# 1. Mencione al menos 3 ejemplos donde pueda encontrarse concurrencia

- Descargar múltiples archivos al mismo tiempo
- Múltiples usuarios conectándose a un mismo servidor
- Hacer tarea mientras estás en clase virtual

# 2. Escriba una definición de concurrencia. Diferencie procesamiento secuencial, concurrente y paralelo.

La concurrencia es la capacidad de ejecutar múltiples actividades en paralelo, está caracterizada por el no determinismo.
Procesamiento secuencial se refiere a ejecutar una tarea a la vez, en un orden determinado, con un único flujo de ejecución donde cada instrucción se ejecuta de a una por vez
El procesamiento concurrente se refiere a una sola entidad de procesamiento, la cual dedica una parte de tiempo a cada una de las tareas
El procesamiento paralelo refiere a varias entidades de procesamiento, ejecutando en todas tareas distintas al mismo tiempo

# 3. Describa el concepto de deadlock y qué condiciones deben darse para que ocurra.

Deadlock sucede cuando dos o más procesos se quedan esperando a que otro libere un recurso compartido
Condiciones para que ocurra (se deben cumplir las cuatro simultáneamente)

- Recursos reusables serialmente
  - Los procesos comparten recursos que pueden usar con exclusión mutua
  - Hay exclusión mutua
- Adquisición incremental
  - Los procesos mantienen los recursos que poseen mientras esperan adquirir recursos adicionales
  - Un proceso que ya tiene uno o más recursos, puede pedir más y mientras espera, sigue reteniendo los que ya posee
- No preemption
  - Una vez que los recursos son adquiridos por un proceso, los mismos no pueden quitarse de manera forzada, sinó que son liberados voluntariamente
- Espera cíclica
  - Existe una cadena circular de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir

# 4. Defina inanición. Ejemplifique.

La inanición ocurre cuando un proceso que desea ejecutar, nunca es ejecutado ya que otro toma el lugar. Por ejemplo, una cola de procesos en espera, ordenada por el tiempo que necesitan para ejecutar. Si un proceso tiene un tiempo extraordinariamente grande y todos los demás procesos tienen tiempos que siempre son más pequeños, el proceso de mayor tiempo nunca se va a ejecutar.

# 5. ¿Qué entiende por no determinismo? ¿Cómo se aplica este concepto a la ejecución concurrente?

El no determinismo es una característica de la concurrencia, donde varias ejecuciones dan lugar a resultados distintos, dado que el orden en que se ejecutan las cosas no está determinado.
Este concepto aplica a la ejecución concurrente, ya que al tener varios procesos ejecutando al mismo tiempo, se pueden tener resultados distintos entre varias ejecuciones
Como por ejemplo, si tengo 10 procesos que cuentan números del 1 al 100, y en un momento determinado imprimo la cantidad que van contando todos los procesos, esta cantidad es muy poco probable que sea siempre la misma

# 6. Defina comunicación. Explique los mecanismos de comunicación que conozca.

La comunicación entre procesos concurrentes indica el modo en que se organizan y transmiten datos entre las tareas concurrentes. Esta organización requiere especificar protocolos para controlar el progreso y la corrección.
Los mecanismos de comunicación pueden ser

- Por memoria compartida
  - Los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella
  - Como no se puede operar simultáneamente sobre un mismo dato, obliga a bloquear y liberar el acceso al mismo
  - Los tipos que dimos son
    - Variables compartidas (de datos o de sincronización)
    - Semáforos
    - Monitores
- Por pasaje de mensajes
  - Es necesario establecer un canal (lógico o físico) para transmitir información entre procesos
  - Para que la comunicación sea efectiva, los procesos deben saber cuándo tienen mensajes para leer y cuándo deben transmitir mensajes

# 7.

## A) Defina sincronización. Explique los mecanismos de sincronización que conozca.

La sincronización es la posesión de información acerca de otro proceso para coordinar actividades
Los mecanismos de sincronización son

- Por exclusión mutua
  - Asegurar que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo
- Por condición
  - Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada

## B) ¿En un programa concurrente pueden estar presentes más de un mecanismo de sincronización? En caso afirmativo, ejemplifique

Sí, en un programa concurrente puede haber varios mecanismos de sincronización presentes. Por ejemplo, si necesito hacer una barrera con un contador. Todos los procesos van a tener que contar en una variable compartida utilizando exclusión mutua, y al llegar a una cantidad determinada, se activa la condición para que todos los procesos avancen

# 8. ¿Qué significa el problema de “interferencia” en programación concurrente? ¿Cómo puede evitarse?

La interferencia ocurre cuando un programa toma una acción que invalida las suposiciones hechas por otro proceso. Esto ocurre mayormente cuando hay un acceso en simultáneo a un mismo recurso y al menos uno de ellos lo modifica. Por ejemplo

```
int x, y, z;

process A1
 { ....
   y = 0;
   ....
 }

process A2
 { .....
   if (y <> 0)   z = x/y;
   .....
 }
```

El proceso A2 podría pasar el if, y luego el proceso A1 modificarlo, provocando una división por 0

Para evitarla, hay que realizar esas operaciones con exclusión mutua, por ejemplo con el uso de semáforos. Usando tecnicas de sincronizacion como exclusion mutua o sincronizacion por condicion

# 9. ¿En qué consiste la propiedad de “A lo sumo una vez” y qué efecto tiene sobre las sentencias de un programa concurrente? De ejemplos de sentencias que cumplan y de sentencias que no cumplan con ASV.

La propiedad de a lo sumo una vez consiste en que como máximo puede haber una variable compartida y puede ser referenciada como máximo una vez. Esta propiedad da el efecto de que la ejecución de una sentencia de asignación parezca atómica, puesto que una variable compartida será leída o escrita sólo una vez. Si no se cumple esta propiedad, hay riesgo de interferencia

**Ejemplo 1**

No hay ninguna referencia crítica en ningún proceso, por lo que no hay interferencia

```
int x = 0, y = 0;
co x = x + 1 // y = y + 1 oc;
```

**Ejemplo 2**

Hay una sola referencia crítica en el primer proceso ( `y` ), y ninguna en el proceso 2. Así que no hay interferencia

```
int x = 0, y = 0;
co x = y + 1 // y = y + 1 oc;
```

**Ejemplo 3**

Ninguna asignación satisface ASV.

```
int x = 0, y = 0;
co x = y + 1 // y = x + 1 oc:
```

# 10. Dado el siguiente programa concurrente:

```
x = 2; y = 4; z = 3;
co
    x = y - z // z = x * 2 // y = y - 1
oc
```

NOTA 1: Las instrucciones NO SON atómicas

NOTA 2: No es necesario que liste TODOS los resultados, pero sí los que sean representativos de las diferentes situaciones que pueden darse

## A) ¿Cuáles de las asignaciones dentro de la sentencia co cumplen con ASV?. Justifique claramente.

- `y = y - 1`
- `z = x * 2`
  Pero por qué? Si tiene una referencia crítica del lado derecho y del lado izq está siendo utilizada por otro proceso

Una sentencia de asignación `x = e` satisface la propiedad de "A lo sumo una vez" si:

- `e` contiene a lo sumo una referencia crítica y `x` no es referenciada por otro proceso, o sinó
- `e` no contiene referencias críticas, en cuyo caso `x` puede ser leída por otro proceso

Una expresión `e` que no está en una sentencia de asignación satisface la propiedad de "A lo sumo una vez" si no contiene más de una referencia crítica

> Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez

## B) Indique los resultados posibles de la ejecución

- x = 1, y = 3, z = 2
- x = 0, y = 3, z = 4
- x = 1, y = 3, z = 4

# 11. Defina acciones atómicas condicionales e incondicionales. Ejemplifique.

**Acción atómica condicional**:
Acciones que sólo se ejecutan si se cumple una condición específica (están hechas con sentencias await)

**Acción atómica incondicional**:
Es una acción que no está hecha con sentencias await (sentencia común y corriente que utiliza variables locales al proceso) o está hecha con sentencias await sin condición booleana. Se ejecutan sin necesidad de una condición.

# 12. Defina propiedad de seguridad y propiedad de vida.

**Seguridad**:

- Nada malo le ocurre a un proceso (asegura estados consistentes)
- Ejemplo: Exclusión mutua, ausencia de interferencia, partial correctness (un programa termina y la finalización es correcta, pero no se asegura que siempre termine)

**Vida:**

- Eventualmente ocurre algo bueno con una actividad, es decir, progresa y no hay deadlocks
- Ejemplo: Ausencia de deadlock, ausencia de inanición, que un proceso eventualmente alcanza su SC

# 13. ¿Qué es una política de scheduling? Relacione con fairness. ¿Qué tipos de fairness conoce? ¿Por qué las propiedades de vida dependen de la política de scheduling?

