Threads

CS449 Fall 2017
Parallelism and Concurrency

• Often, multiple tasks can be performed simultaneously
  – Web server: multiple page request tasks
  – Text editor: auto-save, spell checking, and text entry tasks

• Parallelism
  – Running multiple tasks on multiple CPUs to improve performance
  – Goal: finish given computation faster
  – E.g. Spell checker analyzes each page on separate CPU

• Concurrency
  – Running multiple tasks, appearing to make simultaneous progress
  – Goal: make program more responsive
  – E.g. Spell checker should work without holding up text entry task
  – Can achieve concurrency with single CPU (through multitasking)
Parallelism Example

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

while (TRUE) {
    waitForWork(&buf);
    lookForPageInCache(&buf, &page);
    if (pageNotInCache(&page)) {
        readPageFromDisk(&buf, &page);
    }
    returnPage(&page);
}
```
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 dictate that Governments long established should not be changed for light and transient causes; and accordingly all

Kernel
How should Tasks be Created?

• We already learned one way: fork()
  – Uses a process as a task container
  – Can be used for both parallelism and concurrency
  – But sometimes not the best task container

• Threads: alternative task container to processes
Task Creation using Fork()
Fork() Involves Cloning

- (Logically) Entire process state is cloned
  - All data in memory
  - All data in CPU registers
  - All OS state (page table, file descriptors table, etc.)

```c
int main() {
    if (fork() == 0) {
        // Child process
    } else {
        // Parent process
    }
}
```

Parent Process

- Heap
- Stack
- Registers
- OS State

Child Process

- Heap
- Stack
- Registers
- OS State
Processes Sometimes Infeasible

- Expensive in terms of **performance**
  - Forking: copy-on-write allows lazy copying of pages, but still must duplicate page table immediately as well as other OS state
  - Context Switching: must flush cached page table entries in MMU

- Expensive in terms of **memory**
  - Must maintain OS state for each process, including page table
  - All modified pages have to be duplicated

- Memory sometimes needs to be **shared**, not cloned
  - When there is a data structure needing to be shared by all tasks
  - Changes made in one task must be made visible to another task (Impossible for processes with segregated memory spaces)
Thread

A stream of instructions and its associated state
Processes and Threads

- By default, there is only one thread per process
  - Only a single stream of instructions pointed to by $EIP$
- Multi-threaded programs have multiple threads per process
  - Multiple simultaneous points of execution in program
Threads and Processes

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

- **Thread State**: Bare minimum required to support a stream
  - Thread is also called a “lightweight process”
# Thread State

## Per process items
- Address space:
  - Heap
  - Global variables
- OS state (PCB):
  - Page table
  - Open files
  - Child processes

## Per thread items
- Stack
- Registers($EIP$, $ESP$, …)

- Stack
- Registers($EIP$, $ESP$, …)

- Stack
- Registers($EIP$, $ESP$, …)
Thread Pros / Cons

• Pros
  – Less memory state (just needs extra stack space)
  – Less performance overhead
    • No page table duplication on thread creation
    • Less costly context switches (just save/restore registers)
  – Can share data through heap / global variables

• Cons
  – Crash for one means crash for all
    • A segfault in one thread will crash the entire process
  – Shared data means less isolation
    • One thread can write to shared data causing another thread to malfunction (a malfunction is not isolated within a thread)
Malfunction due to Shared Data

- Globals and heap are potential problem locations
  - Not just your global variables, also library global variables
- C Library file I/O modifies global \texttt{errno} to indicate errors
  - What if two threads operate on files simultaneously?

- C Library creates thread-local copy of \texttt{errno} per thread
Thread Implementation

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

• User-level threading
  – User-level **threading library**. No kernel privileges are required:
    – On thread create
      • Reserve new stack space using `mmap()`
      • Update stack pointer to point to new stack space
      • Update instruction pointer to point to beginning of thread
    – On context-switch
      • Save / restore processor registers
        (Will switch instruction pointer, stack pointer in the process)

• Kernel-level threading
  – **Kernel** maintains thread states, just like process states
  – Use system calls for thread creation / management
User Threads vs. Kernel Threads

- **Kernel threads (right):** kernel sees the threads
  - Kernel can run each thread on a different CPU
- **User threads (left):** kernel is not aware of threads
  - All threads within one process must share a single CPU
User Threads vs. Kernel Threads

- User-level threading has its advantages:
  - Does not require any OS support
    (Can even run on very simple kernels without multitasking!)
  - Applications have full control over scheduling decisions
    (Can use app-specific knowledge to achieve better schedule)
- But also some issues:
  - One thread can hog CPU since there is no OS intervention
  - If one thread performs I/O and blocks, all threads stop
    - The entire process transitions into a blocked state
  - Cannot run on multiple CPUs
    - To the kernel, it is still just one process
CPU Hogging Solution

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

• To guarantee thread pre-emption without \textit{yield()} calls, 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
- `Poll()` and `select()` system calls inform whether an I/O on a file descriptor will block or not
- Provide wrapper non-blocking I/O functions for all blocking I/O functions using `poll()` or `select()`
  1. Check with `poll()` or `select()` before performing I/O
  2. If I/O will block, yield and schedule another thread
  3. If I/O will not block, call the blocking I/O function
Multiple Processors Solution?

• None: Can’t allocate CPU with no OS involvement
• Cannot use user threads to achieve parallelism
• But *can* use user threads to achieve concurrency
  – Can be better than kernel threads for response times:
    • Cooperative scheduling often results in a tighter schedule
    • Context switches are much faster (do not require trip to kernel)
  – Often used in single-CPU embedded systems
    • To achieve concurrency w/o violating real time constraints
    • Ability to run on simple kernels (w/o even a scheduler) is a plus
OS Threading Requirements

• User-level threading (very light support):
  – Support for non-blocking I/O in OS

• Kernel-level threading (more support):
  – Support for multitasking in OS
  – Support for thread management in OS

• Most general purpose kernels support kernel threading
  – Allows use of multiple processors using threads
  – E.g. Linux Native POSIX Thread (LNPT) Library
  – E.g. Windows Threading API

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