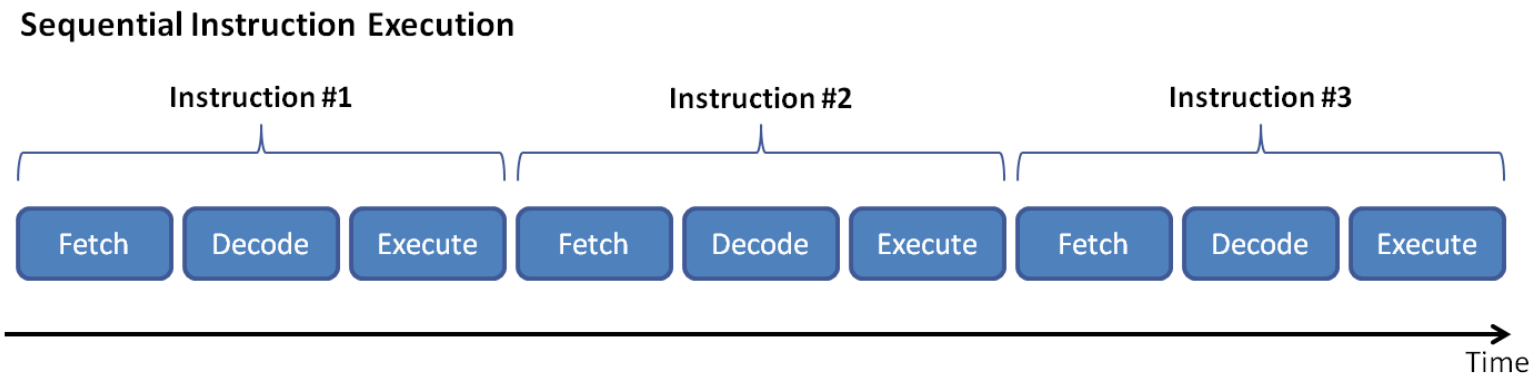
CPSC 457 Tutorial
Week 2

Sequential Instruction Execution

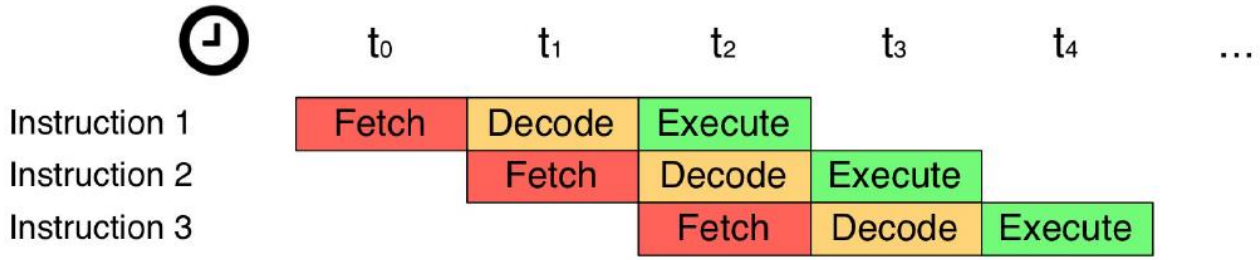


Sequential instruction execution

Taken from http://microchipdeveloper.com/32bit:mz-arch-pipeline

- Fetch - Decode - Execute
  - Read the instruction
  - Figure it out
  - Execute it
- The next instruction is not fetched until the current one is executed

Pipelining



Three stage instruction pipeline

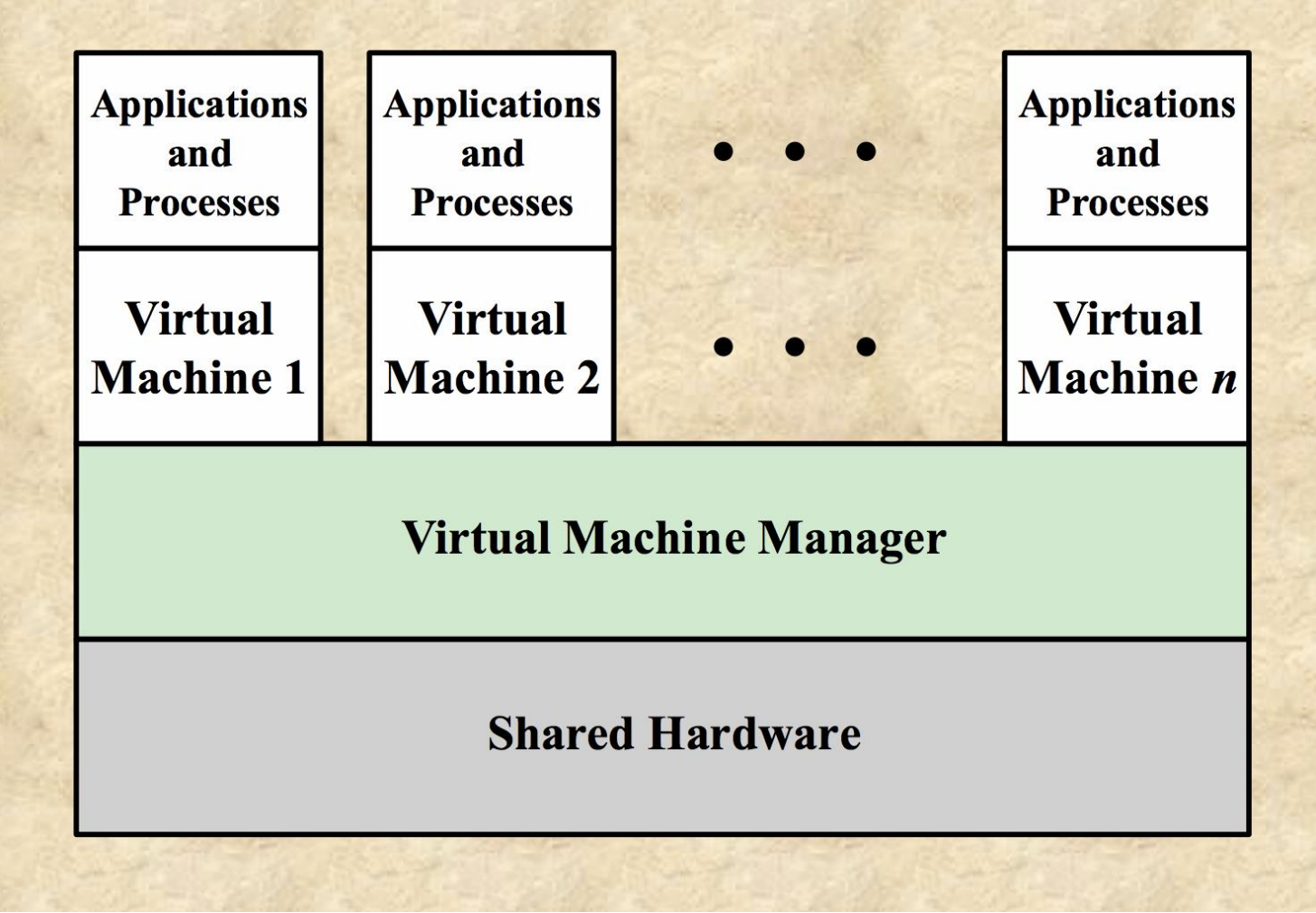
Taken from http://note_embedded2016.hackpad.com

- In pipelined CPU, each step is implemented as an independent stage
- While the first instruction is executed, the second one can be decoded and the third one can be fetched
- The idea is to keep units in CPU busy as much as possible
- The stages work in parallel and execution time of the sequence of instructions is reduced

Virtual machines

- Enables a single computer to simultaneously run multiple operating systems
- Provides complete protection of the resources since each VM is isolated from all the others
- They are portable
- No direct sharing of resources between VMs
- Difficult to implement

Virtual machines



Virtual machine concept

Taken from https://www.unf.edu/public/cop4610/ree/Notes/PPT/PPT8E/CH14-OS8e.pdf

wc and **time** utilities

- wc prints newline, word and byte counts for each file
  - man wc
- wc –l romeo-and-juliet.txt
- Compile countLines.cpp and run it as
  - ./countLines romeo-and-juliet.txt
- time is used to determine the duration of execution of a particular program
  - which time --> /usr/bin/time
- Detail information for the usage can be found in its manual page
  - man time
- Example usage
  - time ls

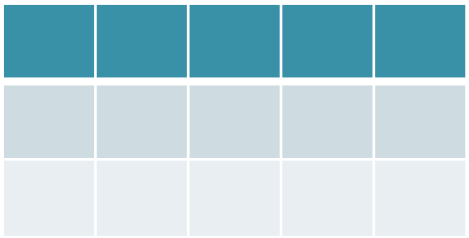
strace utility

- It is used to trace system calls and signals
  - man strace
  - strace –h for help
- Especially used for diagnostic, instructional and debugging operations
- In order to display only specific system calls:
  - strace –e open ls
- In order to get statistics, use –c flag
  - strace –c ./a.out romeo-and-juliet.txt
- ssize_t read(int *fildes*, void **buf*, size_t *nbyte*);

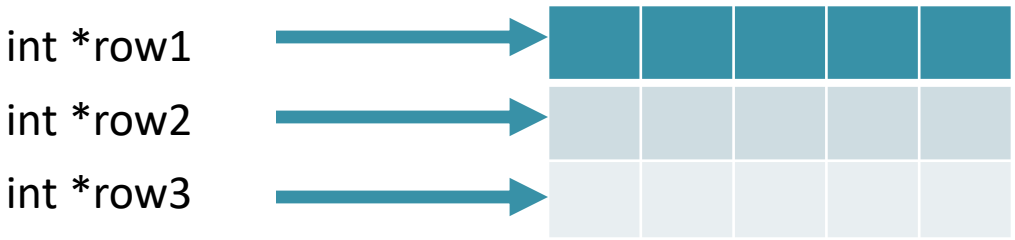
Pointer to pointers

```
int *ptr = (int*) malloc(5 * sizeof(int))
```

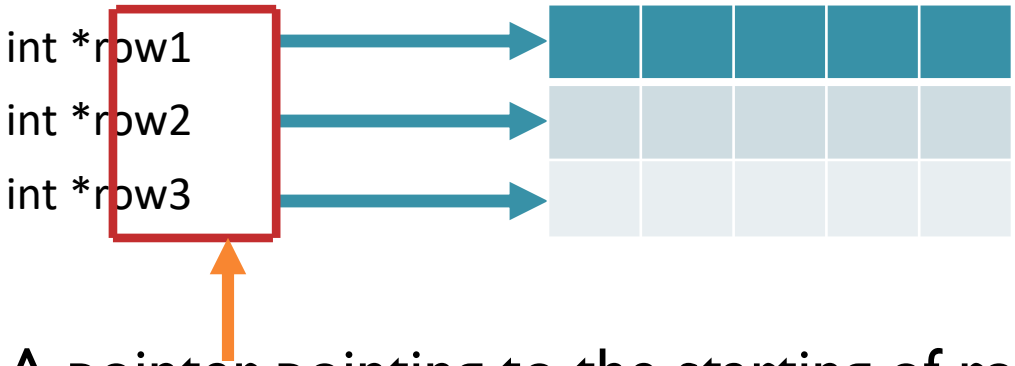
What about dynamic 2D arrays?



Pointer to pointers (cont'd)



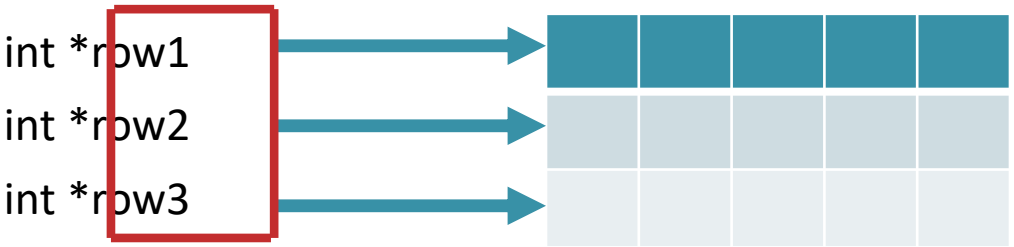
The number of rows is not known in compile time, too. We need to allocate memory for it dynamically.



A pointer pointing to the starting of row list

Pointer to pointers (cont'd)

row1 points to a block of memory containing int. That is why its type is int*



A pointer pointing to the beginning of a memory block containing int* will have the type int**

First allocate memory for the pointer of rows, then allocate memory for every row

Call by reference

- Call by reference method copies the reference to the original object while passing argument to a function
  - int solve(vector<string> &names)
- The object used in the function is the original object
- It is a way of avoiding making copies of objects during function calls
- To make sure that call by reference object is not modified in the function, use const keyword
  - int solve(const vector<string> &names)

Templates

- Templates allow us to write routines that work for multiple types without having to know what these types will be
- Two types of templates:
  - Function templates
  - Class templates

Function templates

- A function template is not an actual function, but a template for a function
- The compiler creates the actual function based on the way that the function is called

```
template<class T>
void swap(T &lhs, T &rhs) {
        T temp = lhs;
        lhs = rhs;
        rhs = temp;
}
```

Class templates

- Class templates are used to define generic classes

```
template<class T>
class MyStack{
        private:
                list<T> data;
        public:
                void pop();
                void push(const T& value);
}
```

Class templates(cont'd)

- Each member function must be declared as a template
- All member functions must be implemented in the header file

```
template<class T>
void MyStack<T> :: push(const T& value){
        //method body
}

template<class T>
void MyStack<T> :: pop(){…
}
```

Error handling

- In C, errors are reported by returning error codes from functions
- In C++ : **try, throw, catch**
- The function that encounters an error throws an exception
- Exceptions must be caught, otherwise the program will abnormally terminate