Operating Systems
Threads

Yanyan Zhuang

Department of Computer Science

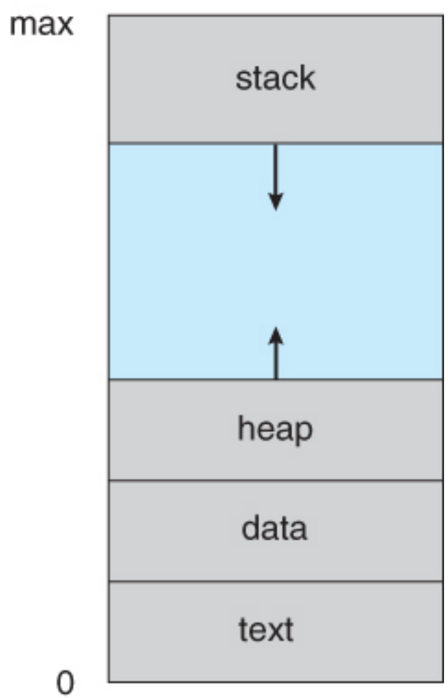
http://www.cs.uccs.edu/~yzhuang

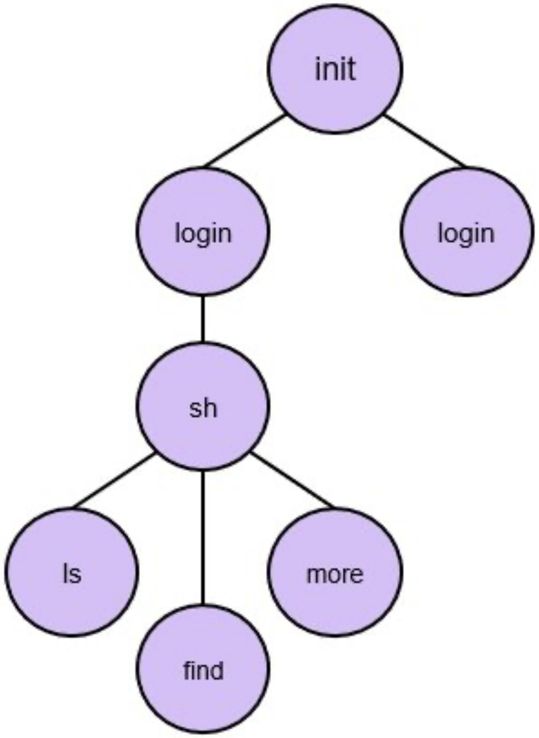
Recap of the Last Class

- Processes
  - A program in execution
  - 5 (3)-state process model
  - Process control block
- Linux processes
  - The `task_struct` structure

Thread and Multithreading

- Process
  - Resource grouping and execution





So far, we assumed 1 thread of execution (except fork)

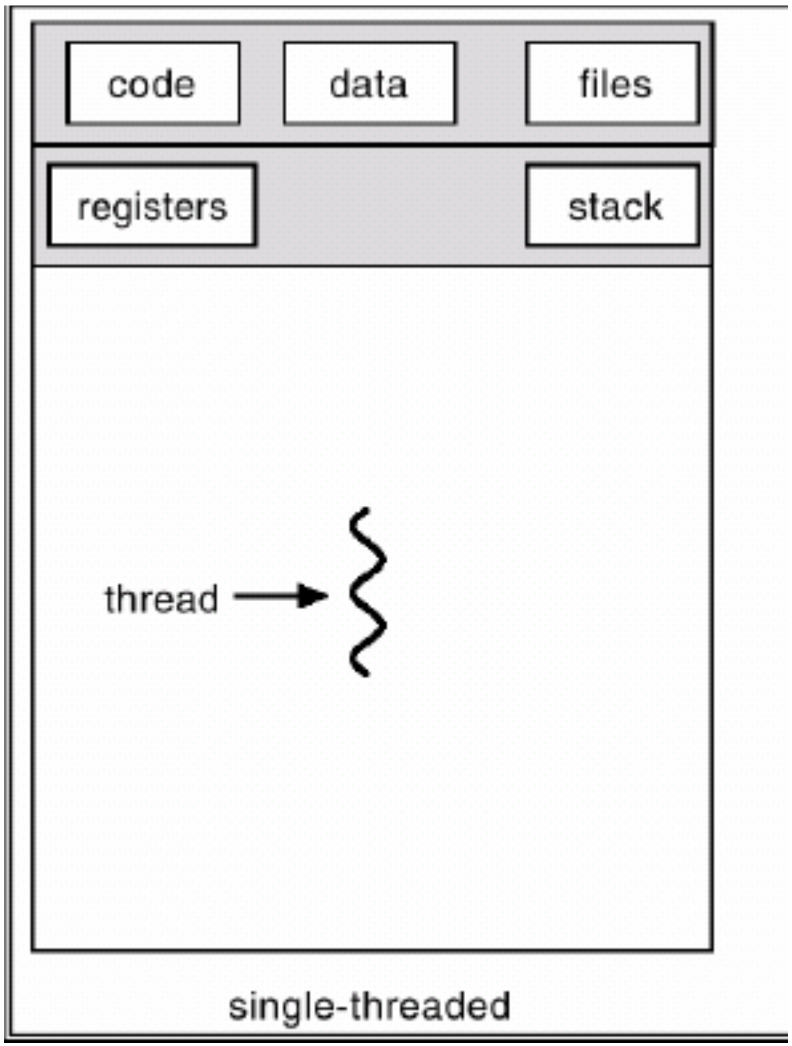
Thread and Multithreading

- Process
  - Resource grouping and execution

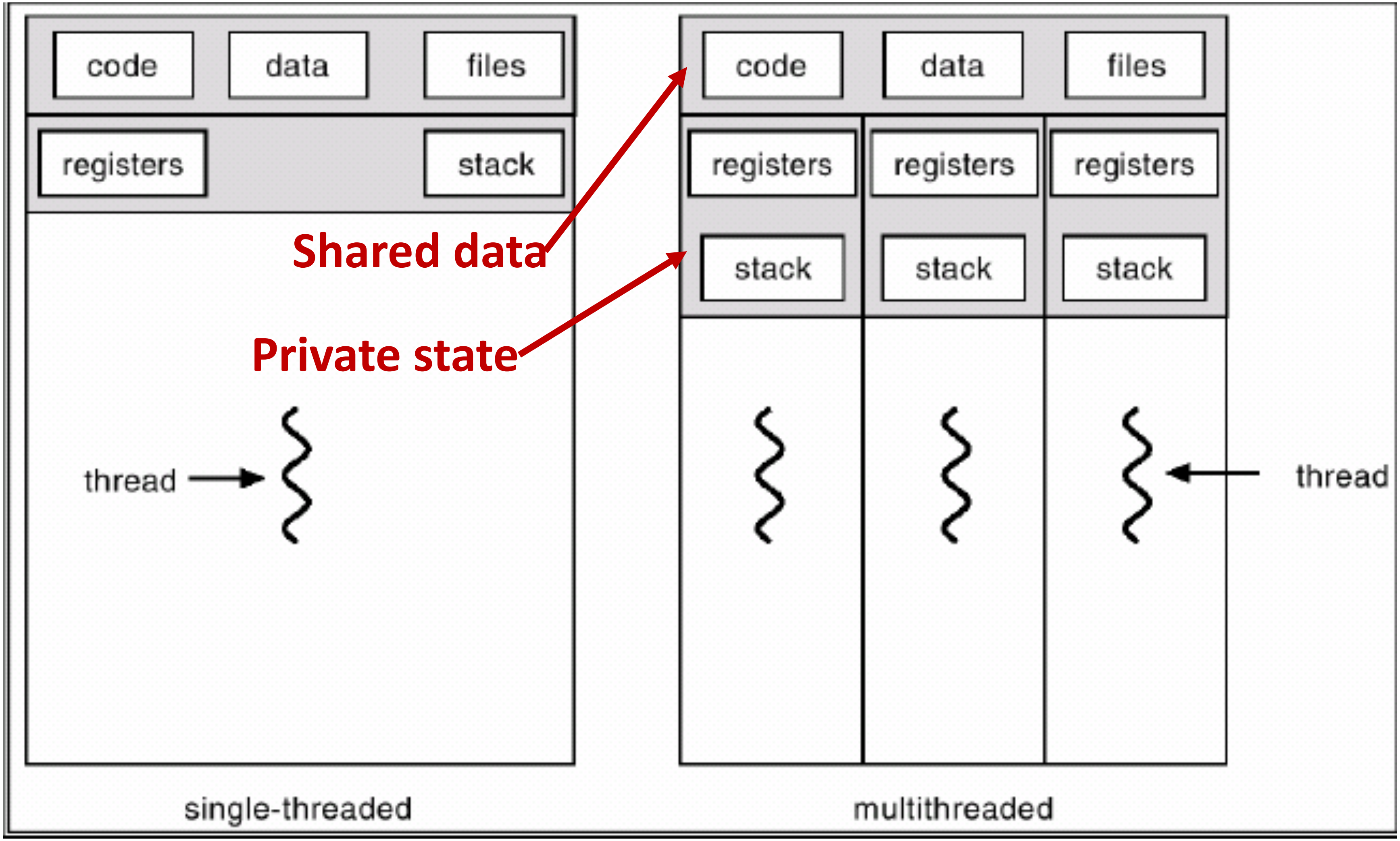
- Thread (or multi-threaded execution)

