Server Software 2016/17
Resit Coursework

This coursework contains two tasks: channels and read-write locking. Each task will be marked individually and both tasks have the same weight. Please make sure you follow the rule for both tasks and the general notes on points that will gain or lose you marks in your code.

Task 1 – channels

There are two main approaches to concurrency. The first is to use global shared data and synchronisation primitives such as mutexes or semaphores to protect access to it. Threads communicate by reading and writing shared data.

The second approach uses channels. In this approach, each data item belongs to exactly one thread at any one time (apart perhaps from data used internally by the channels) and threads communicate by sending each other messages.

This task is about implementing a channel-based concurrency system using pthreads methods. Some example programs are provided that should run correctly together with your channels implementation.

Set-up and rules

Download the channels.zip file from the unit website. Your task is to implement the channels system defined in channels.h. If you put your implementation in a file called channels.c, the provided makefile will work without changes – you may also use more than one file for your implementation, in which case you must adapt the makefile.

- You must submit your implementation either as one or more .c/.h files or a .zip file containing your files.
- You must only submit your source files. If you do not submit all your source code, you will get 0 marks. If you submit binary, object code (.o) or other non-source files along with your source code, you will lose marks.
- If you need to modify the makefile, you must submit your modified makefile too.
- You **must not** modify channels.h or any of the provided example programs (producer.c and pipeline.c). When marking your work, I will create a folder with your submitted files and my version of channels.h and the provided example programs, and try and compile everything. If your submission does not compile with my files, you will get 0 marks.
- Your channels implementation must not abort the program in case of errors. Instead, the functions should return an error code (a value <0).

Assignment

Your task is to implement the methods declared in channels.h using pthread synchronisation primitives such as mutexes, condition variables or semaphores. Please also read the general notes at the end of this document and the more detailed documentation in the channels.h file.

Your implementation should be in one or more .c and .h files. The simplest option is to place all your code in a file named channels.c although you may split your code over multiple files. But you must not modify the existing .c and .h files.

Using channels

The idea behind channels is that if thread A wants to send a message to another thread B, then the program should pick a channel number n for these two threads and have thread A call ch_send(n, message) whereas thread B should call ch_recv(n, dest) to receive the message into the destination dest. Of course threads A and B need to 'agree' on the channel number, which could be accomplished by hard-coding the channel numbers for a program's intended purpose.

Using channels is sending and receiving data over sockets with a send and a recv (receive) method. However, for this coursework we fix that each channel has a 'capacity' of one message. This means that the first time a thread calls ch_send(n, message) the message is transferred to the channel and the ch_send call returns[1]. If you try and send a second message before anyone has received the first one, the second ch_send call blocks until the first message has been read. Calling ch_recv on a channel containing a message returns with the message; ch_recv on an empty channel blocks until a message is available. There is also a method ch_tryrecv that returns whether there is a message in the channel or not. In a table:

method	**channel empty**	**channel full**
ch_send	Transfer message to channel and return. The channel is now full.	Block until channel empty, then transfer message to channel and return. The channel is now full.
ch_recv	Block until the channel is full, then get the message. The channel is now empty.	Return with the message. The channel is now empty.
ch_tryrecv	Set *dest to NULL and return. The channel is still empty.	Like ch_recv.

For simplicity, we assume that there are exactly 8 channels numbered 0-7. This means that your implementation must work for these channel numbers and should report an error for any other channel numbers.

We assume that the following ownership rules apply to channel messages.

- All messages must live on the heap, never on a thread's stack. (You do not have to test for this in your implementation.)
- Calling ch_send transfers ownership of the message from the calling thread to the channel.
- When a thread gets a message with ch_recv, it obtains ownership of the message.

In practice, this means that a thread that wants to send a message must malloc() space for the message and the thread receiving the message is responsible for calling free() on the message.

More than one thread may send or receive messages on the same channel. If multiple threads try and send a message on a full channel or receive a message from an empty channel, they all block and the order in which they take their turns is undefined. However, each message must be received exactly once, i.e. if two threads are waiting on an empty channel and another thread sends a message on it then exactly one of the waiting threads gets the message and there is no rule to say which thread gets the message. In practice this means that you can use wait/signal or sem_wait/sem_post to synchronise sending and receiving on channels.

[1] ch_send and ch_recv may block temporarily on a mutex or semaphore, but if the channel is empty then ch_send must not wait for another thread to call ch_recv.

Example programs

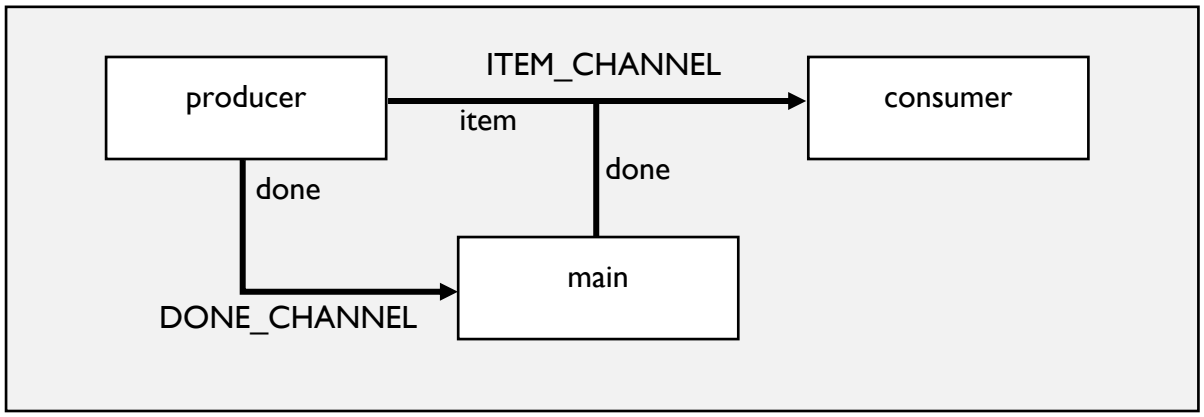
Two example programs are provided which you can link against your channels implementation. You may look at the provided programs but your channels implementation must get these programs to run without changing their source code.

A makefile is provided. Note that all code is set to compile to the 'gnu99' standard (C99 with GNU extensions such as usleep) and that we link two libraries: 'pthread' for threading and 'rt' to use the system's real-time clock for exact timing.

Producer-Consumer

The first example is a simple producer-consumer demo with one producer and one consumer. The source of this program is in producer.c and 'make producer' will compile it once you have implemented the channels. You can run the program with './producer'.

In this program, the producer produces 10 items and sends them over the channel to the consumer. When the producer is finished, it tells the main thread which shuts down both the producer and the consumer, then the program exits. The demo uses two channels as follows:



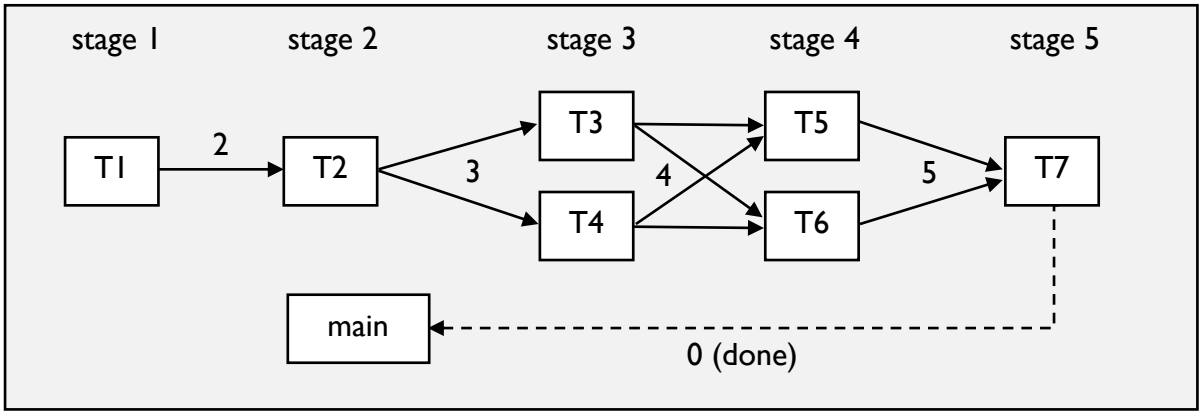
The producer produces items and sends them on the item channel; the consumer receives items on the item channel and consumes them. When the producer is done, it sends a 'done' message on the done channel which the main thread is listening on. When the main thread gets the 'done' message it joins the producer, sends 'done' on the item channel which causes the consumer to finish. Then the main thread joins the consumer and exits the program.

The consumer always takes 0.3s to consume an item whereas the producer takes 0.1s for the first item and then becomes 0.1s slower for each subsequent item, so the 10th item takes a whole second to produce. The expected output of the program is therefore that the producer initially must wait for the consumer, then from the 4th item onwards the consumer waits for the producer as the producer has become slower than the consumer.

Pipeline

The second example is a multi-stage pipeline. The source is in pipeline.c and the build command is 'make pipeline'. The pipeline consists of 5 stages. Stage 1 produces items and hands them off to stage 2, which hands them on to stage 3 etc. However, for stages 3 and 4 there are 2 threads instead of 1.

The channels are used as follows: the stage n threads listen on channel n and output their items on channel n+1 (except that stage 5 just consumes the items). Stage 5 sends a message to the main thread on channel 0 when all items are consumed.



The use of the different channels is indicated in the diagram above. Once thread 7 tells the main thread that it is done, the main thread sends 'done' messages to threads 2-6 as well on their input channels. Thread 1 terminates by itself when it is finished producing items.

The time taken to produce, consume or modify an item as it passes through the pipeline is non-deterministic so the program will produce different output each time.

Task 2 – read/write locking

Read/write locking increases performance compared to standard mutual exclusion when reading can be done concurrently and reading happens more often than writing. The basic idea is to do mutual exclusion on writing – only one thread can be writing at any one time, and only when no other thread is reading or writing. As long as no thread is writing, any number of threads can be reading concurrently. In a real implementation, one could have any number of read/write locks – in this coursework, we will use a single one and simulate a "reading room" where threads go to read and write data.

Assignment

The aim of the exercise is to write a C program that simulates readers and writers. Use the template file readingroom.c provided and complete the reader and writer functions. You will need to add global variables and synchronisation primitives. You may add initialisation in main() if you want to but this should not be necessary (my sample solution uses main() as it is now).

A reader thread must enter the room, execute the line read_documents(i.delay, i.id); when it is safe to do so and then leave the room. A writer thread must do the same with write_documents. The following rules apply:

1. A writer may only be in the room if they are the only person in the room. If any readers or writers are in the room, a writer must wait until the room is empty.
2. As long as no writers are in the room, any number of readers can enter the room.
3. Once someone has entered the room, they stay there for the required time (read_documents/write_documents causes a time delay).
4. The simulation must not deadlock under any circumstances.
5. If any writers are waiting to enter the room, no more readers may enter the room. Instead they must wait outside until the current occupants have vacated the room and any waiting writers have had their turn.

The final condition addresses the property called "starvation". If a writer is waiting and readers are constantly entering and leaving the room, but the room is never completely empty, there is a risk that the writer never gets a turn. Since the longest time that a reader can spend in the room is 12 "minutes", this guarantees that a waiting writer has at most 12 minutes to wait. We ignore the opposite problem of readers waiting for writers since we assume (a) readers are more numerous than writers and (b) readers don't mind waiting for writers, since they get the most up-to-date data when they do wait.

Submission

You must submit the following on SAFE:

1. The source code for your program in .c and .h files. You may place these in a single .zip file if you want – but you shouldn't need too many files (the sample solution is a single c file of around 200 lines, not counting the pthreads library).
2. NO object code or binary files.
3. A makefile called `makefile` that produces an executable named `readingroom` when the command `make` is run on one of the lab machines. I recommend using the 'gnu99' standard option to gcc.
4. Your program will be tested automatically and then marked by hand. For this reason it is important that your program's output is exactly in the specified format.

General notes

These notes apply to both tasks.

- You must submit the source code of your programs only as indicated in the individual tasks. Submitting binaries, object code, temporary files etc. will lose you marks; forgetting to submit your source code gets you 0 marks.
- The code you submit must compile on a lab machine. If your code does not compile as submitted you will lose marks; if your code does not compile and it is not easy to fix the mistake(s) you will get 0 marks.
- I recommend the following check after submitting. If it works on a lab machine then your submission counts as 'compiles correctly'.
    - Create an empty folder.
    - Download your submission into this folder.
    - If the assignment has files that you are not allowed to change (and do not need to submit), copy these files into your new folder too.
    - Try and compile and run the program(s) in this folder.
- I have zero tolerance for plagiarism. All your code will be checked against other students' code and code on the internet – anyone cheating will be penalised, in the worst case of multiple serious offences this could result in a student failing their degree.
    - All code that you submit must be your own work or properly referenced.
    - You must not share your code with or copy from another student under any circumstances.
- You are expected to code to a professional standard. This includes understanding how the functions that you call can indicate errors and how to handle them.
    - You must check the return value of malloc().
    - You must check the return value of functions that could do dynamic allocation such as pthread_mutex_init().
    - You should check the return value of all pthreads functions (and you will get more marks if you do).

Depending on the operating system you are using, what you are doing with pthreads (there are many more techniques such as re-entrant locks that we have not covered in the unit) and what other programs are doing (in particular regarding POSIX signals), it may or may not be necessary to check the return value of a specific call. But it is never wrong to check for errors after every single pthreads call, and doing this is the only way to write code that is (a) portable, (b) least likely to contain bugs and (c) remain like this when you or someone else changes it later on.

- You must not use busy-waiting, nor any variant of busy-waiting like checking a variable and then sleeping a fixed time all in a loop. If you want to wait for another thread, your options are pthread_cond_wait or sem_wait.
- You must use the correct pattern when waiting on a condition variable (predicate and loop) as spurious wakeups can happen.
- Your code must not contain any data races – all global data must be protected by mutexes or semaphores. This requires a bit of thinking as well as typing when you create your program, and you should review your code after writing it.
- When a variable is shared between threads, you must protect reads as well as writes unless the variable is read-only for all threads as soon as more than one thread is alive.