Today’s Plan

Upcoming:
► Quiz 3
► Assignment 2

Today’s topics:
► From last time:
  ► Semaphore Implementation
  ► Readers-Writers Problem
  ► Dining Philosophers Problem
  ► Monitors

Last time:
► Semaphores
► Producer / Consumer
Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write

Problem:
- Allow multiple readers to read at the same time
- Only one single writer can access the shared data at the same time
Readers-Writers Problem (Cont.)

⇒ Shared Data

⇒ Data set
⇒ Semaphore *mutex* – to control access to critical section
⇒ Semaphore *wrt* – to ensure mutual exclusivity when writing
⇒ Integer *readcount* – to count the readers
The structure of a writer process:

```
writer: while (1) {
    P (wrt) ;
    // writing is performed
    V (wrt) ;
}
```
Readers-Writers Problem (Cont.)

- The structure of a reader process:

```c
while (1) {
    P (mutex) ;
    readcount++ ;
    if (readcount == 1)
        P (wrt) ;
    V (mutex) 

    // reading is performed
    P (mutex) ;
    readcount-- ;
    if (readcount == 0)
        V (wrt) ;
    V (mutex) ;
}
```
Dining-Philosophers Problem

- Shared data
  - Bowl of rice (data set)
  - Semaphore *chopstick* [5] initialized to 1
Dining-Philosophers Problem (Cont.)

The structure of Philosopher $i$:

```c
while (1) {
    P ( chopstick[i] );
    P ( chopStick[ (i + 1) % 5 ] );
    // eat
    V ( chopstick[i] );
    V ( chopstick[ (i + 1) % 5 ] );
    // think
}
```
## Monitors

- Higher-level, considered more intuitive than semaphores
- Usually requires a language construct (i.e. C doesn’t have it)
- Consists of shared data, subroutines, initialization code
Monitors

- To use a monitor to ensure mutual exclusion for a critical section:
  - Put the CS in a subroutine that can be called by all processes that want access to the CS
  - Put that subroutine in a monitor (e.g. put it in the "procedures" section of the monitor)

- easy to extend to several related critical sections
  - e.g. list manipulation routines
Schematic View of a Monitor
Monitors

- **Wait(C)**
  - Suspends the current process until another process calls `Signal(C)`, `C` is a *condition variable*
  - A suspended process is considered to be “out of the monitor” so other processes can access monitor routines

- **Signal(C)**
  - If there is a process suspended on `C`, resume it and wait until it has left the monitor before proceeding
  - Otherwise, continue execution

- **Notify(C)**
  - Like `Signal`, but resume the suspended process after the notifying process leaves the monitor
Monitor with Condition Variables

- Shared data
  - Queues associated with x, y conditions
  - Operations
  - Initialization code

- Entry queue
BinarySemaphore via a Monitor

MONITOR SEM
- shared data:
  int busy;
  condition nonbusy;
- Procedures: P() {
  if (busy)
    nonbusy.wait();
  busy++;
}
V() {
  busy = 0;
  nonbusy.signal();
}
- Init Code: begin() {
  busy = 0;
}
END SEM
Producer/Consumer via Monitor

MONITOR LISTMON

- shared data:

  LIST itemList;
  int maxListSize = N;
  condition bufavail, itemavail;

- Procedures:
  enqueueItem(item) {
    if(ListCount(itemList) == maxListSize)
      bufAvail.wait();
      ListPrepend(itemList, item);
    itemAvail.signal();

  }
Producer/Consumer via Monitor

def getItem() {
    if(ListCount(itemList) == 0) /* list is empty */
        itemAvail.wait();
    item = ListTrim(itemList);
    bufAvail.signal();
    return(item);
}

-Init Code:
    itemList = ListCreate();
END LISTMON
Solution to Dining Philosophers

monitor DP
{
    enum { THINKING; HUNGRY, EATING} state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self [i].wait();
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test(((i + 4) % 5);
        test(((i + 1) % 5);
    }
}
Solution to Dining Philosophers (cont)

```c
void test (int i)
{
    if ( (state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code()
{
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
```
Solution to Dining Philosophers (cont)

Each philosopher $i$ invokes the operations in the following sequence: