# Chap. 4) Threads & Concurrency

경희대학교 컴퓨터공학과

조 진 성

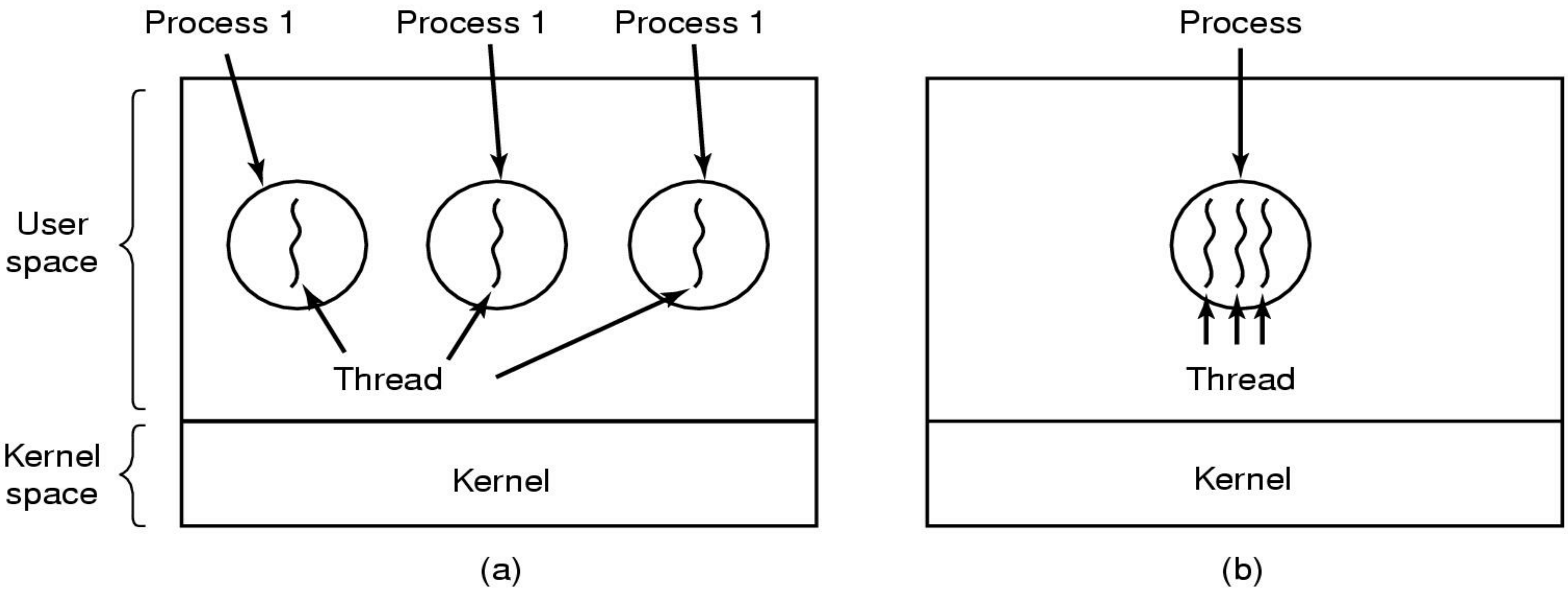
# Process

- Heavy-weight
  - ✓ A process includes many things:
    - An address space (all the code and data pages)
    - OS resources (e.g., open files) and accounting info.
    - Hardware execution state (PC, SP, registers, etc.)
  - ✓ Creating a new process is costly because all of the data structures must be allocated and initialized
    - Linux: over 100 fields in task_struct
      (excluding page tables, etc.)
  - ✓ Inter-process communication is costly, since it must usually go through the OS
    - Overhead of system calls and copying data

# Thread Concept: Key Idea

- Separate the concept of a process from its execution state
  - ✓ Process: address space, resources, other general process attributes (e.g., privileges)
  - ✓ Execution state: PC, SP, registers, etc.
  - ✓ This execution state is usually called
    - a thread of control,
    - a thread, or
    - a lightweight process (LWP)

# Thread Concept: Key Idea



# Single and Multithreaded Processes

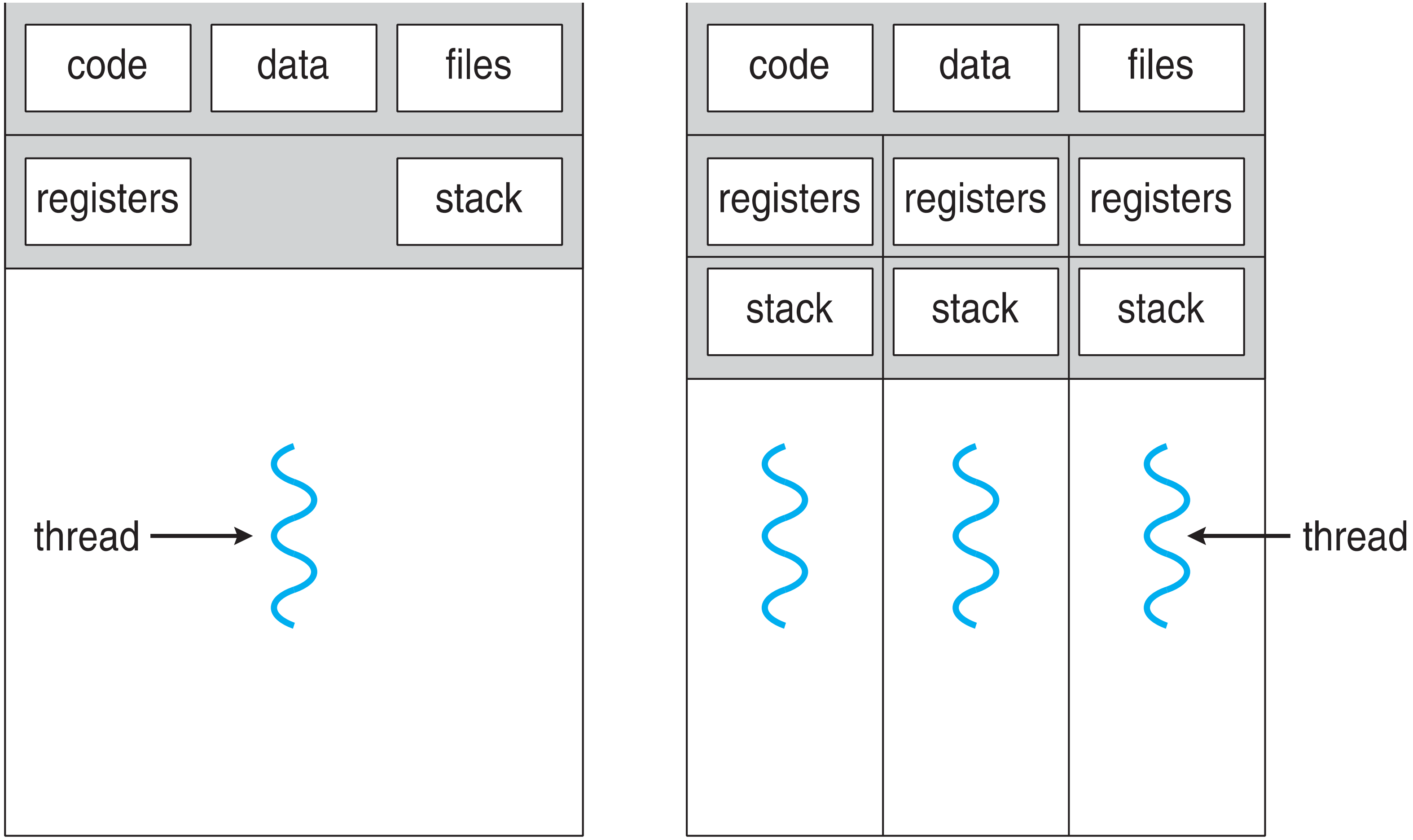
- Single-threaded process

```
void func1(void *p) { … }
void func2(void *p) { … }

main()
{
    func1(…);
    func2(…);
    …
}
```

- Multithreaded process

```
void func1(void *p) { … }
void func2(void *p) { … }

main()
{
    thread_create(func1, …);
    thread_create(func2, …);
    …
}
```

