Thread

Disclaimer: some slides are adopted from the book authors' slides with permission

Recap

- IPC
  - Shared memory
    - share a memory region between processes
    - read or write to the shared memory region
    - fast communication
    - synchronization is very difficult
  - Message passing
    - exchange messages (send and receive)
    - typically involves data copies (to/from buffer)
    - synchronization is easier
    - slower communication

KU THE UNIVERSITY OF KANSAS

Recap

- Process
  - **Address space**
    - The process's view of memory
    - Includes program code, global variables, dynamic memory, stack
  - **Processor state**
    - Program counter (PC), stack pointer, and other CPU registers
  - **OS resources**
    - Various OS resources that the process uses
    - E.g.) open files, sockets, accounting information

Concurrent Programs

- Objects (tanks, planes, …) are moving simultaneously
- Now, imagine you implement each object as a process. Any problems?

Why Processes Are Not Always Ideal?

- Not memory efficient
  - Own address space (page tables)
  - OS resources: open files, sockets, pipes, …
- Sharing data between processes is not easy
  - No direct access to others' address space
  - Need to use IPC mechanisms

KU THE UNIVERSITY OF KANSAS

Better Solutions?

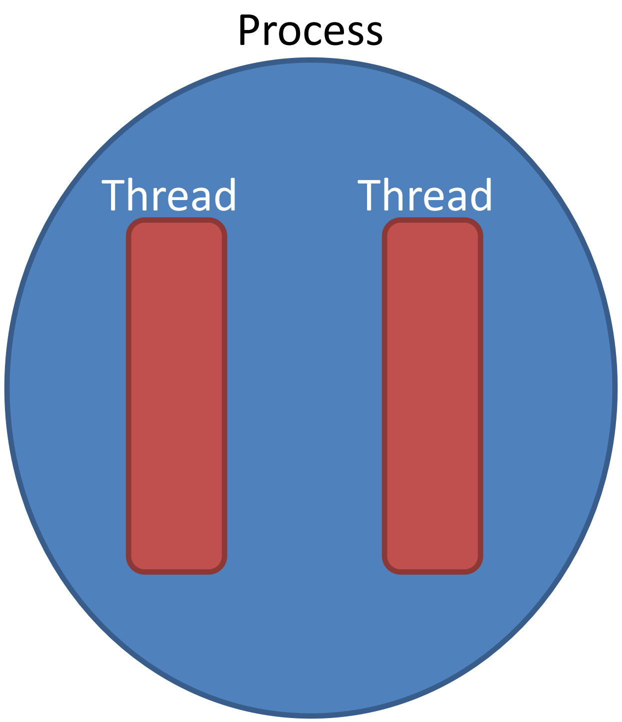
- We want to run things concurrently
  - i.e., multiple independent flows of control
- We want to share memory easily
  - Protection is not really big concern
  - Share code, data, files, sockets, …
- We want do these things efficiently
  - Don’t want to waste memory
  - Performance is very important

