A **race condition** is a situation in which the outcome of a program depends upon timing between entities. Race conditions can lead to program malfunction during critical sections. Locks around critical sections prevent races/damage. Semantics of critical sections are subtle. Atomic actions, including actions on locks and semaphores, disambiguate race conditions.
We have still another issue.
"Too many" locks are as damaging as "too few".
Deadlock

- Other side of spectrum (from race conditions)
- Nothing can continue.
- Common forms
  - Circular dependencies: processes waiting for one another.
  - Resource saturation/starvation: not enough resources for anyone to continue.

Two forms:
Deadlock of threads in a process.
Deadlock of processes in an operating system.
**Deadlock concepts**

**Deadlock schedule:** a sequence of events in time that creates a deadlock.
- Deadlock depends upon **what happens when**.
- It is often a **chance occurrence**.

**Completion schedule:** a sequence of events in time that demonstrates that a situation is **not** a deadlock.
- A situation is **not a deadlock** if there is any schedule of events under which computation can complete.
Basic assumptions

What a process asks for is what it needs.
If a process gets what it asks for, it eventually completes and releases resources.
There are no outside influences.

A deadlock occurs when
processes ask for what they need,
and they won't complete,
without outside intervention (control-C).
Circular dependencies

- Process 1 waits for Process 2
- Process 2 waits for Process 3
- Process 3 waits for Process 1

Circular dependencies
Circular dependency example

Two resources: A and B
Two programs: P and Q

P's schedule:
- Lock A
- Lock B
- Do something
- Release A
- Release B

Q's schedule:
- Lock B
- Lock A
- Do something
- Release B
- Release A

Circular dependency example

Wednesday, October 13, 2004  3:08 PM

Deadlock Page 8
Deadlock Analysis

If "P:Lock A" and "Q:Lock B" happen before "P: Lock B" and "Q: Lock A", then

- P waits forever for "Lock B" (held by Q)
- Q waits forever for "Lock A" (held by P)
If P, Q agree on order, deadlock is impossible.

Real cause of deadlock: legacy code
No agreement on order of locking.
No communication between programmers.
millions of lines of code to fix.
More than two processes
Processes P1, P2, ...
Resources R1, R2, ...

<table>
<thead>
<tr>
<th>P1:</th>
<th>P2:</th>
<th>P3:</th>
<th>P4:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock R1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lock R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lock R3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lock R4</td>
</tr>
<tr>
<td>Lock R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lock R3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lock R4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lock R1</td>
</tr>
</tbody>
</table>

But what if every process asks for all locks at the same time?
Compare the previous schedule with:

<table>
<thead>
<tr>
<th>P1:</th>
<th>P2:</th>
<th>P3:</th>
<th>P4:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock R1, R2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lock R2,R3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lock R3,R4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lock R4,R1</td>
</tr>
</tbody>
</table>

If every process asks for everything it needs in one atomic action, no deadlock is possible.
If every process asks for everything it needs all at once, **deadlock cannot occur.**

It is the process of **deciding what one wants** that causes deadlock.

Reason is that a process **waits for some resources** but **already holds others.**

Example: **reading a big file into memory as a linked list.**
Resource starvation

Special case of circular dependency
Limited resources
Demand exceeds supply
Incremental allocation
Limited number of array elements.

Process 1 requests half, then more than half
Process 2 requests half, then more than half
Result: both processes wait forever
Cause of deadlock: **incremental allocation**

Note that:

The whole problem is that Process 1 and process 2 are concurrently consuming "a little at a time" of the resource.
If they both ask for all they want, atomically, then there's no problem.
But they can't, because they can't predict their future needs!

Suppose $P_1$ needs 6/10
$P_2$ needs 7/10

\[ P_1 \text{ waits } \]

\[ P_1 \text{ needs } 6/10 \]
\[ P_2 \text{ needs } 7/10 \]

\[ P_1 \text{ waits } \]

\[ P_1 \text{ needs } 1 \]
\[ P_1 \text{ ends } \]
\[ P_2 \text{ waits } \]
It is very important to remember that:

When a process awaits resources, it is not ready or runnable.
--> The process cannot correct the problem; it's blocked.
--> Anything we're going to do to correct the problem has to come from outside.

(Livelock is the situation in which processes are running and getting nothing done. We'll consider that Wed.)
Example of resource deadlock:

- Only 6 memory elements available
- Process 1 and 2 interleave requests:
  ```c
  int *a[5]; for (i=0; i<5; i++) a[i]=getmem();
  ```

Example of resource locking
One aid to understanding resource-based deadlock: 

**resource allocation graph**

- A process
- A resource
- A resource with multiple instances

Natural transition:

- Process requests resource
- Resource held by process
- Resource granted to process

During granting of requests.
Circular dependency

Circular dependency=deadlock

![Diagram showing a circular dependency leading to deadlock]
No deadlock

Wednesday, October 13, 2004       6:03 PM

No deadlock
Suppose we have processes p1, p2, p3 competing for resources r1, r2, r3. Suppose that

- p1 holds r1
- p2 holds r2
- p3 holds r3
- p2 requests r1

Is this a deadlock situation?
The opposite of deadlock is a **completion schedule**. Say what happens when. Demonstrate that there are enough resources to do it.

Really important
- If P requests something and doesn't get it, P blocks. Once P is waiting, it is (typically) not in the position to make more requests or to retract them!
For example, consider the previous problem:
   p1 holds r1
   p2 holds r2
   p3 holds r3
   p2 requests r1

One completion schedule is as follows:
   p1 holds what it needs, so it completes.
   p2 is granted r1, so it completes.
   p3 holds what it needs, so it completes.

The existence of a completion schedule is an instantaneous state.
Some facts about completion schedules

Usually more than one.
Only need one.
Depends on what we know now, rather than in the future, when more requests may be made.
Coping with deadlock

○ Prevention through programming practice
○ Prevention through OS features
○ **Passive defenses**: detecting and breaking locks

**Why are most defenses passive?**

**Controversy: Bug or feature?**

○ One opinion: a deadlock is **always** due to a program bug and is the **responsibility of the programmer**.
  - **For**: faster, no overhead.
  - **Against**: wastes resources on buggy programs.
○ Contrary opinion: preventing deadlock is the **job of the operating system**.
  - **For**: more efficient.
  - **Against**: makes arbitrary (uninformed) decisions.
Key to deadlock prevention: atomicity of allocation

- Request resources "all at once"
- Avoid resource allocation "races"
- This is the deep reason behind capability to perform several semaphore operations simultaneously (semop).
Atomic allocation of two resources avoids crossovers.
Limiting resource allocation deadlock

Process P and Q:

```plaintext
request (a,5);  # five units of something
// do something
release (a,5);
```

![Diagram](Image)
OS Deadlock prevention methods
○ Provide atomic multiple-resource locks (semop)
○ Prioritize locks, so that circular dependencies cannot occur.
○ Detect and break deadlocks proactively
○ Analyze requests before granting locks or resources
Detecting deadlock of binary locks

Compute the resource allocation graph for active processes.
Check the graph for cycles.
If there is a cycle in the graph, there is deadlock.
The linux approach: kill one process at random.
Simple cycle detection

start with the resource allocation graph
repeat while possible:
    remove one node with indegree 0, and all of its edges.
end

If there are nodes left over, the graph has a cycle.
In the OS, edges are added to this graph one edge at a time.
If we remember a completion schedule whereby processes are removed,
  adding an edge can only move a process down in the schedule!
  removing an edge can only move a process up.
Detecting deadlock among multi-state locks
This is more of a problem...
Deadlock might be due to a process not involved in a loop:
We need to handle large arrays of different kinds of resources.
We will exploit the idea that small changes in process state lead to small changes in completion schedule.