# COMP1521 23T3 — Concurrency, Parallelism, Threads

https://www.cse.unsw.edu.au/~cs1521/23T3/

## Concurrency + Parallelism

- Concurrency vs Parallelism
- Flynn's taxonomy
- Threads in C
- What can go wrong?
- Synchronisation with mutexes
- What can still go wrong?
- Atomics
- Lifetimes + Thread barriers

## Concurrency? Parallelism?

Concurrency:
multiple computations in overlapping time periods …
does *not* have to be simultaneous

Parallelism:
multiple computations executing *simultaneously*

Common classifications of types of parallelism (Flynn's taxonomy):

- SISD: Single Instruction, Single Data ("no parallelism")
  - e.g. our code in `mipsy`
- SIMD: Single Instruction, Multiple Data ("vector processing"):
  - multiple cores of a CPU executing (parts of) same instruction
  - e.g., GPUs rendering pixels
- MISD: Multiple Instruction, Single Data ("pipelining"):
  - data flows through multiple instructions; very rare in the real world
  - e.g., fault tolerance in space shuttles (task replication), sometimes A.I.
- MIMD: Multiple Instruction, Multiple Data ("multiprocessing")
  - multiple cores of a CPU executing different instructions

## Distributed Parallel Computing: Parallelism Across Many Computers

Parallelism can also occur between multiple computers!

Example: Map-Reduce is a popular programming model for

- manipulating *very large* data sets
- on a large network of computers — local or distributed
  - spread across a rack, data center or even across continents

The *map* step filters data and distributes it to nodes

- data distributed as (key, value) pairs
- each node receives a set of pairs with common key

Nodes then perform calculation on received data items.

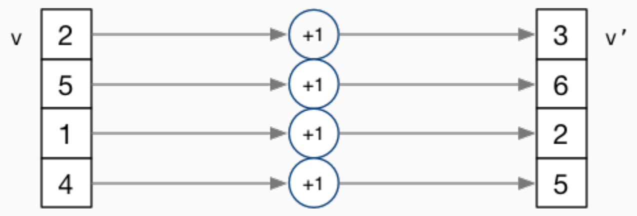
The *reduce* step computes the final result

- by combining outputs (calculation results) from the nodes

There also needs a way to determine when all calculations completed.

## Data Parallel Computing: Parallelism Across An Array

- multiple, identical processors
- each given one element of a data structure from main memory
- each performing same computation on that element: SIMD
- results copied back to data structure in main memory



- But not totally independent: need to *synchronise* on completion
- Graphics processing units (GPUs) provide this form of parallelism
  - used to compute the same calculation for every pixel in an image quickly
  - popularity of computer gaming has driven availablity of powerful hardware
  - there are tools & libraries to run some general-purpose programs on GPUs
  - if the algorithm fits this model, it might run 5-10x faster on a GPU
  - e.g., GPUs used heavily for neural network training (deep learning)

## Parallelism Across Processes

One method for creating parallelism:
create multiple processes, each doing part of a job.

- child executes concurrently with parent
- runs in its own address space
- inherits some state information from parent, e.g. open fd's

Processes have some disadvantages:

- process switching is *expensive*
- each require a *significant* amount of state — memory usage
- communication between processes potentially limited and/or slow

But one big advantage:

- separate address spaces make processes more robust.

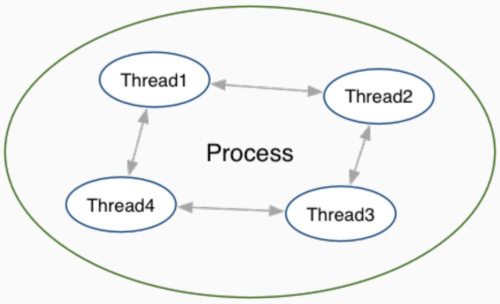
The web server providing the class website uses process-level parallelism

An android phone will have several hundred processes running.

## Threads: Parallelism within Processes

Threads allow us parallelism *within* a process.

- Threads allow *simultaneous* execution.
- Each thread has its own execution state
  often called Thread control block (TCB).
- Threads within a process *share* address space:
  - threads share code: functions
  - threads share global/static variables
  - threads share heap: `malloc`
- But a *separate* stack for each thread:
  - local variables *not* shared
- Threads in a process share file descriptors, signals.



## Threading with POSIX Threads (pthreads)

POSIX Threads is a widely-supported threading model.
supported in most Unix-like operating systems, and beyond

Describes an API/model for managing threads (and synchronisation).

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

More recently, ISO C:2011 has adopted a pthreads-like model...
less well-supported generally, but very, very similar.

## *pthread_create(3)*: create a new thread

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

- Starts a new thread running the specified `thread_main(arg)`.
- Information about newly-created thread stored in `thread`.
- Thread has attributes specified in `attr` (`NULL` if you want no special attributes).
- Returns 0 if OK, -1 otherwise and sets **`errno`**
- analogous to *posix_spawn(3)*

## *pthread_join(3)*: wait for, and join with, a terminated thread

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

- waits until `thread` terminates
    - if `thread` already exited, does not wait
- thread return/exit value placed in `*retval`
- if `main` returns, or *exit(3)* called, *all* threads terminated
    - program typically needs to wait for all threads before exiting
- analogous to ***waitpid(3)***

## *pthread_exit(3)*: terminate calling thread

```
void pthread_exit (void *retval);
```

- terminates the execution of the current thread (and frees its resources)
- `retval` returned — see *pthread_join(3)*
- analagous to ***exit(3)***

## Example: `two_threads.c` — creating two threads #1

```c
#include <pthread.h>
#include <stdio.h>
// This function is called to start thread execution.
// It can be given any pointer as an argument.
void *run_thread(void *argument) {
    int *p = argument;
    for (int i = 0; i < 10; i++) {
        printf("Hello this is thread #%d: i=%d\n", *p, i);
    }
    // A thread finishes when either the thread's start function
    // returns, or the thread calls `pthread_exit(3)'.
    // A thread can return a pointer of any type --- that pointer
    // can be fetched via `pthread_join(3)'
    return NULL;
}
```

source code for two_threads.c

## Example: `two_threads.c` — creating two threads #2

```c
int main(void) {
    // Create two threads running the same task, but different inputs.
    pthread_t thread_id1;
    int thread_number1 = 1;
    pthread_create(&thread_id1, NULL, run_thread, &thread_number1);
    pthread_t thread_id2;
    int thread_number2 = 2;
    pthread_create(&thread_id2, NULL, run_thread, &thread_number2);
    // Wait for the 2 threads to finish.
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);
    return 0;
}
```

source code for two_threads.c

## Example: `n_threads.c` — creating many threads

```c
    int n_threads = strtol(argv[1], NULL, 0);
    assert(0 < n_threads && n_threads < 100);
    pthread_t thread_id[n_threads];
    int argument[n_threads];
    for (int i = 0; i < n_threads; i++) {
        argument[i] = i;
        pthread_create(&thread_id[i], NULL, run_thread, &argument[i]);
    }
    // Wait for the threads to finish
    for (int i = 0; i < n_threads; i++) {
        pthread_join(thread_id[i], NULL);
    }
    return 0;
}
```

source code for n_threads.c

## Example: `thread_sum.c` — dividing a task between threads (i)

```c
struct job {
    long start, finish;
    double sum;
};
void *run_thread(void *argument) {
    struct job *j = argument;
    long start = j->start;
    long finish = j->finish;
    double sum = 0;
    for (long i = start; i < finish; i++) {
        sum += i;
    }
    j->sum = sum;
```

source code for thread_sum.c

## Example: `thread_sum.c` — dividing a task between threads (ii)

```c
printf("Creating %d threads to sum the first %lu integers\n"
       "Each thread will sum %lu integers\n",
       n_threads, integers_to_sum, integers_per_thread);
pthread_t thread_id[n_threads];
struct job jobs[n_threads];
for (int i = 0; i < n_threads; i++) {
    jobs[i].start = i * integers_per_thread;
    jobs[i].finish = jobs[i].start + integers_per_thread;
    if (jobs[i].finish > integers_to_sum) {
        jobs[i].finish = integers_to_sum;
    }
    // create a thread which will sum integers_per_thread integers
    pthread_create(&thread_id[i], NULL, run_thread, &jobs[i]);
}
```

