Pep 2 - Hebras, Concurrencia y Deadlock

Hebras

Elementos de una hebra:

- Estado de ejecución
- Un contexto de hebra: Se almacena cuando no está en ejecución.
- Una pila/stack de ejecución
- Identificador de hebra
- Acceso a datos globales y recursos del proceso al que pertenece
- En un entorno multihebra hay un un PCB y un EVD asociado al proceso pero ahora hay varios Stacks separados para cada hebra así como un TCB (thread control block) que contiene valores referentes al estado de la hebra (como la prioridad)

Beneficios de las hebras:

- Demora menos en crear y eliminar que un proceso
- Demora menos hacer un 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 al kernel
- Permite ejecución paralela en multiprocesadores

Hebras a nivel de usuario y de kernel

ULT

Ventajas ULT sobre KLT

- Cambio de hebra no requiere privilegios de kernel: Gestión de hilos es a través de librerías de manejo de hebras.
- La planificación puede especificarse por parte de la aplicación.
- Pueden ser implementadas en cualquier SO.

Desventajas de ULT

- Cuando un ULT realiza una llamada al sistema, no solo se bloquea ese hilo, sino que se bloquean

todos los hilos del proceso.
- En una estrategia ULT pura, una aplicación multihilo no puede sacar ventaja del multiproceso

KLT

- El núcleo puede planificar simultáneamente múltiples hebras de un solo proceso en múltiples procesadores
- Si se bloquea una hebra de proceso, el núcleo puede planificar otra hebra del mismo proceso
- **La principal desventaja de un KLT es que la transferencia de control entre hebras de un mismo proceso requiere cambio de modo**
- Si no usa una api/biblioteca estandar los programas son menos portables entre SO's

Symmetric Multiprocessing

- Conjunto de procesadores comparten un mismo bus para acceder a memoria, I/O devices, etc pero cada uno tiene su cache
- SO puede correr en cualquiera de los procesadores
- SO puede asignar hebras o procesos a cualquiera de los procesadores
- SO puede correr paralelamente en los procesadores
- SMP + Multihebras -> PARALELISMO

Concurrencia y sincronización

- **La concurrencia abarca aspectos como:**
  - Comunicación entre procesos
  - Compartición/Competencia por recursos
  - Sincronización de actividades de múltiples procesos y la reserva de tiempo de procesador para los procesos.
- Multiprogramación: Gestión de múltiples procesos dentro de un sistema monoprocesador
- Multiprocesamiento: Gestión de múltiples procesos dentro de un multiprocesador
- Hay problemas de sincronización tanto en sistemas multiprocesadores como en monoprocesadores

Palabras clave:

- **Sección crítica:** Sección dentro de un proceso que requiere acceso a recursos compartidos y que no puede ser ejecutada mientras otro proceso este en una sección de código correspondiente
- **Deadlock:** Situación en la cual dos o más proceso son incapaces de actuar porque cada uno esta esperando que alguno de los otros haga algo.

- **Círculo Vicioso(livelock):** Situación en la cual dos o más procesos cambian continuamente su estado en respuesta a cambios en los otros procesos, sin realizar ningún trabajo útil.
- **Exclusión Mutua:** Requisito de que cuando un proceso esté en una SC que accede a recursos compartidos , ningún otro proceso pueda estar en una sección crítica que acceda a ninguno de esos recursos compartidos.
- **Condición de carrera:** Situación en la cual múltiples hebras leen y escriben un dato compartido y el resultado final depende de la coordinación relativa de sus ejecuciones.
- **Inanición:** Situación en la cual un proceso preparado para avanzar es evitado indefinidamente por el planificador; aunque es capaz de avanzar, nunca se le escoge.
- Problemas de sincronización pueden ocurrir tanto en multi y mono procesador

```
typedef struct list {

        int data;
        struct list *next;
} Lista;

//Toda la funcion insert es seccion critica debido a que es pequeña por lo que puedo
externalizar mis mecanismos de sincronización

void insert(int item) {

        struct list *p;
        p = malloc(sizeof(struct list)); //El system call debe ser thread-safe de lo
contrario seria SC
        p->data = item;
        p->next = Lista; //read
        Lista = p; //write
        // Si no existiera ese write, no hay SC

}
```

```
Función es correcta si no tuviera multithread
```

Condición de carrera:

- Competencia entre hebras por escribir un resultado a tal punto de que el resultado final de las acciones que realizan depende del orden en que se ejecuten las instrucciones de cada hebra.
- Una CC siempre representa un error potencial de concurrencia
- Revisar apendice A del libro

"En mi código hay tal condición de carrera"

Sincronización:

Método o protocolo para que se acceda de forma segura a los datos

Sección crítica:

- Trozo de código ejecutado por múltiples hebras en el cual está la condición de carrera (se accede a datos compartidos y al menos una de las hebras escribe sobre los datos)
- Suele ser un código no atómico compuesto de muchas interrupciones

No hay SC si:

- Ninguna hebra modifica los datos (hacer write)
- Si no hay recursos compartidos
- Si el proceso es monohebra

Cual es la SC en la función insert?

Exclusión Mutua

- Requerimiento sobre una SC que dice que sólo una hebra puede estar ejecutando dicha SC.

La aplicación de EM crea dos problemas:

- Deadlock: Por ejemplo, considere dos procesos, P1 y P2, y dos recursos, R1 y R2. Suponga que cada proceso necesita acceder a ambos recursos para realizar parte de su función. Entonces es posible encontrarse la siguiente situación: el sistema operativo asigna R1 a P2, y R2 a P1. Cada proceso está esperando por uno de los dos recursos. Ninguno liberará el recurso que ya posee hasta haber conse□guido el otro recurso y realizado la función que requiere ambos recursos. Los dos procesos están in□terbloqueados
- Inanición: Suponga que tres procesos (P1, P2, P3) requieren todos accesos periódicos al recurso R. Considere la situación en la cual P1 está en posesión del recur□so y P2 y P3 están ambos retenidos, esperando por ese recurso. Cuando P1 termine su sección crítica, debería permitírsele acceso a R a P2 o P3. Asúmase que el sistema operativo le concede acceso a P3 y que P1 solicita acceso otra vez antes de completar su sección crítica. Si el sistema operativo le conce□de acceso a P1 después de que P3 haya terminado, y posteriormente concede alternativamente acceso a P1 y a P3, entonces a P2 puede denegársele indefinidamente el acceso al recurso, aunque no suceda un interbloqueo.

```
// SC
void pop() {
        if (! empty()}
                top--;
}
```

```
// SC
void push(int item) {
        if (! full()) {
                top++;
                data[top] = item
        } else error()
}
```

La EM dice si hay una hebra que esta en pop no puede haber otra en push y viceversa

Requerimientos para la EM

1. Exclusión Mutua: Debe hacerse cumplir: sólo se permite un proceso al tiempo dentro de su sección crítica, de entre todos los procesos que tienen secciones criticas para el mismo recurso u objeto compartido
2. Sin deadlock: todas las hebras quedan ejecutando enterSC() indefinidamente
3. Sin inanición: Una hebra entra en inanición si nunca logra entrar a su SC
4. Progreso: Si una hebra ejecuta enterSC() y ninguna otra está en la SC entonces se le debe permitir entrar a la SC
5. Notas:
    1. Una hebra puede tener mas de una SC
    2. Deadlock $\rightarrow$ Inanición pero no así al revés
    3. No se debe realizar suposiciones sobre la velocidad relativa y orden de ejecución de las hebras

Modelo de Concurrencia

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

```
                codigo no critico
        }
}
```

cuando la puerta esta cerrada la hebra invasora se queda

Locks

Atomicidad

- Una operación es atómica si es que no puede dividirse en partes. Se ejecuta completamente o no se ejecuta y no puede ser interrumpida
- Si es atómico -> Es exclusivo
- Pero si es Exclusivo no necesariamente es atómico ¿Por qué?
  - Se refiere a código, operaciones, cualquier cosa ejecutable
- **Ilusión de atomicidad:** Uno a veces cree que las cosas son atómicas

```
inc(int &i) {      LOAD i, ACC // carga el acumulador
        i = i + 1;     INC ACC // incrementa el acumulador
}                  STORE ACC, i // almacena el resultado en memoria
```

