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 new Item
    nextIn = (nextIn + 1) mod M;
    Signal(enter);
    Signal(notEmpty);
  } until (DONE);

PRODUCER-CONSUMER PROBLEM

- Consumer Process
  
  do {
    Wait(notEmpty);
    Wait(enter);
    outItem = buffer[nextOut]; // Get item
    nextOut = (nextOut + 1) mod M;
    Signal(enter);
    Signal(freeSlot);
    ... Consume outItem ...
  } 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**

// 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);

- nR: Number of readers in critical section
- nW: Number of writers in critical section
- nR': Total number of readers executing code
- nW': Total number of writers executing code

**DINING PHILOSOPHER PROBLEM**

- A philosopher eats only if both forks are free
  - i.e., eats only if both neighbors are not eating
- Philosopher States
  - Thinking: Has no forks
  - Hungry: Wants both forks
  - Eating: Has both forks
- Example
  - 4 philosophers (0, 1, 2, 3) and 4 forks
DINING PHILOSOPHER PROBLEM

• Shared Data
  Semaphore  fork[4] = {1, 1, 1, 1};

• Philosopher i Algorithm
  do {
    Wait (fork[i]);
    Wait (fork[(i+1) mod 4]);
    ... Eat ...
    Signal (fork[i]);
    Signal (fork[(i+1) mod 4]);
    ... Think ...
  } until (DONE);

HIGHER LEVEL SYNCHRONIZATION CONSTRUCTS

• Semaphore 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

HOARE MONITOR CONSTRUCT

• Syntax
  MONITOR M {   // Hoare monitor M
    ... Variable Declarations ...
    P1 (...) { }   // procedure 1
    ...
    Pn (...) { }   // procedure n
    begin {
      ... initialization code ...
    }
  }

• Semantics
  • Only one process at a time can be active within a monitor; i.e., executing a monitor procedure.
  • The variables within a monitor can only be accessed from within the the monitor
  • Signalling between processes is done through condition variables in a monitor

DINING PHILOSOPHER USING MONITORS

• A philosopher eats only if both forks are free
  • i.e., eats only if both neighbors are not eating

• Philosopher States
  • Thinking: Has no forks
  • Hungry: Wants both forks
  • Eating: Has both forks

• Example
  • 4 philosophers (0, 1, 2, 3) and 4 forks
  • Philosopher i may wake up neighbor(s) after releasing both forks
CONDITION VARIABLES

- Condition variables allow processes to wait within a monitor
  
  \[\text{CONDITION \ V1, V2, \ldots}\]

- Condition variables can only be used with the Wait and Signal operations

  - V.Wait means wait for a matching V.Signal call
  - V.Signal resumes exactly one suspended process. If there is no suspended process, the operation has no effect.

DINING PHILOSOPHER USING MONITORS

(Part 1)

\[
\text{MONITOR diningPhilosophers \{ // Hoare monitor semantics} \\
\quad \text{// Entry by a process excludes other processes} \\
\quad \text{State state[4]; // HUNGRY, THINKING, EATING} \\
\quad \text{CONDITION phil[4]; // CV: wait/signal; "lost signal problem"} \\
\quad \text{pickup (i) \{} \\
\quad \quad \text{state[i] = HUNGRY;} \\
\quad \quad \text{test(i);} \\
\quad \quad \text{phil[i].Wait;} \\
\quad \quad \text{// can I eat?} \\
\quad \text{\} \}
\]

\[
\text{putdown (i) \{} \\
\quad \text{state[i] = THINKING;} \\
\quad \text{test ((i+3) mod 4); // Wake up right neighbor} \\
\quad \text{test ((i+1) mod 4); // Wake up left neighbor} \\
\quad \text{\} \}
\]

DINING PHILOSOPHER USING MONITORS

(Part 2)

\[
\text{test (k) \{ \quad // Wakeup philosopher k if possible} \\
\quad \text{if ((state[(k+3) mod 4] != EATING) and} \\
\quad \quad \text{(state[(k+1) mod 4] != EATING) and} \\
\quad \quad \text{(state[k] == HUNGRY)) \{} \\
\quad \quad \text{// ASSERT: 1) R and L not eating; and 2) I'm hungry.} \\
\quad \quad \text{state[k] = EATING;} \\
\quad \quad \text{phil[k].Signal} \\
\quad \quad \text{\} \}
\]

\[
\text{begin \{ // Initialization} \\
\quad \text{for (i=0; i<4; i++) state[i] = THINKING;} \\
\quad \text{\} \}
\]

MONITOR USAGE

\[
\text{PROCESS philosopher (i) \{} \\
\quad \text{do forever \{} \\
\quad \quad \text{dp.pickup(i);} \\
\quad \quad \text{... eat ...} \\
\quad \quad \text{dp.putdown(i);} \\
\quad \text{\} \}
\]

\[
\text{diningPhilosophers dp;} \\
\text{philosopher d[4] = \{0, 1, 2, 3\};}
\]