LabX-6, BY 26 Nov 2021

How to do the exercises (partially repeat of text from EX-1)

We expect that after doing the exercises your skills in handling Linux OS and programming in C will increase. What it takes, is memorizing commands by typing the commands on the Linux terminal and observing the result. This is what the first part is about. This part is for the self-learning.

The second part is about writing a C program. At the end of the course you should be able to translate your research task into a C program based on libraries of different functions found in the Linux environment.

In this assignment we consider interprocesses communication. To accomplish this task you will use a function from standard library. Part of the exercise is to study the manual pages related to the suggested functions and add error checking into the programs.

You write the program yourself based on the guidance we give in the assignment. You can use the C book of Kernighan & Ritchie (see Pages/Textbooks on canvas[1]). In canvas Files you have a summary of C language in slides file 3.1_Linux-C[2] . Also use the summary slides in file 3.1_Linux_Bash[3]

You compile and run the program as in the instructions in this assignments and solving error messages from the compiler. After you convince yourself that the program is doing what it supposed to do you copy it to directory for us to check (see below).

You pass successfully if the program is running and produces the desired result. We will give you feedback on your programming style, as this is important going forward. In program text, we ask you to comment the statements in the programs. Also, the header in form of a comment should contain your name and date of creation and a summary of the intended program behavior.

On the delivery of your results: When you are ready with your program assignment, please create a directory with your name and copy the files there:

```
mkdir /trinity/home/LINUX_SUPERCOMPUTERS/yourname/   #it should exist already from exe1

mkdir /trinity/home/LINUX_SUPERCOMPUTERS/yourname/labx6

cp yourprogram /trinity/home/LINUX_SUPERCOMPUTERS/yourname/labx6/yourprogram
```

After you have finished copying and ready to report, please send us e-mail.If there are any problems or questions, please email me or Andrey:

"Igor Zacharov" I.Zacharov@skoltech.ru

"Andrey Kardashin" Andrey.Kardashin@skoltech.ru

[1] http://cslabcms.nju.edu.cn/problem_solving/images/c/cc/The_C_Programming_Language_(2nd_Edition_Ritchie_Kernighan).pdf

[2] https://skoltech.instructure.com/files/257760/download?download_frd=1

[3] https://skoltech.instructure.com/files/262799/download?download_frd=1

Further commands and Bash structure

Login to your sandbox account: ssh –p 2200 yourname@sandbox.zhores.net

Get used to the bash script

a) Open the editor and create file run1.sh with the following structure:

```
#!/usr/bin/bash

for x in 1 2 3 4
do
        echo $x
done
```

Note that x assumes values given after the "in" keyword and the value is used in the loop as $x

Make it a file called run1.sh (or similar) and chmod u+x run1.sh to start it as ./run1.sh

Similar exercise for another for () loop expression:

Call it run2.sh and make sure it prints the Iteration number.

```
#!/usr/bin/bash

NTRIAL=5

for (( k=0; k < $NTRIAL; k++ ))
do
        echo "Iteration: $k "
done
```

b) Usage of functions in bash scripts.

Here is the skeleton. Type it in file run3.sh and try:

This should print the largest size file in directory.
Run as follows:

```
Chmod u+x run3.sh
./run3.sh    #it will take /usr/bin by default
./run3.sh /tmp
./run3.sh /usr/include

Observe that func takes an argument as $n
from the invocation line.
Consider the filter

/bin/ls –l $1 | tr –s " " | cut –d  "  " –f 5
```

```
#!/usr/bin/bash

dir=${1:-/usr/bin}

function func () {

  sizes=$(/bin/ls -l $1 | tr -s " " | cut -d " " -f 5)

  local large=0        #strict local variable
  for x in $sizes
  do
                if [ $x -gt $large ]
                then
                        large=$x
                fi
        done
        echo $large

}
large=$(func $dir)
echo "largest size in directory $dir $large"
```

Here the ls command output is going to the tr (translate) command, which with the flag –s will squize all characters given in the flag parameter (thus multiple blanks will become a single blank). Then, the cut command will output only the 5th field (-f 5) from the command line while field delimiter is given with the –d " " flag (blank is the delimiter, thus).

