

*This homework covers OSTEP chapters 30 and 31. It is due in class Friday, April 17. Hand in a hardcopy of your answers for the chapter 30 problems in class. Hand in an electronic copy of your programs for the chapter 31 problems by copying the whole `threads-sema` folder into your handin folder (`/classes/cs331/username`).*

*While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself and **you must write up your own solutions in your own words**. You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.*

## Preliminaries

- Copy the directory `threads-cv` (and its contents) from `/classes/cs331` to your `~/cs331` directory. (You won't be writing any code so this doesn't need to go into your `workspace` directory.)
- Copy the directory `threads-sema` (and its contents) from `/classes/cs331` to your `~/cs331/workspace` directory. (You will be writing some code.)

Both directories contain C programs, a `Makefile` (just run `make` in the directory to compile everything), and `READMEs` with instructions. You can also find the text of the `READMEs` in the appendix sections [A](#) and [B](#) at the end of this document.

## Exercises

Use Linux for all of the exercises.

Chapter 30 homework problems (at the very end of chapter 30) —

1. Do #1–2. `main-two-cvs-while.c` is the final (working) version of the full producer/consumer implementation discussed in class. Include `-v` when you run the program to see the output. Look at the `README` to understand the output produced and to see how to specify the numbers of producers and consumers, the buffer size, etc. What does the sleep string `-C 0,0,0,0,0,0,1` mean? The goal here is to understand how to run the program and what you are looking at for the output — when you answer the questions posed in the problem (about larger buffers and a different consumer sleep string), also explain what's going on and why.

2. Do #4–5. Think about the expected timings before running the programs. Only the 1-second pauses take any appreciable time, so the question is really asking you to think about when and how often consumer threads pause. Check your answer by running the command lines specified. Were you correct? For your written answer, compare the times for the two runs and explain what is going on — why are they the same/different?
3. Do #6–7. Repeat what you did for the previous problem for this scenario.
4. Do #8–11. Give the sleep strings and explain your answers — what problems do the sleep string(s) you give cause and why?

Chapter 31 homework problems (at the very end of chapter 31) —

A `Makefile` has been provided to simplify compilation — run `make` to compile everything.

5. Do #1. “As described in the text” means the code is in the chapter — this problem really is just typing in code from the book (and adding the `sleep(1)` call).
6. Do #2–3. In both cases, include comment(s) in your code explaining your solutions — how does each ensure the desired behavior?
7. Do #4. “As described in the text” means the code is in the chapter — this problem really is just typing in code from the book (and adding the `sleep()` call).
8. Do #5. The book gives a hint about a strategy. Include comment(s) in your code explaining your solution — how does it ensure the desired behavior?

## A threads-cv

This homework lets you explore some real code that uses locks and condition variables to implement various forms of the producer/consumer queue discussed in the chapter. You'll look at the real code, run it in various configurations, and use it to learn about what works and what doesn't, as well as other intricacies.

The different versions of the code correspond to different ways to "solve" the producer/consumer problem. Most are incorrect; one is correct. Read the chapter to learn more about what the producer/consumer problem is, and what the code generally does.

The first step is to download the code and type `make` to build all the variants. You should see four: - `main-one-cv-while.c`: The producer/consumer problem solved with a single condition variable. - `main-two-cvs-if.c`: Same but with two condition variables and using an `if` to check whether to sleep. - `main-two-cvs-while.c`: Same but with two condition variables and `while` to check whether to sleep. This is the correct version. - `main-two-cvs-while-extra-unlock.c`: Same but releasing the lock and then reacquiring it around the `fill` and `get` routines.

It's also useful to look at `pc-header.h` which contains common code for all of these different main programs, and the `Makefile` so as to build the code properly.

Each program takes the following flags:

```
-l <number of items each producer produces>
-m <size of the shared producer/consumer buffer>
-p <number of producers>
-c <number of consumers>
-P <sleep string: how producer should sleep at various points>
-C <sleep string: how consumer should sleep at various points>
-v [verbose flag: trace what is happening and print it]
-t [timing flag: time entire execution and print total time]
```

The first four arguments are relatively self-explanatory: `-l` specifies how many times each producer should loop (and thus how many data items each producer produces), `-m` controls the size of the shared buffer (greater than or equal to one), and `-p` and `-c` set how many producers and consumers there are, respectively.

What is more interesting are the two sleep strings, one for producers, and one for consumers. These flags allow you to make each thread sleep at certain points in an execution and thus switch to other threads; doing so allows you to play with each solution and perhaps pinpoint specific problems or study other facets of the producer/consumer problem.

The string is specified as follows. If there are three producers, for example, the string should specify sleep times for each producer separately, with a colon separator. The sleep string for these three producers would look something like this:

```
sleep_string_for_p0:sleep_string_for_p1:sleep_string_for_p2
```

Each sleep string, in turn, is a comma-separated list, which is used to decide how much to sleep at each sleep point in the code. To understand this per-producer or per-consumer sleep string better, let's look at the code in `main-two-cvs-while.c`, specifically at the producer code. In this code snippet, a producer thread loops for a while, putting elements into the shared buffer via calls to `do_fill()`. Around this filling routine are some waiting and signaling code to ensure the buffer is not full when the producer tries to fill it. See the chapter for more details.

```
void *producer(void *arg) {
    int id = (int) arg;
    int i;
    for (i = 0; i < loops; i++) {
        p0;
        Mutex_lock(&m);          p1;
        while (num_full == max) { p2;
            Cond_wait(&empty, &m); p3;
        }
        do_fill(i);              p4;
        Cond_signal(&fill);      p5;
        Mutex_unlock(&m);        p6;
    }
    return NULL;
}
```

As you can see from the code, there are a number of points labeled `p0`, `p1`, etc. These points are where the code can be put to sleep. The consumer code (not shown here) has similar wait points (`c0`, etc.).

Specifying a sleep string for a producer is easy: just identify how long the producer should sleep at each point `p0`, `p1`, ..., `p6`. For example, the string `1,0,0,0,0,0,0` specifies that the producer should sleep for 1 second at marker `p0` (before grabbing the lock), and then not at all each time through the loop.

Now let's show the output of running one of these programs. To begin, let's assume that the user runs with just one producer and one consumer. Let's not add any sleeping at all (this is the default behavior). The buffer size, in this example, is set to 2 (`-m 2`).

First, let's build the code:

```
prompt> make
gcc -o main-two-cvs-while main-two-cvs-while.c -Wall -pthread
gcc -o main-two-cvs-if main-two-cvs-if.c -Wall -pthread
gcc -o main-one-cv-while main-one-cv-while.c -Wall -pthread
gcc -o main-two-cvs-while-extra-unlock main-two-cvs-while-extra-unlock.c -Wall -pthread
```

Now we can run something:

```
prompt> ./main-two-cvs-while -l 3 -m 2 -p 1 -c 1 -v
```

In this case, if you trace the code (with the verbose flag, `-v`), you will get the following output (or something like it) on the screen:

```

NF          P0 C0
0 [*--- --- ] p0
0 [*--- --- ]  c0
0 [*--- --- ] p1
1 [u  0 f--- ] p4
1 [u  0 f--- ] p5
1 [u  0 f--- ] p6
1 [u  0 f--- ] p0
1 [u  0 f--- ]  c1
0 [ --- *--- ]  c4
0 [ --- *--- ]  c5
0 [ --- *--- ]  c6
0 [ --- *--- ]  c0
0 [ --- *--- ] p1
1 [f--- u  1 ] p4
1 [f--- u  1 ] p5
1 [f--- u  1 ] p6
1 [f--- u  1 ] p0
1 [f--- u  1 ]  c1
0 [*--- --- ]  c4
0 [*--- --- ]  c5
0 [*--- --- ]  c6
0 [*--- --- ]  c0
0 [*--- --- ] p1
1 [u  2 f--- ] p4
1 [u  2 f--- ] p5
1 [u  2 f--- ] p6
1 [u  2 f--- ]  c1
0 [ --- *--- ]  c4
0 [ --- *--- ]  c5
0 [ --- *--- ]  c6
1 [f--- uEOS ] [main: added end-of-stream marker]
1 [f--- uEOS ]  c0
1 [f--- uEOS ]  c1
0 [*--- --- ]  c4
0 [*--- --- ]  c5
0 [*--- --- ]  c6

```

Consumer consumption:

C0 -> 3

Before describing what is happening in this simple example, let's understand the depiction of the shared buffer in the output, as is shown on the left. At first it is empty (an empty slot indicated by ---, and the two empty slots shown as [`*---` `---`]); the output also shows the number of entries in the buffer (`num_full`), which starts at 0. Then, after P0 puts a value in it, its state changes to [`u 0 f---`] (and `num_full` changes to 1). You might also notice a few additional markers here; the `u` marker shows where the `use_ptr` is (this is where the next consumer to consume a value will get something from); similarly, the `f` marker shows where the `fill_ptr` is (this is where the next producer will produce a value). When you see the `*` marker, it just means the `use_ptr` and `fill_ptr` are pointing to the same slot.

Along the right you can see the trace of which step each producer and consumer is about to execute. For example, the producer grabs the lock (`p1`), and then, because the buffer has an empty slot, produces a value into it (`p4`). It then continues until the point where it releases the lock (`p6`) and then tries to reacquire it. In this example, the consumer acquires the lock instead and consumes the value (`c1`, `c4`). Study the trace further to understand how the producer/consumer solution works with a single producer and consumer.

Now let's add some pauses to change the behavior of the trace. In this case, let's say we want to make the producer sleep so that the consumer can run first. We can accomplish this as follows:

```
prompt> ./main-two-cvs-while -l 1 -m 2 -p 1 -c 1 -P 1,0,0,0,0,0,0 -C 0 -v
```

The results:

```
NF          P0 C0
0 [*--- ---] p0
0 [*--- ---]  c0
0 [*--- ---]  c1
0 [*--- ---]  c2
0 [*--- ---] p1
1 [u 0 f---] p4
1 [u 0 f---] p5
1 [u 0 f---] p6
1 [u 0 f---] p0
1 [u 0 f---]  c3
0 [--- *---]  c4
0 [--- *---]  c5
0 [--- *---]  c6
0 [--- *---]  c0
0 [--- *---]  c1
0 [--- *---]  c2
...
```

As you can see, the producer hits the `p0` marker in the code and then grabs the

first value from its sleep specification, which in this case is 1, and thus each sleeps for 1 second before even trying to grab the lock. Thus, the consumer gets to run, grabs the lock, but finds the queue empty, and thus sleeps (releasing the lock). The producer then runs (eventually), and all proceeds as you might expect.

Do note: a sleep specification must be given for each producer and consumer. Thus, if you create two producers and three consumers (with `-p 2 -c 3`), you must specify sleep strings for each (e.g., `-P 0:1` or `-C 0,1,2:0:3,3,3,1,1,1`). Sleep strings can be shorter than the number of sleep points in the code; the remaining sleep slots are initialized to be zero.

## B threads-sema

Included in this directory are a number of skeletal code fragments, which you can fill out to solve various synchronization problems.

And see Allen Downey's book, "A Little Book of Semaphores", for even more fun. It's also free!

To compile any one of them (say, one called `foo.c`):

```
prompt> gcc -o foo foo.c -Wall -pthread
```

And then, to run it:

```
prompt> ./foo
```

(maybe with some optional arguments)