THE UNIVERSITY OF WARWICK

High Performance Computing *Course Notes*

Shared Memory Parallel Programming - II

Dr Ligang He

Techniques

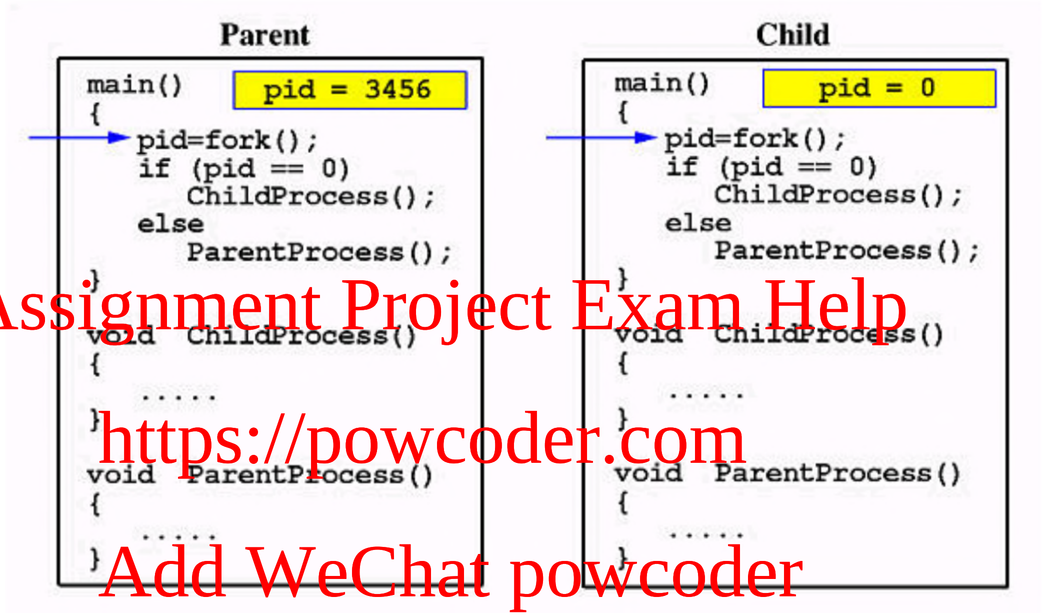
- Thread creation and running in OpenMP is implemented by calling the multi-threading APIs in POSIX operating systems
  - For example, during the compilation, if OpenMP compiler realizes that the threads need to be generated, it inserts the instructions of invoking the relevant API provided by OS
- In this part, we are going to learn the multithreading support in POSIX OS, including
  - Multiprocessing
  - Multithreading: user- and kernel-space multithreading

Multiprocessing

What is a process?

- A process is a running instance of a program
- A process records the running state of the program