- Each one is doing something
- They **share** the same data but look at different parts
- They have **private** state but can communicate easily
- They must coordinate!

An Illustration: from OS point of view



An Illustration: from OS point of view



Thread and Multithreading

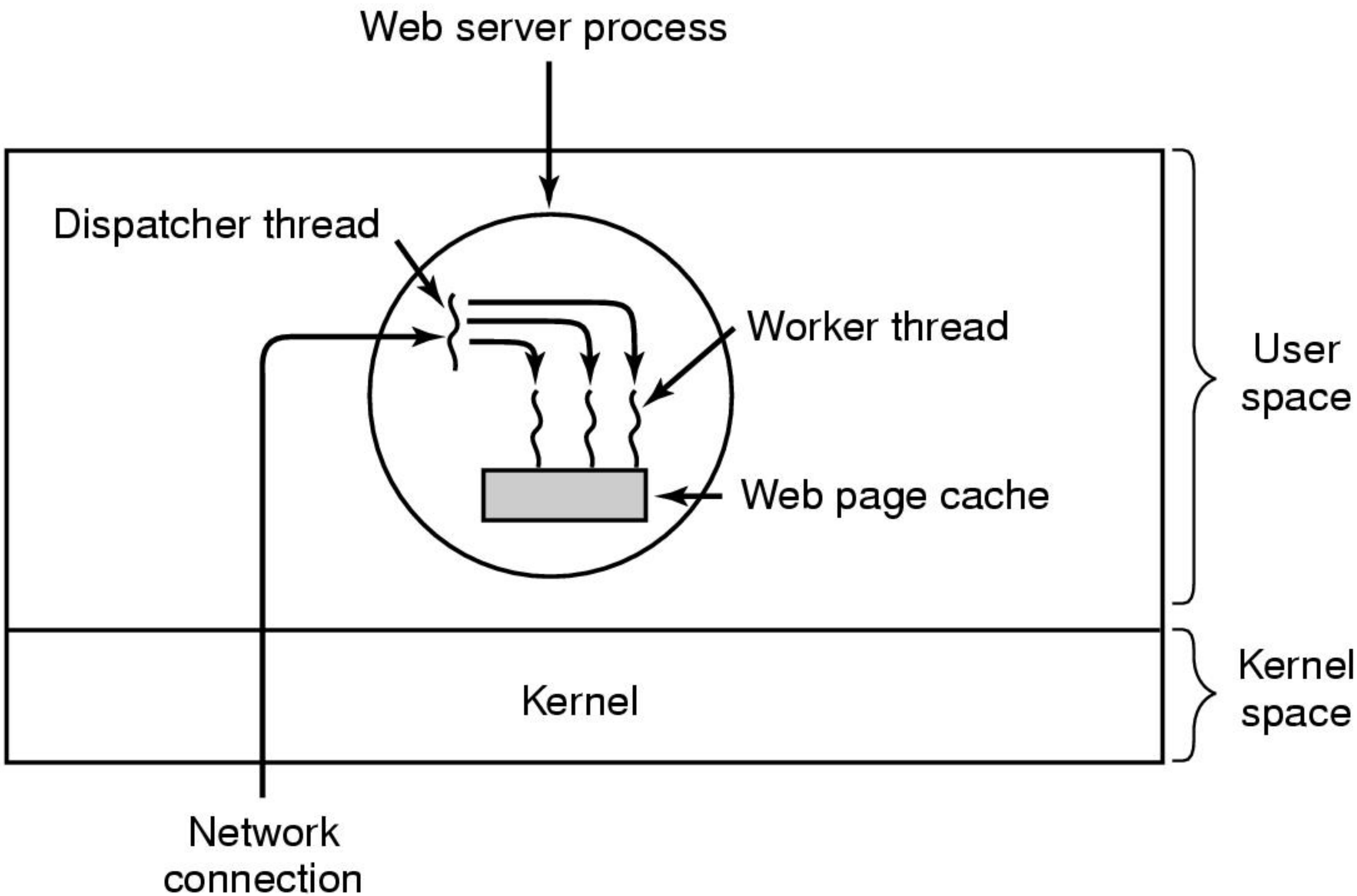
- Process
  - Resource grouping and execution
- Thread
  - A program in execution without dedicated address space: *threads of the same process share address space*
  - Efficient communication
    - Inter-**thread** communication can be carried out via shared data objects within the shared address space
    - Inter-**process** communication usually requires other OS services
  - Efficient creation
    - Only create thread context

Processes v.s. Threads: A Closer Look

- Threads
  - No data segment or heap
  - Multiple can coexist in a process
  - Share code, data, heap, and I/O
  - Have own stack and registers
  - Inexpensive to create
  - Inexpensive context switching
  - Efficient communication

- Processes
  - Have data/code/heap
  - Include at lease one thread
  - Have own address space, isolated from other processes
  - Expensive to create
  - Expensive context switching
  - IPC can be expensive

Thread Usage



A multithreaded Web server.

Thread interfaces in UNIX: POSIX threads

- UNIX systems
  - IEEE Portable Operating System Interface (POSIX)
  - Implementations of threads that adhere to this standard are referred to as POSIX threads, or Pthreads

Pthread function

- int pthread_create(pthread_t *thread,

  const pthread_attr_t *attr,

  void *(*start_routine) (void *),

  void *arg)

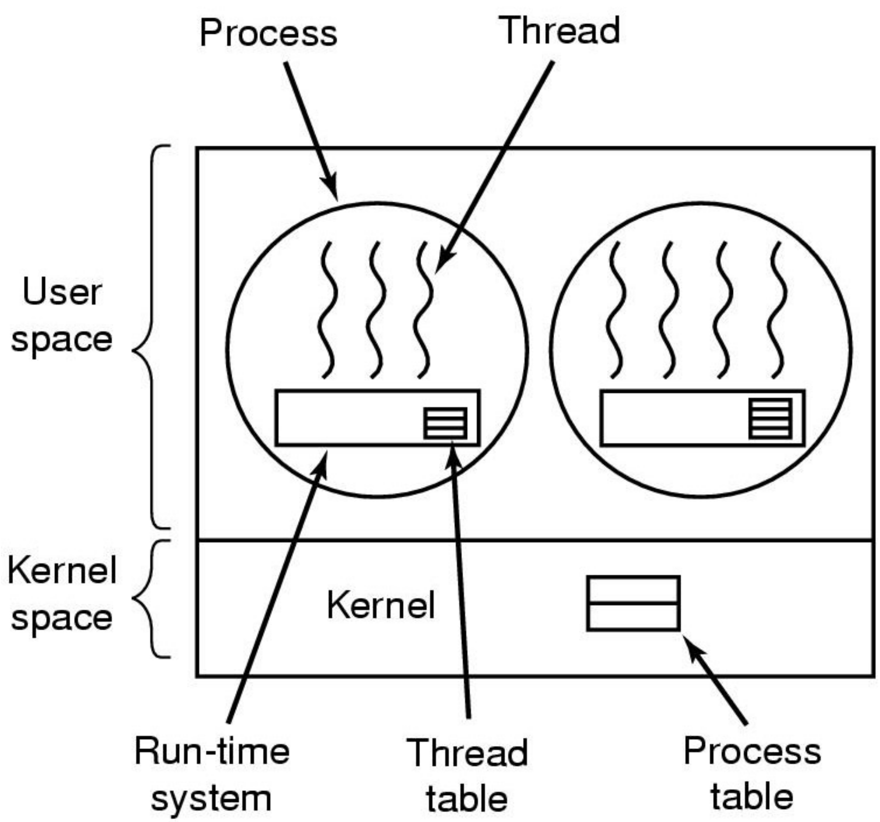
  - Create a new thread, with attributes attr (attributes can include scheduling policies, stack size, etc.)
  - The thread is created by executing start_routine with arg as the only argument
  - Upon success, stores the ID of the thread in the location referenced by thread

An Example

```
void *my_thread(void *arg)
{
        int *tid = (int *)arg;
        printf("Hello from child thread: %d\n", *tid);
        return NULL;
}
int main(int argc, char *argv[]){
        pthread_t threads[NR_THREADS];
        for (i = 1; i < NR_THREADS; i++){
                printf("In main: creating thread %ld\n", i);
                tid[i] = i;
                pthread_create(&threads[i], &a, my_thread, &tid[i]);
        }
}
```

Implementing Threads in User-Space

- User-level threads: the kernel knows nothing about them



A user-level threads package

- OS thinks there's only a single-threaded process
- Threads in same process don't involve multiplexing between processes so no kernel privilege required

User-level Thread - Discussions

- Advantages
  - No OS thread-support needed
  - Lightweight: thread switching vs. process switching
    - Local procedure (no mode switch) vs. system call (trap to kernel)
  - Each has its own customized scheduling algorithms
    - thread_yield()

User-level Thread - Discussions

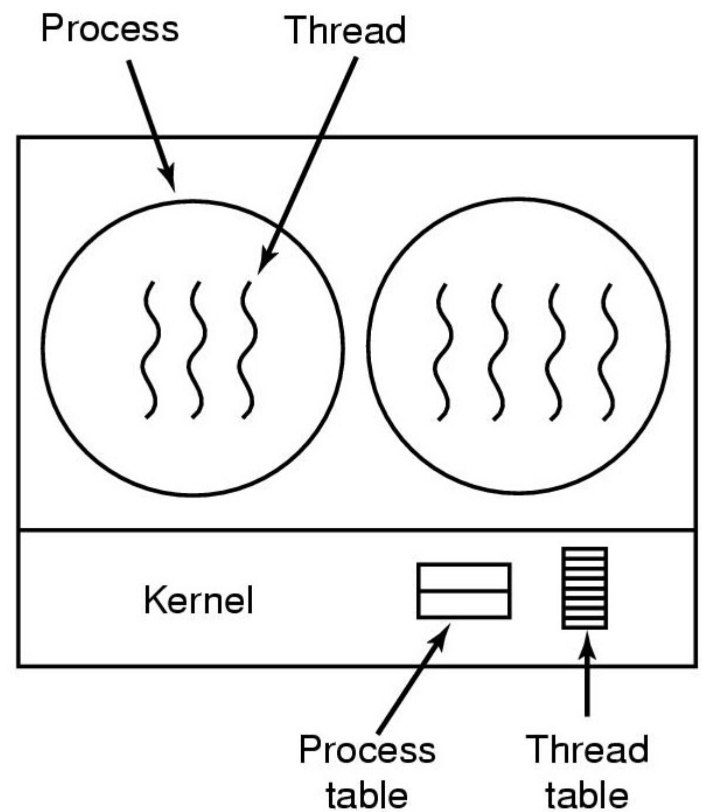
- Advantages
  - No OS thread-support needed
  - Lightweight: thread switching vs. process switching
    - Local procedure (no mode switch) vs. system call (trap to kernel)
  - Each has its own customized scheduling algorithms
    - thread_yield()

