Chapter 4: Threads

Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

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

- To explore several strategies that provide implicit threading

- To examine issues related to multithreaded programming

- To cover operating system support for threads in Windows and Linux

Threads

- A thread is a basic unit of CPU utilization;

- it comprises a thread ID, a program counter (PC), a register set, and a stack.

- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

- A traditional process has a single thread of control.

- If a process has multiple threads of control, it can perform more than one task at a time.
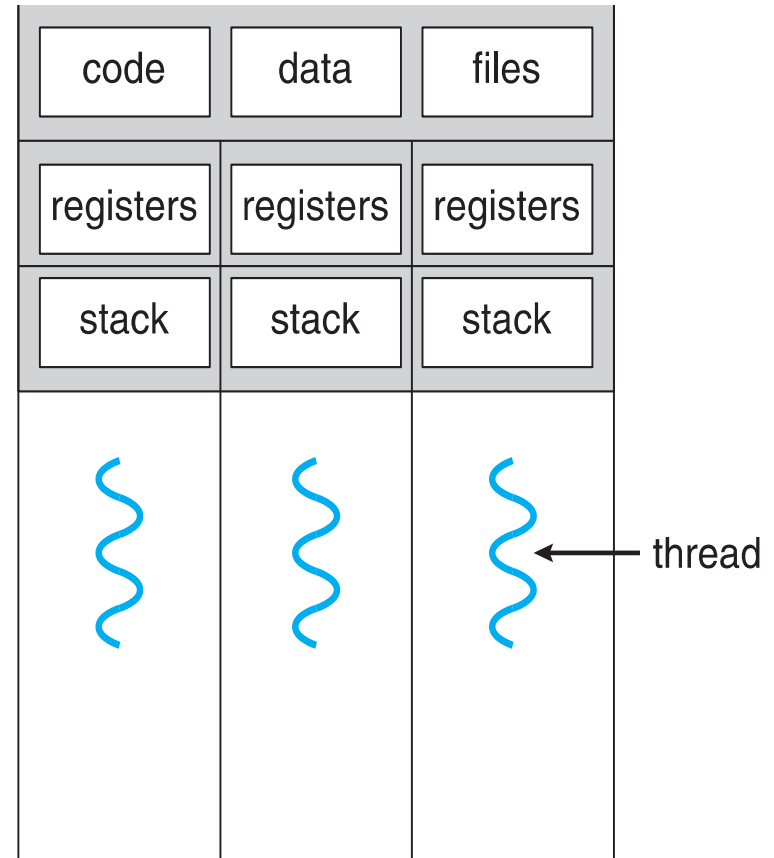
Single and Multithreaded Processes

code	data	files
registers		stack
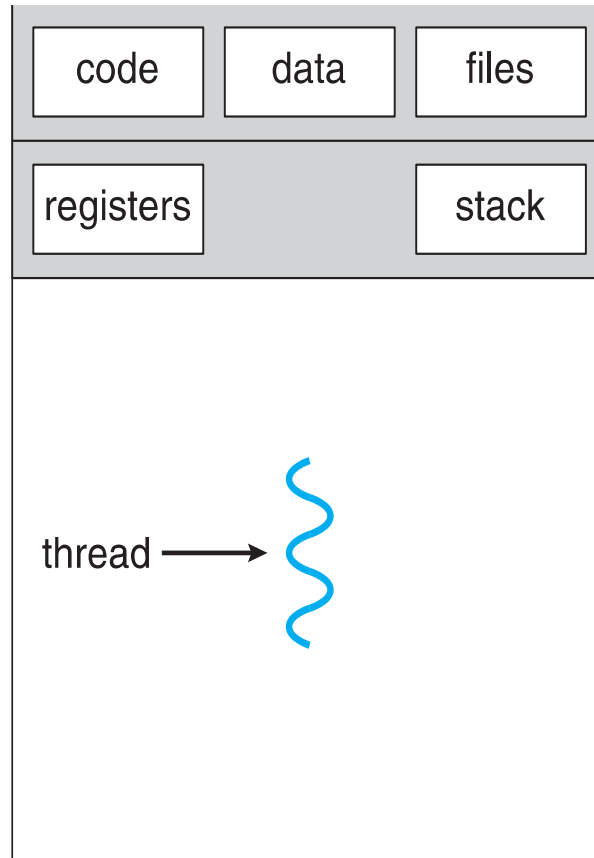
thread →

single-threaded process

code	data	files
registers	registers	registers
stack	stack	stack

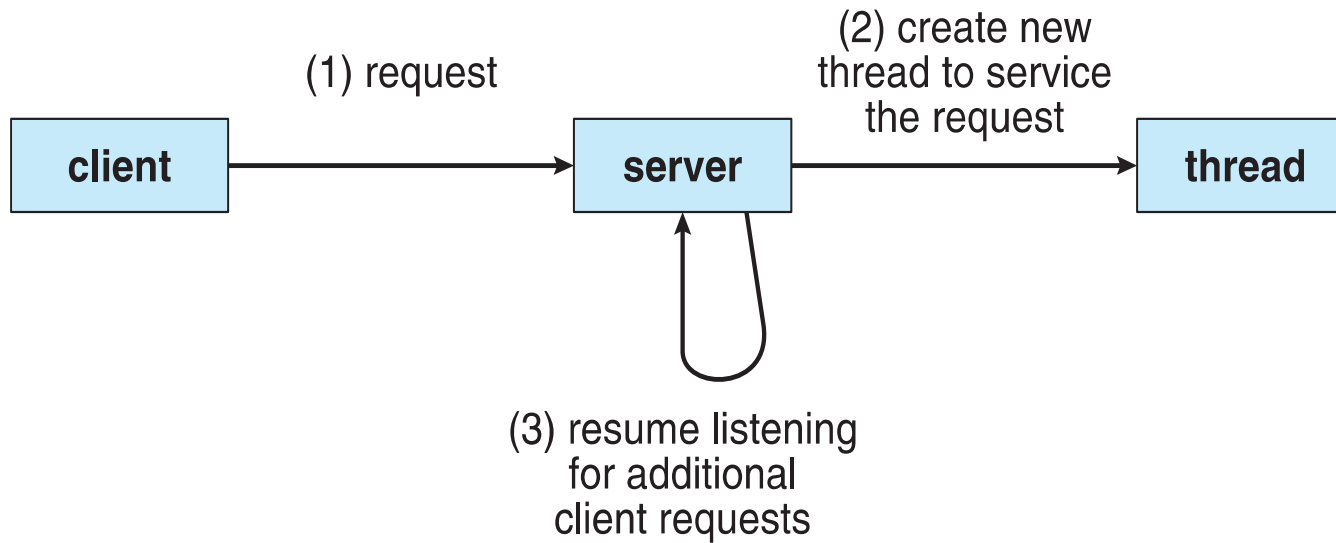← thread

multithreaded process

Motivation

- Most modern applications are multithreaded

- Threads run within application

- Multiple tasks with the application can be implemented by separate threads

  - Update display

  - Fetch data

  - Spell checking

  - Answer a network request

- Process creation is heavy-weight while thread creation is light-weight

- Kernels are generally multithreaded

Multithreaded Server Architecture

(1) request

(2) create new
thread to service
the request

client → server → thread

(3) resume listening
for additional
client requests

Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces

- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing

- **Economy –** cheaper than process creation, thread switching lower overhead than context switching

- **Scalability –** process can take advantage of multiprocessor architectures
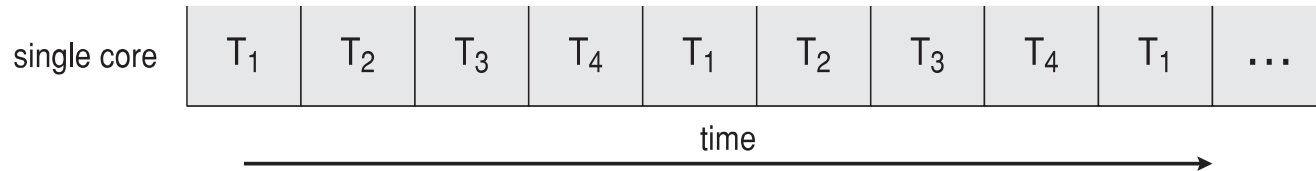
Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- *Parallelism* implies a system can perform more than one task simultaneously
- *Concurrency* supports more than one task making progress
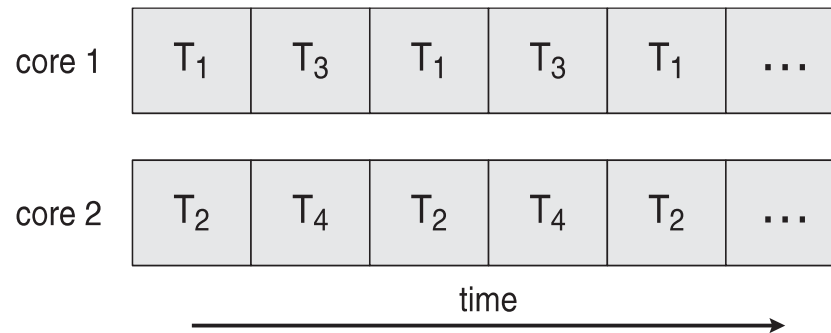 - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

| single core | T_1 | T_2 | T_3 | T_4 | T_1 | T_2 | T_3 | T_4 | T_1 | ... |

time →

- **Parallelism on a multi-core 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 (Cont.)

- Types of parallelism

  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each

  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
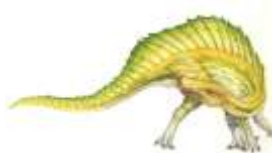
Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components

- *S* is serial portion

- *N* number of (threads)

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library

- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads

- **Kernel threads** - Supported by the Kernel

- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- Many-to-One

- One-to-One

- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread

- One thread blocking causes all to block

- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time

- Few systems currently use this model

- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**
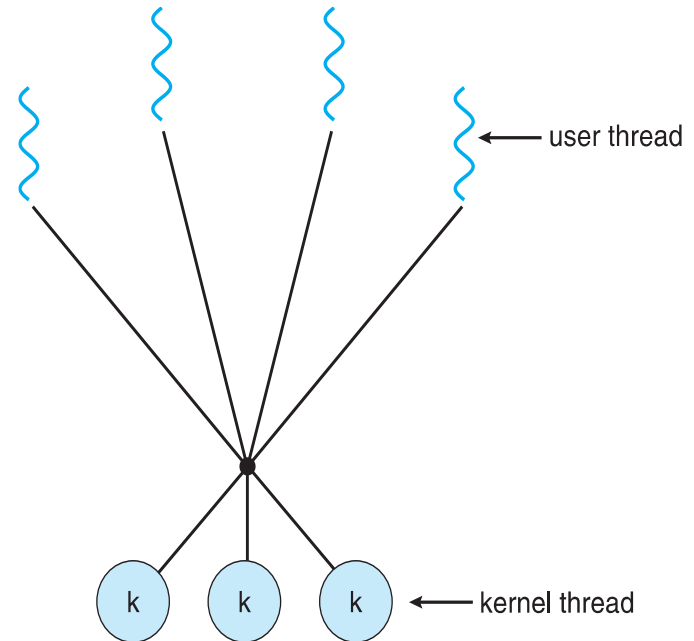

user thread

kernel thread

k
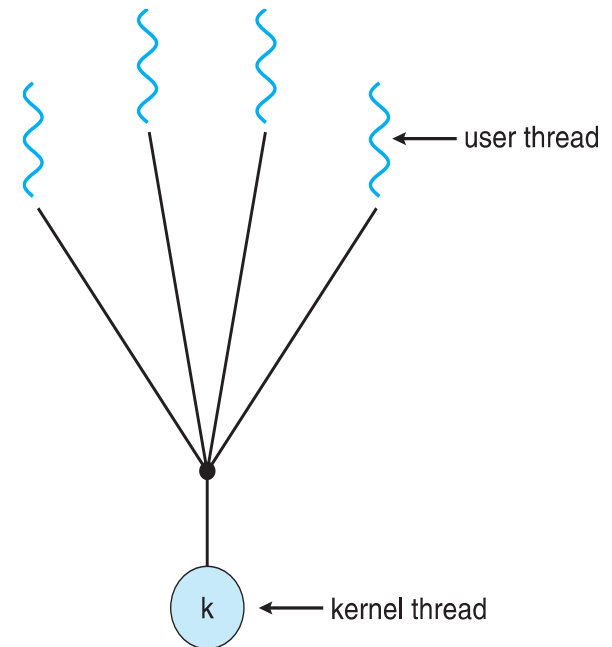
One-to-One

- Each user-level thread maps to kernel thread

- Creating a user-level thread creates a kernel thread

- More concurrency than many-to-one

- Number of threads per process sometimes restricted due to overhead

- Examples
  - Windows
  - Linux
  - Solaris 9 and later

← user thread
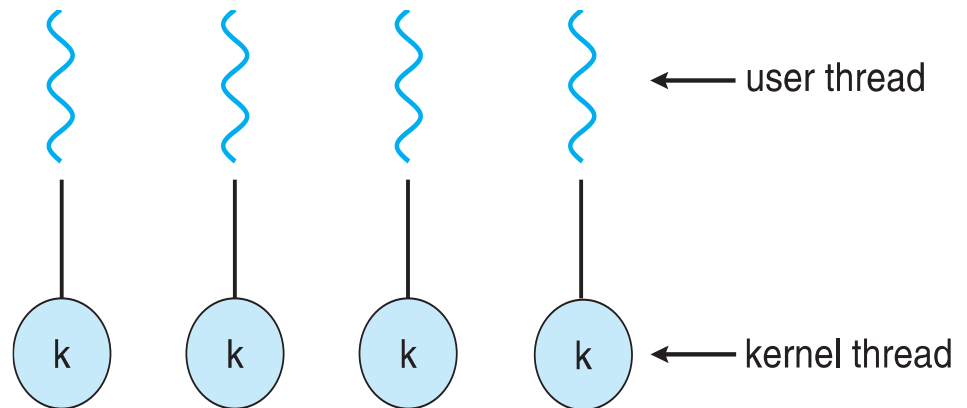
k   k   k   k   ← kernel thread

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads

- Allows the operating system to create a sufficient number of kernel threads

- Solaris prior to version 9

- Windows with the *ThreadFiber* package

Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

- Examples
    - IRIX
    - HP-UX
    - Tru64 UNIX
    - Solaris 8 and earlier

← user thread

k k k   k ← kernel thread
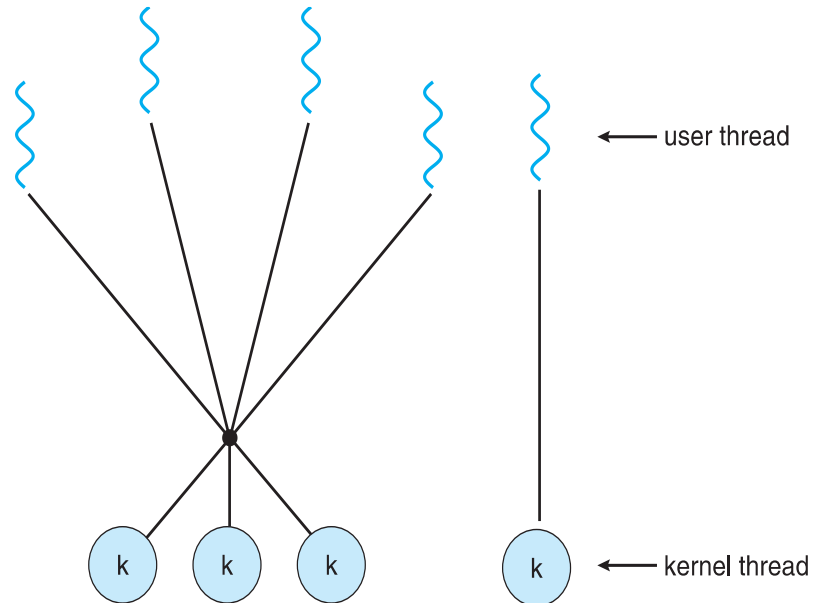
Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads Example

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

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

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;
   }
```

```c
/* 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);
}


/* 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);
}
```

```
#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);
```

Thread Pools

- Create a number of threads in a pool where they await work

- Advantages:

  - Usually slightly faster to service a request with an existing thread than create a new thread

  - Allows the number of threads in the application(s) to be bound to the size of the pool

  - Separating task to be performed from mechanics of creating task allows different strategies for running task

    - i.e.Tasks could be scheduled to run periodically

- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
     * this function runs as a separate thread.
     */
}
```

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.

- A **signal handler** is used to process signals
    1. Signal is generated by particular event
    2. Signal is delivered to a process
    3. Signal is handled by one of two signal handlers:
        1. default
        2. user-defined

- Every signal has **default handler** that kernel runs when handling signal

    - **User-defined signal handler** can override default
    - For single-threaded, signal delivered to process

Signal Handling (Cont.)

n   Where should a signal be delivered for multi-threaded?

- l   Deliver the signal to the thread to which the signal applies

- l   Deliver the signal to every thread in the process

- l   Deliver the signal to certain threads in the process

- l   Assign a specific thread to receive all signals for the process

Thread Cancellation

- Terminating a thread before it has finished

- Thread to be canceled is **target thread**

- Two general approaches:

  - **Asynchronous cancellation** terminates the target thread immediately

  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

.  .  .

/* cancel the thread */
pthread_cancel(tid);
```

Operating System Examples

- Windows Threads
- Linux Threads

Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7

- Implements the one-to-one mapping, kernel-level

- Each thread contains

  - A thread id

  - Register set representing state of processor

  - Separate user and kernel stacks for when thread runs in user mode or kernel mode

  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)

- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (Cont.)

- The primary data structures of a thread include:

  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space

  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space

  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows Threads Data Structures
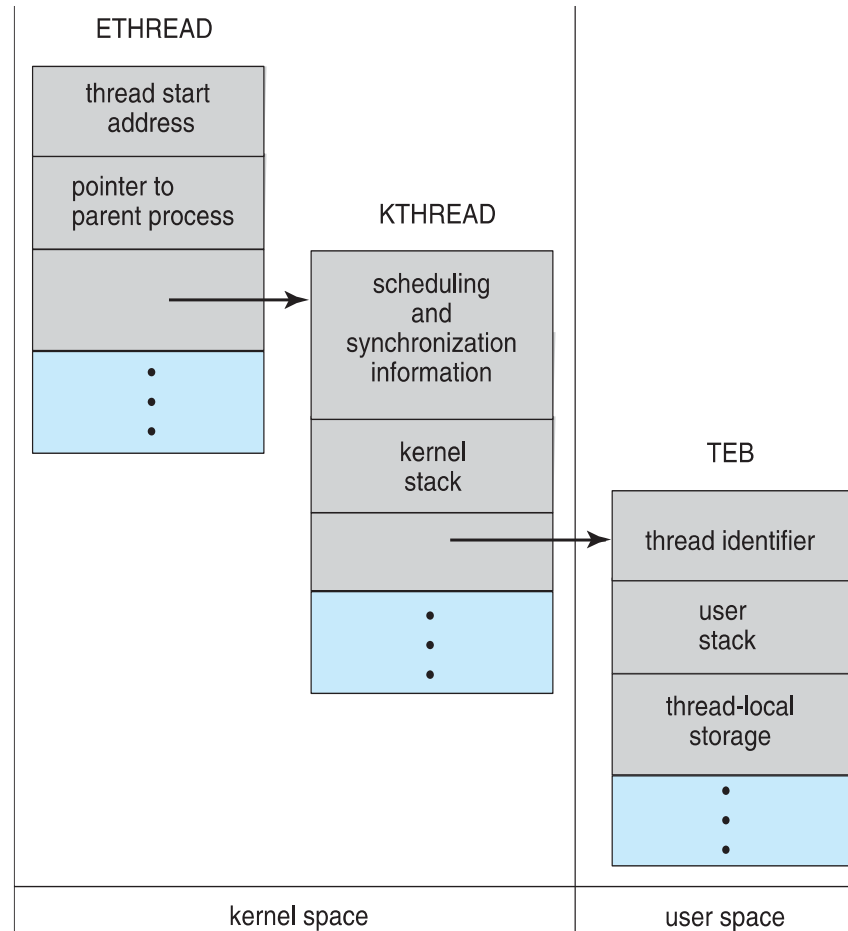
Linux Threads

- Linux refers to them as *tasks* rather than *threads*

- Thread creation is done through `clone()` system call

- `clone()` allows a child task to share the address space of the parent task (process)

  - Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- `struct task_struct` points to process data structures (shared or unique)

End of Chapter 4