

Multithreading

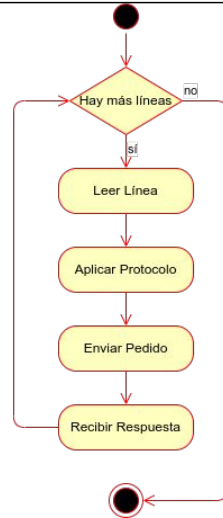
Taller de Programación I - FIUBA

Agenda

- Ejecución virtualmente en paralelo
- Acceso concurrente
- Operaciones atómicas
- Exclusión mutua de regiones críticas
- Monitores

¿Qué hicimos en otras materias?

- Un punto de entrada
- Ejecución secuencial (y estructuras de control)
- Terminamos ordenadamente



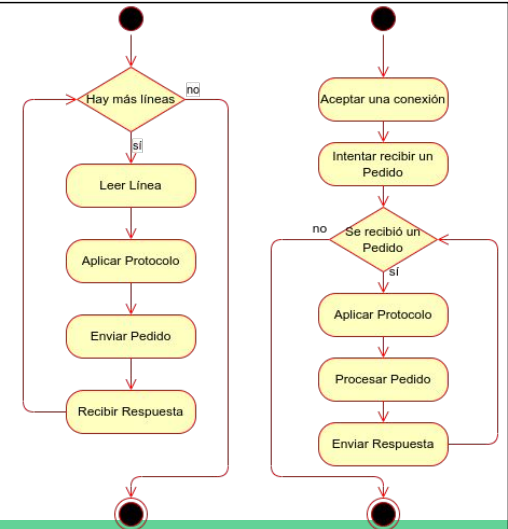
En otras materias hicimos programas secuenciales, que se parecen al cliente que programamos en el sprint 1

El programa empieza, ejecuta una lógica, con un par de estructuras de control, y termina

Nuestro código es una secuencia independiente de instrucciones

¿Qué hicimos en Taller?

- Lanzamos dos instancias de estos programas secuenciales
- Los comunicamos con un fin común



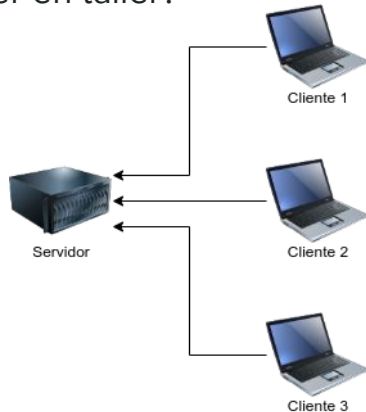
Lo que le agregamos hasta ahora en taller, es que lanzamos dos de esos programas secuenciales, y les agregamos comunicación para que interactúen

Entonces todo se complejizó: si teníamos un problema, no sabíamos si estaba en el server, en el client, en el protocolo...

Por qué? Porque ahora tenemos dos **secuencias independientes de ejecución**, dos **procesos**, ejecutando dos **programas** distintos...

Spoiler: ¿Qué vamos a hacer en taller?

- Más de un Cliente
- Un Server
- En el diagrama hay dos **programas**, pero cuatro **procesos**



Todos en la misma computadora

- Nosotros venimos programando como si tuviésemos un procesador “dedicado”
- Una computadora actual puede correr muchos procesos “al mismo tiempo”

Más adelante en la materia, como habrán imaginado, vamos a agregarle más clientes al trabajo práctico

En el ejemplo de la imagen tenemos 3 vs 1

Hay dos **programas**, pero cuatro **procesos**, el cliente tiene su código (su programa), el servidor tiene su código (su programa), pero para que “vivan” en cada computadora le tenemos que asignar recursos. Cuando les asignamos recursos creamos un **proceso** para cada uno.

Esos procesos los podemos correr en distintas computadoras o todos en la misma, y los estaríamos corriendo “en paralelo”

El caso en el que cada proceso corre en una computadora no nos interesa por ahora. Ahora lo que importa es el caso en el que todo corre en la misma.

Nosotros venimos programando como si tuviésemos un procesador “dedicado”. Escribimos una secuencia de instrucciones en cierto lenguaje, lo compilamos (traducimos nuestro código fuente a código máquina), y lo ejecutamos (lanzamos un proceso con ese código máquina). Ese proceso no libera al procesador por motus propio sino hasta que termina

Pero nuestras computadoras corren muchos procesos “al mismo tiempo”, y muchos de ellos se comunican entre sí como nuestro client-server

Por ejemplo, en este momento yo estoy corriendo un browser para mostrar esta presentación, mientras un proceso del escritorio le dice al browser cuál tecla estoy tocando para que el browser decida pasar a la siguiente diapositiva

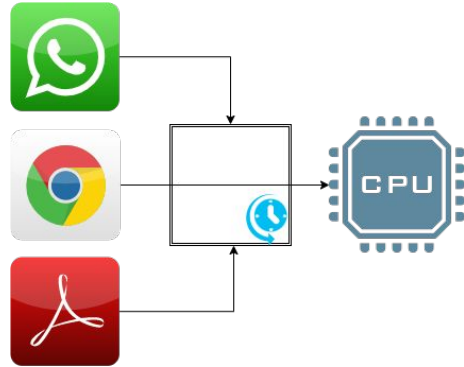
Para ver qué procesos están corriendo en sus compus pueden usar el comando htop

En la diapositiva se puede ver que esa computadora tiene cuatro cores, pero está corriendo varios procesos (muchos más que cuatro)

Pero si hay varios procesos corriendo y ninguno libera su core ¿Cómo hacen nuestras computadoras para correr tantos procesos con un número limitado de cores?

