### (over due)

### Milestone 6 (Final Milestone for Assignment 1)

#### *Using Pipes*

The standard shell allows programs to be chained up through pipes. This way the output of one program can be used as the input into the next program. For example, executing ls | sort -r will list the content of the current directory in reverse alphabetical order.

##### How does the shell achieve the above behaviour?

It runs the /bin/ls command in the background and re-directs its standard output into a pipe. The pipe is then fed into the /usr/bin/sort program: its standard input is re-directed from the pipe.

##### What is a pipe?

A pipe is a mechanism that lets one process send data to another through normal file system functions such as [read()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?read+2) and [write()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?write+2) (and all the higher level functions on top of that, e.g. [fgets()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?fgets" \t "_blank), [fprintf()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?fprintf" \t "_blank), etc.). A pipe does not actually create a file on disk, but is provided as an inter-process communication mechanism by the Operating System and treated like a virtual file. Any data written into the writing end of the pipe can be read from the reading end (e.g. by another process).

#### *Running Scripts*

Most shells allow running scripts by redirecting input from a file. E.g. if you enter the command source somefile.sh this will cause the shell to read and execute the commands from the file somefile.sh before transferring control back to standard input.

#### *Command Line Editing*

Modern shells have more comfortable command line editing. For example, they allow the cursor keys to be used to go back and forth through command history and for editing within the current line. Most shells also allow command completion using the TAB key.

For this, the terminal allows setting raw mode through the [tcsetattr()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?tcsetattr" \t "_blank) function. However, reading and interpreting raw keystrokes through interpreting the [termincal capabilities (termcap)](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?termcap" \t "_blank) can be cumbersome. A simpler way is to use the [readline() library](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?readline" \t "_blank) that comes with most POSIX Operating systems.

#### *Implementation Steps*

1. Allow the vertical bar | character to be used to separate multiple programs (instead of the semicolon), in which case stdout of the previous program should be directed into a pipe that is fed into stdin of the subsequent program.
2. Commit the previous step to the repository!
3. Add a source command with a file name parameter, causing your shell to read commands from that file and then return control back to standard input.
4. Commit the previous step to the repository!
5. Add [readline()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?readline" \t "_blank) support so users can use the cursor keys to browse history and edit commands they enter. Make sure you only use [readline()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?readline" \t "_blank) when reading from standard input (but not while commands are read from a file using your source command)!
6. Commit the previous step to the repository!

Use the git tag command to tag your code as **milestone6** before creating and **submitting a ZIP file containing the folder with all your files (including everything from the previous milestones as well)**!

* Milestone 7 **(over due)**

***GIT Repository***

Before you start, make sure you create*a new GIT* repository for assignment 2 inside your (empty) working directory, e.g.:

mkdir assignment2  
cd assignment2  
git init

Don't forget to commit all changes you are making in the assignment (using git commit).  To get marks for the repository, after all the steps in the following sub-sections, you need to create a milestone1 tag for your final milestone submission using the git tag command.

Make sure you only perform the last step above **after you have completed all the steps below** but before you zip up and upload your assignment to be marked!

***First Steps in Synchronisation***

The objective of this part is to explore simple thread synchronisation using mutexes (simple locks) by implementing a program called *sync* that creates a child thread, reads (in the child thread) a line of input from the user, and sends that line to the parent thread for printing (to stdout). The child thread should then wait for the user to hit return (or enter), then notify the parent thread and exit.  The parent thread should wait for the child to exit, then print "child thread is gone" and also exit.

***Implementation Steps***

* 1. Create a new program sync.c and add it to the Makefile and the subversion repository.
  2. Add a buffer (that holds at least 256 characters) to your parent thread. Pass the address of that buffer to the child thread.
  3. In the child thread, read one line of input into the buffer.  Print the buffer to stdout from within the parent thread (no need to wait for the child at this stage). Explain in your README what happens.
  4. Use a mutex to make the parent wait for the child to have read a line before displaying the line. Hint: you will need to pass both the buffer and the mutex to the child. **Don't ever use global variables;** you can instead create a struct that contains all the variables you need to pass to the child thread!
  5. After the parent thread has printed the output, make the child thread prompt the user to hit Enter (and wait for the user to do so). Hint: You will need a second mutex for this. Explain in your README why this cannot be done reliably with a single mutex! How reliable is an approach that uses 2 mutexes? Explain if there is a scenario where even two mutexes might not be enough.
  6. Make the child notify the parent once the user has hit the enter key and then exit. The parent should wait for the notification, print that the child is exiting, then wait for the child to exit (and print the "child thread is gone" afterwards) and then end the whole program (process). Do you need a third mutex for this? Will this be reliable under all circumstances? Explain why/why not!

Your program needs to compile without warning when using

clang -std=gnu99 -Wall -o sync sync.c -lpthread

Add your program to the repository using git add sync.c and commit it after every change using git commit -am "*log message*" (replace *log message* with a text that descripts the changes you have made).

***Makefile***

Create a Makefile that compiles your program so that you can run it from the command line using ./sync (followed by enter).

Add your Makefile to the repository using git add Makefile and commit it after every change using git commit -am "*log message*" (again, replace *log message* with a descriptive log message).

***Targets: clean and all***

Add a clean target that removes all files that are generated when you run make (e.g. all .o files and the final executables).

Add an all target as the very first target in your Makefile to eventually compile all milestones in this assignment (of course, this only needs to compile the first milestone at this stage).

***Submission***

Don't forget to commit all steps to the repository individually.  When you are done with the milestone, use the git tag command to **tag everything as milestone1**! Make sure you extend the clean and all targets of your Makefile for all the subsequent milestones in the coming weeks!

* https://bblearn.griffith.edu.au/images/ci/sets/set01/assignment_on.gif

Milestone 8 **(over due)**

***Creating a Semaphore***

In the previous milestone, there was one race condition that would still remain. *This problem cannot be avoided by using mutexes alone!*

This can be prevented by using a Semaphore (since Semaphores can to be unlocked by a different thread than the one that originally locked it). The following pseudo-code implements a generic Semaphore that, unlike the traditional Semaphores discussed in the lecture, *can be initialised with negative values:*

procure(Semaphore \*semaphore)

{

begin\_critical\_section(semaphore); // make the following concurrency-safe

while (semaphore->value <= 0)

wait\_for\_vacate(semaphore); // wait for signal from vacate()

semaphore->value--; // claim the Semaphore

end\_critical\_section(semaphore);

}

vacate(Semaphore \*semaphore)

{

begin\_critical\_section(semaphore); // make the following concurrency-safe

semaphore->value++; // release the Semaphore

signal\_vacate(semaphore); // signal anyone waiting on this

end\_critical\_section(semaphore);

}

In the above code, the procure() function is similar to semWait() from the lecture, and vacate() is similar to semSignal().

***Semaphore Implementation***

* 1. Create a new module sema.c with an accompanying sema.h that implements the above algorithm in C (and add it to your git repository). Embed all the data needed to implement your Semaphore in a single struct. Don't use any existing POSIX or other Semaphore implementation in your code! *Hint:* you can use a *mutex* to protect your critical section, and a *condition variable* to wait/signal that a Semaphore has been vacated (see also the hints section on the top level assignment page!).
  2. Add an *initialiser* function to your code that sets up a Semaphore and intialises it to a pre-defined (integer) value.
  3. Add a *destructor* function to your code that cleans up properly after the use of a Semaphore (e.g. destroys all mutexes, and condition variables, and releases all memory allocated for a Semaphore).
  4. Test your code with a separate program sematest.c that re-implements last week's synchronisation steps but this time uses semaphores.  Use the smallest number of semaphores necessary (and make sure you don't use any other synchronisation constructs outside your semaphore implemetation (with the only exception being pthread\_join()))!  Your test needs to include a case that clearly shows the difference between using a mutex and a Semaphore! *Hint:* if you add a sleep at the beginning of the child thread in last week's code you should be able to demonstrate how your Semaphore implementation prevents the mutex problem discussed above! If you do use last week's code, make sure you copy the code into a new file first (don't modify last week's milestone directly)!
  5. Add a short test report to your documentation!

***Event Counter***

An Event Counter is another task synchronisation instrument that consists of an integer (with an initial value of 0) and atomic operations. The atomic operations are as follows:

**read(Eventcounter)**

returns the current (integer) value of Eventcounter

**advance(Eventcounter)**

increments the Eventcounter by one

**await(Eventcounter, value)**

blocks the current thread until Eventcounter >= value

***Sequencer***

Sequencers are closely linked with event counters. A Sequencer is also an integer (initialised to 0) that is used to serialise the occurrence of events. A sequencer only has one atomic operation, ticket(), that returns the current value of the sequencer and then increments the sequencer by one.

