CIT 5950 Recitation 2

I/O, POSIX, and System Calls!

Logistics

Due Next Friday:

Homework 1 @ 11:59 pm

POSIX

Posix is a family of standards specified by the IEEE. These standards maintains compatibility across variants of Unix-like operating systems by defining APIs and standards for basic I/O (file, terminal, and network) and for threading.

1. What does POSIX stand for?

   Portable Operating System Interface

1.` Why might a POSIX standard be beneficial? From an application perspective? Versus using the C stdio library?

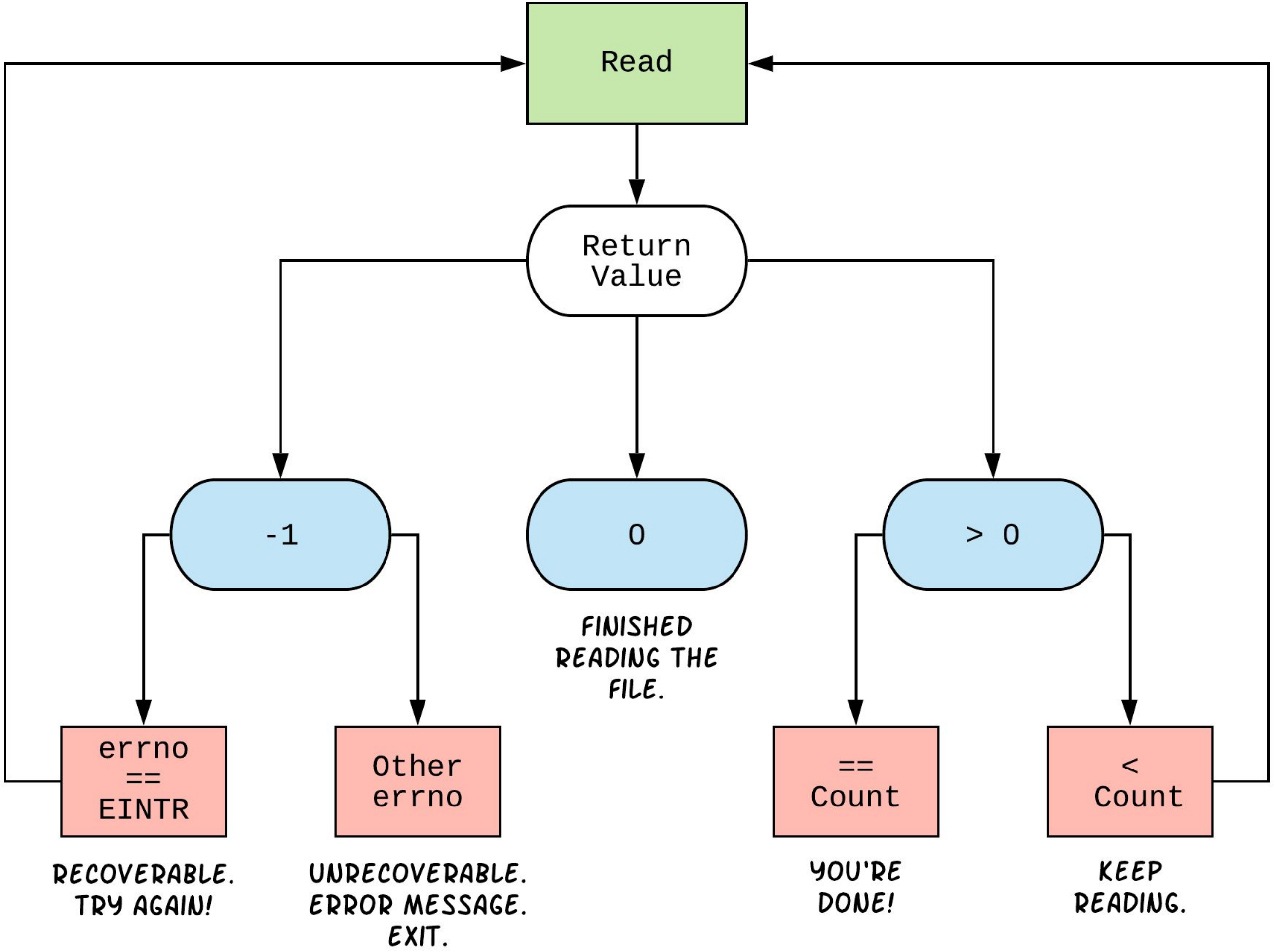
- **More explicit control since read and write functions are system calls and you can directly access system resources.**
- **POSIX calls are unbuffered so you can implement your own buffer strategy on top of read()/write().**
- **There is no standard higher level API for network and other I/O devices**

