Lab 4: Inter-Process Communication (Part I)

Yu-Lun Huang

2011-07-11

1 Objective

- Be familiar with inter-process communication: semaphore, mutex, etc.

2 Prerequisite

- Read man pages of `semop()`, `semctl()`, `semget()`, `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, etc.

3 Semaphore

Unix allocates arrays of semaphores, rather than creating them one at a time. When you create such an array, you must supply the following information:

- An integer called the "key" which acts as the semaphore array's "name" on the system
- The number of semaphores in the array
- Ownership of semaphore array (permission bits/mode)

The system call `semget(2)` is used by a program to ask the operating system if it knows of a semaphore. The program passes the semaphore's key. If the semaphore exists, then the operating system returns an integer which is used by the program as the semaphore's identifier for the duration of that program. (NOTE: The semaphore key is permanent for as long as the semaphore exists. The semaphore ID returned by the operating system is just a temporary "handle" for accessing the semaphore and may be different each time.) `semget(2)` can also be used to create semaphores that don't exist by passing additional options. `semctl(2)` is used to set the value of a semaphore, remove it from the system, etc. Think of it as the way to "manage" the semaphore array. At it's simplest, `semop(2)` is used to implement `P()` (wait) and `V()` (signal). However, `semop(2)` can do multiple semaphore operations with one call. For our programs, we will only be creating arrays with one semaphore on them and doing only one operation at a time.

- Run the following commands on both your Linux host and target, and observe the status of System V IPC status.
  - ipcs
  - ipcs -s
- Here is a program, **makesem.c**. Compile it and run it with two parameters:
  - the first parameter should be a large number;
  - the second one is number '1' or '0'.

For example: makesem 428361733 1.

Now run `ipcs` again. You should see your semaphore in the list. Note that the key is the number you entered that converted to hex.

```
/* makesem.c
 *
 * This program creates a semaphore. The user should pass
 * a number to be used as the semaphore key and initial
 * value as the only command line arguments. If that
 * identifier is not taken, then a semaphore will be created.
 * If a semaphore is set so that then no semaphore will be
 * created. The semaphore is set so that anyone on the system
 * can use it.
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sem.h>

#define SEMMODE 0666 /* rw(owner)-rw(group)-rw(other) permission */

int main (int argc, char **argv)
{
   int s;
   long int key;
   int val;

   if (argc != 3)
   {
      fprintf(stderr,
         "%s: specify a key (long) and initial value (int)\n",
         argv[0]);

      exit(1);
   }

   /* get values from command line */
   if (sscanf(argv[1], "%ld", &key) != 1)
   {
      /* convert arg to long integer */
      fprintf(stderr, "%s: argument #1 must be an long integer\n",
            argv[0]);
      exit(1);
   }
   if (sscanf(argv[2], "%d", &val) != 1)
   {
      /* convert arg to long integer */
      fprintf(stderr, "%s: argument #2 must be an integer\n",
            argv[0]);
      exit(1);
```

```
    }

    /* semget() takes three parameters */
    /* Options:
     *    IPC_CREAT - create a semaphore if not exists
     *    IPC_EXCL  - creation fails if it already exists
     *    SEM_MODE  - access permission
     */
    s = semget(
          key, /* the unique name of the semaphore on the system */
          1,   /* we create an array of semaphores, but just need 1. */
          IPC_CREAT | IPC_EXCL | SEM_MODE);

    /* If semget () returns -1 then it failed. However,
     * if it returns any other number >= 0 then that becomes
     * the identifier within the program for accessing the semaphore.
     */
    if (s < 0)
    {
       fprintf(stderr,
             "%s: creation of semaphore %ld failed: %s\n", argv[0],
             key, strerror(errno));
       exit(1);
    }
    printf("Semaphore %ld created\n", key);

    /* set semaphore (s[0]) value to initial value (val) */
    if ( semctl(s, 0, SETVAL, val) < 0 )
    {
       fprintf(stderr,
             "%s: Unable to initialize semaphore: %s\n",
             argv[0], strerror(errno));
       exit(0);
    }
    printf("Semaphore %ld has been initialized to %d\n", key, val);

    return 0;
}
```

- Program **rmsem.c** removes the semaphore identified by its key from the system. Compile and run the program with your key from the last step, then run `ipcs` to see if your semaphore is still listed. Note: The easiest way to create the file is probably to copy `makesem.c` and make the modifications.

```
/* rmsem.c
 *
 * This program destroys a semaphore. The user should pass a number
 * to be used as the semaphore key.
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <sys/sem.h>

int main(int argc, char **argv)
{
   int s;
   long int key;

   if (argc != 2)
   {
      fprintf(stderr, "%s: specify a key (long)\n", argv[0]);
      exit(1);
   }

   /* get values from command line */
   if (sscanf(argv[1], "%ld", &key)!=1)
   {
      /* convert arg to long integer */
      fprintf(stderr,
         "%s: argument #1 must be an long integer\n", argv[0]);
      exit(1);
   }

   /* find semaphore */
   s = semget(key,1,0);
   if (s < 0)
   {
      fprintf(stderr,    "%s: failed to find semaphore %ld: %s\n",
         argv[0], key, strerror(errno));
      exit(1);
   }
   printf("Semaphore %ld found\n",key);

   /* remove semaphore */
   if (semctl (s, 0, IPC_RMID, 0) < 0)
   {
      fprintf (stderr, "%s: unable to remove semaphore %ld\n",
         argv[0], key);
      exit(1);
   }
   printf("Semaphore %ld has been remove\n", key);

   return 0;
}
```

- The program **doodle.c** has definitions for the operations P() (wait) and V() (signal). The program first "opens" the semaphore with `semget()`, then it waits for the user to type in a small integer number (number of seconds to stay in the critical section). It then performs P() on a semaphore. Once inside the critical section, it doodles for the number of seconds you specified, then it leaves performing V() on the semaphore.

To demonstration the operation of `doodle`, please run the following two experiments:

1. Use `makesem` to create a semaphore with initial value `1` and run three `doodle` programs to acquire the same semaphore simultaneously.
2. Use `makesem` to create a semaphore with initial value `2` and run three `doodle` programs to acquire the same semaphore simultaneously.

Observe the inter-operations between the three `doodle` programs.

```
/* doodle.c
 *
 * This program shows how P () and V () can be implemented,
 * then uses a semaphore that everyone has access to.
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sem.h>
#include <unistd.h>

#define DOODLE_SEM_KEY 1122334455

/* P () - returns 0 if OK; -1 if there was a problem */
int P(int s)
{
   struct sembuf sop;   /* the operation parameters */
   sop.sem_num =  0; /* access the 1st (and only) sem in the array */
   sop.sem_op  = -1;    /* wait..*/
   sop.sem_flg =  0;    /* no special options needed */

   if (semop (s, &sop, 1) < 0) {
      fprintf(stderr,"P(): semop failed: %s\n",strerror(errno));
      return -1;
   } else {
      return 0;
   }
}

/* V() - returns 0 if OK; -1 if there was a problem */
int V(int s)
{
   struct sembuf sop;   /* the operation parameters */
   sop.sem_num =  0; /* the 1st (and only) sem in the array */
   sop.sem_op  =  1; /* signal */
   sop.sem_flg =  0; /* no special options needed */

   if (semop(s, &sop, 1) < 0) {
      fprintf(stderr,"V(): semop failed: %s\n",strerror(errno));
      return -1;
   } else {
```

```
        return 0;
    }
}

int main ( int argc, char **argv)
{
    int s, secs;
    long int key;

    if  (argc != 2) {
        fprintf(stderr, "%s: specify a key (long)\n", argv[0]);
        exit(1);
    }

    /* get values from command line */
    if (sscanf(argv[1], "%ld", &key)!=1)
    {
        /* convert arg to long integer */
        fprintf(stderr,
            "%s: argument #1 must be an long integer\n", argv[0]);
        exit(1);
    }


    s = semget(key, 1, 0);

    if (s < 0) {
        fprintf (stderr,
            "%s: cannot find semaphore %ld: %s\n",
            argv[0], key, strerror(errno));
        exit(1);
    }

    while (1) {
        printf ("#secs to doodle in the critical section? (0 to exit):");
        scanf ("%d",&secs);

        if (secs == 0)
            break;

        printf ("Preparing to enter the critical section..\n");
        P(s);
        printf ("Now in the critical section! Sleep %d secs..\n", secs);

        while (secs) {
            printf ("%d...doodle...\n",secs--);
            sleep (1);
        }

        printf ("Leaving the critical section..\n");
        V(s);
```

```
    }

    return 0;
}
```

- Race Condition
  Race conditions arise in software when separate processes or threads of execution depend on some shared state. Operations upon shared states are critical sections that must be mutually exclusive in order to avoid harmful collision between processes or threads that share those states. Assume that two threads (T1 and T2) want to increment the value of a global integer by one. Ideally, the following sequence of operations would take place:

  - Integer i = 0; (memory)
  - T1 reads the value of i from memory into register1: 0
  - T1 increments the value of i in register1: (register1 contents) + 1 = 1
  - T1 stores the value of register1 in memory: 1
  - T2 reads the value of i from memory into register2: 1
  - T2 increments the value of i in register2: (register2 contents) + 1 = 2
  - T2 stores the value of register2 in memory: 2
  - Integer i = 2; (memory)

  In the above case, the final value of i is 2, as expected. However, if the two threads run simultaneously without locking or synchronization, the outcome of the operation could be wrong. The alternative sequence of operations below demonstrates this scenario:

  - Integer i = 0; (memory)
  - T1 reads the value of i from memory into register1: 0
  - T2 reads the value of i from memory into register2: 0
  - T1 increments the value of i in register1: (register1 contents) + 1 = 1
  - T2 increments the value of i in register2: (register2 contents) + 1 = 1
  - T1 stores the value of register1 in memory: 1
  - T2 stores the value of register2 in memory: 1
  - Integer i = 1; (memory)

  The final value of i is 1 instead of the expected result of 2. This occurs because the increment operations of the second case are not mutually exclusive.

  The following example shows a race condition between two processes.

  Please create a file `counter.txt` and set initial value `0` to the file by performing "`echo 0 > counter.txt`". Compile the codes with and without the flag '-D USE_SEM' and check the result in `counter.txt`. Explain the result if any difference.

```
/*
 * race.c
 */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#ifdef USE_SEM
#define SEM_MODE 0666 /* rw(owner)-rw(group)-rw(other) permission */
#define SEM_KEY  1122334455

int sem;

/* P () - returns 0 if OK; -1 if there was a problem */
int P (int s)
{
   struct sembuf sop;   /* the operation parameters */
   sop.sem_num =  0; /* access the 1st (and only) sem in the array */
   sop.sem_op  = -1;    /* wait..*/
   sop.sem_flg =  0;    /* no special options needed */

   if (semop (s, &sop, 1) < 0) {
      fprintf(stderr,"P(): semop failed: %s\n",strerror(errno));
      return -1;
   } else {
      return 0;
   }
}

/* V() - returns 0 if OK; -1 if there was a problem */
int V(int s)
{
   struct sembuf sop;   /* the operation parameters */
   sop.sem_num =  0; /* the 1st (and only) sem in the array */
   sop.sem_op  =  1; /* signal */
   sop.sem_flg =  0; /* no special options needed */

   if (semop(s, &sop, 1) < 0) {
      fprintf(stderr,"V(): semop failed: %s\n",strerror(errno));
      return -1;
   } else {
      return 0;
   }
}

#endif

/* increment value saved in file */
void Increment()
```

```
{
    int ret;
    int fd;     /* file descriptor */
    int counter;
    char buffer[100];
    int i = 10000;

    while(i)
    {
        /* open file */
        fd = open("./counter.txt", O_RDWR );
        if (fd < 0)
        {
            printf("Open counter.txt error.\n");
            exit(-1);
        }

#ifdef USE_SEM
        /* acquire semaphore */
        P(sem);
#endif

        /**************** Critical Section *****************/
        /* clear */
        memset(buffer, 0, 100);

        /* read raw data from file */
        ret = read(fd, buffer, 100);
        if (ret < 0)
        {
            perror("read counter.txt");
            exit(-1);
        }

        /* transfer string to integer & increment counter */
        counter = atoi(buffer);
        counter++;

        /* write back to counter.txt */
        lseek(fd, 0, SEEK_SET); /* reposition to the head of file */

        /* clear */
        memset(buffer, 0, 100);
        sprintf(buffer, "%d", counter);
        ret = write(fd, buffer, strlen(buffer));
        if (ret < 0)
        {
            perror("write counter.txt");
            exit(-1);
        }
        /**************** Critical Section *****************/
```

```
#ifdef USE_SEM
      /* release semaphore */
      V(sem);
#endif

      /* close file */
      close(fd);

      i--;
   }
}

int main(int argc, char **argv)
{
   int childpid;
   int status;

#ifdef USE_SEM
   /* create semaphore */
   sem = semget(SEM_KEY, 1, IPC_CREAT | IPC_EXCL | SEM_MODE);
   if (sem < 0)
   {
      fprintf(stderr, "Creation of semaphore %ld failed: %s\n",
            SEM_KEY, strerror(errno));
      exit(-1);
   }

   /* initial semaphore value to 1 (binary semaphore) */
   if ( semctl(sem, 0, SETVAL, 1) < 0 )
   {
      fprintf(stderr, "Unable to initialize semaphore: %s\n",
            strerror(errno));
      exit(0);
   }

   printf("Semaphore %ld has been created & initialized to 1\n",
      SEM_KEY);
#endif

   /* fork process */
   if ((childpid = fork()) > 0)  /* parent */
   {
      Increment();
      waitpid(childpid, &status, 0);
   }
   else if (childpid == 0) /* child */
   {
      Increment();
      exit(0);
   }
```

```
    else        /* error */
    {
        perror("fork");
        exit(-1);
    }

#ifdef USE_SEM
    /* remove semaphore */
    if (semctl (sem, 0, IPC_RMID, 0) < 0)
    {
        fprintf (stderr, "%s: unable to remove semaphore %ld\n",
                argv[0], SEM_KEY);
        exit(1);
    }
    printf("Semaphore %ld has been remove\n", SEM_KEY);
#endif

    return 0;
}
```

