COMPUTACIÓN DE ALTA PERFORMANCE

Curso 2023

Sergio Nesmachnow (sergion@fing.edu.uy)

Centro de Cálculo

TEMA 6
PROGRAMACIÓN MULTITHREADING

CONTENIDO

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

1. ¿Qué es un hilo?
2. Modelos de planificación
3. POSIX threads
4. Pool de threads
5. Reentrancia vs thread-safe
6. Patrones paralelos de control
7. Paralelismo automático
8. Optimizando C
9. Arquitecturas paralelas

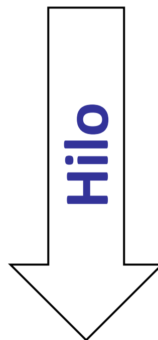
6.1: ¿QUÉ ES UN HILO?

¿QUÉ ES UN HILO?

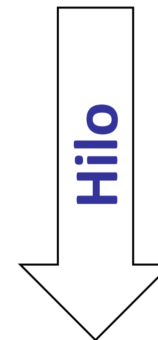
- Un thread (proceso ligero) es una unidad básica de uso de la CPU.
- Una proceso está formado por uno o varios threads, sin embargo un thread puede pertenecer solamente a único proceso.
- A nivel de SO cada thread tendrá su propio:
  - TID (Thread IDentifier).
  - Conjunto de Registros, Program Counter, Stack.
  - Prioridad.
  - Máscara de señales.
  - Datos privados.
- Los thread de un mismo proceso compartirán:
  - Espacio de memoria virtual (tablas de páginas).
  - Archivos abiertos, pipes, sockets, etc.

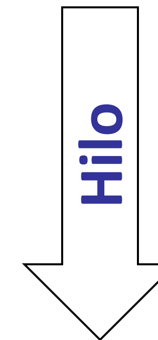
¿QUÉ ES UN HILO?

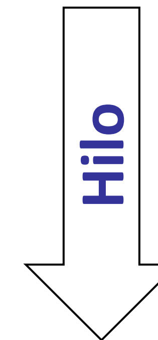
Proceso monohilado



Proceso multihilado







¿QUÉ ES UN HILO?

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Entradas en la tabla de procesos de un proceso que utiliza 4 procesadores mediante la primitiva `fork()`:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
4478	brunolop	25	0	3559m	1.5g	6516	R	100.0	2.2	2996:27	caffa3d.MBRi.mp
4479	brunolop	25	0	3559m	1.5g	7844	R	100.0	2.2	2996:14	caffa3d.MBRi.mp
4480	brunolop	25	0	3559m	1.5g	8468	R	100.0	2.2	2997:04	caffa3d.MBRi.mp
4482	brunolop	25	0	3559m	1.5g	7464	R	100.0	2.2	2997:37	caffa3d.MBRi.mp

- Entrada en la tabla de procesos de un proceso que utiliza 4 procesadores mediante multihilado:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
14397	aguti	25	0	1819m	1.5g	9.9m	R	400.1	2.1	1350:53	wrf.exe

¿QUÉ ES UN HILO?

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Beneficios:
  - Es más rápido un cambio de contexto entre threads de un mismo proceso que entre dos procesos diferentes.
  - Es más "liviano" para el sistema operativo crear un thread que crear un proceso nuevo.
  - Los threads de un proceso comparten toda la memoria y los recursos que utilizan.
  - Es más fácil utilizar mecanismos de sincronización y comunicación.
- Desventajas:
  - Programación más difícil. ¡Todo está compartido!
  - Poca tolerancia a fallos. Si alguno de los threads provoca un fallo, todo el proceso es detenido.

6.2: MODELOS DE PLANIFICACIÓN

MODELOS DE THREADS

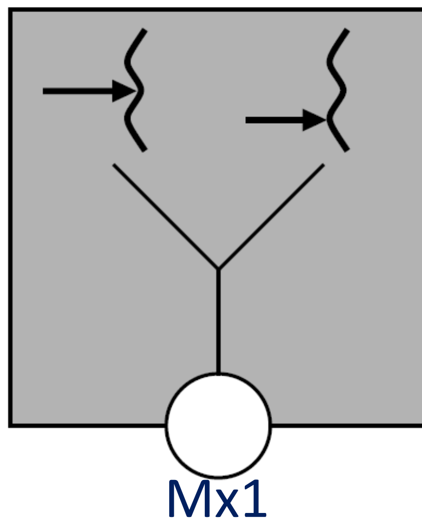
UNIVERSIDAD DE LA REPÚBLICA URUGUAY

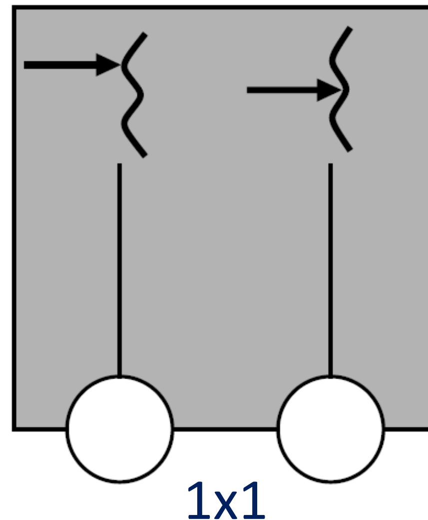
- Hilos a nivel de usuario (user threads)
  - Son implementados en una biblioteca en espacio de usuario
  - Permite para crear, planificar y administrar threads sin soporte del SO
  - El SO solo reconoce un hilo de ejecución en el proceso
- Hilos a nivel del núcleo (kernel threads)
  - El SO soporta la creación, planificación y administración de los threads
  - Reconoce tantos hilos de ejecución como threads se hayan creado

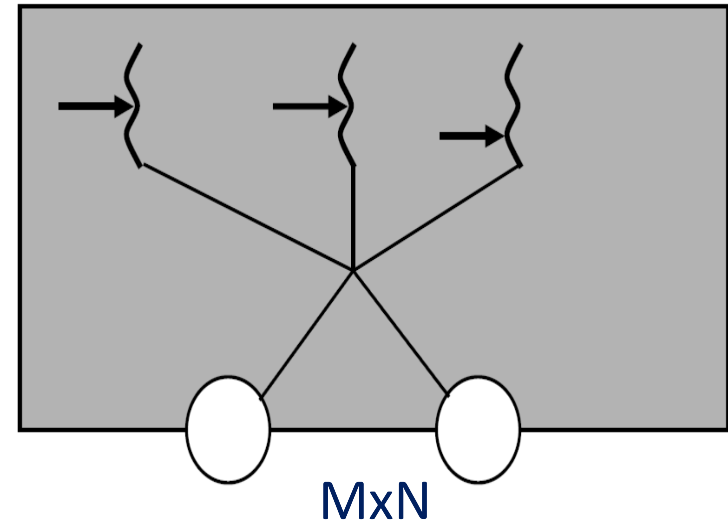
MODELOS DE PLANIFICACIÓN

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- La mayoría de los sistemas proveen threads tanto a nivel de usuario como de sistema operativo. Surgen varios modelos:
  - Mx1 (Many-To-One): varios threads de usuario a un único thread de núcleo.
  - 1x1 (One-to-One): cada threads de usuario se corresponde con un thread a nivel del núcleo.
  - MxN (Many-To-Many): varios threads a nivel de usuario (M) se corresponde con varios threads a nivel del núcleo (N), con M > N.


Mx1


1x1


MxN

LINUX THREADS

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

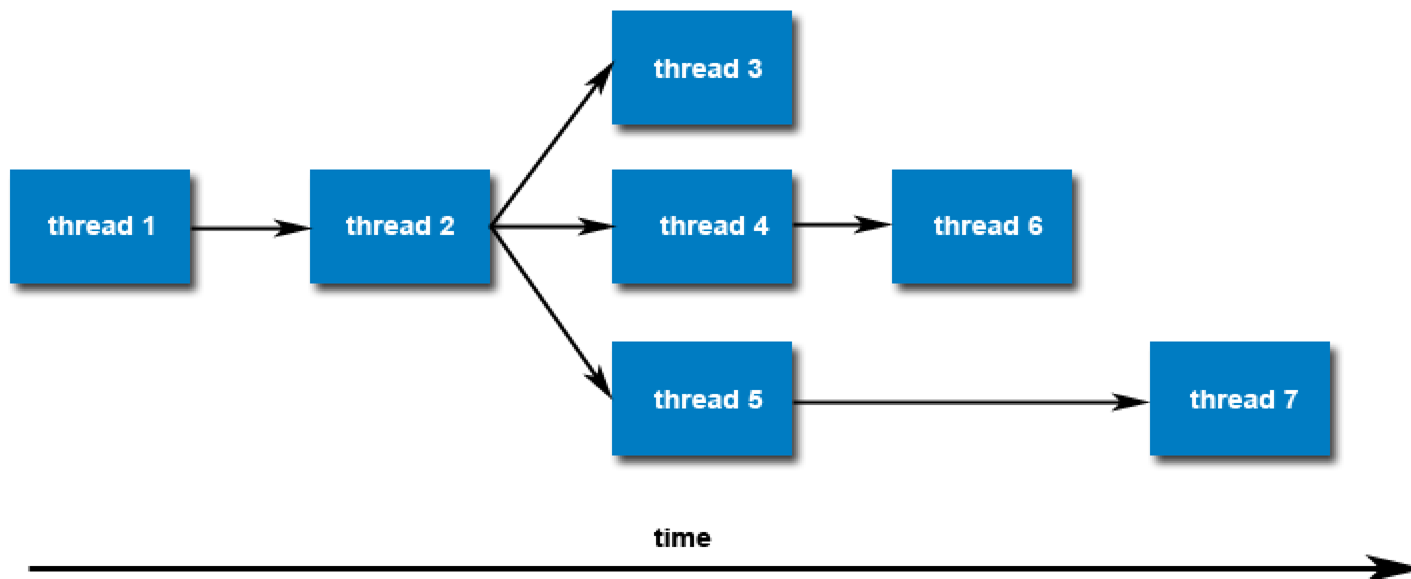
- Native POSIX Thread Library (NPTL) es la implementación por defecto de hilos en Linux desde la versión 2.6 del núcleo.
- Utiliza un modelo de planificación 1x1.
- Implementa la sincronización usando *Fast Userspace muTEX* (FUTEX).
- Un FUTEX opera casi completamente en espacio de usuario utilizando operaciones atómicas.
- Solo realiza system calls para dormir un hilo, despertar un hilo, o cuando se produce una condición de carrera.

