CLASSIC SYNCHRONIZATION PROBLEMS

Ken Wong
Washington University, St. Louis

kenw@arl.wustl.edu
www.arl.wustl.edu/~kenw

PRODUCER-CONSUMER PROBLEM

- Shared Data
  ```
  int buffer[M];
  int nextIn = 0, nextOut = 0;
  Semaphore freeSlot = M, notEmpty = 0, enter = 1;
  ```

- Producer Process
  ```
  do {
    ... Produce inItem ...
    Wait(freeSlot);
    Wait(enter);
    buffer[nextIn] = inItem;  // Put newItem
    nextIn = (nextIn + 1) mod M;
    Signal(enter);
    Signal(notEmpty);
  } until (DONE);
  ```

READERS-WRITERS PROBLEM

- An object is shared among M readers and N writers

- Conditions that must be satisfied
  - Any number of readers can simultaneously read the shared object
  - Only 1 writer at a time may modify the object
  - If a writer is modifying the object, no reader may read it.

- Is the producer-consumer problem a special case of the readers-writers problem?
  - No.

- Two solutions
  - Readers have priority over writers
  - Writers have priority over readers
**READERS HAVE PRIORITY**

```c
// SHARED DATA
int nReaders = 0; // num of active readers
Semaphore enter = 1, writeOk = 1;

// READER PROCESS
Wait (enter);
nReaders = nReaders + 1;
if (nReaders == 1) Wait(writeOk); // Place A
Signal (enter); // Place B
... Read object ...
Wait (enter); // Place C
nReaders = nReaders - 1;
if (nReaders == 0) Signal (writeOk);
Signal (enter);

// WRITER PROCESS
Wait (writeOk);
... Write object ...
Signal (writeOk);
```

**DINING PHILOSOPHER PROBLEM**

- **Shared Data**
  
  ```c
  Semaphore fork[4] = 1, 1, 1, 1;
  ```

- **Philosopher i Algorithm**
  ```c
  do {
      Wait (fork[i]);
   Wait (fork[(i+1) % 4]);
   ... Eat ...
   Signal (fork[i]);
   Signal (fork[(i+1) % 4]);
   ... Think ...
  } until (DONE);
  ```

**DINING PHILOSOPHER PROBLEM**

**HIGHER LEVEL SYNCHRONIZATION CONSTRUCTS**

- **Semaphores give the programmer too much freedom and are error prone**
  - Hard to detect timing errors
  - Obscure code (widely separated synchronization pairs)
- **Need higher level synchronization constructs**
  - Promote good coding
- **Constructs**
  - Conditional Critical Region
  - Monitor
DINING PHILOSOPHER USING MONITORS
(Part 1)

MONITOR diningPhilosophers {
    State state[4]; // HUNGRY, THINKING, EATING
    CONDITION phil[4]; // condition variables
    pickup (i) {
        state[i] = HUNGRY;
        test(i); // can I eat?
        phil[i].Wait;
    }
    putdown (i) {
        state[i] = THINKING;
        test ((i+3) mod 4); // Wake up right neighbor
        test ((i+1) mod 4); // Wake up left neighbor
    }
}

DINING PHILOSOPHER USING MONITORS
(Part 2)

test (k) {
    if ((state[(k+3) mod 4] != EATING) and
        (state[k] = HUNGRY) and
        (state[(k+1) mod 4] != EATING)) {
        state[k] = EATING;
        phil[k].Signal }
}

begin { // initialization
    for (i=0; i<4; i++) state[i] = THINKING;
}

MONITOR USAGE

PROCESS philosopher (i) {
    do forever {
        dp.pickup(i);
        ... eat ...
        dp.putdown(i)
    }
}
diningPhilosophers dp;
philosopher d[4] = {0, 1, 2, 3};

BOUNDED BUFFER USING CRITICAL REGION

// SHARED VARIABLES
SHAREED struct {
    int pool[n];
    int count, in, out;
} buffer;

// PRODUCER
REGION buffer WHEN (count < n) {
    pool[in] = ... item ...; // put item into buffer
    in = (in + 1) mod n;
    count = count + 1;
}

// CONSUMER
REGION buffer WHEN (count > n) {
    ... item ... = pool[out]; // get item from buffer
    out = (out + 1) mod n;
    count = count - 1;
}
Entry Section

[Diagram showing Entry Section with conditions B? (TRUE or FALSE)]

Critical Section

Exit Section

---

**CRITICAL REGION IMPLEMENTATION 1**

Get lock;
EvalB: While (B == FALSE) {
    Release lock;
    Enter Q and wait for wakeup signal;
    Get lock;
}
Execute critical section;
If (Q is NUT empty)
    Wakeup first process in Q;
    Release lock;

* Consider the following case:
  * Process J is at the head of the reevaluation queue Q
  * Process K is in the reevaluation queue Q
  * As Process I leaves the CS, B is TRUE for Process K, but FALSE for Process J

---

**CRITICAL REGION IMPLEMENTATION 3**

wait (lock);
while (!B) {
    nq1 = nq1 + 1;
    if (nq2 > 0)
        signal (q2); // wakeup Q2
    signal (lock);
    wait (q1);
    nq1 = nq1 - 1;
    nq2 = nq2 + 1;
    if (nq1 > 0)
        signal (q1); // wakeup others in Q1
    else
        signal (q2); // last one in Q1 execs
    wait (q2); // waiting for chance to reevaluate B
    nq2 = nq2 - 1;
    wait (lock);
}
S; // Critical Section
if (nq1 > 0)
    signal (q1);
else if (nq2 > 0)
    signal (q2);
else signal (lock);