xv6 study
(Lab: Xv6 and Unix utilities)

(0) xv6

Scheduling: Introduction

- xv6 is a reimplementation of UNIX for x86 and RISC-V
- It was created for educational purpose.
- List of System Call

int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

- "Lab: Xv6 and Unix utilities" helps us familiarize with xv6 and it's system calls.
  - There are five problems : sleep, pingpong, primes, find and xargs.

(1) sleep

sleep : solution

```
$ make qemu
...
init: starting sh
$ sleep 10
(nothing happens for a little while)
$
```

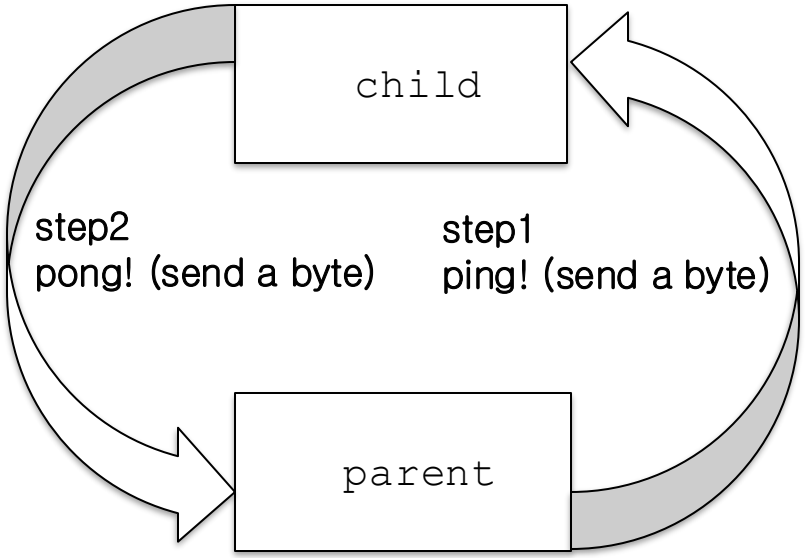
```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: sleep <time>₩n");
        exit(1);
    }
    int time = atoi(argv[1]);
    if (time <= 0) {
        printf("Error: sleep time must be a positive integer₩n");
        exit(1);
    }
    sleep(time);
    exit(0);
}
```

(2) pingpong

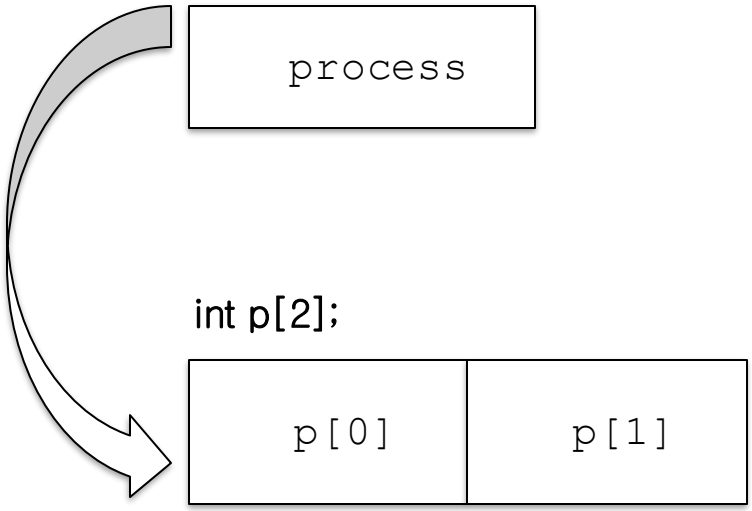
pingpong : solution

```
int p[2];
int q[2];
pipe(p);
pipe(q);
char c;
if(fork() == 0) //child
{
    read(p[0], &c, 1);
    printf("%d: received ping₩n", getpid());
    write(q[1], &c, 1);
}
else //parent
{
    write(p[1], &c, 1);
    read(q[0], &c, 1);
    printf("%d: received pong₩n", getpid());
}
exit(0);
```

- Step1
  - The parent sends a byte to the child
  - The child receive a byte
    - print "<pid>:received ping
- Step2
  - The child sends a byte to the parent
  - The parent receive a byte
    - print "<pid>:received pong

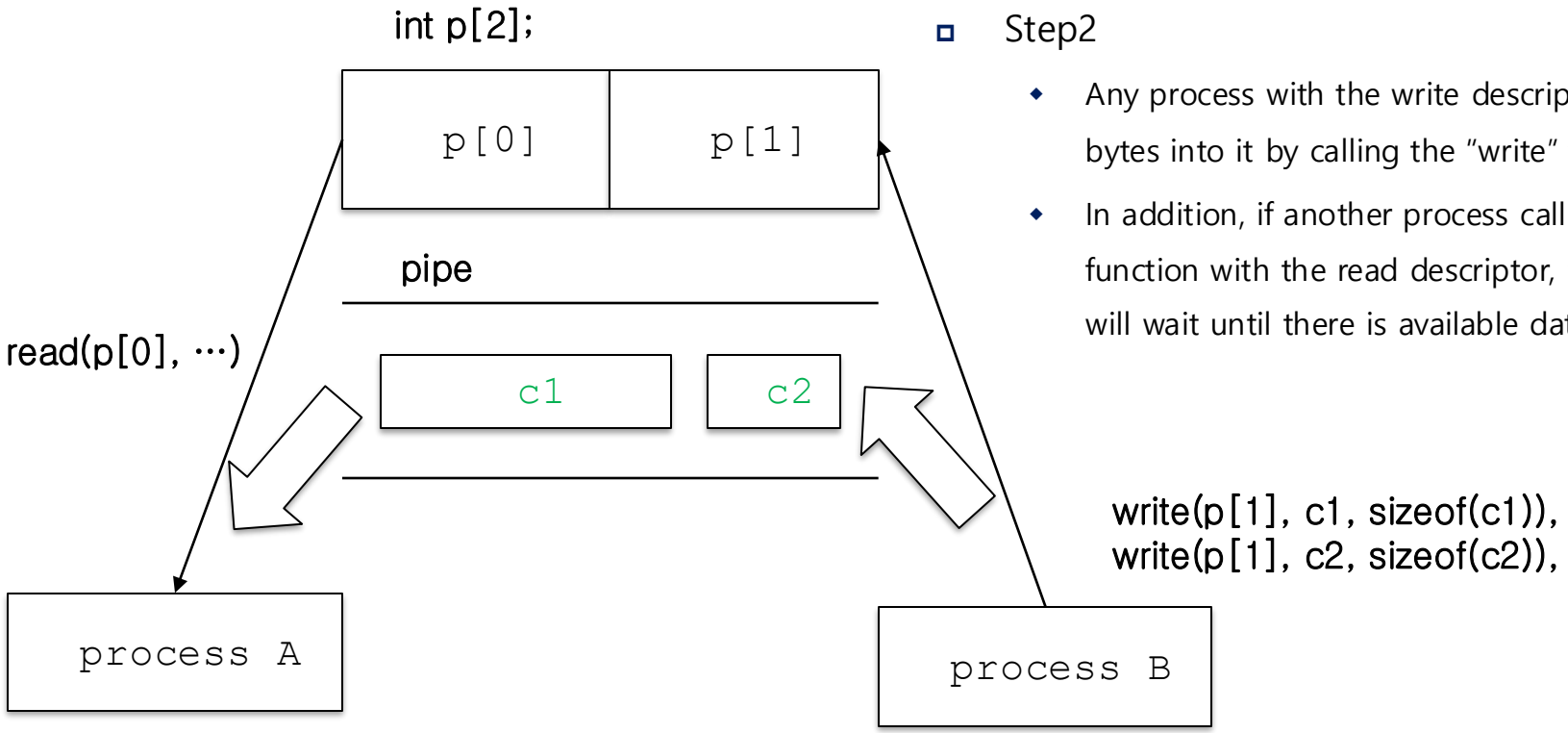


pipe, read, write



- Step1
  - call the system-call "pipe" with an integer array of two elements.
  - The first element of the array hold the file descriptor for reading.
  - The second element of the array hold the file descriptor for writing.

