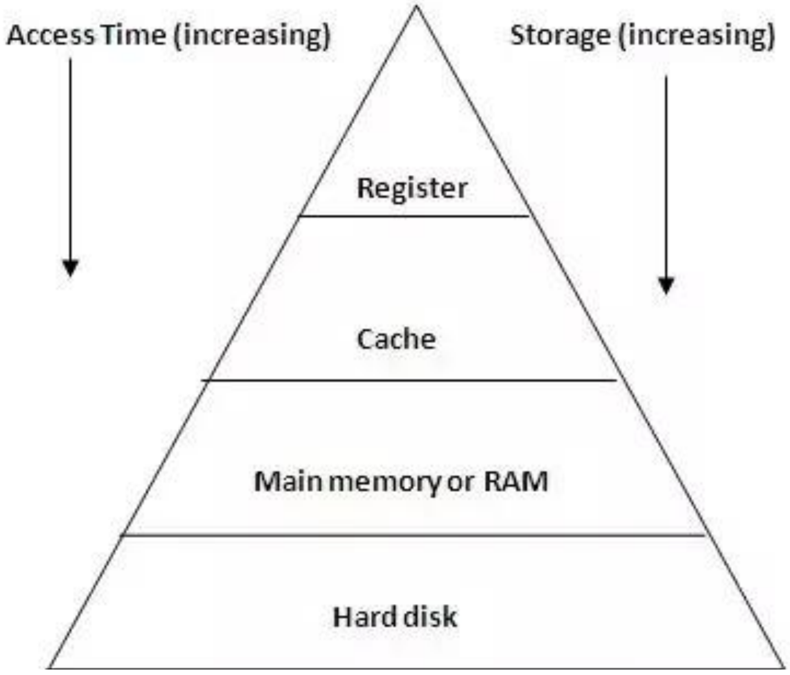
ITSC 3146 Test 1 Review

Volatile vs. Non-Volatile Memory

- **Volatile** memory is temporary memory.
- The main type of this is **RAM** or **Random Access Memory**
- 2 types of RAM:
  - SRAM (Static RAM): retains value as long as power is on
  - DRAM (Dynamic RAM): value must be refreshed every 10-100 ms

- **Non-volatile** memory is permanent memory (ie. retains data permanently)
- Most permanent memory is in the form of **ROM** (read-only memory [misleading since it can be modified]) and **Read-Write memory** (ie. disks)

Memory Hierarchy Comparison



- ➔ **Register**: very small and very fast memory in the CPU
- ➔ **Cache**: small, fast memory for frequently used data, organized in a fashion similar to main memory. Uses principle of *locality of reference*. System may have multiple levels of cache.
- ➔ **RAM/Main Memory**: volatile memory closer to CPU [compared to hard disk], collection of cells (1 bit each) organized into sets called supercells
- ➔ **Hard Disk**: very cheap and very large non-volatile storage. Random access of data is very slow since it is a mechanical device organized into tracks and sectors

Modes of I/O Operations

- CPU *stops* current activity
- CPU *asks* if device is ready, e.g.
  - Device has new input
  - Device is done with previous command/output
- If yes, CPU services device
- CPU moves to next device
- Once done, CPU *resumes* previously stopped activity
- CPU repeats above steps *periodically*

Sequence of activities...

- CPU executing instructions
- *Interrupt* received
- CPU
  - Finishes current instruction
  - Recognizes interrupt
  - Saves current state
  - Services interrupt
  - Resumes normal activity

- Most programs need to take input of some kind
- Multiple ways of actually implementing this system
  - Polling
    - Continually asking if the I/O device has anything new to send
      - Costly and inefficient usually
  - Interrupts
    - Computer sends a signal to the device letting it know it needs I/O
      - The I/O device then interrupts the program when it has completed its task

Embedded Real Time Systems

- Correct system function depends on timeliness
- Need special OS to ensure timeliness

- Hard Real Time Systems:
  - Violation of timing could be catastrophic (think of airplane example)
  - FAILURE if response time too long
  - Secondary storage is limited
- Soft Real Time Systems:
  - Time deadline is not as critical
  - LESS ACCURATE if response time is too long
  - Useful in applications such as multimedia, virtual reality

OS Overview

- **Operating System** or **OS** is the software that sits between hardware and application/user programs. It provides a virtual interface and acts as a resource manager.

