CS 31: Intro to Systems
Misc. Threading

Kevin Webb
Swarthmore College
December 6, 2018
Agenda

• Classic thread patterns

• Pthreads primitives and examples of other forms of synchronization:
  – Condition variables
  – Barriers
  – RW locks
  – Message passing

• Message passing: alternative to shared memory
Common Thread Patterns

• Producer / Consumer (a.k.a. Bounded buffer)

• Thread pool (a.k.a. work queue)

• Thread per client connection
The Producer/Consumer Problem

- Producer produces data, places it in shared buffer
- Consumer consumes data, removes from buffer
- Cooperation: Producer feeds Consumer
  - How does data get from Producer to Consumer?
  - How does Consumer wait for Producer?
Producer/Consumer: Shared Memory

shared int buf[N], in = 0, out = 0;

**Producer**
while (TRUE) {
    buf[in] = Produce ();
    in = (in + 1)%N;
}

**Consumer**
while (TRUE) {
    Consume (buf[out]);
    out = (out + 1)%N;
}

- Data transferred in shared memory buffer.
Producer/Consumer: Shared Memory

shared int buf[N], in = 0, out = 0;

**Producer**
while (TRUE) {
    buf[in] = Produce ();
    in = (in + 1)%N;
}

**Consumer**
while (TRUE) {
    Consume (buf[out]);
    out = (out + 1)%N;
}

• Data transferred in shared memory buffer.

• Is there a problem with this code?
  A. Yes, this is broken.
  B. No, this ought to be fine.
This producer/consumer scenario requires synchronization to:

A. Avoid deadlock

B. Avoid double writes or empty consumes of buf[] slots

C. Protect a critical section with mutual exclusion

D. Copy data from producer to consumer

```c
shared int buf[N], in = 0, out = 0;

Producer
while (TRUE) {
    buf[in] = Produce ();
    in = (in + 1)%N;
}

Consumer
while (TRUE) {
    Consume (buf[out]);
    out = (out + 1)%N;
}
```
Adding Semaphores

```c
shared int buf[N], in = 0, out = 0;
shared sem filledslots = 0, emptyslots = N;

Producer
while (TRUE) {
    wait (X);
    buf[in] = Produce ();
in = (in + 1)%N;
signal (Y);
}

Consumer
while (TRUE) {
    wait (Z);
    Consume (buf[out]);
    out = (out + 1)%N;
signal (W);
}

• Recall semaphores:
  – wait(): decrement sem and block if sem value < 0
  – signal(): increment sem and unblock a waiting process (if any)
```
Suppose we now have two semaphores to protect our array. Where do we use them?

```c
shared int buf[N], in = 0, out = 0;
shared sem filledslots = 0, emptyslots = N;

Producer
while (TRUE) {
    wait (X);
    buf[in] = Produce ();
    in = (in + 1)%N;
    signal (Y);
}

Consumer
while (TRUE) {
    wait (Z);
    Consume (buf[out]);
    out = (out + 1)%N;
    signal (W);
}
```

<table>
<thead>
<tr>
<th>Answer choice</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>emptyslots</td>
<td>emptyslots</td>
<td>filledslots</td>
<td>filledslots</td>
</tr>
<tr>
<td>B.</td>
<td>emptyslots</td>
<td>filledslots</td>
<td>filledslots</td>
<td>emptyslots</td>
</tr>
<tr>
<td>C.</td>
<td>filledslots</td>
<td>emptyslots</td>
<td>emptyslots</td>
<td>filledslots</td>
</tr>
</tbody>
</table>
Add Semaphores for Synchronization

shared int buf[N], in = 0, out = 0;
shared sem filledslots = 0, emptyslots = N;

**Producer**
while (TRUE) {
    wait (emptyslots);
    buf[in] = Produce ();
    in = (in + 1) % N;
    signal (filledslots);
}

**Consumer**
while (TRUE) {
    wait (filledslots);
    Consume (buf[out]);
    out = (out + 1) % N;
    signal (emptyslots);
}

- Buffer empty, Consumer waits
- Buffer full, Producer waits
- Don’t confuse synchronization with mutual exclusion
Synchronization: More than Mutexes

• “I want to block a thread until something specific happens.”
  – Condition variable: wait for a condition to be true
Condition Variables

• In the pthreads library:
  – pthread_cond_init: Initialize CV
  – pthread_cond_wait: Wait on CV
  – pthread_cond_signal: Wakeup one waiter
  – pthread_cond_broadcast: Wakeup all waiters

• Condition variable is associated with a mutex:
  1. Lock mutex, realize conditions aren’t ready yet
  2. Temporarily give up mutex until CV signaled
  3. Reacquire mutex and wake up when ready
while (TRUE) {
    // independent code

    lock(m);
    while (conditions bad)
        wait(cond, m);

    // proceed knowing that conditions are now good

    signal (other_cond);  // Let other thread know
    unlock(m);
}

Condition Variable Pattern
Condition Variable Example

shared int buf[N], in = 0, out = 0;
shared int count = 0;  // # of items in buffer
shared mutex m;
shared cond notempty, notfull;

**Producer**
while (TRUE) {
    item = Produce();

    lock(m);
    while (count == N)
        wait(m, notfull);
    buf[in] = item;
    in = (in + 1)%N;
    count += 1;

    signal (notempty);
    unlock(m);
}

**Consumer**
while (TRUE) {
    lock(m);
    while (count == 0)
        wait(m, notempty);
    item = buf[out];
    out = (out + 1)%N;
    count -= 1;

    signal (notfull);
    unlock(m);
    Consume(item);
}
Synchronization: More than Mutexes

• “I want to block a thread until something specific happens.”
  – Condition variable: wait for a condition to be true

• “I want all my threads to sync up at the same point.”
  – Barrier: wait for everyone to catch up.
Barriers

• Used to coordinate threads, but also other forms of concurrent execution.

• Often found in simulations that have discrete rounds. (e.g., game of life)
shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}

Barrier Example, N Threads

T₀ T₁ T₂ T₃ T₄

Barrier (0 waiting)
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}

Threads make progress computing current round at different rates.
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}

Threads that make it to barrier must wait for all others to get there.

Barrier (3 waiting)
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
  while (...) {
    compute_sim_round()
    barrier_wait(&b)
  }
}

Barrier allows threads to pass when N threads reach it.
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round();
        barrier_wait(&b)
    }
}

Threads compute next round, wait on barrier again, repeat...

T0  T1  T2  T3  T4
Synchronization: More than Mutexes

• “I want to block a thread until something specific happens.”
  – Condition variable: wait for a condition to be true

• “I want all my threads to sync up at the same point.”
  – Barrier: wait for everyone to catch up.

• “I want my threads to share a critical section when they’re reading, but still safely write.”
  – Readers/writers lock: distinguish how lock is used
Readers/Writers

• Readers/Writers Problem:
  – An object is shared among several threads
  – Some threads only read the object, others only write it
  – We can safely allow multiple readers
  – But only one writer

• pthread_rwlock_t:
  – pthread_rwlock_init: initialize rwlock
  – pthread_rwlock_rdlock: lock for reading
  – pthread_rwlock_wrlock: lock for writing
Common Thread Patterns

• Producer / Consumer (a.k.a. Bounded buffer)

• Thread pool (a.k.a. work queue)

• Thread per client connection
Thread Pool / Work Queue

- Common way of structuring threaded apps:
Thread Pool / Work Queue

• Common way of structuring threaded apps:

Queue of work to be done:  

Thread Pool
Thread Pool / Work Queue

• Common way of structuring threaded apps:

Queue of work to be done:

Farm out work to threads when they’re idle.
Thread Pool / Work Queue

• Common way of structuring threaded apps:

  Queue of work to be done:

  As threads finish work at their own rate, they grab the next item in queue.

  Common for “embarrassingly parallel” algorithms.

  Works across the network too!
Thread Per Client

• Consider Web server:
  – Client connects
  – Client asks for a page:
    • http://web.cs.swarthmore.edu/~kwebb/cs31
    • “Give me /~kwebb/cs31”
  – Server looks through file system to find path (I/O)
  – Server sends back html for client browser (I/O)

• Web server does this for MANY clients at once
Thread Per Client

• Server “main” thread:
  – Wait for new connections
  – Upon receiving one, spawn new client thread
  – Continue waiting for new connections, repeat...

• Client threads:
  – Read client request, find files in file system
  – Send files back to client
  – Nice property: Each client is independent
  – Nice property: When a thread does I/O, it gets blocked for a while. OS can schedule another one.
Message Passing

- Operating system mechanism for IPC
  - send (destination, message_buffer)
  - receive (source, message_buffer)
- Data transfer: in to and out of kernel message buffers
- Synchronization: can’t receive until message is sent
Suppose we’re using message passing, will this code operate correctly?

/* NO SHARED MEMORY */

Producer
int item;

while (TRUE) {
    item = Produce ();
    send (Consumer, &item);
}

Consumer
int item;

while (TRUE) {
    receive (Producer, &item);
    Consume (item);
}

A. No, there is a race condition.
B. No, we need to protect item.
C. Yes, this code is correct.
This code is correct and relatively simple. Why don’t we always just use message passing (vs semaphores, etc.)?

A. Message passing copies more data.
B. Message passing only works across a network.
C. Message passing is a security risk.
D. We usually do use message passing!
Issues with Message Passing

- Who should messages be addressed to?
  - ports (mailboxes) rather than processes/threads
- What if it wants to receive from anyone?
  - \texttt{pid = receive (*, msg)}
- Synchronous (blocking) vs. asynchronous (non-blocking)
- Kernel buffering: how many sends w/o receives?
- Good paradigm for IPC over networks
Summary

• Many ways to solve the same classic problems
  – Producer/Consumer: semaphores, CVs, messages

• There’s more to synchronization than just mutual exclusion!
  – CVs, barriers, RWlocks, and others.

• Message passing doesn’t require shared mem.
  – Useful for “threads” on different machines.