Threads

Fall 2016

Based on Lecture Slides by Andrew Tanenbaum

Reading

- Chapter 2, Sec 2.2, “Modern Operating Systems, Fourth Ed.”, Andrew S. Tanenbaum
  - You can skip 2.2.7, 2.2.8, 2.2.9

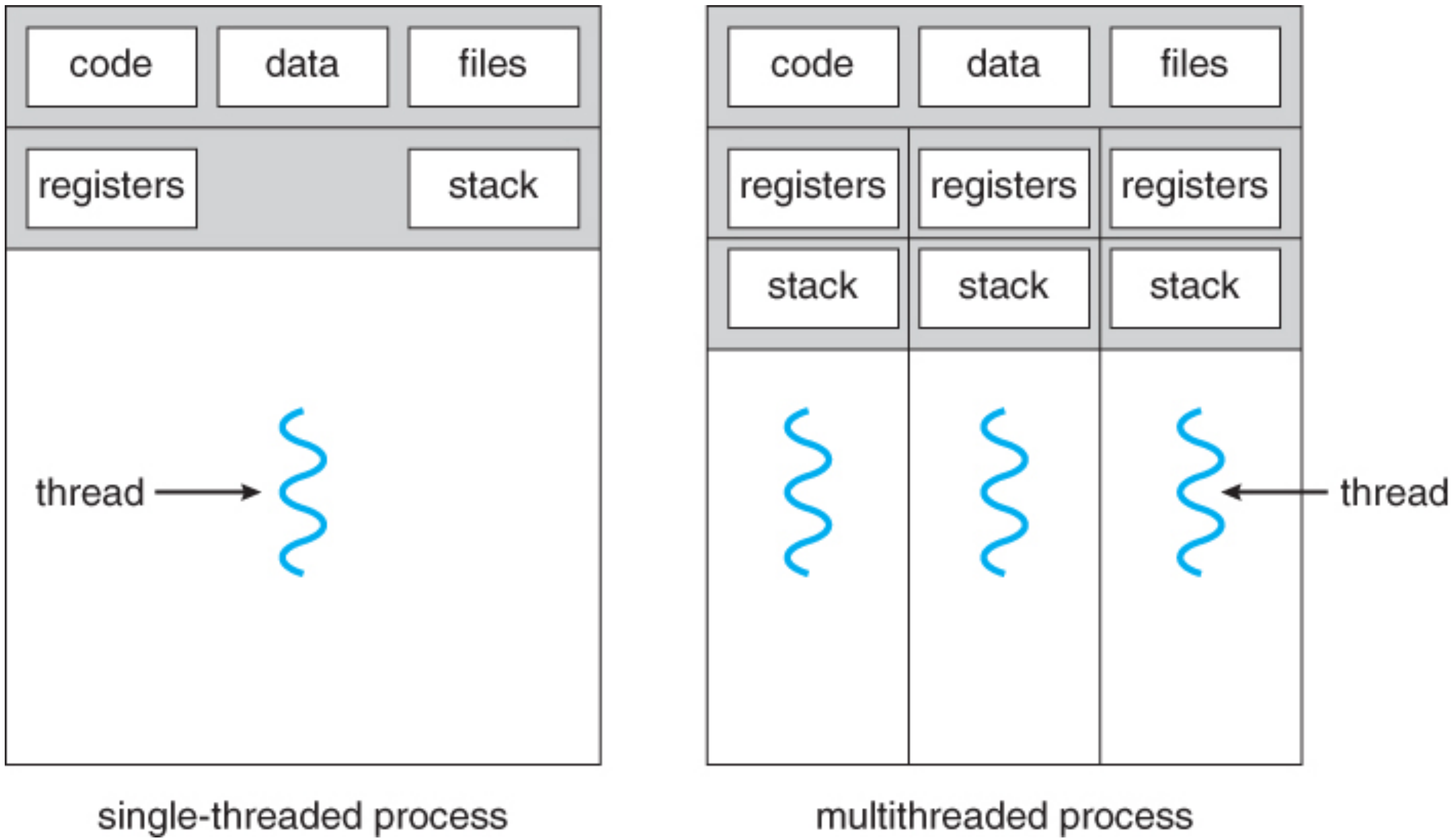
Outline

- **Why Threads?**
- The Thread Model
- Threads in User Space
- Threads in the Kernel
- Hybrid Threads
- Threads in Unix/Linux

Processes and Threads

- Process
  - Single address space
  - Single execution path
- Threads within a process
  - Share same address space
  - Multiple execution paths

Single-Threaded and Multithreaded Processes



Why Threads?

- In many applications, multiple activities are going on at once
  - Some activities may block from time to time
  - They need to share the address space
  - Decomposing these applications into multiple threads (execution paths) makes the programming model simpler
- Lighter weight than a process
  - Easier (Faster) to create and destroy than a process

Why Threads? (cont’d)

- For applications performing both CPU and I/O
  - CPU and I/O activities can overlap
  - Therefore, speedup the application
- Parallelism can be achieved if there are multiple CPUs in the system

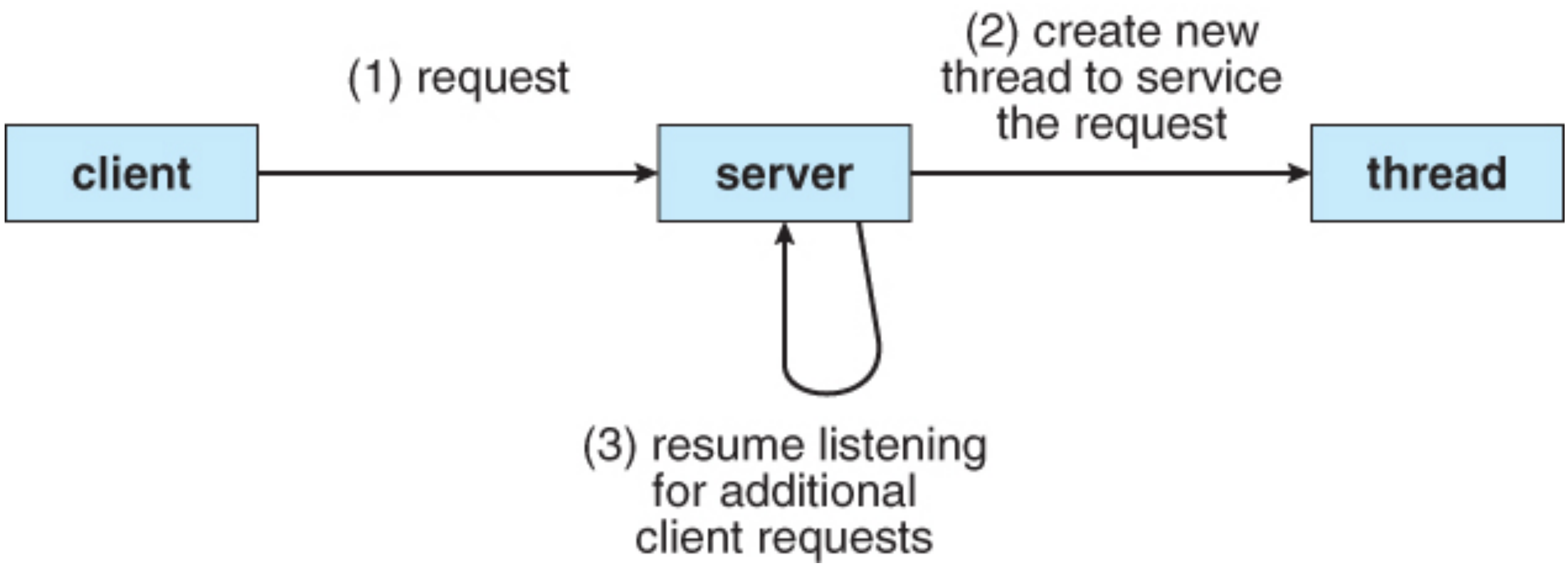
Types of Parallelism

- Data parallelism:
  - divide the data among multiple threads
  - each thread does the same task on the subset of data assigned to it
- Task parallelism:
  - divide different tasks to be performed by different threads
  - tasks are preformed simultaneously

Examples

- Word processor, Spreadsheet
  - A computation thread, an interactive thread, a backup thread
- Web server
  - A thread to process each client's request
- Applications that must process very large amounts of data
  - An input thread, a processing thread, and an output thread

Example: Multithreaded Server



Web Server Example

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

Dispatcher Thread

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

Worker Thread

Outline

- Why Threads?
- **The Thread Model**
- Threads in User Space
- Threads in the Kernel
- Hybrid Threads
- Threads in Unix/Linux

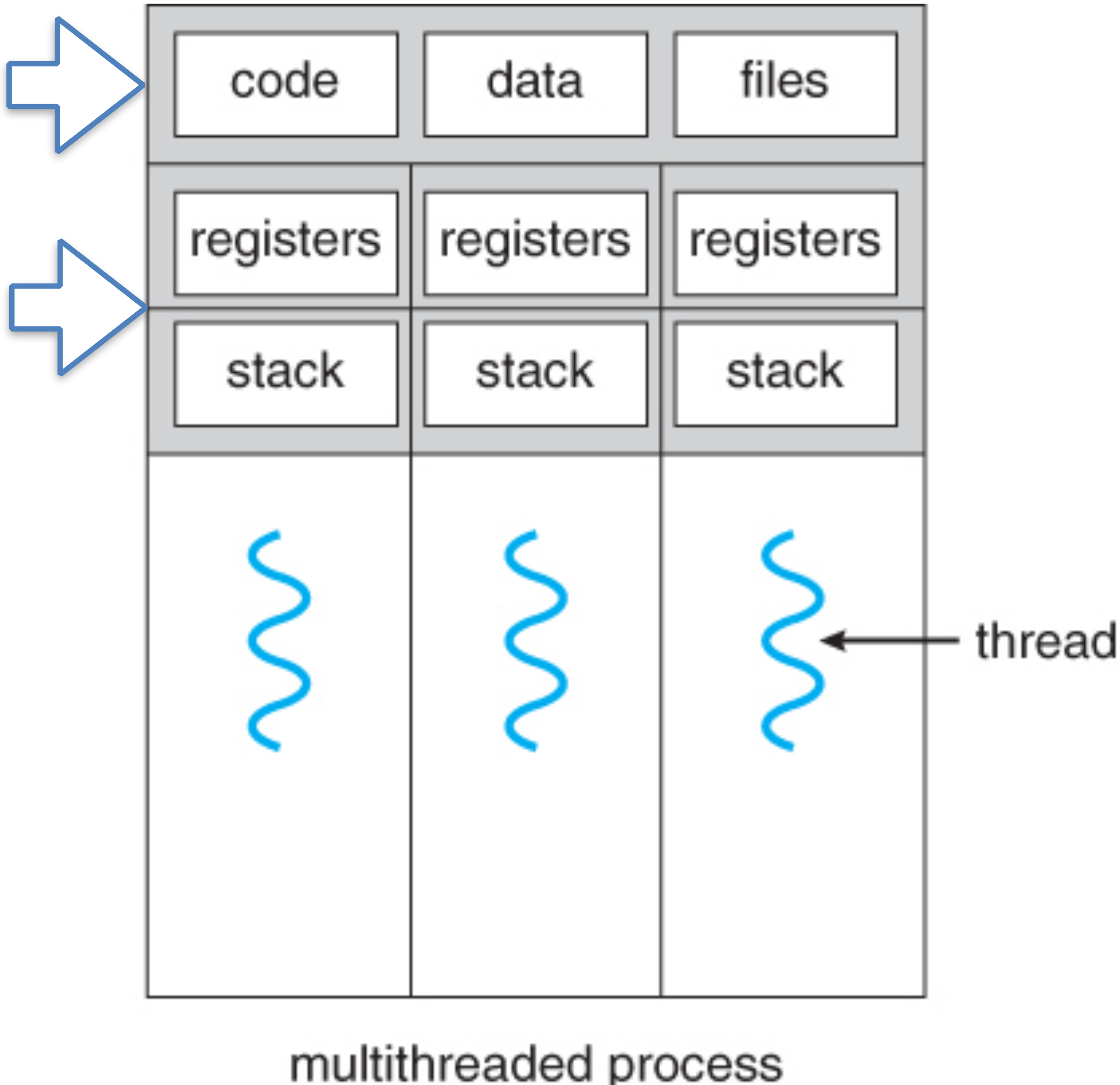
Address Space and Execution

- Shared resources by a process:
    - Address space containing code and data
    - Opened Files
    - Child processes
- Path of Execution (Thread) – multiple per process
    - Program counter to identify next instruction to execute
    - Registers containing current working variables
    - Stack

Multithreaded Processes

Shared resources by all threads in a process

Per each thread in a process



Therefore, ….

