CSE 120
Principles of Operating Systems

Spring 2023

Lecture 6: Semaphores

Amy Ousterhout

Administrivia

- Project 1
  - Extended, due 5/2
- Homework #2
- Thanks for the #FinAid feedback

Synchronization

- Interleaved executions and shared resources can lead to race conditions
  - Results depend on the timing execution of the code
- Critical sections
  - Sections of code in which only one thread may be executing at a given time
- Locks can solve this problem by providing mutual exclusion

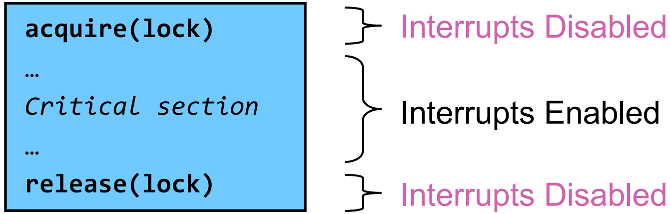
```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

critical section

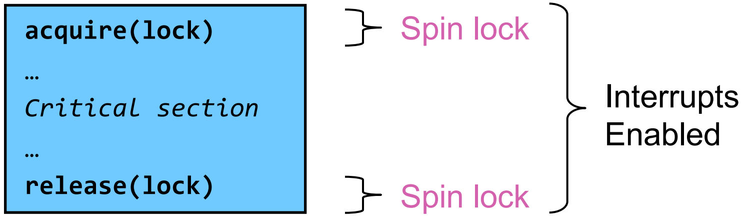
Implementing Locks

- Use a queue to block waiters
- Leave interrupts enabled within the critical section
- Use disabling interrupts or spinning only to protect the critical sections within acquire/release

Implementing a lock by disabling interrupts



Implementing a lock with test_and_set



Synchronization Primitives

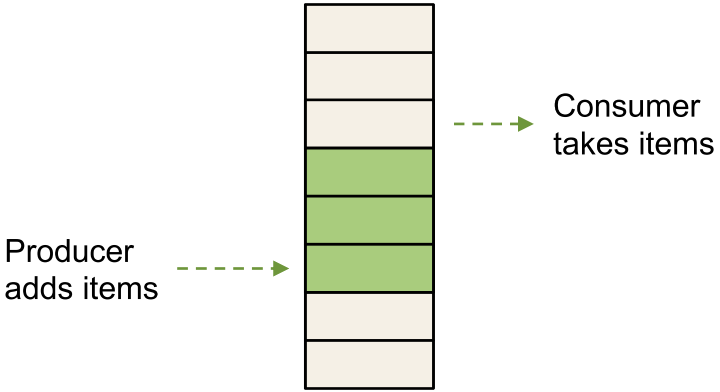
- Locks are useful for implementing critical sections
- But locks have limited semantics
  - Just provide mutual exclusion
- Mutual exclusion does not solve all synchronization problems
- Sometimes we want other semantics, for example:
  - Wait for shared resources to become available
  - Allow multiple threads to generate different resources
  - Use certain conditions to decide when to enter a critical section

Today’s Outline

- Other synchronization primitives
  - Why would we want more than just locks?
- Semaphores
  - What is a semaphore?
  - How can we use them?
  - How can we implement them?

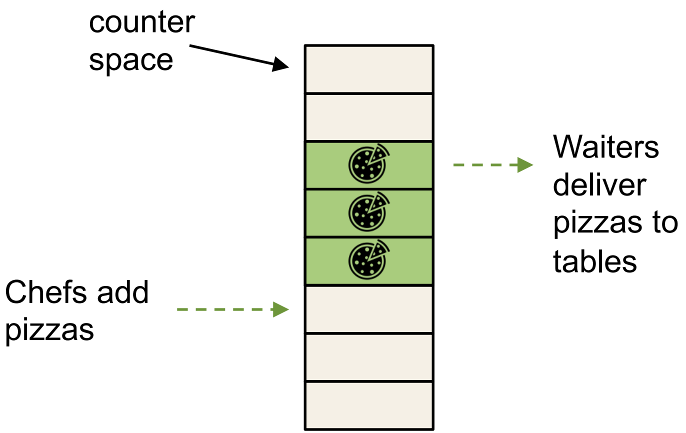
Producer-Consumer Problem

- Also known as the Bounded Buffer problem
- Producer: generates resources
- Consumer: uses up resources
- Buffers: fixed size, used to hold resources between production and consumption



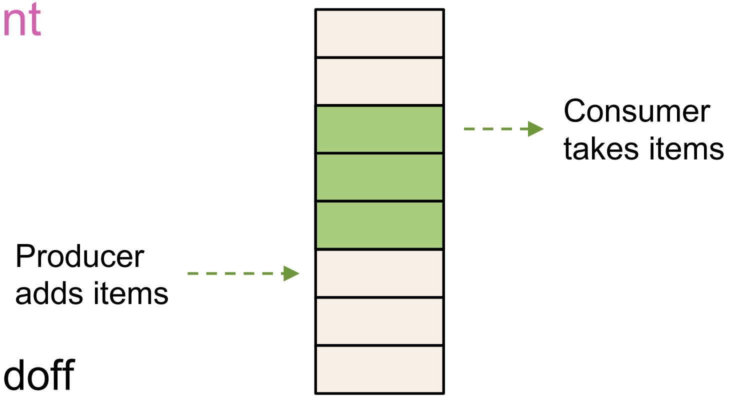
Producer-Consumer Examples

- Real-life example: restaurant
  - Chefs produce pizza
  - Waiters “consume” pizza to deliver it to customers
  - Limited counter space to hold food
- Operating system examples
  - Memory pages
  - Disk blocks
  - I/O



Producer-Consumer Problem

- Producer and consumer can execute at different rates
  - No serialization of one behind the other
  - There can be multiple producers and multiple consumers
  - Tasks are independent
  - The buffer allows each to run without explicit handoff
- Synchronization: ensuring concurrent producers and consumers access the buffer in a correct way
  - What is a “correct way”?

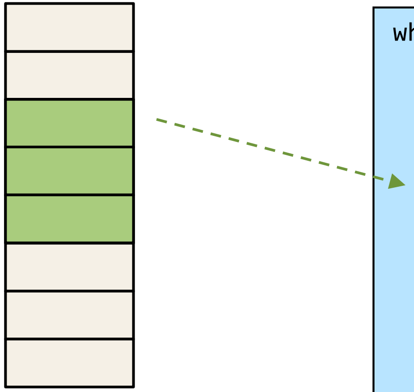


Producer-Consumer

Producer

```
while (1) {
  produce an item


  insert item in buffer
  count++;
}
```



count = 3

Consumer

```
while (1) {


  remove item from buffer
  count--;


  consume an item
}
```

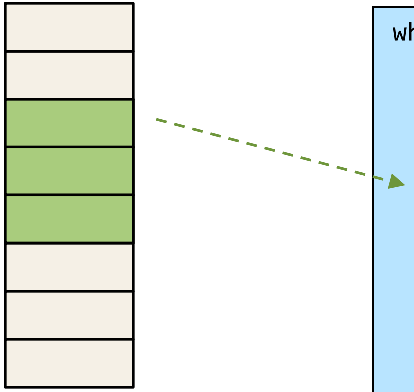
- What's wrong with this naïve solution?

Producer-Consumer with Locks

Producer

```
while (1) {
  produce an item


  acquire(lock);
  insert item in buffer
  count++;
  release(lock);
}
```



count = 3

Consumer

```
while (1) {


  acquire(lock);
  remove item from buffer
  count--;
  release(lock);


  consume an item
}
```

- Use a lock to protect the `count` variable and the buffer
- Does this work?

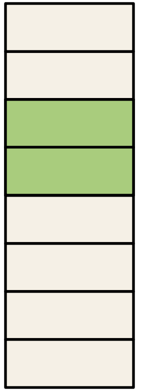
Limitations of Locks

- Locks provide mutual exclusion
  - Only one thread can be in the critical section at one time
- Locks do not provide ordering or sequencing
  - How does the producer know when to stop producing?
  - How does the consumer know when it can consume?

Producer-Consumer with Locks and Sleep/Wake

Producer

```
while (1) {
  produce an item
  if (count == N)
    sleep();
  acquire(lock);
  insert item in buffer
  count++;
  release(lock);
  if (count == 1)
    wakeup(consumer)
}
```



count = 2

N = 8

Consumer

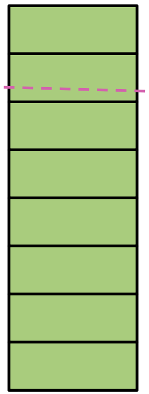
```
while (1) {
  if (count == 0)
    sleep();
  acquire(lock);
  remove item from buffer
  count--;
  release(lock);
  if (count == N-1)
    wakeup(producer)
  consume an item
}
```

- Use sleep/wakeup to manage buffer capacity
- Does this work?

Producer-Consumer with Locks and Sleep/Wake

Producer

```
while (1) {
  produce an item
  if (count == N)
    sleep();
  acquire(lock);
  insert item in buffer
  count++;
  release(lock);
  if (count == 1)
    wakeup(consumer)
}
```



count = 8

N = 8

Consumer

Context switch

```
while (1) {
  if (count == 0)
    sleep();
  acquire(lock);
  remove item from buffer
  count--;
  release(lock);
  if (count == N-1)
    wakeup(producer)
  consume an item
}
```

- Both sleep and never wake up
- Lost the wakeup – is there any way to “remember” it?

Limitations of Locks and Sleep/Wake

- Need a way to count or remember the number of events
- Need more powerful synchronization mechanisms
  - Semaphores
  - Condition variables
  - Monitors
  - Etc.

Today's Outline

- Other synchronization primitives
  - Why would we want more than just locks?
- Semaphores
  - What is a semaphore?
  - How can we use them?
  - How can we implement them?

Semaphores

- A synchronization variable that takes on non-negative integer values
  - Invented by Edsger Dijkstra in the mid 60's
- Semaphores support two operations:
  - `wait()`: an atomic operation that waits for the semaphore to become greater than 0, then decrements it by 1
    - » Also `P()` after the Dutch word for "try to reduce"
  - `signal()`: an atomic operation that increments the semaphore by 1
    - » Also `V()` after the Dutch word for increment
  - Initialize the semaphore to some value
  - Cannot read the semaphore's value directly

Semaphores

- Spinning version

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

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

- Blocking version

```
wait(s) {
  if (s <= 0)
    sleep();
  s--;
}
```

```
signal(s) {
  if (queued thread)
    wakeup();
  s++;
}
```

executed atomically!

Blocking Semaphores

- Each semaphore is associated with a queue of waiting threads
- When `wait()` is called by a thread:
  - If semaphore is open (positive), thread continues
  - If semaphore is closed (non-positive), thread blocks on queue
- The `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
    - `signal()` has "history"
    - The "history" is a counter

```
wait(s) {
  if (s <= 0)
    sleep();
  s--;
}
```

```
signal(s) {
  if (queued thread)
    wakeup();
  s++;
}
```

Semaphore Types

- Semaphores come in two types
- Binary semaphore
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore
  - Represents a resource with many units available
  - Multiple threads can pass the semaphore at once
  - Number of threads determined by the semaphore "count"
- Binary has count = 1, counting has count = N

Semaphore Example: Binary Semaphore

- What happens if initially s = 1 and three threads want to execute:
  - Thread 1: `wait(), …, signal()`
  - Thread 2: `wait(), …, signal()`
  - Thread 3: `wait(), …, signal()`

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

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

Semaphore Example: Binary Semaphore

- Execution, starting with s = 1:
  - Thread 1: `wait(), …, signal()`
  - Thread 2: `wait()` `…, signal()`
  - Thread 3: `wait()` `…, signal()`
- The semaphore behaves like a lock!

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

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

Semaphore Example: Counting Semaphore

- What happens if initially s = 2 and three threads want to execute:
  - Thread 1: `wait(), …, signal()`
  - Thread 2: `wait(), …, signal()`
  - Thread 3: `wait(), …, signal()`

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

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

Semaphore Example: Counting Semaphore

- Execution, starting with s = 2:
  - Thread 1: `wait(), …, signal()`
  - Thread 2: `wait(), …, signal()`
  - Thread 3: `wait()` `…, signal()`
- Multiple threads can run at once

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

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

Benefits of Semaphores over Locks

- Semaphores have a value, enabling more semantics:
  - When at most one, can be used for mutual exclusion (only 1 thread in a critical section)
  - When greater than 1, can allow multiple threads to access resources
- Two use cases:
  - Mutual exclusion – only 1 thread accessing a resource at a time
  - Event sequencing – permit threads to wait for certain things to happen

Today's Outline

- Other synchronization primitives
  - Why would we want more than just locks?
- Semaphores
  - What is a semaphore?
  - How can we use them?
  - How can we implement them?

Producer-Consumer with Semaphores

- `signal(s)` increments s
  - "just produced an item"
  - s value = how many items have been produced
- `wait(s)` will return without waiting only if s > 0
  - "wait until there is at least one item and then consume one item"
- What resources are we producing/consuming?
  - Items and empty spaces

Producer-Consumer with Semaphores

- Two constraints:
  - Consumer must wait for the producer to produce items
  - Producer must wait for the consumer to empty spaces
- Use a separate semaphore for each constraint:
  - `full_count = 0`
  - `empty_count = N`

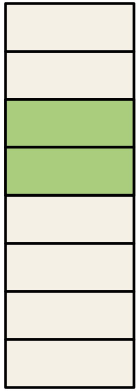
Producer-Consumer with Semaphores

Producer

```
while (1) {
  produce an item
  wait(empty_count)


  insert item in buffer
  count++;

  signal(full_count)
}
```



count = 2

N = 8

Consumer

```
while (1) {
  wait(full_count)


  remove item from buffer
  count--;

  signal(empty_count)

  consume an item
}
```

- Initialization: full_count = 0, empty_count = N
- Does this work?

Producer-Consumer with Semaphores

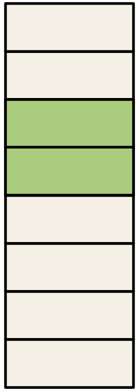
- Three constraints:
  - Consumer must wait for the producer to produce items
  - Producer must wait for the consumer to empty spaces
  - Only one thread can manipulate the buffer at once
- Use a separate semaphore for the first two constraints:
  - `full_count = 0`
  - `empty_count = N`
- And a lock or semaphore for the third

Producer-Consumer with Semaphores

Producer

```
while (1) {
  produce an item
  wait(empty_count)

  acquire(lock);
  insert item in buffer
  count++;
  release(lock);
  signal(full_count)
}
```



count = 2

N = 8

Consumer

```
while (1) {
  wait(full_count)

  acquire(lock);
  remove item from buffer
  count--;
  release(lock);
  signal(empty_count)

  consume an item
}
```

- Does this work?
- Yes!

Readers-Writers Problem

- An object is shared among several threads
- Some threads only read the object, others only write it
- We can allow multiple readers but only one writer
- Used with many data objects
  - Bank account example
  - Linked list, tree, …

Readers-Writers with Semaphores

- Constraints:
  - Writers can only proceed if there are no readers or writers
  - Readers can only proceed if there are no writers
- How can we use semaphores to implement this protocol?
- Use three variables:
  - int read_count: number of threads currently reading
  - semaphore mutex: lock to control access to read_count
  - semaphore block_write: allows one writer or many readers

Readers-Writers with Semaphores

Initialization

```
int read_count = 0;
semaphore mutex = 1;
semaphore block_write = 1;
```

Writer

```
write() {
  wait(block_write);

  do the writing

  signal(block_write);
}
```

wait until there are no readers or writers (→ `wait(block_write);`)

Reader

```
read() {
  wait(mutex);
  read_count++;
  if (read_count == 1)
    wait(block_write);
  signal(mutex);

  do the reading

  wait(mutex);
  read_count--;
  if (read_count == 0)
    signal(block_write);
  signal(mutex);
}
```

are we the first reader? (→ `if (read_count == 1)`)

wait until there are no writers (→ `wait(block_write);`)

are we the last reader? (→ `if (read_count == 0)`)

let a writer run (→ `signal(block_write);`)

- When there's a writer, where do readers block?
- Which reader runs first after a writer?
- If multiple readers, will they all run before a writer?
- Is this approach fair?

Today's Outline

- Other synchronization primitives
  - Why would we want more than just locks?
- Semaphores
  - What is a semaphore?
  - How can we use them?
  - How can we implement them?

Implementing Semaphores

- Use a queue to block waiters, guard on lock, and a count of waiters

```
struct semaphore {
  int count = 1;
  bool guard = False;
  queue Q;
}
```

```
void wait(s) {
  while (test_and_set(&s->guard));
  if (s->count <= 0) {
    put current thread on s->Q;
    block current thread and
        s->guard = False;
  }
  s->count--;
  s->guard = False;
}
```

```
void signal(s) {
  while (test_and_set(&s->guard));
  if (s->Q is empty)
      s->count++;
  else
     move a waiting thread to the ready
queue;
  s->guard = False;
}
```

- Similar to implementing a lock (check!) but we need to maintain the count

Semaphore Summary

- Semaphores can be used to solve traditional synchronization problems
  - For example: Producer-Consumer and Reader-Writer
  - Enforce critical sections (mutual exclusion)
  - Enable coordination between threads (scheduling)
- But they have some drawbacks:
  - No coordination between the semaphore and the controlled data
  - Used for both critical sections and coordination - this can be confusing!
  - Sometimes hard to use and prone to bugs
- What can we do instead?
  - Next week…

For next class…

- Read chapters 30 and 32