Chapter 4: Multithreaded programming

- Introduction
- Processes and threads
- Multithreading Models
- Threading Issues
- Pthreads
- Solaris 2 Threads
- WINDOWS2000
- LINUX
- JAVA

References

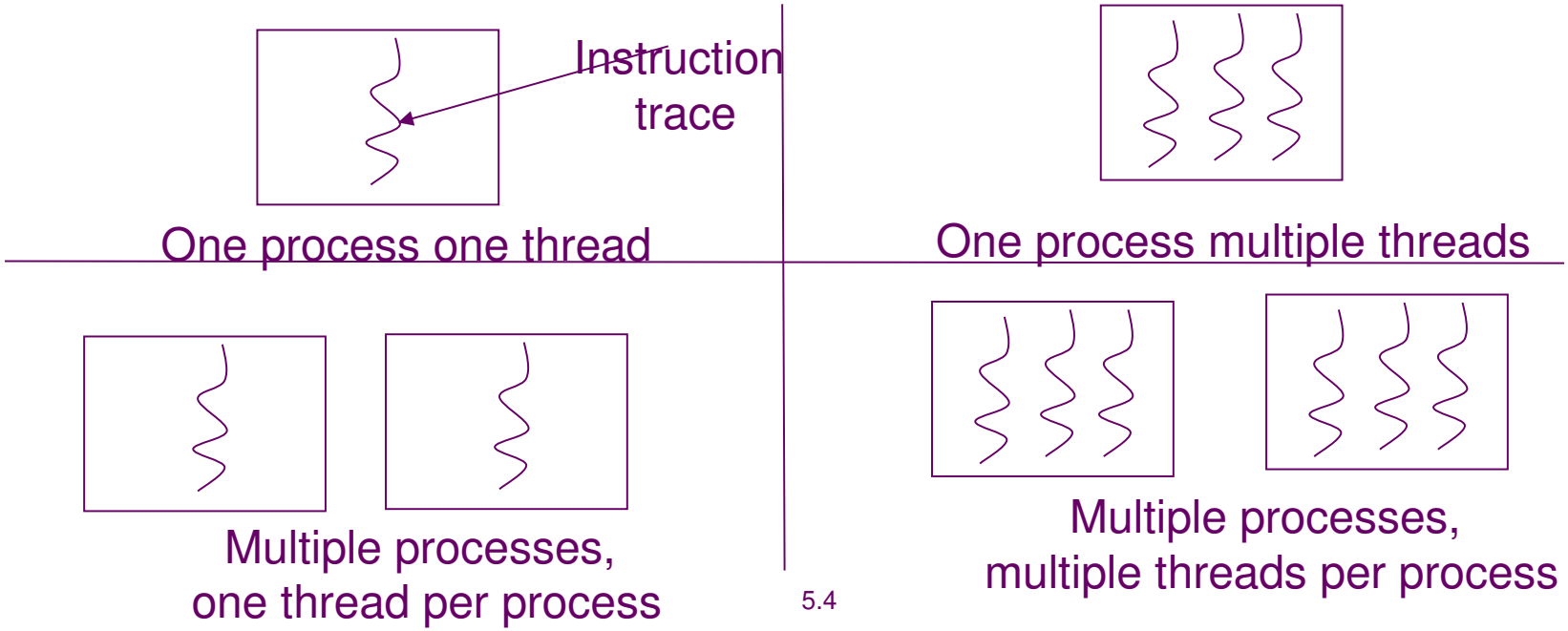
- Chapter 4 of Operating Systems Concepts, 8th edition, Silberschatz, Galvin, Gagne, Wiley
- Chapter 4, Operating systems, Fourth edition, William Stallings, Pearson education.

Threads: introduction

- Process has two characteristics
  - Resource ownership
    - Virtual address space is allocated
    - From time to time a process may be assigned main memory plus other resources such as I/O channels, I/O devises and files.
    - The OS performs a protection function to prevent unwanted interference between processes with respect to resources.
  - Scheduling/execution
    - The execution of a process follows an execution path (trace) through one or more programs.
    - The execution can be interleaved with that of other processes.
    - It is an entity that is scheduled and dispatched by the operating system.
- The two characteristics are independent
  - The unit of dispatching is called thread
  - The unit of resource ownership is called as a process or task.
- Examples: writing of file to disk
  - Compute on one batch of data while reading another batch of data.

Multi-threading

- Multithreading refers to the ability on an operating system to support multiple threads of execution within a single process.
- Traditionally there is s single thread of execution per process.
  - Example: MSDOS supports a single user process and single thread.
  - UNIX supports multiple user processes but only support one thread per process.
- Multithreading
  - Java run time environment is an example of one process with multiple threads.
  - Examples of supporting multiple processes, with each process supporting multiple threads
    - Windows 2000, Solaris, Linux, Mach, and OS/2

