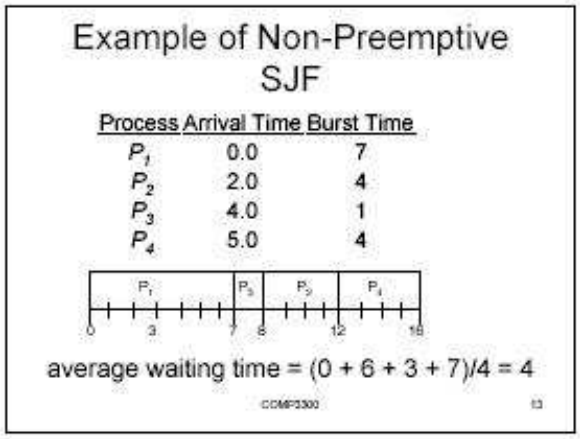

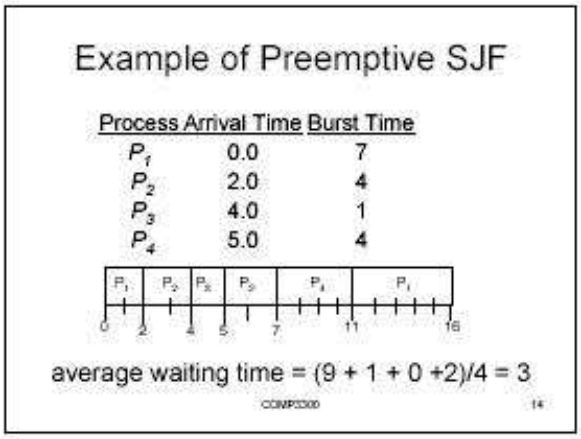

Figure 1: Example of non-preemptive SJF and example of preemptive SJF.

- Shortest Remaining Time Next (SRTF)
  - preemptive version of the SJF
  - if new process arrives with CPU burst length remaining time of current executing process, preempt: Shortest-Remaining-Time-First

0.1 Scheduling in Interactive Systems

- Round-Robin Scheduling (RR) (See Fig. 2)
  - RR reduces the penalty that short jobs suffer with FCFS by preempting running jobs periodically
  - Scheduled thread is given a time slice
  - The CPU suspends the current job when the reserved *quantum* (*time-slice*) is exhausted
  - The job is then put at the end of the ready queue if not yet completed
  - Advantages;
    * no starvation
    * Fair allocation of CPU across jobs

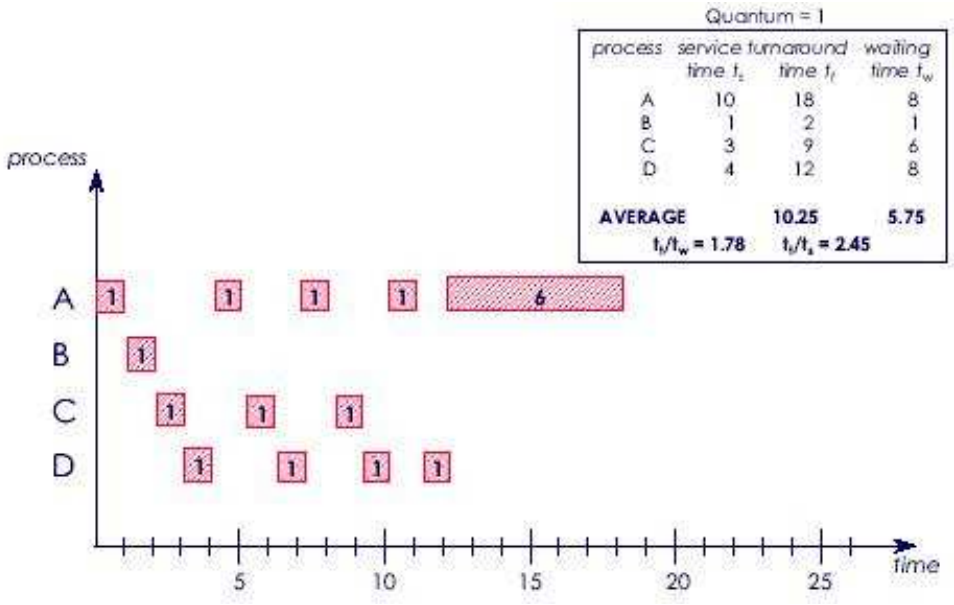

Figure 2: An example to Round Robin.

