ECE 3574: Applied Software Design

Introduction to Concurrency: Processes and Threads

Spring 2017

Today we are going to introduce the notion of concurrency.

- Concurrency
- Engineering View: Operating Systems and Concurrency
- CS Theory View: Communicating Sequential Processes
- Examples

Concurrent Programming

Concurrency extends the *sequential* programming model

```
Do A then B then if C Do D Else Do E
------------------------------------> time
```

to include multiple executing computations or *processes*.
A concurrent model is a much more realistic model of the world.

Concurrency is a way to structure software, particularly as a way to write clean code that interacts well with the real world. – Rob Pike

Examples

- Your operating system (even on a single CPU)
- A web server responding to multiple requests.
- A graphical program responding to user input
- Numerical simulations
- Robot making multiple complex movements
- Control system in a car

Note: concurrent (virtual time) does not (necessarily) imply parallelism (real time)
On a single CPU a program may be concurrent, but cannot be parallel.
A concurrent program *might* take advantage of multiple CPUs though.

Concurrency is essentially an abstraction of time, or an ordering of events

```
T1 :>>>>>>>>>>>>>>>>>>>>>x
T2        :>>>>>X
T3                :>>>>>>>X
-----------------------------> time
```

Concurrency is essentially an abstraction of time, or an ordering of events

These events may be linked so there is a dependency

```
T1 :>>>>>>>>>>>>>>>>>>>>>x
T2        :>>>>>X
                 \
T3                :>>>>>>>>>>
-----------------------------> time
```

The input to T3 depends on the output of T2.

Two views of concurrency

- Engineering view: how do you implement concurrency
- CS Theory view: how do you *think* and reason about concurrent programs

Basic of Operating Systems: The process

A process is an abstraction by operating systems that allow it to *virtualize* a CPU.
This allows more than one program to *run*, that is appearing to *execute*, even with a single processor, a.k.a multi-tasking.
Thus, OS's were the first place concurrent programming was encountered.

(greatly) simplified execution on bare hardware (no OS)

- After power-on set PC to a specific address
- starts the fetch-decode-execute cycle from there
- continues until a halt instruction is executed, or the power is cycled.

An operating system is a program that creates a virtual representation (abstraction) of both hardware and time.

The OS is the code that starts running at power-up (or shortly thereafter in the case of BIOS) and

- Virtualizes CPU
- Virtualizes Memory
- Virtualizes IO Devices
- Virtualizes Time

The core of the operating system is the *kernel*

The kernel executes for a while, then might allow some non-kernel code (a program) to run by entering it's main function. When main returns the kernel picks up and keeps on going until another program starts. This is how I started programming (Sinclair,Commodore, Apple II)
Of course this requires the programs be really short, so instead programs were written to give up control periodically, or *yield*. The OS then put these programs in different blocks of virtual memory, execute one until it yielded, then execute another until it yielded and so on. This is called *cooperative multi-tasking*.
The earliest operating systems worked this way: DOS, Windows before Windows95, Mac OS $<=$ 9,

Preemptive Multi-tasking

Cooperative multi-tasking required cooperation among **all** programs running. One bad program failing to yield could lock up a computer, requiring a power-cycle.
So kernels evolved to enable them to interrupt another program in order to either execute itself, or to execute another program, possibly the one previously interrupted. This happened transparently to the program.

- This is called *preemptive multi-tasking* and is the dominant form of operating systems.
- The interruption and change of executing code is called a *context switch*.
- Note this removes the requirement of cooperation from individual programs but places increased responsibility on the OS to share time fairly.

The abstraction of a running program in an OS is a Process

```
   Memory                                        CPU
+-----------+ (low address)                   Registers
|   code    |                                    PC
+-----------+
|   heap    |
|    |      |
|    v      |
|           |
|  (free)   |
|           |
|    ^      |
|    |      |
|  stack    |
+-----------+ (high address)
```

Each process has its own memory region.

Processes are relatively expensive to create, the solution?

threads, sometimes called light-weight processes

- multiple threads of execution that have separate stacks, but share the heap
- much faster to start and stop threads because it just is a re-partitioning of the existing process memory

Side Note: the kernel may also be threaded, so there is (sometimes) a distinction between kernel threads and user threads.

The process abstraction changes to (two threads)

```
   Memory                                      CPU
+-----------+ (low address)                 Registers
|   code    |                                  PC
+-----------+                               Interrupts
|   heap    |
|    |      |
|    v      |
|    ^      |
|    |      |
| stack 2   |
|    ^      |
|    |      |
| stack 1   |
+-----------+ (high address)
```

The OS schedules both process and threads

The main kernel loop (in essence)

```
do{
  // choose new process or thread to run
  // save running process state (start context switch)
  // set a timer to interrupt (e.g. 8254 or HPET)
  // load new process state (end context switch)
  // it runs until timer interrupts
  // enters OS code and loops
}while(true);
```

Process versus thread

The essential difference between a process and a thread is how they communicate.
Process may communicate using

- Pipes / Sockets
- Message Queues
- Shared Memory

This is OS specific, but see `boost::interprocess`
Threads communicate using a shared heap, over which message queues and other forms of communication can be written. As of C++11 this is now standardized, but it used to be OS specific (win-threads versus pthreads).

An important aspect of concurrency at the OS level is the scheduler.

The scheduler is the part of the kernel that decides who runs during a given time-slice.

- Based on a priority system
- Based on what a process is doing
- Schedules both processes and threads within them.

Now for a different view: Communicating Sequential Processes

The theory behind concurrent programming dates back to Dijkstra, *Guarded Commands, Nondeterminacy, and Formal Derivation of Programs*, Comm. ACM 18(8):453–457 (1975).
Hoare, *Communicating Sequential Processes*, Comm. ACM, 21(8):666–677 (1978).
This is where much of the nomenclature and design patterns of concurrency come from.

CSP is a formal mathematical language

Given an event x and a Process P (x->P) describes an object that engages the event x (“if x”) then behaves as the process P, “x then P”.
Processes can be *composed*
(x -> (y -> P)) means “x then the process (y then P)”
can be recursive
P = (x -> P) means “x then x then x then . . . ”
and can make choices
(x -> P | y -> Q) “if x then P or if y then Q”

Example: Vending Machine

VendingMachine = (money (snack VendingMachine))
says
A vending machine receives the "money" event then does the process "give snack" event ad-infinitum.
In this view programs are *compositions* of processes and can be viewed as state diagrams.

Now, we allow the processes to operate in lock-step

(P || Q) means two processes P and Q synchronized also form a composed process.

Example: A Greedy Customer meets the Vending Machine

Customer = (snack -> Customer | money -> (snack -> Customer))
VendingMachine = (money (snack VendingMachine))
The greedy customer would take a snack for free if it can but would also choose to give money to obtain a snack.
When these two processes execute together
(Customer || VendingMachine)

The laws of CSP

The CSP theory goes on to establish logical laws about the state of the world described by a CSP.
If the event sets of two executed processes overlap then they are communicating.

How we will view concurrency

We will take a more practical engineering view of concurrency. We will not discuss CSP much more, but I want you to be aware that the design patterns we will use originate from there.

Next Actions and Reminders

- Read about Qt inter-process communication
- Project 2 beta due Tuesday at 8 am.