A report on measuring the time it takes for a signal to be passed from one process to another.

Shah Dhruv Amitkumar

(Redid: 819653114)

Course Number: COMPE 571

Professor Name: Dr. Yusuf Ozturk

Date Submitted: 10-19-2015

Contents

Introduction

The main purpose of the experiment is to measure the time it take for a signal to be passed from one process to another.

In multiprocessor system, shared memory plays a vital role. In multiprocessor system, when processes try to access shared memory which probably will be in critical section and only one process will be able to get access of shared memory due to semaphore protection. So with the help of semaphores, only one copy of shared variable will be created which will be used by every processes to update that shared variable. Let's say with a multiprocessor system, we have to print one to hundred with the help of shared memory variable, say index. If every process could access that variable then printing will be done repeatedly more than one time. So shared variable is used in multiprocessor system.

In this project, parent project will call child process to start the experiment and experiment is to signal one child process to other child process continuously. For this 2 shared memory will be created. First is the array in which the time will be noted and other one is the index variable and with that variable we can access shared array. And with the help of that we have to find time taken for a signal to pass from one process to other.

One of the important facets impacting on multiprocessor system is Latency. Latency is Time delay between input event being applied to a system and the associated output action from the system. Here, latency will be calculated by taking difference between successive elements of shared variable array. Average latency gives you approximate time for signal to be passed from one process to other. Latency is very important in any field. It gives you information about how fast the system is. Lesser the latency, faster the system will be.

In this experiment, we are going to study about how to create shared memory and how to send signal from one child to other child in ping pong pattern. Moreover we will learn how to calculate average latency, maximum latency and minimum latency.

Discussion on the methodology to solve the problem

Algorithm to find the time it take for a signal to be passed from one process to another.

1) Create a System V semaphore set with a single semaphore.
2) Create and attach the shared memory.
3) Setup a pointer to point to shared memory.
4) Setup a default signal mask which blocks SIGUSR1, the signal that we are going to send between the children.
5) Create 2 children.
6) Setup the state data, the child number and the child's pid.
7) Call SigChild() for both child to setup signal handler.
8) In parent process, send the first child to start ping pong.
9) Call second child with kill function using first child.
10) repeat 8 and 9 for number of times given in the program.
11) Delete shared memory.
12) Delete semaphore.

First of all, we will understand how to create semaphore to begin the process.

```
#define SEM_KEY 0x1234
...
struct sembuf g_lock_sembuf[1];
struct sembuf g_unlock_sembuf[1];
```

The first 2 line of code creates two array of structures of type sembuf, each with 1 element. One will be used to perform a lock semaphore and other one will be used to perform the unlock the semaphore. These structures are initialized in main function.

```
g_lock_sembuf[0].sem_num = 0;
g_lock_sembuf[0].sem_op = -1;
g_lock_sembuf[0].sem_flg = 0;
 g_unlock_sembuf[0].sem_num = 0;
g_unlock_sembuf[0].sem_op = 1;
g_unlock_sembuf[0].sem_flg = 0;
```

There are three elements to the structure, sem_num, sem_op, and sem_flg. For both structures.

1.Sem_num is initialized to 0 defining that we are operating on first semaphore in the set.
2.Sem_flg is initialized to 0 stating that no special action is to be taken here.
3.Sem_op is initialized to -1 for the lock and 1 for the unlock it.

```
if( ( g_sem_id = semget( SEM_KEY, 1, IPC_CREAT | 0666 ) ) == -1 )
{
fprintf(stderr,"semget() call failed, could not create semaphore!");
exit(1);
 }

if( semop( g_sem_id, g_unlock_sembuf, 1 ) == -1 )
{
fprintf(stderr,"semop() call failed, could not initalize semaphore!");
exit(1);
}
```

Next, the semaphore with key SEM_KEY is created using the semget() call. The permissions are set with 0666. As stated previously, immediately after creating the semaphore it must be initialize by unlocking it. The semop() call does this.

```
#define SHARED_MEMORY_ID 1234
```

Here, it is given how we define shared memory id.
Now, let us understand how to create shared memory and attach it.

```
if( (shm_id = shmget(IPC_PRIVATE, sizeof(time_stats_t), IPC_CREAT | 0666)) == -1 )
          {
fprintf(stderr,"Couldn't create shared memory segment!\n");
          exit(1);
          }
```

First of all, shmget() is used to create share memory. We have to take a global variable shm_id to get return value from the function. And if this will not be created then we get a message of ‘couldn’t create shared segment’.

```
    if( (tmp_addr = (char *)shmat(shm_id, NULL, 0)) == (char *)-1 )
          {
fprintf(stderr,"Couldn't attach to shared memory segment!\n");
          exit(1);
      }
```

Here, we have attached the created shared variable with shmat() function. And if the return valu wil be -1 then this memory is not attached and a message will be printed stating 'couldn't attach shared memory'.

```
g_time_stats = (time_stats_t *)tmp_addr;
```

A pointer is used to point to the shared memory.

Here, there will be 2 shared memory used. One to store values of latencies and other one will be to store the index and it will be used when only one will be able to access the critical section in SigHandler to access shared memory array for storing the latencies.

Now, we are going to study about default mask which blocks SIGUSR1 and this signal we are going to send between 2 children.

```
sigemptyset( &sigset );
sigaddset( &sigset, SIGUSR1 );
sigprocmask( SIG_BLOCK, &sigset, NULL );
```

sigemptyset() initializes the signal set given by set to empty, with all signals excluded from the set.
sigaddset() add signal signum from set.
sigprocmask() is used to change the signal mask, the set of currently blocked signals.

Here, we are going to study about how to create child process.

```
rtn = 1;
    for( count = 0; count < 2; count++ )
        {
        if( rtn != 0 ) {
            rtn = fork();
        } else {
            break;
        }
     }
```

If rtn==0 then it will be child process.
If rtn has other positive values then it will be parent process.

After that child will call SigChild() to Setup Sighandler and in there mainly these functions will be used.

sigfillset() initializes set to full, including all signals.
sigdelset() delete respectively signal signum from set.
sigsuspend() temporarily replaces the signal mask of the calling process with the mask given by mask and then suspends the process until delivery of a signal whose action is to invoke a signal handler or to terminate a process.

And when parent process will run it will wait for some time to start child and after that parent will signal first child to start ping pong process.

```
gettimeofday( &g_time_stats->timings[*p], NULL );
        *p=*p+1;
kill( g_time_stats->child_pid[g_child_no % 2], SIGUSR1 );
```

When SigChild will be called it will run sigsuspend signal which will wait for SigUSR1 signal and after that only it will call Sighandler function. In that critical section will be accessed and with the help of shared index variable, shared array will be filled with gettimeoftheday function. And at the that process will call other child with the help of kill function and this process will be repeated specified number of times given.

When this child processes get over, parent process start continuing when it left of its execution. This parent process will calculate average latency, maximum latency, minimum latency needed for the experiment. It can be counted with this given code written down below.

```
for( count = 0; count < (NO_OF_ITERATIONS*2)-1; count++ )
                {


            delta  = g_time_stats->timings[count+1].tv_sec
                    - g_time_stats->timings[count].tv_sec;
            delta += (g_time_stats->timings[count+1].tv_usec
                    - g_time_stats->timings[count].tv_usec)/(float)MICRO_PER_SECOND;

            total += delta;

            if( delta > maximum )
                    {
                maximum = delta;
```

```
        }

                if(delta<minimum)
                {

                        minimum=delta;
                }
        }


    printf("\nThe average signal latency was: %.6f\n", total/(float)(NO_OF_ITERATIONS*2));
    printf("The maximum signal latency was: %.6f\n", maximum );
    printf("The minimum signal latency was: %.6f\n", minimum );
```

After that shared memory will be deleted.

```
if( shmctl(shm_id,IPC_RMID,NULL) != 0 )
                {
        fprintf(stderr,"Couldn't remove the shared memory segment!\n");
        exit(1);
    }
```

