Universidad Nacional de Tierra del Fuego Instituto de Desarrollo Económico e Innovación

Analista Universitario en Sistemas - Licenciatura en Sistemas

# Trabajo Practico N° 2

SCS – Sincronización con Semáforos Ejercicios a entregar: 3, 4, 5c, 8b, 10, 11, 12.

1. Explique las diferencias entre espera ocupada o activa (busy waiting) y bloqueo. ¿Es la espera activa menos eficiente (en términos de uso del procesador) que el bloqueo?

**La espera activa** (o espera ocupada o busy waiting) es un mecanismo de sincronización que se basa en hacer esperar a un proceso (mantenerlo en un bucle) mediante la comprobación continua del valor de una variable compartida, hasta que la condición se cumpla. Esto mantiene ocupada a la CPU.

**El bloqueo** es un mecanismo de sincronización que se basa en poner a un proceso en estado de “espera” hasta que una condición se cumpla o se le notifique que puede continuar. Esto libera a la CPU por un tiempo.

Por lo tanto, es evidente que la espera activa activa es menos eficiente que el mecanismo de bloqueo ya que mantiene a la CPU ocupada, verificando constantemente si el proceso puede continuar, mientras que la espera por bloqueo libera a la CPU permitiendo que otros procesos se continuen ejecutando con mas rapidez (ya que la CPU tiene menos carga de procesamiento).

1. ¿Por qué las operaciones sobre los semáforos deben ser atómicas?

Un semáforo es una estructura de sincronización a nivel de software (controlado por el programador) que consiste en una variable numérica (que indica la cantidad de procesos que pueden acceder a un recurso) y una cola donde se irán ingresando los procesos bloqueados cuando éstos no puedan realizar una operación y así evitar la espera activa. Se usan para evitar condiciones de carrera, sincronizar procesos y proteger secciones críticas.

Cuentan con dos instrucciones elementales:

* Wait() o P(): Sirve para intentar ingresar a una sección crítica. Si el valor del semáforo es mayor a cero, el proceso decrementa el semáforo y continúa; si el valor del semáforo es <= cero, el proceso se pone en la cola de espera (proceso bloqueado) ya que el recursos se encuentra ocupado por otro proceso.
* Signal() o V(): Sirve para salir de la sección crítica y liberar el recursos. En este punto el proceso incrementa el valor del semáforo y permite a un proceso de la cola de espera ingresar a la sección crítica.

Estas instrucciones son atómica ya que es importante que no sean interrumpidas durante su ejecución, evitando que dos procesos modifiquen el semáforo al mismo tiempo, lo cual generaría errores en la sincronización.

1. Suponga que una máquina tiene instrucciones atómicas de incremento y decremento: **INC(var)** y **DEC(var)**. Estas operaciones suman y restan respectivamente 1 a var. Asumiendo que las lecturas y escrituras a memoria son operaciones atómicas:
   1. Es posible simular las operaciones wait y signal de un semáforo general s?. Si lo es, proporcione la simulación y explique el funcionamiento. Caso contrario, explique detalladamente porqué no es posible.

Con solo estas dos instrucciones atómicas no se puede simular un semáforo general s ya que, si bien podría controlar el valor entero del semáforo no tengo forma de controlar la cola de procesos en espera y tampoco bloquear o liberar procesos desde ella sin las operaciones wait y signal.

* 1. Ahora asuma que INC y DEC también devuelven el bit de signo del valor final de var. En particular, si el valor final de var es negativo, las operaciones devuelven 1, en otro caso, 0. Es posible en este caso simular las operaciones wait y signal para un semáforo general s?. Si lo es, escriba la simulación y explique su funcionamiento, sino, explique detalladamente porqué no es posible.

1. Implemente la operación **TestAndSet** en JR, utilizando los semáforos provistos.

Para lograr la sincronización a nivel de hardware se puede implementar el algoritmo TestAndSet, el cual utiliza una variable booleana que permite el acceso de un proceso a la vez a una sección crítica o un recurso. Utiliza la instrucción atómica *< await (!lock) lock = true; >* para la consulta de acceso. Los procesos sin embargo se mantienen en espera activa verificando a cada instante el valor de la variable compartida. Otra desventaja es que no se garantiza la espera limitada, ya que un proceso podría nunca llegar a ingresar a esa sección.

**🡪 Ver implementación en tp**

Definición de Stallings: Su funcionamiento es atómico a nivel de hardware: verifica si una variable es 0, la pone en 1, y devuelve el valor anterior.

1. Utilice semáforos en JR para proporcionar una solución al clásico problema “productores

/ consumidores” con las siguientes variantes:

* 1. búffer simple, **1** productor y **1** consumidor.
  2. búffer ilimitado, **1** productor y **1** consumidor.
  3. búffer ilimitado, **n** productores y **m** consumidores.
  4. búffer acotado, **1** productor y **1** consumidor.
  5. búffer acotado, **n** productores y **m** consumidores.

1. Implemente semáforos binarios en JR valiéndose de los semáforos que éste provee.
2. Un grafo de precedencia, es un grafo acíclico dirigido. Los nodos representan tareas y los arcos indican el orden en el que las tareas se ejecutan. En este caso, una tarea puede ejecutarse tan pronto como todos sus predecesores hayan terminado. Asumiendo que las tareas son procesos con el siguiente código:

1

process T[i]; while(true)begin

WAIT a los predecesores, si los hay; {toma tenedor izquierdo/derecho} cuerpo de la tarea;

SIGNAL a los sucesores si loas hay; end;

end.

2

3

4

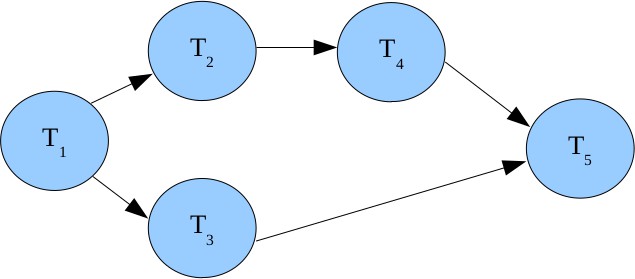
5

6

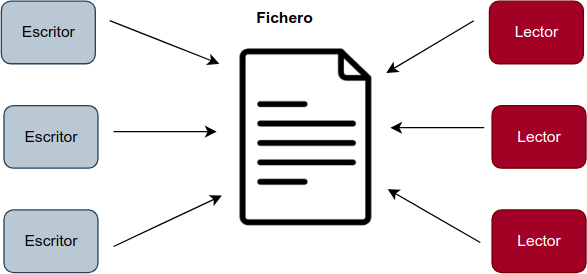
7

8

* 1. Utilizando semáforos, muestre cómo sincronizar cinco procesos en JR cuyo orden de ejecución está especificado por el siguiente grafo de precedencia:



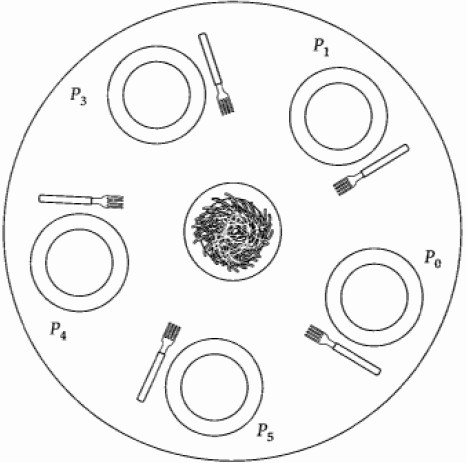
1. Problema de lectores y escritores: Hay un objeto de datos (por ejemplo, un arreglo) que es utilizado por varios procesos, unos leen y otros que escriben. Mientras uno escribe, ningún otro proceso podrá acceder. Sí se permite a varios procesos leer simultáneamente.



Implemente soluciones al problema de “lectores / escritores” utilizando semáforos en JR en las que:

* 1. Los lectores tienen prioridad.
  2. Los escritores tienen prioridad.
  3. Una solución “justa”.

1. En 1965 Dijkstra planteó y resolvió un problema de sincronización llamado el problema de la cena de los filósofos, que se puede enunciar como sigue. Cinco filósofos se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de arroz y un palillo a la izquierda de su plato. Para comer el arroz son necesarios dos palillos y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo coge un palillo y el otro está ocupado, se quedará esperando, con el palillo en la mano, hasta que pueda coger el otro palillo, para luego empezar a comer. Si dos filósofos adyacentes intentan tomar el mismo palillo a la vez, se produce una condición de carrera: ambos compiten por tomar el mismo palillo, y uno de ellos se queda sin comer. Si todos los filósofos cogen el palillo que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguien debe liberar el palillo que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje su palillo). Entonces los filósofos se morirán de hambre. Este bloqueo mutuo se denomina interbloqueo. El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre.



Se propone la siguiente solución al problema de la cena de filósofos. Usted deberá indicar si la misma es correcta o no, en caso negativo proporcione alguna solución, en caso positivo, justifique.

1

program filosofos;

procedure COMER; begin sleep(random(10)); end; procedure PENSAR; begin sleep(random(10)); end;

process type tfilosofo(i:integer); begin

repeat

wait(fork[i]):

{toma tenedor izquierdo}

wait(fork[(i+1) mod 5]): {toma tenedor derecho} COMER;

wait(fork[i]):

wait(fork[(i+1) mod 5]): {libera tenedor izquierdo} PENSAR; {libera tenedor derecho}

forever; end;

var

i:integer;

fork: array[1..5] of semaphore; filosofo: array[1..5] of semaphore;

begin

for i:=1 to 5 do initial(fork[i],1);

cobegin

for i:=1 to 5 do filosofo[i](i);

coend; end.

{programa principal}

{5 semaforos inicializados a 1}

{comienza tareas}

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

En principio el primer error que noto es que luego de los dos wait en vez de hacer signal vuelve a hacer wait. Eso se corrige. El resto si va a funcionar.

1. El problema del barbero durmiente consiste en una barbería en la que trabaja un barbero que tiene un único sillón de barbero y varias sillas para esperar. Cuando no hay clientes, el barbero se sienta en una silla y se duerme. Cuando llega un nuevo cliente, éste o bien despierta al barbero o —si el barbero está afeitando a otro cliente— se sienta en una silla (o se va si todas las sillas están ocupadas por clientes esperando). El problema consiste en realizar la actividad del barbero sin que ocurran condiciones de carrera.



Implemente una solución al mismo utilizando semáforos en JR con las siguientes res- tricciones:

* 1. Puede haber N clientes esperando en la barbería, además de quien está sentado en el sillón de corte.
  2. Hay un único sillón para corte y un único Barbero que además cobra en la única caja disponible.

1. Un Chef y tres ayudantes trabajan en una Pizzería de un modo muy particular. Para hacer una pizza se requiere de tres ingredientes (además de la masa): salsa, queso y morrones. Uno de los ayudantes tiene una cantidad infinita de salsa, otro de queso y el restante de morrones, mientras que el Chef cuenta con una cantidad infinita de los tres ingredientes. Las pizzas se hacen del siguiente modo: el Chef coloca sobre la masa dos de los ingredientes de la pizza, avisa que hay una pizza lista para completar y continúa con la preparación de la siguiente. El ayudante que tiene el ingrediente faltante la completa y avisa que terminó. Recién en este momento, el Chef puede poner nuevamente una pizza disponible a los ayudantes. Ud. debe escribir en JR utilizando semáforos un programa que simule esta situación y resuelva los problemas de sincronización que haya, asegurando que tampoco se produzcan interbloqueos. Esta solución debe arrojar además como resultado la cantidad de pizzas que se han hecho al final de la jornada.
2. Para poder determinar el futuro de un ecosistema, a Ud. se le encarga que construya un simulador sobre parte del mismo, en el que conviven diversos organismos y también gran diversidad de bacterias que los atacan. Particularmente el interés es el estudio de los organismos X que son atacados por las bacterias **Blancas**y **Rojas**. Estas bacterias compiten por atacar a los X y se sabe que aleatoriamente éstos se vuelven vulnerables; si esto ocurre, es a partir de ese momento que las bacterias comienzan su ataque que

siempre culmina exitosamente y en una forma muy precisa; cuando 4 bacterias Blancas y 2 Rojas lograron introducirse en el organismo X, éste muere. Se sabe además que las bacterias blancas se pueden “introducir” en X de a 2 a la vez, mientras que las rojas sólo de a 1.

* 1. Utilice semáforos en JR para construir el simulador que debe ser capaz de correr para 1 organismo X que es atacado por n bacterias Blancas y m bacterias Rojas (con n y m grandes).
  2. Generalice para más de 1 X.