**Roll no : 19MX218 (SELVA BHARATHI S)**

**III – MCA – OS Lab - Unix System Calls Practice Exercise**

Unix system calls can be loosely grouped into the following three main categories:

* File management
* Process management
* Error handling

**HS1 Problems**

**File management system calls**

1. open()

Syntax:  int open(char \*filename, int mode, int permissions)

Example: fd =  open(“text.data”, O\_CREAT | O\_RDWR, 0600)

Note: open( ) allows to open or create a file for reading and/or writing. Filename is the absolute or relative pathname and mode is a bit wise ORing of read/write flags together with zero or more miscellaneous flags. permissions is a number that encodes the value of the file permission flags, and should be supplied to when a file is being created. Main flags are O\_RDONLY (open for read only), O\_RDWR (open for write and read), O\_WRONLY (open for write only).    O\_APPEND, O\_CREAT, O\_EXCL are miscellaneous flags. open () returns a  non negative file descriptor if successful; otherwise it returns –1.

2. read()

Syntax:  int read(int fd, char \*buf, int count)

Example: charsRead = read(fd, buffer, BUFFER\_ASIZE);

Note: read() copies count number of bytes from a file referenced by a file descriptor fd into a buffer  buf. If successful it returns the number of bytes that it read; otherwise, it returns –1.

3.write()

Syntax:  int write(int fd, char \*buf, int count)

Example: charsWritten = write(fd, buffer, BUFFERSIZE);

Note: write() copies count bytes from a buffer to a file referenced by the file descriptor. If successful it returns the number of byes that were written; otherwise, it returns –1.

4. close()

Syntax:  int close (int fd)

Note: close() frees the file descriptor. If successful it returns 0; otherwise, it returns –1.

5. lseek()

Syntax:  long lseek( int fd,long offset, int mode)

Example: currentOffset = lseek(fd, 0, SEEK\_CUR);

Note:  lseek() allows to change a file descriptor's current file position. fd is the file descriptor, offset is a long integer, and mode describes how offset should be interpreted.

SEEK\_SET    offset is relative to the start of the file

SEEK\_CUR   offset is relative to the current file position

SEEK\_END   offset is relative to the end of file

6. unlink()

Syntax:  int unlink(const char\*filename)

Note: unlink() removes the hard link from the name filename to its file. If filename is the last link to the file, the file’s resources are deallocated. If successful, unlink() returns zero; otherwise, it returns  –1.

**Exercise: 1. Copy the content of one file to other.**

**2. Use lseek() to copy different parts(initial, middle and last) of the file to**

**other.**

**// copy it to another file using lseek**

**#include <stdio.h>**

**#include <unistd.h>**

**#include <sys/types.h>**

**#include <fcntl.h>**

**void func(char arr[], int n)**

**{**

**// Open the file for READ only.**

**int f\_write = open("start.txt", O\_RDONLY);**

**// Open the file for WRITE and READ only.**

**int f\_read = open("end.txt", O\_WRONLY);**

**int count = 0;**

**while (read(f\_write, arr, 1))**

**{**

**// to write the 1st byte of the input file in**

**// the output file**

**if (count < n)**

**{**

**// SEEK\_CUR specifies that**

**// the offset provided is relative to the**

**// current file position**

**lseek (f\_write, n, SEEK\_CUR);**

**write (f\_read, arr, 1);**

**count = n;**

**}**

**// After the PART BY PART (now taking the alternate**

**// PART)**

**else**

**{**

**count = (2\*n);**

**lseek(f\_write, count, SEEK\_CUR);**

**write(f\_read, arr, 1);**

**}**

**}**

**close(f\_write);**

**close(f\_read);**

**}**

**// Driver code**

**int main()**

**{**

**char arr[100];**

**int n;**

**n = 5;**

**// Calling for the function**

**func(arr, n);**

**return 0;**

**}**

**output :**

**COPIED**

7. fork()

Syntax:  int fork( )

Note: fork() causes a process to duplicate. If it succeeds, it returns the PID of the child to the parent process, and returns 0 to the child process. If it fails, it returns –1 to the parent process and no child is created.

main()

{

  /\* common code \*/

  pid = fork()

  if (pid == -1)

   {

     /\* error \*/

   }

   else if (pid != 0)

    {

     /\* parent code \*/

     }

   else

    {

      /\* child code \*/

    }

   /\* optional common termination part of parent and child \*/

} /\* end of main() \*/

8. getpid() and getppid()

Syntax: int getpid(void)

             int getppid(void)

Note:  getpid() and getppid() return a process’ ID and parent process’ ID numbers, respectively. They always succeed.

**Exercise: 3. Write a  program to create a child process. Display the process IDs of  the process, parent and child(if any) in both the parent and child processes.**

**SOLUTION :**

**int main()**

**{**

**int i;**

**printf("hello before fork \n");**

**printf("i : %d\n",i);**

**i=fork();**

**printf("\n");**

**if(i==0)**

**{**

**printf("Child has started\n\n");**

**printf("child printing first time \n");**

**printf("getpid : %d getppid : %d \n",getpid(),getppid());**

**sleep(5);**

**printf("\nchild printing second time \n");**

**printf("getpid : %d getppid : %d \n",getpid(),getppid());**

**}**

**else**

**{**

**printf("parent has started\n");**

**printf("getpid : %d getppid : %d \n",getpid(),getppid());**

**printf("\n");**

**}**

**printf("Hi after fork i : %d\n",i);**

**return 0;**

**}**

**Output:**

$ ./a.out

hello before fork

i : 134514088

Child has started

child printing first time

getpid : 8354 getppid : 8353

parent has started

getpid : 8353 getppid : 5656

Hi after fork i : 8354

$child printing second time

getpid : 8354 getppid : 1

Hi after fork i : 0

**4. Create a orphan process (parent dies before child – adopted by “init”**

**process) and display the PID of parent  of child before and after it becomes orphan. Use sleep(n) in the child to delay the termination.**

**SOLUTION :**

**#include<stdio.h>**

**#include <sys/types.h>**

**#include <unistd.h>**

**int main()**

**{**

**int pid = fork();**

**if (pid > 0)**

**printf("in parent process");**

**else if (pid == 0)**

**{**

**sleep(30);**

**printf("in child process");**

**}**

**return 0;**

**}**

Output : **in parent process**

**In child process**

9. exit()

Syntax:  int exit (int status)

Note: exit() closes all of a process’s file descriptor, deallocated code, data and stack, and then terminated the process.  When a child process terminates, it sends its parent a SIGCHLD signal and waits for its termination code status to be accepted.

10. wait()

Syntax:  int wait ( int \*status)

Note: wait() causes a  process to suspend until one of its children terminates. A successful call to wait() returns the PID of  the child that terminated and places a status code into status.

**Exercise: 5. Create a zombie (defunct) child process (a child with exit() call , but no corresponding  wait() in the sleeping parent) and allow the init process to adopt it  (after parent terminates). Run the process as background process and run “ps” command.**

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include <fcntl.h>**

**#include <stdio.h>**

**#include <unistd.h>**

**int main( void)**

**{**

**int pid;**

**pid= fork();**

**if (pid == 0) //we are in the child**

**exit(0);**

**sleep(1);**

**execlp(“ps”, “ps”, “-la”, NULL);**

**}**

**6. Modify the above program to include wait(&status) in the parent and to display the exit return code(left most byte of status) of the child.**

**#include <sys/types.h>**

**#include <sys/wait.h>**

**pid\_t wait(int \*status);**

**int status;**

**pid\_t fork\_return;**

**fork\_return = fork();**

**if (fork\_return == 0) /\* child process \*/**

**{**

**printf("\n I'm the child!");**

**exit(0);**

**}**

**else if (fork\_return > 0) /\* parent process \*/**

**{**

**wait(&status);**

**printf("\n I'm the parent!");**

**if (WIFEXITED(status))**

**printf("\n Child returned: %d\n", WEXITSTATUS(status));**

**}**

11.exec()

Syntax:  int execl( char\* path, char\* arg0, char\*arg1,…… ,char\* argn, NULL)

Note: it replaces the calling process’s code, data, and stack from the executable whose path name is given.

**Exercise 7: Replace a child process by an executable file (say, “ls” –list command).**

Replace the current process with myprog as if a user had typed:

myprog ARG1 ARG2

at the shell:

#include <stddef.h>

#include <process.h>

execl( "myprog", "myprog", "ARG1", "ARG2", NULL );

In this example, myprog will be found if it exists in the current working directory.

12.nice()

Syntax:  int nice(int delta)

Note: nice() adds delta to a process’ current priority value(-20 to 19: only a superuser and kernel processes can have a negative priority value and login shells start with priority 0). If succeeds, it returns the new nice value; otherwise it returns –1.

Exercise : Run two processes (dumping n strings to the screen), in a concurrent

                manner(foreground & background / parent & child). Let the outputs of two be

                interleaved.

                Modify one of them and include a nice(n) command, so that it runs with a

                lower priority. Not the change  in the order of output.

**Signals**

Programs must sometimes deal with unexpected or unpredictable events, such as any of the following:

* a floating error
* a power failure
* an alarm clock “ring”
* the death of a child process
* a termination request from a user(^C)
* a suspend request from a user(^Z)

These kinds of events are sometimes called interrupts, since they must interrupt the regular flow of a program in order to be processed. When UNIX recognizes that such an event has occurred, it sends the corresponding process a signal. There is a unique, numbered signal for each possible event. The kernel isn’t the only one that can send a signal; any process can send any other process a signal, as long as it has permission. By means of a special piece of code called a signal handler, a programmer may arrange for a particular signal to be ignored or to be processed.

13. alarm()

Syntax:  unsigned int alarm( unsigned int count)

Note: alarm() instructs the kernel to send the SIGALRM signal to the calling process after count seconds. The default handler for this signal displays the message “Alarm clock” and terminates the process.

**Exercise 8: Write a program that uses alarm(n), together with a forever loop to display a message  or a pause() call, which waits for a signal.**

void display\_message(int s);

void display\_message(int s) {

printf("copyit: Still working...\n" );

alarm(1);

signal(SIGALRM, display\_message);

}

while(1)

{

signal(SIGALRM, display\_message);

alarm(1);

}

14. signal()

Syntax:  void (\*signal (int sigCode, void (\*func()))) ( )

Note: signal allows a process to specify the action that it will take when a particular signal is received. It can be used to override the default action.

**Exercise 9: Modify the alarm() exercise to override the default alarm handler using signal(). Include a alarmHandler () to display a message and set a global variable to end the forever loop of main().**

An example program :

**#include<stdio.h>**

**#include<signal.h>**

**#include<unistd.h>**

**void sig\_handler(int signo)**

**{**

**if (signo == SIGINT)**

**printf("received SIGINT\n");**

**}**

**int main(void)**

**{**

**if (signal(SIGINT, sig\_handler) == SIG\_ERR)**

**printf("\ncan't catch SIGINT\n");**

**// A long long wait so that we can easily issue a signal to this process**

**while(1)**

**sleep(1);**

**return 0;**

**}**

Press Ctrl + C when the program is run. You will get the output,

received SIGINT

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <wait.h>**

**#include <signal.h>**

**void handler( int );**

**int i = 1, j = 1;**

**int main( void )**

**{**

**if( fork() != 0 )**

**{**

**while( 1 )**

**{**

**signal( SIGALRM, handler );**

**if( i == 1 )**

**{**

**alarm( 5 );**

**i--;**

**}**

**printf( "[PARENT] Serving HTTP Session 2\n" );fflush( stdout );**

**pause();**

**}**

**}**

**else**

**{**

**while( 1 )**

**{**

**signal( SIGALRM, handler );**

**if( j == 1 )**

**{**

**alarm( 5 );**

**j--;**

**}**

**printf( "[CHILD] Serving HTTP Session 1\n" );fflush( stdout );**

**pause();**

**}**

**}**

**exit( EXIT\_SUCCESS );**

**}**

**void handler( int SIG )**

**{**

**printf( "[HANDLER] Signal recieved!\n" );fflush( stdout );**

**if( i == 0 )**

**i++;**

**if( j == 0 )**

**j++;**

**}**

15. kill()

Syntax:  int kill (int pid, int sigCode)

Note: kill ( ) sends the signal with the value sigCode to the process with PID pid. 

