CS 5323 – OS II

Lecture 7 – Process Scheduling

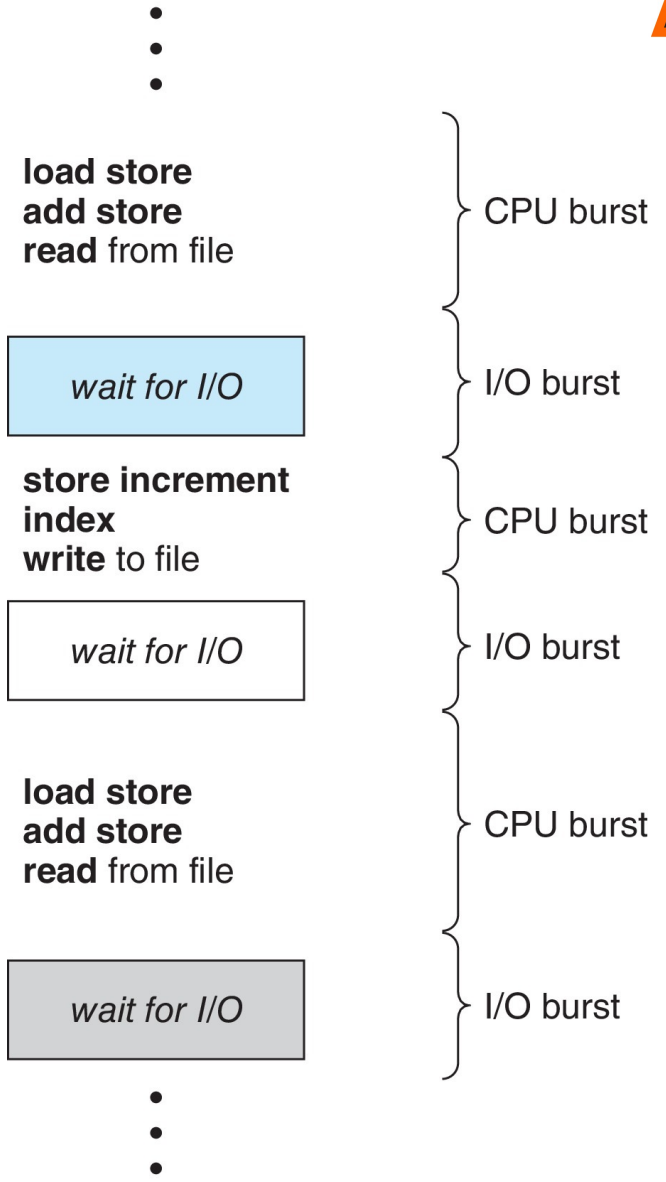
Logistics

- Quiz 3 posted.
  - Due 02/14/2022 11:59 pm
- Assignment 2.
  - Due 02/25/2022 11:59 pm
- Midterm 02/21/2022
  - Covers everything up to and including scheduling and deadlocks

Process Scheduling

Basic Concepts

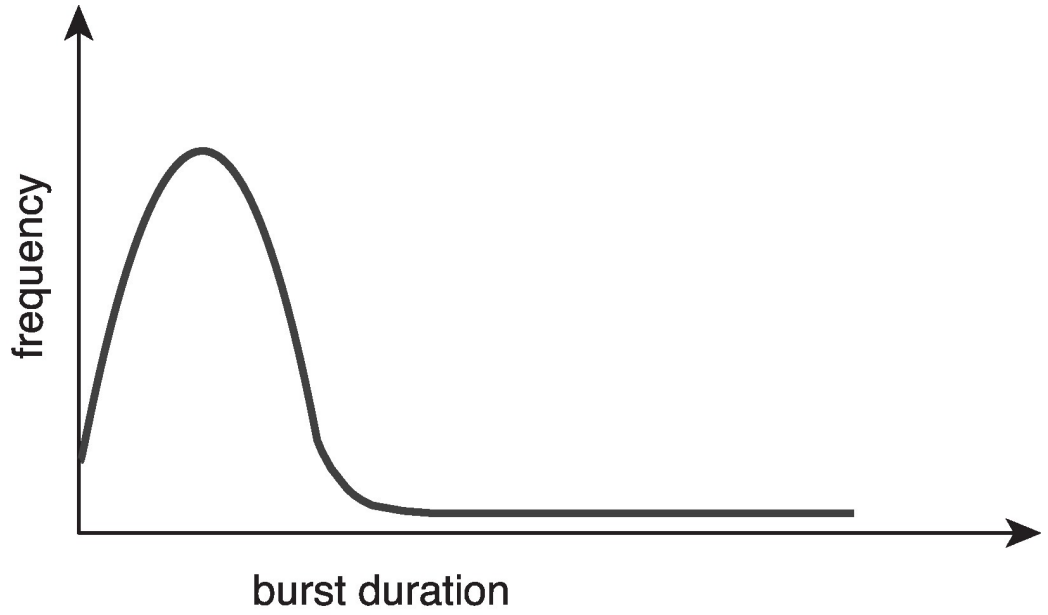
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times

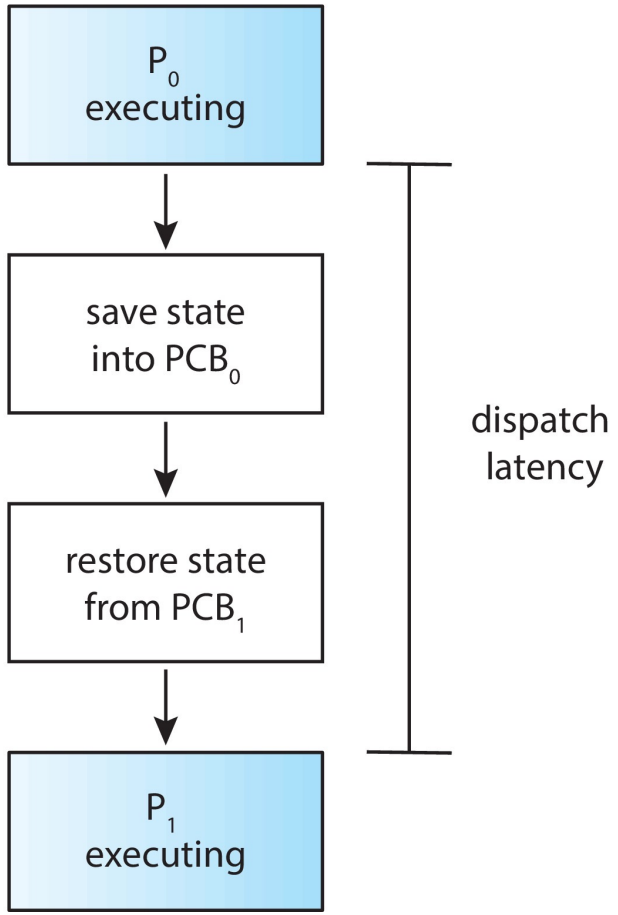
Large number of short bursts

Small number of longer bursts



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



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 (for time-sharing environment)

Scheduling Metrics

- **Completion Time**: Time taken for the execution to complete, starting from arrival time.
- **Turn Around Time**: Time taken to complete after arrival. In simple words, it is the difference between the Completion time and the Arrival time.
- **Waiting Time**: Total time the process has to wait before it's execution begins. It is the difference between the Turn Around time and the Burst time of the process.

Non-preemptive Scheduling

First- Come, First-Served (FCFS) Scheduling

- Simplest scheduling scheme
- Any process that requests the CPU time is given access through a queue
- When process requests CPU time, its PCB added to a linked-list, at the tail.
- Process at the head of the queue is removed and executed.

FCFS (contd.)

- Non-preemptive
- Once CPU has been allocated to a process, it runs till completion
  - Or I/O-bound waiting
- Troublesome for time-sharing systems.
  - Why?
  - Each user should get a share of the CPU at regular intervals.
    - Disastrous to allow one process to use CPU for extended periods

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:

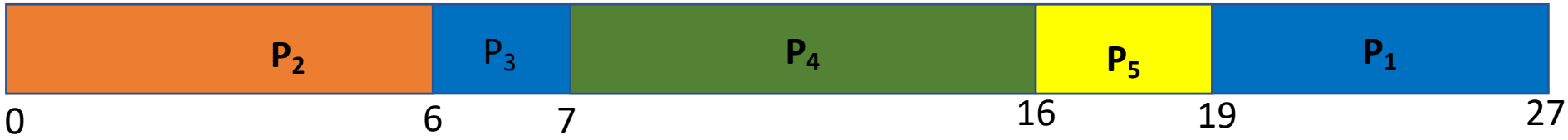
$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

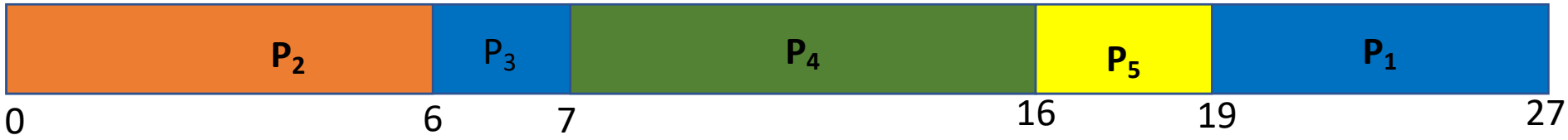
- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  1. Consider one CPU-bound and many I/O-bound processes

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)
- **Higher priority executes first!**



Process ID	Arrival Time	Burst Time	Priority
1	0	8	4
2	0	6	1
3	0	1	2
4	0	9	2
5	0	3	3



Process ID	Arrival Time	Priority	Burst Time	Completion Time	Turnaround time	Wait time
1	0	4	8	27	27	19
2	0	1	6	6	6	0
3	0	2	1	7	7	6
4	0	2	9	16	16	7
5	0	3	3	19	19	16

Av. TAT = 15 ms Av. WT = 9.6 ms

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

SJF – Example 2

Process ID	Arrival Time	Burst Time
1	0	8
2	0	6
3	0	1
4	0	9
5	0	3

Process ID	Arrival Time	Burst Time	Completion Time	Turnaround time	Wait time
1	0	8	18	18	10
2	0	6	10	10	4
3	0	1	1	1	0
4	0	9	27	27	18
5	0	3	4	4	1

