Semaphores

Processes, Synchronization, & Deadlock

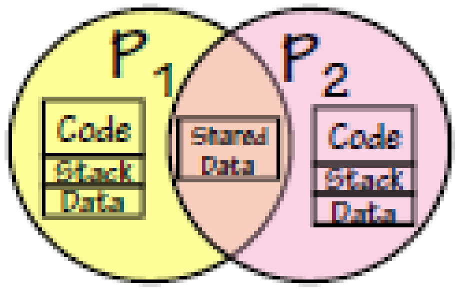
Interprocess Communication

Why communicate?

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

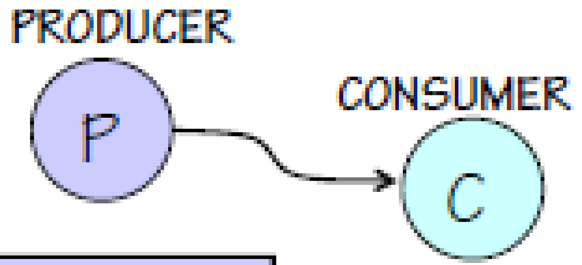
How to communicate?

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



Classic Example:

"PRODUCER-CONSUMER" Problem:



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

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

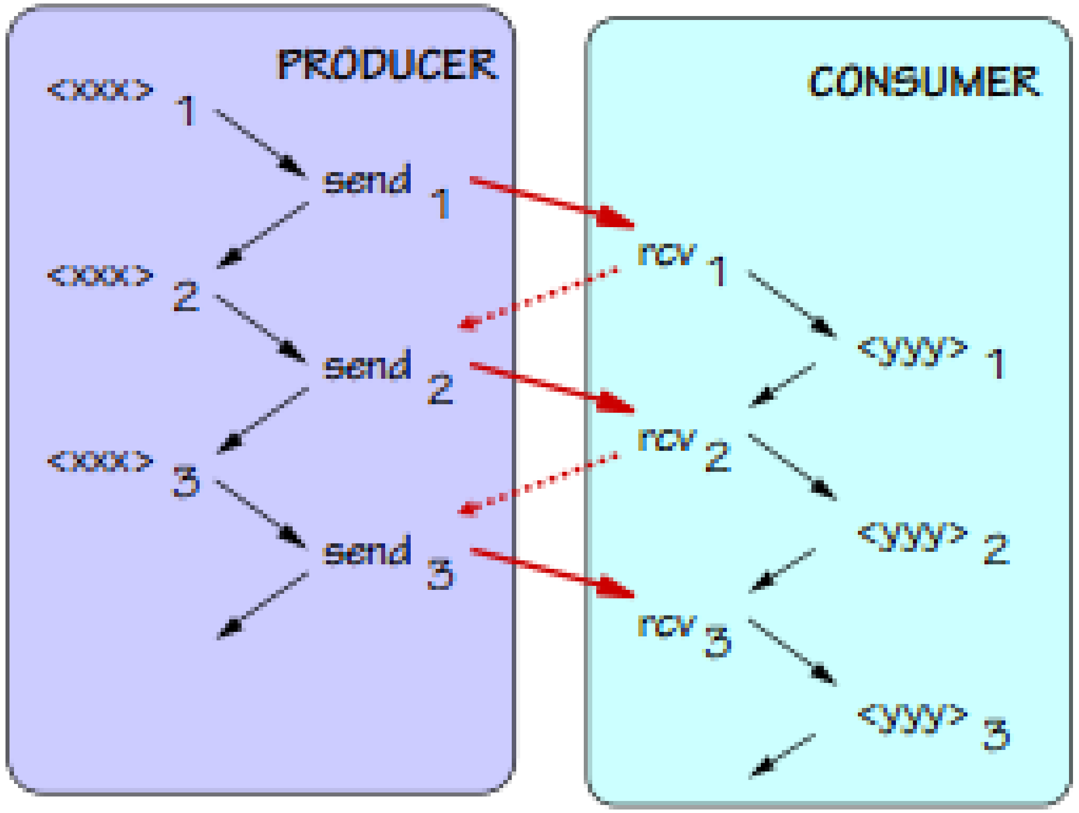
Real-World Examples:

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

Synchronous Communication

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

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



Precedence Constraints:

$$\boxed{\alpha \preceq \beta}$$

"**α** *precedes* **β**"

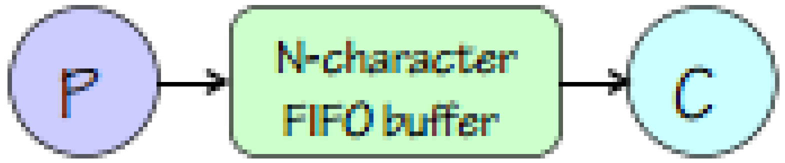
- Can't CONSUME data before it's PRODUCED

$$\boxed{send_i \preceq rcv_i}$$

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

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

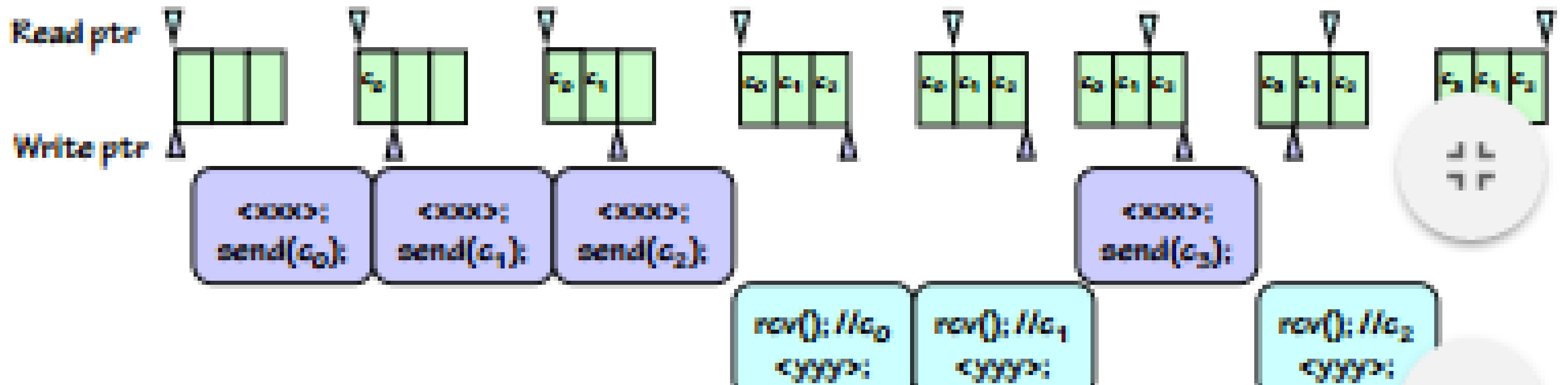
FIFO Buffering



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

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

"Ring 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 (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*:

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)

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

This is a *precedence* relationship,
meaning that the (i+K)th
call to wait
cannot proceed before
the
ith call to
signal completes.

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:

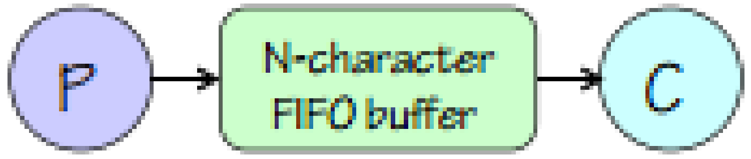
```
  semaphore s = K;  /* K resources  */
```

In each process:

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

Invariant: Semaphore value = number of resources left in pool

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. D's and F's in MIT courses.

WHAT we've *got* thus far:

$$\boxed{send_i \preceq rcv_i}$$

WHAT we still *need:*

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

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:

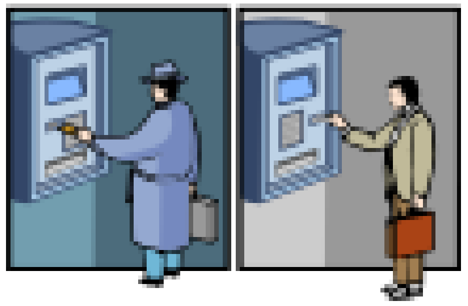
```
char rcv()
{
   char c;
   wait(chars);
   c = buf[out];
   out = (out+1)%N;
   signal(space);
   return c;
}
```

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

WORKS with **single** producer, consumer. But what about…

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)

```
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 assure 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 */
}
```

$\alpha \divideontimes \beta$

"α *precedes* β
or
β *precedes* α"
(*i.e., they don't overlap*)

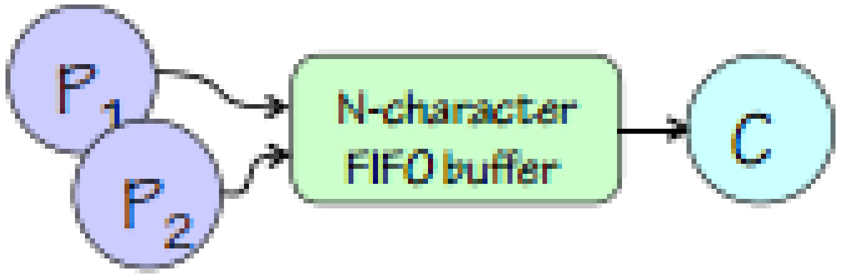
RESOURCE managed by "lock" semaphore: Access to critical section

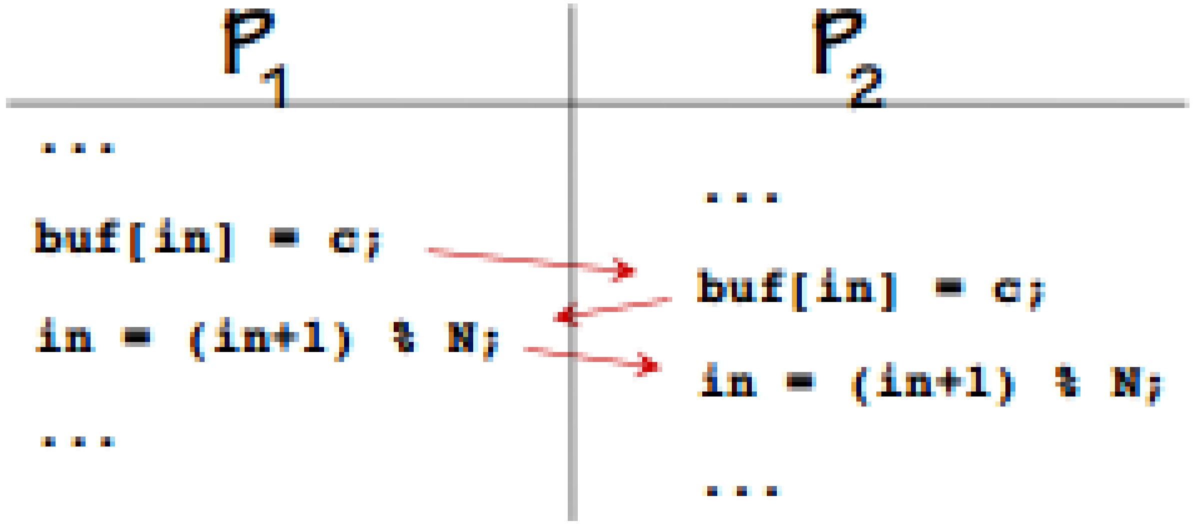
ISSUES:

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

Producer/Consumer Atomicity Problems

Consider multiple PRODUCER processes:





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

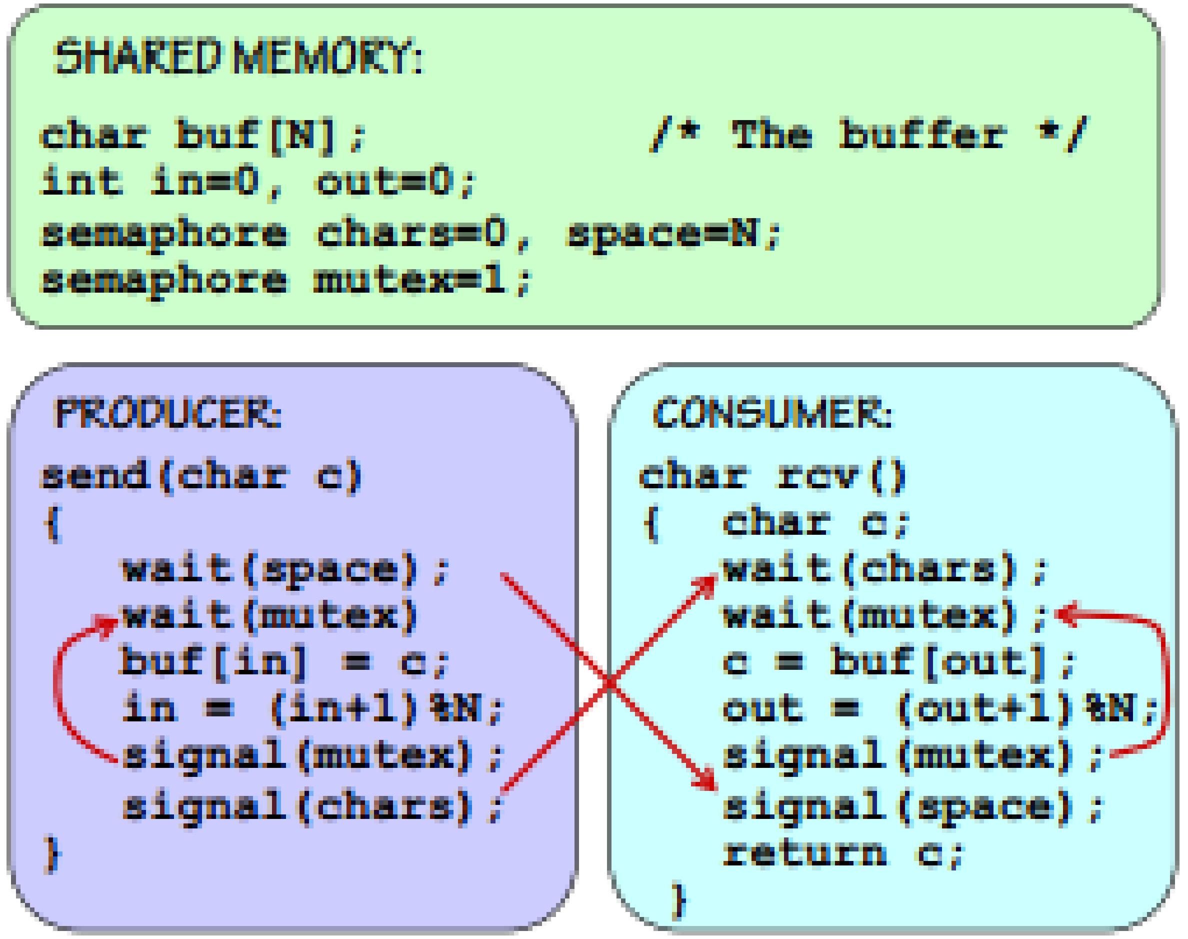
PRODUCER:

```
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:

$$\begin{aligned} send_i &\preceq rcv_i \\ rcv_i &\preceq send_{i+N} \end{aligned}$$

Mutual-exclusion relationships:

protect variables *in* and *out*

Semaphore Implementations

Semaphore implementation must address a basic arbitration problem: how to choose among simultaneously waiting processes when a signal occurs. This involves some basic atomicity assumption in the implementation technology.

Approaches:

- SVC implementation, using atomicity of kernel handlers. Works in timeshared processor sharing a single uninterruptable kernel.
- Implementation by a special instruction (e.g. "test and set"), using atomicity of single instruction execution. Works with shared-bus multiprocessors supporting atomic read-modify-write bus transactions.
- Implementation using atomicity of individual read or write operations. Complex, clever, 2-phase scheme devised by Dijkstra. Unused in practice.

Bootstrapping: A simple lock ("binary semaphore") allows easy implementation of full semaphore support.

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
```

Atomicity Guaranteed, e.g. by Bus protocols

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)
```

Synchronization: The Dark Side

The indiscriminate 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]);
}
```

DEAD-
LOCK!

Transfer(6001, 6004, 50)

Transfer(6004, 6001, 50)

Famous Toy Problem:

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 (2 sticks are required, to avoid an ancient Zen paradox about the sound of one stick feeding).

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

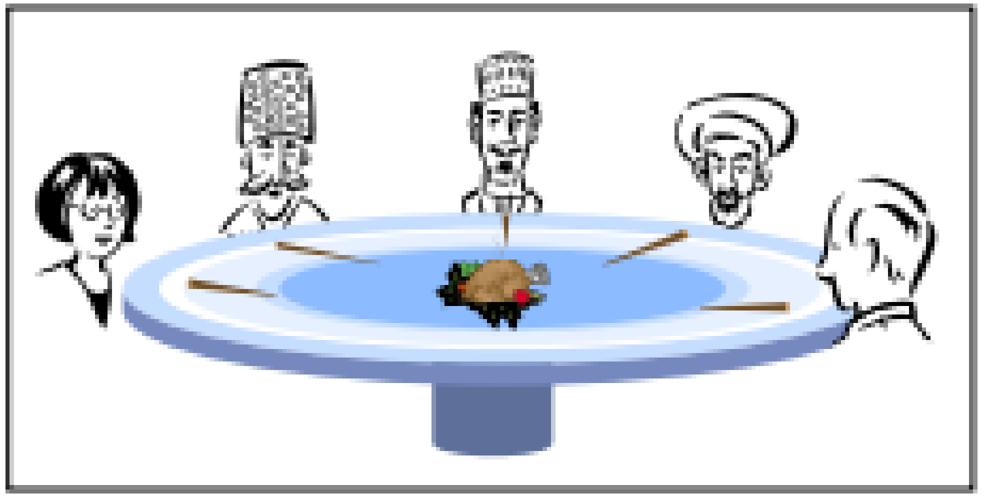


Figure by MIT OpenCourseWare.

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

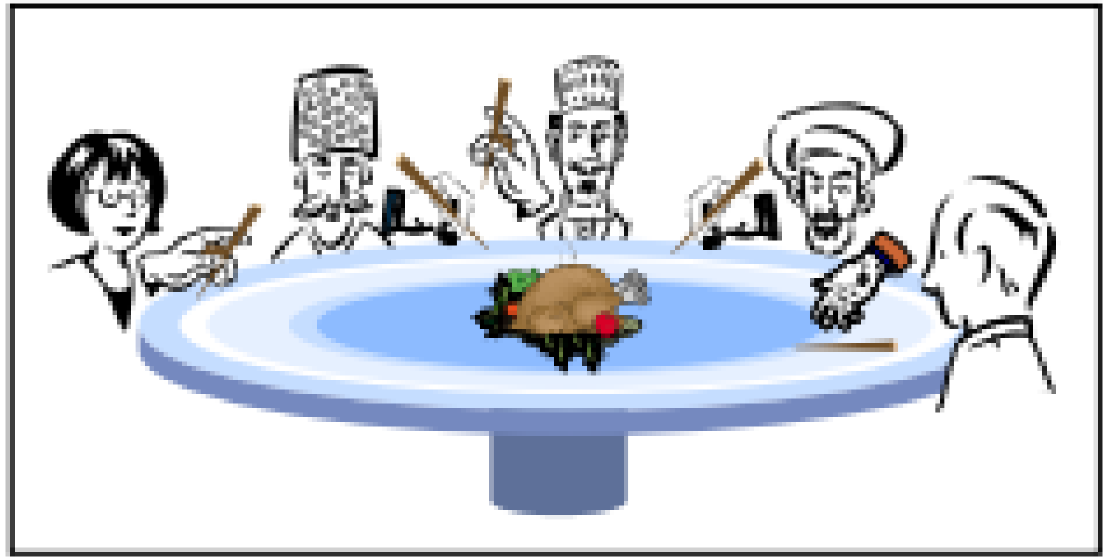


Figure by MIT OpenCourseWare.

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.

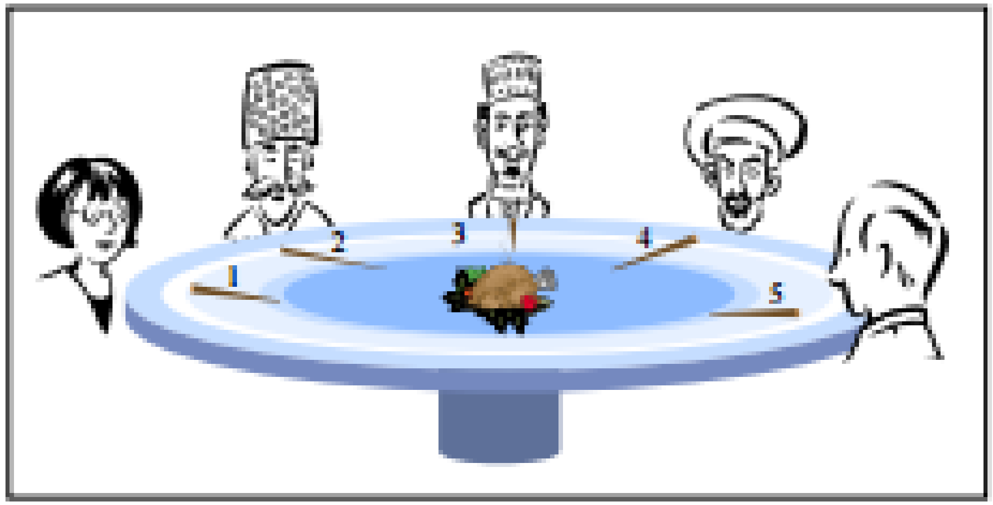


Figure by MIT OpenCourseWare.

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*

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, 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]);
}
```

Transfer(6001, 6004, 50) Transfer(6004, 6001, 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.*