CS 344: Operating Systems I
03.06: Part IV – Semaphore

Mon/Wed 12:00 – 1:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

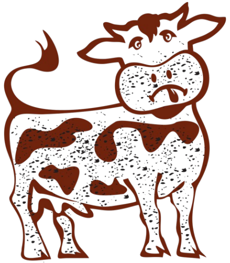
Oregon State University

SAIL
Secure AI Systems Lab

Notice

- Announcements
  - Extra credit opportunities on Canvas (12%)
    - Rust Programming Practice (+2%)
    - Build an ML classifier (+2%)
    - Multi-process data loader (+3%)
    - Some articles about Linus Torvalds (+5%)

Race condition often creates a security vulnerability



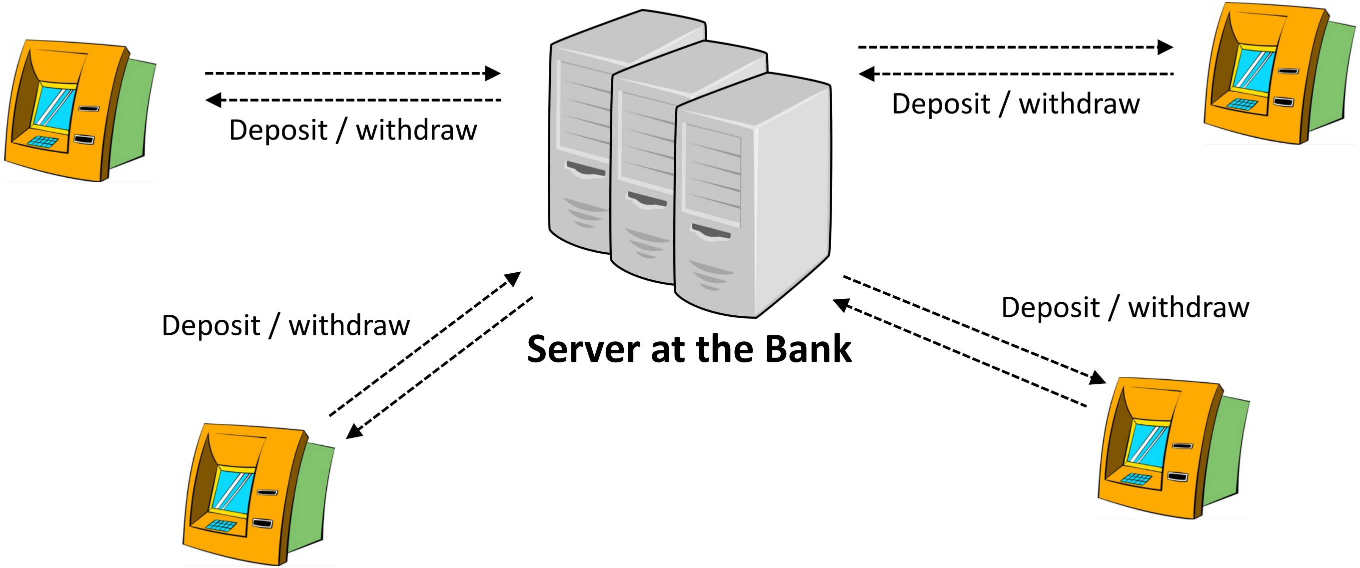
DIRTY COW Vulnerability ([Video])

Topics for today

- Part IV – Synchronization II
  - Recap:
    - Race condition (ATM server’s problem)
    - Mutual exclusion (Mutex)
  - Manage resources
    - Producer-consumer problem (Coke machine)
    - Deadlock
    - Semaphore
  - Provide abstraction & Offer standard interface
    - Semaphore
    - Semaphore in C

SYNCHRONIZATION

- **ATM bank's server**
  - The server(s) takes care of multiple deposit / withdrawal requests
  - Bank want to make sure all the transactions are correct



Synchronization: ATM bank server v0.1

- **Server in C**
  - Receive a request
  - Process the request
  - Perform those actions *iteratively*

- **Potential problem**
  - A single request at a time
  - Problem: ~470k ATMs in the US (2018)

```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         Deposit(accountId, amount);
      case OP_WITHDRAW:
         Withdraw(accountId, amount);
         … <here, you can define more ops…>
   }
}

void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId = -1;
   int amount = -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;        // code only reaches here if the server terminates
}
```

