Concurrent Computing 1: Introduction

1. **Introduction**
   - **thread of execution**
   - **OS threads**
   - **multi-process vs. multithreaded**
   - **advantages and disadvantages**
2. Terminology
3. Synchronization
4. C and Concurrency
5. More Asynchrony
6. Event-Driven Programming

Concurrent Computing

Concurrent computing (or **concurrent programming**) refers to programs that create *multiple* **threads of execution**.

A **thread of execution** (or **thread of control**) is a *sequence of program statements* being executed as a unit.

A concurrent program uses *multiple threads of execution* to together accomplish the overall goals of the program.

The multiple threads are run *concurrently*, i.e., *"in parallel."*

We say *"in parallel"* because parallelism will be merely *simulated* unless the hardware contains *multiple* **CPUs/cores**.

Concurrent Computing (contd.)

Parallel execution is simulated with a single core through **time slicing**: one thread runs for a brief period of time, then a second is run, etc., in a *"round robin"* manner more or less.

So the term **concurrent execution** means there are multiple threads of execution, which may be executing simultaneously or may be *interleaved* via time slicing.

The term **parallel execution** indicates *true simultaneous execution*.

Thread of Execution vs. OS Thread

An **OS thread** is an operating system construct that is used to encapsulate one of possibly multiple *threads of execution* running within the context of a *single process*.

OS threads allow concurrent programs where the program statements in the multiple threads of execution are all part of the same program.

Virtually all modern OS kernels support **multithreading** (OS threads).

The Linux/UNIX OS threads model is known as **POSIX threads** or **Pthreads**.

Thread of Execution vs. OS Thread (contd.)

The term *thread* derives from "thread of execution," but the word thread alone can refer to either (1) a thread of execution or (2) an OS thread.

The focus of these slides is on general concepts in concurrent programming.

"Thread" will be used in the generic *thread of execution* sense.

When we need to refer to the operating system capability, we will use the phrase *"OS thread."*

Multi-Process vs. Multithreaded Programs

A concurrent program can be implemented in one of two ways:

- as *multiple* **processes** (e.g., created with `fork()`)
- as *multiple* **OS threads** (e.g., created with `pthread_create()`)

(It is also possible to mix the two approaches: use multiple processes, some of which contain multiple OS threads.)

A concurrent program that uses *multiple processes* is most commonly called a **multi-process program**.

A concurrent program that uses *multiple OS threads* is most commonly called a **multithreaded program**.

Multi-Process vs. Multithreaded (contd.)

The key difference is whether each thread of execution is a *separate process* or whether all threads are *within a single process*:

- *processes* run in *separate address spaces*
- *OS threads* run in a *single, shared address space*

Normally a running program (a **process**) is being executed at a single point in its address space.

However, OS threads allow simultaneous execution at multiple points in a process' address space.

(Each OS thread has its own separate **program counter**.)

Multi-Process vs. Multithreaded (contd.)

While many of the complications of concurrent programming are common to both approaches, there are some key differences:

- Processes will have to explicitly transfer any data needed by other processes, using **interprocess communication (IPC) mechanisms**.
- OS threads automatically share data due to their shared address space.
- Processes are (largely) independent, so they do not involve as many **shared resources** that need to be protected from corruption.
- OS threads can have significant problems avoiding corruption of shared resources.

Language Support

Some programming languages contain language constructs to support concurrent programming, such as Java (`Thread` class, `run` method of `Runnable` interface, etc.).

Prior to C11, C did *not* provide language support for concurrent programming.

Concurrent programming in C can be accomplished through the use of OS **system calls**.

C11 added support for multithreading to C (more later).

Concurrency Advantages

Important reasons for implementing concurrent programs are:

- program operations that can **block** *indefinitely* may cause a program to be *unresponsive*
- programs may have to be able to rapidly respond to various **asynchronous events**
- programs may have to be able to simultaneously interact with multiple network clients, I/O devices, etc.
- large-scale applications may prefer to run certain functionality "**in the background**," using "spare" CPU cycles
- machines that have multiple CPUs/cores can support true **parallel execution** of program operations

Concurrency Advantages: Blocking

Certain operations involving *I/O or signals* can have the potential to *block indefinitely*.

In a single-thread program, this would mean that the program would get *suspended* while blocked.

The program would not be able to do other useful work or respond to events like user keyboard input, possibly for very significant periods of time.

The result is a program that appears to be "slow" or unresponsive (have poor **interactivity**).

Dividing a program's functionality into multiple threads allows a program to continue to do useful work and/or respond to inputs even while some of the thread(s) are blocked.

Concurrency Advantages: Asynchronous Events

Programs such as those involving GUIs often have to be able to respond relatively rapidly to any of several asynchronous events, such as a keyboard inputs, mouse movements/clicks, and so forth.

If significant computations must be done for each event, a single-thread program might be unable to immediately respond to new events.

This can lead to the GUI, etc. being very unresponsive.

If a separate thread is created to handle each type of event, event handling and event processing will be automatically interleaved via OS thread scheduling.

Concurrency Advantages: Multiple Clients

Most server programs will need to be able to handle more than one client at a time.

However, requests from clients will be received asynchronously and usually unpredictably, and may require varying amounts of time to service.

The most straightforward method to produce a **concurrent server** is to have the server program create a separate thread to handle each client's requests.

Concurrency Advantages: Background Tasks

Large-scale software systems may have components that perform some tasks that should have lower **priority**, i.e., be run only when there are spare CPU cycles.

Example tasks include **memory management** functions such as **garbage collection** and **defragmentation**, updating database summaries/reports, and so forth.

By assigning these tasks to separate threads with lower priority, the OS will take care scheduling them to run only when the system has nothing else useful to do.

Concurrency Advantages: Parallelism

One way to *decrease the runtime* (elapsed time) of a program is to perform at least some of the program's operations *in parallel*, using a machine with multiple CPUs/cores.

This is the standard approach for building **supercomputers**: massively parallel multi-processor systems consisting of thousands of CPUs.

Providing multiple cores is now the most cost-effective approach for increasing the computational capacity of microprocessors.

It is becoming increasingly common that laptops, tablets, and even phones contain two or more CPUs/cores.

Concurrency Advantages: Parallelism (contd.)

While it is obvious that machines with multiple CPUs can run multiple (separate) programs faster than can machines with a single CPU, increasing the speed of a single program on multi-CPU machines requires more effort.

Not only must a program use concurrent programming techniques, the program must be written in such a way that the multiple threads of execution can truly run simultaneously on different CPUs/cores.

How much parallel execution is possible will depend on the program's algorithmic logic, as dependencies between the program's steps will limit it.

Concurrency Alternatives

Of the listed situations where concurrency is an advantage, all but achieving true parallelism can be handled *without multiple threads of execution*.

In most cases, however, the resulting code will be much more complicated, requiring advanced system call programming.

A big advantage of concurrent programs is that they shift the responsibility for recognizing when program subtasks are able to be executed and scheduling among them, from the program to the OS.

In an analogous way, programming languages like Java and Lisp that include *garbage collection* runtime systems, simplify the programming task by shifting much of the burden of memory management away from the program code itself.

Concurrency Alternatives (contd.)

Writing a single-thread program that avoids blocking issues can require use of techniques such as **nonblocking I/O** and testing for **pending signals**, and may require *inefficient* **polling** designs.

Enabling a single-thread program to be able to be responsive in handling multiple events requires using techniques such as **I/O multiplexing**, **signal handlers**, **timers**, and **polling**.

Concurrent servers can be implemented using **I/O multiplexing**, but this approach can require the user of **timers** unless server requests involve little computation.

Given the increasing prevalence of multi-CPU machines, the use of concurrent programming techniques will become increasingly advantageous.

Concurrency Disadvantages

While concurrent programs have clear advantages for certain applications, they generally involve several sources of complexity that are not encountered in traditional single process, single thread programs.

The key source of complexity is that the operations in each thread occur **asynchronously** relative to those in the other threads.

This means that programmers cannot be certain what order the operations in the different threads will end up being *interleaved* (or even which may be executed simultaneously).

Furthermore, operation order may differ from run to run!

Concurrency Disadvantages (contd.)

The reason for this is the decisions about when and for how long to execute any of the threads (whether processes or OS threads) are made by the kernel **scheduler** rather than the programmer.

In traditional single thread programs, operations are **synchronous**: the programmer knows exactly what *order* all operations will be executed in (though he may not know which branches will be taken) because operation order will be based on the order he used for the program statements.

Concurrent programs can suffer from **race conditions**, where the (correct) output of the program depends critically on the *relative order* in which certain operations in the threads end up being executed.

Concurrency Disadvantages (contd.)

When operations in different threads get scheduled in undesired or unforseen orders, a concurrent program with a race condition may produce incorrect output or even **deadlock** (hang).

Concurrent programs must either be designed to work with *any possible interleaving* of thread operations or else the processes must include **synchronization** operations.

Synchronization can ensure that operations get interleaved only in appropriate orders and/or that certain operations do not occur simultaneously.

Unfortunately, testing and debugging concurrent programs is difficult; even if a concurrent program is run very many times with a range of inputs, this does not guarantee that every possible operation ordering has been seen.