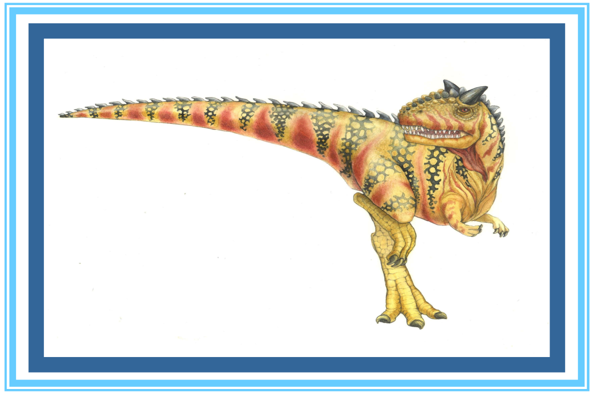
Chapter 4: Multithreaded Programming

School of Computing, Gachon Univ.
Jungchan Cho

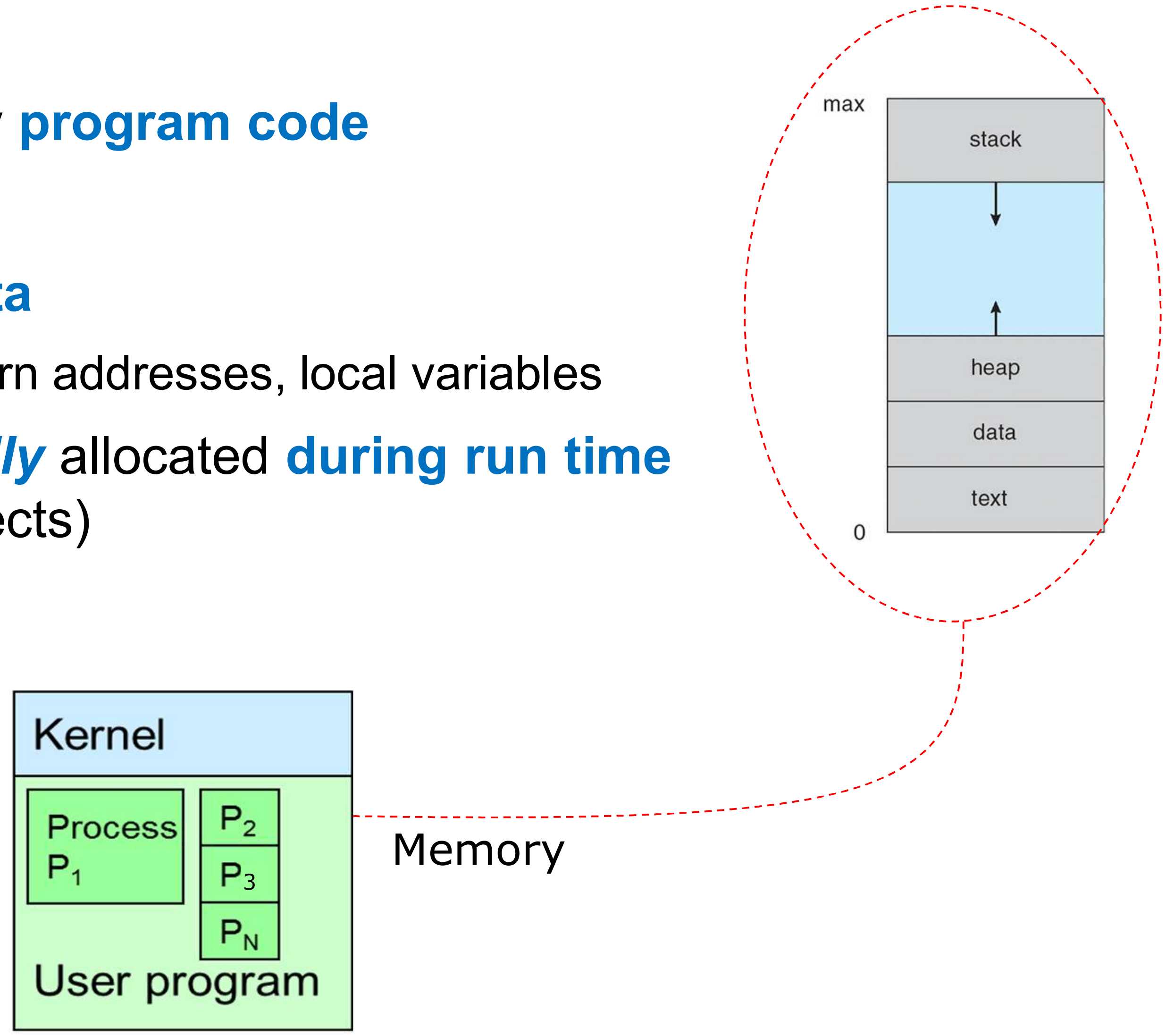


Most slides from "Operating System Concepts – 10th Edition".
Many slides are taken from lecture notes of Prof. Joon Yoo.

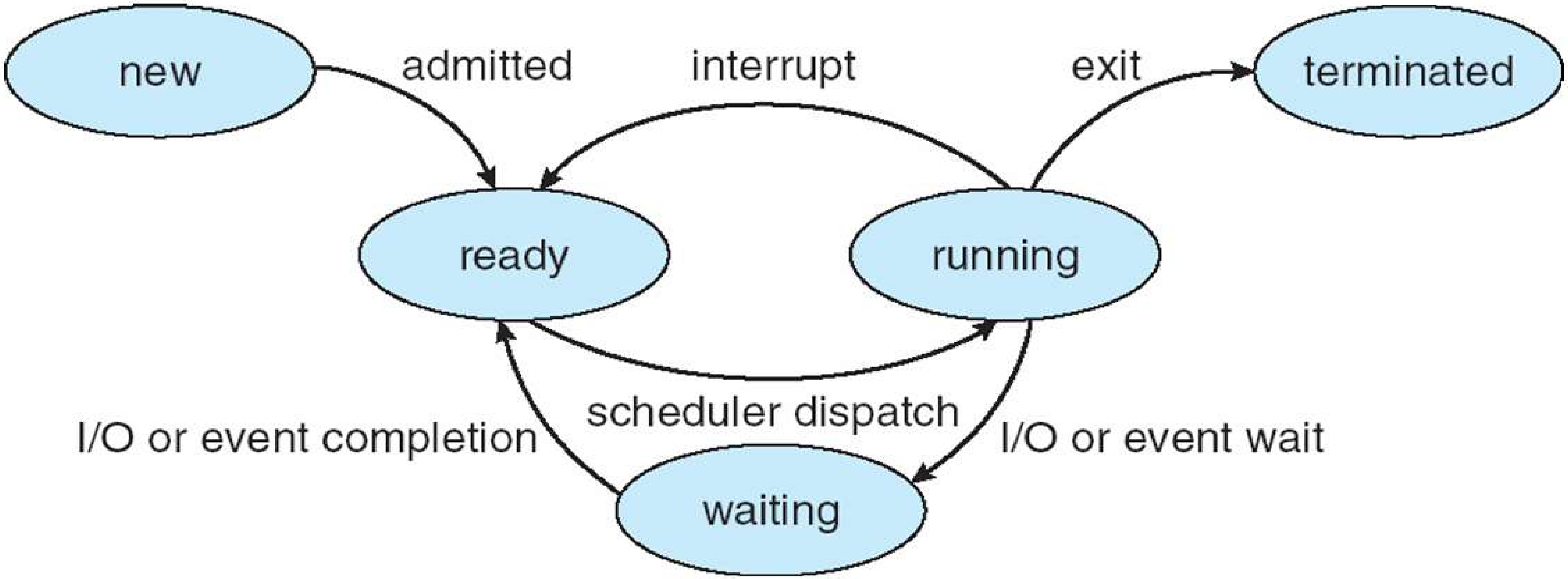
미래창조과학부 SW중심대학
가천대학교 소프트웨어학과
Gachon University Department of Software

review

- Process in memory
  - **Text**: The binary assembly **program code**
  - **Data**: global **variables**
  - **Stack**: temporary local **data**
    - Function parameters, return addresses, local variables
  - **Heap**: memory ***dynamically*** allocated **during run time** (e.g., C malloc(), Java objects)

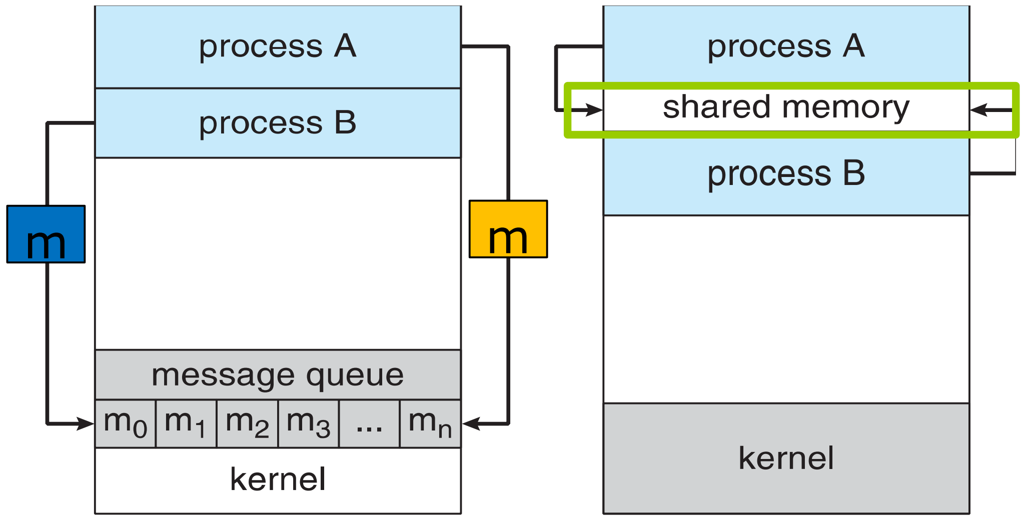


review


new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Communications Models

- Mechanism
  - for processes to **communicate** & to **synchronize** their actions
- Two fundamental models of IPC
  - **Shared memory**
  - **Message passing**



(a) Message passing (b): Shared memory

Chapter 4: Multithreaded Programming

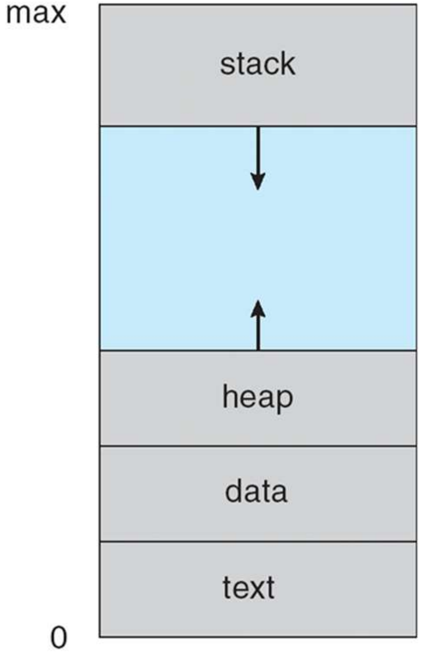
- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Process vs. Thread

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, Windows, and Java thread libraries

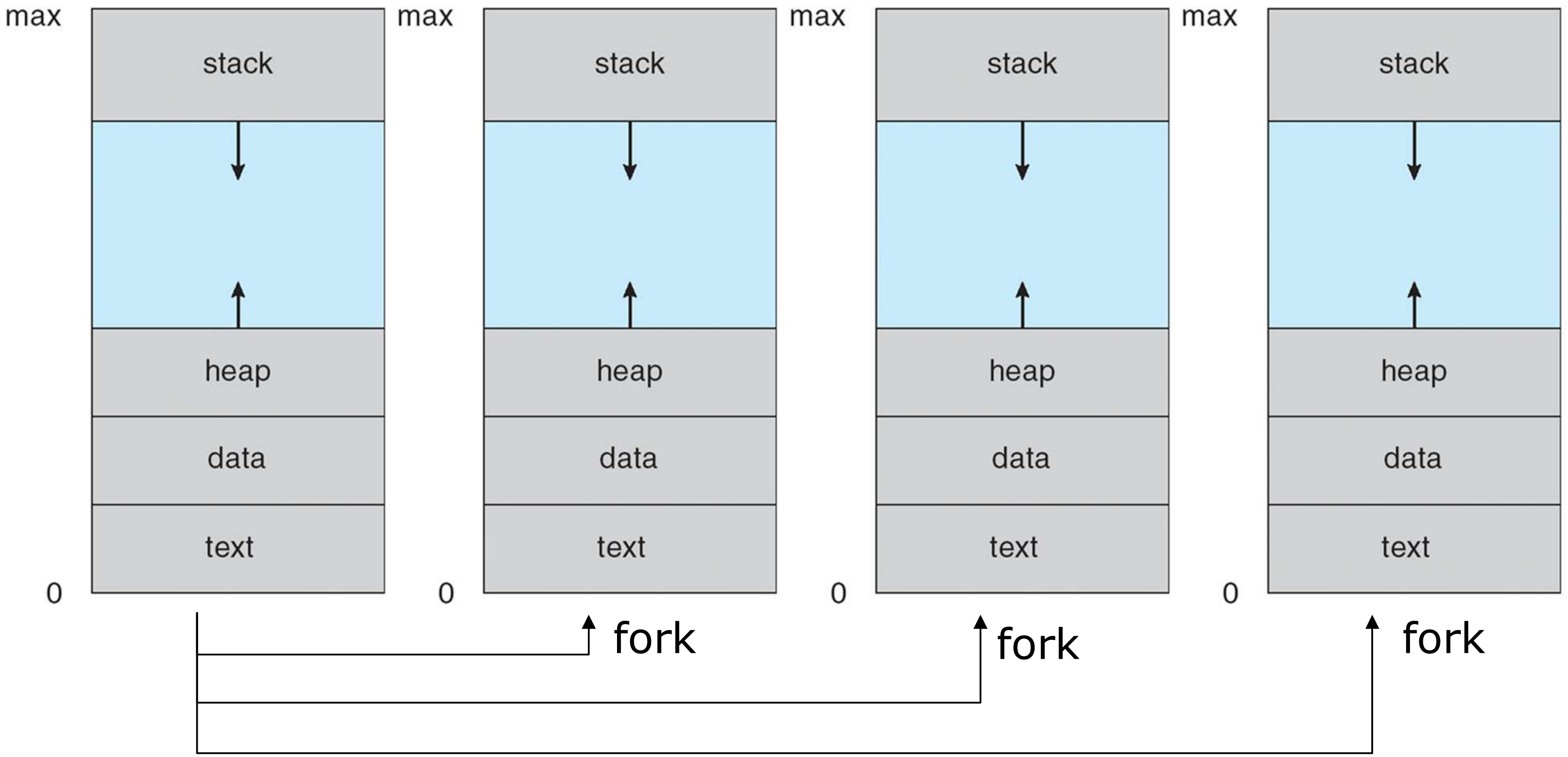
Insight

- A process is working



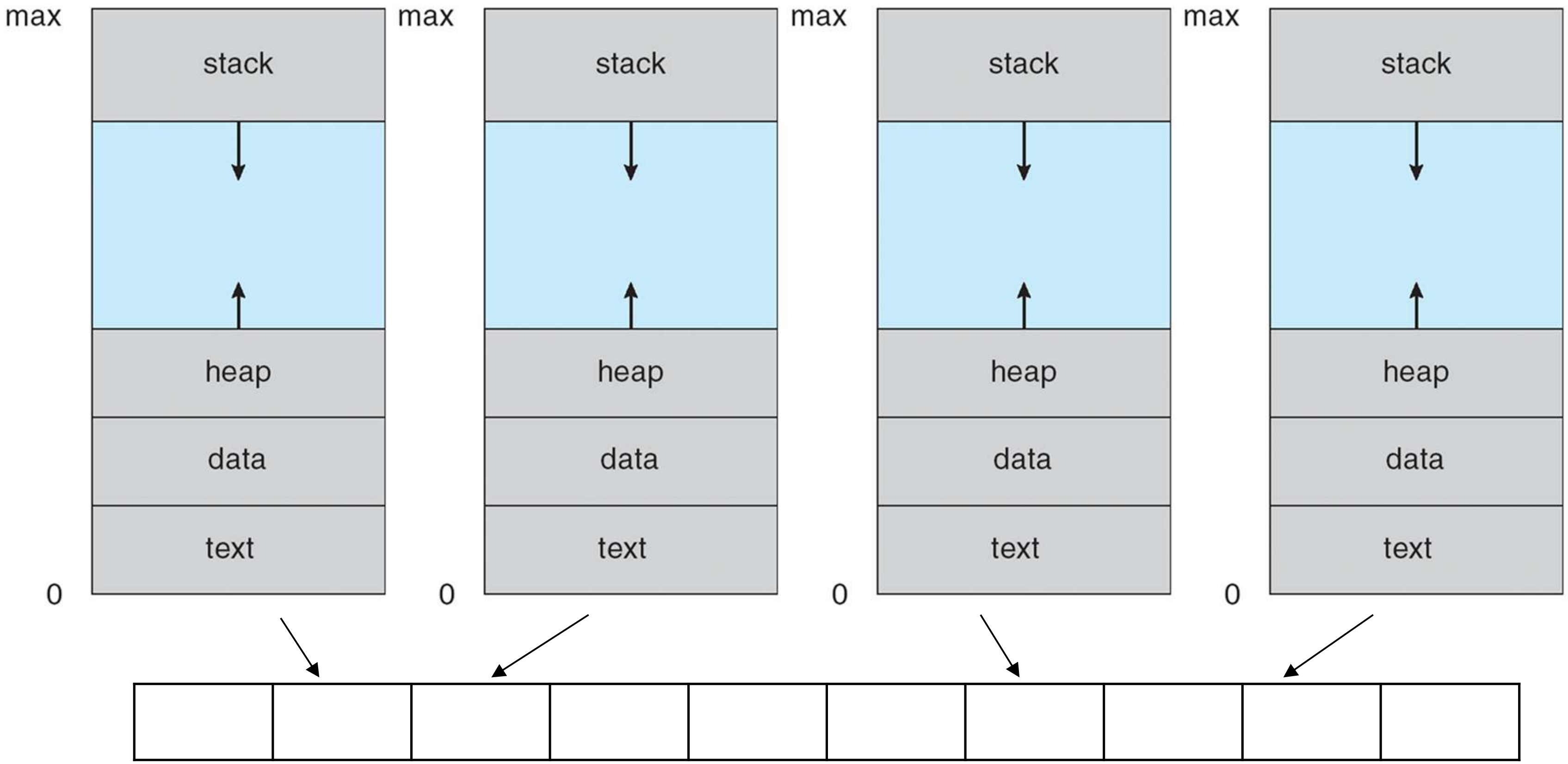
Insight

- A similar process is created by their parent process



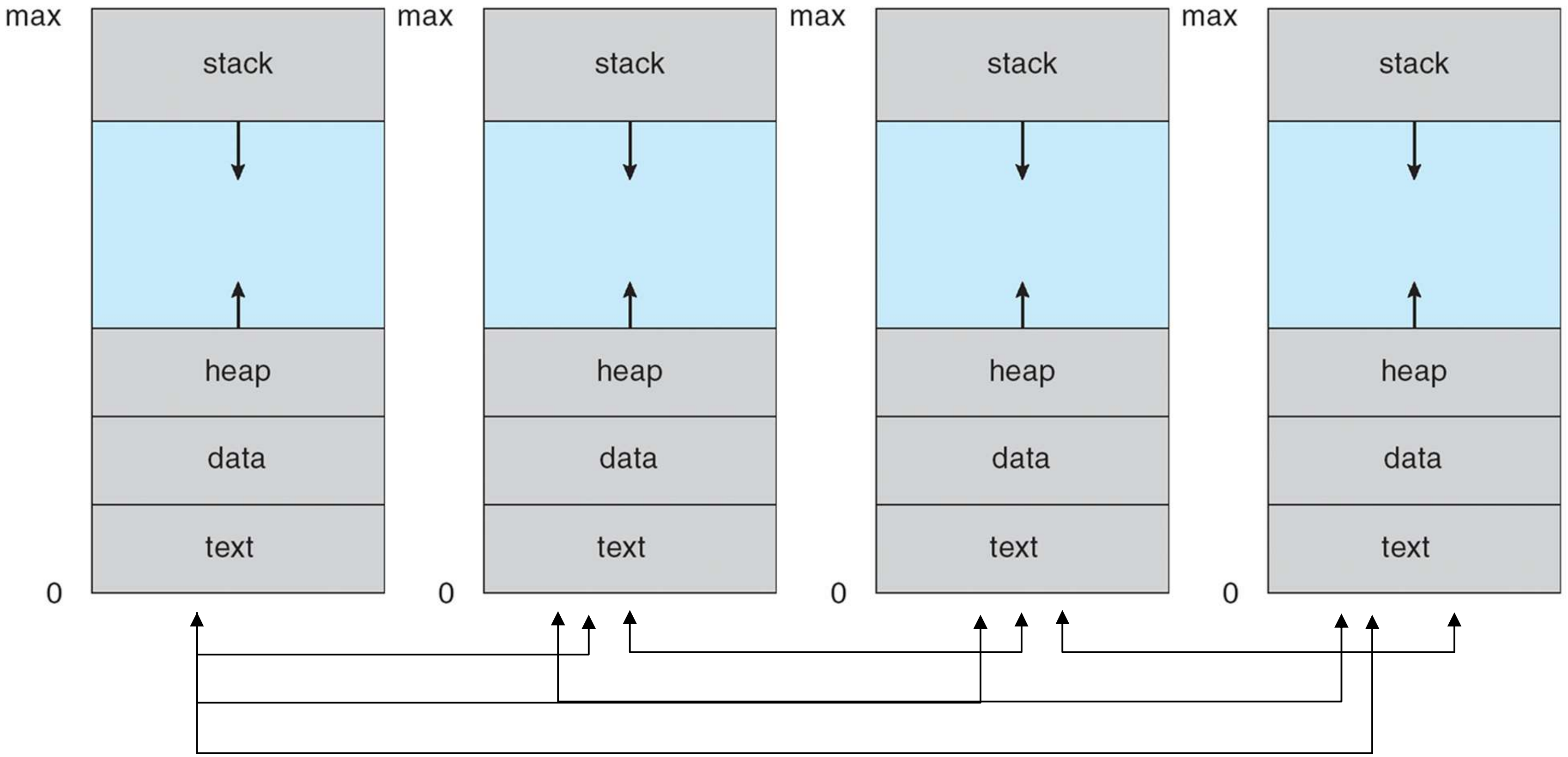
Insight

- All created processes would be in the ready cue with the parent process



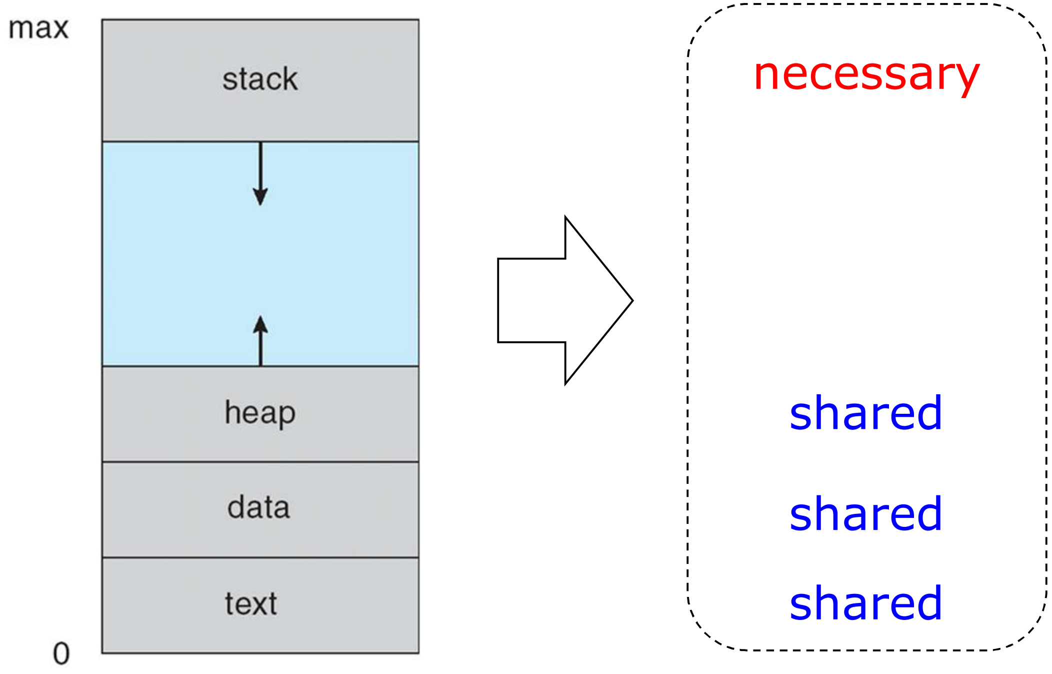
Insight

- What if processes need to send and receive data frequently?
  → Interprocess Communication (IPC)



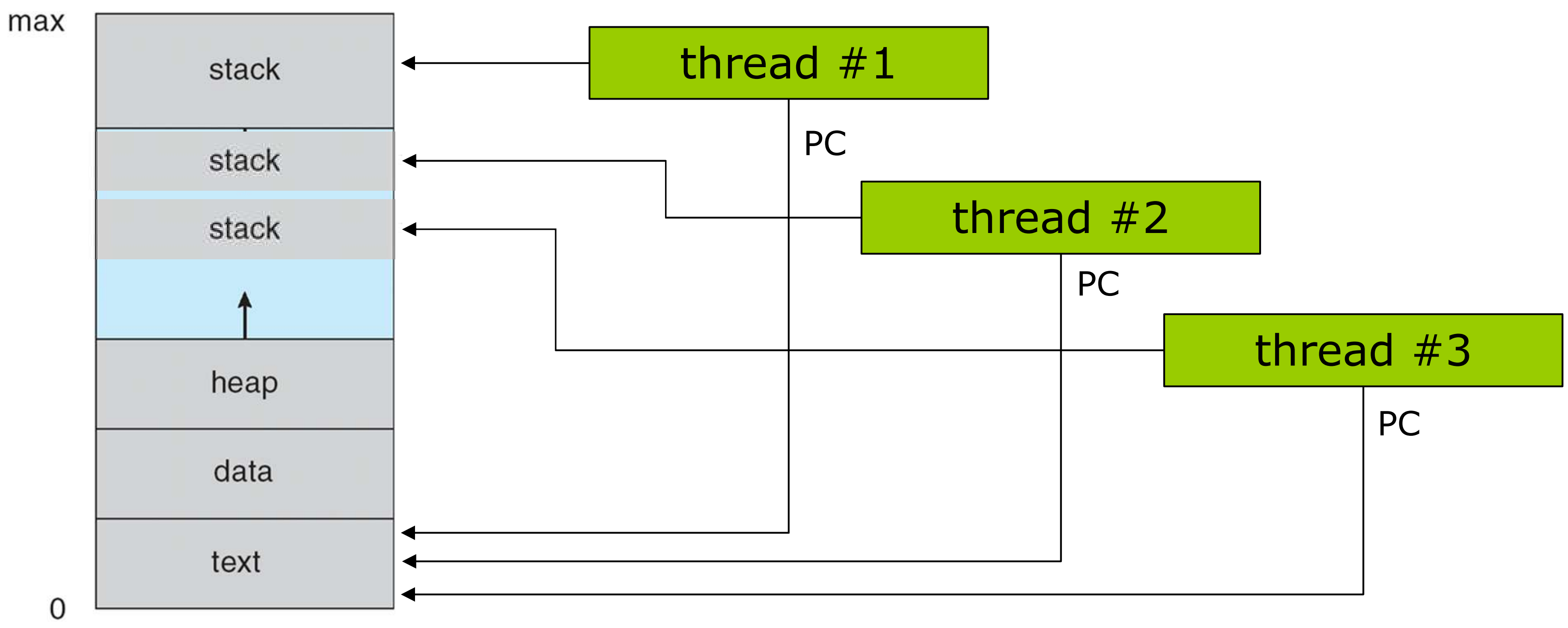
Insight

- Is there a simple way to implement processes that have similar functions with their parents?
- What is necessary? What is not necessary?



Insight

- Even the stack for a new thread is located in the memory space of the same process.
- Each thread has own Program Counter.



Processes are not always ideal...

Protection vs. Sharing

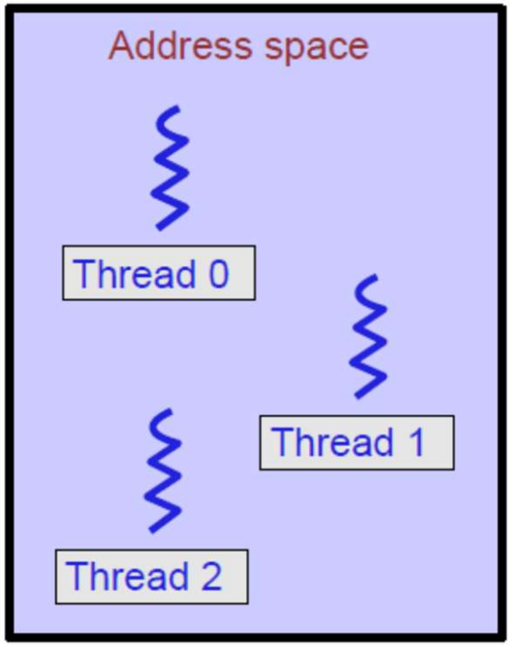
- Processes are not very efficient
  - **Each process** has its own PCB and OS resources
  - Creating a new process is often **very expensive**
- Processes don't (directly) share memory
  - Each process has **its own address space**
  - Parallel and concurrent programs often want to **directly manipulate** the same memory
    - e.g., When processing elements of a large array in parallel
  - Note: Many OS's provide some form of inter-process shared memory
    - Still, this requires more programmer work and does not address the efficiency issues.

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't we share?)
  - Execution state: CPU registers, stack, and program counter
- Key idea of this lecture:
  - Separate the concept of a process from a thread of control
  - The process is the address space and OS resources
  - Each thread has its own CPU execution state

Processes and Threads

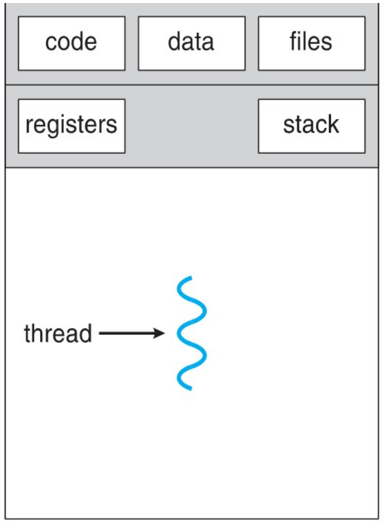
- **Thread**?
  - A basic unit of **CPU scheduling**
  - A process is just a "container" for its threads
  - Each thread is bound to its containing process
- Each thread has its own stack, CPU registers, etc.
- All threads within a process share the same address space and OS resources
  - **Threads share (process) memory**, so they can communicate directly!



Process

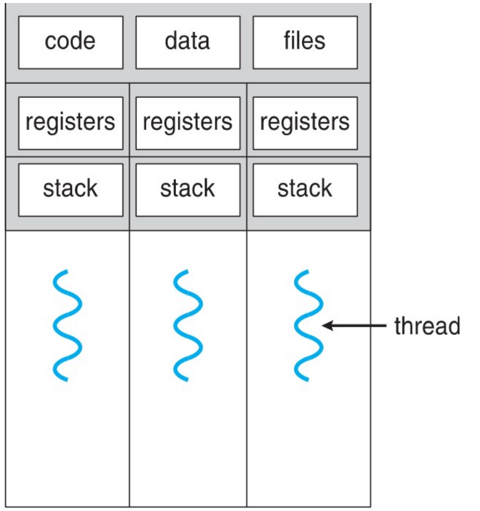
Thread

- Simple programs can have one thread per process
  - ***single-threaded*** *process*



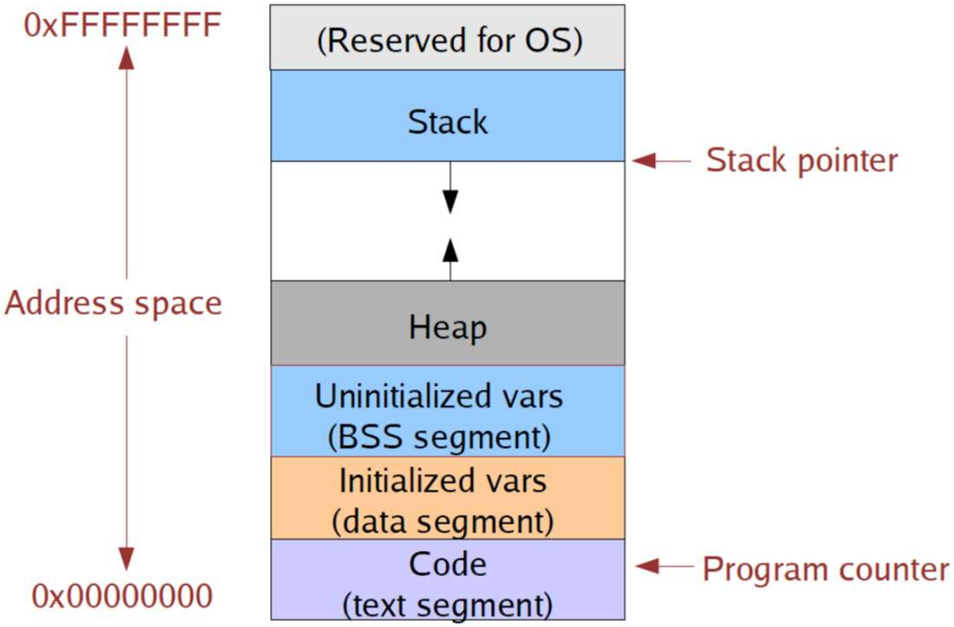
single-threaded process

- Complex programs can have multiple threads
  - ***multi-threaded*** *process*
  - Multiple threads running in same process's address space
  - Each thread can run on *different CPUs* (cores) while *sharing memory* resource

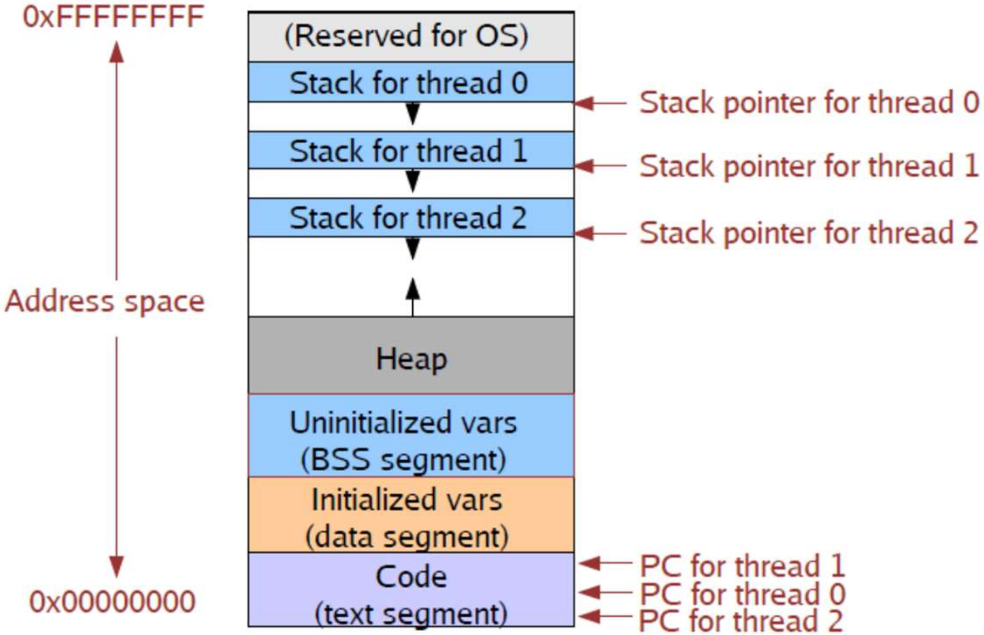


multithreaded process

Process Address Space with Threads



Process Address Space (Old)

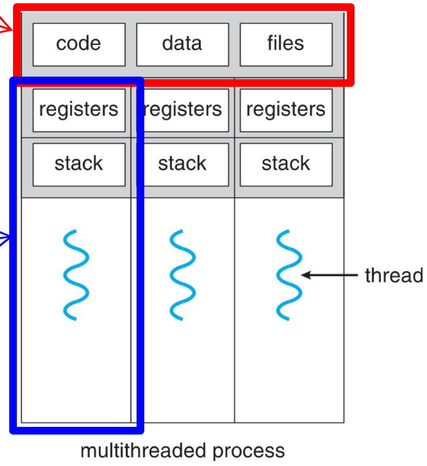


Process Address Space with Threads

All threads in a single process
share the same address space!

