Universidad de Santiago de Chile

Facultad de Ingeniería Informática

Sistemas Operativos

Profesor Fernando Rannou

PEP 2
Hebras y Concurrencia

Nicolás Aguilera

Junio 2023

Índice

1. Hebras

1.1. Definición

Una hebra (hilo) es una secuencia de instrucciones dentro de un proceso que puede ejecutarse de forma concurrente con otras hebras dentro del mismo proceso. **Todas las hebras dentro de un proceso comparten el mismo espacio de memoria y otros recursos del proceso**, como archivos abiertos y señales. Las hebras pueden ejecutarse de manera simultánea en múltiples núcleos de CPU o en un solo núcleo mediante la técnica de multiprogramación.

1.1.1. Diferencia con proceso

Mientras que cada proceso tiene su propio espacio de memoria y recursos asignados, las hebras comparten entre sí el mismo espacio y los recursos del proceso, permitiendo una comunicación más eficiente a la vez que se necesita de sincronización y coordinación para evitar problemas de concurrencia.

1.1.2. Ventajas de las hebras

- Demora menos crear y eliminar una hebra que un proceso.
- Demora menos hacer cambio de contexto entre hebras de un mismo proceso que entre dos procesos.
- Ya que las hebras de un proceso comparten memoria y archivos, ellas se pueden comunicar sin necesidad de invocar rutinas del kernel.
- Permitiría la ejecución paralela de hebras cuando hay varios procesadores.

1.2. Elementos de una hebra

A continuación se mencionan y describen los principales elementos que componen una hebra, los cuales pueden verse en la Figura 1:

- **Identificador de hebra:** Es un valor único que se asigna a cada hebra dentro de un proceso y se utiliza para identificarla de manera única.
- **Contexto de ejecución:** Es el estado actual de la hebra, que incluye los *valores de los registros de la CPU, la pila de ejecución* y cualquier otro contexto necesario para reanudar la ejecución de la hebra en un momento dado.
- **Pila de ejecución:** Es una región de memoria utilizada para almacenar las *variables locales y los datos temporales* durante la ejecución de la hebra. Cada hebra tiene su propia pila de ejecución.
- **Program Counter (PC):** Es un registro que indica la dirección de la próxima instrucción que se ejecutará en la hebra.

- **Recursos compartidos:** Las hebras dentro de un proceso comparten el mismo espacio de memoria y otros recursos del proceso, como archivos abiertos, señales y controladores. Esto permite la comunicación y la sincronización entre las hebras mediante el acceso a estas áreas compartidas de memoria y recursos.
- **Prioridad:** En algunos sistemas operativos, se puede asignar una prioridad a las hebras para determinar su orden de ejecución y asignación de recursos.

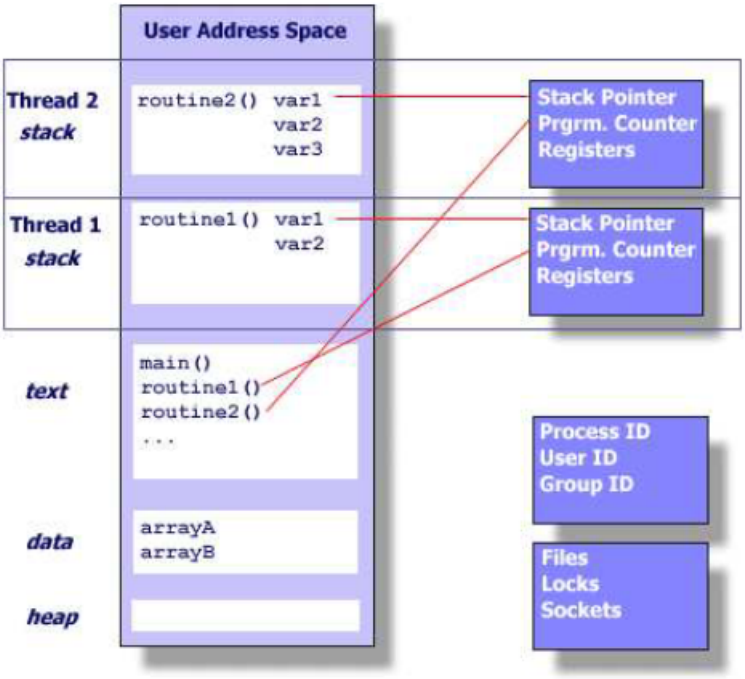

Figura 1: Elementos de una hebra.

1.3. Multihebras en monoprocesador

En un sistema con un solo procesador (mono procesador), las hebras no pueden ejecutarse de forma verdaderamente simultánea como lo harían en un sistema multiprocesador. En cambio, el sistema operativo utiliza un enfoque llamado **multiprogramación o programación cooperativa** para alternar la ejecución de las hebras y simular la concurrencia.

En un mono procesador, las hebras se ejecutan de manera secuencial, pero el sistema operativo divide el tiempo de CPU disponible en pequeños intervalos de tiempo llamados slices o **quantum**. Durante cada slice, se le asigna un intervalo de tiempo a una hebra para ejecutarse. Cuando se agota el tiempo asignado, el sistema operativo interrumpe la ejecución de esa hebra y pasa a otra hebra en espera.

El cambio entre hebras se realiza mediante una técnica llamada **conmutación de contexto**. El contexto de ejecución actual de una hebra se guarda en la memoria y se carga

el contexto de la siguiente hebra que se va a ejecutar. Esto permite que múltiples hebras se turnen para utilizar el tiempo de CPU de manera eficiente.

Aunque las hebras no se ejecutan simultáneamente en un mono procesador, la multiprogramación proporciona la **ilusión de ejecución concurrente**, lo que permite aprovechar los beneficios de la concurrencia, como la capacidad de respuesta y la ejecución paralela de tareas. Un ejemplo de esto puede verse en la Figura 2.

Es importante destacar que, en un mono procesador, si una hebra se bloquea o realiza una operación de espera (por ejemplo, una operación de I/O), el sistema operativo puede ceder el control a otra hebra disponible en lugar de esperar a que se desbloquee. Esto permite que otras hebras continúen su ejecución y aprovechen mejor el tiempo de CPU disponible.

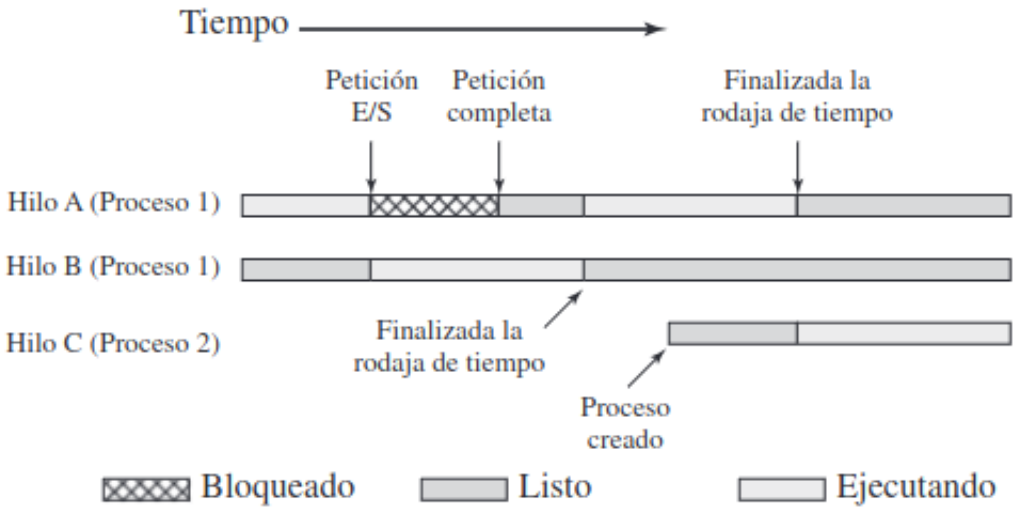

Figura 2: Ejemplo multihilo en monoprocesador.

1.4. Hilos a nivel de usuario

Los hilos a nivel de usuario, también conocidos como *hilos de biblioteca* o *hilos ligeros*, son hilos **gestionados completamente por una biblioteca de programación** en lugar del sistema operativo. Los hilos a nivel de usuario se crean y se planifican en el espacio de usuario sin intervención directa del sistema operativo.

Cuando se utilizan hilos a nivel de usuario, la biblioteca de programación proporciona una implementación propia del modelo de hilo y se encarga de la planificación y ejecución de los hilos. Estos hilos se ejecutan dentro de un solo proceso y comparten el mismo espacio de memoria. Un ejemplo de esto puede verse en la Figura 3.a.

