Práctica 1: Construcción de una biblioteca de casos aplicando problemas clásicos de Concurrencia usando IPC

Modalidad de construcción de la biblioteca de casos entre todos.

Los ejercicios de esta práctica son ejercicios para diseñar en grupo y luego cada integrante programa una parte de la solución. Se entrega el diseño de la solución usando diagramas UML 2.x de las distintas etapas del ciclo de vida y luego la implementación. Toda la documentación (diagramas) y fuentes de los programas se usarán para luego resolver el TP y estarán disponibles para todos. La corrección de los ejercicios se hace en clase entre todos y en la fecha de vencimiento. Es importante que por lo menos uno de los integrantes del grupo esté presente en la corrección, para explicarle a los otros grupos la solución encontrada. La implementación de la solución puede tener problemas de funcionamiento que no pudo resolver el grupo, se tratará de encontrar como solucionar este problema entre todos. Las fechas de entregas de cada parte (diagramas e implementación) se determinará en la clase en la cual se empieza a desarrollar esta práctica, para coordinar la actividad conjunta de corrección.

Se trabajará en grupos de 2 personas y en clase se elegirán los ejercicios a resolver, tal de dividir la construcción de la biblioteca de casos entre todos. Una vez que cada ejercicio funcione adecuadamente, se incluirán en la biblioteca clasificados por los paradigmas que incluyen para ser una referencia de solución (nuestros patrones) para la solución de los distintos problemas del TP.

Los ejercicios son combinaciones de los distintos paradigmas de concurrencia por lo tanto no hay una única solución al problema. Como es la primera vez que hacen todos los diagramas y de un problema pequeño pero complejo (no complicado) y que puede presentar errores simples de programación pero dificiles de encontrar en las pruebas, se solicita que consulten las dudas tanto en clase, por e-mail o coordinen con los docentes para una consulta mas extensa fuera del horario de clases. En todos los casos de una consulta, para ayudar a los docentes a comprender cual es el duda o problema que no puede resolver, describa muy brevemente dónde tiene el problema o en el caso de programación, las pruebas que hizo y lo que “supone” que está ocurriendo.

Cuando se estén programando los últimos ejercicios de esta práctica se comenzará con el TP, que se desarrollará parcialmente en clase, en especial lo que corresponde a la diagramación y pruebas de programas. Dado que el equipamiento del laboratorio tiene poca memoria y es lento y que programación es una actividad individual, se sugiere que programen en sus propias computadoras. El TP se resuelve en múltiples iteraciones. Una vez diseñados los subsistemas, cada uno resolverá algunos subsistemas y se integrará en clase. Una explicación mas completa de las técnicas a usar se acompañará con el enunciado del TP.

Recomendaciones para la resolución de las prácticas y el TP en clase y en grupo.

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 la distribución y la red con computadoras cuyo comportamiento es asincrónico.

Dado que queremos analizar y diseñar cada problema una única vez, usaremos aquellos paradigmas de concurrencia que son los mas apropiados para resolver el problema (generaremos patrones de solución concurrentes en una máquina). Esta técnica nos permite encontrar una solución en el ambiente distribuido o sea aplicar los patrones distribuidos adecuados para cada patrón concurrente, lo cual nos posibilita encontrar una solución al problema, estudiando el comportamiento del patrón concurrente. La idea es transformar un problema complejo en múltiples problemas mas simples y mostrar su interrelación para distribuirlos adecuadamente.

Las recomendaciones y su justificación que sirve como un breve repaso de los conceptos de sistemas operativos y de concurrencia mas una solución posible, son las siguientes:

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 el resultado de un **fork seguido de una exec** de algún tipo que carga el ejecutable del proceso a correr.

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 por 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.

En sistemas Unix que tiene soporte para memoria virtual (casi todos los sistemas oeprativos modernos), la operación fork crea un espacio de direcciones separado para el hijo. El proceso hijo es una copia exacta de todos los segmentos de memoria del proceso padre. En el caso que se implemente la semántica copy-on-write, no es necesario hacer una copia física de la memoria, sino que las páginas virtuales de memoria apuntan a las mismas páginas de memoria física hasta que uno de ellos escriba sobre alguna página, en ese caso se copia. Esta optimización es importante en el caso mas común en cual se usa un fork seguido de una exec para ejecutar un

nuevo programa. En general, el proceso realiza solo unas pocas acciones antes de ceder la ejecución al programa que tiene que ejecutar y esto solo requiere ninguna o muy pocas estructuras del proceso padre.

