Operating systems

Lecture 4, 5, 6

Michal Vrábel, 06/11/2019 - 20/11/2019

Concurrency

The central themes of operating system design are all concerned with the management of processes and threads:

- **Multiprogramming:** The management of multiple processes within a uniprocessor system
- **Multiprocessing**: The management of multiple processes within a multiprocessor
- **Distributed processing:** The management of multiple processes executing on multiple, distributed computer systems. The recent proliferation of clusters is a prime example of this type of system.

Concurrency arises in three different contexts:

- **Multiple applications:** Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.
- **Structured applications:** As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.
- **Operating system structure:** The same structuring advantages apply to systems programs, and we have seen that operating systems are themselves often implemented as a set of processes or threads.

Key Terms Related to Concurrency

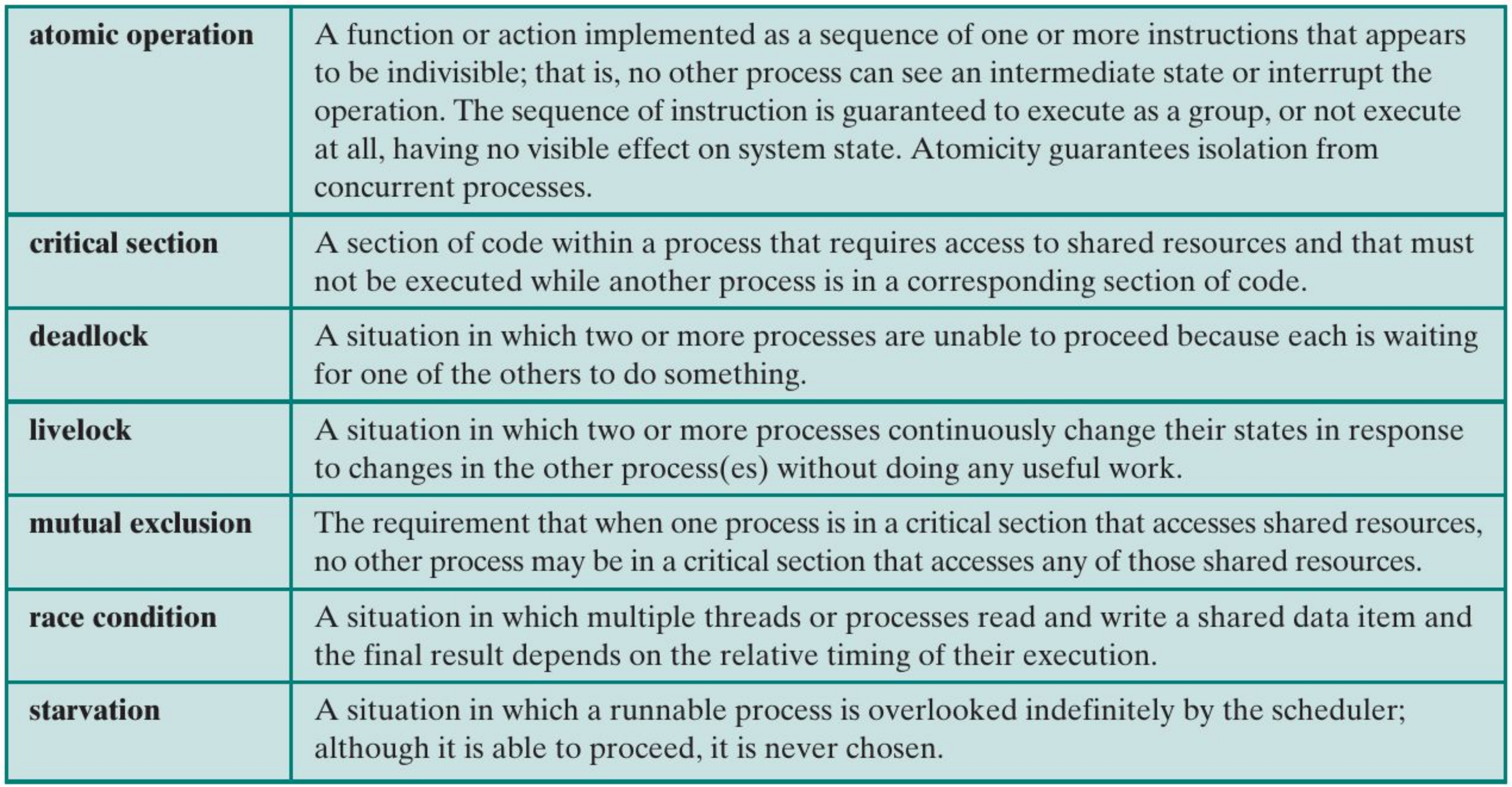
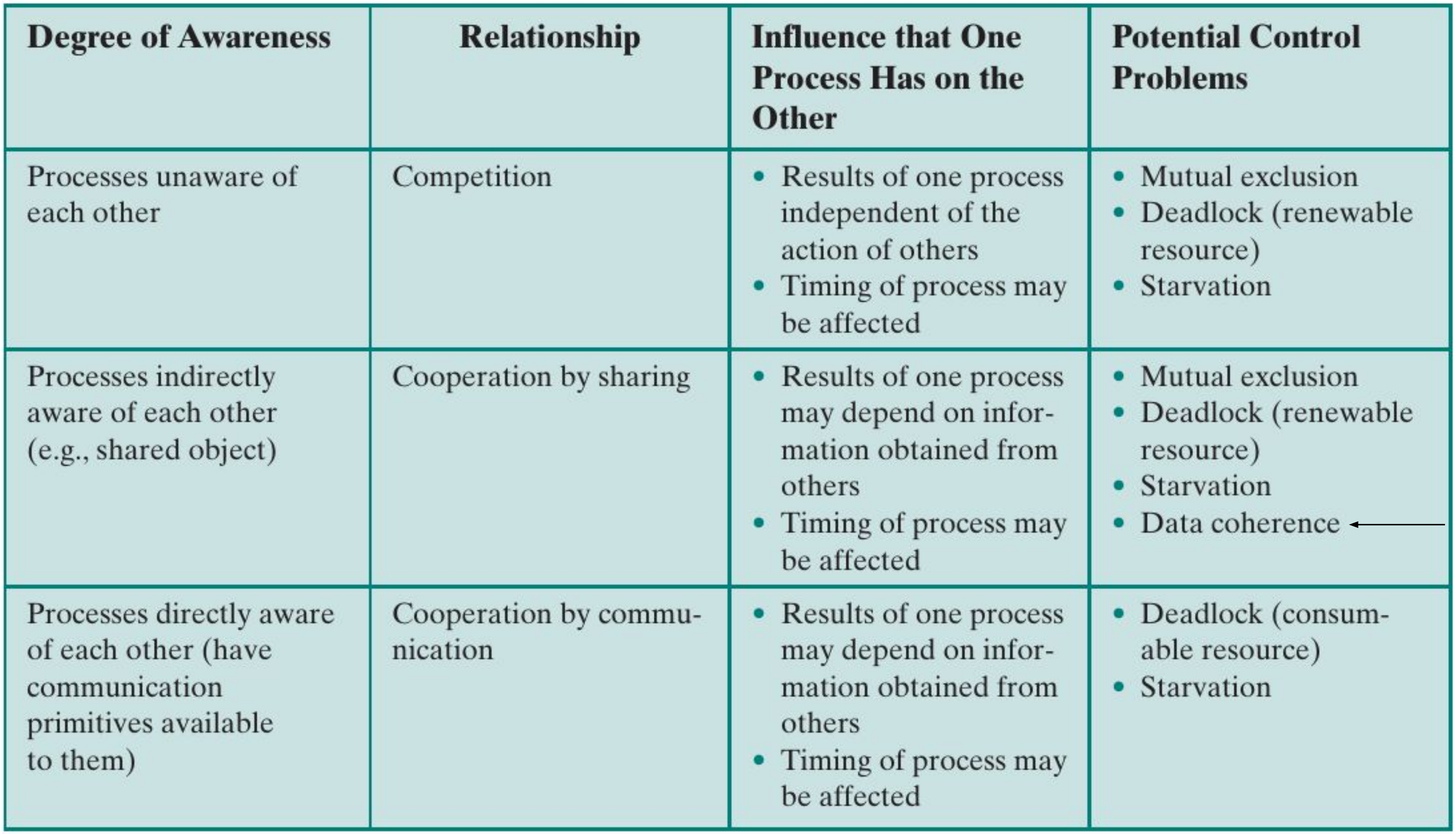


Table 5.1 Some Key Terms Related to Concurrency

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Process Interaction

Table 5.2 Process Interaction

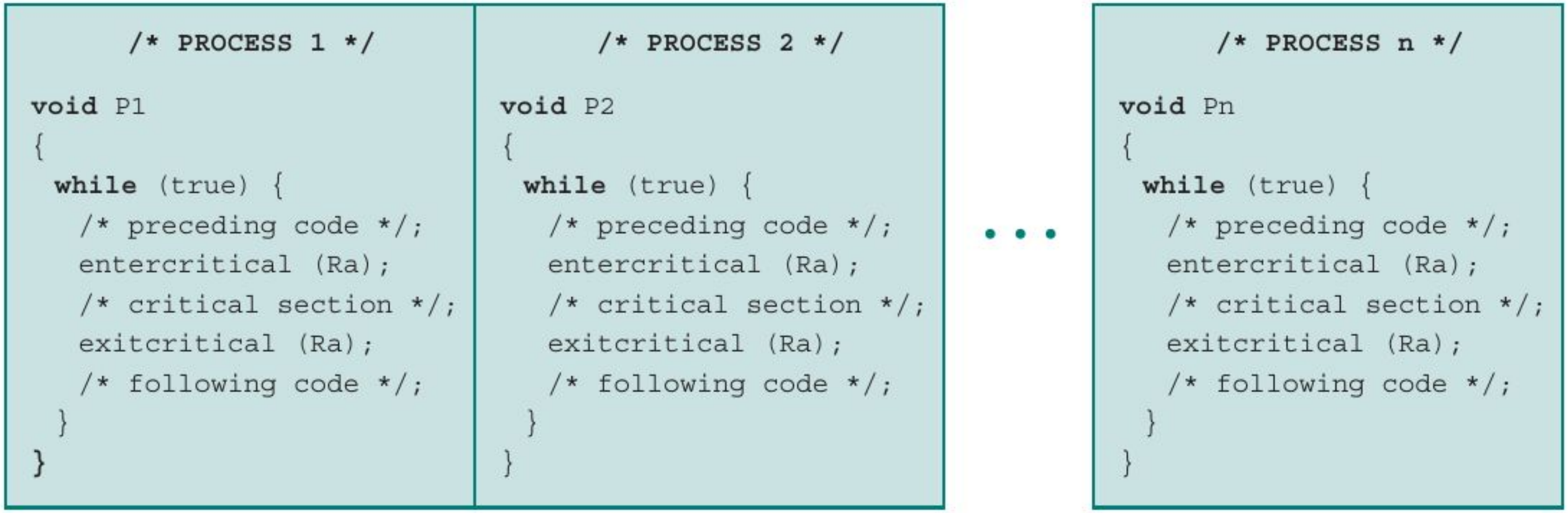


Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Competition among processes for resources

- **Critical resource**
  - a single nonsharable resource required by several processes
- **Critical section** of the program
  - the portion of the program that uses the critical resource
- In the case of competing processes three control problems must be faced.
  - Need for **mutual exclusion**
  - Deadlock
  - Starvation

Mutual exclusion



```
/* PROCESS 1 */

void P1
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

```
/* PROCESS 2 */

void P2
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

• • •

```
/* PROCESS n */

void Pn
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

Figure 5.1 **Illustration of Mutual Exclusion**

Mutual exclusion: hardware support

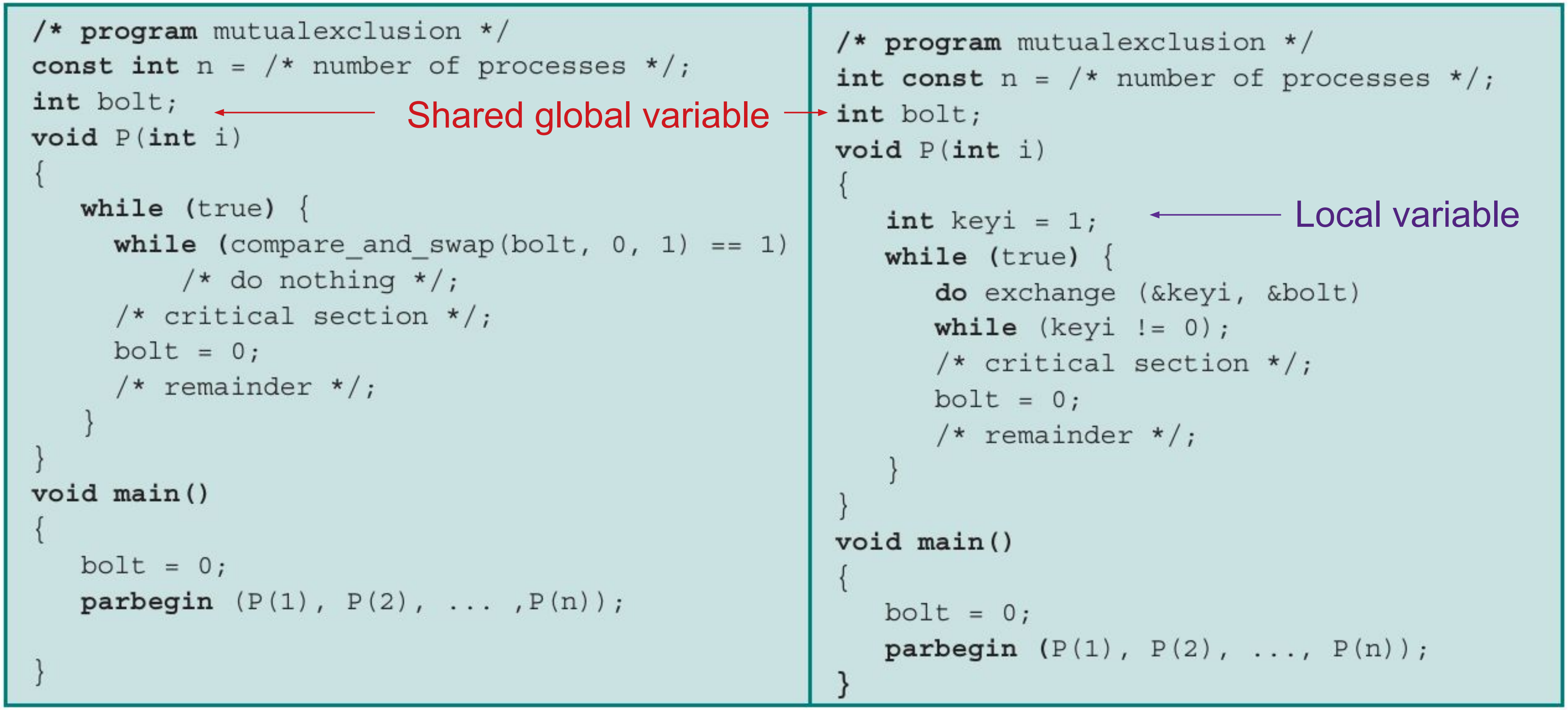
- **Interrupt Disabling** - In a uniprocessor system, concurrent processes cannot have overlapped execution
- **Special Machine Instructions**
  - Compare&swap instruction,
  - Exchange instruction

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

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

```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
    return oldval;
}
```

Hardware Support for Mutual Exclusion


Shared global variable
Local variable

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

}
```

(a) Compare and swap instruction

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

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Mutual exclusion: hardware support - compare_and_swap

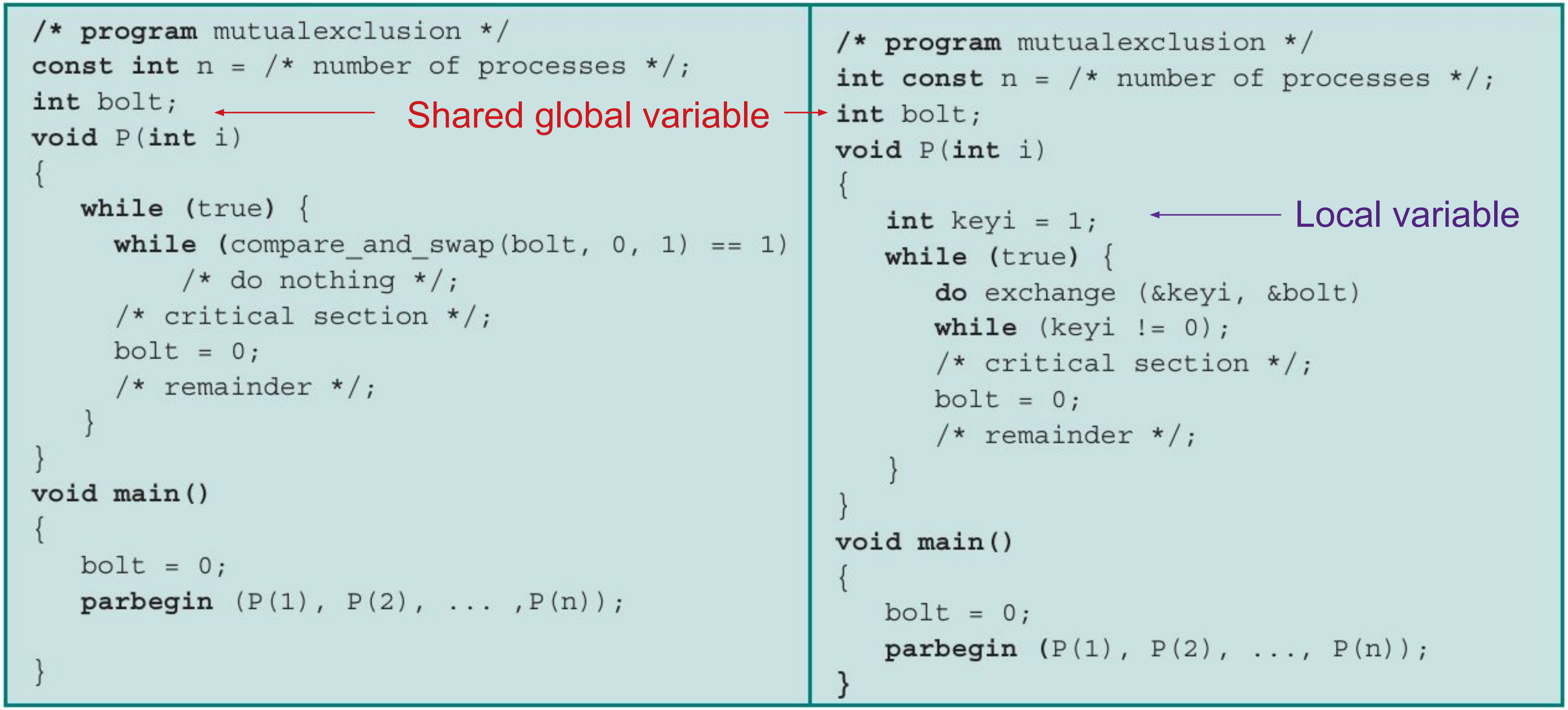
```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
    return oldval;
}
```

```
while (compare_and_swap(bolt, 0, 1) == 1)
    /* do nothing */;
/* critical section */;
bolt = 0;
/* remainder */;
```

Mutual exclusion: hardware support - compare_and_swap

- A **shared variable bolt is initialized to 0**.
- The only process that may enter its critical section is **one that finds bolt equal to 0**.
  - All other to enter their critical section go into a busy waiting mode.
    - or spin waiting, refers to a technique in which a process can do nothing until it gets permission to enter its critical section but continues to execute an instruction or set of instructions that tests the appropriate variable to gain entrance.
  - When a process leaves its critical section, **it resets bolt to 0;**
    - at this point one and only one of the waiting processes is granted access to its critical section.
- Another version of this instruction returns a Boolean value: true if the swap occurred; false otherwise.
- Some version of this instruction is available on nearly all processor families (x86, IA64, sparc, IBM z series, etc.), and most operating systems use this instruction for support of concurrency.

Hardware Support for Mutual Exclusion


Shared global variable
Local variable

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

}
```

(a) Compare and swap instruction

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

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Mutual exclusion: hardware support - exchange

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

```
int keyi = 1;
while (true) {
    do exchange (&keyi, &bolt)
    while (keyi != 0);
    /* critical section */;
    bolt = 0;
    /* remainder */;
```

Local variable

Shared global variable

Mutual exclusion: hardware support - exchange

- A **shared variable bolt is initialized to 0**.
- Each process uses a **local variable key** that **is initialized to 1**.
- The only process that may enter its critical section is **one that finds bolt equal to 0**.
  - It excludes all other processes from the critical section by setting bolt to 1.
  - When a process leaves its critical section, it resets bolt to 0, allowing another process to gain access to its critical section.
- exchanges the contents of a register with that of a memory location. Both the Intel IA-32 architecture (Pentium) and the IA-64 architecture (Itanium) contain an XCHG instruction.

Properties of the machine - instruction approach

- Advantages
  - any number of processes; It is simple - easy to verify; can be used to support multiple critical sections.
- **Disadvantages**
  - **Busy waiting is employed** - consumes processor time
  - **Starvation** is possible
  - **Deadlock** is possible
    - Process P1 executes the special instruction and enters its critical section.
    - P1 is then **interrupted** to give the processor to P2, which has higher priority.
    - If **P2 now attempts to use the same resource as P1**, it will be denied access because of the mutual exclusion mechanism.
      - Thus, it will go into a busy waiting loop.
      - However, P1 will never be dispatched because it is of lower priority than another ready process, P2.

Table 5.3 Common Concurrency Mechanisms

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. Also known as a **counting semaphore** or a **general semaphore.**
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 zero) 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 associate 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

```
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**

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count++;
   if (s.count<= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

(a) Compare and Swap Instruction

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts*/;
   }
   else
      allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count<= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

(b) Interrupts

Figure 5.14 Two Possible Implementations of Semaphores

Semaphores – Mutual exclusion

```
/* 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**

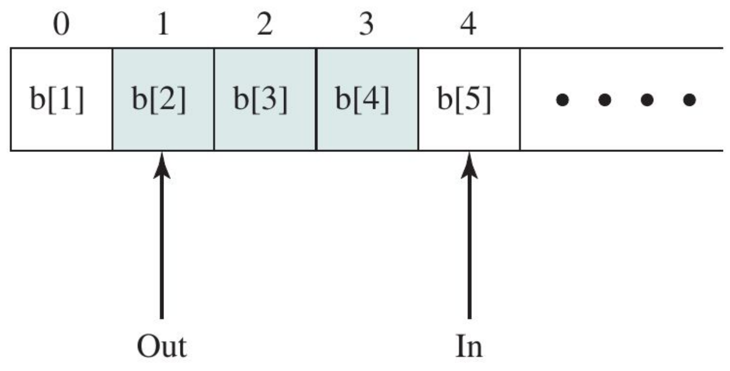
The Producer/Consumer Problem

- one or more producers generating some type of data (records, characters) and placing these in a buffer
- There is a single consumer that is taking items out of the buffer one at a time.

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

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

```
/* program producerconsumer */
     int n;
     binary_semaphore s = 1, 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);
     }
```

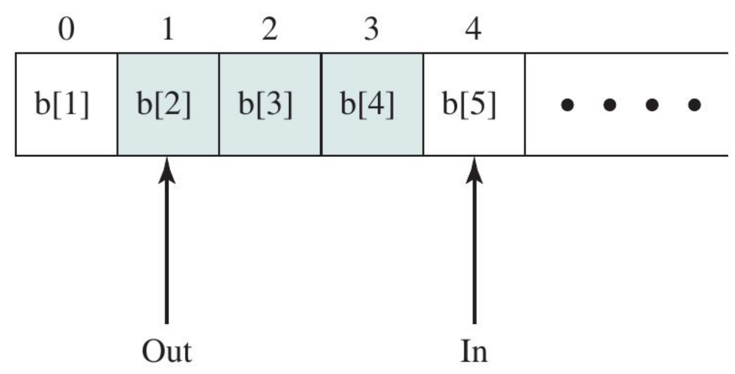


Note: Shaded area indicates portion of buffer that is occupied

Figure 5.8 **Infinite Buffer for the Producer/Consumer Problem**

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

```
/* program producerconsumer */
      int n;
      binary_semaphore s = 1, 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);
      }
```



Note: Shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

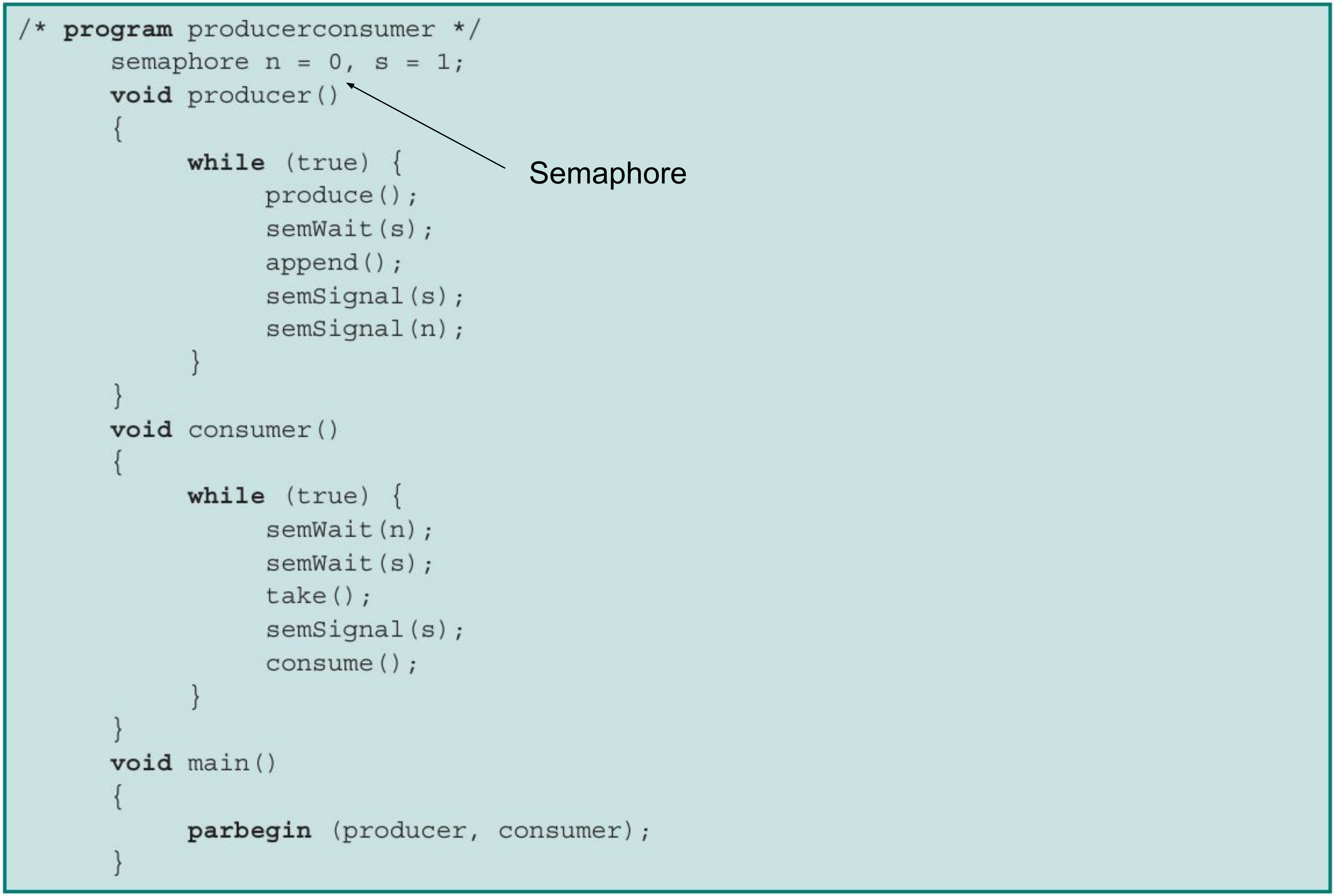
When the consumer has exhausted the buffer, it needs to reset the delay semaphore so that it will be forced to wait until the producer has placed more items in the buffer.

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

Table 5.4 Possible Scenario for the Program of Figure 5.9

	Producer	Consumer	s	n	Delay
1			1	0	0
2	`semWaitB(s)`		0	0	0
3	`n++`		0	1	0
4	`if (n==1) (semSignalB(delay))`		0	1	1
5	`semSignalB(s)`		1	1	1
6		`semWaitB(delay)`	1	1	0
7		`semWaitB(s)`	0	1	0
8		`n--`	0	0	0
9		`semSignalB(s)`	1	0	0
10	`semWaitB(s)`		0	0	0
11	`n++`		0	1	0
12	`if (n==1) (semSignalB(delay))`		0	1	1
13	`semSignalB(s)`		1	1	1
14		`if (n==0) (semWaitB(delay))`	1	1	1
15		`semWaitB(s)`	0	1	1
16		`n--`	0	0	1
17		`semSignalB(s)`	1	0	1
18		`if (n==0) (semWaitB(delay))`	1	0	0
19		`semWaitB(s)`	0	0	0
20		`n--`	0	–1	0
21		`semSignalB(s)`	1	–1	0

Note: White areas represent the critical section controlled by semaphore `s`.



```
/* program producerconsumer */
     semaphore n = 0, 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);
     }
```

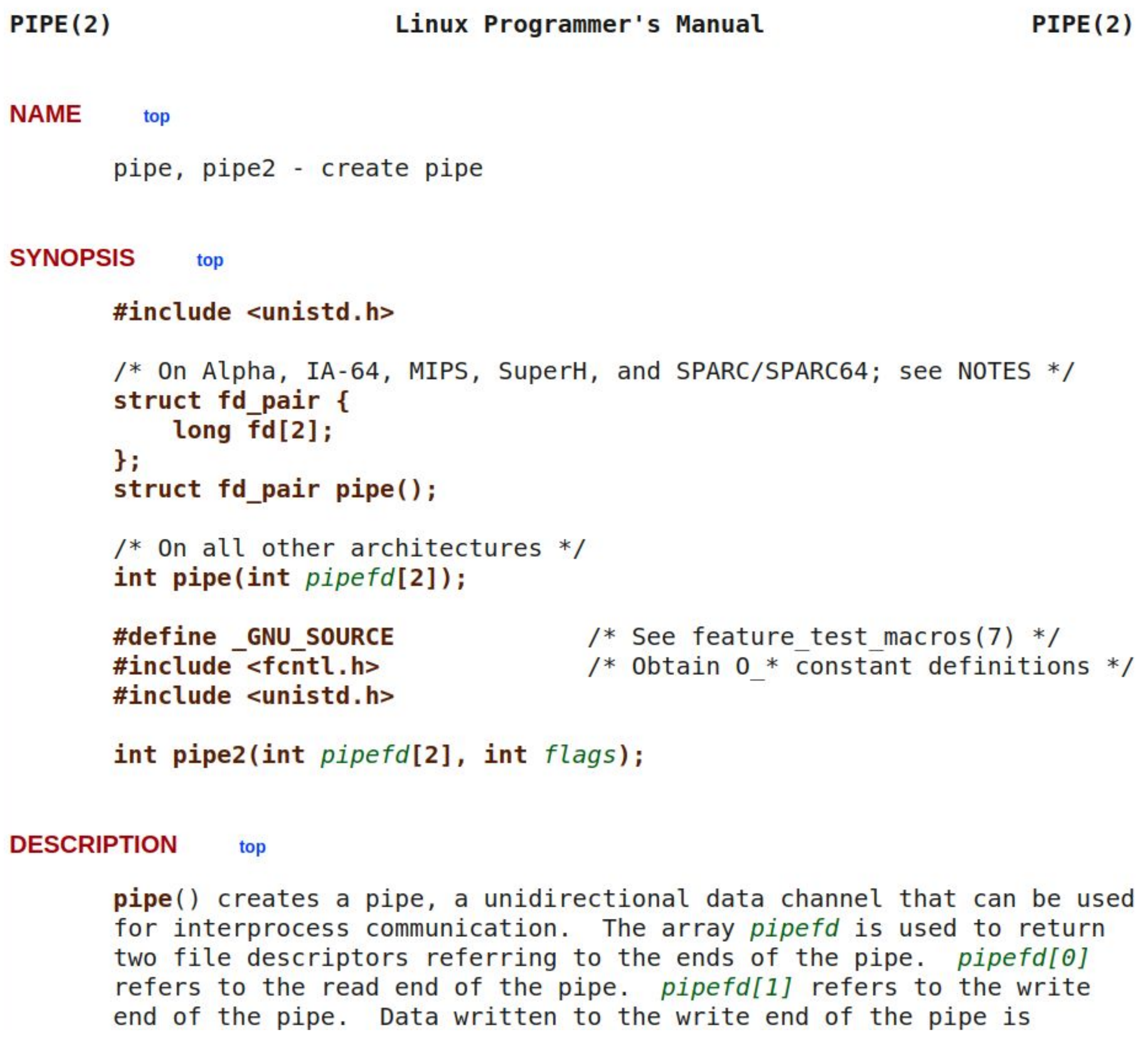
Semaphore

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

```
/* program boundedbuffer */
      const int sizeofbuffer = /* buffer size */;
      semaphore s = 1, n = 0, 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**

Pipes



```
PIPE(2)                 Linux Programmer's Manual                PIPE(2)
```

NAME top

```
       pipe, pipe2 - create pipe
```

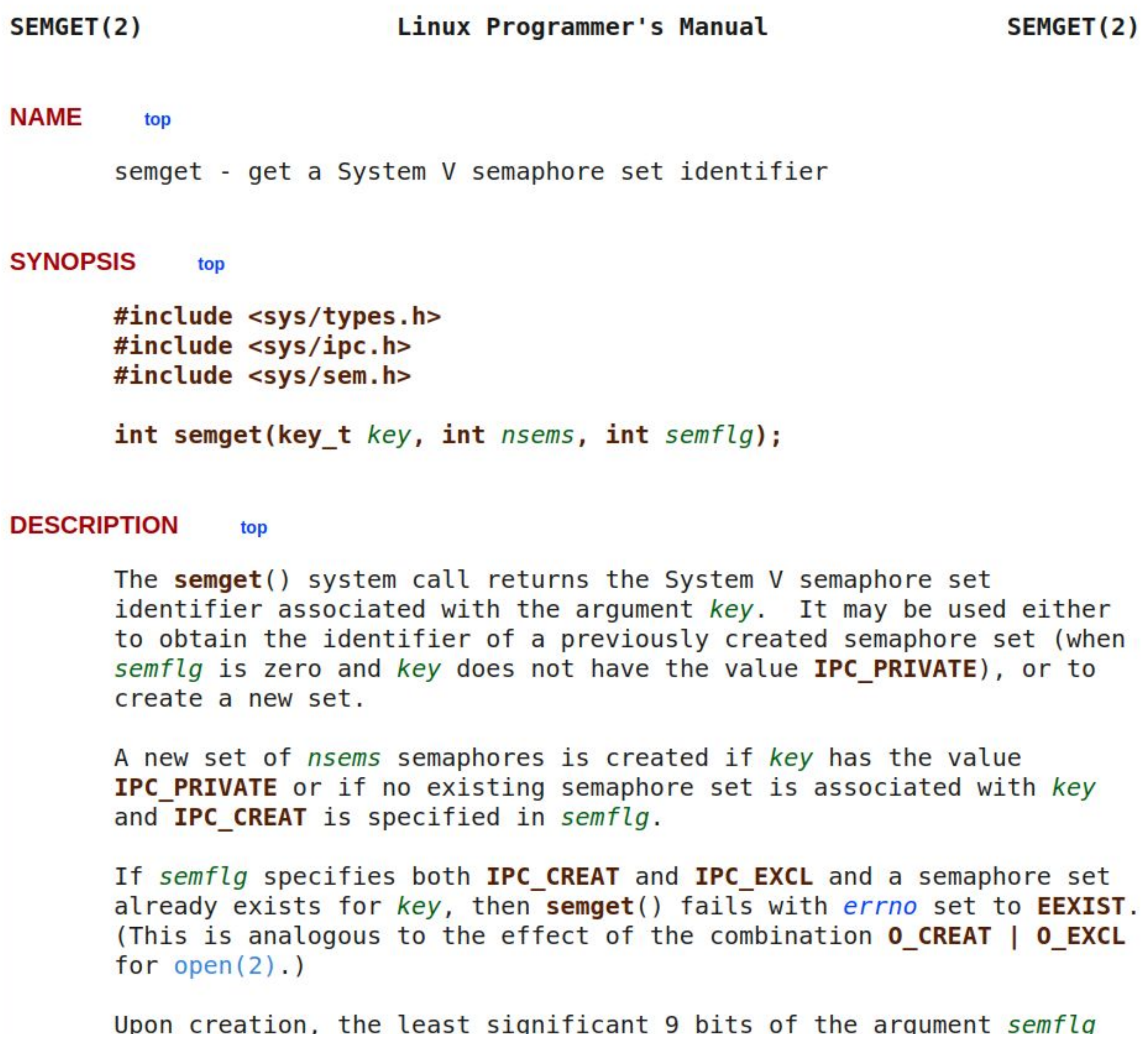
SYNOPSIS top

```
       #include <unistd.h>

       /* On Alpha, IA-64, MIPS, SuperH, and SPARC/SPARC64; see NOTES */
       struct fd_pair {
           long fd[2];
       };
       struct fd_pair pipe();

       /* On all other architectures */
       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);
```

DESCRIPTION top

```
       pipe() creates a pipe, a unidirectional data channel that can be used
       for interprocess communication.  The array pipefd is used to return
       two file descriptors referring to the ends of the pipe.  pipefd[0]
       refers to the read end of the pipe.  pipefd[1] refers to the write
       end of the pipe.  Data written to the write end of the pipe is
```

Semaphores in POSIX (system calls)



SEMGET(2) Linux Programmer's Manual SEMGET(2)

NAME top

semget - get a System V semaphore set identifier

SYNOPSIS top

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

DESCRIPTION top

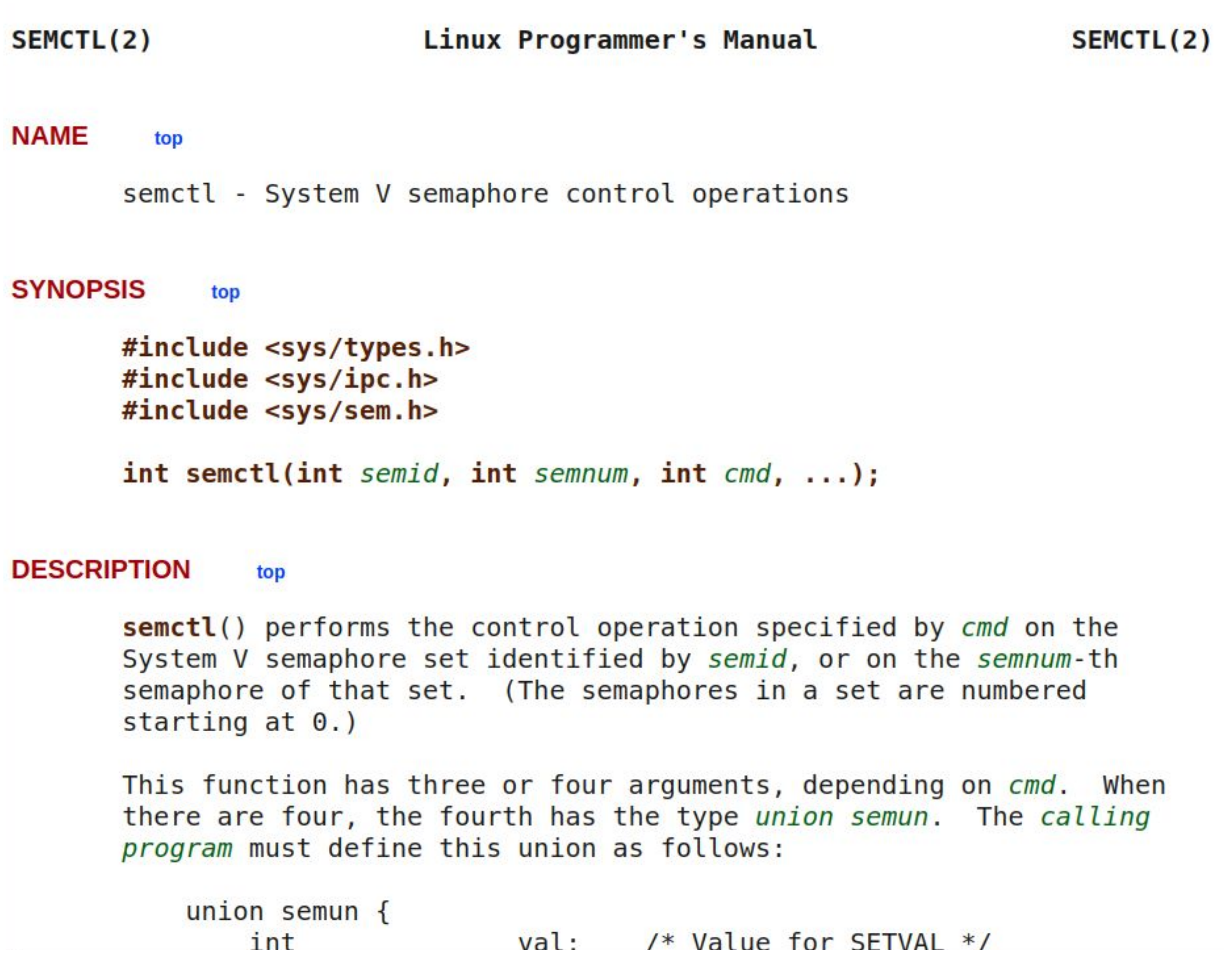
The **semget**() system call returns the System V semaphore set identifier associated with the argument *key*. It may be used either to obtain the identifier of a previously created semaphore set (when *semflg* is zero and *key* does not have the value **IPC_PRIVATE**), or to create a new set.

A new set of *nsems* semaphores is created if *key* has the value **IPC_PRIVATE** or if no existing semaphore set is associated with *key* and **IPC_CREAT** is specified in *semflg*.

If *semflg* specifies both **IPC_CREAT** and **IPC_EXCL** and a semaphore set already exists for *key*, then **semget**() fails with *errno* set to **EEXIST**. (This is analogous to the effect of the combination **O_CREAT | O_EXCL** for open(2).)

Upon creation, the least significant 9 bits of the argument *semflg*

Semaphores in POSIX (system calls)



```
SEMCTL(2)                 Linux Programmer's Manual                 SEMCTL(2)
```

NAME top

```
       semctl - System V semaphore control operations
```

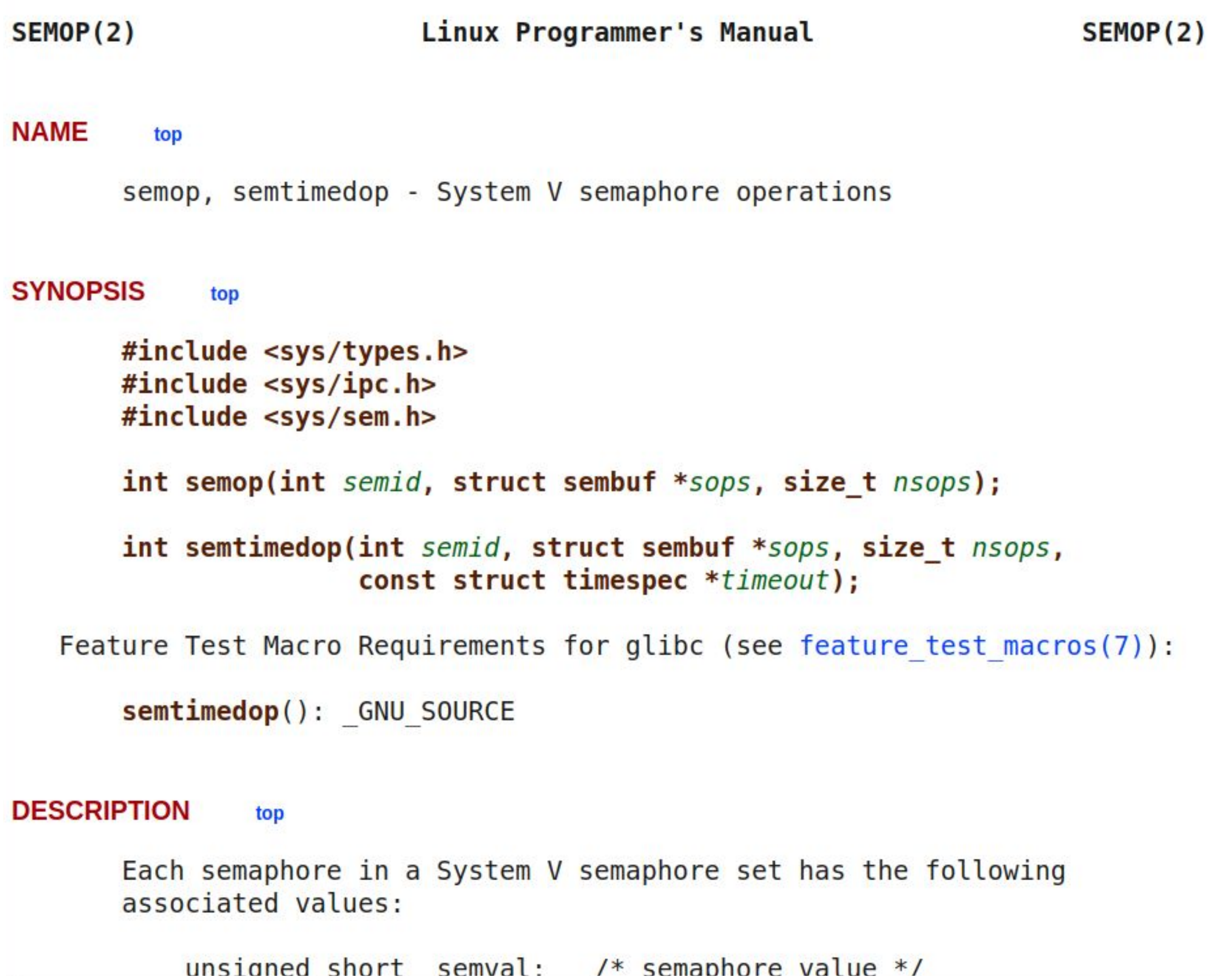
SYNOPSIS top

```
       #include <sys/types.h>
       #include <sys/ipc.h>
       #include <sys/sem.h>

       int semctl(int semid, int semnum, int cmd, ...);
```

DESCRIPTION top

```
       semctl() performs the control operation specified by cmd on the
       System V semaphore set identified by semid, or on the semnum-th
       semaphore of that set.  (The semaphores in a set are numbered
       starting at 0.)

       This function has three or four arguments, depending on cmd.  When
       there are four, the fourth has the type union semun.  The calling
       program must define this union as follows:

           union semun {
               int              val;    /* Value for SETVAL */
```

Semaphores in POSIX (system calls)



SEMOP(2) Linux Programmer's Manual SEMOP(2)

NAME top

semop, semtimedop - System V semaphore operations

SYNOPSIS top

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);

int semtimedop(int semid, struct sembuf *sops, size_t nsops,
               const struct timespec *timeout);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

semtimedop(): _GNU_SOURCE

DESCRIPTION top

Each semaphore in a System V semaphore set has the following associated values:

```
    unsigned short  semval;   /* semaphore value */
```