Mutual Exclusion and Critical Sections in Robotic Manufacturing Systems: an example

Technical Report of the ISIS Group
at the University of Notre Dame
ISIS-97-005
October , 1997

Kevin He, Xuping Xu, and Michael Lemmon
Department of Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556

Interdisciplinary Studies of Intelligent Systems

Mutual Exclusion and Critical Sections in Robotic Manufacturing Systems: an example

Kevin He, Xuping Xu, and Michael Lemmon *
Complex Systems Simulation Facility
Dept. of Electrical Engineering
University of Notre Dame

October 11, 1997

This project considers the coordination of two robotic systems which must share a single resource in a mutually exclusive manner. Each system is assumed to have a local controller and all systems are connected over a 10BaseT local area network. The problem involves determining the protocol which ensures safe coordination of the processes in this distributed environment.

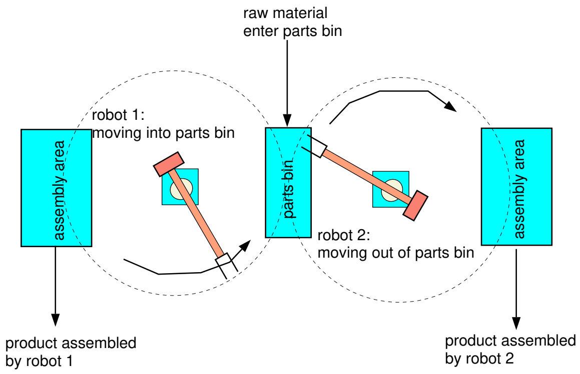

Figure 1: Manufacturing System with a Mutual Exclusion Constraint

1 Problem Statement

Consider two robots who must obtain components from a parts bin and move the components to a work area where an assembly operation takes place. The task of fetching the workpiece, transporting it to the work area, and then returning to get another workpiece is performed in a periodic manner. As illustrated in figure 1, however, it is assumed that the parts bin is common to both robots. This

*The authors gratefully acknowledge the partial finanical support of the Army Research Office (DAAH04-96-1-0285)

shared resource immediately introduces a mutual exclusion requirement on the entire system since both robots cannot enter the parts bin at the same time. The problem that is involved here, therefore, is concerned with coordination of the individual robot's motions so that the task is completed (in a timely manner) while respecting the mutual exclusion condition on the parts bin.

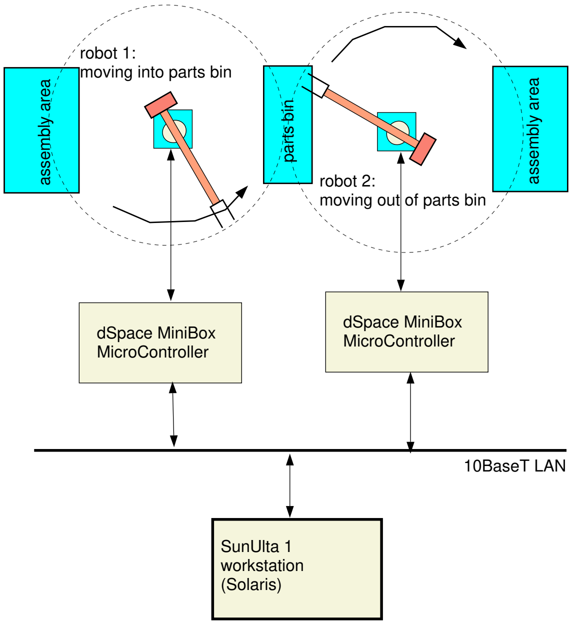

Figure 2: Manufacturing System with a Mutual Exclusion Constraint

To coordinate the action of both systems it is essential that information be passed back and forth between the robots over their LAN connection. Binary semaphores are the most common synchronization structures used in such situations. The LAN connection, however, takes about 5 milliseconds to establish over a standard TCP/IP network and during this period it is possible for both robots to obtain a lock on the semaphore. As a result, both robots will attempt to access the parts bin at the same time.

One solution to this problem, is to use a semaphore which is controlled by one of the supervisory workstations. The workstation, as shown above in figure 2 controls access to the **global semaphore** in such a way that only one robot can obtain a lock on the semaphore at a time.

2 Mutual Exclusion Protocol through a Shared Global Sempahore

To coordinate the action of both systems it is essential that information be passed back and forth between the robots over their LAN connection. Binary semaphores are the most common synchronization structures used in such situations. The LAN connection, however, takes about 5 milliseconds to establish over a standard TCP/IP network and during this period it is possible for both robots to obtain a lock on the semaphore. As a result, both robot arms will attempt to access the parts bin at the same time.

One solution to this problem, is to use a semaphore which is controlled by one of the supervisory workstations. The workstation, as shown above in Figure 2 controls access to the global semaphore in such a way that only one robot can obtain a lock on the semaphore at a time.

It is assumed that the supervisory workstation runs a parent process which initializes a binary semaphore and spawns four children at the start. Among those four children, two are to listen for requests from the two dSpace miniboxes respectively to post the global semaphore(*listening processes*). And another two are to access the semaphore for the first two child processes respectively(*semaphore processes*), and their actions are upon the signal of the parent process. When one of the listening processes hears a posting request, it will signal the parent of the request. Then the parent process will signal the corresponding semaphore process of the request. If the semaphore process has finished doing what the request orders, it will signal back the parent and the parent will signal back the corresponding listening process, and then the corresponding robot will continue to do what it has requested for. Because the listening processes, the semaphore processes and the parent process all reside on the same workstation, the posting and waiting operations for the semaphore are done in an atomic manner, thereby guaranteeing that no more than one process can lock the semaphore at a time. Figure 3 shows the implementation outline.

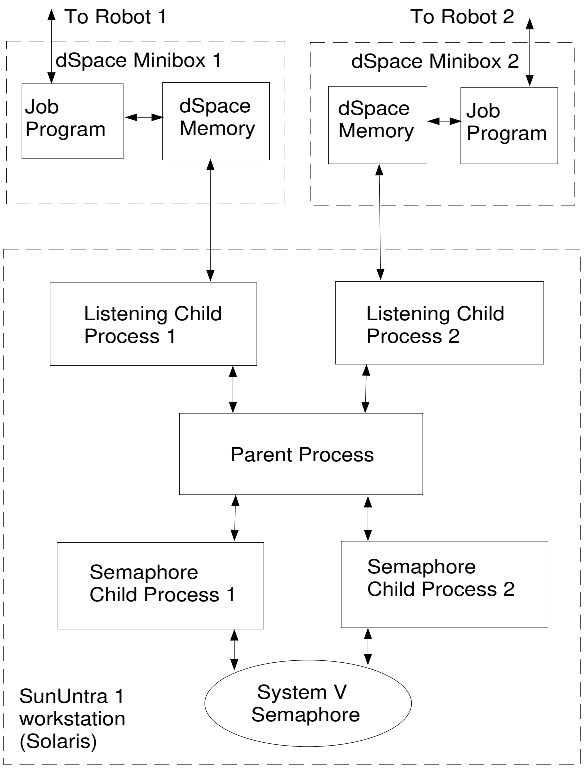

Figure 3: Coordination Protocol

If the semaphore is locked, then the listening process goes on without acknowledging the lock has been set. In this situation, the dSpace process will not proceed until it receives confirmation or the granting of its request. If the semaphore is unlocked, then the semaphore is set and the parent process signals back so that the dSpace box can let the robot arm proceed into the parts bin.

3 Lab Implementation of the MuTeX Protocol

This section discuss in detail the realization of the Unix program for the supervisory workstation whose function is outlined in the preceding section.

In the implementation, we use a variable called position to represent the location of the robotic arms. If the arm is in the parts bin, the value of position is 1, if the arm is not in the parts bin, the value of position is 2.

