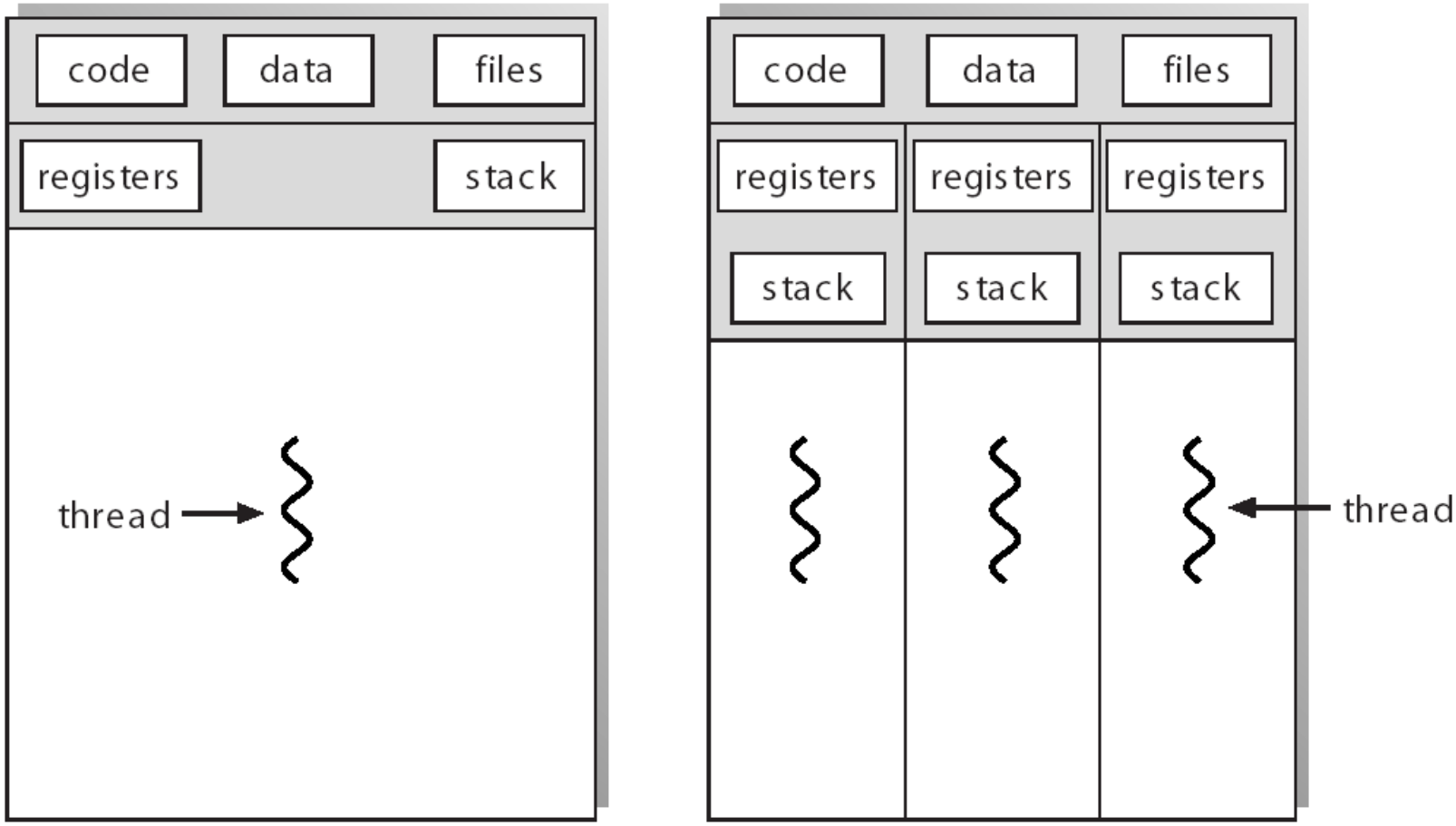
Threads

Chapter 4

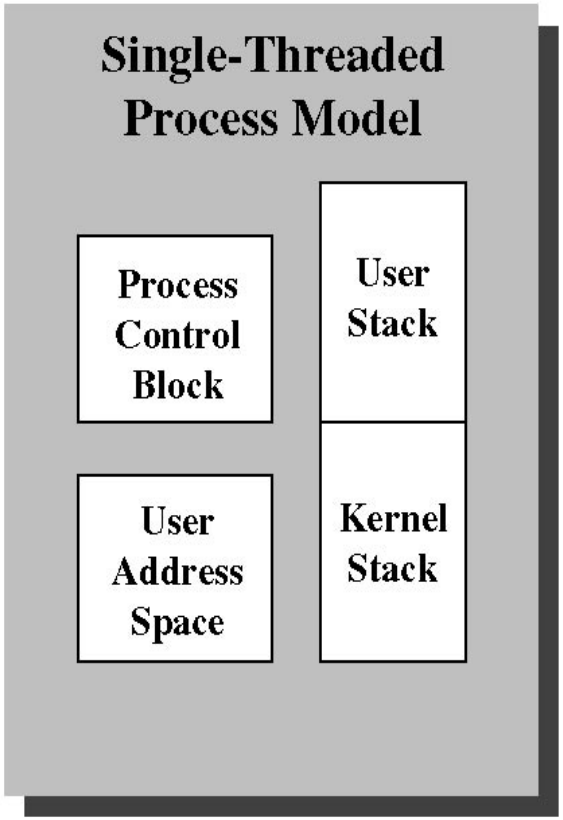
Single and Multithreaded Processes

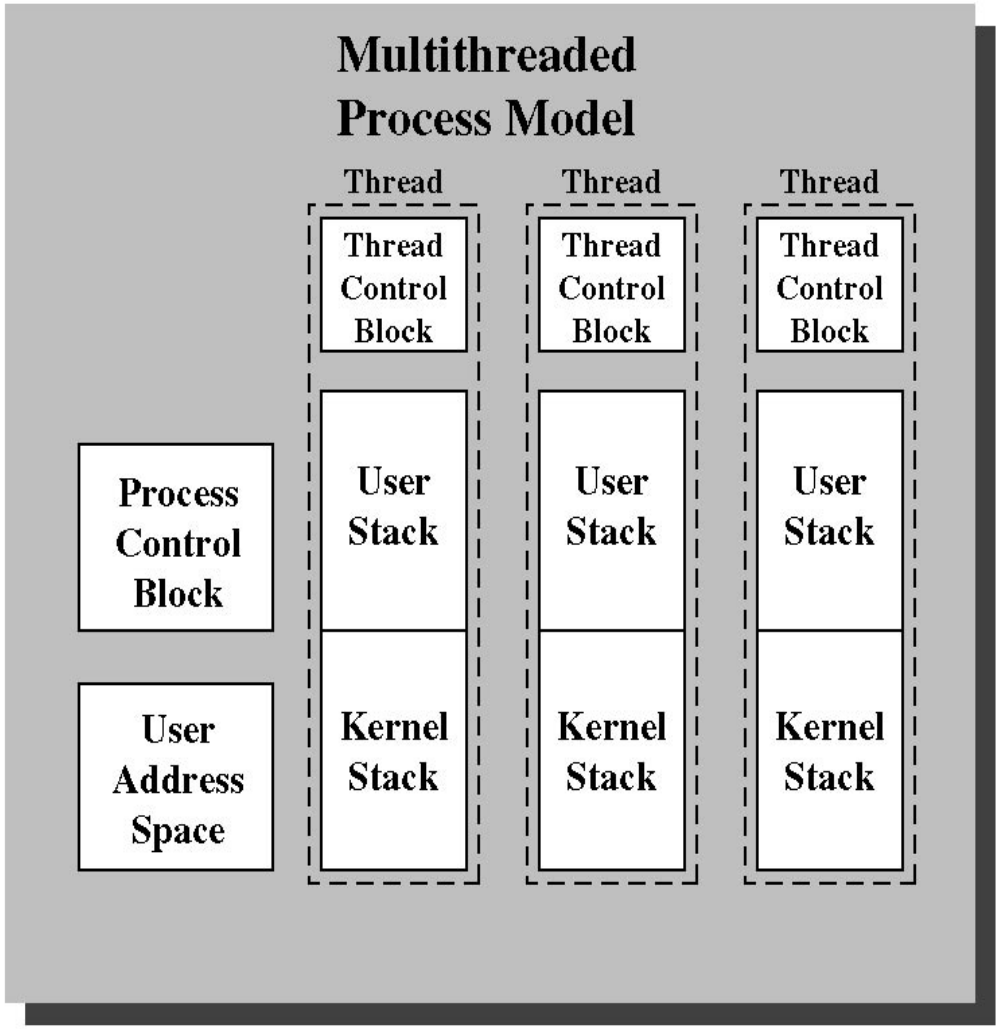


single-threaded process

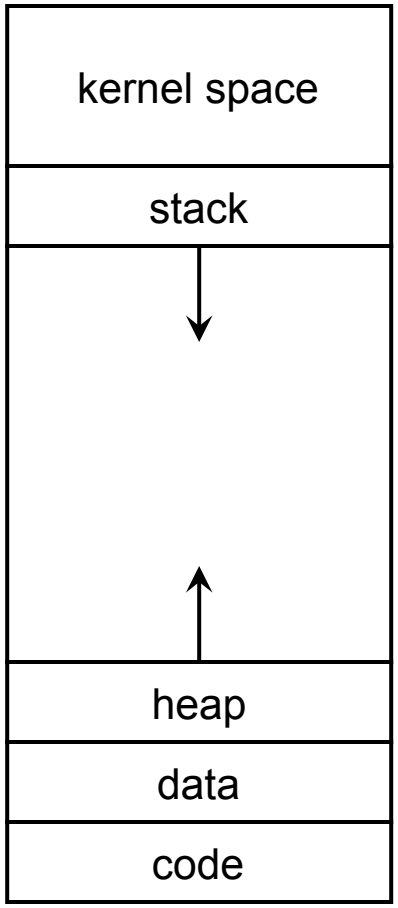
multithreaded process

Single and Multithreaded Processes

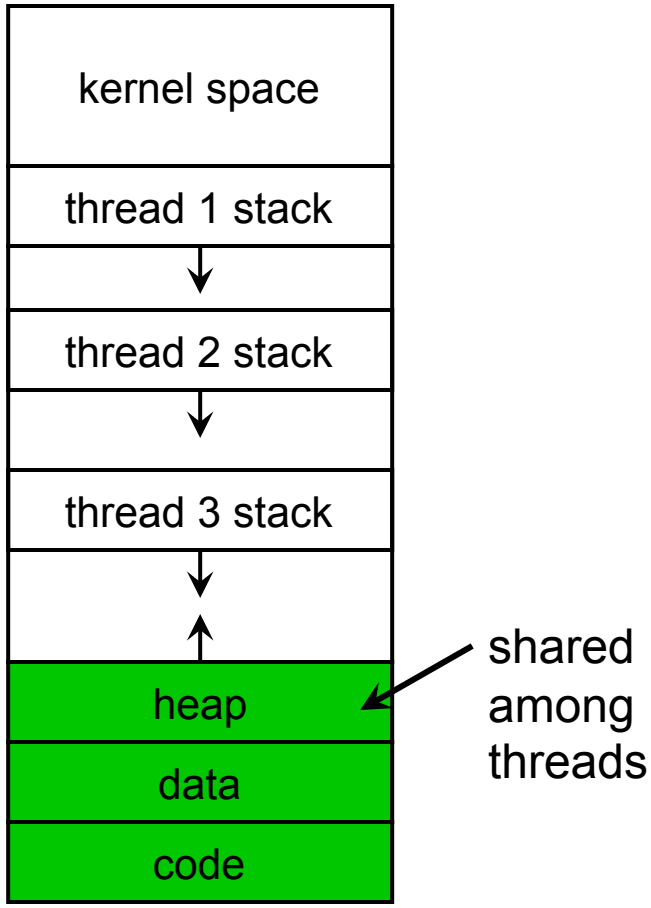




Process Address Space



single threaded address space



multi-threaded address space

Thread

- In single threaded systems, a ***process*** is:
  - Resource owner
    - memory address space, files, I/O resources
  - Scheduling/execution unit
    - execution state/context, dispatch unit

- Multithreaded systems
  - Separation of resource ownership & execution unit
  - A ***thread*** is unit of execution, scheduling and dispatching
  - A ***process*** is a container of resources, and a collection of threads

Thread

- All threads of a process share resources
  - Memory address space: global data, code, heap …
  - Open files, network sockets, other I/O resources
  - User-id
  - IPC facilities

- Private state of each thread:
  - Execution state: running, ready, blocked, etc..
  - Execution context: Program Counter, Stack Pointer, other user-level registers
  - Per-thread stack

