COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/
Today

• Threads
• Mutexes
• Condition variables, semaphores
• Interprocess communication (IPC)
• Asynchronous IO
Threads

- `int pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start)(void*), void * arg)`
- `int pthread_join(pthread_t th, void ** retval)`
- `int pthread_detach(pthread_t th)`
Thread Termination

- void pthread_cleanup_push(void (*routine)(void*), void*arg)
- void pthread_cleanup_pop(int execute)
- void pthread_exit(void * retval)
Thread Cancellation

- `int pthread_cancel(pthread_t thread)`
- `void pthread_testcancel()`
Mutex

- `int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * mutexattr)`
- `int pthread_mutex_lock(pthread_mutex * mutex)`
- `int pthread_mutex_unlock(pthread_mutex * mutex)`
- `int pthread_mutex_destroy(pthread_mutex * mutex)`
Semaphores

(Counting) semaphores are a synchronization mechanism used for:

- Implementation of bounded queues
- Limiting number of concurrent activities in critical code
- Coordination between multiple threads or processes

A mutex is a binary semaphore.
Implementing Semaphores: struct

```c
struct Semaphore
{
    int v;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} Semaphore;
```
Implementing Semaphores: create

```c
struct Semaphore *
semaphore_create (int value)
{
    struct Semaphore *s;
    s = malloc (sizeof (struct Semaphore));
    s->v = value;
    pthread_mutex_init (&s->mutex, NULL);
    pthread_cond_init (&s->cond, NULL);
    return s;
}
```
Implementing Semaphores: Up

```c
void semaphore_up (struct Semaphore * s) {
    pthread_mutex_lock (&s->mutex);
    ++(s->v);
    pthread_cond_signal (&s->cond);
    pthread_mutex_unlock (&s->mutex);
}
```
Implementing Semaphores: Down

```c
void semaphore_down (struct Semaphore * s)
{
    int ret;

    pthread_mutex_lock (&s->mutex);
    while (s->v <= 0)
        pthread_cond_wait (&s->cond, &s->mutex);
    --(s->v);
    pthread_mutex_unlock (&s->mutex);
}
```
Using Semaphores: create queue

```c
struct queue *
queue_create (unsigned int length)
{
    queue = malloc(sizeof(struct queue));
    queue->queue_size = length;
    queue->semaphore_in = semaphore_create(length);
    queue->semaphore_out = semaphore_create(0);
    queue->write_pos = 0;
    queue->read_pos = 0;
    queue->elements = malloc(length * sizeof(void*));
    return queue;
}
```
**Using Semaphores: queue put**

void
queue_put (void * value)
{
    semaphore_down(queue->semaphore_in);
    queue->elements[queue->write_pos++] = value;
    if (queue->write_pos == queue_size)
        queue_write_pos = 0;
    semaphore_up(queue->semaphore_out);
}
Using Semaphores: queue get

```c
void *
queue_get ()
{
    void * ret;
    semaphore_down(queue->semaphore_out);
    ret = queue->elements[queue->read_pos++];
    if (queue->read_pos == queue_size)
        queue_read_pos = 0;
    semaphore_up(queue->semaphore_in);
}
```
volatile

- volatile tells the C compiler that a variable may change “at any time”

- Examples:
  - hardware devices mapped to memory locations
  - state modified by other threads without locking
  - state modified by signal handlers

⇒ Compiler must not do load elimination optimizations
Example: volatile

```c
volatile int flag;
static void handler(int s) { flag = 1; }
int main(int argc, char ** argv) {
    signal(SIGALRM, &handler);
    alarm(2);
    while (! flag) ;
    return 0;
}
```
Data Races

- Multiple processes or threads depend on mutable shared state

⇒ Result of execution depends on scheduling

⇒ Data races result in unpredictable (and usually undesirable) computational results
Memory Models

• Scheduling used to refer only to the operating system scheduler

• These days, out-of-order execution by processor or compiler also matters!

⇒ Memory models restrict what optimizations compilers can do

⇒ Synchronization operations restrict optimizations
Deadlock

A deadlock occurs if two or more activities are waiting for each other to finish before continuing:

- Thread 1 has lock $A$, waits for lock $B$
- Thread 2 has lock $B$, waits for lock $A$

If this kind of cycle exists, no progress will happen.
Deadlock: Solutions

- Have all threads acquire locks in the same order (always lock $A$ before lock $B$)
- Use fewer locks
- Release locks earlier
Livelock

- Similar to deadlock: no progress happens
- Main difference: CPU is still consumed
- Can happen if activities try to handle the deadlock (i.e. by giving up their locks) but run into an equivalent problem a few instructions later
Asynchronous IO

- read and write block until operation is complete (are synchronous)

- IO operations can take a long time where the CPU could do other things

⇒ Other processes could use the CPU, or
⇒ Other threads could use the CPU, or
⇒ the code could use asynchronous IO
Asynchronous IO: aiocb

struct aiocb {
    int aio_fildes;
    off_t aio_offset;
    volatile void *aio_buf;
    size_t aio_nbytes;
    /* ... */
};
Asynchronous IO: read

- `int aio_read(struct aiocb *aiocbp)`
- `int aio_error(const struct aiocb *aiocbp)`
- `ssize_t aio_return(struct aiocb *aiocbp)`
- `int aio_cancel(int fildes, const struct aiocb *aiocbp)`
- `int aio_fsync(int op, struct aiocb *aiocbp)`
Asynchronous IO: write

- `int aio_write(struct aiocb *aiocbp)`
- `int aio_error(const struct aiocb *aiocbp)`
- `ssize_t aio_return(struct aiocb *aiocbp)`
- `int aio_cancel(int fildes, const struct aiocb *aiocbp)`
- `int aio_fsync(int op, struct aiocb *aiocbp)`
Asynchronous IO: suspend

- `int aio_suspend(const struct aiocb *const list[], int nent, const struct timespec *timeout)`
Low-level APIs

- clone
- futex
- tgkill
futex

futex is how mutex is implemented on Linux:

- int futex(int *uaddr, int op, int val, const struct timespec *timeout, int *uaddr2, int val3);
tgkill

tgkill is how pthread_kill is implemented on Linux:

- int tgkill(int tgid, int tid, int sig)
Questions

?