Operating System

Ch04: Thread & Concurrency

BeomSeok Kim
Department of Computer Engineering
KyungHee University
passion0822@khu.ac.kr

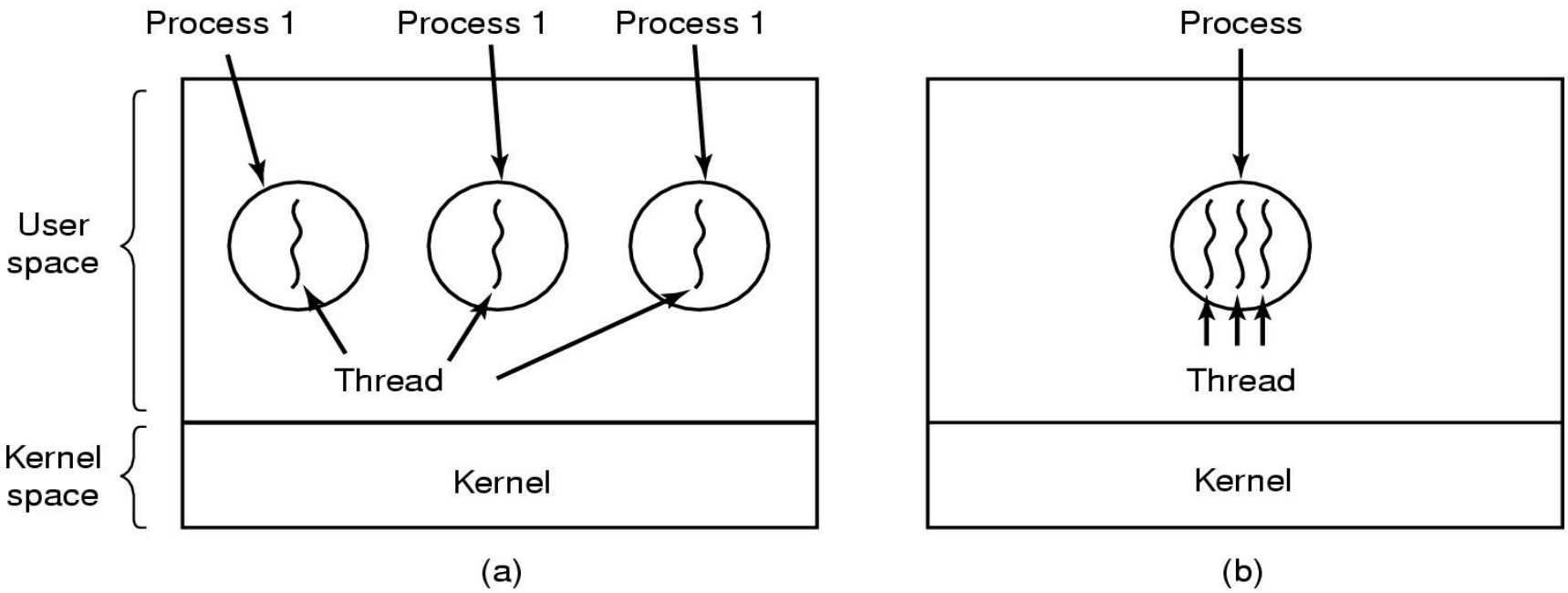
Process

- 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

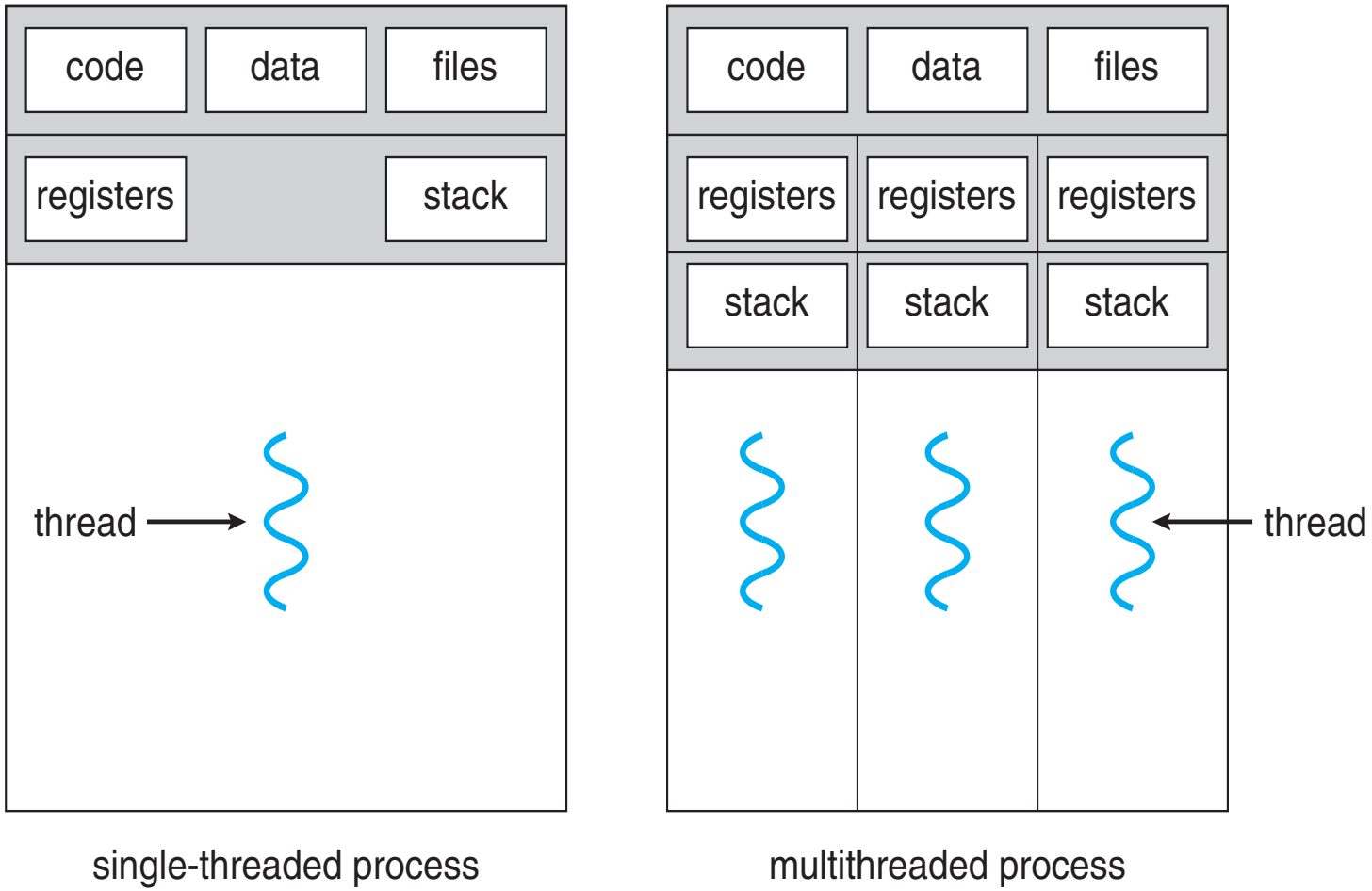
- Single-threaded process

```
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, …);
   …
}
```