Synchronization: ATM bank server v0.2

- **Event-driven ATM bank server**
  - Receive/process events
  - Store them to a buffer
  - Deposit when "account" is available
- **Potential problem:**
  - Increase implementation complexity
  - How many events do we need?

```
struct Event {
   int eventType;
   int accountId;
   int amount;
   struct account* account;
};

void PullAccount(struct Event* event) {
   event->account = GetAccount(event->accountId);
}

void Deposit(struct Event* event) {
   event->account->balance += event->amount;
   event->amount = 0;
}

int main(void) {
   …

   while (1) {
      event = Wait4NextEvent();
      if (event->eventType == RequestReceived)      PullAccount(event);
      else if (event->eventType == DepositReady)    Deposit(event);
   }

   return 0;        // code only reaches here if the server terminates
}
```

SYNCHRONIZATION: ATM BANK SERVER V0.3

- **Threaded ATM bank server**
  - Receive a request
  - Create a thread for processing it
  - Multiple threads can co-exist

- **Potential problem:**

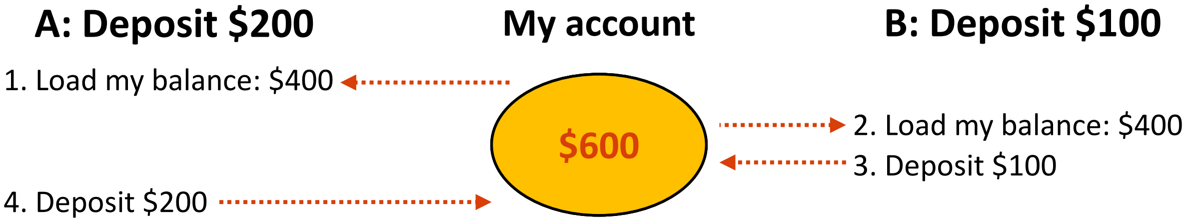
Thread A	**Thread B**
1. Load my balance: $400	
	2. Load my balance: $400
	3. Deposit $100
4. Deposit $200	

Now, What's My Balance?

```
void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Deposit, info);
      case OP_WITHDRAW:
         pthread_t *newTh = <mem alloc>;
         pthread_create(newTh, Withdraw, info);
   }
}

void Deposit(accountId, amount) {
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;          // code only reaches here if the server terminates
}
```

Synchronization: Race Condition

- **Race condition:**
  - **Definition:** an undesirable scenario; performs multiple operations on a shared resource
  - **Example:** two “deposit” threads, running *concurrently*, increase the balance



How Can We Make Sure My Balance Is $700 at the End?

Synchronization: atomic operation

- **Solution approach:**
  - Deposit() is not *indivisible*
  - Make sure to execute "Deposit()" at once
- **Atomic operation:**
  - Code should be executed w/o interrupt
  - **TL; DR:** Code should be run *at once*

```
void ProcessRequest(op, accountId, amount) {
  switch (op) {
    case OP_DEPOSIT:
      pthread_t *newTh = <mem alloc>;
      pthread_create(newTh, Deposit, info);
    case OP_WITHDRAW:
      pthread_t *newTh = <mem alloc>;
      pthread_create(newTh, Withdraw, info);
  }
}

void Deposit(accountId, amount) {
  account = GetAccount(accountId);
  account->balance += amount;
  StoreAccount(account);
}

int main(void) {
  int op = -1;
  int accountId, amount = -1, -1;

  while (1) {
    ReceiveRequest(&op, &accountId, &amount);
    ProcessRequest(op, accountId, amount);
  }

  return 0;       // code only reaches here if the server terminates
}
```

Synchronization: mutual exclusion (mutex)

- Mutex (lock)
  - Prevents two+ process access the code
  - Supports three operations
    - Lock before running atomic code
    - Unlock after running the code
    - Wait while someone locked the code

```
pthread_mutex_t deposit_lock;

void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         …
   }
}

void Deposit(accountId, amount) {
   pthread_mutex_lock(&foo_mutex);         // lock before the atomic op.
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
   pthread_mutex_unlock(&foo_mutex);       // unlock after the atomic op.
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;
   pthread_mutex_init(&deposit_lock, NULL);

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;         // code only reaches here if the server terminates
```

Synchronization: mutual exclusion (mutex)

- Mutex (lock)
  - Prevents two+ process access the code
  - Supports three operations
    - Lock before running atomic code
    - Unlock after running the code
    - Wait while someone locked the code
- Critical section
  - A code section protected by lock & unlock

```
pthread_mutex_t deposit_lock;

void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         …
   }
}

void Deposit(accountId, amount) {
   pthread_mutex_lock(&foo_mutex);          // lock before the atomic op.
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
   pthread_mutex_unlock(&foo_mutex);        // unlock after the atomic op.
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;
   pthread_mutex_init(&deposit_lock, NULL);

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;          // code only reaches here if the server terminates
```

Synchronization: Mutual Exclusion (Mutex)

- Mutex (lock)
  - Prevents two+ process access the code
  - Supports three operations
    - Lock before running atomic code
    - Unlock after running the code
    - Wait while someone locked the code
- Critical section
  - A code section protected by lock & unlock
- Note
  - Must use the *same* lock for a critical section
  - Must be careful in declaring a critical section
    - What if lock and sleep(10000000000);

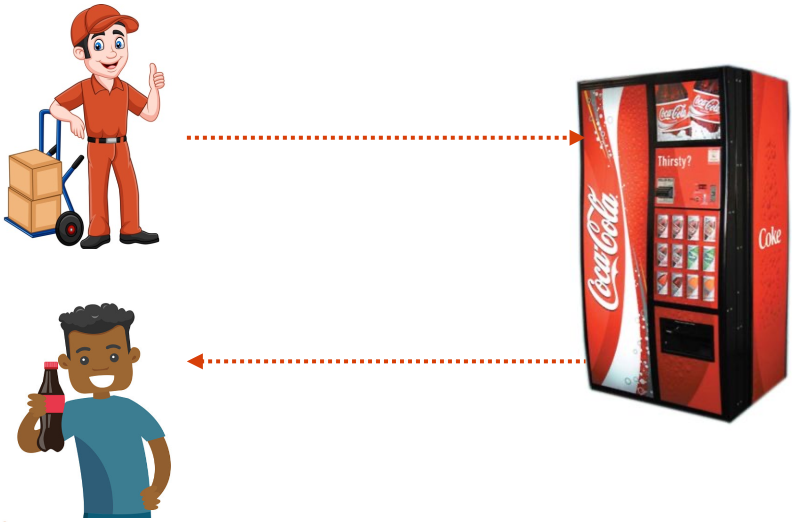
```
pthread_mutex_t deposit_lock;

void ProcessRequest(op, accountId, amount) {
   switch (op) {
      case OP_DEPOSIT:
         …
   }
}

void Deposit(accountId, amount) {
   pthread_mutex_lock(&foo_mutex);        // lock before the atomic op.
   account = GetAccount(accountId);
   account->balance += amount;
   StoreAccount(account);
   pthread_mutex_unlock(&foo_mutex);      // unlock after the atomic op.
}

int main(void) {
   int op = -1;
   int accountId, amount = -1, -1;
   pthread_mutex_init(&deposit_lock, NULL);

   while (1) {
      ReceiveRequest(&op, &accountId, &amount);
      ProcessRequest(op, accountId, amount);
   }

   return 0;         // code only reaches here if the server terminates
```

Topics for today

- Part IV – Synchronization II
  - Recap:
    - Race condition (ATM server's problem)
    - Mutual exclusion (Mutex)
  - Manage resources
    - Producer-consumer problem (Coke machine)
    - Deadlock
    - Semaphore
  - Provide abstraction & Offer standard interface
    - Semaphore
    - Semaphore in C

Synchronization problem: a coke machine

- **A coke machine**
  - Two workers (or threads):
    - Producer: fills the coke machine
    - Consumer: takes cokes from the machine



Synchronization problem: a coke machine v0.1

- **Coke machine in C**
  - A coke machine (can hold 64 cokes)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke
- **Problem I:**
  - Producer puts cokes when it's full
  - Consumer takes a coke when it's empty

```c
#define  MACHINE_CAPACITY        64
static struct coke_machine;

void producer_fn() {
   while (1) {
      enqueue(acoke, coke_machine);
   }
}

void consumer_fn() {
   while (1) {
      acoke = dequeue(coke_machine);
   }
}

int main(void) {
   pthread_t producer, consumer;

   pthread_create(&producer, NULL, producer_fn, NULL);
   pthread_create(&consumer, NULL, consumer_fn, NULL);

   pthread_join(producer, NULL);
   pthread_join(consumer, NULL);

   return 0;                  // code only reaches here if the machine is broken
}
```

