Synchronization Problems

Producer Consumer Problem

There is a buffer of n slots and each slot can store 1 unit of data. Two processes operate on the buffer: producer and consumer. A producer tries to insert data into an empty slot of the buffer while a consumer tries remove data from a filled slot.

`

Producer

```
do{
   // empty is a counting semaphore
   // wait until buffer has at least 1 empty slot, empty > 0
   wait(empty);
   wait(mutex); // mutex is a binary semaphore
   // decrement empty and now acquire lock so consumer
   // does not access until producer completes critical section

   /* perform insert operation in a slot */
```

Consumer

```
do{
   // full is a counting semaphore
   // wait until buffer has at least 1 full slot, full > 0
   wait(full);
   wait(mutex); // mutex is a binary semaphore
   // decrement full and now acquire lock so consumer
   // does not access until producer completes critical section

   /* perform remove operation in a slot */
```

Dining Philosopher Problem

There are 5 philosopher sitting at a circular table. The table has 5 chop sticks. If a philosopher wants to eat, a chopstick must be grabbed from left and right to eat, and when philosopher wants to think, both chopsticks must be placed down.

```
* i = position of philosopher
while(true){
   wait(stick[i]);
   wait(stick[(i+1)%5]);
   /* eat */
   signal(stick[i]);
   signal(stick[(i+1)%5]);
   /* think */
}
```

Readers Writer Problem

2 types of processes: reader and writer. Readers can access the shared resource simultaneously but only 1 writer can access the shared resource at a time - no other writers or readers can access at this time.

Writer

```
// w is a semaphore
while(true){
   wait(w);
   /* perform write */
   signal(w);
}
```

Reader

```
while(true){
   /* acquire mutex lock because even though multiple readers can read at the same time, read_count should only be accessed by 1 process at a time */
   wait(m);
   read_count++;
   if(read_count == 1){
      // if the current process is the first reader then wait until writer is done using the shared resource
      // since writer and readers cannot access shared resource simultaneously
      wait(w);
   }
   signal(m); //release lock so many readers can perform read at the same time
   /* perform read */
   wait(m); //acquire mutex lock
   read_count--;
   if(read_count == 0){ /* if the current process is the last reader then release w semaphore so that a waiting writer can acquire the lock  */
      signal(w);
   }
   signal(m); //release lock
}
```

Forking a separate process using UNIX

```
#include <stdio.h>
void main(int argc, char *argv[]){
   int pid;
   pid = fork() // makes a copy of this entire process from this point on
   if (pid<0){
      fprintf(stderr, "Fork Failed");
      exit(-1);
   }else if (pid == 0){
      // Child process
      // execlp can only run compiled/binary files ie: hello.c Run: gcc -o hello hello.c
      // 2nd param is the 1st arg and list of args must be terminated by a null pointer
      execlp("./hello", "hello", NULL);
   }else{
      // Parent process and it will wait until child completes
      wait(NULL);
      printf("Child complete");
      exit(0);
   }
}
```

* If fork is called n times, there will be 2^n total processes