Course TÖL401G: Stýrikerfi / Operating Systems 4. Threads

Mainly based on slides and figures subject of copyright by Silberschatz, Galvin and Gagne

Chapter Objectives

- Identify the basic components of a thread, and contrast threads and processes.
- Describe the major benefits and significant challenges of designing multithreaded processes.
- Describe different multithreading models.
- Design multithreaded applications using the POSIX Pthreads and Java threading APIs.
- Have heard about implicit threading approaches.
- Know about issues of using threads.

Contents

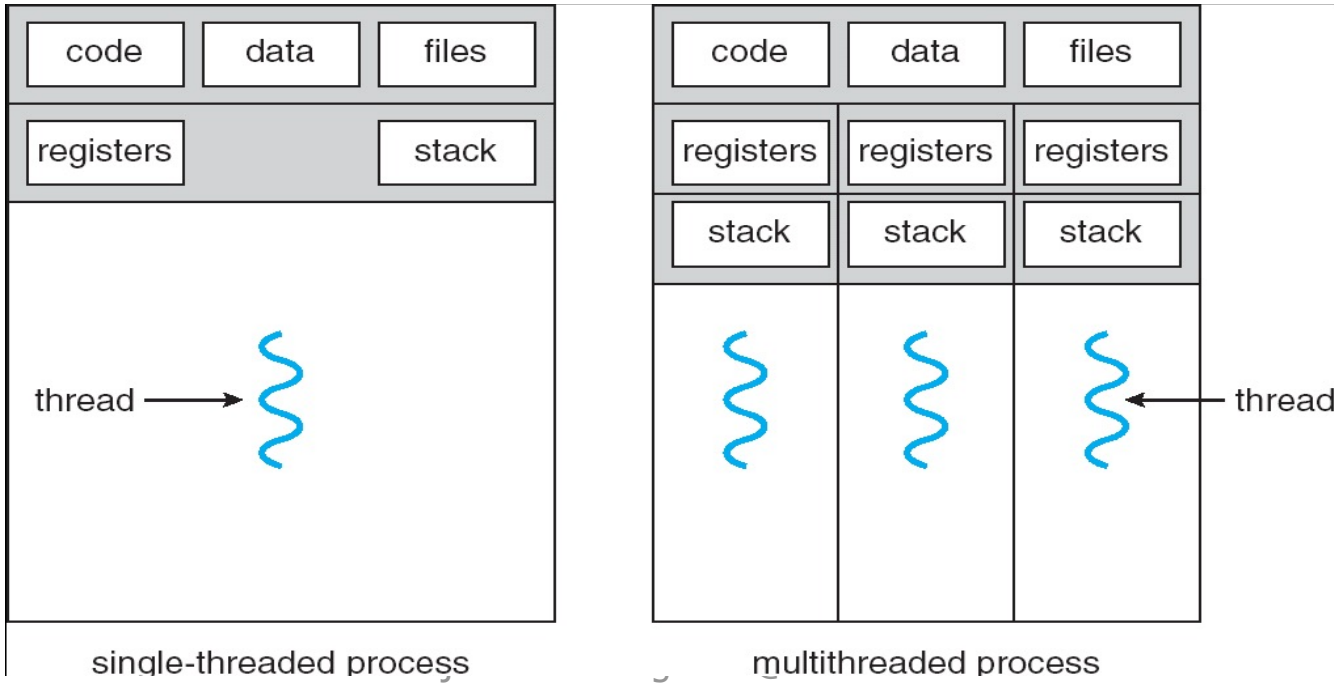
4.1 Overview Threads

- Threads are some sort of "process within a process".
- Threads are parallel flows of control
  - within one process.
  - that are not separated from each other, instead
  - they share resources of the process (in particular the memory of the process).

- Threads are also called lightweight processes:
  - Creation of a thread and context switch is more efficient/faster compared to processes (as long as switching threads within the same process – switching threads of different processes: full context switch):
    - Only a minimal part of the hardware context needs to be saved and restored: Just the CPU registers (incl. program counter (PC) and stack pointer).
    - No switch of address space (hardware for memory management that controls accesses to memory needs not to be reprogrammed during a context switch).

