Announcements

• Practice midterm available on Canvas (with solution)
  ♦ Real midterm will have similar format (e.g., T or F, multiple choices, short answer, large questions), but different content
  ♦ We will ask one question from project 1 => Do your own work!

• Midterm covers everything until this Friday’s lecture
  ♦ Backup slides not in exam

• Midterm review next Monday

• Midterm next Wed 7pm - 8:50pm (LEDDN AUD 216)
Deep thinking (not in exam, take home reading)

- How can we separate “checking” from “buying milk” and only lock “checking”?

  ```c
  local_flag = FALSE;

  Acquire(lock);
  if (no note && noMilk){
    leave note;
    local_flag = true;
  }
  Release(lock);

  If (local_flag) buy milk;
  If (local_flag){
    local_flag = FALSE;
    remove note;
  }
  ```
Producer & Consumer – CV - problem?

Producer

while (1) {
    produce an item;
    acquire(mutex);
    if (pool is Full) {
        release(mutex);
        wait(NotFULL);
        acquire(mutex);
    }
    record if pool was empty;
    insert(item);
    if (pool was empty)
        signal(NotEMPTY);
    release(mutex);
}

Consumer

While (1) {
    acquire(mutex);
    if (pool is Empty) {
        release(mutex);
        wait(NotEMPTY);
        acquire(mutex);
    }
    record if pool was full;
    remove(item);
    if (pool was Full)
        signal(NotFULL);
    release(mutex);
    use the item for some work;
}
Producer & Consumer -- use condition variables

Producer

while (1) {
    produce an item;

    acquire(mutex);
    if (pool is Full) {
        wait(NotFULL);
    }
    record if pool was empty;
    insert(item);

    if (pool was empty)
        signal(NotEMPTY);
    release(mutex);
}

Consumer

While (1) {
    acquire(mutex);
    if (pool is Empty) {
        wait(NotEMPTY);
    }
    record if pool was full;
    remove(item);

    if (pool was Full)
        signal(NotFULL);
    release(mutex);

    use the item for some work;
}

The simplification implies NotFull is tied to mutex
Signal Semantics

- signal() places a waiter on the ready queue, but signaler continues inside lock
  - Known as “Mesa” style, easy to implement
  - Another early-time semantics is Hoare style (signaler gives up lock, waiter runs immediately)

- What’s the implication of when signaler releases lock?
  - When consumer thread has some other operations to do after signal but before releasing lock
What could go wrong?

• What can happen when the awaken thread gets a chance to run?
  ♦ E.g. pool is full, producer 1 waits; consumer signals it; p1 in ready queue; consumer release(lock); p2 comes along…

• Condition not necessarily true when waiter runs again
  ♦ Returning from wait() is only a hint that something changed
  ♦ Must recheck conditional case
Producer & Consumer – use condition variables – how to fix?

### Producer

```c
while (1) {
    produce an item;
    
    acquire(mutex);
    while (pool is Full) {
        wait(NotFULL);
    }
    record if pool was empty;
    insert(item);
    
    if (pool was empty)
        signal(NotEMPTY);
    release(mutex);
}
```

### Consumer

```c
While (1) {
    acquire(mutex);
    while (pool is Empty) {
        wait(NotEMPTY);
    }
    record if pool was full;
    remove(item);
    
    if (pool was Full)
        signal(NotFULL);
    release(mutex);
    use the item for some work;
}
```

*Is this busy waiting?*
Be Careful About Pitfalls: CVs Cannot Be “Tested”

- Do not use a CV as a predicate
- Need to use a separate flag

```c
acquire(lock);
...
while (CV != true) {
    wait(CV);
}
...
release(lock);
```

```c
acquire(lock);
...
while (flag != true) {
    wait(CV);
}
...
release(lock);
```
Be Careful About Pitfalls: CVs Require Holding Lock

- Do not release the lock before using the CV
  - Using a CV requires a thread to hold the lock
- Purpose of a CV is to enable threads to block while in a critical section

```c
acquire(lock);
...
release(lock);
wait(CV);
acquire(lock);
...
release(lock);
```

```c
acquire(lock);
...
wait(CV);
...
acquire(lock);
```
Be Careful About Pitfalls: Need Lock When Testing Flag

... if (check-condition) {
    acquire(lock);
    wait(CV);
    release(lock);
}
...

acquire(lock);
...
if (check-condition) {
    wait(CV);
}
...
release(lock);

• Testing a condition needs to be done while holding the lock
• It is a shared variable that can lead to race conditions
Semaphore

• A synchronization variable that takes on non-negative integer values
  ♦ Invented by Edsger Dijkstra in the mid 60’s

• Two primitive operations
  ♦ `wait(semaphore)`: an atomic operation that waits for semaphore to become greater than 0, then decrements it by 1
  ♦ `signal(semaphore)`: an atomic operation that increments semaphore by 1
Semaphore

```c
wait(S) {
    while (S<=0)
        ;
    S--;
}

signal(S) {
    S++;
}
```

- Historically, `wait()` is known as `P()`, signal is known as `V()``
- In reality, `wait/signal` are not implemented as above
Blocking in Semaphores

- Associated with each semaphore is a queue of waiting threads
- When `wait()` is called by a thread:
  - If semaphore is greater than 0, thread continues
  - Otherwise, thread blocks on queue
- Then `signal()` opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread
    » In other words, `signal()` has “history” (c.f., condition vars)
    » This “history” is a counter
Binary Semaphore

Init: $S = 1$;

```c
wait(S) {
  while (S==0)
  ;
  S--;
}
```

```c
signal(S) {
  if (S == 0)
    S++;
}
```

- **Binary semaphores**: only take 0 or 1
- Sounds familiar?
  - $S=0 \rightarrow$ someone is holding the lock!
What happens if initially $S = 1$

- **T1**: $P(S)$, ..., $V(S)$
- **T2**: $P(S)$, ..., $V(S)$
- **T3**: $P(S)$, ..., $V(S)$
Semaphore

P(S)

wait(S) {
    while (S<=0);
    S--;
}

V(S)

signal(S) {
    S++;
}

What happens if initially S = 1

- T1: P(S), …, V(S)
- T2: P(S) --------→, …, V(S)
- T3: P(S) --------------------------→, …, V(S)
Semaphore

\[
P(S) \\
\text{wait}(S) \{ \\
\quad \text{while} \ (S \leq 0); \\
\quad S--; \\
\} \\
\]

\[
V(S) \\
\text{signal}(S) \{ \\
\quad S++; \\
\} \\
\]

What happens if initially \( S = 1 \)

- T1: \( P(S), \ldots, V(S) \)
- T2: \( V(S), \ldots, P(S) \)
- T3: \( V(S), \ldots, P(S) \)
Semaphore

P(S)

wait(S) {
    while (S<=0);
    S--;
}

V(S)

signal(S) {
    S++;
}

What happens if initially S = 2

• T1: P(S), ..., V(S)
• T2: P(S), ..., V(S)
• T3: P(S), ..., V(S)
Semaphore

P(S)

wait(S) {
    while (S<=0);
    S--;
}

V(S)

signal(S) {
    S++;
}

What happens if initially S = 2

• T1: P(S), …, V(S)
• T2: P(S), …, V(S)
• T3: P(S)\rightarrow, …, V(S)
semaphore has built-in counting!

- signal(S) simply increments S
  - produce an "item"
  - S value = how many "items" have been produced

- wait(S) will return without waiting only if S > 0;
  - Wait(S) is saying wait until there is at least one "item", and consume an "item"
Two usages of semaphores

• For mutual exclusion:
  ♦ to ensure that only one thread is accessing shared info at a time.
  ♦ Non-binary semaphores or binary semaphores?

• For condition synchronization:
  ♦ to permit threads to wait for certain things to happen
  ♦ Non-binary semaphores or binary semaphores?
Semaphore benefits over locks

- Has a value => more semantics
  - When greater than 1, can allow multiple threads to access critical resource
  - When equal to 1, can be used for mutual exclusion (only one thread in critical section)
Producer & Consumer (semaphore version)

- Define constraints (what is “correct”)
  - Consumer must wait for producer to fill buffers (mutual excl. or condition sync?)
  - Producer must wait for consumer to empty buffers, if all buffer space is in use (mutual excl. or condition sync?)

- Use a separate semaphore for each constraint
  - FILLED = 0
  - EMPTY = N
  - (assuming the shared buffer size is N)
Producer & Consumer – semaphore attempt, what’s wrong?

Producer

\[
\text{while (1) { }
\]

\[
\text{produce an item; }
\]

\[
\text{wait(EMPTY); }
\]

\[
\text{insert(item to pool); }
\]

\[
\text{signal(FILLED)}
\]

\[
}\]

Consumer

\[
\text{While (1) { }
\]

\[
\text{wait(FILLED); }
\]

\[
\text{remove(item from pool); }
\]

\[
\text{signal(EMPTY); }
\]

\[
\text{consume the item; }
\]

\[
}\]

Init: FILLED = 0; EMPTY = N;
int buffer[MAX];
int fill = 0;
int use = 0;

insert (int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX
}

int get() {
    int tmp = buffer[use]
    use = (use + 1) % MAX
    return tmp;
}

Need to protect shared resource (critical section)!
Producer & Consumer (cont)

• Define constraints (what is “correct”)
  ♦ Consumer must wait for producer to fill buffers (mutual excl. or condition sync?)
  ♦ Producer must wait for consumer to empty buffers, if all buffer space is in use (mutual excl. or condition sync?)
  ♦ Only one thread must manipulate buffer pool at once (mutual excl. or condition sync?)

• Use a separate semaphore for each constraint
  ♦ FILLED = 0
  ♦ EMPTY = N
  ♦ MUTEX = 1 (binary semaphore), or use a lock
Producer & Consumer – semaphore attempt 2, what’s wrong?

Producer

while (1) {

produce an item;
wait(MUTEX);
wait(EMPTY);
insert(item to pool);
signal(FILLED);
signal(MUTEX);
}

Consumer

While (1) {

wait(MUTEX);
wait(FILLED);
remove(item from pool);
signal(EMPTY);
signal(MUTEX);
consume the item;
}

Init: FILLED = 0; EMPTY = N; MUTEX = 1;

Deadlock!
Producer & Consumer – semaphore working

Producer

while (1) {
produce an item;
wait(EMPTY);
wait(MUTEX);
insert(item to pool);
signal(MUTEX);
signal(FILLED);
}

Consumer

While (1) {
wait(FILLED);
wait(MUTEX);
remove(item from pool);
signal(MUTEX);
signal(EMPTY);
consume the item;
}

Init: FILLED = 0; EMPTY = N; MUTEX = 1;
Readers-Writers problem

• A data object is shared among multiple threads
• Allow concurrent reads (but no writes)
• Only allow exclusive writes (no other writes or reads)

• Used a lot in many data accessing systems
  ♦ E.g., our old bank account balance example
Semaphore Summary

• Semaphores can be used to solve many synchronization problems

• However, they have some drawbacks
  ♦ They are essentially shared global variables
    » Can potentially be accessed anywhere in program
    » Relies on programmers to properly guard and use it
    » No control or guarantee of proper usage
  ♦ No connection between the semaphore and the data being controlled by the semaphore
  ♦ Sometimes hard to use and prone to bugs
    » Difficult to get the counting right (e.g., initial value)
Synchronization Primitives

Summary

• Lock
  ♦ Only achieves mutual exclusion

• Semaphores
  ♦ Has built-in counters, and thus can express more semantics
  ♦ Can be inconvenient to use

• Condition variables
  ♦ Used by threads as a synchronization point to wait for events
  ♦ Used with locks or inside monitors

• Monitors
  ♦ Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
  ♦ Relies upon high-level language support
Next time...

- Read Chapter 30, 32
Readers-Writers problem (Solution 1)

- Constraints:
  - Writers can only proceed if there are no readers/writers
  - Readers can proceed only if there are no writers

- use a semaphore BlockWrite
  to block writers when there are readers or a writer

- use a shared variable
  to count the current number of readers

- use semaphore Mutex
  only one process manipulates the shared variable at once

- Initialization:
  - semaphore BlockWrite = 1; // used to allow ONE writer or MANY readers
  - int Readers = 0; // count of readers reading in critical section
  - semaphore Mutex = 1; // binary semaphore (basic lock)
Writer

P(BlockWrite); // wait until there is no readers

< Do the Writing >

V(BlockWrite);
Readers++;

< Do the Reading >

Readers--;
Reader

Readers++;
if (Readers == 1) // first reader needs to wait until no writer
    P(BlockWrite);

< Do the Reading >

Readers--;
if (Readers == 0) // last (only) reader, allows writer to go
    V(BlockWrite);
Reader

P(Mutex); // protect the manipulation of “Readers”
Readers++;
if (Readers == 1) // first reader needs to wait until no writer
    P(BlockWrite);
V(Mutex);

< Do the Reading >

P(Mutex);
Readers--;
if (Readers == 0) // last (only) reader, allows writer to go
    V(BlockWrite);
V(Mutex);
What will happen in different scenarios?

1. The first reader blocks if there is a writer; any other readers who try to enter block on the mutex.
2. The last reader exiting signals a waiting writer.
3. When a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler.
4. If a writer exits and a reader goes next, then all readers that are waiting will fall through.
5. Does this solution guarantee all threads will make progress? Is it fair?

Writes can starve
=> Read preference
What is a good solution?

- Only one thread inside a critical section
- Threads outside of critical section should not block other processes
- No one waits forever (no starvation)
- No assumption about CPU speeds or scheduling decisions
- Works for multiprocessors
Readers-Writers problem (Solution 2)

- How do we let reads yield to writes?
  - semaphore BlockWrite = 1; // used to allow ONE writer or MANY readers
  - int Readers = 0; // count of readers in critical section
  - semaphore RMutex = 1; // binary semaphore for Readers
  - semaphore BlockRead = 1; // used to block readers
  - int Writers = 0; // count of writers in critical section
  - semaphore WMutex = 1; // binary semaphore for Writers
Reader

P(BlockRead); // at most one reader can go before a pending write
P(RMutex);
Readers++;
if (Readers == 1) // first reader needs to wait until no writer
    P(BlockWrite);
V(RMutex);
V(BlockRead);

< Do the Reading >

P(RMutex);
Readers--;
if (Readers == 0) // last (only) reader, allows writer to go
    V(BlockWrite);
V(RMutex);
Write

P(WMutex);
Writers++;
if (Writers == 1) // block readers
    P(BlockRead);
V(Wmutex);

P(BlockWrite); // ensures only one writer
< Do the Writing >
V(BlockWrite);

Any problem?

P(WMutex);
Writers--;
if (Writers == 0) // enable readers
    V(BlockRead);
V(WMutex);
Problem of solution 2

• Reader starvation

• Is there a solution that’s fair to both reads and writes?
  ♦ An idea: use a FIFO queue for all readers and writers
  ♦ Work this out on your own
Semaphore implementation

• Can they be implemented in the user space?
  ♦ An intuitive argument?

• No existing hardware implements them directly
  ♦ Scheduling/queuing cannot be easily done in HW

→ Semaphore must be done in OS, typically with low-level synchronization support from hardware

```c
wait(S) {
    while (S<=0);
    S--;
}
```
Use TAS to implement semaphores on multiprocessor

```c
void wait(semaphore s)
{
    disable interrupts;
    while (1 == tas(&lock,1));
    if (s->count > 0) {
        s->count --;
        lock = 0;
        enable interrupts;
        return;
    }
    add(s->q, current_thread);
    lock=0;
    sleep(); /* re-dispatch */
    enable interrupts;
}

void signal(semaphore s)
{
    disable interrupts;
    while (1 == tas(&lock,1));
    if (isEmpty(s->q) {
        s->count ++;
    } else {
        thread = removeFirst(s->q);
        wakeup(thread);
        /* put thread on Ready Q */
    }
    lock = 0;
    enable interrupts;
}
```
Semaphores in Nachos

To reference current thread: `KThread.currentThread()`

`KThread.sleep()` assumes interrupts are disabled

- Note that interrupts are disabled only to enter/leave critical section
- How can it sleep with interrupts disabled?

```java
P () { // wait
    Disable interrupts;
    if (value == 0) {
        add currentThread to waitQueue;
        KThread.sleep(); // currentThread
    } else {
        value = value - 1;
    }
    Enable interrupts;
}

V () { // signal
    Disable interrupts;
    thread = get next on waitQueue;
    if (thread) {
        thread.ready();
    } else {
        value = value + 1;
    }
    Enable interrupts;
}
```