Una política de scheduling determina la manera en la que se va a seleccionar la próxima acción a ejecutar. La fairness trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás.
Los tipos de fairness son

- Incondicional
  - Si toda acción atómica incondicional que es elegible, eventualmente es ejecutada
- Débil
  - Si es incondicionalmente fair y además, toda acción atómica condicional que se vuelve elegible, eventualmente es ejecutada, asumiendo que su condición se vuelve verdadera y permanece así hasta que es vista por el proceso que ejecuta la acción atómica condicional
- Fuerte
  - Es incondicionalmente fair y además, toda acción atómica condicional que se vuelve elegible, eventualmente es ejecutada, pues su guarda se convierte en verdadera con infinita frecuencia
  Las propiedades de vida dependen de la política de scheduling dado que si nunca se le da lugar a un proceso para que se ejecute, nunca podría progresar.

# 14. Dado el siguiente programa concurrente, indique cuál es la respuesta correcta (justifique claramente)

```
int a = 1; b = 0;
co <await(b = 1) a = 0> // while (a = 1) { b = 1; b = 0; } oc
```

## A) Siempre termina

## B) Nunca termina

## C) Puede terminar o no

Esta opción es la correcta, depende de la política de scheduling y la cant de cpus dado que si el primer proceso tiene la posibilidad de ver cuando b = 1, va a ejecutar, pero si el proceso dos siempre hace toda la operación y llega al b = 0, nunca va a terminar.

# 15. ¿Qué propiedades que deben garantizarse en la administración de una sección crítica en procesos concurrentes? ¿Cuáles de ellas son propiedades de seguridad y cuáles de vida? En el caso de las propiedades de seguridad, ¿Cuál es en cada caso el estado “malo” que se debe evitar? Para cada propiedad explique una solución que no la cumpla.

Las propiedades que deben garantizarse en la administración de una sección crítica en procesos concurrentes son

- Exclusión mutua
  - A lo sumo un proceso está en su SC
  - Es una propiedad de seguridad
  - El caso "malo" que se debe evitar es el caso donde haya más de un proceso en su sección crítica
  - Ejemplo donde no se cumple

```
int x = 0;
Process EjemploSC[ id = 1 .. 2] {
    while(true){
        x++;
    }
}
```

- Ausencia de deadlock
  - Si dos o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito
  - Es una propiedad de vida
  - Ejemplo donde no se cumple

```
x = 0, y = 0;
Process EjemploDeadlock1 {
    while(y == 0) skip;
    Otras tareas
    x++;
}

Process EjemploDeadlock2 {
    while(x == 0) skip;
    Otras tareas
    y++;
}
```

- Ausencia de demora innecesaria
  - Si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC
  - Es una propiedad de vida
  - Ejemplo donde no se cumple

```
x = 1;
Process EjemploDemoraInnecesaria1 {
    while(true){
        while(x == 2) skip;
        SC;
        x = 2;
        SNC;
    }
}
```

```
Process EjemploDemoraInnecesaria2 {
    while(true){
        while(x == 1) skip;
        SC;
        x = 1;
        SNC;
    }
}
```

- Eventual entrada
  - Un proceso que intenta entrar a su SC tiene posibilidades de hacerlo
  - Es una propiedad de vida
  - Un ejemplo donde nunca se cumple es donde se tiene procesos con prioridad y siempre los procesos con mayor prioridad le ganan al que tiene la menor prioridad
  - tie breaker sin alternancia

# 16. Resuelva el problema de acceso a sección crítica para N procesos usando un proceso coordinador. En este caso, cuando un proceso SC[i] quiere entrar a su sección crítica le avisa al coordinador, y espera a que éste le otorgue permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador. Desarrolle una solución de grano fino usando únicamente variables compartidas (ni semáforos ni monitores). C

Se puede hacer sin busy waiting? Se puede usar Test and Set, Fetch and Add, Compare and Swap?

```
int solicitud[1 .. N] = [(N) 0];
int permiso[1 .. N] = [(N) 0];
int salida[1 .. N] = [(N) 0];

Process ProcesoSC[id = 1 .. N] {
    while(true) {
        SNC

        solicitud[id] = 1;
        while(permiso[id] == 0) skip;

        permiso[id] = 0
        solicitud[id] = 0;

        SC

        salida[id] = 1;
    }
```

```
}

Process Coordinador {
    while(true) {
        for(int i = 1; i <= N; i++) {
            if(solicitud[i] == 1) {
                permiso[i] = 1;
                while(salida[i] == 0) skip;
                salida[i] = 0;
            }
        }
    }
}
```

# 17. ¿Qué mejoras introducen los algoritmos Tie-breaker, Ticket o Bakery en relación a las soluciones de tipo spin locks? C

