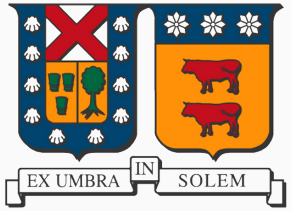


Sistemas Operativos

Hebras & Concurrencia

Viktor Andrés Tapia Vásquez

Segundo Semestre 2021

Departamento de Informática, Campus SSJJ.

Índice de Contenidos

Hebras

Hebras

- SO's modernos permiten que un proceso tenga múltiples hebras.
- Esto afecta directamente el diseño del SO.
- Aplicaciones modernas están construídas con hebras.
- La idea es simple, separar tareas dentro de un proceso:
  - ✓ Actualizar pantalla de datos.
  - ✓ Obtención de la fecha.
  - ✓ Corrección ortográfica.
  - ✓ Responder una solicitud de red.

Hebras

- Una hebra es una secuencia simple de ejecución que representa una tarea itinerante separada. Es una unidad básica de uso de CPU.
- Compuesta por un id, contadores, registros y un stack.
- Comparte con otras hebras, que son parte del mismo proceso, la sección de código, datos y recursos que pueda tener asignado.

Hebras

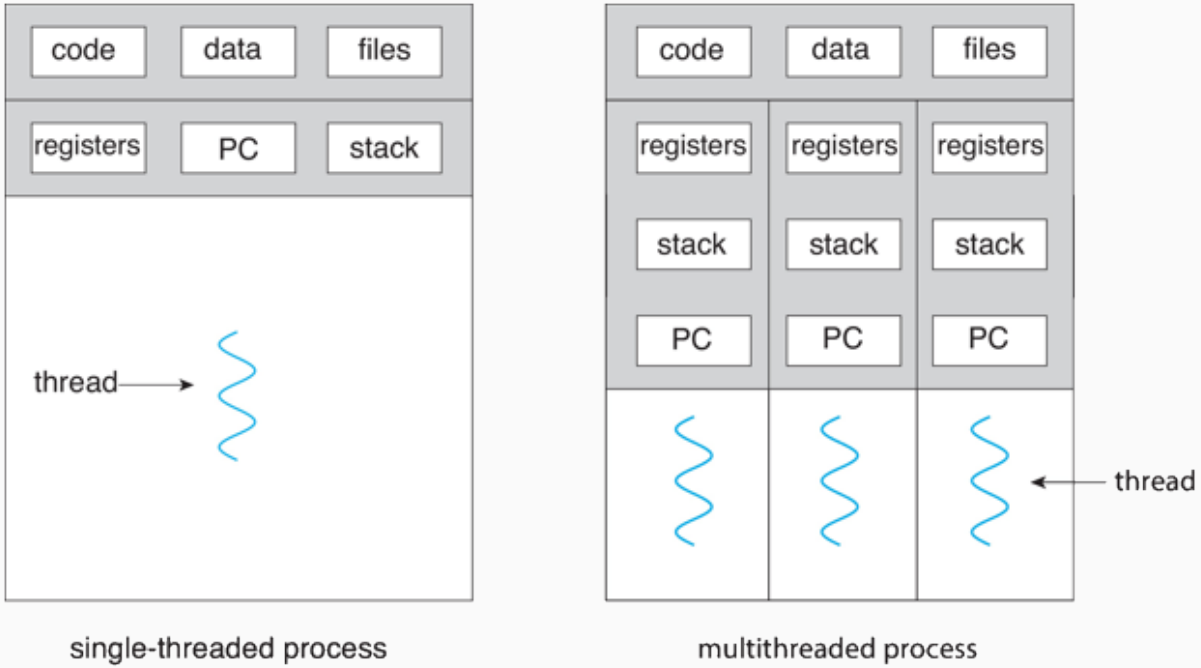


Figure 1: Comparación procesos con hebras

Hebras - Beneficios

- **Capacidad de respuesta:** Programa puede continuar su ejecución a pesar de que parte de él esté bloqueda.
- **Recursos compartidos:** Ventaja sobre los IPC. Es mucho más fácil.
- **Economía:** Creación y cambios de contexto son menos costosos.
- **Escalabilidad:** Ventajas en arquitecturas con múltiples núcleos.

Hebras

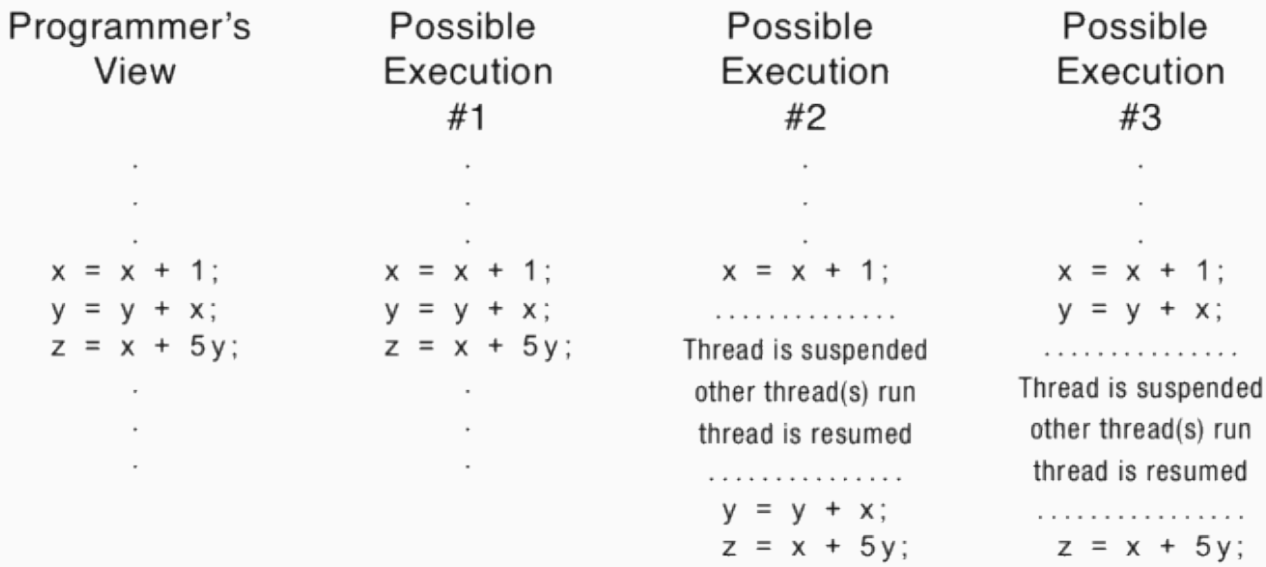


Figure 2: Concurrencia con hebras

Hebras

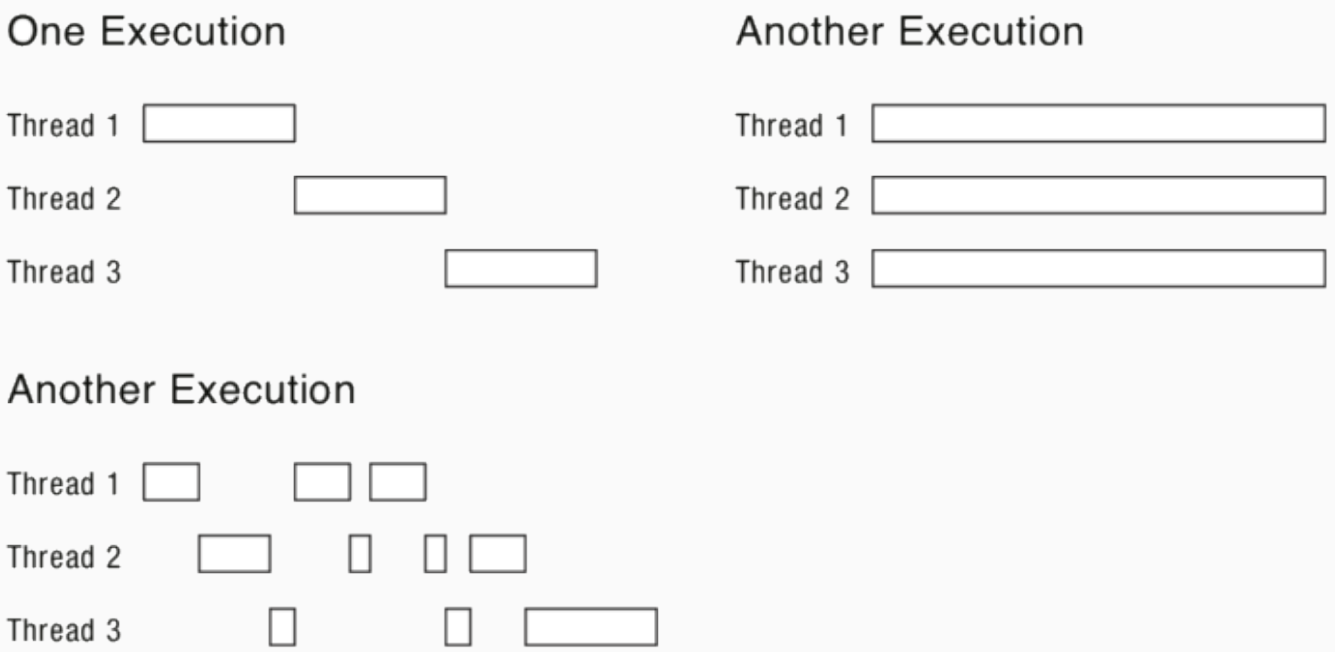


Figure 3: Concurrencia con hebras

API simple para Hebras

API simple para Hebras

- **sthreads:** basada en POSIX, pero simplifica su uso omitiendo opciones y manejo de errores. La mayoría de los paquetes de gestión de hebras son similares.
- Necesita sthread.c, sthread.h. Disponibles en:

  aula.usm.cl

API simple para Hebras

- **void sthread_create(thread, func, arg)**. Crea una nueva hebra almecanando la información en **thread**. Ejecuta la función **func**, la cual se llamará con el argumento **arg**.
- **void sthread_yield()**. La hebra que invoca esta función voluntariamente abondana el procesador para dejar que otra hebra lo pueda utilizar. El itinerador decide cuando continuar con la hebra.

API simple para Hebras

- **int sthread_join(thread)**. Espera que una hebra específica termine. Retorna el valor de sthread exit(valor). Notar que se puede llamar una vez para hebra.
- **void sthread_exit(ret)**. Termina la hebra que lo invoca. Almacena el valor de ret en la estructura de datos de la hebra. Si una hebra está esperando en un join, se despierta.

API simple para Hebras

- El siguiente ejemplo muestra el programa threadHola.c que utiliza la API sthreads. Para compilar:

```
$ gcc -g -Wall -Werror -D_POSIX_THREAD_SEMANTICS threadHola.c -c -o threadHola.o
$ gcc -g -Wall -Werror -D_POSIX_THREAD_SEMANTICS sthread.c -c -o sthread.o
$ gcc -lpthread threadHola.o sthread.o -o threadHola
  * Para correrlo
$ ./threadHola
```

API simple para Hebras

```c
/*
  * threadHello.c -- Simple multi-threaded program.
  *
  * Compile with
  *    > gcc -g -Wall -Werror -D_POSIX_THREAD_SEMANTICS threadHello.c -c -o
threadHello.o
  *    > gcc -g -Wall -Werror -D_POSIX_THREAD_SEMANTICS sthread.c -c -o sthread.o
  *    > gcc -lpthread threadHello.o sthread.o -o threadHello
  * Run with
  *    > ./threadHello
  */
#include <stdio.h>
#include "sthread.h"

static void go(int n);

#define NTHREADS 10
static sthread_t threads[NTHREADS];

int main(int argc, char **argv)
{
  int ii;

  for(ii = 0; ii < NTHREADS; ii++){
    sthread_create(&(threads[ii]), &go, ii);
  }
  for(ii = 0; ii < NTHREADS; ii++){
    long ret = sthread_join(threads[ii]);
    printf("Thread %d returned %ld\n", ii, ret);
  }
  printf("Main thread done.\n");
  return 0;
}

void go(int n)
{
  printf("Hello from thread %d\n", n);
  sthread_exit(100 + n);
  // Not reached
}
```

Figure 4: Ejemplo

API simple para Hebras

- ¿Qué otra salida es posible?
- ¿Cuál es el máximo número de threads corriendo al mismo tiempo?
- ¿El mínimo?

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Figure 5: Salida

Implementación

Implementación

- El SO entrega la ilusión de que cada hebra usa su propio procesador.
- Esto lo logra suspendiendo y despertando las hebras.
- Será necesario utilizar estados internos. Se almacenarán en el Kernel.
- La estructura de datos se denomina TCB. Guarda:
  - ✓ Estado de la computación.
  - ✓ Metada de la hebra.

Implementación

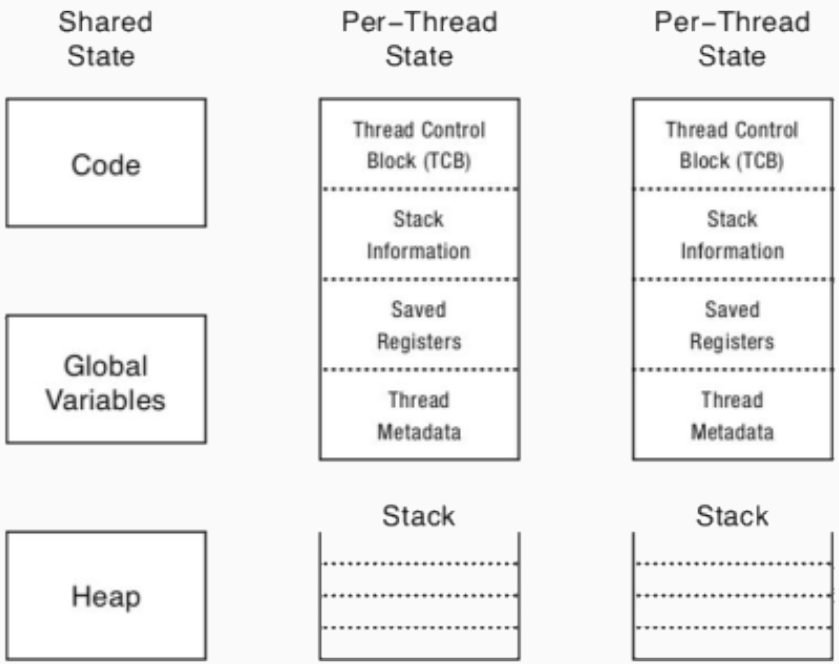


Figure 6: TCB

Implementación

Cada hebra contiene dos elementos de estado que lo determinan:

- **Stack:** Información de procedimientos y funciones anidadas de la hebra que está en ejecución. Se almacenan variables temporales, parámetros y direcciones de retorno.
- **Metadata:** Por ejemplo el tid, prioridades, etc.

Implementación

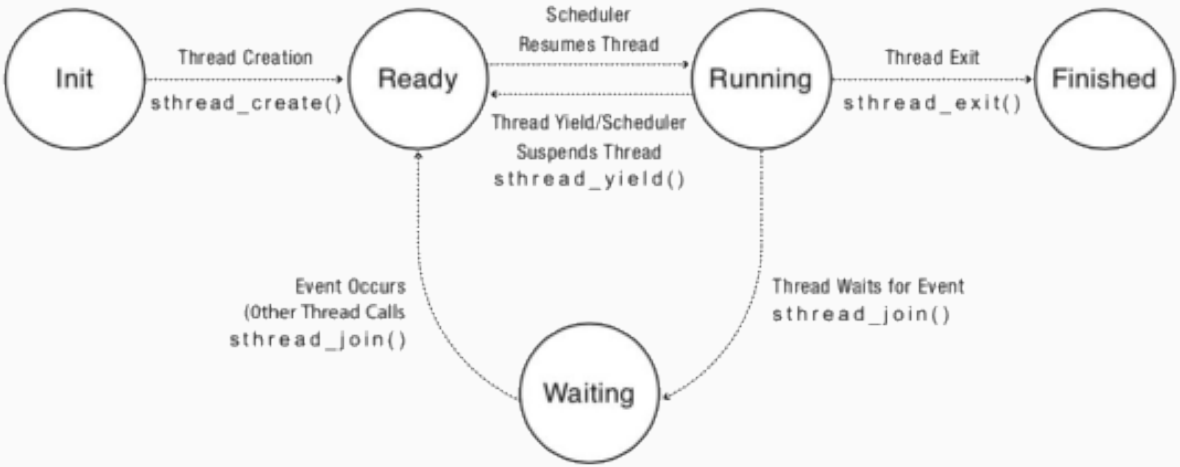


Figure 7: Diagrama de estados

Implementación

Estado	Ubicación TCB	Ubicación Registros
INIT	Siendo creado	TCB
READY	Ready List	TCB
RUNNING	Running List	Procesador
WAITING	Cola espera (variable de sincronización)	TCB
FINISHED	Lista Finnished y eliminación	TCB

Figure 8: Tabla de estados

Implementación

Podremos clasificar las hebras en dos tipos:

- **Usuario:** La gestión la realizan librerías de hebras.
  - ✓ POSIX Pthreads.
  - ✓ Windows Threads.
  - ✓ Java Threads.
- **Kernel:** Soportadas por el kernel del SO.
  - ✓ Su objetivo es ejecutar instrucciones privilegiadas.
  - ✓ Ejemplo: Llamadas al sistema.

