COS 318: Operating Systems

Deadlocks

Prof. Margaret Martonosi
Computer Science Department
Princeton University

http://www.cs.princeton.edu/courses/archive/fall11/cos318/

Announcements

- Project 2 due: Weds Oct 19
- Midterm Thursday Oct 27
  - Sample on webpage…
- Facebook TechTalk
  - The HipHop Virtual Machine
  - Guilherme Ottoni *08
  - Today at 5:30pm IN THIS ROOM!
- From last time:
  - signal vs. broadcast
  - Java: notify vs. notifyAll

Dennis Ritchie: 1941-2011

- With Bell Labs' Ken Thompson, Ritchie helped develop Unix, running on a DEC PDP-11, and released the first edition of the operating system in 1971.
- Two years later, Ritchie came up with the C language, building on B. C offered the concise syntax, functionality and detail features necessary to make the language work for programming an operating system. Most of Unix's components were re-written in C, with the kernel published the same year.
- Received the 1983 Turing Award and a 1997 US National Medal of Technology
  - both with Thompson for his work on C and Unix

Today's Topic: Deadlock…

- Conditions for a deadlock
- Strategies to deal with deadlocks

Background Definitions

- Use processes and threads interchangeably
- Resources
  - Preemptable: CPU (can be taken away)
  - Non-preemptable: Disk, files, mutex, ... (can't be taken away)
- Use a resource
  - Request, Use, Release
- Starvation
  - A process waits indefinitely

