



Inter-Process Communication

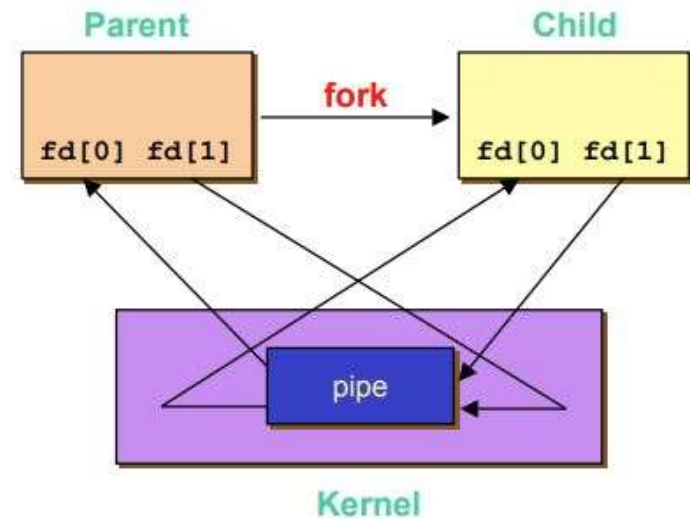
Inter-Process Communication

- Mechanisms for processes to communicate with each other
- UNIX IPC
 - ✓ pipes
 - ✓ FIFOs
 - ✓ message queue
 - ✓ shared memory
 - ✓ sockets

Pipes

- The oldest form of UNIX IPC
 - ✓ Half-duplex
 - ✓ Between processes that have a common ancestor
- Pipe

```
#include <unistd.h>
int pipe(int fd[2]);
return: 0 if OK, -1 on error
```
- Half-duplex pipe after a fork
- IPC through pipes
 - ✓ Once we have created a pipe using pipe
 - ✓ We can use the normal file I/O functions (e.g., read, write)



Exercise

- Send data from parent to child over a pipe

```
$gcc -o pipe pipe.c (or make pipe)  
$./pipe
```

- Synchronization between parent and child using pipe

```
$gcc -o sync sync.c synchlib.c (or make sync)  
$./sync
```

FIFOs

- Named pipes

- ✓ Full-duplex
- ✓ Between unrelated processes that don't have a common ancestor

- Create a FIFO

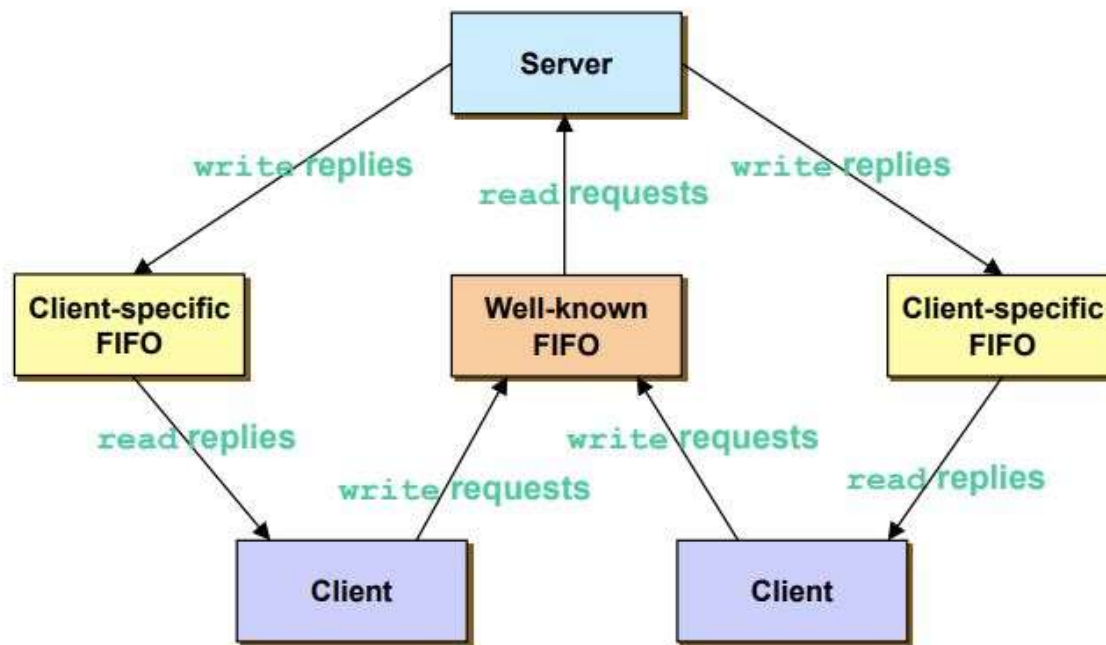
```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(char *pathname, mode_t mode);
return: 0 if OK, -1 on error
```

