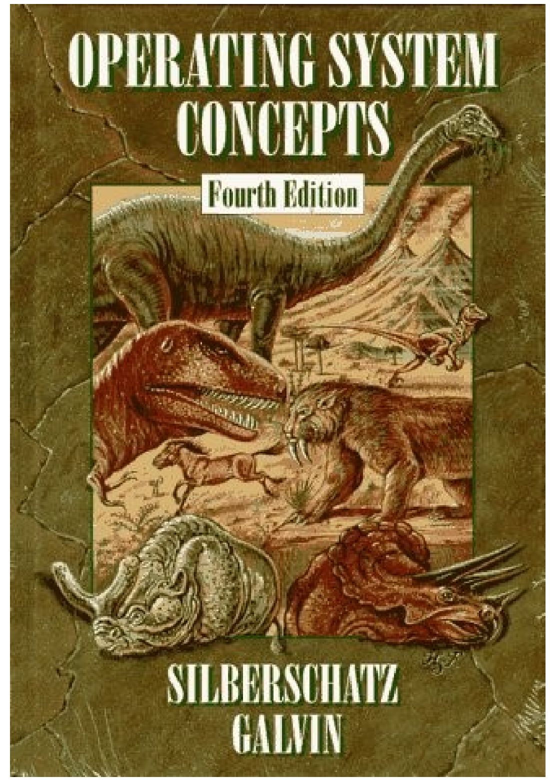
Chapter 4: Multithreaded Programming

School of Computing, Gachon Univ.
Joon Yoo



Gachon University
School of Computing

Chapter 4: Multithreaded Programming

- Overview
- Benefits of Multithreading
- Multithreading Models
- Thread Libraries
- Process vs. Thread

Objectives

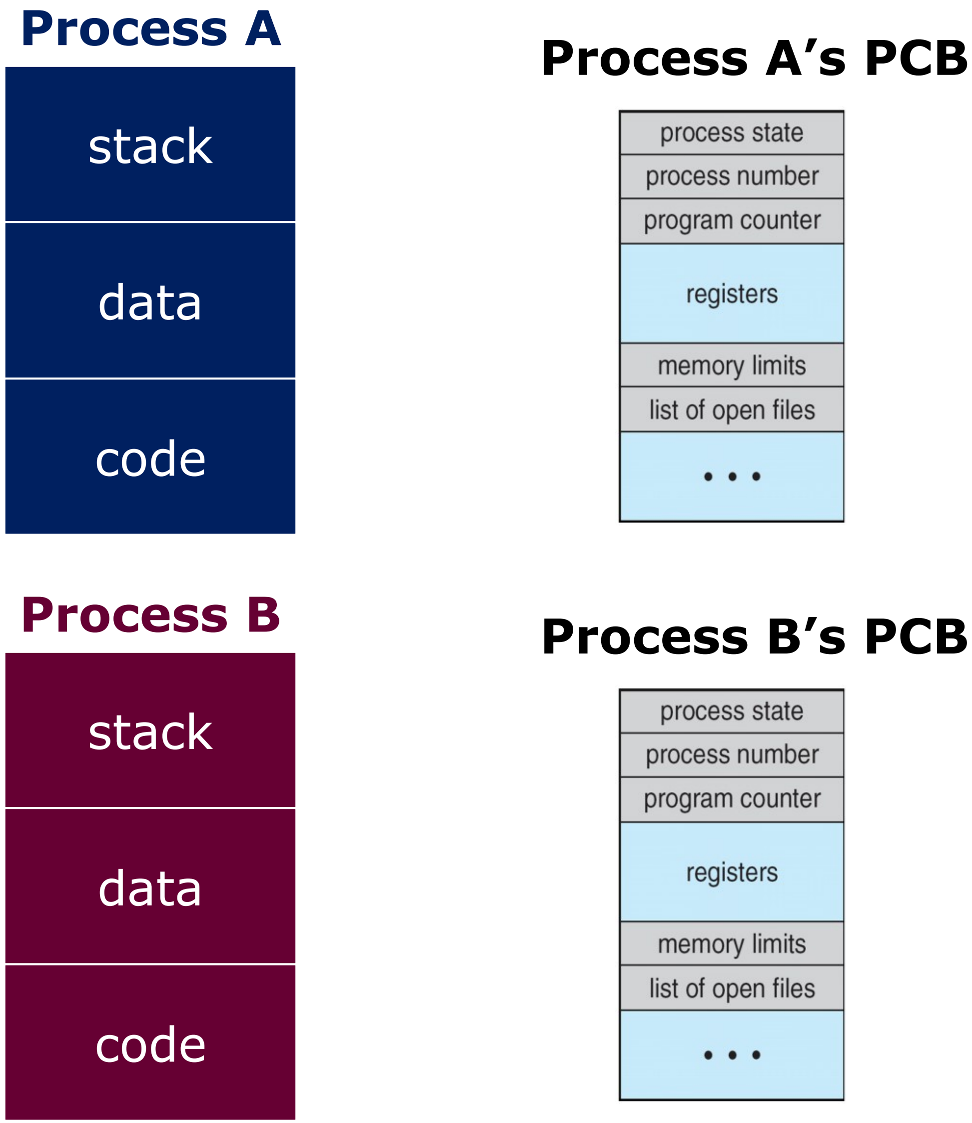
- To introduce the notion of a **thread**—a fundamental unit of **CPU utilization** that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads and Java thread libraries

Multi-Process (Multi-tasking)

- User opens the same program two times
  - e.g., opens two web browsers, opens two Word files, Web server executes similar tasks for each user
- The two programs will be
  - executing **same code**,
  - may want to **share data**

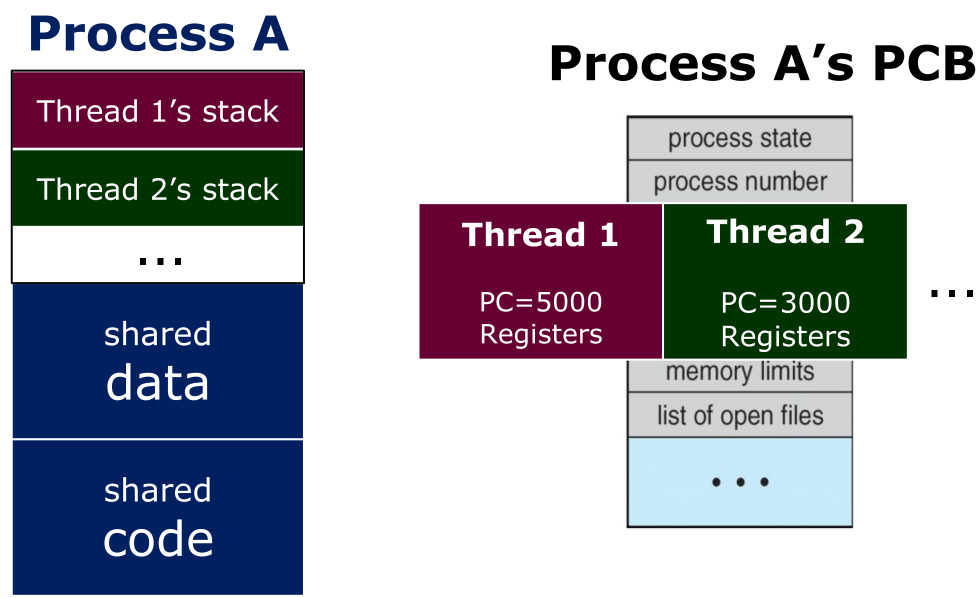
Multi-Process (Multi-tasking)

- Processes are not very efficient
  - Each process has its own PCB and OS resources
- Processes don't (directly) share memory
  - Each process has its own address space
  - Need IPC (shared memory, message passing)
- Can make it more efficient by **sharing**?



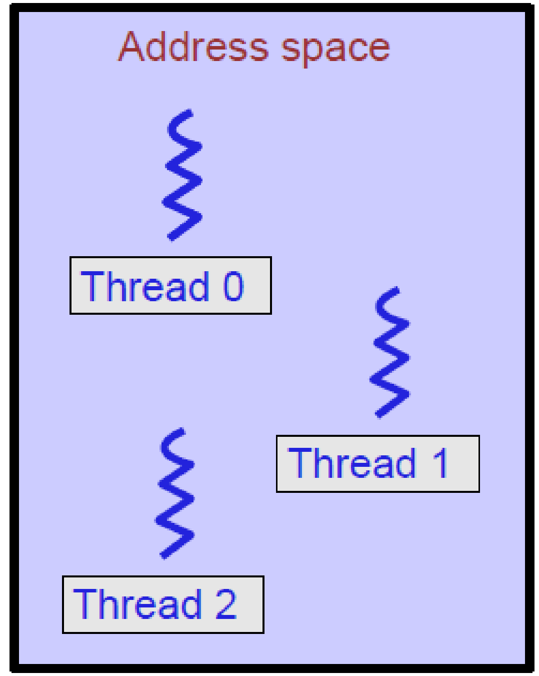
What can we do? Let us share...

- What can we **share** across all of these processes?
  - Same code: generally running the same or similar programs
  - Same data
- What is private to each process? (i.e., what can we **not share**?)
  - Execution context: CPU registers, stack, and program counter (PC)



Processes and Threads

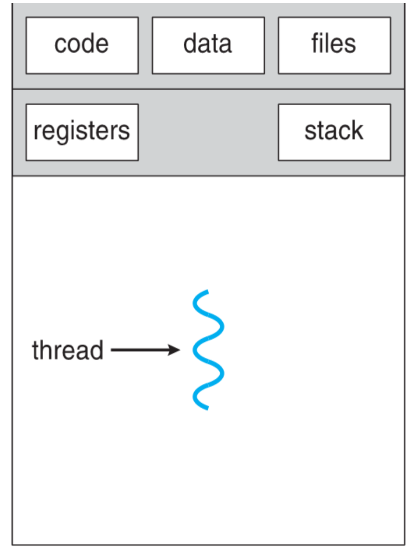
- **Thread**?
  - A thread (or lightweight process) is a basic unit of **CPU scheduling**
  - A process is just a “container” for its threads
  - Each thread is bound to its containing process
- Each thread has its own (**not share**)
  - **stack, CPU registers, PC**
- All threads within a process **share** memory space
  - **text, data, and OS resources**
  - **Threads in same process** can communicate directly via shared memory (no need for IPC)