Resumiendo, 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 comparten el mismo segmento de texto.

Las implementación de copy-on-write (COW) en Linux es la siguiente. Las páginas que se comparten entre el padre y el hijo tienen sus protecciones cambiadas a read-only. Si uno de los procesos trata de cambiar una página, el kernel hace una copia de ella (o sea cambia el puntero de una página en el sistema de memoria virtual). 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":
(Ver resaltado en celeste)

```
/* 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));

/*
 * Si entre el fork y la exec no se usa ninguna variable
 * del proceso padre, Linux usa COW.
/* convertir el parámetro de la exec de integer a string ANTES DEL FORK */
  sprintf(el_parametro, "%d\n",parametro);            /*

 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 o con un semáforo para cada proceso. 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 un set de semaforos que 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 en "C++" (IPC System V)

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

SemaforoUnitario.h

```
#include "inet.h"

class SemaforoUnitario {
public:
    SemaforoUnitario(int);
    bool iniSem(int);
    bool getSem();
    bool creaSem();
    bool p();
    bool v();
    bool eliSem();
```

```
private:
    int identificador;  // nro de IPC del mismo directorio p/clave
    int semid;          // el handler del semaforo
    key_t key;          // la clave para obtener el semaforo
    bool existe;        // si ha sido creado
};
```

SemaforoUnitario.cpp

```
#include "SemaforoUnitario.h"
using namespace std;

SemaforoUnitario::SemaforoUnitario(int identificador) {
    this->identificador = identificador;
    key = ftok(DIRECTORIO,identificador);
}
//crear un semaforo que no existe
bool SemaforoUnitario::creaSem() {
   semid=semget(key,1, IPC_CREAT | IPC_EXCL | 0660);
   if (semid==-1)   return false ;
   existe=true;
   return true;
}

// adquirir derecho de acceso al semaforo existente
bool SemaforoUnitario::getSem(){
   semid=semget(key,1,0660);
   if (semid==-1)   return false ;
   existe=true;
   return true;
}

// inicializar el semáforo
bool SemaforoUnitario::iniSem(int val){
    union semun{
        int val;
        struct semid_ds *buf;
        unsigned short *array;
        struct seminfo *_buf;
    }arg;
   if (!existe) return false;
   arg.val = val;
   return ( semctl(semid, 0, SETVAL, arg)!=-1);
}

//  liberar el semáforo  (v) SIGNAL
bool SemaforoUnitario::v(){
   struct sembuf oper;
   oper.sem_num=0;
   oper.sem_op=1;
   oper.sem_flg=0;
   if (!existe) return false;
   return(semop(semid,&oper,1)==-1);
}
```

```
// ocupar al semáforo (p) WAIT
bool SemaforoUnitario::p(){
    struct sembuf oper;
    oper.sem_num=0;
    oper.sem_op=-1;
    oper.sem_flg=0;
    if (!existe) return false;
    return(semop(semid,&oper,1)==-1);
}
// eliminar el semáforo del sistema
bool SemaforoUnitario::eliSem() {
  return(semctl(semid,0,IPC_RMID,(struct semid_ds*)0)==-1);
}
```

b) para un arreglo de semáforos:

SemaforoMultiple.h

```
#include "inet.h"

class SemaforoMultiple {
public:
    SemaforoMultiple(int);
    bool iniSem(int, int);
    bool getSem(int);
    bool creaSem(int);
    bool p(int);
    bool v(int);
    bool eliSem();

private:
    int identificador;  // nro de IPC del mismo directorio p/clave
    int semcant;        // cantidad de semaforos del arreglo
    int semid;          // el handler del semaforo
    key_t key;          // la clave para obtener el semaforo
    bool existe;        // si ha sido creado
};
```

SemaforoMultiple.cpp

```
#include "SemaforoMultiple.h"
using namespace std;

SemaforoMultiple::SemaforoMultiple (int identificador) {
    this->identificador = identificador;
    key = ftok(DIRECTORIO,identificador);
}
//crear un arreglo de semaforos que no existe
bool SemaforoUnitario::creaSem(int semcant) {
   this->semcant = semcant;
   semid=semget(key, semcant, IPC_CREAT | IPC_EXCL | 0660);
   if (semid==-1)   return false ;
   existe=true;
   return true;
}
```

```
// adquirir derecho de acceso al semaforo existente
bool SemaforoUnitario::getSem(int semcant){
   this->semcant = semcant;
   semid=semget(key, semcant, 0660);
   if (semid==-1)   return false ;
   existe=true;
   return true;
}

// inicializar el semáforo
bool SemaforoUnitario::iniSem(int indice, int val){
    union semun{
        int val;
        struct semid_ds *buf;
        unsigned short *array;
        struct seminfo *_buf;
    }arg;
   if (!existe) return false;
   arg.val = val;
   return ( semctl(semid, indice, SETVAL, arg)!=-1);
}

//  liberar el semáforo  (v) SIGNAL
bool SemaforoUnitario::v(int indice){
   struct sembuf oper;
   oper.sem_num=indice;
   oper.sem_op=1;
   oper.sem_flg=0;
   if (!existe) return false;
   return(semop(semid,&oper,1)==-1);
}

// ocupar al semáforo (p) WAIT
bool SemaforoUnitario::p(int indice){
    struct sembuf oper;
    oper.sem_num=indice;
    oper.sem_op=-1;
    oper.sem_flg=0;
    if (!existe) return false;
    return(semop(semid,&oper,1)==-1);
}
// eliminar el semáforo del sistema
bool SemaforoUnitario::eliSem() {
  return(semctl(semid,0,IPC_RMID,(struct semid_ds*)0)==-1);
}
```

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 un semáforo que no existe e inicializarlo */

    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** usando 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) 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.

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);
        . . .
```

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
 */
        id_a_recibr = 1;
        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);
...
```

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 realizan alguna tarea con ellos. 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.

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.

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.

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.

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)`

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.

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.

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.

Ejercicios para armar la biblioteca de casos.

Para cada ejercicio se pide:

- *Análisis de requerimientos del problema:*
  - *Elabore una lista con los requerimientos NO funcionales y justifique muy brevemente.*
  - *Muestre el diagrama con los casos de uso y límites del sistema.*
  - *Escriba cada caso de uso normal y alternativo.*
  - *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:*
  - *Explique cuales paradigmas de concurrencia están presentes en el problema planteado.*
  - *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:*
  - *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.*

Ejercicio 1)

Una empresa de mantenimiento de dispositivos ofrece un sistema de testeo remoto de dispositivos que detectan que tienen algún tipo de problema de hardware o software. Cuando un dispositivo detecta una falla se comunica con el tester remoto usando uno de los 5 testers que existen en el sistema a su elección. Para brindar un servicio confiable el sistema de testeo no admite mas de 100 dispositivos en testing en el sistema independientemente de cual tester está atendiendolos. Si la capacidad del sistema está cubierta, se avisa al dispositivo y el dispositivo reintenta luego de

una cierta cantidad de tiempo. Si hay capacidad disponible: cuando el tester recibe un requerimiento de testeo, envía un programa de test al dispositivo y espera el resultado. En base al resultado el tester envía un orden de reinicio al dispositivo o le avisa que se pare y emite una orden de reparación que envía a los técnicos.
El sistema es totalmente automático y no hay intervención de personas en la operación del sistema.

Ejercicio 2)

Para evitar el tiempo de espera de la terminación de un test en un dispositivo, se quiere agregar la siguiente mejora: cuando el tester recibe un requerimiento de testeo, envía un programa de test al dispositivo y **mientras espera el resultado** atiende a otro dispositivo. El resultado de un testeo es prioritario con respecto a dispositivo que recién pide un testeo. En base al resultado el tester envía un orden de reinicio al dispositivo o le avisa que se pare y emite una orden de reparación que envía a los técnicos. Se mantienen las mismas restricciones que en el caso anterior.

Ejercicio 3)

Otro sector de la empresa de mantenimiento trabaja con dispositivos mas complejos que requiere de un mas de un programa de testing para determinar el problema en el funcionamiento. Sus limites y cantidad de testers son iguales al del ejercicio 1. Este sistema tiene el siguiente esquema de funcionamiento, cada tester atiende hasta 20 equipos que pidieron testing y en base al resultado del testing realiza un segundo test adicional antes de solicitar reparación. El resto del sistema es idéntico al ejercicio 1 y su esquema de prioridades es que se atiende primero al dispositivo que está mas adelantado en el testeo si varios responden al mismo tiempo.

Ejercicio 4)

Para mejorar el sistema de tests se decidió hacer 4 tests sobre el dispositivo en forma independiente, por consiguiente hay 4 testers especializados esperando por el pedido de test de un dispositivo. En este caso atienden los 4 testers (1 de cada tipo) que estén libres. Cada tester trabaja independientemente como en el caso anterior, o sea el dispositivo va a recibir 4 diagnosticos y si alguno es apagar, apagará el dispositivo y si los 4 diagnósticos son reiniciar, se reiniciará. El resto del sistema se mantiene con las mismas restricciones

Ejercicio 5)

Uno de los técnicos de la empresa de mantenimiento determinó que se puede hacer mas eficiente el sistema e incrementar la cantidad de tests de dispositivos complejos, ya que una vez determinado el tipo de falla, se podría interactuar con el dispositivo testeando diferentes funciones, en vez de hacer los 4 tests obligatoriamente y que a su vez confunden a los técnicos, si reciben mas de un reporte de fallas del mismo dispositivo.
En este caso, se mantiene el esquema inicial del ejercicio 3, pero en vez de empezar el segundo test, se asigna a testers especiales y diferenciados (cada uno solo realiza un determinado tipo de test) (entre 2 y 4) que deben simultáneamente empiezar a

interactuar con el dispositivo solicitándole tareas especifícas y analizando sus resultados.
La relación entre los dispositivos que requieren mas de un testeo y los testers especiales es el siguiente: El dispositivo espera por ordenes de los testers especiales. Recibe una orden, la procesa y devuelve el resultado, luego procesa la siguiente y asi sucesivamente. La cantidad de ordenes que envía cada tester es variable y depende de los resultados que obtiene. Cada tester especial que termina le avisa al tester inicial y cuando todos terminaron este tester le avisa al dispositivo si tiene que reiniciarse o apagarse esperando el técnico y además, envía la orden de reparación a los técnicos. El resto del sistema se mantiene con las mismas restricciones.

Ejercicio 6)

Analizando la eficiencia del sistema, los técnicos de la empresa de mantenimiento decidieron que sería mas eficiente el sistema propuesto en el ejercicio 5, si el tester que envió la última orden le avisa al dispositivo que tiene que reiniciarse o si la falla es grave, apagarse y envía la orden de reparación a los técnicos. Por consiguiente, el tester inicial queda libre al pasarle dispositivo a los testers especiales y puede atender a otro dispositivo, o sea que se podría usar el esquema del ejercicio 1.
El resto del sistema se mantiene con las mismas restricciones.

Ejercicio 7)

Se descubrió que hay veces que el test falla porque el dispositivo tiene alguna condición inicial producida por la falla que determinó su pedido de testing. Por lo cual si alguno de los tests falla si se rehace todo el proceso de testeo de los testers especiales varias veces, el problema se corrige. Por lo cual mientras hay un error fatal en por lo menos un test, todos los tests se repiten hasta 3 veces. El resto del sistema se mantiene con las mismas restricciones y funcionamiento del ejercicio 6.

Ejercicio 8)

Para la eficiencia se decidió agregar un equipo especial que resume el testing y decide si hay que rehacer el testing o no y es el que envía el resultado al dispositivo y emite una sola orden de reparación para los técnicos. El sistema mantiene las mismas restricciones del ejercicio 6.

Ejercicio 9)

Analizando los tests realizados sobre los dispositivos, los técnicos determinaron que hay veces que el error en el funcionamiento se debe a que se alteraron variables de configuración del dispositivo, por lo cual decidieron agregar unos tests adicionales. Estos tests adicionales que corren al mismo tiempo que los tests especiales, y solamente modifican las variables de configuración del dispositivo. De acuerdo al tipo de dispositivo las variables que se pueden modificar son diferentes. Mientras se altera el contenido de una variable de configuración del dispositivo, no se puede procesar una orden, esta actividad es prioritaria. El resto del sistema es idéntico al ejercicio 8.

Fechas de vencimientos de cada parte de la práctica:

Ejercicio	Diagramas	Programación	Grupo
1-			
2			
3			
4			
5			
6			
7			
8			
9			

Los diagramas se escanean y se guardan en un zip con el número de ejercicio y se suben al moodle. Se solicita que agreguen la carátula correspondiente que armaremos en clase para poder clasificarlos y para que otra persona pueda encontrar las partes que se podrían reusar para el TP.