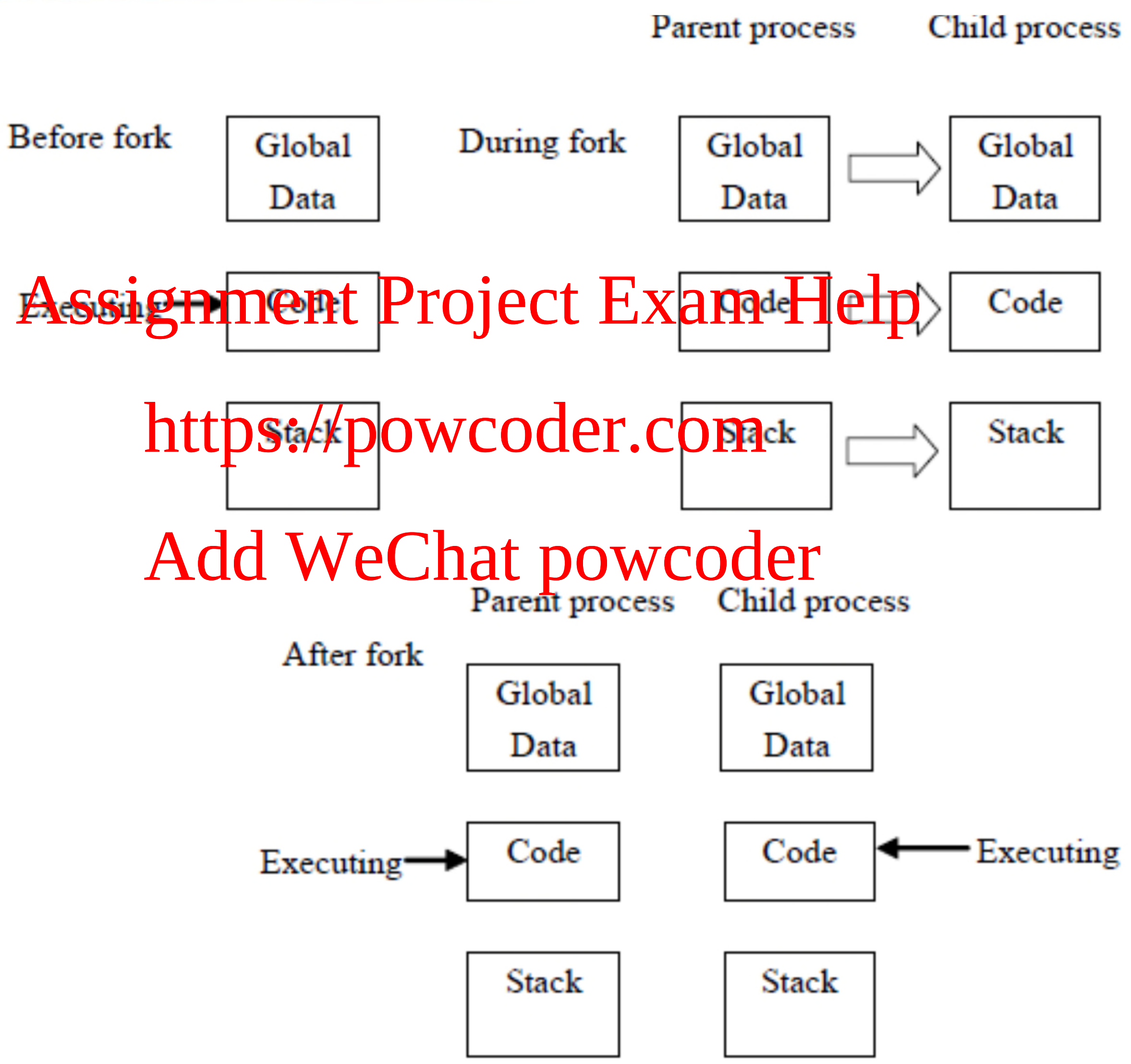
Generate a new process: the Fork function



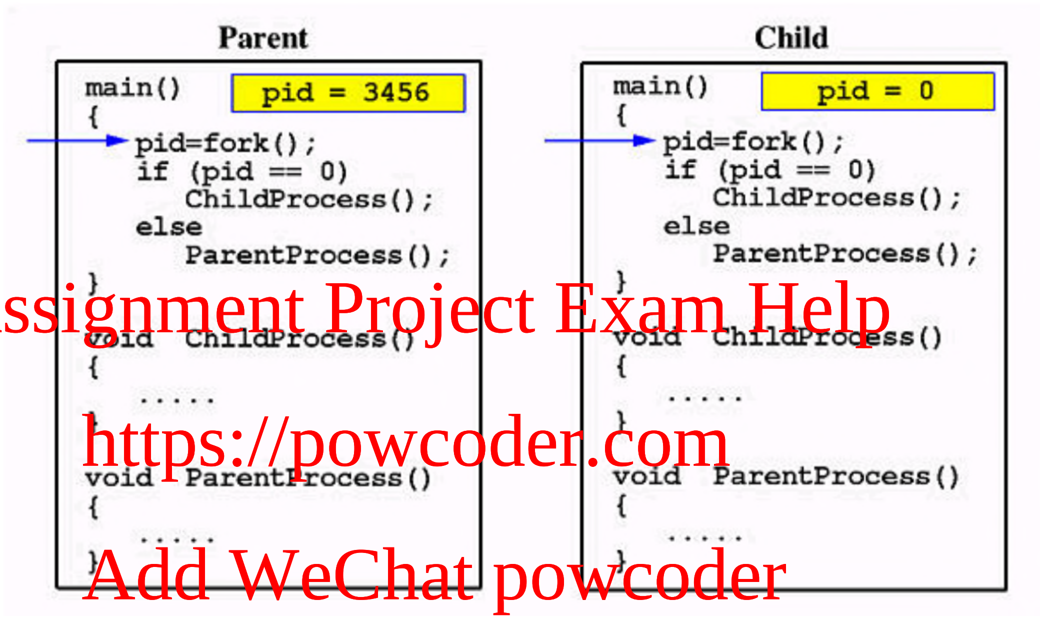
```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}
void  ChildProcess()
{
    .....
}
void  ParentProcess()
{
    .....
}
```

