Monitors

Monitors

- Design: similar to a Java/C++ class
    - private attributes/variables
    - private/public functions
- Condition Variables:
    - Special variables for synchronization; with two operations: [m_]wait() and [m_]signal()
    - They are NOT boolean variables
    - When the context is unclear, the m_ prefix will be used to distinguish between semaphore or condition variable operations
- A user process **enters** the monitor by invoking its **public** functions
    - Only **one process can enter the monitor** (hence invoke the monitor function) at any time

Monitors: Condition Variables

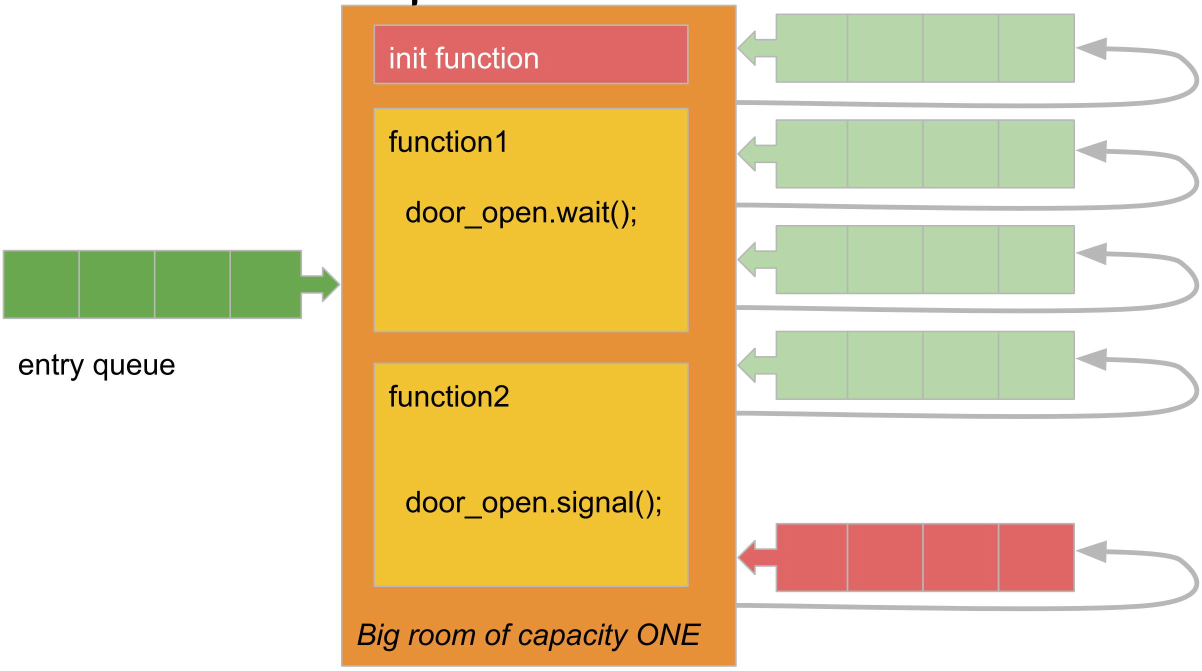
`condition z;`

- `z.wait()`: ALWAYS suspend the process who invokes this operation until another process invokes `z.signal()`. *The monitor is now available again for use by another process*
- `z.signal()`: resumes exactly ONE process. If no process is currently suspended, the operation has no effect.
- **Condition variables ARE NOT boolean variables**

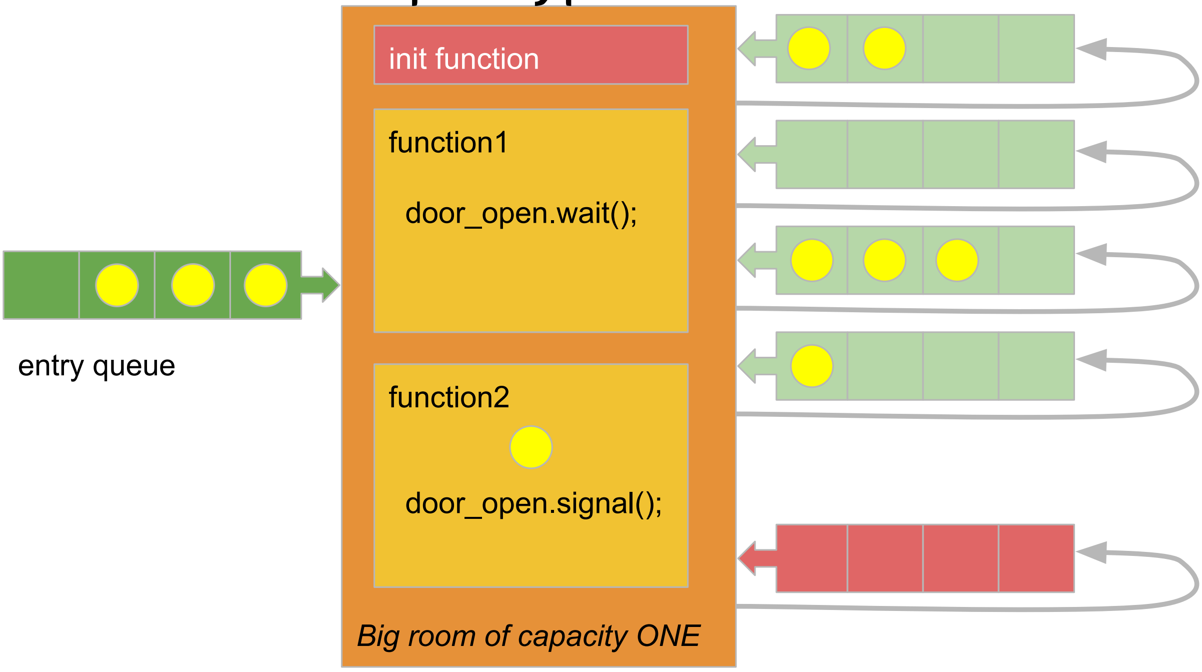
Monitor Architecture

- Queues
    - one entry queue: for holding processes about to invoke one of the monitor functions
    - condition variable queues (one queue per condition variable) for holding processes suspended on the condition var
    - urgent queue: temporary spot for resolving cv.signal() issue (details later)
- Functions
    - public function implementing synchronization logic
    - initialization function
- Data

Architecture of a (vacant) Monitor



Architecture of a Hypothetical Monitor



Monitors: `cv.signal()` issue

- Process P (inside the monitor) is executing `cv.signal()`, process Q is at the front of the cv's queue, and it is now ready to resume its `cv.wait()`
- Two processes (P and Q) are now potentially inside the monitor
  - Set a policy that `cv.signal()` must be the last statement executed in a monitor function
  - When `cv.signal()` is not the last statement:
    - signal-and-continue: let P continue, move Q to the **urgent queue**
    - signal-and-wait: move P to the **urgent queue**, let Q resume its `cv.wait()`
  - Resume process in the urgent queue as soon as monitor is empty

---

Semaphores vs. Monitors

- `sem.wait()` **may** get blocked
- `sem.signal()` is memorized (semaphore value is incremented)

- `cv.wait()` is **always** blocked
- `cv.signal()` is NOT memorized, it has no effect when the cv queue is empty

Writing Monitor Solutions

- Write (public) monitor functions to be called for **entry section** and **exit section**
  - The entry section code typically invokes `cv.wait()`
  - The exit section code typically invokes `cv.signal()`
  - Write private helper functions when needed
- Examples (in a separate handout)
  - Dining Philosophers: attemptToDine (entry section) and finishDining (exit section)
  - Readers/Writer: startReading, startWriting (entry sections) and finishReading, finishWriting (exit sections)

Implementing a Monitor using Semaphores

- Requirements
  a. At most one process at a time inside a monitor (at most one process can CALL any monitor public function)
  b. Processes can be blocked on a condition variable
  c. Processes can be blocked on the urgent Q and should be dequed ahead of other processes (urgent Q has higher prio)
  d. Calling wait() on a condition variable ALWAYS block its caller
- How many semaphores do we need?
- We will use “modern” semaphores (a queue is already builtin!)

Semaphore Requirements for Monitors

- A binary semaphore (`mutex = 1`): mutual exclusive access to monitor functions
- A binary semaphore (`urgent = 0`): block a process in the urgent queue
- For each condition variable `cond_var`
    - `int cv_count = 0;`   /* the number of processes blocked on the cond-var */
    - `semaphore cv_sem = 0;` /* hold blocked processes inside the semaphore's queue*/

Monitor Functions: PRELUDE and POSTLUDE

```
// prelude
mutex.wait();
// function body

   cv_x.m_wait();


   cv_y.m_signal()


// postlude
mutex.signal();
```

Requirement (a)
at most ONE process can invoke any monitor function

- `m_wait()` and `m_signal()` are operations on CONDITION VARS
- `wait()` and `signal()` are operations on SEMAPHORE

mutex is a SEMAPHORE
cv_x, cv_y are CONDITION VARIABLES

Monitor Functions: PRELUDE and POSTLUDE

```
// prelude
mutex.wait();

// function body

   cv_x.m_wait();


   cv_y.m_signal()


// postlude
if (urgent_count > 0)
  urgent.signal();
else
  mutex.signal();
```

Requirement(a)
at most ONE process can invoke any monitor function

Requirement (c)
Processes in Urgent Q have a higher priority to (re)enter the monitor

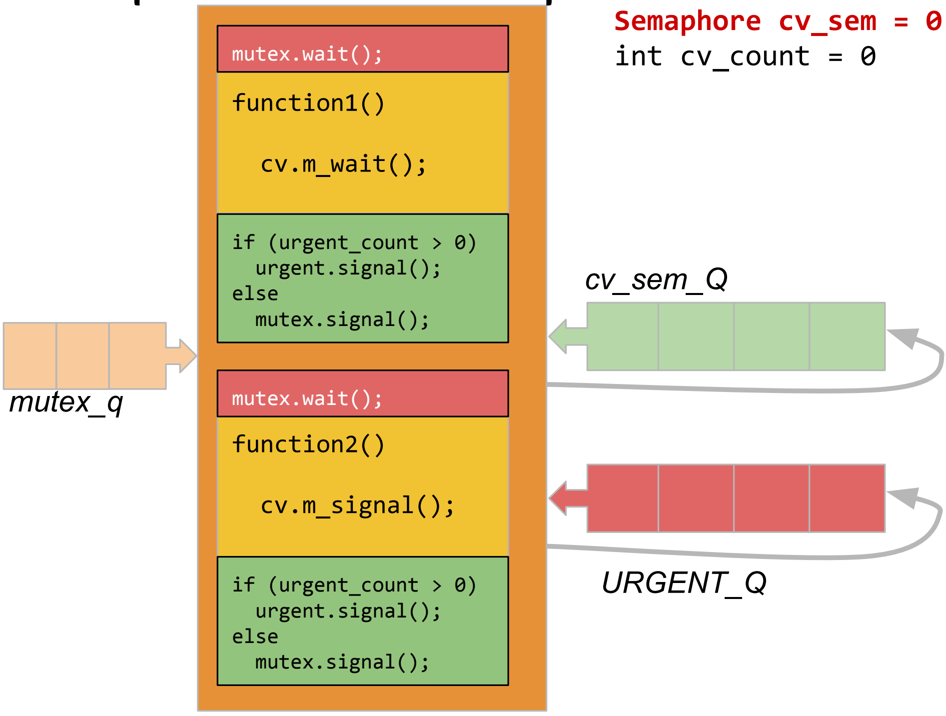
mutex, urgent are SEMAPHORES
cv_x, cv_y are CONDITION VARIABLES

Monitor Condition Variables

- `cv.m_signal()` has no affect when no one is blocked on `cv.m_wait()`
- When there are several processes blocked on `cv.m_wait()`, calling `cv.m_signal()` will release only ONE process
- `cv.m_wait()` ALWAYS block its caller

Implementation of cv.m_wait() and cv.m_signal()

```
Semaphore cv_sem = 0;  /* a "barrier" */
int cv_count = 0
```



```
mutex.wait();
function1()

  cv.m_wait();

if (urgent_count > 0)
  urgent.signal();
else
  mutex.signal();
```

```
mutex.wait();
function2()

  cv.m_signal();

if (urgent_count > 0)
  urgent.signal();
else
  mutex.signal();
```

```
// cv.m_wait()
cv_count++;
if (urgent_count > 0)
  urgent.signal();
else
  mutex.signal();
cv_sem.wait();
cv_count--;
```

```
// cv.m_signal()
if (cv_count > 0) {
  urgent_count++;
  cv_sem.signal();
  urgent.wait();
  urgent_count--;
}
```

Java Synchronized Methods

```
public class Database {
  public void methodOne() {

  }

  public synchronized void methodTwo() {


  }
}
```

```
// One.java
Database db = new Database();

class Worker implements Runnable {
   public void run() {

      db.methodTwo();
   }
}


//main: two concurrent threads
Thread one = new Worker().start();
Thread two = new Worker().start();
```

pthread mutex (binary semaphores)

```
#include <pthread.h>
pthread_mutex_t mtx;

void* myfunc (void * ___) {


  pthread_mutex_lock (&mtx);
  /* critical section here */
  pthread_mutex_unlock (&mtx);

}

int main() {
   pthread_mutex_init (&mtx, NULL);

   pthread_create (_, _, myfunc, _);
}
```

```
#include <pthread.h>
pthread_cond_t cv;
pthread_mutex_t mtx;

void* func1 (void * ___) {

  pthread_cond_wait (&mtx, &cv);

}
void* func2 (void * ___) {

  pthread_mutex_signal (&cv);

}

int main() {
   pthread_cond_init (&cv, NULL);

   pthread_create (_, _, func1, _);
   pthread_create (_, _, func2, _);
}
```

POSIX semaphores (counting semaphores)

```
#include <semaphore.h>
sem_t mtx;
void * myfunc (void *arg) {

  sem_wait (&mtx);
  // mutually exclusive code here
  sem_post (&mtx);

}

int main() {
  sem_init (&mtx, 0, init_val); /* 0:shared among threads */

  pthread_create (____, ___, myfunc, ____);
}
```

SysV Semaphores

- semget(): create one or more semaphores
- semctl()
  - set initial value
  - semaphore management (remove, query status, ….)
- semop(): modify semaphore value (for implementing lock/unlock or wait/signal)

C++11 Synchronization

- `#include <mutex>`
- `#include <condition_variable>`
- `std::mutex`: binary semaphores
- `std::condition_variable`: monitor semaphore variables
- Examples
  - Implementation of counting semaphores using std::mutex
  - Producer-Consumer (semaphore solution)
  - Implementation of monitor using semaphores
  - Dining-Philosopher (monitor solution)