(p_3) = t(p_3)+t(p_1)+t(p_4) = 250+125+75 = 450$ $W(p_3) = 200$

$T_{TRnd}(p_4) = t(p_4) = 75$ $W(p_4) = 0$

Shortest Job Next (Avg Trnd Time)

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75

0	75	200	450	800	1275
p_4	p_1	p_3	p_0	p_2	

$T_{TRnd}(p_0) = t(p_0)+t(p_3)+t(p_1)+t(p_4) = 350+250+125+75 = 800$ $\quad W(p_0) = 450$

$T_{TRnd}(p_1) = t(p_1)+t(p_4) = 125+75 = 200$ $\quad W(p_1) = 75$

$T_{TRnd}(p_2) = t(p_2)+t(p_0)+t(p_3)+t(p_1)+t(p_4) = 475+350+250+125+75$ $\quad W(p_2) = 800$
$= 1275$

$T_{TRnd}(p_3) = t(p_3)+t(p_1)+t(p_4) = 250+125+75 = 450$ $\quad W(p_3) = 200$

$T_{TRnd}(p_4) = t(p_4) = 75$ $\quad W(p_4) = 0$

$Trnd_{avg}$ = (800+200+1275+450+75)/5 = 2800/5 = 560

Shortest Job Next (Average Wait Time)

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75

Minimizes avg wait time

May starve large jobs

Must know service times

0 – 75	75 – 200	200 – 450	450 – 800	800 – 1275
p_4	p_1	p_3	p_0	p_2

$T_{TRnd}(p_0) = t(p_0)+t(p_3)+t(p_1)+t(p_4) = 350+250+125+75 = 800$ $\quad W(p_0) = 450$

$T_{TRnd}(p_1) = t(p_1)+t(p_4) = 125+75 = 200$ $\quad W(p_1) = 75$

$T_{TRnd}(p_2) = t(p_2)+t(p_0)+t(p_3)+t(p_1)+t(p_4) = 475+350+250+125+75$
$= 1275$ $\quad W(p_2) = 800$

$T_{TRnd}(p_3) = t(p_3)+t(p_1)+t(p_4) = 250+125+75 = 450$ $\quad W(p_3) = 200$

$T_{TRnd}(p_4) = t(p_4) = 75$ $\quad W(p_4) = 0$

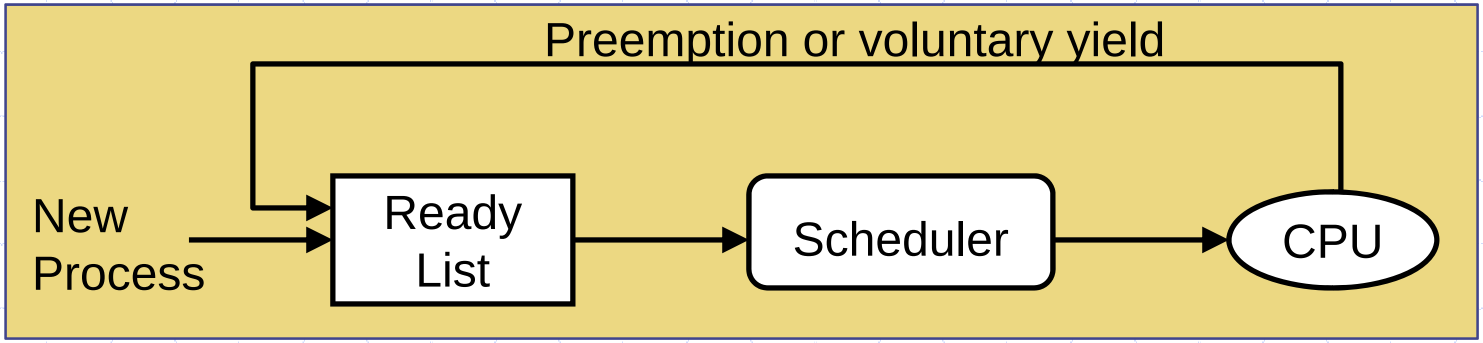
$\mathbf{W_{avg} = (450+75+800+200+0)/5 = 1525/5 = 305}$

Review of last few topics ...

Key Points:

- a model to study scheduling relies on:
  - service time: the total cpu time used by a process or thread
  - arrival time: the time that a job is placed in the ready queue
- The model allows us to measure:
  - wait time: the time a process or thread waits before first being scheduled
  - turn-around time: the total time the process or thread exists (ie: from when it first becomes ready, to when it is done)
- if we know all the service times $\tau(p_i)$ in advance, we can compute an optimal schedule
  - this is not practical because we don't know the times, and we don't know them in advance, so we approximate
- some *common non-preemptive scheduling algorithms* include
  - first come first served
  - shortest job next

Preemptive Schedulers (1)



- Scheduler is executed ...
  - each time a *thread changes to READY state*
  - each time the *interval timer expires*
  - any time a *thread gives up the CPU*
    - process terminates
    - process blocks on wait() or IO

- Scheduler applies its policy to select the highest priority thread
  - highest priority thread will take over the CPU

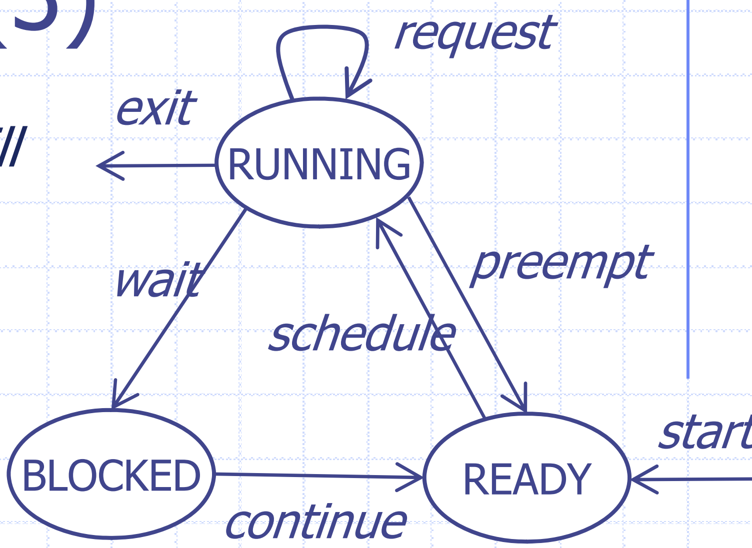
Preemptive Schedulers (2)

- the non-preemptive algorithms (FCFS, SJN etc) can all be run in a preemptive manner
  - SJN would compare the *remaining service time* of each process to select the next one to run
    - note: only needs to compare newly arrived processes as shortest one is already on the CPU
  - *What should FCFS Compare?*

FCFS:

- should compare thread start time
- it is possible that the earliest job was blocked on IO and is now finished so it becomes available
- an interval timer would not make sense with preemptive FCFS

Preemptive Schedulers (3)



Question: Can you list all the events that will cause the scheduler to run?

... need to look at the state machine ...

Case 1: interval timer fires

- this is **event 1:** interval timer fires (preempt event)

Case 2: thread changes to ready state

- what can cause this?

... two possible events ...

event 2: new thread or process is created (start event)

event 3: thread was blocked and is now able to run again (continue event)

Case 3: thread yields the CPU

event 4: the thread needs a resource, changes to blocked state (wait event)

event 5: thread is done; exit; (exit event)

Event 1: Interval timer fires ...

Let's look at what happens in detail ...

- a thread is happily running on CPU
- time slice expires, so CPU receives an interrupt
- interrupt handler code runs
  - executes the enqueuer code
    - change process state from ***running*** to ***ready***, add to ready queue
  - executes context switch code
    - save state of current process in process descriptor
  - determines that interval timer caused the interrupt
    - loads the context for the dispatcher
    - jumps to an entry point in the scheduler code (calls the dispatcher)
- dispatcher selects next job P_n according to scheduling policy
- dispatcher runs context switch code (again)
    - save state of dispatcher
    - change process state of P_n from ***ready*** to ***running***
    - load context for next job P_n

Event 2: New Process/thread created by parent

Let's look at what happens in detail ...

- a thread is happily running on CPU
- the thread executes CreateProcess()
- thread traps to kernel and runs createprocess system call
  - system call creates descriptors for the new process
  - initializes default descriptor values
  - copy values / resource handles etc from current context as requested
  - ***set new process state to ready, add to ready queue***
  - system call jumps to an entry point in the scheduler (call the scheduler)
- scheduler runs the enqueuer code
  - change current process state from running to ready, add to ready queue
- scheduler runs context switch code
  - save state of current process in process descriptor
  - load dispatcher context
- scheduler runs the dispatcher
  - select next job P_n according to scheduling policy
- scheduler runs context switch code (again)
  - save state of dispatcher
  - change process state of P_n from ready to running
  - load context for next job P_n

Event 3: Resources are allocated to a blocked thread

Let's look at what happens in detail ...

... but this time you work through it ...

... a thread Pij is happily running on CPU ...

... another thread Pmn is waiting on Pij ...

... what happens next?

(you have 1 minute – so work fast!)

Algorithm: Round Robin Scheduling

Goal of RR

- equitable sharing or CPU between ready threads

General idea

- each thread uses the CPU for one time slice, then goes to the end or ready queue

For example, 3 threads (P1, P2, P3) with a *time slice of 50 ...*

t=0 50 100 150 200 250 300 310 360 410 425 475...

| P1 | P2 | P3 | P1 | P2 | P3 | P1 | P2 | P3 | P2 | P3 | P3 | P3 | P3 . . .

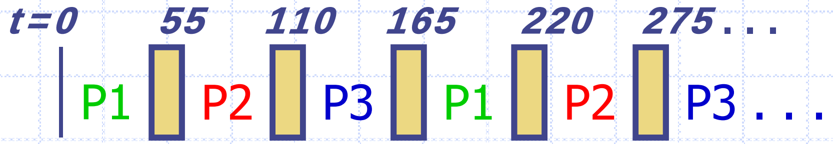
assume that P1 finishes in this time slice, after using only 10 time units

assume that P2 finishes in this time slice, after using only 15 time units

RR – Time Slice Overhead

Time slice overhead

- should factor in the overhead of doing the context switch
- for example, assume time slice=50, overhead=5



RR – Ready Queue Data Structure

- need some way to store the ready threads
- there are a variety of approaches which all have different queuing semantics

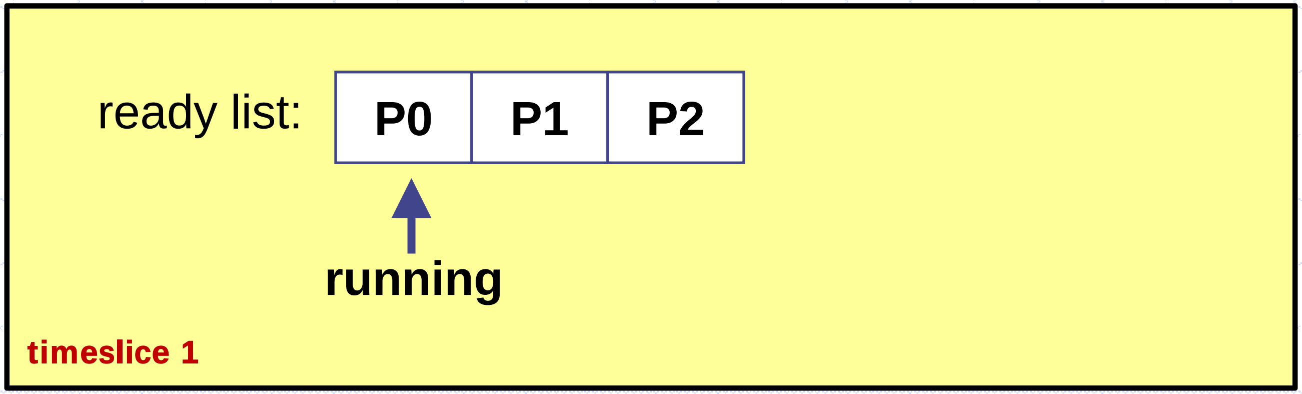
The approach we will use is exactly as given in the textbook:

1. the ready queue is a FIFO queue
2. the currently running process is at the front of the queue
3. new processes are added at the back of the queue
4. preempted processes are moved from the front to the back of the queue
5. if a process arrives at the same time a process is preempted, the new arrival is placed on the queue first

Example of Ready Queue (1)

Consider the following example:

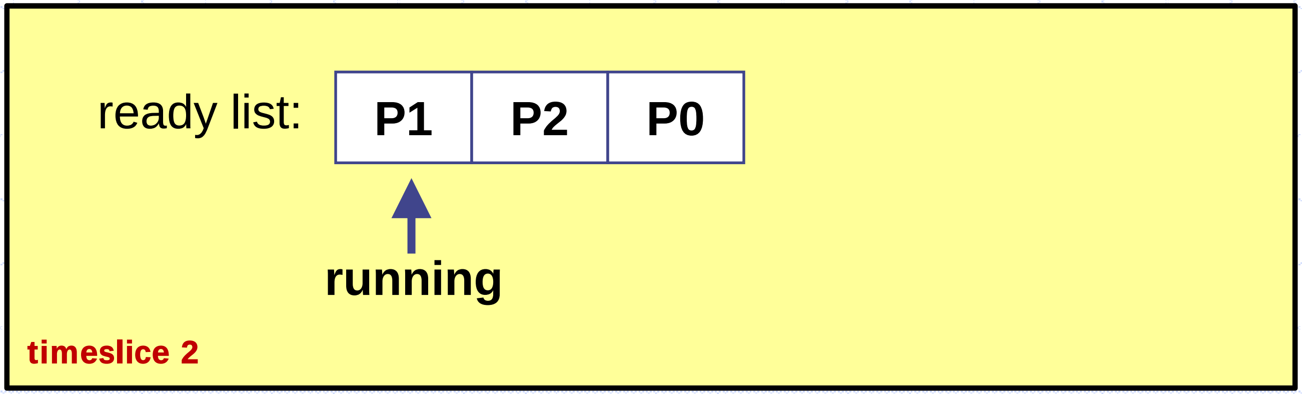
- P0,P1,P2 are enqueued at start
- P0 is at the front and is running in ts 1



Example of Ready Queue (2)

Consider the following example:

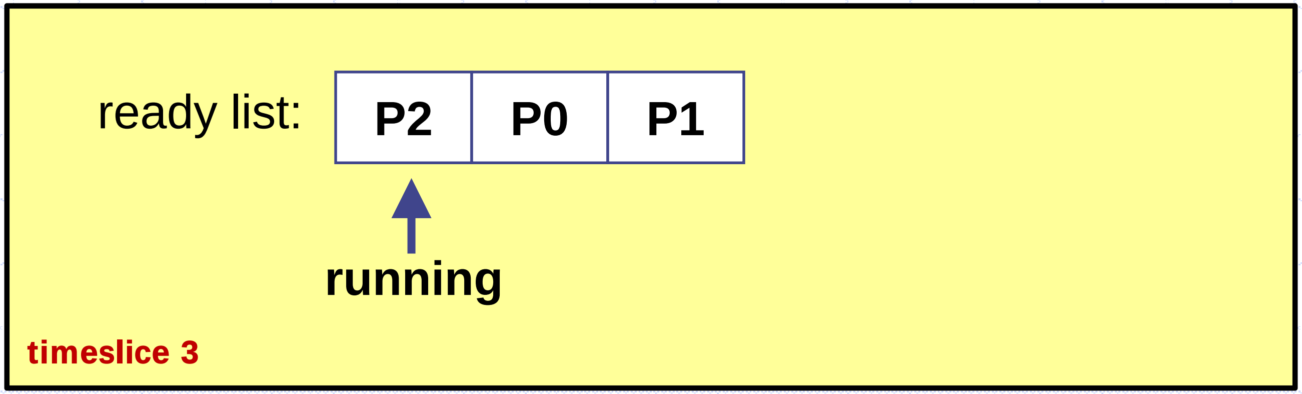
- interval timer fires at end of ts 1
- P0 goes to back of queue
- P1 is running in ts 2



Example of Ready Queue (3)

Consider the following example:

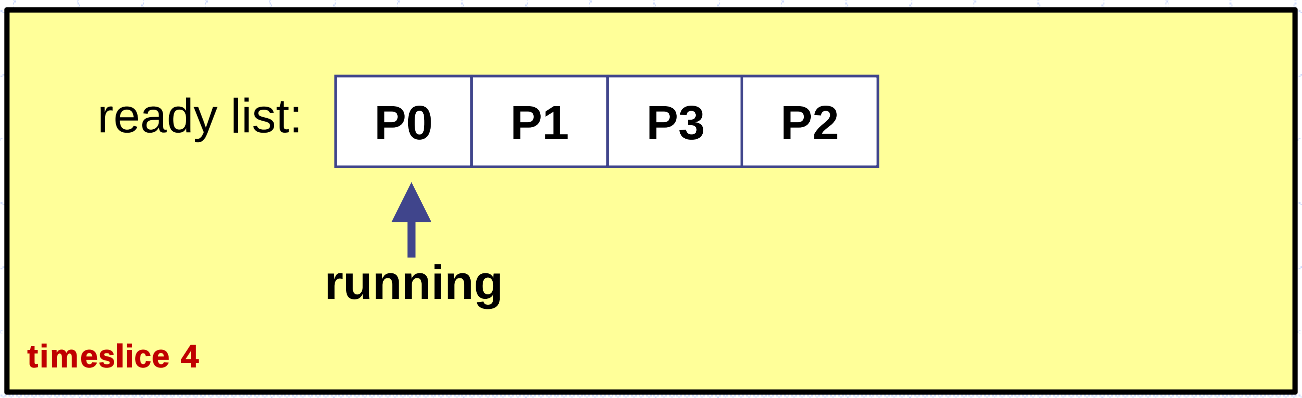
- interval timer fires at end of ts 2
- P1 goes to back of queue
- P2 is running in ts 3



Example of Ready Queue (4)

Consider the following example:

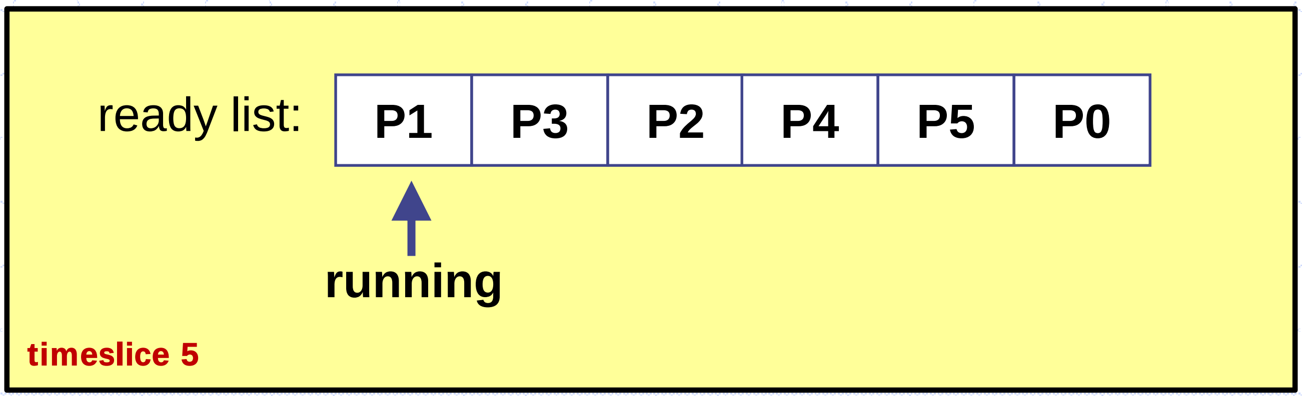
- interval timer fires at end of ts 3
- P3 arrives at end of ts 3
- P3 is put on end of queue, then
- P2 is moved to back of queue
- P0 is running in ts 4



Example of Ready Queue (5)

Consider the following example:

- interval timer fires at end of ts 4
- P4 and P5 arrive at end of ts 4
- P4 and P5 are put on end of queue, then
- P0 is moved to back of queue
- P1 is running in ts 5



Round Robin Example 1

- time slice = 25, overhead is ignored
- all threads already exist in queue
- 4 threads with service times are P0=60, P1=30, P2=80, P3=50
- we want to:
  - draw a gantt chart showing cpu usage
  - calculate average wait time
  - calculate average turn-around time

Solution to example 1

- example done on overheads

Round Robin Example 2

- same time slice and processes as example 1
- add variable arrival times, as follows:

i	$t(p_{ij})$	Arrival
0	60	0
1	30	10
2	80	60
3	50	100

Solution to example 2

- example done on overheads

Multi-level Queues

- essentially we have a number of ready lists
- the ready lists are "priority groups"
- new jobs are assigned to a group based on initial priority
- scheduling within a group is performed using a fair algorithm (eg (RR)

Multi-level Queues (2)

- we need a way to assign new jobs to a group
  - typically this is done based on priority
  - eg: BSD Unix
    - 32 queues
      - queues 0-7 are for OS processes
      - queues 8-31 are for user space processes

Multi-level Queues (3)

- once jobs (processes) are in the queues, we need to select which job to execute

  - BSD approach:
    - always select next job from highest priority non-empty queue
    - use RR in the queue until it is empty, then move to next queue

    - this is a reasonable approach, but could lead to starvation on a heavily loaded system

Multi-level Example 1

- assume you have a multi-level queue with 3 levels
- the *BSD scheduling model* is used
- assume the *time quantum is 10*, and ignore overhead
- draw a gantt chart showing order of execution for the following processes:

i	**t**(p_{ij})	Arrival	Priority
0	60	0	3
1	30	10	2
2	40	60	1
3	30	75	4
4	20	85	3
5	10	90	1

ML Example 1 Solution

- done on overheads

Multi-level Example 2

- assume you have a multi-level queue with 2 levels
- assume that level 1 gets four tq's for every 1 tq in level 2
- assume the time quantum is 10, and ignore overhead
- round robin is used for selecting processes within each level
- draw a gantt chart showing order of execution for the following processes:

i	$t(p_{ij})$	Arrival	Priority
0	40	0	2
1	30	10	2
2	70	30	1
3	30	30	1
4	20	60	1
5	10	100	2

The End