El modelo de hilos a nivel de usuario puede tener las siguientes características:

- **Planificación a nivel de biblioteca:** La biblioteca de programación es responsable de la planificación y el cambio de contexto de los hilos. Puede utilizar algoritmos de planificación personalizados y tomar decisiones basadas en políticas específicas de la aplicación.
- **Bloqueo:** En un sistema operativo típico muchas llamadas al sistema son bloqueantes. Como resultado, cuando un hilo a nivel de usuario realiza una llamada al sistema, no sólo se bloquea ese hilo, sino que se bloquean todos los hilos del proceso. Para solucionar este problema se utiliza una técnica llamada *jacketing* (revestimiento). El objetivo de esta técnica es convertir una llamada al sistema bloqueante en una llamada al sistema no bloqueante.
- **Scheduling a nivel de proceso:** Los hilos a nivel de usuario dependen de la cooperación voluntaria entre sí para ceder el control y permitir que otros hilos se ejecuten. No hay intervención directa del sistema operativo para forzar el cambio de contexto.
- **Menor costo de creación y cambio de contexto:** En comparación con los hilos a nivel de núcleo, los hilos a nivel de usuario son más ligeros en términos de recursos requeridos para su creación y cambio de contexto. Esto se debe a que la biblioteca de programación puede implementar estos mecanismos de manera eficiente sin la necesidad de llamadas al sistema operativo.
- **Sin aplicación multiprocesador:** Dado que la planificación y la gestión de los hilos a nivel de usuario están en manos de la biblioteca de programación, no se pueden aprovechar automáticamente múltiples núcleos o procesadores en un sistema.

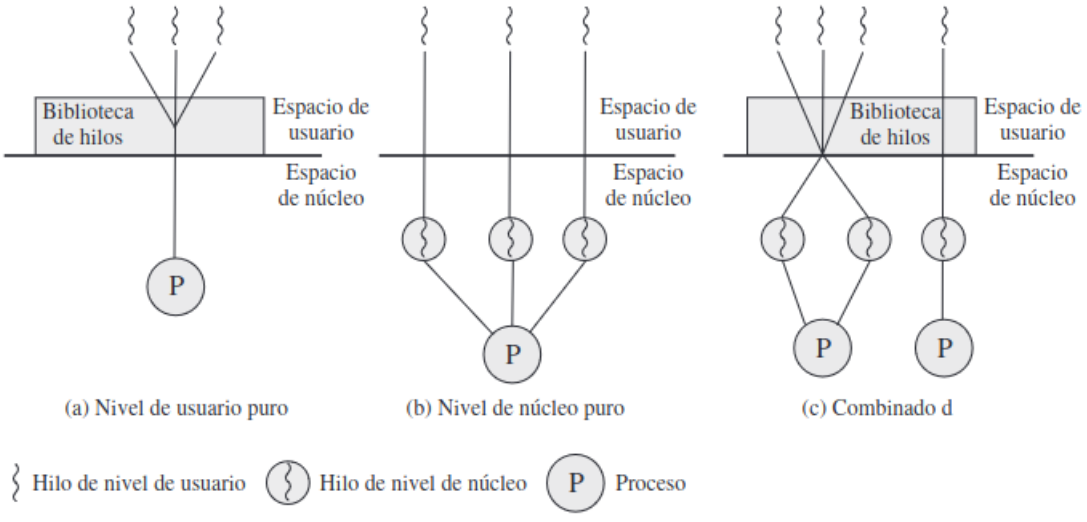

Figura 3: Ejemplos de hilo a nivel usuario y kernel.

1.5. Hilos a nivel de kernel

Los hilos a nivel de kernel, también conocidos como *hilos de núcleo* o *hilos pesados*, son hilos **gestionados directamente por el sistema operativo**, a diferencia de los hilos a nivel de usuario, que son gestionados por una biblioteca de programación.

Aquí hay una descripción básica de cómo funcionan los hilos a nivel de kernel:

- **Cambio de contexto:** Cuando el sistema operativo decide cambiar de un hilo a otro, realiza un cambio de contexto. En este proceso, **el contexto de ejecución actual** del hilo en ejecución, que incluye los registros de CPU y otros detalles, **se guarda en la memoria**, y luego se carga el contexto del siguiente hilo seleccionado para ejecutarse.
- **Sincronización y bloqueo:** Los hilos a nivel de kernel pueden utilizar mecanismos de sincronización proporcionados por el sistema operativo, como semáforos o mutex, para coordinar y controlar el acceso a recursos compartidos. Además, pueden bloquearse cuando esperan eventos o recursos mediante llamadas al sistema específicas.
- **Paralelismo y aprovechamiento de hardware:** Los hilos a nivel de kernel pueden aprovechar directamente los recursos de hardware, como múltiples núcleos de CPU o procesadores, ya que son gestionados por el sistema operativo. Esto permite la ejecución paralela y simultánea de múltiples hilos en diferentes núcleos, lo que mejora el rendimiento y la capacidad de procesamiento.
- **Mayor costo de creación y cambio de contexto:** En comparación con los hilos a nivel de usuario, la creación y el cambio de contexto de los hilos a nivel de kernel generalmente tienen un mayor costo en términos de recursos y tiempo. Esto se debe a la intervención directa del sistema operativo y los mecanismos más complejos involucrados.

