Synchronization

Part II

and

Conclusion

Definitions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

Lock: prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- Unlock when leaving, after done accessing shared data
- Wait if locked (all synchronization involves waiting!)

Homework Solution: Bounded Buffer

```
get() {
  lock(_key)
  {
    while (front == tail) {
      monitor.wait(_key);
    }
    item = buf[front % MAX];
    front++;
    monitor.signal(_key);
  }
  return item;
}
```

```
put(item) {
  lock.(_key);
  {
    while ((tail – front) == MAX) {
      monitor.wait(_key);
    }
    buf[tail % MAX] = item;
    tail++;
    monitor.signal(_key);
  }
}
```

Initially: front = tail = 0; MAX is buffer capacity

empty/full are condition variables

Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - front <= tail
  - front + MAX >= tail
- These are also true on return from wait
- And at lock release
- Allows for proof of correctness

Pre/Post Conditions

```
methodThatWaits() {
   lock.acquire();
   // Pre-condition: State is consistent

   // Read/write shared state

   while (!testSharedState()) {
       cv.wait(&lock);
    }
   // WARNING: shared state may
   // have changed!  But
  // testSharedState is TRUE
  // and pre-condition is true

  // Read/write shared state
   lock.release();
}
```

```
methodThatSignals() {
   lock.acquire();
   // Pre-condition: State is consistent

   // Read/write shared state

   // If testSharedState is now true
   cv.signal(&lock);

   // NO WARNING: signal keeps lock

   // Read/write shared state
   lock.release();
}
```

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock

  จะไม่มีการเกิด Interupt หรือ Context switch จะไม่มีการแทรกกลางของโปรเซสอื่นๆ ระหว่างการทำงานได้

  - What if wait, then release?
  - What if release, then wait? This is a correct version

Condition Variables, cont’d

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

```
while (needToWait()) {
    condition.Wait(lock);
}
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Synchronization Performance

- A program with lots of concurrent threads can still have poor performance on a multiprocessor:
  - Overhead of creating threads, if not needed
  - Lock contention: only one thread at a time can hold a given lock
  - Shared data protected by a lock may ping back and forth between cores
  - False sharing: communication between cores even for data that is not shared

Reducing Lock Contention

- Fine-grained locking
  - Partition object into subsets, each protected by its own lock
  - Example: hash table buckets
- Per-processor data structures
  - Partition object so that most/all accesses are made by one processor
  - Example: per-processor heap
- Ownership/Staged architecture
  - Only one thread at a time accesses shared data
  - Example: pipeline of threads

Synchronization without Lock

Communicating Sequential Processes (CSP/Google Go)

- A thread per shared object
  - Only thread allowed to touch object's data
  - To call a method on the object, send thread a message with method name, arguments
  - Thread waits in a loop, get msg, do operation
- No memory races!

Locks/CVs vs. CSP

- Create a lock on shared data

  = create a single thread to operate on data
- Call a method on a shared object

  = send a message/wait for reply
- Wait for a condition

  = queue an operation that can’t be completed just yet
- Signal a condition

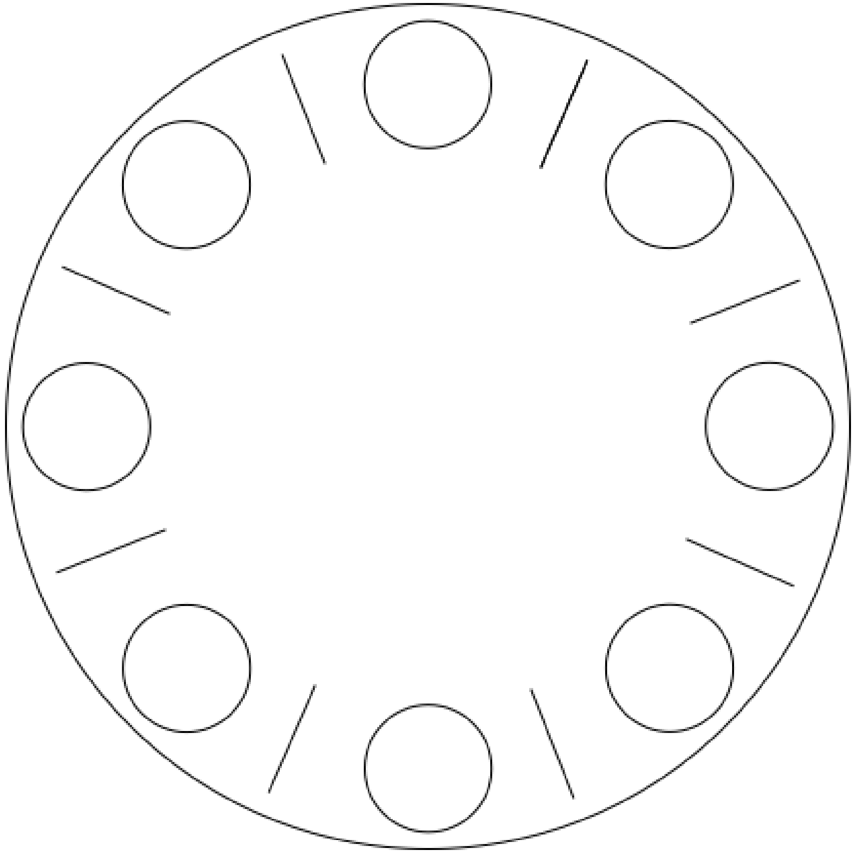
  = perform a queued operation, now enabled

Multi-Object Synchronization

Multi-Object Programs

- What happens when we try to synchronize across multiple objects in a large program?
  - Each object with its own lock, condition variables
- Performance
- Semantics/correctness
- Deadlock
- Eliminating locks

Dining Lawyers



Each lawyer needs two chopsticks to eat.
Each grabs chopstick on the right first.

Deadlock Definition

- Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
  - Preemptable: can be taken away by OS
  - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
  - Deadlock => starvation, but not vice versa

Example: two locks

Thread A

```
lock1.acquire();
lock2.acquire();
lock2.release();
lock1.release();
```

Thread B

```
lock2.acquire();
lock1.acquire();
lock1.release();
lock2.release();
```

Two locks and a condition variable

Thread A

```
lock1.acquire();
…
lock2.acquire();
while (need to wait) {
    condition.wait(lock2);
}
lock2.release();
…
lock1.release();
```

Thread B

```
lock1.acquire();
…
lock2.acquire();
…
condition.signal(lock2);
…
lock2.release();
…
lock1.release();
```

Necessary Conditions for Deadlock

- Limited access to resources
  - If infinite resources, no deadlock!
- No preemption
  - If resources are virtual, can break deadlock
- Multiple independent requests
  - “wait while holding”
- Circular chain of requests

Preventing Deadlock

- Exploit or limit program behavior
  - Limit program from doing anything that might lead to deadlock
- Predict the future
  - If we know what program will do, we can tell if granting a resource might lead to deadlock
- Detect and recover
  - If we can rollback a thread, we can fix a deadlock once it occurs

Exploit or Limit Behavior

- Provide enough resources
  - How many chopsticks are enough?
- Eliminate wait while holding
  - Release lock when calling out of module
  - Telephone circuit setup
- Eliminate circular waiting
  - Lock ordering: always acquire locks in a fixed order
  - Example: move file from one directory to another

Semaphores

- Semaphore has a non-negative integer value
  - P() atomically waits for value to become > 0, then decrements
  - V() atomically increments value (waking up waiter if needed)
- Semaphores are like integers except:
  - Only operations are P and V
  - Operations are atomic
    - If value is 1, two P’s will result in value 0 and one waiter
- Semaphores are useful for
  - Unlocked wait: interrupt handler, fork/join

Example:

semaphore smp = new semaphore;	
smp.v();	smp’s value = 1
smp.p();	try to reduce smp by 1 (acquire lock)
ShR = 100;	Working with shared resource(s)
smp.v();	try to increse smp by 1 (release lock)