Roadmap

• Principals of Concurrency
• Mutual Exclusion: Hardware Support
• Semaphores

Monitors

• Message Passing
• Readers/Writers Problem
Monitors

• The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.

• Implemented in a number of programming languages, including
  – Concurrent Pascal, Pascal-Plus,
  – Modula-2, Modula-3, and Java.
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
Synchronization

• Synchronisation achieved by **condition variables** within a monitor
  – only accessible by the monitor.

• Monitor Functions:
  – \( \text{Cwait}(c) \): Suspend execution of the calling process on condition \( c \)
  – \( \text{Csignal}(c) \): Resume execution of some process blocked after a \text{cwait} on the same condition
Structure of a Monitor
Bounded Buffer Solution Using Monitor

```c
/* program producerconsumer */
monitor boundedbuffer;
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) cwait(notfull);   /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);                /* resume any waiting consumer */
}

void take (char x) {
    if (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                         /* one fewer item in buffer */
    /* monitor body */
    csignal(notfull);                /* resume any waiting producer */
}

{ nextin = 0; nextout = 0; count = 0; /* buffer initially empty */
}```
Solution Using Monitor

```c
void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}

void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}

void main()
{
    parbegin (producer, consumer);
}
```
void append (char x)
{
    while(count == N) cwait(notfull); /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    / * one more item in buffer */
    cnotify(notempty);
    / * notify any waiting consumer */
}

void take (char x)
{
    while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    / * one fewer item in buffer */
    cnotify(notfull);
    / * notify any waiting producer */
}
Roadmap

• Principals of Concurrency
• Mutual Exclusion: Hardware Support
• Semaphores
• Monitors

Message Passing

• Readers/Writers Problem
Process Interaction

• When processes interact with one another, two fundamental requirements must be satisfied:
  – synchronization and
  – communication.

• Message Passing is one solution to the second requirement
  – Added bonus: It works with shared memory and with distributed systems
Message Passing

• The actual function of message passing is normally provided in the form of a pair of primitives:
  • send (destination, message)
  • receive (source, message)
Synchronization

• Communication requires synchronization
  – Sender must send before receiver can receive

• What happens to a process after it issues a send or receive primitive?
  – 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
• Known as a *rendezvous*
• Allows for tight synchronization between processes.
Non-blocking Send

• More natural for many concurrent programming tasks.
• 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

- Sendin process need to be able to specify which process should receive the message
  - Direct addressing
  - Indirect 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 expected
• Receive primitive could use source parameter to return a value when the receive operation has been performed
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
Indirect Process Communication

(a) One to one

(b) Many to one

(c) One to many

(d) Many to many

Figure 5.18 Indirect Process Communication
General Message Format

- **Header**
  - Message Type
  - Destination ID
  - Source ID
  - Message Length
  - Control Information

- **Body**
  - Message Contents

Figure 5.19  General Message Format
Mutual Exclusion Using Messages

```c
/* program mutualexclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section */
        send (box, msg);
        /* remainder */
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), ..., P(n));
}
```

Figure 5.20  Mutual Exclusion Using Messages
const int
capacity = /* buffering capacity */ ;
null = /* empty message */ ;
int i;
void producer()
{
  message pmsg;
  while (true) {
    receive (mayproduce, pmsg);
    pmsg = produce();
    send (mayconsume, pmsg);
  }
}
void consumer()
{
  message cmsg;
  while (true) {
    receive (mayconsume, cmsg);
    consume (cmsg);
    send (mayproduce, null);
  }
}
void main()
{
  create_mailbox (mayproduce);
  create_mailbox (mayconsume);
  for (int i = 1; i <= capacity; i++) send (mayproduce, null);
  parbegin (producer, consumer);
}