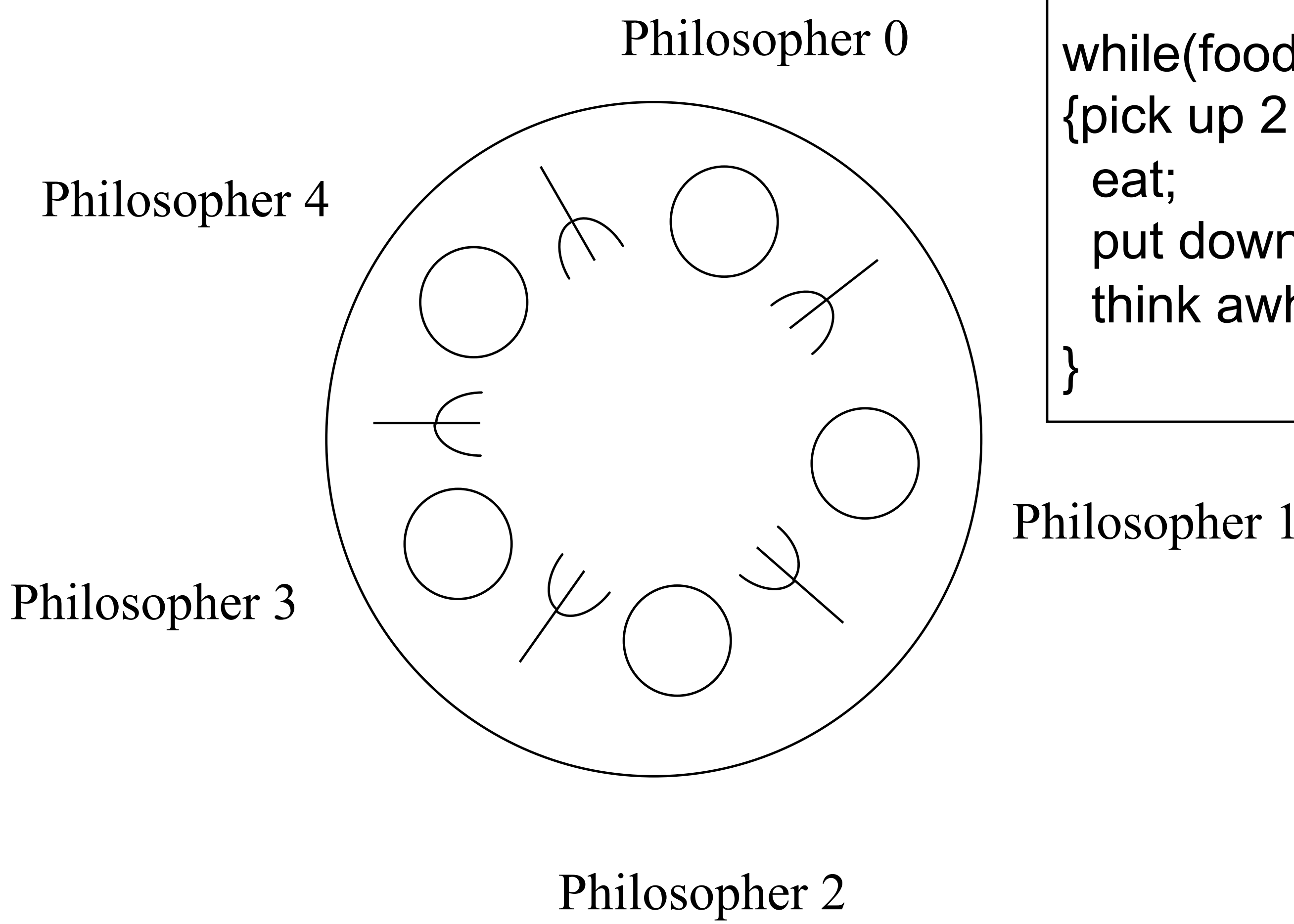
Deadlock

- A set of processes have a deadlock if each process is waiting for an event that only another process in the set can cause

Conditions for Deadlock

- Mutual exclusion condition
  - Each resource is assigned to exactly one process
- Hold and Wait
  - Processes holding resources can request new resources
- No preemption
  - Resources cannot be taken away
- Circular chain of requests
  - One process waits for another in a circular fashion

5 Dining Philosophers




```
while(food available)
{pick up 2 adj. forks;
  eat;
  put down forks;
  think awhile;
}
```

Template for Philosopher

```
while (food available)
{                                              /*pick up forks*/

    eat;
                                               /*put down forks*/

    think awhile;
}
```

Naive Solution

```
while (food available)
{
    P(fork[left(me)]);
    P(fork[right(me)]);
  eat;
    V(fork[left(me)]);
    V(fork[right(me)]);
  think awhile;
}
```

/*pick up forks*/

/*put down forks*/

Does this work?

Simplest Example of Deadlock

Thread 0	Interleaving	Thread 1
P(R1)	P(R1)	P(R2)
P(R2)	P(R2)	P(R1)
V(R1)	P(R1) waits	V(R2)
V(R2)	P(R2) waits	V(R1)

R1 and R2 initially 1 (binary semaphore)

Conditions for Deadlock

- Mutually exclusive use of resources
  - Binary semaphores R1 and R2
- Hold and wait
  - Holding either R1 or R2 while waiting on other
- No pre-emption
  - Neither R1 nor R2 are removed from their respective holding Threads.
- Circular waiting
  - Thread 0 waits for Thread 1 to V(R2) and
    Thread 1 waits for Thread 0 to V(R1)

Dealing with Deadlock

It can be ***prevented*** by breaking one of the prerequisite conditions:

- Mutually exclusive use of resources
  - Example: Allowing shared access to read-only files (readers/writers problem)
- circular waiting
  - Example: Define an ***ordering*** on resources and acquire them in order
- hold and wait
- no pre-emption

Circular Wait Condition

```
while (food available)
{  if (me == 0) {P(fork[left(me)]); P(fork[right(me)]);}
   else {(P(fork[right(me)]); P(fork[left(me)]); }
   eat;
   V(fork[left(me)]); V(fork[right(me)]);

   think awhile;
}
```

Hold and Wait Condition

```
while (food available)
{   P(mutex);
    while (forks [me] != 2)
        {blocking[me] = true; V(mutex); P(sleepy[me]); P(mutex);}
    forks [leftneighbor(me)] --;  forks [rightneighbor(me)]--;
    V(mutex):
    eat;
    P(mutex);
    forks [leftneighbor(me)] ++;  forks [rightneighbor(me)]++;
    if (blocking[leftneighbor(me)]) {
        blocking [leftneighbor(me)] = false; V(sleepy[leftneighbor(me)]); }
    if (blocking[rightneighbor(me)]) {
        blocking[rightneighbor(me)] = false; V(sleepy[rightneighbor(me)]); }
    V(mutex);
    think awhile;
}
```

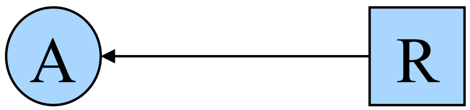
Starvation

The difference between deadlock and starvation is subtle:

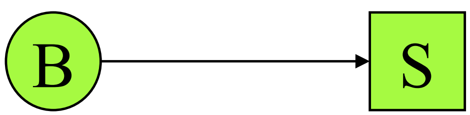
- Once a set of processes are deadlocked, there is no future execution sequence that can get them out of it.
- In starvation, there does exist some execution sequence that is favorable to the starving process although there is no guarantee it will ever occur.
- Rollback and Retry solutions are prone to starvation.
- Continuous arrival of higher priority processes is another common starvation situation.