- IPC through FIFOs

- ✓ Once we have created a FIFO using `mkfifo`,
- ✓ We can use the normal file I/O functions (e.g., `open`, `read`, `write`, `close`)

FIFOs

- Client-server communication using FIFOs



Exercise

- Client-server communication using FIFOs

```
$gcc -o fifos fifos.c (or make fifos)
```

```
$gcc -o fifoc fifoc.c (or make fifoc)
```

```
$./fifos
```

```
$./fifoc
```

Message Queues

- A linked list of messages stored within the kernel
- A message consists of
 - ✓ A long integer that have the positive integer message type
 - ✓ Message data

```
struct mymsg {  
    long mtype;    /* positive message type  
*/  
    char mtext[512]; /* message data */  
};
```

- IPC through message queues
 - ✓ msgget : Open an existing queue or create a new one
 - ✓ msgsnd : Place a message onto the queue
 - ✓ msgrcv : Fetch a message from the queue

System Calls for Message Queues

- Obtain a message queue ID

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

```
int msgget(key_t key, int flag);
return: message queue ID if OK, -1 on error
```

- Message queue control operations

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

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
return: 0 if OK, -1 on error
```

System Calls for Message Queues

- Message queue control operations

- ✓ The second argument, `cmd`

- `IPC_STAT` : fetch the `msqid_ds` structure for this queue

- `IPC_SET` : set the part of `msqid_ds` structure

- `IPC_RMID` : remove the message queue from the system

- ✓ The third argument, `buf`

```
struct msqid_ds {
    struct ipc_perm    msg_perm;           //IPC structure: permission and owner
    struct msg         *msg_first;         //ptr to first message on queue
    struct msg         *msg_last;         //ptr to last message on queue
    ulong              msg_cbytes;         //current # of bytes on queue
    ulong              msg_qnum;           //# of messages on queue
    ulong              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
};
```

System Calls for Message Queues

▪ Send a message

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

```
int msgsnd(int msqid, void *ptr, size_t nbytes, int flag);
return: 0 if OK, -1 on error
the fourth argument, flag: IPC_NOWAIT
```

▪ Receive a message

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

```
int msgrcv(int msqid, void *ptr, size_t nbytes, long type, int
flag);
return: size of data portion of message if OK, -1 on error
the fifth argument, flag: IPC_NOWAIT
```

System Calls for Message Queues

- Receive a message
 - ✓ If `type == 0`,
 - The first message on the queue is returned
 - ✓ If `type > 0`,
 - The first message on the queue whose message type equals `type` is returned
 - ✓ If `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
 - ✓ If `IPC_NOWAIT` is specified in `flag`,
 - Return is made with an error of `EAGAIN`

Exercise

- IPC between two processes using message queue

```
$gcc -o msgq1 msgq1.c (or make msgq1)
$gcc -o msgq2 msgq2.c (or make msgq2)
$./msgq1
```

```
$./msgq2
```

- Note:

- ✓ If a process creates a message queue, its data structure remains in the kernel even though the process has terminated
- ✓ You have to remove it through `msgctl()` with `IPC_RMID` parameter (or, reboot the system !!)

Shared Memory

- Allows two or more processes to share a given region of memory
 - ✓ The fastest form of IPC because data does not need to be copied between them
 - ✓ Need to synchronize shared memory access
 - Semaphores are used often
- IPC through shared memory
 - ✓ shmget
 - Obtain a shared memory identifier
 - Open an existing segment or create a new one
 - ✓ shmat
 - Attach a shared memory segment to the process' address space
 - ✓ shmdt
 - Detach a shared memory segment
 - shmdt does not remove the identifier and its associated data structure from the system

System Calls for Shared Memory

- Obtain a shared memory ID

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int flag);
return: shared memory ID if OK, -1 on error
```

- ✓ The second argument, **size**

- If a new segment is being created, we must specify its minimum size, **size**
- If we are referencing an existing segment, we can specify **size** as 0

- Shared memory control operations

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
return: 0 if OK, -1 on error
```

System Calls for Shared Memory

- Shared memory control operations

- ✓ The second argument, `cmd`

<code>IPC_STAT</code>	:fetch the <code>shmid_ds</code> structure for this shared memory
<code>IPC_SET</code>	:set the part of <code>shmid_ds</code> structure
<code>IPC_RMID</code>	:remove the shared memory segment from the system
<code>SHM_LOCK</code>	:lock the shared memory in memory
<code>SHM_UNLOCK</code>	:unlock the shared memory segment

- ✓ The third argument, `buf`

```
struct shmid_ds {
    struct ipc_perm    shm_perm;           //IPC structure: permission and owner
    struct anon_map    *shm_amp;           //pointer in kernel
    int                shm_segsz;          //size of segment in bytes
    ushort             shm_lkcnt;          //# of times segment is being locked
    pid_t              shm_lpid;           //pid of last shmop()
    pid_t              shm_cpid;           //pid of creator
    ulong              shm_nattch;         //# of current attaches
    ulong              shm_cnattch;        //used only for shminfo
    time_t             shm_atime;          //last-attach time
    time_t             shm_dtime;          //last-detach time
    time_t             shm_ctime;          //last-chang time
};
```

System Calls for Shared Memory

- Attach a shared memory segment

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int *shmat(int shmid, void *addr, int flag);
return: pointer to shared memory segment if OK, -1 on error
```

- ✓ The fourth argument, `flag`
 - `SHM_RND`, `SHM_RDONLY`
- ✓ If `addr == 0`, (Recommended)
 - The segment is attached at the first available address selected by the kernel
- ✓ If `addr != 0` and `SHM_RND` is not specified,
 - The segment is attached at the address given by `addr`
- ✓ If `addr != 0` and `SHM_RND` is specified,
 - The segment is attached at the address given by `(addr - (addr modulus SHMLBA))`
- ✓ If `SHM_RDONLY` is specified,
 - The segment is attached read-only

System Calls for Shared Memory

- Detach a shared memory segment

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(void *addr);
return: 0 if OK, -1 on error
```

Exercise

- Shared memory example (&memory map)

```
$gcc -o shm shm.c (or make shm)  
$./shm
```

- IPC between two processes using shared memory

```
$gcc -o sipc1 sipc1.c (or make sipc1)  
$gcc -o sipc2 sipc2.c (or make sipc2)  
$./sipc1  
  
$./sipc2
```

- Note:

- ✓ If a process creates a shared memory, its data structure remains in the kernel even though the process has terminated
- ✓ You have to remove it through `shmctl()` with `IPC_RMID` parameter (or, reboot the system !!)



Synchronization

Synchronization

- Thread cooperate in multithreaded programs
 - ✓ To **share** resources, access shared data structures
 - ✓ Also, to **coordinate** their execution
- For correctness, we have to control this cooperation
 - ✓ Must assume threads interleave executions arbitrarily and at different rates
 - ✓ We control cooperation using **synchronization**
 - Enables us to restrict the interleaving of execution
 - ✓ (Note) This also applies to processes, not just threads
 - And it also applies across machines in a distributed system

An Example

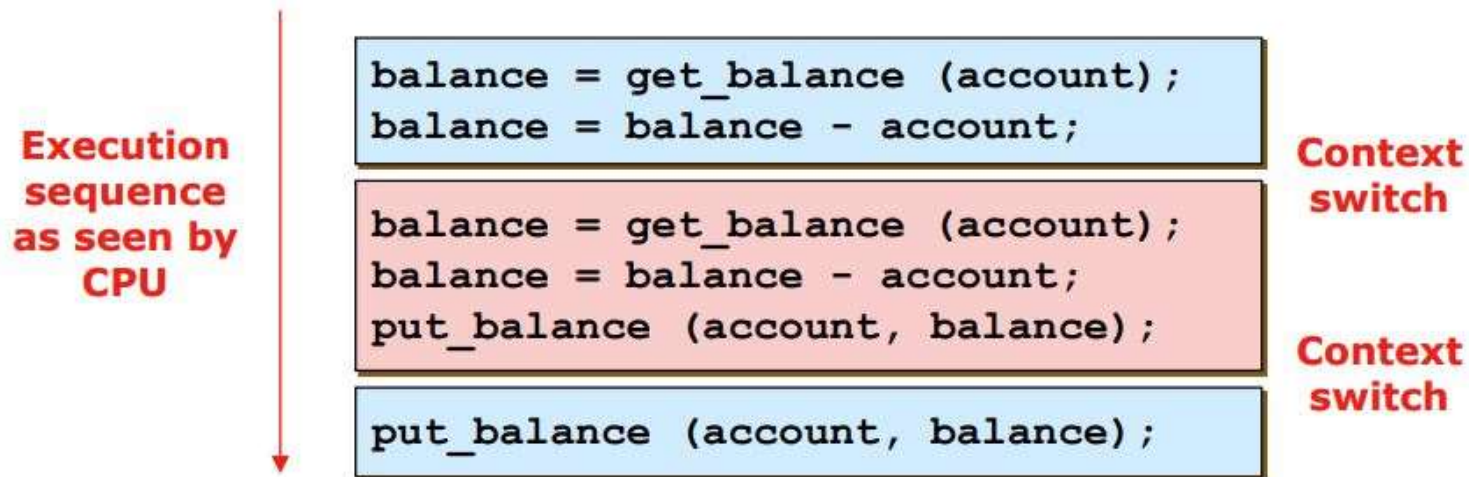
- Withdraw money from a bank account
 - ✓ Suppose you and your friend share a bank account with a balance of 1,000,000 won
 - ✓ What happens if both go to separate ATM machines, and simultaneously withdraw 100,000 won from the account?

```
int withdraw (account, amount)
{
    balance = get_balance (account);
    balance = balance - amount;
    put_balance (account, balance);
    return balance;
}
```

An Example

- Interleaved schedules

- ✓ Represent the situation by creating a separate thread for each person to do the withdrawals
- ✓ The execution of the two threads can be interleaved, assuming preemptive scheduling:

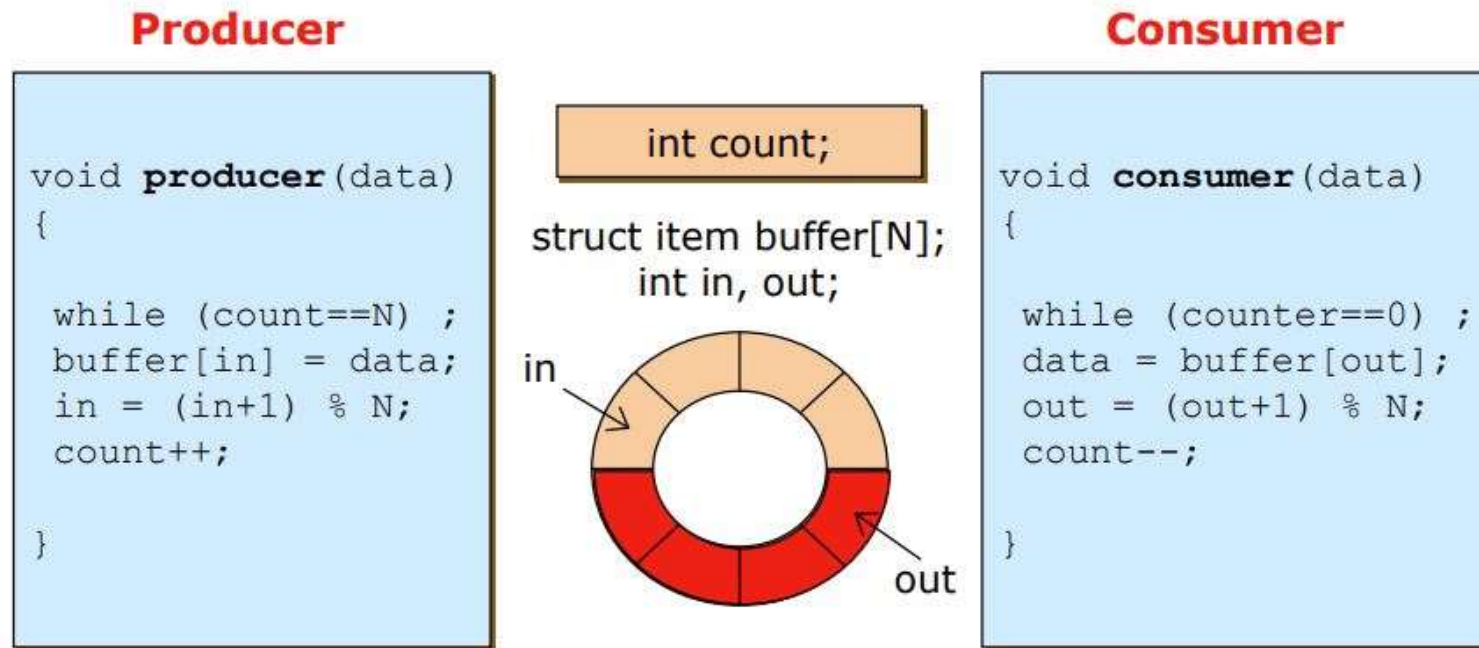


Synchronization Problem

- Two concurrent threads (or processes) access a **shared resource** without any **synchronization**
- Creates a race condition
 - ✓ The situation where several processes access and manipulate shared data concurrently
 - ✓ The result is non-deterministic and depends on timing
- **Critical section**
 - ✓ Code segment in which the shared data is accessed
- We need mechanisms for controlling access to critical sections in the face of concurrency
 - ✓ So that we can reason about the operation programs
- Synchronization is necessary for any shared data structure
 - ✓ Buffers, queues, lists, etc.

Another Example: Bounded Buffer

- No synchronization



Exercise

- Producer & Consumer sharing bounded buffer

```
$ gcc -o producer producer.c (or make producer)
$ gcc -o consumer consumer.c (or make consumer)
$ ./producer
$ ./consumer
```

- Synchronization problem

- ✓ The statement “count++” may be implemented in machine language as:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- ✓ The statement “count--” may be implemented as:

```
register2 = counter
register2 = register2 + 1
counter = register2
```

Exercise

- Synchronization problem (cont'd)

- ✓ Assume `counter` is initially 5. One interleaving of statements is:

```
producer: register1 = counter      (register1 = 5)
producer: register1 = register1 + 1 (register1 = 6)
consumer: register2 = counter      (register2 = 5)
consumer: register2 = register2 - 1 (register2 = 4)
producer: counter = register1      (counter = 6)
consumer: counter = register2      (counter = 4)
```

- ✓ The value of `counter` may be either 4 or 6, where the correct result should be 5

Synchronization Mechanisms

- Semaphores
 - ✓ Basic, easy to get the hang of, hard to program with
 - ✓ Binary semaphore = mutex (lock)
 - ✓ Counting semaphore

- Monitors
 - ✓ High-level, requires language support, implicit operations
 - ✓ Each to program with: Java “synchronized”

- Mutex + Condition variables
 - ✓ Pthreads

Semaphores

- Semaphore

- ✓ A counter used to provide access to a shared data object for multiple processes or threads
- ✓ Two operations
 - Wait or P
 - Signal or V

- Synchronization procedure using semaphores

- ✓ Test the semaphore that controls the resource
- ✓ If the value of the semaphore is positive, the process can use the resource
 - The process decrements the semaphore value by 1, indicating that it has used one unit of the resource
- ✓ If the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0
 - When the process wakes up, it returns to above step

Semaphore Implementations

- System V semaphore
 - ✓ Named semaphore → between processes
 - ✓ Shared key (number) between processes
 - ✓ Serviced by kernel
- POSIX semaphore
 - ✓ Unnamed semaphore → between threads or related processes
 - ✓ Shared variable in `sem_t` type between threads or related processes
 - ✓ Serviced by libraries or kernel
 - ✓ Most implementation doesn't support synchronization between processes yet, including Solaris and Linux

System calls for System V Semaphores

- Obtain a semaphore set ID

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

```
int semget(key_t key, int nsems, int flag);
return: semaphore ID if OK, -1 on error
```

- Semaphore control operations

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

```
int semctl(int semid, int semnum, int cmd, union semun arg);
return: non-negative value depending on cmd if OK, -1 on error
```

System calls for System V Semaphores

■ Semaphore control operations

✓ The third argument, cmd

```
IPC_STAT: fetch the semid_ds structure for this semaphore set
IPC_SET  : set the part of semid_ds structure
IPC_RMID: remove the semaphore set from the system
GETVAL   : return the semaphore value
SETVAL   : set the semaphore value
GETPID   : get pid of the process which do the last access to the semaphore
GETNCNT  : return the number of processes which wait for the semaphore to increase
GETZCNT  : return the number of processes which wait for the semaphore to be zero
GETALL   : fetch all the semaphore values in the set
SETALL   : set all the semaphore values in the set
```

✓ The fourth argument, arg

```
union semun {
    int          val;          /* for SETVAL */
    struct semid_ds *buf;      /* for IPC_STAT and IPC_SET */
    ushort       *array;       /* for GETALL and SETALL */
};
```

System calls for System V Semaphores

- Semaphore operations

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf semop[], size_t nops);
return: 0 if OK, -1 on error
```

The second argument, semop

```
struct sembuf {
    ushort sem_num; // member # in set (0, 1, ..., nsems-1)
    short  sem_op;  // operation (negative, 0, or positive)
    short  sem_flg; // IPC_NOWAIT, SEM_UNDO
};
```

System calls for System V Semaphores

■ Semaphore operations

- ✓ If `sem_op > 0`,
 - The value of `sem_op` is added to the semaphore's value
- ✓ If `sem_op < 0` and the semaphore's value $\geq$ the value of `sem_op`,
 - The value of `sem_op` is added to the semaphore's value
- ✓ If `sem_op < 0` and the semaphore's value $<$ the value of `sem_op`,
 - If `IPC_NOWAIT` is not specified, the calling process is suspended until the semaphore's value $\geq$ the absolute value of `sem_op`
 - If `IPC_NOWAIT` is specified, return is made with an error of `EAGAIN`
- ✓ If `sem_op == 0` and the semaphore's value is not zero
 - If `IPC_NOWAIT` is not specified, the calling process is suspended until the semaphore's value becomes zero
 - If `IPC_NOWAIT` is specified, return is made with an error of `EAGAIN`

■ Semaphore adjustment on `exit`

- ✓ What happens if a process terminates while it has resources allocated through a semaphore?
- ✓ Use `SEM_UNDO` flags

POSIX Semaphores

- POSIX semaphore libraries

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
int sem_destroy(sem_t *sem);
return: 0 if OK, non-zero value on error
```

- Note: link option (dependent on semaphore packages)

- ✓ Solaris

- lposix4

- ✓ Linux

- lpthread

Exercise

- Implementation of semaphores similar to POSIX semaphores using System V semaphores & shared memory

```
$gcc semlib.c (or make semlib.o)
```

- Producer & Consumer example using `semLib` library

```
$gcc -o producer_s producer_s.c semlib.c (or make producer_s)
```

```
$gcc -o consumer_s consumer_s.c semlib.c (or make consumer_s)
```

```
$./producer_s
```

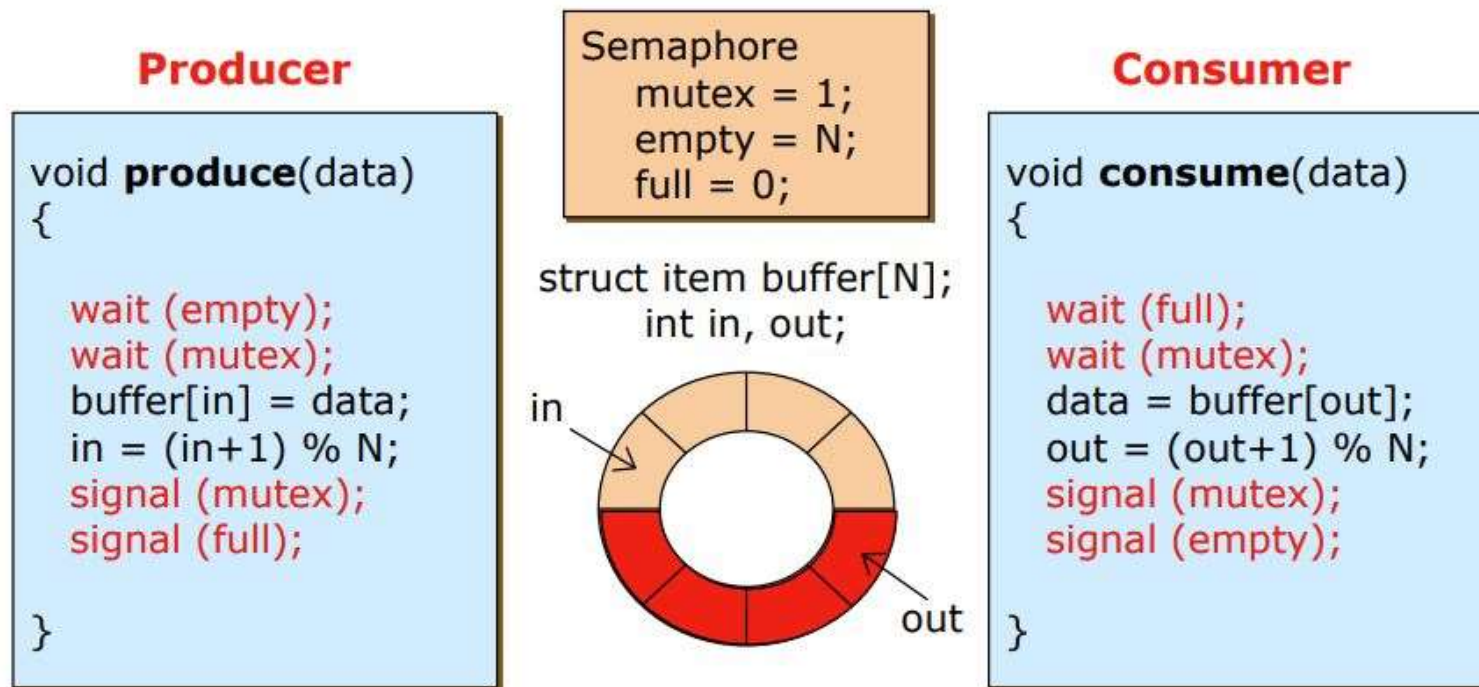
```
$./consumer_s
```

- Note:

- ✓ If a process creates a system V semaphore, its data structure remains in the kernel **even though the process has terminated**
- ✓ You have to remove it through `semctl()` with `IPC_RMID` parameter (or, reboot the system !!!)

Exercise

- Bounded buffer implementation with semaphores



Exercise

- Producer & consumer example using pthreads & POSIX semaphores

```
$gcc -o prodcons prodcons.c -lposix4 -lpthread (or make prodcons)  
$./prodcons
```

Mutexes

- Mutexes

- ✓ Mutual exclusive locks for threads
- ✓ Serviced by Pthread libraries
- ✓ Similar to binary semaphore

- Pthread libraries for mutexes

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *mattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
return: 0 if OK, non-zero value on error
```

Condition Variables

- Condition variables

- ✓ A synchronization device that allows threads to suspend and resume execution until some predicate on shared data is satisfied
- ✓ Serviced by Pthread libraries
- ✓ Associated with a mutex, to avoid the race condition

- Pthread libraries for condition variables

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cattr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
return: 0 if OK, non-zero value on error
```

Exercise

- Producer & Consumer example using mutexes and condition variables

```
$gcc -o prodcons_m prodcons_m.c -lpthread (or make prodcons_m)  
$./prodcons_m
```

- Implementation of POSIX semaphores using mutexes and condition variables

```
$gcc semlib2.c (or make semlib2.o)
```

- Producer & Consumer example using `semlib2` library

```
$gcc -o prodcons_s prodcons_s.c semlib2.c -lpthread (or make prodcons_s)  
$./prodcons_s
```



END