Csci 4061 - Meeting 1

- *Administrative*
  * Introductions
  * Syllabus
  * Waiting List Policy
- *Goals:*
  * Understand concurrency
  * Overview of the course
- *Topics:*
  * 1.1 Multiprogramming and Multitasking
  * 1.2 Concurrency at the Application Level
  * 1.3 Unix Standards
  * 1.4 Programming in UNIX
  * 1.5 Making functions safe
- *Readings: Chapter 1 (Robbins, pp.76-137)*
- *Readings: Appendix A.1-3 (Robbins, pp.577-589)*
- *Recommended Exercises:*

Administrative

- *Introductions*
  * Instructor
  * TAs
  * Peers
- *Syllabus*
  * Schedules: lectures, homeworks, exams, recitations
  * Textbooks, Reference material
  * Topics
- *Policies*
  * Late homeworks
  * Cheating
  * Waiting List - Attendance*, estimates

Course Goals

- *Concepts: Understand concurrency*
  - * Why concurrency?
  - * Sources of Concurrency
  - - I/O, signals, processes, threads, client-server
  - * Effects of concurrency
- *Focus*
  - * Server - software concurrently shared by many
  - * User level - commands, shell
  - * Power Users - system calls, C programs
- *Out of Scope*
  - * Operating System Theory - e.g. CPU scheduling
  - * Vendor specific features - e.g. Win32

1. What is Concurrency?

- *Concurrency:*
  * Sharing of resource in the same time-frame
  * Ex. two program executing concurrently
  * Q? Which resources are they sharing?
- *Trends leading to Concurrency*
  * Computer speed >> Human typing speed
  * CPU speed >> I/O (e.g. disk drives)
    - See Table 1.1 (pp. 5)
  * Multiprocessors
  * Distributed Systems
  * Graphical User Interfaces
    - Animation of multiple objects
- *What is hard about Concurrency?*
  * Non-deterministic behaviour
  * Bugs do not show up on a regular basis

1.1 Multiprogramming and Multitasking

- *Multiprogramming*
  * Process: instance of a program in execution
  * More than one processes can be ready to execute
  * OS choose one to execute
  * Context switch to another process when
  - this process needs I/O
- *Q? What if a program has an infinite loop?*
- *Timesharing*
  * Context switch to another process when
  - when quantum is over
  * Pros: Reduce waiting time for small jobs
  * Cons: overhead of context switch
- *Multitasking - Similar to multiprogramming*
  * finer granualarity (e.g. threads within a process)
  * Sharing even user resource, e.g. global variables
- *Why do I care about these? I am not writing an O.S.!*
  * Web servers: search engines, databases, e-commerce

1.2 Concurrency at the Application Level

- *Concurrency Levels*
  - * Hardware
  - - CPU controlling peripherals, multi-processors
  - * Software - OS
  - - signal handling
  - - overlap of I/O and processing
  - - communication
  - - resource sharing among processes and threads
- *Outline*
  - * 1.2.1 Interrupts (Recall assembly language course)
  - * 1.2.2 Signals
  - * 1.2.3 Input and Output
  - * 1.2.4 Threads and Resource Sharing
  - * 1.2.5 Network as the Computer
- *Q? Map these to chapters in the book.*

1.2.1 Interrupts

- * Peripheral generates an electrical signal
- * Sets a flag in CPU
- * CPU checks flag in each instruction cycle
- * Interrupt service routine called

• *Example: Timesharing implemented with*

- * alarm interrupts

• *Concurrency: CPU and peripheral device*

- * Shared resource - bus

• *Event types by time of occurrence*

- * Asynchrounous - time not determined by receiver
- * Synchronous - time determined by receiver

1.2.2 Signals

- *Motivation*
  * Q? How do you stop a program in an infinite loop?
  * Other usage: timers, job control, aynch. I/O, ...
- *Signal = software notification of an event*
  * Ex. hardware events, e.g. ctrl-c, I/O complete
  * Q? Provide examples of synchronous signals.
- *Life cycle of a Signal*
  * Event of interest occurs
  * Signal is generated
  * OS sets a flag for the relevant process
  * Signal is caught by the process
  * Process invokes a handler subroutine
  * Analogy - "You have mail" flag
- *Concurrency: main program, signal handler subroutine*
  * Implication: restriction on signal handler
  * Sharing a global variable => special protection

1.2.3 Input and Output

- *Motivation*
  - * Coordinate resources with varying speed
  - * But isn’t that the job of O.S.?
  - * Why should an application developer learn this?
  - * You may develop performance critical applications
  - - Ex. real-time - Pacemaker
  - - Ex. Web servers, transaction processors - ebay, amazon, ...
- *Ex. asynchronous I/O*
  - * A process itseld can do other things
  - * while waiting for an I/O, i.e. synchronous read()
  - * instead of getting swapped out by OS
- *Ex. monitoring multiple input source on network*
  - * Standard blocking I/O is not suitable!
- *Concurrency*
  - * Subprogram handling file/network I/O
  - * Subprograms computing during wait for I/O

