Parallel Programming Languages

COMP360

“The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it.”

Steve Jobs

Processes

- All multiprogramming OSs are built around the concept of processes
- A process is something akin to a program
- User programs are processes
- Some programs have multiple processes and the OS has several processes
- A process has:
  - memory
  - ability to use the CPU (a thread)
  - resources allocated to it

Multiple Processes

- There are usually many processes existing in the system
- If there is only one CPU, then only one thread can be running at any one time
- Other threads might be waiting to run
- Some (most) processes will be blocked waiting for something to happen (i.e. I/O)

Threads

- A **process** is a unit of resource ownership
- A **thread** is a unit of dispatching
- A thread is another flow of control through the same program
- Threads have an entry in the OS scheduling queue
- Each thread has its own stack

Creating Threads

- Each thread has its own set of registers and its own stack
- Threads share the program memory and resources (e.g. files) of their process.
- It is easier for the OS to create a thread than a process because it does not have to copy the memory image

If a thread opens a file, other threads in the program can

A. only read the file
B. access the file just as the opening thread
C. not access the file
D. access if they are children threads

Sharing the System

- Most programs perform a lot of I/O and therefore spend a lot of time waiting for I/O to complete
- When a program is waiting for I/O, another program can be running
- Some programs wait for a very long time for input (such as input from a keyboard, mouse or network)

OS Dispatcher

- A part of the Operating System that gives the processor from one thread to another
- Selects a thread from the queue to execute after interrupt or thread termination
- Prevents a single thread from monopolizing the processor time

Creating a new Process

- In C or C++ under Unix, you can create a new process with the fork function

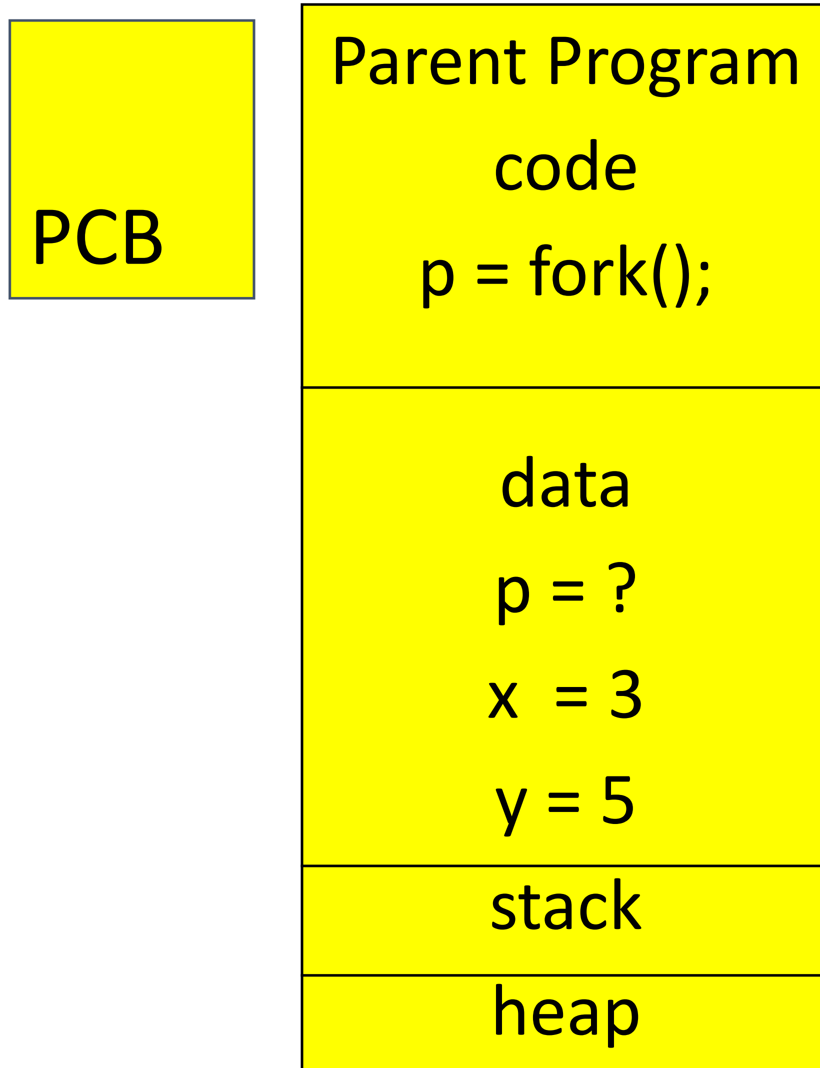
```
int fork();
```

- When fork is called, the RAM of the program is copied to another location is RAM
- A new thread is created to run in the new program address space

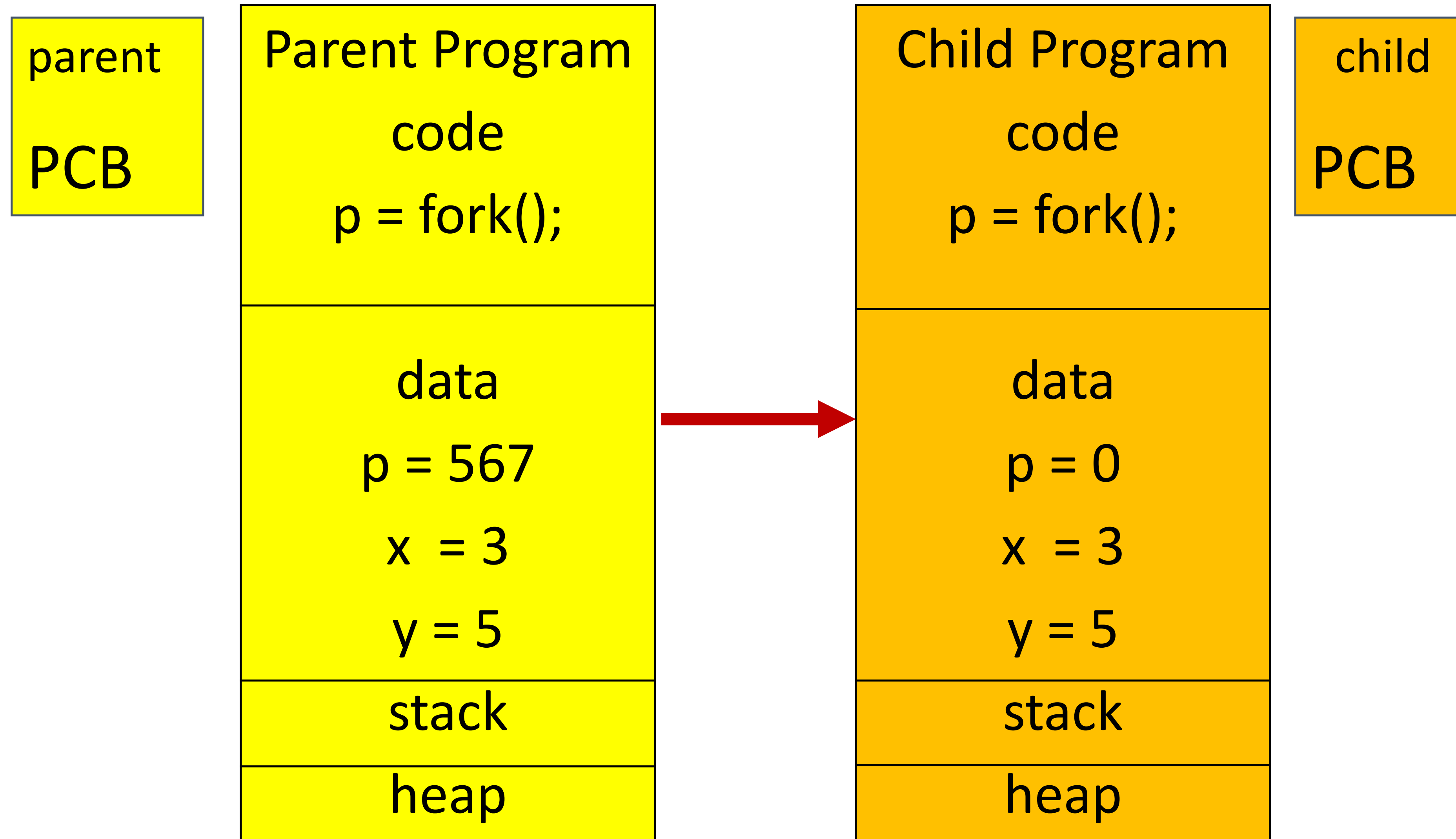
fork() Return Values

Return	Meaning
-1	Error
0	This is the child process just created
number	This is the parent process that executed fork(). The number is the PID of the child

Fork Action



Fork Action



What will be displayed?

```
int x = 3;
int p = fork();
if ( p == 0)
        x = 7;
print x;
```

A. 3 3
B. 7 7
C. 3 7
D. None of the above

Starting Another Program

- Application programs are processes in the Windows world.
- You can start a new process with the **`_spawn`** functions.
- The process calling **`_spawn`** can wait for the child to complete, continue running or be overlaid.

Versions of Spawn

- There are many versions of spawn
- _spawnl _spawnv, _spawnle, _spawnve, _spawnlp, _spawnvp, _spawnlpe, _spawnvpe

suffix	description
e	pass environmental variable
p	use PATH environmental variable to find file to execute
l	command line arguments passed as separate variables
v	argv array contains command line arguments

Comparison of Threads and Processes

- Cheaper to create a new thread
- Easier to switch between threads in the same process than between different processes
- Threads can easily share data
- Processes are protected from one another

Threads and Processes

- A thread is a flow of execution in a process
- Each thread belongs to one process
- A process may have one or many threads

Uneven Execution Speed

- Threads are frequently interrupted. This means their execution "speed" moves forward in an uneven manner
- At any instance, a thread can temporarily stop executing while the CPU does something else

Shared Memory Execution Assumptions

- Values to be manipulated are loaded into registers, changes, then stored back to memory
- Each thread has its own set of registers
- Shared values exist once in RAM
- Any intermediate results of a complex expression are stored in private memory (i.e. stack)

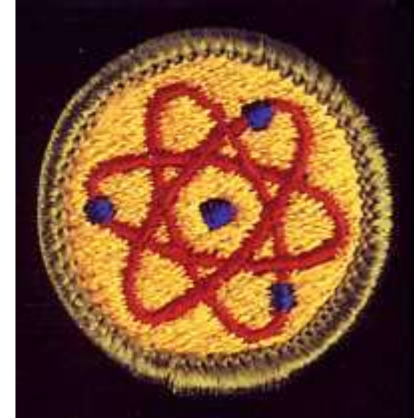
Nondeterministic Execution

- When multiple threads access shared data, the results may be nondeterministic.
- Consider two threads executing in parallel

```
Thread 1            Thread 2
x = 13;             x = 47;
```

- **`Depending upon which thread the OS executes first, the final value of x may be 13 or 47.`**

Atomic Actions



- Reads and writes to basic data types are atomic
- An atomic action is indivisible
- You can see the state of an object before or after an atomic action, but you cannot see any intermediate states
- If you atomically change a byte in memory, you will never catch it half way with only some of the bits changed

Sequentially Equivalent

- We consider the results of the parallel execution of two threads to be ***Sequentially Equivalent*** if the results are equal to
  - thread 1 running to completion and then thread 2
  - thread 2 running to completion and then thread 1
- Results not equal to running the threads sequentially are not Sequentially Equivalent
- We often consider the results of parallel execution to be *"correct"* if it is sequentially equivalent

Erroneous Results

- Sometimes when multiple threads access shared data, the results may not be sequentially equivalent
- Each thread has its own set of registers values. When the OS switches between threads, it saves and restores the registers
- Values in the register may not match RAM is another thread changed a variable

X++ in Machine Language

Thread 1	Thread 2
LOAD R1,X	
ADD R1,1	

OS switches to another thread

Thread 1	Thread 2
	LOAD R4,X
	ADD R4,1
	STORE R4,X

OS switches to another thread

Thread 1	Thread 2
STORE R1,X	

What is displayed when `run()` is executed in parallel by two threads?

```
static int count = 0;    // shared counter

public void run() {      // parallel method
    for (int i = 0; i < 1000000; i++) {
        count = count + 1;
    }
    System.out.println(count);
}
```

A. 1000000 1000000
B. 1000000 2000000
C. 2000000 2000000
D. None of the above

Mutual Exclusion

- To avoid problems when sharing data between threads, each thread has to have exclusive access to the data
- A segment of code that can only be executed by one thread at a time is called a ***critical section***
- If a second thread attempts to execute the critical section while another thread is already executing there, the second thread will be suspended until the first thread exits the critical section

Mutual Exclusion Format

- The general format for creating a mutually exclusive critical section is:

…

```
Critical Section Entry code
// Critical Section
Critical Section Entry code
```

…

- Some languages do this automatically

Mutual Exclusion Techniques

- Busy Waiting
- Disabling interrupts
- Test and Set instructions
- Semaphores
- Synchronized methods or blocks

Low Level Synchronization

- Busy Waiting wastes CPU time and keeps the CPU from running the thread that will resolve the conflict
- Disabling interrupts can only be done by the OS on single CPU systems
- Most machines have hardware instructions for synchronization
- The Intel **`XCHG EAX, dog`** atomically swaps that value in the EAX register with the value in memory location dog locking memory until complete

If you write a parallel program using busy waiting, Dr. Williams will

A. Snarl
B. Give you a bad grade
C. Patiently explain how to do it better
D. Some of the above

Semaphores

- Semaphores were originally described by E. Dykstra
- A semaphore has an integer counter and a queue of threads suspended on the semaphore
- Semaphores can only be modified by two functions **P** *(or wait)* and **V** *(or signal)*
- Supported by POSIX Threads (pthreads) and Java

P or Wait Function

```
void P( Semaphore s ) {
  while (s.counter == 0) {/* nothing */}
  s.counter--;
}
```

- If the semaphore counter is zero, the thread waits until it is nonzero
- The counter is decremented

