Concurrency/Parallelism

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

Parallelism multiple computations executing simultaneously

Parallel computation occurs at different level:

- spread across computers (e.g., with MapReduce)
- multiple cores of a CPU executing different instructions (MIMD)
- multiple cores of a CPU executing same instruction (SIMD)
  - e.g. GPU rendering pixels

Both parallelism and concurrency need to deal with *synchronisation*.

Parallel Computing Across Many Computers

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*(s)

Nodes then perform calculation on received data items

The *reduce* step computes the final result

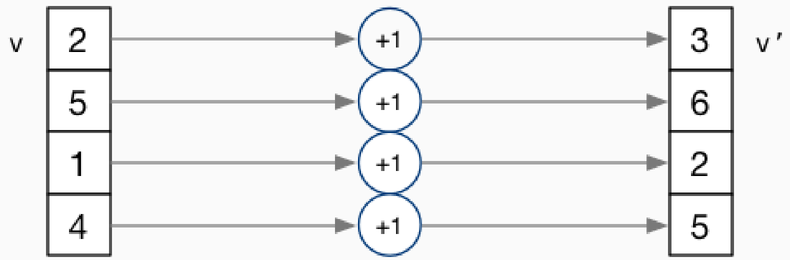
- by combining outputs (calculation results) from the nodes

Also needs a way to determine when all calculations completed

Parallelism Across a an Array

GPU rendering pixels

- 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 main memory data structure



But not totally independent: need to *synchronise* on completion

Parallelism Across Processes

One method for creating parallelism:

Use `posix_spawn()`to create multiple processes each of which does part of 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 expensive
- each require a significant amount of state (RAM)
- communication between processes limited and/or slow

Parallelism with Processes

threads - mechanism for parallelism within process.

- threads allow simultaneous execution within process
- each threads has its own execution state
- threads within a process spare address space:
  - threads share code (functions)
  - threads share global & static variables
  - threads share heap (`malloc`)
- but separate stack for each thread
  - local variables not shared
- threads share file descriptor
- threads share signals

POSIX threads (pThreads)

```
// POSIX threads widely supported in Unix-like
// and other systems (Windows).  Provides functions
// to create/synchronize/destroy/... threads

#include <pthread.h>
```


Create A POSIX Thread

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

- creates a new thread with specified **attributes** (can be `NULL`)
- thread info stored in **`*thread`**
- thread starts by executing **`start\_routine(arg)`**
- returns 0 if OK, -1 otherwise and sets **`errno`**
- analogous to **`posix\_spawn()`**

Wait for A POSIX Thread

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

- wait until **`thread`** terminates
- **`thread`** return (or **`pthread\_exit()`**) value is placed in **`*retval`**
- if **`thread`** has already exited, does not wait
- if main returns or exit called, all threads terminated
- so typically wait for all threads
- analogous to **`waitpid`**

Terminate A POSIX Thread

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

- terminate execution of thread (and free resources)
- **retval** is returned (see **pthread_join**)
- if **thread** has already exited, does not wait
- analagous to **exit**

Global Variable: Race Condition

Incremeing a global variable is not an atomic (indivisible) operation.

```
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 Variable: Race Condition

If **bank_account** == 42 and two threads increment simultaneously.

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

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

One increment is lost.

Note threads don't share registers or stack (local variable).

They do share global variables.

Global Variable: Race Condition

If **bank_account** == 100 and two threads change simultaneously.

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

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

Exclude Other Threads from Code

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

- only one thread can enter a *critical section*
- establishes mutual exclusion — *mutex*
- call **pthread_mutex_lock** before
- call **pthread_mutex_unlock** after
- only 1 thread can execute in

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

Semaphores are special variables which provide a more general synchronisation mechanism than mutexes.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,
             unsigned int value);
int sem_post(sem_t *sem);
```


Allow n threads access to a resource

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

sem_wait(&sem);
// only n threads can be in executing
// in here simultaneously
sem_post(&sem);
```


File Locking

```
int flock(int FileDesc, int Operation)
```

Similar to mutexes for a file.

- controls access to shared files (**note**: files not fds)
- possible operations
  - `LOCK_SH` ... acquire shared lock
  - `LOCK_EX` ... acquire exclusive lock
  - `LOCK_UN` ... unlock
  - `LOCK_NB` ... operation fails rather than blocking
- in blocking mode, `flock()` does not return until lock available
- only works correctly if all processes accessing file use locks
- return value: 0 in success, -1 on failure

File Locking

If a process tries to acquire a *shared lock* ...

- if file not locked or other shared locks, OK
- if file has exclusive lock, blocked

If a process tries to acquire an *exclusive lock* ...

- if file is not locked, OK
- if any locks (shared or exclusive) on file, blocked

If using a non-blocking lock

- `flock()` returns 0 if lock was acquired
- `flock()` returns -1 if process would have been blocked

Concurrent Programming is Complex

Concurrency is *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