CSC 1600: Operating Systems

Thread Concurrency

Concurrent Threads

- Concurrent threads come into conflict with each other when they come to use shared resources
- Atomic actions are indivisible. In hardware, loads and stores are indivisible.
- On a processor, a thread switch can occur between any two atomic actions; thus the atomic actions of concurrent threads may be interleaved in any possible order.
- Result of concurrent execution should not depend on the order in which atomic instructions are interleaved.

badcnt.c: An Incorrect Program

```
#define NITERS 100000000

unsigned int count = 0; /* shared */

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL,
                   Increment, NULL);
    pthread_create(&tid2, NULL,
                   Increment, NULL);

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

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

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

```
linux> ./badcnt
BOOM! count=198841183

linux> ./badcnt
BOOM! count=198261801

linux> ./badcnt
BOOM! count=198269672
```

cnt should be equal to 200,000,000. What went wrong?!

Atomic Operations

- The statement

  `cnt++;` —machine level→

  ```
  R = cnt
  R = R + 1
  cnt = R
  ```

  must be performed atomically (cnt is shared)

- An atomic operation is an operation that completes in its entirety, without interruption.

Race Conditions

- One possible interleaving of statements is:

```
/*T1*/  R1 = in
/*T1*/  R1 = R1 + 1
        <timer interrupt!>
/*T2*/  R2 = in
/*T2*/  R2 = R2 + 1
/*T2*/  cnt = R2
        <timer interrupt!>
/*T1*/  cnt = R1
```

Race condition:

The situation where several processes access shared data concurrently. The final value of the shared data may vary from one execution to the next.

- Then `cnt` ends up incremented once only! Threads overwrite each other's data.

Critical Sections

- Critical sections are blocks of code that access shared data.

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

- The objective is to make critical sections behave as if they are atomic operations: if one process uses a shared piece of data, other processes can't access it.