Thread

Thread in OS

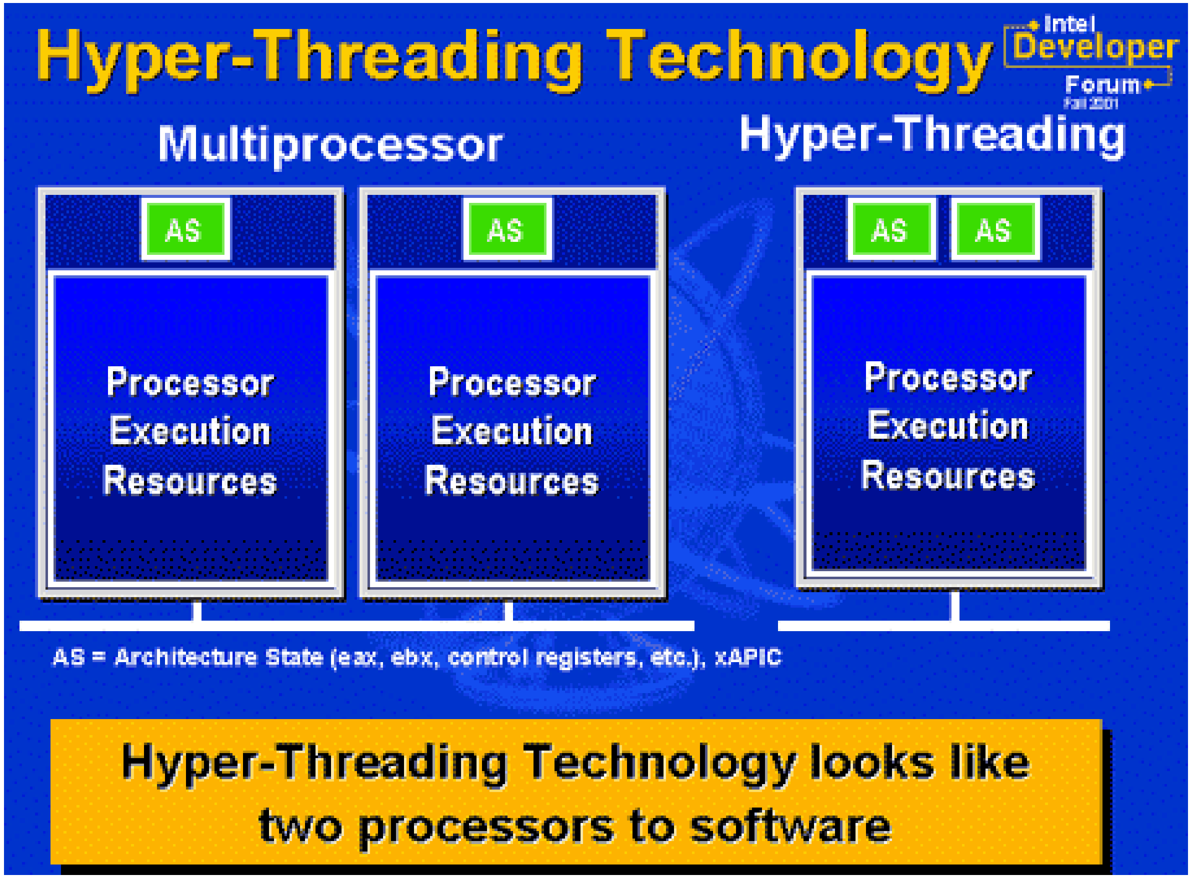
- Lightweight process

- Process
  - Address space
  - CPU context: PC, registers, stack, …
  - OS resources
- Thread
  - ~~Address space~~
  - CPU context: PC, registers, stack, …
  - ~~OS resources~~



Thread in Architecture

- Logical processor



http://www.pcstats.com/articleview.cfm?articleID=1302

Thread

- Lightweight process
  - Own independent flown of control (execution)
  - Stack, thread specific data (tid, …)
  - Everything else (address space, open files, …) is shared

Shared	Private
- Program code - (Most) data - Open files, sockets, pipes - Environment (e.g., HOME)	- Registers - Stack - Thread specific data - Return value

Process vs. Thread

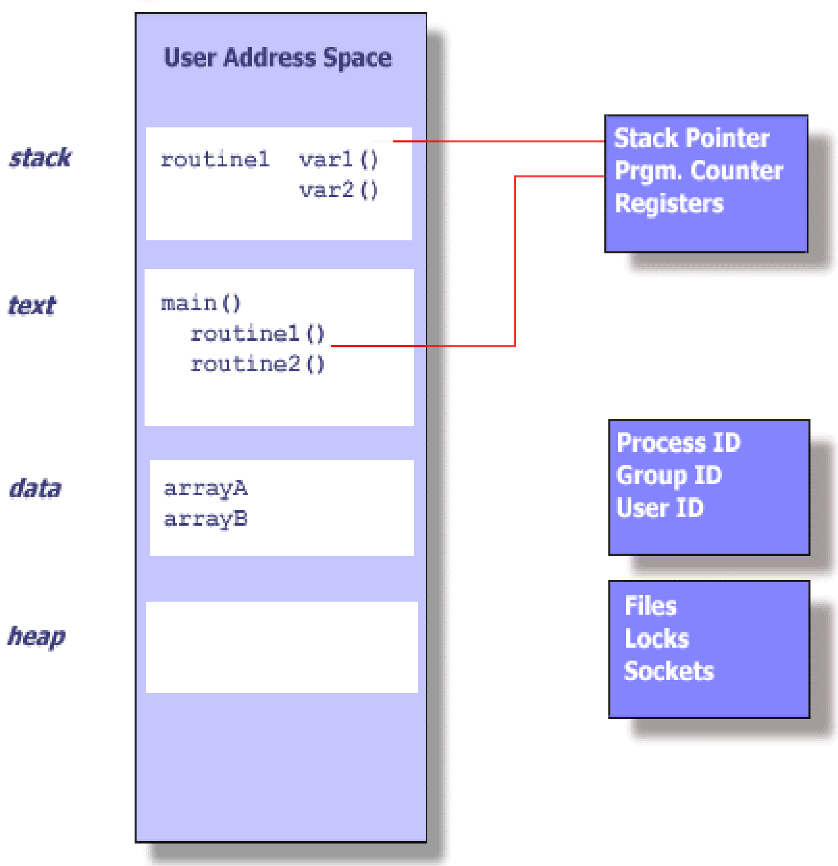


Figure source: https://computing.llnl.gov/tutorials/pthreads/

Process vs. Thread

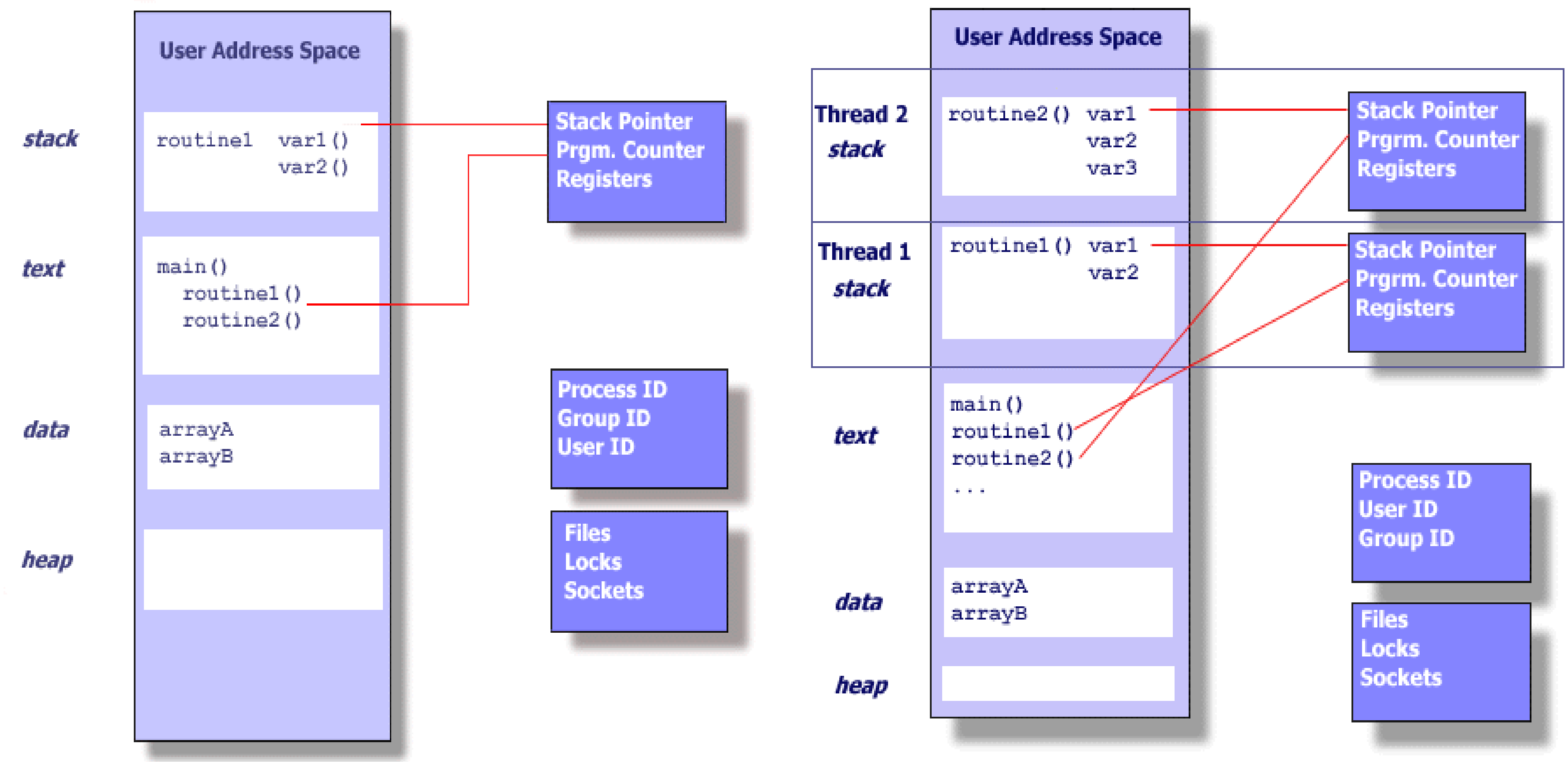


Figure source: https://computing.llnl.gov/tutorials/pthreads/

Thread Benefits

- Responsiveness
  - Simple model for concurrent activities.
  - No need to block on I/O
- Resource Sharing
  - Easier and faster memory sharing (but be aware of synchronization issues)
- Economy
  - Reduces context-switching and space overhead → better performance
- Scalability
  - Exploit multicore CPU

Thread Programming in UNIX

- Pthread
  - IEEE POSIX standard threading API
- Pthread API
  - Thread management
    - create, destroy, detach, join, set/query thread attributes
  - Synchronization
    - Mutexes –lock, unlock
    - Condition variables – signal/wait

Pthread API

- pthread_attr_init – initialize the thread attributes object
  - int pthread_attr_init(pthread_attr_t *attr);
  - defines the attributes of the thread created
- pthread_create – create a new thread
  - int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);
  - upon success, a new thread id is returned in thread
- pthread_join – wait for thread to exit
  - int pthread_join(pthread_t thread, void **value_ptr);
  - calling process blocks until thread exits
- pthread_exit – terminate the calling thread
  - void pthread_exit(void *value_ptr);
  - make return value available to the joining thread

Pthread Example 1

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

int sum; /* data shared by all threads */
void *runner (void  *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for(i=1 ; i<=upper ; i++)
        sum += i;
    pthread_exit(0);
}

int main (int argc, char *argv[])
{
    pthread_t tid; /* thread identifier */
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);
    fprintf(stdout, “sum = %d\n”, sum);
}
```

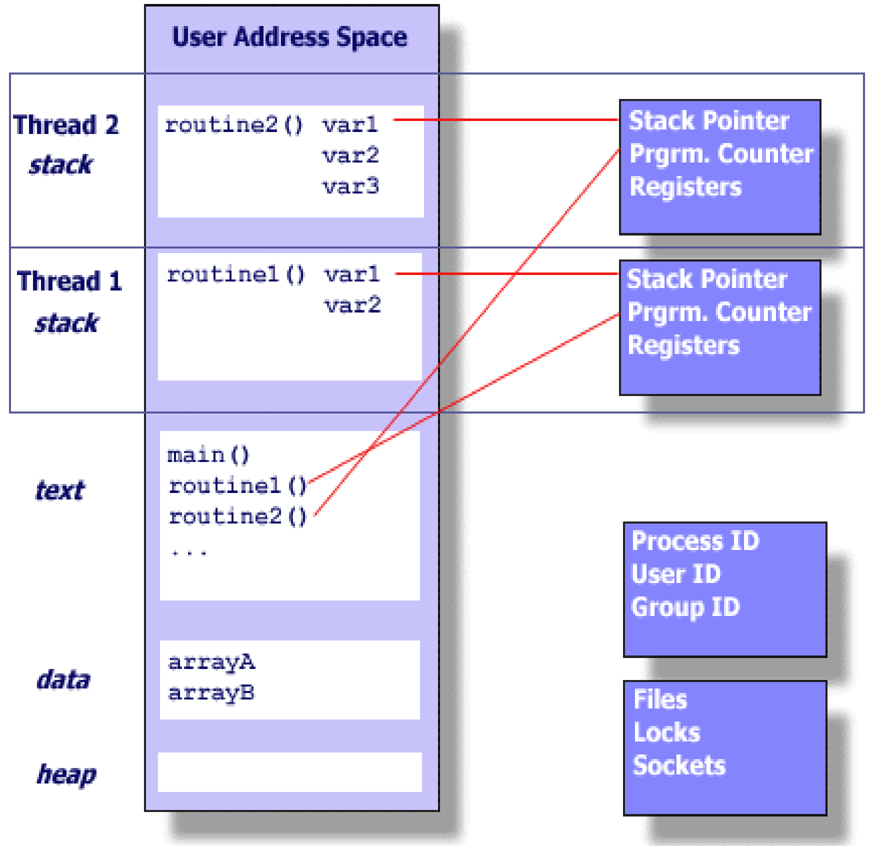
Quiz: Final ouput?

$./a.out 10

sum = 55

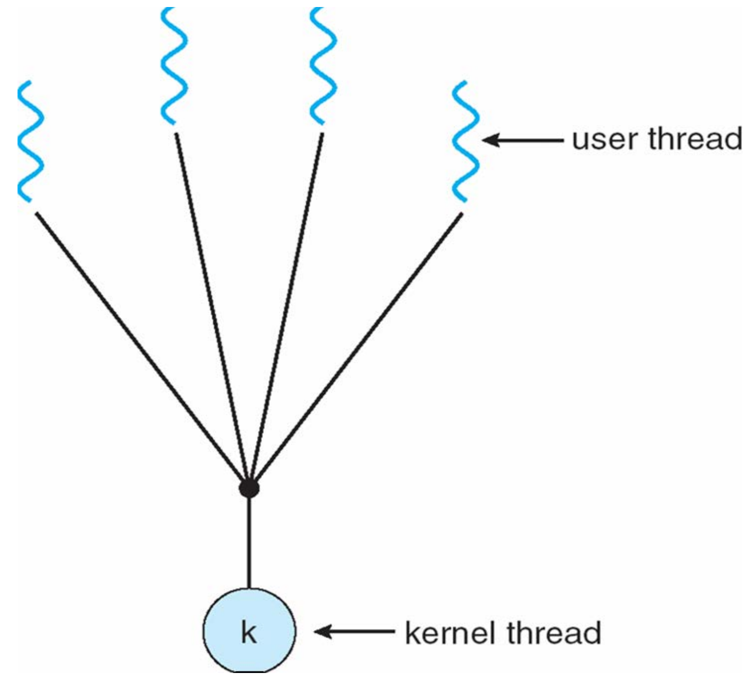
Pthread Example 2

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

int arrayA[10], arrayB[10];

void *routine1(void  *param)
{
    int var1, var2
    …
}
void *routine2(void  *param)
{
    int var1, var2, var3
    …
}


int main (int argc, char *argv[])
{
    /* create the thread */
    pthread_create(&tid[0], &attr, routine1, NULL);
    pthread_create(&tid[1], &attr, routine2, NULL);
    pthread_join(tid[0]); pthread_join(tid[1]);
}
```



