CO3

Session - 1

COURSE NAME: OPERATING SYSTEMS

COURSE CODE: 23CS2104R/A

Inter Process Communication, Threads & Thread API

AIM OF THE SESSION

To familiarize students with the basic concept of Threads.

INSTRUCTIONAL OBJECTIVES

This Session is designed to:

1. Demonstrate what is meant by Inter-Process Communication.
2. Demonstrate what is meant by a Thread.
3. Describe the types of Threads.
4. Describe the Thread Models.

LEARNING OUTCOMES

At the end of this session, you should be able to:

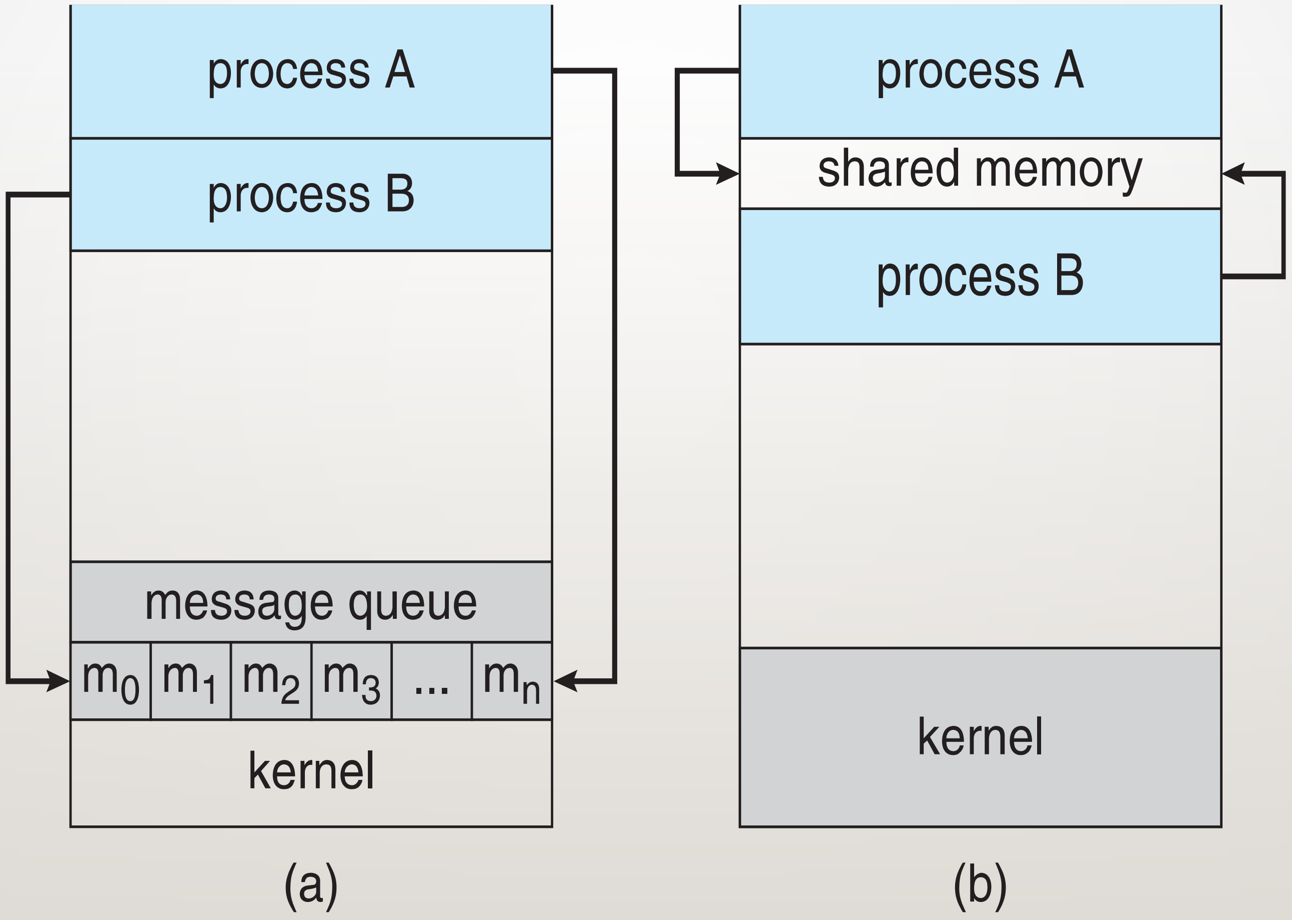
1. Defines what inter-process communication is.
2. Describe Thread Models.
3. Summarize the Role of Thread.

INTER PROCESS COMMUNICATION

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **inter process communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message Passing**

COMMUNICATIONS MODELS

(a) Message passing. (b) shared memory.



COOPERATING PROCESSES

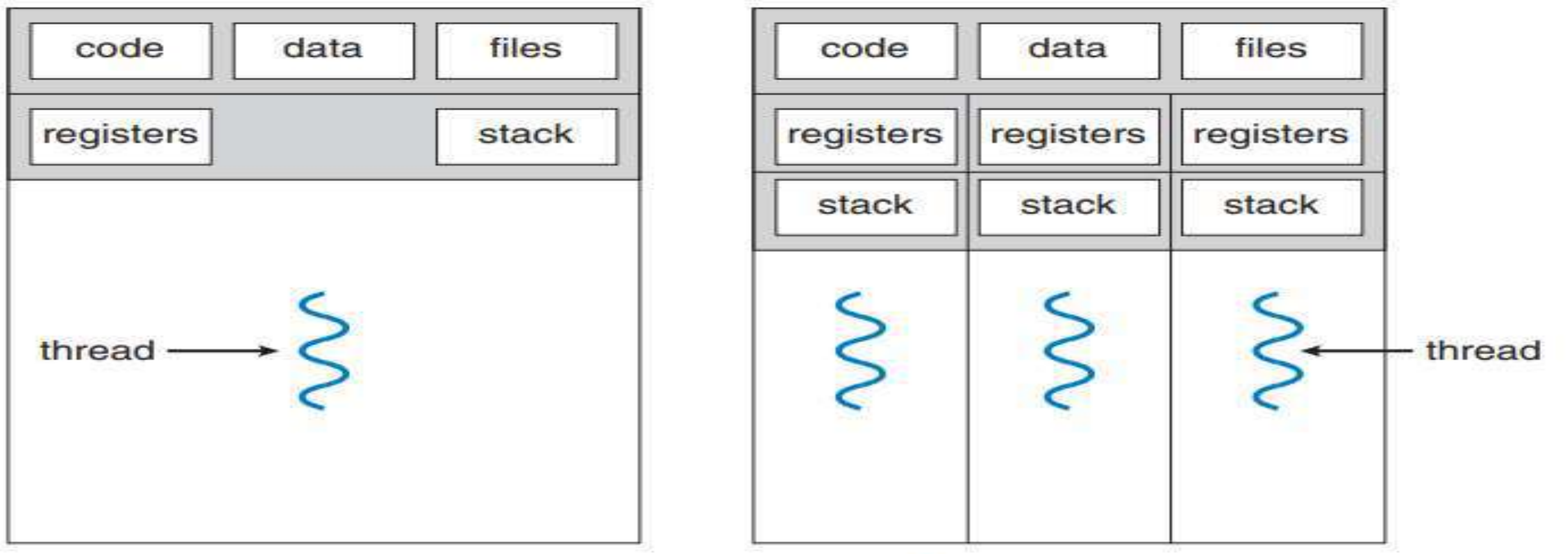
- ***Independent*** process cannot affect or be affected by the execution of another process
- ***The cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

THREAD

Thread is an execution unit that consists of its **program counter**, **a stack**, and a set of **registers** where the program counter mainly keeps track of which instruction to execute next, a set of registers mainly holds its current working variables, and a stack mainly contains the history of execution

