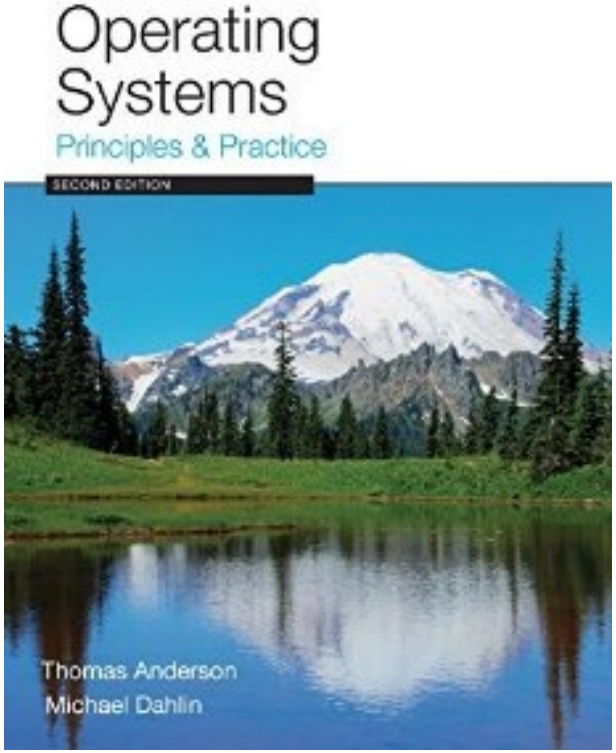
Sistemas Operativos

Capítulo 4 Concurrencia

Prof. Javier Cañas R.

Estos apuntes están tomados en parte del texto: “Operating System: Principles and Practice” de T. Anderson y M. Dahin



Motivación

- Los SO necesitan ser capaces de hacer **M**últiples **C**osas al **M**ismo **T**iempo (MCMT)
  - procesos, servir interrupciones, mantención en background
- Servidores necesitan manejar MCMT
  - Múltiples conexiones con clientes simultáneamente
- Programas paralelos necesitan hacer MCMT
  - Para lograr Speedup
- Programas con GUI necesitan manejar MCMT
  - Para lograr responsividad mientras se hace otra computación
- Programas que usan red y discos necesitam MCMT para ocultar latencias.

Temario

1. Introducción
2. Abstracción de Threads
3. API Simple
4. Implementación

Definiciones

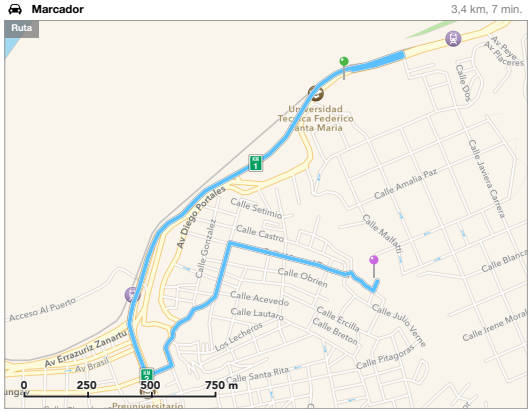
- Un thread es una secuencia simple de ejecución que representa una tarea itinerable separada

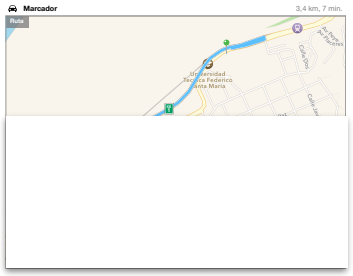
- La protección de threads opera de diversas formas
  - Pueden existir uno o muchos thread por dominio de protección
  - Programa usuario con un thread simple: un thread, un dominio de protección.
  - Programa usuario con múltiples threads: múltiples threads que comparten las mismas estructuras de datos, aislado de otros programas de usuario.
  - Múltiples threads de kernel: múltiples threads que comparten estructuras de datos del kernel, habilitados para ejecutar instrucciones privilegiadas.

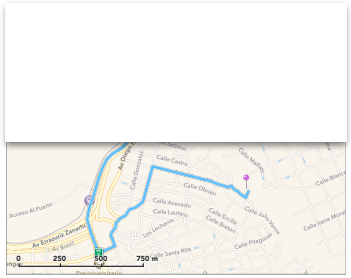
1 El Thread

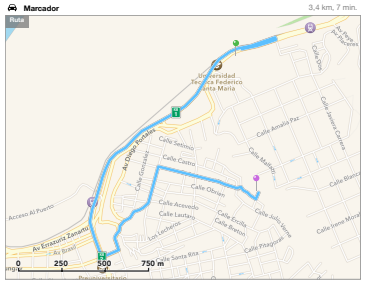
Threads permiten a los programadores definir una secuencia de tareas que podrían ejecutarse concurrentemente, pero nos permiten escribir su código en forma secuencial

En un visualizador de mapas, dos threads dibujan parte de la escena. Un tercer thread maneja la interfaz usuario un cuarto obtiene nuevos datos de un servidor remoto









Data

Thread 1
DrawScene()
Code

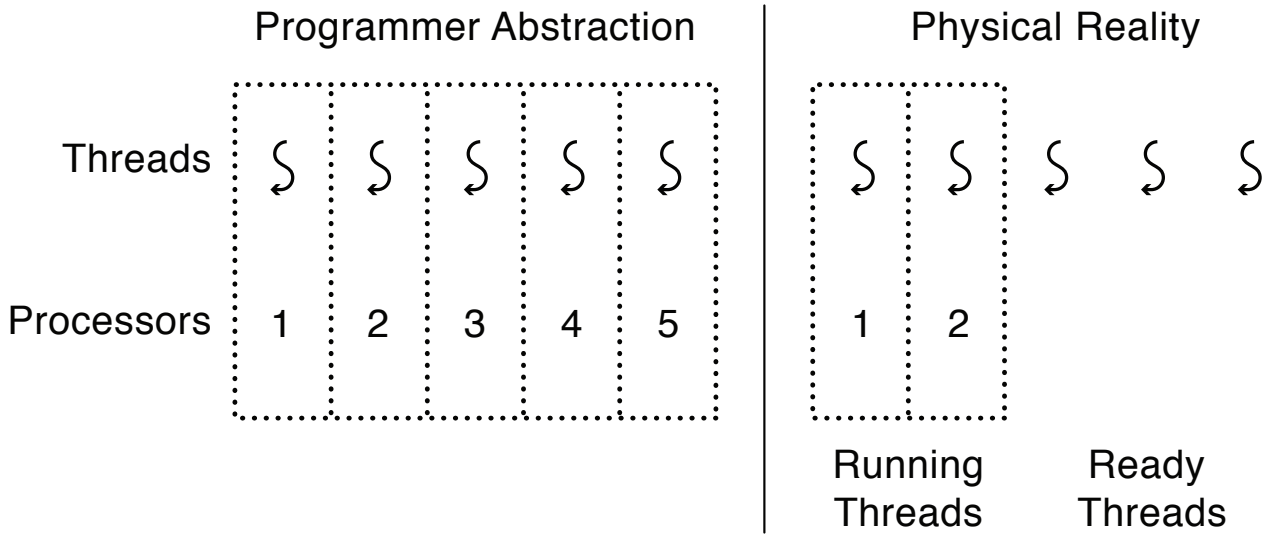
Thread 2
DrawScene()
Code

Thread 3
DrawWidget()
Code

Thread 4
GetData()
Code

2 Abstracción de Threads

- Infinito número de procesadores
- Ejecución de threads con velocidades variables
  - Programas pueden diseñarse para trabajar con cualquier algoritmo de itineración

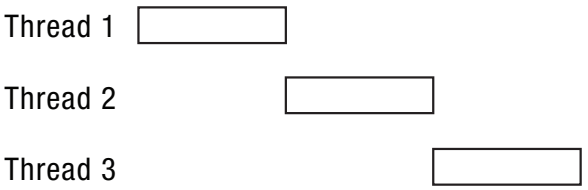


Visión programador y visión del procesador

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	.....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended	.....
.	.	other thread(s) run	Thread is suspended
.	.	thread is resumed	other thread(s) run
.	.	.....	thread is resumed
		y = y + x;	.....
		z = x + 5y;	z = x + 5y;

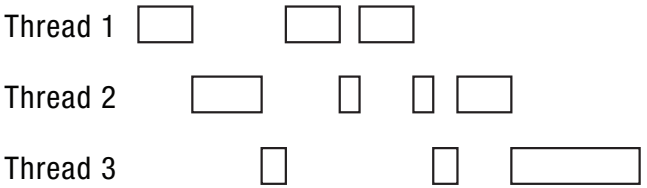
Posibles secuencias de ejecución

One Execution



Another Execution

Another Execution



3 API Simple Thread

- Sthreads está basada en Posix, pero simplifica su uso omitiendo opciones y manejo de errores. La mayoría de los paquetes de manejo de threads son similares.

- Necesita sthread.c sthread.h que están disponibles en el Moodle y también:

http://www.cs.utexas.edu/users/dahlin/osbook/code/

Operaciones sobre SThreads

- `sthread_create(thread, func, args)`
  - Crear un nuevo thread para correr la función func(args)
- `sthread_yield()`
  - Entrega el procesador voluntariamente
- `sthread_join(thread)`
  - El padre, espera que el thread termine, entonces return
- `sthread_exit`
  - Abandona el thread y limpia estructura y dispierta procesos si los hay

… API Simple Thread

```
void sthread_create(thread, func, arg)
```

- Crea un nuevo thread, almacenando la información sobre él en **thread**. El thread ejecuta la función **func**, la cual se llamará con el argumento **arg**.

… API Simple Thread

```
void sthread_yield()
```

- El thread que invoca esta función voluntariamente abandona el procesador para dejar que corra otro thread. El Scheduler puede continuar con el thread cuando le parezca.

… API Simple Thread

```
int sthread_join(thread)
```

- Espera que un thread específico termine. Retorna el valor de **`sthread_exit(valor)`**. Notar que sólo se puede llamar una vez para cada thread.

… API Simple Thread

```
void sthread_exit(ret)
```

- Termina el thread que la invoca. Almacena el valor de ret en la estructura de dato del thread. Si un thread está esperando en un join, lo despierta.

Ejemplo

- El siguiente ejemplo muestra el programa threadHola.C que utiliza 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
```

```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
}
```

```
#define NTHREADS 10
thread_t threads[NTHREADS];

for (i = 0; i < NTHREADS; i++)
    thread_create(&(threads[i]), &go, i);
for(i = 0; i < NTHREADS; i++){
    exitValue = thread_join(threads[i]);
    printf("Thread %d returned with %ld\n", i, exitValue);
  }
printf("Main thread done.\n");
```

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

Salida `threadHello`

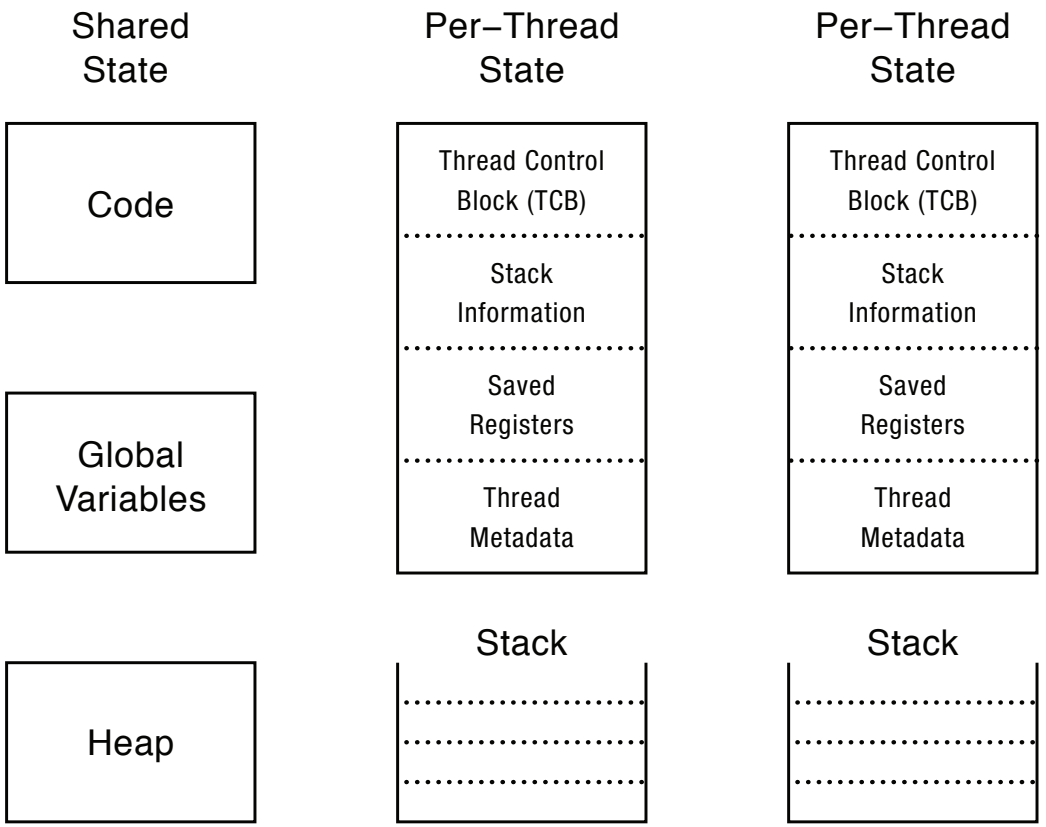
- ¿Qué otra salida es posible?
- ¿Cuál es el máximo número de threads corriendo al mismo tiempo?
- ¿El mínimo?

```
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.
```

4 Implementación

- El OS entrega la ilusión que cada thread corre en su propio procesador virtual. Esto lo hace transparentemente suspendiendo y reasumiendo threads.
- Un thread necesita estados internos que deben mantenerse en el kernel.
- La estructura de datos se denomina TCB Thread Control Block.
- La TCB mantiene dos tipos de información
  - El estado de la computación
  - Metadata sobre el thread

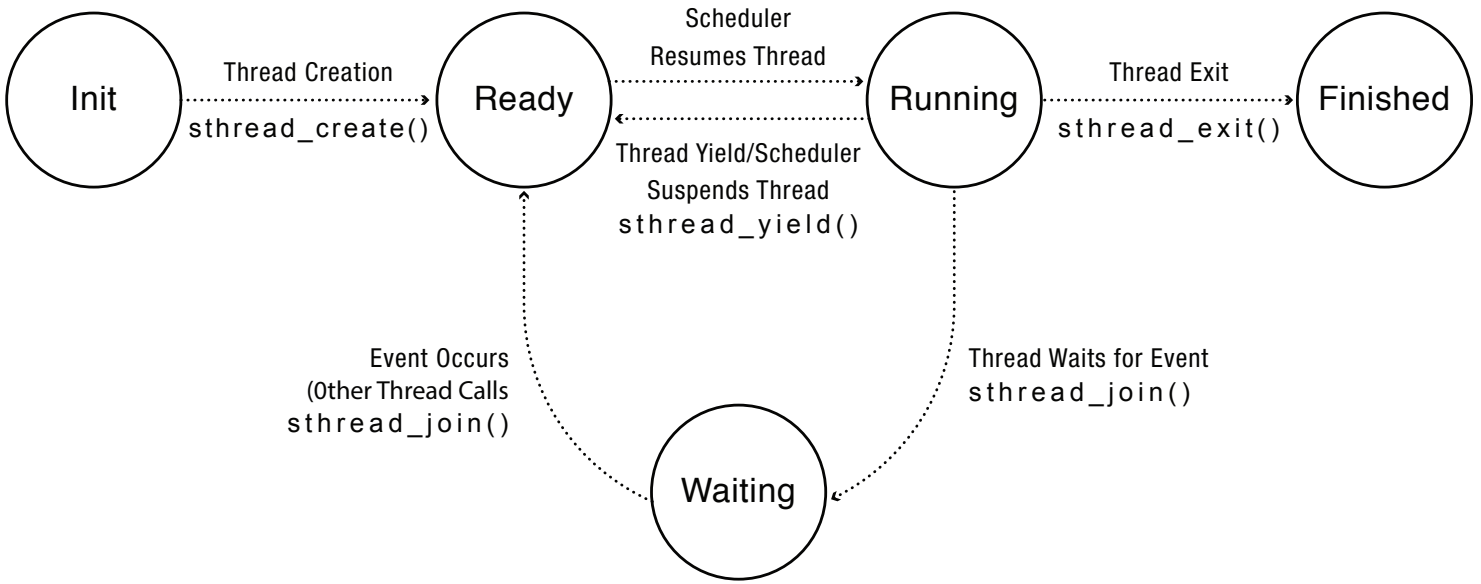
Estructura de datos



Estados del Thread

- Cada thread contiene dos elementos de estado que expresan su estado:
  - Stack: Almacena información para procedimientos y funciones anidadas del thread que está coriiendo. El stack almacena los "*stack frames*" de cada procedimiento. Cada stack frame contiene las variables locales del procedimiento, parámetros y direcciones de retorno.
  - Copia de los registros (register file) del procesador.