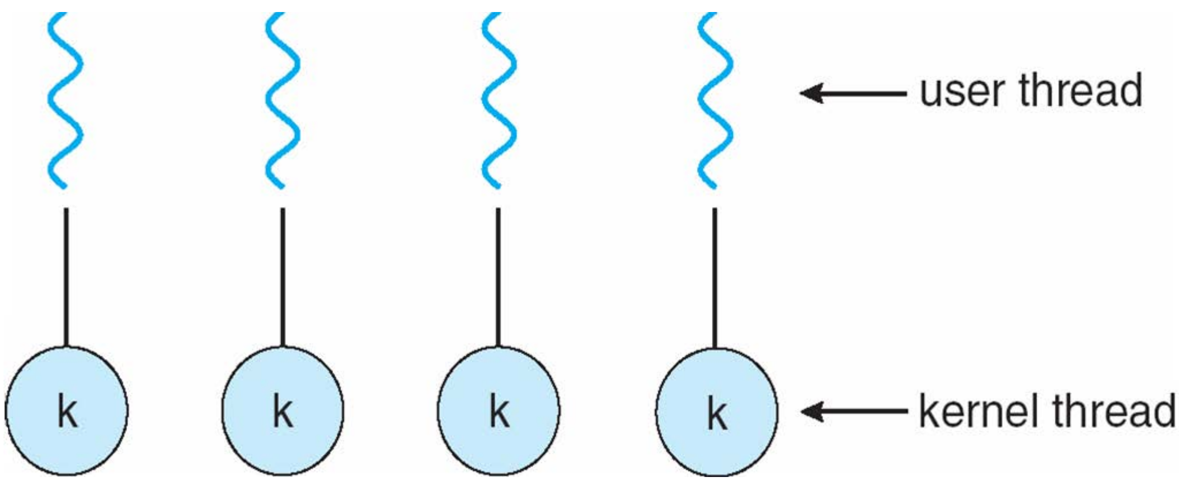
User-level Threads

- Kernel is unaware of threads
  - Early UNIX and Linux did not support threads
- Threading runtime
  - Handle context switching
    - Setjmp/longjmp, …
- Advantage
  - No kernel support
  - Fast (no kernel crossing)
- Disadvantage
  - Blocking system call. What happens?



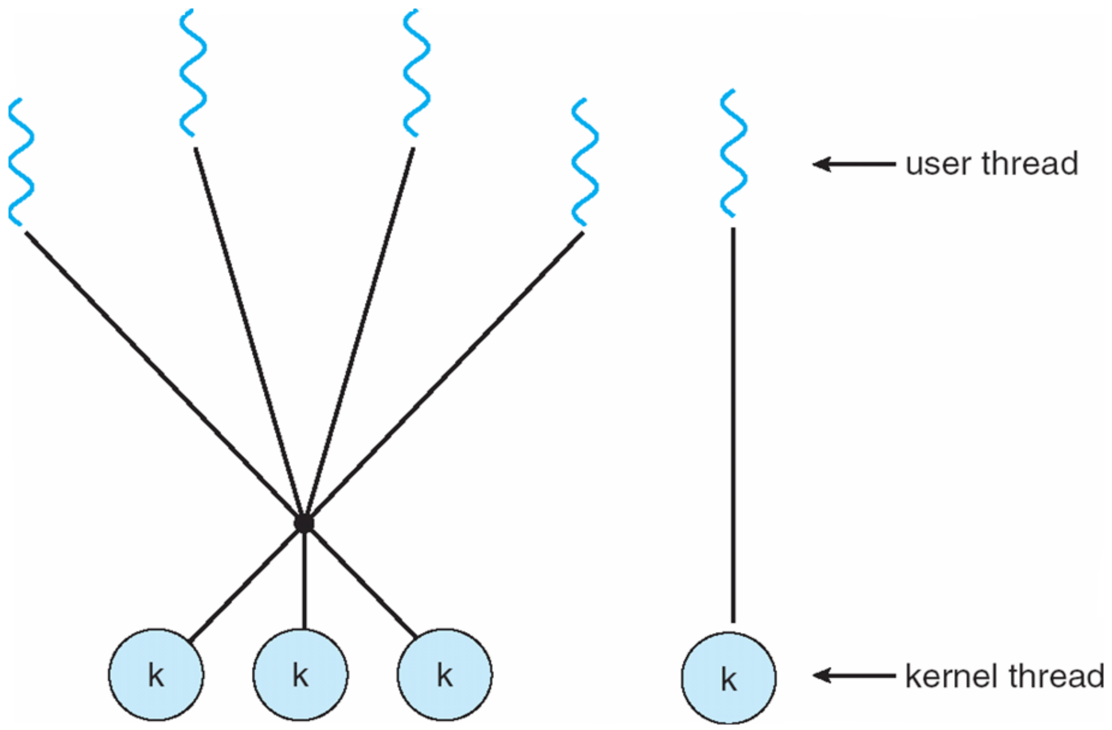
Kernel-level Threads

- Native kernel support for threads
  - Most modern OS (Linux, Windows NT)
- Advantage
  - No threading runtime
  - Native system call handing
- Disadvantage
  - Overhead



Hybrid Threads

- Many kernel threads to many user threads
  - Best of both worlds?



Threads: Advanced Topics

- Semantics of Fork/exec()
- Signal handling
- Thread pool
- Multicore

Semantics of fork()/exec()

- Remember fork(), exec() system calls?
  - Fork: create a child process (a copy of the parent)
  - Exec: replace the address space with a new pgm.

- Duplicate *all* threads or *the caller* only?
  - Linux: the calling thread only
  - Complicated. Don't do it!
    - Why? Mutex states, library, …
    - Exec() immediately after Fork() may be okay.

