CSU23016

Concurrent Systems
&
Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

Student Online Teaching Advice Notice

The materials and content presented within this session are intended solely for use in a context of teaching and learning at Trinity.

Any session recorded for subsequent review is made available solely for the purpose of enhancing student learning.

Students should not edit or modify the recording in any way, nor disseminate it for use outside of a context of teaching and learning at Trinity.

Please be mindful of your physical environment and conscious of what may be captured by the device camera and microphone during videoconferencing calls.

Recorded materials will be handled in compliance with Trinity's statutory duties under the Universities Act, 1997 and in accordance with the University's policies and procedures.

Further information on data protection and best practice when using videoconferencing software is available at https://www.tcd.ie/info_compliance/data-protection/

‘Shells’

- A ‘shell’ is a program that allows you to give commands to a computer and get responses.
  - Common GUI-based shells include ‘Finder’ for Mac OS X and ‘Explorer’ for Windows. These are easy to learn but hard to automate.
  - Command-line text-based shell programs are harder to learn, but support a high degree of automation. These are typically accessed through a "console window" program
    - called "Terminal" on Unix/Linux/macOS
    - several, such as "Cmd" or "Powershell", on Windows
  - Common UNIX shell languages are SH, BASH, CSH, TCSH, ASH, ZSH. On Windows, the provided shells use a custom language, and a totally different approach to running programs. However, UNIX shell support is possible
    - try using "Cygwin" (http://www.cygwin.com) or "Windows Subsystem for Linux" (WSL)
    - Or use PuTTY (https://www.putty.org), or similar to make an SSH connection to macneill.scss.tcd.ie

The bash shell

- A early shell, called "sh" was developed for an early version of Unix,
  - by Stephen Bourne, at AT&T.
- A modified free version, called "bash" was written for Linux
  - It's the "Bourne-Again Shell" !
- We will use this shell for this course.
- The Shell Command-Line/Scripting Primer
  - see links in READING content area on Blackboard

Important shell principles

- Shell commands generally invoke programs to so the work:
  - e.g. when you write `ls` on the command line, the shell looks for a program called `ls` in a few designated places. Once if finds the program, it executes it.
- Most programs work with standard character-based input sources and output sinks — 'standard input', 'standard output' and 'standard error'. These are 'pipes' and can be connected to files, the terminal window, other programs, network feeds, etc. By default,
  - standard input ("stdin") is connected to your keyboard.
  - standard output and error ("stdout", "stderr") are connected to your console/terminal window.
- Most of these programs do something simple but very well — to do complex things, you string programs together.

Example: Practical I

1. Download the file `sumofhellos.c`) attached to this practical.

2. Modify line 6 (#define USERNAME "username") to replace username with your TCD username.

3. Compile the code - warnings are OK, but there should be no errors:
   ```
   cc -o sum sumofhellos.c -pthread
   ```

4. Run the executable (your username should appear seven times, as well as some number between 1 and 6 at the end):
   ```
   ./sum
   ```

5. Obtain a log of the compilation and running of the program by either:
   1. selecting it from your terminal window, copying it, and pasting in a text file called `practical1.log`.
   2. Or running the command
      ```
      ./sum > practical1.log
      ```

6. Bundle both your source code and logfile into a "tar" archive
   ```
   tar -r -f P1.tar sumofhellos.c practical1.log
   ```
   This will create a file called `P1.tar`.

7. Submit file `P1.tar` through Blackboard by the deadline.

Follow these steps precisely! This is really what is being assessed.

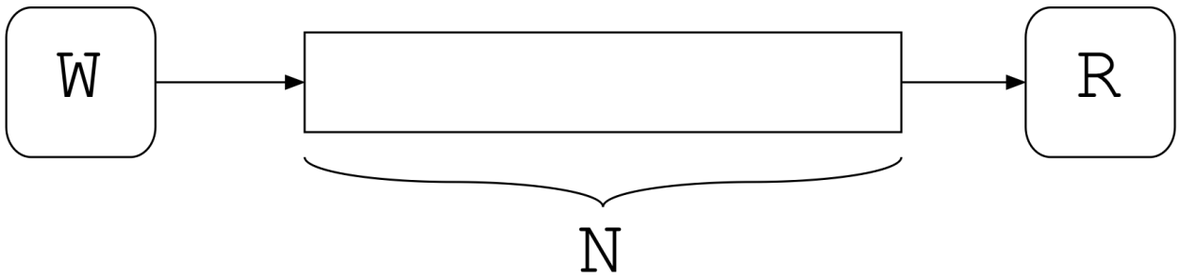
Combining commands

- Multiple commands:
  - Issue a sequence of commands on one line by separating them with a semicolon (“ **;** ”).
- Piping:
  - Using the bar symbol (“ | ”), you can direct (“pipe”) the standard output of one command into the standard input of the next one.
- Redirection:
  - You can redirect standard input to accept the contents of a file using symbol "<", and standard output to go to a file using symbol ">".
- Automation:
  - You can write a text file containing sequence of commands and shell commands and constructs. This can be executed as a *shell script*.