## Tie breaker

- Permite controlar el orden en el que los procesos acceden a la sección crítica, pero este "empate" sólo ocurre entre dos procesos a la vez
- Complejo

## Ticket

- Permite controlar el orden en el que los procesos acceden a la sección crítica, pero requiere de que exista la instrucción Fetch and Add para ser fair
- Posible overflow de número y próximo
- Si no existe Fetch and Add, se debe simular con otra SC y la solución puede no ser fair

## Bakery

- Algoritmo más complejo pero es fair y no requiere instrucciones especiales

# 18. Analice las soluciones para las barreras de sincronización desde el punto de vista de la complejidad de la programación y de la performance. C

## Contador compartido

```
int cantidad = 0;
process Worker[i=1 to n]
{ while (true)
        {  código para implementar la tarea i;
           FA (cantidad, 1);
           while (cantidad <> n) skip;
        }
}
```

### Complejidad

- Simple y con poca cantidad de variables

### Performance

- Hay que ver cuándo resetear el contador a 0
- Contención de memoria dado que todos los procesos acceden a la misma variable
- Ineficiente

## Flags y Coordinadores

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);

process Worker[i=1 to n]
{ while (true)
        {     código para implementar la tarea i;
              arribo[i] = 1;
              while (continuar[i] == 0) skip;
              continuar[i] = 0;
         }
}

process Coordinador
{ while (true)
        {     for [i = 1 to n]
                   { while (arribo[i] == 0)  skip;
                     arribo[i] = 0;
                   }
              for [i = 1 to n] continuar[i] = 1;
         }
}
```

## Complejidad

- Más complejo de implementar que el contador compartido
- Más variables que el contador compartido

## Performance

- Requiere un proceso extra
- El tiempo de ejecución del coordinador es proporcional a N
- Se reduce la contención de memoria dado que cada proceso toca unicamente sus variables
- Requiere un procesador extra

# Combining Tree Barrier

## Complejidad

- Alta complejidad
- Difícil debugging
- Mayor coordinación entre procesos

## Performance

- Mejor escalabilidad con orden Log N
- División de trabajo

## Butterfly Barrier

```
int E = log(N);
int arribo[1:N] = ([N] 0);

process P[i=1..N]
{  int j;
   while (true)
      {  //Sección de código anterior a la barrera.
         //Inicio de la barrera
         for (etapa = 1; etapa <= E; etapa++)
            {  j =  (i-1) XOR (1<<(etapa-1));     //calcula el proceso con cual sincronizar
               while (arribo[i] == 1) → skip;
               arribo[i] = 1;
               while (arribo[j] == 0) → skip;
               arribo[j] = 0;
            }
         //Fin de la barrera
         //Sección de código posterior a la barrera.
      }
}
```

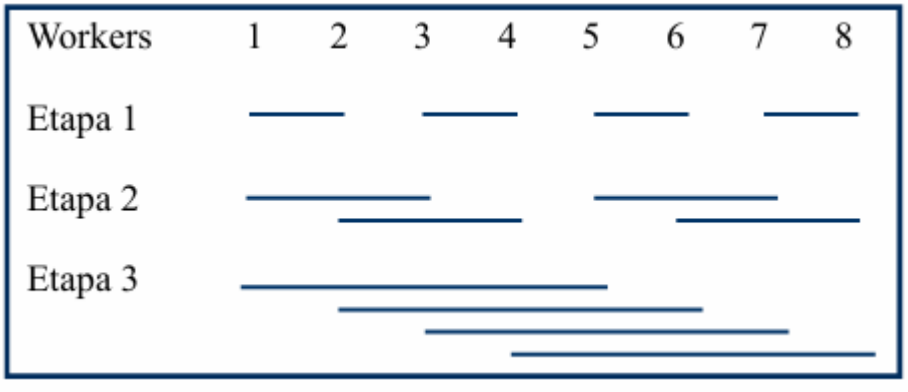
### Complejidad

- Muy alta, requiere operaciones bit wise
- Sólo funciona con un N potencia de 2
- Difícil debugging

### Performance

- Muy buena, del orden Log N
- No requiere coordinador aparte, cada worker se sincroniza con otro
- Buena escalabilidad

# 19. Explique gráficamente cómo funciona una butterfly barrier para 8 procesos usando variables compartidas.




- $\log_2 n$ etapas
- Cada worker sincroniza con uno distinto en cada etapa

- En la etapa S, un worker sincroniza con otro a distancia $2^{S-1}$
- Cuando cada worker pasó $\log_2 n$ etapas, todos pueden seguir