CSCE 313-200
Introduction to Computer Systems
Spring 2018

Practice II

Dmitri Loguinov
Texas A&M University

March 6, 2018

Homework #2

- Request buffer allocated once per thread:

```
#define MAX_BATCH 10000
// set up initial buffer to hold header + MAX_BATCH rooms
char *request = new char [...];
CommandRobotHeader *crh = (...) request;
DWORD *roomArray = (...) (crh + 1);
```

- Then, batch-mode pop works as following:

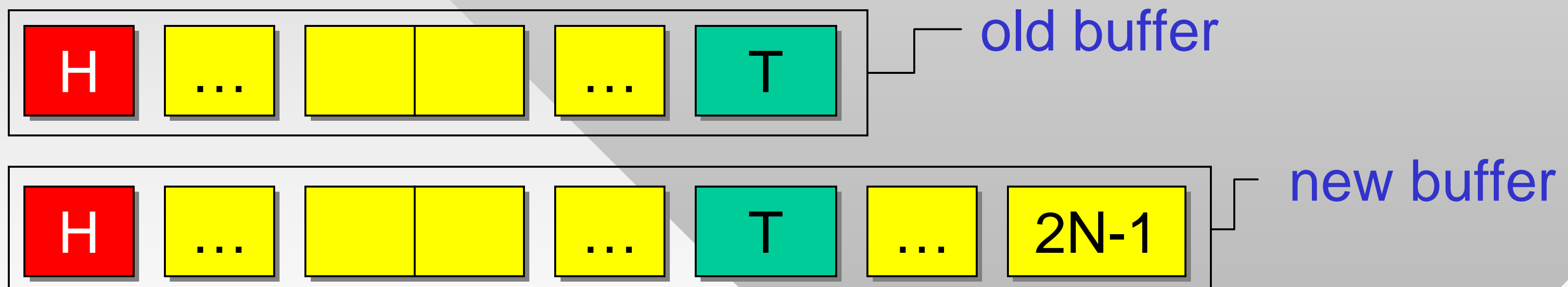
```
int nPopped = Q[cur].pop (roomArray, MAX_BATCH);
// compute msg size based on nPopped
pipe.SendMsg (request, requestSize);
```

- BFS queue class – needs to be written from scratch
  - Encapsulates a circular buffer with two offsets: head & tail
- Use a private heap inside the queue class
  - HeapCreate(), HeapAlloc(), HeapFree() instead of new/delete

Homework #2

```
// double queue size
size <<= 1;
buf = HeapReAlloc (heap, HEAP_NO_SERIALIZE,
                   buf, size);
```

- Simplified queue without concurrent push/pop
  - Push moves tail by batch size
  - Pop moves head similarly
- When buffer overflows, what operations are needed to double the queue size?



- Simplest is to use HeapReAlloc()
- Batch push/pop using memcpy

Homework #2

- Hash tables
  - 4B bits in a 512-MB buffer represent all possible nodes
  - Interlocked operations to access the bits
  - LONG array of $2^{32}/32 = 2^{27}$ words (each word is 4 bytes)
- Given room ID x, what is the offset and bit # in array?
  - Offset = x >> 5 (equivalent to x / 32)
  - Bit = x & 0x1F (equivalent to x % 32)

  bit ops are faster
- Batch pop directly from queue into the pipe buffer
  - Batch push is probably useful too
- Try to devise a method to interlock less frequently when the number of unique rooms drops close to 0%

Homework #2

- General structure, gets you to about 4M/sec

```
char *request = new char
          [sizeof(CommandRobotHeader) +
          MAX_BATCH * sizeof(DWORD)];
CommandRobotHeader *crh =
             (CommandRobotHeader*)request;
crh->command = MOVE;
DWORD *rooms = (DWORD *) (cr + 1);
while (true) {
    if (quit)          // flag set?
        break;

    int batch = 0;
    CS.lock();         // PC 3.4
    if (Q[cur].sizeQ > 0) {
        batch = Q[cur].pop (rooms, MAX_BATCH);
        activeThreads ++;
        // other stats go here
    }
    CS.unlock();
    if (batch == 0) {    // got nothing from Q?
        Sleep (100);
        continue;
    }
    pipe.SendMsg (...);        // send request[]
    pipe.RecvMsg (...);        // read response
    while (rooms left in response) {
        DWORD ID = ... // get next room
        DWORD offset = ...
        DWORD bit = ...
        if (LockedBitTestSet(hashTable,
                    offset, bit) == 0)
          localQ.push (ID);
    }

    CS.lock();
    // batch-pop all elements from
    // localQ into Q[cur^1]
    activeThreads --;
    if (this BFS level is over)
        if (next level empty)
          quit = true;
        else
          cur ^= 1;
    CS.unlock();
}
```

- Target delay is below 130 sec on P30

Quiz #2

- Much harder to write your own solutions than to understand others'
  - Stop by during office hours to brainstorm through your version
- Problem #1: Goats and bears want to party
  - Allowed to freely enter/leave unless pig is crashing
  - Pig can enter any time there are at least 50 animals inside
  - Nobody leaves or enters while pig is crashing
- Partially solved in class

Quiz #2

- Start with v1
  - Non-pig animals may deadlock even without the pig

```
void Animal::EnterBarn (void)
{
  Pig.Wait();
  Pig.Release();    // blocks arrivals

  m.Lock();
  inside ++;
  m.Unlock();

  if (inside >= 50)
    PigCanCrash.Release();

  Party();

  Pig.Wait();         // blocks departures
  m.Lock();
  inside --;
  m.Unlock();
  Pig.Release();

  if (inside == 49)
    PigCanCrash.Wait();   <-- some threads hang here
}
```

```
void Pig::EnterBarn (void)
{
  PigCanCrash.Wait();

  Pig.Wait();
  CrashParty();
  Pig.Release();

  PigCanCrash.Release();
}
```

- Lesson: mutex around *any* access to shared variables as long as they are modified elsewhere

Quiz #2

- Now v2
  - Find another deadlock, now involving the pig

```
void Animal::EnterBarn (void)
{
  Pig.Wait();
  Pig.Release();     // blocks arrivals

  m.Lock();
  if (++inside >= 50)
    PigCanCrash.Release();
  m.Unlock();

  Party();

  Pig.Wait();        // blocks departures

  m.Lock();
  if (inside-- == 50)
    PigCanCrash.Wait();  <---- deadlock here
  m.Unlock();

  Pig.Release();
}
```

```
void Pig::EnterBarn (void)
{
  PigCanCrash.Wait();

  Pig.Wait();   <---- deadlock here
  CrashParty();
  Pig.Release();

  PigCanCrash.Release();
}
```

- Lesson: never lock semaphores in opposite order in different threads

Quiz #2

- Finally v3
  - Releasing binary semaphore more than once back-to-back is undefined behavior in theory

```
void Animal::EnterBarn (void)
{
  Pig.Wait();
  Pig.Release();     // blocks arrivals

  m.Lock();
  if (++inside >= 50)
    PigCanCrash.Release();  <-- invalid release
  m.Unlock();

  Party();

  Pig.Wait();        // blocks departures
  Pig.Release();

  m.Lock();
  if (inside-- == 50)
    PigCanCrash.Wait();
  m.Unlock();
}
```

```
void Pig::EnterBarn (void)
{
  PigCanCrash.Wait();
  PigCanCrash.Release();

  Pig.Wait();
  CrashParty();
  Pig.Release();
}
```

- Lesson: do not release semaphore past its maximum
- In some cases leads to unintended behavior

Quiz #2

- Problem #5: up to 3 people can use the resource
  - If 3 are caught concurrently using it, all must depart before the next may enter
- Start with v1 →
  - One thread goes in, releases semaphore twice, allows 2 threads to pass s.Wait()
- Now v2 →
  - Suppose had3 = true
  - But all threads see inside == 0, release semaphore by 9

```
Semaphore s = {1,1};    // (s,max)
Mutex m;
int inside = 0;

s.Wait();   <---- context switch
m.Lock();
inside ++;
if (inside < 3)
    s.Release();   // allow one more
m.Unlock();

// use resource

m.Lock();
inside --;
if (inside == 0)
    s.Release();
m.Unlock();
```

allows unlimited # of threads in critical section

```
Semaphore s = {3,3};    // (s,max)
bool had3 = false;
int inside = 0;

s.Wait ();
InterlockedInc (inside);
if (inside == 3)
    had3 = true;

// use resource

InterlockedDec (inside);
if (had3)
    if (inside == 0) // last thread?
        had3 = false;
        s.Release (3);
else
    s.Release (1)
```

allows unlimited # of threads in critical section

Quiz #2

- Problem #3: savages and the cook
  - V1

```
Semaphore cook = {0, 1};
int chunks = 0;

Cook::Run (void) {
    while (true) {
        cook.Wait ();
        MakeFood ();
        chunks = M;
        cook.Release ();
    }
}
```

```
void Savage::AttemptToEat (void) {
    m.Lock();
    if (chunks == 0)
        cook.Release ();
        cook.Wait ();
    chunks --;
    m.Unlock();
    StartEating();
}
```

savages eat from empty pot or cook makes food non-stop

  - V2

```
Cook::Run (void) {
    while (true) {
        empty.Wait ();
        MakeFood ();
        chunks = M;
        full.Release ();
    }
}
```

```
void Savage::AttemptToEat (void) {
    if (chunks == 0) {
        empty.Release ();
        full.Wait ();
    }
    m.Lock();
    chunks --;
    m.Unlock();
    StartEating();
}
```

unlimited # of savages in critical section, cook burns savages

Quiz #2

- Now V3

```
Semaphore empty = {0, 1};
Semaphore full = {0, 1};
int chunks = 0;

Cook
while (true) {
    empty.Wait ();
    MakeFood ();
    chunks = M;
    full.Release ();
}
```

```
Savage
m.Lock();
if (chunks == 0)
    empty.Release ();
    full.Wait ();
chunks --;
m.Unlock();
StartEating();
```

cook burns savages

- Now V4:

```
Semaphore empty = {0, 1};
Semaphore full = {0, 1};
int chunks = 0;

Cook
while (true) {
    empty.Wait ();
    MakeFood ();
    chunks = M;
    full.Release ();
}
```

```
Savage
m.Lock();
if (chunks == 0)
    empty.Release ();
    full.Wait ();
chunks --;
StartEating();
m.Unlock();
```

inefficient, but avoids all other problems

- Finally V5

```
Semaphore cook = {1, 1};
Semaphore s = {0, M};
Cook
while (true) {
    cook.Wait ();
    MakeFood ();
    chunks = M;
    s.Release (M);
}
```

```
Savage
    s.Wait ();
    StartEating();

    m.Lock();
    if (--chunks == 0)
        cook.Release ();
    m.Unlock();
```

correct and most efficient

Quiz #2

- Problem #6: bus can carry up to 50 passengers
  - V1 has two problems: 1) deadlocks passengers, and 2) allows bus to close doors while someone is still boarding

```
int passengers = 0;
Semaphore AllAboard = {0, 1};

Bus
m.Lock();
if (passengers == 0)
   m.Unlock();
   return;
m.Unlock();

StopOpenDoors();
// allow passengers to board
BusArrived.Release();

// wait for passengers
AllAboard.Wait();
// prevent new ones from boarding
BusArrived.Wait();
CloseDoors ();
```

```
Semaphore s = {50, 50};
Semaphore BusArrived = {0, 1};

Passenger
s.Wait();        <-- never released
m.Lock();
passengers ++;
m.Unlock();

BusArrived.Wait();
BusArrived.Release();

BoardBus();

m.Lock();
passengers --;
if (passengers == 0)
    AllAboard.Release();
m.Unlock();
```

Quiz #2

- Now V2
- Finally V3

```
int allow = 0;

Bus
m.Lock();
allow = min(passengers, 50);

if (allow > 0) {
    m.Unlock();
    StopOpenDoors();
    BoardNow.Release (allow);

    // wait for passengers
    AllAboard.Wait ();
    CloseDoors ();
}
else
    m.Unlock(); // do not stop
```

```
Semaphore BoardNow = {0, 50};

Passenger
m.Lock();
passengers ++;
m.Unlock();

BoardNow.Wait();
BoardBus();

m.Lock();
passengers --;
allow --;
if (allow == 0)
    AllAboard.Release();
m.Unlock();
```

correct, but a bit complex

```
Bus
m.Lock();
// set some local variable
int grab = min(passengers, 50);
passengers -= grab;
m.Unlock();

if (grab > 0) {
    StopOpenDoors();
    Invited.Release (grab);

    // wait for passengers
    for (int i = 0; i < grab; i++)
        Done.Wait();
    CloseDoors ();
}
```

```
Semaphore Invited = {0, 50};
Semaphore Done = {0, 50};

Passenger
m.Lock();
passengers ++;
m.Unlock();

Invited.Wait();
BoardBus();

Done.Release(1);
```

correct and simple

Quiz #2

```
sA = (0,1);
thread1 () {
    sA.Wait();
    print A
    sB.Release();
}
```

```
sB = (1,1);
thread2 () {
    sB.Wait();
    print B
    sA.Release();
}
```

- Print ABAB… or BABA…
  - Many solutions are possible, one of the shortest is above
- However, it restricts the pattern to always start with B
  - What if B takes a long time to get there?
- Finding a flaw in a synchronization method means
  - Deadlock
  - Failed mutex (multiple threads in critical section)
  - Incorrect final result (numerically or otherwise)

```
bool want [2] = {false, false};
int turn = 0;
void Mutex::Lock (int id)     // process id = 0 or 1
{
1    want [id] = true;
2    while (turn != id) // other thread's turn?
     {
        // wait until other thread doesn't want it
3       while (want [1 - id])
4          ;
5       turn = id; // make the turn ours
     }
}
void Mutex::Unlock (int id)
{
6    want [id] = false;
}
```