You can tune this as follows:

```
ls –l /usr/bin | grep bash | cut –d " " –f 1
ls –l /usr/bin | grep bash | cut –d " " –f 2
….etc
```

When you arrive at 5th field you will see that only blanks are output. Remove the extra blanks with the `tr –s " "` command prior to the `cut` command in the filter. The `grep bash` there is to reduce the amount of the output to better see the result of the `cut` command.

Note that bash functions return as their result the status of the latest expression in the functions. The standard way to use the functions is to let them output to stdout and collect it with a variable.

c) You will need the following script to work with the solution of exercise (4) in Writing Programs.

```
#!/usr/bin/bash

export NUM_THREADS
NTRIAL=5


function pthr () {
        rt=$(/usr/bin/time -p ./pthr_atomic 2>&1 | grep real | cut -d" " -f 2)
        echo $rt
}

for x in 1 2 4
do
        NUM_THREADS=$x
        mintime=1000000
        echo ""

        for (( k=0; k < $NTRIAL; k++ ))
        do
            echo -n "trial $k:"    # the -n switch will not put \n

            rt=`pthr`
            echo -n " $rt   "
            ss=$(echo "$rt < $mintime" | bc)  # use bc for float comparions
            if [[ $ss -eq 1 ]];
            then
                mintime=$rt
                #echo "$ss assign $mintime from $rt"
            fi
        done
        echo " "
        echo "NUM_THREADS: $NUM_THREADS min_real_time: $mintime"
done
```

Observe the function, which runs the pthreads program (with atomic update or with the mutex or with the semaphore – script should be edited to put in the right program). It is using the time command – observe the /usr/bin/time to make sure we do not use the builtin bash shell time.

The redirection 2>&1 is there to take the stderr from the time to the stdout and the pipe.

The grep takes all lines with the word “real”, while cut takes second field with blank separating fields

The NUM_THREADS is initialized from the expression `for x in 1 2 4 ; do …..; done`

Another loop is for a number of trials $NTRIAL (set to 5, but you can modify to 20) to compare the run times for each trial and to select the smallest with the `if [[ … ]] ; then … ; fi` statement.

An important point here, is that bash can compare only integer. It does not work with the floating point numbers. Therefore we pipe the comparison to the bc command.

Try this expression separately: echo “1 < 2 “ | bc or any other like: echo “3.141 * 25” | bc

The bc reads from standard input. Try from the command line:

```
bc
3.141 *25
^D
```

Finish with Control-D.

Writing C programs

1) Private VA programs created with fork(). The parent and the child make a pipe channel between them. Note that while the virtual address spaces of the parent and the child are separated after the fork(), the opened file descriptors keep their meaning.

Make **two** programs:

a) parent is waiting for the data from child and prints it to stdout, while child is reading text from terminal and sends to parent

b) the roles of parent and child is reversed

While the difference between (a) and (b) is minor, the major task is to decorate the program(s) with the error checking for all library/system calls.

For example: int pid = fork(); should be

```
pid_t pid;
if ( ( pid = fork() ) < 0) {
        perror("fork");
        exit(1);
}
```

And similar rewrite for each library/sys call. The return codes are on the man page (eg. man fork or man 2 fork).

Specify in the comment header of the program the usual ident (name, date, version) and the details of the program:
i) what the program is for
ii) how to compile it
iii) how to run and the expected results of the run. This may be copied from the tests you do to verify that the program is working correctly.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>



main()
{
        int pfd[2];

        pipe(pfd);

        int pid = fork();

        if(pid) close(pfd[1]);  // write side, parent reading
        else    close(pfd[0]);  // read side, child writing

        while(1) {
                int inchar;
                int n;
                if(pid) {
                        n = read(pfd[0],&inchar,sizeof(inchar));
                        printf("%6d   got %d n=%d\n", pid, inchar,n);
                }
                else  {   // child code
                        //inchar = getchar();
                        scanf("%d",&inchar);
                        printf("%6d  send %d n=%d\n", pid, inchar,n);
                        sleep(1);
                        n = write(pfd[1],&inchar,sizeof(inchar));
                }
        }
        return 0;
}
```

2) System V shared memory program(s). This consists of two programs compiled and run separately.

Your task is to decorate the programs with the error handling code (see exe (1) for the explanation) and provide extended comments in the header for how to compile and run the program with test results.

Please, choose your own SHMKEY value such as not to interfere with other people using this facility.

Таб. 1 The program reading data from the shared memory (left) and writing data to the shared memory (right)

