Week 2

Monday, April 8, 2019 9:31 AM

Unix

- System calls
  - Ex: ls > foo := ls contents pipes to foo
  - Ex: ls 2> foo := ls stderr contents pipes to foo
- File descriptors
  - When you open a file, you need to know everything like buffers and access info
  - Stored with array of pointers
  - We care about 0,1,2 which point to stdin, stdout, stderr
  - Redirect stdin
    - F = open(" ", …)
    - Close(0)
    - Dup(f)
    - Close(f)
  - Pipe
    - Int p[2]
    - Pipe(p)
    - p[1] := write
    - p[0] := read
  - Ls | wc
    - Parent needs to declare pipe so children can both access
    - Ls child close(1) dup(p[1]) close(p[0])
    - Wc child close(0) dup(p[0]) close(p[1])
    - Parent close(0) dup(p[0])

What is a process

- Code, data, stack
  - Usually but not always has its own address space
- Program state
  - CPU registers
  - Program counter
  - Stack pointer
- Only one process can be running in a single cpu core at any given time
  - Multi-core CPUs can support multiple processes

The process model

- Multiprogramming of four programs
- 4 independent processes
- Processes run sequentially
- Only one program active at any instant
  - Each instant can be incredibly short
  - Only applies if there is a single CPU with a single core in the system
- Context switching

When is a process end

- Either voluntary or involuntary
  - Voluntary
    - Exit or Error eixt
  - Involuntary
    - Fatal error
    - Killed by another process

Process hierarchies

- Parents create child processes
    - Childs can create their own child processes
- Unix calls these groups Process Groups
- If a Process is terminated, its children are inherited by the parent

Process States

- Process is in one of 5 states
    - Created
    - Ready
    - Running
    - Blocked/waiting
    - Exit
- Transition
    - Process enters ready queue
    - Scheduler pickes this process
    - Scheduler picks a different process
    - Process waits for an event such as IO
    - Event occurs
    - Process Exits
    - Process ended by another process

Processes in the OS

- Two layers for processes
- Lowest layer of process-structured OS handles interrupts, scheduling
- Above that layer are sequential processes
    - Processes tracked in the process table
    - Each process has a process table entry

Process table entry

- Process management
    - Registers
    - Program counter
    - CPU status word
    - Stack pointer
    - Process state
    - Priority
    - Process id
    - Parent id
    - Signals
    - Process start time
    - Total CPU usage
- File management
    - Root directory
    - Current directory
    - File descriptions
    - User id
    - Group id
- Memory management
    - Pointers to text, data, stack
    - Pointer to page table

What happens on a trap/interrupt

- Hardware saves program counter

- Hardware loads new PC, identifies interrupt
- Assembly language routine saves registers
- Assembly language sets up the stack
- Assembly language calls C to run service routine
- Service routine calls scheduler
- Scheduler selects a process to run next might be interrupted
- Assembly language routine loads PC and registers for the selected process

Threads: "processes" sharing memory
- Process :: address space
- Thread :: program counter / stream of intructions
- Two examples
    - Three processes each with one thread
    - One process with three threads

Each thread needs its own stack

Why use threads?
- Allow a single apllication to do many things at once
    - Simple programming model
    - Less waiting
- Threads are faster to create or destroy
    - No separate address space
- Overlap computation and IO
    - Could be done without threads but its harder
- EX: word processor
    - Thread to read from keyboard
    - Thread to format document
    - Thread to write to disk

Issue with threads
- May be tricky to convert single thread code to multithread code
- Re-entrant code
    - Code must function properly when multiple threads are using it at same time
    - Be careful of using static or global variables

User level threads
- + No need for kernel support
- - Maybe slower than kernel threads
- - Harder to do non-blocking IO

Kernel level threads
- + More flexible scheduling
- + Non-blocking IO
- - Not necessarily portable

Posix threads
- Standard interface for threading library