UNIT-3
INTERPROCESS COMMUNICATION MECHANISMS
Cooperating processes

Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

Independent Process: A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.

Cooperating Process: A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

Reasons for providing cooperation

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads,
- **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

IPC between processes on a single computer system, IPC between processes on different systems

Cooperating processes require an **inter-process communication (IPC)** mechanism that will allow them to exchange data and information.

The following are the different forms of IPC mechanisms,

- Pipes
- FIFOs
- Message Queues
- Shared memory
- Sockets
- Steams

The first 4 are usually restricted to IPC between processes on the same host. The final two are the only support IPC between processes on different hosts.

Pipes

Definition: A **pipe** acts as a conduit or channel allowing two processes to communicate. Pipes are the oldest form of IPC

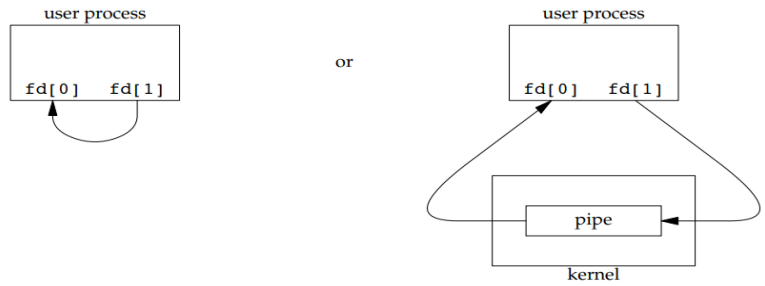
Common types of pipes

Two common types of pipes used on both UNIX and Windows systems:

- Ordinary pipes or Pipes
- Named pipes or FIFOs

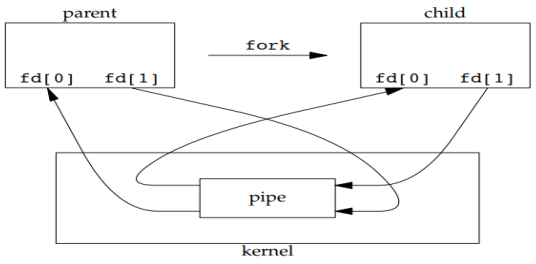
Ordinary pipes or Pipes

- Ordinary pipes allow two processes to communicate in standard producer– consumer fashion: the producer writes to one end of the pipe (the **write-end**) and the consumer reads from the other end (the **read-end**).
- Ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.
- On UNIX systems, A pipe is created by calling the pipe function
  General Form: int pipe(int fd[2]);
  Two file descriptors are returned through the fd argument: fd[0] is open for reading, and fd[1] is open for writing. The output of fd[1] is the input for fd[0].
- Two ways to picture a half-duplex pipe



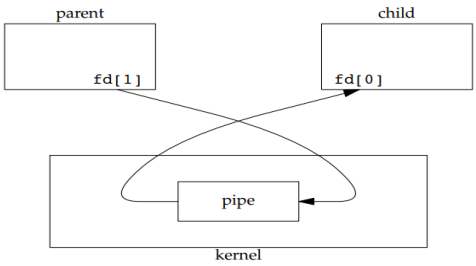
Two ways to view a half-duplex pipe

- A pipe in a single process is next to useless.
- The process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa.



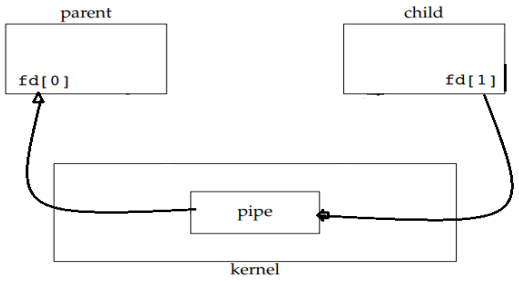
Half-duplex pipe after a fork

- What happens after the fork depends on which direction of data flow we want.
  a) For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]).



Pipe from parent to child

b) For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].



Pipe from child to parent

- When one end of a pipe is closed, two rules apply.
  a) If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
  b) If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated which is either ignored or catched. PIPE_BUF specifies the kernel's pipe buffer size.
- Ordinary pipes on Windows systems are termed **anonymous pipes.** They employ parent–child relationships between the communicating processes. In addition, reading and writing to the pipe can be accomplished with the ordinary ReadFile () and WriteFile () functions. The Windows API for creating pipes is the CreatePipe() function.
- Ordinary pipes provide a simple mechanism for allowing a pair of processes to communicate. However, ordinary pipes exist only while the processes are communicating with one another. On both UNIX and Windows systems, once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist.

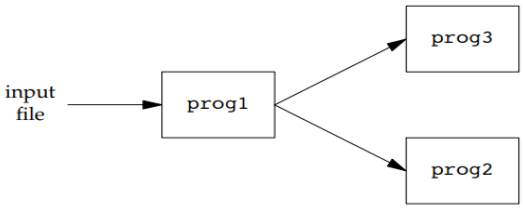
Named Pipes or FIFOs

- Named pipes are referred to as FIFOs in UNIX systems.
- Both UNIX and Windows systems support named pipes.
- Named pipes provide a much more powerful communication tool.
- Communication can be bidirectional
- Unnamed pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, however, unrelated processes can exchange data.
- Creating a FIFO is similar to creating a file.
  General Form: int mkfifo(const char *path, mode_t mode);
- Once we have used mkfifo to create a FIFO, we open it using open. Normal file I/O functions(close, read,write,unlink etc) all work with FIFOs.
- There are two uses for FIFOs.

a) Duplication of Output stream
b) Client server communication

- **Using FIFOs to Duplicate Output Streams**

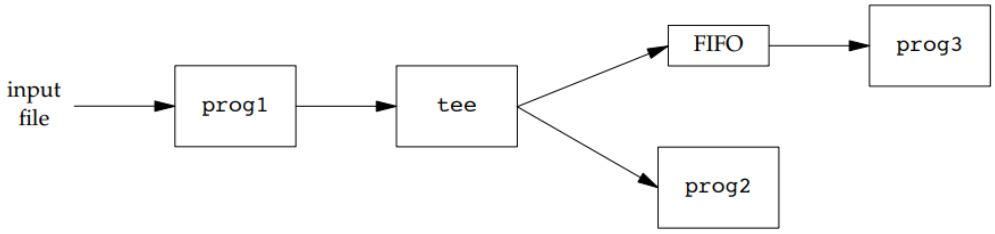
FIFOs can be used to duplicate an output stream and can be used for nonlinear connections.



With a FIFO and the UNIX program tee(1), we can accomplish this procedure without using a temporary file. (The tee program copies its standard input to both its standard output and the file named on its command line.)

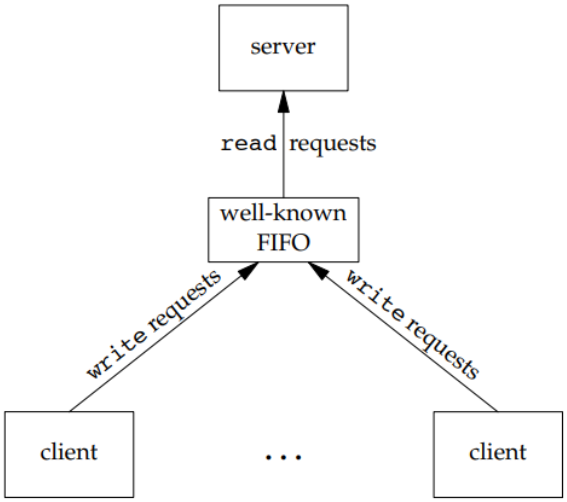
```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

We create the FIFO and then start prog3 in the background, reading from the FIFO. We then start prog1 and use tee to send its input to both the FIFO and prog2.



- **Client–Server Communication Using a FIFO**

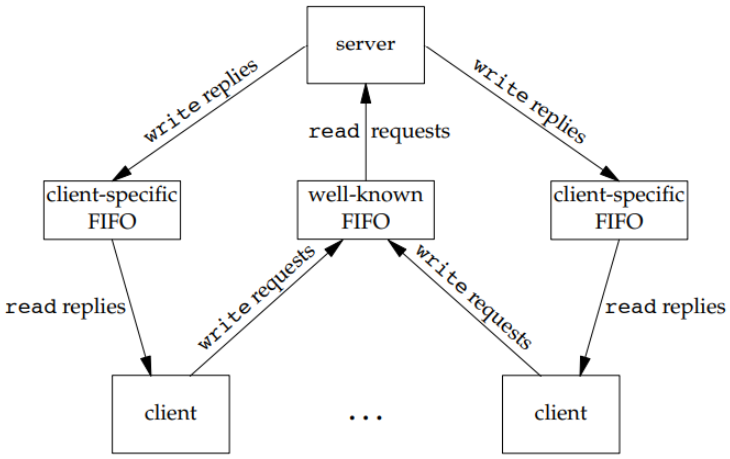
Another use for FIFOs is to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates.



Clients sending requests to a server using a FIFO

The problem in using FIFOs for this type of client–server communication is how to send replies back from the server to each client. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One

solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client's process ID.



Client–server communication using FIFOs

- Although FIFOs allow bidirectional communication, only half-duplex transmission is permitted. If data must travel in both directions, two FIFOs are typically used. Additionally, the communicating processes must reside on the same machine. If inter machine communication is required, sockets must be used.
- Named pipes on Windows systems provide a richer communication mechanism than their UNIX counterparts.
- Full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines.
- Named pipes are created with the CreateNamedPipe () function and a client can connect to a named pipe using ConnectNamedPipe ().
- Communication over the named pipe can be accomplished using the ReadFile () and WriteFile () functions.

Message Queues

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.
- Each queue has the following msqid_ds structure associated with it:

```
struct msqid_ds
 {
 struct ipc_perm msg_perm;              /* defines permissions */
msgqnum_t msg_qnum;                  /* # of messages on queue */
msglen_t msg_qbytes;                    /* max # of bytes on queue */
pid_t msg_lspid;                     /* pid of last msgsnd() */
 pid_t msg_lrpid;                     /* pid of last msgrcv() */
time_t msg_stime;                       /* last-msgsnd() time */
time_t msg_rtime;                       /* last-msgrcv() time */
time_t msg_ctime;                       /* last-change time */
 .
 .
 .
```

```
};
```

This structure defines the current status of the queue.

- **msgget**

The first function normally called is msgget to either open an existing queue or create a new queue.

General Form: int msgget(key_t key, int flag);

Key is converted into identifier of the process and used to decide whether a new queue is created or an existing queue is referenced.

When a new queue is created, the following members of the msqid_ds structure are initialized.

- The mode member of this structure is set to the corresponding permission bits of flag.
- msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are all set to 0.
- msg_ctime is set to the current time.
- msg_qbytes is set to the system limit.

On success, msgget returns the non-negative queue ID. This value is then used with the other three message queue functions.

- **msgsnd**

Data is placed onto a message queue by calling msgsnd.

General Form: int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);

Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data.

Message structure:

```
struct mymesg
 {
 long mtype;              /* positive message type */
char mtext[512];          /* message data, of length nbytes */
};
```

The ptr argument is then a pointer to a mymesg structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.

A flag value of IPC_NOWAIT can be specified, this causes msgsnd to return an error message when the queue is full.

When msgsnd returns successfully, the msqid_ds structure associated with the message queue is updated to indicate the process ID that made the call (msg_lspid), the time that the call was made (msg_stime), and that one more message is on the queue (msg_qnum).

- **msgrcv**

Messages are retrieved from a queue by msgrcv.

General Form: ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);

the ptr argument points to a long integer (where the message type of the returned message is stored) followed by a data buffer for the actual message data. nbytes specifies the size of the data buffer.

If the returned message is larger than nbytes and the MSG_NOERROR bit in flag is set, the message is truncated.

The type argument lets us specify which message we want.

type == 0	The first message on the queue is returned.
type >0	The first message on the queue whose message type equals type is returned.
type <0	The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.
type= nonzero	Used to read the messages in an order other than first in, first out such as using priority

We can specify a flag value of IPC_NOWAIT to make the operation nonblocking, If IPC_NOWAIT is not specified, the operation blocks until a message of the specified type is available, the queue is removed from the system (−1 is returned with errno set to EIDRM), or a signal is caught and the signal handler returns

When msgrcv succeeds, the kernel updates the msqid_ds structure associated with the message queue to indicate the caller's process ID (msg_lrpid), the time of the call (msg_rtime), and that one less message is on the queue (msg_qnum).

- **msgctl**

The msgctl function performs various operations on a queue.

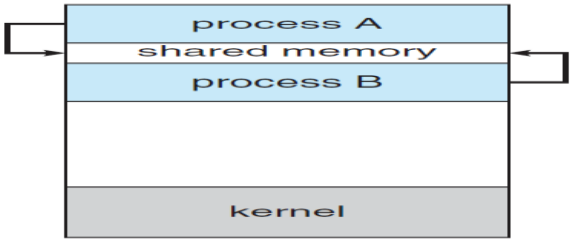
General Form: int msgctl(int msqid, int cmd, struct msqid_ds *buf);

The cmd argument specifies the command to be performed on the queue specified by msqid.

IPC_STAT	Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf.
IPC_SET	Copy the following fields from the structure pointed to by buf to the msqid_ds structure associated with this queue: msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes.
IPC_RMID	Remove the message queue from the system and any data still on the queue. This removal is immediate.

Shared memory

- Shared memory allows two or more processes to share a given region of memory.
- This is the fastest form of IPC, because the data does not need to be copied between the client and the server.
- The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access.



- The kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds
{
struct ipc_perm shm_perm;          /* defines permissions */
size_t shm_segsz;                  /* size of segment in bytes */
pid_t shm_lpid;                    /* pid of last shmop() */
pid_t shm_cpid;                    /* pid of creator */
shmatt_t shm_nattch;               /* number of current attaches */
time_t shm_atime;                  /* last-attach time */
time_t shm_dtime;                  /* last-detach time */
 time_t shm_ctime;                 /* last-change time */
.
.
.
};
```

- **shmget**

The first function called is usually shmget, to obtain a shared memory identifier

General Form: int shmget(key_t key, size_t size, int flag);

Key is converted into an identifier and whether a new segment is created or an existing segment is referenced.

When a new segment is created, the following members of the shmid_ds structure are initialized.

- The ipc_perm structure is initialized . The mode member of this structure is set to the corresponding permission bits of flag.
- shm_lpid, shm_nattch, shm_atime, and shm_dtime are all set to 0.
- shm_ctime is set to the current time.
- shm_segsz is set to the size requested.

The size parameter is the size of the shared memory segment in bytes.

- **shmctl**

The shmctl function is the catchall for various shared memory operations.

General Form: **int shmctl(int shmid, int cmd, struct shmid_ds *buf);**

The cmd argument specifies one of the following five commands to be performed, on the segment specified by shmid.

IPC_STAT Fetch the shmid_ds structure for this segment, storing it in the structure pointed to by buf.

IPC_SET	Set the following three fields from the structure pointed to by buf in the shmid_ds structure associated with this shared memory segment: shm_perm.uid, shm_perm.gid, and shm_perm.mode.
IPC_RMID	Remove the shared memory segment set from the system.

Two additional commands are provided by Linux and Solaris, but are not part of the Single UNIX Specification.

SHM_LOCK	Lock the shared memory segment in memory. This command can be executed only by the superuser.
SHM_UNLOCK	Unlock the shared memory segment. This command can be executed only by the superuser

- **shmat**

Once a shared memory segment has been created, a process attaches it to its address space by calling shmat.

General Form: void *shmat(int shmid, const void *addr, int flag);

The address in the calling process at which the segment is attached depends on the addr argument and whether the SHM_RND bit is specified in flag.

- If addr is 0, the segment is attached at the first available address selected by the kernel. This is the recommended technique.
- If addr is nonzero and SHM_RND is not specified, the segment is attached at the address given by addr.
- If addr is nonzero and SHM_RND is specified, the segment is attached at the address given by (addr − (addr modulus SHMLBA)). The SHM_RND command stands for ‘‘round.’’ SHMLBA stands for ‘‘low boundary address multiple’’ and is always a power of 2.

The value returned by shmat is the address at which the segment is attached, or −1 if an error occurred.

If shmat succeeds, the kernel will increment the shm_nattch counter in the shmid_ds structure associated with the shared memory segment.

- **shmdt**

When we’re done with a shared memory segment, we call shmdt to detach it. This does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process (often a server) specifically removes it by calling shmctl with a command of IPC_RMID.

General Form: int shmdt(const void *addr);

The addr argument is the value that was returned by a previous call to shmat. If successful, shmdt will decrement the shm_nattch counter in the associated shmid_ds structure.

PROCESS MANAGEMENT AND SYNCHRONIZATION

Race Condition

Definition: A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.

To guard against the race condition we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

The Critical Section Problem

Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}. Each process has a segment of code, called a **critical section or region**, in which the process may be changing common variables, updating a table, writing a file, and so on.

The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

The ***critical-section problem*** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

General structure of process P_i

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Synchronization Hardware

Software-based solutions such as Peterson's does not guaranteed to work on modern computer architectures. The following are several more solutions to the critical-section

problem using techniques ranging from hardware to software-based on the premise of **locking** —that is, protecting critical regions through the use of locks.

Single Processor Systems: Disabling of Interrupts

The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption.

No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels.

```
Disable Interrupts
        Critical section
Enable Interrupts
        Remainder section
```

Multi Processor Systems: Special hardware instructions

Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

Many modern computer systems therefore provide special hardware instructions like "test _and _set" and "Compare _and_Swap" that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**—that is, as one uninterruptible unit.

1. **Test _and _Set**
   - The important characteristic of this instruction is that it is executed atomically.
   - Thus, if two test and set () instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
   - If the machine supports the test and set () instruction, then we can implement mutual exclusion by declaring a boolean variable lock, initialized to false.

Definition of the test_and _set() instruction:

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
*target = TRUE;
return rv:
}
```

Mutual-exclusion implementation with test _and _set ()

```
do
{
while (test_and_set (&lock))
        ; /* do nothing */
/* critical section */
lock = false;
/* remainder section */
```

```
} while (true);
```

2. Compare _and_Swap

- The compare and swap () instruction, in contrast to the test and set () instruction, operates on three operands.
- The operand value is set to new value only if the expression (*value == expected) is true. Regardless, compare and swap () always returns the original value of the variable value.

Definition of compare _and_swap

```
int compare _and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

Mutual-exclusion implementation with compare _and_swap

```
do
{
   while (compare_and_swap(&lock, 0, 1) != 0)
     ; /* do nothing */
        /* critical section */
  lock = 0;
    /* remainder section */
} while (true);
```

Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement. The following is another algorithm using the test and set () instruction that satisfies all the critical-section requirements.

```
do
{
    waiting[i]=true;
    key=true;
   while (waiting[i] && key)
      key = test_and_set (&lock);
     waiting[i] = false;
         /* critical section */
     j = (i + 1) % n;
    while ((j != i) && !waiting[j])
         j = (j + 1) % n;
     if (j == i)
        lock = false;
    else
       waiting[j] = false;
```

```
        /* remainder section */
} while (true);
```

Semaphores

Hardware based solutions to critical section problems are complicated, so we use a more robust software tool called Semaphore.

Definition: A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().

Definition of the wait () operation

```
Wait(S)
{
   while (S <= 0)
     ; // busy wait
   S--;
}
```

Definition of the signal () operation

```
Signal(S)
{
   S++;
}
```

1. **Semaphore Usage**

   Operating systems often distinguish between counting and binary semaphores.

   a. **Counting Semaphore**

   - The value of a **counting semaphore** can range over an unrestricted domain.
   - Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available.
   - Each process that wishes to use a resource performs a wait () operation on the semaphore (thereby decrementing the count).
   - When a process releases a resource, it performs a signal () operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

   b. **Binary Semaphore**

   - The value of a **binary semaphore** can range only between 0 and 1.
   - **Example:** Consider two concurrently running processes: *P*1 with a statement *S*1 and *P*2 with a statement *S*2. Suppose we require that *S*2 be executed only after *S*1 has completed. We can implement this scheme readily by letting *P*1 and *P*2 share a common semaphore synch, initialized to 0.

     In process *P*1, we insert the statements

     ```
     S1;
     signal (synch);
     ```

     In process *P*2, we insert the statements

```
wait (synch);
S2;
```

Because synch is initialized to 0, *P*2 will execute *S*2 only after *P*1 has invoked signal (synch), which is after statement *S*1 has been executed.

2. Semaphore Implementation

The previous definition of wait () and signal () has a problem of busy waiting which wastes CPU cycles. To overcome this, we will modify the definition of wait () and signal ().

Wait ()

- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.
- However, rather than engaging in busy waiting, the process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.

Signal ()

- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal () operation.
- The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state.
- The process is then placed in the ready queue.

Defining a semaphore

```
typedef struct
{
  int value;
  struct process *list;
} semaphore;
```

wait () semaphore operation can be defined as,

```
wait (semaphore *S)
{
   S->value--;
   if(S->value<0)
  {
     add this process to S->list;
     block ();
   }
}
```

signal () semaphore operation can be defined as,

```
signal (semaphore *S)
{
   S->value++;
```

```
    if(S->value<=0)
    {
       remove a process P from S->list;
       wakeup (P);
    }
  }
```

3. Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal () operation. When such a state is reached, these processes are said to be **deadlocked**.

Example: consider a system consisting of two processes, *P*0 and *P*1, each accessing two semaphores, S and Q, set to the value 1:

```
        P0                  P1
wait(S);            wait(Q);
wait(Q);            wait(S);
   .                   .
   .                   .
   .                   .
signal(S);          signal(Q);
signal(Q);          signal(S);
```

Suppose that *P*0 executes wait(S) and then *P*1 executes wait (Q).When *P*0 executes wait (Q), it must wait until *P*1 executes signal (Q). Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S). Since these signal () operations cannot be executed, *P*0 and *P*1 are deadlocked.

Another problem related to deadlocks is **indefinite blocking** or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Classic Problems of Synchronization

1. The Bounded-Buffer Problem

The producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The structure of the producer process

```
do
{
   ...
  /* produce an item in next_produced */
```

```
    ...
    wait (empty);
    wait (mutex);
      ...
    /* add next produced to the buffer */
      ...
    signal (mutex);
    signal (full);
  } while (true);
```

The structure of the consumer process

```
  do
  {
    wait (full);
    wait (mutex);
      ...
    /* remove an item from buffer to next_consumed */
      ...
    signal (mutex);
    signal (empty);
      ...
    /* consume the item in next consumed */
      ...
  } while (true);
```

2. The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as ***readers*** and to the latter as ***writers***. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, problems may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers problem**.

Variations in readers–writers problem

The readers–writers problem has several variations, all involving priorities.

- The simplest one, referred to as the ***first*** readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. In this writers may starve.

- The ***second*** readers –writer's problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading. In this readers may starve.

Solution to the first readers–writers problem

Data structures:

```
semaphore rw mutex = 1;
semaphore mutex = 1;
int read count = 0;
```

The semaphores mutex and rw mutex are initialized to 1; read count is initialized to 0. The semaphore rw mutex is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated. The read count variable keeps track of how many processes are currently reading the object. The semaphore rw mutex functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The structure of a writer process

```
do
{
   wait (rw_mutex);
     ...
     /* writing is performed */
     ...
  signal(rw_mutex);
} while (true);
```

The structure of a reader process

```
do
{
   wait(mutex);
   read_count++;
   if (read_count == 1)
     wait(rw_mutex);
   signal(mutex);
     ...
   /* reading is performed */
     ...
   wait(mutex);
   readcount--;
   if (read_count == 0)
   signal(rw_mutex);
   signal(mutex);
} while (true);
```

Reader–Writer Locks

The readers–writers problem and its solutions have been generalized to provide **reader–writer** locks on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either *read* or *write* access. When a process wishes only to read shared data, it requests the reader–writer lock in read mode. A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers.

3. The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the centre of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbours). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbour. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

Solution

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore. She releases her chopsticks by executing the signal () operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick [5];

where all the elements of chopstick are initialized to 1.

The structure of Philosopher *i*:

```
do
 {
   wait (chopstick[i]);
       wait (chopStick [(i + 1) % 5]);
```

```
        // eat for a while
    signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

            // think for a while
} while (TRUE);
```

Although this solution guarantees that no two neighbours are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in errors such as following,

- Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal (mutex);
...
critical section
...
wait (mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections.

- Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

```
wait (mutex);
...
critical section
...
wait (mutex);
```

In this case, a deadlock will occur.

- Suppose that a process omits the wait (mutex), or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

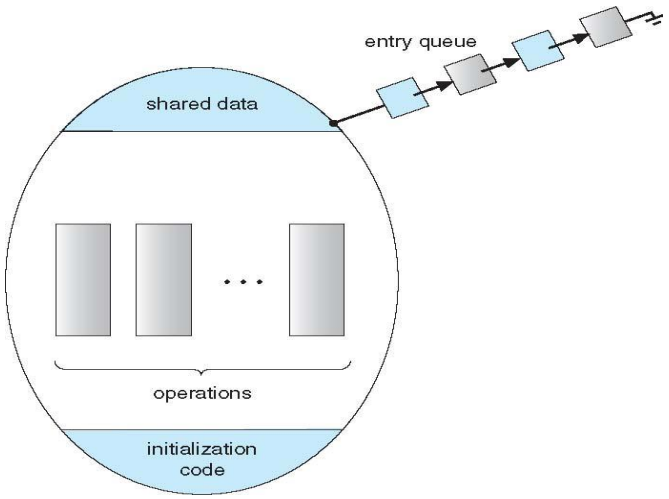
To deal with such errors, researchers have developed one fundamental high-level synchronization construct—the **monitor** type.

1. Monitor Usage

A ***monitor type*** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.

```
Monitor monitorname
{ /* shared variable declarations */
function P1 ( . . . ) { . . .
}
function P2 ( . . . ) { . . .
}
.
.
.
function Pn ( . . . ) { . . .
}
initialization code ( . . . ) { . . .
}
}
```

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly.



Condition Variables: The monitor construct is not sufficiently powerful for modelling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the ***condition*** construct,

```
condition x, y;
```

The only operations that can be invoked on a condition variable are wait () and signal (). The operation

```
x.wait ();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal ();
```

The x.signal () operation resumes exactly one suspended process. If no process is suspended, then the signal () operation has no effect.

2. Dining-Philosophers Solution Using Monitors

Monitor concepts presents a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

The following are the data structure:

enum {THINKING, HUNGRY, EATING} state [5];

Philosopher *i* can set the variable state[i] = EATING only if her two neighbours are not eating: (state[(i+4) % 5] != EATING) and(state[(i+1) % 5] != EATING).

We also need to declare: condition self [5];

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

```
monitor DiningPhilosophers
{
        enum {THINKING, HUNGRY, EATING} state [5];
        condition self [5];
        void pickup (int i)
        {
                state[i] = HUNGRY;
                test (i);
                if (state[i]! = EATING)
                self[i].wait ();
        }
        void putdown (int i)
        {
                 state[i] = THINKING;
                test ((i + 4) % 5);
                test ((i + 1) % 5);
        }
        void test (int i)
        {
                 if ((state [(i + 4) % 5]! = EATING) && (state[i] == HUNGRY) && (state [(i
+ 1) % 5]! =                    EATING))
                {
                        state[i] = EATING;
                        self[i].signal ();
                }
        }
        initialization code ()
        {
                 for (int i = 0; i < 5; i++)
                state[i] = THINKING;
        }
```

}

3. Implementing a Monitor Using Semaphores

Variables

The following are the variables,

semaphore mutex=1;

For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait (mutex) before entering the monitor and must execute signal (mutex) after leaving the monitor.

semaphore next=0;

Since a signalling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, The signalling processes can use next to suspend themselves

int next_count = 0;

An integer variable next count is also provided to count the number of processes suspended on next.

Procedure

Each procedure ***F*** will be replaced by,

```
wait (mutex);
...
        body of F
...
if (next _count > 0)
        signal (next);
else
        signal (mutex);
```

Mutual exclusion within a monitor is ensured.

Condition Variables

For each condition x, we introduce a semaphore x _sem and an integer variable x _count, both initialized to 0.

The operation x.wait () can now be implemented as,

```
x_count++;
if (next_count > 0)
        signal (next);
else
        signal (mutex);
wait (x_sem);
x_count--;
```

The operation x.signal () can be implemented as,

```
if (x_count > 0)
 {
```

```
        next_count++;
        signal (x_sem);
        wait (next);
        next_count--;
    }
```

4. Resuming Processes within a Monitor

If several processes are suspended on condition x, and an x.signal () operation is executed by some process, then we have to determine which of the suspended processes should be resumed next.

First-Come, First-Served (FCFS)

One simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. This is a simple scheduling scheme but not adequate.

Conditional-Wait

This construct has the form

```
x.wait(c);
```

where c is an integer expression that is evaluated when the wait () operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process that is
suspended. When x.signal () is executed, the process with the smallest priority number is resumed next.

ResourceAllocator monitor

It controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
        ...
        access the resource;
        ...
R.release();
```

where R is an instance of type ResourceAllocator.

Monitor to allocate a single resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time)
    {
         if (busy)
        x.wait(time);
        busy = true;
    }
```

```
        void release()
        {
                busy = false;
                x.signal();
        }
        initialization code()
        {
                busy = false;
        }
}
```

Problems

The following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

One possible solution is to include the resource access operations within the ResourceAllocator monitor.