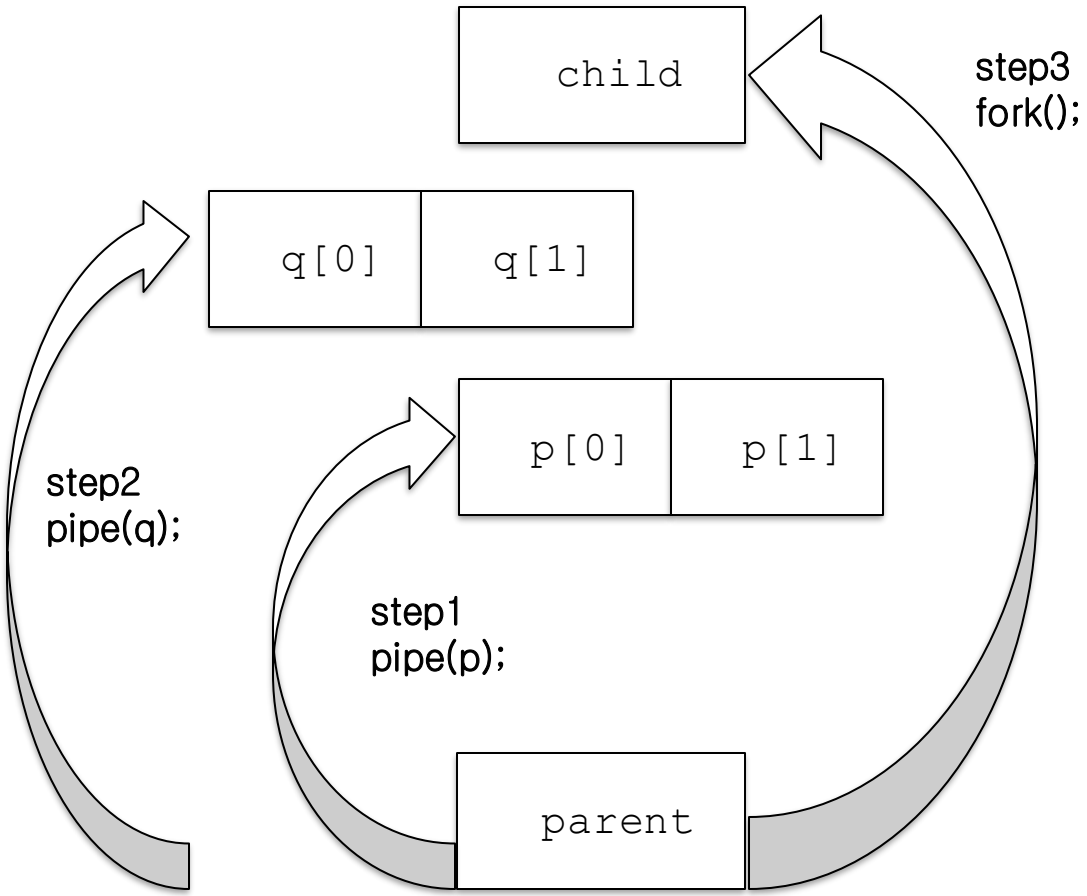
pipe, read, write



- Step1
  - call the system-call "pipe" with an integer array of two elements.
  - The first element of the array hold the file descriptor for reading.
  - The second element of the array hold the file descriptor for writing.
- Step2
  - Any process with the write descriptor can write bytes into it by calling the "write" function.
  - In addition, if another process call the "read" function with the read descriptor, the process will wait until there is available data in pipe

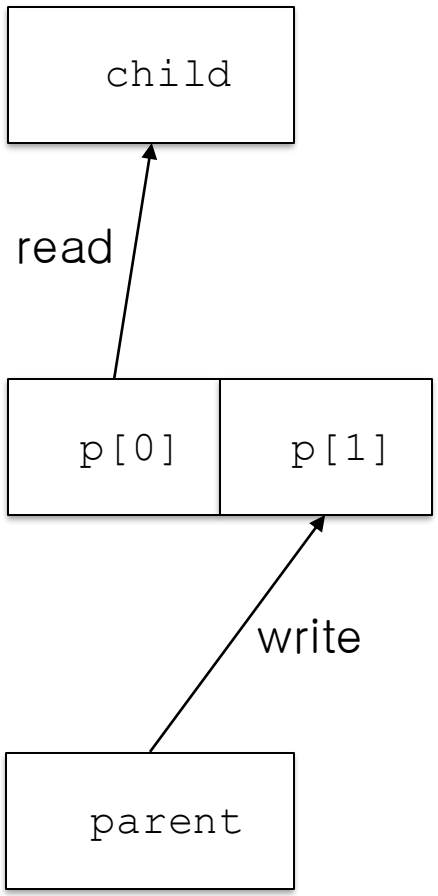
pingpong : solution

```
int p[2];
int q[2];
pipe(p);
pipe(q);
char c;
if(fork() == 0) //child
{
    read(p[0], &c, 1);
    printf("%d: received ping₩n", getpid());
    write(q[1], &c, 1);
}
else //parent
{
    write(p[1], &c, 1);
    read(q[0], &c, 1);
    printf("%d: received pong₩n", getpid());
}
exit(0);
```



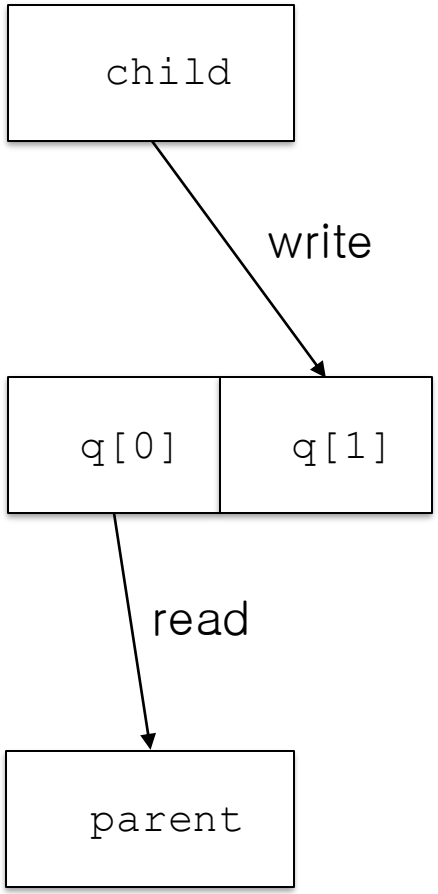
pingpong : solution

```
int p[2];
int q[2];
pipe(p);
pipe(q);
char c;
if(fork() == 0) //child
{
    read(p[0], &c, 1);
    printf("%d: received ping₩n", getpid());
    write(q[1], &c, 1);
}
else //parent
{
    write(p[1], &c, 1);
    read(q[0], &c, 1);
    printf("%d: received pong₩n", getpid());
}
exit(0);
```



pingpong : solution

```
int p[2];
int q[2];
pipe(p);
pipe(q);
char c;
if(fork() == 0) //child
{
    read(p[0], &c, 1);
    printf("%d: received ping₩n", getpid());
    write(q[1], &c, 1);
}
else //parent
{
    write(p[1], &c, 1);
    read(q[0], &c, 1);
    printf("%d: received pong₩n", getpid());
}
exit(0);
```

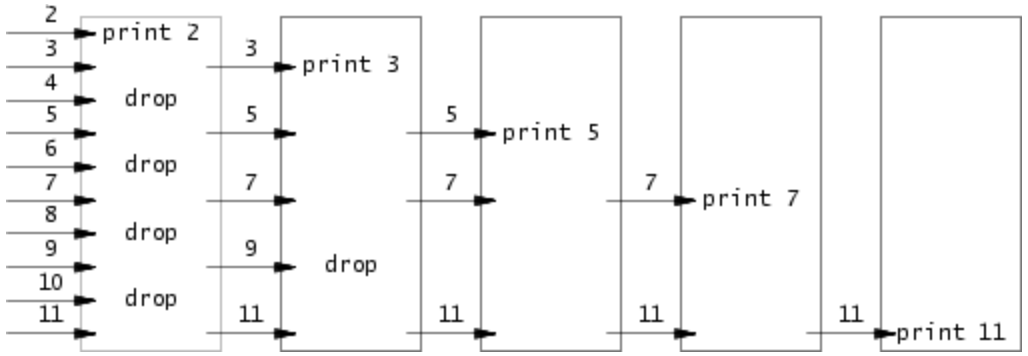


(3) primes

primes : solution

```
int ps[2];
pipe(ps);
{//step1
    ...
}
int prime;
while(read(ps[0], &prime, sizeof(int))){
    printf("prime %d₩n", prime);
    int new_ps[2];
    pipe(new_ps);
    if(!fork())
    {//step2
        //
    }
    else
    {
        wait(0);
    }
    close(ps[0]);
    close(new_ps[1]);
    ps[0] = new_ps[0];
}
```

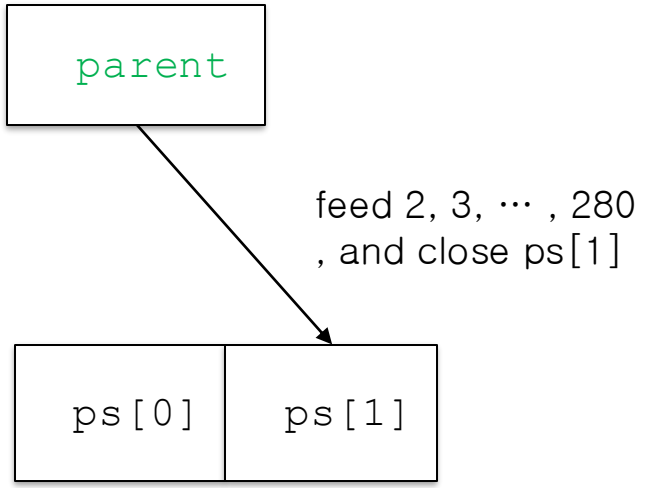
- step1
  - The first process feed the number 2 through 280 into the left end of the pipeline
- step2
  - For each prime p, eliminates the multiples of p, and feed the remained numbers into the left end of the pipeline



primes : solution

```
int ps[2];
pipe(ps);
{//step1
    for(int i=2;i<=280;i++)write(ps[1], &i, sizeof(i));
}
int prime;
while(read(ps[0], &prime, sizeof(int))){
    printf("prime %d₩n", prime);
    int new_ps[2];
    pipe(new_ps);
    if(!fork())
    {//step2
        //
    }
    else
    {
        wait(0);
    }
    close(ps[0]);
    close(new_ps[1]);
    ps[0] = new_ps[0];
}
```

- step1
  - The first process feed the number 2 through 280 into the left end of the pipeline

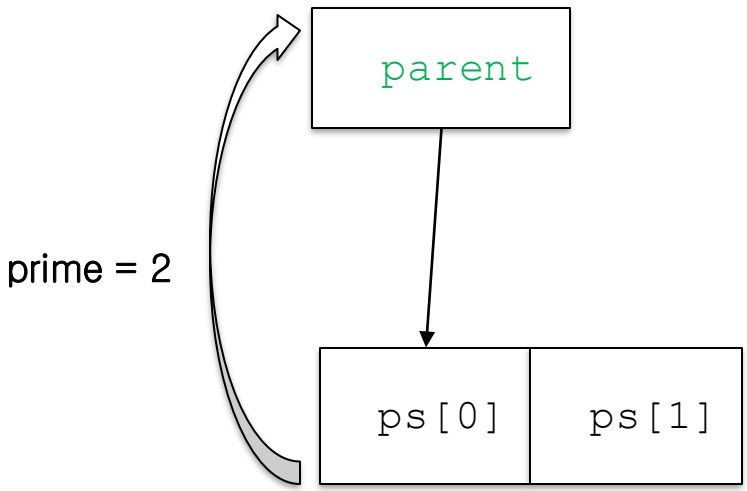


{2, 3, 4, … , 279, 280}

primes : solution

```
…
int prime;
while(read(ps[0], &prime, sizeof(int))){
    printf("prime %d₩n", prime);
    int new_ps[2];
    pipe(new_ps);
    if(!fork())
    {//step2
        //
    }
    else
    {
        wait(0);
    }
    close(ps[0]);
    close(new_ps[1]);
    ps[0] = new_ps[0];
}
```

- step2
  - For each prime p, eliminates the multiples of p, and feed the remained numbers into the left end of the pipeline

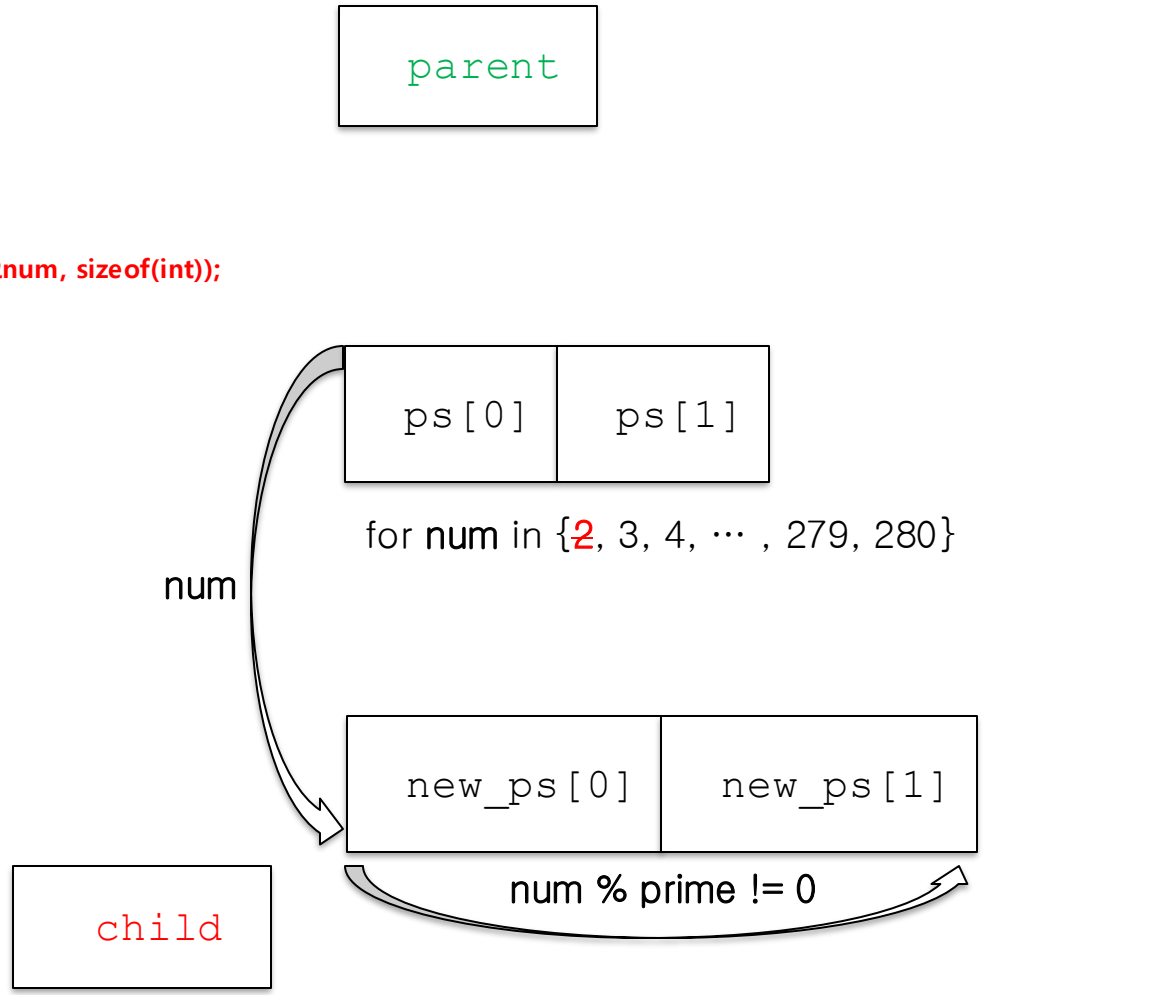


{~~2~~, 3, 4, … , 279, 280}

primes : solution

```
...
while(read(ps[0], &prime, sizeof(int))){
    ....
    int new_ps[2];
    pipe(new_ps);
    if(!fork())
    {//step2
        int num;
        while(read(ps[0], &num, sizeof(int)))
            if(num % prime != 0) write(new_ps[1], &num, sizeof(int));
        exit(0);
    }
    else
    {
        wait(0);
    }
    close(ps[0]);
    close(new_ps[1]);
    ps[0] = new_ps[0];
}
```

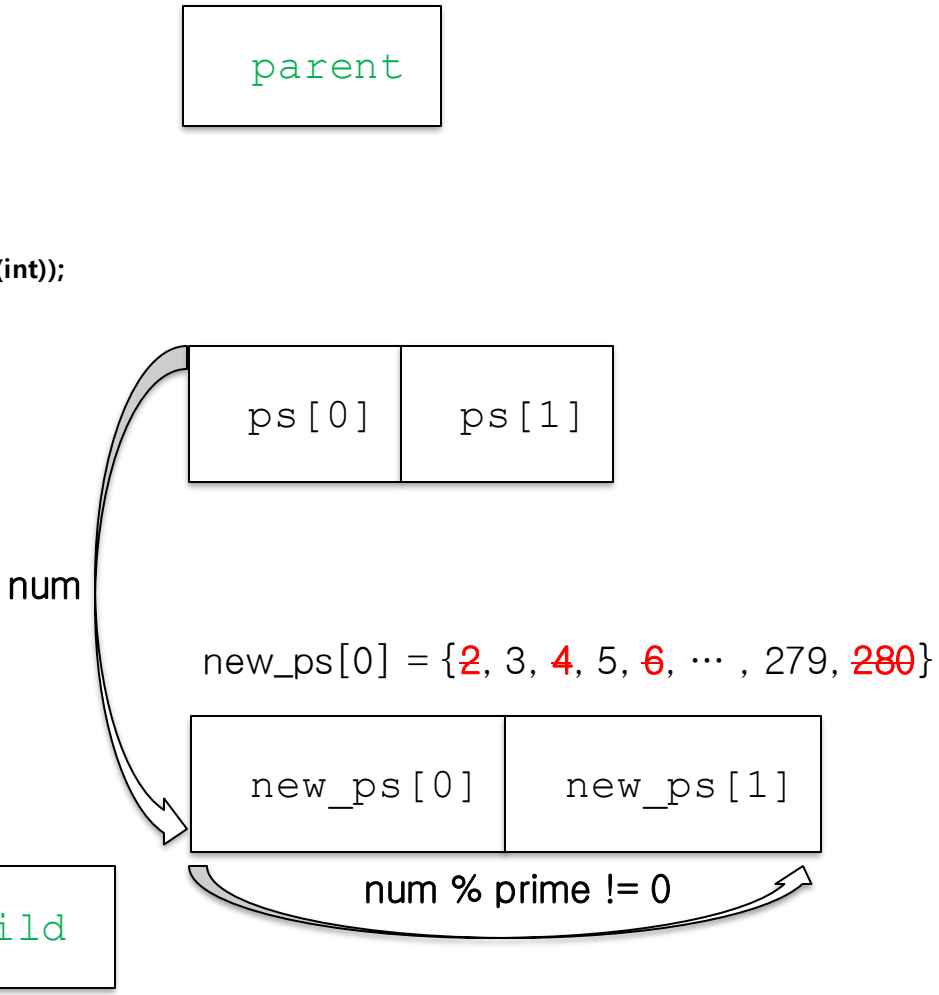
- step2
  - For each prime p, eliminates the multiples of p, and feed the remained numbers into the left end of the pipeline



primes : solution

```
...
while(read(ps[0], &prime, sizeof(int))){
    ....
    int new_ps[2];
    pipe(new_ps);
    if(!fork())
    {//step2
        int num;
        while(read(ps[0], &num, sizeof(int)))
            if(num % prime != 0) write(new_ps[1], &num, sizeof(int));
        exit(0);
    }
    else
    {
        wait(0);
    }
    close(ps[0]);
    close(new_ps[1]);
    ps[0] = new_ps[0];
}
```

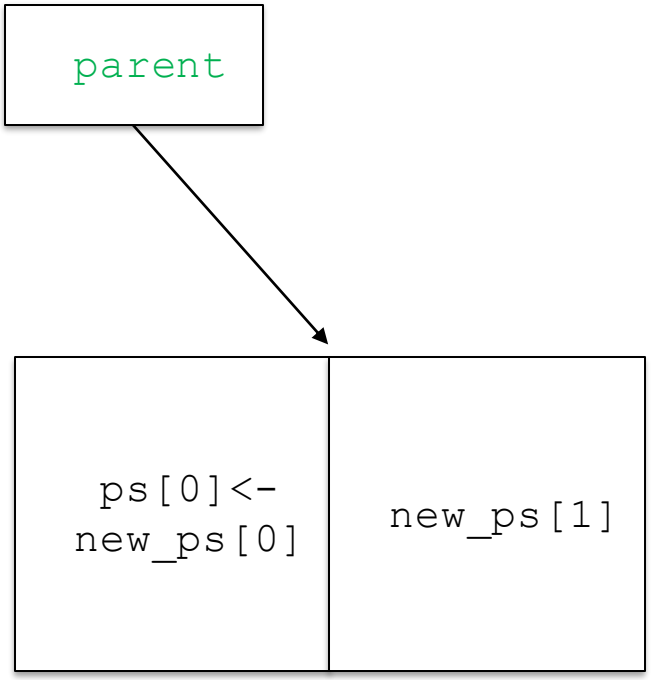
- step2
  - For each prime p, eliminates the multiples of p, and feed the remained numbers into the left end of the pipeline



primes : solution

```
...
while(read(ps[0], &prime, sizeof(int))){
    ....
    int new_ps[2];
    pipe(new_ps);
    if(!fork())
    {//step2
        int num;
        while(read(ps[0], &num, sizeof(int)))
            if(num % prime != 0) write(new_ps[1], &num, sizeof(int));
        exit(0);
    }
    else
    {
        wait(0);
    }
    close(ps[0]);
    close(new_ps[1]);
    ps[0] = new_ps[0];
}
```

- step2
  - For each prime p, eliminates the multiples of p, and feed the remained numbers into the left end of the pipeline



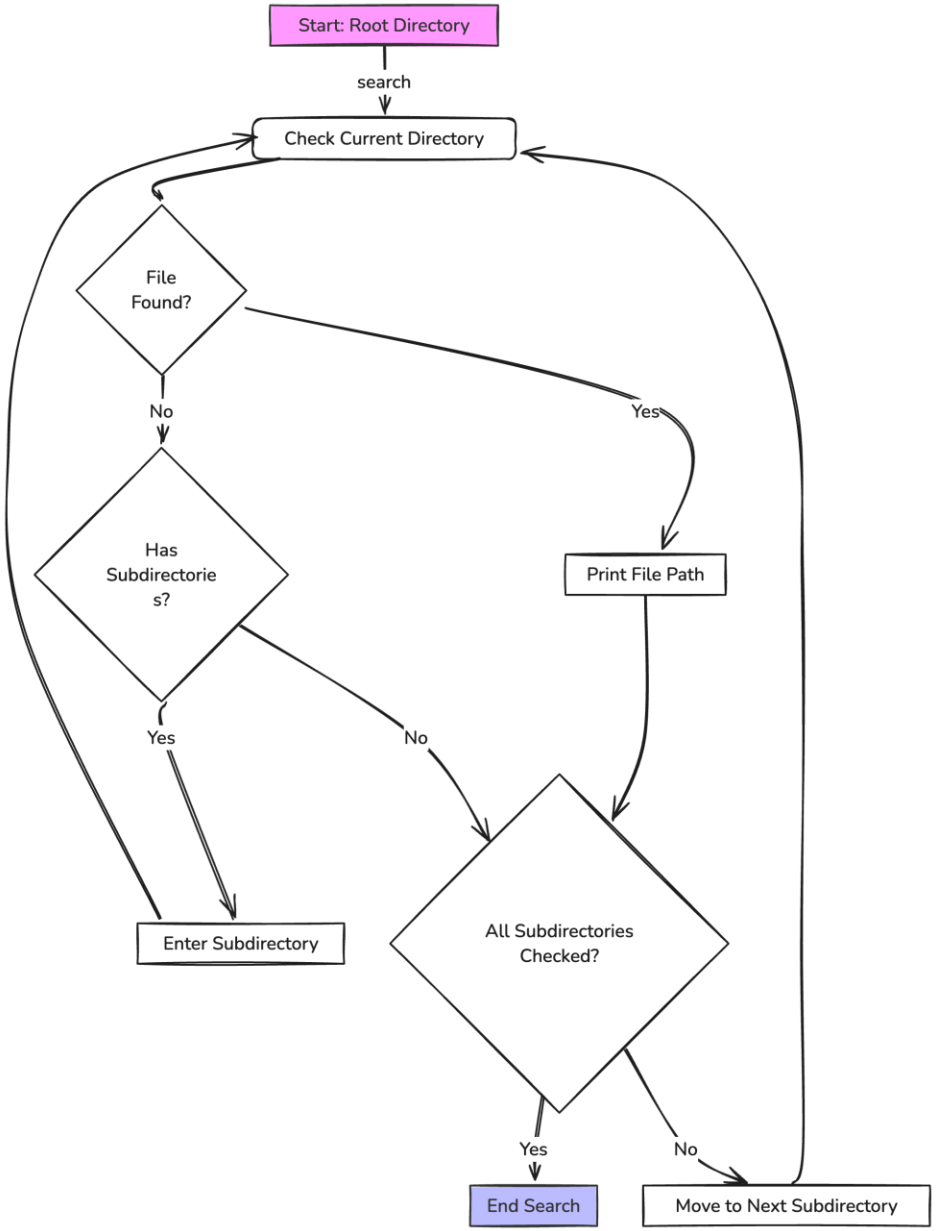
ps[0] = {~~2~~, 3, ~~4~~, 5, ~~6~~, ⋯ , 279, ~~280~~}

(4) find

find : overview

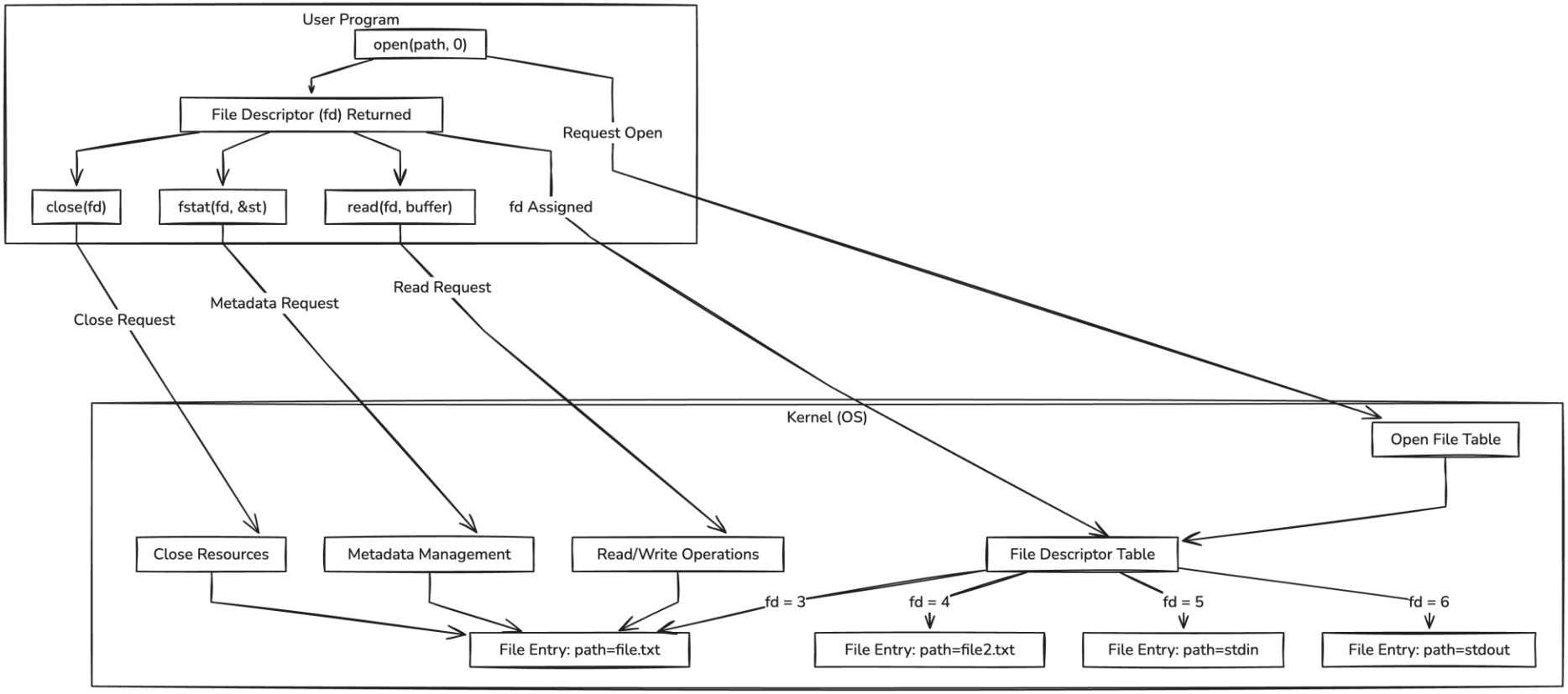
- Search for files recursively in a directory.

```
+------------------------------+      +--------------------------+
|      User Program (find)     |      |        Kernel (OS)       |
+------------------------------+      +--------------------------+
|                              |      |                          |
| open(path, 0)                | ---> | Open the directory file  |
|                              |      | Return File Descriptor FD|
|                              | <--- |                          |
|                              |      |                          |
| fstat(fd, &st)               | ---> | Fetch metadata (st.type) |
|                              |      | Return file status       |
|                              | <--- |                          |
|                              |      |                          |
| read(fd, &de, sizeof(de))    | ---> | Read directory entry     |
|                              |      | Return entry information |
|                              | <--- |                          |
|                              |      |                          |
| stat(buf, &st)               | ---> | Get metadata for file path|
|                              |      | Return file status       |
|                              | <--- |                          |
|                              |      |                          |
| close(fd)                    | ---> | Close the file descriptor |
|                              |      | Release resources        |
|                              | <--- |                          |
+------------------------------+      +--------------------------+
```



find : overview

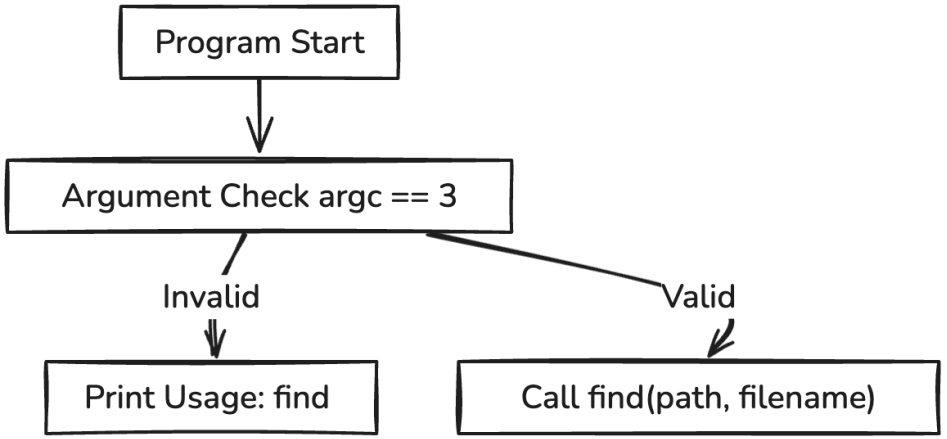
- System Call Flow



find : solution

- Main Function

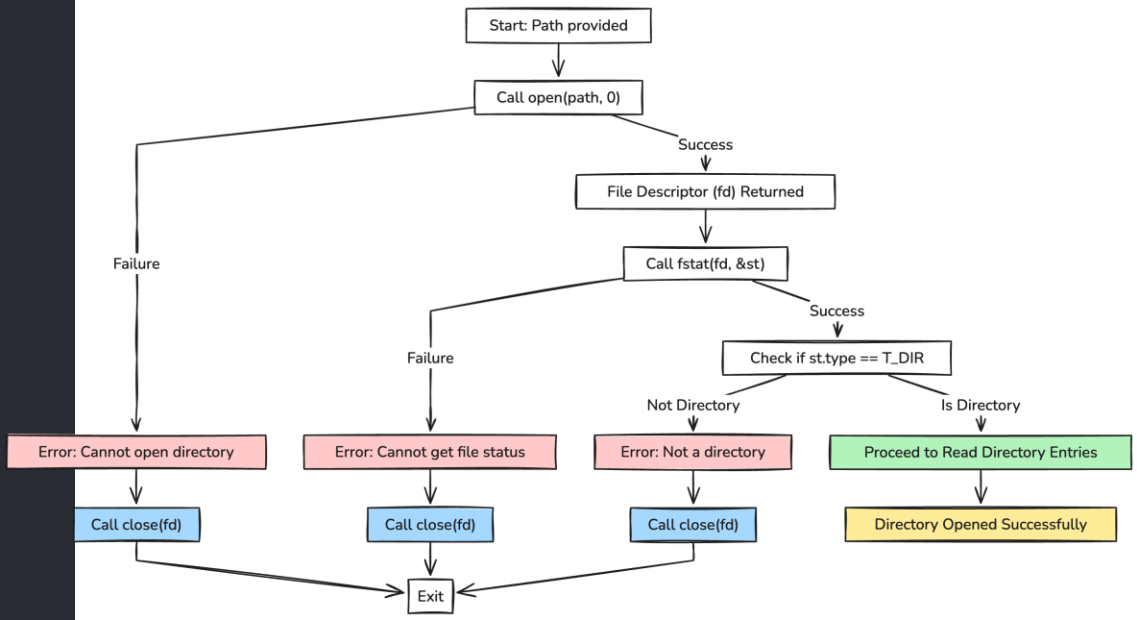
```
73  int main(int argc, char *argv[]) {
74      if (argc != 3) {
75          printf("Usage: find <path> <filename>\n");
76          exit(1);
77      }
78
79      find(argv[1], argv[2]);
80      exit(0);
81  }
82
```



find : solution

- Find Function : Opening and Verifying the Directory: open() and fstat()