source code for thread_sum.c

## Example: `thread_sum.c` — dividing a task between threads (iii)

```c
double overall_sum = 0;
for (int i = 0; i < n_threads; i++) {
    pthread_join(thread_id[i], NULL);
    overall_sum += jobs[i].sum;
}
printf("\nCombined sum of integers 0 to %lu is %.0f\n", integers_to_sum,
       overall_sum);
return 0;
```

source code for thread_sum.c

## `thread_sum.c` performance

Seconds to sum the first 1e+10 (10,000,000,000) integers using double arithmetic, with $N$ threads, on some different machines...

| host | 1 | 2 | 4 | 12 | 24 | 50 | 500 |
|---|---|---|---|---|---|---|---|
| *5800X* | 6.6 | 3.3 | 1.6 | 0.8 | 0.6 | 0.6 | 0.6 |
| *3900X* | 6.9 | 3.6 | 1.8 | 0.6 | 0.3 | 0.3 | 0.3 |
| *i5-4590* | 8.6 | 4.3 | 2.2 | 2.2 | 2.2 | 2.2 | 2.2 |
| *E7330* | 12.9 | 6.3 | 3.2 | 1.0 | 0.9 | 0.9 | 0.8 |
| *IIIi* | 136.6 | 68.4 | 68.6 | 68.4 | 68.5 | 68.6 | 68.6 |

*5800X*: AMD Ryzen 5800X; 8 cores, 16 threads, 3.8 GHz, 2020

*3900X*: AMD Ryzen 3900X; 12 cores, 24 threads, 3.8 GHz, 2019

*i5-4590*: Intel Core i5-4590; 4 cores, 4 threads, 3.3 GHz, 2014

*E7330*: Intel Xeon E7330; 4 sockets, 4 cores, 4 threads, 2.4 GHz, 2007

*IIIi*: Sun UltraSPARC IIIi; 2 sockets, 1 core, 1 thread, 1.5 GHz, 2003

## Example: `two_threads_broken.c` — shared mutable state gonna hurt you

```c
int main(void) {
    pthread_t thread_id1;
    int thread_number = 1;
    pthread_create(&thread_id1, NULL, run_thread, &thread_number);
    thread_number = 2;
    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, run_thread, &thread_number);
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);
    return 0;
}
```

source code for two_threads_broken.c

- variable `thread_number` will probably change in `main`, *before* thread 1 starts executing...
- $\Longrightarrow$ thread 1 will probably print **`Hello this is thread 2`** ... ?!

## Example: `bank_account_broken.c` — unsafe access to global variables (i)

```c
int bank_account = 0;
// add $1 to Andrew's bank account 100,000 times
void *add_100000(void *argument) {
    for (int i = 0; i < 100000; i++) {
        // execution may switch threads in middle of assignment
        // between load of variable value
        // and store of new variable value
        // changes other thread makes to variable will be lost
        nanosleep(&(struct timespec){ .tv_nsec = 1 }, NULL);
        // RECALL: shorthand for `bank_account = bank_account + 1`
        bank_account++;
    }
    return NULL;
}
```

source code for bank_account_broken.c

## Example: `bank_account_broken.c` — unsafe access to global variables (ii)

```c
int main(void) {
    // create two threads performing the same task
    pthread_t thread_id1;
    pthread_create(&thread_id1, NULL, add_100000, NULL);
    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, add_100000, NULL);
    // wait for the 2 threads to finish
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);
    // will probably be much less than $200000
    printf("Andrew's bank account has $%d\n", bank_account);
    return 0;
}
```

source code for bank_account_broken.c

## Global Variables and Race Conditions

Incrementing a global variable is not an *atomic* operation.

- (*atomic*, from Greek — “indivisible”)

```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, initially, `bank_account = 42`, and two threads increment simultaneously…

```
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 do not share registers or stack (local variables)…
but they *do* share global variables.

## Global Variable: Race Condition

If, initially, `bank_account = 100`, and two threads change it simultaneously…

```
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 processes in the critical section — we must establish *mutual exclusion*.

## *pthread_mutex_lock(3), pthread_mutex_unlock(3)*: Mutual Exclusion

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

- We associate a *mutex* with the resource we want to protect.
  - in the case the resources is access to a global variable
- For a particular mutex, only one thread can be running between `_lock` and `_unlock`
- Other threads attempting to `pthread_mutex_lock` will block (wait) until the first thread executes `pthread_mutex_unlock`

For example:

```
    pthread_mutex_lock (&bank_account_lock);
    andrews_bank_account += 1000000;
    pthread_mutex_unlock (&bank_account_lock);
```

## Example: `bank_account_mutex.c` — guard a global with a mutex

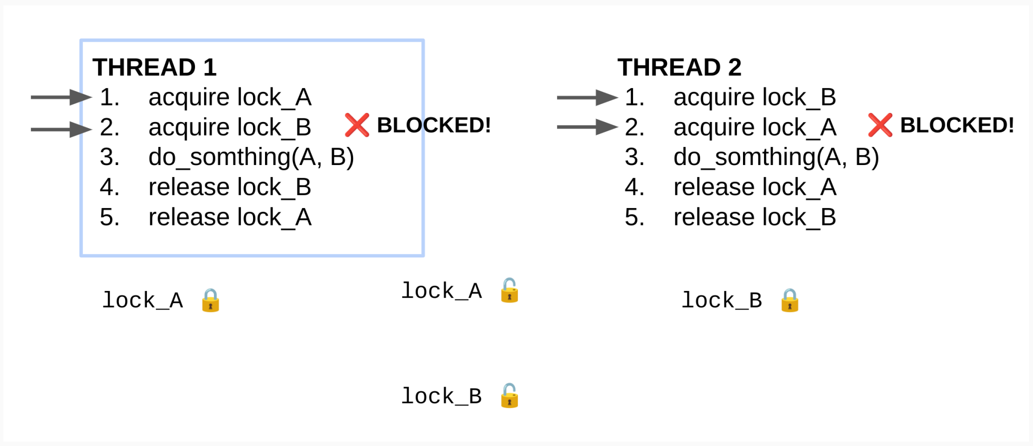
```
int bank_account = 0;
pthread_mutex_t bank_account_lock = PTHREAD_MUTEX_INITIALIZER;
// add $1 to Andrew's bank account 100,000 times
void *add_100000(void *argument) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&bank_account_lock);
        // only one thread can execute this section of code at any time
        bank_account = bank_account + 1;
        pthread_mutex_unlock(&bank_account_lock);
    }
    return NULL;
}
```

source code for bank_account_mutex.c

## 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
- Python does this - *the global interpreter lock* (GIL)
  - although they are (trying to stop)[https://peps.python.org/pep-0703/]
- Linux used to do this - the *Big Kernel Lock*
  - removed in 2011

## Deadlock



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

## Example: `bank_account_deadlock.c` — deadlock with two resources (i)

```c
void *andrew_send_xavier_money(void *argument) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&andrews_bank_account_lock);
        pthread_mutex_lock(&xaviers_bank_account_lock);
        if (andrews_bank_account > 0) {
            andrews_bank_account--;
            xaviers_bank_account++;
        }
        pthread_mutex_unlock(&xaviers_bank_account_lock);
        pthread_mutex_unlock(&andrews_bank_account_lock);
    }
    return NULL;
}
```

source code for bank_account_deadlock.c

## Example: `bank_account_deadlock.c` — deadlock with two resources (ii)

```c
void *xavier_send_andrew_money(void *argument) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&xaviers_bank_account_lock);
        pthread_mutex_lock(&andrews_bank_account_lock);
        if (xaviers_bank_account > 0) {
            xaviers_bank_account--;
            andrews_bank_account++;
        }
        pthread_mutex_unlock(&andrews_bank_account_lock);
        pthread_mutex_unlock(&xaviers_bank_account_lock);
    }
    return NULL;
}
```

source code for bank_account_deadlock.c

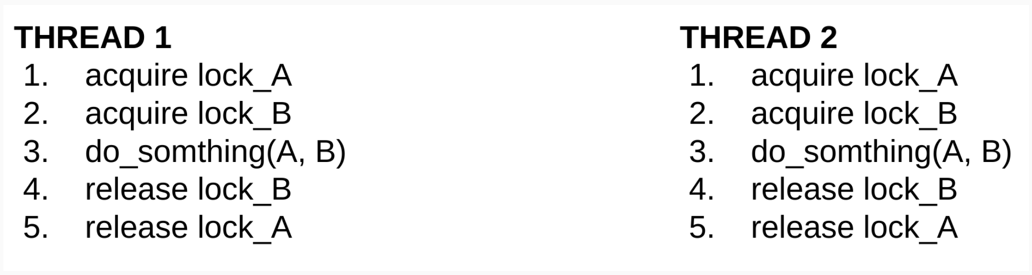
## Example: bank_account_deadlock.c — deadlock with two resources (iii)

```c
int main(void) {
    // create two threads sending each other money
    pthread_t thread_id1;
    pthread_create(&thread_id1, NULL, andrew_send_xavier_money, NULL);
    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, xavier_send_andrew_money, NULL);
    // threads will probably never finish
    // deadlock will likely likely occur
    // with one thread holding  andrews_bank_account_lock
    // and waiting for xaviers_bank_account_lock
    // and the other thread holding  xaviers_bank_account_lock
    // and waiting for andrews_bank_account_lock
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);
    return 0;
}
```

source code for bank_account_deadlock.c

## Avoiding Deadlock

- A simple rule can avoid deadlock in many programs
- All threads should acquire locks in same order
    - also best to release in reverse order (if possible)



**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

## Avoiding Deadlock

- Previous program deadlocked because one thread executed:

```c
pthread_mutex_lock(&andrews_bank_account_lock);
pthread_mutex_lock(&xaviers_bank_account_lock);
```

and the other thread executed:

```c
pthread_mutex_lock(&xaviers_bank_account_lock);
pthread_mutex_lock(&andrews_bank_account_lock);
```

- Deadlock avoided if same order used in both threads, e.g

## Atomics!

Atomic instructions allow a small subset of operations on data, that are guaranteed to execute atomically! For example,

| | |
|---|---|
| fetch_add: | `n += value` |
| fetch_sub: | `n -= value` |
| fetch_and: | `n &= value` |
| fetch_or: | `n |= value` |
| fetch_xor: | `n ^= value` |

*compare_exchange*:

```
if (n == v1) {
    n = v2;
}
return n;
```

Complete list: https://en.cppreference.com/w/c/atomic

## Atomics!

- With mutexes, a program can lock mutex A, and then (before unlocking A) lock some mutex B.
  - multiple mutexes can be locked simultaneously.
- Atomic instructions are (by definition!) atomic, so there's no equivalent to the above problem.
  - Goodbye deadlocks!
- Atomics are a fundamental tool for lock-free/wait-free programming.
- Non-blocking: If a thread fails or is suspended, it cannot cause failure or suspension of another thread.
- Lock-free: non-blocking + the system (as a whole) always makes progress.
- Wait-free: lock-free + every thread always makes progress.

## Example: `bank_account_atomic.c` — safe access to a global variable

```c
#include <stdatomic.h>
atomic_int bank_account = 0;
// add $1 to Andrew's bank account 100,000 times
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
        // and,     `atomic_fetch_add(&bank_account, 1)` would also be okay
        bank_account += 1;
    }
    return NULL;
}
```

source code for bank_account_atomic.c

## What’s the catch with atomics?

- Specialised hardware support is required
    - essentially all modern computers provide atomic support
    - may be missing on more niche / embedded systems.
- Although faster and simpler than traditional locking, there is still a performance penalty using atomics (and increases program complexity).
- Can be incredibly tricky to write correct code at a low level (e.g. memory ordering, which we won’t cover in COMP1521).
- Some issues can arise in application; e.g. ABA problem.

## 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 e.g more on deadlock
    - COMP3151: Foundations of Concurrency
    - COMP6991: Solving Modern Programming Problems with Rust - e e.g safety through types
    - and more!