CSCE 313-200
Introduction to Computer Systems
Spring 2024

Synchronization IV

Dmitri Loguinov
Texas A&M University

February 16, 2024

Updates

divide by elapsed time

- 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

Updates

- Idle
- Lowest
- Below normal
- Normal
- Above normal
- Highest
- Time critical

- 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)

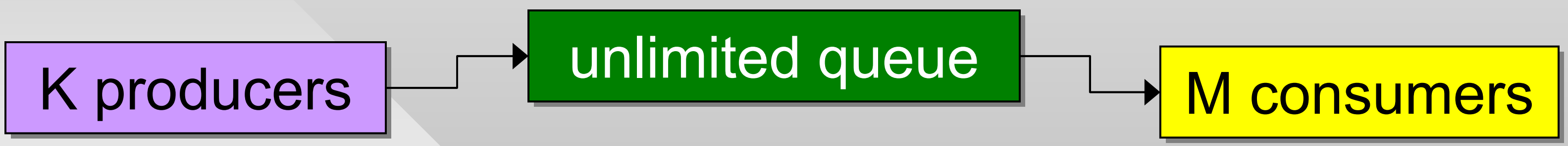
- Non-blocking pipes with ReadFile/WriteFile
  - Approach below is asynchronous, but not truly overlapped as it keeps only one pending request to the handle
  - We’ll see another version when dealing with file I/O

```
// 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?

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



- 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

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

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

  - PC v1.1

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

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

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

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

problems?

```
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

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

```
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
    }
}
```

- 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

```
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);
    }
}
```

```
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
    }
}
```

Homework #1

- Multi-threaded search algorithm (rough idea)

```
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:

```
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

Homework #1

- In order to wait on two objects (i.e., semaphore and event), we need
  - bWaitAll = false means any of the handles can wake up this thread
  - Otherwise, all handles must be simulataneously ready

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

- When handle `lpHandles[k]` is triggered, this function returns `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

Wrap-up

should the event be manual or auto?

- 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;
```

```
        int counter = 0;    // local var
        // deposit neighbors ----------
        m.Lock();
        for (each y = neighbor of x)
            if (D->CheckAdd (y) == false)
                 U->add (y);
                 counter ++;
        activeThreads --;
        if (U->size() == 0 &&
                  activeThreads == 0)
            eventQuit.Signal();
        m.Unlock();
        if (counter > 0)
            sema.Release(counter);
    }
}
```

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