Signal Handling

- What is *Singal*?
  - $ man 7 signal
  - OS to process notification
    - "hey, wake-up, you've got a packet on your socket,"
    - "hey, wake-up, your timer is just expired."
- Which *thread* to deliver a signal?
  - Any thread
    - e.g., kill(pid)
  - Specific thread
    - E.g., pthread_kill(tid)

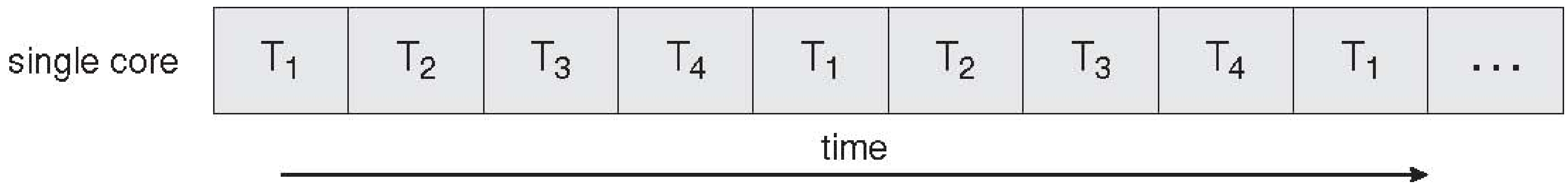
KU THE UNIVERSITY OF KANSAS

Thread Pool

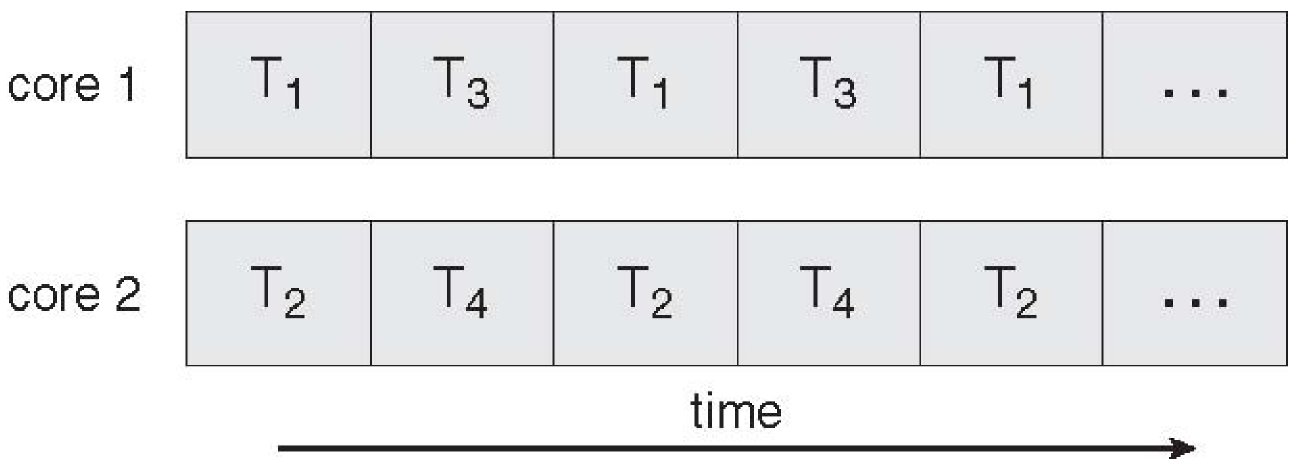
- Managing threads yourself can be cumbersome and costly
  - Repeat: create/destroy threads as needed.

- Let's create a set of threads ahead of time, and just ask them to execute my functions
  - #of thread ~ #of cores
  - No need to create/destroy many times
  - Many high-level parallel libraries use this.
    - e.g., Intel TBB (threading building block), …

Single Core Vs. Multicore Execution



Single core execution



Multiple core execution

Challenges for Multithreaded Programming in Multicore

- How to divide activities?
- How to divide data?
- **How to synchronize accesses to the shared data? → next class**
- **How to test and dubug?**

EECS750

Summary

- Thread
  - What is it?
    - Independent flow of control.
  - What for?
    - Lightweight programming construct for concurrent activities
  - How to implement?
    - Kernel thread vs. user thread
- Next class
  - How to synchronize?