Pledge on honor

1. Carefully read the text below

2. Once you have understood this text, and agreed with it, recopy it in the text field below.

3. Once you have recopied this text, you can proceed to the end of the quizz and to the final exam

Pledge on honor

C program

Check all the true claims about a c program

- [ ] The execution of a C program creates a new process in the OS if enough memory is available

- [ ] The main function of a C program cannot call sub-functions

- [ ] A call to exit() makes the program returns to the OS after cleanup handlers have been called

- [ ] A call to_exit() provokes the immediate end of the program

- [ ] When calling _exit(), no allocated resources are freed by the OS

Compilation and OS

Check all the true claims about compilation of a C program P.

- [ ]

Once compiled, P can always be executed on another computer running the same Operating System

☐

Once compiled, P can be executed on another computer running the same Operating System and having the same hardware architecture

☐

The compilation of P generates an executable file only if no errors are detected by the compiler

☐

The compilation of P generates an executable file only if no warnings are detected by the compiler

Data streams of processes

Check all the true claims about data streams of processes.

☐

A process has 3 default data streams

☐

The standard output stream (stdout) of a process can be redirected to the standard error stream (stderr)

☐

The output stream of a process can be redirected to the input stream of a process

☐

"$ ls | grep bin" creates two processes, with the first one forwarding its output data stream to the input data stream of the second.

☐

Files cannot be used as input data stream

gcc

Check all the true claims about the C compiler "gcc".

☐

gcc can take as input a C file from which it can generate an executable file specific to the OS

☐

gcc can print all warnings of the C input code when required

☐ An executable file generated by gcc can be executed only once

Inter Process Communication

Check all the true claims about Inter Process Communications.

☐ In the shell, the "|" symbol represents a communication between two processes

☐

Memory can be shared between processes without the permission of the OS

☐

Files can be used to exchange data between processes

☐

The ">" symbol of the shell cannot be used to store information in files

Inter Process Communication: the basics

Check all the true claims about the communication between processes.

☐

Processes can share part of their memory without asking the OS

☐

OS can prevent processes from communicating together

☐

Killing a process is a communication from the sending process to the killed process

☐

ls | grep ".tgz* is a shell command that uses communication between processes

☐

Entering CTRL-C is handled as a communication between two processes

Malloc() and Brk()

Check all the true claims about brk() and malloc(). To help you, we provide the manul page of *brk*() below.

--

BRK(2) Linux Programmer's Manual BRK(2)

NAME
brk, sbrk - change data segment size

SYNOPSIS
#include <unistd.h>

int brk(void *addr);

void *sbrk(intptr_t increment);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

brk(), sbrk():
Since glibc 2.19:
_DEFAULT_SOURCE ||
(_XOPEN_SOURCE >= 500) &&
! (_POSIX_C_SOURCE >= 200112L)
From glibc 2.12 to 2.19:
_BSD_SOURCE || _SVID_SOURCE ||
(_XOPEN_SOURCE >= 500) &&
! (_POSIX_C_SOURCE >= 200112L)
Before glibc 2.12:
_BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION
brk() and sbrk() change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

brk() sets the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size (see setrlimit(2)).

sbrk() increments the program's data space by increment bytes. Calling sbrk() with an increment of 0 can be used to find
the current location of the program break.

RETURN VALUE

On success, brk() returns zero. On error, -1 is returned, and errno is set to ENOMEM.

On success, sbrk() returns the previous program break. (If the break was increased, then this value is a pointer to the
start of the newly allocated memory). On error, (void *) -1 is returned, and errno is set to ENOMEM.

CONFORMING TO

4.3BSD; SUSv1, marked LEGACY in SUSv2, removed in POSIX.1-2001.

NOTES

Avoid using brk() and sbrk(): the malloc(3) memory allocation package is the portable and comfortable way of allocating memory.

Various systems use various types for the argument of sbrk(). Common are int, ssize_t, ptrdiff_t, intptr_t.