1.2.4 Threads and Resource Sharing

- *Motivation* - *What is the unit of concurrency?*
  * Traditional unit = process
  * Emerging finer unit = thread
- *Processes* - *Generated via fork() call*
  * Coordinate termination via wait()
  * Communicate via pipes (common ancestors),
  - or signals, messages, shared memory, etc.
  * Pros: stronger security boundaries
  * Cons: high overhead
- *Threads* - *provide concurrency within a process*
  * threads of execution = program counter value streams
  * Finer level of concurrency
  * Low overhead in creating and context switching
  * standards are emerging now!
- *Concurrency*
  * Multiple processes or Multiple threads within a process

1.2.5 Network as the Computer

- *Motivation - internet!, intranet, networks, ...*
  * Multiple services: ftp, email, ...
  * Million of clients accessing Web services
- *Client-Server = A model of distributed computing*
  * Client = caller of a service
  * Server = provider of a service
  * Analogy with procedure call, caller, callee
- *Details*
  * Clients and Servers may be on different machines
  * Communication via messages or remote procedure calls
  * Signals, Pipes, shared memory are not common
- *Concurrency*
  * Server and client are concurrent
  * Multiple Servers and multiple clients

1.3 Unix Standards

- *Why Standards?*

  * Multiple flavours of Unix: HPUX, Solaris, Linux, ...
  - Two distinct lineage - BSD and System V

  * Non-Unix OS: NT, Windows 3.1/95/98/..., MacOS, ...

  * System calls are often OS specific!

  * Overhead of porting across OS.

- *Which Standards?*

  * ANSI C

  * POSIX - IEEE Portable Operatig System Interface
  - Table 1.3 provide POSIX standards

  * if not covered by POSIX
  - Spec 1170
  - System V Release 4

- *How do I check POSIX support in my OS?*

  * unistd.h header file

  * Table 1.4 shows the compile time options

1.4 (Concurrency) Programming in UNIX

- *Concurrency programming*
  - * Language constructs, e.g. Java
  - * OS Libraries, e.g. Unix system call
- *System call* - *a procedure provided by OS*
  - * An entry into the kernel (heart) of OS
  - * To get access to system resources
- *Standard C Library*
  - * e.g. string handling, memory management
  - * Some subroutine contain system calls
  - * Hard to tell the difference from system call!
- *Resources* - *Unix man pages (Appendix A.1)*
  - * header files needed by system call
  - * prototype of system call- name, parameters
- *Apendix A.1-3commands: man, cc, make*

1.4 Programming in UNIX

- *Conventions- error situation*

  * system call returns -1 or NULL

  \- Sets global variable "errno" to error code

  * Application programmer should check for these

  \- perror() - Example 1.2 (pp. 15)

  \- strerror() - Example 1.5 (pp.16)

- *Newer Style - use exception handling (C++, Java)*

\- new system calls return error code as result

\- avoid global variables, e.g. "errorno"

- *Other conventions - (See bullets on pp. 16-17)*

  * Q? Identify 3 bullet related to concurrency?

  * Q? Which bullet relates to memory leaks?

  * Q? List problems with global variable "errno" .

1.4 Programming in UNIX

- *Extended example - argument arrays!*
  * Review pointers, argv[], argc, parameter passing
- *Ex. Review Program 1.1 and 1.2 to answer the following:*
  * What are argv[] and argc used for?
  * What is the parameter passing mode in C?
  * What are the data types of arguments to makeargv()?
  * What does makeargv() return?
  * List a few possible error situations for makeargv().
    - How does makeargv() respond to those errors?
  * Is it possible to rewrite makeargv() with following header?
    - Headers from Example 1.8, Example 1.10

    ```
    int makeargv(char *s, char *delimiters, char **argvp)
    ```
  * What is maximum number of arguments allowed?
  * Is there any memory leak? Justify your answer.
    - Consider memory allocated to ’t’ and ’*argvp’
  * What the following loop do?

    ```
    for (i=1; i< numtokens + 1; i++)
            *((*argvp) + i) = strtok(NULL, delimiters);
    ```
  * Why is the above loop not followed by free(t)?

1.5 Making functions safe (for reentry)

- *Non-Reentrant functions*
  * Self modifying code
  * functions using static/global variables
  * Problems with multiple simultaneous invocations
- *Reentrant functions*
  * Allow multiple simultaneous invocations
  * Needed for signal handler, server with many clients, ...
  * Two aspects -
  - Thread safe: can be called concurrently by 2 threads
  - Async. Signal safe: can be called inside a signal handler
  - without restriction

1.5 Making functions safe (for reentry)

- *Q? Which POSIX system calls thread safe?*
  - Not those using global variable "errno", e.g. read()
  - reentrant functions provided for non-reentrant ones
  - Ex. strtok_r() for strtok()
  - Trend towards thread safe system calls!
- *Q? Which POSIX system calls async signal safe?*
  - See Table 5.3, pp. 191
  - Double check with man page on your system!
- *Q? Is makeargv() (Program 1.2) a reentrant function?*
  - Is it signal safe? Is it thread safe? Why?
  - How can you make it thread safe?