- Services provided by OS:
  - Program Execution
  - Memory Management
  - File Management
  - I/O Management
  - Information Maintenance
  - Communication Services
  - User Management
  - Error Management
  - Accounting Services

OS Mechanism vs. Policy

- **Mechanisms** are the data structures and operations that are used to implement abstractions/services (ie. the “How”)
- **Policies** are the procedures/rules to guide the selection of an action from possible alternatives (ie. the “What/When/Which”)
- OS design typically separates the two

System Operation Modes: Kernel and User

System operation can be split into two modes:
User and **Kernel**.

- **User** mode:
  - Execution on behalf of user (ie. protected mode)
  - No direct access to hardware
  - Can execute only subset of instructions
  - Can access only restricted memory areas

- **Kernel** (monitor/supervisor/system) mode:
  - Execution on behalf of operating system (ie. privileged mode)
  - Complete access to hardware
  - Can execute any instruction
  - Can access any memory area
  - Invoking a system call will switch you into kernel mode (as will any interrupt)

Monolithic Architecture

- **Monolithic Architecture**: the entire OS is a single program, essentially a collection of procedures linked into single executable. Program runs fully in kernel mode
- Any procedure can call any other directly (Efficient procedure calls)
- Design, implementation, debugging etc. can be hard
- OS could become unwieldy & difficult to understand
- Error in one part of OS can bring down entire OS

Layered Architecture

- **Layered Architecture**: divides OS into multiple layers, each layer is responsible for certain operations/services.
- Layers are independent of layers above them.
- Example:

