# CSSE2310 — 9.1

Threads and Synchronization continued

# Semaphores in action

See race3.c

- sem_init() — set the initial value of the semaphore
- sem_wait() — decrement the semaphore
- sem_post() — increment the semaphore
- sem_destroy() --- clean up

Note: Always pass pointers to semaphores, do not copy the value itself.

# Mutual exclusion

Task 1: mutex

- `sem_init()`: set value to 1
- `sem_wait()`: acquire the lock / mutex. Only one thread can succeed until post
- `sem_post()`: release the lock

Provided that all paths into the critical section(s)[1][2] require waiting and all paths out post, this will ensure mutual exclusion.

[1] No more than one thread should be in a critical section at a time.

[2] Different parts of a program could belong to the same critical section.

# Waiting

Task 2: Non-busy waiting

- `sem_init()`: set value to 0
- `sem_wait()`: block until semaphore is available.
- `sem_post()`: let other thread know “it” happend.

# Other things to do with semaphores

Limit maximum threads active (not as common)

- `sem_init()`: set value to $N$
- at most $N$ threads can pass (wait) until one or more threads leave (post).

Producer and consumer tasks

- `sem_init()`: set value to 0
- Each time the "producer" adds a job to the queue, `post`
- Consumer threads all `wait` on the semaphore.

Notes:

- in this case some threads only wait and other threads only post
- You still need a separate mutex to control accessing the queue.

# Volatility

Consider the following code:

```
total=0;
if (*a > 0) {
    total++;
}
if (*a > 0) {
    total++;
}
if (*a > 0) {
    total++;
}
// total is either 0 or 3?
```

# Volatility

Or:

```
total = 0;
ref = *a;
total += *a;
total += *a;
total += *a;
total += *a;
// total is 4 * ref?
```

The compiler won't see any reason that it can't optimise the calculation (and most of the time it would be correct).

- What if another thread also knows pointer *a*?
- ... and modifies it

# Volatility

The `volatile` keyword warns the compiler that the value of the variable may change in ways (and at times) when the compiler won't predict.

- ie. Don't get too clever with this one.

eg:

```
volatile int* p;
            // p is a pointer to a volatile int
volatile const int x=6;
            // yes this is legal
const int* const p=&global;
            // not volatile but still legal
```

# Not volatile

Use `volatile` when any variable may be modified by one thread and read in others.

```
void f(...) {
    int a=...;   // not volatile

}
```

Even if multiple threads call `f()` at the same time, they will each see a different (local) variable.

# thread safety

An operation is "thread safe"[3] if multiple threads can have active calls to the function at the same time.[4].

Things to look for:

- A value could be modified by one thread while another is using it.
  - This includes freeing and mallocing, removing entries from lists, ...
  - Can be tricky to spot (eg i++) if you are used to thinking of them as atomic

[3] You may also see this called "reentrant"

[4] Could also consider recursion or nested signal handlers here

# thread safety

Look for calls to non-threadsafe functions.

- How to tell? Look at the (up to date) doco.
- eg: `rand_r`
- `_r` normally indicates “reentrant”
- `man 3 rand_r`
- On moss, says `rand()` and `rand_r()` are both thread safe.
- (and that `rand_r` is deprecated).

Some functions make use of hiden `static` state which might not be obvious.

# Non-obvious non-atomics

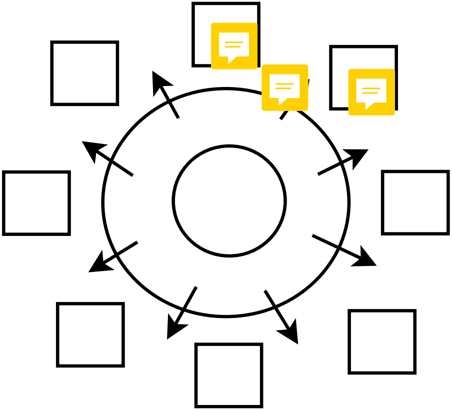
Suppose many threads execute the following at the same time:

```
*p = xn;
```

Thread 1 stores $x_1$, … Thread $n$ stores $x_n$.
What happens?

- There is a race condition where it is not known which thread will write last.
- Will *p store one of $\{x_1, \ldots, x_n\}$?
    - It depends on the size of $x_i$ vs the size of a processor word.
    - eg `long double` on moss.
    - long on 32bit systems?

# Dining philosophers



# Deadlock

Thread 1:

```
wait(l1);
wait(l2);
    // do something
post(l1);
post(l2);
```

Thread 2:

```
wait(l2);
wait(l1);
    // do something
post(l2);
post(l1);
```

- For simple cases: Everyone should request resources in the same order.
- More complex cases: beyond the scope of this course.

# Dining philosophers

Potential problems:

- Deadlock
- Livelock
- Starvation[5]

[5] Again beyond the scope of the course

# Misc pthreads

# fork() and threads

According to the documentation, calling fork() in a multithreaded progam only duplicates the thread which called fork().

See forknthread.c as an example.

You must also be careful of locks etc.

# semaphores

- `sem_trywait()`
  - Either lock immediately or error
- `sem_timedwait()`
  - Error if lock can't be aquired before timeout
  - Note: this function takes an absolute time not a delta.
- semaphores between processes? (Not in 2310).
  - `sem_create()` / `sem_destroy()` in shared memory
  - `sem_open()` / `sem_unlink()` named semaphores

# mutex and condition vars

pthreads also specifies:

- `pthread_mutex_t`
  - Only mutual exclusion
  - Can only unlock from the thread which locked
- `pthread_cond_t`
  - For waiting for something
  - Each condition var is linked to a mutex