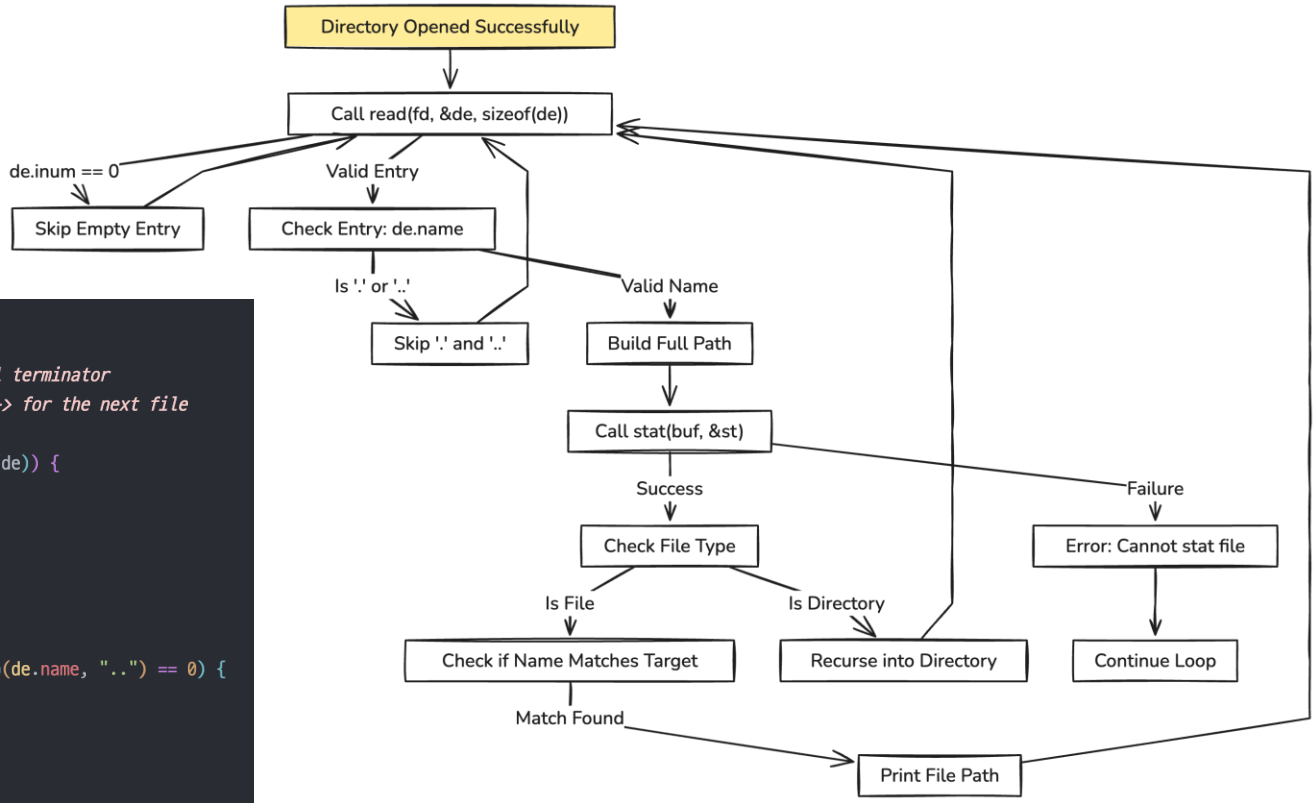
```c
void find(const char *path, const char *target) {
    char buf[512], *p;
    int fd;
    struct dirent de;
    struct stat st;

    // open the directory
    if ((fd = open(path, 0)) < 0) {
        fprintf(2, "find: cannot open %s\n", path);
        return;
    }

    // get the status of the directory
    if (fstat(fd, &st) < 0) {
        fprintf(2, "find: cannot open %s\n", path);
        close(fd);
        return;
    }

    // ensure it is a directory
    if (st.type != T_DIR) {
        fprintf(2, "find: %s is not a directory\n", path);
        close(fd);
        return;
    }
```



find : solution

- Find Function : Reading Directory Entries: read() and Iterating Entries

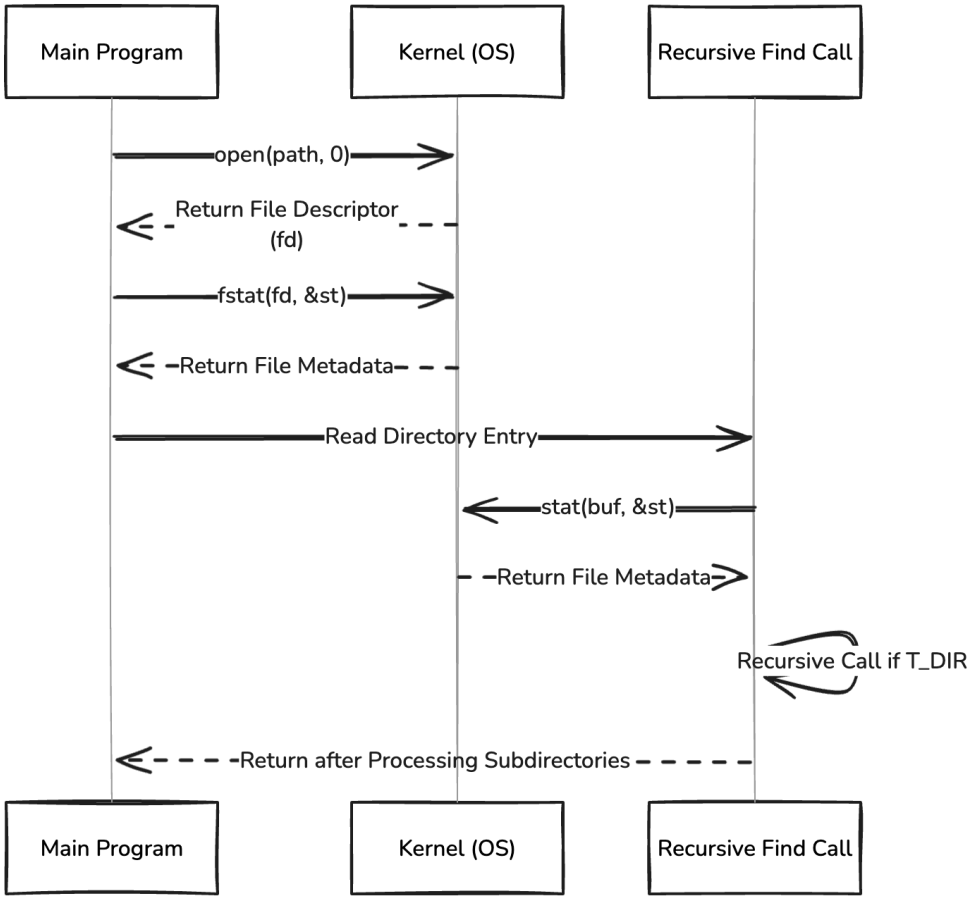
```
32      // read directory entries
33      strcpy(buf, path);
34      p = buf+strlen(buf); // point p to the null terminator
35      *p++ = '/'; // append a slash to the path -> for the next file
36
37      while (read(fd, &de, sizeof(de)) == sizeof(de)) {
38          // skip empty directory entries
39          if (de.inum == 0) {
40              continue;
41          }
42
43          // skip "." and ".."
44          if (strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0) {
45              continue;
46          }
47
48          // Build the full path
49          memmove(p, de.name, DIRSIZ);
50          // ensure the path is null-terminated
51          p[DIRSIZ] = 0;
52
53          // get the status of the current file
54          if (stat(buf, &st) < 0) {
55              printf("find: cannot stat %s\n", buf);
56              continue;
57          }
```



find : solution

- Find Function : Path Construction and Recursive Call

```
59          // check if the current file matches the target
60          if (strcmp(de.name, target) == 0) {
61              printf("%s\n", buf);
62          }
63
64          // if it's a directory, recurse into it
65          if (st.type == T_DIR) {
66              find(buf, target);
67          }
68      }
69
70      close(fd);
```

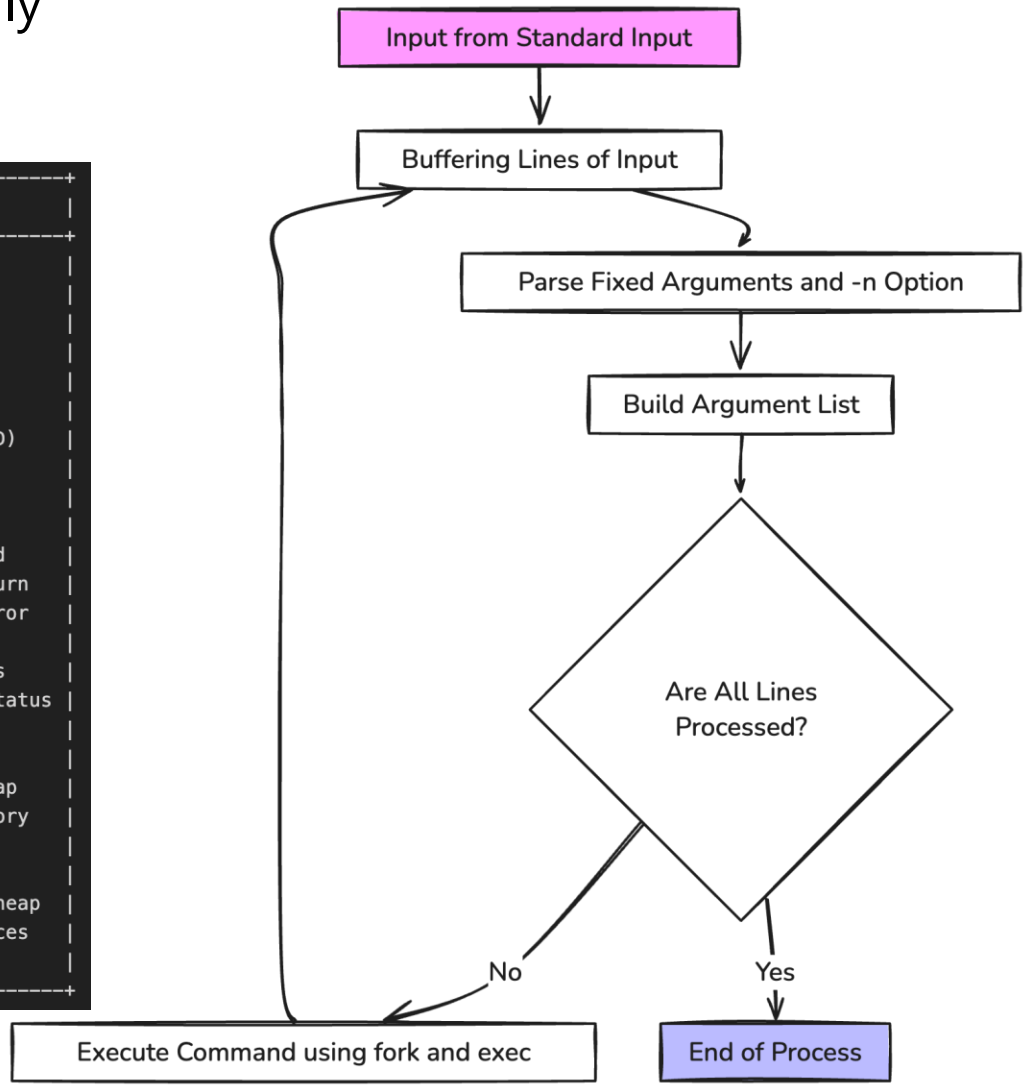


(5) xargs

xargs : overview

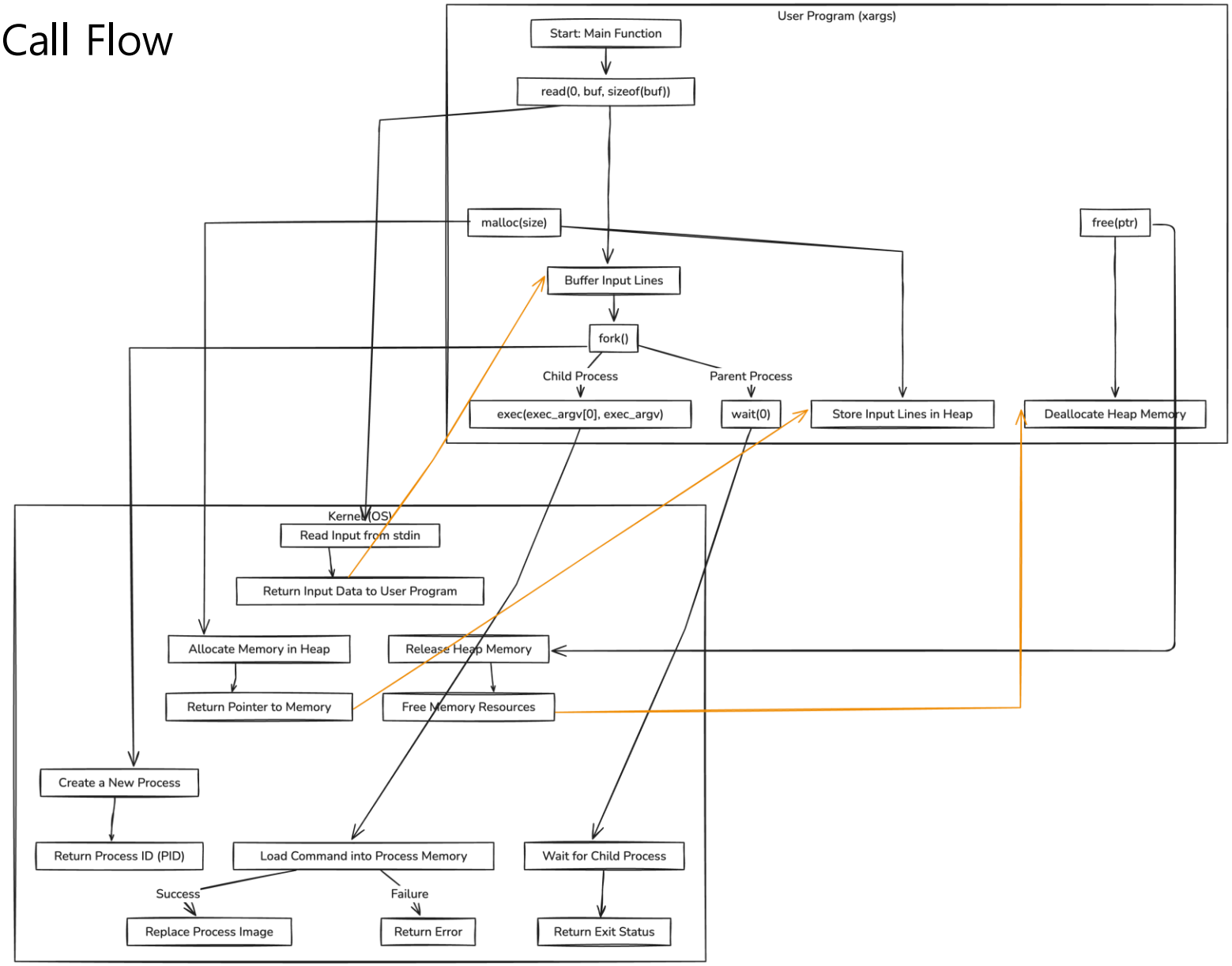
- Execute commands dynamically using input lines.

```
+-----------------------------------+      +---------------------------+
|       User Program (xargs)        |      |         Kernel (OS)       |
+-----------------------------------+      +---------------------------+
|                                   |      |                           |
| read(0, buf, sizeof(buf))         | ---> | Read input from stdin     |
|                                   |      | Return buffer data        |
|                                   | <--- |                           |
|                                   |      |                           |
| fork()                            | ---> | Create a new process      |
|                                   |      | Return process ID (PID)   |
|                                   | <--- |                           |
|                                   |      |                           |
| exec(exec_argv[0], exec_argv)     | ---> | Replace process image     |
|                                   |      | Load specified command    |
|                                   |      | If successful, no return  |
|                                   | <--- | On failure, return error  |
|                                   |      |                           |
| wait(0)                           | ---> | Wait for child process    |
|                                   |      | Return child's exit status|
|                                   | <--- |                           |
|                                   |      |                           |
| malloc(size)                      | ---> | Allocate memory in heap   |
|                                   |      | Return pointer to memory  |
|                                   | <--- |                           |
|                                   |      |                           |
| free(ptr)                         | ---> | Deallocate memory in heap |
|                                   |      | Release memory resources  |
|                                   | <--- |                           |
+-----------------------------------+      +---------------------------+
```



xargs : overview

- System Call Flow

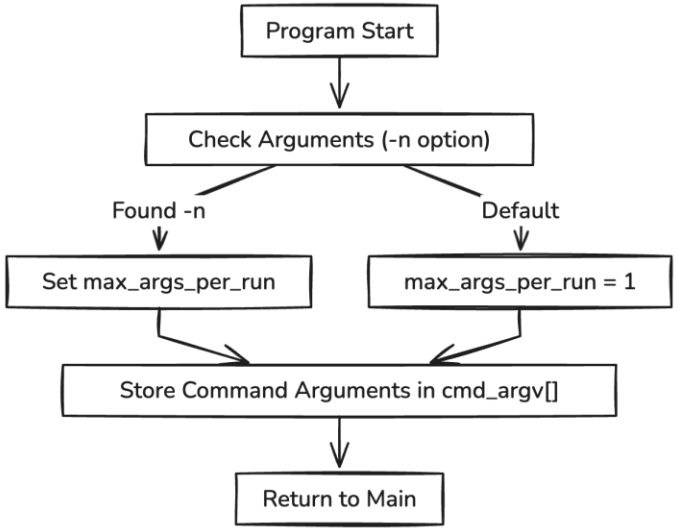


xargs : solution

- Argument Parsing

```
113  int main(int argc, char **argv) {
114      if (argc < 2) {
115          printf("Usage: xargs [-n num] <command> [args...]\n");
116          exit(1);
117      }
118
119      int max_args_per_run = 1, start_index = 1, cmd_argc = 0;
120      char *cmd_argv[MAXARG];
121      parse_arguments(argc, argv, &max_args_per_run, &start_index, cmd_argv, &cmd_argc);
```

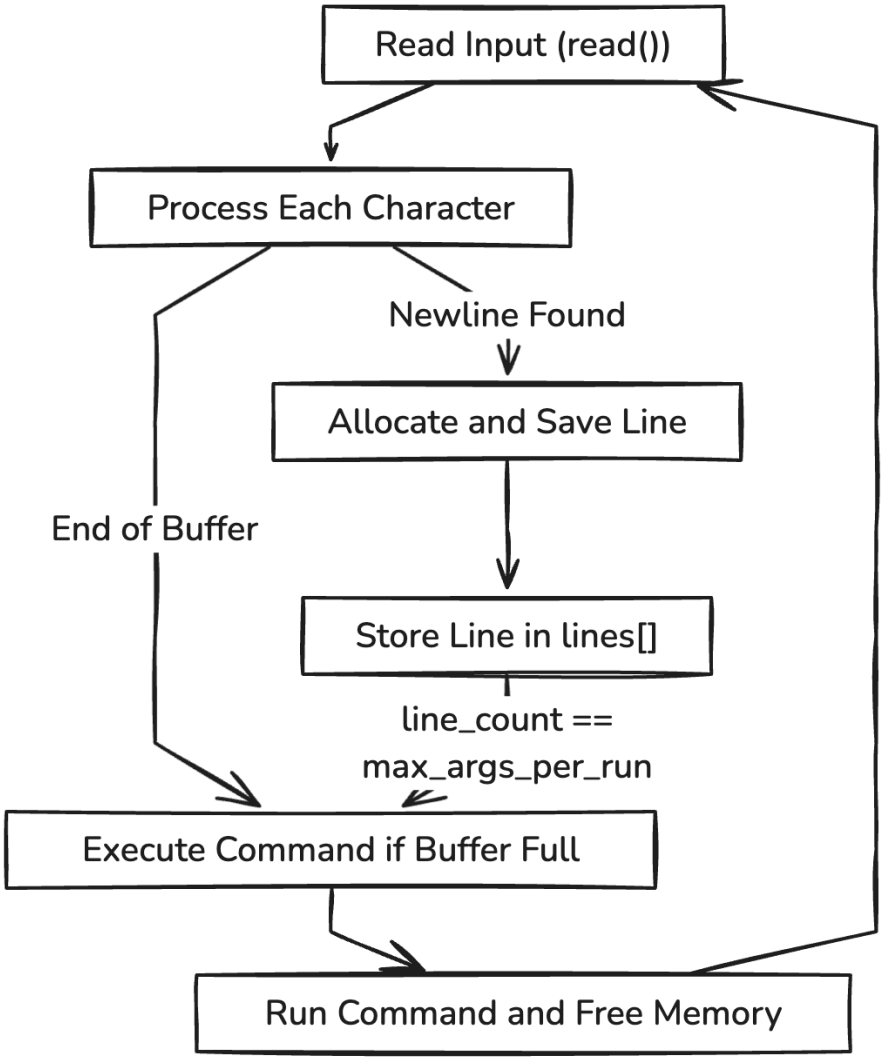
```
47  void parse_arguments(int argc, char **argv, int *max_args_per_run, int *start_index, char **cmd_argv, int *cmd_argc) {
48      if (argc > 2 && strcmp(argv[1], "-n") == 0) {
49          *max_args_per_run = atoi(argv[2]);
50          if (*max_args_per_run < 1) {
51              printf("xargs: invalid value for -n\n");
52              exit(1);
53          }
54          *start_index = 3;
55      }
56      for (int i = *start_index; i < argc && *cmd_argc < MAXARG - 1; i++) {
57          cmd_argv[(*cmd_argc)++] = argv[i];
58      }
59      cmd_argv[*cmd_argc] = 0;
60  }
```



xargs : solution

- Reading Input and Storing Lines

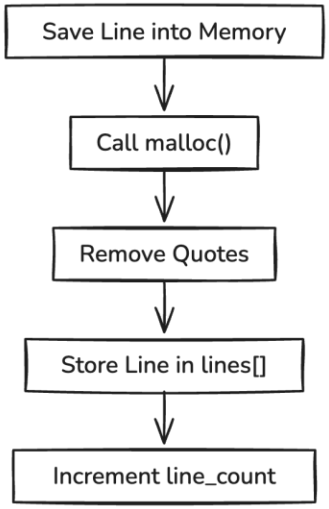
```c
void process_input(char **cmd_argv, int cmd_argc, int max_args_per_run) {
    char buf[BUFFER_SIZE], line[BUFFER_SIZE];
    int n, line_idx = 0, line_count = 0;
    char *lines[MAXARG];

    while ((n = read(0, buf, sizeof(buf))) > 0) {
        for (int i = 0; i < n; i++) {
            if (buf[i] == '\n') {
                line[line_idx] = 0;
                allocate_and_save_line(line, lines, &line_count);
                if (line_count == max_args_per_run) {
                    execute_lines(cmd_argv, cmd_argc, lines, line_count);
                    line_count = 0;
                }
                line_idx = 0;
            } else {
                line[line_idx++] = buf[i];
            }
        }
    }
    if (line_idx > 0) {
        line[line_idx] = 0;
        allocate_and_save_line(line, lines, &line_count);
    }
    execute_lines(cmd_argv, cmd_argc, lines, line_count);
}
```



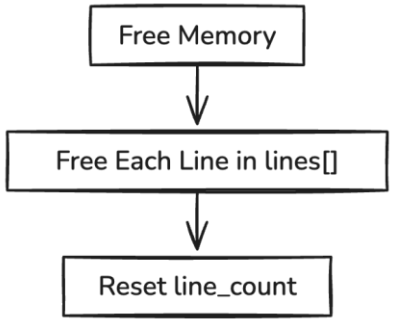
xargs : solution

- Memory Allocation and Cleanup

```c
void allocate_and_save_line(char *line, char **lines, int *line_count) {
    char *saved_line = malloc(strlen(line) + 1);
    if (!saved_line) {
        printf("xargs: malloc failed\n");
        exit(1);
    }
    strcpy(saved_line, line);
    remove_quotes(saved_line);
    lines[(*line_count)++] = saved_line;
}
```




```c
void cleanup_lines(char **lines, int line_count) {
    for (int i = 0; i < line_count; i++) {
        free(lines[i]);
    }
}
```



xargs : solution

- Executing Commands

```
25  void run_command(int cmd_argc, char **cmd_argv, int line_count, char **lines) {
26      char *exec_argv[MAXARG];
27      int i, j = 0;
28
29      for (i = 0; i < cmd_argc; i++) {
30          exec_argv[j++] = cmd_argv[i];
31      }
32      for (i = 0; i < line_count; i++) {
33          exec_argv[j++] = lines[i];
34      }
35
36      exec_argv[j] = 0;
37
38      if (fork() == 0) {
39          exec(exec_argv[0], exec_argv);
40          printf("xargs: exec failed for command %s\n", exec_argv[0]);
41          exit(1);
42      } else {
43          wait(0);
44      }
45  }
```