COMP1521 26T1

Week 10 Lecture 1

Concurrency, Parallelism and Threads and Revision

Announcements: Assessments

Lab10 is due Tuesday midday in week 11 due to the Monday public holiday

Assignment 2: Get started ASAP if you have not already. Reach out for help to your tutor and help session tutors.

Announcements: Exam

Exam Preferences form:

You should have received an email with a form to choose exam preferences.
Please refer to this forum post if you didn’t.

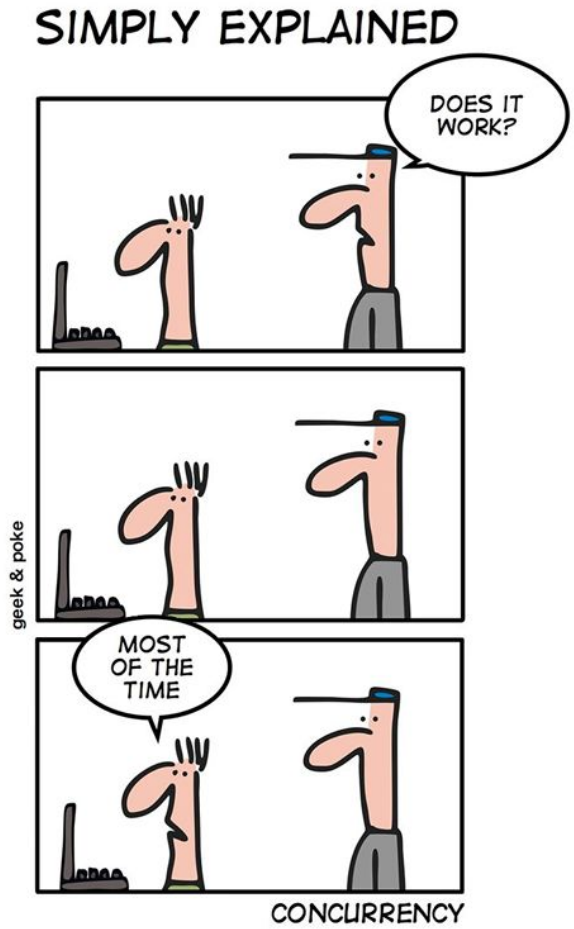
Final Exam Time Preference Form Is Available - announcements - COMP1521

Announcements: Revision

- **Practice Exam** this week in in-person labs.
  - These questions will not be released
- **Past exam** papers
  - First paper release on Wednesday week 10.
  - 4 will be released in total
- **Revision Sessions in week 11**
    - First session basics (MIPS, bitwise, basic files)
    - Second session (files, UTF-8, processes, threads)
- **Week 10** Final Lecture:
    - Poll to choose the topics you want to see

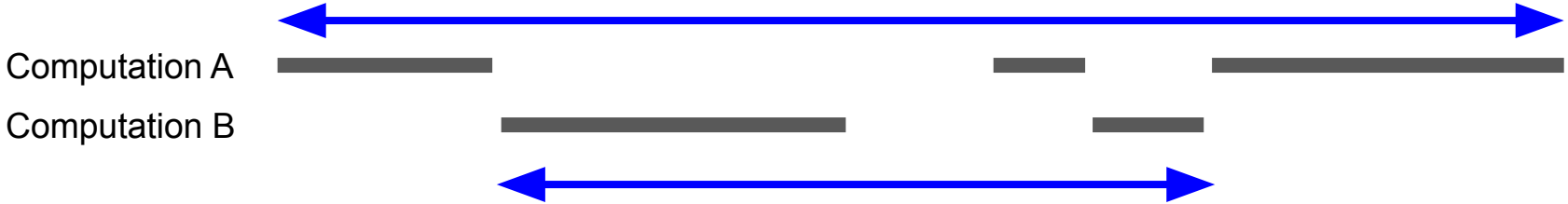
Today's Lecture

- Concurrency
  - Recap
  - Data lifetime issues
  - Mutexes
  - Atomics
- Revision
  - Files

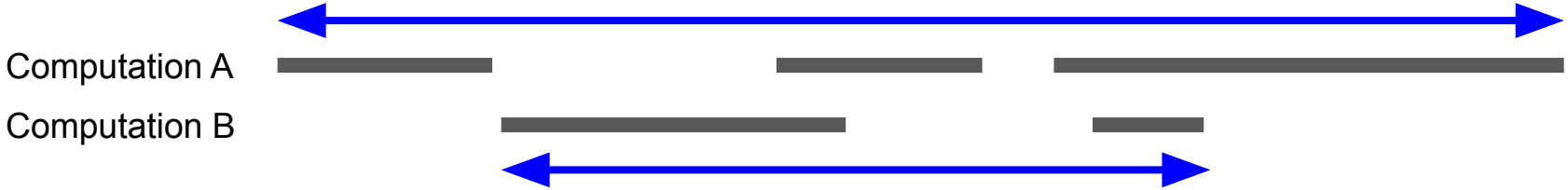


Concurrency & Parallelism

Concurrency: Multiple computations with *overlapping* time periods. Does not *have* to be simultaneous.



Parallelism: Multiple computations executing *simultaneously*.



Threads: concurrency *within* a process

- Threads allows us to create concurrency ***within*** a process
- Threads within a process ***share*** the **address space**:
  - threads share code
  - threads share global variables
  - threads share the heap (`malloc`)
  - cheap communication!
- Some other process state is shared
  - environment variables, file descriptors, current working directory, …

Threads: concurrency *within* a process

- Threads allows us to create concurrency ***within*** a process
- Each thread has a separate execution state
  - Separate **registers**, separate **program counter**
- Each thread has a **separate stack**
  - but a thread can still read/write to another thread's stack
- Each thread gets its own copy of errno!

Creating threads with `pthread_create`

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);
```

- Starts a new thread running `start_routine(arg)`
- An ID for the thread is stored in `thread`
- Thread has attributes specified in `attr` (NULL if you don't want special attributes)
- Returns 0 if OK, otherwise an error number (**does not set errno!**)
- Analogous to ***posix_spawn***.

Waiting for threads with `pthread_join`

```
int pthread_join(pthread_t thread, void **retval);
```

- Waits for `thread` to terminate, if it hasn't already terminated
- Return/exit value of thread placed in `*retval`
- Analogous to `waitpid`

- When **main** returns, *all* threads terminate

Examples

two_threads.c (recap example)

threads_return.c

threads_of_threads.c

Data Lifetime Issues

- When sharing data with a thread we pass in **addresses of data**
  - What if by the time the thread reads the data, that data no longer exists?
- So far we have put data in local variables in main
  - Main outlives all of the created threads
- What if we create threads from functions other than main?

- Demo: thread_data_broken.c
- Demo: thread_data_malloc.c

Data Races, Deadlock and Disasters

Unsafe Access to Global Variables

Demo: bank_account_broken.c

Incrementing a global variable is **NOT** an atomic operation

```c
int bank_account;

void *thread(void *a) {
    // ...
    bank_account++;
    // ...
```

```
la   $t0, bank_account
lw   $t1, ($t0)
addi $t1, $t1, 1
sw   $t1, ($t0)
.data
bank_account:  .word 0
```

Global Variables and Race Condition

If bank_account = 42 and two threads execute concurrently

```
la      $t0, bank_account
# {| bank_account = 42 |}
lw      $t1, ($t0)
# {| $t1 = 42 |}
addi    $t1, $t1, 1
# {| $t1 = 43 |}
sw      $t1, ($t0)
# {| bank_account = 43 |}
```

```
la      $t0, bank_account
# {| bank_account = 42 |}
lw      $t1, ($t0)
# {| $t1 = 42 |}
addi    $t1, $t1, 1
# {| $t1 = 43 |}
sw      $t1, ($t0)
# {| bank_account = 43 |}
```

Oops! We lost an increment.
Threads share global variables!

Global Variables and Race Condition

If bank_account = 100 and two threads execute concurrently

```
la      $t0, bank_account
# {| bank_account = 100 |}
lw      $t1, ($t0)
# {| $t1 = 100 |}
addi    $t1, $t1, 100
# {| $t1 = 200 |}
sw      $t1, ($t0)
# {| bank_account = ...? |}
```

```
la      $t0, bank_account
# {| bank_account = 100 |}
lw      $t1, ($t0)
# {| $t1 = 100 |}
addi    $t1, $t1, -50
# {| $t1 = 50 |}
sw      $t1, ($t0)
# {| bank_account = 50 or 200 |}
```

- This is a critical section.
- We don’t want two threads in the critical section
  - We must establish mutual exclusion.

A solution: mutexes

- We need a way of guaranteeing *mutual exclusion* for certain shared resources (such as ***bank_account***)
- We associate each of those resources with a ***mutex***
- Only one thread can hold a mutex, any other threads which attempt to lock the mutex must wait until the mutex is unlocked
- So only one thread will be executing the section between the mutex lock and the mutex unlock

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

bank_account_mutex.c

```
int bank_account=0;
pthread_mutex_t bank_account_lock=PTHREAD_MUTEX_INITIALIZER;

void *add_100000(void*argument){
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&bank_account_lock);
        // only one thread can execute this
        // section of code at any time
        bank_account = bank_account + 1;
        pthread_mutex_unlock(&bank_account_lock);
    }
```

Mutex the world!

- Mutexes solve all our data race problems!
- So… just put a mutex around everything!
- This works… but then we lose the advantages of parallelism
- Mutexes also have overhead
- Python 🐍 does this
    - Global Interpreter Lock (GIL) (although can be disabled now…)
- Linux used to do this (they removed the ‘Big Kernel Lock’ in 2011)

Code Demo: Deadlock

bank_account_deadlock.c

Deadlocks

- No thread can make progress!
- The system is deadlocked

THREAD 1

1. acquire lock_A
2. acquire lock_B ❌ **BLOCKED!**
3. do_somthing(A, B)
4. release lock_B
5. release lock_A

THREAD 2

1. acquire lock_B
2. acquire lock_A ❌ **BLOCKED!**
3. do_somthing(A, B)
4. release lock_A
5. release lock_B

`lock_A` 🔒

`lock_A` 🔓

`lock_B` 🔒

`lock_B` 🔓

This slide has animations, use the 'slideshow' button to view it.

Solving deadlocks

- A simple rule to avoid deadlocks:
  - All thread *must* acquire locks in the same order
  - (also good if locks are released in reverse order, if possible)
- e.g., always acquire lock_A before lock_B

THREAD 1

1. acquire lock_A
2. acquire lock_B
3. do_somthing(A, B)
4. release lock_B
5. release lock_A

THREAD 2

1. acquire lock_A
2. acquire lock_B
3. do_somthing(A, B)
4. release lock_B
5. release lock_A

Atomics

- With hardware support, we can avoid data races without needing to use locks!
- In C, we can use **atomic types**, which guarantee that certain operations using them will be performed *atomically* (indivisibly) ⇒ no data race!
- Also avoids overhead of mutexes
- And since no locks are involved, we can't introduce deadlock
- Atomics don't solve all concurrency problems
- There are still some subtle problems (which we don't cover in COMP1521)

Atomics

- Declaring an **atomic variable**
  - `atomic_int x = 10;`
  - `x += 1;` // Will be done atomically
  - `x = x + 1;` //Will **NOT** be done atomically!!!!
- Works with
  - ++, +=, - -, -=
  - |=, &=, ^=

Atomics

- Declaring an **atomic variable**
  - `atomic_int x = 10;`
  - `x += 1;` // Will be done atomically
  - `x = x + 1;` //Will **NOT** be done atomically!!!!
- A subset of functions in **stdatomic.h**:
  - atomic_fetch_add
    - `atomic_int x = 10;`
    - `int old = atomic_fetch_add(&x, 1);`
  - atomic_fetch_sub
  - atomic_fetch_or, atomic_fetch_xor, atomic_fetch_and

Add code with atomic in it

```c
atomic_int bank_account = 0;


void *add_100000(void *argument) {
    for (int i = 0; i < 100000; i++) {
        // NOTE: This *cannot* be `bank_account = bank_account + 1`,
        // as that will not be atomic!
        // However, `bank_account++` would be okay
        // `atomic_fetch_add(&bank_account, 1)` would also be okay
        bank_account += 1;
    }
```

Concurrency is really complex!

- This is just a taste of concurrency!
- Other fun concurrency problems/concepts: livelock, starvation, thundering herd, memory ordering, semaphores, software transactional memory, user threads, fibers, etc.

- A number of courses at UNSW offer more:
  - COMP3231/COMP3891: [Extended] operating systems
  - COMP6991: Solving Modern Programming Problems with Rust
  - … and more!

Revision

- File revision exercises
  - contains_byte_stdio.c
  - contains_byte_libc.c
  - over_write_last_byte_stdio.c
  - over_write_last_byte_libc.c
  - print_to_start.c
  - check_metadata.c
  - directory_search.c
  - recursive_directory_search.c

What we learnt Today

- Concurrency and threads
  - Recap
  - Data lifetime issues, Mutexes, Atomics
- Revision
  - Files
    - Basics with stdio
    - Basics with libc wrappers
    - stat
    - opendir and readdir

Next Lecture

Find out about the Final Exam!

More revision. Let me know what topics you want!

Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/ttMzsAC9b6

Reach Out

Content Related Questions:

Forum

Admin related Questions email:

cs1521@cse.unsw.edu.au

Student Support | I Need Help With…

My Feelings and Mental Health
Managing Low Mood, Unusual Feelings & Depression

Mental Health Connect
student.unsw.edu.au/**counselling**
Telehealth

Mind HUB
student.unsw.edu.au/**mind-hub**
Online Self-Help Resources

In Australia Call Afterhours UNSW Mental Health Support Line
1300 787 026
5pm-9am

Outside Australia Afterhours 24-hour Medibank Hotline
+61 (2) 8905 0307

Uni and Life Pressures
Stress, Financial, Visas, Accommodation & More

Student Support
Indigenous Student Support
student.unsw.edu.au/**advisors**

Reporting Sexual Assault/Harassment

Equity Diversity and Inclusion (EDI)
edi.unsw.edu.au/**sexual-misconduct**

Educational Adjustments
To Manage my Studies and Disability / Health Condition

Equitable Learning Service (ELS)
student.unsw.edu.au/**els**

Academic and Study Skills

Academic Language Skills
student.unsw.edu.au/**skills**

Special Consideration
Because Life Impacts our Studies and Exams

Special Consideration
student.unsw.edu.au/**special-consideration**