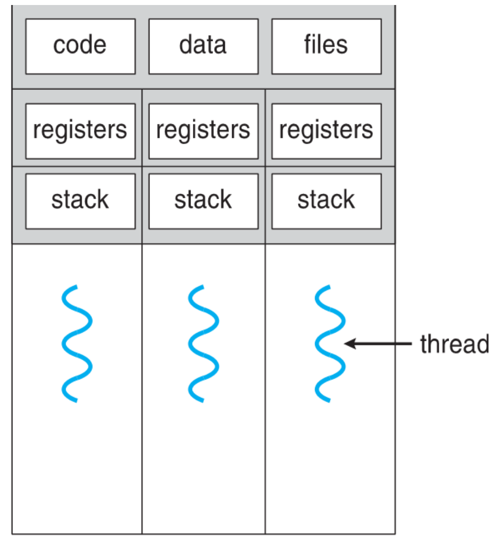
Process

Single-threaded vs. Multi-threaded Process

- Simple programs can have one thread per process
  - ***single-threaded*** *process*
- Complex programs can have multiple threads
  - ***multi-threaded*** *process*
  - Multiple threads running in same process's address space



single-threaded process



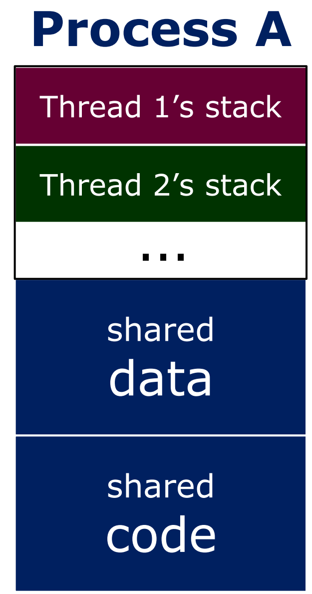
multithreaded process

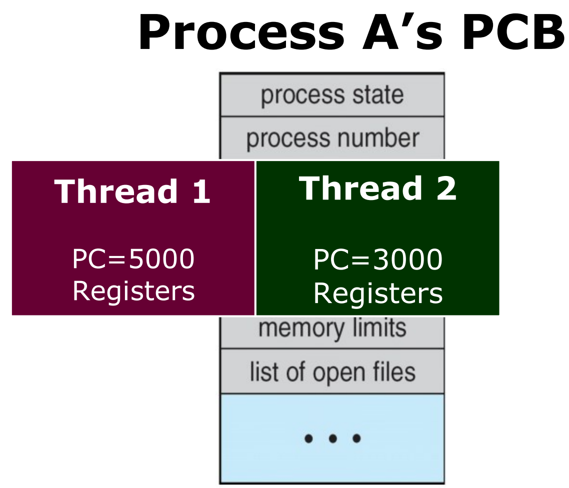
Chapter 4: Multithreaded Programming

- Overview
- Benefits of Multithreading
- Multithreading Models
- Thread Libraries
- Process vs. Thread

Benefits of Multithread

- **Resource Sharing**
  - All threads in one process share memory resources (code, data) of process
  - easier to share than IPC
- **Lighter weight**
  - lighter weight than process
    - creation/deletion, context-switching faster
  - Easier to create/delete 10 threads than 10 processes

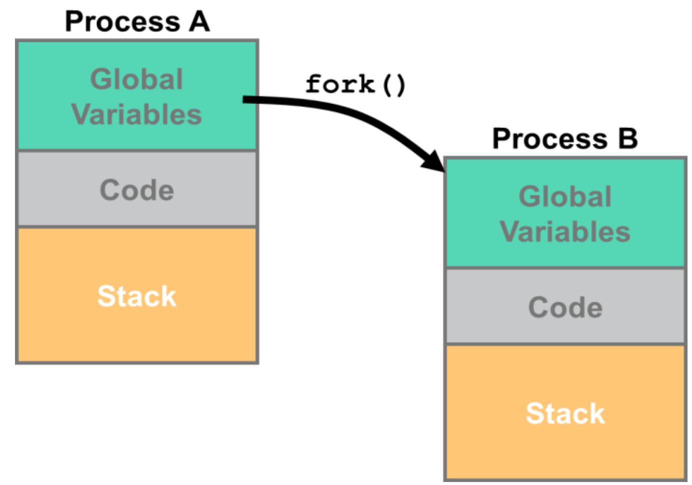




Thread vs. Process creation

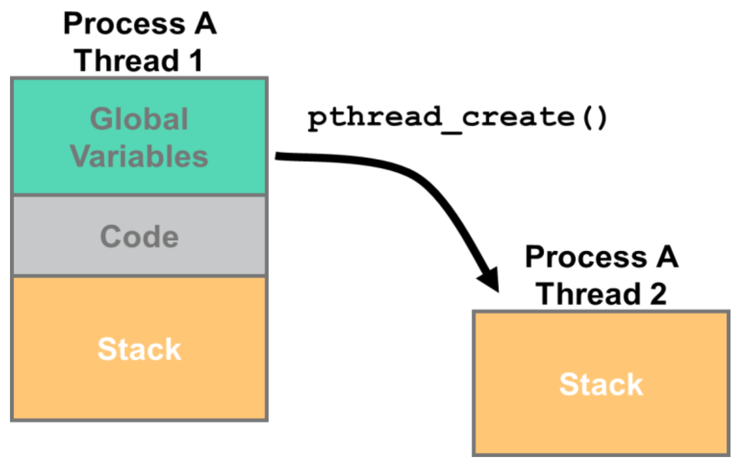
Creation of a new process using *fork()* is *expensive* (time & memory).

Need new PCB



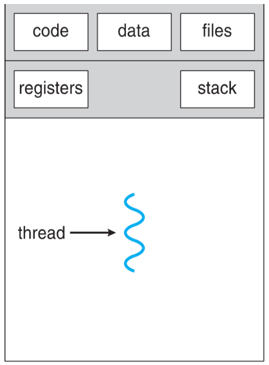
A thread creation using pthread_create() does ***not*** require a lot of memory or startup time.

No need for new PCB

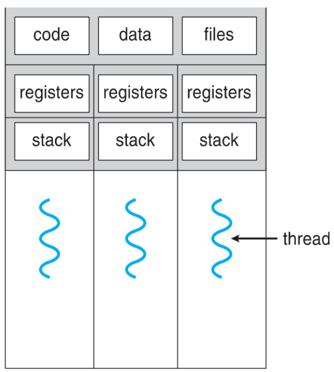


Benefits of Multithread

- **Non-blocking System Call (Responsiveness)**
  - may allow continued execution if part of **process** is blocked
    - why blocked? – time consuming operation (e.g., I/O such as printing, network)
    - single-threaded process: <u>**wait**</u> until I/O operation is complete
    - multithreaded process: one thread must wait, but another thread in same process can continue



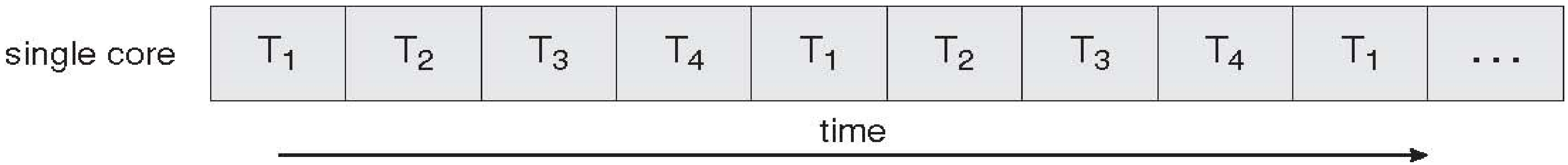
single-threaded process



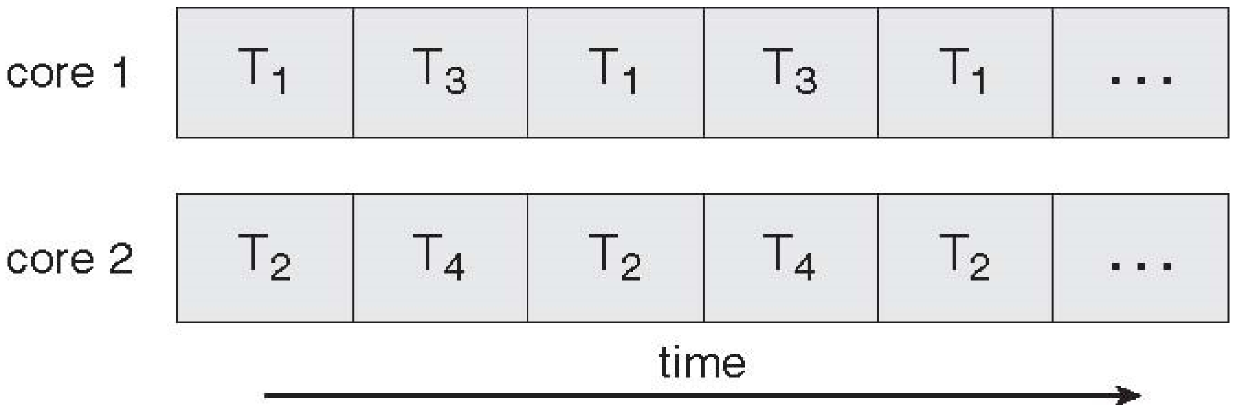
multithreaded process

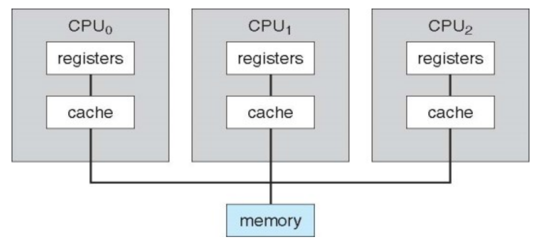
Benefits of Multithreading: Multicore Programming

- ***Concurrent*** Execution on a Single-core System



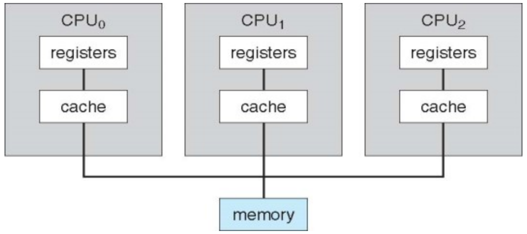
- ***Parallel*** Execution on a Multicore System