Este enfoque soluciona los problemas del ULT, ya que aprovecha los multinúcleos y si se bloquea un hilo de un proceso, el núcleo puede planificar otro hilo del mismo proceso.

1.6. Multiprocesamiento Simétrico (SMP)

Un multiprocesador simétrico, también conocido como sistema multiprocesador simétrico (SMP, por sus siglas en inglés), es una arquitectura de sistema que consta de múltiples **procesadores idénticos y homogéneos que comparten el mismo espacio de memoria** y están conectados a través de un bus de sistema.

En un multiprocesador simétrico, **todos los procesadores tienen el mismo acceso y capacidad para ejecutar tareas y acceder a la memoria compartida**. No hay un procesador principal o maestro (organización maestro/esclavo), sino que todos los procesadores son iguales en términos de capacidad de procesamiento y responsabilidades. Cada

procesador puede ejecutar su propia secuencia de instrucciones y acceder a cualquier ubicación de memoria en el sistema.

Algunas características importantes de los multiprocesadores simétricos son:

- **Paralelismo y ejecución concurrente:** Los procesadores en un multiprocesador simétrico pueden ejecutar tareas simultáneamente y en paralelo, lo que permite una ejecución más rápida y eficiente de los programas. Debido a que múltiples procesadores pueden ejecutar la misma o diferentes partes del código del núcleo, las tablas y la gestión de las estructuras del núcleo deben ser gestionas apropiadamente para impedir interbloqueos u operaciones inválidas (Stallings, 2005).
- **Tolerancia a fallos y escalabilidad:** El sistema operativo no se debe degradar en caso de fallo de un procesador. El planificador y otras partes del sistema operativo deben darse cuenta de la pérdida de un procesador y reestructurar las tablas de gestión apropiadamente.
- **Acceso compartido a memoria:** Todos los procesadores en un multiprocesador simétrico tienen acceso directo a la memoria compartida. Esto permite la comunicación y el intercambio de datos entre los procesadores sin tener que recurrir a mecanismos de comunicación externos, lo que facilita la programación y el desarrollo de algoritmos paralelos.
- **Sincronización y coherencia de memoria:** Debido a que los procesadores comparten la misma memoria, es importante garantizar la coherencia y consistencia de los datos entre los procesadores. Los sistemas SMP incluyen mecanismos de sincronización y protocolos de coherencia de memoria para garantizar que todos los procesadores vean una imagen coherente y actualizada de los datos en la memoria compartida.

2. Concurrencia

2.1. Conceptos clave

A continuación se presentan algunos términos clave asociados a la concurrencia:

- **Condición de carrera (CC):** Situación en la cual múltiples hilos o procesos leen y escriben un dato compartido y el resultado final depende de la coordinación relativa de sus ejecuciones.
- **Sección crítica (SC):** Sucede cuando un trozo de código es ejecutado por múltiples hebras, en el cual se accede a datos compartidos y, al menos una de las hebras escribe sobre los datos.
- **Exclusión mutua (EM):** Requisito de que cuando un proceso esté en una sección crítica que accede a recursos compartidos, ningún otro proceso pueda estar en una sección crítica que acceda a ninguno de esos recursos compartidos.

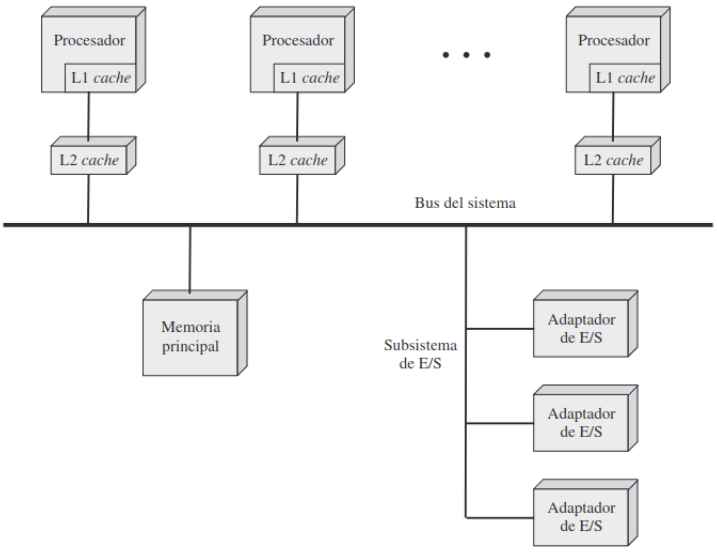

Figura 4: Organización de los multiprocesador simétricos.

- **Deadlock:** Situación en la cual dos o más procesos son incapaces de actuar porque cada uno está esperando que alguno de los otros haga algo.

El siguiente código es un ejemplo de cómo dos hebras podrían insertar un nodo, donde dependiendo el orden de ejecución, puede que solamente uno de los nodos sea insertado:

```
typedef struct list {
    int data;
    struct list *next;
} Lista;

void insert(int item) {
    struct list *p;
    p = malloc(sizeof(struct list));
    p->data = item;
    p->next = Lista;
    Lista = p;  // Esta es una SC
}
```

2.2. Modelo de concurrencia

El código es ejecutado concurrentemente por múltiples tareas $enterCS()$ representa las acciones que la hebra debe realizar para poder entrar a su SC en forma segura. $exitSC()$ representa las acciones que una hebra debe realizar para salir la SC y dejarla habilitada a otras hebras. El *código no crítico* es cualquier código donde no se accede recursos compartidos.

Se asumen una colección de tareas del siguiente tipo:

```
Process(i) {
    while (true) {
        codigo no critico
        ...
        enterCS();
        SC();
        exitSC();
        ...
        codigo no critico
    }
}
```

2.3. Requerimientos para la EM

La exclusión mutua es un concepto que se refiere a la garantía de que solo un hilo o proceso puede acceder a un recurso compartido en un momento dado. Para lograr la exclusión mutua, es necesario cumplir con ciertos requisitos:

- **Exclusión mutua:** Cuando T_i está en su *SC*, ninguna otra hebra T_j puede estar en *SC*.
- **Ausencia de deadlock:** Deadlock es cuando todas las hebras quedan ejecutando *enterCS*() indefinidamente (ya sea bloqueadas o en busy-waiting), y por lo tanto, ninguna entra.
- **Sin inanición:** No debe ser posible que un proceso que solicite acceso a una sección crítica sea postergado indefinidamente: ni interbloqueo ni inanición.
- **Progreso:** Si una hebra T_i ejecuta *enterCS()*, y ninguna otra T_j está en la *SC*, se debe perimitir a T_i entrar a la *SC*.

2.4. Soluciones por hardware

2.4.1. Solución 1: Deshabilitar interrupciones

Recordar que una hebra se ejecuta hasta que invoca un llamado al SO o es interrumpida, por ejemplo por *quantum*. Entonces para garantizar EM, la hebra deshabilita las interrupciones justo antes de entrar en su SC.

```
while (true) {
    deshabilitar_interrupciones();
    SC();
    habilitar_interrupciones();
}
```

Desventajas: La eficiencia de ejecución podría degradarse notablemente porque se limita la capacidad del procesador de entrelazar programas. Un segundo problema es que esta solución no funcionará sobre una arquitectura multiprocesador. Cuando el sistema de cómputo incluye más de un procesador, es posible (y típico) que se estén ejecutando al tiempo más de un proceso.

2.4.2. Solución 2: Uso de testset()

Es una instrucción máquina útil en **escenarios donde múltiples hilos o procesos intentan modificar el mismo valor compartido** y se desea garantizar la EM. Al utilizar *testset()*, se puede verificar y modificar el valor en un solo paso **atómico**, evitando condiciones de carrera y garantizando que solo un hilo o proceso realice la asignación en un momento dado. La instrucción puede definirse de la siguiente manera:

```
boolean testset(int i) {
    if (i == 0) {
        i = 1;
        return true;
    } else return false;
}
```

En la Figura 5.a se explica un protocolo de EM basado en esta instrucción. La instrucción *paralelos()* suspende la ejecución del programa actual e inicia la ejecución concurrente de los procedimientos $P_1, P_2, ..., P_n$, de la siguiente manera: La variable *cerrojo* permite el paso a SC de solo un procedimiento cuando es igual a 0. Todos los demás procesos caen en una **busy waiting** no pudiendo hacer nada hasta obtener el permiso para entrar en su SC. Una vez ha terminado, reestablece el valor de *cerrojo* en 0, y solo uno de los procedimientos que se encontraba en espera es permitido entrar en SC nuevamente.

