Lecture 5: IPC—Message Queues, Semaphore and Shared Memory

References for Lecture 5:

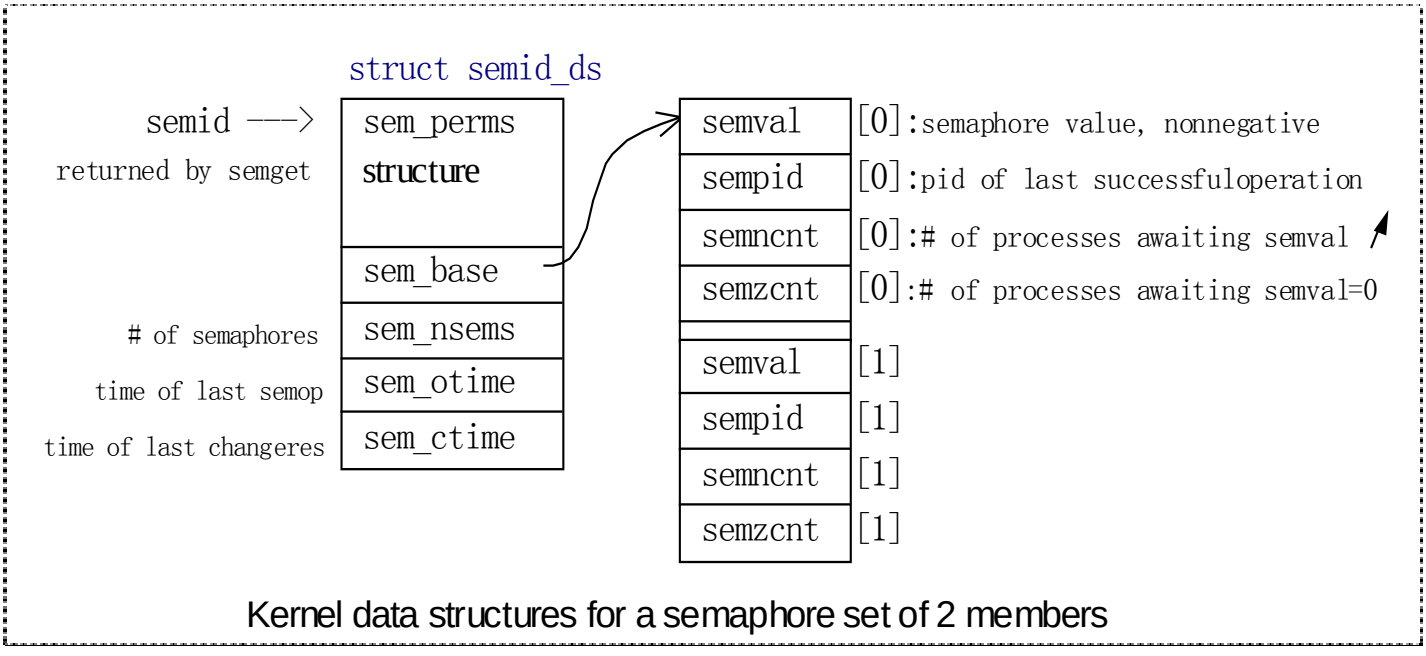
1) Unix Network Programming, W.R. Stevens, 1990,Prentice-Hall, Chapter 3.
2) Unix Network Programming, W.R. Stevens, 1999,Prentice-Hall, Chapter 2-6.

Semaphore

Semaphores are not used to exchange a large amount of data. Semaphores are used synchronization among processes. Other synchronization mechanisms include record locking and mutexes. Why necessary? Examples include: shared washroom, common rail segment, and common bank account.

Further comments:

1) The semaphore is stored in the kernel:
   - Allows atomic operations on the semaphore.
   - Processes are prevented from indirectly modifying the value.
2) A process acquires the semaphore if it has a value of zero. The value of the semaphore is then incremented to 1. When a process releases the semaphore, the value of the semaphore is decremented.
3) If the semaphore has non-zero value when a process tries to acquire it, that process blocks.
4) In comments 2 and 3, the semaphore acts as a customer counter. In most cases, it is a resource counter.
5) When a process waits for a semaphore, the kernel puts the process “to sleep” until the semaphore is available. This is better (more efficient) than busy waiting such as TEST&SET.
6) The kernel maintains information on each semaphore internally, using a data structure **struct semis_ds** that keeps track of permission, number of semaphores, etc.
7) Apparently, a semaphore in Unix is not a single binary value, but a set of nonnegative integer values



Kernel data structures for a semaphore set of 2 members

8) There are 3 (logical) types of semaphores:
   - Binary semaphore – have a value of 0 or 1. Similar to a mutex lock. 0 means locked; 1 means unlocked.
   - Counting semaphore – has a value ⩾ 0. Used for counting resources, like the producer-consumer example. Note that value =0 is similar to a lock (resource not available).
   - Set of counting semaphores – one or more semaphores, each of which is a counting semaphore.
9) There are 2 basic operations performed with semaphores:
   - Wait – waits until the semaphore is > 0, then decrements it.
   - Post – increments the semaphore, which wakes waiting processes.

Suppose you know:
concurrent process, critical region, shared resource, deadlock, mutual exclusion, primitive, atomic operation.

A semaphore set is created using

int semget(key_t *key*, int *nsems*, int *semflag*); -- returns int *semid*; the id of the semaphore set; -1 on error.

nsems — # of semaphores, use multiple semaphores for multiple resources.

semflag — Same as *msgflag*, sets permission and creation options. See Lecture 4's Comments 1.

Operations on a semaphore are performed using:

int semop(int *semid*, struct sembuf **opsptr*, unsigned int *nops*)

semid — value returned by semget.

nops — # of operations to perform, or the number of elements in the *opsptr* array.

opsptr — points to an array of one or more operations. Each operation is defined as:

```
struct sembuf { ushort  sem_num;   /* semaphore #, numbered from 0, 1, 2 … */
                short   sem_op;    /* semaphore operation */
                short   sem_flg;   /*operations flags, such as 0, IPC_NOWAIT for nonblocking call,
                                     or SEM_UNDO to have the semaphore automatically released when the
                                     process is terminated prematurely.*/
              };
```

sem_op = 0 – wait until the semaphore is 0. IPC_NOWAIT causes an error if semval≠0.

sem_op > 0 – increment the semaphore value: semval + sem_op, (acquire)

sem_op < 0 – wait until the semaphore value⩾|sem_op| and
decrement the semaphore value: semval - |sem_op|, (release)

More notes:

- As a customer counter, a semaphore is acquired doing the first two operations in one call; a semaphore is released using the third operation. See the following program example.
- Blocking calls end when the request is satisfied, the semaphore set is deleted, or a signal is received.
- **Keep in mind: all operations in one semop( ) must be finished atomically by the kernel. Either all or none of operations will be done.**

Control operations are performed using:

int semctl(int *semid*, int *semnum*, int *cmd*, union semun *arg*); — Return value depends on *cmd*, -1 on error.

```
union semun { int          val;    /* used for SETVAL only */
              stuct semid_ds *buff   /* used for IPC_STAT and IPC_SET */
              ushort         *array  /* used for GETALL and SETALL*/
              } arg;
```

Which field is used in the union depends on the *cmd.*

cmd --- IPC_RMID to remove a semaphore set. Union semun *arg* is not used in this case.

GETVAL /SETVAL to fetch/set a specific value. *semnum* can specify a member of the semaphore set.

GETALL/SETALL to fetch/set all values of the semaphore set.

A semaphore set with one semaphore would be initialized using:

```
union semun arg;
arg.val =1;
semctl(semid, 0, SETVAL, arg);
```

Example: How to write lock/unlock (somewhat like P/V operations)

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#define     SEMKEY   123456L   /* key value for semget() */
#define     PERMS     0666

static struct sembuf op_lock[2]= { 0, 0, 0,                  /* wait for sem #0 to become 0 */
                                   0, 1, SEM_UNDO   /* then increment sem #0 by 1 */ };

static struct sembuf op_unlock[1]= { 0, -1, (IPC_NOWAIT | SEM_UNDO)
                                    /* decrement sem #0 by 1 (sets it to 0) */ };

int semid = -1;   /* semaphore id. Only the first time will create a semaphore.*/

my_lock( )
{    if   (semid <0) {
            if ( ( semid=semget(SEMKEY, 1, IPC_CREAT | PERMS )) < 0 ) printf("semget error"); }

     if (semop(semid, &op_lock[0], 2) < 0)     printf("semop lock error");
}

my_unlock( )
{
     if (semop(semid, &op_unlock[0], 1) < 0)     printf("semop unlock error");
}
```

Questions: how to rewrite the above program to make the semaphore as a resource counter? What if the resource allows 3 or more processes to use at the same time?

Solutions:
static struct sembuf op_lock[1]= { 0, -1, SEM_UNDO, };
static struct sembuf op_unlock[1]= { 0, 1, (IPC_NOWAIT | SEM_UNDO) }

Don't forget to set the initial value of the semaphore as 1 or 3.

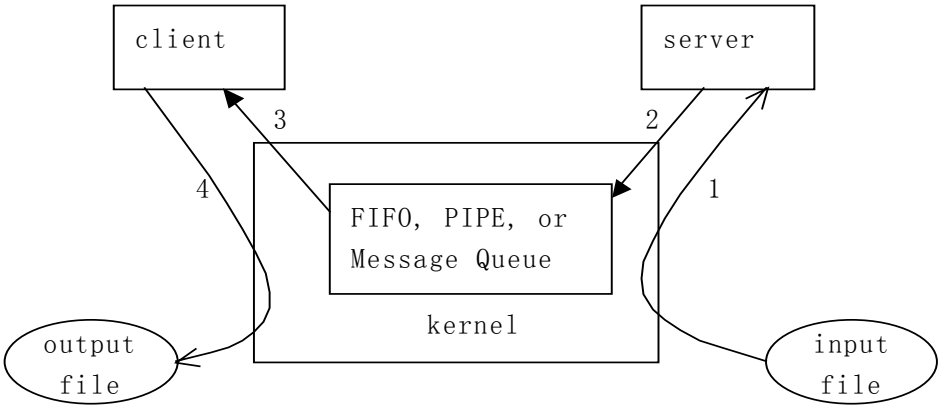
Homework:

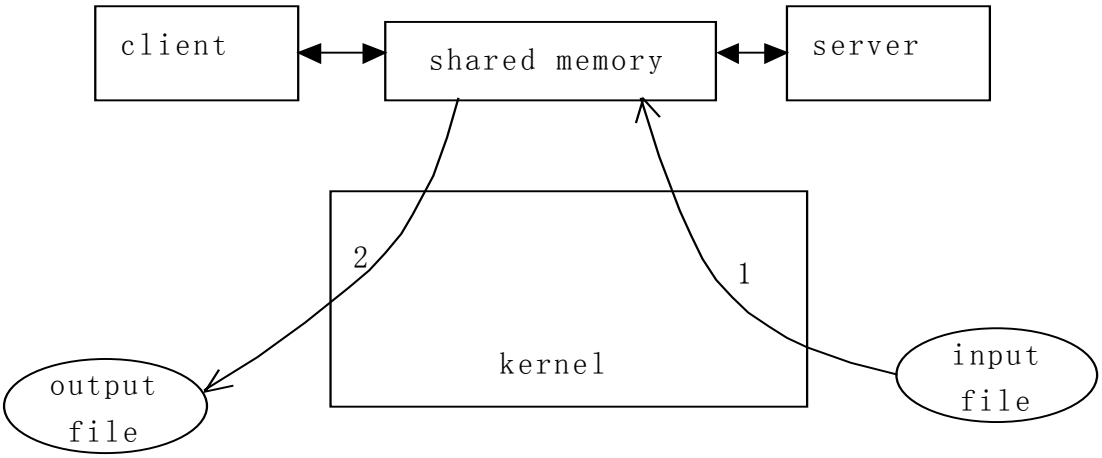
1) Open as many windows as you like. Run your program on each window simultaneously. Your program will iteratively read a common file that contains an integer counter, increment the counter value, print the new value, and write back the new value into the file. Please use semaphore in order to avoid the same value to be generated by 2 different processes. Delete the semaphore part in your program to see the different result.
2) Use only one pipe combined with semaphore to implement bi-directional communication.

One step further : the set of initial value or the semid in the above example is a kind of race condition which in itself need to be synchronized.

Shared Memory

Using a pipe or a message queue requires multiple exchanges of data through the kernel. Shared memory can be used to bypass the kernel for faster processing.





The kernel maintains information about each shared memory segment, including permission, size, access time, etc in **struct shmid_ds** .

struct shmid_ds looks like **struct semid_ds** or as **struct msqid_ds** .

A share memory segment is created using:
int shmget(key_t *key*, int *size*, int *shmflag*);
size --- size of the shared memory segment in bytes.
shmflag --- same as for msgget() and semget(), see Lecture 4's Comments 1.
Return value --- *shmid*, the shared memory identifier, -1 on error.

Attach to the shared memory segment using:
char *shmat(int *shmid*, char **shmaddr*, int *shmflag*)
shmid --- return value of shmget, that is, the id of the created shared memory.
shmaddr--- 0: let the kernel select the address.
shmflag--- SHM_RDONLY for read_only access.
returns the starting address of the shared memory, and thus we can read/write on the shared memory after getting its starting address.

Detach the shared memory segment using:
char *shmdt(char **shmaddr)*
shmaddr --- the return value of shmat(), that is, the starting address of the shared memory.
returns –1 on failure.

To remove a shared memory segment:
Int shmctl(int *shmid*, int *cmd*, shmid_ds **buf*);
cmd--- IPC_RMID to delete, e.g., shmctl(shmid, IPC_RMID, 0) .

Homework: How to use the shared memory to implement C/S example, don't forget to use semaphores to synchronize the client and the server.