High Performance Computing
Course Notes 2009-2010

Shared Memory Parallel Programming
Techniques

- Multiprocessing
- User space multithreading
- Operating system-supported (or kernel) multithreading
- Distributed Shared Memory
Multiprocessing
Features of Processes

A process is associated with a particular program

A process has its own state information

A process has its separate address space
Fork process in C

→ fork() is used to create a child process

○ input and output

→ 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

```c
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 */
    }
}
```
How a new process is created

Before fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

During fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

After fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

Before fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

During fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

After fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

Before fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

During fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

After fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

Before fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

During fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack

After fork
- Parent process: Global Data
- Child process: Global Data
- Executing: Code
- Stack
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 Multithreading
Threads

- Thread – short for thread of execution/control

Before create

Before create

Executing

Global Data

Code

Stack

During create

Executing

Global Data

Code

Stack

Stack

After create

Executing

Global Data

Code

Stack

Stack
Threads

- Used to split a program into separate tasks, one per thread, that can execute concurrently

- “Light weight process”: multiple threads exist within the context of a single process, sharing the process’s code, global information, other resources

- Threads usually communicate by processing shared global data values

  *Compare to the communication between processes*

**global shared space** – global data accessed from single global address space shared among the threads

**local private space** – each thread also has its own local private data that is not shared
Two key functions in C: `pthread_create` and `pthread_join`

```c
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*/
    ret=pthread_join(tid, NULL);    /*wait for the new thread to return*/
}

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

Unlike process switch, there is no time slice for each thread; a thread needs to call thread switch explicitly.

There is a kernel thread running in the background.

When a thread calls an explicit switch, the kernel thread jumps in and schedules the threads in the system.

A thread can hog the CPU so as to starve other threads.

User space threads usually switch fast.
Kernel Space (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);
}

Calc(void *param) {
    int a, b;
    b=a+(int) param;
}
Scheduling OS-supported Threads

Time slicing

System calls

trap

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 structure at the same time

  - Thread 1: \( A \) writes \( X \)
  - Thread 2: \( B \) writes \( X \)

- **Deadlock** – to avoid race conditions threads lock data

  - Thread 1: \( A \) locks \( X \)
  - Thread 2: \( B \) locks \( Y \)

- **Starvation** – low priority threads don’t get scheduled

- **Nondeterminism** – different executions of multi-threaded program yield different results
Coordination among threads

- **asynchronous execution** – threads execute asynchronously, **But**,

- **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 mutex or semaphore or lock) - used to enforce mutual exclusion of threads in a critical section

  Mutex has two states: *locked* and *unlocked*. Initially unlocked, calling a certain statement locks the critical section (so that only the current thread can access it), executes the code, and then unlocks the mutex

  All shared data in a program, i.e. data used in critical sections, must be associated with an appropriate mutex, so locking the mutex locks access to the data

- **If another thread attempts to access a locked mutex, it will block until the mutex is unlocked**
Synchronisation in C

```c
...  
pthread_mutex_t 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);
  ...
}