Av. TAT = 12 ms Av. WT = 6.6 ms

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)
  - Preemptive
  - Nonpreemptive
- Problem ≡ **Starvation** – low priority processes may never execute
  - Famous case of a process on an IBM 7094 computer at MIT
    - Low priority process initiated in 1967 had been blocked until the system was shut down in 1973!
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Preemptive Scheduling

Preemptive Scheduling

- **Preemptive Scheduling** is defined as the scheduling which is done when the process changes from running state to ready state or from waiting for the state to ready state.
- Resources are allocated to execute the process for a certain period. After this, the process is taken away in the middle and is placed in the ready queue its bursts time is left and this process will stay in ready line until it gets its turn to execute.

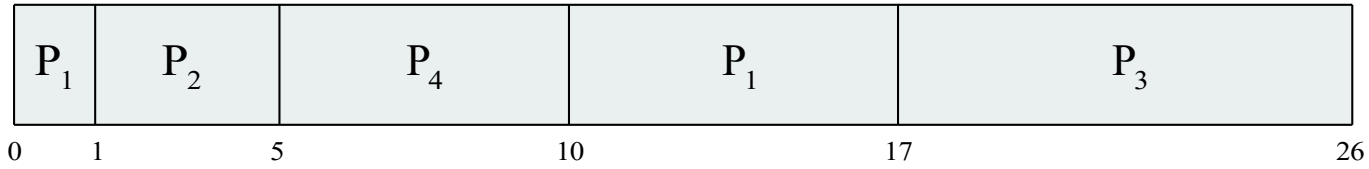
Shortest Remaining Time First

- Preemptive version of Shortest Job First scheduling
- Will reduce average wait time

Shortest Remaining Time First

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

Shortest Remaining Time First



Process ID	Arrival Time	Burst Time	Completion Time	Turnaround Time	Wait time
1	0	8	17	17	9
2	1	4	5	4	0
3	2	9	26	24	15
4	3	5	10	7	2

Av. TAT = 13 ms Av. WT = 6.5 ms

Round Robin (RR)

- 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.
- 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
  - q large $\Rightarrow$ Same as FCFS
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

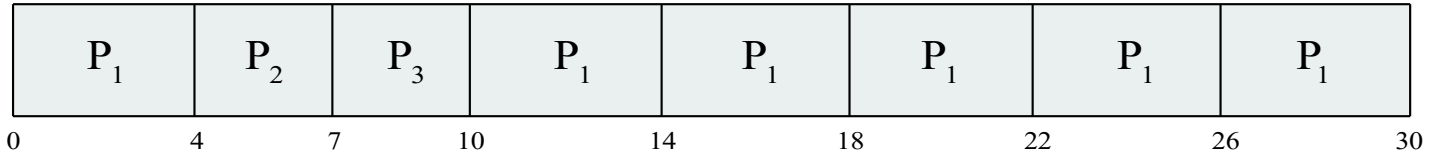
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



Process ID	Arrival Time	Burst Time	Completion Time	Turnaround time	Wait time
1	0	24	30	30	6
2	0	3	7	7	4
3	0	3	10	10	7

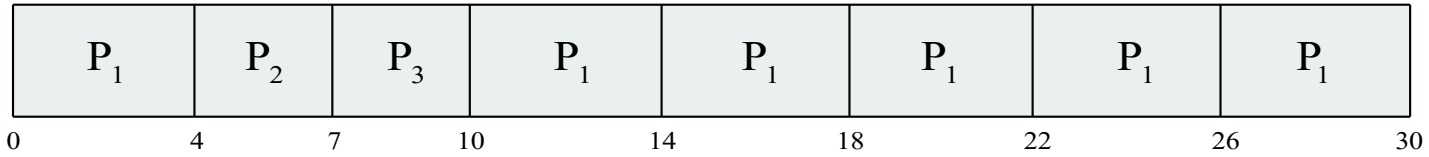
Av. TAT = 15.67 ms

Av. Wait Time = 5.67 ms

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

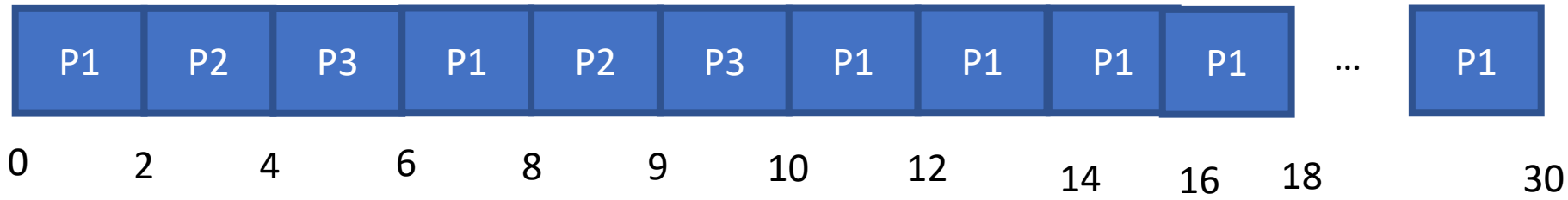
Example of RR with Time Quantum = 2

Process	Burst Time
P_1	24
P_2	3
P_3	3

Example of RR with Time Quantum = 2

Process	Burst Time
P_1	24
P_2	3
P_3	3

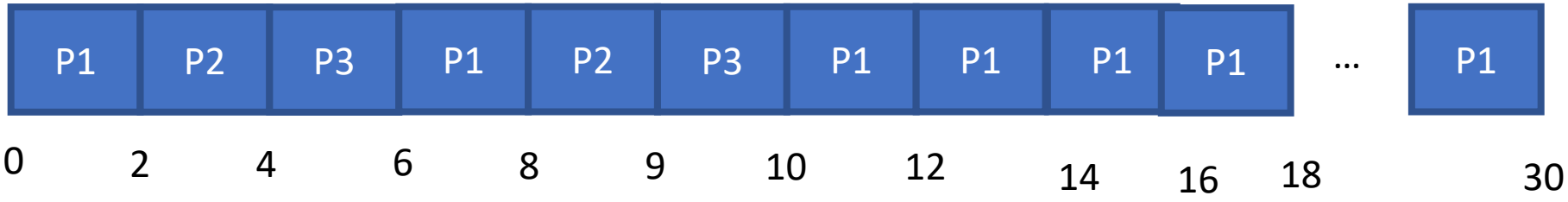
- The Gantt chart is:



Example of RR with Time Quantum = 2

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



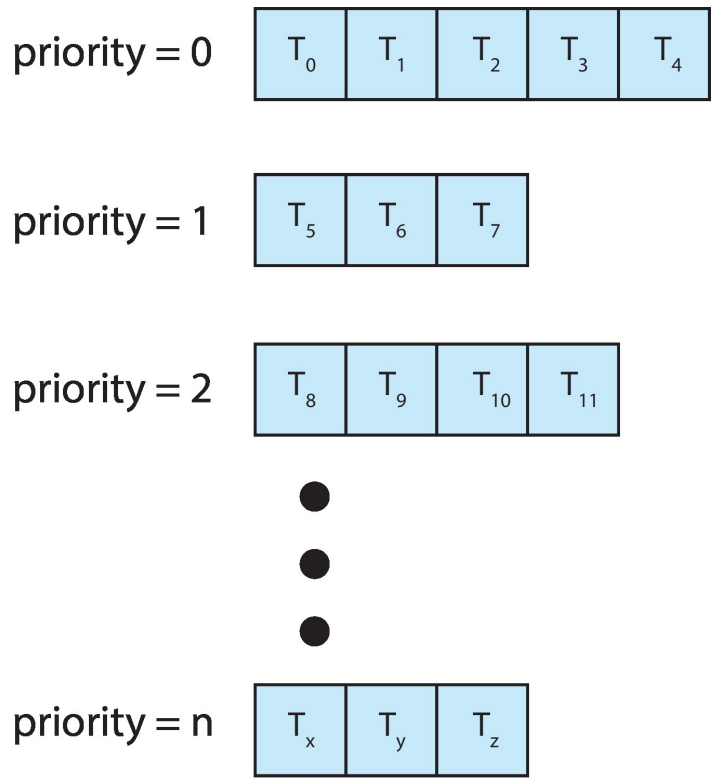
Process ID	Arrival Time	Burst Time	Completion Time	Turnaround time	Wait time
1	0	24	30	30	6
2	0	3	9	9	6
3	0	3	10	10	7

Av. WT = 6.33 ms

Av. TAT = 16.33 ms

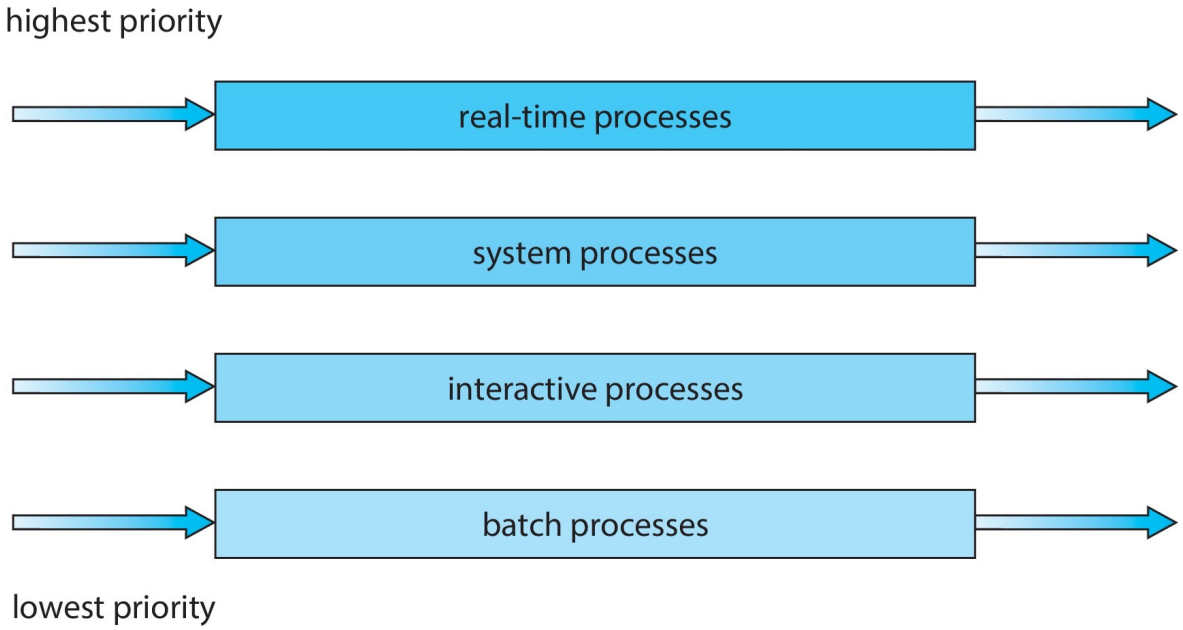
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



