RESUMEN CONCURRENCIA

1. ¿Para qué se usan los hilos? Mencione ejemplos
   Los hilos son una abstracción de un solo proceso en ejecución. Un programa multihilo tiene más de un punto de ejecución a diferencia de un proceso que sólo tiene un punto de ejecución. Cada hilo es como un proceso separado, excepto porque los hilos comparten el mismo espacio de direcciones y acceden a los mismos datos. Los hilos se usan para dividir una tarea determinada como un proceso. Ejemplos: aplicaciones modernas basadas en servidor (servidores web, sistemas de administración de bases de datos), hardware moderno.
2. ¿Cuáles son los beneficios de crear hilos?
   Al crear hilos se recibe el beneficio del paralelismo, esto es que cada hilo ejecuta una porción de un trabajo completo por medio de un procesador por hilo. También se usan hilos para evitar que el progreso de un programa se bloquee debido a tareas de I/O, la CPU puede ser usada por otros hilos que están listos para correr para hacer algo útil. Así como también usar hilos permite la multiprogramación.
3. ¿Cuál es la diferencia entre crear un hilo y un proceso?
   Un proceso tiene un solo punto de ejecución, mientras que un programa multihilo tiene muchos puntos de ejecución. Los procesos no comparten el espacio de direccionamiento entre ellos, los hilos sí comparten el espacio de direcciones, ya que acceden a los mismos datos. Con los procesos se salva su estado el PCB (Process Control Block), en los hilos el estado se almacena en TCB (Thread Control Blocks), almacena el estado de cada hilo de un proceso. El espacio de direcciones también cambia en los procesos y en los hilos.
4. ¿Cuáles son los nuevos retos que debe tener en cuenta un programador en un sistema multiprocesador? Explique cada uno
   Los retos a tener en cuenta en un sistema multiprocesador son:
   - División de actividades
   - Balance de carga
   - Segmentación de datos
   - Dependencia de datos
   - Testeo y depuración
5. ¿Cuál es la diferencia entre paralelismo y concurrencia?
   El paralelismo implica que un sistema puede hacer a la vez más de una tarea, es propio de sistemas multicore o multiprocesador.
   La concurrencia permite la ejecución de más de una tarea conforme transcurre el tiempo.
6. ¿Qué tipos de paralelismo existen? Explicar cada uno
   - Paralelismo de datos: Divide los datos en subconjuntos, los cuales son procesados por diferentes cores realizando el mismo conjunto de operaciones.
   - Paralelismo de tareas: Distribuye hilos a través de los núcleos de tal manera que cada hilo realiza una única operación.
7. En clase se discutió acerca del navegador Google Chrome y su técnica de abrir cada nuevo sitio web usando un proceso diferente, ¿se habrían logrado los mismos beneficios si Chrome hubiera sido diseñado para abrir cada nuevo sitio web en un hilo separado? Explicar la respuesta
   Sí, puesto que como cada nuevo sitio web se abre en un nuevo proceso diferente y los hilos se comportan como procesos excepto porque los hilos tienen un espacio de direcciones que comparten y además comparten también los datos y cuando hay un cambio de contexto, el espacio de direcciones no cambia, es el mismo.

