• **Deadlock Introduction**
  o Mutual exclusion can lead to deadlock
    ▪ Process A has exclusive control of something that process B needs, and vice versa
    ▪ Both processes block indefinitely waiting for the other
    ▪ This isn’t restricted to objects in memory
  o Resources
    ▪ To maintain generality, we’ll study all entities that can be given mutually exclusive access
      • Hardware – tape drive, CD-ROM, printer, etc.
      • Information – record in database, mutex, shared memory
    ▪ Resource: anything that must be acquired, used, and released over the course of time
  o Two types – preemptable and nonpreemptable
    • Preemptable – resource that can be taken away from a process with no ill effects
      o Example: main memory
        ▪ Swapping can be used to give memory allocated to a process A to process B without A ever knowing
    • Nonpreemptable – resource that cannot be taken away arbitrarily without causing the computation to fail
      o Example: CD writer
        ▪ If burning a CD has already started, taking the writer away from the current process will likely ruin the CD
  o Assumptions when studying deadlocks
    • Only consider nonpreemptable resources
      o Deadlocks involving preemptable resources are usually easily resolved by reassigning the resource temporarily
    • Resource use will be restricted to the following
      o Request the resource
      o Use the resource
      o Release the resource
    • Processes are blocked when resource isn’t immediately available
      o Could also enter a tight loop requesting resource, but it might as well be blocked because no active work can be done
    • No interrupts possible to wake up a blocked process
      o Ensures that alarms can’t be sent to wake up processes
      o Removes the possibility of external interference with resource allocation/use, which simplifies our analysis of the situation
Mutexes are often used to protect resources (either in the OS or in the process itself)

```c
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

- Deadlock occurs when process_A successfully calls down on resource_1 and process_B calls down on resource_2

- Formalizing Deadlocks
  - Definition: 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
    - All processes are waiting, so none will be able to wake up and cause an event that will wake up other members of the set.
  - There are multiple kinds of deadlocks (we’ll introduce more later)
- Resource deadlock – most common
  - The event that each process is waiting for is the release of some resource currently held by another process
  - Four conditions must hold for there to be a resource deadlock
    - Mutual exclusion condition – Each resource is either currently assigned to exactly one process or is available
      - If multiple processes can have access to the resource whenever they want, deadlock is impossible
    - Hold and wait condition – Processes currently holding resources that were granted earlier can request new resources
      - If every process can only hold one resource at a time, they will each either have the resource they currently need or will be blocked waiting for it (without holding any other resource). No process can be blocked indefinitely while holding a resource, so deadlock is impossible
- No preemption condition – Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them
  - If resources can be preempted, deadlocks can easily be avoided
- Circular wait condition – There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.
  - Without the circular wait, there is some resource that, when made available, will cause a cascade of activity that will avoid deadlock
- Resource Allocation Graphs – modeling resource deadlocks
  - We can use directional graphs to model the four conditions
  - Assumptions
    - Two kinds of nodes
      - Processes – represented as circles
      - Resources – represented as squares
    - Directed links between the nodes have meaning
      - Arc from resource node to process node (arrow pointing to circle)
        - Resource currently held by that process
      - Arc from process node to resource node (arrow pointing to square)
        - Process blocked waiting for the resource
- Deadlock example
• C is waiting for resource T, and D is waiting for resource U
  o C holds U and D holds T, so deadlock occurs
• A cycle in this directed graph means there is a deadlock
  ▪ Resource allocation graphs can be used as a tool to see if a given request/release sequence leads to deadlock
  • Carry out requests/releases, and after every step check to see if there is a cycle in the graph.
  • Assume processes A, B, and C and resources R, S, and T with sequences of requests and releases as follows

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request R</td>
<td>Request S</td>
<td>Request T</td>
<td>Request R</td>
</tr>
<tr>
<td>Request S</td>
<td>Request T</td>
<td>Request T</td>
<td></td>
</tr>
<tr>
<td>Release R</td>
<td>Release S</td>
<td>Release T</td>
<td></td>
</tr>
<tr>
<td>Release S</td>
<td>Release T</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Possible option – avoid parallelism
  o OS is free to run unblocked processes in any way it wants
  o Run A until completion, then B until completion, then C
    ▪ No resource contention, so no deadlock possible
  o If there is no I/O with these processes, that might be efficient
    ▪ That’s usually not the case, so it’s not an appropriate solution
• Round robin
  o Might produce the following sequence
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R

Because there is a cycle, this sequence produces a deadlock

- Modified – if it is known that scheduling in a particular way will cause a deadlock, simply don’t schedule the process until it is safe to do so
  - Suspend B instead of granting it S

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

- After last step, B can request and receive S, then wait on T:
  - There is no cycle, so no deadlock can occur
  - B will just wait for C to release T.
Knowing that a sequence of requests/releases will cause a deadlock is not trivial
  - There is a lot of debate about how to deal with deadlocks in general
  - Four strategies are employed for dealing with deadlocks
    - Ignore the problem – hope they don’t happen or don’t cause major problems if they do
    - Detection and recovery – let deadlocks occur, detect them, then fix the problem
    - Dynamic avoidance – Carefully allocate resources to ensure no deadlocks every occur
    - Prevention – Negate one of the four required conditions to make it impossible for deadlocks to occur
  - Ostrich Algorithm – ignore the problem
    - All detection/recovery/avoidance/prevention techniques will require some processing overhead and loss of general OS efficiency.
    - If deadlocks occur very rarely, but the system must be rebooted often due to other problems, why not ignore it?
      - Rebooting will solve any deadlock, so taking a performance/convenience penalty doesn’t seem like a good tradeoff
    - Not suitable for systems that must be reliable or cannot be rebooted
      - Security systems, nuclear plant controls, etc.
  - Next time: Deadlock detection and recovery