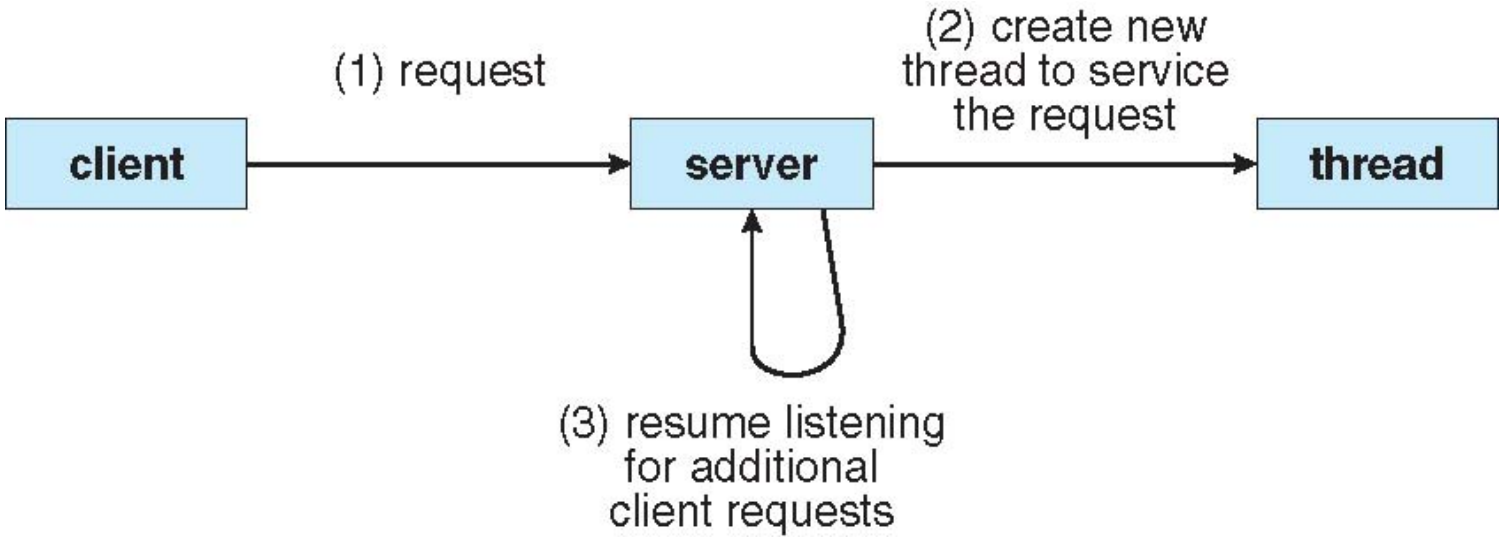
Threads vs. Processes

- Why multiple threads? Why not multiple processes?
  - Yes, all roads lead to Rome
  - But no, all roads are not equal

Threads vs. Processes

- Advantages of multi-threading over multi-processes
  - Far less time to create/terminate thread than process
    - An order of magnitude improvement
    - Less states to create/copy/inherit, or to release
  - Context switch is quicker between threads of the same process
  - Communication btw. threads of the same process is more efficient
    - Through shared memory
  - Makes complex program control structure simpler
    - Blocking I/O vs. non-blocking I/O

Multithreaded Server Architecture



Threads vs. Processes: An Example

```
main()
{
  sock = socket( … );
  listen( … );
  while (1) {
    new_sock = accept (sock);

    if ( fork()== 0 ){
      http_service(new_sock);
      exit(0);
    }
  }
}
http_service(int s){
  /*serve HTTP request & exit*/
  …
}
```

```
main()
{
  sock = socket( … );
  listen( … );
  while (1) {
    new_sock = accept (sock);

    pthread_create(&tid, NULL,
        http_service, new_sock);
  }
}

http_service(int s){
  /*serve HTTP request & exit*/
  …
}
```

When to, and not to use threads?

- Applications
  - Multiprocessor machines
  - Handle slow devices
  - Background operations
  - Windowing systems
  - Server applications to handle multiple requests
- No threads cases
  - When each unit of execution require different authentication/user-id
  - E.g., secure shell server

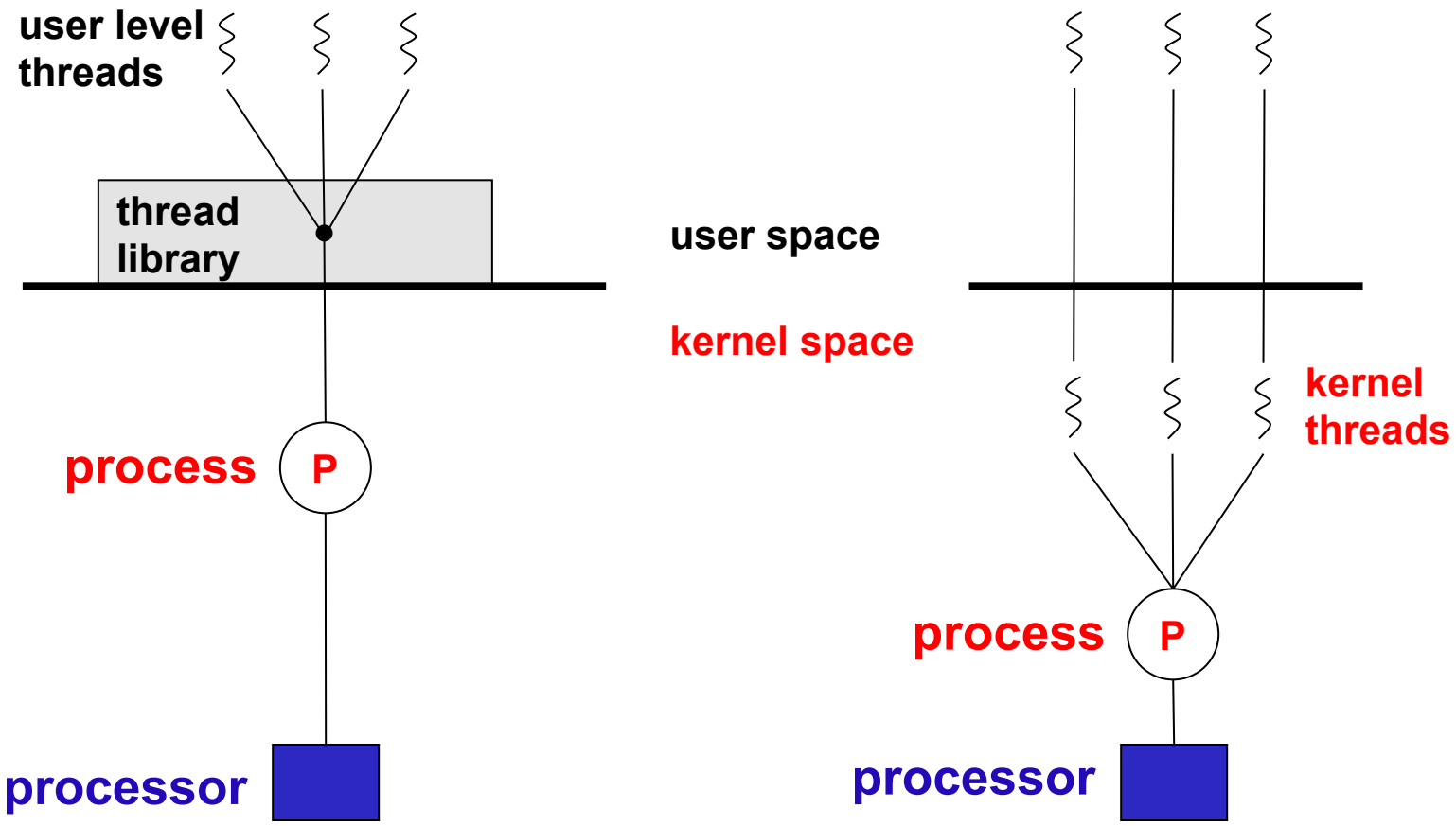
Thread context switch

- A context switch takes place when
  - Async. events (e.g., time quota expires, hardware interrupt)
  - Sync. call
    - Thread performs a blocking system call
    - Thread voluntarily yields, e.g., lock unavailable
- Save state of currently executing thread
  - Copy all user registers to thread control block
  - Save control information: PC, SP
- Restore state of thread to run next
  - Copy values of user/control registers from thread control block to processor registers

Thread Implementation: User vs. Kernel level

- User-level threads
  - Thread management by user space thread library
  - Kernel not aware of thread activities
  - Thread library performs like an operating system
    - Thread creation & termination
    - Thread scheduling and context switches
    - Maintains control and context information
      - State, priority
      - User registers, stack, stack pointers
  - Can be preemptive or non-preemptive
- Kernel-level threads (lightweight processes)
  - Kernel supports multiple execution contexts (kernel threads)
  - Thread management done by the kernel

User level vs. Kernel level Threads



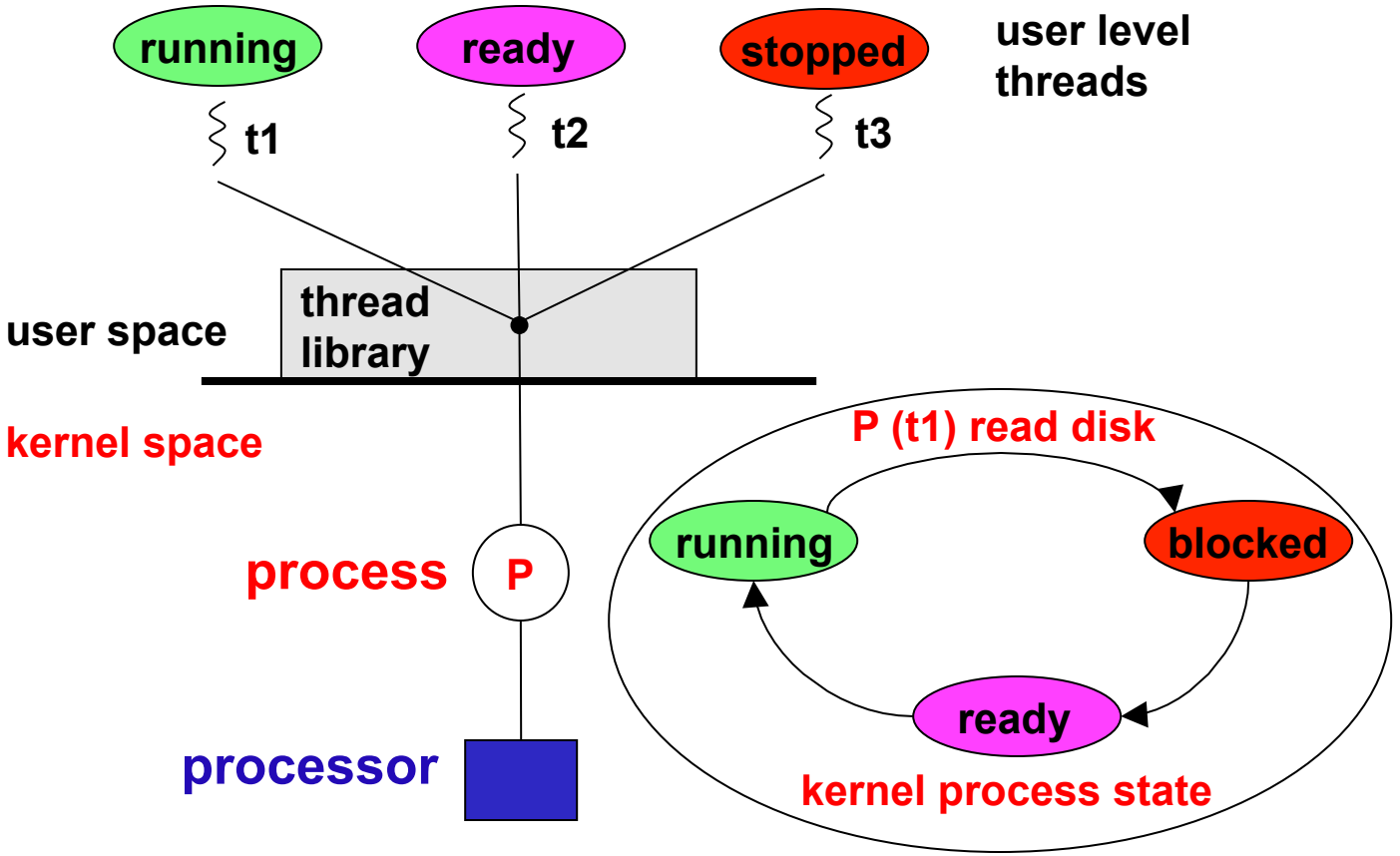
User Level Threads

- Advantages of user-level threads
  - Performance: low-cost thread operations and context switching
    - No kernel involvement, saves user/kernel mode switches
    - VAX results: 7 μs (procedure call) vs. 17 μs (kernel trap)
  - Flexibility: scheduling can be application-specific w/o kernel change
  - Portability: user-level thread library easy to port

- Disadvantages of user-level threads
  - If a user-level thread is blocked in the kernel, the entire process (all threads of that process) are blocked
  - Cannot take advantage of multiprocessing (the kernel assigns one process to only one processor)

User Level Thread: Blocking Problem

- Thread t1 issues blocking disk read
- Process p suspended by kernel
- Although thread t2 is ready to run, kernel not aware of user threads, and all threads are blocked

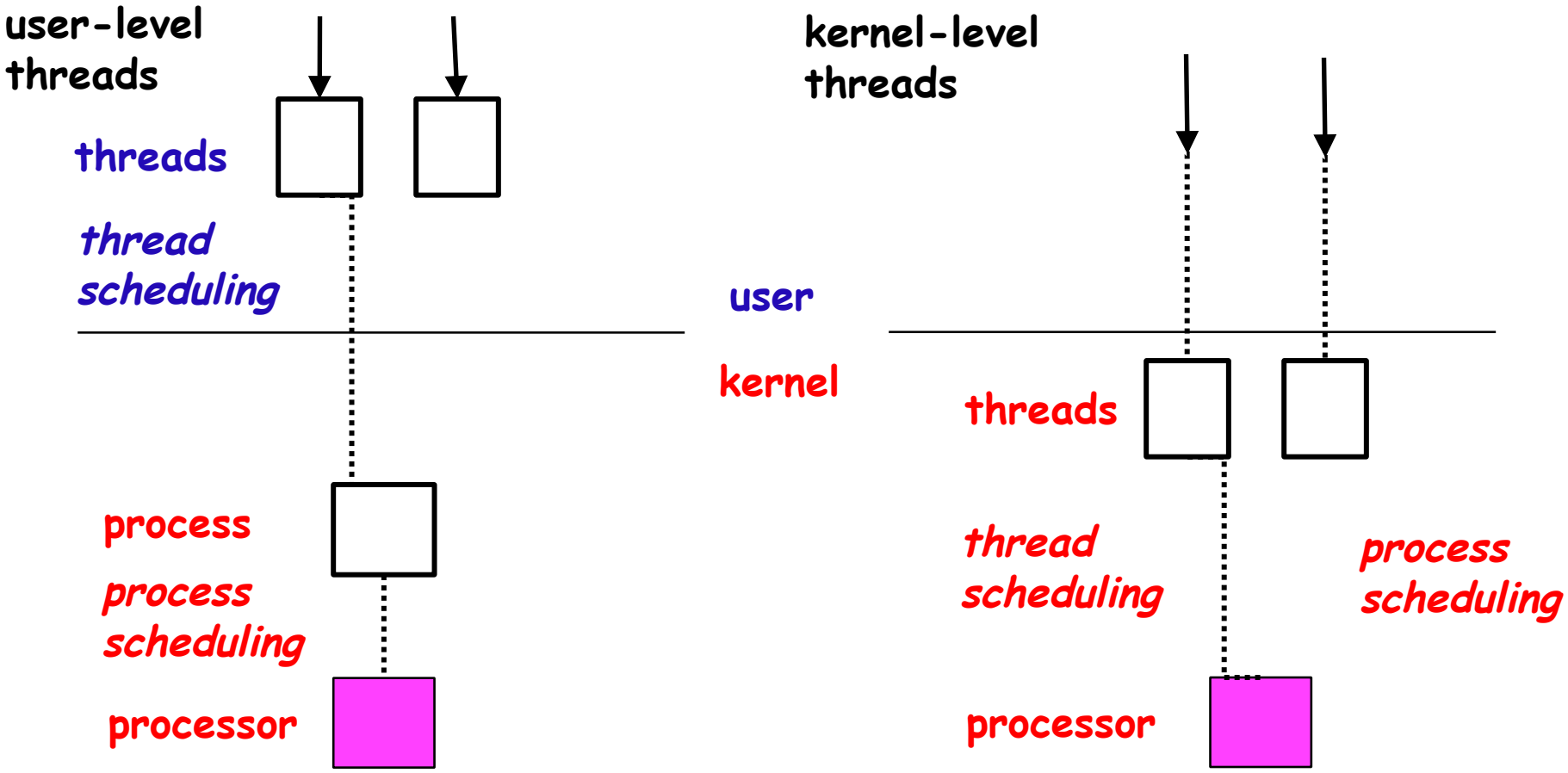


Kernel Level Threads

- Advantages of kernel-level threads
  - Blocking of one thread does not block other threads in a process
  - Threads within one process can be assigned to different processors simultaneously, taking advantage of SMP
- Disadvantages of kernel-level threads
  - Context switch btw. threads of one process requires two user/kernel mode switches
  - Inflexible in scheduling: entirely relies on a generic kernel scheduler
  - Portability: OS has to provide support

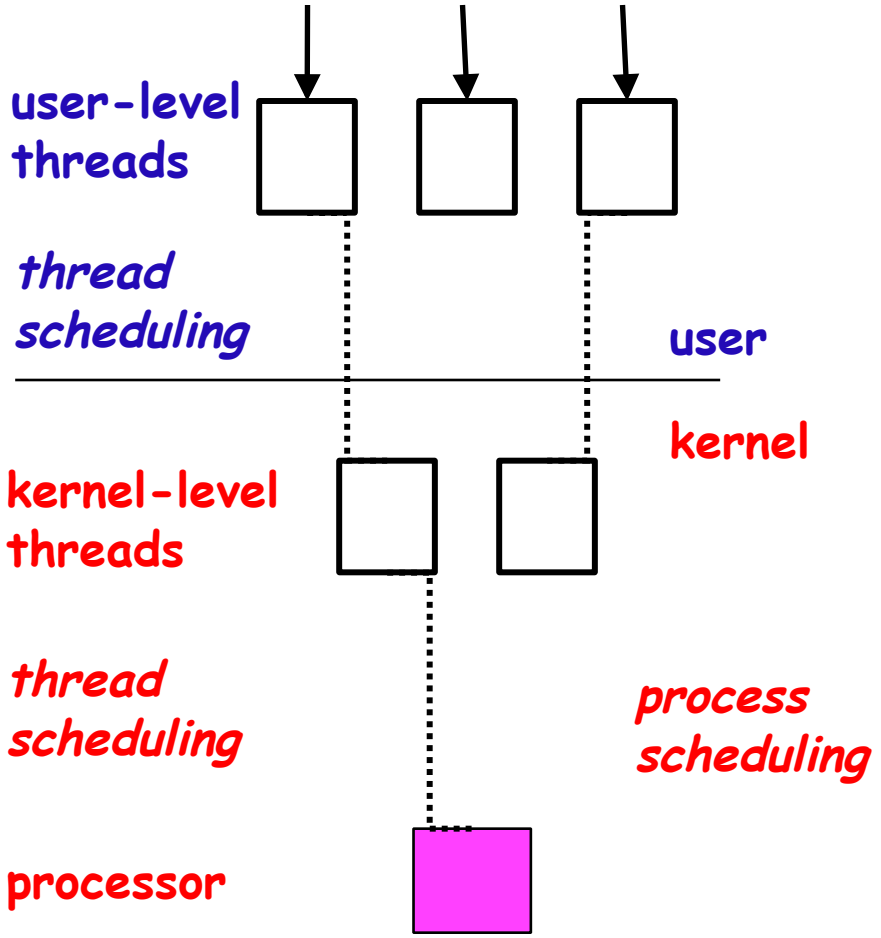
Operation	User thread (µs)	Kernel thread (µs)	Processes (µs)
Null fork	34	948	11,300
Signal-wait	37	441	1,840

User-Level vs. Kernel-Level Threads



User-Level vs. Kernel-Level Threads

- No reason why we shouldn't have both (e.g. Solaris)
- Most systems now support kernel threads
- User-level threads are available as linkable libraries

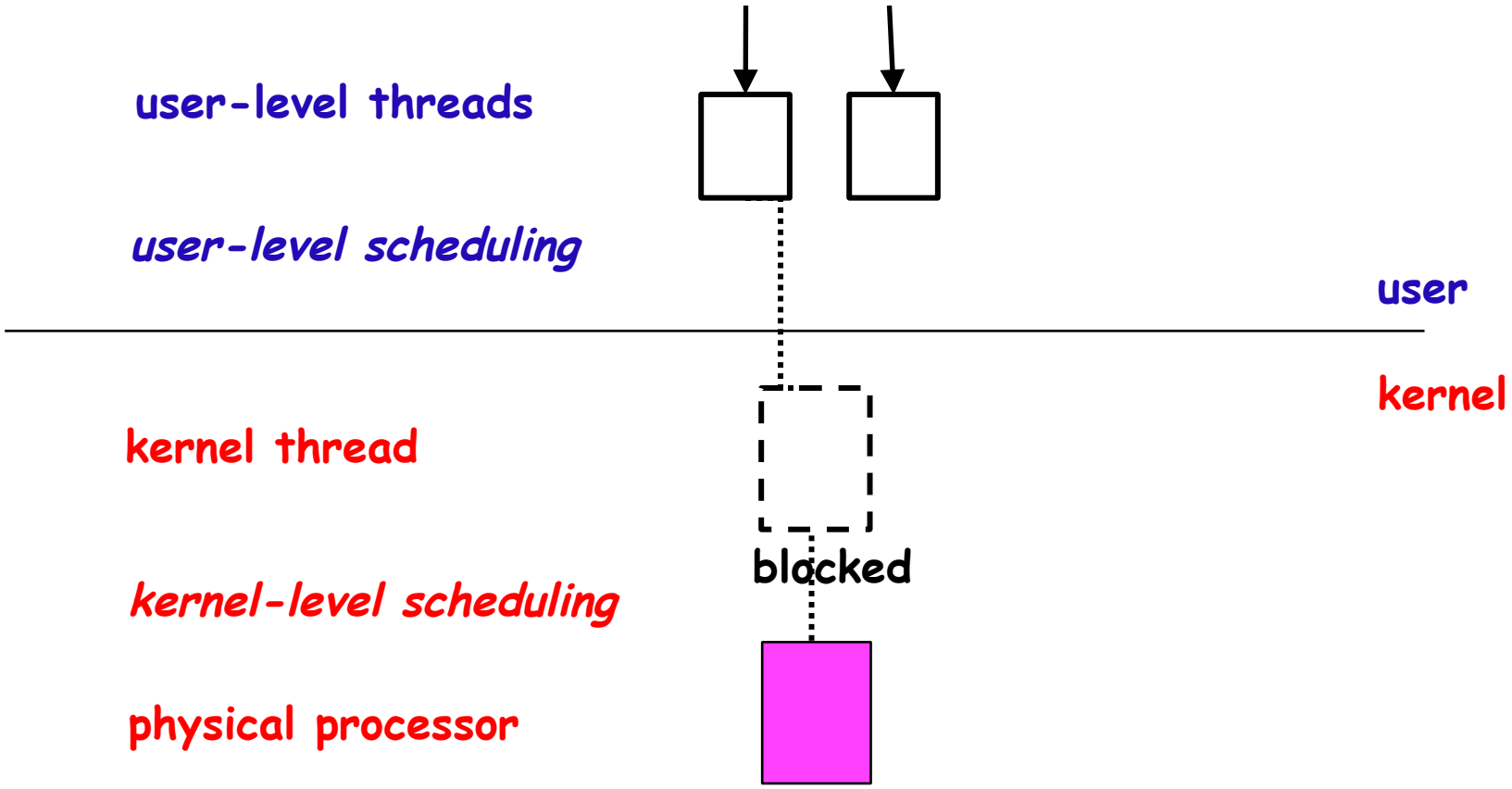


Kernel Support for User-Level Threads

- Even kernel threads are not quite the right abstraction for supporting user-level threads

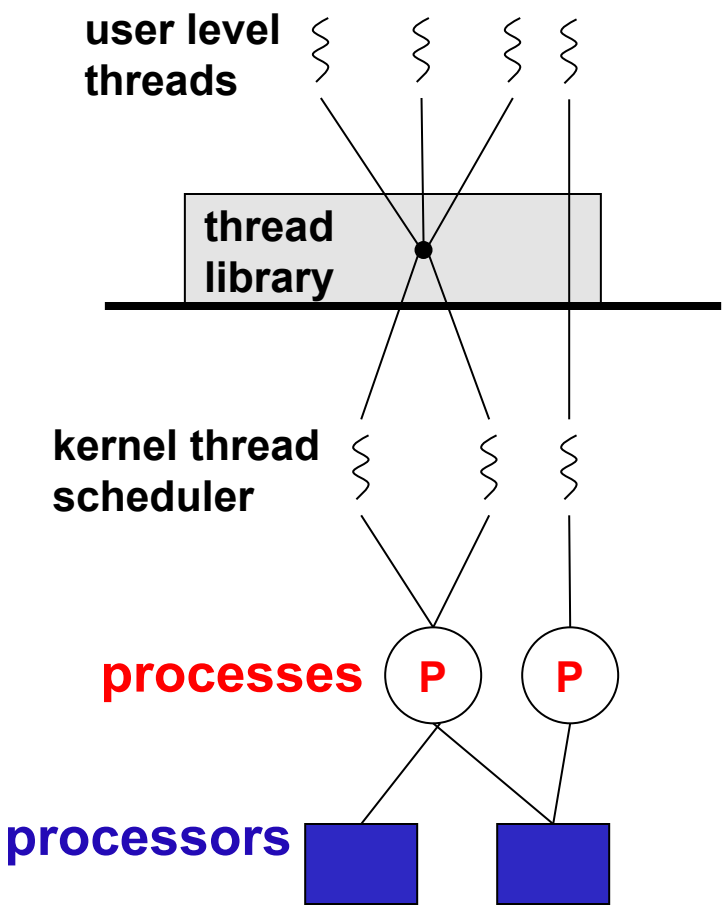
- Mismatch between where the scheduling information is available (user) and where scheduling on real processors is performed (kernel)
  - When the kernel thread is blocked, the corresponding physical processor is lost to all user-level threads although there may be some ready to run.
  - Scheduler Activation

Why Kernel Threads Are Not The Right Abstraction



Combining the user & kernel level threads

- Hybrid approach (e.g. Solaris)
- User level thread library
  - Thread creation/termination
  - Most of scheduling and synchronization
- User threads mapped to a set of kernel threads
  - Programmer can adjust how the mapping is done
  - 1 user thread – 1 kernel thread
  - m user thread – 1 kernel thread
  - m user thread – n kernel thread



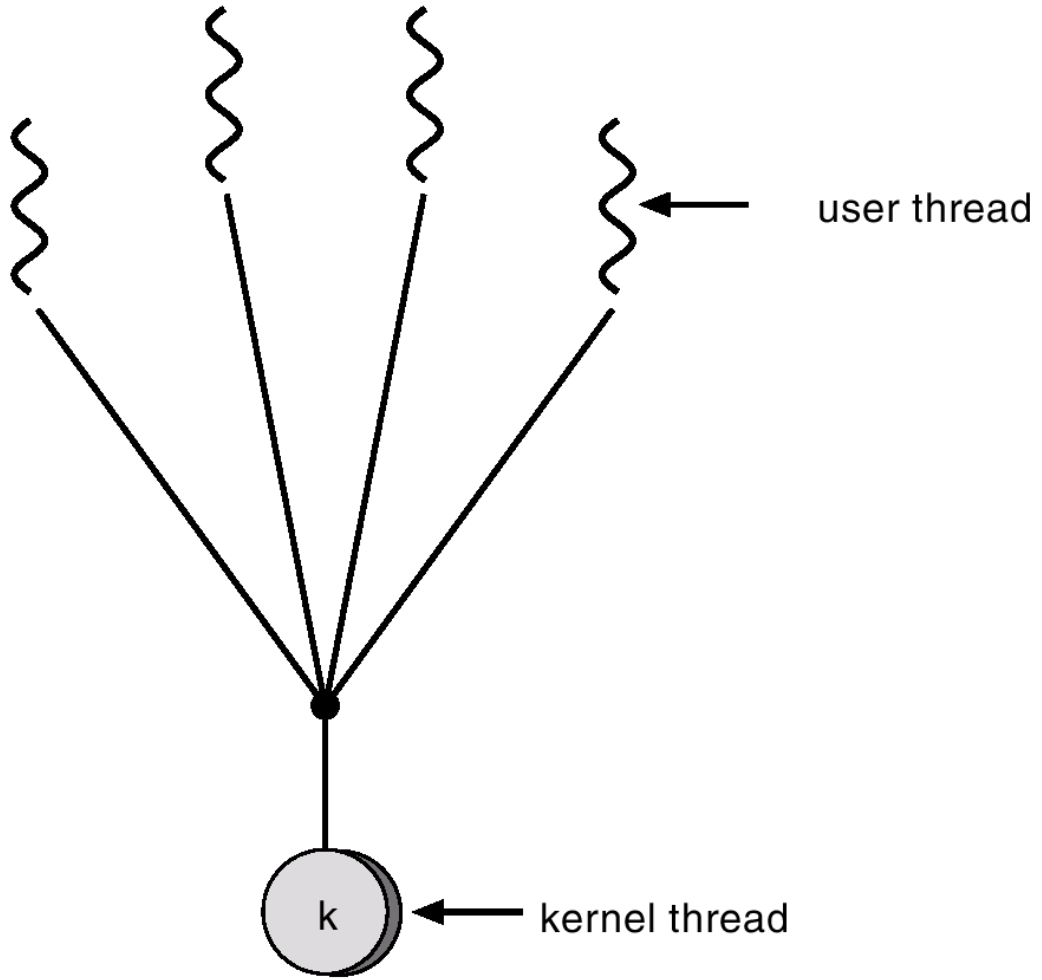
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many
- Two-Level

Many-to-One

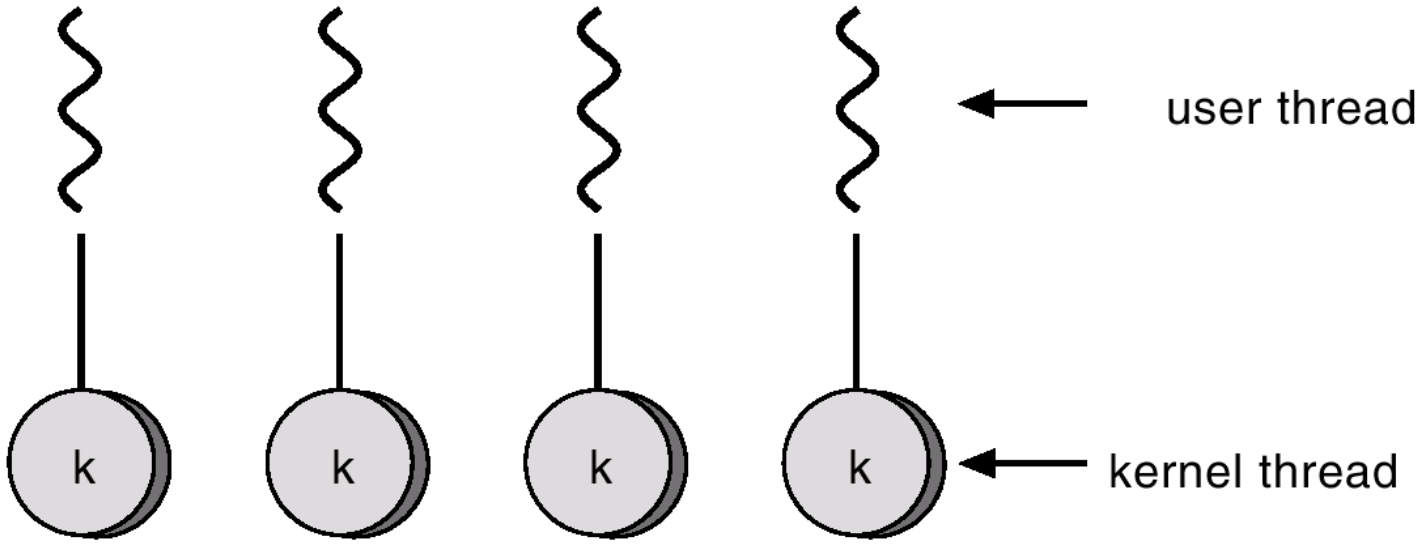
- Many user-level threads mapped to single kernel thread

- Examples
  - Solaris Green Threads
  - GNU Portable Threads



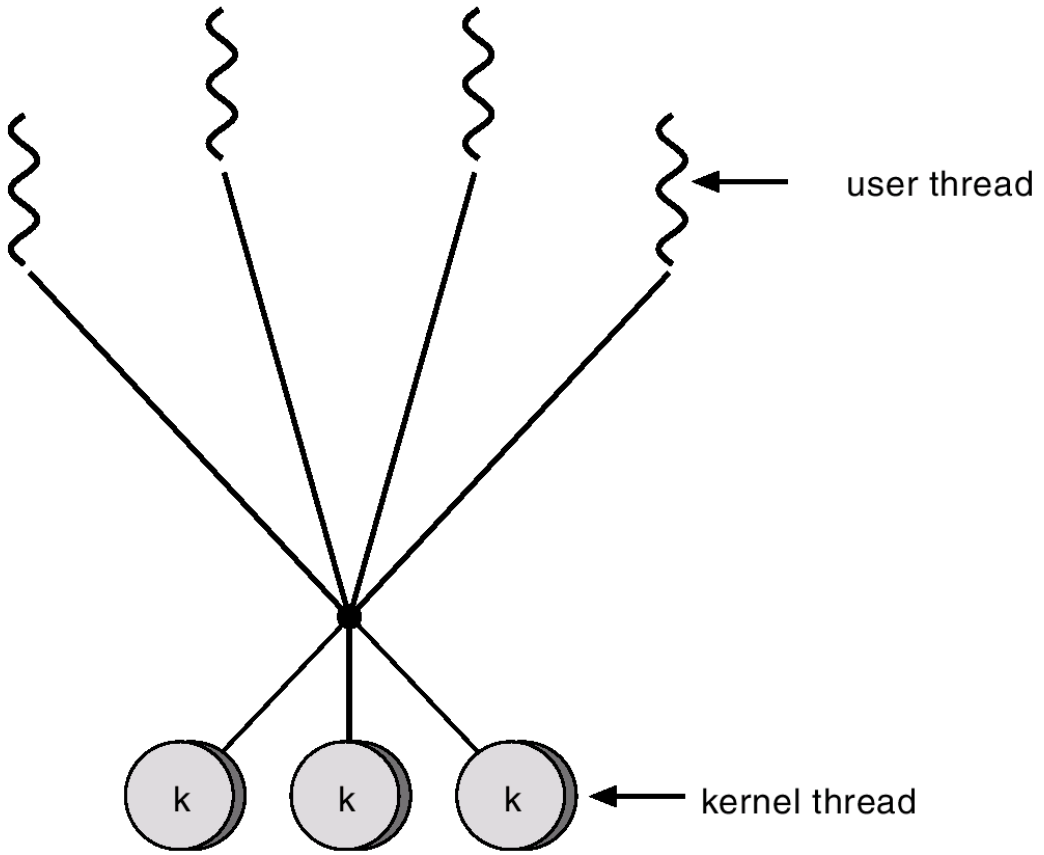
One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



Many-to-Many Model

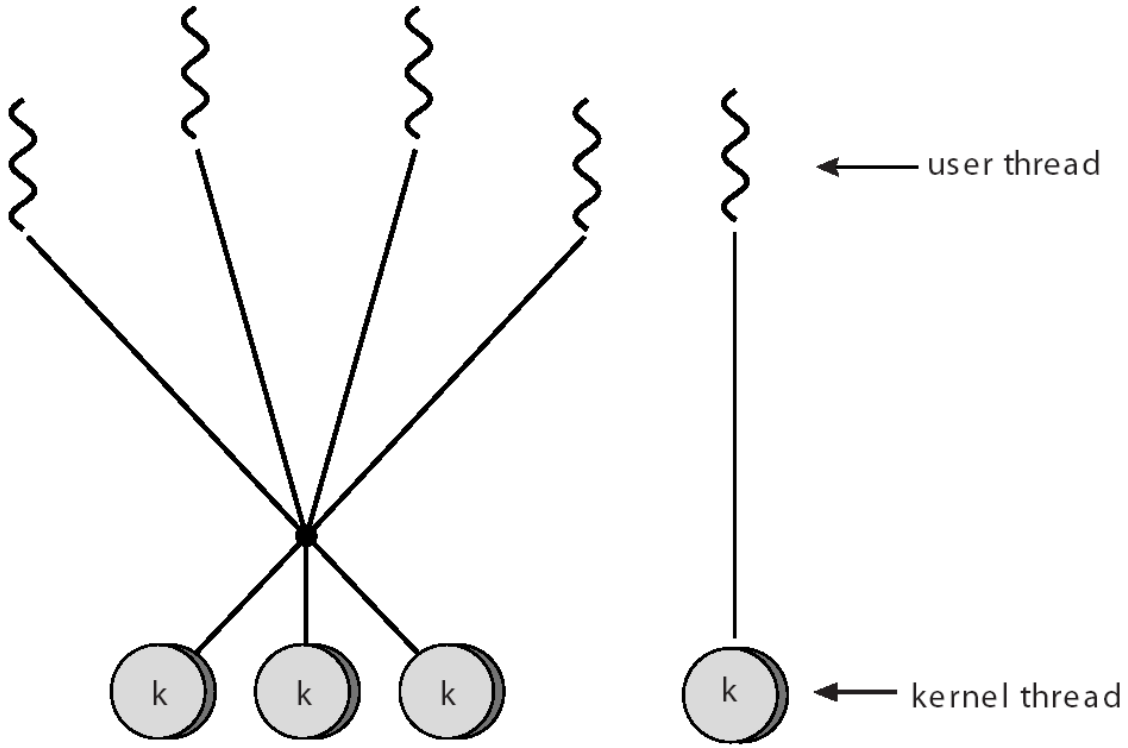
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



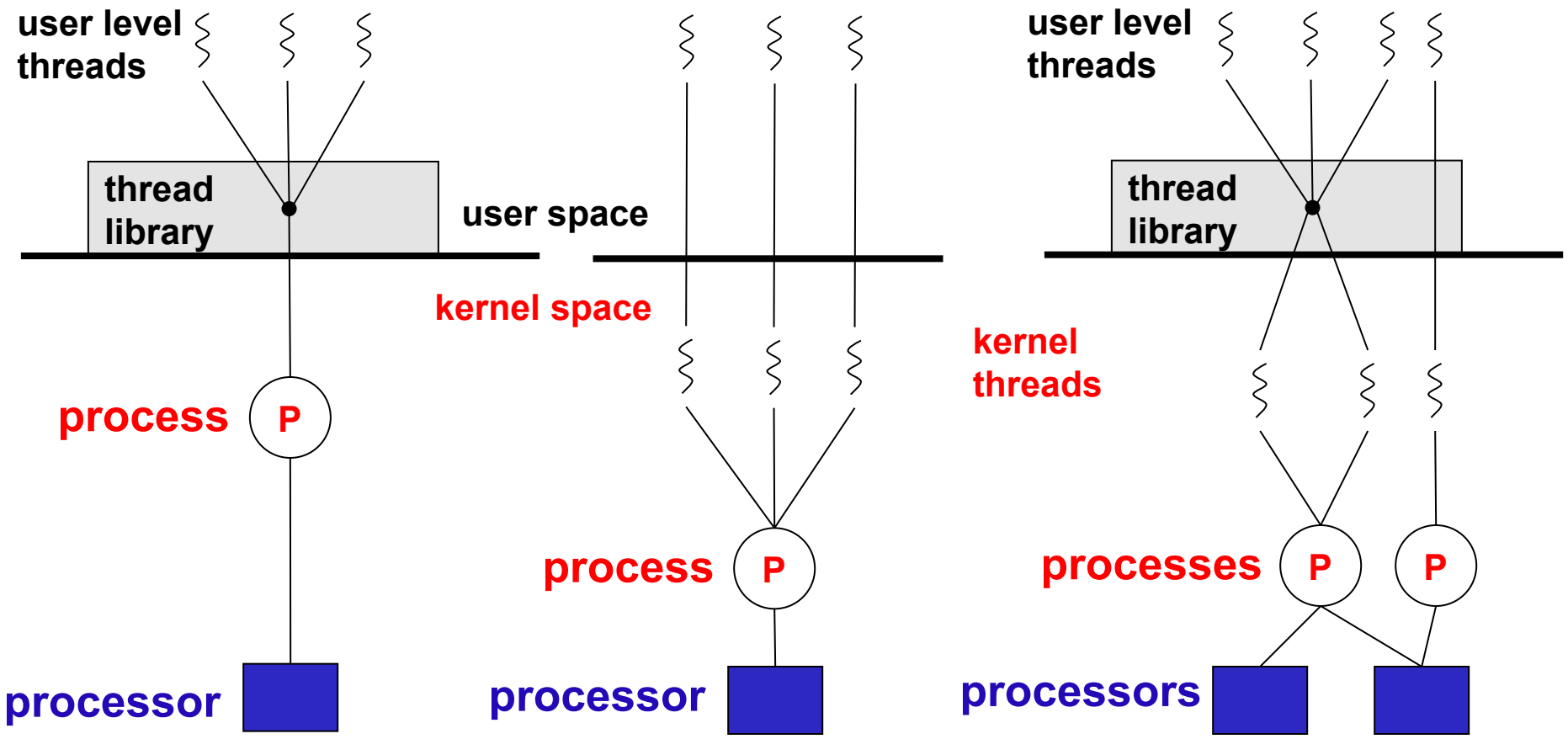
Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



User level, Kernel level, and Hybrid Threads



Signal Handling in Multithreaded Process

- Signal is process-level concept
  - Handler associated with the process
  - When a thread register a signal handler, it applies to all threads in the same process
  - Signal is only delivered to a single thread, to avoid multiple delivery
  - A thread, however, can ignore/mask certain signals
- Signal delivery
  - What if all threads are blocked when event occurs?
  - OS writes into PCB: that a signal should be delivered to the process
  - Next time when a thread that does not mask the signal is dispatched, the signal is delivered
- Common strategy:
  - One thread responsible to field all signals
    - calls sigwait() in an infinite loop
  - All other threads mask all signals

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled