VGP393 – Week 2

- Agenda:
  - Synchronization
    - Critical sections
    - Deadlock
    - Synchronization primitives
  - Win32 / MFC threading API, part 1
    - Creating / destroying threads
    - Events
    - Semaphores
    - Mutexes
    - Critical sections
  - Assignment #1

Synchronization

- Consider the following linked-list insertion code:

```
void insert_after(node *pre, node *next) {
    next->next = pre->next;
    next->prev = pre;
    next->next->prev = next;
    pre->next = next;
}
```

  - What happens if two threads try to insert nodes after the same `pre` at the "same" time?
    - Almost certainly the list will be corrupted
    - Timing sensitive bugs like this is called *race condition*
- By synchronizing access to shared data, race conditions can be avoided

Critical Sections

- "...a critical section is a piece of code that accesses a shared resource...that must not be concurrently accessed by more than one thread of execution.[1]"
  - In other words, the critical section is the area around which synchronization is required
  - We generally associate the synchronization with the data, not the the code

[1] http://en.wikipedia.org/wiki/Critical_section

Synchronization Primitives

- Numerous primitives with slightly different semantics have been developed over the years
  - Counting semaphore
  - Locks
    - Spin-lock
    - Mutex
    - Recursive locks
    - Read-write locks
  - Condition variables

Counting Semaphore

- Special counter that, when > 0 allows access to the critical section
  - Presented by Dijkstra in 1968, it is the *original* synchronization primitive
- Semaphore has three functions
  - `init` – sets the initial count, usually 0 or 1
  - `v` – increases the count and wakes sleepers
    - From the Dutch *verhogen* (increase)
    - Sometimes called `up`
  - `p` – decreases count and sleeps if result < 0
    - From the Dutch *probeer te verlagen* (try to decrease)
    - Sometimes called `down`

Counting Semaphore

- Implementation of `down` from Linux 2.6.25.9:

```
    lock decl     [%ebx]
    jns           2
    lea           [%ebx], %eax
    call          __down_failed
2:
```

`lock` prefix ensures that fetching the value from memory, decrementing, and writing back happen atomically

Puts thread on the sleep queue

Counting Semaphore

- Implementation of `down` from Linux 2.6.25.9:

```
        lock incr    [%ebx]
        jg           2
        lea          [%ebx], %eax
        call         __up_wakeup
2:
```

`lock` prefix ensures that fetching the value from memory, incrementing, and writing back happen atomically

Wakes up next waiting thread

Lock

- A lock is essentially a binary semaphore
  - A lock is either locked (has a count $\leq$ zero) or unlocked (has a count of one)
  - Lock operations `acquire` and `release` are analogous to the semaphore operations `p` ad `v`
    - Except that releasing a lock with a count of one is an error
  - Also called a *mutex*
    - Short for *mutual exclusion*

Spin-lock

- Very simple type of lock that doesn't sleep
  - Instead of sleeping, it loops testing the variable...waiting for it to change
  - Simple spin-lock implementation:

```
        movl          %eax, $1
1:      lock xchg     %eax, [%ebx]
        test          %eax, %eax
        jnz           1
```

  - Doesn't work well on uniprocessor systems
    - Unless the lock is waiting for something to happen in an interrupt
  - Hurts performance if the expected wait time is more than 50% of the per-thread time slice

Recursive Lock

- What happens with a simple mutex or spin-lock in the following code?

```
void recursive_func(...)
{
    acquire(l);
    ...
    recursive_func(...);
    ...
    release(l);
}
```

Recursive Lock

- What happens with a simple mutex or spin-lock in the following code?

```
void recursive_func(...)
{
    acquire(l);
    ...
    recursive_func(...);
    ...
    release(l);
}
```

First recursive call will block here *forever*

Recursive Lock

- Allows the lock's holder to acquire the lock repeatedly
  - Each acquire *must* have a matching release
  - May be more expensive than non-recursive locking primitive

Read-Write Lock

- Allows either a single writer or multiple readers access to the critical section
  - Called a *shared-exclusive lock* in distributed computing because the lock is either held in shared mode (read) or in exclusive mode (write)
- Difficult to implement *well*
  - Obvious implementation may trivially starve either writers (most common) or readers
  - This makes them much more expensive in the presence of reader / writer contention

Condition Variables

- Condition variables combine the availability of a lock *and* the existence of some condition
  - Three operations exist for condition variables:
    - `wait` – releases lock, waits until condition is signaled, returns with lock held
    - `signal` – wakes up one waiting thread, returns with lock held
    - `broadcast` – wakes up all waiting threads, returns with lock held

Condition Variables

- How is this useful?
  - One thread produces data items that will be used by other threads
    - *Consumer* threads want to sleep until data is ready
    - *Producer* threads signal consumers when data is ready
      - Producers may also want to sleep if the communication buffer is full

Condition Variables

```
void producer() {
    while (1) {
        acquire(lock);
        while (data_ready) {
            wait(cond, lock);
        }
        /* Do something to generate data */
        data_ready = true;
        signal(cond);
        release(lock);
    }
}

void consumer() {
    while (1) {
        acquire(lock);
        while (!data_ready) {
            wait(cond, lock);
        }
        /* Do something with the data */
        data_ready = false;
        signal(cond);
        release(lock);
    }
}
```

Condition Variables

- How can a condition variable be implemented?
  - Multiple threads need to wait
  - Either one or many threads need to be woken at once
    - Almost like multiple threads need to be in a critical section...
- Simplest implementation combines a lock and a semaphore
  - Lock controls a counter for the number of waiting threads
  - Waiting threads queue on the semaphore

Condition Variables

```
void wait(cond *cv, lock *l) {
    acquire(cv->lock);
    cv->waiting++;
    release(cv->lock);

    release(l);
    down(cv->sem);
    acquire(l);
}

void broadcast(cond *cv) {
    acquire(cv->lock);
    while (cv->waiting) {
        up(cv->sem);
    }

    cv->waiting = 0;
    release(cv->lock);
}
```

Fence

- Causes all memory operations issued before the fence to complete before any memory operations issued after the fence
  - May be called a *memory barrier* or a *memory fence*
  - Out-of-order architectures can reorder independent reads and writes
- Don't *usually* need to issues fences by hand
  - Synchronization primitives imply fences and prevent the compiler from reordering memory accesses around the synchronization primitive
- Wikipedia has good info on the subject: http://en.wikipedia.org/wiki/Memory_barrier

Barrier

- All threads block at a barrier until a certain number of threads have reached the barrier
  - The converse of a counting semaphore
  - Useful for certain parallel structures that we'll examine later...

Multi-threading on Win32

- Two different interfaces exist
  - Low-level win32 threads
  - Slightly higher-level MFC thread *objects*
    - Really just wrapper classes around win32 threads
- We'll use low-level win32 threads this term
  - MFC won't work with SDL, and some of the assignments use SDL

Thread Function

- Each thread starts with a function:

```
unsigned __stdcall my_thread_func(void *param);
```

  - This function is essentially the per-thread `main`
  - `param` points to arbitrary data passed in by the thread's creator

Thread Creation

- Several ways to create a new thread

```
HANDLE CreateThread(
    LPSECURTY_ATTRIBUTES thread_attributes,
    SIZE_T stack_size,
    LPTHREAD_START_ROUTINE start,
    void *parameter,
    unsigned creation_flags,
    unsigned *thread_id);
```

Thread Creation

- Several ways to create a new thread

```
uintptr_t _beginthreadex(
    void *security,
    unsigned stack_size,
    unsigned start(void *),
    void *parameter,
    unsigned creation_flags,
    unsigned *thread_id);
```

  - Parameters have same meaning as `CreateThread`
  - `start` function must be delcared `__stdcall`
  - Also configures C run-time support
    - Allows thread to use `printf`, for example

Thread Creation

- Several ways to create a new thread

```
uintptr_t _beginthread(
    void start(void *),
    unsigned stack_size,
    void *parameter);
```

  - Much like `_beginthreadex`, but assume default values for most parameters
  - `start` function must be delcared `__cdecl`
  - Thread also *cannot* return a value

Thread Termination

- Each thread can terminate itself in several ways
  - Simply return from the `start` function passed to the thread creation routine
  - Call `ExitThread`
    - Releases thread resources, cancels pending file I/O, etc.
    - Implicitly called by returning
    - Kills the thread *without* calling destructors, etc.
  - Call `_endthread` / `_endthreadex`
    - Works like `ExitThread`
    - Invokes destructors before terminating

Thread Termination

- Creating thread can force a thread to die by calling `TerminateThread`
  - Really dangerous!
  - Thread has no chance to clean-up before dying
    - Cannot free memory
    - Cannot close files
    - Cannot release synchronizations objects!!!

Waiting for Threads

- It is possible, and useful, sometimes to wait for a thread to terminate

```
unsigned WaitForSingleObject(
    HANDLE hHandle,
    unsigned milliseconds);
```

  - Returns `WAIT_OBJECT_0` on success
  - `WAIT_TIMEOUT` means the nothing happend in the allotted time
  - `WAIT_FAILED` means an error occured
  - `WAIT_ABANDONED` means the thread owning a mutex terminated before releasing the mutex

Waiting for Threads

- It is possible, and useful, sometimes to wait for a thread to terminate

```
unsigned WaitForMultipleObjects(
    unsigned count,
    const HANDLE *handles,
    BOOL wait_all
    unsigned milliseconds);
```

  – Returns `WAIT_OBJECT_0` + *n* on success
    – *n* is the element of `handles` that had something happen
  – Other return codes the same as `WaitForSingleObject`

Events

- Sends a signal to a thread
  - Event state change with `SetEvent` and `ResetEvent`
  - Waiting thread is notified when the event changes from reset to set state
    - This means the thread should reset `manual_reset` events after receiving

```
HANDLE CreateEvent(
    LPSECURTY_ATTRIBUTES thread_attributes,
    BOOL manual_reset
    BOOL initial_state
    LPCTSTR name);
BOOL SetEvent(HANDLE event);
BOOL ResetEvent(HANDLE event);
```

Semaphores

- Dijkstra-style counting semaphore

```
HANDLE CreateSemaphore(
    LPSECURTY_ATTRIBUTES thread_attributes,
    unsigned initial_count,
    unsigned maximum_count,
    LPCTSTR name);
```

  – Semaphores are named, and can be opened in other processes

```
HANDLE OpenSemaphore(
    unsigned desired_access,
    BOOL inherit_handle,
    LPCTSTR name);
```

Semaphores

- Semaphores have handles, and re-use `WaitForSingleObject` and `WaitForMultipleObjects`
  - A successful wait is a “p” operation on the semaphore
  - The “v” operation is `ReleaseSemaphore`

```
BOOL ReleaseSemaphore(
   HANDLE semaphore,
   unsigned release_count,
   unsigned *previous_count);
```

Mutexes

- Mutexes work just like semaphores, but are binary instead of counting
  - Use `CreateMutex` to create
  - Use `OpenMutex` to open
  - Use `ReleaseMutex` to release
  - Win32 mutexes are recursive

Critical Sections

- Critical sections look more like traditional locks
  - Unlike mutexes, critical sections are *not* fair
  - Win32 critical sections are recursive
  - Critical sections must be initialized before use

```
void InitializeCriticalSection(
    LPCRITICAL_SECTION crit_sect);
```

  - Acquire the “lock” with `EnterCriticalSection`

```
void EnterCriticalSection(
    LPCRITICAL_SECTION crit_sect);
```

  - Non-blocking acquire returns false if lock cannot be acquired

```
BOOL TryEnterCriticalSection(
    LPCRITICAL_SECTION crit_sect);
```

Critical Sections

- Critical sections look more like traditional locks
  - Release lock with `ReleaseCriticalSection`

```
void LeaveCriticalSection(
   LPCRITICAL_SECTION crit_sect);
```

  - Destroy lock with `DeleteCriticalSection`

```
void DeleteCriticalSection(
   LPCRITICAL_SECTION crit_sect);
```

Kernel Object vs. Per-process Object

- Semaphores and mutexes are kernel objects
  - Can be used to synchronize across process boundaries
  - Each operation has to go into the kernel
    - Expensive!
  - Multi-threaded, single process programs should prefer critical sections instead
    - Assuming the lack of fairness is acceptable

Condition Variables

- Windows only has *native* condition variables on Vista
  - In some cases events might be sufficient
  - In other cases a custom condition variable implementation must be created
    - Can use the “simple” implementation from earlier in the presentation
    - More efficient, complex implementations exist...see:
      - http://www.cs.wustl.edu/~schmidt/win32-cv-1.html
      - Search for “lock-free win32 condvar” messages by SenderX

Next week...

- Quiz #1
- Program decomposition
  - Task decomposition
  - Data decomposition
  - Data flow decomposition
- Parallel algorithm structure patterns
  - Task-level parallelism
  - Divide and conquer
  - Geometric decomposition
  - Pipeline
  - Wavefront

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.