CSC 4304 - Systems Programming
Fall 2010

LECTURE - XIII

CONCURRENT PROGRAMMING - I

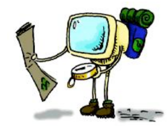
Tevfik Koşar

Louisiana State University

October 28th, 2010

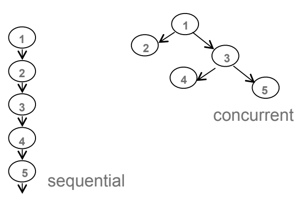
Roadmap

- Sequential vs Concurrent Programming
- Shared Memory vs Message Passing
- Divide and Compute
- Threads vs Processes
- POSIX Threads



Concurrent Programming

- So far, we have focused on sequential programming: all computational tasks are executed in sequence, one after the other.
- Today, we will start learning concurrent programming: multiple computational tasks are executed simultaneously, at the same time.



Concurrent Programming

- Implementation of concurrent tasks:
  - as separate programs
  - as a set of processes or threads created by a single program
- Execution of concurrent tasks:
  - on a single processor
  - ➔ Multithreaded programming
  - on several processors in close proximity
  - ➔ Parallel computing
  - on several processors distributed across a network
  - ➔ Distributed computing

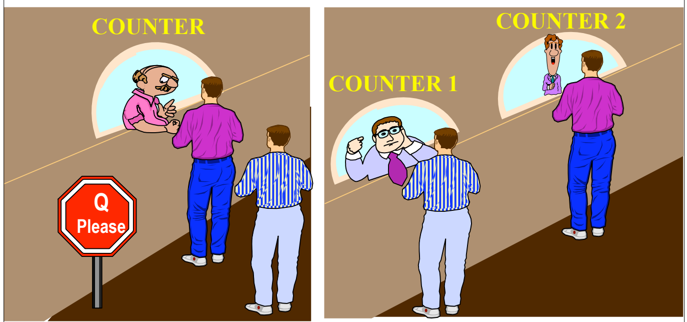
Why Threads?

- In certain cases, a single application may need to run several tasks at the same time
  - Creating a new process for each task is time consuming
  - Use a single process with multiple threads
    - faster
    - less overhead for creation, switching, and termination
    - share the same address space

Motivation

- Increase the performance by running more than one tasks at a time.
  - divide the program to n smaller pieces, and run it n times faster using n processors
- To cope with independent physical devices.
  - do not wait for a blocked device, perform other operations at the background

Serial vs Parallel



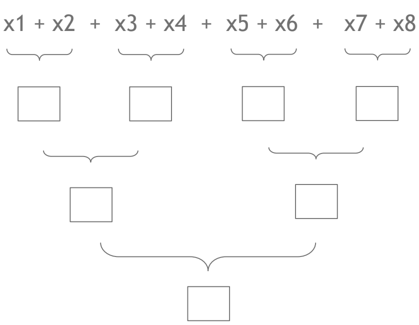
Divide and Compute

x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8

How many operations with sequential programming?
7

Step 1: x1 + x2
Step 2: x1 + x2 + x3
Step 3: x1 + x2 + x3 + x4
Step 4: x1 + x2 + x3 + x4 + x5
Step 5: x1 + x2 + x3 + x4 + x5 + x6
Step 6: x1 + x2 + x3 + x4 + x5 + x6 + x7
Step 7: x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8

Divide and Compute



Step 1: parallelism = 4

Step 2: parallelism = 2

Step 3: parallelism = 1

Gain from parallelism

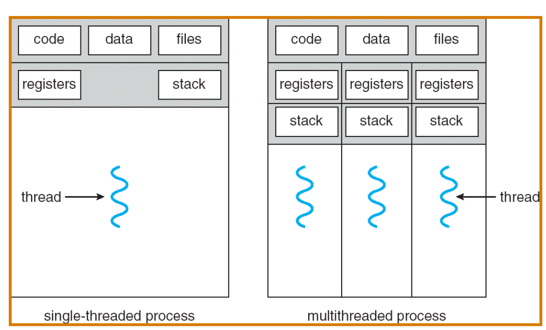
In theory:
- dividing a program into n smaller parts and running on n processors results in n time speedup

In practice:
- This is not true, due to
  - Communication costs
  - Dependencies between different program parts
    - Eg. the addition example can run only in log(n) time not 1/n

Prevent Blocking

- Do not wait for a blocked device, perform other operations at the background
  - During I/O perform computation
  - During continuous visualization, handle key strokes and I/O
    - Eg. video games
  - While listening to network, perform other operations
    - Listening to multiple sockets at the same time
  - Concurrent I/O, concurrent transfers
    - Eg. Web browsers

Single and Multithreaded Processes



Communication Between Tasks

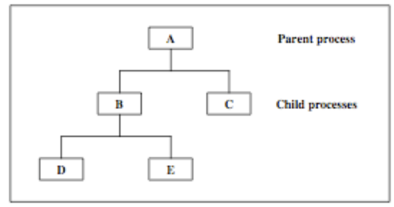
Interaction or communication between concurrent tasks can done via:

- Shared memory:
  - all tasks has access to the same physical memory
  - they can communicate by altering the contents of shared memory
- Message passing:
  - no common/shared physical memory
  - tasks communicate by exchanging messages

Multi-process model

Process Spawning:

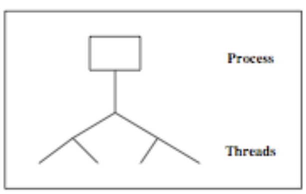
Process creation involves the following four main actions:
• setting up the process control block,
• allocation of an address space and
• loading the program into the allocated address space and
• passing on the process control block to the scheduler



Multi-thread model

Thread Spawning:

• Threads are created *within and belonging to* processes
• All the threads created within one process share the resources of the process including the address space
• Scheduling is performed on a per-thread basis.
• The thread model is a *finer grain scheduling model* than the process model
• Threads have a similar *lifecycle* as the processes and will be managed mainly in the same way as processes are



Threads vs Processes

- A common terminology:
  - Heavyweight Process = Process
  - Lightweight Process = Thread

Advantages (Thread vs. Process):

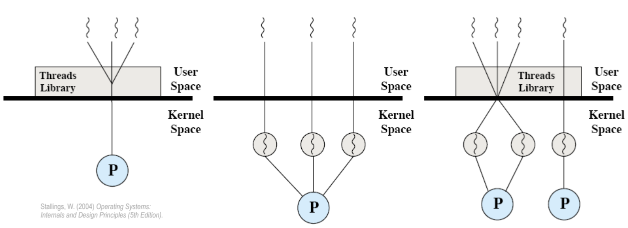
- Much quicker to create a thread than a process
  - spawning a new thread only involves allocating a new stack and a new CPU state block
- Much quicker to switch between threads than to switch between processes
- Threads share data easily

Disadvantages (Thread vs. Process):

- Processes are more flexible
  - They don't have to run on the same processor
- No security between threads: One thread can stomp on another thread's data
- For threads which are supported by user thread package instead of the kernel:
  - If one thread blocks, all threads in task block.

Thread Implementation

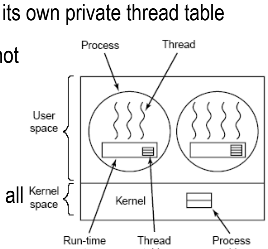
- <u>Two broad categories of thread implementation</u>
  - ✓ User-Level Threads (ULTs)
  - ✓ Kernel-Level Threads (KLTs)



Pure user-level (ULT), pure kernel-level (KLT) and combined-level (ULT/KLT) threads

Thread Implementation

- <u>User-Level Threads (ULTs)</u>
  - ✓ the kernel is not aware of the existence of threads, it knows only processes with one thread of execution (one PC)
  - ✓ each user process manages its own private thread table
  - <u>light thread switching:</u> does not need kernel mode privileges
  - <u>cross-platform:</u> ULTs can run on any underlying O/S
  - <u>if a thread blocks, the entire process is blocked</u>, including all other threads in it

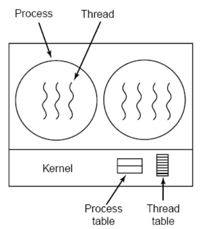


Tanenbaum, A. S. (2001) Modern Operating Systems (2nd Edition)

A user-level thread package

Thread Implementation

- **Kernel-Level Threads**
  - the kernel knows about and manages the threads: creating and destroying threads are system calls
    - fine-grain scheduling, done on a thread basis
    - if a thread blocks, another one can be scheduled without blocking the whole process
    - heavy thread switching involving mode switch



A kernel-level thread package

Tanenbaum, A. S. (2001) Modern Operating Systems (2nd Edition)

Thread Creation

- **pthread_create**

```
// creates a new thread executing start_routine
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

- **pthread_join**

```
// suspends execution of the calling thread until the target
// thread terminates
int pthread_join(pthread_t thread, void **value_ptr);
```


Thread Example

```
    main()
{
pthread_t thread1, thread2;  /* thread variables */

pthread_create(&thread1, NULL, (void *) &print_message_function,(void*)"hello ");
pthread_create(&thread2, NULL, (void *) &print_message_function,(void*)"world!");

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);

printf("\n");
exit(0);
}
```

Why use pthread_join?
To force main block to wait for both threads to terminate, before it exits.
If main block exits, both threads exit, even if the threads have not finished their work.

Thread Example *(cont.)*

```
    void print_message_function ( void *ptr )
{
    char *cp = (char*)ptr;
    int i;
    for (i=0;i<3;i++){
        printf("%s \n", cp);
        fflush(stdout);
        sleep(1);
    }

    pthread_exit(0); /* exit */
}
```


Example: Interthread Cooperation

```
void* print_count ( void *ptr );
void* increment_count ( void *ptr );

int NUM=5;
int counter =0;

int main()
{
   pthread_t thread1, thread2;

   pthread_create (&thread1, NULL, increment_count, NULL);
   pthread_create (&thread2, NULL,  print_count, NULL);

   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   exit(0);
}
```


Interthread Cooperation *(cont.)*

```
void* print_count ( void *ptr )
{
   int i;
   for (i=0;i<NUM;i++){
      printf("counter = %d \n", counter);
      //sleep(1);
   }
   pthread_exit(0);
}

void* increment_count ( void *ptr )
{
   int i;
   for (i=0;i<NUM;i++){
      counter++;
      //sleep(1);
   }
   pthread_exit(0);
}
```

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), J.Kim (KAIST), and J. Schaumann (SIT).