CS 571 - Lecture 3

Threads

George Mason University

Spring 2010

Threads

- Overview
- Multithreading
- Example Applications
- User-level Threads
- Kernel-level Threads
- Hybrid Implementation
- Observing Threads

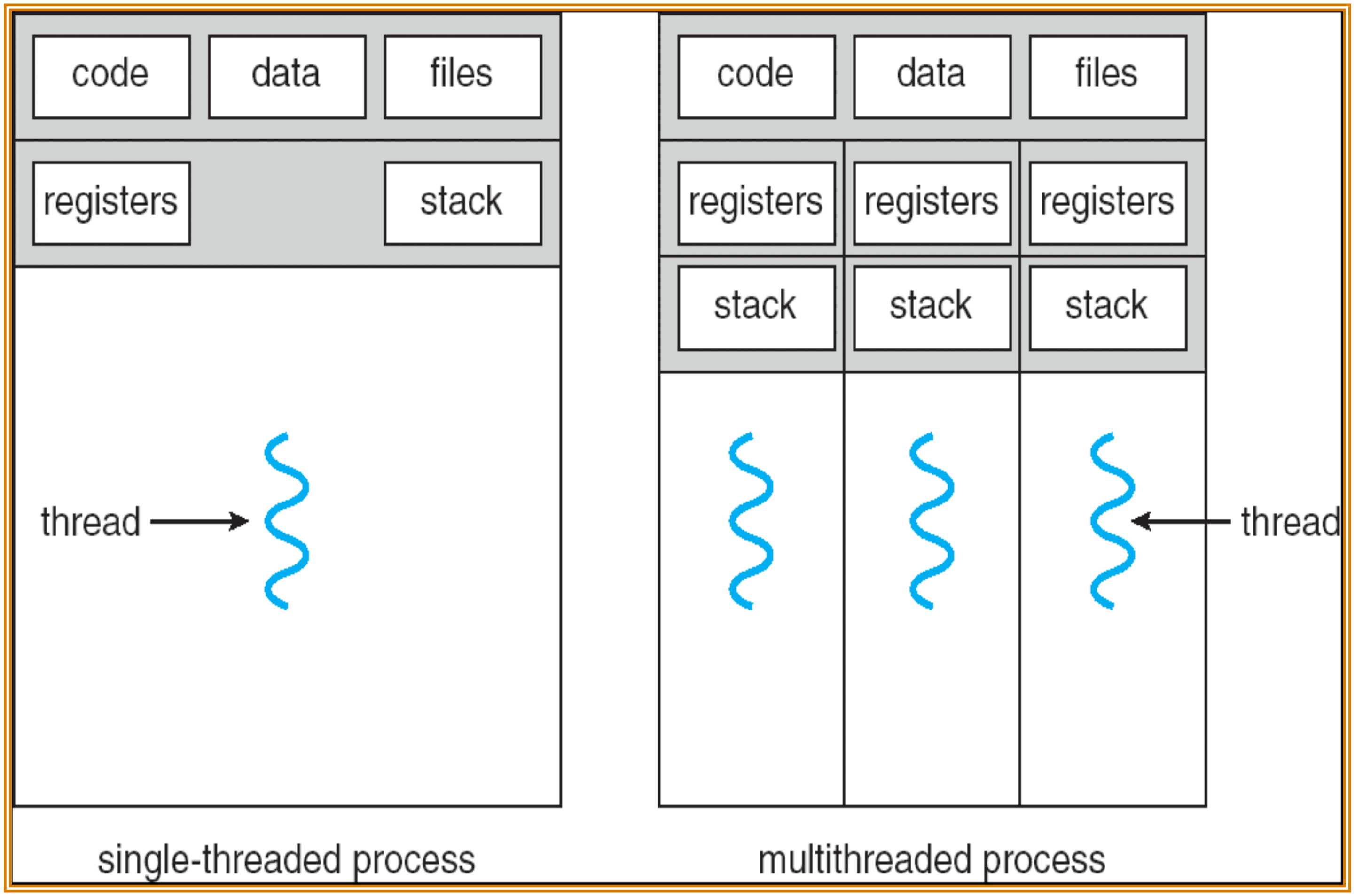
Threads

- A process, as defined so far, has only one *thread of execution.*
- *Idea:* Allow multiple threads of execution within the same process environment, to a large degree independent of each other.
  - Why? To take advantage of llism
- Multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer.

Threads (Cont.)

- **Multiple threads within a process will *share***
  - **The address space**
    - **and data**
  - **Open files**
  - **Other resources**
- **Potential for efficient and close cooperation**

Single and Multithreaded Processes



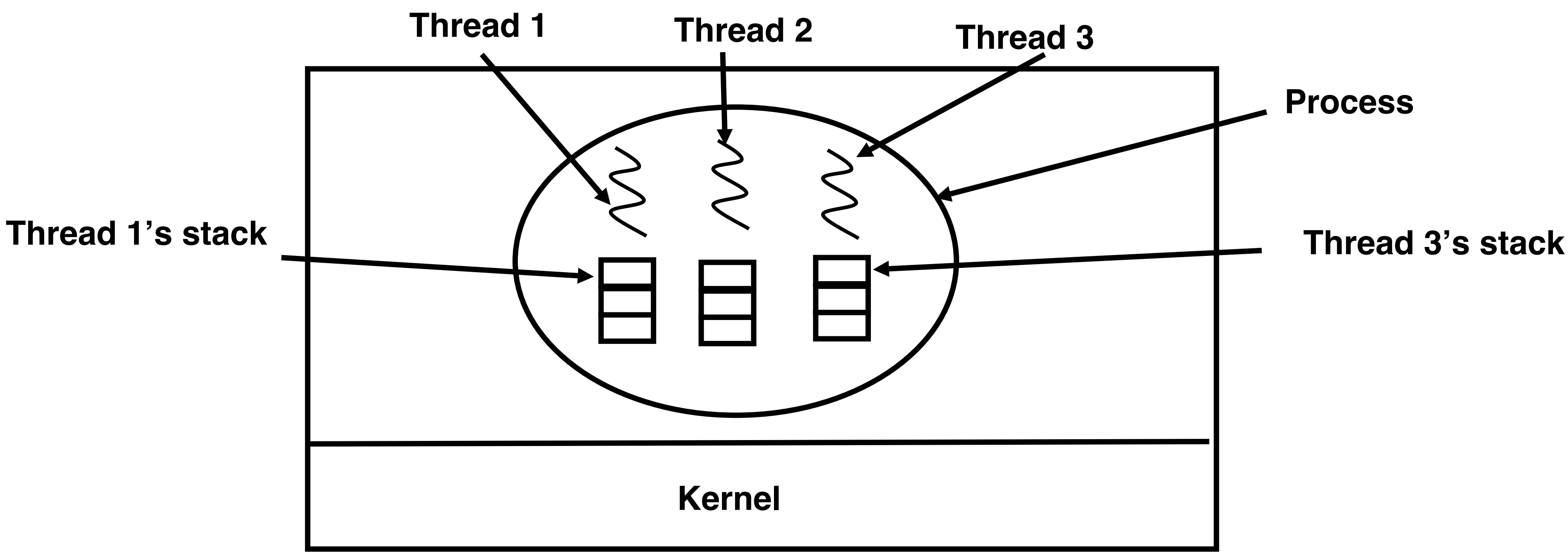
Multithreading

- When a multithreaded process is run on a single CPU system, the threads take turns running.
- All threads in the process have exactly the same address space.

Per Process Items	Per Thread Items
Address Space	Program Counter
Global Variables	Registers
Open Files	Stack
Accounting Information	State

Multithreading (Cont.)

- Each thread can be in any one of the several states, just like processes.
- Each thread has its own stack.



Benefits

- Responsiveness
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or performing a lengthy operation.
- Resource Sharing
  - Sharing the address space and other resources may result in high degree of cooperation
- Economy
  - Creating / managing processes is much more time consuming than managing threads.
- Better Utilization of Multiprocessor Architectures
  - in particular, CMT (SPARC), HyperThreading (Intel, AMD)
  - thread switching is FAST

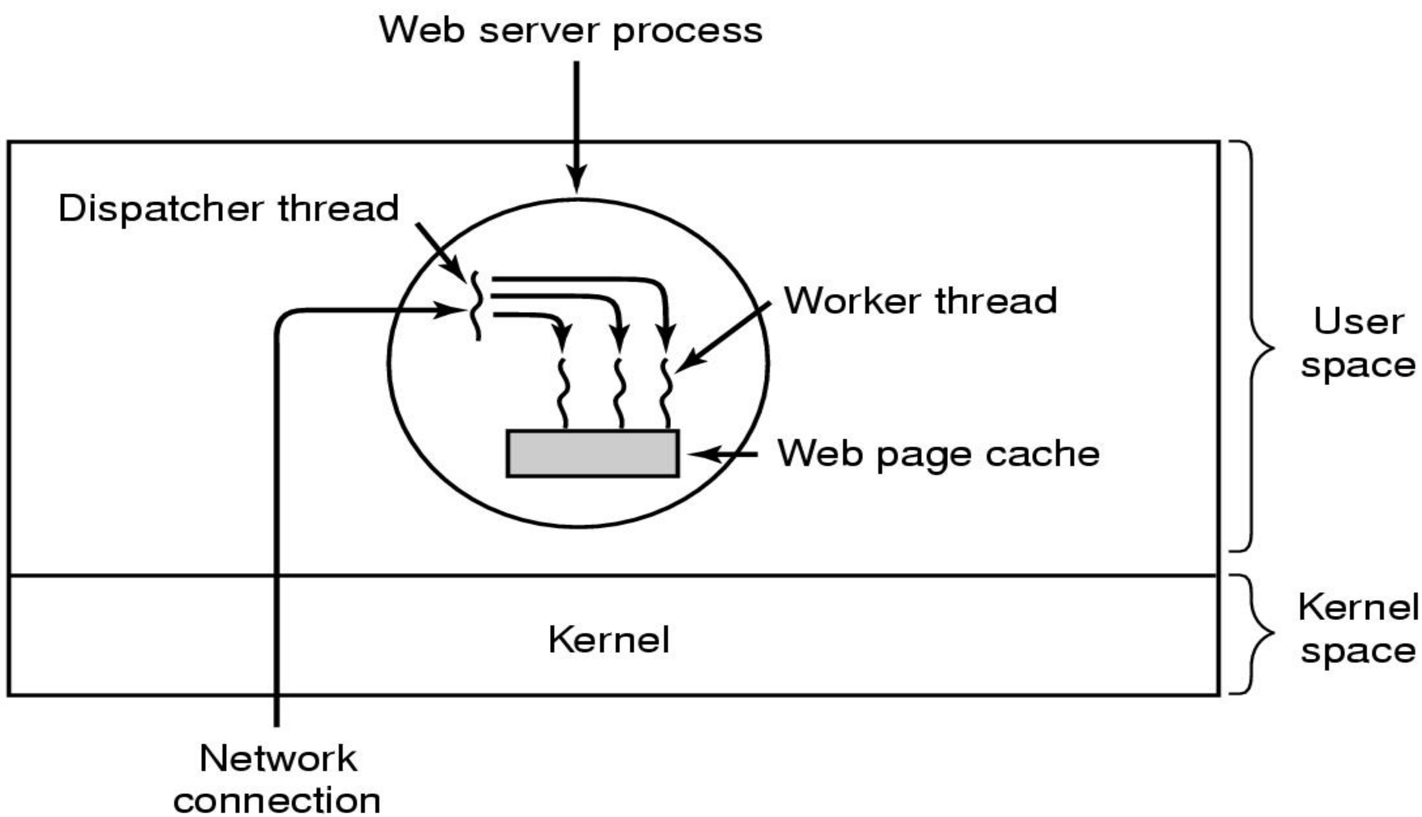
Example Multithreaded Applications

- **A word-processor with three threads**
  - **Re-formatting**
  - **Interacting with user**
  - **Disk back-up**

- **What would happen with a single-threaded program?**

Example Multithreaded Applications

- **A multithreaded web server**



Example Multithreaded Applications

- **The outline of the code for the dispatcher thread (a), and the worker thread (b).**

