Frequently asked questions from the previous class survey

- Who needs time-sharing?
- Why do we need multiprogramming? Do we need it?
- Difference between a CPU and core?
- How do cores coordinate? Do they share the hard disk?
- How do CPUs interact with specialized processors such as GPU?
- Main memory?
- What is a buffer? Where is it used?
- Are cache sizes increasing as the number of cores increase?
- Why so many buses?
- Difference between a device driver and controller?
- Difference between CPU and the OS?
- Is there a difference between a 2.5 and 3.5 GHz CPU on a personal computer with so many bottlenecks?
- If multiple interrupts are received, how does the CPU choose which process to service?

- What is a thread? Really like to learn multithreading? Where is multithreading used?
- Single threaded vs multi-threaded ... hardware threads and hyper-threading
- How do multiple execution pipelines increase throughput? The ALU is still only processing one instruction at a time?
- What is a context-switch? How often can a process be context-switched?
- Why do transfer speeds increase on devices such as USB, Thunderbolt, etc.
- Why the Security of the USB is Fundamentally Broken By Andy Greenberg (7/31/2014)
  - http://www.wired.com/2014/07/usb-security/
  - BadUSB Malware: Alter files, redirect e-mail, etc.

Topics covered in this lecture

- Processes
- Interrupts & Context switches
- Operations on processes
- Creation

Process

- The oldest and most important abstraction that an operating system provides
- Support the ability to have (pseudo) concurrent operation
  - Even if there is only 1 CPU

Shriddeep Pallickara
Computer Science
Colorado State University
All modern computers do several things at a time
- Browsing while e-mail client is fetching data
- Printing files while burning a CD-ROM

Multiprogramming
- CPU switches from process-to-process quickly
- Runs each process for 10s-100s of milliseconds

Multiprogramming and parallelism
- At any instant of time the CPU is running only one process
- In the course of 1 second, it is working on several of them
- Gives the illusion of parallelism
  - Pseudoparallelism

A process is the unit of work in most systems
- Arose out of a need to compartmentalize and control concurrent program executions
- A process is a program in execution
- Essentially an activity of some kind
  - Has a program, input, output and a state.

A process is just an instance of an executing program
- Conceptually each process has its own virtual CPU
- In reality, the CPU switches back-and-forth from process to process
- Processes are not affected by the multiprogramming
  - Or relative speeds of different processes

An example scenario: 4 processes
- Four Program Counters
- 4 processes in memory
- A, B, C, D
Example scenario: 4 processes

- At any instant only one process executes
- Viewed over a long time, all processes have made progress

Programs and processes

- Programs are passive, processes are active
- The difference between a program and a process is subtle, but crucial

Analogy of a culinary-minded computer scientist baking cake for daughter

Scientist’s son comes in screaming about a bee sting

In our example, the scientist has switched to a higher priority process...

- FROM Baking
  - Program is the cake recipe
  - TO administering medical care
  - Program is the first-aid book
- When the bee sting is taken care of
  - Scientist goes back to where he was in the baking
Key concepts

- Process is an activity of some kind; it has a
  - Program
  - Input and Output
  - State
- Single processor may be shared among several processes
  - Scheduling algorithm decides when to stop work on one, and start work on another

How a program becomes a process

When a program is executed, the OS copies the program image into main memory

- Allocation of memory is not enough to make a program into a process
- Must have a process ID
- OS tracks IDs and process states to orchestrate system resources

Program in memory (I)

- Program image appears to occupy contiguous blocks of memory
- OS maps programs into non-contiguous blocks

Program in memory (II)

- Mapping divides the program into equal-sized pieces: pages
- OS loads pages into memory
- When processor references memory on page
  - OS looks up page in table, and loads into memory

A process in memory

- stack
- heap
- data
- text

- (Function parameters, return addresses, and local variables)
- (Memory allocated dynamically during runtime)
- (Global variables)
- (Program code)
Advantages of the mapping process

- Allows large logical address space for stack and heap
- No physical memory used unless actually needed
- OS hides the mapping process
  - Programmer views program image as logically contiguous
  - Some pages may not reside in memory

Finite State Machine

- An initial state
- A set of possible input events
- A finite number of states
- Transitions between these states
- Actions

Process state transition diagram: When a process executes it changes state

- New
- Ready
- Running
- Waiting
- I/O or event completion
- I/O or event wait
- Exit
- Terminated
- Admitted
- Scheduler dispatch

Each process is represented by a process control block (PCB)

- Process state
- Process number
- Program counter
- Registers
- Memory limits
- List of open files
- PCB is a repository for any information that varies from process to process.

An example of CPU switching between processes

- Process A
  - Operating System
  - Process B
    - Save state into PCB
    - Idle
    - Restart from PCB

Scheduling Queues

- Job Queue: Contains all processes
  - A newly created process enters here first
- Ready Queue
  - Processes residing in main memory
  - Ready and waiting to execute
  - Typically a linked list
- Device Queue
  - Processes waiting for a particular I/O device
Throughout its lifetime a process migrates among various scheduling queues

- **Long-term scheduler:** Batch systems
  - Executes much less frequently
  - Can take more time to decide what to select

- **Short-term scheduler**
  - Select process for CPU frequently
  - Selected process executes for few milliseconds
  - Typically, execute once every 10-100 milliseconds

UNIX and Windows systems often have no long-term scheduler

- Put every new process in memory for the short-term scheduler
- **System stability** depends on:
  - Physical limitations: Number of terminals
  - Self-adjusting nature of users

Somewhere in between: The medium term scheduler

- **Premise:** It can be advantageous to reduce degree of multiprogramming
  - Remove processes from memory
  - Reduce active contention for the CPU
  - Reintroduce processes later on **Swapping**
  - Swapping improves the process mix
  - Cope with strains on resources such as memory

Interrupts and Contexts

- **Interrupt** causes the OS to change CPU from its current task to run a kernel routine
- Save current context so that suspend and resume are possible
- Context is represented in the PCB
  - Value of CPU registers
  - Process state
  - Memory management information

**INTERUPTS & CONTEXTS**

January 26, 2016
CS370: Operating Systems (Spring 2016)
Dept. Of Computer Science, Colorado State University

Instructor: SHRIDEEP PALICKARA

January 26, 2016
CS370: Operating Systems (Spring 2016)
Dept. Of Computer Science, Colorado State University

L3.34
Context switch refers to switching from one process to another

1. **Save** state of current process
2. **Restore** state of a different process

Context switch time is pure **overhead**
- No useful work done while switching

Factors that impact the speed of the context switch

- Memory speed
- Number of registers to copy
- Special instructions for loading/storing registers
- Memory management: Preservation of address space

Factors that impact the speed of the context switch

- Memory speed
- Number of registers to copy
- Special instructions for loading/storing registers
- Memory management: Preservation of address space

Process Creation: A process may create new processes during its execution

- **Parent** process: The creating process
- **Child** process: New process that was created
  - May itself create processes: Process tree
- All processes have **unique** identifiers

Operations on Processes

Processes execute concurrently
- Can be created and deleted dynamically.

Example: Process tree in Solaris

Init: Root parent process for all user processes

- Get a listing of processes with `ps` command
  - `ps`: List of all processes associated with user
  - `ps -a`: List of all processes associated with terminals
  - `ps -A`: List of all active processes
Resource sharing between a process and its subprocess

- Child process may obtain resources directly from OS
- Child may be constrained to a subset of parent’s resources
  - Prevents any process from overloading system
- Parent process also passes along initialization data to the child
  - Physical and logical resources

Parent/Child processes: Execution possibilities

- Parent executes concurrently with children
- Parent waits until some or all of its children terminate

Parent/Child processes: Address space possibilities

- Child is a duplicate of the parent
  - Same program and data as parent
- Child has a new program loaded into it

Process creation in UNIX

- Process created using `fork()`
  - `fork()` copies parent’s memory image
  - Includes copy of parent’s address space
- Parent and child continue execution at instruction after `fork()`
  - Child: Return code for `fork()` is 0
  - Parent: Return code for `fork()` is the non-ZERO process-ID of new child

`fork()` results in the creation of 2 distinct processes

Parent

```
PID=abc
...
...  
... id=fork()
...  
...  
```

Child

```
PID=xyz
...
... id=fork()
...  
...  
```

Results in

- Child: `id = xyz here`
- Parent: `id = 0 here`

Child will execute from here
Simple example:

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int x;
    x=0;
    fork();
    x=1;
    ...
}
```

Both parent and child execute this after returning from fork().

Another example:

```c
#include <stdio.h>
#include <unistd.h>

int main () {
    printf("Hello World\n");
    fork();
    printf("Hello World\n");
}
```

```c
#include <stdio.h>
#include <unistd.h>

int main () {
    printf("Hello World\n");
    if (fork()==0) {
        printf("Hello World\n");
    }
}
```

Hello World
Hello World
Hello World
Hello World
Hello World
Hello World

What happens when `fork()` fails?

- No child is created
- `fork()` returns -1 and sets `errno`
- `errno` is a global variable in `errno.h`

The contents of this slide-set are based on the following references: