Monitors and Thread-Safe Classes

Daniel Zappala

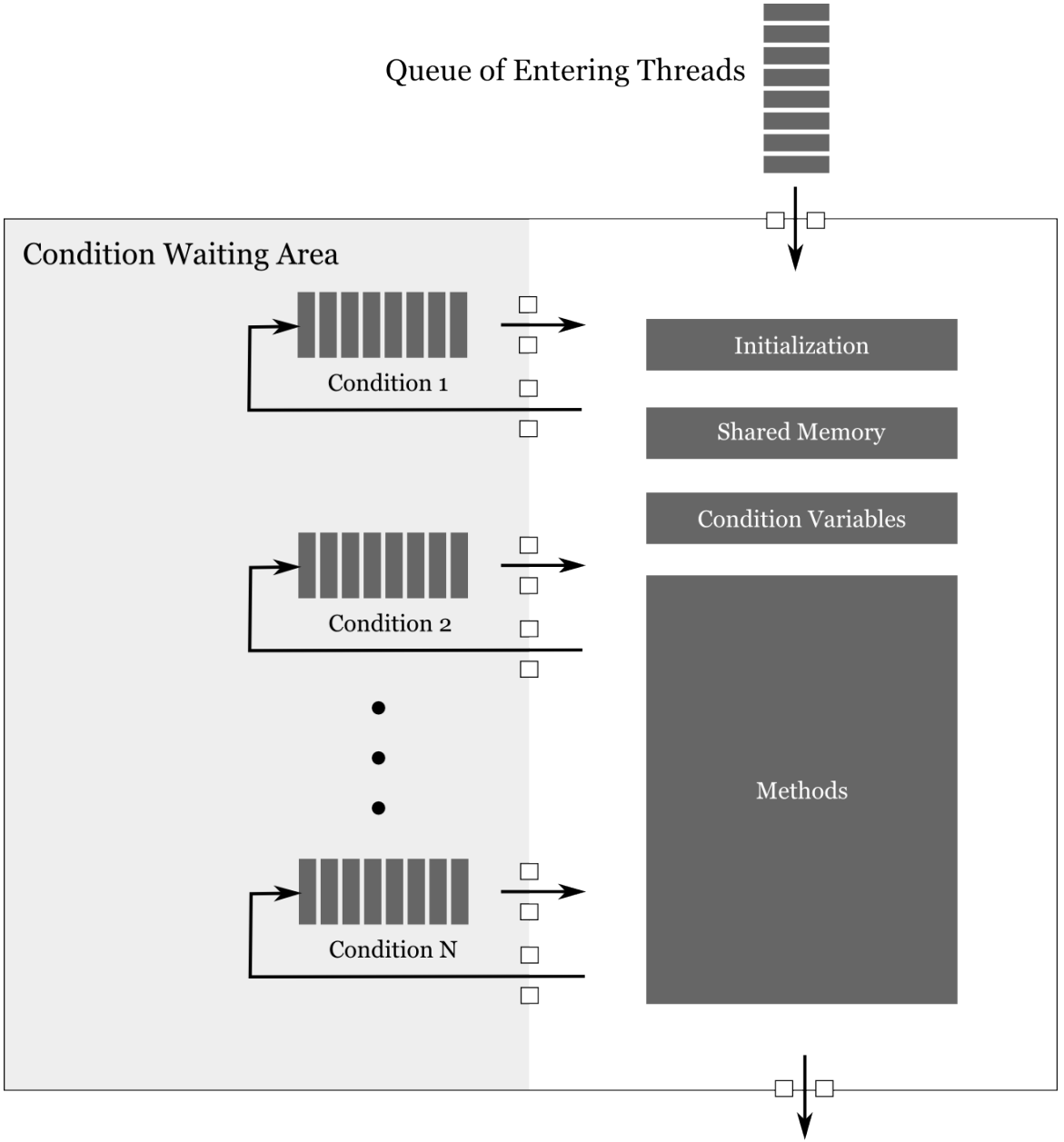
CS 360 Internet Programming
Brigham Young University

