CS 3502
Operating Systems

Thread

Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

Outline

- What is thread?
  - Multiple thread application
  - Thread vs Process
  - Advantage and disadvantage of thread
- Thread in Linux
- Thread design
  - Kernel space vs User space
  - Local thread vs Global thread scheduling

Process review

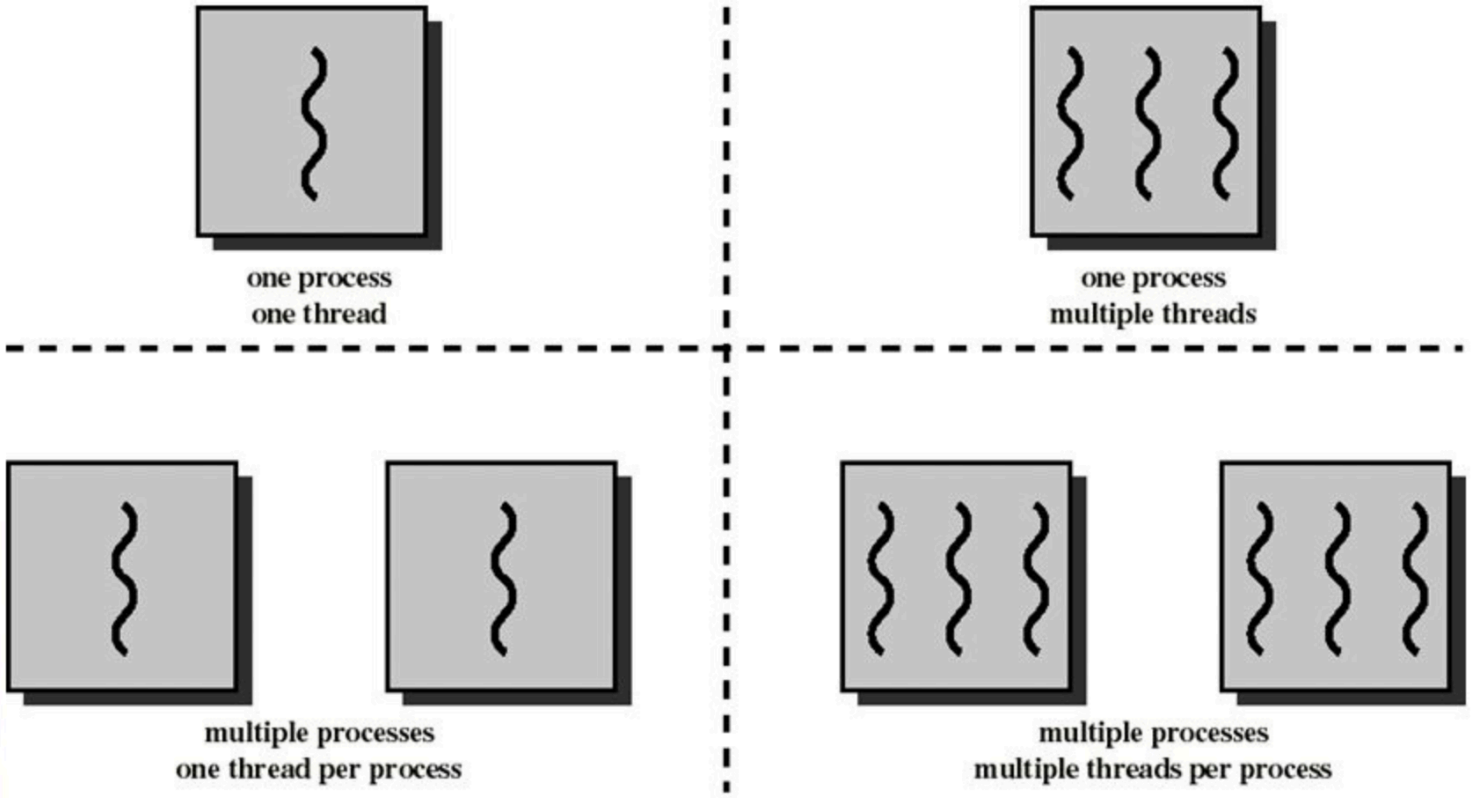
- Definition
  - An instance of a *program* running on a computer
  - An *abstraction* that supports running programs - -> cpu virtualization
  - An *execution stream* in the context of a particular *process state* - -> dynamic unit
  - A *sequential* stream of execution in its *own address space* - -> execution code line by line

What is a thread?

- Thread
  - A finer-granularity entity for execution and parallelism
  - Lightweight process
  - A program in execution without dedicated address space

- Multithreading
  - Running multiple threads within a single process

Finer-granularity entity



Process : thread = 1: N

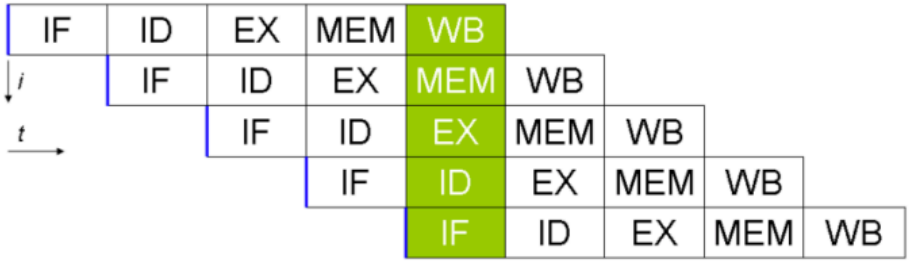
What is a thread?

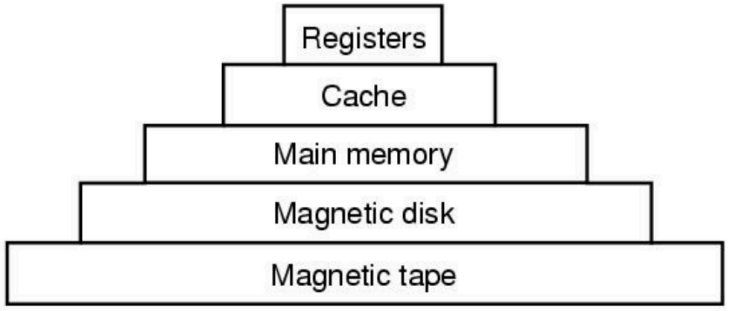
- Thread
  - A finer-granularity entity for execution and parallelism
  - Lightweight process
  - A program in execution without dedicated address space

- Multithreading
  - Running multiple threads within a single process

Process review

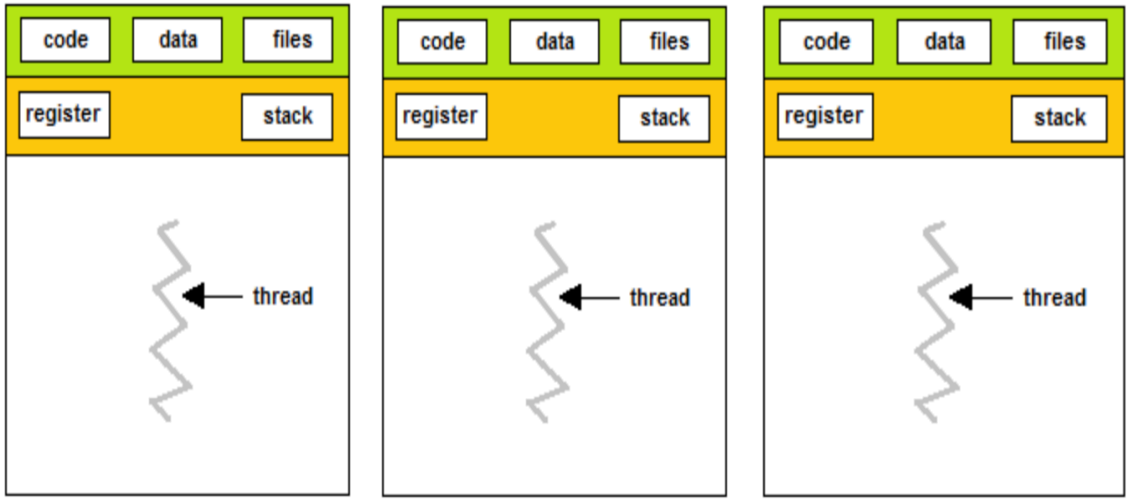
- Two parts of a process
  - Sequential execution of instructions
  - Process state
    - registers: PC (program counter), SP (stack pointer),…
    - Memory: address space, code, data, stack, heap …
    - I/O status: open files …



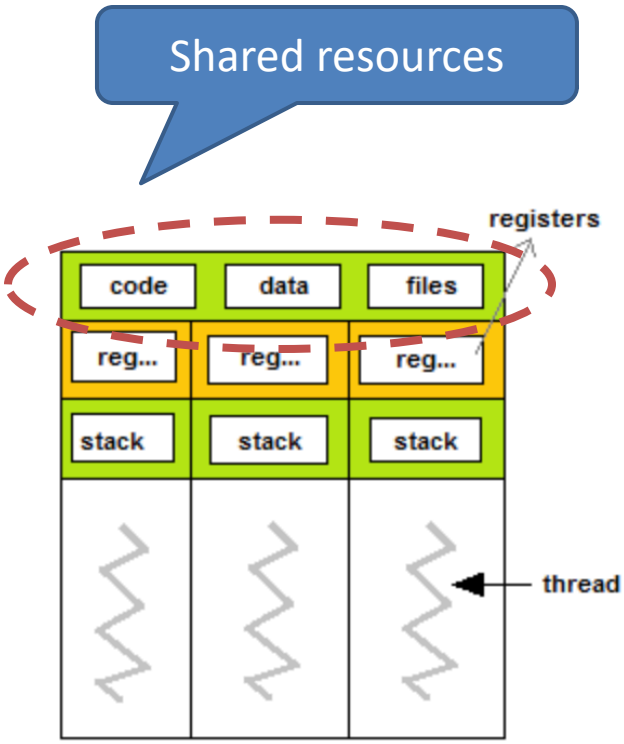


Lightweight process

Occupy more memory,
complex switching (e.g., save old data, switch to new data),
low CPU utilization (e.g., slow context switch)



Three processes



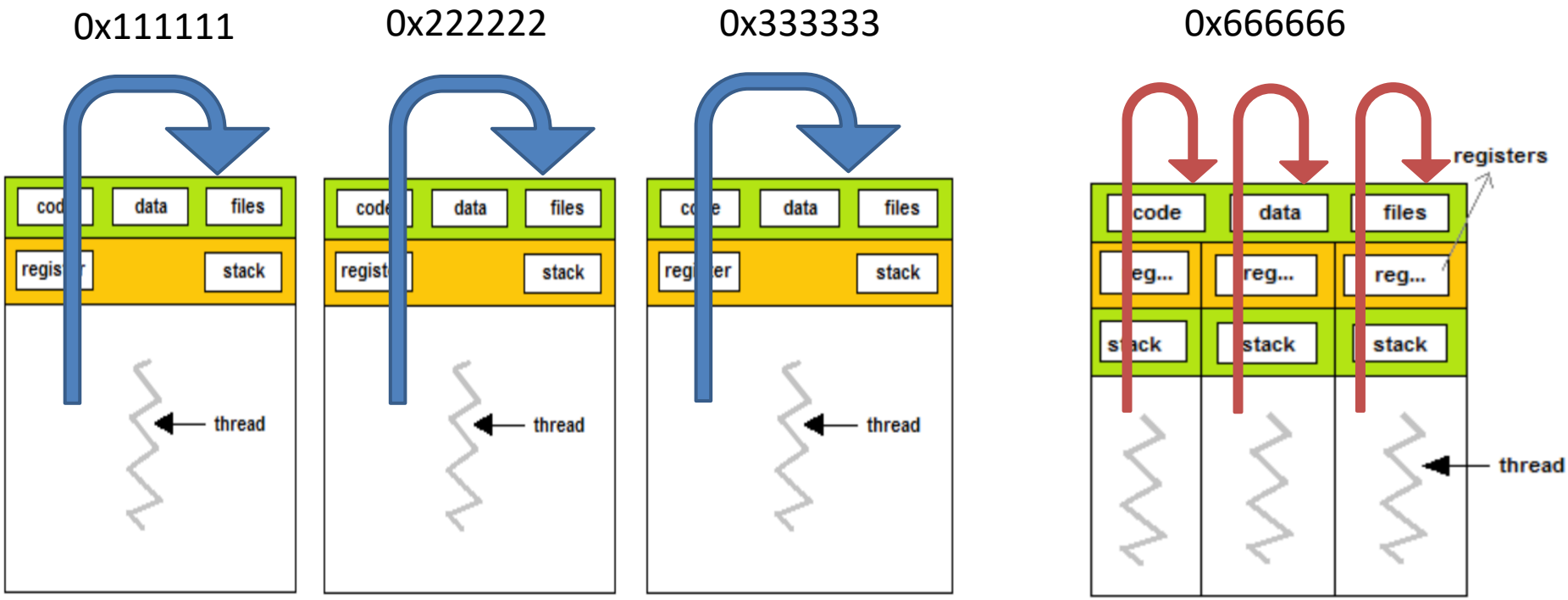
Three threads

What is a thread?

- Thread
  - A finer-granularity entity for execution and parallelism
  - Lightweight process
  - A program in execution without dedicated address space

- Multithreading
  - Running multiple threads within a single process

Dedicated address space

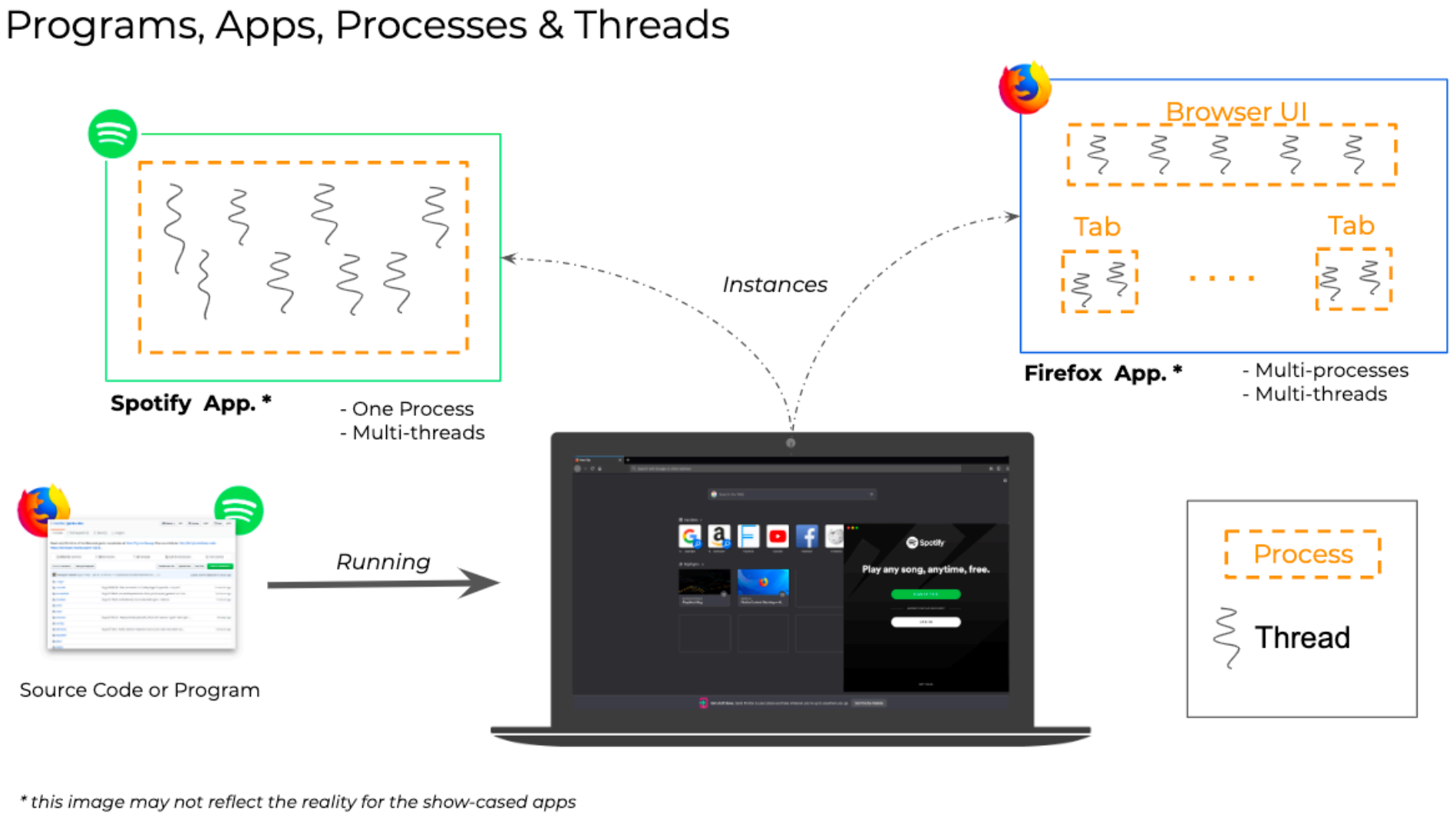


What is a thread?

- Thread
  - A finer-granularity entity for execution and parallelism
  - Lightweight process
  - A program in execution without dedicated address space

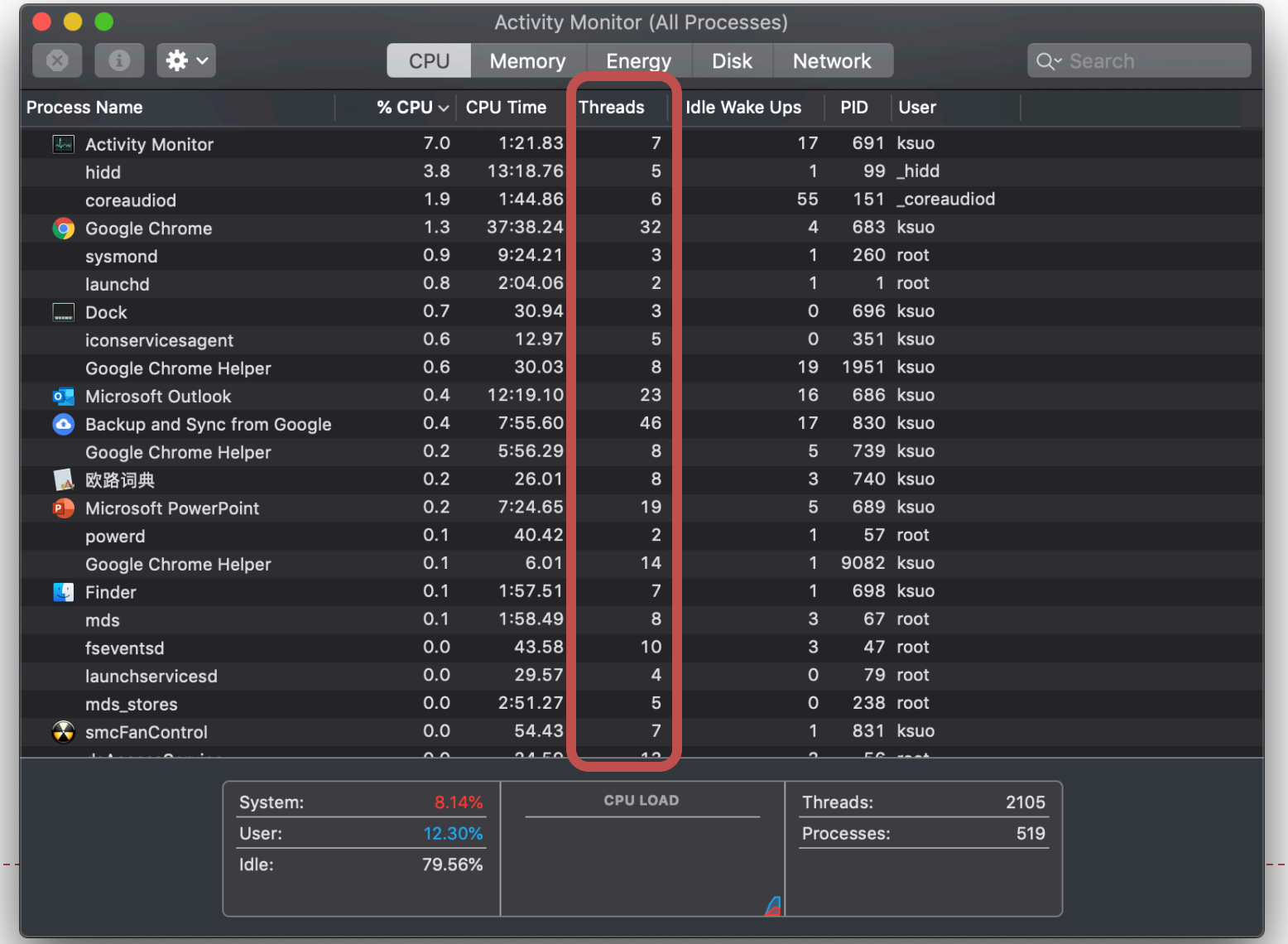
- Multithreading
  - Running multiple threads within a single process

What is a thread? Multithreading apps



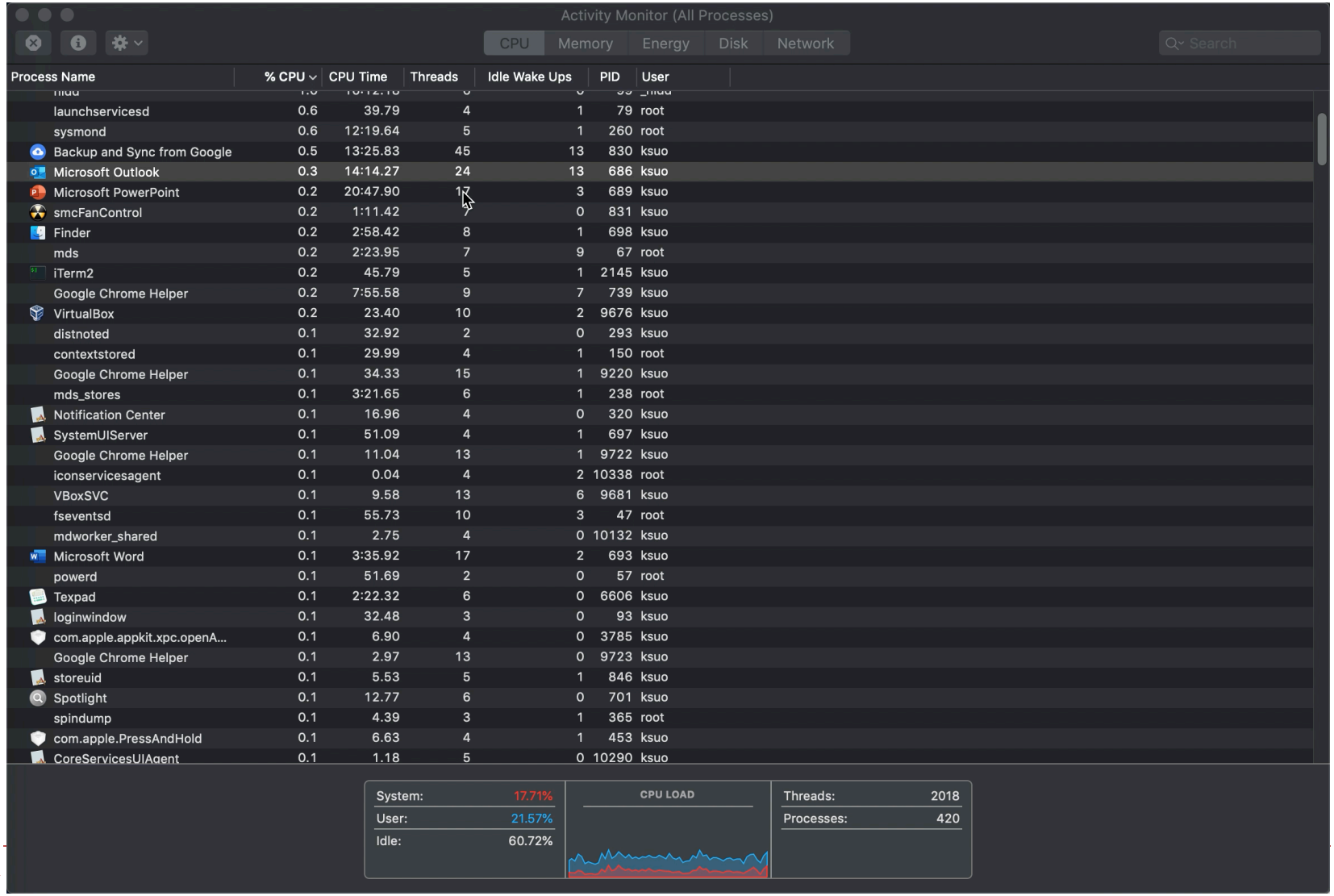
** this image may not reflect the reality for the show-cased apps*

What is a thread? Multithreading apps

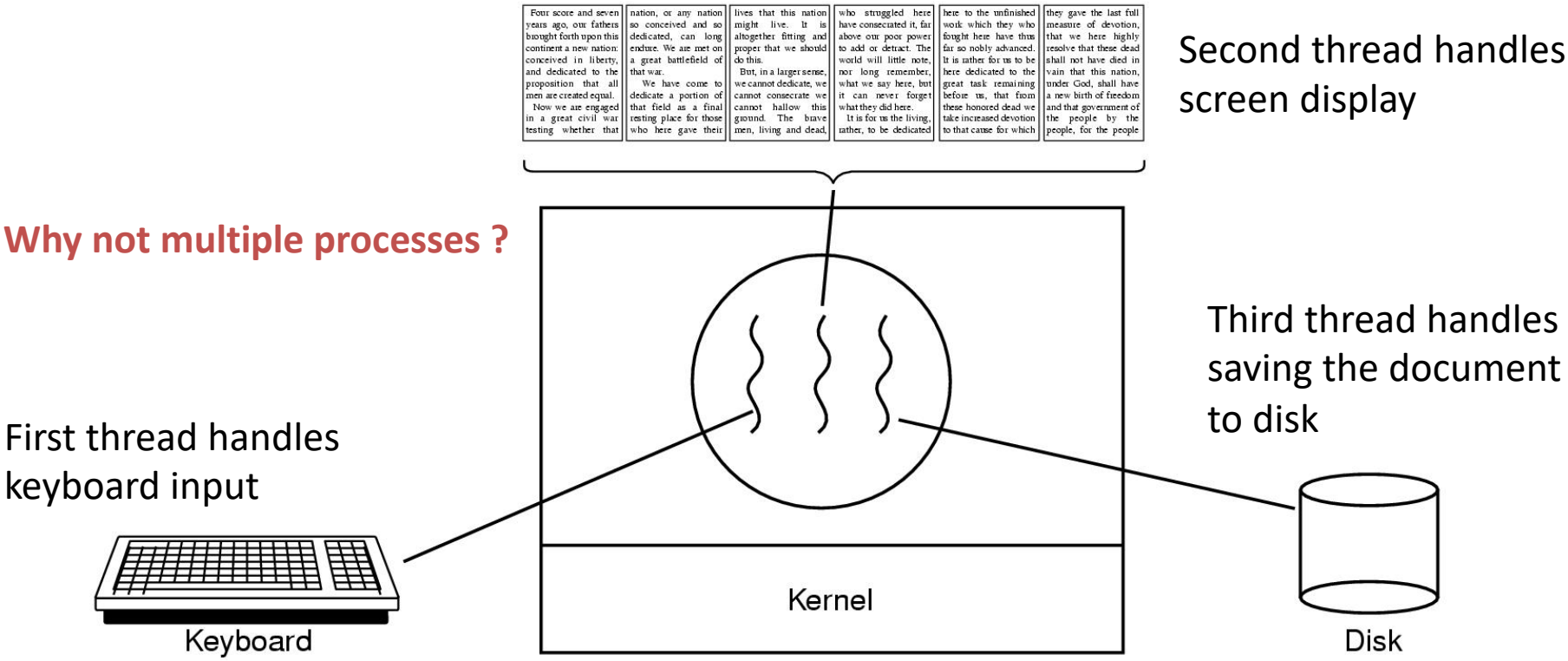


Multithreading apps

https://youtu.be/Me0mL44TxW0



Example 1: A word process with three threads

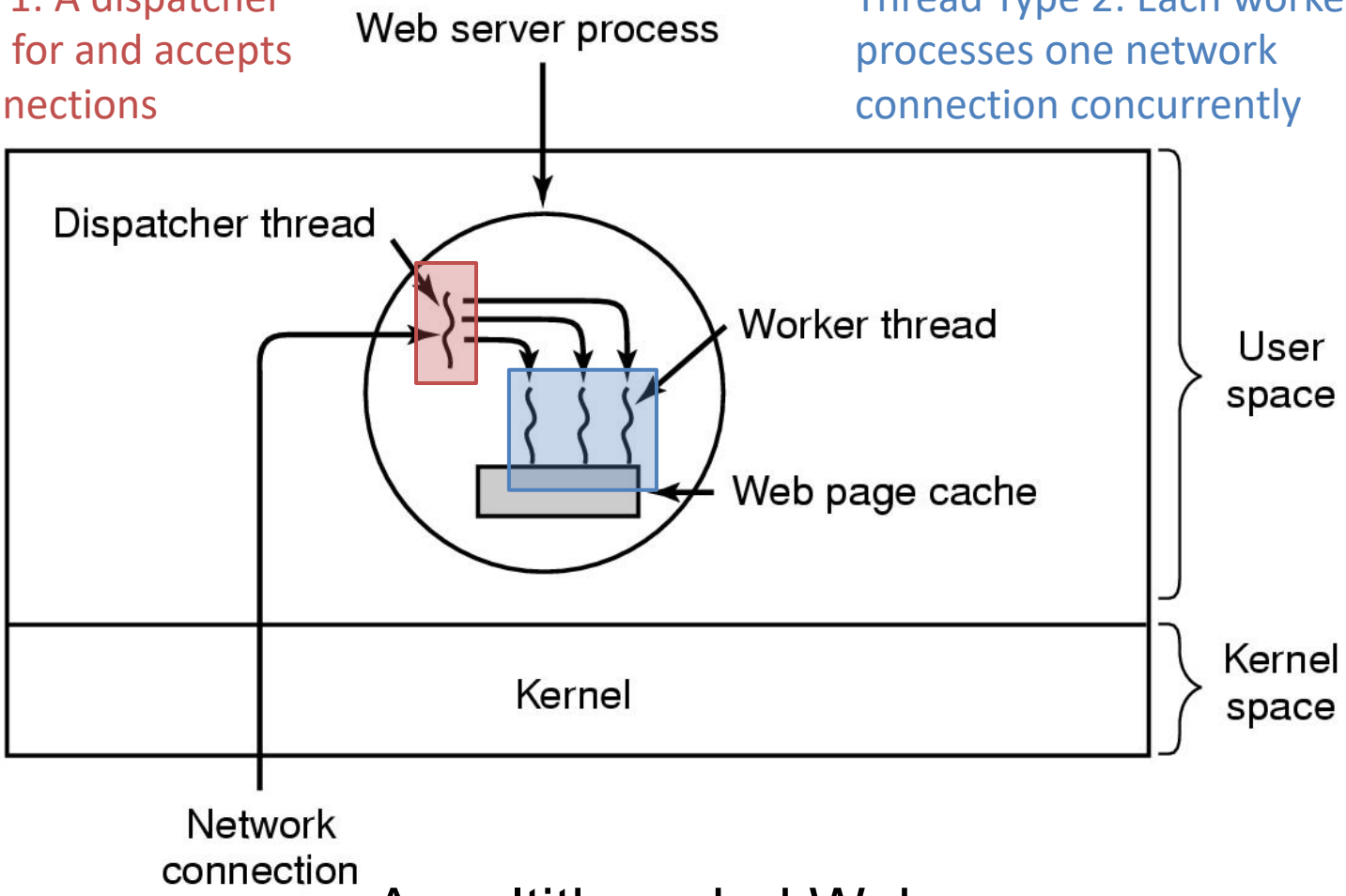


A word process with three threads.

