**Classic Synchronization Problems (CS422S)**

Ken Wong  
Washington University  
kenw@wustl.edu  
www.arl.wustl.edu/~kenw

---

### Producer-Consumer Problem (1)

- **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  
  - `notEmpty = 0;` // signal event  
  - `enter = 1;` // protect critical section

---

### Producer-Consumer Problem (2)

**Process Producer**

- ... produce newItem ...  
  - `Wait(freeSlot);`  
  - `Wait(enter);`  
  - `buffer[nxtIn] = newItem;`  
  - `nxtIn = (nxtIn+1) mod N;`  
  - `Signal(enter);`  
  - `Signal(notEmpty);`  

**Process Consumer**

- `Wait(notEmpty);`  
  - `Wait(enter);`  
  - `outItem = buffer[nxtOut];`  
  - `nxtOut = (nxtOut+1) mod N;`  
  - `Signal(enter);`  
  - `Signal(freeSlot);`  
  - ... consume outItem ...

---

### The Dining Philosopher Problem

- **Philosopher States**
  - Thinking  
  - Has no forks  
  - Hungry  
  - Wants both forks  
  - Eating  
  - Has both forks

- **Example**
  - 4 philosophers 0, 1, 2, 3
  - 4 forks
**Bad Dining Philosopher Algorithm**

Semaphore
fork[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) := fork[i], if even(i)
R(i) := fork[(i+1) mod 4], otherwise
L(i) := fork[(i+1) mod 4], if even(i)
L(i) := fork[i], otherwise

- Algorithm deadlocks
  - How? Why?
- Why do we care?
- Fix: 2-philosopher case
  - Increase resources
  - Larger atomic operation
  - Assymetric algorithm
- Extend to N philosophers?
- What about starvation?
  - Only 1 philosopher starves?

*defined to be*

**An Assymetric Algorithm**

Semaphore
fork[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) := fork[i], if even(i)
A(i) := fork[(i+1) mod 4], otherwise
B(i) := fork[(i+1) mod 4], if even(i)
B(i) := fork[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 (Writers-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
  - Writers have priority over readers
- Algorithm is more complex than Readers-First algorithm???
  - Can I just do some lexical substitutions that switch the role of the reader and the writer processes???

**Readers-First Algorithm**

int nR = 0; // 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???
**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);
```

- Mutual exclusion for writing?
- Deadlock free?
- Fair?
- Responsive?

**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);
```

- Mutual exclusion for writing?
- Deadlock free?
- Fair?
- Responsive?

**Better Writers-First Algorithm**

```c
int nW = 0;
Semaphore enter = 1, writeOk = 1;
readOk = 1;
Process writer(i) {
    Wait(enter);
    nW = nW + 1;
    if (nW == 1)  Wait(readOk);
    Signal(enter);
    Wait(writeOk);
    ... Write object ...
    Wait(enter);
    nW = nW - 1;
    if (nW == 0)  Signal(readOk);
    Signal(enter);
}
Process reader(i) {
    Wait(readOk);
    Wait(writeOk);
    ... Read object ...
    Signal(writeOk);
    Signal(readOk);
```

**Higher Level Synchronization Constructs**

- Semaphores are error prone
  - Hard to detect timing errors
  - Obscure code (widely separated synchronization pairs)
- The **Hoare 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
- Condition Variables
- Procedure 1
- Procedure n
- Initialization

**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)

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

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

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

Bounded Char Buffer Monitor (2)

```
Monitor
  Variables: buffer[N], nxtIn, nxtOut, count
  - Only one process at a time can access these shared variables
  - 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 from buffer
  - Consumer: Removes characters from buffer
  - Execute in parallel
```