CPS 356 Lecture notes: Classical Problems of Synchronization

Coverage: [OSCJ8] Chapter 6, §6.6 (pp. 255-264)

Classical probems of synchronization

- *The Bounded Buffer Problem* (also called the *The Producer-Consumer Problem*)
- *The Readers-Writers Problem*
- *The Dining Philosophers Problem*

These problems are used to test nearly every newly proposed synchronization scheme or primitive.

The Bounded Buffer Problem

Consider:

- a buffer which can store *n* items
- a producer process which creates the items (1 at a time)
- a consumer process which processes them (1 at a time)

A producer cannot produce unless there is an empty buffer slot to fill.

A consumer cannot consume unless there is at least one produced item.

```
Semaphore empty=N, full=0, mutex=1;

process producer {
   while (true) {
      empty.acquire();
      mutex.acquire();
      // produce
      mutex.release();
      full.release();
   }
}

process consumer {
   while (true) {
      full.acquire();
      mutex.acquire();
      // consume
      mutex.release();
      empty.release();
}
```

The semaphore `mutex` provides mutual exclusion for access to the buffer.

What are the semaphores `empty` and `full` used for?

The Readers-Writers Problem

A data item such as a file is shared among several processes.

Each process is classified as either a *reader* or *writer*.

Multiple readers may access the file simultaneously.

A writer must have exclusive access (i.e., cannot share with either a reader or another writer).

A solution gives priority to either readers or writers.

- *readers' priority*: no reader is kept waiting unless a writer has already obtained permission to access the database
- *writers' priority*: if a writer is waiting to access the database, no new readers can start reading

A solution to either version may cause starvation

- in the readers' priority version, writers may starve
- in the writers' priority version, readers may starve

A semaphore solution to the *readers' priority version* (without addressing starvation):

```
Semaphore mutex = 1;
Semaphore db = 1;
int readerCount = 0;

process writer {

   db.acquire();
   // write
   db.release();
}

process reader {

   // protecting readerCount
   mutex.acquire();
   ++readerCount;
   if (readerCount == 1)
      db.acquire();
   mutex.release();

   // read

   // protecting readerCount
   mutex.acquire();
   --readerCount;
   if (readerCount == 0)
      db.release();
   mutex.release();
}
```

`readerCount` is a <*cs*> over which we must maintain control and we use *mutex* to do so.

Self-study: address starvation in this solution, and then develop a *writers' priority* semaphore solution, and then address starvation in it.

The Dining Philosophers Problem

n philosophers sit around a table thinking and eating. When a philosopher thinks she does not interact with her colleagues. Periodically, a philosopher gets hungry and tries to pick up the chopstick on his left and on his right. A philosopher may only pick up one chopstick at a time and, obviously, cannot pick up a chopstick already in the hand of neighbor philosopher.

The dining philosophers problems is an example of a large class or concurrency control problems; it is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

A semaphore solution:

```
// represent each chopstick with a semaphore
Semaphore chopstick[] = new Semaphore[5]; // all = 1 initially

process philosopher_i {

   while (true) {
      // pick up left chopstick
      chopstick[i].acquire();

      // pick up right chopstick
      chopstick[(i+1) % 5].acquire();

      // eat

      // put down left chopstick
      chopstick[i].release();

      // put down right chopstick
      chopstick[(i+1) % 5].release();

      // think
   }
}
```

This solution guarantees no two neighboring philosophers eat simultaneously, but has the possibility of creating a deadlock (how so?).

Self-study: develop a deadlock-free semaphore solution.

Play the *what if* scenario game with all of these problems.

Classical probems of synchronization (concepts explored)

- The Bounded Buffer Problem (explores relationship preservation)
- The Readers-Writers Problem (explores *starvation*)
- The Dining Philosophers Problem (explores *deadlock*)

Uses of semaphores

Semaphores can be used for more than simply protecting a critical section.

- to protect acccess to a *critical section* (e.g., the database in the R/Ws problem)
- as a *mutex lock* (e.g., to protect the shared variables used in solving a probem such as `readerCount` in the R/Ws problem above)
- to protect a *relationship* (e.g., `empty` and `full` as used in the P/C problem)
- to support *atomicity* (e.g., you pickup either both chopsticks as the same time or neither; you cannot pickup just one)
- to enforce an order on the interleaving of the statements in the processes to by synchronized (e.g., `Thread2` printing before `Thread1` in `Order.java`)

Real-world Applications of these Classical Problems

- Producer-Consumer
  - web servering architecture: content or server thread and consumer or browser thread
  - streaming audio or video: network and display threads communicate through a shared buffer
- Readers-Writers
  - shared file on disk
  - database applications
- Dining Philosophers cooperating processes that need to share limited resources
  - a set of processes that require access to multiple resources at same time (e.g., file and printer)
  - Online course scheduling: user account, couse database, timetable of classes

---

References

[OSCJ8] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating Systems Concepts with Java*. John Wiley and Sons, Inc., Eighth edition, 2010.