# Sincronización de procesos

David Alejandro González Márquez

Clase disponible en: https://github.com/fokerman/computingSystemsCourse

# Sincronización de Procesos

En un entorno de multiprogramación, donde **más de un proceso** es ejecutado para resolver un problema, surge la necesidad de mecanismos de **comunicación** y **sincronización** entre procesos.

# Sincronización de Procesos

En un entorno de multiprogramación, donde **más de un proceso** es ejecutado para resolver un problema, surge la necesidad de mecanismos de **comunicación** y **sincronización** entre procesos.

A un conjunto de procesos que cooperan entre sí para resolver un problema los llamamos **procesos cooperativos**.

Por ejemplo, los múltiples procesos de un navegador web, o el conjunto de rutinas ejecutadas en paralelo por un software de procesamiento de imagenes, o incluso una IDE de desarrollo. Cualquier aplicación que utilice eficientemente los recursos del suele utilizar multiples procesos o *threads*

# Sincronización de Procesos

En un entorno de multiprogramación, donde **más de un proceso** es ejecutado para resolver un problema, surge la necesidad de mecanismos de **comunicación** y **sincronización** entre procesos.

A un conjunto de procesos que cooperan entre sí para resolver un problema los llamamos **procesos cooperativos**.

Por ejemplo, los múltiples procesos de un navegador web, o el conjunto de rutinas ejecutadas en paralelo por un software de procesamiento de imagenes, o incluso una IDE de desarrollo. Cualquier aplicación que utilice eficientemente los recursos del suele utilizar multiples procesos o *threads*

**Mecanismos de comunicación**

- **Memoria compartida**
- **Pasaje de mensajes**

**Mecanismos de sincronización**

- **Mutex**
- **Semaforos**

# Condición de carrera (*race condition*)

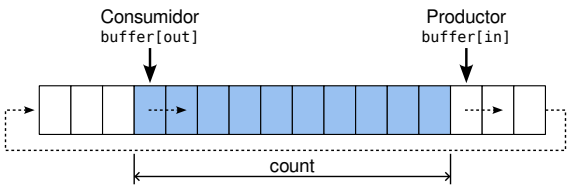
Supongamos el siguiente código:

**Proceso Productor**

```
while(true) {
   item = newItem();

   while(count == BUFFERSIZE){};

   buffer[in] = item;
   in = (in + 1) % BUFFERSIZE;
   count++;
}
```

**Proceso Consumidor**

```
while(true) {
    while(count == 0){};

    item = buffer[out]
    out = (out + 1) % BUFFERSIZE;
    count--;

    processItem(item);
}
```




El productor construye items y los guarda en un buffer, mientras que el consumidor, toma los items del buffer y los procesa.

# Condición de carrera (*race condition*)

La cantidad de items en el *buffer* es una variable compartida entre el productor y el consumidor.

Al estar compartida, **su edición esta sujeta a condiciones de carrera.**

Veamos un ejemplo en ASM:

# Condición de carrera (*race condition*)

La cantidad de items en el *buffer* es una variable compartida entre el productor y el consumidor.

Al estar compartida, **su edición esta sujeta a condiciones de carrera.**

Veamos un ejemplo en ASM:

```
...
MOV EAX, [count]
ADD EAX, 1
MOV [count], EAX
...
```

```
...
MOV EAX, [count]
SUB EAX, 1
MOV [count], EAX
...
```

El *scheduler* no nos garantiza que se ejecute primero el proceso productor y luego el consumidor. Tampoco nos garantiza, **en el caso de un *scheduler* apropiativo**, que se ejecuten todas las instrucciones una despues de la otra.

# Condición de carrera (*race condition*)

La cantidad de items en el *buffer* es una variable compartida entre el productor y el consumidor.

Al estar compartida, **su edición esta sujeta a condiciones de carrera.**

Veamos un ejemplo en ASM:

```
...
MOV EAX, [count]
ADD EAX, 1
MOV [count], EAX
...
```

```
...
MOV EAX, [count]
SUB EAX, 1
MOV [count], EAX
...
```

El *scheduler* no nos garantiza que se ejecute primero el proceso productor y luego el consumidor. Tampoco nos garantiza, **en el caso de un *scheduler* apropiativo**, que se ejecuten todas las instrucciones una despues de la otra.

Por lo tanto el valor final de `count` puede **no ser el esperado**, veamos un ejemplo.

# Condición de carrera (*race condition*)

Supongamos una posible ejecución **concurrente** de ambos procesos.

