Using Semaphores

• It is difficult to use semaphores
  – see example in Fig 5.9
  – semaphores may be scattered throughout the program
    • difficult to assess overall effect
• Monitors provide similar functionality
  – but are easier to control
  – implemented in languages like Concurrent Pascal, Pascal-Plus, Modula-2 & 3, and Java

Monitors

• A Monitor is a software module
• Chief characteristics
  – Local data variables are accessible only by the monitor
  – Process enters monitor by invoking one of its procedures
  – Only one process may be executing in the monitor at a time
Monitors

• Provides mutual exclusion facility
• Shared data structure can be protected by placing it into a monitor
• If the data in a monitor represents some resource, then mutual exclusion is guaranteed for that resource

Monitors

• Synchronization support is needed
  – implemented using special data types called *condition variables*
  – these variables are affected by two functions
    • *cwait*(c)
      – suspend calling process on condition c
      – now monitor can be used by other process
    • *csignal*(c)
      – resume blocked process after *cwait* on same condition c
Monitors

• So what is the difference between the use of cwait and csignal in monitors and the wait and signal of semaphores?
  – Hint: remember what got us in trouble when using semaphores

Monitors

• Monitor wait and signal operations are different from their counterparts in semaphores
  – If a process in a monitor signals and corresponding queue is empty then signal is lost
/* program producer_consumer */
monitor bounded_buffer;
    char buffer[N];        /* space for N items */
    int nextin, nextout;   /* buffer pointers */
    int count;             /* number of items in buffer */
    cond notfull, notempty; /* condition variables for synchronization */

void append (char x)
{
    if (count == N)
        wait(notfull);    /* buffer is full: avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    /* one more item in buffer */
    signal(notempty);     /* resume any waiting consumer */
}

void take (char *)
{
    if (count == 0)
        wait(notempty);    /* buffer is empty: avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    /* one fewer item in buffer */
    signal(notfull);    /* resume any waiting producers */
}{
    nextin = 0; nextout = 0; count = 0;        /* buffer initially empty */
Message Passing

- Interaction between processes
  - synchronization
  - communication

- One solution to this is message passing
  - works in tightly and loosely coupled systems
Message Passing

- Enforce mutual exclusion
- Exchange information

send (destination, message)
receive (source, message)

Synchronization

- Sender and receiver may or may not be blocking (waiting for message)

- Blocking send, blocking receive
  - Both sender and receiver are blocked until message is delivered
  - This is called a *rendezvous*
Synchronization

• Nonblocking send, blocking receive
  – Sender continues on
  – Receiver is blocked until the requested message arrives

• Nonblocking send, nonblocking receive
  – Neither party is required to wait

Addressing

• Direct addressing
  – Send primitive includes a specific identifier of the destination process
  – Receive primitive could know ahead of time which process a message is expecting
  – Receive primitive could use source parameter to return a value when the receive operation has been performed
Addressing

• Indirect addressing
  – Messages are sent to a shared data structure consisting of queues
  – Queues are called *mailboxes*
  – One process sends a message to the mailbox and the other process picks up the message from the mailbox
  – relationship between sender & receiver
    • 1-to-1, many-to-1, 1-to-many, many-to-many

![Diagram of indirect process communication (b) Many to one, (c) One to many, (d) Many to many]
Message Format

Assumptions:
- blocking receive
- non-blocking send

```c
/* program mutual_exclusion */
const int n = /* number of processes */;
void print()
{
    message msg;
    while (true)
    {
        receive (mutex, msg);
        /* critical section */
        send (mutex, msg);
        /* remainder */
    }
}
void main()
{
    create_mailbox (mutex);  // What happens if the send is omitted?
    send (mutex, null);
    parbegin (D[1], D[2], ..., D[n]);
}
```

Figure 5.20 Mutual Exclusion Using Messages
What does the for loop do?

Readers/Writers Problem

• Different variations on the theme, e.g.,
  – dedicated readers and dedicated writers
  – they all can read and write

• Here we look at the “dedicated” case
  – Any number of readers may simultaneously read the file
  – Only one writer at a time may write to the file
  – If a writer is writing to the file, no reader may read it
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true)
    {
        semWait (x);
        readcount++;
        if (readcount == 1)
            semWait (wsem);
        semSignal (x);
        semWait (x);
        readcount--;
        if (readcount == 0)
            semSignal (wsem);
        semSignal (x);
    }
}

void writer()
{
    while (true)
    {
        semWait (wsem);
        waitUnit (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}

Figure 5.22 A Solution to the Readers/ Writers Problem Using Semaphores: Readers Have Priority

/* program readersandwriters */
int readcount, writecount;
semaphores x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
        semWait (z);
        readcount++;
        if (readcount == 1)
            semWait (rsem);
        semSignal (x);
        semWait (y);
        readcount--;
        if (readcount == 0)
            semSignal (wsem);
        semSignal (y);
    }
}

void writer()
{
    while (true)
    {
        semWait (w);
        writecount++;
        if (writecount == 1)
            semWait (wsem);
        semWait (y);
        waitUnit (y);
        if (writecount == 0)
            semSignal (rsem);
        semSignal (y);
    }
}

void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}

Figure 5.23 A Solution to the Readers/ Writers Problem Using Semaphores: Writers Have Priority

x: controls updating readcount
wsem: controls writing

z: prevent long reader queue; only 1 reader lines up at rsem, other readers line up at z

y: controls updating of writecount