En el código cada instrucción assembler es atómica pero una de alto nivel *no lo es*

i es un dato compartido y acc es un dato local de la hebra

Una asignación simple del tipo `i = 0` es atómica

Solucion por hardware 1:

Deshabilitar interrupciones:

- La sección crítica se ejecuta como si fuera atómica, tiene todo el tiempo del mundo para ingresar a la sección critica, ejecutar todas sus tareas y luego salir de la sección critica y las demás hebras se verán obligadas a esperar
- Puede producir inanición
- No sirve en multiprocesadores
- Si hay un IO en la SC no se le podrá avisar al programa que e IO ha terminado

Solución por hardware 2:

testset()

- Instrucciones que se realizan/ejecutan de forma atómica
- Se inicializa el cerrojo (bolt) en cero
- La primera hebra en hacer testset setea bolt=1 y bloquea el acceso a las demás hebras
- cuando la hebra que entró a la sc sale de esta, setea bolt=0 permitiendo que las demás hagan testset y logren entrar
- Mientras bolt=1 quedan en busy-waiting
- No produce deadlock pero sí puede producir inanición

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

T(i) {
        while (true) {
                ...
                while (!testset(bolt)); //busy-waiting or spin-lock, hebra queda ocupada
esperando
                SC();
                bolt = 0;
                ...
        }
}

int bolt; // shared
void main() {
        bolt = 0;
        parbegin(T(1), T(2),...,T(n));
}
```

Solución por hardware 3:

exchange()

- Mientras la llave no sea la que me sirva entonces la cambio con bolt

```
void exchange(int a, int b) {
        int tmp;
        tmp = a;
        a = b;
```

```
        b = tmp;
}

T(i) {
        int keyi;
        while (true) {
                keyi = 1; //ENTERSC de aqui
                while (keyi != 0)
                        exhange(keyi, bolt); //busy-waiting ;;a aqui
                SC();
                exchange(keyi, bolt);
                ...
        }
}

int bolt; // shared
void main() {
        bolt = 0;
        parbegin(T(1), T(2),...,T(n));
}
```

- EnterSC de esta solución lamentablemente no es atómico
- ¿Esta solucion satisface el progreso? Cuando una hebra comienza su enter "yo ahora estoy compitiendo por entrar a la SC" si la SC esta vacia ella deberia ser capaz de entrar (bolt = 0) entra a enterSC, toma key=1 y repentinamente hay un CC y ahora entra otra hebra y tambien toma el key=1
- Ahora hay dos hebras con key=1. Sí satisface el progreso pero puede producir inanición.

En general las soluciones por hardware generan un uso del procesador innecesario (spinlock) además de la posibilidad de producir inanición

Solución por software 1:

```
int turno;
T0 {
        while (true) {
                while (turno != 0);
                SC();
                turno = 1;
        }
}
```

```
T1 {
        while (true) {
                while (turno != 1);
                SC();
                turno = 0;
        }
}

void main() {
        turno = 0;
        parbegin(T0, T1);
}
```

- Garantiza exclusión mutua. No es posible que dos hebras estén en la sección crítica ya que el turno es cero o es uno.
- Si quiero demostrar que algo no funciona doy un ejemplo, si quiero demostrar que algo funciona debe ser una demostración formal.
- No hay concurrencia, es una ejecución secuencial

Solución por software 2:

```
boolean flag[2];
T0 {
        while (TRUE) {
                while (flag[1]);
                flag[0] = true;
                SC();
                flag[0] = FALSE;
        }
}

T1 {
        while (TRUE) {
                while (flag[0]);
                flag[1] = true;
                SC();
                flag[1] = FALSE;
        }
}

void main() {
```

```
        flag[0] = flag[1] = FALSE;
        parbegin(T0, T1);
}
```

- Satisface progreso pero no el req de EM
- Suben su bandera demasiado tarde

Solución por software 3:

Solución de Peterson para dos Hebras:

- Hasta ahora todas son busy-waiting
- Satisface todos los requerimientos
- Yo quiero entrar (subo mi bandera) pero le doy el paso a la otra, mientras la otra quiere entra y sea el turno de la otra, la otra puede entrar de lo contrario si no se cumplen esas dos condiciones yo puedo entrar.
- **¿Cómo demostrar que satisface EM?**
  - Demostración por contradicción:
    - Hipótesis: No satisface la EM
      - Entonces la h0 está en SC y la h1 está en SC
      - Si ambas están en SC entonces el while anterior fue false para ambas hebras.
      - t0: `( flag[1]==T && turno==1 )` es false
      - ``
      - t1: `( flag[0]==T && turno==0 )` es false

flag[0] y flag[1] deben ser verdaderas y por ende turno=0 y turno=1 lo cual es una contradicción entonces decir "No satisface a la EM" es falso por tanto sí satisface la EM

“photo_5037722340577881390_y.jpg” could not be found.

panaderia saltado

Solución de S0:

Semáforos y Locks:

El principio fundamental es éste: dos o más procesos pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una

señal específica

1. Un semáforo puede ser inicializado a un valor no negativo.
2. La operación semWait decrementa el valor del semáforo. **Si el valor pasa a ser negativo, entonces el proceso que está ejecutando semWait se bloquea.** En otro caso, el proceso continúa su ejecución.
3. La operación semSignal incrementa el valor del semáforo. **Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados** en la operación semWait.

wait(s):

- decrementa el semáforo, si el valor resultante es negativo, el proceso se bloquea; sino, continúa suejecución.

signal(s):

- incrementa el semáforo, si el valor resultante es menor o igual que cero, entonces se despierta un proceso que fue bloqueado por wait, puede despertar una hebra

el SO garantiza que wait y signal se ejecuten de manera atómica

Semáforo binario o mutex

Problemas

El problema del productor/consumidor

El problema es la sincronización

Buffer circular/finito:

in = (in + 1)%n

in: posición donde ingresar el próximo item
out: Posicion del proximo elemento a consumir

buffer infinito solución correcta foto 28/05 13:57 (ejemplo en libro igualmente)

Semaforo contador

buffer circular: full y empty son semaforos de comunicación y s es un semáforo de EM foto sacada luego de la anterior

Problema de los filósofos comensales

Produce deadlock pero satisface pero satisface el requerimiento de coordinación correcta(?)

Problema de los lectores/escritores

Pueden acceder a un recurso todos los lectores que quieran pero no pueden entrar multiples escritores ya que puede desencadenar ¿una CC? y tampoco si hay un escritor con el recurso puede acceder a este un lector y viceversa.

En la primera solución lectores tienen prioridad y puede generar inanición de lectores

El problema lectores/escritores se define como sigue: Hay un área de datos compartida entre un número de procesos. El área de datos puede ser un fichero, un bloque de memoria principal o incluso un banco de registros del procesador. Hay un número de procesos que sólo leen del área de datos (lectores) y otro número que sólo escriben en el área de datos (escritores). Las siguientes condiciones deben satisfacerse.

1. Cualquier número de lectores pueden leer del fichero simultáneamente.
2. Sólo un escritor al tiempo puede escribir en el fichero.
3. Si un escritor está escribiendo en el fichero ningún lector puede leerlo.

Solución de libro

- Escritores con prioridad

Problema típico con semáforos

- No implementan la solución al problema de la EM
- Esto lo vienen a solucionar los monitores
- HW,SW,SO,LENG

Monitores

4 condiciones: EM, Deadlock, Inanicion, Progreso

- Garantiza EM
- Monitor es un módulo de software, es una propiedad/ objeto del lenguaje de programación. Va a encapsular mi aplicación
-

Qué pasa si pongo un semáforo dentro de un monitor???????

Sincronización con hebras POSIX

Mutex

- Provee el mismo servicio que un semáforo binario

trylock: intentar bloquear el mutex

```
pthread_mutex_t m;
pthread_mutex_init(&m, NULL); //importante inicializarlo donde en null van los atributos por defecto
pthread_mutex_lock(m); //entersc()
SC(); //todos los que hagan lock quedan bloqueados en FIFO
pthread_mutex_unlock(m); //exitsc()
```

¿ Qué sucedería si una hebra se bloquea dentro de una SC?
Una VC permite que una hebra se bloquee dentro de una seccion critica y al mismo tiempo libere la SC

Signal despierta a una hebra que este durmiendo pero no abre el mutex, el wait abre el mutex y bloquea(duerne) la hebra que esta haciendo el wait

broadcast de una variable de condicion que es un signal a todas las hebras

Deadlock e inanición

Recursos Reutilizables/Reusables

- Es aquél que no se destruye cuando se usa, como un canal de I/O o una región de memoria

Recursos Consumibles

- Un recurso consumible es aquél que se destruye cuando lo adquiere un proceso; algunos ejemplos son los mensajes y la información almacenada en buffers de I/O

Condiciones necesarias para deadlock

- **Exclusión Mutua:** Solo un proceso puede utilizar un recurso en cada momento. Ningún proceso puede acceder a una unidad de un recurso que se ha asignado a otro proceso
- **Hold and Wait:** Un proceso puede mantener los recursos asignados mientras espera la asignación de otros recursos
- **Sin expropiación:** No se puede forzar la expropiación de un recurso a un proceso que lo posee

Estas condiciones **pueden** llevar a deadlock pero no son suficientes:

- **Espera Circular:** Cadena cerrada de procesos tal que cada proceso mantiene al menos un recurso que es solicitado por otro proceso en la cadena
  La cuarta condición es, realmente, una consecuencia potencial de las tres primeras.

Estrategias para el tratamiento de deadlock

a. Prevención

1. Método indirecto

- Impide la aparición de una de las 3 condiciones necesarias listadas previamente
  - EM: EN general no puede eliminarse. Si el acceso a un recurso requiere EM el SO debe proporcionarlo.
  - Hold and wait: Puede eliminarse estableciendo que un proceso debe solicitar al mismo tiempo todos sus recursos requeridos, bloqueandolo hasta que se le pueda conceder todos los que necesite.
  - Sin desapropiación: Si un priceso mantiene varios recursos y se le deniega una posterior solicitud, ese proceso deberá liiberar los que ya tiene.

2. Método Directo

- Impide que se produzca una espera circular
  - Espera circular:
- Uso ineficiente de recursos

b. Predicción Evitar (Avoidance)

Permite las 3 condiciones necesarias pero toma decisiones razonables para asegurar de que nunca se alcanza el punto de deadlock. Se decide dinámicamente si la petición actual de reserva de un recurso podría potencialmente causar un deadlock **Requiere conocimiento de las futuras solicitudes de recursos del proceso**

1. No iniciar proceso

No iniciar un proceso si sus demandas podrían llevar a DL

2. No conceder petición de recurso (BANQUERO)

No conceder una petición de un recurso por parte de un proceso si esto implica un DL

- Estado seguro: Hay al menos una **secuencia de asignación** de recursos a los procesos que no implica un DL
- Estado inseguro: estado seguron't

c. Detección

En contraste a las demás no limita el acceso a los recursos ni restringe las acciones de los procesos. Con la detección de DL los recursos pedido se conceden. Periódicamente el SO realiza un algoritmo que le permite detectar la condición de espera circulas descrita anteriormente.

Problema de los filósofos comensales

Algoritmo del banquero

Repasar en tablet

CUESTIONES DE REPASO

6.1. Cite ejemplos de recursos reutilizables y consumibles.
6.2. ¿Cuáles son las tres condiciones que deben cumplirse para que sea posible un interbloqueo?
6.3. ¿Cuáles son las cuatro condiciones que producen un interbloqueo?
6.4. ¿Cómo se puede prever la condición de retención y espera?
6.5. Enumere dos maneras cómo se puede prever la condición de sin expropiación.
6.6. ¿Cómo se puede prever la condición de espera circular?
6.7. ¿Cuál es la diferencia entre predicción, detección y prevención del interbloqueo?