Operating System 2
Dr. Khaled Morsy

Inter-Process Communications

Reference : Modern Operating Systems (Andrew Tanenbaum) 3rd edition

Operating systems 2
Dr. Khaled Morsy

Inter-Process Communications

Other Reference : Modern Operating Systems (Andrew Tanenbaum) 3rd edition

Interprocess communication IPC

- Processes frequently need to communicate with other processes.
- Inter-Process Communication, which in short is known as IPC, deals mainly with the techniques and mechanisms that facilitate communication between processes.

Inter-process communication IPC

- Enables one application to control another one, and for several applications to share the same data without interfering with each other
- Inter-Process Communication, required in all multiprocessing systems. Not supported by single-process operating system (as DOS)

Interprocess communication IPC

- OS/2 and Microsoft Windows support IPC mechanism called DDE ***(D****ynamic* ***D****ata* ***E****xchange),* an IPC built into the Macintosh, Windows, and OS/2 operating systems.

Interprocess communication IPC

- Each process is assigned some part of the available memory. i.e., it will have its unique address space.
- In no way will the memory assigned for one process overlap with the memory assigned to another process.
- Operating System will act as the communication channel among processes.

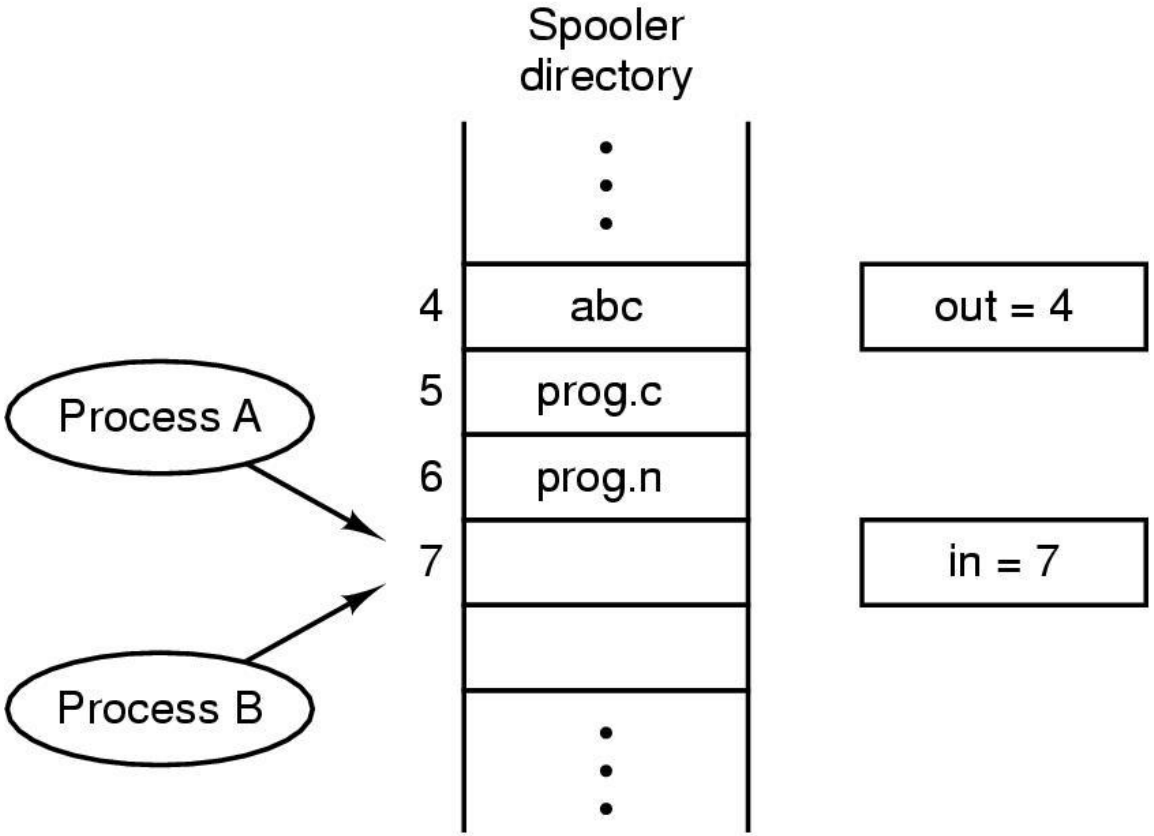
Interprocess Communication

We can summarize the roles of IPC as follows:

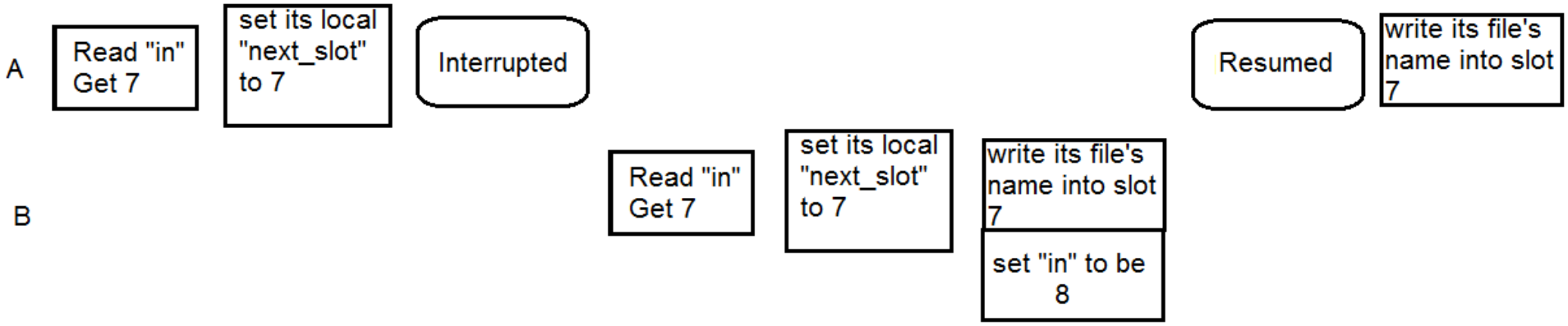
1. Provides ways of passing information among processes.
2. Protects critical activities (e.g. updating shared data)
3. Proper sequencing in case of dependencies

Race Conditions

Two processes want to access shared data at the same time.



Race Conditions



Critical Section

A part or a region of code that need to be executed atomically (with no interrupt) such as accessing a resource (file, input or output port, global data, etc.).

Critical Section

General structure of process p_i is:

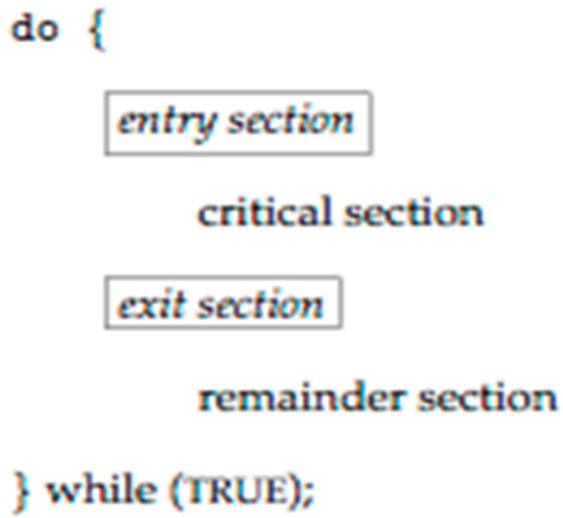
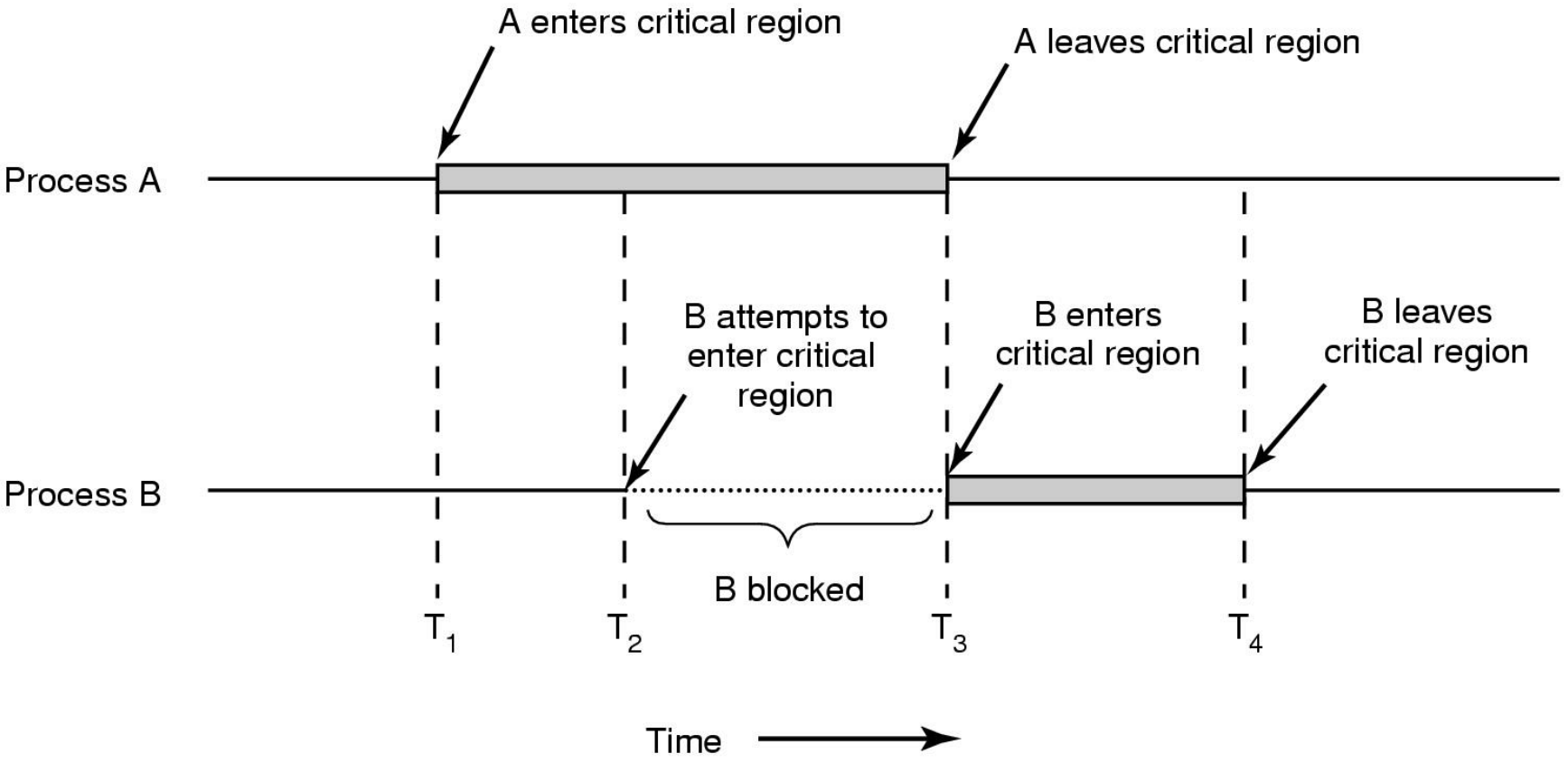


Figure 6.1 General structure of a typical process P_i.

Critical Section (region)



Mutual exclusion using critical regions

We need four conditions to provide correct mutual exclusion

- 1- No two processes simultaneously in critical region
- 2- no assumption about speeds and number of CPUs
- 3- No process running outside its critical region may block other processes
- 4- No process must wait forever to enter its critical region

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 solution to critical region problem

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

Peterson's Solution

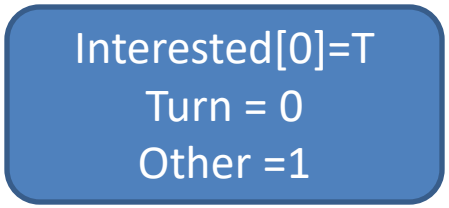
```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
      int other;                        /* number of the other process */

      other = 1 – process;              /* the opposite of process */
      interested[process] = TRUE;       /* show that you are interested */
      turn = process;                   /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
      interested[process] = FALSE;      /* indicate departure from critical region */
}
```

`Interested(process)=False` => process is not interested and does not want to enter critical section

If both are interested, a process can enter only if it sets "turn" first.

Case 1 : P0 interested
P1 : not interested

- 0 1



Turn=0

exit_loop --- because interested[1] = F

enter_region(0)

interested[0]=F

Case 2 : P0 interested ,Run first P1 : interested , Run Second

- 0

Interested[0]=T
Turn = 0
Other =1

Turn = 1

exit_loop

enter_region(0)

interested[0]=F

1

Interested[1]=T
Turn =1
Other=0

Turn = 1

continue_loop

interested[0]=F

exit_loop

enter_region(1)

interested[1] = F

Case 3 : P0 interested ,Run second P1 : interested , Run First

- 0 1

Interested[0]=T
Turn = 0
Other =1

Interested[1]=T
Turn =1
Other=0

Turn = 0

continue_*loop*

interested[1]=F

exit_loop

enter_region(0), interested[0]=F

Turn = 0

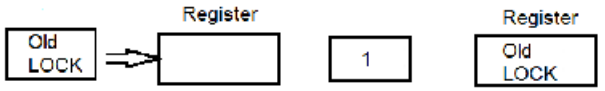
exit_loop

enter_region(1)

interested[1]=F

Test-and-set lock

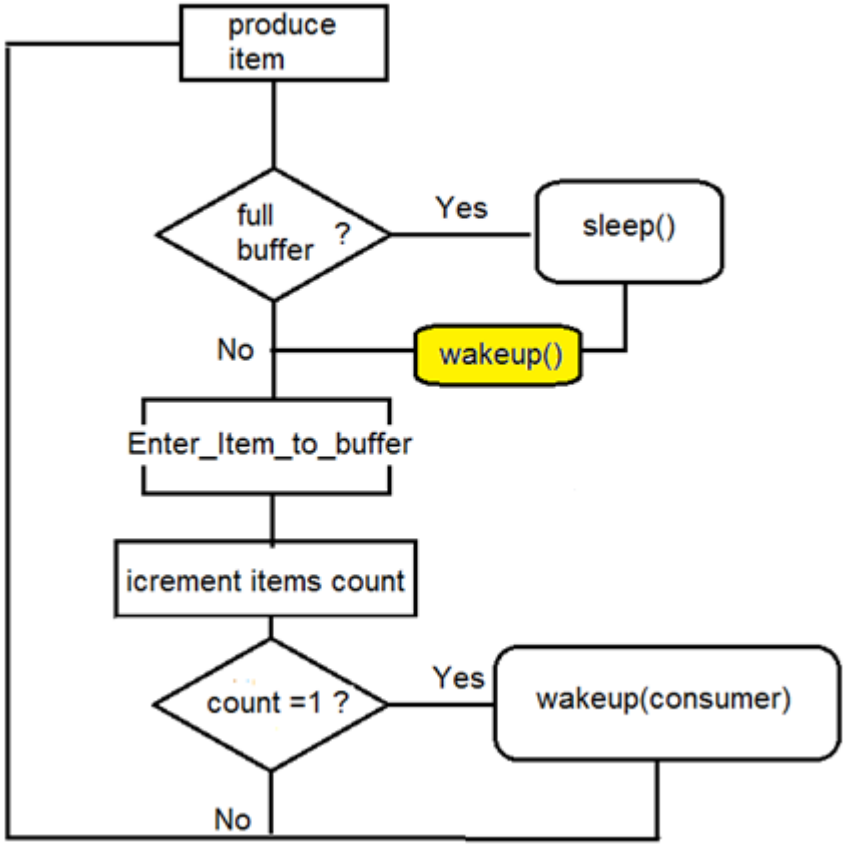
```
enter_region:
    TSL REGISTER, LOCK    | copy lock to register and set lock to 1
    CMP REGISTER,#0       | was lock zero?
    JNZ enter_region      | if it was non zero, lock was set, so loop
    RET                   | return to caller; critical region entered

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



Entering and leaving a critical region using the TSL instruction

Producer-consumer problem

- Producer side



- Consumer side ?

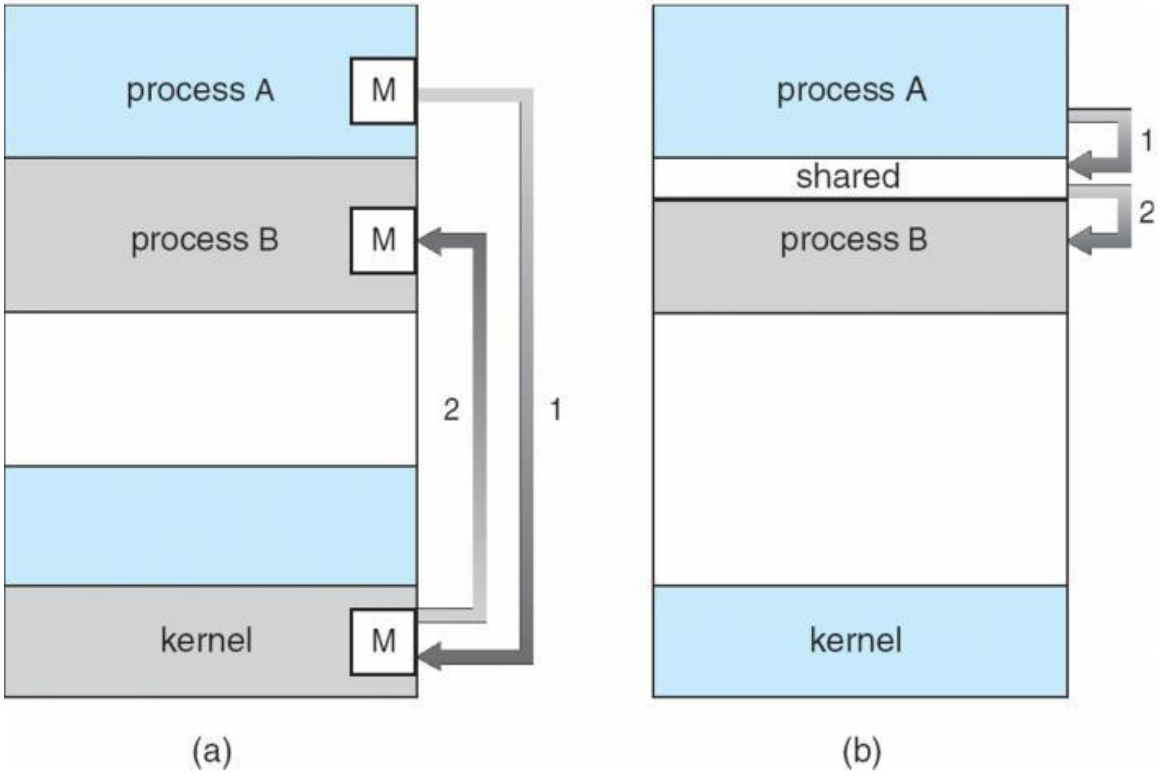
Sleep and Wakeup

```
#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 */
     }
}
```

Producer-consumer problem with fatal race condition

Communication using Shared Memory and Message passing

- (a) Message Passing (b) Shared Memory



Message-passing VS. Shared-Memory

- Message passing is useful for exchanging smaller amounts of data.
- Message passing is also easier to implement than is shared memory for inter-computer communication.
- Shared memory is faster than message passing, as message passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.

- in shared memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Semaphores

Integer variable with two atomic operations

- *down*: if 0, then go to sleep;
  if >0, then decrement value
- *up*: increment value and let a sleeping process to perform a *down*

Implementation by disabling all interrupts by the kernel.

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 */
    }
}
```

The producer-consumer problem using semaphores

Mutexes

```
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
```

Implementation of *mutex_lock* and *mutex_unlock* for synchronization of threads in user space

see Java : ReadWriteLock

- Java provide public interface **ReadWriteLock**
- A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers.
- The write lock is exclusive.
  All ReadWriteLock implementations must guarantee that the memory synchronization effects of writeLock operations (as specified in the Lock interface) also hold with respect to the associated readLock. That is, a thread successfully acquiring the read lock will see all updates made upon previous release of the write lock.
- http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html

Monitors (1)

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Example of a monitor - only one process inside the monitor at any time

Monitors (2)

```
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;
```

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has *N* slots

Condition variables with *wait* and *signal*

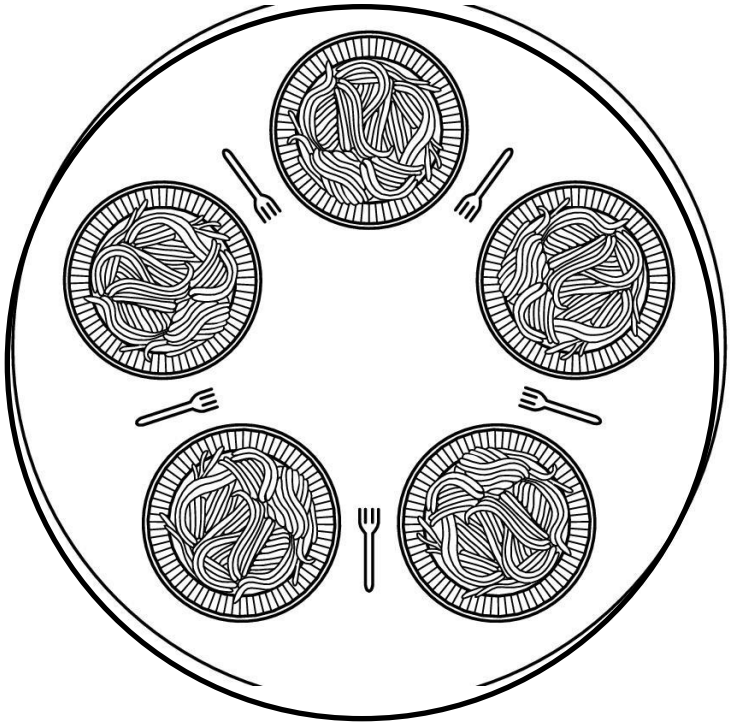
Message Passing

```
#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 */
     }
}
```

The producer-consumer problem with N messages

Dining Philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



A non-solution to the dining philosophers problem

```
#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 */
    }
}
```

Deadlock-free code for Dining Philosophers (1)

```
#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 */
    }
}
```

Deadlock-free code for Dining Philosophers (2)

```
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-Writers problem

- An object is shared among many threads, each belonging to one of
- two classes:
- – Readers: read data, never modify it
- – Writers: read data and modify it
- Using a single lock on the data object is overly restrictive
- => Want many readers reading the object at once
- – Allow only one writer at any point
- – How do we control access to the object to permit this protocol?
- Correctness criteria:
- – Each read or write of the shared data must happen within a critical section.
- – Guarantee mutual exclusion for writers.
- – Allow multiple readers to execute in the critical section at once

The 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 */
      }
}
```

The Sleeping Barber Problem

Solution to the Sleeping Barber Problem

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting – 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

Assignment (1)

- Implement at least 3 of the following using Java or C++:
- 1)the producer-consumer problem using sleep and wakeup
- 2) Peterson's algorithm
- 3) The reader-writer problem
- 4) The producer-consumer problem using semaphore
- 5) Dinning Philosophers problem
- 6) Sleeping barber problem