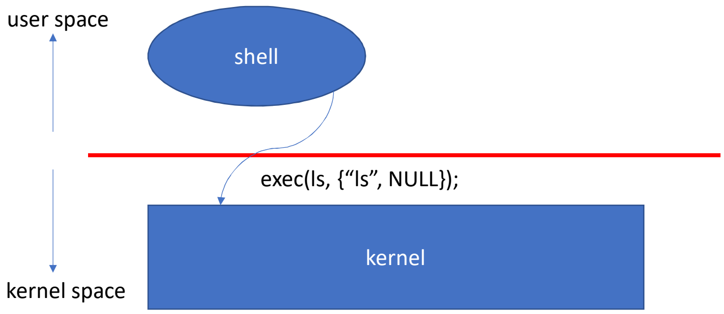
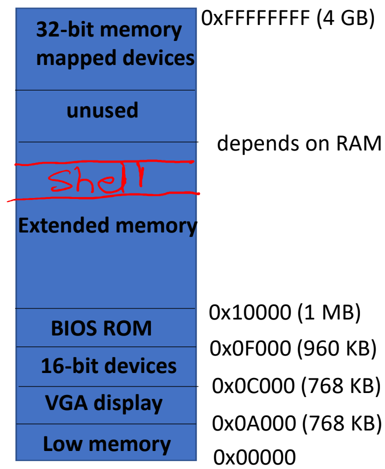
- Read chapter-0 from the xv6 book

before exec

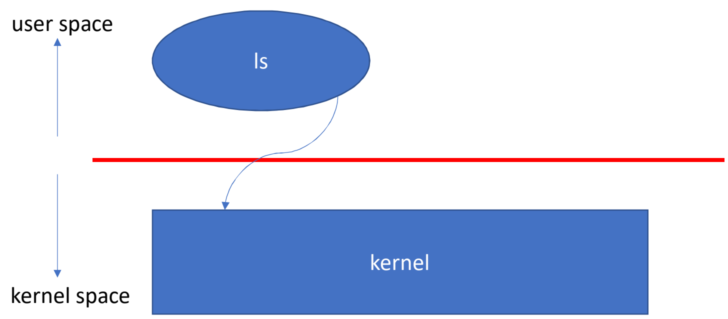


before exec

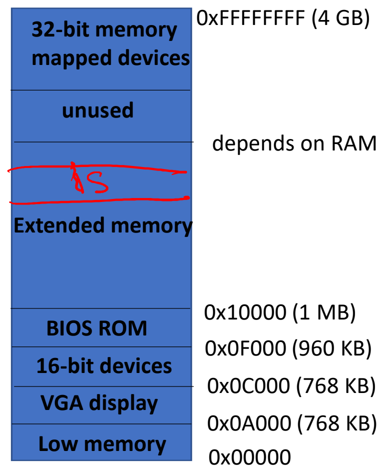


32-bit memory mapped devices
unused
shell
Extended memory
BIOS ROM
16-bit devices
VGA display
Low memory
0xFFFFFFFF (4 GB)
depends on RAM
0x10000 (1 MB)
0x0F000 (960 KB)
0x0C000 (768 KB)
0x0A000 (768 KB)
0x00000

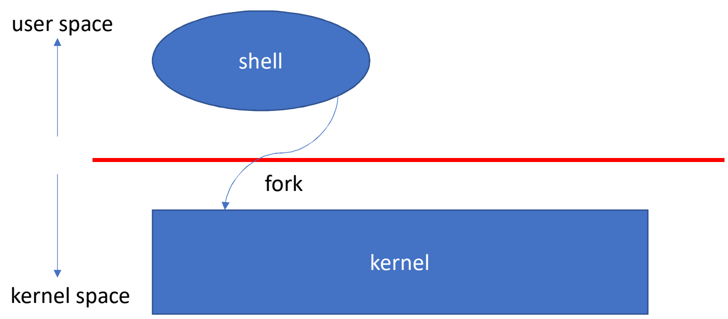
after exec



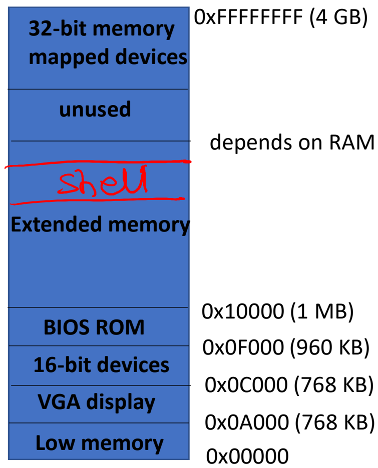
after exec


32-bit memory mapped devices
unused
OS
Extended memory
BIOS ROM
16-bit devices
VGA display
Low memory
0xFFFFFFFF (4 GB)
depends on RAM
0x10000 (1 MB)
0x0F000 (960 KB)
0x0C000 (768 KB)
0x0A000 (768 KB)
0x00000

before fork

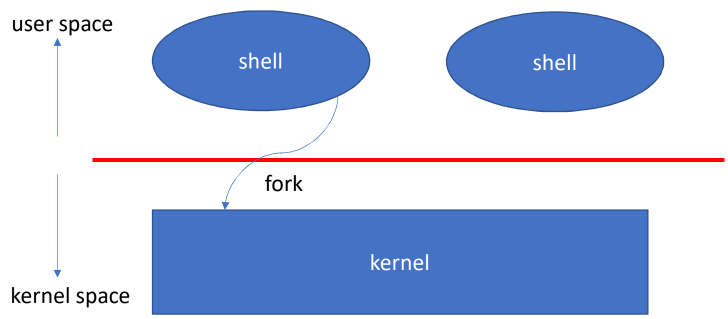


before fork

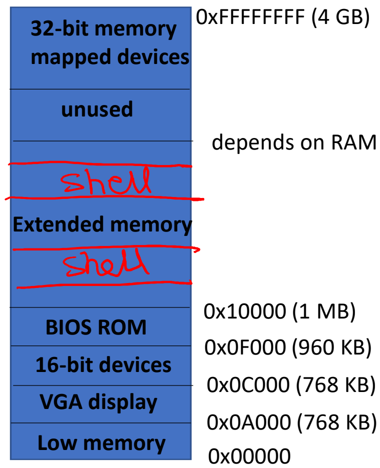


32-bit memory mapped devices
unused
Shell
Extended memory
BIOS ROM
16-bit devices
VGA display
Low memory
0xFFFFFFFF (4 GB)
depends on RAM
0x10000 (1 MB)
0x0F000 (960 KB)
0x0C000 (768 KB)
0x0A000 (768 KB)
0x00000

after fork



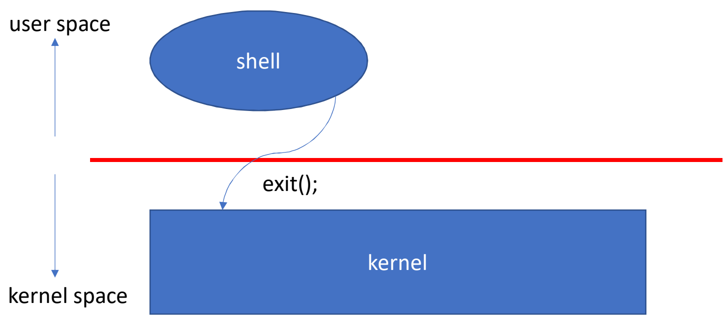
after fork


32-bit memory mapped devices
0xFFFFFFFF (4 GB)
unused
depends on RAM
shell
Extended memory
shell
0x10000 (1 MB)
BIOS ROM
0x0F000 (960 KB)
16-bit devices
0x0C000 (768 KB)
VGA display
0x0A000 (768 KB)
Low memory
0x00000

