# Operating Systems
## Synchronization

**Yanyan Zhuang**

Department of Computer Science

http://www.cs.uccs.edu/~yzhuang

# Inter-Process Communication (IPC)

- Fundamental issues:
  - How to make sure two or more processes do not get in each other's way when engaging in critical activities
  - How to maintain proper sequencing when dependencies present

# Inter-Process Communication (IPC)

- Fundamental issues:
  - How to make sure two or more processes do not get in each other's way when engaging in critical activities
  - How to maintain proper sequencing when dependencies present

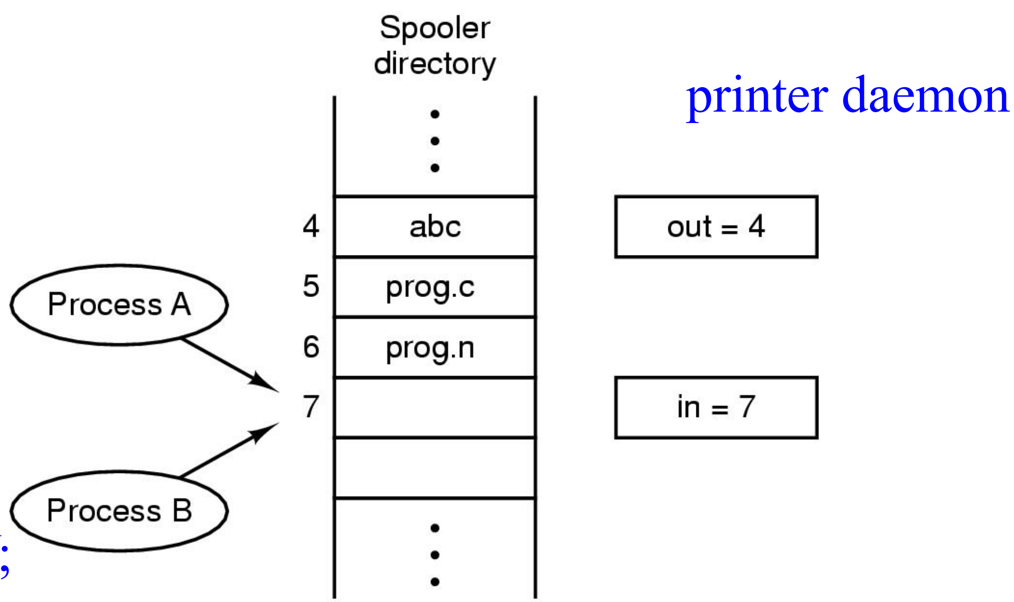
Inter-thread communication?

Same problems exist & same solutions apply

# Race Conditions

- Race conditions: when two or more processes/threads are reading or writing some *shared data* and the final results depend on who runs precisely when
  - Interrupts, interleaved operations/execution

```
free_slot = in;
Dir[free_slot] = X;
in ++;
```



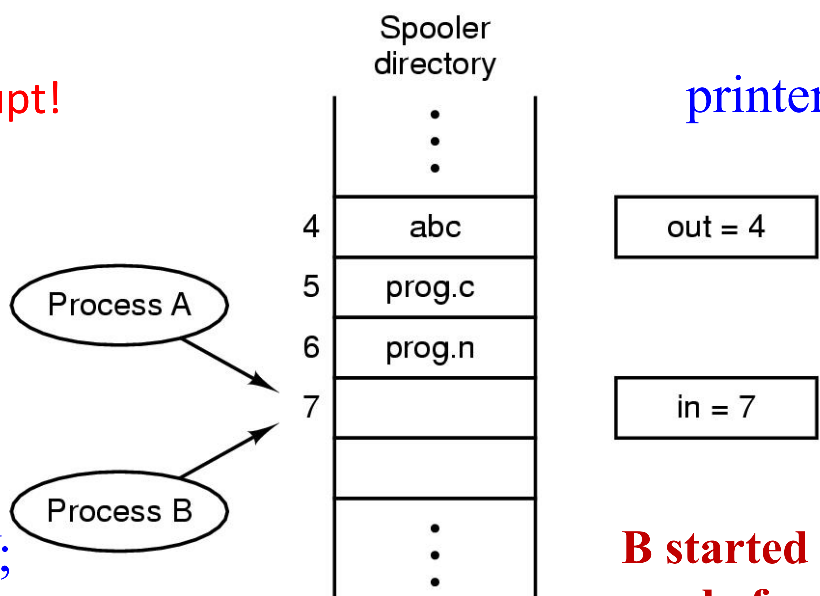

```
free_slot = in;
Dir[free_slot] = Y;
in ++;
```

# Race Conditions

- Race conditions: when two or more processes/threads are reading or writing some *shared data* and the final results depend on who runs precisely when
  - Interrupts, interleaved operations/execution

Clock interrupt!

```
free_slot = in;
Dir[free_slot] = X;
in ++;
```




printer daemon

```
free_slot = in;
Dir[free_slot] = Y;
in ++;
```

**B started using a shared var before A was finished**

# Mutual Exclusion and Critical Regions

- Mutual exclusion: makes sure if one process is using a shared variable or file, the other processes will be excluded from doing the same thing
  - Main challenge/issue to OS: to design appropriate primitive operations for achieving mutual exclusion in user space

- Critical regions: the part of the program where the shared resource is accessed

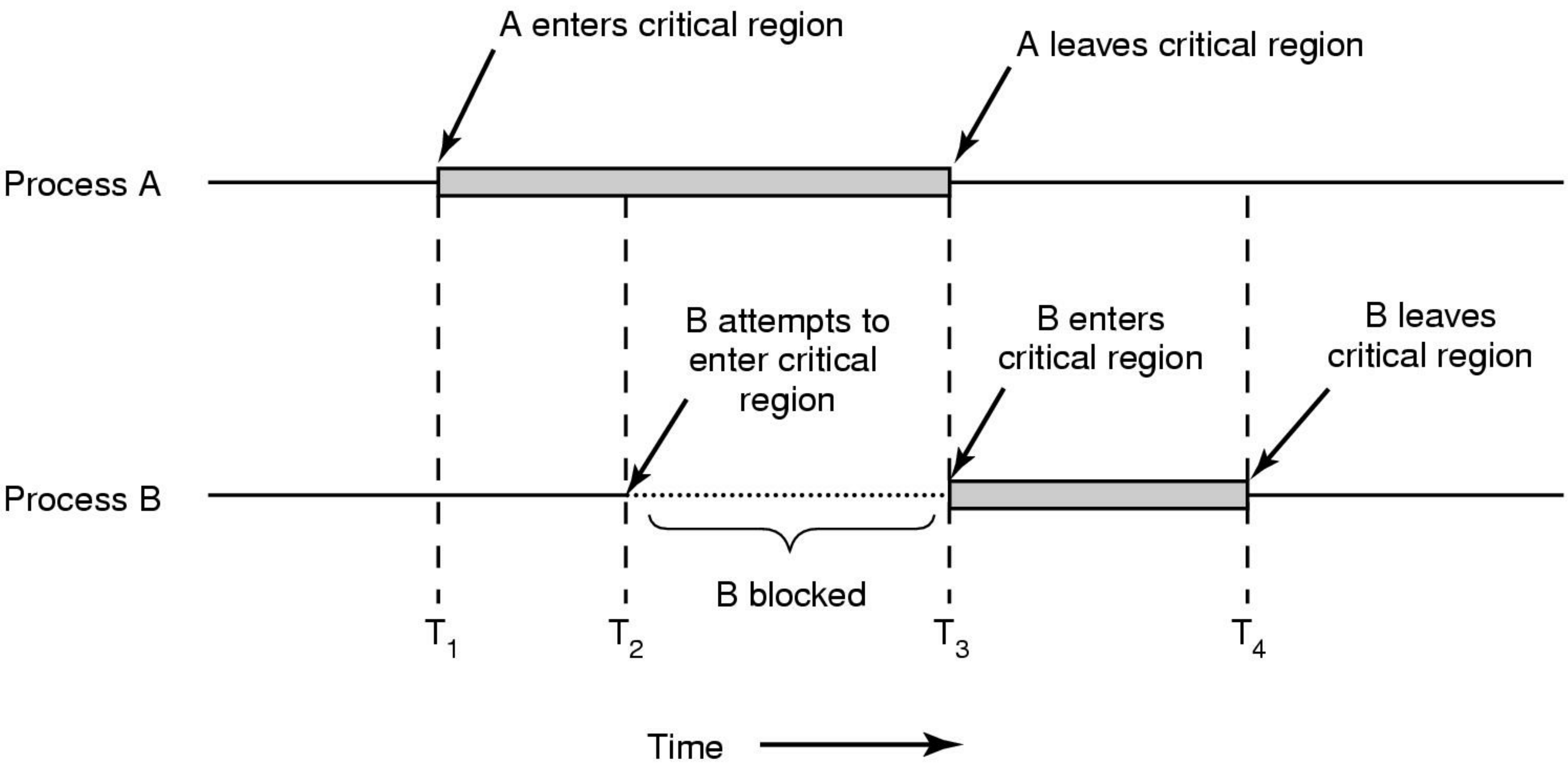
# Mutual Exclusion and Critical Regions

- Mutual exclusion: makes sure if one process is using a shared variable or file, the other processes will be excluded from doing the same thing
  - Main challenge/issue to OS: to design appropriate primitive operations for achieving mutual exclusion in user space

- Critical regions: the part of the program where the shared resource is accessed

- Four conditions to provide mutual exclusion
  - No two processes simultaneously in critical region
  - No assumptions made about speeds or numbers of CPUs
  - No process running outside its critical region may block another process
  - No process must wait forever to enter its critical region

# Mutual Exclusion Using Critical Regions




Mutual exclusion using critical regions

# Mutual Exclusion with Busy Waiting

- Disabling interrupts:
  - OS technique, not users'
  - Multi-CPU?
- Lock variables:
  - Test-set is a two-step process, not atomic
- Busy waiting:
  - Continuously testing a variable until some value appears (*spin lock*)

# Busy Waiting: Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Proposed *strict alternation* solution to critical region problem

(a) Process 0. (b) Process 1.

# Busy Waiting: Strict Alternation

```
while (TRUE) {
    while (turn != 0) ;        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1) ;        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Proposed *strict alternation* solution to critical region problem

(a) Process 0. (b) Process 1.

# Busy Waiting: Strict Alternation

```
while (TRUE) {
    while (turn != 0) ;        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1) ;        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Proposed *strict alternation* solution to critical region problem

(a) Process 0. (b) Process 1.

What if P1's noncritical_region() has lots more work than P0's?

Taking turns is not a good idea when one is much slower than the other

# Busy Waiting: TSL

- TSL (Test and Set Lock)
  - Indivisible (atomic) operation, how? Hardware (multi-processor)
  - How to use TSL to prevent two processes from simultaneously entering their critical regions? 0 can enter; 1 can't enter

```
enter_region:
    TSL REGISTER,LOCK         | copy lock to register and set lock to 1
    CMP REGISTER,#0           | was lock zero?
    JNE enter_region          | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered


leave_region:
    MOVE LOCK,#0              | store a 0 in lock
    RET | return to caller
```

Entering and leaving a critical region using the TSL instruction

# Busy Waiting: Pros and Cons

- Cons
  - Wastes CPU cycles
  - For single-core system, user-level threads
    - T1 waiting for T2 to change lock to 0, but T2 never gets to run when T1 is running
    - Priority inversion
      - 2 processes on 1 CPU. Process H with high priority and L with low priority, L is in its critical region and H becomes ready; *does L have chance to leave critical region*?
- Pros
  - Avoids expensive context switch when critical region is *very short*
    - Sleep/wakeup (an alternative to busy waiting) requires context switch
      - Time to put a thread to sleep/wake up might exceed the time a thread has actually slept

# Sleep and Wakeup

- Issue I: avoid CPU-costly busy waiting
- Issue II: avoid priority inversion problem
- Some IPC primitives that **block** instead of wasting CPU time when they are not allowed to enter their critical regions
  - Sleep and wakeup

# Sleep and Wakeup – Producer-Consumer Problem

```
#define N 100                               /* number of slots in the buffer */
int count = 0;                              /* number of items in the buffer */

void producer(void)
{
      int item;

      while (TRUE) {                        /* repeat forever */
            item = produce_item( );         /* generate next item */
            if (count == N) sleep( );       /* if buffer is full, go to sleep */
            insert_item(item);              /* put item in buffer */
            count = count + 1;              /* increment count of items in buffer */
            if (count == 1) wakeup(consumer);   /* was buffer empty? */
      }
}


void consumer(void)
{
      int item;

      while (TRUE) {                        /* repeat forever */
            if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
            item = remove_item( );          /* take item out of buffer */
            count = count − 1;              /* decrement count of items in buffer */
            if (count == N − 1) wakeup(producer); /* was buffer full? */
            consume_item(item);             /* print item */
      }
}
```

# Sleep and Wakeup – Producer-Consumer Problem

```
#define N 100                               /* number of slots in the buffer */
int count = 0;                              /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                         /* repeat forever */
          item = produce_item( );           /* generate next item */
          if (count == N) sleep( );         /* if buffer is full, go to sleep */
          insert_item(item);                /* put item in buffer */
          count = count + 1;                /* increment count of items in buffer */
          if (count == 1) wakeup(consumer); /* was buffer empty? */
     }
}
```

Producer: buffer full → sleep, wakeup when item removed

Consumer: buffer empty → sleep, wakeup when item inserted

```
void consumer(void)
{
     int item;

     while (TRUE) {                         /* repeat forever */
          if (count == 0) sleep( );         /* if buffer is empty, got to sleep */
          item = remove_item( );            /* take item out of buffer */
          count = count − 1;                /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);               /* print item */
     }
}
```

# Sleep and Wakeup – Producer-Consumer Problem

```
#define N 100                           /* number of slots in the buffer */
int count = 0;                          /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                      /* repeat forever */
        item = produce_item( );         /* generate next item */
        if (count == N) sleep( );       /* if buffer is full, go to sleep */
        insert_item(item);              /* put item in buffer */
        count = count + 1;              /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);   /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                      /* repeat forever */
        if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
        item = remove_item( );          /* take item out of buffer */
        count = count − 1;              /* decrement count of items in buffer */
        if (count == N − 1) wakeup(producer); /* was buffer full? */
        consume_item(item);             /* print item */
    }
}
```

Q1: What if the wakeup signal sent to a non-sleep process?

Interrupt!

# Sleep and Wakeup – Producer-Consumer Problem

```
#define N 100                                  /* number of slots in the buffer */
int count = 0;                                 /* number of items in the buffer */

void producer(void)
{
      int item;

      while (TRUE) {                           /* repeat forever */
            item = produce_item( );            /* generate next item */
            if (count == N) sleep( );          /* if buffer is full, go to sleep */
            insert_item(item);                 /* put item in buffer */
            count = count + 1;                 /* increment count of items in buffer */
            if (count == 1) wakeup(consumer);  /* was buffer empty? */
      }
}

void consumer(void)
{
      int item;

      while (TRUE) {                           /* repeat forever */
            if (count == 0) sleep( );          /* if buffer is empty, got to sleep */
            item = remove_item( );             /* take item out of buffer */
            count = count − 1;                 /* decrement count of items in buffer */
            if (count == N − 1) wakeup(producer); /* was buffer full? */
            consume_item(item);                /* print item */
      }
}
```

Q2: wakeup waiting bit
wakeup() sent to a process that is still awake, wakeup waiting bit is set

# Semaphores and P&V Operations

- Semaphores: a variable to indicate the # of pending wakeups (Dijkstra)
- *Down* operation (request): lock/sleep
  - Checks if a semaphore is > 0,
    - if so, it decrements the value and just continue
    - Otherwise (==0), the process is put to sleep

# Semaphores and P&V Operations

- Semaphores: a variable to indicate the # of pending wakeups (Dijkstra)
- *Down* operation (request): lock/sleep
  - Checks if a semaphore is > 0,
    - if so, it decrements the value and just continue
    - Otherwise (==0), the process is put to sleep
- *Up* operation (release): unlock/wakeup
  - Increments the value of the semaphore
    - But if one or more processes are sleeping on the semaphore (==0), one of them is chosen (randomly) and wake up (semaphore will still be 0)

# Semaphores and P&V Operations

- Semaphores: a variable to indicate the # of pending wakeups (Dijkstra)
- *Down* operation (request): lock/sleep
  - Checks if a semaphore is > 0,
    - if so, it decrements the value and just continue
    - Otherwise (==0), the process is put to sleep
- *Up* operation (release): unlock/wakeup
  - Increments the value of the semaphore
    - But if one or more processes are sleeping on the semaphore (==0), one of them is chosen (randomly) and wake up (semaphore will still be 0)
- P & V operations are **atomic**, how to implement?
  - Single CPU: system calls, OS disabling interrupts temporarily
  - Multiple CPUs: TSL

# The Producer-consumer Problem w/ Semaphores

```
#define N 100                        /* number of slots in the buffer */
typedef int semaphore;               /* semaphores are a special kind of int */
semaphore mutex = 1;                 /* controls access to critical region */
semaphore empty = N;                 /* counts empty buffer slots */
semaphore full = 0;                  /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                   /* TRUE is the constant 1 */
        item = produce_item( );      /* generate something to put in buffer */
        down(&empty);                /* decrement empty count */
        down(&mutex);                /* enter critical region */
        insert_item(item);           /* put new item in buffer */
        up(&mutex);                  /* leave critical region */
        up(&full);                   /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                   /* infinite loop */
        down(&full);                 /* decrement full count */
        down(&mutex);                /* enter critical region */
        item = remove_item( );       /* take item from buffer */
        up(&mutex);                  /* leave critical region */
        up(&empty);                  /* increment count of empty slots */
        consume_item(item);          /* do something with the item */
    }
}
```

For mutual exclusion and synchronization

Binary semaphores: if each process does a down before entering its critical region and an up just leaving it, mutual exclusion is achieved

# The Producer-consumer Problem w/ Semaphores

```
#define N 100                              /* number of slots in the buffer */
typedef int semaphore;                     /* semaphores are a special kind of int */
semaphore mutex = 1;                       /* controls access to critical region */
semaphore empty = N;                       /* counts empty buffer slots */
semaphore full = 0;                        /* counts full buffer slots */

void producer(void)
{
      int item;

      while (TRUE) {                       /* TRUE is the constant 1 */
            item = produce_item( );        /* generate something to put in buffer */
            down(&empty);                  /* decrement empty count */
            down(&mutex);                  /* enter critical region */
            insert_item(item);             /* put new item in buffer */
            up(&mutex);                    /* leave critical region */
            up(&full);                     /* increment count of full slots */
      }
}

void consumer(void)
{
      int item;

      while (TRUE) {                       /* infinite loop */
            down(&full);                   /* decrement full count */
            down(&mutex);                  /* enter critical region */
            item = remove_item( );         /* take item from buffer */
            up(&mutex);                    /* leave critical region */
            up(&empty);                    /* increment count of empty slots */
            consume_item(item);            /* do something with the item */
      }
}
```

For mutual exclusion and synchronization

P: down(&empty) → empty = N → empty = N-1

Binary semaphores: if each process does a down before entering its critical region and an up just leaving it, mutual exclusion is achieved

C: down(&full) → full=0 → sleep on full

# The Producer-consumer Problem w/ Semaphores

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

For mutual exclusion and synchronization

P: down(&empty) → empty = N → empty = N-1

P: up(&full) → C sleeps on full → wakeup C, full=0

Binary semaphores: if each process does a down before entering its critical region and an up just leaving it, mutual exclusion is achieved

C: down(&full) → full=0 → sleep on full

C: up(&empty) → empty = N-1 → empty = N

# Mutexes

- Mutex:
  - A variable that can be in one of two states: unlocked or locked
  - A simplified version of the semaphores [0, 1]

```
mutex_lock:
      TSL REGISTER,MUTEX              | copy mutex to register and set mutex to 1
      CMP REGISTER,#0                 | was mutex zero?
      JZE ok                          | if it was zero, mutex was unlocked, so return
      CALL thread_yield               | mutex is busy; schedule another thread
      JMP mutex_lock                  | try again later
ok:   RET | return to caller; critical region entered

mutex_unlock:
      MOVE MUTEX,#0                   | store a 0 in mutex
      RET | return to caller
```

**Give others chance to run;**
**What about mutex_trylock()? Either gets the lock or returns a code for failure → does not block**

# Mutexes – User-space Multi-threading

- What is a key difference between *mutex_lock* and *enter_region* in multi-threading and multi-processing?
  - For single-core, user-space multi-threading, a thread has to allow other thread to run and release the lock so as to enter its critical region, which is impossible with busy waiting *enter_region*

```
enter_region:
    TSL REGISTER,LOCK           | copy lock to register and set lock to 1
    CMP REGISTER,#0             | was lock zero?
    JNE enter_region            | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered


leave_region:
    MOVE LOCK,#0                | store a 0 in lock
    RET | return to caller
```

Two *processes* entering and leaving a critical region using the TSL instruction

# When to Use Busy-waiting/Sleep-wakeup

- On a single-core system
  - Using busy-waiting makes no sense
    - No other thread can run, locks won't be unlocked
  - Use sleep-wakeup instead
- On a multi-core systems, (only when) locks are held for a very short time
  - Time for putting threads to sleep/waking up might decrease runtime performance
  - Use busy-waiting instead

# Semaphores/Mutexes easy?

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N; 0                  /* counts empty buffer slots */
semaphore full = 0; N                   /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Swap the order of mutex and empty

If the buffer is **full**

Must be careful using semaphores!

# Monitors (1)

- Monitor: a higher-level synchronization primitive
  - Only **one process can be active in a monitor** at any instant, *with compiler's help*; thus, how about putting all the critical regions into monitor procedures for *mutual exclusion*?

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
    end;

    procedure consumer( );
    .
    .
    .
    end;
end monitor;
```

**Processes can call procedures in a monitor, but not directly access monitor's internal data**

Programming-language construct: compiler knows they are special and handle calls differently

# Monitors (1)

- Monitor: a higher-level synchronization primitive
  - Only **one process can be active in a monitor** at any instant, *with compiler's help*; thus, how about putting all the critical regions into monitor procedures for *mutual exclusion*?

```
monitor example
      integer i;
      condition c;

      procedure producer( );
      .
      .
      .
      end;

      procedure consumer( );
      .
      .
      .
      end;
end monitor;
```

**But, how processes block when they cannot proceed?**

**Condition variables, and two operations: *wait()* and *signal()***

# Monitors (2)

Wakeup and sleep signals can be lost, but not wait and signal, why?

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

# Monitors (2)

Wakeup and sleep signals can be lost, but not wait and signal, why?

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;
    count := 0;
end monitor;
```

Consumer not allowed in

Producer not allowed in

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Mutual exclusion guarantees e.g., producer will be able to complete wait() without switching to consumer before wait() completes

# Monitors (3)

- Pros
  - Make mutual exclusion automatic
  - Make parallel programming less error-prone
- Cons
  - Compiler support
    - Keyword **synchronized** in Java

# All previous methods…

- Work on a single computer
  - Assuming data can be shared in some way
- Data exchange between machines?
  - Message passing
    - Inter-process communication using two primitives, send/receive
    - send(destination, &message);
    - receive(source, &message);
    - System calls rather than language constructs

# Message Passing

```
#define N 100                          /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                         /* message buffer */

    while (TRUE) {
        item = produce_item( );        /* generate something to put in buffer */
        receive(consumer, &m);         /* wait for an empty to arrive */
        build_message(&m, item);       /* construct a message to send */
        send(consumer, &m);            /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);         /* get message containing item */
        item = extract_item(&m);       /* extract item from message */
        send(producer, &m);            /* send back empty reply */
        consume_item(item);            /* do something with the item */
    }
}
```

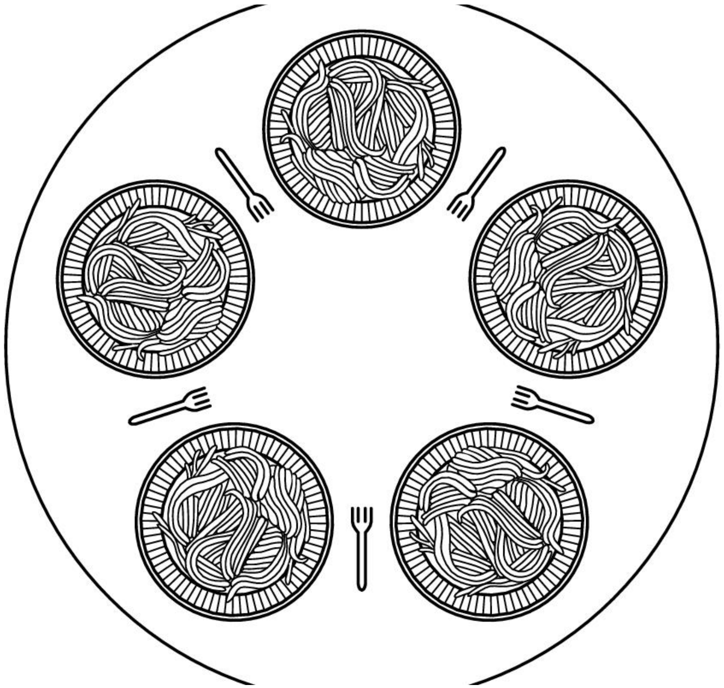
Communication without sharing memory:
Information exchange between machines

Whenever producer has an item for consumer,
it takes an empty msg&sends back a full one

The producer-consumer problem with N messages

# Class IPC Problems: Dining Philosophers

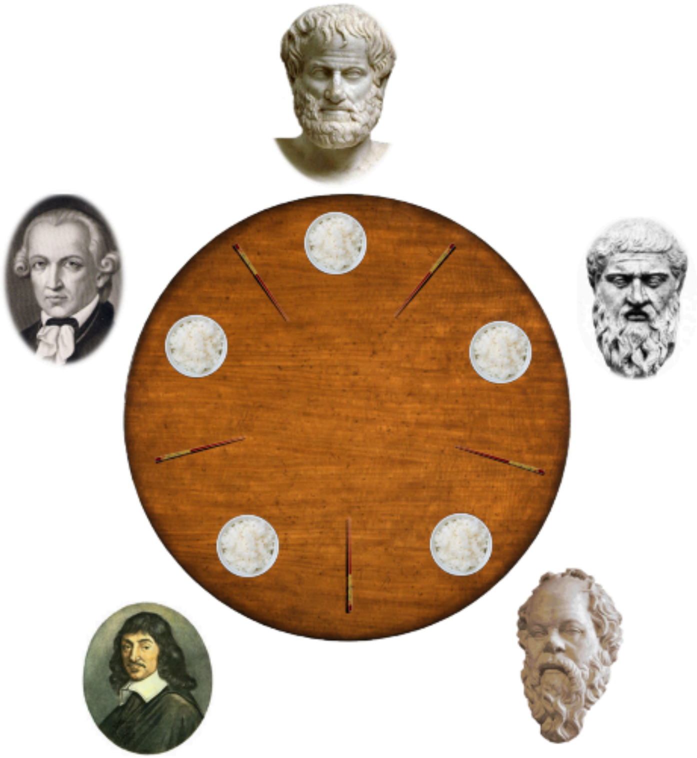
- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock & starvation
  - *Deadlock*: processes are blocked on some resource
  - *Starvation*: waiting for scheduler, no progress can be made



**The problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices**

# Class IPC Problems: Dining Philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock & starvation
  - *Deadlock*: processes are blocked on some resource
  - *Starvation*: waiting for scheduler, no progress can be made



**The problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices**

# Dining Philosophers (2)

```
#define N 5                              /* number of philosophers */

void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
      while (TRUE) {
            think( );                    /* philosopher is thinking */
            take_fork(i);                /* take left fork */
            take_fork((i+1) % N);        /* take right fork; % is modulo operator */
            eat( );                      /* yum-yum, spaghetti */
            put_fork(i);                 /* put left fork back on the table */
            put_fork((i+1) % N);         /* put right fork back on the table */
      }
}
```

# Dining Philosophers (2)

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                    /* philosopher is thinking */
        take_fork(i);                /* take left fork */
        take_fork((i+1) % N);        /* take right fork; % is modulo operator */
        eat( );                      /* yum-yum, spaghetti */
        put_fork(i);                 /* put left fork back on the table */
        put_fork((i+1) % N);         /* put right fork back on the table */
    }
}
```

**A non-solution to the dining philosophers problem**

What happens if all philosophers pick up their left forks simultaneously?

# Dining Philosophers (2)

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                    /* philosopher is thinking */
        take_fork(i);                /* take left fork */
        take_fork((i+1) % N);        /* take right fork; % is modulo operator */
        eat( );                      /* yum-yum, spaghetti */
        put_fork(i);                 /* put left fork back on the table */
        put_fork((i+1) % N);         /* put right fork back on the table */
    }
}
```

**A non-solution to the dining philosophers problem**

What happens if all philosophers pick up their left forks simultaneously?

# Dining Philosophers (2)

```
#define N 5                             /* number of philosophers */

void philosopher(int i)                 /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                       /* philosopher is thinking */
        take_fork(i);                   /* take left fork */
        take_fork((i+1) % N);           /* take right fork; % is modulo operator */
        eat( );                         /* yum-yum, spaghetti */
        put_fork(i);                    /* put left fork back on the table */
        put_fork((i+1) % N);            /* put right fork back on the table */
    }
}
```

**A <u>non-</u>solution to the dining philosophers problem**

What if *down* and *up* on *mutex* before acquiring/replacing a fork?
Only one can be eating at any instant…

# Dining Philosophers (3): Solution part1

```
#define N           5          /* number of philosophers */
#define LEFT        (i+N−1)%N  /* number of i's left neighbor */
#define RIGHT       (i+1)%N    /* number of i's right neighbor */
#define THINKING    0          /* philosopher is thinking */
#define HUNGRY      1          /* philosopher is trying to get forks */
#define EATING      2          /* philosopher is eating */
typedef int semaphore;         /* semaphores are a special kind of int */
int state[N];                  /* array to keep track of everyone's state */
semaphore mutex = 1;           /* mutual exclusion for critical regions */
semaphore s[N];                /* one semaphore per philosopher */

void philosopher(int i)        /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {             /* repeat forever */
        think( );              /* philosopher is thinking */
        take_forks(i);         /* acquire two forks or block */
        eat( );                /* yum-yum, spaghetti */
        put_forks(i);          /* put both forks back on table */
    }
}
```

# Dining Philosophers (4): Solution part2

```
void take_forks(int i)              /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                 /* enter critical region */
      state[i] = HUNGRY;            /* record fact that philosopher i is hungry */
      test(i);                      /* try to acquire 2 forks */
      up(&mutex);                   /* exit critical region */
      down(&s[i]);                  /* block if forks were not acquired */
}

void put_forks(i)                   /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                 /* enter critical region */
      state[i] = THINKING;          /* philosopher has finished eating */
      test(LEFT);                   /* see if left neighbor can now eat */
      test(RIGHT);                  /* see if right neighbor can now eat */
      up(&mutex);                   /* exit critical region */
}

void test(i)                        /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

# Readers and Writers Problem

- Models access to a database
  - Acceptable to have multiple processes reading the database at same time
  - One process writing to the database: no other processes may access (even readers)

# Readers and Writers Problem

```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;            /* controls access to ’rc’ */
semaphore db = 1;               /* controls access to the database */
int rc = 0;                     /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {             /* repeat forever */
          down(&mutex);         /* get exclusive access to ’rc’ */
          rc = rc + 1;          /* one reader more now */
          if (rc == 1) down(&db); /* if this is the first reader ... */
          up(&mutex);           /* release exclusive access to ’rc’ */
          read_data_base( );    /* access the data */
          down(&mutex);         /* get exclusive access to ’rc’ */
          rc = rc - 1;          /* one reader fewer now */
          if (rc == 0) up(&db); /* if this is the last reader ... */
          up(&mutex);           /* release exclusive access to ’rc’ */
          use_data_read( );     /* noncritical region */
     }
}


void writer(void)
{
     while (TRUE) {             /* repeat forever */
          think_up_data( );     /* noncritical region */
          down(&db);            /* get exclusive access */
          write_data_base( );   /* update the data */
          up(&db);              /* release exclusive access */
     }
}
```

# Readers and Writers Problem

```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to 'rc' */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                 /* repeat forever */
          down(&mutex);             /* get exclusive access to 'rc' */
          rc = rc + 1;              /* one reader more now */
          if (rc == 1) down(&db);   /* if this is the first reader ... */
          up(&mutex);               /* release exclusive access to 'rc' */
          read_data_base();         /* access the data */
          down(&mutex);             /* get exclusive access to 'rc' */
          rc = rc − 1;              /* one reader fewer now */
          if (rc == 0) up(&db);     /* if this is the last reader ... */
          up(&mutex);               /* release exclusive access to 'rc' */
          use_data_read();          /* noncritical region */
     }
}


void writer(void)
{
     while (TRUE) {                 /* repeat forever */
          think_up_data();          /* noncritical region */
          down(&db);                /* get exclusive access */
          write_data_base();        /* update the data */
          up(&db);                  /* release exclusive access */
     }
}
```

# Readers and Writers Problem

```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;            /* controls access to 'rc' */
semaphore db = 1;               /* controls access to the database */
int rc = 0;                     /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {              /* repeat forever */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc + 1;            /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        read_data_base();       /* access the data */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc − 1;            /* one reader fewer now */
        if (rc == 0) up(&db);   /* if this is the last reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        use_data_read();        /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {              /* repeat forever */
        think_up_data();        /* noncritical region */
        down(&db);              /* get exclusive access */
        write_data_base();      /* update the data */
        up(&db);                /* release exclusive access */
    }
}
```

# Readers and Writers Problem

- Problem
  - Additional readers admitted as they come along
  - As long as there's a steady supply of readers, writer will be suspended
- Solution
  - When a reader arrives, if a writer is waiting, the reader is suspended behind the writer

# Summary

- Race condition, mutual exclusion
- Classic IPC problems
- Additional practice
  - Read Linux documentation: LINUX_SRC/Documentation/spinlocks.txt
  - Find the implementation of `down` and `up` in LINUX_SRC/kernel/semaphore.c
  - Spinlock v.s. Mutex: http://stackoverflow.com/questions/5869825/when-should-one-use-a-spinlock-instead-of-mutex