- Processes are used to group resources together
- Threads are the entities that are scheduled on the processor

Thread Model

- Threads allow multiple execution paths within the same process
- Threads share the same resources of the process that they work within
- Threads are sometimes called light weight processes
- Multithreading refers to allowing multiple threads within a process

Multithreading

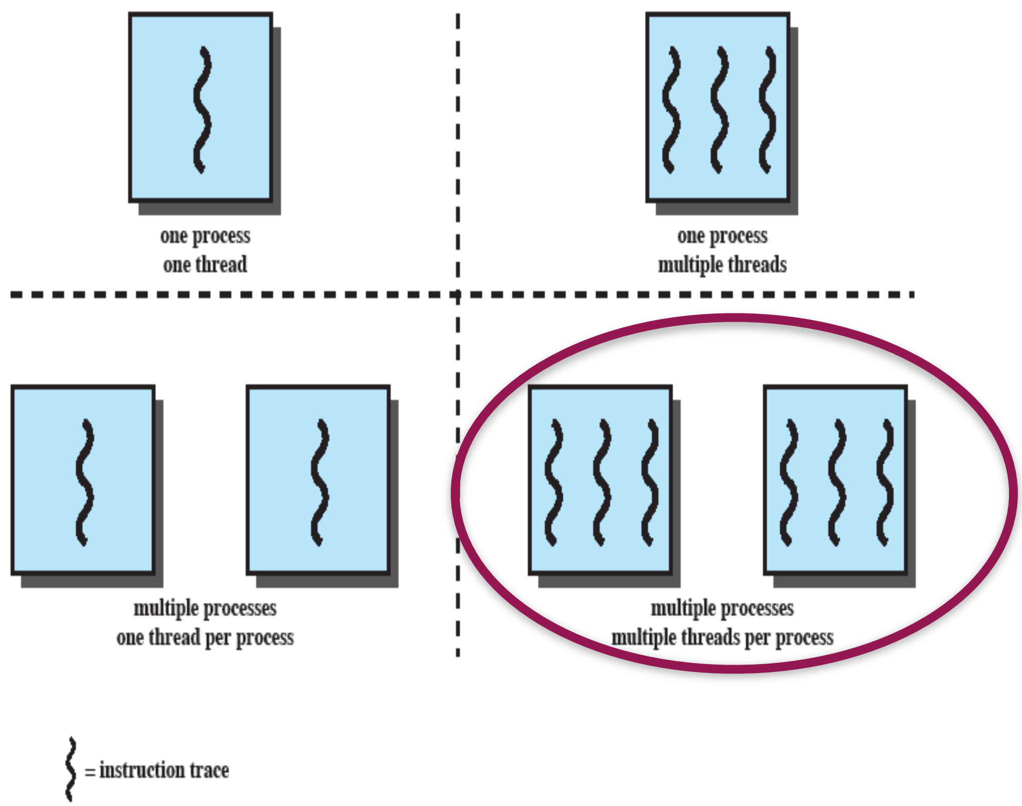
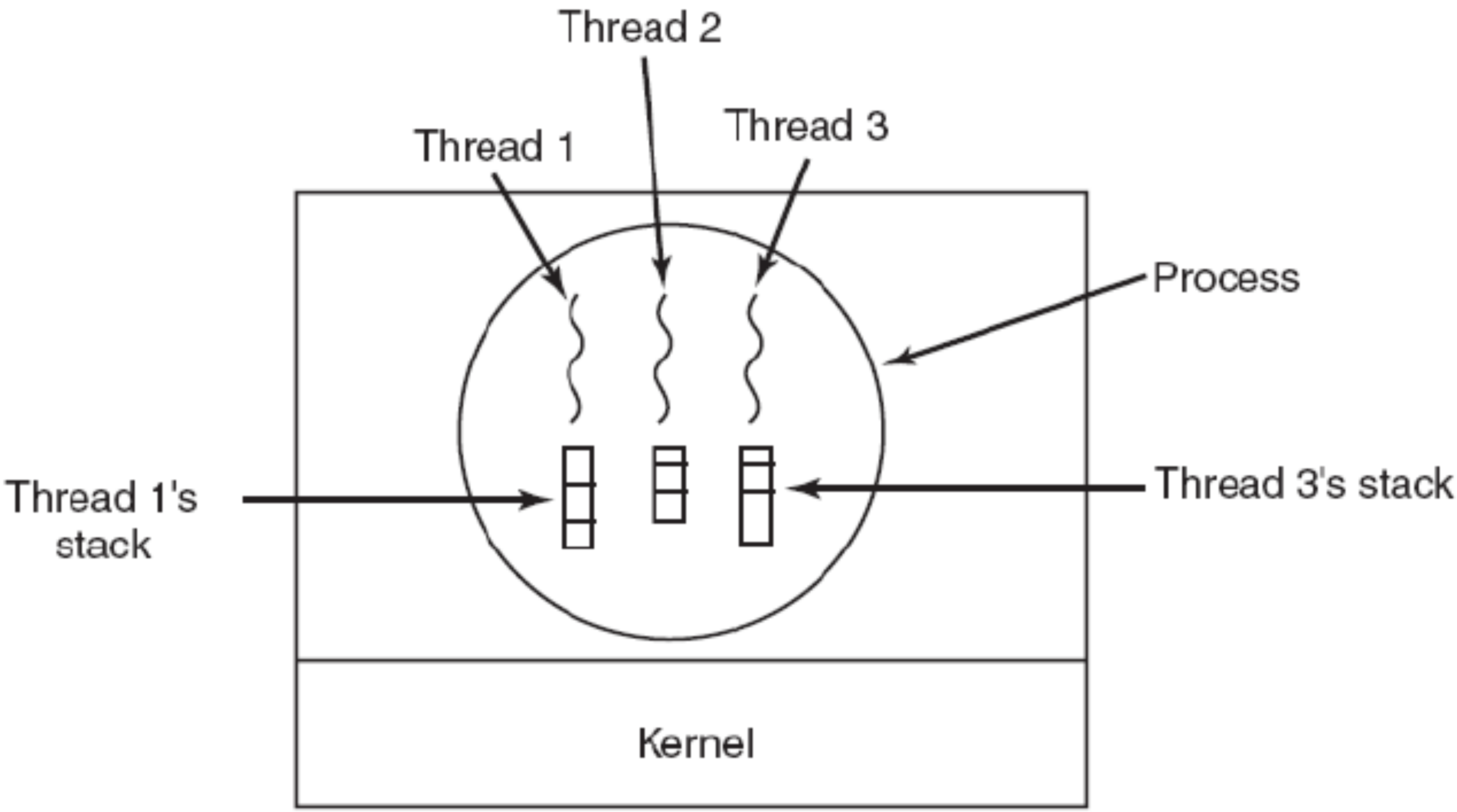