Multilevel Queue

- Prioritization based upon process type

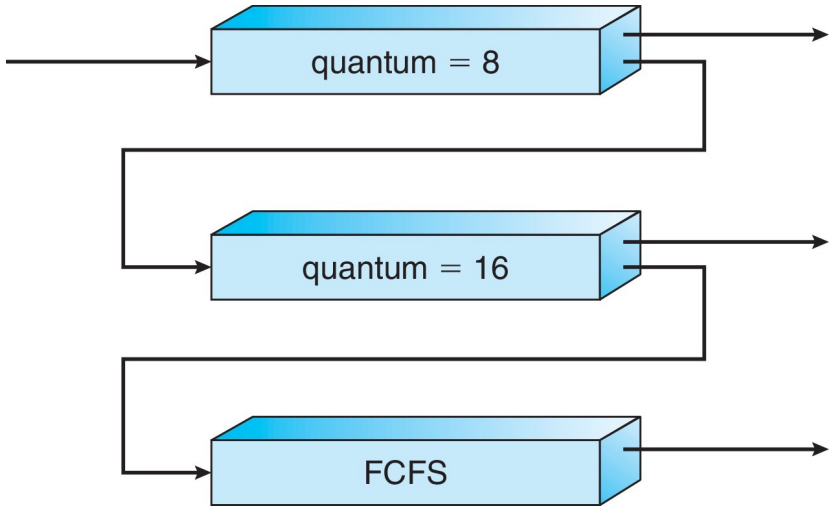


Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- 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

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 which is served FCFS
    - 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 served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Thread Scheduling

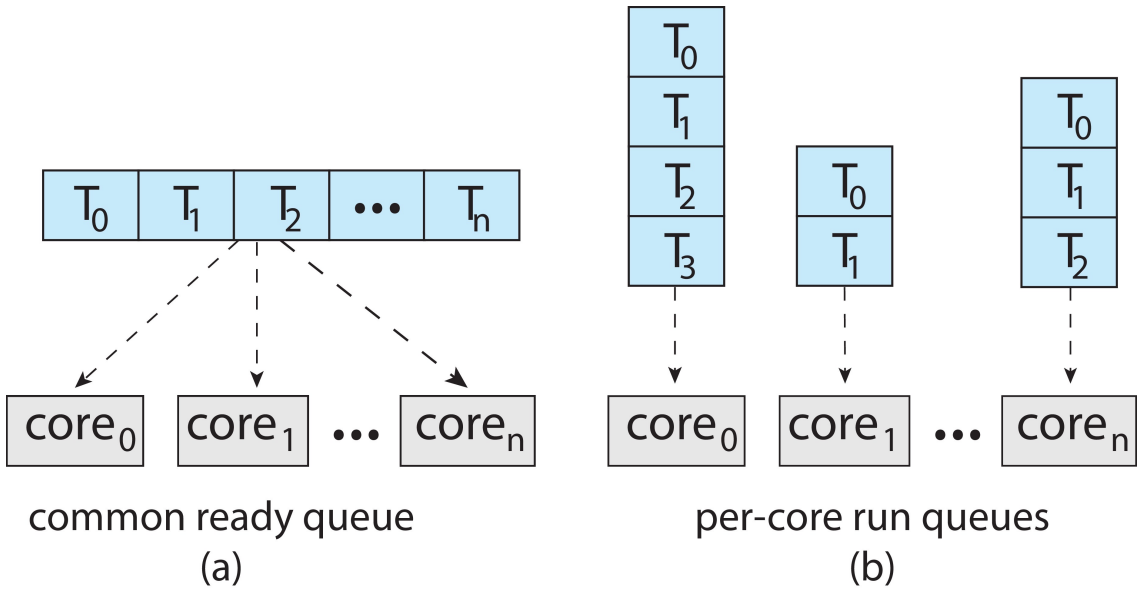
- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing

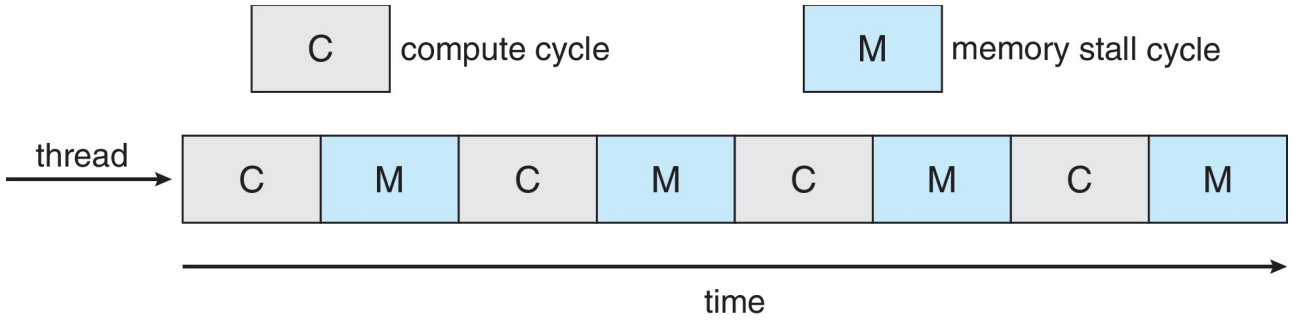
Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)



Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

Multiple-Processor Scheduling – Processor Affinity

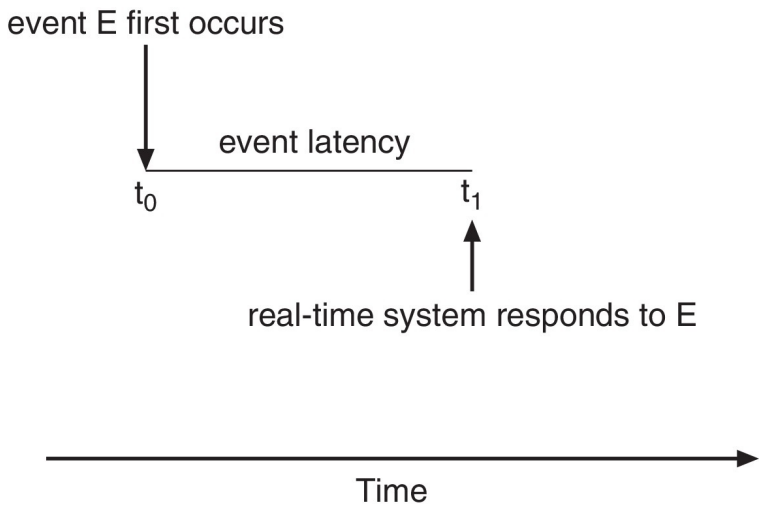
- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e. “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline

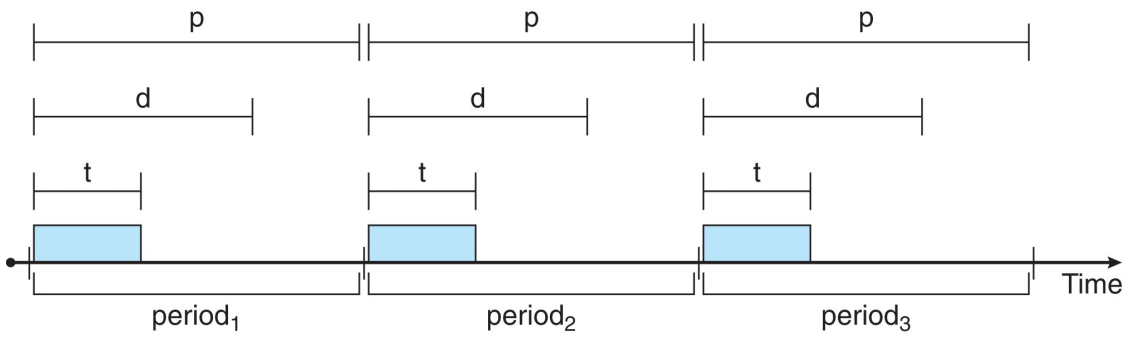
Real-Time CPU Scheduling

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** – time for schedule to take current process off CPU and switch to another



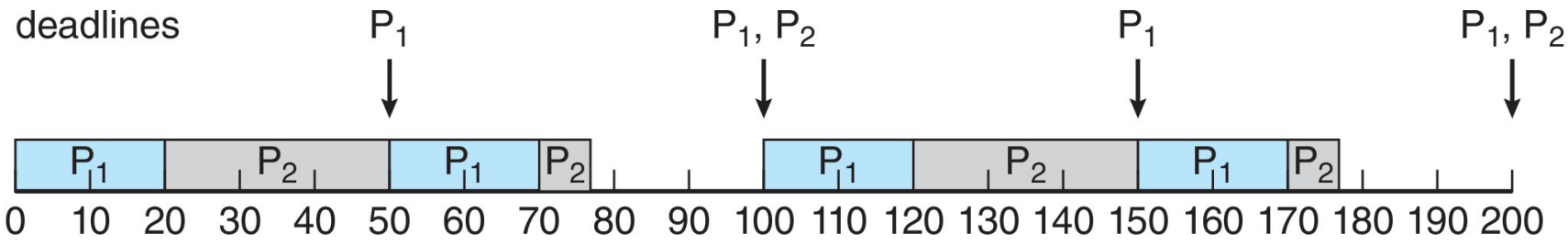
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d,* period *p*
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$



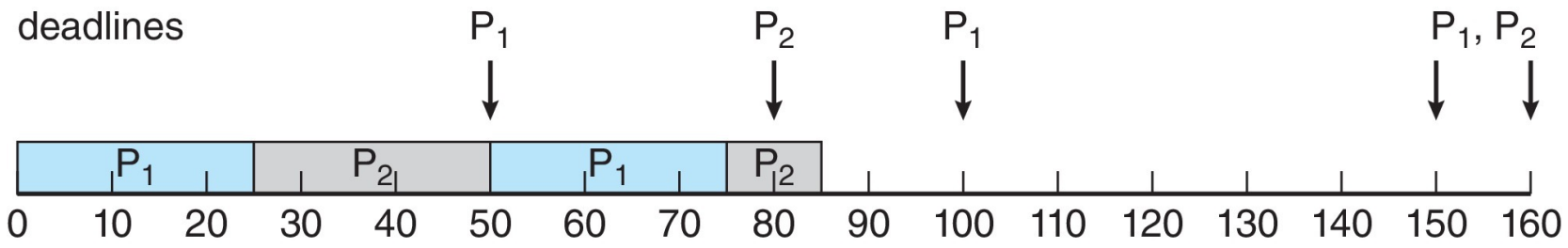
Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
  - Assumes that processing time of periodic process is same for each CPU burst
- Shorter periods = higher priority;
- Longer periods = lower priority
- Consider 2 processes – P1 and P2.
  - P1 = 50 ms and P2 = 100 ms. Processing time for P1 and P2 are 20 and 35.
- P_1 is assigned a higher priority than P_2.



Missed Deadlines with Rate Monotonic Scheduling

What if P1 runs for 25ms every 50 ms and P2 runs 35 ms for every 80 ms?



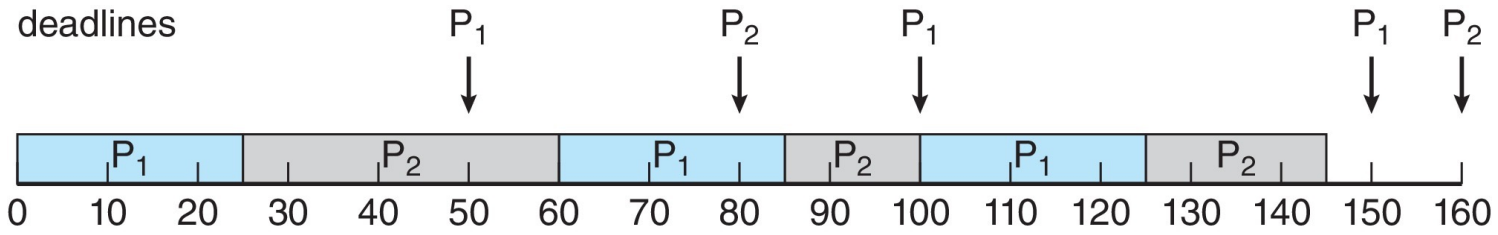
Process P2 misses finishing its deadline at time 80

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority
  - the later the deadline, the lower the priority
- Each process should declare deadline.
  - Priorities are dynamic!

EDFS, Example 1

- P1 runs for 25 ms every 50 ms. P2 runs 35 for every 80 ms.



Deadlocks

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

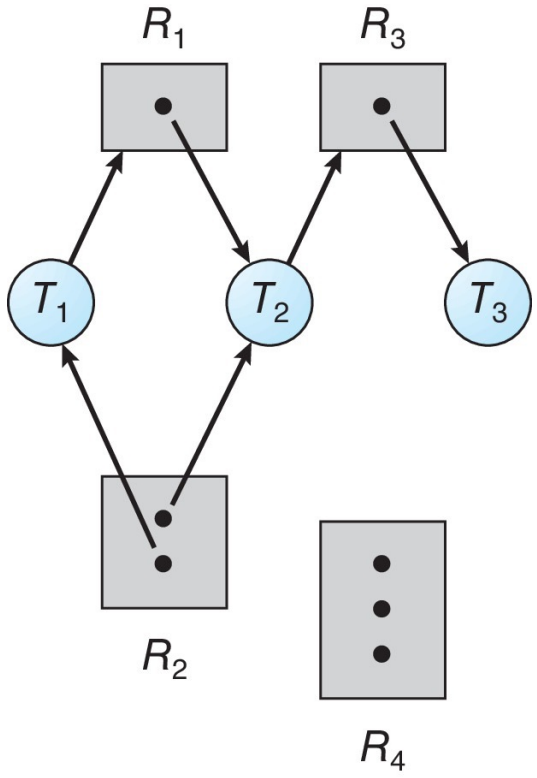
- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that 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, …, 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