Benefits of Multithread

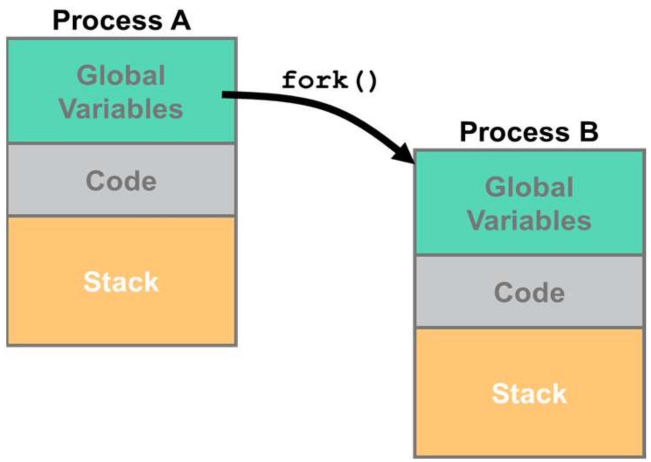
- **Resource Sharing**
  - threads share memory resources of process
    - CPU run-time resources (e.g., *register, stack, PC*) are <u>*not shared*</u>
  - easier to share than IPC



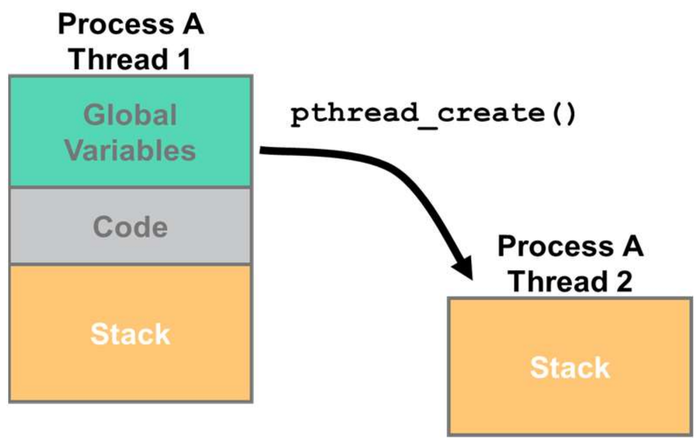
- **Lighter weight**
  - lighter weight than process
    - creation/deletion, context-switching faster
    - e.g., Solaris system time
      - creating thread 30x faster, context switching thread 5x faster

Thread vs. Process creation

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

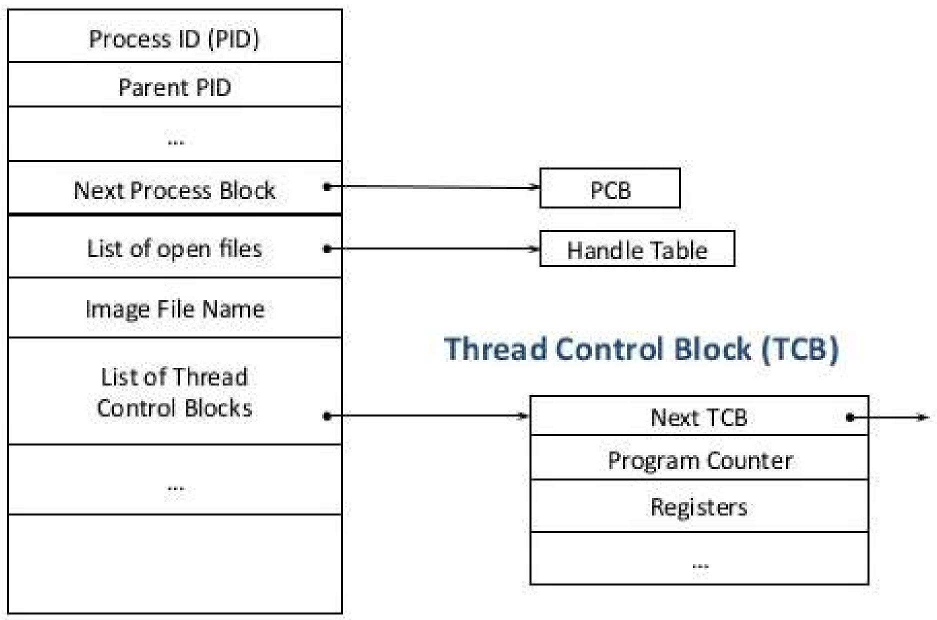


A thread creation using pthread_create() does ***not*** require a lot of memory or startup time.



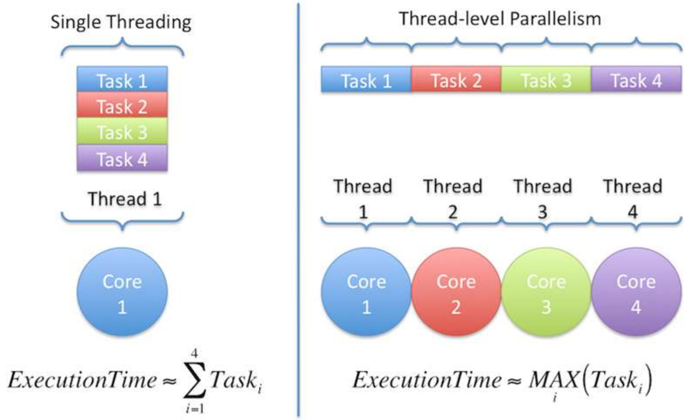
Control Blocks

- **Thread Control Block (TCB)**
  - Created for each thread
  - Contains Program Counter (PC), registers, and stack
- **Ready queue** is now a list of **TCB**s waiting for CPU resource
- **Context switching** is done for **TCB**s



Benefits of Multithread

- **Allows <u>one</u> process to use multiple CPUs or cores**
  - A multithreaded process can take advantage of multiprocessor (=multicore) architectures
  - A process can run many threads **<u>in parallel</u>** on different processor **<u>cores</u>**

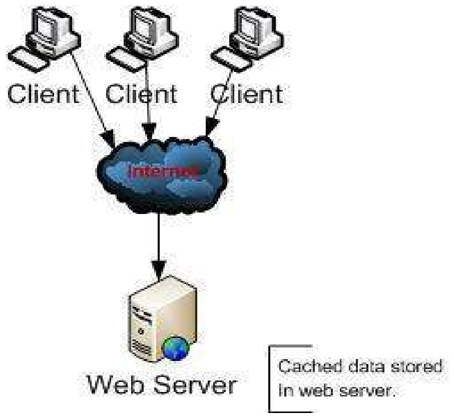


- **Non-blocking System call**
  - may allow continued execution if part of **process** is blocked
    - why blocked? – time consuming operation (e.g., I/O such as printing, network)
    - single-threaded process: wait until I/O operation is complete
    - multithreaded process: another thread can continue
  - Better ***<u>responsiveness</u>*** to the user

Wifi

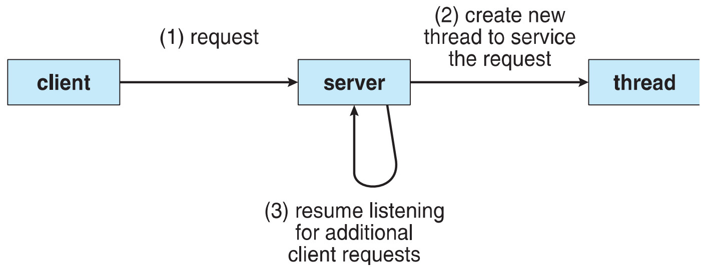
Example

- Web server (Process)
  - a busy web server may have several clients connected
  - Needed to perform **several similar tasks**
  - Traditional method: multi-process
    - when web server receives a request, create a new ***process***



Example contd.