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

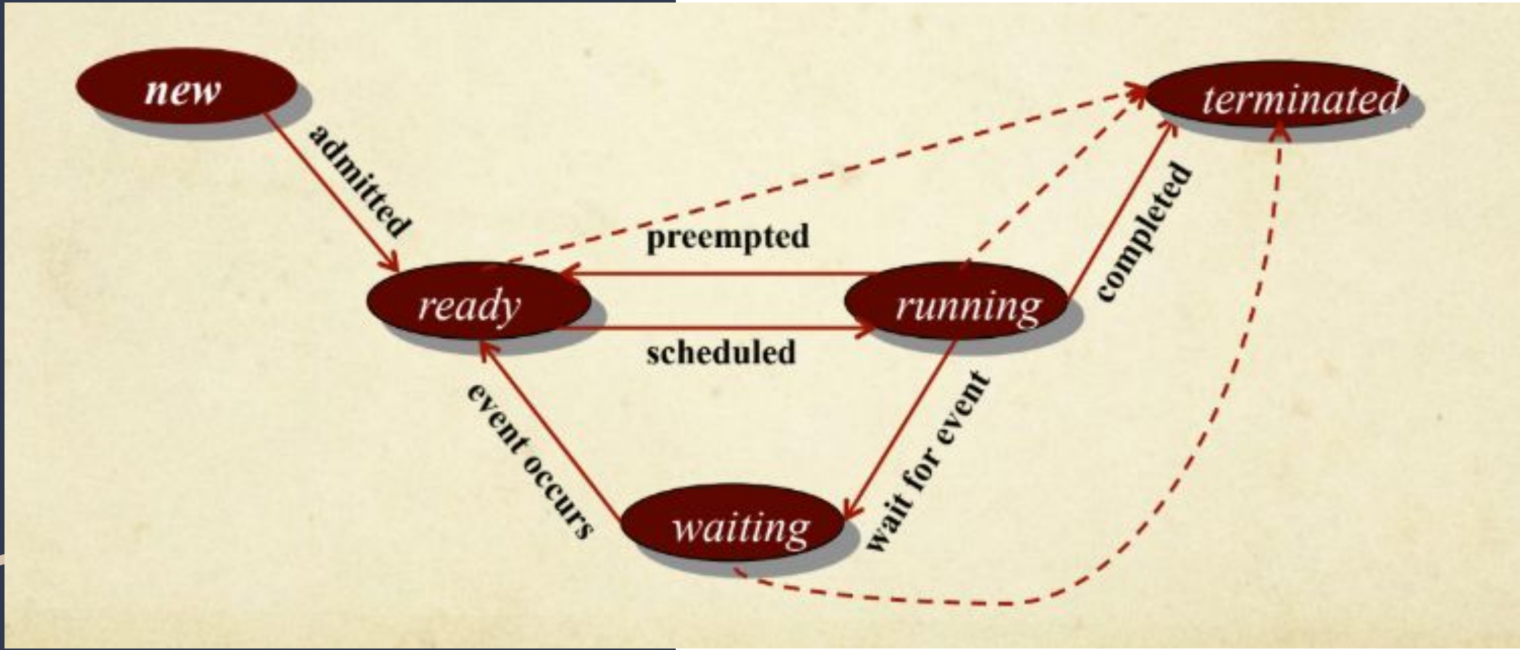
Microkernel Architecture

- **Microkernel Architecture**: splits OS functionality into multiple small modules.
- The core module, called microkernel, runs in kernel mode. All other modules run in user mode.
- Communication between modules is accomplished using message passing.
- More commonly used in embedded/real-time systems.

Microkernel Architecture – continued

- ➔ Easier to design, implement & debug
- ➔ More flexible & easier to extend
- ➔ More isolation of faults/errors
- ➔ Error in one module need not bring down entire OS
- ➔ More reliable & more secure
- ➔ Significant performance overhead

Processes


new
admitted
ready
preempted
scheduled
running
completed
terminated
wait for event
waiting
event occurs

Processes

Process control block used for this

- Process ID
- Process State (ready, running etc.)
- Program Counter – address of next instruction to be executed
- Registers – general purpose registers, stack pointer etc.
- Scheduling information
- Memory management information
- Accounting information – time limits, etc.
- ...

Process ID
Process State
Program Counter
Registers
Memory limits
....

Code Review

Code Review- Data Types

Given the following code, what is the output?

Answer: 2

```cpp
#include <iostream>
using namespace std;

int main() {

    float num1 = 1.45;
    double num2 = 2;
    int result;

    result = num1 * num2;

    cout << result <<endl;

    return 0;
}
```

Code Review- Pointers

```
//To Declare a Pointer:
int myInt = 15;
int *myPointer = &myInt;

cout << myPointer << endl; //value of myPointer
cout << &myInt << endl; //memory address of myInt

cout << *myPointer << endl << endl; //value pointed to by myPointer
cout << myInt << endl; //value stored in myInt
```

Code Review- Structs and Functions

```
struct point{
  float x;
  float y;
};

void displayArray(point myPoints[]){
    for(int i=0; i<2; i++){
        cout << myPoints[i].x << " " << myPoints[i].y << endl;
    }
}

int main()
{
    struct point point_list[2] = {
        {1.0,2.5},
        {3.2,5.4}
    };

    displayArray(point_list);


    return 0;
};
```

Code Review- Structs and Functions

Struct called point containing two fields of type float x and y

```
struct point{
  float x;
  float y;
};

void displayArray(point myPoints[]){
    for(int i=0; i<2; i++){
        cout << myPoints[i].x << " " << myPoints[i].y << endl;
    }
}

int main()
{
    struct point point_list[2] = {
        {1.0,2.5},
        {3.2,5.4}
    };

    displayArray(point_list);

    return 0;
}
```

Takes in an point array as a parameter in the displayArray function

return type

accessing index ø's x and y fields

declaring an array of Points each index contains an x and y

calling displayArray and Passing point_list as an argument

Coding Review – Forks

Assume that the code given is executed once.

1. How many times (if any) will the code at line 7 get executed?
2. Would the parent or child process execute line 8?
3. How many times would “I’m done!” be printed to the console?

```
1  #include<iostream>
2  #include<unistd.h>
3  using namespace std;
4
5  int main(){
6
7      if(!fork()){
8          fork();
9          fork();
10     }
11     else{
12         fork();
13     }
14
15     cout<<"I'm done!"<<endl;
16
17 return 0;
18 }
```

1. The code will be executed once by the initial parent process.
2. The child process would execute the code at line 8.
3. 6 times.

Code Review- Processes

pid_t wait(int *status*);

pid_t waitpid(pid_t *pid*, int *status*, int *options*);

```
pid_t id = fork();
if(id == -1)
{
    std::cout << “Error creating process\n”;
}
else if (id == 0)
{
    // child process functionality
    char* args[] = {"echo", "hello", NULL};
    execvp(args[0], args);
}
else
{
    std::cout << “I just became a parent!\n”;
}
```