Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process.
Binary semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to 0) must be the one to unlock it (sets the value to 1).
Condition variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures, and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event flags	A memory word used as a synchronization mechanism. Application code may associ- ate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

Semaphores

semSignal(s):

- To transmit signal via semaphore ***s***
- Increments semaphore value: if result is less than or equal 0 —> if there is a blocked process it is unblocked

semWait(s):

- To recieve signal via semaphore ***s***
- Decrements semaphore value: if result ibecomes negative—> the process is a blocked

Semaphore:

- Positive: ***s*** value = number of processes that can issue a wait and immediately continue to execute.
- Negative: ***s*** value = number of processes waiting to be unblocked.

Consequences

1. No way to know before a process decrements a s whether it will be blocked or not.
2. After a process increments s and another process gets woken up, both continue running concurrently. No way to know which process, if either, will continue immediately on a uniprocessor system.
3. After a process increments s, number of woken up may be 0 or 1; you don't know if there is a process waiting or not.

Strong Semaphore (FIFO)

The process that has been blocked the longest is released from the queue first

Weak Semaphore

The order in which processes are removed from the queue is not specified

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s) {
     s.count--;
     if (s.count < 0) {
         /* s.queue push(this process) */;
         /* block this process */;
      }
}
void semSignal(semaphore s) {
     s.count++;
     if (s.count<= 0) {
       /* s.queue pop() P */;
       /* P -> ready list */;
     }
}
```

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s) {
     if (s.value == one)
          s.value = zero;
     else {
          /* s.queue push(this process) */;
          /* block this process */;
     }
}
void semSignalB(semaphore s) {
     if (s.queue is empty())
          s.value = one;
     else {
          /* s.queue pop() P */;
          /* P -> ready list */;}
}
```

Mutual Exclusion Lock (mutex)

```
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));
}
```

Producer/Consumer Infinite Buffer (Wrong Solution)

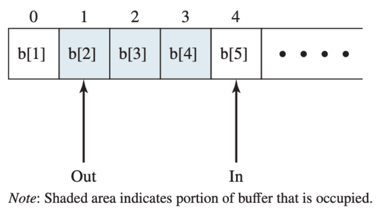
0
1
2
3
4
b[1]
b[2]
b[3]
b[4]
b[5]
• • • •
Out
In
Note: Shaded area indicates portion of buffer that is occupied.

Figure 5.11 **Infinite Buffer for the Producer/ Consumer Problem**

```
int n;
binary_semaphore s = 1, delay = 0;
```

Producer:

```
While (true) {
     produce();
     semWaitB(s);
     append();
     n++;
     if (n == 1)
          semSignalB(delay);
     semSignalB(s);
}
```

Consumer

```
semWaitB(delay);
While (true) {
     semWaitB(s);
     take();
     N——;
     semSignalB(s);
     consume();
     if (n == 0)
          semWaitB(delay);
}
```

Main

```
n = 0;
parbegin(producer, consumer);
```

The producer

- free to add to the buffer at any time.
- Performs semWaitB (s) before appending
  and semSignalB (s) afterward
  to prevent the consumer (or any other producer) from accessing the buffer during the append operation.
- while in the critical section, the producer increments the value of *n*
- If n = 1, then buffer was empty so semSignalB(delay) to add product in bufffer (delay = 1)

The consumer

- begins by waiting for the first item to be produced, using semWaitB (delay).
- It then takes an item and decrements *n* in its critical section.
- If n = 0 means empty buffer, we set delay = 0

Producer /Consumer Infinite Buffer (Correct Solution)

```
int n;
binary_semaphore s = 1, delay = 0;
```

Producer:

```
While (true) {
      produce();
      semWaitB(s);
      append();
      n++;
      if (n == 1)
            semSignalB(delay);
      semSignalB(s);
}
```

Consumer

```
int m;
semWaitB(delay);
While (true) {
      semWaitB(s);
      take();
      n--;
      m = n;
      semSignalB(s);
      consume();
      if (m == 0)
            semWaitB(delay);
}
```

Main

```
n = 0;
parbegin(producer, consumer);
```

Producer/Consumer Bounded-Buffer

Producer:

```
While (true) {
      produce(v);
      while((in+1)%n == out):
            //do nth;
      b[in] = v;
      in = (in+1) % n;
}
```

Consumer

```
While (true) {
      while(in == out):
            //do nth;
      W = b[out];
      Out = (out+1) % n;
      consume();
}
```

```
int sizeofbuffer;
semaphore s = 1, n = 0, e = sizeofbuffer;
```

Producer:

```
While (true) {
     produce();
     semWait(e)
     semWait(s);
     append();
     semSignal(s);
     semSignal(n);
}
```

Consumer

```
While (true) {
     semWait(n);
     semWait(s);
     take();
     semSignal(s);
     semSignal(e);
     consume();
}
```

Main

```
parbegin(producer, consumer);
```

e —> keeps track of number of empty spaces

Using Compare & Swap

semWait(s)

```
While(compare_and_swap(s.flag, 0, 1) == 1)
      //do nth;
s.count--;
if(s.count < 0) {
      /* s.queue push(this P) */;
      /* block P */;
}
s.flag = 0;
```

semSignal(s)

```
While(compare_and_swap(s.flag, 0, 1) == 1)
      //do nth;
s.count++;
if(s.count <= 0) {
      /* s.queue pop() P */;
      /* P -> ready list */;
}
s.flag = 0;
```

Using Interrupts

semWait(s)

```
Inhibit interrupts;
s.count--;
if(s.count < 0) {
      /* s.queue push(this P) */;
      /* block P */;
}
allow interrupts;
```

semSignal(s)

```
Inhibit interrupts;
s.count++;
if(s.count <= 0) {
      /* s.queue pop() P */;
      /* P -> ready list */;
}
allow interrupts;
```

Monitors

```
/* 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 */
}
```

```
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);
}
```

Mesa Monitor

```
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 */
}
```

When a process executing in a monitor executes cnotify(x), it causes the *x* condition queue to be notified, but the signaling process continues to execute. The result of the notification is that the process at the head of the condition queue will be resumed at some convenient future time when the monitor is available.

However, because there is no guarantee that some other process will not enter the monitor before the waiting process, the waiting process must recheck the condition
therefore—>**While** loop instead of if condition on **count**

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  – Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data

Monitor Characteristics

- Local data variables are accessible only by the monitor's procedures and not by any external procedure
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time

Message Passing

Communication of a message between two processes implies **synchronization** between the two
– The receiver cannot receive a message until it has been sent by another process

If recieve(source, message) is being executed, then two possibilities:

1. The message has already been sent —> message recieved and execution continues
2. No message was sent —> process is blocked until message is sent, or continues to execute leaving the message recieving part.

Blocking Send, Blocking Recieve (rendezvous)

- Both sender and receiver are blocked until the message is delivered
- Tight synchronization

Nonblocking Send, Blocking Recieve

- Sender continues on but receiver is blocked until the requested message arrives
- Most useful combination
- Sends one or more messages to a variety of destinations as quickly as possible

Nonblocking send, nonblocking receive

Addressing

Direct

- Send includes identifier of the destination process
- Recieve handled:
    1. Require that the process explicitly designate a sending process (Effective for cooperating concurrent processes)
    2. Implicit Addressing: recieve includes a value for source parameter that is returmed when recieve operation is performed

Indirect

- Messages are sent to a shared data structure consisting of **queues(Mailboxes)** that can temporarily hold messages
- Greater flexibility

Mutual Exclusion Using Messages

```
receive (mailbox, msg);
/* critical section */;
send (mailbox, msg);
.
.
```

Producer/Consumer Bounded Buffer Using Messages

```
const int capacity = /*buffer capacity*/,
null = /*empty message*/;
```

Producer:

```
message pmsg;
While (true) {
     recieve(mayproduce, pmsg);
     pmsg = produce();
     send(mayconsume, pmsg);
}
```

Consumer

```
message cmsg;
While (true) {
     recieve(mayconsume, cmsg);
     consume(cmsg);
     send(mayproduce, null);
}
```

Main

```
create_mailbox(mayproduce);
create_mailbox(mayconsume);
for(int i = 1; i <= capacity; i++) send(mayproduce, null);
parbegin(producer, consumer);
```

Readers/Writers Problem

In a shared memory among readers and writers,
Conditions that must be satisfied:

1. Any number of readers may simultaneously read the file
2. Only one writer at a time may write to the file
3. If a writer is writing to the file, no reader may read it

Semaphores Readers have priority

- The semaphore ***wsem*** is used to enforce mutual exclusion.
- when there are no readers reading —> the first reader should wait on wsem.
- subsequent readers need not wait before entering.

(Once a single reader has begun to access the data area, as long as there is at least one reader in the act of reading, they have control)

writers are subject to starvation.

Semaphores Writers have priority

```
int readcount, writecount;
Semaphore controlReadCount = 1, controlWriteCount = 1;
Semaphore mayread = 1, maywrite = 1;
Semaphore newreaders = 1;
```

Reader while true:

```
semWait(newreaders);
semWait(readers);
semWait(controlReadCount);
readcount++;
if(readcount == 1)
      semWait(maywrite);
semSignal(controlReader);
semSignal(mayread);
semSignal(newreaders);
READUNIT();
semWait(controlReadCount);
readcount−−;
if(readcount == 0)
      semSignal(maywrite);
semSignal(contolReaderCount);
```

Writer while true:

```
semWait(controlWriteCount);
writecount++;
if(writecount == 1)
      semWait(mayread);
semSignal(controlWriteCount);
semWait(maywrite);
WRITEUNIT();
semSignal(maywrite);
semWait(controlWriteCount);
writecount−−;
if(writecount == 0)
      semSignal(mayread);
semSignal(controlWriteCount);
```

Main

```
readcount = writecount = 0;
parbegin(reader, writer);
```

Readers only in the system	• *maywrite locked* • no queues
Writers only in the system	• *maywrite* and *mayread* locked • writers queue on *maywrite*
Both readers and writers with read first	• one reader queues on *maywrite* • *maywrite locked* by reader • *mayread locked* by writer until writers finish • all writers queue on *maywrite* • other readers queue on *newreaders*
Both readers and writers with write first	• *maywrite* locked and set by writer • writers queue on *maywrite* • one reader queues on *maywrite* • other readers queue on *newreaders* • *mayread* set by writer