Resource Allocation Graph

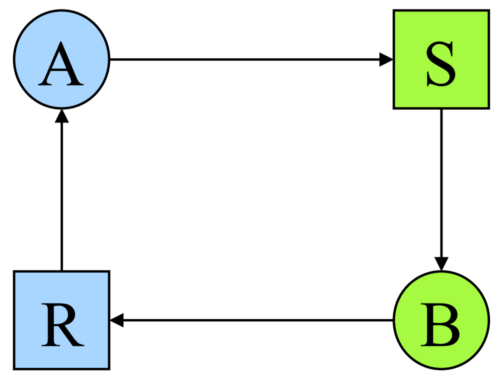
- Process A is holding resource R



- Process B requests resource S



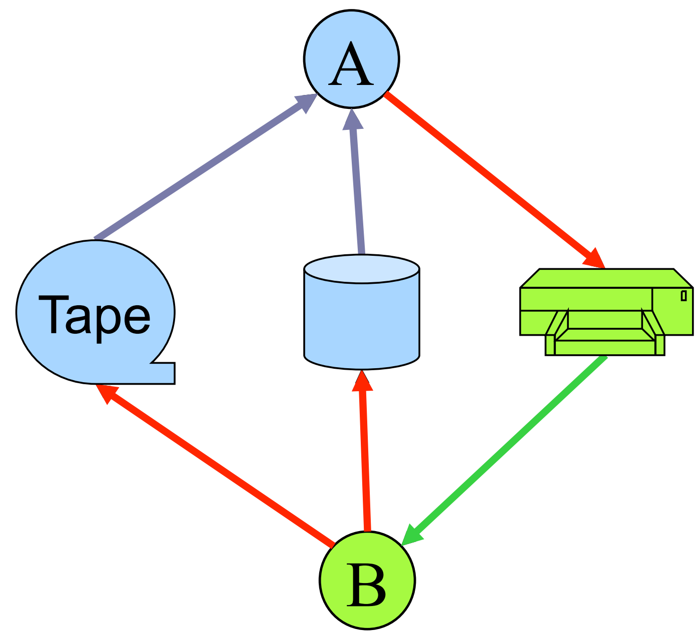
- A cycle in resource allocation graph ⇒ deadlock
- If A requests for S while holding R, and B requests for R while holding S, then



How do you deal with multiple instances of a resource?

An Example

- A utility program
  - Copy a file from tape to disk
  - Print the file to printer
- Resources
  - Tape
  - Disk
  - Printer
- A deadlock
  - **A** holds tape and disk, then requests for a printer
  - **B** holds printer, then requests for tape and disk



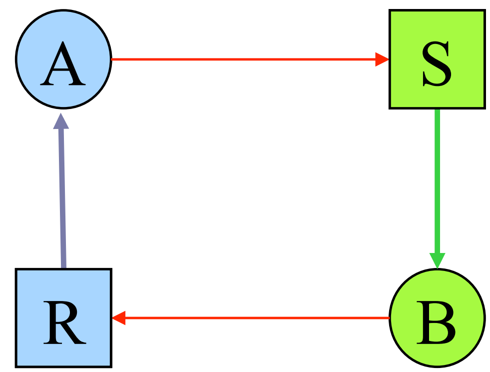
Conditions for Deadlock

- Mutual exclusion condition
  - Each resource is assigned to exactly one process
- Hold and Wait
  - Processes holding resources can request new resources
- No preemption
  - Resources cannot be taken away
- Circular chain of requests
  - One process waits for another in a circular fashion

- Question
  - Are all conditions necessary?

Eliminate Competition for Resources?

- If running A to completion and then running B, there will be no deadlock
- Generalize this idea for all processes?
- Is it a good idea to develop a CPU scheduling algorithm that causes no deadlock?



Previous example

Strategies

- Ignore the problem
  - It is user's fault
- Detection and recovery
  - Fix the problem afterwards
- Dynamic avoidance
  - Careful allocation
- Prevention
  - Negate one of the four conditions

Ignore the Problem

- The OS kernel locks up
  - Reboot
- Device driver locks up
  - Remove the device
  - Restart
- An application hangs (“not responding”)
  - Kill the application and restart
  - Familiar with this?
- An application ran for a while and then hang
  - Checkpoint the application
  - Change the environment (reboot OS)
  - Restart from the previous checkpoint

Detection and Recovery

- Detection
  - Scan resource graph
  - Detect cycles
- Recovery (difficult)
  - Kill process/threads (can you always do this?)
  - Roll back actions of deadlocked threads
- What about the tape-disk-printer example?

