Overview

- Process creation and termination.
- Semaphore calls: creation, initialization, wait and signal, and the system calls used to implement these calls.
- Shared memory allocation and attachment.

<u>Process Creation</u>

- why would you create multiple processes?
- how would you keep track of these processes?

int fork ( );

- called by an existing process – the **parent**
- creates a duplicate – the **child** - of the process that called fork
- fork() returns the process ID of the new child process to the parent
- fork() returns 0 to the child if the process was successful, -1 otherwise

<u>Process suspension</u>

pid_t wait (int *status)

- suspends execution of a current process until the child has exited, this is useful when we look at semaphores since children should exit before the parent to make sure that only the parent releases the semaphore.
- this will return –1 if an error occurs

Semaphores

- **definition :**

Allows multiple processes to access critical sections, by use of wait() and signal() operations. This is relevant to shared memory.

```
wait (s)          s.val = s.val – 1
                  if s.val < 0 then
                          block and add process to tail of s.list
                  end

signal (s)        s.val = s.val + 1
                  if s.val <= 0 then
                          remove process from front of s.list and add to RTR
                          queue
                  end
```

- **usage :**

To use a semaphore you must first create it (semget(…)), and then initialize it (semclt(…)). At this point you can perform operations on the semaphore to control access to the critical section.

Semaphore Creation

```
int semget(key_t, int nsems, int semflag);
```

- this function is used to obtain the semaphore ID
- the value returned is the semaphore identifier **sem_id** or –1 in case of error

parameters

key_t: always set to IPC_PRIVATE (defined in sys/ipc.h), this guarantees that a brand new semaphore is created.

nsems: number of semaphores in the set- for the assignment this will likely be set to 1

semflag: defines access rights to the semaphore, set to SEM_MODE

Semaphore Initialization

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

```
union semun {
                int val;
                struct semid_ds *buf;
                ushort * array;
                } arg ;
```

- This system call is normally used for initializing or removing the semaphore set

parameters

semid: The semid argument is the value returned from the semget() call

semnum: Specifies one of the semaphores in the set, usually set to 0 , since only have one semaphore in the set.

cmd: Specifies the command to be performed, usually SETVAL or IPC_RMID

arg: the initialization value of the semaphore is taken from the arg’s val field

To **initialize** a single semaphore to value sem_init:

```
semctl(semID_1, 0, SETVAL, sem_init);
```

To **remove** a semaphore:

```
semctl(semID_1, 0, IPC_RMID, 0);
```

Where sem_init is a data structure with a field that contains the initialization value for the semaphore.

<u>Semaphore Operations</u>

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

```
struct sembuf          {
        ushort sem_num;
        short sem_op;
        short sem_flg;
                         }
```

- Operations are performed on one or more semaphores in the set using the semop call. Basically this operation adds or subtracts a value (specified by sem_op in the sembuf struct) to the current value of the semaphore.
- To call the function will pass in the address of the struct: semop (semid, &sstruct, 1);

parameters

nsops: Specifies the number of operations to be performed on the semaphore.

*** sops:** Pointer to an array where each element is of type sembuf (the struct defined above). This means that each element in the array defines an operation for one specific semaphore in the set.

defining sembuf’s fields:

sem_num: This field specifies the particular semaphore in the set that the current operation will be applied to.

sem_op: Determines the operation to be performed on the semaphore.

If **sem_op > 0,** sem_op gets added to the current value of the semaphore- this corresponds to releasing semaphore resources.

If **sem_op = 0,** the caller waits until the semaphore's value becomes 0.

If **sem_op < 0,** the caller waits until the semaphore's value becomes >= the absolute value of sem_op. This corresponds to the allocation of resources. Then the semaphore's value becomes:

semaphore's current value – abs(sem_op)

sem_flg: Sets options for the semop( ) call.

If **sem_flg = IPC_NOWAIT,** the caller does not get blocked if the operation can not be completed.

If **sem_flg = IPC_WAIT,** the caller waits.

If **sem_flg = SEM_UNDO**, the kernel remembers how many resources are allocated to that particular semaphore and when the process terminates the kernel releases all allocated resources.

4 generic semaphore functions:

```
int SemaphoreWait(int semid, int iMayBlock ) {
        struct sembuf           sbOperation;

        sbOperation.sem_num = 0;
        sbOperation.sem_op = -1;
        sbOperation.sem_flg = iMayBlock;

        return semop( semid, &sbOperation, 1 );
}


int SemaphoreSignal( int semid ) {
        struct sembuf           sbOperation;
        sbOperation.sem_num = 0;
        sbOperation.sem_op = +1;
        sbOperation.sem_flg = 0;
        return semop( semid, &sbOperation, 1 );
}

void SemaphoreRemove( int semid ) {
        if(semid != IPC_ERROR )
                semctl( semid, 0, IPC_RMID , 0);
}

int SemaphoreCreate(int iInitialValue) {
        int semid;
        union semun suInitData;
        int iError;

        /* get a semaphore */
        semid = semget( IPC_PRIVATE, 1, SEM_MODE );

        /* check for errors */
        if( semid == IPC_ERROR )
                return semid;

        /* now initialize the semaphore */
        suInitData.val = iInitialValue;

        if(semctl( semid, 0, SETVAL, suInitData) == IPC_ERROR )
        { /* error occurred, so remove semaphore */
                SemaphoreRemove(semid);
                return IPC_ERROR
        }

return semid;
}
```

Some syntax examples using semaphore operations:

```
  // Create a semaphore:
  int s_mutex;
  if  (( s_mutex = SemaphoreCreate(1)) == IPC_ERROR) {
        SemaphoreRemove(s_mutex);
        perror("Error in SemaphoreCreate");
        exit(1);
  }
```

```
// Use the semaphore to access a critical section:

SemaphoreWait(s_mutex, SEMOP_BLOCKING);
// access the critical section here.
SemaphoreSignal(s_mutex);
```

Shared Memory

- Provides a way for 2 or more processes to share a memory segment.
- When multiple processes access the same memory segment they have to coordinate this access among themselves.
- This is where the semaphores described in the last section are used.

<u>Shared Memory Creation</u>

```
int shmget ( key_t key, int size, int shmflag);
```

- Creates a shared memory segment.
- The value returned by shmget(…) is the shared memory identifier , shmid, or –1 if an error occurs.

parameters

key: Always set to IPC_PRIVATE (defined in sys/ipc.h), this guarantees a new segment of memory.

size: Specifies the size of the shared memory required in bytes.

shmflg: Determines access rights to the shared memory, should be set to SHM_MODE.

<u>Shared Memory Attachment</u>

- After the shared memory is created it must be attached to an address space.
- The system call returns the starting address of the shared memory segment, in the form of a pointer.

```
void *shmat (int shmid, char *shmaddr, int shmflag)
```

parameters

shmid: Shared memory identifier obtained through the shmget( ) system call.

shmaddr: Usually set to 0, this means the memory segment is attached at the first available address selected by the kernel.

shmflag: Set to 0 for read and write access.

A shared memory syntax example:

```
// get shared memory:

int shmid1;

typedef struct{
// declare an array of a certain size
} buffer1;

int shm_size1 = sizeof(buffer1);

if ((shmid_1 = shmget(shm_key1, shm_size1, SHM_MODE)) <=0) {
        perror( “Error in shmget”);
}
```

Now what about attaching the shared memory?

Shared Memory Example:

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

main( )  {
        int childpid;
        int shm_id;
        int status;
        int *shared_int;
        int inshared_int;

        if ( ( shm_id = (shmget  (IPC_PRIVATE, 1, SHM_MODE)
                perror( “shmget error”);

        if ( ( shared_int = (int * ) shmat (shm_id, (char *)0, 0 )) == (int *)-1)
```

```
                perrr(“shmat”);

        *shared_int = 7;
        unshared_int =7;

 if (( childpid = fork( )) == -1) {
        perror (“can’t fork”);
        exit(1);
 } else if (childpid == 0)      {

        printf(“child: child pid + %d, parent pid = %d\n”, getpid( ), getppid( ));
        *shared_int = 10;
        unshared_int = 10;
        exit(0);
 }
 else   {

 /* parent process */

 if (wait(& status) == -1)
        perror(“wait error”);

 printf(“parent: child pid = %d, parent pid = %d\n”, childpid, getpid( ));
 printf(“the value of the shared int is %d\n”, *shared_int);
 printf(“the value of the unshared int is %d\n”, unshared_int);

 if (shmctl(shm_id, IPC_RMID, (struct shmid_ds *) 0) < 0)
        perror(“can’t IPC_RMID shared”);
 exit(0);
 }
}

/* sample output
child: child pid = 647, parent pid = 646
parent: child pid = 647, parent pid = 646
the value of the shared int is 10
the value of the unshared int is 7
*/
```