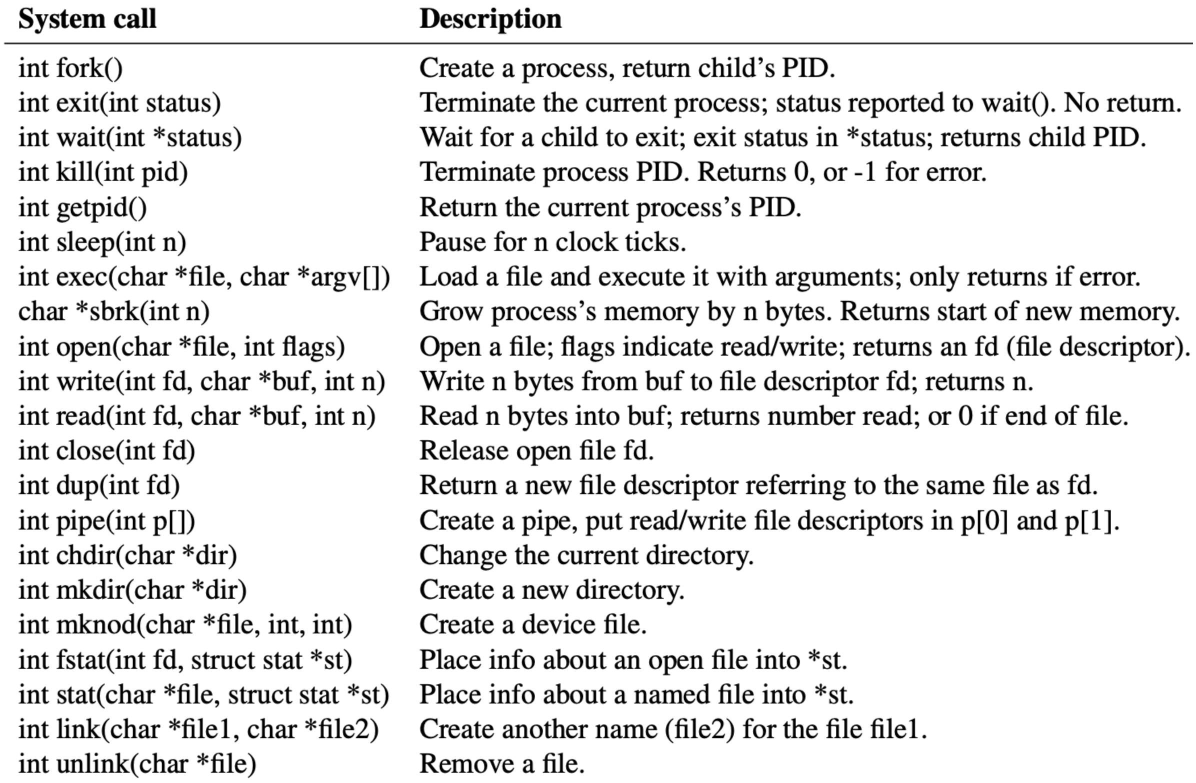
https://pdos.csail.mit.edu/6.828/2021/labs/util.html

Syscalls



System call	Description
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.

Figure 1.2: Xv6 system calls. If not otherwise stated, these calls return 0 for no error, and -1 if there's an error.

Note:

for each file descriptor, there is a unique offset associated with it and each `read` syscall picks up the offset of previous `read` or `write` syscalls.

Command type and shell

1. Simple Execution Command (`execcmd`)

Shell Command:

```
ls
```

This command simply lists the files and directories in the current working directory.

```
  case EXEC:
    ecmd = (struct execcmd*)cmd;
    if(ecmd->argv[0] == 0)
      exit(1);
    exec(ecmd->argv[0], ecmd->argv);
    fprintf(2, "exec %s failed\n", ecmd->argv[0]);
    break;
```

check if the first argument is null

if not directly run the `exec` syscall

2. Redirection Command (`redircmd`)

Shell Command:

```
ls > out.txt
```

This command lists the files and directories and redirects the output to a file named `out.txt`.

```
  case REDIR:
    rcmd = (struct redircmd*)cmd;
    close(rcmd->fd);
    if(open(rcmd->file, rcmd->mode) < 0){
      fprintf(2, "open %s failed\n", rcmd->file);
      exit(1);
    }
    runcmd(rcmd->cmd);
    break;
```

close the fd to be redirected, which denotes the standard output of `ls`. The next open will return the lowest number file descriptor, so file descriptor 2 is assigned to the file nad subsequent write to file descriptor 2 will go to the file.

3. Pipeline Command (`pipecmd`)

Shell Command:

```
ls | grep txt
```

This command lists the files and directories, pipes the output to `grep`, and filters the list to show only those lines containing the string "txt".

```
  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
      panic("pipe");
    if(fork1() == 0){
      close(1);
      dup(p[1]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->left);
    }
    if(fork1() == 0){
      close(0);
      dup(p[0]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->right);
    }
    close(p[0]);
    close(p[1]);
```

```
      wait(0);
      wait(0);
      break;
```

forks two children, one for the left command and one for the right command. Redirect the standard output of the left command to the standard input of the right command.

4. List Command (`listcmd`)

Shell Command:

```
ls; echo done
```

This command first lists the files and directories, and then, regardless of the result of the `ls` command, it prints "done" to the terminal.

5. Background Command (`backcmd`)

Shell Command:

```
sleep 10 &
```

This command runs the `sleep 10` command in the background, which causes the shell to wait for 10 seconds before the command completes, allowing the user to continue using the shell immediately.

Mapping to the Provided Code

Below is the C code representation of each shell command using the provided functions:

Simple Execution Command

```
c
Copy code
struct cmd* exec = execcmd();
// Set up the command details (e.g., setting the command to "ls" and its
arguments)
```

Redirection Command

```
c
Copy code
struct cmd* subcmd = execcmd(); // Set up the sub-command details (e.g., setting
the command to "ls")
struct cmd* redir = redircmd(subcmd, "out.txt", "out.txt" + strlen("out.txt"),
O_WRONLY | O_CREAT, 1);
```

Pipeline Command

```
c
Copy code
struct cmd* left = execcmd();  // Set up the left command details (e.g., setting the command to "ls")
struct cmd* right = execcmd(); // Set up the right command details (e.g., setting the command to "grep txt")
struct cmd* pipe = pipecmd(left, right);
```

List Command

```
c
Copy code
struct cmd* list_left = execcmd();  // Set up the left command details (e.g., setting the command to "ls")
struct cmd* list_right = execcmd(); // Set up the right command details (e.g., setting the command to "echo done")
struct cmd* list = listcmd(list_left, list_right);
```

Background Command

```
c
Copy code
struct cmd* subcmd = execcmd(); // Set up the sub-command details (e.g., setting the command to "sleep 10")
struct cmd* back = backcmd(subcmd);
```

Mknod

```
mknod("/console", 1, 1);
```

Mknod creates a special file that refers to a device. Associated with a device file are the major and minor device numbers (the two arguments to mknod), which uniquely identify a kernel device. When a process later opens a device file, the kernel diverts read and write system calls to the kernel device implementation instead of passing them to the file system.

Sleep

```
#include "kernel/types.h"
#include "user/user.h"

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        printf("Usage: Sleep t, where t is the time to sleep in seconds.\n");
        exit(-1);
    }
    int t = atoi(argv[1]);
    sleep(t);
    exit(0);
}
```

pingpong

parent send "114514" to child over pipe `p1` and child respond `received ping`. Child send "114514" over pipe `p2` and parent respond `received pong`

```
#include "kernel/types.h"
#include "user/user.h"

int main()
{
    int p1[2];
    pipe(p1); // parent to child
    int p2[2];
    pipe(p2);        // child to parent
    if (fork() == 0) // child
    {
        char bufRead[100];
        if (read(p1[0], bufRead, 100) > 0)
        {
            if (strcmp(bufRead, "114514") == 0)
            {
                fprintf(1, "%d: received ping\n", getpid());
            }
            else
            {
                fprintf(1, "%d: received wrong ping message\n", getpid());
            }
        }
        else
        {
            exit(-1);
        }
        char *bufWrite = "114514";
        if (write(p2[1], bufWrite, strlen(bufWrite)) == strlen(bufWrite))
        {
            exit(0);
        }
    }
    else // parent
    {
        char *bufWrite = "114514";
        if (write(p1[1], bufWrite, strlen(bufWrite)) == strlen(bufWrite))
        {
            char bufRead[100];
            if (read(p2[0], bufRead, 100) > 0)
            {
                if (strcmp(bufRead, "114514") == 0)
                {
                    fprintf(1, "%d: received pong\n", getpid());
                }
                else
                {
                    fprintf(1, "%d: received wrong ping message\n", getpid());
                }
            }
            else
            {
                exit(-1);
            }
        }
    }
    exit(0);
}
```