SYS2(OS) Lab 5: Threads & Synchronisation

Dr Poonam Yadav and Sven Signer

POSIX Threads

Last week's lab session looked at creating new processes with the fork() system call, and sending simple signals to processes.

This practical looks at implementing multiple threads within a process using the POSIX pthreads library.

Unlike separate processes created with fork(), threads run in the same address space.

Unexpected behaviour can occur if access to shared memory is not carefully coordinated.

Thread Management

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

- Creates a new thread in the current process.
- Function pointer `start_routine` is invoked in the new thread with arguments `arg`.
- Thread ID is stored into `thread` argument.
- Function returns 0 on successful thread creation.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

- Makes the current thread wait for another thread (specified by `thread` argument) to complete.
- If `retval` is not null, the return value of the joined thread is stored into it.
- Function returns 0 on success.

Thread Creation Example

```
#include <pthread.h>

void *some_fn(void *arg) { … }

int main() {
    …
    p_thread tid;
    pthread_create(&tid, NULL, some_fn, arg);
    …
    pthread_join(tid, NULL);
    …
```

Function containing thread implementation

Create a new thread that executes the specified function.

Wait for thread completion.

Thread Synchronisation

- Threads operate in the same address space, meaning global memory can be accessed from different threads simultaneously.
- Without thread synchronisation, it is easy to introduce race conditions, where program behaviour depends on the random scheduling behaviour.
- Mutexes allow the program to eliminate race conditions by ensuring that only one thread at a time will execute in a critical region.

Synchronisation with Mutexes

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(phtread_mutex_t *mutex);
```

- Once initialised, mutexes can be locked/unlocked to ensure only a single thread executes in a critical region at any given time.
- As always, see the man pages for specifics on the arguments.

Mutex Example Program

```
#include <pthread.h>
pthread_mutex_t mutex;

void *thread_fn(void *param) {
    ...
    pthread_mutex_lock(&mutex);
    // <Critical Section>
    pthread_mutex_unlock(&mutex);
    ...
}

int main(int argc, char *argv[]) {
    pthread_mutex_init(&mutex, NULL);
    // <Launch Threads>
    pthread_mutex_destroy(&mutex);
}
```

Thread(s) acquire a mutex before a critical section to guarantee mutual exclusion.

Mutex must be initialised before it can be used, for example in the main function before launching worker threads.

Condition Variables

- While mutexes implement synchronisation by controlling thread access to data, condition variables allow threads to synchronise based upon the actual value of data.
- A program might need to acquire a lock (mutex) on some shared data, but might require this to be in a certain state before it can proceed.
- Condition variables allow program to efficiently guarantee mutual exclusion while testing some predicate.

Condition Variables

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

- Releases the mutex and waits for the condition variable to be signalled, then reacquires the mutex prior to continuing.
- Spurious wakeups are explicitly permitted, so typically included inside a while loop that re-checks some predicate.

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

- Signal a thread waiting on the condition variable to unblock.
- Has no effect if no thread is waiting on the condition.

Condition Variable Example

```
#include <pthread.h>

pthread_cond_t cond;
pthread_mutex_t mutex;
void *thread1(void *param);
void *thread2(void *param);

int main() {
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    // <Launch & Wait for Threads>

    pthread_mutex_destroy(&mutex);
    pthread_cond_destory(&cond);
}
```

Like mutexes, condition variables need to be initialised before they can be used.

Condition Variable Example

```
void *thread1(void *param) {
    ...
    pthread_mutex_lock(&mutex);
    while (!predicate) {
        pthread_cond_wait(&cond, &mutex);
    }

    // <Critical Section>

    pthread_mutex_unlock(&mutex);
    ...
}
```

Thread locks mutex, then waits for predicate to hold before executing the critical section. Condition variable handles releasing and reacquiring the mutex.

Condition Variable Example

```
void *thread2(void *param) {
    ...
    pthread_mutex_lock(&mutex);

    // <Critical Section>

    if (predicate) {
        pthread_cond_signal(&cond);
    }
    pthread_mutex_unlock(&mutex);
    ...
}
```

Other thread(s) check the predicate after completing their own critical section, signalling any blocked threads to continue if the predicate now holds.

Assignment

This lab session looks at thread creation and race conditions that can be caused by simultaneous access to global variables.

An example program `simple_threads.c` is provided on the VLE to get you started.