Classical IPC Problems

Now that we have learnt about semaphores and mutexes, let us write programs using these for the classical IPC problems. I give solution to one version(either using semaphores or using mutexes) per problem. The other versions for each problem is left as an exercise for the reader;).

I will be giving solutions to 4 different problems.

Note: I will be using the functions which I have provided in my previous post, for semaphores, which is in the file "mysem.c".

Producer-Consumer Problem:Semaphore Version

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<pthread.h>
#include "mysem.c"

#define MAX_BUF 10


int    *buffer;
int    head=0,tail=0,item_id=1;
int    count=0;
int    sem_ID; // 3 semaphores... { Mutex, Empty, Full}
int    fflag=0,eflag=0;

void producer()
{


printf("Producer started\n");

 while(1)
 {
  int v=0;
  sem_change(sem_ID, 0, -1);
  if(count==MAX_BUF)
  {
   printf("Producer waiting\n");
   fflag=1;
   sem_change(sem_ID, 0, 1);
   sem_change(sem_ID, 2, -1);
   sem_change(sem_ID, 0, -1);    //Waits until a free slot is available
  }

  count++;
  buffer[head]=item_id++;

  if(count==1 && eflag==1)
  {
  eflag=0;
  sem_change(sem_ID, 1, 1);    //To signal consumer thread that buffer is not empty
  }
  printf("produced item:%d \n", buffer[head]);
  head=(head+1)%MAX_BUF;
  sem_change(sem_ID, 0, 1);

  sleep(rand()%3);

 }
}

void consumer()
{
printf("consumer started\n");
 while(1)
 {
```

```
   int v=0;
   sem_change(sem_ID, 0, -1);
   if(count==0)
   {
    printf("Consumer Waiting\n");
    eflag=1;
    sem_change(sem_ID, 0, 1);
    sem_change(sem_ID, 1, -1);
    sem_change(sem_ID, 0, -1);
       //Waits until an item is produced
   }
   count--;

   printf("consumed item:%d \n", buffer[tail]);
   tail=(tail+1)%MAX_BUF;
   if(count==MAX_BUF-1 && fflag==1)
   {
    fflag=0;
    sem_change(sem_ID, 2, 1);                //To signal the producer thread that
    }
buffer is not full
   sem_change(sem_ID, 0, 1);
  sleep(rand()%4);
 }
}
int main()
{
 buffer=(int*)calloc(MAX_BUF,sizeof(int));
 pthread_t pro,con;
 int values[]={1,0,0};
 sem_ID=sem_init_diff_val(3, values); //initialize with 1, 0, 0

 pthread_create(&pro, NULL, (void*)&producer, NULL);
 pthread_create(&con, NULL, (void*)&consumer, NULL);

 pthread_join(pro,NULL);

 return 0;
}
```

Dining Philosophers Problem:Mutex Version

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#define PHILOSOPHERS 5

pthread_mutex_t   spoon[PHILOSOPHERS];

//Intializes the philosopher threads
void phil_init(int a, int* b, int* c)
{
 *b=(a>0) ? a-1 : PHILOSOPHERS;
 *c=a;
 printf("Philosopher %d started\n", a+1);
 return;
}


int  check_If_Spoons_Are_Available(int a, int b, int c)
{
 int  sum=0;
 if(a&1)
 {
  sum =  pthread_mutex_trylock(&spoon[c])==0 ? 0 : 10;
  sum += pthread_mutex_trylock(&spoon[b])==0 ? 0 :  1;
 }
 else
 {
  sum = pthread_mutex_trylock(&spoon[b])==0 ? 0 : 1;
  sum += pthread_mutex_trylock(&spoon[c])==0 ? 0 : 10;
 }
 return sum;
```

```
}

void Release_Spoons(int a, int b, int c)
{
 if(a&1)
 {
  pthread_mutex_unlock(&spoon[b]);
  pthread_mutex_unlock(&spoon[c]);
 }
 else
 {
  pthread_mutex_unlock(&spoon[c]);
  pthread_mutex_unlock(&spoon[b]);
 }
}

void wait_for_others_to_finish(int a, int b ,int c, int d)
{
 switch(a)
 {
  case 1:
   printf("Philosopher %d waiting since right spoon is unavailable\n",b+1);
   pthread_mutex_lock(&spoon[c]);
   break;
   case 10:
    printf("Philosopher %d waiting since left spoon is unavailable\n", b+1);
    pthread_mutex_lock(&spoon[d]);
    break;
    case 11:
     printf("Philosopher %d waiting since both spoons are unavailable\n", b+1);
     if(a&1)
     {
      pthread_mutex_lock(&spoon[d]);
      pthread_mutex_lock(&spoon[c]);
     }
     else
     {
      pthread_mutex_lock(&spoon[d]);
      pthread_mutex_lock(&spoon[c]);
     }
     break;
 }
 return;
}

void Eat(int a)
{
  printf("philosopher %d eating\n", a+1);
  sleep(rand()%5);
  printf("philosopher %d finished eating\n", a+1);
}
void philo(void * arg)
{

 int  back;
 int  front;
 int  tmp;
 int  id=*((int*)arg);
 phil_init(id, &back, &front);


 while(1)
 {
  printf("philosopher %d thinking\n", id+1);
  sleep(rand()%6);

  if((tmp=check_If_Spoons_Are_Available(id, back, front))!=0)
   wait_for_others_to_finish(tmp, id, back, front);

   Eat(id);

  Release_Spoons(*((int*)arg), back, front);
 }
}
```

```
int main(int argc, char* argv[])
{
 pthread_t  S[PHILOSOPHERS];
 int   *g;

 for(int i=0; i<PHILOSOPHERS; i++)
 pthread_mutex_init(&spoon[i], NULL);

  g=(int*)malloc(PHILOSOPHERS*sizeof(int));
 for(int i=0; i<PHILOSOPHERS; i++)
 {
   g[i]=i;
  pthread_create(&S[i], NULL, (void*)&philo,(void*)&g[i]);
 }
 pthread_join(S[0], NULL);
exit(0);
}
```

Sleeping Barber Problem:Semaphore Version

```
#include<stdlib.h>
#include<stdio.h>
#include<pthread.h>
#include "mysem.c"
#define MAX_C 5
int sem_ID;         //----> creates a set of 5 semaphores. { mutex, cond_empty, counting semaphore, waiting semaphore, barber sema
int customers_count=0;
int eflag=fflag=0;
void barber()
{
 printf("barber started\n");
 while(1)
 {

  sem_change(sem_ID, 0, -1);

  if(customers_count==0)
  {
   printf("barber sleeping\n");
   eflag=1;
   sem_change(sem_ID, 0, 1);
   sem_change(sem_ID, 1, -1);
   sem_change(sem_ID, 0, -1);
  }

  customers_count--;

  sem_change(sem_ID, 0, 1);
  sem_change(sem_ID, 3, 1);
 // printf("tail:%d\n", tail);
 // pthread_mutex_lock(&B);

  sem_change(sem_ID, 4, -1);
 // pthread_mutex_unlock(&B);

  }
}
void customer(void *arg)
{

  //printf("C\n");
 sem_change(sem_ID, 0, -1);
 if(customers_count==MAX_C)       // If all seats are occupied exit the thread
  {
   int *ptr=(int*)arg;
   *ptr=0;
   printf("No place for customer %d so leaving\n", pthread_self());
   sem_change(sem_ID, 0, 1);
   return;
  }
```

```
  customers_count++;

  if(customers_count==1 && eflag==1)
  {
  sem_change(sem_ID, 1, 1);
  eflag=0;
  }
  sem_change(sem_ID, 0, 1);
  printf("Customer %d got a place\n", pthread_self());
  sem_change(sem_ID, 3, -1);
  printf("Cutting for %d customer\n", pthread_self());
  sleep(rand()%5+4);
  sem_change(sem_ID, 4, 1);
 // printf("head:%d\n", head);
 // pthread_mutex_lock(&B);

 // pthread_mutex_unlock(&B);


  int *ptr=(int*)arg;
  *ptr=0;
}
int main(int argc, char* argv[])
{
pthread_t  barber_thread;
int   live_threads[MAX_C+2];     // 0 = no thread is created with this index,

          // 1=there is a live thread with this index

pthread_t  customer_thread[MAX_C +2];


for(int i=0; i<MAX_C+2; i++)
 live_threads[i]=0;     // initially all are dead state


int array[]={1, 0, MAX_C, 0, 0};     // Initial values of different semaphores
sem_ID=sem_init_diff_val(5,array);

pthread_create(&barber_thread, NULL, (void*)&barber, NULL);

sleep(2);
//Continuous thread generator....
while(1)
{
 for(int i=0; i<MAX_C+2; i++)
 {
  if(live_threads[i]==0)
  {
   live_threads[i]=1;
   pthread_create(&customer_thread[i], NULL, (void*)&customer, (void*)&live_threads[i]);
   sleep(rand()%4);
  }

 }
}



exit(0);
}
```

Readers Writers Problem:Mutex Version

Note: The execution depends on the selection algorithm of the mutex.

```
#include<stdio.h>
#include<pthread.h>
pthread_mutex_t no_wait, no_acc, counter;
int no_of_readers=0;
```

```
void reader(void *arg)
{

 int id=*((int*)arg);
 printf("reader %d started\n", id);
 while(1)
 {
  sleep(rand()%4);
 check_and_wait(id);
 read(id);
}
}


void writer(void* arg)
{
 int id=*((int*)arg);
 printf("writer %d started\n", id);
 while(1)
{
 sleep(rand()%5);
 check_and_wait_if_busy(id);
 write(id);
}
}


/*sub functions for reader and writer threads*/
void check_and_wait_if_busy(int id)
{
  if(pthread_mutex_trylock(&no_wait)!=0){
  printf("Writer %d Waiting\n", id);
  pthread_mutex_lock(&no_wait);
 }
}
void check_and_wait(int id)
{
 if(pthread_mutex_trylock(&no_wait)!=0){
  printf("Reader %d Waiting\n", id);
  pthread_mutex_lock(&no_wait);
 }
}
void read(int id)
{
 pthread_mutex_lock(&counter);
 no_of_readers++;
 pthread_mutex_unlock(&counter);
 if(no_of_readers==1)
  pthread_mutex_lock(&no_acc);
 pthread_mutex_unlock(&no_wait);
 printf("reader %d reading...\n", id);
 sleep(rand()%5);
 printf("reader %d finished reading\n", id);

 pthread_mutex_lock(&counter);
 no_of_readers--;
 pthread_mutex_unlock(&counter);
 if(no_of_readers==0)
  pthread_mutex_unlock(&no_acc);

}

void write(int id)
{
 pthread_mutex_lock(&no_acc);
 pthread_mutex_unlock(&no_wait);
 printf("Writer %d writing...\n", id);
 sleep(rand()%4+2);
 printf("Writer %d finished writing\n", id);
 pthread_mutex_unlock(&no_acc);
}

/*********************************************/
//MAIN
```

```
int main(int argc, char* argv[])
{
 pthread_t R[5],W[5];
 int ids[5];
 for(int i=0; i<5; i++)
 {
  ids[i]=i+1;
  pthread_create(&R[i], NULL, (void*)&reader, (void*)&ids[i]);
  pthread_create(&W[i], NULL, (void*)&writer, (void*)&ids[i]);
 }
 pthread_join(R[0], NULL);
 exit(0);
}
```