Lab 5: Inter-Process Communication (Part II)

Yu-Lun Huang

2011-11-21

1 Objective

- Be familiar with inter-process communication: pipe, shared memory, etc.

2 Prerequisite

- Read man pages of `pipe()`, `shmget()`, `shmopt()`, `shmat()`, `shmdt()`, etc.

3 Pipe

A pipe creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing. An unnamed pipe can be viewed and accessed by processes with parent-child relationship. To share a pipe among all processes, you need to create a named pipe.

The following program creates a pipe, and then fork(2) to create a child process. After the fork(2), each process closes the descriptors that it doesn't need for the pipe (see pipe(7)). In the following sample code, the child process reads the file specified in `argv[1]`, and writes the file content to the parent process through the pipe. The parent process reads the data from the pipe and echoes the data on the screen.

```
/* pipe.c
 *
 * child process read the content of file
 * and write the content to parent process through pipe
 */

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int pfd[2]; /* pfd[0] is read end, pfd[1] is write end */

void ChildProcess(char *path)
{
    int fd;
```

```c
    int ret;
    char buffer[100];

    /* close unused read end */
    close(pfd[0]);

    /* open file */
    fd = open(path, O_RDONLY);
    if (fd < 0) {
        printf("Open %s failed.\n", path);
        exit(EXIT_FAILURE);
    }

    /* read file and write content to pipe */
    while (1) {
        /* read raw data from file */
        ret = read(fd, buffer, 100);

        if (ret < 0) {     /* error */
            perror("read()");
            exit(EXIT_FAILURE);
        }
        else if (ret == 0) { /* reach EOF */
            close(fd);  /* close file */
            close(pfd[1]); /* close write end, reader see EOF */
            exit(EXIT_SUCCESS);
        }
        else {          /* write content to pipe */
            write(pfd[1], buffer, ret);
        }
    }
}

void ParentProcess()
{
    int ret;
    char buffer[100];

    /* close unused write end */
    close(pfd[1]);

    /* read data from pipe until reach EOF */
    while(1) {
        ret = read(pfd[0], buffer, 100);

        if (ret > 0) { /* print data to screen */
            printf("%.*s", ret, buffer);
        }
        else if (ret == 0) { /* reach EOF */
            close(pfd[0]); /* close read end */
            wait(NULL);
```

```
            exit(EXIT_SUCCESS);
        }
        else {
            perror("pipe read()");
            exit(EXIT_FAILURE);
        }
    }
}

int main(int argc, char *argv[])
{
    pid_t cpid;

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

    /* create pipe */
    if (pipe(pfd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    /* fork child process */
    cpid = fork();
    if (cpid == -1) { /* error */
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0)
        ChildProcess(argv[1]);
    else
        ParentProcess();

    return 0;
}
```

Monitoring multiple file descriptors with polling strategy may cause busy waiting, which lowers down the system performance. To effectively monitor multiple descriptors, you can use the system call `select()` to perform block waiting, rather than busy waiting. The calling process specifies the interesting file descriptors to `select()` and performs blocking waiting. When one or more descriptors become "ready" (ready for read or ready for write), `select()` informs the calling process to awake from blocking state. Then, a user can check each file descriptor to perform read/write operation. In following program, two child processes sleep a random time, then send a message to the parent process. The parent process uses `select()` to perform blocking waiting, until one of the pipes is ready to read.

```
/*
 * select.c
 */

```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

#define max(a, b) ((a > b) ? a : b)

void ChildProcess(int *pfd, int sec)
{
   char buffer[100];

   /* close unused read end */
   close(pfd[0]);

   /* sleep a random time to wait parent process enter select() */
   printf("Child process (%d) wait %d secs\n", getpid(), sec);
   sleep(sec);

   /* write message to parent process */
   memset(buffer, 0, 100);
   sprintf(buffer, "Child process (%d) sent message to parent process\n",
      getpid());
   write(pfd[1], buffer, strlen(buffer));

   /* close write end */
   close(pfd[1]);

   exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[])
{
   int pfd1[2], pfd2[2];   /* pipe's fd */
   int cpid1, cpid2; /* child process id */
   fd_set rfds, arfds;
   int max_fd;
   struct timeval tv;
   int retval;
   int fd_index;
   char buffer[100];

   /* random seed */
   srand(time(NULL));

   /* create pipe */
   pipe(pfd1);
   pipe(pfd2);

```

```c
/* create 2 child processes and set corresponding pipe & sleep time */
cpid1 = fork();
if (cpid1 == 0)
   ChildProcess(pfd1, random() % 5);

cpid2 = fork();
if (cpid2 == 0)
   ChildProcess(pfd2, random() % 4);

/* close unused write end */
close(pfd1[1]);
close(pfd2[1]);

/* set pfd1[0] & pfd2[0] to watch list */
FD_ZERO(&rfds);
FD_ZERO(&arfds);
FD_SET(pfd1[0], &arfds);
FD_SET(pfd2[0], &arfds);
max_fd = max(pfd1[0], pfd2[0]) + 1;

/* Wait up to five seconds. */
tv.tv_sec = 5;
tv.tv_usec = 0;

while(1)
{
   /* config fd_set for select */
   memcpy(&rfds, &arfds, sizeof(rfds));

   /* wait until any fd response */
   retval = select(max_fd, &rfds, NULL, NULL, &tv);

   if (retval == -1) {  /* error */
      perror("select()");
      exit(EXIT_FAILURE);
   }
   else if (retval) {   /* # of fd got respone */
      printf("Data is available now.\n");
   }
   else {   /* no fd response before timer expired */
      printf("No data within five seconds.\n");
      break;
   }

   /* check if any response */
   for (fd_index = 0; fd_index < max_fd; fd_index++)
   {
      if (!FD_ISSET(fd_index, &rfds))
         continue;   /* no response */

      retval = read(fd_index, buffer, 100);
```

```
        if (retval > 0)      /* read data from pipe */
          printf("%.*s", retval, buffer);
        else if (retval < 0) /* error */
          perror("pipe_read()");
        else {          /* write fd closed */
          /* close read fd */
          close(fd_index);
          /* remove fd from watch list */
          FD_CLR(fd_index, &arfds);
        }
      }
    }

    return 0;
}
```

4 Shared Memory

Shared memory is a memory space that may be simultaneously accessed by multiple processes with an intent to provide inter-process communication among them or avoid redundant copies. Unlike unnamed pipes, only exist among processes with parent-child relationship, every process can access the shared memory space with the `share memory key` specified.

The followings are two processes communicating via shared memory: `shm_server.c` and `shm_client.c`. The two programs here illustrate the passing of a simple piece of memory (a string) between the processes if running simultaneously:

```
/*
 * shm_server.c -- creates the string and shared memory.
 */

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

#define SHMSZ     27

int main(int argc, char *argv[])
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;
    int retval;

    /* We'll name our shared memory segment "5678" */
    key = 5678;

```

```c
    /* Create the segment */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Now we attach the segment to our data space */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    printf("Server create and attach the share memory.\n");

    /* Now put some things into the memory for the other process to read */
    s = shm;

    printf("Server write a ~ z to share memory.\n");
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = '\0';

    /*
     * Finally, we wait until the other process changes the first
     * character of our memory to '*', indicating that it has read
     * what we put there.
     */
    printf("Waiting other process read the share memory ...\n");
    while (*shm != '*')
        sleep(1);
    printf("Server read * from the share memory.\n");

    /* Detach the share memory segment */
    shmdt(shm);

    /* Destroy the share memory segment */
    printf("Server destroy the share memory.\n");
    retval = shmctl(shmid, IPC_RMID, NULL);
    if (retval < 0)
    {
        fprintf(stderr, "Server remove share memory failed\n");
        exit(1);
    }

    return 0;
}
```

```c
/*
 * shm_client.c -- attaches itself to the created shared memory
 *                 and uses the string (printf).
 */

```

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

#define SHMSZ     27

int main(int argc, char *argv[])
{
    int shmid;
    key_t key;
    char *shm, *s;

    /* We need to get the segment named "5678", created by the server */
    key = 5678;

    /* Locate the segment */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Now we attach the segment to our data space */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    printf("Client attach the share memory created by server.\n");

    /* Now read what the server put in the memory */
    printf("Client read characters from share memory ...\n");
    for (s = shm; *s != '\0'; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the segment to '*',
     * indicating we have read the segment.
     */
    printf("Client write * to the share memory.\n");
    *shm = '*';

    /* Detach the share memory segment */
    printf("Client detach the share memory.\n");
    shmdt(shm);

    return 0;
}
```

Lab 5 Assignments

Yu-Lun Huang

2011-11-21

- Rewrite your Lab3 assignment and use `pipe` to transfer LED control signals between processes. (Note: Do not create a child process for each control signal)
- Rewrite your Lab4 assignment and replace the two files with **`shared memories`**.
- Design a program to calculate the summation of a series.
  - The program allows user to input two integers, `N` and `M`.
  - The parent process then creates `N` child processes to calculate the summation from `1` to `N×M`
  - Each child process receives a serial number **`n`** (ranged from 0 to $N-1$) and the integer `M` from parent process.
  - The child process calculates $\sum_{(nM+1)}^{(nM+M)}$ and writes the summation result back to parent process using `pipe`.
  - The parent process uses `select()` to read the summation results of child processes and sums up the final result.
  - Display the last 4 digits of the final result on 7-segment display.