COMP1521 26T1

Week 9 Lecture 1

Processes and Pipes

Announcements

- **Week 10 lab Practice Exam**
  - In person lab classes
    - Please attend one even if you are usually online
  - Questions are not released you need to attend to see them
  - Not worth marks
  - Regular lab questions for marks to do too
- **Assignment 2**: Get started ASAP if you have not already.
  - Start now while
    - Help sessions and forums are not super busy!
    - You still have a chance to get help in tut/labs!
  - Look at style rubric to see what we are looking for

Today’s Lecture

- Processes
  - execve recap
  - fork
  - wait
  - posix_spawn
- File redirection
  - dup and dup2
- Inter Process Communication
  - Pipes

Recap: Processes

- A **process** is an instance of an executing program.
- Each process has an execution state, defined by...
  - current values of CPU registers
  - current contents of its memory
  - information about open files (and other results of system calls)
- each process has a unique process ID, or PID: a positive integer, type **pid_t**, defined in <unistd.h>
- Each process has a parent process
- A process may have child processes

execve(2)

```
#include <unistd.h>
int execve(const char *file,
           char *const argv[],
           char *const envp[]);
```

- Run another program in place of the current process:
  - **file**: an executable — either a binary, or script starting with #!
  - **argv**: arguments to pass to new program
  - **envp**: environment variables to pass to new program
- if successful, exec does not return … where would it return to?
  - on error, returns -1 and sets errno

execve(2) replace yourself

- Most of the current process is re-initialized:
    - e.g. **new address space** is created - all variables lost
- Open file descriptors survive
    - e.g, stdin & stdout remain the same
- **PID unchanged**

Recap: Example: using execve()

```
int main(void) {
    char *echo_argv[] = {"/bin/echo","good-bye","cruel","world",NULL};
    execv("/bin/echo", echo_argv);
    // if we get here there has been an error
    perror("execv");
```

```
$ dcc exec.c
$ a.out
good-bye cruel world
$
```

Demo: exec.c

Execve Family

`execve(2)` is a syscall wrapper

There are a whole bunch of related convenience functions that are similar: execv(3), execvp(3) etc

They are usually fine to use.
Just be careful in the assignment as some of them are explicitly not allowed.

fork() — clone yourself

```c
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- Creates new process by duplicating the calling process.
  - new process is the child, calling process is the parent
- Both child and parent return from fork() call... how to distinguish?
  - in the child, fork() returns 0
  - in the parent, fork() returns the pid of the child
  - if the system call failed, fork() returns -1
- Child inherits copies of parent's address space, open files ...

Example: using fork()

```c
// fork creates 2 identical copies of program
// only return value is different
pid_t pid = fork();
if (pid == -1) {
    perror("fork"); // print why the fork failed
} else if (pid == 0) {
    printf("I am the child because fork() returned %d.\n", pid);
} else {
    printf("I am the parent because fork() returned %d.\n", pid);
}
```

Demo: fork.c, fork2.c

Exercise: How many processes?

How many processes are created when we run this program? What will it print?

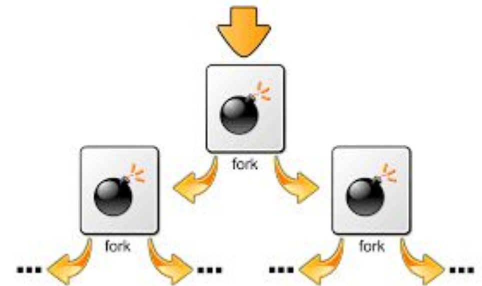
```
int main(void) {
    printf("Hello\n");
    fork();
    fork();
    fork();
    printf("Goodbye\n");
    return 0;
}
```

Demo: fork_ex1.c

Fork dangers, e.g. a fork bomb

WARNING!
DON'T EVEN THINK ABOUT IT!

```c
#include <stdio.h>
#include <unistd.h>
// DO NOT RUN CODE LIKE THIS.
int main(void) {
    // End up with 1024 processes
    for(int i = 0; i < 10; i++) {
        printf("In %d fork returned %d\n", getpid(), fork());
        sleep(1);
    }
    return 0;
}
```



fork_gotcha.c

```
int main(void) {
    printf("about to fork getpid() = %d... ", getpid());
    pid_t fork_return = fork();
    printf("fork() = %d, getpid() = %d...\n",fork_return,
                                                      getpid());
    return 0;
}
```

Demo: fork_var.c
fork_gotcha.c

waitpid() — wait for process to change state

```
pid_t waitpid(pid_t pid, int *wstatus, int options)
```

- `wstatus` is set to hold info about pid.
  - e.g. exit status if pid terminated
  - macros allow precise determination of state change
    - (e.g. WIFEXITED(status), WCOREDUMP(status))
- `options` provide variations in waitpid() behaviour
  - default: wait for child process to terminate
  - WNOHANG: return immediately if no child has exited
  - WCONTINUED: return if a stopped child has been restarted
- For more information, `man 2 waitpid`.

fork_wait.c

exit() — terminate yourself

```
#include <stdlib.h>

void exit(int status);
```

- triggers any functions registered as atexit()
- **flushes stdio buffers; closes open FILE *'s**
- terminates current process
- a SIGCHLD signal is sent to parent
- returns status to parent (via waitpid())
- any child processes are inherited by init (pid 1)

_exit() - terminate yourself without ...

```c
#include <stdlib.h>
void _exit(int status);
```

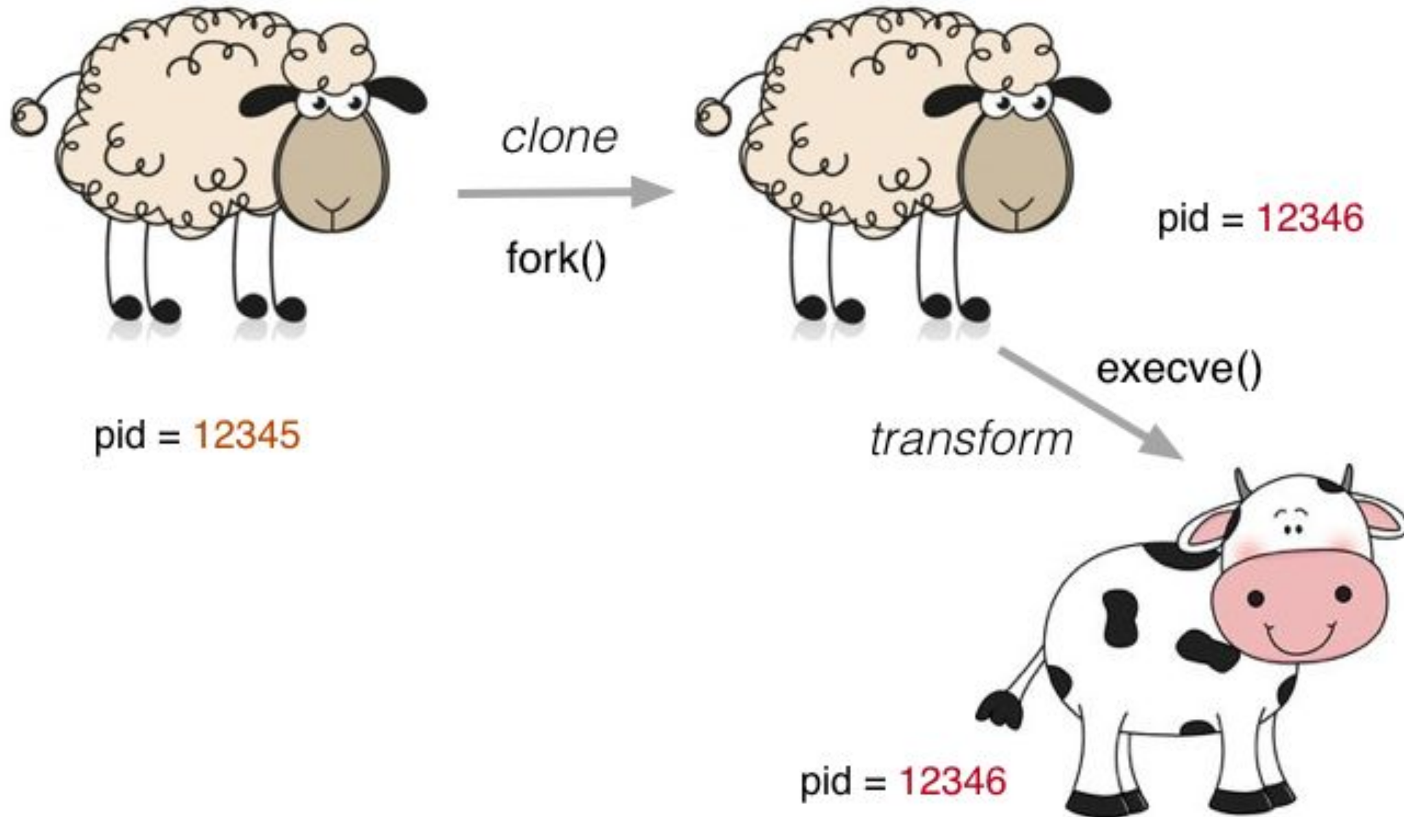
- The same as exit except:
  - Does not trigger functions registered as atexit()
  - **Does not flush stdio buffers**
- Sometimes used by children of fork() when exiting

Aside: Zombie Processes

Aside: Zombie Processes

- When a process terminates, some of its details remain in the process table, and the process is called a **zombie**.
- A **zombie** remains until its parent calls wait() or waitpid() (reaps the process)
  - this can be a problem for long running processes that don't reap their children.
  - **zombie**s that hang around waste system resources.
- Orphan process = a process whose parent has exited
  - when parent exits, orphan assigned PID 1 (init) as its new parent
  - init always accepts notifications of child terminations

Fork and Execve Together!



Example: fork() and exec() to run /bin/date

```
pid_t pid = fork();
if (pid == -1) {
    perror("fork"); // print why fork failed
} else if (pid == 0) { // child
    char *date_argv[] = {"/bin/date", "--utc", NULL};
    execv("/bin/date", date_argv);
    perror("execv"); // print why exec failed
} else { // parent
    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        exit(1);
    }
    printf("/bin/date exit status was %d\n", exit_status);
}
```

Demo: fork_exec.c, fork_exec2.c

system(): convenient but unsafe

```
#include <stdlib.h>
int system(const char *command)
```

- Creates another process
  - runs command via /bin/sh.
  - waits for command to finish and returns exit status
- Don't use in code which handles untrusted input or needs to be reliable!
  - Only use for quick debugging and throwaway code

system() — convenient but risky

- Convenient … but extremely dangerous —
  - very brittle; highly vulnerable to security exploits
  - especially dangerous in code which handles untrusted input!
  - https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=OS+Command+Injection
  - use for quick debugging and throw-away programs only

```
// run date --utc to print current UTC
int exit_status = system("/bin/date --utc");
printf("/bin/date exit status was %d\n", exit_status);
return 0;
```

Demo: system.c

Making Processes

- Old-fashioned way **fork()** then **execve()**
  - **fork()** duplicates the current process (parent+child)
  - **execve()** “overwrites” the current process (run by child)
- Scary unsafe way **system()**
- New, standard way **posix_spawn()**

posix_spawn() — Run a new process

```
#include <spawn.h>
int posix_spawn(
    pid_t *pid, const char *path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *attrp,
    char *const argv[], char *const envp[]);
```

- **pid**: returns process id of new program
- **path**: path to the program to run
- file_actions: specifies file actions to be performed before running program (advanced)
  - can be used to redirect stdin, stdout to file or pipe

posix_spawn() — Run a new process

```
#include <spawn.h>
int posix_spawn(
    pid_t *pid, const char *path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *attrp,
    char *const argv[], char *const envp[]);
```

- attrp: specifies attributes for new process (not covered in COMP1521)
- **argv**: arguments to pass to new program
- **envp**: environment to pass to new program
- can also use **posix_spawnp** which searches PATH

Example: posix_spawn() to run /bin/date

```
pid_t pid;
extern char **environ;
char *date_argv[] = {"/bin/date", "--utc", NULL};
// spawn "/bin/date" as a separate process
int ret = posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv, environ);
if (ret != 0) {
    errno = ret; //posix_spawn returns error code, does not set errno
    perror("spawn"); exit(1);
}
// wait for spawned processes to finish
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); exit(1);
}
printf("/bin/date exit status was %d\n", exit_status);
```

Demo: spawn.c
: spawnp.c

Setting environment var for child process

```
// set environment variable STATUS
setenv("STATUS", "great", 1);
char *getenv_argv[] = {"./get_status", NULL};
pid_t pid;
extern char **environ;
int ret = posix_spawn(&pid, "./get_status", NULL, NULL,
             getenv_argv, environ);
if (ret != 0) {
    errno = ret; perror("spawn"); return 1;
}
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); exit(1);
```

Demo: set_status.c

Change behaviour with an environment var

```
pid_t pid;
char *date_argv[] = { "/bin/date", NULL };
char *date_environment[] = { "TZ=Australia/Perth", NULL };
// print time in Perth
int ret = posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv,
date_environment);
if (ret != 0) {
    errno = ret; perror("spawn"); return 1;
}
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); return 1;
}
printf("/bin/date exit status was %d\n", exit_status);
```

Demo: spawn_env.c

Exercise: posix_spawn_cat.c

Use posix_spawn to create

- a child that runs the cat command on a file given as a command line argument

Advanced: File Redirection

Aside: Duplicating File Descriptors

Libc wrappers for system calls **dup** and **dup2**

```
int dup(int oldfd);
```

- **dup**licates file descriptor **oldfd** and returns the new file descriptor number
- the next available fd number is chosen
- returns -1 on failure

```
int dup2(int oldfd,int newfd);
```

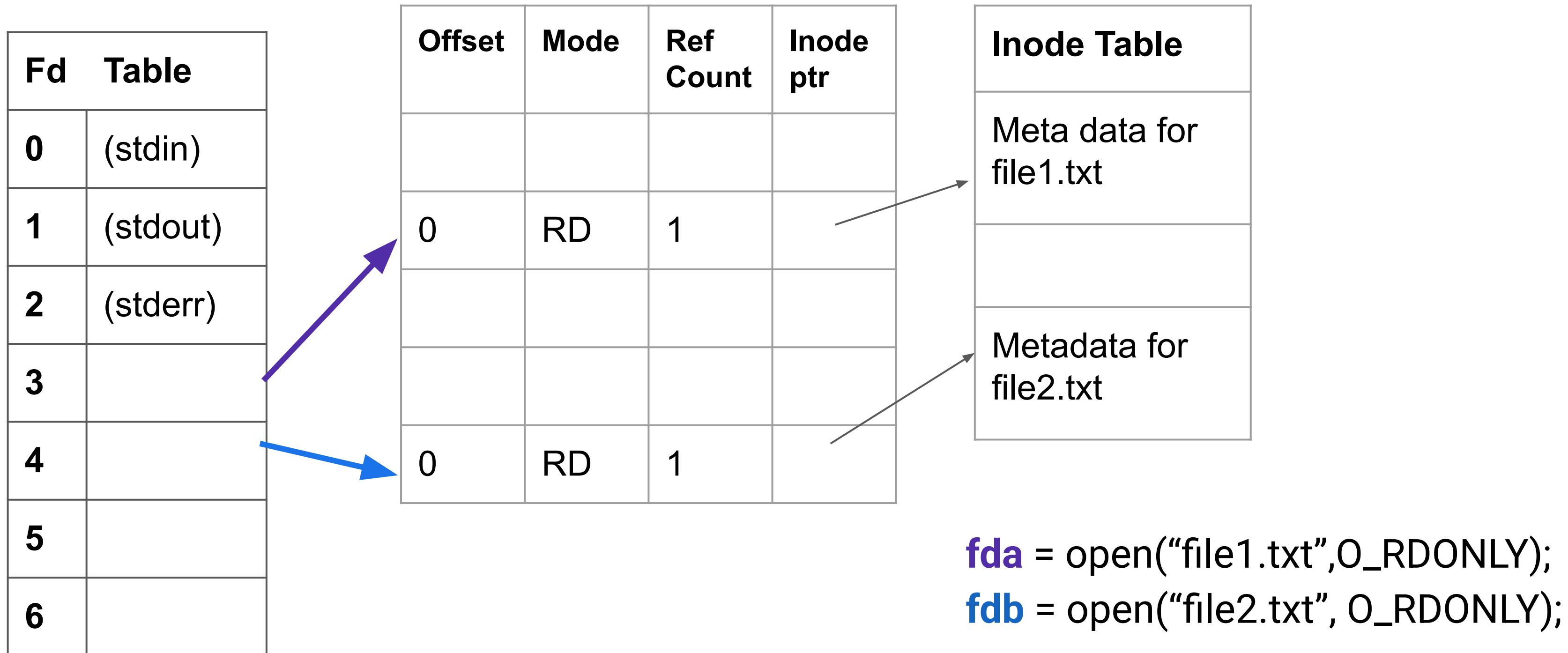
- The same as dup but you can specify the new fd number
- If the newfd is already open, dup2 will close it

Aside: Duplicating File Descriptors

When file descriptors are duplicated:

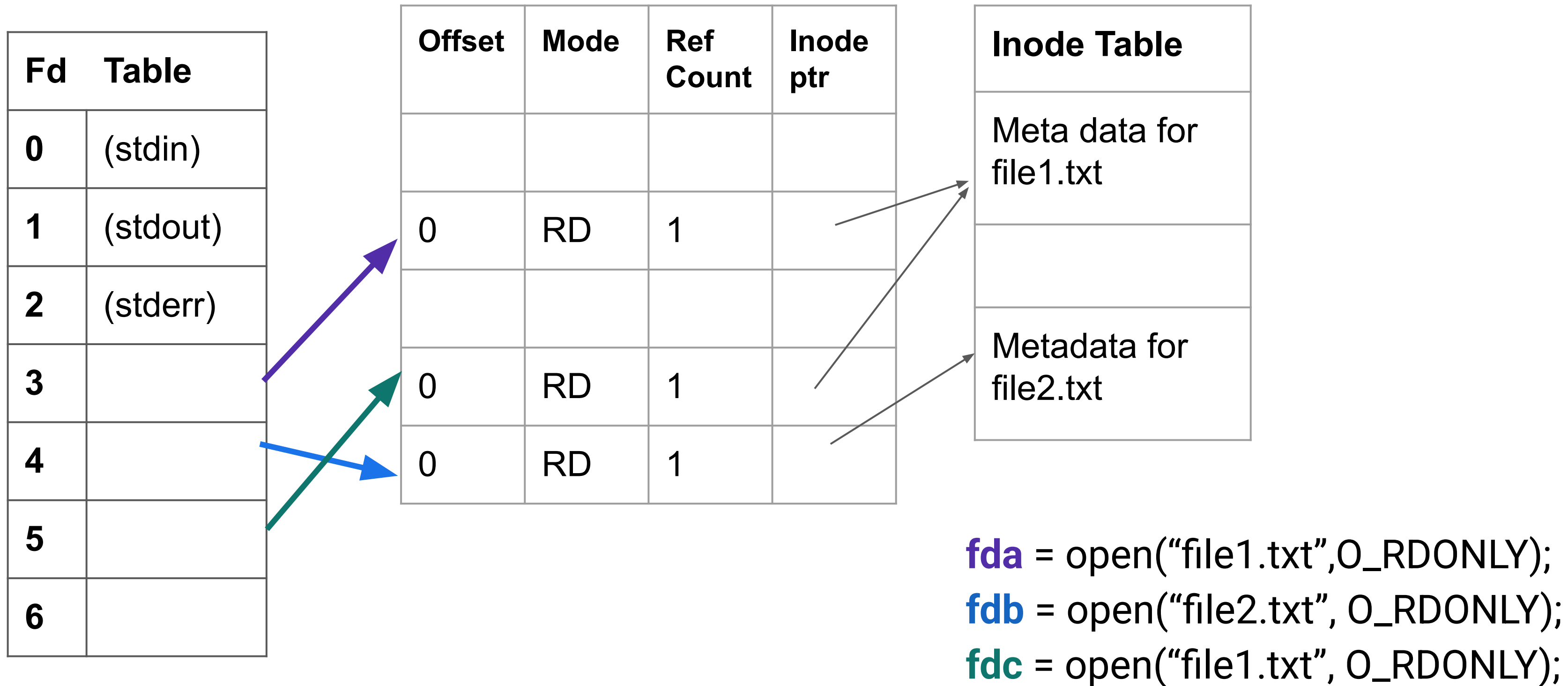
- The **current position** is shared between **oldfd** and **newfd**
  - read/writes to **oldfd** also update position in **newfd**
  - read/writes to **newfd** also update position of **oldfd**
- The system-wide open file table maintains a reference count so it knows when it can free the entry

dup2

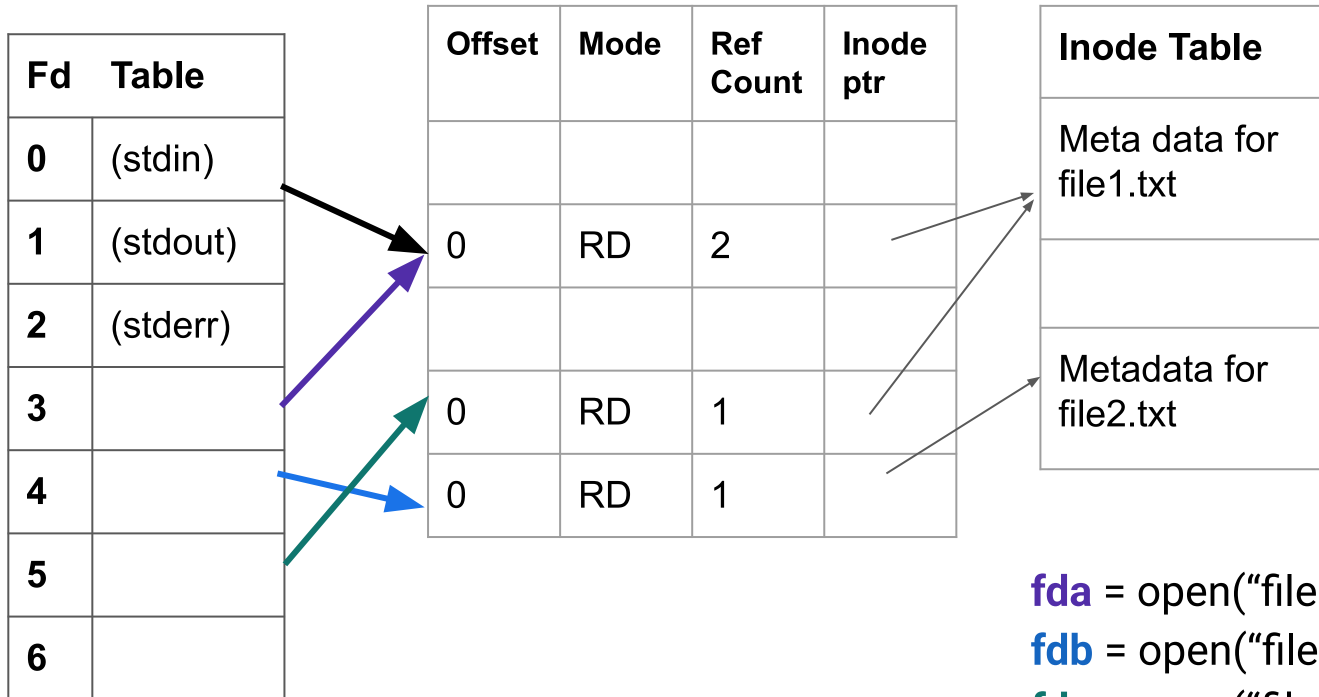



```
fda = open("file1.txt",O_RDONLY);
fdb = open("file2.txt", O_RDONLY);
```

dup2

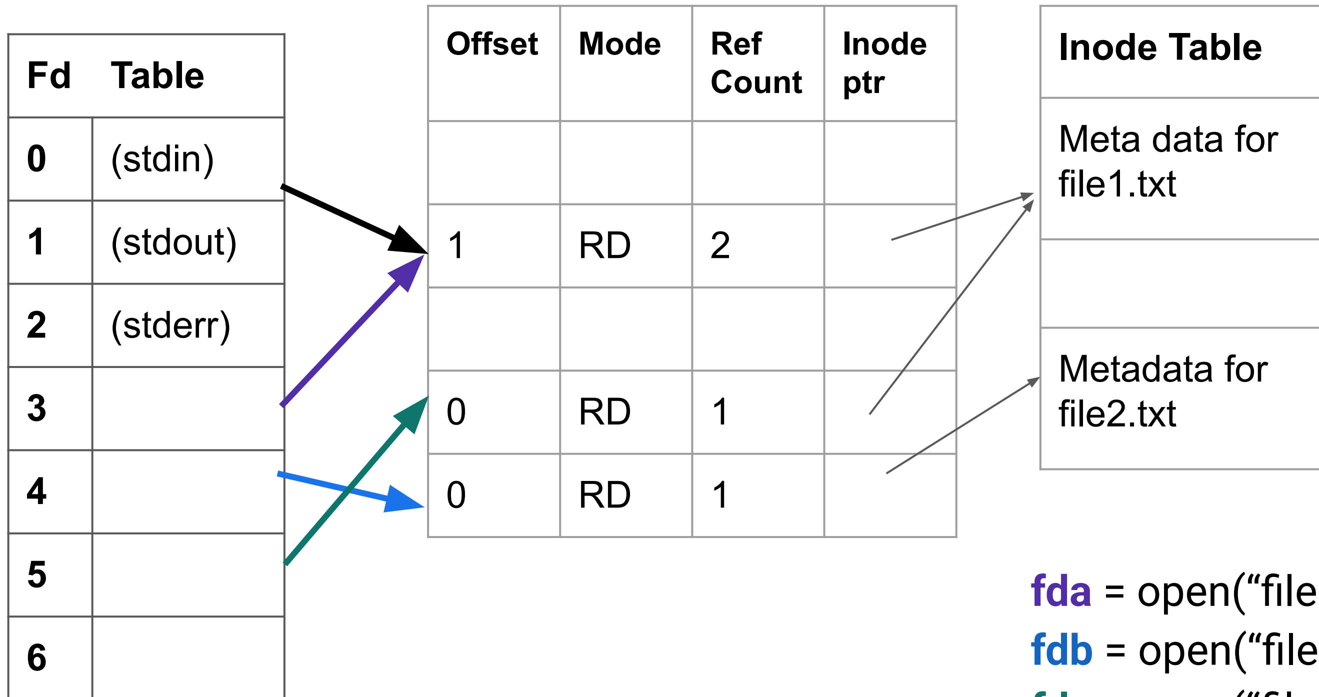


dup2



fda = open(“file1.txt”,O_RDONLY);
fdb = open(“file2.txt”, O_RDONLY);
fdc = open(“file1.txt”, O_RDONLY);
dup2(fd1, **0**);

dup2




```
fda = open("file1.txt",O_RDONLY);
fdb = open("file2.txt", O_RDONLY);
fdc = open("file1.txt", O_RDONLY);
dup2(fd1, 0);
read(0, &c, 1);
```

Advanced: File Redirection

```
//Like running tr [a-z] [A-Z] < file.txt
int fd = open("file.txt", O_RDONLY);
dup2(fd, 0);
close(fd);
char *tr_argv[] = {"/usr/bin/tr","[a-z]","[A-Z]", NULL};
execve("/usr/bin/tr", tr_argv, environ);
```

```
//Like running echo My lecturer is great > message.txt
int fd = open("message.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
dup2(fd, 1);
close(fd);
char *echo_argv[] = {"/usr/bin/echo","My","lecturer","is","great!",NULL};
execve("/usr/bin/echo", echo_argv, environ);
```

Demos: redirect_stdin.c, redirect_stdout.c,

Advanced: Pipes

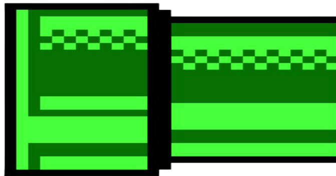
Inter-Process Communication(IPC)

- Processes have their own address spaces
- IPC allows processes to share data with each other.
- **Pipes** are just one form of IPC.
- There are others you may learn about in other courses.

pipe() — stream bytes between processes

Send output of one process as input to another

Process A
WRITES
to the
pipe



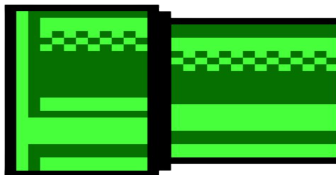
Process B
READS
from the
pipe

pipe() — stream bytes between processes

Demo: on the command line:

```
seq 1 10 | wc
```

Process A
WRITES
to the
pipe



Process B
READS
from the
pipe

pipe() — stream bytes between processes

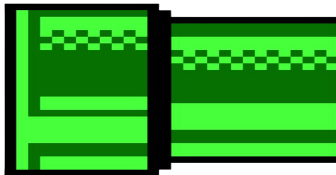
```
#include <unistd.h>

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

- Pipes: unidirectional byte streams provided by operating system
  - **pipefd[0]**: set to file descriptor of read end of pipe
  - **pipefd[1]**: set to file descriptor of write end of pipe
  - bytes written to **pipefd[1]** will be read from **pipefd[0]**
- Child processes (by default) inherit file descriptors including pipes

pipe() — stream bytes between processes

```
Process A
 WRITES
   to
```
pipefd[1]



```
Process B
  READS
  from
```
pipefd[0]

Closing pipes

- Parent can send/receive bytes (not both) to child via pipe
  - parent and child should both close unused pipe file descriptors
  - e.g if bytes being written (sent) parent to child
    - parent should close read end **pipefd[0]**
    - child should close write end **pipefd[1]**
- Pipe file descriptors can be used with stdio via **fdopen()**

Demo: pipe_fork1.c, pipe_fork2.c

popen() — convenient way to set up pipe

```c
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- runs command via /bin/sh
- if type is “w” pipe to stdin of command created
- if type is “r” pipe from stdout of command created
- FILE * stream returned - get then use fgetc/fputc etc
  - NULL returned if error
- close stream with pclose (not fclose)
  - pclose waits for command and returns exit status

popen() — unsafe

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- convenient but brittle
- vulnerable to command injection (same as system())
- try to avoid use except in debugging and throw-away programs

Example: process output with popen()

```
// popen passes string to a shell for evaluation
// brittle and highly-vulnerable to security exploits
// popen is suitable for quick debugging and throw-away programs only
FILE *p = popen("/bin/date --utc", "r");
if (p == NULL) {
    perror(""); return 1;
}
char line[256];
if (fgets(line, sizeof line, p) == NULL) {
    fprintf(stderr, "no output from date\n"); return 1;
}
printf("output captured from /bin/date was: '%s'\n", line);
pclose(p); // returns command exit status
```

Demo: read_popen.c

Example: input to a process with popen()

```c
int main(void) {
    // popen passes command to a shell for evaluation
    // brittle and highly-vulnerable to security exploits
    //
    // tr a-z A-Z - passes stdin to stdout converting lower case to upper case
    FILE *p = popen("tr a-z A-Z", "w");
    if (p == NULL) {
        perror("");
        return 1;
    }
    fprintf(p, "hello, i am a COMP1521 aficionado\n");
    pclose(p); // returns command exit status
    return 0;
}
```

Demo: write_popen.c

posix_spawn and pipes (advanced topic)

```
int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_init(posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *file_actions,
                                       int fildes);
int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *file_actions, int fildes,
                                      int newfildes);
```

- functions to combine file ops with posix_spawn process creation
- awkward to understand and use but **robust**

Example: capturing output from a process: spawn_read_pipe.c
Example: sending input to a process: spawn_write_pipe.c

What we learnt Today

- Processes
  - execve, fork, waitpid
  - posix_spawn
  - exit, _exit
- Redirecting Files
  - dup, dup2 (advanced)
- Pipes
  - posix_spawn (advanced)

```
system("ls -l");
```

```
pid_t pid;
char *argv[] = {"/bin/ls", "-l", cmd, NULL};
posix_spawn(&pid, "/bin/ls", NULL, NULL, argv, NULL);
```

Next Lecture

- Concurrency and Parallelism
    - Threads
    - Mutexes
    - Atomics

Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/F5Nh1svm2e

Reach Out

Content Related Questions:

[Forum]

Admin related Questions email:

cs1521@cse.unsw.edu.au

Student Support | I Need Help With...

My Feelings and Mental Health
Managing Low Mood, Unusual Feelings & Depression

Mental Health Connect — student.unsw.edu.au/**counselling** Telehealth

In Australia Call Afterhours UNSW Mental Health Support Line — 1300 787 026 5pm-9am

Mind HUB — student.unsw.edu.au/**mind-hub** Online Self-Help Resources

Outside Australia Afterhours 24-hour Medibank Hotline — +61 (2) 8905 0307

Uni and Life Pressures
Stress, Financial, Visas, Accommodation & More

Student Support Indigenous Student Support — student.unsw.edu.au/**advisors**

Reporting Sexual Assault/Harassment

Equity Diversity and Inclusion (EDI) — edi.unsw.edu.au/**sexual-misconduct**

Educational Adjustments
To Manage my Studies and Disability / Health Condition

Equitable Learning Service (ELS) — student.unsw.edu.au/**els**

Academic and Study Skills

Academic Language Skills — student.unsw.edu.au/**skills**

Special Consideration
Because Life Impacts our Studies and Exams

Special Consideration — student.unsw.edu.au/**special-consideration**