Introduction

The commodity computers available today have capabilities that were previously only available on large mainframes and expensive workstations. However, new programming techniques may be required in order to be able to take advantage of these capabilities.

Introduction, continued

A series of presentations will cover
- shared memory programming using the pthread library
- distributed memory programming using TCP sockets and MPI
- performance and architectural issues

Shared Memory Programming Outline

- Process state, process creation (30 min)
- Basics of threads (30 min)
- Thread synchronization (60 min)
- Symmetric multiprocessors (45 min)

The Process State Diagram

- A process is a program in execution!

fork() Creates Unix Processes

main()
parent

\[ \text{if (pid = fork()) return 0; else pid = fork();} \]

Parent executes first

Child is an exact copy except for process ID!

Gets control at fork()
Process Creation in Unix

```c
main()
{
    int pid;
    ... if (pid = fork()) != 0 { /* child is here */ }
    else { /* parent is here */ }
}
```

In the parent, the returned value of the fork is the (new) child's process ID.

In child, returned value is 0.

execve() is used to overlay child with a new program.

A typical Unix process tree

```
$ pstree
init+-+crond
    |----httpd ---5* [httpd]
    |----inetd ---in.telnetd ---login ---tcsh +-pстree
    |----login ---bash
    |----lpd
    |----qmail-send + qmail-clean
        |----qmail-lspawn
        |----qmail-rspawn
```

Process status

```
$ ps
PID TTY         TIME  CMD
24897 pts/0    00:00:00 tcsh
24937 pts/0    00:00:00 ps
```

$ ps aux | more 
(to see all processes)
$ top 
(to see busy processes)
$ man command 
(to get more information)

Processes

- Don’t share memory by default
  – can do this with shmat(), but high overhead!
- Have an entry in the process table
- Generally have high overhead for creation and for a context switch
- For more info, see Operating Systems, by Silberschatz and Galvin

Basics of Threads

- Threads are a fundamental tool for shared memory programming!

pthread Library

- POSIX standard thread library
- Include in C, C++ programs
- Portable across all Unix platforms
- Some similarities and some differences with Windows threads
- Fully compatible with Java, Solaris threads
Thread Creation

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

void * hello (void * parm)  {
    printf("Hello World! My parameter is %d\n", (int) parm);
    pthread_exit(0);
}

main() {
    pthread_t tid;
    pthread_create( &tid,   NULL,   hello,   1 );
    pthread_join(tid, NULL);
}
```

To Execute a Thread Program

- Compile in Unix
  ```bash
  $ gcc -o simple simple.c -lpthread
  ```
- Execute
  ```bash
  $ simple  (or ./simple
  if "." not on path)
  ```

### A Thread is a “Lightweight Process”

- Executes and context switches like a process
- Has its own ID and program counter
- Shares code, global variables, open file pointers with creating process and threads
- Has its own local variables, stack space

### Shared Global Variables

```c
int globalvar = 0;
pthread_t tid[3];

void * ChangeVar (void * parm)  {
    printf("I changed globalvar to %d", globalvar);
    globalvar++;
}

main() {
    int i;
    for( i=0; i<3; i++)
        pthread_create( &tid[i], NULL, ChangeVar, NULL );
    for( i=0; i<3; i++ )
        pthread_join( tid[i], NULL );
}
```

### A Context Switch Can Occur Anytime!

```c
int globalvar = 0;

void * ChangeVar (void * parm)  {
    printf("I changed globalvar to %d", globalvar);
    globalvar++;
}

main() {
    int i;
    for( i=0; i<3; i++)
        pthread_create( &tid[i], NULL, ChangeVar, NULL );
    for( i=0; i<3; i++ )
        pthread_join( tid[i], NULL );
}
```
Race Condition

- A race condition occurs whenever the outcome of the program depends on which thread modifies a shared memory location first (i.e., “wins the race”)
- A piece of code that accesses a shared memory location is called a critical section.
- Synchronization is required so that the shared access happens in mutual exclusion

Synchronization is Needed

Example 1: Producer/Consumer Problem
- producers place items into a shared buffer, consumers remove items from the buffer
- producers must not write into a full buffer
- consumers must not remove the same item
This occurs with network printer queues.

Example 2: Reader/Writer Problem
- Writers update a shared data item, readers read the item
- Writers must write in mutual exclusion, any number of readers can read at a time
Occurs with distributed database systems.

Example 3: Barrier Synchronization
- All threads must come to a common stopping place before any can proceed

Thread Synchronization Tools

- mutex variables
- semaphores
- condition variables
Only access to shared variables must be controlled. Local variables in a thread are in private memory.

Mutex Variables

```c
pthread_mutex_t mutex;
```
- Works like the service station key!
- One thread has the “key” at a time.
- Don’t forget to give the key back when finished!
**Mutex Example**

```c
pthread_mutex_t mut;
int globalvar = 0;

void * ChangeGlobalVar(void * parm) {
    int localvar;
    pthread_mutex_lock(&mut);
    localvar = ++globalvar;
    pthread_mutex_unlock(&mut);
    printf("I changed globalvar to %d", localvar);
}

main() {
    pthread_mutex_init(&mut, NULL);
    /* create threads here */
}
```

**Semaphores**

```c
#include <semaphore.h>
sem_t semA, semB;

Two primary operations on semaphores:
- sem_post(&semA);
- sem_wait(&semB);
```

**Semaphores**

- Count open service positions, like at a bank
  - **Sem_wait**: /* if a position is not open, wait */
    ```c
    sem_wait(&semB);
    // if (sem<0) wait;
    sem--;
    // I take the open position */
    sem_post:
    sem++;  /* when I leave, a position is open */
    ```

- Using Semaphores for Resource Allocation
  ```c
  main() {
    sem_init(&res_sem, 0, 5);
    sem_wait(&res_sem);
    /* use resource here */
    sem_post(&res_sem);
  }
  ```

- Using Semaphores for Barrier Synchronization
  ```c
  main() {
    sem_init(&semA, 0, 0);
    sem_init(&semB, 0, 0);
    sem_post(&semB);
    sem_wait(&semA);
    sem_post(&semA);
    sem_wait(&semB);
  }
  ```

- Condition Variables
  - Based on “Monitors”, by C. A. R. Hoare
  - Allow threads to wait for a resource to become available
  - Always used with a mutex
    ```c
    pthread_mutex_t mutex;
    pthread_cond_t notempty, notfull;
    ```
Condition Variables

- Give a thread waiting for a resource the first opportunity to use the mutex when the resource becomes available.

**mutex**

- A waits on condition
- A enters critical section
- B releases resource
- A obtains resource inside critical section

<table>
<thead>
<tr>
<th>Condition Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

Using Conditions Variables for Producer/Consumer

**Producer**

```c
/* produce item in local buffer */
pthread_mutex_lock(&mut);
while (/* buffer is full */)  
    pthread_cond_wait(&notfull, &mut);
/* put an item in buffer */
pthread_cond_signal(&notempty);
pthread_mutex_unlock(&mut);
```

**Consumer**

```c
/* consume item */
pthread_mutex_lock(&mut);
while (/* buffer is empty */)  
    pthread_cond_wait(&notempty, &mut);
/* get an item from buffer */
pthread_cond_signal(&notfull);
pthread_mutex_unlock(&mut);
```

Shared Memory Programming

We have covered:

- Process state, process creation
- Basics of pthreads
- Thread synchronization using mutex, semaphores, and condition variables

Useful pthread information

- Getting Started With POSIX Threads, by Tom Wagner and Don Towsley
  http://centaurus.cs.umass.edu/~wagner/threads_html/tutorial.html
- On-line thread tutorial from Sun
- Programming With Posix Threads, by David R. Butenhof (Addison-Wesley Professional Computing Series)

What is an SMP? (Symmetric Multiprocessor)

- A computer with more than one CPU
- Disk subsystem, network, main memory, I/O devices, … are all equally accessible to all processors
- However, each processor has its own private cache

Cache Memory

- Very fast memory, close to CPU
Cache Memory

- Makes main memory appear faster (on the average)
- If an item is in the cache, get it
- Otherwise, there is a cache miss and the item must be retrieved from memory
- Cache is 20 times (or more!) faster than memory

Cache Memory

- Works because of
  - spatial locality (next item to be accessed is likely to be close by)
  - temporal locality (this item is likely to be accessed again soon)

Think of the way we program, using loops, array access . . .

Cache Memory

- Is organized in cache lines
- A miss loads a cache line from memory

Any of these memory "lines"

Can be loaded into this cache location (in a direct mapped cache)

Cache Memory

- If a line is cache needs to be replaced, then it must be copied back to memory

The location in memory is wrong until the cache line is copied back!

SMP’s and Cache Memory

- SMP cache and memory can be wrong! -- the cache coherence problem

new (correct) value

old (incorrect) values!

Snoopy Bus

- Most common solution to the cache coherence problem on SMP’s
- The bus watches all reads and writes
- A cache miss causes the bus to broadcast a request for the newest value
- A write sends an invalidate message

Copyright, 1999 © Amy Apon, Ph.D.
Shared Memory Parallel Programming
**Snoopy Bus**
- Is a very busy bus!
- Works well for two (four?) processors, but is a classic “Von Neumann bottleneck”
- Does not perform well as the number of the processors in the SMP gets larger!

**Symmetric Multiprocessor**
- Composed of two or more (symmetric) processors, one of every other subsystem
- Each processor has a private cache
- Snoopy bus is the most common solution to the cache coherence problem

**Symmetric Multiprocessors**
- Can degrade in performance as the number of processes increases
- Performance depends on the amount of data that is shared in the application!

**Symmetric Multiprocessors**
- For more information, see:
  - Or a book on computer architecture

**Presentation Two**
**Distributed Memory Programming**
- Distributed memory processing
- TCP client/server examples
- How MPI works over TCP
- Programming in MPI
- MPI set up, further information