

Operating Systems ICS 431



Weeks 5-6

Ch. 4: Multi-Threading Programming

Dr. Tarek Helmy El-Basuny

Ch 4: Multi-Threading Programming

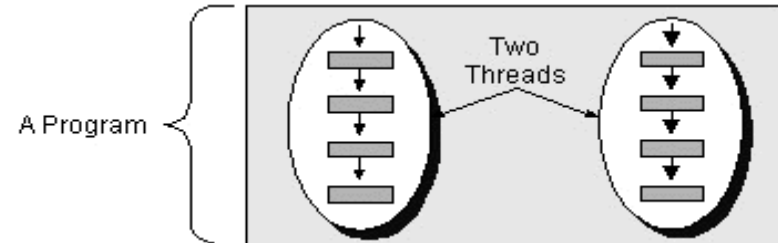
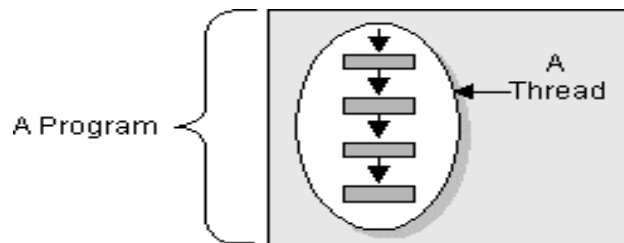
- Threads Definition
- Processing Modes in the OSs.
- Why does the OS support multithreading?
- Examples of Multithreaded Programs
- Thread's Control Block
- Benefits of Threads vs. Processes
- Thread's Life Cycle
- User's and Kernel's Level Threads
- Combining ULT and KLT Models
 - Many-to-One
 - One-to-One
 - Many-to-Many
- Threading Issues
 - Thread Cancellation, Threads Pool, Signal Handling
- Threads Scheduling
 - Priority Scheduling, and Priority Inversion/Inheritance Mechanisms
- Threading in Different Platforms:
 - Windows, Solaris, Linux, Mac OS, etc.

A Process Concept

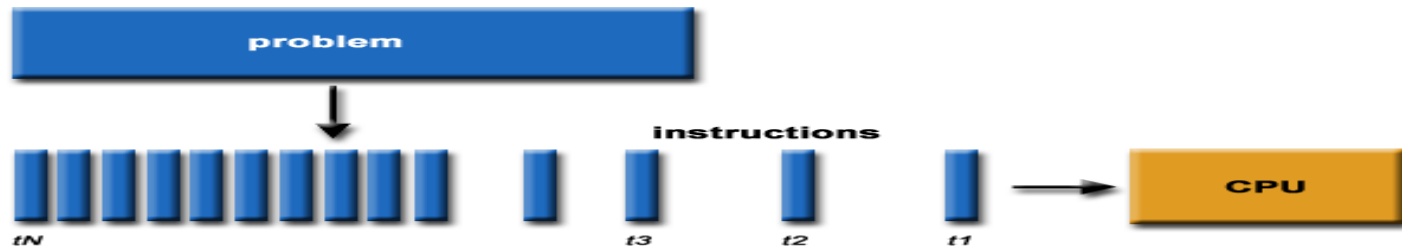
- A process is a key OS abstraction that users see.
- The environment you interact with when you use your computer is built up out of processes.
 - The power point we use is a process.
 - The browser you use is a process.
 - The shell you type commands into is a process.
 - When you execute a program you have just compiled, the OS generates a process to run that program.
- Let us think of the browser as a process.
 - Does it support concurrency (i.e. browsing a page, down-loading, playing a video, ...)?
 - Is it a responsive process?
 - If yes, why?

Processing Modes

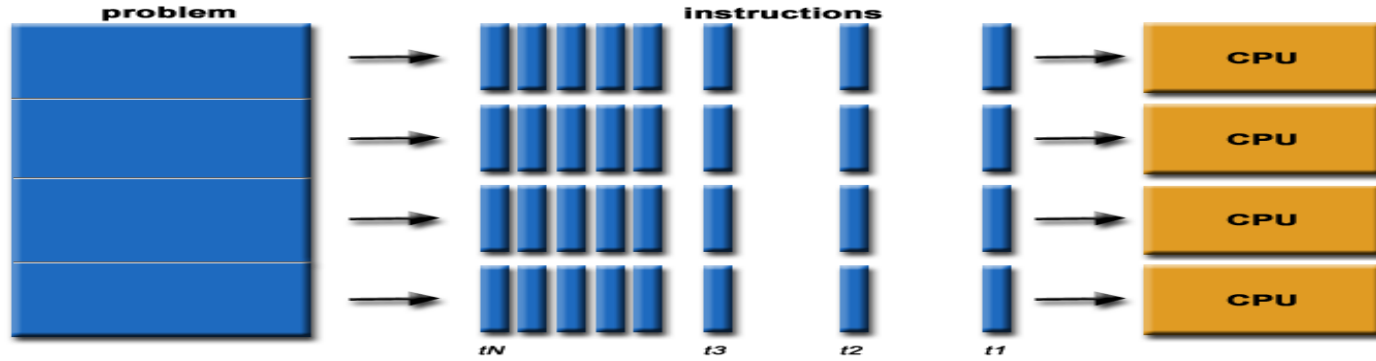
- In uni-processing mode: the OS supports a single process to run.
- In Multiprocessing/Multitasking mode: the OS is sharing a single processor or multiple processors among several processes concurrently, interleaving I/O pound with CPU pound processes.
- In parallel processing mode: The OS uses more than one processor to simultaneously run multiple processes in parallel.
- Multithreading is a kind of multitasking/multiprocessing with low overheads and no protection of tasks from each other, all threads share the same address space (of the parent process) in memory.
- Processes can do several things concurrently by running more than one thread.
- A process (Web Browser) may consist of the following threads:
 - GUI thread, I/O thread, Computation thread.
- A word processing consists of multiple threads, i.e. spell checker, auto save, .



Sequential vs. Parallel Processing Modes



Sequential Processing

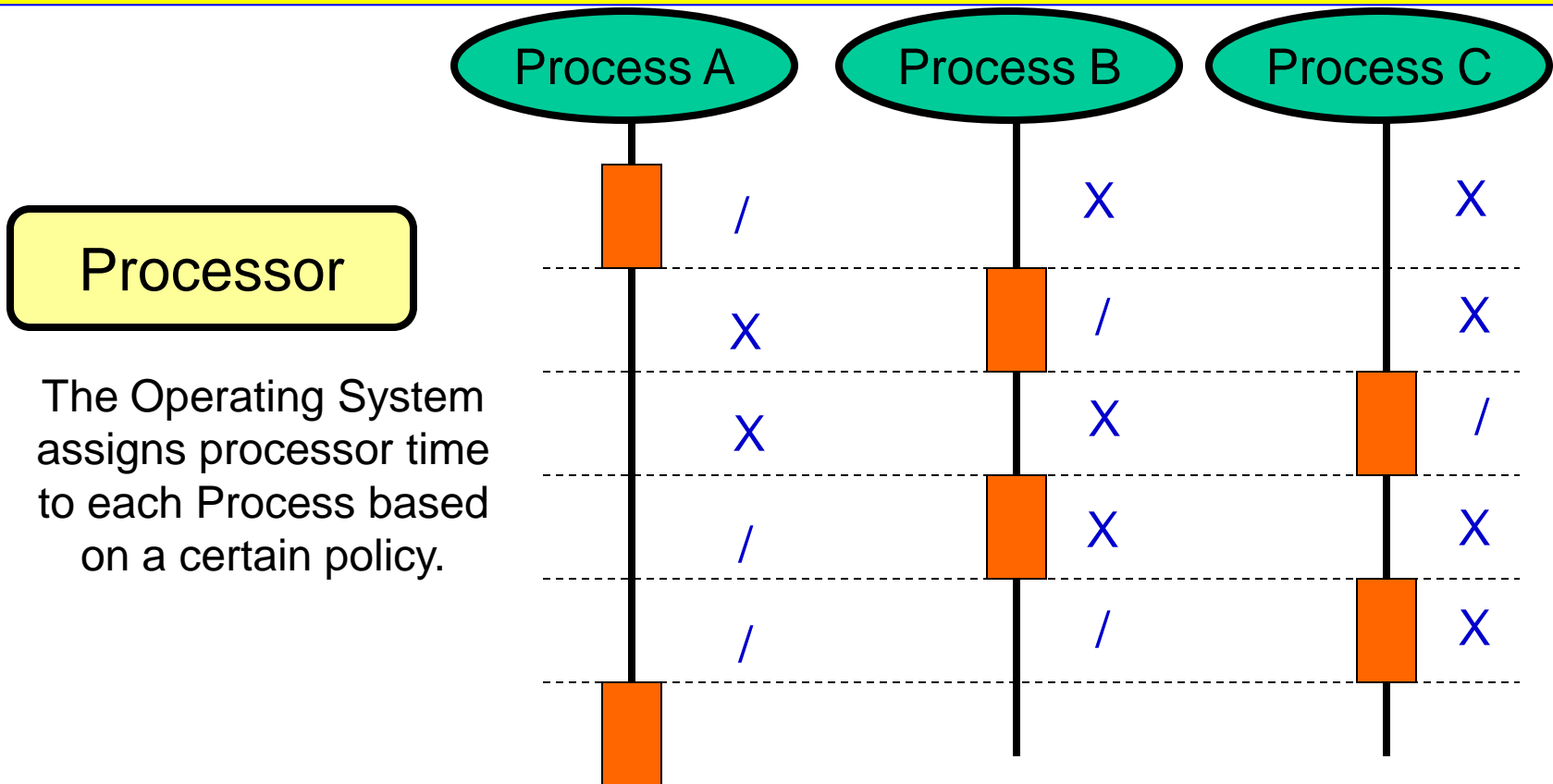


Parallel Processing

- Processes running on multiple-processors may be **Independent** or **Dependent**.
- No synchronization is required for independent processes but it is needed for dependent processes.
- We are going to study latter different synchronization methods.

Multi-Processing with **Single** Processor Mode

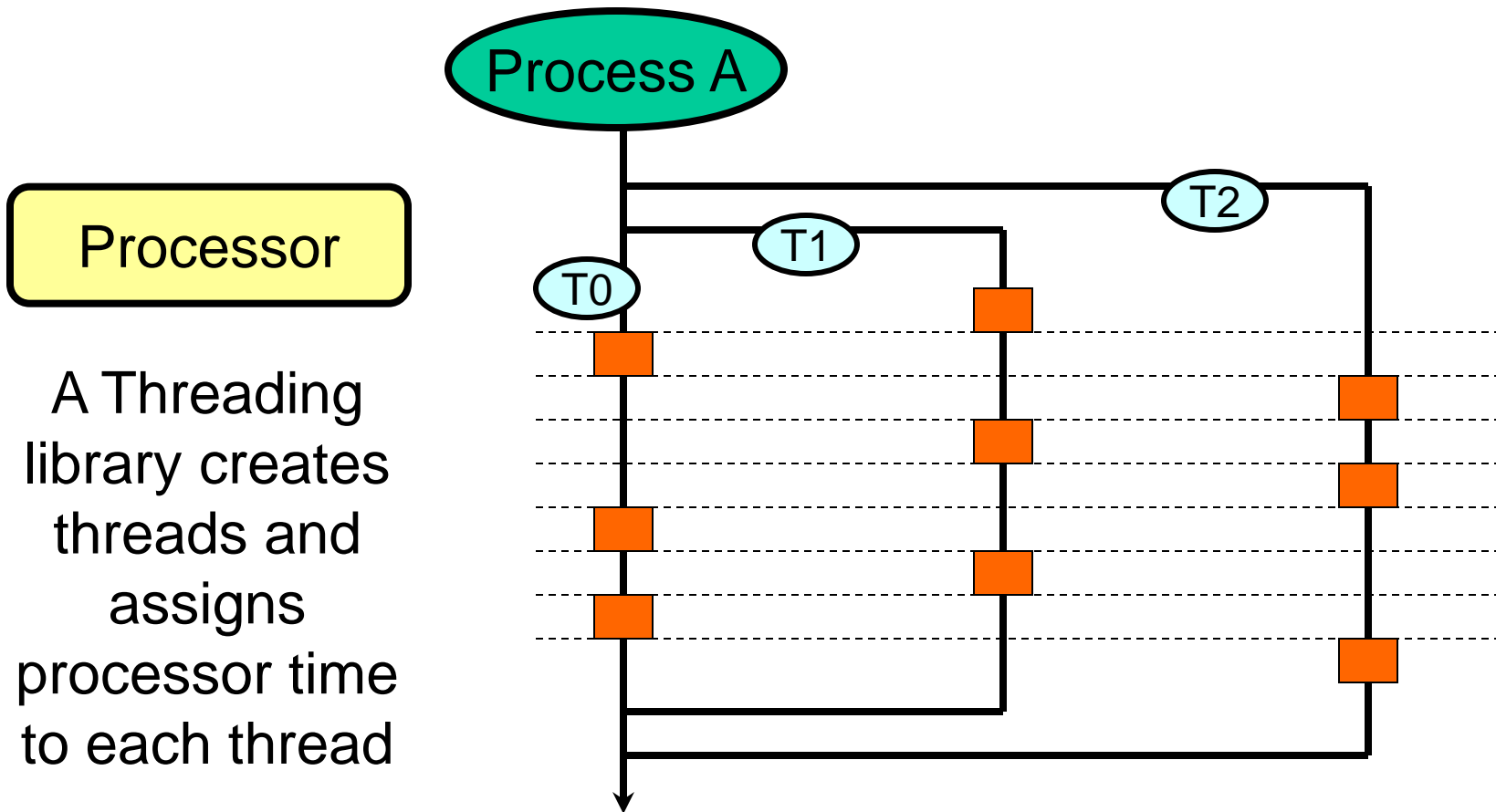
- Do you remember, single-processing, multi-processing, single-processing with multithreading, multi-processing with multithreading modes we discussed earlier in the course?



- Processor time will be shared among concurrently running processes.

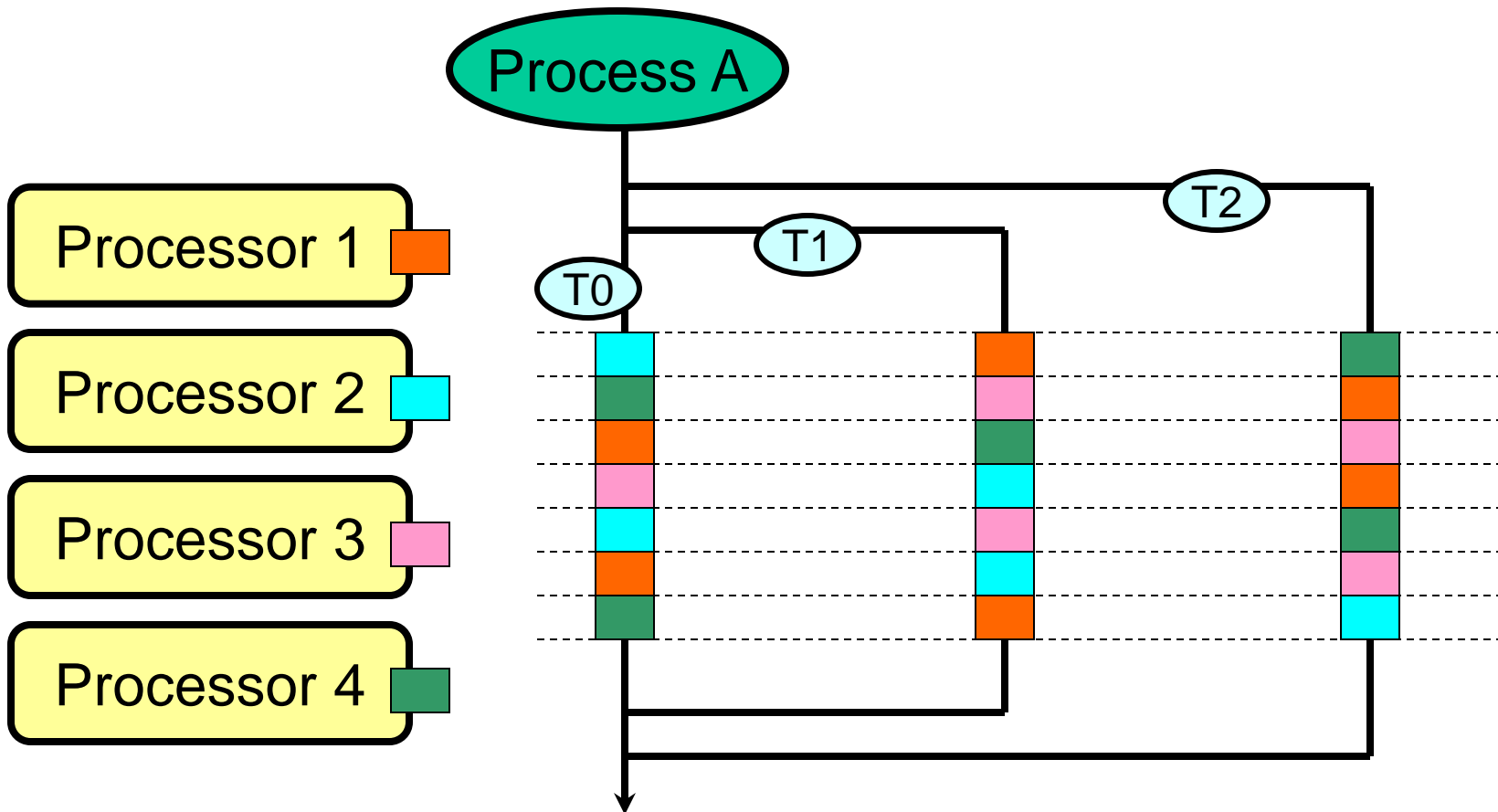
The Multi-Threading Mode

- Processor time assigned to Process A will be shred among its threads.



Multi-Threading in **Multi-Processors** Mode

- Processors time assigned to Process A will be shared among its threads.



What is a Thread?

- A thread **is a code section in a process** that can execute concurrently with other sections in the parent **process** (multithreading).
- Thread/Lightweight Process/Execution Context **is a single sequential flow of control within a process.**
- A thread likes a sequential program, it has:
 - A beginning, a sequence of execution, and an end.
 - Has a single point of execution, at any given time.
- A thread cannot live on its own, it must live within a process.
- **Each process has its own memory space**, but threads share memory space of the parent process.
- Therefore processes are “**heavyweight**” while threads are “**lightweight**”.
- A Browser is a multi-threaded program. The Browser can perform multiple simultaneous tasks:
 - Fetch the source code of the main page,
 - Download and play a media file,
 - Activate separate threads for other parts of the page,
 - Each thread sets up a separate connection with the server:
 - Uses blocking calls
 - Each part (an image) fetched separately and in parallel.

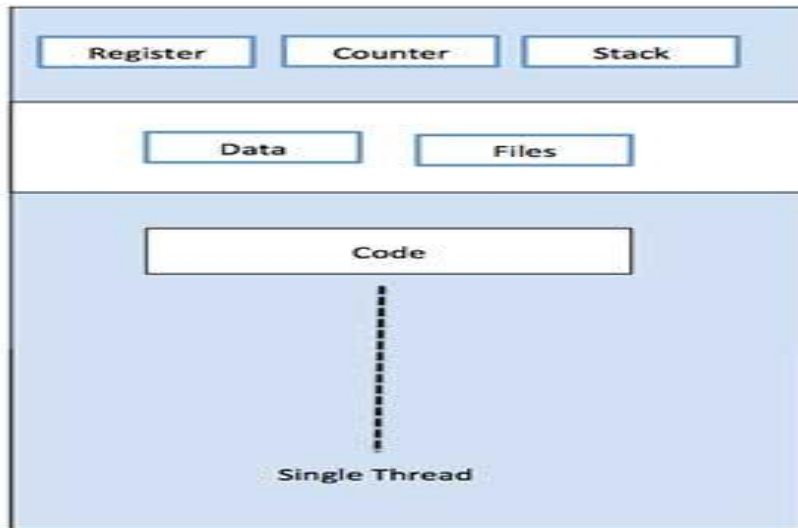
Thread's Control Block

- Thread **C**ontrol **B**lock (**TCB**) is a data structure contains thread's information:
 - Thread's State (ready, or running, or blocked),
 - Starting Address (Program Counter),
 - Registers, Execution Stack.
- Parent's process control block contains everything else (e.g. process id, open files, code segment, global data, etc.)
- TCB is a subset of the parent's process control block (PCB).
- The parent's (PCB) is the union of all TCBs of its children threads.
 - When a child thread alters non-private data, all other threads of the process can see this.
 - Threads communicate via shared variables.
 - A file opened by one thread is available to other threads of the parent process.

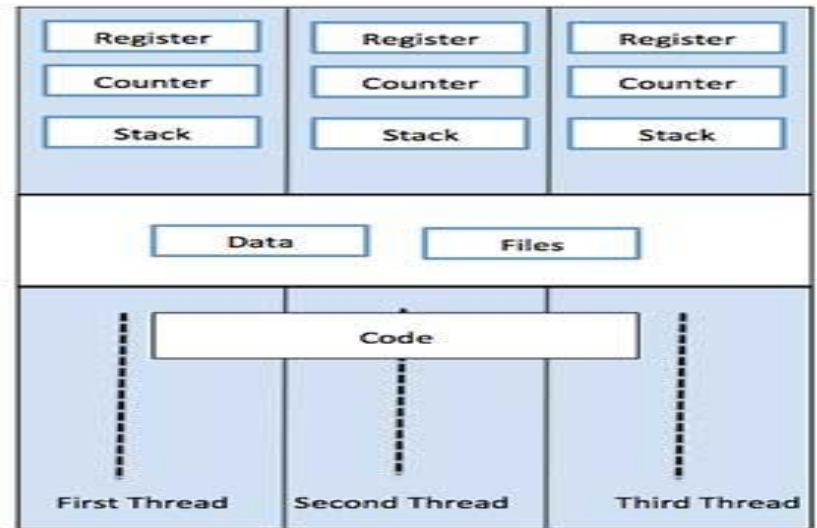
What does a Thread share with the Parent Process?

- Multiple threads within a single process share:
 - Process ID (PID)
 - Address space
 - Code section
 - Global data section
 - Open file descriptors
 - Signals and signal handlers
 - Current working directory
 - User and group ID
- Each thread has its own
 - Thread ID (TID)
 - Set of registers, including Program Counter and Stack Pointer
 - Stack for local variables and return addresses
 - Signal mask

Single Threaded & Multithreaded Process Models



Single Process P with single thread



Single Process P with three threads

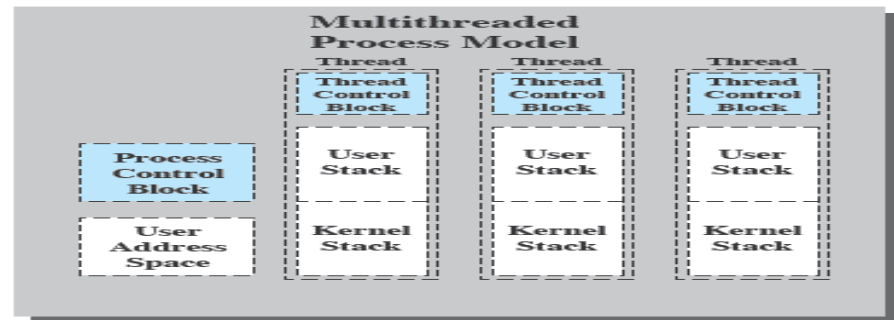
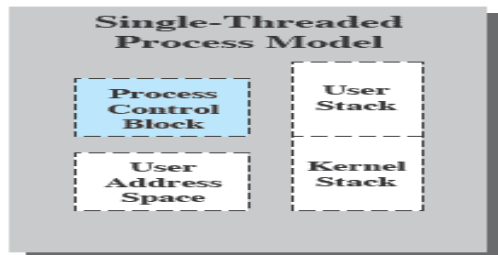


Figure 4.2 Single Threaded and Multithreaded Process Models

Thread Control Block contains a register image, thread priority and thread state information

Processes vs. Threads

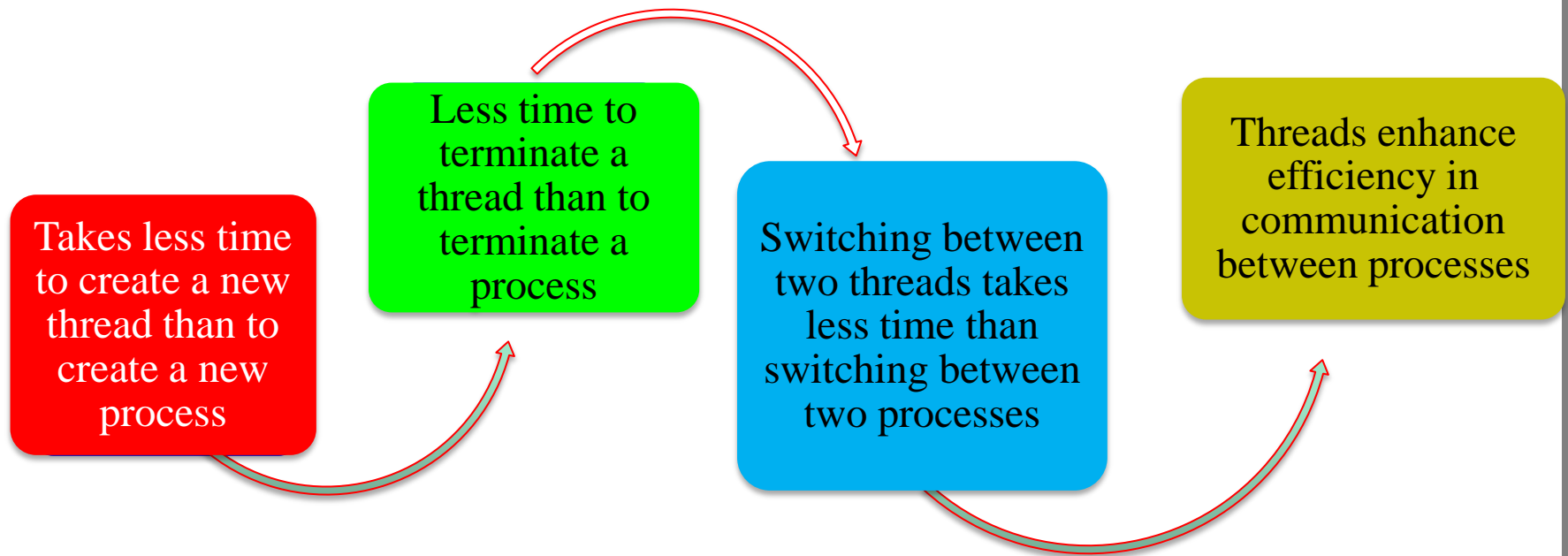
Which of the following belongs to the process and which to the thread?

Program code:	Process
Local or temporary data:	Thread
Global data:	Process
Allocated resources:	Process
Execution stack:	Thread
Memory management info:	Process
Program counter:	Thread
Parent identification:	Process
Thread state:	Thread
Registers:	Thread

Threads vs. Processes

- If two processes want to access shared data structures, the OS must be involved.
 - OS involvement requires system calls, mode switches, extra execution time.
- Creating new processes, switching between processes, etc. is slower than performing same operations on threads.
- Two threads of the same process can share global data automatically without the OS involvement (same as two functions in a single process).
- Compared to using several processes, **threads are more economical way to manage an application with parallel activities.**

Benefits of Threads



Benefits of Multi-Threads per Processes

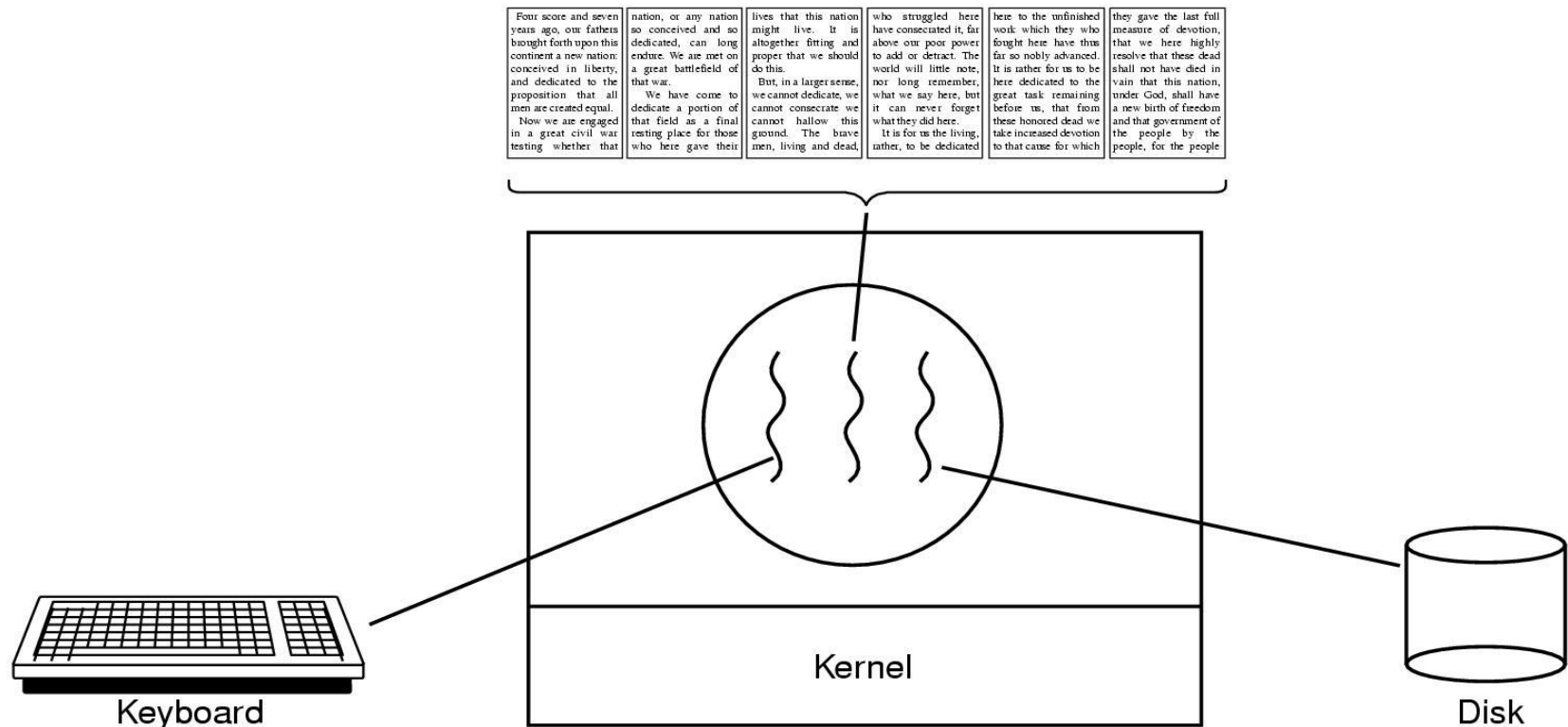
- **Responsiveness:** Multithreading allow the process to continue running even if part of it (a thread) is blocked or is performing a lengthy operation. To enable cancellation of separable tasks.
- **Speed up the Execution:** On a multiprocessor machine, multiple kernel level threads from the same process can execute simultaneously.
- **Resource sharing:** Threads share the resources and memory of the process to which they belong. This allows an application to have several threads within the same address space.
- **Economy:** Allocating memory and resources for each process is costly, while threads within the same process share memory and files.
- **Supports of asynchronous processing:** Independent parts of an application that do not need to run in sequence can be threaded,
 - i.e. auto-saving of RAM into disk. A thread schedules itself to come-alive every 1 minute to do this saving concurrently with main processing.

Benefits of Multithreading

- Multithreaded programs appear to do more than one thing at a time (same ideas as multiprocessing, but within a single program).
 - While you are browsing a web page,
 - Download several files in the background,
 - Play a music file.
- Multithreading is essential for some applications (i.e. games, graphics, ...)
 - One thread does the animation,
 - Second thread responds to user inputs,
 - Third thread is downloading an image.
- From the management point of view:
 - Takes less time to create a new thread than a process
 - Less time to terminate a thread than a process
 - Less time to switch between two threads within the same process
 - Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel.

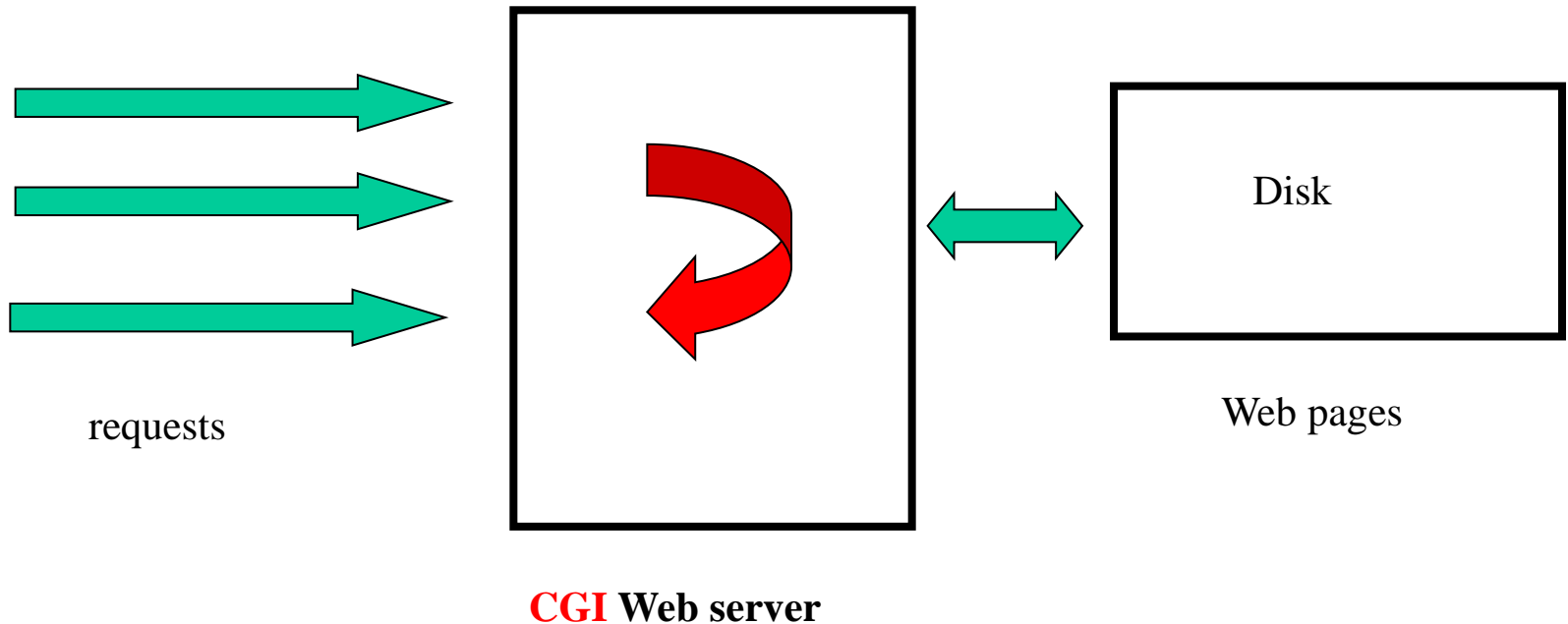
Example: Multi-Threaded Process

- **Word Processor with 3 Threads**
 - Thread 1: Interacts with user, and gets the pressed characters.
 - Thread 2: Reformats the text (in background).
 - Thread 3: Periodically backups the file into the HDD.



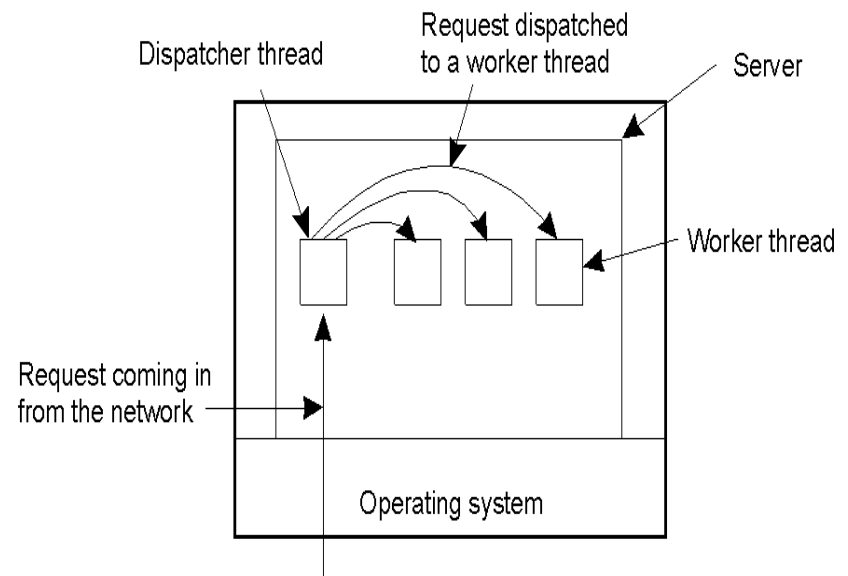
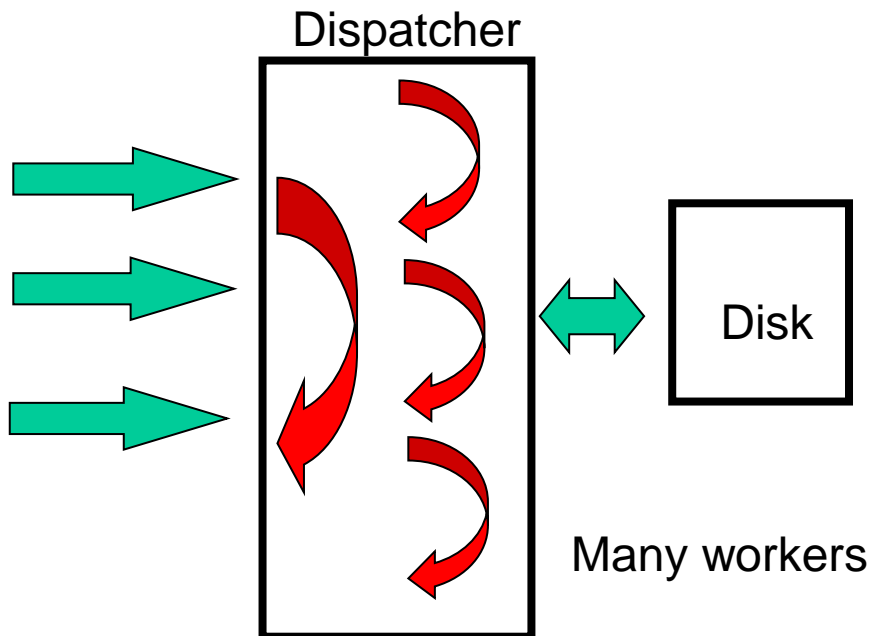
Example: Single Threaded Web Server

- If we have a single threaded server like this:
 - How long does the client request wait?
 - Is it going to support the responsiveness goal of the OS?
 - Is it going to be productive and maximize the throughput?



Example: Multi Threaded Web Server

- Multi-threaded Web server:
 - Is capable of processing multiple simultaneous service requests in parallel which increases the throughput.
 - Gets requests, sends web pages back quickly, **be responsive**.
 - Keep popular pages in cache memory, i.e. some pages much more popular than others.