# Condición de carrera (*race condition*)

Supongamos una posible ejecución **concurrente** de ambos procesos.

| Proceso Productor | Proceso Consumidor | Valor de count |
|---|---|---|
| `...` | | X |
| `MOV EAX, [count]` | | X |
| `ADD EAX, 1` | | X |
| | `...` | X |
| | `MOV EAX, [count]` | X |
| | `SUB EAX, 1` | X |
| | `MOV [count], EAX` | X-1 |
| | `...` | X-1 |
| `MOV [count], EAX` | | X+1 |
| `...` | | X+1 |

# Condición de carrera (*race condition*)

Supongamos una posible ejecución **concurrente** de ambos procesos.

| Proceso Productor | Proceso Consumidor | Valor de count |
|---|---|---|
| `...` | | X |
| `MOV EAX, [count]` | | X |
| `ADD EAX, 1` | | X |
| | `...` | X |
| | `MOV EAX, [count]` | X |
| | `SUB EAX, 1` | X |
| | `MOV [count], EAX` | X-1 |
| | `...` | X-1 |
| `MOV [count], EAX` | | X+1 |
| `...` | | X+1 |

El *scheduler* puede desalojar a los procesos en **cualquier momento** de su ejecución.
Por lo tanto, se puede producir un entrelazamiento (**interleaving**) de los procesos.

# Condición de carrera (*race condition*)

Supongamos una posible ejecución **concurrente** de ambos procesos.

| Proceso Productor | Proceso Consumidor | Valor de count |
|---|---|---|
| `...` | | X |
| `MOV EAX, [count]` | | X |
| `ADD EAX, 1` | | X |
| | `...` | X |
| | `MOV EAX, [count]` | X |
| | `SUB EAX, 1` | X |
| | `MOV [count], EAX` | X-1 |
| | `...` | X-1 |
| `MOV [count], EAX` | | X+1 |
| `...` | | X+1 |

El *scheduler* puede desalojar a los procesos en **cualquier momento** de su ejecución.
Por lo tanto, se puede producir un entrelazamiento (**interleaving**) de los procesos.
No tenemos garantías de la **atomicidad** de secuencias de instrucciones. ¿qué hacemos?

# Condición de carrera (*race condition*)

En procesos cooperativos, puede ocurrir que no todas las posibles ejecuciones entrelazadas generen resultados válidos. Es decir, que exista una **traza de ejecución** que genere un resultado invalido, exponiendo una **condición de carrera** en nuestro código.

# Condición de carrera (*race condition*)

En procesos cooperativos, puede ocurrir que no todas las posibles ejecuciones entrelazadas generen resultados válidos. Es decir, que exista una **traza de ejecución** que genere un resultado invalido, exponiendo una **condición de carrera** en nuestro código.

Para resolver este problema, nuestro código debe **restringir las trazas de ejecución** que lleven a resultados invalidos.

# Condición de carrera (*race condition*)

En procesos cooperativos, puede ocurrir que no todas las posibles ejecuciones entrelazadas generen resultados válidos. Es decir, que exista una **traza de ejecución** que genere un resultado invalido, exponiendo una **condición de carrera** en nuestro código.

Para resolver este problema, nuestro código debe **restringir las trazas de ejecución** que lleven a resultados invalidos.

**¿Cómo?**, por medio de herramientas de **sincronización**.

# Condición de carrera (*race condition*)

En procesos cooperativos, puede ocurrir que no todas las posibles ejecuciones entrelazadas generen resultados válidos. Es decir, que exista una **traza de ejecución** que genere un resultado invalido, exponiendo una **condición de carrera** en nuestro código.

Para resolver este problema, nuestro código debe **restringir las trazas de ejecución** que lleven a resultados invalidos.

**¿Cómo?**, por medio de herramientas de **sincronización**.

Este es uno de los problemas más complejos de la Computación, y los Sistemas Operativos están no son la excepción.

# Condición de carrera (*race condition*)

En procesos cooperativos, puede ocurrir que no todas las posibles ejecuciones entrelazadas generen resultados válidos. Es decir, que exista una **traza de ejecución** que genere un resultado invalido, exponiendo una **condición de carrera** en nuestro código.

Para resolver este problema, nuestro código debe **restringir las trazas de ejecución** que lleven a resultados invalidos.

**¿Cómo?**, por medio de herramientas de **sincronización**.

Este es uno de los problemas más complejos de la Computación, y los Sistemas Operativos están no son la excepción.

Los diseñadores de sistemas buscan máximizar la eficiencia de sus soluciones generando múltiples procesos concurrentes, pero reduciendo su exposición a condiciones de carrera. Considerando que cuantas más restricciones agreguemos a las soluciones, más seriales serán y menos aprovecharán las ventajas de la multiprogramación.

# Sección Crítica

Una **sección crítica** es como se denomina a una parte de nuestro código que puede ser ejecutada por un solo proceso a la vez.

Una solución para el problema de sección crítica debe garantizar:

1. **Exclusión mutua**: Si un proceso ejecuta en la sección crítica, ningún otro proceso puede ejecutar.
2. **Avance**: La elección de que proceso va a entrar en la sección crítica se decide entre los procesos que no están en la sección crítica.
3. **Limitación de espera**: Existe un límite de espera para poder entrar a la sección crítica.

```
...
while(true) {
    ENTRY CRITICAL SECTION
    código en la sección crítica
    EXIT CRITICAL SECTION
    resto del código
}
...
```

# Soporte para la sincronización

Para poder implementar mecanismos de sección crítica en procesadores modernos.

Es necesario conocer el **modelo de memoria** con el que opere nuestro sistema.

- **Consistencia fuerte**
  Si un procesador modifica la memoria, el resto de los procesadores ven este cambio inmediatamente y de forma ordenada.
- **Consistencia debil**
  Si un procesador modifica la memoria, puede que no todos los procesadores vean este cambio inmediatamente.

Dependiendo que modelo de memoria tengamos, existe soporte en *hardware* para **forzar la sincronización** de la memoria.

Estos mecanismos se conocen como **barreras de memoria** (*memory fences*).

# Soporte para la sincronización

Existen además instrucciones especiales para implementar mecanismos de sincronización denomiandas **atómicas**.

Conceptualmente estas instrucciones realizan acciones de forma atómica.

- **Test And Set**

```
bool test_and_set(bool *target) {
    bool value = *target
    *target = true;
    return value
}
```

- **Compare And Swap**

```
int compare_and_swap(int *value, int expected, int newValue) {
    int tmp  *value;
    if (*value == expected) {
        *value = new_value;
    }
    return tmp;
}
```

# Exclusión mutua (mutex - *mutual exclusion*)

Utilizando estas instrucciones podemos implementar un mecanismo de sección crítica garantizando la exclusión mutua.

```
do {
  while(test_and_set(&lock)){};
  CODIGO EN LA SECCION CRITICA
  lock = false;
  RESTO DEL CODIGO
} while(true);
```

**Candados (*locks*)**

Variables *booleanas* compartidas.

**Para tomar el candado:**

Si el candado está en `true`, espero a que cambie en `false`.
Si el candado está en `false`, lo cambio a `true` para que nadie me interrumpa.

**Para liberar el candado:**

Cuando termino cambio el candado a `false`, para que otro lo pueda usar.

# Spin Locks

La desventaja de la implementación anterior es que para obtener el *lock* estamos realizando una **espera activa** (busy waiting).

**Estamos utilizando la CPU para no hacer nada productivo, solo esperar.**
**Impidiendo además que otros procesos puedan acceder al procesador.**

# Spin Locks

La desventaja de la implementación anterior es que para obtener el *lock* estamos realizando una **espera activa** (busy waiting).

**Estamos utilizando la CPU para no hacer nada productivo, solo esperar.**
**Impidiendo además que otros procesos puedan acceder al procesador.**

Sin embargo, los *spin locks* tienen ventajas, ya que no tenemos que realizar ningún cambio de contexto.

En determinados casos, cuando el tiempo de espera es muy corto y los eventos de sincronización se dan muy seguidos.
La solución de *spin locks* es la más eficiente.

# Spin Locks

La desventaja de la implementación anterior es que para obtener el *lock* estamos realizando una **espera activa** (busy waiting).

**Estamos utilizando la CPU para no hacer nada productivo, solo esperar.**
**Impidiendo además que otros procesos puedan acceder al procesador.**

Sin embargo, los *spin locks* tienen ventajas, ya que no tenemos que realizar ningún cambio de contexto.

En determinados casos, cuando el tiempo de espera es muy corto y los eventos de sincronización se dan muy seguidos.
La solución de *spin locks* es la más eficiente.

**¿y qué tal sí el sistema espera por nosotros?**

# Semáforos

Un semáforo es una variable entera que solo puede ser modificada por las operaciones:

- `signal()`: Incrementa el valor del semáforo en 1.

```
signal(int* S) {
  (*S)++;
}
```

- `wait()`: Espera hasta que el valor del semáforo sea menor o igual a 0, y luego decrementa el valor del semáforo en 1.

```
wait(int* S) {
  while(*S<=0){};
  (*S)--;
}
```

Estas dos primitivas deben ser ejecutadas de forma **atómica** por el sistema operativo.
Pero como las implementamos para no hacer una espera activa.

