CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Classic Synchronization Problems

Now that we have these tools, how are we going to use them?

Look at three well-studied synchronization problems to understand common ways to use semaphores

Note that we will use semaphores to implement mutual exclusion, but we could just as easily use mutexes for that purpose

Producer-Consumer Problem

Also known as bounded-buffer problem

We've seen this one before -- two processes have a shared buffer, with one writing to the buffer (producing) and the other reading from the buffer (consuming)

Semaphores can make our lives easier here

In addition to shared buffer, assume process share these values for synchronization

```
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

Producer:

```
do {
    ...
    /* produce an item in next produced */
    ...

    wait(empty);
    wait(mutex);

    ...
    /* add next produced to the buffer */
    ...

    signal(mutex);
    signal(full);
} while (true);
```

Notice the two distinct ways we use semaphores

`empty` and `full` keep track of available resources (empty buffer slots and full buffer slots)

`mutex` provides mutual exclusion

Helpful to have both types

Readers-Writers Problem

Database is shared among several processes

Readers only need to read from database

Writers need to read and write to database

Any number of readers can access the database simultaneously without issue

A writer needs exclusive access while writing -- no other readers or writers can access at same time

We add additional rule that readers will not wait until a writer is active

If a writer requests access to the database, it must wait for all readers to finish

Our rule implies that a new reader can start reading even though the writer is waiting

This can result in writer starvation. There are other variants that can result in reader starvation or that allow starvation-free solutions

Processes share the following data:

```
semaphore rw_mutex = 1
semaphore mutex = 1
int read_count = 0;
```

Writer process is quite simple

```
do {
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
} while (true);
```

Reader process

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

`rw_mutex` protects access to database

Multiple readers can effectively "share" `rw_mutex`

`mutex` protects read count and ensures just one reader will try to grab `rw_mutex`

If a writer is in its critical section, only one reader will wait on `rw_mutex`. The rest will wait on `mutex`

When a writer signals `rw_mutex`, scheduler will choose whether another writer or a reader is woken up

Some systems have special reader-writer locks to solve this problem directly

Processes request the lock in either read mode or write mode, and system ensures that writers get exclusive access but allows several processes to have read-mode access at once

We will look at third classic problem next time