CSC 1600: Chapter 6

Classical Synchronization Problems

The Producer-Consumer Problem
The Dining Philosophers Problem

Concurrency control problems

- Critical Section (mutual exclusion)
  - only one thread can be in its CS at a time
- Deadlock
  - each thread in a set of threads is holding a resource and waiting to acquire a resource held by another thread
- Starvation
  - a thread is repeatedly denied access to some resource protected by mutual exclusion, even though the resource periodically becomes available

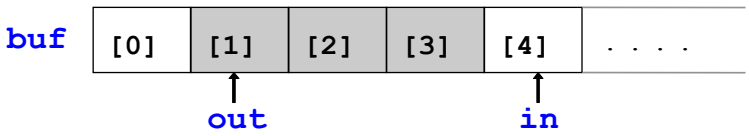
The Producer – Consumer problem

- Common synchronization pattern:
  - Producer waits for slot, inserts item in buffer, and "*signals*" consumer.
  - Consumer waits for item, removes it from buffer, and "signals" producer.

- Example: multimedia processing:
  - Producer creates MPEG video frames
  - Consumer renders the frames

Example: Circular buffer

- A *circular* buffer **`buf`** holds items that are produced and eventually consumed




```
#define MAX 10  /* maximum number of slots */

typedef struct {
  int buf[MAX]; /* shared var */
  int in;       /* buf[in%MAX] is the first empty slot */
  int out;      /* buf[out%MAX] is the first busy slot */
  sem_t full;   /* keep track of the number of full spots */
  sem_t empty;  /* keep track of the number of empty spots */
  sem_t mutex;  /* enforce mutual exclusion to shared data */
} sbuf_t;

sbuf_t shared;
```

General: Buffer that holds multiple items

Initially:
empty = MAX,
full = 0.

```
/* producer thread */
void *producer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* produce item */
    item = i;
    printf("produced %d\n",
            item);

    sem_wait(&shared.empty);
    sem_wait(&shared.mutex);
    /* write item to buf */
    shared.buf[shared.in] = item;
    shared.in = (shared.in+1)%MAX;
    sem_post(&shared.mutex);
    sem_post(&shared.full);
  }
  return NULL;
}
```

General: Buffer that holds multiple items

```
/* consumer thread */
void *consumer(void *arg) {
  int i, item;

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

    sem_wait(&shared.full);
    sem_wait(&shared.mutex);
    /* read item from buf */
    item = shared.buf[shared.out];
    shared.out = (shared.out+1)%MAX;
    sem_post(&shared.mutex);
    sem_post(&shared.empty);

    /* consume item */
    printf("consumed %d\n", item);
  }
  return NULL;
}
```

Dining Philosophers and the concept of Deadlock

Dining philosophers

- A problem that was invented to illustrate a different aspect of communication
- Our focus here is on the notion of sharing resources that *only one user at a time* can own
- Idea is to capture the concept of multiple processes competing for limited resources

Dining philosophers

- Five philosophers sit at a round table set with 5 plates and 5 forks, one between each plate
- Philosophers alternate between thinking and eating
  - Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done

A flawed conceptual solution

```
# define N  5

Philosopher P_i (0, 1, .. 4)
repeat forever {
    think();
    take_fork(i);
    take_fork((i+1)%N);
    eat(); /* yummy */
    put_fork(i);
    put_fork((i+1)%N);
}
```

How to implement?

- Represent each fork by a semaphore

```
Semaphore fork[5];
```


```
do
{
   DOWN(fork[i]);          /* pick up left fork*/
   DOWN(fork[(i+1)%5]);    /* pick up right fork*/
   ... Eat ...                  /* yummy ..*/
   UP(fork[(i+1)%5]);      /* put down left fork*/
   UP(fork[i]);            /* put down right fork*/
   ... Think ...
} while(1);
```

What could go wrong?

- Deadlock: All philosophers are "stuck", so that nobody can do anything at all
  - if they all philosophers pick up their "right" fork simultaneously!
- Starvation: Some philosopher is always hungry
- Deadlock ⇒ Starvation but not vice versa

Solutions

1. Allow only 4 philosophers to sit simultaneously

2. Asymmetric solution
   - Odd philosopher picks left fork followed by right
   - Even philosopher does vice versa

3. Pass a token

4. Allow philosopher to pick fork only if both available

Solution 1 example

- Use another semaphore **room** that limits the number of philosophers in the dining room to 4

```
Semaphore fork[5];
Semaphore room;

do
{
   DOWN(room);
   DOWN(fork[i]);          /* pick up left fork*/
   DOWN(fork[(i+1)%5]);    /* pick up right fork*/
   ... Eat ...                  /* yummy ..*/
   UP(fork[(i+1)%5]);      /* put down left fork*/
   UP(fork[i]);            /* put down right fork*/
   UP(room);
   ... Think ...
} while(1);
```

Summary

- Threads
  - Share global data
  - Access to shared data must be mutually exclusive
  - Synchronize access with semaphores

- Classical Synchronization Problems
  - The Producer-Consumer Problem
  - The Dining Philosophers Problem