```c
/*
 * Prog 1 - sender: create shared segment, attach to it, write content
 * Prog 2 - receiver: attach to shared segment, read content
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>


#define SHMKEY          1234    // shm identifier
#define SHMSIZE         1024    // shm size
#define BUFSIZE          128


main()
{
        key_t key;
        size_t  size;
        int     shmid;
        void  * shared_memory;
        char buff[BUFSIZE];

        shmid = shmget((key_t) SHMKEY, SHMSIZE, 0666|IPC_CREAT);
        shared_memory = shmat(shmid, NULL, 0);

        //fgets(buff, BUFSIZE, stdin);
        strcpy(buff,shared_memory);

        printf("Attached to shmem key %d, id: %x output: %s\n",SHMKEY,shmid,buff);

        return 0;
}
```

```c
/*
 * Prog 1 - sender: create shared segment, attach to it, write content
 * Prog 2 - receiver: attach to shared segment, read content
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>


#define SHMKEY          1234    // shm identifier
#define SHMSIZE         1024    // shm size
#define BUFSIZE          128


main()
{
        key_t key;
        size_t  size;
        int     shmid;
        void  * shared_memory;
        char buff[BUFSIZE];

        shmid = shmget((key_t) SHMKEY, SHMSIZE, 0666|IPC_CREAT);
        shared_memory = shmat(shmid, NULL, 0);

        fgets(buff, BUFSIZE, stdin);
        strcpy(shared_memory,buff);

        printf("Attached to shmem key %d, id: %x input: %s\n",SHMKEY,shmid,buff);

        return 0;
}
```

Note:

a) The program will create a permanent shared memory area as an object in the file system. This has to be cleared after all tests are finished and as an exercise with the commands.

```
ipcs -m
ipcrm -m <shmid>
```

b) add the calls to detach from the memory: `shmdt(shared_memory)` and to remove the shared segment: `shmctl(shmid, IPC_RMID, NULL);` to the program(s).

To which program it should be added without any hassle?

c) Advanced:

Propose a piece of code such that the calls detaching from and removing the shared memory (shmdt, shmctl) can be safely added to both programs. For example, the write program can be allowed to remove the shmat only after the read program has read the data.

(Suggestion: come back to this point after you have done the other exercises. There might be something useful for this task. Also note: since no conditions are given how to use the data, you can use same shared area for the synchronization. – Or not at all).

(Another suggestion: this is an optional part, so if you spend more than 10 minutes thinking about the solution – abandon).

3) Program to demonstrate the RACE condition in an access to a shared variable. The race is between pthreads that are created as separate light-weight threads of control in share-all program. The threads are created with the calls

```
/* create the thread, does not block
*/
pthread_create(int *tid, NULL, void *(void*), void *);
```

```
/* block for completion
*/
pthread_join(int tid, NULL);
```

```c
/*
 * program to demonstrate RACE condition in access to shared variable
 * compile with gcc -o pthr pthr.c -pthreads
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NITER           500000000

static long long sum = 0;

void *func(void * arg){
    long n = *(long *) arg;
    long i;
    for (i=0; i<NITER; i++) {
        sum = sum + n;
    }
    pthread_exit(NULL);
}

int main(int argc, char **argv)
{
    long n1, n2 ;
    pthread_t tid1, tid2;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
        n1 =  1;
    pthread_create(&tid1, &attr, fun, &n1);
        n2 = -1;
    pthread_create(&tid2, &attr, fun, &n2);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("sum = %lld\n", sum);
    return 0;
}
```

Serial execution if create-join, create-join and parallel execution if create-create, join- join.

The program example shows 2nd case, the two threads run at the same time.

a) add the error handling to the program and the appropriate description into the header.

b) test that the program works when threads are running one after another. The total sum must be zero.

c) run the two threads in parallel. Observe the value of the sum after each run, i.e. verify that the program is erroneous.

d) go down with NITER. At what values the sum appears to be correct after 20 trials. Suggestion: use a script to run multiple times and grep the sum result for a comparison.

e) add the mutex lock to protect the shared variable (critical section). Here is the code:

```
pthread_mutex_lock(&mutex);
        sum = sum + n;
pthread_mutex_unlock(&mutex);
```

The mutex variable must be initialized in the global section of the program with `pthread_mutex_t mutex =  PTHREAD_MUTEX_INITIALIZER;`

Test the program for correctness (Warning – the program will run slower, adjust NITER as necessary)

f) With the correct program (using mutex) test another possible flaw. Name the n1 and n2 variables used as arguments to pthread_create same name (i.e. n) and run. What is the problem? – make a warning in the comments to the pthread_create calls about this condition and restore the correctness of the program for the submission.

g) Instead of the mutex lock, use semaphore as follows:
use the definition of a global shared variable and initialize it
in the main program (prior to the pthread calls): `sem_init(&sem, 0, 1);`

```
#include <semaphore.h>

Sem_t sem;
```

```
sem_wait(&sem);
        sum = sum + n;
sem_post(&sem);
```

Here the value of 1 is the initial value: at each invocation of sem_wait() this value will be decremented atomically and thread is allowed to pass if >= 0, otherwise blocks until the value is >=0. The post() has opposite effect, it increases the value and lets waiting threads continue execution.

4) Program to demonstrate the solution to the RACE condition with the atomic update.

The changed function and the definitions are shown here. Note that main was changed slightly to accommodate more flexible setting of number of running threads from an environment variable NUM_THREADS:

```c
/*
 * program to demonstrate solution to the
 * RACE condition in access to shared variable
 * using atomic operation on the shared variable
 *
 * compile with gcc -o pthr pthr.c -pthread
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>

#define NITER           500000000
#define NTHRS           16

atomic_long sum = 0;

void *func(void * arg){
    long n = *(long *) arg;
    long i;
    for (i=0; i<NITER; i++) {
        sum += n;  // note: only update op is atomic
    }
    pthread_exit(NULL);
}

int main(int argc, char **argv)
{
    long na[NTHRS];
    pthread_t tid[NTHRS];
    pthread_attr_t attr;
    int nth = 1;
    char *cnth = getenv("NUM_THREADS");
    int i;

    if(cnth) nth = atoi(cnth);
    if ( nth < 0 || nth >= NTHRS ) nth = 2;
    pthread_attr_init(&attr);

    printf("starting %d threads: ",nth);
    for ( i=0; i < nth; i++) {
        na[i] = (i & 1)? -1: 1;
        pthread_create(&(tid[i]), &attr, func, &(na[i]));
        printf("    [ %d: %x %d ]",i,tid[i],na[i]);
    }
    printf("\n");
    for ( i=0; i < nth; i++) {
       pthread_join(tid[i], NULL);
    }
    printf("sum = %lld\n", sum);
    return 0;
}
```

a) add the error handling to the program.

Note: before the compilation use modern compiler with the `module load` command

```
module load compilers/gcc-7.3.1
gcc -o prog prog.c -pthread
export NUM_THREADS=2
time ./prog
```

b) The time command prints the real (elapsed) time needed to complete the program. The user time is the count of the time spent on all CPUs.

With the real time measurement, complete the table of the running time for all RACE solutions:

Таб. 2 Comparison of the run time with RACE resolution.

RACE resolution	Real time[s] for threads		
	1	2	4
mutex			
semaphore			
atomic			

Choose a suitable NITER (same for all programs) such that the runtime of the slowest program is not more than about 1 minute with 2 processors. Run several times and take the **smallest** measured value, as it is not a clean experiment (other people will be using computer as well and will interfere with your timing). Therefore odd (strange) measurements can be discarded.

Note here: for faster experimentation here reimplement the mutex and semaphore using the atomic main program code above (thus you use the flexible main program where you can set NUM_THREADS, but remove the atomic_long and put in the critical section with the mutex or the semaphore).

Use the bash script (see first part of this exercise set) to automate the runs.