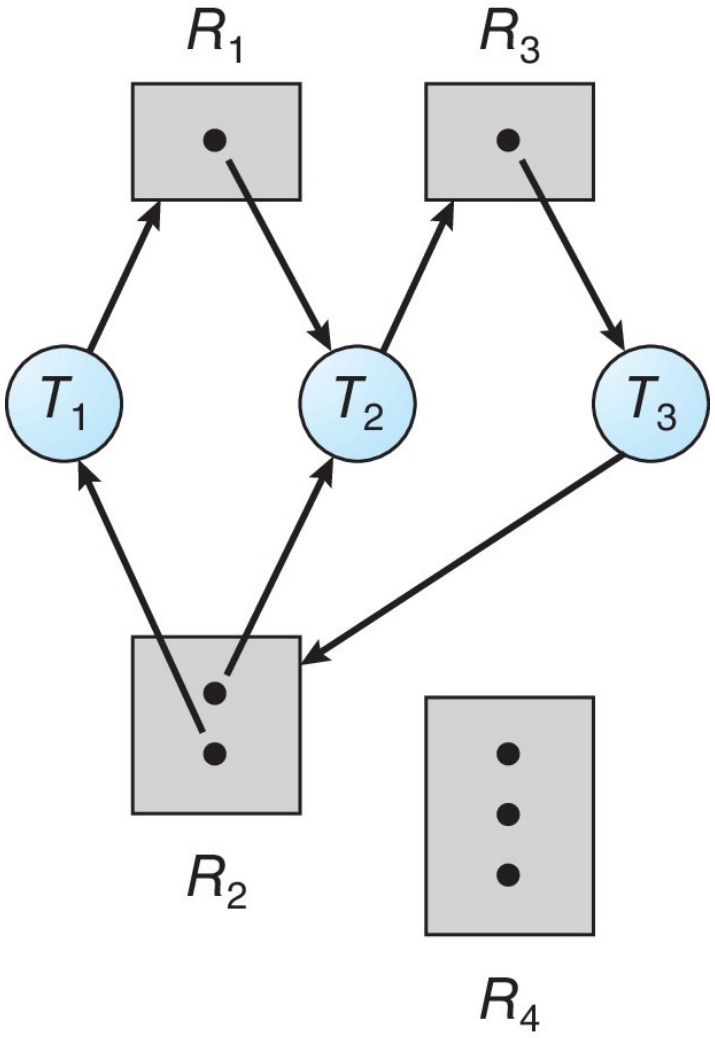
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i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

Resource Allocation Graph Example

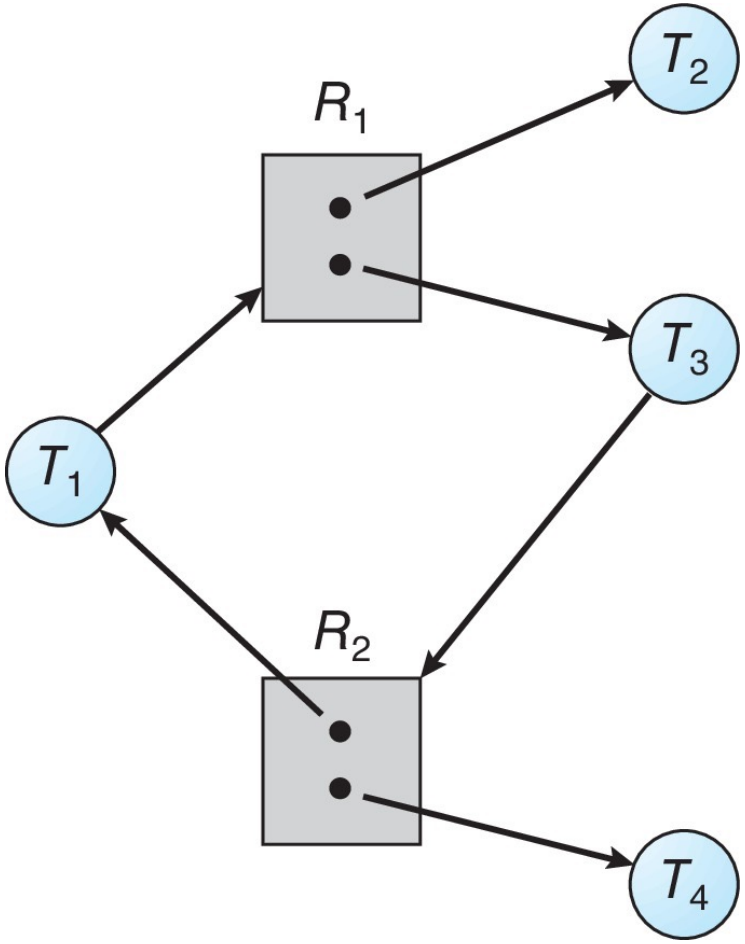
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.
  - Ostrich Algorithm

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **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, or allow process to request resources only when the process has none allocated to it.
  - 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, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e. mutex locks) a unique number.
- Resources must be acquired in order.
- If:

`first_mutex = 1`
`second_mutex = 5`

code for **`thread_two`** could not be written as follows:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```