Lecture 10
Threads and Concurrency

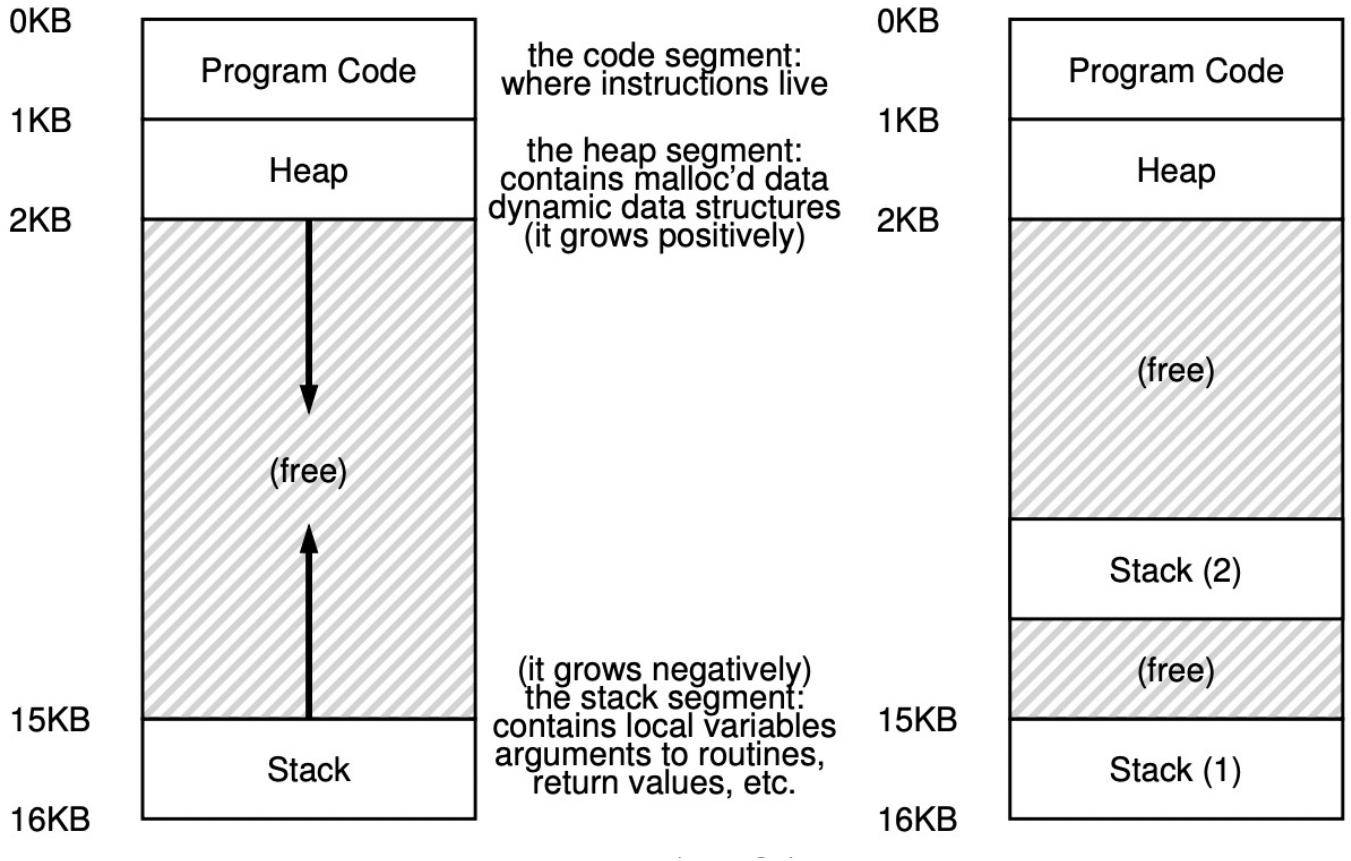
Prof. Yinqian Zhang

Spring 2022

What is a Thread?

- Thread is an **abstraction** of the execution of a program
  - A single-threaded program has one point of execution
  - A multi-threaded program has more than one points of execution
- Each thread has its own **private** execution state
  - Program counter and a private set of registers
  - A private stack for thread-local storage
  - CPU switching from one thread to another requires context switch
- Threads in the same process **share** computing resources
  - Address space, files, signals, etc.

Single-Threaded and Multi-Threaded



Why Use Thread?

- Increase parallelism
  - One thread per CPU makes better use of multiple CPUs to improve efficiency
- Avoid blocking program progress due to slow I/O
  - Threading enables overlap of I/O with other activities within a single program
  - e.g., many modern server-based applications (web servers, database management systems, and the like) make use of threads
- And allow resource sharing !!!

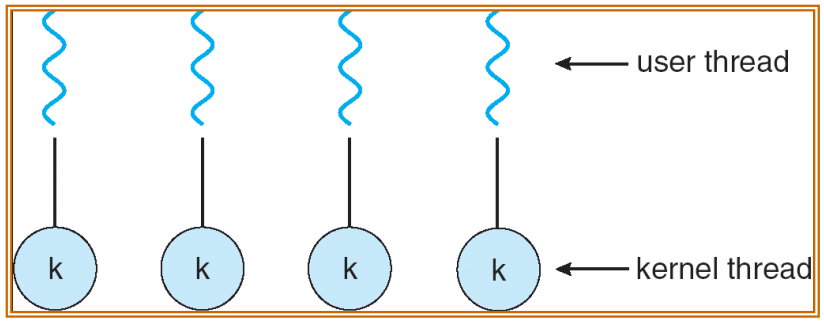
Thread Implementation

- User-level thread
  - Thread management (e.g., creating, scheduling, termination) done by user-level threads library
  - OS does not know about user-level thread

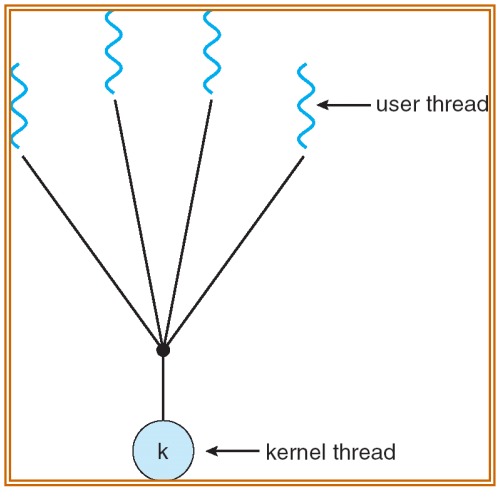
- Kernel-level thread
  - Threat management done by kernel
  - OS is aware of each kernel-level thread

Thread Models

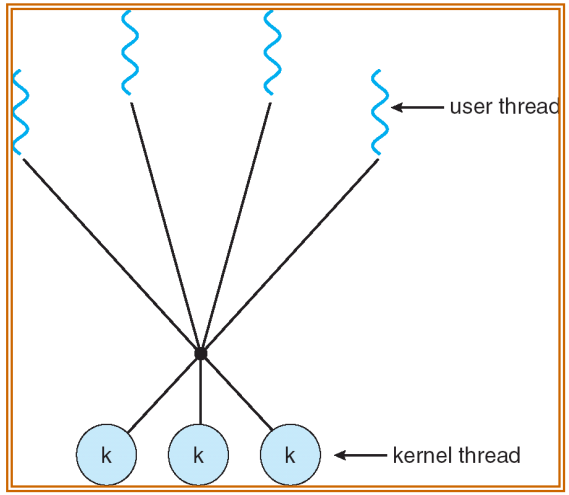
- One-to-one mapping
  - One user-level thread to one kernel-level thread
- Many-to-one mapping
  - Many user-level thread to one kernel-level thread
- Many-to-many mapping
  - Many user-level thread to many kernel-level thread



One-to-One



Many-to-One



Many-to-Many

Pros and Cons

- Many-to-one mapping
  - Pros: context switch between threads is cheap
  - Cons: When one thread blocks on I/O, all threads block
- One-to-one mapping
  - Pros: Every thread can run or block independently
  - Cons: Need to make a crossing into kernel mode to schedule
- Many-to-many mapping
  - Many user-level threads multiplexed on less or equal number of kernel-level threads
  - Pros: best of the two worlds, more flexible
  - Cons: difficult to implement

POSIX Thread

- A POSIX standard API for thread creation and synchronization
  - **Portable Operating System Interface** (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.
  - POSIX APIs are defined for software compatibility on variants of Unix and other operating systems.
  - POSIX APIs are common in UNIX-like operating systems (e.g., Solaris, Linux, Mac OS X)
- The API specifies behavior of the thread library, but the implementation is up to development of the library

Process-wide Attributes

- POSIX thread may be implemented either as user-level or kernel-level
- POSIX thread requires threads of the same process share
  - process ID
  - parent process ID
  - process group ID and session ID
  - user and group IDs
  - open file descriptors
  - current directory
  - resource limits
  - measurements of the consumption of CPU time and resources

Pthread Programming

```c
/* thread.c */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int sum = 0; //shared by all threads
void *runner(void *param) {
        int i, upper = atoi(param);
        for (i=1;i<=upper;i++)
                sum += i;
        printf("PID: %d, PPID: %d\n", (int)getpid(), (int)getppid());
        pthread_exit(0);
}
int main() {
        pthread_t tid; // thread identifier
        pthread_attr_t attr; //set of thread attributes
        pthread_attr_init(&attr); //get the default attributes
    printf("PID: %d, PPID: %d\n", (int)getpid(), (int)getppid());
        /* create the thread */
        pthread_create(&tid, &attr, runner, "10");
        /* wait for the thread to exit */
        pthread_join(tid, NULL);
        printf("sum = %d\n", sum);
}

/* gcc -o test thread.c -lpthread */
```

Thank you!