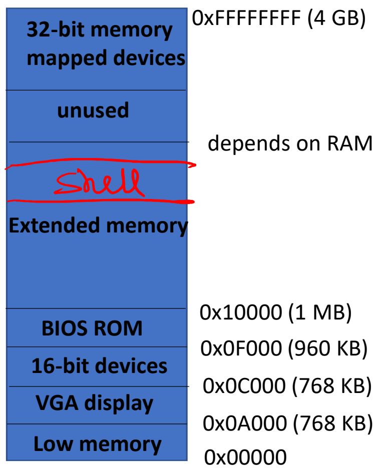
fork

- fork system call creates a new process that is identical to the child process
  - new memory (heap+stack+code+global) is allocated for the child process
  - the entire memory of the parent process is copied to the child process
  - the register values of the parent process are copied to the corresponding registers in the child process
  - after the fork returns changing the value of a variable in the parent doesn’t change the variable’s value in child and vice versa

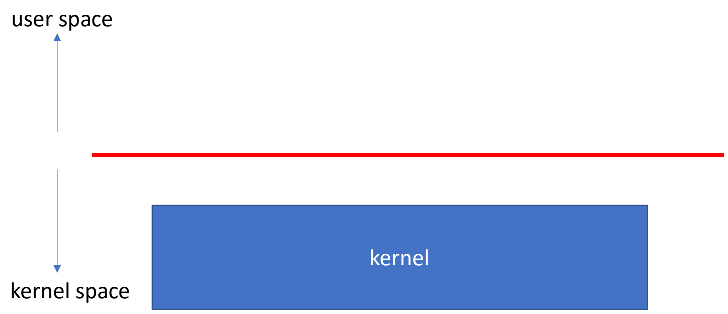
before exit



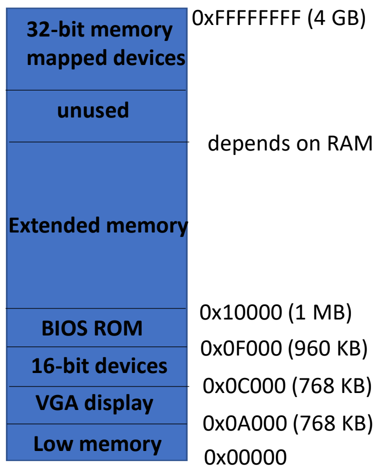
before exit


32-bit memory mapped devices
0xFFFFFFFF (4 GB)
unused
depends on RAM
Shell
Extended memory
0x10000 (1 MB)
BIOS ROM
0x0F000 (960 KB)
16-bit devices
0x0C000 (768 KB)
VGA display
0x0A000 (768 KB)
Low memory
0x00000

after exit



after exit

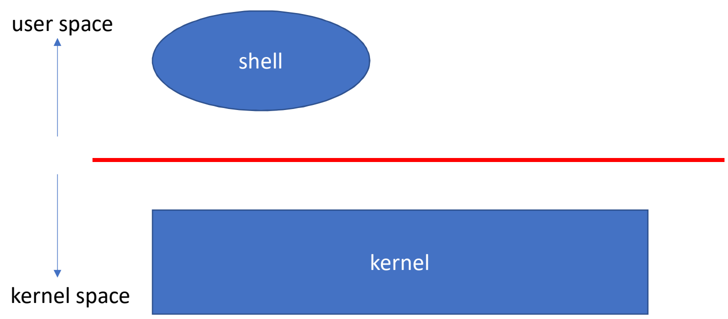

32-bit memory mapped devices
unused
Extended memory
BIOS ROM
16-bit devices
VGA display
Low memory
0xFFFFFFFF (4 GB)
depends on RAM
0x10000 (1 MB)
0x0F000 (960 KB)
0x0C000 (768 KB)
0x0A000 (768 KB)
0x00000

System calls

- exec
- fork
- exit
- wait
  - the parent waits until one of the children exits

shell

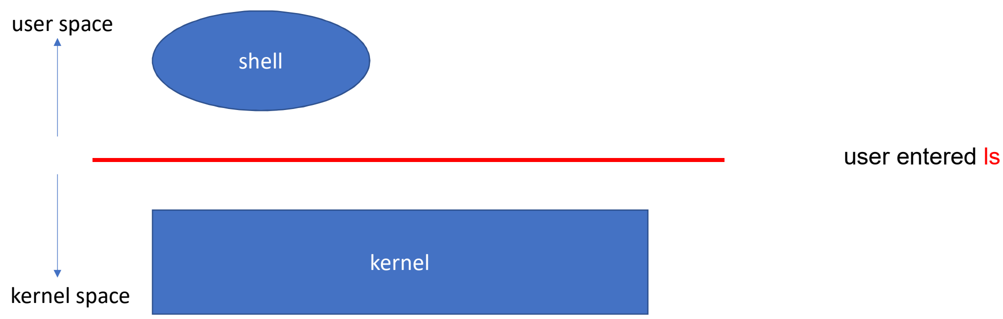
$



shell is waiting for the user input

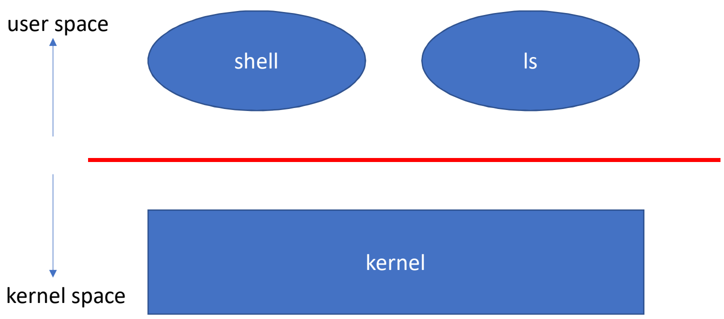
shell

$ ls



shell

$ ls
a.out
a.c

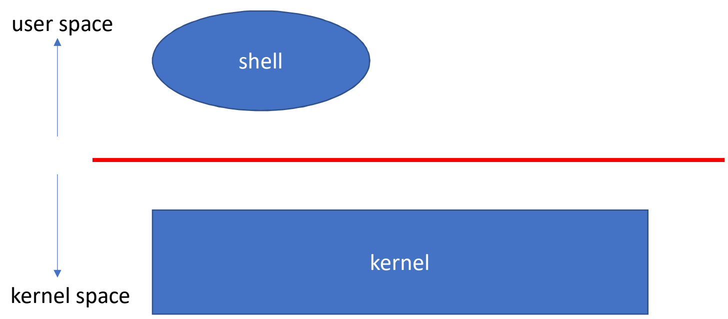


ls is executing.

shell is waiting
for ls to finish

shell

$ ls
a.out
a.c
$



ls has exited

shell is waiting for the user input

shell

```
while (1) {
  write(1, "$ ", 2);
  read_command(0, cmd, args);
  pid = fork();
  if (pid == 0) {
    exec(cmd, args);
  } else if (pid > 0) {
    wait();
  } else
    printf("Failed to fork\n");
}
```

current working directory is copied to the child during fork.

ls program gives the same result in the parent and the child.

The child process does the exec system call to transform itself into an executable (in cmd) program. The parent (shell) process waits until the child (cmd) terminates.

creat

- int creat(const char *pathname, int flags);
  - creates a new file
  - if the file already exists, truncates it
  - permissions (who can access the file) are set according to flags
  - On successful, an integer file descriptor is returned
  - the file descriptor can be used to write to file

file descriptor

- file descriptor can refer to a file as well as a device

- file descriptor 0 refers to the input device (e.g., keyboard)
  - 0 points to standard input

- file descriptor 1, 2 refers to the output device (e.g., a display screen)
  - 1 points to standard output
  - 2 points to standard error

write

- ssize_t write(int fd, const void *buf, size_t count);
  - write up to count bytes from buf to file referred by the file descriptor fd
  - returns the number of bytes that were written to the file

- To know more
  - type man 2 write in your terminal

read

- ssize_t read(int fd, void *buf, size_t count);
  - read up to count bytes from the file referred by file descriptor fd into buf
  - returns the number of bytes read
  - If the fd is standard input, read system call waits for the user input

- To know more
  - type man 2 read in your terminal

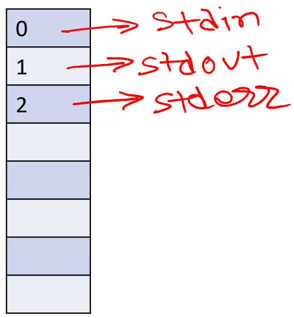
demo

- creat
- write
- write to console
- read from console

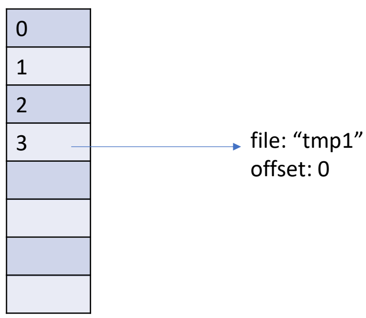
File descriptor table

- A file descriptor is a small integer
- For every process, OS maintains a table of descriptors
  - open files, devices, etc.
- 0, 1, and 2 are reserved for standard input, standard output, and standard error
- The process may create/open new files using creat and open system calls
  - creat and open system calls add a new entry to the file descriptor table

File descriptor table



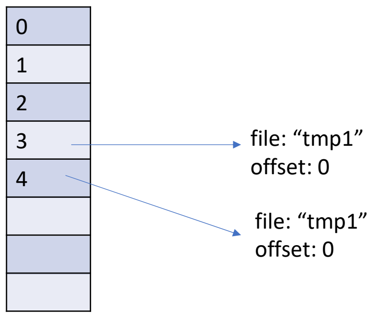
File descriptor table



fd1 = creat(“tmp1”, 0666);

The create system call allocates an available descriptor in the file descriptor table. The file descriptor table entry points to a structure that contains an offset field. The offset is advanced by the number of bytes written during the write system call. The offset is also modified during the read if the file was opened for reading.

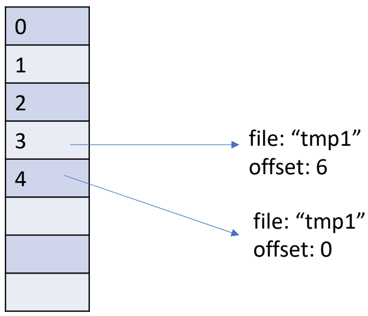
File descriptor table




```
fd1 = creat(“tmp1”, 0666);
fd2 = creat(“tmp1”, 0666);
```

Two descriptors can point to the same file, but their offsets are different.

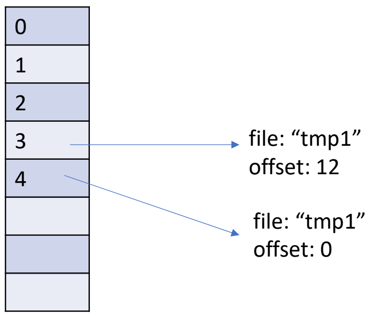
File descriptor table




```
fd1 = creat(“tmp1”, 0666); // fd1 == 3
fd2 = creat(“tmp1”, 0666); // fd2 == 4
write(fd1, “hello ”, 6);
```

Write to fd1 updates the offset in the target pointer to struct at index 3 in the file descriptor table.

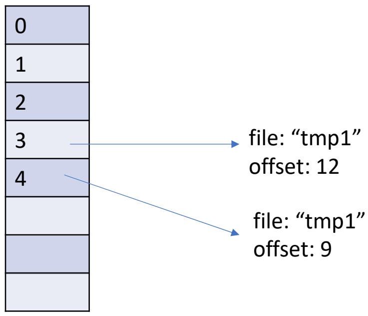
File descriptor table




```
fd1 = creat(“tmp1”, 0666);
fd2 = creat(“tmp1”, 0666);
write(fd1, “hello ”, 6);
write(fd1, “world\n”, 6);
```

Write to fd1 updates the offset in the target pointer to struct at index 3 in the file descriptor table.