6.3: POSIX THREADS

CREACIÓN DE THREADS

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Función de creación: `pthread_create()`
- Recibe un puntero a la función donde comenzará a ejecutar el hilo creado.
- Esta función debe tener un cabezal predefinido.
- La creación de threads genera dentro del proceso una estructura de árbol.



CREACIÓN DE THREADS

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

`pthread_create`

- Argumentos:
  - `pthread_t  *`

    TID del thread creado.
  - `const pthread_attr_t *`

    Parámetro de inicialización.
  - `void * (* función_thread) (void *)`

    Puntero a función donde comenzará a ejecutar el thread creado.
  - `void *`

    Parámetro de entrada para el puntero a función.
- Retorno:
  - `int`

    Valor de retorno (0 ok, != 0 error).

CREACIÓN DE THREADS

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Si el parámetro `pthread_attr_t` es nulo, el thread será creado con los valores por defecto:

Atributo	Valor por defecto	Efecto
Scope	PTHREAD_SCOPE_SYSTEM	El thread compite con el resto de los procesos del sistema
Detachstate	PTHREAD_CREATE_JOINABLE	El valor de retorno del thread es preservado
Stack size	1 MB	Tamaño del stack
Priority		Es heredada de la prioridad del thread padre

CREACIÓN DE THREADS

- Prototipo de la función debe ser el siguiente:

```
void* func_thread (void*)
```

- El parámetro de tipo `void*` permite pasar cualquier tipo de datos a la función.

```
struct persona {
  struct fecha     nacimiento;
  int              sexo;
};
struct persona p;

pthread_create (…., func_thread, (void *) p)
```

TERMINACIÓN DE THREADS

- Función de destrucción: `pthread_exit`.

- Típicamente es ejecutada cuando un thread finaliza su ejecución y no tiene nada más para realizar.
- Se puede asignar un status a la terminación para que algún otro threads (usando `pthread_join`) obtenga un valor de la ejecución.
- La función no elimina ningún recurso asignado al proceso que haya sido pedido por el thread.
- Si el thread `main` finaliza con un `pthread_exit`, todos los threads que hayan sido creados permanecerán activos.

PEQUEÑO EJEMPLO HILADO

- La función pthread_join permite a un thread esperar por la finalización de otro thread y obtener su estado de finalización.
- La función pthread_join es bloqueante sobre el thread que la invoca.
- No es posible esperar por varios threads como la función wait de C.

```
void* getNextPicture(void *arg){
  buff = loadFile(arg);
  pthread_exit(buff);
}

int main() {
  pthread_t pic_t;
  buff = loadFile((void*)argv[1]);
  for (i = 2; i < argc; i++) {
    pthread_create(&pic_thread, NULL, getNextPicture, ...);
    display(buff);
    pthread_join(pic_thread,(void**) &buff);
  }
}
```

¿Y LA SECCIÓN CRÍTICA?

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Una sección de código desde donde se accede a un recurso compartido que no debe ser accedido por más de un hilo de ejecución.
- En programación concurrente los hilos de ejecución generalmente comparten datos.
- Se debe sincronizar el acceso para garantizar su integridad.
- POSIX brinda mecanismos de sincronización a través de funciones y de variables de sincronización:
  - Mutex
  - Variables de condición
  - Reader/Writer locks
  - Barriers

MUTEX

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Mantiene MUTua EXclusión (MUTEX) en el acceso a secciones críticas entre los distintos hilos de un proceso.
- Garantiza que solamente un thread ejecutará la sección crítica.
- Cuando un thread está en una sección crítica, los demás threads que quieran acceder serán bloqueados.
- Una vez que este thread sale de la sección crítica, se despierta a otro thread que esté bloqueado para que pueda entrar.
- POSIX no garantiza un orden.

MUTEX

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- La estructura `pthread_mutex_t` permite definir variables de sincronización de mutua exclusión.
- Las variables mutex deben ser inicializadas y destruidas a través de las funciones:

```
pthread_mutex_init(pthread_mutex_t*, pthread_mutexattr_t*)
pthread_mutex_destroy(pthread_mutex_t*)
```

- El tipo de dato `pthread_mutexattr_t` permite inicializar el mutex con diferentes atributos.
- Por ejemplo:

```
pthread_mutex_t mutex1;
pthread_mutex_init(&mutex1,null);
…
pthread_mutex_destroy(&mutex1);
```

MUTEX

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Para «ganar» un mutex se debe obtener un lock sobre el mismo. Existe una operación lock bloqueante y otra no bloqueante.

- Función bloqueante: bloquea al thread que invoca la función hasta que este obtenga el lock.

```
int pthread_mutex_lock(pthread_mutex_t*);
```

- Función no bloqueante: el thread que ejecuta esta operación no quedará bloqueado en caso que el lock esté asignado a otro thread.

```
int pthread_trymutex_lock(pthread_mutex_t*);
```

- Para liberar un mutex ganado se tiene la operación:

```
int pthread_mutex_unlock(pthread_mutex_t*);
```

MUTEX

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Un ejemplo clásico de la programación concurrente es el problema de los lectores y escritores.
- Un conjunto de threads realiza operaciones de lectura en su sección crítica, mientras otro conjunto realiza operaciones de escritura.
- Si la cantidad de escrituras es significativamente menor que las lecturas, este problema se resuelve a través de la utilización de dos variables de sincronización.

```
Escritores

Mutex_lock(mtxEscr)
Sección crítica
Mutex_unlock(mtxEscr)
```

```
Lectores
Mutex_lock(mtxLec)
 lectores++
 if (lectores == 1)
    Mutex_lock(mtxEscr)
Mutex_unlock(mtxLec)
Sección crítica
Mutes_lock(mtxLec);
  lectores--
  if (lectores == 0)
    Mutex_unlock(mtxEscr)
Mutex_unlock(mtxLec)
```

¿BUSY WAITING ES MALA PRÁCTICA?

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- ¿Siempre resulta más eficiente bloquear un thread en espera?
- Recordar que bloquear un thread implica: entrar en modo kernel, bloquear el thread en una correspondiente cola, ejecutar el planificador del sistema, …
- Para secciones críticas largas es mejor bloquear el hilo.
- Para secciones críticas cortas es mejor hacer busy waiting.
- Los spinlocks son idénticos a los mutex salvo porque en lugar de bloquearse reintentan hasta obtener el lock (busy-waiting).

SPINLOCK

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- POSIX provee una implementación de spinlocks.

```
int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int psh);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

- Al invocar la operación `pthread_spin_lock`
  - el hilo de ejecución adquirirá el spinlock en caso de que se encuentre disponible
  - en caso contrario el hilo entrará en bucle hasta obtenerlo (busy waiting).
- La mayoría de las bibliotecas diseñadas para HPC utilizan spinlocks puros o híbridos (p.ej. Intel MPI, OpenMPI, etc.)

VARIABLES DE CONDICIÓN

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Las variables de condición permiten bloquear a un proceso hasta que cierta condición sea verdadera.
- Siempre deben ser usadas junto con un mutex asociado.
- La condición es controlada bajo la protección del mutex.
- Cuando la condición es falsa, el thread es bloqueado y libera el lock del mutex.
- Cuando otro thread cambia algún componente de la condición, este puede enviar un signal a threads que estén esperando.
- Los threads serán despertados, obtendrán el lock y evaluarán nuevamente la condición para determinar sus próximas acciones.

VARIABLES DE CONDICIÓN

- Funciones:

```
pthread_cond_init(pthread_cond_t*, pthread_condattr_t*)
pthread_cond_destroy(pthread_cond_t*)
pthread_cond_wait(pthread_cond_t* cv, pthread_mutex_t* mtx)
pthread_cond_signal(pthread_cond_t * cv)
pthread_cond_broadcast(pthread_cond_t * cv)
```

- Para ejecutar `wait` el thread debe tener ganado el mutex asociado
- `wait` bloquea el thread hasta que otro thread ejecute `signal` sobre la variable de condición.
- El mutex `mtx` es liberado al invocar `wait`.
- Al invocar `signal` un thread es despertado y queda en espera para la obtención del mutex `mtx`.

VARIABLES DE CONDICIÓN

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

```
Thread 1

pthread_mutex_t   m;
pthread_cond_t    c;
...
pthread_mutex_lock(&m);
while (!operado(dato_compartido))
   pthread_cond_wait(&c,&m);
trabajar(dato_compartido);
pthread_mutex_unlock(&m);
```

```
Thread 2

...
pthread_mutex_lock(&m);
operar(dato_compartido);
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
...
```

- Al invocar a `pthread_cond_wait` se libera la variable de mutua exclusión asociada.
- Cuando un thread es despertado por `pthread_cond_signal` se pone en espera para obtener la variable de mutua exclusión.
- `pthread_cond_broadcast` despierta y pone en espera a todos los threads que están en una variable de condición.

VARIABLES DE CONDICIÓN

- Volviendo a los lectores-escritores. Se quiere dar prioridad a los escritores. La estrategia es que cuando llegue un escritor “avise” de forma que los próximos lectores sean bloqueados.

```
Escritores

1. Mutex_lock(mtxEscr)
2. Sección crítica
3. Mutex_unlock(mtxEscr)
```

```
Lectores

1. Mutex_lock(mtxLec)
2.  lectores++
3.  if (lectores == 1)
4.      Mutex_lock(mtxEscr)
5. Mutex_unlock(mtxLec)
6. Sección crítica
7. Mutes_lock(mtxLec);
8.   lectores--
9.   if (lectores == 0)
10.     Mutex_unlock(mtxEscr)
11.Mutex_unlock(mtxLec)
```

VARIABLES DE CONDICIÓN

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Volviendo a los lectores-escritores. Se quiere dar prioridad a los escritores. La estrategia es que cuando llegue un escritor “avise” de forma que los próximos lectores sean bloqueados.

```
Lectores
Mutex_lock(mtxEscrEspera)
While (Escritores != 0)
   Esperar(cond,mtxEscrEspera)
Mutex_lock(mtxLec)
 lectores++
 if (lectores == 1)
    Mutex_lock(mtxEscr)
Mutex_unlock(mtxLec)
Mutex_unlock(mtxEscrEspera)
Sección crítica
Mutes_lock(mtxLec);
  lectores--
  if (lectores == 0)
    Mutex_unlock(mtxEscr)
Mutex_unlock(mtxLec)
```

```
Escritores
Mutex_lock(mtxEscrEspera)
Escritores++
Mutex_unlock(mtxEscrEspera)

Mutex_lock(mtxEscr)
Sección crítica
Mutex_unlock(mtxEscr)

Mutex_lock(mtxEscrEspera)
Escritores--
Signal_broadcast(cond)
Mutex_unlock(mtxEscrEspera)
```

READ-WRITE LOCKS

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Funciones:

1. `pthread_rwlock_init(pthread_rwlock_t*, pthread_rwlockattr_t*)`
2. `pthread_rwlock_destroy(pthread_rwlock_t*)`
3. `pthread_rwlock_rdlock (pthread_rwlock_t * rw)`
4. `pthread_rwlock_tryrdlock (pthread_rwlock_t * rw)`
5. `pthread_rwlock_wrlock (pthread_rwlock_t * rw)`
6. `pthread_rwlock_trywrlock (pthread_rwlock_t * rw)`
7. `pthread_rwlock_unlock(pthread_rwlock_t *)`

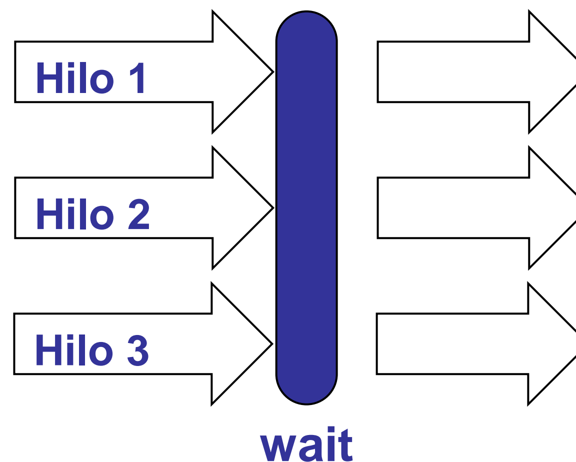
READ-WRITE LOCKS

- Un hilo puede tener más de un lock de lectura (lock recursivo).
- POSIX no define claramente si los escritores o lectores tienen prioridad por lo que se tendrá que consultar la implementación utilizada.
- En la implementación actual de Linux los lectores tienen prioridad sobre los escritores.
- Puede modificarse con la función `pthread_rwlockattr_setkind_np`
  - `PTHREAD_RWLOCK_PREFER_READER_NP`
  - `PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP`

BARRIER

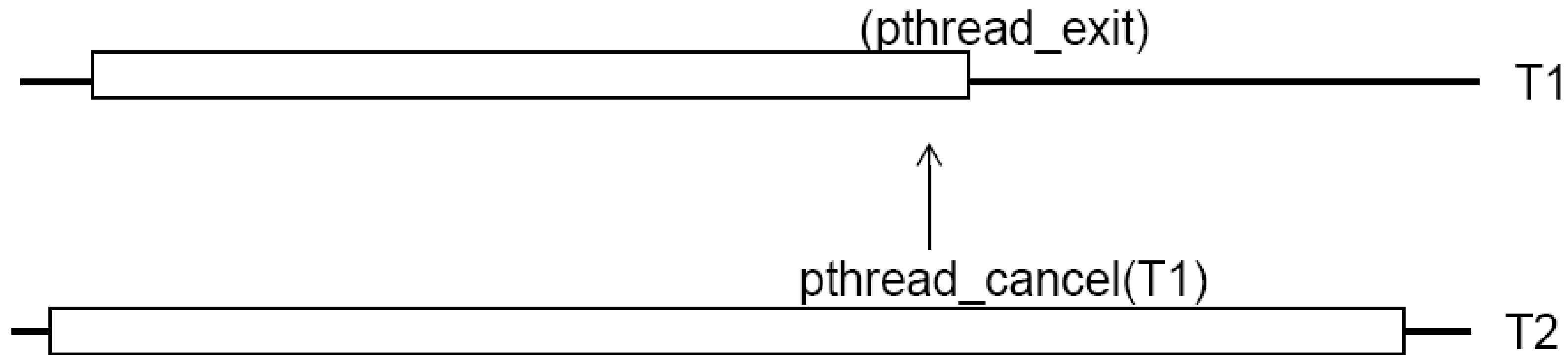
- POSIX brinda la función de sincronización barrier que permite sincronizar varios threads en un mismo punto.
- Una vez que todos los threads llegan a ese punto se continua la ejecución.

```
pthread_barrier_init(pthread_barrier_t *, pthread_barrierattr_t,
      unsigned count)

pthread_barrier_wait(pthread_barrier_t *)
```



CANCELACIÓN DE THREADS

- A veces es necesario finalizar un thread que está ejecutando.



- La función `pthread_cancel` permite que un thread cancele otro en la mitad de su ejecución.

CANCELACIÓN DE THREADS

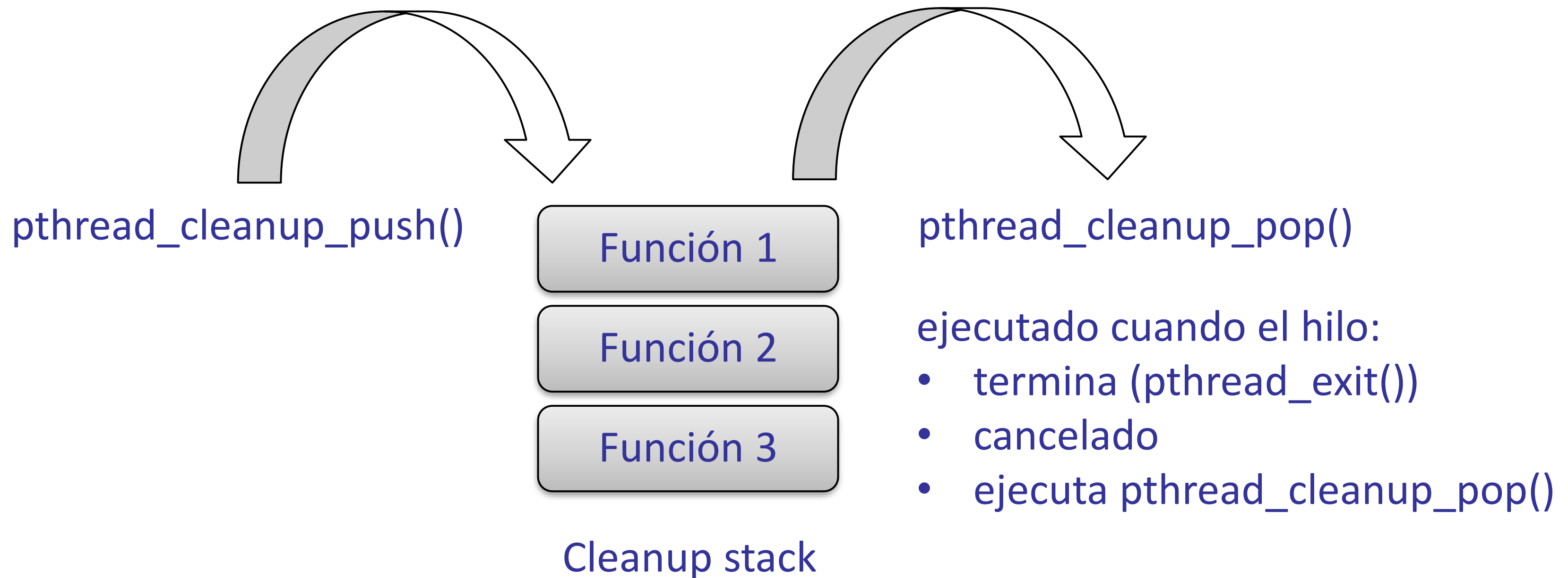
- El thread tiene un estado privado que habilita su cancelación o no.
- Antes de entrar en una sección crítica se puede deshabilitar la cancelación y luego habilitarla nuevamente al salir.
- El estado es configurado a través de la función `pthread_setcancelstate`
- Se definen dos tipos de cancelación.
  - Asincrónica (asynchronous).
    - El thread es cancelado inmediatamente.
  - Diferida (deferred) – por defecto.
    - El thread es cancelado cuando consulta si fue cancelado utilizando la primitiva `pthread_test_cancel`
    - También las funciones de biblioteca pthread son un punto de cancelación. Ej: `pthread_cond_wait, pthread_join`, etc.

CANCELACIÓN DE THREADS

- Problemas que surgen con la cancelación:
  - Que pasa con los recursos que tiene ganados (ej: mutex)?
  - Cómo se libera la memoria pedida dinámicamente (malloc)?
- Solución: handlers de limpieza en cancelaciones.
  - Se provee de una estructura LIFO (stack) que permite configurar funciones de limpieza del thread antes de finalizar su ejecución.
  - Las funciones de limpieza reciben un parámetro de tipo **`void*`**
- Handlers de limpieza:

```
pthread_cleanup_push(void*)
pthread_cleanup_pop(void*)
```

CANCELACIÓN DE THREADS



SEÑALES

- Asignación de señales al igual que a un proceso tradicional.
- Cada thread cuenta con una mascara que le permite filtrar ciertas señales.
  - Un thread hereda la mascara de señales de su padre.
- Si el proceso recibe una señal, se ejecutará la rutina (handler) para **uno** de los threads que no la tenga deshabilitada.
- Se permite generar una señal para un thread específico a través de la primitiva:

```
pthread_kill(pthread_t, int signal)
```

6.4: POOL DE THREADS

POOL DE THREADS

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- La creación y destrucción repetida de threads desperdicia ciclos de procesamiento para la aplicación.
- Para evitar desperdiciar ciclos de procesador creando y destruyendo threads, una solución consiste en la utilización de un pool de threads:
  - Se crea un conjunto de threads los cuales esperan en una variable de condición por una señal.
  - Cuando un thread recibe la señal, este comienza su procesamiento.
  - Al finalizar su trabajo vuelve a esperar en la variable de condición a recibir más trabajo.
- De esta forma se evita estar creando y destruyendo continuamente threads.

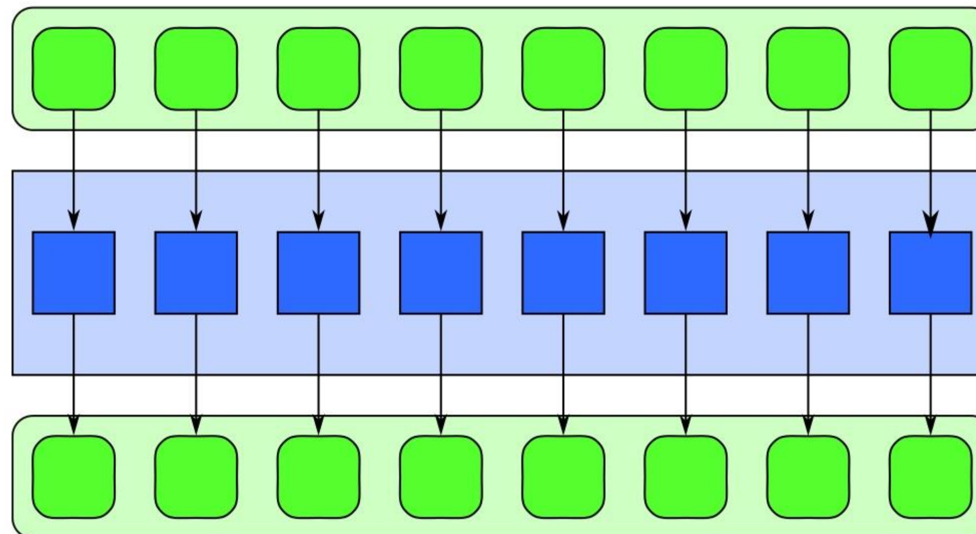
POOL DE THREADS

```
while (1) {
   pthread_mutex_lock(&mtx);
   pthread_cond_wait(&cond,&mtx);

   if (salir) {
     pthread_mutex_unlock(&mtx);
     return NULL;
   }

   pthread_mutex_unlock(&mtx);

   ...
}
```

6.5: PATRONES PARALELOS DE CONTROL

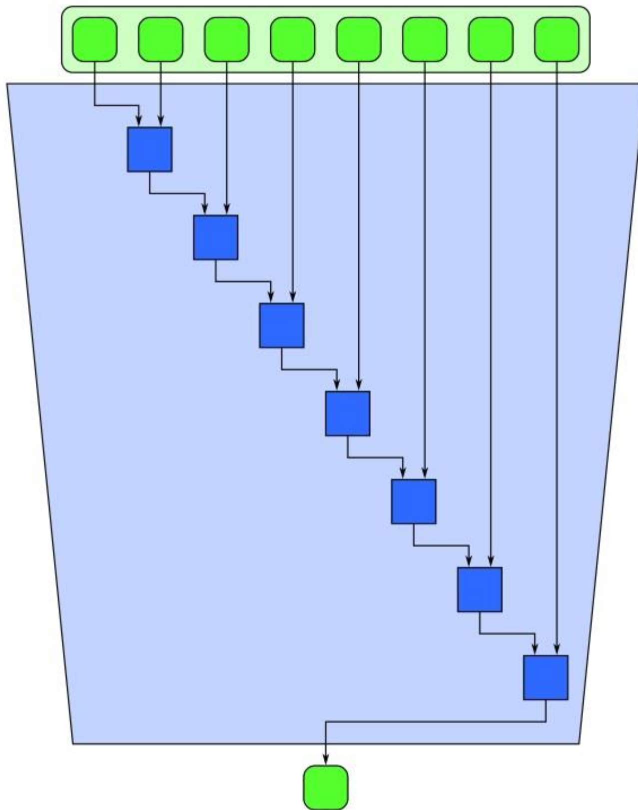
PATRÓN MAP

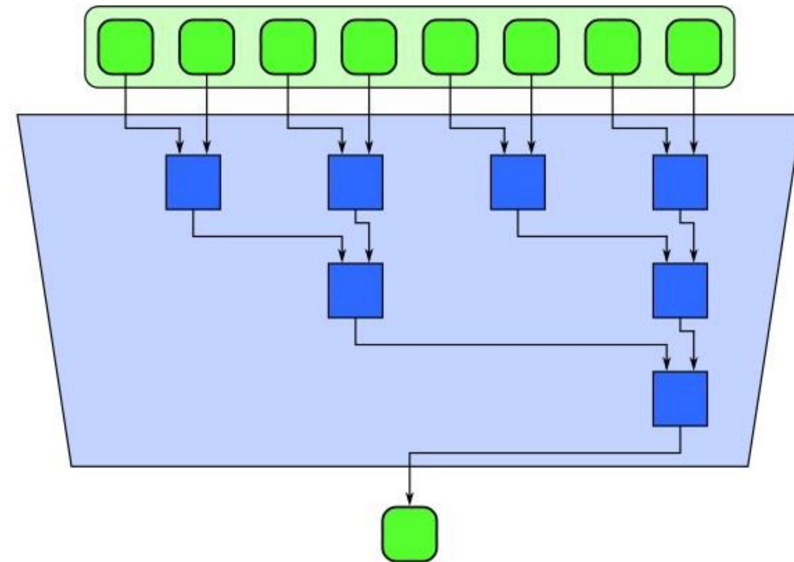
- Map aplica una función sobre cada elemento de un conjunto
- Patrón muy simple:
  - La aplicación de la función es independiente para cada elemento
  - La cantidad de elementos es conocida
- Muy relacionado con arquitecturas SIMD
- Se corresponde con la paralelización de un loop para el caso particular en el que todas las iteraciones son independientes



PATRÓN REDUCCIÓN

- Reducción combina todos los elementos de un conjunto en un único elemento utilizando una función de agregación





6.6: REENTRANCIA VS THREAD-SAFE

REENTRANCIA VS THREAD-SAFE

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Una función es thread-safe cuando su código puede ser ejecutado en paralelo por más de un thread y su resultado es el mismo que ejecutado en forma serial.
  - Puede utilizar locks para proteger el acceso a recursos compartidos.
- Una función es reentrante cuando su código puede ser interrumpido y vuelto a ejecutar sin que la ejecución de ninguna de sus instancias se vea afectada.
  - Como regla general no debe referenciar datos estáticos o globales. Si lo hace, debe manipularlos de forma atómica.
  - Una función reentrante no debe hacer uso de una función no-reentrante.
- Una función puede ser reentrante, thread-safe, ambos, o ninguno.

REENTRANCIA VS THREAD-SAFE

- Los threads comparten el espacio de direccionamiento global, excepto el program counter, el stack y los registros.
- Cada thread tiene su propio stack, las variables locales a una función NO SON COMPARTIDAS entre threads.
- Sin embargo, variables globales y las estáticas definidas en un thread SI SON COMPARTIDAS entre los threads.

- Algunas funciones en las bibliotecas de C no son thread-safe.
  - `strtok` para la extracción de tokens de un string.
    - «The strtok() function uses a static buffer while parsing, so it's not thread safe.»
  - `rand` para la generación de un números aleatorios.
    - «The function rand() is not reentrant or thread-safe, since it uses hidden state that is modified on each call.»

REENTRANCIA VS THREAD-SAFE

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i;

   /* Inicializa la semilla */
   srand(1);

   for( i = 0 ; i < 5 ; i++ ) {
      printf("%d\n", rand());
   }

   return(0);
}
```

