IPF Machine Check (MC) error inject tool

IPF Machine Check (MC) error inject tool is used to inject MC errors from Linux. The tool is a test bed for IPF MC work flow including hardware correctable error handling, OS recoverable error handling, MC event logging, etc.

The tool includes two parts: a kernel driver and a user application sample. The driver provides interface to PAL to inject error and query error injection capabilities. The driver code is in arch/ia64/kernel/err_inject.c. The application sample (shown below) provides a combination of various errors and calls the driver's interface (sysfs interface) to inject errors or query error injection capabilities.

The tool can be used to test Intel IPF machine MC handling capabilities. It's especially useful for people who can not access hardware MC injection tool to inject error. It's also very useful to integrate with other software test suits to do stressful testing on IPF.

Below is a sample application as part of the whole tool. The sample can be used as a working test tool. Or it can be expanded to include more features. It also can be a integrated into a library or other user application to have more thorough test.

The sample application takes err.conf as error configuration input. GCC compiles the code. After you install err_inject driver, you can run this sample application to inject errors.

Errata: Itanium 2 Processors Specification Update lists some errata against the pal_mc_error_inject PAL procedure. The following err.conf has been tested on latest Montecito PAL.

err.conf:

```
#This is configuration file for err_inject_tool.
#The format of the each line is:
#cpu, loop, interval, err_type_info, err_struct_info, err_data_buffer
#where
#     cpu: logical cpu number the error will be inject in.
#     loop: times the error will be injected.
#     interval: In second. every so often one error is injected.
#     err_type_info, err_struct_info: PAL parameters.
#
#Note: All values are hex w/o or w/ 0x prefix.


#On cpu2, inject only total 0x10 errors, interval 5 seconds
#corrected, data cache, hier-2, physical addr(assigned by tool code).
#working on Montecito latest PAL.
2, 10, 5, 4101, 95

#On cpu4, inject and consume total 0x10 errors, interval 5 seconds
#corrected, data cache, hier-2, physical addr(assigned by tool code).
#working on Montecito latest PAL.
4, 10, 5, 4109, 95

#On cpu15, inject and consume total 0x10 errors, interval 5 seconds
#recoverable, DTR0, hier-2.
#working on Montecito latest PAL.
0xf, 0x10, 5, 4249, 15
```

The sample application source code:

err_injection_tool.c:

```
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, GOOD TITLE or
 * NON INFRINGEMENT.  See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 *
 * Copyright (C) 2006 Intel Co
 *    Fenghua Yu <fenghua.yu@intel.com>
 *
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <sys/shm.h>

#define MAX_FN_SIZE           256
#define MAX_BUF_SIZE          256
#define DATA_BUF_SIZE                 256
#define NR_CPUS               512
#define MAX_TASK_NUM          2048
```

```
#define MIN_INTERVAL          5       // seconds
#define       ERR_DATA_BUFFER_SIZE    3       // Three 8-byte.
#define PARA_FIELD_NUM              5
#define MASK_SIZE             (NR_CPUS/64)
#define PATH_FORMAT "/sys/devices/system/cpu/cpu%d/err_inject/"

int sched_setaffinity(pid_t pid, unsigned int len, unsigned long *mask);

int verbose;
#define vbprintf if (verbose) printf

int log_info(int cpu, const char *fmt, ...)