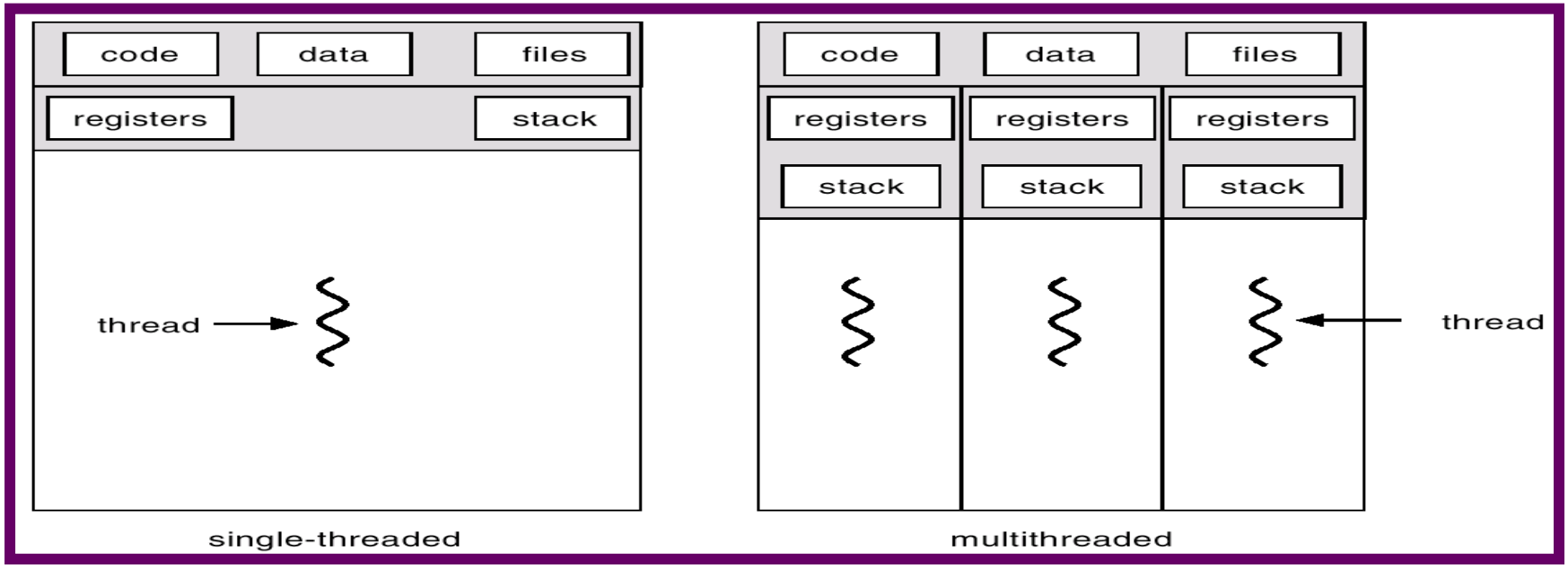


Process and Thread

- The following are associated with the process.
  - A virtual address space that holds the process image.
  - Protected access to processors, and others processes, files, and I/O resources (devices and channels)
- Within a process there may be one or more threads, each with the following
  - A thread execution state (Running, Ready, etc.)
  - Individual execution state; one way to view a thread is an independent program counter operating within a process.
  - Each thread has a control block, with a state (Running/Blocked/etc.), saved registers, instruction pointer
  - Separate stack
  - **Shares memory and files with other threads that are in that process**

Single and Multithreaded Processes

- Faster to create a thread than a process
- In a single threaded process model, the representation of a process includes its PCB, user address space, as well as user and kernel stacks.
  - When a process is running. The contents of these registers are controlled by that process, and the contents of these registers are saved when the process is not running.
- In a multi threaded environment
  - There is a single PCB and address space,
  - However, there are separate stacks for each thread as well as separate control blocks for each thread containing register values, priority, and other thread related state information.



Threads: Benefits

- All the threads of a process share the state and the resources of the process.
- They reside in the same address space and have access to the same data.
  - When one thread alters an item of data in memory, other threads see the results whenever they access the item.
- Key Benefits
  - **Responsiveness:**
    - Allow program to continue to run even a part of it is blocked or performing lengthy operation.
      - Example: a multi-threaded application still allow user interaction in one thread while an image is being loaded in another thread.
  - **Resource sharing:** application can have different threads of activity all within the address space.
  - **Economy:**
    - It takes less time to create a new thread in an existing process than to create a new process.
      - 10 times improvement in the speed: MAC over UNIX
      - In Solaris creating a process is 30 times slower than is creating a thread and context switching is 30 times slower.

Threads: Benefits…

- **Economy...**
  - ✔ It takes less time to terminate a thread than a process.
  - ✔ It takes less time to switch between two threads within the same process.
  - ✔ Communication is easy with threads.
    - Communication between the processes required OS intervention.
    - Communication between threads can be through shared memory without OS intervention.
- **Utilization of multi processor architectures**
  - ✔ Multithreading exploits multi-processor architectures very well
  - ✔ Each thread may be running parallel on a different processor.
  - ✔ A single threaded process can only run on single CPU, no matter how many are available.
  - ✔ Multithreading increases concurrency.
  - ✔ Single processor architecture only creates illusion of parallelism.

- If there is an application or function that should be implemented as a set of related units of execution, it is far more efficient to do as a collection of threads rather than a collection of processes.

Examples of use of threads

- Single user multiprocessing system
  - **Foreground/Background**
    - ✔ In a spreadsheet program, one thread could display menus and read user input, while another thread executes user commands and updates the spreadsheet.
  - **Asynchronous Processing** – Backing up in background
    - ✔ Design of word processor write RAM buffer to disk once in every minutes.
  - **Faster Execution** – Read one set of data while processing another set
    - ✔ A multithreaded process can compute one batch of data while reading the next batch from a device.
    - ✔ One a multiprocessor system multiple threads from the same process may be able to execute simultaneously.
  - **Modular program structure**
    - ✔ Programs that involve variety of activities or a variety of resources and destinations of input and output are easier to design.

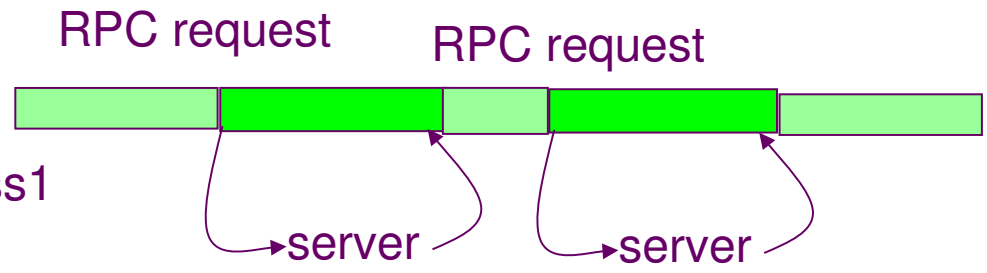
Thread Scheduling and dispatching

- Scheduling and dispatching is done at thread level.
- The actions that affect all the threads in the process should be managed at the process level.
- Suspension involves swapping of address space out of memory
  - Since all the threads in the process share the address space, all threads must enter the suspend state at the same time.
  - Termination of a process terminates all the threads in the process.

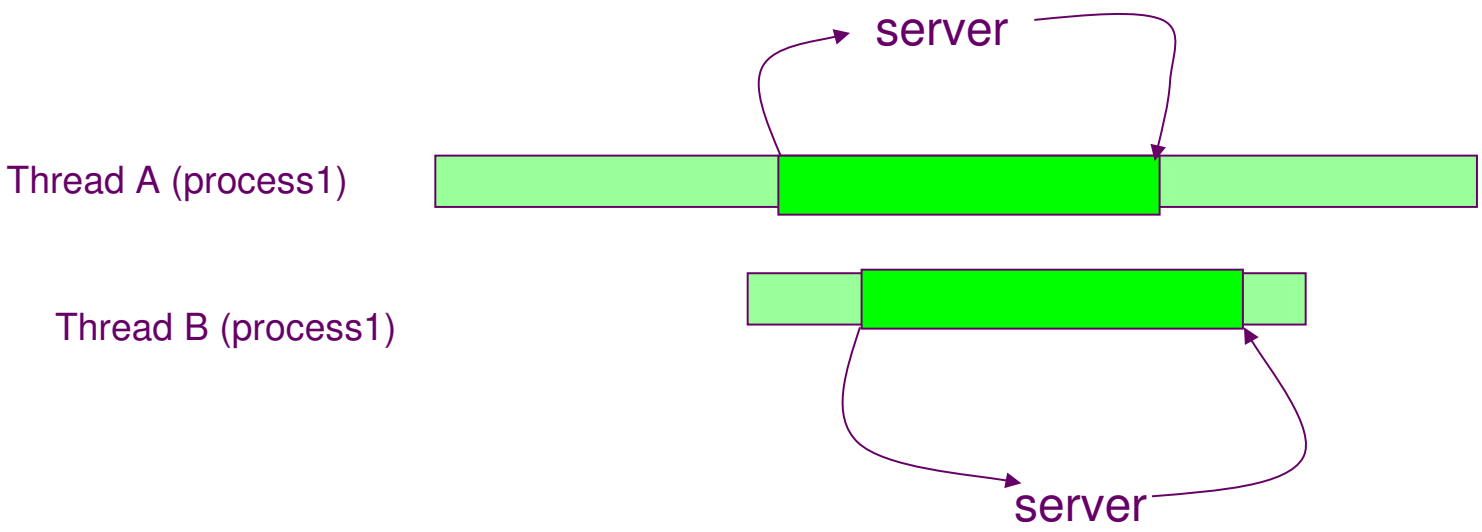
Thread Scheduling and dispatching

- Thread operations
  - *Spawn* – Creating a new thread
  - *Block* – Waiting for an event
  - *Unblock* – Event happened, start new
  - *Finish* – This thread is completed
- Generally a thread can block without blocking the remaining threads in the process
  - Allow the process to start two operations at once, each thread blocks on the appropriate event
- Must handle synchronization between threads
- System calls or local subroutines
  - Thread generally responsible for getting/releasing locks, etc.

Thread interleaving example



RPC using single thread



RPC using one thread per server

Multicore Programming

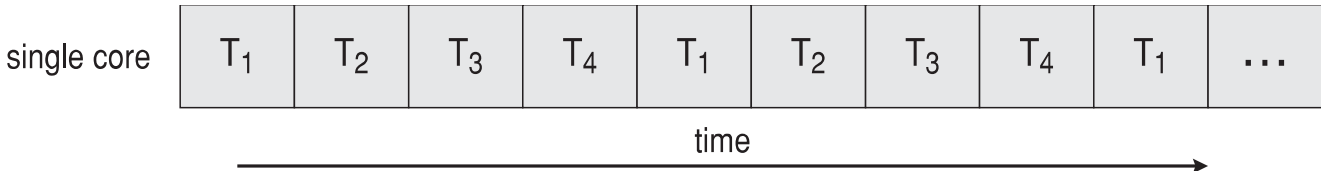
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Load Balance: Equal load on every core**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

