LabX-4, BY 19 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 look closer into the memory allocation functions. 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.

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]

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 may exist already from exe1

mkdir /trinity/home/LINUX_SUPERCOMPUTERS/yourname/labx4

cp yourprogram /trinity/home/LINUX_SUPERCOMPUTERS/yourname/labx4/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

Further commands

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

```
Ulimit
du
free
watch
errno
du
```

Get used to useful commands

a) Try `ulimit -a` this is bash buildin to display the limitations imposed for the usage of the resources by the OS and administrators. The limits can be extended by setting the limit, for example the maximum number of processes can be set with ulimit –u <number>. The soft limits (ulimit –aS) can be moved to the number presented by the hard limits (ulimit –aH).

b) Usage of the disk blocks of files and summary for the directories (with –s flag): du

Try du –ks /gpfs/gpfs0/ParallelComputingShared   same with du –sh , same with

du –sh /gpfs/gpfs0/ParallelComputingShared/MATMUL_LAB/matmulG.optrpt and compare with ls.

c) Commands to see available memory: `free -h`

```
[i.zacharov@an LabX_4]$ free -h
              total        used        free      shared  buff/cache   available
Mem:            16G        564M         15G        7.5M        324M         15G
Swap:          8.0G        191M        7.8G
```

Try also free –m
Notice that: available = free + shared + buff/cache

d) A way to repeatedly run the same command `watch -n 1 free -h` will display the output of free every second (-n, 1 second by default).
e) The errno command, try errno <number>
This corresponds to the errors that are printed with the function perror() in C.

f) The dumping of information into a file, typically used for backup. Here is an example of file creation:
dd if=/dev/zero of=/tmp/bigfile bs=1M count=1000
Note a different style as compared to other Linux commands!

Editor vi (vim) – A reminder if you did not get the habit yet (this is same as EX-1)

Create an empty file with name xx (or your choice) or just type in: vi xx

```
vi fname
```

Editor vi commands to enter the insert mode: i

Exit insert mode: key **Esc** on the keyboard

To write out the changed file: type in : (colon), then type in w (letter w)

To exit the editor: ZZ (Capitals) or via the command line: type :q

Watch (7m) https://www.youtube.com/watch?v=pU2k776i2Zw to learn vi.

Writing C programs

1) Obtaining information from the OS about the memory usage:

```
#define MiB     1.0/(1024*1024)
#define GiB     1.0/(1024*1024*1024)
unsigned long long freeram()  // get the available memory from system
{
        struct sysinfo info;
        unsigned long long ram = 0;

        if (sysinfo(&info) < 0) perror("sysinfo");
        else ram = info.freeram ;
        //printf("free: %llu %.1f GB mem unit: %d\n",
        //              info.freeram, (double)ram*GiB,  info.mem_unit);
        return ram;
}
```

```
/* example structure to allocate
   You can also define your own
*/
typedef struct mys {
       double x;
       double y;
} mys_type;
```

The program allocates maximum supported space and fills it up with zeros, until killed by the OS.

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>                 // get LONG_MAX
#include <sys/sysinfo.h>
#include <errno.h>                    // the error number and codes system

#define MiB     1.0/(1024*1024)
#define GiB     1.0/(1024*1024*1024)

unsigned long long freeram()
{
        struct sysinfo info;
        unsigned long long ram = 0;

        if (sysinfo(&info) < 0) perror("sysinfo");
        else ram = info.freeram;
        //fprintf(stderr,"free: %llu %.1f GiB mem unit: %d\n", info.freeram, (double)ram*GiB,  info.mem_unit);
        return ram;
}

int main()
{
        size_t mys_bytes = LONG_MAX;
        char * data;
        long ptr = NULL;
        long onegig = 1024*1024*1024;

        unsigned long long freememory = freeram();

        fprintf(stderr,"Available memory: %.2f GiB\n",  (double)freememory*GiB);
        fprintf(stderr,"Asking allocation max size: %lld %.2f GB\n", mys_bytes, mys_bytes*GiB);

        if ( malloc(mys_bytes) == NULL) perror("malloc" );
/*      an alternative way to look at the errors:
 */
        fprintf(stderr,"error number %d text: %s\n", errno, strerror(errno));

        while ( (data = (char *) malloc(mys_bytes)) == NULL) mys_bytes >>= 1;
        fprintf(stderr,"successfuly allocated %lld %.2f GB\n",mys_bytes, mys_bytes*GiB);


        do {
                memset( data+ptr, (int) 0, (size_t) onegig );
                sleep(1);
                ptr += onegig;
                fprintf(stderr,"memory %10X-%10X initialized %10lld %.2f GB\n", data+ptr-onegig, data+ptr-1, ptr, ptr*GiB);
        } while (ptr < mys_bytes);

        free(data);
}
```

Compile and run as follows:

```
Gcc -o maxmem maxmem.c
./maxmem &> log &
Watch free -h
```

The program maxmem will be killed when it reaches the maximum available memory. You will notice it based on the output of the free command. Then you kill the watch command with ^C (CntrC keys).

2) Program to allocate memory with malloc. The important parameter is the number of elements of a structure

```
/* example structure to allocate
   You can also define your own
*/
typedef struct mys {
        double x;
        double y;
} mys_type;
```

The program should be started with prog –s <elements>

Here is an example:

```
long num_elements = getparams(argc,argv);

unsigned long long freememory = freeram();
unsigned long long freeleft, usedmem;

size_t mys_bytes;
mys_type *data;

printf("Number of elements:  %d available memory: %.2f MB\n",  num_elements, (double)freememory*MiB);

mys_bytes = num_elements * sizeof(mys_type);
if ( (mys_bytes <= 0) || (mys_bytes > freememory ) ) { // free memory is not quite right in sandbox
        fprintf(stderr,"illegal numer of elements: %d, FreeMem: %u\n",num_elements,freememory);
        exit(4);
}
data = (mys_type *) malloc(mys_bytes);
if ( data == NULL ) perror("malloc");
//else memset( data, (int)0, mys_bytes);

for( i=0; i < num_elements; i++ ) {
        data[i].x = 1.0;
        data[i].y = 2.0;
}

freeleft = freeram() ;
usedmem = freememory - freeleft;

fprintf(stderr,"memory allocation size %lld Bytes %.1f MiB done. Used memory: %llu Bytes %.1f MiB\n",
        mys_bytes,(double)mys_bytes*MiB, usedmem, (double)usedmem*MiB);

//sleep(10);

free(data);
```

Running for several different number of elements, making sure you understand the size of memory in Bytes (MB or GB) to # elements. Do the same for a matrix: double m[N][N]; with N=# elements, i.e. the program will run as: prog –s <N>.

Notice the free() call to free the allocated space. However, it is freed automatically when program terminates.

Make this program and compile without errors. What is N to use 6 GB of allocated memory? You can verify with `free -h` to see how the free memory has decreased when the program is holding the memory (activate the sleep function commented out in the code to hold the memory while you check).

3) the memory can be allocated with mmap command. This has the advantage that several programs can map out same address space and establish inter-process communication. The allocation unit is the page size (it allocates an integral number of pages).

```
const long pagesize = sysconf(_SC_PAGESIZE);   // get the default page size to use in allocation

mys_type *data;
long num_elements = getparam(argc, argv);
size_t mys_bytes = num_elements * sizeof(mys_type);  // bytes needed to allocate that many elements
long pagesdef = 1 + (mys_bytes + pagesize -1)/pagesize;  // #pages needed for the allocation
```

The relevant code is as follows:

```
mys_bytes = pagesdef * pagesize;               // allocate memory with integral #pages
data = (mys_type *) mmap(NULL,mys_bytes, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1, (off_t)0);
if ( data == NULL ) perror("mmap");
else memset( data, (int)0xFF, mys_bytes);
```

This mmap call is suitable for single program allocation and for parallel programs created by fork or for shared memory programming models (such as OpenMP or PThreads). For independent programs they have to open/create a well known file name and pass the file descriptor (instead of -1 above).

Replace the mmaloc with mmap function in the program exercise (2) and verify it works same way.

4) Program to count words in a text file by mapping out the text file and using pointers to navigate the data. As a side note, you cannot map out the input stream like this, only files in the file system.

Here is the main program:

```
        const long pagesize = sysconf(_SC_PAGESIZE);    // get the default page size to use in allocation

        char   *data;
        struct stat file_status;
        int     fd;
        char * file_name;
        size_t alloc_bytes;
        long    pagesdef;

        long    word_count = 0;

        if(argc <= 1) usage(argc,argv);
        else file_name = argv[1];

        if ( ( fd = open(file_name,O_RDONLY) ) < 0 ) { perror("open"); exit(1); }
        if ( fstat(fd, & file_status) < 0) { perror("fstat"); exit(1); }
        pagesdef = 1 + (file_status.st_size + pagesize -1)/pagesize;    // integral # pages in which file fits

        alloc_bytes = pagesdef * pagesize;                 // allocate memory with integral #pages
        data = (char *) mmap(NULL, alloc_bytes, PROT_READ, MAP_PRIVATE, fd, (off_t)0);
        if ( data == NULL ) { perror("mmap"); exit(1); }
        else close(fd);

/*      the inormation in the mapped file is now available as dereference from *data
 */
        word_count = process_words(data);


        printf("number of words in file %s (size %ld): %ld\n",file_name, file_status.st_size, word_count);


        munmap(data,alloc_bytes);
        exit(0);
```

And now you can manipulate all information with the file using pointers:

```
char *get_word(char **ps)             // search the data for the end of word, triggered by a punktuation character, blank or \0
{                                     // single letter words count (may need changing)
        char *p1, *p2;

        p1 = *ps;
        while ( ! isalnum(*p1) && *p1 ) p1++;   // skip all "not letter" at the beginning of the data and break if '\0' contents
        *ps = p2 = p1;                          // when condition fails the pointer is at the first letter of a word


        while (   isalnum(*p2) && *p2 ) p2++;   // when condition fails, the pointer will be pointing to first "not letter"

#ifdef DEBUG
        {
                char word[50];
                int  wsize = p2 - p1;       // assume < 50, for serious prog needs checking
                strncpy(word, p1, wsize);
                word[wsize] = '\0';
                printf("word found: %d -%s-\n",wsize, word);
        }
#endif
        return p2;
}

long process_words(char *data)
{
        char *p1, *p2;          // pointers to the start and the end of each word
        long word_count = 0;

        p2 = data;

        do {
                p1 = p2;
                p2 = get_word(&p1);
                if ( p2 == NULL ) break;
                word_count++;
        } while ( p2 > p1 );

        return word_count;
}
```

Note that to activate the DEBUG section in function get_word you compile with gcc –DDEBUG flag.

Make sure the program runs as follows: prog blackhol.txt

It should produce output similar to wc –w blackhol.txt (difference due to definition of word).

5) Advanced program computing frequency of words with a hash table. Same file mapping as in exercise (4). Basically it is the same program with additional functions. Here are parts of the program.

Definitions:

```c
#define HASHTAB_SIZE            ((1<<9)-1)      // assume the initial table size, must be power 2 (mask),
#define HASHSEED                0x12345678      // hash starting value
#define NUMBUCKETS              10              // number of alternative hash tables when collisions occure
#define MAXWORD_LEN             20              // assume maximum word length

typedef unsigned int uint32_t;

typedef struct hashdata{
        long count;                     // number of occurencies
        long colis;                     // count collisions
        char word[MAXWORD_LEN];         // word
} hashdata_type;

static long _collisions = 0;                                    // count the total number of collisions
static hashdata_type *hashtabs[NUMBUCKETS] = {0};                       // exapand hash if collisions
```

The hash function:

```c
uint32_t MurmurOAAT_32(const char* str, int strlen, uint32_t h)
{
    // One-byte-at-a-time hash based on Murmur's mix
    // Source: https://github.com/aappleby/smhasher/blob/master/src/Hashes.cpp
        int i;

        for (i=0; i < strlen; i++ ) {
                        h ^= *str++;
                        h *= 0x5bd1e995;
                        h ^= h >> 15;
        }
        return h;
}
```

Additional functions:

```c
hashdata_type * allhash()
{
        hashdata_type * hashtable;

        if ( ( hashtable = malloc(sizeof(hashdata_type) * HASHTAB_SIZE) ) == NULL ) { perror ("malloc hashdata"); exit(4); }
        else memset(hashtable, (int) 0, sizeof(hashdata_type) * HASHTAB_SIZE);
        return hashtable;
}

void hashfree()
{
        int i;
        for ( i=0; i < NUMBUCKETS; i++) if ( hashtabs[i] != NULL ) free(hashtabs[i]);
}

void printinfo()
{
        hashdata_type *htab;
        int i,j, newl;
        int sparce[NUMBUCKETS] = {0};

        printf("%32s   Freq  Colis\n","word");
        for ( i=0; i < HASHTAB_SIZE; i++ ) {
                newl = 0;
                for (j=0; j < NUMBUCKETS; j++) {
                        if ( (htab = hashtabs[j]) == NULL) break;
                        if ( htab[i].count > 0 ) {
                                printf("%16s    %d    %d", htab[i].word,htab[i].count, htab[i].colis);
                                sparce[j]++;
                                newl = 1;
                        }
                }
                if ( newl ) printf("\n");
        }
        for (j=0; j < NUMBUCKETS; j++) {
                printf(" %.2f ", (double)sparce[j]/HASHTAB_SIZE);
        }
        printf("collisions: %ld\n",_collisions);
}
```

The actual processing:

```
long process_words(char *data)
{
	char *p1, *p2;		// pointers to the start and the end of each word
	hashdata_type *htab;
	long word_count = 0;
	uint32_t hashvalue;
	int hindex;

	p2 = data;

	do {
		int wsize;
		int i;
		p1 = p2;
		p2 = get_word(&p1);
		if ( p2 == NULL ) break;
		wsize = ((p2 - p1) < MAXWORD_LEN ? (p2-p1): (MAXWORD_LEN-1));

		word_count++;

		hashvalue = MurmurOAAT_32(p1, wsize, HASHSEED);
		hindex = hashvalue & HASHTAB_SIZE;
	/*	Try the index in available hash tables		*/

		for(i=0; i < NUMBUCKETS; i++) {

			htab = hashtabs[i];
			if ( htab == NULL ) htab = hashtabs[i] = allhash();

			if( htab[hindex].count == 0 ) {
				strncpy(htab[hindex].word, p1, wsize);
				htab[hindex].word[wsize] = '\0';
				htab[hindex].count++;
				break;
			}
			else {
				if ( strncmp(htab[hindex].word, p1, wsize ) == 0 ) {
					htab[hindex].count++;
					break;
				}
				else {
					htab[hindex].colis++;
				}
			}
		}
		if (i == NUMBUCKETS) _collisions++;

	} while ( p2 > p1 );

	return word_count;
}
```

Make sure the program runs as follows: prog blackhol.txt and prints out a table with words and the number of their occurrences. It gives some additional information on the work of the hash.

Run with different sizes of the hash table (HASHTAB_SIZE) and notice how the occupancy of buckets changes.

6) same as above, but replace the malloc in allhash() by a mmap.

Hint: there should be one big mmap for the has tables in main, the allhash() should just initialize the relevant parts and setup the pointers for the hash tables.