- Disadvantages
  - How blocking system calls implemented? Called by a thread?
    - Goal: to allow each thread to use blocking calls, but to prevent one blocked thread from affecting the others
  - How to deal with page faults?
  - How to stop a thread from running forever?
    - No clock interrupts in a single process

Implementing Threads in the Kernel

- Kernel-level threads: when a thread blocks, kernel re-schedules another thread
  - Threads known to OS
    - Scheduled by the scheduler
  - Slow
    - Trap into the kernel mode
  - Expensive to create and switch
    - Less expensive if in the same process
      - Registers, PC, stack pointer need to be created/changed
      - Not the memory info

Any problems?



A threads package managed by the kernel

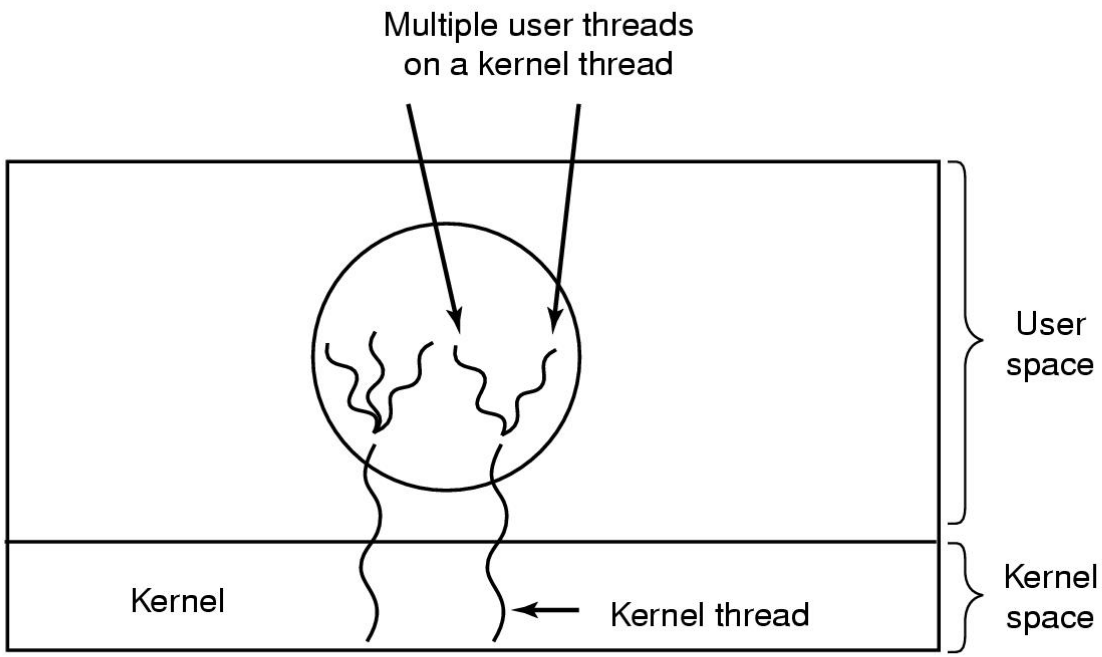
What happens when forking a multithreaded process

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

Threading Models

- N:1 (User-level threading)
  - N user threads that look like 1 to the OS kernel
- 1:1 (Kernel-level threading)
  - Each user-thread is paired with a kernel-thread (aka native thread)
- M:N (Hybrid threading)
  - Solaris

Threads in Linux

- Thread control block (TCB)
  - The `thread_stuct` structure
  - Includes registers and processor-specific context
- Linux treats threads like processes
  - Use `clone()` to create threads instead of using fork()
  - `clone()` is usually not called directly, but from some threading libraries

Summary

- Processes v.s. threads?
- Why threads?
  - Concurrency + lightweight
- Threading models
  - N:1, 1:1, M:N