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 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

Example: GPU rendering pixels or neural network

---

Parallelism Across Processes

One method for creating parallelism:

Use *posix_spawn()* to create multiple processes, each 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

One big advantage - separate address spaces make processes more robust.
Parallelism within Processes

- threads - mechanism for parallelism within process.
  - threads allow simultaneous execution within process
  - each thread has its own execution state
  - threads within a process have same 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 attributes specied in attr
  - attr 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
  - programs typically need to wait for all threads before main returns/exit called
  - 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
• analogous to exit

Simple example - Creating Two threads

#include <pthread.h>

// this function is called to start thread execution
// it can be given any pointer as argument (int *) in this example

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 the function returns or thread_exit
    // a pointer of any type can be returned
    // this can be obtained via thread_join's 2nd argument
    return NULL;
}

source code for two_threads.c

Classic Bug - Sharing a Variable Between Threads

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);

source code for two_threads_broken.c

• variable thread_number will probably have changed in main before thread 1
  starts executing
• so thread 1 will probably print Hello this is thread 2
**Simple example - Creating Many threads**

```c
int n_threads = strtol(argv[1], NULL, 0);
assert(n_threads > 0 && 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);
}
```

**Source code for n_threads.c**

---

**Simple example - Dividing a task between threads**

```c
struct job {
    long start;
    long 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**

---

**Simple example - Dividing a task between threads**

```c
printf("Creating %d threads to sum the first %lu integers\n",
    n_threads, integers_to_sum);
printf("Each thread will sum %lu integers\n", 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]);
}
// wait for the threads to finish
for (int i = 0; i < n_threads; i++) {
    pthread_join(thread_id[i], NULL);
    overall_sum += jobs[i].sum;
}
// Combined sum of integers 0 to %lu is %.0f\n",
integers_to_sum, overall_sum);
```

**Source code for thread_sum.c**
Simple example - Dividing a task between threads

double overall_sum = 0;
for (int i = 0; i < n_threads;i++) {
    pthread_join(thread_id[i], NULL);
    overall_sum += jobs[i].sum;
}

//
printf("Combined sum of integers 0 to %lu is %.0f
", integers_to_sum, overall_sum);

Example - Unsafe Access to Global Variable

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
", bank_account);
    return 0;
}

Example - Unsafe Access to Global Variable

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(&({.tv_nsec = 1}, NULL);
        bank_account = bank_account + 1;
    }
    return NULL;
}
### Global Variable: Race Condition

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

```c
int bank_account;

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

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

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

```asm
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.

### Exclude Other Threads from Code

```c
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 protected code
- for example:

```c
pthread_mutex_lock(&bank_account_lock);
andrews_bank_account += 1000000;
pthread_mutex_unlock(&bank_account_lock);
```
Example - Protecting Access to Global Variable 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
        // bank_account = bank_account + 1;
        pthread_mutex_unlock(&bank_account_lock);
    }
    return NULL;
}
```

source code for bank_account_mutex.c

Semaphores

Semaphores are special variables which provide 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` initialises `sem` to `value`
- `sem_wait` - classically called `P()`
  - if `sem > 0`, decrement `sem` and continue
  - otherwise, wait until `sem > 0`
- `sem_post` - classically called `V()`
  - increment `sem` and continue

Allow n threads access to a resource

```c
#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);
```

Protecting Access to Global Variable with a Semaphore

```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
        // because bank_account_semaphore was initialized to 1
        bank_account = bank_account + 1;
        // increment bank_account_semaphore
        sem_post(&bank_account_semaphore);
    }
    return NULL;
}
```

source code for bank_account_semaphore.c
Protecting Access to Global Variable with a Semaphore

```c
// 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);
```

File Locking

```c
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
```

Concurrent Programming is Complex

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

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
Example - deadlock accessing two resources

```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 - deadlock accessing two resources

```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

Example - deadlock accessing two resources

```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 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