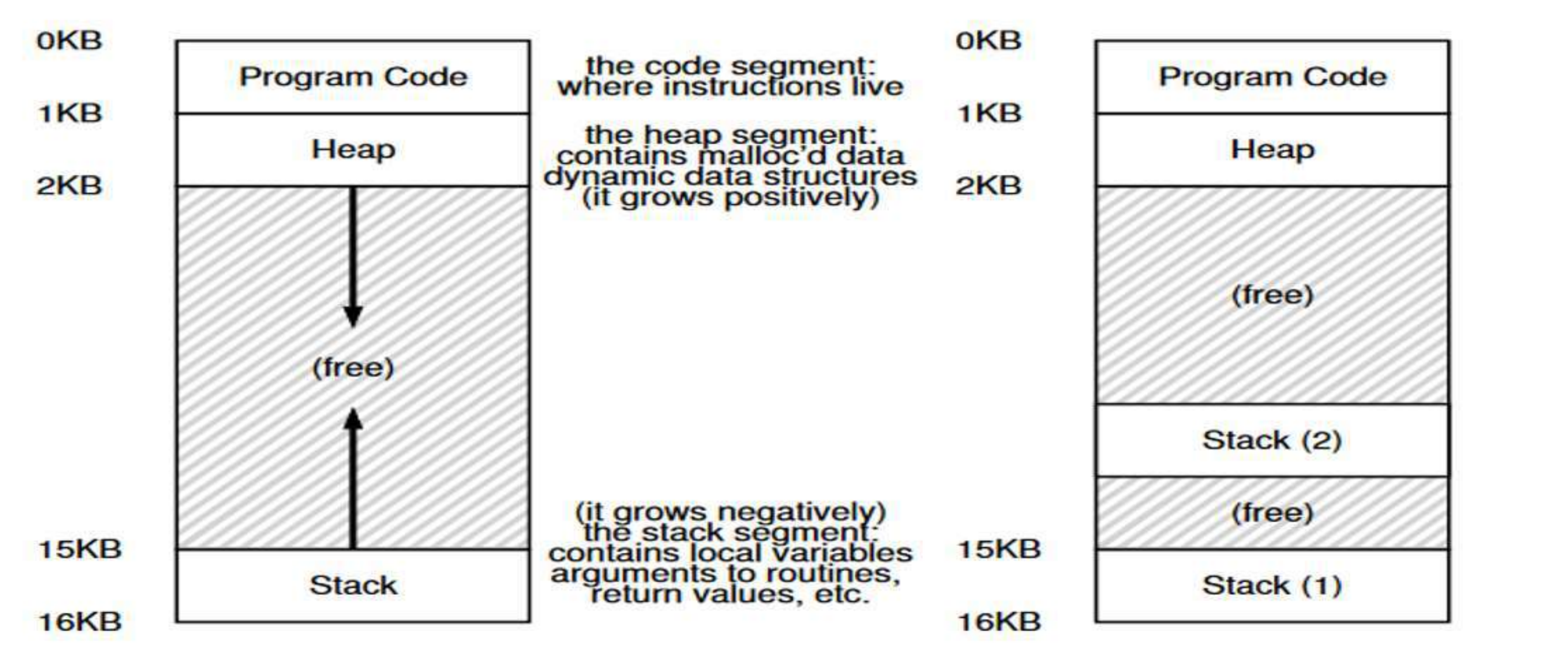
Threads are also known as **Lightweight processes**. Threads are a popular way to improve the performance of an application through parallelism.

SINGLE-THREAD AND MULTI-THREAD PROCESS



single-threaded process
multithreaded process

SINGLE-THREAD AND MULTI-THREAD PROCESS



WHY DO WE NEED THREAD?

- Threads run in parallel improving the application performance.
- Each such thread has its own CPU state and stack, but they share the address space of the process and the environment.
- Threads can share common data so they do not need to use inter-process communication. Like the processes, threads also have states like ready, executing, blocked, etc.
- Priority can be assigned to the threads just like the process, and the highest priority thread is scheduled first.
- Each thread has its own Thread Control Block (TCB). Like the process, a context switch occurs for the thread, and register contents are saved in (TCB).

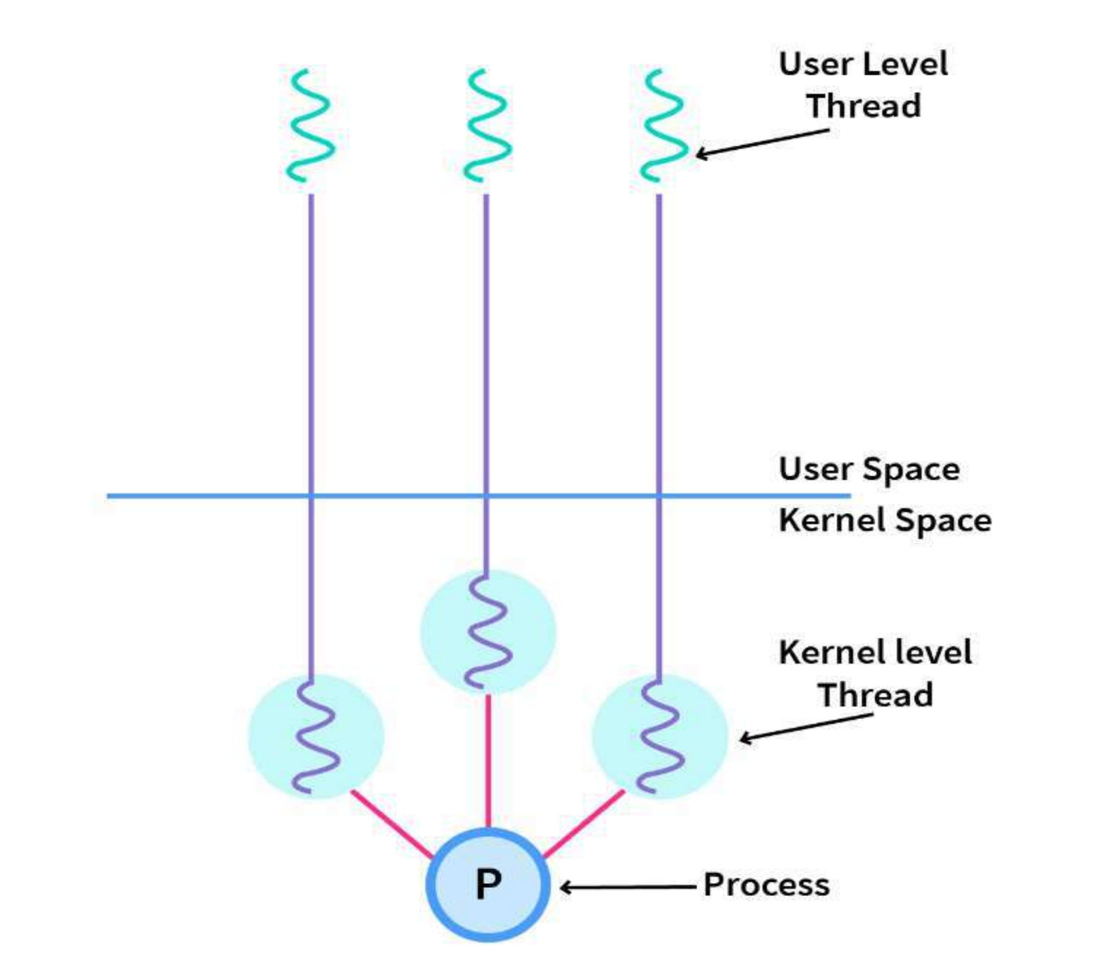
DIFFERENCES BETWEEN PROCESS AND THREAD

Process	Thread
A Process simply means any program in execution.	Thread simply means a segment of a process.
The process consumes more resources	Thread consumes fewer resources.
The process requires more time for creation.	Thread requires comparatively less time for creation than process.
The process is a heavyweight process	Thread is known as a lightweight process
The process takes more time to terminate	The thread takes less time to terminate.
Processes have independent data and code segments	A thread mainly shares the data segment, code segment, files, etc. with its peer threads.
The process takes more time for context switching.	The thread takes less time for context switching.
Communication between processes needs more time as compared to thread.	Communication between threads needs less time as compared to processes.

TYPES OF THREADS

In the operating systems, there are two types of threads.

1. Kernel-level thread.
2. User-level threads.



CONTEXT SWITCH BETWEEN THREADS

- Each thread has its own program counter and set of registers.
- if there are two Threads that are running on a single processor, when switching from T1 to T2, a **context switch must take place.**
- The context switch between threads is quite similar to the context switch between processes, as the register state of T1 must be saved and the register state of T2 restored before running T2.
- With processes, we saved the state to a **process control block (PCB); now, we'll need one or more thread control blocks (TCBs)** to store the state of each thread of a process.
- There is one major difference, though, in the context switch we perform between threads as compared to processes: the address space remains the same (i.e., there is no need to switch which page table we are using).

CONTEXT SWITCH BETWEEN THREADS

- In our simple model of the address space of a classic process (which we can now call a **single-threaded process), there is a single stack,** usually residing at the bottom of the address space.
- In a multi-threaded process, each thread runs independently.
- Thus, any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called **thread-local storage, i.e., the stack of the relevant** thread.

MULTITHREADING MODELS

The user threads must be mapped to kernel threads, by one of the following strategies:

1)Many to One Model

2)One to One Model

3)Many to Many Model

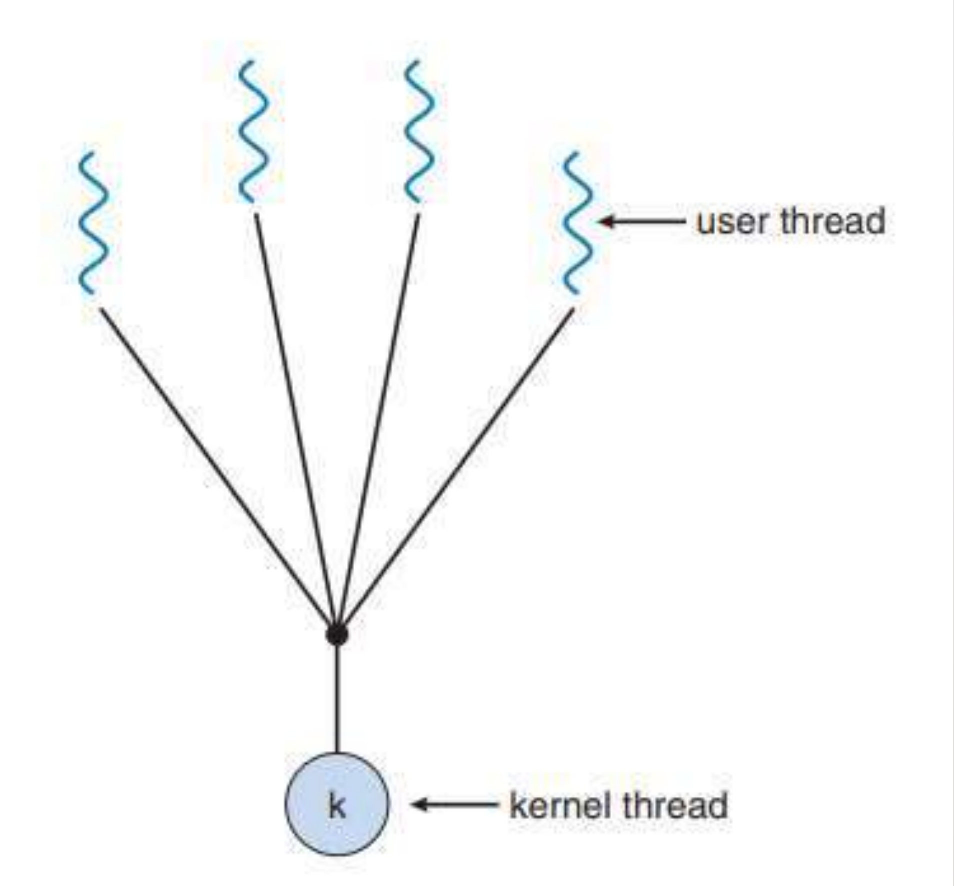
MANY-TO-ONE MODEL

1. Many user-level threads mapped to a single kernel thread
2. Thread library is in user space

Drawback: The Entire process will block if a thread makes a blocking system call

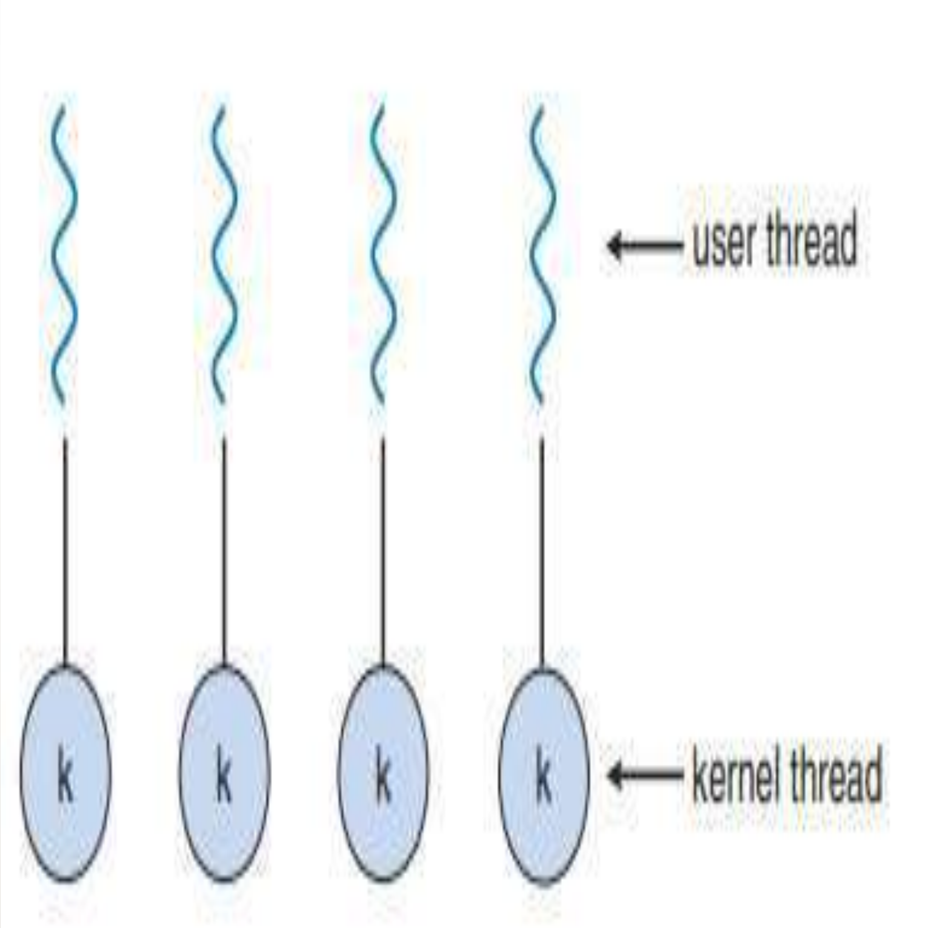
Examples:

1. **Solaris Green Threads**
2. **GNU Portable Threads**



ONE TO ONE MODEL

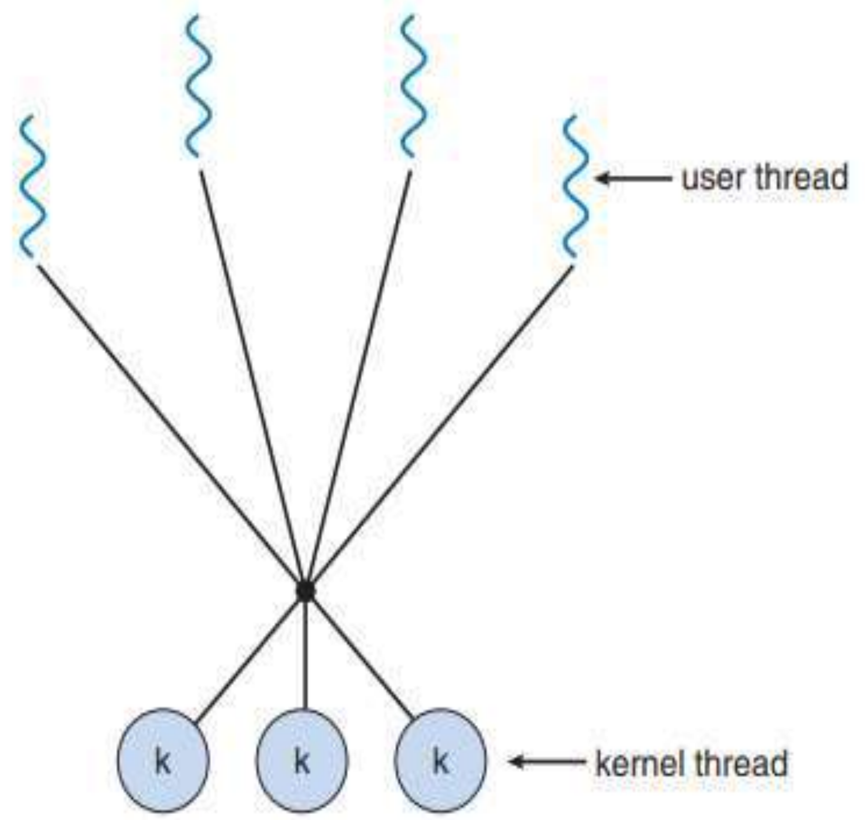
- The **one-to-one** model creates a separate kernel thread to handle every user thread.
- Most implementations of this model limit how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.
- This model provides more concurrency than that of the many-to-one Model.



