**Classic Synchronization Problems (CSE 422S)**

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

---

**Producer-Consumer Problem (1)**

```
Producer
Producer:  
  Produce newItem;  
  Wait(freeSlot);  
  Buffer[nxtIn] = newItem;  
  nxtIn = (nxtIn+1) mod N;  
  Signal(enter);  
  Signal(notEmpty);
```

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

---

**Producer-Consumer Problem (2)**

```
int buffer[N];  
int nxtIn = 0;  
int nxtOut = 0;  
Semaphore enter = 1,  
Semaphore freeSlot = N,  
Semaphore notEmpty = 0;  
```

---

**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
**Bad Dining Philosopher Algorithm**

Semaphore
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) := stick[i]
    L(i) := stick[(i+1) mod 4]
}

"defined to be"

- 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?

**An Assymetric Algorithm**

Semaphore
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) := stick[i], if even(i)
    A(i) := stick[(i+1) mod 4], otherwise
    B(i) := stick[(i+1) mod 4], if even(i)
    B(i) := 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**

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);

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

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

Swap Roles of Reader and Writer
Structure of a Monitor

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