4 Mutex

A mutex is abbreviated from "MUTual EXclusion", and is useful for protecting shared data structures from concurrent modifications, and implementing critical sections and monitors. A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first. Some POSIX library APIs for mutex are:

- `pthread_mutex_init()` initializes the mutex referenced by mutex with attributes specified by *attr*. Upon successful initialization, the state of the mutex becomes initialized and unlocked. For more information, please see its man page.
- `pthread_mutex_lock()` locks the mutex object referenced by the mutex. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.
- `pthread_mutex_unlock()` unlocks the mutex object referenced by the mutex.
- `pthread_mutex_destroy()` destroys the mutex object referenced by mutex; the mutex object becomes, ineffective, uninitialized.

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

#define checkResults(string, val) {                    \
    if (val) {                                         \
        printf("Failed with %d at %s", val, string);   \
        exit(1);                                       \
    }                                                  \
```

```
}

#define                 NUMTHREADS    3

pthread_mutex_t         mutex = PTHREAD_MUTEX_INITIALIZER;
int                     sharedData  = 0;
int                     sharedData2 = 0;

void *theThread(void *parm)
{
   int    rc;

   printf("\tThread %lu: Entered\n", (unsigned long) pthread_self());

   /* lock mutex */
   rc = pthread_mutex_lock(&mutex);
   checkResults("pthread_mutex_lock()\n", rc);

   /**************** Critical Section ****************/
   printf("\tThread %lu: Start critical section, holding lock\n",
         (unsigned long) pthread_self());

   /* Access to shared data goes here */
   ++sharedData;
   --sharedData2;

   printf("\tsharedData = %d, sharedData2 = %d\n",
      sharedData, sharedData2);

   printf("\tThread %lu: End critical section, release lock\n",
      (unsigned long) pthread_self());
   /**************** Critical Section ****************/

   /* unlock mutex */
   rc = pthread_mutex_unlock(&mutex);
   checkResults("pthread_mutex_unlock()\n", rc);

   return NULL;
}

int main(int argc, char **argv)
{
   pthread_t               thread[NUMTHREADS];
   int                     rc = 0;
   int                     i;

   /* lock mutex */
   printf("Main thread hold mutex to prevent access to shared data\n");
   rc = pthread_mutex_lock(&mutex);
   checkResults("pthread_mutex_lock()\n", rc);
```

```
    /* create thread */
    printf("Main thread create/start threads\n");
    for (i = 0; i < NUMTHREADS; ++i) {
        rc = pthread_create(&thread[i], NULL, theThread, NULL);
        checkResults("pthread_create()\n", rc);
    }

    /* wait for thread creation complete */
    printf("Main thread wait a bit until 'done' with the shared data\n");
    sleep(3);

    /* unlock mutex */
    printf("Main thread unlock shared data\n");
    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_lock()\n",rc);

    /* wait thread complete */
    printf("Main thread Wait for threads to complete, "
           "and release their resources\n");
    for (i=0; i <NUMTHREADS; ++i) {
        rc = pthread_join(thread[i], NULL);
        checkResults("pthread_join()\n", rc);
    }

    /* destroy mutex */
    printf("Main thread clean up mutex\n");
    rc = pthread_mutex_destroy(&mutex);

    printf("Main thread completed\n");

    return 0;
}
```

5 PXA I/O: 7-Segment

- Header files: asm-arm/arch-pxa/lib/creator_pxa270_lcd.h
- Function: ioctl(fd, command, data)
  - Command:
    _7SEG_IOCTL_ON: turn on 7 segment LED (no data is needed)
    _7SEG_IOCTL_OFF: turn off 7 segment LED (no data is needed)
    _7SEG_IOCTL_SET: set 7 segment LED (_7seg_info_t)
  - Data:

```
typedef struct _7Seg_Info{
    unsigned char Mode;  // _7SEG_MODE_PATTERN or _7SEG_MODE_HEX_VALUE
    unsigned char Which; // D5 ~ D8
    unsigned long Value; // pattern or hex
} _7seg_info_t;
```

- The setting of _7Seg_Info:
  - flags used in Mode:

```
#define _7SEG_MODE_PATTERN          0
#define _7SEG_MODE_HEX_VALUE        1
```

  - flags used in Which:

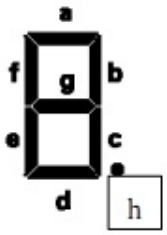
```
#define _7SEG_D5_INDEX   8  // Segment D5 (1)
#define _7SEG_D6_INDEX   4  // Segment D6 (2)
#define _7SEG_D7_INDEX   2  // Segment D7 (3)
#define _7SEG_D8_INDEX   1  // Segment D8 (4)
#define _7SEG_ALL
        (_7SEG_D5_INDEX|_7SEG_D6_INDEX|_7SEG_D7_INDEX|_7SEG_D8_INDEX)
```

The following is the sample code of 7 segment display control.

```
/*
 * 7segment.c -- the sample code for controlling 7-segment dispaly.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include "asm-arm/arch-pxa/lib/creator_pxa270_lcd.h"

int main(int argc, char* argv[])
{
    int fd;
    _7seg_info_t    data;

    /* Open device /dev/lcd */
    if((fd = open("/dev/lcd", O_RDWR)) < 0)
    {
        printf("Open /dev/lcd faild.\n");
    }

    /* Open 7 Segment LED */
    ioctl(fd, _7SEG_IOCTL_ON, NULL);

    /*
     * In HEX_VALUE mode, the settings is based on hex value
     * Each segment is represented by 4 bits (0 ~ f)
     */
    data.Mode = _7SEG_MODE_HEX_VALUE;
    data.Which = _7SEG_ALL;
    data.Value = 0x2004;
    ioctl(fd, _7SEG_IOCTL_SET, &data);  /* Setup 7 Segment LED */
    sleep (3);
```

the bit value 1: on

the bit value 0: off



8 bits to represents each Segment

h	g	f	e	d	c	b	a

Figure 1: The layout of 7 segment display

```
    /*
     * In PATTERN mode, the settings is based on the layout of 7 segment
     *    (The layout is shown in Figure 1)
     * Each segment is represented by 8 bits
     *    the bit value 0: off
     *    the bit value 1: on
     */
    data.Mode = _7SEG_MODE_PATTERN;
    data.Which = _7SEG_D5_INDEX | _7SEG_D8_INDEX;
    data.Value = 0x6d7f; /* the output is 5008 */
    ioctl(fd, _7SEG_IOCTL_SET, &data);  /* Setup 7 Segment LED */
    sleep (3);

    /* Close 7 Segment LED */
    ioctl(fd, _7SEG_IOCTL_OFF, NULL);

    close(fd);

    return 0;
}
```

Lab 4 Assignments

Create two files, File1 and File2, storing two fibonacci numbers, f(i) and f(i+1), where i ≥ 1. Write a program running two threads/processes. Each thread/process:

- Compete to read the values from File1 and File2.
- Replace f(i) in File1 with f(i+1) in File2.
- Calculate f(i+2) and replace it with f(i+1) in File2.
- Repeat the above steps for 100 times.
- Show the results in File2 on the 7-Segment.

If the calculation result is larger than 4 digits, show the last 4 digits on the 7 segment display.