Implementación

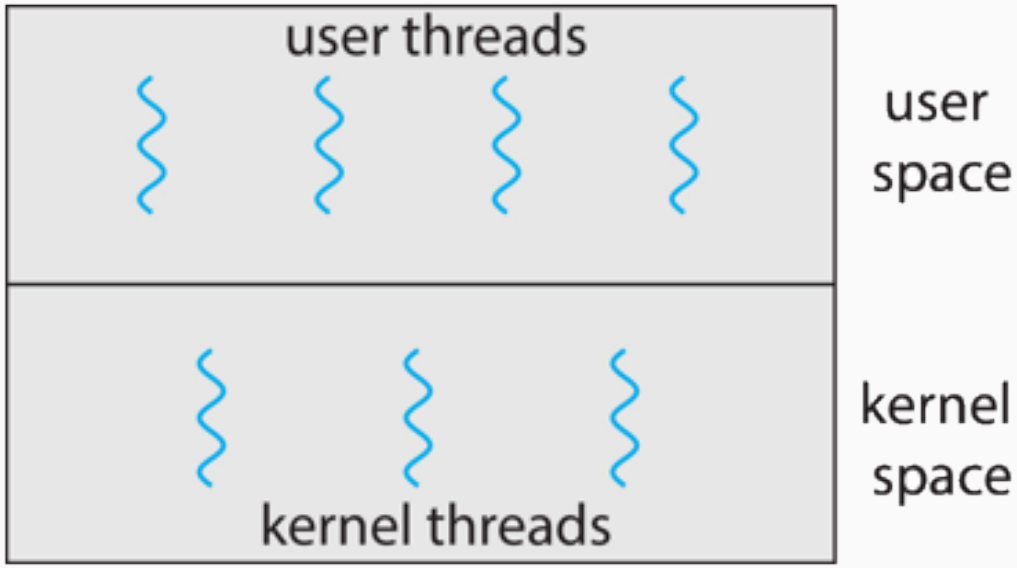


Figure 9: Tipos de hebras

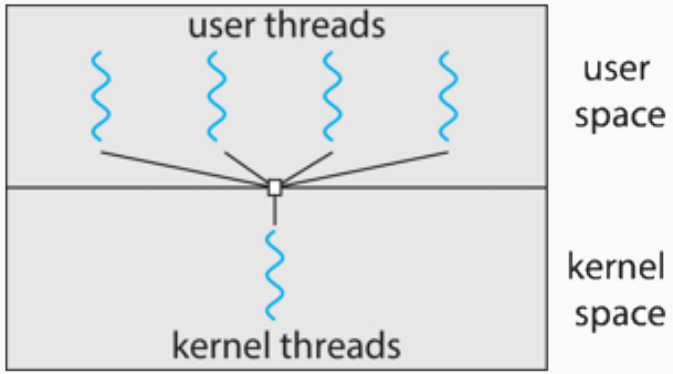
Implementación - Modelos

Esta clasificación se puede implementar de 3 formas:

- Muchas hebras de usuario a una hebra de kernel.
- Una hebra de usuario a una hebra de kernel.
- Muchas hebras de usuario a muchas hebras de kernel.

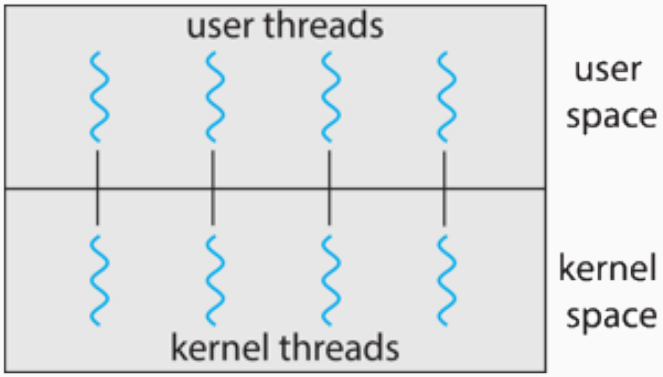
Implementación - Modelos - Muchas a una

- Una hebra puede bloquear el proceso completo.
- No existe paralelismo real sobre arquitecturas multicore.
- Pocos sistemas usan este modelo:
  - ✓ Solaris Green threads.
  - ✓ GNU Porable threads.



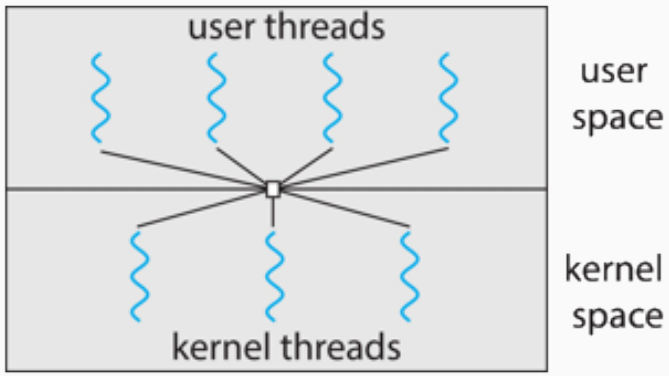
Implementación - Modelos - Una a una

- Mayor grado de concurrencia.
- Si una se bloquea, las otras pueden seguir.
- Paralelismo en arquitecturas multicore.
- Es costosa, ya que se crean tantos threads de kernel como de usuario. Impacta en el rendimiento.
- Windows y Linux sigue este modelo.



Implementación - Modelos - Muchas a muchas

- Total de hebras de kernel depende del SO.
- Su uso no es muy común.
- Se multiplexan muchas hebras de usuario a kernel.