# Implementación de Semáforos

Se declara a nivel del sistema operativo la estructura del semáforo.

```
struct semaphore {
    int value;
    struct process *list;
}
```

```
void wait(semaphore* s) {
    s->value--;
    if(s->value < 0) {
        addList(s->list,current_process)
        sleep();
    }
}
```

Decrementa el semáforo y consulta su estado.
Si debe esperar, agrega al proceso a una lista
de procesos en espera para adquirir el semáforo.

```
void signal(semaphore* s) {
    s->value++;
    if(s->value <= 0) {
        process = removeList(s->list)
        wakeup(process);
    }
}
```

Incrementa el semáforo, compara si hay procesos
esperando, en cuyo caso los despierta.

# Ejemplo de uso de Semáforos

El uso de semáforos permite resolver diferentes problemas de sincronización.

En el ejemplo, el proceso A ejecuta CODE A y luego llama a `signal`.

Por otro lado, el proceso B llama `wait` y luego ejecuta CODE B.

**Proceso A**

```
...
CODE A
signal(sync)
...
```

**Proceso B**

```
...
wait(sync)
CODE B
...
```

# Ejemplo de uso de Semáforos

El uso de semáforos permite resolver diferentes problemas de sincronización.

En el ejemplo, el proceso A ejecuta CODE A y luego llama a `signal`.

Por otro lado, el proceso B llama `wait` y luego ejecuta CODE B.

**Proceso A**

```
...
CODE A
signal(sync)
...
```

**Proceso B**

```
...
wait(sync)
CODE B
...
```

Esto garantiza que CODE B se ejecute después de CODE A.

# Interbloqueo (*deadlock*)

Cualquier mecanismo de sincronización donde múltiples procesos esperen por un recurso nos puede llevar a una situación de *deadlock*. En particular el uso de semáforos.

Cuando conjunto de procesos se quedan simultáneamente esperando que ocurra un evento, que los mismos procesos que esperan deben generar, entonces estamos en un **deadlock**.

Ejemplo:

# Interbloqueo (*deadlock*)

Cualquier mecanismo de sincronización donde múltiples procesos esperen por un recurso nos puede llevar a una situación de *deadlock*. En particular el uso de semáforos.

Cuando conjunto de procesos se quedan simultáneamente esperando que ocurra un evento, que los mismos procesos que esperan deben generar, entonces estamos en un **deadlock**.

Ejemplo:

A no continúa hasta que B no envíe un `signal()`.

# Interbloqueo (*deadlock*)

Cualquier mecanismo de sincronización donde múltiples procesos esperen por un recurso nos puede llevar a una situación de *deadlock*. En particular el uso de semáforos.

Cuando conjunto de procesos se quedan simultáneamente esperando que ocurra un evento, que los mismos procesos que esperan deben generar, entonces estamos en un **deadlock**.

Ejemplo:

A no continúa hasta que B no envíe un `signal()`.
Pero B no continúa hasta que C no envíe un `signal()`.

# Interbloqueo (*deadlock*)

Cualquier mecanismo de sincronización donde múltiples procesos esperen por un recurso nos puede llevar a una situación de *deadlock*. En particular el uso de semáforos.

Cuando conjunto de procesos se quedan simultáneamente esperando que ocurra un evento, que los mismos procesos que esperan deben generar, entonces estamos en un **deadlock**.

Ejemplo:

A no continúa hasta que B no envíe un `signal()`.
Pero B no continúa hasta que C no envíe un `signal()`.
Pero C no continúa hasta que D no envíe un `signal()`.

# Interbloqueo (*deadlock*)

Cualquier mecanismo de sincronización donde múltiples procesos esperen por un recurso nos puede llevar a una situación de *deadlock*. En particular el uso de semáforos.

Cuando conjunto de procesos se quedan simultáneamente esperando que ocurra un evento, que los mismos procesos que esperan deben generar, entonces estamos en un **deadlock**.

Ejemplo:

A no continúa hasta que B no envíe un `signal()`.
Pero B no continúa hasta que C no envíe un `signal()`.
Pero C no continúa hasta que D no envíe un `signal()`.
Pero D no continúa hasta que A no envíe un `signal()`.

# Interbloqueo (*deadlock*)

Cualquier mecanismo de sincronización donde múltiples procesos esperen por un recurso nos puede llevar a una situación de *deadlock*. En particular el uso de semáforos.

Cuando conjunto de procesos se quedan simultáneamente esperando que ocurra un evento, que los mismos procesos que esperan deben generar, entonces estamos en un **deadlock**.

Ejemplo:

A no continúa hasta que B no envíe un `signal()`.
Pero B no continúa hasta que C no envíe un `signal()`.
Pero C no continúa hasta que D no envíe un `signal()`.
Pero D no continúa hasta que A no envíe un `signal()`.

También se puede dar la situación en que un proceso espera indefinidamente.
Esta situación se conoce como **inanición** o ***starvation***.

# *Deadlock*: ejemplo gráfico

# *Deadlock*: ejemplo gráfico

# Barreras o *barriers*

Existen problemas donde es necesario varios procesos se sincronicen en un punto del programa.
El problema de sincronización consiste en hacer esperar a todos los procesos y una vez que todos llegaron, pueden continuar.

# Barreras o *barriers*

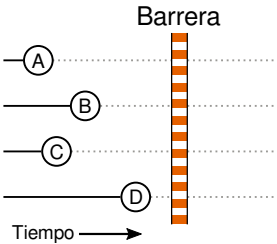
Existen problemas donde es necesario varios procesos se sincronicen en un punto del programa.
El problema de sincronización consiste en hacer esperar a todos los procesos y una vez que todos llegaron, pueden continuar.

Para estos casos, existe un mecanismo de sincronización distinto conocido como **barrera** o ***barrier***.

# Barreras o *barriers*

Existen problemas donde es necesario varios procesos se sincronicen en un punto del programa.
El problema de sincronización consiste en hacer esperar a todos los procesos y una vez que todos llegaron, pueden continuar.

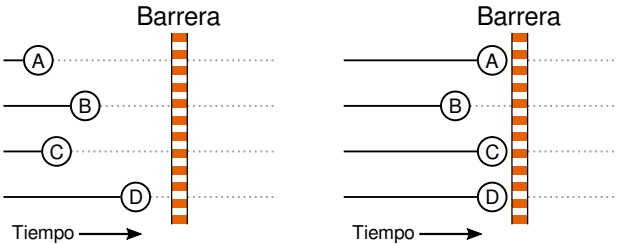
Para estos casos, existe un mecanismo de sincronización distinto conocido como **barrera** o ***barrier***.



# Barreras o *barriers*

Existen problemas donde es necesario varios procesos se sincronicen en un punto del programa.
El problema de sincronización consiste en hacer esperar a todos los procesos y una vez que todos llegaron, pueden continuar.

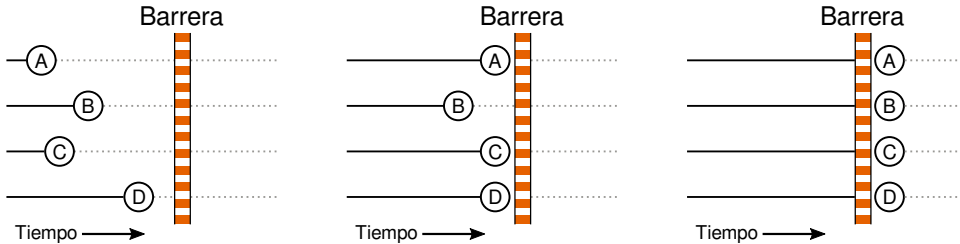
Para estos casos, existe un mecanismo de sincronización distinto conocido como **barrera** o ***barrier***.



# Barreras o *barriers*

Existen problemas donde es necesario varios procesos se sincronicen en un punto del programa.
El problema de sincronización consiste en hacer esperar a todos los procesos y una vez que todos llegaron, pueden continuar.

Para estos casos, existe un mecanismo de sincronización distinto conocido como **barrera** o ***barrier***.



# Problemas clásicos de sincronización

Existen varios problemas teóricos clásicos sobre sincronización de procesos:

- Productores-consumidores.
- Los lectores y escritores.
- La cena de los filósofos.
- El barbero dormilón.

A quienes estén interesados en profundizar sobre el tema,
les recomendamos leerlos de la bibliografía.

# Bibliografía


- Silberschatz, “Fundamentos de Sistemas Operativos”, 7ma Edición, 2006.
  - **Capítulo 6 - Sincronización de procesos**, páginas 171-186 y 161-165

- Tanenbaum, “Modern Operating Systems”, 4th Edition, 2015.
  - **Chapter 2 - Processes and Threads**
    - 2.3 Interprocess communication - Páginas 119-137 y 146-148
    - 2.4 Classical IPC problems - Páginas 167-172

# ¡Gracias!

Recuerden leer los comentarios adjuntos
en cada clase por aclaraciones.