CSC B09H1 S 2017 Final exam
Duration — 150 minutes
Aids allowed: Handwritten single-sided crib sheet

Student Number: |_|_|_|_|_|_|_|_|_|_|

Lab day, time: _____

Last Name: _____ **First Name:** _____

Instructor: Bianca Schroeder

---

*Do **not** turn this page until you have received the signal to start.*
(Please fill out the identification section above and read the instructions below.)
Good Luck!

This midterm consists of 7 questions on 30 pages (including this one). *When you receive the signal to start, please make sure that your copy is complete.* Comments are not required except where indicated, although they may help us mark your answers. Note, that on the last pages of the exam we are providing some information from various Unix man pages that might be useful.
If you use any space for rough work, indicate clearly what you want marked.

1: _____/ 6
2: _____/ 9
3: _____/ 9
4: _____/ 8
5: _____/ 9
6: _____/ 9
7: _____/ 7

TOTAL: _____/57

Questions 1, 4
are relevant
2 maybe as well, not sure

Question 1. [6 MARKS]

Some of the code fragments below have a problem. For each fragment indicate whether the code works as intended or whether there is an error (logical error, compile-time error/warning, or runtime error). Assume all programs are compiled using the C99 standard. For this question, we'll assume programs which do not terminate are errors as well. If there is an error in a fragment, explain **briefly** what is wrong in the box. We have intentionally omited the error checking of the system calls to simplify the examples. Do not report this as an error.

Part (a) [1 MARK]

```
int x = 5;
// checking whether x equals 6
if ( x = 6 ) {
  printf("x equals 6\n");
}
```

☐ Works as intended ☑ Error

x = 6 is not an equality check, it's an assignment operator.

Part (b) [1 MARK]

```
int x;
//reading a value for x
scanf("%d", &x);
```

☑ Works as intended ☐ Error

Part (c) [1 MARK]

```
struct student {
    int age;
    char *name;
}

// Increase the age of a student by amt.
void increase_age(struct student s, int amt) {
    s.age += amt;
}

int main() {
    struct student rob;
    rob.age = 10;
    increase_age(rob, 5);
    printf("%d should be 15\n", rob.age);
}
```

☐ Works as intended ☑ Error

you copied the rob object and changed the age of the copy, not the original.

Part (d) [1 MARK]

```
char * st = malloc(31);
// reading a string into st
// you may assume that not more than 30 characters are read
scanf("%s", &st);
```

☐ Works as intended ☑ Error

should be st

Part (e) [1 MARK]

```
#include <string.h>
int main()
{
  char * st;
  // copying "abc" into st
  strcpy(st, "abc");
  return 0;
}
```

☐ Works as intended ☑ Error

gotta malloc() for st

Part (f) [1 MARK]

```
char st1[] = "abc";
char st2[] = "abc";
if ( st1 == st2 )
  printf("Strings are identical");
else
  printf("Not identical");
```

☐ Works as intended ☑ Error

need to use strcmp instead of st1 == st2

Question 2. [9 MARKS]

In this question you will write a function `hash` that will take as arguments the name of a file `filename` and a hash block size blocksize. It will then read the contents of file `filename` byte by byte and compute (and return) a hash with the size specified in `blocksize`. The hash you implement should be based on xor and your function should be able to handle text files as well as binary files. You can assume that `blocksize` passed to the function is a valid block size and that no error occur during any of the system calls you might make. You can also assume that any libraries you need have been included.

```
char *hash(char *filename, int blocksize) {

}
```

Question 3. [9 MARKS]

Suppose you want write a program with two processes that communicate through a pipe. Decide for each of the following statements whether they are correct or not:

You need to call pipe() before you call fork().
☐ True ☐ False

You need to call fork() before you call pipe().
☐ True ☐ False

You need to call pipe, but you don't necessarily have to call fork.
☐ True ☐ False

Pipes are uni-directional, which means that only the parent can read and only the child can write.
☐ True ☐ False

It is important to close the unused end of a pipe because otherwise the read end of the pipe will block STDIN.
☐ True ☐ False

It is important to close the unused end of a pipe because otherwise the tobacco will spill out of the pipe ends left open.
☐ True ☐ False

A read call will block if the pipe is empty.
☐ True ☐ False

A write call will block if the pipe is full.
☐ True ☐ False

A child inherits all the open file descriptors from the parent, including those corresponding to pipes.
☐ True ☐ False

Question 4. [8 MARKS]

Consider the following makefile:

```
FLAGS = -Wall -std=gnu99
DEPENDENCIES = hash.h ftree.h

all: fcopy

fcopy: fcopy.o ftree.o hash_functions.o
gcc ${FLAGS} -o $@ $^

%.o: %.c ${DEPENDENCIES}
gcc ${FLAGS} -c $<

clean:
rm *.o fcopy
```

gcc FLAGS -c ftree.c hash.h ftree.h

ftree.o

ftree.c

Assume the directory that contains the makefile contains the following files (and no others):

hash.h ftree.h ftree.c fcopy.c hash_functions.c.

Suppose the following commands are run one after the other. Fill in the table to show what files are created, deleted or modified as a result of running each command. If no files are created, deleted or modified after a particular command, write "NO CHANGE".

Command	Names of files created or changed	Names of files deleted
make ftree.o	ftree.o	
make fcopy	fcopy.o hash_functions.o fcopy	
rm ftree.o		ftree.o
make all	ftree.o fcopy	

Question 5. [9 MARKS]

Consider the program below and assume that it runs without encountering any errors.

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

void brain_pain(int signal) {
    printf("My brain is hurting!\n");
    printf("So much!\n");
}

int main() {

    printf("The final:\n");

    struct sigaction sa;
    sa.sa_handler = brain_pain;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    //  sigaddset(&sigset, SIGTERM);   // STATEMENT A
    sigaction(SIGINT, &sa, NULL);

    printf("I can't wait ...\n");
    kill(getpid(), SIGINT);
    printf("...until it's over.\n");

    exit(0);

}
```

Part (a) [3 MARKS]

Assuming that the program does not encounter any errors and does not receive any signals besides the ones it is sending itself. Decide for each of the following output sequences whether they could have been generated by the program.

```
The final:
I can't wait ...
My brain is hurting!
So much!
...until it's over.
```

☐ Possible ☐ Not Possible

```
The final:
I can't wait ...
My brain is hurting!
```

☐ Possible ☐ Not Possible

```
The final:
I can't wait ...
...until it's over.
```

☐ Possible ☐ Not Possible

```
The final:
I can't wait ...
My brain is hurting!
So much!
```

☐ Possible ☐ Not Possible

Part (b) [3 MARKS]

Now assume while the program is running the user sends a SIGTERM signal from the command line (e.g. using the kill command) at a random point during the execution. Which of the following outputs are possible?

```
The final:
I can't wait ...
My brain is hurting!
So much!
...until it's over.
```

☐ Possible ☐ Not Possible

```
The final:
I can't wait ...
My brain is hurting!
```

☐ Possible ☐ Not Possible

```
The final:
I can't wait ...
...until it's over.
```

☐ Possible ☐ Not Possible

```
The final:
I can't wait ...
My brain is hurting!
So much!
```

☐ Possible ☐ Not Possible

Part (c) [3 MARKS]

Now assume that we uncomment STATEMENT A (i.e. we remove the comment signs at the beginning of the line) and recompile. Then again while the program is running the user sends a SIGTERM signal from the command line (e.g. using the kill command) at a random point during the execution. Which of the following outputs are possible?

```
The final:
I can't wait ...
My brain is hurting!
So much!
...until it's over.
```

☐ Possible ☐ Not Possible

```
The final:
I can't wait ...
My brain is hurting!
```

☐ Possible ☐ Not Possible

```
The final:
I can't wait ...
...until it's over.
```

☐ Possible ☐ Not Possible

```
The final:
I can't wait ...
My brain is hurting!
So much!
```

☐ Possible ☐ Not Possible

Question 6. [9 MARKS]

The (incomplete) function `echo` below is part of a server program that serves as the middleman between two clients that want to communicate, while remaining anonymous to each other.

The server first establishes a connection (using sockets) with each of the two clients, and then echos data back and forth between the clients, i.e. it takes any data it reads from the first client and forwards it to the second client, and vice versa.

The `echo` function is a helper function used by the server program, after it has established the two connections with the clients. The function takes the file descriptors for the two client connections as arguments and implements the echoing of data between the two clients. The function should never block on a read and should not make any assumptions about the order in which the clients write data. It should keep running until one of the two clients closes their side of the connection.

Complete the function below. You can assume that no errors occur in any system calls you might make and omit error handling.

```
#define MAX_SIZE 128

void echo (int fd1, int fd2) {
    char buf[MAX_SIZE];
```

Question 7. [7 MARKS]

The program below expects as command line arguments an IP address and a port number, then establishes a connection to a server at that IP address and port, reads a message from the server and writes it to standard output. You can assume that all the proper libraries have been included. Unfortunately, the programmer made 7 mistakes when writing this program, which can lead to either compile-time errors/warnings or runtime errors.

```
0       #define MAX_SIZE 128
1
2       int main(int argc, char **argv) {
3               char *buf;
4
5               // create the socket
6               int soc = socket(AF_INET, 0);
7               if (soc < 0) {
8                   perror("socket");
9                  exit(1);
10              }
11
12              // set up the sockaddr_in struct for connecting
13              struct sockaddr_in peer;
14              peer.sin_port = htons(argv[2]);
15              if (inet_pton(AF_INET, argv[1], peer.sin_addr) < 1) {
16                  perror("inet_pton");
17                  close(soc);
18                  exit(1);
19              }
20
21              // connect the socket
22              if (connect(soc, (struct sockaddr *)&peer, 0) < -1) {
23                  perror("connect");
24                  close(soc);
25                  exit(1);
26              }
27
28              read(soc, buf, sizeof(buf));
29              write(STDOUT_FILENO, buf, MAX_SIZE);
30
31       }
```

Identify the mistakes and, on the next two pages, state for each mistake which line of the program is affected, and explain **briefly** what the mistake is. You can list the errors in any order.

Part (a) [2 MARKS]

Error 1:

Part (b) [2 MARKS]

Error 2:

Part (c) [2 MARKS]

Error 3:

Part (d) [2 MARKS]

Error 4:

Part (e) [2 MARKS]

Error 5:

Part (f) [2 MARKS]

Error 6:

Part (g) [2 MARKS]

Error 7:

(Extra space, if needed)

(Extra space, if needed)

Appendix

Useful structs

```
struct sockaddr_in {
    sa_family_t sin_family;
    unsigned short int sin_port;
    struct in_addr sin_addr;
    unsigned char pad[8]; /*Unused*/
}
```

Man page for sigemptyset, sigfillset, sigaddset, sigdelset, sigismember

SYNOPSIS

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signum);

int sigdelset(sigset_t *set, int signum);

int sigismember(const sigset_t *set, int signum);
```

These functions allow the manipulation of POSIX signal sets.

sigemptyset() initializes the signal set given by set to empty, with all signals excluded from the set.

sigfillset() initializes set to full, including all signals.

sigaddset() and sigdelset() add and delete respectively signal signum from set.

sigismember() tests whether signum is a member of set.

Objects of type sigset_t must be initialized by a call to either sigemptyset() or sigfillset() before being passed to the functions sigaddset(), sigdelset() and sigismember() or the additional glibc functions described below (sigisemptyset(), sigandset(), and sigorset()). The results are undefined if this is not done.

RETURN VALUE

sigemptyset(), sigfillset(), sigaddset(), and sigdelset() return 0 on success and -1 on error.

sigismember() returns 1 if signum is a member of set, 0 if signum is not a member, and -1 on error.

Man page for sigaction

SYNOPSIS

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

DESCRIPTION

The sigaction() system call is used to change the action taken by a process on receipt of a specific signal. (See signal(7) for an overview of signals.)

signum specifies the signal and can be any valid signal except SIGKILL and SIGSTOP.

If act is non-NULL, the new action for signal signum is installed from act. If oldact is non-NULL, the previous action is saved in oldact.

The sigaction structure is defined as something like:

```
struct sigaction {
    void     (*sa_handler)(int);
    sigset_t   sa_mask;
    int        sa_flags;
};
```

sa_handler specifies the action to be associated with signum and may be SIG_DFL for the default action, SIG_IGN to ignore this signal, or a pointer to a signal handling function. This function receives the signal number as its only argument.

sa_mask specifies a mask of signals which should be blocked (i.e., added to the signal mask of the thread in which the signal handler is invoked) during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the SA_NODEFER flag is used.

sa_flags specifies a set of flags which modify the behavior of the signal.

RETURN VALUE

sigaction() returns 0 on success and -1 on error.

Man page for select, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform the corresponding I/O operation (e.g., read(2)) without blocking.

Three independent sets of file descriptors are watched. Those listed in readfds will be watched to see if characters become available for reading (more precisely, to see if a read will not block; in particular, a file descriptor is also ready on end-of-file), those in writefds will be watched to see if a write will not block, and those in exceptfds will be watched for exceptions. On exit, the sets are modified in place to indicate which file descriptors actually changed status. Each of the three file descriptor sets may be specified as NULL if no file descriptors are to be watched for the corresponding class of events.

Four macros are provided to manipulate the sets. FD_ZERO() clears a set. FD_SET() and FD_CLR() respectively add and remove a given file descriptor from a set. FD_ISSET() tests to see if a file descriptor is part of the set; this is useful after select() returns.

nfds is the highest-numbered file descriptor in any of the three sets, plus 1.

The timeout argument specifies the minimum interval that select() should block waiting for a file descriptor to become ready. (This interval will be rounded up to the system clock granularity, and kernel scheduling delays mean that the blocking interval may overrun by a small amount.) If both fields of the timeval structure are zero, then select() returns immediately. (This is useful for polling.) If timeout is NULL (no timeout), select() can block indefinitely.

RETURN VALUE

On success, select() returns the number of file descriptors contained in the three returned descriptor sets (that is, the total number of bits that are set in readfds, writefds, exceptfds) which may be zero if the timeout expires before anything interesting happens. On error, -1 is returned, and errno is set appropriately; the sets and timeout become undefined, so do not rely on their contents after an error.

Man page for socket

SYNOPSIS

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

DESCRIPTION

socket() creates an endpoint for communication and returns a descriptor. The domain argument specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in ¡sys/socket.h¿. The currently understood formats include:

Name Purpose Man page AF_UNIX, AF_LOCAL Local communication unix(7) AF_INET IPv4 Internet protocols ip(7) AF_INET6 IPv6 Internet protocols ipv6(7) AF_IPX IPX - Novell protocols

The socket has the indicated type, which specifies the communication semantics. Currently defined types are:

SOCK_STREAM Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported. SOCK_DGRAM Supports datagrams (connectionless, unreliable messages of a fixed maximum length). SOCK_RAW Provides raw network protocol access.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the 'communication domain' in which communication is to take place; see protocols(5). See getprotoent(3) on how to map protocol name strings to protocol numbers.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. They do not preserve record boundaries. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect(2) call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) calls. When a session has been completed a close(2) may be performed. Out-of-band data may also be transmitted as described in send(2) and received as described in recv(2).

The communications protocols which implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered to be dead. When SO_KEEPALIVE is enabled on the socket the protocol checks in a protocol-specific manner if the other end is still alive. A SIGPIPE signal is raised if a process sends or receives on a broken stream; this causes naive processes, which do not handle the signal, to exit. SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read(2) calls will return only the amount of data requested, and any data remaining in the arriving packet will be discarded. Also all message boundaries in incoming datagrams are preserved.

RETURN VALUE

On success, a file descriptor for the new socket is returned. On error, -1 is returned, and errno is set appropriately.

Man page for connect and inet_pton

```
SYNOPSIS

#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr,            socklen_t
addrlen);
```

DESCRIPTION

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. The addrlen argument specifies the size of addr. The format of the address in addr is determined by the address space of the socket sockfd; see socket(2) for further details. If the socket sockfd is of type SOCK_DGRAM then addr is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type SOCK_STREAM or SOCK_SEQPACKET, this call attempts to make a connection to the socket that is bound to the address specified by addr.

Generally, connection-based protocol sockets may successfully connect() only once; connectionless protocol sockets may use connect() multiple times to change their association. Connectionless sockets may dissolve the association by connecting to an address with the sa_family member of sockaddr set to AF_UNSPEC (supported on Linux since kernel 2.2).

