CSC B09H1 S 2017 Final Exam
Duration — 150 minutes
Aids allowed: Handwritten two-page double-sided crib sheet

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

Utorid: _____

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 final exam consists of 7 questions on 28 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: _____/ 5
2: _____/10
3: _____/10
4: _____/10
5: _____/11
6: _____/ 9
7: _____/12

TOTAL: _____/67

Question 1. [5 MARKS]

Part (a)

A parent process and a child share the same address space.

☐ True ☐ False

Part (b)

A process can call wait to wait for a process that it is not related to.

☐ True ☐ False

Part (c)

Signals can be used for communication between processes on different machines.

☐ True ☐ False

Part (d)

Internet routers guarantee reliable delivery of network packets.

☐ True ☐ False

Part (e)

TCP is more efficient, but less reliable than UDP.

☐ True ☐ False

Part (f)

A string's length (as indicated by strlen) and its size (as indicated by sizeof) are always equal.

☐ True ☐ False

Part (g)

Pipes can be used for communication between processes on different machines.

☐ True ☐ False

Part (h)

Sockets can be used for communication between processes running on the same machine.

☐ True ☐ False

Question 2. [10 MARKS]

Part (a) [2 MARKS]

Considering the following piece of code, fill the table below with the values of the array elements after the code is done executing. Be careful with the difference between pointers and values, and pointer arithmetic.

```
int a[4] = {0, 1, 2, 3};
int b = 1;
int *p = a;

p = p + b;
b++;
*p += b;
p = p + b;
*p += 2;
p--;
*p *= 4;
p = p - b;
*p = p - a;
```

a[0]	a[1]	a[2]	a[3]

Part (b) [2 MARKS]

What is the output of the following program?

```
#include <stdio.h>

int func(int a, int *b, int *c) {
    a += 5;
    *b += a;
    c = b;
    return a;
}

int main() {
    int x = 5, y = 8, z = 3, t = 0;
    t = func(x, &y, &z);

    printf("x:%d    y:%d    z:%d    t:%d\n", x, y, z, t);
    return 0;
}
```

Answer:

Part (c) [4 MARKS]

There are several errors in the following program. Use the appropriate letter to label the lines of code where the corresponding error occurs.
P - dereferencing a pointer without having allocated memory for it.
D - deallocating memory that has already been deallocated.
H - deallocating memory that is not located on the heap.
M - memory leak.

```
int a = 0, b = 1, c = 2;
int *p, *q, *r;

p = malloc(sizeof(int));
*p = b;
q = &c;
p = &a;
*q = b;
q = malloc(sizeof(10));
*r = a + b + c;
free(p);
r = q;
p = malloc(sizeof(int));
*p = *r;
free(q);
free(r);
free(p);
```

Part (d) [2 MARKS]

What does the following piece of code print?

```
struct Point {
    int x;
    int y;
};

struct Point p1, p2;
struct Point *q1, *q2;

p1.x = 5;  p1.y = 10;
p2.x = 1;  p2.y = 10;

q1 = &p2;
q2 = &p1;
q1->x += 2;    q1->y += 7;
q2->x += 4;    q2->y += 3;

printf("P1(X,Y) = (%d, %d)\n", p1.x, p1.y);
printf("P2(X,Y) = (%d, %d)\n", p2.x, p2.y);
```

Answer:

Question 3. [10 MARKS]

Each example below contains an independent code fragment. In each case, there are variables x and y that are missing declaration statements. In the boxes to the right of the code, write those declaration statements so that the code fragment would compile and run without warnings or errors. If there is no declaration that could lead to a compilation without warnings or errors, write "ERROR". The first is done for you as an example.

Code Fragment	**Declaration for x**	**Declaration for y**
`x = 10;` `y = 'A';`	`int x;`	`char y;`

`double length = 25;`
`x = &length;`
`y = &x;`		

`char *id[6];`
`x = id[3];`
`// some hidden code`
`id[3] = "c3new";`
`y = *x[3];`		

`char *name = "John Tory";`
`x = &name;`
`y = *(name + 3);`		

`struct node {`
`  int value;`
`  struct node *next;`
`};`
`typedef struct node List;`
`List *head;`
`// some hidden code`
`x = head->next;`
`y.value = 14;`
`y.next = x;`		

`char fun(char *str, int n) {`
`  return str[n];`
`}`
 	
`y = fun("hello", 1);`
`x = &y;`		

CONT'D...

Question 4. [10 MARKS]

Some of the code fragments below have a problem. For each fragment indicate whether the code works as intended or there is an error (logical error, compile-time error/warning, or runtime error). Assume all programs are compiled using the *C*99 standard. For this question, we will 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. For parts where there is an error you will only receive marks if you can correctly explain what is wrong. We have intentionally omitted the error checking of the system calls to simplify the examples. Do **not** report this as an error.

Some of the parts will use the following struct definition:

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

```
char *s = "Hello";
strcat(s, ", World!");
```

☐ Works as intended ☐ Error

```
int main(int argc, char **argv) {
    char ch;
    char *p = &ch;
    ch = argv[argc-1][0];
    printf("%c\n", p[0]);
    return 0;
}
```

☐ Works as intended ☐ Error

```
struct student hannah;
strcpy(hannah.name, "Hannah");
```

☐ Works as intended ☐ Error

```
struct student hannah = NULL;
// ... missing code ...
if (hannah != NULL) {
    hannah.age = 10;
}
```

☐ Works as intended ☐ Error

```
// 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);
    return 0;
}
```

☐ Works as intended ☐ Error

```
// Compute the sum of an array of integers
int compute_sum(int numbers[]) {
    int sum = 0;
    for (int i = 0; i < sizeof(numbers); i++) {
        sum += numbers[i];
    }
    return sum;
}
```

☐ Works as intended ☐ Error

```
int fd[2];
int result = fork();
pipe(fd);
if (result == 0) {
    close(fd[0]);
    write(fd[1], "cscb09", 7);
}
else {
    close(fd[1]);
    char buf[7];
    read(fd[0], buf, 7);
    printf("%s\n", buf);
}
exit(0);
```

☐ Works as intended ☐ Error

```
// Read all bytes from the file descriptors in ‘fds’ as characters,
// and print them. ‘num_fds’ is the number of file descriptors, and
// ‘max_fd’ is the value of the largest one.
void read_ints(int *fds, int num_fds, int max_fd) {
    char data;
    fd_set set;
    FD_ZERO(&set);
    for (int i = 0; i < num_fds; i++) {
        FD_SET(fds[i], &set);
    }

    while (select(max_fd + 1, &set, NULL, NULL, NULL) > 0) {
        for (int i = 0; i < num_fds; i++) {
            if (FD_ISSET(fds[i], &set)) {
                if (read(sum, &data, 1) > 0) {
                    printf("%c\n", data);
                }
            }
        }
    }
}
```

☐ Works as intended ☐ Error

```
struct node {
    int item;
    struct node *next;
};

// Compute the sum of the items in a linked list with the given head,
// but do not modify the list.
int sum(struct node *head) {
    int s = 0;
    while (head != NULL) {
        s += head->item;
        *head = *(head->next);
    }
    return s;
}
```

☐ Works as intended ☐ Error

```
// Remove the dots from word
char *word = "Ex.ampl.e";
char *result = malloc(strlen(word) + 1); // upper-limit if word has no dots
for (int i = 0; i < strlen(word); i++) {
    if (word[i] != '.') {
        strncat(result, word[i], 1);
    }
}
```

☐ Works as intended ☐ Error

Question 5. [11 MARKS]

Assume a linked list structure, which contains some information about a student, as follows:

```
typedef struct student {
    int student_number;
    char *last_name;
    struct student *next;
} Student;
```

Write a function that traverses a given student list and inserts a new student in alphabetical order (function prototype given below). If the new student has the same last name as another student from the list, insert in ascending order of the student number.

Note: The new student name passed to the function may be deallocated once the function exits, so make sure to create a new copy of the student name when inserting.

```
Student *insert_new_student(Student *list, int newStudentNo, char *newName) {

}
```

Question 6. [9 MARKS]

Consider the following program that runs to completion without errors:

```
int main() {
    int var = 1;
    int status;

    int r = fork();
    if(r == 0) {
        var++;
        r = fork();
        if(r == 0) { // process X
            var++;
            exit(var);
        } else { // process Y
            if(wait(&status) != -1) {
                if(WIFEXITED(status)) {
                    printf("A %d ", WEXITSTATUS(status));
                }
            }
            var += 2;
        }
    } else { // process Z
        printf("W %d ", var);
        if(wait(&status) != -1) {
            if(WIFEXITED(status)) {
                printf("B %d ", WEXITSTATUS(status));
            }
        }
    }
    printf("C %d ", var);
    return 0;
}
```

Part (a) [5 MARKS]

Write all the possible output orders for this program.

Part (b) [2 MARKS]

If all processes run to completion, can process X or process Y become an orphan? If yes, explain the sequence of events that would cause the process to become an orphan. If no, then explain what modification would need be made to the code so that a process might become an orphan. Clearly identify which process is the orphan.

Part (c) [2 MARKS]

If all processes run to completion, can process X or process Y become a zombie? If yes, explain the sequence of events that would cause the process to become a zombie. If no, then explain what modification would need be made to the code so that a process might become a zombie. Clearly identify which process is the zombie.

Question 7. [12 MARKS]

Implement a "parrot" server, which receives messages from clients and repeats each message back to the client who originally sent it. The server communicates with clients using *reliable* connections and uses the *I/O multiplexing* model.

Once a client connection is established, the server adds the new communication channel (socket) to a local client array to keep track of it (the code for initializing this is provided). Any new messages received from such a client will be repeated back to the client over the corresponding communication channel. The server does *not* timeout while waiting for client connections.

Notes:

- Use the API sheet. Syntax does not have to be perfect, but **do not write pseudocode**.
- Explain in comments all design decisions (size of queue for pending connections, etc.).
- Some code is already provided to guide you: extracting the port, initializing the array that will keep track of the sockets for clients, initializing file descriptor sets, etc. Have a look over the next page before starting.

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

    // Extract port form command line.
    int portno = strtol(argv[1], NULL, 10);

    // Initialize array of client socket descriptors.
    int i, clients[FD_SETSIZE];
    for(i = 0; i < FD_SETSIZE; i++)
        clients[i] = -1;

    // Fill server information.
    struct sockaddr_in serveraddr;
    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons((unsigned short) portno);

    // Add server operations for incoming client connections.
    int serverfd;
    ...
```

```
    // Add serverfd to read set, set maxfd.
    fd_set rset, allset;
    FD_ZERO(&allset);
    FD_SET(serverfd, &allset);
    int maxfd = serverfd;

    // Write the main server loop.
    for( ; ; ) {









    } // end main server loop.

    close(serverfd);
    return 0;
} // end main.
```

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 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

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 listen

```
SYNOPSIS

int listen(int sockfd, int backlog);

DESCRIPTION

listen() marks the socket referred to by sockfd as a passive socket,
that is, as a socket that will be used to accept incoming connection
requests using accept(2).

The sockfd argument is a file descriptor that refers to a socket of
type SOCK_STREAM or SOCK_SEQPACKET.

The backlog argument defines the maximum length to which the queue of
pending connections for sockfd may grow.  If a connection request
arrives when the queue is full, the client may receive an error with
an indication of ECONNREFUSED or, if the underlying protocol supports
retransmission, the request may be ignored so that a later reattempt
at connection succeeds.

RETURN VALUE

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

Man page for bind

```
SYNOPSIS

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

DESCRIPTION

When a socket is created with socket(2), it exists in a name space
(address family) but has no address assigned to it.  bind() assigns
the address specified by addr to the socket referred to by the file
descriptor sockfd.  addrlen specifies the size, in bytes, of the
address structure pointed to by addr.  Traditionally, this operation
is called assigning a name to a socket.

It is normally necessary to assign a local address using bind()
before a SOCK_STREAM socket may receive connections (see accept(2)).

The rules used in name binding vary between address families.

The actual structure passed for the addr argument will depend on the
address family.  The sockaddr structure is defined as something like:

struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
}

The only purpose of this structure is to cast the structure pointer
passed in addr in order to avoid compiler warnings.  See EXAMPLE
below.

RETURN VALUE

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

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 strcat and strncat

```
SYNOPSIS

#include <string.h>

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

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

DESCRIPTION

The strcat() function appends the src string to the dest string, overwriting the terminating null

The strncat() function is similar, except that

*
it will use at most n bytes from src; and
*
src does not need to be null-terminated if it contains n or more bytes.
```

```
As with strcat(), the resulting string in dest is always null-terminated.

If src contains n or more bytes, strncat() writes n+1 bytes to dest (n from src plus the terminati
```

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 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.
```

Man page for wait

```
Name

wait - wait for process to change state

Synopsis

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

pid_t wait(int *status);

All of these system calls are used to wait for state changes in a child of the calling
process, and obtain information about the child whose state has changed. A state change
is considered to be: the child terminated; the child was stopped by a signal; or the child
was resumed by a signal. In the case of a terminated child, performing a wait allows the
system to release the resources associated with the child; if a wait is not performed, then
```

```
the terminated child remains in a "zombie" state.

If a child has already changed state, then these calls return immediately. Otherwise they
block until either a child changes state or a signal handler interrupts the call (assuming
that system calls are not automatically restarted using the SA_RESTART flag of sigaction(2)).
In the remainder of this page, a child whose state has changed and which has not yet been waited
upon by one of these system calls is termed waitable.


The wait() system call suspends execution of the calling process until one of its
children terminates.

WIFEXITED(status)
returns true if the child terminated normally, that is, by calling exit(3) or _exit(2),
or by returning from main().
WEXITSTATUS(status)
returns the exit status of the child. This consists of the least significant 8 bits of
the status argument that the child specified in a call to exit(3) or _exit(2) or as the
argument for a return statement in main(). This macro should only be employed if WIFEXITED
returned true.


RETURN VALUE
wait(): on success, returns the process ID of the terminated child; on error, -1 is returned.
```