Classic Synchronization Problems (CSE 422S)

Ken Wong
Washington University

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

Producer-Consumer Problem (1)

```
Producer
Consumer
```

Shared Data

- `int buffer[N]`: N integer buffers
- `int nxtIn = 0`: index to next input slot
- `int nxtOut = 0`: index to next output slot
- `Semaphore freeSlot = N;`: # resources
- `Semaphore notEmpty = 0;`: signal event
- `Semaphore enter = 1;`: protect critical section

```
Producer-Consumer Problem (2)
```

```
Producer {  
  ... produce newItem ...
  Wait(freeSlot);
  Wait(enter);
  buffer[nxtIn] = newItem;
  nxtIn = (nxtIn+1) % N;
  Signal(enter);
  Signal(notEmpty);
}

Consumer {  
  Wait(notEmpty);
  Wait(enter);
  outItem = buffer[nxtOut];
  nxtOut = (nxtOut+1) % N;
  Signal(enter);
  Signal(freeSlot);
  ... consume outItem ...
}
```

The Dining Philosopher Problem

Philosopher States

- **Thinking**
  - Has no chopsticks
- **Hungry**
  - Wants both chopsticks
- **Eating**
  - Has both chopsticks

Example

- 4 philosophers 0, 1, 2, 3
- 4 chopsticks

```
[graph showing philosopher states and chopsticks]
```

Producer-Consumer Problem (1)

```
0
...
N-1
```

```
Producer
Consumer
```

Producer
Consumer

Shared Data

- `int buffer[N]`: N integer buffers
- `int nxtIn = 0`: index to next input slot
- `int nxtOut = 0`: index to next output slot
- `Semaphore freeSlot = N;`: # resources
- `Semaphore notEmpty = 0;`: signal event
- `Semaphore enter = 1;`: protect critical section

```
Producer-Consumer Problem (2)
```

```
Producer {  
  ... produce newItem ...
  Wait(freeSlot);
  Wait(enter);
  buffer[nxtIn] = newItem;
  nxtIn = (nxtIn+1) % N;
  Signal(enter);
  Signal(notEmpty);
}

Consumer {  
  Wait(notEmpty);
  Wait(enter);
  outItem = buffer[nxtOut];
  nxtOut = (nxtOut+1) % N;
  Signal(enter);
  Signal(freeSlot);
  ... consume outItem ...
}
```

The Dining Philosopher Problem

Philosopher States

- **Thinking**
  - Has no chopsticks
- **Hungry**
  - Wants both chopsticks
- **Eating**
  - Has both chopsticks

Example

- 4 philosophers 0, 1, 2, 3
- 4 chopsticks

```
[graph showing philosopher states and chopsticks]
```
Bad Dining Philosopher Algorithm

- Algorithm deadlocks
  - How? Why?
- Why do we care?
- Fix: 2-philosopher case
  - Increase # resources
  - Larger atomic operation
  - Resource order (0<2<1<3)
- Extend to N philosophers?
- What about starvation?
  - Only 1 philosopher starves?

Algorithm deadlocks

Semaphore

\[ \text{stick}[4] = \{1, 1, 1, 1\} \]

Process philosopher(i) {
  do {
    Wait (R(i));
    Wait(L(i));
    . . . Eat . . .
    Signal (R(i));
    Signal (L(i));
    . . . Think . . .
  } until (DONE);

  where \( R(i) := \text{stick}[i] \)
  \( L(i) := \text{stick}[(i+1) \mod 4] \)

"defined to be"

An Assymetric Algorithm

Semaphore

\[ \text{stick}[4] = \{1, 1, 1, 1\} \]

Process philosopher(i) {
  do {
    Wait (A(i));
    Wait(B(i));
    . . . Eat . . .
    Signal (B(i));
    Signal (A(i));
    . . . Think . . .
  } until (DONE);

  where
  \( A(i) := \text{stick}[i] \), if even(i)
  \( A(i) := \text{stick}[(i+1) \mod 4] \)
  \( B(i) := \text{stick}[(i+1) \mod 4] \), if even(i)
  \( B(i) := \text{stick}[i] \), otherwise

i.e.,

\( A(i) := R(i), \) if even(i)
\( L(i), \) otherwise

\( B(i) :=  L(i), \) if even(i)
\( R(i), \) otherwise

Readers-Writers (Readers-First)

An object is shared among M readers and N writers

Requirements
  - Only 1 writer at a time may modify the shared object
  - If a writer is modifying the object, no reader may read it
  - Any number of readers can simultaneously read the object
  - Readers have priority over writers

Algorithm is much simpler than Writers-First algorithm???
  - Can I just do some lexical substitutions that switch the role of the reader and the writer processes???

Readers-First Algorithm

\[ \text{int} \quad nR = 0; \quad // \#active rdrs \]

Semaphore lock = 1, writeOk = 1;

Process reader(i) {
  Wait(lock);
  nR = nR + 1;
  if (nR == 1) Wait(writeOk);
  Signal(lock);
  . . . Read object . . .
  Wait(lock);
  nR = nR - 1;
  if (nR == 0) Signal(writeOk);
  Signal(lock);
}

Process writer(i) {
  Wait(writeOk);
  . . . Write object . . .
  Signal(writeOk);

  writeOk
  - Mutual exclusion for writing
  lock
  - Mutual exclusion for updating nR

Main Ideas???
Readers-First Algorithm

- **Writer Process**
  - Provide critical section for writing object
  - Provide process queue for writers

- **Reader Process**
  - Fit in with writer process control structure
  - *First reader* blocks all writers
  - *Last reader* unblocks writer

- **Shared Variables**
  - nR: Number of readers
  - lock: Protect CS for updating nR
  - writeOK: Protect CS for writing object

Bad Writers-First Algorithm

```c
int nW = 0;
Semaphore lock = 1, readOk = 1;

Process writer(i) {
    Wait(lock);
    nW = nW + 1;
    if (nW== 1)  Wait(readOk);
    Signal(lock);
    . . .   Write object   . . .
    Wait(lock);
    nW = nW – 1;
    if (nW == 0)   Signal(readOk);
    Signal(lock);
}

Process reader(i) {
    Wait(readOk);
    . . .   Read object   . . .
    Signal(readOk);
}
```

Still Bad Writers-First Algorithm

```c
int nW = 0;
Semaphore lock = 1, writeOk = 1, readOk = 1;

Process writer(i) {
    Wait(lock);
    nW = nW + 1;
    if (nW== 1)  Wait(readOk);
    Signal(lock);
    . . .   Write object   . . .
    Wait(lock);
    nW = nW – 1;
    if (nW == 0)   Signal(readOk);
    Signal(lock);
}
```

Higher Level Synchronization Constructs

- **Semaphores** are error prone
  - Hard to detect timing errors
  - Obscure code (widely separated synchronization pairs)

- A **monitor** is a higher level synchronization construct

- **Semantics**
  - Only 1 process at a time can be active in a monitor
  - A monitor variable can only be accessed within the monitor
  - Signalling between processes is done through **condition variables** in a monitor
**Structure of a Monitor**

- Entering Processes
- MONITOR
  - Local Data
  - Condition Variables
  - Procedure 1
  - Procedure n
- Initialization
- Exit

**Condition Variables**

- Condition variables allow processes to wait within a monitor
  - Cond V1, V2, ...
- Condition variables can only be used with the Cwait and Csignal operations
  - Cwait(V) means wait for a matching Csignal(V) call
  - Csignal(V) resumes exactly one suspended process
    - The operation has no effect if there is no suspended process
  - Cwait and Csignal behave differently from semaphores!!!

**Bounded Char Buffer Monitor (1)**

```c
Monitor boundedBuffer {
  char buf[N];
  int nxtIn, nxtOut;
  int count;
  cond notFull, notEmpty;

  put (int char x) {  // put (in char x)
    if (count == N) Cwait(notFull);
    buf[nxtIn] = x;
    count = count + 1;
    Csignal(notEmpty);
  }

  get (Out char x) {  // get (out char x)
    if (count == 0) Cwait(notEmpty);
    x = buf[nxtOut];
    nxtOut = (nxtOut + 1) mod N;
    count = count - 1;
    Csignal(notFull);
  }

  begin { // initialization
    nxtIn = 0;
    nxtOut = 0;
    count = 0;
  }
}
```

**Bounded Char Buffer Monitor (2)**

```c
Monitor boundedBuffer {
  // Variables:  buffer[N], nxtIn, nxtOut, count
  // Only one process at a time can access these shared variables
  cond notFull, notEmpty

  // Two external functions
  put(x):  Put character x into buffer
  get(x):  Get character x from buffer

  // Initialization:
  nxtIn, nxtOut, count

  // Two Processes
  Producer:  Inserts characters into buffer
  Consumer:  Removes characters from buffer
  Execute in parallel
```