CS 344: Operating Systems I

03.08: Part IV – Monitor

Mon/Wed 12:00 – 1:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

Notice

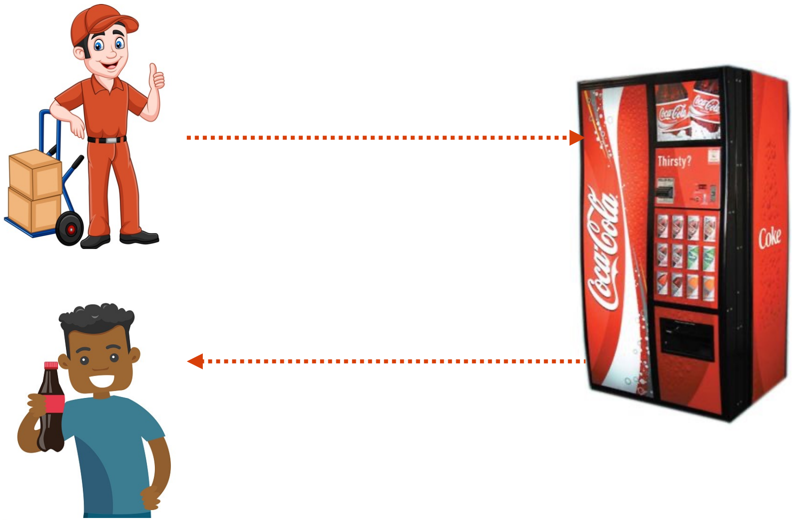
- Announcements
  - Extra credit opportunities on Canvas (12%)
    - Rust Programming Practice (+2%)
    - Build an ML classifier (+2%)
    - Multi-process data loader (+3%)
    - Some articles about Linus Torvalds (+5%)

Topics for today

- Part IV – Synchronization III
  - Recap:
    - Coke machine problem (deadlock)
    - Semaphore (in C)
  - Monitor
    - Meeting room booking system
    - Monitor
    - Monitor implemented in C

Coke machine (producer-consumer) problem

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



Coke machine v0.1

- **Coke machine in C**
  - A bounded buffer (64 coke slots)
  - Two workers (or threads):
    - Producer thread puts cokes
    - Consumer thread gets a coke
- **Problem I:**
  - Producer puts a coke 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
}
```

Coke machine v0.2

- **Coke machine in C**
  - A bounded buffer (64 coke slots)
  - 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…

```c
#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
}
```

enqueue(acoke, coke_machine):

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

acoke = dequeue(coke_machine):

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

Coke machine v0.3

- **Solution II:**
  - **Mutex**
  - En-/de-queue are atomic operations
  - Make them into critical sections

- **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
}
```

Coke machine v0.4

- **Solution III:**
  - **Mutex**
  - But do not include busy-waiting
- **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
}
```

Coke machine v0.5

- **Solution IV:**
  - **Semaphore**
  - Mutex + Variable (counter) + Signal
  - Supported operations
    - P() decrease the var. by 1
    - P() waits until the var. becomes positive
    - V() increases the var. by 1
    - V() wake up any threads that waits by P()

```
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)

Coke machine v0.5

- **Example scenario I:**
  - Producer tries to put cokes
  - Consumer is getting a coke at that time
  - Producer waits (lock mutex)
  - Consumer gets the coke (unlock)
  - (Unlock) signals the producer
  - Producer puts the cokes

```
sem_t mutex;              // mutex = semaphore with the conditional var 1
sem_t slots_filled;
sem_t slots_empty;

void producer_fn() {
   while (1) {
      sem_wait(&slots_empty);
      sem_wait(&mutex);        <-- 1) wait while any consumer is dequeuing
      enqueue(acoke, coke_machine);
      sem_post(&mutex);
      sem_post(&slots_filled);
   }
}
                                   3) signals the waiting thread(s)
void consumer_fn() {
   while (1) {
      sem_wait(&slots_filled);
      sem_wait(&mutex);
      acoke = dequeue(coke_machine);
      sem_post(&mutex);        <-- 2) finish dequeuing
      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
}
```

Coke machine v0.5

- **Remember:**
  - Order matters
    - Flip "slots_empty" and "mutex"
    - It can lead to the deadlock

```
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(&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, MACHINE_CAPACITY);
   ret = sem_init(&slots_filled, 0, 0);

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

Deadlock scenario:

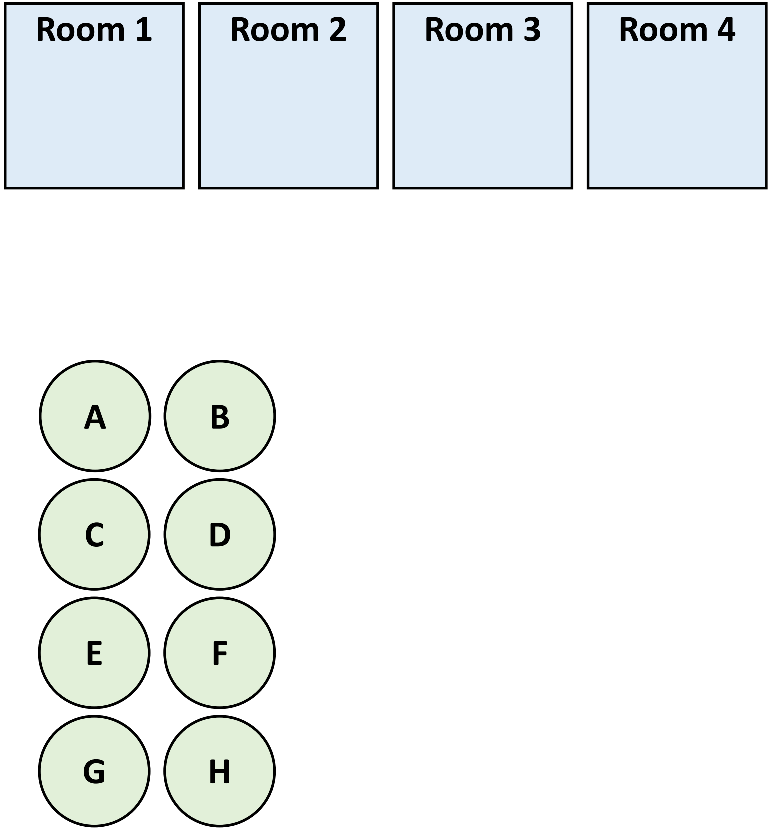
1) Producer locks the mutex
2) Producer waits for an empty slot
3) Consumer can't dequeue
4) Deadlock!

Topics for today

- Part IV – Synchronization III
  - Recap:
    - Coke machine problem (deadlock)
    - Semaphore (in C)
  - Monitor
    - Meeting room booking system
    - Monitor
    - Monitor implemented in C

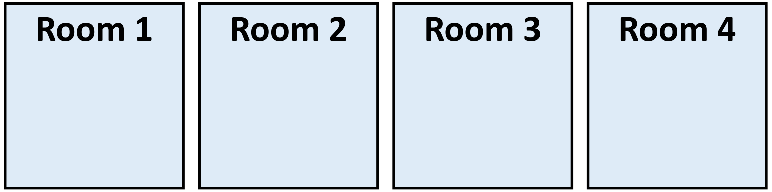
Meeting room booking system

- **Scenario**
  - A bounded buffer (4 rooms)
  - Only one person can be in a room
  - No room left: anyone should wait



Meeting room booking system: semaphore

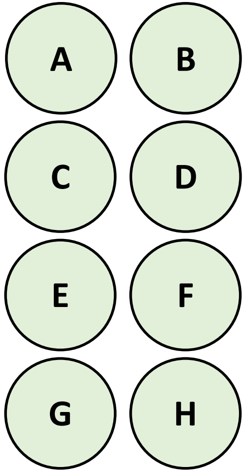
- **Solution**
  - **Semaphore**
  - 4 (conditional) variables := 4 rooms



Mutex: no one can enter if someone is in the room

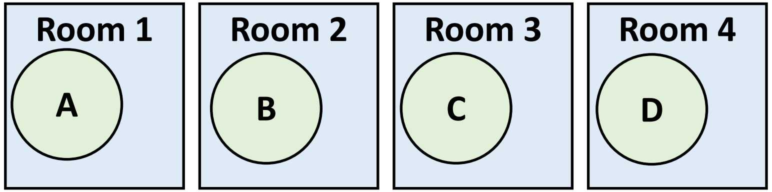
- **Illustrative example:**
  - 8 employees (A-H) uses the 4 rooms

Semaphore
> Conditional var: **0**
(No rooms are available)



Meeting room booking system: semaphore

- **Solution**
  - **Semaphore**
  - 4 (conditional) variables := 4 rooms

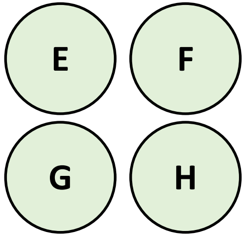


Mutex: no one can enter if someone is in the room