File descriptor table




```
fd1 = creat(“tmp1”, 0666);
fd2 = creat(“tmp1”, 0666);
write(fd1, “hello ”, 6);
write(fd1, “world\n”, 6);
write(fd2, “overwrite”, 9);
```

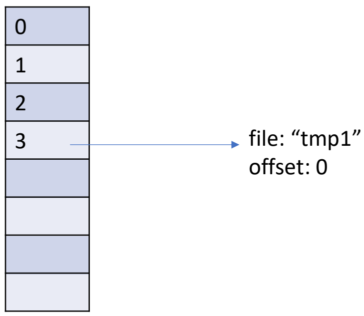
Write to fd2 updates the offset in the target pointer at index 4 in the file descriptor table. This write will overwrite the contents of the tmp1 file, starting at offset 0.

I/O redirection

```
int main() {
  write(1, "hello\n", 6);
  return 0;
}
```

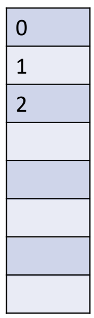
a.out
./a.out
hello
./a.out > tmp
creates a new file tmp
writes "hello\n" in tmp

File descriptor table



fd1 = creat(“tmp1”, 0666);
creat always returns the lowest available descriptor in the file descriptor table

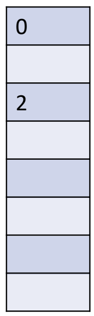
File descriptor table



fd1 = creat("tmp1", 0666);
creat always returns the lowest available descriptor in the file descriptor table

close(fd1);
close system call removes the file descriptor fd1 from the file descriptor table

File descriptor table

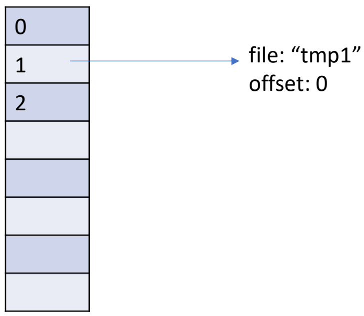


fd1 = creat("tmp1", 0666);
creat always returns the lowest available descriptor in the file descriptor table

close(fd1);
close system call removes the file descriptor fd1 from the file descriptor table

close(1);

File descriptor table



fd1 = creat(“tmp1”, 0666);
creat always returns the lowest available descriptor in the file descriptor table

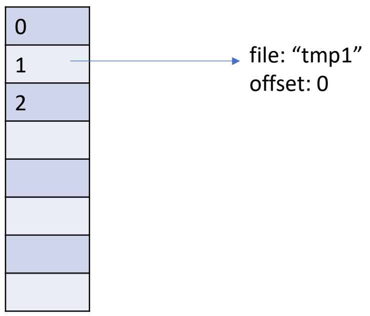
close(fd1);
close system call removes the file descriptor fd1 from the file descriptor table

close(1);
fd1 = creat(“tmp1”, 0666);
// fd1 is 1 at this point

fork

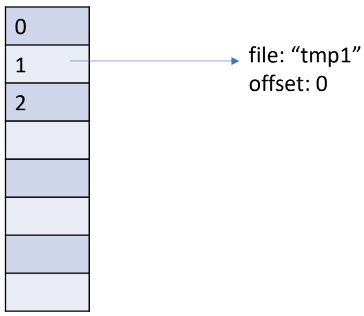
- The child process inherits the file descriptor table from the parent
- The OS creates an identical file descriptor table for the child

Before fork



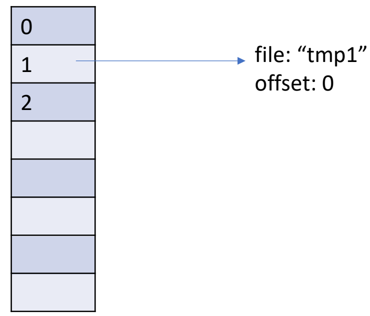
After fork

After exec



Parent

On exec, the process retains file descriptors.