**Exercise: 10. Use kill(), SIGCHLD signal(terminating signal from a child) and SIGINT**

                    signal (interrupt to terminate the process) to limit the amount of time that a

                    command takes to execute. Note: command can be passed as command ine

                    argument and can be used to replace a child process. Use childHandler() to

                    receive the child’s exit code and to display a message “ child with PID x is

                    terminated within the specified time.

                11. Create a parent and two child processes. The child processes should include

                    a forever loop to display a message including their PID. The parent process

                    suspends one of the children for some period p1, resumes it and kills both the

                    children after a period p2. Use SIGSTOP and SIGCONT signals to suspend

                    and resume the child process..

**include <signal.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <sys/types.h>**

**#include <unistd.h>**

**void sighup();**

**void sigint();**

**void sigquit();**

**void main()**

**{**

**int pid;**

**if ((pid = fork()) < 0) {**

**perror("fork");**

**exit(1);**

**}**

**if (pid == 0) { /\* child \*/**

**signal(SIGHUP, sighup);**

**signal(SIGINT, sigint);**

**signal(SIGQUIT, sigquit);**

**for (;;) ;**

**}**

**else**

**{**

**printf("\nPARENT: sending SIGHUP\n\n");**

**kill(pid, SIGHUP);**

**sleep(3);**

**printf("\nPARENT: sending SIGINT\n\n");**

**kill(pid, SIGINT);**

**sleep(3);**

**printf("\nPARENT: sending SIGQUIT\n\n");**

**kill(pid, SIGQUIT);**

**sleep(3);**

**}**

**}**

**void sighup()**

**{**

**signal(SIGHUP, sighup); /\* reset signal \*/**

**printf("CHILD: I have received a SIGHUP\n");**

**}**

**void sigint()**

**{**

**signal(SIGINT, sigint);**

**printf("CHILD: I have received a SIGINT\n");**

**}**

**void sigquit()**

**{**

**printf("My DADDY has Killed me!!!\n");**

**exit(0);**

**}**

**Error Handling**

16. perror()

Syntax:  void peeor(char \*str)

Note: errno, a global variable, hold the numeric code of the last system call error and perror() displays the string str, followed by a colon, followed by a description of the last system call error.

**Exercise 12: Open a non-existent file to cause an error(no creation) and display the corresponding error number and description.**

**#include <ftw.h>**

**#include <unistd.h>**

**int unlink\_cb(const char \*fpath, const struct stat \*sb, int typeflag, struct FTW \*ftwbuf)**

**{**

**int rv = remove(fpath);**

**if (rv)**

**perror(fpath);**

**return rv;**

**}**

**int rmrf(char \*path)**

**{**

**return nftw(path, unlink\_cb, 64, FTW\_DEPTH | FTW\_PHYS);**

**}**

**HS2 Problems**

Problems:

1. Implement in C the following UNIX commands using System calls:

Cat, ls and mv

Program :

**#include<sys/types.h>**

**#include<sys/stat.h>**

**#include<stdio.h>**

**#include<fcntl.h>**

**main( int argc,char \*argv[3] )**

**{**

**int fd,i;**

**char buf[2];**

**fd=open(argv[1],O\_RDONLY,0777);**

**if(fd==-argc)**

**{**

**printf("file open error");**

**}**

**else**

**{**

**while((i=read(fd,buf,1))>0)**

**{**

**printf("%c",buf[0]);**

**}**

**close(fd);**

}

}

**Output**

selvaBharathi@ubuntu:~$gcc –o prgcat.out prgcat.c

selvaBharathi@ubuntu:~$cat > ff

hello

hai

selvaBharathi@ubuntu:~$./prgcat.out ff

hello

hai

**ls**

**#include <sys/types.h>**

**#include <sys/dir.h>**

**#include <sys/param.h>**

**#include <stdio.h>**

**#define FALSE 0**

**#define TRUE 1**

**extern int alphasort();**

**char pathname[MAXPATHLEN];**

**main() {**

**int count,i;**

**struct dirent \*\*files;**

**int file\_select();**

**if (getwd(pathname) == NULL )**

**{**

**printf("Error getting pathn");**

**exit(0);**

**}**

**printf("Current Working Directory = %sn",pathname);**

**count = scandir(pathname, &files, file\_select, alphasort);**

**if (count <= 0)**

**{**

**printf("No files in this directoryn");**

**exit(0);**

**}**

**printf("Number of files = %dn",count);**

**for (i=1;i<count 1; i)**

**printf("%s \n",files[i-1]->d\_name);**

**}**

**int file\_select(struct direct \*entry)**

**{**

**if ((strcmp(entry->d\_name, ".") == 0) ||(strcmp(entry->d\_name, "..") == 0))**

**return (FALSE);**

**else**

**return (TRUE);**

**}**

**Output**

SelvaBharathi@ubuntu:~$ gcc list.c

SelvaBharathi@ubuntu:~$ ./a.out

Current working directory=/home/student/

Number of files=57

**mv**

**#include<sys/types.h>**

**#include<sys/stat.h>**

**#include<stdio.h>**

**#include<fcntl.h>**

**main( int argc,char \*argv[] )**

**{**

**int i,fd1,fd2;**

**char \*file1,\*file2,buf[2];**

**file1=argv[1];**

**file2=argv[2];**

**printf("file1=%s file2=%s",file1,file2);**

**fd1=open(file1,O\_RDONLY,0777);**

**fd2=creat(file2,0777);**

**while(i=read(fd1,buf,1)>0)**

**write(fd2,buf,1);**

**remove(file1);**

**close(fd1);**

**close(fd2);**

**}**

**Output**

selvaBharathi@ubuntu:~$gcc –o mvp.out mvp.c

selvaBharathi@ubuntu:~$cat > ff

hello

hai

selvaBharathi@ubuntu:~$./mvp.out ff ff1

selvaBharathi@ubuntu:~$cat ff

cat:ff:No such file or directory

selvaBharathi@ubuntu:~$cat ff1

hello

hai

2. Determine the size of a file using the lseek command. Once you found out the size, calculate the number of blocks assigned for the file. Compare these results with the similar results obtained when using the function stat.

**#include <unistd.h>**

**#include <stdio.h>**

**#include <fcntl.h>**

**int main(int ac, char \*av[])**

**{**

**if ( ac < 2 ) return 0;**

**int fd = open(av[1],O\_RDONLY);**

**int size = lseek(fd, 0, SEEK\_END);**

**printf("%d\n", size);**

**close(fd);**

**return 0;**

**}**

**Using stat**

**#include <sys/stat.h>**

**#include <stdio.h>**

**int main(int ac, char \*av[])**

**{**

**if ( ac < 2 ) return 0;**

**struct stat stbuf;**

**stat( av[1], &stbuf);**

**printf ("%lld\n", stbuf.st\_size);**

**return 0;**

**}**

3. Write a C program that finds a file in a file-tree starting from a given directory. The name of the file for which we are searching for, as well as the name of the starting directory should be read from the command line. Optionally, the name of the file can be specified as a pattern using the ‘\*’ character.

**#include <stdio.h>**

**#include <dirent.h>**

**int main(void)**

**{**

**struct dirent \*de; // Pointer for directory entry**

**// opendir() returns a pointer of DIR type.**

**DIR \*dr = opendir(".");**

**if (dr == NULL) // opendir returns NULL if couldn't open directory**

**{**

**printf("Could not open current directory" );**

**return 0;**

**}**

**// Refer http://pubs.opengroup.org/onlinepubs/7990989775/xsh/readdir.html**

**// for readdir()**

**while ((de = readdir(dr)) != NULL)**

**printf("%s\n", de->d\_name);**

**closedir(dr);**

**return 0;**

**}**

4. Write a C program that deletes a directory with all its subfolders. The name of the directory should be read from the command line.

**#include <ftw.h>**

**#include <unistd.h>**

**int unlink\_cb(const char \*fpath, const struct stat \*sb, int typeflag, struct FTW \*ftwbuf)**

**{**

**int rv = remove(fpath);**

**if (rv)**

**perror(fpath);**

**return rv;**

**}**

**int rmrf(char \*path)**

**{**

**return nftw(path, unlink\_cb, 64, FTW\_DEPTH | FTW\_PHYS);**

**}**

5. Write a program in C that creates a child process, waits for the termination

of the child and lists its PID, together with the state in which the process was terminated (in decimal and hexadecimal).

**#include <stdio.h>**

**#include <signal.h>**

**#include <sys/types.h>**

**#include <unistd.h>**

**int main()**

**{**

**int pid, state;**

**printf(“before the fork\n”);**

**if ((pid =fork() != 0))**

**{**

**printf(“parent\n”);**

**wait(&state);**

**}**

**else**

**{**

**printf(“child\n”);**

**execl(“./child”, “child”, 0);**

**perror(“Error exec”);**

**}**

**printf(“after fork state = %d\n”,state);**

**printf(“PID child = %d terminated\n”, pid);**

**}**

6. In a C program, print the address of the variable and enter into a long loop (say using while(1)).

• Start three to four processes of the same program and observe the printed address values.

• Show how two processes which are members of the relationship parent-child are concurrent from execution point of view, initially the child is copy of the parent, but every process has its own data.

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include <fcntl.h>**

**#include <stdio.h>**

**#include <unistd.h>tty**

**int main( void)**

**{**

**int pid;**

**int n=0;**

**int status;**

**pid= fork();**

**while (n<10)**

**{**

**if (pid == 0)**

**{**

**printf(“this is the son working: %d\n”, n);**

**n = n + 2;**

**sleep(1);**

**}**

**else**

**{**

**printf(“this is the parent working: %d\n”, n);**

**n = n +3;**

**sleep(1);**

**}**

**}**

**}**

7. Write two programs: one called client.c , the other called server.c. The client program lists a prompter and reads from the keyboard two integers and one of the characters ’+’ or ’-’. The read information is transmitted with the help of the system call excel to a child process, which executes the server code. After the child (server) process fiishes the operation, it transmits the result to parent process (client) with the help of the system call exit. The client process prints the result on the screen and also reprints the prompter, ready for a new reading.

#include <sys/types.h>

**#include <sys/stat.h>**

**#include <fcntl.h>**

**#include <stdio.h>**

**#include <unistd.h>**

**#include <sys/types.h>**

**#include <sys/wait.h>**

**int main()**

**{**

**int pid;**

**char op1, op2;**

**char sign;**

**int status;**

**printf(“>”);**

**scanf(“%c %c %c”, &op1, &sign, &op2);**

**pid = fork();**

**if (pid == 0)**

**{**

**if (execl(“./child.exe”, &op1, &op2, &sign, NULL)<0)**

**perror(“eroare”);**

**}**

**else if (pid>0)**

**{**

**waitpid(pid, &status, 0);**

**printf(“result %d\n”, WEXITSTATUS(status));**

**}**

**exit(0);**

**}**

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include <fcntl.h>**

**#include <stdio.h>**

**#include <unistd.h>**

**int main( int argc, char \*\*argv)**

**{**

**int a;**

**int b;**

**int res=0;**

**a = \*argv[0] – 48;**

**b = \*argv[1] – 48;**

**printf (“\n %d %d “, a, b );**

**if ( \*argv[2] == ‘+’)**

**res = a+b;**

**else res = a-b;**

**return res;**

**}**

8. Test the source code below:

f or (i= 1; I ≤ 10; I + +)

{

fork();

printf(“The process with the PID=%d”,getpid());

}

In the next phase, modify the code, such as after all created processes have finished execution, in a file process management.txt the total number of created processes should be stored.

9. Try this code:

#include "types.h"

#include "stat.h"

#include "user.h"

void periodic();

int

main(int argc, char \*argv[])

{

  int i;

  printf(1, "alarmtest starting\n");

  alarm(10, periodic);

  for (i = 0; i < 50\*500000; i++){

    if ((i++ % 500000) == 0)

      write(2, ".", 1);

  }

  exit();

}

void

periodic()

{

  printf(1, "alarm!\n");

}

Output:

**$ alarmtest**

**alarmtest starting**

**.....alarm!**

**....alarm!**

**.....alarm!**

**......alarm!**

**.....alarm!**

**....alarm!**

**....alarm!**

**......alarm!**

**.....alarm!**

**...alarm!**

**...$**

10. #include "types.h"

#include "user.h"

#include "date.h"

int

main(int argc, char \*argv[])

{

  struct rtcdate r;

  if (date(&r)) {

    printf(2, "date failed\n");

    exit();

  }

  // your code to print the time in any format you like...

printf(1, "%d-%d-%d %d:%d:%d\n", r.year, r.month, r.day, r.hour, r.minute, r.second);

  exit();

}