- **Illustrative example:**
  - A moves out from Room 1
  - Semaphore increases from 0 to 1
  - It signals to all waiting process (E-H)
  - OS picks one (H) to run
  - H enters Room 1
  - Semaphore decreases to 0

Semaphore
> Conditional var: **0**

Signal: once Room A is empty, OS lets others (E, F, G, H) its empty



Semaphore is good, but...

- **Potential problems:**
  - Semaphore offers lock and scheduling together
  - Make it hard to
    - Check the implementation correctness
    - Implement fine-grained scheduling controls

- **Solution:**
  - Decompose the lock and scheduling
  - It typically requires a user-level implementation (ex. in C)

Monitor

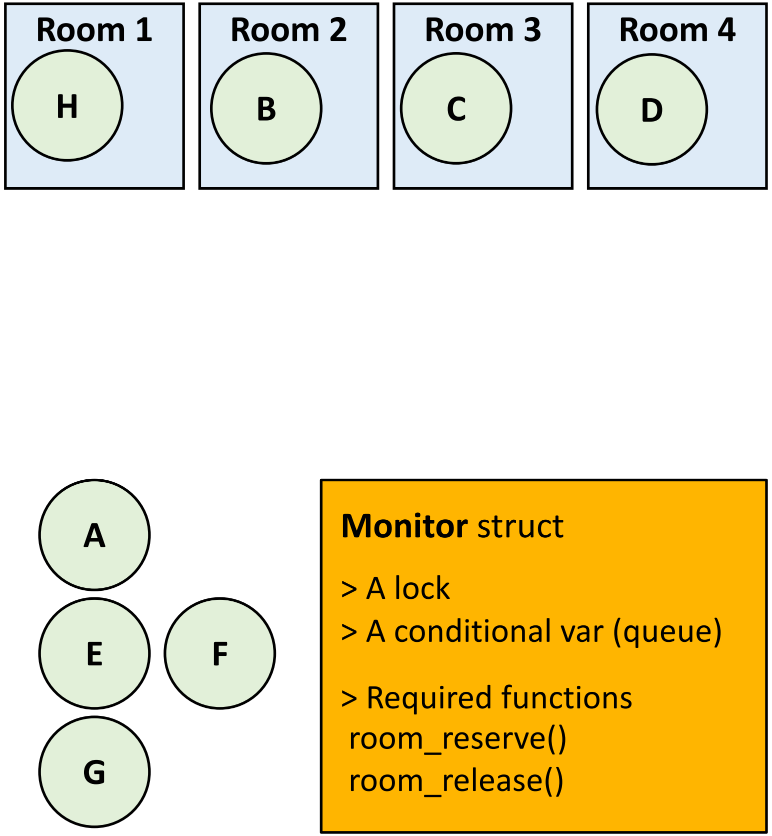
- Monitor:
  - **Def:** a synchronization *object*
    - Conditional variable
    - Monitoring mechanism

- Supported operations:
  - wait(&lock): release lock and sleep
  - signal(): wake up one waiting worker
  - broadcast(): wake up *all* waiting jobs

Monitor

- Monitor:
  - **Def:** a synchronization *object*
    - Conditional variable
    - Monitoring mechanism

- Supported operations:
  - wait(&lock): release lock and sleep
  - signal(): wake up one waiting worker
  - broadcast(): wake up *all* waiting jobs



Monitor in C

- Monitor:
  - **Def:** a synchronization *object*
    - Conditional variable
    - Monitoring mechanism

- Supported operations:
  - wait(&lock): release lock and sleep
  - signal(): wake up one waiting worker
  - broadcast(): wake up *all* waiting jobs

monitor.h

```c
#ifndef  MONITOR_H
#define  MONITOR_H

#define  NUM_ROOMS     4

void reserve_a_room(int room_num, struct user_t* employee);
struct user_t* release_a_room(int room_num);

#endif
```

monitor.c

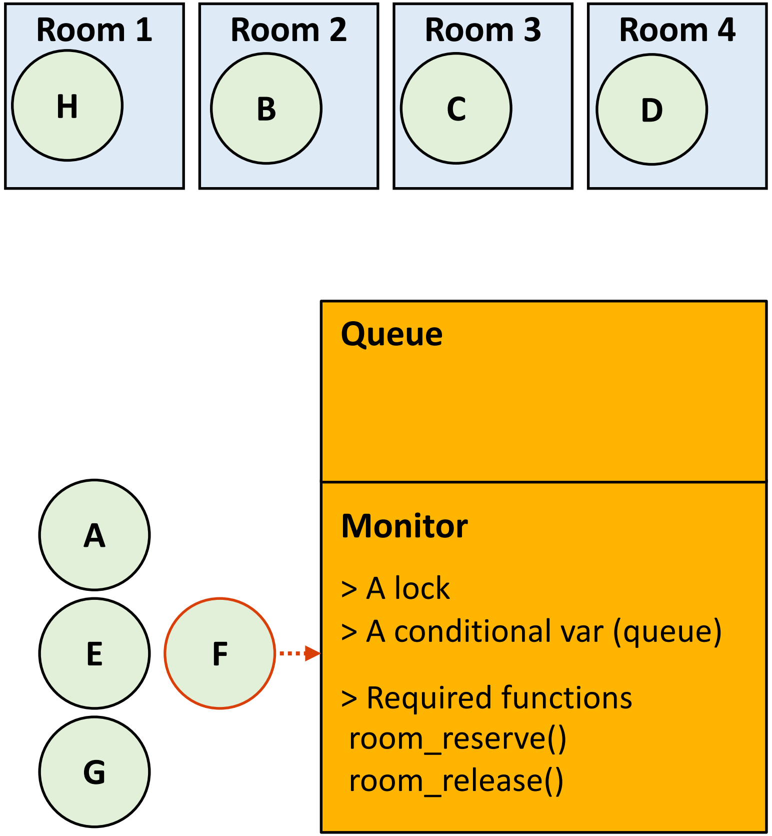
```c
static lock monitor_lock;                          // lock
static struct queue wait_queue;                    // conditional variable
static struct room_t meeting_rooms[4];

void reserve_a_room(int room_num, struct user_t* employee) {
  acquire(&monitor_lock);
  while (meeting_rooms[room_num] != empty) {
    wait(&wait_queue, &monitor_lock);      // wait + unlock + sleep
  }
  room_assign(room_num, employee);
  release(&monitor_lock);
}

struct user_t* release_a_room(int room_num) {
  acquire(&monitor_lock);
  employee = room_empty(room_num);
  signal(&wait_queue);                       // wake up one of them
  release(&monitor_lock);
  return employee;
}
```

monitor.h

```c
#ifndef  MONITOR_H
#define  MONITOR_H

#define  NUM_ROOMS      4

void reserve_a_room(int room_num, struct user_t* employee);
struct user_t* release_a_room(int room_num);

#endif
```

monitor.c

```c
static lock monitor_lock;                    // lock
static struct queue wait_queue;              // conditional variable
static struct room_t meeting_rooms[4];

void reserve_a_room(int room_num, struct user_t* employee) {      <-------- Runs
   acquire(&monitor_lock);
   while (meeting_rooms[room_num] != empty) {
      wait(&wait_queue, &monitor_lock);      // wait + unlock + sleep
   }
   room_assign(room_num, employee);
   release(&monitor_lock);
}

struct user_t* release_a_room(int room_num) {
   acquire(&monitor_lock);
   employee = room_empty(room_num);
   signal(&wait_queue);                      // wake up one of them
   release(&monitor_lock);
   return employee;
}
```



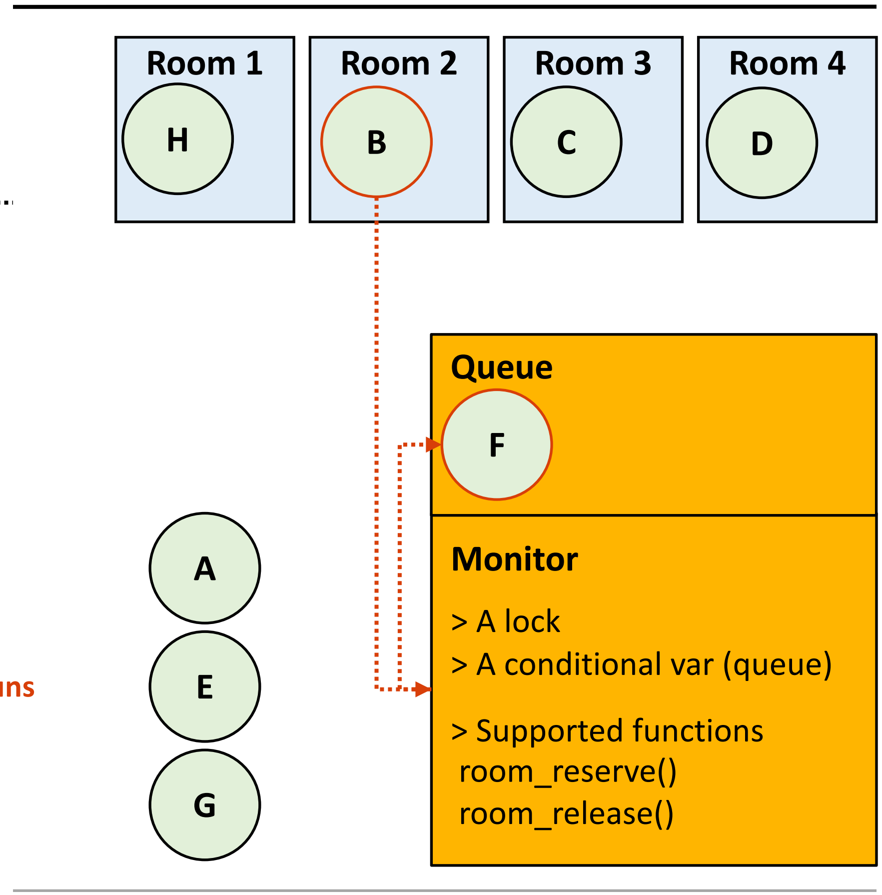
monitor.h

```c
#ifndef  MONITOR_H
#define  MONITOR_H

#define  NUM_ROOMS     4

void reserve_a_room(int room_num, struct user_t* employee);
struct user_t* release_a_room(int room_num);

#endif
```

monitor.c

```c
static lock monitor_lock;                    // lock
static struct queue wait_queue;              // conditional variable
static struct room_t meeting_rooms[4];

void reserve_a_room(int room_num, struct user_t* employee) {
   acquire(&monitor_lock);
   while (meeting_rooms[room_num] != empty) {
      wait(&wait_queue, &monitor_lock);      // wait + unlock + sleep
   }
   room_assign(room_num, employee);
   release(&monitor_lock);
}

struct user_t* release_a_room(int room_num) {
   acquire(&monitor_lock);
   employee = room_empty(room_num);
   signal(&wait_queue);                      // wake up one of them
   release(&monitor_lock);
   return employee;
}
```

Runs



Coke machine v0.6

- **Coke machine with "monitor"**
  - A monitor object for the coke machine
  - It implements two functions
    - produce_fn
    - consumer_fn

```
static lock machine_lock;
static struct queue producer_wait;
static struct queue consumer_wait;
static struct machine coke_machine[NUM_SLOTS];

void produce_fn() {
   acquire(&machine_lock);
   while (machine == full) {
      wait(&producer_wait, &machine_lock); // wait + unlock + sleep
   }
   enqueue(acoke, &coke_machine);
   signal(&consumer_wait);                 // wake up a consumer
   release(&machine_lock);
}

struct coke_t* consumer_fn() {
   acquire(&machine_lock);
   while (machine == empty) {
      wait(&consumer_wait, &machine_lock);// wait + unlock + sleep
   }
   acoke = dequeue(&coke_machine);
   signal(&producer_wait);                 // wake up a producer
   release(&machine_lock);
   return acoke;
}
```

Topics for today

- Part IV – Synchronization III
  - Recap:
    - Coke machine problem (deadlock)
    - Semaphore (in C)
  - Monitor
    - Meeting room booking system
    - Monitor
    - Monitor implemented 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