Lab Notes
CMPU4021 Distributed Systems

Concurrent Programming

Concurrency

- Concurrency in distributed system**s**
  - Concurrent requests to its resources
  - Each resource must be designed to be safe in a concurrent environment

- *Computer program*
    - *Collection of instructions*
      - *process is the actual execution of those instructions*

- Concurrent programming
  - Systems can do more than one thing at a time
  - E.g. streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display.
  - The word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display.
  - Software that can do such things is known as *concurrent* software.

- In concurrent programming, there are two basic units of execution:
  - *Processes* and *threads*

Process

- An executing instance of a program.
- Has a self-contained execution environment
- Has a complete, private set of basic run-time resources
- Each process has its own memory space
- To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources
  - such as pipes and sockets
- IPC is used not just for communication between processes on the same system, but processes on different systems

Threads

- Sometimes called *lightweight processes*
- A **thread** is a single sequential flow of execution that runs through a program.
- Threads exist within a process
  - every process has at least one
- Unlike a process, a thread does not have a separate allocation of memory, but shares memory with other threads created by the same application.
- Thread context switching can be done entirely independent of the operating system.

Threads vs Processes

Processes

- Composed of 1 or more threads
- Have their own address space
- Communicate via message passing

Threads

- Share common memory and resources
- Can communicate via shared memory

Concurrency vs. Parallelism

- Concurrency is related to how an application handles *multiple* tasks
  - It may process one task at a time (sequentially) or work on multiple tasks at the same time - concurrently.

- Parallelism is related to how an application handles each *individual* task.
  - The task may be processed serially from start to end, or split it up into subtasks which can be completed in parallel.

- Parallelism
  - To achieve parallelism your application must have more than one thread running, or at least be able to schedule tasks for execution in other threads, processes, CPUs.

Threads

- Threads share the process's resources, including memory and open files.
  - This makes for efficient, but potentially problematic, communication

- Multithreading
  - You can have more than one thread running at the same time inside a single program
    - which means it shares memory with other threads created by the same application.

- Every application has at least one thread (started with `main()` in Java) — or several,
  - if you count "system" threads that do things like memory management and signal handling.

Why use threads?

- Threads can help in creating better applications, e.g.:
  - print in the ‘background’ (while editing);
  - scrolling through web pages texts while the browser is busy fetching the images;
  - a typical search engine would normally include many concurrent threads of execution, some serving its clients and others running web crawlers.
- Threads can quicken calculations
  - having different threads executing sub-tasks

Concurrent execution and context switching

- Two types of phases:
  - Compute bound phases - CPU is utilised for various calculations.
  - I/O bound phases – require the various input and output devices (printers, hard-disks, network cards etc.), CPU is mostly idle, waiting for the I/O device to do its task.

- Interleaving of phases.

- *Context switch*:
  - the process when one thread relinquish the CPU and another thread starts running it.

- Thread *context:*
  - each running thread has its own point of execution and private view of the values of local variables.

Why not use threads?

- Context switch is costly
- Extra CPU tasks:
  - ‘freeze’ and ‘de-freeze’ state of threads
- Single CPU multithreading can take more time than having linear code to return the same result

Single processor threading

- Threads with the same priority are each given an equal **time-slice** or time **quantum** for execution on the processor;

- **Pre-emption** (more urgent attention)
  - if thread is assigned higher priority.

Interrupts

- An interrupt is an indication to a thread that it should stop what it is doing and do something else.

- It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

- A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted.

Thread usage in distributed systems

- Provide a way of allowing blocking system calls without blocking the entire process in which the thread is running
  - Allows maintaining multiple logical connections at the same time.

Using threads at the client side (1)

- Multithreaded web client - hiding network latencies

- The usual way to hide communication latencies is to initiate communication and immediately proceed with something else.

- E.g. developing the browser as a multithreaded client
  – Web browser scans an incoming HTML page, and finds that more files need to be fetched (e.g. images)
  – Each file is fetched by a separate thread, each doing a (blocking) HTTP request.
  – As files come in, the browser displays them.
  – Some browsers start displaying data while it is still coming in.
    - While the text is made available to the user, including the facilities for scrolling and such, the browser continues with fetching other files that make up the page (e.g. images)
    - The user need thus not wait until all the components of the entire page are fetched before the page is made available.

Using threads at the client side (2)

- Multiple request-response calls to other machines (RPC)
  - A client does several calls at the same time, each one by a different thread.
  - It then waits until all results have been returned

Using threads at the server side

- Improve performance
  - Starting a thread is cheaper than starting a new process.
  - Having a single-threaded server prohibits simple scale-up to a
  - multiprocessor system.
  - As with clients: hide network latency by reacting to next request while
  - previous one is being replied.

- Better structure
  - Most servers have high I/O demands. Using simple, well-understood blocking calls simplifies the overall structure.
  - Multithreaded programs tend to be smaller and easier to understand due to simplified flow of control.

Threads usage: multiple servers

Example

- Web servers replicated across multiple machines, where each server provides exactly the same set of Web documents.

- When a request for a Web page comes in, the request is forwarded to one of the servers

- When using a multithreaded client, connections may be set up to different server replicas
  - allowing data to be transferred in parallel, effectively establishing that the entire Web document is fully displayed in a much shorter time than with a nonreplicated server.

- This approach is possible only if the client can handle parallel streams of incoming data
  - Threads are ideal for this purpose.

Summary

- Processes play a fundamental role in distributed systems as they form a basis for communication between different machines.
- Threads in distributed systems are useful to continue using the CPU when a blocking I/O operation is performed.
  - It then becomes possible to build highly-efficient servers that run multiple threads in parallel, of which several may be blocking to wait until disk I/O or network communication completes.
- In general, threads are preferred over the use of processes when performance is at stake.

Reference

- Chapter 3: Maarten van Steen, Andrew S. Tanenbaum Distributed Systems, 3rd edition (2017)

- Chapter 7: Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 5/E