Agenda

• What is monitor?
• Condition variables
• Hoare monitor vs Mesa monitor
• Solution to Dining Philosophers

Background

Motivation
• Users can inadvertently misuse locks and semaphores
  – E.g., never unlock a mutex
• Address the key usability issues that arise with semaphores

Idea
• Provide language support to automatically lock and unlock monitor lock when in critical section
  – Lock is added implicitly, never seen by user
• Provide condition variables for scheduling constraints

Monitors

A monitor is:
• a programming language construct that supports controlled access to shared data
  – Synchronization code is added by the compiler
A monitor encapsulates:
• shared data structures
• procedures that operate on the shared data
• synchronization between concurrent threads that invoke those procedures

Data can only be accessed from within the monitor, using the provided procedures
• protects the data from unstructured access
Schematic View of a Monitor

waiting queue of threads trying to enter the monitor

at most one thread in monitor at a time

Monitor Lock

“Automatic” mutual exclusion
- only one thread can be executing inside at any time
  - thus, synchronization is implicitly associated with the monitor – it “comes for free”
- if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
  - more restrictive than semaphores
  - but easier to use (most of the time)

Monitor lock
- Lock before accessing shared data
- release after finishing with shared data
- Lock is initially free

Semaphore vs Monitor

Semaphores used for both mutual exclusion and scheduling constraints
+ elegant (one mechanism for both purpose)
- code can be hard to read and hard to get right

Monitors use separate mechanisms for the two purposes
- use locks for mutual exclusion
- use condition variables for scheduling constraints

Condition Variables

Main idea: make it possible for thread to sleep inside a critical section, by atomically
- Releasing lock
- Putting thread on wait queue and go to sleep
- A rendezvous point

Each condition variable has a queue of waiting thread
Monitor with Condition Variables

Operations on Condition Variables

Three operations on condition variables

- **wait(c)**
  - release monitor lock, so somebody else can get in
  - wait for somebody else to signal condition
  - thus, condition variables have associated wait queues

- **signal(c)**
  - wake up at most one waiting thread
  - if no waiting threads, signal is lost
    - this is different than semaphores: no history!

- **broadcast(c)**
  - wake up all waiting threads

After Signal(c) …

Who runs when the signal(c) is done and there is a thread waiting on the condition variable?

Depends on who runs (waiter or signaler), we have two types of monitors

- **Hoare monitors**: waiter runs
  - run waiter immediately
    - give higher priority to waiter
  - signaler blocks immediately
    - condition guaranteed to hold when waiter runs
    - but, signaler must restore monitor invariants before signaling!
      - cannot leave a mess for the waiter, who will run immediately!
  - lock switch
    - signal gives up lock
    - wait acquires lock
    - signaler re-acquires lock after waiter unlock

- **Mesa monitors**: signaler continues
Mesa Monitors

signal(c) means
• waiter is made ready, but the signaler continues
  – waiter runs when signaler leaves monitor (or waits to re-enter monitor)
• signaler need not restore invariant until it leaves monitor
• being woken up is only a hint that something has changed
  – signaled condition may no longer hold
  – must re-check conditional case

Hoare vs. Mesa Monitors

Hoare monitors:
if (notReady). wait(c)
Mesa monitors:
while (notReady). wait(c)

Mesa monitors easier to use
• more efficient
• fewer switches
• directly supports broadcast
• Most OSes use

Hoare monitors leave less to chance
• when wake up, condition guaranteed to be what you expect

More about Monitor

Language supports monitors
• A monitor is a software module with OO design feature
  – encapsulation

Compiler understands them
• compiler inserts calls to runtime routines for
  – monitor entry
  – monitor exit
  – signal
  – wait

Runtime system implements these routines
• moves threads on and off queues
• ensures mutual exclusion!

Solution to Dining-Philosophers

Introduce state variable for each philosopher i
state[i] = THINKING, HUNGRY, or EATING

Safety: No two adjacent philosophers eat simultaneously
for all i: !(state[i]==EATING && state[i+1%5]==EATING)

Liveness: Not the case that a philosopher is hungry and his neighbors are not eating
for all i: !(state[i]==HUNGRY && (state[i+4%5]==EATING & state[i+1%5]==EATING))
Solution to DP

monitor DP
{
  enum { THINKING; HUNGRY, EATING) state[5];
  condition self[5];

  void pickup (int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING) self[i].wait;
  }

  void putdown (int i) {
    state[i] = THINKING;
    // test left and right neighbors
    test((i + 4) % 5);
    test((i + 1) % 5);
  }
}

Solution to DP (cont)

void test (int i) {
  if (state[(i + 4) % 5] != EATING) &&
  (state[i] == HUNGRY) &&
  (state[(i + 1) % 5] != EATING) ) {
    state[i] = EATING;
    self[i].signal();
  }
}

initialization_code() {
  for (int i = 0; i < 5; i++)
    state[i] = THINKING;
}