Assignment 3: Condition Variables

2INC0 - Operating systems

From

Full Name	Student ID
Daniel Tyukov	1819283
Nitin Singhal	1725963
Ben Lentschig	1824805

Eindhoven, January 23, 2025

1 | Basic solution

1.1 | Problem Introduction

The traditional bounded buffer problem is introduced and solved in the scope of a real life warehouse scenario in which multiple producers fetch various items (in a random order) for a single consumer. These items are put in the buffer from which consumers have to take their respective item. The main issue is that one consumer/producer can only handle one item at a time giving possibilities of buffer handling issues.

1.2 | Basic Solution Implementation

To stop any issues from occurring with the buffer, three mutexes are used. They are *cv*, *in_mutex* and *out_mutex*. The condition variable is used to ensure that items are placed to the buffer in numerical order (a producer thread can only write to it if it has the exact correct job). The in mutex only allows 1 producer to write to the buffer simultaneously. The out mutex is condition synchronized to prevent the consumer from reading from buffer when it is empty.

```
static void* producer(){
    while(true){
        get_current_job();
        sleep(time);
        lock(&in_mutex);
        while(job!=condition_var){
            cond_wait(&cv,&in_mutex);
        }
        buffer[in] = curr_job;
        update_variables(in,count,cv);
        cond_signal(&out_mutex);
        unlock(&in_mutex);
    }
}
```

```
static void* consumer(){
    ITEM curr_job = 0;
    while (curr_job<NROF_ITEMS){
        lock(&in_mutex);
        while(count==0){
            cond_wait (&out_mutex, &in_mutex);
        }
        curr_job = buffer[out];
        if(curr_job==NROF_ITEMS) break;
        update_variables(out,count,cv);
        cond_broadcast(&cv);
        unlock(&in_mutex);
        sleep (time);
}
```

The condition variable for the producer is implemented such that the value of the condition variable is the same as the last (job id + 1) put into the buffer. This way, the producers are forced to put jobs into the buffer in numerical order. Furthermore the value of the condition variable is set to the negative of the latest job id if the buffer is full to prevent producers from putting in jobs at that point. Assuming that at least one of the producer threads has a correct next job, the only thing stopping them from writing to the buffer is if it is full. Only the consumer can empty the buffer so only it is allowed to signal on the condition variable.

2 | Advanced solution

The advanced solution implements the same buffer management, however, this version avoids broadcast signaling. For the advanced solution, each producer holds exactly one item at a time (which it got from *get_next_item*()) and stores that item in *producer_items*[id]. If *producer_done*[id] is true, that producer has no more items to fetch.

```
static ITEM buffer[BUFFER_SIZE]; //global buffer
static int  in   = 0; // next free position in circular buffer
static int  out  = 0; // position of next item to consume
static int  count= 0; // how many items are in the buffer right now
static int  next_to_produce = 0; //item number that must be produced *next* in ascending
     order
 //Each producer can only produce its item when item == next_to_produce.
static int  items_consumed = 0; //num of items consumed
 producer(){
while(){
get_next_item();
if (item == NROF_ITEMS){
    no more real items => stop
}else{
sleep(time);
}
wait('my item' == next_to_produce);
wait(buffer!=full)
produce(buffer);
update(next_to_produce);
if(buffer==empty){
signal_buff_empty();
}
signal_next_prod();
}
```

```
consumer(){
while(true){
wait(buff==empty){
consume_next_item(out);
print(consumed_item);
if(buff==full){
signal_producer();
    }
if(num_items_consumed == NROF_ITEMS){
break;
            }
        }
    }
}
```

```
get_next_item(){
 return ( values of 0..NROF_ITEMS-1 in *arbitrary order*);
 else
 Return (value == NROF_ITEMS) //meaning all items have been handed out already.
}
```

Now that every producer has only a single item and a unique ID, we keep track of which item needs to be produced or consumed. We only wake up the producer who has to produce next. The consumer, again, only removes items if there are any in the buffer; otherwise, it waits until items are placed within. This is all done to ensure that we never have any broadcast signaling.

3 | Comparison of calls

By adding the variables, *signal_count* and *broadcast_count* to the code, the number of signals and broadcasts can be counted, then output on stderr once the function finishes. The outputs for the basic and advanced solution are shown respectively as:

```
Signals: 2001
Broadcasts: 2000
```

```
SSignals: 3509
Broadcasts: 0
```

Since a signal involves sending a message to 1 thread, while with 10 producers, a broadcast sends a message to 10 threads, the basic solution is sending a total of 22001 messages to threads, while the advanced solution is sending 3509 messages to threads. Running the advanced solution 100 times leads to the average number of signals as 3546.57. This can be verified by running the following bash script (after modifying the 2 print statements in prodcons.c to only print number of signals and nothing else)

```bash
#!/bin/bash

sum=0
num=0
loops=100

for ((i=0; i<=$loops; i++)); do
  num=$(./prodcons)
  num=$(python -c "print($num/$loops)")
  sum=$(python -c "print($sum+$num)")
done

python -c "print($sum)"
```

Of the 3546.57 signals made on average, at least 2000 are from the consumer to producers (1 per job), which is the absolute minimum number of signals. The remainder are from the producers signaling back to the consumer that jobs were put into the buffer. In total, the advanced solution individually signals approximately 16.12% as many threads as the basic one.