The sketch of the algorithm is already shown in figure 3. The parent process generates four child processes. Child processes 1 and 2 are the Listening processes communicate with arms 1 and 2, and child process 3 and 4 are Semaphor processes communicate with the Semaphore created by the parent process. We will use the semaphor to control the mutural exclusive environmant in such way that there can be only one arm in the parts bin section at any time.

Every 10 mili-second, child processes 1 and 2 read arm 1 and 2's position value from a memory location in the Dspace boxes. If the value changes, the corresponding child process will stop the arm and send a signal to the parent process, The content of this signal can be either one of the following two.

1. Position has changed from 1 to 2, i.e. arm is leaving the parts bin.
2. Position has changed from 2 to 1, i.e. arm is entering the parts bin.

After receiving the "position has changed" signal, the parent process do the following: First, it signals back to child process 1 or 2, notifing that the signal has been received. Next, it sends signal correspondingly to child process 3 or 4, i.e. if the parent process received signal from child process 1, it sends signal to child process 3, if it received signal from child process 2, it sends signal to child process 4.. The content of this signal is the same as indicated above.

Upon receiving the signal from the parent process, child precesses 3 or 4 communicates with the semaphor in the following way.

1. If arm 1 or 2 is entering the parts bin and the semaphor is locked, i.e., another arm is currrent in the parts bin, then send a "wait" signal back to the parent process.
2. If arm 1 or 2 is entering the parts bin and the semaphor is unlocked, i.e., no arm is in the parts bin now, then lock the semaphor and send a "go ahead" signal back to the parent process.
3. If arm 1 or 2 is leaving the parts bin, then unlocked the semophor and send a "go ahead" signal back to the parent process.

The parent process then transfers the signal form child process 3 or 4 to child process 1 or 2. Child processses 1 and 2 will send command to hold or move the arm accordingly.

The communication between child processes 1 and 2 and the arms are connected through two Dspace boxes. Child processes 1 and 2 write the "hold" and "go ahead" command to another memory location in the Dspace boxes. There are two Dspace processes(job programs) reading the two memory locations in the two Dspace boxes corresponding to the command from child process 1 and 2, and the position value of the arms, every 1 mili second. They change the position value if they detect the arm is entering or leaving the parts bin. In response to the "hold" and "go ahead" command, these two processes can hold and move the arms by sending a zero or positive constant input to the arms.

4 How to run the program

The executable code of the program in currently in the directory /home/maddog1/xhe/robot, to run the program, please type

```
/home/maddog1/xhe/robot/go_arm
```

```
To terminate the program, please press Ctrl-c, then type
```

```
/home/maddog1/xhe/robot/qt_arm
```

5 Program Listings

This section contains listings for the Mutex problem.

5.1 main.c

"main.c" is the main program body that runs on the Sparc station. It creates a semaphore and forks 4 child processes. The following listing also contains the source code for the interrupt service routines associated with the 4 child processes.

```
/* Main program  */

#include "proj1.h"
```

```
void main(int argc, char ** argv)
{
 int i;
 pid_t childpid;

 dsp_board="ds1102";

 semid = -1;
 changed = 0;

/* Create semaphor */

/*remove existing semaphore first*/
 if (semid < 0) {
        if( (semid = semget(SEMKEY, 1, IPC_CREAT | PERMS)) < 0)
            perror("semget error");
   }

 if(semctl(semid, 0, IPC_RMID, 0)<0)
   perror("Can't IPC_RMID");

  semid=-1;/*reset semid to -1 so that new semaphore can be obtained*/

  init_parent_proc_intrpt();

/* Fork child processes */

 for(i=1;i<=4;i++)
 {
    if((childpid=fork())<=0) break;
    chpid[i-1]=childpid;
  }

 if (childpid == 0) /* child code goes here */
 {
   ind=i;
   pid=getpid();
   if ((ind==1) || (ind==2))
   {
    init_obs_child_intrpt();
   }
  else{
   ind=ind-2;
   initialize_dspace(ind);
   printf("This is child process %d, pid=%ld\n", ind, pid);


   init_child_timer_intrpt(ind);
```

```
   setup_notify_handler(ind);
   setup_interval_timer(ind);
   }
   /* execute the rest program here */
   while(1) ;

 }
 else if (childpid > 0)/* parent code goes here */
 {
     /* execute the rest program here */
   while (1) ;
 }

}
```

5.2 proj1.h

This is the header file for main.c.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <time.h>
#include <clib.h>

#define OUR_CLOCK CLOCK_REALTIME
#define MQUEUE_IN 0

int ind;
pid_t pid;
float old;
float new;
UInt32 old32;
UInt32 new32;

unsigned int board_index;
char *dsp_board;

\subsection{childhd.c}
This source code contains the handlers  for child process 1 and child
process 2.  These codes are concerned with communication between
the Sparc station and the dSpace box.
```

```
\begin{verbatim}
#include "proj1.h"

#define OUR_CLOCK CLOCK_REALTIME



void close_and_exit(int error_code);

/* set the  interval timer for 10-millisecond intervals */
void setup_interval_timer(int index)
{
   timer_t timer1,timer2;


   struct itimerspec i;
   struct timespec resolution;
   struct sigevent timer_event;

   if (clock_getres(OUR_CLOCK, &resolution) < 0){
      perror("clock_getres error");
      exit(1);
        }

   i.it_interval.tv_sec=0;
   i.it_interval.tv_nsec=1e7;
   i.it_value=i.it_interval;

   if (index==1)
   {
    timer_event.sigev_notify=SIGEV_SIGNAL;
    timer_event.sigev_signo= SIGRTMIN;
    timer_event.sigev_value.sival_ptr = (void *)NULL;/* parameter convey
not used by now */
    if (timer_create(OUR_CLOCK, &timer_event, &timer1) < 0){
                perror("timer create error");
                exit(1);
         }
    if(timer_settime(timer1, 0 , &i, NULL) < 0){
                perror("settimer");
                exit(3);
        }
    }
    else if (index==2)
   {
    timer_event.sigev_notify=SIGEV_SIGNAL;
    timer_event.sigev_signo= SIGRTMAX;
```

```
    timer_event.sigev_value.sival_ptr = (void *)NULL;/* parameter convey
not used by now */
    if (timer_create(OUR_CLOCK, &timer_event, &timer2) < 0){
                perror("timer create error");
                exit(1);
         }
    if(timer_settime(timer2, 0 , &i, NULL) < 0){
                perror("settimer");
                exit(3);
        }
    }

}

/* Set up timer interupt signal action for child processes 1 and 2. */

void child_timer_handler(int signo, siginfo_t* info, void *context)
{
  int error;
  union sigval value;

  value.sival_int=ind;


  error = DSP_lock_board(board_index);
  if(error != DSP_NO_ERROR){
        printf("Error: can't lock board  error = %d.\n\n",error);
        close_and_exit(5);
    }

  DSP_read_dual_port_memory(board_index, MQUEUE_READ, &new32);
   DSP_unlock_board(board_index);
  new = DSP_cvt_ti_to_ieee(new32);

/*((new-old) < -EPSLON) || ((new-old) > EPSLON)*/
  if ( ((new-old) < -EPSLON) || ((new-old) > EPSLON) )
{  if (changed==0) /*yes, this is the first time the memory was changed*/
   {changed=1;

  value.sival_int = (int) (new+0.01);

  if (ind==1)
   sigqueue(getppid(), SIGUSR1, value);
  else if (ind==2)
   sigqueue(getppid(), SIGUSR2, value);
   }
  }
}
```

