19. Concurrency & Synchronization

6.004x Computation Structures

Part 3 – Computer Organization

Interprocess Communication

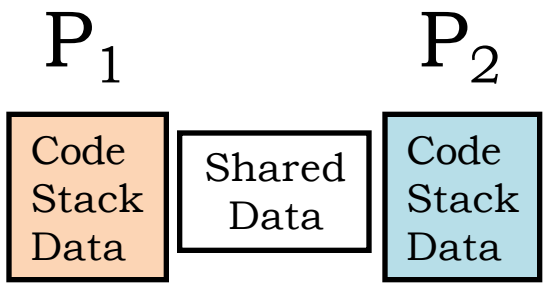
Interprocess Communication

Why have multiple processes?

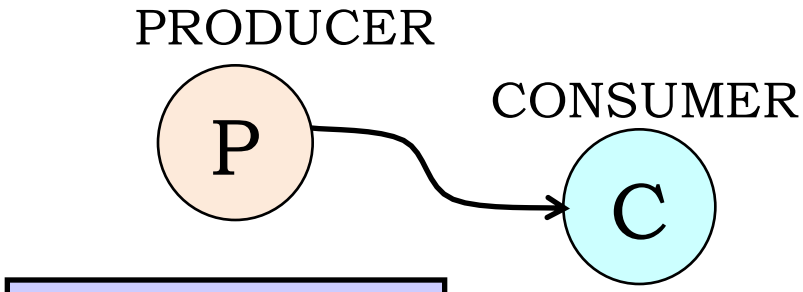
- Concurrency
- Asynchrony
- Processes as a programming primitive
- Data-/Event-driven

How to communicate?

- Shared Memory (overlapping contexts)…
- Synchronization instructions (hardware support)
- Supervisor calls



Classic Example:
"Producer-Consumer" Problem




```
loop: <xxx>;
      send(c);
      goto loop
```

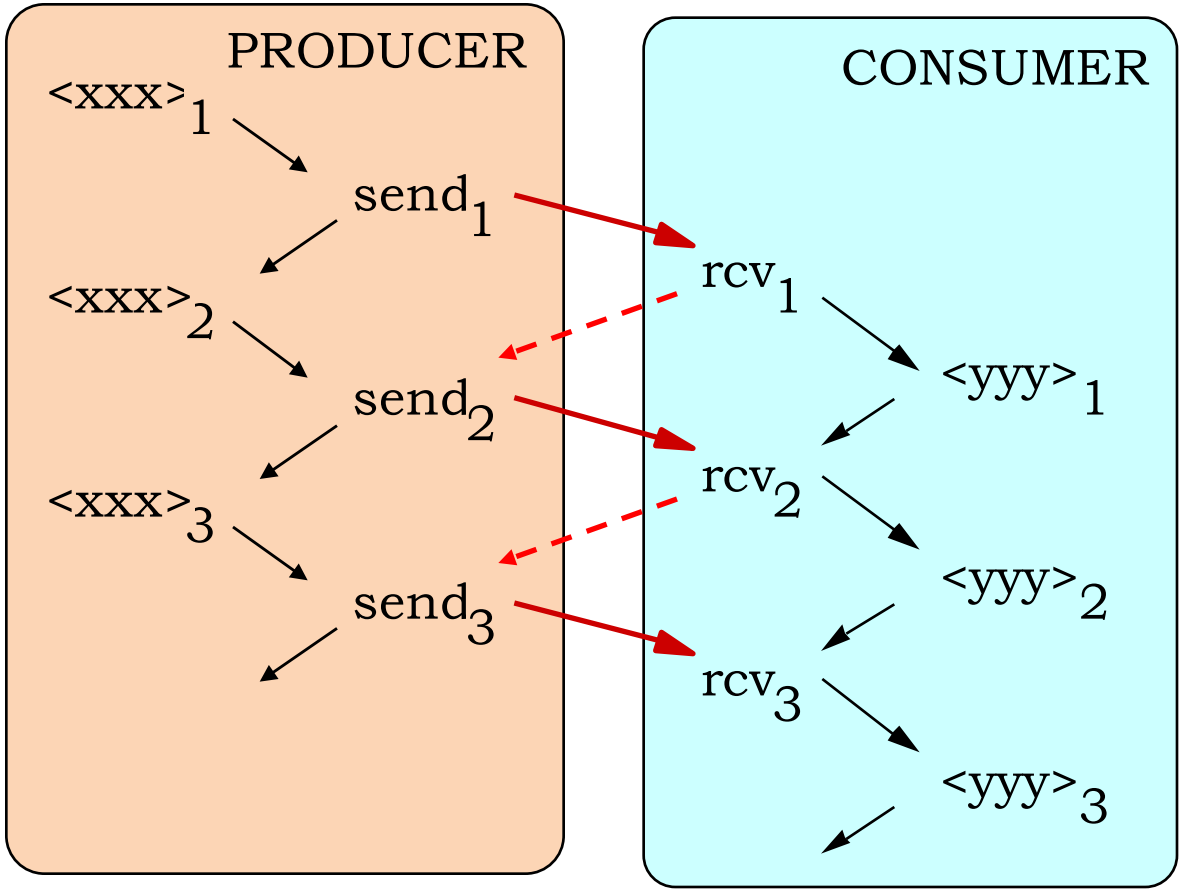
```
loop: c = rcv();
      <yyy>;
      goto loop
```

Real-World Examples:
Compiler/Assembler,
Application Frontend/Backend,
UNIX pipeline

Synchronous Communication

```
loop:  <xxx>;
       send(c);
       goto loop
```

```
loop:  c = rcv();
       <yyy>;
       goto loop
```



Precedence Constraints:

$a \prec b$

"a *precedes* b"

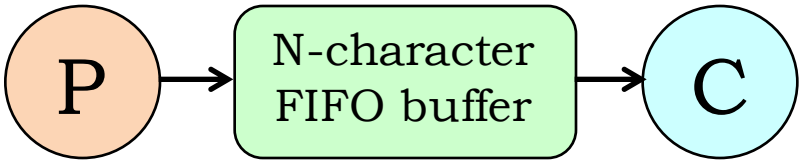
- Can't consume data before it's produced

$send_i \prec rcv_i$ →

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

$rcv_i \prec send_{i+1}$ ----→

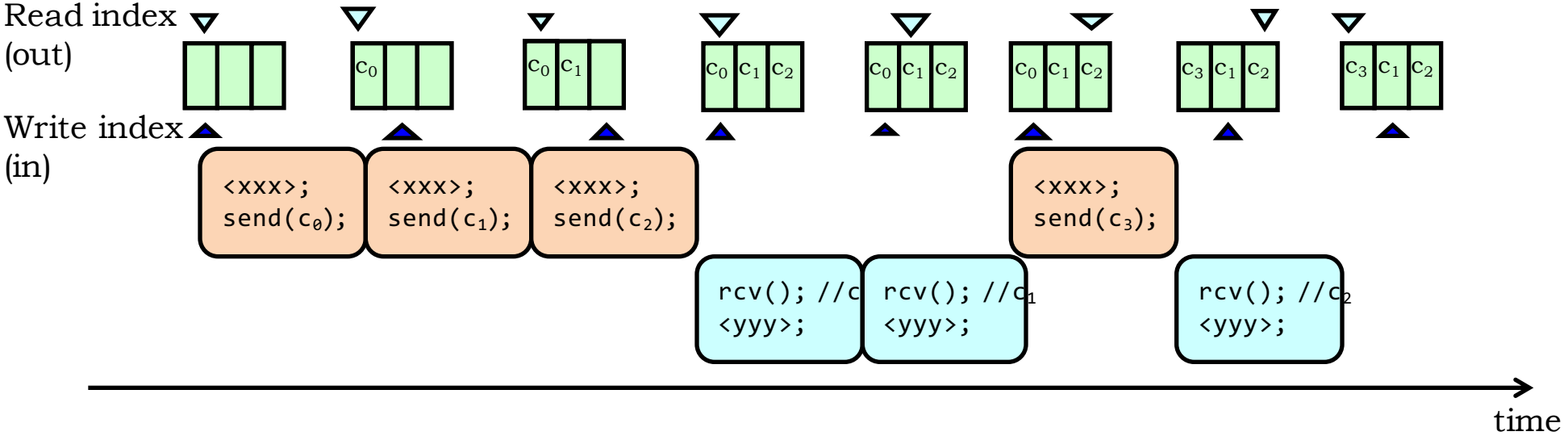
FIFO Buffering



RELAXES interprocess synchronization constraints. Buffering relaxes the following OVERWRITE constraint to:

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

"Circular Buffer:"



Example: Bounded Buffer Problem

SHARED MEMORY:

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

PRODUCER:

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

CONSUMER:

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

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

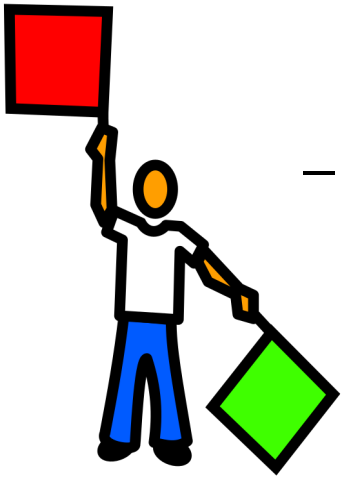
Semaphores

Semaphores (Dijkstra)

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)`

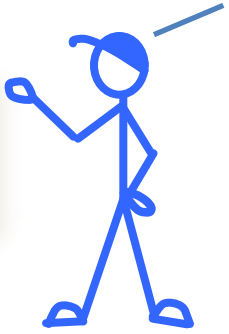
    wait until s > 0, then s = s – 1
  - `signal(semaphore s)`

    s = s + 1 (one WAITing process may now be able to proceed)

- SEMANTIC GUARANTEE: A semaphore s initialized to K enforces the constraint:

Often you will see
P(s) used for wait(s)
and
V(s) used for signal(s)!
P = "proberen" (test) or
"pakken" (grab)
V= "verhogen" (increase)

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



This is a precedence relationship: the i^{th} call to signal must complete before the the $(i+K)^{th}$ call to wait will succeed.

Semaphores for Precedence

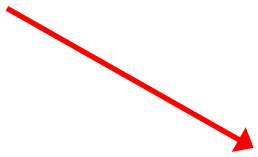
```
semaphore s = 0;

Process A        Process B

A1;              B1;

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

A5;              B5;
```



Goal: want statement A2 in process A to complete before statement B4 in Process B begins.

A2 ≺ 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 processes, each needs resource for occasional uninterrupted period
- MUST guarantee that at most K resources are in use at any time.

Semaphore Solution:

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

Bounded Buffer Problem w/ Semaphores

SHARED MEMORY:

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

PRODUCER:

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

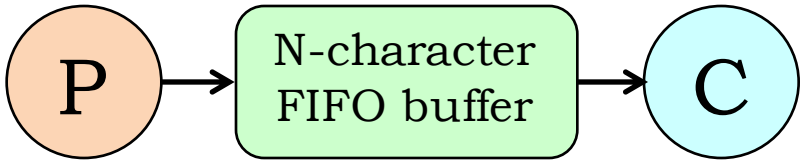
CONSUMER:

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

Does this work?

PRECEDENCE managed by semaphore: $send_i < rcv_i$
RESOURCE managed by semaphore chars: # of chars in buf

Flow Control Problems



Q: What keeps PRODUCER from putting N+1 characters into the N-character buffer?

A: Nothing.

Result: OVERFLOW.

WHAT we've got thus far:

$send_i < rcv_i$

WHAT we still need:

$rcv_i < send_{i+N}$

Bounded Buffer Problem w/ more Semaphores

SHARED MEMORY:

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

PRODUCER:

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

CONSUMER:

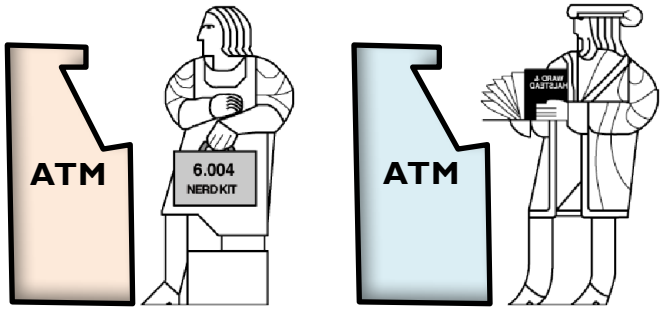
```
char rcv()
{
    char c;
    wait(chars);
    c = buf[out];
    out = (out+1)%N;
    signal(space);
    return c;
}
```

Resources managed by semaphore: characters in FIFO, spaces in FIFO. Works with single producer, consumer. But what about multiple producers and consumers?

Atomic Transactions

Simultaneous Transactions

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



Withdraw(6004, 50) Withdraw(6004, 50)

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

What is *supposed* to happen?

Process # 1	Process #2
`LD(R10, balance, R0)`	
`SUB(R0, R1, R0)`	
`ST(R0, balance, R10)`	
…	…
	`LD(R10, balance, R0)`
	`SUB(R0, R1, R0)`
	`ST(R0, balance, R10)`

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

But, What If…

```
Process # 1          Process #2
LD(R10, balance, R0)
                     …
                     LD(R10, balance, R0)
                     SUB(R0, R1, R0)
                     ST(R0, balance, R10)
                            …
SUB(R0, R1, R0)
ST(R0, balance, R10)
…
```

NET: You have $100 and your bank balance is $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;

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

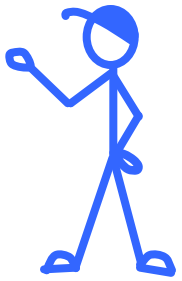
a <> b

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

RESOURCE managed by "lock" semaphore:
Access to critical section

ISSUES:

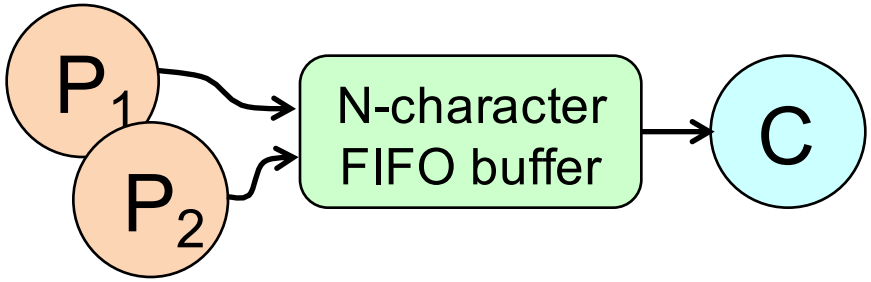
Granularity of lock
1 lock for whole balance database?
1 lock per account?
1 lock for all accounts ending in 004



Look up "database" on Wikipedia to learn about systems that support efficient transactions on shared data.

Producer/Consumer Atomicity Problems

Consider multiple PRODUCER processes:



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

BUG: Producers interfere with each other.

Bounded Buffer Problem w/ even more Semaphores

SHARED MEMORY:

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

PRODUCER:

```
send(char c)
{
   wait(space);
   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(space);
   return c;
 }
```

The Power of Semaphores

SHARED MEMORY:

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

PRODUCER:

```
send(char c)
{
   wait(space);
   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(space);
   return c;
}
```

A single synchronization primitive that enforces both:

Precedence relationships:

$send_i < rcv_i$

$rcv_i < send_{i+N}$

Mutual-exclusion relationships:

protect variables *in* and *out*

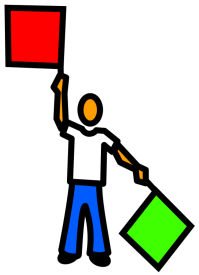
Semaphore Implementation

Semaphore Implementation

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

Approaches:

- SVC implementation, using atomicity of kernel handlers. Works in timeshared processor sharing a single uninterruptable kernel.
- Implementation of a simple lock using a special instruction (e.g. "test and set"), depends on atomicity of single instruction execution. Works with shared-bus multiprocessors supporting atomic read-modify-write bus transactions. Using a simple lock to implement critical sections, we can use software to implement other semaphore functionality.
- Implementation using atomicity of individual read or write operations. For example, see "Dekker's Algorithm" on Wikipedia.



Semaphores as a Supervisor Call

```
wait_h( ) {
    int *addr;
    addr = VtoP(User.Regs[R0]);     // get arg
    if (*addr <= 0) {
        User.Regs[XP]  = User.Regs[XP] – 4;
        sleep(addr);
    } else
        *addr = *addr - 1;
}

signal_h( ) {
    int *addr;
    addr = VtoP(User.Regs[R0]);     // get arg
    *addr = *addr + 1;
    wakeup(addr);
}
```

Calling sequence:

```
…
// put address of lock
// into R0
CMOVE(lock, R0)
SVC(WAIT) or SVC(SIGNAL)
```

SVC call is not interruptible since it is executed in kernel mode.

Hardware Support for Semaphores

TCLR(RA, literal, RC) test and clear location

```
PC ← PC + 4
EA ← Reg[Ra] + literal
Reg[Rc] ← MEM[EA]
MEM[EA] ← 0
```

} *Atomicity guaranteed by memory*

Executed ATOMICALLY (cannot be interrupted)

Can easily implement mutual exclusion using binary semaphore

```
wait:   TCLR(R31, lock, R0)
        BEQ(R0,wait)
        … critical section …
        CMOVE(1,R0)
        ST(R0, lock, R31)
```

} wait(lock)

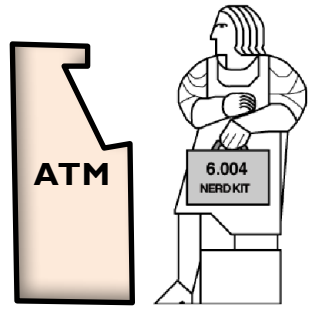
} signal(lock)

Deadlock

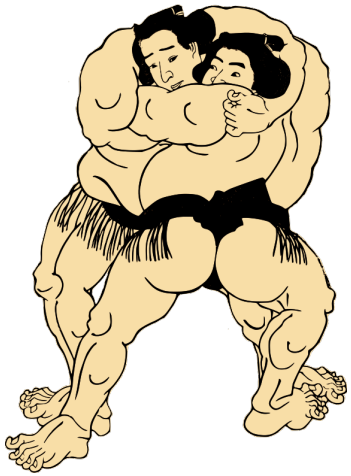
Synchronization: The Dark Side

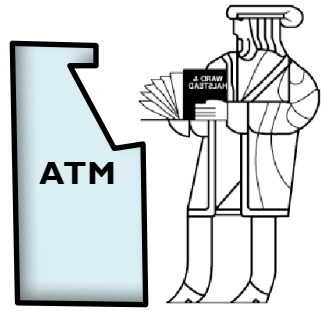
The naïve use of synchronization constraints can introduce its own set of problems, particularly when a process requires access to more than one protected resource.

```
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(6005, 6004, 50)





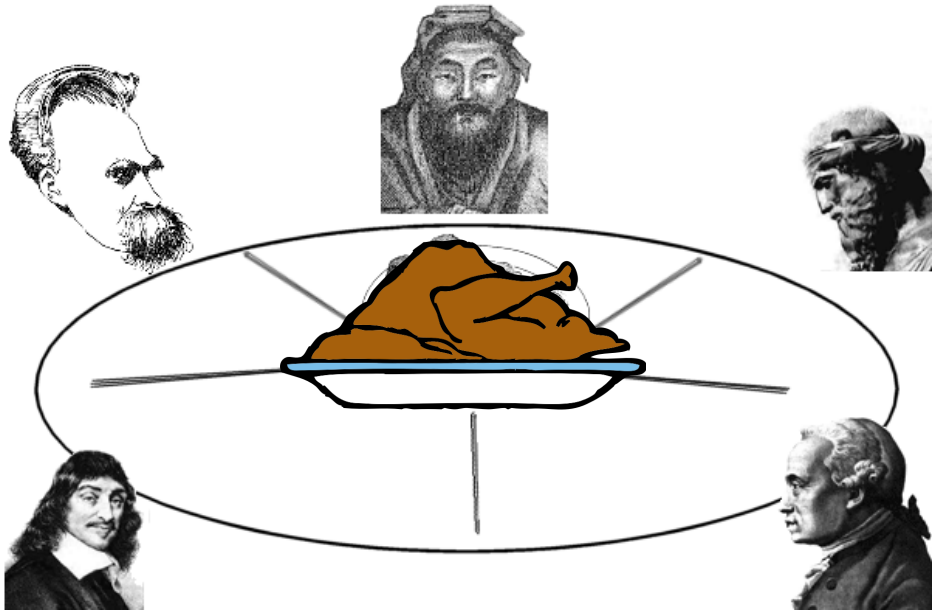
Transfer(6004, 6005, 50)

DEADLOCK (aka "deadly embrace")!

Dining Philosophers

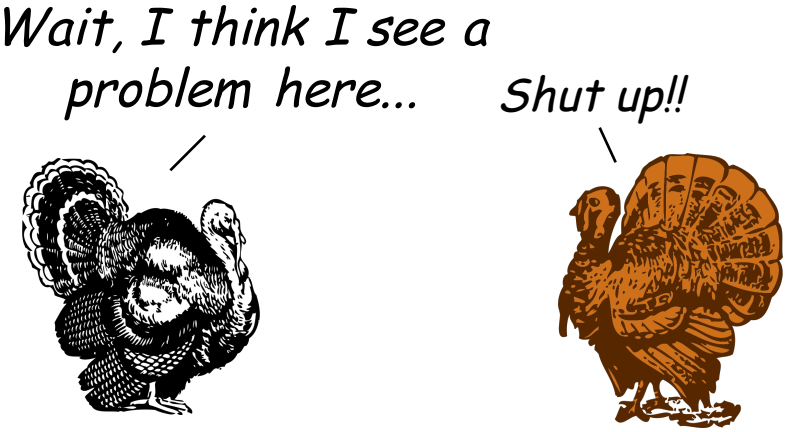
Philosophers think deep thoughts, but have simple secular needs. When hungry, a group of N philosophers will sit around a table with N chopsticks interspersed between them. Food is served, and each philosopher enjoys a leisurely meal using the chopsticks on either side to eat.

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

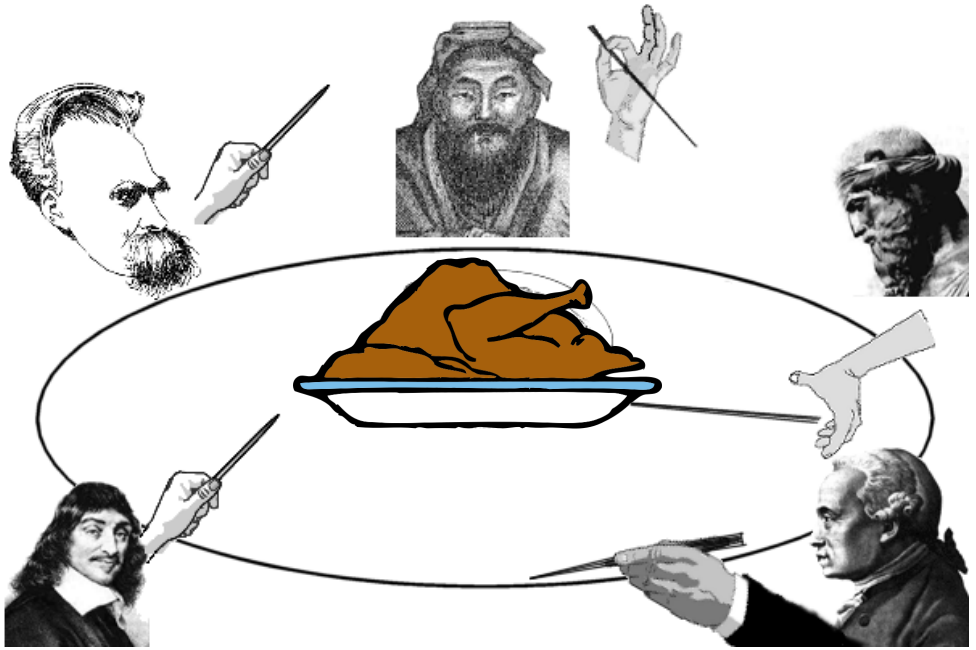
CONDITIONS:

1) Mutual exclusion - only one process can hold a resource at a given time

2) Hold-and-wait - a process holds allocated resources while waiting for others

3) No preemption - a resource can not be removed from a process holding it

4) Circular Wait



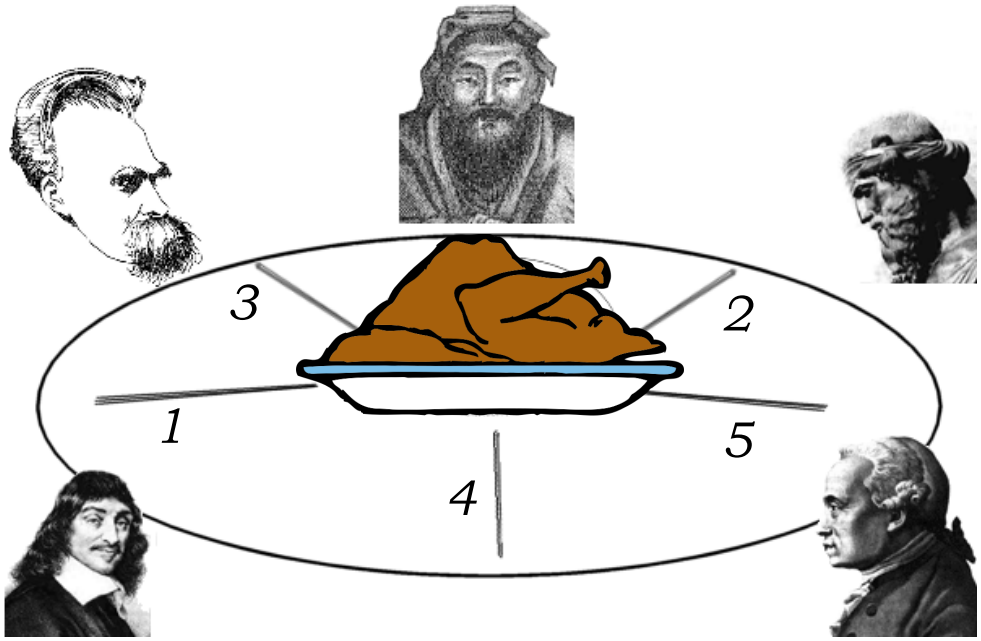
SOLUTIONS:
Avoidance
-or-
Detection and Recovery

One Solution

KEY: Assign a unique number to each chopstick, request resources in globally 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) …

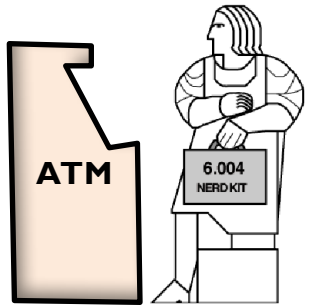
Thus, there can be no deadlock.

Dealing With Deadlocks

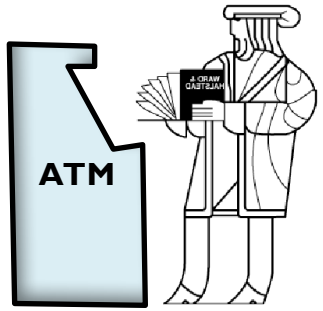
Cooperating processes:

- Establish a fixed ordering to shared resources and require all requests to be made in the prescribed order

```
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(6005, 6004, 50)



Transfer(6004, 6005, 50)

Unconstrained processes:

- O/S discovers circular wait & kills waiting process
- Transaction model
- Hard problem

Summary

Communication among 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!

Deadlocks:

- Consequence of undisciplined use of synchronization mechanism
- Can be avoided in special cases, detected and corrected in others.