Concurrency/synchronization using UML state models

Overview
- State models of monitor objects
- Monitor-process pattern
- Heuristic for "generating" correct synchronization code from a state model
- Use of signal vs. broadcast to notify waiting threads
- Examples
- Synchronization that is more difficult to model using state diagrams

Method for using models to design concurrent software
- State model for each system object
  - Passive objects modeled using monitor-process pattern
- Models then refined into code skeletons
  - Active objects incorporate infrastructure needed to host a thread:
    - E.g., a distinguished run method whose activation corresponds to the lifetime of a thread
  - Passive objects designed to behave as monitors
    - Design pattern for passive-object model guarantees monitor-object pattern can be applied to develop code
    - Guarded transitions in state model engender condition synchronization in code
- Simple monitor-process pattern
  - Simple pattern:
    - Distinguished initial state Idle
    - Composite state for each monitor operation
      - reachable only from Idle via an accept-call action
      - Entry is named according to the name of the operation
      - Possibly guarded with an enabling condition
      - Must transition back to Idle via a reply action upon exit
      - Names the entry to which it corresponds
      - Includes return value (if any)
      - No accept-call actions within the composite state
    - Variations on this pattern will relax some of these restrictions

Simple monitor-process pattern

Bounded Buffer Example

Monitor process

Idle

Operation body

 Pulling

 do / accept-call (pull)

 [queue.size != 0] / accept-call (pull)

 Pushing

 do / queue.push(x)

 [queue.size == MAX] / accept-call (push(x))

 BoundedBuffer

 Idle

 do / reply (pull, rv)

 do / reply (push, x)
From simple monitor process to monitor object (code)

- Class declares a private mutex variable `lock`
- Each operation state in the model becomes a monitor method in the code
- Method body bracketed by acquire/release of `lock`
- If any transition out of Idle is guarded:
  - Class must declare a condition variable, associated with `lock`, to wait on when guard condition is NOT satisfied
  - Method body includes a condition-wait loop immediately following acquisition of `lock`
  - All other methods must notify this condition variable if they might serve to satisfy this guard condition
  - Notification could be `signal` or `broadcast`

Example: Code for Buffer::push

```cpp
void Buffer::push(int x)
{
    lock_.acquire();
    while (queue_.size() == MAX) {
        full_.wait();
    }
    queue_.push(x);
    empty_.signal();
    lock_.release();
}
```

Example: Code for Buffer::pull

```cpp
int Buffer::pull()
{
    lock_.acquire();
    while (queue_.size() == 0) {
        empty_.wait();
    }
    int rv = queue_.pull();
    full_.signal();
    lock_.release();
    return rv;
}
```
More complex guard conditions

Consider a banking application that allows multiple clients to deposit and withdraw funds from shared accounts.

Goals:
- Protect against data races, so that money is not accidentally created or destroyed
- Prevent overdrafts by making withdraw requests block if account has insufficient funds
- Question: How should we model the behavior of an account object?

Question: How should we model the behavior of an account object?

More complex monitors...

- Simple monitor-process pattern insufficient for handling some forms of synchronization
  - E.g., barriers, using which two or more threads must arrive before either may pass
  - Not so easily modeled using guarded transitions
- May require:
  - Embedding accept-call actions involving other operations within an operation state
  - Explicit modeling, storage, and retrieval of request objects associated with calls into an entry

Monitor-process model of BankAccount

Monitor-process model of BankAccount

Code for BankAccount::withdraw

```cpp
void BankAccount::withdraw(int amount)
{
    lock_.acquire();
    while (amount > balance_) {
        okToWithdraw_.wait();
    }
    balance_ -= amount;
    okToWithdraw_.broadcast();
    lock_.release();
}
```

Code for BankAccount::deposit

```cpp
void BankAccount::deposit(int amount)
{
    lock_.acquire();
    balance_ += amount;
    okToWithdraw_.broadcast();
    lock_.release();
}
```

Signal vs. Broadcast

- When one thread changes the value of a condition upon which others might be waiting, the modifier is obliged to notify these waiting threads
- Always safest, though perhaps not very efficient, to use broadcast to notify waiting threads after a change
- Question: When is it safe to use signal?
Party-admission problem

- Models the admission of (boy–girl) couples to a party
  - Boys and girls arrive independently but are blocked from entering the party except as couples
  - When a boy arrives, he must block unless and until a girl has arrived for him to hook up with
- Example of something called a barrier synchronization
- Requires embedding accept-call action for girlArrives inside operation state for boyArrives and vice versa
- Violates our simple monitor-process conventions
  - Still, may be modeled without too much trouble and without any use of guards on transitions

Example: Part of the model

Example: Other part of the model