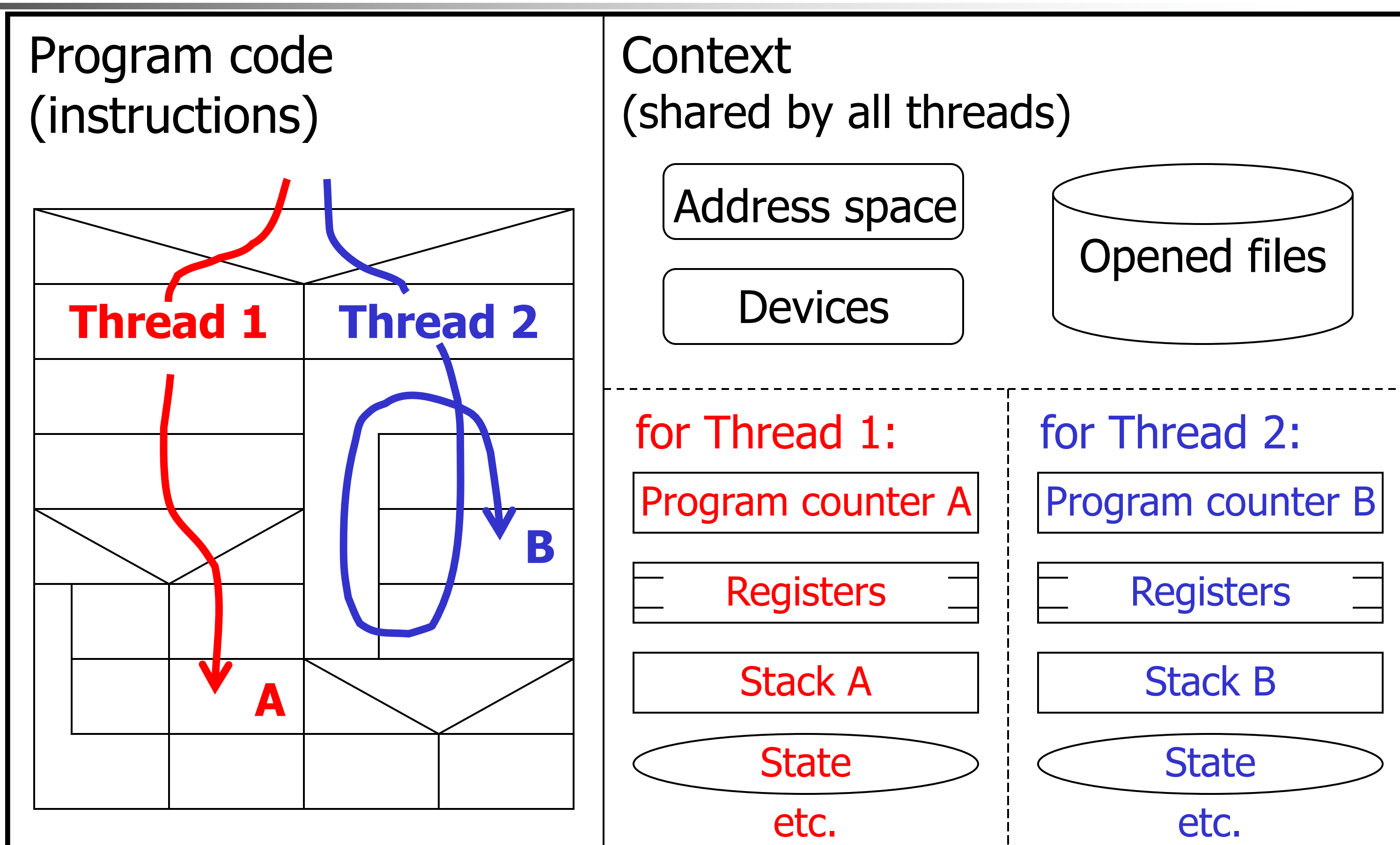
Single and Multithreaded Processes

- On operating systems that support threads, each process has at least one thread that executes the instructions of that process.
- Further threads may be created on request.
  - If process terminates (=one thread calls `exit()`), all contained threads are terminated.
  - However, if a thread just finishes, the surrounding process does not terminate (as long as at least one thread is still running.) – But a crashing thread may crash the whole surrounding process (that's way Google Chrome browser is multiprocess, not multithreaded).



A Closer Look on a Multi-threaded Process

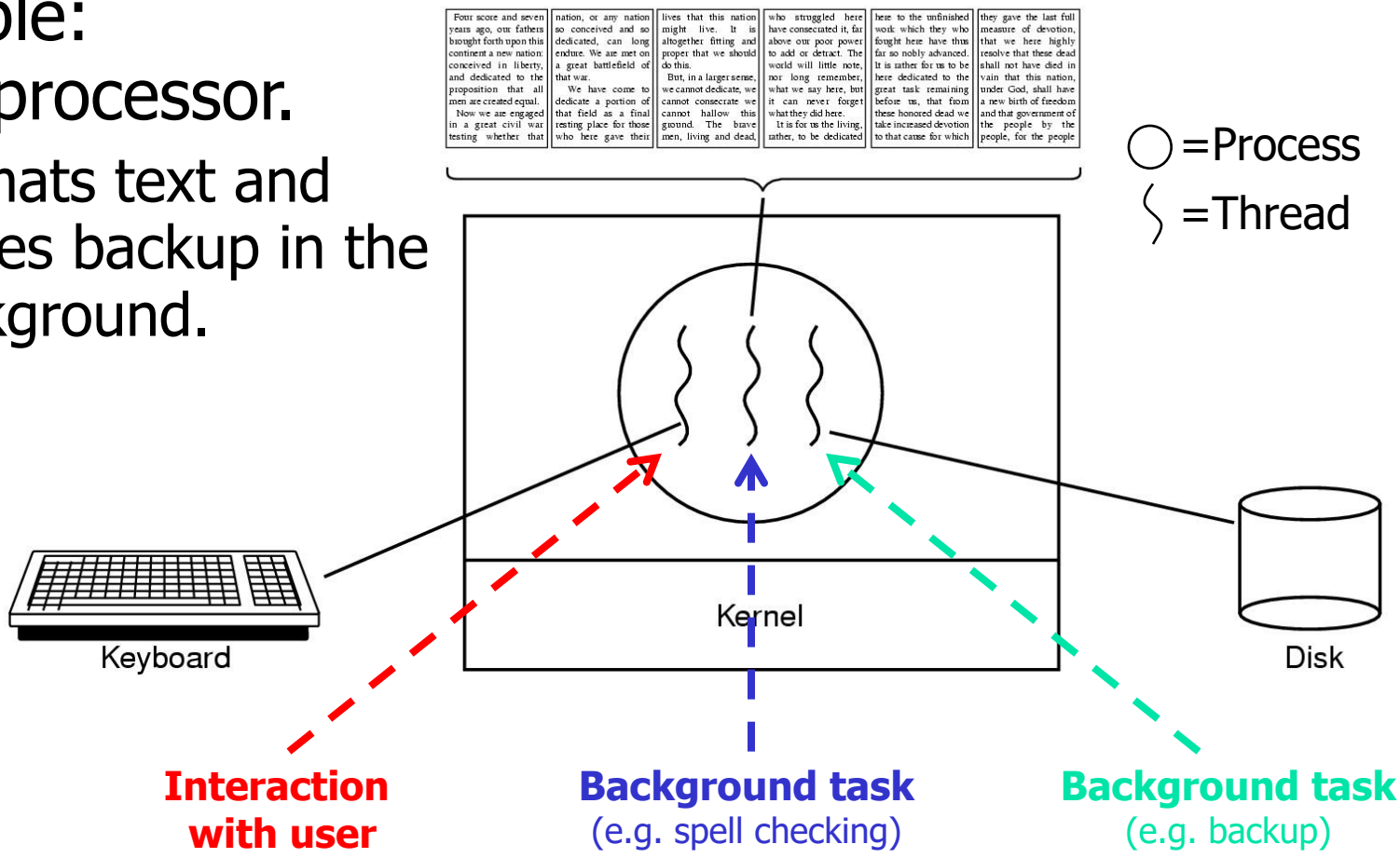
Process (Task)



- In addition to process information stored in PCB, thread information (program counter, registers, thread state) needs to be stored for each thread in a thread control block. No process state anymore, instead each thread has a state.