Synchronization problem: a coke machine v0.2

- **Coke machine in C**
  - A coke machine (can hold 64 cokes)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke

- **Problem I:**
  - Producer puts cokes when it's full
  - Consumer takes a coke when it's empty

- **Solution I:**
  - Busy-waiting (or spinning)
  - *Repeatedly* checks if a condition is true

```
#define  MACHINE_CAPACITY        64
static struct coke_machine;

void producer_fn() {
   while (1) {
      while (machine == full) {};
      enqueue(acoke, coke_machine);
   }
}

void consumer_fn() {
   while (1) {
      while (machine == empty) {};
      acoke = dequeue(coke_machine);
   }
}

int main(void) {
   pthread_t producer, consumer;

   pthread_create(&producer, NULL, producer_fn, NULL);
   pthread_create(&consumer, NULL, consumer_fn, NULL);

   pthread_join(producer, NULL);
   pthread_join(consumer, NULL);

   return 0;                    // code only reaches here if the machine is broken
}
```

Synchronization problem: a coke machine v0.2

- **Coke machine in C**
  - A coke machine (can hold 64 cokes)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke
- **Problem II:**
  - Race condition can occur
  - Total # cokes can be incorrect
    - Producer gets the coke #
    - Consumer takes a coke
    - Producer increases the coke # by 3
    - Ugh…

```
#define  MACHINE_CAPACITY        64
static struct coke_machine;

void producer_fn() {
   while (1) {
      while (machine == full) {};
      enqueue(acoke, coke_machine);
   }
}

void consumer_fn() {
   while (1) {
      while (machine == empty) {};
      acoke = dequeue(coke_machine);
   }
}

int main(void) {
   pthread_t producer, consumer;

   pthread_create(&producer, NULL, producer_fn, NULL);
   pthread_create(&consumer, NULL, consumer_fn, NULL);

   pthread_join(producer, NULL);
   pthread_join(consumer, NULL);

   return 0;                 // code only reaches here if the machine is broken
}
```

```
ncoke = count(coke_machine)
ncoke += 1
coke_machine->cokes = ncoke
```

```
ncoke = count(coke_machine)
ncoke -= 1
coke_machine->cokes = ncoke
```

Synchronization problem: a coke machine v0.3

- **Coke machine in C**
  - A coke machine (can hold 64 cokes)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke

- **Problem III:**
  - Deadlock can occur
  - **Def.:** a scenario where *no* thread can continue running b/c locks
  - Suppose:
    - Producer locks it when it's full, or
    - Consumer locks it when it's empty

```
#define  MACHINE_CAPACITY        64
static struct coke_machine;

void producer_fn() {
   while (1) {
      pthread_mutex_lock(&machine);
      while (machine == full) {};
      enqueue(acoke, coke_machine);
      pthread_mutex_unlock(&machine);
   }
}

void consumer_fn() {
   while (1) {
      pthread_mutex_lock(&machine);
      while (machine == empty) {};
      acoke = dequeue(coke_machine);
      pthread_mutex_unlock(&machine);
   }
}

int main(void) {
   pthread_t producer, consumer;

   ….

   return 0;                    // code only reaches here if the machine is broken
}
```

Topics for today

- Part IV – Synchronization II
  - Recap:
    - Race condition (ATM server's problem)
    - Mutual exclusion (Mutex)
  - Manage resources
    - Producer-consumer problem (Coke machine)
    - Deadlock
    - Semaphore
  - Provide abstraction & Offer standard interface
    - Semaphore
    - Semaphore in C

Synchronization problem: a coke machine v0.4

- **Coke machine in C**
  - A coke machine (can hold 64 cokes)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke
- **Solution III:**
  - Do *not* include busy-waiting code inside the critical sections

```
#define  MACHINE_CAPACITY        64
static struct coke_machine;

void producer_fn() {
   while (1) {
      while (machine == full) {};
      pthread_mutex_lock(&machine);
      enqueue(acoke, coke_machine);
      pthread_mutex_unlock(&machine);
   }
}

void consumer_fn() {
   while (1) {
      while (machine == empty) {};
      pthread_mutex_lock(&machine);
      acoke = dequeue(coke_machine);
      pthread_mutex_unlock(&machine);
   }
}

int main(void) {
   pthread_t producer, consumer;

   ….

   return 0;                     // code only reaches here if the machine is broken
}
```

Synchronization problem: a coke machine v0.4

- **Coke machine in C**
  - A coke machine (can hold 64 cokes)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke

- **Problem IV:**
  - Producer/consumer can wait forever
  - "Busy-wait" does *not* guarantee running

```
#define  MACHINE_CAPACITY        64
static struct coke_machine;

void producer_fn() {
   while (1) {
      while (machine == full) {};
      pthread_mutex_lock(&machine);
      enqueue(acoke, coke_machine);
      pthread_mutex_unlock(&machine);
   }
}

void consumer_fn() {
   while (1) {
      while (machine == empty) {};
      pthread_mutex_lock(&machine);
      acoke = dequeue(coke_machine);
      pthread_mutex_unlock(&machine);
   }
}

int main(void) {
   pthread_t producer, consumer;

   ….

   return 0;                    // code only reaches here if the machine is broken
}
```

Synchronization problem: a coke machine v0.4

- **Coke machine in C**
  - A coke machine (can hold 64 cokes)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke

- **Problem IV:**
  - Producer/consumer can wait forever
  - "Busy-wait" does *not* guarantee running

```
#define  MACHINE_CAPACITY        64
static struct coke_machine;

void producer_fn() {
   while (1) {
      while (machine == full) {};
      pthread_mutex_lock(&machine);
      enqueue(acoke, coke_machine);
      pthread_mutex_unlock(&machine);
   }
}

void consumer_fn() {
   while (1) {
      while (machine == empty) {};
      pthread_mutex_lock(&machine);
      acoke = dequeue(coke_machine);
      pthread_mutex_unlock(&machine);
   }
}

int main(void) {
   pthread_t producer, consumer;

   ....

   return 0;                    // code only reaches here if the machine is broken
}
```

Synchronization: semaphore

- Semaphore
  - **Definition:** a variable used to control access to a shared resource
  - **TL; DR:** Mutex + Variable + Signal

- Semaphore operations
  - **P():** *wait* until a semaphore becomes positive and *decrease* it by 1
  - **V():** *increase* a semaphore by 1 and *wake up* any thread that waits by P()

Synchronization: A coke machine v0.5

- **Coke machine in C**
  - A coke machine (can hold 64 cokes)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke

- **Solution IV:**
  - Use semaphore
  - P() is sem_wait()
  - V() is sem_post()

```
sem_t mutex;
sem_t slots_filled;
sem_t slots_empty;

void producer_fn() {
  while (1) {
    sem_wait(&slots_empty);
    sem_wait(&mutex);
    enqueue(acoke, coke_machine);
    sem_post(&mutex);
    sem_post(&slots_filled);
  }
}

void consumer_fn() {
  while (1) {
    sem_wait(&slots_filled);
    sem_wait(&mutex);
    acoke = dequeue(coke_machine);
    sem_post(&mutex);
    sem_post(&slots_empty);
  }
}

int main(void) {
  int ret;
  ret = sem_init(&mutex, 0, 1);
  ret = sem_init(&slots_empty, 0, 64);
  ret = sem_init(&slots_filled, 0, 0);
  ....

}
```

The semaphore only allows one thread to enqueue (or dequeue)

It decreases “filled slot” by one

It increases “empty slot” by one, and wakes up any thread *(i.e., producer thread)* by sending a *signal* to that thread

Initialize with the # resources
1) Mutex := lock := 1
2) Empty slots := 64 (capacity)
3) Filled slots := 0 (empty at first)

SYNCHRONIZATION: A COKE MACHINE V0.5

- **Possible scenario I**
  - Initially the coke machine is empty
  - Consumer tries to get a coke
    - It decreases “slots_filled” by one
    - “slots_filled” becomes -1
    - The thread sleeps
  - Producer runs
    - It decreases “slots_empty” by one
    - It adds a coke to the machine
    - It signals the thread waiting by “slots_filled”
  - Consumer wakes up and run

```
sem_t mutex;
sem_t slots_filled;
sem_t slots_empty;

void producer_fn() {
   while (1) {
      sem_wait(&slots_empty);
      sem_wait(&mutex);
      enqueue(acoke, coke_machine);
      sem_post(&mutex);
      sem_post(&slots_filled);
   }
}

void consumer_fn() {
   while (1) {
      sem_wait(&slots_filled);
      sem_wait(&mutex);
      acoke = dequeue(coke_machine);
      sem_post(&mutex);
      sem_post(&slots_empty);
   }
}

int main(void) {
   int ret;
   ret = sem_init(&mutex, 0, 1);
   ret = sem_init(&slots_empty, 0, 64);
   ret = sem_init(&slots_filled, 0, 0);
   ....

}
```

The semaphore only allows one thread to enqueue (or dequeue)

It decreases “filled slot” by one

It increases “empty slot” by one, and wakes up any thread *(i.e., producer thread)* by sending a *signal* to that thread

Initialize with the # resources
1) Mutex := lock := 1
2) Empty slots := 64 (capacity)
3) Filled slots := 0 (empty at first)

Synchronization: a coke machine v0.5

- **Possible scenario II**
  - The coke machine is full
  - Producer tries to put a coke
    - It decreases "slots_empty" by one
    - "slots_empty" becomes -1
    - The thread sleeps
  - Consumer runs
    - It decreases "slots_filled" by one
    - It gets a coke from the machine
    - It signals the thread waiting by "slots_empty"
  - Producer wakes up and run

```
sem_t mutex;
sem_t slots_filled;
sem_t slots_empty;

void producer_fn() {
  while (1) {
    sem_wait(&slots_empty);
    sem_wait(&mutex);
    enqueue(acoke, coke_machine);
    sem_post(&mutex);
    sem_post(&slots_filled);
  }
}

void consumer_fn() {
  while (1) {
    sem_wait(&slots_filled);
    sem_wait(&mutex);
    acoke = dequeue(coke_machine);
    sem_post(&mutex);
    sem_post(&slots_empty);
  }
}

int main(void) {
  int ret;
  ret = sem_init(&mutex, 0, 1);
  ret = sem_init(&slots_empty, 0, 64);
  ret = sem_init(&slots_filled, 0, 0);
  ….

}
```

The semaphore only allows one thread to enqueue (or dequeue)

It decreases "filled slot" by one

It increases "empty slot" by one, and wakes up any thread *(i.e., producer thread)* by sending a *signal* to that thread

Initialize with the # resources
1) Mutex := lock := 1
2) Empty slots := 64 (capacity)
3) Filled slots := 0 (empty at first)

SYNCHRONIZATION: A COKE MACHINE V0.5

- Order of P() and V()
  - Switch sem_wait()s in Producer
    - Producer locks "mutex"
    - Producer waits for "empty slot"
    - Consumer can't dequeue b/c of "mutex"
    - Deadlock!
  - Switch sem_post()s in Producer
    - Producer decreases "slots_filled"
    - Consumer wakes up
    - Consumer can't dequeue
    - Producer unlocks the mutex
    - Runs… (no deadlock)

Care Must Be Taken!

```
sem_t mutex;
sem_t slots_filled;
sem_t slots_empty;

void producer_fn() {
  while (1) {
    sem_wait(&mutex);
    sem_wait(&slots_empty);
    enqueue(acoke, coke_machine);
    sem_post(&slots_filled);
    sem_post(&mutex);
  }
}

void consumer_fn() {
  while (1) {
    sem_wait(&slots_filled);
    sem_wait(&mutex);
    acoke = dequeue(coke_machine);
    sem_post(&mutex);
    sem_post(&slots_empty);
  }
}

int main(void) {
  int ret;
  ret = sem_init(&mutex, 0, 1);
  ret = sem_init(&slots_empty, 0, MACHINE_CAPACITY);
  ret = sem_init(&slots_filled, 0, 0);

  ……                    // omit the pthread_create / _join
}
```

Topics for today

- Part IV – Synchronization II
  - Recap:
    - Race condition (ATM server’s problem)
    - Mutual exclusion (Mutex)
  - Manage resources
    - Producer-consumer problem (Coke machine)
    - Deadlock
    - Semaphore
  - Provide abstraction & Offer standard interface
    - Semaphore
    - Semaphore in C

Thank You!

Mon/Wed 12:00 – 1:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab