As a project wears on, standards for success slip lower and lower.

0 Hours

Okay, I should be able to dual-boot BSD soon.

6 Hours

I'll be happy if I can get the system working like it was when I started.

10 Hours

Well, the desktop's a lost cause, but I think I can fix the problems the laptops developed.

24 Hours

If we're lucky, the sharks will stay away until we reach shallow water.

If we make it back alive, you're never upgrading anything again.
Reminders: February 24, 2014

- I hope you had a good Reading Week!
- Don’t forget about Assignment 2, due next week
- Exercise 3 is due this Wednesday
- Any questions about anything?
Comments on Exercise 2

- Confusion on kernel mappings (e.g. many-to-many mappings)

- If the many-to-many model of multithreading suffers from neither of the shortcomings of the many-to-one or one-to-one models why do Linux and Windows implement the one-to-one and not the many-to-many?

- Some confusion on busy waiting versus blocking

```c
// busy wait
// use CPU to loop
while (no_lock);
```

```c
// blocking wait
// process is preempted
acquire (lock);
```
Objectives

- Develop a description of deadlocks, which prevents sets of concurrent processes from completing their tasks
- Present several methods for preventing, avoiding and recovering from deadlocks
What is the deadlock problem?

- Imagine you come to unlock your barn, but the lock is just too rusty...we have a dead-lock

- Just kidding.
What is the deadlock problem?

• It’s not a physical hardware problem, but rather a problem that occurs between processes

• We’ll use traffic as an analogy for process traffic
...And it can be horrible
What is a deadlock?

- **Deadlock** is a permanent blocking of a set of processes that compete for system resources.
- Deadlock occurs when a set of processes are in a wait state, because each process waiting for a resource held by some other waiting process.
- Therefore, all deadlocks involve conflicting resources needed by two or more processes.
Why is dealing with deadlocks hard?

- Problem: there is no (known) efficient solution to the general deadlock problem
- Alternative approaches:
  - ignore deadlocks
  - prevent or avoid deadlocks
  - recover from deadlocks (i.e. allow them to happen and recover)
Classification of resources - I

- **Reusable**: can be used by one process at a time and not depleted by that use
  - e.g. CPU, memory, some I/O devices, files

- **Consumable**: can be created and destroyed; when resource acquired by process, the resource ceases to exist
  - e.g. interrupts, signals or messages
Classification of resources - 2

• **Preemptable**: can be taken away from a process with no ill effects (assuming appropriate save/restore)
  - e.g. CPU or memory

• **Non-preemptable**: cannot be taken away from its current owner without causing the computation to fail
  - e.g. printer or floppy disk

• **Deadlock occurs when sharing reusable and non-preemptable resources**
Conditions that must hold for a deadlock to occur

• **Mutual exclusion**: processes require exclusive control of its resources (not sharing)

• **Hold and wait**: process may wait for a resource while holding other

• **No preemption**: process will not give up a resource until it is finished with it. In addition, processes are irreversible: unable to reuse to an earlier state where resources not held

• **Circular Wait**: each process in the chain holds a resource requested by another
Example of deadlock with four conditions

1. **C1**: Refuses to share intersection (mutual exclusion).
2. **C2**: Holds the intersection (hold and wait).
3. **C3**: Will not give up the intersection (no preemption).
4. **C4**: Circular wait.

- Back-up not possible (no rollback).
Each condition necessary for deadlock but not sufficient on its own

• Negating “mutual exclusion”: systems with only simultaneously shared resources cannot deadlock

• Negating “hold and wait”: systems that abort processes which request a resource that is in use

• Negating “no preemption”: systems that might preempt resource use even if process has not acquired all its needs

• Negating “irreversibility”: transaction system provide checkpoints so process may back out of transaction

• Negating “circular wait”: systems that prevent, detect or avoid cycles (often the preferred solution for deadlocks)
How do we analyze these conditions?

- Formal description using resource allocation graphs

  Set of processes,
  \[ P = \{ P_1, P_2, \ldots, P_n \} \]

  Set of resources,
  \[ R = \{ R_1, R_2, \ldots, R_n \} \]

  Vertex set: \( V = P \cup R \)

  Request edge: \( P_i \rightarrow R_j \)

  Assignment edge: \( R_j \rightarrow P_i \)
Resource allocation graph with a deadlock

![Resource allocation graph with a deadlock diagram](image-url)
Resource allocation graph with a cycle but no deadlock
Video break: brought to you by another amazing classmate!
Useful fact 1: deadlocks and cycles

• There cannot be a deadlock if the graph does not contain cycles.

• If the graph does contain cycles:
  • There is definitely a deadlock if there is only one instance per resource type and there is a cycle.
  • If there are several instances of a resource type, then there is only the possibility of a deadlock.
Cycles with several instances of a resource

- Cycle is a **necessary condition** for a deadlock, but — given multiple units of a resource — is **not sufficient**
  
  - a deadlock implies a cycle, i.e. If deadlock then a cycle must exist

- A **knot** is a **necessary and sufficient condition**
  
  - a deadlock exists if-and-only-if a knot exists

- A **knot** is a set of vertices such that when starting at any vertex in the knot, there is a path to any other vertex in the knot, but not to vertices outside the knot
Knots in a resource allocation graph

Graph with knot

Graph without knot
Exercise: how do we test for knots?

- Graph theory is a difficult and an active area of research
- But, either way, you might be surprised with your ingenuity!
- How would you try to test for a knot?
One technique for detecting a knot

1. Start with the resource allocation graph
2. Remove processes that are not waiting on any resource
3. Remove processes that are waiting only on resources that are not fully allocated
4. Repeat step 3 until no more processes can be removed
5. If the algorithm is able to remove all processes, then no knot is present
Example of detecting a knot

Graph with knot

Graph without knot
How do we deal with deadlocks?

- **Ignore**: pretend there is no problem (let system crash)

- **Prevent**: design the system so that deadlock is impossible
  - e.g. compile-time/statically, by design

- **Avoid**: make decisions dynamically that do not allow requests that could lead to a deadlock
  - run-time/dynamically, before it happens

- **Detect**: let the deadlock occur, detect it when it happens and recover from it after the fact
  - run-time/dynamically, after it happens
Strategy 1: Ignore Deadlocks

- Typical UNIX approach since occasional deadlock preferable to
  - overhead from dealing with deadlocks
  - restrictions on using resources

- Trade-off between correctness and performance — with more research into the problem, maybe the typical strategy will not be to ignore
Strategy 2: Prevent Deadlocks

- Many different ways to prevent deadlocks, from
  - very restrictive but easy to guarantee no deadlock to
  - less restrictive but harder to guarantee no deadlock
- Preferable to minimize restrictions on resource use while still guaranteeing that no deadlocks can occur
Options for preventing deadlocks

Can prevent deadlock by preventing any one of the four necessary conditions

- **Mutual exclusion**: cannot prevent this requirement, since non-sharable resources must have mutually exclusive access
  
  - else, can only use sharable resources like read-only files, which is restrictive

- **Hold-and-Wait**: can prevent deadlock by ensuring process can only request a resource if it does not hold any others OR generally request all required resources at once
  
  - quite restrictive with low resource utilization
Options for preventing deadlocks

• **No preemption**: can prevent deadlock using preemption by
  
  • requiring that if process is denied access to a requested resource, then it must release all resources it currently held
  
  • preempted resources added to list of resources for which process waiting
  
  • process restarted only when can regain all resources it requested

• **Circular Wait**: can prevent deadlock by imposing a total ordering on all resource types and requiring that each process request resources in increasing order
  
  • creates a directed acyclic graph (no cycles!)