Implementación - Ejemplos

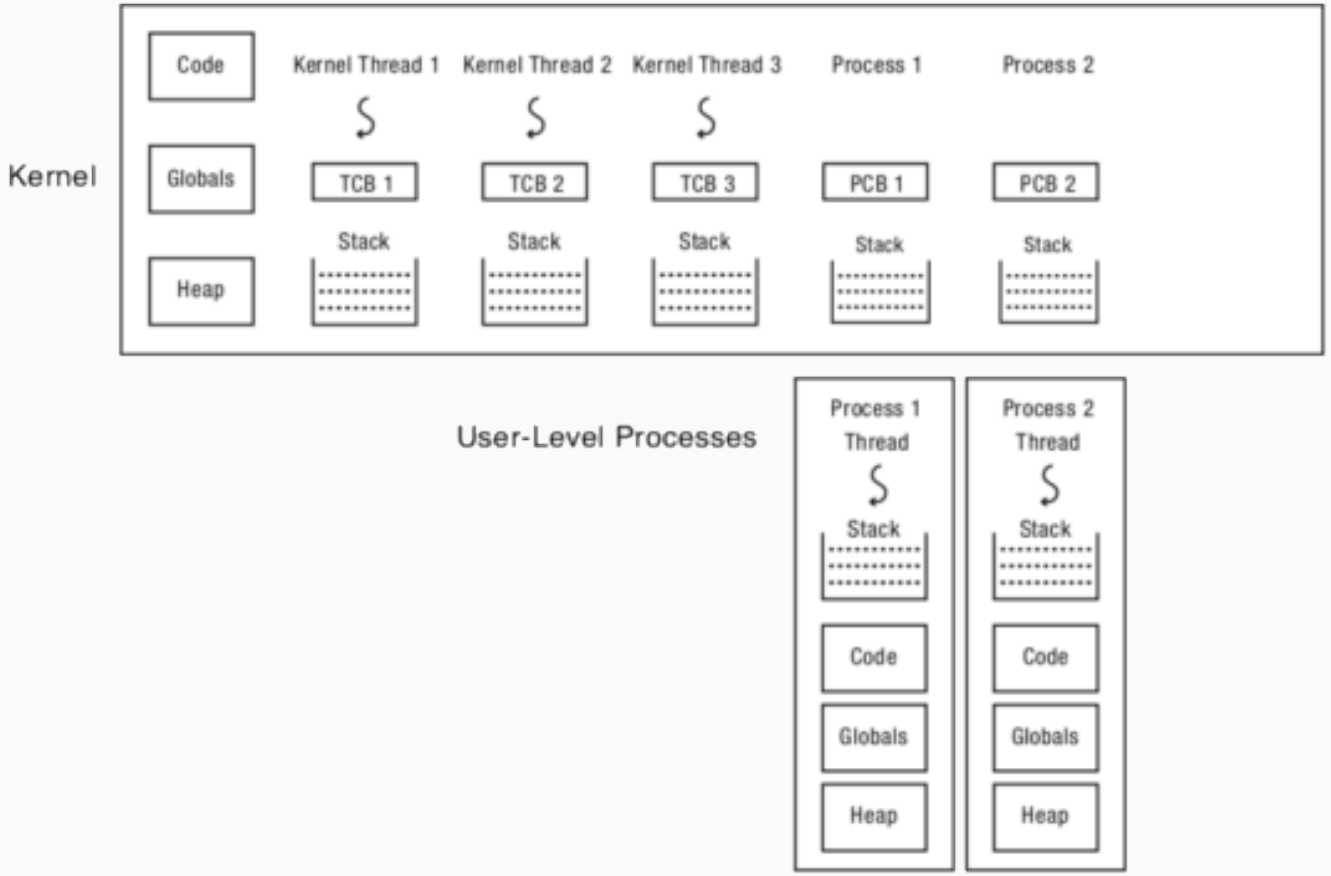


Figure 10: Ejemplo I

Implementación - Ejemplos

- Cada proceso tiene una hebra principal.
- Un proceso es más que una hebra.
- Cada proceso tiene su espacio de direcciones.
- Cada proceso tiene su propia vista de memoria, código, stack, heap, variables globales, etc.
- La PCB necesita mayor información que la TCB.
- Tanto la PCB y la TCB representan una hebra. La ready list del kernel contiene una mezcla de PCB y TCB.

Implementación - Ejemplos

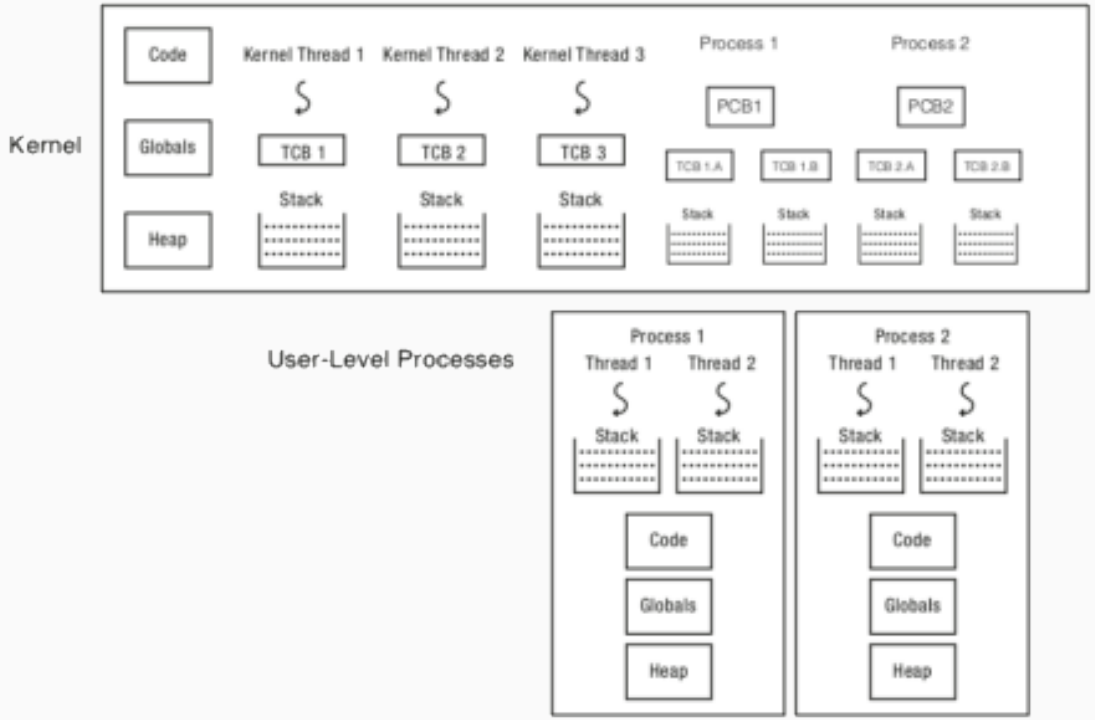


Figure 11: Ejemplo II

Implementación - Ejemplos

- La ready list del kernel incluye los TCB de las hebras del kernel y uno o más PCB para proceso de usuario.
- El mecanismo de cambio de contexto permite conmutar:
  - ✓ Hebras de kernel.
  - ✓ Hebras de kernel y hebras de procesos.
  - ✓ Hebras de diferentes procesos.
  - ✓ Hebras de un mismo proceso.
- ¿Por qué un proceso requiere usar hebras?
  - ✓ Estructurar programas concurrentes.
  - ✓ Explotar arquitecturas multicore.

Aspectos Adicionales

Aspectos Adicionales - Sistemas Multicore

Agregan un desafío mayor para el programador:

- Dividir actividades.
- Balancear.
- Separación de datos.
- Dependencia de datos.
- Pruebas y debug.

Aspectos Adicionales - Sistemas Multicore

- **Paralelismo:** Ejecutar más de una tarea simultáneamente.
- **Concurrencia:** Permite que más de una tarea en ejecución avance. Arquitecturas con un procesador lo logran utilizando el itinerador.

Aspectos Adicionales - Sistemas Multicore

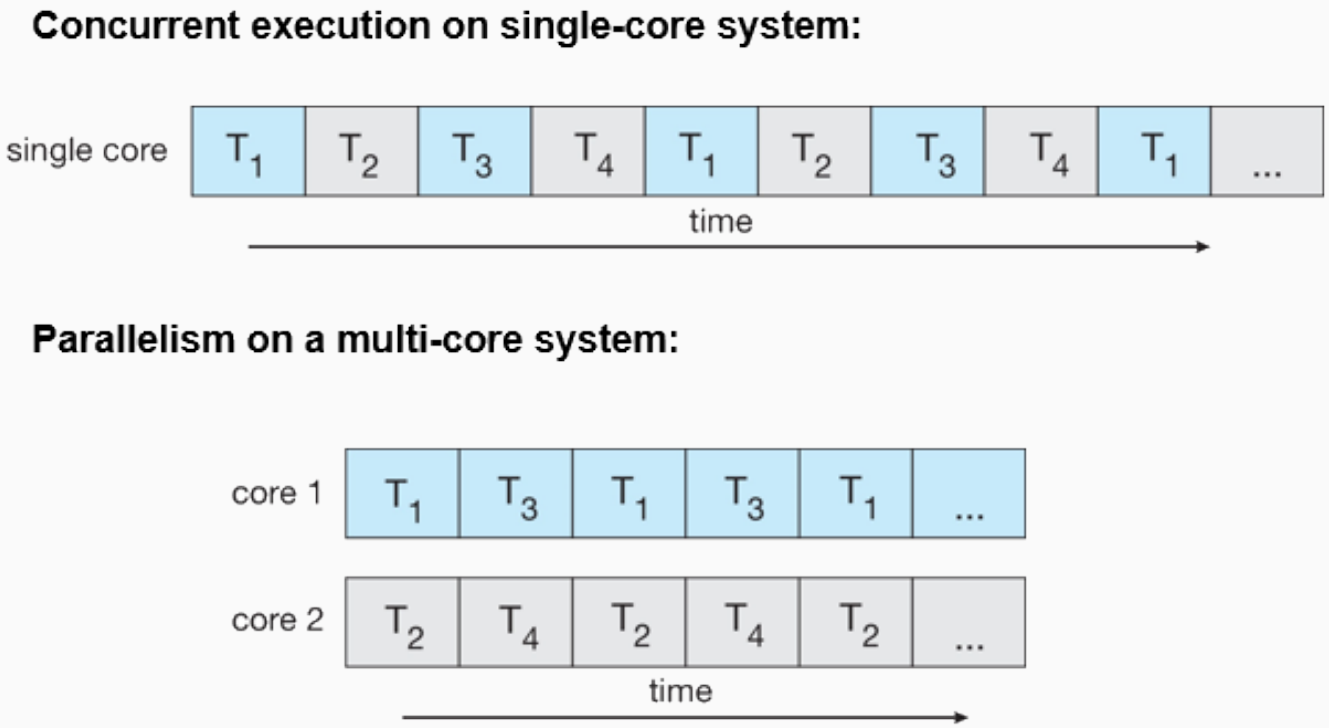


Figure 12: Concurrencia v/s Paralelismo

Aspectos Adicionales - Sistemas Multicore

Existen dos tipos de paralelismos:

- **Datos:** Se distribuyen set's de datos ejecutando la misma operación.
- **Operaciones:** Las hebras se distribuyen en los núcleos. Cada hebra ejecuta operaciones distintas.

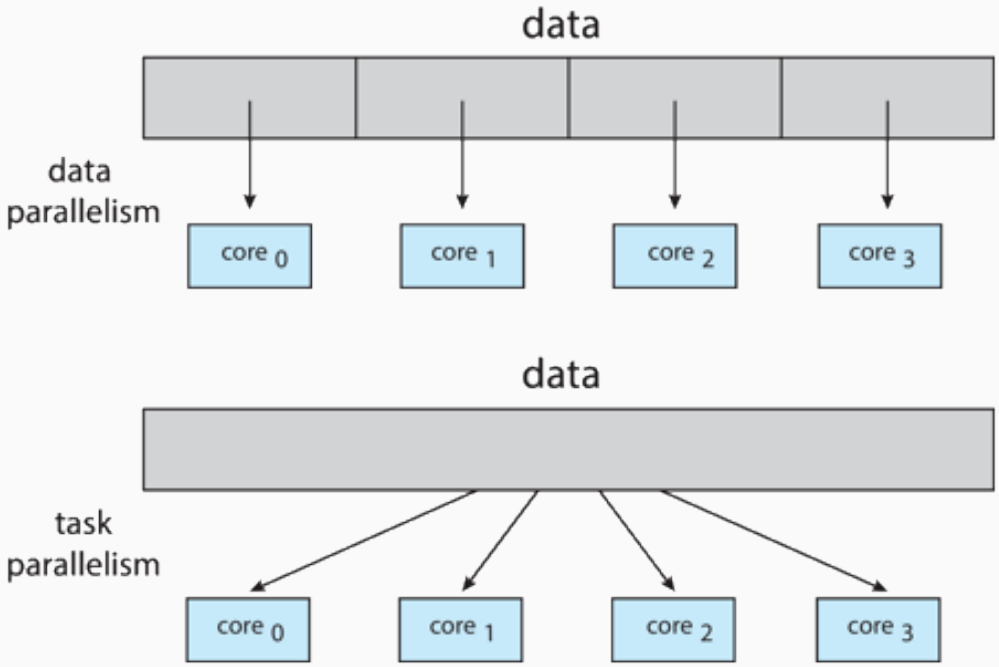


Figure 13: Tipos de paralelismos

Aspectos Adicionales - Issues

¿Qué pasa con fork()? ¿Duplica la hebra que lo invoca? ¿Todas?

- La semántica de fork() y exec() cambia en programas multi hebras.
- Algunos SO's tienen dos versiones para fork().
- Si una hebra invoca exec(), se sustituye todo el programa.

Aspectos Adicionales - Issues

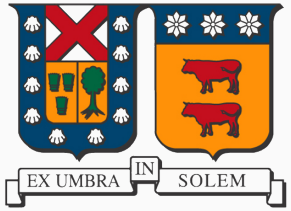
¿Qué pasa con las señales?

- UNIX usa señales para notificar la ocurrencia de un evento particular.
- Para gestionarlas se utiliza un *signal handler*.
- Cuando se genera una, ¿A qué hebra se entrega el mensaje?:
  - ✓ Entregarla a aquellas hebras que aplique.
  - ✓ Entregarla a todas las hebras.
  - ✓ Entregarla a un subconjunto de hebras.
  - ✓ Definir una hebra de gestión para recibir todas.

Aspectos Adicionales - Issues

¿Podemos cancelar una hebra antes de que termine?

- Se denomina hebra objetivo.
- La cancelación puede ser:
  - ✓ **Asíncrona:** Termina la hebra inmediatamente.
  - ✓ **Diferida:** La hebra periodicamente verifica si se debe cancelar.
- En Linux, la cancelación se realiza a través de señales.



Sistemas Operativos

Hebras & Concurrencia

Viktor Andrés Tapia Vásquez

Segundo Semestre 2021

Departamento de Informática, Campus SSJJ.