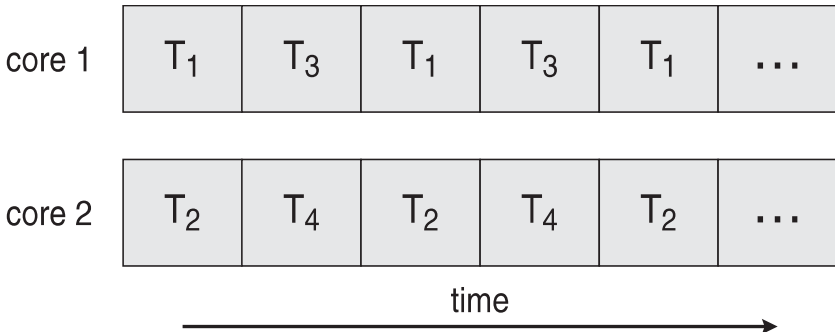
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



Multithreading Models

- Many systems provide support for both user and kernel threads resulting different multi threading models. The following are common models.
  - Many-to-One
  - One-to-One
  - Many-to-Many

Managing threads

- Support for threads can be provided at either user level or at kernel.
  - **User-level threads**
  - **Kernel-level threads**

User-level Threads

- Thread management is done by user-level threads library
- Application creates/manages all threads using a library
- System schedules the process as a unit
  - Scheduling, etc. is all in user space (faster)
  - Scheduling can be application specific
  - Does not require O.S. support
- Has problems with blocking system calls
- Cannot support multiprocessing
- Examples

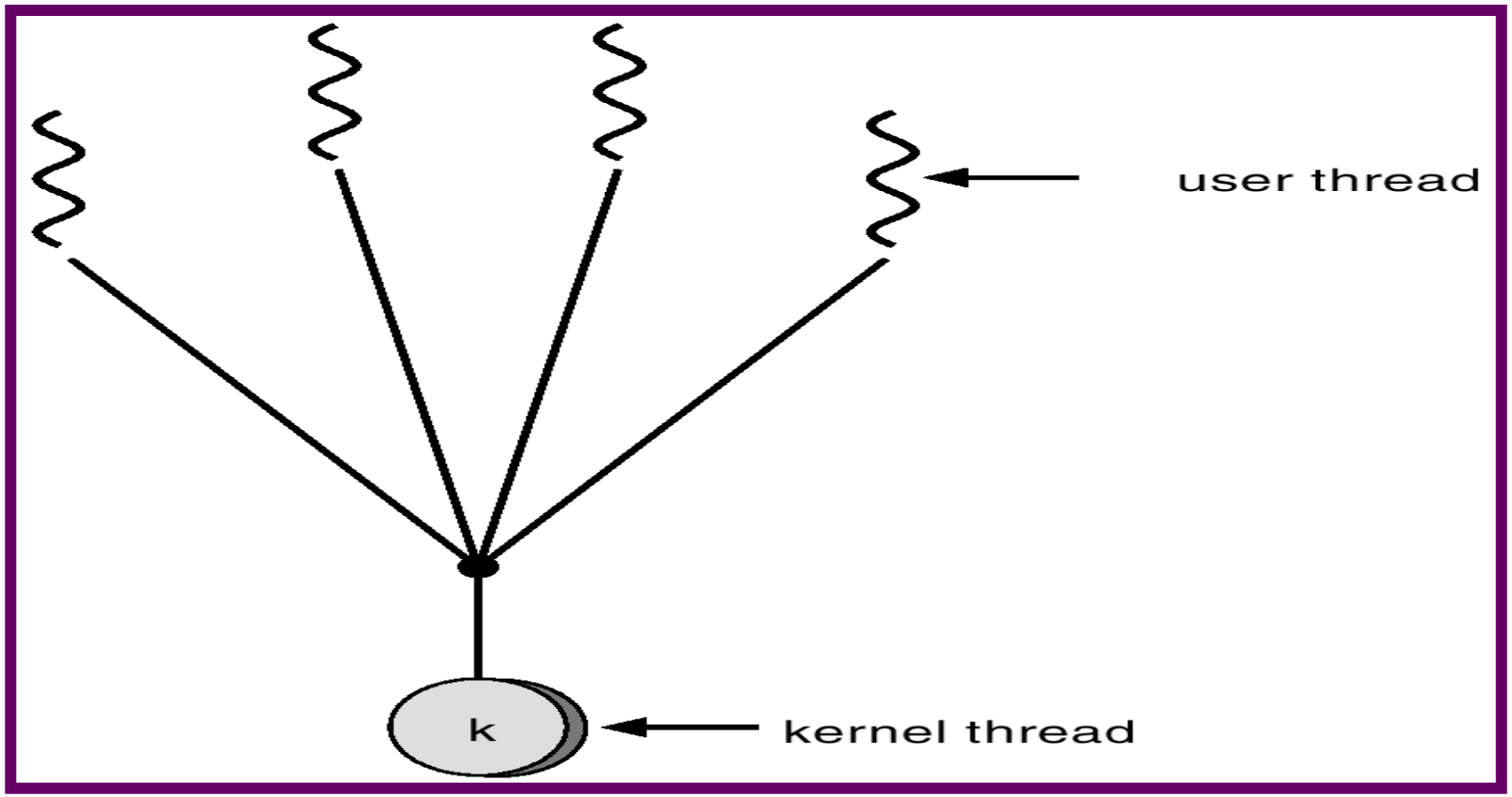
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

Kernel-level Threads

- Supported by the Kernel
- Kernel handles managing threads
- Easier to support multiple processors
- Kernel itself may be multithreaded
- Need user/kernel mode switch to change threads
- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux
- Other Approaches
  - Each kernel thread may have multiple user threads

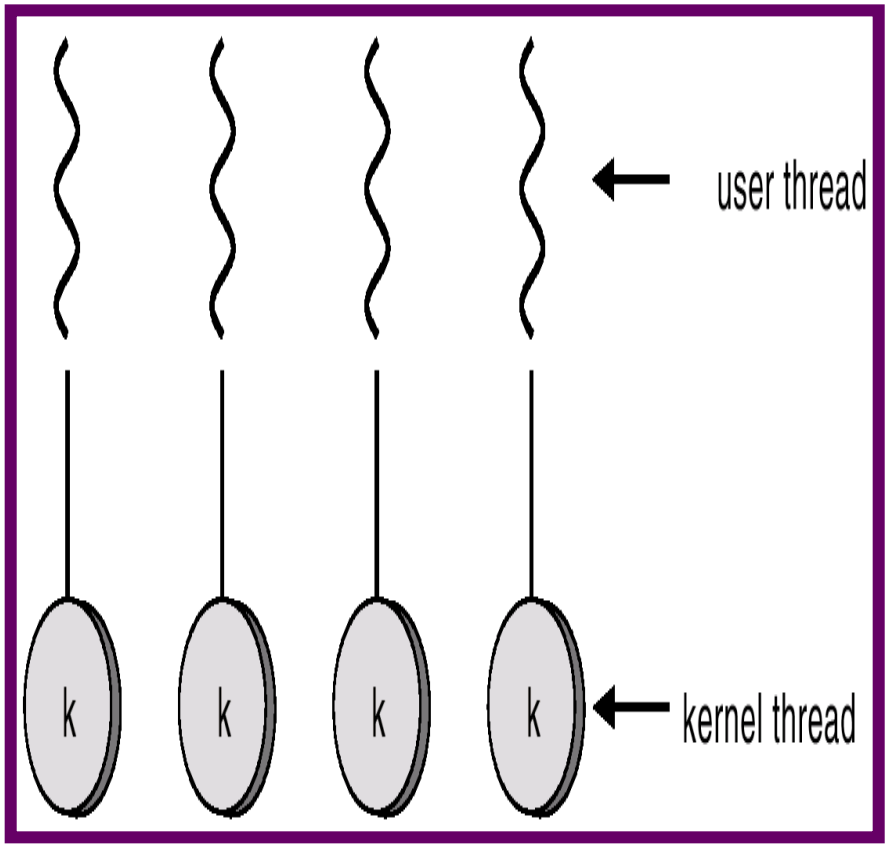
Many-to-One

- Many user-level threads mapped to single kernel thread.
- Thread management is done in user space.
- Entire process will block if a thread makes a blocking system call.
- Used on systems that do not support kernel threads.



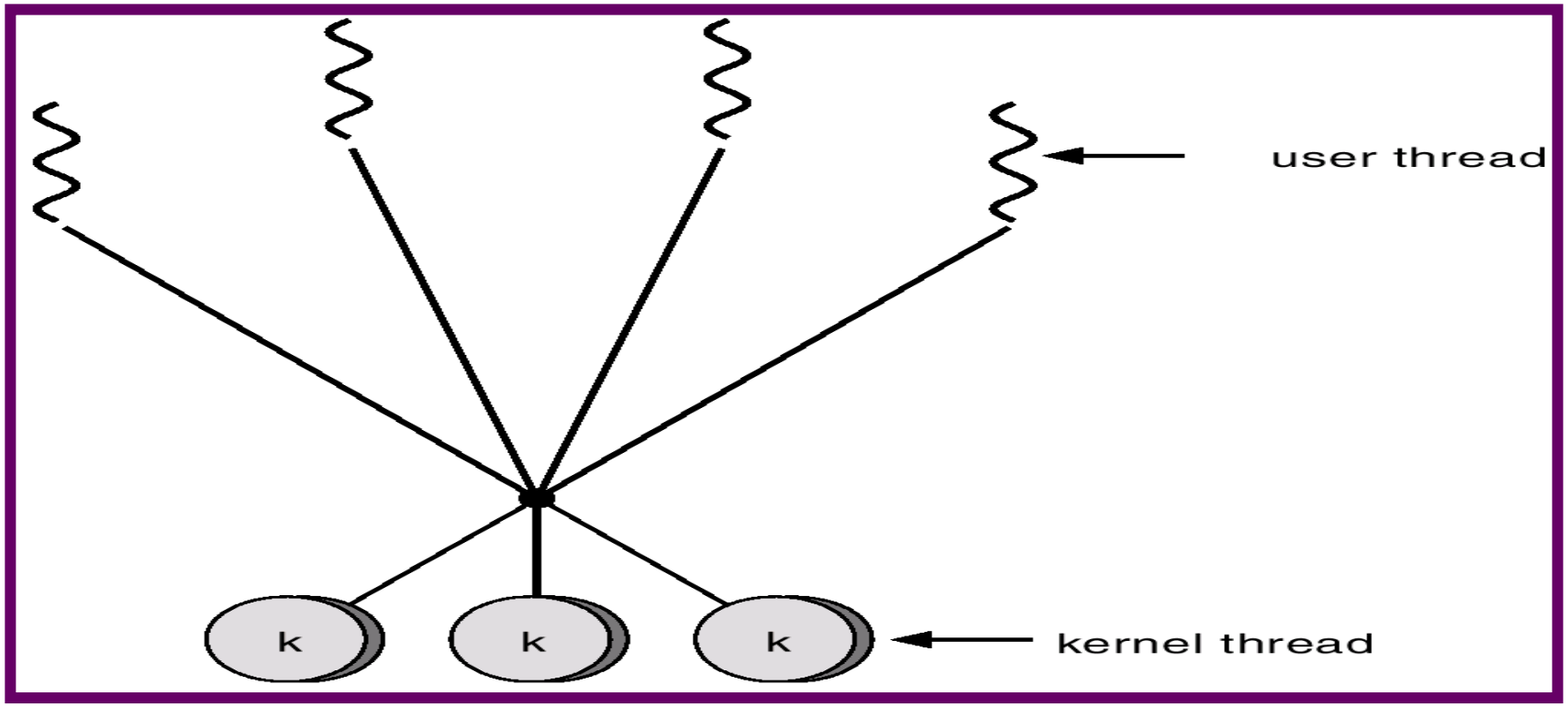
One-to-One

- Each user-level thread maps to kernel thread.
- Provides more concurrency than many-one model
- Allows other thread to run if a thread makes blocking system call
- Drawback: creating user thread requires creating the corresponding kernel thread.
- Most systems restrict the number of threads supported by the system
- Examples
  - Windows 95/98/NT/2000
  - OS/2

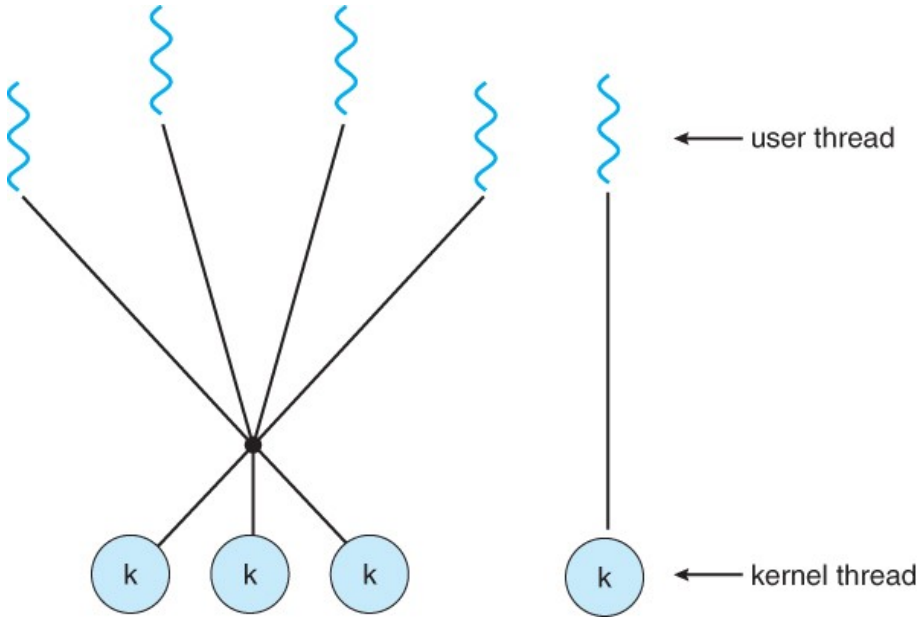


Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- When thread performs a blocking system call, the kernel can schedule another thread for execution.
- Solaris 2 and Windows NT/2000 with the *ThreadFiber* package



Two-level Many to Many model



- Allows
  - multiplexing of many user-level threads to a small number or equal number of kernel threads, and
  - User level thread to be bound to a kernel thread

Threading Issues

- Some of the issues to be considered for multi-threaded programs
  - Semantics of fork() and exec() system calls.
  - Thread cancellation.
  - Signal handling
  - Thread pools
  - Thread-specific data

Threading Issues…

- **Semantics of fork() and exec() system calls.**
  - In multithreaded program the semantics of the fork and exec systems calls change.
  - If one thread calls fork, there are two options.
    - ✔ New process can duplicate all the threads or new process is a process with single thread
    - ✔ Some systems have chosen two versions of fork.
- **Thread cancellation.**
  - Task of terminating thread before its completion
    - ✔ Example: if multiple threads are searching database, if one gets the result others should be cancelled.
  - **Asynchronous cancellation**
    - ✔ One thread immediately terminates the target thread.
  - **Deferred cancellation**
    - ✔ The target thread can periodicallly check if it should terminate.

Threading Issues: Signal handling

- A signal is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received synchronously or asynchronously.
- All signals follow specific patterns
  - A signal is generated by the occurrence of a particular event.
  - A generated signal is delivered to a process
  - On delivered, the signal must be handled.
- Examples: illegal memory access or division by zero.
- In single threaded program, it is easy, But in multithreaded program several options exist.
  - Deliver a signal to specific thread
  - Deliver a signal to every thread in the process
  - Deliver the signal to certain threads in the process.
  - Assign a specific thread to receive all signals for the process.
    - Solaris 2
- It depends on the type of the signal
  - (<control>C) should be sent to all the threads.

Threading Issue: Thread pools

- Consider a scenario of multithreading a web server.
- Whenever a server receives a request, it creates a separate thread.
  - First problem is creating thread before hand and discarding it after completion.
  - Second problem is unlimited threads could exhaust system resources such as CPU time and main memory.
- One solution to this issue is **thread pools**
  - The idea is create a number of threads at process startup and place them in a pool, where they sit and wait for work.
  - If the pool has no available thread, the server waits until one is free.
- **Benefits:**
  - It is faster to service with an exiting thread than waiting to create a new thread.
  - A thread pool limits the number of threads at any point which is important on systems that can not support a large number of concurrent threads.

Threading Issues: Thread specific data

- Threads share the data
- Sharing is one of the benefits of multiprogramming.
- However, each thread might needs own copy of certain data in some circumstances.
- Such data is called thread-specific data.
  - For example in transaction processing system, each transaction may be assigned a unique number.
  - To associate each thread with its unique identifier, we could use thread specific data.

Threading Issues: Scheduler activations

- Communication between the kernel and the thread library.
- In many to many model, intermediate data structure is placed betwee kernel and user-level threads.
- To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run.
- If the kernel thread blocks, the LWP blocks as well.
- Typically, the application requires many LWPs run efficiently.
  - One LWP is required for each concurrent blocking system call.
  - Suppose, five file reads occur concurrently, five LWPs are needed. If the process has four LWPs, then fifth must wait for one of the LWPs.
- For communication, the scheme “scheduler activation” is used.
- Scheduler activation: upcalls
  - The kernel informs application about certain events wich is called **upcall** procedure
  - Upcalls are handled with thread library “**upcall handler**”
  - The kernel makes an **upcall** to the application regarding blocking of thread.
  - The upcall handler saves the state of blocking thread relinquishes the virtual processor on which blocking thread is running.
  - The upcall handler schedules another thread.
  - When waiting event occurs, the kernel maks another upcall to thread library informing it that the previously blocked thread is now eligible to run.

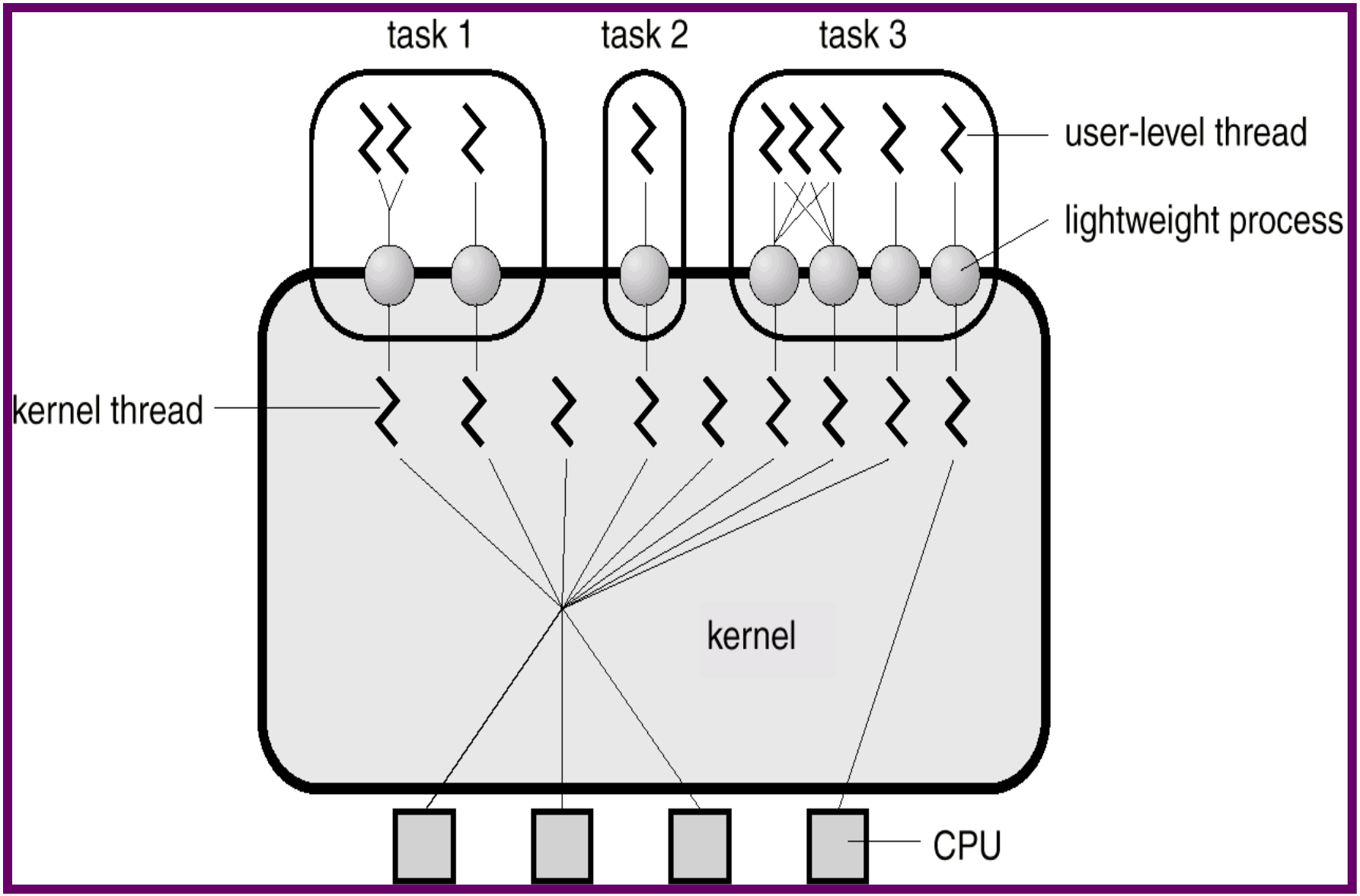
Pthreads

- a POSIX (Portable Operating System Interface for uniX) 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.
  - MULTICS: Multiplexed Information and Computing System
  - UNICS: Uniplexed Operating and Computing System
  - UNICS ➔ UNIX
  - LINUX: A UNIX- like operating system named after Linus Torvalds (Norwegian programmer) who initiated the work. LINUX is free and source code is open. Anyone can work on LINUX and post new code to improve it.

Solaris threads

- Implements Many-to-Many mapping
- Solaris uses four thread related concepts:
  - **Process:** Normal Unix process which includes user's address space, stack, and PCB
  - **User-level threads((ULTs):** Implemented through a thread library through the address of a process. These are invisible to OS.
  - **Lightweight processes (LWP):** mapping between ULTs and kernel threads.
    - ✔ Each LWP supports one or more ULTs and maps to one kernel thread. LWPs can be scheduled independently.
  - **Kernel threads**: These are fundamental entities that can be scheduled and dispatched to run on one of the system processors.

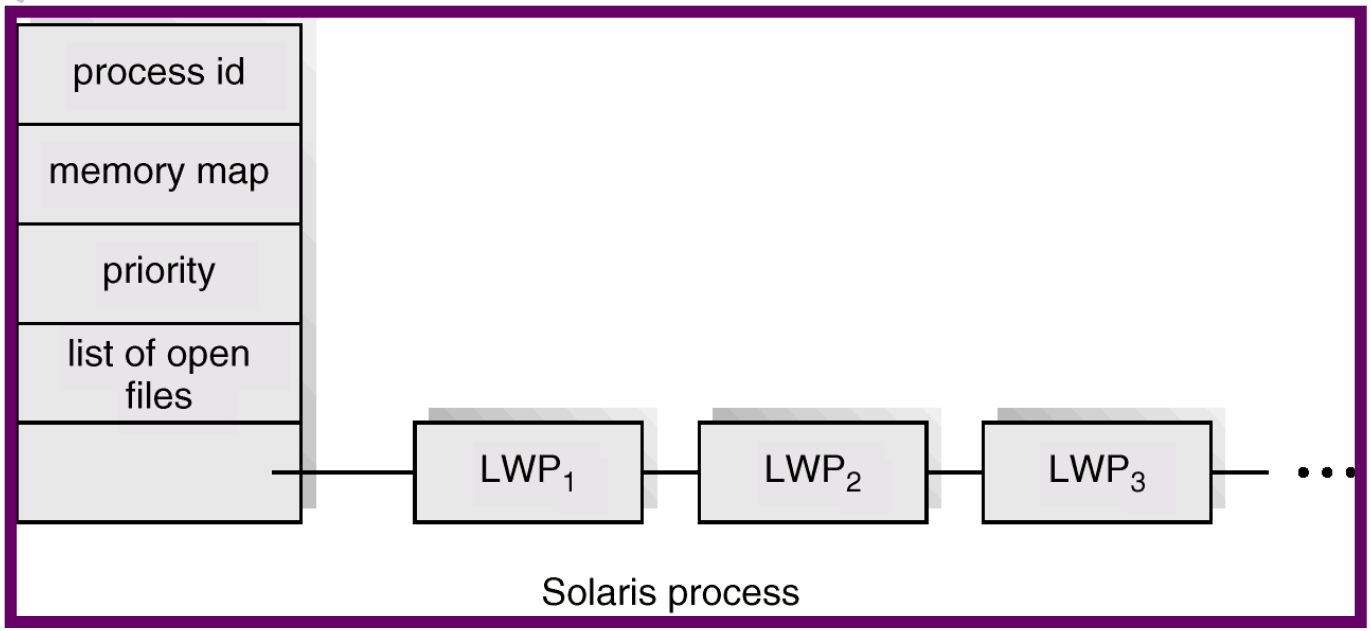
Solaris 2 Threads



Data structures of Solaris threads

- A user level thread contains a thread ID; register set, stack and its priority. None of these are kernel resources. All exists in user space.
- An LWP has a register set for the user-level thread it is running, as well as memory and account information.
  - A LWP is a kernel data structure and it resides in kernel space.
- A kernel thread has only a small data structure and a stack. The data structure includes
  - a copy of the kernel registers,
  - a pointer to the LWP to which it is attached,
  - priority and scheduling information.

Solaris Process



Windows 2000 Threads

- Implements the one-to-one mapping.
- Each thread contains
  - a thread id
  - register set
  - separate user and kernel stacks
  - private data storage area

Linux Threads

- Linux refers to them as *tasks* rather than *threads*.
- Thread creation is done through clone() system call.
- Clone() allows a child task to share the address space of the parent task (process)

Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java threads are managed by the JVM.
- Difficult to classify as a user or kernel threads.

Java Thread States