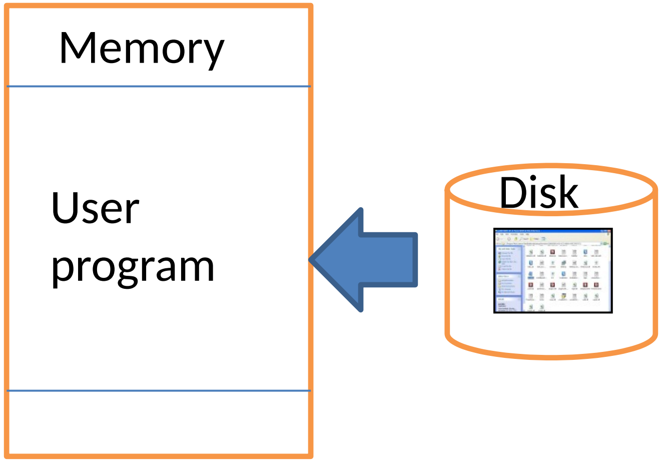
Process management

What are we going to learn?

- ***Processes :*** Concept of processes, process scheduling, co-operating processes, inter-process communication.
- ***CPU scheduling :*** scheduling criteria, preemptive & non-preemptive scheduling, scheduling algorithms (FCFS, SJF, RR, priority), algorithm evaluation, multi-processor scheduling.
- ***Process Synchronization :*** background, critical section problem, critical region, synchronization hardware, classical problems of synchronization, semaphores.
- ***Threads :*** overview, benefits of threads, user and kernel threads.
- ***Deadlocks :*** system model, deadlock characterization, methods for handling deadlocks, deadlock prevention, deadlock avoidance, deadlock detection, recovery from deadlock.

Process concept

- Process is a dynamic entity
  - Program in execution
- Program code
  - Contains the text section
- Program becomes a process when
  - executable file is loaded in the memory
  - Allocation of various resources
    - Processor, register, memory, file, devices
- One program code may create several processes
  - One user opened several MS Word
  - Equivalent code/text section
  - Other resources may vary



Memory

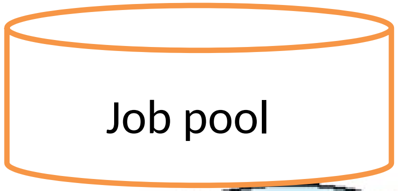
User program

Disk

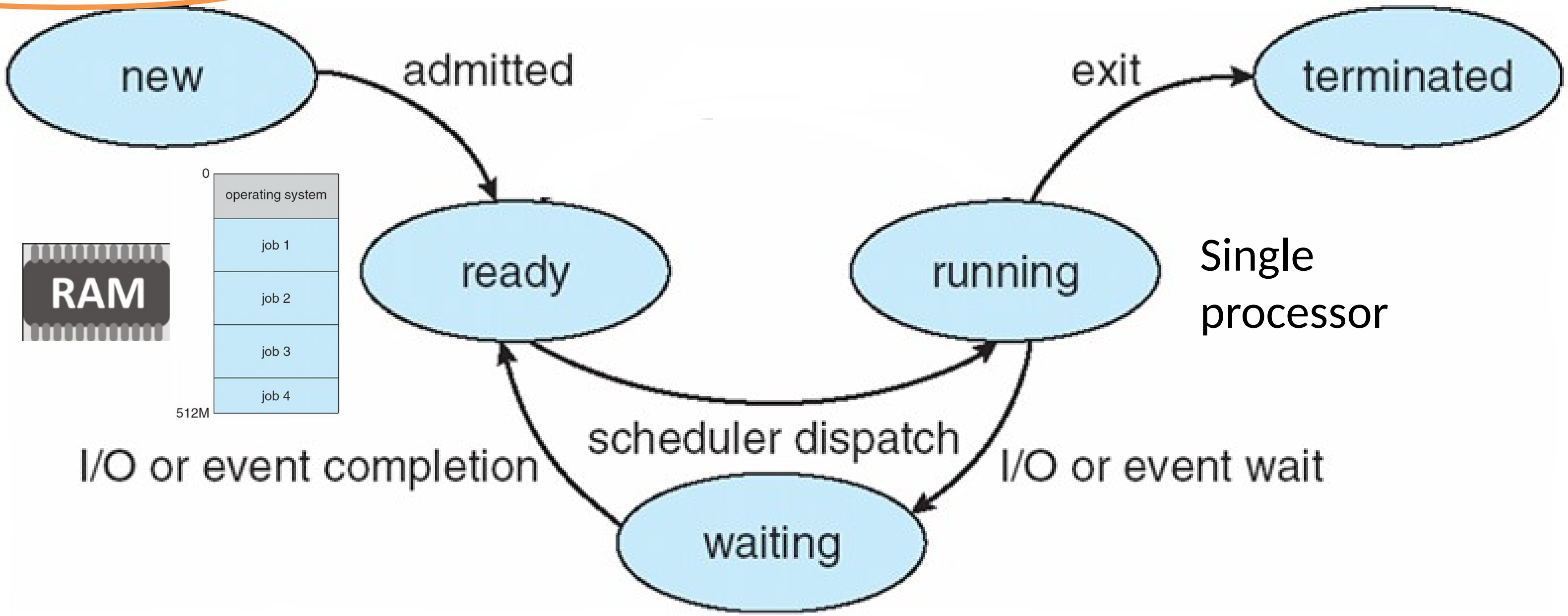
Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution

Process State diagram



Job pool



Single processor

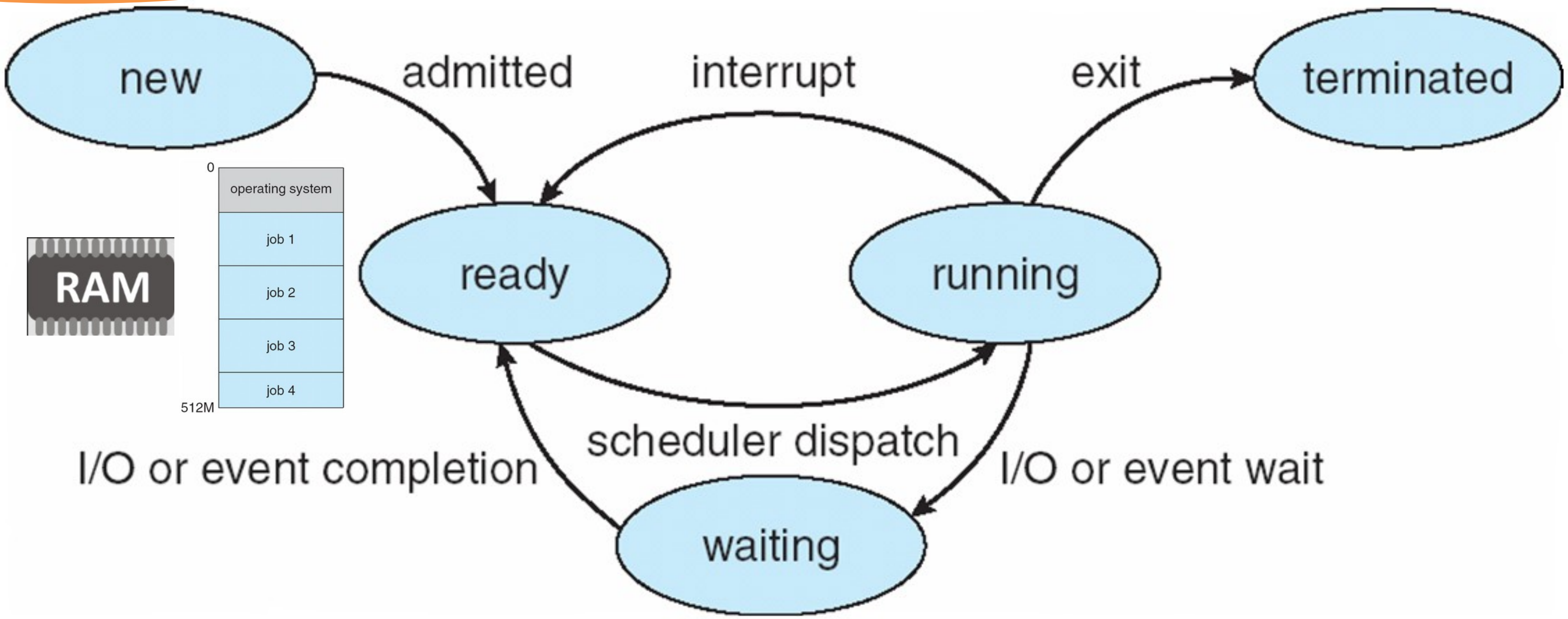
Multiprogramming

As a process executes, it changes *state*

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

Job pool

Process State diagram



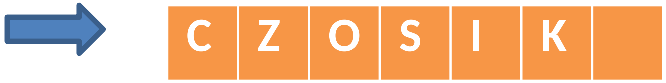
Multitasking/Time sharing

As a process executes, it changes *state*

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

How to represent a process?

- Process is a dynamic entity
  - Program in execution
- Program code
  - Contains the text section
- Program counter (PC)
- Values of different registers
  - Stack pointer (SP) (maintains process stack)
    - Return address, Function parameters
  - Program status word (PSW)
  - General purpose registers
- Main Memory allocation
  - Data section
    - Variables
  - Heap
    - Dynamic allocation of memory during process execution



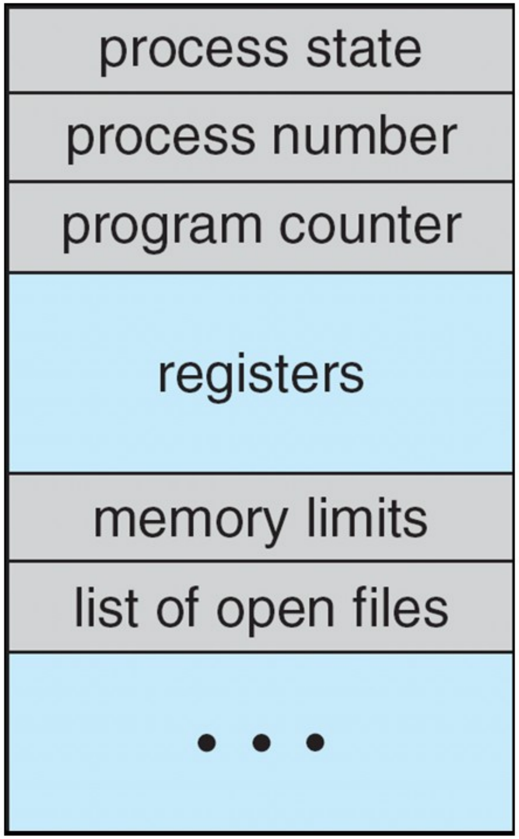
C	Z	O	S	I	K	

Process Control Block (PCB)

- Process is represented in the operating system by a Process Control Block

Information associated with each process

- Process state
- Program counter
- CPU registers
  - Accumulator, Index reg., stack pointer, general Purpose reg., Program Status Word (PSW)
- CPU scheduling information
  - Priority info, pointer to scheduling queue
- Memory-management information
  - Memory information of a process
  - Base register, Limit register, page table, segment table
- Accounting information
  - CPU usage time, Process ID, Time slice
- I/O status information
  - List of open files=> file descriptors
  - Allocated devices

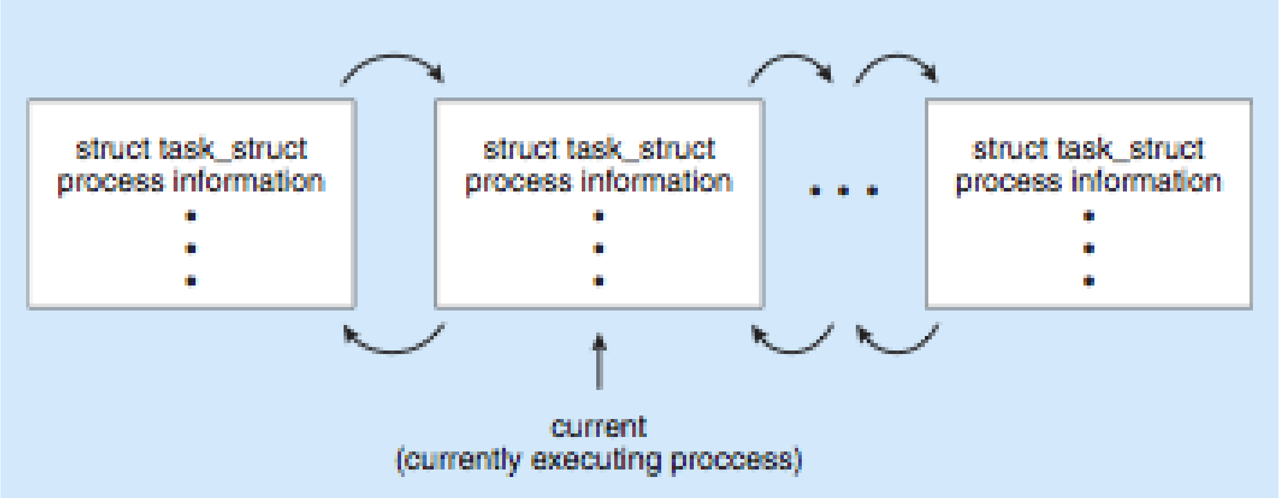


Process Representation in Linux

Represented by the C structure `task_struct`

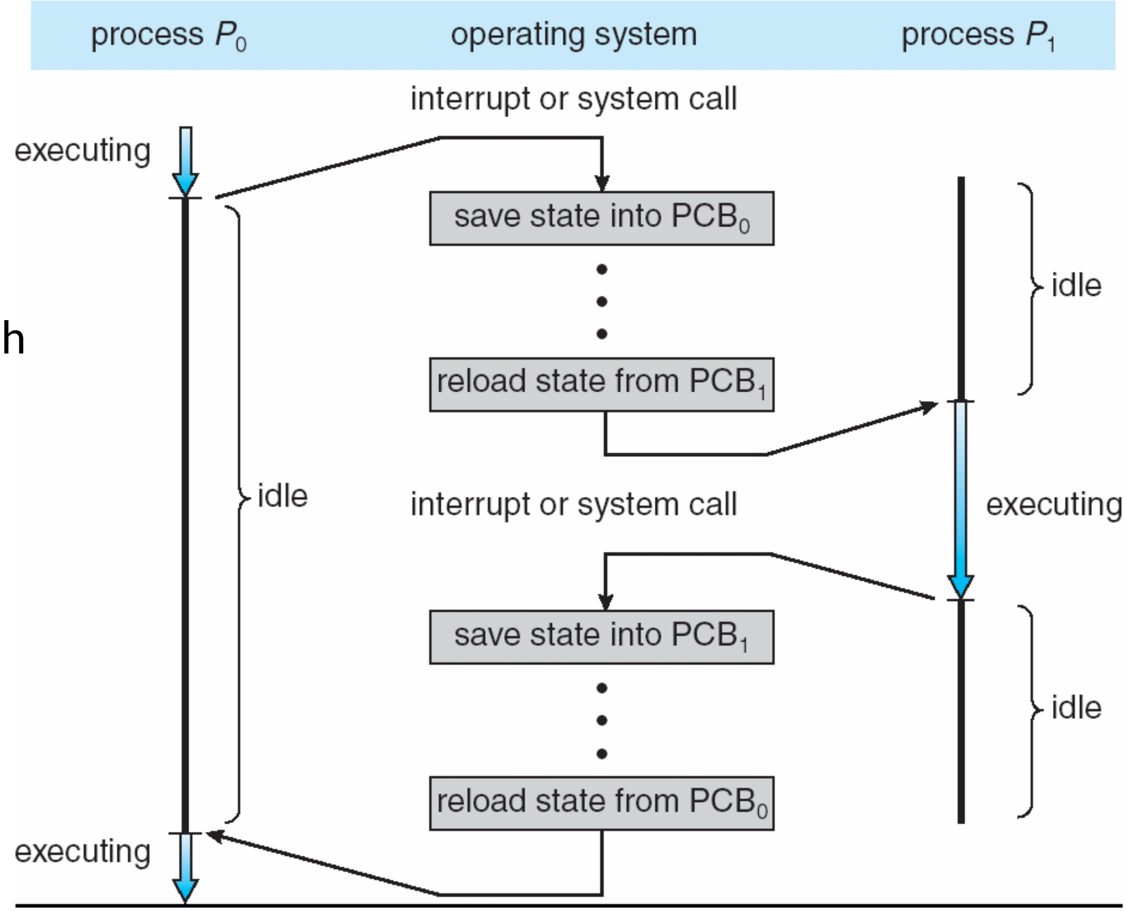
```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task struct *parent; /* this process’s parent */
struct list head children; /* this process’s children */
struct files struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this pro */
```

Doubly
linked list

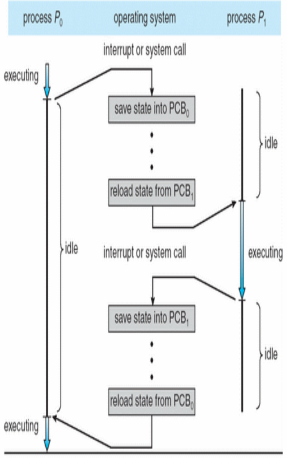


CPU Switch From Process to Process

Context switch



Context Switch



- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**.

- **Context** of a process represented in the PCB

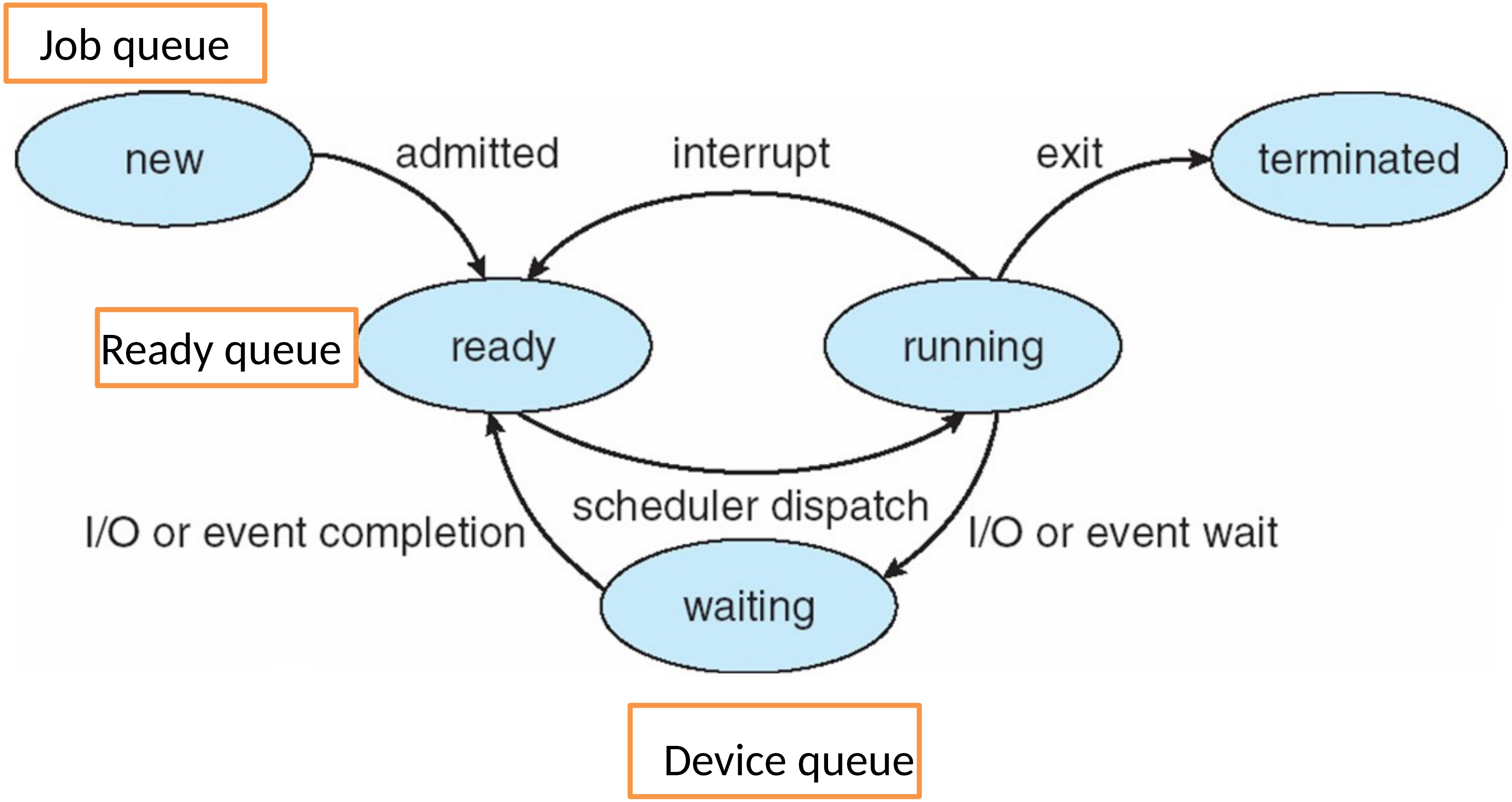
- Context-switch time is overhead; the system does no do useful work while switching
  - The more complex the OS and the PCB -> longer the context switch

- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

Scheduling queues

- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Scheduling queues



Job queue

Ready queue

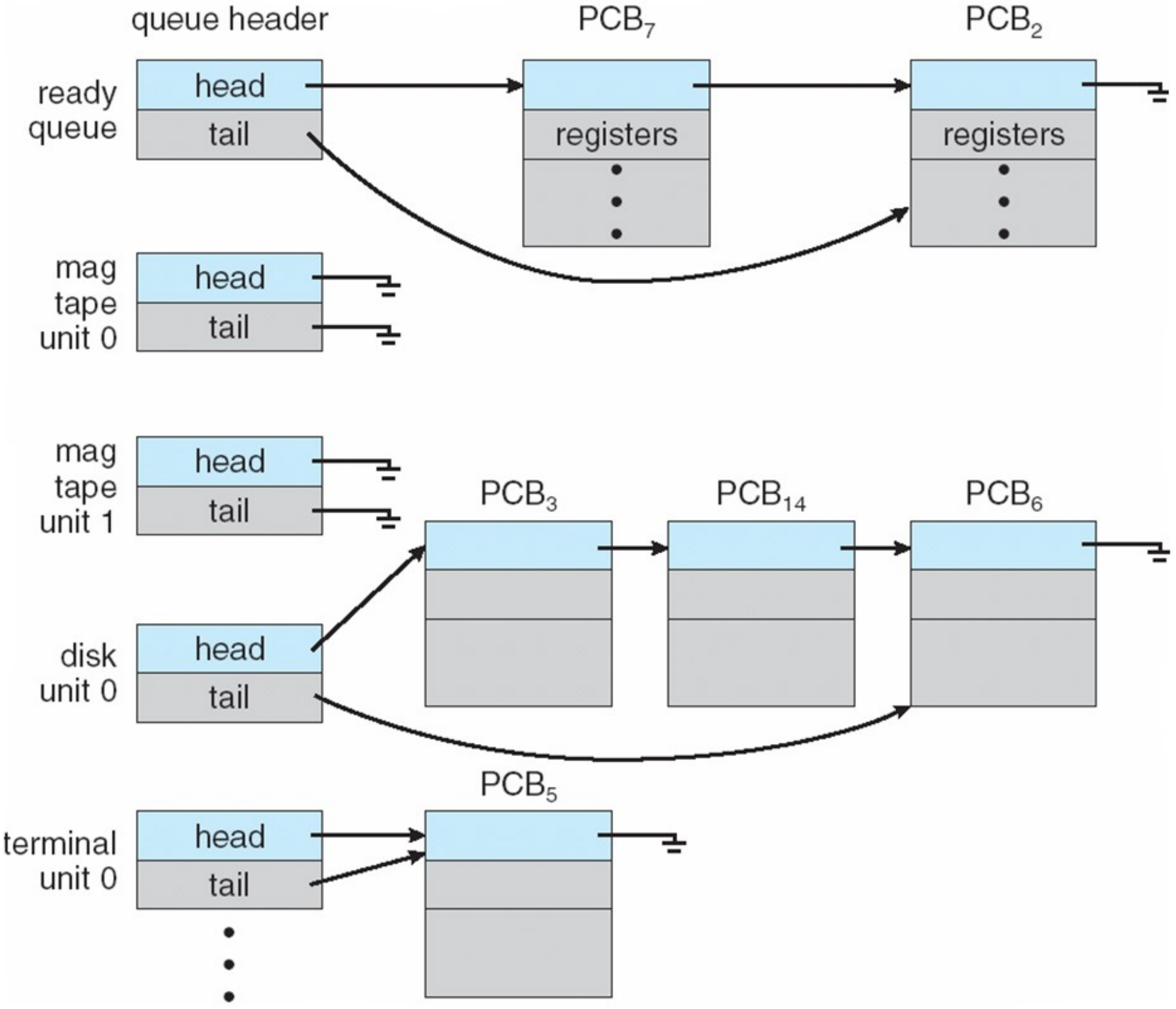
Device queue

Ready Queue And Various I/O Device Queues

Queues are linked list of PCB's

Device queue

Many processes are waiting for disk



Process Scheduling

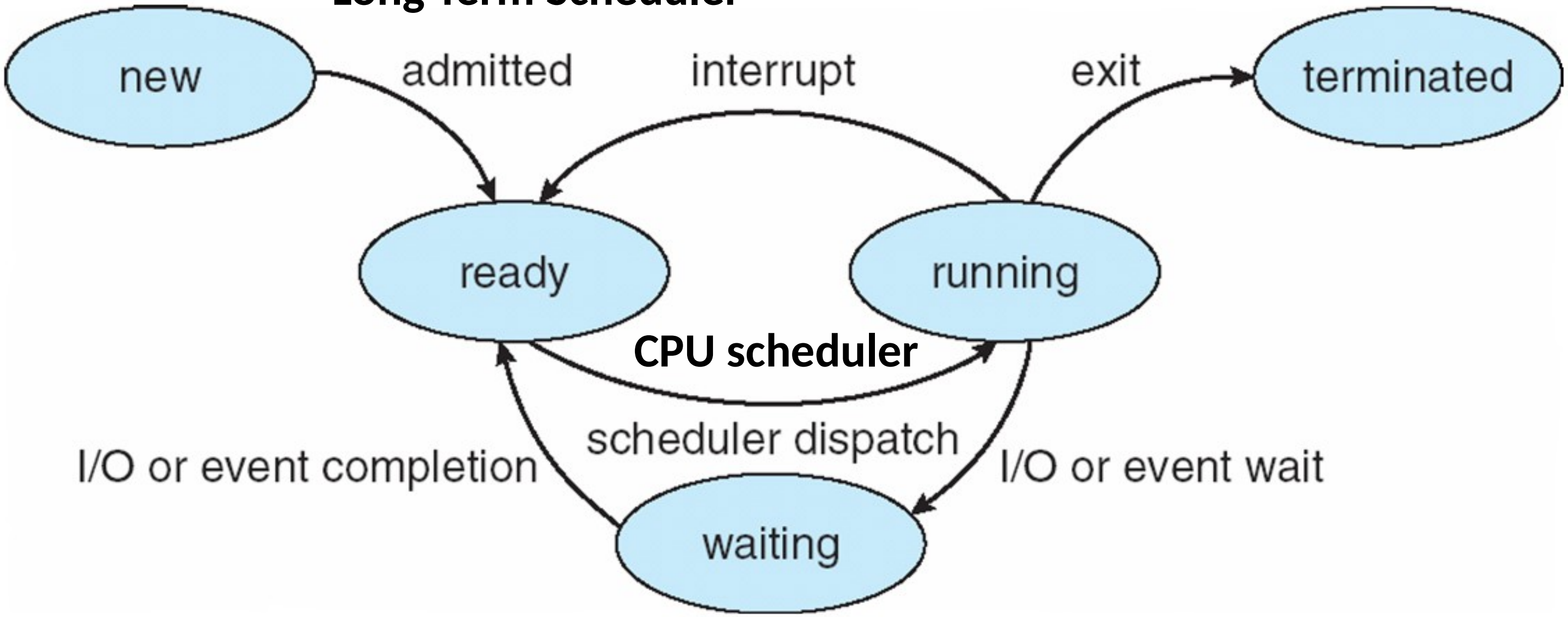
- We have various queues
- Single processor system
  - Only one CPU=> only one running process
- Selection of one process from a group of processes
  - **Process scheduling**

Process Scheduling

- Scheduler
  - Selects a process from a set of processes
- Two kinds of schedulers

1. Long term schedulers, job scheduler
   - A large number of processes are submitted (more than memory capacity)
   - Stored in disk
   - Long term scheduler selects process from job pool and loads in memory
2. Short term scheduler, CPU scheduler
   - Selects one process among the processes in the memory (ready queue)
   - Allocates to CPU

Long Term Scheduler

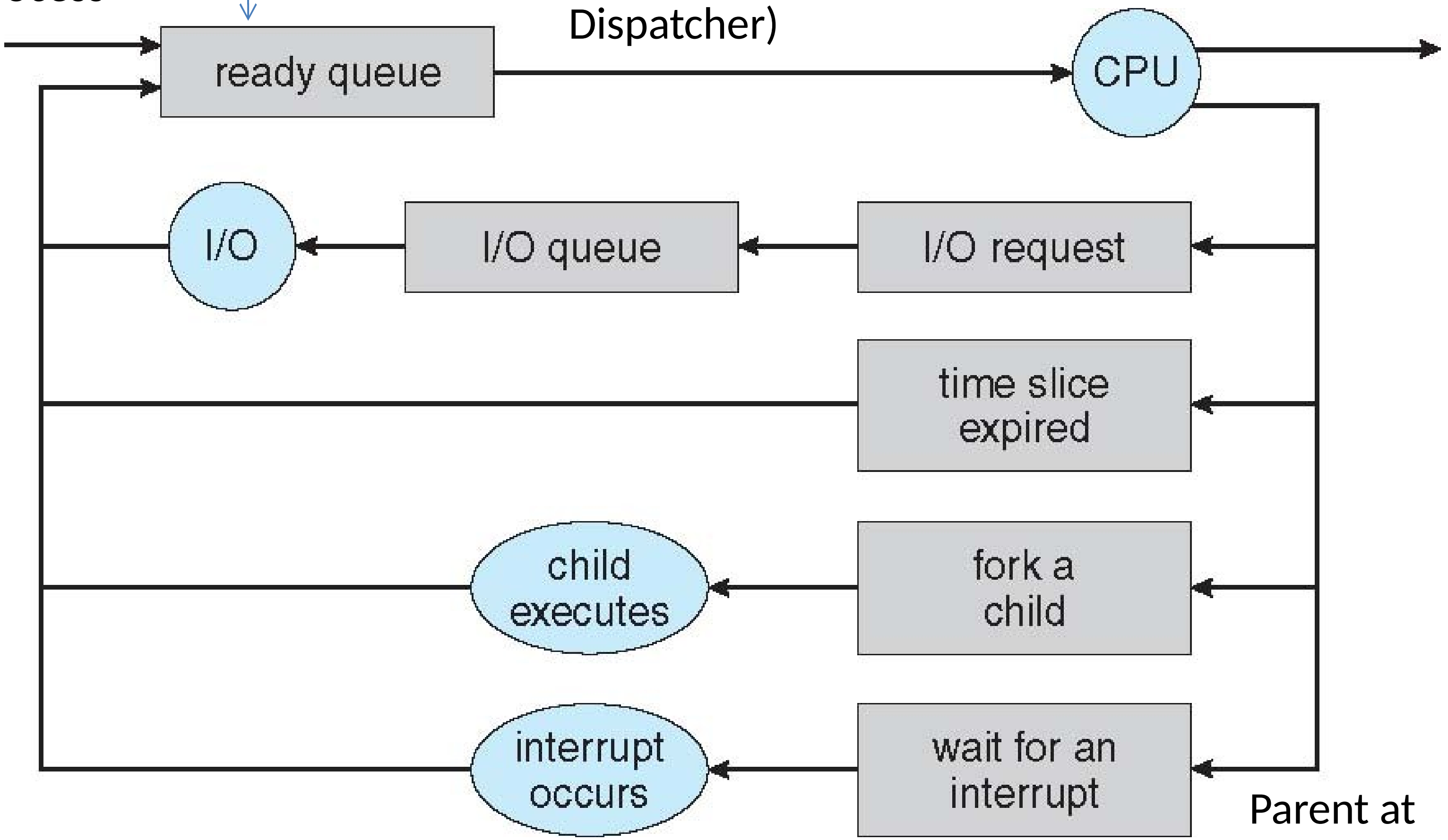


CPU scheduler

Representation of Process Scheduling

CPU scheduler selects a process

Dispatched (task of Dispatcher)



Parent at wait()

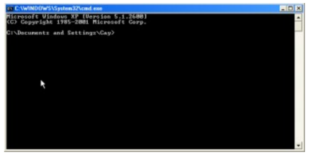
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

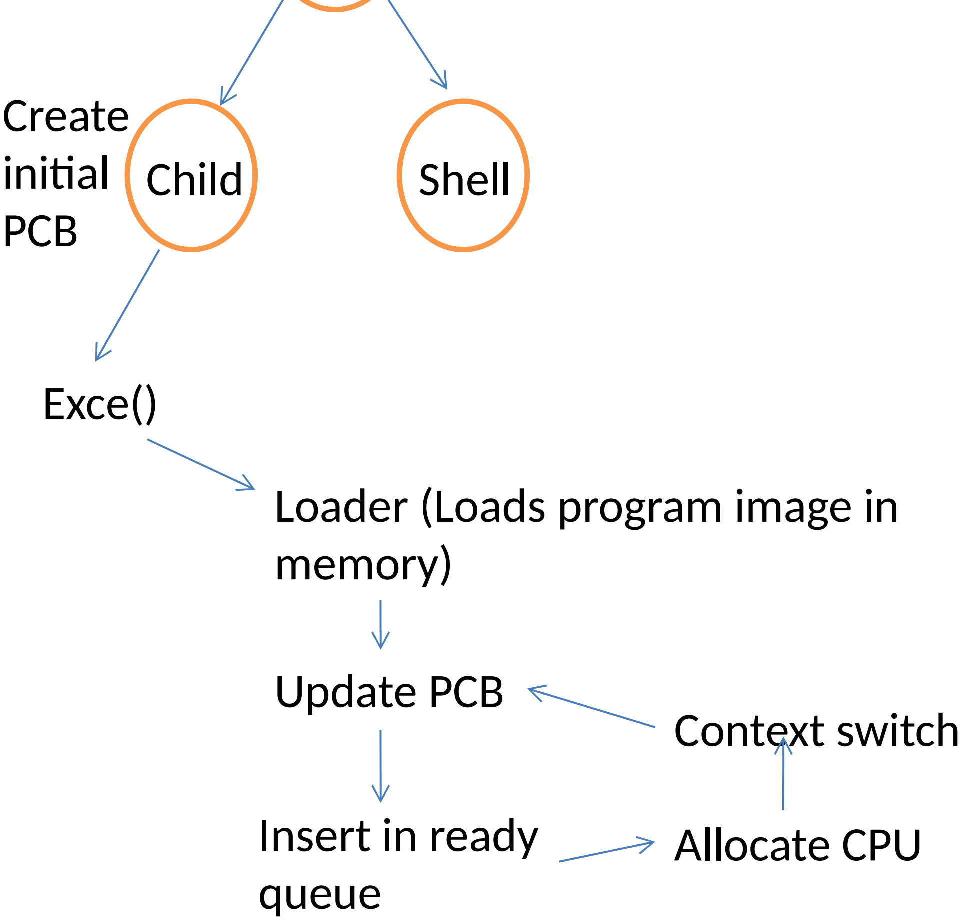
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Creation of PCB

a.out



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Gay>
```



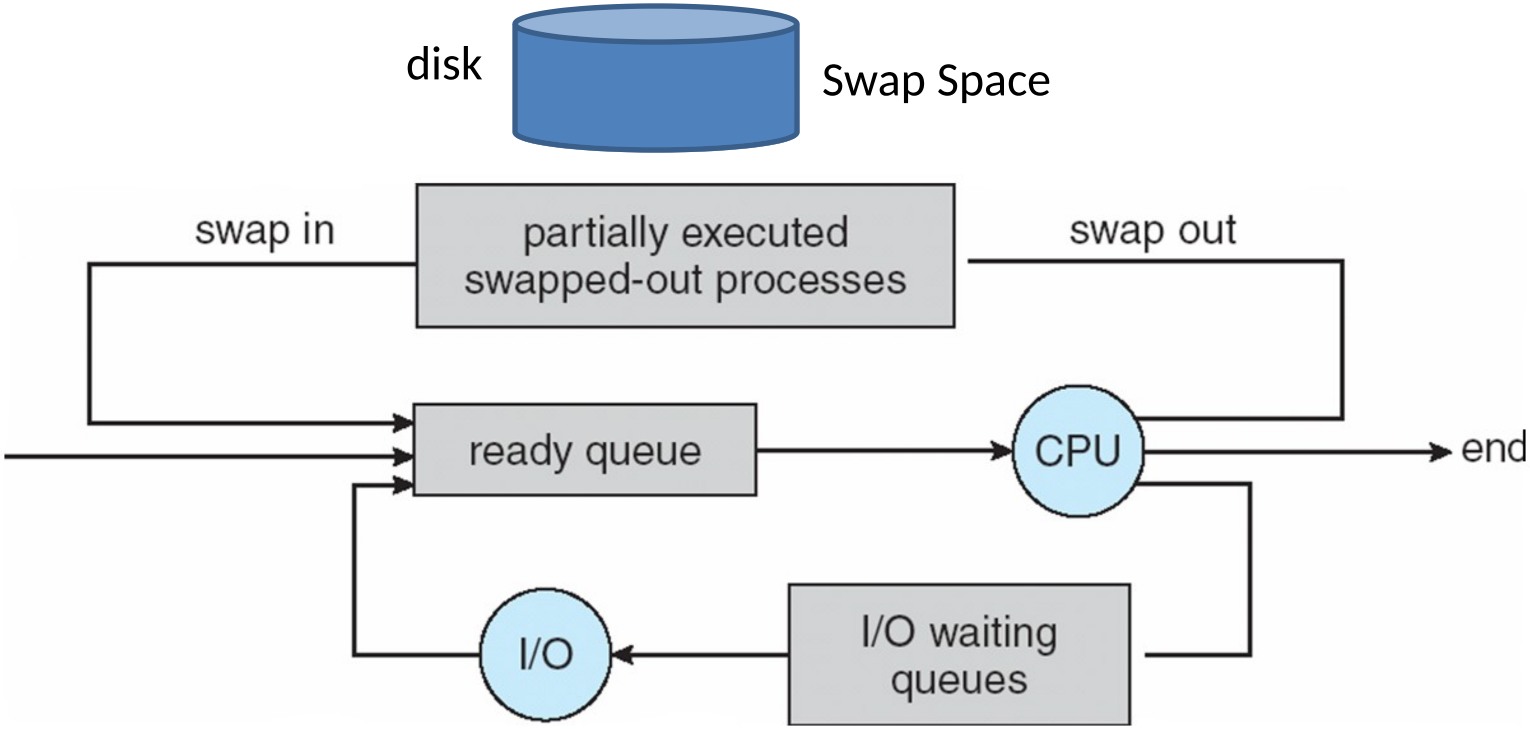
Schedulers

- **Scheduler**
  - Selects a process from a set
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system

Schedulers: frequency of execution

- Short-term scheduler is invoked very frequently (milliseconds) ⏶ (must be fast)
  - After a I/O request/ Interrupt
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⏶ (may be slow)
  - The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
    - Ready queue empty
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
    - Devices unused
- Long term scheduler ensures good process mix of I/O and CPU bound processes.

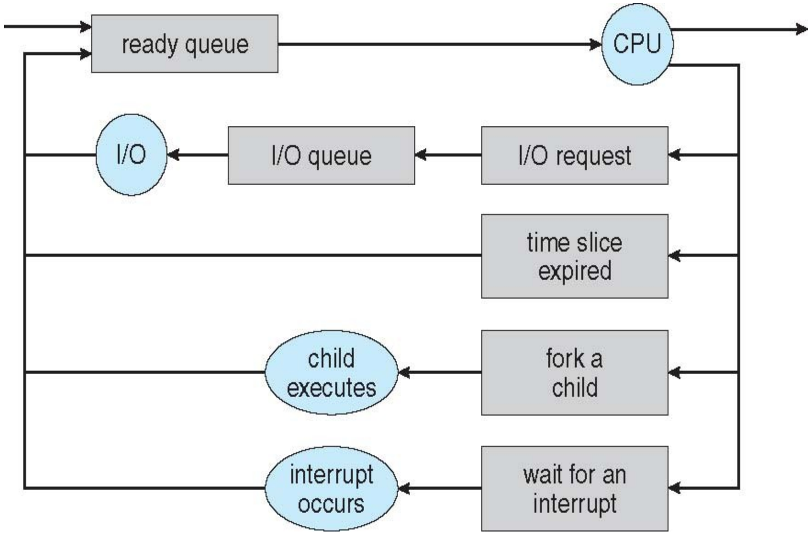
Addition of Medium Term Scheduling



Swapper

ISR for context switch

```
Current <- PCB of current process
Context_switch()
{
    switch to kernel mode
    Disable interrupt;
                Save_PCB(current);
    Insert(ready_queue, current);
    next=CPU_Scheduler(ready_queue);
    remove(ready_queue, next);
    Enable Interrupt;

    Dispatcher(next);
}
Dispatcher(next)
{
    Load_PCB(next); [update PC]
    switch to user mode;

}
```



CPU Scheduling

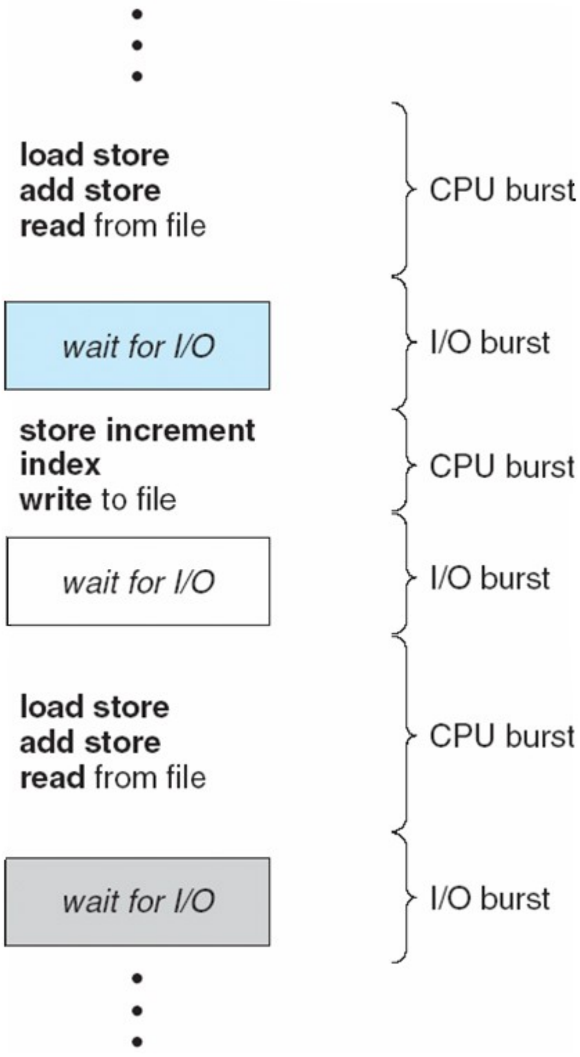
- Describe various CPU-scheduling algorithms

- Evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

Basic Concepts

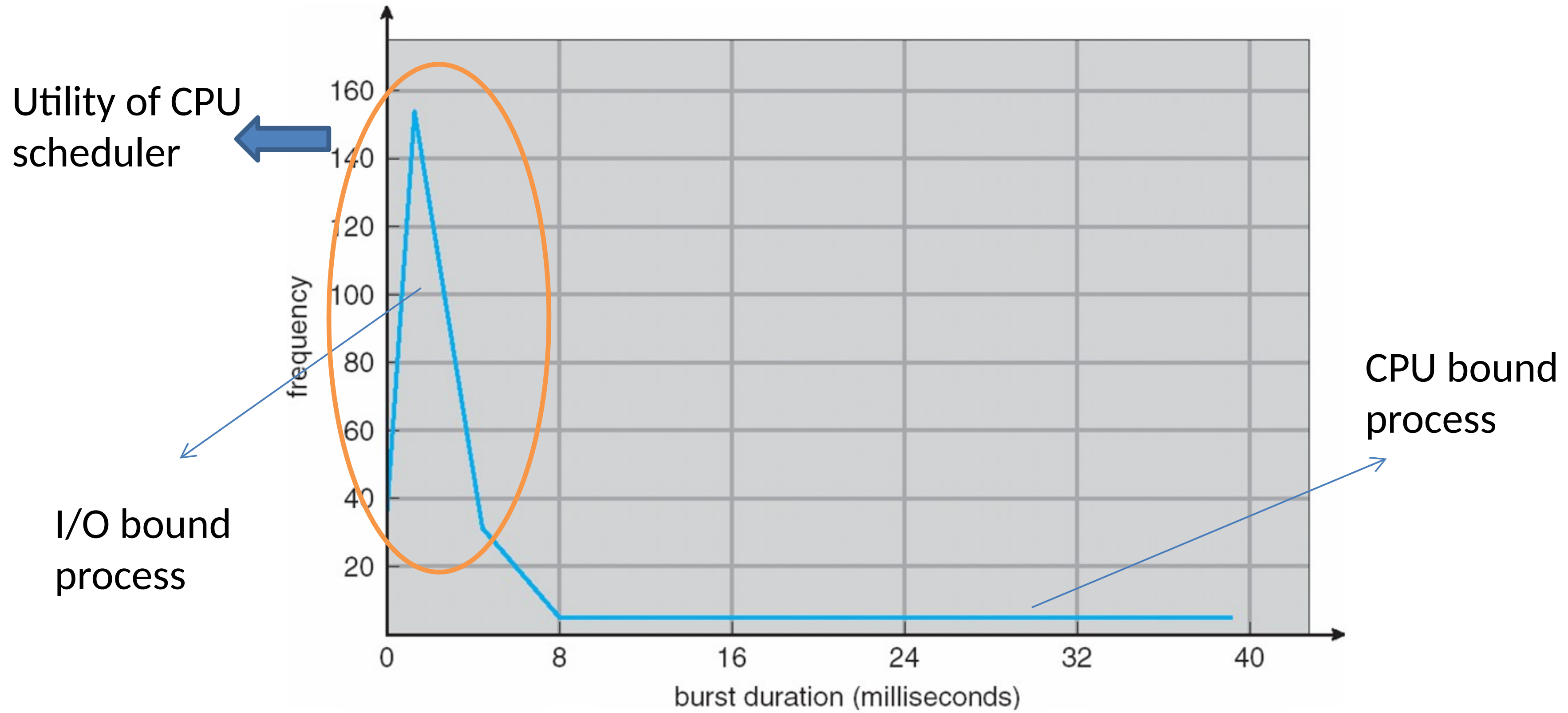
- Maximum CPU utilization obtained with multiprogramming
  - Several processes in memory (ready queue)
  - When one process requests I/O, some other process gets the CPU
  - Select (schedule) a process and allocate CPU

Observed properties of Processes



- CPU–I/O Burst Cycle
- Process execution consists of a *cycle* of CPU execution and I/O wait
- Study the duration of CPU bursts

Histogram of CPU-burst Times



Utility of CPU scheduler

I/O bound process

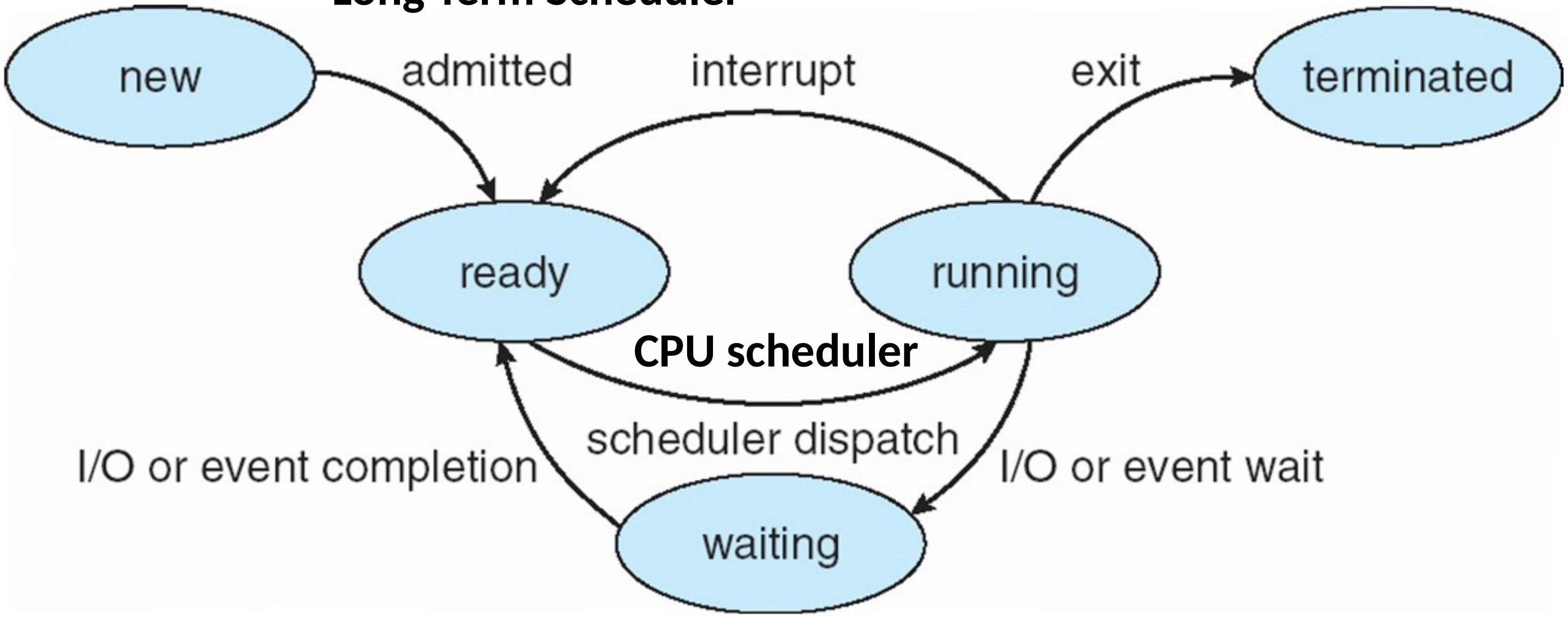
CPU bound process

Large number of short CPU bursts and small number of long CPU bursts

Preemptive and non preemptive

- Selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways (not necessarily FIFO)
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Long Term Scheduler



CPU scheduler

Preemptive scheduling

Preemptive scheduling

Results in cooperative processes

Issues:

- Consider access to shared data
  - Process synchronization
- Consider preemption while in kernel mode
  - Updating the ready or device queue
  - Preempted and running a “ps -el”

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

- Mostly optimize the average
- Sometimes optimize the minimum or maximum value
  - Minimize max response time

- For interactive system, variance is important
  - E.g. response time
- System must behave in predictable way

Scheduling algorithms

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority Scheduling
- Round Robin (RR)

First-Come, First-Served (FCFS) Scheduling

- Process that requests CPU first, is allocated the CPU first
- Ready queue=>FIFO queue
- Non preemptive
- Simple to implement

Performance evaluation

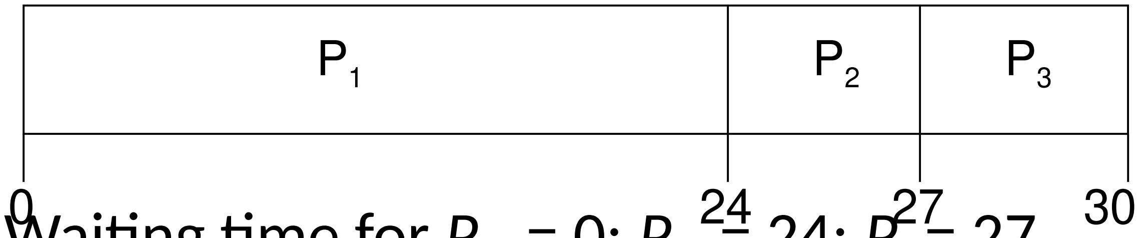
- Ideally many processes with several CPU and I/O bursts

- Here we consider only one CPU burst per process

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:



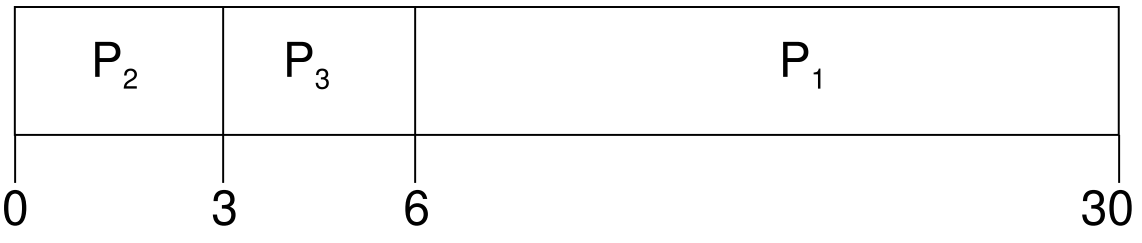
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- Average waiting time under FCFS heavily depends on process arrival time and burst time
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

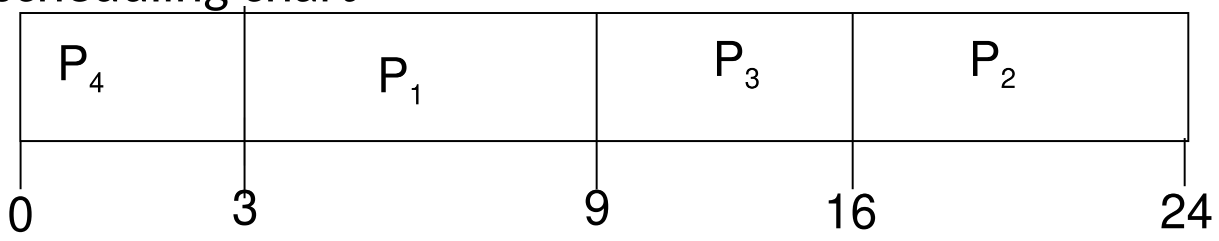
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Allocate CPU to a process with the smallest next CPU burst.
  - Not on the total CPU time
- Tie=>FCFS

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

  Avg waiting time for FCFS?

SJF

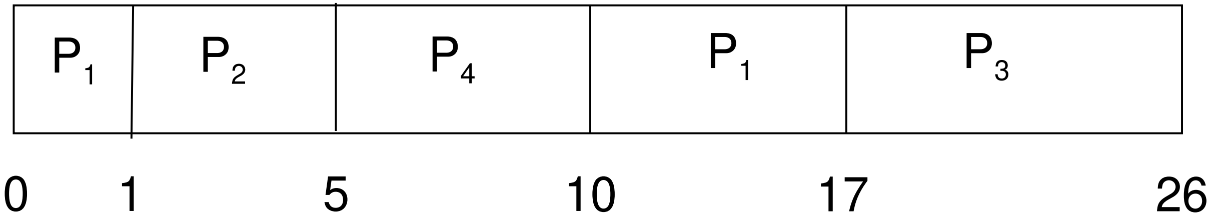
- SJF is optimal – gives minimum average waiting time for a given set of processes (Proof: home work!)
- The difficulty is knowing the length of the next CPU request
- Useful for Long term scheduler
  - Batch system
  - Could ask the user to estimate
  - Too low value may result in "time-limit-exceeded error"

Preemptive version
Shortest-remaining-time-first

- Preemptive version called **shortest-remaining-time-first**
- Concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



P_1 P_2 P_4 P_1 P_3

0 1 5 10 17 26

- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

Avg waiting time for non preemptive?

Determining Length of Next CPU Burst

- Estimation of the CPU burst length – should be similar to the previous burst
  - Then pick process with shortest predicted next CPU burst
- Estimation can be done by using the length of previous CPU bursts, using time series analysis

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

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

Boundary
cases $\alpha=0, 1$

- Commonly, α set to ½

Examples of Exponential Averaging

- ➔ =0
  - 🔀$_{n+1}$ = 🔀$_{n}$
  - Recent burst time does not count
- ➔ =1
  - 🔀$_{n+1}$ = t_{n}
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

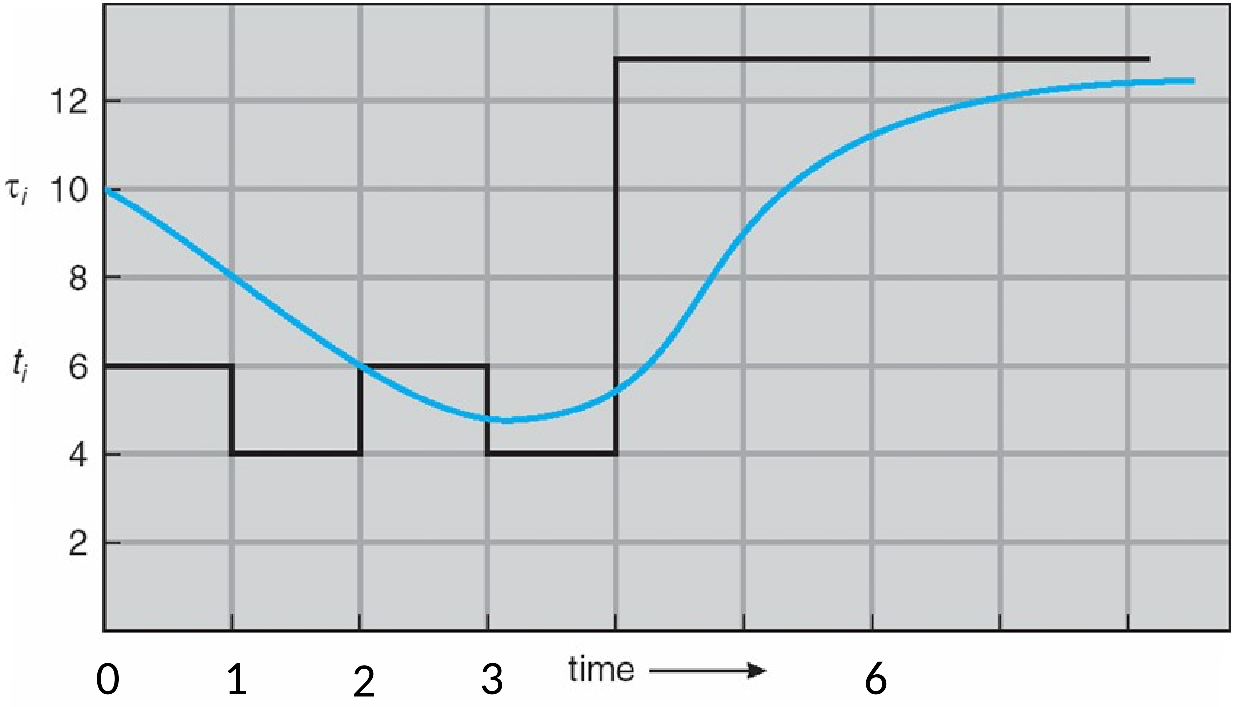
  🔀$_{n+1}$ = ➔ t_{n}+(1 - ➔)➔ t_{n-1} + …

  +(1 - ➔)^{j}➔ t_{n-j} + …

  +(1 - ➔)$^{n+1}$🔀$_{0}$

- Since both ➔ and (1 - ➔) are less than or equal to 1, each successive term has less weight than its predecessor

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer 📻 highest priority)

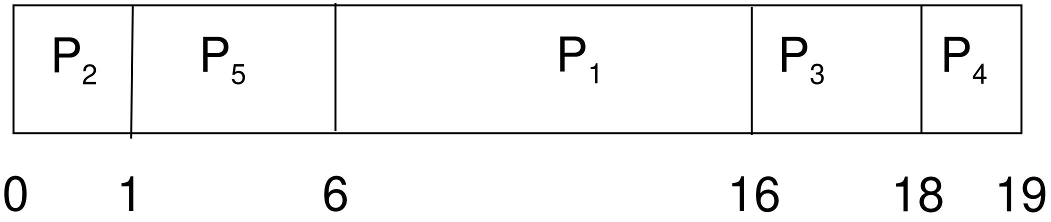
nice

- Set priority value
  - Internal (time limit, memory req., ratio of I/O Vs CPU burst)
  - External (importance, fund etc)
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Two types
  - Preemptive
  - Nonpreemptive
- Problem 📻 **Starvation** – low priority processes may never execute
- Solution 📻 **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



P_2 P_5 P_1 P_3 P_4

0 1 6 16 18 19

- Average waiting time = 8.2 msec

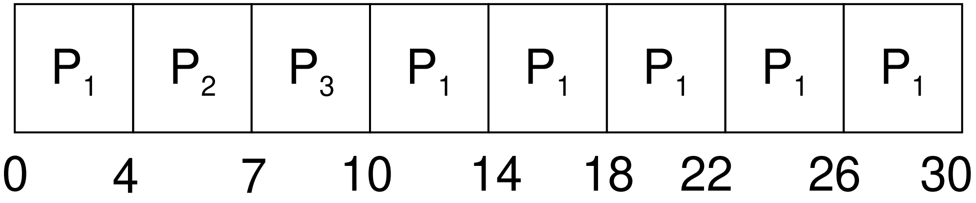
Round Robin (RR)

- Designed for time sharing system
- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Implementation
  - Ready queue as FIFO queue
  - CPU scheduler picks the first process from the ready queue
  - Sets the timer for 1 time quantum
  - Invokes despatcher
- If CPU burst time < quantum
  - Process releases CPU
- Else Interrupt
  - Context switch
  - Add the process at the tail of the ready queue
  - Select the front process of the ready queue and allocate CPU

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



P_1 P_2 P_3 P_1 P_1 P_1 P_1 P_1

0 4 7 10 14 18 22 26 30

- Avg waiting time = ((10-4)+4+7)/3=5.66

Round Robin (RR)

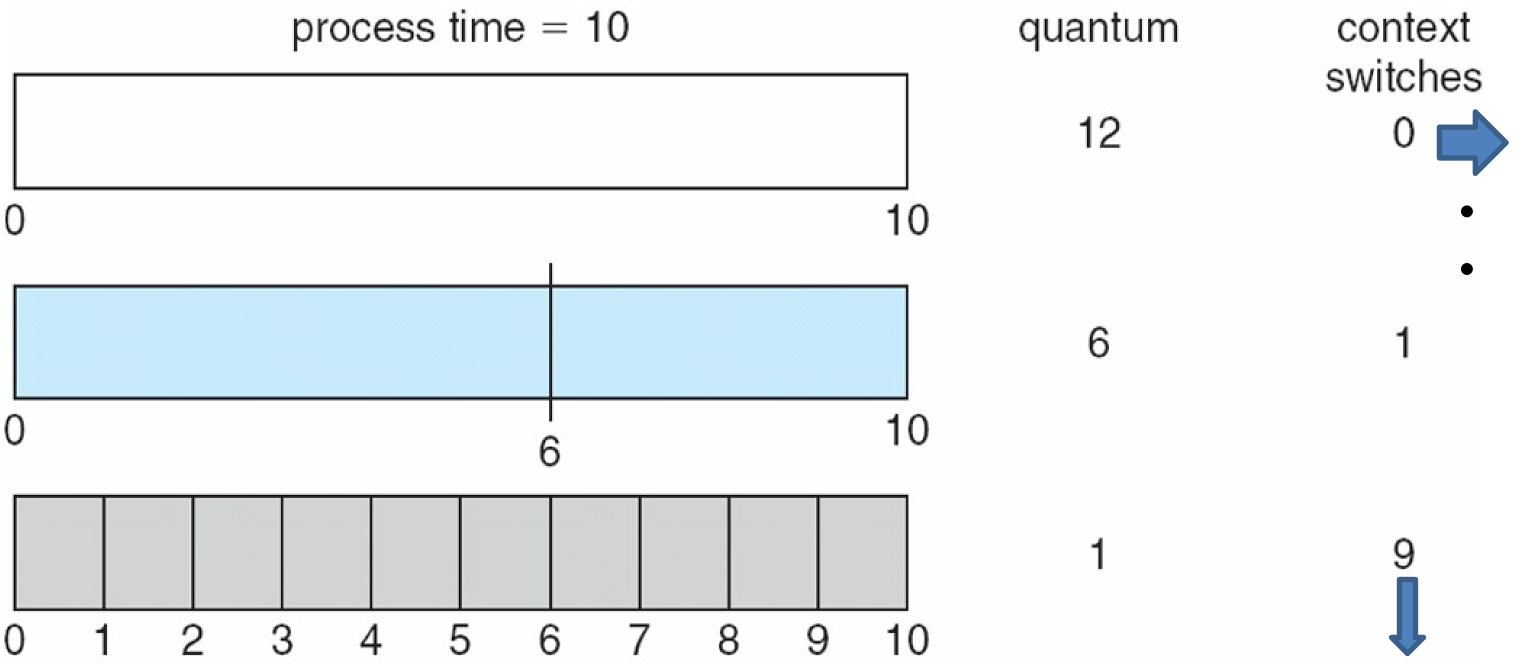
- Each process has a time quantum *T* allotted to it
- Dispatcher starts process P_0, loads a external counter (timer) with counts to count down from *T* to 0
- When the timer expires, the CPU is interrupted
- The context switch ISR gets invoked
- The context switch saves the context of P_0
  - PCB of P_0 tells where to save
- The scheduler selects P_1 from ready queue
  - The PCB of P_1 tells where the old state, if any, is saved
- The dispatcher loads the context of P_1
- The dispatcher reloads the counter (timer) with *T*
- The ISR returns, restarting P_1 (since P_1's PC is now loaded as part of the new context loaded)

Round Robin (RR)

- If there are *n* processes in the ready queue and the time quantum is *q*
  - then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once.
  - No process waits more than (*n*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance depends on time quantum q
  - *q* large ⏶ FIFO
  - *q* small ⏶ Processor sharing (n processes has own CPU running at 1/n speed)

Effect of Time Quantum and Context Switch Time

Performance of RR scheduling



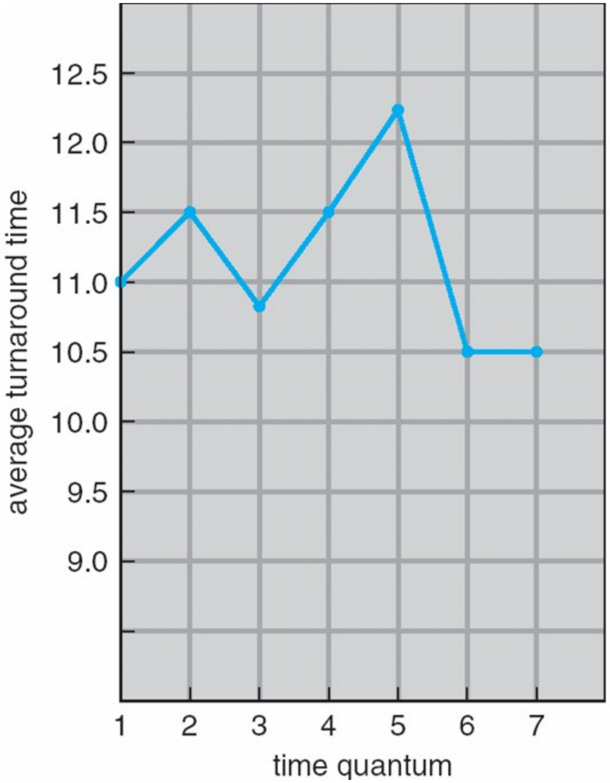
- No overhead
- However, poor response time

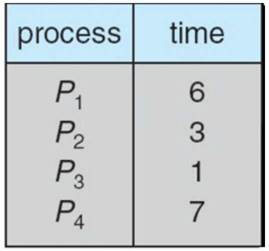
- Too much overhead!
- Slowing the execution time

- *q* must be large with respect to context switch, otherwise overhead is too high
- q usually 10ms to 100ms, context switch < 10 microsec

Effect on Turnaround Time

- TT depends on the time quantum and CPU burst time
  - Better if most processes complete there next CPU burst in a single q





process	time
P_1	6
P_2	3
P_3	1
P_4	7

- Large q=> processes in ready queue suffer
- Small q=> Completion will take more time

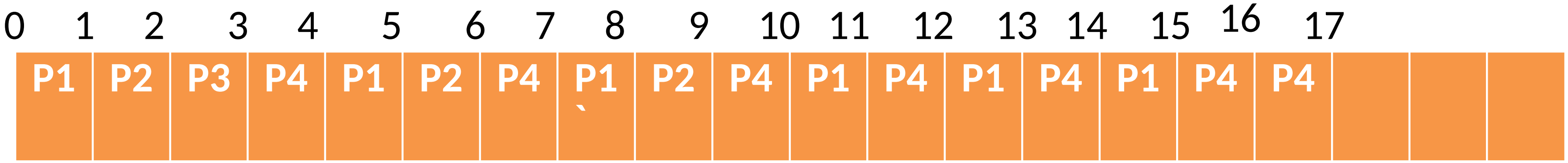
80% of CPU bursts should be shorter than q

Response time

Typically, higher average turnaround than SJF, but better *response time*

Turnaround Time

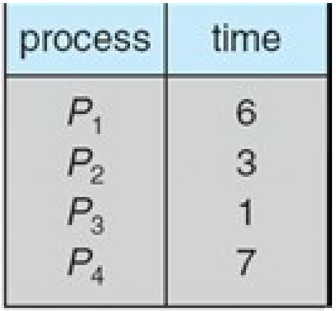
q=1



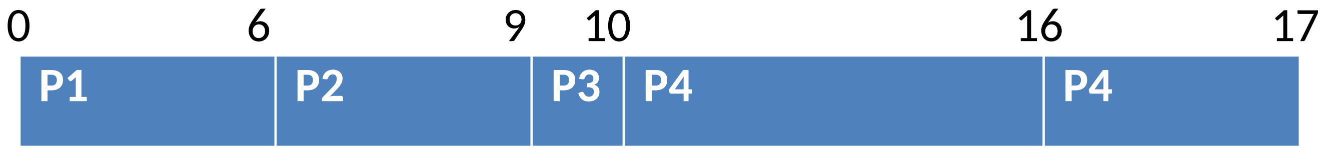
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

P1 P2 P3 P4 P1 P2 P4 P1 ` P2 P4 P1 P4 P1 P4 P1 P4 P4

Avg Turnaround time=
(15+9+3+17)/4=11



process	time
P_1	6
P_2	3
P_3	1
P_4	7



0 6 9 10 16 17

P1 P2 P3 P4 P4

q=6

(6+9+10+17)/4=10.5

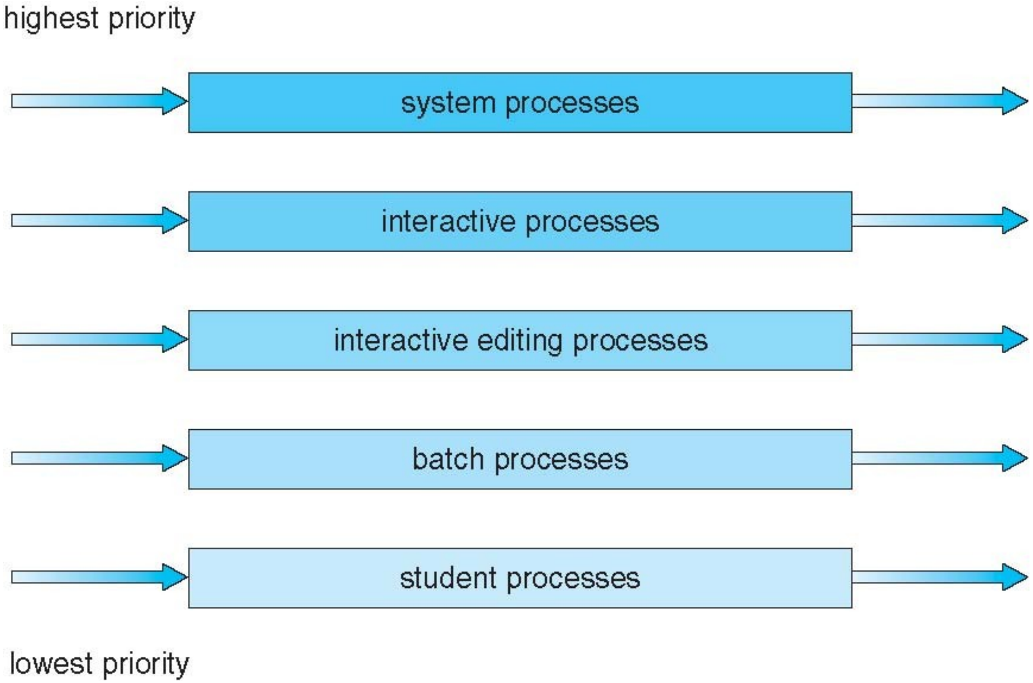
Process classification

- Foreground process
  - Interactive
  - Frequent I/O request
  - Requires low response time
- Background Process
  - Less interactive
  - Like batch process
  - Allows high response time
- Can use different scheduling algorithms for two types of processes ?

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - foreground (interactive)
  - background (batch)
- Process permanently assigned in a given queue
  - Based on process type, priority, memory req.
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background).
  - Possibility of starvation.

Multilevel Queue Scheduling



- No process in batch queue could run unless upper queues are empty

- If new process enters
  - Preempt

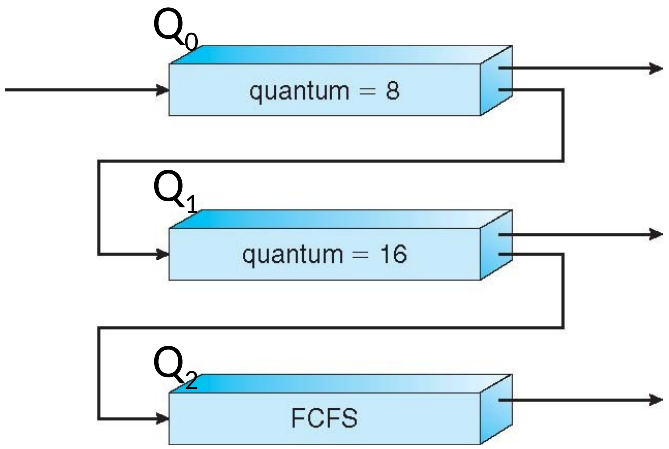
Another possibility

- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
- 20% to background in FCFS

Multilevel Feedback Queue

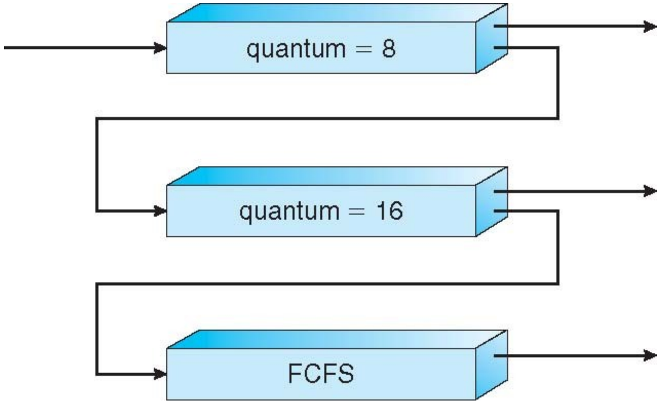
- So a process is permanently assigned a queue when they enter in the system
  - They do not move
- **Flexibility!**
  - Multilevel-feedback-queue scheduling
- A process can move between the various queues;
- Separate processes based of the CPU bursts
  - Process using too much CPU time can be moved to lower priority
  - Interactive process => Higher priority
- Move process from low to high priority
  - Implement aging

Example of Multilevel Feedback Queue



- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again receives 16 milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

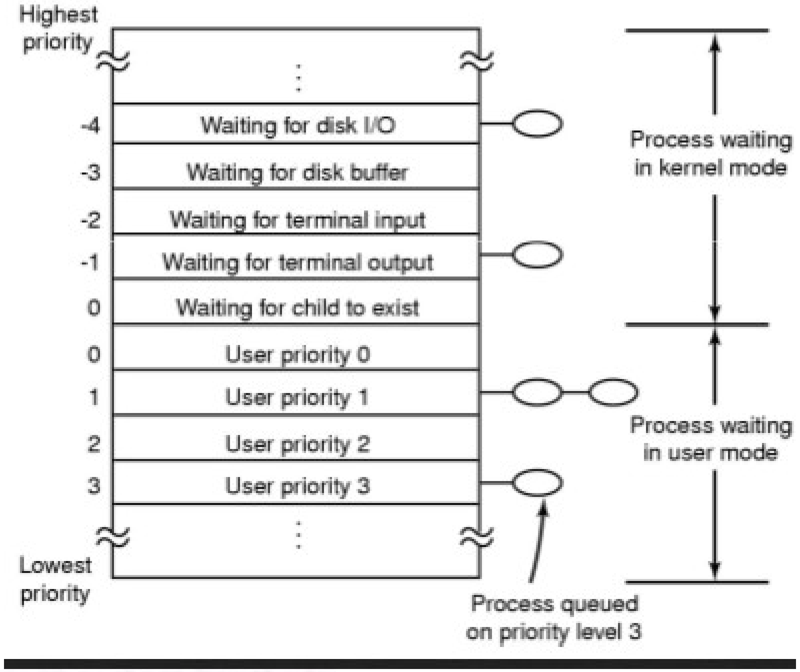
Multilevel Feedback Queues

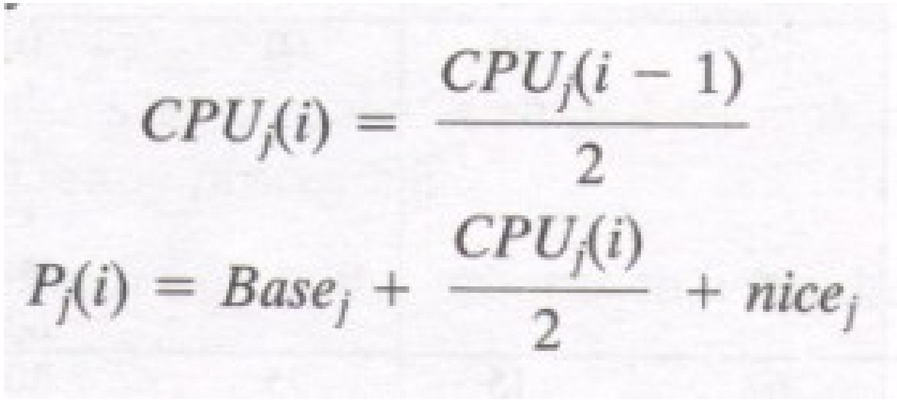


- Highest Priority to processes CPU burst time <8 ms
- Then processes >8 and <24

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

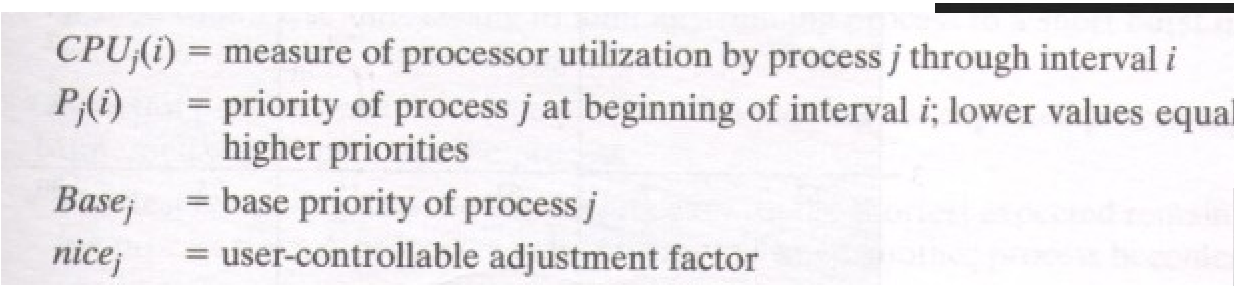
Unix scheduling


Highest priority
-4
-3
-2
-1
0
0
1
2
3
Waiting for disk I/O
Waiting for disk buffer
Waiting for terminal input
Waiting for terminal output
Waiting for child to exist
User priority 0
User priority 1
User priority 2
User priority 3
Lowest priority
Process waiting in kernel mode
Process waiting in user mode
Process queued on priority level 3



$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

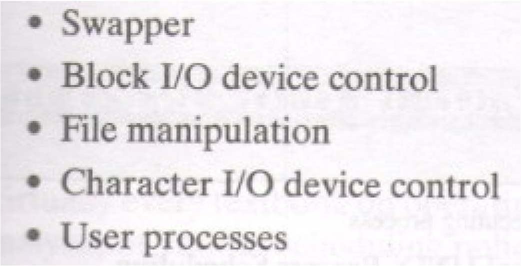


$CPU_j(i)$ = measure of processor utilization by process j through interval i

$P_j(i)$ = priority of process j at beginning of interval i; lower values equal higher priorities

$Base_j$ = base priority of process j

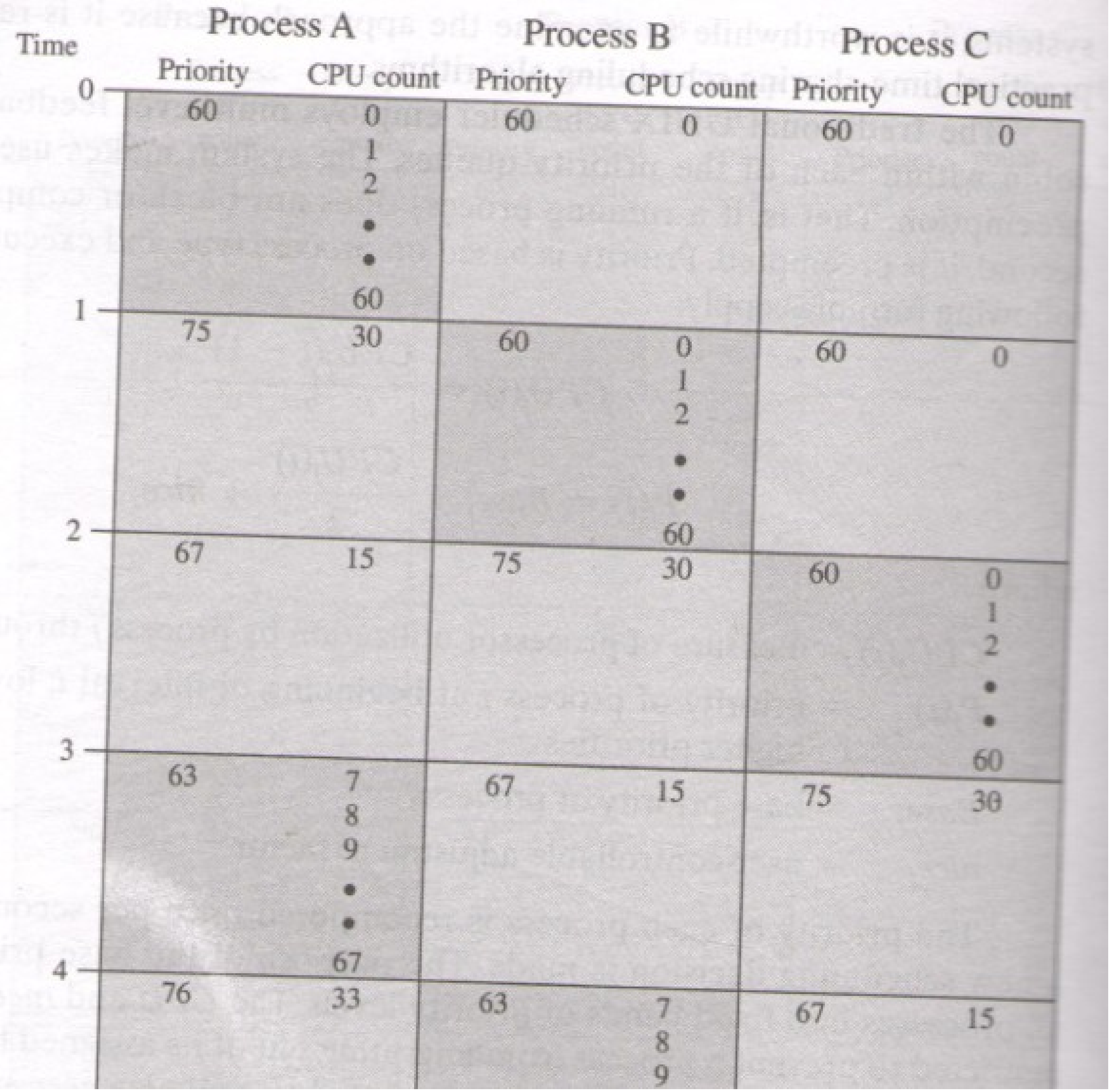
$nice_j$ = user-controllable adjustment factor



- Swapper
- Block I/O device control
- File manipulation
- Character I/O device control
- User processes

Time_critical, Highest, Above_Normal, normal, Below_Normal
Lowest, Idle

Unix scheduling



Time	Process A		Process B		Process C	
	Priority	CPU count	Priority	CPU count	Priority	CPU count
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9	67	15

Linux scheduler

Pre-emptive, priority based scheduling

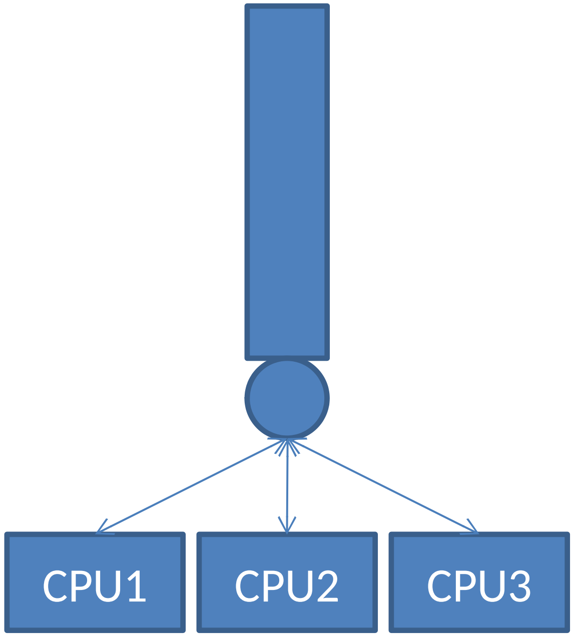
Linux version	**Scheduler**
Linux pre 2.5	Multilevel Feedback Queue
Linux 2.5-2.6.23	O(1) scheduler
Linux post 2.6.23	Completely Fair Scheduler

We will be talking about the O(1) scheduler

Runqueue in 2.4 and 2.6 versions

List of all runnable processes

2.4 Kernel

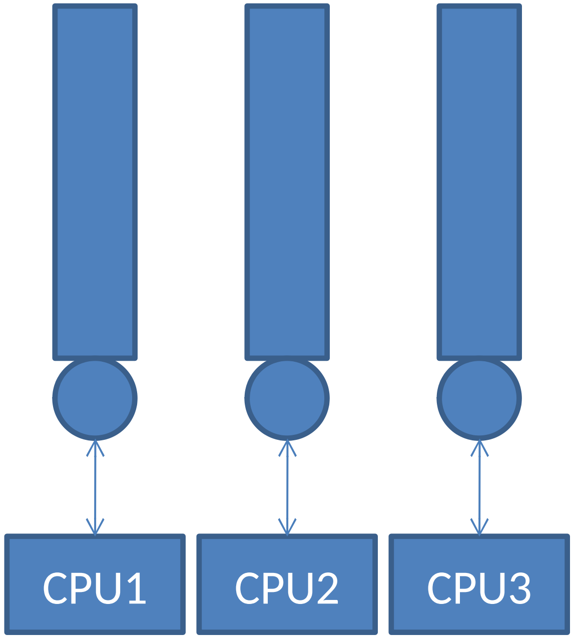


CPU1

CPU2

CPU3

2.6 Kernel

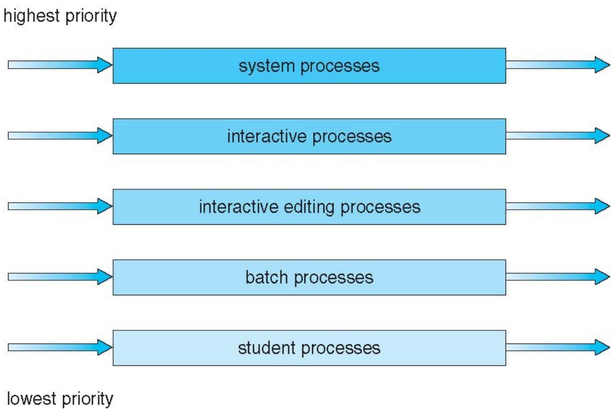


CPU1

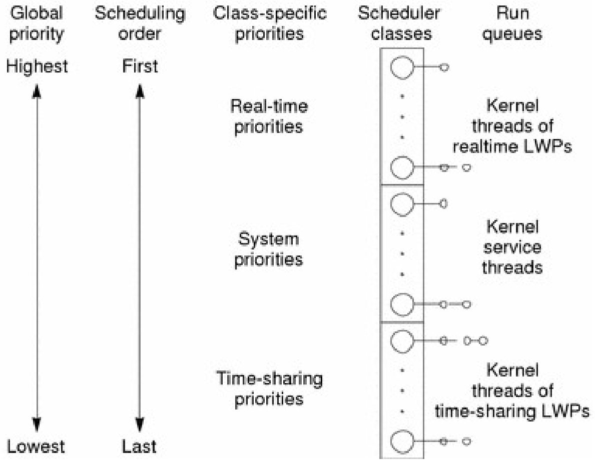
CPU2

CPU3

Linux Scheduling



- 3 scheduling classes
  - SCHED_FIFO and SCHED_RR are real-time classes
  - SCHED_OTHER is for the rest
- 140 Priority levels
  - 0-99 : RT priority
  - 100-139 : User task priorities
- Three different scheduling policies
  - One for normal tasks
  - Two for Real time tasks



Real-Time Policies

- First-in, first-out: SCHED_FIFO
  - Static priority
  - Process is only preempted for a higher-priority process
  - No time quanta; it runs until it blocks or yields voluntarily
- Round-robin: SCHED_RR
  - As above but with a time quanta (800 ms)
- Normal processes have SCHED_OTHER scheduling policy

Basic Philosophies

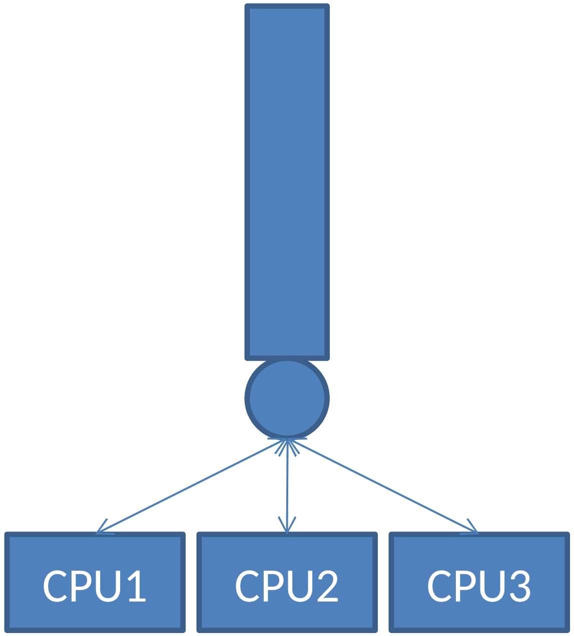
SCHED_OTHER

- Priority is the primary scheduling mechanism
- Priority is *dynamically adjusted* at run time
  - Processes denied access to CPU get increased
  - Processes running a long time get decreased
- Try to distinguish interactive processes from non-interactive
  - Bonus or penalty reflecting whether I/O or compute bound
- Use large quanta for important processes
  - Modify quanta based on CPU use
- Associate processes to CPUs
- Do everything in O(1) time

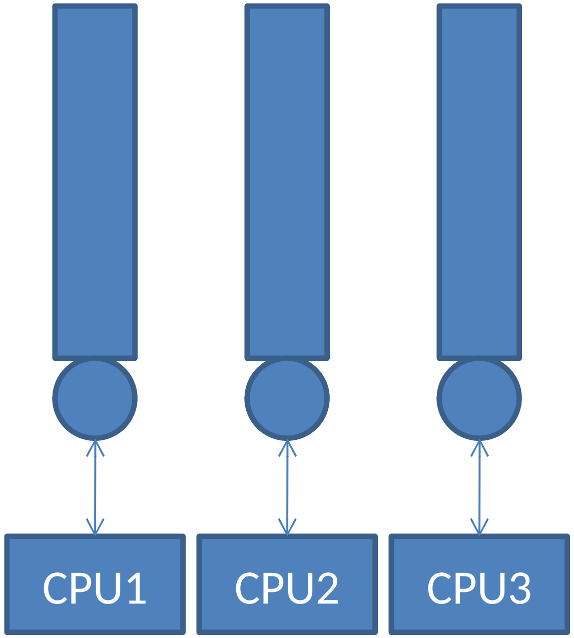
Runqueue in 2.4 and 2.6 versions

List of all runnable processes

2.4 Kernel



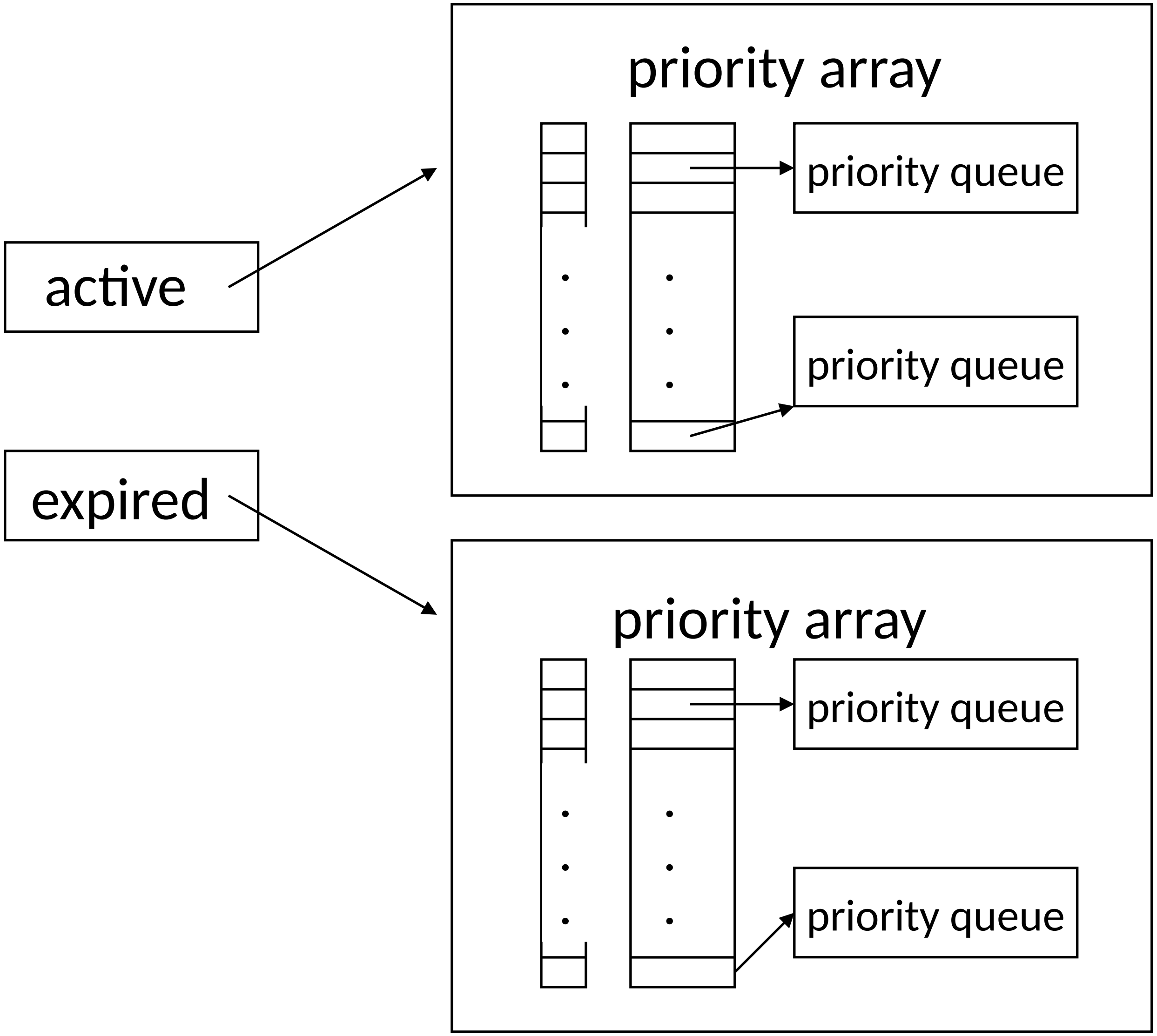
2.6 Kernel



Runqueue

```
struct runqueue {
spinlock_t lock; /* spin lock which protects this runqueue */
unsigned long nr_running; /* number of runnable tasks */
unsigned long nr_switches; /* number of context switches */
unsigned long expired_timestamp; /* time of last array swap */
struct task_struct *curr; /* this processor's currently running task */
struct task_struct *idle; /* this processor's idle task */
struct mm_struct *prev_mm; /* mm_struct of last running task */
struct prio_array *active; /* pointer to the active priority array */
struct prio_array *expired; /* pointer to the expired priority array */
struct prio_array arrays[2]; /* the actual priority arrays */
int prev_cpu_load[NR_CPUS];/* load on each processor */
struct list_head *migration_queue; /* the migration queue for this processor */
atomic_t nr_iowait; /* number of tasks waiting on I/O */ }
```

Runqueue for O(1) Scheduler



active

expired

priority array

priority queue

priority queue

priority array

priority queue

priority queue

Higher priority
more I/O
800ms quanta

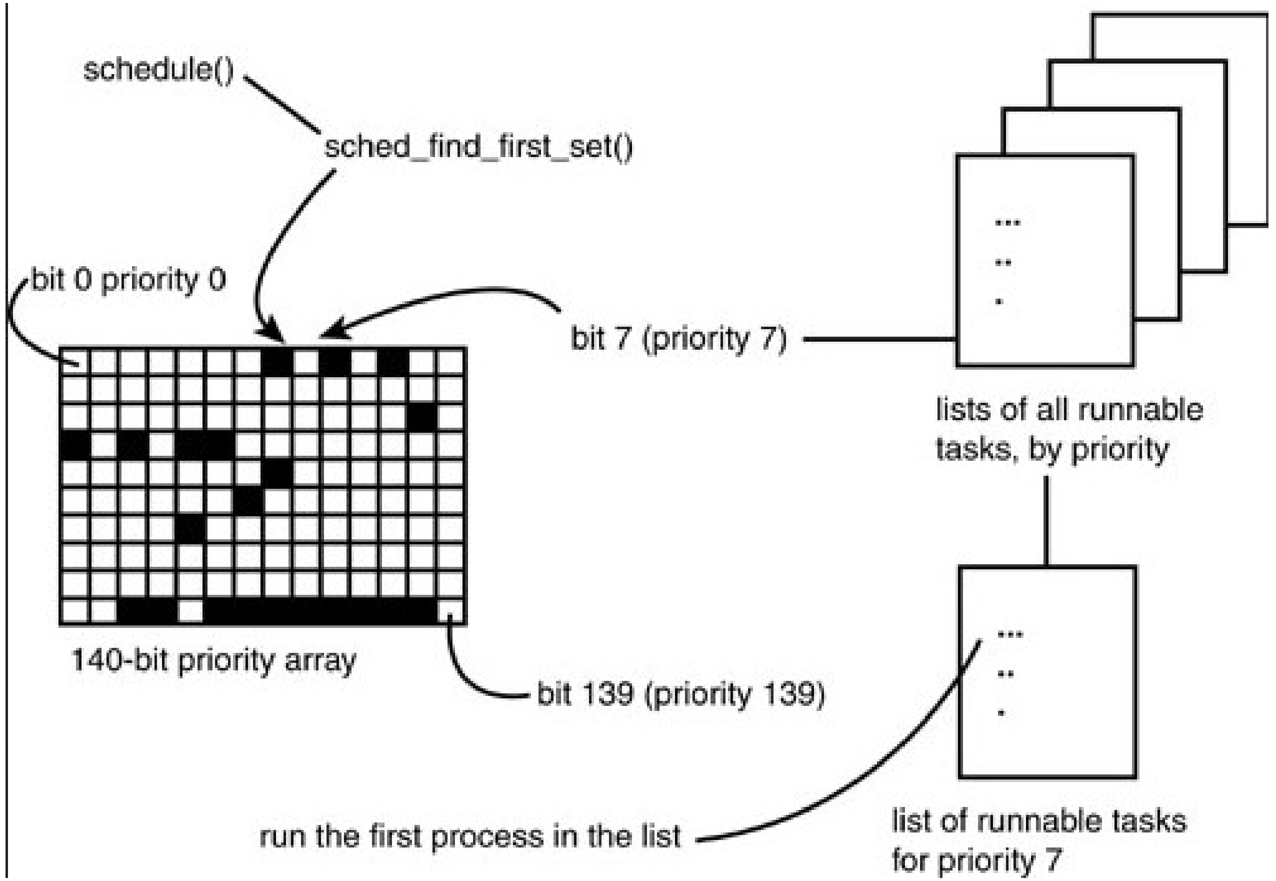
lower priority
more CPU
10ms quanta

The Runqueue

- 140 separate queues, one for each priority level
- Actually, two sets, active and expired
- Priorities 0-99 for real-time processes
- Priorities 100-139 for normal processes; value set via nice()/setpriority() system calls

The Priority Arrays

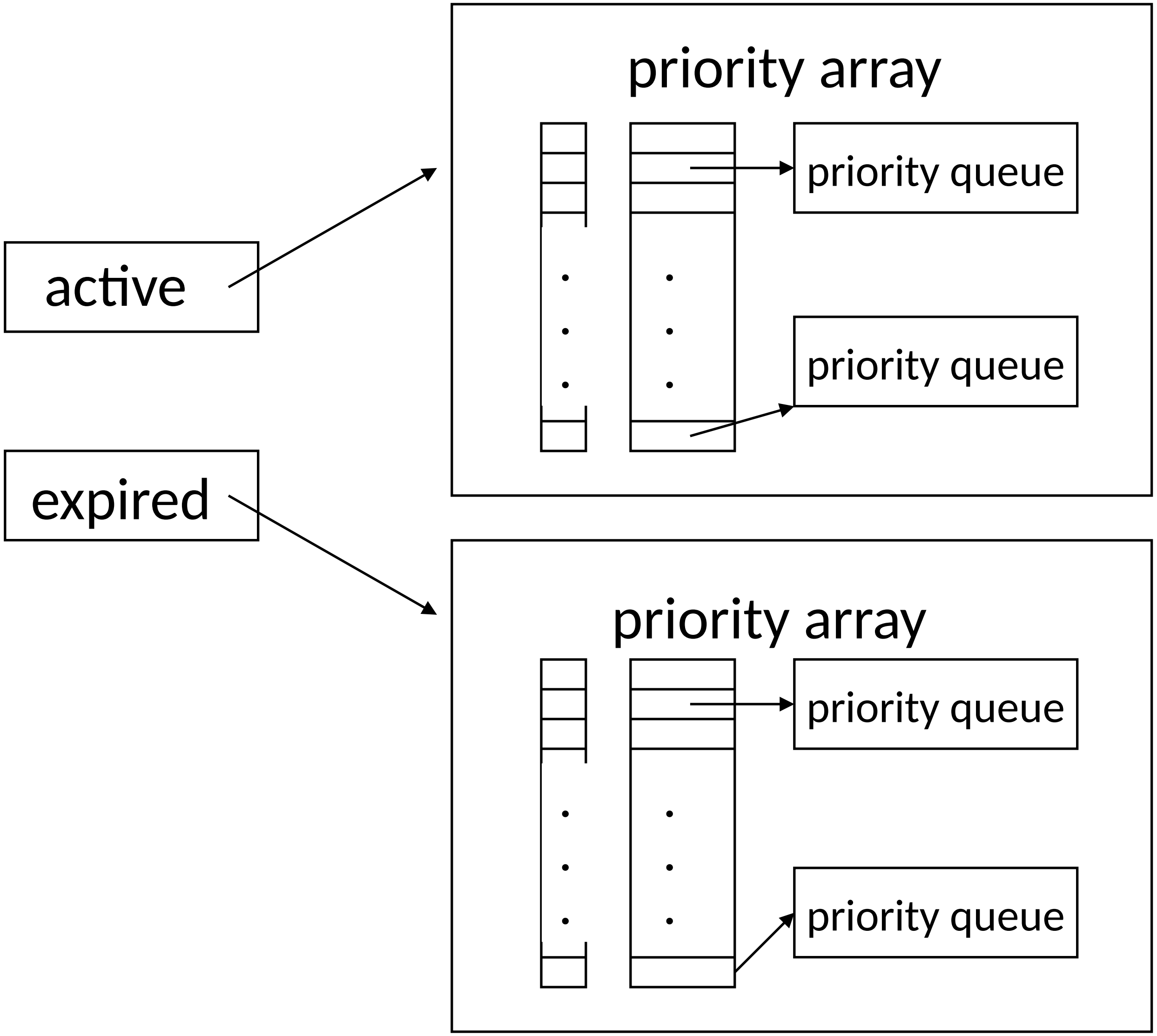
```
struct prio_array {
     unsigned int nr_active;
     unsigned long bitmap[5];
     struct list_head queue[140];
};
```


schedule()
sched_find_first_set()
bit 0 priority 0
bit 7 (priority 7)
140-bit priority array
bit 139 (priority 139)
lists of all runnable tasks, by priority
run the first process in the list
list of runnable tasks for priority 7

Basic Scheduling Algorithm

- Find the highest-priority queue with a runnable process
- Find the first process on that queue
- Get its quantum size
- Let it run
- When its time is up, put it on the expired list
  - Recalculate priority/quantam first
- Repeat

Runqueue for O(1) Scheduler



active

expired

priority array

priority queue

priority queue

priority array

priority queue

priority queue

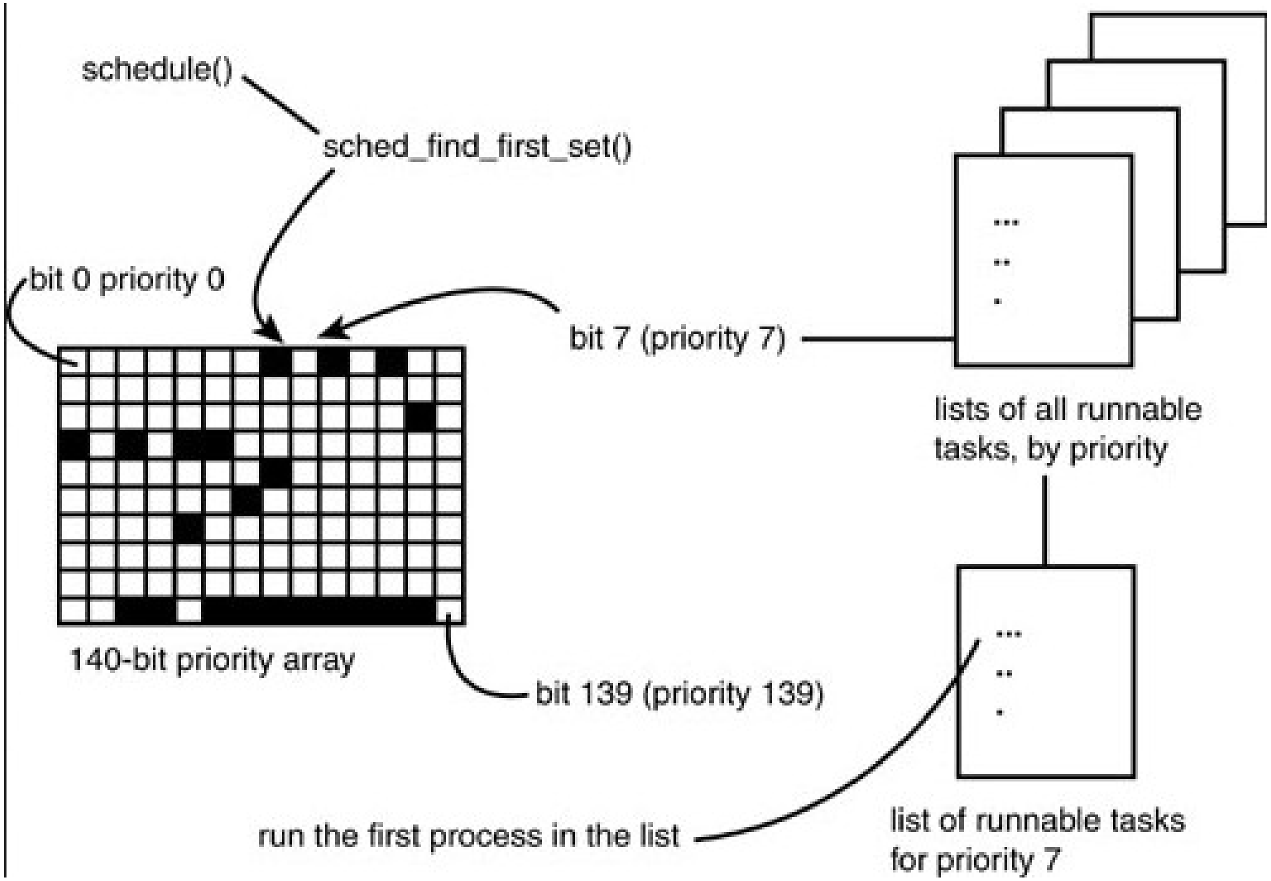
Higher priority
more I/O
800ms quanta

lower priority
more CPU
10ms quanta

The Highest Priority Process

- There is a bit map indicating which queues have processes that are ready to run
- Find the first bit that’s set:
  - 140 queues ✉ 5 integers
  - Only a few compares to find the first that is non-zero
  - Hardware instruction to find the first 1-bit
    - bsfl on Intel
  - Time depends on the number of priority levels, not the number of processes

The Highest Priority Process


schedule()
sched_find_first_set()
bit 0 priority 0
bit 7 (priority 7)
140-bit priority array
bit 139 (priority 139)
lists of all runnable tasks, by priority
list of runnable tasks for priority 7
run the first process in the list

The Highest Priority Process

```
struct task_struct *prev, *next;
struct list_head *queue;
struct prio_array array;
int idx;
prev = current;
array = rq->active;
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next);
```

Swapping Arrays

```
struct prioarray *array =
    rq->active;
if (array->nr_active == 0) {
    rq->active = rq->expired;
    rq->expired = array;
}
```

Runqueue

```
struct runqueue {
spinlock_t lock; /* spin lock which protects this runqueue */
unsigned long nr_running; /* number of runnable tasks */
unsigned long nr_switches; /* number of context switches */
unsigned long expired_timestamp; /* time of last array swap */
struct task_struct *curr; /* this processor's currently running task */
struct task_struct *idle; /* this processor's idle task */
struct mm_struct *prev_mm; /* mm_struct of last running task */
struct prio_array *active; /* pointer to the active priority array */
struct prio_array *expired; /* pointer to the expired priority array */
struct prio_array arrays[2]; /* the actual priority arrays */
int prev_cpu_load[NR_CPUS];/* load on each processor */
struct list_head *migration_queue; /* the migration queue for this processor */
atomic_t nr_iowait; /* number of tasks waiting on I/O */ }
```

Scheduler Runqueue

- A scheduler runqueue is a list of tasks that are runnable on a particular CPU.
- A *rq* structure maintains a linked list of those tasks.
- The runqueues are maintained as an array *runqueues*, indexed by the CPU number.
- The *rq* keeps a reference to its idle task
  - The idle task for a CPU is never on the scheduler runqueue for that CPU (it's always the last choice)
- Access to a runqueue is serialized by acquiring and releasing *rq->lock*

Calculating Time Slices

- *time_slice* in the *task_struct*
- Calculate Quantum where
  - If (SP < 120): Quantum = (140 − SP) × 20
  - if (SP >= 120): Quantum = (140 − SP) × 5

  where SP is the *static priority*
- Higher priority process get longer quanta
- Basic idea: important processes should run longer

Static Priority

- Each task has a static priority that is set based upon the nice value specified by the task.
  - *static_prio* in *task_struct*
- The nice value is in a range of -19 to 20, with the default value being 0. Only privileged tasks can set the nice value below 0.
- For normal tasks, the static priority is base priority + the nice value.
- Each task has a dynamic priority that is set based upon a number of factors

Nice Value vs. static priority and Quantum

	Static Priority	NICE	Quantum
High Priority	100	-20	800 ms
	110	-10	600 ms
	120	0	100 ms
	130	+10	50 ms
Low Priority	139	+19	5 ms

$$\text{Quantum} = \begin{cases} (140 - \text{SP}) \times 20 & \text{if SP} < 120 \\ (140 - \text{SP}) \times 5 & \text{if SP} \geq 120 \end{cases}$$

Interactive Processes

- A process is considered interactive if

  bonus − 5 >= (Static Priority / 4) − 28

- Low-priority processes have a hard time becoming interactive:
  - A high static priority (100) becomes interactive when its average sleep time is greater than 200 ms
  - A default static priority process becomes interactive when its sleep time is greater than 700 ms
  - Lowest priority (139) can never become interactive
- The higher the bonus the task is getting and the higher its static priority, the more likely it is to be considered interactive.

Using Quanta

- At every time tick (in *scheduler_tick*) , decrement the quantum of the current running process (*time_slice*)
- If the time goes to zero, the process is done
- Check interactive status:
  - If non-interactive, put it aside on the *expired* list
  - If interactive, put it at the end of the *active* list
- Exceptions: don't put on *active* list if:
  - If higher-priority process is on *expired* list
  - If expired task has been waiting more than *STARVATION_LIMIT*
- If there's nothing else at that priority, it will run again immediately
- Of course, by running so much, its bonus will go down, and so will its priority and its interactive status

Avoiding Starvation

```
struct task_struct *task = current;
struct runqueue *rq = this_rq();
if (!—task->time_slice)
{ if (!TASK_INTERACTIVE(task) ||
EXPIRED_STARVING(rq))
    enqueue_task(task, rq->expired);
else
    enqueue_task(task, rq->active); }
```

Bonus, Sleep Average (I/O Wait)

- Interactivity heuristic: sleep ratio
  - Mostly sleeping: I/O bound
  - Mostly running: CPU bound
- Sleep ratio approximation
  - *sleep_avg* in the *task_struct*
  - Range: 0 .. *MAX_SLEEP_AVG*
- When process wakes up (is made runnable), *recalc_task_prio* adds in how many ticks it was sleeping (blocked), up to some maximum value (*MAX_SLEEP_AVG*)
- When process is switched out, *schedule* subtracts the number of ticks that a task actually ran (without blocking)
- *sleep_avg* scaled to a bonus value

Average Sleep Time and Bonus Values

Average sleep time	**Bonus**
>= 0 but < 100 ms	0
>= 100 ms but < 200 ms	1
>= 200 ms but < 300 ms	2
>= 300 ms but < 400 ms	3
>= 400 ms but < 500 ms	4
>= 500 ms but < 600 ms	5
>= 600 ms but < 700 ms	6
>= 700 ms but < 800 ms	7
>= 800 ms but < 900 ms	8
>= 900 ms but < 1000 ms	9
1 second	10

Multiprocessor Scheduling

- Each processor has a separate run queue
- Each processor only selects processes from its own queue to run
- Yes, it’s possible for one processor to be idle while others have jobs waiting in their run queues
- Periodically, the queues are rebalanced: if one processor’s run queue is too long, some processes are moved from it to another processor’s queue

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data

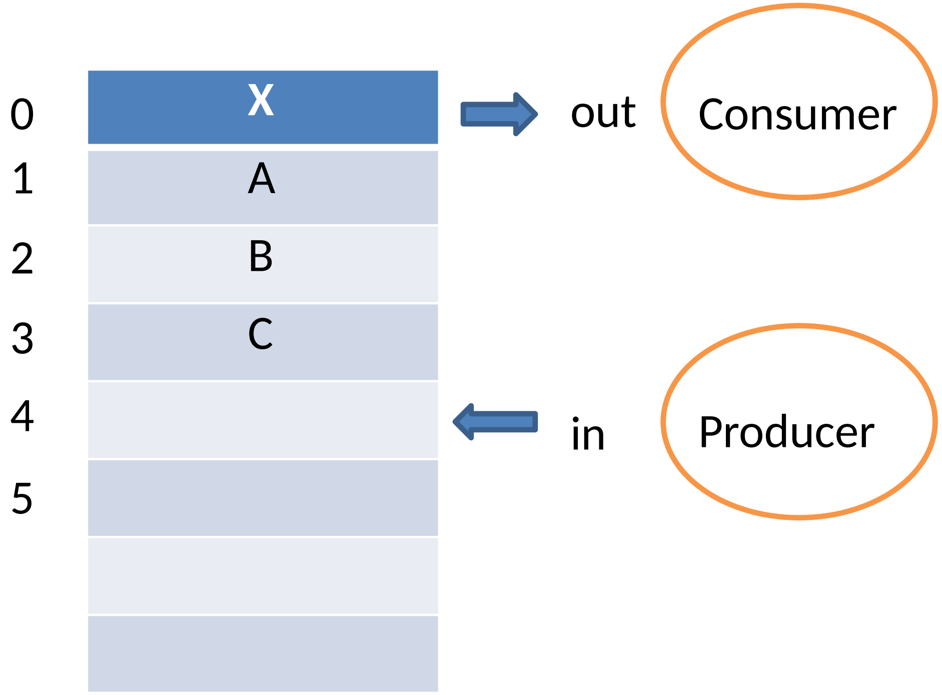
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

Buffer empty=>
in=out

Buffer full=>
(in+1)%size=out



- *unbounded-buffer* places no practical limit on the size of the buffer
- *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer –
Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded-Buffer – Producer

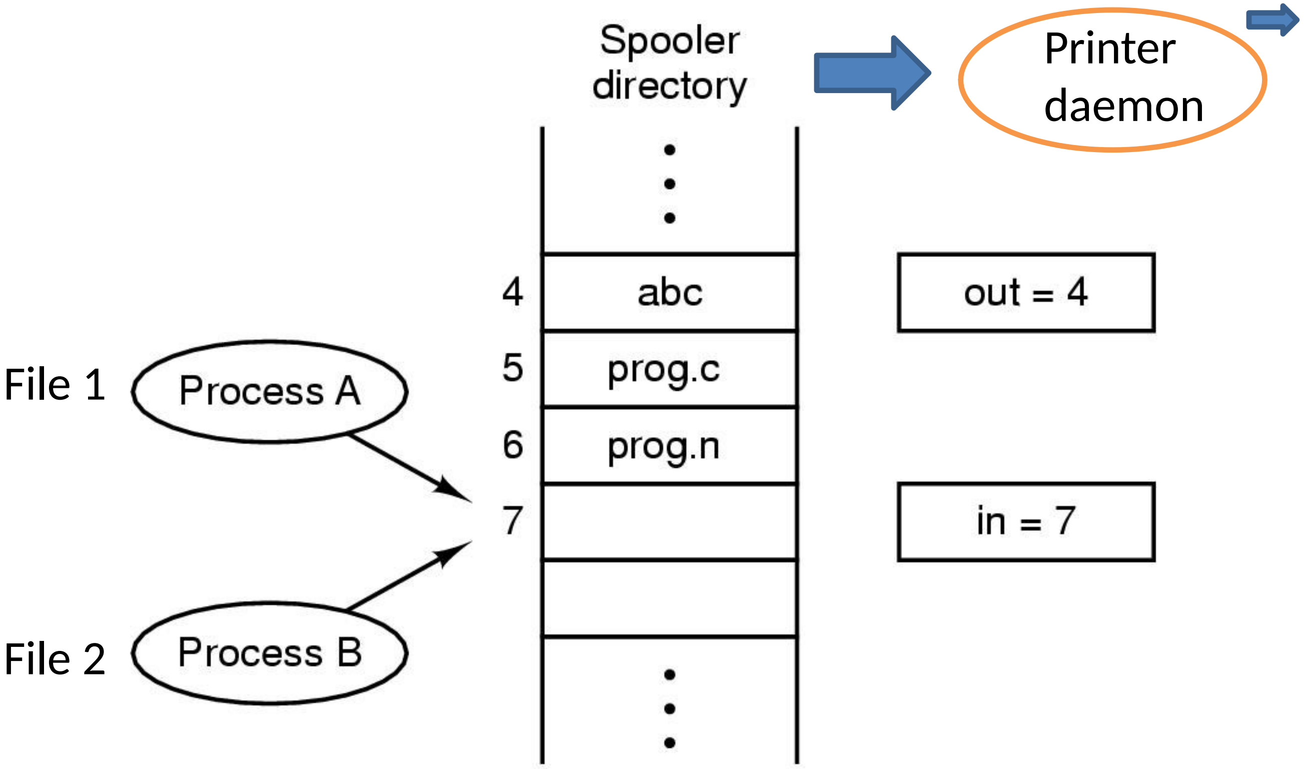
```
while (true) {
  /* Produce an item */
   while (((in + 1) % BUFFER SIZE)  == out)
    ;   /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
 }
```

Bounded Buffer – Consumer

```
while (true) {
     while (in == out)
            ; // do nothing -- nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
return item;
 }
```

Interprocess Communication

Race Conditions



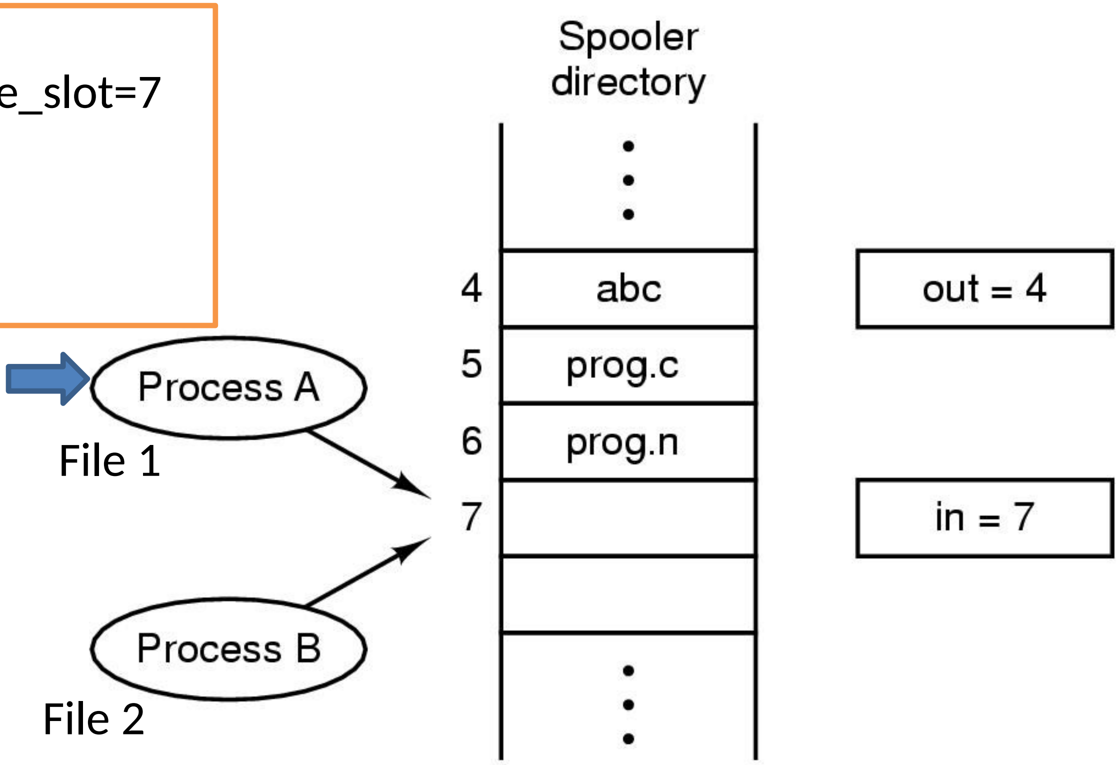
Two processes want to access shared memory at same time

Interprocess Communication

Race Conditions

Process A

Next_free_slot=7



File 1

File 2

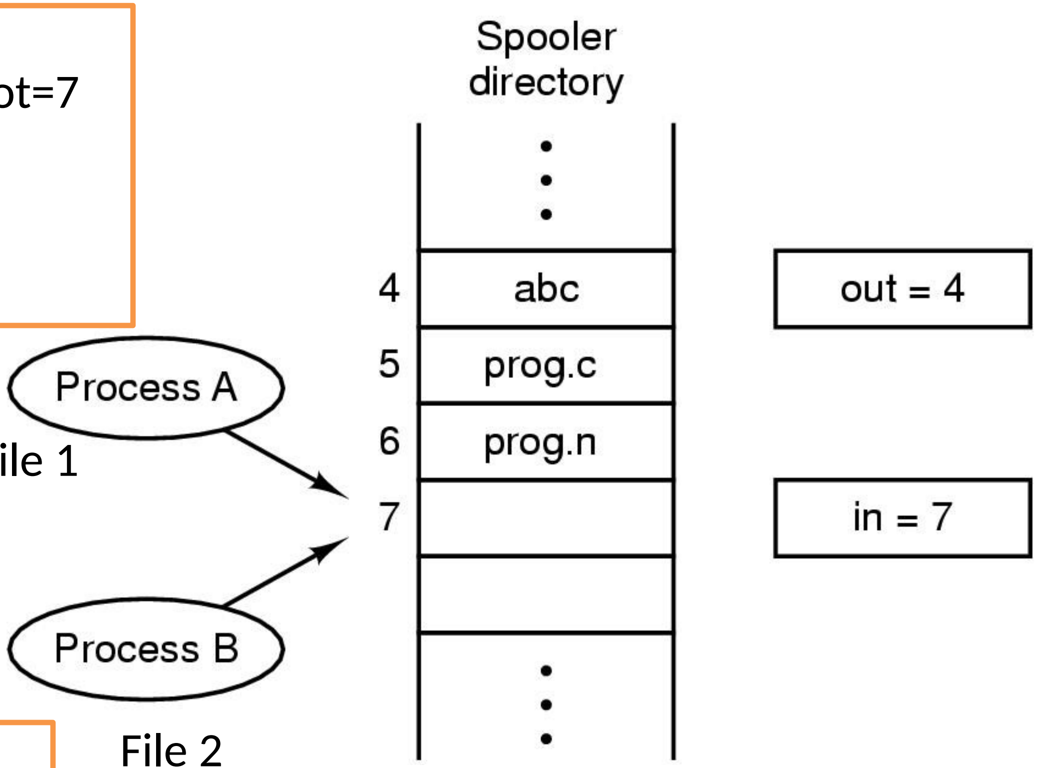
Two processes want to access shared memory at same time

Interprocess Communication

Race Conditions

Process A

Next_free_slot=7



File 1

Process B

Next_free_slot=7

File 2

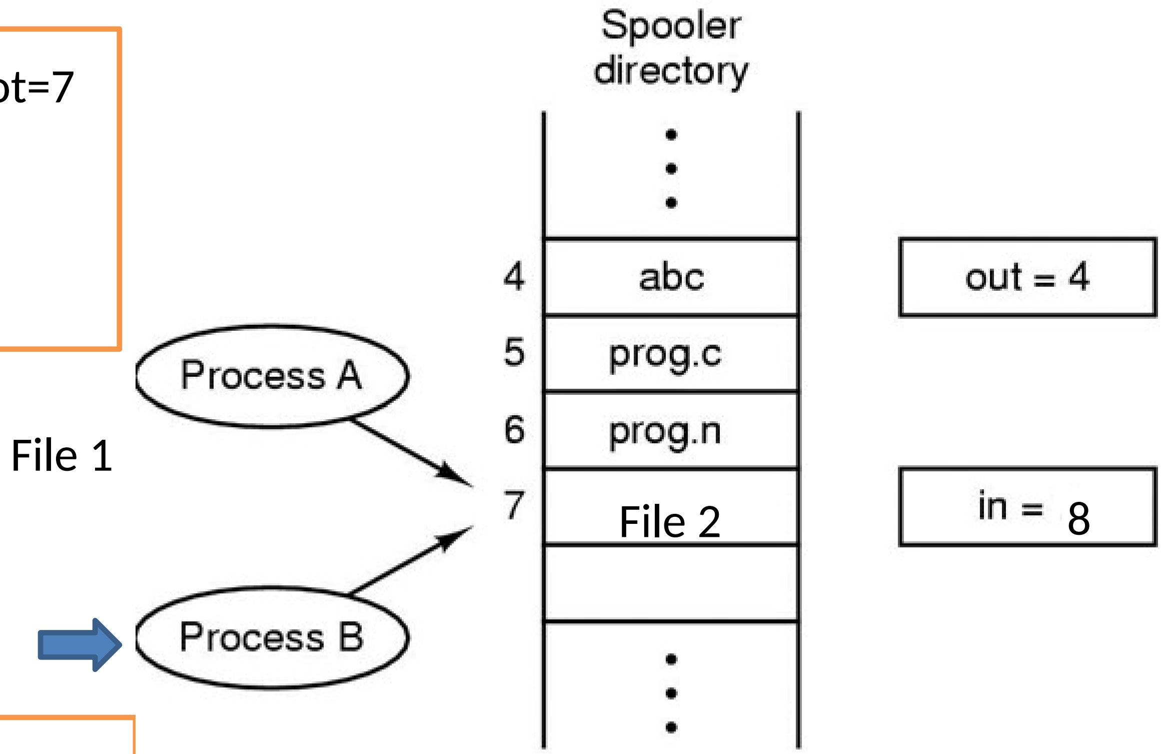
Two processes want to access shared memory at same time

Interprocess Communication

Race Conditions

Process A

Next_free_slot=7



Process B

Next_free_slot=8

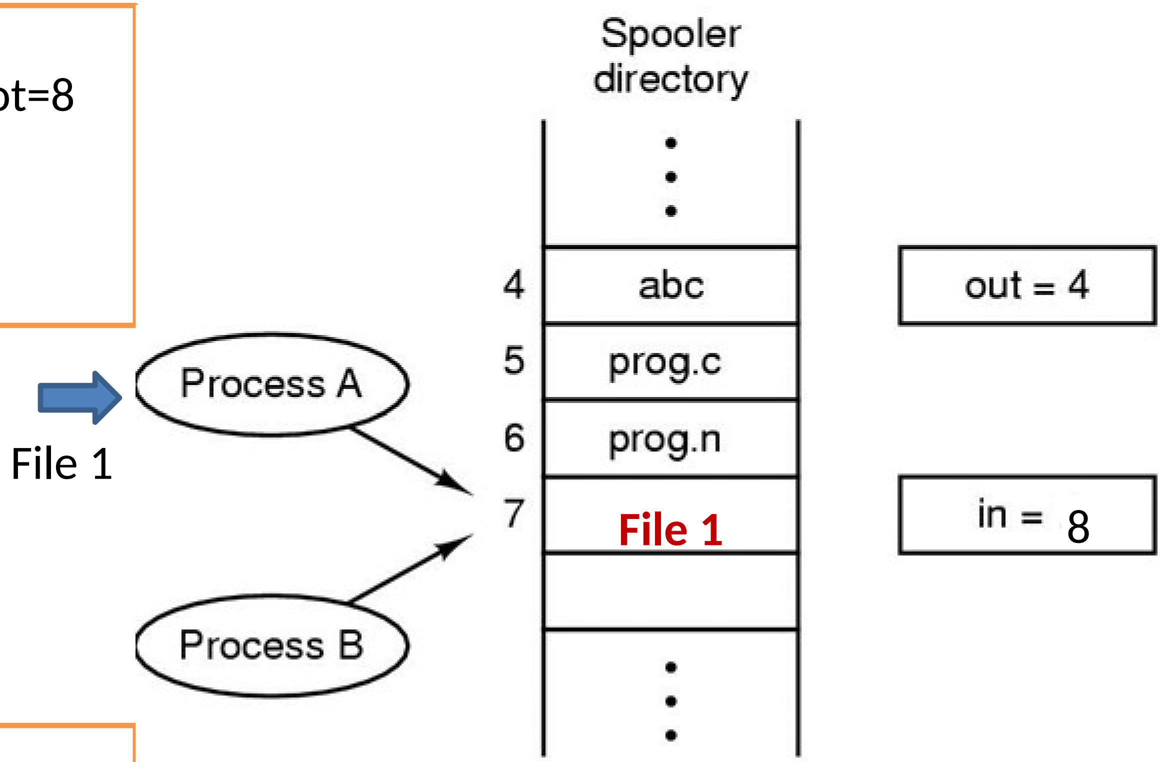
Two processes want to access shared memory at same time

Interprocess Communication

Race Conditions

Process A

Next_free_slot=8



Next_free_slot=8

Race condition

Two processes want to access shared memory at same time

Race condition

- Race condition
  - Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
  - In our former example, the possibilities are various
  - Hard to debug

Critical Section Problem

- Critical region
  - Part of the program where the shared memory is accessed

- Mutual exclusion
  - Prohibit more than one process from reading and writing the shared data at the same time

Critical Section Problem

- Consider system of n processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section Problem

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

General structure of a typical process P_i

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress** –

- If no process is executing in its critical section
- and there exist some processes that wish to enter their critical section
- then only the processes outside remainder section (i.e. the processes competing for critical section, or exit section) can participate in deciding which process will enter CS next

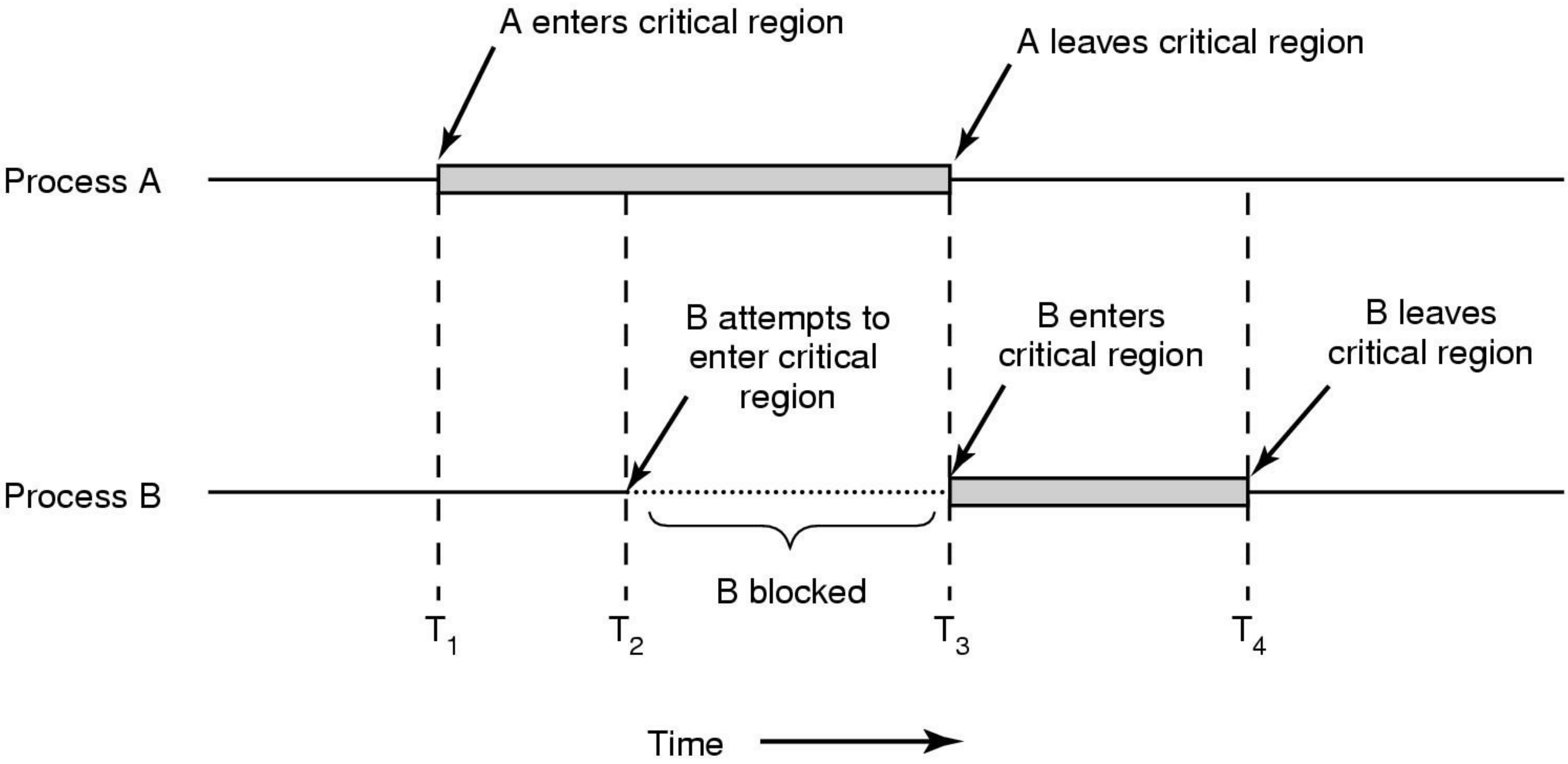
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

Critical Section Problem

```
do {
     entry section

          critical section

     exit section

          remainder section

} while (TRUE);
```

General structure of a typical process P_i

Critical Section Problem



Mutual exclusion using critical regions

Mutual Exclusion

- Disable interrupt
  - After entering critical region, disable all interrupts
  - Since clock is just an interrupt, no CPU preemption can occur
  - Disabling interrupt is useful for OS itself, but not for users…

Mutual Exclusion with busy waiting

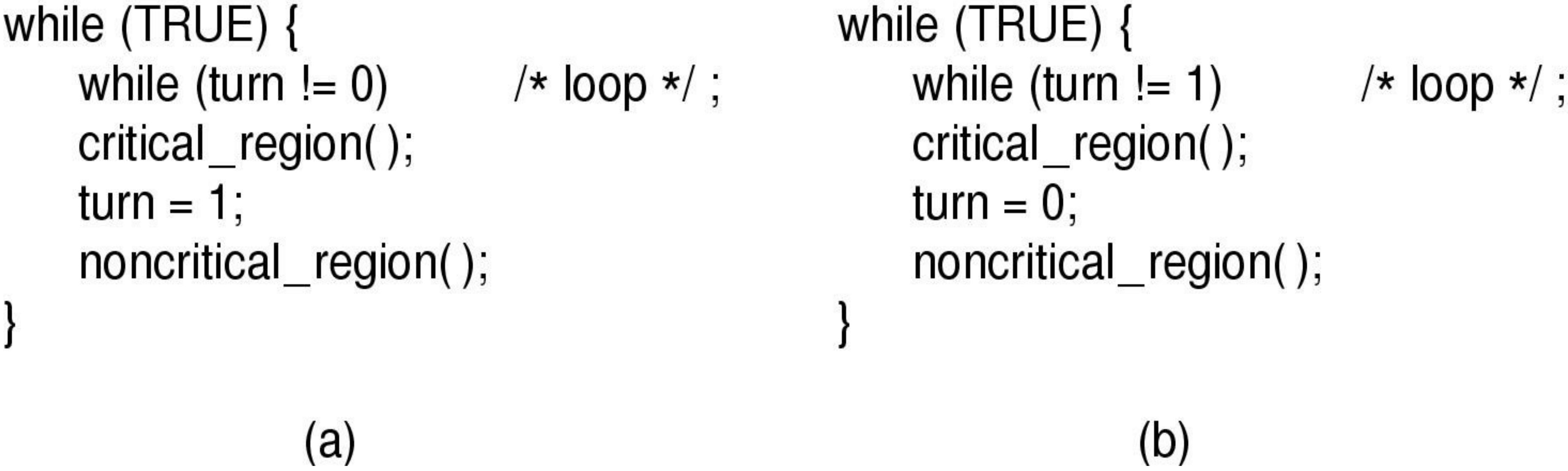
- Lock variable
  - A software solution
  - A single, shared variable (lock)
  - before entering critical region, programs test the variable,
  - if 0, enter CS;
  - if 1, the critical region is occupied
  - **What is the problem?**

```
While(true)
{
        while(lock!=0);
        Lock=1
        CS()
        Lock=0
        Non-CS()
}
```

Concepts

- Busy waiting
  - Continuously testing a variable until some value appears
- Spin lock
  - A lock using busy waiting is call a spin lock
- CPU time wastage!

Mutual Exclusion with Busy Waiting : strict alternation



```
while (TRUE) {
     while (turn != 0)          /* loop */ ;
     critical_region( );
     turn = 1;
     noncritical_region( );
}
```

(a)

```
while (TRUE) {
     while (turn != 1)          /* loop */ ;
     critical_region( );
     turn = 0;
     noncritical_region( );
}
```

(b)

Proposed solution to critical region problem

(a) Process 0.   (b) Process 1.

Peterson’s Solution

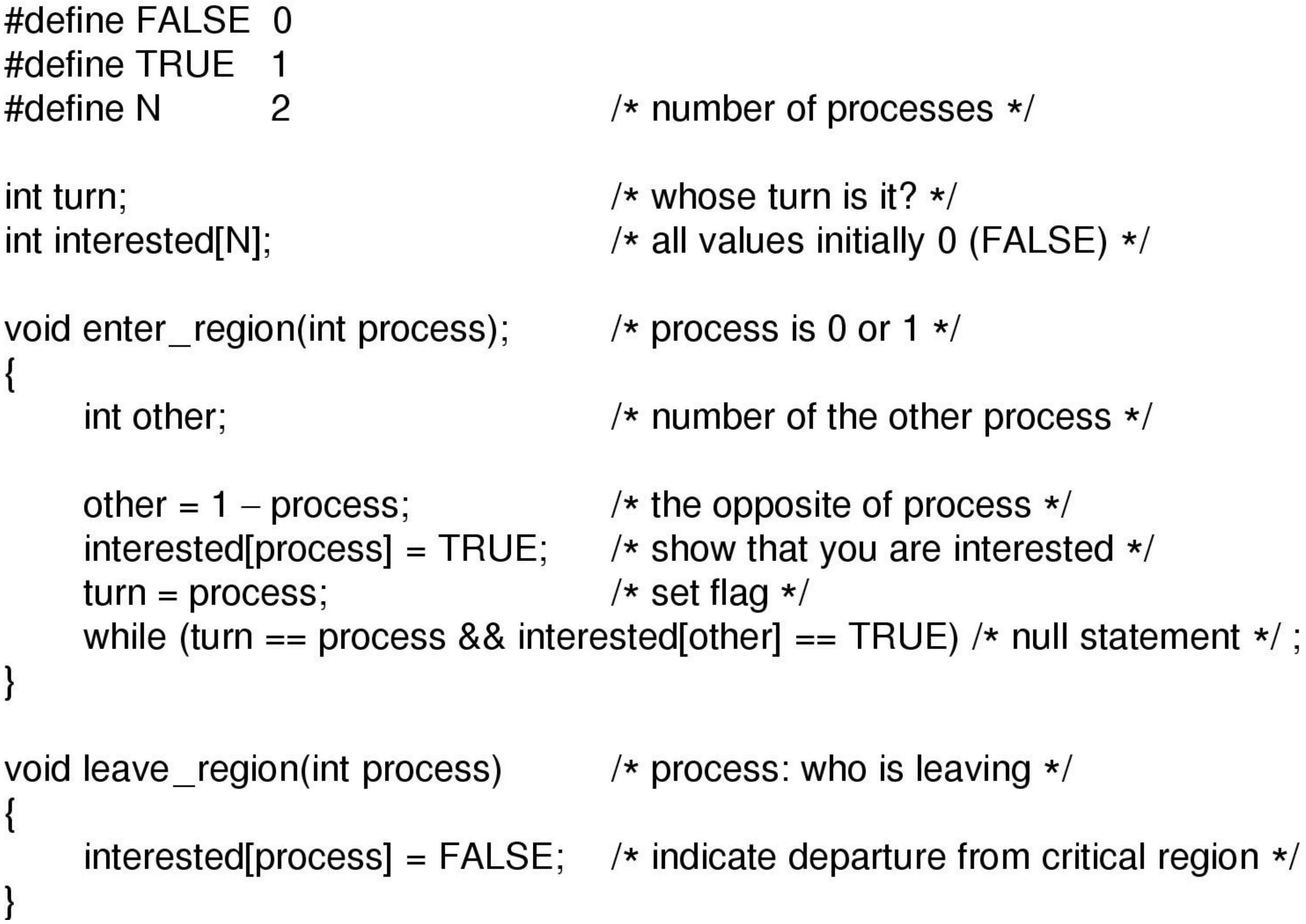
- Two process solution

- The two processes share two variables:
  - int **turn**;
  - Boolean **interested [2]**

- The variable **turn** indicates whose turn it is to enter the critical section

- The **interested** array is used to indicate if a process is interested to enter the critical section.
- **interested[i]** = true implies that process $\mathbf{P}_i$ is interested!

Mutual Exclusion with Busy Waiting (2) : a workable method



```
#define FALSE  0
#define TRUE   1
#define N      2                     /* number of processes */

int turn;                            /* whose turn is it? */
int interested[N];                   /* all values initially 0 (FALSE) */

void enter_region(int process);      /* process is 0 or 1 */
{
     int other;                      /* number of the other process */

     other = 1 – process;            /* the opposite of process */
     interested[process] = TRUE;     /* show that you are interested */
     turn = process;                 /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)       /* process: who is leaving */
{
     interested[process] = FALSE;    /* indicate departure from critical region */
}
```

Peterson's solution for achieving mutual exclusion

Algorithm for Process P_i

```
do {
 interested[i] = TRUE;
turn = j ;
while (interested[j] && turn == j);
critical section
 interested[i] = FALSE;
remainder section
} while (TRUE);
```

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

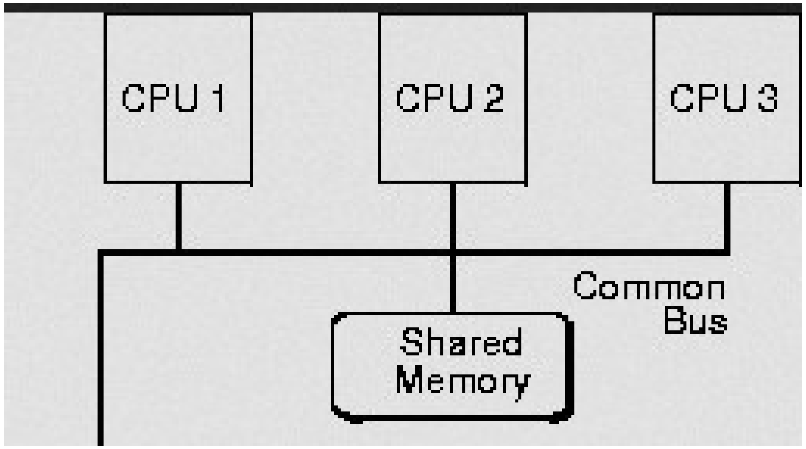
Does this alter the sequence?

Hardware Instruction Based Solutions Multiprocessor system

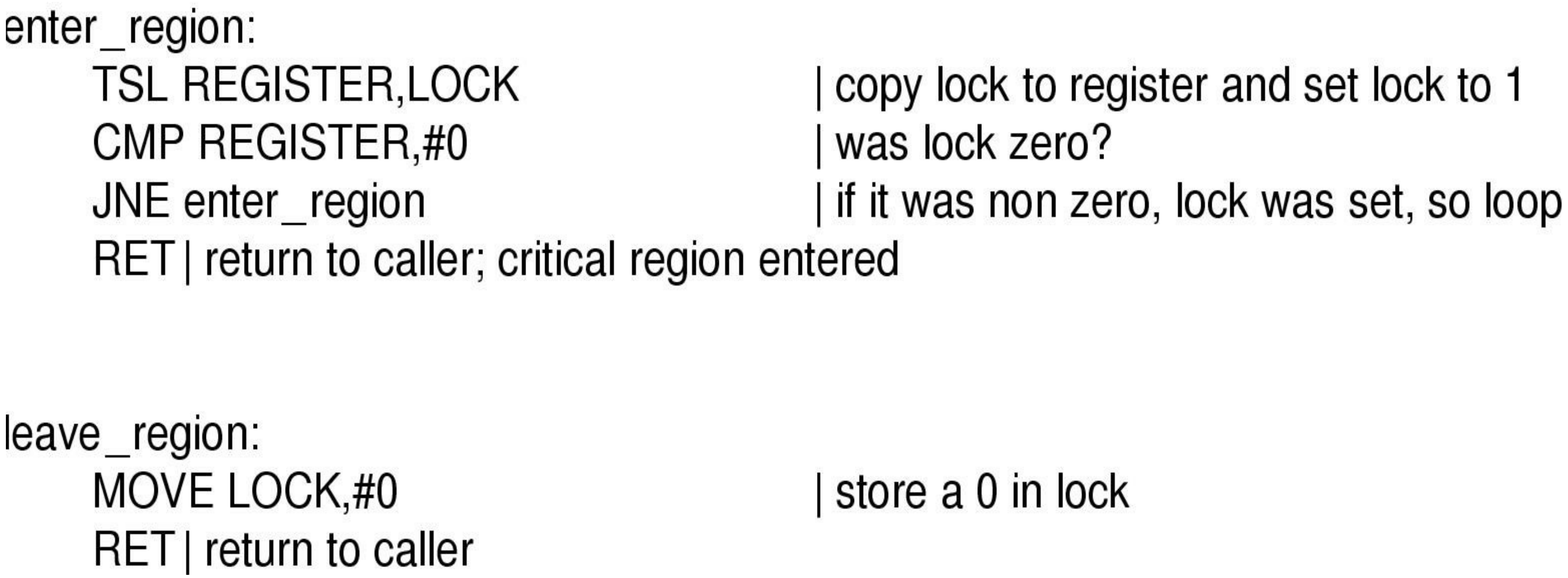
- Some architectures provide special instructions that can be used for synchronization

- TSL: Test and modify the content of a word atomically

```
 TSL Reg, lock
{
   Reg= lock;
   lock = true;
}
```



Hardware Instruction Based Solutions



```
enter_region:
      TSL REGISTER,LOCK              | copy lock to register and set lock to 1
      CMP REGISTER,#0                | was lock zero?
      JNE enter_region               | if it was non zero, lock was set, so loop
      RET | return to caller; critical region entered


leave_region:
      MOVE LOCK,#0                   | store a 0 in lock
      RET | return to caller
```

Does it satisfy all the conditions?

Entering and leaving a critical region using the
TSL instruction

System call version

- Special system call that can be used for synchronization
- TestAndSet: Test and modify the content of a word atomically

```
boolean TestAndSet (boolean &target) {
    boolean v = target;
    target = true;
    return v;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:

```
boolean lock = false;
```

Does it satisfy all the conditions?

- Process P_i

```
do {
while (TestAndSet(lock)) ;
critical section
lock = false;
remainder section
}
```

Swap Instruction

- Definition:

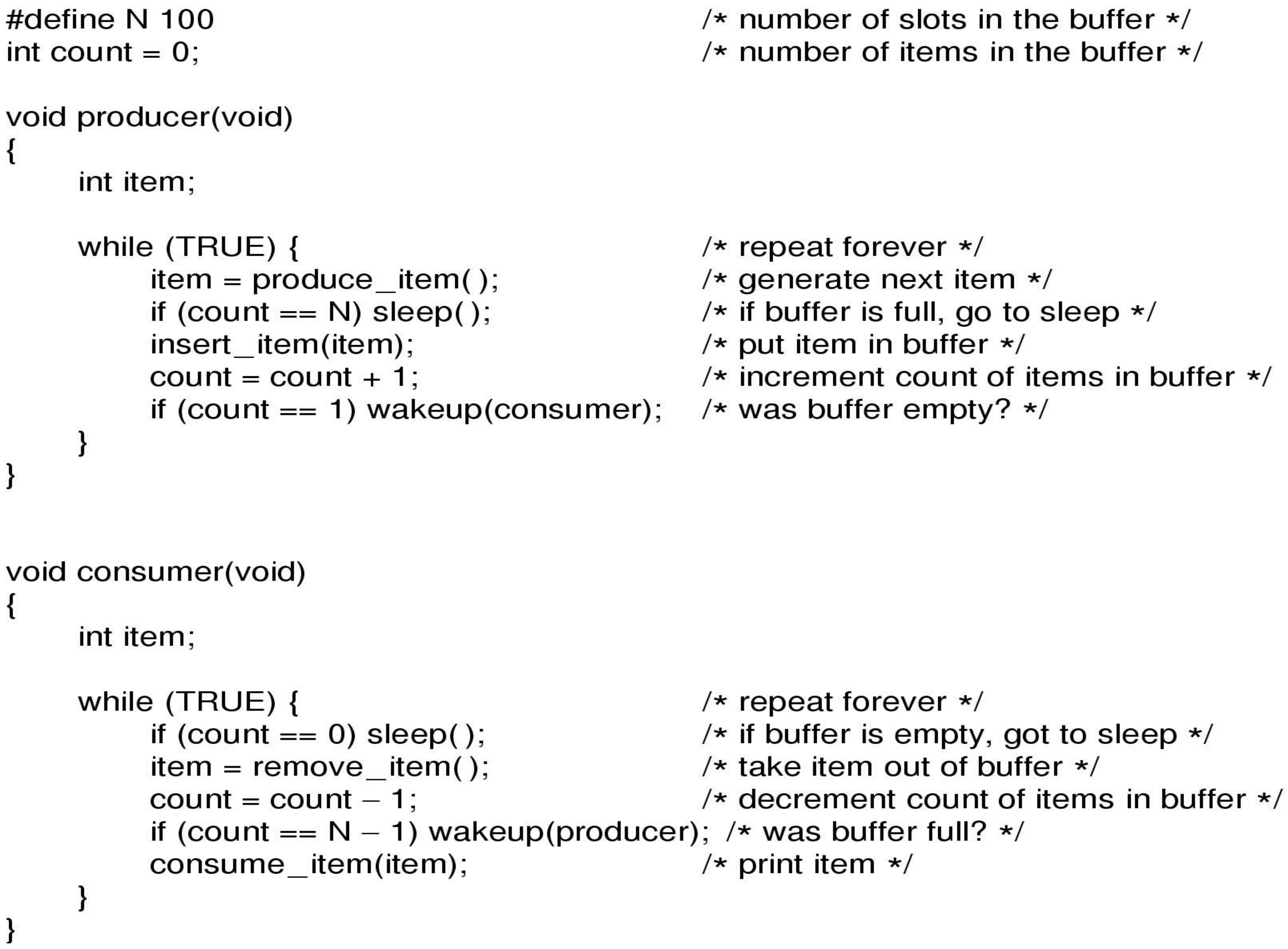
```
void Swap (boolean *a, boolean *b)
 {
        boolean temp = *a;
        *a = *b;
        *b = temp:
 }
```

- Drawback of Busy waiting
  - A lower priority process has entered critical region
  - A higher priority process comes and preempts the lower priority process, it wastes CPU in busy waiting, while the lower priority don’t come out
  - Priority inversion problem

Producer-consumer problem

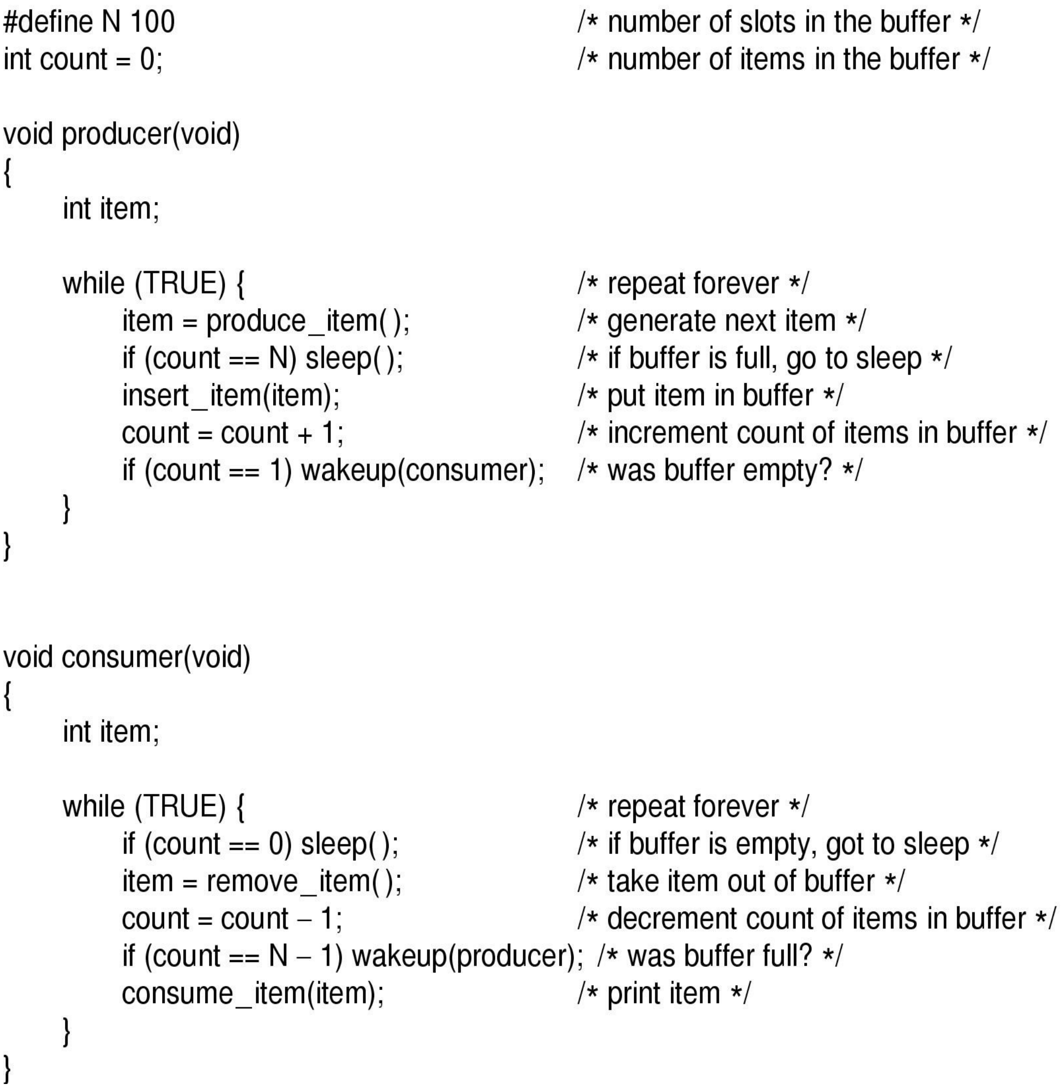
- Two processes share a common, fixed-sized buffer
- Producer puts information into the buffer
- Consumer takes information from buffer
- A simple solution

Sleep and Wakeup



```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Producer-Consumer Problem



```
#define N 100                               /* number of slots in the buffer */
int count = 0;                              /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                          /* repeat forever */
        item = produce_item( );             /* generate next item */
        if (count == N) sleep( );           /* if buffer is full, go to sleep */
        insert_item(item);                  /* put item in buffer */
        count = count + 1;                  /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);   /* was buffer empty? */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                          /* repeat forever */
        if (count == 0) sleep( );           /* if buffer is empty, got to sleep */
        item = remove_item( );              /* take item out of buffer */
        count = count − 1;                  /* decrement count of items in buffer */
        if (count == N − 1) wakeup(producer);  /* was buffer full? */
        consume_item(item);                 /* print item */
    }
}
```

- What can be the problem?

- Signal missing
  - Shared variable: counter
  - When consumer read count with a 0 but didn’t fall asleep in time
  - then the signal will be lost

Producer-consumer problem with fatal race condition

Tasks

- We must ensure proper process synchronization
  - Stop the producer when buffer full
  - Stop the consumer when buffer empty
- We must ensure mutual exclusion
  - Avoid race condition
- Avoid busy waiting

Semaphore

- Widely used synchronization tool
- Does not require busy-waiting
  - CPU is not held unnecessarily while the process is waiting
- A Semaphore *S* is
  - A data structure with an integer variable *S.value* and a queue *S.list* of processes (shared variable)
  - The data structure can only be accessed by two atomic operations, *wait(S)* and *signal(S)* (also called *down(S), P(S)* and *Up(s), V(S)*)

- Value of the semaphore S = value of the integer *S.value*

```
typedef struct {
 int value;
 struct process *list;
 } semaphore
```

Semaphore

Wait(S) S<= semaphore variable

- When a process P executes the wait(S) and finds
- S==0
    - Process must wait => block()
    - Places the process into a waiting queue associated with S
    - Switch from running to waiting state

Signal(S)

When a process P executes the signal(S)

- Check, if some other process Q is waiting on the semaphore S
- Wakeup(Q)
- Wakeup(Q) changes the process from waiting to ready state

Semaphore (wait and signal)

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

List of PCB

Atomic/
Indivisible

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Note: which process is picked for unblocking may depend on policy.

Usage of Semaphore

- **Counting** semaphore – integer value can range over an unrestricted domain
  - Control access to a shared resource with finite elements
  - Wish to use => wait(S)
  - Releases resource=>signal(S)
  - Used for synchronization
- **Binary** semaphore – integer value can range only between 0 and 1
  - Also known as **mutex locks**
  - Used for mutual exclusion

Ordering Execution of Processes using Semaphores (Synchronization)

- Execute statement *B* in P_j only after statement *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
■	■
Stmt. *A*	*wait*(*flag*)
signal(*flag*)	Stmt. *B*

- Multiple such points of synchronization can be enforced using one or more semaphores

Semaphore: Mutual exclusion

- Shared data:

```
semaphore mutex;   /* initially mutex = 1 */
```

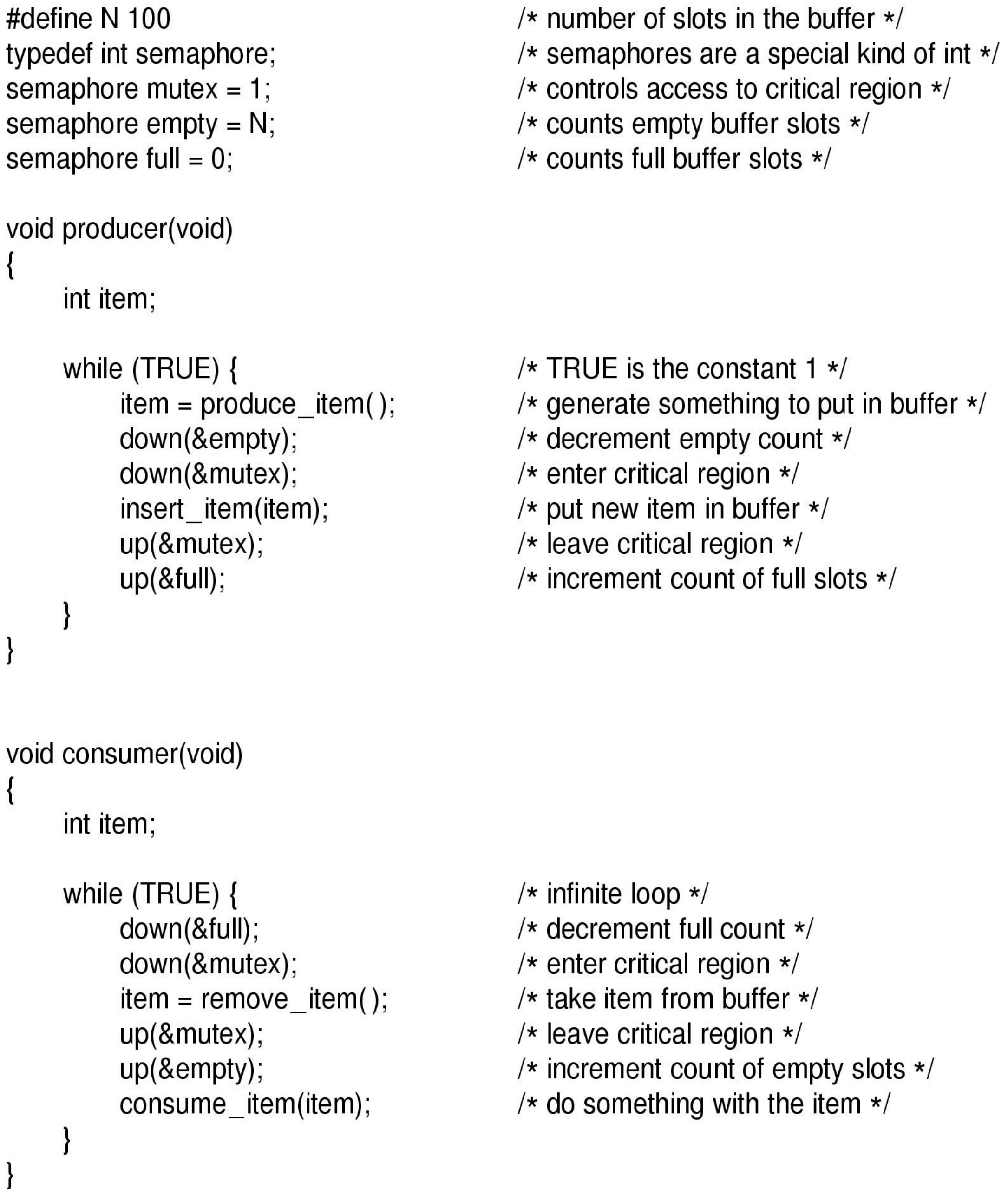
- Process P_i:

```
do {
  wait(mutex);

      critical section

    signal(mutex);

          remainder section

   } while (1);
```

Producer-consumer problem : Semaphore

- Solve producer-consumer problem
  - Full: counting the slots that are full; initial value 0
  - Empty: counting the slots that are empty, initial value N
  - Mutex: prevent access the buffer at the same time, initial value 0 (**binary semaphore**)
  - Synchronization & mutual exclusion

Semaphores



```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

P_0 P_1

```
   wait (S);                    wait (Q);
   wait (Q);                    wait (S);
.             .
.             .
.             .
   signal (S);                 signal (Q);
   signal (Q);                 signal (S);
```

Let S and Q be two semaphores initialized to 1

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
    P0                  P1

   wait (S);            wait (Q);
   wait (Q);            wait (S);
.     .
.     .
.     .
   signal (S);          signal (Q);
   signal (Q);          signal (S);
```

- **Starvation** – indefinite blocking
  - LIFO queue
  - A process may never be removed from the semaphore queue in which it is suspended

Readers-Writers Problem

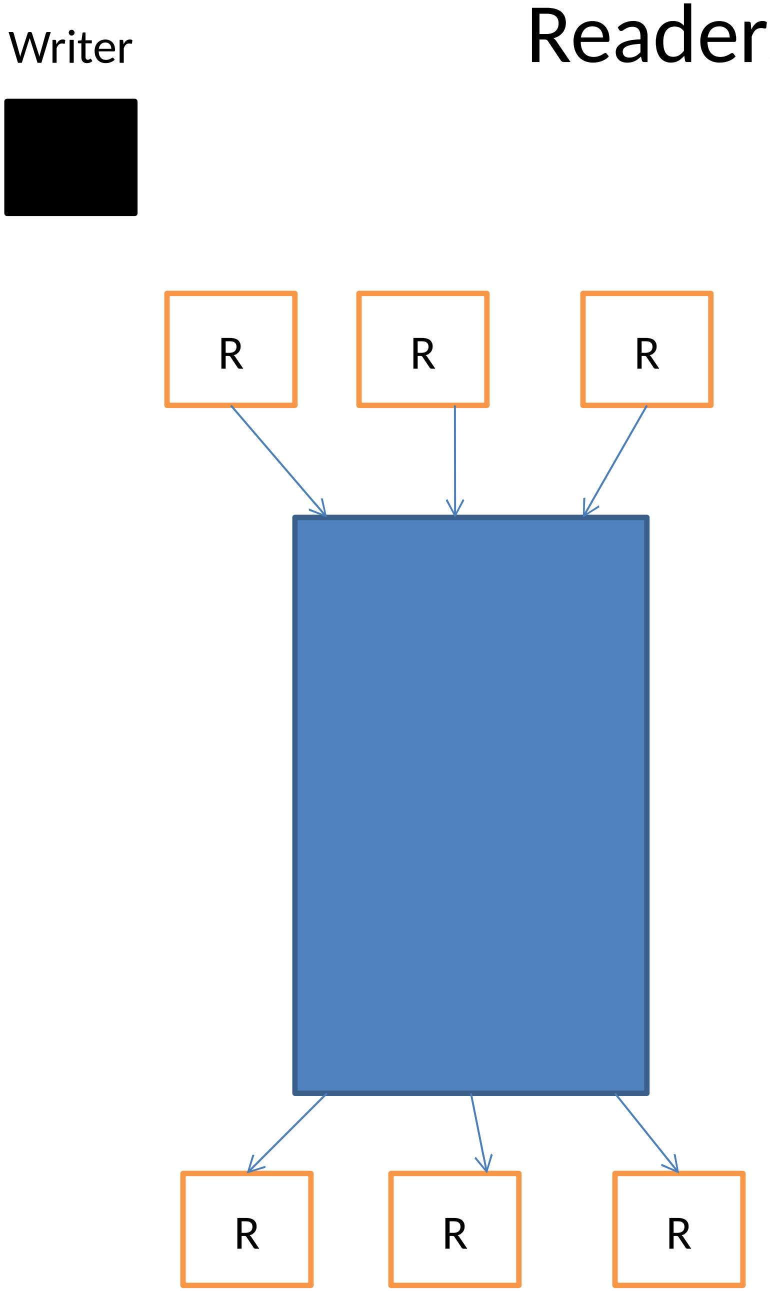
- A database is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write

- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time

- Several variations of how readers and writers are treated – all involve priorities

- Shared Data
  - Database
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers Problem



Writer

- Task of the writer
  - Just lock the dataset and write

Reader

- Task of the **first** reader
  - Lock the dataset
- Task of the **last** reader
  - Release the lock
  - Wakeup the any waiting writer

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
 do {
          wait (wrt) ;

               //    writing is performed

          signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

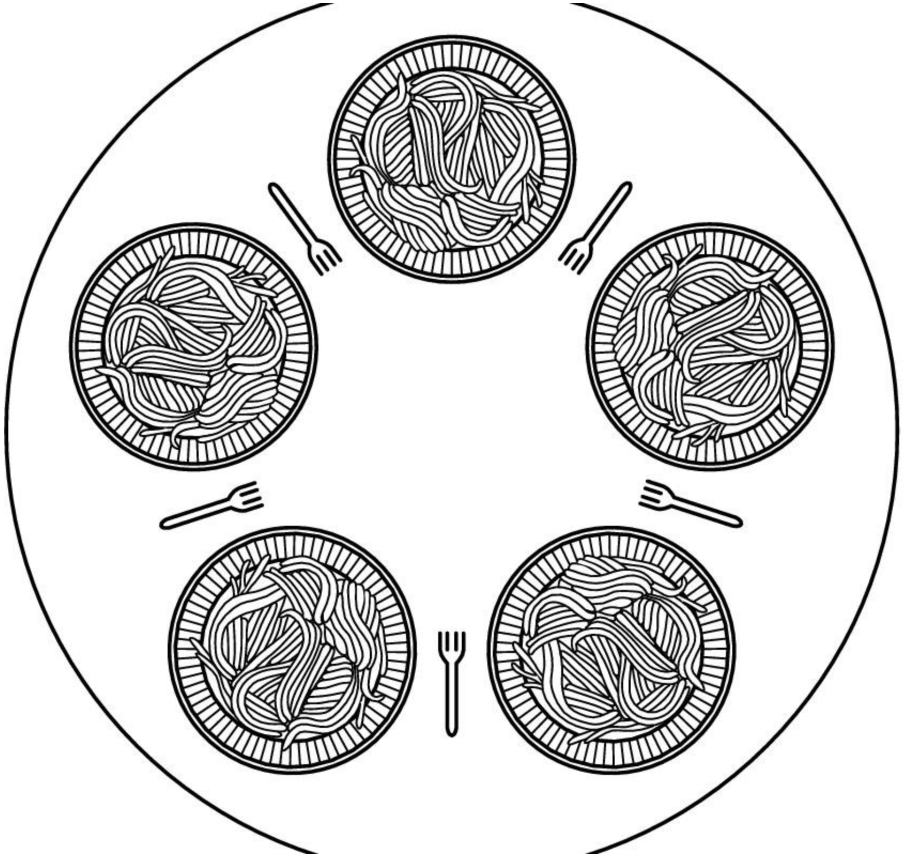
- The structure of a reader process

```
do {
            wait (mutex) ;
            readcount ++ ;
            if (readcount == 1)
        wait (wrt) ;
            signal (mutex)

                 // reading is performed

             wait (mutex) ;
             readcount  - - ;
             if (readcount  == 0)
       signal (wrt) ;
             signal (mutex) ;
      } while (TRUE);
```

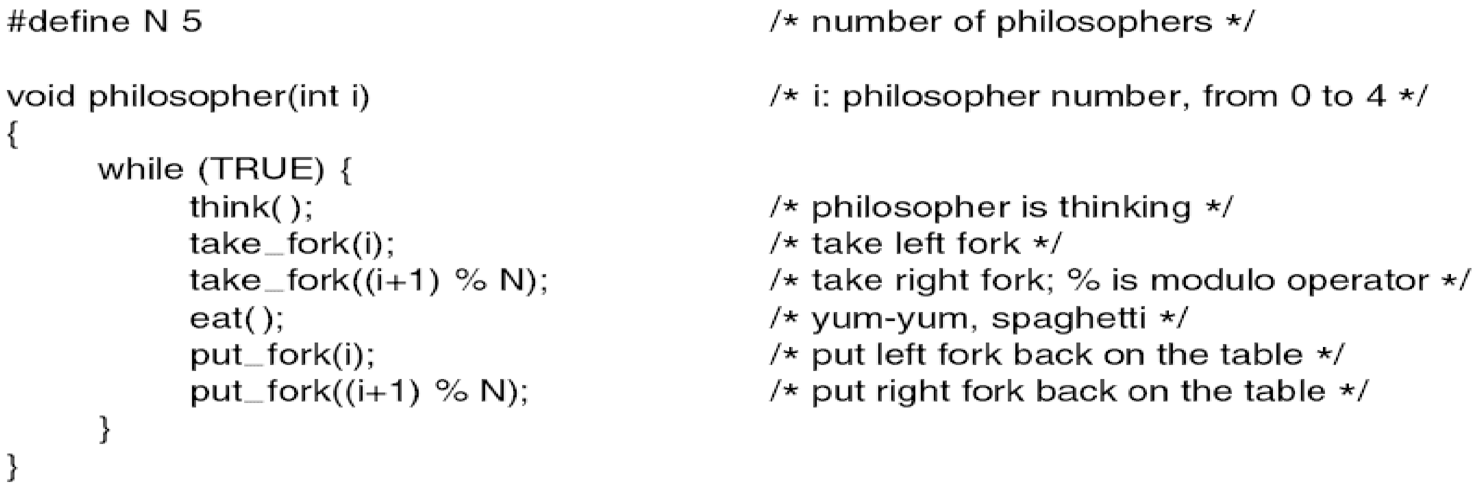
Readers-Writers Problem (Cont.)

- Models database access
- Current solution
  - Reader gets priority over writer
- Home work
  - Writer gets priority

Dining Philosophers Problem



Dining Philosophers Problem
First solution



```
#define N 5                              /* number of philosophers */

void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                      /* philosopher is thinking */
          take_fork(i);                  /* take left fork */
          take_fork((i+1) % N);          /* take right fork; % is modulo operator */
          eat( );                        /* yum-yum, spaghetti */
          put_fork(i);                   /* put left fork back on the table */
          put_fork((i+1) % N);           /* put right fork back on the table */
     }
}
```

- Take_fork() waits until the fork is available
- Available? Then seizes it

- **Ensure that two neighboring philosopher should not seize the same fork**

Dining Philosophers Problem

Each fork is implemented as a semaphore

- The structure of Philosopher *i*:

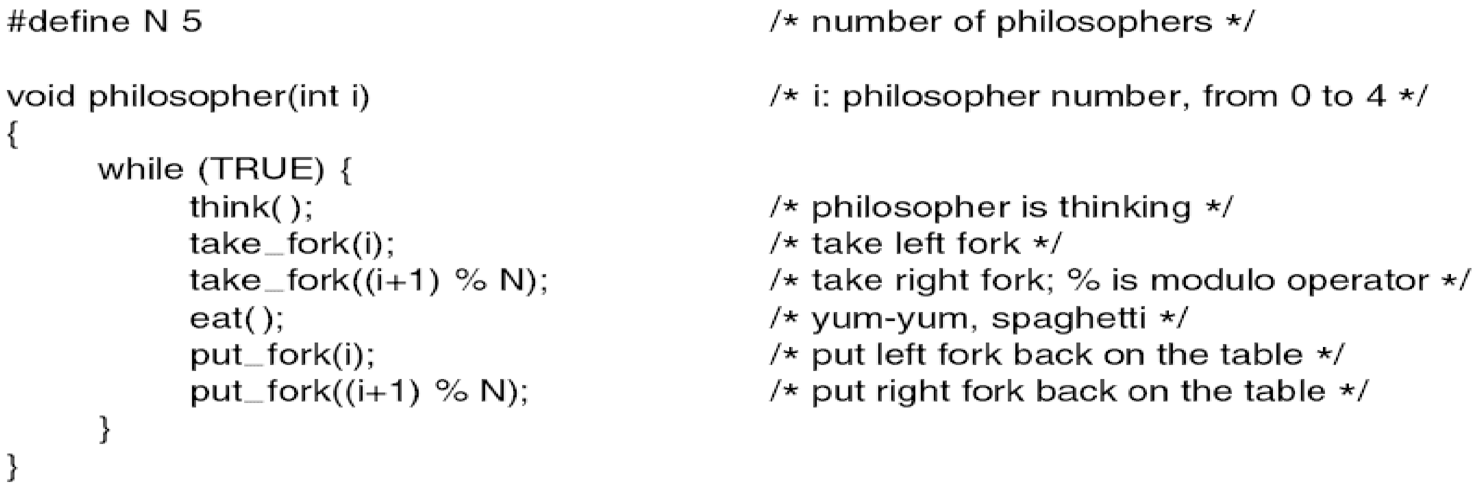
Semaphore fork [5] initialized to 1

```
do  {
          wait ( fork[i] );
          wait ( fork[ (i + 1) % 5] );

                //  eat

          signal ( fork[i] );
          signal (fork[ (i + 1) % 5] );

               //  think

} while (TRUE);
```

Ensures no two neighboring philosophers can eat simultaneously

- What is the problem with this algorithm?

Dining Philosophers Problem
First solution



```
#define N 5                              /* number of philosophers */

void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                      /* philosopher is thinking */
          take_fork(i);                  /* take left fork */
          take_fork((i+1) % N);          /* take right fork; % is modulo operator */
          eat( );                        /* yum-yum, spaghetti */
          put_fork(i);                   /* put left fork back on the table */
          put_fork((i+1) % N);           /* put right fork back on the table */
     }
}
```

- Take_fork() waits until the fork is available
- Available? Then seizes it

- Suppose all of them take the left fork simultaneously
  - None of them will get the right fork
  - Deadlock

Dining Philosophers Problem
Second solution

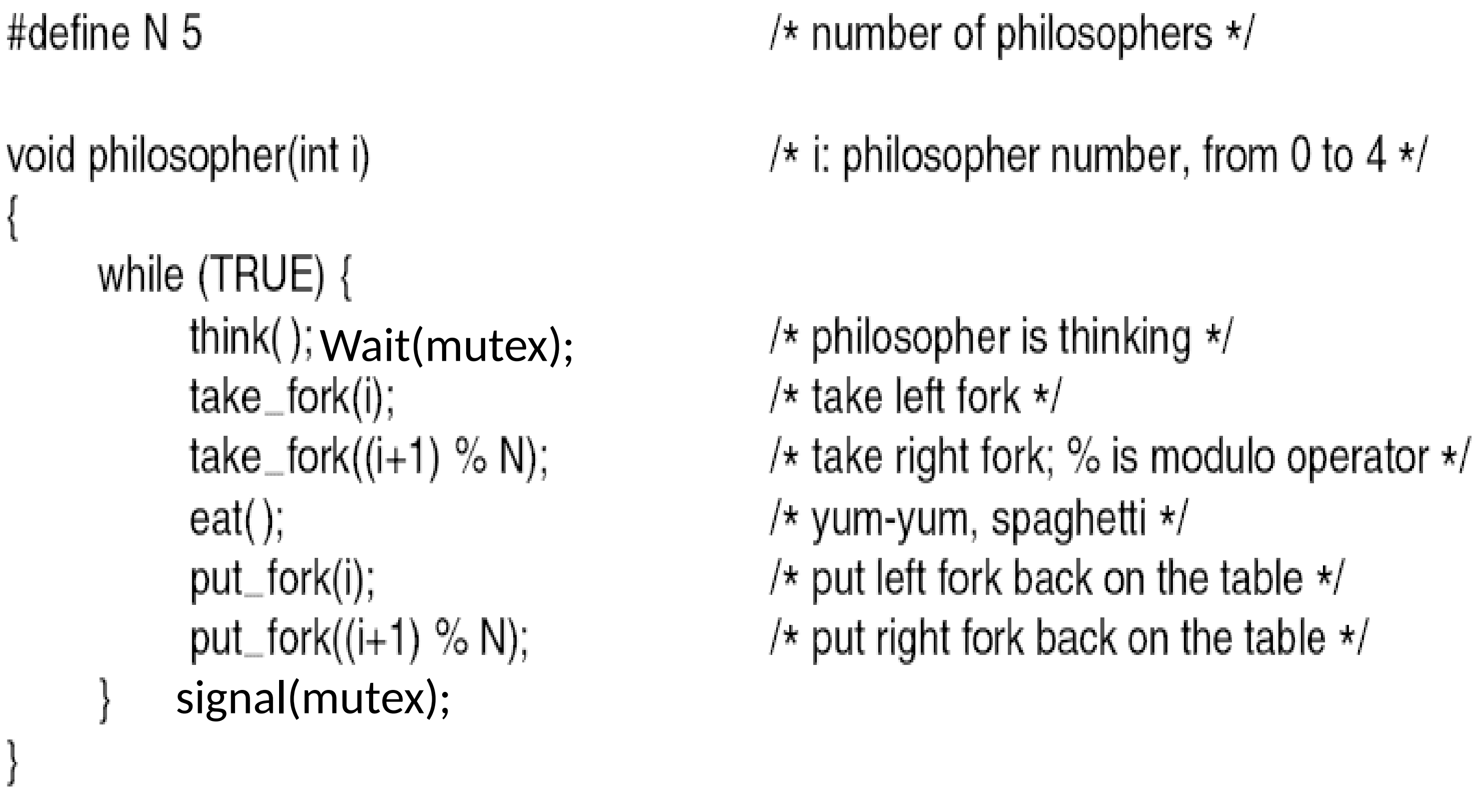
- After taking the left fork, philosopher checks to see if right fork is available
  - If not, puts down the left fork

Limitation

- All of them start simultaneously, pick up the left forks
- Seeing that their right forks are not available
  - Putting down their left fork
- Starvation

- Random delay (Exponential backoff) not going to help for critical systems

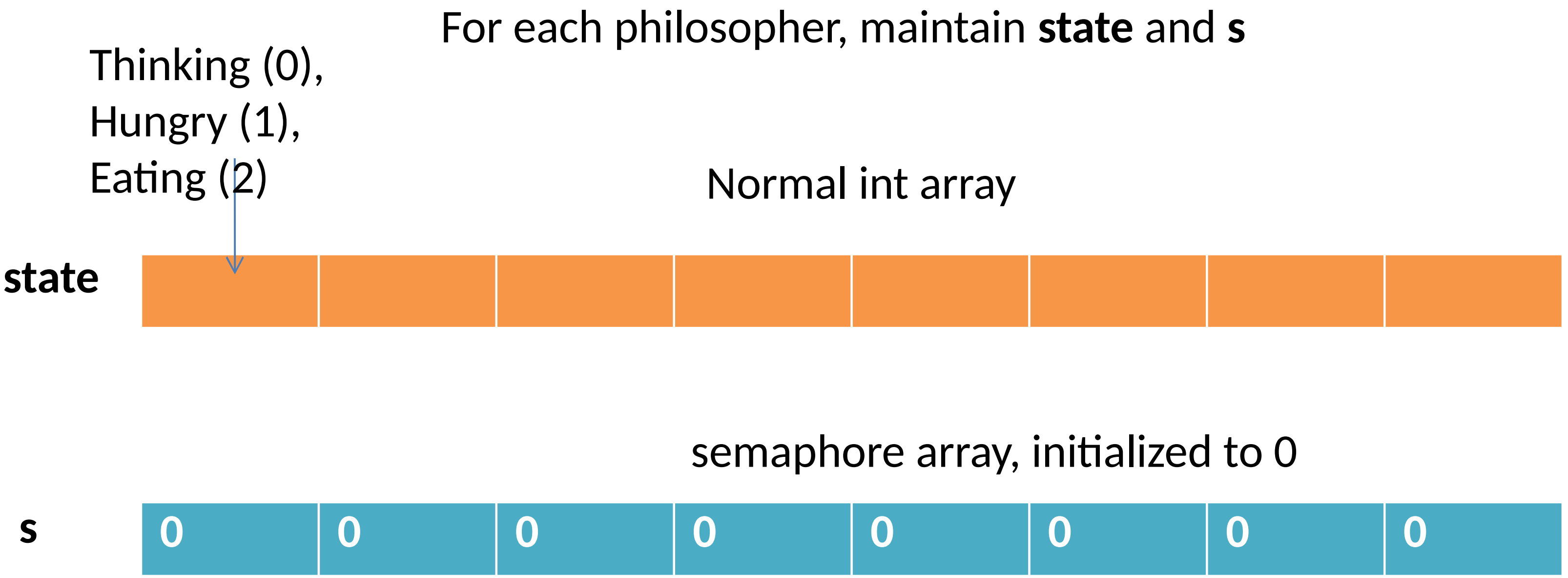
Dining Philosophers Problem
Third solution



```
#define N 5                              /* number of philosophers */

void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( ); Wait(mutex);   /* philosopher is thinking */
                take_fork(i);            /* take left fork */
                take_fork((i+1) % N);    /* take right fork; % is modulo operator */
                eat( );                  /* yum-yum, spaghetti */
                put_fork(i);             /* put left fork back on the table */
                put_fork((i+1) % N);     /* put right fork back on the table */
        }      signal(mutex);
}
```

Poor resource utilization

Dining Philosophers Problem
Final solution



For each philosopher, maintain **state** and **s**

Thinking (0),
Hungry (1),
Eating (2)

Normal int array

state

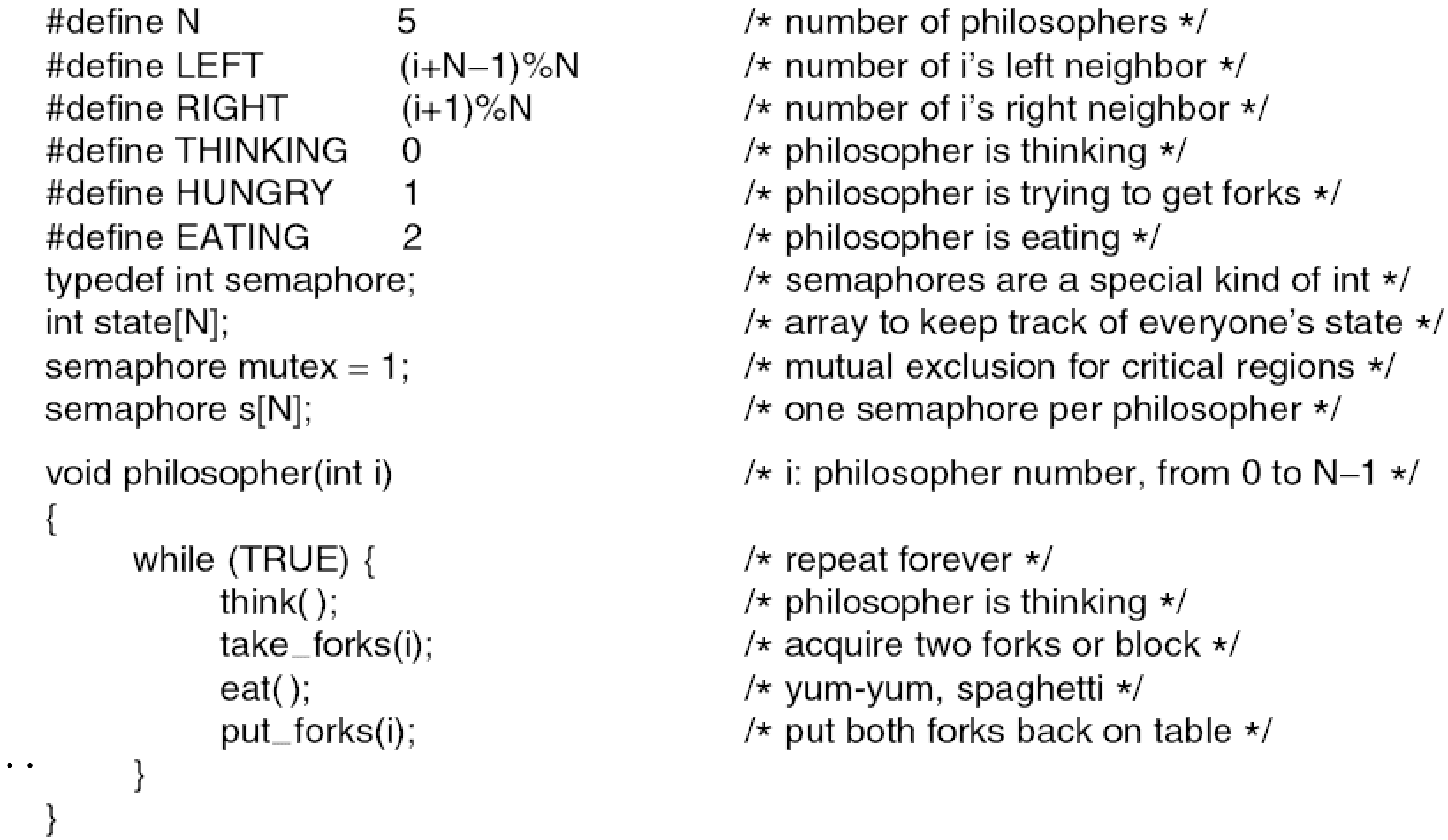
semaphore array, initialized to 0

s

0	0	0	0	0	0	0	0

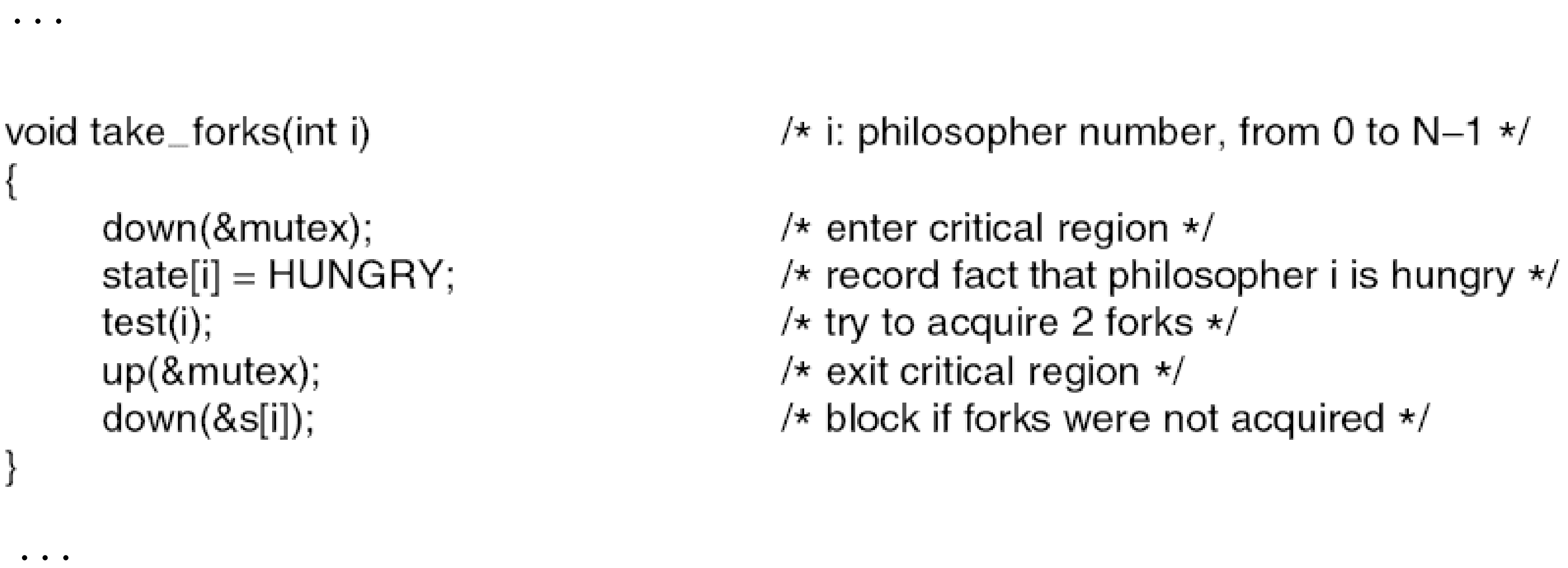
- **State** takes care of acquiring the fork
- **s** stops a philosopher from eating when fork is not available

Dining Philosophers Problem
Final solution



```
#define N            5              /* number of philosophers */
#define LEFT         (i+N−1)%N      /* number of i's left neighbor */
#define RIGHT        (i+1)%N        /* number of i's right neighbor */
#define THINKING     0              /* philosopher is thinking */
#define HUNGRY       1              /* philosopher is trying to get forks */
#define EATING       2              /* philosopher is eating */
typedef int semaphore;              /* semaphores are a special kind of int */
int state[N];                       /* array to keep track of everyone's state */
semaphore mutex = 1;                /* mutual exclusion for critical regions */
semaphore s[N];                     /* one semaphore per philosopher */

void philosopher(int i)             /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                  /* repeat forever */
        think( );                   /* philosopher is thinking */
        take_forks(i);              /* acquire two forks or block */
        eat( );                     /* yum-yum, spaghetti */
        put_forks(i);               /* put both forks back on table */
. . }
}
```

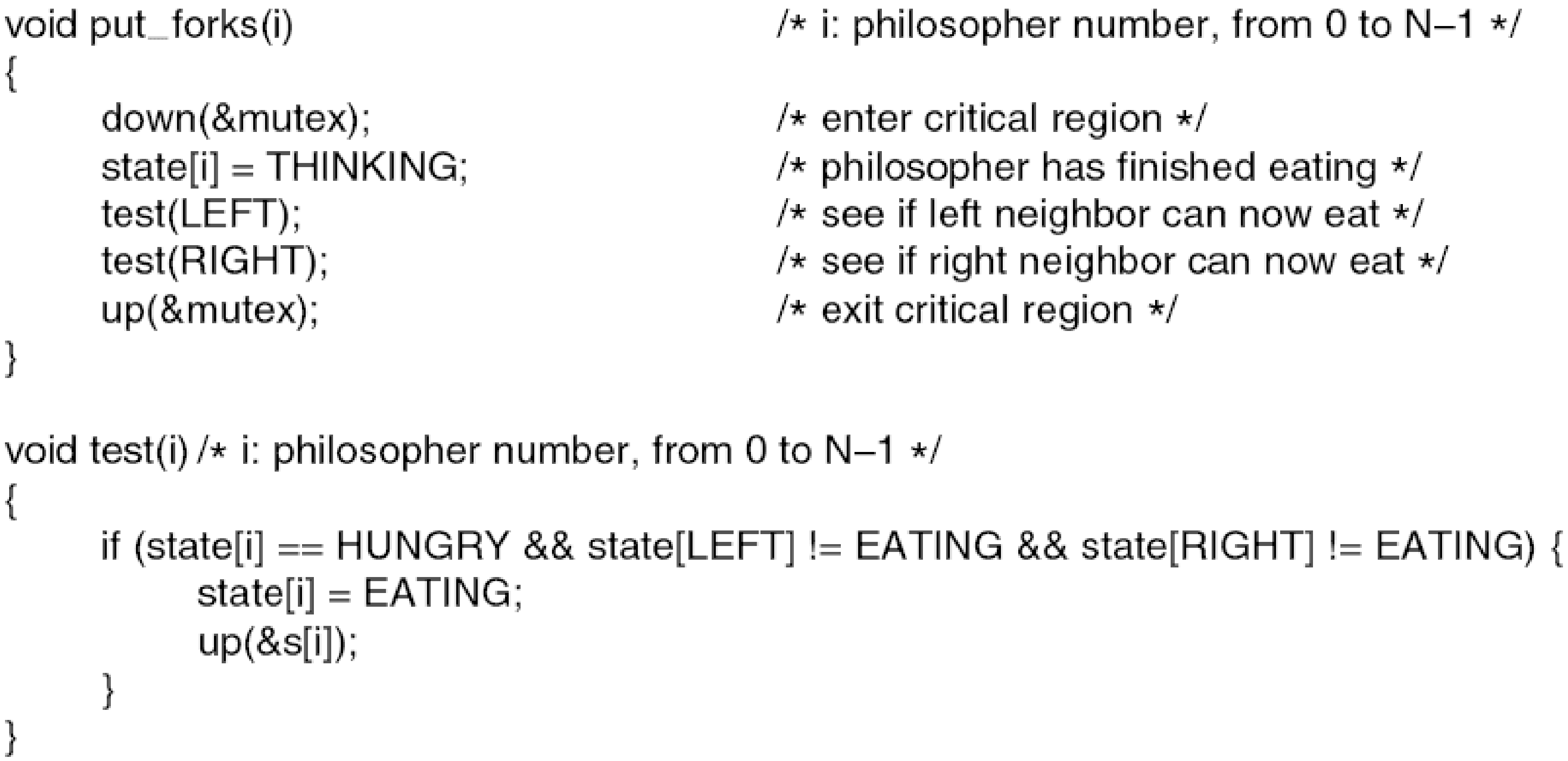
Dining Philosophers Problem
Final solution



```
. . .

void take_forks(int i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                        /* enter critical region */
        state[i] = HUNGRY;                   /* record fact that philosopher i is hungry */
        test(i);                             /* try to acquire 2 forks */
        up(&mutex);                          /* exit critical region */
        down(&s[i]);                         /* block if forks were not acquired */
}

 . . .
```

Dining Philosophers Problem
Final solution

. . .



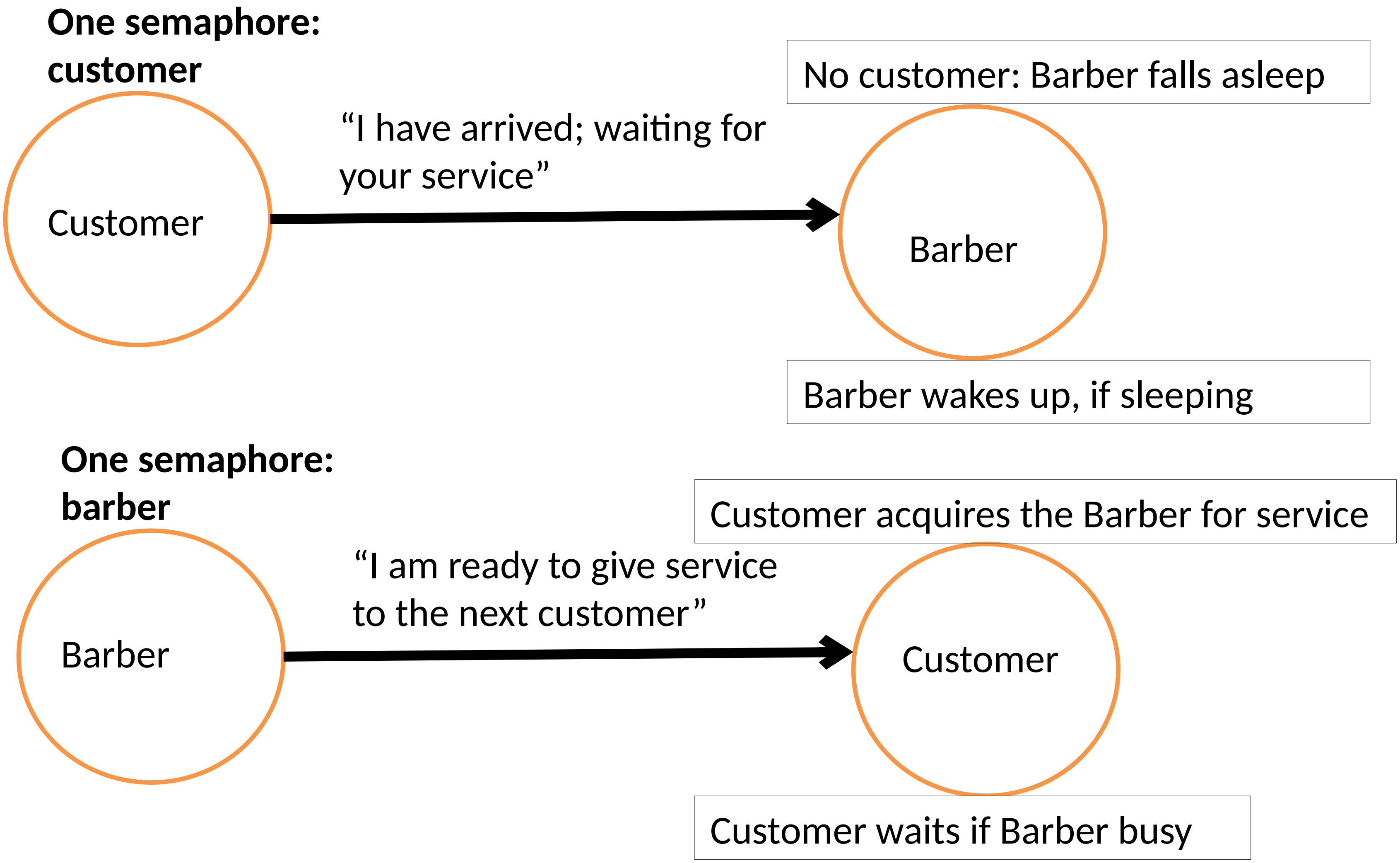
```
void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
     down(&mutex);                      /* enter critical region */
     state[i] = THINKING;               /* philosopher has finished eating */
     test(LEFT);                        /* see if left neighbor can now eat */
     test(RIGHT);                       /* see if right neighbor can now eat */
     up(&mutex);                        /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

The Sleeping Barber Problem

Challenges

- Actions taken by barber and customer takes unknown amount of time (checking waiting room, entering shop, taking waiting room chair)
- Scenario 1
  - Customer arrives, observe that barber busy
  - Goes to waiting room
  - While he is on the way, barber finishes the haircut
  - Barber checks the waiting room
  - Since no one there, Barber sleeps
  - The customer reaches the waiting room and waits forever
- Scenario 2
  - Two customer arrives at the same time
  - Barber is busy
  - Both customers try to occupy the same chair!

Barber sleeps on "**Customer**"
Customer sleeps on "**Barber**"



One semaphore: customer

Customer

"I have arrived; waiting for your service"

Barber

No customer: Barber falls asleep

Barber wakes up, if sleeping

One semaphore: barber

Barber

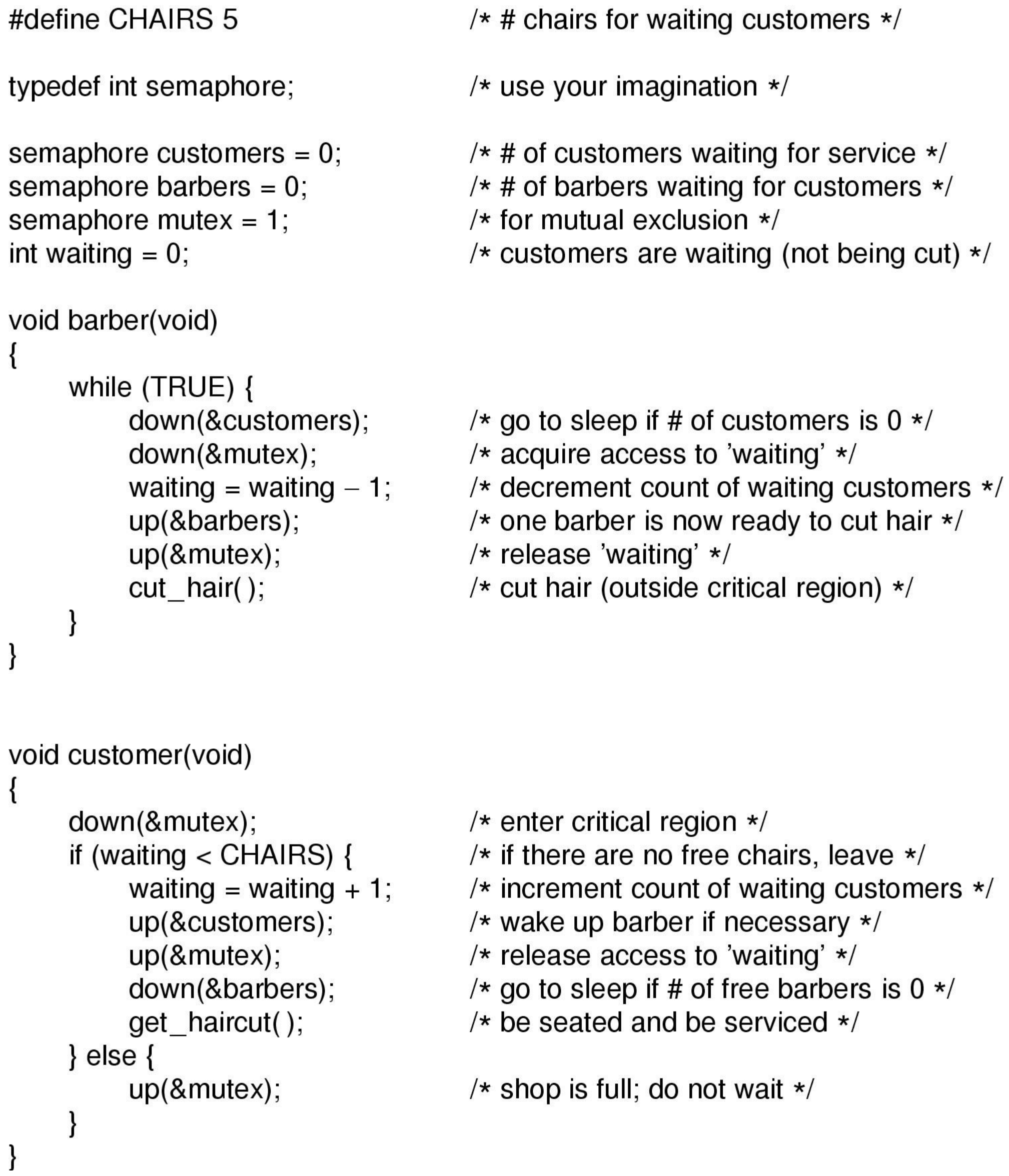
"I am ready to give service to the next customer"

Customer

Customer acquires the Barber for service

Customer waits if Barber busy

The Sleeping Barber Problem



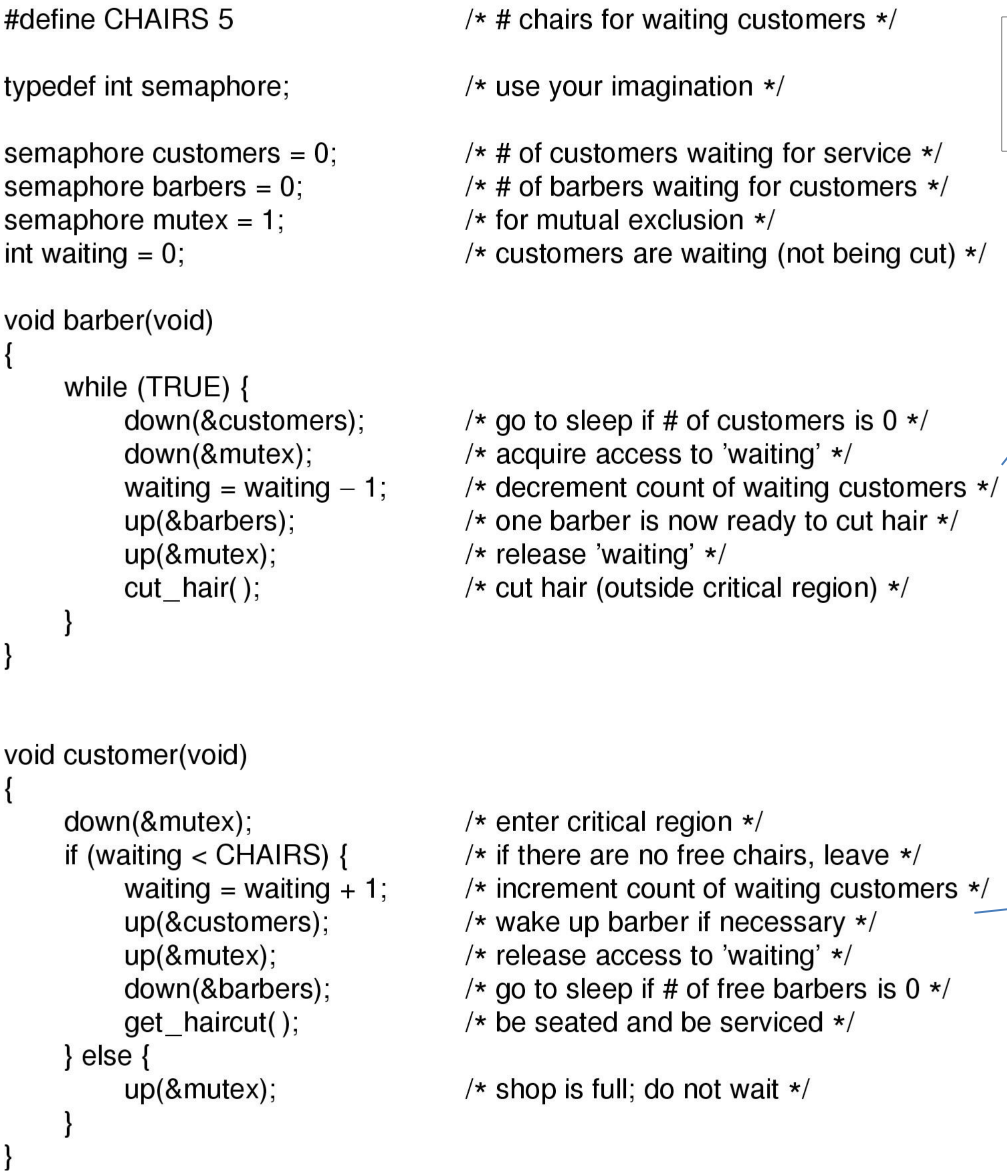
```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting − 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair( );              /* cut hair (outside critical region) */
     }
}


void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

Semaphore Barber: Used to call a waiting customer.
Barber=1: Barber is ready to cut hair and a customer is ready (to get service) too!
Barber=0: customer occupies barber or waits

Semaphore customer: Customer informs barber that “I have arrived; waiting for your service”

Mutex: Ensures that only one of the participants can change state at once

The Sleeping Barber Problem



```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting − 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair( );              /* cut hair (outside critical region) */
     }
}


void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

Barber sleeps on “**Customer**”
Customer sleeps on “**Barber**”

For Barber: Checking the waiting room and calling the customer makes the **critical section**

For customer: Checking the waiting room and informing the barber makes its **critical section**

Deadlock

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one

- Example
  - semaphores *A* and *B*, initialized to 1

P_0 P_1

wait (A); wait (B);

wait(B) wait(A)

Introduction To Deadlocks

Deadlock can be defined formally as follows:

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

System Model

- Resource types R_1, R_2, . . ., R_m

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.

- Each process requests for an instance of a resource type

- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock: necessary conditions

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by

  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

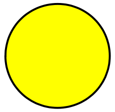
Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

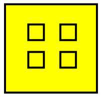
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

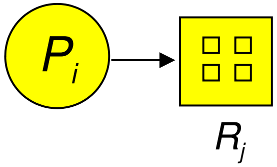
- Process



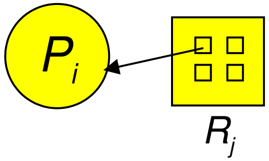
- Resource Type with 4 instances



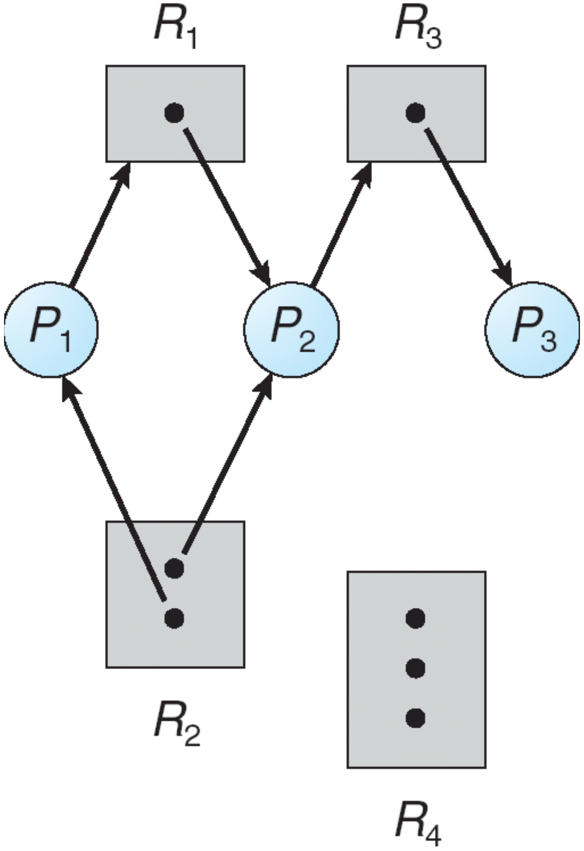
- P_i requests instance of R_j



- P_i is holding an instance of R_j

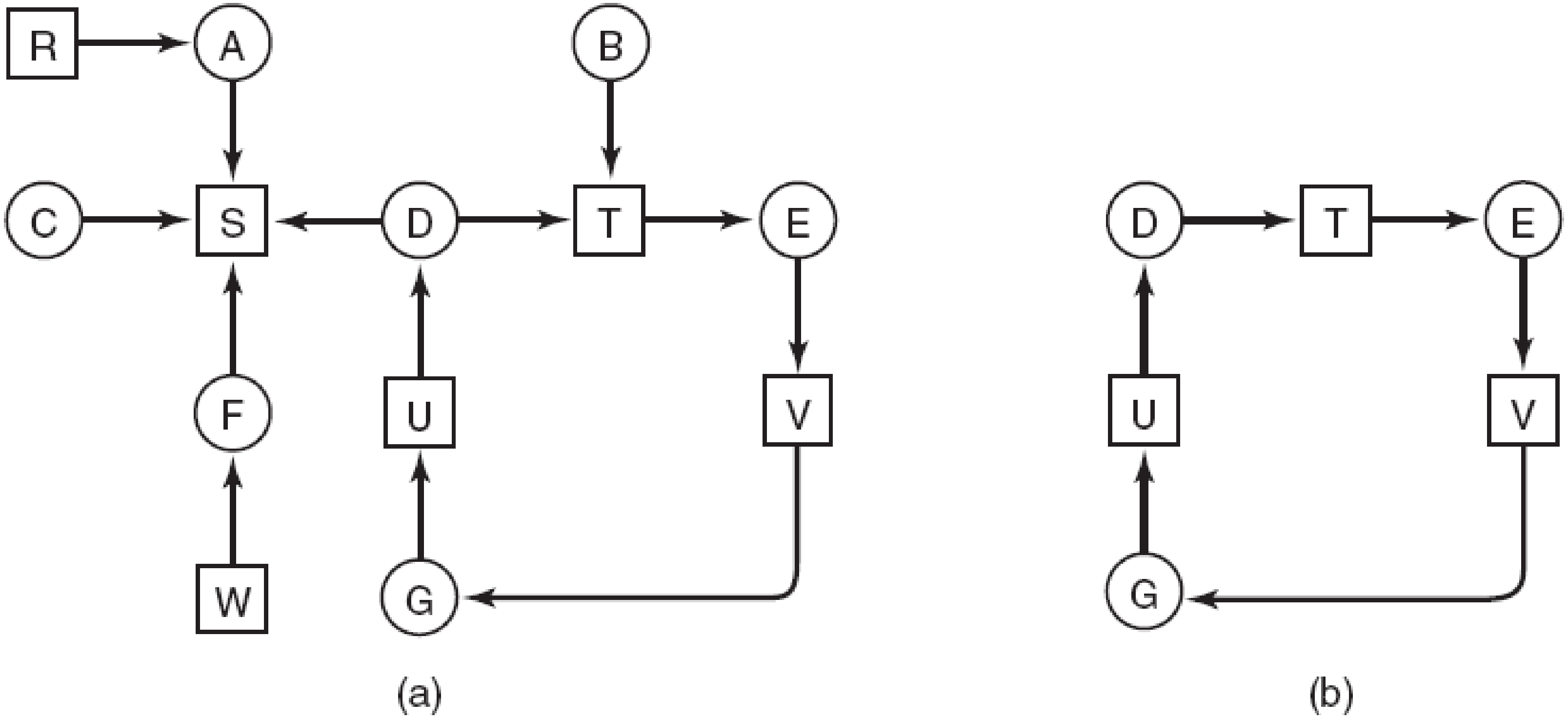


Example of a Resource Allocation Graph



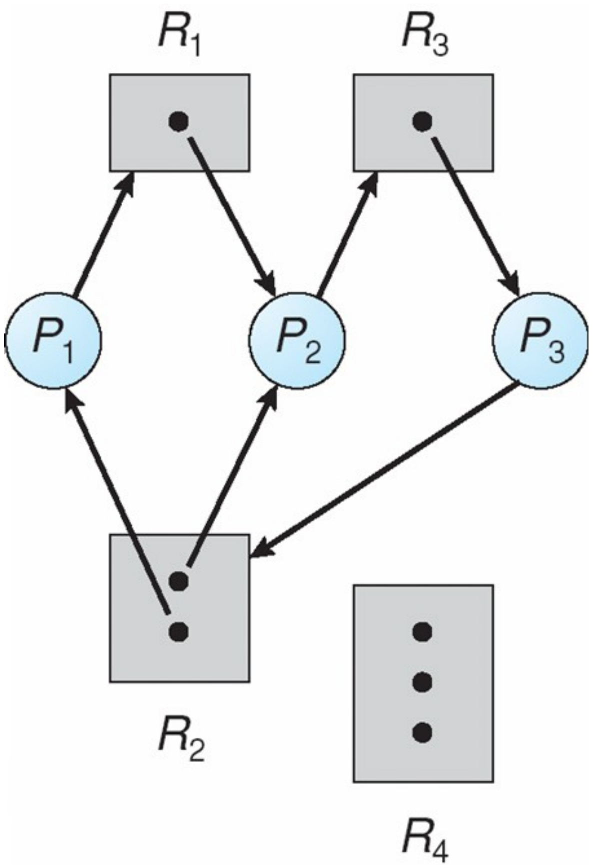
No cycle; No deadlock

One Resource of Each Type



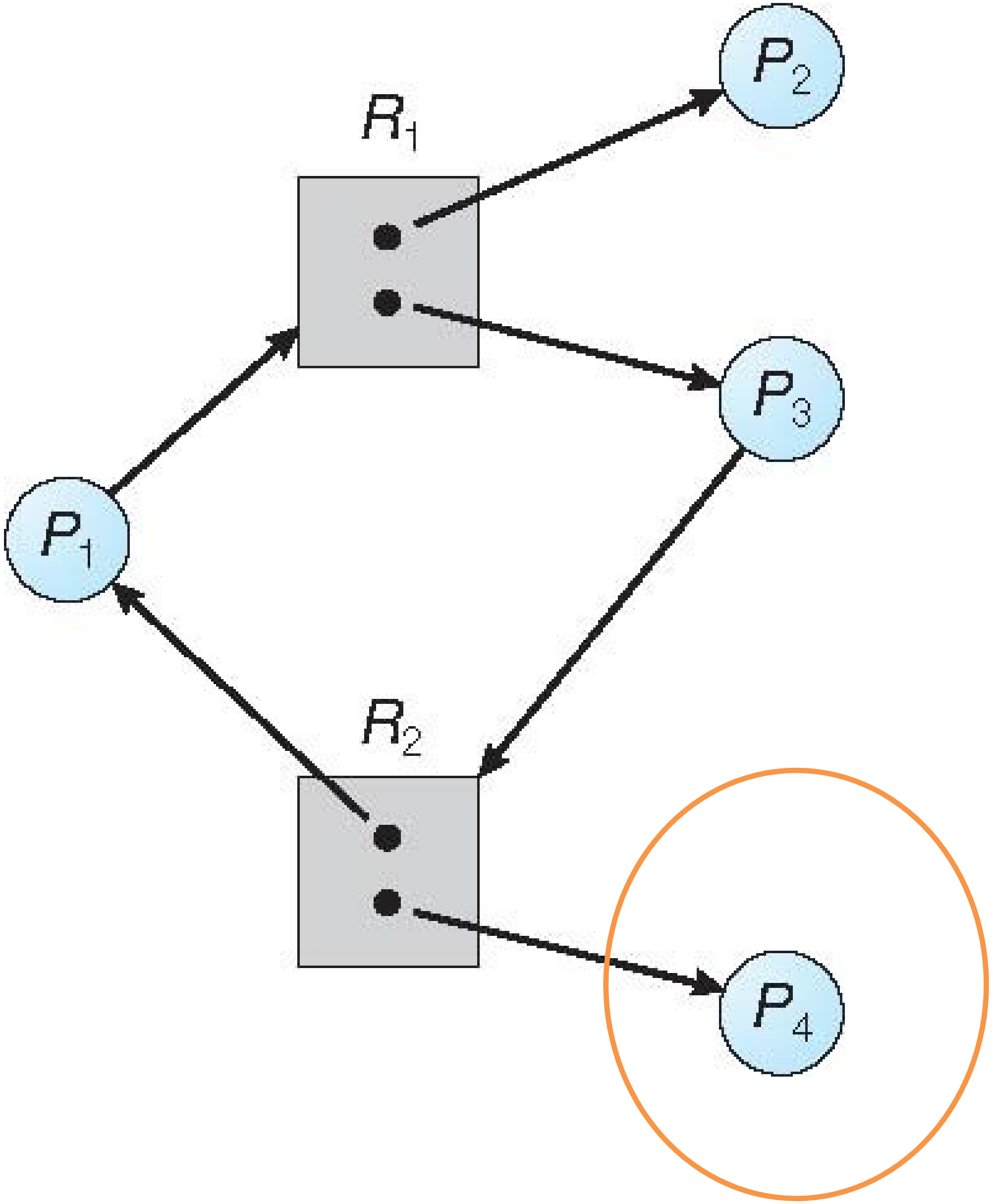
Contains Cycle; Deadlock

Resource Allocation Graph With A Deadlock



P_3 requests R_2

Graph With A Cycle But No Deadlock

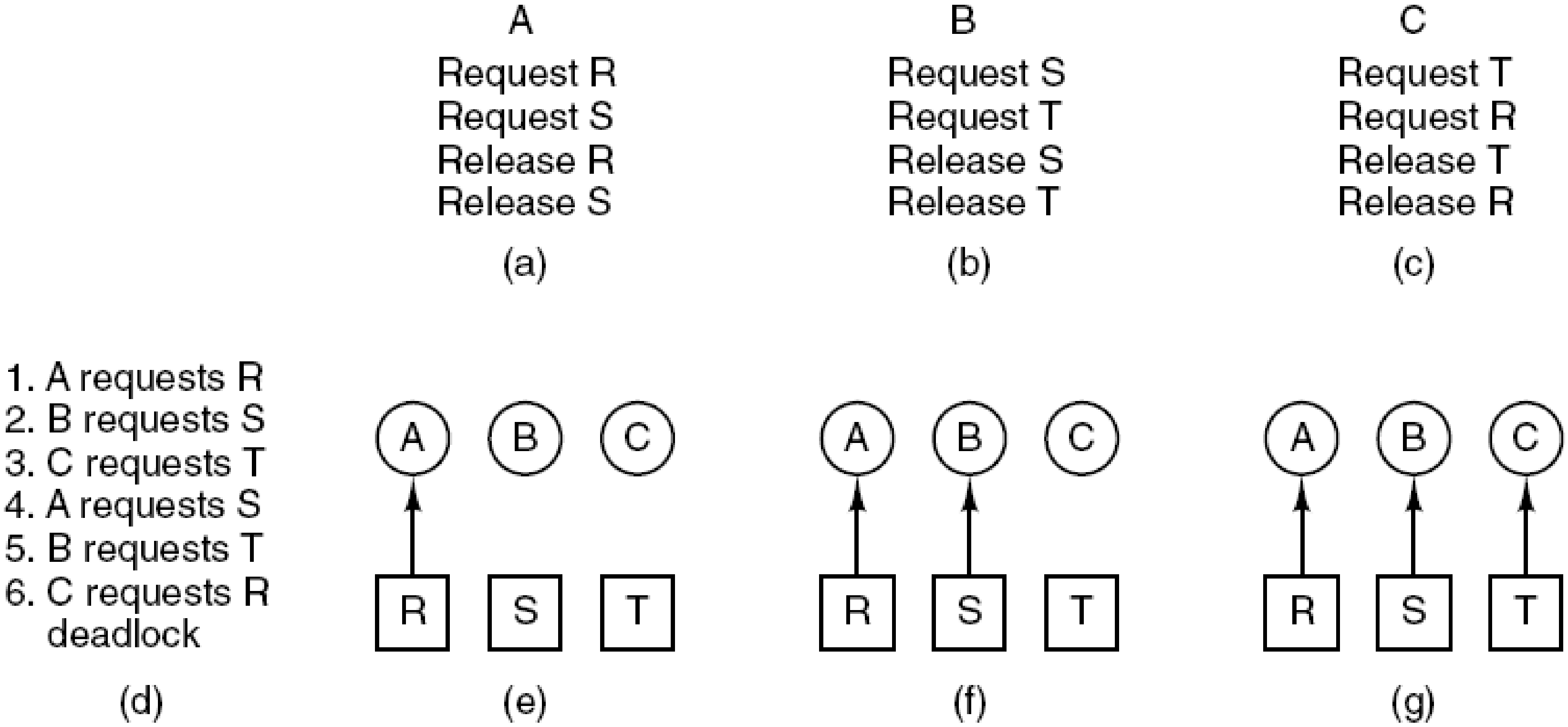


P1->R1->P3->R2->P1

- If the resource allocation graph does not have a cycle
  - System is not in a deadlocked state

- If there is a cycle
  - May or may not be in a deadlocked state

Deadlock Modeling



A

Request R
Request S
Release R
Release S

(a)

B

Request S
Request T
Release S
Release T

(b)

C

Request T
Request R
Release T
Release R

(c)

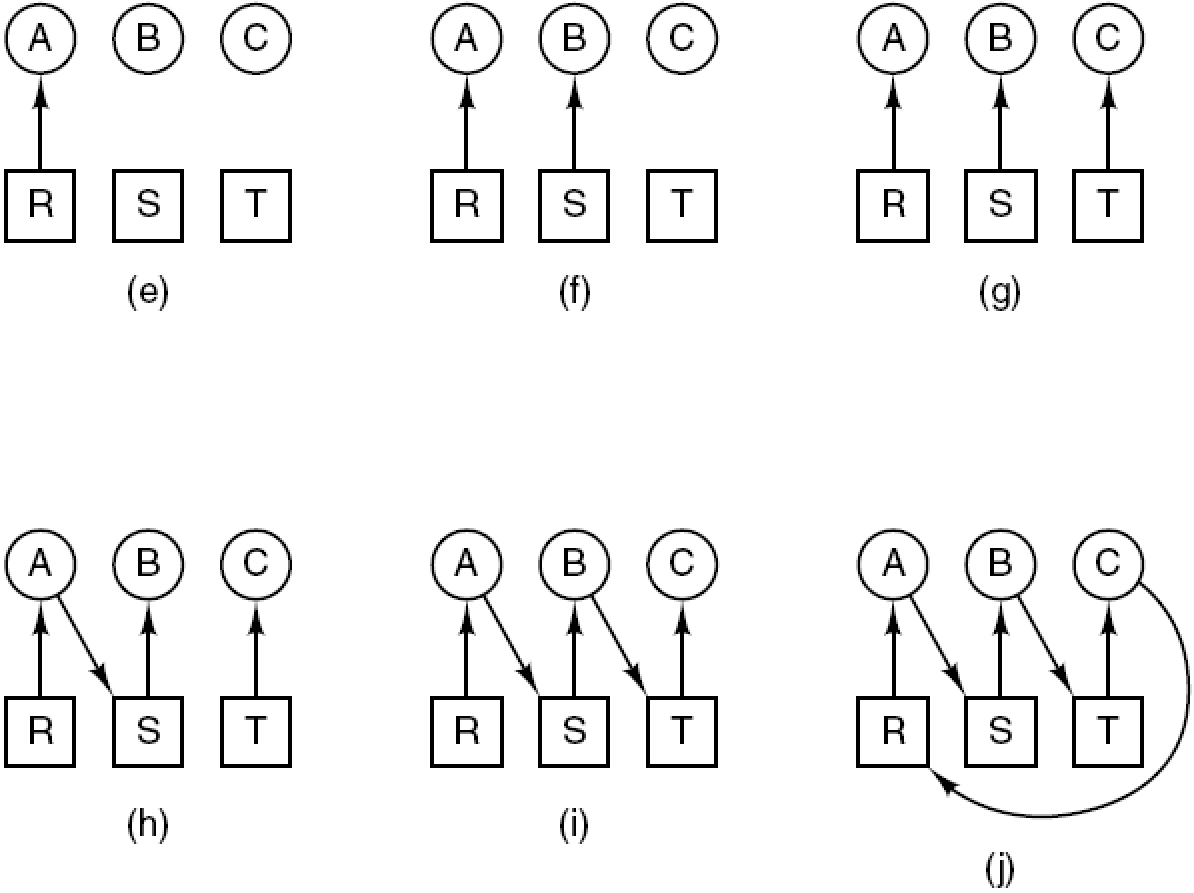
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

(d)

Deadlock Modeling

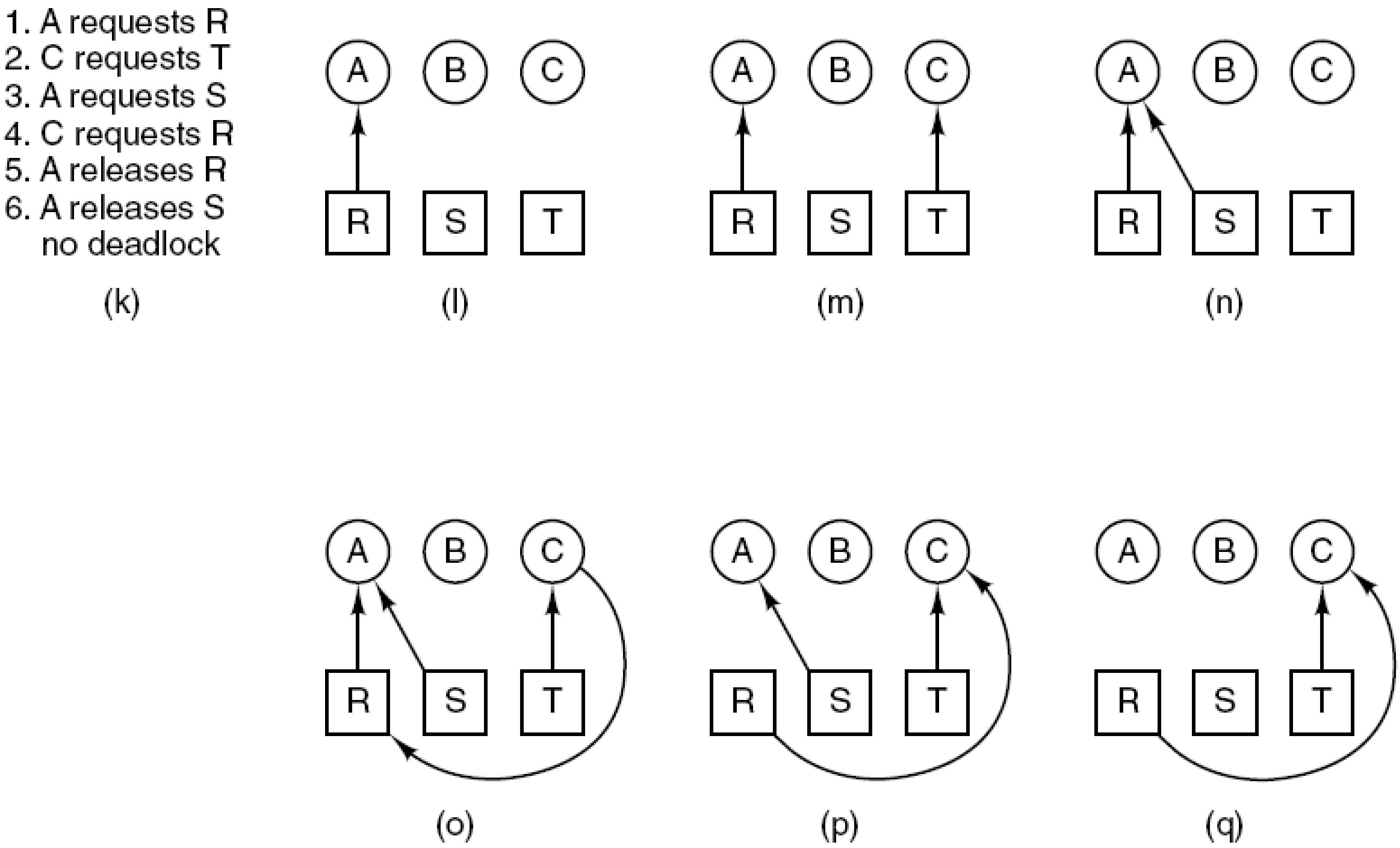
- 1. A requests R
  2. B requests S
  3. C requests T
  4. A requests S
  5. B requests T
  6. C requests R
  deadlock

  (d)



Deadlock

Deadlock Modeling



1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock

Suspend process B

Deadlock Handling

Strategies for dealing with deadlocks:

1. Detection and recovery. Let deadlocks occur, detect them, take action.
2. Dynamic avoidance by careful resource allocation.
3. Prevention, by structurally negating one of the four required conditions.
4. Just ignore the problem.

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

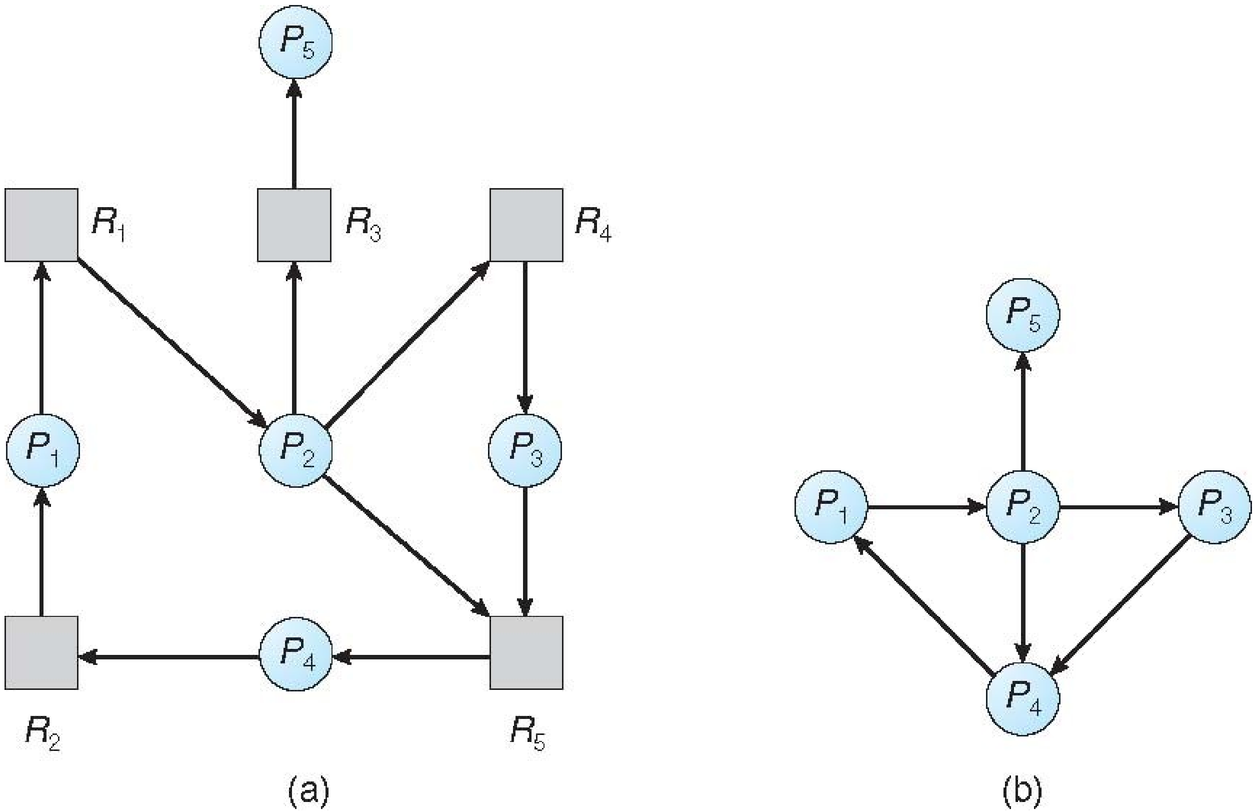
- *In resource graph*
  - – $P_i \rightarrow R$ *and* $R \rightarrow P_j$

- Maintain *wait-for* graph
  - – Nodes are processes
  - – $P_i \rightarrow P_j$ if P_i is waiting for P_j

- Periodically invoke an algorithm that searches for a cycle in the graph.

- If there is a cycle, there exists a deadlock

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

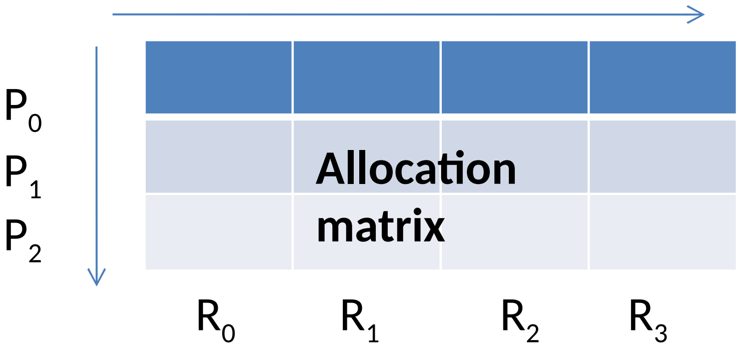
Several Instances of a Resource Type

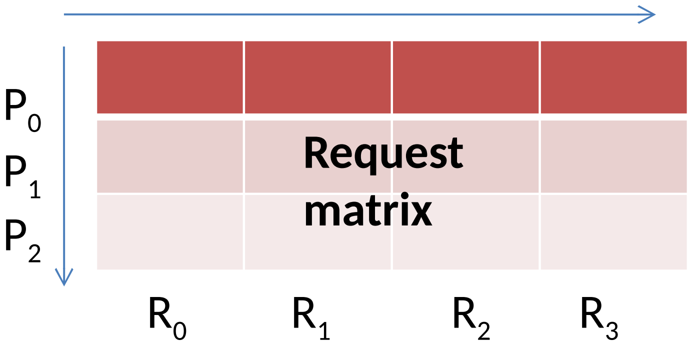
Let *n* = number of processes, and *m* = number of resources types.

- **Available**: A vector of length *m* indicates the number of available resources of each type.

R0	R1	R2	R3

- **Allocation**: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

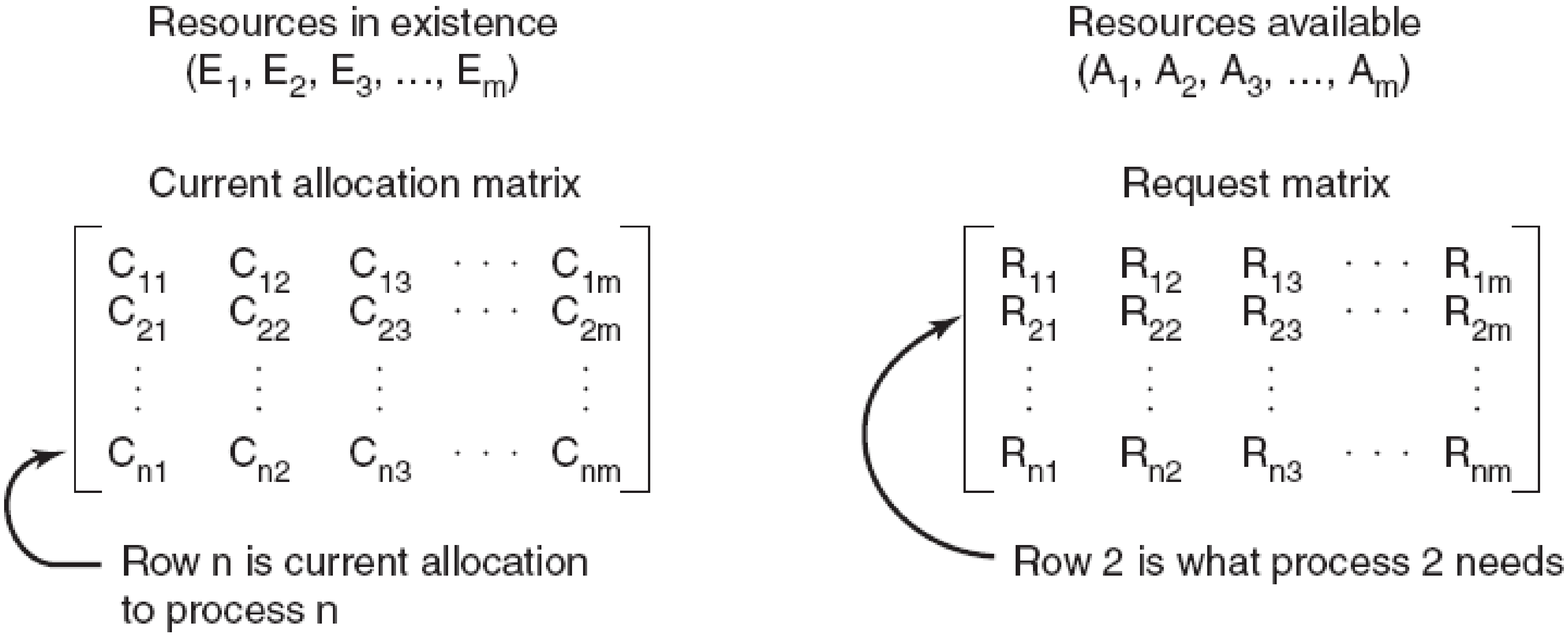




- **Request**: An *n* x *m* matrix indicates the current request of each process. If *Request* [*i*][*j*] = *k*, then process P_i is requesting *k* more instances of resource type R_j.

Several Instances of a Resource Type

Let *n* = number of processes, and *m* = number of resources types.



Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Detection Algorithm

- Define a relation ✹ over two vectors

- X and Y are two vectors of length n

- We say X ✹ Y

  Iff X[i] ✹Y[i] for all i=1, 2, …, n

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively

Initialize:

(a) *Work* = *Available*

(b) For *i* = 1,2, …, *n*, if *Allocation*$_i$ 📹 0, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*

2. Find an index *i* such that both:

(a) *Finish*[*i*] == *false*

(b) *Request*$_i$ ✹ *Work*

If no such *i* exists, go to step 4

Detection Algorithm (Cont.)

3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2

4. If *Finish*[*i*] == false, for some *i*, 1 ✹ *i* ✹ *n*, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then *P*$_i$ is deadlocked

Example of Detection Algorithm

- Five processes P_0 through P_4;
- three resource types
  A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*

Example (Cont.)

- P_2 requests an additional instance of type C

  Request

	A	*B*	*C*
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Home work

$$\begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ E = (& 4 & 2 & 3 & 1\ ) \end{array}$$

$$\begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ A = (& 2 & 1 & 0 & 0\ ) \end{array}$$

Available

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock?
- If deadlock frequent
  - Invoke detection algo frequently

- Invoke after each (waiting) resource request
  - Huge overhead
- CPU utilization drops

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
  - Expensive
- Abort one process at a time until the deadlock cycle is eliminated
  - Overhead=> invoke detection algo
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock:
Resource Preemption

- Selecting a victim – minimize cost
  - (# of resources holding, duration)

- Rollback – return to some safe state, restart process from that state

- Starvation – same process may always be picked as victim, include number of rollback in cost factor

Deadlock Avoidance

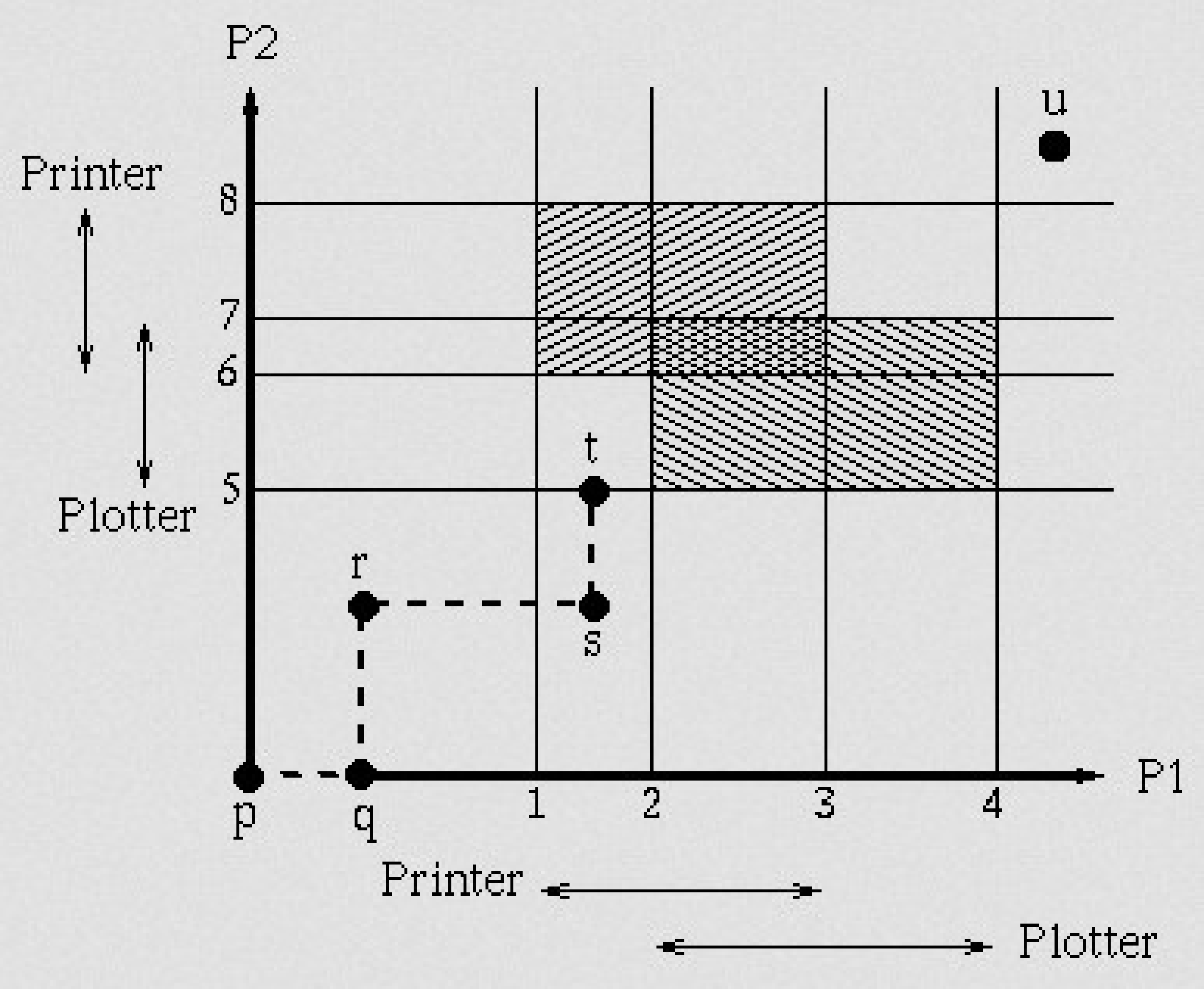
Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State



Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

- System is in **safe state** if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems
  - such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

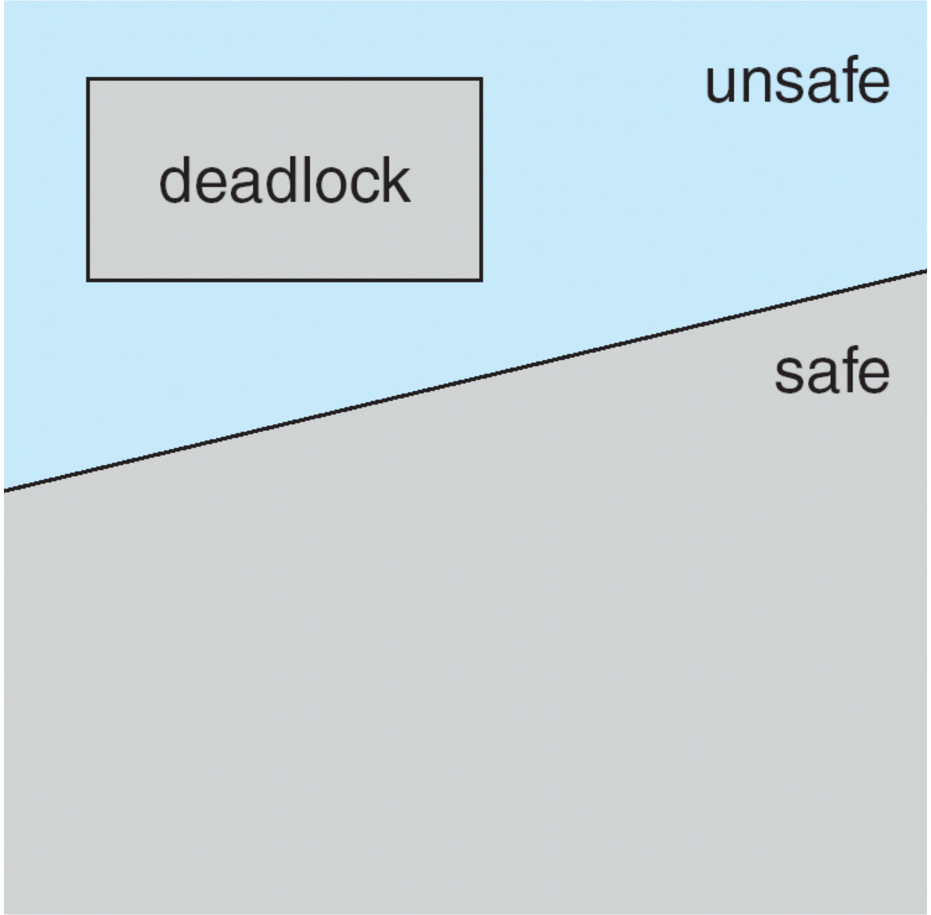
Basic Facts

- If a system is in safe state ▲ no deadlocks

- If a system is in unsafe state ▲ possibility of deadlock

- Avoidance ▲ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



Three processes P0, P1, P2

Resource R=12

State at time t_0

	Maximum need	Current allocation
P0	10	5
P1	4	2
P2	9	2

Free resource = 3

Safe sequence <P1, P0, P2> **Safe state**

State at time t_1

Allocate one resource to P2

Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the banker’s algorithm

Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line

- Claim edge converts to request edge when a process requests a resource

- Request edge converted to an assignment edge when the resource is allocated to the process

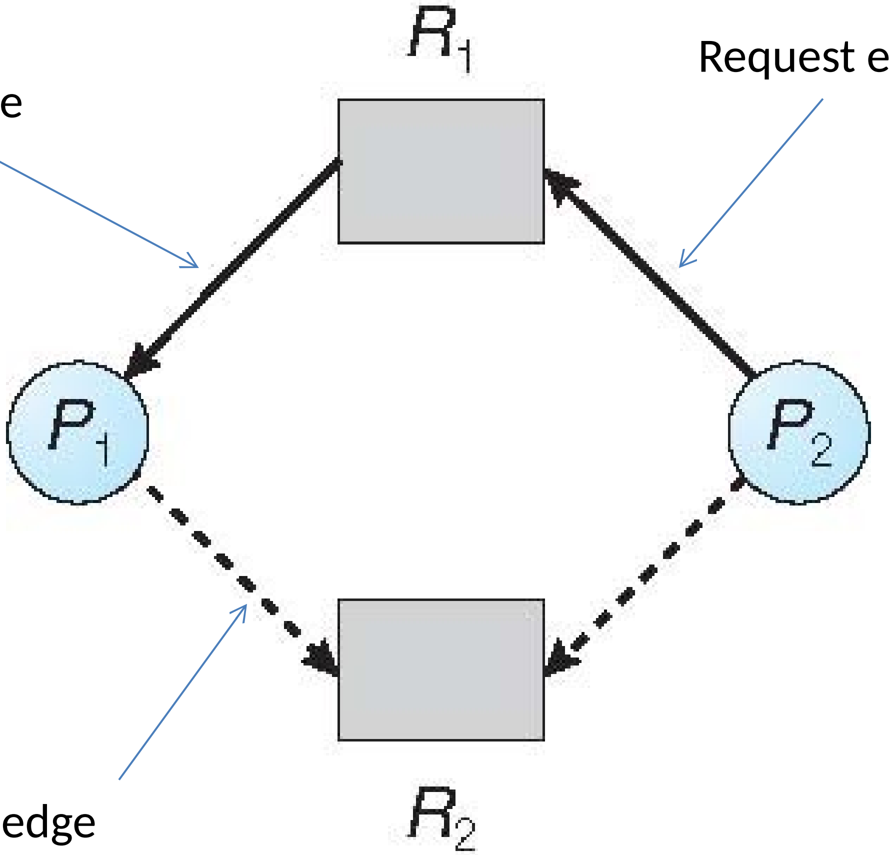
- When a resource is released by a process, assignment edge reconverts to a claim edge

- Resources must be claimed *a priori* in the system

Resource-Allocation Graph

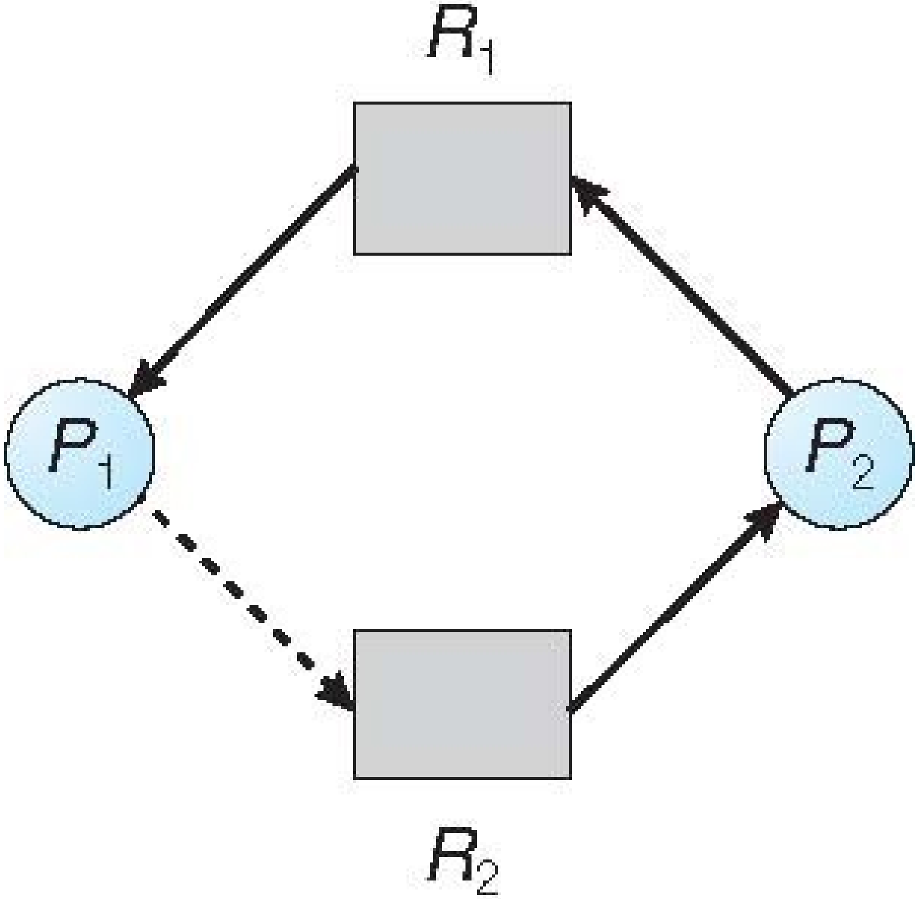
Assignment edge

Request edge



Claim edge

Unsafe State In Resource-Allocation Graph



Suppose that process P_2 requests a resource R_2

Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge **does not result in the formation of a cycle** in the resource allocation graph
- If no cycle
  - Safe state

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a set of resources
  - System decides whether the allocation is safe
- When a process requests a resource
  - it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max**: n x m matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task

***Need* [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*]**

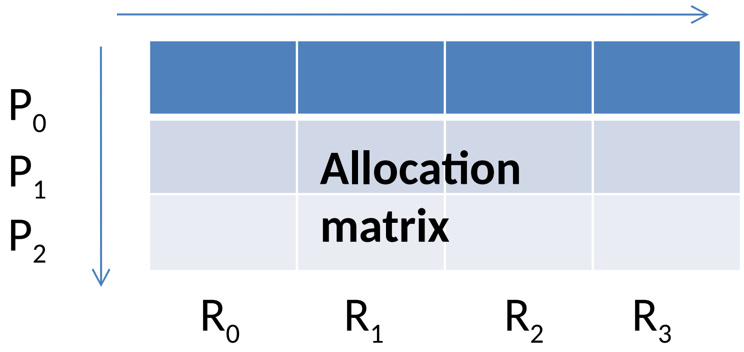
Several Instances of a Resource Type

Let *n* = number of processes, and *m* = number of resources types.

- **Available**: A vector of length *m* indicates the number of available resources of each type.

R0	R1	R2	R3

- **Allocation**: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.



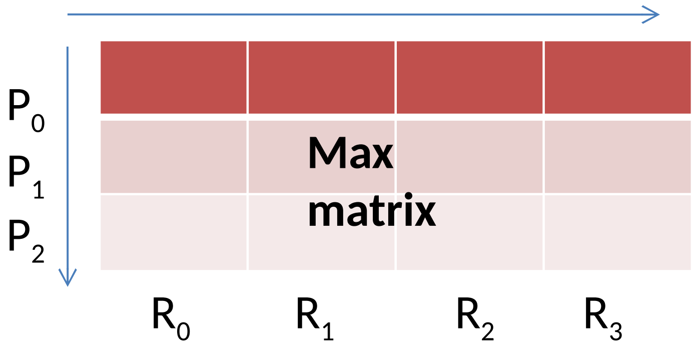
P_0

P_1

P_2

Allocation matrix

R_0 R_1 R_2 R_3



P_0

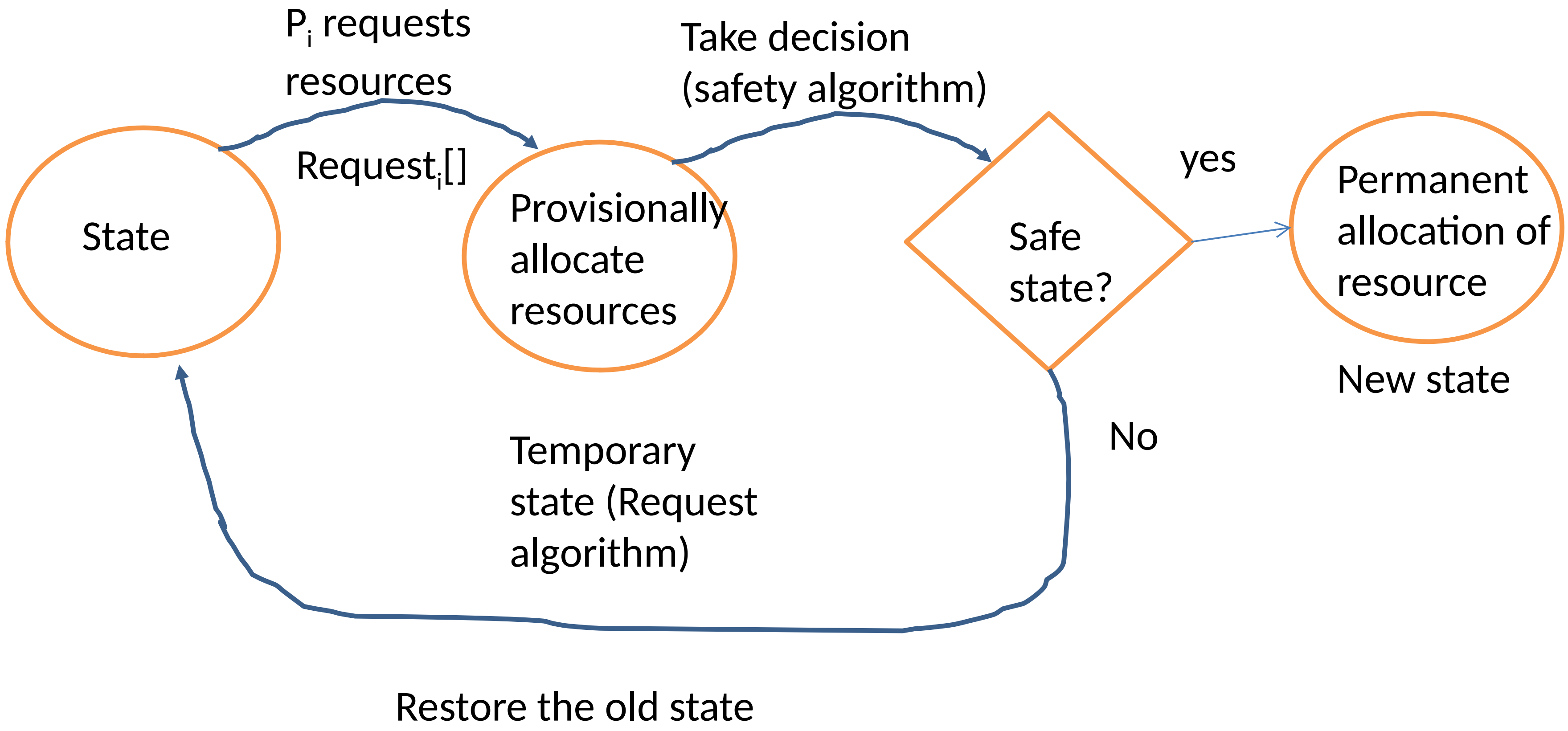
P_1

P_2

Max matrix

R_0 R_1 R_2 R_3

Deadlock avoidance : Flow chart for P_i



P_i requests resources

Request$_i$[]

Take decision (safety algorithm)

State

Provisionally allocate resources

Safe state?

yes

Permanent allocation of resource

New state

No

Temporary state (Request algorithm)

Restore the old state

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1

2. Find an *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i$ ✹ *Work*

   If no such *i* exists, go to step 4

3. Work = Work + Allocation$_i$
Finish[*i*] = *true*
go to step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i.

If $Request_i\,[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i$ ✹ $Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i$ ✹ *Available*, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- *If safe ▲ the resources are allocated to Pi*
- *If unsafe ▲ Pi must wait, and the old resource-allocation state is restored*

Example of Banker’s Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5instances), and *C* (7 instances)

Snapshot at time T_0:

	Allocation	***Max***	***Available***
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max – Allocation*

	<u>*Need*</u>
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max – Allocation*

Need

	A	*B*	*C*
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ✹ Available (that is, (1,0,2) ✹ (3,3,2) ⏶ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	**3 0 2**	**0 2 0**	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Deadlock Handling

Strategies for dealing with deadlocks:

1. Detection and recovery. Let deadlocks occur, detect them, take action.
2. Dynamic avoidance by careful resource allocation.
3. Prevention, by structurally negating one of the four required conditions.
4. Just ignore the problem.

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources
  - Read only file

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution,
  - Allow process to request resources only when the process has none
    - Release all the current resource and then try to acquire
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources,
  - Requests another resource that cannot be immediately allocated to it
    - Resources were allocated to some waiting process
  - Preempt the desired resource from waiting process
  - Allocate to current process
- **Circular Wait** – Impose a total ordering of all resource types
  - Require that each process requests resources in an increasing order of enumeration

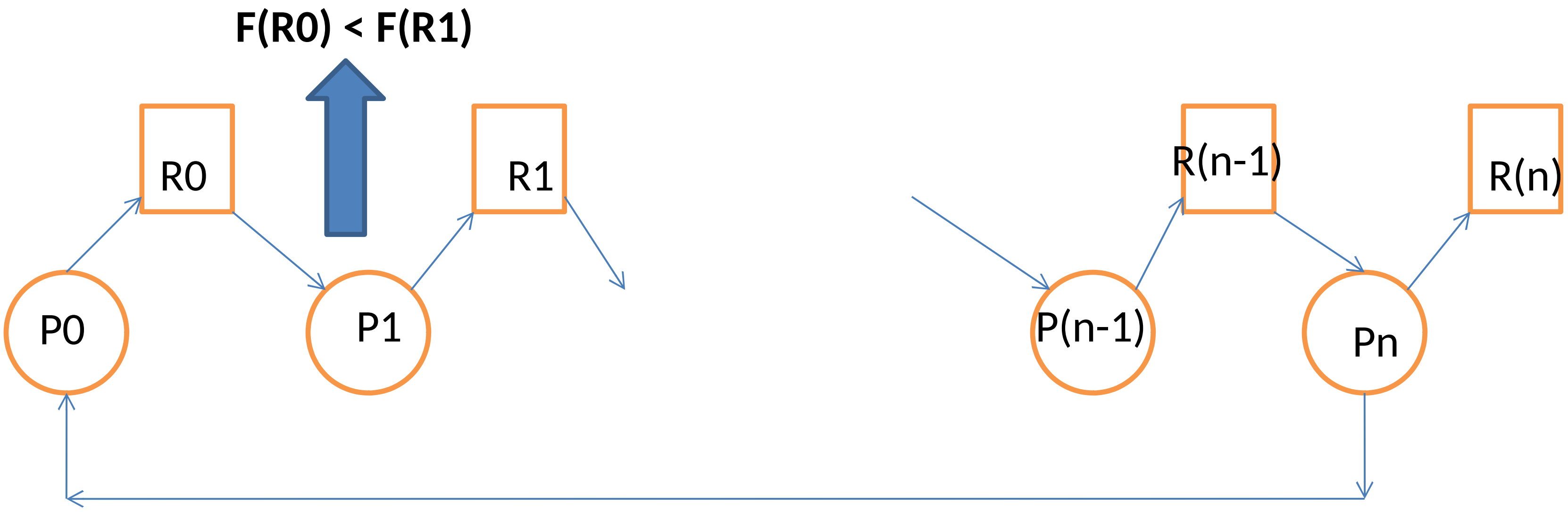
- Let R={R1, R2,……,Rm} set of resource type
- We assign unique integer with each type
- One to one function F:R✉N

F(tape drive)=1

F(disk)=5

F(printer)=12

- Protocol: Each process can request resource only in an increasing order.
- Initially request R_i, after that, it can request R_j
  - If and only if $F(R_j)>F(R_i)$
- Currently holding $R_{j;}$ Want to request R_i.
  - Must have released R_j



F(R0) < F(R1) < F(R2) <…………………….< F(Rn) < F(R0)