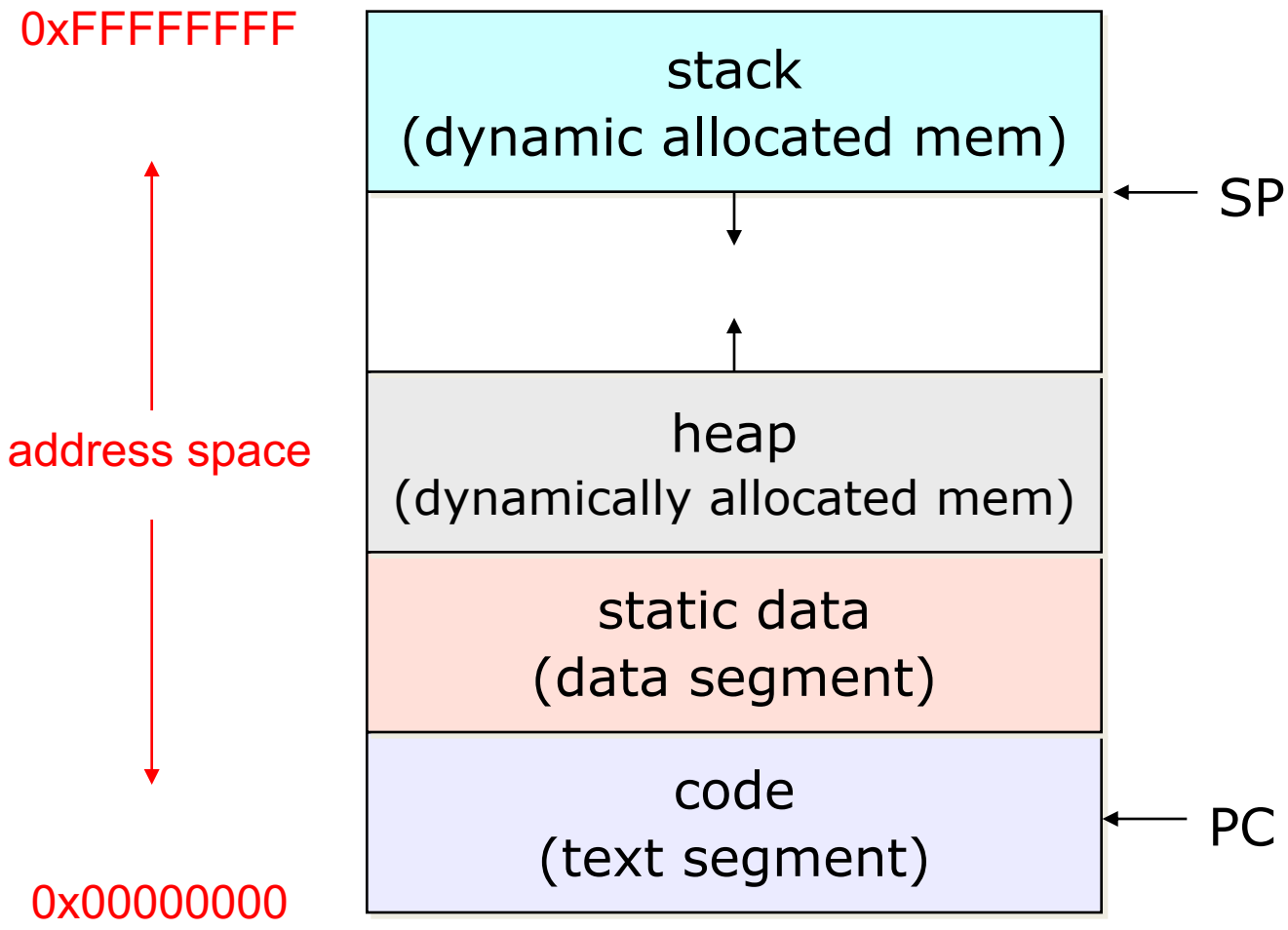
Single and Multithreaded Processes



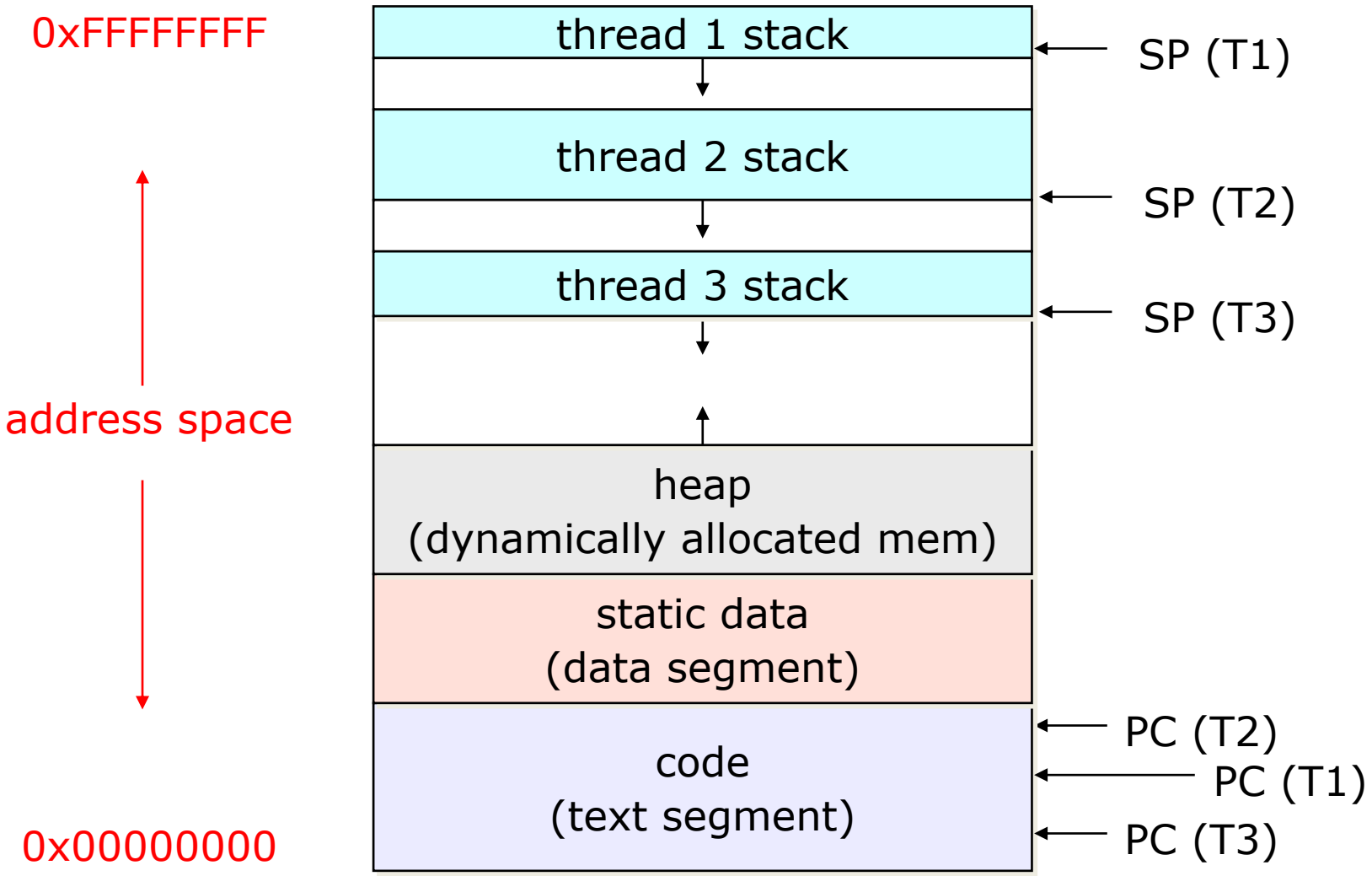
single-threaded process

multithreaded process

Revisited: Process Address Space



Address Space with Threads

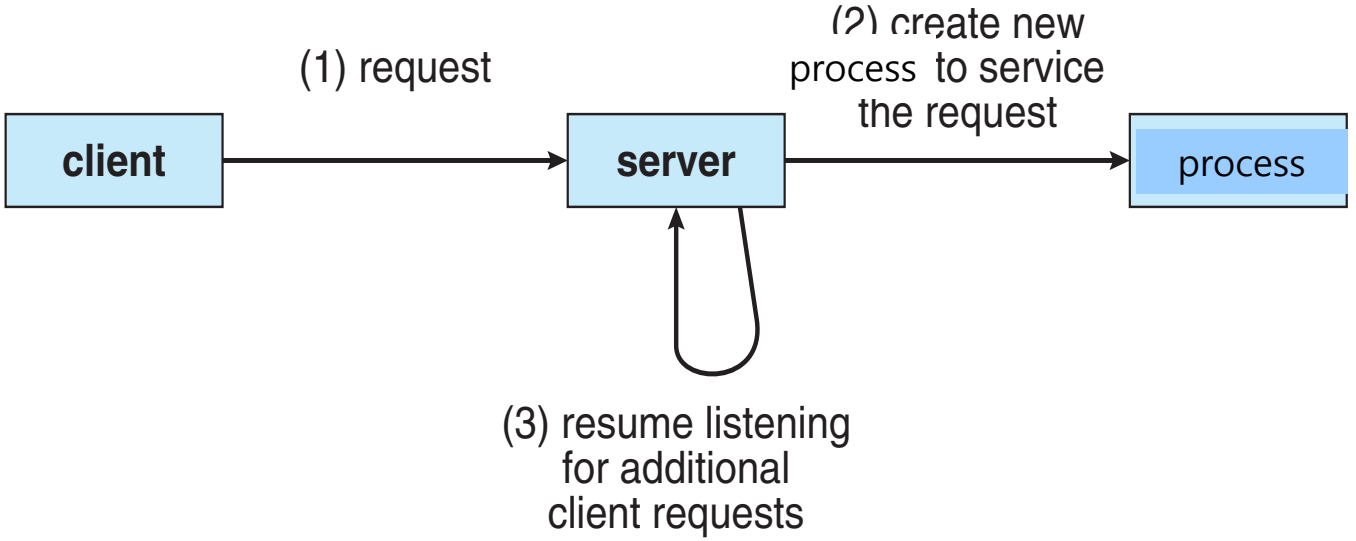


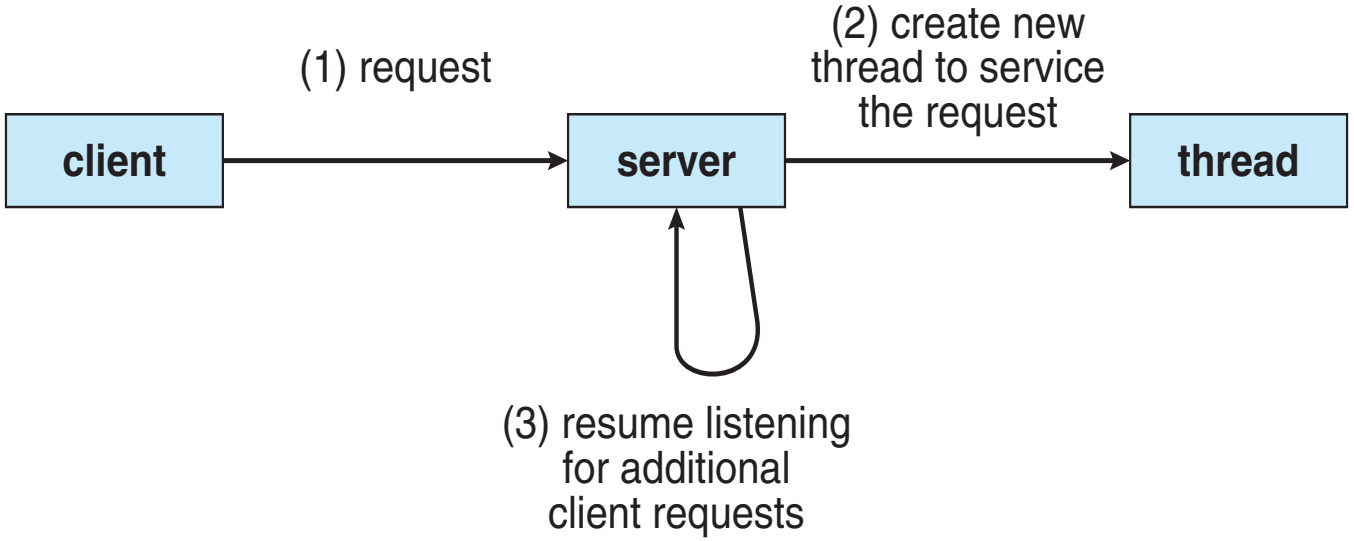
Concurrent Servers: Multiprocess Model

- 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





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);
}
```

Single-Process (Iteration)

```
#include <stdio.h>
#define  MAX_CMD  256

void DoCmd(char *cmd)
{
    printf("New command: %s\n", cmd);
    sleep(1);
    printf("Done\n");
}

int main()
{
    char  cmd[MAX_CMD];

    while (1)  {
      printf("CMD> ");
      fgets(cmd, MAX_CMD, stdin);
      if (cmd[0] == 'q')
          break;

      DoCmd(cmd);
    }

    return 0;
}
```

Multi-Processes

```
#include <stdio.h>
#define  MAX_CMD  256

void DoCmd(char *cmd)
{
    printf("New command: %s\n", cmd);
    sleep(1); printf("Done\n");
    exit(0);
}
int main()
{
    char    cmd[MAX_CMD]; int  pid;
    while (1)  {
      printf("CMD> ");
      fgets(cmd, MAX_CMD, stdin);
      if (cmd[0] == 'q') break;
      if ((pid = fork()) == 0)  {
          DoCmd(cmd);
      }
#if 1
      else  { wait(pid); }
#endif
    }
    return 0;
}
```

Multi-Threads

```
#include <stdio.h>
#include <pthread.h>
#define  MAX_CMD  256

void DoCmd(char *cmd)
{
    printf("New command: %s\n", cmd);
    sleep(1); printf("Done\n");
    pthread_exit(NULL);
}
int main()
{
    char cmd[MAX_CMD]; pthread_t tid;
    while (1)  {
      printf("CMD> ");
      fgets(cmd, MAX_CMD, stdin);
      if (cmd[0] == 'q') break;
      pthread_create(&tid, NULL,(void *)DoCmd, (void *)cmd);
#if 1
      pthread_join(tid, NULL);
#endif
    }
    return 0;
}
```

Makefile

```
CC = gcc
CFLAGS =
LDFLAGS =
LIB = -lpthread
OBJ1 = iteration.o
OBJ2 = process.o
OBJ3 = thread.o
TARGET = iteration process thread

.SUFFIXES: .c .o
.c.o:
    $(CC) $(CFLAGS) -c $<

default: $(TARGET)
iteration: $(OBJ1)
    $(CC) -o $@ $(LDFLAGS) $(OBJ1)
process: $(OBJ2)
    $(CC) -o $@ $(LDFLAGS) $(OBJ2)
thread: $(OBJ3)
    $(CC) -o $@ $(LDFLAGS) $(OBJ3) $(LIB)
clean:
    rm -f *.o $(TARGET)
```

Two Programs on Arduino

```
#define LED 5
void setup() {
    pinMode(LED, OUTPUT);
}
void loop() {
    digitalWrite(LED, HIGH);
    delay(500);
    digitalWrite(LED, LOW);
    delay(500);
}
```

```
#define BUZZER 6
enum { DO=262, RE=294, MI=330, FA=349 };
int Num = 4;
int Frequency[] = { DO, RE, MI, FA };
int Delay[] = { 300, 300, 300,  300 };
void setup() {
    pinMode(BUZZER, OUTPUT);
}
void loop() {
    for(int i=0; i<Num; i++) {
          tone(BUZZER, Frequency[i]);
          delay(Delay[i]);
     }
}
```

Single-Task on Arduino

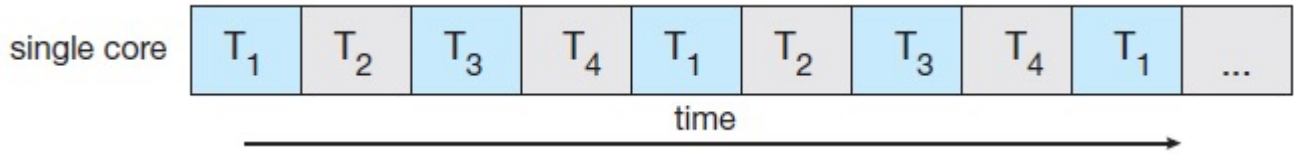
```
#define LED    5
#define BUZZER 6
enum { DO=262, RE=294, MI=330, FA=349 };

int Num = 4;
int Frequency[] = { DO, RE, MI, FA };
int Delay[] = { 300, 300, 300,  300 };

