Week 11: Testing multithreaded programs

- Background
- Origins
- Overlapped I-O
- Multiprocessing and multitasking
- Testing practices and difficulties

Von-Neuman machine model (ca. 1954)

Efficiency drawback of early strictly sequential process

- The problem:
  - CPU operations measured in milliseconds or faster.
  - Magnetic tape read or write an order of magnitude slower.
  - Unit-record (card-reader, printer) devices slower still.

- Many programs were
  - I-O bound some of the time: program waiting for input or for place to store output.
  - compute bound some of the time: no I-O to be done.

  Wasn’t there a more efficient way?

Early parallel process: overlapped I-O (ca. 1958)

- The solution:
  - I-O devices attached to data channels.
  - CPU starts a read or write operation to move data between main memory and external medium.
  - CPU can work on something else while the I-O is taking place.
  - CPU can test whether channel is busy.

- Sequential output: buffer areas in main memory queued for writing.
- Sequential input: anticipatory buffering before program needs the data.
What could go wrong with that?
- Program bug could overwrite output buffers in memory before they got written.
  - If debugging output was buffered, it could be hard to determine a point of failure.

Why?
- Reliably exploiting overlap required more complicated operating systems:
  - Some applications bypassed I-O buffering
  - Early operating systems had bugs

Historical note
- IBM models for scientific computing:
  - 704 (1954): I-O not overlapped
  - 709 (1958): overlapped data channels
  - 7090 (1960): data channels with interrupts

Why is this interesting today?
- Operating system I-O components designed to work on either 709 or 7090
  - How could they do that?
  - How does that affect testing?
  - Does anyone bother today?

An incremental improvement: I-O interrupts (ca. 1960)
- Instead of the CPU having to test repeatedly whether a channel is busy, the channel would interrupt the normal instruction sequence when a transmission was completed.
- That made operation potentially even more efficient, but also more error prone:
  - Critical parts of the operating system had to protect themselves against being interrupted
  - Some O.S. code had to be re-entrant.

For the first time timing dependencies could affect software reliability.

Re-entrant code
- A subroutine that could be called by one task while it was executing for another task.
  - Initially that was only to handle I-O interrupts while a CPU program was executing
  - Later, with multiprogramming (ca. 1965), there might by dozens of tasks potentially calling the same routine.

Rule:
- A re-entrant routine must not store any information in internal private or static storage

Are re-entrant routines recursive?
Are recursive routines always re-entrant?
Can a routine be both?
Two breakthroughs (ca. 1965)

- **Multiprocessing** hardware (often called *multicore*)
  - Multiple CPUs sharing memory
  - Dispatcher decides which process to assign to which processor

- **Multiprogramming**
  - Independent jobs
  - Operating system supports memory protection and paging
  - Multitasking (or multithreading) within a job
    - Parallel algorithms
    - Foreground user interface, background processing (common for GUI, esp. Internet, applications)

Why bother?

- Application will still work if "foreground" process just waits for the "background" activity to finish (i.e. no multitasking)
- It will take about the same elapsed time

- But note negative impact on users:
  - High-volume users can be entering data for next transaction or doing something else to get ready to act on information being retrieved or computed in the background.
  - Users will get impatient and may wonder whether the application is still running.
    (cf. progress indicators)

What about games?

But always have a reason!

- Multicore machines can be often exploited by the operating system running multiple concurrent jobs.
- Therefore, it's not always necessary to exploit multitasking within a single job in order to use the hardware efficiently.
  - Multitasking may make debugging much more difficult.
  - Your test worked fine, but users complain of failures or mysterious results *now and then*.

- Two justifications for multitasking:
  - Enhance usability, user-friendliness
  - Speed up a very time-consuming process (e.g. sort, matrix multiply, search)

Design for testing

- Will it still work with interrupts disabled?
- If so, will it still work without multi-tasking?
  - i.e. finish executing child process before proceeding with parent process.
- If so, initial testing should validate that the program works correctly in that disabled mode.
  - If it does then try enabling multitasking without interrupts.
  - If that works then try enabling interrupts

Is that always possible?
Vary the environment

- Try to expose timing dependencies by running tests:
  - On several hardware configurations of different sizes and speeds
  - With varying concurrent workloads.
  - With and without extra debugging code.

- If all those appear to work, does that validate the MUT?
  - No! Multithreading is complicated and subtle.
  - Symbolic execution and rigorous desk-checking by diabolically-inclined testing specialists is strongly advised.

Language support

- Early programming languages (Fortran, Cobol, Algol-60) ignored multitasking, except (sometimes) for overlapped ("buffered") I-O
  - If you wanted to exploit parallelism at any level you had to use either an assembly language or an experimental higher-level language (e.g. University of Illinois Fortran, Ferranti Algol)

- PL/I (1964) Algol-68, & Ada supported multitasking
  - Relied on new operating system support
  - Few took advantage of it.

- C/C++ relied on third-party libraries until recently

Java support

for multithreading

- Java initially provided
  - `synchronized` attribute to support locking, exclusive access, etc.
  - `Thread` class and `Runnable` interface to construct a thread.
  - `wait()` and `notify()` methods

- Early Java Zealots claimed "Java is the first programming language to support multithreading as a built-in feature."
  - Wrong! But language and library features made it easier than in older languages.

New Java support

"In release 1.5 a set of concurrency utilities was added, in the package `java.util.concurrent`. This package contains both high-level concurrency utilities to simplify the task of multithreaded programming and low-level concurrency primitives to allow experts to write their own high-level concurrent abstractions. The high-level parts of `java.util.concurrent` should also be part of every programmer’s basic toolkit.


  - Chapter 13: "Doing several things at once: Threads",
  - Chapter 14: "Advanced Thread Topics"
Textbook support

- Pezze & Collins provide little if any help
  - Sidebar, p. 356, "Concurrency Faults" says it's hard.
- Surprisingly the index contains no entries for
  - multiprogramming
  - multi-tasking
  - multi-threading
  - overlap
  - parallel
- That's OK.
  - Many programmers live their whole careers without having occasion to need it.

More Java help

- In addition to the excellent books by Bloch and by van der Linden, you can find web tutorials, such as these:
- Emphasis is on how to design & code programs more than on how to validate them.
  - You'll find some information on the web, but much of it promotes products.
- If you know of or discover something useful, please share it with the class.

Week 11: Testing user interfaces, esp. GUI

Does it work?
Is it friendly?
Does it adhere to design standards?

A problem

- It's hard to distinguish between
  - a. design quality, ease of use
  - b. testing, correctness
- The program may work (produce correct results) but still be unacceptable.
- What are the responsibilities of
  - the systems analyst?
  - the programmer?
  - the users?
  - professional testers?