Programming Assignment

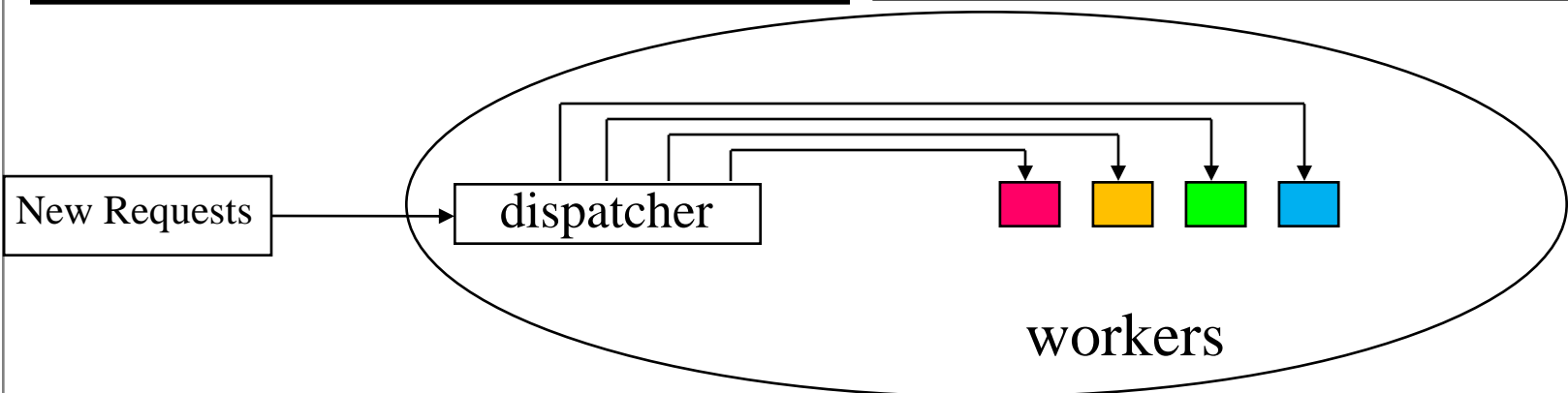
- In the lab, you need to code a Multi-Threaded program that will be able to process multiple simultaneous service requests in parallel.
 - We want to compare Multi-Threaded process with single threaded process performance.
 - See the effect of the number of threads on the response time.

Dispatcher

```
While (1) {  
    get_request(&req);  
    start_new_worker(req);  
}
```

Worker

```
Worker_thread(req) {  
    fetch_webpage(req,&page);  
    return_page(req, page);  
}
```



Other Examples of Multithreaded Programs

- Modern OS kernels
 - Deal with concurrent requests by mapping each user's request to a corresponding thread.
 - But no protection needed within kernel.
- Database Servers
 - Responsive access to shared data by many concurrent users.
- Network Servers
 - Responsive support to concurrent requests from network.
 - Multiple concurrent operations; File server, Web server, and airline reservation systems.
- Parallel Processing (More than one physical CPU)
 - Split program into multiple threads for parallelism.
- Embedded systems
 - Single Program that supports concurrent operations through multithreading.

Summary: **Threads** vs. Processes

- A thread has no data or code segments.
 - A thread cannot live on its own, it must live within a process.
 - There can be more than one thread in a process, the first thread calls main & has the process's stack.
 - Inexpensive creation
 - Inexpensive context switching.
 - If a thread dies, its stack is reclaimed.
 - While one thread is blocked and waiting, a second thread in the same task can run.
 - Multiple threaded processes use fewer resources.
- A process has code/data/heap & other segments.
 - There must be at least one thread in a process.
 - Threads within a process share code/data/heap, share I/O, but each has its own stack & registers.
 - Expensive creation
 - Expensive context switching.
 - If a process dies, its resources are reclaimed & all threads die.
 - If one process is blocked, then no other process can execute until the first process is unblocked.

OSs that Support Threads and Processes

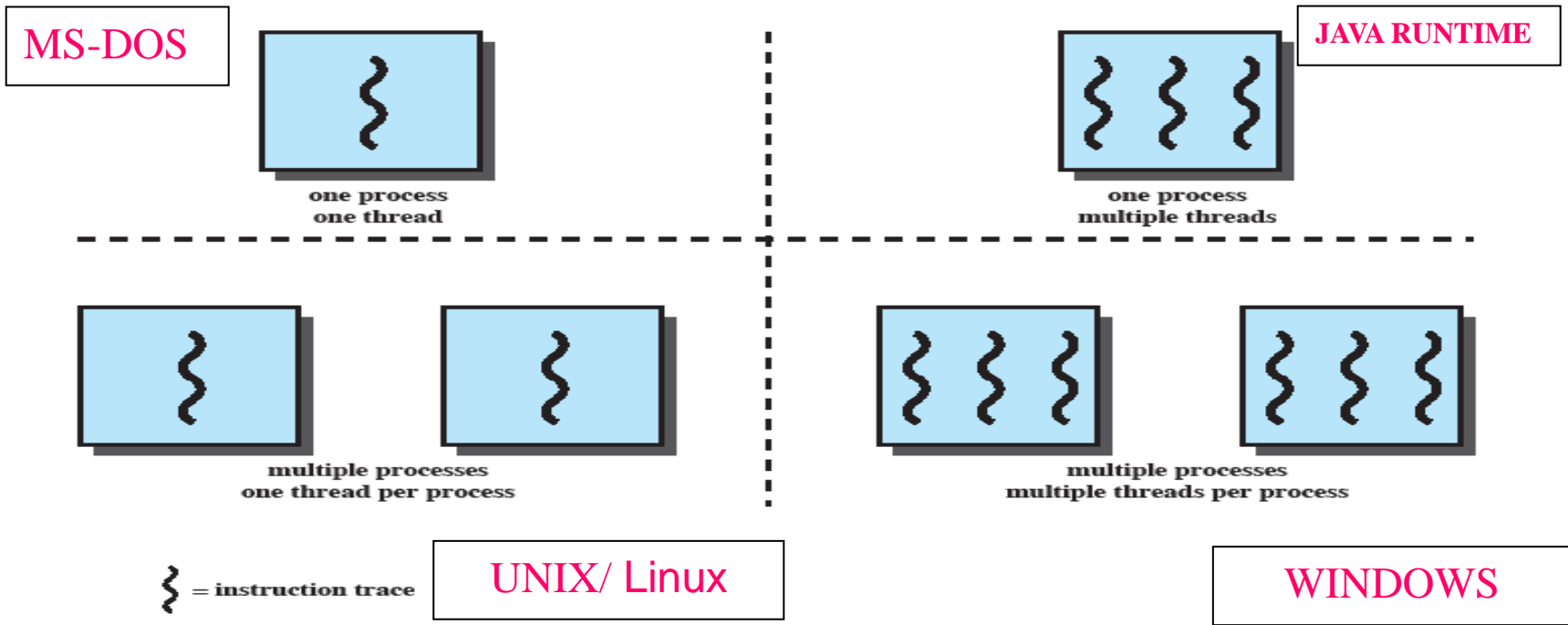
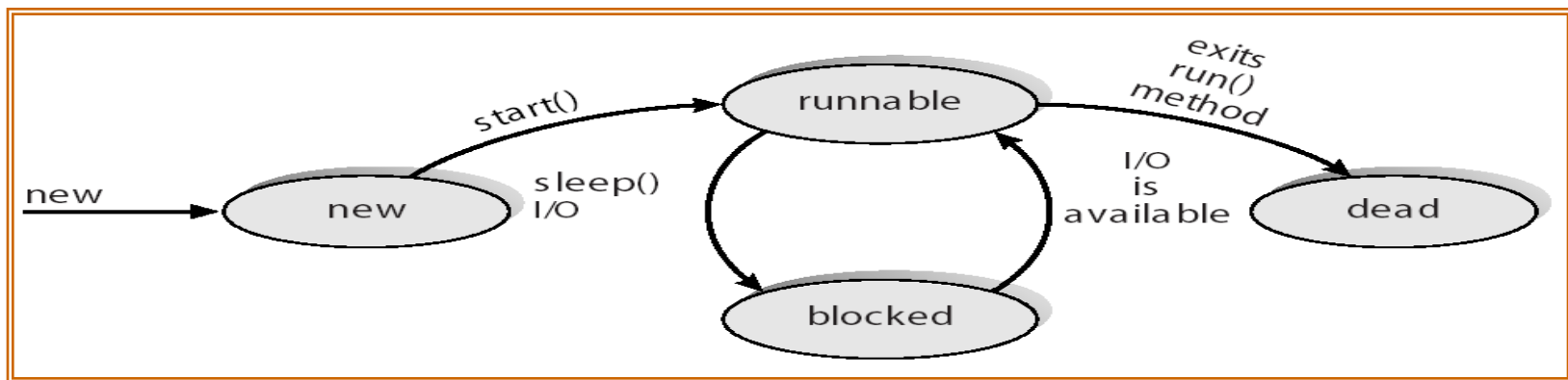


Figure 4.1 Threads and Processes [ANDE97]

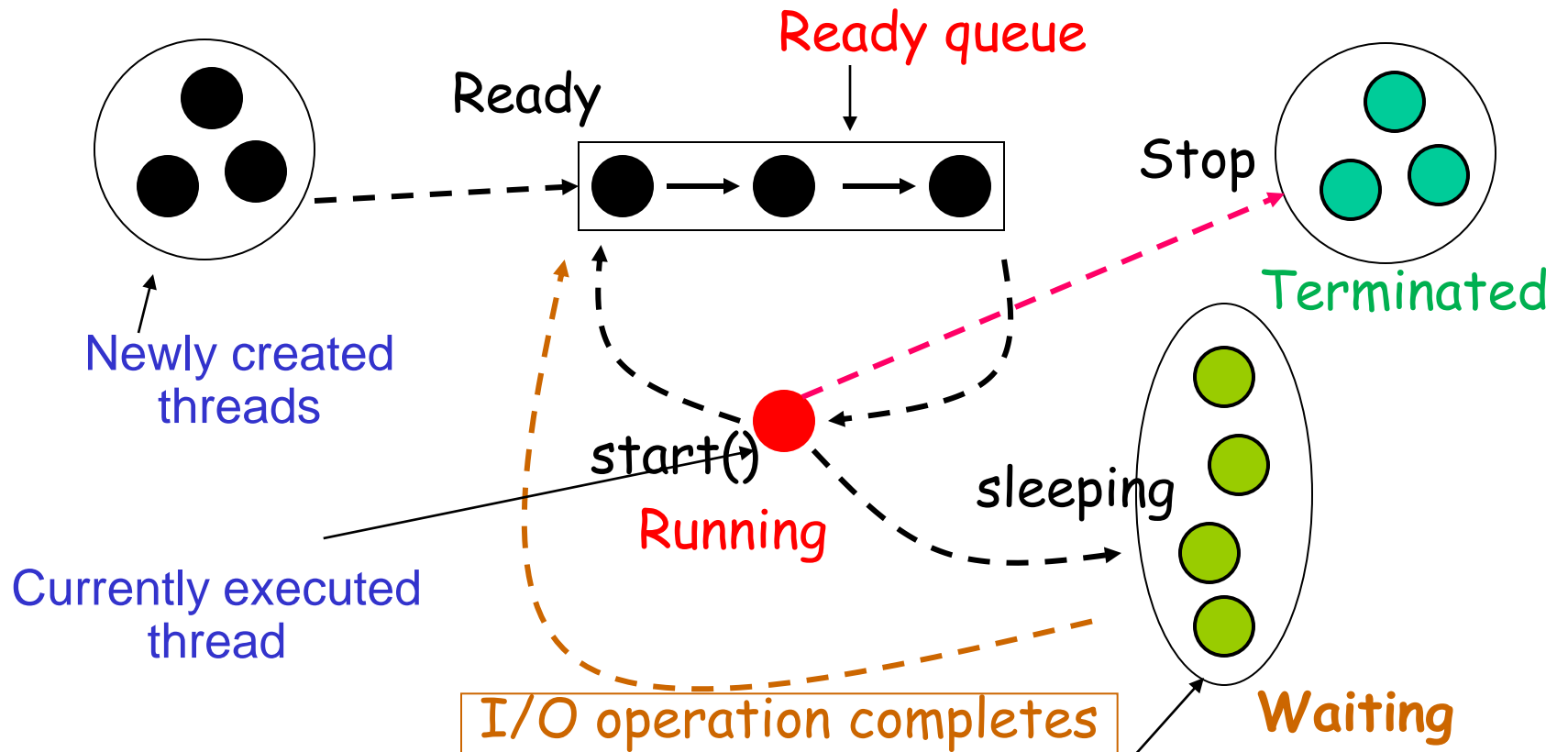
- MS-DOS supports a single user process and a single thread
- UNIX/ Linux supports multiple user processes but only one thread per process.
- A Java runtime environment supports one process with multiple threads
- Solaris, Windows family, OS/2 support multiple processes with multiple threads.

Threads Life Cycle

- Three key states: **new**, **running**, **waiting** [Blocked]
- Thread's States:
 - **New**: a thread is created by a process/thread using a command often called **spawn/fork/start**.
 - **Running**: doing the assigned job.
 - **Blocked**: when a thread needs to wait for an I/O event or asked to sleep for some time.
 - **Dead**: when a thread completes its job.
- Termination of a process, terminates all threads within the process **in windows**.
- They have no **suspend** state because all threads within the same process share the same address space. **Indeed**: suspending a single thread involves suspending all threads of the same process **if they are of type user level threads**.



Threads States



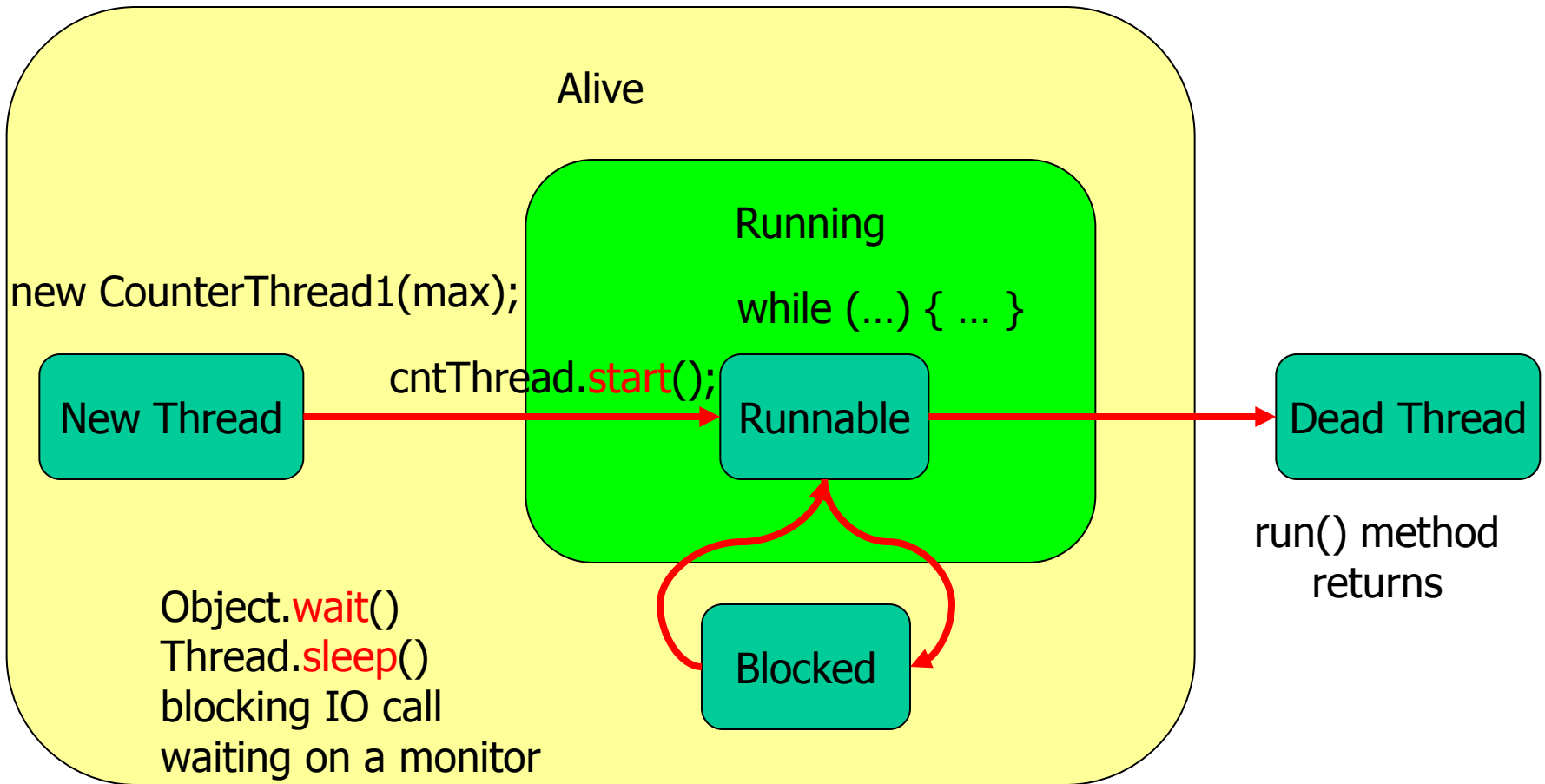
- Waiting for I/O operation to complete
- Waiting to be notified
- Sleeping
- Waiting to enter a synchronized section

Java's Thread Life Cycle

- **Newly Created State**
 - Thread `myThread` = `new` `MyThreadClass()`;
- **Runnable State**
 - After calling `start()` in which `run()` is executed `myThread.start()`;
 - Logically it is running, but physically it can be in one of the two states
 - **Running State** (Physically running on CPU)
 - **Ready State** (Waiting for its turn in the ready queue)
- **Blocked State**
 - Enters to **Blocked** State if the thread ...
 - Calls an objects `wait()` method
 - Calls `sleep()` method
 - Waits for I/O
 - Exits from **Blocked** State if the thread ...
 - Is waiting for an object, and on that object `notify()` or `notifyAll()` is called.
 - Is sleeping and the sleeping time elapsed.
 - Is waiting for I/O, and I/O is completed.
- **Dead State**
 - When finishes the `run()` process
 - `myThread.stop()`;

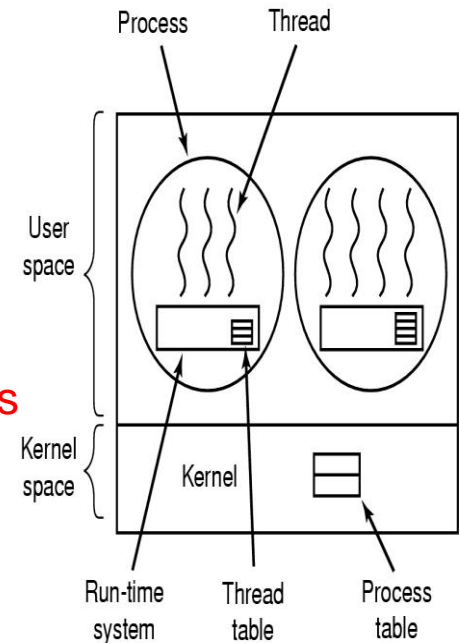
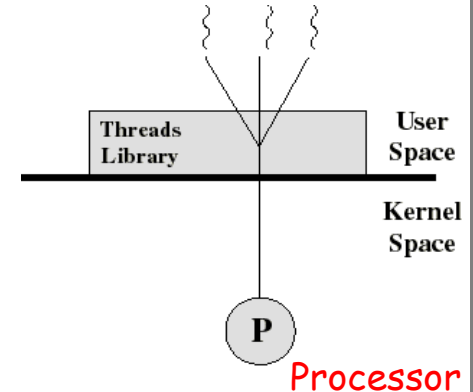
```
public void run() { int i = 0;
while (i < 100) { i++;
System.out.println ("i = " + i); }
}
```

Thread State Diagram from the Parent process point of view



User-Level Threads (ULT)

- User-level thread management done by threads library in the **user space**. The library provides support for thread creation, scheduling. There is no support from the OS kernel.
- **Threads scheduling is application specific.** The OS kernel is not aware of the existence of user's level threads.
- User's level Thread switching does not require kernel mode privileges (**no mode switch**).
- Blocking of any user's level thread blocks the entire process if the kernel is single threaded.
- When a user-level thread makes a system call (**e.g., reading a file from disk**), the OS moves the process to the waiting state and will not schedule it until the I/O has completed. **Thus, even if there are other user-level threads within that process, they have to wait, too.**
- User level threads are fast to create and manage.
- Examples user thread libraries:
 - Solaris 2 UI-threads, Mach C-Threads, pthreads, etc...



- The threads-support library in the user's space contains code for:
 - Creating and destroying threads.
 - Passing messages and data between threads.
 - Scheduling thread execution.
 - Saving and restoring thread contexts.

Kernel Activity for ULTs

- The kernel is not aware of user's level thread activity **but it is still managing the parent process activity.**
- When a user's level thread makes a system call, the whole process will be blocked.
- But for the thread library, that thread is still in the running state.
- **So thread states are independent of process states.**

Advantages and Disadvantages of ULT

- Advantages

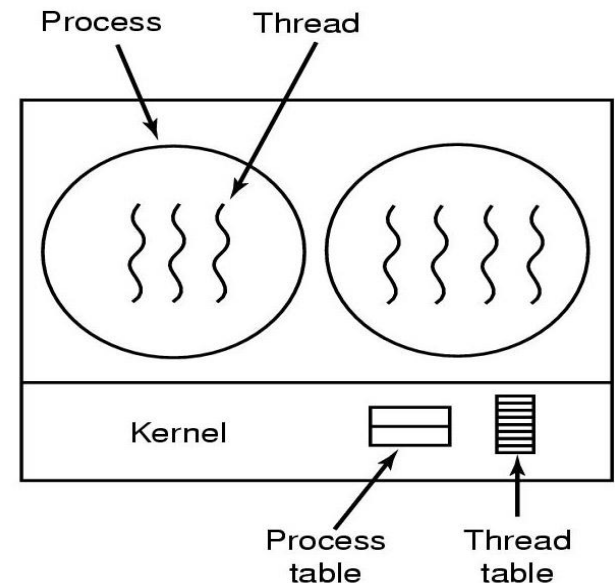
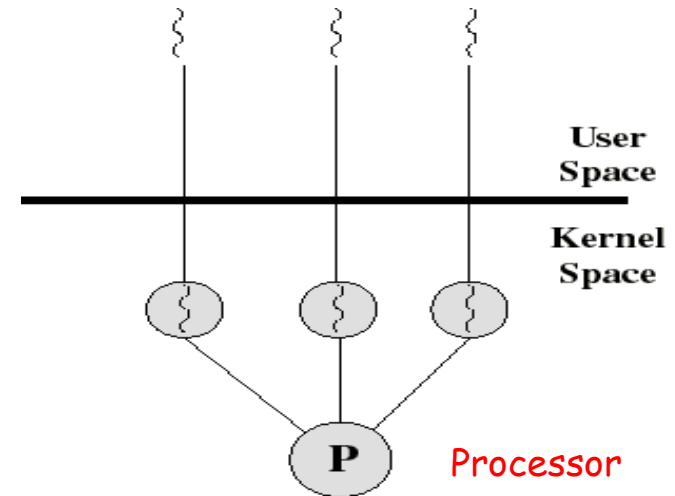
- Thread switching does not involve the OS kernel:
no mode switching
- Scheduling can be application specific:
choose the best algorithm.
- ULTs can run on any OS. Only needs a thread library to be installed
(more Portable)

- Disadvantages

- If one ULT makes a system call, the OS kernel blocks the process.
So all threads within the process will be blocked.
- The kernel can only assign processes to processors. **Two threads within the same process cannot run simultaneously on two processors.** (less concurrency and parallelization)

Kernel-Level Threads (KLT)

- Supported and managed by the OS Kernel (slower to create).
- No thread library but an API (i.e. system calls) to the OS kernel thread facility.
- OS Kernel maintains the the process and the threads.
- Switching between threads requires the OS kernel involvement.
- Scheduling on a thread basis (another thread can be scheduled in case of a system call by others).
- Examples OS support KLT:
 - Windows ...
 - Solaris
 - Tru64 UNIX
 - Linux



Advantages and Disadvantages of KLT

- Advantages

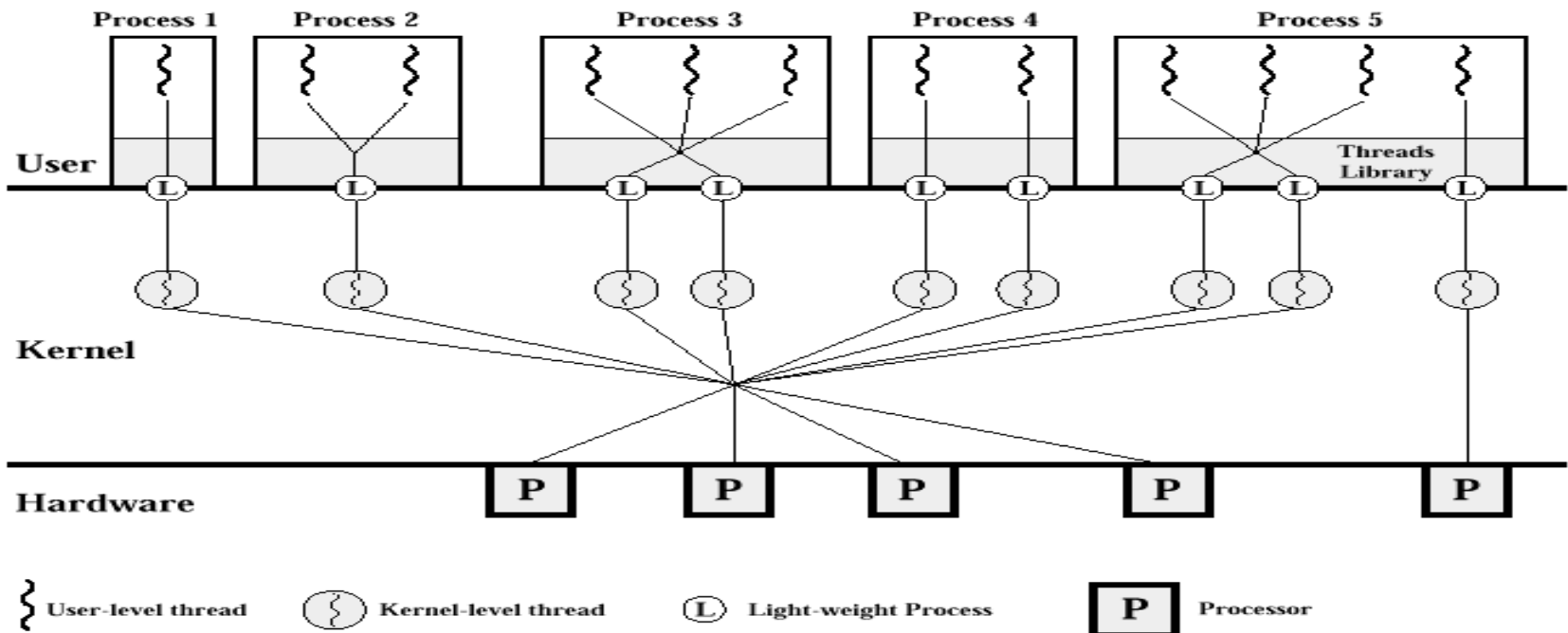
- The kernel can simultaneously schedule many threads of the same process on many processors (**good for multiprocessor environment**)
- Blocking is done on a thread level not on the process level.
- Kernel routines can be multithreaded.

- Disadvantages

- Thread switching within the same process involves the OS kernel.
- There are 2 mode switches per thread switch:
 - **User to kernel**
 - Kernel to user.
- This results in a significant slow down the performance.

Combined ULT/KLT Approaches

- To get the advantages of both threads type, modern OSs support the existence of both kernel and user level threads.
- Special type of processes called Lightweight processes (LWP) will be used to support the mapping of ULTs into KLTs.
- We will present now several ways of the mapping process.



Multithreading Models

- How do user's and kernel's threads map into each other?
- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One Model

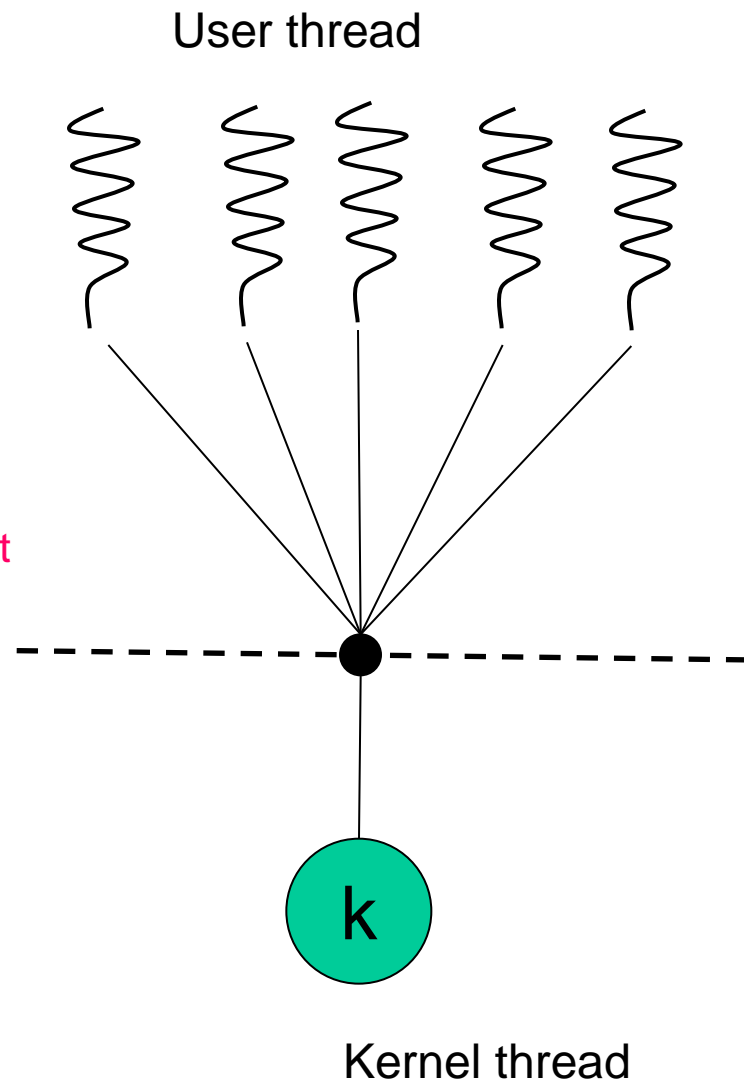
- Used on OSs that do not support multiple kernel's level threads.
- Many user's level threads mapped to a single kernel's level thread.
- Many-to-One allows a developer to create as many threads as s/he likes, but only one kernel thread can be scheduled at a time.

Advantages:

- Thread management is done in user space, so it is efficient.

Disadvantages

- The entire process will block if one thread makes a blocking system call.
- Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- Example: Solaris Green Threads work this way.



One-to-One Model

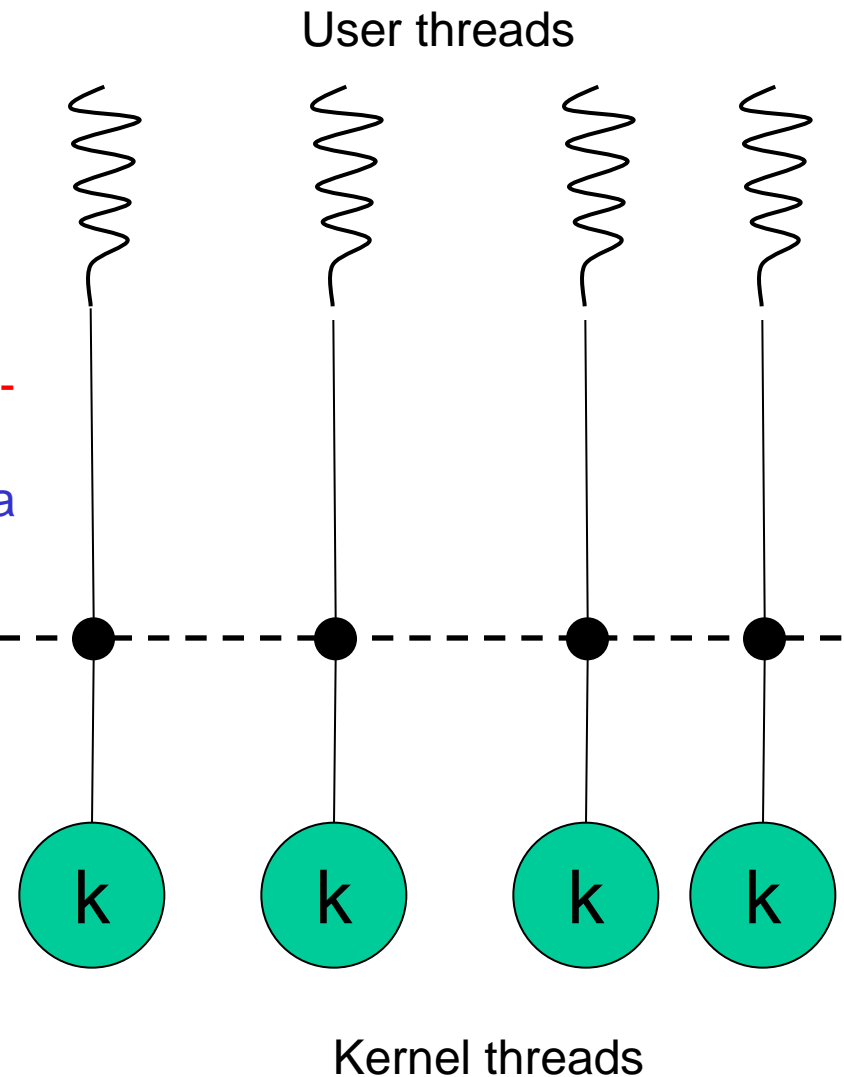
- Used on OSs that support multiple kernel's level threads.
- Each user's level thread maps to a kernel's level thread.
- **Examples:** Windows Family

Advantages:

- Provides more concurrency than **many-to-one** model by allowing another thread to run when one thread makes a blocking system call.
- It allows multiple threads to run in parallel on multiprocessors.

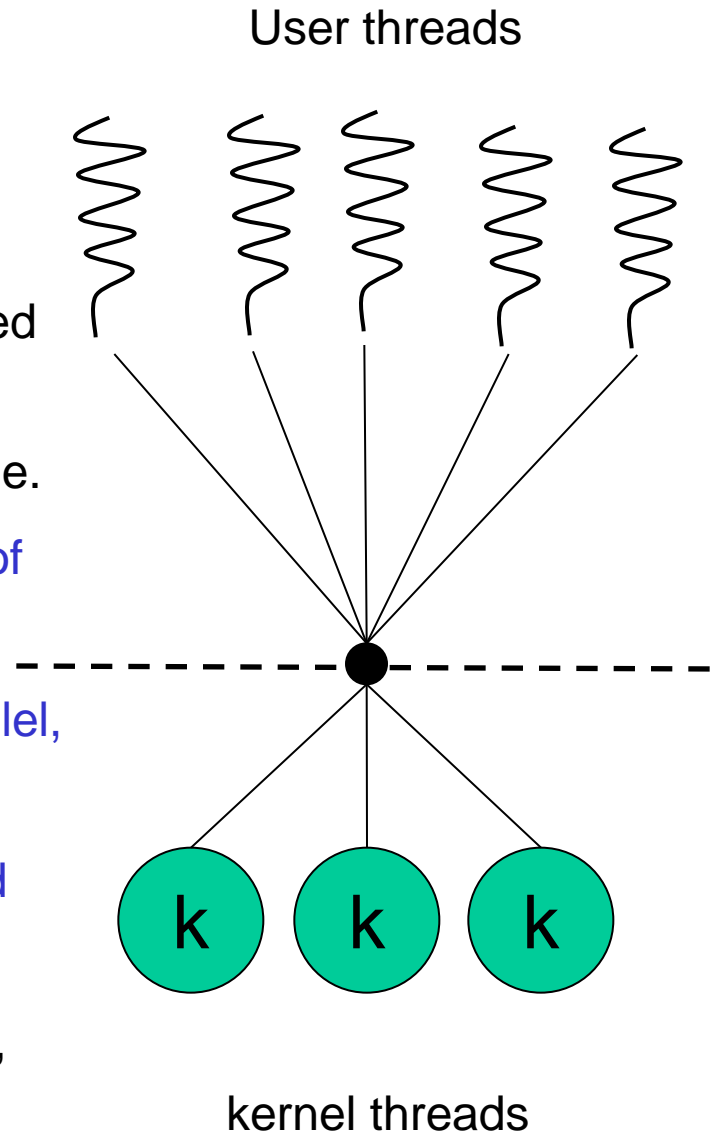
Disadvantage:

- Creating a user's level thread requires creating a corresponding kernel's level thread which can burden the performance of an application.



Many-to-Many Model

- Allows many user's level threads to be mapped to many kernel's level threads.
- The number of threads may be specific to either a particular application or a particular machine e.g. an application may be allocated more kernel threads on a multiprocessor machine than on a single processor machine.
- Allows the OS to create sufficient number of kernel threads.
- Many kernel's level threads can run in parallel,
- When a user level thread makes a system call, the kernel can schedule another thread for execution.
- Examples: Solaris 2, Windows Family, IRIX, HP-UX and Tru64.



Ch 4: Multi-Threading Programming

- Threads Definition
- Processing Modes in the OSs.
- Why does the OS support multithreading?
- Examples of Multithreaded Programs
- Thread's Control Block
- Benefits of Threads vs. Processes
- Thread's Life Cycle
- User's and Kernel's Level Threads
- Combining ULT and KLT Models (Sec. 2)
 - Many-to-One
 - One-to-One
 - Many-to-Many
- Threads Issues: i.e.
 - Threads scheduling, Priority Inversion/Inheritance in Threads scheduling,
 - Threads cancellation, Threads Pool,
 - Signal handling by Threads
- Threading in Different Platforms:
 - Windows, Solaris, Linux, Mac OS, etc.

Threads Issues

- **Threads Scheduling:** Which scheduling policy is used to schedule the kernel threads?
- **Thread cancellation/terminating:** When one thread returns a result, the others should be cancelled or not?
- **Threads pool:** How many Kernel Threads does the OS create?
- **Signal handling:** How does a parent process notify its threads that an event has occurred and which thread is going to respond?

Kernel Threads Scheduling

- Preemptive priority scheduling policy is used to schedule the kernel threads:
 - Each thread is given a global priority number.
 - Highest priority thread gets the CPU (preemption may occur, it means the CPU can be taken away from the thread if more higher priority thread is ready for running).
 - Round-robin based on the priority.
- Preemption is essential in OS to be responsive with real-time threads.
- Example 1: Single Processor (Two Threads)
 - Thread A (high priority), B (low priority), but A is waiting for a resource held by B.
 - When B releases the resource that A is waiting (sleeping) for.
 - B gives/preempts up CPU to allow A to run.
- Example 2: 2 Processors (Three Threads)
 - Thread A (high priority), B (medium priority), C (low priority), but B is waiting for a resource held by A.
 - A and C running on CPUs, B waiting on resource owned by A.
 - A releases the resource.
 - Signal C to give up the CPU so B can run.

Priority Inheritance/Inversion

- **Priority scheduling problem:** a high priority thread is blocked for a resource held by a low priority thread, which means it cannot get the CPU cycles to run while a medium priority thread is running!!!
- Example:
 - Thread **C** (**low priority**) holds resource **M**.
 - Thread **B** (**medium priority**) takes CPU.
 - Thread **A** (**high priority**) blocks on **M** (held by **C**).
 - **So the execution returns to B; that means B runs for a long time!**
 - **A** is locked out of CPU for a long time, even though it is the highest priority thread!
- Solution: **Priority inheritance/Inversion**
 - Since **A** blocks on **M** (waiting for **C**), **C** gets (**inherits**) **A**'s priority.
 - **C** will do its job and releases **M** then **A** gets its highest priority back
 - **A** can run now.

Threads Cancellation

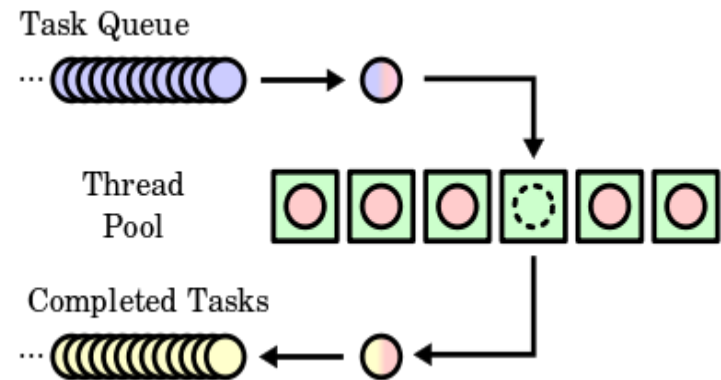
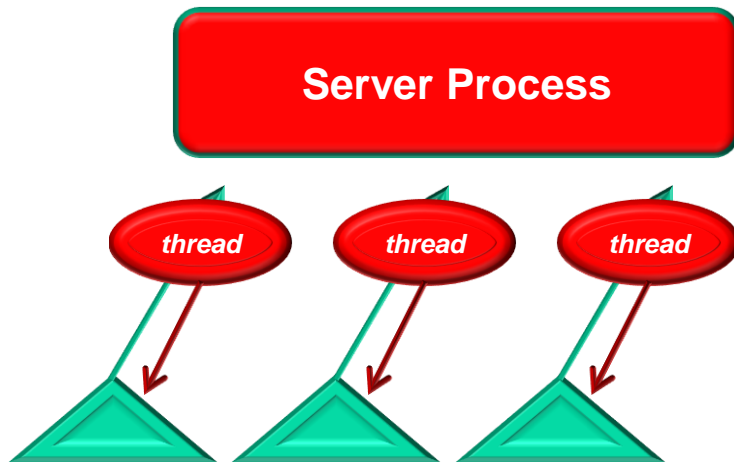
- **Cancellation:** means terminating a target thread before finishing its job.
- **Cancellation of a target thread may occur in 2 different scenarios:**
 - **Asynchronous/Unsafe cancellation:** terminates the target thread immediately. (windows platform supports safe and unsafe cancelation)
 - **Safe cancellation:** allows the target thread periodically to check if it should be cancelled or not, If yes terminate itself normally.
- In some OSs, termination of a process terminates all threads within the process. (Unix/Linux platform supports safe cancelation)
- Think about the following scenarios:
 1. Two threads searching a DB and one thread returns the result, the remaining might be canceled without causing any troubles (safe cancellation).
 2. When a user presses the stop button in the browser process then the thread loading the page is canceled (causes a problem (unsafe cancellation)).

Difficulties with **Asynchronous/Unsafe** Cancellation

- Difficulty with asynchronous/unsafe cancellation:
 - Canceling the thread while it is in the middle of updating data shared with other threads.
- Canceling a thread asynchronously
 - May cause inconsistency of the global variable's values.
 - May not free a necessary system-wide resources.
- Global variable, i.e. if **counter = 0** is a shared global variable.
- Thread 1 does increment **counter++** without updating the global value.
- Thread 2 does decrement it **counter--** // “at the same time”
- What is the order of **counter's** values ?
 - **0 : 1 : 0?**
 - **0 : -1 : 0?**
- Shared resources, i.e. a file is shared between two threads
 - One thread closed the file while the other one is reading from it.

Threads Pool

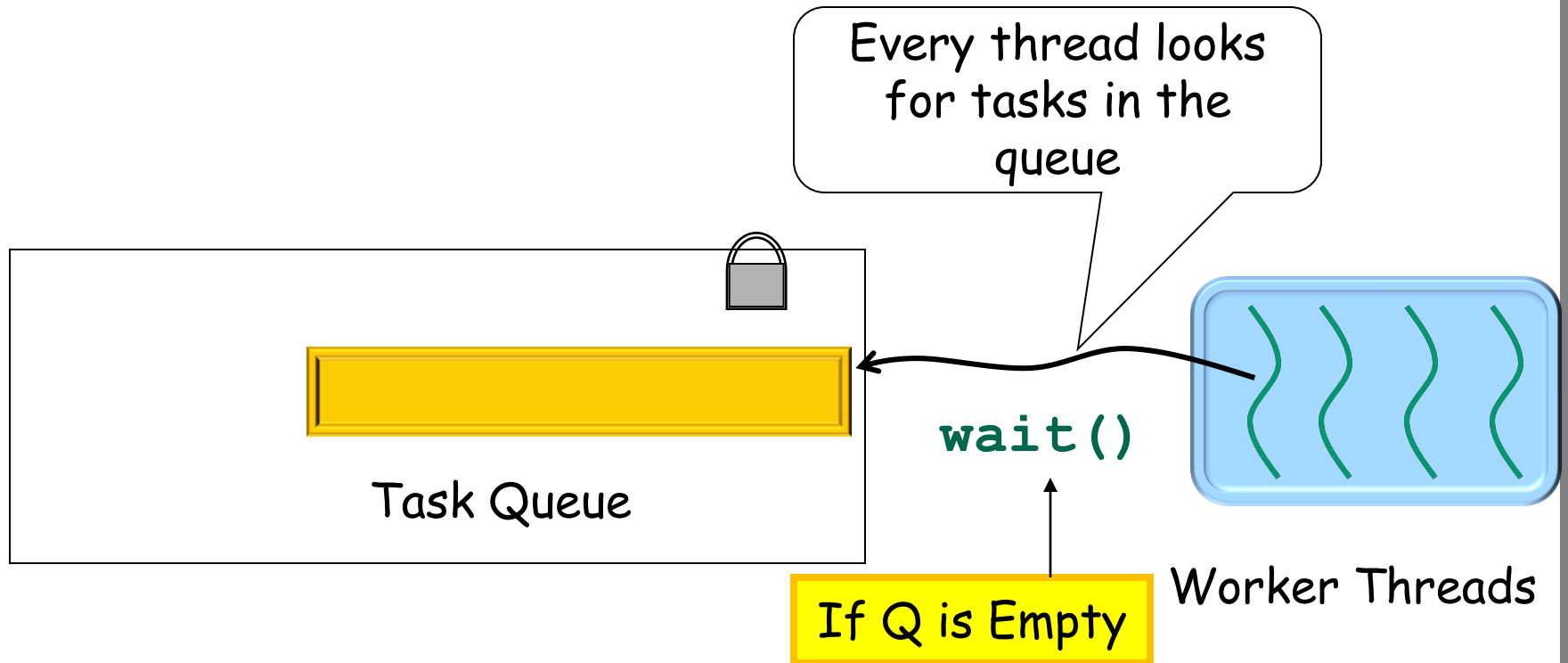
- The server process creates a number of threads at the process start up and places them in a pool where they wait for work or can be used in **the many to many mapping mode**.
 - When a server receives a request, assigns it to a thread from the pool.
 - Once a thread finishes its service, it returns to the pool and waits for a work again.
 - That means, no need to create a new thread for every client request, **it can be taken from the pool quickly**.



Threads Pool

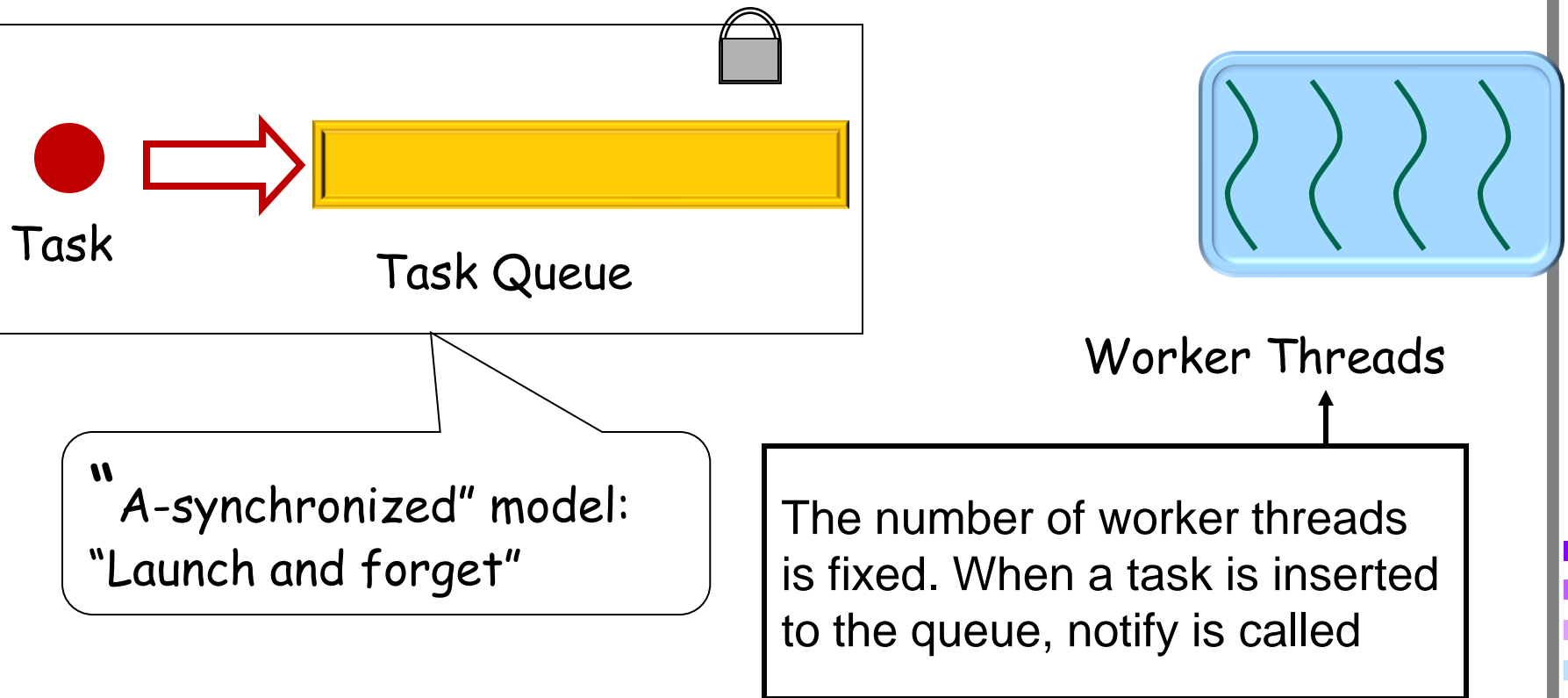
- If the pool is empty, the server waits until one becomes free.
- The server process dynamically adjusts the **number of threads** in the pool [optimize memory use] based upon factors such as:
 - The number of CPUs in the system,
 - The amount of physical memory,
 - Expected number of client requests.
- **Advantages:**
 - Slightly faster to serve a request with an existing thread than creating a new thread. Avoiding the overhead of creating a new thread.
 - Thread pools improve resource utilization
 - Allows the number of threads in the pool to be dynamic.
- **Disadvantages:**
 - Creating too many threads in one machine can cause the system to run out of memory and even crash.

Threads Pool Implementation

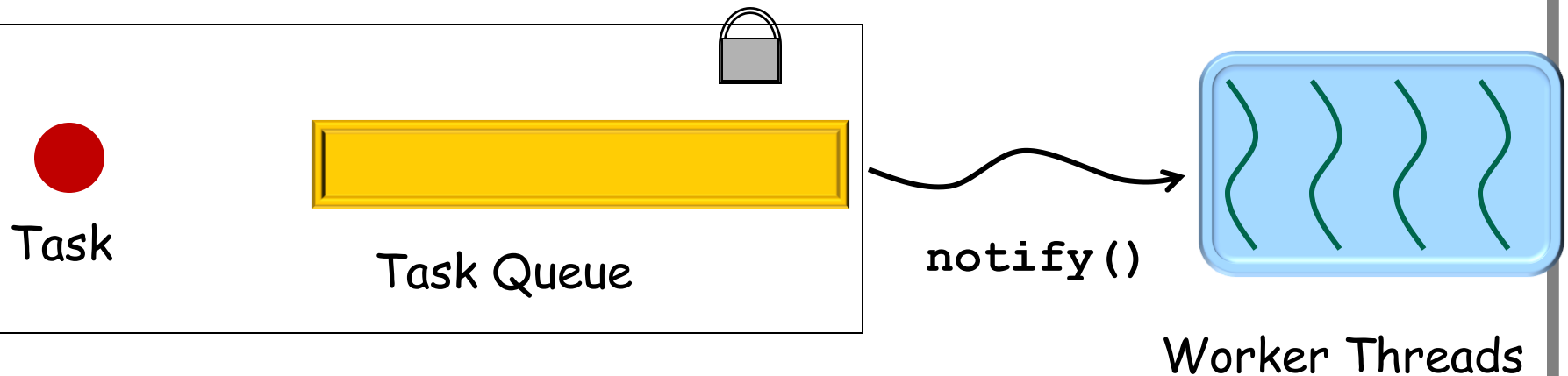


All the worker threads wait for tasks

Threads Pool Implementation



Threads Pool Implementation



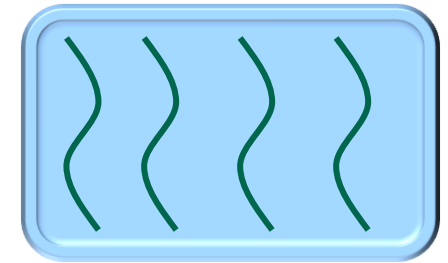
The number of worker threads is fixed. When a task is inserted to the queue, notify is called

Threads Pool Implementation

The task will be assigned to a thread in the pool



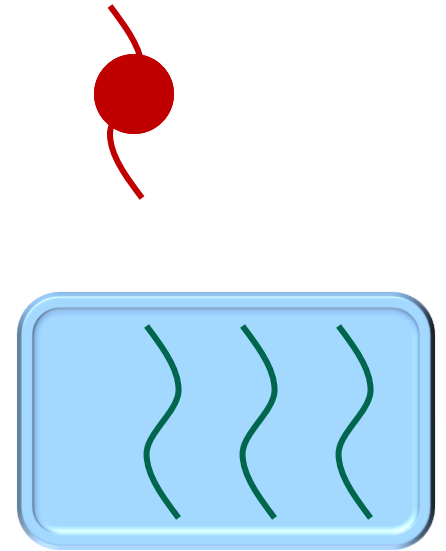
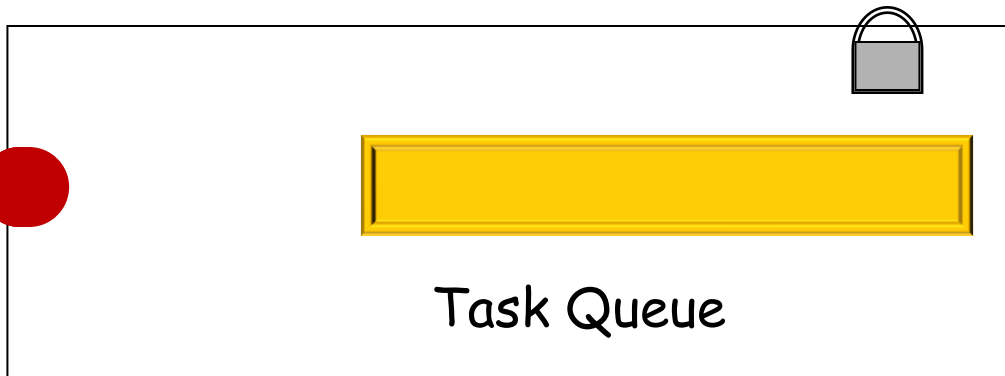
Task Queue



Worker Threads

Threads Pool Implementation

The task is executed by the thread



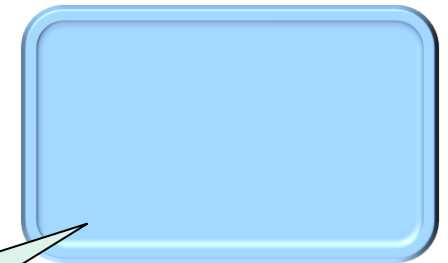
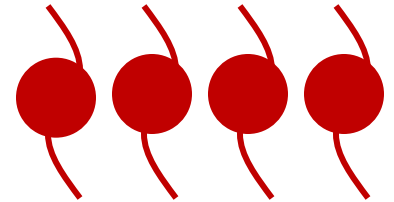
The remaining tasks are executed by the other threads

Threads Pool Implementation

When a task ends, the thread is released



Task Queue

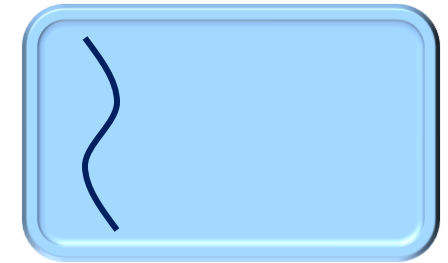
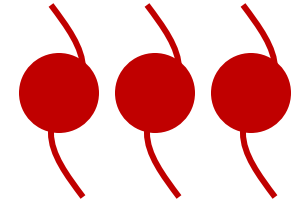
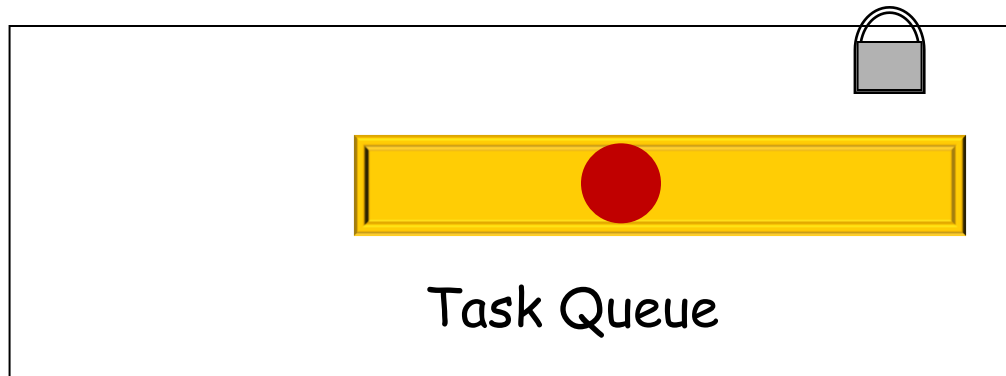


Worker Threads

While the Q is not empty, take the task from the Q and run it (if the Q was empty, wait() would have been called)

Threads Pool Implementation

A new task is executed by the released thread



Worker Threads

Signal Handling

- Signals are used in OS to notify a process/thread that a particular event has occurred.
- **All signals follow the same pattern:**
 - The signal is generated due to the occurrence of a particular event.
 - The generated signal is delivered to a thread or a process.
 - Once delivered, the signal must be handled.
- A signal may be received either **synchronously** or **asynchronously**:
 - Depending upon the source and the reason for the event being signaled.
- **Asynchronous signal:** The process/thread **does not know ahead of time** exactly when a signal will occur. i.e. a running program performs **illegal memory access or division by zero**.
- **Synchronous signal:** The process/thread **knows ahead of time** exactly when a signal will occur, i.e. expiration of assigned CPU time.

Signal Handling

- Signals can be sent by:
 - The OS kernel to a process/thread.
 - One process to another process.
 - A process to its threads.
- Signals may be handled by one of two possible handlers:
 - A default signal handler.
 - A user-defined signal handler [overriding the default one]
- When a process/thread receives a signal, it may perform one of the following:
 - Ignores the signal.
 - Performs the default operation.
 - Catches the signal (perform the user defined operation).

Signal Delivering

- In single-threaded programs
 - Straightforward: deliver the signal to the thread.
- In multiple-threaded programs
 - Deliver the signal to every thread in the process. or
 - Deliver the signal to certain threads in the process. or
 - Assign a specific thread to receive all signals for the process.
- In Windows Os for example:
 - Windows OS does not explicitly provide support for signals, but it emulates the signals using Asynchronous Procedure Calls (APCs).
 - APC is straightforward and is delivered to a particular thread in that process.
 - The APC facility allows a user thread to specify the thread that is to be called when the user thread receives notification for a particular event.

Java Threads

- Java threads are implemented by the JVM but their behavior is heavily influenced by the underlying OS and its characteristics.
- They do not fall under the category of either ULT or KLTs.
- The actual scheduling policy is OS-dependent, and determined together by the host OS and the JVM implementation.
- Java offers concurrency mechanisms as a built-in part of the language:
 - Built-in class Thread, with the run method as its "main"
 - **Synchronized** methods, and **synchronized** code blocks
 - Monitor locks and condition (wait) queues
 - Thread priorities
- Green threads exist only at the user-level and are not mapped to multiple kernel threads by the operating system.
- "Native threads" are the threads that are provided by the native OS.
- Native threads can realize the performance enhancement from parallelism (multiple CPUs).
- Java is naturally multi-threaded and because of this the underlying OS implementation can make a substantial difference in the performance of your application.

Linux Threads

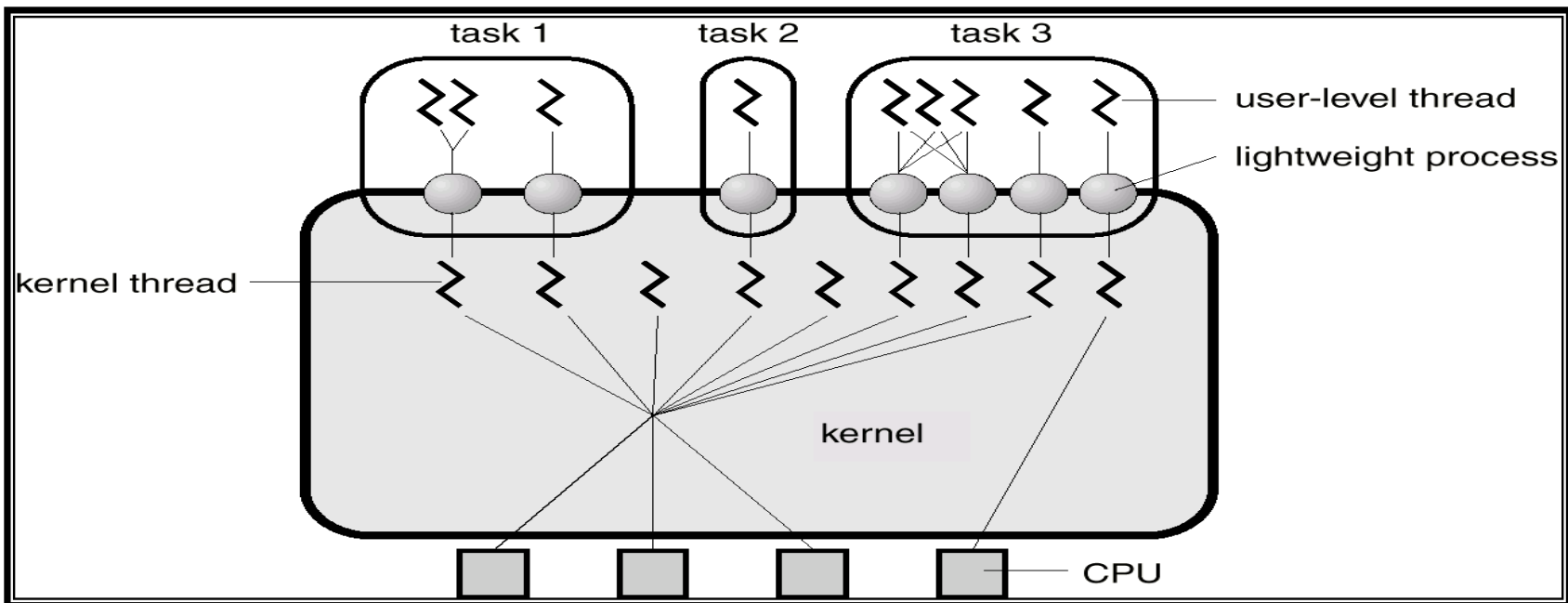
- From the **Linux** OS point of view, there is no concept of a thread.
- **Linux** implements all threads as standard processes.
 - Linux does not distinguish between processes and threads
- The **Linux** kernel does not provide any special scheduling semantics or data structures to represent threads.
- Instead, a thread is merely a process that shares certain resources with other processes.
- **Linux** uses the concept of **task** rather than threads and processes.
- Each thread has a unique **task_struct** and appears to the kernel as a normal process.
- Linux provides kernel-level **tasks**:
 - **Tasks** are created with the **clone() system** call and all scheduling is done in the kernel.
- **Clone()** allows a child **task** to share the address space of the parent **task**.
 - The flags provided to **clone()** command help specify the behavior of the new process and detail what resources the parent and child will share.
 - i.e. **clone_files**, **clone_newns** (share files, or name space)

Solaris 2 Threads

- Solaris 2 is a version of UNIX with support for threads at the kernel and user levels and real-time scheduling.
- It implements the Pthread API in addition to supporting user-level threads with library of API for creation and management.
- Process includes the user's address space, stack, and process control block
- User-level threads (threads library)
 - Invisible to the OS
 - Are the interface for application parallelism
- Kernel threads
 - The unit that can be dispatched on a processor and it's structures are maintain by the kernel
- Lightweight processes (LWP)
 - Each LWP supports one or more ULTs and maps to exactly one KLT
 - Each LWP is visible to the application

Solaris 2 Threads

1. It defines an intermediate level of threads between kernel and user levels called Light Weight Processes [LWP].
2. Each process contains at least one LWP
3. The thread library multiplexes user level threads on the pool of LWPs



Task 2 is equivalent to a pure KLT approach

We can specify a different degree of parallelism (Task 1 and 3)

Decomposition of user-level Active state

- When a ULT is active, it is associated to a LWP and thus to a KLT.
- Transitions among the LWP states is under the exclusive control of the OS kernel.
- A LWP can be in the following states:
 - **Running**: assigned to CPU = executing
 - **Blocked** because the KLT issued a blocking system call (but the ULT remains bound to that LWP and remains **active**)
 - **Runnable**: waiting to be dispatched to CPU
 - **Stopped**: e.g. waiting for synchronization event

Windows Family Threads

- Implements the Win32 API, it is the primary API for MS OS family, it uses the **one-to-one mapping**.
- Each thread contains
 - Thread id
 - Register set
 - Separate user and kernel stacks
 - Private data storage area used by dynamic Link Libraries (DLL).

The primary data structure of Windows thread includes:

- TEB [thread environment block], contains **thread identifier, user stack and thread local storage**.
- ETHREAD [executive thread block], contains **thread start address and pointer to the corresponding KTHREAD**.
- KTHREAD [kernel thread block], contains **scheduling and synchronization information and the kernel stack**.

Windows Family Threads

- Threads are scheduling using a priority-based preemptive scheduling using a dispatcher
- 32 priority levels
 - 1-15: Variable class
 - 16-31: Real time
 - 32: Dispatcher
 - Idle thread is executed if no other thread is ready
 - Interactive tasks can get up to 3 scheduling quantum over time sharing applications.

Pthreads

- Traditional Unix's are multi-tasking OSs. UNIX permits a user to run multiple processes with single thread per each simultaneously.
- Each process has its own address space, with its own copies of its variables, which are completely independent.
- This independence, while providing memory protection and therefore stability, causes problems when you want to have multiple processes working on the same task/problem.
- The cost of switching between multiple processes is relatively high.
- For these reasons, and others, threads or Light Weight Processes (LWP) can be very useful.
- Threads share a common address space, and are often scheduled internally in a process, thereby avoiding a lot of the inefficiencies of multiple processes.
- One very popular API for threading an application is **Pthreads**, also known as POSIX threads.
- The **Pthread library** describes general thread behavior, and the functions which control threads.
- **Libraries implementing Pthreads specification are restricted to Unix-based systems such as Solaris 2.**
- **Pthread library should be included**
- **Some Pthread attributes include:**
 - A thread has a priority for scheduling
 - Threads may use several scheduling methods, some of which use priority.
 - A thread may have local or global scope of contention
 - It may compete with all threads in the system for CPU time, or it may compete only with threads in the same task (process).

A Reminder for Major Exam 1

Time: 7:00 – 9:0 PM,

Date: On Saturday, Feb. 29th, 2020

Location: Building 24, Room: 120/121

Materials: Every thing we presented up to the end of Ch.4 “Threads”.

Try to prepare your self well



The End!!

Thank you

Any Questions?