void setup() {
    pinMode(LED, OUTPUT);
    pinMode(BUZZER, OUTPUT);
}

void loop() {









}
```

Multi-Tasks on Arduino

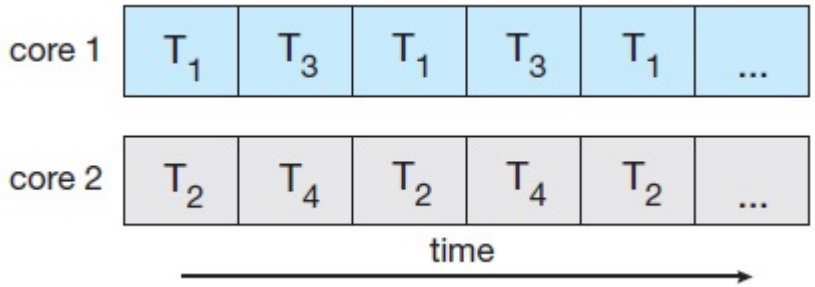
```
void LedTask() {
    while (1)  {
      digitalWrite(LED, HIGH);
      delay(500);
      digitalWrite(LED, LOW);
      delay(500);
    }
}
void BuzzerTask() {
    while (1)  {
      for(int i=0; i<Num; i++) {
          tone(BUZZER, Frequency[i]);
          delay(Delay[i]);
      }
    }
}
void setup() {
    xTaskCreate(LedTask, NULL, 200, NULL, 1,                    NULL);
    xTaskCreate(BuzzerTask, NULL, 200, NULL, 2,                 NULL);
    vTaskStartScheduler();
}
void loop() {
}
```

Multicore Programming

- Concurrent execution on a single-core system

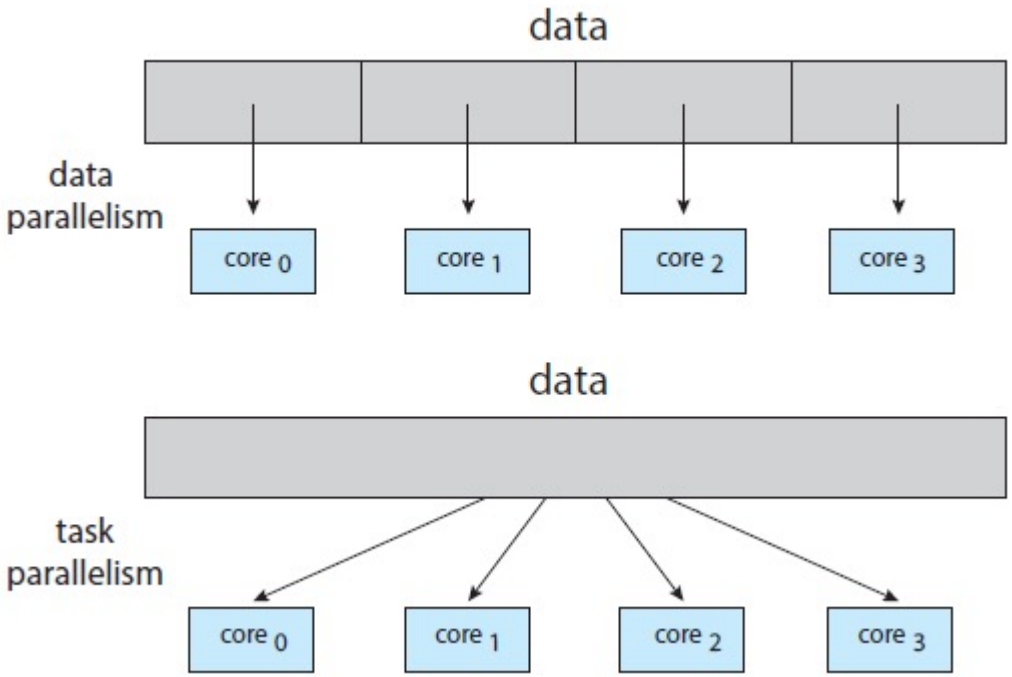


- Parallel execution on a multicore system



Multicore Programming

- Data vs. Task parallelism



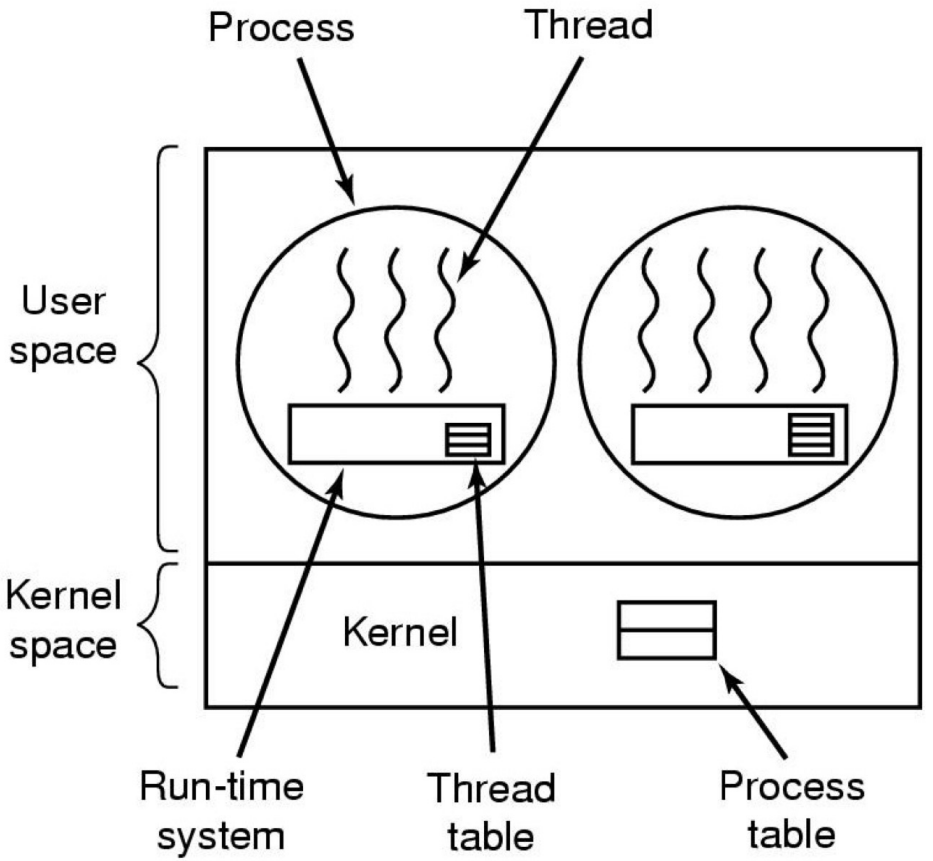
Parallel Programming

- 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)

User Threads

- Thread management done by user-level threads library

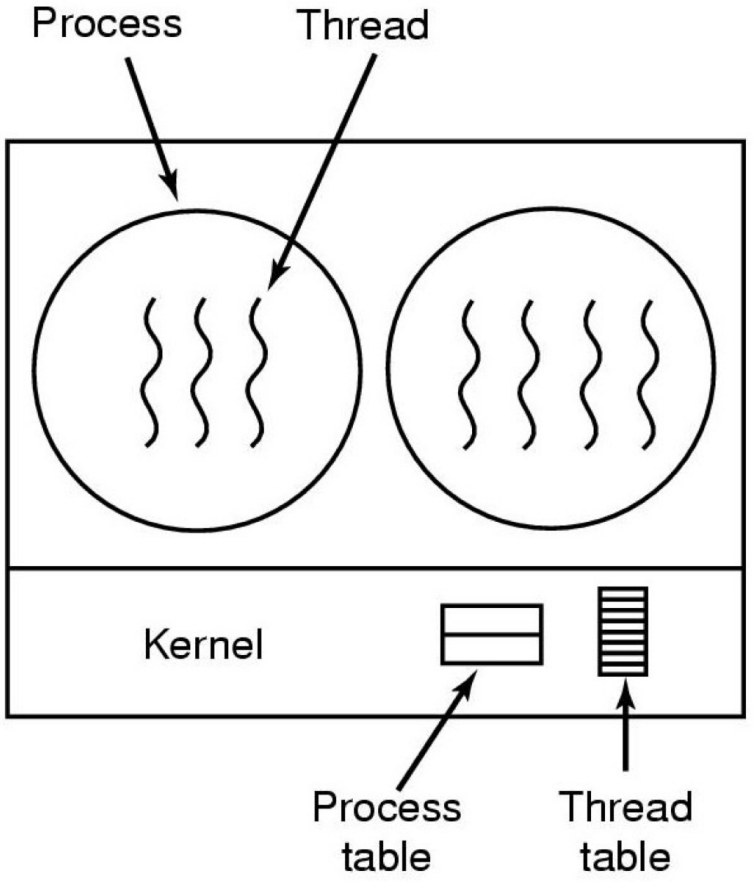
- Example
  - ✓ POSIX *Pthreads*
  - ✓ Mach *C-thread*
  - ✓ Solaris *threads*



Kernel Threads

- Supported by the Kernel
  - ✓ Thread creation and management requires system calls

- Example
  - ✓ Windows 95/98/NT/2000
  - ✓ Solaris
  - ✓ Tru64 UNIX
  - ✓ BeOS
  - ✓ Linux

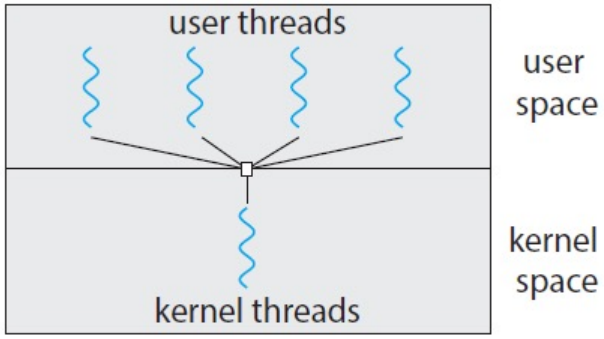


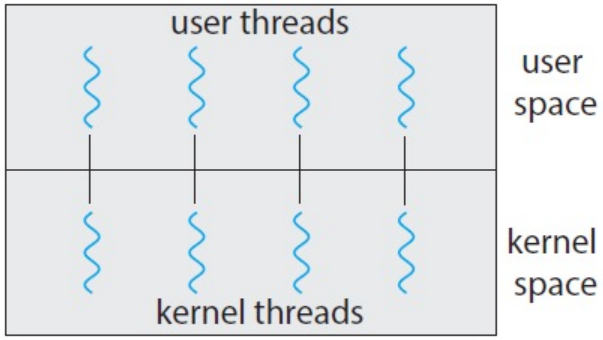
User-level Thread vs. Kernel-level Threads

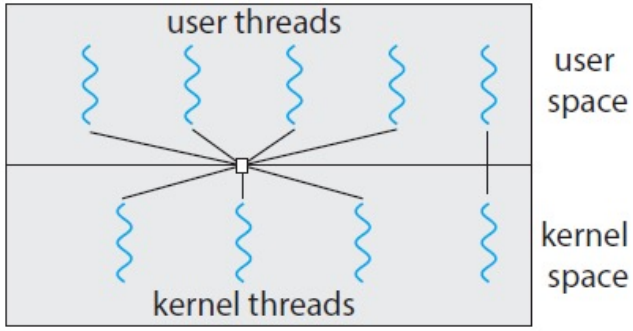
- User-level threads
  - ✓ The user-level threads library implements thread operations
  - ✓ They are small and fast
  - ✓ User-level threads are invisible to the OS
  - ✓ OS may make poor decisions
    - ➢ E.g. bloking I/O
  - ✓ Thread scheduling
    - ➢ Non-preemptive scheduling: yield()
    - ➢ Preemptive scheduling: timer through signal
- Kernel-level threads
  - ✓ All threads operations are implemented in the kernel
  - ✓ The OS schedules all of the threads in a system
  - ✓ Kernel threads are cheaper than processes
  - ✓ They can still be too expensive

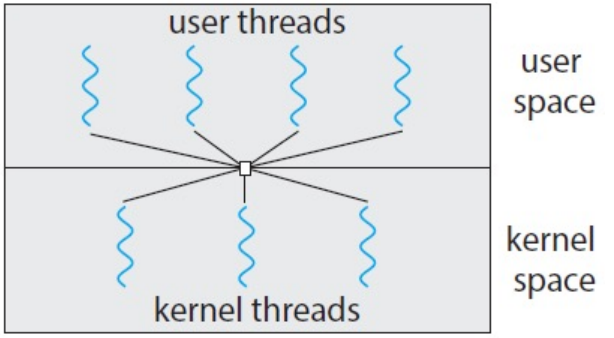
Multithreading Models

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









Threading Issues

- Semantics of fork() and exec() system call
  - ✓ Two versions of fork()
- Thread cancellation
  - ✓ Asynchronous cancellation
  - ✓ Deferred cancellation
- Signal handling
  - ✓ To the thread to which the signal applies
  - ✓ To every thread in the process
  - ✓ To certain threads in the process
  - ✓ Assign a specific thread to receive all signals for the process
- Thread pools
  - ✓ Create a number of threads at process startup
- Thread specific data

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);
```