{
      FILE *log;
      char fn[MAX_FN_SIZE];
      char buf[MAX_BUF_SIZE];
      va_list args;

      sprintf(fn, "%d.log", cpu);
      log=fopen(fn, "a+");
      if (log==NULL) {
              perror("Error open:");
              return -1;
      }

      va_start(args, fmt);
      vprintf(fmt, args);
      memset(buf, 0, MAX_BUF_SIZE);
      vsprintf(buf, fmt, args);
      va_end(args);

      fwrite(buf, sizeof(buf), 1, log);
      fclose(log);

      return 0;
}

typedef unsigned long u64;
typedef unsigned int  u32;

typedef union err_type_info_u {
      struct {
              u64     mode            : 3,    /* 0-2 */
                      err_inj         : 3,    /* 3-5 */
                      err_sev         : 2,    /* 6-7 */
                      err_struct      : 5,    /* 8-12 */
                      struct_hier     : 3,    /* 13-15 */
                      reserved        : 48;   /* 16-63 */
      } err_type_info_u;
      u64     err_type_info;
} err_type_info_t;

typedef union err_struct_info_u {
      struct {
              u64     siv             : 1,    /* 0     */
                      c_t             : 2,    /* 1-2   */
                      cl_p            : 3,    /* 3-5   */
                      cl_id           : 3,    /* 6-8   */
                      cl_dp           : 1,    /* 9     */
                      reserved1       : 22,   /* 10-31 */
                      tiv             : 1,    /* 32    */
                      trigger         : 4,    /* 33-36 */
                      trigger_pl      : 3,    /* 37-39 */
                      reserved2       : 24;   /* 40-63 */
      } err_struct_info_cache;
      struct {
              u64     siv             : 1,    /* 0     */
                      tt              : 2,    /* 1-2   */
                      tc_tr           : 2,    /* 3-4   */
                      tr_slot         : 8,    /* 5-12  */
                      reserved1       : 19,   /* 13-31 */
                      tiv             : 1,    /* 32    */
                      trigger         : 4,    /* 33-36 */
                      trigger_pl      : 3,    /* 37-39 */
                      reserved2       : 24;   /* 40-63 */
      } err_struct_info_tlb;
      struct {
              u64     siv             : 1,    /* 0     */
                      regfile_id      : 4,    /* 1-4   */
                      reg_num         : 7,    /* 5-11  */
                      reserved1       : 20,   /* 12-31 */
                      tiv             : 1,    /* 32    */
                      trigger         : 4,    /* 33-36 */
                      trigger_pl      : 3,    /* 37-39 */
                      reserved2       : 24;   /* 40-63 */
      } err_struct_info_register;
      struct {
              u64     reserved;
      } err_struct_info_bus_processor_interconnect;
      u64     err_struct_info;
} err_struct_info_t;

typedef union err_data_buffer_u {
      struct {
              u64     trigger_addr;           /* 0-63         */
              u64     inj_addr;               /* 64-127       */
              u64     way             : 5,    /* 128-132      */
                      index           : 20,   /* 133-152      */
                                      : 39;   /* 153-191      */
      } err_data_buffer_cache;
```

```
	struct {
		u64	trigger_addr;		/* 0-63		*/
		u64	inj_addr;		/* 64-127	*/
		u64	way		: 5,	/* 128-132	*/
			index		: 20,	/* 133-152	*/
			reserved	: 39;	/* 153-191	*/
	} err_data_buffer_tlb;
	struct {
		u64	trigger_addr;		/* 0-63		*/
	} err_data_buffer_register;
	struct {
		u64	reserved;		/* 0-63		*/
	} err_data_buffer_bus_processor_interconnect;
	u64 err_data_buffer[ERR_DATA_BUFFER_SIZE];
} err_data_buffer_t;

typedef union capabilities_u {
	struct {
		u64	i		: 1,
			d		: 1,
			rv		: 1,
			tag		: 1,
			data		: 1,
			mesi		: 1,
			dp		: 1,
			reserved1	: 3,
			pa		: 1,
			va		: 1,
			wi		: 1,
			reserved2	: 20,
			trigger		: 1,
			trigger_pl	: 1,
			reserved3	: 30;
	} capabilities_cache;
	struct {
		u64	d		: 1,
			i		: 1,
			rv		: 1,
			tc		: 1,
			tr		: 1,
			reserved1	: 27,
			trigger		: 1,
			trigger_pl	: 1,
			reserved2	: 30;
	} capabilities_tlb;
	struct {
		u64	gr_b0		: 1,
			gr_b1		: 1,
			fr		: 1,
			br		: 1,
			pr		: 1,
			ar		: 1,
			cr		: 1,
			rr		: 1,
			pkr		: 1,
			dbr		: 1,
			ibr		: 1,
			pmc		: 1,
			pmd		: 1,
			reserved1	: 3,
			regnum		: 1,
			reserved2	: 15,
			trigger		: 1,
			trigger_pl	: 1,
			reserved3	: 30;
	} capabilities_register;
	struct {
		u64	reserved;
	} capabilities_bus_processor_interconnect;
} capabilities_t;

typedef struct resources_s {
	u64	ibr0		: 1,
		ibr2		: 1,
		ibr4		: 1,
		ibr6		: 1,
		dbr0		: 1,
		dbr2		: 1,
		dbr4		: 1,
		dbr6		: 1,
		reserved	: 48;
} resources_t;


long get_page_size(void)
{
	long page_size=sysconf(_SC_PAGESIZE);
	return page_size;
}

#define PAGE_SIZE (get_page_size()==-1?0x4000:get_page_size())
#define SHM_SIZE (2*PAGE_SIZE*NR_CPUS)
#define SHM_VA 0x2000000100000000

int shmid;
void *shmaddr;

int create_shm(void)
{
	key_t key;
```

```
	char fn[MAX_FN_SIZE];

	/* cpu0 is always existing */
	sprintf(fn, PATH_FORMAT, 0);
	if ((key = ftok(fn, 's')) == -1) {
		perror("ftok");
		return -1;
	}

	shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT);
	if (shmid == -1) {
		if (errno==EEXIST) {
			shmid = shmget(key, SHM_SIZE, 0);
			if (shmid == -1) {
				perror("shmget");
				return -1;
			}
		}
		else {
			perror("shmget");
			return -1;
		}
	}
	vbprintf("shmid=%d", shmid);

	/* connect to the segment: */
	shmaddr = shmat(shmid, (void *)SHM_VA, 0);
	if (shmaddr == (void*)-1) {
		perror("shmat");
		return -1;
	}

	memset(shmaddr, 0, SHM_SIZE);
	mlock(shmaddr, SHM_SIZE);

	return 0;
}

int free_shm()
{
	munlock(shmaddr, SHM_SIZE);
	  shmdt(shmaddr);
	semctl(shmid, 0, IPC_RMID);

	return 0;
}

#ifdef _SEM_SEMUN_UNDEFINED
union semun
{
	int val;
	struct semid_ds *buf;
	unsigned short int *array;
	struct seminfo *__buf;
};
#endif

u32 mode=1; /* 1: physical mode; 2: virtual mode. */
int one_lock=1;
key_t key[NR_CPUS];
int semid[NR_CPUS];

int create_sem(int cpu)
{
	union semun arg;
	char fn[MAX_FN_SIZE];
	int sid;

	sprintf(fn, PATH_FORMAT, cpu);
	sprintf(fn, "%s/%s", fn, "err_type_info");
	if ((key[cpu] = ftok(fn, 'e')) == -1) {
		perror("ftok");
		return -1;
	}

	if (semid[cpu]!=0)
		return 0;

	/* clear old semaphore */
	if ((sid = semget(key[cpu], 1, 0)) != -1)
		semctl(sid, 0, IPC_RMID);

	/* get one semaphore */
	if ((semid[cpu] = semget(key[cpu], 1, IPC_CREAT | IPC_EXCL)) == -1) {
		perror("semget");
		printf("Please remove semaphore with key=0x%lx, then run the tool.\n",
			(u64)key[cpu]);
		return -1;
	}

	vbprintf("semid[%d]=0x%lx, key[%d]=%lx\n",cpu,(u64)semid[cpu],cpu,
		(u64)key[cpu]);
	/* initialize the semaphore to 1: */
	arg.val = 1;
	if (semctl(semid[cpu], 0, SETVAL, arg) == -1) {
		perror("semctl");
		return -1;
	}

	return 0;
```

```
}

static int lock(int cpu)
{
      struct sembuf lock;

      lock.sem_num = cpu;
      lock.sem_op = 1;
      semop(semid[cpu], &lock, 1);

        return 0;
}

static int unlock(int cpu)
{
      struct sembuf unlock;

      unlock.sem_num = cpu;
      unlock.sem_op = -1;
      semop(semid[cpu], &unlock, 1);

        return 0;
}

void free_sem(int cpu)
{
      semctl(semid[cpu], 0, IPC_RMID);
}

