UNIVERSIDAD FRANCISCO DE VITORIA
Computación de Alto Rendimiento

COMPUTACION DE ALTO RENDIMIMIENTO

Mecanismos de sincronismo: Semáforos y Paso de Mensajes

Semáforos

- **Semáforo [Dijkstra 1965]**
  - Mecanismo de sincronización que se utiliza en sistemas con memoria compartida (monoprocesador, multiprocesador).
  - Herramienta de sincronización que brinda una solución al problema de la exclusión mutua restringiendo el acceso simultáneo a los recursos compartidos.
  - Es un objeto con un valor entero al que se le asigna un valor inicial no negativo y al que sólo se puede acceder utilizando 2 operaciones **atómicas**: **wait** y **signal**.
    - El valor de un semáforo representa la cantidad de instancias libres de un recurso determinado.

Semáforos

- De hecho un semáforo es una estructura:

```
type semaphore = record
                    contador: integer;
                    cola: lista de procesos
                 end;
var S: semaphore;
```

- Un proceso esperando un semáforo S, está bloqueado y puesto en la cola del semáforo
- Signal(S) libera (siguiendo una política justa, ej: FIFO) un proceso de S.cola y lo pondrá en la lista de preparados.

Semáforos: funcionamiento

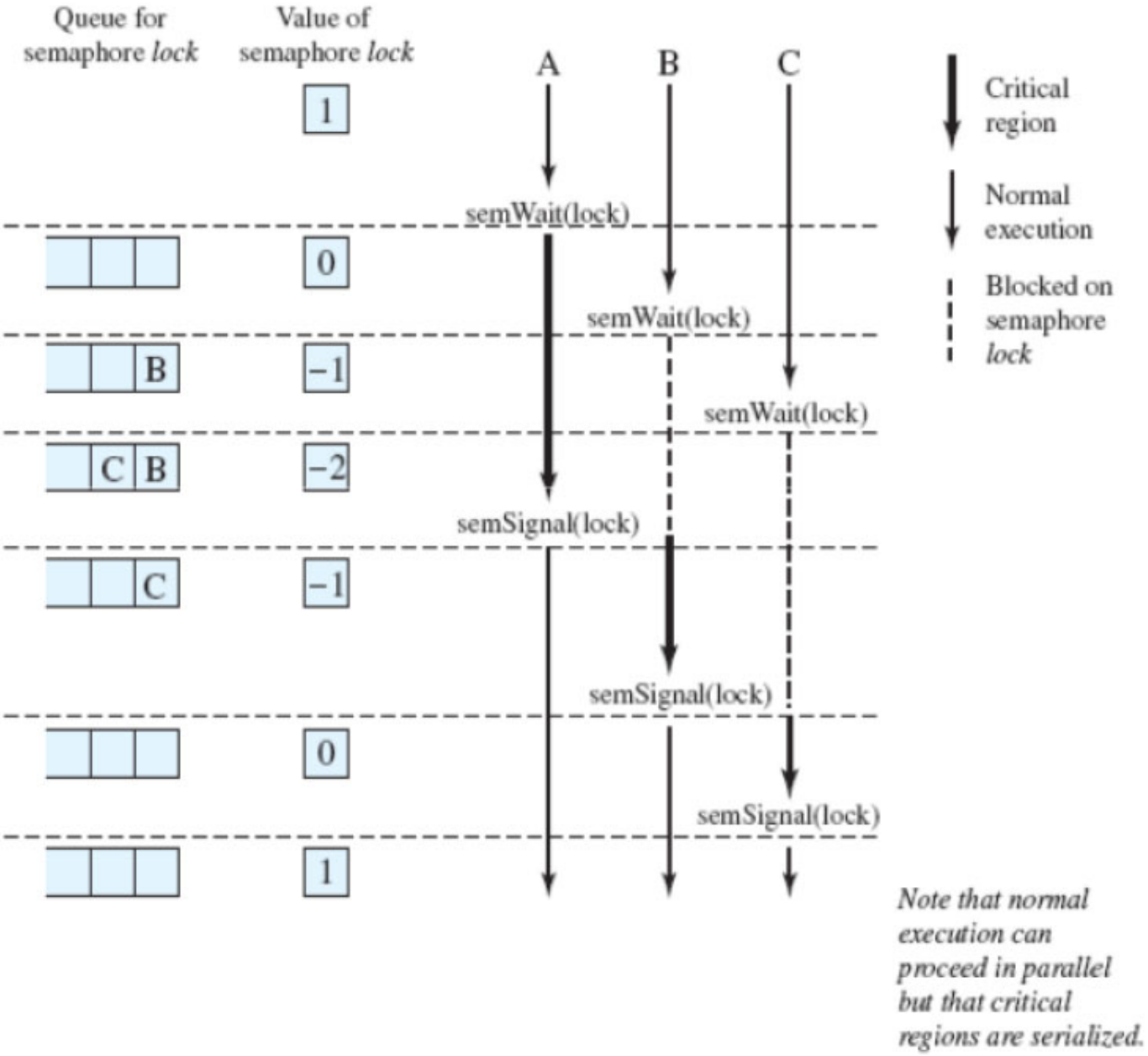
- El semáforo se inicializa con el número total de recursos disponibles.
- Las operaciones ***wait*** y ***signal*** se diseñan de modo que se impida el acceso al recurso protegido por el semáforo cuando el valor de éste es menor o igual que cero.
- Al solicitar un recurso, el semáforo se **decrementa.**
- Al liberar un recurso se **incrementa.**
- Si la operación ***wait*** se ejecuta cuando el semáforo tiene un valor menor que uno, el proceso debe quedar en espera de que la ejecución de una operación ***signal*** libere alguno de los recursos.

