# Fuck Nokovic's Class

Tuesday, October 17, 2023 9:16 PM

## Introduction

### Interrupts
- Transfers control to interrupt service routine
- OS IS INTERRUPT DRIVEN

### I/O Interrupts

Synchronous:
- User process waits for I/O request and completion
- Must know I/O latency, so knows wait duration

Asynchronous:
- I/O process returns without waiting for I/O completion
- Interrupt scheduled at completion of I/O
- Interrupt handler signals user process when I/O is done
- Concurrent

### Interrupt-Handling

Polling: Sends signal to each device to see which one sent signal

Vectored: Signal sent includes identity of sender

↑ slower ↓ faster

Maskable: Can be disabled/"masked" by CPU instructions

Non-maskable: Can not be "masked" by CPU instructions

Response Time: Elapses from interrupt signal and execution of FIRST statement in corresponding handler

### Classes

Program:
- Arithmetic overflow
- Division by 0
- Reference outside users allowed memory space

Timer: Timer within program

I/O: I/O controller

Hardware: Power failure or memory parity

## Storage Structure

### Main Memory
- Random Access: DRAM, SRAM

Secondary Memory: Extension of main memory
- Hard Disk Drive (HDD)
- Non-volatile memory (NVM), SSD (solid state drive), etc.

capacity: ↑ smaller, ↓ larger
access time: ↑ faster, ↓ slower

Volatile: Does not save when no power (DRAM, SRAM). Faster
Non-volatile: Does save when no power (HDD, SSD). Slower

### I/O Techniques
- Programmed I/O
- Interrupt Driven I/O
- Direct Memory Access (DMA)

### Programmed I/O
- Performs requested action, then sets appropriate bits
- The processor periodically checks status of I/O module
- Performance SIGNIFICANTLY WORSE/ degraded

### Interrupt-Driven I/O
- Processor issues I/O command to module
- I/O module will interrupt processor service
- Executes data transfer
- More efficient, but requires active intervention of processor

### Direct Memory Access (DMA)
- Performed by separate module on system bus
- Most efficient
- Process involved only at beginning and end of transfer
- Processor is slower during transfer when access to bus is required

efficiency ↑ +

### Archetectures

Von Neumann: 1 bus used for both data transfers and instruction fetches

Harvard Architecture: Separate data and instruction busses

### Processor Systems

Single General Purpose

Multiprocessors:
- Increased throughput, economy of scale, increased reliability
- Data inconsistencies
- Load balancing
- I/O bottleneck
- Cache coherency

Types:

Symmetric: Each processor performs all tasks.

Asymmetric: Each processor performs specific task.

### OS Operations

Bootstrap: Simple code to initialize system & load kernal

## Computer System Structure
- Hardware
- OS
- Application Programs
- Users

## Processes
- Program in execution is most frequently referenced one
- Program → process when executable file loaded into memory
- 1 program can be several processes
- 2 parts; threads (concurrency) and address spaces (protection)

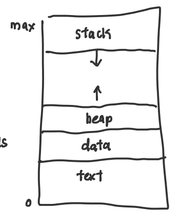
### Address Space
- Each process has one
- Provides protection
- Processes represented by PCB (Process control blocks)
  - Address space
  - Execution state

### Process in Memory

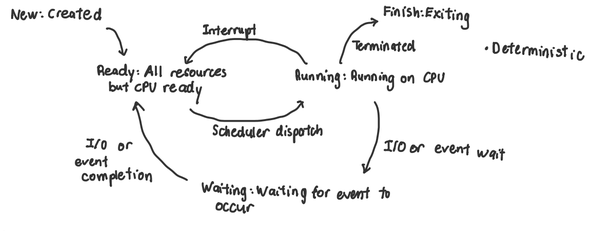
Text: Executable code
Data: Global variables
Heap: Dynamically allocated memory
Stack: Temporary data

Stack & heap grow towards each other

Heap: All the malloc, alloc calls in C (Pointers)
Stack: int ..., String...




### States




- Deterministic
- One process can run on any processor core at any instant

### PCB
- Represents process
- Scheduling information (priority)
- Accounting time (CPU time)
- Open files
- Other miscellaneous information

### Process Scheduling
- Maximize CPU use
- Quickly switch processes
  - Ready queues
  - Wait queues
  - Process migrate
- As processes enter a system, they are in ready queue until selected for execution or dispatched
- Processes waiting for certain events are placed in wait queue

Multiple scenarios:
1) Issue an I/O request and put in I/O wait queue
2) Create child process and in wait queue until child termination
3) Forcibly removed. b/c of time slice expire OR interrupt

### Dispatcher

OS must take care of scheduling (fairshare of CPU time) and protection (processes don't modify each other

- Run process for a while
- Pick process for ready queue
- Save state
- Load state
- Run

- State must be saved by dispatcher to avoid damage from next process
  - Program counter
  - Status word
  - Registers

### Context Switch
- Switch CPU core to another process requires state save of current process & state restore of different process

### Exceptions / Traps
- CPU only 1 process
- When user process is running, dispatcher is not
- OS regain CPU control?
- User process gives up CPU to OS (internal events)
  - System call
  - Error (e.g. bus, segmentation, array index, etc.)
  - Page fault
  - Yield

### Interrupts
- OS interrupts user process (EXTERNAL events)
  - Completion of input
  - Completion of output
  - Completion of disk transfer
  - Data packet → Network
  - Timer
- Electric signal signals from one component to another that event requires specific action

## Threads
- Lightweight process
- Sequential execution stream
- Process has at least 1 thread

Most apps today are multithreaded
- Displaying graphics
- Keystrokes
- Spelling & grammar

- ↑ efficiency, simplify code
- Parallel coding

### Thread Creation

0) Import pthread.h
```
#include <pthread.h>
```
1) Declare thread ID
```
pthread_t thread_id;
```
2) Create thread with pthread_create();
```
pthread_create(a,b,c,d);
```
where
- a: pointer to thread ID (&thread_id)
- b: Attributes. Usually NULL or 0 for default
- c: function
- d: function variables

3) Use pthread_join(); for threads (same as wait())
```
pthread_join(a,b)
```
where
- a: thread ID [&thread_id]
- b: NULL

### Thread Termination

1) Use pthread_cancel(tid);
- Only indicates a request to terminate thread
- Actual termination depends on target thread set up to handle request
- Can set cancellation state & type using API
- Only occurs when pthread_testcancel() invoked, then cleanup handler

## Process Creation & Termination

Creation:
- Load code from scratch
- Set up stacks
- Initialize PCB
- Make process known to dispatcher

Forking Process
- Make sure parent has saved state AND IS NOT RUNNING
- Copy of code, data, stack
- Copy of PCB of parent & child process
- Child process made known to dispatcher

- Parent makes children processes, which creates more processes
- Use pid to distinguish (process ID)

OPTIONS
- Sharing options
  - P and C share everything
  - C has subset of P
  - No sharing
- Execution options
  - Concurrently
  - P warts for C

fork(): new process
exec(): used after fork to replace process' memory with new program
wait(): P waits for C to terminate

Termination:
- Last statement with exit()
- Deallocates memory (physical & virtual)
- Close open files
- Notify parent functions (wait())
- Can be terminated with abort() or kill() when:
  - Child too many resources
  - Task assigned no longer needed
  - Parent exiting and OS does not allow child to live w/o parent

Cascading Termination: All children, grandchildren are terminated

### Communication Models
- Shared memory (shm)
- Message passing (pipes)

### Interprocess communication
- Info sharing
- Computation speed up
- Modularity
- Convinience

## Parent & Child Functions

fork(): Creates duplicate function (1 child 1 parent)

When called as follows:
```
pid_t pid;
pid = fork();
```
If pid < 0, fork failed
pid == 0, child process
pid > 0, parent process

Parent & Child execute every line after fork!

Each fork doubles # of processes

wait(NULL): Used on parent function to wait for children functions to terminate

If not used, one of 2 functions could occur:

Zombie Function:
- Process that has finished execution but still has entry in process table
- Child process done, but parent function is waiting (e.g. sleep(50): sleep for 50 seconds)
- Wastes system resources
- Child finishes first but waits while alive

Orphan:
- Process where parent no longer exists (terminated or finished) without waiting for child to finish
- Orphan process stays and can not properly terminate

### Functions to Remember

exec functions execute functions

execvp(args[0], args);
- Executes and creates new child function
- Child process does not have to run same program as parent process

args[0] & args are pointers to char *args[].
- execvp parameters are ALL pointers
- First variable is a filename

execv(args[0], args);
- Same as execvp BUT args[0] is path to executable