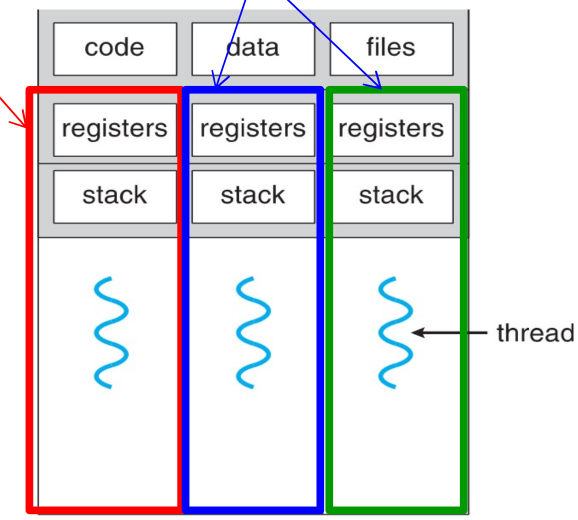
- Web Server using Multi-processes ?
  - Problems :
    - Time consuming and resource intensive
    - All new processes perform the same tasks - serving Web client



- Multithreaded method:
  - Share the resources with other threads belonging to the same process: <u>code, data, and files</u>
  - e.g, Web server
    - a thread to **listen** to the request from user.
    - When a new request is made, the server creates another thread to **serve** the request

Listen to client requests

Serve new request (send Web page)



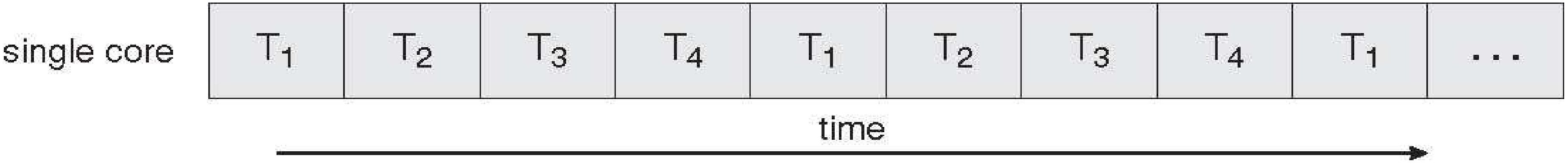
multithreaded process

Chapter 4: Multithreaded Programming

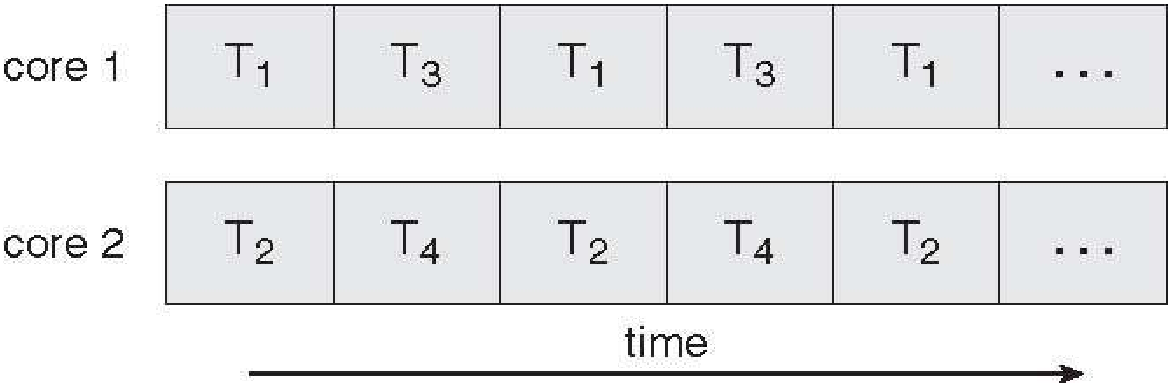
- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Process vs. Thread

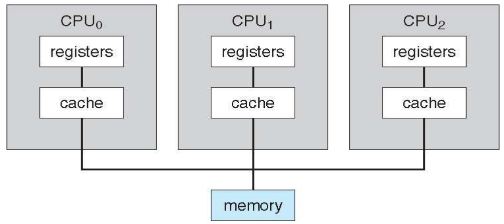
Multicore Programming

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



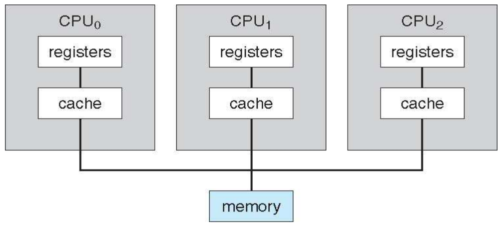
- ***Parallel*** Execution on a Multicore System





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

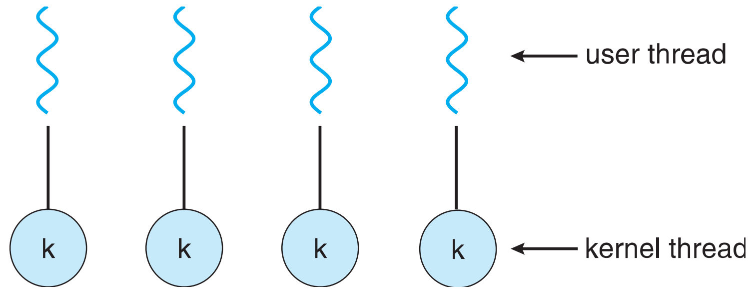


- By using multithreading, **one** process can use multiple processors (cores)!

Chapter 4: Multithreaded Programming

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Process vs. Thread

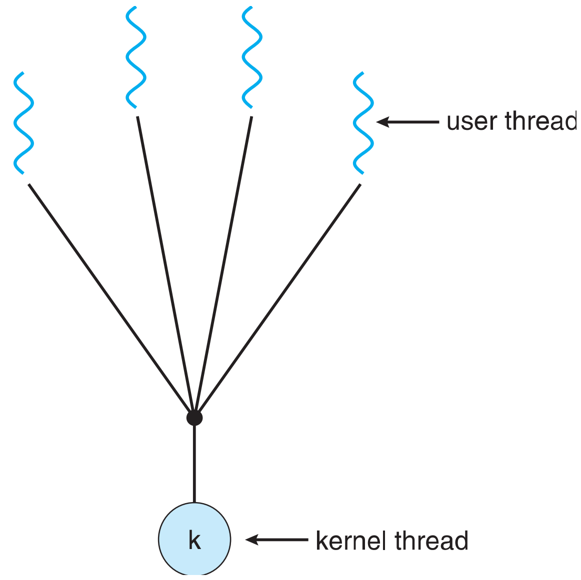
Kernel Threads



- **Kernel threads**
  - Threads supported by the **Kernel**
    - created by system call
    - each thread needs thread control block (TCB)
  - Pros
    - 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
    - every thread operation must go through kernel
      - still much lighter than process

User thread

- **User Thread**
  - Implement thread in <u>user library</u>
  - created/managed without kernel support (no need for system call)
  - many user threads mapped to single kernel thread
- Pros
  - Thread management is done by the thread library in <u>user space</u>
  - Fast and efficient (10-30x faster than kernel thread)
- Cons: kernel doesn't know the user thread, so…
  - A thread makes a system call - one thread blocking causes all to **block**
  - Multiple user-level threads may ***not*** run in parallel on multicore system



why?

Operating delay of thread and process

unit: μs

operation	User thread	Kernel thread	Process
Null fork	34	948	11300
Signal-wait	37	441	1840

Benchmarked under UNIX system.

Chapter 4: Multithreaded Programming

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Process vs. Thread

POSIX threads

- POSIX (IEEE 1003.1c-1995) provided a standard known as ***pthreads***
  - Common in UNIX operating systems (Solaris, Linux, Mac OS X)
  - Thread management
    - Creating, detaching, joining, etc. Set/query thread attributes
  - Mutexes
    - Synchronization

Multithreaded C program using the Pthreads API (1)

```
#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 */

    if( argc != 2 ) {
          fprintf( stderr, "usage: a.out <integer value>\n" );
          return -1;
    }
    if( atoi(argv[1]) < 0 ) {
          fprintf( stderr, "%d must be >= 0\n", atoi( argv[1] ) );
          return -1;
    }

    /* get 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 );
}
```

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

Separate Thread does this..

① Thread Creation

Unique thread identifier (ID) returned from call

Attributes structure (NULL for defaults)

```
int pthread_create(pthread_t *tid,
                   const pthread_attr_t *attr,
                   void *(*func)(void *),
                   void *arg);
```

zero for success, else error number

main routine for child thread

Argument passed

`func` is the function to be called.

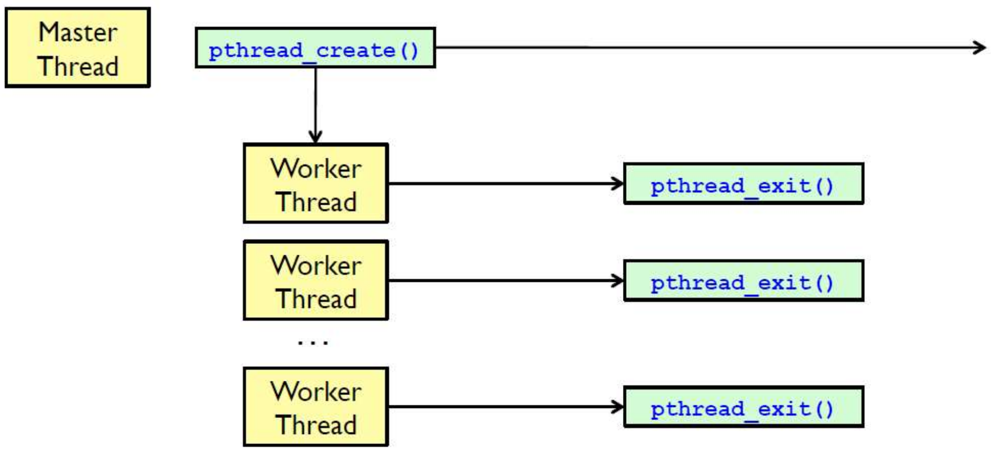
When `func()` returns, the thread is terminated.

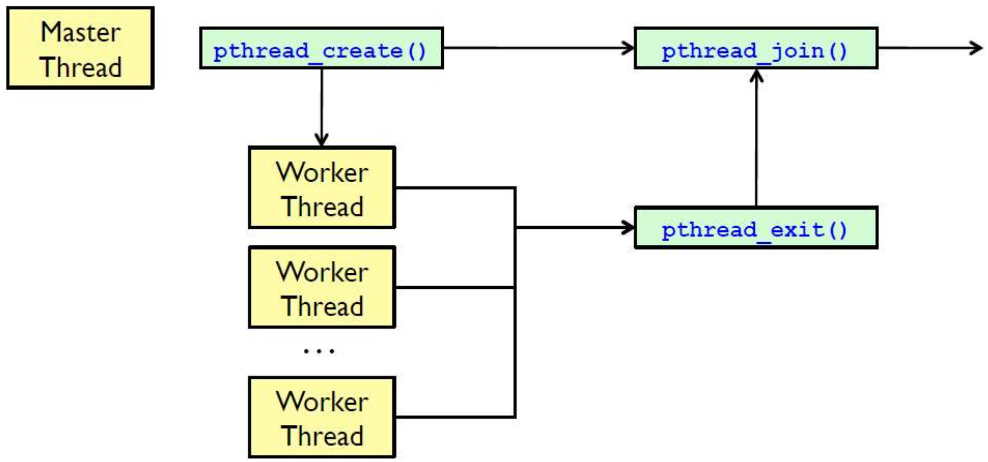
② Thread Function

```
/* 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 );
   return NULL;
}
```

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

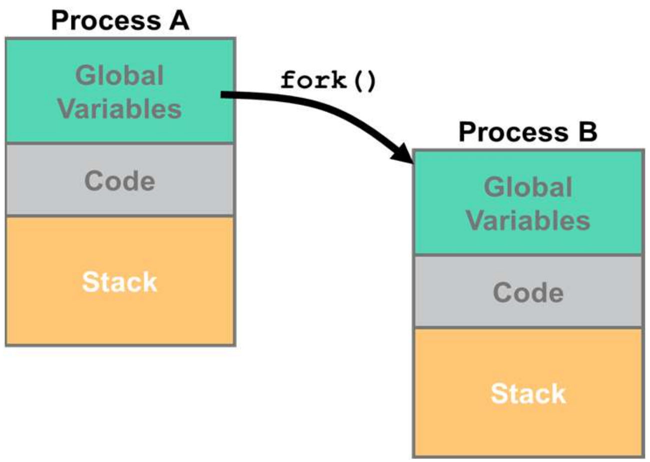
similar to wait system call in process

Chapter 4: Multithreaded Programming

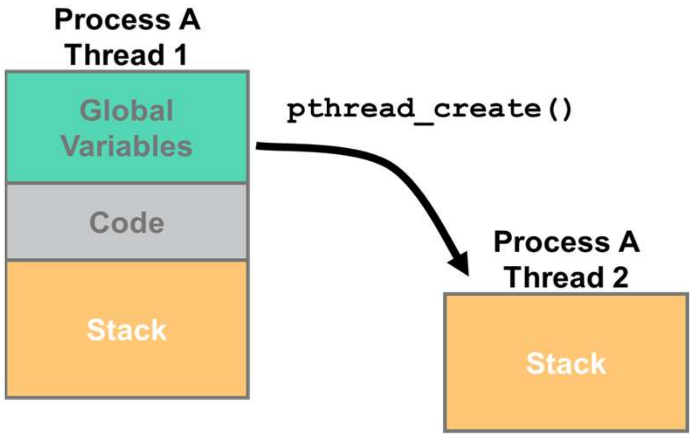
- Overview
- Benefits of Multithreading
- Multithreading Models
- Thread Libraries
- Process vs. Thread

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



Possible output?

- 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: 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;
}
```

time

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

Conclusion

- Thread concept
  - Basic unit of CPU scheduling
  - Shares with other threads belonging to the same process <u>its code, data, and other resources</u> such as open files and signals (not register, stack)

- Multi-threading models (Kernel thread, User thread)

- Thread libraries: pthread, Java thread
- Process vs. Thread

Future work

- Who gets to go next when a thread blocks/yields?
  - Scheduling!

- What happens when multiple threads are sharing the same resource?
  - Synchronization!

- Appendix
  - Java Threads

Java Threads

```java
class Sum
{
  private int sum;

  public int getSum() {
   return sum;
  }

  public void setSum(int sum) {
   this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }

  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setSum(sum);
  }
}
```

thread function

```java
public class Driver
{
  public static void main(String[] args) {
   if (args.length > 0) {
    if (Integer.parseInt(args[0]) < 0)
     System.err.println(args[0] + " must be >= 0.");
    else {
     Sum sumObject = new Sum();
     int upper = Integer.parseInt(args[0]);
     Thread thrd = new Thread(new Summation(upper, sumObject));
     thrd.start();
     try {
       thrd.join();
       System.out.println
             ("The sum of "+upper+" is "+sumObject.getSum());
    } catch (InterruptedException ie) { }
    }
   }
   else
    System.err.println("Usage: Summation <integer value>"); }
}
```

Java Threads contd.

```java
class Sum
{
  private int sum;

  public int getSum() {
   return sum;
  }

  public void setSum(int sum) {
   this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }

  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setSum(sum);
  }
}
```

```java
public class Driver
{
  public static void main(String[] args) {
   if (args.length > 0) {
    if (Integer.parseInt(args[0]) < 0)
     System.err.println(args[0] + " must be >= 0.");
    else {
     Sum sumObject = new Sum();
     int upper = Integer.parseInt(args[0]);
     Thread thrd = new Thread(new Summation(upper, sumObject));
     thrd.start();
     try {
        thrd.join();
        System.out.println
              ("The sum of "+upper+" is "+sumObject.getSum());
    } catch (InterruptedException ie) { }
    }
   }
   else
    System.err.println("Usage: Summation <integer value>"); }
}
```