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
int cerrojo;
void P(int i)
{
    while (true)
    {
        while (!testset (cerrojo))
            /* no hacer nada */;
        /* sección crítica */;
        cerrojo = 0;
        /* resto */
    }
}
void main()
{
    cerrojo = 0;
    paralelos (P(1), P(2), . . . ,P(n));
}
```

(a) Instrucción *test and set*

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
int cerrojo;
void P(int i)
{
    int llavei = 1;
    while (true)
    {
        do exchange (llavei, cerrojo)
        while (llavei != 0);
        /* sección crítica */;
        exchange (llavei, cerrojo);
        /* resto */
    }
}
void main()
{
    cerrojo = 0;
    paralelos (P(1), P(2), . . ., P(n));
}
```

(b) Instrucción *exchange*

Figura 5.2. Soporte hardware para la exclusión mutua.

Figura 5: Soporte de hardware para EM.

2.4.3. Solución 3: Uso de exchange()

La instrucción *exchange()* es otra instrucción de máquina que se utiliza cuando se quiere **intercambiar el valor de una ubicación de memoria con el contenido de un**

registro o una ubicación de memoria diferente. Esta operación se realiza de forma **atómica**, lo que significa que garantiza la consistencia y evita condiciones de carrera cuando se accede a la memoria compartida. Puede definirse de la siguiente manera:

```
void exchange(int registro, int memoria) {
    int temp;
    temp = memoria;
    memoria = registro;
    registro = temp;
}
```

En la Figura 5.b se explica un protocolo de EM basado en esta instrucción. La variable *cerrojo* se inicia en 0. Cada procedimiento tiene una variable local $llave_i$ iniciada en 1. Cuando se encuentra un procedimiento P_i que puede entrar setea *cerrojo* a 1 y entra en SC, excluyendo a los demás procesos. Una vez ha terminado, reestablece el valor de *cerrojo* en 0, y permitiendo a otro proceso entrar en SC nuevamente.

2.4.4. Diferencias entre testset() y exchange()

A continuación se presenta una tabla que muestra las principales diferencias entre las instrucciones *testset()* y *exchange()*:

	testset()	exchange()
Funcionalidad	Verificar si un valor en una ubicación de memoria cumple una condición determinada y, en caso afirmativo, asignar un nuevo valor a esa ubicación de memoria.	Intercambiar el valor de una ubicación de memoria con el contenido de un registro o una ubicación de memoria diferente.
Resultado	Devuelve el valor original de la ubicación de memoria que se verifica y, en caso de que se realice una asignación, devuelve el nuevo valor asignado.	Devuelve el valor original de la ubicación de memoria que se intercambia.

Cuadro 1: Diferencias entre testset y exchange.

2.4.5. Ventajas y desventajas de las instrucciones de máquina

- **Ventajas:**
  - Aplicable a cualquier número de hebras y procesadores que comparten memoria física.
  - Puede ser usada con varias SC, cada una controlada por su propia variable.
- **Desventajas:**

- Busy-waiting.
- Posibilidad de inanición.

2.5. Soluciones por SO

2.5.1. Semáforos contadores

En los sistemas operativos, los semáforos son una herramienta fundamental para la sincronización y la EM entre hilos o procesos. Un semáforo es un tipo de variable especial que se utiliza para controlar el acceso a recursos compartidos y para coordinar la ejecución de hilos o procesos concurrentes.

El funcionamiento básico de los semáforos se basa en dos operaciones principales:

- **wait(s)**, se usa para adquirir el semáforo y acceder a un recurso compartido. Si el valor resultante es negativo, el proceso/hilo se bloquea; sino, continúa su ejecución.
- **signal(s)**, se usa para liberar el semáforo y notificar el recurso compartido como disponible. Si el valor resultante es menor o igual que cero, entonces se despierta un proceso que fue bloqueado por *wait()*. Si hay hilos o procesos bloqueados en la cola de espera del semáforo, se desbloquea uno de ellos y se le permite continuar ejecutándose.

```
void wait(semaphore s) {
    s.count--;
    if (s.count < 0) {
        place process in s.queue;
        block this process;
    }
}
```

```
void signal(semaphore s) {
    s.count++;
    if (s.count <= 0) {
        remove from s.queue;
        place P on ready queue;
    }
}
```

Los bloques de código anterior muestran las primitivas de los **semáforos contadores** para las instrucciones *wait()* y *signal()*.

2.5.2. Semáforos binarios o mutex

Similares a los semáforos contadores, pero solo puede tomar valores binarios. En este caso las operaciones cambian un poco:

- **wait(s)**, comprueba el valor del semáforo. Si el valor es cero, entonces el proceso ejecutado se bloquea. Si el valor es uno, se cambia a cero y sigue su ejecución.
- **signal(s)**, comprueba si hay algún proceso bloqueado en el semáforo. Si lo hay, entonces se desbloquea uno de los procesos bloqueados en *wait()*. Si no hay procesos bloqueados, entonces el valor del semáforo cambia a 1.

A continuación se muestra las primitivas para los **semáforos binarios**:

```
void wait(semaphore s) {
    if (s.value == 1) {
        s.value = 0;
    }
    else {
        place process in s.queue;
        block this process;
    }
}
```

```
void signal(semaphore s) {
    if (s.queue is empty()) {
        s.value = 1;
    }
    else {
        remove from s.queue;
        place P on ready queue;
    }
}
```

2.5.3. EM con semáforos

El siguiente código es una solución al problema de la EM usando semáforos. Se consideran n procesos P_i los que necesitan acceder al mismo recurso. El semáforo se inicializa en 1, de tal manera que el primer proceso que ejecute *wait(s)* entrará inmediatamente a su **SC**, poniendo el valor del semáforo en 0. De aquí en adelante cualquier proceso P_i que quiera entrar a SC usando *wait(s)* será bloqueado. Cuando el proceso que está en SC termine, hará *signal(s)*, aumentando el valor de s, sacando un proceso de la cola de bloqueados y colocandolo en la cola de listos.

```
semaphore s;
Process(i) {
    while (true) {
        codigo no critico
        ...
        wait(s);
        SC();
        signal(s);
        ...
        codigo no critico
    }
}

void main() {
    s = 1;
    paralelos(P(1), P(2),...,P(n));
}
```

En la Figura 6, se muestra el ejemplo de un semáforo para tres procesos usando como base el código anterior.

El valor del semáforo es inicializado en 1, de esta manera el proceso A que es el primero en hacer *wait(s)* es capaz de entrar en su sección crítica. Luego B y C, también lo intentan, pero como el valor del semáforo es negativo (-1 y -2), ambos procesos son bloqueados. Una vez que A sale de su SC, recién ahí B es desbloqueado y comienza su SC. Lo mismo para C cuando B termina.

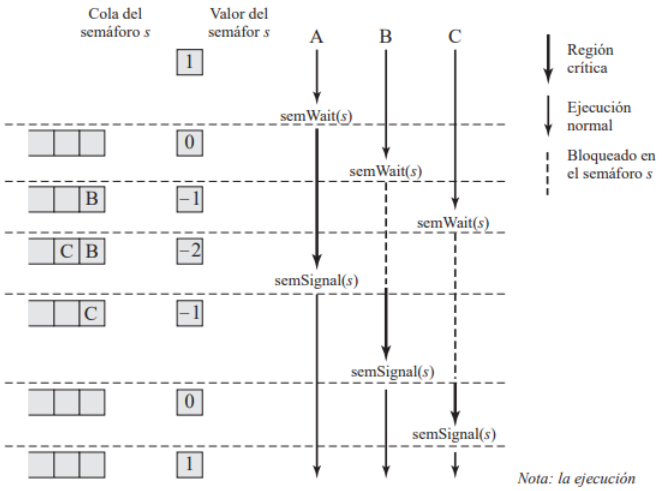

Figura 6: Tres procesos accediendo a un recurso compartido protegidos por un semáforo.

2.5.4. Problema del productor/consumidor

Uno o más tareas **productoras** generan datos y los almacenan en un buffer b compartido. Una tarea **consumidora** toma (consume) los datos del buffer uno a la vez. Claramente, sólo un agente (productor o consumidor) puede acceder el buffer a la vez. El comportamiento del productor y el consumidor pueden verse de la siguiente manera:

```
producer() {
    v = produce();
    b[in] = v;
    in++;
}
```

```
consumer() {
    while (in == out); // do nothing
    w = b[out];
    out++;
    consume(w);
}
```

2.5.5. Buffer finito - semáforos contadores

El buffer es tratado como un buffer circular, El comportamiento está dado por el Cuadro 2 y en el siguiente código.

```
semaphore s = 1;
semaphore full = 0;
semaphore empty = sizeofbuffer;

producer() {
    while(true) {
        v = produce();
        wait(empty);
        wait(s);
        put_in_buffer(v);
        signal(s);
        signal(full);
    }
}
```

```
consumer() {
    while(true) {
        wait(full);
        wait(s);
        w = take_from_buffer();
        signal(s);
        signal(empty);
        consume(w);
    }
}
```

	Bloqueo	Desbloqueo
Productor	Insertar con el buffer lleno	Al insertar dato
Consumidor	Extraer con el buffer vacio	Al consumir dato

Cuadro 2: Comportamiento buffer finito.

En este caso, el semáforo **empty** es usado para contar la cantidad de espacios vacíos. El semáforo **full** hace referencia a la cantidad de espacios usados y **s** es usado para respetar la EM.

2.5.6. Buffer infinito - semáforos contadores

El comportamiento del productor y del consumidor en un buffer infinito está dado en el siguiente código:

```
semaphore s = 1;
semaphore n = 0;

producer() {
    while(true) {
        v = produce();
        wait(s);
        put_in_buffer(v);
        signal(s);
        signal(n);
    }
}
```

```
consumer() {
    while(true) {
        wait(n);
        wait(s);
        w = take_from_buffer();
        signal(s);
        consume();
    }
}
```

2.5.7. Monitores

```c
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t{
    int a;
    int b;
} myarg_t;

typedef struct __myret_t{
    int x;
    int y;
}myret_t;

void *mythread(void *arg){
    // Se crea la variable m que ser  de tipo myarg_t
    myarg_t *m = (myarg_t*) arg;
    //Se imprime sus valores
    printf("%d %d\n", m->a, m->b);
    // Se crea la variable r que ser  de tipo myret_t
    myret_t r;
    // Se asignan sus valores
    r.x = 1;
    r.y = 2;
    //Retorna el variable
    return (void*) &r;
}

int main(int argc, char *argv[]){
    // Se crea una variable de tipo entero llamado rc
    int rc;
    // Se crea una hebra p
    pthread_t p;
    // Se crea una variable de tipo myret_t llamada m
    myret_t *m;
    // Se crea una variable de tipo myarg_t llamada args
    myarg_t args;
    // Se le asignan los valores a esta  ltima  estructura.
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void**)&m);
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

```
int i, sum;
enum {THREAD_FAIL, THREAD_SUCCESS};

void *runner(void *param){
  int limit = atoi(param);
  int i, check;
  sum=0; check=0;
  if (limit > 0){
    for (i = 1; i <= limit; i++){
    sum += i;
    check += i;
    }
  }
  if (check == sum) pthread_exit((void *) THREAD_SUCCESS);
  else pthread_exit((void *) THREAD_FAIL);
}

char* statusToString (int status){
  return status == THREAD_SUCCESS ? "success!":"fail";
}

main(int argc, char *argv[]) {
  pthread_t threads[MAX_THREADS];
  int status;
  for (i=0;i<MAX_THREADS;i++)
    pthread_create(&threads[i], NULL, (void*)runner, (void *) argv[1]);
  for (i=0;i<MAX_THREADS;i++){
    pthread_join(threads[i], (void **)&status);
    printf("sum = %d, status = %s\n", sum, statusToString(status));
  }
}
```

Considere MAX_THREAD = 5 y que el usuario ingresa un número positivo mayor a cero por consola. Para probar el código, se corrió el programa con distintos valores de entrada obteniendo los siguientes resultados:

- entrada: 1, cantidad de success: 5
- entrada: 10, cantidad de success: 5
- entrada 100, cantidad de success: 5
- entrada 1.000, cantidad de success: 3
- entrada 10.000, cantidad de success: 5
- entrada 100.000, cantidad de success: 1
- entrada 1.000.000, cantidad de success: 0

Explique por qué se da este comportamiento.

Suponga que existen las funciones enterSC(sc) y exitSC(sc) que definen la sección de código donde se proveerá EM. Se define la siguiente estructura de datos conocida como contador. Suponga que varias hebras acceden al mismo contador. Modifique el código y cree una estructura thread-safe, para ello:

- Identifique la o las SCs
- Provee EM a través de las funciones enterSC(sc) y exitSC(sc)

```
typedef struct __counter_t{
    int value;
} counter_t;

void init_counter(counter_t *c){
    enterSC();
    c->value = 0;
    exitSC();
}

void increment_counter(counter_t *c){
    enterSC();
    c->value++;
    exitSC();
}

void decrement_counter(counter_t *c){
    enterSC();
    c->value--;
    exitSC();
}

int get_counter(counter_t *c){
    int aux;
    enterSC();
    aux = c->value;
    exitSC();
    return aux;
}
```

3. Bibliografía y referencias

1. Stallings W. (2005). *Sistemas Operativos. Aspectos internos y aspectos de diseño.* (5ta ed.). Pearson.

2. Modelo GPT-3.5, OpenAI (2021). ChatGPT. [Software informático]. Disponible en https://openai.com