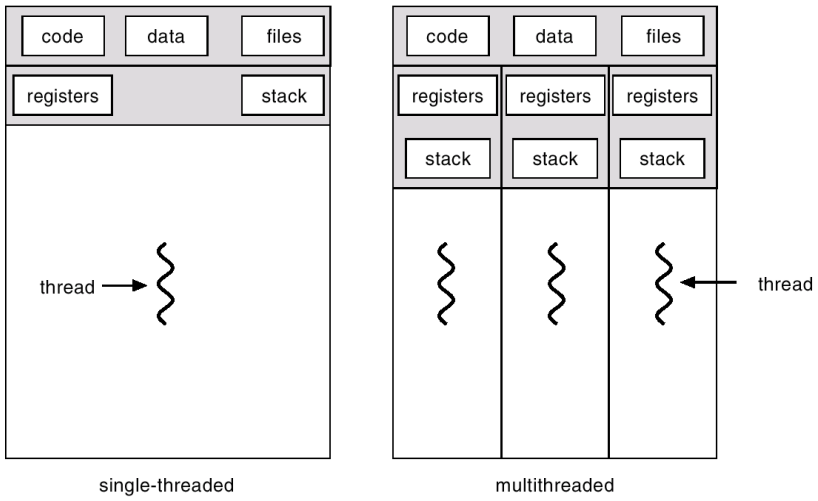
Synchronizing Threads

with

Semaphores

Review: Multi-Threaded Processes



Review: Race Conditions

- Assume that two threads execute concurrently

```
cnt++;        /* cnt is global shared data */
```

- One possible interleaving of statements is:

```
R1 = cnt
R1 = R1 + 1
<timer interrupt ! >
R2 = cnt
R2 = R2 + 1
in = R2
<timer interrupt !>
cnt = R1
```

Race condition:

The situation where several threads 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!

Race conditions

- A *race* occurs when the correctness of the program depends on one thread reaching a statement before another thread.

- No races should occur in a program.

Review: Critical Sections

- Blocks of code that access shared data:

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

- Threads must have mutual exclusive access to critical sections: two threads cannot simultaneously execute **`cnt++`**

The Critical-Section Problem

- To prevent races, concurrent threads must be synchronized. General structure of a thread T_i :

```
while(1) {
    entry section                    Thread T_i

        Critical Section (CS)

    exit section

        Remainder Section (RS)
}
```

- Critical section problem: design mechanisms that allow a single thread to be in its CS at one time

```
while(1) {
    entry section
        Critical Section (CS)
    exit section
        Remainder Section (RS)
}
```

Thread T_1

```
while(1) {
    entry section
        Critical Section (CS)
    exit section
        Remainder Section (RS)
}
```

Thread T_2

Solving the CS Problem - Semaphores (1)

- A semaphore is a synchronization tool provided by the operating system.
- A semaphore S can be viewed as an integer variable that can be accessed through 2 *atomic* operations:

`DOWN(S)`	also called	`wait(S)`
`UP(S)`	also called	`signal(S)`

Atomic means *indivisible*.

- When a thread has to wait, put it in a *queue of blocked threads* waiting on the semaphore.

Semaphores (2)

- In fact, a semaphore is a structure:

```
struct Semaphore {
        int count;
        Thread * queue;      /*  blocked */
};                            /* threads  */
struct Semaphore S;
```

- **`S.count`** must be initialized to a nonnegative value (depending on application)

OS Semaphores - DOWN(S) or wait(S)

- When a process must wait for a semaphore S, it is blocked and put on the semaphore's queue

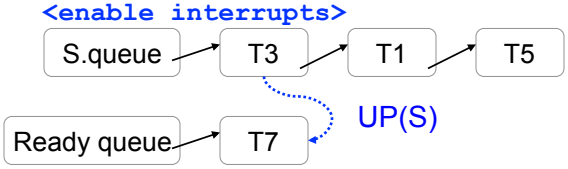
```
DOWN(S):
  <disable interrupts>
  S.count--;
  if (S.count < 0) {
    block this thread
    place this thread in S.queue
  }
  <enable interrupts>
```

- Threads waiting on a semaphore S:

OS Semaphores - UP(S) or signal(S)

- The UP or signal operation removes one thread from the queue and puts it in the list of ready threads

```
UP(S):
  <disable interrupts>
  S.count++;
  if (S.count <= 0) {
    remove a thread T from S.queue
    place this thread T on Ready list
  }
  <enable interrupts>
```



OS Semaphores - Observations

- When `S.count >= 0`
  - the number of threads that can execute `wait(s)` without being blocked is equal to `S.count`
- When `S.count < 0`
  - the number of threads waiting on S is equal to |`S.count`|
- Atomicity and mutual exclusion
  - no 2 threads can be in `wait(S)` and `signal(S)` (on the same `S`) at the same time
  - hence the blocks of code defining `wait(S)` and `signal(S)` are, in fact, critical sections

Using Semaphores to Solve CS Problems

```
Thread Ti:

   DOWN(S);
   <critical section CS>
   UP(S);
   <remaining section RS>
```

- To allow only one thread in the CS (mutual exclusion):
  - initialize **S.count** to ____
- What should be the value of s to allow k threads in CS?
  - initialize **S.count** to ____

Solving the Earlier Problem

```
/* Thread T1 routine */

void * count(void *arg)
{
    int i;

    for (i=0; i<10; i++)
    {

        cnt++;

    }
    return NULL;
}
```

```
/* Thread T2 routine */

void * count(void *arg)
{
    int i;

    for (i=0; i<10; i++)
    {

        cnt++;

    }
    return NULL;
}
```

- Solution: use Semaphores to impose mutual exclusion to executing cnt++

How are Races Prevented?

Semaphore mutex; 1

```
/* Thread T1 routine */

void * count(void *arg)
{
    int i;

    for (i=0; i<10; i++ {
        DOWN(mutex);
        R1 ← cnt
        R1 ← R1+1
        cnt ← R1
        UP(mutex);
    }
    return NULL;
}
```

```
/* Thread T2 routine */

void * count(void *arg)
{
    int i;

    for (i=0; i<10; i++) {
        DOWN(mutex);
        R2 ← cnt
        R2 ← R2+1
        cnt ← R2
        UP(mutex);
    }
    return NULL;
}
```

Solving the Earlier Problem

Semaphore mutex; 1

```
/* Thread T1 routine */

void * count(void *arg)
{
    int i;

    for (i=0; i<10; i++)
    {
        DOWN(mutex);
        cnt++;
        UP(mutex);
    }
    return NULL;
}
```

```
/* Thread T2 routine */

void * count(void *arg)
{
    int i;

    for (i=0; i<10; i++)
    {
        DOWN(mutex);
        cnt++;
        UP(mutex);
    }
    return NULL;
}
```

Exercise - Understanding Semaphores

- What are possible values for g, after the three threads below finish executing?

```
// global shared variables
int g = 10;

Semaphore s = 0;
```

Thread A

```
g = g * 2;
```

Thread B

```
g = g + 1;
UP(s);
```

Thread C

```
DOWN(s);
g = g - 2;
UP(s);
```

```
int g = 10;
Semaphore s = 0;
```

Thread A

```
R1 = g
R1 = R1*2
G = R1
```

Thread B

```
R2 = g
R2 = R2+1
g=R2
UP(s);
```

Thread C

```
DOWN(s);
R3=g
R3=R2-2
g=R3
UP(s);
```

Using OS Semaphores

- Semaphores have two uses:
  - Mutual exclusion: making sure that only one thread is in a critical section at one time
  - Synchronization: making sure that T1 completes execution before T2 starts?

Synchronizing Threads with Semaphores

- Suppose that we have 2 threads: T1 and T2
- How can we ensure that a statement S1 in T1 executes before statement S2 in T2?

```
Semaphore sync;   0
```

Thread T1

```
S1;

UP(sync);
```

Thread T2

```
DOWN(sync);

S2;
```

Exercise

- Consider two concurrent threads T1 and T2. T1 executes statements S1 and S3 and T2 executes statement S2.

T1	T2
S1	S2
S3	

- Use semaphores to ensure that S1 always gets executed before S2 and S2 always gets executed before S3

Review: Mutual Exclusion

```
Semaphore mutex;   1

Thread T1

      DOWN(mutex);
      critical section
      UP(mutex);

Thread T2

      DOWN(mutex);
      critical section
      UP(mutex);
```

Review: Synchronization

- T2 cannot begin execution until T1 has finished:

```
Semaphore mutex;   0

Thread T1

        Code for T1
        UP(mutex);

Thread T2

        DOWN(mutex);
        Code for T2
```

Summary

- Problem with Threads:
  - Races
- Eliminating Races:
  - Mutual Exclusion with Semaphores
- Thread Synchronization:
  - Use Semaphores
- POSIX Threads