Concurrency? Parallelism?

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

Parallelism:
multiple computations executing simultaneously

Parallel computations occur at different levels:

- **SIMD**: Single Instruction, Multiple Data (“vector processing”):
  - multiple cores of a CPU executing (parts of) same instruction
  - e.g., GPUs rendering pixels
- **MIMD**: Multiple Instruction, Multiple Data (“multiprocessing”)
  - multiple cores of a CPU executing different instructions
distributed: spread across computers
  - e.g., with MapReduce

Both parallelism and concurrency need to deal with *synchronisation*. 
Example: Map-Reduce is a popular programming model for

- manipulating very large data sets
- on a large network of computers — local or distributed

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

(This 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

Common use-case for GPUs, neural network processors, etc.
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.

(You’re probably using a process-parallel program right now!)
Threads allow parallelism within a process.

- Threads allow *simultaneous* execution.
- Each thread has its own execution state (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.
POSIX Threads is a widely-supported threading model. supported in most *nix-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

```c
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` (possibly NULL).  
- 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

```c
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**

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

- terminates the execution of the current thread (and frees its resources)
- `retval` returned — see `pthread_join(3)`
- analogous to `exit(3)`
#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
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;
}

```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;
```
```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

https://www.cse.unsw.edu.au/~cs1521/21T2/
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]);
}
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);
thread_sum.c performance

Summing the first $1e+10$ (10,000,000,000) integers, with $\mathcal{N}$ threads, on some different machines... 

<table>
<thead>
<tr>
<th>host</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>12</th>
<th>24</th>
<th>50</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceyx</td>
<td>6.9</td>
<td>3.6</td>
<td>1.8</td>
<td>0.6</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>lisbon</td>
<td>7.6</td>
<td>3.9</td>
<td>2.0</td>
<td>0.8</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
</tr>
</tbody>
</table>

ceyx: AMD Ryzen 3900X (12c/24t), 3.8 GHz  

lisbon: AMD Ryzen 4750U (8c/16t), 4.1 GHz
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;
}
```

- variable `thread_number` will probably change in `main`, before thread 1 starts executing...
- 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);
        bank_account = bank_account + 1;
    }
    return NULL;
}
```

source code for bank_account_broken.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;
}
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++;
    // ...
}
```

```assembly
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, \( \text{bank\_account} = 42 \), and two threads increment simultaneously...

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

```assembly
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...

This is a critical section.

We don’t want two processes in the critical section — we must establish mutual exclusion.
We associate a resource with a `mutex`.
For a particular mutex, only one thread can be running between `lock` and `unlock`.
Other threads attempting to `lock` will block.
(Other threads attempting to `trylock` will fail.)

For example:

```c
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

```c
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

https://www.cse.unsw.edu.au/~cs1521/21T2/
Semaphores are a more general synchronisation mechanism than mutexes.

```c
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

- `sem_init(3)` initialises `sem` to `value`.
- `sem_wait(3)` — classically P
  - if `sem > 0`, then `sem := sem - 1` and continue...
  - otherwise, `wait` until `sem > 0`
- `sem_post(3)` — classically V, also `signal`
  - `sem := sem + 1` and continue...
Example: Allow $n$ threads to access a resource

```c
#include <semaphore.h>
sem_t sem;
sem_init (&sem, 0, n);

sem_wait (&sem);
// only $n$ threads can be executing here simultaneously
sem_post (&sem);
```
Example: bank_account_sem.c: guard a global with a semaphore (i)

```c
sem_t bank_account_semaphore;
// add $1 to Andrew's bank account 100,000 times
void *add_100000 (void *argument) {
    for (int i = 0; i < 100000; i++) {
        // decrement bank_account_semaphore if > 0
        // otherwise wait until > 0
        sem_wait (&bank_account_semaphore);
        // only one thread can execute this section of code at any time
        // because bank_account_semaphore was initialized to 1
        bank_account = bank_account + 1;
        // increment bank_account_semaphore
        sem_post (&bank_account_semaphore);
    }
    return NULL;
}
```
int main (void)
{
    // initialize bank_account_semaphore to 1
    sem_init (&bank_account_semaphore, 0, 1);
    // 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 always be $200000
    printf ("Andrew's bank account has $%d\n", bank_account);
    sem_destroy (&bank_account_semaphore);
    return 0;
}
Concurrent Programming is Complex

Concurrency is *really complex* with many issues beyond this course:

- **Data races** thread behaviour depends on unpredictable ordering; can produce difficult bugs or security vulnerabilities
- **Deadlock** threads stopped because they are wait on each other
- **Livelock** threads running without making progress
- **Starvation** threads never getting to run
Example: bank_account_deadlock.c — deadlock with two resources (i)

```c
void *swap1 (void *argument)
{
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock (&bank_account1_lock);
        pthread_mutex_lock (&bank_account2_lock);
        int tmp = andrews_bank_account1;
        andrews_bank_account1 = andrews_bank_account2;
        andrews_bank_account2 = tmp;
        pthread_mutex_unlock (&bank_account2_lock);
        pthread_mutex_unlock (&bank_account1_lock);
    }
    return NULL;
}
```

source code for bank_account_deadlock.c
Example: bank_account_deadlock.c — deadlock with two resources (ii)

```c
void *swap2 (void *argument)
{
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock (&bank_account2_lock);
        pthread_mutex_lock (&bank_account1_lock);
        int tmp = andrews_bank_account1;
        andrews_bank_account1 = andrews_bank_account2;
        andrews_bank_account2 = tmp;
        pthread_mutex_unlock (&bank_account1_lock);
        pthread_mutex_unlock (&bank_account2_lock);
    }
    return NULL;
}
```

source code for bank_account_deadlock.c

https://www.cse.unsw.edu.au/~cs1521/21T2/
Example: bank_account_deadlock.c — deadlock with two resources (iii)

```c
int main (void)
{
    // create two threads performing almost the same task
    pthread_t thread_id1;
    pthread_create (&thread_id1, NULL, swap1, NULL);
    pthread_t thread_id2;
    pthread_create (&thread_id2, NULL, swap2, NULL);
    // threads will probably never finish
    // deadlock will likely occur
    // with one thread holding bank_account1_lock
    // and waiting for bank_account2_lock
    // and the other thread holding bank_account2_lock
    // and waiting for bank_account1_lock
    pthread_join (thread_id1, NULL);
    pthread_join (thread_id2, NULL);
    return 0;
}
```

source code for bank_account_deadlock.c

https://www.cse.unsw.edu.au/~cs1521/21T2/