REENTRANCIA VS THREAD-SAFE

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Se ofrecen versiones reentrantes de muchas funciones. Por ejemplo: `rand_r` , `strtok_r`, etc.

Uso de rand()

```
srand(1);
int p = rand();
```

Uso de rand_r()

```
int seed = 1;
int p = rand_r(&seed);
```

6.7: PARALELISMO AUTOMÁTICO

OPENMP

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Open Multi-Processing (OpenMP) es una API para el desarrollo de aplicaciones multihiladas con memoria compartida
- Compuesta por tres grandes componentes:
  - Directivas del compilador (`#pragma`)
  - Bibliotecas en tiempo de ejecución
  - Variables de entorno
- ¿Qué NO es OPENMP?
  - No es una biblioteca para desarrollar aplicaciones distribuidas
  - No está necesariamente implementado siempre igual
  - No garantiza el uso más eficiente de la memoria compartida
  - No resuelve condiciones de carrera, deadlocks, conflictos de datos, etc.
  - No está diseñada para manejar I/O en paralelo

Se presentará en detalle OpenMP en el tema 7

¿CUÁLES SON SUS OBJETIVOS?

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Estandarización
  - Proveer una API estándar para las diferentes arquitecturas y plataformas de memoria compartida
  - Desarrollo conjunto por un grupo importante de empresas
    - AMD, IBM, Intel, HP, Fujitsu, Nvidia, NEC, Red Hat, Oracle, etc.
- Interfaz clara y directa
  - Conjunto simple y limitado de directivas
  - Tan solo 3 o 4 directivas son suficientes para muchos casos
- Facilidad de uso
  - Permitir la paralelización incremental de un programa serial
  - Permitir implementar paralelismo de grano grueso y fino
- Portabilidad
  - Especificación en C/C++ y Fortran
  - Muchas plataformas, incluidas Unix/Linux y Windows

OPENMP: BUCLE PARALELO

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

```c
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```

OPENMP VS POSIX THREADS

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- OpenMP incrementa la robustez de las aplicaciones, pero es menos flexible y en general menos eficiente que el paralelismo de bajo nivel.
  - Requiere que el problema admita una partición de dominio acorde a sus primitivas paralelas (parallel regions).
- POSIX ofrece una interfaz menos *elegante* a cambio de mayor libertad y potencialidad para el programador.
- ¿Cuándo utilizar OpenMP?
  - Aplicaciones regulares.
  - Aplicaciones multiplataforma.
  - Existen loops paralelizables.
  - Necesidad de optimizaciones de último minuto o incremental.
- ¿Cuándo usar pthreads?
  - Eficiencia es realmente un objetivo fundamental.
  - Operaciones no regulares.
  - Sacar mejor partido de la eficiencia del compilador.

6.8: OPTIMIZANDO C

OPTIMIZACIÓN AL COMPILAR

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Los compiladores modernos realizan muchas optimizaciones sobre el código para mejorar su desempeño
- Un ejemplo: eliminación de punteros
  - Un puntero puede ser eliminado si apunta a algo conocido. Por ejemplo:

```
int a;
int *p = &a;
*p = *p + 2;
```

  - Puede ser optimizado a:

```
int a;
a += 2;
```

- Otro ejemplo: loop unrolling
  - Algunos compiladores pueden hacer unroll de los loops

```
int i, a[2];
for (i=0;i<2;i++) a[i]=i+1;
```

  - Puede ser optimizado a:

```
int a[2];
a[0]=1; a[1]=2;
```

OPTIMIZACIÓN AL COMPILAR

- Flags de optimización de GCC
  - Por defecto el compilador reduce el tiempo de compilación y produce binarios debuggeables
  - `-O0`: por defecto
  - `-O1`: reduce el tamaño del código y del tiempo de ejecución sin reducir demasiado mucho tiempo de compilación
  - `-O2`: aplica todas las optimizaciones con un compromiso de velocidad y tamaño
  - `-O3`: enciende todo `-O2` y otras que optimizan velocidad a costa de tamaño
  - `-Ofast`: optimizaciones `-O3` y otras que ignoran la conformidad a estándares

¿DÓNDE ALMACENAR VARIABLES?

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Uso de variables globales o estáticas
  - Pueden ser inicializadas junto con la carga del programa
  - Ocupan memoria hasta la finalización del programa
- Uso del heap
  - Almacena memoria dinámica pedida utilizando `new/delete` y `malloc/free`
  - Fácilmente fragmentable: uso poco eficiente del caché
  - El sistema usa mecanismos para serializar el acceso a las estructuras del heap
  - Los compiladores no son eficientes optimizando memoria dinámica
- Uso del stack
  - Todas (casi) las variables declaradas en una función están en el stack
  - Lugar muy eficiente para almacenar variables porque es de uso muy frecuente
    - Casi seguramente se encuentre almacenado totalmente en el caché L1 de la CPU
  - Es posible reducir el scope de ciertas variables usando { }
  - El tamaño del stack es limitado

¿DÓNDE ALMACENAR VARIABLES?

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Hay que evitar utilizar grandes estructuras estáticas en el stack

```
void memoryHog() {
    char array[10*1024*1024]; // 10 MB en el stack
}
```

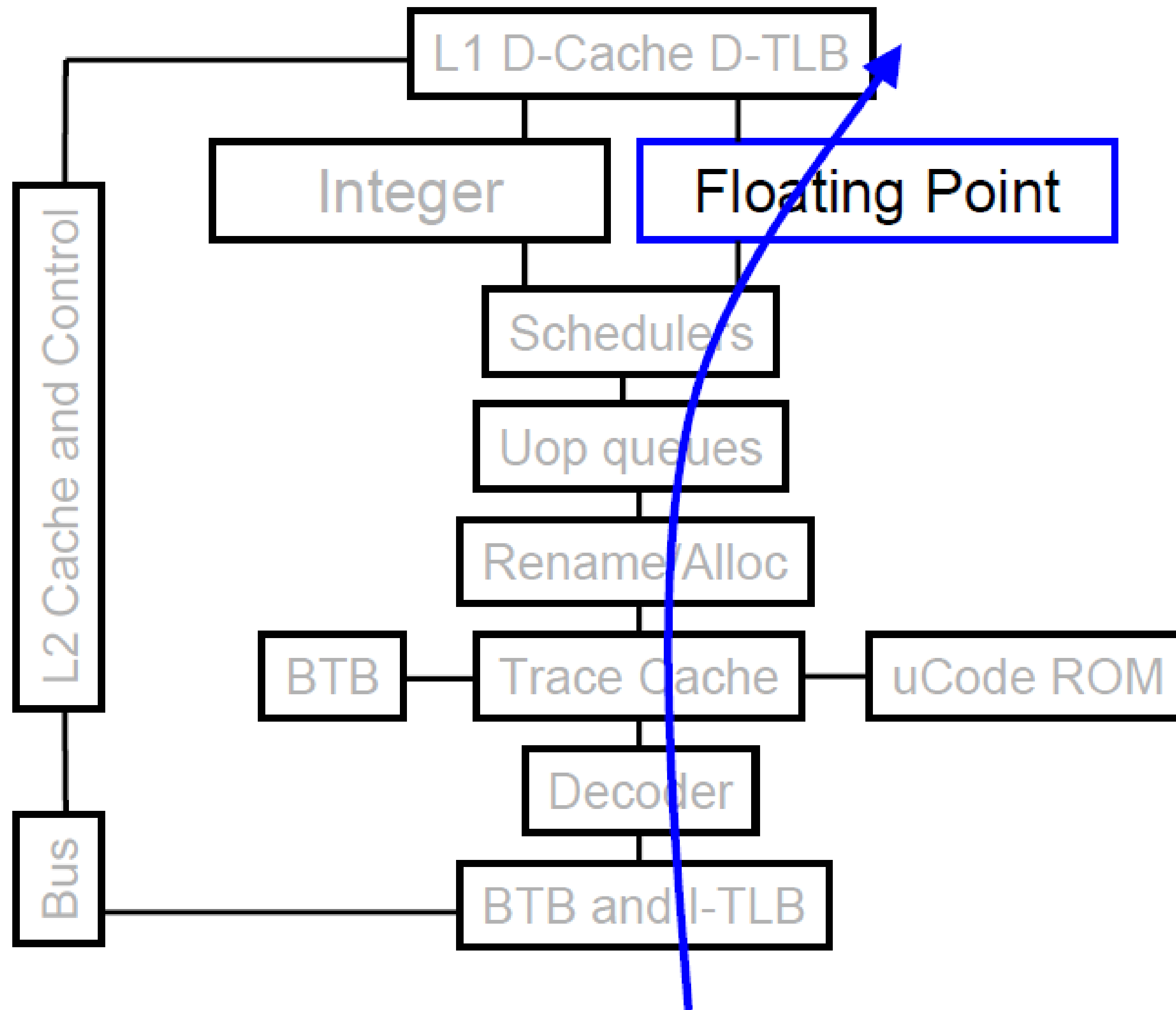
- El stack tiene tamaño limitado
- Atenta contra el uso eficiente del caché
- La mejor alternativa es utilizar variables estáticas o globales
    - …aunque parezca «feo»

6.9: ARQUITECTURAS PARALELAS

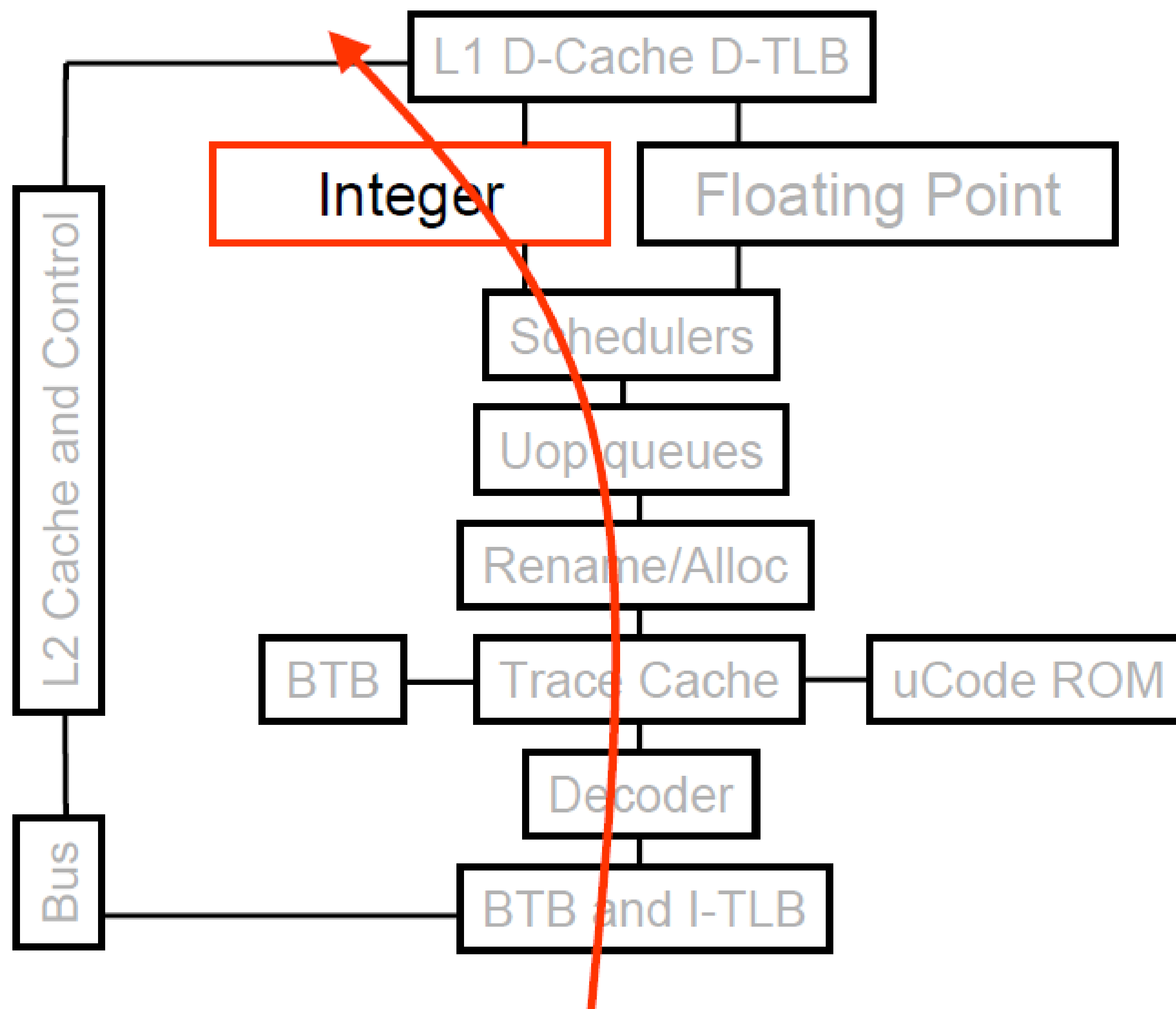
ARQUITECTURAS PARALELAS

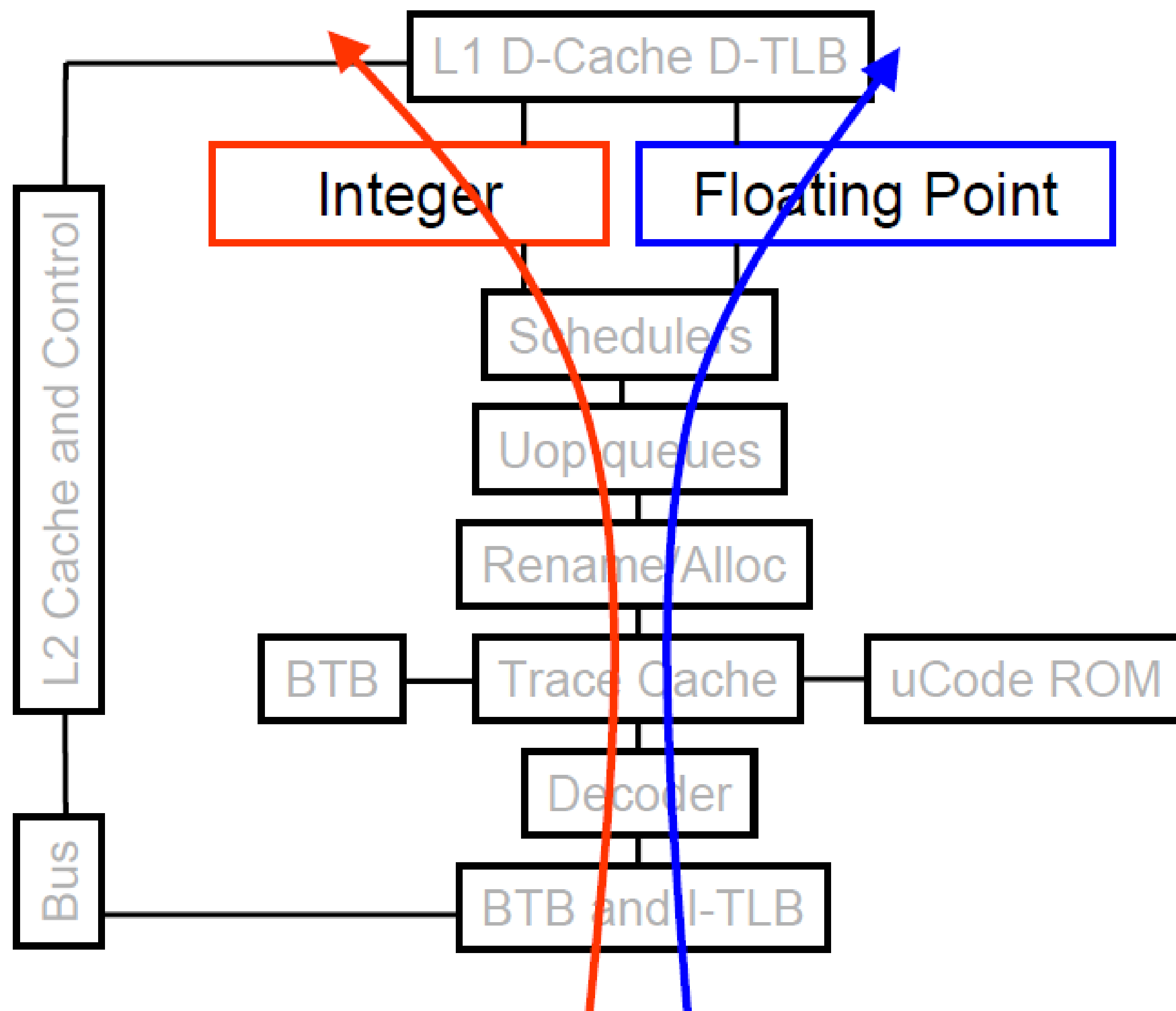
UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Existen diferentes niveles de paralelismo: de instrucción y de thread
- A nivel de instrucción
  - El CPU puede reordenar instrucciones, pipeline, branching, predicción, etc.
  - Puede aprovecharse con un único procesador
- A nivel de hilo
  - Paralelismo de grano más grueso
  - Requiere más de un procesador
- La tendencia actual es al paralelismo multinúcleo
- Recientemente surgió una nueva técnica complementaria al multinúcleo: multihilado simultáneo (SMT)
  - Permite que múltiples hilos ejecuten simultáneamente en el mismo núcleo
  - Por ejemplo: cuando un hilo está esperando por una operación de punto flotante, otro hilo puede utilizar la unidad entera

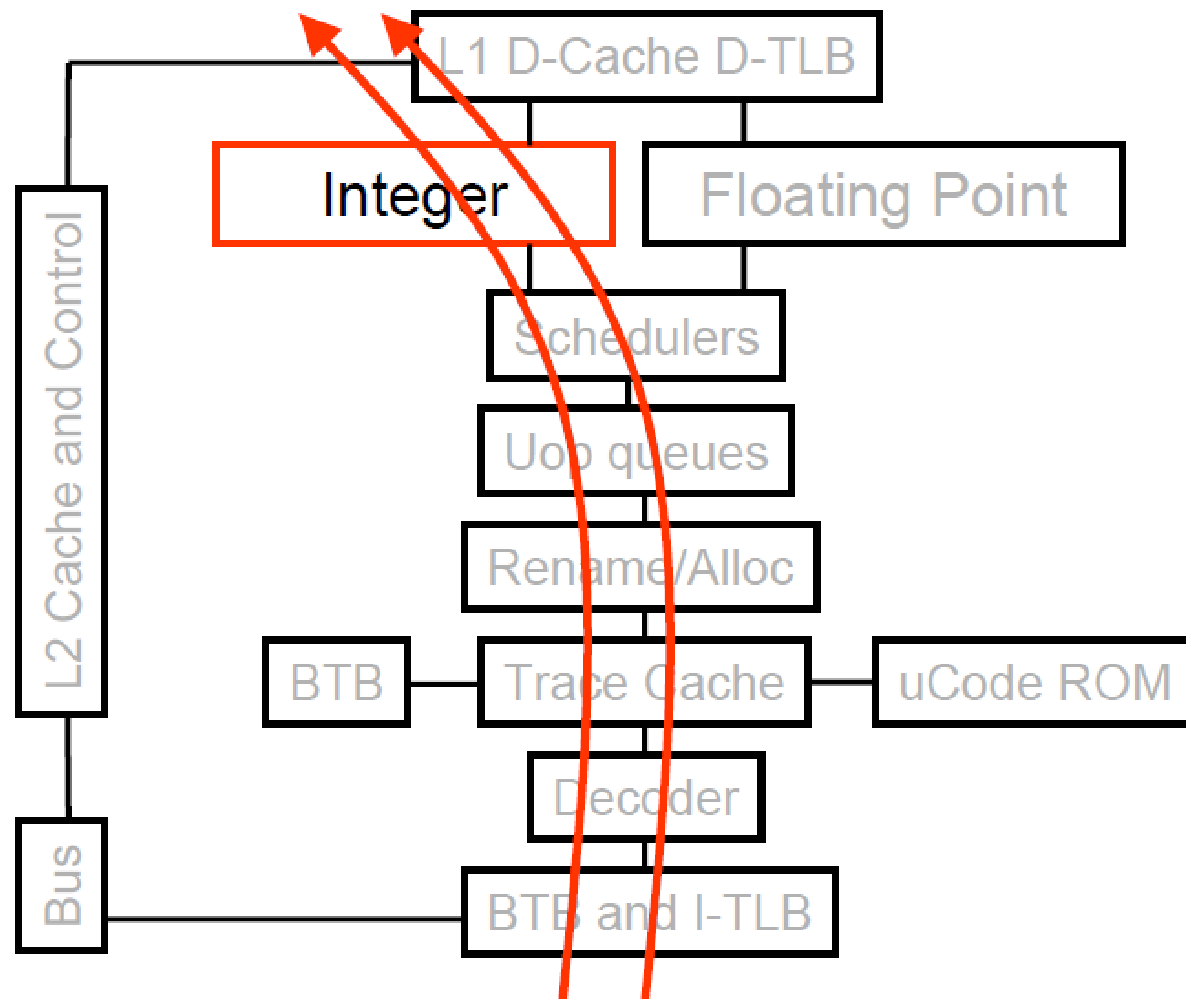

L1 D-Cache D-TLB
Integer
Floating Point
Schedulers
Uop queues
Rename/Alloc
BTB
Trace Cache
uCode ROM
Decoder
BTB and I-TLB
L2 Cache and Control
Bus

ARQUITECTURAS PARALELAS




L1 D-Cache D-TLB
Integer
Floating Point
L2 Cache and Control
Schedulers
Uop queues
Rename/Alloc
BTB
Trace Cache
uCode ROM
Decoder
Bus
BTB and I-TLB

ARQUITECTURAS PARALELAS

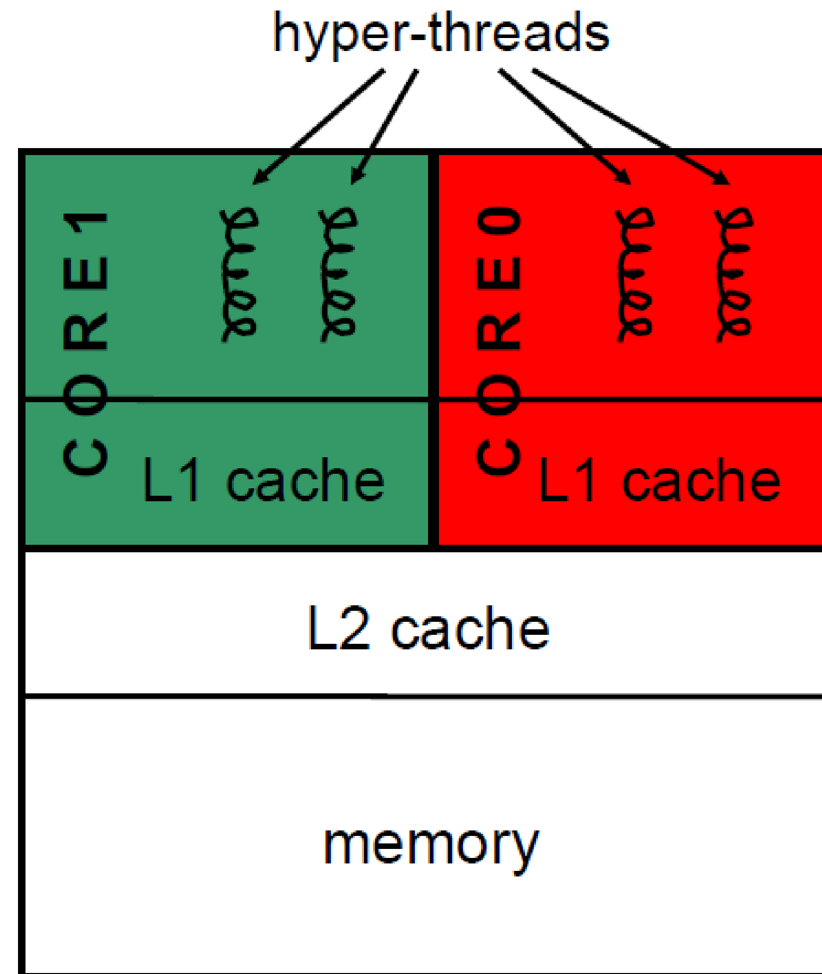


ARQUITECTURAS PARALELAS

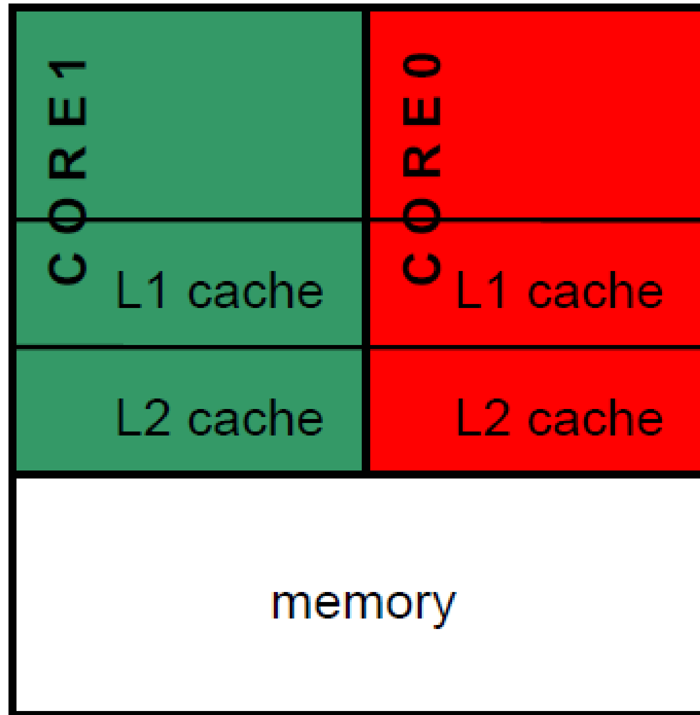
- La propuesta fue implementada por la tecnología Hyperthreading de Intel.
- El SO ve cada SMT como un núcleo virtual separado
  - SMT no es realmente multinúcleo
  - Principalmente se basa en explotar mecanismos de paralelismo a nivel de instrucción
- Para aplicaciones intensivas en cómputo (como HPC) el hyperthreading tiende a ser contraproducente
  - Existe mucha competencia por las unidades de procesamiento
  - Mejores resultados de eficiencia (reportados por Intel): 15% a 25%
  - Las mejoras son muy dependientes de la aplicación
  - En general, cuando dos aplicaciones que demandan intensivamente el procesador trabajan con hyperthreading, se degrada la performance de ambas
  - No es eficiente desde el punto de vista energético

JERARQUÍA DE MEMORIA

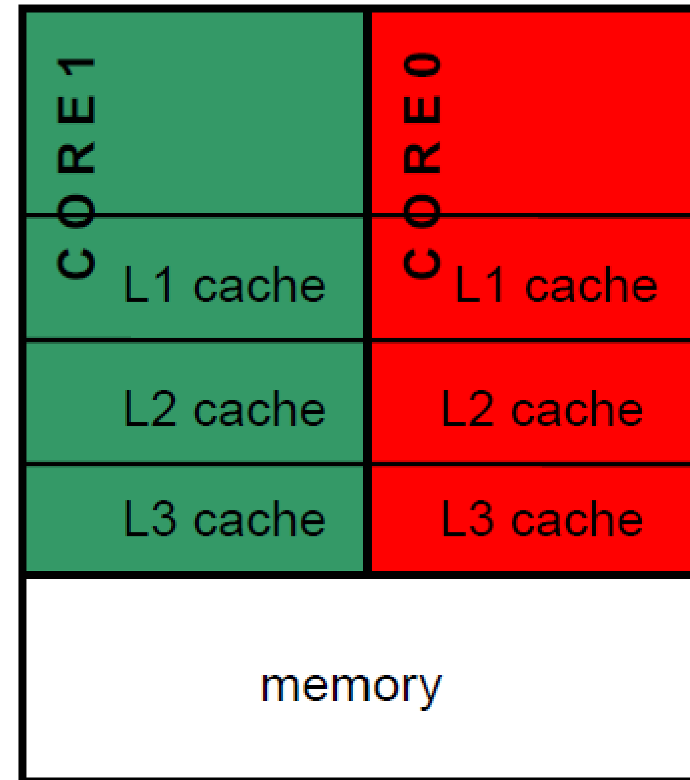
- Arquitectura Intel Xeon
- Cada core tiene hyperthreading
- Caché L1 privado
- Caché L2 compartido



JERARQUÍA DE MEMORIA



- Caché L1 y L2 privados
- Arquitecturas AMD Opteron, AMD Athlon, Intel Pentium D

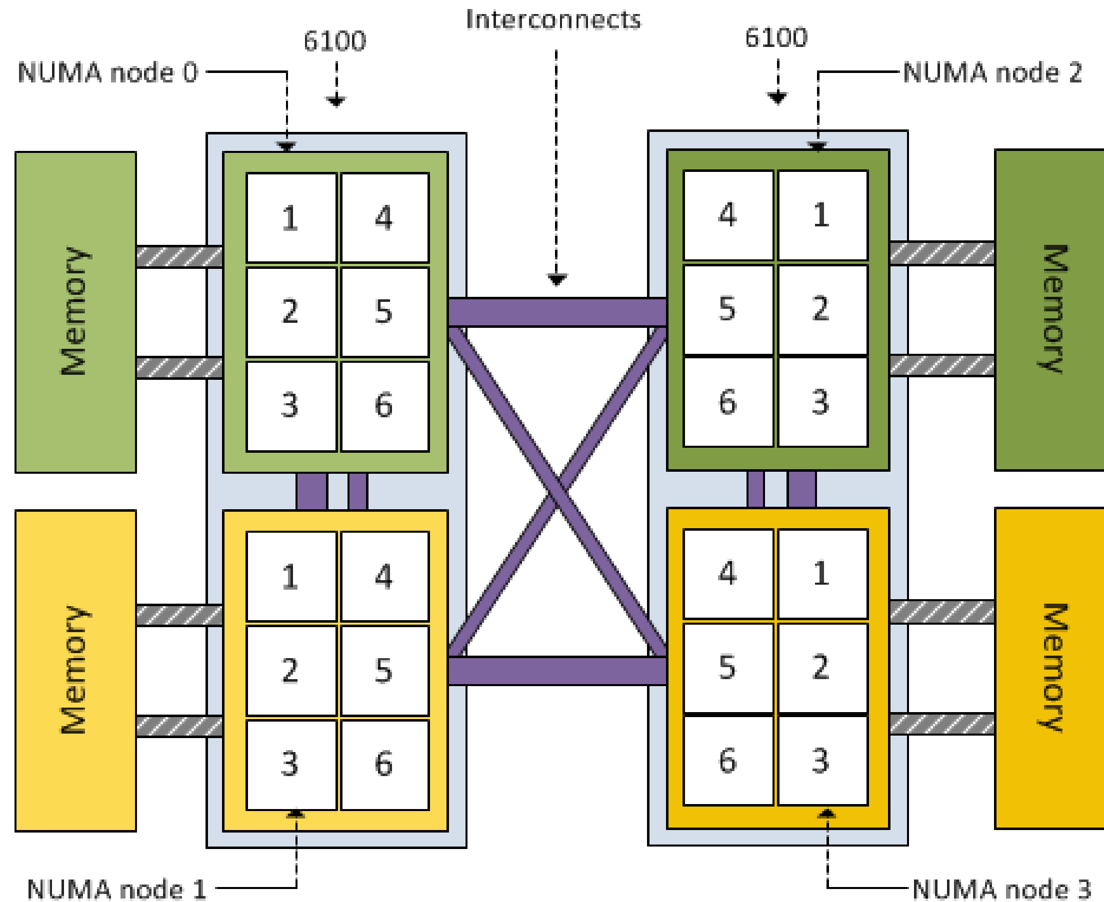


- Diseño con cachés privados y L3
- Arquitectura Intel Itanium 2

JERARQUÍA DE MEMORIA

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- La memoria RAM es siempre compartida
  - … aunque para arquitecturas de muchos núcleos su acceso puede ser no uniforme (arquitecturas NUMA)



¿DÓNDE EJECUTAN LOS HILOS?

- El SO intenta balancear la carga y migrar hilos entre núcleos
- La migración de un hilo es costosa. Incluye:
  - Reiniciar el pipeline de ejecución, invalidar el caché, etc.
- Se evita la migración y se mantiene afinidad con el núcleo inicial
  - El modelo se llama «soft affinity»
- El programador puede especificar una afinidad dura («hard affinity»)
  - Cada hilo/proceso tiene una máscara de afinidad (affinity mask)
  - En la máscara de afinidad se especifica que núcleos pueden ejecutar el hilo
- Existen bibliotecas que ayudan a este tipo de optimización
  - Por ejemplo: Portable Hardware Locality (hwloc)

RECURSOS ONLINE

- POSIX Threads Programming Tutorial, Lawrence Livermore National Laboratory.
  - https://computing.llnl.gov/tutorials/pthreads/
- Portable Hardware Locality (hwloc)
  - https://www.open-mpi.org/projects/hwloc/