# Práctica 1: Repaso de Concurrencia: Problemas clásicos e IPC

## Modalidad de entrega de la práctica y TP en clase resueltos:

Los ejercicios de esta práctica son ejercicios de resolución individual. Cada uno entrega su solución al problema en dos etapas, primero el diseño de la solución usando UML 2.x y luego la implementación (ver fechas de vencimiento en la última página). La entrega de los fuentes es por el moodle y la entrega del diseño es al comienzo de la clase del jueves.

La corrección de los ejercicios se hace en clase.

La práctica debe resolverse completamente y se espera que haga las consultas necesarias si tiene problemas de diagramación o implementación de la solución. Si los problemas de resolución están bien documentados, se permiten múltiples entregas.

Si no asiste a clase, debe entregar los ejercicios pedidos para esa clase. Las asistencia a la clase de corrección es OBLIGATORIA.

El TP del cuatrimestre se comienza resolviendo en clase y se entregan los diagramas UML 2.x completos en la fecha de vencimiento en forma individual. El TP se resuelve en múltiples iteraciones. Una vez diseñados los subsistemas, cada uno resolverá algunos subsistemas y se integrará en clase.

## Modalidad de resolución de las prácticas y TP en clase.

Todos los problemas que siguen, salvo los mas simples, tienen aplicación en el ambiente de comunicaciones y en la construcción de un Sistema Operativo. Al distribuirlos sobre una red la solución va a ser distinta y tendrá que contemplar los nuevos paradigmas que impone una red con computadoras cuyo comportamiento es asincrónico.

Todos los programas se deben resolver usando procesos (NO SE ADMITEN THREADS: ya que no se puede distribuir en diferentes computadoras), y cada proceso, si es producto de un **fork**, se carga con un **exec** de algún tipo.

**Recuerde**: no usar funciones con buffer para mostrar información en pantalla (*printf, cin, cout, o cualquier instrucción de entrada/salida que empiece con f)*, ya que puede perderse la secuencia real de acciones y el entrelazado de ejecución de los distintos procesos. Debe usar el *system call* **write** sobre el *file descriptor* **stdout** (ver texto con fondo gris en el ejemplo).

### Uso eficiente del fork seguido de exec (repaso de Sistemas Operativos: Linux)

Un proceso existente puede crear un nuevo proceso usando la función fork

```
#include <unistd.h>  pid_t fork(void);
```

Devuelve: 0 en el proceso hijo, el process ID of hijo en el proceso padre, 1 si hay error

El nuevo proceso creado por el fork se denomina el proceso hijo. Esta función se invoca una sola vez y retorna dos veces. La diferencia en cada retorno es que el valor que devuelve para el hijo es 0, en cambio, para el padre es el process ID del hijo. No hay ninguna función para obtener los process ID de los procesos hijos. La razón por la cual el fork devuelve 0 en el hijo, es porque un proceso solo puede tener un proceso padre y siempre se puede obtener process ID del padre por medio de una llamada a getppid.

Ambos procesos siguen ejecutando la instrucción que sigue al fork(). El hijo es una copia del padre. Por ejemplo, el hijo obtiene una copia de espacio de datos del padre, heap y stack. Padre e hijo no comparten las mismas áreas de memoria. Padre e hijo comparte el mismo segmento de texto.

Las implementaciones actuales no hacen una copia completa de los datos del padre y del heap, ya que en general, un fork es seguido inmediatamente por una exec. En cambio, se usa la técnica llamada copy-on-write (COW). Estas regiones se comparten entre el padre y el hijo y tienen sus protecciones cambiadas a read-only. Si uno de los procesos trata de cambiar estas regiones, el kernel hace una copia de esta parte de la memoria (o sea de una página en el sistema de memoria virtual).

Según el manual de Linux, **fork**() usa “copy-on-write pages”, de este modo la única penalidad en la cual incurre es el tiempo y memoria necesarios para duplicar las tablas del padre y crear un nueva “task structure” para el hijo.

En nuestro caso, que queremos que el proceso hijo ejecute un programa diferente al programa padre, el hijo hace un exec después que retorna del fork. Cuando un proceso llama a una de las funciones exec (hay 6 funciones diferentes), el proceso se reemplaza por el nuevo programa y

este programa empieza su ejecución en la función main. El process ID no cambia, ya que NO se crea un nuevo proceso, se cambia el texto, datos, heap y stack en el proceso hijo.

El proceso hijo hereda del proceso padre:

- los archivos abiertos
- Real user ID, real group ID, effective user ID, effective group ID.
- group IDs suplementarios
- Process group ID
- Session ID
- la terminal de control
- Los set-user-ID y set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask y dispositions
- El close-on-exec flag para cualquier file descriptor abierto (open)
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

Las diferencias entre padre e hijo son

- El valor de retorno del `fork`
- Los process IDs son diferentes
- Los dos procesos tiene diferente parent process IDs: el parent process ID del hijo es el de padre; el parent process ID del padre no cambia
- En el hijo, los valores de `tms_utime`, `tms_stime`, `tms_cutime` y `tms_cstime` se ponen en 0.
- File locks que puso el padre no son heredados por el hijo.
- Alarmas pendientes se borran en el hijo.
- El conjunto de signals pendientes es un conjunto vacío en el hijo.

## Ejemplo de uso de fork y exec y de la estructura de un programa en "C":

```
/* Parte de un programa que crea los procesos para resolver el problema.
* Descripcion:      describir la funcion del programa
* Sintaxis:  como se invoca el programa y sus parametros
* Usa: funciones propias que invoca
*/
#include    "xx.h"    /* include que contiene las constantes
                         definiciones de tipos y bibliotecas */

/* instrucciones del programa principal */

int main (int argc, char** argv) {
 int  parametro;         /* parametro a pasar al proceso hijo */
 unsigned childpid;      /* pid del hijo */
 char mostrar[80];       /* mensaje para mostrar en pantalla */
```

```
 char *pname;           /* nombre del programa */

 static char el_parametro[15];/*parámetro string para el proceso hijo */

 pname = argv[0]; /* obtiene el nombre del programa en ejecucion */
/*
 *    Instrucciones con todas las inicializaciones
 *     y creación de los IPC
 */
      . . .
/*
 *                CREACION DE UN PROCESO HIJO
 */
 /* se muestra un mensaje en la pantalla indicando el contenido del
    parametro que se pasa al programa invocado con la exec en el hijo */
   sprintf (mostrar,"%s: parametro que va a recibir el proceso %d\n",
           parametro);
   write(fileno(stdout), mostrar, strlen(mostrar));

/* convertir el parametro de integer a string */
  sprintf(el_parametro, "%d\n",parametro);            /*

/* Si entre el fork y la exec no se usa ninguna variable
 * del proceso padre, Linux usa COW.
 */
 if ((childpid = fork()) < 0) {/* se crea el proceso hijo */
      perror("Error en el fork"); /* muestra el mensaje de error que
                                  corresponde al contenido del errno */
      exit(1);
      }
 else if (childpid == 0)  {
/*
 *           PROCESO HIJO (child)
 */
/*
/* se reemplaza el ejecutable del proceso padre por el nuevo programa
*/
  execlp("./pepe", "pepe", el_parametro, (char *)0);
    /* si sigue es que no se ejecuto correctamente el comando execlp */
  perror("Error al lanzar el programa pepe");
  exit(3);
  }
 /*
  *          PROCESO PADRE, sigue ejecutando
  */
 /* siguen las instrucciones del proceso principal */
. . .
} /* fin del programa */
```

# Repaso de semáforos.

Cuando se comparten datos entre *threads o hilos de control* que corren en el mismo espacio de direcciones o entre **procesos** que comparten un área de memoria (*shared memory*) se requiere algún mecanismo de sincronización para mantener la consistencia de esos datos compartidos. Si dos *threads* o dos procesos simultáneamente intentan actualizar una variable de un contador global es posible que sus operaciones se intercalen entre si, de tal forma que el estado global no se actualiza correctamente. Aunque ocurra una vez en un millón de accesos, los programas concurrentes deben coordinar sus actividades, ya sea de sus *threads* o con otros procesos usando "algo" mas confiable que solo confiar en que no ocurra porque la interferencia es rara u ocasional. Los semáforos se diseñaron para este propósito.

Un semáforo es similar a una variable entera, pero es especial en el sentido que está garantizado que sus operaciones (incrementar y decrementar) son atómicas. No existe la posibilidad que el incremento de un semáforo sea interrumpido en la mitad de la operación y que otro *thread* o proceso pueda operar sobre el mismo semáforo antes que la operación anterior esté completa. Se puede incrementar y decrementar el semáforo desde múltiples *threads* y/o procesos sin interferencia.

Por convención, cuando el semáforo es cero, está "bloqueado" o "en uso". Si en cambio tiene un valor positivo está disponible. Un semáforo jamás tendrá un valor negativo.

Los semáforos se diseñaron específicamente para soportar mecanismos de espera eficientes. Si un *thread* o un proceso no puede continuar hasta que ocurra algún cambio, no es conveniente que ese *thread* o proceso esté ciclando hasta que se verifique que el cambio esperado ocurrió (*busy wait* o espera activa). En este caso se pueden usar un semáforo que le indica al proceso o *thread* que está esperando por ese evento, cuando puede continuar. Un valor distinto de cero indica que continúa, un valor cero significa que debe esperar. Cuando el *thread* o proceso intenta decrementar (***wait***) un semáforo que no está disponible (tiene valor cero), espera hasta que otro lo incremente (***signal***) que indica que el estado cambió y que le permite continuar.

En general, los semáforos se proveen como ADT (*Abstract Data Types*) por un paquete específico del sistema operativo en uso. Por consiguiente, como todo ADT solo se pueden manipular las variables por medio de la subrutinas, funciones o métodos de la interfaz. Por ejemplo, en Java son ***SemaphoreWait*** y ***SemaphoreSignal*** para la sincronización de sus *threads*. No hay una facilidad general estándar para sincronizar *threads* o procesos, pero todas son similares y actúan de manera similar.

Históricamente, **P** es un sinónimo para ***SemaphoreWait***. **P** es la primera letra de la palabra *prolagen* que en holandés es una palabra formada por las palabras *proberen* (probar) y *verlagen* (decrementar). **V** es un sinónimo para ***SemaphoreSignal*** y es la primera letra de la palabra *verhogen* que significa incrementar en holandés.

En el curso crearemos una ADT para los semáforos con los métodos **P** o ***Wait*** y **V** o ***Signal***.

## P (Semaforo s)

Las primitivas a usar se basan en las definiciones clásicas de semáforos dadas por Dijkstra.

**P(s)** Operación **P** sobre un semáforo s. Conocida además como *down* o *wait* equivale al siguiente seudocódigo:

```
while  (s==0) <bloquear>
s--
```

El paquete controla los *threads*/procesos que están bloqueados sobre un semáforo en particular y los bloquea hasta que el semáforo sea positivo. Muchos de los paquetes garantizan un comportamiento sobre una cola FIFO para el desbloqueo de los *threads/*procesos para evitar inanición (*starvation*). Este es el caso de los semáforos que proveen los sistemas basados en UNIX.

## V (Semaforo s)

**V(s)** Operación **V** sobre un semáforo s. Conocida además como *up o signal* equivale al siguiente seudocódigo:

```
s++
<liberar un thread/proceso bloqueado en s>
```

El *thread/*proceso liberado se encola para ejecución y correrá en algún momento dependiendo de las decisiones del *scheduler* (planificador) del S.O.

## ValorSemaforo (Semaforo s) (NO EXISTE)

Una particularidad sobre semáforos es que no existe una función para obtener el valor de un semáforo. Solo se puede operar sobre el semáforo con P y V. No es útil obtener el valor de un semáforo ya que no hay garantía que el valor no haya sido cambiado por otro proceso/*thread* desde el momento que se pidió el valor y lo procesa el programa que lo solicitó.

## Uso del semáforo

La llamada a **P** (***SemaphoreWait***) es una especie de *checkpoint*. Si el semáforo está disponible (valor positivo), se decrementa el valor del semáforo y la llamada finaliza y continúa el proceso/*thread*. Si el semáforo no está disponible (está en cero) se bloquea el proceso/*thread* que hizo la llamada hasta que el semáforo esté disponible. Es necesario entonces que una llamada **P** en un proceso/*thread* esté balanceada con una llamada **V** en este o en otro proceso/*thread*, para que el semáforo esté disponible nuevamente.

## Semáforos Binarios

Un semáforo **binario** solamente puede tener valores 0 o 1. Son los semáforos usados para exclusión mutua cuando se accede a memoria compartida. Es una estrategia de *lock* para serializar el acceso a datos compartidos. Se lo conoce como semáforo ***mutex***.

Es muy importante que los semáforos se inicialicen con los valores correctos para que el sistema comience a funcionar. Además, se debe tener cuidado que siempre un semáforo no disponible, vuelva a estar disponible posteriormente. Es muy importante al serializar el acceso a una sección critica, que solo se bloquee el acceso mientras se opera con las variables de la sección crítica y se libere el semáforo lo antes posible.

## Semáforos generales (o "counting")

Un semáforo **general** puede tomar cualquier valor no negativo y se usa para permitir que simultáneamente múltiples tareas (procesos/*threads*) puedan ingresar a una sección crítica. Se usan también para representar la cantidad disponible de un recurso (*counting*).

**No usaremos estos semáforos en el curso,** porque se pueden simular con un semáforo *mutex* y una variable contador en memoria compartida. Los semáforos binarios son mas simples para distribuir.

## ADTs para semáforos binarios en "C" (IPC System V)

### *a) con un semáforo por arreglo (posición 0):*

```
/*
 *  ADT para semaforos: semaforos.h
 *  definiciones de datos y funciones de semáforos
 *
 *  Created by Maria Feldgen on 3/10/12.
 *  Copyright 2012 Facultad de Ingenieria U.B.A. All rights reserved.
 */
int inisem(int, int);
int getsem(int);
int creasem(int);
int p(int);
int v(int);
int elisem(int);
/*      Funciones de semaforos
 *    crear el set de semaforos (si no existe)
 */
int creasem(int identif) {
    key_t clave;
    clave = ftok(DIRECTORIO, identif);
    return( semget(clave, 1, IPC_CREAT | IPC_EXCL | 0660));
          /* da error si ya existe */
}
/*    adquirir derecho de acceso al set de semaforos existentes
 */
int getsem(int identif){
    key_t clave;
    clave = ftok(DIRECTORIO, identif);
    return( semget(clave, 1, 0660));
}
/*    inicializar al semáforo del set de semaforos
 */
int inisem(int semid,  int val){
 union semun {
      int              val;    /* Value for SETVAL */
      struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
      unsigned short  *array;  /* Array for GETALL, SETALL */
      struct seminfo  *__buf;  /* Buffer for IPC_INFO(Linux specific)*/
     } arg;
     arg.val = val;
     return( semctl(semid, 0, SETVAL, arg));
```

```
}
/*    ocupar al semáforo  (p) WAIT
 */
int p(int semid){
      struct sembuf oper;
      oper.sem_num = 0;        /* nro. de semáforo del set */
      oper.sem_op =  -1;       /* p(sem) */
      oper.sem_flg = 0;
      return (semop (semid, &oper, 1));
}
/*    liberar al semáforo  (v) SIGNAL
 */
int v(int semid){
      struct sembuf oper;
      oper.sem_num = 0;        /* nro. de semáforo */
      oper.sem_op = 1;         /* v(sem) */
      oper.sem_flg = 0;
      return (semop (semid, &oper, 1));
}
/*     eliminar el set de semaforos
 */
int elisem(int semid){
      return (semctl (semid, 0, IPC_RMID, (struct semid_ds *) 0));
}
```

*b) con un arreglo de semáforos:*

```
/*
 *  semaforosMultiples.h
 *  Primitivas para la operacion con un arreglo de semaforos (ADT)
 *
 *  Created by Maria Feldgen on 3/10/12.
 *  Copyright 2012 Facultad de Ingenieria U.B.A. All rights reserved.
 *
 *  definiciones de datos y funciones de semaforos
 */
int inisem(int, int, int);
int getsem(int, int);
int creasem(int, int);
int p(int, int);
int v(int, int);
int elisem(int);
/*      Funciones de semaforos
 *    crear el set de semaforos (si no existe)
 */
int creasem(int identif, int cantsem){
    key_t clave;
    clave = ftok(DIRECTORIO, identif);
    return( semget(clave, cantsem, IPC_CREAT | IPC_EXCL | 0660));
                                /* da error si ya existe */
}
/*    adquirir derecho de acceso al set de semaforos existentes
 */
int getsem(int identif, int cantsem){
    key_t clave;
    clave = ftok(DIRECTORIO, identif);
```

```
    return( semget(clave, cantsem, 0660));
}
/*    inicializar al semáforo del set de semaforos
 */
int inisem(int semid, int indice, int val){
      union semun {
       int              val;    /* Value for SETVAL */
       struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
       unsigned short  *array;  /* Array for GETALL, SETALL */
       struct seminfo  *__buf;  /* Buffer for IPC_INFO(Linux specific)*/
      } arg;
      arg.val = val;
      return( semctl(semid, indice, SETVAL, arg));
}
/*    ocupar al semáforo  (p) WAIT
 */
int p(int semid, int indice){
      struct sembuf oper;
      oper.sem_num = indice;         /* nro. de semáforo del set */
      oper.sem_op =  -1;             /* p(sem) */
      oper.sem_flg = 0;
      return (semop (semid, &oper, 1));
}
/*    liberar al semáforo  (v) SIGNAL
 */
int v(int semid, int indice){
      struct sembuf oper;
      oper.sem_num = indice;         /* nro. de semáforo */
      oper.sem_op = 1;               /* v(sem) */
      oper.sem_flg = 0;
      return (semop (semid, &oper, 1));
}
/*     eliminar el set de semaforos
 */
int elisem(int semid){
      return (semctl (semid, 0, IPC_RMID, (struct semid_ds *) 0));
}
```

## ADTs para semáforos binarios (IPC POSIX)

POSIX tiene operaciones para crear, inicializar y realizar operaciones con semáforos. POSIX tiene dos tipos de semáforos: con nombre y sin nombre.

### *Semáforos con nombre*

Los semáforos se identifican por un nombre en forma similar a los semáforos System V y tienen persistencia en el *kernel*. Abarcan todo el sistema y hay una cantidad limitada de ellos activos en un determinado momento. Proveen sincronización entre procesos no relacionados, relacionados o entre *threads* (ídem System V).

```
/*  funciones de la biblioteca para la operacion con semaforos POSIX
 *  semaphore.h
 */
char *identif;     /* nombre del semaforo, debe comenzar con /*/
int inicial;        /* valor inicial del semaforo */
sem_t *semaforo;   /* semaforo creado */
int resultado;      /* resultado de la operación sobre el semaforo */
/*
 *    crear el semaforos e inicializarlo (si no existe)
 */

    semaforo = sem_open(identif,O_CREAT | O_EXCL ,0644, inicial));
                     /* da error si ya existe */
/*          adquirir derecho de acceso al semaforo existente
 */
    semaforo = sem_open(identif,0,0644, 0);
                    /* da error si no existe */);
/*    ocupar al semaforo  (p) WAIT
 */
    resultado = sem_wait(semaforo);

/*    liberar al semaforo  (v) SIGNAL
 */
    resultado = sem_post(semaforo);

/*    terminar de usar el semaforo en un proceso
 */
    resultado = sem_close(semaforo);

/*    eliminar el semaforo del sistema
 */
    resultado = sem_unlink(identif);
```

A diferencia de los semáforos System V, hay una función para cerrar la referencia al semáforo y otra función para eliminarlo del sistema. El semáforo se elimina inmediatamente, pero se destruye en el sistema cuando todos los *thread*/procesos lo cerraron.

### *Semaforos sin nombre*

Un semáforo sin nombre se guarda en una región de memoria compartida entre todos los threads de un proceso (por ejemplo, una variable global) o procesos relacionados (similar a una shared memory).

No requiere usar la función `sem_open`. Se reemplaza por `sem_init`

```
sem_t *semáforo;
int sem_init(sem_t *sem, int pshared, unsigned  value); }
```

`pshared :` El semáforo se comparte entre, si el argumento es: = 0 entre *threads*,
> 0 entre procesos

Para destruir este tipo de semáforos se usa la función `sem_destroy`.

**En la materia usaremos semáforos con nombre.**

A continuación se analizará cada uno de los problemas clásicos de concurrencia.

# Problemas clásicos de concurrencia. Características y ejercicios.

## Implementación en la materia:

Para poder aplicar los paradigmas de programación distribuida sobre los programas desarrollados usando paradigmas de programación concurrente usaremos solamente:

- **Lenguajes C++ o C** en la distribución en Linux elegida en clase.
- **Semáforos binarios** (No se usarán semáforos generales o counting, ya que no es posible emular su comportamiento en un ambiente distribuido)
- **Procesos** (Los *threads* no se pueden distribuir, ya que comparten el área de memoria del proceso)

A modo de repaso de los conceptos de concurrencia, se explica brevemente cada problema y en algunos casos, la solución que debe adoptarse y ejercicios que combinan ese problema con los problemas vistos anteriormente.

Los problemas clásicos que se describen son:

1. Exclusión Mutua
2. Productor/Consumidor con sus variantes:
    a. con buffer infinito generalizado a N productores y M consumidores, y con la condición que:
        i. cada consumidor consume un elemento distinto del buffer.
        ii. cada consumidor consume todos los elementos del buffer (todos los consumidores consumen todos los elementos).
    b. con buffer acotado para 1 productor y 1 consumidor.
    c. con buffer acotado generalizado para N productores y M consumidores, y con la condición que
        i. cada consumidor consume un elemento distinto del buffer.
        ii. cada consumidor consume todos los elementos del buffer
3. Secuencia de procesos
4. Barrera
5. Rendezvous
6. Lectores/Escritores (sin inanición)
    a. con prioridad a los lectores
    b. con prioridad a los escritores

## 1.- Exclusión Mutua.

En este problema hay dos o mas *threads*/procesos que actualizan información en un área compartida. La solución del problema debe garantizar que los *threads*/procesos se serialicen de tal forma que la operación de actualización de una o mas variables del área compartida sea atómica. Si un *thread*/proceso está actualizando el área, cualquier otro que quiera realizar una operación sobre el área compartida quedará bloqueado esperando que el anterior termine su operación.

La implementación clásica de la solución de este problema es por medio de un área de memoria compartida que contiene a las variables compartidas y cuyo acceso es controlado por un semáforo binario (***mutex***). La característica relevante de la solución es que el *thread*/proceso que hace P( ) del semáforo, cuando el semáforo lo permite, entra a la sección crítica, hace la actualización y hace V( ) del semáforo para liberar el acceso al área compartida. EL MISMO thread/proceso HACE LA SECUENCIA P( )…V( ) DEL MUTEX.

| *Proceso/thread 1* | *Proceso/thread 2* |
|---|---|
| `P(mutex)`<br>`operación sobre la sección crítica`<br>`V(mutex)` | `P (mutex)`<br>`operación sobre la sección crítica`<br>`V(mutex)` |

El semáforo mutex es binario, lo cual garantiza que el P(mutex) solo permite continuar ejecutando si se puede decrementar (o sea debe estar en 1), por consiguiente solo un proceso puede realizar la operación sección crítica por vez.

Las áreas de memoria compartida de IPC (shared memory) no pueden generalizarse creando una ADT como se hizo para los semáforos, ya que son directamente dependientes del tipo de datos que va a contener esa área de memoria y no es conveniente definirlo como un conjunto de bytes.

### Ejemplo de codificación de shared memory System V en "C" usando procesos:

En el archivo Ventas.h está definidad la estructura de los productos.

```
#define SHM 200
#define MUTEX 201
#define DIRECTORIO "/home/mariafeldgen/ventas"
/*
 * El directorio para los IPC debe ser el path completo desde la raíz
 * según el Linux varia el path desde la raiz a un usuario y dentro de
 * este directorio se crea un directorio con su apellido
 * y un subdirectorio con el ejercicio y version.
 */

typedef struct      {          /* Producto a vender */
       int codigo;             /* Codigo */
       char denominacion[25]; /* Denominacion */
       float precio;
       int cantidad;           /* stock o cantidad vendida */
} PRODUCTO;

typedef struct {               /* stock de productos */
       int cantProductos;      /* Cantidad de productos a vender */
       PRODUCTO prod[25];      /* tabla de productos */
```

```
} STOCK;
```

En el programa que lanza los procesos.

```
#include "Ventas.h"
int main(int argc, char *argv[]) {
        char mostrar[120];      /* mensaje para mostrar en pantalla */
        char *pname;            /* nombre del programa */
        int pid;
        int mutex, shmid;       /* file descriptor de los IPCs */
        STOCK *shmem;
        key_t clave;
        char archivo[]="Productos.txt";  /* contiene los productos */

        FILE*       fp;   /* archivo y estructura de los productos */
        char primeraLinea[80];
        /*
         *   crear la shared memory en la cual se mantiene el estado
         *   hacer el attach e inicializar las variables compartidas
         */
        clave = ftok(DIRECTORIO,SHM);
        if ((shmid1 = shmget (clave, sizeof(STOCK), IPC_CREAT|IPC_EXCL|0660))
                                                              == -1) {
             perror ("Lanzador: error al crear la shared memory ");
             exit (1);
        }

        if ((shmem = (STOCK *) shmat(shmid1,0,0)) == (STOCK *) -1 ) {
             perror ("Lanzador: error en el attach a shared memory ");
             exit (1);
        }
        /*
         * Cargar los datos en la memoria compartida
         */
        if( (fp=fopen(archivo,"r") ) == NULL){
             sprintf (mostrar,"%s (%d): NO se puede abrir el archivo con
los productos %s\n", pname, pid_pr, archivo);
             write(fileno(stdout), mostrar, strlen(mostrar));
             exit(1);
        }
        else{
             fgets(primeraLinea,80,fp); /* Ignorar la primera linea. */
             int i=0;
             while(!feof(fp)){   /* descargar todo el archivo */
                   fscanf(fp,"%d %s %f %d",
                            &shmem->prod[i].codigo,
                            shmem->prod[i].denominacion,
                            &shmem->prod[i].precio,
                            &shmem->prod[i].cantidad);
                   i++;
                   shmem->cantProductos++;
             }
             fclose(fp);
        }
```

```
        /*
         *    crear e inicializar el IPC semaforo mutex
         */
        if ((mutex = creasem (MUTEX)) == -1) /* Semáforo exclusion mutua */ {
             perror ("Lanzador: error al crear el semaforo mutex");
             exit (1);
        }
        inisem (mutex, 1);                  /* inicializarlo */
        /*
. . .
```

En el programa que actualiza la shared memory

```
#include "Ventas.h"
int main(int argc, char *argv[]) {
        key_t clave;       /* clave que devuelve ftok */
        char mostrar[200];/* mostrar acciones en pantalla */
        char pname;        /* nombre del programa en ejecución
        int pid;           /* process id del proceso */
        int shmid;         /* file descriptor shm */
        STOCK *shmem;      /* puntero a la shared memory */
        pname = argv[0];  /* nombre de este programa */
        pid_cl = getpid();/* pid del programa que esta corriendo */
        /*
         *   acceder a la memoria compartida
         */
        clave1 = ftok(DIRECTORIO,SHM);
        if ((shmid = shmget (clave1,sizeof (STOCK),0660)) == -1) {
             perror ("Vendedor: error en el get a la shared memory ");
             exit(1);
        }
        if ((shmem = (STOCK *) shmat(shmid,0,0)) == (STOCK *) -1 ) {
             perror ("Vendedor: error en el attach a shared memory ");
             exit(1);
        }
        /*
         *    obtener el mutex
         */
        if ((mutex = getsem (MUTEX)) == -1) {
                   /* IPC para exclusion mutua */
             perror ("Vendedor: error en el get del semaforo mutex");
             exit(1);
       }
       . . .
        /*
         * actualizar stock
         */
         p(mutex);
         shmem->prod[producto].cantidad -= cantidad;
         v(mutex);
        . . .
```

En el programa que destruye los IPC y obliga a terminar a todos los procesos

```
#include "Ventas.h"
int main(int argc, char *argv[]){
        int shmid;
        int mutex;
        key_t clave;
        STOCK a;

        clave = ftok(DIRECTORIO,SHM);
        shmid = shmget (clave, sizeof(a), 0660);
        mutex = getsem (MUTEX);
        shmctl(shmid,IPC_RMID,(struct shmid_ds *)  0);
        elisem(mutex);
        . . .
```

## Ejercicio Nº 1:

Hay 4 robots que agregan pilas a dispositivos móviles y los activan, dejándolos en una plataforma. Los 4 robots operan independientemente. Cada dispositivo tarda un tiempo al azar en activarse y emite una frecuencia. El primer robot que detecta le emisión, lo saca de la plataforma y lo pone en la zona de testing (donde hay personas que los distribuyen). La plataforma tiene una capacidad máxima de dispositivos en espera de que se complete su proceso de activación. Si está llena, no se pueden agregar mas dispositivos, hay que esperar que algún dispositivo abandone la plataforma. El sistema se activa cuando el supervisor de turno, pone los cajones con los dispositivos sin pilas y las pilas al pie de cada robot, e informa el total de dispositivos y de pilas que agregó (los dos valores no son múltiplos). Cada robot tiene sus propios cajones, no se comparten con los otros robots.

Cada robot termina cuando no hay mas dispositivos o no hay mas pilas.

### *Para la implementación:*

Hay un proceso inicial que inicializa los IPCs y las restricciones del sistema, crea los procesos robots y le asigna el total de dispositivos y pilas. Cada proceso robot simula dos acciones que se ejecutan al azar:

1) sacar un dispositivo del cajón, generando un dispositivo con un tipo al azar (1 al 10) al cual le pone 2 pilas y lo deja en la bandeja.
2) sacar un dispositivo activo de la bandeja y ponerlo en la zona de testing. Debe tomar en cuenta que el contador de dispositivos en la plataforma no puede ser negativo.

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*
  - *Haga el diagrama de clases completo.*

- *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*

- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*
  - *Escriba los programas del problema*
  - *Agregue un makefile para compilarlos.*
- *Requisitos y recomendaciones:*
  - *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
  - *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
  - *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

## 2.- Productor-Consumidor

### 2a) con buffer no acotado (infinito) (N a M).

#### 2ai) cada consumidor consume un elemento distinto.

En este problema hay dos o mas *threads/*procesos que intercambian información por medio de un buffer no acotado. Los productores agregan elementos al buffer y los consumidores los extraen en la misma secuencia y los muestran. Solamente los consumidores tienen una situación de bloqueo que ocurre cuando no hay elementos para extraer. El problema es hacerlos cooperar y bloquearlos eficientemente cuando es necesario.

La implementación clásica de este problema es por medio de colas del sistema, los productores encolan los elementos y los consumidores los desencolan. El sistema operativo bloque a los productores si la cola está llena y bloquea a los consumidores si la cola está vacía. No requiere semáforos para el acceso a la cola. La cola es FIFO, al igual que el acceso a la misma.

Para facilitar la distribución y determinar el destino de los mensajes es mas simple usar las colas de IPC System V que las colas de POSIX. El identificador del mensaje de la cola en System V permite trabajar con múltiples receptores conectados a la misma cola y simplifica el diseño inicial, y nos permite una distribución mas simple.

Las colas de IPC **no pueden generalizarse** creando una **ADT generalizada** como se hizo para los semáforos, ya que cada cola es directamente dependiente del tipo de mensaje que traslada y de su tamaño. La generalización o sea la ADT tiene los mismos métodos o funciones que las que provee la cola IPC System V. **NO LA GENERALICE, QUE HACE MAS DIFICIL EL TRABAJO POSTERIOR de distribución.**

## Ejemplo de codificación de colas System V en "C" usando procesos:

En el archivo Ventas.h tiene la estructura de datos del mensaje.

```
#define COLAVTA 202
#define DIRECTORIO "/home/mariafeldgen/ventas"

typedef struct      {            /* compra de un cliente */
       long int destinatario;    /* identificador para el consumidor */
       int producto;             /* código del producto a comprar */
} COMPRA;
```

En el programa que lanza los procesos productores, consumidores, etc.

```
#include "Ventas.h"
int main(int argc, char *argv[]){
       int compras;      /* File descriptor IPC de la cola */
       key_t clave;      /* clave que devuelve ftok */
       char mostrar[128];/* mostrar acciones en pantalla */
       char pname;       /* nombre del programa en ejecución
       int pid;          /* process id del proceso */
       /*
        *   crear una cola COMPRAS nueva
        *  da error si hay una cola con la misma clave
        */
       clave = ftok(DIRECTORIO,COLAVTA);
       if ((compras = msgget (clave,IPC_CREAT|IPC_EXCL|0660)) == -1) {
            perror("LanzadorVtas: error al crear la cola COMPRAS ");
            exit (1);
       }
```

En el programa productor que envía mensajes a la cola

```
#include "Ventas.h"
int main(int argc, char *argv[]) {
       int compras;      /* File descriptor IPC de la cola */
       key_t clave;      /* clave que devuelve ftok */
       char mostrar[200];/* mostrar acciones en pantalla */
       char pname;       /* nombre del programa en ejecución
       int pid;          /* process id del proceso */
       COMPRA msg;       /* mensaje a poner en la cola */
       long int nro = 1; /* identificador para el destinatario */
       pname = argv[0];  /* nombre de este programa */
       pid_cl = getpid();/* pid del programa que esta corriendo */
       /*
        *   adherir a una cola COMPRAS creada
        *   da error si no hay una cola creada con esa clave
        */
       clave = ftok(DIRECTORIO,COLAVTA);
       if ((compras = msgget (clave, 0660)) == -1) {
            perror("Cliente: error en el get de la cola COMPRAS ");
            exit (1);
       }
. . .
       /*
```

```
         * arma el registro con la compra
         */
        msg.destinatario = nro;
        msg.producto = producto;
        sprintf (mostrar,"\n%s %d (%d): Compra producto: %d\n",
                 pname, nro, pid_cl, msg.producto);
        write(fileno(stdout), mostrar, strlen(mostrar));
        /*
         * poner el mensaje en la cola compras
         */
        if (msgsnd(compras, (COMPRA *)&msg, sizeof(msg)-sizeof(long),0) == -1)
        {
             perror ("Cliente: Error en el envio cola compras ");
             exit(1);
             }
. . .
```

En el programa consumidor que recibe los mensajes de la cola

```
#include "Ventas.h"
int main(int argc, char *argv[])
{
        int compras;      /* File descriptor IPC de la cola */
        key_t clave;      /* clave que devuelve ftok */
        char mostrar[200];/* mostrar acciones en pantalla */
        char pname;       /* nombre del programa en ejecución
        int pid;          /* process id del proceso */
        COMPRA msg;       /* mensaje a leer de la cola */
        long int id_a_recibir; /* identificador para el destinatario */

        pname = argv[0];  /* nombre de este programa */
        pid_cl = getpid();/* pid del programa que esta corriendo */
        /*
         *   adherir a una cola COMPRAS creada
         *   da error si no hay una cola creada con esa clave
         */
        clave = ftok(DIRECTORIO,COLAVTA);
        if ((compras = msgget (clave, 0660)) == -1) {
             perror("Vendedor: error en el get de la cola COMPRAS ");
             exit (1);
        }
…
/*
 * esperar cualquier mensaje de algun cliente sobre la cola COMPRAS
 */
        if (msgrcv(compras,(COMPRA *)&msg,sizeof(COMPRA)-sizeof(long),0,0)==-1)
             if (errno == EINVAL || errno == EIDRM) {
        /* verifica si se destruyeron los IPC (para terminar) */
               sprintf (mostrar,"%s %d (%d): TERMINA\n", pname,
                                vendedor, pid_ve);
               write(fileno(stdout), mostrar, strlen(mostrar));
               exit(0);
             }
             else {
```

```
                perror ("Vendedor: Error en la recepcion de la compra) ");
                exit(1);
               }
        sprintf (mostrar,"-->%s %d (%d): Compra producto: %d\n", pname,
             vendedor, pid, msg.producto);
        write(fileno(stdout), mostrar, strlen(mostrar));
```

...

Si se espera un identificador de mensaje específico

```
/*
 * esperar mensajes con id = 1 de algun cliente sobre la cola COMPRAS
 */
        if (msgrcv(compras, (COMPRA *)&msg, sizeof(COMPRA)-sizeof(long), 0,
                                              id_a_recibir) == -1)
             if (errno == EINVAL || errno == EIDRM) {
        /* verifica si se destruyeron los IPC (para terminar) */
               sprintf (mostrar,"%s %d (%d): TERMINA\n", pname,
                                 vendedor, pid_ve);
               write(fileno(stdout), mostrar, strlen(mostrar));
               exit(0);
             }
             else {
              perror ("Vendedor: Error en la recepcion de la compra) ");
              exit(1);
             }
        sprintf (mostrar,"-->%s %d (%d): Compra producto: %d\n", pname,
             vendedor, pid, msg.producto);
        write(fileno(stdout), mostrar, strlen(mostrar));
```

...

En el programa que destruye todos los IPC y obliga a terminar a todos los procesos

```
#include "Ventas.h"
int main(int argc, char *argv[])
{
        int compras;      /* File descriptor IPC de la cola */
        key_t clave;      /* clave que devuelve ftok */
        char mostrar[200];/* mostrar acciones en pantalla */
        char pname;       /* nombre del programa en ejecución
        int pid;          /* process id del proceso */
        /*
         *   adherir a una cola COMPRAS creada
         *   sin verificar errores.
         */
        clave = ftok(DIRECTORIO,COLAVTA);
        compras = msgget (clave1,0660);
        /*
         *   destruir la cola COMPRAS
         *   sin verificar errores.
         */
        msgctl(compras, IPC_RMID, NULL);
```

...

## Ejercicio Nº 2:

Copie la implementación del ejercicio anterior. En este caso los dispositivos sin pilas por medio de una única cinta. Los robots disponen de las pilas. Simule a los dispositivos por medio de procesos. Los procesos dispositivos se crean por un proceso inicial. Cada dispositivo que es atendido por un robot en particular, permanece una cierta cantidad de tiempo inactivo (determinado por el disposito) y luego se dirige a los robots para que lo saquen de la plataforma. Cada proceso dispositivo termina cuando lo sacan de la plataforma. Muestre en la pantalla la interacción del dispositivo con el sistema y el dispositivo usa una cola para ser atendido por el robot tanto para que le ponga las pilas, como para sacarlo de la plataforma. Los robots sacan los dispositivos activos de la plataforma y los ponen en una cinta de acuerdo a su tipo de dispositivo y al final de la cinta hay tantos robots como tipos distintos de dispositivos hay, que sacan el dispositivo de la cinta, lo apagan y lo ponen en un contenedor con capacidad suficiente para almacenar la producción de un día. El sistema se comunica con el dispositivo por medio de mensajes sobre una cola.

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*
  - *Haga el diagrama de clases completo.*
  - *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*
- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*
  - *Escriba los programas del problema*
  - *Agregue un makefile para compilarlos.*
- *Requisitos y recomendaciones:*
  - *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
  - *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
  - *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

**Nota**: Verifique que cada vez que lo corre, la salida por pantalla será diferente porque los procesos no se planifican exactamente en el mismo orden.

### 2aii) cada consumidor consume todos los elementos del buffer

En este problema hay dos o mas *threads/*procesos que intercambian información por medio de un buffer no acotado. Los productores agregan elementos al buffer y los consumidores deben extraer todos los elementos del buffer y los muestran. Solamente los consumidores tienen una situación de bloqueo que ocurre cuando no hay elementos para extraer. El problema es hacerlos cooperar y bloquearlos eficientemente cuando es necesario.

La implementación clásica de este problema, ya que es no acotado, es por medio de colas del sistema. Hay que tener en cuenta que en una cola FIFO si se desencola un elemento por definición se elimina de la cola. Por lo tanto, los productores deben encolar los elementos repetidos tantas veces como consumidores se encuentran en el sistema y los mensajes deben estar identificados por consumidor, tal que ningún consumidor desencole dos veces el mismo elemento. Recuerde, los consumidores son concurrentes y no secuenciales. Si la cola está llena, los productores se bloquean y si la cola está vacía, se bloquean los consumidores. No requiere semáforos para el acceso a la cola. La cola es FIFO y la implementación mas simple es usando colas System V con tipo para identificar a cada consumidor. El consumidor solo desencola los elementos del tipo que le corresponden.

## Ejercicio Nº 3:

Hay 5 procesos productores concurrentes que generan ordenes de producción de computadoras. Hay 3 procesos consumidores distintos y concurrentes en el almacén, que se encargan de buscar los componentes: Un proceso consumidor se encarga de los discos, otro se encarga de los procesadores y otro del motherboard. Tome en cuenta que los 3 procesos consumidores y los productores son concurrentes (no hay serialización).

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*
  - *Haga el diagrama de clases completo.*
  - *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*
- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*
  - *Escriba los programas del problema*
  - *Agregue un makefile para compilarlos.*
- *Requisitos y recomendaciones:*

- *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
- *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
- *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

**Nota**: Verifique que cada vez que lo corre, la salida por pantalla será diferente porque los procesos no se planifican exactamente de la misma forma.

### 2b) Productor/consumidor con buffer acotado (con x elementos máximo) (1 : 1).

En este problema hay dos *threads/*procesos (un productor y un consumidor) que intercambian información por medio de un buffer de **longitud fija**. El productor llena el buffer con datos siempre que hay por lo menos un lugar disponible para hacerlo. El consumidor lee los datos del buffer, si hay por lo menos uno, y lo muestra. Ambos *threads/*procesos tienen una situación de bloqueo. El productor se bloquea cuando el buffer está lleno y el consumidor se bloquea cuando el buffer está vacío. El problema es hacerlos cooperar y bloquearlos eficientemente solamente cuando es necesario.

Para este problema en general se usan semáforos *counting* o generalizados. El valor cero significa bloqueado y cualquier valor positivo significa disponible. En este curso lo vamos a resolver con semáforos binarios y variables contador.

El consumidor empieza a leer del buffer en la posición indicada por la variable **punLeer** y el productor escribe desde la posición indicada por la variable **punEsc**. No se requieren *locks* para proteger estas variables ya que no son compartidas y son locales al proceso que la requiere. Los semáforos aseguran que el productor solamente escriba en la posición indicada por **punEsc** cuando hay por lo menos un lugar disponible, de la misma forma, el consumidor lee a partir de la posición indicada por **punLeer,** si hay elementos no leídos.

Se necesitan dos semáforos: un semáforo para indicar que en el buffer hay por lo menos un lugar ocupado (**lleno**) y otro para indicar que hay por lo menos un lugar vacío (**vacio**). Hay una variable **contador** que indica cuantos lugares están ocupados.

Los semáforos **mutex**, **vacio** y **lleno** son binarios.

El productor **incrementa el contador** cada vez que agrega un nuevo número y solamente pone el **semáforo lleno** cuando verifica que el **contador estaba en cero** antes que de escribir el elemento actual en el buffer. Si luego de escribir el elemento el **contador está en el máximo** de elementos que puede contener el buffer, espera sobre el **semáforo vacio**.

El consumidor **decrementa el contador** cada vez que consume un elemento y solamente pone el **semáforo vacio** cuando verifica que el **contador estaba en el máximo** antes de consumir el elemento. Si luego de consumir el elemento el **contador está en cero**, espera sobre el **semáforo lleno**. Recuerde, son semáforos binarios, cuyos únicos valores posibles son 0 y 1.

## Ejercicio Nº 4:

Si tiene un proceso productor que ingresa personas desde una cola a una sala de espera para 25 personas. Cuando llega el cable carril (consumidor) sube las personas de a una por vez y se va cuando cargó 50 personas o no hay mas personas esperando. El cable carril sube la montaña y

cuando llega a la estación tope, baja a las personas y repite el proceso para bajar. Simule las personas por procesos, que suben a la montaña y se quedan una cierta cantidad de tiempo y luego bajan. Un proceso genera n personas que se activan en diferentes intervalos de tiempo. La persona sube y baja de la montaña y se va.

Hay un buffer compartido de longitud fija = 5 posiciones. El buffer se encontrará en una memoria compartida (*shared memory*).

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*
  - *Haga el diagrama de clases completo.*
  - *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*
- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*
  - *Escriba los programas del problema*
  - *Agregue un makefile para compilarlos.*
- *Requisitos y recomendaciones:*
  - *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
  - *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
  - *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

**Nota**: Verifique que cada vez que lo corre, la salida por pantalla será diferente porque los procesos no se planifican exactamente de la misma forma.

## 2c) Productor/consumidor con buffer acotado (N a M).

### 2ci) cada consumidor consume un elemento distinto.

En este problema hay mas de dos *threads/*procesos (N productores y M consumidores) que intercambian información por medio de un buffer de **longitud fija**. Los productores llena el buffer con datos siempre que hay por lo menos un lugar disponible para hacerlo. A diferencia del problema anterior, los productores deben compartir la variable **punEsc**, para saber cual es la

próxima posición que pueden escribir. Ídem los consumidores con la variable **punLeer**. El consumidor lee los datos del buffer, si hay por lo menos uno, y lo muestra. Ambos *threads/*procesos tienen una situación de bloqueo. Un productor se bloquea cuando el buffer está lleno y un consumidor se bloquea cuando el buffer está vacío.

El problema adicional que se presenta es que no todos los productores van a estar bloqueados al mismo tiempo y no todos los consumidores van a estar bloqueados al mismo tiempo. Por lo tanto, cada productor tiene su propio semáforo vacío y cada consumidor tiene su propio semáforo vacío. Hace falta indicar si está o no esperando sobre su semáforo al proceso que puede habilitarlo, para evitar que se transforme en un semáforo *counting*.

**Sugerencia**: implementar con arreglos de semáforos de System V.

## Ejercicio Nº 5:

Generalizar el ejercicio anterior para n productores que llenan la sala que agregan elementos al mismo buffer y m consumidores (cable carriles) que consumen un elemento distinto por vez.

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*
  - *Haga el diagrama de clases completo.*
  - *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*
- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*
  - *Escriba los programas del problema*
  - *Agregue un makefile para compilarlos.*
- *Requisitos y recomendaciones:*
  - *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
  - *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
  - *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

**Nota**: Verifique que cada vez que lo corre, la salida por pantalla será diferente porque los procesos no se planifican exactamente de la misma forma.

### 2cii) cada consumidor consume todos los elementos

Ídem anterior, pero cada consumidor debe consumir todos los elementos del buffer. Los procesos hacen V() de los semáforos que le corresponden solamente si tienen la certeza que todos los procesos consumidores consumieron todos los elementos, antes de avisar que hay un lugar disponible a los productores, si el buffer estaba lleno. Por lo tanto, cada consumidor debe contar cuantos elementos consumió del total de elementos.

## Ejercicio Nº 6:

Un sistema que toma muestras de agua en un puerto tiene 6 tomas de agua (productores) en diferentes puntos del puerto. Cada productor deposita su muestra en un porta muestras de 6 contenedores, indicando la fecha y hora de la muestra. Cada una de las muestras es analizada por 8 analizadores independientes de sustancias contaminantes. Los analizadores trabajan simultáneamente (concurrentemente) sobre las muestras y cada muestra debe ser analizada por los todos los analizadores, mostrando el resultado antes de ser descartada, si no contiene contaminates o se guarda (en un archivo), si las contiene. Se quiere saber cuales de las muestras contienen contaminantes y su fecha y hora de toma de la muestra y cuantas muestras se analizaron.

Es un problema con múltiples productores (los procesos que toman las muestras) y múltiples consumidores (los procesos que analizan cada muestra). Es acotado, porque se dispone de un porta muestras de 5 contenedores. Se requieren los mismos semáforos que para los ejercicios anteriores, para la coordinación de cada productor y de cada consumidor. Además cada consumidor necesita saber si ya analizó una determinada muestra al recorrer el porta muestras y saber si cuando se encuentra frente a una nueva muestra cuando vuelve a la primera posición. Todos los procesos tardan diferente cantidad de tiempo en hacer su tarea.

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*
  - *Haga el diagrama de clases completo.*
  - *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*
- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*
  - *Escriba los programas del problema*

- *Agregue un makefile para compilarlos.*

- *Requisitos y recomendaciones:*
  - *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
  - *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
  - *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

**Nota**: Verifique que cada vez que lo corre, la salida por pantalla será diferente porque los procesos no se planifican exactamente de la misma forma.

## 3.- Secuencia de threads/procesos.

En este problema, hay n *threads*/procesos en secuencia, cada *thread*/proceso inicia su procesamiento cuando terminó el *thread*/proceso anterior. La secuencia de procesamiento se reinicia cuando el último *thread*/proceso terminó. Este es un problema que se resuelve con semáforos binarios: cada *thread*/proceso espera sobre su semáforo para procesar, procesa y pone el semáforo del *thread*/proceso siguiente en la secuencia. Es diferente a la exclusión mutua, en este caso un *thread*/proceso hace P( ) de su propio semáforo y el *thread*/proceso anterior hace V() del semáforo de su sucesor para habilitarlo.

**Sugerencia**: implementar con arreglos de semáforos de System V.

Ejemplo para dos procesos sincronizados:

| *Proceso/thread 1* | *Proceso/thread 2* |
|---|---|
| `P(semáforo del proceso/thread 1)`<br>`procesar`<br>`V(semáforo del proceso/thread 2)` | `P (semáforo del proceso/thread 2)`<br>`procesar`<br>`V(semáforo del proceso/thread 1)` |

## Ejercicio Nº 7:

Una planta de fabricación de automóviles tiene el siguiente esquema de fabricación. A partir de una orden de fabricación, la primera estación, fabrica cada uno de los chasis para la cantidad de autos que indica la orden, cada chasis fabricado pasa a la estación de pintura, se pinta y pasa a la estación de secado y de ahí a la estación de armado, en el cual se agrega el motor. En la siguiente estación, le agregan las ruedas y los interiores y en la siguiente estación se completa el auto con las puertas, capó y baúl. Cuando esta estación termina, el auto pasa a la playa de autos terminados. Cada estación tiene un cierta cantidad de materiales/piezas para hacer su trabajo. Cuando se queda sin stock, solicita a un repositor que busca el material solicitado del almacén y lo deja en la estación que lo solicitó. Recuerde, que debe sincronizar los procesos, ya que no hay forma de almacenar un auto en construcción entre estación y estación. Una estación que tiene su parte del auto lista, debe esperar que la estación siguiente lo tome, antes de fabricar el siguiente. Las ordenes de producción se leen de un archivo. Un proceso inicial lanza todos los procesos correspondientes a las estaciones con su stock inicial. La fabricación se simula con un slip y el consumo de materiales/piezas es aleatorio (no siempre consume lo mismo).

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*
  - *Haga el diagrama de clases completo.*
  - *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*
- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*
  - *Escriba los programas del problema*
  - *Agregue un makefile para compilarlos.*
- *Requisitos y recomendaciones:*
  - *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
  - *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
  - *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

**Nota**: Verifique que cada vez que lo corre, la salida por pantalla será diferente porque los procesos no se planifican exactamente de la misma forma.

## 4.- Barrera.

En este problema un conjunto de *threads*/procesos procesan independientemente cada uno sobre un elemento distinto. Recién pueden operar sobre un nuevo elemento si todos los *threads*/procesos terminaron con el procesamiento de su elemento.

Para resolver este problema con semáforos: cada *thread*/proceso tiene su semáforo sobre el cual hace P( ). El último *thread*/proceso que termina hace V( ) de todos los semáforos de los *threads*/procesos que están bloqueados. Se requiere una variable **contador** compartida, para determinar que ese *thread*/proceso fue el último en terminar. Cada proceso que termina suma 1 en el contador, el *thread*/proceso que detecta que contador contiene el total de procesos, borra el contador y habilita a todos los procesos a seguir procesando (V( ) de todos los semáforos). Recuerde, el *interleave* de los *threads*/procesos depende del SO y no de la secuencia de lanzamiento de los mismos.

**Sugerencia**: implementar con **arreglos de semáforos de System V**, es la solución mas adecuada.

Para resolver este **problema con colas**: cada thread/proceso debe esperar por tantos mensajes como threads/procesos están procesando. Se sugiere implementar con colas de System V cuyos mensajes tienen tipo que un solo proceso puede desencolar.

### Ejercicio Nº 8 (Implementación con semáforos binarios):

Hay n embotelladoras sobre una única cinta transportadora. Cada embotelladora llena una botella vacía y espera que terminen todas las otras. La última que termina activa la cinta, para que un nuevo conjunto de botellas vacías estén frente a las embotelladoras. El proceso es continúo y no para nunca.

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*
  - *Haga el diagrama de clases completo.*
  - *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*
- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*
  - *Escriba los programas del problema*
  - *Agregue un makefile para compilarlos.*
- *Requisitos y recomendaciones:*
  - *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
  - *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
  - *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

**Nota**: Verifique que cada vez que lo corre, la salida por pantalla será diferente porque los procesos no se planifican exactamente de la misma forma.

## 5.- Rendezvous o punto de encuentro.

Es similar a la barrera, pero en este caso, cuando todos los *threads*/procesos concurrentes terminan, ejecuta otro proceso en secuencia. El *thread*/proceso de la barrera que termina último habilita a este proceso, que habilita nuevamente a los *threads*/procesos de la barrera. Se puede

resolver eficientemente con semáforos o con colas.

## Ejercicio Nº 9 usando semáforos y shared memory solamente:

Se debe simular la asociación de átomos de hidrogeno con átomos de oxigeno para formar moléculas de agua. Cada átomo se representa por un proceso diferente. Necesitamos asociar dos procesos de hidrogeno con uno de oxigeno para crear agua, luego de lo cual los tres procesos terminan. Suponga que tiene N átomos de hidrogeno y M átomos de oxigeno y que N no es par y $N \neq M * 2$, o sea que deben quedar átomos de hidrogeno sin usar y átomos de oxigeno sin usar.

Este es un ejemplo de un "rendezvous" o "punto de encuentro". Cuando están listos los dos procesos hidrogeno, empieza a trabajar el proceso oxigeno, que cuando termina, permite que los dos procesos hidrogeno terminen y otros hidrogeno comiencen a generar una nueva molécula.

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*
  - *Haga el diagrama de clases completo.*
  - *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*
- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*
  - *Escriba los programas del problema*
  - *Agregue un makefile para compilarlos.*
- *Requisitos y recomendaciones:*
  - *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
  - *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
  - *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

**Nota**: Verifique que cada vez que lo corre, la salida por pantalla será diferente porque los procesos no se planifican exactamente de la misma forma.

## 6.- Lectores y Escritores.

En este problema un conjunto de *threads*/procesos deben acceder a variables de memoria compartida en algún momento, algunos de estos *threads*/procesos para leer información y otros para modificar (escribir) información, con la restricción que ningún *thread*/proceso de lectura o escritura puede acceder a la memoria compartida si otro *thread*/proceso está escribiendo. En particular, está permitido que varios *threads*/procesos que leen información accedan simultáneamente.

Una solución sería proteger la memoria compartida con un *mutex* de exclusión mutua, o sea, no hay *thread*/procesos que puedan acceder a la memoria compartida al mismo tiempo. Sin embargo, esta solución no es óptima, porque si R1 y R2 quieren leer, se estaría secuenciando su acceso, cuando expresamente se autoriza el acceso simultáneo.

Por este motivo, el problema se divide en dos tipos diferentes:

- **Prioridad a los lectores**: se agrega la restricción que ningún lector espera si la memoria compartida está usada para leer
- **Prioridad a los escritores**: si hay muchos lectores o se requiere información actualizada, se agrega la restricción que si hay un escritor esperando, tiene prioridad sobre los lectores que están esperando.

Sin embargo, las soluciones planteadas resultan en inanición (*starvation*). Si los lectores tienen prioridad, puede ser que los escritores esperen indefinidamente, si siempre hay lectores para leer. Ídem para el segundo tipo, si siempre hay escritores para escribir y estos tienen prioridad, los lectores esperarán indefinidamente (inanición).

Si agregamos la restricción que no se permite que un *thread*/proceso se vea afectado por inanición, o sea, la espera por el acceso a la memoria compartida está acotada en el tiempo. Se puede implementar de la siguiente forma: el tipo de *thread*/proceso con menor prioridad accede a la memoria compartida cada cierta cantidad (parámetro) de accesos de los *threads*/procesos con prioridad y luego vuelve a concederle la prioridad al tipo correspondiente, o sea se cuenta cuantos threads/procesos con prioridad esperó a que terminaran.

### Ejercicio Nº 10 (Lectores con prioridad):

En una casa de cambio que informa a sus clientes el precio del dólar blue. Los clientes consultan permanentemente el valor de la moneda y los operadores del mercado actualizan el precio de la moneda cuando se producen cambios. Simule a los clientes como lectores y a los operadores del mercado como escritores. Implemente sin inanición y verifique que aunque la cantidad de clientes es muy superior a los operadores, los operadores pueden modificar el valor de la moneda.

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*

- *Haga el diagrama de clases completo.*
- *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*

- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*
  - *Escriba los programas del problema*
  - *Agregue un makefile para compilarlos.*
- *Requisitos y recomendaciones:*
  - *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
  - *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
  - *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

**Nota**: Verifique que cada vez que lo corre, la salida por pantalla será diferente porque los procesos no se planifican exactamente de la misma forma.

## Ejercicio Nº 11 (Escritores con prioridad):

Ídem anterior, pero los operadores tienen prioridad sobre los clientes.

Se pide:

- *Análisis de requerimientos del problema:*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso.*
  - *Haga el diagrama de clases preliminar (solamente entidades)*
- *Análisis:*
  - *Haga el diagrama de comunicaciones de cada caso de uso.*
  - *Haga el diagrama de clases completo.*
  - *Haga el diagrama de secuencia de cada caso de uso y cada escenario.*
- *Diseño:*
  - *Marque los objetos activos y pasivos en el diagrama de secuencia.*
  - *Cree las interfaces provistas y requeridas para cada clase de un objeto activo.*
  - *Haga un diagrama de secuencia incluyendo las interfaces y los IPC como objetos del problema planteado.*
- *Implementación*
  - *Escriba un programa inicial para inicializar los IPC y para lanzar los procesos, tal como se explicó en clase.*
  - *Escriba un programa final para destruir los IPC y parar los procesos.*

- *Escriba los programas del problema*
- *Agregue un makefile para compilarlos.*

- *Requisitos y recomendaciones:*
  - *Si los datos para el intercambio en el problema son números use secuencias de números, si son de varios tipos, use archivos distintos para cada proceso.*
  - *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
  - *Las corridas deben ser significativas. No haga pruebas con solamente uno o dos valores. Ud. debe asegurar que no hay un busy wait, ni starvation (inanición) ni un deadlock.*

**Nota**: Verifique que cada vez que lo corre, la salida por pantalla será diferente porque los procesos no se planifican exactamente de la misma forma.

Fechas de vencimientos de cada parte de la práctica:

| Ejercicio | Diagramas | Programación |
|---|---|---|
| 1-ExMutua | 27/3/2014 | 10/4/2014 |
| 2-PC no acotado y 3-PC todos los elementos | 3/4/2014 | 10/4/2014 |
| 4-PC acotado 1 a 1 y 5- PC acotado N-M | 10/4/2014 | 24/4/2014 |
| 6-PC acotado todos los elementos y 7-Secuencia | 10/4/2014 | 24/4/2014 |
| 8-Barrera y 9- Rendezvous | 24/4/2014 | 8/5/2014 |
| 10-LE prioridad lectores y 11-LE prioridad escritores | 24/4/2014 | 8/5/2014 |

Los diagramas se entregan en papel en clase el día del vencimiento en un folio con su nombre y apellido o se escanean y se guardan en un zip y se suben al google con su apellido y ejercicio.