

OS Lab 8

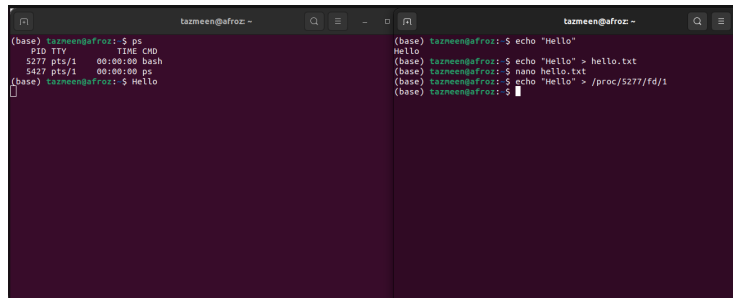
Tazmeen Afroz

1 Redirecting Output to Another Terminal

To send the output of a command from one terminal to another, you can use the following command:

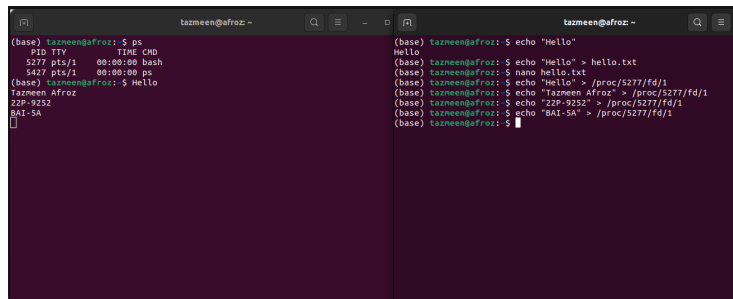
```
echo "Hello" > /proc/X/fd/1
```

Here, X is the process ID of the target terminal.



```
(base) tazmeen@afroz: ~$ ps
PID TTY      TIME CMD
5277 pts/1    00:00:00 bash
5427 pts/1    00:00:00 ps
(base) tazmeen@afroz: ~$ echo Hello
Hello
(base) tazmeen@afroz: ~$ echo "Hello"
Hello
(base) tazmeen@afroz: ~$ echo "Hello" > hello.txt
(base) tazmeen@afroz: ~$ nano hello.txt
(base) tazmeen@afroz: ~$ echo "Hello" > /proc/5277/fd/1
(base) tazmeen@afroz: ~$
```

Figure 1: X Process Example



```
(base) tazmeen@afroz: ~$ ps
PID TTY      TIME CMD
5277 pts/1    00:00:00 bash
5427 pts/1    00:00:00 ps
Tazmeen Afroz
229-9232
041-54
(base) tazmeen@afroz: ~$ echo Hello
Hello
(base) tazmeen@afroz: ~$ echo "Hello" > hello.txt
(base) tazmeen@afroz: ~$ nano hello.txt
(base) tazmeen@afroz: ~$ echo "Hello" > /proc/5277/fd/1
(base) tazmeen@afroz: ~$ echo "Tazmeen Afroz" > /proc/5277/fd/1
(base) tazmeen@afroz: ~$ echo "229-9232" > /proc/5277/fd/1
(base) tazmeen@afroz: ~$ echo "041-54" > /proc/5277/fd/1
(base) tazmeen@afroz: ~$
```

Figure 2: Output after redirecting

2 List of All Processes

Navigate to the `/proc/` directory using the following command:

```
cd /proc/
```

```
(base) tazmeen@afroz:~$ cd /proc/
(base) tazmeen@afroz:/proc$ ls
1          1487 1645 23 38 5209 666  asound      meminfo
16         17 1645 238 39 5227 666  bootconfig mtrc
1814      1519 1649 24 41 5277 669  buddyinfo  modules
1821      1525 165 26 40 53 67  hwp         mounts
21        1526 1657 27 402 5384 671  cgroups    mtd
318       1555 1661 2737 4154 56 472  cmdline    mtrr
3181      1536 1674 28 42 5428 678  consoles   net
32        154 1694 2829 4338 1081 60  cpufreq     pagetypeinfo
3291      1541 17 2866 4398 5598 681  crypto      partitions
3384      1548 1734 289 48 56 684  devices     procps
3296      155 1737 29 445 5001 687  diskstats   schedstat
33        1561 1739 2916 448 5621 690  dma          acpi
3261      1568 1747 2995 449 5655 696  driver       self
3340      1570 1756 3 45 57 699  /dev/shm     slabinfo
3311      1579 1798 30 451 5789 7  execdomains  softirqs
3267      1586 18 3002 452 5789 70  fb           stat
3316      1596 1887 3014 453 58 700  filesystems  swaps
3319      16 19 3025 4657 5849 703  fs           smp
3386      1607 1906 317 47 5884 705  interrupts  sysrq-trigger
3339      1609 1937 3189 48 59 706  iomem        sysvipc
3340      161 1941 31 60 5941 71  lparports    thread-self
3343      161 1959 3259 4903 6 72  /           timer_list
3340      1620 198 33 499 60 720  kallsyms     tty
3299      1621 1982 34 1 6084 73  kcore        uptime
3373      1624 199 343 58 61 74  keys         version
3376      1625 2 348 583 6104 749  key-users    version_signature
24        1626 20 3460 607 6224 756  kmsg         vmallocinfo
3437      1636 2068 3483 51 6153 81  kpagecgroup vmtstat
3441      1632 2012 35 511 62 813  kpagecount   zoneinfo
3467      1634 2060 36 514 66 814  kpageflags
3455      1636 2069 3617 5162 65 895  latency_stats
3427      1639 31 5423 5164 66 90  loadavg
3477      164 22 575 5197 663 897  locks
3484      1642 2310 3758 52 664  acpi         mdstat
```

Figure 3: List of processes in the `/proc/` directory

3 Navigating to a Specific Process

Go to the directory of a specific process by using the following command:

```
cd /proc/5277/
```

```
(base) tazmeen@afroz:~$ cd /proc/5277/
(base) tazmeen@afroz:/proc/5277$ ls
arch_status  fdinfo      ns          smaps_rollop
attr         gid_map     numa_maps  stack
autogroup    io          oom_adj     stat
auxv         ksm_merging_pages oom_score  statm
cgroup       ksm_stat    oom_score_adj status
clear_refs   latency     pagemap     syscall
cmdline      limits      patch_state task
comm         loginuid    personality timens_offsets
coredump_filter map_files   projid_map  timers
cpu_resctrl_groups maps         root        timerslack_ns
cpuset       mem         sched       uid_map
cwd          mountinfo   schedstat   wchan
environ      mounts      sessionid
exe          mountstats  setgroups
fd           net         smaps
```

Figure 4: Navigating to Process 5277

4 File Descriptor Directory of a Process

Navigate to the `fd` (File Descriptor) directory of the process:

```
cd fd
```

```
(base) tazmeen@afroz:/proc/5277$ cd fd
(base) tazmeen@afroz:/proc/5277/fd$ ls
0 1 2 255
(base) tazmeen@afroz:/proc/5277/fd$
```

Figure 5: Navigating to `fd` directory of the process

```
ls
```

```
(base) tazmeen@afroz:~$ ps
PID TTY          TIME CMD
5277 pts/1    00:00:00 bash
5427 pts/1    00:00:00 ps
(base) tazmeen@afroz:~$ cd /proc/5277/
(base) tazmeen@afroz:/proc/5277$ ls
arch_status  autogroup  auxv       cgroup     cmdline    comm       coredump_filter
cpu_restr1_groups  cpuset     environ    exe        fd          gld_map    io           ksm_merging_pages
kvm_stat     latency    loginuid   maps       mem         mountinfo  mountstats  numa_maps
oom_adj      oom_score  oom_score_adj  patch_state  personality  projfd_map  rseq        sched
schedstat    sessionid  setgroups   smaps      stack       stat       status      sysctl
syscall      timers     timerslack_ns  uid_map    wchan
(base) tazmeen@afroz:/proc/5277$ ls
0 1 2 255
```

Figure 6: Listing file descriptors in the `fd` directory

5 C Code for File Creation

Execute the following C code to create a file:

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char* argv[])
{
    char *path = argv[1];
    int fd = open(path, O_WRONLY | O_EXCL | O_CREAT);
    if (fd == -1)
    {
        printf("Error: - File - not - Created\n");
        return 1;
    }
}
```

```
    return 0;
}
```

Compile and run the program:

```
gcc first.c -o first
./first createThisFile
```

A terminal window titled 'tazmeen@afroz: ~/os_lab_08' showing the execution of a C program. The user runs 'gcc first.c -o first', then './first createThisFile'. The program outputs 'createThisFile first first.c' in red text. The user then runs 'ls' and the terminal shows 'first first.c'.

