What is the difference between the string processing function `scanf()` and the system call function `read()`?

- `scanf()`:
  - Is a C standard library function (part of `stdio.h`).
  - It's used for formatted input from `stdin` (or other streams specified by `fscanf`).
  - It parses input based on a format string (e.g., `%d`, `%s`, `%f`) and can convert input into different data types.
  - It is typically buffered, meaning it might read a block of data into a buffer and then process it.
  - It often stops reading for certain format specifiers (like `%s`) when it encounters whitespace.
- `read()`:
  - Is a POSIX standard system call (part of `unistd.h`).
  - It's used to read a raw stream of bytes from a file descriptor.
  - It does not interpret the data's format; it simply reads a specified number of bytes from the kernel into a user-supplied buffer.
  - It is generally unbuffered (or more accurately, it interacts directly with the kernel's buffers, and user-level buffering isn't directly controlled unless using standard I/O wrappers).
  - It returns the number of bytes actually read.

What does the numeric value returned by the `read()` function represent?

- The read()

  function returns an integer value that represents:
  - **A positive number**: The number of bytes successfully read. This value might be less than the number of bytes requested (e.g., if the end of the file was reached before reading the requested count, or if reading from a pipe with insufficient data).
  - **Zero (0)**: Indicates that the end of the file (EOF) has been reached. For pipes or sockets, this usually means the writing end has been closed.
  - **Minus one (-1)**: Indicates that an error occurred. The specific error code is stored in the global variable `errno`, and `perror()` can be used to print a human-readable error message.

What is a process descriptor table?

- A process descriptor table (more commonly referred to as a file descriptor table) is a data structure maintained by the operating system kernel for each process.
- This table keeps track of all the files, pipes, sockets, and other I/O resources that the process has opened.
- Each entry in the table is a file descriptor (a small, non-negative integer, like 0, 1, 2, ...).
- Each file descriptor points to an entry in a system-wide open file table, which contains information about the open file (like the current file offset, access modes, etc.) and a pointer to the actual file (i-node) in the file system.

- For example, file descriptor 0 is typically standard input (stdin), 1 is standard output (stdout), and 2 is standard error (stderr).

What is the purpose of the `open()` system call?

- The primary purpose of the `open()` system call is to open or create a file (or device) and return a file descriptor for that file.
- This file descriptor can then be used by other I/O system calls (like `read()`, `write()`, `close()`, `lseek()`, etc.) to perform operations on the file.
- The `open()` call allows specifying the file name, the mode in which to open it (e.g., read-only, write-only, read-write, append), and permissions if a new file is being created.

Why do you need to use the `close()` system call?

- Using the close()

  system call is important for several reasons:

  - **Resource Release**: Each process has a limit on the number of file descriptors it can have open. Closing a file descriptor when it's no longer needed releases it back to the process's pool of available descriptors, allowing it to be reused by subsequent `open()` calls.
  - **Ensuring Data Integrity**: For output files, `close()` often ensures that any data buffered in memory is flushed (written) to the disk. If a file is not closed, some data might be lost.
  - **Updating File Metadata**: Closing a file allows the operating system to update file metadata, such as the last modification time.
  - **Allowing File Deletion/Unmounting**: On some systems, a file cannot be deleted, or its filesystem cannot be unmounted if it is still open by a process.
  - **Pipe and FIFO Communication**: When using pipes for inter-process communication, properly closing the unused ends of the pipe (e.g., the writer closes its read end, the reader closes its write end) is crucial for correctly detecting EOF (when the write end is closed) or avoiding deadlocks.

Ques a

cc p4a.c -o p4a
./p4a

```
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$ ./p4a
I am the parent
Which process am I?
I am the child
```

Ques b

cc p4b.c -o p4b

./p4b

```
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$ cc p4b.c -o p4b
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$ ./p4b
I am the parent
Which process am I?
I am the child
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$ Which process am I?
```

Ques c

cc p4c.c -o p4c

./p4c

```
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$ ./p4c
 lab04.zip   p4a    p4a.c   p4b    p4b.c   p4c    p4c.c   p4d.c   p4e.c   p4f.c   p4g.c   p4h.c  'Practical 4.pdf'
Child 2488198 complete
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$
```

