1 Part IX: Concurrency

- Concurrency vs. Distribution, Parallelity.
- Processes & Threads.
- Synchronisation.
- Deadlocks.

2 Concurrency

- Concurrency means, that two or more things happen seemingly at the same time.
- Concurrency may be interesting already on one computer with one processor.
- The motivation is often that the user of some service wants to do many things at the same time.
- For example printing one file and editing another and running a simulation.
- It should seem to happen at the same time, but it's OK if the computer switches between the tasks fast enough.
- In the case of printing, editing and simulation this is also more productive.
- If the computer is just printing (or just running the editor), the CPU would spend some time waiting for the printer.

3 Distribution

- Distribution means that parts of a task happen on different machines.
- The motivation can lie in the distribution of the original problem (e.g. airline booking system, WWW)
- Another motivation may be saving resources (e.g. one printer/file server for everyone).
- If we have distribution we practically have concurrency.
- We may have concurrency without distribution.
- Additional issues in distribution are e.g. reliability, local/non-local data.
- The programmer has to care about the case, where the other computer is unreachable.
- The programmer has to take care on which computer which data is.

4 Parallelity

- Parallelity means that two things happen really at the same time, either on different machines or on more proccesors on one machine.
- The motivation is often a gain in efficiency.
  - numerical calculations
  - simulations
- Parallelity implies concurrency.
- We may have concurrency without parallelity.
- The programmer has to take care on which computer which data is.
- Here this is often very important for the efficiency.
- Typically the machines are close to each other, and e.g. reliability is not an issue.

5 Problems of Concurrency

- We will only deal with concurrency.
- Consider the program fragment:
  if (a != 0) { b = b / a; }.
- This is reasonable code in a sequential program.
- In a concurrent program at "the same time"
  a = 0
  might happen.
- Then a division by zero error would still occur.
- Problem: How do we deal with this?
- Once we add concurrency, this affects all the program!

6 Processes

- A typical example of a concurrent application is an operating system.
- The user of a computer wants to run more than one application at a time.
- There is more than one *process* running concurrently.
- A process is a program in execution. It is active.
- A *program* is a piece of executable code. I is passive.
- There might exist at the same time more than one process executing the same program (text editor).

7 Process States

A process can be in different states:

- *new*: This process is just newly created.
- *ready*: This process is ready to run and only has to wait for its turn to run.
- *running*: The process is actually running on the CPU right now. If there's only one CPU, only one process is running.
- *waiting*: The process is waiting for something to happen (i.e. user input). At the moment it couldn't possibly run.
- *terminated*: The process is done.

8 Process State Transitions

A process may change its state.

- new to ready: *admitted*
- ready to running: *scheduler dispatch*
- running to ready: *interrupt*
- running to waiting: *I/O or wait*
- waiting to ready: *I/O or wait completion*
- running to terminated: *exit*

9 Process Representation

A process is represented in the system by a *process control block*.

- process id
- process state (ready, running, ...)
- program counter
- CPU registers
- scheduling information
- memory managment information
- IO status information
- accounting information (CPU - time used)
- ...

The exact format depends on the operation system.

10 Scheduling

- Only one process is running (in a one processor system).
- There are typically many processes ready. We organize them in the ready queue.
- Also typically some processes are waiting for I/O. We organize them in a queue for each device.
- If a process is interrupted or has to leave the running state because it's waiting for I/O, we have to decide which process is running next.
- This is the task of the *scheduler*.

11 Scheduler

- The scheduler has to run often enough, to give the parallel impression to the user.
- If it runs too often, we spent too much time scheduling.
- The scheduler shouldn't run too long, because again we would spend too much time scheduling otherwise.
- It should run long enough to make reasonable decisions.
- We have to make a trade-off decisions.
- Some systems have more than one scheduler.
- The short term scheduler runs often and is very fast.
- The long-term scheduler runs less often and takes more time.
- There are often two kind of processes (I/O-bound and CPU-bound processes)
- It is the task of the scheduler to pick a good mix of them.

12 Context Switch

- Exchanging the running process is called a *context switch*.
- A context switch is pure overhead and should therefore be fast.
- It will typically take 1 to 1000 μs
- This is highly dependent on Hardware support.
- It is often a performance bottleneck.
- That is where threads were invented.

13 Threads

A thread is also often called a *lightweight process*.

- The idea is that some lightweight processes run within one process.
- Each thread has its own thread-id, program counter, registers.
- But all threads share code, data and other resources like open files.
- This is the basic difference to *heavyweight processes*.

14 Thread Advantages

- Economy
  - A context switch for threads is much cheaper.
  - The same holds for thread creation.
  - On Solaris thread creation is about 30 times as fast as process creation.
  - The context switch is about 5 times as fast.
- Responsiveness
  - An application can continue running, though part of it is blocking or performing a lengthy operation.

15 Thread Advantages (2)

- Resource sharing (code, memory, I/O)
  - We can have several threads in the same address space.
  - This allows easy and efficient communication between threads.
- On multiprocessor architectures threads may run on different processors.
  - Here, resource sharing is the natural model, because the processors actually share memory, ...

16 Threaded Applications

Many applications are multithreaded.

Often we want to do things concurrently within one application.

- A Web - Browser
  - Downloading a file
  - Displaying images
- A word processor
  - Building up a graphic
  - Reacting to keystrokes
  - Spell-checker
  - Resource sharing is important (one copy of the text).

17 Threaded Applications (2)

We may have many similar tasks.

- A Web - Server
  - One single threaded process would make response very bad.
  - New process for every request would create many processes with the same code.
  - Threads are faster and a better solution here.
  - Here we have resource sharing of code.

Java has other uses for threads.

- In Java there is no *asynchronous* behaviour.
- If a java thread connects to a server it will block until the connections is made.
- If we had only one thread, the whole application would block until the connection is made (imagine the server being down).
- The solution is to create a new thread that connects to the server.
- Another thread will check after a while whether the connection was made and interrupt the other thread in case of no success.

Q: Do we still need processes?

18 Process Advantages

- A process needs the code (program) for all of its threads
- Programs can be compiled separately.
- After recompilation of one thread, we have to restart the whole program.
- A process has separate memory, it may not change memory from another process.
- For distribution and parallelity this is the natural model.
- Largely independent tasks should be done by separate processes.

19 Process Applications

- Application programs.
- X-Server.
- Print-Server.

20 Java Threads

- Java has support for threads at a language level.
- Each Java program creates at least one thread (This could be only the thread running the main routine).
- Java provides classes and commands that allow the developer to manipulate threads.

21 Thread Creation

- The class Thread has a method start, which creates and starts a new Thread.
- **new** Thread() would just create a Java Thread object, but would not actually create a thread.
- When we write code for a thread we extend Thread and override its run method.

```
class SimpleThread extends Thread {
        public void run() {
                System.out.println("I'm a simple thread!");
        }
}
```


- the code for a program running a SimpleThread would look like that

```
class ThreadExample {
        static public void main() {
                SimpleThread st = new SimpleThread();
                System.out.println("I'm the main thread!");
                st.start();
        }
}
```


22 A Second Method

- Sometimes we want to make a class a separate thread, that already extends another class.
- In Java we can always only extend one class.

```
class SimpleRunnable extends Other implements Runnable {
        public void run() {
                System.out.println("I'm a simple thread!");
        }
}
```

```
class ThreadExample2 {
    static public void main() {
        SimpleRunnable sr = new SimpleRunnable();
        Thread t = new Thread(sr);
        System.out.println("I'm the main thread!");
        t.start();
    }
}
```