Synchronization

Reminders:

Lab 7 due Sun 11/29

Quiz 3, Thu Dec 3rd (covers L17-L21)

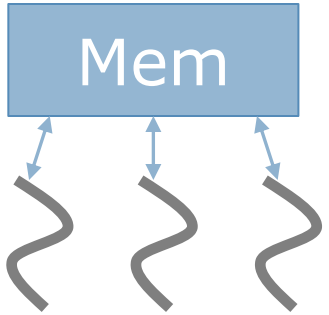
Quiz 3 review, Tue Dec 1st, 7:30-9:30pm

Practice quizzes should be released by Saturday

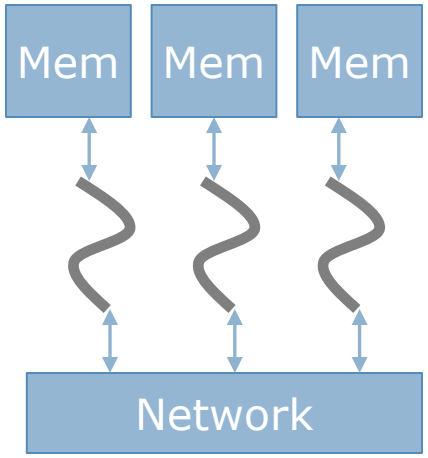
Thread-Level Parallelism

- Divide computation among multiple threads of execution
  - Multiple *independent sequential threads* which compete for shared resources such as memory and I/O devices
  - Multiple *cooperating sequential threads*, which communicate with each other
- Communication models:
  - Shared memory:
    - Single address space
    - Implicit communication by memory loads & stores
  - Message passing:
    - Separate address spaces
    - Explicit communication by sending and receiving messages

Shared Memory

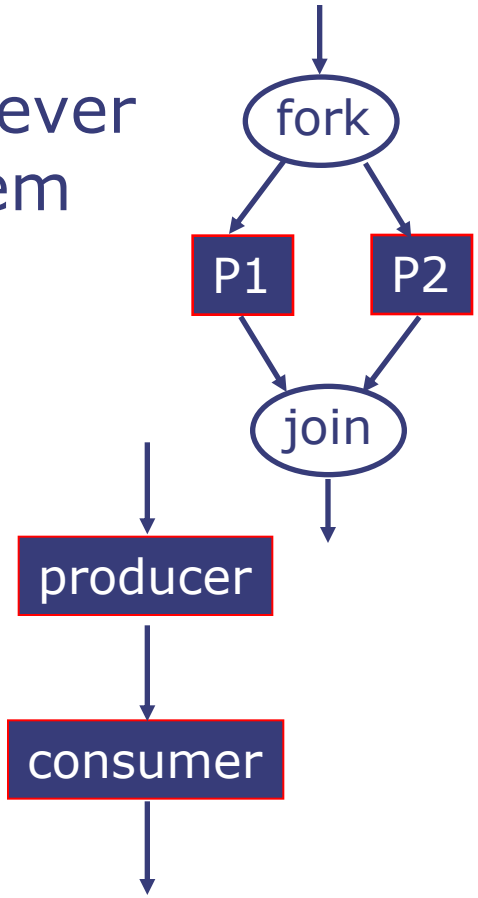


Message Passing



Synchronization

- Need for synchronization arises whenever there are parallel processes in a system
  - *Forks and Joins: A* parallel process may want to wait until several events have occurred
  - *Producer-Consumer:* A consumer process must wait until the producer process has produced data
  - *Mutual Exclusion:* Operating system has to ensure that a resource is used by only one process at a given time

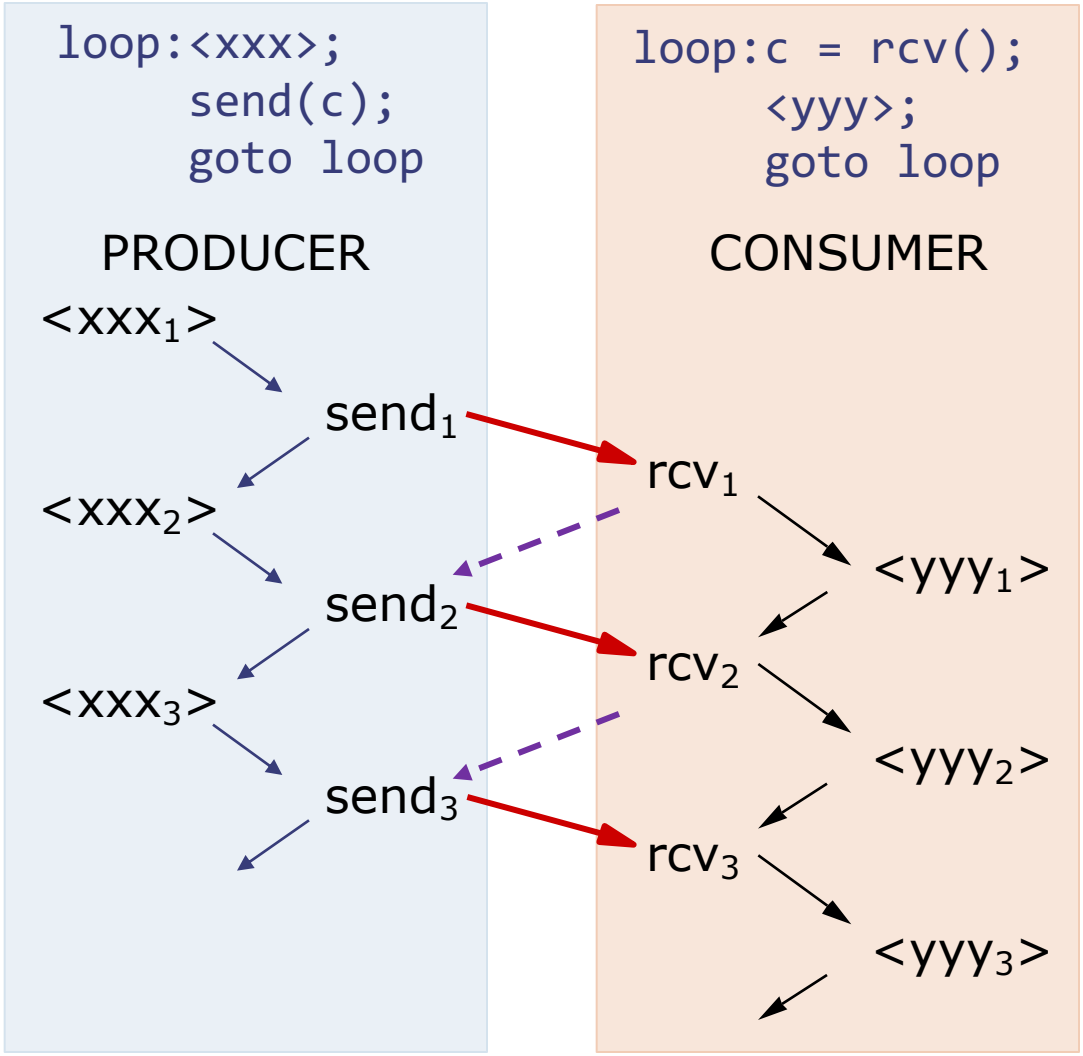


Thread-safe programming

- Multithreaded programs can be executed on a uniprocessor by *timesharing*
  - Each thread is executed for a while (timer interrupt) and then the OS switches to another thread, repeatedly
- *Thread-safe* multithreaded programs behave the same way regardless of whether they are executed on multiprocessors or a single processor

In this lecture, we will assume that each thread has its own processor to run on

Synchronous Communication



Precedence Constraints:

$a \preceq b$

"a *precedes* b"

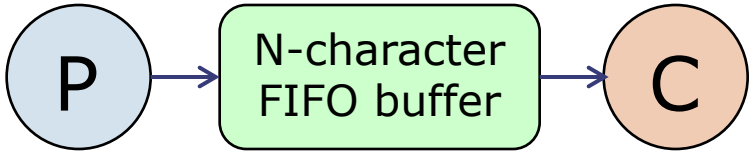
- Can't consume data before it's produced

$send_i \preceq rcv_i$

- Producer can't "overwrite" data before it's consumed

$rcv_i \preceq send_{i+1}$

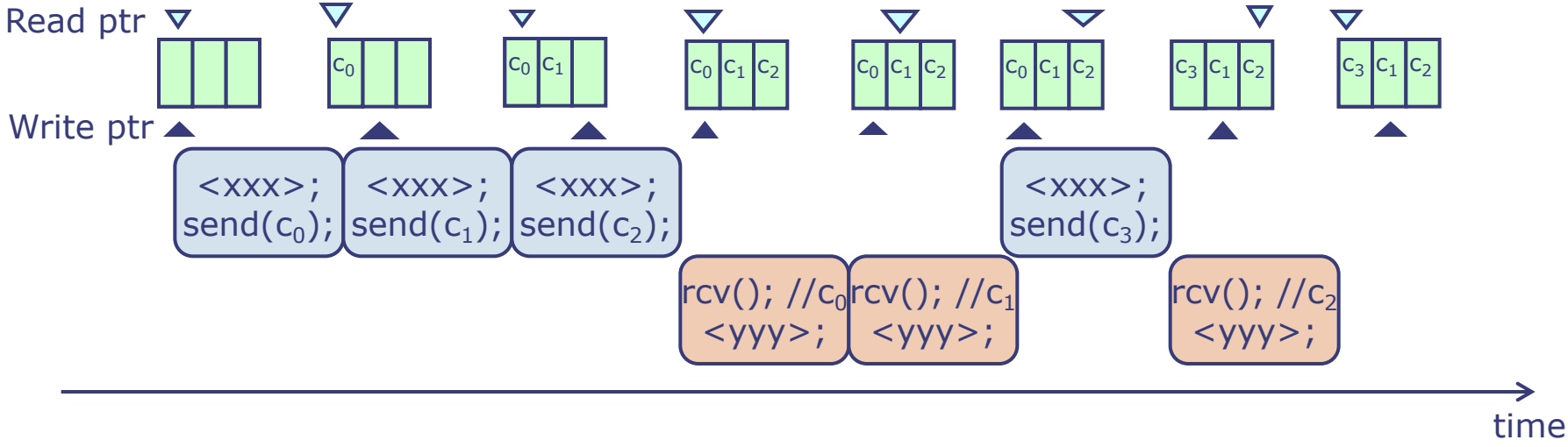
FIFO (First-In First-Out) Buffers



FIFO buffers relax synchronization constraints. The producer can run up to N values ahead of the consumer:

$$rcv_i \preceq send_{i+N}$$

Typically implemented as a "Ring Buffer" in shared memory:



Shared-memory FIFO Buffer: First Try

SHARED MEMORY:

```
char buf[N];           /* The buffer */
int in=0, out=0;
```

PRODUCER:

```
void send(char c){
  buf[in] = c;
  in = (in + 1) % N;
}
```

CONSUMER:

```
char rcv(){
  char c;
  c = buf[out];
  out = (out + 1) % N;
  return c;
}
```

Not correct. Why?

Doesn't enforce any precedence constraints
(e.g., rcv() could be invoked prior to any send())

Semaphores (Dijkstra, 1962)

Programming construct for synchronization:

- New data type: *semaphore*, an integer ≥ 0

```
semaphore s = K;  // initialize s to K
```

- New operations (defined on semaphores):
  - `wait(semaphore s)`

    wait until s > 0, then s = s – 1
  - `signal(semaphore s)`

    s = s + 1 (one waiting thread may now be able to proceed)

- Semantic guarantee: A semaphore s initialized to K enforces the precedence constraint:

$$\text{signal}(s)_i \prec \text{wait}(s)_{i+K}$$

The i^{th} call to signal(s) must complete before the $(i+K)^{th}$ call to wait(s) completes

Semaphores for Precedence

```
semaphore s = 0;

Thread A        Thread B

A1;             B1;

A2;             B2;
signal(s);
A3;             B3;
                wait(s);
A4;             B4;

A5;             B5;
```

Goal: Want statement A2 in thread A to complete before statement B4 in thread B begins.

$A2 \prec B4$

Recipe:

- Declare semaphore = 0
- `signal(s)` at start of arrow
- `wait(s)` at end of arrow

Semaphores for Resource Allocation

Abstract problem:

- Pool of K resources
- Many threads, each needs resource for occasional uninterrupted period
- Must guarantee that at most K resources are in use at any time

Solution using semaphores:

In shared memory:

```
semaphore s = K;  // K resources
```

Using resources:

```
wait(s);    // Allocate a resource
…           // use it for a while
signal(s);  // return it to pool
```

Invariant: Semaphore value = number of resources left in pool

FIFO Buffer with Semaphores: 2nd Try

SHARED MEMORY:

```
char buf[N];            /* The buffer */
int in = 0, out = 0;
semaphore chars = 0;
```

PRODUCER:

```
void send(char c) {
   buf[in] = c;
   in = (in + 1) % N;
   signal(chars);
}
```

CONSUMER:

```
char rcv() {
   char c;
   wait(chars);
   c = buf[out];
   out = (out + 1) % N;
   return c;
}
```

Precedence managed by semaphore: $send_i \prec rcv_i$
Resource managed by semaphore: # of chars in buf

Still not correct. Why? Producer can overflow buffer.
Must enforce $rcv_i \prec send_{i+N}$!

FIFO Buffer with Semaphores

Correct implementation (for single producer + single consumer)

SHARED MEMORY:

```
char buf[N];              /* The buffer */
int in = 0, out = 0;
semaphore chars = 0, spaces = N;
```

PRODUCER:

```
void send(char c) {
   wait(spaces);
   buf[in] = c;
   in = (in + 1) % N;
   signal(chars);
}
```

CONSUMER:

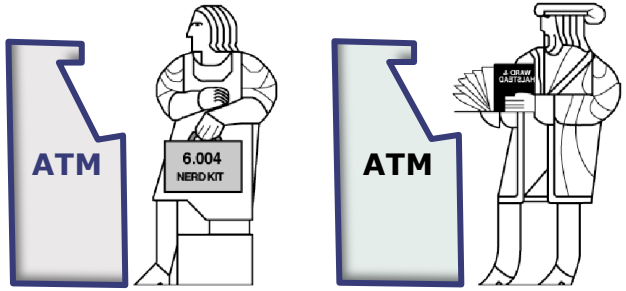
```
char rcv() {
   char c;
   wait(chars);
   c = buf[out];
   out = (out + 1) % N;
   signal(spaces);
   return c;
}
```

Resources managed by semaphores: characters in FIFO, spaces in FIFO.

Works with single producer and consumer. But what about multiple producers and consumers?

Simultaneous Transactions

Suppose you and your friend visit the ATM at exactly the same time, and remove $50 from your account. What happens?



debit(6004, 50) debit(6004, 50)

```
void debit(int account, int amount) {
   t = balance[account];
   balance[account] = t – amount;
}
```

What is *supposed* to happen?

```
// assume t0 has address of balance[account]
```

Thread # 1

```
lw t1, 0(t0)
sub t1, t1, a1
sw t1, 0(t0)
…
```

Thread #2

```
…
lw t1, 0(t0)
sub t1, t1, a1
sw t1, 0(t0)
```

Result: You have $100, and your bank balance is $100 less.

But What If Both Calls Interleave?

```
// assume t0 has address of balance[account]
```

Thread # 1	Thread #2
`lw t1, 0(t0)`	
	`lw t1, 0(t0)`
	`sub t1, t1, a1`
	`sw t1, 0(t0)`
`sub t1, t1, a1`	
`sw t1, 0(t0)`	
…	…

Result: You have $100, and your bank balance is only $50 less!

We need to be careful when writing concurrent programs. In particular, when modifying shared data.

For certain code segments, called critical sections, we would like to ensure that no two executions overlap.

This constraint is called mutual exclusion.

Solution: embed critical sections in wrappers (e.g., "transactions") that guarantee their atomicity, i.e., make them appear to be single, instantaneous operations.

Semaphores for Mutual Exclusion

```
semaphore lock = 1;

void debit(int account, int amount) {
   wait(lock);   // Wait for exclusive access
   t = balance[account];
   balance[account] = t – amount;
   signal(lock); // Finished with lock
}
```

a <> b

"a *precedes* b
or
b *precedes* a"
(i.e., they don't overlap)

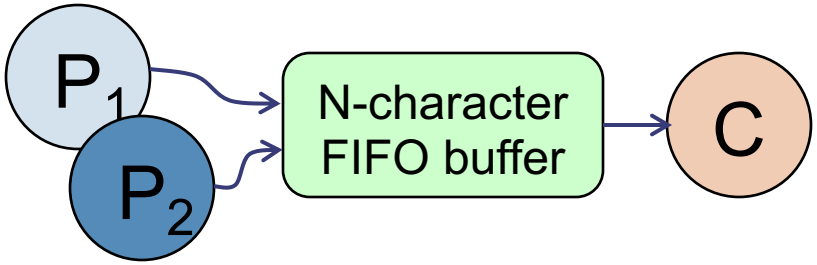
Lock controls access to critical section

Issue: Lock granularity

- One lock for all accounts?
- One lock per account?
- One lock for all accounts ending in 004?

Producer/Consumer Atomicity Problems

Consider multiple producer threads:



P_1	P_2
...	...
buf[in] = c;	buf[in] = c;
in = (in+1) % N;	in = (in+1) % N;
...	...

Problem: Producers interfere with each other

FIFO Buffer with Semaphores

Correct multi-producer, multi-consumer implementation

SHARED MEMORY:

```
char buf[N];            /* The buffer */
int in = 0, out = 0;
semaphore chars = 0, spaces = N;
semaphore lock = 1;
```

PRODUCER:

```
void send(char c) {
  wait(spaces);
  wait(lock);
  buf[in] = c;
  in = (in + 1) % N;
  signal(lock);
  signal(chars);
}
```

CONSUMER:

```
char rcv() {
  char c;
  wait(chars);
  wait(lock);
  c = buf[out];
  out = (out + 1) % N;
  signal(lock);
  signal(spaces);
  return c;
}
```

The Power of Semaphores

SHARED MEMORY:

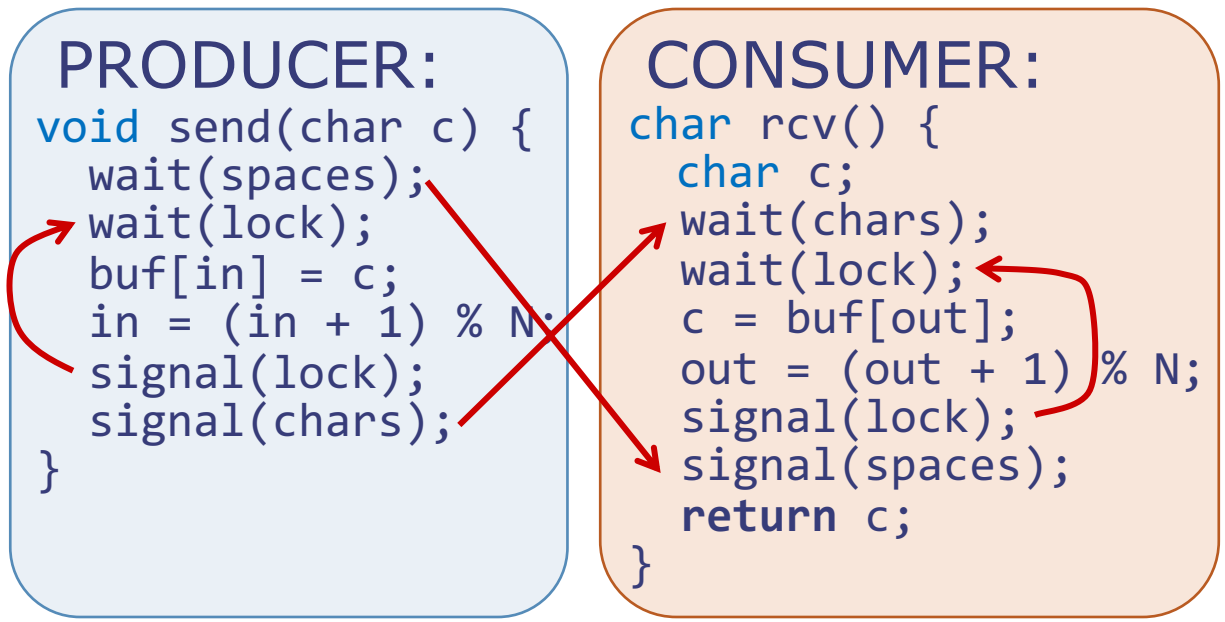
```
char buf[N];            /* The buffer */
int in = 0, out = 0;
semaphore chars = 0, spaces = N;
semaphore lock = 1;
```

PRODUCER:

```
void send(char c) {
  wait(spaces);
  wait(lock);
  buf[in] = c;
  in = (in + 1) % N;
  signal(lock);
  signal(chars);
}
```

CONSUMER:

```
char rcv() {
  char c;
  wait(chars);
  wait(lock);
  c = buf[out];
  out = (out + 1) % N;
  signal(lock);
  signal(spaces);
  return c;
}
```



A single synchronization primitive that enforces both:

Precedence relationships:

$send_i \prec rcv_i$

$rcv_i \prec send_{i+N}$

Mutual-exclusion relationships:

protect variables *in* and *out*

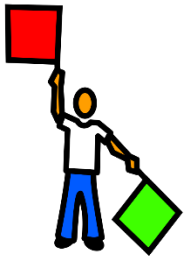
Semaphore Implementation

Semaphores are themselves shared data and implementing `wait` and `signal` operations require read/modify/write sequences that must be executed as critical sections. So how do we guarantee mutual exclusion in these particular critical sections without using semaphores?

Approaches:

- Use a special instruction (e.g., "test and set") that performs an atomic read-modify-write. Depends on atomicity of single instruction execution. This is the most common approach.

- Implement them using system calls. Works in uniprocessors only, where the kernel is uninterruptible.

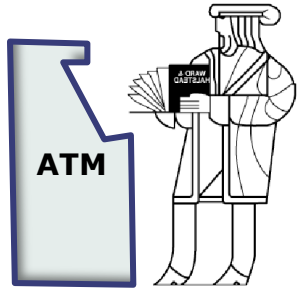


Synchronization: The Dark Side

The naïve use of synchronization constraints can introduce its own set of problems, particularly when a thread requires access to more than one protected resource.

```
void transfer(int account1, int account2, int amount) {
   wait(lock[account1]);
   wait(lock[account2]);
   balance[account1] = balance[account1] - amount;
   balance[account2] = balance[account2] + amount;
   signal(lock[account2]);
   signal(lock[account1]);
}
```

transfer(6031, 6004, 50)



transfer(6004, 6031, 50)

What can go wrong here?

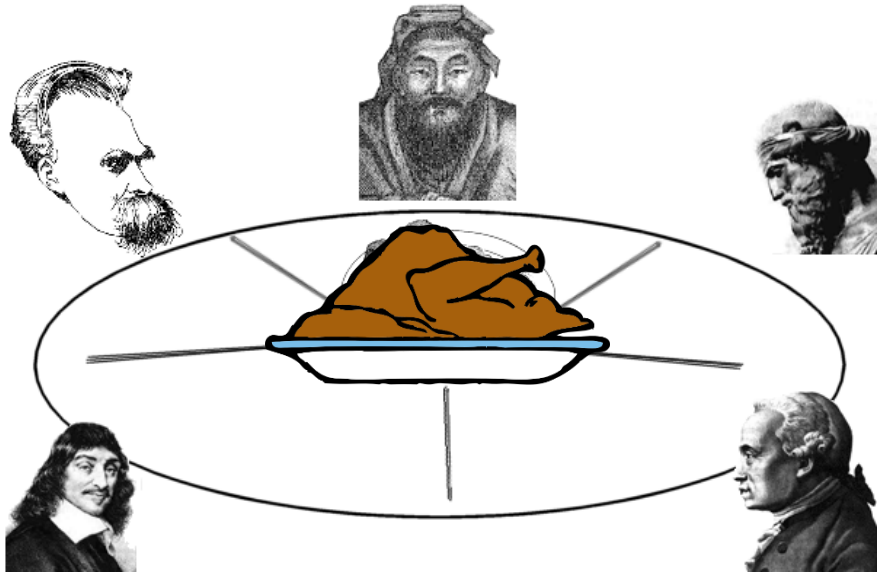
```
Thread 1: wait(lock[6031]);
Thread 2: wait(lock[6004]);
Thread 1: wait(lock[6004]); // cannot complete
                            // until thread 2 signals
Thread 2: wait(lock[6031]); // cannot complete
                            // until thread 1 signals
```

No thread can make progress → Deadlock

Dining Philosophers

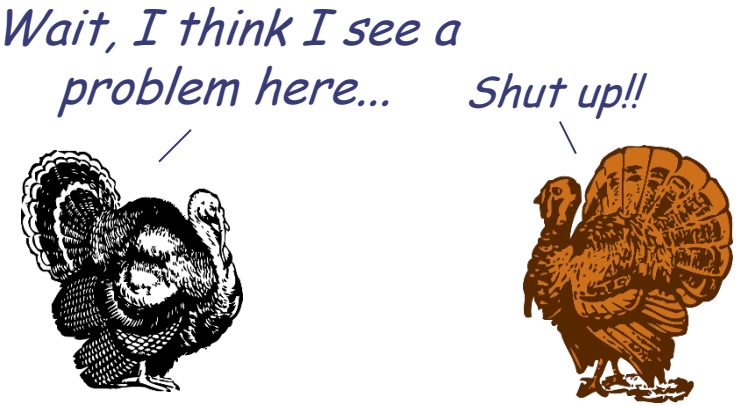
Philosophers think deep thoughts, but have simple secular needs. When hungry, a group of N philosophers will sit around a table with N chopsticks interspersed between them. Food is served, and each philosopher enjoys a leisurely meal using the chopsticks on either side to eat.

They are exceedingly polite and patient, and each follows the following dining protocol:



Philosopher's algorithm:

- Take (wait for) LEFT stick
- Take (wait for) RIGHT stick
- EAT until sated
- Replace both sticks



Deadlock!

No one can make progress because they are all waiting for an unavailable resource.

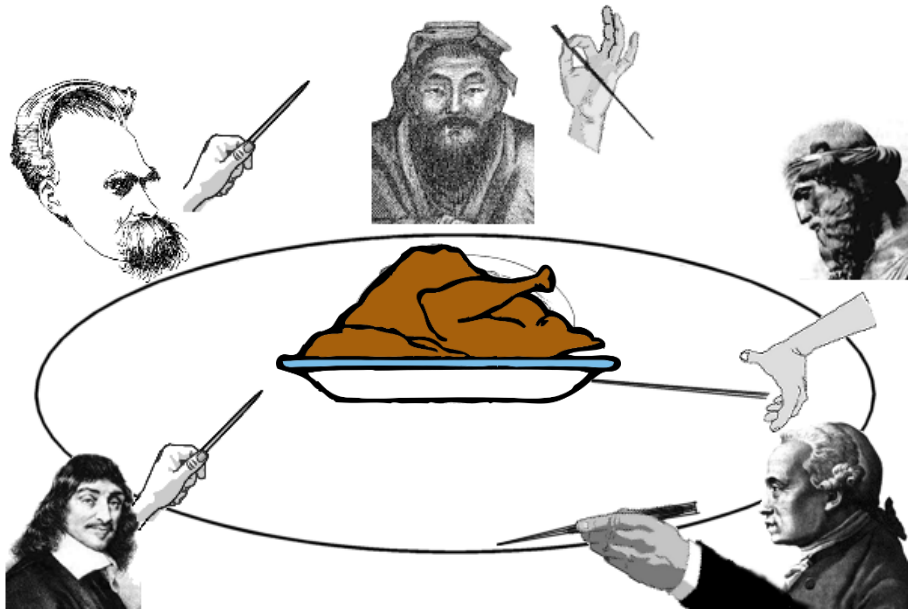
CONDITIONS:

1) Mutual exclusion: Only one thread can hold a resource at a given time

2) Hold-and-wait: A thread holds allocated resources while waiting for others

3) No preemption: A resource can not be removed from a thread holding it

4) Circular wait



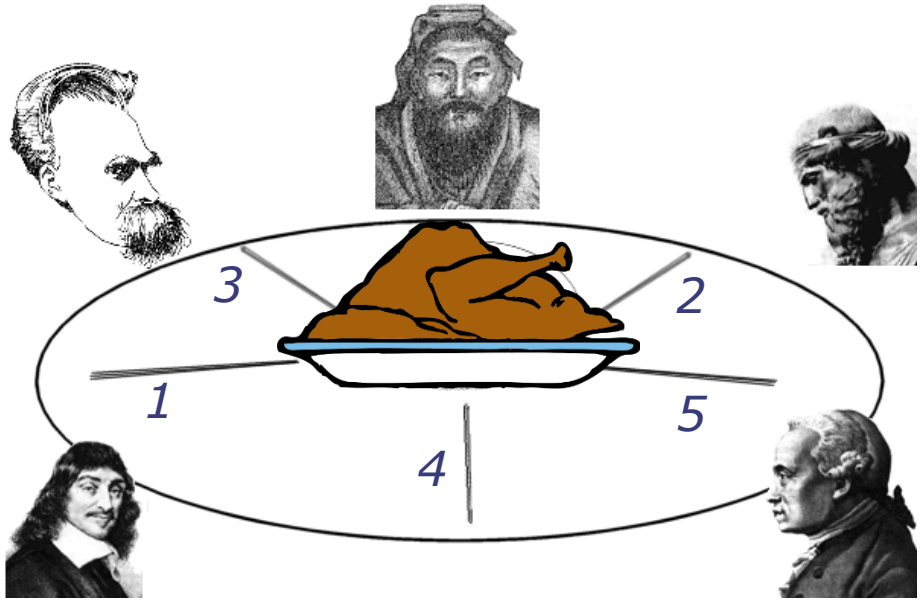
SOLUTIONS:
Avoidance
-or-
Detection and Recovery

One Solution

Assign a unique number to each chopstick, request resources in a consistent order:

New Algorithm:

- Take LOW stick
- Take HIGH stick
- EAT
- Replace both sticks.



Simple proof:

Deadlock means that each philosopher is waiting for a resource held by some other philosopher …

But, the philosopher holding the highest numbered chopstick can't be waiting for any other philosopher (no hold-and-wait cycle) …

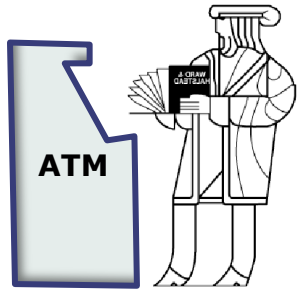
Thus, there can be no deadlock.

Example: Dealing With Deadlocks

Can you fix the transfer method to avoid deadlock?

```
void transfer(int account1, int account2, int amount) {
  int a = min(account1, account2);
  int b = max(account1, account2);
  wait(lock[a]);
  wait(lock[b]);
  balance[account1] = balance[account1] - amount;
  balance[account2] = balance[account2] + amount;
  signal(lock[b]);
  signal(lock[a]);
}
```

Transfer(6031, 6004, 50)



Transfer(6004, 6031, 50)

Summary

- Communication among parallel threads or asynchronous processes requires synchronization
  - Precedence constraints: a partial ordering among operations
  - Semaphores as a mechanism for enforcing precedence constraints
  - Mutual exclusion (critical sections, atomic transactions) as a common compound precedence constraint
  - Solving Mutual Exclusion via binary semaphores
  - Synchronization serializes operations, limits parallel execution
- Many alternative synchronization mechanisms exist!
- Deadlock:
  - Consequence of undisciplined use of synchronization mechanism
  - Can be avoided in special cases, detected and corrected in others

Thank you!

Next lecture:
Cache Coherency