RETURN VALUE

If the connection or binding succeeds, zero is returned. On error, -1 is returned, and errno is set appropriately.

```
SYNOPSIS

#include <arpa/inet.h>

int inet_pton(int af, const char *src, void *dst);
```

DESCRIPTION

This function converts the character string src into a network address structure in the af address family, then copies the network address structure to dst. The af argument must be either AF_INET or AF_INET6.

inet_pton() returns 1 on success (network address was successfully converted). 0 is returned if src does not contain a character string representing a valid network address in the specified address family. If af does not contain a valid address family, -1 is returned and errno is set to EAFNOSUPPORT.

Man page for htonl, htons, ntohl, ntohs

SYNOPSIS

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

DESCRIPTION

The htonl() function converts the unsigned integer hostlong from host byte order to network byte order. The htons() function converts the unsigned short integer hostshort from host byte order to network byte order.

The ntohl() function converts the unsigned integer netlong from network byte order to host byte order.

The ntohs() function converts the unsigned short integer netshort from network byte order to host byte order.

On the i386 the host byte order is Least Significant Byte first, whereas the network byte order, as used on the Internet, is Most Significant Byte first.

Man page for fread

```
SYNOPSIS

#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmembFILE *" stream );

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

DESCRIPTION

The function fread() reads nmemb elements of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr. The function fwrite() writes nmemb elements of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr.

RETURN VALUE

On success, fread() and fwrite() return the number of items read or written. This number equals the number of bytes transferred only when size is 1. If an error occurs, or the end of the file is reached, the return value is a short item count (or zero). fread() does not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred.

Man page for read

```
SYNOPSIS

#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

DESCRIPTION

read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. On files that support seeking, the read operation commences at the current file offset, and the file offset is incremented by the number of bytes read. If the current file offset is at or past the end of file, no bytes are read, and read() returns zero.

If count is zero, read() may detect the errors described below. In the absence of any errors, or if read() does not check for errors, a read() with a count of 0 returns zero and has no other effects.

If count is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. On error, -1 is returned, and errno is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

Man page for write

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 pointed 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 handler 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 current 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) which can be proved to occur after a write() has returned returns the new data. Note that not all file systems are POSIX conforming.

RETURN VALUE

On success, the number of bytes written is returned (zero indicates nothing was written). 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, 0 will be returned without causing any other effect. If count is zero and fd refers to a file other than a regular file, the results are not specified.

Man page for strcpy and strncpy

```
SYNOPSIS
       #include <string.h>

       char *strcpy(char *dest, const char *src);

       char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION
       The  strcpy()  function  copies the string pointed to by src, including the
       terminating null byte, to the buffer pointed to by dest.  The strings
       may not overlap, and the destination string dest must be large enough to
       receive the copy.

       The strncpy() function is similar, except that at most n bytes of src are copied.
       Warning: If there is no null byte among the first n bytes of src, the
       string placed in dest will not be null terminated.

       If the length of src is less than n, strncpy() pads the remainder of dest with null bytes.


RETURN VALUE
       The strcpy() and strncpy() functions return a pointer to the destination string dest.
```

Man page for strlen

```
SYNOPSIS

       #include <string.h>
       size_t strlen(const char *s);

DESCRIPTION

       The strlen() function calculates the length of the string s, excluding the
       terminating null byte.

RETURN VALUE

       The strlen() function returns the number of bytes in the string s.
```

Man page for fopen

```
SYNOPSIS

        #include <stdio.h>

        FILE *fopen(const char *path, const char *mode);

DESCRIPTION

        The fopen() function opens the file whose name is the string pointed to by path and
        associates a stream with it.

        The argument mode points to a string beginning with one of the following sequences
        (possibly followed by additional characters, as described below):

        r
        Open text file for reading. The stream is positioned at the beginning of the file.

        r+
        Open for reading and writing. The stream is positioned at the beginning of the file.

        w
        Truncate file to zero length or create text file for writing. The stream is positioned
        at the beginning of the file.

RETURN VALUE

        Upon successful completion fopen() returns a FILE pointer.
        Otherwise, NULL is returned and errno is set to indicate the error.
```

Man page for fork

```
SYNOPSIS

       #include <unistd.h>
       pid_t fork(void);

DESCRIPTION

       fork() creates a new process by duplicating the calling process. The new process,
       referred to as the child, is an exact duplicate of the calling process, referred to
       as the parent, except for the following points:
       * The child has its own unique process ID, and this PID does not match the ID of
         any existing process group (setpgid(2)).
       * The child's parent process ID is the same as the parent's process ID.

RETURN VALUE

       On success, the PID of the child process is returned in the parent, and 0 is returned
       in the child. On failure, -1 is returned in the parent, no child process is created,
       and errno is set appropriately.
```

Man page for execl

```
SYNOPSIS

       #include <unistd.h>
       int execl(const char *path, const char *arg, ...);

DESCRIPTION

       The exec() family of functions replaces the current process image with a
       new process image. The initial argument for these functions is the name
       of a file that is to be executed.

       The const char *arg and subsequent ellipses in the execl() function can
       be thought of as arg0, arg1, ..., argn. Together they describe a list
       of one or more pointers to null-terminated strings that represent the
       argument list available to the executed program. The first argument,
       by convention, should point to the filename associated with the file
       being executed. The list of arguments must be terminated by a NULL pointer,
       and, since these are variadic functions, this pointer must be cast (char *) NULL.

RETURN VALUE

       The exec() functions only return if an error has occurred. The return value is -1, and
       errno is set to indicate the error.
```

Man page for scanf

```
SYNOPSIS

#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

DESCRIPTION

The scanf() family of functions scans input according to format as described below. This format may contain conversion specifications; the results from such conversions, if any, are stored in the locations pointed to by the pointer arguments that follow format. Each pointer argument must be of a type that is appropriate for the value returned by the corresponding conversion specification.

If the number of conversion specifications in format exceeds the number of pointer arguments, the results are undefined. If the number of pointer arguments exceeds the number of conversion specifications, then the excess pointer arguments are evaluated, but are otherwise ignored.

The scanf() function reads input from the standard input stream stdin, fscanf() reads input from the stream pointer stream, and sscanf() reads its input from the character string pointed to by str.

The format string consists of a sequence of directives which describe how to process the sequence of input characters. If processing of a directive fails, no further input is read, and scanf() returns. A "failure" can be either of the following: input failure, meaning that input characters were unavailable, or matching failure, meaning that the input was inappropriate.

Man page for malloc

```
SYNOPSIS

       #include <stdlib.h>
       void *malloc(size_t size);

DESCRIPTION

       The malloc() function allocates size bytes and returns a pointer to the
       allocated memory. The memory is not initialized. If size is 0, then malloc()
       returns either NULL, or a unique pointer value that can later be successfully
       passed to free().

RETURN VALUE

       The malloc() function return a pointer to the allocated memory that is suitably
       aligned for any kind of variable. On error, these functions return NULL. NULL
       may also be returned by a successful call to malloc() with a size of zero, or
       by a successful call to calloc() with nmemb or size equal to zero.
```

From the man page on signals

Each signal has a current disposition, which determines how the process behaves when it is delivered the signal.

The entries in the "Action" column of the tables below specify the default disposition for each signal, as follows:

```
Term   Default action is to terminate the process.

Ign    Default action is to ignore the signal.

Core   Default action is to terminate the process and dump core (see
       core(5)).

Stop   Default action is to stop the process.

Cont   Default action is to continue the process if it is currently
       stopped.
```

Linux supports the standard signals listed below. Several signal numbers are architecture-dependent, as indicated in the "Value" column. (Where three values are given, the first one is usually valid for alpha and sparc, the middle one for x86, arm, and most other architectures, and the last one for mips. (Values for parisc are not shown; see the Linux kernel source for signal numbering on that architecture.) A dash (-) denotes that a signal is absent on the corresponding architecture.

First the signals described in the original POSIX.1-1990 standard.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating-point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process

```
SIGTSTP   18,20,24    Stop    Stop typed at terminal
SIGTTIN   21,21,26    Stop    Terminal input for background process
SIGTTOU   22,22,27    Stop    Terminal output for background process
```

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.