Producers and Consumers

- Essentially a *pipeline* of separate sequential programs.
  - E.g. concatenation of unix commands.
- Programs communicate via buffers.
  - Implemented in different ways, but, e.g.
    - Shared memory, flags, semaphores, etc.
    - Message passing over a network
- Flow of data is essentially one-way.

Imagine a Situation…



- We have a buffer that holds N items, with a counter recording how many items present.
- A 'Producer' writes (W) items to the buffer from time to time, incrementing its counter.
  - if the buffer is full, then the Producer should wait until room is freed up.
- A 'Consumer' reads (R) items from the buffer now and then, decrementing its counter.
  - if the buffer is empty, the Consumer should wait for a new item to arrive
- We could use a mutex to control access to the buffer & counter.

Write/Read behaviour (pseudo-code)

- Write(item,buffer) should do the following:
  - `if full(buffer) then wait_until_room(buffer);`
    `insert(item,buffer).`
- item = Read(buffer) should do the following:
  - `if empty(buffer) the wait_until_something(buffer);`
    `item = extract(buffer).`
- How do we use mutexes here?

Write/Read behaviour (pseudo-code)

- What's wrong with the following?:

  - ```
    lock(mutex);
    if full(buffer) then wait_until_room(buffer);
    insert(item,buffer);
    unlock(mutex).
    ```

  - ```
    lock(mutex);
    if empty(buffer) the wait_until_something(buffer);
    item = extract(buffer);
    unlock(mutex).
    ```

Both will wait forever, because, holding a lock on the mutex **while waiting,** prevents the other from ever getting the lock.

(A situation known as "DEADLOCK" !)

Write/Read behaviour

- We need to be able to:
  - wait for a state-change in global shared state (our buffer+counter),
  - without holding the lock all the time.
- The following does give the Consumer a chance to grab the lock. Will this work?
  -
```
wait = true;
while wait do
  lock(mutex); wait=full(buffer); unlock(mutex)
endwhile;
lock(mutex); insert(item,buffer); unlock(mutex).
```

Here the Producer is rapidly locking, checking, unlocking
- the Consumer risks "STARVATION" !

Mutexes aren't enough!

- Mutexes by themselves prevent efficient solutions
  - particularly when waiting for a resource to be in some specific state before proceeding with an operation on that resource.
- One solution to the previous problem is to require the waiting thread to sleep for a time interval before checking again:
  - ```
    wait = true;
    while wait do
      lock(mutex); wait=full(buffer); unlock(mutex);
      if (wait) then sleep(100);
    endwhile;
    lock(mutex); insert(item,buffer); unlock(mutex).
    ```
- The problem here is it is hard to find an optimal value for the sleep interval
  - why 100 here? why not 50? why not 500?, why not 5 ?(what units of time are they anyway?!)
    ```

Condition Variables

- A generalised synchronisation construct.
- Allows you to acquire mutex lock when a *condition* relying on a shared variable is true and to sleep otherwise.
- The ‘condition variable’ is used to manage the mutex lock and the signalling.
- Slightly tricky.

Condition Variables - Dramatis Personae

- A mutex variable M,
- A shared resource R to be guarded by M,
- A condition variable V,
- A condition C,
- A thread U that wishes to use R, protected by M, on condition that C is true,
- A thread S that will signal V, presumably when it has done something that might indirectly change the value of C.

Condition Variables vs. Conditions

- We need to be careful here to distinguish between "Conditions" and "Condition-Variables".
  - If not, much confusion will ensue
- Condition C
  - This is an application dependent true/false statement about something we want to be true in order to proceed
    - e.g. buffer is not full
- Condition-Variable V
  - this is a complex datastructure provided by a thread library to manage the mutexes and signalling required

From Thread U's POV (1)

- Acquire Mutex M
  - This controls access to R,
  - but also must control access to any shared variables that C depends on.
- if C is true, the mutex can be used as normal.
  - so proceed to access R as planned
- if C is false…

From Thread U's POV (2)

- If C is false, wait for condition variable V to be *signalled*.
- This is tricky, as access to R is controlled by M.
  - So, Mutex M is unlocked,
  - Thread sleeps, to be woken when V is signalled,
  - Then, M is re-acquired.
    - No guarantee you'll get it right away–maybe another thread will.
- So now, if V has been signalled…

From Thread U's POV (3)

- Since V has been signalled, there is a chance that the condition C is now true.
  - but another thread may have slipped in and done something that makes it false again!
- Re-evaluate C:
  - If true, then the mutex is available for you to use,
  - If false, back to previous page.
- Our 'if' needs to be replaced by a 'while'…

From Thread U's POV (4)

- Acquire Mutex M
- While !C
  - Unlock M
  - Wait for V to be signalled
  - Lock M
- process R as planned
- Unlock M (finished)

```
pthread_lock(&m)
while (!C) {
   // m is locked by U here
   pthread_cond_wait(&v,&m)
   // m is locked by U here
}
… access and modify R …
pthread_unlock(&m)
```

Thread S (1)

- Thread U is the 'user' of the condition variable V.
- If the condition is not true, U unlocks V's mutex M and sleeps, waiting for some other thread S to signal V and thereby waking U to check the condition again.

Thread S (2)

- Acquire Mutex M

```
pthread_lock(&m)
```

- Do something that affects R (and indirectly C)

```
Do something to R
```

- Signal the condition variable

```
pthread_cond_signal(&v)
```

  - This will awaken a thread that is waiting on the condition variable.

- Unlock Mutex M

```
pthread_unlock(&m)
```

My head hurts !

- Pthreads are not simple, nor straightforward
- There are (literally) a ***lot*** of moving parts!
- There are some key concepts that we need to understand
- The Lawrence Livermore tutorial gives a comprehensive overview
  - linked to from Blackboard, in the "READING/POSIX Threads" content area

“Formalising” Pthreads

- It will help to have a “formal” description of the PThread API calls we use.
- Why “formal” and not formal?
  - A Formal approach would use some form of mathematical logic to give a precise, unambiguous description of the API behaviour and usage.
    - We won’t go that far!
  - This “Formal” approach will state the properties needed using careful natural language

Properties of Interest

- Prototypes: types of values passed to and returned from API calls
  - Sometimes referred to as "(type-)signatures".
- Pre-Conditions: what needs to be true when we call an API function
- Post-Conditions: what will be true when we return from an API call
  - Provided the Pre-Condition was satisfied when it was called (!!)
- Invariants: what should always be true.
- In what follows,
  - we describe properties observed by the thread that calls the pthread function under discussion

pthread_mutex_init()

- Prototype: `int pthread_mutex_init( pthread_mutex_t *`
  `, const pthread_mutexattr_t * );`
- Call: `rc = pthread_mutex_init( &mymutex, NULL );`
- Precondition: None.
- Postcondition: `mymutex` is initialised, and unlocked.
- Invariant: all mutex variables (should be) initialised before use

pthread_mutex_lock()

- Prototype: `int pthread_mutex_lock( pthread_mutex_t * );`
- Call: `rc = pthread_mutex_lock( &mymutex );`
- Precondition: `mymutex` was initialised.
- Postcondition: `mymutex` is locked, and the calling thread "owns" that lock.
  - time may have passed if another thread owned the lock when this thread made the call.
- Invariant:
  - When a mutex is unlocked, no thread "owns" it.
  - When a mutex is locked, exactly one thread "owns" it.

pthread_mutex_unlock()

- Prototype: `int pthread_mutex_unlock( pthread_mutex_t * );`
- Call: `rc = pthread_mutex_unlock( &mymutex );`
- Precondition: `mymutex` is locked and calling thread "owns" that lock
- Postcondition: this thread no longer "owns" the lock
- Invariant:
  - When a mutex is unlocked, no thread "owns" it.
  - When a mutex is locked, exactly one thread "owns" it.

pthread_cond_init()

- Prototype: `int pthread_cond_init( pthread_cond_t *, const pthread_condattr_t * );`
- Call: `rc = pthread_cond_init( &myconvar, NULL );`

- Precondition: None.
- Postcondition: `myconvar` is initialised, and has not been "signalled".
- Invariant: all condition variables (should be) initialised before use

pthread_cond_wait()

- Prototype:
```
int pthread_cond_wait( pthread_cond_t *
                     , pthread_mutex_t * );
```
- Call:
```
rc = pthread_cond_wait( &myconvar, &mymutex );
```
- Precondition:
  - `myconvar` is initialised, and `mymutex` is locked.
  - the calling thread "owns" the lock
- Postcondition: `mymutex` is locked, and the calling thread "owns" that lock.
  - time may have passed until another thread signalled to `myconvar`.
- Invariant: ?

pthread_cond_signal()

- Prototype: `int pthread_cond_signal( pthread_cond_t * );`
- Call: `rc = pthread_cond_signal( &myconvar );`
- Precondition:
  - a previous call to `pthread_cond_wait`, usually by some other thread, has associated `myconvar` with `mymutex`.
  - `mymutex` is locked and "owned" by the calling thread.
- Postcondition:
  - a signal has been sent to `myconvar` to tell it to wake a thread sleeping on. on it.
  - `mymutex` is still locked and "owned" by the calling thread.
- Invariant: ?