Threads

CISC 3320 — CUNY Brooklyn College
Lecture Notes

Threads

In this chapter, we will:

1. Define and exemplify threads,
2. Get to know what computing concepts such as parallelism, concurrence, multitasking, and multithreading indicate,
3. Learn what the benefits and challenges of using threads are,
4. Distinguish between three different threading models,
5. Find what famous thread libraries are available for use, and
6. Study an example of a multithreading Java program.

Threads

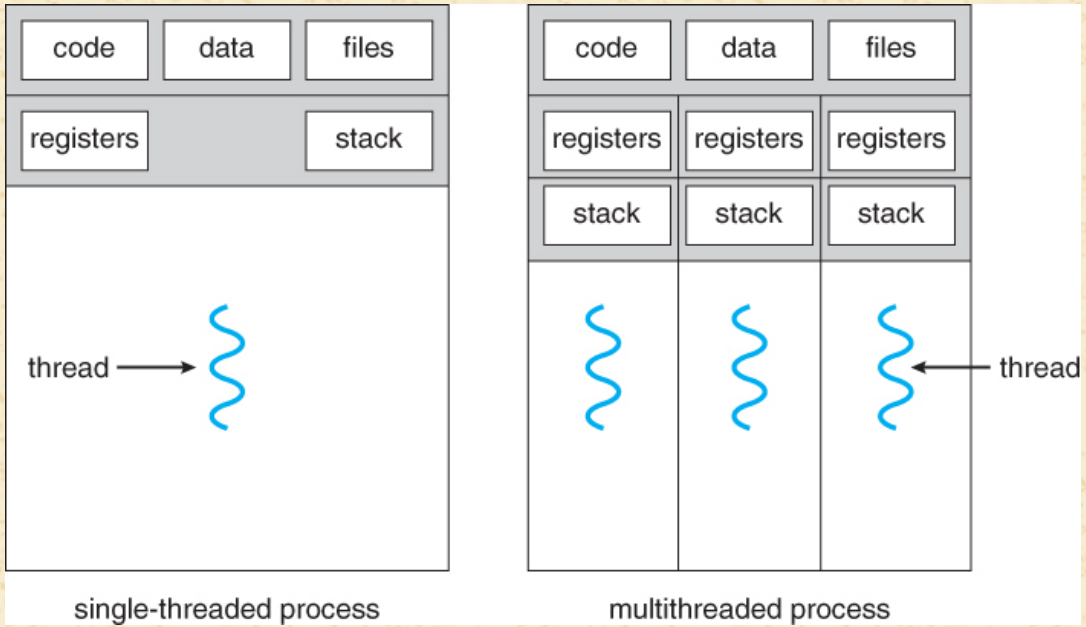
Recall from the previous topic that **processes** are programs that are being executed by being loaded to memory.

To increase the efficiency of running a program that contains similar tasks (like those found in loops) or independent tasks, operating systems support the execution of threads.

A **thread** is a basic unit of CPU execution, consisting of a thread ID, a program counter, a stack, and a set of registers. In the past, most processes were single-threaded; that is, each process had exactly one unit of execution, but most contemporary OSs support **multi-threading** (see the definition in 2 slides from now.)

Threads that belong to the same process will share the same text, data, and heap memory sections and the same process resources, including access to the files that were opened by the process and devices used by the process. The stack memory section, however, isn't common to all threads, as each thread has its own stack as mentioned above.

Threads



Single-threaded and multithreaded processes. Taken from Bell, John T. "Threads." University of Illinois, Chicago.

Threads

The following video explains the idea behind the behavior of threads:

https://www.youtube.com/watch?v=O3EyzlZxx3g

Some Computing Concepts

Parallelism (or **True Parallelism**) is when the instructions of two or more processes (or threads) run simultaneously on two or more CPUs (or multicore processors.)

Concurrence (or **Pseudo** [= false] **Parallelism**) is when the operating system uses context switches to quickly switch between processes (or threads) on a CPU to *create the feeling* that these processes (or threads) run in parallel, without them actually running in parallel.

Multi-programming is the execution of two or more programs at the same time by the OS. The idea is that the OS can store two or more programs in memory simultaneously while the programs wait for a CPU to become available.

