CSCI 360
Introduction to Operating Systems

Process Management

Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

Outline

- Inter Process Communications (IPC)
  - Pipes
  - Message Passing
  - Shared Memory
- Process Synchronization
  - Race Condition
  - Critical Region Problem: Peterson's Solution
  - Producer Consumer Problem
  - Semaphore
  - Mutex
  - The Dinning Philosophers Problem
  - The Readers and Writers Problem

Inter Process Communication

- Processes within a system may be ***independent*** or ***cooperating***
- ***Independent*** process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)

Inter Process Communication

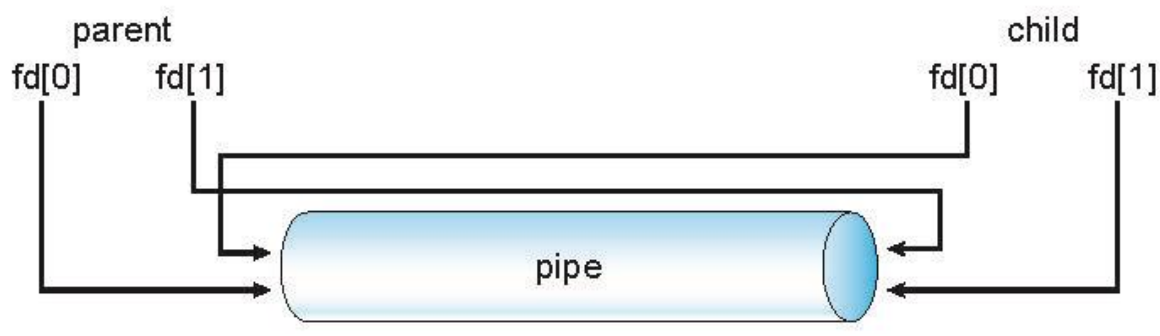
- Three models of IPC
  - **Pipe**
  - **Shared Memory**
  - **Message Passing (Message Queue)**

IPC: Pipe

- Pipe acts as a conduit allowing two processes to communicate
- **Ordinary pipes** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- **Named pipes** – can be accessed without a parent-child relationship.

IPC: Pipe

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes

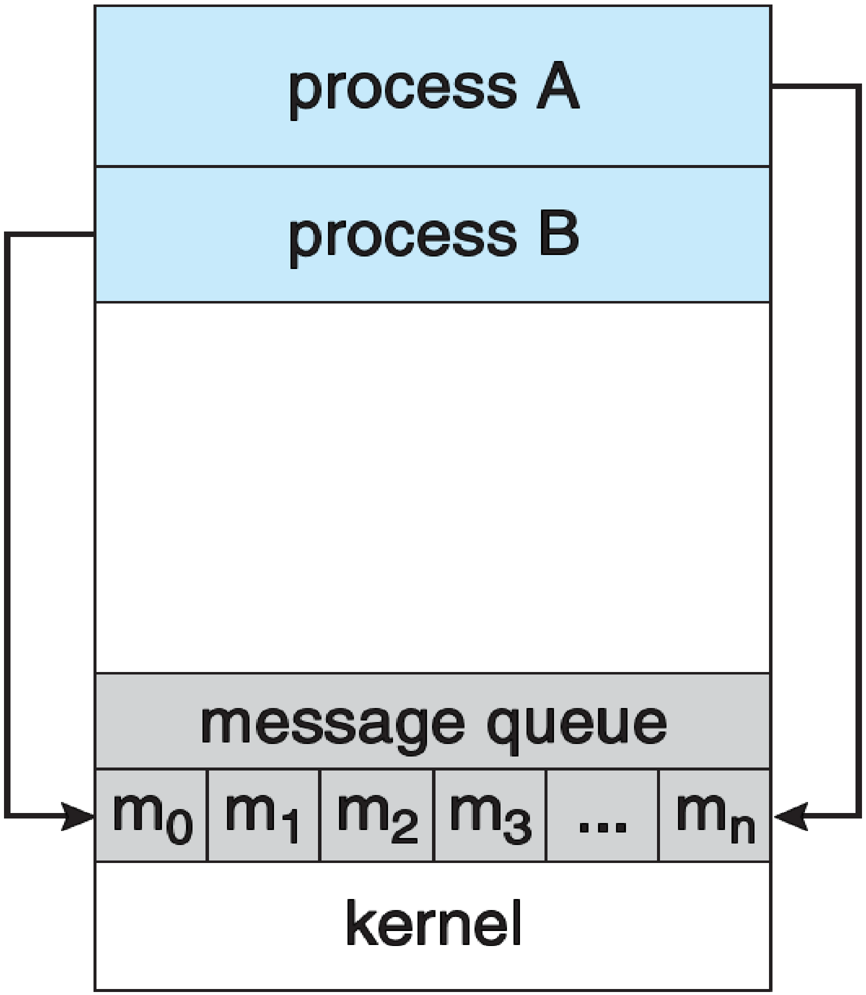


- Windows calls these **anonymous pipes**

IPC: Pipe

- Named Pipes are more powerful than ordinary pipes
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Communication is unidirectional in most of the systems
- Provided on both UNIX and Windows systems

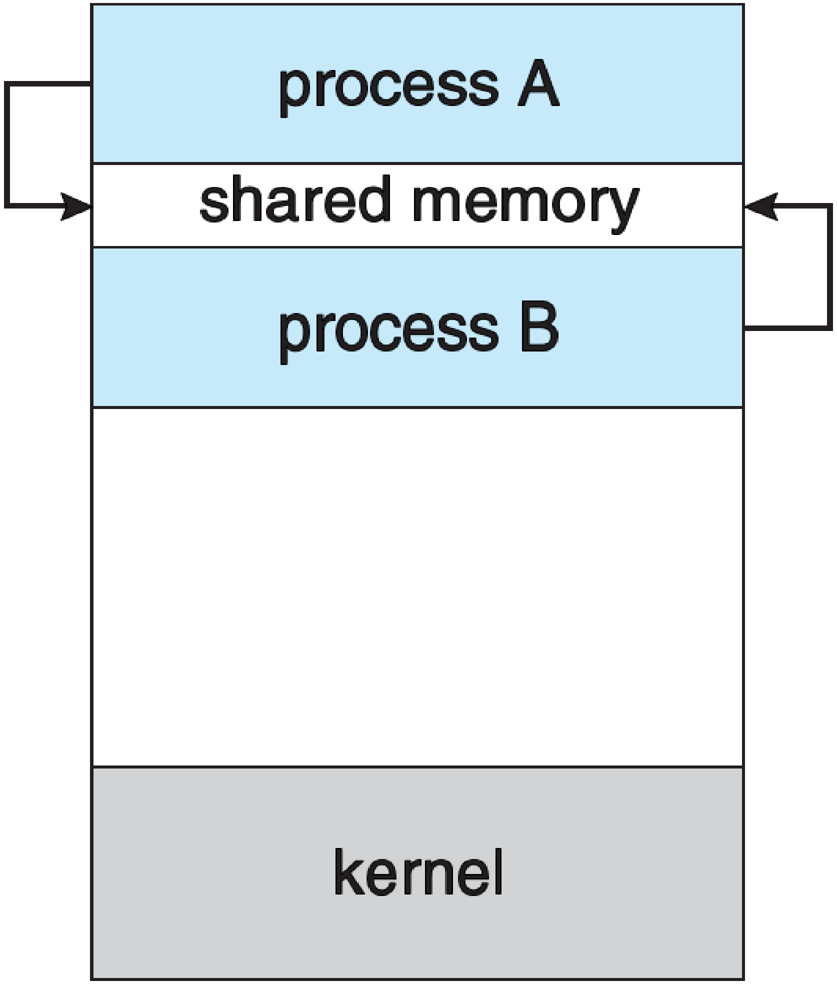
Message Queue



Message Queue

- Message queue has configurable internal structure
  - **size** of each message
  - **size** of the queue
- Provides two operations to the communicating processes
  - **`send`**(*message*)
  - **`receive`**(*message*)
- More than two processes can send and receive
- Send operation assigns a priority to each message
- Oldest message with the highest priority goes to the front of the queue
- Receive operation gets and removes the front message from the queue.
- A process can check the status of the queue.

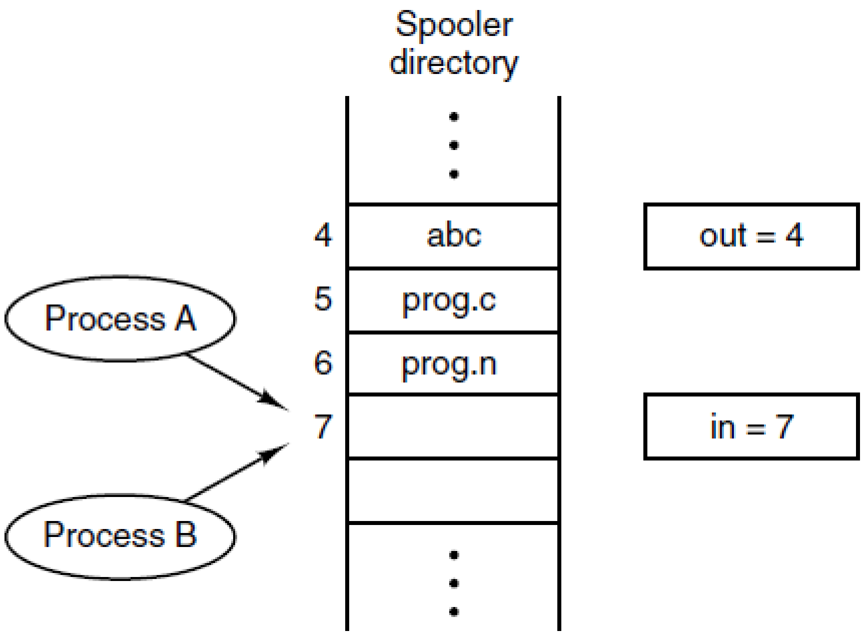
Shared Memory



Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Race Condition



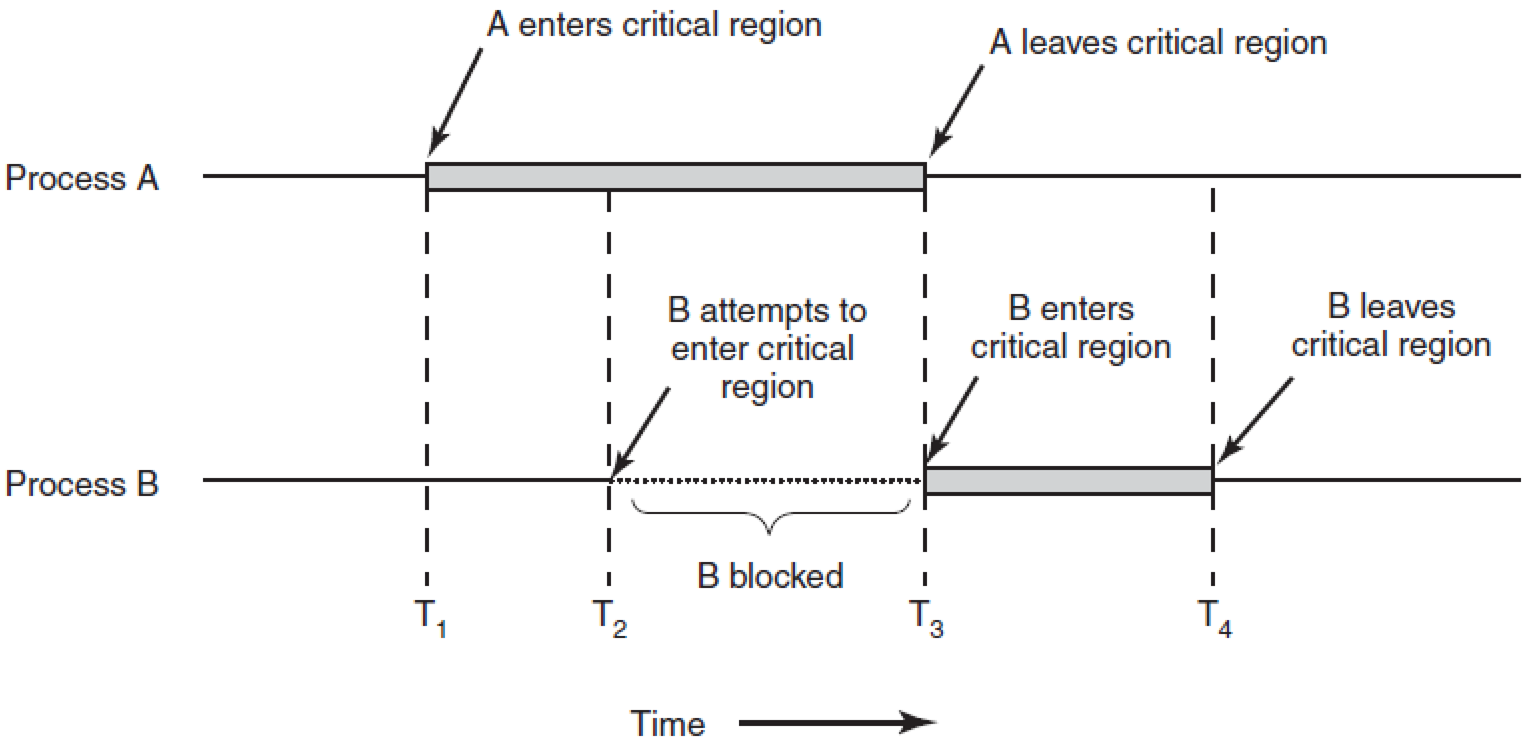
Two processes want to access shared memory at the same time.

Critical Regions

Requirements to avoid race conditions:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

Critical Regions



Mutual exclusion using critical regions.

Mutual Exclusion with 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)

A proposed solution to the critical region problem.
(a) Process 0. (b) Process 1.

Mutual Exclusion with Busy Waiting:
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 */
}
```

Peterson's solution for achieving mutual exclusion.

Mutual Exclusion with Busy Waiting:

The TSL Instruction

```
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 nonzero, 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.

Mutual Exclusion with Busy Waiting:
The TSL Instruction

```
enter_region:
        MOVE REGISTER,#1        | put a 1 in the register
        XCHG REGISTER,LOCK      | swap the contents of the register and lock variable
        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 XCHG instruction

Sleep and Wakeup

The 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)
{
```

The producer-consumer problem with a fatal race condition.

Sleep and Wakeup

The Producer-Consumer Problem

```
            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 */
        }
}
```

The producer-consumer problem with a fatal race condition.

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)
```

The producer-consumer problem using semaphores.

Semaphores

```
            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
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*.

Mutexes in Pthreads

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads

```c
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                         /* put item in buffer */
                pthread_cond_signal(&condc);        /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
```

Using threads to solve the producer-consumer problem.

Mutexes in Pthreads

```
        pthread_exit(0);
}

void *consumer(void *ptr)                          /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                        /* take item out of buffer */
                pthread_cond_signal(&condp);       /* wake up producer */
                pthread_mutex_unlock(&the_mutex);  /* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
```

Using threads to solve the producer-consumer problem.

Mutexes in Pthreads

```
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Using threads to solve the producer-consumer problem.

The Producer-Consumer Problem with 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)
```

The producer-consumer problem with *N* messages.

The Producer-Consumer Problem with Message Passing

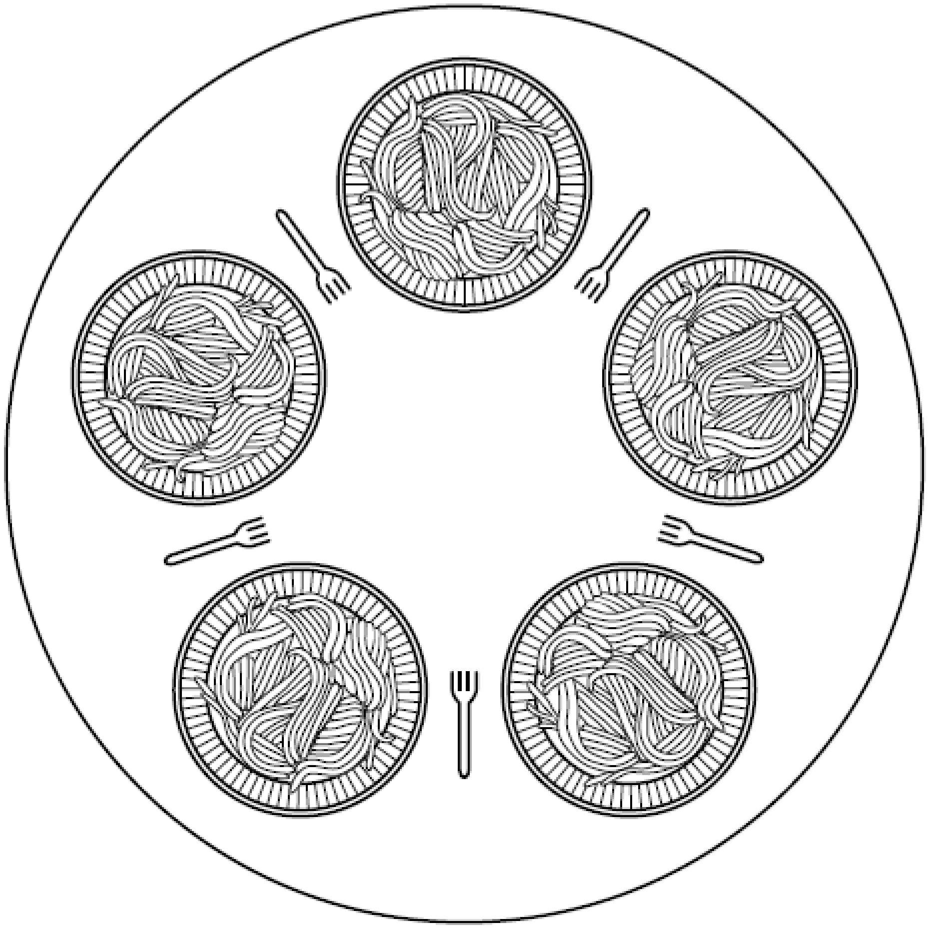
```
            send(consumer, &m);                     /* send item to consumer */
        }
}

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.

The Dining Philosophers Problem



Lunch time in the Philosophy Department.

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 */
     }
}
```

A nonsolution to the dining philosophers problem.

The Dining Philosophers Problem

```
#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 */
    }
}
```

A solution to the dining philosophers problem.

The Dining Philosophers Problem

```
        put_forks(i);                     /* put both forks back on table */
    }
}

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 */
```

A solution to the dining philosophers problem.

The Dining Philosophers Problem

```
}

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]);
      }
}
```

A solution to the dining philosophers problem.

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)
```

A solution to the readers and writers problem.

The Readers and Writers Problem

```
        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 */
    }
}
```

A solution to the readers and writers problem.

Summary

- Pipe
- Message Passing
- Shared Memory
- Race Condition
- Critical Region Problem: Peterson’s Solution
- Producer Consumer Problem
- Semaphore
- Mutex
- The Dinning Philosophers Problem
- The Readers and Writers Problem

Next

Memory Management

- Address Space
- Memory allocation algorithms
- Swapping and compaction
- Virtual Memory
- Paging