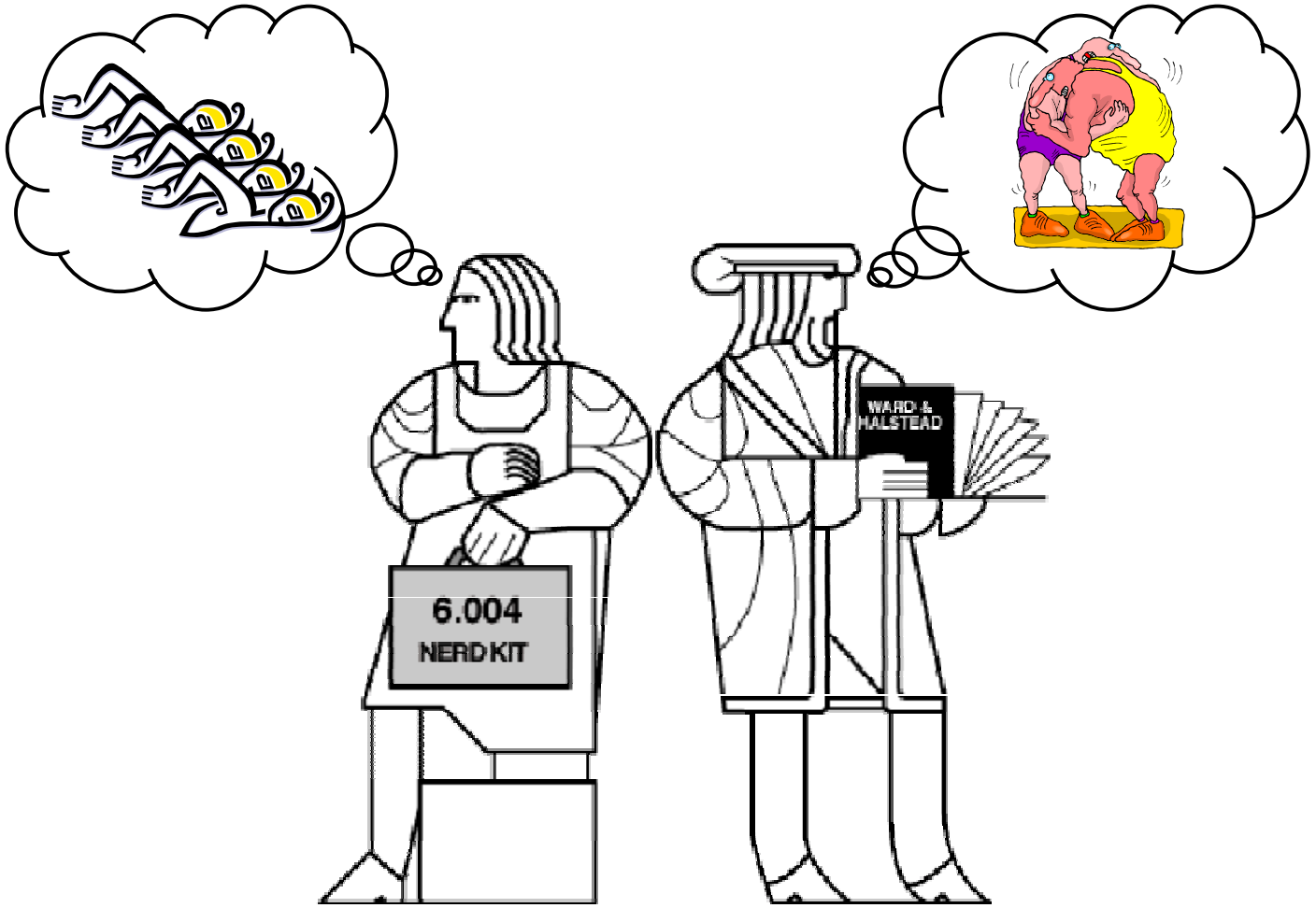
Synchronization & Deadlock



Handouts: Lecture Slides

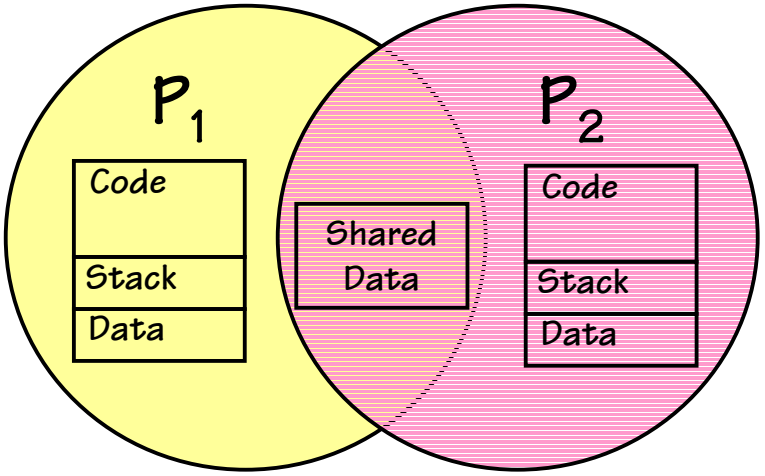
Interprocess Communication

Why communicate?

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

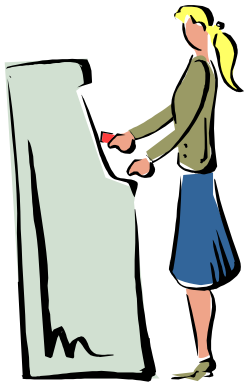
How to communicate?

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



Problems with Concurrency

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)

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

What could possibly 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
LD(R10, balance, R0)

SUB(R0, R1, R0)
ST(R0, balance, R10)
…

Process #2

…
LD(R10, balance, R0)
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 assure that no two executions overlap.

This constraint is called MUTUAL-EXCLUSION.

Semaphores (Dijkstra)

Programming construct for synchronization:

- NEW DATA TYPE: *semaphore*, integer-valued

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

- NEW OPERATIONS (defined on semaphores):

  `wait(semaphore s)`

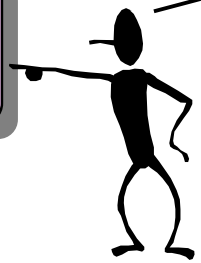
  stall current process if (s <= 0), otherwise s = s – 1

  `signal(semaphore s)`

  s = s + 1, (can have side effect of letting other processes proceed)

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

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



This is a precedence relationship, meaning that the $(i+K)^{th}$ call to wait cannot complete before the i^{th} call to signal completes.

Implementing 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 */
}
```

ISSUES:

- Granularity of lock
  - 1 lock for whole balance database?
  - 1 lock per account?
  - 1 lock for all accounts ending in 004?
- Implementation of wait() and signal() functions

Semaphores as Supervisor Call

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

Calling sequence:

```
...
; put address of lock
; into R0
CMOVE(lock, R0)
SVC(WAIT)
```

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

H/W 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

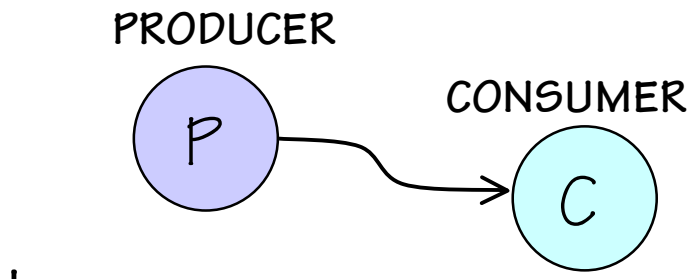
Executed ATOMICALLY (cannot be interrupted)

Can easily implement mutual exclusion using binary semaphore

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

More Advanced Example

PRODUCER-CONSUMER Problem:

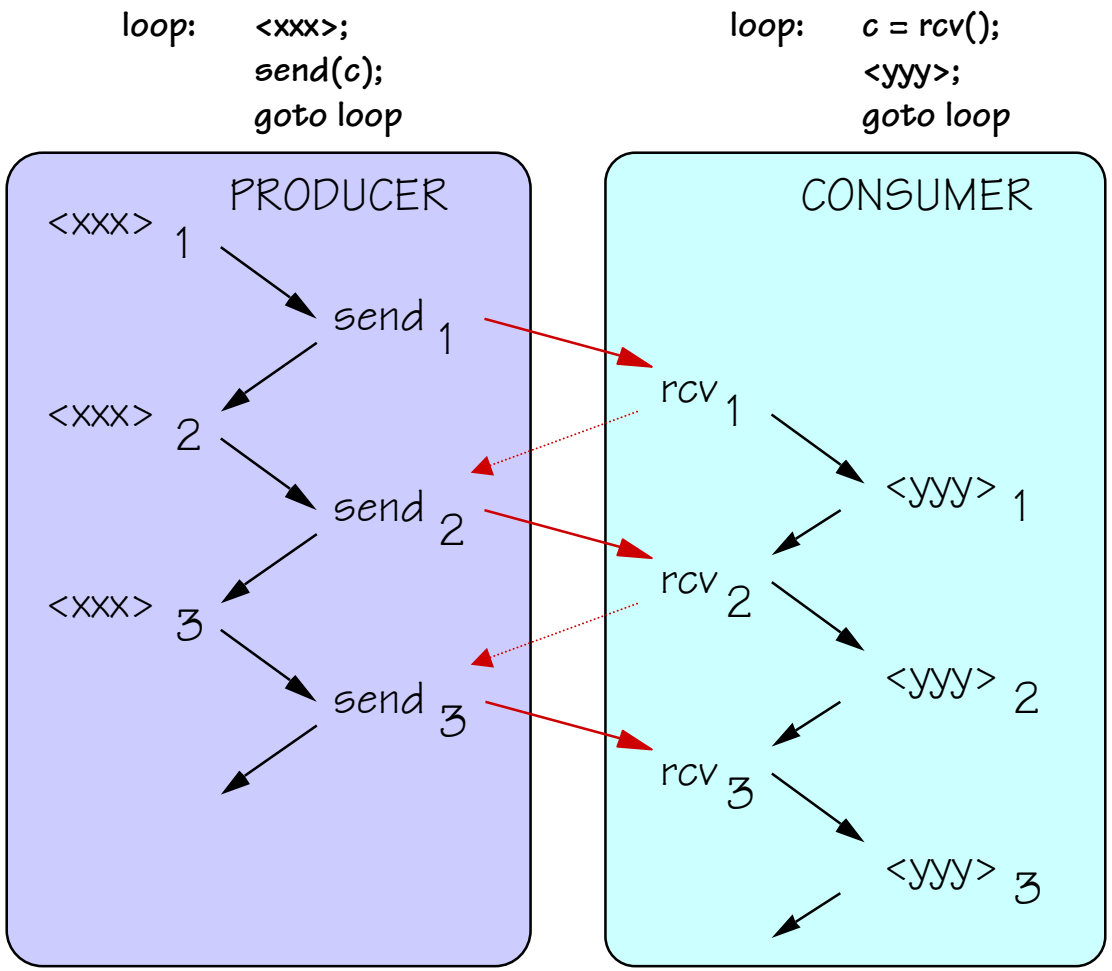



```
loop:  <xxx>;
       send(c);
       goto loop
```

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

Examples: UNIX pipeline, Word processor/Printer Driver, Preprocessor/Compiler, Compiler/Assembler

Synchrony of Communication



Precedence Constraints:

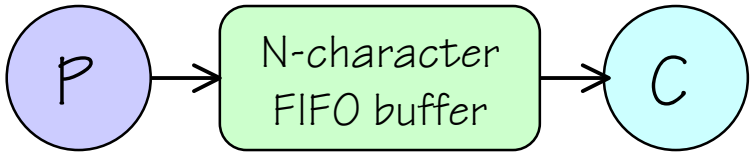
- Can't CONSUME data before it's PRODUCED

$$send_i \preceq rcv_i$$

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

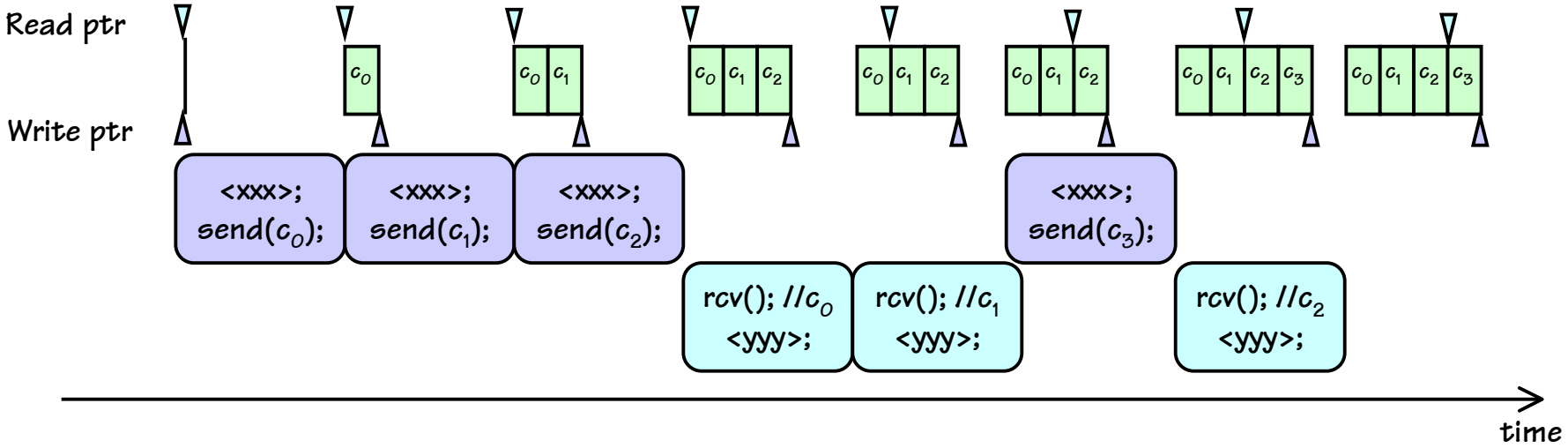
$$rcv_i \preceq send_{i+1}$$

FIFO Buffering



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

$rcv_i \preceq send_{i+N}$



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
(i.e. rcv( ) could be invoked prior to any send())

Using Semaphores for Resource Allocation

ABSTRACT PROBLEM:

- POOL of K resources
- Many processes, each needs resource for occasional uninterrupted periods
- MUST guarantee that at most K resources are in use at any time.

Semaphore Solution:

In shared memory:

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

In each process:

```
...
wait(s);     /* Allocate one       */
...          /* use it for a while */
signal(s);   /* return it to pool  */
...
```

Bounded Buffer Problem w/Semaphores

SHARED MEMORY:

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

PRODUCER:

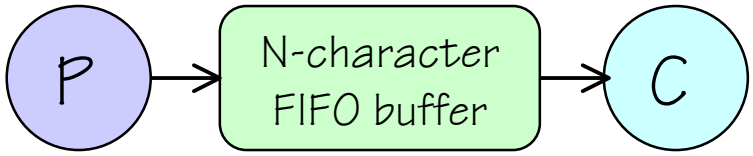
```
send(char c)
{
   buf[in] = c;
   in = (in+1)%N;
   signal(chars);
}
```

CONSUMER:

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

RESOURCE managed by semaphore: Characters in FIFO.
DOES IT WORK?

Flow Control Problems



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

A: Nothing.

Result: OVERFLOW. Randomness. Havoc. Smoke. Pain. Suffering.

WHAT we've got thus far:

$send_i \preceq rcv_i$

WHAT we still need:

$rcv_i \preceq send_{i+N}$

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);
}
```

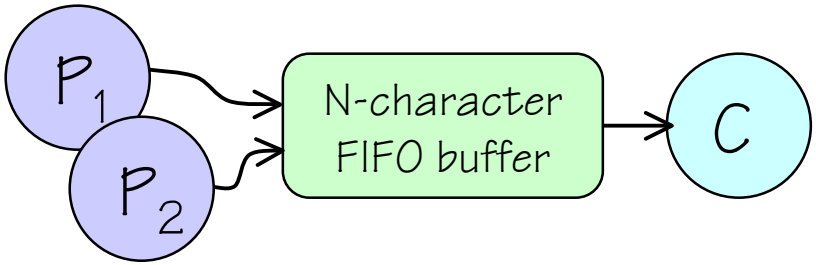
CONSUMER:

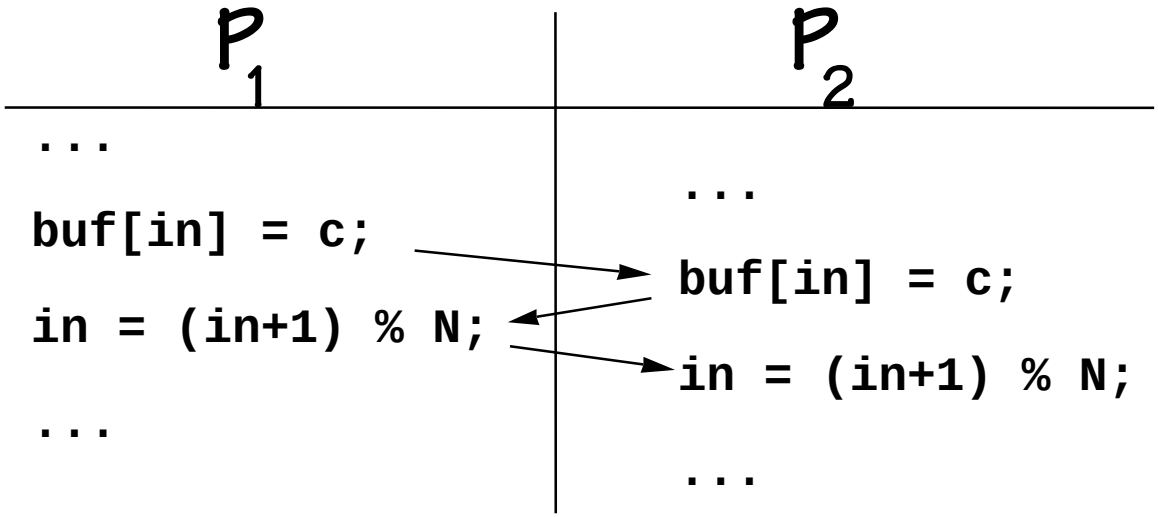
```
char rcv()
{
    char c;
    wait(chars);
    c = buf[out];
    out = (out+1)%N;
    signal(space);
    return c;
}
```

RESOURCEs managed by semaphore: Characters in FIFO, Spaces in FIFO

Atomicity Problems

Consider multiple PRODUCER processes:





BUG: Producers interfere with each other, MUTUAL EXCLUSION

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 mutex=1;
```

PRODUCER:

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

CONSUMER:

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

The Power of Semaphores

SHARED MEMORY:

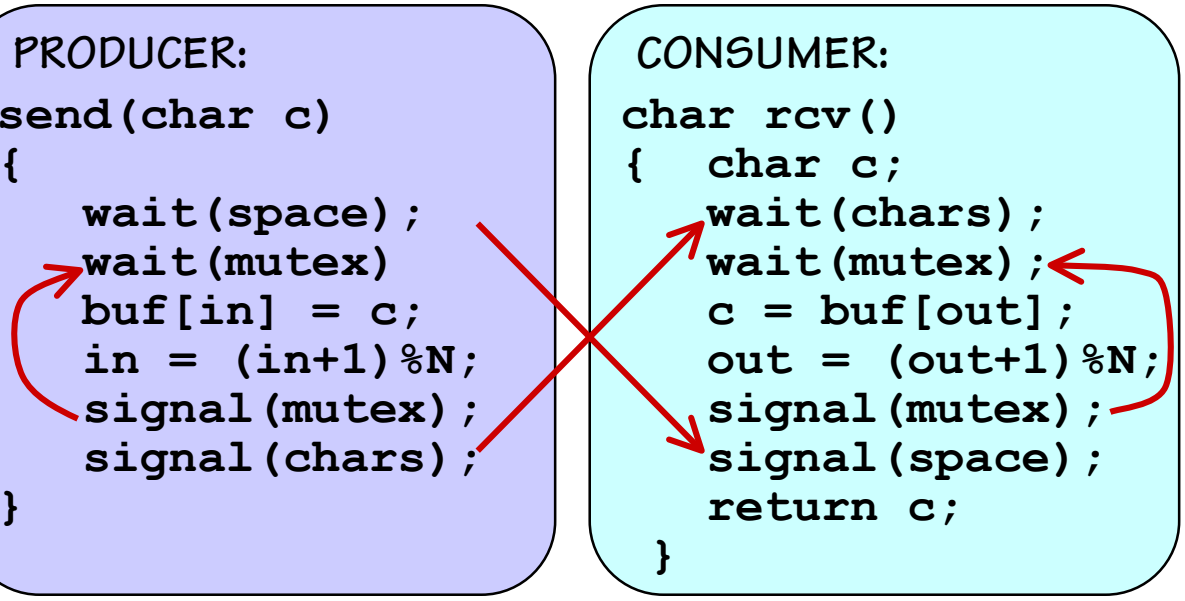
```
char buf[N];              /* The buffer */
int in=0, out=0;
semaphore chars=0, space=N;
semaphore mutex=1;
```

PRODUCER:

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

CONSUMER:

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



A single synchronization primitive that enforces both:

Precedence relationships:

$$send_i \preceq rcv_i$$

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

Mutual-exclusion primitives:

protect variables *in* and *out*

Problems with Mutual Exclusion

The indiscriminate use of mutual exclusion can introduce its own set of problems. Particularly when a process requires access to more than one protected resource.

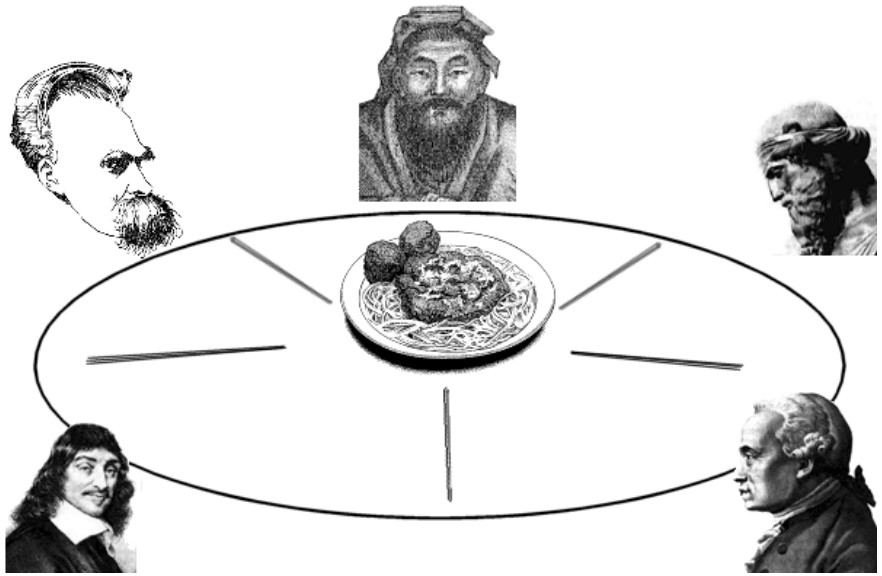
```
Transfer(int account1, int account2, int amount)
{
    wait(lock[account1]);
    wait(lock[account2]);
    balance[account1] = balance[account1] - amount;
    balance[account2] = balance[account2] + amount;
    signal(lock[account2]);
    signal(lock[account1]);
}
```

Transfer(6001, 6004, 50)

Transfer(6004, 6001, 50)

Toy Problem: Dining Philosophers

Philosophers do one of two things. They either think, or they eat. And, when they eat, they always eat spaghetti (with chopsticks no less). Unfortunately, when they are hungry, they are unable to think. Philosophers also obey a strict protocol when eating (albeit, and unsanitary one). This protocol is described in the algorithm below.

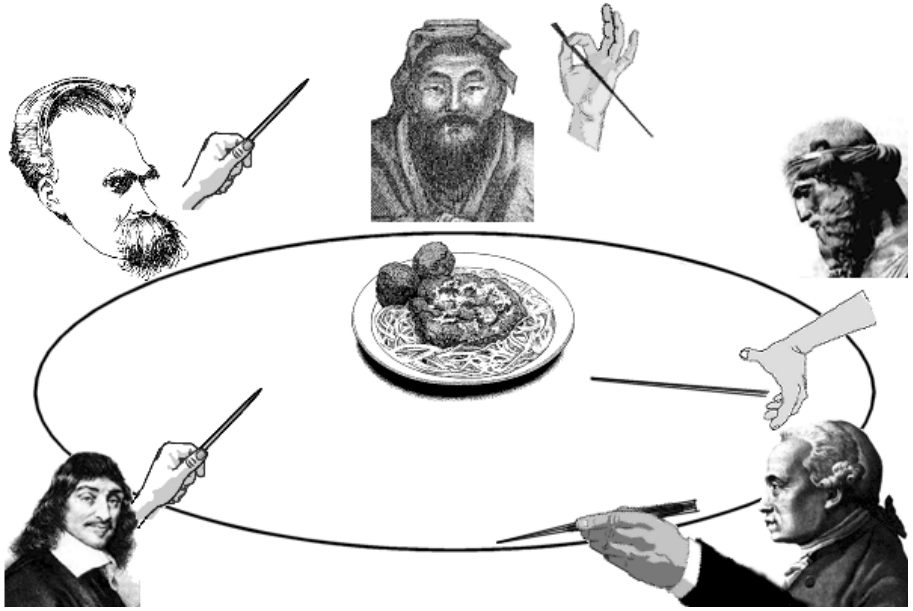


PHILOSOPHER'S ALGORITHM:

- Take LEFT stick
- Take RIGHT stick
- EAT
- 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

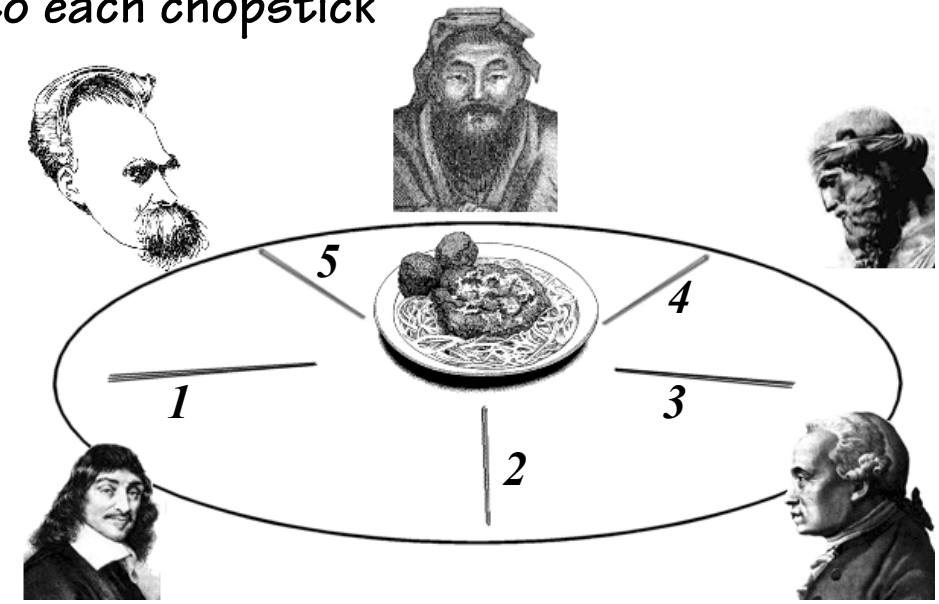
3) No preemption - a resource can not be removed from a process holding it

4) Circular Wait

SOLUTIONS: Avoidance -or- Detection and Recovery

One Solution

KEY: Assign a unique number to each chopstick



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) …

Thus, there can be no deadlock

Dealing with Deadlocks

Cooperating processes:

- Establish a fixed ordering to shared resources and require all locks are acquired according to it

```
Transfer(int account1, int account2, int amount)
{
    int a, b;
    if (account1 > account2) { a = account1; b = account2; } else {a = account2; b = account1; }
    wait(lock[a]);
    wait(lock[b]);
    balance[account1] = balance[account1] - amount;
    balance[account2] = balance[account2] + amount;
    signal(lock[b]);
    signal(lock[a]);
}
```

Independent processes:

- O/S discovers circular wait & kills waiting process
- Reserve all resources prior to process execution
- Hard problem