Experiment #3
Multi-Processing Environment under Unix/Linux
Process Management for Real-Time Applications

1. Prerequisite

ENCS 538, C programming language, basics of inter-process communications under Unix/Linux.

2. Objectives

- To learn how to create and manage processes on a multi-tasking environment.
- To learn how to establish inter-task communication between the different tasks.
- To learn and implement different control algorithms in embedded systems using software and programming in a multi-tasking environment.

3. Background

The realization and control of embedded system needs some tools to manage multitasks and concurrent processes. This experiment aims to introduce some of these concepts using Linux as multi-tasking environment.

3.1 Semaphore

A semaphore isn't a form of IPC similar to the others that we've described (pipes, FIFOs, and message queues). A semaphore is a counter used to provide access to a shared data object for multiple processes.

The Single UNIX Specification includes an alternate set of semaphore interfaces in the semaphore option of its real-time extensions. We do not discuss these interfaces in this text.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.

2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.

3. Otherwise, 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 step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel.

A common form of semaphore is called a *binary* semaphore. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

The kernel maintains a *semid_ds* structure for each semaphore set:

```
struct semid_ds {
  struct ipc_perm  sem_perm;
  unsigned short   sem_nsems; /* # of semaphores in set */
  time_t           sem_otime; /* last-semop() time */
  time_t           sem_ctime; /* last-change time */
  .
  .
  .
};
```

The Single UNIX Specification defines the fields shown, but implementations can define additional members in the *semid_ds* structure.

Each semaphore is represented by an anonymous structure containing at least the following members:

```
struct {
  unsigned short  semval;   /* semaphore value, always >= 0 */
  pid_t           sempid;   /* pid for last operation */
  unsigned short  semncnt;  /*# processes awaiting semval>curval*/
  unsigned short  semzcnt;  /* # processes awaiting semval==0 */
  .
  .
  .
};
```

The following header summarizes how the semaphore can be used:

Header file name	**`#include <semaphore.h>`**
Semaphore data type	**`sem_t`**
Initialization	**`int sem_init(sem_t *sem, int pshared, unsigned value);`**

Semaphore Operations	**`int sem_destroy(sem_t *sem);`**
	`int sem_wait(sem_t *sem);`
	`int sem_post(sem_t *sem);`
	`int sem_trywait(sem_t *sem);`

sem_init function initializes the semaphore to have the value **value**. The **value** parameter cannot be negative. If the value of **pshared** is not **0**, the semaphore can be used between processes (i.e. the process that initializes it and by children of that process). Otherwise it can be used only by threads within the process that initializes it.

sem_wait is a standard semaphore wait operation. If the semaphore value is 0, the **sem_wait** blocks unit it can successfully decrement the semaphore value.

sem_trywait is similar to **sem_wait** except that instead of blocking when attempting to decrement a zero-valued semaphore, it returns **-1**.

sem_post is a standard semaphore signal operation. The POSIX.1b standard requires that **sem_post** be reentrant with respect to signals, that is, it is asynchronous-signal safe and may be invoked from a signal-handler.

3.2 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 Single UNIX Specification includes an alternate set of interfaces to access shared memory in the shared memory objects option of its real-time extensions. We do not cover the real-time extensions in this text.

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

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* see Section 15.6.2 */
  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 */
  .
  .
  .
};
```

The type **shmatt_t** is defined to be an unsigned integer at least as large as an **unsigned short**.

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

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

int shmget(key_t key, size_t size, int flag);
```

Returns: shared memory ID if OK, 1 on error

The *size* parameter is the size of the shared memory segment in bytes. Implementations will usually round up the size to a multiple of the system's page size, but if an application specifies *size* as a value other than an integral multiple of the system's page size, the remainder of the last page will be unavailable for use.

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

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Returns: 0 if OK, 1 on error

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**. This command can be executed only by a process whose effective user ID equals **shm_perm.cuid** or **shm_perm.uid** or by a process with superuser privileges.

IPC_RMID
Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the **shm_nattch** field in the **shmid_ds** structure), the segment is not removed until the last process using the segment terminates or detaches it. Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that **shmat** can no longer attach the segment. This command can be executed only by a process whose effective user ID equals **shm_perm.cuid** or **shm_perm.uid** or by a process with superuser privileges.

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

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

void *shmat(int shmid, const void *addr, int flag);
```

Returns: pointer to shared memory segment if OK, 1 on error

If the **SHM_RDONLY** bit is specified in *flag*, the segment is attached read-only. Otherwise, the segment is attached readwrite.

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.

When we're done with a shared memory segment, we call **shmdt** to detach it. Note that this does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process specifically removes it by calling **shmctl** with a command of **IPC_RMID**.

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

int shmdt(void *addr);
```

Returns: 0 if OK, 1 on error

4. Procedure

Inter-process communication

1. Execute a man **ipcs** and discover what options can be used with it. Special attention should be given to the **-m, -s** and **-q** options that print information about the active shared memory segments, active semaphores and active message queues respectively.

2. Execute the command **ipcs** and notice how many message queues, semaphores and shared memory segments are currently in use. Notice the different columns that are printed on the screen.

3. To remove a message queue, semaphore or shared memory segment, use the command **ipcrm** with the options **-q, -s** and **-m** respectively. Execute man **ipcrm** and discover the other options.

Semaphores

1. Execute **man semget**, **man semctl**, **man semop**, **man ftok** and check what these functions do.
2. Type the following program that reads data from a user input and writes it to the file **input.txt**. Name the file write **sem.c**. Note how writing to the file is protected using a semaphore so as to inhibit the access to it if that file is being accessed by another process:

```
/*
  * Writing  in  a  file  protected  by  a  semaphore
  */

#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>
#include  <unistd.h>

/*  This  declaration  is *MISSING*  is many  solaris  environments.
      It should  be  in  the  <sys/sem.h>  file  but  often  is not!  If you  receive  a
      duplicate  definition  error  message  for  semun then  comment  out  the
      union  declaration.
      */

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

struct  sembuf  acquire  = {0,  -1,  SEM_UNDO},
                       release  = {0,         1,
                       SEM_UNDO};
#define  SEM_FILE    0
main(void)
{
    int                          sem1;
    key_t                        ipc_key;
    char                         string[128]; static
    ushort                       initSem[1]  = {1};
    union  semun                 arg;
    FILE                        *pFile;

    ipc_key  = ftok(".",  'W');

    /*
      *  Access  the  semaphore  set.
      */
    if ( (sem1  = semget((int)  ipc_key,  1,
            IPC_CREAT  | 0666))  !=  -1  ) {

            arg.array  = initSem;

        if ( semctl(sem1,  0,  SETALL,  arg)  == -1  ) { perror("semctl  --
            write_sem  --  initialization"); exit(1);
  }
    }
    else  {
        perror("semget  --  write_sem  --  access file");
    exit(1);
}

while  ( 1  ) {
    printf("New  string:  ");
    scanf("%s",  &string[0]);

    acquire.sem_num  = SEM_FILE;

    /*
      * Acquiring  the  semaphore  set  before  writing  to  the  file.
      */
    if ( semop(sem1,  &acquire,  1)  == -1  ) {
        perror("semop  --  write_sem  --  acquire  semaphore");
        exit(1);
    }

    if ( (pFile  = fopen("./input.txt",  "r"))  !=  NULL  ) {
        fclose(pFile);

        /*
          * Opening  the  file  for  appending.
          */
        if ( (pFile  = fopen("./input.txt",  "a"))  == NULL  ) { perror("fopen  --
                write_sem  --  append  to  file"); exit(2);
        }
    }
         else  {
            /*
          * Opening  the  file  for  writing.
          */
        if ( (pFile  = fopen("./input.txt",  "w"))  == NULL  ) { perror("fopen  --
                write_sem  --  write  to  file"); exit(3);
        }
    }

    fprintf(pFile,  "%s\n",  string);
    fclose(pFile);

    /*
      * Releasing  the  semaphore  set  after  writing  to  the  file.
      */
          release.sem_num  = SEM_FILE;

    if ( semop(sem1,  &release,  1)  == -1  ) {
        perror("semop  --  write_sem  --  release  semaphore");
        exit(4);
    }
}
}
```

3. Type the following program that reads data from the file input.txt, displays that data to the screen and erases the file. Name the file read **sem.c**. As mentioned above, note how the access to the file is protected using a semaphore

```
/*
 * Reading from a file protected by a semaphore
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>

/* This declaration is *MISSING* is many solaris environments.
    It should be in the <sys/sem.h> file but often is not! If you receive a
    duplicate definition error message for semun then comment out the
    union declaration.
    */

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

struct sembuf acquire = {0, -1, SEM_UNDO},
              release = {0,    1,
              SEM_UNDO};
#define SEM_FILE   0
main(void)
{
   int         sem1;
   key_t       ipc_key;
   char        string[128];
   static ushort      initSem[1] = {1};
   union semun        arg;
   FILE      *pFile;

   ipc_key = ftok(".", ’R’);

   /*
    * Access the semaphore set.
    */
   if ( (sem1 = semget((int) ipc_key, 1,
        IPC_CREAT | 0666)) != -1 ) {

      arg.array = initSem;

      if ( semctl(sem1, 0, SETALL, arg) == -1 ) {
      perror("semctl -- read_sem -- initialization"); exit(1);
```

```
        }
    }
    else  {
        perror("semget  --  read_sem  --  access file");
        exit(2);
    }

    while  ( 1  ) {

        acquire.sem_num  = SEM_FILE;

        /*
          * Acquiring  the  semaphore  set  before  reading  from  the  file.
          */
        if ( semop(sem1,  &acquire,  1)  == -1  ) { perror("semop  --  read_sem
            --  acquire  semaphore"); exit(3);
        }

        /*
          * Opening  the  file  for  reading.
          */
        if ( (pFile  = fopen("./input.txt",  "r"))  !=  NULL  ) {

            while  ( ! feof(pFile)  ) { fscanf(pFile,  "%s",
                  &string[0]); printf("%s\n",  string);
                  memset(string,  0,  128);
            }

            fclose(pFile);

            unlink("input.txt");
        }

        /*
          * Releasing  the  semaphore  set  after  reading  from  the  file.
          */
        release.sem_num  = SEM_FILE;

        if ( semop(sem1,  &release,  1)  == -1  ) { perror("semop  --  read_sem
            --  release  semaphore");
            exit(4);
        }
    }
  }
```

4. Compile the two programs using the gcc compiler to create the executables **write sem** and **read sem** respectively.

5. From a shell window, run the program **write sem**. From a different shell window, run then program **read sem**. Type any sequence of characters and numbers in the window where you ran the **write sem** program. Notice what happens in the **read sem** window.

6. Can you see that the 2 processes are communicating correctly?
7. Execute the command **ps -ef** in a different window. Notice that the 2 programs are active.
8. Terminate the 2 processes **write sem** and **read sem** either by using the **kill** command as described above or using **Ctrl-C** (in case they run in the foreground).
9. Execute the command **ps -ef**. Are your processes still active?

TODO:

It will be given during the lab based on material covered in this section.

Shared Memory

1. Execute **man shmget**, **man shmctl**, **man shmat** and check what these functions do. Check also the different arguments these functions require.

2. We intend to implement a producer-consumer communication technique using Shared Memory. Both the producer and consumer applications run on the same platform. Briefly, the steps taken by the processes involved are as follows:

   - A parent process forks the producer and consumer processes.
   - The producing process generates a series of random messages that are stored in a shared memory segment for the consumer process to read.
   - Since the producer and consumer may operate at different rates, an array with six message buffers is used.
   - The message buffer array is treated as a queue, whereby new messages are added to the tail of the list and messages to be processed are removed from the head of the list.
   - The two integer indices, referencing the head and tail of the list respectively, are also stored in the shared memory segment.

3. Type the following program which creates the parent part. Name the file parent **shmem.c**:

```
/*
  * The  PARENT
  */
#include  "local_shmem.h"
main(int  argc, char *argv[])
{
    static  struct        MEMORY   memory;
    static  ushort        start_val[2]  = {N_SLOTS,  0};
    int                   semid,  shmid,  croaker;
    char                 *shmptr;
    pid_t                 p_id,  c_id,  pid  = getpid();
    union  semun          arg;

    memory.head  = memory.tail  = 0;
```

```
if ( argc  !=  3  ) {
   fprintf(stderr,  "%s  producer_time  consumer_time\n",  argv[0]);
   exit(-1);
}

/*
  *  Create,  attach  and  initialize  the  memory  segment
  */
if ( (shmid  = shmget((int)  pid,  sizeof(memory),
       IPC_CREAT  | 0600))  !=  -1  ) {

   if ( (shmptr  = (char  *)  shmat(shmid,  0,  0))  == (char  *)  -1  ) {
      perror("shmptr  --  parent  --  attach");
      exit(1);
   }
   memcpy(shmptr,  (char  *)  &memory,  sizeof(memory));
}
else  {
   perror("shmid  --  parent  --  creation");
   exit(2);
}

/*
  *  Create  and  initialize  the  semaphores
  */
if ( (semid  = semget((int)  pid,  2,  IPC_CREAT  | 0666))  !=  -1  ) {
   arg.array  = start_val;

   if ( semctl(semid,  0,  SETALL,  arg)  == -1  ) {
      perror("semctl  --  parent  --  initialization");
      exit(3);
   }


   }
   else  {
      perror("semget  --  parent  --  creation");
      exit(4);
   }

   /*
     *  Fork  the  producer  process
     */
   if ( (p_id  = fork())  == -1  ) {
      perror("fork  --  producer");
      exit(5);
   }
   else  if ( p_id  == 0  ) {
      execl("./producer",  "./producer",  argv[1],  (char  *)  0);
      perror("execl  --  producer");
```

```
        exit(6);
    }

    /*
     *  Fork  the  consumer  process
     */
    if ( (c_id  = fork())  == -1  ) {
        perror("fork  --  consumer");
        exit(7);
    }
    else  if ( c_id  == 0  ) {
        execl("./consumer",  "./consumer",  argv[2],  (char  *)  0);
        perror("execl  --  consumer");
        exit(8);
    }

    croaker  = (int)  wait((int  *)  0);  /*  wait  for  1  to  die  */
    kill(  (croaker  == p_id)  ?  c_id  : p_id,  SIGKILL);

    shmdt(shmptr);
    shmctl(shmid,  IPC_RMID,  (struct  shmid_ds  *)  0);  /*  remove  */
    semctl(semid,  0,  IPC_RMID,  0);

    exit(0);
}
```

4. Type the following program which creates the producer part. Name the file **producer.c**:

```
/*
 * The  PRODUCER
 */

#include  "local_shmem.h"

 main(int  argc,  char  *argv[])
{
    static  char         *source[ROWS][COLS]  = {
       {"A",  "The",  "One"},
        {" red",  " polka-dot",  " yellow"},
        {" spider",  " dump  truck",  " tree"},
        {" broke",  " ran",  " fell"},
        {" down",  " away",  " out"}
    };

    static  char            local_buffer[SLOT_LEN];
    int                     i, r, c,  sleep_limit,  semid,  shmid;
    pid_t                   ppid  = getppid();
    char                   *shmptr;
    struct  MEMORY   *memptr;
```

```
if ( argc  !=  2  ) {
   fprintf(stderr,  "%s  sleep_time\n",  argv[0]);
   exit(-1);
}

/*
  *  Access,  attach  and  reference  the  shared  memory
  */
if ( (shmid  = shmget((int)  ppid,  0,  0))  !=  -1  ) {
   if ( (shmptr  = (char  *)  shmat(shmid,  (char  *)0,  0))  == (char  *)  -1  ) {
      perror("shmat  --  producer  --  attach");
      exit(1);
   }
   memptr  = (struct  MEMORY   *)  shmptr;
}
else  {
   perror("shmget  --  producer  --  access");
   exit(2);
}

/*
  *  Access  the  semaphore  set
  */
if ( (semid  = semget((int)  ppid,  2,  0))  == -1  ) {
   perror("semget  --  producer  --  access");
   exit(3);
}

sleep_limit  = atoi(argv[1])  %   20; i  =  20

- sleep_limit; srand((unsigned)  getpid());

while  ( i--  ) {
   memset(local_buffer,  '\0',  sizeof(local_buffer));

   for  ( r = 0;  r < ROWS;  ++r  ) {  /*  Make  a  random  string  */
      c  = rand()  %   COLS;
      strcat(local_buffer,  source[r][c]);

   }
   acquire.sem_num  = AVAIL_SLOTS;

   if ( semop(semid,  &acquire,  1)  == -1  ) {
      perror("semop  --  producer  --  acquire");
       exit(4);
   }
   strcpy(memptr->buffer[memptr->tail],  local_buffer);
   printf("P:  [%d]  %s.\n",  memptr->tail,  memptr->buffer[memptr->tail]);
```

```
        memptr->tail  = (memptr->tail  + 1)  %   N_SLOTS;
        release.sem_num  = TO_CONSUME;

        if ( semop(semid,  &release,  1)  == -1  ) {
           perror("semop  --  producer  --  release");
           exit(5);
        }
        sleep(rand()  %   sleep_limit  + 1);
     }

     exit(0);
           }
```

5. Type the following program which creates the consumer part. Name the file **consumer.c**:

```
/*
  * The  CONSUMER
  */
#include  "local_shmem.h" main(int
argc,  char  *argv[])
{
   static  char            local_buffer[SLOT_LEN];
   int                     i, sleep_limit,  semid,  shmid;
   pid_t                   ppid  = getppid();
   char                   *shmptr;
   struct  MEMORY   *memptr;

   if ( argc  !=  2  ) {
      fprintf(stderr,  "%s  sleep_time\n",  argv[0]);
      exit(-1);
   }

   /*
    *  Access,  attach  and  reference  the  shared  memory
    */
   if ( (shmid  = shmget((int)  ppid,  0,  0))  !=  -1  ) {
      if ( (shmptr  = (char  *)  shmat(shmid,  (char  *)0,  0))  == (char  *)  -1  ) {
         perror("shmat  --  consumer  --  attach");
         exit(1);
      }
      memptr  = (struct  MEMORY   *)  shmptr;
   }
   else  {
      perror("shmget  --  consumer  --  access");
      exit(2);
   }
```

```
    /*
     * Access the semaphore set
     */
    if ( (semid = semget((int) ppid, 2, 0)) == -1 ) {
        perror("semget -- consumer -- access");
        exit(3);
    }

    sleep_limit = atoi(argv[1]) % 20; i = 20
    - sleep_limit;

    srand((unsigned) getpid());
    while ( i ) {
        acquire.sem_num = TO_CONSUME;

        if ( semop(semid, &acquire, 1) == -1 ) {
            perror("semop -- consumer -- acquire");
            exit(4);
        }
        memset(local_buffer, '\0', sizeof(local_buffer));
        strcpy(local_buffer, memptr->buffer[memptr->head]);
         printf("C: [%d] %s.\n", memptr->head, local_buffer);

        memptr->head = (memptr->head + 1) % N_SLOTS;

        release.sem_num = AVAIL_SLOTS;

        if ( semop(semid, &release, 1) == -1 ) {
            perror("semop -- consumer -- release");
            exit(5);
        }
        sleep(rand() % sleep_limit + 1);
    }
    exit(0);
}
```

6. Type the following header file which is common between the producer and consumer processes. Name the file **local shmem.h**:

```
#ifndef    LOCAL_SH_H_
#define    LOCAL_SH_H_

/** Common header file: parent, producer and consumer*/

 #include <stdio.h>
 #include <unistd.h>
 #include <stdlib.h>
 #include <string.h>
```

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

#define    ROW     5
#define    COLS    3

#define  SLOT_LEN  50
#define  N_SLOTS    6

/*  This  declaration  is *MISSING*  is many  solaris  environments.
     It should  be  in  the  <sys/sem.h>  file  but  often  is not!  If you
     receive  a  duplicate  definition  error  message  for  semun then
     comment  out  the  union  declaration.
     */

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

struct  MEMORY   {
   char  buffer[N_SLOTS][SLOT_LEN];
   int    head,  tail;
};

struct  sembuf  acquire  = {0,  -1,  SEM_UNDO},
                        release  = {0,       1,
                        SEM_UNDO};

enum  {AVAIL_SLOTS,  TO_CONSUME};

#endif
```

7. Compile the parent, the producer and the consumer files to create the programs parent **shmem**, producer and consumer respectively.
8. Run the parent process and notice the output you get. Change the sleeping time for the producer and consumer processes and notice how the output differs in each case.

TODO:

It will be given during the lab based on material covered in this section.