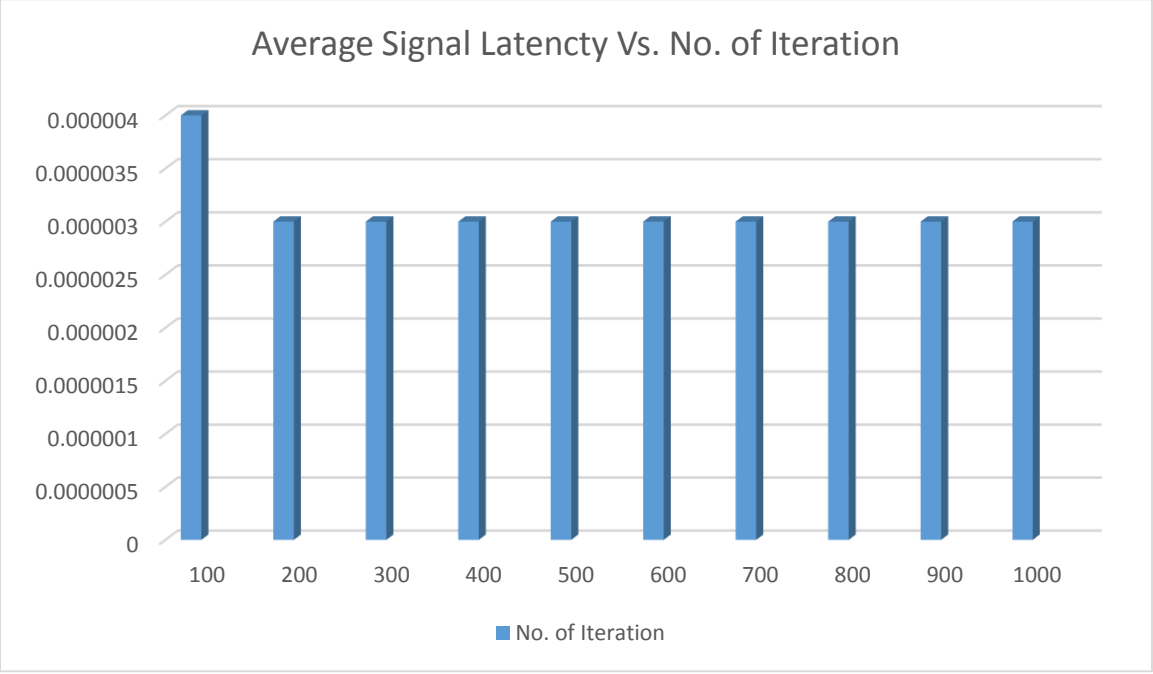
And at last, the semctl() call removes semaphore from the system.

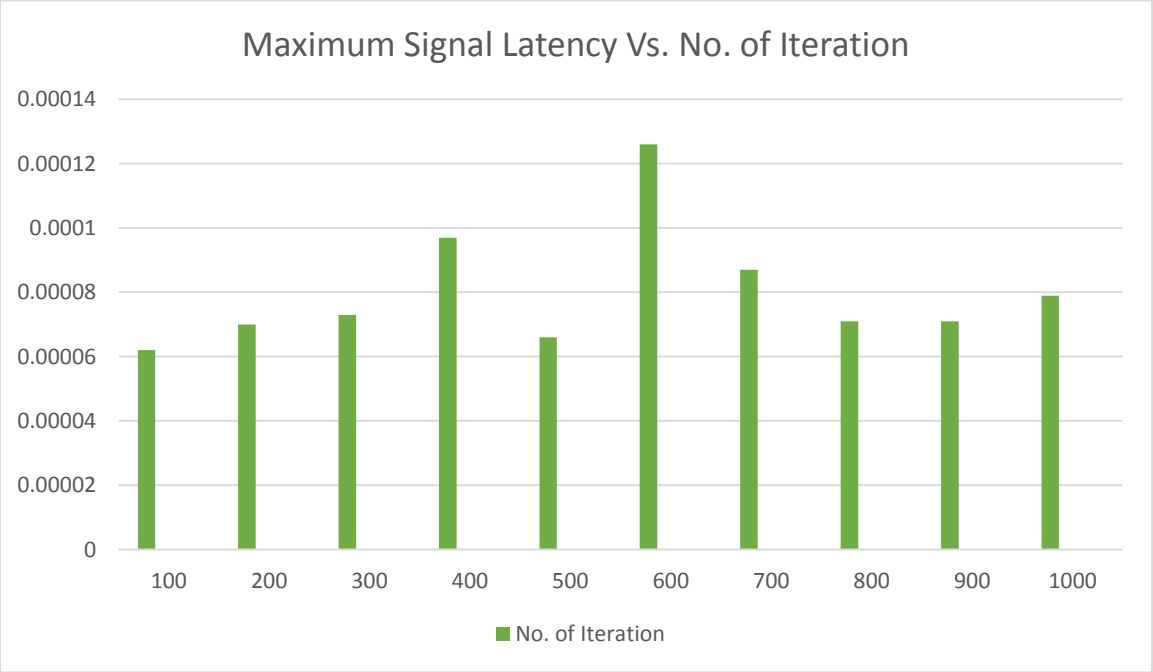
```
if( semctl( g_sem_id, 0, IPC_RMID, 0) != 0 )
{
fprintf(stderr,"Couldn't remove the semaphore!\n");
exit(1);
}
```

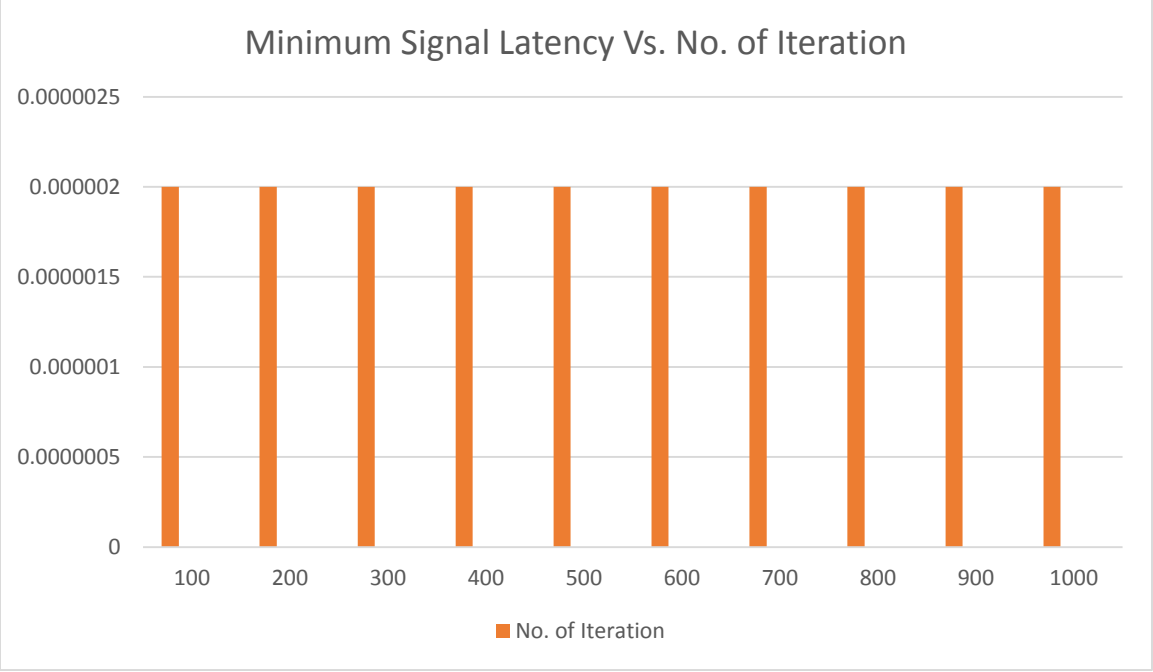
Results of running program

1. No. of Iteration Vs. The average signal latency Vs. maximum signal latency Vs. Minimum Signal Latency for a signal to be passed from one process to other process.

No of Iteration	Average Latency(Sec)	Maximum latency(Sec)	Minimum Latency(sec)
100	**0.000004**	**0.000062**	**0.000002**
200	**0.000003**	**0.000070**	**0.000002**
300	**0.000003**	**0.000073**	**0.000002**
400	**0.000003**	**0.000097**	**0.000002**
500	**0.000003**	**0.000066**	**0.000002**
600	**0.000003**	**0.000126**	**0.000002**
700	**0.000003**	**0.000087**	**0.000002**
800	**0.000003**	**0.000071**	**0.000002**
900	**0.000003**	**0.000071**	**0.000002**
1000	**0.000003**	**0.000079**	**0.000002**




Maximum Signal Latency Vs. No. of Iteration
0.00014
0.00012
0.0001
0.00008
0.00006
0.00004
0.00002
0
100
200
300
400
500
600
700
800
900
1000
No. of Iteration


Minimum Signal Latency Vs. No. of Iteration
0.0000025
0.000002
0.0000015
0.000001
0.0000005
0
100
200
300
400
500
600
700
800
900
1000
No. of Iteration

Analysis of Results

To calculate the time it take for a signal to be passed from one process to another, I did this experiment repeatedly. I tried to find out that with the increase in number of iteration, output average latency or maximum latency or minimum latency gets affected or not. With the experiments analyses I found out that if we increase or decrease the number of iteration, output average latency and minimum latency remain constant. And maximum latency will be different in microseconds due to hardware limitations.

I did this experiments with the number of iteration equals to 100, 200,500 and even for 1000. But the output average latency and minimum latency remain same.

Average latency = 3 microseconds

Minimum latency = 2 microseconds

Latency is the amount of time a message takes to traverse a system. So less the latency is more efficient the system will be.

This latencies are useful in many scenarios given below.

1. In this cutting edge world, artificial intelligence field is one of the main fields concerning about technology. In this field, Latency plays a huge role. System's speediness is society's need right now so less the latency the more powerful the system will be. For example, consider a touch cell phone. If the responsiveness of the touchscreen depends on time difference between signals sent by touching the screen and receiving that signal by system. This is one type of latency. Smaller this latency, speedy the responsiveness of the cellphone.

2. In playing computer-based musical instruments, latency greater than 100 milliseconds make it difficult for players to get the nearly instantaneous feedback that they require that time.

Here are some screenshots of output. i.e the average signal latency time(microsecond) and maximum signal latency time(microsecond) for a signal to be passed from one process to other process.

No of Iteration = 100

```
22 [volta]/home/student/dshah/dhruv/midterm> gcc midterm.c
23 [volta]/home/student/dshah/dhruv/midterm> ./a.out
Child 1 started ...
Child 2 started ...

The average signal latency was: 0.000004
The maximum signal latency was: 0.000079
The minimum signal latency was: 0.000002
24 [volta]/home/student/dshah/dhruv/midterm>
```

No of Iteration = 500

```
25 [volta]/home/student/dshah/dhruv/midterm> gcc midterm.c
26 [volta]/home/student/dshah/dhruv/midterm> ./a.out
Child 1 started ...
Child 2 started ...

The average signal latency was: 0.000003
The maximum signal latency was: 0.000066
The minimum signal latency was: 0.000002
27 [volta]/home/student/dshah/dhruv/midterm>
```

No of Iteration = 1000

```
30 [volta]/home/student/dshah/dhruv/midterm> gcc midterm.c
31 [volta]/home/student/dshah/dhruv/midterm> ./a.out
Child 1 started ...
Child 2 started ...

The average signal latency was: 0.000003
The maximum signal latency was: 0.000062
The minimum signal latency was: 0.000002
32 [volta]/home/student/dshah/dhruv/midterm>
```

Conclusion

Latency can be defined by Time delay between input event being applied to a system and the associated output action from the system. From the latency, one can find out about the speed of the system. To create a system with the least malfunction and high speed, one has to design RTOS with the least latency.