Operaciones (atómicas) para semáforos

```
wait(S):
  S.contador--;
  if (S.contador<0) {
    suspender ESTE proceso
    poner ESTE proceso in S.cola
  }

signal(S):
  S.contador++;
  if (S.contador<=0) {
    retirar un proceso P de S.cola
    poner P en la lista de preparados
  }
```

S.contador debe ser inicializado a un valor (normalmente = 1)

Procesos utilizando semáforos



Exclusión mutua y Sincronización

- Uso de semáforos para obtener exclusión mutua

  `semaforo mutex = 1`

```
Proceso P1() {
    down(mutex);
    /* RC */
    up(mutex);
}
```

```
Proceso P2() {
    down(mutex);
    /* RC */
    up(mutex);
}
```

- Uso de semáforos para sincronización de procesos

  "P2 debe ejecutarse luego de P1"

  `semaforo s = 0`

```
Proceso P1() {
    ...
    /* RC */
    up(s);
}
```

```
Proceso P2() {
    down(s);
    /* RC */
    ...
}
```

Semáforos: observaciones

- Si S.contador >=0: el número de procesos que pueden ejecutar wait(S) in bloquearse es = S.contador
- Cuando S.contador<0: el número de procesos bloqueados será S = |S.contador|
- Atomicidad y exclusión mutua: un solo proceso puede ejecutar wait(S) y signal(S) (sobre el mismo semáforo S) en cualquier instante (incluso si hay varios CPUs)

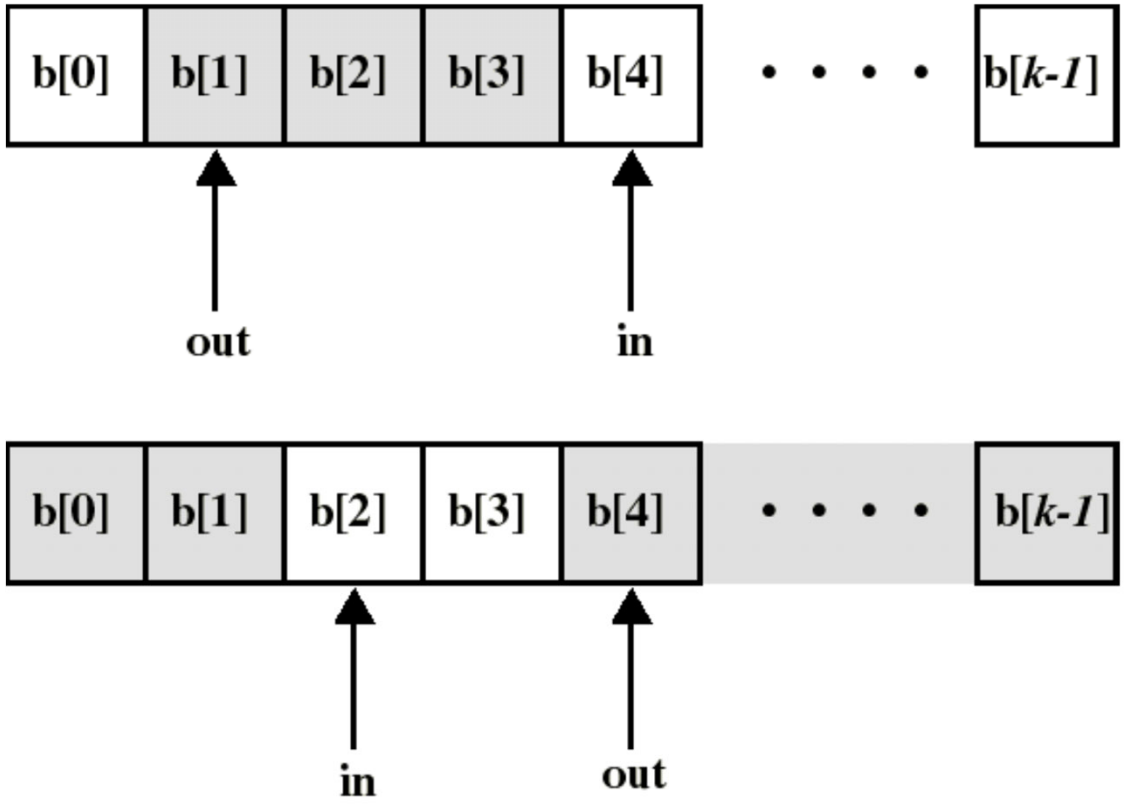
El problema del productor/consumidor

- Un proceso productor produce información que será utilizada por un proceso consumidor
  - Ej1: un pgm de impresión produce caracteres -- a ser "consumidos" por un pgm que imprime.
  - Ej2: un ensamblador produce módulos objeto que serán consumidos par un cargador
- Necesitamos de un buffer (tampón) para almacenar los ítems producidos (esperando a ser consumidos)
- Este tipo de problemas se conocen como de " procesos cooperativos"

El problema del productor / consumidor

- **Enunciado**:
  - Uno o varios procesos generan datos y los colocan en un *buffer*.
  - Se asume que el *buffer* es un vector de datos finito (situación real).
  - Existe un único **consumidor** que extrae los datos del *buffer* de uno en uno.
  - Sólo un **productor** o **consumidor** puede acceder al *buffer* en un momento determinado (sección crítica).
    - El **Productor** no puede añadir datos si el *buffer* está lleno.
    - El **Consumidor** no puede eliminar datos del *buffer* si está vacío.

P/C: buffer circular de dimensión k



- Puede consumir solamente si el número de N ítems (consumibles) es al menos 1 (N!=in-out)
- Puede producir solamente si el número de espacios libres libres es al menos 1

P/C: buffer circular de dimensión k

- Utilizamos un semáforo S para la exclusión mutua en el acceso al buffer
- Utilizamos un semáforo N para sincronizar el productor y el consumidor en el número de ítems consumibles dentro del .
- Y además:
- Utilizamos un semáforo E para sincronizar el productor y el consumidor en cuanto al número de espacios libres.

Solución del P/C: buffer circular de dimensión k

```
Inicialización: S.contador:=1;
                N.contador:=0;
                E.contador:=k;
```

```
añade(v):
b[in]:=v;
in:=(in+1)
      mod k;
```

```
toma():
w:=b[out];
out:=(out+1)
       mod k;
return w;
```

```
Producer:
repeat
   produce v;
   wait(E);
   wait(S);
   añade(v);
   signal(S);
   signal(N);
forever
```

```
Consumer:
repeat
   wait(N);
   wait(S);
   w:=toma();
   signal(S);
   signal(E);
   consume(w);
forever
```

Semáforos POSIX

Librería Posix Linux

- `sem_init(sem_t *sem, int shared, int val);`
  - Inicializa un semáforo sin nombre
- `int sem_destroy(sem_t *sem);`
  - Destruye un semáforo sin nombre
- `sem_t *sem_open(char *name,int flag,mode_t mode,int val);`
  - Abre (crea) un semáforo con nombre.
- `int sem_close(sem_t *sem);`
  - Cierra un semáforo con nombre.
- `int sem_unlink(char *name);`
  - Borra un semáforo con nombre .
- `int sem_wait(sem_t *sem);`
  - Realiza la operación *s_espera* sobre un semáforo.
- `int sem_post(sem_t *sem);`
  - Realiza la operación *s_abre* sobre un semáforo.

Productor – Consumidor con semáforos POSIX

```
#define MAX_BUFFER  1024
#define DATOS_A_PRODUCIR  100000
sem_t elementos; /* elementos buffer */
sem_t huecos; /* huecos buffer */
sem_t mutex;  /* controla excl.mutua */

int buffer[MAX_BUFFER];/*buffer común*/
void main(void)
{
   pthread_t th1, th2; /* id. threads*/
/* inicializar los semaforos */
   sem_init(&elementos, 0, 0);
   sem_init(&huecos, 0, MAX_BUFFER);
   sem_init(&mutex, 0, 1);
```

```
/* crear los procesos
ligeros */
pthread_create(&th1, NULL,
Productor, NULL);
pthread_create(&th2, NULL,
Consumidor, NULL);

/* esperar su finalizacion
*/
pthread_join(th1, NULL);
pthread_join(th2, NULL);

sem_destroy(&huecos);
sem_destroy(&elementos);
sem_destroy(&mutex);

exit(0);
}
```

Productor – Consumidor con semáforos POSIX

```
void Productor(void)    /* código del productor */
{
   int pos = 0;   /* posición dentro del buffer */
   int dato;      /* dato a producir */
   int i;

   for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
      dato = i;              /* producir dato */
      sem_wait(&huecos);   /* un hueco menos */
      sem_wait(&mutex);   /* entra en la sección crítica */
      buffer[pos] = dato;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&mutex);   /* deja la sección crítica */
      sem_post(&elementos);   /* un elemento más */
   }
   pthread_exit(0);
}
```

Productor – Consumidor con semáforos POSIX

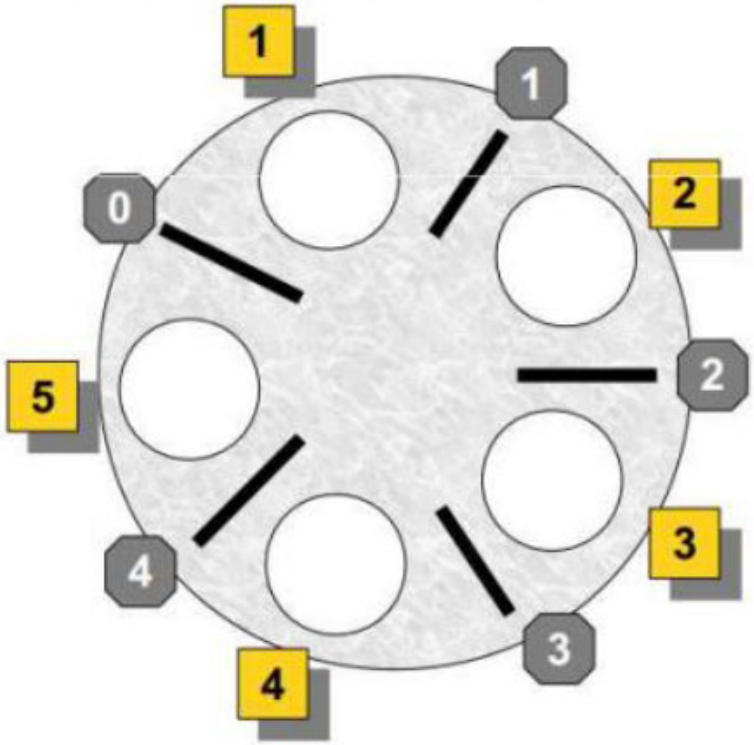
```
void Consumidor(void)   /* código del Consumidor */
{
   int pos = 0;
   int dato;
   int i;

   for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
      sem_wait(&elementos);     /* un elemento menos */
      sem_wait(&mutex);  /* entra en la sección crítica */
      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&mutex);  /* deja la sección crítica */
      sem_post(&huecos);     /* un hueco más */
      /* consumir dato */
   }
   pthread_exit(0);
}
```

Cena de los filósofos

Típico problema donde un conjunto de procesos compiten por recursos compartidos

Definido por Dijkstra en 1965, Hoare propuso la fantasía

5 filósofos = 5 procesos (N=5)
Cada filósofo precisa 2 palillos (recursos) para comer (ejecutarse) arroz

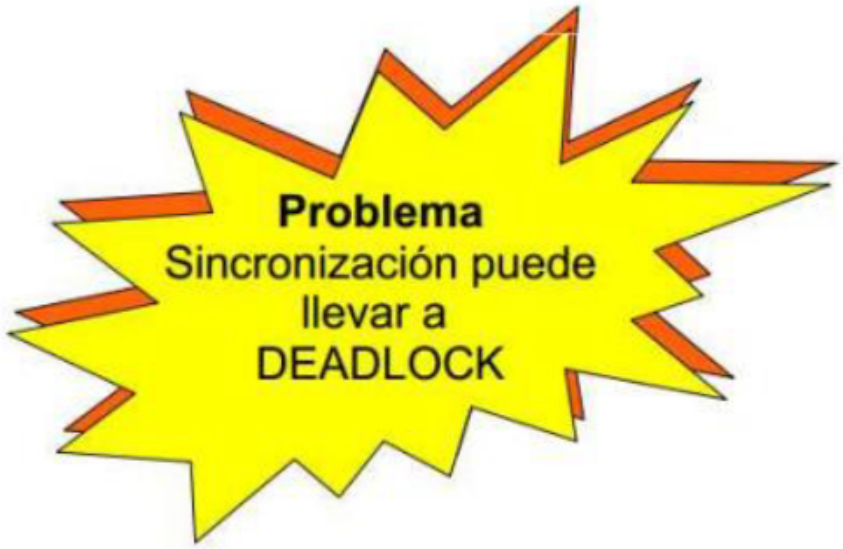


	IZQ	DER
i	i%N	(i-1)%N
1	1	0
2	2	1
3	3	2
4	4	3
5	0	4

Cena de los filósofos

- **Cena de los filósofos: Solución 1**

```
#define N 5
#define IZQ(i) i%N
#define DER(i) (i-1)%N

void filosofo (int i) {
    while (1) {
        pensar();
        tomar_palillo(IZQ(i));
        tomar_palillo(DER(i));
        comer();
        soltar_palillo(IZQ(i));
        soltar_palillo(DER(i));
    }
}
```



Cena de los filósofos

- **Cena de los filósofos: Solución 2 (con semáforos)**

```
#define N 5
#define IZQ(i) i%N
#define DER(i) (i-1)%N
semaforo mutex = 1

void filosofo (int i) {
    while (1) {
        pensar();
        down(mutex);
        tomar_palillo(IZQ(i));
        tomar_palillo(DER(i));
        comer();
        soltar_palillo(IZQ(i));
        soltar_palillo(DER(i));
        up(mutex);
    }
}
```

Cena de los filósofos

- **Cena de los filósofos: Solución 3**

```
#define N 5
#define IZQ i%N
#define DER (i-1)%N
#define PENSAR 0
#define APETITO 1
#define COMER 2
semaforo mutex=1;
int estado[N];
semaforo s[N]; //todos en cero

void filosofo(int i) {
    while (1) {
        pensar();
        tomar_palillo(i);
        comer();
        poner_palillo(i);
    }
}

void tomar_palillo(int i) {
    down(mutex);
    estado[i] = APETITO;
    test(i);
    up(mutex);
    down(s[i]);
}

void poner_palillo(int i) {
    down(mutex);
    estado[i] = PENSAR;
    test(IZQ(i));
    test(DER(i));
    up(mutex);
}

void test(int i) {
   if (estado[i]==APETITO && estado[IZQ(i)]!=COMER && estado[DER(i)]!=COMER) {
        estado[i] = COMER;
        up(s[i]);
   }
}
```

Semáforos binarios

- Les semáforos convencionales se llaman enumerados
- Otro tipo de semáforos son los semáforos binarios
  - similares pero el contador es booleano
  - los semáforos binarios pueden servir para construir semáforos enumerados ...
  - Generalmente mas difíciles de utilizar que los semáforos enumerados (no pueden inicializarse a k > 1)

Semáforos binarios

```
waitB(S):
    if (S.value = 1) {
      S.value := 0;
    } else {
      suspender este proceso
      poner este proceso en S.cola
   }

signalB(S):
   if (S.cola está vacía) {
      S.value := 1;
   } else {
      remover el proceso P de S.cola
      poner P en la lista de preparados
   }
```

Problemas asociados a la utilización semáforos

- Constituyen una herramienta útil para solucionar el problema de la exclusión

- wait(S) y signal(S) se vuelven difíciles de manejar cuando hay muchos procesos concurrentes

Paso de Mensajes

- Es un mecanismo general para la comunicación entre procesos
- para procesos de un mismo ordenador
  - para procesos en un sistema distribuido
- Puede servir tanto para exclusión mutua como para sincronización entre procesos.
- Se dispone de dos primitivas:
  - send(destino, mensaje)
  - receive(fuente, mensaje)
- Los procesos emisor y receptor pueden ser bloqueantes o no

Sincronización entre transmisor y receptor

- Para el transmisor: Es natural que no se bloquee luego de hacer un send().
  - Puede transmitir mensajes a varios destinos
  - Normalmente recibe una confirmación de mensaje recibido. Si hay errores se pueden generar mensajes por un tiempo infinito.
- Para el receptor: es conveniente que se bloquue en la operación de receive(.,.)
  - Normalmente necesita el mensaje para continuar con su trabajo
  - Si sucede algo extraño con el transmisor, puede bloquearse indefinidamente

Sincronización entre transmisor y receptor

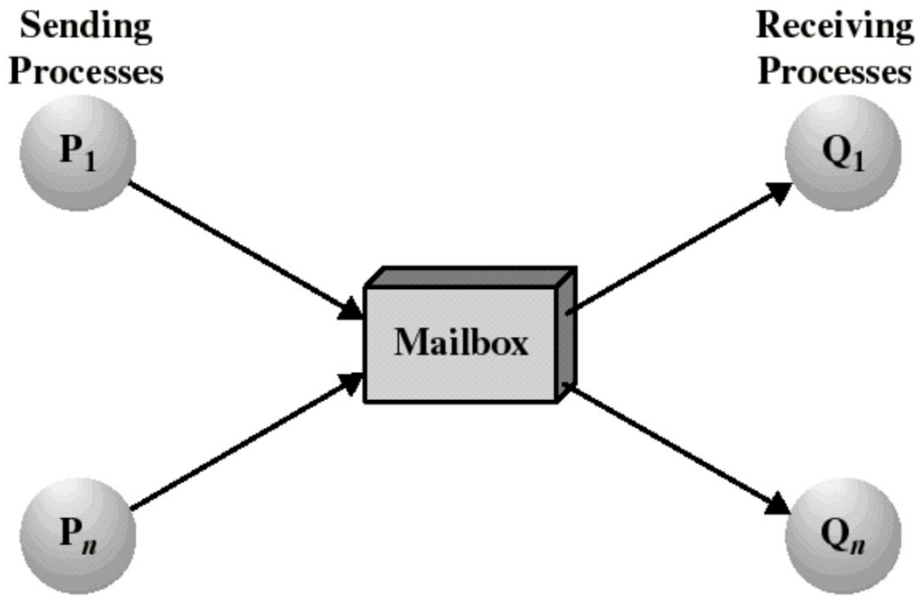
- Existen otras posibilidades
- Ej: send(,) y receive(,) bloqueantes los dos:
  - los 2 procesos se bloquean hasta que el mensaje sea enviado y recibido.
  - NO es necesario mantener una cola de mensajes
  - Se fuerza una sincronización entre los dos (*rendez-vous*)

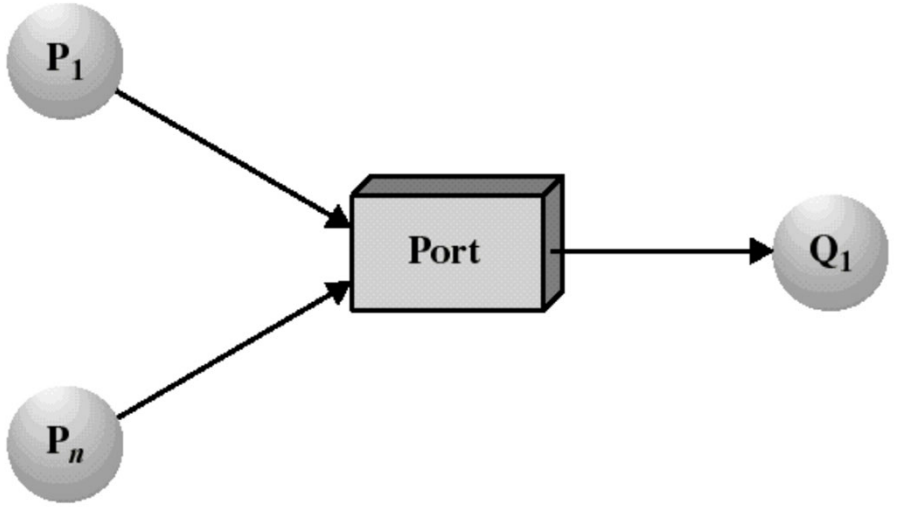
Direccionamiento de mensajes

- direccionamiento directo:
  - Se necesita un identificador específico para el transmisor (fuente) y el receptor (destino)
  - No es posible conocer la identificación del emisor antes de recibir un mensaje(ej: un servidor de impresión)
- direccionamiento indirecto (mas práctico):
  - mensajes se envían a una estructura de datos compartida
  - les transmisores escriben los mensajes en un “mailbox” y los receptores lo leen.

Buzones (mailboxes) y puertos

- Un buzón puede ser privado a un par transmisor-receptor
- Puede ser compartido entre varios transmisores/receptores
  - EL SO permite la utilización de un tipo para la selección de mensajes
- Puerto: Es un buzón asociado a un receptor y varios transmisores
  - En aplicaciones cliente/servidor el receptor es el servidor



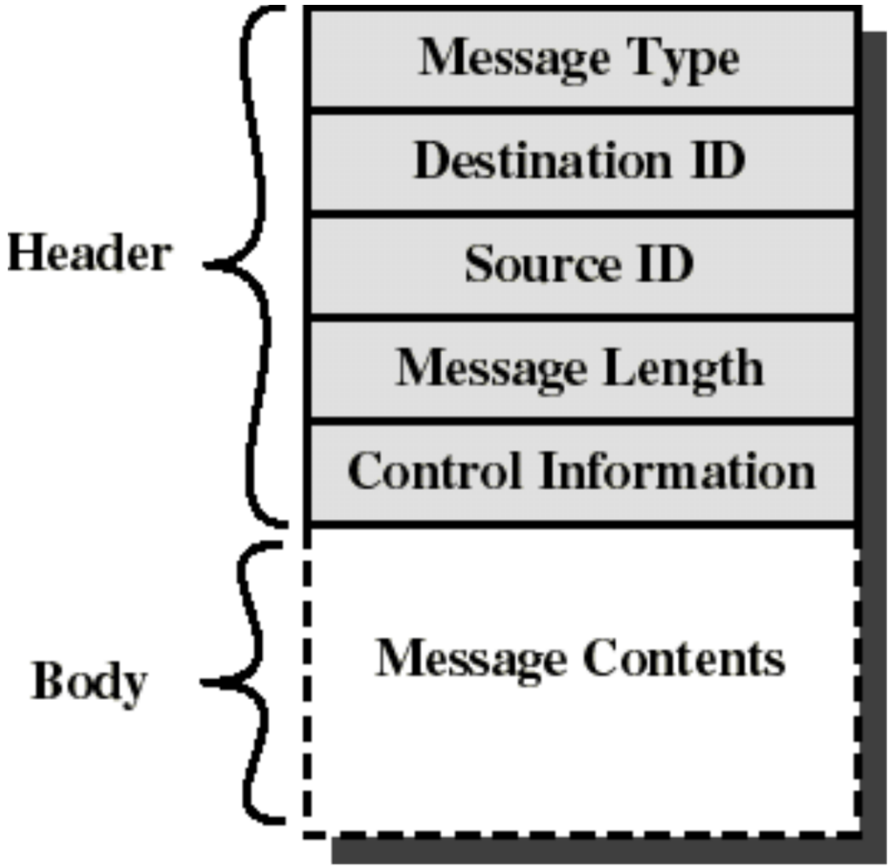


Propiedades de los puertos y buzones

- Un puerto se crea habitualmente por los procesos receptores
- El puerto se destruye una vez que el receptor termina
- El SO ofrece un servicio para la creación de buzones
- EL buzón se destruye una vez que el proceso que lo ha creado termina o bien mediante una llamada explícita a un servicio del SO

Formato de mensajes

- Encabezamiento y cuerpo
- Unix: ID, tipo
- info de control:
  - puntero a la lista de maensajes
  - número de secuencias
  - prioridad...
- **Tipo de tratamiento de la cola, puede ser FIFO incluyendo prioridades.**



Exclusión mutua mediante paso de mensajes

- **Creamos un buzón mutex compartido por n procesos**
- **send() no es bloqueante**
- **receive() bloqueado si mutex está vacío.**
- **Inicialización: send(mutex, "go");o de alguna forma ponemos "NULL"en el buzón.**
- **El 1er Pi que ejecute receive() entra en la SC. Los otros procesos están bloqueados hasta que Pi reenvíe el mensaje.**

```
Process Pi:
var msg: message;
repeat
   receive(mutex,msg);
   SC
   send(mutex,msg);
   SR
forever
```

Paso de Mensajes: P/C

```
struct tipo_mensaje = ... ;
const int capacidad = ... ;
void productor (void) {
    tipo_mensaje pmsg;
    while ( cierto ) {
        receive (puedep,pmsg);
        pmsg = producir();
        send (puedec, pmsg);
        otras_ope_productor;
    } // End while.
} // End productor.
void main {
    crear_buzón(puedep);
    crear_buzón(puedec);
    for (int i = 0; i < capacidad; i++)
        send(puedep,null);
    parbegin(productor,consumidor);
}
```

```
void consumidor (void ) {
    tipo_mensaje cmsg;
    while ( cierto ) {
        receive (puedec,cmsg);
        consumir (cmsj);
        send (puedep,cmsg);
        otras_ope_consumidor;
    } // End while.
} // End consumidor.
```

Unix SVR4: mecanismos de concurrencia

- Para comunicar datos entre entre procesos:
  - Canales de comunicación (“Pipes”)
  - Mensajes
  - Memoria común
- Para coordinación:
  - Señales
  - Semáforos

Para programación multihilo tenemos POSIX y dentro de ésta mecanismos de sincronismo como semáforos generales y binarios (posix).