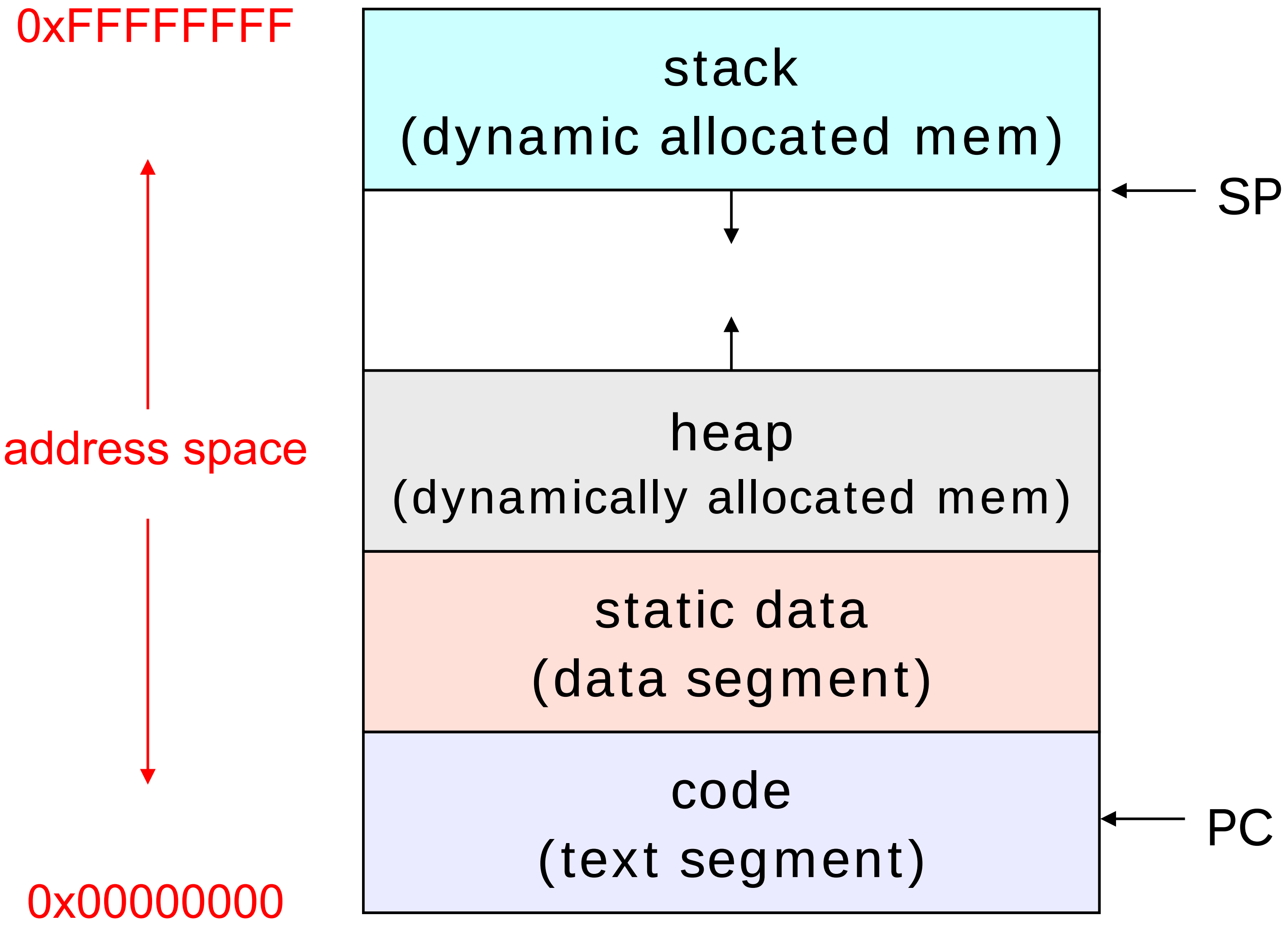
# Single and Multithreaded Processes



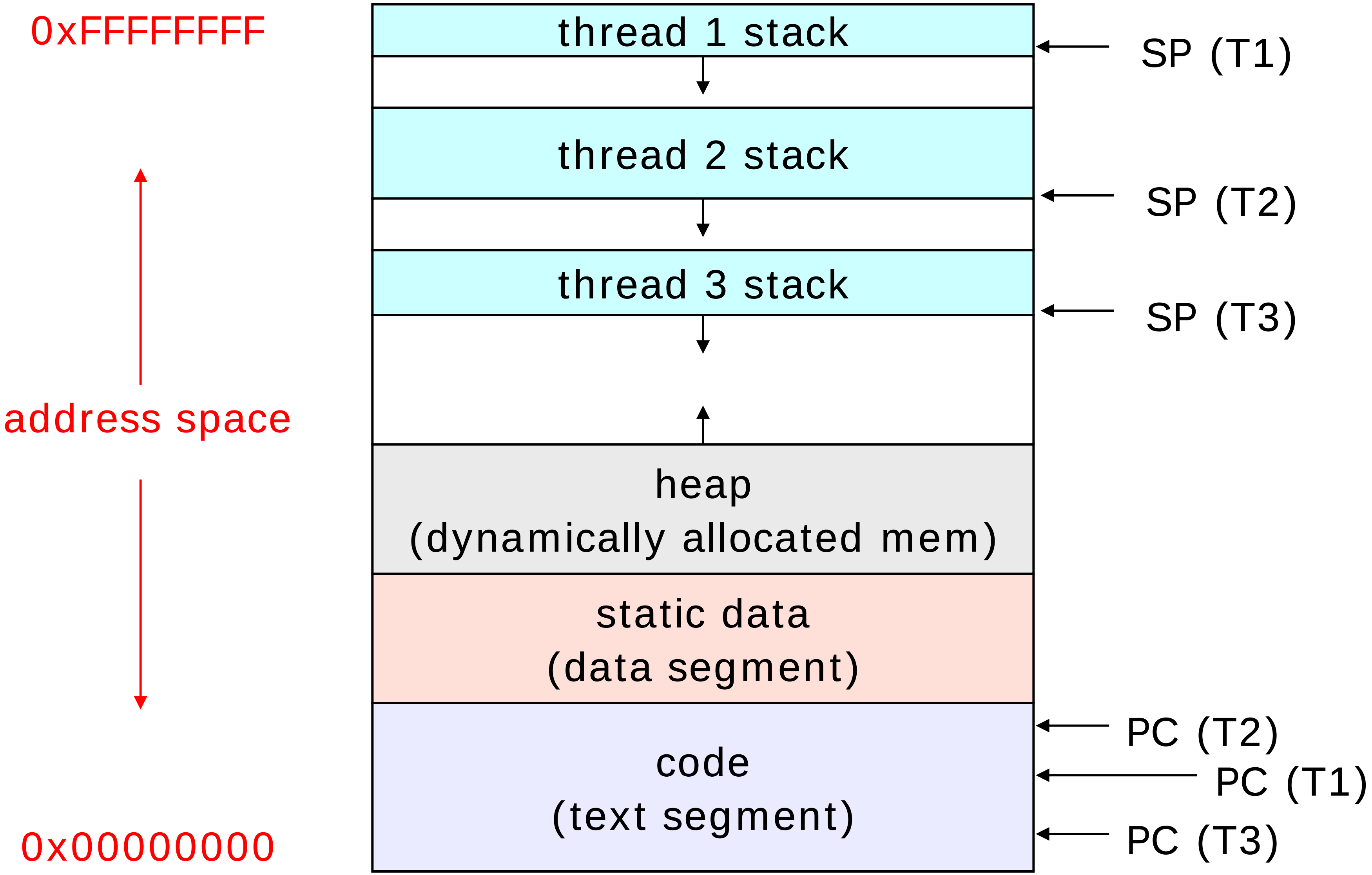
single-threaded process

multithreaded process

# Revisited: Process Address Space



# Address Space with Threads

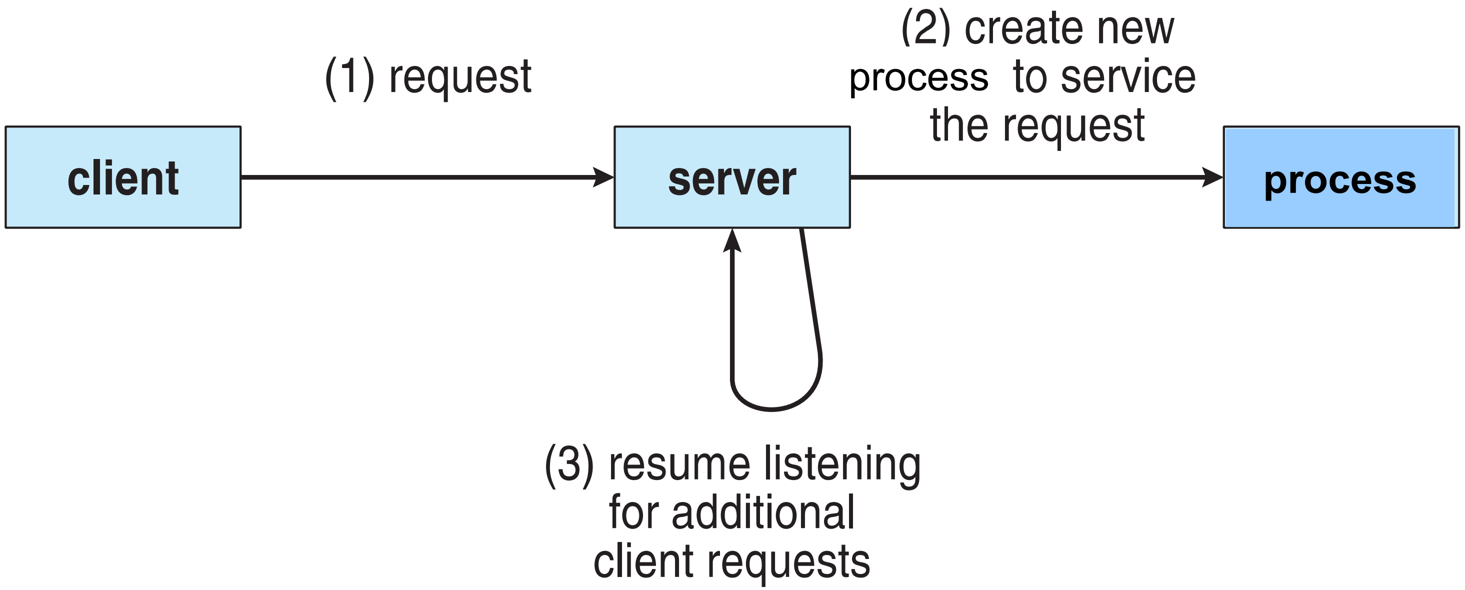


# Concurrent Servers: Multiprocess Model

- Web server example
  - ✓ Using fork() to create new processes to handle requests in parallel is overkill for such a simple task

