Lecture 12: Threads and Concurrency

Mythili Vutukuru
IIT Bombay
Single threaded process

• So, far we have studied **single threaded programs**
• Recap: process execution
  – PC points to current instruction being run
  – SP points to stack frame of current function call
• A program can also have multiple threads of execution
• What is a thread?
Multi threaded process

• A thread is like another copy of a process that executes independently
• Threads share the same address space (code, heap)
• Each thread has separate PC
  — Each thread may run over different part of the program
• Each thread has separate stack for independent function calls
Process vs. threads

• Parent P forks a child C
  – P and C do not share any memory
  – Need complicated IPC mechanisms to communicate
  – Extra copies of code, data in memory

• Parent P executes two threads T1 and T2
  – T1 and T2 share parts of the address space
  – Global variables can be used for communication
  – Smaller memory footprint

• Threads are like separate processes, except they share the same address space
Why threads?

- **Parallelism**: a single process can effectively utilize multiple CPU cores
  - Understand the difference between concurrency and parallelism
  - **Concurrency**: running multiple threads/processes at the same time, even on single CPU core, by interleaving their executions
  - **Parallelism**: running multiple threads/processes in parallel over different CPU cores
- **Even if no parallelism**, concurrency of threads ensures effective use of CPU when one of the threads blocks (e.g., for I/O)
Scheduling threads

• OS schedules threads that are ready to run independently, much like processes
• The context of a thread (PC, registers) is saved into/restored from thread control block (TCB)
  – Every PCB has one or more linked TCBs
• Threads that are scheduled independently by kernel are called kernel threads
  – E.g., Linux pthreads are kernel threads
• In contrast, some libraries provide user-level threads
  – User program sees multiple threads
  – Library multiplexes larger number of user threads over a smaller number of kernel threads
  – Low overhead of switching between user threads (no expensive context switch)
  – But multiple user threads cannot run in parallel
Creating threads using pthreads API

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

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Figure 26.2: Simple Thread Creation Code (t0.c)
Example: threads with shared data

```c
static volatile int counter = 0;

void *mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()

int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```
Threads with shared data: what happens?

- What do we expect? Two threads, each increments counter by $10^7$, so $2 \times 10^7$

```bash
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

- Sometimes, a lower value. Why?

```bash
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```
What is happening?

- Assembly code of
  \[\text{counter} = \text{counter} + 1\]

<table>
<thead>
<tr>
<th>OS</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>(after instruction)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>before critical section</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov 0x8049a1c, %eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add $0x1, %eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>interrupt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>save T1's state</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>restore T2's state</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov 0x8049a1c, %eax</td>
<td>mov 0x8049a1c, %eax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add $0x1, %eax</td>
<td>add $0x1, %eax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>interrupt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>save T2's state</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>restore T1's state</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 26.7: The Problem: Up Close and Personal
Race conditions and synchronization

• What just happened is called a race condition
  – Concurrent execution can lead to different results
• Critical section: portion of code that can lead to race conditions
• What we need: mutual exclusion
  – Only one thread should be executing critical section at any time
• What we need: atomicity of the critical section
  – The critical section should execute like one uninterruptible instruction
• How is it achieved? Locks (topic of next lecture)