MANY TO MANY MODEL

The **many to many** model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.

- Users can create any number of threads.
- Example: Solaris prior to version 9



BENEFITS OF THREADS

- **Responsiveness**
- **Resource Sharing**
- **Economy**
- **Scalability**

THREAD TRACE

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Figure 26.3: Thread Trace (1)

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.4: Thread Trace (2)

THREAD API

- int pthread_create (pthread_t* thread, const pthread_attr_t * attr, void* (*start_routine)(void*), void* arg)
- int pthread_join(pthread_t thread, (void *)*value_ptr)
- int pthread_mutex_lock(pthread_mutex_t *mutex)
- int pthread_mutex_unlock(pthread_mutex_t *mutex)
- int pthread_mutex_trylock(pthread_mutex_t *mutex)
- int pthread_mutex_timelock(pthread_mutex_t *mutex, struct timespec *abs_timeout);
- int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
- int pthread_cond_signal(pthread_cond_t *cond)

THREAD CREATION

How to create and control threads?

```
#include <pthread.h>

int
pthread_create(       pthread_t*       thread,
                const pthread_attr_t* attr,
                      void*            (*start_routine)(void*),
                      void*            arg);
```

- thread: Used to interact with this thread (OUT).
- attr: Used to specify any attributes this thread might have.
  - Stack size, Scheduling priority, … (IN)
- start_routine: the function this thread start running in (IN)
- arg: the argument to be passed to the function (start routine) (IN)
  - *a void pointer* allows us to pass in *any type of* argument.
- Returns 0 if went good (a error code otherwise)

If `start_routine` instead required another type argument, the declaration would look like this (example):

An integer argument:

```
int
pthread_create(…, // first two args are the same
               void*   (*start_routine)(int),
               int     arg);
```

Input is anything (usually a pointer to struct for multiple arguments or even internal returns), return an integer:

```
int
pthread_create(…, // first two args are the same
               int   (*start_routine)(void*),
               void*     arg);
```

WAIT FOR A THREAD TO COMPLETE

```
int pthread_join(pthread_t thread, (void *)*value_ptr);
```

`thread`: Specify which thread *to wait for*

`value_ptr`: A pointer we want to put the return value of the start routine

- Because `pthread_join()` routine changes the value, you need to pass in a pointer to that value.

Returns 0 if good, or EINVAL if err

POSSIBLE OUTCOMES

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Figure 26.3: Thread Trace (1)

POSSIBLE OUTCOMES

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.4: Thread Trace (2)

POSSIBLE OUTCOMES

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.5: Thread Trace (3)

THREADS IN DIFFERENT ENVIRONMENTS

1. **POSIX:(UNIX OS):** The POSIX thread libraries are a C/C++ thread API based on standards. It enables the creation of a new concurrent process flow.
2. **Win32(WINDOWS OS):** These Threads are provided as a kernel-level library on Windows systems.
3. **Java Threads(Platform Independent)**: Since Java generally runs on a Java Virtual Machine, implementing threads is based upon whatever OS and hardware the JVM is running on, i.e., Pitheads or Win32 threads depending on the system.

SIMPLE THREAD CREATION

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Figure 26.2: Simple Thread Creation Code (t0.c)

THANK YOU

Team – Operating System