Application of Threads (1)

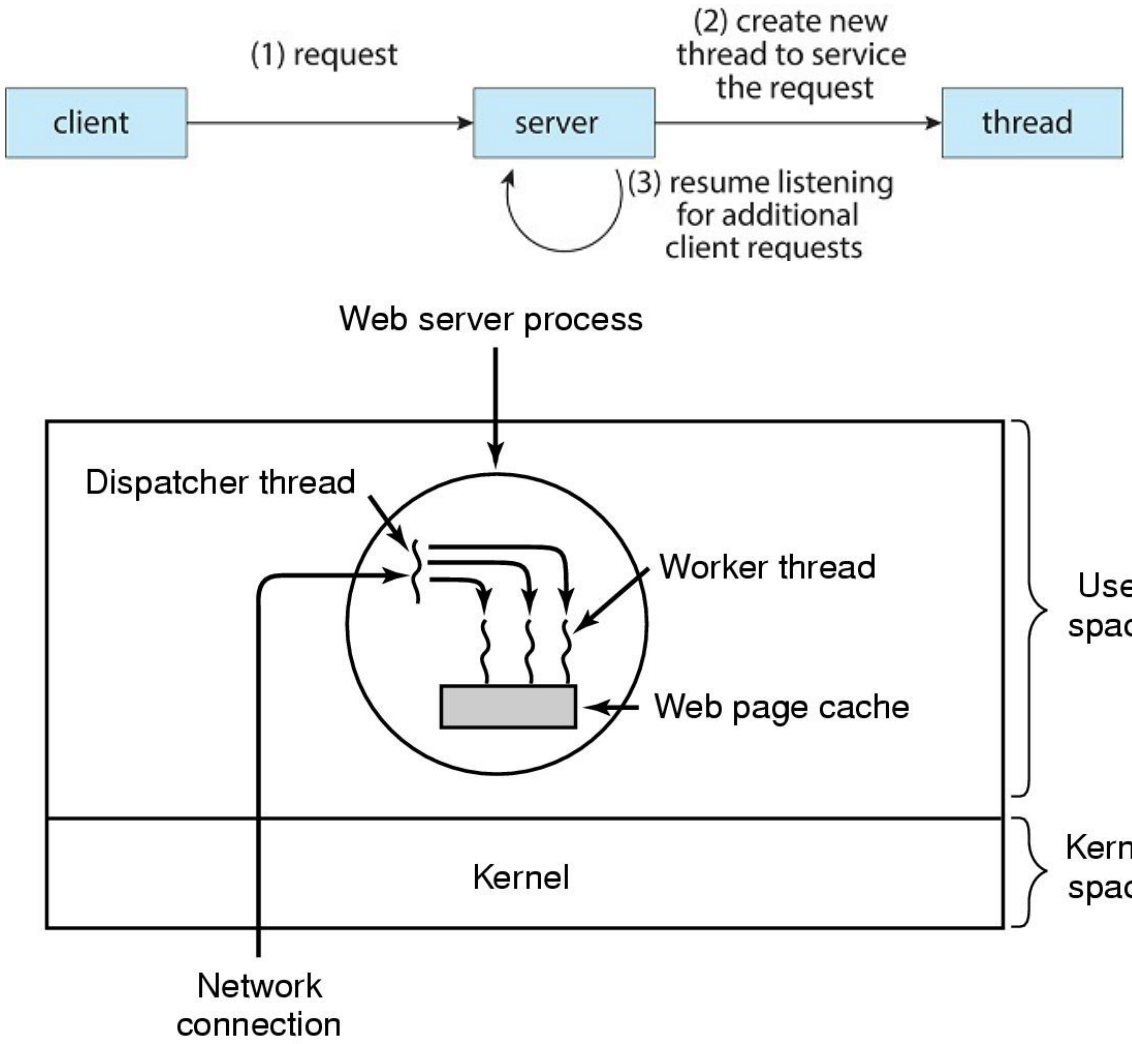
- Example:
  Word processor.
  - Formats text and makes backup in the background.



Application of Threads (2)

- Example: Web server:
  - Incoming request for HTML page:
    - HTML page must be read from disk (or cache) and sent to client.
    - What if a further request arrives during that time?
- Possibilities for designing a server, e.g.:
  1. One process (i.e. one thread):
     - No concurrency, blocking system calls.
       - While processing request (involves blocking system calls, e.g. to read file from disk), no new requests possible. (Client blocked or even aborts due to timeout).
  2. One process with multiple threads (multi-threaded server):
     - Concurrency by threads, blocking system calls are no problem:
       - While one thread processes request (which may involve blocking system calls), a further thread may process further requests.

Application of Threads (3)



Example for possibility 2: Multi-threaded Web Server

- One *dispatcher thread* accepts requests and distributes work to worker threads.
- *Worker threads* load Web page using (blocking) system calls.
  - Variant 1: Each request creates a new thread that terminates itself again.
    - Creating thread and terminating it after request processing finished adds overhead.
  - Variant 2: A pool of worker threads is created once at the beginning: thread from pool is either idle or processes a request.
    - See next slide.

Application of Threads (4)

- Dispatcher thread:

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

(a)

- Each worker thread from thread pool:

```
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);
}
```

(b)

Coordination

Read will block until disk I/O has finished.

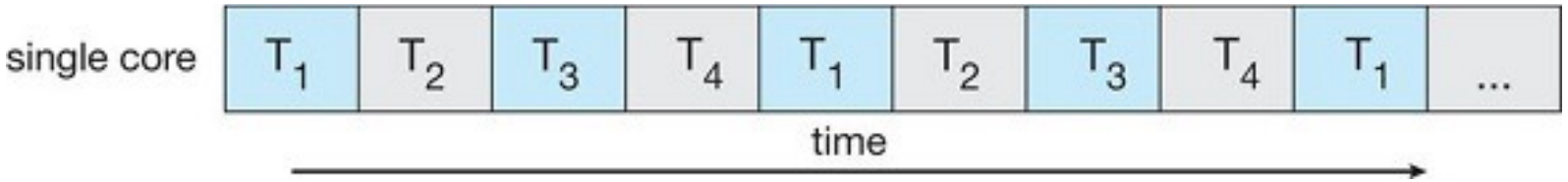
- Not shown:
  - Creation of thread pool.
  - Coordination between dispatcher and worker threads (=Implementation of **`handoff_work()`** & **`wait_for_work()`**).
    - **`wait_for_work()`** will put worker thread to sleep if no work is available.
    - **`handoff_work()`** will put dispatcher thread to sleep if all workers are busy.
    - Note: This is essentially the producer-consumer problem with one producer and multiple consumers.

Benefits of Threads

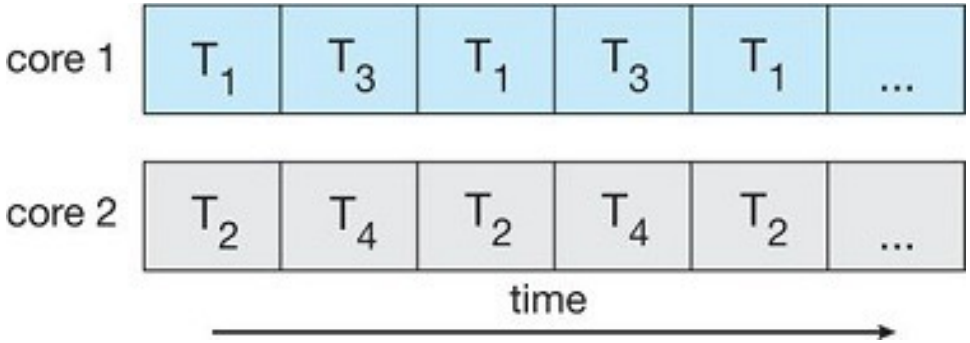
- Responsiveness:
  - A single-threaded process would not be able to perform user interactions while doing some computation. However, a multi-threaded process may start one thread for computation, one thread for user interaction, etc.
  - A multi-threaded server is able to handle multiple client requests at the same time (even if request processing involves blocking system calls).
- Resource sharing:
  - Memory of process is shared between threads. No system calls required for creating shared memory area or for message passing.
    - Nevertheless, synchronisation between threads may be required using system calls. (See later chapter on synchronisation.)
- Economy (reduced overhead):
  - Creating threads is faster than creating processes. Context switch (between threads of same process) is faster for threads than for processes.
- Scalability/Utilization of multiprocessor/multicore architectures:
  - Each thread can be executed by a different processor/core, achieving a speed-up by parallel processing (→next slide).

Concurrency vs. Parallelism

- Concurrent execution on single-core system:
  - Scheduler uses time slices to interleave threads/processes, i.e. they run concurrently (e.g. fight for the same resources), but not really in parallel.



- Multi-core/multi processor system, allows to turn concurrency into real parallelism:



  - If there are more threads/processes than cores/processors, still scheduler needs to interleave time slices (i.e. again concurrency, but true parallelism at least for a subset of the execution).

Multicore (Multiprocessor) Programming

- Threads help to utilise multicore (multiprocessor) systems:
  - If only one process (or thread) would be running, all the other cores/processors would be idle.
  - By writing programs that make use of multithreading, a significant speed-up can be achieved on multicore (multiprocessor) systems.
- Challenges in multicore (multiprocessor) programming:
  - Dividing activities: which activities can run in parallel?
  - Balance: overhead of thread handling/communication/synchronisation may outweigh performance gain (if too small tasks are divided on threads).
  - Data splitting: not only a challenge how to split activities, but also how to divide data processed by different threads.
  - Data dependency: if a thread depends on data produced by another thread, synchronisation between threads needed (& reduced concurrency).
  - Testing and debugging: inherently more difficult than single-threaded applications.

4.2 Multithreading Models Kernel Threads vs. User Threads

- Two possibilities to provide thread support:
- Kernel threads:
  - Thread management provided by the kernel ("kernel space").
    - CPU scheduler responsible for context switch of processes and threads.
- User threads:
  - Thread management provided by a user-level library ("user space").
    - OS is not aware that processes uses multiple threads.
    - Process creates threads via user-level library that manages these.
    - A thread runs until it voluntarily decides to give control to another thread of the same process:
      - Thread has to call the user-level thread library to hand over control to another thread: ordinary (=fast) function call, instead of "expensive" scheduler interrupts and context switch.

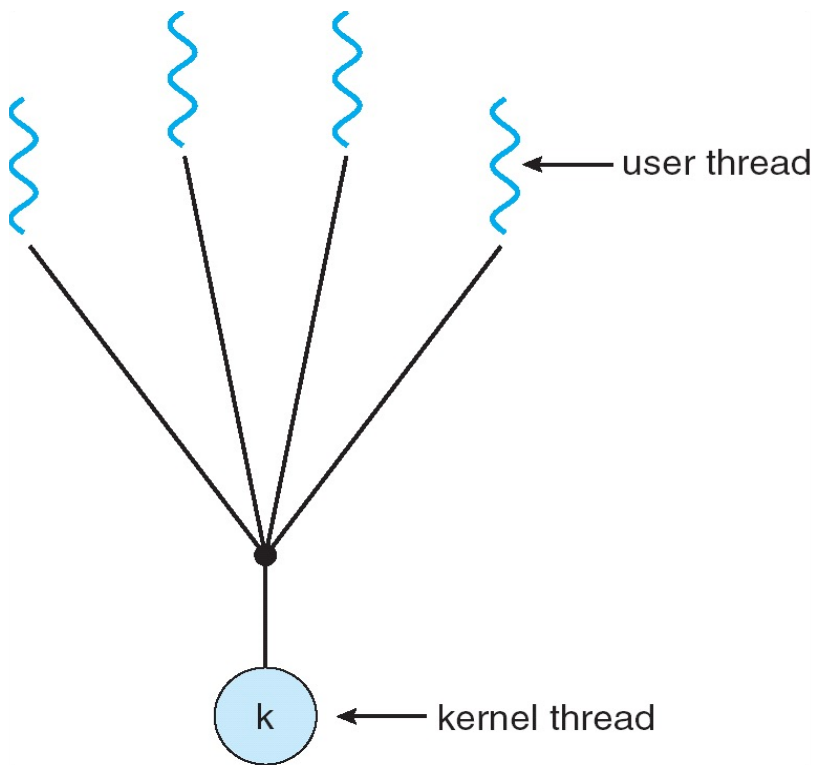
  ⇒Less overhead, but no automatic periodic switching between treads.

Multithreading Models

- Even if kernel-level threads are supported by an OS, user-level threads may be used on top of it.
- Depending on how user-level threads are mapped on kernel-level threads, different multithreading models result:
  - Many-to-One,
  - One-to-One,
  - Many-to-Many.
  - (We will have a closer look on each of them on the following slides…)

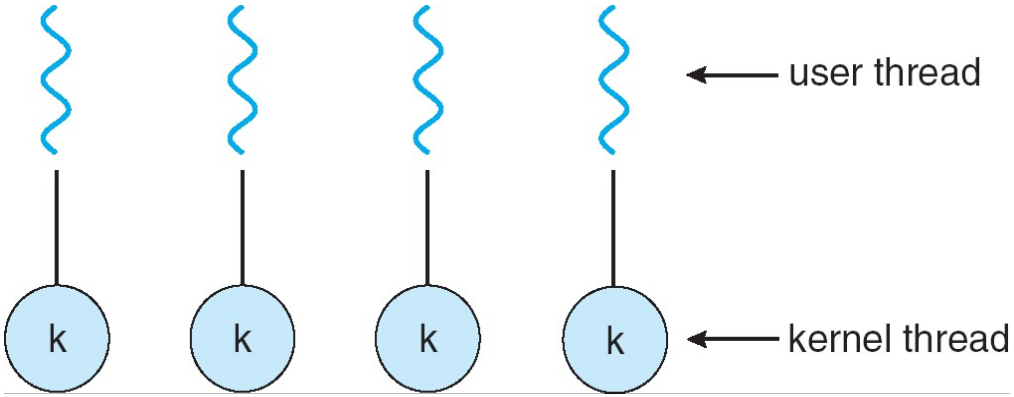
Many-to-One Multithreading Model

- One kernel-level thread per process.
- Multiple threads within a process achieved using user-level threads (less overhead).
  - OS is not aware of user-level threads:
    - If one of the user-level threads makes a blocking system call, all other user-level threads of that process are blocked as well.
    - Cannot make use of multiprocessor/-core architecture to run multiple threads of the same process in parallel (only the OS can distribute threads on different processors/cores).
- Examples:
  - Solaris *Green Threads*,
  - GNU *Portable Threads*.



One-to-One Multithreading Model

- Each user-level thread maps to kernel thread.

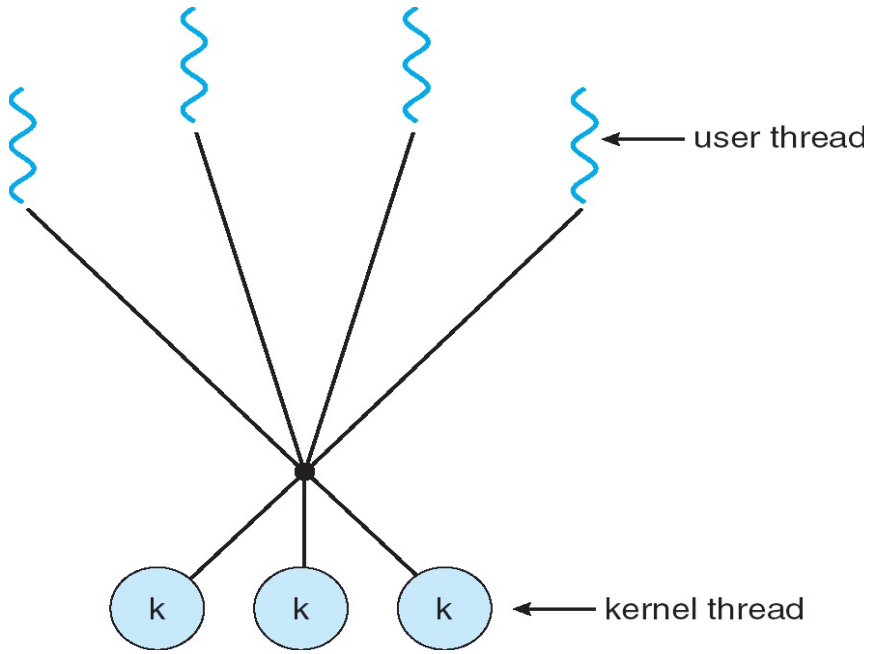


  - Full concurrency:
    - Non-blocked threads of a process may execute while another thread of that process is blocked.
    - Utilisation of multiprocessor/-core architectures (=run threads in parallel.)
  - Higher overhead for creating/switching threads.
- Examples:
  - Windows NT/2000/XP/Vista/7/8/10,
  - Solaris 9 and later,
  - Max OS X,
  - Linux.

$\Rightarrow$ Default in most OSes today
(unless you use a special user thread library).

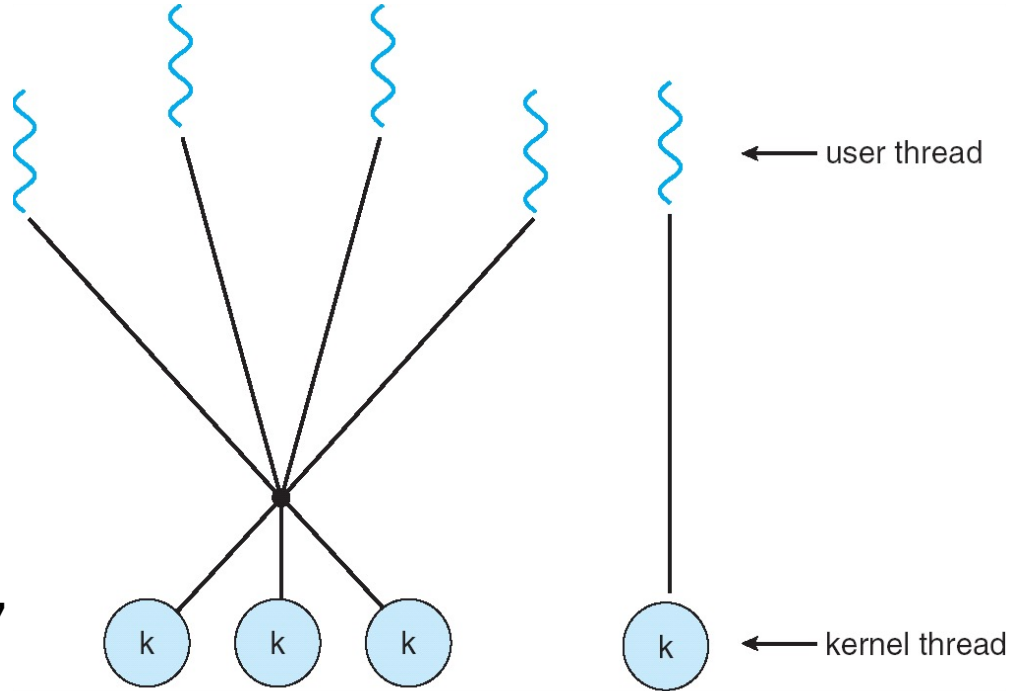
Many-to-Many Multithreading Model

- Many user-level threads mapped to a smaller (or equal) number of kernel threads.
  - Best of both worlds:
    - Application may use efficient user-level threads.
    - However, when thread is blocked, a new kernel level thread is created that may execute one of the remaining non-blocked user-level threads.
- Examples:
  - Solaris prior to version 9,
  - Microsoft Windows ≥NT<7 with the *ThreadFiber* package.
  - Microsoft Windows ≥7 with the *user-mode scheduling (UMS)* threads.



Two-level Multithreading Model

- Similar to Many-to-many model, except that it allows additionally to bind a user thread to a kernel thread.
  - Allows an application to tell the OS which user-level threads needs always real concurrency.



- Examples:
  - HP-UX,
  - Solaris prior to version 9.
  - Microsoft Windows ≥NT<7 the *ThreadFiber* package.
  - Microsoft Windows ≥7 with the *user-mode scheduling (UMS)* threads.

4.3 Thread Libraries

- Thread libraries provide programmers an API for thread handling.
  - May be implemented in user space (i.e. no system calls, no context switch – instead: simple local function calls between threads and thread library are used) or in kernel space (i.e. system calls to use kernel-level threads).
- Three main APIs for threads:
  - POSIX Pthreads (Portable library. May be a kernel- or user-level library.)
  - Win32 threads (MS Windows specific kernel-level library.)
  - Java threads (Java library. On POSIX systems implemented using Pthreads, on MS Windows using Win32 threads.)
  - (In the following, we will discuss Pthreads and Java threads…)

POSIX threads (Pthreads)

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronisation. (C language)
- API specifies interface and behaviour of the thread library.
  - It is the choice of the Pthread library developer how to implement POSIX Pthread API:
    - May be a kernel- or user-level library (i.e. various multithreading models).
    - (Pthread API is flexible enough to support both styles of threads. But: if Pthread library implementation is user-level, switching between threads can only occur when calls to the Pthread library are made – otherwise the thread library does not get control and cannot switch between threads.)
- Common in UNIX operating systems (Solaris, Linux, Mac OS X), but not Microsoft Windows.

Selected Pthread functions

- Create new thread and return its thread Id:

```
int pthread_create (pthread_t *thread,
const pthread_attr_t *attr,
void* (*start_routine)(void), void *arg)
```

Predefined data structure for thread Id

Desired thread attributes

Parameter of C function

Pointer to C function that will be executed in its own thread.

- Thread hands over control to thread library:

```
int pthread_yield() or
int sched_yield()
```

Must be called to switch to another thread if Pthreads are implemented as user-level threads.

- Thread terminates itself:

```
void pthread_exit(void *retval)
```

(Alternatively: `start_routine` finishes)

Return value of Thread

- Wait for termination of a thread:

```
int pthread_join(pthread_t thread,
void **thread_return)
```

ID of thread to wait for.

Pointer to memory location where return value of thread will be stored.

4.4 Java Thread API

- Threads provided by Java Virtual Machine and Java Thread API library.
  - Threads are part of the language: no import of Java API packages required.
- Each Java program comprises at least one thread for running the **`main()`** method.
- Additional Java threads may be defined by:
  - either extending class **`Thread`**.
  - or implementing interface **`Runnable`**.
  - However, this does not actually create or start a new thread:
  - Only by calling the method **`start()`** (provided by class **`Thread`**), a new thread is created and executes instructions contained in method **`run()`**.

(Examples on next slides…)

Java Thread Creation (1) Approach extending class **`Thread`**

- Example extending class **`Thread:`**

```
class MyThread extends Thread {
  public void run() {
    System.out.println("I Am a Further Thread");
  }
}

class MainThread {
  public static void main(String args[]) {
    Thread runner = new MyThread();
    runner.start();
    System.out.println("I Am The Main Thread");
  }
}
```

Class must extend class `Thread`.

Code that shall be executed within a thread.

Create instance of class for executing further thread.

Start new thread. Note: it is not allowed to call `start()` a second time! (For starting multiple threads, each thread must be started in its own instance of the `Thread` class.)

Each instance of class `Thread` provides just a thread control block for one thread.

Java Thread Creation (2) Approach using interface **Runnable**

- If your class already inherits from another class, you may want to implement interface **Runnable** instead:

```
class MyThread2 implements Runnable {
   public void run() {
      System.out.println("I Am a Worker Thread ");
   }
}

class MainThread2 {
   public static void main(String args[]) {
      Runnable runner = new MyThread2();
      Thread thread = new Thread(runner);
      thread.start();
      System.out.println("I Am The Main Thread");
   }
}
```

Class must implement `Runnable`.

Code that shall be executed within a thread.

Pass an implementation of interface `Runnable` as parameter.

Instance of class containing `run()` method.

Create instance of class for executing further thread.

Start new thread. Note: it is not allowed to call `start()` a second time!

Joining Threads

- Method `void join() throws InterruptedException` may be called on any instance of a class `Thread` to wait for the termination of that thread.

```
class JoinableThread extends Thread {
  public void run() {
    System.out.println("Worker working");
  }
}

class MainThread3 {
  public static void main(String[] args) {
    Thread runner = new JoinableThread();
    runner.start();
    try {
      runner.join();
    } catch (InterruptedException ie) {
      System.out.println("Interrupted while waiting thread");
    }
    System.out.println("Worker done");
  }
}
```

Wait for thread to terminate (i.e. `run()` method finishes/returns)

Someone may want to interrupt us while waiting the `join()` is waiting: this will throw an `InterruptedException` that always needs to be caught.

Excursion: Running Lambda Expressions inside Java Threads

- For those familiar with Lambda expressions introduced in Java (1.)8:
  - You can use a Lambda expression (right example) in order to avoid having to create a class containing the thread (left example):

```
class MyThread4 implements Runnable {
   public void run() {
      System.out.println("I am a
Thread");
   }
}
…
Thread thread = new Thread(MyThread4);

thread.start();
```

```
…
Runnable task = () -> {
   System.out.println("I am a thread.");
};


Thread thread = new Thread(task);

thread.start();
```

- In this course, we will use the old-fashion style from slides 4-24 and 4-25.

The JVM and the host OS

- JVM is running on top of an OS and some libraries.
- JVM may decide to map each Java thread on a kernel-thread (one-to-one model), to user-level threads (many-to-one model), or the many-to-many model in-between.
- The Java standard leaves this decision to the particular implementation of the JVM.
  - Typically, if underlying OS supports kernel-threads, one-to-one model is used by a JVM implementation for that specific OS.

4.5 Implicit Threading

- Designing multi-threaded applications not trivial, e.g.
  - When to start a thread,
  - When to terminate a thread,
  - How to exchange data between threads.
- Instead of needing to create threads
- Implicit threading: Instead of letting an application developer writing code for creation and management of threading, let compilers and run-time libraries do this!

Implicit Threading: Thread Pools

- As discussed on slides 4-9 & 4-10, instead of creating a new thread for each request of a multithreaded server, a pool of worker threads that await work can be created in advance.
- Advantages:
  - Usually slightly faster to service a request with an existing thread from the pool than to create a new thread for each request (and terminate it again).
  - Allows to restrict the number of parallel threads in the server.
    - Otherwise, system resources (memory, CPU) might get exhausted and affect also other processes ("denial of service" attacks) if too many request arrive.
- Disadvantage:
  - Management of thread pool more difficult to implement.
- Note: the Java package **`java.util.concurrent`** provides advanced threading support, including thread pools, i.e. let that Java library do all the work needed for thread pool management.
  - Not explained here, but if you need it: look in **`java.util.concurrent`**.

4.6 Threading Issues

Issues mainly due to the fact that threads have been added late to Unix, when processes and process system calls did already exist.

Semantics of Processes Management System Calls

- Consider the fork() system call that creates a copy of the calling process:
  - Calling process may have created multiple threads.
  - Design decision to make:
    - fork() copies all threads of that process,
    - fork() copies only thread that called fork().
  - Both variants of fork() semantics can be found in the different UNIX-based systems.
    - Sometimes, different system calls are offered to allow the application to decide on the semantics to be used.
      - E.g. if anyway exec() is called immediately after the fork to replace the instructions (& threads) of the child process, then it would make no sense to create copies of all the threads.

Threading Issues: Thread Cancellation

- Either a thread terminates itself, or it may be cancelled by another thread before it has finished.
  - Typically, only threads within the same process may cancel each other.
- Two general approaches when thread cancellation takes places:
  - Asynchronous cancellation terminates the target thread immediately. This may leave resources/data in an undefined state.
    - Requires intervention of OS, thus it is typically only available in kernel-level threads.
  - Deferred cancellation allows the target thread to check if it should be cancelled; if yes, it may decide to terminate in an orderly fashion.
    - When threads are implemented by a user-level library outside the OS kernel, typically only deferred cancellation is supported.
  - Note:
    - "Asynchronous" usually refers to a decoupling of events with respect to time. You might wonder, why immediate cancellation is asynchronous?! Here, the term asynchronous refers rather to the fact that the target thread and the thread that tries to cancel the target thread are different threads.

Thread Cancellation in Java

- Asynchronous cancellation possible by calling **`stop()`** method on an instance of class `Thread`.
  - However, asynchronous cancellation may leave resources of the thread in an inconsistent state. (Thread is immediately cancelled, hence no chance for threads to do some clean-up before.)
    - Hence, using the **`stop()`** method is discouraged ("deprecated" =feature may not be anymore available in future Java versions).
- Deferred cancellation possible by calling **`interrupt()`** method on an instance of class `Thread`.
  - Thread itself can (periodically) check using **`isInterrupted()`** method (or by catching **`InterruptedException`**) whether someone requests its cancellation. If yes, the thread may terminate in an orderly fashion by releasing all resources and then simply terminates by leaving it's `run()` method.

Excursion: Signal Handling

- Signals are used in POSIX systems to notify a process that a particular event has occurred. (MS Windows has similar thing: Asynchronous Procedure Calls.)
- Reminder ch. 3: **`kill (pid, SIGKILL)`**

  sends signal **`SIGKILL`** to process **`pid`** , thus terminating it. (Not all signals send via **`kill()`** do necessarily kill a process. E.g.: **`SIGSTOP`** puts a process to sleep, **`SIGCONT`** awakes process again.)
- A signal is called synchronous if source and target are the same, e.g. when a process does a division by zero, it will receive a signal to indicate this.
- A signal is called asynchronous if source and target are different, e.g. when one processes kills another process (or sends some other signal).
- If a signal occurs, it is processed by a signal handler:
  - Either default signal handler provided by operating system (typically just terminates the process)
  - Or user-defined signal handler (may release resources in an orderly fashion before shutting down).
    - POSIX system call **`signal(int signum, sighandler_t handler)`** instructs OS to call function **`handler`** when signal of type **`signum`** is received.

