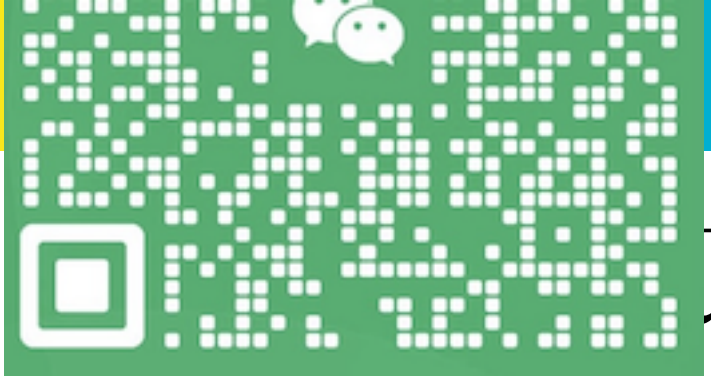
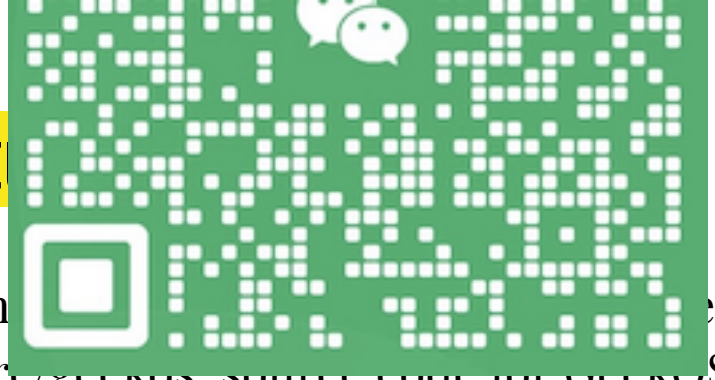


t o Pipes

Deadline: February 4, Friday at 11.59 pm
Minimum Requirements: None

http://www.cs.umd.edu/class/fall2018/cmsc412/project0-cleanedup.pdf

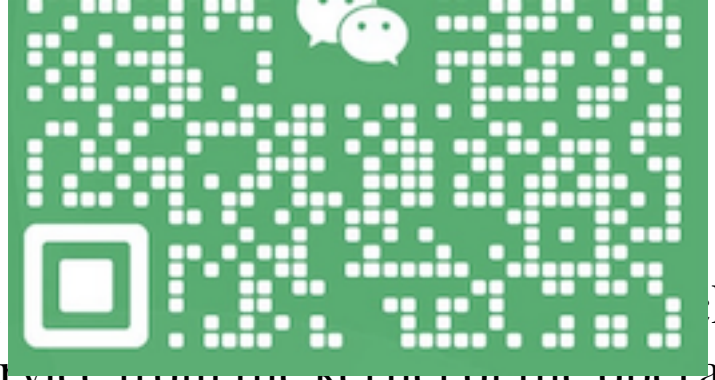
Folder Struct



1. GEEKOS_1/in ernel
2. GEEKOS_1/src/geekos: source code for GeekOS kernel
3. GEEKOS_1/src/user: test code that will get compiled to executables after booting up geekOS

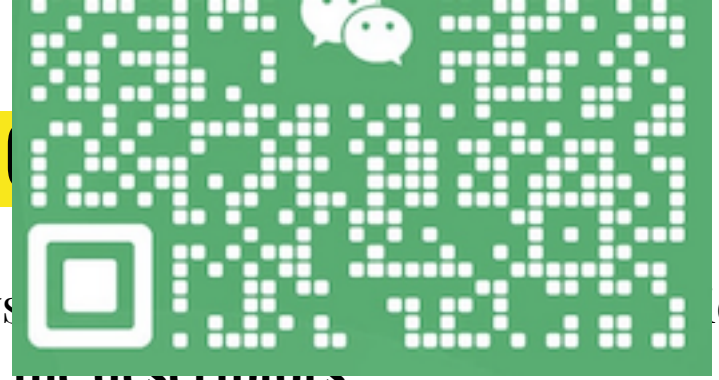
You can add any helper function you like

System Calls



1. A system call [illegible] a computer program requests a service from the kernel of the operating system it is executed on.
2. Calling function(Pipe,read,write) in an user executable(pipe-p1.c) will end up automatically calling its corresponding system call (Sys_Pipe, sys_read, sys_write). Please note that binding of all system calls is in fileio.c
3. Flow of Pipe-Create
    a. Pipe-p1.c(src/user/)--↠ fileio.c(src/libc/) --↠ syscall.c (src/geekos/) -↠pipe.c (src/geekos/)

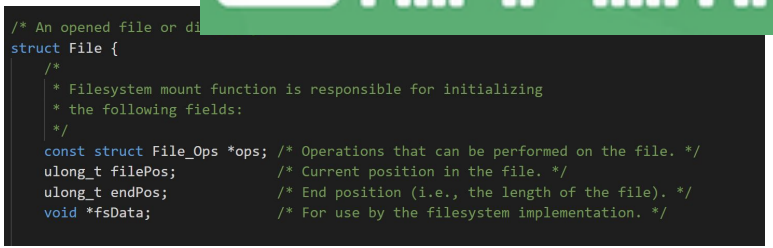
Pipe System C



- A pipe is a sys onal **communication link between two file descriptors**.
- A file descriptors is a **number that uniquely identifies an open file in a computer's operating system**.
- Pipe() takes two arguments: **each a pointer to an integer location**.
- In pipe-p1.c => `int read_fd, write_fd;` `pipe_retval = Pipe(&read_fd, &write_fd);`
- When Pipe() returns successfully, it would have created a pipe and filled the two location with file descriptors(integers), one pointing to the reading end of the pipe and the other to the writing end of the pipe

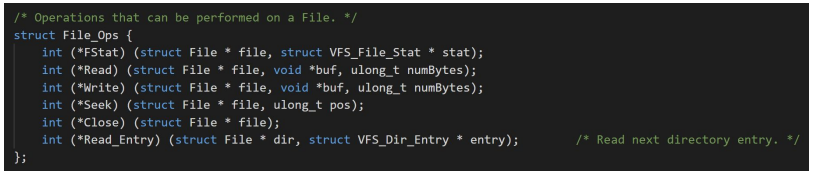
Structs

- Struct File: (in



```
/* An opened file or di
struct File {
    /*
     * Filesystem mount function is responsible for initializing
     * the following fields:
     */
    const struct File_Ops *ops; /* Operations that can be performed on the file. */
    ulong_t filePos;            /* Current position in the file. */
    ulong_t endPos;             /* End position (i.e., the length of the file). */
    void *fsData;               /* For use by the filesystem implementation. */
```

- Struct FileOps: (in vfs.h)

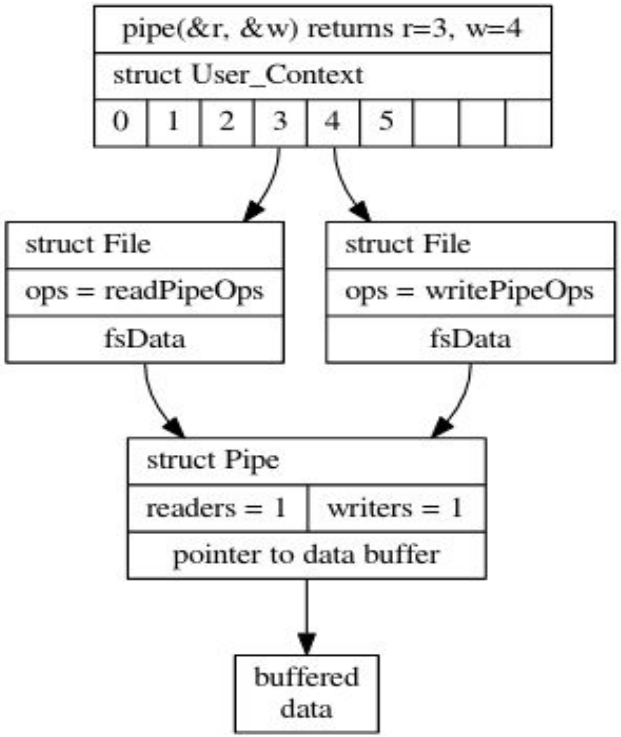


```
/* Operations that can be performed on a File. */
struct File_Ops {
    int (*FStat) (struct File * file, struct VFS_File_Stat * stat);
    int (*Read) (struct File * file, void *buf, ulong_t numBytes);
    int (*Write) (struct File * file, void *buf, ulong_t numBytes);
    int (*Seek) (struct File * file, ulong_t pos);
    int (*Close) (struct File * file);
    int (*Read_Entry) (struct File * dir, struct VFS_Dir_Entry * entry);       /* Read next directory entry. */
};
```

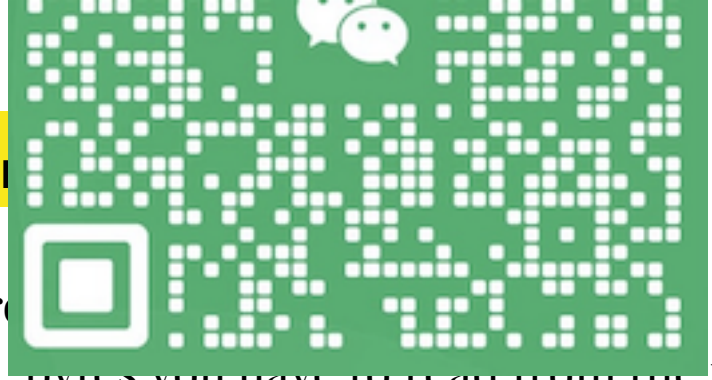
- Struct pipe: you need to create

Pipe_Create()