Avoidance

- Safety Condition:
  - It is not deadlocked
  - There is some scheduling order in which every process can run to completion (even if all request their max resources)
- Banker's algorithm (Dijkstra 65)
  - Single resource
    - Each process has a credit
    - Total resources may not satisfy all credits
    - Track resources assigned and needed
    - Check on each allocation for safety
  - Multiple resources
    - Two matrices: allocated and needed
    - See textbook for details

Examples (Single Resource)

Total: 8

	Has	Max
P_1	2	6
P_2	2	3
P_3	3	5

Free: 1

	Has	Max
P_1	2	6
P_2	**3**	3
P_3	3	5

Free: **0**

	Has	Max
P_1	2	6
P_2	**0**	**0**
P_3	3	5

Free: **3**

	Has	Max
P_1	2	6
P_2	0	0
P_3	**5**	5

Free: **1**

	Has	Max
P_1	2	6
P_2	0	0
P_3	**0**	**0**

Free: **6**

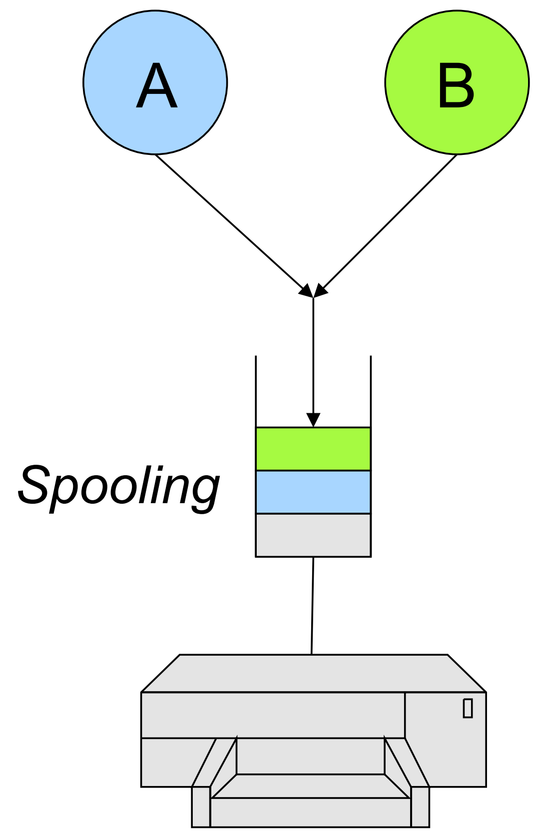
	Has	Max
P_1	4	6
P_2	1	3
P_3	2	5

?

Free: 1

Prevention: Avoid Mutual Exclusion

- Some resources are not physically sharable
  - Printer, tape, etc
- Some can be made sharable
  - Read-only files, memory, etc
  - Read/write locks
- Some can be virtualized by spooling
  - Use storage to virtualize a resource into multiple resources
  - Use a queue to schedule
  - Does this apply to all resources?
- What about the tape-disk-printer example?



Prevention: Avoid Hold and Wait

- Two-phase locking

  Phase I:

  - Try to lock all resources at the beginning

  Phase II:

  - If successful, use the resources and release them
  - Otherwise, release all resources and start over
- Application
  - Telephone company's circuit switching
- What about the tape-disk-printer example?

Prevention: No Preemption

- Make the scheduler be aware of resource allocation
- Method
  - If the system cannot satisfy a request from a process holding resources, preempt the process and release all resources
  - Schedule it only if the system satisfies all resources
- Alternative
  - Preempt the process holding the requested resource
- What about the tape-disk-printer example?

Prevention: No Circular Wait

- Impose an order of requests for all resources
- Method
  - Assign a unique id to each resource
  - All requests must be in an ascending order of the ids
- A variation
  - Assign a unique id to each resource
  - No process requests a resource lower than what it is holding
- What about the tape-disk-printer example?
- Can we prove that this method has no circular wait?

Which Is Your Favorite?

- Ignore the problem
  - It is user's fault
- Detection and recovery
  - Fix the problem afterwards
- Dynamic avoidance
  - Careful allocation
- Prevention (Negate one of the four conditions)
  - Avoid mutual exclusion
  - Avoid hold and wait
  - No preemption
  - No circular wait

Tradeoffs and Applications

- Ignore the problem for applications
  - It is application developers' job to deal with their deadlocks
  - OS provides mechanisms to break applications' deadlocks
- Kernel should not have any deadlocks
  - Use prevention methods
  - Most popular is to apply no-circular-wait principle everywhere

Break + Deadlock-related Story Time

- [The Zax]

OpenLDAP deadlock, bug #3494

```
{
 lock(A)
 ...
 lock(B)
 ...
 unlock(A)
 ...
 if ( cursize > maxsize) {
  ...
   for (...)
    ...
     lock(A)
     ...
     unlock(A)
     ...
   }
 }
 ....
 unlock(B)
}
```

OpenLDAP deadlock, fix #1

```
{
 lock(A)
 ...
 lock(B)
 ...
 unlock(A)
 ...
 if ( cursize > maxsize) {
  ...
   for (...)
    ...
     lock(A)
     ...
     unlock(A)
     ...
   }
 }
 ....
 unlock(B)
}
```

```
{
 lock(A)
 ...
 lock(B)
 ...
 unlock(A)
 ...
 if ( cursize > maxsize) {
  ...
   for (...)
    ...
     if ( ! try_lock(A)) break;
     ...
     unlock(A)
     ...
   }
 }
 ....
 unlock(B)
}
```

Changes the algorithm, but maybe that's OK

OpenLDAP deadlock, fix #2

```
{
  lock(A)
  ...
  lock(B)
  ...
  unlock(A)
  ...
  if ( cursize > maxsize) {
   ...
     for (...)
      ...
       lock(A)
       ...
       unlock(A)
       ...
     }
  }
  ....
  unlock(B)
}
```

```
{
  lock(A)
  ...
  lock(B)
  ...
  ...
  if ( cursize > maxsize) {
   ...
     for (...)
      ...

      ...

       ...
     }
  }
  unlock(A)
  ....
  unlock(B)
}
```

Conditions for Deadlock

- Mutual exclusion condition
  - Each resource is assigned to exactly one process
- Hold and Wait
  - Processes holding resources can request new resources
- No preemption
  - Resources cannot be taken away
- Circular chain of requests
  - One process waits for another in a circular fashion

Apache bug #42031

http://issues.apache.org/bugzilla/show_bug.cgi?id=42031
Summary: EventMPM child process freeze
Product: Apache httpd-2 Version: 2.3-HEAD
Platform: PC
OS/Version: Linux
Status: NEW
Severity: critical
Priority: P2
Component: Event MPM
AssignedTo: bugs@httpd.apache.org
ReportedBy: serai@lans-tv.com

Child process freezes with many downloading against MaxClients.

How to reproduce:

(1) configuration to httpd.conf StartServers 1 MaxClients 3 MinSpareThreads 1 MaxSpareThreads 3 ThreadsPerChild 3 MaxRequestsPerChild 0 Timeout 10 KeepAlive On MaxKeepAliveRequests 0 KeepAliveTimeout 5

(2) put a large file "test.mpg" (about 200MB) on DocumentRoot

(3) apachectl start

(4) execute many downloading simultaneously. e.g. bash and wget:

```
$ for (( i=0 ; i<20 ; i++ )); do wget -b http://localhost/test.mpg; done;
```

Then the child process often freezes. If not, try to download more.

(5) terminate downloading e.g. bash and wget: $ killall wget

(6) access to any file from web browser. However long you wait, server won't response.

Apache deadlock, bug #42031

```
listener_thread(...) {
  lock(timeout)
  ...
  lock(idlers)
  ...
  cond_wait (wait_for_idler, idlers)
  ...
  unlock(idlers)
  ...
  unlock(timeout)
}
```

```
worker_thread(...) {
  lock(timeout)
  ...
  unlock(timeout)
  ...
  lock (idlers)
  ...
  signal (wait_for_idler)
  ...
  unlock(idler)
  ...
}
```

Conditions for Deadlock

- Mutual exclusion condition
  - Each resource is assigned to exactly one process
- Hold and Wait
  - Processes holding resources can request new resources
- No preemption
  - Resources cannot be taken away
- Circular chain of requests
  - One process waits for another in a circular fashion

Summary

- Deadlock conditions
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular chain of requests
- Strategies to deal with deadlocks
  - Simpler ways are to negate one of the four conditions