Figure 4.1 Threads and Processes [ANDE97]

Resources for Processes and Threads

Per Process Items	*Per Thread Items*
Address space *Global variables* *Open files* *Child processes* *Pending alarms* *Signals and signal handlers* *Accounting information*	*Program counter* *Registers* *Stack* *State*

Stack Per Thread



Thread Lifecycle

- A process starts with a single thread
- That thread can create multiple threads using the procedure: ***thread…create***
  - input: the procedure to run
  - no information about the address space
  - output: identifier of the created thread
- When a thread finishes, it terminates by calling the procedure: ***thread…exit***
- Other: ***thread…join*** and ***thread…yield***

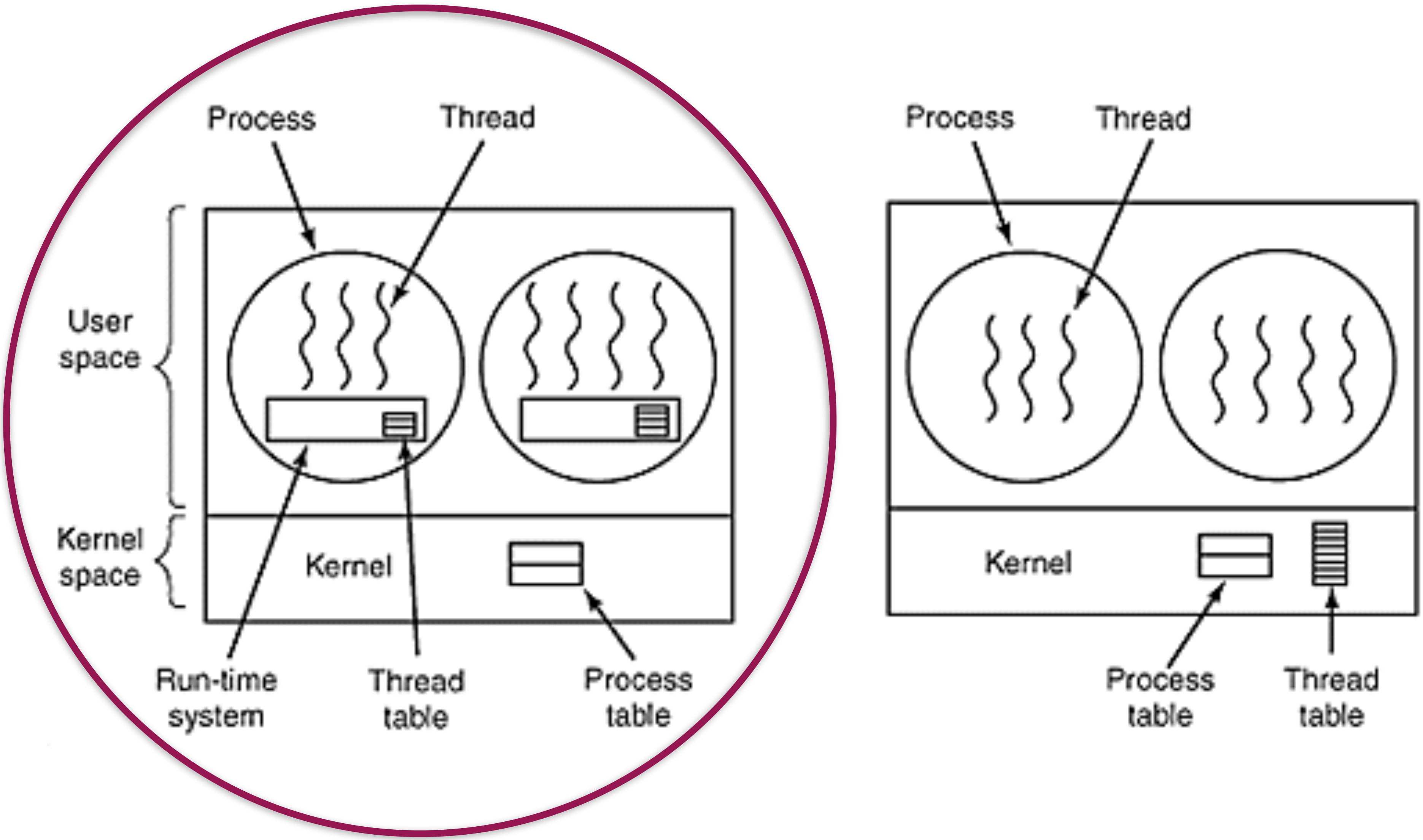
Outline

- Why Threads?
- The Thread Model
- **Threads in User Space**
- Threads in the Kernel
- Hybrid Threads
- Threads in Unix/Linux

Implementing Threads in User Space

- AKA User-Level Thread (ULT)
- Thread management is done by the application
- The kernel knows nothing about the threads
- Each process has its own thread table (similar to the process table kept by the kernel)
- A thread scheduler to schedule threads within a process
- Thread switching is at least an order of magnitude faster than process switching

User Level Threads



User Level Thread States and Process States

Process B
Th1
Th2

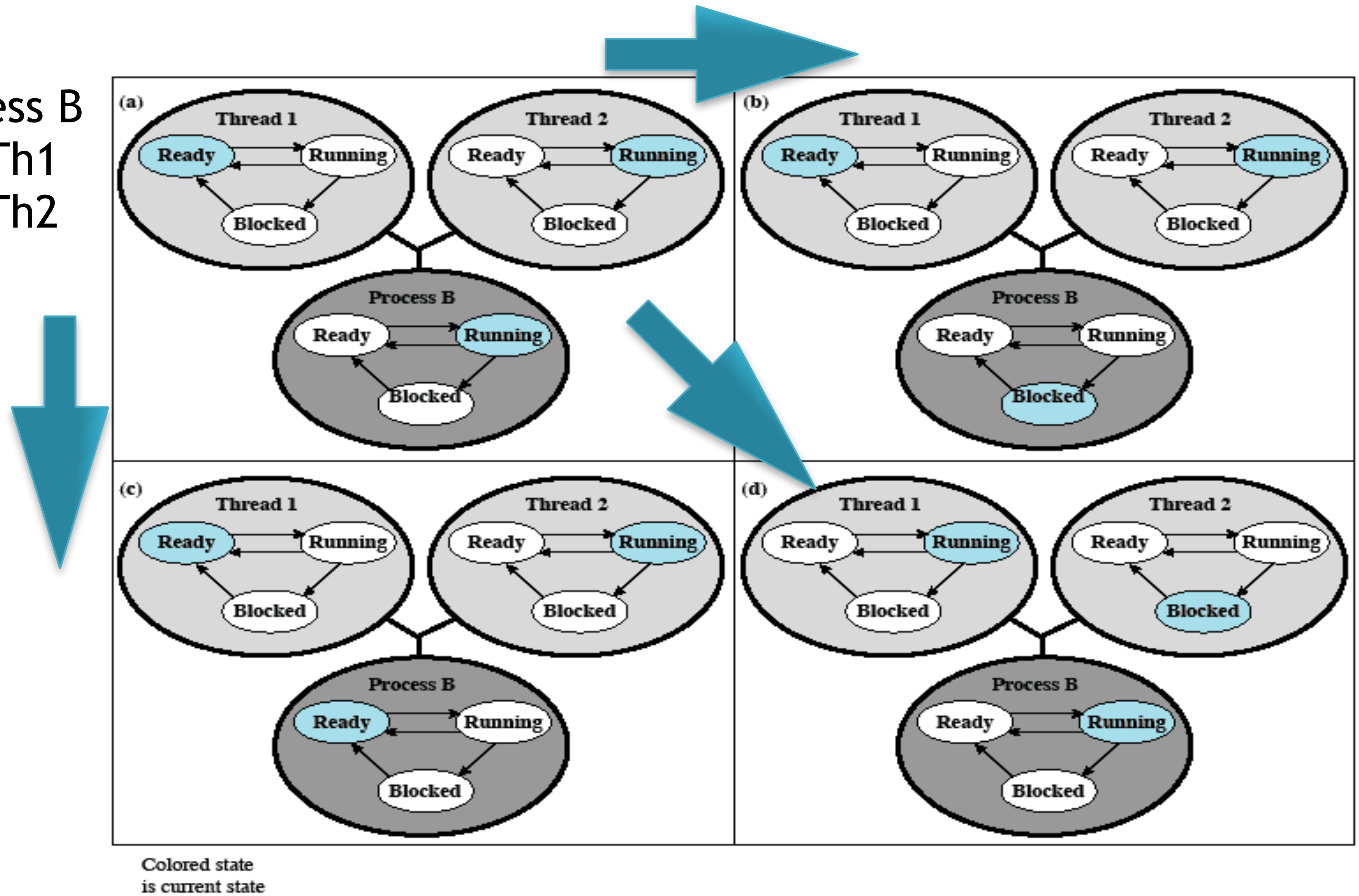


Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Advantages of User Level Threads

Thread switching does not require kernel mode privileges

→

Scheduling can be application specific

→

ULTs can run on any OS

Disadvantages of User Level Threads

- In a typical OS many system calls are blocking
  - as a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked
- Switching to another thread can only be done through "yielding" of the running thread
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing

Overcoming Process Blocking User Level Threads

- Making all system calls as non-blocking
  - >>> need to change the OS :(
- Writing a process as multiple processes rather than multiple threads
- Jacketing/wrapping: rewriting a blocking system call into a non-blocking system call

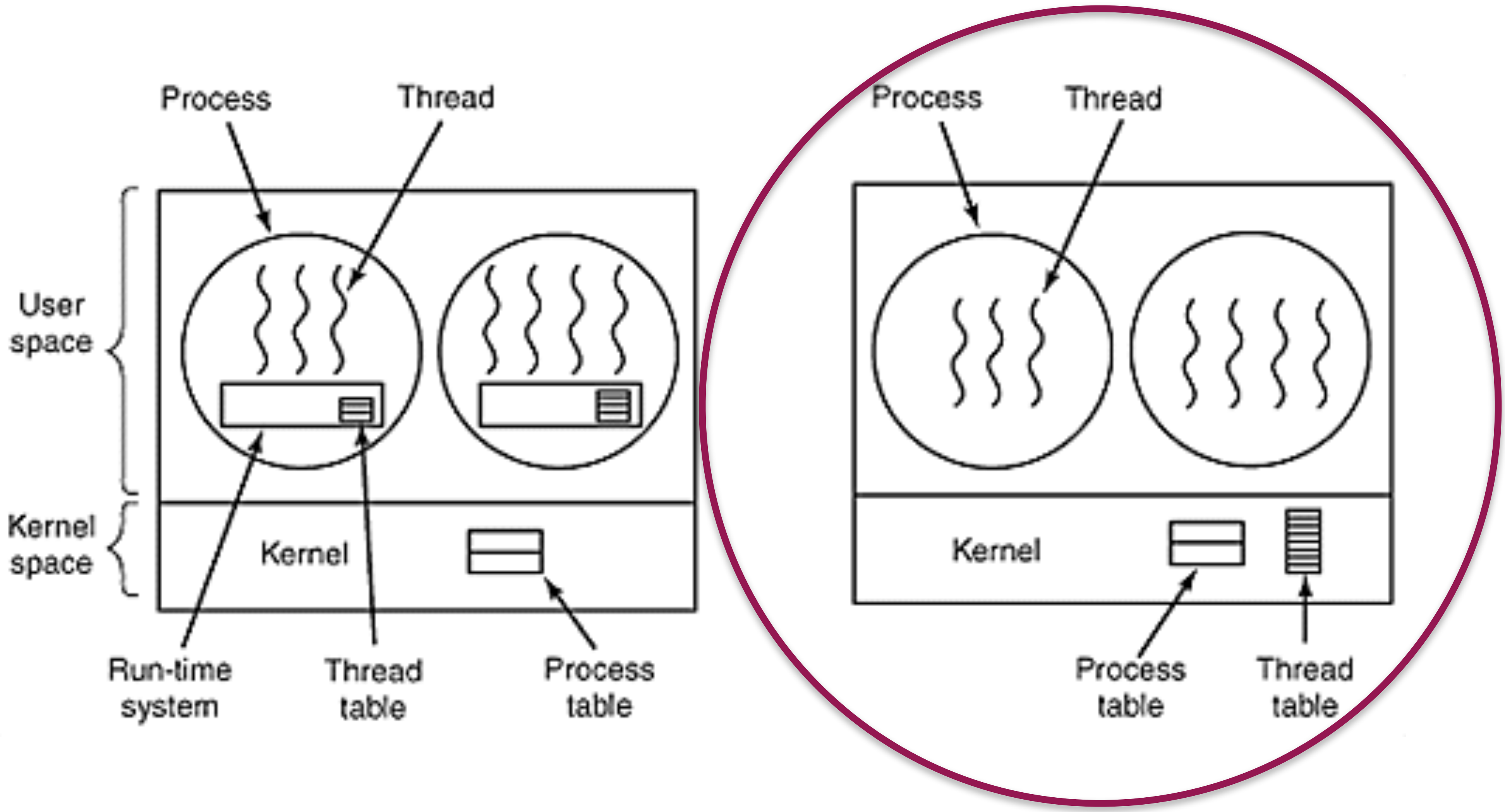
Outline

- Why Threads?
- The Thread Model
- Threads in User Space
- **Threads in the Kernel**
- Hybrid Threads
- Threads in Unix/Linux

Implementing Threads in the Kernel

- AKA Kernel Level Threads (KLT)
- The kernel maintains a thread table to keep track of all threads in the system
- The thread table keeps the same information kept for ULT in the thread table
- A system (kernel) call is needed to create a new thread or to destroy an existing one
- When a thread blocks, the kernel selects another thread from the same process or another process

Kernel Level Threads



Advantages of KLT

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines can be multithreaded

Disadvantages of KLT (1)

- Switching between kernel level threads is now expensive since it requires mode switch to the kernel.

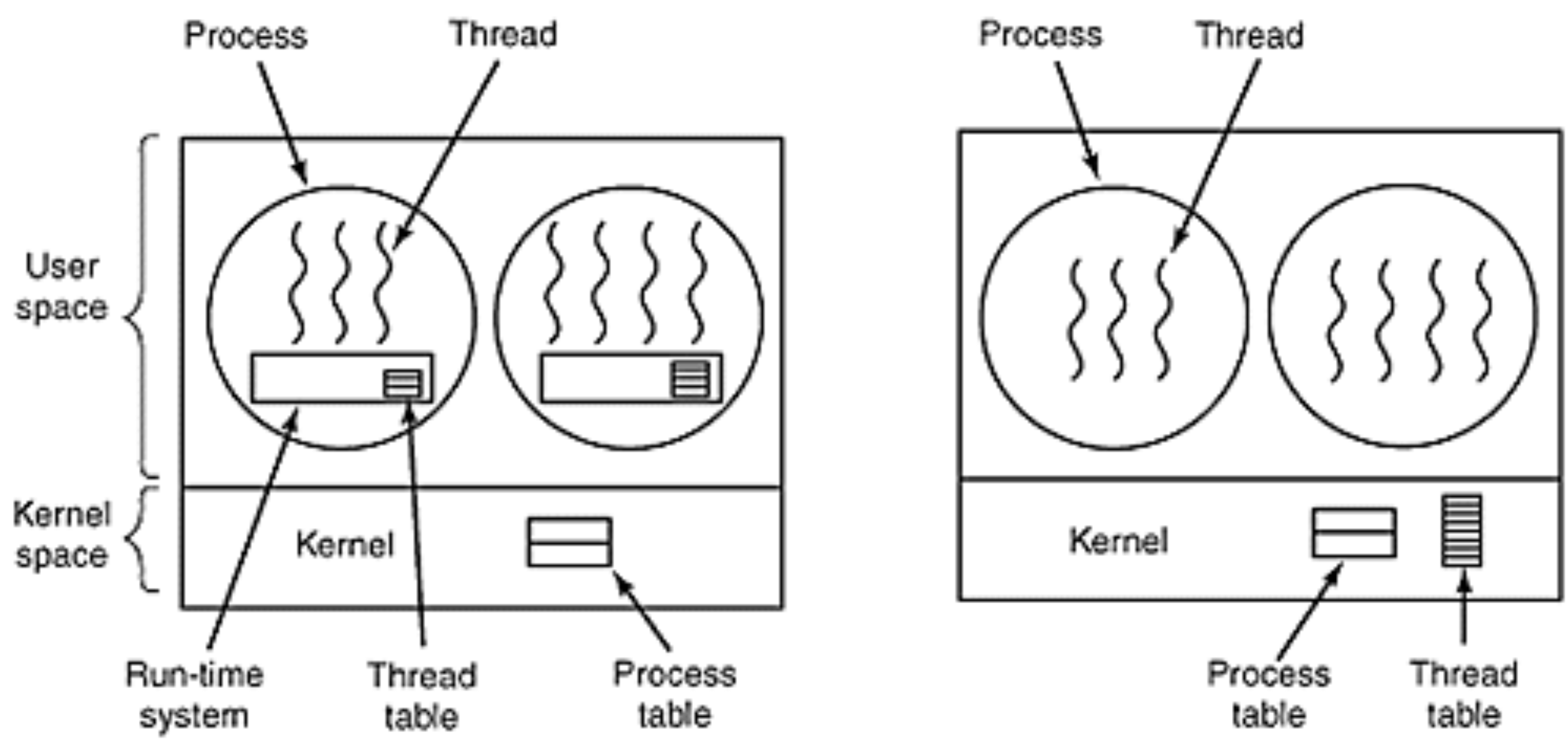
Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

•**Null fork latency** is calculated as the time to create, schedule, execute, and complete a process/thread that invokes null procedure.
•**Signal wait latency** is calculated as the time for a process/thread to signal a waiting process/thread and then wait on a condition.

Disadvantages of KLT (2)

- Greater cost for creating and destroying threads
- Solution: applications recycle threads
  - When a thread is destroyed, it is marked as not runnable, but still exist in the kernel
  - When a new thread must be created, an old thread is reactivated

User Level Threads vs Kernel Level Threads



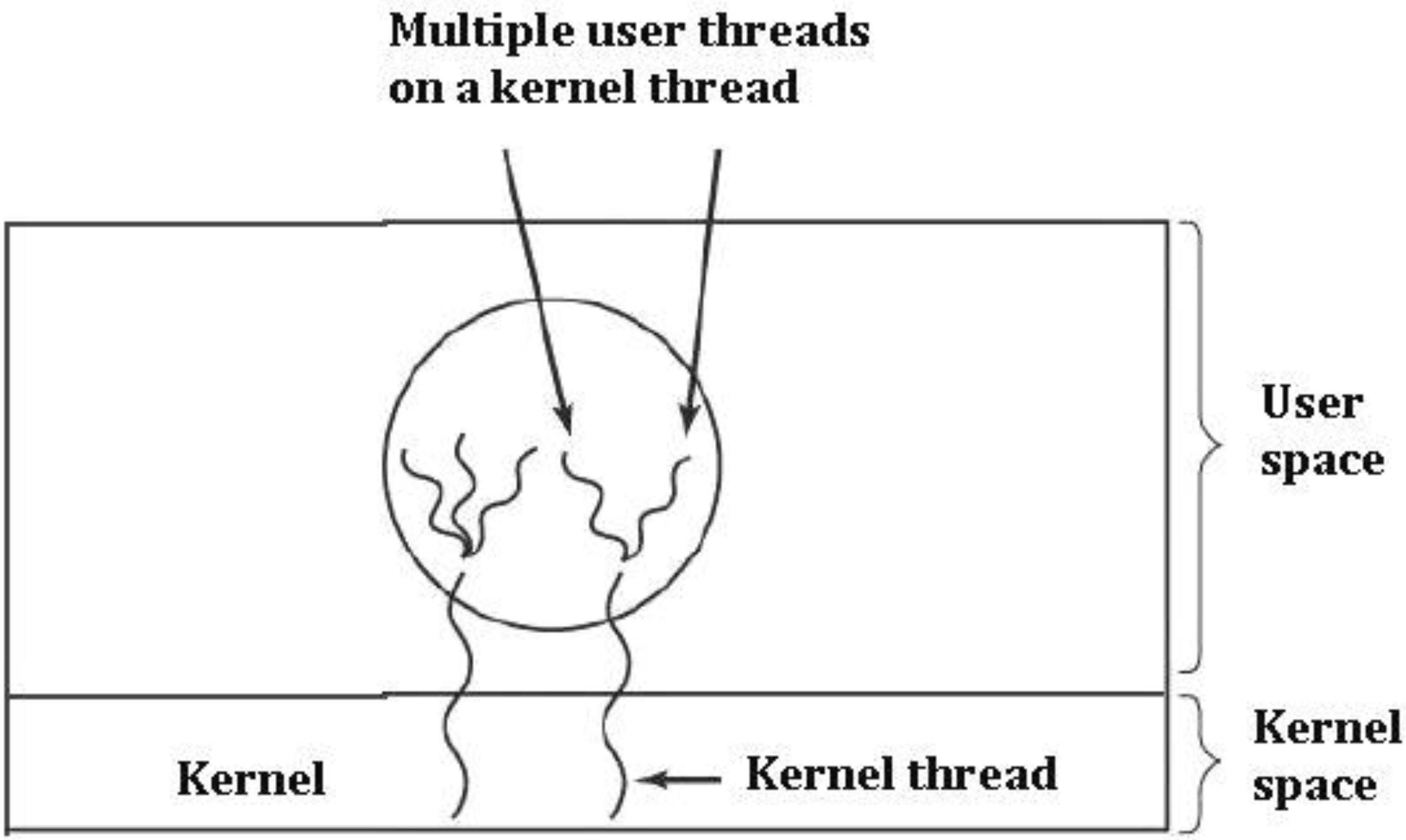
Outline

- Why Threads?
- The Thread Model
- Threads in User Space
- Threads in the Kernel
- **Hybrid Threads**
- Threads in Unix/Linux

Hybrid Implementation of Threads

- Threads are created and destroyed similar to user-level threads
- Kernel-level threads are used and then user-level threads are multiplexed onto some or all of them
- The programmer determines how many kernel threads to use and how many user-level threads to multiplex on each one
- The kernel is aware of only the kernel-level threads and schedules those

Combining User-Level Threads and Kernel-Level Threads



Multiplexing user-level threads onto kernel-level threads.

Outline

- Why Threads?
- The Thread Model
- Threads in User Space
- Threads in the Kernel
- **Hybrid Threads**
- **Threads in Unix/Linux**

Linux Threads

- Traditional UNIX systems support a single thread of execution per process
- Modern UNIX systems typically provide support for multiple kernel-level threads per process
- POSIX Thread (pThread) library was used to allow users to create threads that are mapped to one process

POSIX Threads

- 60 function calls.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure 1. Some of the Pthreads function calls

POSIX Thread: Creation

- A new thread is created using the ***pthread_create*** call
- The thread identifier of the newly created thread is returned as the function value

POSIX Thread: Termination

- When a thread has finished the work it has been assigned, it can terminate by calling ***pthread_exit***
- The thread is stopped and its stack is released
- If a thread needs to wait for another thread to terminate, it needs to call ***pthread_join***

POSIX Thread: Yielding

- A running thread will run forever until it voluntarily yields to another thread from the same process
- The thread can call ***pthread_yield*** to allow another thread to run

POSIX Thread: Attributes

- ***Pthread_attr_init*** creates the attribute structure associated with a thread and initializes it to the default values
- ***Pthread_attr_destroy*** removes a thread's attribute structure, freeing up its memory

Pthread Example

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

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d0, tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d0, i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d0, status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Figure 2. An example program using threads

Outline

- Why Threads?
- The Thread Model
- Threads in User Space
- Threads in the Kernel
- **Hybrid Threads**
- Threads in Unix/Linux
- **Notes**

Parallelism vs Concurrency

- A system is parallel if it can perform more than one task <u>simultaneously</u>
  - multicore, multiprocessor, GPU
- A concurrent system supports more than one task by allowing <u>all the tasks to make progress</u>
  - one processor
- Hyper-threading: supporting multiple threads per core
  - E.g. Intel supports 2 threads per core
  - Multiple threads loaded to the same core for faster switching

More on Thread Libraries

- Common libraries: POSIX Pthreads, Windows, and Java
- Pthread: provided as either a user-level or a kernel-level library
- Windows thread library: a kernel-level library available on Windows systems
- Java thread library: allows threads to be created and managed directly in Java programs
  - However, JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system

Thank You !