TECHNISCHE
UNIVERSITÄT
WIEN

Exercise 2: Processes & Pipes

191.002 Operating Systems VU
2024W

Florian Mihola, Axel Brunnbauer, Luca Di Stefano, Peter Puschner

2024-11-12

Processes

Why should we create processes?

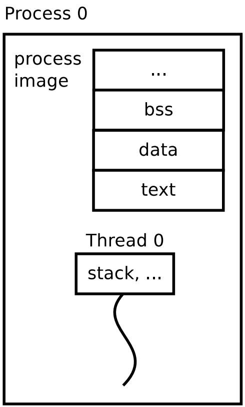
- Divide up a task
  - Simpler application design
  - Greater concurrency

Example

A server listens to client requests. The server process starts a new process to handle each request and continues to listen for further connections.
The server can handle several client requests simultaneously.

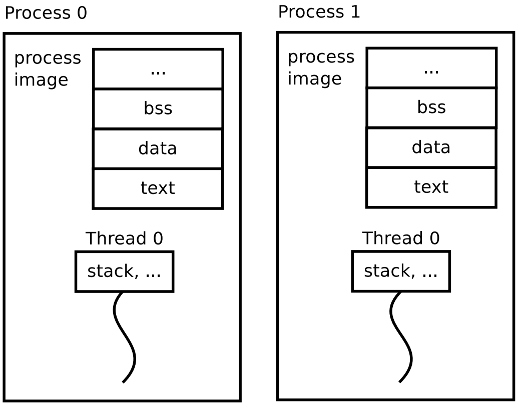
Process vs. Thread

fork(2) vs. pthreads(7)



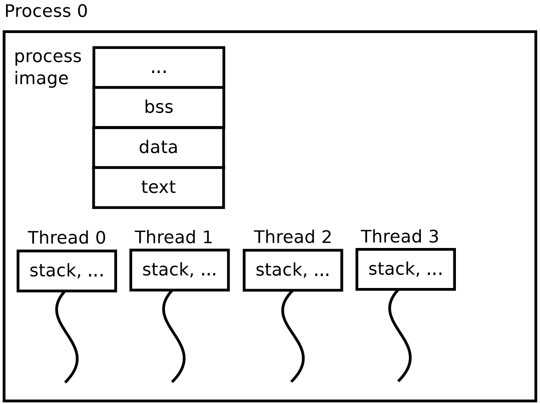
Process vs. Thread

fork(2) vs. pthreads(7)



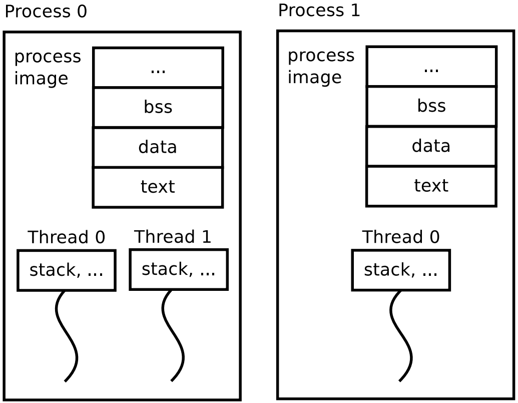
Process vs. Thread

fork(2) vs. pthreads(7)



Process vs. Thread

fork(2) vs. pthreads(7)



Process Hierarchy

- Every process has a parent process
- Exception: init process (`init`, `systemd`)
- Every process has a unique ID (`pid_t`)
- Show process hierarchy: `pstree(1)`

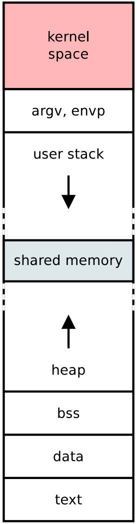
```
systemd-+-ModemManager---2*[{ModemManager}]
        |-NetworkManager-+-dhclient
        |                `-2*[{NetworkManager}]
        |-abrt-dbus---{abrt-dbus}
        |-2*[abrt-watch-log]
        |-abrtd
        |-acpid
        |-agetty
        |-alsactl
        |-atd
        |-auditd-+-audispd-+-sedispatch
        |        |         `-{audispd}
        |        `-{auditd}
        |-automount---7*[{automount}]
        |-avahi-daemon---avahi-daemon
        |-chronyd
        |-colord---2*[{colord}]
        |-crond
        |-cupsd
        |-dbus-daemon
        |-dnsmasq---dnsmasq
        |-firewalld---{firewalld}
        .
        .
```

Properties of a Process in Linux

State	Running, waiting, …
Scheduling	Priority, CPU time, …
Identification	PID, owner, group, …
Memory Management	Pointer to MMU information
Signals	Mask, pending
Process Relations	Parents, siblings

- Show process info: `cat /proc/<pid>/status`
- See `struct task_struct` in `sched.h`

virtual memory of a process



kernel space

argv, envp

user stack

shared memory

heap

bss

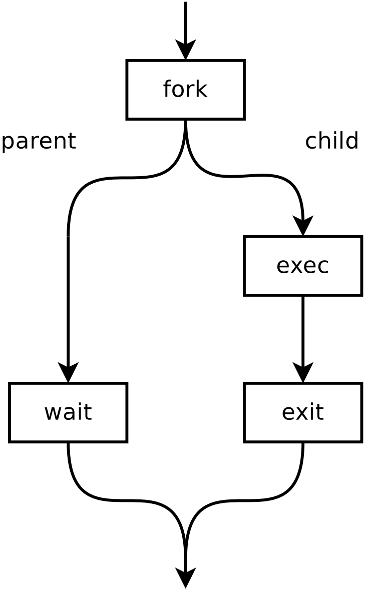
data

text

Interface

fork / exec / exit / wait

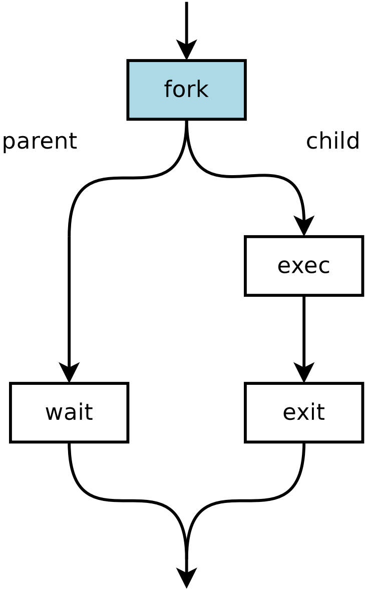
- `fork(2)` – creates a process (copies the process image)
- `exec(3)` – loads a program (replaces the process image of a process with a new one)
- `exit(3)` – exits a process
- `wait(2)` – awaits the exit of child processes



Process Creation

fork

- Creates a new process
- New process is an identical copy of the calling process – except PID, pending signals, ...
- Calling process is the parent of the created process, the child – processes are related
- Both processes run parallel and execute the same program (from the `fork` call on)



Process Creation

fork

- Create the process

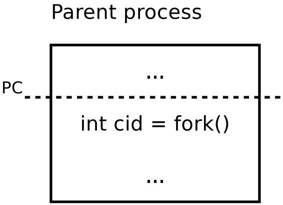
```c
#include <unistd.h>

pid_t fork(void);
```

- Distinguish between parent and child via return value of `fork`

-1	On error
0	In the child process
>0	In the parent process

Process Creation

Before `fork()`



Parent process

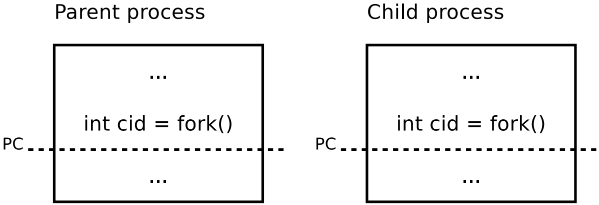
PC

...

int cid = fork()

...

After `fork()`



Parent process

PC

...

int cid = fork()

...

Child process

PC

...

int cid = fork()

...

Process Creation

Example

```c
pid_t pid = fork();

switch (pid) {
  case -1:
    fprintf(stderr, "Cannot fork!\n");
    exit(EXIT_FAILURE);

  case 0:
    [...] // child tasks
    break;
  default:
    [...] // parent tasks
    break;
}
```

Process Creation

Child

Child inherits from parent:

- Opened files (common access!)
- File buffers
- Signal handling
- Current values of variables

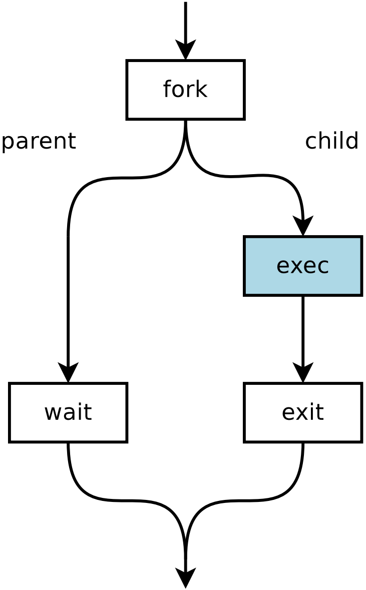
But:

- Variables are local to process (no influence)
- Signal handling can be re-configured
- Communication (IPC) via pipes, sockets, shared memory, ...

Program Execution

exec

- Load a new program into a process's memory
- Executes another program
- In the same process (PID remains the same)



Program Execution

exec Family[1]

```c
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);

int execle(const char *path, const char *arg, ...,
           char *const envp[]);

int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

[1] Frontend of `execve(2)`

Program Execution

exec Family

- `execl⬚` – variable number of arguments
- `execv⬚` – arguments via array
- `exec⬚p` – searching the environment variable `$PATH` for the program specified
- `exec⬚e` – environment[2] can be changed
- `fexecve` – accepts file descriptor (instead of path)

Note Argument Passing!

- 1st argument is the program's name (`argv[0]`)!
- Last argument must be a `NULL` pointer!

[2] FYI: environ(7)

Program Execution

Example: execv(), execvp()

```
#include <unistd.h>


char *cmd[] = { "ls", "-l", (char *) 0 };

execv("/bin/ls", cmd);
// or:
// execvp("ls", cmd);

fprintf(stderr, "Cannot exec!\n");
exit(EXIT_FAILURE);
```

Program Execution

Example: execl(), execlp()

```c
#include <unistd.h>

execl("/bin/ls", "ls", "-l", NULL);
// or:
// execlp("ls", "ls", "-l", NULL);
fprintf(stderr, "Cannot exec!\n");
exit(EXIT_FAILURE);
```

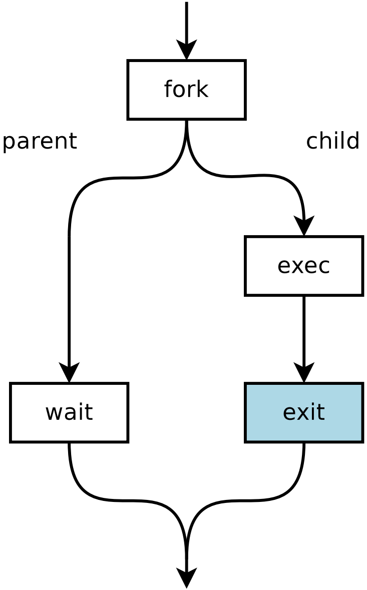
Attention - this will not work:

```c
execl("/bin/ls", "ls -l", NULL);

int a = 1;
execl("myprog", "myprog", "-a", a, NULL);
    // e.g., use a char-buffer and snprintf(3)
```

Process Termination

exit

- Terminates a process (normally)
- Termination status can be read by parents
- Actions performed by `exit()`
  - Flush and close stdio stream buffers
  - Close all open files
  - Delete temporary files (created by `tmpfile(3)`)
  - Call exit handlers (`atexit(3)`)



Process Termination

exit

- Terminate a process normally

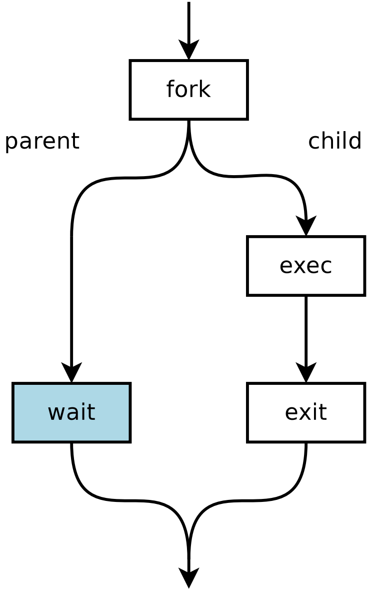
```c
#include <stdlib.h>

void exit(int status);
```

- Status: 8 bit (0-255)
- By convention
  - `exit(EXIT_SUCCESS)` – process completed successfully
  - `exit(EXIT_FAILURE)` – error occurred
- More return values
  - BSD: sysexits.h
  - http://tldp.org/LDP/abs/html/exitcodes.html

Waiting on a Child Process

wait

- Wait until a child process terminates
- Returns the PID and status of the terminated child



Waiting on a Child Process

wait

- Wait for a child to terminate

```
#include <sys/wait.h>

pid_t wait(int *status);
```

- `wait()` blocks[3] until a child terminates or on error
- Return value
  - PID of the terminated child
  - -1 on error (→ `errno`, e.g., `ECHILD`)
- Status includes exit value and signal information
  - `WIFEXITED(status)`, `WEXITSTATUS(status)`
  - `WIFSIGNALED(status)`, `WTERMSIG(status)`
  - See `wait(2)`

[3] ≠ busy waiting

Waiting on a Child Process

Zombies and Orphans

- UNIX: Terminated processes remain in the process table
- No more space in process table → no new process can be started!
- After `wait()` the child process is removed from the process table

Zombie Child terminates, but parent didn't call `wait` yet

- State of the child is set to "zombie"
- Child remains in process table until parent calls `wait`

Orphan Parent terminates before child

- Child becomes an orphan and is inherited to the init process
- When an orphan terminates, the init process removes the entry in the process table

Waiting on a Child Process

Example

```c
#include <sys/wait.h>

int status;
pid_t child_pid, pid;
...
while ((pid = wait(&status)) != child_pid) {
  if (errno == EINTR) continue;
  if (pid == -1) {
    fprintf(stderr, "Cannot wait!\n");
    exit(EXIT_FAILURE);
  }
}

if (WEXITSTATUS(status) == EXIT_SUCCESS) {
  ...
```

Waiting on a Child Process

waitpid

- Wait on a specific child process

```c
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

- Examples

```c
waitpid(cid, &status, 0);
      // waits on a child process with PID ’cid’

waitpid(-1, &status, 0);
      // equivalent to wait

waitpid(-1, &status, WNOHANG);
      // does not block
```

Notification

on Termination of a Child

If parent should not block

- Synchronous
  - `waitpid(-1, &status, WNOHANG)`
  - Returns exit status when a child terminates
  - Repeating calls → polling
- Asynchronous
  - Signal `SIGCHLD` is sent to the parent process whenever one of its child processes terminates
  - Catch by installing a signal handler (`sigaction`)
  - Call `wait` in the signal handler

Pitfalls

```c
int main(int argc, char **argv)
{
    fprintf(stdout, "Hello");

    (void) fork();
    return 0;
}
```

Output: `"HelloHello"`

Why?

Pitfalls

```c
int main(int argc, char **argv)
{
    fprintf(stdout, "Hello");
    fflush(stdout);
    (void) fork();
    return 0;
}
```

Output: `"Hello"`

→ for all opened streams

Debugging

gdb

- Before `fork` is executed:
  `set follow-fork-mode [child|parent]`

Example

```
  $ gdb -tui ./forktest
  (gdb) break main
  (gdb) set follow-fork-mode child
  (gdb) run
  (gdb) next
  (gdb) :
  (gdb) continue
  (gdb) quit
```

Inter-Process Communication

Recall

So far:

- Signals (e.g., to synchronise between parent and child)
  → see Development in C I

New:

- Pipes

Other lectures:

- Shared Memory
- Sockets

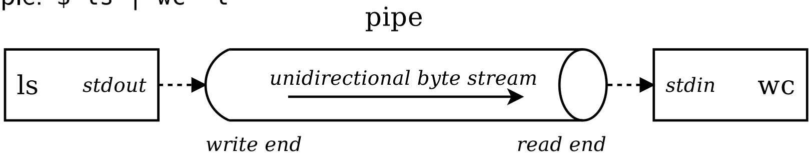
Pipes

(Unnamed) Pipe

= unidirectional data channel

= enables communication between related processes

- Example: `$ ls | wc -l`



- Access to read and write end of the pipe via file descriptors
- Pipe is an unidirectional byte stream
- Buffered
- Implicit synchronisation

Pipes

Create

- Create a pipe

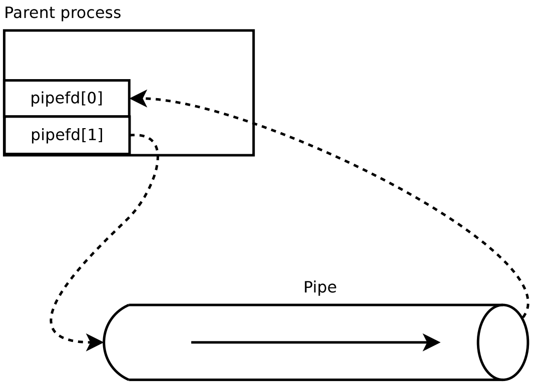
```c
#include <unistd.h>

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

- File descriptors of read and write end are returned in specified integer array `pipefd`
  - `pipefd[0]` – read end
  - `pipefd[1]` – write end
- Close unused ends
- Use read/write end via stream-IO (`fdopen`, etc.)
- A child process inherits the pipe → common access

Unnamed Pipes

Illustration

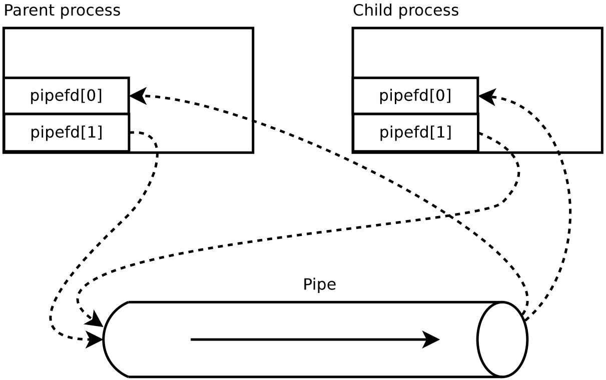
`pipe;`



Unnamed Pipes

Illustration

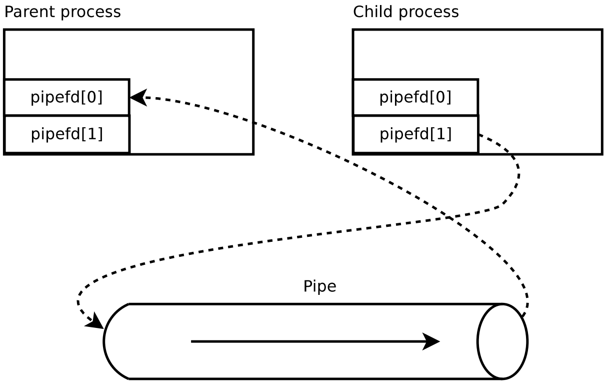
```
pipe;
fork;
```



Unnamed Pipes

Illustration

`pipe`;
`fork`;
close unused ends;



Unnamed Pipes

Implicit Synchronisation

- `read` blocks on empty pipe
- `write` blocks on full pipe

- `read` indicates end-of-file if all write ends are closed (return value 0)
- `write` creates signal `SIGPIPE` if all read ends are closed (if signal ignored/handled: `write` fails with errno `EPIPE`)

Therefore...

... close unused ends, to get this behaviour (end-of-file and `SIGPIPE`/`EPIPE`).

Besides, the kernel removes pipes with all ends closed.

Unnamed Pipes

What about named pipes?

- Unnamed pipes
  - `|`
  - `pipe(2)`
- Named pipes
  - `mkfifo(1)`, `mknod(2)`
  - Usage similar to files.
  - (Will not be dealt with any further throughout this course.)

Redirection of stdin/stdout

Why?

- Main application: pipes
- Example: shell redirection of `stdin` and `stdout`

Scenario:

- A process may be forked or not
  → uses standard IO
- A parent process forks and executes another program
- Parent usually wants to use the child's output
  → redirect stdin (file descriptor 0, `STDIN_FILENO`) and/or stdout (file descriptor 1, `STDOUT_FILENO`) in new process

Redirection of stdin/stdout

Approach

- Close file descriptors for standard I/O (`stdin`, `stdout`)
- Duplicate opened file descriptor (e.g., a pipe's end) to the closed one

```c
#include <unistd.h>

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

- Close duplicated file descriptor

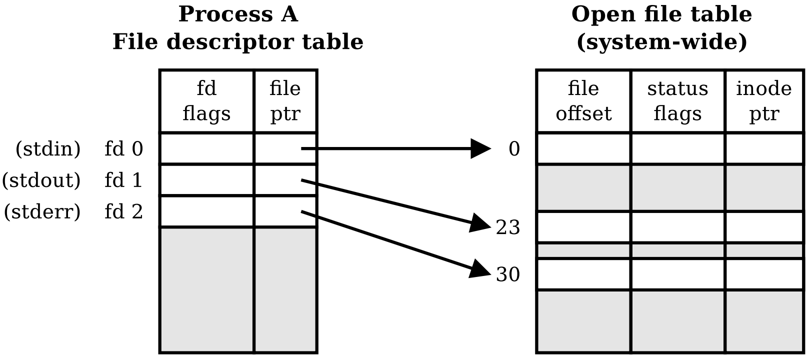
Redirection of stdin/stdout

dup / dup2

- `dup(oldfd)` duplicates file descriptor `oldfd`
  - New file descriptor uses smallest unused ID
    = entry in file descriptor table
  - Duplicated file descriptor points to the same open file description (equal file offset, status flags) → see `open(2)`
- `dup2(oldfd, newfd)` duplicates `oldfd`
  - New file descriptor uses ID `newfd`
  - (Implicitly) closes the file descriptor `newfd` (if necessary)
  - `newfd` points to the same open file description like `oldfd`

Redirection of stdin/stdout

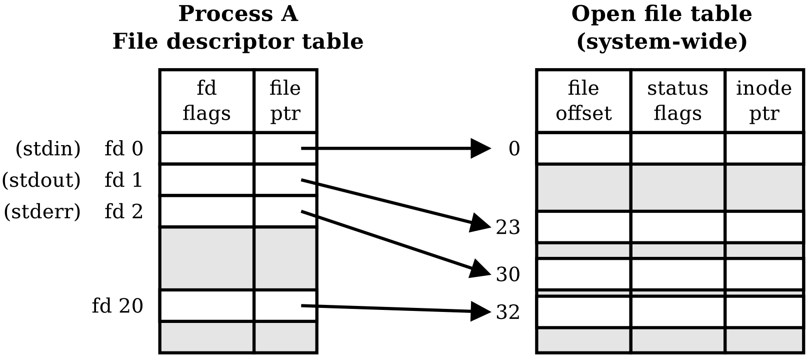
Example: redirect stdout to opened file



Redirection of stdin/stdout

Example: redirect stdout to opened file

open file;



Process A
File descriptor table

fd flags

file ptr

(stdin) fd 0

(stdout) fd 1

(stderr) fd 2

fd 20

Open file table
(system-wide)

file offset

status flags

inode ptr

0

23

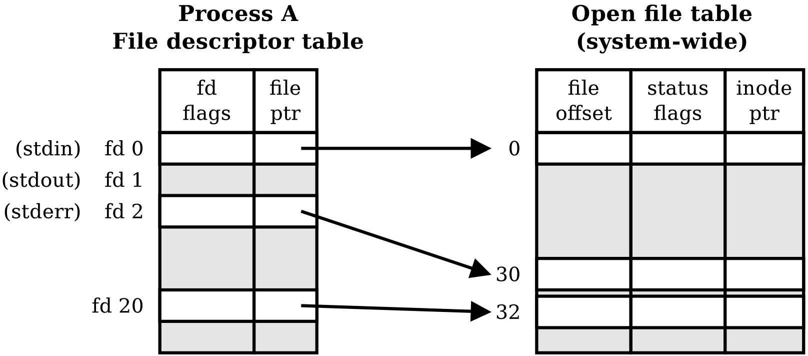
30

32

Redirection of stdin/stdout

Example: redirect stdout to opened file

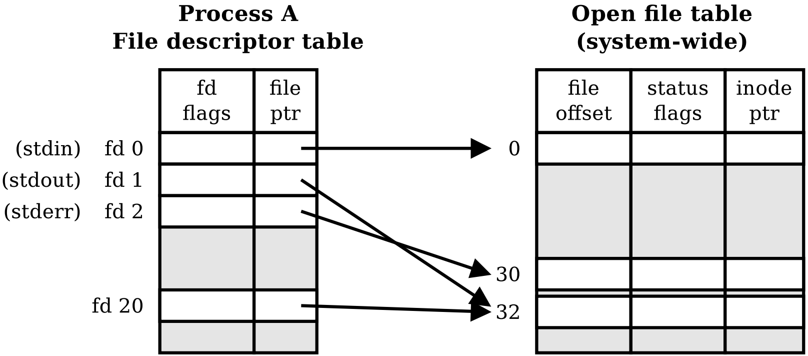
open file; close stdout;



Redirection of stdin/stdout

Example: redirect stdout to opened file

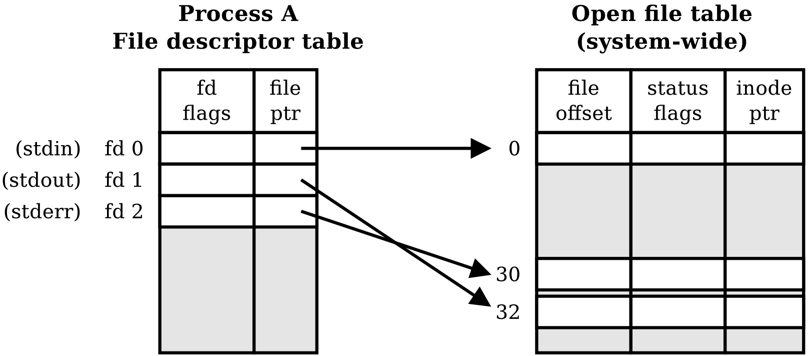
open file; close stdout; dup;



Redirection of stdin/stdout

Example: redirect stdout to opened file

open file; close stdout; dup; close file;



Redirection of stdin/stdout

Example: redirect stdout to log.txt

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

int fd;

// TODO error handling!

fd = open("log.txt", O_WRONLY | O_CREAT);

dup2(fd,              // old descriptor
     STDOUT_FILENO);  // new descriptor

close(fd);

execlp("ls", "ls", NULL);
```

Redirection of stdin/stdout

Example: redirect stdin to pipe

```c
// TODO error handling!
int pipefd[2];
pipe(pipefd);           // create pipe

pid_t pid = fork();
switch(pid) {
  [...]
  case 0: // child counting lines from parent
    close(pipefd[1]);   // close unused write end

    dup2(pipefd[0],     // old descriptor - read end
         STDIN_FILENO); // new descriptor
    close(pipefd[0]);

    execlp("wc", "wc", "-l", NULL);
    // should not reach this line
```

Pitfalls

- Pipes are unidirectional
- Bidirectional: two pipes, but ...
  - Erroneous synchronisation (deadlock, e.g., both processes `read` from empty pipe)
- Synchronisation & Buffer
  - Use `fflush()`
  - Configure buffer (`setbuf(3)`, `setvbuf(3)`)

Pipe & Fork

- “Yes, a programmer’s strength flows from fork(). But beware of the dark side.”
- fork is no ordinary function

```
int pipefd[2];
pipe(pipefd);       // 1. create pipe
pid_t pid = fork(); // 2. fork
// two processes
// one pipe
```

```
pid_t pid = fork(); // 1. fork
// two processes
int pipefd[2];
pipe(pipefd);       // 2. create pipeS!
// two pipes!
```

Tips for the Exercise

- Try to parallelize the functionality of your program (as much as possible)

Example

DO NOT: The parent first reads all input from a file to an array. It then sends the data within one burst to the child. The child processes the data and outputs the result.

INSTEAD DO: The parent reads line-by-line from a file. Each line is sent to the client immediately. Reading and processing of the lines happens in parallel.

Tips for the Exercise

- Communicate over pipes (do not exploit inherited memory areas)

Example

DO NOT: The parent reads a file and saves its content into an array and forks a child. The child processes the data from the array.
INSTEAD DO: The parent communicates the data from the file over a pipe.

- However, you may pass options/flags/settings to the child (process). For example, use inherited variable `argv` to set arguments when using `exec`.

Material

- Michael Kerrisk: A Linux and UNIX System Programming Handbook, No Starch Press, 2010.
- man pages: fork(2), exec(3), execve(2), exit(3), wait(3), pipe(2), dup(2)
- gdb - Debugging Forks:
  `https://sourceware.org/gdb/onlinedocs/gdb/Forks.html`