Multi-processing is the use of two or more CPUs on the same device to run different parts of a process's code or to run two or more different processes. For example, process A could first run on CPU #0 and then on CPU #1 (or run two of its threads on both CPU #0 and CPU #1 in parallel.)

Some Computing Concepts

Multi-tasking is the ability of the OS to switch (fast) between processes (or threads) on a CPU to *create the feeling* that these processes (or threads) run in parallel, while they actually run concurrently. That is, multi-tasking is the ability of the OS to run processes (or threads) **concurrently**. [When there is more than one CPU, though, processes (or threads) could run **in parallel**.]

Multi-threading is the ability of the OS to switch (fast) between threads on a CPU to *create the feeling* that these threads run in parallel. That is, multi-tasking is the ability of the OS to run threads **concurrently**. [When there is more than one CPU, though, threads could run **in parallel**.]

Some Computing Concepts

Concurrent processes execute in an interleaving manner, "taking turns" on the same CPU, while parallel processes execute simultaneously on two or more CPUs:

- Illustration of Multitasking = Concurrency:

Process 1, CPU 1:

Process 2, CPU 1:

- Illustration of Parallelism:

Process 1, CPU 1:

Process 2, CPU 2:

Concurrency vs. Parallelism of processes. Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Some Computing Concepts

Any questions?

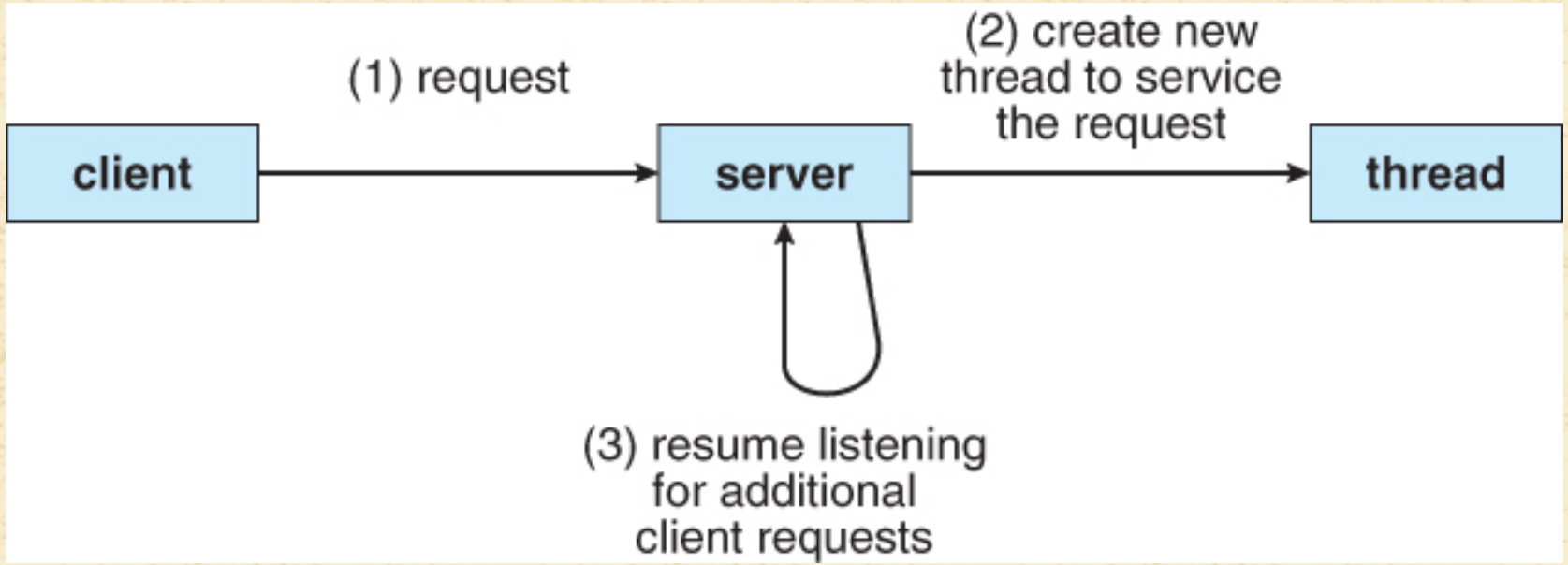
Thread Real-Life Examples

What types of programs could benefit from executing in threads? A program that can be further divided into smaller sets of independent instructions could be made more efficient when we run each such set of instructions in its own thread.

Examples:

- IDE platforms (like Android Studio and Visual Studio) benefit from multithreading since one thread would read the programmer's keyboard input, another thread would check for the correct spelling of keywords and for syntax, a third thread would compile code, etc.
- Web servers (= web applications whose purpose is to perform actions for or to provide information to clients [= users]) benefit from multithreading because each thread could be used to establish a connection between the server and some client. For instance, when three clients make connection requests to the server, at least 3 threads will be created since each thread will manage each connection instance.

Thread Real-Life Examples



Multithreaded server architecture. Taken from Bell, John T. "Threads." University of Illinois, Chicago.

Benefits of Multithreading

- **Programming abstraction**: dividing up work and assigning a separate thread to a subtask. Since threads are independent, each thread is aware only of its own job, without getting into the details of other threads' jobs.
- **Parallelism and utilization**: with multiple processors, threads can achieve true parallelism, as each thread is an independently schedulable entity.
- **Improving responsiveness**: one thread might be dedicated to responding to user input, thus always keeping user interaction responsive.
- **Blocking doesn't influence other threads**: while one thread could be blocking (waiting for I/O or interrupt handling,) other threads will continue running.
- **Faster context switches**: the OS only needs to back up the thread's program counter, stack, and set of registers to memory, but doesn't need to flush the cache back to memory.
- **Memory saving**: threads of the same process share the same memory space, unlike several processes, each of which needs allocation of its own memory space.

Challenges / Possible Issues of Multithreading

It is a challenge to identify:

- What code sections in a program could be divided into tasks that could run concurrently (**Identifying tasks**.)
- What code sections in a program could be divided into tasks that are roughly the same number of instructions so as not to waste CPU time when one thread runs for much more time than others (**Balance**.)
- What code sections in a program could be divided into independent tasks to be run by threads (**Data splitting**.)
- How to synchronize the execution of the threads. If the execution of thread B depends on the results of the execution of thread A, then running B before running A will cause some error in B's results or corrupt data (**Data dependency**.)
- Whether the program runs correctly since timing and order of the execution of multiple threads can be unpredictable (**Testing and debugging**.)

Context Switches between Threads

When we have one CPU and 2 or more threads in a process, the OS will sooner or later perform a context switch between threads. The data of the currently executing thread will be copied from the CPU back to memory, while the data of a thread-to-be-run will be copied thereafter from memory to the CPU to run on it.

The above scenario is quite like the one we saw for processes. The difference is that **intra-process** context switches (that is, those that happen *within* a process, a.k.a. between threads) are faster than **inter-process** context switches (those that happen *between* processes.)

The reason is that, when we switch between one process to its friend, we also need to flush (= save recent changes in) the cache to memory, which is a time-expensive action. Nonetheless, when we switch between threads, there is no need in flushing the cache since the threads of the same process share the same memory and hence the same cache.

Multithreading Models

We can identify two types of threads in contemporary operating systems: user threads and kernel threads.

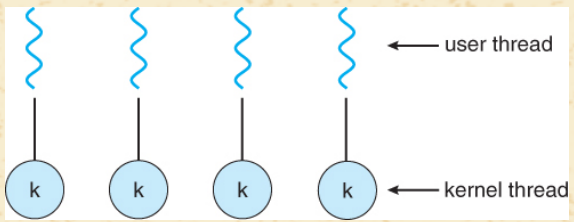
User threads are those coded by programmers and are used in user programs. Example: a multi-threaded Java program consists of user threads.

Kernel threads are internal threads that the operating system decides to create for simultaneous, and therefore efficient, execution of OS code.

Since user programs will resort to kernel services, we need to figure out how user threads will eventually be mapped to kernel threads. We describe three mapping models.

Multithreading Models

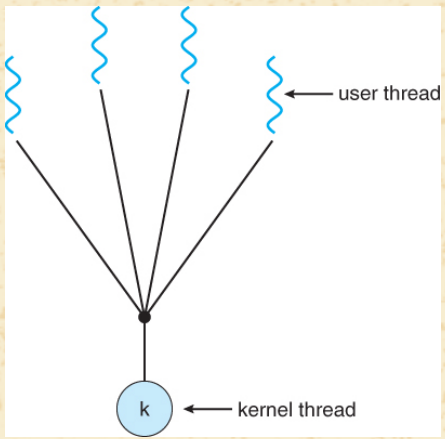
1. **One-To-One (1:1) Model**: For every user thread, the OS creates a separate kernel thread to handle it.
    - Also known as **kernel-level threading** since the kernel implements these threads as processes.
    - An API library creates new threads via the `clone()` system call, similar to `fork()` which creates processes.
    - Since many kernel threads could be created (depending on the number of user threads,) the OS will spend more time switching between them, thus making the OS work slower.
    - What's good about this model is that, when one of the user threads blocks [= waits for I/O or some other activity], the other can continue to operate (since the underlying other kernel threads aren't blocking.)



One-to-one multithreading model. Taken from Bell, John T. "Threads." University of Illinois, Chicago.

Multithreading Models

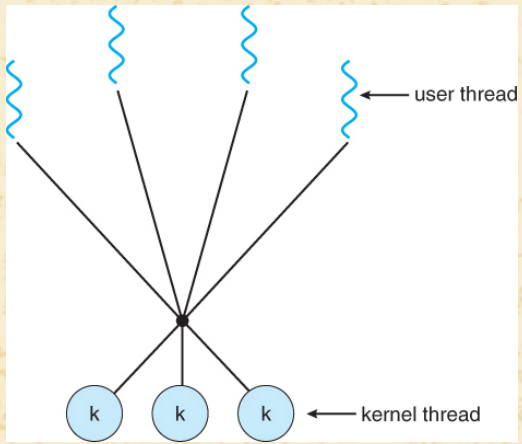
2. **Many-To-One (*N*:1) Model**: A process with *N* threads will map to a single kernel process.
    - It is also known as **user-level threading** since the implementation of the threads is user-program-dependent.
    - Thread management is handled by the thread library in the user space, which is very efficient.
    - The problem is that, when one of the user threads blocks, the other threads block as well since they all depend on a single kernel thread. In addition, due to the single kernel thread, which can run on only 1 CPU, threads can’t run in true parallelism.



Many-to-one multithreading model. Taken from Bell, John T. "Threads." University of Illinois, Chicago.

Multithreading Models

3. **Many-To-Many (*N*:*M*) Model**: Several (*N*) user threads will map to several (*M*) kernel threads. Here, $N \geq M$.
   - Also known as ***N*:*M* threading** and as **hybrid threading**.
   - Attempts to achieve the best of both worlds (true parallelism from the 1:1 model, free context switches from the *N*:1 model.)
   - The number of created kernel threads will vary depending on the number of available CPUs, etc.
   - This model is difficult to implement, making 1:1 model the currently popular choice.



Many-to-Many multithreading model. Taken from Bell, John T. "Threads." University of Illinois, Chicago.

Thread Libraries

With the help of thread libraries, programmers can implement multi-threading in their programs. Libraries could be either user space (C++, Java, etc.) or kernel space (system calls.)

The three most popular thread libraries are:

1. POSIX Pthreads – A Linux (user or kernel) library that is provided as an extension to the POSIX standard.
2. Win32 threads – provided as a kernel-level library on Windows systems.
3. Java threads – since the execution of Java programs depends on the OS on which the JVM is running, the underlying library will be either POSIX Pthreads or Win32 threads.

Here are links to screenshots with sample C programs that use the POSIX Pthreads library and the Win32 threads library.

In the next slides, we describe and illustrate how a Java threads example works.

Thread Libraries

The Java `Driver` program at https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3320/lecture_notes/topic_05/Driver.java creates a thread (besides the main thread) that runs a method that sums up all the integers between 1 and our input integer, inclusively. For example, at the input of 5, the sum 1+2+3+4+5=15 will be returned.

It does so by creating the `SummationThread` class that extends the `Thread` class. The sum is computed inside the `run()` method of the class and is executed when we call the `start()` method on the `SummationThread` object that we create inside the `main()` function. We then call the `getSum()` method on this object to return and print the result of the summation.

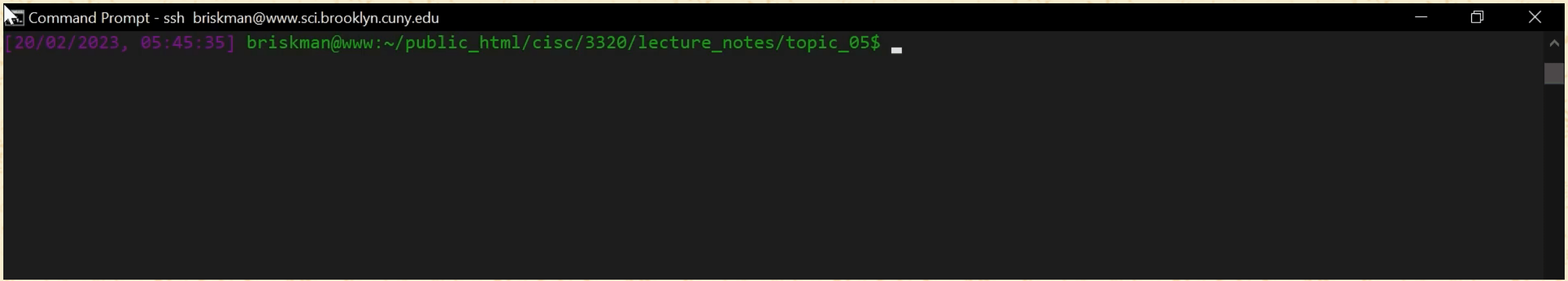
When running the program using the command line/terminal, the argument to the program should be the greatest integer of those we are summing. For example, calling:

```
java Driver 5
```

will output: `sum = 15` since 1+2+3+4+5=15.

Thread Libraries

Below, we compile the `Driver` program and run it with the command-line arguments 3, 4, and 5, respectively:



Running the `Driver` program. Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Thread Libraries

Any questions?

Speeding a Program's Execution using Threads

As we learned, when several CPUs are installed in a computer, programs can create several threads, each of which will run, possibly, on its own CPU. If we divide some work in a relatively even manner across the threads, we decrease the time it takes for the program to run, and thereby make it faster.

For example, if a certain program contains a loop of code, and each of the loop iterations is independent from other iterations, we can divide the work in the loop among several threads to speed the loop's execution.

If the computer contains, say, 4 CPUs, the program could spawn 4 threads (besides the main thread,) each of which will be assigned to a part of the job (in our example, loop,) thereby speeding the job's time *close to* 4 times. **Why *close to* 4 and not *exactly* 4 times faster?** Because the operating system spends some extra time on the creation and management of each thread (a.k.a., "overhead",) which reduces the overall speedup.

A pretty simple math formula that calculates the maximum possible speedup is called **Amdahl's Law**.

Speeding a Program's Execution using Threads

The formula of Amdahl's Law is:

$$Speedup = \frac{1}{(1-p) + \frac{p}{n}}.$$

In the above equation, p is the portion of the program that runs in parallel (a number from 0 to 1,) and n is the number of available CPUs in the device (integer greater or equal to 1.)

That is, to calculate the maximum possible speedup, you need to know (1) what chunk of the program will be run by threads on multiple CPUs, and (2) how many CPUs the computer has. Then, just plug-in the quantities to find the actual speedup.

Speeding a Program's Execution using Threads

An interesting fact that we learn from this formula is that the number of CPUs (processors) in the device is a limiting factor: creating more and more threads won't make code run faster. To actually reach the maximum speedup, the program must create a number of threads that is equal to the number of existing CPUs (not more and not less) that will run that section of code.

Example: If a computer has 4 CPUs, and the portion of the program that is supposed to manifest parallelism is 30%, then the overall speedup of the whole program is:

$$Speedup = \frac{1}{(1-0.3)+\frac{0.3}{4}} = \frac{1}{0.7+0.075} = \frac{1}{0.775} = 1.2903,$$

so the program will run about 1.3 times faster.

Speeding a Program's Execution using Threads

Any questions?

Threads: Sources

Bell, John T. "Threads." *University of Illinois, Chicago*, Accessed 11 July 2022. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Chapter 2: Threads.

Hailperin, Max (2019). Operating Systems and Middleware: Supporting Controlled Interaction. (Revised edition 1.3.1) S.l.: Gustavus Adolphus College. This work is licensed under the Creative Commons Attribution-ShareAlike3.0 Unported License. URL: https://gustavus.edu/mcs/max/os-book/osm-rev1.3.1.pdf#chapter.2