  - Metadata: Por ejemplo el tid, prioridad

Diagrama de Estados del Ciclo de Vida



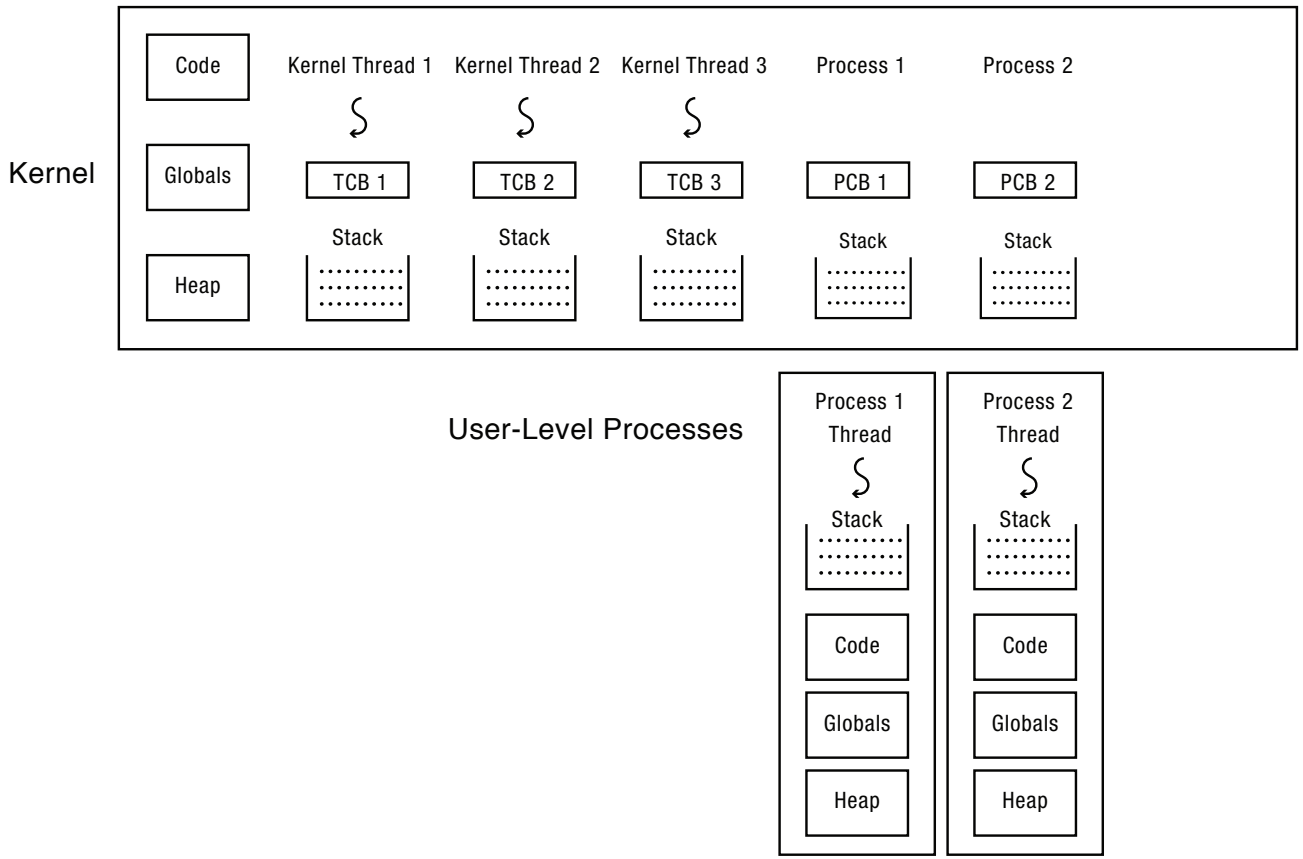
Ubicación de TCB y registros en distintos estados

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

Alternativas de implementación

- Sólo threads de kernel:
  - La abstracción de thread, sólo disponible para el kernel
  - Para el kernel, se ven en forma similar un proceso usuario con un solo thread que un thread del kernel.
- Procesos multi thread que usan threads del kernel (Linux, MacOSX)
  - Operaciones sobre threads disponibles vía llamadas al sistema
- Threads a nivel usuario
  - Operaciones sobre threads sin llamadas al sistema

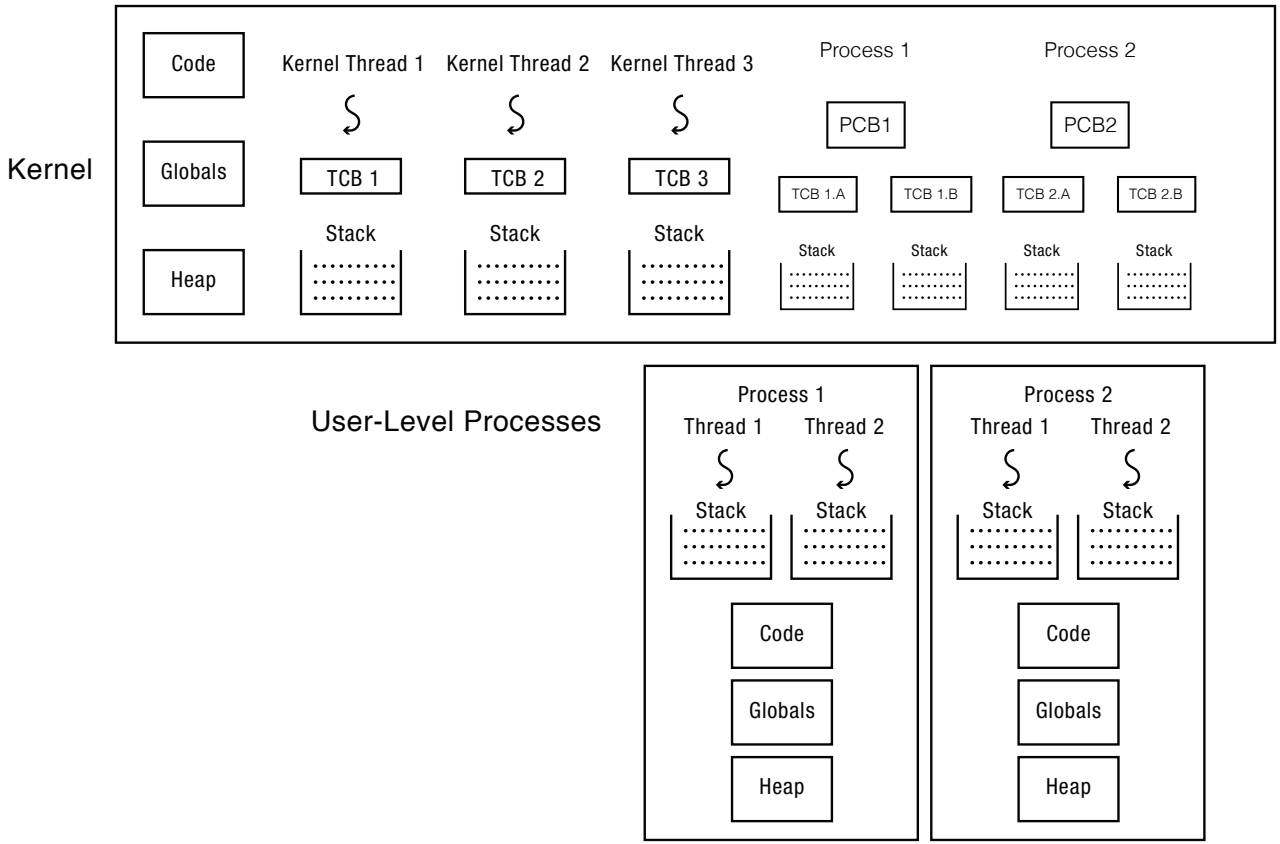
Kernel Multi-thread con 3 thread y dos procesos de un thread



Análisis

- Se observa que cada proceso incluye el thread del proceso.
- Un proceso es más que un thread, porque cada proceso tiene su propio espacio de direcciones. El Proceso1 tiene su propia vista de memoria, su código, su heap, y sus variables globales que difieren del proceso 2.
- El PCB necesita mayor información que el TCB.
- Como en este caso tanto PCB y TCB representan un thread, la Ready List del kernel contiene una mezcla de PCB para procesos y TCB para threads del kernel.

Kernel Multi-thread con 3 thread y dos procesos cada uno con dos thread



Análisis

- La Ready List del kernel incluye los TCB de los threads del kernel y uno o más PCB para cada proceso de usuario.

- El mecanismo de context switch permite conmutar entre:
  - kernel threads
  - kernel threads y threads de procesos
  - threads de diferentes procesos
  - threads de un mismo proceso

- ¿Por qué un proceso necesita crear múltiples threads)
  - Estructurar programas concurrentes
  - Explotar procesadores multicore

Sistemas Operativos

Capítulo 4 Concurrencia

Prof. Javier Cañas R.