

# Introduction to Operating Systems

CPSC/ECE 3220 Summer 2018

Lecture Notes  
OSPP Chapter 6 – Part B

(adapted by Mark Smotherman from Tom Anderson's slides on OSPP web site)

# Deadlock Definition

- Resource: a physical or virtual entity that can be assigned to a user or application
  - Anything needed by a thread to do its job (CPU, disk space, memory, lock)
  - Preemptable: can be taken away by OS
  - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
  - Deadlock  $\Rightarrow$  starvation, but not vice versa

# Example: two locks

Thread A

```
lock1.acquire();  
lock2.acquire();
```

```
// critical section
```

```
lock2.release();  
lock1.release();
```

Thread B

```
lock2.acquire();  
lock1.acquire();
```

```
// critical section
```

```
lock1.release();  
lock2.release();
```

# Bidirectional Bounded Buffer

Thread A

```
buffer1.put(data);  
buffer1.put(data);  
  
buffer2.get();  
buffer2.get();
```

Thread B

```
buffer2.put(data);  
buffer2.put(data);  
  
buffer1.get();  
buffer1.get();
```

Suppose buffer1 and buffer2 both start almost full.

# Two locks and a condition variable

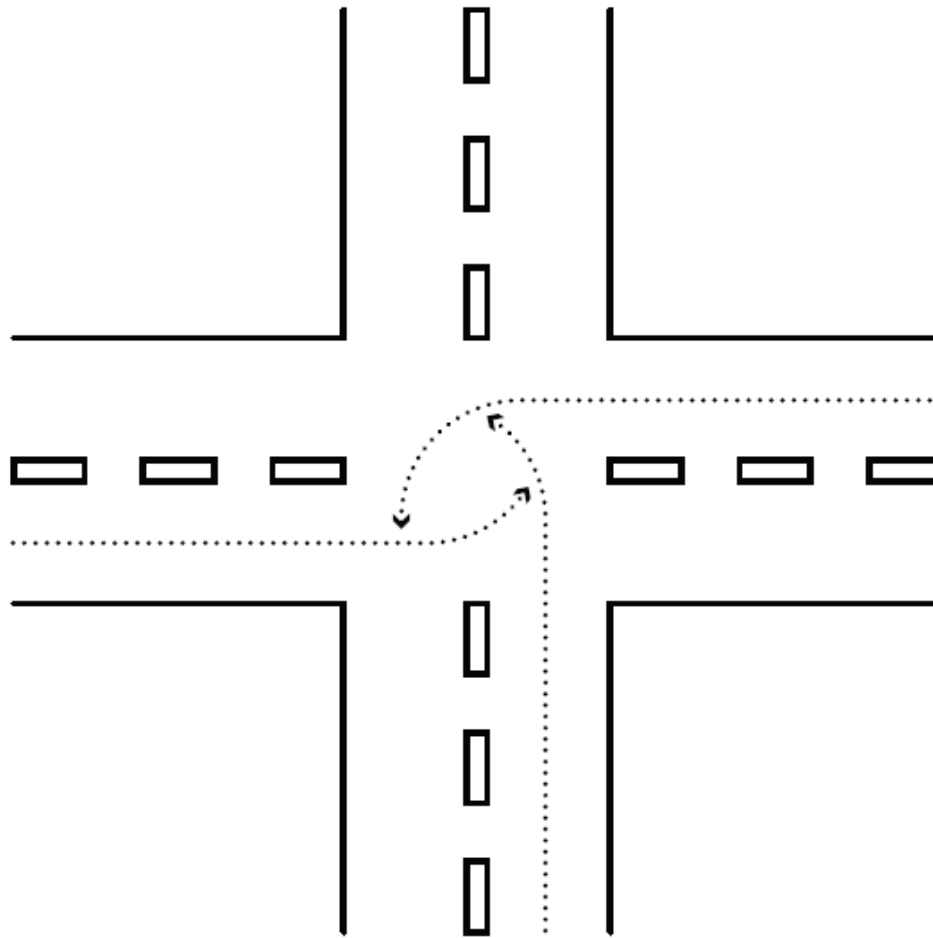
Thread A

```
lock1.acquire();  
...  
lock2.acquire();  
while (need to wait) {  
    condition.wait(lock2);  
}  
lock2.release();  
...  
lock1.release();
```

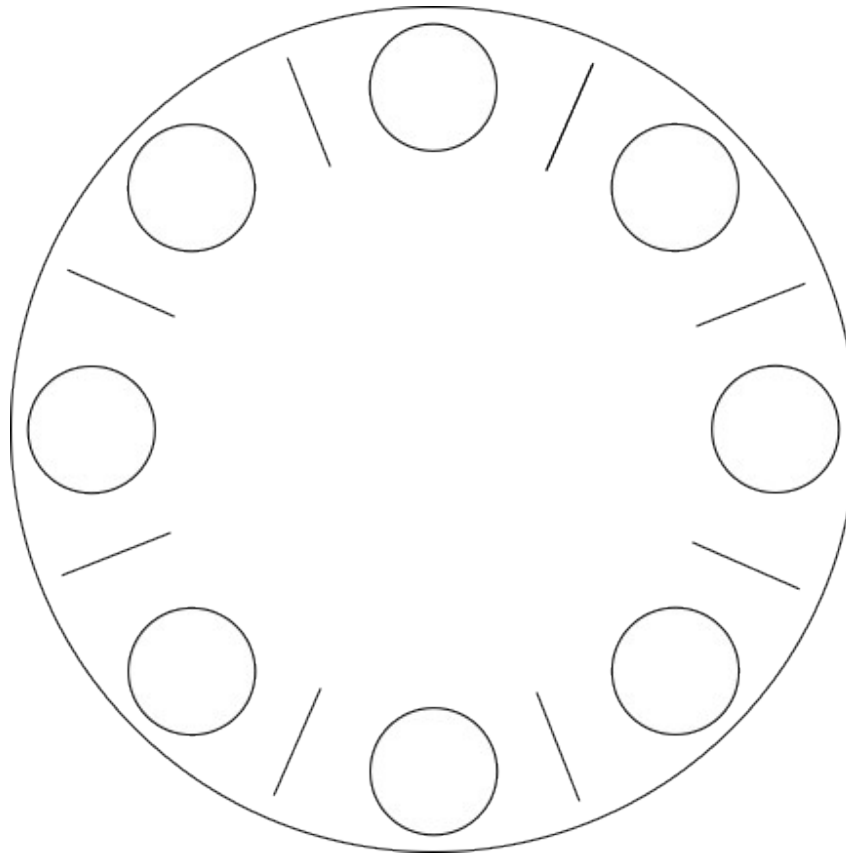
Thread B

```
lock1.acquire();  
...  
lock2.acquire();  
...  
condition.signal(lock2);  
...  
lock2.release();  
...  
lock1.release();
```

# Yet another Example



# Dining Philosophers



Each philosopher needs two chopsticks to eat.  
Each grabs chopstick on the right first.

# Necessary Conditions for Deadlock

- Limited access to resources
  - If infinite resources, no deadlock!
- No preemption
  - If resources are virtual, can break deadlock
- Multiple independent requests
  - “wait while holding”
- Circular chain of requests



# Question

- How does Dining Philosophers meet the necessary conditions for deadlock?
  - Limited access to resources
  - No preemption
  - Multiple independent requests (wait while holding)
  - Circular chain of requests
- How can we modify Dining Philosophers to prevent deadlock?

# Approaches to Handling Deadlock

- Prevent by limiting program behavior
  - Limit program from doing anything that might lead to deadlock
- Avoid by predicting the future
  - If we know what program will do, we can tell if granting a resource might lead to deadlock
- Detect and recover
  - If we can rollback a thread, we can fix a deadlock once it occurs

# Exploit or Limit Behavior

- Provide enough resources
  - How many chopsticks are enough?
- Eliminate wait while holding
  - Release lock when calling out of module
  - Telephone circuit setup
- Eliminate circular waiting
  - Lock ordering: always acquire locks in a fixed order
  - Example: move file from one directory to another

# Example

## Thread 1

1. Acquire A
- 2.
3. Acquire C
- 4.
5. If (maybe) Wait for B

## Thread 2

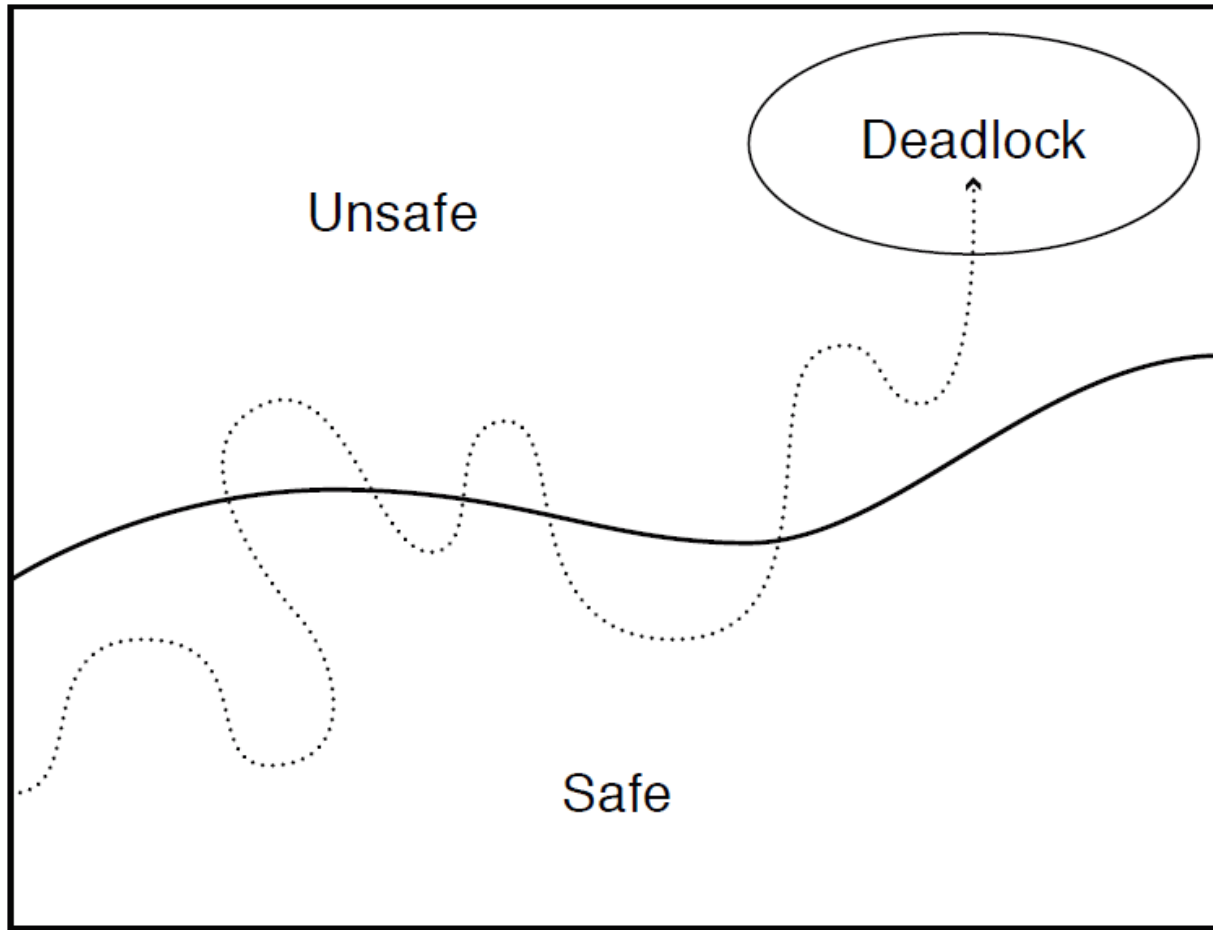
- 1.
2. Acquire B
- 3.
4. Wait for A

How can we make sure to avoid deadlock?

# Deadlock Dynamics

- Safe state:
  - For any possible sequence of future resource requests, it is possible to eventually grant all requests
  - May require waiting even when resources are available!
- Unsafe state:
  - Some sequence of resource requests can result in deadlock
- Doomed state:
  - All possible computations lead to deadlock

# Possible System States



# Predict the Future

- Banker's algorithm (Dijkstra)
  - State maximum resource needs in advance
  - Allocate resources dynamically when resource is needed -- wait if granting request would lead to deadlock
  - Request can be granted if some sequential ordering of threads is deadlock free
  - Extends to multiple resource types

# Banker's Algorithm

- Grant request iff result is a safe state
- Sum of maximum resource needs of current threads can be greater than the total resources
  - Provided there is some way for all the threads to finish without getting into deadlock
- Example: proceed iff
  - $\text{total available resources} - \# \text{ allocated} \geq \text{max remaining that might be needed by this thread in order to finish}$
  - Guarantees this thread can finish



# Banker's Algorithm Example (1a)

example using total units = 10

<u>thread</u> <u>finish?</u>	<u>max need</u>	<u>allocated</u>	<u>remaining need</u>	<u>able to</u>
A	8	2	6	?
B	6	3	3	?
C	4	2	2	?

available units = 3

# Banker's Algorithm Example (1b)

example using total units = 10

<u>thread</u> <u>finish?</u>	<u>max need</u>		<u>allocated</u>	<u>remaining need</u>	<u>able to</u>
A	8	2	6	?	
B	6	3	3	?	
C	4	2	2	?	

available units = 3

Show that this is a safe state since there is a sequence of thread executions that allows each thread to obtain its maximum resource need, complete its work, and release its resources:

<u>thread</u>	<u>allocation in steps</u>
A	2
B	3
<u>C</u>	<u>2</u>
allocated	7
available	3

# Banker's Algorithm Example (1c)

example using total units = 10

<u>thread</u> <u>finish?</u>	<u>max need</u>		<u>allocated</u>		<u>remaining need</u>	<u>able to</u>
A	8	2	6		?	
B	6	3	3		?	
C	4	2	2		true	

available units = 3

Show that this is a safe state since there is a sequence of thread executions that allows each thread to obtain its maximum resource need, complete its work, and release its resources:

<u>thread</u>	<u>allocation in steps</u>	
A	2	2
B	3	3
C	2 grant 2=>4	
allocated	7	9
available	3	1

# Banker's Algorithm Example (1d)

example using total units = 10

<u>thread</u> <u>finish?</u>	<u>max need</u>		<u>allocated</u>	<u>remaining need</u>	<u>able to</u>
A	8	2	6	?	
B	6	3	3	?	
C	4	2	2	?	

available units = 3

Show that this is a safe state since there is a sequence of thread executions that allows each thread to obtain its maximum resource need, complete its work, and release its resources:

<u>thread</u>	<u>allocation in steps</u>		
A	2	2	2
B	3	3	3
C	2 grant 2=>4 release=>0		
allocated	7	9	5
available	3	1	5

# Banker's Algorithm Example (1e)

example using total units = 10

<u>thread</u> <u>finish?</u>	<u>max need</u>		<u>allocated</u>	<u>remaining need</u>	<u>able to</u>
A	8	2	6	?	
B	6	3	3	true	
C	4	2	2	true	

available units = 3

Show that this is a safe state since there is a sequence of thread executions that allows each thread to obtain its maximum resource need, complete its work, and release its resources:

<u>thread</u>	<u>allocation in steps</u>				
A	2	2	2	2	2
B	3	3	3	grant 3 => 6	release => 0
C	2	grant 2 => 4	release => 0	0	0
allocated	7	9	5	8	2
available	3	1	5	2	8

# Banker's Algorithm Example (1f)

example using total units = 10

<u>thread</u> <u>finish?</u>	<u>max need</u>		<u>allocated</u>		<u>remaining need</u>	<u>able to</u>
A	8	2	6		true	
B	6	3	3		true	
C	4	2	2		true	

available units = 3

Show that this is a safe state since there is a sequence of thread executions that allows each thread to obtain its maximum resource need, complete its work, and release its resources:

<u>thread</u>	<u>allocation in steps</u>							
A	2	2	2	2	2	grant 6=>8 release=>0		
B	3	3	3	grant 3=>6 release=>0		0	0	
C	2	grant 2=>4 release=>0		0	0	0	0	
allocated	7	9	5	8	2	8	0	
available	3	1	5	2	8	2	10	

# Banker's Algorithm Example (1g)

example using total units = 10

<u>thread</u> <u>finish?</u>	<u>max need</u>		<u>allocated</u>		<u>remaining need</u>	<u>able to</u>
A	8	2	6		?	
B	6	3	3		?	
C	4	2	2		?	

available units = 3

This is a safe state since there is a sequence of thread executions that allows each thread to obtain its maximum resource need, complete its work, and release its resources:

<u>thread</u>	<u>allocation in steps</u>
A	2
B	3
<u>C</u>	<u>2</u>
allocated	7
available	3

Can you show a second sequence that leads to the recovery of all 10 resources?

# Banker's Algorithm Example (2)

example using total units = 10

<u>thread</u>	<u>max need</u>	<u>allocated</u>	<u>remaining need</u>
A	8	2	6
B	6	3	3
C	4	2	2

available units = 3

Initial state

thread A requests 2 units (of the three unallocated)

<u>thread</u>	<u>max need</u>	<u>allocated</u>	<u>remaining need</u>
A	8	4	false
B	6	3	false
C	4	2	false

available units = 1

Unsafe state  
if request is  
granted

Cannot grant this request since there would not be enough unallocated units to satisfy the remaining need for any thread!



# Banker's Algorithm Example (3)

Example using total units = 10

<u>thread</u>	<u>max need</u>	<u>allocated</u>	<u>remaining need</u>
A	8	2	6
B	6	3	3
C	4	2	2

available units = 3

Initial state

thread C requests 1 units (of the three unallocated)

<u>thread</u>	<u>max need</u>	<u>allocated</u>	<u>remaining need</u>
A	8	2	6
B	6	3	3
C	4	<b>3</b>	<b>1</b>

available units = **2**

Safe state if  
request is  
granted

Can grant this request!

# Banker's Algorithm Example (4)

Algorithm extends to vectors of resources.

	max need			allocated			remaining			available		
	<u>R1</u>	<u>R2</u>	<u>R3</u>	<u>R1</u>	<u>R2</u>	<u>R3</u>	<u>R1</u>	<u>R2</u>	<u>R3</u>	<u>R1</u>	<u>R2</u>	<u>R3</u>
A	4	1	2	3	0	0	1	1	2	5	1	1
B	7	4	4	2	4	3	5	0	1			
C	3	3	3	0	2	1	3	1	2			

This is a safe state since the unused units (5,1,1) can satisfy B's remaining claim of (5,0,1); when B ends it will release its resources and thus increase the unused units to (7,5,4). This can satisfy both A and C's remaining need.

# Detect and Repair

- Algorithm
  - Scan wait for graph
  - Detect cycles
  - Fix cycles
- Proceed without the resource
  - Requires robust exception handling code
- Roll back and retry
  - Transaction: all operations are provisional until have all required resources to complete operation

# Detecting Deadlock