```
While (1) {
    int sock = accept();
    if ((pid = fork()) == 0) {
        /* Handle client request */
    } else {
        /* Close socket */
    }
}
```

# Concurrent Servers: Multiprocess → Multithread



client

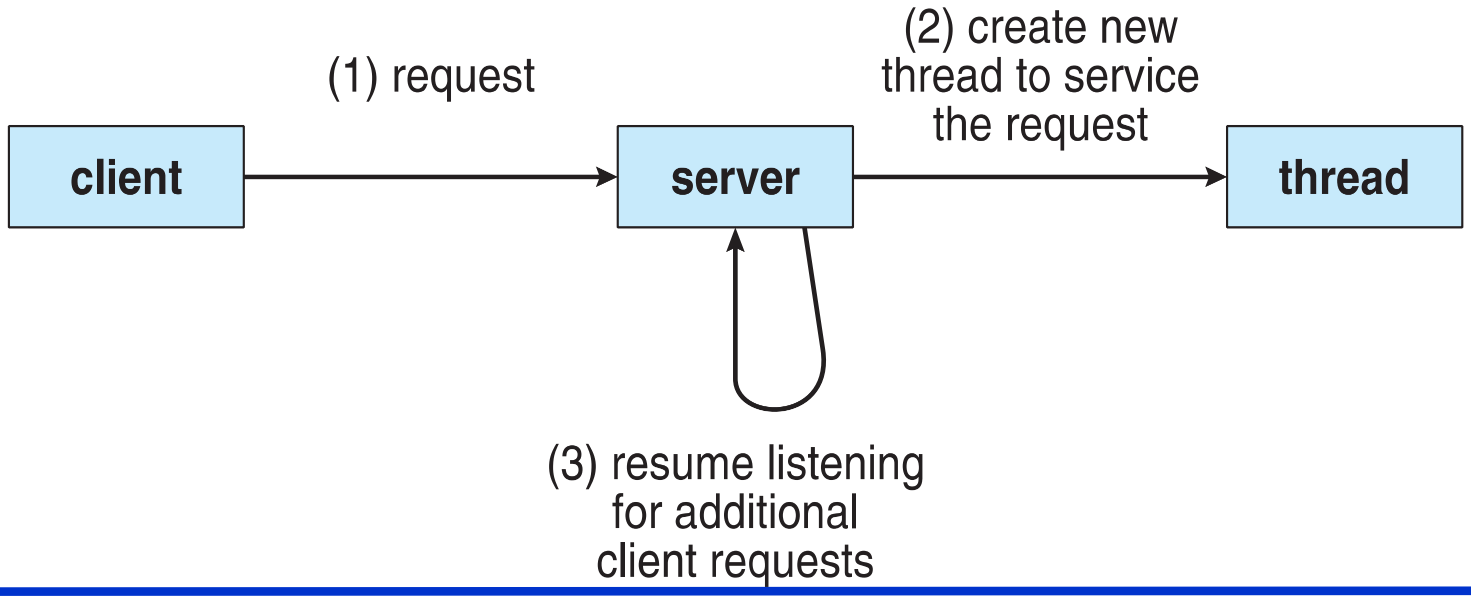
(1) request

server

(2) create new process to service the request

process

(3) resume listening for additional client requests



client

(1) request

server

(2) create new thread to service the request

thread

(3) resume listening for additional client requests

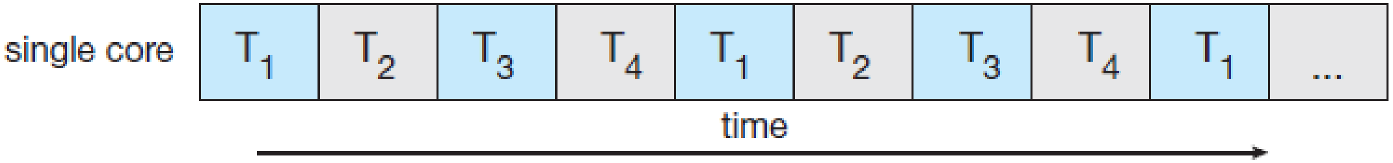
# Concurrent Servers: Multithread Model

- Using threads
  - ✓ We can create a new thread for each request

```
webserver ()
{
    While (1) {
        int sock = accept();
        thread_fork (handle_request, sock);
    }
}
handle_request (int sock)
{
    /* Process request */
    close (sock);
}
```

# Multicore Programming

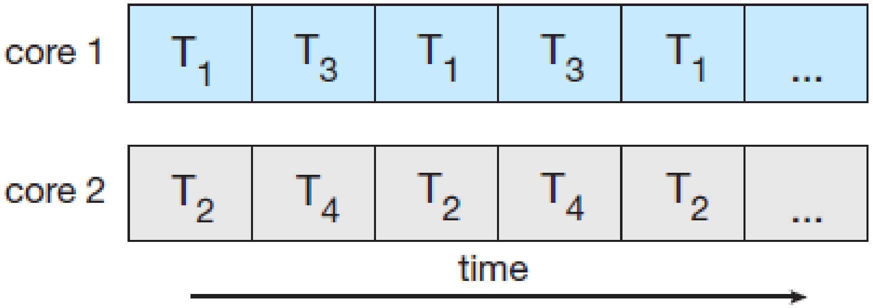
- Concurrent execution on a single-core system



single core $T_1$ $T_2$ $T_3$ $T_4$ $T_1$ $T_2$ $T_3$ $T_4$ $T_1$ ...

time

- Parallel execution on a multicore system



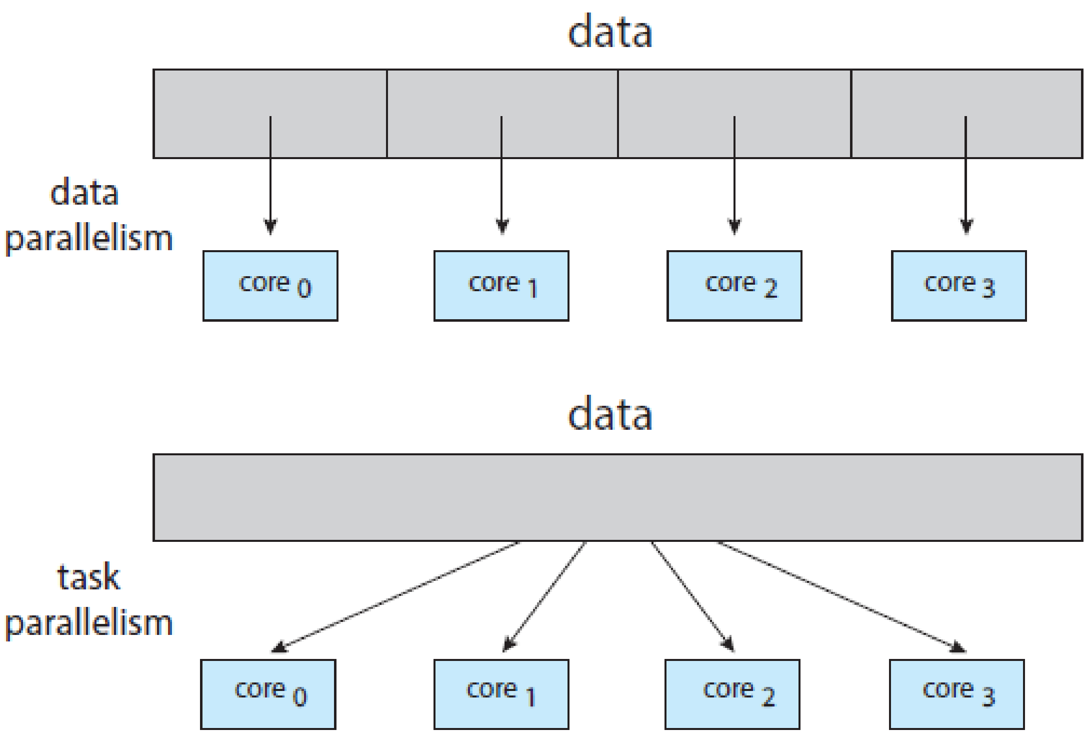
core 1 $T_1$ $T_3$ $T_1$ $T_3$ $T_1$ ...

core 2 $T_2$ $T_4$ $T_2$ $T_4$ $T_2$ ...

time

# Multicore Programming

- Data vs. Task parallelism

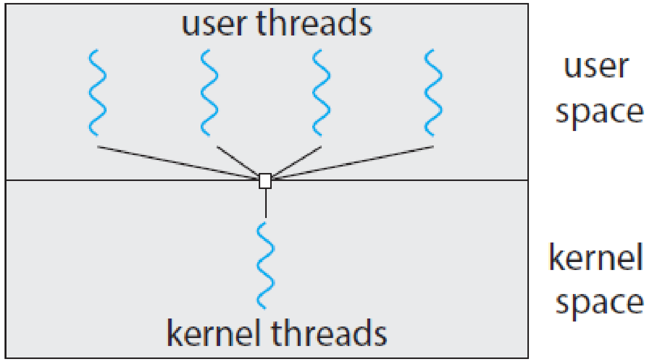


# Parallel Programming

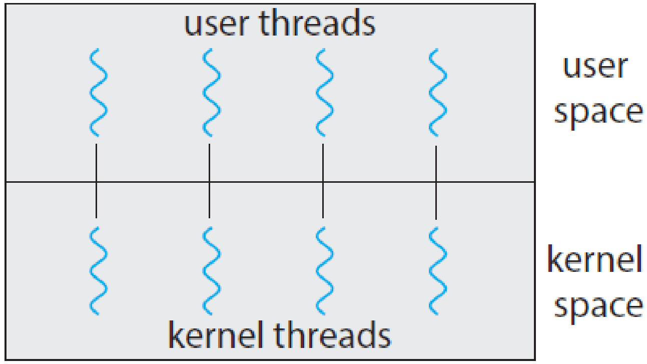
- Pthreads (POSIX threads)
- OpenMP (Open Multi-Processing)
- Open MPI (Message Passing Interface)
- SIMD (Single Instruction Multiple Data)
- GPGPU (General Purpose computing on GPUs)
  - ✓ CUDA (Compute Unified Device Architecture)
  - ✓ OpenCL (Open Computing Language)

# Multithreading Models

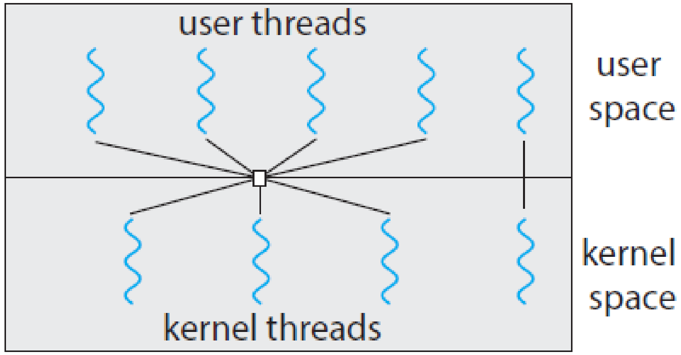
- Many-to-One

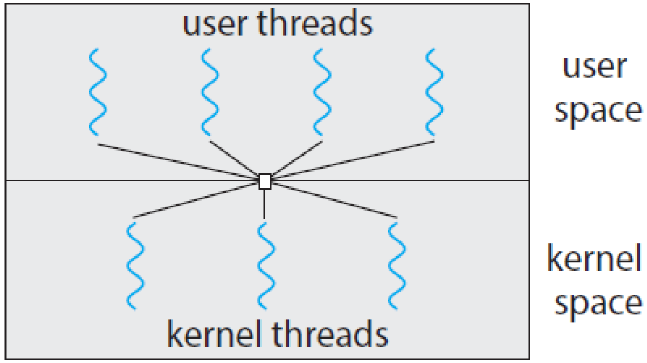


- One-to-One



- Many-to-Many
- Two-level
  - ✓ Many-to-Many + One-to-One





# Pthreads (POSIX threads)

- Thread creation/termination

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

```
void pthread_exit   (void *retval);
```

```
int pthread_join     (pthread_t tid,
                       void **thread_return);
```

# Pthreads

- Mutexes

```
int pthread_mutex_init
                  (pthread_mutex_t *mutex,
                   const pthread_mutexattr_t *mattr);
```

```
int pthread_mutex_destroy
                  (pthread_mutex_t *mutex);
```

```
int pthread_mutex_lock
                  (pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock
                  (pthread_mutex_t *mutex);
```

# Pthreads

- Condition variables

```
int pthread_cond_init
                    (pthread_cond_t *cond,
                     const pthread_condattr_t *cattr);
```

```
int pthread_cond_destroy
                    (pthread_cond_t *cond);
```

```
int pthread_cond_wait
                    (pthread_cond_t *cond,
                     pthread_mutex_t *mutex);
```

```
int pthread_cond_signal
                    (pthread_cond_t *cond);
```

```
int pthread_cond_broadcast
                    (pthread_cond_t *cond);
```

# Windows Threads

- Thread creation/termination

```
HANDLE CreateThread (lpThreadAttributes, dwStackSize,
                      lpStartAddress, lpParameter,
                      dwCreationFlags, lpThreadId);
```

```
void ExitThread   (dwExitCode);
```

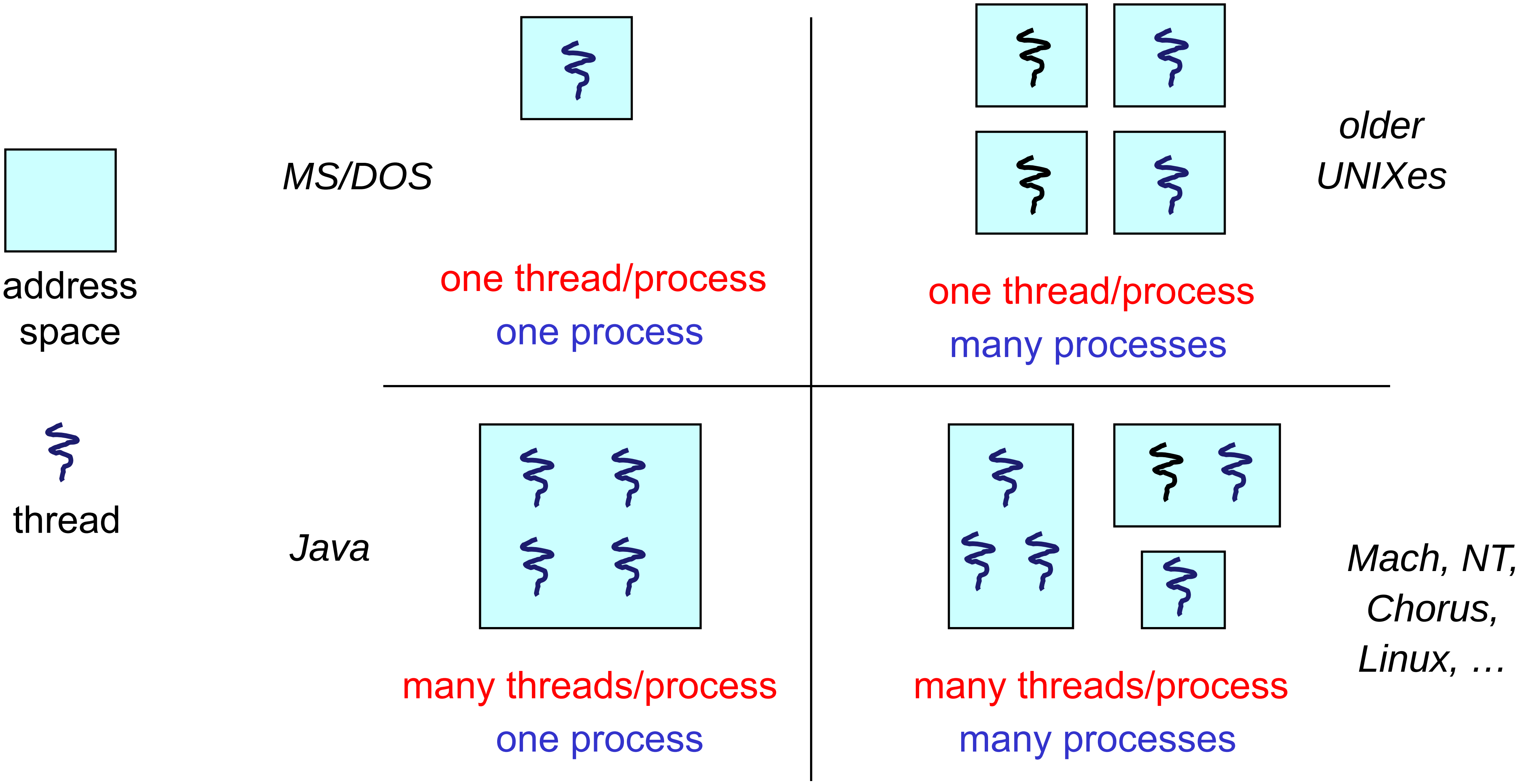
# Java Threads

- Thread creation/termination

Create a new class derived from **Thread** class
Override run() method

Create a new class that implements the **runnable** interface

# Threads Design Space



address space

thread

MS/DOS

one thread/process

one process

older UNIXes

one thread/process

many processes

Java

many threads/process

one process

Mach, NT, Chorus, Linux, …

many threads/process

many processes