```
/*
 * This is a free program sample that may be reproduced in any form.
 * The author's information should be retained to preserve its identity.
 *
 * Date written: November 19, 2001
 * Written by: Peraphon Sophatsathit
 * Department of Mathematics, Faculty of Science, Chulalongkorn University.
 * email: Peraphon.S@chula.ac.th, http://pioneer.netserv.chula.ac.th/~sperapho
 *
 * Distributed Operating Systems (2301462) classnote.
 * Description: This sample program illustrates multiple access to shared
 *              memory area under multitasking IPC setting.
 */

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

/*
 * porting from BSD to SVR4
 */
#ifdef __USE_BSD
#  include     <machine/param.h>
#endif

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

/*
 * prototypes
 */
unsigned int   alarm(unsigned int);
void           (*signal(int, void (*disp)(int))) (int);
void           handler(int);
void           t_out(int);
void           racing(int *, int);
int            proc_loop(int);
int            driver(int *);

/*
 * globals
 */
#define        Small          20
#define        TRUE           1
#define        FALSE          0
#define        TRY            5             /* loop only TRY times for demo purpose      */
#define        SEC            1             /* just to use up processing time slice      */
#define        ILLEGAL        -9999         /* undesirable value to signal child process */
#define        ILL_range      10            /* depends on how fast swapping takes place  */
#define        MemSize        4             /* shared memory size                        */
#define        Test_val       5             /* arbitary initial shared value             */
```

```
int            old_val;
int            flag    = FALSE;
int            freq    = TRY;
unsigned int   Default = 1;

/*
 * The purpose of signal calls employed in this shared memory IPC is to
 * prevent runaway processing.  The user may terminate (kill) the process
 * any time via user command (ctrl C) or timer.  The latter can be set to any
 * positive integer ranging from 0 to N (N is recommneded to be small to have
 * any effect).
 * Note also that all pending signals are not inherited by the child process,
 * hence the extra trick to inform the child process to end.
 */
int
main(int ac, char **av)
{
       char           buf[Small];
       int            rt_code;
       unsigned int   sec;

       signal(SIGINT, handler);
       signal(SIGQUIT, handler);
       signal(SIGALRM, t_out);
       switch (ac)
       {
              case 4:
                     freq    = atoi(av[3]);
              case 3:
                     Default = (unsigned int)atoi(av[2]);
              case 2:
                     strcpy(buf, av[1]);
                     break;
              default:
                     printf("\nUsage: %s   wait_sec  [  parent_pause_sec  #_of_try  ]\n", av[0]);
                     printf("Example: %s   0            (no timer is set)\n", av[0]);
                     printf("Example: %s   0   3        (parent pause 3 seconds)\n", av[0]);
                     printf("Example: %s   0   3   10   (repeat loop 10 times)\n\n", av[0]);
                     printf("1) Wait time for 0 second is recommended.  Any other values\n");
                     printf("can be used as a precaution to prevent the processes from\n");
                     printf("running away, but will cause an abnormal termination.\n");
                     printf("However, too long a wait will have no effect if both processes\n");
                     printf("have already terminated.\n");
                     printf("2) The parent process wlll pause 1 second (default) to\n");
                     printf("relinguish the CPU to child process.  A larger setting\n");
                     printf("will yield faster swap out of parent process.\n\n");
                     return 1;
       }
       sec = atoi(buf);
       if (sec > 0)
              alarm(sec);
       rt_code = proc_loop((int)sec);
       if (rt_code > 0 || flag == TRUE)
              perror("Abnornal termination of IPC loop");
       fflush(stdout);
       fflush(stderr);
       return 0;
}
```

```
/*
 * setup shared memory area.
 */
int
proc_loop(int num)
{
        key_t   key;
        int     shm_flag;
        int     shmid;
        int     *ptr;
        void    *shm_addr;

        /*
         * create a shared memory area and grant R/W permission to child process
         */
        key = IPC_PRIVATE;
        shm_flag = IPC_CREAT | SHM_R | SHM_W;
        if ((shmid = shmget(key, MemSize, shm_flag)) == -1)
        {
                perror("Unable to get shared memory");
                return 2;
        }
        /*
         * attach the shared memory area to the IPC pipeline
         */
        shm_flag = IPC_SET | (SHM_R | SHM_W);
        if ((shm_addr = shmat(shmid, (void *)0, shm_flag)) == (void *)-1)
        {
                perror("Unable to attach shared memory");
                return 3;
        }

        /*
         * begin process creation and communication
         */
        ptr     = shm_addr;
        *ptr    = Test_val;
        old_val = *ptr;
        printf("\nInitial shared value is [%d]\n\n", old_val);
        if (driver(ptr) > 0)
        {
                perror("fork and exec failed");
        }

        /*
         * detach the shared memory area from the IPC pipeline
         */
        if (shmdt(shm_addr) == -1)
        {
                perror("Unable to detach shared memory");
                return 4;
        }
        return 0;
}
```

```
/*
 * The driver function spawns a child process and starts a race situation to
 * access shared memory.  The two processes will terminate abnormally when
 * timer (alarm) is set or interrupt signal is received by the parent process.
 * In which case, an 'illegal' value is passed from the parent to child,
 * informing the child process to stop processing immediately.  This simple minded
 * approach merely illustrates how to prevent both processes from running away.
 */
int
driver(int *sm)
{
	int	pid = 0;
	int	i;

	pid = fork();
	if (pid == 0)
	{
		printf("begin child process\n");
		for (i = 0; i < freq; i++)
		{
			if (old_val > ILLEGAL-ILL_range && old_val < ILLEGAL+ILL_range)
				break;
			if (*sm != old_val)
			{
				old_val = *sm;
				sleep(SEC);
			}
			racing(sm, pid);
		}
		if (i >= freq && *sm > ILLEGAL+ILL_range)
			printf("\nNormal termination of child process\n\n");
		else
			printf("\nAbnormal termination of child process\n\n");
	}
	else if (pid > 0)
	{
		printf("parent: spawn succeeded!\n");
		sleep(Default);
		for (i = 0; i < freq && flag == FALSE; i++)
		{
			racing(sm, pid);
		}
		if (flag == FALSE)
			printf("\nNormal termination of parent process\n\n");
		else
		{
			*sm = ILLEGAL;
			printf("\nAbnormal termination of parent process\n\n");
		}
	}
	else
	{
		printf("fork failed: parent exiting...\n");
		return 5;
	}
	return 0;
}
```

```
/*
 * modify the value stored in shared memory area.
 */
void
racing(int *sm, int id)
{
	(*sm)++;
	if (id == 0)
		printf("called by child process with the shared value  = %d\n", *sm);
	else
		printf("called by parent process with the shared value = %d\n", *sm);
	return;
}

/*
 * time out by alarm clock.
 */
void
t_out(int sig)
{
	signal(SIGALRM, t_out);
	flag = TRUE;
	printf("timeout by ALARM signal\n");
	return;
}

/*
 * interrupt and kill signals.
 */
void
handler(int sig)
{
	signal(SIGINT, handler);
	signal(SIGQUIT, handler);
	flag = TRUE;
	printf("receiving INT/QUIT signal\n");
	return;
}
```