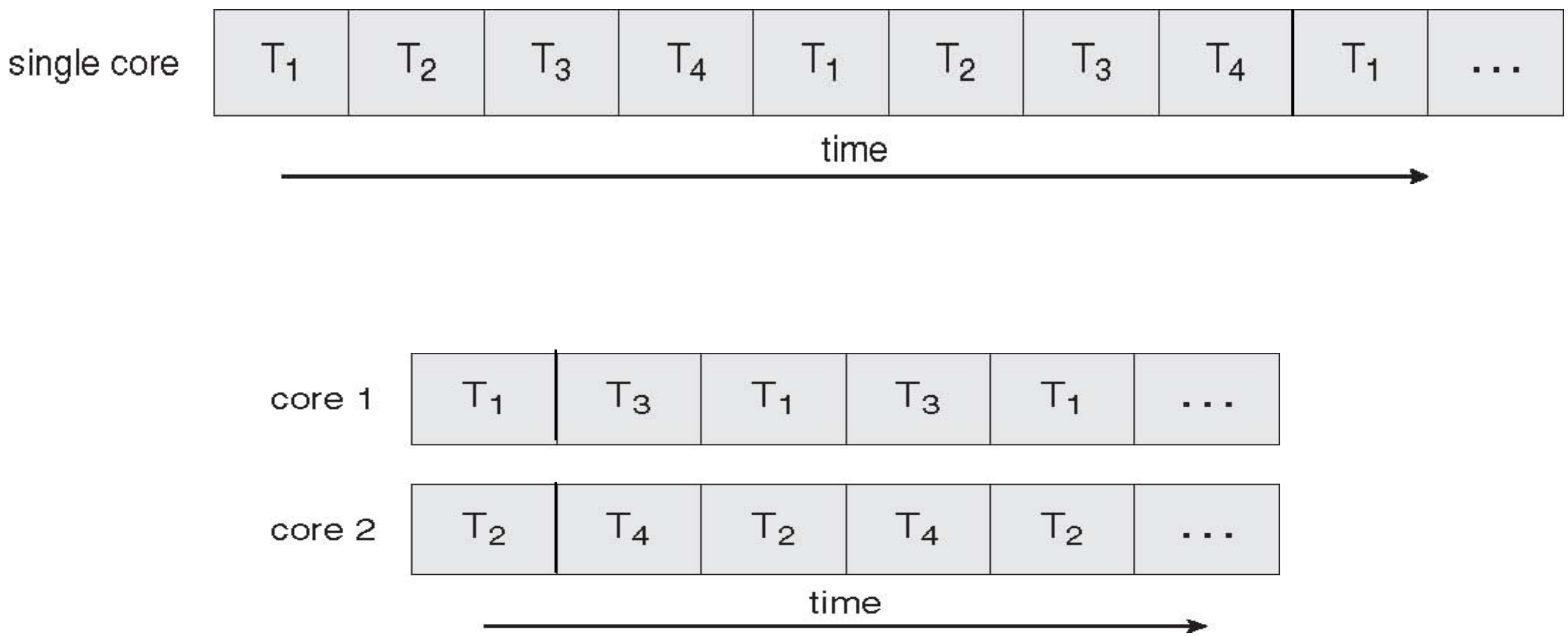
```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while(TRUE) {
  wait_for_work(&buf);
  check_cache(&buf; &page);
  if_not_in_cache(&page)
      read_page_from_disk(&buf,  &page);
  return_page(&page);
}
```

(b)

Threads in Multicore Platforms



- **Concurrent and parallel execution of threads**

Threads in Multicore Platforms (Cont.)

- **Challenge:** modify old programs and design new programs that are multithreaded

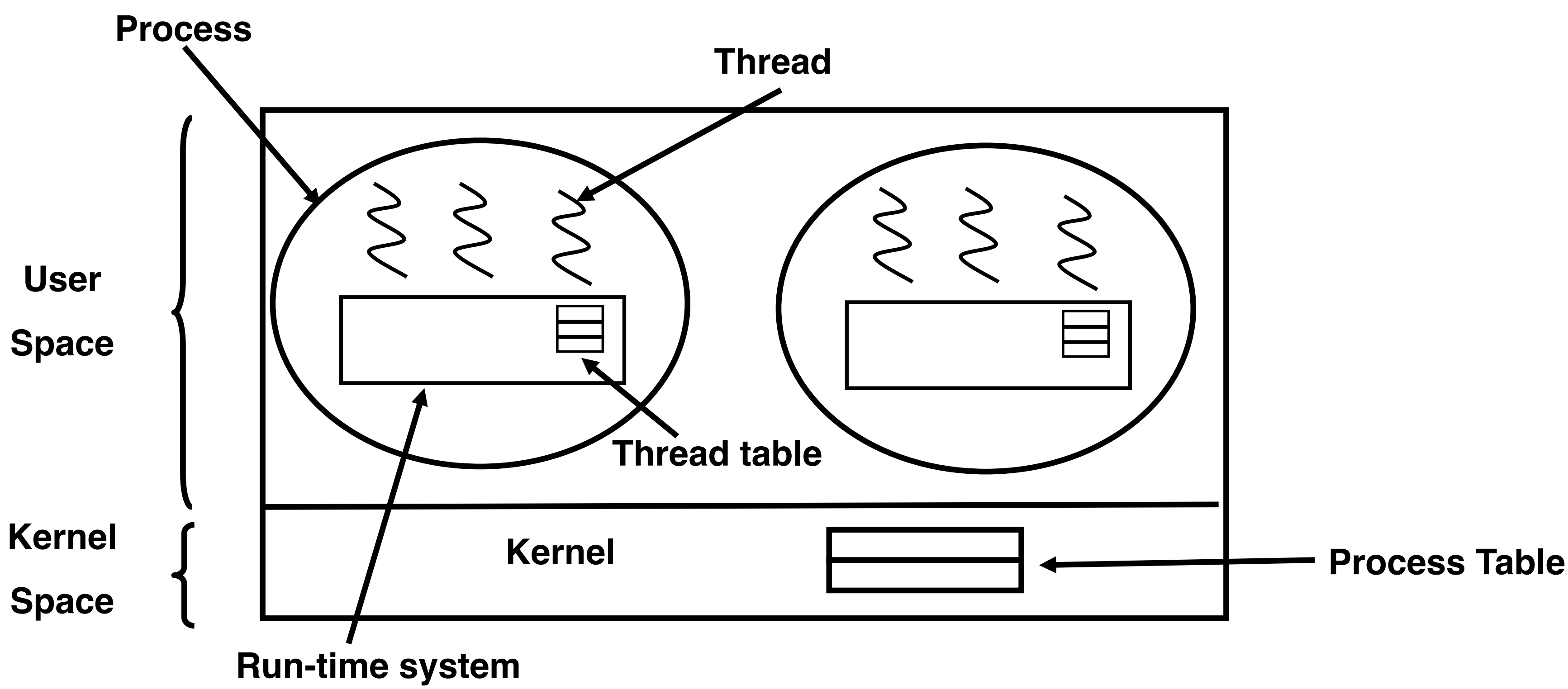
- **Issues:**
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency !!!
    - synchronization !!
  - Testing and debugging

Implementing Threads

- **Processes usually start with a single thread**
- **Usually, library procedures are invoked to manage threads**
  - ***Thread_create:* typically specifies the name of the procedure for the new thread to run**
  - ***Thread_exit***
  - ***Thread_join:* blocks the calling thread until another (specific) thread has exited**
  - ***Thread_yield:* voluntarily gives up the CPU to let another thread run**
- **Threads may be implemented in the *user space* or in the *kernel space***

User-level Threads

- User threads are supported above the kernel and are implemented by a thread library at the user level.
- The library (or run-time system) provides support for thread *creation*, *scheduling* and *management* with no support from the kernel.



User-level Threads (Cont.)

- **When threads are managed in user space, each process needs its own private *thread table* to keep track of the threads in that process.**
- **The thread-table keeps track only of the per-thread items (program counter, stack pointer, register, state..)**
- **When a thread does something that *may* cause it to become blocked *locally* (e.g. wait for another thread), it calls a run-time system procedure.**
- **If the thread must be put into blocked state, the procedure performs *thread switching.***

User-level Threads: Advantages

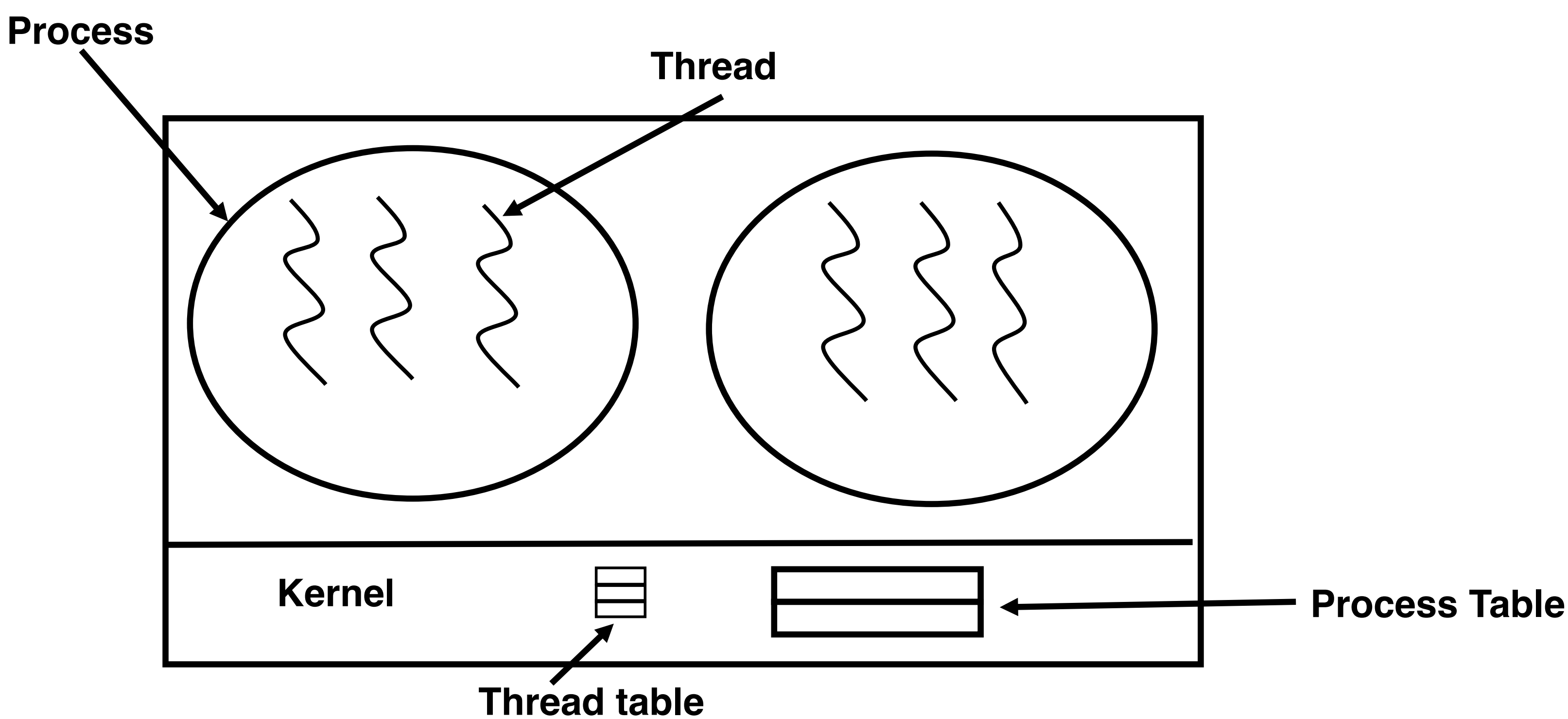
- The operating system does not need to support multi-threading.
- Since the kernel is not involved, thread switching may be very fast.
- Each process may have its own customized thread scheduling algorithm.
- Thread scheduler may be implemented in the user space very efficiently.

User-level Threads: Problems

- The implementation of *blocking system calls* is highly problematic (e.g. read from the keyboard). *All* the threads in the process risk being blocked!
- Possible Solutions:
  - Change all system calls to non-blocking
  - Sometimes it may be possible to tell in advance if a call will block (e.g. *select* system call in some versions of Unix) → "jacket code" around system calls
- How to deal with page faults?

Kernel-level threads

- **Kernel threads are supported directly by the OS: The kernel performs thread creation, scheduling and management in the kernel space**

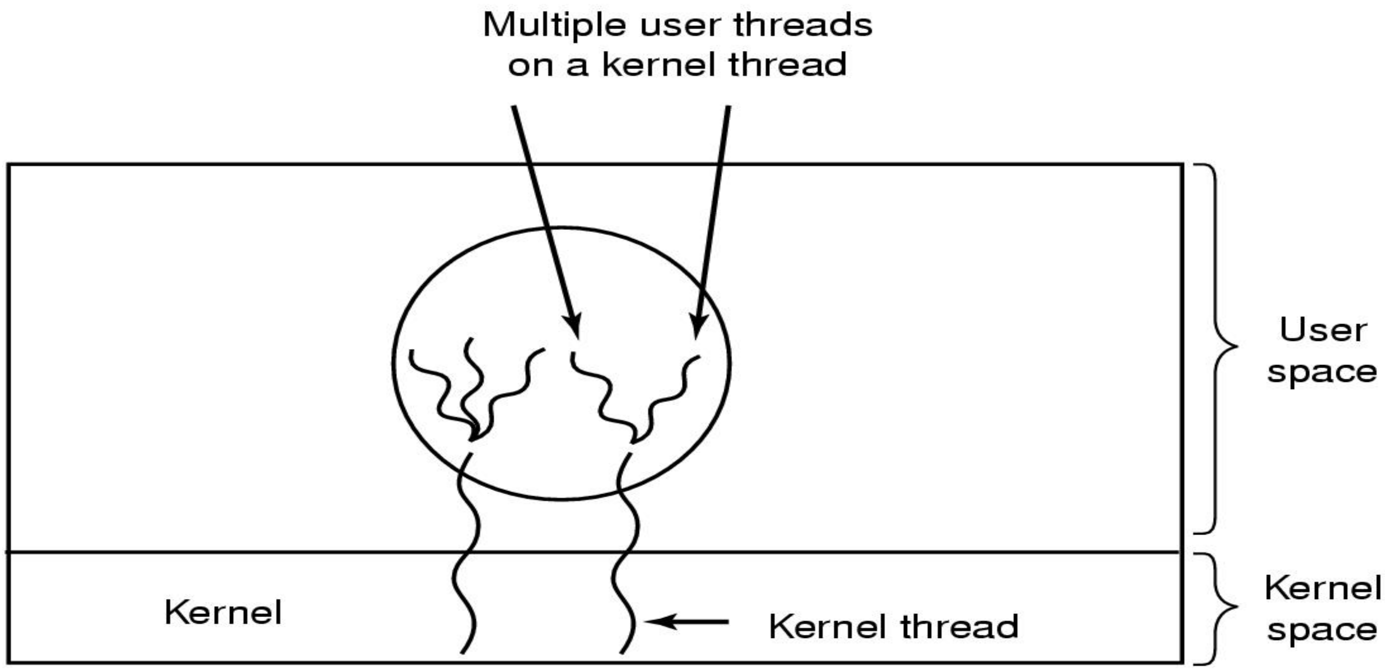


Kernel-level threads

- **The kernel has a thread table that keeps track of all threads in the system.**
- **All calls that *might* block a thread are implemented as system calls (greater cost).**
- **When a thread blocks, the kernel may choose another thread from the same process, or a thread from a different process.**
- **Some kernels *recycle* their threads, new threads use the data-structures of already completed threads.**

Hybrid Implementations

- An alternative solution is to use kernel-level threads, and then multiplex user-level threads onto some or all of the kernel threads.
- A kernel-level thread has some set of user-level threads that take turns using it.



Pthreads

- **A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.**
- **API specifies behavior of the thread library, implementation is up to development of the library.**
- **Common in UNIX operating systems**
- **Pthread programs use various statements to manage threads: *pthread_create, pthread_join, pthread_exit, pthread_attr_init,…***

Thread Calls in POSIX

Thread Call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Windows XP Threads

- Windows XP supports kernel-level threads
- The primary data structures of a thread are:
  - ETHREAD (executive thread block)
    - Thread start address
    - Pointer to parent process
    - Pointer to the corresponding KTHREAD
  - KTHREAD (kernel thread block)
    - Scheduling and synchronization information
    - Kernel stack (used when the thread is running in kernel mode)
    - Pointer to TEB
  - TEB (thread environment block)
    - Thread identifier
    - User-mode stack
    - Thread-local storage

Linux Threads

- **In addition to *fork()* system call, Linux provides the *clone()* system call, which may be used to create threads**

- **Linux uses the term *task* (rather than process or thread) when referring to a flow of control**

- **A set of flags, passed as arguments to the *clone ()* system call determine how much sharing is involved (e.g. open files, memory space, etc.)**

Observing Threads

- **top –H**
- **ps –eLf**
- **pstree**