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.

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

Roadmap

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

Divide and Compute

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

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 \( \frac{1}{n} \)

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

<table>
<thead>
<tr>
<th>code</th>
<th>data</th>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

single-threaded process  multithreaded process
Communication Between Tasks

Interaction or communication between concurrent tasks can be done via:

- **Shared memory:**
  - All tasks have 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
- Loading the program into the allocated address space
- Passing on the process control block to the scheduler

Thread Implementation

- Two broad categories of thread implementation
  - User-Level Threads (ULTs)
  - Kernel-Level Threads (KLTs)

**User-Level Threads (ULTs):**

- 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
- Light thread switching: does not need kernel mode privileges
- Cross-platform: ULTs can run on any underlying OS
- If a thread blocks, the entire process is blocked, including all other threads in it

**Kernel-Level Threads (KLTs):**

- The kernel is aware of the existence of threads
- Each process has its own thread list
- Each thread has its own control block
- Thread switching is done at kernel level
- Cross-platform: KLTs can run on any underlying OS
- If a thread blocks, the entire process is blocked, including all other threads in it
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

---

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

```c
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!");
    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.)

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

---

Example: Interthread Cooperation

```c
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.)

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

```c
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).