Monitors

Monitor

- difficult to get semaphores, mutexes, condition variables right
  - match wait and signal
  - put in right order
  - scattered throughout code
- monitor: programming language construct
  - equivalent functionality
  - easier to control
  - mutual exclusion constraints can be checked by the compiler
  - used in versions of Pascal, Modula, Mesa
  - Java also has a Monitor object but compliance cannot be checked at compile time

Hoare Monitor



Hoare Monitor

- monitor can only be entered through methods
- shared memory can only be accessed by methods
- only one process or thread in monitor at any time
- may suspend and wait on a condition variable
- like object-oriented programming with mutual exclusion added in

Hoare Synchronization

- `cwait(c)`: suspend on condition c
- `csignal(c)`: wake up one thread waiting for condition c
  - do nothing if no threads waiting (signal is lost)
  - different from semaphore (number of signals represented in semaphore value)

Producer Consumer with a Hoare Monitor

```
1  vector buffer;
2  condition notfull, notempty;
```

```
1  append(item) {
2    if buffer.full()
3      cwait(notfull);
4    buffer.append(item);
5    csignal(notempty);
6  }
```

```
1  take() {
2    if buffer.empty();
3      cwait(notempty);
4    item = buffer.remove();
5    csignal(notfull);
6    return item;
```

Producer Consumer with a Hoare Monitor

producer:

```
1  while (True) {
2     item = produce();
3     append(item);
4  }
```

consume:

```
1  while (True) {
2     item = take();
3     consume(item);
4  }
```

- advantages
  - moves all synchronization code into the monitor
  - monitor handles mutual exclusion
  - programmer handles synchronization (buffer full or empty)
  - synchronization is confined to monitor, so it is easier to check for correctness
  - write a correct monitor, any thread can use it

Lampson and Redell Monitor

- Hoare monitor requires that signaled thread must run immediately
  - thread that calls `csignal()` must exit the monitor or be suspended
  - for example, when `notempty` condition signaled, thread waiting must be activated immediately or else the condition may no longer be true when it is activated
  - usually restrict `csignal()` to be the last instruction in a method (Concurrent Pascal)
- Lampson and Redell
  - replace `csignal()` with `cnotify()`
  - `cnotify(x)` signals the condition variable, but thread may continue
  - thread at head of condition queue will run at some future time
  - must recheck the condition!
  - used in Mesa, Modula-3

Producer Consumer with a Lampson Redell Monitor

```
1  vector buffer;
2  condition notfull, notempty;
```

```
1  append() {
2    while buffer.full()
3      cwait(notfull);
4    buffer.append(item);
5    cnotify(notempty);
6  }
```

```
1  take() {
2  while buffer.empty()
3    cwait(notempty);
4    item = buffer.remove();
5    cnotify(notfull);
6    return item;
7  }
```

Lampson Redell Advantages

- allows processes in waiting queue to awaken periodically and reenter monitor, recheck condition
  - prevents starvation
- can also add `cbroadcast(x)`: wake up all processes waiting for condition
  - for example, append variable block of data, consumer consumes variable amount
  - for example, memory manager that frees k bytes, wake all to see who can go with k more bytes
- less prone to error
  - process always checks condition before doing work

Thread-Safe Classes

Organizing Semaphores

- difficult to get semaphores right
  - match wait and signal
  - put in right order
  - scattered throughout code
- put them in a class, with the data structures they use
  - private data structures, public methods
  - any object calling this class is thread-safe

Thread-Safe Classes

```
1  class Buffer {
2    public:
3      append(item) {
4        pthread_mutex_lock(&lock);
5        while buffer.full() {
6          pthread_cond_wait(&not_full,&lock);
7        }
8        buffer.append(item);
9        pthread_cond_signal(&not_empty);
10       pthread_mutex_unlock(&lock);
11     };
12     take() {
13       pthread_mutex_lock(&lock);
14       while buffer.empty() {
15         pthread_cond_wait(&not_full,&lock);
16       }
17       item = buffer.remove();
18       pthread_cond_signal(&not_full);
19       pthread_mutex_unlock(&lock);
20       consume(item);
21     };
```