- Two File doub
  WRITE_FILE)
- Create new struct File instance using Malloc()
- Initialize files by referring to appropriate File_Ops defined in the **pipe.c** file
- Need to have your own pipe struct to hold data and other variables of importance (as per your judgement). Use **fsData(void* pointer)** in file to point to the instance of your pipe struct
- Check for appropriate **error** conditions wherever necessary
- **Return o** if successful

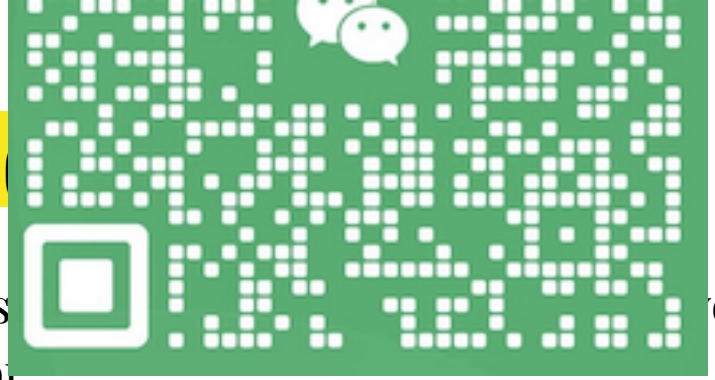


Pipe_Read() (



- Reads data fr
- Inputs: num_bytes you have to read from the pipe, a buffer to copy data into and a struct pointer (File *f) ↠ a read file descriptor
- Check for appropriate error conditions
  - pipe has writers but no data, return EWOULDBLOCK
  - Pipe has no writers and no data, return 0
- Copy the data into the buffer (it's a void *)
  - E.g, You can use **memcpy(to , from , how many bytes you want to copy)**
    - Reading 4 bytes to the pipe from the beginning (memcpy(pipe↠data_buffer, buf, 4);)
  - If there is data, Read() returns at most as much data as it was asked for.
- Delete the data from the pipe's buffer (remove the data you have just read out or mark the data you have read out as invalid)
- Return number of bytes copied

Pipe_Write()



- Same params ou copy data from and write file descriptor
- Implement the buffer like a queue; write appends data, does NOT overwrite
- If there is a reader and the pipe has space for data, pipe_Write() returns the number of bytes written.
- Error conditions:
  - No reader, return EPIPE
  - If you choose to implement a fixed size buffer(suggested 32K)): if buffer is full, return 0
  - If you choose to implement dynamically allocated buffer: if malloc() fails, return ENOMEM

VFS Layer

- Pipe-p1.c(src/ yscall.c (src/geekos/) ↠ vfs.c(src/)-↠ pipe.c(src/geekos/)
- In geekos/vfs.c

```
int Read(struct File *file, void *buf, ulong_t len) {
    if(file->ops->Read == 0)
        return EUNSUPPORTED;
    else
        return file->ops->Read(file, buf, len);
}
```

- It calls Pipe_Read() and Pipe_Write() using the function pointer
- Assign the function pointer under ops correctly(in Pipe_Create())

Pipe_Close() (

- Check if func side or the write side and then act appropriately by closing the side on which it was called.
- Destroy data if there is no reader but there is still data.
- Pipe can also be destroyed if there are no readers and no writers.

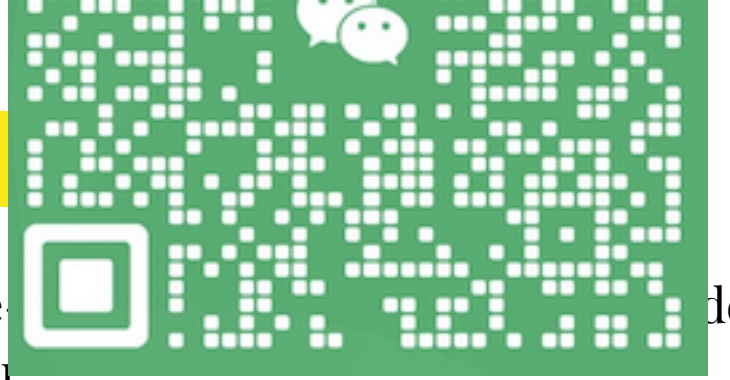
Sys_Pipe()

- This is what is executed in test files

```
/* Print("calling pipe"); */
pipe_retval = Pipe(&read_fd, &write_fd);
assert(pipe_retval == 0);
```

- Create the pipe (call Pipe_Create()).
- Add files to the descriptor table (check for error conditions here)(use **add_file_to_descriptor_table** method).
- Use Copy_To_User (ulong_t destInUser, const void *srcInKernel, ulong_t bufSize) to copy the file descriptors to the user addresses stored in the state registers (refer to geekos slides on how to use).
- Return o if successful.

Testing Your



- pipe-p1, pipe- ded to you
- Check src/usr

Debugging

1. Use Print() fu
2. Use GDB
   a. Make dbgrun on one window
   b. After VGA Blank mode shows up, open another window and make dbg
   c. A GDB Cheat Sheet https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf