COMP1521 25T1

Week 9 Lecture 2

Concurrency, Parallelism and Threads

Adapted from Xavier Cooney,
Andrew Taylor and John Shepherd's slides

Announcements

Weekly test 8 due tomorrow
Extra help sessions and catch up classes

COMP1521 25T1 — COMP1521 Help Sessions

Exam Preferences form:

https://cgi.cse.unsw.edu.au/~exam/25T1/seating/register.cgi

Assignment 2: Get started ASAP if you have not already.

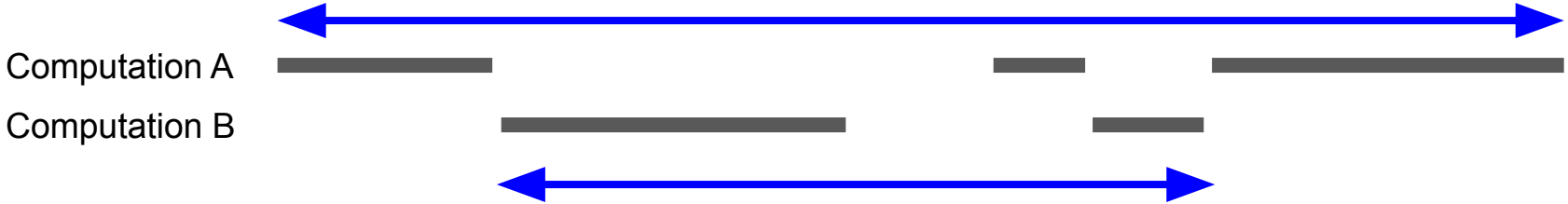
- Help sessions and forums will be very BUSY soon…
- You still have a chance to get help in tut/labs now too!

Today's Lecture

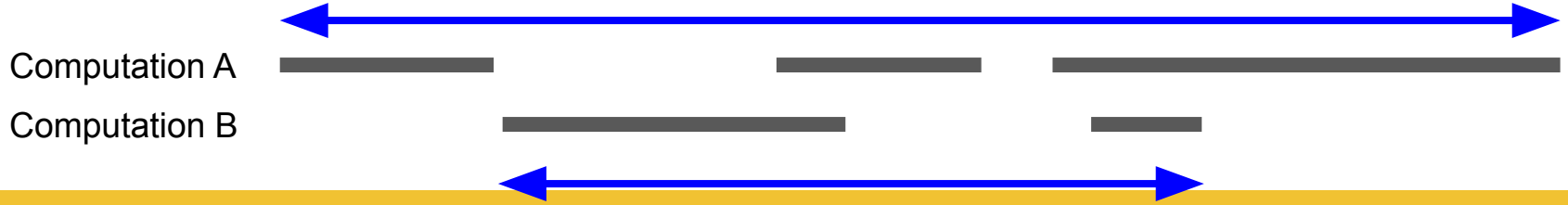
- Concurrency
- Threads
- Mutexes
- Atomics

Concurrency & Parallelism

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



Parallelism: Multiple computations executing *simultaneously*.



Question

Have we already seen concurrency in this course?

What about parallelism?

Very High-Speed Computing Systems

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—Very high-speed computers may be classified as follows:

1) Single Instruction Stream–Single Data Stream (SISD)
2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream–Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD).

"Stream," as used here, refers to the sequence of data or instructions as seen by the machine during the execution of a program.

The constituents of a system: storage, execution, and instruction handling (branching) are discussed with regard to recent developments and/or systems limitations. The constituents are discussed in terms of concurrent SISD systems (CDC 6600 series and, in particular, IBM Model 90 series), since multiple stream organizations usually do not require any more elaborate components.

Representative organizations are selected from each class and the arrangement of the constituents is shown.

Manuscript received June 30, 1966; revised August 16, 1966. This work was performed under the auspices of the U. S. Atomic Energy Commission.

The author is with Northwestern University, Evanston, Ill., and Argonne National Laboratory, Argonne, Ill.

INTRODUCTION

MANY SIGNIFICANT scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to input/output requirements (a partially de facto situation

Flynn's Taxonomy for Classifying Parallelism

SISD: Single Instruction, Single Data ("no parallelism")

- e.g. mipsy

SIMD: Single Instruction, Multiple Data ("vector processing")

- Multiple cores of a CPU executing (parts of) same instruction
- e.g. GPUs (graphics rendering and training and running neural networks e.g. LLMs)

MISD: Multiple Instruction, Single Data

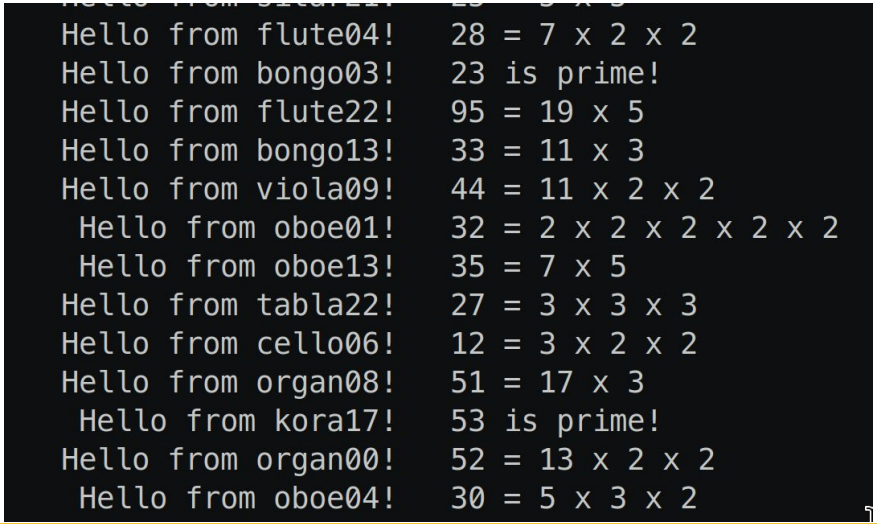
- e.g., fault tolerance in space shuttles (task replication)

MIMD: Multiple Instruction, Multiple Data ("multiprocessing")

- Multiple cores of a CPU executing different instructions

Parallel computing

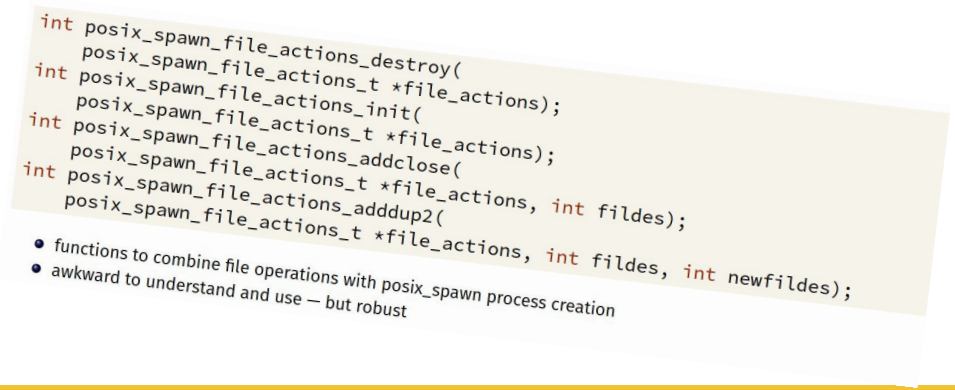
- Distributing computation across multiple computers
  - One popular framework is MapReduce
  - Necessary for *very big* computations and *very large* sets of data
  - Can be difficult to deal with synchronisation and failure of machines
  - Out of scope for COMP1521



Concurrency with processes

- Create multiple processes, and split the job across them
- Each process
  - runs concurrently
  - has its own address space (giving **isolation**)
- Processes can be distributed across cores, giving parallelism
- But this strategy is expensive!
  - Creation/teardown expensive
  - Switching expensive
  - Lots of state per process

    ⇒ costs memory
  - Communication can be complicated and expensive



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 or errno!

Using POSIX Threads (pthreads)

- POSIX Threads is a widely-supported threading model
- Provides an API/model for managing threads (and synchronisation)

```c
#include <pthread.h>
```

- Sometimes need **`-pthread`** when compiling
- C11 and later also provides a model/API similar to pthreads
  - Has some small differences with pthreads, and generally less-supported and less used (for now…)

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***.

Examples

- one_thread_infinite_loops.c
- one_thread_infinite_loops_my_puts.c
- one_thread.c

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

Some examples

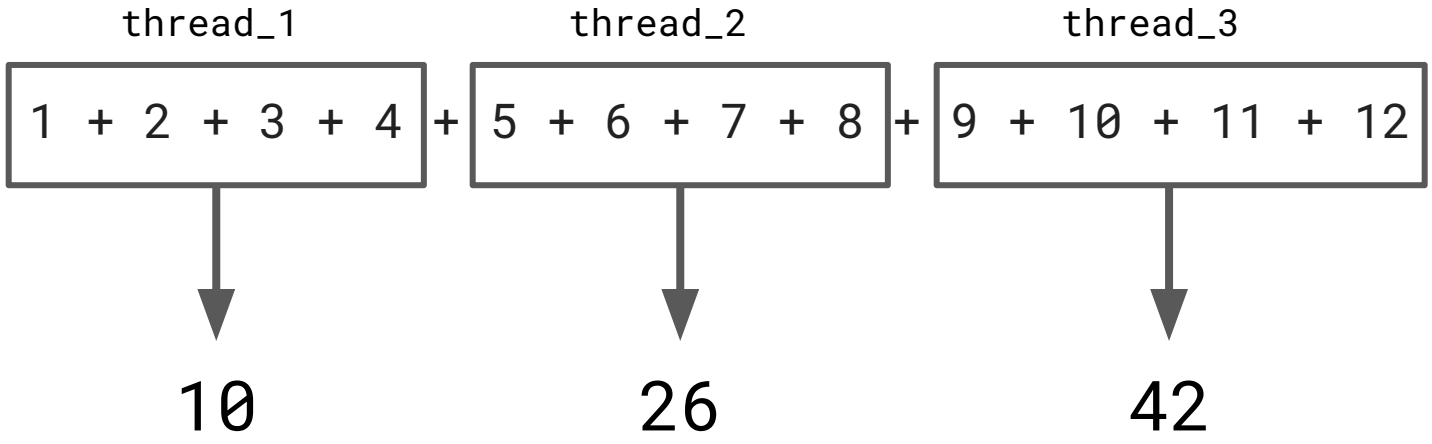
- two_threads_broken.c
- two_threads.c
- nthreads.c
- threads_return.c

Something Useful with Threads!

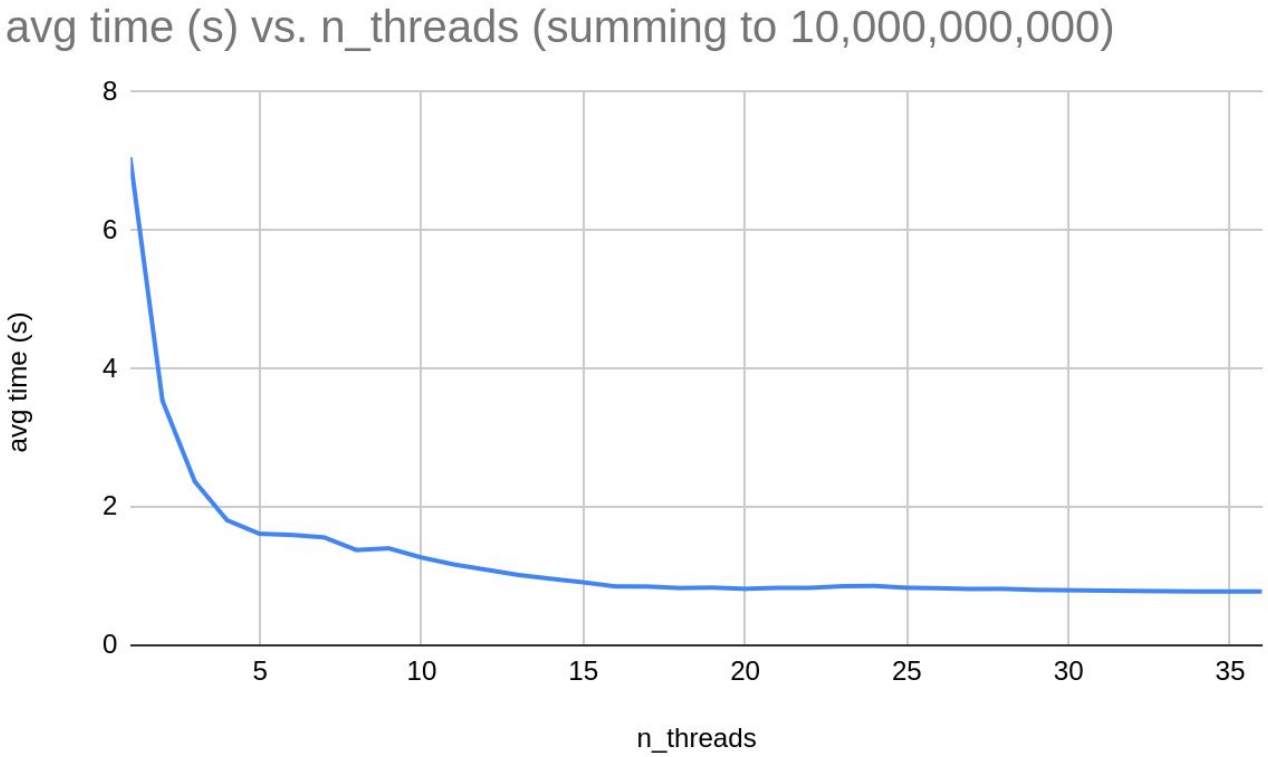
naive_sum.c

thread_sum.c

Example thread_sum



A graph of the performance of thread_sum.c



Some other concurrency benefits

- One thread can wait for I/O (block) while others make progress or wait for separate I/O
- Useful for user interface programming

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

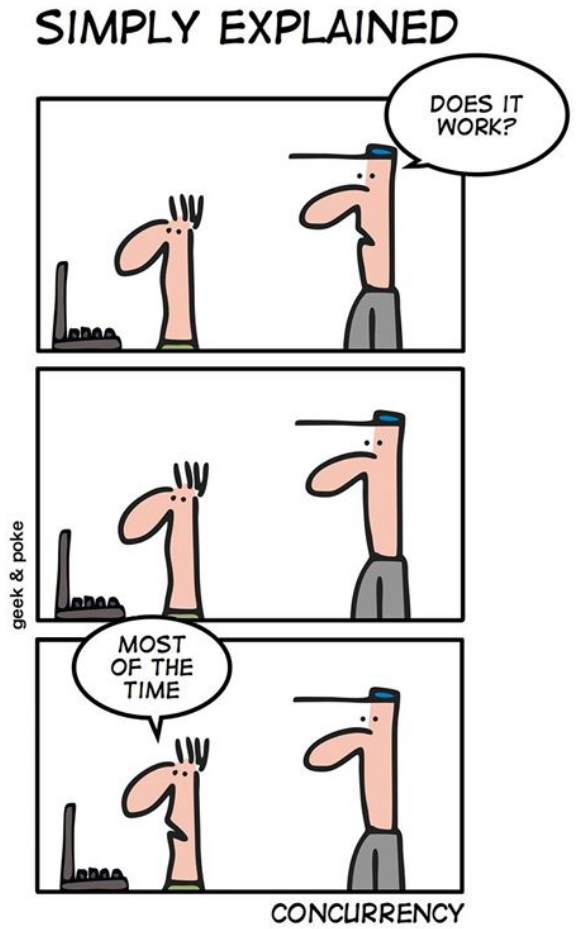
```
int bank_account=0;
pthread_mutex_t bank_account_lock=PTHREAD_MUTEX_INITIALIZER;

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 they're trying to stop…)
- 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!!!!
- 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

```
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
  - COMP3151: Foundations of Concurrency
  - COMP6991: Solving Modern Programming Problems with Rust
  - … and more!

What we learnt Today

- Concurrency
- Threads
- Data lifetime issues, Data races, deadlocks
- Mutexes, atomics

Next Lecture

- Pre-recorded:
  - Finish concurrency and threads if needed
  - Revision coding examples
    - Files, directories
    - Processes

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/z2D9Rsm9yH

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

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**