Review from Lecture

```
ssize_t read(int fd, void *buf, size_t count)
```

An error occurred	`result = -1`
	`errno = error`

Already at EOF	`result = 0`
Partial Read	`result < count`
Success!	`result == count`



Read

Return Value

-1

0

> 0

FINISHED READING THE FILE.

errno == EINTR

RECOVERABLE. TRY AGAIN!

Other errno

UNRECOVERABLE. ERROR MESSAGE. EXIT.

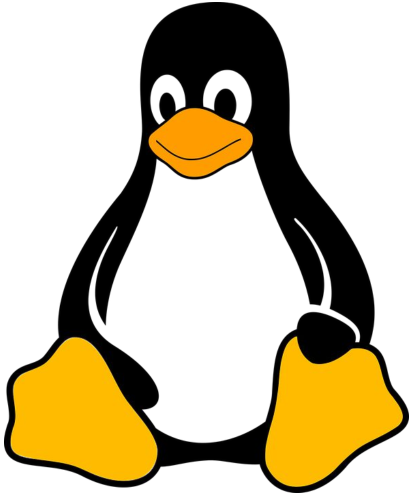
== Count

YOU'RE DONE!

< Count

KEEP READING.

New Scenario - Messy Roommate



- The Linux kernel now lives with you in room #595
- There are N pieces of trash in the room
- There is a single trash can, `char bin[N]`
    - (For some reason, the trash goes in a particular order)
- You can tell your roommate to pick it up, but he/she is unreliable

New Scenario - Messy Roommate

```
NumTrash pickup(roomNum, trashBin, Amount)
```

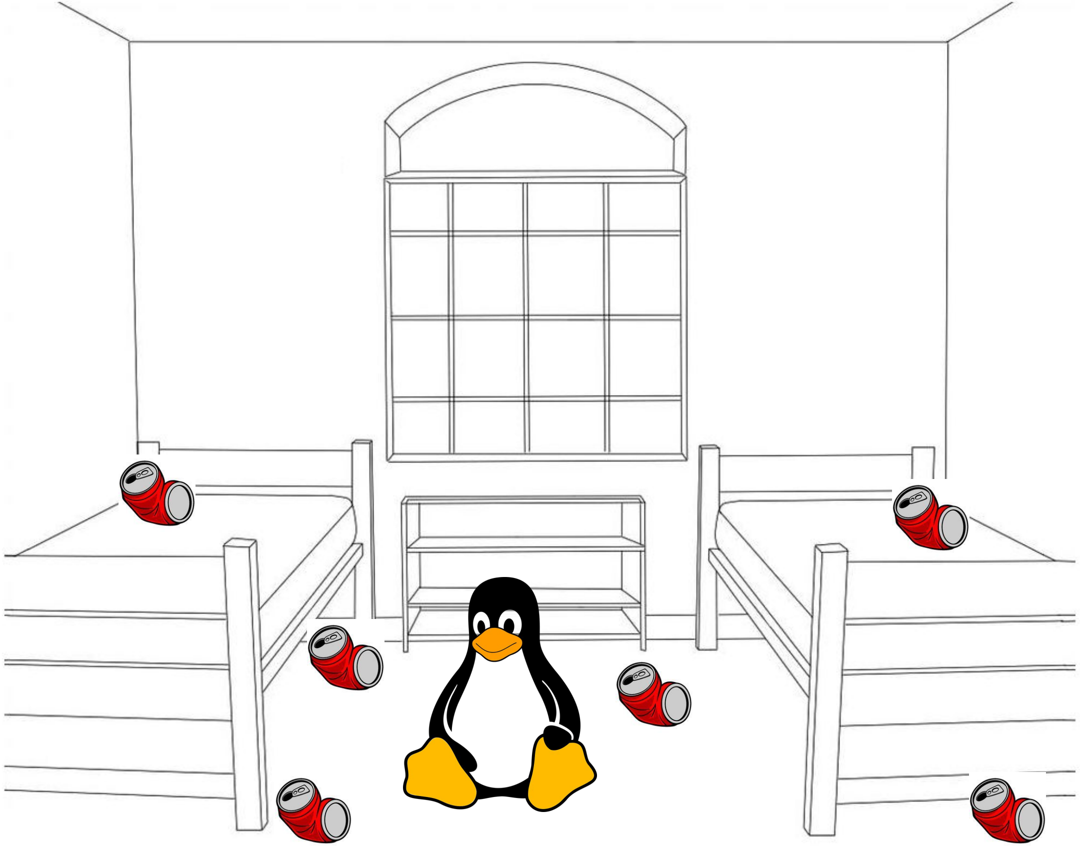
"*I tried to start cleaning, but something came up*"
(got hungry, had a midterm, room was locked, etc.)
	`NumTrash == -1`
	`errno == excuse`

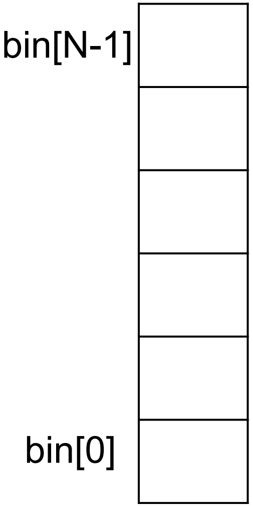
"*You told me to pick up trash, but the room was already clean*"	`NumTrash == 0`
"*I picked up some of it, but then I got distracted by my favorite show on Netflix*"	`NumTrash < Amount`
"*I did it! I picked up all the trash!*"	`NumTrash == Amount`

How do we get room 595 clean?

```
NumTrash pickup(roomNum, trashBin, Amount)
```

NumTrash == -1, errno == excuse
NumTrash == 0
NumTrash < Amount
NumTrash == Amount



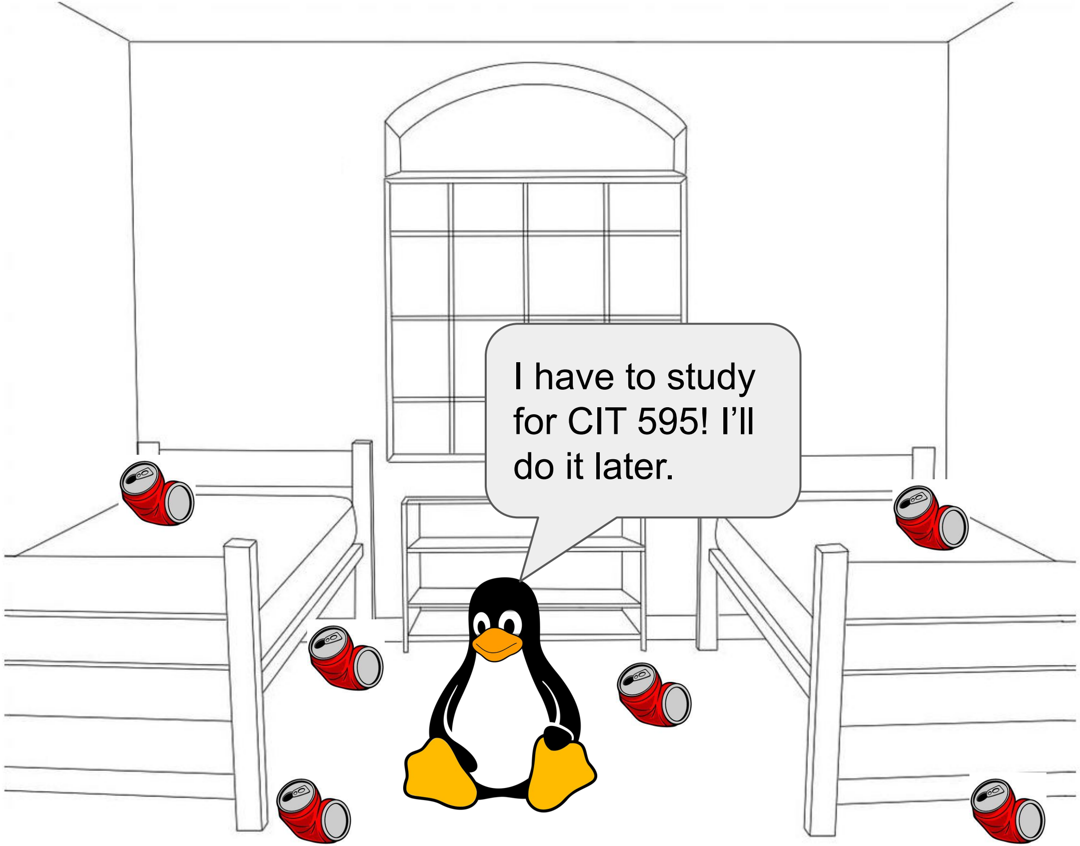


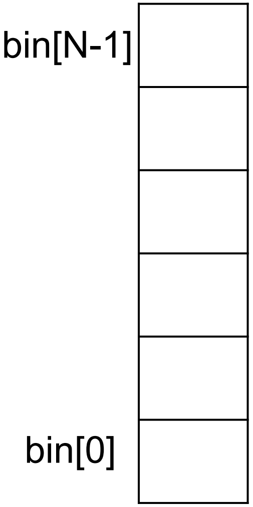
What do we do in the following scenarios?

How do we get room 595 clean?

```
NumTrash pickup(roomNum, trashBin, Amount)
```

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`



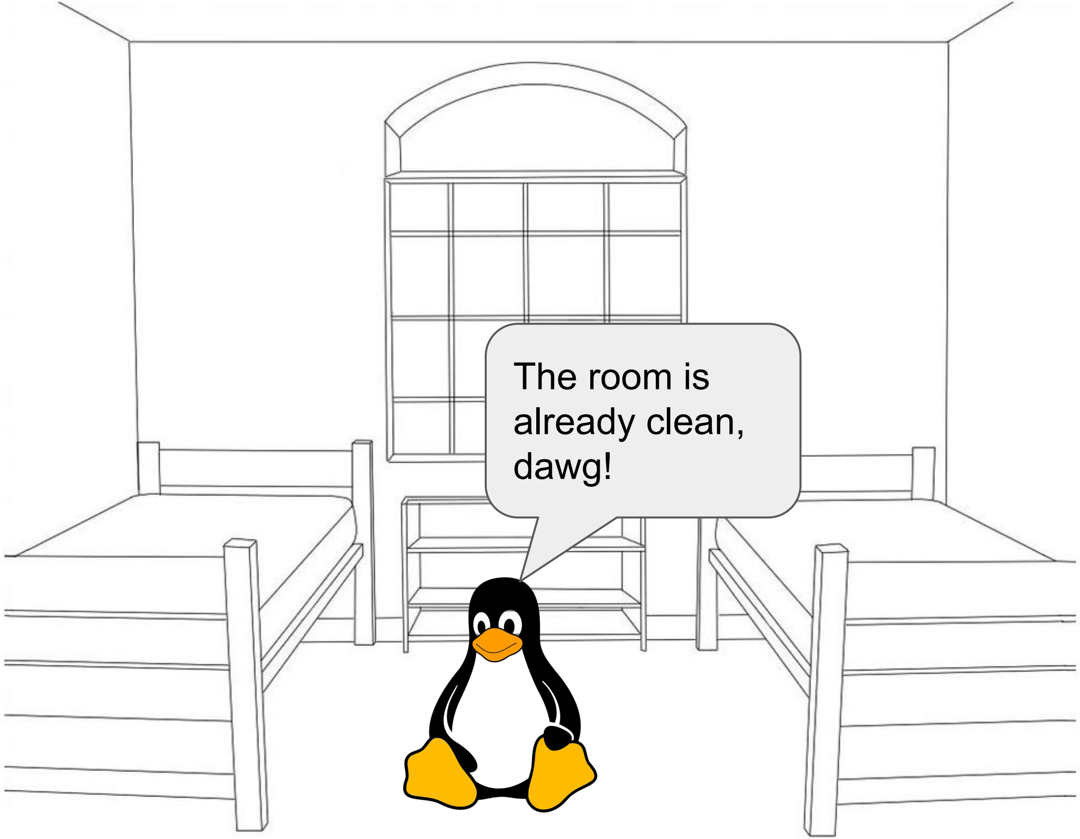


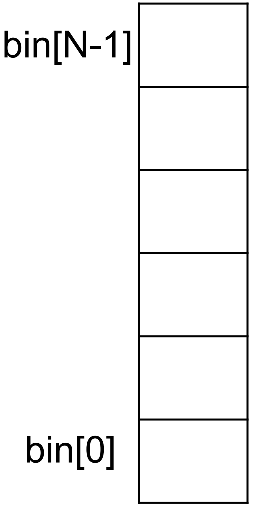
Decide if the excuse is reasonable, and either let it be or ask again.

How do we get room 595 clean?

```
NumTrash pickup(roomNum, trashBin, Amount)
```

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`



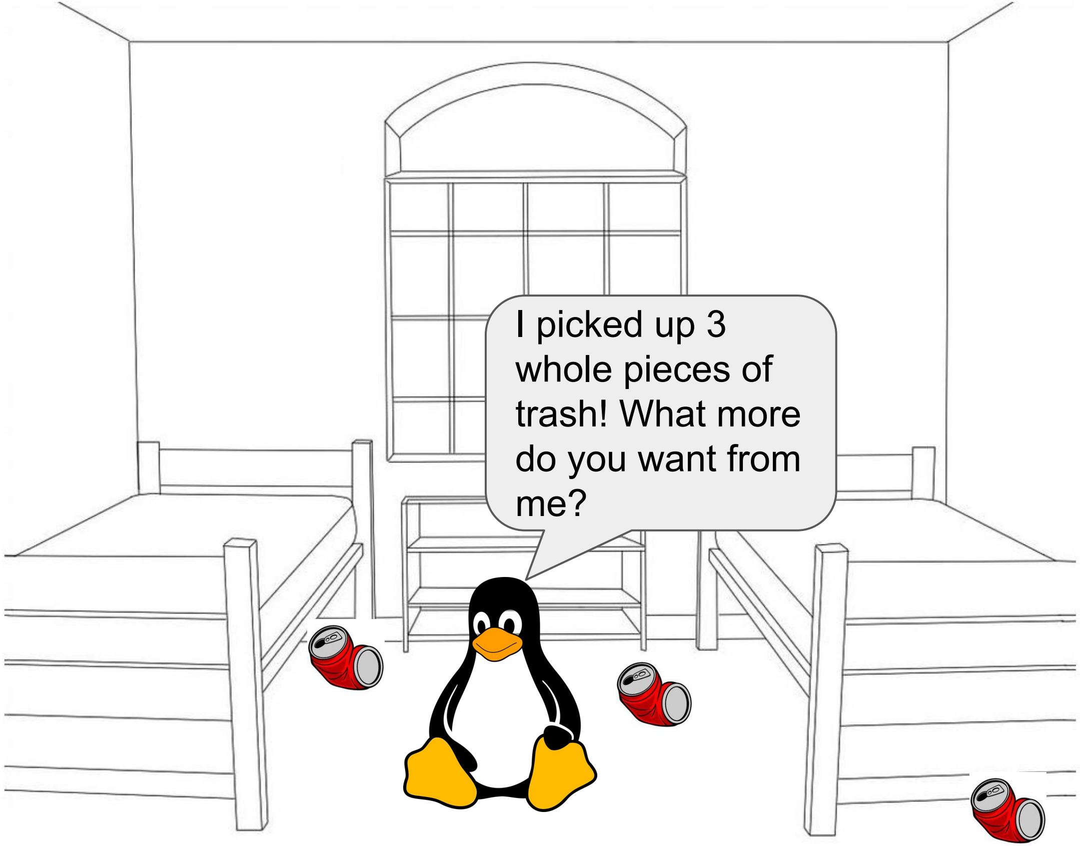


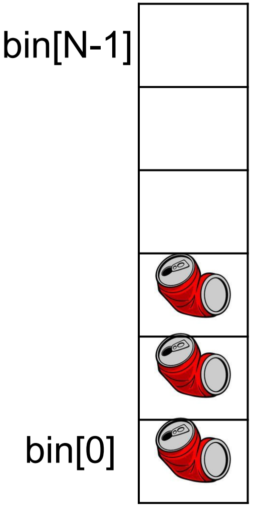
Stop asking them to clean the room! There’s nothing to do.

How do we get room 595 clean?

```
NumTrash pickup(roomNum, trashBin, Amount)
```

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`


I picked up 3 whole pieces of trash! What more do you want from me?

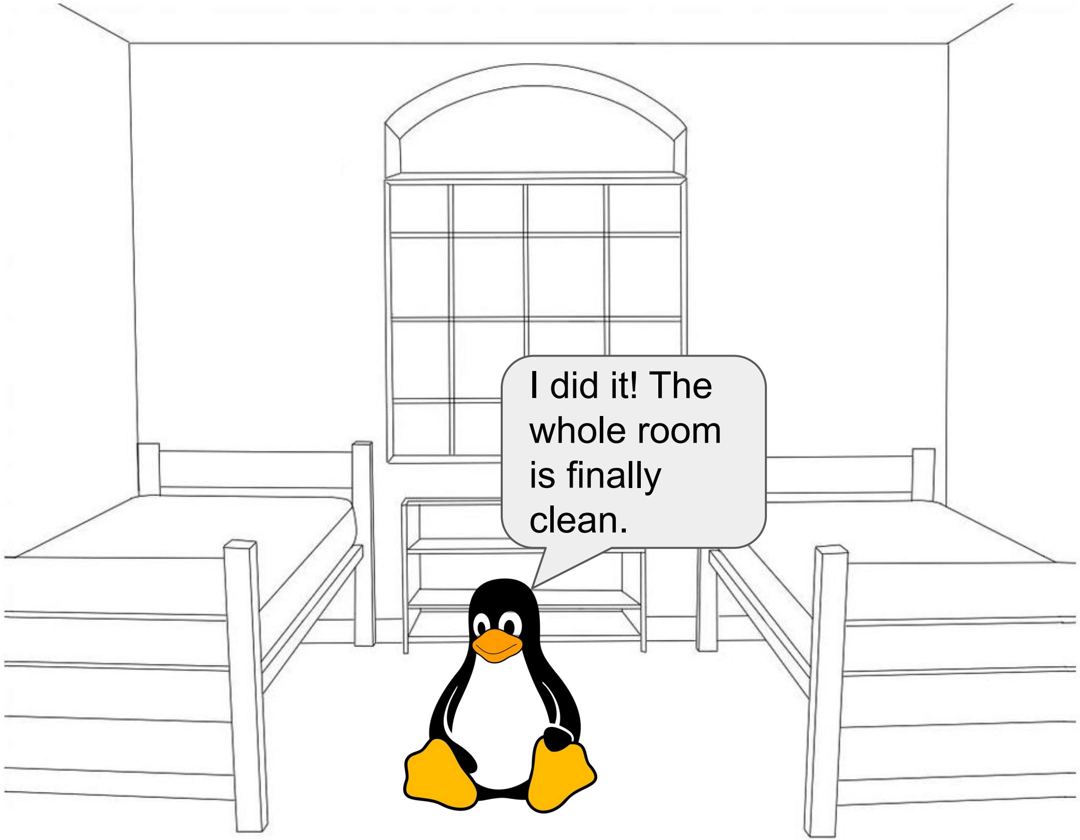


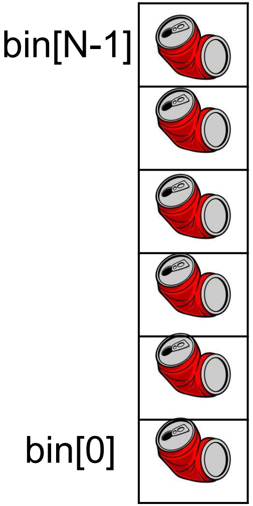
Ask them again to pick up the rest of it.

How do we get room 595 clean?

```
NumTrash pickup(roomNum, trashBin, Amount)
```

`NumTrash == -1, errno == excuse`
`NumTrash == 0`
`NumTrash < Amount`
`NumTrash == Amount`





They did what you asked, so stop asking them to pick up trash.

```
NumTrash pickup(roomNum, trashBin, Amount)
```

How do we get room 5950 clean?

NumTrash == -1, errno == excuse
NumTrash == 0
NumTrash < Amount
NumTrash == Amount

```
int pickedUp = 0;
while ( _____________ ) {











}
```

```
NumTrash pickup(roomNum, trashBin, Amount)
```

How do we get room 5950 clean?

NumTrash == -1, errno == excuse
NumTrash == 0
NumTrash < Amount
NumTrash == Amount

```
int pickedUp = 0;
while ( pickedUp < N ) {
    NumTrash = pickup( 5950, bin + pickedUp, N - pickedUp )
    if ( NumTrash == -1 ) {
        if ( excuse not reasonable )
            ask again
        stop asking and handle the excuse
    }
    if ( NumTrash == 0 )  // we over-estimated the trash
         stop asking since the room is clean
    add NumTrash to pickedUp
}
```

```
NumTrash pickup(roomNum, trashBin, Amount)
```

How do we get room 5950 clean?

NumTrash == -1, errno == excuse
NumTrash == 0
NumTrash < Amount
NumTrash == Amount

```
int pickedUp = 0;
while ( pickedUp < N ) {
    result = read( 5950, bin + pickedUp, N - pickedUp )
    if ( result == -1 ) {
        if ( errno == EINTR )
            continue;
        break;
    }
    if ( result == 0 )
        break;
    pickedUp += result;
}
```

Some Final Notes...

We assumed that there were exactly N pieces of trash (N bytes of data that we wanted to read from a file). How can we modify our solution if we don't know N?

(Answer): Keep trying to `read(...)` until we get 0 back (EOF / clean room)

We determine N dynamically by tracking the number of bytes read until this point, and use `malloc` to allocate more space as we read.

(This case comes up when reading/writing to the network!)

There is no one true loop (or true analogy).

Tailor your POSIX loops to the specifics of what you need!

Back to the worksheet (Q3)

Exercise

```
int fd = __________________________________________;  // open 595.txt
int n = 1024;
array<char,1024> buf{}; //  buf initialized with size n
int result;

______________________________;   // initialize variable for loop

...   // code that populates buf happens here

while (_______________________) {

    result = write(_______,_______________,_______________________);

    if (result == -1) {
        if (errno != EINTR) {
            // a real error happened, return an error result
            _______________________;  //  cleanup
            perror("Write failed");
            return -1;
        }
        continue;  // EINTR happened, so loop around and try again
    }
   ________________________________;   // update loop variable
}
________________; //  cleanup
```

```
 int fd = open("595.txt", O_WRONLY);  // open 595.txt
 int n = 1024;
 array<char,1024> buf{}; //  buf initialized with size n
 int result;

char *ptr = buf.data();   // initialize variable for loop

 ...   // code that populates buf happens here

 while (ptr < buf.data() + n) {

     result = write(fd, ptr, buf.data() + n - ptr);

     if (result == -1) {
         if (errno != EINTR) {
             // a real error happened, return an error result
             close(fd);  //  cleanup
             perror("Write failed");
             return -1;
         }
         continue;  // EINTR happened, so loop around and try again
     }
     ptr += result;   // update loop variable
 }
close(fd); //  cleanup
```

**This is one way to solve this exercise. There exist other correct solutions

More Posix!

4) Why is it important to store the return value from the `write()` function? Why do we not check for a return value of 0 like we do for `read()`?

5) Why is it important to remember to call the `close()` function once you have finished working on a file?

More Posix!

4) Why is it important to store the return value from the `write()` function? Why do we not check for a return value of 0 like we do for `read()`?

write() may not actually write all the bytes specified in count.
Writing adds length to your file, so you don't need to check for end of file.

5) Why is it important to remember to call the `close()` function once you have finished working on a file?

In order to free resources i.e. other processes can acquire locks on those files.

HW1 Overview

Overview

There are two FileReaders you are implementing as part of the Homework

1. SimpleFileReader
    a. A wrapper around posix, supports getting one or more characters from a file and other minor features

2. BufferedFileReader
    a. Like SimpleFileReader, but buffered and has the ability to read tokens

Internal Buffer Management

There are four pieces of data relevant to managing the buffer

- **`static constexpr uint64_t BUF_SIZE = 1024;`**
  - A constant that represents the size/capacity of the buffer
- **`array<char, BUF_SIZE> buffer_;`**
  - The buffer itself, which has size 1024

Internal Buffer Management

- `int curr_length_;`
    - A data member that represents the current length of data in the buffer
    - The buffer is 1024 long, but we may not have 1024 characters to store
    - Consider the file "hi.txt" which has the contents "hello"
        - After initially populating the buffer, curr_length_ should be 5

Internal Buffer Management

- `int curr_index_;`
  - A data member that represents the offset we are into the buffer
  - (which characters in the buffer have been returned to the user, which are still to be processed.)
  - Consider the file "hi.txt" which has the contents "hello"
    - Curr_index should start at 0
    - After reading 2 characters, curr_index_ should be 2 (so that next time we read, we read the first 'l'

Red arrow = next line to execute

Internal Buffer Examples

```
BufferedFileReader bf("hi.txt", " /t/n");
char c = bf.get_char()
c = bf.get_char();
c = bf.get_char();
```

curr_length_ 0

curr_index_ 0

	0	1	2	3	4	…	1023
buffer_	--	--	--	--	--	…	--

Red arrow = next line to execute

Internal Buffer Examples

```
   BufferedFileReader bf("hi.txt", " /t/n");
   char c = bf.get_char()  // returns 'h'
→  c = bf.get_char();
   c = bf.get_char();
```

curr_length_ 5

curr_index_ 1

	0	1	2	3	4	…	1023
buffer_	'h'	'e'	'l'	'l'	'o’	…	--

Red arrow = next line to execute

Internal Buffer Examples

```
  BufferedFileReader bf("hi.txt", " /t/n");
  char c = bf.get_char()  // returns 'h'
  c = bf.get_char();  // returns 'e'
→ c = bf.get_char();
```

curr_length_	5
curr_index_	2

	0	1	2	3	4	…	1023
buffer_	'h'	'e'	'l'	'l'	'o’	…	--

Red arrow = next line to execute

Internal Buffer Examples

```
BufferedFileReader bf("hi.txt", " /t/n");
char c = bf.get_char()  // returns 'h'
c = bf.get_char();  // returns 'e'
c = bf.get_char(); // returns 'l'
```

curr_length_	5
curr_index_	3

	0	1	2	3	4	…	1023
buffer_	'h'	'e'	'l'	'l'	'o’	…	--

Internal Buffer: Other details

- If we reach the end of the buffer, refill the buffer and start at index `0`
- If the we can’t refill the buffer due to EOF (end of file), then make sure all member functions handle the EOF behaviour correctly
  - e.g. **`get_char()`** returns **`EOF`**, **`good()`** returns false …

Any questions?