V or Signal Function

```
void V( Semaphore s ) {
  s.counter++;
}
```

- Increments semaphore counter
- If another thread was waiting in the P function, it will be allowed to continue

OS Semaphore Implementation

- The previous definitions of the P and V functions used busy waiting
- A much better implementation asks the OS to suspend the thread

Better P and V

```
void P( Semaphore s) {
  if (s.counter == 0) { put thread on s.queue }
  s.counter--;
}
void V( Semaphore s) {
  s.counter++;
  if (s.queue != NULL) { activate one thread }
}
```

Semaphore Critical Sections

- You can create a critical section where only one thread can execute

```
Semaphore  mutex = 1;

P( mutex );
    // Critical Section
V( mutex );
```

Producer Consumer Problem

- Assume there are many producer threads and consumer threads
- Producer threads make stuff and give them to any available consumer thread
- Consumer threads must wait if no stuff is available

- This is the basis for most concurrent programming solutions

Producer Consumer with Semaphores

```
Semaphore mutex = 1, qsize = 0;

void producer (Thing x) {
  p(mutex);
  put thing on queue
  v(mutex);
  v(qsize);
}

Thing consumer () {
  p(qsize);
  p(mutex);
  get thing from queue
  v(mutex);
  return thing;
}
```

Will this work?

```
void producer (Thing x) {
   p(mutex);
   put thing on queue
   v(qsize);
   v(mutex);
}
```

A. True

B. False

Will this work?

```
Thing consumer () {
   p(mutex);
   p(qsize);
   get thing from queue
   v(mutex);
   return thing;
}
```

A. True
B. False

Parallel Programming Environments

- Shared memory
  - All threads or processes can access all the memory
  - Any thread can run on any CPU
  - Threads can communicate by changing variables
- Separate memory
  - Processes run on separate CPUs with separate memory
  - Threads communication by sending messages

Some Parallel Programming Languages

- Java
- Ada
- APIs providing parallel programming
  - pThreads – C++ functions
  - Message Passing Interface – runs on both shared and separate memory systems
  - OpenMP – Extension to C++ or Fortran
  - CUDA – Allows C++ programs to access GPU
  - Hadoop – Big data Map Reduce

Have you written a Java program with multiple threads?

A. Yes
B. No

Threads in Java

- Threads are built into Java as part of the standard class library.
- There are two ways to create threads in Java:
  - Extend the class **`Thread`**
  - Implement the interface **`Runnable`**

run Method

- Both means of creating threads in Java require the programmer to implement the method:

```
public void run() { … }
```

- When a new thread is created, the **run** method is executed in parallel with the calling thread.

Extending Java Thread Class

```
public class Example extends Thread {
  int classData;    // example data
  // optional constructor
  public Example(int something) {
      classData= something;
  }

  public void run() {
  // runs in parallel
  . . .
  }
}
```

Starting a Thread Object

- Parallel execution of the run method of a Thread object is initiated by:

```
// create Thread Object
Example xyz = new Example(143);
// start execution of run method
xyz.start();
```

Runnable Interface

- Java does not support multiple inheritance. If a class extends Thread, it cannot extend another class.
- Programmers frequently want to use multiple threads and extend another class, such as Applet.
- The Runnable interface allows a program use multiple threads and inheritance.

Implementing Runnable Interface

```
public class RExample implements Runnable {
  int classData;    // example data
  // optional constructor
   public RExample(int something) {
      classData= something;
  }

  public void run() {
  // runs in parallel
  . . .
  }
}
```

Starting a Runnable Object

- Parallel execution of the run method of a Runnable object is initiated by:

```
// create Thread Object
RExample xyz = new RExample(143);
// start execution of run method
new Thread(xyz).start();
```

Java Thread Termination

- Similar to pthreads, Java threads terminate when the run method returns
- The System.exit(int) method will terminate the entire process

Only One Run

- An annoying feature about Java threads is that they always start a method named run
- Other thread systems allow you to specify the method to be executed
- The run method has no parameters. Data must be passed to the method through common variables
- A thread can only be started once

Java Thread Errors

```
public class MultiRun extends Thread {
    int counter = 0;
    public void run() {
        System.out.print(counter);
        counter++;
    }
    public static void main(String[] args) {
        MultiRun yarn = new MultiRun();
        yarn.start();
        yarn.start(); // This causes an error
    }
}
```

Fixing the Java Thread Errors

```
public class MultiRun extends Thread {
    static int counter = 0;
    public void run() {
        System.out.print(counter);
        counter++;
    }
    public static void main(String[] args) {
        MultiRun yarn = new MultiRun();
        yarn.start();
        MultiRun yarn2 = new MultiRun();
        yarn2.start();
    }
}
```