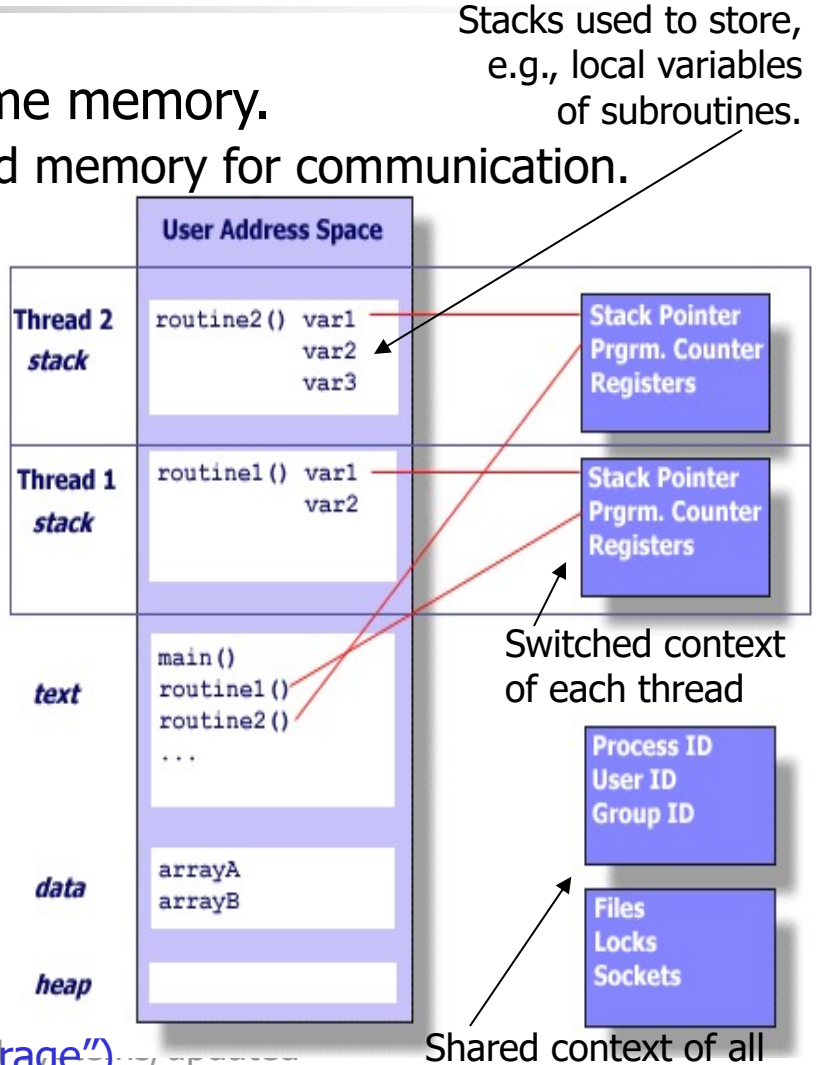
Threading Issues: Signal Handling and Threads

- How to deal with cases where several threads of a process install their own signal handler: to which thread shall a signal that is sent to a process be delivered? (A signal can only be send to a process, but not to an individual thread – POSIX signals are older than POSIX threads.)
  - Option 1: Only one signal handler per process allowed. (Last thread that installs a signal handler wins.)
  - Option 2: Deliver the signal to every thread in the process.
  - Option 3: Deliver the signal to the first thread found that has registered a handler for the signal.
  - Option 4: Assign a specific thread to receive all signals for the process.
  - Further option, only applicable in cases of synchronous signals: Deliver the signal to the thread that triggered the signal.

  - POSIX does not standardise these parts of threading/signal handling:
    - Different implementations of POSIX may handle this differently.

Threading Issues: Thread-specific Data ("Thread-safe" programming)

- All threads of a process share the same memory.
  - Nice when threads want to use shared memory for communication.
  - However, when the same instructions are executed by several threads and each thread shall have its own data, some care is required to achieve that each thread uses its own data.
  - Thread-safe programming using high-level programming languages:
    - Use local variables instead of global variables: local variables are stored on the stack (each thread has its own stack), global variables are shared by all threads.
    - If local variables are not sufficient, most thread-libraries offer calls for requesting a global memory area for thread-specific data ("thread-local storage").



Threading Issues: Scheduler Activations

- Many-to-one multithreading model (user-level threads): thread must call a function of the user-level library to activate thread scheduler.
- One-to-one multithreading model: OS CPU scheduler is activated just as like for processes (i.e. by timer interrupt).
- Many-to-many and Two-level multithreading models:
  - Typically, all threads are user-level, but when a user-level thread is blocked, a further kernel-level thread needs to be created to execute one of the remaining non-blocked user-level threads:
    - Operating system informs user-level thread library when one of the threads is about to block ("upcall": from OS to process).
    - Upcall handler of user-level thread library then schedules/selects the next non-blocked user-level thread to be executed by a new kernel-level thread.

4.8 Summary

- Thread: flow of control within a process.
  - Multithreaded process: multiple flows of control within the same address space.
- User-level threads:
  - Provided by user-level library.
  - Only visible to programmer, unknown to the kernel.
  - Fast, but no real concurrency.
- Kernel-level threads:
  - Provided by kernel.
  - Slower, but full concurrency.
- Threading models:
  - Many-to-one: All user-level threads of a process are mapped to the single kernel-level thread of that process.
  - One-to-one: Each user-level thread maps to one kernel-level thread.
  - Many-to-many: Maps multiple user-level thread maps to a smaller or equal number of kernel-level threads.
- Multiple Thread APIs exist: POSIX Pthreads, Win32, Java Thread API.
- Implicit threading: describe the "what", not the "how" (let library do the work).
- There may be issues, e.g. when using process-only concepts with threads.