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 3 is due **this Friday**
  - Lab 1 is due on Tuesday 2/7 at 11:59 pm
  - Project 1 is due on Friday 2/17 at 11:59 pm
    - Discussed in this week’s recitations

Previous lecture …

- It is easy to make mistakes when using semaphores
- Solution: Mutex and Condition Variables

Problem of the Day

Readers & Writers

- Many processes that may read and/or write
- Only one writer allowed at any time
- Many readers allowed, but not while a process is writing
- Real-world Applications
  - Database queries
  - We have this problem in Project 1

Semaphore-based Solution

Shared variables

```
int nreaders;
Semaphore mutex(1), writing(1);
```

Reader process

```
…
mutex.down();
nreaders += 1;
if (nreaders == 1) // wait if
  writing.down();  // 1st reader
mutex.up();
// Read some stuff
mutex.down();
nreaders -= 1;
if (nreaders == 0)          // signal if
  writing.up();   // last reader
mutex.up();
```

Writer process

```
…
writing.down();
// Write some stuff
writing.up();
…
```

Solution Tracing

- enterRead

Reader process

```
…
mutex.down();
nreaders += 1;
if (nreaders == 1) // wait if
  writing.down();  // 1st reader
mutex.up();
// Read some stuff
mutex.down();
nreaders -= 1;
if (nreaders == 0)         // signal if
  writing.up();   // last reader
mutex.up();
…
```

Solution Tracing

- read

Reader process

```
…
mutex.down();
nreaders += 1;
if (nreaders == 1) // wait if
  writing.down();  // 1st reader
mutex.up();
// Read some stuff
mutex.down();
nreaders -= 1;
if (nreaders == 0)         // signal if
  writing.up();    // last reader
mutex.up();
…
```

Solution Tracing

- doneRead

Reader process

```
…
mutex.down();
nreaders += 1;
if (nreaders == 1) // wait if
  writing.down();  // 1st reader
mutex.up();
// Read some stuff
mutex.down();
nreaders -= 1;
if (nreaders == 0)        // signal if
  writing.up();   // last reader
mutex.up();
…
```

Writer Events

- enterWrite

```
Writer process
…
writing.down();
// Write some stuff
writing.up();
…
```

Writer Events

- write

Writer process

```
…
writing.down();
// Write some stuff
writing.up();
…
```

Writer Events

- doneWrite

Writer process

```
…
writing.down();
// Write some stuff
writing.up();
…
```

Sequence 1

- W0 enterWrite
- W0 write
- R0 enterRead
- R1 enterRead
- R2 enterRead
- W0 doneWrite
- R2 read
- W1 enterWrite
- R2 doneRead
- W1 write

Reader process

```
…
mutex.down();
nreaders += 1;
if (nreaders == 1) // wait if
  writing.down();  // 1st reader
mutex.up();
// Read some stuff
mutex.down();
nreaders -= 1;
if (nreaders == 0)      // signal if
  writing.up();         // last reader
mutex.up();
…
```

Writer process

```
…
writing.down();
// Write some stuff
writing.up();
…
```

Sequence 2

- R0 enterRead
- R0 read
- R1 enterRead
- R1 read
- W0 enterWrite
- R2 enterRead
- R2 read
- R2 doneRead
- R1 doneRead
- R0 doneRead
- W0 write
- W0 doneWrite

Reader process

```
…
mutex.down();
nreaders += 1;
if (nreaders == 1) // wait if
  writing.down();  // 1st reader
mutex.up();
// Read some stuff
mutex.down();
nreaders -= 1;
if (nreaders == 0)      // signal if
  writing.up();         // last reader
mutex.up();
…
```

Writer process

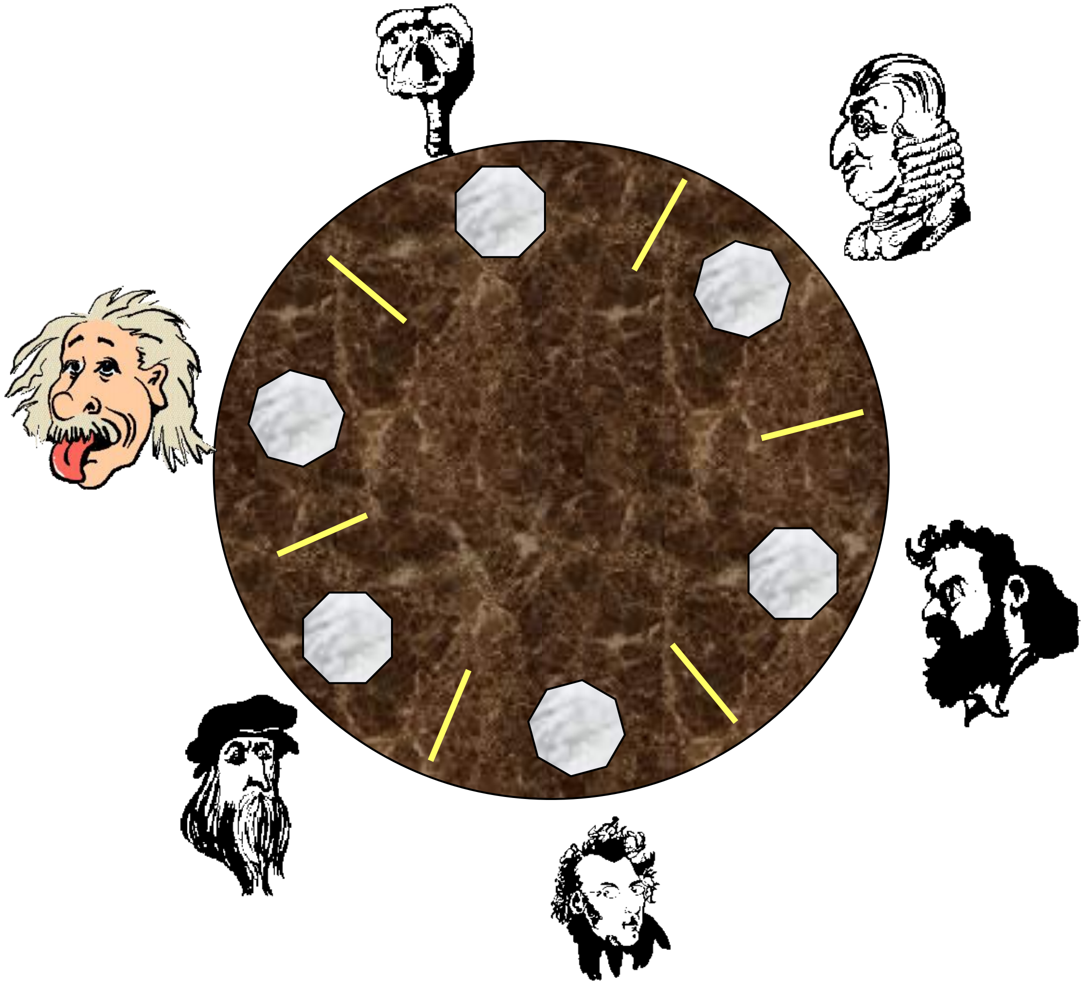
```
…
writing.down();
// Write some stuff
writing.up();
…
```

Solution using Mutex and Condition Variables

- https://cs1550-2214.github.io/cs1550-code-handouts/ProcessSynchronization/Slides/

Dining Philosophers Problem

- *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++)
  - Pi 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

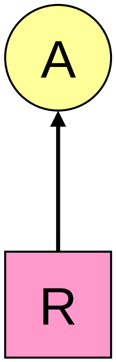
- Pretend there's no problem
- Reasonable if
  - Deadlocks occur very rarely
  - Cost of prevention is high
- UNIX and Windows take this approach
  - Resources (memory, CPU, disk space) are plentiful
  - Deadlocks over such resources rarely occur
  - Deadlocks typically handled by rebooting
- Trade off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

Deadlock Detection

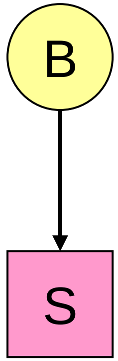
How can the OS detect a deadlock?

Resource allocation graphs

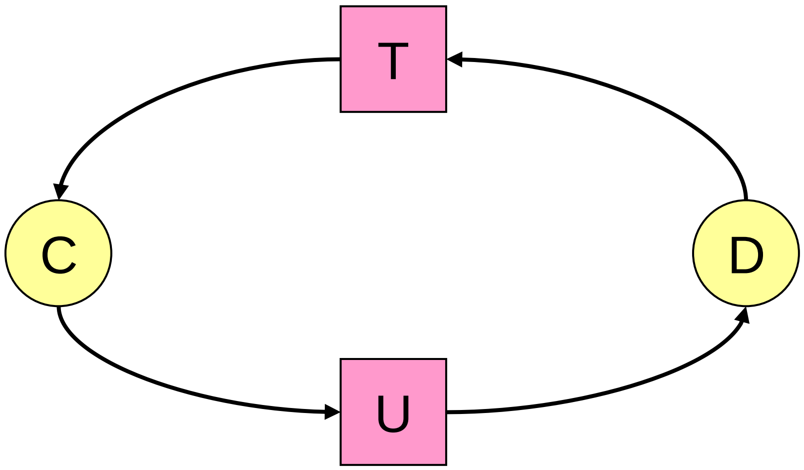
- Resource allocation modeled by directed graphs
- Example 1:
  - Resource R assigned to process A



- 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
}
```

Deadlock Avoidance

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

Deadlock detection 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) { //reached end of loop
   printf "Deadlock!\n";
   // finished[k]==0 means process is in
   // the deadlock
   break;
 }
}
```

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