C library/kernel differences

The return value described above for brk() is the behavior provided by the glibc wrapper function for the Linux brk() system
call. (On most other implementations, the return value from brk() is the same; this return value was also specified in
SUSv2.) However, the actual Linux system call returns the new program break on success. On failure, the system call re-
turns the current break. The glibc wrapper function does some work (i.e., checks whether the new break is less than addr)
to provide the 0 and -1 return values described above.

On Linux, sbrk() is implemented as a library function that uses the brk() system call, and does some internal bookkeeping so
that it can return the old break value.

SEE ALSO

execve(2), getrlimit(2), end(3), malloc(3)

COLOPHON

This page is part of release 4.16 of the Linux man-pages project. A description of the project, information about reporting
bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

- [ ] This manual page of *brk*() applies to all Linux 4 versions
- [ ] *brk*() can be used to allocate memory
- [ ] *brk*() can be used to disallocate memory
- [ ] *malloc*() can be used to allocate memory
- [ ] *malloc*() can be used to disallocate memory
- [ ] *brk*() is a syscall
- [ ] *malloc*() is a syscall
- [ ] *brk*() fails if not enough memory is available

Malloc: memory areas

Check all the true claims about *malloc()*.

- [ ] *malloc()* can be used to allocate the mémory of global variables
- [ ] *malloc()* can be used to allocate memory on the stack
- [ ] *malloc()* can be used to allocate memory on the heap
- [ ] *malloc()* can be used to allocate memory for the code of the program

Manual page of mq_receive

Check all the true claims about the manual page of mq_receive provided below.

--

```
MQ_RECEIVE(3)                          Linux Programmer's
Manual                       MQ_RECEIVE(3)

NAME
    mq_receive, mq_timedreceive - receive a message from a message queue

SYNOPSIS
    #include <mqueue.h>

    ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                    size_t msg_len, unsigned int *msg_prio);

    #include <time.h>
    #include <mqueue.h>

    ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr,
                    size_t msg_len, unsigned int *msg_prio,
                    const struct timespec *abs_timeout);

    Link with -lrt.

  Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

    mq_timedreceive():
        _POSIX_C_SOURCE >= 200112L
```

DESCRIPTION
mq_receive() removes the oldest message with the highest priority from the message queue referred to by the message queue de-
scriptor mqdes, and places it in the buffer pointed to by msg_ptr. The msg_len argument specifies the size of the buffer pointed
to by msg_ptr; this must be greater than or equal to the mq_msgsize attribute of the queue (see mq_getattr(3)). If msg_prio is
not NULL, then the buffer to which it points is used to return the priority associated with the received message.

If the queue is empty, then, by default, mq_receive() blocks until a message becomes available, or the call is interrupted by a
signal handler. If the O_NONBLOCK flag is enabled for the message queue description, then the call instead fails immediately
with the error EAGAIN.

mq_timedreceive() behaves just like mq_receive(), except that if the queue is empty and the O_NONBLOCK flag is not enabled for
the message queue description, then abs_timeout points to a structure which specifies how long the call will block. This value
is an absolute timeout in seconds and nanoseconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC), specified in the following
structure:

```
struct timespec {
    time_t tv_sec;        /* seconds */
    long   tv_nsec;       /* nanoseconds */
};
```

If no message is available, and the timeout has already expired by the time of the call, mq_timedreceive() returns immediately.

RETURN VALUE
On success, mq_receive() and mq_timedreceive() return the number of bytes in the received message; on error, -1 is returned, with
errno set to indicate the error.

- [ ] No specific include is necessary for using mq_receive()

- [ ] mq_receive() is a syscall

- [ ]

An extra library must be added to the compilation line to use mq_receive()

☐

O_NONBLOCK cannot be used with mq_timedreceive()

☐

mq_receive() always returns the number of bytes read

Manual page of write

Check all the true claims about the following manual page

--

WRITE(2) Linux Programmer's
Manual WRITE(2)

NAME
write - write to a file descriptor

SYNOPSIS
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.

The number of bytes written may be less than count if, for example, there is insufficient space on the underlying physical
medium, or the RLIMIT_FSIZE resource limit is encountered (see setrlimit(2)), or the call was interrupted by a signal han-
dler after having written less than count bytes. (See also pipe(7).)

For a seekable file (i.e., one to which lseek(2) may be applied, for example, a regular file) writing takes place at the
file offset, and the file offset is incremented by the number of bytes actually written. If the file was open(2)ed with
O_APPEND, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the
write operation are performed as an atomic step.

POSIX requires that a read(2) that can be proved to occur after a write() has returned will return the new data. Note that
not all filesystems are POSIX conforming.

According to POSIX.1, if count is greater than SSIZE_MAX, the result is implementation-defined; see NOTES for the upper
limit on Linux.

RETURN VALUE
On success, the number of bytes written is returned (zero indicates nothing was written). It is not an error if this num-
ber is smaller than the number of bytes requested; this may happen for example because the disk device was filled. See
also NOTES.

On error, -1 is returned, and errno is set appropriately.

If count is zero and fd refers to a regular file, then write() may return a failure status if one of the errors below is
detected. If no errors are detected, or error detection is not performed, 0 will be returned without causing any other ef-
fect. If count is zero and fd refers to a file other than a regular file, the results are not specified.

- [ ] write() is a system call
- [ ] To compile a C program with write(), unistd.h must be included
- [ ] write() takes as input 3 parameters
- [ ] write() always returns the number of written bytes
- [ ] When write() returns less than "count", an error occurred in the system
- [ ] Written bytes are always placed at the beginning of a file

Manual pages

Check all the true claims about manual pages.

- [ ]

Manual pages are divided in sections

☐

Section 2 of manual pages is for system calls

☐

Section 2 of manual pages is specific to the installed kernel

☐

Manual pages give information about the functions of the C library (a.k.a. libc)

Memory allocation

Check all the true claims about the execution of the C code provided below.

--

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

int funnyAllocation(char *buf, int b) {
  buf = (char *) ( malloc(sizeof(char) * 5));
  strcpy(buf, "hello");

  return 7;
}

int main( int argc, char*argv[] ) {
  int b = 3;

  char *buf = (char *) ( malloc(sizeof(char) * 5));

  int returned = funnyAllocation(buf, b);

  printf("The content of buf is: %s\n", buf);
}
```

☐

argc is not used

☐

The execution can provoke a memory allocation error

☐

hello is printed in the standard output

Memory allocation: At run time or not?

Check the memory sections of processes containing memory dynamically allocated during run time

☐

Global variables

☐

Code of the program

☐

Stack

☐

Heap

Memory management

Check all the true claims about memory management.

☐

OS translates logical memory addresses into physical addresses each time a user process performs a read/write operation in physical memory

☐

OS configures the Memory Management Unit for address translation

☐

Memory Management Unit must be configured by user processes before accessing to the physical memory

☐

Programmers usually directly use system calls to allocate memory

Message queues: Linux kernel code

The following code is taken from the ipc/msg.c file of the Linux Kernel 4.19.225. Check all the true claims that follow.

```
/**
 * newque - Create a new msg queue
 * @ns: namespace
 * @params: ptr to the structure that contains the key and msgflg
 *
 * Called with msg_ids.rwsem held (writer)
 */
static int newque(struct ipc_namespace *ns, struct ipc_params *params)
{
	struct msg_queue *msq;
	int retval;
	key_t key = params->key;
	int msgflg = params->flg;

	msq = kvmalloc(sizeof(*msq), GFP_KERNEL);
	if (unlikely(!msq))
		return -ENOMEM;

	msq->q_perm.mode = msgflg & S_IRWXUGO;
	msq->q_perm.key = key;

	msq->q_perm.security = NULL;
	retval = security_msg_queue_alloc(&msq->q_perm);
	if (retval) {
		kvfree(msq);
		return retval;
	}

	msq->q_stime = msq->q_rtime = 0;
	msq->q_ctime = ktime_get_real_seconds();
	msq->q_cbytes = msq->q_qnum = 0;
	msq->q_qbytes = ns->msg_ctlmnb;
	msq->q_lspid = msq->q_lrpid = NULL;
	INIT_LIST_HEAD(&msq->q_messages);
	INIT_LIST_HEAD(&msq->q_receivers);
	INIT_LIST_HEAD(&msq->q_senders);

	/* ipc_addid() locks msq upon success. */
	retval = ipc_addid(&msg_ids(ns), &msq->q_perm, ns->msg_ctlmni);
	if (retval < 0) {
		ipc_rcu_putref(&msq->q_perm, msg_rcu_free);
		return retval;
	}

	ipc_unlock_object(&msq->q_perm);
```

```
        rcu_read_unlock();

        return msq->q_perm.id;
}
```

- [ ] The function creates a new message
- [ ] The function returns a pointer to a message queue
- [ ] This function returns an error if no "id" is available
- [ ] The function allocates a new structure but has to disallocate it in case of error
- [ ] The created object can handle receivers

Processor execution mode

Processors usually have two executions modes: the kernel mode, and the user mode. Check all the true claims among the following ones.

- [ ] Privileged assembly instructions can be executed only in user mode
- [ ] Switching from user mode to kernel mode requires a privileged assembly instruction
- [ ] The administrator (i.e., root) of a machine can only execute privileged assembly instructions
- [ ] Interrupt handlers are executed in kernel mode

☐

System calls are executed in kernel model

Sending and receiving signals

Check all the correct claims which concern the code below.
The following code makes it possible to exchange signals between a sender and a receiver. We assume that the receiver is started a few seconds before the sender. Also, the command line to start the sender provides the process id of receiver. Last but not least, we assume that all works as expected (no process is killed during execution, etc.)

Receiver code:

```
void getSignal(int signo) {

if (signo == SIGUSR1) {
printf("Received SIGUSR1\n");
} else {
printf("Received%d\n", signo);
}

}

int main(void) {
printf("Registering SIGUSR1 signal / #SIGUSR1=%d\n", SIGUSR1);
signal(SIGUSR1, getSignal);
sleep(30);

}
```

Sender code:

```
int main(int argc, char**argv) {
int pid;
if (argc <2) {
printf("Usage: sender <destination process pid>\n");
exit(-1);
}
pid = atoi(argv[1]);
printf("Sending SIGURG to %d\n", pid);
kill(pid, SIGURG);
printf("Sending SIGUSR1 to %d\n", pid);
kill(pid, SIGUSR1);
printf("Sending SIGUSR1 to %d\n", pid);
kill(pid, SIGUSR1);
```

}

- [ ] getSignal() is called at most three times

- [ ] getSignal() is called at most two times

- [ ] getSignal() is called at most one time

- [ ] If SIGKILL were to be sent at first by sender instead of SIGURG, the behavior of the receiver would be different
behavior of receiver would be the same.

- [ ] The value returned by all system calls are checked for errors

Sending and receiving signals

Check all the correct claims which concern the code below.

The following code makes it possible to exchange signals between a sender and a receiver. We assume that the receiver is started a few seconds before the sender. Also, the command line to start the sender provides the process id of receiver. Last but not least, we assume that all works as expected (no process is killed during execution, etc.)

Receiver code:

```
void getSignal(int signo) {

  if (signo == SIGUSR1) {
    printf("Received SIGUSR1\n");
  } else {
    printf("Received%d\n", signo);
  }
```

```
}

int main(void) {
  printf("Registering SIGUSR1 signal / #SIGUSR1=%d\n", SIGUSR1);
  signal(SIGUSR1, getSignal);
  sleep(30);
}
```

<u>Sender code:</u>

```
int main(int argc, char**argv) {

  int pid;
  if (argc <2) {
    printf("Usage: sender <destination process pid>\n");
    exit(-1);
  }
  pid = atoi(argv[1]);
  printf("Sending SIGURG to %d\n", pid);
  kill(pid, SIGURG);
  printf("Sending SIGUSR1 to %d\n", pid);
  kill(pid, SIGUSR1);
  printf("Sending SIGUSR1 to %d\n", pid);
  kill(pid, SIGUSR1);
 }
```

- [ ] getSignal() is called three times
- [ ] The return values of all system calls are checked for errors
- [ ] getSignal() is called two times
- [ ] getSignal() is called one time
- [ ] If SIGKILL were to be sent at first by sender instead of SIGURG, the behavior of receiver would be the same.

Stat on a file

Check all the true claims about the result of the stat command given below.

--

```
$ stat test
  File: test
  Size: 8811          Blocks: 24         IO Block: 4096   regular file
Device: fd01h/64769d    Inode: 22814078    Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 8003/apvrille)   Gid: (  105/soc_staff)
Access: 2022-09-16 18:10:22.641433749 +0200
Modify: 2022-09-16 18:10:22.625433571 +0200
Change: 2022-09-16 18:10:22.625433571 +0200
 Birth: -
\end{lstlisting}
```

☐

All persons belonging to the "soc_staff" group can modify "test"

☐

"test" is a directory

☐

As a student, you can read the content of "test"

☐

"test" uses 22814078 inodes on the disk

System calls vs. librady functions

Which following claims are correct? Thee claims are related to the differences between system calls and functions of libraries.

☐

System calls can execute privileged assembly instructions but functions of libraries cannot

☐

The manual pages of system calls are listed in a different section than the ones of library functions

☐

There are more system calls than functions of libraries

☐

Function of libraries cannot call memory allocations routines while system calls can

The kill bash command

Check all the true claims about the manual page of kill(1) provided below.

--

KILL(1) User
Commands KILL(1)

NAME
kill - send a signal to a process

SYNOPSIS
kill [options] <pid> [...]

DESCRIPTION
The default signal for kill is TERM. Use -l or -L to list available signals.
Particularly useful signals include HUP,
INT, KILL, STOP, CONT, and 0. Alternate signals may be specified in three ways: -9, -SIGKILL or -KILL. Negative PID val-
ues may be used to choose whole process groups; see the PGID column in ps command output. A PID of -1 is special; it indi-
cates all processes except the kill process itself and init.

OPTIONS
<pid> [...]
Send signal to every <pid> listed.

-<signal>
-s <signal>
--signal <signal>
Specify the signal to be sent. The signal can be specified by using name or number. The behavior of signals is ex-
plained in signal(7) manual page.

-l, --list [signal]
List signal names. This option has optional argument, which will convert signal number to signal name, or other way
round.

-L, --table
List signal names in a nice table.

NOTES Your shell (command line interpreter) may have a built-in kill command. You may need to run the command described here as /bin/kill to solve the conflict.

EXAMPLES
kill -9 -1
Kill all processes you can kill.

kill -l 11
Translate number 11 into a signal name.

kill -L
List the available signal choices in a nice table.

kill 123 543 2341 3453
Send the default signal, SIGTERM, to all those processes.

SEE ALSO
kill(2), killall(1), nice(1), pkill(1), renice(1), signal(7), skill(1)

- [ ] kill can only be used to terminate a process
- [ ] kill takes as input at least one process id or -1
- [ ] -9 is a process id
- [ ] kill -9 -1 terminates all processes of the OS
- [ ] Specifiying a signal number is optional
- [ ] The kill command accepts more than one pid

When will my program crash?

The following program has a memory allocation issue. At which loop index will our program generate a segmentation fault?

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

int main(int argc, char ** argv) {
  char *name;
```

```
  long long i;

  name = (char *) (malloc (20 * sizeof (char)));

  for(i=0; i>-1; i++) {
    name[i] = (char)i;
    printf("i=%lld\n", i);
  }

}
```

- [ ] We cannot predict exactly, but it may crash when i >= 20
- [ ] At i = 0
- [ ] At i = 20
- [ ] At i = 21
- [ ] at i = 19

Segmentation fault or not?

Does this program always provoke a segmentation fault?

```
#include <stdlib.h>

int main(int argc, char ** argv) {
  char *name;

  name = (char *) (malloc (20 * sizeof (char)));
  name[22] = 'h';
}
```

- [ ] True
- [ ] False

Envoyer