CS 31: Intro to Systems
More Thread Synchronization

Martin Gagne
Swarthmore College
April 20, 2016
Common Thread Patterns

• Thread pool (a.k.a. work queue)

• Thread per client connection

• Producer / Consumer (a.k.a. Bounded buffer)
Common Thread Patterns

• Thread pool (a.k.a. work queue)

• Thread per client connection

• Producer / Consumer (a.k.a. Bounded buffer)
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
Thread Pool / Work Queue

- Common way of structuring threaded apps:

Queue of work to be done:  

Thread Pool  

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 a web server:
  • Client connects
  • Client asks for a page:
    • https://www.cs.swarthmore.edu/~mgagne1/teaching/2016_17/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.
The Producer/Consumer Problem

- Producer produces data, places it in shared buffer
- Consumer consumes data, removes from buffer

All kinds of real-world examples:

- print queue: printer is consumer
- CPU queue of ready processes/threads to run on CPU
Semaphores

• Semaphore: synchronization variable
  – Has integer value
  – List of waiting threads

• Works like a gate

• If sem > 0, gate is open
  – Value equals number of threads that can enter

• Else, gate is closed
  – Possibly with waiting threads
Semaphores

• Associated with each semaphore is a queue of waiting threads
• When wait() is called by a thread:
  – If semaphore is open, thread continues, value decreased
  – If semaphore is closed, thread blocks on queue
• Then signal() opens the semaphore:
  – the value of the semaphore is increased
  – If a thread is waiting on the queue, the thread is unblocked
Semaphore Operations

```c
sem s = n;       // declare and initialize

wait (sem s)     // Executes atomically
    decrement s;
    if s < 0, block thread (and associate with s);

signal (sem s)   // Executes atomically
    increment s;
    if blocked threads, unblock (any) one of them;
```
The Producer/Consumer Problem

Semaphore fillcount = 0;
Semaphore emptycount = size;
in = 0;
out = 0;

Producer:
while (true):
    item = produce()
    wait(emptycount)
    buffer[in] = item
    in = (in+1) % size
    signal(fillcount)

Consumer:
while (true):
    wait(fillcount)
    item = buffer[out]
    out = (out+1) % size
    signal(emptycount)
    consumeItem(item)
The Producer/Consumer Problem

Semaphore fillcount = 0; Semaphore emptycount = size;
in = 0; out = 0;

**Producer:**

```java
while (true):
    item = produce()
    wait(emptycount)
    buffer[in] = item
    in = (in+1) % size
    signal(fillcount)
```

**Consumer:**

```java
while (true):
    wait(fillcount)
    item = buffer[out]
    out = (out+1) % size
    signal(emptycount)
    consumeItem(item)
```

must be atomic if many producers/consumers
Condition Variables

A condition variable is queue of threads (TIDs) associated to a condition by the programmer.

• In the pthreads library:
  • `pthread_cond_init`: Initialize CV
  • `pthread_cond_wait`: Wait on CV
  • `pthread_cond_signal`: Wake up one waiter
  • `pthread_cond_broadcast`: Wake up all waiters

• Condition variable is associated with a mutex:
  1. Lock mutex, realize conditions aren’t ready yet.
  2. Cond_Wait to release the mutex and wait for signal.
  3. Automatically regain mutex when woken.
Using Condition Variables

(global) mutex m;
(global) cond_var cond;

while (TRUE) {
    //independent code

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

    //proceed knowing that conditions are now good

    unlock(m);
}
Read/Write locks

• Readers/Writers Problem:
  • An object is shared among several threads.
  • Some threads only read the object, others may write it.
  • We can safely allow multiple readers.
  • But writers need exclusive access.

• pthread_rwlock_t:
  • pthread_rwlock_init: initialize rwlock
  • pthread_rwlock_rdlock: lock for reading
  • pthread_rwlock_wrlock: lock for writing
Dining Philosopher Problem

- Five silent philosophers
- Plate of spaghetti in front of each
- Philosophers alternate between thinking and eating (infinite plate, infinite stomach)
- Philosophers need 2 forks to eat
- Philosopher can only take forks one at a time

What rules should we impose to ensure the philosophers don’t starve?
Dining Philosopher Problem

Example of incorrect solution:

• When a philosopher becomes hungry
  – take a fork as soon as one becomes available (left one if both available)
  – take a second fork as soon as it becomes available
  – eat
  – put back forks on the table

If all philosophers become hungry at the same time, and all take the left fork at the same time, then they all starve.

Deadlock!
What is Deadlock?

• Deadlock is a problem that can arise…
  • when processes compete for access to limited system resources.
  • when threads are incorrectly synchronized.

• Definition:
  • Deadlock exists among a set of threads if every thread is waiting for an event that can be caused only by another thread in the set.
Traffic Jam as Example of Deadlock

• Cars A, B, C, D

• Road W, X, Y, Z

• Car A holds road space Y, waiting for space Z

• “Gridlock”

Cars deadlocked in an intersection
Traffic Jam as Example of Deadlock

Cars deadlocked in an intersection

Resource Allocation Graph
Four Conditions for Deadlock

1. Mutual Exclusion
   • Only one thread may use a resource at a time.

2. Hold-and-Wait
   • Thread holds resource while waiting for another.

3. No Preemption
   • Can’t take a resource away from a thread.

4. Circular Wait
   • The waiting threads form a cycle.
Four Conditions for Deadlock

1. Mutual Exclusion
   • Only one thread may use a resource at a time.

2. Hold-and-Wait
   • Thread holds resource while waiting for another.

3. No Preemption
   • Can’t take a resource away from a thread.

4. Circular Wait
   • The waiting threads form a cycle.
Why are all four necessary?

For each condition, assume it doesn’t occur, but the other 3 do, and explain why deadlock can’t happen.

1. Mutual Exclusion
2. Hold-and-Wait
3. No Preemption
4. Circular Wait
What rules should we impose to ensure the philosophers don’t starve?
Up Next

More on deadlocks

• examples of how they might happen
• how to prevent them