Práctica 1: Repaso de Concurrencia: Problemas clásicos e IPC

Modalidad de entrega de la práctica resuelta:

Las prácticas se resuelven en grupos de hasta 2 personas, sin embargo las entregas de ejercicios resueltos es individual y en las fechas de vencimiento estipuladas, para que las aclaraciones puedan ser personalizadas. 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 el grupo que no cumple con las entregas o faltan ejercicios en las entregas y no se informan los problemas, se tomará un parcial al grupo.

Si no asiste a clase, debe entregar los ejercicios pedidos en esa clase.

La entrega de los ejercicios en grupo se divide en dos etapas:
Cada uno de los integrantes del grupo entrega los ejercicios que resolvió en forma individual. Los diagramas en un folio o conjunto abrochado y los programas en un archivo.zip distinto.

1. Entrega de los **diagramas de secuencia de UML (Vto: 26 de marzo de 2013)**:
   - Los diagramas pueden ser manuscritos y en lápiz. No es necesario que use una herramienta de graficación UML. Los únicos diagramas que se piden corresponden a escenarios sin errores.
   - Indique el nro. de ejercicio al cual corresponde el diagrama y entréguelos en el orden de la práctica.
   - Se debe completar el primer formulario de **s1_113Practica_1Entregas.docx** que se acompaña como carátula de la entrega.
2. Entrega de los **programas que resolvió** y que corresponden a los diagramas de secuencia entregados. Si cambió los diagramas de secuencia vuelva a entregarlos: (**Vto: a definir en la clase**).
   - Se entregan en un zip cuyo nombre es grupo + Nº de grupo+apellido + Nº de práctica. Ejemplo ***grupo01perez.zip.*** Todos los programas se encuentran agrupados bajo el directorio que tiene el nombre del grupo (**grupo01**).
   - Cada programa tiene un subdirectorio diferente dentro del directorio, cuyo nombre corresponde al nombre del directorio del zip +apellido+ Nº ejercicio. Ejemplo: ***grupo01perez01.*** Dentro de este subdirectorio, los programas pueden tener cualquier nombre.
   - Se debe completar el primer formulario de **s1_113Practica_1Entregas.docx** que y agrégarlo al zip.

El ejercicio que se resuelve en clase, se entrega individualmente, la clase siguiente, siguiendo la misma modalidad de los ejercicios de clase y usando el segundo formulario de s1_113Practica_1Entregas.docx que se acompaña.

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 puede 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* 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 thread/proceso bloqueado en s>
```

El *thread* o 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 cambiado por otro proceso/*thread* desde el momento que se pidió el valor y lo procesó 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 los 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

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 (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 threads/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)`
`operación sobre la sección crítica`
`V(mutex)`
	`P (mutex)`
	`operación sobre la sección crítica`
	`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.

Ejercicio Nº 1:

Un museo dispone de varias puertas de ingreso/egreso de personas. Cada vez que una persona ingresa al museo por cualquiera de las puertas se incrementa una variable compartida en 1 persona mas en el sistema y cada vez que egresa, se descuenta una persona.

El proceso inicial crea un proceso Puerta por cada puerta del museo. Cada proceso Puerta genera personas a intervalos (al azar) que ingresan/egresan al museo. Debe tomar en cuenta que el contador de personas no puede ser negativo.

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

Ejercicio Nº 2:

Copie la implementación del ejercicio anterior y agregue un control para que solamente puedan ingresar al museo una máxima cantidad de personas y que cuando el museo cierra, no pueda ingresar ninguna persona y solo puedan salir personas. Todos los procesos Puerta deben terminar cuando se cierra el museo y no hay personas en el museo.

Para las personas que quieren ingresar al museo: Se deben generar personas si el museo está abierto, si está lleno deben esperan para ingresar, si el museo cierra, no se generan mas personas para ingresar ni entran las que estaban esperando.

- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado. Explique cuales cambios tuvo que aplicar.*
- *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 y agregue un makefile para compilarlos.*
- *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 es diferente porque los procesos no se planifican exactamente en el mismo orden.

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. 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 3:

Para un recital hay múltiples vendedores que concurrentemente venden los tickets para el recital. La cantidad de tickets disponibles y el precio de un ticket están en un área compartida. Cada cliente que compra un ticket es atendido por el primer vendedor disponible. 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".

El proceso inicial crea una cierta cantidad de procesos vendedor (>2) y una cierta cantidad de procesos cliente (>2) a intervalos variables. Modele la espera por el vendedor y el intercambio de información entre cliente y vendedor por medio de colas, y la cantidad de tickets por medio de memoria compartida. Cada cliente extrae los datos de la compra de un archivo, para simular las distintas situaciones.

- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
- *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 y agregue un makefile para compilarlos.*

- *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.

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 4:

Hay 5 procesos productores concurrentes que generan etiquetas de identificación de medicamentos y cada uno toma un recipiente estéril. Hay 3 procesos consumidores distintos y concurrentes. Un proceso consumidor llena el recipiente con el medicamento basado en la información de la etiqueta, otro proceso pega la etiqueta en el recipiente y el último imprime el texto de la caja del recipiente según la información de la etiqueta. Tome en cuenta que los 3 procesos consumidores y los productores son concurrentes (no hay serialización).

- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
- *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 y agregue un makefile para compilarlos.*
- *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 5:

Si tiene un proceso productor que escribe números correlativos en un buffer de 5 posiciones y un proceso consumidor que los lee y los muestra.

Hay un buffer compartido de longitud fija = 5 posiciones. El buffer se encontrará en una memoria compartida (*shared memory*). Ambos procesos terminan cuando el consumidor terminó de consumir el nro. 99 y se deben destruir los IPC asociados.

- *Haga el diagrama de secuencia correspondiente incluyendo los IPC como objetos del problema planteado.*

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 6:

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

- *Describa un escenario en el cual los semáforos lleno y vacío se transforman en counting, si solo se tiene un único semáforo lleno y un único semáforo vacío para todos los procesos.*
- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*

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 7:

Un sistema que toma muestras de agua de un rio, tiene 5 tomas de agua, que recorren el rio de costa a costa, tomando las muestras. Las muestras se depositan en un porta muestras de 5 contenedores. Cada una de las muestras es analizada por 6 analizadores independientes de sustancias tóxicas. Los analizadores trabajan simultáneamente (concurrentemente) 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 se guarda (en un archivo), si las contiene. Se quiere saber cuales de las muestras contienen 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 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.

- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
- *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 y agregue un makefile para compilarlos.*
- *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)`
`procesar`
`V(semáforo del proceso/thread 2)`
	`P (semáforo del proceso/thread 2)`
	`procesar`
	`V(semáforo del proceso/thread 1)`

Ejercicio Nº 8:

Sobre una cinta de una embotelladora de gaseosa, hay múltiples equipos que trabajan sobre la botella en la siguiente secuencia: por cada botella vacía, hay una embotelladora que llena la botella, luego pasa a la etiquetadora que le pone la etiqueta, luego pasa a la tapadora que le pone la tapa y finalmente el último equipo la pone en una caja (almacenadora) y pone una nueva botella vacía sobre la cinta y la secuencia se reinicia, si hay una botella vacía. La cantidad de botellas vacías es un parámetro del sistema que se ingresa cuando se inicia el proceso inicial, que lanza los procesos embotelladora, etiquetadora, tapadora y almacenadora.

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

- *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 y agregue un makefile para compilarlos.*
- *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.

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 9 (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 así sucesivamente hasta llegar a 100.

- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
- *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 y agregue un makefile para compilarlos.*
- *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 10 (Implementación con colas):

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

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

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 11 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.

- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
- *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 e incluya un makefile para compilarlos.*
- *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 12 usando colas:

Resuelva el mismo problema anterior usando colas.

- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado. Explique las diferencias con respecto al problema anterior.*

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 13 (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.

- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
- *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 e incluya un makefile para compilarlos.*
- *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 14 (Escritores con prioridad):

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

- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado. Explique los cambios con respecto al problema anterior.*

Combinación de problemas de concurrencia

Ejercicio Nº 15. Ejercicio a resolver en clase.

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

Se quiere automatizar y controlar el ingreso/egreso de las personas. Las personas pueden entrar y salir del parque por alguna de sus cuatro puertas. La persona entra y sale por cualquiera de las puertas, no tiene una puerta prefijada. En cada puerta se debe contar cuantas personas hay en el parque en cada momento. El zoológico tiene una capacidad máxima de 200 personas. Si hay 200 personas en el zoológico no se permite el acceso de nuevas personas al parque y se cierran las puertas de acceso. La persona que llega y encuentra la puerta cerrada, se va. Recién puede ingresar una persona cuando otra abandone el parque por alguna de sus puertas.

Dado que el zoológico se encuentra a varios kilómetros de las entradas al parque, en cada entrada y en el acceso al zoológico hay salas de espera para esperar al bus que los lleva al zoológico/puerta de salida. Las salas son amplias y tienen capacidad para 200 personas. Hay un bus distinto para cada puerta de E/S que hace el recorrido de la puerta al acceso del zoológico ida y vuelta.

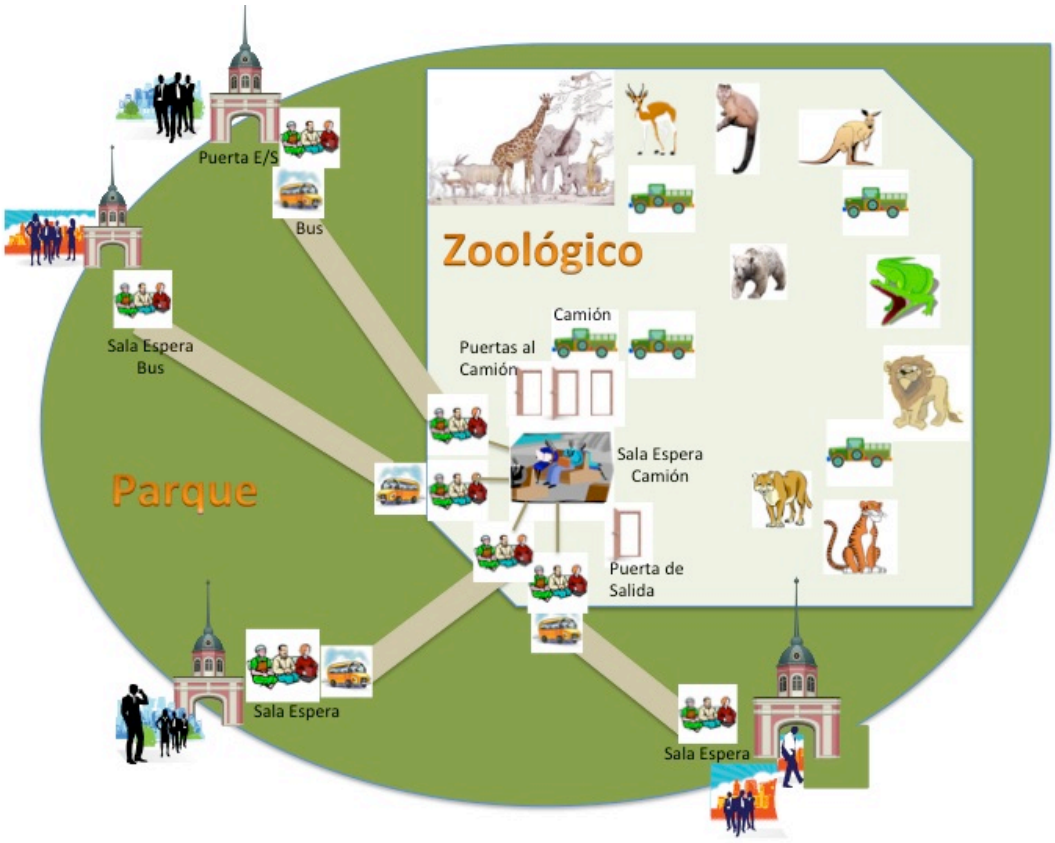
Un bus tiene una capacidad máxima de 25 pasajeros. Cuando el bus llega a la puerta/acceso al zoológico bajan las personas que viajan en el bus, y suben las personas que quieren viajar al zoológico/puerta de salida. Hay un bus por puerta de E/S.

El bus parte cuando tiene 25 pasajeros o cuando cargó por lo menos un pasajero y no hay mas pasajeros esperando. El bus sale sin pasajeros únicamente si no hay pasajeros esperando en la parada en la cual se encuentra y hay pasajeros esperando en la otra parada.

Una vez que la persona llega al acceso al zoológico ingresa a una sala de espera que tiene 3 puertas para acceder a la zona donde se encuentran los camiones de recorrida del zoológico que están protegidos porque los animales se encuentran sueltos en predios alambrados que se recorren por dentro. Las puertas de esta sala solamente están abiertas si hay por lo menos un camión esperando con espacio para cargar 40 personas. Se debe garantizar que hasta que todas las personas de la sala no hallan abordado el camión, ninguna otra persona pueda ingresar a la sala. El zoológico dispone de 5 camiones. Cuando el camión está lleno o subieron una o mas personas y no hay mas personas esperando, comienza con el recorrido por el zoológico que le insume media hora. Cuando el camión vuelve, las personas bajan y se dirigen a las salas de espera de los buses de salida del parque por una puerta de salida del zoológico. Recién cuando el camión está vacío se habilitan las puertas de acceso para que suban personas, si no estaban habilitadas previamente.

La solución del problema debe mostrar la cantidad de personas que hay en el parque. Cada vez que ingresa una persona: debe mostrar los datos de la persona (nombre) y mostrar cada vez que se mueve (entra o sale de las puertas del zoológico, está esperando en la sala de espera, viaja en el bus al zoológico o hacia la puerta, está esperando en la sala de espera del camión, viaja en el camión, etc.

Gráfico del parque con el zoológico



Simule usando procesos para las personas, puertas de E/S del parque, salas de espera del bus y del camión, camiones y buses. No hay otros procesos. Simule los tiempos de espera por sleep o usleep usando valores random.

- *Explique y justifique cuales de los problemas clásicos de concurrencia están presentes (de los que se explican en esta práctica).*
- *Haga un diagrama de secuencia incluyendo los IPC como objetos del problema planteado.*
- *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 e incluya un makefile para compilarlos.*
- *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.