Threads

Chapter 4

Processes

- When creating a process, the child process gets its own resources:
  - Address Space (global variables, files, etc)
    - heap
  - "Thread" of execution
    - Program Counter
    - Stack
- This can be a good or bad thing…

Inter-Process Communication

- Two common types:
  - Shared Memory
  - Message Passing

- POSIX Pipes
  - Typically implemented using shared memory
  - FIFO, one way communication
  - Also partially supported on Windows

PIPES

```
int pfd[2];  // file descriptors

int result = pipe(pdf);
```

After pipe call pfd[0] contains descriptor for read end of pipe, pfd[1] contains the descriptor for write end.

Use read and write system calls to read and write pipe. These calls require an fd, which is the file descriptor (pfd[0] for read, pfd[1] for write.

```
int read(int fd, char * buffer, int maxlen)

int write (int fd, char * buffer, int numBytes);
```

Exec System Call

- Having two processes active in the same code causes confusion:
  - Rarely used.
- Normally when processes are created, they are created to run **different** programs.
- POSIX: exec*
- Win32: CreateProcess (Windows does not provide a separate fork)

Program 4

- Echo
  - *Inter-process communication with separate programs*

Threads

- Can be considered *lightweight* alternative to using multiple processes
- Threads **share** global address space, heap, and code
- User v.s Kernel Threads
  - Pthreads, Win32, Java

Thread Resources

- One process can contain many threads
  - Each thread has its own execution context
  - All threads share the same address space and resources
- Advantages: Easier to work together
- Disadvantages: OS does not provide protection among common threads!

Scheduling Threads

- We know an OS scheduler switches between "ready" processes. How does it determine which thread to run?

Depends on the implementation:

Kernel Threads v.s. User Threads

User Threads

- User Threads: The OS knows nothing
  - Implemented by a runtime system, linked to the program code itself.
  - This runtime is almost like a tiny OS!
  - Used to be common (dominant), no longer.
  - Scheduling is not pre-emptive.
  - Major advantage: Extremely fast.
  - Major disadvantages:
    - Multi-processor?

Kernel Threads

- Kernel threads are managed by OS:
  - OS contains thread tables, indicating how many runnable threads are in the process
  - OS Schedules not only process, but choses among threads
  - Switching threads is more costly (why?)
  - Advantage:
    - Multiple threads can run on separate processors
    - One thread blocks on read, others can continue processing.

Why use threads?

- We need to remember programs usually cycle between CPU use and I/O use
  - The overall system is better utilized if CPU **AND** I/O are always busy
  - Your program is optimized if it can use the CPU **and** I/O simultaneously!
- This optimization is only available when using kernel threads (why?)

Classic Programming Models

- Multi-Threaded Web Server
- Multi-Threaded User Interface
- Multi-Process distributed system
  - Multiple machines may/may not be involved

More considerations

- Threads are faster to start/create than processes
  - Very fast for User Threads
  - Fast for Kernel Threads (less is copied)
- Thread Pools
- Termination
- Synchronization and Data Protection
  - Chapter 6

First: Review of Function Ptrs

- In both POSIX and Win32, creating a thread requires you to provide a function pointer
  - CreateThread calls this function (with whatever parameters you give it) as soon as it creates the new thread
  - Just like variables, functions have "types"
    - Type equivalent to function *signature*
    - The function's name is a pointer to "code" instead of data

Function Pointers

```
// Two function, both with the same signature
int add(int a, int b);
int sub(int a, int b);
void print(int a, int b);

// Function that takes a and b and performs the operation
// by executing the function provided:
int execute(int a, int b, int (*function)(int, int) ) {
    return function(a, b);
}

// a call to the execute function:
execute (7, 6, sub);
execute(7, 6, print);  // print doesn't have the required signature
```

Thread Libraries

- pthreads: POSIX standard
  - C procedure interface
  - Kernel or User depending on underlying OS
- Win32 Thread Model:
  - Kernel threads

Pthreads

- Must include phtread.h
- Pthreads start in a user defined function with specific signature:
  - void * function_name(void * param)
- Functions:
  - pthread_attr_init (set thread attributes to defaults)
  - pthread_create (create thread)
  - pthread_join(id) (wait for thread (id) to terminate
- compile with -lpthread option
  - g++ -o myprog myprog.cpp -lpthread

Program

- Pthreads

Win32 Thread API

- Windows uses HANDLES to represent system-wide identifiers:
  - processes, threads, files, etc.
- HANDLE CreateThread(…)
  - security attributes (NULL)
  - default Stack Size (0)
  - thread function
  - parameters to thread function
  - creation flags (0)
  - &thread identifier (DWORD, not really used)

Win32 API

- Thread function has strange signature:
  - DWORD WINAPI functionName(LPVOID parameters)
- To wait for a thread to terminate, you specify the handle:
  - WaitForSingleObject(ThreadHandled, INFINITE)
    - You can also specify milliseconds to wait…
  - After thread has terminate, use CloseHandle() to clear up resources

Program

- Simple Win32 example
- Thread Summation Example

Threads in Java

- Java also supports threads - in fact, quite elegantly
  - Objects can implement the *Runnable* interface
    - required to have a void run() function
  - The object can be run as a separate thread (starting in the run function)
    - Member variables are shared.

Complications

- Do Processes inherit all threads?
  - Only an issue with POSIX, since Win32 always executes a new program image
  - What thread receives keyboard / mouse input?
    - What thread receives any OS signal?
- Typical Solution: Its up to the programmer…

Summary

- Multi-Process Programs:
  - Increases parallelism
  - Good when a job can be broken into multiple *independent tasks*
- Multi-Threaded Programs:
  - Increases parallelism
  - Good when job can be broken into multiple *complimentary tasks*
  - Multi-Threaded programs are extremely common and simplify design of complex systems

Exam 1

Exam Details

- Exam is closed book / closed notes
- Covers Chapters 1 - 4
- You will have the full class period for the exam
  - Short Answer/Multiple Choice
  - Longer Problems
  - You are **not** required to memorize function names/parameters
    - Any code you are asked to write will be psuedo-code

Responsibilities of an OS

- CPU Allocation
- Memory and Storage management
- Regulate and Provide Access to Peripherals

Defining/Characterizing an OS

- Services provided to the user ***
- Architecture
- System Calls

System Calls

- OS exposes functionality to applications via **system calls**
  - From a programmer's perspective, they are nothing special…
  - Their implementation is **very** different however…
- *Understand dual-mode execution and interrupt handling*

Processes

- Process v.s Program
  - A Process is a *running* program, with data, stack, heap, etc.
- Process can be in 5 different states
- OS represents each process by a **P**rocess **C**ontrol **B**lock
- *Understand what fork and exec calls do…*

IPC

- Inter-process Communication:
  - Shared memory
  - Message Passing
- Understand pipes and how they can be implemented

Threads

- Threads consist of separate stack, registers, program counter
  - All threads within a process share code and heap
- Kernel v.s. User Threads
- Understand thread creation API calls