***Implementation***

Implement separate sequencer and event counter modules (with separate implementation (.c) and header (.h) files) in a similar fashion to the semaphore implementation above.  Create a separate test module that re-implements last week's synchronisation steps but this time uses your sequencers and event counters.  Make sure you use the smallest number sequencers and event counters necessary (and make sure you don't use any other synchronisation constructs outside your sequencer/event counter implemetation (with the only exception being pthread\_join())).  Test appropriately (see above).

***Submission***

Don't forget to any add all new files to the Makefile and your repository as usual. When done with the milestone, use the git tag command to **tag your files as milestone2** before zipping up your folder and submitting it!

* https://bblearn.griffith.edu.au/images/ci/sets/set01/assignment_on.gif

Milestone **9 (over due)**

***64 Bit Random Number Generator***

The goal of this milestone is to solve the Producer/Consumer problem for a random number generator by using one of the algorithms discussed in the lecture, i.e., either by using your semaphores from last week, or by implementing a monitor using condition variables and mutexes, or by using blocking message passing primitives.  One thread acts as a random number generator that reads true random numbers from the reliable kernel random number source device [/dev/random](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?random+4) and puts them into a buffer.  Another thread consumes these random numbers as they come in and prints them on standard output.

It is important that the buffer does not overflow (i.e., the random number generator does not try to put more random numbers into the buffer than the buffer can hold). It is also imporant that the buffer does not underflow (i.e., a random number may only be taken out of the buffer if that does not take the total number that remains in the buffer below the minimum fill level).

***Implementation Steps***

(Don't forget to also have a look at the hints section!)

* 1. Create a function that, in a loop, reads high quality 64 bit random numbers of type uint64\_t (as defined in <stdint.h>) from [/dev/random](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?random+4) as fast as it can.
  2. Create a data type (e.g. a struct) for the buffer (queue) with an appropriate constructor (intialiser) and destructor (cleanup function). The constructor should take at least two (integer) parameters: a (maximum) buffer size, and a minimum fill level.
  3. Create a put\_buffer() function that puts a value into the buffer so that it can be read later. For the moment, you can ignore the maximum buffer size and concurrency.
  4. Create a get\_buffer() function that reads the oldest value that was put into the buffer, removes that value from the buffer and returns it. For the moment, you can ignore the minimum fill level and concurrency.
  5. Create your own synchronisation data types or use the operating system provided ones to implement your choice of synchronisation (i.e. monitor, semaphores, or message passing)
  6. Modify your protection code to respect the maximum buffer fill level.  Make sure that the producer waits if the maximum fill level is reached!
  7. Modify your protection code to respect the minimum buffer fill level.  Make sure that the consumer waits if the minimum fill level is reached!
  8. Create a test program that uses threads or dispatch\_async() to spawn the concurrent producer as a child thread. I.e. the task should use the function you created earlier to read values from [/dev/random](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?random+4) as fast as it can and immediately put each value into the buffer. A separate task should act as the consumer: it should read and remove one number from the buffer, and print it to stdout as a 16 digit lower-case *hex* numbers prefixed by $ (e.g. $feed0badbeef1234 for the 64 bit decimal number 18369351295974707764).
  9. Add two*command line parameters*, a maximum buffer size, and a minimum fill level. These command line parameters should be optional to the user – by default, the maximum buffer size should be 4, and the minimum fill level should be 0 (zero).
  10. Add a loop to the main thread that reads user input from stdin, one line at a time. The user input should be interpreted as the number of integers that next should be read from the buffer and printed to stdout in one go (i.e. before the parent waits for the next input from the user). Ensure that the integers are read and printed out in the order they were produced!  Print each integer immediately after reading it from the buffer!  E.g., if the user types 25 (followed by enter), 25 integers should be read from the buffer and printed to stdout; then if the user types 3 another three integers should be read from the buffer and printed.
  11. Add an *exit* command that cleans up any resources and exits the program.

Don't forget to any add new files to the Makefile and your repository as usual. When done with the milestone, use the git tag command to **tag your files as milestone3** before zipping up your folder and submitting it!

* https://bblearn.griffith.edu.au/images/ci/sets/set01/assignment_on.gif

Milestone 10 **(due date : in 3days)**

Attached Files:

* 1. [[File](https://bblearn.griffith.edu.au/bbcswebdav/pid-903378-dt-content-rid-2106501_1/xid-2106501_1) NetStat.tar.bz2](https://bblearn.griffith.edu.au/bbcswebdav/pid-903378-dt-content-rid-2106501_1/xid-2106501_1) (8.748 KB)

***Network Status Monitor***

Solve the reader/writer problem using semaphores by implementing a network status monitor that tracks network statistics on the system. The writer should monitor packet numbers once per second and update a status area containing the number of correctly transmitted and erroneous input and output packets as well as a time stamp of the update. The reader(s) should be able to read and display the latest status information at any time. A writer trying to update the status information needs to have priority over readers trying to read the information.

***Implementation Steps***

(Don't forget to also have a look at the hints on the main assignment page!)

* 1. Download and test NetStat.tar.bz2 (the attachment on this page). This contains a function get\_net\_statistics() that retrieves network statistics. (You can use this function unchanged in your code by adding netstat.c and netstat.h to your own Makefile.)
  2. Create a write\_net() function that writes the current network status (struct net\_stat) into the status area.
  3. Create a read\_net() function that returns the current information from the status area.
  4. Use *semaphores* for synchronisation and to protect your status area. I.e., you need to solve the reader/writer problem to make sure that the network statistics can be read and written atomically. Make sure that writers have priority over readers (see the lecture notes).
  5. Create a test program that spawns a child thread that acts as the writer. I.e. the child should use the get\_net\_statistics() function from the example code to retrieve network status information once a second and write that to the status area. The main (parent) thread should, for the moment, act as the reader (i.e. read the network statistics from the status area and print everything to stdout).
  6. Add a loop to the main thread that reads user input from stdin, one line at a time. The user input should be interpreted as the number of reader threads (additional child threads) that should read the the status information and print to stdout before the parent waits for the next input from the user. Print each statistic immediately after reading it from the buffer (from within these child threads), but make sure threads don't interfere with each other when displaying the information by adding mutual exclusion around your output to stdout! E.g., if the user types 10 (followed by enter), the status information should be read and printed 10 times, once by each reader thread; then if the user types 3 another three lines of output should be read and printed. Make sure every intermediate update from the writer is reflected in that output (testing writer priority). You can keep around reader threads for re-use or exit them immediately after having read and printed their status information.
  7. Add an exit command that tells the child thread to immediately exit. The parent thread should then wait for the child thread(s) to exit, clean up any resources and exit the program.

Don't forget to add new files to your repository and the Makefile as usual. When done with the milestone, use the git tag command to **tag all your files as milestone4, then submit**!

Hints :

### Hints for Milestone 6

**Creating Pipes**

A(n unnamed) pipe can be created using the [pipe()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pipe+2) system call. This system call fills in an array of two integers with the file descriptors for the corresponding pipe. The first integer is the file descriptor for reading from the pipe (thereading end of the pipe), while the second integer is the file descriptor for writing data into the pipe (the writing end of the pipe). These file descripts will be inherited by child processes when [fork()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?fork) is called. It is important that each process closes the end of the pipe that it does not use (e.g. the writing process should close the reading end and vice versa). See the [PipeExample code](http://www.ict.griffith.edu.au/teaching/3420ICT/software/PipeExample.tar.bz2) for a simple example.

**Re-directing stdio to and from Pipes**

You can re-direct stdout and stdin to and from a pipe using [dup2()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?dup2). The [dup2()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?dup2) function takes two file descriptors (ints) and copies the first file descriptor over the second. This way, you can copy the file descriptor you received from [pipe()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pipe+2) to STDIN\_FILENO (for stdin) or STDOUT\_FILENO (for stdout).

**Using readline()**

Read the [readline user manual on the info system](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/info2html.cgi?(rluserman)Top" \t "_blank) for a comprehensive guide on how to use history and command line completion. You can also [download a readline example here](http://www.ict.griffith.edu.au/teaching/3420ICT/software/rltest.tar.bz2).

### Hints for Milestone 7 (and later milestones)

**Creating and Exiting Threads**

The [pthread\_create()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_create" \t "_blank) function can be used to create a child thread. Unlike fork(), [pthread\_create()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_create" \t "_blank) actually takes a function to invoke when the child thread gets started. The child thread ends when that function returns or [pthread\_exit()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_exit" \t "_blank) is called by the child. (Warning: never call [exit()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?exit+3) from within a child thread, as this will exit the whole process, not just the thread!)

**Waiting for Threads**

The [pthread\_join()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_join" \t "_blank) function waits for a child thread to exit and collects its exit status. Simpilar to processes, by default, all exited children need to be joined (waited for) to prevent hitting the maximum number of allowed child threads within a process!

**Creating and Destroying Mutexes**

A mutex is a simple variable of type pthread\_mutex\_t. Before you can use a mutex, you will need to initialise it by passing its address to [pthread\_mutex\_init()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_mutex_init" \t "_blank). If you no longer need a mutex, you need to call [pthread\_mutex\_destroy()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_mutex_destroy" \t "_blank) to free all resources allocated for that mutex.

**Locking a Mutex**

You can lock a mutex using [pthread\_mutex\_lock()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_mutex_lock" \t "_blank) – this will wait for the mutex if it is currently locked. Note: by default, mutexes are not recursive, so you must not attempt to lock a locked mutex from within the same thread!

**Unlocking a Mutex**

To unlock (release) a mutex you have locked (and signal all threads waiting for that mutex), use [pthread\_mutex\_unlock()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_mutex_unlock" \t "_blank).

### Hints for Milestone 8 (and later milestones)

**Creating and Destroying Condition Variables**

A condition variable is of type pthread\_cond\_t. Before you can use a condition variable, you will need to initialise it by passing its address to [pthread\_cond\_init()](http://www.ict.griffith.edu.au/teaching/3420ICT/wrapper.php?link=cgi-bin/man.cgi?pthread_cond_init). If you no longer need a condition variable, you need to call [pthread\_cond\_destroy()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_cond_destroy" \t "_blank)

**Waiting on a Condition Variable**

You can wait on a condition variable using [pthread\_cond\_wait()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_cond_wait" \t "_blank). Note that this function takes a mutex as a second parameter. That mutex must be locked at the time [pthread\_cond\_wait()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_cond_wait" \t "_blank) gets called and will get unlocked for the period of time (if any) the thread is actually waiting for the condition.

**Signalling a Condition**

You can signal any thread waiting on a condition variable using [pthread\_cond\_signal()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?pthread_cond_signal" \t "_blank), which will wake up one thread waiting on the given condition variable.

### Hints for Milestone 9 (and later milestones)

**Reading random numbers from**[**/dev/random**](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?random+4)

[/dev/random](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?random+4) is a simple file that you can [open()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?open+2) and [read()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?read+2) from. The easiest way to read a (binary) integer, is to use the [read()](http://www.ict.griffith.edu.au/teaching/3420ICT/wrapper.php?link=cgi-bin/man.cgi?read+2) function (simply pass the address of a uint64\_t variable as the buffer to read into, and sizeof(uint64\_t) as the size (count)).

**Printing 64 bit hex numbers**

You can cast your uint64\_t to unsigned long long and print it using the %llx format placeholder in [printf()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?printf+3" \t "_blank).

**Creating a dynamic buffer**

Just use a circular buffer as discussed in the lecture. Since the buffer size is a command line parameter, you can use [malloc()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?malloc" \t "_blank) to allocate memory for your buffer (don't forget that [malloc()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?malloc" \t "_blank) requires the size of the buffer in bytes!).

**Using GCD**

There are some useful hints in the [Dispatch Queues chapter in the GCD Concurrency Programming Guide](http://developer.apple.com/library/mac/#documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationQueues/OperationQueues.html#//apple_ref/doc/uid/TP40008091-CH102-SW1).

**Using GCD Semaphores**

Instead of your own semaphores, you can use [dispatch\_semaphore\_create()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?dispatch_semaphore_create" \t "_blank) to create libdispatch semaphores and then use these.

### Hints for Milestone 10 (and later milestones)

### Dispatching

A good way to implement dispatching is to use [dispatch\_sync\_f()](http://www.ict.griffith.edu.au/teaching/3420ICT/cgi-bin/man.cgi?dispatch_sync_f" \t "_blank). Alternatively, you can use threads and semaphores.