Computer Systems

Introduction to Threads

Race Conditions

Single- vs. Multi-Threaded Processes

*single-threaded*  
*multi-threaded*
What are Threads?

- A thread is an independent stream of instructions
  - Basic unit of CPU utilization

- A thread has its own
  - thread ID
  - execution stack

- A thread shares with its sibling threads
  - The code, data and heap section
  - Other OS resources, such as open files and signals

Multi-Threaded Processes

- Each thread has a private stack

- But threads share the process address space!

- There is no memory protection!

- Threads could potentially write into each other’s stack
Why use Threads?

- A specific example, a Web server:

  in an infinite loop
  
  {  
  get web page request from client
  check if page exists and client has permissions
  create a thread to transmit web page back to client
  }

Concurrent execution on a single-core system

Parallel execution on a dual-core system
**pthreads**

- Refers to the POSIX standard (IEEE 1003.1c)
- API for thread creation and synchronization
- Common in UNIX operating systems

---

**Java Threads**

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

- JVM manages Java threads
  - Creation
  - Execution
  - Etc.
BIG Issue:
Shared Data,
which leads to
Race Conditions

An Example: badcnt.c

```c
#define NITERS 1000000

unsigned int cnt = 0; /* shared */
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, Count, NULL);
  pthread_create(&tid2, NULL, Count, NULL);

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  if (cnt != (unsigned)NITERS*2)
    printf("BOOM! cnt=%d\n", cnt);
  else
    printf("OK cnt=%d\n", cnt);
}

/* thread routine */
void * Count(void *arg) {
  int i;
  for (i=0; i<NITERS; i++)
    cnt++;
  return NULL;
}
```

```bash
linux> ./badcnt
BOOM! cnt=1988411
linux> ./badcnt
BOOM! cnt=1982618
linux> ./badcnt
BOOM! cnt=1982696
```

cnt should be
equal to 2,000,000.
What went wrong?!
Critical Sections

- Critical section = block of code that accesses *shared data*

```c
/* thread routine */
void * count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++; /* global, shared */
    return NULL;
}
```

- The statement `cnt++` is **non-atomic**: at machine level, it translates into three machine instructions

  ```
  mov R, cnt ; atomic (no interrupts)
  add R, 1 ; atomic (no interrupts)
  mov cnt, R ; atomic (no interrupts)
  ```

Race Condition

- First thread executes `cnt++` implemented as
  ```
  register1 = cnt
  register1 = register1 + 1
  cnt = register1
  ```

- Second thread executes `cnt++` implemented as
  ```
  register2 = cnt
  register2 = register2 + 1
  cnt = register2
  ```

- Consider this execution interleaving with “cnt = 5” initially:
  ```
  first thread executes register1 = cnt {register1 = 5}
  first thread executes register1 = register1 + 1 {register1 = 6}
  Timer Interrupt
  second thread executes register2 = cnt {register2 = 5}
  second thread executes register2 = register2 + 1 {register2 = 6}
  Timer Interrupt
  first thread executes cnt = register1 {cnt = 6}
  second thread executes cnt = register2 {cnt = 6}
  ```
Race Conditions

- Race Conditions
  - occur when several threads access shared data concurrently
  - *the result may vary from one execution to the next*

- We’d like to execute this instruction *atomically*:

```
cnt++; println
```

<table>
<thead>
<tr>
<th>machine level</th>
<th>cnt = R</th>
</tr>
</thead>
<tbody>
<tr>
<td>R = cnt</td>
<td></td>
</tr>
<tr>
<td>R = R + 1</td>
<td></td>
</tr>
<tr>
<td>cnt = R</td>
<td></td>
</tr>
</tbody>
</table>

(An *atomic* operation completes without interruption)

- Need some kind of mutual exclusion primitives
  - only a thread should ever enter a critical section (next lecture)

Practice Exercise

- Consider two threads sharing a *global* variable *count*, with initial value equal to 10:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>count++;</td>
<td>count--;</td>
</tr>
</tbody>
</table>

- What are the possible values for *count* after both threads finish executing:
Hands-on Session

- Complete the hands-on POSIX programming exercises posted on the class website
  - Creating threads
  - Passing data to threads
    - `pthread_create`
    - `pthread_exit`
    - `pthread_self`
    - `pthread_join`
    - `pthreadAttach`
    - `pthread_detach`