- fork() is used to create a child process

How a new process is created



Generate a new process: the Fork function



```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}
void  ChildProcess()
{
    .....
}
void  ParentProcess()
{
    .....
}
```

- fork() is used to create a child process
    - input and output of fork()
- The child process is exactly the same as the parent except the returned value of fork()
- Use parent and child to do different tasks

Load a new program after fork

```
main () {
   int pid, ret;
   pid=fork();        /*generate a child process*/
  if(pid==0) {        /*run by the new process*/
     ret=execl("/bin/ls", "/");       /*load a new program*/
  }
  else { /*run by the parent process*/
     perform whatever operation
     ret=wait(&status);   /* wait the child process exit */
  }
}
```

Scheduling Processes

- When to switch the processes (timing for scheduling)
  - time slice runs out
  - System call
  - trap

- Overhead of switching processes is relatively high, have to save and load the following information, for example,

  - Three segments
  - Open File descriptors
  - Signal handler table
  - program counter

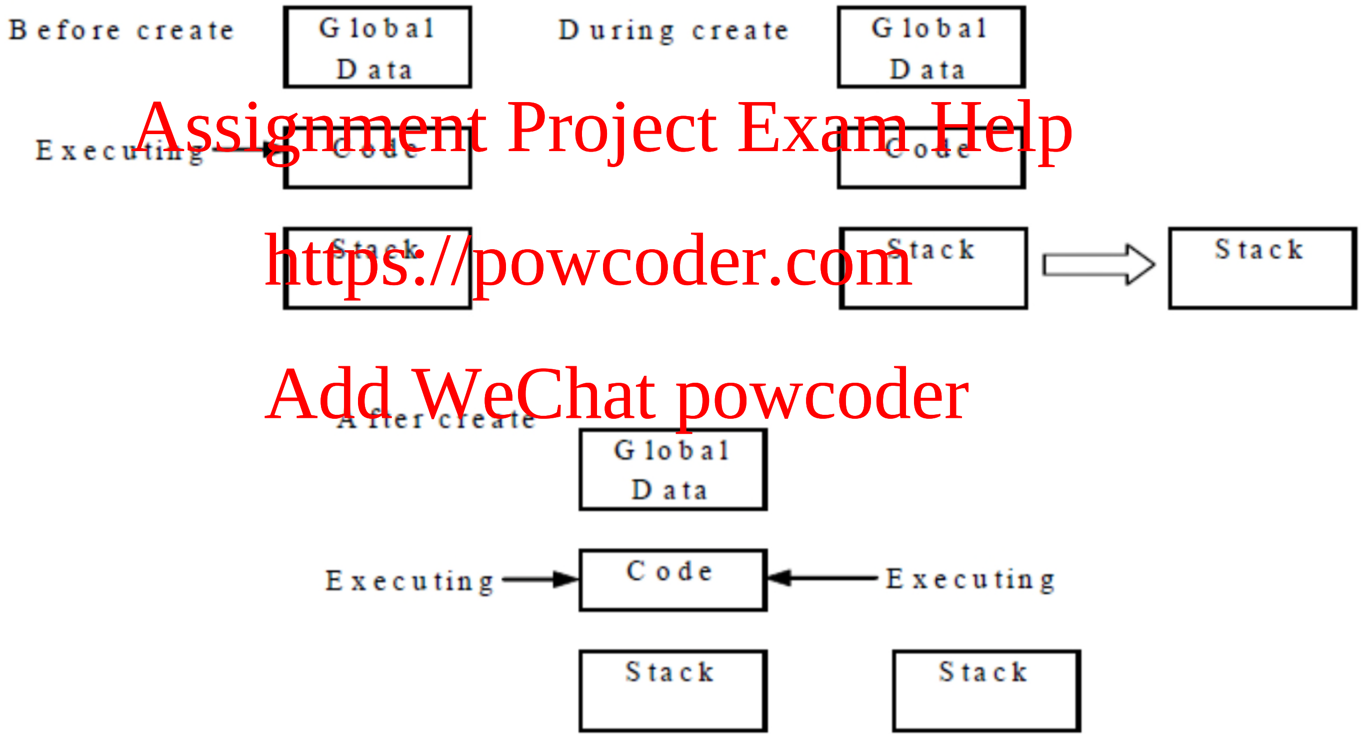
Each entry in the process table (in kernel space) contains the following:

- Process ID
- Parent process ID
- Real and effective user and group IDs
- State
- Pending signals
- Code segment
- Data segment (static data)
- Stack segment (temporary data)
- User area
- Signal handler table
- Open file descriptors
- Recent CPU usage
- Hardware register contents (unless running)
- Page table

User Space Thread

Create a new thread

- Thread – short for thread of execution/control



Two key functions in C: pthread_create and pthread_join

```
void * Calc(void *)
main() {
            int ret, param, tid;
            /*create a new thread*/
            ret=pthread_create(&tid, NULL, Calc, (void *) param);
            continue to do other tasks
            /*wait for the new thread to return*/
            ret=pthread_join(tid, NULL);
}

void *Calc(void *param) {
            int a, b; b=a+(int) param;
}
```

- **create** - create new thread of execution that runs specified procedure with specified arguments

- **join -** used to wait for the return from a specified thread

Scheduling User Space Threads

- User threads exist within a process; they are managed by the process
- Unlike process switching, there is no time slice for each thread; a thread needs to call the thread switching explicitly
- When a thread calls an explicit switch, another thread gets the opportunity to run
- A thread can hog the CPU so as to starve other threads
- User space threads usually switch fast

Kernel Space(OS-managed) Thread

OS-supported Multithreading

```
void *Calc(void *)
main() {
        int ret, param;
        pthread_t tid;
        pthread_attr_t attr; pthread_attr_init(&attr);
        pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
        ret=pthread_create(&tid, &attr, Calc, (void *) param);
        ret=pthread_join(tid, NULL); int a, b; b=a+(int) param;
}
Calc(void *param){
        int a, b;
        b = a+(int) param;
        }
```

Scheduling OS-supported Threads

Time slicing

System calls

Traps

The switching overhead stands between processes and user space threads

Dealing with Multithreading

Problems with concurrency

- Race conditions – threads trying to update the same data at the same time

thread 1	thread 2	
A writes X	B writes X	Time

- Deadlock – to avoid race conditions threads lock data

thread 1	thread 2	
A locks X	B locks Y	
A runs Y	B runs X	Time
A unlocks X	B unlocks Y	

- Starvation – low priority threads don’t get scheduled

Coordination among threads

- In parallel computing, threads run **asynchronously** (i.e., at their own pace),
- But, they **synchronize** at certain points, for example accessing global shared memory locations.
- Two types of synchronisation: mutual exclusion and cooperation
- Techniques for synchronisation
  - Mutex (semaphore, lock) to address mutual exclusion
  - wait and notify to address cooperation

Mutual exclusion

- Critical section - section of code that access the global shared data and therefore should be accessed by one thread at a time
- **Mutex** (or lock) - used to enforce mutual exclusion of threads in a critical section
- Mutex has two states: locked and unlocked.
  - Initially unlocked; entering the critical section locks the mutex, unlocks the mutex when exiting the critical section
  - If another thread attempts to access a locked mutex, it blocks until the mutex is unlocked
    - so only one thread can access the critical section at a time

Synchronisation in C

```
pthread_mutex_my_mutex;  /* declared in global area*/


void *Calc(void *);
main(){
        pthread_mutex_init(&my_mutex, NULL) /* before pthread_create*/
        for(ith=0; ith<NUM_THREADS; ith++) pthread_create(… , Calc, …);
}
void *Calc(void *param){
        pthread_mutex_lock(&my_mutex);
        critical section;
        pthread_mutex_unlock(&my_mutex);
}
```

Synchronization in Java – Java Monitor

- **In Java, a monitor is automatically generated for a class or object**

- **The monitor of a class/object can be associated with the methods/statements in the class**

  - **The methods and statements that are associated with a monitor is called synchronized methods or statements.**

- **The synchronised codes are called the monitor region**

  - **The synchronized segment of statements or methods are regarded as the critical section, and therefore can only be run by one thread at a time.**

Synchronization of Java Thread

- **Two types of synchronization**
  - **mutual exclusion**
  - **cooperation**

- **Use keyword "Synchronized" to implement mutual exclusion**

  - **Synchronized method**

  - **Synchronized statements**
- **Use "wait" and "notify" method to achieve cooperation**
  - **wait, notify, notifyAll**

Synchronized methods and statements

- Synchronized method

```
public class ClassA {
        private long c1 = 0;
        private long c2 = 0;
        public synchronized void addc1() { c1++; }
        public synchronized void addc2() { c2++;}
}
```

- addc1 and addc2 methods are associated with the monitor of ClassA;
- **Question: can two threads run addc1 and addc2 methods concurrently?**

- Synchronized statements must specify the object that provides the lock
- synchronized (object_name) { … statements; … }
- The statements are associated with the monitor of the object

Synchronized Statements

- Synchronized statements can be used to improve concurrency with fine-grained synchronization;
- See the following example and note the difference if using synchronized methods

```
public class ClassA {
        private long c1 = 0;
        private long c2 = 0;
        private Object lock1 = new Object();
        private Object lock2 = new Object();
        public void addc1() {
                synchronized (lock1) { c1++; }
        }
        public void addc2() {
                synchronized (lock2) { c2++; }
        }
}
```

How to generate threads in Java

- Using one of the two ways to implement threads in Java
  - Either extend the **Threads** class
  - Or implement the associated **Runnable** interface
- Programmer must extend (implement) a **run** method that specifies what the thread should do
- Create a new instance of the class for each thread and invoke the **start** method
- **start** initializes a new thread of control that executes the **run** method

Example of using Thread Class

```
public class MyThread extends java.lang.Thread
{
        int threadNum;                              // data
        public MyThread(int num)                    // constructor
        {
                threadNum = num;
        }
        public void run()
        {
                for (int i = 0; i<20; i++)
                {
                        System.out.println("Hello from thread " + threadNum);
                }
        }
```

continue to next page…

Example of using Thread Class

```
        public static void main(String[ ] args)
        {
                System.out.println("Simple Thread Demonstration");
                System.out.println(" Extending Thread");
                for (int i = 0; i < 4; i++)
                {
                        MyThread newThread = new MyThread( i );
                        newThread.start( );
                }
        }
}
```

Example of using Runnable interface

```
class RunBasicThread implements Runnable {
    char c;
    RunBasicThread(char c) {
                this.c = c;
    }
    // override run() method in interface
    public void run() {
        for(int i=0; i<100; i++) {
                System.out.print(c);
                try{
                    Thread.sleep((int)(Math.random() * 10));
                } catch( InterruptedException e ) {
                    System.out.println("Exception caught");
                }
        }
    }
    public static void main(String[ ] args) {
                RunBasicThread bt = new RunBasicThread('!');
                RunBasicThread bt1 = new RunBasicThread('*');
                // start RunBasicThread objects as threads
                new Thread(bt).start();
                new Thread(bt1).start();
    }
}
```

Assignment 1 - OpenMP

- Use OpenMP to parallelize the deqn code
  - The overall objective is to achieve good speedup by inserting OpenMP directives in the deqn code
  - You also need to
  - benchmark the runtime of each relevant loop and the runtime of the whole parallel program against the number of threads
  - Analyze the overhead of OpenMP

Assignment 1 - OpenMP

- Write a report
  - Explain in detail what you did with the sequential code
  - benchmark the runtime of each relevant loop and the runtime of the whole parallel program against the number of threads; present the runtimes in graph or table; analyze the results
  - Discuss the iteration scheduling in your program
  - Analyze the overhead of OpenMP
  - Presentation skills, spelling, punctuation and grammar
  - Up to four A4 pages

Submission

- Put all the codes and the report (pdf file) in a package and submit the package through Tabula

- Deadline: 12 noon, Feb 5th, Monday, 2018