    - Low average waiting time when job lengths vary widely
  - Disadvantages;
    - Poor average waiting time when job lengths are identical; Imagine 10 jobs each requiring 10 time slices, all complete after about 100 time slices, even FCFS is better!
    - The critical issue with the RR policy is the length of the quantum. If it is too short, then the CPU will be spending more time on context switching. Otherwise, interactive processes will suffer

- Priority Scheduling (See Fig. 3)
  - Each process is assigned a priority (e.g., a number)
  - The ready list contains an entry for each process ordered by its priority
  - The process at the beginning of the list (highest priority) is picked first
    - Scheduler will always choose a thread of higher priority over one of lower priority

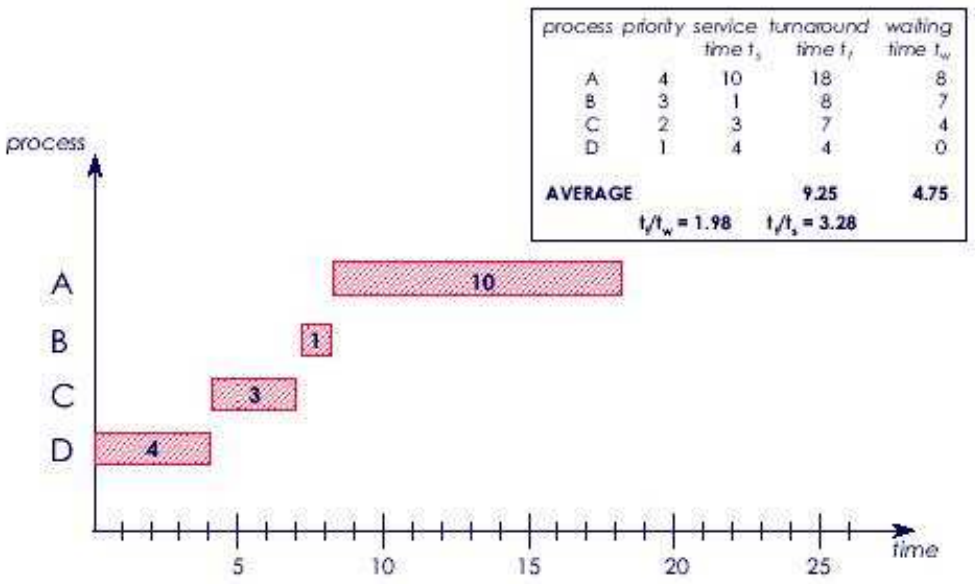

Figure 3: An example to Priority-based Scheduling.

  - ∗ Implemented via multiple FCFS ready queues (one per priority)
- Lower-priority may suffer starvation
- A variation of this scheme allows preemption of the current process when a higher priority process arrives
- Another variation of the policy adds an aging scheme where the priority of a process increases as it remains in the ready queue; hence, will eventually execute to completion

• Multiple Queues (See Fig. 4)

- Multi-Level Queue (MLQ) scheme solves the mix job problem (e.g., batch, interactive, and CPU-bound) by maintaining separate "ready" queues for each type of job class and apply different scheduling algorithms to each
- Multi-level feedback queue
  - ∗ this is a variation of MLQ where processes (jobs) are *not* permanently assigned to a queue when they enter the system

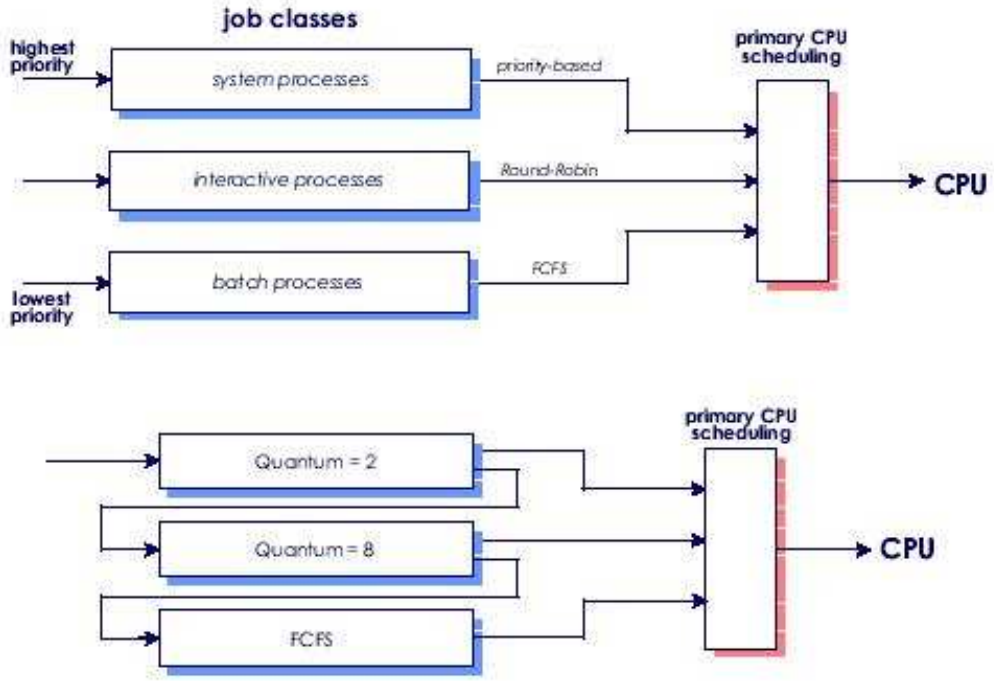

Figure 4: Multi-level queue and Multi-level feedback queue (lower).

  - In this approach, if a process exhausts its time quantum (i.e., it is CPU-bound), it is moved to another queue with a longer time quantum and a lower priority
  - The last level usually uses FCFS algorithm in this scheme

- Lottery Scheduling
  - Implemented guaranteed access to resources is, in general, difficult!
  - process gets "lottery tickets" for various resources
  - more lottery tickets imply better access to resource
  - Advantages: Simple, Highly responsive, Allows cooperating processes/threads to implement individual scheduling policy (exchange of tickets)
  - Process A: 15% of CPU time, Process B: 25% of CPU time, Process C: 5% of CPU time, Process D: 55% of CPU time How many tickets should each process get to achieve this?

0.2 Policy versus Mechanism

- Separate what is allowed to be done with how it is done; a process knows which of its children threads are important and need priority
- Scheduling algorithm parameterized; mechanism in the kernel
- Parameters filled in by user processes; policy set by user process

1 Deadlock

- **Deadlock** is defined as the *permanent* blocking of a set of processes that compete for system resources, including database records or communication lines
- Unlike other problems in multiprogramming systems, there is no efficient solution to the deadlock problem in the general case
- Deadlock **prevention**, **by design**, is the "best" solution
- Deadlock occurs when a set of processes are in a wait state, because each process is waiting for a resource that is held by some other waiting process
- None will release what they hold until they get what they are waiting for
- Therefore, all deadlocks involve conflicting resource needs by two or more processes
- Example: Unordered Mutex; Two threads accessing two locks

$Semaphore\ m[2] = \{1, 1\};$	$//binarysemaphore$
$Thread1$	$Thread2$
$m[0].P();$	$m[1].P();$
$m[1].P();$	$m[0].P();$
$//access\ shared\ data$	$//access$
$m[1].V();$	$m[0].V();$
$m[0].V();$	$m[1].V();$

- What happens if Thread1 grabs $m[0]$ and Thread2 grabs $m[1]$? (P means down operation and V means up operation)

1.1 Resources

- Classification of resources–I, Two general categories of resources can be distinguished:
  - **Reusable**: something that can be safely used by one process at a time and is not depleted by that use.
    - Processes obtain resources that they later release for reuse by others.
    - Examples are processors, I/O channels, main and secondary memory, files, specific I/O devices, databases, and semaphores.
    - In case of two processes and two resources, deadlock occurs if each process holds one resource and requests the other.
  - **Consumable**: these can be created and destroyed.
    - When a resource is acquired by a process, the resource ceases to exist.
    - Examples are interrupts, signals, messages, and information in I/O buffers
    - Deadlock may occur if a Receive message is blocking
    - May take a rare combination of events to cause deadlock
- Classification of resources–II, One other taxonomy again identifies two types of resources:
  - **Preemptable**: these can be taken away from the process owning it with no ill effects (needs save/restore). E.g., memory or CPU.
  - **Non-preemptable**: cannot be taken away from its current owner without causing the computation to fail. E.g., printer or floppy disk.
- Deadlocks occur when sharing *reusable* and *non-preemptable* resources

1.2 Introduction to Deadlocks

1.2.1 Conditions for Deadlock (See Fig. 5)

- Four conditions that must hold for a deadlock to be possible:
  - **1. Mutual exclusion**: processes require exclusive control of its resources (not sharing), only one process may use a resource at a time

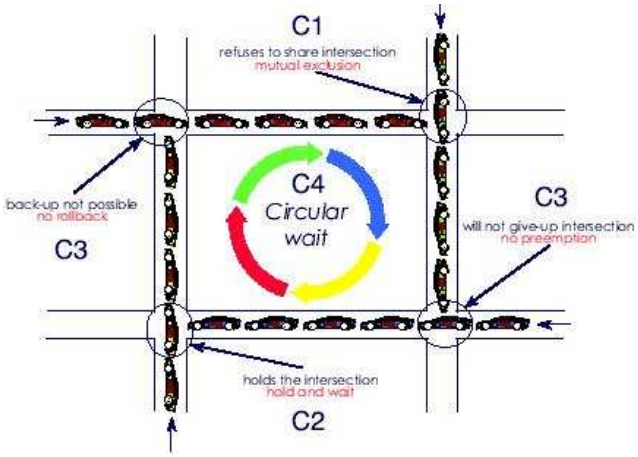

Figure 5: An example to Deadlock.

  - **2. Hold and wait**: process may wait for a resource while holding others
  - **3. No preemption**: process will not give up a resource until it is finished with it. Also, **processes are irreversible**: unable to reset to an earlier state where resources not held
  - **4. Circular wait**: each process in the chain holds a resource requested by another, there exists set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 waiting for resource held by P_1, P_1 waiting for resource held by P_2, ..., P_{n-1} waiting for resource held by P_n, P_n waiting for resource held by P_0

- If any one of the necessary conditions is prevented a deadlock need not occur. For example:
  - Systems with only simultaneously shared resources cannot deadlock; Negates *mutual exclusion*.
  - Systems that abort processes which request a resource that is in use; Negates *hold and wait*.
  - Preemptions may be possible if a process does not use its resources until it has acquired all it needs; Negates *no preemption*.
  - Transaction processing systems provide checkpoints so that processes may back out of a transaction; Negates *irreversible process*.
  - Systems that prevent, detect, or avoid cycles; Negates *circular wait*. Often, the preferred solution.

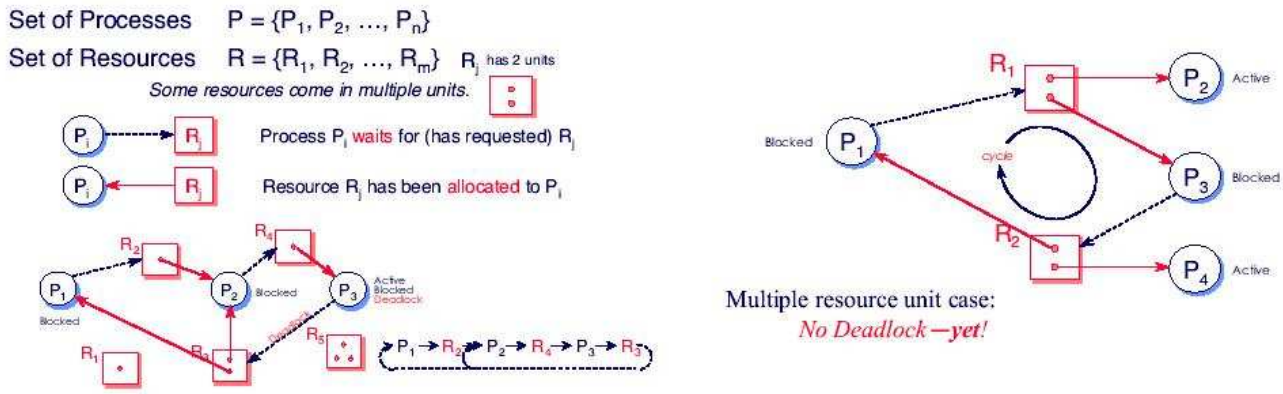

Figure 6: Resource Allocation Graphs, in the right one, either P_2 or P_4 could relinquish a resource allowing P_1 or P_3 (which are currently blocked) to continue.

1.2.2 Deadlock Modeling (See Fig. 6)

- Cycle is a *necessary condition* for a deadlock
- But when dealing with multiple unit resources –*not sufficient*
- A *knot* must exist–a cycle with no non-cycle outgoing path from any involved node
- At the moment assume that:
  - a process *halts* as soon as it waits for one resource,
  - processes can wait for only *one* resource at a time
- In general, four strategies are used for dealing with deadlocks:
  - **Ignore (The Ostrich Algorithm)**: stick your head in the sand and pretend there is no problem at all.
  - **Prevention**: design a system in such a way that the possibility of deadlock is excluded *a priori* (e.g., compile-time/statically, by design)
  - **Avoidance**: make a decision dynamically checking whether the request will, if granted, potentially lead to a deadlock or not (e.g., run-time/dynamically, before it happens)

- **Detection and Recovery**: let the deadlock occur and detect when it happens, and take some action to recover after the fact (e.g., run-time/dynamically, after it happens)

1.3 The Ostrich Algorithm

- Different people react to this strategy in different ways:
  - **Mathematicians**: find deadlock totally unacceptable, and say that it must be prevented at all costs.
  - **Engineers**: ask how serious it is, and do not want to pay a penalty in performance and convenience.
- The UNIX approach is just to ignore the problem on the assumption that most users would prefer an occasional deadlock, to a rule restricting user access to only one resource at a time
- The problem is that the prevention price is high, mostly in terms of putting inconvenient restrictions on processes

1.4 Deadlock Detection and Recovery

- This technique does not attempt to prevent deadlocks; instead, it lets them occur
- The system detects when this happens, and then takes some action to recover after the fact (i.e., is reactive)
- With deadlock detection, requested resources are granted to processes whenever possible
- Periodically, the operating system performs an algorithm that allows it to detect the circular wait condition
- A check for deadlock can be made as frequently as resource request, or less frequently, depending on how likely it is for a deadlock to occur
- Checking at each resource request has two advantages: It leads to early detection, and the algorithm is relatively simple because it is based on incremental changes to the state of the system
- On the other hand, such frequent checks consume considerable processor time

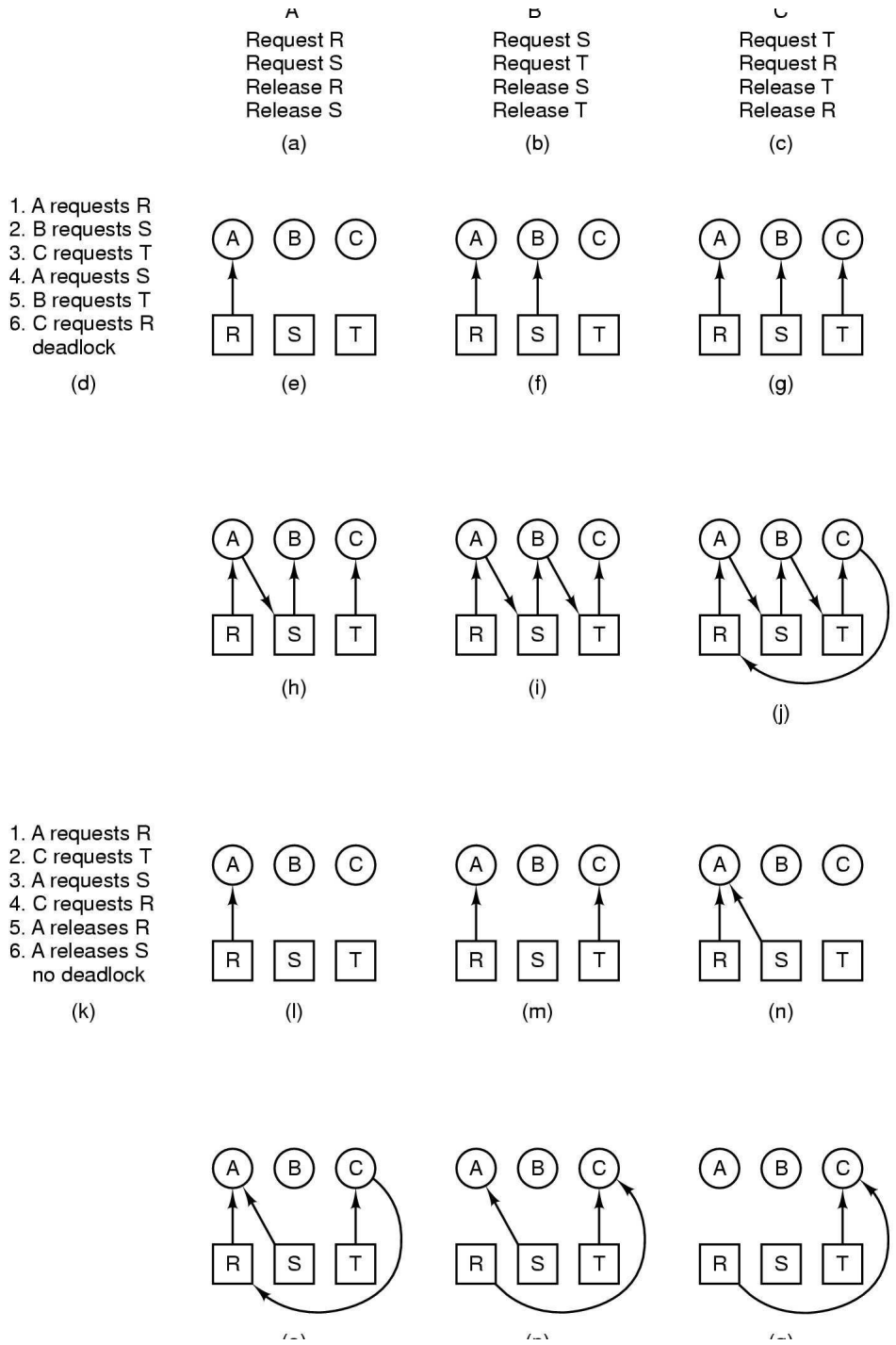

Figure 7: An example of how deadlock occurs and how it can be avoided.

- Once the deadlock algorithm has successfully detected a deadlock, some strategy is needed for recovery
- There are various ways:
  - Recovery through *Preemption*; In some cases, it may be possible to temporarily take a resource away from its current owner and give it to another.
  - Recovery through *Rollback*; If it is known that deadlocks are likely, one can arrange to have processes *checkpointed* periodically. For example, can undo transactions, thus free locks on database records. This often requires extra software functionality.
  - Recovery through *Termination*; The most trivial way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. Warning! *Irrecoverable losses or erroneous results may occur, even if this is the least advanced process.*