CSCE 313-200
Introduction to Computer Systems
Spring 2022

Synchronization III
Dmitri Loguinov
Texas A&M University

February 15, 2022
**Updates**

- Midterm on Thursday
  - Covers everything since the beginning of the semester up to and including last Thursday
  - Questions drawn from lectures and homework #1 parts 1-2
  - Material in the book not discussed in class can be ignored
  - Chapter 5 problems (5.1-5.11) might be useful

- Make sure to understand Windows APIs
  - Meaning of parameters, usage in practice, possible errors
  - Reading/writing of pipes, creation of processes

- Be proficient in the 4 types of searches
  - Able to reproduce and discuss the algorithms, understand necessity for the two data structures (i.e., U and D)
Updates

• How to print statistics every 2 seconds?
  - Separate *stats* thread
  - Your wakeup time may be 2.1, 2.5, or 3 seconds apart!

• Make sure to print correct values
  - `Printf` recommended for progress report
  - Exit room ID when found, distance from rover, steps taken

• Win32 processes max out at ~1400 threads

• Can set thread stack size to 65,536 bytes:
  - Project Properties → Linker → System → Stack Reserve Size
  - Win32: this allows up to 6000 threads, x64: limited by RAM

• All robots initially in the same room with the rover
  - Check discovered set D before dropping initial room into U

divide by elapsed time
• Priorities
  - Thread priority is based on a combination of two things: process priority class and thread priority level within that class
  - SetPriorityClass() and SetThreadPriority()

• When running a massive amount of threads
  - Set priority of search threads to idle, stats to above normal

• CPU affinity
  - CPU restrictions expressed as bit masks
  - SetProcessAffinityMask(), SetThreadAffinityMask()

• How to set mask to include only CPU 0 and 4?
  - UINT64 mask = 1 + (1 << 4)
Homework #1 (Extra Credit)

• Monster randomly rampages in the cave
  – Eats flybots it can find, jams message transmission
  – Monster caves numbered 1000 and above, only planets 6-7

• If flybot is eaten
  – ReadFile/WriteFile block forever or return errors
  – Must re-insert the room where this happened back at the front of the queue and quit thread that experienced this condition

• Jammed transmission
  – Bogus status, truncated messages, or non-integer number of NodeTuple64s in the response
  – Discard invalid response and retry the room in same thread

• Sending robots to invalid room crashes CC.exe
Homework #1 (Extra Credit)

- Non-blocking pipes with ReadFile/WriteFile
  - Approach below is asynchronous, but not truly overlapped as it keeps only one pending request to the handle
  - We’ll see another version when dealing with file I/O

```c
// simple approach to catching timeouts
pipe = CreateFile (..., FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED, ...);

OVERLAPPED ol; // memset ol to zero

bRet = ReadFile (pipe, ..., NULL, &ol); // does not return bytesRead
// if bRet is FALSE, check if GetLastError() equals ERROR_IO_PENDING
// if so, ignore the error, continue; otherwise, terminate thread
bRet = WaitForSingleObject (pipe, timeout);
// bRet could be WAIT_TIMEOUT, WAIT_OBJECT_0, or some error
// if successful, obtain the # of bytes read:
GetOverlappedResult (pipe, &ol, ...);
```

- What’s a good timeout value?
Chapter 5: Roadmap

5.1 Concurrency
5.2 Hardware mutex
5.3 Semaphores
5.4 Monitors
5.5 Messages
5.6 Reader-Writer
Mutex

- Windows kernel mutex has semantics close to a binary semaphore 2.0, with two exceptions:
  - Repeated mutex lock from the same thread does not block it
  - Mutex can only be unlocked by the thread that locked it
- Examples:

```c
Semaphore semaX = {1, 1}; // (s,max)
Thread () {
    semaX.Wait();   // P
    semaX.Wait();   // P
}

Mutex m;              // unlocked
Thread () {
    m.Lock();
    m.Lock();
}
```

deadlocks because it attempts to decrement s twice

works fine as this thread already owns the mutex
### Mutex

- **Examples (cont’d):**

  ```c
  Semaphore semaX = {1, 1}; // (s,max)
  Thread1 () {
    semaX.Wait(); // P
    semaX.Wait(); // P
  }
  
  Semaphore semaX = {1, 1}; // (s,max)
  Thread2 () {
    // some initialization
    semaX.Release(); // V
  }
  ```

  **thread₁ blocks temporarily, then gets unblocked by thread₂**

  ```c
  Mutex m;
  Thread1 () {
    m.Unlock(); // does nothing
  }
  
  Mutex m; // initially unlocked
  Thread2 () { // thread2 runs first
    m.Lock();
    // long critical section
  }
  ```

  **thread₁ deadlocks if thread₂ runs first; how to fix this?**

  ```c
  Mutex m; // initially unlocked
  Thread2 () { // thread2 runs first
    m.Lock();
    // long critical section
  }
  ```

  **thread₁ fails to unlock mutex owned by thread₂**
The last standard synchronization primitive is an event
- An event can be in two states: signaled (1) and non-signaled (0) just like a binary semaphore
- However, it also has two possible modes of operation
  - AUTO = binary semaphore
  - MANUAL = event stays signaled until manually reset

```cpp
class Event {
    int s;  // state
    int mode;
    List blocked;
    Wait (); Set (); Reset ();
}

Event::Wait() {
    if (s == NOT_SIGNALED)
        // block current thread
    else if (mode == AUTO)
        s = NOT_SIGNALED;
}

Event::Set() {
    if (blocked.size() > 0)
        if (mode == AUTO)
            // unblock 1 thread
        else
            // unblock all threads
            s = SIGNALED;
    else
        s = SIGNALED;
}

Event::Reset() {
    s = NOT_SIGNALED;
}
```
Windows APIs

• Semaphore
  – Security is NULL as always
  – Name can be used when multiple processes need to open the same object
• Wait (i.e., P)
  – WaitForSingleObject()
  – Returns WAIT_OBJECT_0 when ready
  – WAIT_TIMEOUT if timeout
  – Otherwise, an error
• Release (i.e., V)
  – ReleaseSemaphore(N)

• CreateMutex/CreateEvent
  – Can specify if this thread initially owns the mutex and initial state for event
• Locking done with
  WaitForSingleObject()
  – Unlocking with
    ReleaseMutex() and signaling with SetEvent()
• Resetting events
  – ResetEvent()
Unbounded Producer-Consumer

- Producer-consumer is the most frequently encountered synchronization problem in programming
  - Will be solved using semaphores and mutexes
- Start with the unbounded version

K producers → unlimited queue → M consumers

- Producer threads create new items and deposit them into the shared buffer/queue
  - Consumer threads read from the buffer and process them
- Note that in some applications the same thread may act as producer and consumer at different times
  - This is the case in homework #1
Unbounded Producer-Consumer

- Several attempts to create a solution
  - PC v1.0

```java
Queue Q;
Producer() {
    while (true) {
        // make item x
        Q.add (x);
    }
}
```

- PC v1.1

```java
Queue Q;
Mutex m;
Producer() {
    while (true) {
        // make item x
        m.Lock();
        Q.add (x);
        m.Unlock();
    }
}
```

```java
Queue Q;
Consumer() {
    while (true) {
        m.Lock();
        if (Q.size() > 0)
            x = Q.pop();
        m.Unlock();
        // consume x
    }
}
```

problems?
Unbounded Producer-Consumer

• Ver 1.0 crashes on access to shared queue if used by multiple threads
• Ver 1.1 busy-spins waiting for queue to be non-empty
• Idea: assign a counting semaphore to control how many threads may attempt to read from the Q

- PC v1.2

```java
Queue Q;
Mutex m;
Semaphore sema = {0, ∞};
Producer() {
    while (true) {
        // make item x
        m.Lock();
        Q.add (x);
        sema.Release();
        m.Unlock();
    }
}
```

```java
Queue Q;
Mutex m;
Semaphore sema = {0, ∞};
Consumer() {
    while (true) {
        sema.Wait ();
        m.Lock();
        // no need to check Q.size
        x = Q.pop();
        m.Unlock();
        // consume x outside
        // the critical section
    }
}
```
Unbounded Producer-Consumer

- Ver 1.2 releases consumer on semaphore, which then gets immediately blocked on mutex; not efficient
  - PC v1.3

```java
Queue Q;
Mutex m;
Semaphore sema = {0, ∞};
Producer() {
    while (true) {
        // make item x
        m.Lock();
        Q.add (x);
        m.Unlock();
        sema.Release();
    }
}
```

```java
Queue Q;
Mutex m;
Semaphore sema = {0, ∞};
Consumer() {
    while (true) {
        sema.Wait ();
        m.Lock();
        x = Q.pop();
        m.Unlock();
        // consume x outside
        // the critical section
    }
}
```

- What if N items are produced in each iteration?
Unbounded Producer-Consumer

• If producer is bursty (i.e., generates many items at once), then ver 1.3 is also inefficient
  – PC v1.4

```plaintext
Queue Q;
Mutex m;
Semaphore sema = {0, ∞};
Producer() {
    while (true) {
        // make x[0],..., x[N-1]
        m.Lock();
        for (i = 0; i < N; i++)
            Q.add (x[i]);
        m.Unlock();
        // Windows allows batch
        // release
        sema.Release(N);
    }
}
```

```plaintext
Queue Q;
Mutex m;
Semaphore sema = {0, ∞};
Consumer() {
    while (true) {
        sema.Wait ();
        m.Lock();
        // no need to check Q.size
        x = Q.pop();
        m.Unlock();
        // consume x outside
        // the critical section
    }
}
```
Multithreaded search algorithm (rough idea)

```cpp
Mutex m; // not locked initially
Semaphore sema = {0, nMax}; // how to choose nMax?

Search::Run (...) // each thread runs this
{
    while (true) {
        // consumer starts here ----------
        sema.Wait ();
        m.Lock();
        x = U->pop();
        m.Unlock();
        // contact the robot and obtain x’s neighbors

        // producer starts here -------------
        counter = 0; // local variable that counts new neighbors
        m.Lock();
        for (each y = neighbor of x)
            if (D->CheckAdd(y) == false)
                U->add (y);
        counter ++;
        m.Unlock();
        sema.Release(counter);
    }
}
```

How does this terminate?
Homework #1

• How about this:

```java
Event eventQuit; // initially not signaled
...
{
    ...

    // contact the robot and obtain x’s neighbors
    if (x == exitNode)
        eventQuit.Signal();

    // producer starts here --------------
    ...
}
```

• Other conditions when we can signal termination?
  - U is empty and no more deposits into it are possible

• How to react to `eventQuit`?
  - Near the end, most threads will be blocked on semaphore
In order to wait on two objects (i.e., semaphore and event), we need
- \( b\text{WaitAll} = \text{false} \) means any of the handles can wake up this thread
- Otherwise, all handles must be simultaneously ready

When handle \( lp\text{Handles}[k] \) is triggered, this function returns \( \text{WAIT\_OBJECT\_0} + k \)

The order of handles in the array is important!
- If multiple handles are simultaneously in the signaled state, the return value indicates the first of them

```
DWORD WINAPI WaitForMultipleObjects(
    DWORD nCount,
    const HANDLE *lpHandles,
    BOOL bWaitAll,
    DWORD dwMilliseconds);
```
Wrap-up

• More complete version:

```
Mutex m; // not locked initially
Semaphore sema = {0, nMax};
Event eventQuit; // signaled to quit
int activeThreads = 0; // shared
```

Search::Run(...) {
    while (true) {
        // need to quit or work?
        if (WaitAny (eventQuit, sema)
            == eventQuit)
            break;
        m.Lock();
        x = U->pop();
        activeThreads ++;
        m.Unlock();

        // check if x is the exit
        if (x == exitNode)
            eventQuit.Signal();
        continue;
    }
}
```

• How to count *running* threads?
  - Printouts must include both running and active threads

should the event be manual or auto?