Tutoriat 3 (Operating Systems)
Threads

Contents

1 Introduction

A **thread** (also known as a **lightweight process**) is the *smallest unit of execution within a process* (a certain sequence of instructions from within a program that can be scheduled for execution on a CPU core). A newly created process is made up of one thread, and the process may be split into multiple threads, each of which executes its own sequence of instructions. Here are some similarities and differences between processes and threads:

- A process is a program in execution, whereas a thread is a part of a process.
- A process has an associated **PCB**, while a thread has an associated **TCB (thread control block)**, containing the thread ID, program counter, register set, and a stack (for *activation records*).
- Threads and processes share the same states (new, ready, running, blocked, terminated).
- Threads are **created**, processes are **forked**. Both of them can perform **context switches**, but creating and context switching between processes take longer than for threads. While a parent process **waits** for its child processes to terminate, a thread can intentionally wait for another thread to finish by calling **join** (however, threads do not have parent-child relationships in the way that processes do).
- Processes communicate with each other through **IPC mechanisms** (*shared memory* and *message passing*), while all threads belonging to the same process share the **code section**, **data section**, **heap section**, and other OS resources (e.g. open files and signals), providing more efficient communication.

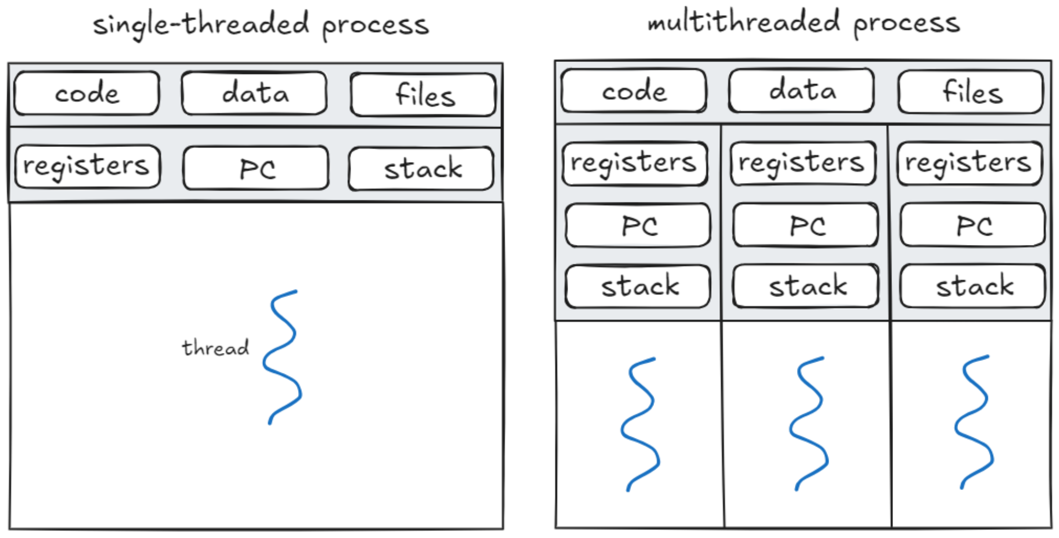

Figure 1: A single-threaded process and a multithreaded process

Benefits of a multithreaded application (possible issues will be discussed later):

- **Responsiveness:** if a part of an interactive multithreaded application blocks or executes a resource-intensive operation, the rest of the application remains responsive. A single-threaded application would have been completely blocked.
- **Resource sharing:** processes can communicate with each other only through *shared memory* or *message passing*. However, threads within the same process implicitly share the process' resources, making communication faster.
- **Economy:** process creation and context switching are costly operations. With threads being smaller and implicitly sharing resources, the overheads are lower.
- **Scalability:** each separate thread can run on a CPU core.

2 Multicore programming

Some of the challenges of creating a multicore system include:

- **Identifying tasks:** examining the application to find areas that can be divided into separate tasks. Ideally, the tasks should be independent of each other, allowing them to run in parallel on individual cores.
- **Balance:** when identifying the tasks that can run in parallel, they should also be performing work of equal value. Using a separate core to run a task that does not contribute as much value to the overall process as other tasks may not be worth it.
- **Data splitting:** the data accessed and manipulated by the tasks must be divided to run on separate cores.
- **Data dependency:** the data accessed by the tasks must be examined for dependencies between two or more tasks; in that case, the execution of the tasks must be synchronized (this will be discussed at another time).
- **Testing and debugging:** when a program is running in parallel on multiple cores, many different execution paths are possible. Thus, testing and debugging inherently becomes much more complex.

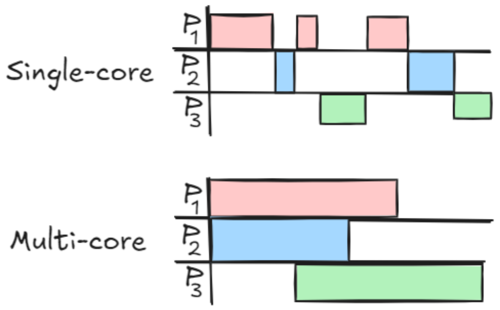

Figure 2: Concurrent execution on a single-core system and paralellism on a multi-core system

Recall the two following notions: **concurrency** is the ability to manage multiple tasks at once by switching between them, whereas **paralellism** is the actual simultaneous execution of multiple tasks. There are two types of parallelism:

- **Data parallelism:** distributes subsets of the same data across multiple cores, each core performing the same operation on the data.
- **Task parallelism:** distributing threads across cores, each thread performing its own unique operation.

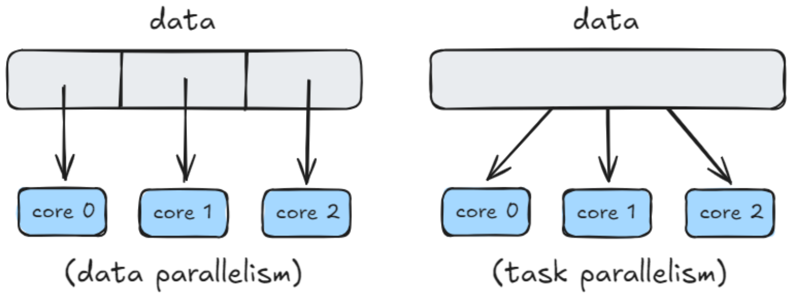

Figure 3: Data parallelism and task paralellism

Assume that we have an application that has both **serial** (nonparallel) and **parallel** components. **Amdahl's Law** is a formula that predicts the theoretical speedup of a task, only when a portion of it can be improved; the law states that the overall performance improvement is limited by the serial (non-parallelizable) fraction of the task, no matter how much the parallel part is improved. Thus, a system is only as fast as its weakest link. If **S** is the portion of the application that must be performed serially on a system with **N** processing cores, then the formula is as follows:

$$\texttt{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

If a task is 75% parallelizable and 25% serial, the maximum speedup achievable with an infinite number of processors would be:

$$\texttt{speedup} \leq \frac{1}{0.25 + \frac{1-0.25}{\text{inf}}} = \frac{1}{0.25} = 4$$

Thus, the task can be performed at most four times faster. Using four processors:

$$\texttt{speedup} = \frac{1}{0.25 + \frac{1-0.25}{4}} \approx 2.28$$

2.1 Multithreading models

A **user thread** is a thread managed entirely by a user-level thread library in user space, with no direct kernel involvement; the kernel is unaware of its existence. As a result, if one user thread performs a blocking operation, the entire process may block, since the kernel blocks at the process level, unaware of the individual user threads. A **kernel thread** is a thread managed directly by the kernel, which can schedule it on a CPU and handle blocking operations.

User threads offer much faster creation, synchronization, and context switching than kernel threads. However, they cannot achieve true *parallelism* unless mapped to kernel threads, since only the kernel can schedule threads onto the CPU cores and manage I/O operations. Thus, a **mapping** between user and kernel threads is necessary.

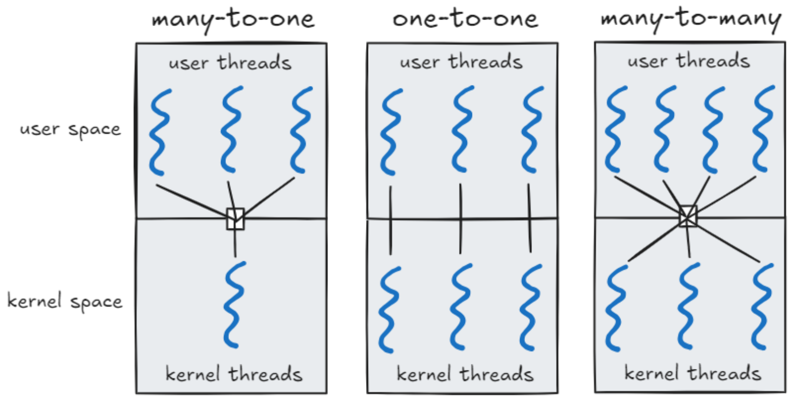

Figure 4: Different user to kernel thread mappings

2.1.1 Many-to-one model

This model maps **many** user-level threads to **one** kernel thread. Thread management is efficient, as it is done by the thread library in user space (kernel threads are slower to create and switch due to kernel involvement). However, since only one thread can access the kernel at a time, this model cannot take advantage of multicore systems because of the inability to provide parallelism. On top of that, if one user thread makes a blocking operation, it halts the single kernel thread, preventing any other user threads within the process from running, consequently blocking the whole process.

2.1.2 One-to-one model

Each user thread is mapped to a kernel thread. This provides more **concurrency**, allowing another thread to run when a thread blocks. Multiple threads may also run in parallel on multicore systems. The main drawback would be having to create a large number of kernel threads, slowing down the performance of the system.

2.1.3 Many-to-many model

In this model, many user threads are mapped to a smaller or equal number of kernel threads. Thus, user threads can run in parallel on multiple cores; if a user thread blocks, another thread can be scheduled to run on a different kernel thread, preventing the entire process from blocking. There is still some overhead from managing the kernel threads, and the implementation may be more complex than the other two models, but this approach is flexible, providing strong concurrency and parallelism.

3 Implicit threading

Designing an application with hundreds (or even thousands) of threads is a complex task. One way to address the difficulties that arise when working with a large number of threads would be to transfer the thread management and creation responsibilities from application developers to compilers and run-time libraries. This strategy is also known as **implicit threading**.

3.1 Thread pools

When working with threads, the amount of time required to create a thread is an issue; additionally, the troublesome act of discarding a thread once it completes its work is a waste of time and resources. A solution to this may be **thread pools**: a collection of pre-instantiated, idle threads that are ready to execute tasks.

Performance is improved because instead of creating a new thread for every task, the thread pool assigns an available thread from the pool to the task, and once the task is complete, the thread sits idle in the pool, waiting to get assigned again. If there are no available threads, the task is queued until one becomes free. A thread pool brings about the following benefits:

- Servicing a request with an existing thread is often faster than waiting to create a thread.
- The number of threads that can exist at any one point is limited. This is important on systems that cannot support a large number of concurrent threads.
- Separating the task to be performed from the mechanics of creating the task allows for more flexibility when running the task.

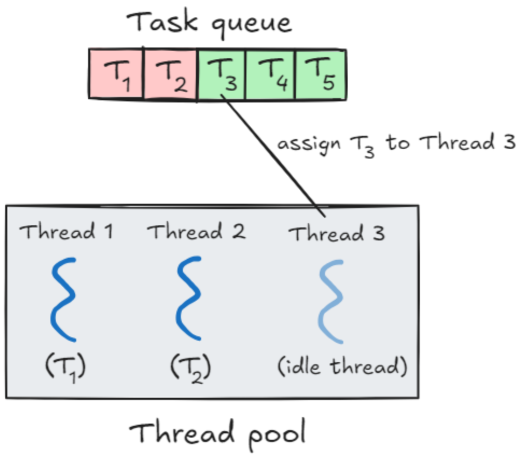

Figure 5: A queue of tasks waiting to get assigned to threads in a thread pool

3.2 Fork join

A framework is used that automatically handles the creation, scheduling, and lifecycle of threads. Programmers work with tasks, not threads, allowing them to focus on the problem logic. This method is divided into two phases:

- **Fork (dividing tasks):** a large, computationally intensive task is identified. A thread begins to execute it. If the task is too large to handle directly, it *forks* itself by recursively breaking it into smaller independent subtasks.
- **Join (collecting results):** when the main thread needs the result of a subtask, it calls the **join** method. The results of the subtasks are combined to produce the final result for the larger task.

3.3 OpenMP

OpenMP is an API for programs written in C, C++, and FORTRAN, that provides support for *parallel programming* in shared-memory environments. **Parallel regions** are blocks of code that may run in parallel. The program begins as a single thread; when OpenMP reaches a parallel region, the main thread *forks* a bunch of new threads that execute the region in parallel and ultimately *join* back into the main thread.

Application developers must identify parallel regions and insert special instructions that instruct the OpenMP library to execute the region in parallel. Additionally, the developer may edit various parameters, such as manually setting the number of threads to be created.

3.4 Grand Central Dispatch

Grand Central Dispatch (GCD) is a technology developed by Apple for its macOS and iOS systems. It allows developers to identify sections of code (*tasks*) to run in parallel. When a task needs to be performed, it is added to a **dispatch queue**, which can be of two types:

- **Serial (main) queue:** tasks are executed one after another in a FIFO order. Programmers can also create additional serial queues.
- **Concurrent queue:** tasks are started in the order they were added, but they can run in parallel.

3.5 Intel Thread Building Blocks (TBB)

TBB is a template library for designing parallel applications in C++. Developers specify tasks that can run in parallel, and the *TBB task scheduler* maps them onto underlying threads.

4 Threading issues

4.1 fork() and exec() syscalls

If one thread in a program calls *fork()*, does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads, and another that duplicates only thread that invoked the fork() call. If a thread invokes the *exec()* syscall, the program specified in the parameter to exec() will replace the entire process, including all threads.

4.2 Signal handling

A **signal** is used to inform a process that a particular event has occurred. All signals follow the same pattern:

- A signal is generated by the occurrence of a particular event.
- The signal is sent to a process.
- Once delivered, the signal must be handled.

Every signal has a **default signal handler** that the kernel runs when handling that signal. This default action can be overridden by a **user-defined signal handler** that is called to handle the signal. The handling of signals in single-threaded programs is straightforward: they are always delivered to a process. However, it is more complicated in multithreaded programs; where should the signal be delivered? The following options exist:

- Deliver the signal to the thread to which the signal applies.
- Deliver the 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.

4.3 Thread cancellation

Thread cancellation means terminating a thread before it completes. For example, if multiple threads are searching through a database and one thread returns the result, the remaining threads can be canceled. The problem with thread cancelation occurs in situations where resources have been allocated to a canceled thread, or when a thread is canceled while in the midst of updating data it is sharing with other threads. A thread that will be canceled is also known as a **target thread**. Cancelation of a target thread may occur in two different scenarios:

- **Asynchronous cancelation:** one thread immediately terminates the target thread.
- **Deferred cancelation:** a thread may indicate that a target thread is to be canceled, but cancelation occurs only after the target thread has checked a flag to determine whether or not it should be canceled (at a **cancelation point**).

4.4 Thread-local storage

Threads belonging to a process share the data of the process. However, in some circumstances, each thread might need its own copy of certain data. Such data is known as **thread-local storage (TLS)**. This storage is different from *local variables*, since those are only visible during a single function invocation, whereas TLS data are visible across multiple function invocations. The TLS is unique to each thread.

4.5 Scheduler activities

One more problem would be the communication between the thread library and the kernel threads. Many systems choose to place an intermediate data structure between the user and kernel threads: a **lightweight process (LWP)**, which is like a virtual processor that user threads can get scheduled to run on. Every LWP is attached to a kernel thread.

One scheme for communication between the thread library and the kernel threads is known as **scheduler activation**. The kernel provides an application a set of virtual processors (LWPs) that the user threads can get scheduled onto. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler** (that must run on a virtual processor).