ELEC 377
Operating Systems

Week 4 Lecture 1

Review

Cannot cover the three weeks in depth, just a quick overview, not comprehensive. There may be material on the quiz not covered today.

Reminder the first Quiz is Tomorrow

- covers weeks 4-6 and lab 2

First Come First Served (FCFS)

- Simple, easy to implement
  - ◊ Ready queue is a first-in-first-out (FIFO) queue
  - ◊ Non-preemptive. Once in the Running state, the process stays running until it asks for I/O or exits.
- 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 in simple OS or in systems with very little variance in CPU burst times

Shortest Job First (SJF)

- scheduling ordered on the length of the next CPU burst
- Nonpreemptive - process gets entire burst time like FCFS

Estimating CPU Burst Times - Exponential Average

- Use length of last CPU burst to predict next burst

t_n = current CPU burst time

τ_0 = initial estimate

τ_n = predicted for current burst

τ_{n+1} = prediction for next CPU burst

α = weighting parameter

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

$\alpha = 0 \rightarrow \tau_{n+1} = \tau_0$ (initial estimate) never changes

$\alpha = 1 \rightarrow \tau_{n+1} = t_n$ (last time slice) only used

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?
  - ◊ α is the tuning parameter
  - ◊ α is low, then past behaviour has more weight, the estimate is slower to change
    - ignore transient behaviour
  - ◊ α is high, then last time slice has heavier weight, the estimate is faster to change
    - more susceptible to transient behaviour

Shortest Remaining Time First (SRTF)

- SJF was non-preemptive
  - ◊ process burst gets entire time needed

- Preemptive version
  - ◊ after an interrupt, calculate remaining time for current burst
  - ◊ will be less than all other process in ready queue
  - ◊ if a new process becomes ready with shorter burst than remaining - dispatch the new process

SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait

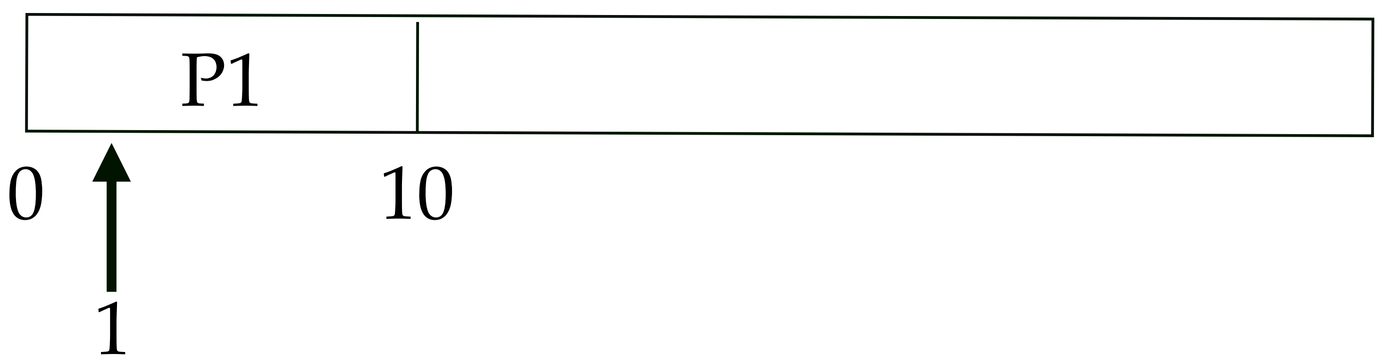
0

SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait

P2 5



SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait

P1	
0	1

P1	9
P2	5

SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait

P1	9

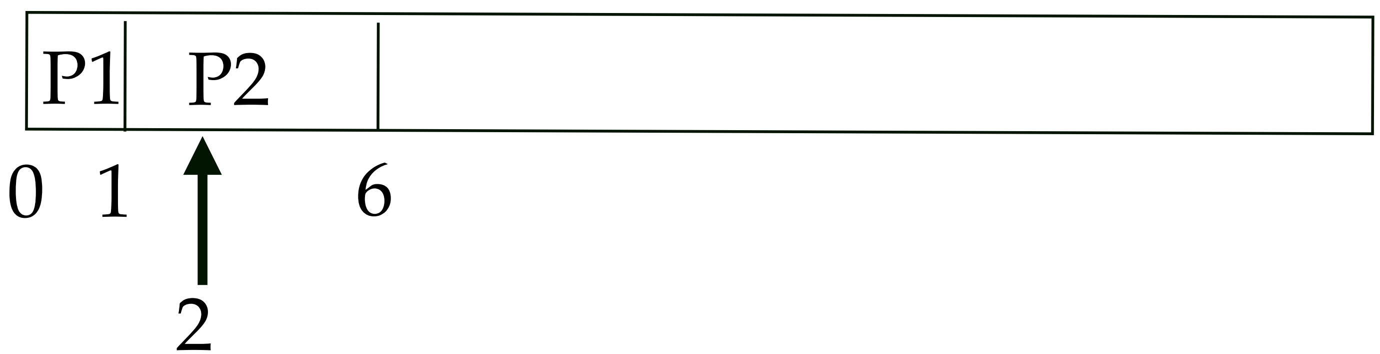
P1	P2	

0 1 6

SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait



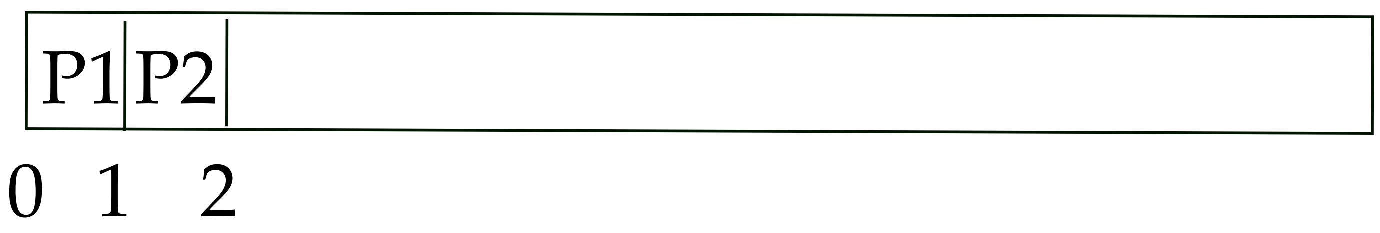
P1 9

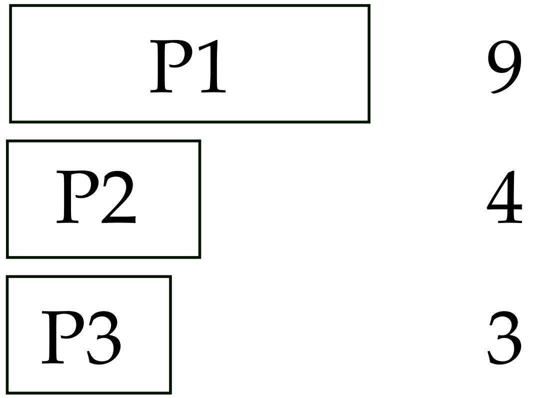
P3 3

SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait





SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait

P1	9
P2	4

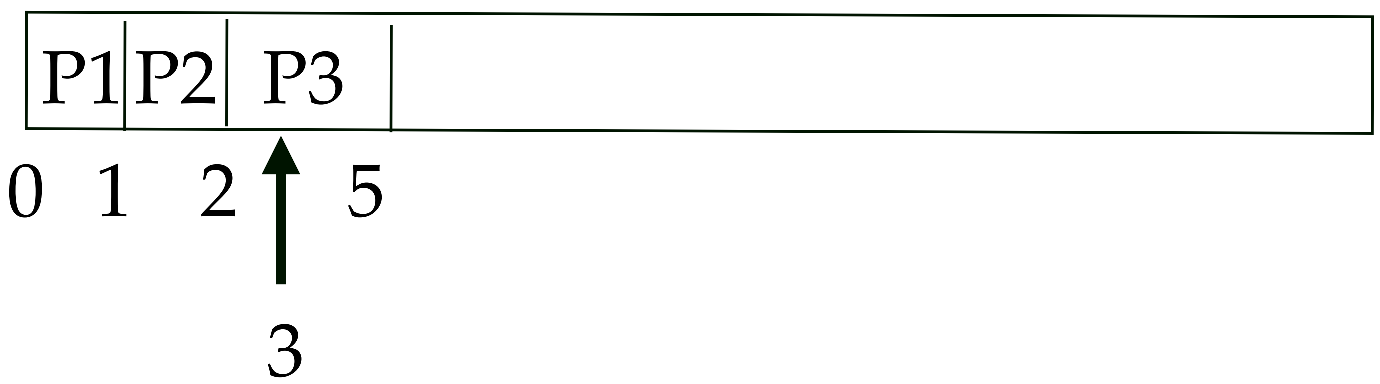
P1	P2	P3	

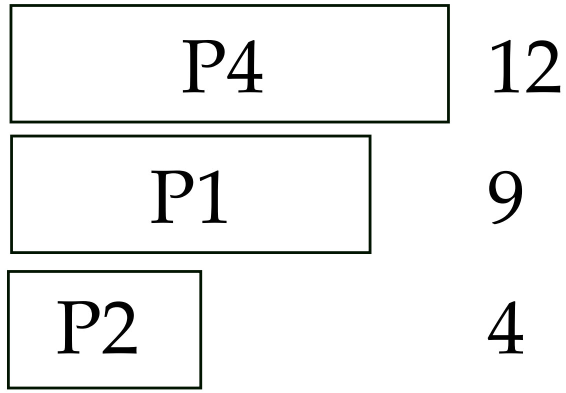
0 1 2 5

SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait

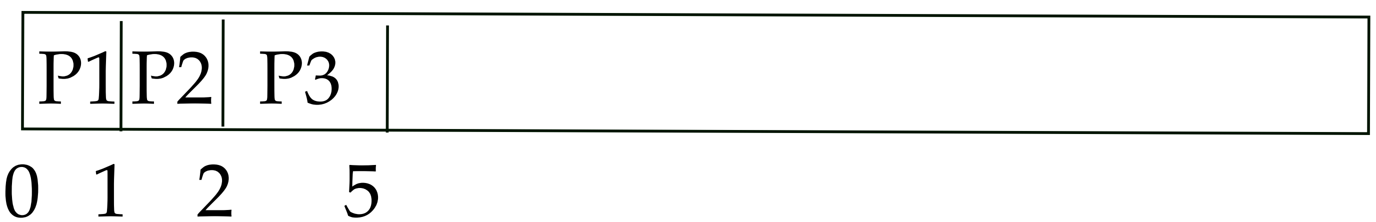


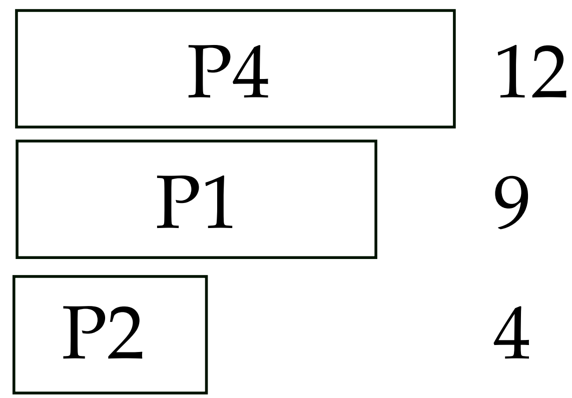


SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait

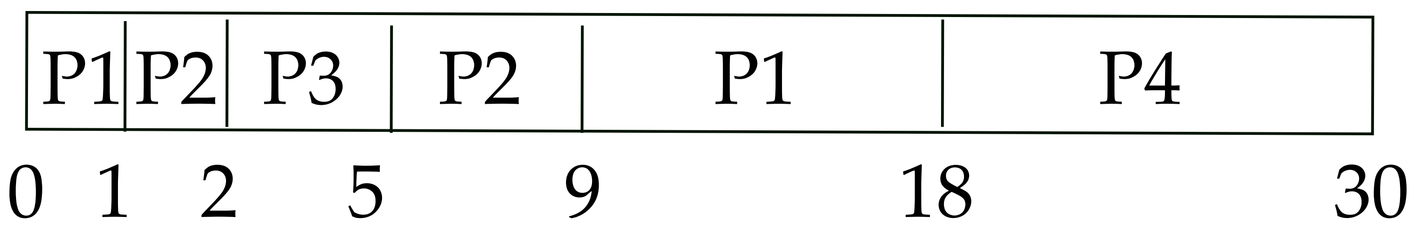




SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

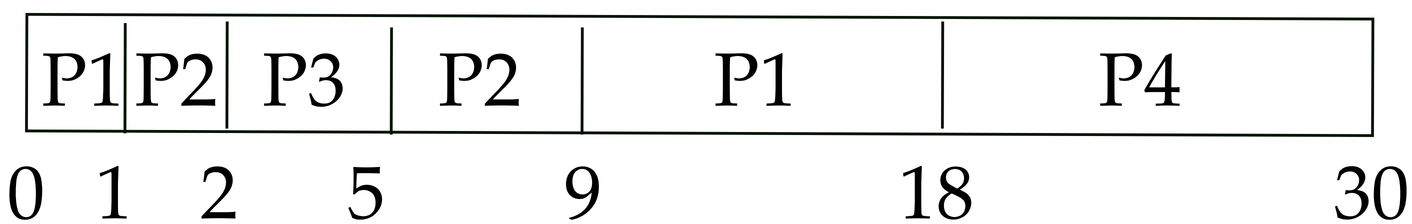
#	Expr	Wait



SRTF Example

#	Arrival	Length
P1	0	10
P2	1	5
P3	2	3
P4	3	12

#	Expr	Wait
P1	0 + (9 - 1)	8
P2	(1-1) + (5 - 2)	3
P3	2 - 2	0
P4	18 - 3	15

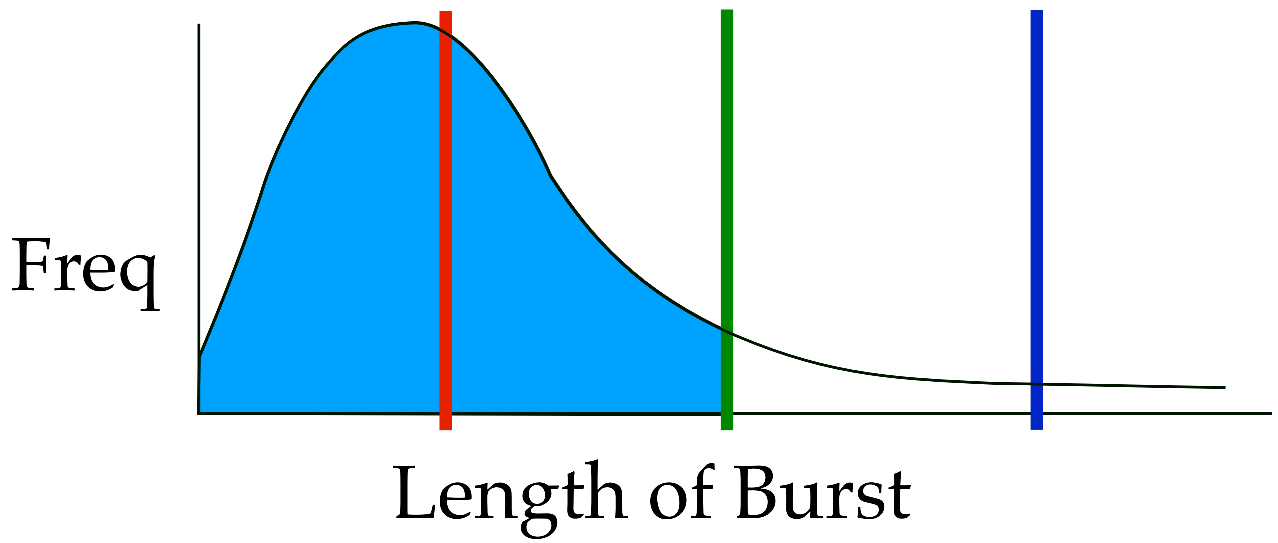


- Total Wait = 26 ms
- Average Wait = 6.5 ms

Round Robin (RR)

- Similar to FCFS, but with 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.
- 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

Round Robin – Quantum Length



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

Priority Scheduling

- Each process has a priority

- CPU goes to process with highest priority
  - ◊ ready queue is sorted based on priority
  - ◊ process with the same priority is handled FCFS
  - ◊ preemptive / nonpreemptive
  - ◊ internal/external
  - ◊ starvation

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) - probably 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

System Model

- Resource Types $R_1, R_2, \ldots, R_n$
  - ◊ Each resource has a number of instances (W_i)
  - ◊ Resource instances are indistinguishable – doesn't matter which one you get.

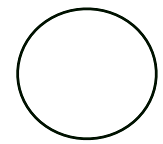
- Process resource protocol
  - ◊ request
  - ◊ use
  - ◊ release

Deadlock Conditions

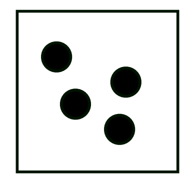
- four conditions necessary for deadlock:
  - ◊ **mutual exclusion**: only a limited number (usually one) process at a time can use a resource
  - ◊ **hold and wait**: a process has (at least) one resource and is waiting for another
  - ◊ **no preemption**: we can't take a resource away from a process
  - ◊ **circular wait**: P_0 waits for a resource held by P_1, which waits for a resource held by P_2, … P_n, which waits for a resource held by P_0

Resource Allocation Graph

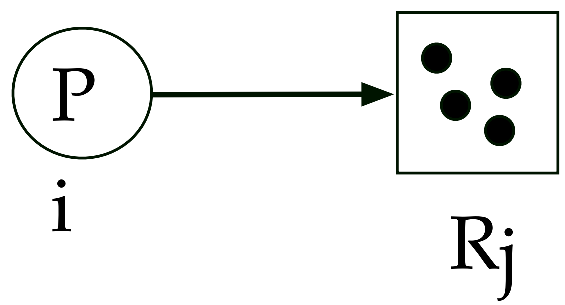
- Process



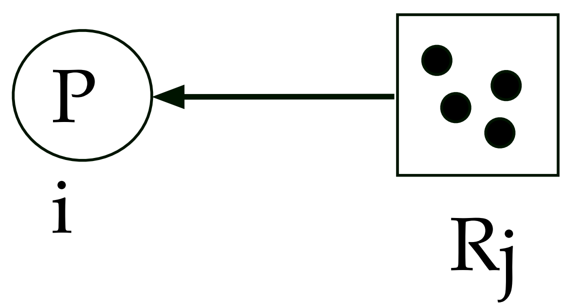
- Resource Type
  - ◊ 4 instances



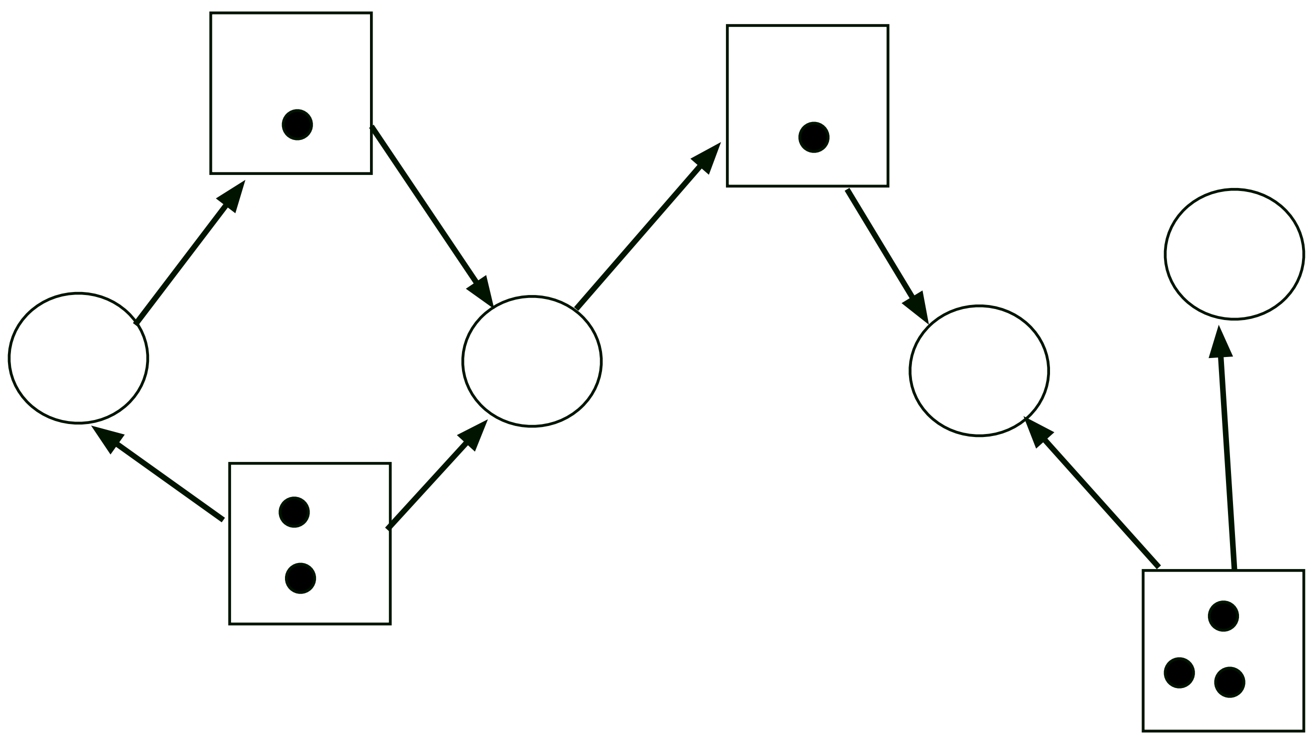
- P_i requests an instance of R_j



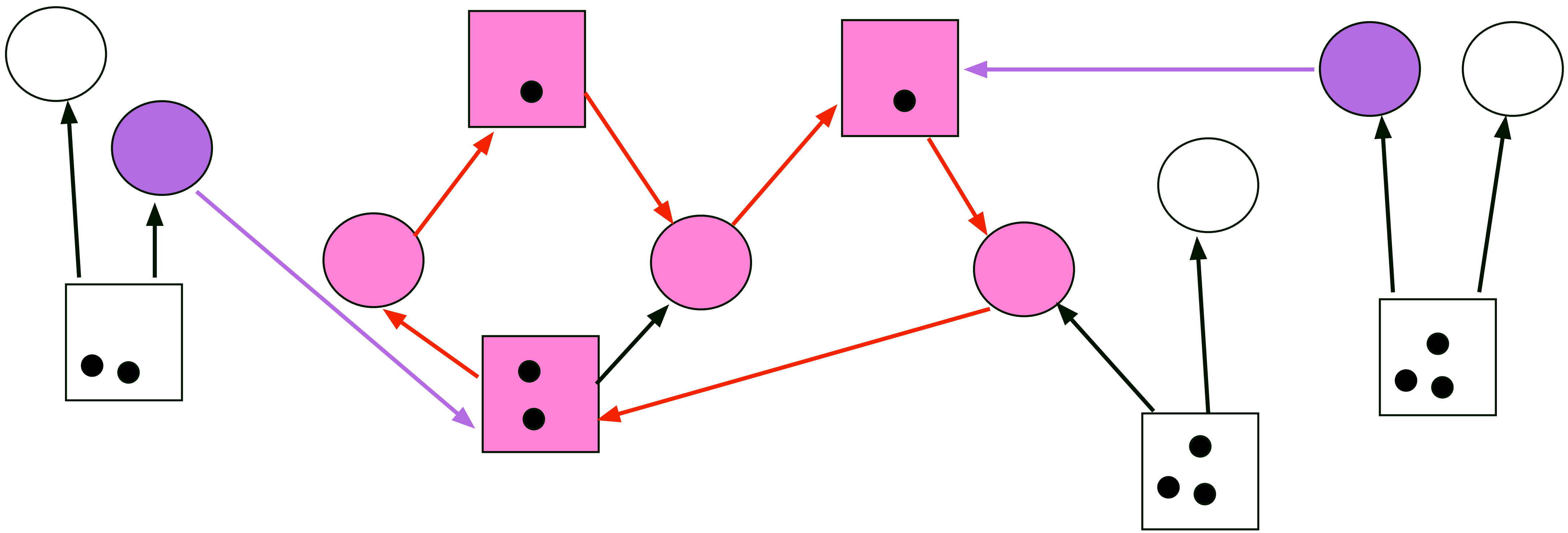
- P_i holds an instance of R_j



Resource Allocation Graph Example



Deadlock Example



What do we do??

- Prevention
  - ensure one of the 4 conditions never happens
- Avoidance:
  - extra information before allocating an available resource
- Recovery:
  - enter deadlock state and recover
- Ignore
  - hope it never happens
  - handle it manually
  - most interactive operating systems use this approach

Safe State

- system is safe if there is some order we can allocate the resources and not produce a deadlock
  - ◊ might not be the order that the processes actually request the resources
  - ◊ safe order means that one of the processes may have to wait
- <P1, P2, …, Pn> is safe if Pi can satisfy the maximum resources with available and the resources owned by previous processes.
  - ◊ P1 max must be satisfied only with free resources
  - ◊ P2 max must be satisfied with free + P1
  - ◊ P3 max gets available + P1 + P2
  - ◊ If not, wait until a previous process finishes.

Safe State – Examples

Process	Current	Max
P0	5	10
P1	2	4
P2	2	9

Total = 12, Free = 3

$< P1 , P0 , P2 >$

P1 (2) + 3 = 5 >= P1Max(4)

5 + P0(5) = 10 >= P0Max(10)

5 + P2(2) = 7   not >= P2Max(9)

10 + P2(2) = 12 >= P2Max(9)

Binding Instructions and Data

- Entities in the original program must be bound to a location in memory
    - ◊ int count;
- Programs may reside in different parts of memory
- Three different stages
    - ◊ Compile (and linkage) time (MS-DOS .COM, Arduino)
    - ◊ Load Time – When loader loads program into memory, addresses are resolved (early Unix, Linux)
    - ◊ Execution Time – programs can move in memory - hardware support required

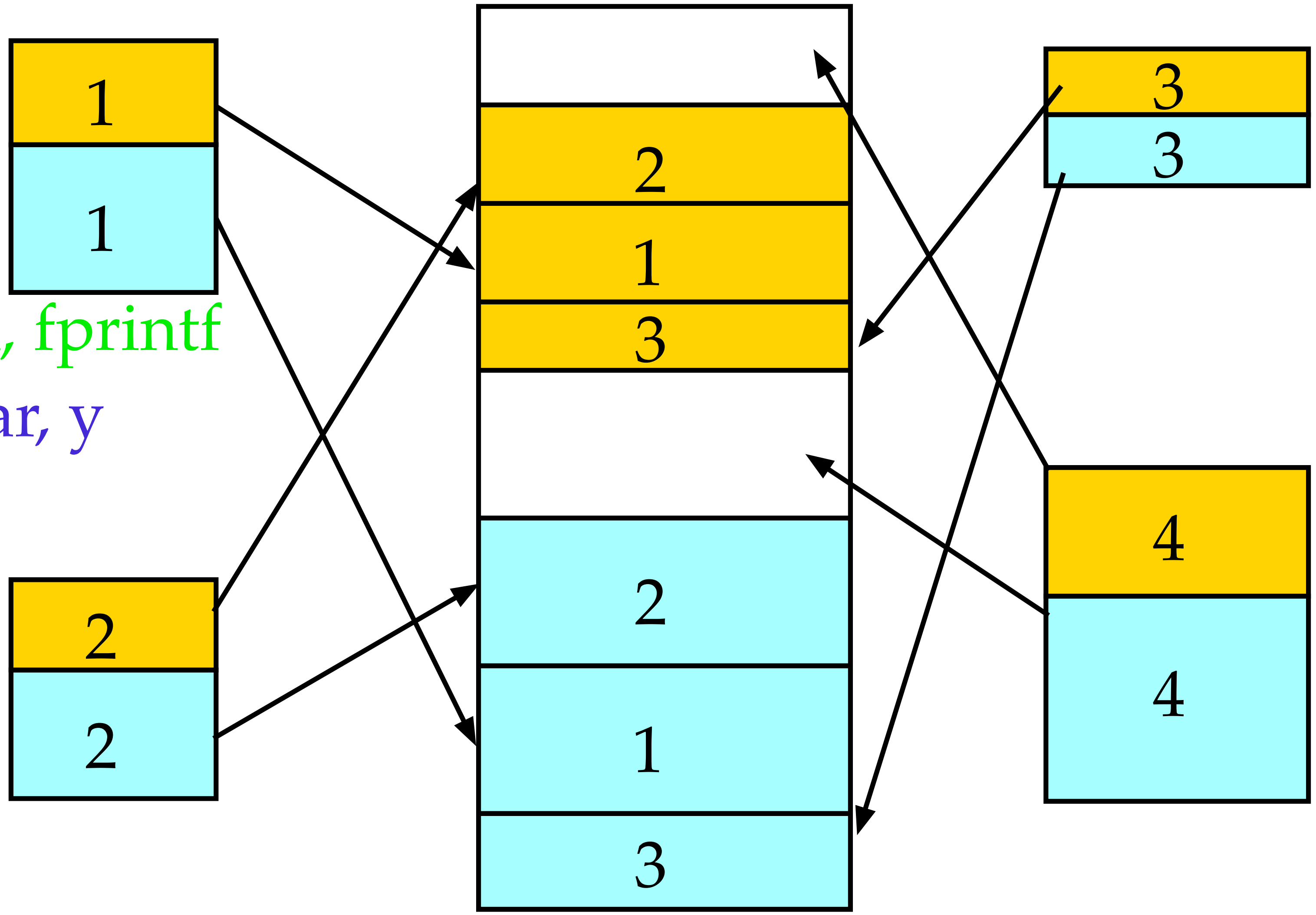
Compilation Process

- This presentation describes how several separate programs are compiled and assembled (linked) into an executable.

Linking–combine to single executable

foo.o
def main, f,
x,foo
undef fopen, fprintf
stderr, bar, y

bar.o
def bar,y
undef foo,x



crt.o
- defines entry
- undef main,
exit

stdio.o
-def fopen,
fprintf, stderr,
exit

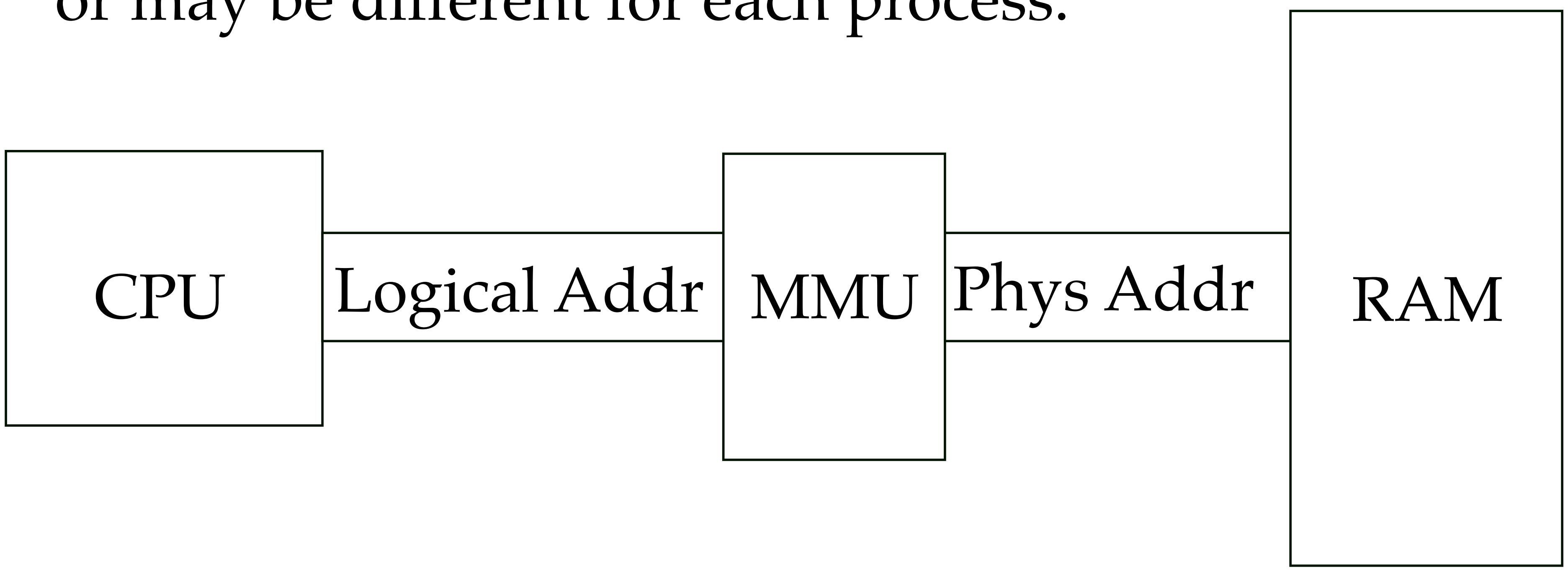
Logical vs Physical Address Space

Central Concept to Memory Management

- Logical Address
  - ◊ address generated by CPU
  - ◊ also known as a virtual address
- Physical Address
  - ◊ location in physical memory
- Logical and Physical address are the same in compile and load time address binding. They differ in execution time binding
- User program only deals with logical addresses. It never sees the physical address

Memory Management Unit (MMU)

- Hardware that maps virtual to physical address
  - ◊ many different approaches
- Logical address may be shared between process, or may be different for each process.



Contiguous Memory

- Using relocation and limit register assumes that the process is given a contiguous chunk of physical memory.
  - ◊ single partition allocation
  - ◊ simple allocation strategy, similar to that used by malloc in C or new in Java.

- main memory is divided into two parts
  - ◊ operating system (usually in same part as the interrupt vector)
  - ◊ User memory (divided among processes)

Storage Allocation

Three general approaches

- First Fit
  - ◊ use the first hole on the free list that is big enough
  - ◊ only look at part of list
- Best Fit
  - ◊ smallest block that is large enough
  - ◊ search entire list
- Worst Fit
  - ◊ largest block (largest remainder)
  - ◊ worst algorithm

Fragmentation

- Internal Fragmentation
  - ◊ if we allocate memory in units larger than a single byte (say 1K)
  - ◊ last block is only partially used
- External Fragmentation
  - ◊ lots of small holes spread throughout memory, none are big enough to satisfy a request
  - ◊ worst fit tries to reduce this
  - ◊ compaction - move blocks (requires execution-time binding)

Shared Libraries

- *Static* linking is when the all of the modules including system libraries are linked together at compile time.
  - ◊ shared libraries must be preallocated into common address space at runtime.
- When dynamic linking, the OS invokes the dynamic linker when the process is loaded.
  - ◊ linking uses the Program Link Table (PLT)
  - ◊ The code must be position independent.

Paging

- Why should memory have to be contiguous?
- Physical memory is divided into frames
  - ◊ typically 512 bytes to 8K in size
  - ◊ Linux is 4K
- Logical memory is divided into pages
  - ◊ same size as frames
- If process needs *n* pages, find *n* free frames in memory
  - ◊ no need to be contiguous
- A table translates from each page to the appropriate frame
- No external fragmentation, but still have internal fragmentation

Paging

- Logical Address Space and Physical Address space may not be the same size!!
  ◊ physical address may be larger
  (e.g. 32 bit logical, 40 bit physical)
  ◊ physical address may be smaller
  (64 bit logical = 1.8e19 bytes)

- Frame and page always the same size
  – always power of 2

Paging - TLB

- Want to minimize extra memory traffic of page tables
- Small cache inside of MMU
  - ◊ TLB - translation look-aside buffer
  - ◊ associative memory

Page #	Frame #

Sharing Pages

- Shared Libraries
- Multiple invocations of a given program (e.g. shell, editor)
- Contiguous memory allocation made sharing difficult
- Shared code must be reentrant
  - ◊ Must not modify itself
  - ◊ Must also be position independent
- Most data is not shared
  - ◊ however IPC Shared Segments are now easy!!
- Page table entries for shared code and data in each process point to the common frames
- Page table entries for private data point to different frames

Page Structures - Summary

- Three ways of reducing the memory requirements of the page tables
- *All* of them increase the cost of converting a logical address to a physical address
  - ◊ TLB absorbs much of the cost
  - ◊ Increase the cost of a TLB miss
  - ◊ Effectiveness of TLB is more important
- Hardware vs software table walk

Virtual Memory

- Not all of the program need be in memory at one time
  - ◊ dynamic loading, overlays
- Some code may never be executed for a given instance of a program.
  - ◊ A word processor may have to store inline video in a document.
- Not all of the code or data in a program is used at the same time
  - ◊ compilers read and analyze source code before generating output assembly code
  - ◊ word processors only read from a file on an open command.
- If the code is not executing, why should it be in memory?