



University of
Pittsburgh

Introduction to Operating Systems CS 1550



Spring 2023
Sherif Khattab
ksm73@pitt.edu

(Some slides are from **Silberschatz, Galvin and Gagne ©2013**)

Announcements

- Upcoming deadlines
 - Homework 4 is due **this Friday**
 - Project 1 is due on Friday 2/17 at 11:59 pm
 - Lab 2 is due on Tuesday 2/28 at 11:59 pm

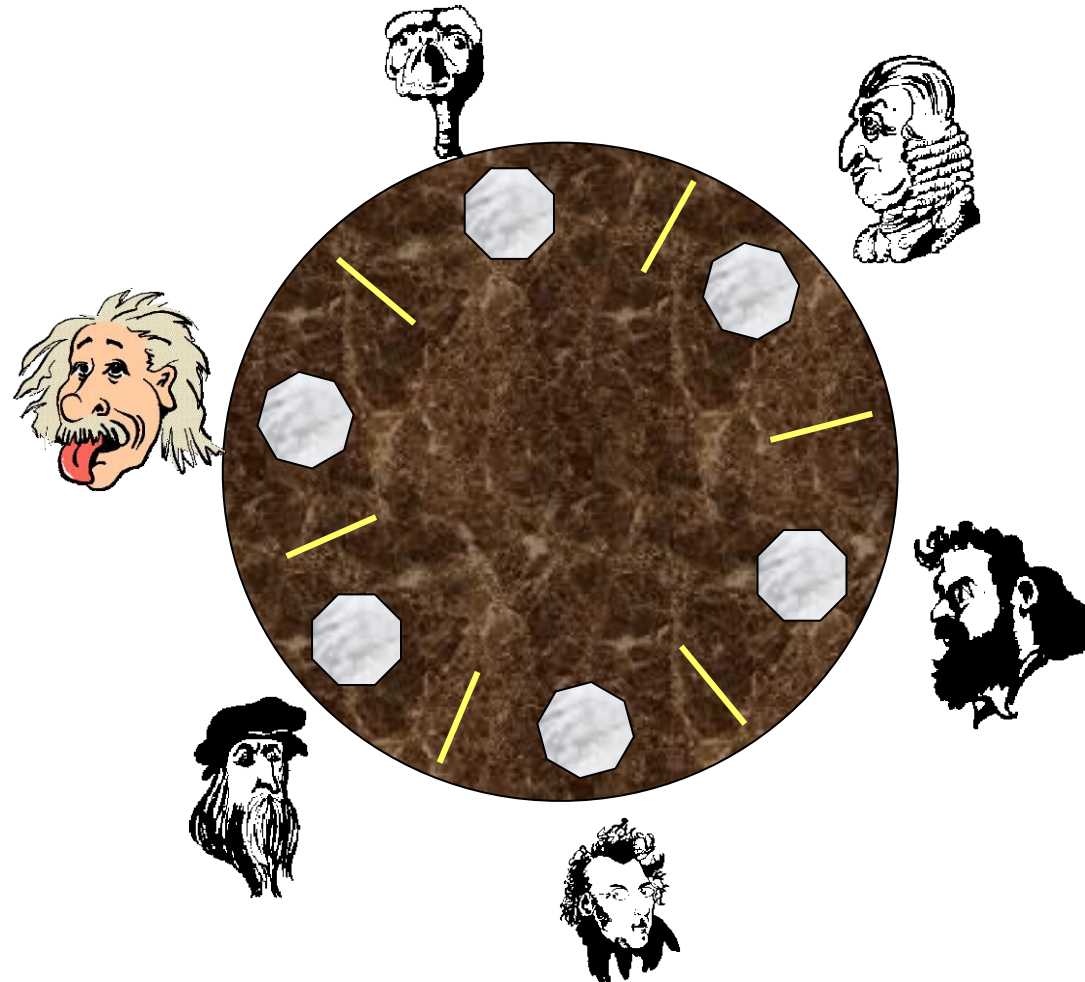
Previous lecture ...

- Readers-Writers problem
 - Solution using Semaphores
 - Solution using Condition Variables

Problem of the Day

Dining Philosophers

- N philosophers around a table
 - All are hungry
 - All like to think
- N chopsticks available
 - 1 between each pair of philosophers
- Philosophers need two chopsticks to eat
- Philosophers alternate between eating and thinking
- Goal: coordinate use of chopsticks



Dining Philosophers: solution 1

- Use a semaphore for each chopstick
- A hungry philosopher
 - Gets the chopstick to his left
 - Gets the chopstick to his right
 - Eats
 - Puts down the chopsticks
- Potential problems?
 - Deadlock
 - Fairness

Shared variables

```
const int n;  
// initialize to 1  
Semaphore chopstick[n];
```

Code for philosopher *i*

```
while(1) {  
    chopstick[i].down();  
    chopstick[(i+1)%n].down();  
    // eat  
    chopstick[i].up();  
    chopstick[(i+1)%n].up();  
    // think  
}
```

Tracing: Sequence 1

- P0 picks left
- P0 picks right
- P3 picks left
- P3 picks right
- P3 eats
- P0 eats
- P3 puts down
- P0 puts down

Shared variables

```
const int n;  
// initialize to 1  
Semaphore chopstick[n];
```

Code for philosopher *i*

```
while(1) {  
    chopstick[i].down();  
    chopstick[(i+1)%n].down();  
    // eat  
    chopstick[i].up();  
    chopstick[(i+1)%n].up();  
    // think  
}
```

Tracing: Sequence 2

- for($i=0$; $i<6$; $i++$)
 - P_i picks left
- P3 eats
- P0 eats
- P3 puts down
- P0 puts down

Shared variables

```
const int n;  
// initialize to 1  
Semaphore chopstick[n];
```

Code for philosopher i

```
while(1) {  
    chopstick[i].down();  
    chopstick[(i+1)%n].down();  
    // eat  
    chopstick[i].up();  
    chopstick[(i+1)%n].up();  
    // think  
}
```

What is a deadlock?

- Formal 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.”
- Usually, the event is release of a currently held resource
- In deadlock, none of the processes can
 - Run
 - Release resources
 - Be awakened

How to solve the Deadlock problem?

- Ignore the problem
- Detect and react
- Prevent (intervene at design-time)
- Avoid (intervene at run-time)

The Ostrich Algorithm

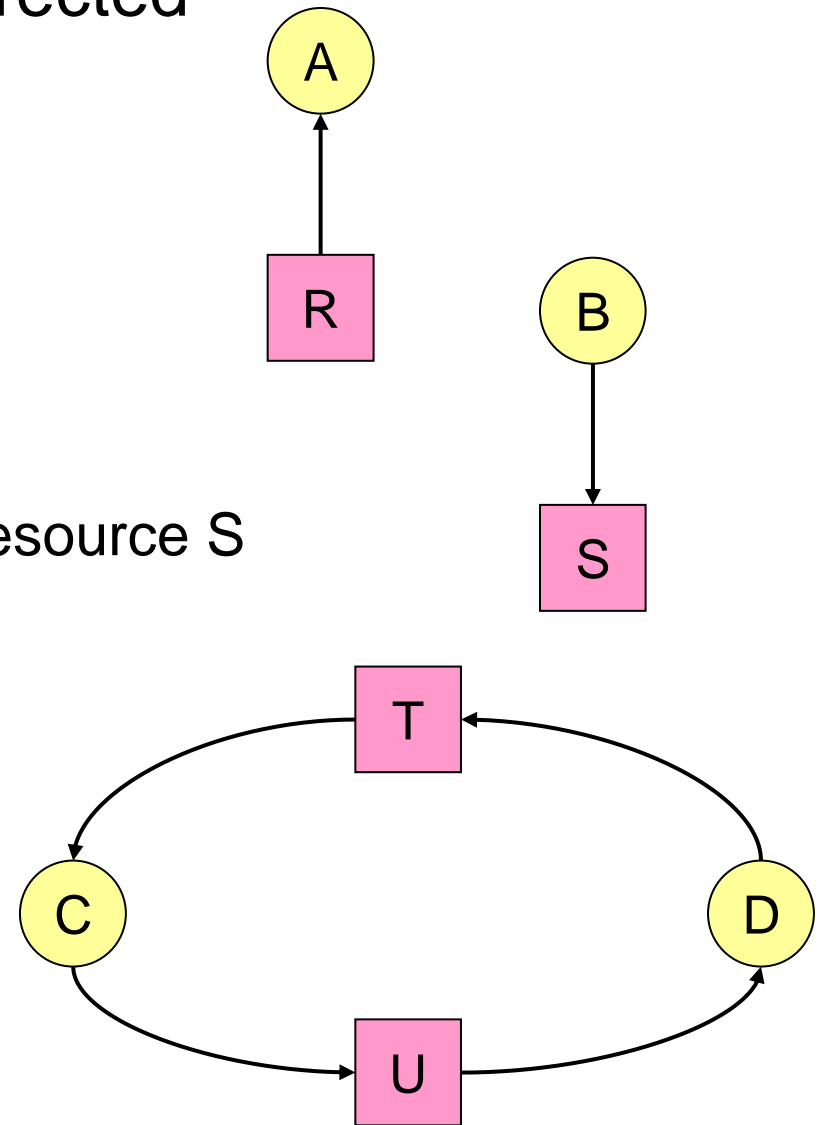
- Pretend there's no problem
- Reasonable if
 - Deadlocks occur very rarely
 - Cost of prevention is high
- UNIX and Windows take this approach
 - Resources (memory, CPU, disk space) are plentiful
 - Deadlocks over such resources rarely occur
 - Deadlocks typically handled by rebooting
- Trade off between convenience and correctness

Deadlock Detection

How can the OS detect a deadlock?

Resource allocation graphs

- Resource allocation modeled by directed graphs
- Example 1:
 - Resource R assigned to process A
- Example 2:
 - Process B is requesting / waiting for resource S
- Example 3:
 - Process C holds T, waiting for U
 - Process D holds U, waiting for T
 - C and D are in deadlock!



Deadlock Prevention

How an application/system designer **prevent** deadlocks?

Dining Philosophers: solution 2

- Use a semaphore for each chopstick
- A hungry philosopher
 - Gets lower, then higher numbered chopstick
 - Eats
 - Puts down the chopsticks
- Potential problems?
 - Deadlock
 - Fairness

Shared variables

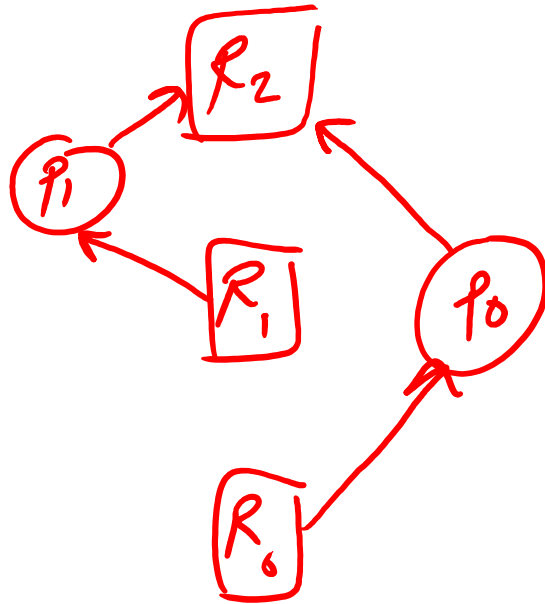
```
const int n;  
// initialize to 1  
Semaphore chopstick[n];
```

Code for philosopher i

```
int i1,i2;  
while(1) {  
    if (i != (n-1)) {  
        i1 = i;  
        i2 = i+1;  
    } else {  
        i1 = 0;  
        i2 = n-1;  
    }  
    chopstick[i1].down();  
    chopstick[i2].down();  
    // eat  
    chopstick[i1].up();  
    chopstick[i2].up();  
    // think  
}
```

Proof sketch for Deadlock Prevention

- If resources are ordered and resource requests within each process follow the resource ordering, the resource allocation graph will have no downward arrows.



no $\downarrow \Rightarrow$ no cycles

P_0 holds R_0
Waits R_2

P_1 holds R_1
Waits R_2

Deadlock Avoidance

How can the OS intervene at run-time to avoid deadlocks?

Banker's Algorithm

We can use the same algorithm for avoiding (and detecting) deadlocks

Banker's Algorithm

	A	B	C	D
Avail	2	3	0	1

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
1	3	2	1	0
2	2	2	0	0
3	3	5	3	1
4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

Banker's Algorithm

	A	B	C	D
current	2	3	0	1

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
1	3	2	1	0
2	2	2	0	0
3	3	5	3	1
4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

P_2

Banker's Algorithm

	A	B	C	D
current	2	3	0	1

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
1	3	2	1	0
✓ 2	2	2	0	0
3	3	5	3	1
4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

P_2

Banker's Algorithm

	A	B	C	D
current	3	3	1	2

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
1	3	2	1	0
✓ 2	2	2	0	0
3	3	5	3	1
4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

P_2

Banker's Algorithm

	A	B	C	D
current	3	3	1	2

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
1	3	2	1	0
✓ 2	2	2	0	0
3	3	5	3	1
4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

P_2

Banker's Algorithm

	A	B	C	D
current	3	3	1	2

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
1	3	2	1	0
✓ 2	2	2	0	0
3	3	5	3	1
4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

P_2

Banker's Algorithm

	A	B	C	D
current	3	3	1	2

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
✓ 1	3	2	1	0
✓ 2	2	2	0	0
3	3	5	3	1
4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

P_2 P_1

Banker's Algorithm

	A	B	C	D
current	3	6	1	2

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
✓ 1	3	2	1	0
✓ 2	2	2	0	0
3	3	5	3	1
4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

P_2 P_1

Banker's Algorithm

	A	B	C	D
current	3	6	1	2

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
✓ 1	3	2	1	0
✓ 2	2	2	0	0
3	3	5	3	1
✓ 4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

P_2 P_1 P_4

Banker's Algorithm

	A	B	C	D
current	5	8	4	2

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
✓ 1	3	2	1	0
✓ 2	2	2	0	0
3	3	5	3	1
✓ 4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

P_2 P_1 P_4

Banker's Algorithm

	A	B	C	D
current	5	8	4	2

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
✓ 1	3	2	1	0
✓ 2	2	2	0	0
✓ 3	3	5	3	1
✓ 4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

Note: want[j], hold[j], current, avail are arrays!

P_2 P_1 P_4 P_3

Banker's Algorithm

	A	B	C	D
current	5	10	5	2

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
✓ 1	3	2	1	0
✓ 2	2	2	0	0
✓ 3	3	5	3	1
✓ 4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

$\Theta(N^2M)$

Note: want[j], hold[j], current, avail are arrays!

$P_2 P_1 P_4 P_3$

Banker's Algorithm

	A	B	C	D
current	5	10	5	2

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
✓ 1	3	2	1	0
✓ 2	2	2	0	0
✓ 3	3	5	3	1
✓ 4	0	4	1	1

```

current=avail;
for (j = 0; j < N; j++) {
    for (k=0; k<N; k++) {
        if (finished[k])
            continue;
        if (want[k] <= current) {
            finished[k] = 1;
            current += hold[k];
            break;
        }
    }
    if (k==N) {
        printf "Deadlock!\n";
        // finished[k]==0 means process is in
        // the deadlock
        break;
    }
}
    
```

No deadlock

Note: want[j], hold[j], current, avail are arrays!

P_2 P_1 P_4 P_3