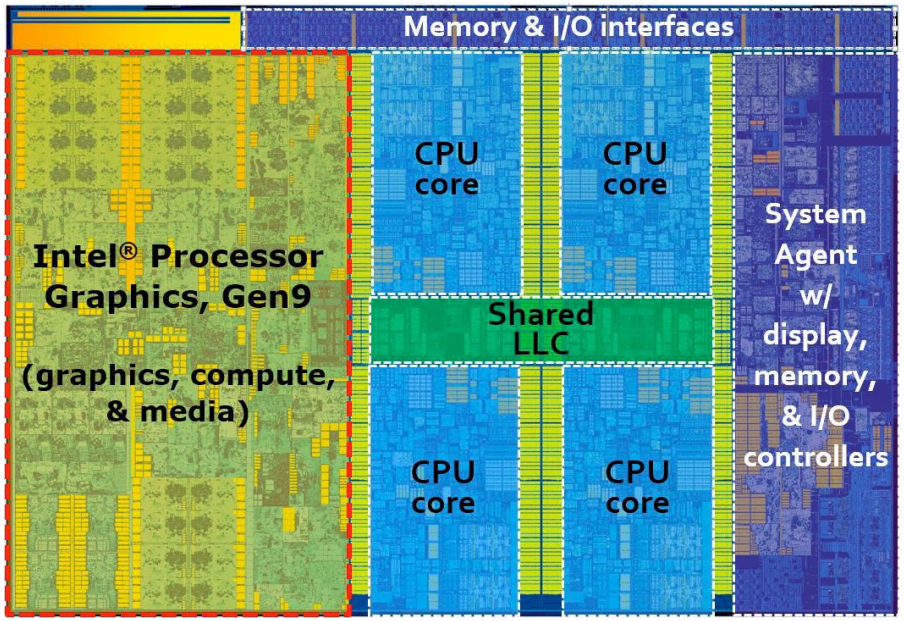
CMSC 691
High Performance Distributed Systems

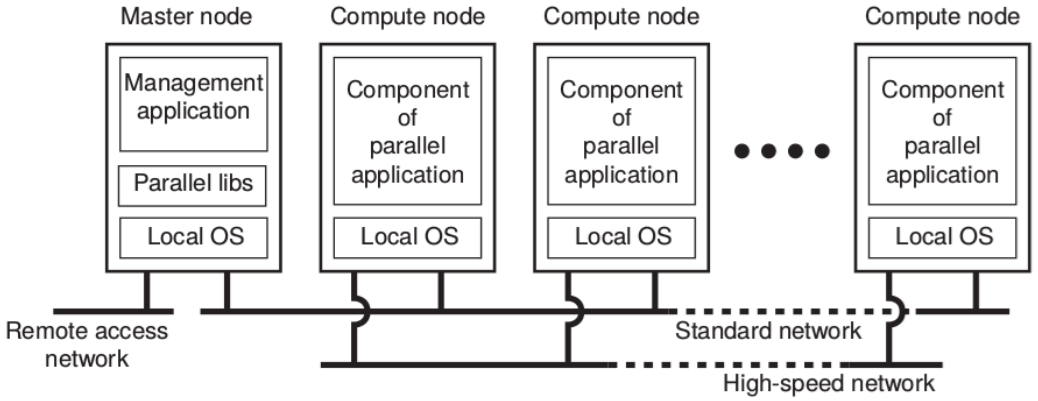
Threading and processes

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

Physical view:

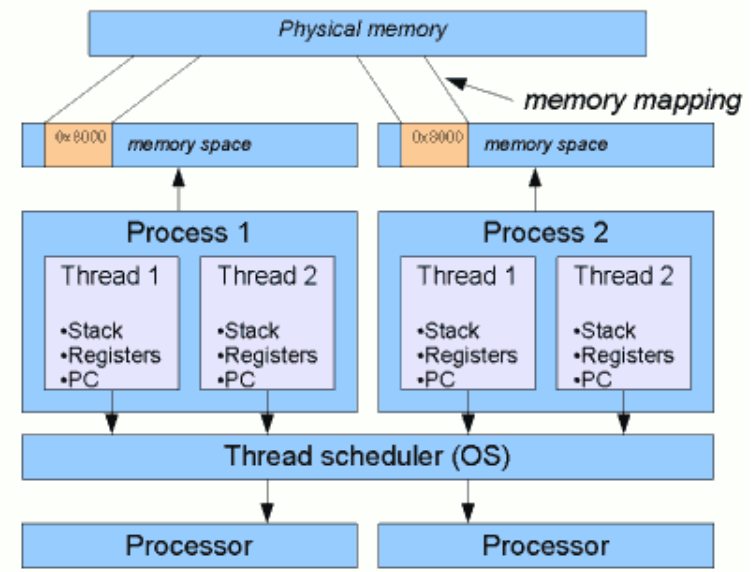
- Single-core CPU
- Multi-core CPU
- Hyper-threading
- Multi-processor system
- Cluster





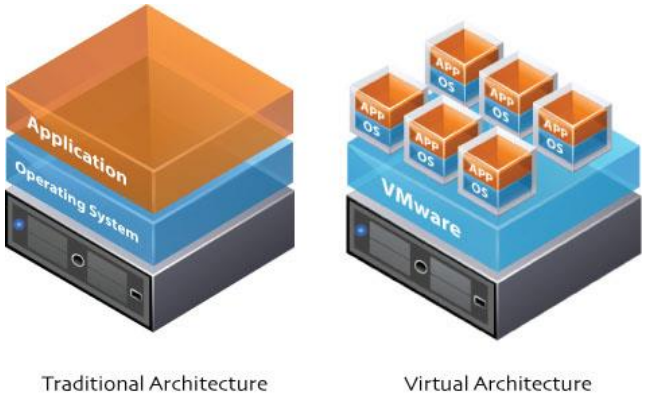
Logical view:

- Process: instance of a computer program containing its own address space / virtual memory, stack / registers, file handles, etc.
- Thread: multiple *lightweight processes* contained within a process sharing the address space (process-level global memory)
- Linux kernel schedules threads
- Thread vs process context switching
- Creating threads is much faster



Virtualization:

- Virtualization hides the physical characteristics of a computing platform from the users, presenting instead an abstraction
- VM Monitor: separate software layer mimics the instruction set of the hardware (VMware, VirtualBox). A large physical server may host multiple guest virtual machines.



Traditional Architecture Virtual Architecture

- Process VM: program is compiled to intermediate code executed by an isolated runtime system (Java VM)

Processes (fork):

- fork produces a second copy of the calling process, which starts execution after the call and copies all the process context.
- The only difference between the copies is the return value: the parent gets the pid of the child, while the child gets 0.

```
pid = fork();
if ( pid < 0) {
   fork_error_function();
} else if ( pid == 0 ) {
   child_function();
} else {
   parent_function();
}

pid_t getpid(void);
```

Creating threads:

- API available by #include <pthread.h>
- Compile with flags: gcc -pthread program.c -o program

```
int pthread_create ( pthread_t*  thread,
                     const pthread_attr_t*  attr,
                     void*  (* start_routine) (void*),
                     void* arg);
```

thread: creates a handle to a thread at pointer location
attr: thread attributes (NULL by default)
start_routine: function to start execution
arg: value to pass to start routine
returns 0 on success, error number otherwise

Waiting for threads:

```
int pthread_join(pthread_t thread,
                 void** retval);
…
void pthread_exit(void *retval);
```

thread: wait for this thread to terminate.
retval: stores exit status of thread (set by pthread_exit) to the location pointed by *retval. NULL is ignored.

returns 0 on success, error number otherwise.

Only call this one time per thread! Multiple calls on the same thread leads to undefined behavior.

If the main process finishes without waiting for joins, threads will be killed!

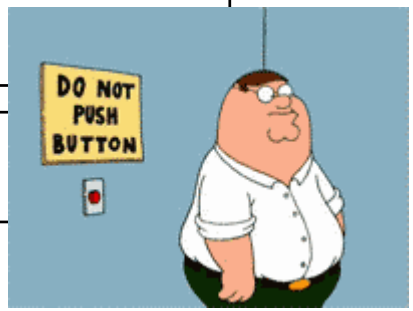
Threads example:

```
#include <stdio.h>
#include <pthread.h>

void* run(void*) {
  ...
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &run, NULL);
  pthread_join(thread, NULL);
}
```

DO NOT

```
int i;
for (i = 0; i < 10; i++)
  pthread_create(&thread[i], NULL, &run, (void*)&i);
```

```
int result;
pthread_exit(&result);
```



Thread safety:

- Code is thread-safe if it manipulates shared data structures only in a manner that guarantees safe execution by multiple threads
- Race conditions and reentrancy

```
int* x;

void* run1(void* arg) {
   *x += 1;
}

void* run2(void* arg) {
   *x += 2;
}
```

```
run1                run2
D.1 = *x;           D.1 = *x;
D.2 = D.1 + 1;      D.2 = D.1 + 2
*x = D.2;           *x = D.2;
```

- Memory reads and writes are key in data races

Detecting data races automatically:

- Dynamic and static tools can help find data races in your program
- *helgrind* analyzes your program (and causes a large slowdown)
- Run with *valgrind --tool=helgrind <prog>*
- It will warn you of possible data races along with locations
- For useful debugging information, compile with debugging information (-g)

Mutual exclusion: locks

- Mutexes are the most basic type of synchronization
- Only one thread at a time can execute code protected by a mutex
- All other threads must wait until the mutex is free before they can execute protected code

```
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);

// code
pthread_mutex_lock(&m);
// protected code, critical section
pthread_mutex_unlock(&m);
// more code

pthread_mutex_destroy(&m);
```

```
mutex m;   // C++11


// code
m.lock();
// protected code, critical section
m.unlock();
// more code
```

Mutual exclusion: semaphores

- Binary: equal to a mutex if the initial value is 1
- Counting semaphores: any number of threads at the same time
- A pair of semaphores can be useful to synchronize the main process and its threads

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_destroy(sem_t *sem);
```

Mutual exclusion: atomic operations

- An atomic operation is indivisible
- Objects of atomic types are the only objects that are free from data races, that is, they may be modified by two threads concurrently or modified by one and read by another.

```
#include <stdatomic.h>    //C11

atomic_int counter;
```

```
#include <atomic>         //C++11

atomic<int> counter;
```

CMSC 691
High Performance Distributed Systems

Threading and processes

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu