Threads

CS449 Fall 2015
Project 2 Statistics

- Max: 115
- Min: 38
- Average: 91.29
- Median: 96
Concurrency and Parallelism

- Many programs need to perform mostly independent tasks that do not need to be serialized
  - Web server: page requests
  - Text editor: auto-save, spell checking, text entry
  - Web client: tabbed browsing

- **Parallel Programming**
  - Make multiple runtime copies of the same task
  - Underlying system runs each task on a separate CPU
  - For performance
  - Makes sense only on a multiprocessor machine

- **Concurrent Programming**
  - Divide program into multiple, generally different, tasks
  - Underlying system provides illusion of concurrent progress
  - For responsiveness (i.e. spell check does not delay text entry)
  - Can make sense even on a single processor machine

- The need for parallelism has become more acute with multicores
Concurrent Example

(Word Processor)

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the pursuit of Happiness. — That to secure these rights, Governments are instituted among Men, deriving their just powers from the consent of the governed. — That whenever any Form of Government becomes destructive of these ends, it is the Right of the People to alter or to abolish it, and to institute new Government, laying its foundation on such principles and organizing its powers in such form, as to them shall seem most likely to effect their Safety and Happiness. Prudence, indeed, will dictat that Governments long established should not be changed for light and transient causes; and accordingly all
Parallelism Example (Webserver)

```
while (TRUE) {
    getNextRequest(&buf);
    handoffWork(&buf);
}
```

```
while (TRUE) {
    waitForWork(&buf);
    lookForPageInCache(&buf,&page);
    if (pageNotInCache(&page)) {
        readPageFromDisk(&buf,&page);
    }
    returnPage(&page);
}
```
Programs and Processes

- **Program**: Executable binary (code and static data)
- **Process**: A program loaded into memory
  - Program (executable binary with data and text section)
  - Execution state (heap, stack, and processor registers)

```c
int foo() {
    return 0;
}

int main() {
    foo();
    return 0;
}
```
**Context Switching**

- OS provides the illusion of a dedicated machine per process
- **Context switch**
  - Saving context of one process, restoring that of another one
  - Processor alternates between executing different processes
Multiprogramming: Concurrently Running Programs

Single $PC
(CPU’s point of view)

Multiple $PCs
(process point of view)

Isolated address spaces
that are not interrupted

One stream of instructions
Dispatch Mechanism

• **OS maintains list of all processes (PCBs)**
• Each process has a mode
  – **Running**: Executing on the CPU
  – **Ready**: Waiting to execute on CPU
  – **Blocked**: Waiting for I/O or synchronization with another thread
• **Dispatch Loop**

```plaintext
while (1) {
    run process for a while;
    stop process and save its state;
    load state of another process;
}
```
Life Cycle of a Process
Processes Sometimes Infeasible

- Expensive in terms of memory
  - Must maintain PCB for each process in OS
  - In particular page tables

- Expensive in terms of performance
  - Forking: must duplicate PCB (including page tables)
  - Context Switching: involves system call into OS and flushing cached page table entries in MMU
  - Inter-process Communication: involves system call to invoke OS to move data

- When memory needs to be *shared* instead of *duplicated*:
  - When there is a shared modified data structure
Thread

A stream of instructions and their associated state
Threads and Processes

- Process: Execution stream + Program State
- Thread: Execution stream + Thread State (in same process address space)
  - “Lightweight process”

- Can have multiple threads within a process
Processes and Threads
Thread State

Per process items
- Address space (page table)
- Open files
- Child processes
- Signals & handlers
- Heap space
- Global variables

Per thread items
- Program counter
- Registers
- Stack & stack pointer
- State
Thread Pros/Cons

- **Pros**
  - Less memory to maintain Thread State (mostly extra stack space)
  - Less performance overhead
  - Automatic data sharing (same address space)

- **Cons**
  - Shared memory can be source of data races
  - Less robust compared to processes:
    - Memory corruption on one thread corrupts entire program
    - A crash in one thread crashes the entire program
Data Races in Multithreading

- Global variables and heap locations are problems
  - Not just your global variables, also library variables
- File operations modify `errno` to indicate errors
  - What if two threads both operate on separate files?
Re-entrant Code

• **Re-entrant**: ability to handle multiple calls to the same function at the same time

• Many libraries are not re-entrant

• Example a library that formats and prints messages to a log file:

```c
int print_log(char * msg) {
    fprintf(logfile, "%s", get_timestamp());
    fprintf(logfile, msg);
    fprintf(logfile, "\n");
}
```
Thread Implementation

User-Level Threading vs. Kernel-Level Threading
Who does thread management?

- **User-level threading**
  - User-level threading library
  - On thread create
    - Reserve new stack space using `mmap()`
  - On context-switch
    - Save / restore processor registers
    - Change stack pointer registers to stack of new thread

- **Kernel-level threading**
  - Kernel maintains thread state internally
  - Use system calls for thread creation / management
User Threads vs. Kernel Threads
User-level vs. Kernel-level

• User-level threading can be faster:
  – Do not need system calls for thread management
  – Context switching happens in user land
  – Can make application specific scheduling decisions for a tighter schedule

• However some issues with user-level:
  – Cannot run on multiple processors
    • To the kernel, it is still just one process
  – One thread can hog CPU with no OS intervention
  – If a thread performs I/O and blocks, all threads stop
    • The entire process transitions into a blocked state
CPU Hogging Solution

• Thread library provides an `yield()` function that allows thread to voluntarily give up CPU
  – Also called cooperative threading
  – Since within same process, no worry about malicious hogging by a third party
  – Voluntary yielding often results in a better schedule than blind OS scheduling

• To pre-empt thread, register a timer signal to fire after time slot
  – Perform context switch in signal handler
Blocking I/O Solution

• Avoid performing blocking I/O in threads
• Threading library provides own version of non-blocking I/O calls
  – Poll() and select() system calls inform whether an I/O on a file descriptor would block or not
  – Always check with poll() or select() before performing actual I/O
  – If I/O will block, sleep thread and schedule another thread
Multiple Processors Solution?

- None: Can’t allocate CPU with no OS involvement
- Prevents user threading from being employed in an environment where parallelism is important (e.g. Servers)
- For small uniprocessor systems, it’s perfectly fine (e.g. Embedded Systems)
  - Memory efficiency of user threading a plus
  - Fast context switches and cooperative scheduling help meet real-time constraints
  - Works even on simple kernels with no threading support
OS Threading Requirements

• User-level threading:
  – Support for non-blocking I/O in OS

• Kernel-level threading:
  – Thread management implementation in OS

• Most general purpose kernels support kernel threading
  – Linux Native POSIX Thread Library
  – Windows Threading API

• How about Java? Usually the Java Virtual Machine…
  – Maps threads to kernel-threads if it’s supported
  – If no support for kernel-threads, uses user-level threading