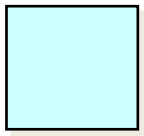
Java Threads

- 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



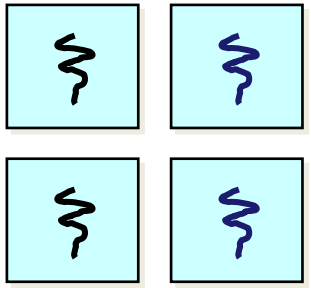
address space

thread

MS/DOS

one thread/process

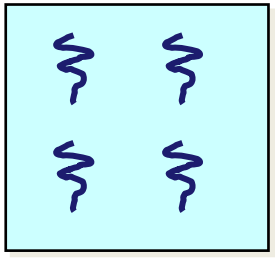
one process



older UNIXes

one thread/process

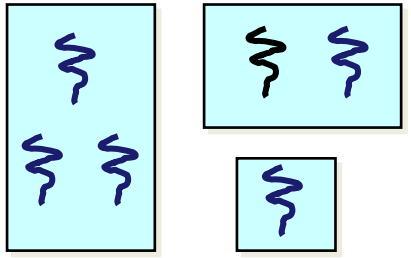
many processes



Java

many threads/process

one process



Mach, NT, Chorus, Linux, …

many threads/process

many processes

Thank You!
Q&A