Example 2: a multi-threaded web server

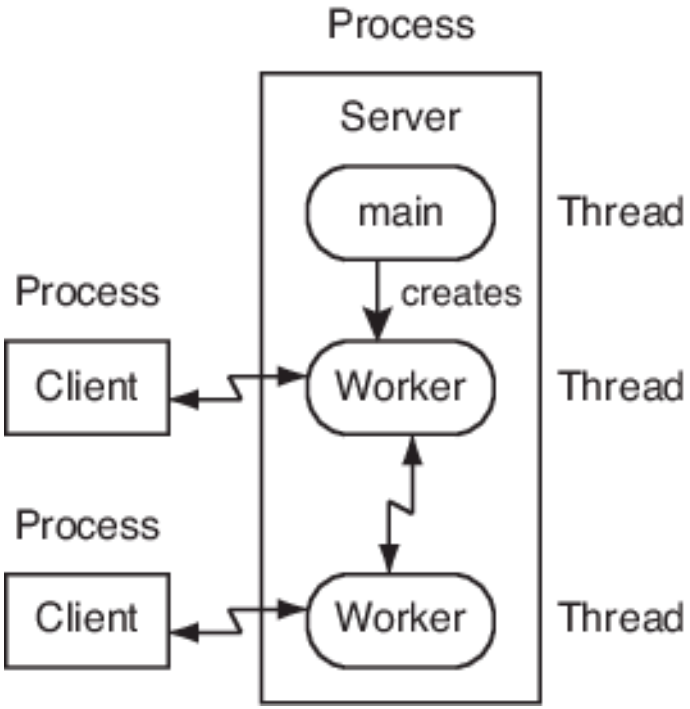
Thread Type 1: A dispatcher thread waits for and accepts network connections

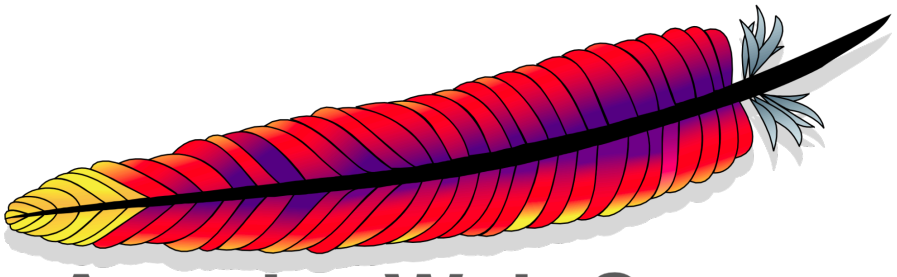
Thread Type 2: Each worker processes one network connection concurrently



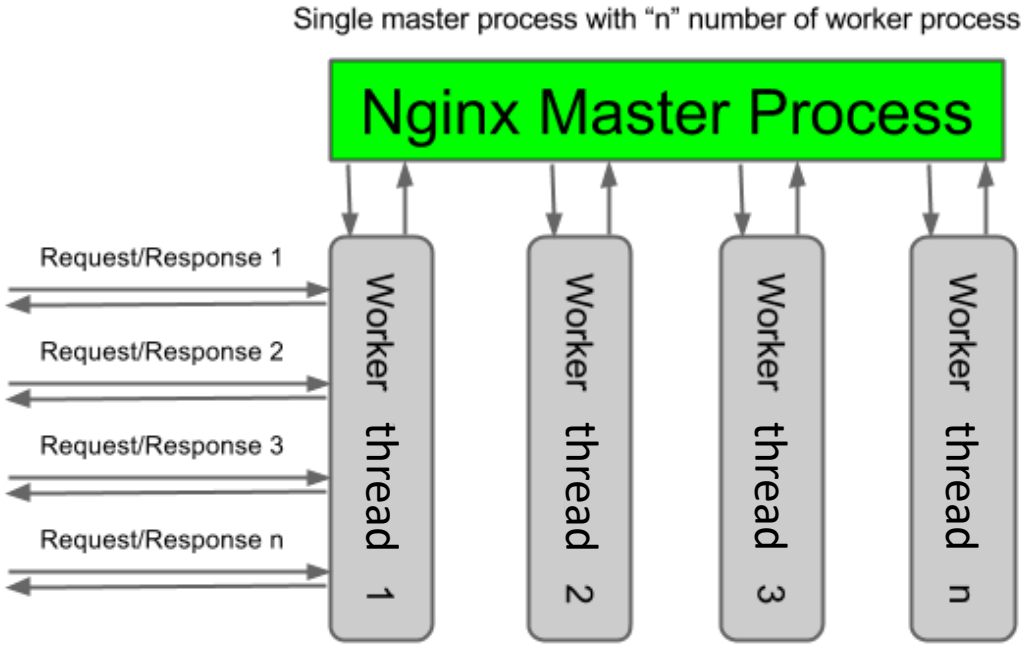
A multithreaded Web server.

Example 2: a multi-threaded web server



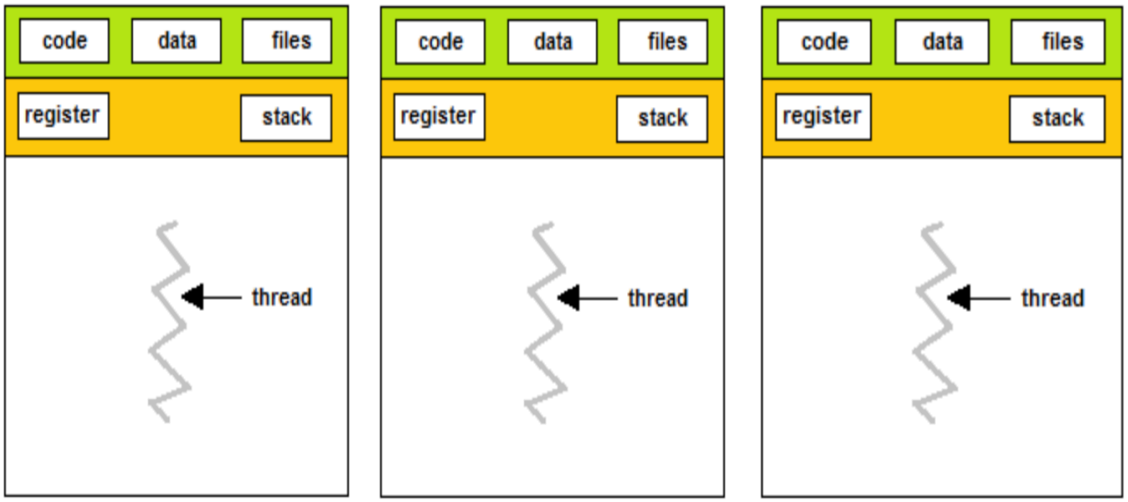


Example 2: a multi-threaded web server

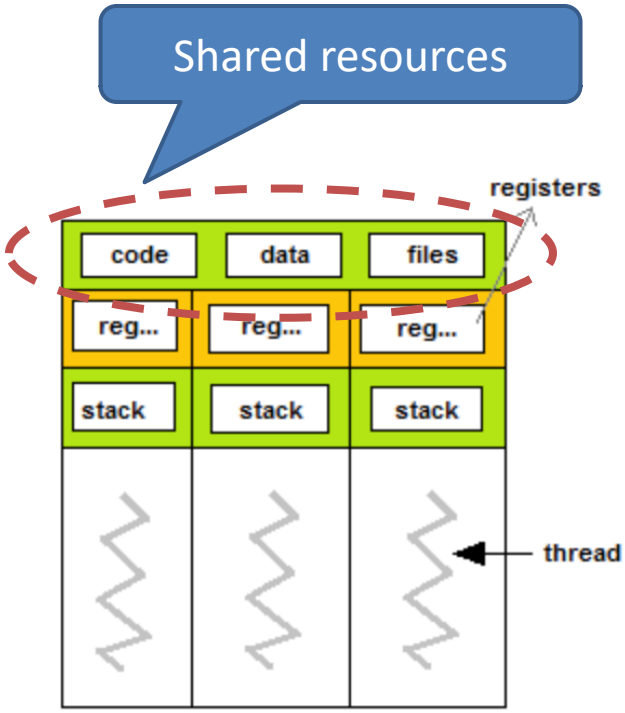


Why not multiple processes ?

Occupy more memory,
complex switching
low CPU utilization
Difficult to communicate
Expensive data sharing



Three processes



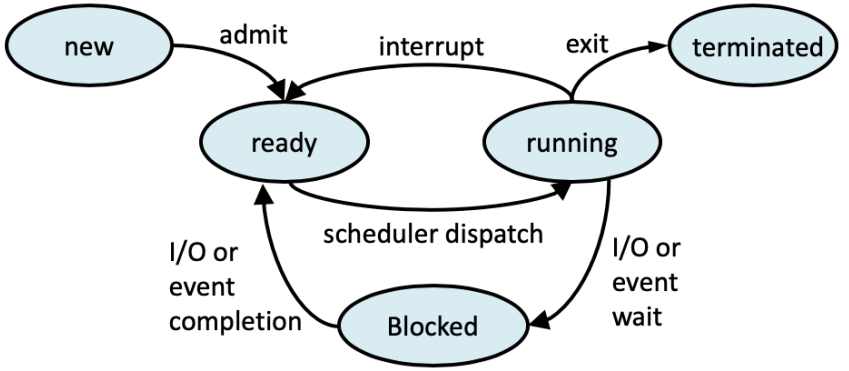
Three threads

Why multiple threads

- Good example from Wikipedia: multiple threads within a single process are like multiple cooks trying to prepare the same meal together.

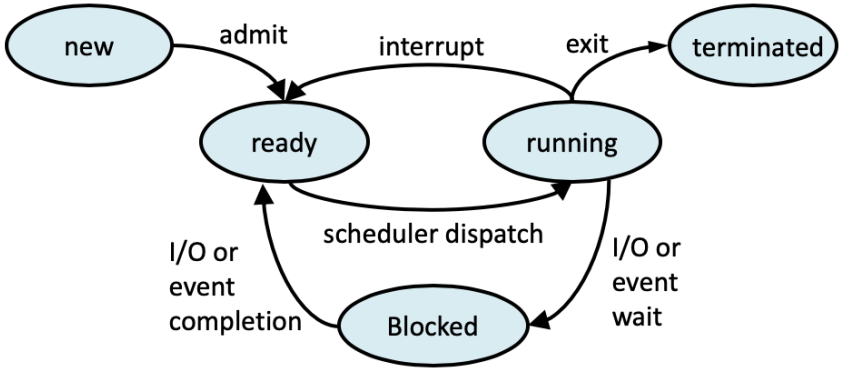
  - Each one is doing one thing.
  - They are probably doing different things.
  - They all share the same recipe but may be looking at different parts of it.
  - They have private state but can communicate easily.
  - They must coordinate!

Thread states



- Running:
  - executing instructions on a CPU core.
- Ready:
  - not executing instructions but capable of being restarted.
- Waiting, Blocked or Sleeping:
  - not executing instructions and not able to be restarted until some event occurs.
- New/Terminated

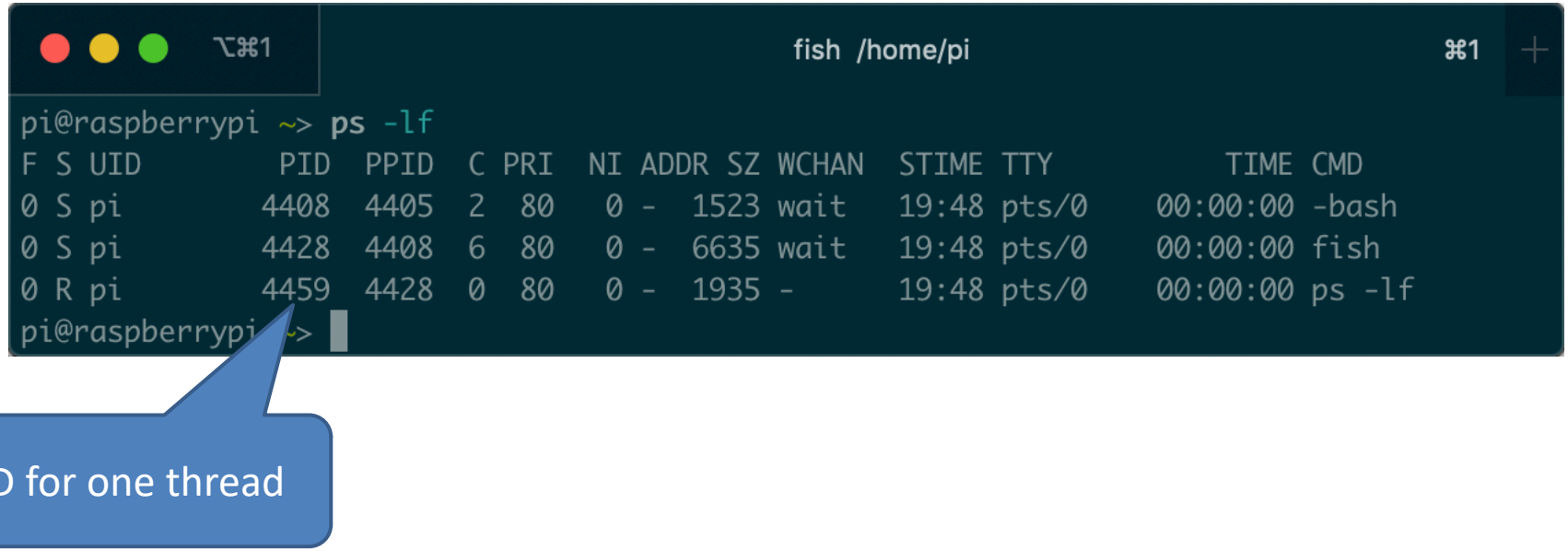
Thread state transitions



- Running -> Ready:
  - a thread was descheduled.
- Running -> Waiting:
  - a thread was blocked for I/O or waited for some events.
- Waiting -> Ready:
  - the event the thread was waiting for happened.
- Ready -> Running:
  - a thread was scheduled.
- Running -> Terminated:
  - a thread exited or hit a fatal exception.