Child after exec

./a.out > tmp1

```
int main() {
  write(1, “hello\n”, 6);
  return 0;
}
```

```
a.out
./a.out
hello
./a.out > tmp1
creates a new file tmp1
writes “hello\n” in tmp1
```

```
while (1) {
  write(1, “$ ”, 2);
  read_command(0, cmd, args);
  pid = fork();
  if (pid == 0) {
    close(1);
    creat(“tmp1”, 0666);
    exec(cmd, args);
  } else {
    wait();
  } else
    printf(“Failed to fork\n”);
}
```

The two statements highlighted in red will make sure that the file descriptor 1 is now pointing to the tmp1 file. When the a.out program is loaded (after the exec system call), the file descriptor table remains the same as the child process. All writes to descriptor 1 will be written to tmp1.

./a.out < tmp1

```
int main() {
  read(0, buf, 128);
  write(1, buf, strlen(buf));
  return 0;
}
```

a.out
./a.out
waiting for input

./a.out < tmp1

```
int main() {
  read(0, buf, 128);
  write(1, buf, strlen(buf));
  return 0;
}
```

a.out
./a.out
waiting for input
hello\n   // user enters
writes “hello\n” to the console

./a.out < tmp1

```
int main() {
  read(0, buf, 128);
  write(1, buf, strlen(buf));
  return 0;
}
```

a.out
./a.out
waiting for input
hello\n // user enters
writes “hello\n” to the console

./a.out < tmp1
writes contents of tmp1 to the console

./a.out < tmp1

```
int main() {
  read(0, buf, 128);
  write(1, buf, strlen(buf));
  return 0;
}
```

a.out
./a.out
waiting for input
hello\n   // user enters
writes “hello\n” to the console

./a.out < tmp1
writes “contents of tmp1” to the console

```
while (1) {
  write(1, "$ ", 2);
  read_command(0, cmd, args);
  pid = fork();
  if (pid == 0) {
    close(0);
    open("tmp1", O_RDONLY);
    exec(cmd, args);
  } else {
    wait();
  } else
    printf("Failed to fork\n");
}
```

This command redirects the contents of tmp1 to the stdin of a.out. Again using the trick highlighted in red, we can redirect the input from tmp1 to the standard input of a.out.

./a.out > tmp1 2>&1

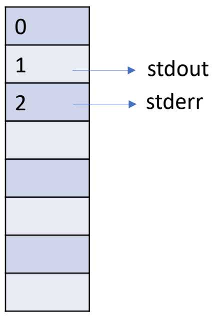
```
int main() {
  write(1, "hello ", 6);
  write(2, "world\n", 6);
  return 0;
}
```

a.out
./a.out
hello world
./a.out > tmp1 2>&1
creates a new file tmp1
writes "hello world\n" in tmp1

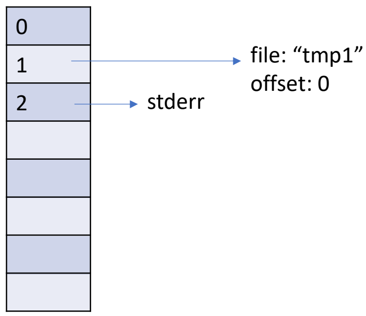
```
while (1) {
  write(1, "$ ", 2);
  read_command(0, cmd, args);
  pid = fork();
  if (pid == 0) {
    close(1);
    creat("tmp1", 0666);
    close(2);
    creat("tmp1", 0666);
    exec(cmd, args);
  } else {
    wait();
  } else
    printf("Failed to fork\n");
}
```

This command redirects the writes to file descriptors 1 and 2 to the tmp1 file. However, the code highlight in red is not suitable for this purpose. In this code, stdout (1) and stderr (2) will overwrite the contents of each other.

File descriptor table



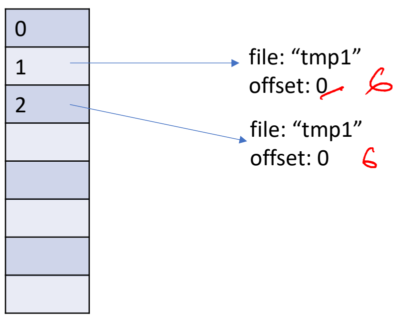
File descriptor table



close(1)
creat(“tmp1”, 0666)

File descriptor table

“hello
world\n”

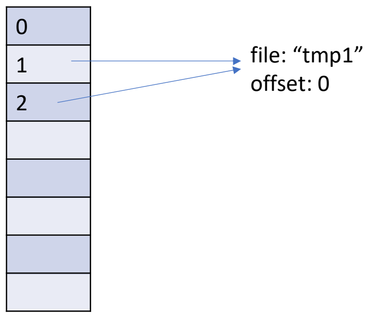


close(1)
creat(“tmp1”, 0666)
close(2)
creat(“tmp1”, 0666)

dup

- int dup(int oldfd);
  - allocates a new descriptor and creates an alias of oldfd in the new file descriptor

File descriptor table



close(2);
dup(1); // returns 2

1 and 2 points to same structure that contains the offset

The dup command takes a file descriptor as input and allocates a new file descriptor. The dup system call make sure that both input and new file descriptor are pointing to the same structure.

./a.out > tmp1 2>&1

```
int main() {
  write(1, "hello ", 6);
  write(2, "world\n", 6);
  return 0;
}
```

a.out
./a.out
hello world
./a.out > tmp1 2>&1
create a new file tmp1
write "hello world\n" in tmp1

```
while (1) {
  write(1, "$ ", 2);
  read_command(0, cmd, args);
  pid = fork();
  if (pid == 0) {
    close(1);
    creat("tmp1", 0666);
    close(2);
    dup(1);
    exec(cmd, args);
  } else {
    wait();
  } else
    printf("Failed to fork\n");
}
```

The code highlighted in red is the correct solution.

pipe

- pipe system call returns a pair of descriptor

- The first descriptor can be used for reading; the second descriptor can be used for writing

```
int p[2];
pipe(p);
p[0]  → reading
p[1] → writing
```

pipe

```
int p[2];
pipe(p);
```

p[1] → writing

p[0] → reading

pipe

- If the input file descriptor in the read or write system call is pipe
  - if no data is available, read system call waits until some data is written at the write end or all the file descriptors at the write end are closed
  - If the pipe is full, write system call waits until some data is consumed at the read end or all the file descriptors at the read end are closed

- read and write semantics for files are non-blocking

pipe

```
int p[2];
pipe(p);
write(p[1], “hello”, 5);
read(p[0], buf, 5);
// buf = “hello”
```

Inter process communication (IPC)

```
int fd[2];
pipe(fd);
pid = fork();
if (pid > 0)
  write(fd[1], "hello", 5);      // parent sends "hello"
else
  read(fd[0], buf, 5);           // child receives "hello"
```

What happens if the child gets scheduled before the parent?

It will wait for the parent to do the write.

pipe in shell

- demo

pipe in shell

```
int main() {
  return write(1, “hello\n”, 6);
}
```
a.out

```
int main() {
  char buf[128];
  read(0, buf, 128);
  return write(1, buf, strlen(buf));
}
```
b.out

pipe in shell

./a.out | ./b.out

No need of temporary files
file I/O is very slow
./a.out | ./b.out is equivalent to
./a.out > tmp
./b.out < tmp

./a.out | ./b.out

```
int fd[2];
pipe(fd);
pid = fork();
if (pid == 0) {
   char *param[2] = {"a.out", NULL};
   close(fd[0]);
   close(1);
   dup(fd[1]);
   close(fd[1]);
   exec("a.out", param);
} else {
   char *param[2] = {"b.out", NULL};
   close(fd[1]);
   close(0);
   dup(fd[0]);
   close(fd[0]);
   exec("b.out", param);
}
```

no disk I/O!
no deletion of temporary files
no disk space needed
parallel execution of pipeline stages
no rewriting of applications
faster IPC via memory.

This entire code is part of the child process. We can implement pipe like behavior using the trick highlighted in red.