Práctica 1: Repaso de Concurrencia: Paradigmas e IPC

Entrega en un zip con su apellido y nro de práctica. Ejemplo ***Perez01.zip.***
Fecha de vencimiento: 17 de setiembre de 2012.

Cuando se comparten datos entre *threads o hilos de control* que corren en el mismo espacio de direcciones o entre procesos que comparte 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 como una variable entera, pero es especial en el sentido que está garantizado que sus operaciones (incrementar y decrementar) sean 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 (*bussy wait* o espera activa). En este caso se pueden usar un semáforo que le indica al proceso o *thread* que está en espera 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 puede manipular las variables por medio de la rutinas 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* y 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* o 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 proceso bloqueado en s>
```

El *thread* o proceso liberado se encola para ejecución y correrá en algún momento dependiendo de las decisiones del *scheduler* 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 cambiado por otro proceso/*thread* desde el momento que se pidió el valor y lo obtuvo el programa.

Uso del semaforo

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 o *thread* que hizo la llamada hasta que el semáforo esté disponible. Es necesario entonces que una llamada P en un proceso o *thread* esté balanceada con una llamada **V** en este o en otro proceso o *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 los conoce como semáforo ***mutex***.

La inicialización con los valores correctos de un semáforo cuando se crea es importante para que el sistema comience a funcionar. Además, se debe tener cuidado que si el semáforo pasa a no estar disponible, siempre 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 critica y se libere el semáforo lo antes posible.

Semáforos generales

Un semáforo **general** puede tomar cualquier valor no negativo y se usa para permitir que simultáneamente múltiples tareas puedan ingresar a una sección critica. 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 (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 semaforo 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 semaforo  (p) WAIT
 */
int p(int semid)
{
     struct sembuf oper;
     oper.sem_num = 0;        /* nro de semaforo del set */
     oper.sem_op =  -1;       /* p(sem) */
     oper.sem_flg = 0;
     return (semop (semid, &oper, 1));
}
/*
 *    liberar al semaforo  (v) SIGNAL
 */
int v(int semid)
{
     struct sembuf oper;
     oper.sem_num = 0;        /* nro de semaforo */
     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 semaforo 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 semaforo  (p) WAIT
 */
int p(int semid, int indice)
{
      struct sembuf oper;
      oper.sem_num = indice;         /* nro de semaforo del set */
      oper.sem_op =  -1;             /* p(sem) */
      oper.sem_flg = 0;
      return (semop (semid, &oper, 1));
}
```

```
/*
 *    liberar al semaforo  (v) SIGNAL
 */
int v(int semid, int indice)
{
      struct sembuf oper;
      oper.sem_num = indice;         /* nro de semaforo */
      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 tiene persistencia en el kernel. Abarcan todo el sistema y una cantidad limitada de ellos activos en un determinado momento. Provee 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 procesos /threads 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 *semaforo;
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.

Productor-Consumidor con buffer no acotado (1 a 1).

En este problema hay dos o mas *threads* o procesos que intercambian información por medio de un buffer no acotado. Los productores agrega elementos en secuencia y los consumidores leen los datos en la misma secuencia y los muestran. Solamente los consumidores tienen una situación de bloqueo que ocurre cuando no hay datos para leer. 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. 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, al igual que el acceso a la misma.

Ejercicio Nº 1:

El uso clásico de un semáforo binario es que solo un proceso por vez pueda acceder al recurso. Supongamos que tenemos una variable compartida que es la cantidad de tickets que faltan vender y queremos coordinar el acceso por múltiples procesos. En este caso es un problema de exclusión mutua, el semáforo asegura que solo un proceso acceda y modifique a la variable en cada momento.

Un proceso inicial crea una cierta cantidad de procesos vendedores (> 2) que venden los tickets y un conjunto de procesos clientes (> 2) que compran los tickets. Los clientes pueden comprar entre 1 y 4 tickets, de a un ticket por vez. El cliente envía un monto de dinero al vendedor, que verifica que el monto sea mayor o igual que el precio del ticket. Si el dinero es mayor o igual al precio del ticket y hay tickets disponibles, el vendedor envía al cliente el Nro de ticket vendido, el vuelto y el aviso que la compra fue exitosa. Si no es suficiente el monto, devuelve un aviso de "monto insuficiente" y si no hay mas tickets, devuelve "no quedan mas tickets".

Cada cliente extrae los datos de la compra de un archivo, para simular las distintas situaciones.

Se pide:

1. *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
2. *Escriba un programa inicial para inicializar los IPCs y para lanzar los procesos, tal como se explicó en clase.*
3. *Escriba un programa final para destruir los IPCs y parar los procesos vendedor y cliente (si hay alguno pendiente).*
4. *Escriba los programas del problema y agregue un makefile para compilarlos.*
5. *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.*
6. *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
7. *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 (innación) 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 2: Productor-Consumidor con buffer no acotado. Cada consumidor debe consumir todos los elementos.

Hay varios (n) procesos productores que generan una etiqueta de identificación de medicamentos, que acompaña a un recipiente esteril. Los procesos consumidores son 3

procesos que deben llenar el recipiente con el medicamento basados en la etiqueta, el proceso que pone la etiqueta en el recipiente y el proceso que imprime el texto de la caja para contener el recipiente. Los 3 procesos consumidores y los productores son concurrentes.

1. *Explique muy brevemente ¿Cuáles son las diferencias con el ejercicio anterior?*
2. *Haga el diagrama de secuencia correspondiente.*

Ejercicio 3: Productor-Consumidor con buffer acotado (N-M).

En este problema hay dos *threads* o procesos que intercambian información por medio de un buffer de longitud fija. El productor llena el buffer con datos siempre que haya lugar para hacerlo. El consumidor lee los datos del buffer, si hay, y los muestra. Ambos *threads* o 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.

Ejemplo con un productor y un consumidor. El productos escribe números corelativos en un buffer de 5 posiciones. El consumidor los lee y los muestra.

Hay un buffer compartido de longitud fija. El buffer se encontrará en una memoria compartida (*shared memory*). Para la exclusión mutua en el acceso a la memoria compartica será necesario un semáforo **mutex.**

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

Se necesitan dos semáforos uno para indicar que en el buffer por lo menos hay 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 elemento, pone el **semáforo lleno** cuando verifica que el **contador estaba en cero** antes que de escribir el elemento actual. 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, 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**.

Ambos procesos terminan cuando el consumidor terminó de consumir el nro 99 y se deben destruir los IPC asociados.

1. *Haga el diagrama de secuencia correspondiente.*

Ejercicio 5 (Productores y consumidores con buffer acotado N-M)

Generalizar para n productores que agregan elementos al mismo buffer y n consumidores que consumen un elemento distinto por vez.

1. *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*

Ejercicio 6:

Idem anterior, pero los consumidores consumen todos los elementos del buffer.

Un sistema de toma muestras de agua de un rio, tiene 5 tomas de agua, que recorren el rio de costa a costa, tomando las muestras, que se depositan en un portamuestras de 5 contenedores. Cada una de las muestras es analizada por 6 analizadores independientes de sustancias tóxicas. Los analizadores trabajan simultáneamente sobre la muestra y la muestra debe ser analizada por los 6 analizadores, mostrando el resultado antes de ser descartada, si no contiene sustancias tóxicas o guardada si las contiene. Se quiere saber cuales de las muestras contiene sustancias tóxicas 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 portamuestras de 5 contenedores. Se requieren los mismos semáforos que para el ejercicio 3, para la coordinación de cada productor y de cada consumidor. Además cada consumidor necesita saber si ya analizó una determinada muestra al recorrer el portamuestras y volver a la primera muestra que analizó en esa vuelta. Todos los procesos tardan diferente cantidad de tiempo en hacer su tarea.

1. *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*

Secuencia de procesos.

Ejercicio 7:

Para un productor y un consumidor. El consumidor consume el buffer cuando está completo. Una vez consumido todo el buffer el productor vuelve a llenar el buffer y así sucesivamente.

1. *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*

Barrera.

Ejercicio 8 (Implementación con semáforos binarios):

Hay n procesos que deben generar mensajes numerados con el mismo número. Por ejemplo, todos los procesos generan un mensaje con el nro 1 en memoria compartida. Cuando todos terminaron de generar el mensaje, pasan a generar el mensaje siguiente con el nro 2 y asi sucesivamente hasta llegar a 100.

1. *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*

Ejercicio 9 (Implementación con colas):

Ídem anterior pero implementado con colas. Explique brevemente, las diferencias en la solución.

1. *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*

Rendezvous

Otro problema simple de concurrencia es la simulación de la asociación de átomos de hidrogeno con átomos de oxigeno para formar moléculas de agua. Cada átomo se representa por un *thread* o proceso diferente. Necesitamos asociar dos *threads*/procesos de hidrogeno con uno de oxigeno para crear agua, luego de lo cual los tres *threads*/procesos terminan.

Necesitamos varios semáforos binarios y contadores para la sincronización de los *threads*/procesos.

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

Ejercicio Nº 10.

Resuelva el problema de rendezvous en forma óptima usando semáforos y shared memory solamente.

1. *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
2. *Escriba un programa inicial para inicializar los IPCs y para lanzar los procesos, tal como se explicó en clase.*

3. *Escriba un programa final para destruir los IPCs y parar los procesos vendedor y cliente (si hay alguno pendiente).*
4. *Escriba los programas del problema e incluya un makefile para compilarlos.*
5. *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.*
6. *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
7. *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 (innación) ni un deadlock.*

Ejercicio Nº 11.

Resuelva el mismo problema anterior usando colas.

1. *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
2. *Escriba un programa inicial para inicializar los IPCs y para lanzar los procesos, tal como se explicó en clase.*
3. *Escriba un programa final para destruir los IPCs y parar los procesos vendedor y cliente (si hay alguno pendiente).*
4. *Escriba los programas del problema e incluya un makefile para compilarlos.*
5. *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.*
6. *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
7. *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 (innación) ni un deadlock.*

Ejercicio Nº 12. Combinando varios paradigmas

Para un gimnasio que se encuentra en el centro de un gran parque arbolado se solicita un sistema automatizado.

En la primera etapa se quiere automatizar y controlar a los socios que ingresan y egresan. Los socios del gimnasio ingresan y egresan al parque por alguna de sus 3 puertas. El socio entra y sale por cualquiera de las puertas, no tiene una puerta prefijada. En cada puerta se cuenta cuantos socios hay en el parque en cada momento.

El gimnasio tiene una capacidad máxima para 200 personas. Si hay 200 personas en el parque no se permite el acceso de nuevos socios. El socio que llega y encuentra la puerta cerrada, se va. Recién puede ingresar un socio cuando algún socio abandone el parque por alguna de sus puertas.

Dado que el gimnasio se encuentra a varios kilómetros de las entradas al parque, en cada entrada y en gimnasio hay una sala de espera para esperar al bus que los lleva al

gimnasio/puerta de salida. Las salas son amplias y tienen capacidad para 200 personas.

El bus tiene una capacidad máxima de 25 pasajeros. Cuando el bus llega a la puerta/gimnasio bajan las personas que viajan en el bus, y suben las personas que quieren viajar al gimnasio/puerta de salida. Hay un bus por puerta de E/S.

El bus parte cuando tiene 25 pasajeros o no hay mas pasajeros esperando. El bus sale sin pasajeros si no hay pasajeros esperando en la zona de partida pero únicamente si hay pasajeros esperando en la otra parada.

La solución del problema debe mostrar la cantidad de socios que hay en el parque. Cada vez que ingresa un socio y debe mostrar los datos del socio (nombre y nro. de socio) y mostrar cuando se mueve (entra o sale de las puertas del gimnasio, esta esperando en la sala de espera, viaja en el bus al gimnasio o hacia la puerta.

Simule usando procesos para los socios, puertas, salas de espera, gimnasio y buses. No hay otros procesos. Simule los tiempos de espera por sleep o usleep usando valores random.

Se pide:

1. *Explique cuales de los paradigmas de concurrencia están presentes (exclusión mutua, serialización, rendezvous, productor consumidor y la variante (acotado o no acotado y consumidor consume un elemento cualquiera o consume todos los elementos).*
2. *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
3. *Escriba un programa inicial para inicializar los IPCs y para lanzar los procesos, tal como se explicó en clase.*
4. *Escriba un programa final para destruir los IPCs y parar los procesos vendedor y cliente (si hay alguno pendiente).*
5. *Escriba los programas del problema e incluya un makefile para compilarlos.*
6. *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.*
7. *Simule el tiempo de procesamiento con un sleep (usleep) con tiempo variable (use un generador de números al azar).*
8. *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 (innación) ni un deadlock.*