```
/* set up the childhandler for signal */

void init_child_timer_intrpt(int index)
{
   struct sigaction child_action;

   child_action.sa_handler = child_timer_handler;
   child_action.sa_flags = SA_SIGINFO;
   sigemptyset(&child_action.sa_mask);
   sigaddset(&child_action.sa_mask,SIGUSR1);

   if (index == 1)
      sigaction(SIGRTMIN, &child_action, NULL);
   else if (index == 2)
       sigaction(SIGRTMAX, &child_action, NULL);
}



void child_notify_handler(int signo, siginfo_t* info, void *context)
{
  /* reset the memory location in the dSpace minibox */
 int error;

 old = new;
 changed=0;

printf("%ld reset %f goofy %d\n", pid, new, ind+1);
/*if (ind==1)
kill(pid1,SIGRTMIN+1);
else if (ind==2)
kill(pid2,SIGRTMIN+2);*/


error = DSP_lock_board(board_index);
 if(error != DSP_NO_ERROR){
       printf("Error: can't lock board  error = %d.\n\n",error);
       close_and_exit(5);
   }

  error = DSP_write_dual_port_memory(board_index, MQUEUE_WRITE, DSP_cvt_ieee_to_ti((Float32) old));
  DSP_unlock_board(board_index);
  if(error != DSP_NO_ERROR)
    {
        printf("Error %d writing the DSP board's dual-port memory!\n\n",
                error);
```

```
        close_and_exit(5);
    }

}

/* set up the parent notification signal handler for SIGUSR1 */
void setup_notify_handler(int index)
{
 struct sigaction notify_action;
 notify_action.sa_handler = child_notify_handler;
 notify_action.sa_flags = SA_SIGINFO;
 sigemptyset(&notify_action.sa_mask);
 sigaddset(&notify_action.sa_mask,SIGRTMIN);
 sigaddset(&notify_action.sa_mask,SIGRTMAX);

  if(sigaction(SIGUSR1, &notify_action, NULL)<0)
   { perror("Sigaction failed for notify handler.\n");
     exit(1);}

/* else if (index == 2)
 {
  if(sigaction(SIGUSR2, &notify_action, NULL)<0)
   { perror("Sigaction failed for notify handler.\n");
     exit(1);}
  } */
}

void close_and_exit(int error_code)
{
    DSP_unregister_host_app();
    exit(error_code);
}


void initialize_dspace(int index)
{
  int error;

 if (index == 1) {
  error = DSP_net_connect(MINIBOX1);
  if(error){
   printf("Error connecting to client %s\n\n", MINIBOX1);
   exit(1);
  }

  error = error = DSP_register_host_app("proc1");
 }
 else if (index == 2) {
```

```
  error = DSP_net_connect(MINIBOX2);
  if(error){
   printf("Error connecting to client %s\n\n", MINIBOX2);
   exit(1);
  }

  error = error = DSP_register_host_app("proc2");
 }

 if(error != DSP_NO_ERROR){
        switch(error){
#ifdef UNIX
          case DSP_NOT_CONNECTED_TO_DAEMON:
            printf("\nCan't connect to DSP daemon for %s.\n","MINIBOX");
            break;
#else
          case DSP_DEVICE_DRIVER_NOT_FOUND:
            printf("\nDevice Driver not installed.\n");
            break;
          case DSP_VXD_NOT_LOADED:
            printf("\nVirtual device driver not installed.\n");
            break;
#endif
          case DSP_NO_FREE_HOST_APP_IDX:
            printf("\nNo free host application index.");
            break;
#ifdef NET
          case DSP_NET_ERROR:
            printf("\nNetwork error.");
            break;
#endif
          default:
            printf("\nDSP_register_host_app: error code %d\n",error);
            break;
        }
        exit(1);
    }

    error = DSP_board_index(dsp_board,&board_index);
    if(error != DSP_NO_ERROR)
    {
        printf("\nBoard %s not registered, error = %d.\n",
               dsp_board,error);
        close_and_exit(2);
    }

 old=2.0;
 new=2.0;
```

```
}
```

5.3 rb0.c

This is the Dspace program controlling the servo systems. This code is run by the dSpace miniboxes.

```
/* Controlling the motion of arms */
/* Code for Dspace */

#include <brtenv.h>                    /* basic real-time environment */

#define DT 1.0e-3                      /* 200 us sampling interval */


#define output1(y) ds1102_da(1,y)           /* send controller output */
/*-------------------------------------------------------------------*/
#define TMR0 0                                    /* timer0 definition */
unsigned long count0;                             /* timer0 time count */
float exec_time;                                     /* execution time */
float u1;
float u_old, u_now;
float ratio=100;

int proceed;
int in_crt_sec;

float *position= (float *) (DP_MEM_BASE);     /* arm position */
float *response= (float *) (DP_MEM_BASE+1);   /* semaphor state */

 /* *position=1  ----- arm in parts bin
            *position=2  ----- arm out of parts bin
            *response=1  ----- semaphor is locked
    *response=2  ----- semaphor is unlocked */

/* error flag for CHKERR at last dual-port memory location */
volatile int *error = (int *) (DP_MEM_BASE + DP_MEM_SIZE - 1);


/*-------------------------------------------------------------------*/
void isr_t0()                     /* timer0 interrupt service routine */
{
  begin_isr_t0(*error);                              /* overload check */
  count0 = count_timer(TMR0);             /* save current timer count */
```

```
/*  input(u); */
 ds1102_ad_start();

u_now=ds1102_ad(1);


if ( (proceed==1) && (in_crt_sec==0) )
{

 if ( (u_now>=0.058) && (u_old<0.058) )/*entering parts bin, apply for            resource*/
  {
     *position=1;
      proceed=0;
      in_crt_sec=1;
  }

}
else if ( (proceed==1) && (in_crt_sec==1) )
{

 if ( (u_now>=0.254) && (u_old<0.254) )/*leaving parts bin, release resource*/
  {
     *position=2;
      proceed=0;
      in_crt_sec=0;
  }

}

 if (proceed==0)
{
 if ( in_crt_sec==1)   /* entering parts bin */
   { if ( ((*response)>=1.9)&& ((*response)<=2.1) )
   /* semaphor locked by another process, waiting for unlock */

     proceed=0;
     else if  ( ((*response)>=0.9)&& ((*response)<=1.1) )
     /* semaphor is unlocked, go ahead */

     proceed=1;
   }

 if ( in_crt_sec==0)  /* leaving parts bin */
   { if ( ((*response)>=1.9)&& ((*response)<=2.1) )
    /* semaphor released, go ahead */

     proceed=1;
     else if  ( ((*response)>=0.9)&& ((*response)<=1.1) )
```

```
     /* semaphor not released, waiting for release */

     proceed=0;
   }
}

u_old=u_now;

if (proceed==1)
 output1(0.15);     /* move the arm */
else if(proceed==0)
 output1(0);        /* hold the arm */

  exec_time = time_elapsed(TMR0, count0);  /* calculate execution time */
  end_isr_t0();                   /* end of interrupt service routine */
}


/*-----------------------------------------------------------------------*/
void main()
{
   proceed=1;
   in_crt_sec=0;
   init();                               /* initialize hardware system */
  *error = NO_ERROR;
                          /* initialize error flag */
  *position=2;
  *response=2;            /* initialize arm position and semaphor state */



  u_now=ds1102_ad(1);
  u_old=u_now;


  start_isr_t0(DT);                 /* initialize sampling clock timer */

   while (*error == NO_ERROR);                     /* background process */

  exit(0);


}
```

5.4 sema.c

These are the semaphore handling routines.

```
#include "proj1.h"

/* Semaphor operation routine, lock and unlock */

void lock_sem()
{
    if (semid < 0) {
        if( (semid = semget(SEMKEY, 1, IPC_CREAT | PERMS)) < 0)
            perror("semget error");
    }
    if (semop(semid, &op_lock[0],2)<0)
        perror("semop lock error");
}

void unlock_sem()
{
    if(semop(semid, &op_unlock[0], 1) < 0)
        perror("semop unlock error");
}
```

5.5 obschild.c

These routines contain the handlers for child process 3 and 4 concerned with communication between dSpace box and the semaphore.

```
#include "proj1.h"



/* Define SIGALACTION of child processes 3 and 4. */
/*
If arm 1 or 2 is entering the parts bin and the semaphor is locked,
i.e., another arm is currrent in the parts bin, then send a "wait"
signal back to the parent process.

If arm 1 or 2 is entering the parts bin and the semaphor is unlocked,
i.e., no arm is in the parts bin now, then lock the semaphor and
send a "go ahead" signal back to the parent process.

If arm 1 or 2 is leaving the parts bin, then unlocked the semophor
and send a "go ahead" signal back to the parent process.
```

```
*/


void obs_handler(int signo, siginfo_t* info, void *context)
{
 union sigval value;
 value.sival_int=0;
 if ((info->si_value).sival_int==1) /* Semaphor unlocked */
 lock_sem();
 else if ((info->si_value).sival_int==2) /* Semaphor locked */
 unlock_sem();
 sigqueue(getppid(),SIGRTMIN+ind,value);
}




/* set up the observer child signal handler for SIGINT */
void init_obs_child_intrpt()
{
 struct sigaction obs_action;
 obs_action.sa_handler = obs_handler;
 obs_action.sa_flags = SA_SIGINFO;
 sigemptyset(&obs_action.sa_mask);
 sigaction(SIGUSR1, &obs_action, NULL);

}
```

5.6 parenthd.c

This code is the parent handler. It includes notification for child process 1 or 2 that their signal was received.

```
#include "proj1.h"

/*********************************************************
 *Set up and define signal handler for the parent process *
*********************************************************

/* Set up signal handler for signal from child process 1 and 2 */

void parent_handler1(int signo, siginfo_t* info, void *context)
{
 union sigval value;

/* Send notify signal back to the child process */
```

```
 printf("receive request %d from %ld\n",info->si_value.sival_int, info->si_pid);
 value.sival_int=info->si_value.sival_int;
 sigqueue( chpid[0],SIGUSR1,value);
}

void parent_handler2(int signo, siginfo_t* info, void *context)
{
 union sigval value;

/* Send notify signal back to the child process */

 printf("receive request %d from %ld\n", info->si_value.sival_int, info->si_pid);
 value.sival_int=info->si_value.sival_int;
 sigqueue( chpid[1],SIGUSR1,value);
}

/* Set up signal handler for signal from child process 1 and 2 */

void parent_handler_obs1(int signo, siginfo_t* info, void *context)
{
 union sigval value;
 value.sival_int=0;
 sigqueue( chpid[2],SIGUSR1,value);
}

void parent_handler_obs2(int signo, siginfo_t* info, void *context)
{
 union sigval value;
 value.sival_int=0;
 sigqueue( chpid[3],SIGUSR1,value);
}

/* Define the parent signal handler for all the signals from its 4 child  processes */
void init_parent_proc_intrpt()
{
 struct sigaction parent_action1,parent_action2,parent_action3,parent_action4;
 parent_action1.sa_handler = parent_handler1;
 parent_action1.sa_flags = SA_SIGINFO;
 sigemptyset(&parent_action1.sa_mask);
 sigaction(SIGUSR1, &parent_action1, NULL);

 parent_action2.sa_handler = parent_handler2;
 parent_action2.sa_flags = SA_SIGINFO;
 sigemptyset(&parent_action2.sa_mask);
 sigaction(SIGUSR2, &parent_action2, NULL);

 parent_action3.sa_handler = parent_handler_obs1;
```

```
 parent_action3.sa_flags = SA_SIGINFO;
 sigemptyset(&parent_action3.sa_mask);
 sigaction(SIGRTMIN+1, &parent_action3, NULL);

 parent_action4.sa_handler = parent_handler_obs2;
 parent_action4.sa_flags = SA_SIGINFO;
 sigemptyset(&parent_action4.sa_mask);
 sigaction(SIGRTMIN+2, &parent_action4, NULL);


}
```