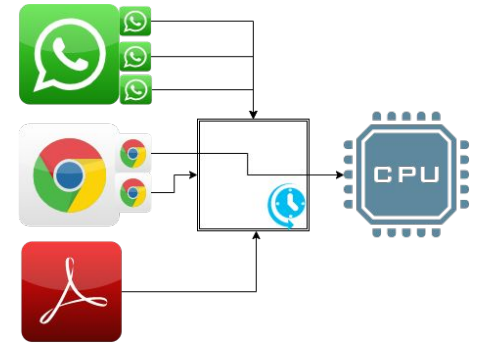
El Scheduler (Planificador)

- Componente del SO
- Decide qué proceso usa cada core en cada instante
- NO nos vamos a detener en su funcionamiento



Cada aplicación puede tener sus hilos

- Y el scheduler hace lo suyo
- Los hilos de cada aplicación corren **virtualmente** en paralelo
- Un **programa** es un ejecutable, podemos compararlo con el código
- Un **proceso** es una secuencia de ejecución que le da “vida” al programa
- Un **hilo** es una secuencia independiente de ejecución dentro de un proceso



El Scheduler es un componente del sistema operativo que decide qué proceso usa la CPU en cada instante.

No vamos a ver cómo funciona sino describir a alto nivel cómo interviene en lo que nos concierne hoy (referencia: esto se ve en Sistemas Operativos, y no queremos pisarles los temas)

La idea del scheduler es esta:

Un usuario puede haber lanzado un WhatsApp web, más un Chrome y un Adobe Reader.

El programador de cada una de estas aplicaciones, las escribió independientemente de las demás aplicaciones que estén corriendo

Y es deseable que una computadora pueda correr todas esas aplicaciones juntas “al mismo tiempo”

Entonces, si bien esas aplicaciones están corriendo “simultáneamente”, solamente uno lo va a estar usando físicamente.

El scheduler se encarga de decidir cuál es ese proceso, y cuando él lo decide, le quita el procesador a uno para dárselo al otro, dando la sensación de que están corriendo en paralelo

Eso es todo lo que vamos a decir del scheduler: es el componente que hace que los procesos corran virtualmente en paralelo

Gracias al scheduler, los hilos de nuestras aplicaciones van a correr también **virtualmente** en paralelo

Entonces suena parecido un proceso, un hilo y un programa, vamos a diferenciarlos a alto nivel:

Un programa es un ejecutable, podemos compararlo con el código

Un proceso es una secuencia de ejecución que le da “vida” al programa.

La clave para entender esa diferencia es que puedo tener varios procesos ejecutando un mismo programa

Y un hilo es una secuencia independiente de ejecución dentro de un proceso

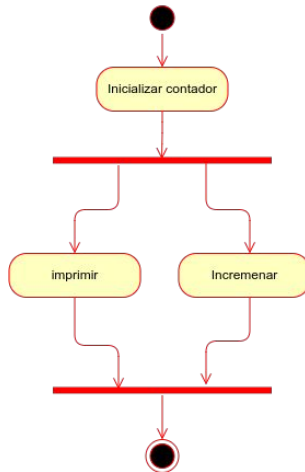
Y la clave acá es que cada proceso puede tener varios hilos. Hasta ahora en la facu venimos escribiendo programas con un solo hilo (el main)

Entonces lo que nos importa es que los hilos corren **virtualmente en paralelo**, y el funcionamiento del scheduler es “virtualmente anecdótico”.

¿Nos importa el scheduler? Sí, claro que nos importa, es la facultad de ingeniería esto. ¿Tiene que ser relevante a la hora de programar? No, en general es irrelevante para la lógica de nuestro programa.

Vamos a C++ (start & join)

```
1| #include <thread>
2| #include <iostream>
3|
4| int contador;
5| void incrementar() {
6|     ++contador;
7| }
8|
9| int main() {
10|     contador = 0;
11|     std::thread unHilo(incrementar);
12|     std::cout << "Hilo main!" << std::endl;
13|     unHilo.join();
14| }
```



Más hilos incrementadores!

```
1| int contador;
2| void incrementar() {
3|     ++contador;
4| }
5|
6| int main() {
7|     contador = 0;
8|     std::vector<std::thread> todosLosHilos;
9|     for (int i = 0; i < 10; ++i)
10|         todosLosHilos.emplace_back(incrementar);
11|     for (std::thread &hilo: todosLosHilos)
12|         hilo.join();
13|     std::cout << "Contador: " << contador << std::endl;
14| }
```

C++, desde su estándar C++11 introduce el objeto `std::thread` a la STL

Para lanzar un thread, basta con construir un objeto `std::thread` y pasarle algo "llamable".

Hasta este ejemplo, vamos a pasarle una función global, pero hay muchas más cosas "llamables" además de una función global

Notar que al principio del programa hay un solo hilo de ejecución (el main), luego ese flujo se bifurca porque se lanza un hilo. Y al terminar el hilo principal los otros hilos tienen que haber terminado (notar que solamente una flecha llega al punto de salida).

Mientras el hilo principal no llama a `join()` hay dos secuencias independientes de ejecución, que corren **virtualmente en paralelo**.

Cómo sabemos que terminaron? Con el método `join`.

En la línea 11, al crear un thread con una función como parámetro, estamos lanzando un hilo. A partir de ese momento estamos reservando un recurso, entonces lo tenemos que "destruir". Destruir un thread es esperar a que termine, y para eso sirve `join()`

Para indicar el punto en el que se lanza un thread en un diagrama de flujo, vamos usar esta notación. Cuando lanzamos un hilo, vamos a hacer un segmento horizontal al que entra una sola línea pero salen dos. Y cuando hacemos un `join()`, lo contrario (entran dos flujos, sale uno)

En este ejemplo vamos a lanzar 10 hilos incrementadores, y luego vamos a imprimir el valor del contador

Cuánto va a valer el contador al momento de imprimirlo?

No es determinístico!

Porque el incremento, por más que se haga en una sola línea de código C++, no es "atómico"

¿Qué pasa si compilamos la función?

```
1 | int contador;  
2 | void incrementar() {  
3 |     ++contador;  
4 | }  
5 |  
  
contador:  
    .zero    4  
incrementar():  
    push    rbp  
    mov     rbp, rsp  
    mov     eax, DWORD PTR contador[rip]  
    add     eax, 1  
    mov     DWORD PTR contador[rip], eax  
    nop  
    pop     rbp  
    ret
```

Este es el assembly generado por un gcc 11.2 para un x86-64.
Si quieren una versión interactiva pueden jugar con un link que les dejo en las notas
<https://godbolt.org/>

Lo más importante de este assembly son las tres líneas de assembly para el incremento (las celestes/turquesas)
Notar que hay una línea que lleva el número de la memoria a un registro de la CPU
Luego se hace el incremento en la CPU (ALU mediante)
Y por último se devuelve el valor modificado a la memoria

Eso quiere decir que una línea de un lenguaje de alto nivel no necesariamente se traduce en una línea de assembly o de código máquina (y la CPU no sabe nada de códigos de alto nivel, sabe de su código máquina)

Veamos dos incrementadores (caso feliz)

Hilo 1:

```
PC1 -->  mov     eax, DWORD PTR contador[rip]  
         add     eax, 1  
         mov     DWORD PTR contador[rip], eax    contador en memoria = 0
```

Hilo 2:

```
PC2 -->  mov     eax, DWORD PTR contador[rip]    contador en eax 1 = ???  
         add     eax, 1                          contador en eax 2 = ???  
         mov     DWORD PTR contador[rip], eax
```

En el ejemplo hemos lanzado dos hilos, y cada uno está empezando a ejecutar el incremento

Antes de que los hilos ejecuten ninguna línea, el contador en memoria está en 0, y los registros para cada hilo no están siquiera inicializados

Digamos que el scheduler le da la CPU al hilo 1

Veamos dos incrementadores (caso feliz)

Hilo 1:

```
mov    eax, DWORD PTR contador[rip]
PC1 --> add    eax, 1
mov    DWORD PTR contador[rip], eax
```

contador en memoria = 0

Hilo 2:

```
PC2 --> mov    eax, DWORD PTR contador[rip]
add     eax, 1
mov    DWORD PTR contador[rip], eax
```

contador en eax 1 = 0
contador en eax 2 = ???

Vemos que el program counter se movió a la siguiente línea, y el registro del hilo 1 tiene un 0
El otro hilo no se enteró que eso sucedió

Supongamos que el scheduler le vuelve a dar tiempo al hilo 1

Veamos dos incrementadores (caso feliz)

Hilo 1:

```
mov    eax, DWORD PTR contador[rip]
add     eax, 1
PC1 --> mov    DWORD PTR contador[rip], eax
```

contador en memoria = 0
contador en eax 1 = 1

Hilo 2:

```
PC2 --> mov    eax, DWORD PTR contador[rip]
add     eax, 1
mov    DWORD PTR contador[rip], eax
```

contador en eax 2 = ???

Vemos que el program counter se movió a la siguiente línea, pero ahora el registro del hilo 1 tiene un 1
El otro hilo sigue sin enterarse de nada

Supongamos que el scheduler le vuelve a dar tiempo al hilo 1

Veamos dos incrementadores (caso feliz)

Hilo 1:

```
mov    eax, DWORD PTR contador[rip]
add     eax, 1
mov     DWORD PTR contador[rip], eax
```

contador en memoria = 1

Hilo 2:

```
PC2 --> mov    eax, DWORD PTR contador[rip]
add     eax, 1
mov     DWORD PTR contador[rip], eax
```

contador en eax 1 = 1
contador en eax 2 = ???

El hilo 1 ya terminó, y dejó un 1 en la memoria

Ahora el scheduler le da tiempo al hilo 2...

Veamos dos incrementadores (caso feliz)

Hilo 1:

```
mov    eax, DWORD PTR contador[rip]
add     eax, 1
mov     DWORD PTR contador[rip], eax
```

contador en memoria = 1
contador en eax 1 = 1

Hilo 2:

```
PC2 --> mov    eax, DWORD PTR contador[rip]
add     eax, 1
mov     DWORD PTR contador[rip], eax
```

contador en eax 2 = 1

Se carga en memoria el 1

Veamos dos incrementadores (caso feliz)

Hilo 1:

```
mov    eax, DWORD PTR contador[rip]
add     eax, 1
mov     DWORD PTR contador[rip], eax
```

contador en memoria = 1

Hilo 2:

```
mov     eax, DWORD PTR contador[rip]
add     eax, 1
PC2 --> mov     DWORD PTR contador[rip], eax
```

contador en eax 1 = 1
contador en eax 2 = 2

Se incrementa el valor

Veamos dos incrementadores (caso feliz)

Hilo 1:

```
mov     eax, DWORD PTR contador[rip]
add     eax, 1
mov     DWORD PTR contador[rip], eax
```

contador en memoria = 2
contador en eax 1 = 1

Hilo 2:

```
mov     eax, DWORD PTR contador[rip]
add     eax, 1
mov     DWORD PTR contador[rip], eax
```

contador en eax 2 = 2

Se escribe en memoria

Entonces se ejecutaron los dos threads, virtualmente en paralelo pero físicamente en orden, y el valor final del contador es un 2
Ahora, qué pasa si el scheduler elige otro orden?

Veamos dos incrementadores (caso infeliz)

Hilo 1:

```
PC1 -->  mov    eax, DWORD PTR contador[rip]
          add    eax, 1
          mov    DWORD PTR contador[rip], eax
```

contador en memoria = 0

Hilo 2:

```
PC2 -->  mov    eax, DWORD PTR contador[rip]
          add    eax, 1
          mov    DWORD PTR contador[rip], eax
```

contador en eax 1 = ???
contador en eax 2 = ???

Empezamos de nuevo.

Digamos que el scheduler le da la CPU al hilo 1

Veamos dos incrementadores (caso infeliz)

Hilo 1:

```
          mov    eax, DWORD PTR contador[rip]
PC1 -->  add    eax, 1
          mov    DWORD PTR contador[rip], eax
```

contador en memoria = 0
contador en eax 1 = 0

Hilo 2:

```
PC2 -->  mov    eax, DWORD PTR contador[rip]
          add    eax, 1
          mov    DWORD PTR contador[rip], eax
```

contador en eax 2 = ???

Se cargó el valor de memoria en el registro del hilo 1.

Ahora el scheduler decide darle tiempo al hilo 2

Veamos dos incrementadores (caso infeliz)

Hilo 1:

```
mov    eax, DWORD PTR contador[rip]
PC1 --> add    eax, 1
mov    DWORD PTR contador[rip], eax
```

contador en memoria = 0

Hilo 2:

```
mov    eax, DWORD PTR contador[rip]
PC2 --> add    eax, 1
mov    DWORD PTR contador[rip], eax
```

contador en eax 1 = 0
contador en eax 2 = 0

Se cargó el valor de memoria en el registro del hilo 2

Y digamos que el scheduler le da tiempo al hilo 2 de nuevo

Veamos dos incrementadores (caso infeliz)

Hilo 1:

```
mov    eax, DWORD PTR contador[rip]
PC1 --> add    eax, 1
mov    DWORD PTR contador[rip], eax
```

contador en memoria = 0

Hilo 2:

```
mov    eax, DWORD PTR contador[rip]
add    eax, 1
PC2 --> mov    DWORD PTR contador[rip], eax
```

contador en eax 1 = 0
contador en eax 2 = 1

El registro del hilo 2 ahora tiene un 1

Y el scheduler le vuelve a dar tiempo al hilo 2

Veamos dos incrementadores (caso infeliz)

Hilo 1:

```
mov    eax, DWORD PTR contador[rip]
PC1 --> add    eax, 1
mov    DWORD PTR contador[rip], eax
```

contador en memoria = 1

Hilo 2:

```
mov    eax, DWORD PTR contador[rip]
add    eax, 1
mov    DWORD PTR contador[rip], eax
```

contador en eax 1 = 0
contador en eax 2 = 1

El hilo 2 terminó, y en memoria quedó un 1

Ahora el scheduler le da tiempo al hilo 1

Veamos dos incrementadores (caso infeliz)

Hilo 1:

```
mov    eax, DWORD PTR contador[rip]
add    eax, 1
PC1 --> mov    DWORD PTR contador[rip], eax
```

contador en memoria = 1
contador en eax 1 = 1

Hilo 2:

```
mov    eax, DWORD PTR contador[rip]
add    eax, 1
mov    DWORD PTR contador[rip], eax
```

contador en eax 2 = 1

El hilo 1 incrementa

Veamos dos incrementadores (caso infeliz)

Hilo 1:

```
mov    eax, DWORD PTR contador[rip]
add     eax, 1
mov     DWORD PTR contador[rip], eax
```

contador en memoria = 1

Hilo 2:

```
mov     eax, DWORD PTR contador[rip]
add     eax, 1
mov     DWORD PTR contador[rip], eax
```

contador en eax 1 = 1

contador en eax 2 = 1

Y a la hora de volcar el valor a memoria, termina escribiendo un 1

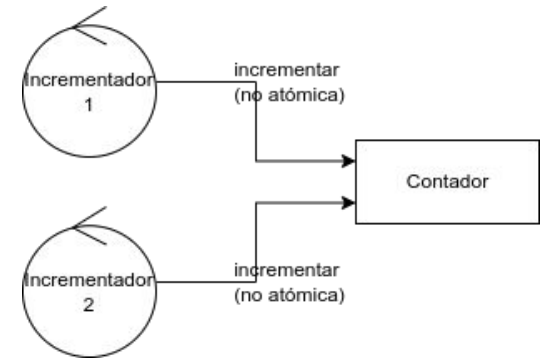
¿Qué pasó? Se dio lo que se conoce como **race condition**, y se generó un resultado que no esperábamos: se ejecutaron dos hilos que tenían que incrementar un contador inicializado en 0, y el contador no terminó con un 2

Podríamos seguir tirando decisiones del scheduler, y algunas van a dar bien y otras mal, a cada uno de estos entrelazamientos se les llama **interleavings**

Por qué pasó esto?

Acceso Concurrente

- Hay un recurso compartido
- Hay más de un thread accediendo al recurso
- Hay operaciones de escritura (al menos una)
- Hay operaciones no atómicas



La race condition se dio por estos 4 motivos combinados:

1. Hay un **recurso compartido** (el contador)
2. Tenemos más de un hilo accediendo a ese recurso
3. Esos hilos acceden al contador, y **al menos uno lo está escribiendo**: En este caso no se ve, los dos hilos están escribiendo, pero si uno lo escribiera y el otro lo leyera estaríamos en problemas igual (de nuevo, en este ejemplo simple no se ve, pero para recursos más complejos es evidente). Hay un ejemplo más adelante en la diapo
4. Las operaciones que se ejecutan sobre el contador **no son atómicas**. Vimos que el incremento no es solamente una línea de assembly sino que son 3

Tenemos que eliminar alguna de los cuatro motivos

1. Los hilos del programa no comparten nada. Por ejemplo: podría tener varios hilos, cada uno mirando una sección específica de un archivo
2. Si cada hilo escribiera en su propio contador, no habría race condition, porque no estaríamos mirando el mismo pedacito de memoria

1. Si todos los hilos estuvieran solamente **mirando/leyendo** el contador, no habría race condition. REPETIMOS, SI UN HILO LEE Y OTRO ESCRIBE SÍ QUE LA HAY. **Tengo que poner el mutex tanto en la escritura como en la lectura...**
2. Si ejecutamos **operaciones atómicas**, no se daría la race condition. Eso significa que solamente puedo hacer cosas que en assembly ocupan una línea? No, para eso vamos a introducir un mecanismo de sincronización que se llama **mutex**, que sirve para tener operaciones "virtualmente atómicas"

Mutex y Critical Sections

```
1| #include <mutex> // y los otros includes
2|
3| int contador;
4| std::mutex elMutex;
5|
6| void incrementar() {
7|     elMutex.lock();
8|     ++contador;
9|     elMutex.unlock();
10| }
11|
12| int main() {
13|     lanzarYJoinarHilos();
14| }
```

C++ nos provee un mecanismo de sincronización llamado "mutex". Para quienes conozcan semáforos, un mutex es un semáforo que solamente toma valor 0 o 1.

Y nos va a servir para delimitar regiones críticas, o critical sections

Vamos al ejemplo del contador

En el ejemplo del contador, el cacho de código que accede de forma no atómica al contador es el incremento.

A esos cachitos de código que nos gustaría que sean atómicos los tenemos que detectar a mano, y los vamos a llamar "Critical sections"

Cuando los detectemos, los vamos a delimitar con un mutex. Al arrancar la CS vamos a usar el método lock(), y al finalizar el método unlock()

Cada critical section es ahora una operación atómica respecto a las demás critical sections delimitadas por un mismo mutex

Y cuando decimos el mismo mutex es EL MISMO, notar que en el código son dos variables globales para que sean compartidas por todos los hilos. y están una "al lado de la otra" (quedate con eso en la memoria)

Mutex y Critical Sections

```
1| int contador;  
2| std::mutex elMutex;  
3|  
4| void incrementar() {  
5|     elMutex.lock();  
6|     ++contador;  
7|     elMutex.unlock();  
8| }  
9|  
10| void decrementar() {  
11|     elMutex.lock();  
12|     --contador;  
13|     elMutex.unlock();  
14| }
```

Para ejemplificar, podríamos agregar un nuevo método para decrementar

Notar que este nuevo método usa el **mismo mutex** para acceder al **mismo recurso**

Para pensar: qué pasa si tuviéramos dos contadores en vez de uno?

RE: Tendríamos que tener un mutex por contador, porque dos recursos -> dos mutex

Esto de un mutex por recurso es importante porque más adelante en la clase vamos a ver una técnica para abstraer la lógica de sincronización y que se mezcle lo menos posible con la lógica del negocio (pensar en protocolo vs. modelo)

Una forma de ver al Mutex (una llave)

- El contador está en una habitación
- La calculadora está afuera, y tenemos que usarla
- El algoritmo para actualizarlo:
 - Entrar a ver el valor actual
 - Calcular el nuevo valor
 - Entrar a cambiar el valor
- ¿Y si hay varias personas haciendo eso? Estaría bueno tener la llave

Una manera didáctica para entender los mutex es pensarlos como una llave

Siguiendo el ejemplo del contador, imaginemos que hay una habitación donde está el contador
<leer "algoritmo">

Si hay varias personas siguiendo el algoritmo, podríamos tener la misma race condition (ahora sin assembly, sino imaginándonos personas)

Ahora, si le metemos una llave al escenario, y cada uno toma la llave desde que decide tocar el contador hasta que está actualizado, dejamos de tener race condition. Eso es el mutex

Ahora le agregamos RAII

```
1| int contador;          1| int contador;
2| std::mutex elMutex;    2| std::mutex elMutex;
3|                          3|
4| void incrementar() {    4| void incrementar() {
5|     elMutex.lock();     5|     std::lock_guard<std::mutex> lock(elMutex);
6|     ++contador;         6|     ++contador;
7|     elMutex.unlock();   7| }
8| }                       8|
```

La STL incluye una clase RAII para manejar los mutex, que se llama `lock_guard`, y sirve para hacer lo mismo, pero sin “olvidarnos” del unlock

Luego de haber venido a la clase de error handling, sabemos que “olvidarnos” puede significar literalmente “olvidarnos”, o que haya alguna excepción que evite que se llame al unlock, es por eso que estos locks RAII se vuelven muy importantes

¿Y qué nos quedó?

```
1| int contador;          // El recurso en una variable global
2| std::mutex elMutex;    // El mutex en una variable global
3|
4| void incrementar() {    // Una operación sobre el
5|     std::lock_guard<std::mutex> l(elMutex); // recurso compartido en
6|     ++contador;         // una función global
7| }
8|
9| void decrementar() {    // Otra operación sobre el
10|    std::lock_guard<std::mutex> l(elMutex); // recurso compartido en
11|    --contador;          // una función global
12| }
13|
```

<leer la diapo>

Esto pide a gritos un encapsulamiento!! Todo global **no es una opción**

Creemos una clase

```
1| class ContadorProtegido {  
2|     int contador;  
3|     std::mutex m;  
4| public:  
5|     ContadorProtegido() : contador(0) {}  
6|     void incrementar() {  
7|         std::lock_guard<std::mutex> l(m);  
8|         ++contador;  
9|     }  
10|    void decrementar() {  
11|        std::lock_guard<std::mutex> l(m);  
12|        --contador;  
13|    }  
14| };
```

Esta clase tiene el recurso como un atributo

El mutex como un atributo

Cada Critical Section es un método público, y NO HAY OTROS MÉTODOS PÚBLICOS

A este tipo de objetos que protegen al recurso, y cuya API son las Critical Sections, les decimos Monitores

Ejercicio: Contador de Caracteres Multi-Thread

- Queremos implementar un contador de caracteres sobre un archivo muy grande
- Como aprendimos multithreading en taller, creemos que nos puede dar un beneficio de performance utilizar N threads, entonces hacemos eso
- Es deseable tener una estructura de datos que nos sirva para tener todos los contadores

Manos a la obra!

```
1| int main() {  
2|     // Abrir el archivo...  
3|     MapaDeContadores contadores;  
4|     std::vector<std::thread> hilos;  
5|     for (int i = 0; i < N; ++i) {  
6|         int inicio = i * tamaño / N;  
7|         int fin = inicio + tamaño / N;  
8|         hilos.emplace_back(contar, archivo, inicio, fin, contadores);  
9|     }  
10|    for (std::thread &hilo : hilos) {  
11|        hilo.join();  
12|    }  
13|    contadores.mostrar();  
14| }
```

Vamos por partes, y usando una técnica top-down

Entonces primero programamos (sobre-simplificando)

Abrimos el archivo

Creamos una estructura de datos "contadores"

Lanzamos estos hilos que ejecutan la función contar sobre un pedazo del archivo

Y luego esperamos los threads

Una vez que todos terminaron, mostramos los resultados

Para acelerar el ejercicio, digamos que la función contar toma el archivo, lo lee desde la posición de inicio hasta la de fin, y en ese punto llama a la siguiente función

Dentro de la función contar se llama a esto...

```
1| void agregarCaracterAlMapa(char c, MapaDeContadores &contadores) {  
2|     if (contadores.contiene(c)) {  
3|         int cuentaParcial = contadores.obtenerCuentaParcial(c);  
4|         contadores.establecerCuentaParcial(c, cuentaParcial + 1);  
5|     } else {  
6|         contadores.establecerCuentaParcial(1);  
7|     }  
8| }
```

Cómo programaríamos esta función?

Tal vez la primera idea sería hacer algo así:

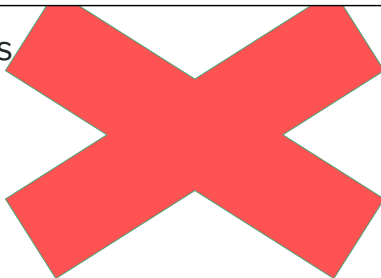
<ver diapo>

Y la clase MapaDeContadores...

```
1| class MapaDeContadores {
2|     std::map<char, int> internos;
3|     std::mutex m;
4| public:
5|     int obtenerCuentaParcial(char c) {
6|         std::lock_guard<std::mutex> l(m);
7|         return internos[c];
8|     }
9|     void establecerCuentaParcial(char c, int n) {
10|         std::lock_guard<std::mutex> l(m);
11|         internos[c] = n;
12|     }
13|     bool contiene(char c) {
14|         std::lock_guard<std::mutex> l(m);
15|         return internos.find(c) != internos.end();
16|     }
17| };
```

La clase MapaDeContadores, entonces, quedaría así.
Protegimos el acceso al recurso??

Y la clase MapaDeContadores



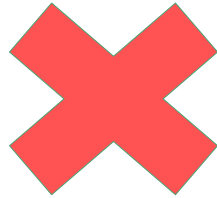
```
1| class MapaDeContadores {
2|     std::map<char, int> internos;
3|     std::mutex m;
4| public:
5|     int obtenerCuentaParcial(char c) {
6|         std::lock_guard<std::mutex> l(m);
7|         return internos[c];
8|     }
9|     void establecerCuentaParcial(char c, int n) {
10|         std::lock_guard<std::mutex> l(m);
11|         internos[c] = n;
12|     }
13|     bool contiene(char c) {
14|         std::lock_guard<std::mutex> l(m);
15|         return internos.find(c) != internos.end();
16|     }
17| };
```

NO

Los métodos de esta clase son mutuamente excluyentes, pero la API es insegura
En otras palabras: hicimos los métodos de map mutuamente excluyentes, pero esas
no son las critical sections de nuestro programa

Volvamos al uso de nuestra clase

```
1| void agregarCaracterAlMapa(char c, MapaDeContadores &contadores) {  
2|     if (contadores.contiene(c)) {  
3|         int cuentaParcial = contadores.obtenerCuentaParcial(c);  
4|         contadores.establecerCuentaParcial(c, cuentaParcial + 1);  
5|     } else {  
6|         contadores.establecerCuentaParcial(1);  
7|     }  
8| }
```



Cuál es el problema? Esta función está siendo llamada **repetidamente** desde N hilos. Qué pasa si cuando un hilo está en la línea 6, otro hilo ya agregó ese caracter? <hacer un seguimiento>

Lo repetimos: si bien los métodos de esta clase MapaDeContadores son mutuamente excluyentes, construimos una API insegura.

Cuál es efectivamente la Critical Section de este programa? Todo el if/else!

Ahora que sabemos la CS, hagámoslo bien

```
1| void agregarCaracterAlMapa(char c, MapaDeContadores &contadores) {  
2|     contadores.incrementarCuenta(c);  
3| }
```

Ahora que sabemos cuál es la CS, sabemos que tiene que ser un método de nuestra clase Monitor (MapaDeContadores).

Y la clase MapaDeContadores...

```
1| class MapaDeContadores {
2|     std::map<char, int> internos;
3|     std::mutex m;
4|
5| public:
6|     int incrementarCuenta(char c) {
7|         std::lock_guard<std::mutex> l(m);
8|         if (internos.find(c) != internos.end())
9|             internos[c] += 1;
10|        else
11|            internos[c] = 1;
12|        }
13|    };
```

Y ahora la clase MapaDeContadores quedaría así.

Notar que ahora el único método de nuestra clase monitor contiene la critical section que detectamos

Esta es la máxima para construir Monitores, y en los TPs TIENEN que implementar el acceso a recursos compartidos utilizando Monitores

Monitores (resumen)

- Al **recurso** lo pensamos (y codificamos) como si no hubiera problemas de concurrencia: el recurso NO CONOCE AL MUTEX
- Creamos una clase “Monitor” con el mutex y el recurso como atributos
- Agregamos al monitor un método por cada Critical Section que hayamos detectado
- LISTO! Hay una clase que tiene la lógica para sincronizar y otra con la lógica de negocio, no se tocan, bajo acoplamiento, alta cohesión

Entonces, en resumen...

Error Handling

- Cada hilo su stack
- Cada hilo su try-catch

```
1| void func_2() { throw -1; }
2| void func_1() { func_2(); }
3|
4| int main() {
5|     try {
6|         func_1();
7|     } catch (int i) {
8|         ...
9|     }
10| return 0;
11| }
```

```
1| void func_2() { throw -1; }
2| void func_1() { func_2(); }
3|
4| int main() {
5|     try {
6|         std::thread t(func_1);
7|         t.join();
8|     } catch (int i) {
9|         ...
10|     }
11| return 0;
12| }
```

Miren la diferencia entre estos dos códigos

El de la izquierda llama a la función 1, que llama a la función 2, y esta última lanza un entero (debería lanzar una excepción, pero quería que entre en una diapo)

Y el de la derecha hace lo mismo, pero la función 1 la llama en otro thread (por supuesto, luego joina ese thread)

Qué pasa? En el primero la excepción es lanzada, no se catchea dentro de la función 1, sube en el stack, no se catchea en la función 2, sube en el stack, se catchea en el main, el programa maneja la exception, todos felices...

En el segundo, la excepción se lanza en la función 2, no se catchea, sube en el stack, no se catchea en la función 1, Y NO HAY MAS STACK! Este código va a terminar saliendo con una señal, y no es algo deseable.

Error Handling

```
1| void func_2() { throw -1; }
2| void func_1() { // Es el punto de entrada de un hilo
3|     try{
4|         func_2();
5|     } catch (int i) {
6|         ...
7|     }
8| }
9| int main() {
10|     std::thread t(func_1);
11|     t.join();
12|     ...
13| }
```

Para evitar esto, cada función (llamable) que le pasemos como punto de entrada a los hilos que lancemos tiene que tener su propio try-catch

En la clase pasada dijimos que en principio el único lugar donde poner un catch en C++ era en el main, y en los lugares donde realmente haga falta manejar un error. Bueno, este es el otro caso donde hay que poner un catch: en la función de entrada de cada hilo que lancemos

Problemas

- Context switch
- Deadlocks
- Starvation
- Contención

Context Switch

- Lanzar muchos hilos no siempre es lo más performante
- Cambiar qué hilo está usando el procesador tiene un costo
- Usar varios hilos tiene que estar justificado

Imaginemos que tenemos que realizar una tarea con muchos cálculos, con un algoritmo iterativo (¿cursaron numérico?)

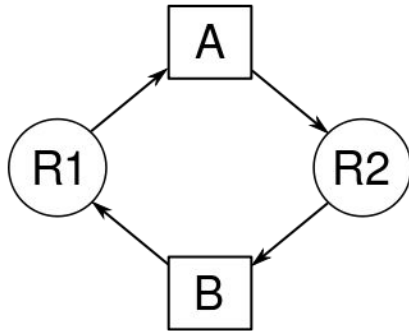
Vamos a necesitar el resultado de la iteración "N" para calcular la iteración "N+1", son algoritmos de naturaleza secuencial.

Si tenemos muchos hilos calculando este tipo de cosas, cada hilo va necesitar el resultado del otro para arrancar el cálculo en sí

¿Para qué usamos muchos hilos en ese caso?

Ahora, si tenemos que multiplicar matrices, hay algoritmos para calcular distintas partes del resultado: en este caso que podemos dividir en subtareas y realmente hay un paralelismo, probablemente veamos una mejora en la performance

Deadlock, o abrazo mortal



Tenemos dos (o más) hilos que necesitan dos recursos para ejecutar una acción...

Termo y mate
Llave y calculadora
etc...

Depende del orden en el que obtenga los recursos

Starvation / Inanición

- Es más difícil de detectar, no vamos a hacer mucho hincapié en esto
- Hay algún hilo que nunca termina ejecutando sus operaciones

Contención

- Las Critical Sections están mal delimitadas, el programa es thread-safe, pero el mutex está tomado “demasiado tiempo”
- El Monitor que vimos no es la técnica de programación concurrente con menos contención, pero es la más fácil de entender y estamos aprendiendo
- Pensar el MapaDeContadores en un escenario client-server, recibir las “requests” por socket, es parte de la critical section? NO