The main purpose of the experiment is to measure the time it takes for a signal to be passed from one process to another. I was able to meet the purpose of the project and came to know that the average latency of the project is 3 microseconds. From this experiment I learned about latency, how to create and attach shared memory, how to send signal from one process to other process and how to find average latency, maximum latency and minimum latency.

Appendix

```
#include <sys/time.h>        /* for gettimeofday() */
#include <unistd.h>          /* for gettimeofday(), getpid() */
#include <stdio.h>           /* for printf() */
#include <unistd.h>          /* for fork() */
#include <sys/types.h>       /* for wait() */
#include <sys/wait.h>        /* for wait() */
#include <signal.h>          /* for kill(), sigsuspend(), others */
#include <sys/ipc.h>         /* for shmget(),shmat(), and shmctl() */
#include <sys/shm.h>         /* for shmget(),shmat(), and shmctl() */
#include <stdlib.h>
#include <sys/sem.h>            /* for semget(), semop(), semctl() */

#define PROTECT

#define NO_OF_ITERATIONS     1000

#define MICRO_PER_SECOND     1000000
#define SHARED_MEMORY_ID     2282

#define SEM_KEY       0x1235
int        g_sem_id;
```

```
struct sembuf    g_lock_sembuf[1];
struct sembuf    g_unlock_sembuf[1];


int numberofDataPoints = NO_OF_ITERATIONS*2;


typedef struct {
      pid_t child_pid[2];
      struct timeval timings[];

} time_stats_t;


time_stats_t *g_time_stats;
int g_child_no;
int *p;               //for accessing timings array
int g_shared;

void SigHandler(int sig);
void SigChild(void);
 //starting main program
int main( int argc, char *argv[] )
{     int k=0;
      int rtn;
      int count;
      float delta;
```

```
 float total = 0;
 float maximum = 0;
float minimum=1;

 int shm_id;
 key_t key = SHARED_MEMORY_ID;
 char *tmp_addr;
 sigset_t sigset;



g_time_stats->timings[numberofDataPoints];



      g_lock_sembuf[0].sem_num   =  0;
      g_lock_sembuf[0].sem_op    = -1;
      g_lock_sembuf[0].sem_flg   =  0;

      g_unlock_sembuf[0].sem_num =  0;
      g_unlock_sembuf[0].sem_op  =  1;
      g_unlock_sembuf[0].sem_flg =  0;


      /*
       * Create the semaphore
       */
```

```
if( ( g_sem_id = semget( SEM_KEY, 1, IPC_CREAT | 0666 ) ) ==
-1 )

{

     fprintf(stderr,"semget() call failed, could not create
     semaphore!");

     exit(1);

}

if( semop( g_sem_id, g_unlock_sembuf, 1 ) == -1 )

{

     fprintf(stderr,"semop() call failed, could not
     initalize semaphore!");

     exit(1);

}
```

```
if( (shm_id = shmget(IPC_PRIVATE, sizeof(time_stats_t),
IPC_CREAT | 0666)) == -1 )

{

 fprintf(stderr,"Couldn't create shared memory
 segment!\n");

 exit(1);

}


 if( (tmp_addr = (char *)shmat(shm_id, NULL, 0)) ==
 (char*)-1 )

{

 fprintf(stderr,"Couldn't attach to shared memory
 segment!\n");
```

```
        exit(1);
      }

g_time_stats = (time_stats_t *)tmp_addr;



      if( ( g_shared = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT
      | 0666)) == -1 )
      {
       fprintf(stderr,"Couldn't create shared memory
       segment!\n");
       exit(1);
      }

      if( (p = (int *)shmat(g_shared, NULL, 0)) == (int *)-1 )
      {
       fprintf(stderr,"Couldn't attach to shared memory
       segment!\n");
       exit(1);
      }*p=0;



       sigemptyset( &sigset );
       sigaddset( &sigset, SIGUSR1 );
       sigprocmask( SIG_BLOCK, &sigset, NULL );
```

```
 rtn = 1;
      for( count = 0; count < 2; count++ )
{
       if( rtn != 0 )
       {
             rtn = fork();
       } else
       {
             break;
       }
 }
      if( rtn == 0 )
      {
       printf("Child %i started ...\n", count);

       g_child_no = count;
       g_time_stats->child_pid[count - 1] = getpid();
       SigChild();
       exit(0);
      }
      else
      {
       sleep(1);

           kill(g_time_stats->child_pid[0], SIGUSR1);

       wait(NULL);
```

```
 wait(NULL);


 if( shmctl(shm_id,IPC_RMID,NULL) != 0 )
      {
        fprintf(stderr,"Couldn't remove the shared memory
        segment!\n");
        exit(1);
      }



 for( count = 0; count < numberofDataPoints-1; count++ )
{



        delta  = g_time_stats->timings[count+1].tv_sec
                    - g_time_stats->timings[count].tv_sec;
        delta += (g_time_stats->timings[count+1].tv_usec
                    -  g_time_stats>timings[count].tv_usec)/(f
                       loat)MICRO_PER_SECOND;


        total += delta;
      //printf("\nDelta: %f Total: %f",delta,total);
        if( delta > maximum )
            {
              maximum = delta;
            }
         if(delta<minimum)
```

```
  {


      minimum=delta;
  }
}



 printf("\nThe average signal latency was: %.6f\n",
 total/(float)(numberofDataPoints));
printf("The maximum signal latency was: %.6f\n", maximum );
printf("The minimum signal latency was: %.6f\n", minimum );


if( shmctl(g_shared,IPC_RMID,NULL) != 0 )
{
       fprintf(stderr,"Couldn't remove the shared memory
       segment!\n");
       exit(1);
   }
/*
   * Delete the semaphore
   */
   if( semctl( g_sem_id, 0, IPC_RMID, 0) != 0 )
      {
      fprintf(stderr,"Couldn't remove the semahpore!\n");
                  exit(1);
      }
```

```
            exit(0);
      }
}

void SigHandler(int sig)
{
      //float time;

#ifdef PROTECT
           if( semop( g_sem_id, g_lock_sembuf, 1 ) == -1 )
                {
                fprintf(stderr,"semop() call failed, could not lock
                semaphore!");
                      exit(1);
                }
#endif

      /*if(getpid()==g_time_stats->child_pid[0])
      {
            printf("ping ProcessId:%d",g_time_stats->child_pid[0]);
      }
      else
      {
            printf("pong ProcessId:%d",g_time_stats->child_pid[1]);
      }*/
```

```
        gettimeofday( &g_time_stats->timings[*p], NULL );
        //time=g_time_stats->timings[*p].tv_usec;
        //printf(" Iteration=%d address=%p x=%6f\n",*p,g_time_stats-
        >timings[*p],time);
        *p=*p+1;
        kill( g_time_stats->child_pid[g_child_no % 2], SIGUSR1 );




#ifdef PROTECT
            if( semop( g_sem_id, g_unlock_sembuf, 1 ) == -1 )
                 {
                         fprintf(stderr,"semop() call failed, could not
lock semaphore!");
                         exit(1);
                 }
#endif




}


void SigChild(void)
```

```
{
	int i;

	struct sigaction act;
	sigset_t suspend_set;


	sigfillset( &suspend_set );
	sigdelset( &suspend_set, SIGUSR1 );

	act.sa_handler = &SigHandler;
	sigfillset( &act.sa_mask );
	act.sa_flags = 0;
	if( sigaction( SIGUSR1, &act, NULL ) == -1 )
	{
		fprintf(stderr, "Child could not establish signal
		handler!\n");
		exit(1);
	}

	for( i = 0; i < NO_OF_ITERATIONS; i++ )
	{
		sigsuspend( &suspend_set );
	}

	return;
}
```