Multicore Programming

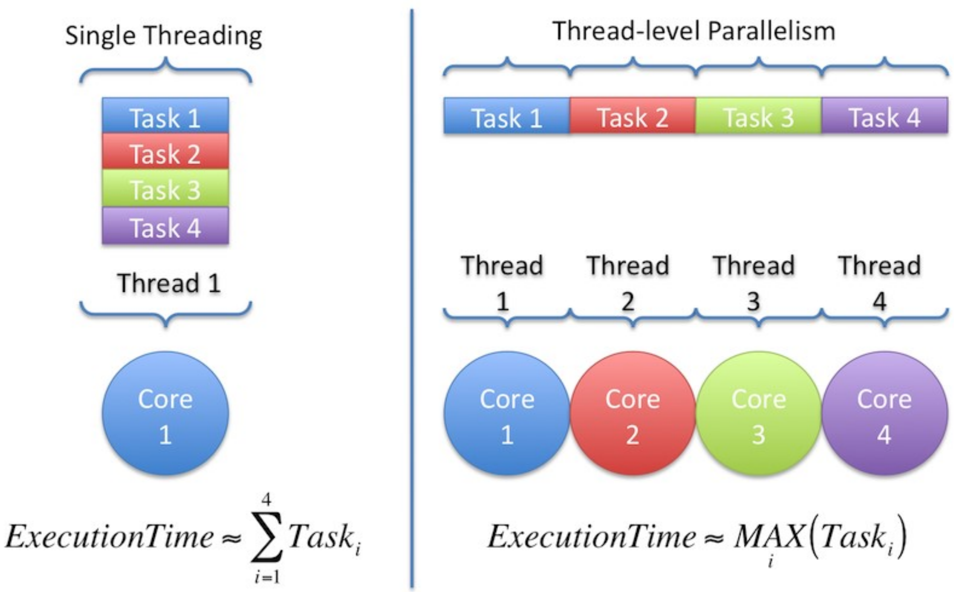
- Multicore Programming provides parallelism!
  - ***Concurrency*** supports more than one task making progress – *Single-core* processor can provide concurrency
  - ***Parallelism*** implies a system can perform more than one task simultaneously – Need *Multi-core* processor



- By using multithreading, **one** process can use multiple cores!
  - Q: Multithreading gives (concurrency/parallelism) to a single process

Multicore Programming

- **Allows one process to use multiple cores**
  - A multithreaded process can take advantage of Multicore CPU architectures
  - A process can run many threads **in parallel** on different processor **cores**



- Task: Add 1 to 4,000,000
  - Divide into 4 subtasks
    - Task1: add 1 to 1,000,000
    - Task2: add 1,00,001 to 2,000,000
    - Task3: add 2,00,001 to 3,000,000
    - Task4: add 3,00,001 to 4,000,000
- Say a core takes 10 seconds to add 1 million numbers
- Single threading: Give all 4 tasks to 1 core
  - Takes __________ seconds
- Multithreading: Give one task to each core (core 1~4)
  - Takes __________ seconds

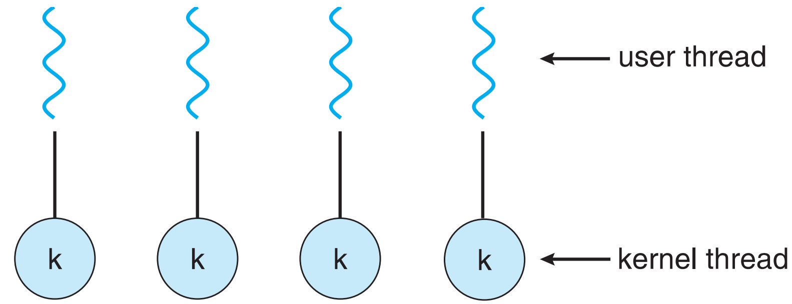
Gachon University
School of Computing

Chapter 4: Multithreaded Programming

- Overview
- Benefits of Multithreading
- Multithreading Models
- Thread Libraries
- Process vs. Thread

Kernel Threads

- **Kernel threads**
  - Threads supported by the **Kernel**
    - created by `thread_create` system call
    - each thread needs **thread control block (TCB)**
  - **Pros**: kernel knows the thread, so…
    - **Parallelism**: Can run multiple threads on **multi-core**
    - **Concurrency**: another thread can run when one thread makes blocking system call (e.g., I/O request)
  - **Cons**: kernel knows the thread, so…
    - every thread operation must go through kernel; heavy weight
    - Syscall 10x-30x slower than user threads



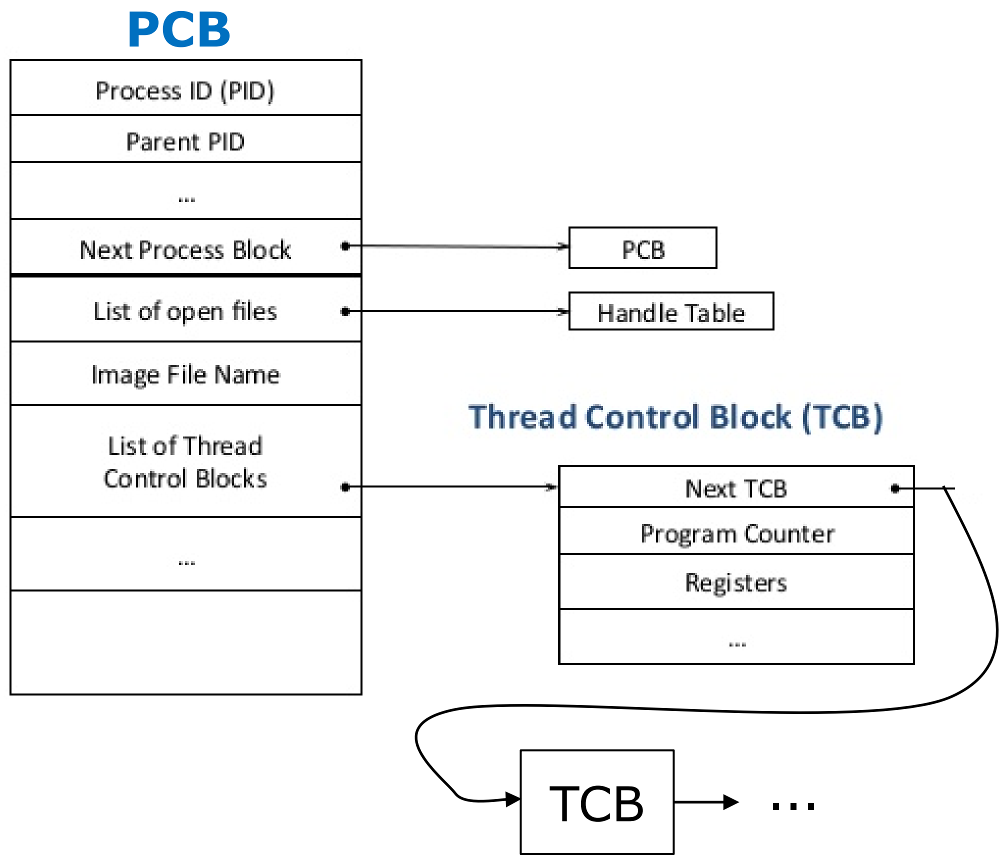
Control Blocks

Thread Control Block (TCB)

- Created for each kernel thread
- Contains Program Counter (PC), and registers
- Other resources shared

Ready queue is now a list of **TCB**s waiting for CPU resource

CPU Context switching is done for **Threads**



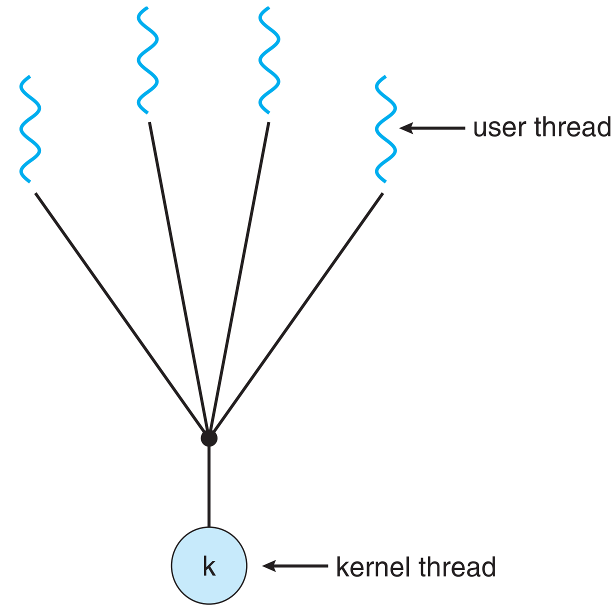
User thread

- **User Thread (=green threads)**
  - Implement thread in **<u>user library</u>**
  - created/managed *without* kernel support: **<u>no need for system call</u>**
  - One kernel thread per process, many user threads mapped to single kernel thread

  Pros: kernel doesn't know the user thread, so…

  Thread management is done by the thread library in <u>user space</u>

  Fast and efficient (10x-30x faster than kernel thread) – no syscall



- **Cons**: kernel doesn't know the user thread, so…
  - A thread makes a system call - one thread blocking causes all threads in process to **block**
  - Multiple user-level threads may ***not*** run in parallel on multicore system

why?

Chapter 4: Multithreaded Programming

- Overview
- Benefits of Multithreading
- Multithreading Models
- Thread Libraries (Pthread)
- Process vs. Thread

Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Common in UNIX operating systems (Linux & Mac OS)

Multithreaded C program using the Pthreads API

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

int sum;        /* this data is shared by the thread(s) */
void *runner( void * param ); /* the thread */

int main( int argc, char * argv[] )
{
   pthread_t tid;              /* the thread identifier */
   pthread_attr_t attr;     /* set of thread attributes */


   /* set the default attributes */
   pthread_attr_init( &attr );
   /* create the thread */
   pthread_create( &tid, &attr, runner, argv[1] );
   /* wait for the thread to exit */
   pthread_join( tid, NULL );

   printf( "sum = %d\n", sum );
}
```

② ① ③

$$sum = \sum_{i=0}^{N} i$$

Separate Thread does this..

① Thread Creation

Unique thread identifier (ID)
returned from call

Attributes structure
(NULL for defaults)

```
int pthread_create(&tid, &attr, runner, argv[1] );
```

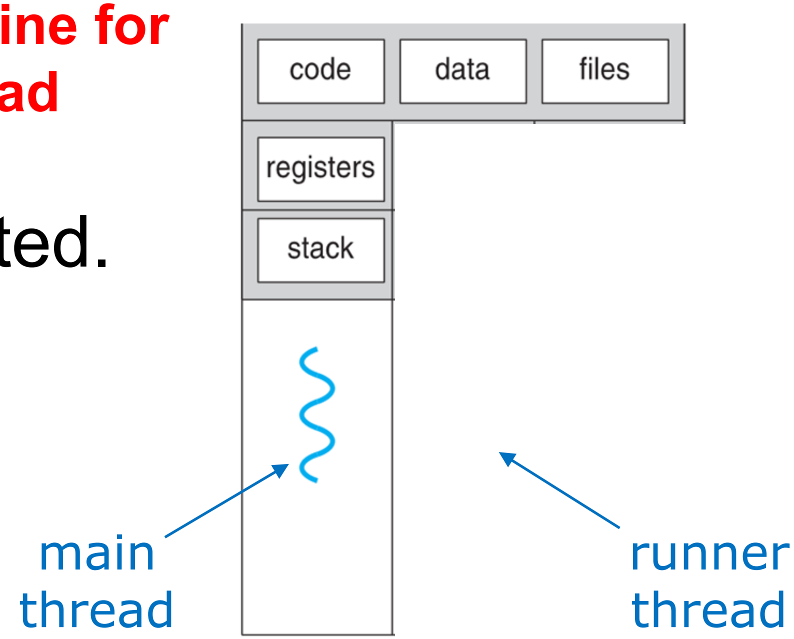
zero for success,
else error number

main routine for child thread

Argument passed

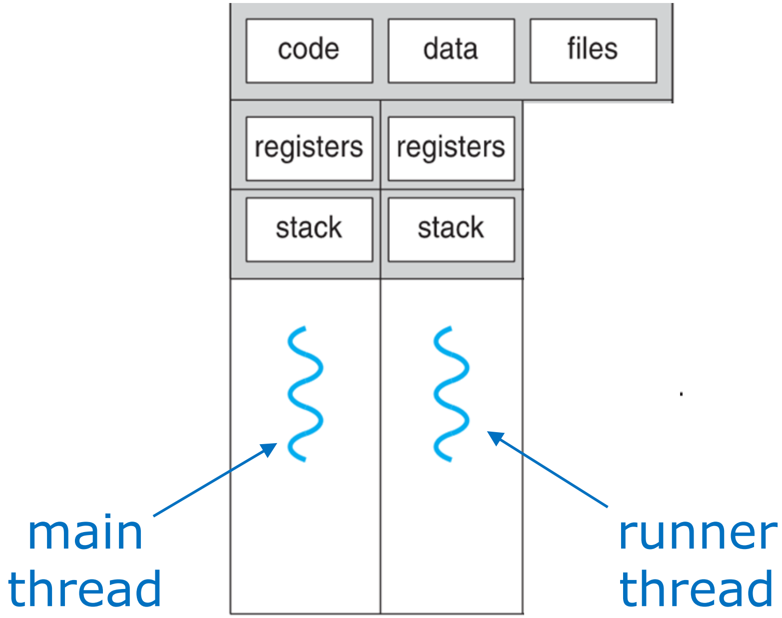
`func` is the function to be called.

When `func()` returns, the thread is terminated.



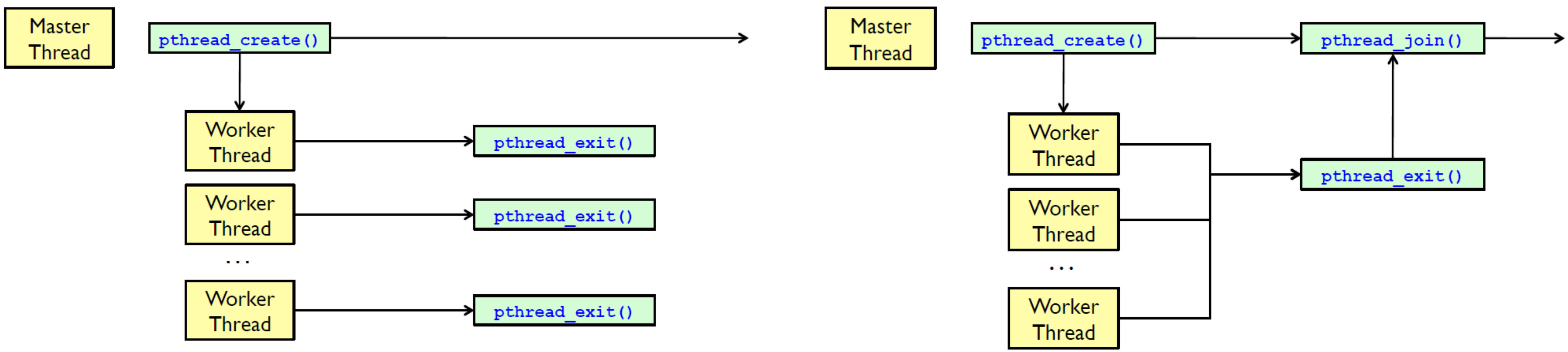
② Thread Function

```
/* The thread will begin control in this function */
void *runner( void * param )
{
   int i, upper = atoi( param );
   sum = 0;

   for( i = 1; i <= upper; i++ )
       sum += i;

   pthread_exit( 0 );
}
```



③ pthread_join()

```
pthread_create( &tid, &attr, runner, argv[1] );
/* wait for the thread to exit */
pthread_join( tid, NULL );
```



Suspends parent thread until child thread terminates

similar to ____________ system call in process

Pthreads Code for Joining 10 Threads

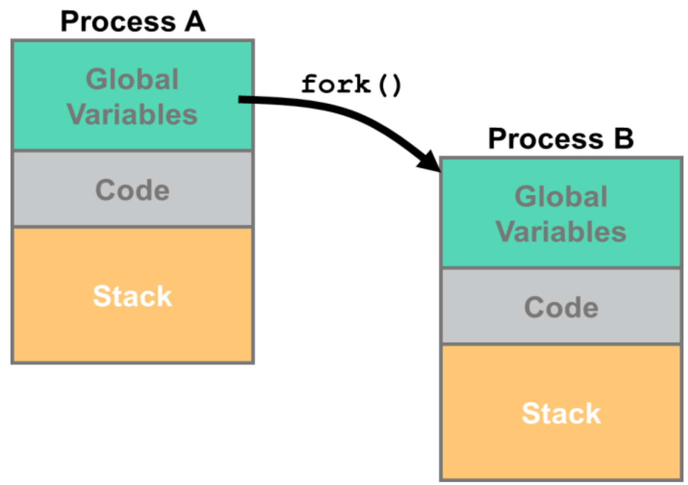
```
#define NUM THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];
...
for (int i = 0; i < NUM_THREADS; i++)
  pthread_join(workers[i], NULL);
```

Chapter 4: Multithreaded Programming

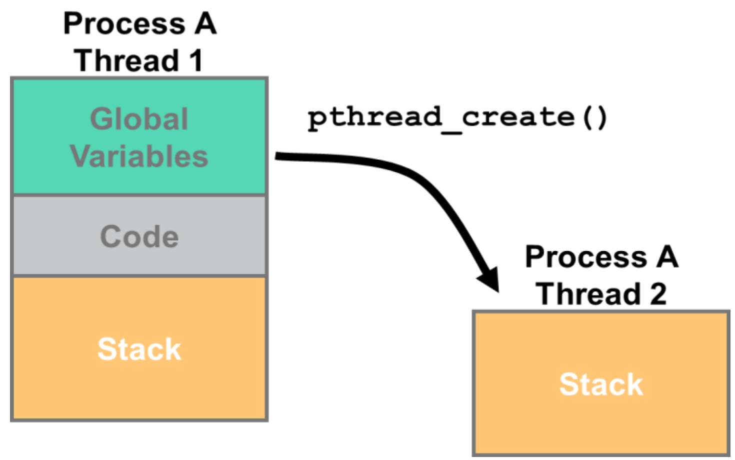
- Overview
- Benefits of Multithreading
- Multithreading Models
- Thread Libraries
- Process vs. Thread

Thread vs. Process Creation

- `fork()`
  - Two separate processes
  - Child process starts from same position as parent (clone)
  - Independent memory space for each process



- `pthread_create()`
  - Two separate threads
  - Child thread starts from a function
  - Share memory



Process vs. Thread Example

- Shared code:

```
int x = 1; //global variable
void* func(void* p){
   x = x + 1;
   printf("x is %d\n”, x);
   return NULL;
}
```

- fork version:

```
main(…) {
     fork();
     func(NULL);
}
```

- threads version:

```
main(…) {
   pthread_t tid;

pthread_create(&tid,NULL,func,NULL);
   func(NULL);
}
```

Possible output: fork() case 1

```
int x = 1; //global variable
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n”, x);
   return NULL;
}
```

```
int x = 1; //global variable
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n”, x);
   return NULL;
}
```

time

Parent process **Child process**

Possible output: fork() case 2

```
int x = 1; //global variable
```

```
int x = 1; //global variable
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```

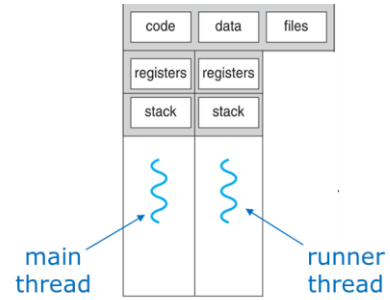
time

Parent process **Child process**

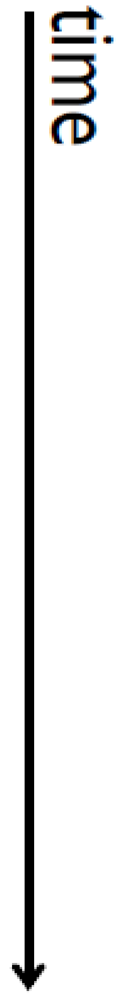
Possible output: threads case 1

```
int x = 1; //global variable
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```




```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```



Parent thread **Child thread**

Possible output: threads case 2

```
int x = 1; //global variable
```

Parent thread

```
void* func(void* p){
   x = x + 1;




   printf("x is %d\n", x);
   return NULL;
}
```

Child thread

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```

time

Possible output: threads case 3

```
int x = 1; //global variable
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);

   // interrupted during printf()

   printf("x is %d\n", x);

   return NULL;
}
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```

time

Parent thread

Child thread

Possible output: threads case 4

Output:

```
x is 2
x is 2
```

- Is it a possible output for this example ??
  - Hint: translate x = x + 1 into assembly instructions
    - lw $t0, 0($gp)
    - addi $t0, $t0, 1
    - sw $t0, 0($gp)

$t0: data register
$gp: memory address of x
lw: load word (from memory)
sw: store word (to memory)

- Bottom line: We cannot predict the results!
  - We need process (thread) synchronization (Ch. 6)

Up Next

- Which thread gets to go next when a thread exits running state?
  - Scheduling Algorithm (Ch. 5)

- What happens when multiple threads want to use the shared resource?
  - Synchronization (Ch. 6)

https://stock.adobe.com/kr/search?k=q%26a