```
tazmeen@afroz: ~/os_lab_08
(base) tazmeen@afroz:~/os_lab_08$ gcc first.c -o first
(base) tazmeen@afroz:~/os_lab_08$ ./first createThisFile
createThisFile first first.c
(base) tazmeen@afroz:~/os_lab_08$ ls
first first.c
(base) tazmeen@afroz:~/os_lab_08$
```

Figure 7: Output after running the file creation code

6 File Size Question

Question: What is the size of the file created?

Answer: The file size is 0 bytes because no data was written to it, only created.

7 Understanding the open() System Call

The `open()` system call returns a file descriptor or an error code:

- **Success:** A non-negative integer representing the file descriptor.
- **Failure:** -1, with the `errno` variable set to indicate the specific error.

Common reasons for failure include:

- File doesn't exist
- Permission denied
- Too many open files

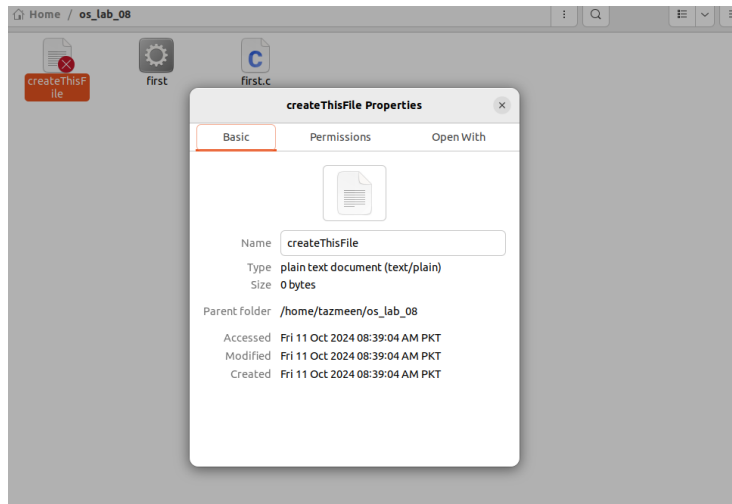


Figure 8: File size is 0 bytes

```
(base) tazmeen@afroz:~/os_lab_08$ gcc first.c -o first
(base) tazmeen@afroz:~/os_lab_08$ ./first testfile
(base) tazmeen@afroz:~/os_lab_08$ gcc first.c -o first
(base) tazmeen@afroz:~/os_lab_08$ ./first test
fd value : ,3(base) tazmeen@afroz:~/os_lab_08$
```

Figure 9: fd return value ,3 used for normal creation of file

8 Command-Line Arguments: argc and argv[]

When a C program is executed, it can take input from the command line. These inputs are passed to the program through the parameters **argc** (Argument Count) and **argv[]** (Argument Vector).

8.1 argc (Argument Count)

- **Type:** int
- **Purpose:** It represents the number of command-line arguments passed to the program.
- **Value:** argc is always at least 1, because the program name itself is considered the first argument.

8.2 argv[] (Argument Vector)

- **Type:** char* argv[] or char** argv

- **Purpose:** It's an array of strings containing the actual command-line arguments.
- **Contents:**
 - `argv[0]` is always the name of the program.
 - `argv[1]`, `argv[2]`, etc., are the additional arguments provided on the command line.

Code Snippet

Listing 1: C Program

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        printf("Error: -Run- like -this:-");
        printf("%6s -name-of-new-file\n", argv[0]);
        return 1;
    }
    char *path = argv[1];
    int i = 0;
    while(i < 2)
    {
        int fd = open(path, O_WRONLY | O_CREAT);
        printf("Created! -Descriptor- is -%d\n", fd);
        close(fd); // Comment this line later for the test
        i++;
    }
    return 0;
}
```

Test: Comment Out `close(fd)`

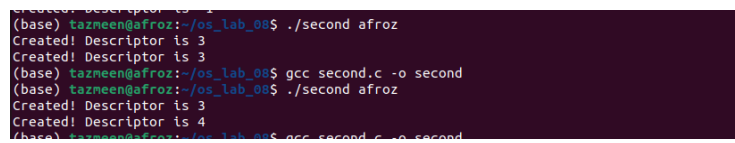
Comment out the line `close(fd);` and then compile and run the program again.

Before Commenting

Created! Descriptor is 3
Created! Descriptor is 3

After Commenting

Created! Descriptor is 3
Created! Descriptor is 4

A terminal window with a dark background and light green text. It shows a sequence of commands and their outputs. The first command is './second afroz', which outputs 'Created! Descriptor is 3'. The second command is 'gcc second.c -o second'. The third command is './second afroz', which outputs 'Created! Descriptor is 3'. The fourth command is 'gcc second.c -o second', which outputs 'Created! Descriptor is 4'. The terminal text is as follows:

```
Created! Descriptor is 3
(base) tazneen@afroz:~/os_lab_08$ ./second afroz
Created! Descriptor is 3
Created! Descriptor is 3
(base) tazneen@afroz:~/os_lab_08$ gcc second.c -o second
(base) tazneen@afroz:~/os_lab_08$ ./second afroz
Created! Descriptor is 3
Created! Descriptor is 4
(base) tazneen@afroz:~/os_lab_08$ gcc second.c -o second
```

Figure 10: Output

Explanation of Output

When the `close(fd);` line is commented out, the file descriptor is not released after each file operation. As a result, the second call to `open()` assigns the next available file descriptor (which is 4, since descriptor 3 is still in use).

When the line is not commented out, the file descriptor is closed after each iteration, and the OS reuses the same descriptor (3) for both operations.

Code Snippet

Listing 2: C Program

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

char* get_timeStamp()
{
    time_t now = time(NULL);
    return asctime(localtime(&now));
}
```

```

int main(int argc, char* argv[])
{
    char *filename = argv[1];
    char *timeStamp = get_timeStamp();
    int fd = open(filename, O_WRONLY | O_APPEND | O_CREAT, 0666);
    size_t length = strlen(timeStamp)-5;
    write(fd, timeStamp, length);
    close(fd);
    return 0;
}

```

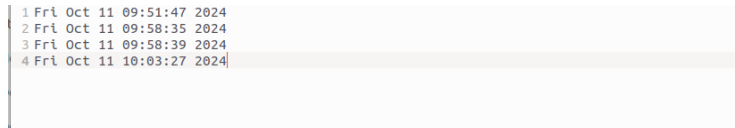
Questions and Answers

Q1: What is 0666 that is specified in the open() call? What does it mean?

A1: The number 0666 represents the file permission mode. It means that the file will be created with read and write permissions for the owner, group, and others. In symbolic notation, it corresponds to `rw-rw-rw-`.

Q2: What is O_APPEND doing in the same call?

A2: The flag `O_APPEND` ensures that data is always written at the end of the file, without modifying the existing content. When this flag is used, every write operation appends the data to the current end of the file.



```

1 Fri Oct 11 09:51:47 2024
2 Fri Oct 11 09:58:35 2024
3 Fri Oct 11 09:58:39 2024
4 Fri Oct 11 10:03:27 2024

```

Figure 11: Output

Q3: Modify the following line in the code and then compile and run the program and check its output. What has happened?

textbfA3: By reducing the length by 5, you're truncating the last 5 characters of the timestamp.

Code Snippet

Listing 3: C Program

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        printf("Error: Run like this: ");
        printf("%6s name-of-existing-file\n", argv[0]);
        return 1;
    }
    char *path = argv[1];
    int fd = open(path, ORDONLY);
    if (fd == -1)
    {
        printf("File does not exist\n");
        return 1;
    }
    char buffer[200];
    read(fd, buffer, sizeof(buffer)-1);
    printf("Contents of File are:\n");
    printf("%s\n", buffer);
    close(fd);
    return 0;
}
```

Explanation of Output

The program opens a file specified by the user as a command-line argument, reads its contents, and prints them to the console.

Case 1: Invalid Input

If the user does not provide the correct argument (file name), the program displays the error:

```
Error: Run like this: ./program_name name-of-existing-file
```

Case 2: File Not Found

If the file does not exist, the output will be:

File does not exist

Case 3: File Found

When the file exists, the program will read and display its contents. The output will be:

Contents of File are:
<file contents here>

Output Image

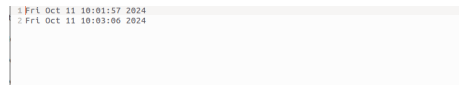


Figure 12: Program Output Displaying File Contents