UPPSALA UNIVERSITET

Informationsteknologi

Today's class

- Mutual exclusion and synchronization
  - Hardware support
  - Semaphores
  - Producer/Consumer problem
  - Readers/Writers problem

UPPSALA UNIVERSITET

Informationsteknologi

Mutual Exclusion: Hardware Support

- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Processor is limited in its ability to interleave programs
  - Multiprocessing - disabling interrupts on one processor will not guarantee mutual exclusion

UPPSALA UNIVERSITET

Informationsteknologi

Mutual Exclusion: Hardware Support

```
while (true) {
   /* disable interrupts */
   /* critical section */
   /* enable interrupts */
   /* remainder */
}
```

UPPSALA UNIVERSITET

Informationsteknologi

Mutual Exclusion: Hardware Support

- Special Machine Instructions
  - Performed in a single instruction cycle
  - Access to the memory location is blocked for any other instructions

UPPSALA UNIVERSITET

Informationsteknologi

Mutual Exclusion: Hardware Support

- Test and Set Instruction

```
boolean testset (int i) {
    if (i == 0) {
        i = 1;
        return true;
    }
    else {
        return false;
    }
}
```

Mutual Exclusion Based on Test and Set

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true)
   {
      while (!testset (bolt))
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));

}
```

Mutual Exclusion: Hardware Support

- Exchange Instruction

```
void exchange(int register,
              int memory) {
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

Mutual Exclusion Based on Exchange

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi;
   while (true)
   {
      keyi = 1;
      while (keyi != 0)
            exchange (keyi, bolt);
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

Mutual Exclusion Machine Instructions

UPPSALA UNIVERSITET

Informationsteknologi

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

Mutual Exclusion Machine Instructions

UPPSALA UNIVERSITET

Informationsteknologi

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs it, the higher priority process will obtain the processor to wait for the critical region

Semaphores

- Special variable called a semaphore is used for signaling
- If a process is waiting for a signal, it is suspended until that signal is sent
- `semSignal(`*s*`)` transmits a signal via semaphore *s*
- `semWait(`*s*`)` receives a signal via semaphore *s*; if the signal has not yet been sent, the process is suspended until the transmission takes place

UPPSALA UNIVERSITET

Informationsteknologi

Semaphores

- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
  - `semWait` operation decrements the semaphore value; if it becomes negative then the process executing `semWait` is blocked, otherwise the process continues execution
  - `semSignal` operation increments the semaphore value; if the value is less than or equal to 0 then a process blocked by `semWait` is unblocked

UPPSALA UNIVERSITET

Informationsteknologi

Semaphore Primitives

```
struct semaphore {
     int count;
     queueType queue;
}

void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0)
     {
          place this process in s.queue;
          block this process
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0)
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.3 A Definition of Semaphore Primitives

UPPSALA UNIVERSITET

Informationsteknologi

Binary Semaphore Primitives

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};

void semWaitB(binary_semaphore s)
{
     if (s.value == 1)
          s.value = 0;
     else
          {
               place this process in s.queue;
               block this process;
          }
}
void semSignalB(semaphore s)
{
     if (s.queue.is_empty())
          s.value = 1;
     else
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

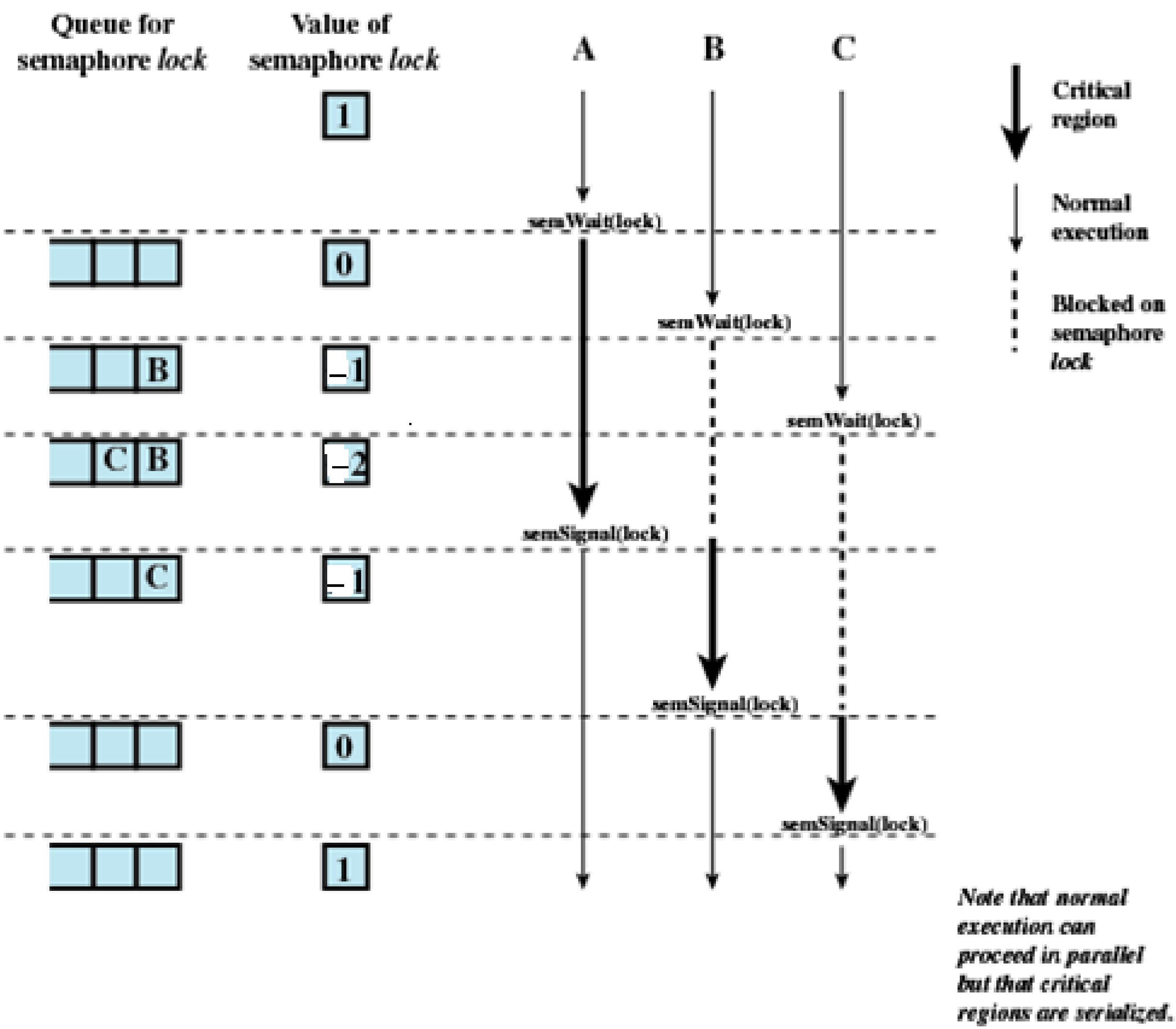
UPPSALA UNIVERSITET

Informationsteknologi

Mutual Exclusion Using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
      while (true)
      {
            semWait(s);
            /* critical section   */;
            semSignal(s);
            /* remainder    */;
      }
}
void main()
{
      parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores


Queue for semaphore *lock*
Value of semaphore *lock*
A
B
C
1
semWait(lock)
0
semWait(lock)
B
–1
semWait(lock)
C
B
–2
semSignal(lock)
C
–1
semSignal(lock)
0
semSignal(lock)
1
Critical region
Normal execution
Blocked on semaphore *lock*
Note that normal execution can proceed in parallel but that critical regions are serialized.

UPPSALA UNIVERSITET

Informationsteknologi

Producer/Consumer Problem

- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time

Producer

```
producer:
while (true) {
  /* produce item v */
  b[in] = v;
  in++;
}
```

UPPSALA UNIVERSITET

Informationsteknologi

Consumer

```
consumer:
while (true) {
  while (in <= out)
     /*do  nothing */;
  w = b[out];
  out++;
  /* consume item w */
}
```

Producer/Consumer Problem

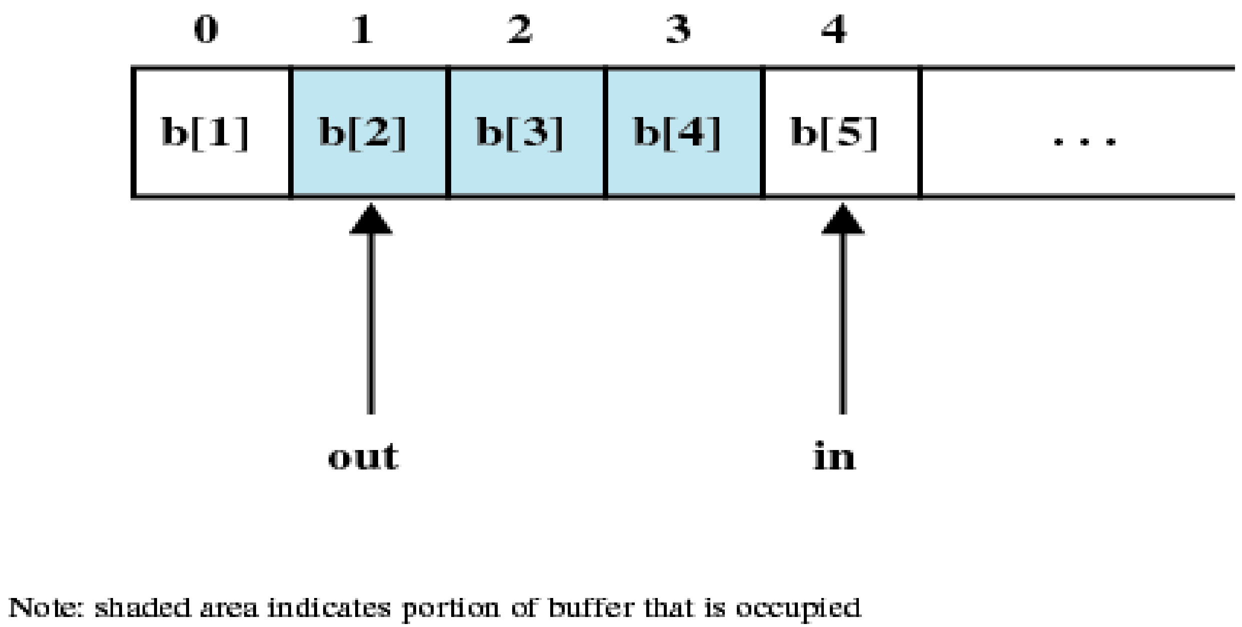


Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

UPPSALA UNIVERSITET

Informationsteknologi

Producer with Circular Buffer

```
producer:
while (true) {
  /* produce item v */
  while ((in + 1) % n == out)
         /* do nothing */;
  b[in] = v;
  in = (in + 1) % n
}
```

Consumer with Circular Buffer

```
consumer:
while (true) {
  while (in == out)
     /* do nothing */;
  w = b[out];
  out = (out + 1) % n;
  /* consume item w */
}
```

UPPSALA UNIVERSITET

Informationsteknologi

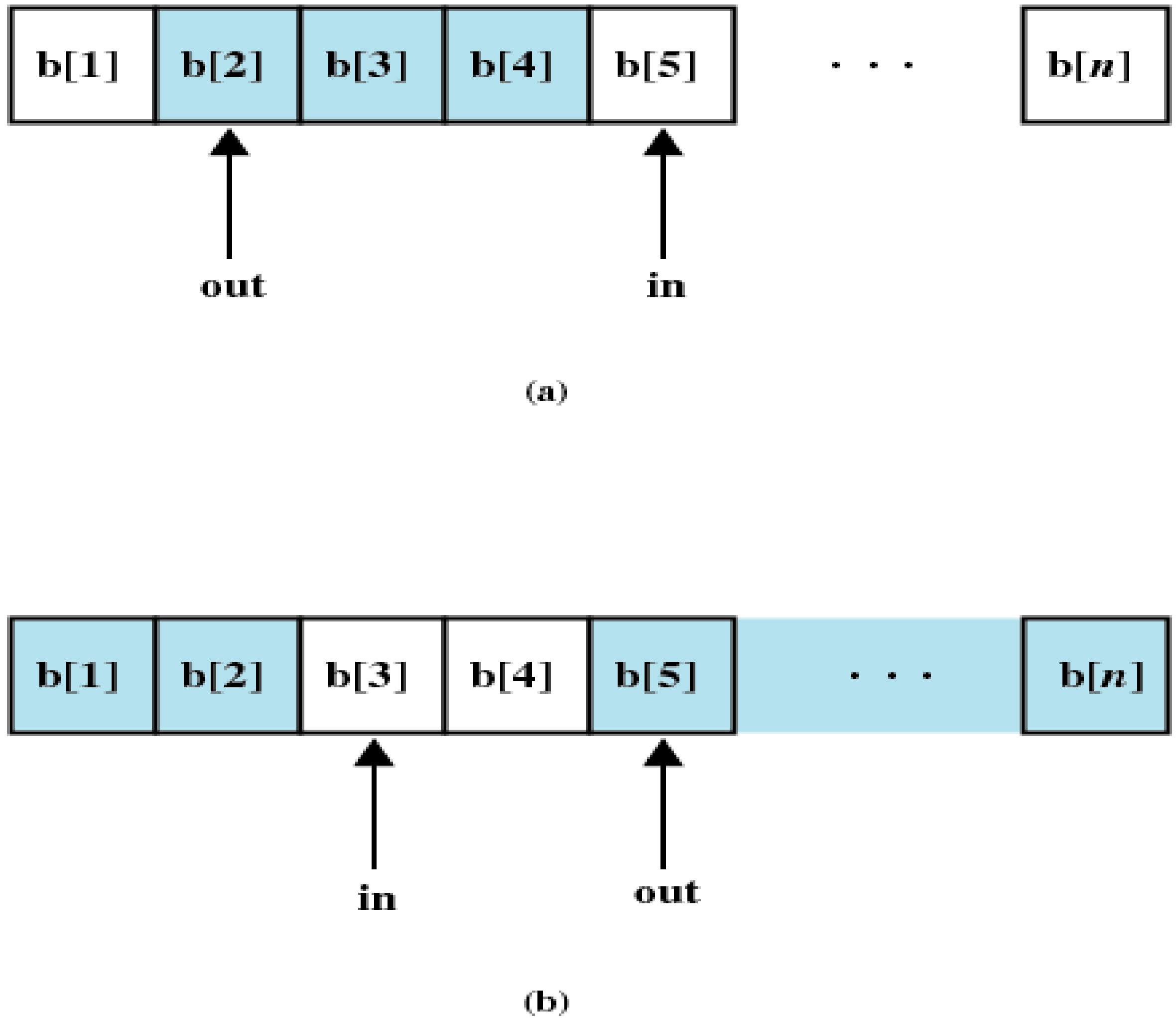


Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

UPPSALA UNIVERSITET

Informationsteknologi

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
      while (true)
      {
            produce();
            semWaitB(s);
            append();
            n++;
            if (n==1)
                  semSignalB(delay);
            semSignalB(s);
      }
}
void consumer()
{
      semWaitB(delay);
      while (true)
      {
            semWaitB(s);
            take();
            n--;
            semSignalB(s);
            consume();
            if (n==0)
                  semWaitB(delay);
      }
}
void main()
{
      n = 0;
      parbegin (producer, consumer);
}
```

Figure 5.9 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
     while (true)
     {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)
     {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Figure 5.10 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

UPPSALA UNIVERSITET

Informationsteknologi

```
/* program  producerconsumer */
semaphore n = 0;
semaphore s = 1;
void producer()
{
     while (true)
     {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true)
     {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.11 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

UPPSALA UNIVERSITET

Informationsteknologi

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1;
semaphore n= 0;
semaphore e= sizeofbuffer;
void producer()
{
      while (true)
      {
            produce();
            semWait(e);
            semWait(s);
            append();
            semSignal(s);
            semSignal(n)
      }
}
void consumer()
{
      while (true)
      {
            semWait(n);
            semWait(s);
            take();
            semSignal(s);
            semSignal(e);
            consume();
      }
}
void main()
{
      parbegin (producer, consumer);
}
```

Figure 5.13 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

UPPSALA UNIVERSITET

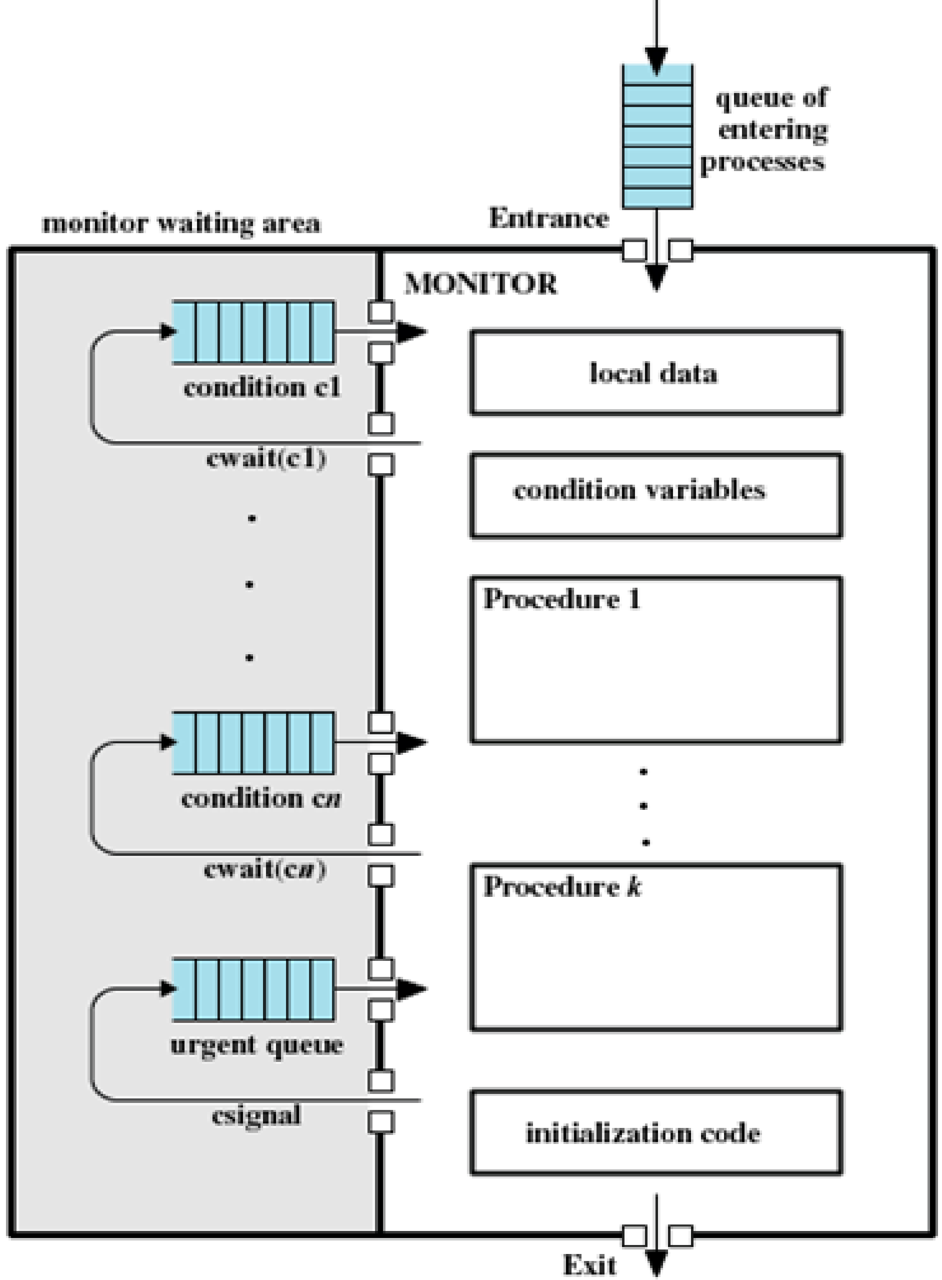
Informationsteknologi

Monitors

- 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

Informationsteknologi

UPPSALA UNIVERSITET



UPPSALA UNIVERSITET

Informationsteknologi

```
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);
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

```
/* 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 */
     csignal(notfull);                            /* resume any waiting producer */
}
{                                                                 /* monitor body */
     nextin = 0; nextout = 0; count = 0;              /* buffer initially empty */
}
```

Informationsteknologi

UPPSALA UNIVERSITET

Message Passing

- Enforce mutual exclusion
- Exchange information

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

UPPSALA UNIVERSITET

Informationsteknologi

Synchronization

- Message communication requires some level of synchronization – a message cannot be received before it is sent
- 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
  - Called a rendezvous

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

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

UPPSALA UNIVERSITET

Informationsteknologi

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

Message Format

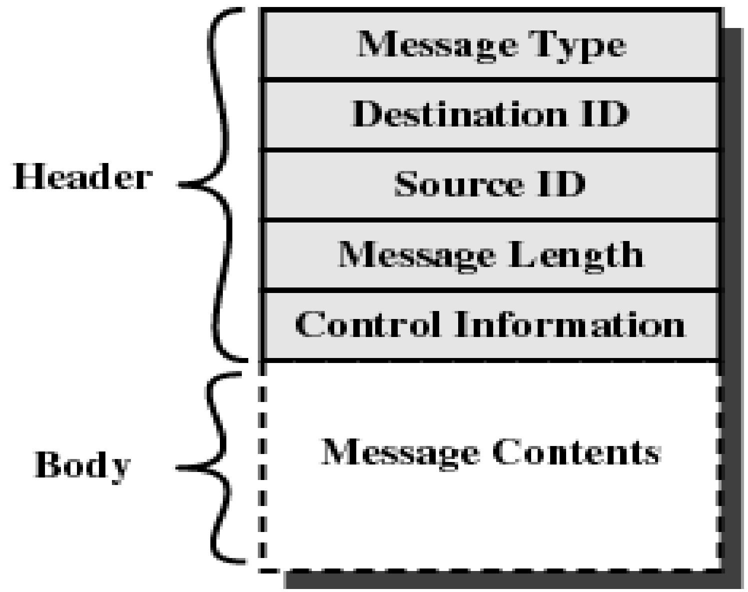


Figure 5.19 General Message Format

UPPSALA UNIVERSITET

Informationsteknologi

Readers/Writers Problem

- 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;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true)
    {
     semWait (x);
     readcount++;
     if (readcount == 1)
          semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0)
          semSignal (wsem);
     semSignal (x);
    }
 }
void writer()
{
    while (true)
    {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

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

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

UPPSALA UNIVERSITET

Informationsteknologi

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