Ques d

cc p4d.c -o p4d

./p4d

```
root         127  0.0  0.0      0     0 ?        I<   4月08   0:00 [kworker/18:0H-events_highpri]
root         128  0.0  0.0      0     0 ?        S    4月08   0:00 [cpuhp/19]
root         129  0.0  0.0      0     0 ?        S    4月08   0:00 [idle_inject/19]
root         130  0.0  0.0      0     0 ?        S    4月08   0:13 [migration/19]
root         131  0.0  0.0      0     0 ?        S    4月08   0:13 [ksoftirqd/19]
root         133  0.0  0.0      0     0 ?        I<   4月08   0:00 [kworker/19:0H-events_highpri]
root         134  0.0  0.0      0     0 ?        S    4月08   0:00 [cpuhp/20]
root         135  0.0  0.0      0     0 ?        S    4月08   0:00 [idle_inject/20]
root         136  0.0  0.0      0     0 ?        S    4月08   0:13 [migration/20]
root         137  0.0  0.0      0     0 ?        S    4月08   0:05 [ksoftirqd/20]
root         139  0.0  0.0      0     0 ?        I<   4月08   0:00 [kworker/20:0H-kblockd]
root         141  0.0  0.0      0     0 ?        S    4月08   0:00 [cpuhp/21]
root         142  0.0  0.0      0     0 ?        S    4月08   0:00 [idle_inject/21]
root         143  0.0  0.0      0     0 ?        S    4月08   0:14 [migration/21]
root         144  0.0  0.0      0     0 ?        S    4月08   0:05 [ksoftirqd/21]
root         146  0.0  0.0      0     0 ?        I<   4月08   0:00 [kworker/21:0H-events_highpri]
root         147  0.0  0.0      0     0 ?        S    4月08   0:00 [cpuhp/22]
root         148  0.0  0.0      0     0 ?        S    4月08   0:00 [idle_inject/22]
root         149  0.0  0.0      0     0 ?        S    4月08   0:13 [migration/22]
root         150  0.0  0.0      0     0 ?        S    4月08   0:05 [ksoftirqd/22]
root         152  0.0  0.0      0     0 ?        I<   4月08   0:00 [kworker/22:0H-events_highpri]
root         153  0.0  0.0      0     0 ?        S    4月08   0:00 [cpuhp/23]
root         154  0.0  0.0      0     0 ?        S    4月08   0:00 [idle_inject/23]
root         155  0.0  0.0      0     0 ?        S    4月08   0:14 [migration/23]
root         156  0.0  0.0      0     0 ?        S    4月08   0:05 [ksoftirqd/23]
root         158  0.0  0.0      0     0 ?        I<   4月08   0:00 [kworker/23:0H-kblockd]
root         159  0.0  0.0      0     0 ?        S    4月08   0:00 [cpuhp/24]
root         160  0.0  0.0      0     0 ?        S    4月08   0:00 [idle_inject/24]
root         161  0.0  0.0      0     0 ?        S    4月08   0:14 [migration/24]
root         162  0.0  0.0      0     0 ?        S    4月08   0:05 [ksoftirqd/24]
root         164  0.0  0.0      0     0 ?        I<   4月08   0:00 [kworker/24:0H-events_highpri]
root         165  0.0  0.0      0     0 ?        S    4月08   0:00 [cpuhp/25]
root         166  0.0  0.0      0     0 ?        S    4月08   0:00 [idle_inject/25]
root         167  0.0  0.0      0     0 ?        S    4月08   0:14 [migration/25]
root         168  0.0  0.0      0     0 ?        S    4月08   0:04 [ksoftirqd/25]
root         170  0.0  0.0      0     0 ?        I<   4月08   0:00 [kworker/25:0H-events_highpri]
root         171  0.0  0.0      0     0 ?        S    4月08   0:00 [cpuhp/26]
root         172  0.0  0.0      0     0 ?        S    4月08   0:00 [idle_inject/26]
root         173  0.0  0.0      0     0 ?        S    4月08   0:13 [migration/26]
```

Ques e

cc p4e.c -o p4e

./p4e

```
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$ ./p4e
read descriptor = 3, write descriptor = 4, buffersize = 5
 Read =Welco
 Read =me to
 Read = Unix
 Read = pipe
 Read =s
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$
```

Ques f

cc p4f.c -o p4f

./p4f

```
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$ ./p4f
read descriptor = 3, write descriptor = 4,buffersize = 10
 Read =Welcome to
 Read = Unix pipe
 Read =s
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$
```

Ques g

cc p4g.c -o p4g

./p4g

```
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$ ./p4g
Parent writing: Welcome to Unix pipes
Child read (21 bytes): Welcome to Unix pipes
Parent writing: Welcome to Unix pipes
Child read (21 bytes): Welcome to Unix pipes
Parent writing: Welcome to Unix pipes
Child read (21 bytes): Welcome to Unix pipes
```

Ques h

cc p4h.c -o p4h

./p4h

```
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$ cc p4h.c -o p4h
(base) fzdx@fzdx-PR4910P:~/cjt/CS240/practical4$ ./p4h
Parent writing: Welcome to Unix pipes
Child read (21 bytes): Welcome to Unix pipes
Parent writing: Welcome to Unix pipes
Child read (21 bytes): Welcome to Unix pipes
Parent writing: Welcome to Unix pipes
Child read (21 bytes): Welcome to Unix pipes
```