8. Explicar qué es una condición de carrera (*Race Condition*) y dar un ejemplo.
   Una condición de carrera es cuando dos o más hilos ingresan a modificar una variable global compartida o una sección de código que está en una región crítica. Un ejemplo sucede cuando se desea sumar a una variable contadora un 1 un determinado número de veces (n) y se hace por medio de hilos, por ejemplo 2: un hilo incrementa el contador n veces y el otro hilo lo incrementa otras n veces para dar un total de 2n veces.
9. ¿Qué es la sección crítica?
   La sección crítica es una variable compartida o una porción de código que genera una condición de carrera para los hilos que la están modificando. Es un recurso compartido.
10. ¿Cuáles son los tres criterios que se evalúan en la solución al problema de la sección crítica? Explicar cada uno
    - **Exclusión mutua:** Propiedad que garantiza que un solo hilo está ejecutando el código de la sección crítica.
    - **Equidad:** Cada hilo tiene igual probabilidad de adquirir el lock.
    - **Desempeño:** Relacionado con time overhead (tiempo tomado en la adquisición/liberación con o sin competencia).
11. ¿En qué consiste la solución de sincronización por hardware?
12. ¿Qué es mutex?, ¿Qué funciones usa?, ¿Qué características deben tener esas funciones?
    - Un mutex es un lock que es un objeto (en memoria) que proporciona exclusión mutua.
    - Usa funciones de adquired() y reléase(), o lock y unlock.
    - Adquired o lock indica que un hilo ha adquirido el lock y que este se está ejecutando en la sección crítica. Release o unlock indica que ningún objeto ha adquirido el lock.
13. ¿En qué consiste el concepto s*pin-waiting?*
    En la implementación de locks una de las formas de implementarlos es mediante el uso de instrucciones Load/Store. Esta implementación presenta un problema que está relacionado con el desempeño, es decir que la CPU realiza un gasto intensivo debido a la forma como el hilo chequea la bandera (*flag)*. El *spin-waiting* es el chequeo continuo del valor de la bandera del mutex. Es una técnica de espera activa.
14. ¿Cuáles son las diferencias entre una variable de condición y un lock?
    - Una variable de condición es útil cuando algún tipo de señalización debe tomar lugar entre hilos, si un hilo está esperando por otro para hacer algo antes de poder continuar. Hay dos rutinas importantes: wait que pone el hilo de llamada a dormir y la función signal que despierta a un hilo dormido.
    - Un lock es una estructura (en memoria) que se encarga de mantener la sección crítica protegida de accesos no autorizados o de que más de un hilo la ejecute a la vez.
15. ¿Cuáles son las similitudes entre las variables de condición y los locks?, ¿En qué casos estas primitivas (locks y variables de condición) podrían interactuar con el scheduler del sistema?
    - Las variables de condición y los locks son estructuras que manejan hilos a su manera y se usan juntas para garantizar la exclusión mutua en la región crítica. Se usan para esperar hasta que una variable cumpla alguna condición o evento. Son una cola de hilos en espera. Las VC reciben un lock como parámetro.
    - Las primitivas (locks y variables de condición) pueden interactuar con el scheduler del sistema cuando hay un cambio de contexto entre hilos y el sistema operativo decide, según su política de planificación efectuarlo.
16. ¿Qué es un semáforo?

Un semáforo es un objeto con un valor entero no negativo (asociado al estado). Puede ser usado como un lock o una variable de condición. Tiene dos funciones por las que se manipulan: sem_wait(), sem_post(). Es una primitiva de sincronización.

17. ¿Cuál es la diferencia entre un mutex y un semáforo? Explicar detalladamente qué hace la función wait() y post().

- La diferencia entre un mutex o un lock y un semáforo es que un lock es una estructura en memoria que tiene la capacidad de proporcionar exclusión mutua y tiene dos funciones que lo permiten manipular: adquired() y release() o lock() y unlock(). La función lock() permite adquirir el bloqueo para ingresar a la sección crítica y el unlock() permite liberar o desbloquear la sección crítica. Los semáforos son objetos con un valor entero no negativo que está asociado a su estado y tiene dos funciones que permiten manipularlos: wait() y post(). La función wait() decrementa el valor del semáforo en uno; lo que hace es que: si el valor del semáforo es mayor o igual a 1 la función retorna inmediatamente, en caso contrario el hilo llamador se suspende (wait) a la espera de una seña de post. La función post() incrementa el valor del semáforo en uno y si hay hilos a la espera en el semáforo, despierta uno de ellos.

18. Explicar cuál es la ventaja del uso de primitivas del tipo dormir (*park*) y despertar (*unpark*) con respecto a la técnica de espera activa *spin waiting*.

La ventaja del uso de primitivas del tipo dormir y despertar con respecto a la técnica de espera activa radica en la eficiencia y el uso de recursos del sistema. Esto es, que en la técnica de espera activa los hilos usan constantemente la CPU, hacen un gasto incensario de recursos, principalmente los hilos que están a la espera y no necesitan usarla realmente y las primitivas del tipo dormir y despertar ponen a dormir a los hilos que no están haciendo uso de CPU y despierta a los hilos que hacen una tarea o trabajo y usan la CPU.

19. Definir con las propias palabras qué es un *Deadlock*.

Un *Deadlock* se da cuando un hilo bloquea la ejecución de otro hilo debido a que un hilo necesita ser despertado por el otro hilo y el otro hilo necesita información del otro hilo. Se da un ciclo sin salida.

20. Definir con las propias palabras qué es *Starvation*.

Starvation es cuando uno o más hilos se quedan sin ser ejecutados durante un período de tiempo debido a que otros hilos están gastando recursos de CPU constantemente y no permiten que otros hilos la usen.

21. Enunciar 3 problemas clásicos de sincronización, ¿en qué consiste cada uno?

- Problema de productor/consumidor: Es un problema de sincronización que consiste en que se tienen uno o más consumidores y uno o más productores representados en hilos. Los hilos productores ponen datos en un buffer compartido limitado, en una posición específica y los hilos consumidores toman los datos que los hilos productores colocan en el buffer. La sección crítica es el buffer, que es una variable compartida y es protegida. El productor y el consumidor se ejecutan a tasas diferentes: no hay serialización de uno tras del otro, las tareas son independientes.
- Problema del lector/escritor: Problema en el cual se tiene un número de operaciones concurrentes sobre los datos compartidos que sólo son lecturas y escrituras. Las siguientes restricciones se aplican:
  - **Escritura:** Solo un escritor puede acceder a los datos compartidos a la vez. El escritor cambia los datos (acceso a la región crítica).

- **Lectura:** Sólo lee los datos. Es posible realizar una lectura de los datos simultáneamente siempre y cuando no haya escrituras realizándose.

- Problema de la cena de los filósofos: Es un problema de concurrencia en donde hay cinco filósofos sentados en una mesa. Entre cada par de filósofos hay un solo palillo chino (cinco en total). Cada filósofo tiene momentos para pensar y no necesitan palillos chinos, y momentos donde ellos comen. Con el fin de comer, un filósofo necesita dos palillos, uno que está a su izquierda y el otro que está a su derecha. Los filósofos compiten por estos palillos.

22. ¿Cuál es la estructura de código para resolver cada uno de los problemas clásicos de sincronización usando semáforos? Explicar el funcionamiento.

- Problema de productor/consumidor:

```
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;    // Line F1
    fill = (fill + 1) % MAX; // Line F2
}

int get() {
    int tmp = buffer[use];   // Line G1
    use = (use + 1) % MAX;   // Line G2
    return tmp;
}
```

Figura 1: Rutinas put y get

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);       // Line P1
        sem_wait(&mutex);       // Line P1.5 (lock)
        put(i);                 // Line P2
        sem_post(&mutex);       // Line P2.5 (unlock)
        sem_post(&full);        // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);        // Line C1
        sem_wait(&mutex);       // Line C1.5 (lock)
        int tmp = get();        // Line C2
        sem_post(&mutex);       // Line C2.5 (unlock)
        sem_post(&empty);       // Line C3
        printf("%d\n", tmp);
    }
}
```

Figura 2: Rutinas producer y consumer

La *Figura 1* muestra la implementación de las rutinas put() y get(), que son usadas por el productor y consumidor, respectivamente. En la función put() el productor coloca un valor en el buffer limitado en una posición específica comenzando desde la posición 0 y aumenta en 1 la posición a medida que se va llenando el buffer. En la función get() el consumidor toma un elemento del arreglo en una determinada posición, lo guarda en una variable y aumenta el índice a medida que se van tomando los valores del buffer.

La *Figura 2: Rutinas producer y consumer* muestra la implementación de los métodos que realizan los productores y consumidores. El método producer() va llenando el buffer por medio de un ciclo for con una variable que incrementa los índices del arreglo y se usan semáforos empty, mutex y full para garantizar la exclusión mutua que tiene que tener la región crítica para su buen funcionamiento. El método consumer() también por medio de un ciclo va consumiendo los datos que el productor va poniendo en el buffer. Este consumo es protegido por mutex que garantizan el acceso a la región crítica protegido.

- Problema de lector/escritor:

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // allow ONE writer/MANY readers
  int   readers;   // #readers in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}
```

Figura 3: Implementación lector/escritor parte 1

```
void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

Figura 4: Implementación lector/escritor parte 2

La *Figura 3* muestra reader/writer lock (rwlock_t) que es un tipo especial de lock implementado para operaciones de lectura y escritura. La implementación es con semáforos y tiene las siguientes variables: readers que significa el número de hilos de lectura que se encuentran actualmente en la estructura de los datos, writelock que es el lock de escritura, y lock que es el lock de lectura. También muestra el método rwlock_init que se encarga de la inicialización de la estructura rwlock_t. La función rwlock_acquire_readlock permite al lector adquirir el lock.

La *Figura 4* muestra las funciones rwlock_release_readlock que permite al lector liberar el lock, rwlock_acquire_writelock permite al escritor adquirir el lock y rwlock_release_writelock permite al escritor liberar el lock.

Las anteriores funciones corresponden a las operaciones de sincronización básicas para la lectura de los datos usando semáforos.

- Problema de la cena de los filósofos:

```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

```
int left(int p)  { return p; }
int right(int p) { return (p + 1) % 5; }
```

Figura 5: Ciclo básico de cada filósofo *Figura 6: Referencia a los palillos*

La *Figura 5* muestra el ciclo básico de cada filósofo, asumiendo que cada uno tiene un hilo cuyo identificador es único y varia desde 0 hasta 4. La *Figura 6* muestra las funciones left(int p) que permiten que el filósofo p se refiera al palillo de la izquierda y la función right(int p), permite que el mismo filósofo p se refiera al palillo de su derecha.

Se necesitan 5 semáforos para cada platillo: sem_t forks[5].

```
void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

```
void get_forks(int p) {
    if (p == 4) {
        sem_wait(&forks[right(p)]);
        sem_wait(&forks[left(p)]);
    } else {
        sem_wait(&forks[left(p)]);
        sem_wait(&forks[right(p)]);
    }
}
```

Figura 7: Rutina put_forks *Figura 8: Rutina get_forks*

La *Figura 7* muestra la implementación de la función put_forks(int p), donde se libran los palillos de la izquierda y de la derecha de cada filósofo y la *Figura 8* muestra la función get_forks(int p) en donde se adquieren los palillos de la izquierda y de la derecha de cada filósofo según su número, sin que ocurra un *Deadlock*.

23. Demostrar que, si las funciones wait() y post() de un semáforo no se ejecutan atómicamente, entonces la exclusión mutua puede ser violada.

Para demostrar que si las funciones wait() y post() no se ejecutan atómicamente, la exclusión mutua puede ser violada, se supone que se tiene un semáforo binario protegiendo a la sección crítica, las operación wait() que disminuye el valor de un semáforo. Si el semáforo tiene un valor de 0, el proceso se bloquea hasta que el semáforo sea mayor que 0, la rutina post() que aumenta el valor del semáforo. Si hay hilos bloqueados, uno de ellos puede continuar. Se tiene además que la exclusión mutua impide que más de un hilo ejecute la sección crítica. Además se tiene la no ejecución atómica, lo que significa que las dos operaciones wait() y post() pueden ser interrumpidas, si no se ejecutan atómicamente se pueden producir condiciones de carrera.

Se tienen dos hilos h1 y h2, ambos queriendo acceder a la región crítica protegida por el semáforo del principio inicialmente en 1. Además, se asume que las anteriores funciones no son atómicas y pueden ser interrumpidas.

El hilo h1 llama a wait(s), donde s es el semáforo, el valor del semáforo se disminuye en 1, quedando en 0 y h1 continuaría. Si el semáforo estaba en 0, h1 se bloquearía. Dado que wait() no es atómica, el hilo puede ser interrumpido en medio de la ejecución de wait(), antes de que ocurra la operación wait(), es decir que el valor se actualice completamente, el sistema operativo decide darle la CPU al hilo h2. El hilo h2 llama a wait(s), y el valor del semáforo no ha sido completamente actualizado a 0, ya que h1 no ha terminado de ejecutar wait(s), por lo que h2 actualiza el valor del semáforo a 0, ya que este se encontraba en 1, lo que permite que h1 y h2 estén en la región crítica al mismo tiempo, lo que lleva a la violación de la propiedad de la exclusión mutua de la sección crítica.

24. Explicar por qué los *spinlocks* no son apropiados para un sistema monoprocesador, a menudo son usados en sistemas multiprocesador.

En los sistemas monoprocesador se puede dar el uso ineficiente de la CPU, sólo puede ejecutarse un hilo en un momento dado, ya que un hilo puede esperar a que se libere el bloqueo y esta espera consume CPU. En los sistemas multiprocesador, los spinlocks son útiles porque otros procesadores pueden liberar el bloqueo, y otros hilos que tengan tareas que hacer pueden usar los procesadores y que el sistema no se quede bloqueado por el hilo que está bloqueado.

25. Explicar por qué deshabilitar interrupciones no es un mecanismo apropiado para implementar primitivas de sincronización en sistemas multiprocesador.

En un sistema multiprocesador no es un mecanismo apropiado deshabilitar interrupciones debido a que múltiples procesadores pueden ejecutar hilos concurrentemente y cuando hay una interrupción que es deshabilitada en un procesador, se puede alterar el trabajo correcto del sistema, puede haber condiciones de carrera o problemas de sincronización, además de poder generar *deadlocks* y problemas de rendimiento, entre otros.

26. Considerar el interbloqueo del tráfico que se muestra en la siguiente figura 1.

a. Indicar el número de cada una de las 4 condiciones necesarias para que exista un interbloqueo. Mostrar que cada condición se presenta en este ejemplo.
    1. Exclusión mutua: Todos los carros o (hilos) esperan salir o (ejecutarse).
    2. Retener y esperar: Cada hilo mantiene los recursos que ya se le han asignado, a la vez que espera adquirir los demás.
    3. No apropiativo: Significa que un recurso sólo puede ser liberado de forma voluntaria por el proceso al que se le ha concedido su uso.
    4. Espera circular: Los hilos interbloqueados forman una cadena circular, de modo que cada hilo mantiene uno o más de los recursos que son solicitados por el siguiente proceso de la cadena.
b. Establecer una regla simple para evitar interbloqueos en este sistema. Explicar por qué es útil la técnica que se propone.
    Una regla simple para evitar interbloqueos en este sistema puede ser la implementación de semáforos en algunas calles dependiendo del tráfico y la hora. Es útil porque así se controla el tráfico que circula por las calles y en qué dirección sucede.

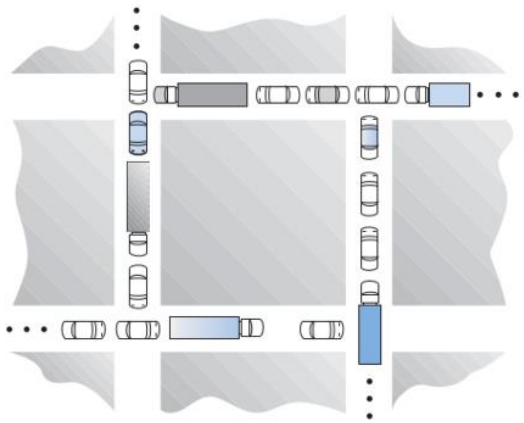

Figura 1.