Thread ID

- Identifiers
  - pid: ID of the thread (Linux treats process and thread equally in management)



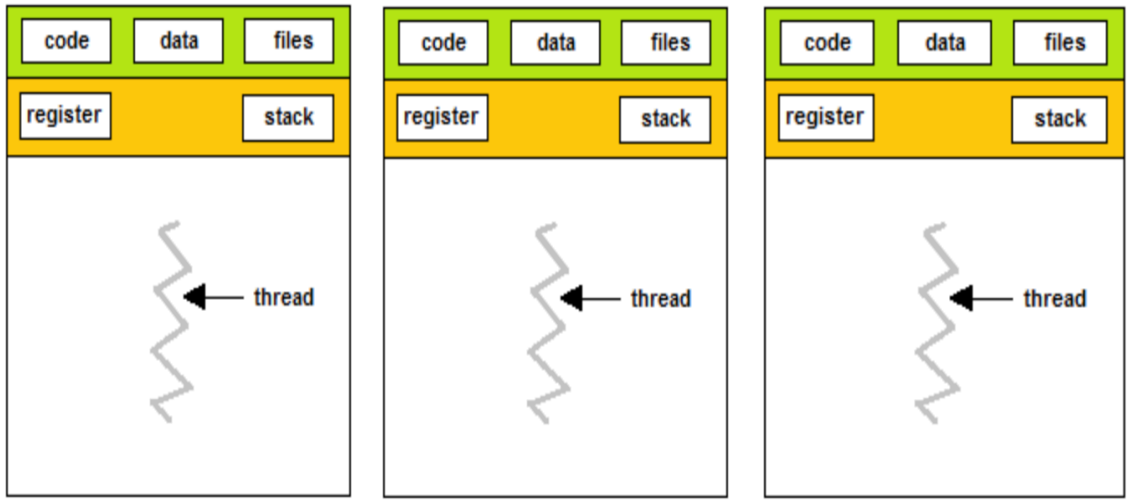
Outline

- What is thread?
  - Multiple thread application
  - Thread vs Process
  - Advantage and disadvantage of thread
- Thread in Linux
- Thread design
  - Kernel space vs User space
  - Local thread vs Global thread scheduling

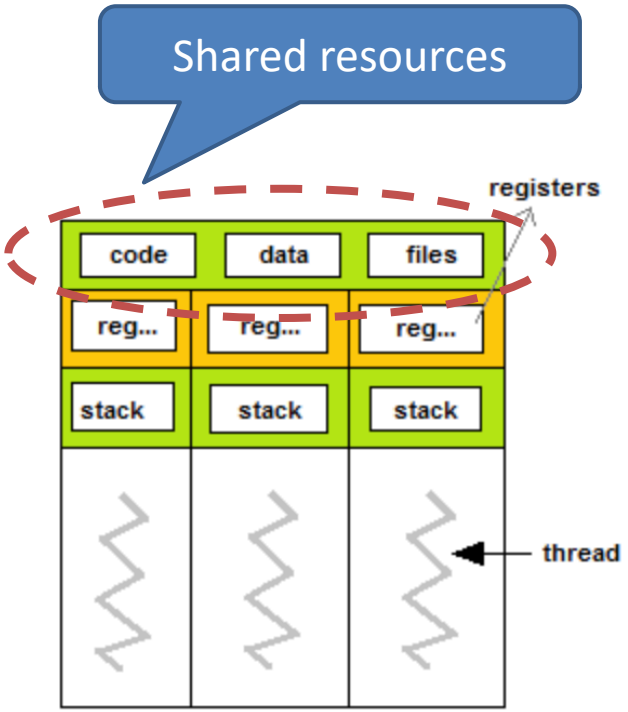
Process vs. Thread comparison

	Process	Thread
Concurrency and protection	?	?
Data structure	?	?
Performance	?	?

Processes vs. Threads (Concurrency and protection)



Three processes

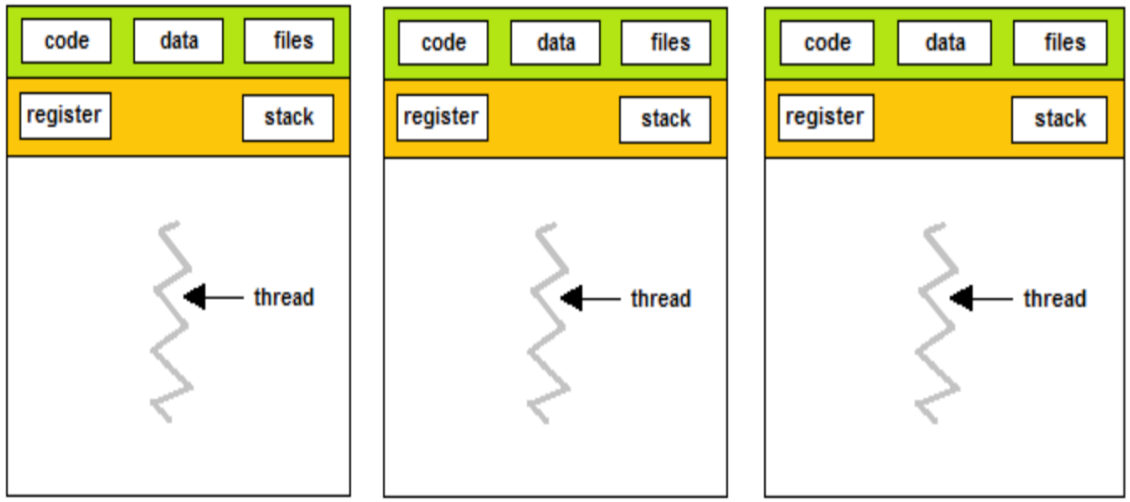


Three threads

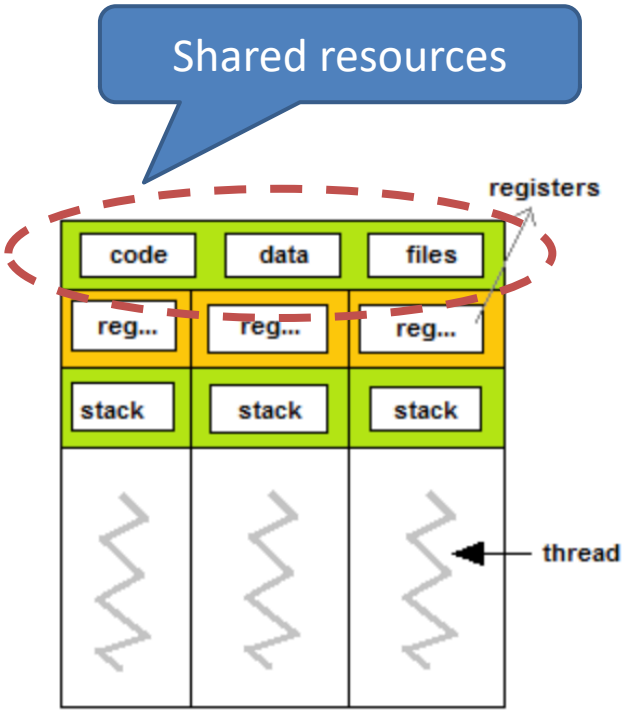
Processes vs. Threads (Concurrency and protection)

	Process	Threads
Concurrency	Parallel execution stream of instructions 	Maintain parallel execution stream of instructions 
Protection		

Processes vs. Threads (Concurrency and protection)



Three processes



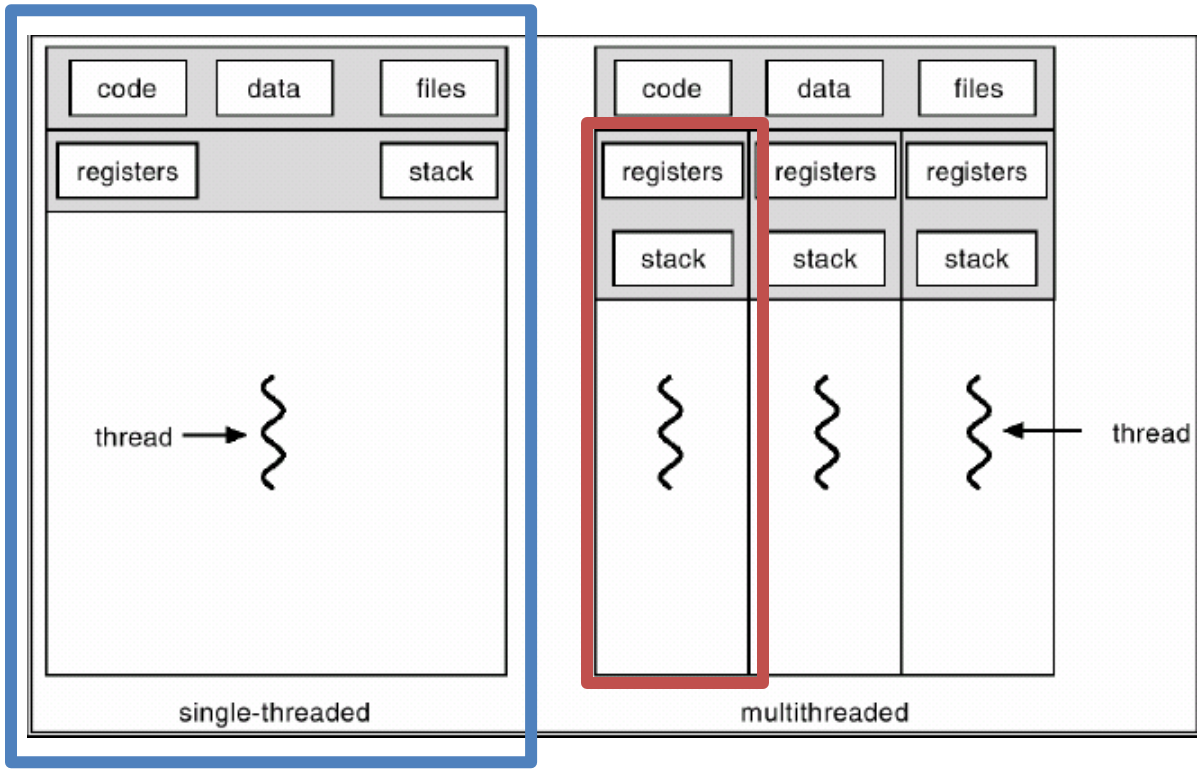
Three threads

Processes vs. Threads (Concurrency and protection)

Separate concurrency from protection

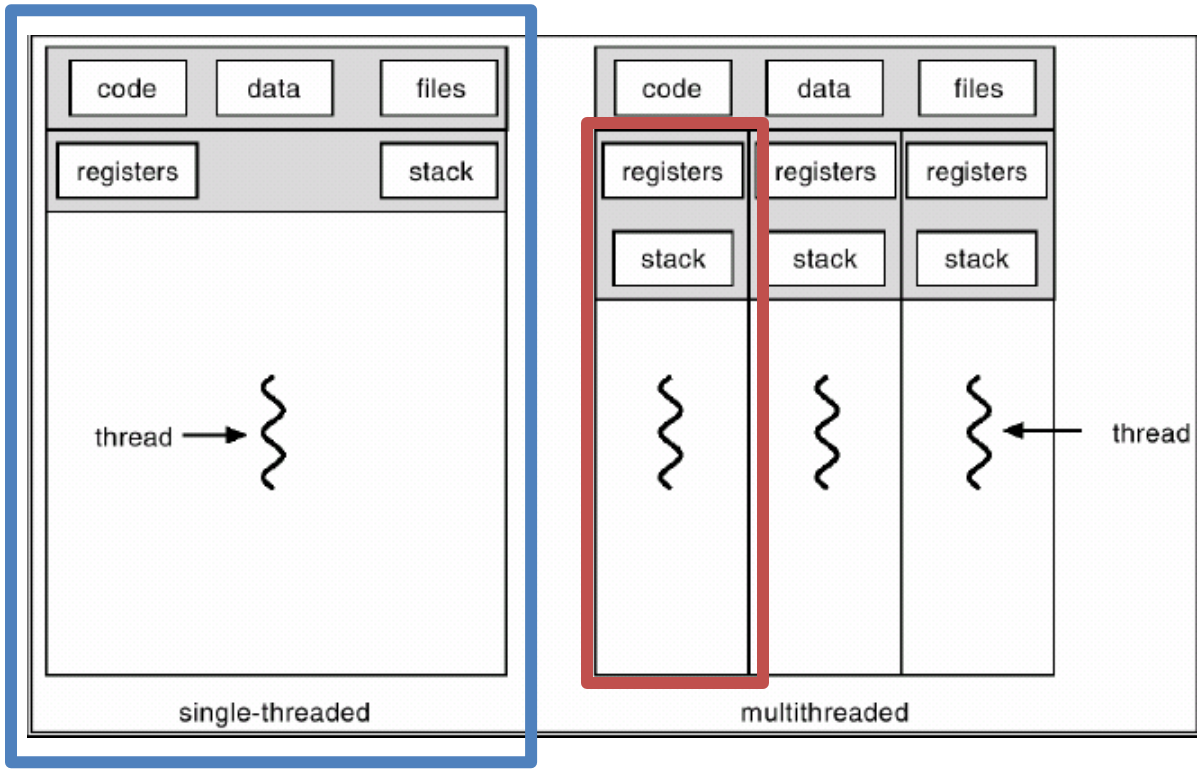
	Process	Threads
Concurrency	Parallel execution stream of instructions 	Maintain parallel execution stream of instructions 
Protection	A dedicated address space 	Share address space with other threads 

Processes vs. Threads (Data structure)



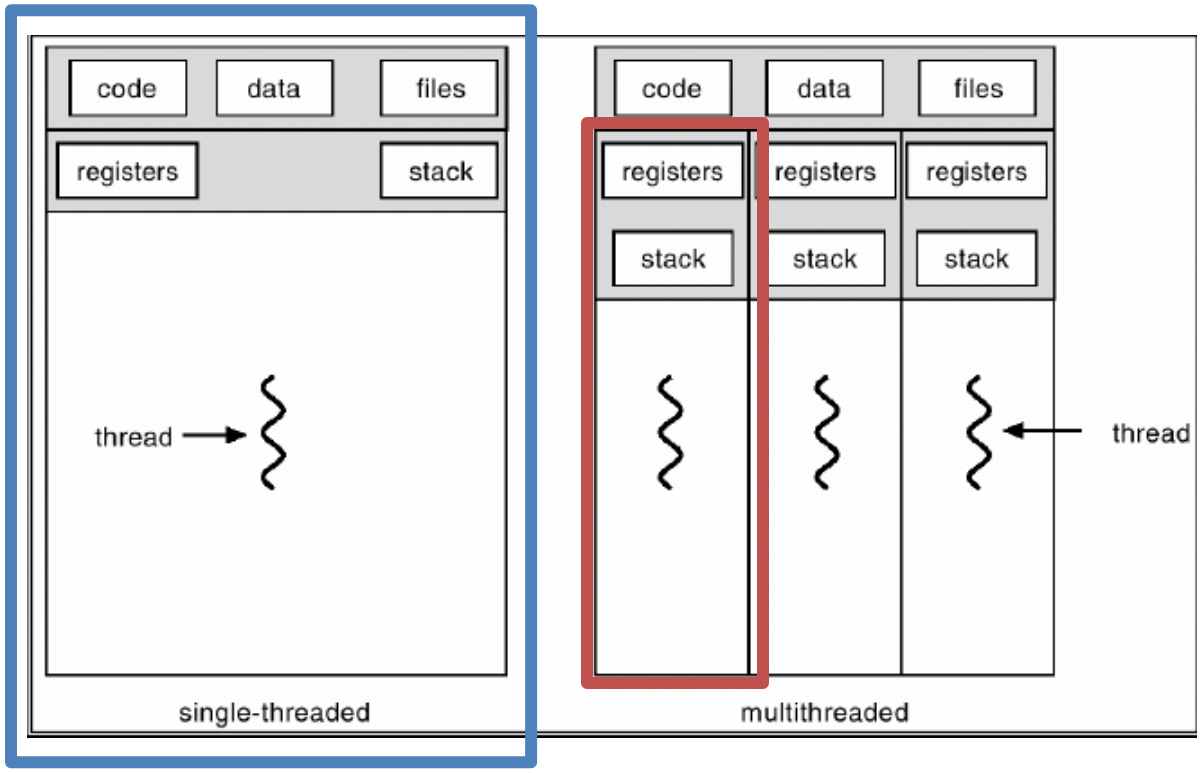
Process	Thread
Have data/code/heap	Share code, data, heap
Include at lease one thread	Multiple can coexist in a process
Have own address space, isolated from other processes	Only have own stack and registers

Processes vs. Threads (Data structure)



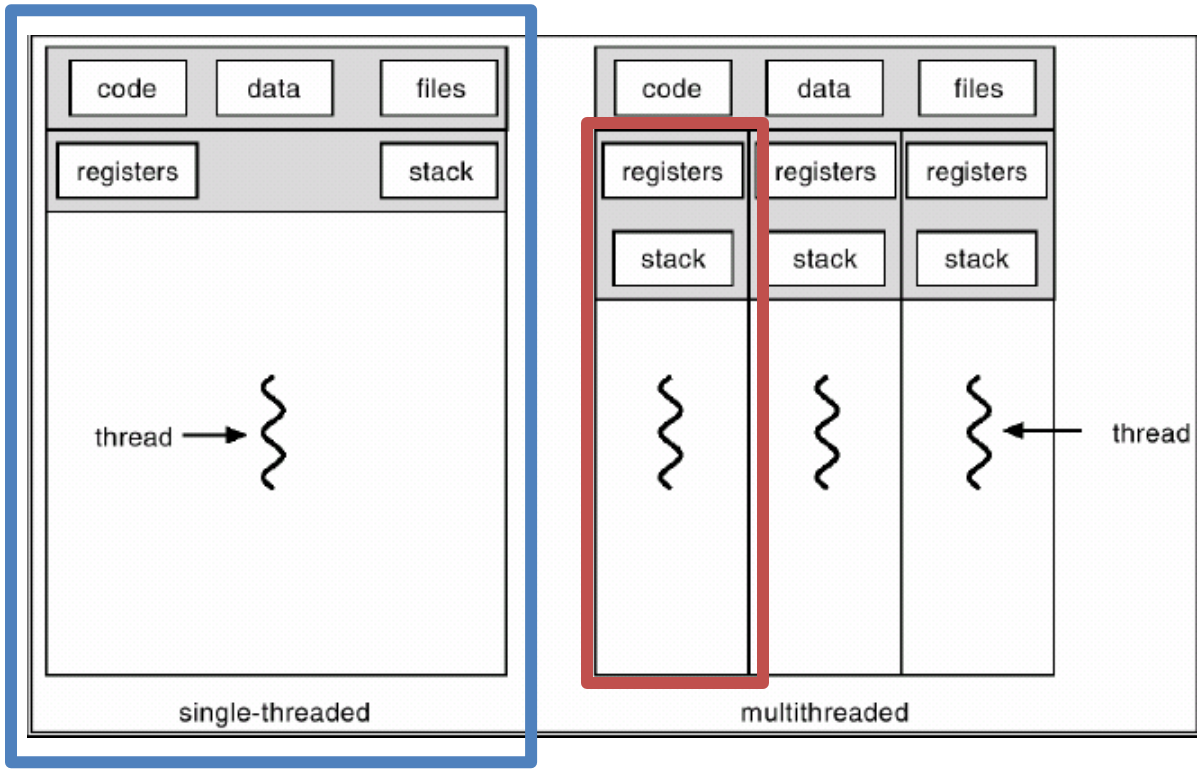
Context	Process	Thread
File pointer	*	
Stack	*	*
Memory	*	
State	*	*
Priority	*	*
I/O state	*	
Authority	*	

Processes vs. Threads (Data structure)



Context	Process	Thread
Scheduling	*	
Statistics	*	
File description	*	
Read/Write pointer	*	
Event/ Signal	*	
Registers	*	*

Processes vs. Threads (Performance)

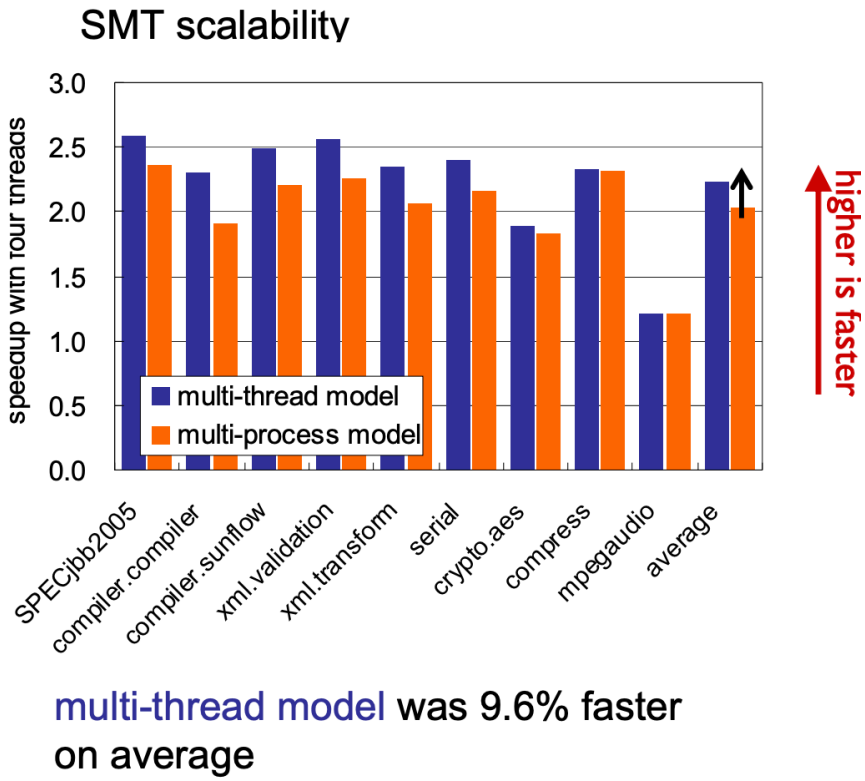


Process	Thread
Expensive to create	Inexpensive to create
Expensive context switching	Inexpensive context switching
IPC can be expensive	Efficient communication

Paper: Performance of Multi-Process and Multi-Thread Processing on Multi-core SMT Processors

- Our results showed that both models (multi-process vs. multi-thread) achieved almost comparable performance, whereas the multi-thread model achieved much better SMT scalability and higher performance.

https://pdfs.semanticscholar.org/dd54/a215131c5eacd997708c5c4612345fe989c7.pdf



multi-thread model was 9.6% faster on average

Outline

- What is thread?
  - Multiple thread application
  - Thread vs Process
  - Advantage and disadvantage of thread
- Thread in Linux
- Thread design
  - Kernel space vs User space
  - Local thread vs Global thread scheduling

Advantages of Threads

- Efficient creation
  - Only create the thread context
- Express concurrency
  - Lightweight, better performance, higher scalability
- Efficient communication
  - Communication can be carried out via shared data objects within the shared address space

Disadvantages of Threads

- Shared data - -> Security
  - Global variables are shared between threads.
  - Accidental data changes can cause errors.
- Lack of robustness
  - Crash in one thread will crash the entire process.
- Some library functions may not be thread-safe
  - Library Functions that return pointers to static internal memory. E.g. gethostbyname()

Outline

- What is thread?
  - Multiple thread application
  - Thread vs Process
  - Advantage and disadvantage of thread
- Thread in Linux
- Thread design
  - Kernel space vs User space
  - Local thread vs Global thread scheduling

Processes vs. Threads in Linux

- On Mon, 5 Aug 1996, Peter P. Eiserloh wrote:
  We need to keep a clear the concept of threads. Too many people seem to confuse a thread with a process. The following discussion does not reflect the current state of linux, but rather is an attempt to stay at a high level discussion.

- http://lkml.iu.edu/hypermail/linux/kernel/9608/0191.html
- The way Linux thinks about this (and the way I want things to work) is that there _is_ no such thing as a “process” or a “thread”. There is only the totality of the COE (called "task" by Linux).

Threads in old Linux

- Thread control block (TCB)
  - The `thread_struct` structure
  - Includes registers and processor-specific context

https://elixir.bootlin.com/linux/v2.6.32.71/source/arch/x86/include/asm/processor.h#L425

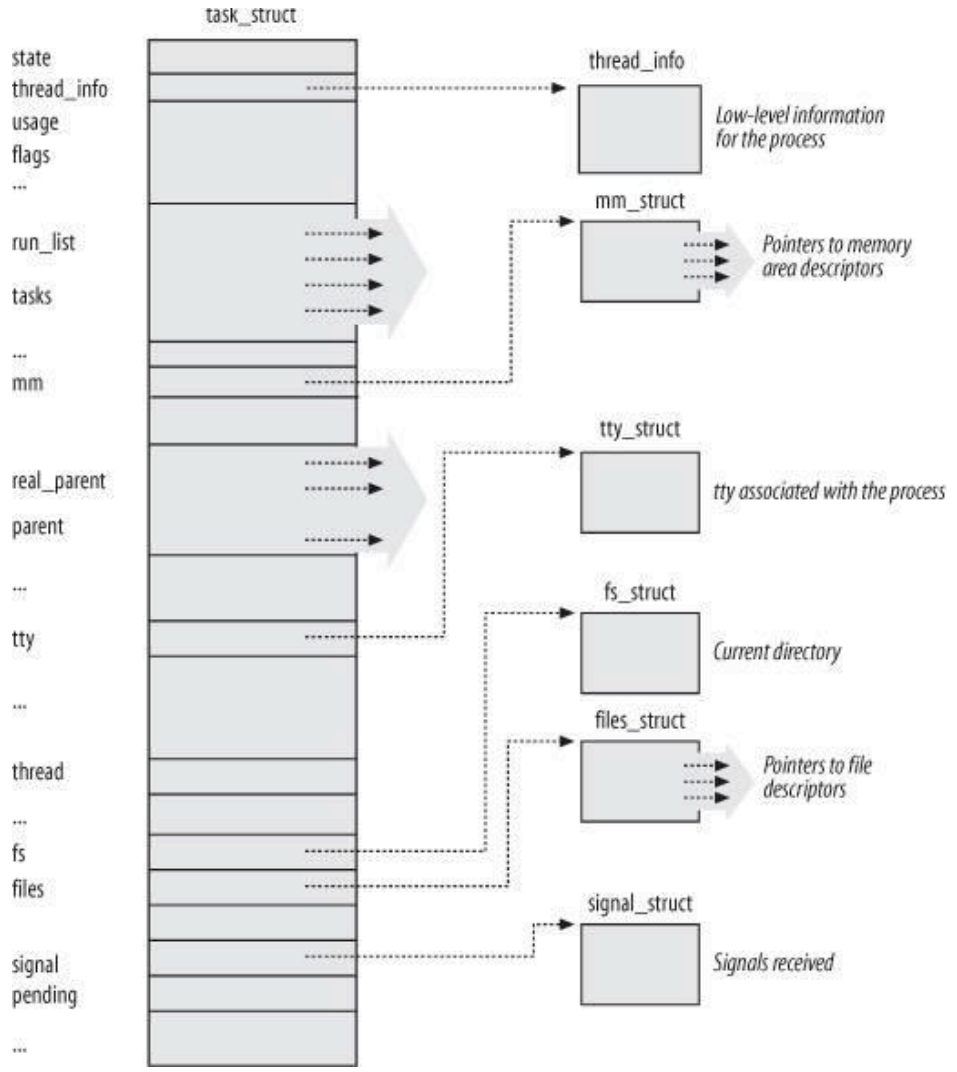
```
struct thread_struct {
        /* Cached TLS descriptors: */
        struct desc_struct      tls_array[GDT_ENTRY_TLS_ENTRIES];
        unsigned long           sp0;
        unsigned long           sp;
#ifdef CONFIG_X86_32
        unsigned long           sysenter_cs;
#else
        unsigned short          es;
        unsigned short          ds;
        unsigned short          fsindex;
        unsigned short          gsindex;
#endif
#ifdef CONFIG_X86_32
        unsigned long           ip;
#endif
#ifdef CONFIG_X86_64
        unsigned long           fs;
#endif
        unsigned long           gs;
```

registers

Threads in latest Linux

https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h#L584

- Linux treats threads like processes
- Process control block (PCB)
  - State
  - Identifiers
  - Scheduling info (shared)
  - File system (shared)
  - Virtual memory (shared)
  - Process specific context
  - …



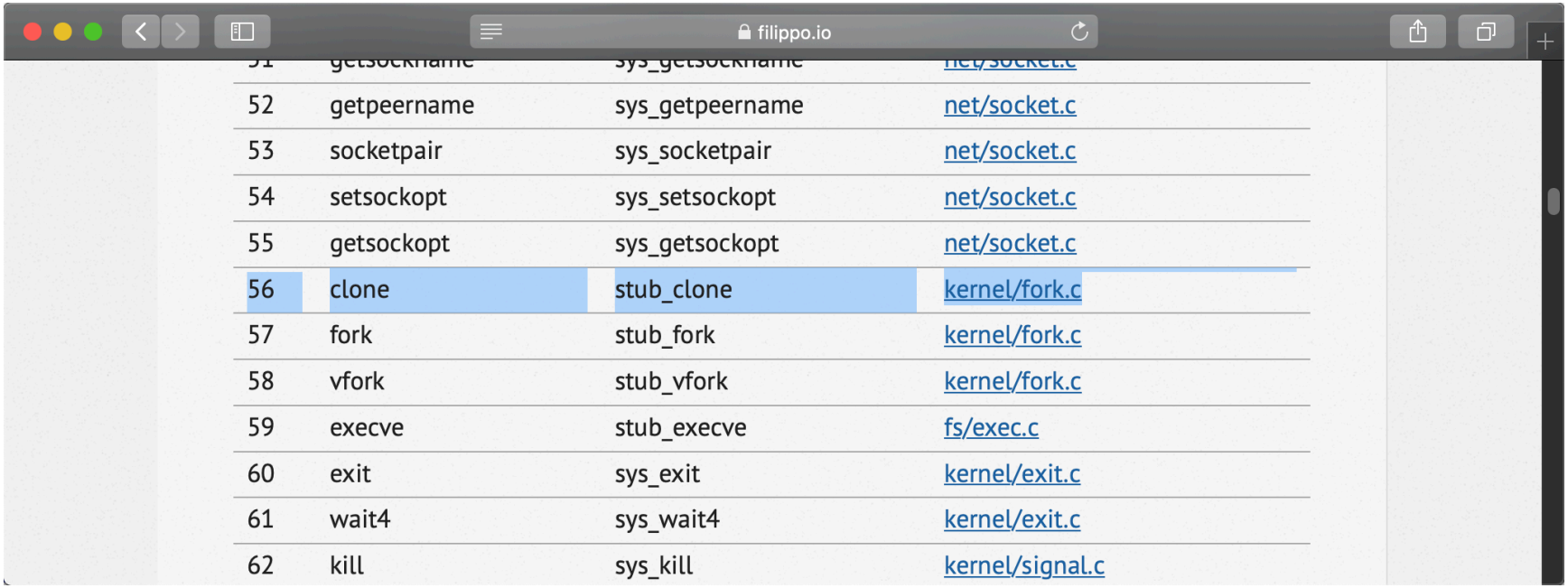
Thread creation: clone(), not fork()

- Create new threads in Linux
  - Use `clone()` to create threads instead of using fork()
  - `clone()` is usually not called directly but from some threading libraries, such as pthread.
  - http://man7.org/linux/man-pages/man2/clone.2.html

```
int main(int argc, char *argv[])
{
        char *stack;                    /* Start of stack buffer */
        char *stackTop;                 /* End of stack buffer */
        pid_t pid;
        struct utsname uts;

        if (argc < 2) {
                fprintf(stderr, "Usage: %s <child-hostname>\n", argv[0]);
                exit(EXIT_SUCCESS);
        }

        /* Allocate stack for child */

        stack = malloc(STACK_SIZE);
        if (stack == NULL)
                errExit("malloc");
        stackTop = stack + STACK_SIZE;  /* Assume stack grows downward */

        /* Create child that has its own UTS namespace;
           child commences execution in childFunc() */

        pid = clone(childFunc, stackTop, CLONE_NEWUTS | SIGCHLD, argv[1]);
        if (pid == -1)
                errExit("clone");
        printf("clone() returned %ld\n", (long) pid);

        /* Parent falls through to here */
```

Clone system call

- No. 56, https://filippo.io/linux-syscall-table/

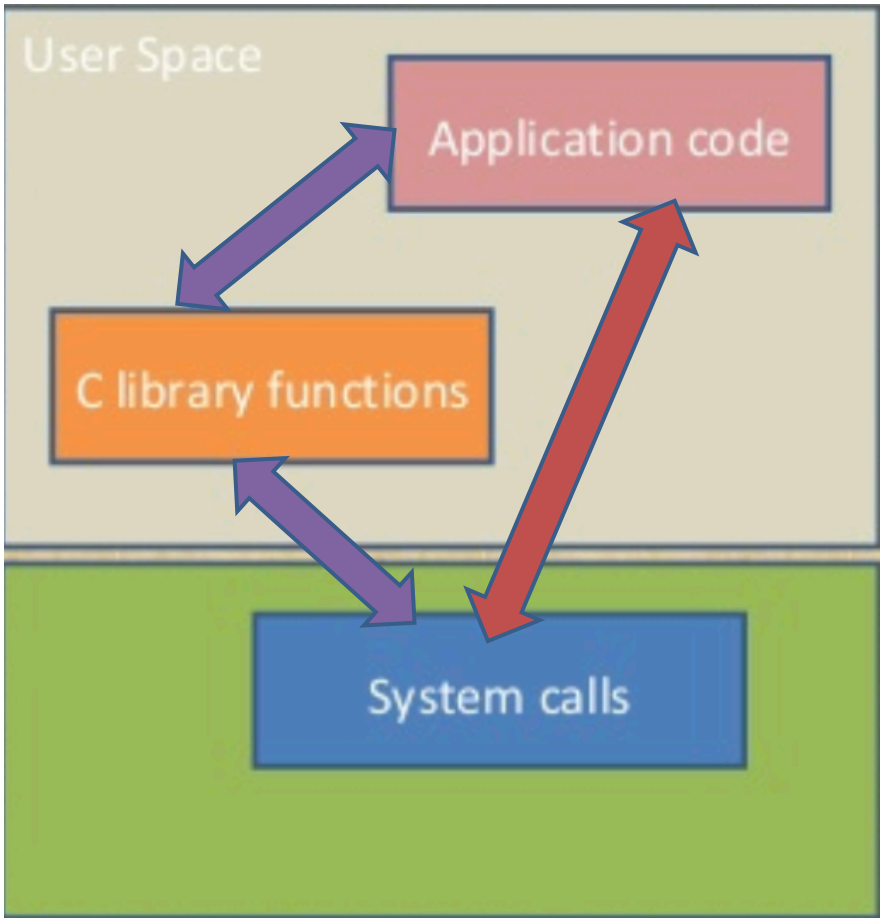


52	getpeername	sys_getpeername	net/socket.c
53	socketpair	sys_socketpair	net/socket.c
54	setsockopt	sys_setsockopt	net/socket.c
55	getsockopt	sys_getsockopt	net/socket.c
56	clone	stub_clone	kernel/fork.c
57	fork	stub_fork	kernel/fork.c
58	vfork	stub_vfork	kernel/fork.c
59	execve	stub_execve	fs/exec.c
60	exit	sys_exit	kernel/exit.c
61	wait4	sys_wait4	kernel/exit.c
62	kill	sys_kill	kernel/signal.c

Clone system call

Pthread_create()
in the user space

Clone()
in the kernel



Pthread Creation Example

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

void *myThread1(void)
{
        int i;
        for (i=0; i<3; i++)
        {
                printf("This is the 1st pthread.\n");
                sleep(1);
        }
}

int main()
{
        int ret=0;
        pthread_t id1;

        printf("This is main thread!\n");

        ret = pthread_create(&id1, NULL, (void*)myThread1, NULL);
        if (ret)
        {
                printf("Create pthread error!/n");
                return 1;
        }

        pthread_exit(NULL);

        return 0;
}
```

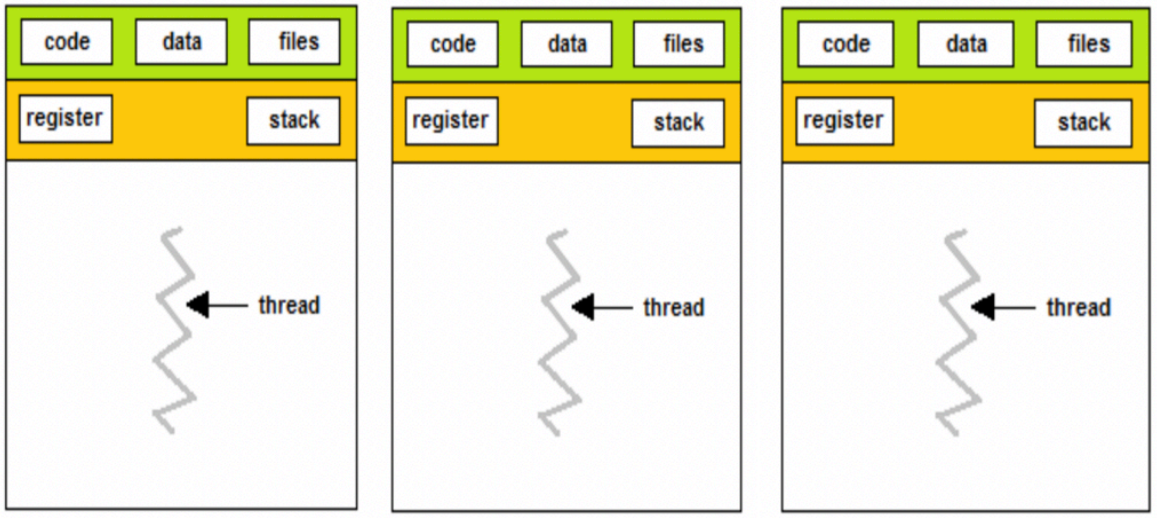
https://github.com/kevinsuo/CS3502/blob/master/pthread_create.c

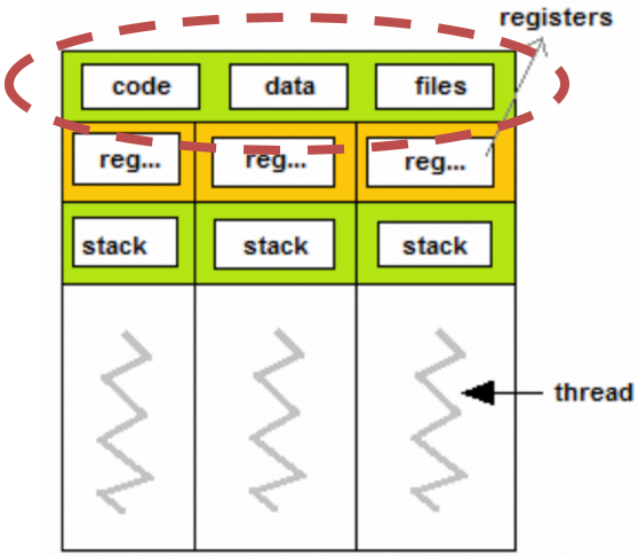
```
pi@raspberrypi ~/Downloads> ./test.o
This is main thread!
This is the 1st pthread.
This is the 1st pthread.
This is the 1st pthread.
```

Clone() in the kernel

Fork vs clone

- Fork:
  - **All resources** in PCB are **copied** from parent process to child process
- Clone:
  - The resources in PCB are **partly copied** from one thread to another thread (the copied context is different in vfork)



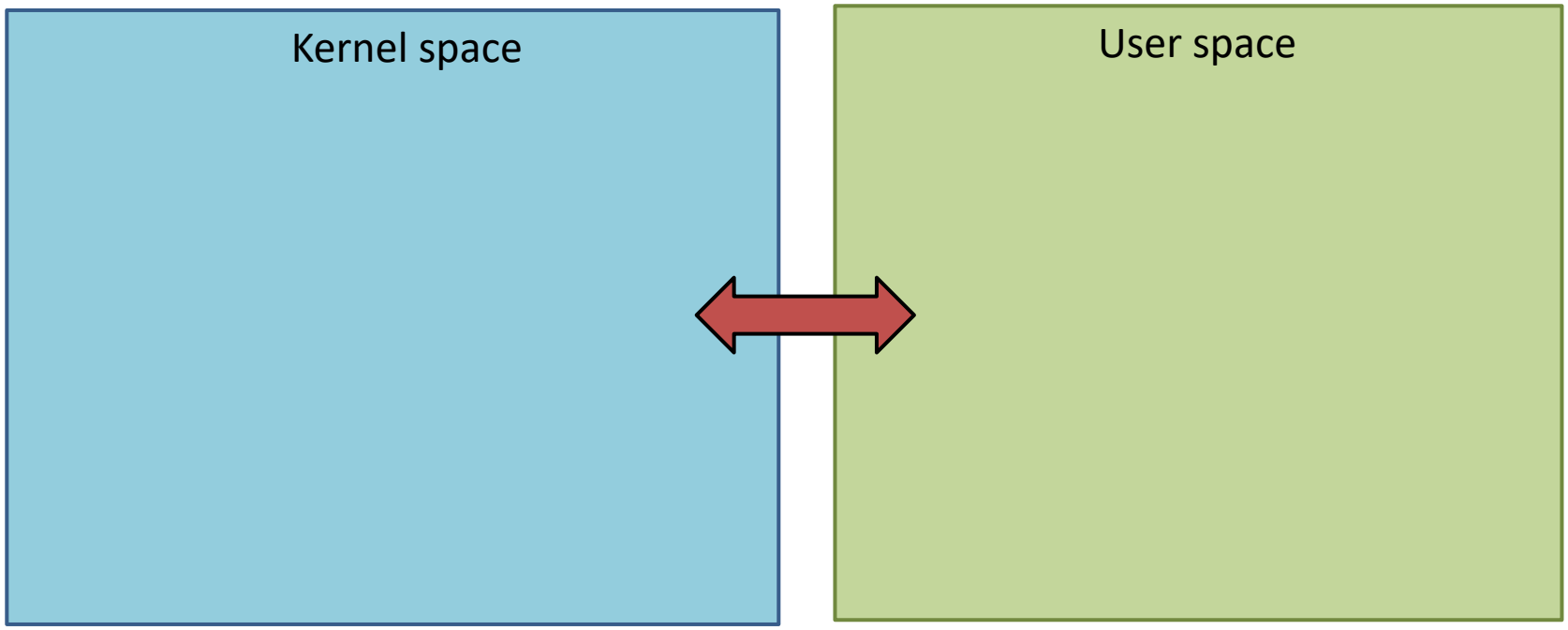


Outline

- What is thread?
  - Multiple thread application
  - Thread vs Process
  - Advantage and disadvantage of thread
- Thread in Linux
- Thread design
  - Kernel space vs User space
  - Local thread vs Global thread scheduling

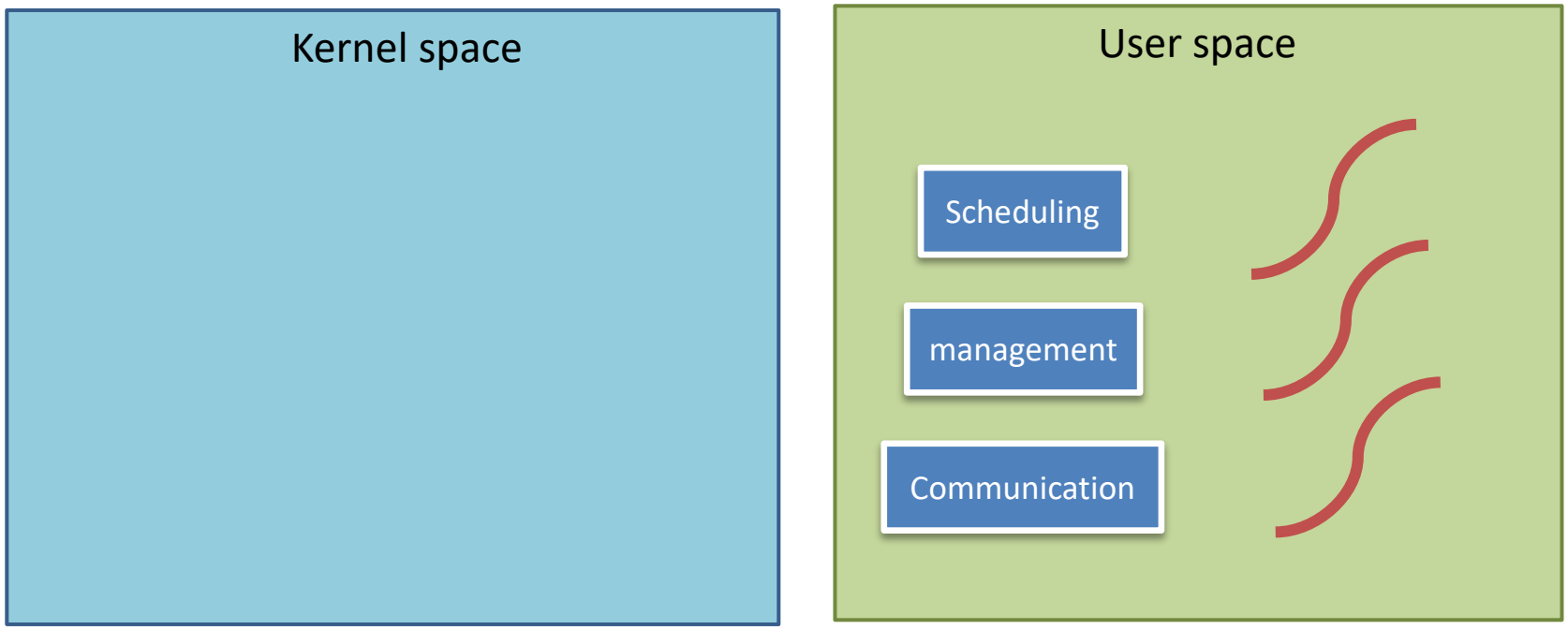
Thread design

- When you have multiple threads, where to put them?



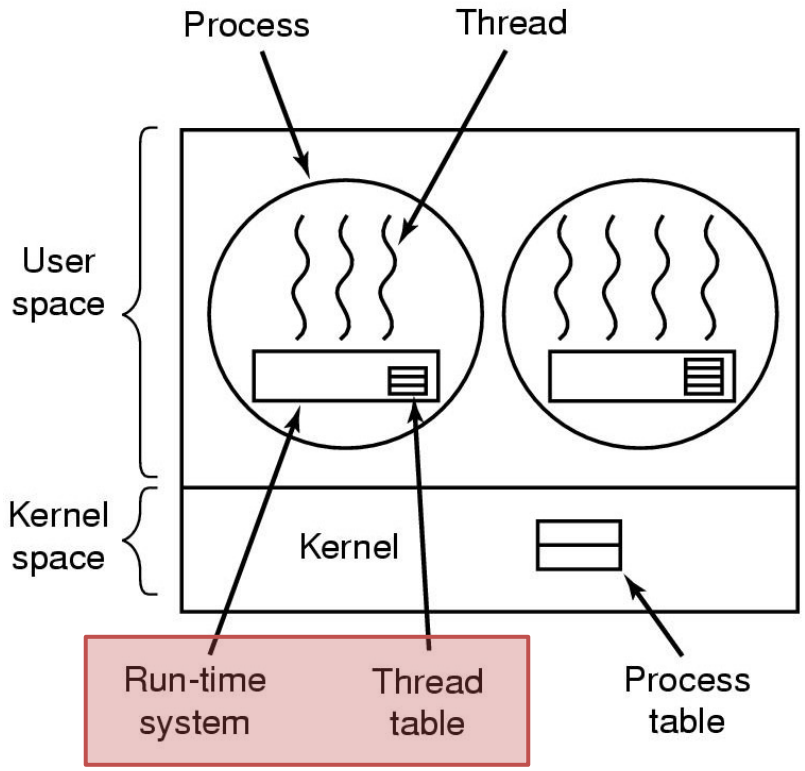
User-level Thread

- When you have multiple threads, where to put them?



User-level Thread

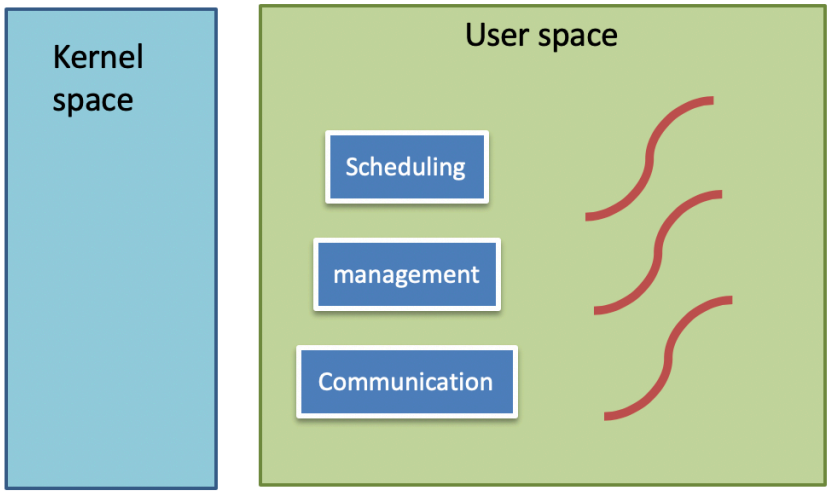
- User-level threads: the kernel knows nothing about them and managed by applications



A user-level threads package

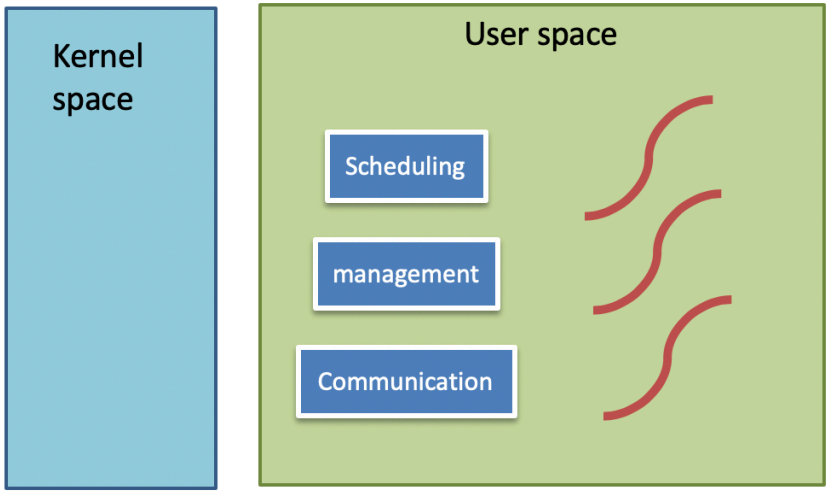
User-level Thread - Discussions

- Advantages
  - No OS thread-support need
  - Lightweight: thread switching vs. process switching
    - Less memory
    - Faster context switch
    - Easier to communicate and data sharing
    - Management in the user space



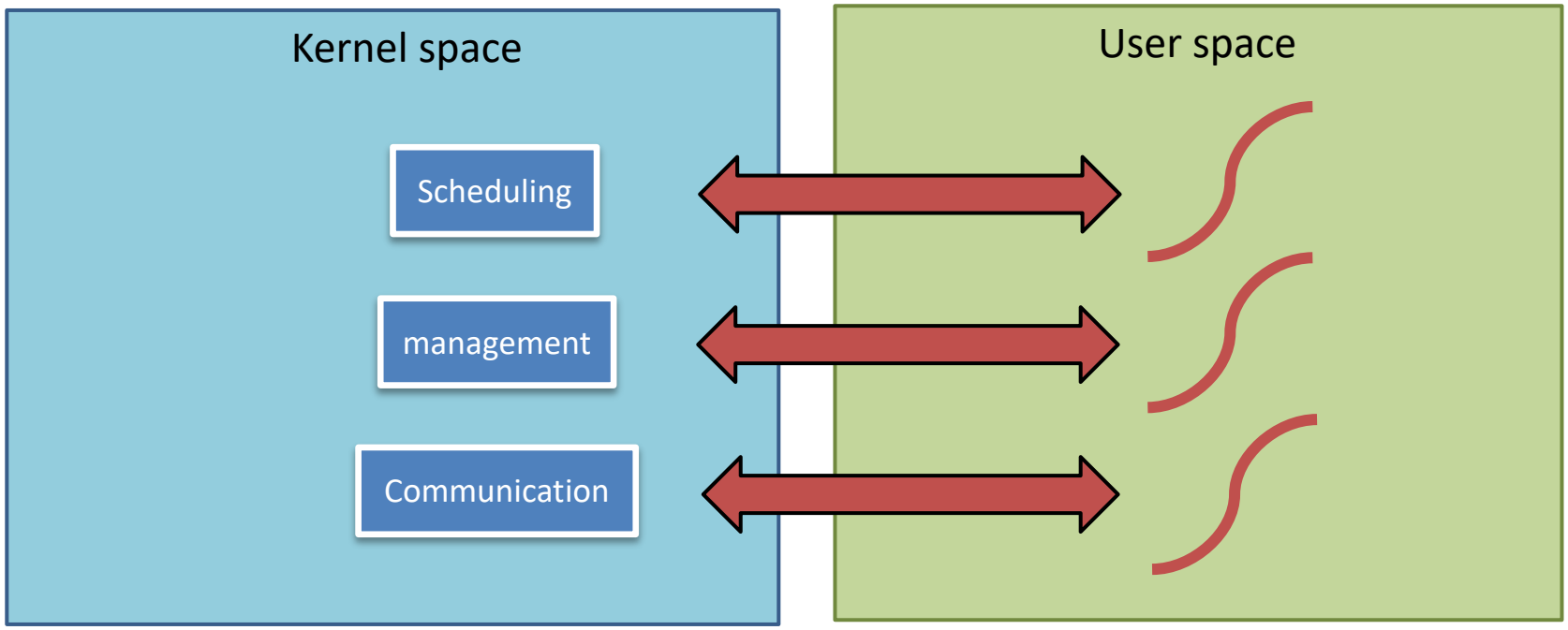
User-level Thread - Discussions

- Disadvantages (less kernel support, need to implement all logic by application itself)
  - Scheduling:
    - How blocking system calls implemented? Called by a thread?
    - How to change blocking system calls to non-blocking?
  - Memory management:
    - How to deal with page faults?
  - Interrupt
    - How to stop a thread from running forever? No clock interrupts



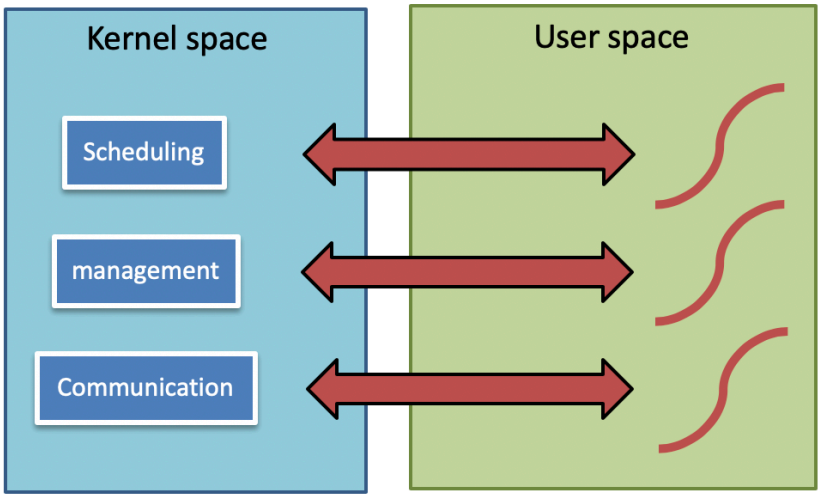
Kernel-level Thread

- When you have multiple threads, where to put them?



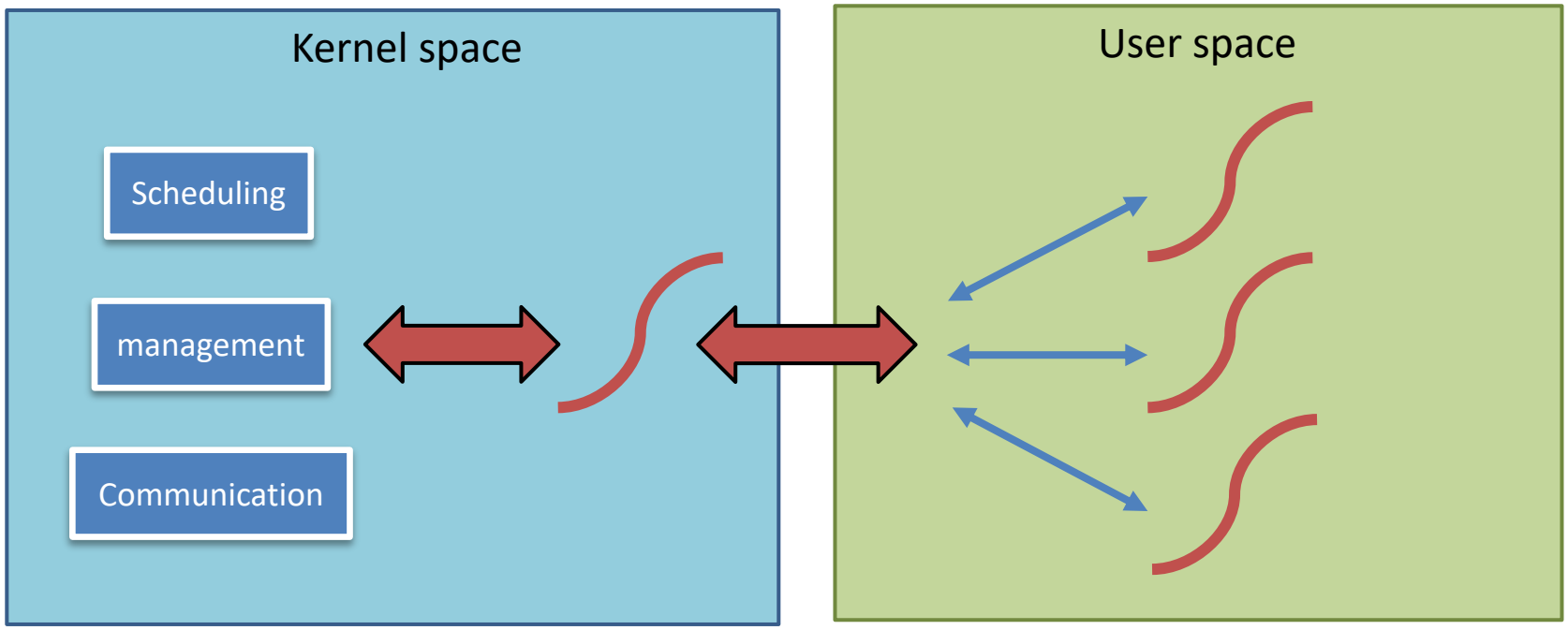
Implementing Threads in the Kernel

- Kernel-level threads: managed by the kernel
  - Threads known to OS, use OS service directly
    - Scheduling, memory management, storage, I/O, etc.
  - Slow
    - Trap into the kernel mode
  - Expensive to create and switch
    - Create memory inside the kernel
    - Context switch between the kernel and user space



Threads in Hybrid-Space

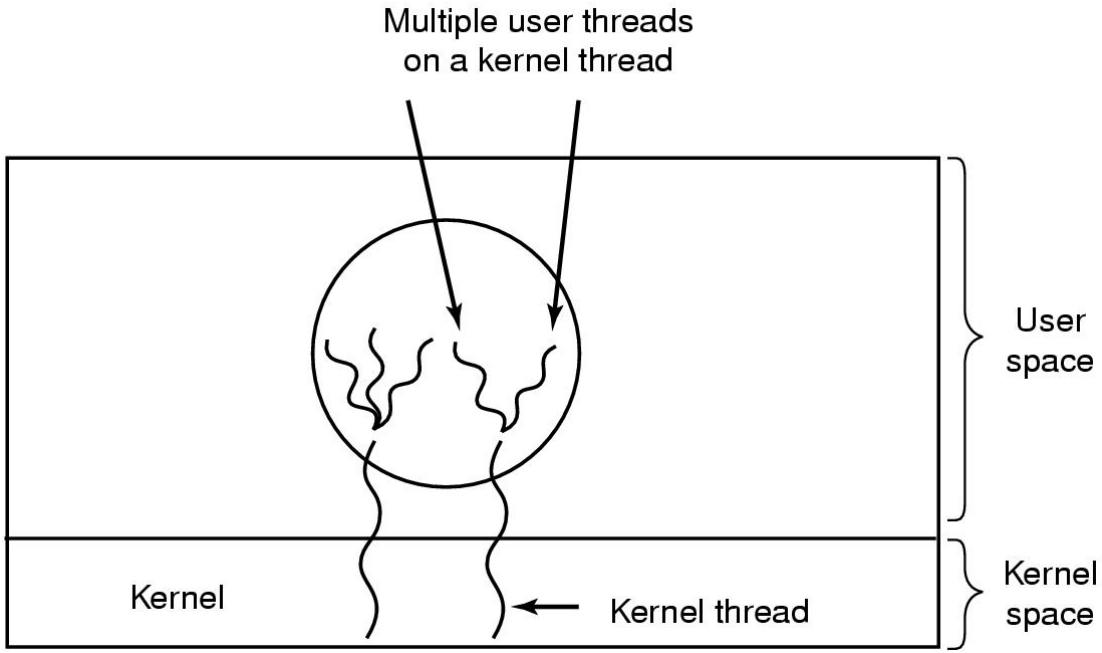
- When you have multiple threads, where to put them?



Hybrid Implementations

- Use kernel-level threads and then *multiplex* user-level threads onto some or all of the kernel-level threads

- Multiplexing user-level threads onto kernel-level threads
- Enjoy the benefits of user and kernel level threads
- Too complex !



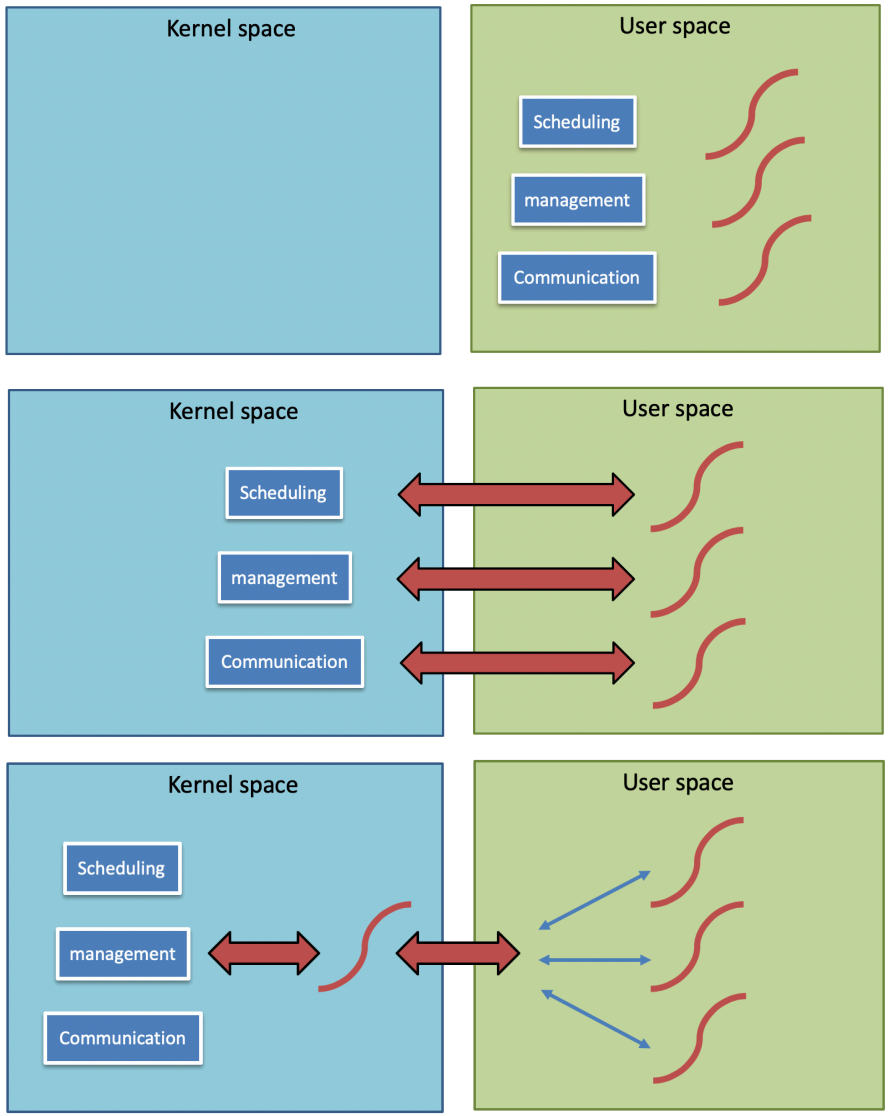
Multiplexing user-level threads onto kernel- level threads

Thread in user level/kernel level/hybrid

user

kernel

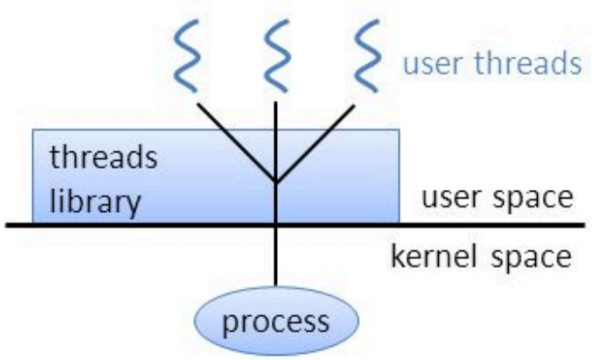
hybrid



Threading Models

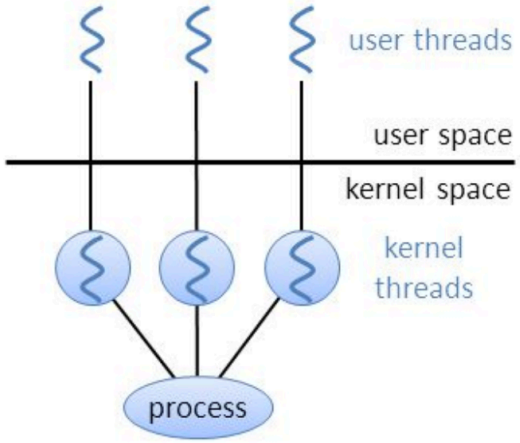
- N:1 (User-level threading)
  - GNU Portable Threads

    https://www.gnu.org/software/pth/

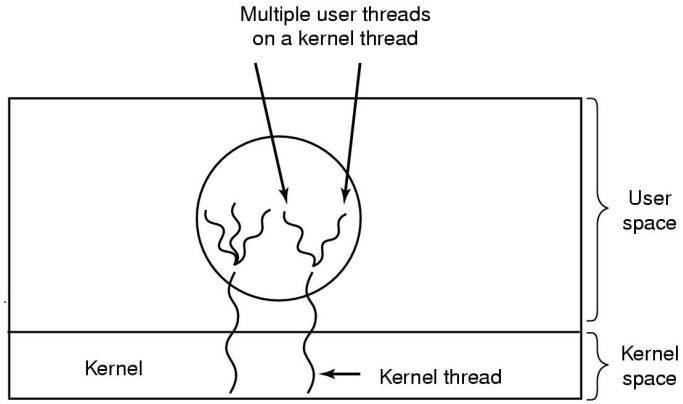


- 1:1 (Kernel-level threading)
  - Native POSIX Thread Library (Pthread)

    https://en.wikipedia.org/wiki/Native_POSIX_Thread_Library
    Linux/MacOS/Windows

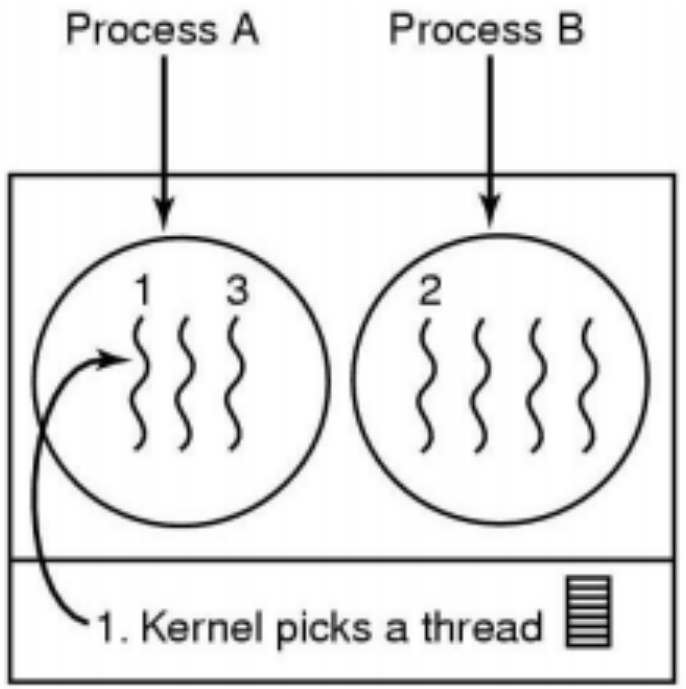


- M:N (Hybrid threading)
  - Solaris https://www.oracle.com/solaris/solaris11/



Thread design

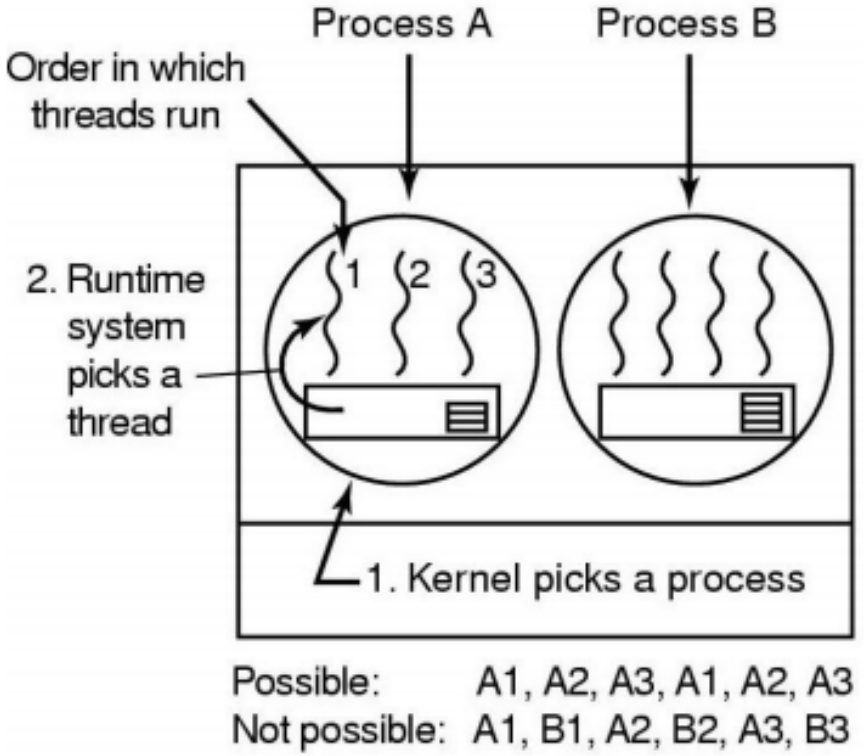
- When you have multiple threads, by what order should we execute them? - -> Scheduling



Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

Local Thread Scheduling

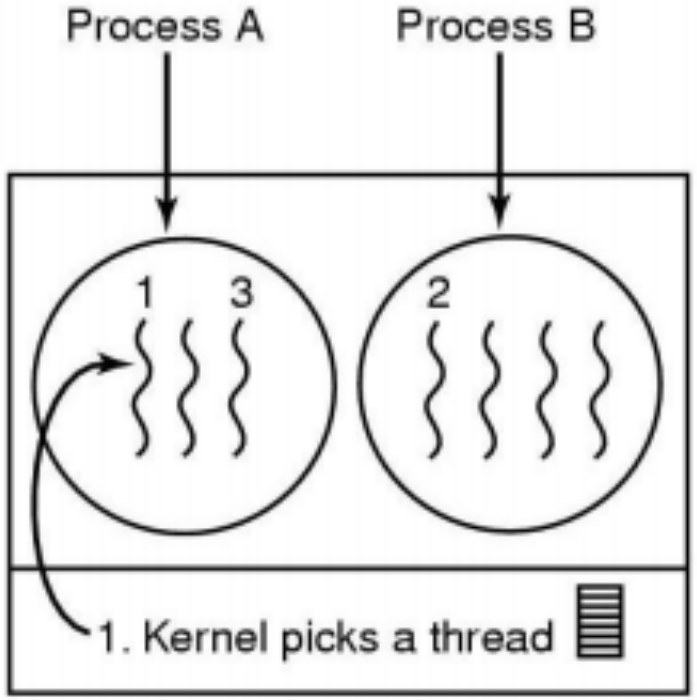
- Next thread is picked from among the threads belonging to the **current process**
- Each process gets a timeslice from kernel.
- Then the timeslice is divided up among the threads within the current process
- Scheduling decision requires only **local knowledge** of threads within the current process. It can be implemented by using **user or kernel level threads**



For example, say process timeslice may be N ms, and each thread within the process runs for N/3 ms per cycle

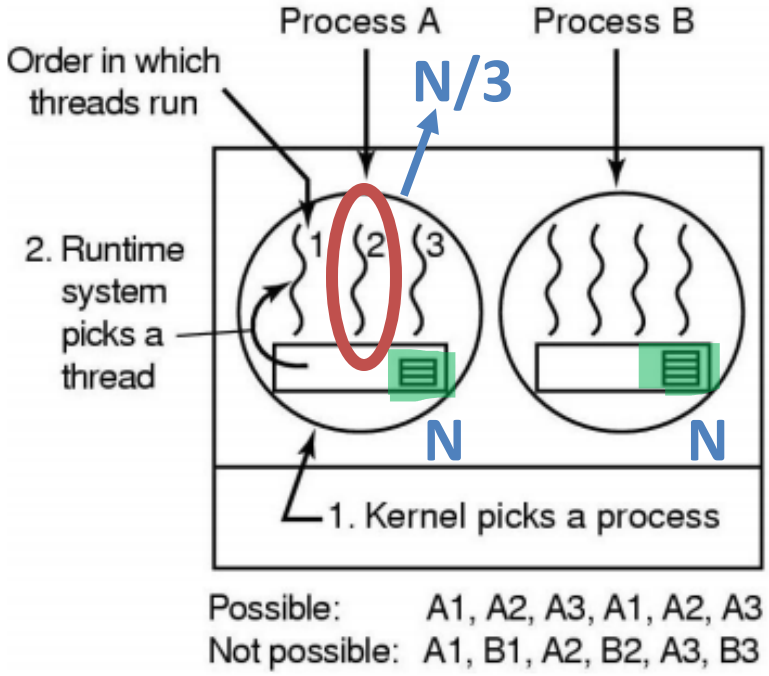
Global Thread scheduling

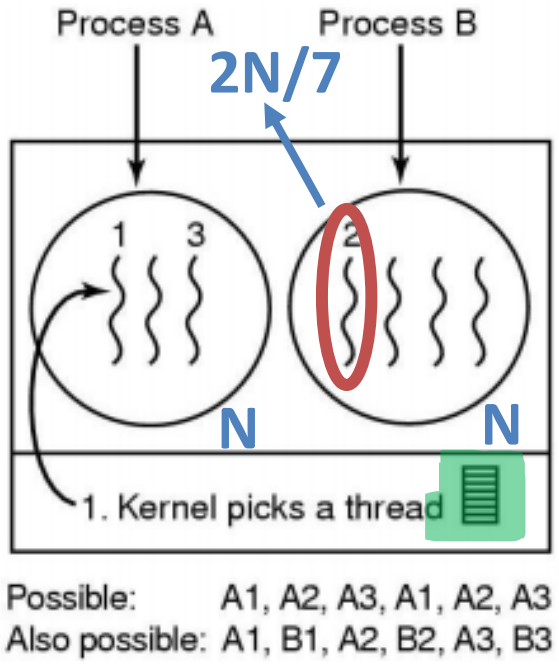
- Next thread to be scheduled is picked up from **ANY process** in the system.
  - Not just the current process
- Timeslice is allocated at the granularity of threads
  - No notion of per-process timeslice
- Global scheduling can be implemented **only with kernel-level threads**
  - Picking the next thread requires global knowledge of threads in all processes.



For example, say process timeslice may be N ms, For example each thread runs for 2N/7 ms per cycle

Local Thread Scheduling vs Global Thread scheduling





1, next thread

2, time slice for each thread

3, implementation: user/kernel level vs. only kernel level thread

Multiple threads → Pandora's Concurrency Box

- Multiple threads → Concurrency
- The illusion of concurrency is both **powerful** and **useful**:
  - It helps us think about how to structure our applications - -> multiple threads apps
  - It hides latencies caused by hardware devices - -> accelerate execution

- Unfortunately, concurrency also creates **problems**:
  - **Coordination**: how do we enable efficient communication between the multiple threads involved in performing a single task?
  - **Correctness**: how do we ensure that shared information remains consistent when being accessed by multiple threads concurrently?

Concurrency example review

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

Output:

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012     // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298     // what the??
```

Concurrency and multi-threads

- Unless precisely synchronized, threads may:
  - Be run in **any order**,
  - Be stopped and restarted at **any time**,
  - Remain stopped for **arbitrary lengths of time**.

} **Problems**

- Generally these are **good things** - - the operating system is responsible for how to allocate resources.

Talk about CPU scheduling, memory management, lock, synchronization in future talks

Conclusion

- What is thread?
  - Multiple thread application
  - Thread vs Process
  - Advantage and disadvantage of thread
- Thread in Linux
- Thread design
  - Kernel space vs User space
  - Local thread vs Global thread scheduling