int wr_multi(char *fn, unsigned long *data, int size)
{
      int fd;
      char buf[MAX_BUF_SIZE];
      int ret;

      if (size==1)
              sprintf(buf, "%lx", *data);
      else if (size==3)
              sprintf(buf, "%lx,%lx,%lx", data[0], data[1], data[2]);
      else {
              fprintf(stderr,"write to file with wrong size!\n");
              return -1;
      }

      fd=open(fn, O_RDWR);
      if (!fd) {
              perror("Error:");
              return -1;
      }
      ret=write(fd, buf, sizeof(buf));
      close(fd);
      return ret;
}

int wr(char *fn, unsigned long data)
{
      return wr_multi(fn, &data, 1);
}

int rd(char *fn, unsigned long *data)
{
      int fd;
      char buf[MAX_BUF_SIZE];

      fd=open(fn, O_RDONLY);
      if (fd<0) {
              perror("Error:");
              return -1;
      }
      read(fd, buf, MAX_BUF_SIZE);
      *data=strtoul(buf, NULL, 16);
      close(fd);
      return 0;
}

int rd_status(char *path, int *status)
{
      char fn[MAX_FN_SIZE];
      sprintf(fn, "%s/status", path);
      if (rd(fn, (u64*)status)<0) {
              perror("status reading error.\n");
              return -1;
      }

      return 0;
}

int rd_capabilities(char *path, u64 *capabilities)
{
      char fn[MAX_FN_SIZE];
      sprintf(fn, "%s/capabilities", path);
      if (rd(fn, capabilities)<0) {
              perror("capabilities reading error.\n");
              return -1;
      }

      return 0;
}

int rd_all(char *path)
```

```
{
      unsigned long err_type_info, err_struct_info, err_data_buffer;
      int status;
      unsigned long capabilities, resources;
      char fn[MAX_FN_SIZE];

      sprintf(fn, "%s/err_type_info", path);
      if (rd(fn, &err_type_info)<0) {
              perror("err_type_info reading error.\n");
              return -1;
      }
      printf("err_type_info=%lx\n", err_type_info);

      sprintf(fn, "%s/err_struct_info", path);
      if (rd(fn, &err_struct_info)<0) {
              perror("err_struct_info reading error.\n");
              return -1;
      }
      printf("err_struct_info=%lx\n", err_struct_info);

      sprintf(fn, "%s/err_data_buffer", path);
      if (rd(fn, &err_data_buffer)<0) {
              perror("err_data_buffer reading error.\n");
              return -1;
      }
      printf("err_data_buffer=%lx\n", err_data_buffer);

      sprintf(fn, "%s/status", path);
      if (rd("status", (u64*)&status)<0) {
              perror("status reading error.\n");
              return -1;
      }
      printf("status=%d\n", status);

      sprintf(fn, "%s/capabilities", path);
      if (rd(fn,&capabilities)<0) {
              perror("capabilities reading error.\n");
              return -1;
      }
      printf("capabilities=%lx\n", capabilities);

      sprintf(fn, "%s/resources", path);
      if (rd(fn, &resources)<0) {
              perror("resources reading error.\n");
              return -1;
      }
      printf("resources=%lx\n", resources);

      return 0;
}

int query_capabilities(char *path, err_type_info_t err_type_info,
                       u64 *capabilities)
{
      char fn[MAX_FN_SIZE];
      err_struct_info_t err_struct_info;
      err_data_buffer_t err_data_buffer;

      err_struct_info.err_struct_info=0;
      memset(err_data_buffer.err_data_buffer, -1, ERR_DATA_BUFFER_SIZE*8);

      sprintf(fn, "%s/err_type_info", path);
      wr(fn, err_type_info.err_type_info);
      sprintf(fn, "%s/err_struct_info", path);
      wr(fn, 0x0);
      sprintf(fn, "%s/err_data_buffer", path);
      wr_multi(fn, err_data_buffer.err_data_buffer, ERR_DATA_BUFFER_SIZE);

      // Fire pal_mc_error_inject procedure.
      sprintf(fn, "%s/call_start", path);
      wr(fn, mode);

      if (rd_capabilities(path, capabilities)<0)
              return -1;

      return 0;
}

int query_all_capabilities()
{
      int status;
      err_type_info_t err_type_info;
      int err_sev, err_struct, struct_hier;
      int cap=0;
      u64 capabilities;
      char path[MAX_FN_SIZE];

      err_type_info.err_type_info=0;                  // Initial
      err_type_info.err_type_info_u.mode=0;           // Query mode;
      err_type_info.err_type_info_u.err_inj=0;

      printf("All capabilities implemented in pal_mc_error_inject:\n");
      sprintf(path, PATH_FORMAT ,0);
      for (err_sev=0;err_sev<3;err_sev++)
              for (err_struct=0;err_struct<5;err_struct++)
                      for (struct_hier=0;struct_hier<5;struct_hier++)
      {
              status=-1;
              capabilities=0;
              err_type_info.err_type_info_u.err_sev=err_sev;
```

```
                err_type_info.err_type_info_u.err_struct=err_struct;
                err_type_info.err_type_info_u.struct_hier=struct_hier;

                if (query_capabilities(path, err_type_info, &capabilities)<0)
                        continue;

                if (rd_status(path, &status)<0)
                        continue;

                if (status==0) {
                        cap=1;
                        printf("For err_sev=%d, err_struct=%d, struct_hier=%d: ",
                                err_sev, err_struct, struct_hier);
                        printf("capabilities 0x%lx\n", capabilities);
                }
        }
        if (!cap) {
                printf("No capabilities supported.\n");
                return 0;
        }

        return 0;
}

int err_inject(int cpu, char *path, err_type_info_t err_type_info,
                err_struct_info_t err_struct_info,
                err_data_buffer_t err_data_buffer)
{
        int status;
        char fn[MAX_FN_SIZE];

        log_info(cpu, "err_type_info=%lx, err_struct_info=%lx, ",
                err_type_info.err_type_info,
                err_struct_info.err_struct_info);
        log_info(cpu,"err_data_buffer=[%lx,%lx,%lx]\n",
                err_data_buffer.err_data_buffer[0],
                err_data_buffer.err_data_buffer[1],
                err_data_buffer.err_data_buffer[2]);
        sprintf(fn, "%s/err_type_info", path);
        wr(fn, err_type_info.err_type_info);
        sprintf(fn, "%s/err_struct_info", path);
        wr(fn, err_struct_info.err_struct_info);
        sprintf(fn, "%s/err_data_buffer", path);
        wr_multi(fn, err_data_buffer.err_data_buffer, ERR_DATA_BUFFER_SIZE);

        // Fire pal_mc_error_inject procedure.
        sprintf(fn, "%s/call_start", path);
        wr(fn,mode);

        if (rd_status(path, &status)<0) {
                vbprintf("fail: read status\n");
                return -100;
        }

        if (status!=0) {
                log_info(cpu, "fail: status=%d\n", status);
                return status;
        }

        return status;
}

static int construct_data_buf(char *path, err_type_info_t err_type_info,
                err_struct_info_t err_struct_info,
                err_data_buffer_t *err_data_buffer,
                void *val)
{
        char fn[MAX_FN_SIZE];
        u64 virt_addr=0, phys_addr=0;

        vbprintf("val=%lx\n", (u64)val);
        memset(&err_data_buffer->err_data_buffer_cache, 0, ERR_DATA_BUFFER_SIZE*8);

        switch (err_type_info.err_type_info_u.err_struct) {
                case 1: // Cache
                        switch (err_struct_info.err_struct_info_cache.cl_id) {
                                case 1: //Virtual addr
                                        err_data_buffer->err_data_buffer_cache.inj_addr=(u64)val;
                                        break;
                                case 2: //Phys addr
                                        sprintf(fn, "%s/virtual_to_phys", path);
                                        virt_addr=(u64)val;
                                        if (wr(fn,virt_addr)<0)
                                                return -1;
                                        rd(fn, &phys_addr);
                                        err_data_buffer->err_data_buffer_cache.inj_addr=phys_addr;
                                        break;
                                default:
                                        printf("Not supported cl_id\n");
                                        break;
                        }
                        break;
                case 2: //  TLB
                        break;
                case 3: //  Register file
                        break;
                case 4: //  Bus/system interconnect
                default:
                        printf("Not supported err_struct\n");
                        break;
```

```
	}

	return 0;
}

typedef struct {
	u64 cpu;
	u64 loop;
	u64 interval;
	u64 err_type_info;
	u64 err_struct_info;
	u64 err_data_buffer[ERR_DATA_BUFFER_SIZE];
} parameters_t;

parameters_t line_para;
int para;

static int empty_data_buffer(u64 *err_data_buffer)
{
	int empty=1;
	int i;

	for (i=0;i<ERR_DATA_BUFFER_SIZE; i++)
	   if (err_data_buffer[i]!=-1)
		empty=0;

	return empty;
}

int err_inj()
{
	err_type_info_t err_type_info;
	err_struct_info_t err_struct_info;
	err_data_buffer_t err_data_buffer;
	int count;
	FILE *fp;
	unsigned long cpu, loop, interval, err_type_info_conf, err_struct_info_conf;
	u64 err_data_buffer_conf[ERR_DATA_BUFFER_SIZE];
	int num;
	int i;
	char path[MAX_FN_SIZE];
	parameters_t parameters[MAX_TASK_NUM]={};
	pid_t child_pid[MAX_TASK_NUM];
	time_t current_time;
	int status;

	if (!para) {
	    fp=fopen("err.conf", "r");
	    if (fp==NULL) {
		perror("Error open err.conf");
		return -1;
	    }

	    num=0;
	    while (!feof(fp)) {
		char buf[256];
		memset(buf,0,256);
		fgets(buf, 256, fp);
		count=sscanf(buf, "%lx, %lx, %lx, %lx, %lx, %lx, %lx, %lx\n",
				&cpu, &loop, &interval,&err_type_info_conf,
				&err_struct_info_conf,
				&err_data_buffer_conf[0],
				&err_data_buffer_conf[1],
				&err_data_buffer_conf[2]);
		if (count!=PARA_FIELD_NUM+3) {
			err_data_buffer_conf[0]=-1;
			err_data_buffer_conf[1]=-1;
			err_data_buffer_conf[2]=-1;
			count=sscanf(buf, "%lx, %lx, %lx, %lx, %lx\n",
				&cpu, &loop, &interval,&err_type_info_conf,
				&err_struct_info_conf);
			if (count!=PARA_FIELD_NUM)
				continue;
		}

		parameters[num].cpu=cpu;
		parameters[num].loop=loop;
		parameters[num].interval= interval>MIN_INTERVAL
					?interval:MIN_INTERVAL;
		parameters[num].err_type_info=err_type_info_conf;
		parameters[num].err_struct_info=err_struct_info_conf;
		memcpy(parameters[num++].err_data_buffer,
			err_data_buffer_conf,ERR_DATA_BUFFER_SIZE*8) ;

		if (num>=MAX_TASK_NUM)
			break;
	    }
	}
	else {
		parameters[0].cpu=line_para.cpu;
		parameters[0].loop=line_para.loop;
		parameters[0].interval= line_para.interval>MIN_INTERVAL
					?line_para.interval:MIN_INTERVAL;
		parameters[0].err_type_info=line_para.err_type_info;
		parameters[0].err_struct_info=line_para.err_struct_info;
		memcpy(parameters[0].err_data_buffer,
			line_para.err_data_buffer,ERR_DATA_BUFFER_SIZE*8) ;

		num=1;
	}
```

```
/* Create semaphore: If one_lock, one semaphore for all processors.
   Otherwise, one semaphore for each processor. */
if (one_lock) {
        if (create_sem(0)) {
                printf("Can not create semaphore...exit\n");
                free_sem(0);
                return -1;
        }
}
else {
        for (i=0;i<num;i++) {
           if (create_sem(parameters[i].cpu)) {
                printf("Can not create semaphore for cpu%d...exit\n",i);
                free_sem(parameters[num].cpu);
                return -1;
           }
        }
}

/* Create a shm segment which will be used to inject/consume errors on.*/
if (create_shm()==-1) {
        printf("Error to create shm...exit\n");
        return -1;
}

for (i=0;i<num;i++) {
        pid_t pid;

        current_time=time(NULL);
        log_info(parameters[i].cpu, "\nBegine at %s", ctime(&current_time));
        log_info(parameters[i].cpu, "Configurations:\n");
        log_info(parameters[i].cpu,"On cpu%ld: loop=%lx, interval=%lx(s)",
                parameters[i].cpu,
                parameters[i].loop,
                parameters[i].interval);
        log_info(parameters[i].cpu," err_type_info=%lx,err_struct_info=%lx\n",
                parameters[i].err_type_info,
                parameters[i].err_struct_info);

        sprintf(path, PATH_FORMAT, (int)parameters[i].cpu);
        err_type_info.err_type_info=parameters[i].err_type_info;
        err_struct_info.err_struct_info=parameters[i].err_struct_info;
        memcpy(err_data_buffer.err_data_buffer,
                parameters[i].err_data_buffer,
                ERR_DATA_BUFFER_SIZE*8);

        pid=fork();
        if (pid==0) {
                unsigned long mask[MASK_SIZE];
                int j, k;

                void *va1, *va2;

                /* Allocate two memory areas va1 and va2 in shm */
                va1=shmaddr+parameters[i].cpu*PAGE_SIZE;
                va2=shmaddr+parameters[i].cpu*PAGE_SIZE+PAGE_SIZE;

                vbprintf("va1=%lx, va2=%lx\n", (u64)va1, (u64)va2);
                memset(va1, 0x1, PAGE_SIZE);
                memset(va2, 0x2, PAGE_SIZE);

                if (empty_data_buffer(err_data_buffer.err_data_buffer))
                        /* If not specified yet, construct data buffer
                         * with va1
                         */
                        construct_data_buf(path, err_type_info,
                                err_struct_info, &err_data_buffer,va1);

                for (j=0;j<MASK_SIZE;j++)
                        mask[j]=0;

                cpu=parameters[i].cpu;
                k = cpu%64;
                j = cpu/64;
                mask[j] = 1UL << k;

                if (sched_setaffinity(0, MASK_SIZE*8, mask)==-1) {
                        perror("Error sched_setaffinity:");
                        return -1;
                }

                for (j=0; j<parameters[i].loop; j++) {
                        log_info(parameters[i].cpu,"Injection ");
                        log_info(parameters[i].cpu,"on cpu%ld: #%d/%ld ",

                                parameters[i].cpu,j+1, parameters[i].loop);

                        /* Hold the lock */
                        if (one_lock)
                                lock(0);
                        else
                        /* Hold lock on this cpu */
                                lock(parameters[i].cpu);

                        if ((status=err_inject(parameters[i].cpu,
                                   path, err_type_info,
                                   err_struct_info, err_data_buffer))
                                   ==0) {
                                /* consume the error for "inject only"*/
```

```
                                        memcpy(va2, va1, PAGE_SIZE);
                                        memcpy(va1, va2, PAGE_SIZE);
                                        log_info(parameters[i].cpu,
                                                "successful\n");
                                }
                                else {
                                        log_info(parameters[i].cpu,"fail:");
                                        log_info(parameters[i].cpu,
                                                "status=%d\n", status);
                                        unlock(parameters[i].cpu);
                                        break;
                                }
                                if (one_lock)
                                /* Release the lock */
                                        unlock(0);
                                /* Release lock on this cpu */
                                else
                                        unlock(parameters[i].cpu);

                                if (j < parameters[i].loop-1)
                                        sleep(parameters[i].interval);
                        }
                        current_time=time(NULL);
                        log_info(parameters[i].cpu, "Done at %s", ctime(&current_time));
                        return 0;
                }
                else if (pid<0) {
                        perror("Error fork:");
                        continue;
                }
                child_pid[i]=pid;
        }
        for (i=0;i<num;i++)
                waitpid(child_pid[i], NULL, 0);

        if (one_lock)
                free_sem(0);
        else
                for (i=0;i<num;i++)
                        free_sem(parameters[i].cpu);

        printf("All done.\n");

        return 0;
}

void help()
{
        printf("err_inject_tool:\n");
        printf("\t-q: query all capabilities. default: off\n");
        printf("\t-m: procedure mode. 1: physical 2: virtual. default: 1\n");
        printf("\t-i: inject errors. default: off\n");
        printf("\t-l: one lock per cpu. default: one lock for all\n");
        printf("\t-e: error parameters:\n");
        printf("\t\tcpu,loop,interval,err_type_info,err_struct_info[,err_data_buffer[0],err_data_buffer[1],err_data_bu
        printf("\t\t   cpu: logical cpu number the error will be inject in.\n");
        printf("\t\t   loop: times the error will be injected.\n");
        printf("\t\t   interval: In second. every so often one error is injected.\n");
        printf("\t\t   err_type_info, err_struct_info: PAL parameters.\n");
        printf("\t\t   err_data_buffer: PAL parameter. Optional. If not present,\n");
        printf("\t\t                    it's constructed by tool automatically. Be\n");
        printf("\t\t                    careful to provide err_data_buffer and make\n");
        printf("\t\t                    sure it's working with the environment.\n");
        printf("\t    Note:no space between error parameters.\n");
        printf("\t    default: Take error parameters from err.conf instead of command line.\n");
        printf("\t-v: verbose. default: off\n");
        printf("\t-h: help\n\n");
        printf("The tool will take err.conf file as ");
        printf("input to inject single or multiple errors ");
        printf("on one or multiple cpus in parallel.\n");
}

int main(int argc, char **argv)
{
        char c;
        int do_err_inj=0;
        int do_query_all=0;
        int count;
        u32 m;

        /* Default one lock for all cpu's */
        one_lock=1;
        while ((c = getopt(argc, argv, "m:iqvhle:")) != EOF)
                switch (c) {
                        case 'm':       /* Procedure mode. 1: phys 2: virt */
                                count=sscanf(optarg, "%x", &m);
                                if (count!=1 || (m!=1 && m!=2)) {
                                        printf("Wrong mode number.\n");
                                        help();
                                        return -1;
                                }
                                mode=m;
                                break;
                        case 'i':       /* Inject errors */
                                do_err_inj=1;
                                break;
                        case 'q':       /* Query */
                                do_query_all=1;
                                break;
                        case 'v':       /* Verbose */
```

```
                                verbose=1;
                                break;
                        case 'l':       /* One lock per cpu */
                                one_lock=0;
                                break;
                        case 'e':       /* error arguments */
                                /* Take parameters:
                                 * #cpu, loop, interval, err_type_info, err_struct_info[, err_data_buffer]
                                 * err_data_buffer is optional. Recommend not to specify
                                 * err_data_buffer. Better to use tool to generate it.
                                 */
                                count=sscanf(optarg,
                                        "%lx, %lx, %lx, %lx, %lx, %lx, %lx, %lx\n",
                                        &line_para.cpu,
                                        &line_para.loop,
                                        &line_para.interval,
                                        &line_para.err_type_info,
                                        &line_para.err_struct_info,
                                        &line_para.err_data_buffer[0],
                                        &line_para.err_data_buffer[1],
                                        &line_para.err_data_buffer[2]);
                                if (count!=PARA_FIELD_NUM+3) {
                                    line_para.err_data_buffer[0]=-1,
                                    line_para.err_data_buffer[1]=-1,
                                    line_para.err_data_buffer[2]=-1;
                                    count=sscanf(optarg, "%lx, %lx, %lx, %lx, %lx\n",
                                                &line_para.cpu,
                                                &line_para.loop,
                                                &line_para.interval,
                                                &line_para.err_type_info,
                                                &line_para.err_struct_info);
                                    if (count!=PARA_FIELD_NUM) {
                                        printf("Wrong error arguments.\n");
                                        help();
                                        return -1;
                                    }
                                }
                                para=1;
                                break;
                        continue;
                                break;
                        case 'h':
                                help();
                                return 0;
                        default:
                                break;
                }

        if (do_query_all)
                query_all_capabilities();
        if (do_err_inj)
                err_inj();

        if (!do_query_all &&  !do_err_inj)
                help();

        return 0;
}
```