Condition Variables

A queue of threads waiting for something *inside* the critical section (why does it not violate safety?)

Condition variables have two operations:

- Wait()
  - Calling thread blocks, releases lock (*gives up monitor*)
  - Wait on a queue until someone signals variable
- Signal()
  - Unblock someone who was waiting on the variable
- Broadcast()
  - Unblock all waiting on the variable

Producer Consumer, again

```
addToBuff() {
    lock.Acquire();          // lock before using shared data
    put item on buffer
    condition.signal();
    lock.Release();
}
removeFromBuff() {
    lock.Acquire();          // lock before using shared data
    while nothing on buffer {
          condition.wait(&lock)
    }
    remove item from buffer
    lock.Release();
    return item;
}
```

A dilemma

- What happens to a thread that signals on a condition variable while in the middle of a critical section?

Hoare Monitors

- Assume thread *Q* waiting on condition *x*
- Assume thread *P* is in the monitor
- Assume thread *P* calls *x*.signal
- *P* gives up monitor, *P* blocks!
- *Q* takes over monitor, runs
- *Q* finishes, gives up monitor
- *P* takes over monitor, resumes

Example: Hoare Monitors

*fn*1(…)
…
x.wait // T1 blocks

*fn*4(…)
…
x.signal // T2 blocks

// T1 resumes
// T1 finishes

// T2 resumes

Per Brinch Hansen's Monitors

- Assume thread *Q* waiting on condition *x*
- Assume thread *P* is in the monitor
- Assume thread *P* calls *x*.signal
- *P* continues, finishes
- *Q* takes over monitor, runs
- *Q* finishes, gives up monitor

Example: Hansen Monitors

*fn*1(…)
…
x.wait // T1 blocks

// T1 resumes
// T1 finishes

*fn*4(…)
…
x.signal // T2 continues
// T2 finishes

Tradeoff

Hoare

- Awkward in programming
- Cleaner, good for mathematical proofs
- Main advantage: when a condition variable is signalled, condition does not change
- Used bymost textbooks

Hansen

- Easier to program
- Can lead to synchronization bugs
- Used by most systems

Example: Java

- Each object has a monitor
- User can specify the keyword "synchronized" in front of methods that must execute with mutual exclusion
- Use Hansen monitors
- No individually named condition variables (just one anonymous condition variable)
  - wait()
  - notify()
  - notifyAll()

Classic Problems 2: Readers-Writers

- Motivation:

  shared database (bank, seat reservation system)
- Two classes of users:
  - readers (never modify database)
  - writers (read and modify database)
- Want to maximize concurrency
  - at most one writer in database
  - if no writer, any number of readers in database

Towards a solution

State variables:

- ar --- active readers (readers in CS)
- aw --- active writers (writers in CS)

Condition variables:

- oktoread (readers can enter critical section)
- oktowrite (writer can enter critical section)

Safety

$$ar \geq 0 \wedge aw \geq 0 \wedge (aw = 0 \vee (aw = 1 \wedge ar = 0))$$

Structure of the solution

Database::read()
  wait until no writers
  access database
  if no readers in database, let writer in

Database::write()
  wait until no readers or writers
  access database
  wake up waiting readers and writers

Use procedures startRead, doneRead, startWrite, doneWrite

The procedures

```
Database::startRead() {
    lock.Acquire();
    while (aw) {
          okToRead.Wait(&lock);
    }
    ar:= ar+1;
    okToRead.signal;
    lock.Release();
}

Database::doneRead() {
    lock.Acquire();
    ar :=ar -1;
    if (ar =0) {okToWrite.signal}
    lock.Release();
}
```

```
Database::startWrite() {
    lock.Acquire();
    while (aw ∨ ar) {
          okToWrite.Wait(&lock)
          }
    aw := 1;
    lock.Release();
}

Database::doneWrite() {
    lock.Acquire();
    aw := 0;
    okToRead.signal;
    okToWrite.signal;
    lock.Release();
}
```

Semaphores and Monitors - I

Can we build monitors using semaphores?

Try 1:

```
Wait() {P(s)}        Signal() {V(s)}
```

Try 2:

```
Wait(Lock *lock) {          Signal() {
   lock.Release();             V(s);
   P(s);                    }
   lock.Acquire();
}
```

Semaphores and Monitors-II

Try 3:

```
Signal() {
   if semaphore queue is not empty {V(s)}
}
```

REMEMBER:

- If a thread signals on a condition on which no thread is waiting, the result is a no-op.
- If a thread executes a V on a semaphore on which no thread is waiting, the semaphore is incremented!

A few matters of style

- Always do things the same way
- Always use monitors (condition variables plus locks)
- Always hold lock when operating on a condition variable
- Always grab lock at the beginning of a procedure and release it before return
- Always use while (not if) to check for the value of a condition variable
- (Almost) never use sleep to synchronize