Synchronizing Threads with POSIX Semaphores

1. Why semaphores?
   2. Posix semaphores are easy to use
      - `sem_init`
      - `sem_wait`
      - `sem_post`
      - `sem_getvalue`
      - `sem_destroy`

3. Activities 1 2

Now it is time to take a look at some code that does something a little unexpected. The program `badcnt.c` creates two new threads, both of which increment a global variable called `cnt` exactly `NITER`, with `NITER = 1,000,000`. But the program produces unexpected results.

**Activity 1.** Create a directory called `posixsem` in your class Unix directory. Download in this directory the code `badcnt.c` and compile it using

```
gcc badcnt.c -o badcnt -lpthread
```

Run the executable `badcnt` and observe the output. Try it on both `tanner` and `felix`.

Quite unexpected! Since `cnt` starts at 0, and both threads increment it `NITER` times, we should see `cnt` equal to `2*NITER` at the end of the program. What happens?

Threads can greatly simplify writing elegant and efficient programs. However, there are problems when multiple threads share a common address space, like the variable `cnt` in our earlier example.

To understand what might happen, let us analyze this simple piece of code:

```
THREAD 1                      THREAD 2
a = data;                     b = data;
a++;                          b--;
data = a;                      data = b;
```

Now if this code is executed serially (for instance, THREAD 1 first and then THREAD 2), there are no problems. However threads execute in an arbitrary order, so consider the following situation:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = data;</code></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td><code>a = a+1;</code></td>
<td><code>b = data; // 0</code></td>
<td>0</td>
</tr>
<tr>
<td><code>---</code></td>
<td><code>b = b + 1;</code></td>
<td>0</td>
</tr>
<tr>
<td><code>data = a; // 1</code></td>
<td><code>---</code></td>
<td>1</td>
</tr>
<tr>
<td><code>---</code></td>
<td><code>data = b; // 1</code></td>
<td>1</td>
</tr>
</tbody>
</table>

So data could end up +1, 0, -1, and there is **NO WAY** to know which value! It is completely non-deterministic!
The solution to this is to provide functions that will block a thread if another thread is accessing data that it is using.

Pthreads may use semaphores to achieve this.

---

### Posix semaphores

All POSIX semaphore functions and types are prototyped or defined in `semaphore.h`. To define a semaphore object, use

```c
sem_t sem_name;
```

To initialize a semaphore, use `sem_init`:

```c
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- `sem` points to a semaphore object to initialize
- `pshared` is a flag indicating whether or not the semaphore should be shared with fork()ed processes. LinuxThreads does not currently support shared semaphores
- `value` is an initial value to set the semaphore to

Example of use:

```c
sem_init(&sem_name, 0, 10);
```

To wait on a semaphore, use `sem_wait`:

```c
int sem_wait(sem_t *sem);
```

Example of use:

```c
sem_wait(&sem_name);
```

- If the value of the semaphore is negative, the calling process blocks; one of the blocked processes wakes up when another process calls `sem_post`.

To increment the value of a semaphore, use `sem_post`:

```c
int sem_post(sem_t *sem);
```

Example of use:

```c
sem_post(&sem_name);
```

- It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

To find out the value of a semaphore, use

```c
int sem_getvalue(sem_t *sem, int *valp);
```
POSIX Semaphores

- gets the current value of sem and places it in the location pointed to by valp

Example of use:

```c
int value;
sem_getvalue(&sem_name, &value);
printf("The value of the semaphors is %d\n", value);
```

To destroy a semaphore, use

```c
int sem_destroy(sem_t *sem);
```

- destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use:

```c
sem_destroy(&sem_name);
```

### Using semaphores - a short example

Consider the problem we had before and now let us use semaphores:

Declare the semaphore global (outside of any function):

```c
sem_t mutex;
```

Initialize the semaphore in the main function:

```c
sem_init(&mutex, 0, 1);
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem_wait (&amp;mutex);</td>
<td>---</td>
<td>0</td>
</tr>
<tr>
<td>---</td>
<td>sem_wait (&amp;mutex);</td>
<td>0</td>
</tr>
<tr>
<td>a = data;</td>
<td>*/ blocked */</td>
<td>0</td>
</tr>
<tr>
<td>a = a+1;</td>
<td>*/ blocked */</td>
<td>0</td>
</tr>
<tr>
<td>data = a;</td>
<td>*/ blocked */</td>
<td>1</td>
</tr>
<tr>
<td>sem_post (&amp;mutex);</td>
<td>*/ blocked */</td>
<td>1</td>
</tr>
<tr>
<td>/* blocked */</td>
<td>b = data;</td>
<td>1</td>
</tr>
<tr>
<td>/* blocked */</td>
<td>b = b + 1;</td>
<td>1</td>
</tr>
<tr>
<td>/* blocked */</td>
<td>data = b;</td>
<td>2</td>
</tr>
<tr>
<td>/* blocked */</td>
<td>sem_post (&amp;mutex);</td>
<td>2</td>
</tr>
</tbody>
</table>

[data is fine. The data race is gone.]

**Activity 2.** Use the example above as a guide to fix the program `badcnt.c`, so that the program always produces the expected output (the value `2*NITER`). Make a copy of `badcnt.c` into `goodcnt.c` before you modify the code.

http://www.csc.villanova.edu/~mdamian/threads/posixsem.html
To compile a program that uses pthreads and posix semaphores, use

```bash
gcc -o filename filename.c -lpthread -lrt
```