1. Producer-Consumer

Consider the wrong solution to the PC problem listed in OSTEP Chapter 30 “Figure 30.8: Producer/Consumer: Single CV And If Statement”. The example in figure 30.9 showed an error trace where one of the consumers read from an empty buffer. Is it possible to construct a scenario where a producer overwrites a full buffer?

Tp1 produces an item : count ==1

Tp2 produces an item: wait <Tp2>

Tp1 produces an item: wait<Tp2,Tp1>

Tc1 consumes: count==0, wait <Tp1>

Tp2 produces an item , signals count =1 <>

Tp1 produces an item //owerwrite //ooops

Tp1	state	Tp2	state	Tc1	state	Count	comment
p1	run		rdy		rdy	0	Tp1 starts producing an item
P2							
P4						1	Write to buf
p5							
p6	rdy						
		P1	run				Tp2 starts producing an item
		P2					Buf already full
		P3	block				Cond Wait queue <Tp2>
p1	Run						Tp1 starts produces an item
P2	Run						Buf already full
p3	block						Cond Wait queue <Tp2,Tp1>
				c1	run		Tc1 consumes an item
				c2			
				c4		0	Item consumed
			rdy	c5			Cond Wait queue <Tp1>
				c6	Rdy		
		p4	run			1	Write to buf (continue producing)
	rdy	p5					Cond Wait queue <>
		p6	rdy				
P4	run						Owerwrite, OOPS!

```
bnielsen@BNKONTOR: /mnt/c/Users/bniel/BNSoftware/Kode
joined producer 1
0,2
bnielsen@BNKONTOR:/mnt/c/Users/bniel/BNSoftware/Kode$ ./a.out 2 100 4 2
0 produced 0
0 produced 1
consumer 0 got 0
consumer 0 got 1
1 produced 0
1 produced 1
consumer 2 got 0
consumer 2 got 1
0 produced 2
0 produced 3
consumer 2 got 2
consumer 2 got 3
1 produced 2
1 produced 3
consumer 3 got 2
consumer 3 got 3
1 produced 4
1 produced 5
consumer 3 got 4
consumer 3 got 5
1 produced 6
1 produced 7
consumer 0 got 6
consumer 0 got 7
1 produced 8
1 produced 9
consumer 1 got 8
```

Sample run with 4 consumers and two producers.

The only “tricky” part of generalizing the pc.c program to multiple producers is to ensure correct termination. In the original pc.c this is done by letting the (single) producer write “end of work” marker (-1 value) into the buffer. With multiple M producers they should in total write N (number of producers) termination marks, in the code below it is simply done by the last producer write the required termination markers (one per consumer).

```c
#include <assert.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "common.h"
#include "common_threads.h"
```

```c
int max;
int loops;
int *buffer;

int use_ptr  = 0;
int fill_ptr = 0;
int num_full = 0;



pthread_cond_t empty  = PTHREAD_COND_INITIALIZER;
pthread_cond_t fill   = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m     = PTHREAD_MUTEX_INITIALIZER;

int consumers = 1;
int producers =1;
int verbose = 1;

int producersDone=0;

void do_fill(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % max;
    num_full++;
}

int do_get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % max;
    num_full--;
    return tmp;
}

void *producer(void *arg) {
    int i;
    long id=(long) arg;
    for (i = 0; i < loops; i++) {
    Mutex_lock(&m);            // p1
    while (num_full == max)     // p2
        Cond_wait(&empty, &m); // p3
    printf("%ld produced %d\n",id,i);
    do_fill(i);                // p4
    Cond_signal(&fill);        // p5
    Mutex_unlock(&m);          // p6
    }
```

```
    // end case: put an end-of-production marker (-1)
    // into shared buffer, one per consumer

    Mutex_unlock(&m);
    producersDone++;
    if(producersDone==producers){  //the last producer terminates consumers
      for (i = 0; i < consumers; i++) {
        while (num_full == max)
            Cond_wait(&empty, &m);
        printf("%ld TERMINATING %d\n",id,i);
        do_fill(-1);
        Cond_signal(&fill);
      }
    }
    Mutex_unlock(&m);

    return NULL;
}

void *consumer(void *arg) {
    int tmp = 0;
    long id=(long) arg;
    // consumer: keep pulling data out of shared buffer
    // until you receive a -1 (end-of-production marker)
    while (tmp != -1) {
    Mutex_lock(&m);           // c1
    while (num_full == 0)      // c2
        Cond_wait(&fill, &m); // c3
    tmp = do_get();           // c4
    printf("consumer %ld got %d\n",id,tmp);
    Cond_signal(&empty);      // c5
    Mutex_unlock(&m);         // c6
    }
    printf("%ld stopping: %d\n",id,tmp);
    return NULL;
}

int
main(int argc, char *argv[])
{
    if (argc != 5) {
    fprintf(stderr, "usage: %s <buffersize> <loops> <consumers> <producers>\n", a
rgv[0]);
    exit(1);
```

```
    }
    max = atoi(argv[1]);
    loops = atoi(argv[2]);
    consumers = atoi(argv[3]);
    producers= atoi(argv[4]);


    buffer = (int *) malloc(max * sizeof(int));
    assert(buffer != NULL);

    int i;
    for (i = 0; i < max; i++) {
    buffer[i] = 0;
    }

    pthread_t pid[producers], cid[consumers];
    for (i = 0; i < producers; i++) {
        Pthread_create(&pid[i], NULL, producer, (void *) (long long int) i);
    }
    //Pthread_create(&pid, NULL, producer, NULL);

    for (i = 0; i < consumers; i++) {
    Pthread_create(&cid[i], NULL, consumer, (void *) (long long int) i);
    }

    for (i = 0; i < consumers; i++) {
      Pthread_join(cid[i], NULL);
       printf("joined consumer %d\n",i);
    }
     for (i = 0; i < producers; i++) {
      Pthread_join(pid[i], NULL);
       printf("joined producer %d\n",i);
    }
    printf("%d,%d\n",num_full,producersDone);
    return 0;
}
```

2. Bank Account with Condition Variables:

Consider the transfer among bank-accounts from last week. Change the transfer function to block the calling thread untill insufficient money is available at the from account instead of returning true/false. Use a single “bank” mutex lock and 1 (or more) condition variables. You may assume that amount is non-negative.

```
transfer(int * from,int * to, int amount){
    assert(amount>=0);
    if(*from<amount)
      return false
    *from-= amount;
    *to+=amount;
    return true;
}
int kontoA=10; int kontoB=20;
Thread mand(){transfer(&kontoA,&kontoB,7));
Thread kone(){transfer(&kontoA,&kontoB,4));
Thread son (){transfer(&kontoB,&kontoA,10));
```

```
pthread_mutex_t bank = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t low = PTHREAD_COND_INITIALIZER;


transfer(int * from,int * to, int amount){
 assert(amount>=0);
 Pthread_mutex_lock(&bank);
    while(*from<amount)
      Pthread_cond_wait(&low, & bank);
    *from-= amount;
    *to+=amount;
    Pthread_cond_broadcast(low); //Coverer all conditions!!
 Pthread_mutex_unlock(&bank);

}
//More refined with two conditions (assume To!=from)

Void transfer(int * from,int * to, int amount, lock_t * bank, cond_t * from_low,
cond_t * to_low){
 assert(amount>=0);
 Pthread_mutex_lock(&bank);
    while(*from<amount)
      Pthread_cond_wait(from_low, bank);
    *from-= amount;
    *to+=amount;
    Pthread_cond_broadcast(to_low,bank); //Coverer all conditions!!
 Pthread_mutex_unlock(&bank);

}
```

3. The rendezvous problem

The rendezvous problem is as follows: you have two threads, each of which are about to enter the rendezvous point in the code. Neither should exit this part of the code before the other enters it. Consider using two semaphores for this task. Sketch how the two threads can make the rendezvous.

```
sem_t s1; sem_t s2=0;
sem_init(&s1,0,0); sem_init(&s2,0,0)
Thread 1 calls void rendezvous1() {
   sem_post (&s2);
   sem_wait (&s1);
}
Thread 2 calls void rendezvous2(){
  sem_post (&s1);
  sem_wait (&s2);
}
```

Eller

```
sem_t s1; sem_t s2=0;
sem_init(&s1,0,0); sem_init(&s2,0,0)
Thread 1 calls void rendezvous1() {
   sem_wait (&s1);
   sem_post (&s2);
}
Thread 2 calls void rendezvous2(){
  sem_post (&s1);
  sem_wait (&s2);
}
```

4. Can the following program deadlock? Why or Why not?

```
sem_t a; sem_t b; sem_t c;

sem_init(&a,0,1); sem_init(&b,0,1); sem_init(&c,0,1);
```

Thread 1	Thread 2

sem_wait(&a);
sem_wait(&b);
sem_post(&b);
sem_wait(&c);
sem_post(&c);
sem_post(&a);
	sem_wait(&c);
	sem_wait(&b);
	sem_post(&b);
	sem_post(&c);

No deadlock, because 1 may (hold a then wait b), or (hold a then wait c) whereas thread 2 (hold c then wait b). The lock both parties need will be released by the other party.

5. Consider the following program that may deadlock

1) For each thread, list the pairs of mutexes it holds simultaneously
2) If a < b < c, which threads violate the locking order rule for deadlock prevention
3) Rewrite each thread to guarantees absence of deadlock

```
sem_t a; sem_t b; sem_t c;

sem_init(&a,0,1); sem_init(&b,0,1); sem_init(&c,0,1);
```

Thread 1	Thread 2	Thread 3

sem_wait(&a);
sem_wait(&b);
sem_post(&b);
sem_wait(&c);
sem_post(&c);
sem_post(&a);
	sem_wait(&c);
	sem_wait(&b);
	sem_post(&b);
	sem_post(&c);
	sem_wait(&a);
	sem_post(&a);
		sem_wait(&c);
		sem_post(&c);
		sem_wait(&b);
		sem_wait(&a);
		sem_post(&a);
		sem_post(&b);

1)

Thread 1: (a,b); (a,c)

Thread 2: (c,b),

Thread 3: (b,a)

2) Thread 2, 3 violates order.

3) If we use order a>b>c then 2 threads need to be rewritten. If we use locking order c > b > a, only thread 1 needs rewrite:

Thread 1	Thread 2	Thread 3

sem_wait(&c);
sem_wait(&b);
sem_wait(&a);
sem_post(&b);
sem_post(&c);
sem_post(&a);
	sem_wait(&c);
	sem_wait(&b);
	sem_post(&b);
	sem_post(&c);
	sem_wait(&a);
	sem_post(&a);
		sem_wait(&c);
		sem_post(&c);
		sem_wait(&b);
		sem_wait(&a);
		sem_post(&a);
		sem_post(&b);

Challenge 13

The Therac 25 therapeutic radiation machine killed several people and injured even more in the 1980'es. When the case was investigated by independent software safety researcher Nancy Leveson, she found several critical software flaws in the system, including several race-conditions.

The machine supports various kinds of treatment and treatment parameters, but we will focus on two:

- Treatment A where a high dose is given to a big area (wide-beam)
- Treatment B where a lower dose is to be given to a focused area (narrow beam)

The radiation beam width is controlled by bending magnets whose position is controlled by the machine.

Deaths or injuries occurred because the machine gave a high dose to a highly focused area.

The code below attempts to present a code-sketch that mimics how one of these race-conditions could emerge. Can you spot it?

There is a secondary problem related to detection of errors in the machine. Can you spot it?

The real system was programmed in assembly, no threads package, and switching between the "threads" in interrupts handlers". The race-condition we seek here is not cause by simple preemption in a test-set update, so you may disregard those.

DISCLAIMER: The code is a simplified presentation of the problem (but maintains the core of one of the issues) for education purposes.

```
char MEOS[2]; //"Mode Energy Offset"
const int PWR=0; //indexes into MEOS
const int FOCUS=1;

//MEOS[PWR] indicates powerlevel (low / high )
//MEOS[FOCUS] indicated beam focus (narrow/high)

unsigned char error=0;
unsigned char dataEntry=0;
unsigned char bendingDone=0;
unsigned char editing=0;

void showMenu(){
    printf("1:edit, 2: Start\n");
};
```

```
thread KeyHandlerInput(){
   while(1){
      showMenu();
      choice=readline();
      if(choice==EDIT)  {
       editing=1;
       //operator chooses between two kinds of treatment
       if(treatmentA) { MEOS[PWR]=High; MEOS[FOCUS]=WideBeam};
       if(treatmentB) { MEOS[PWR]=Low; MEOS[FOCUS]= NarrowBeam;}
       editing=0;
       }
       else if( choice==START) {
         dataEntryDone=1;
       }
    }
}

Thread positionBendingMagnets() {
  while (!dataEntryDone) ; //just wait
  //dataEntryDone
  char focus= MEOS[FOCUS]);
  forEach(magnet ) {
      bend(magnet,focus); //takes 8 seconds
      if(editing) stop();
  }
  bendingDone=1;
}

Thread Radiate(){
  while(bendingDone==0|| dataEntryDone==0) ; //just wait
  if(other_parts_of_machine_OK) {
    // Bending Done && DataENtry==Done
    RADIATE(MEOS[PWR]);
  }  else {
    error++;
  }
  dataEntry=0;
  bendingDone=0;
}
```

Solution:

Operator picks treatment B, and starts, but realizes that he/she should have given treatment A, and changes this. A quick operator can do this within 8 seconds, hence the magnets will be positioned according to the narrow position, but the Radiate thread will deliver the dosis from Treatment A (High)

The problem with error is that it is intended to be a Boolean that indicates that an error has occurred, but it is incremented instead of assigned by 1, so it may overflow, not showing error after an overflow.