Monitors and Thread-Safe Classes

Daniel Zappala

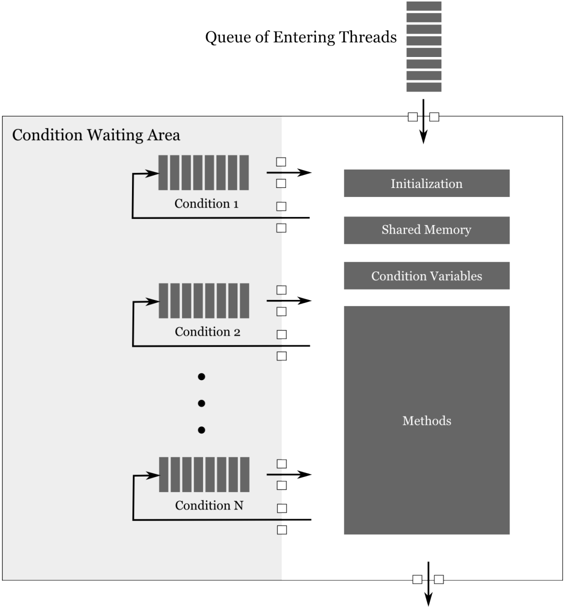
CS 360 Internet Programming
Brigham Young University

Monitors

Monitor

- difficult to get mutexes and condition variables right
  - be sure locks used everywhere they are needed
  - match wait and notify to signal correct conditions
  - 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)

Producer Consumer with a Hoare Monitor

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

```
append(item) {
  if buffer.full()
    cwait(notfull);
  buffer.append(item);
  csignal(notempty);
}
```

```
take() {
  if buffer.empty();
    cwait(notempty);
  item = buffer.remove();
  csignal(notfull);
  return item;
```

Producer Consumer with a Hoare Monitor

producer:

```
while (True) {
  item = produce();
  append(item);
}
```

consume:

```
while (True) {
  item = take();
  consume(item);
}
```

- 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

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

```
append() {
  while buffer.full()
    cwait(notfull);
  buffer.append(item);
  cnotify(notempty);
}
```

```
take() {
while buffer.empty()
  cwait(notempty);
  item = buffer.remove();
  cnotify(notfull);
  return item;
}
```

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 Mutexes and Condition Variables

- difficult to get mutexes and condition variables right
    - be sure locks used everywhere they are needed
    - match wait and notify to signal correct conditions
    - 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

```
class Buffer {
  public:
    append(item) {
      lock.lock();
      while buffer.full() {
        not_full.wait(&lock);
      }
      buffer.append(item);
      not_empty.notify();
      lock.unlock();
    };
    take() {
      lock.lock();
      while buffer.empty() {
        not_empty.wait(&lock);
      }
      item = buffer.remove();
      not_full.notify();
      lock.unlock();
      consume(item);
    };

  private:
    vector buffer;
};
```