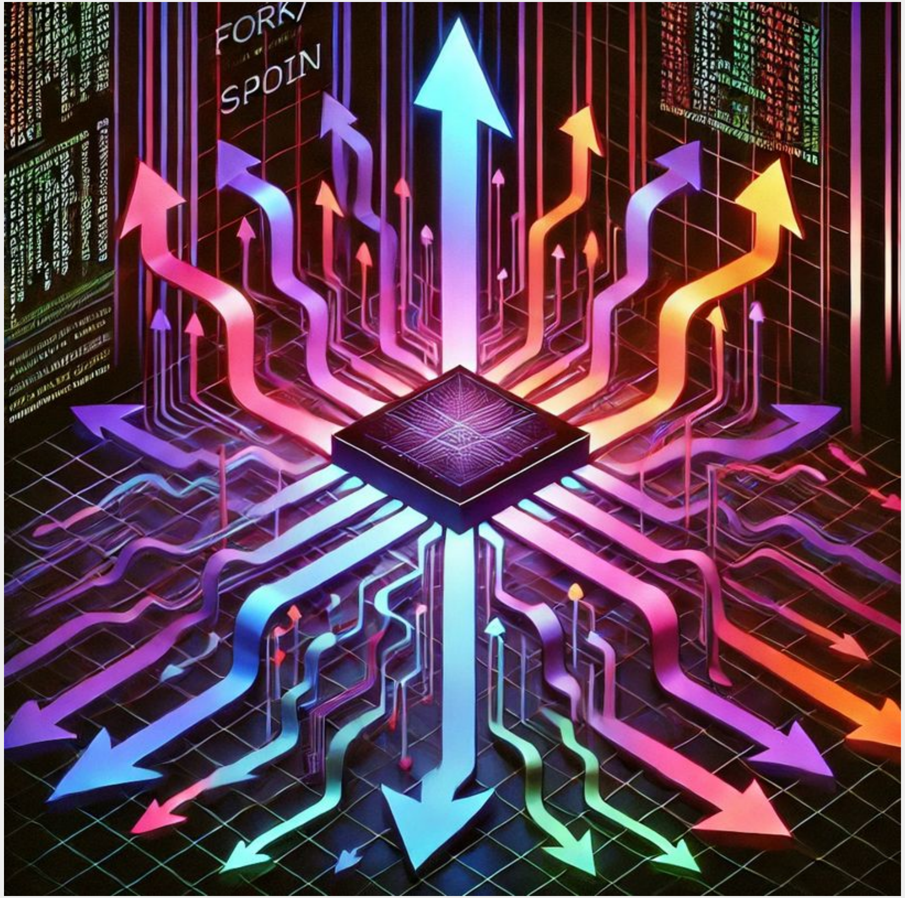
UNSW SYDNEY

COMP1521 24T2 Lec12

Processes



2024
Hammond Pearce
Mostly from Andrew’s slides

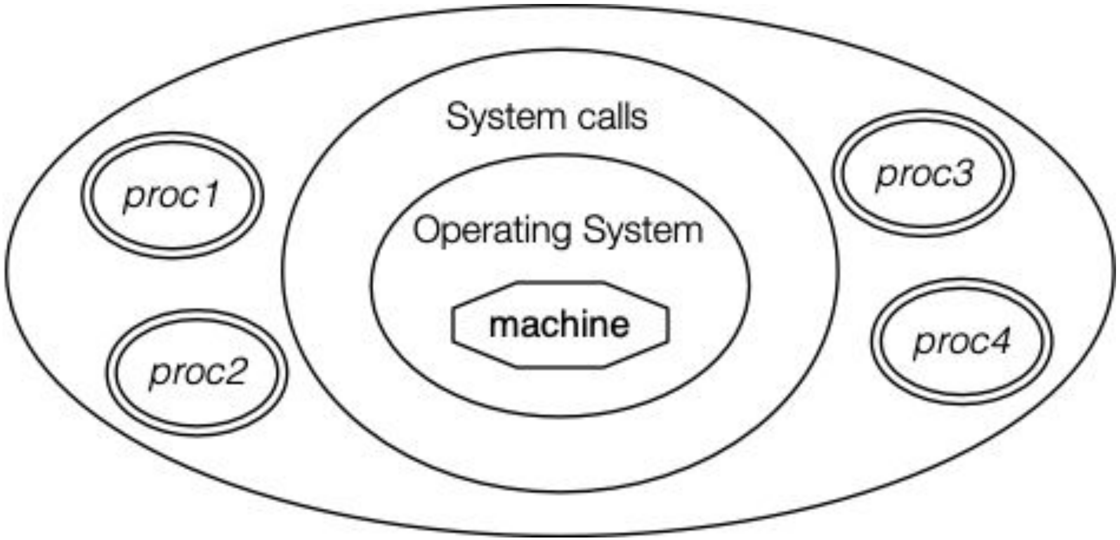
Recap Exercise

Question 1: What are the main differences between UTF-8 and UTF-32?

Question 2: Why was ASCII limited to just western characters?

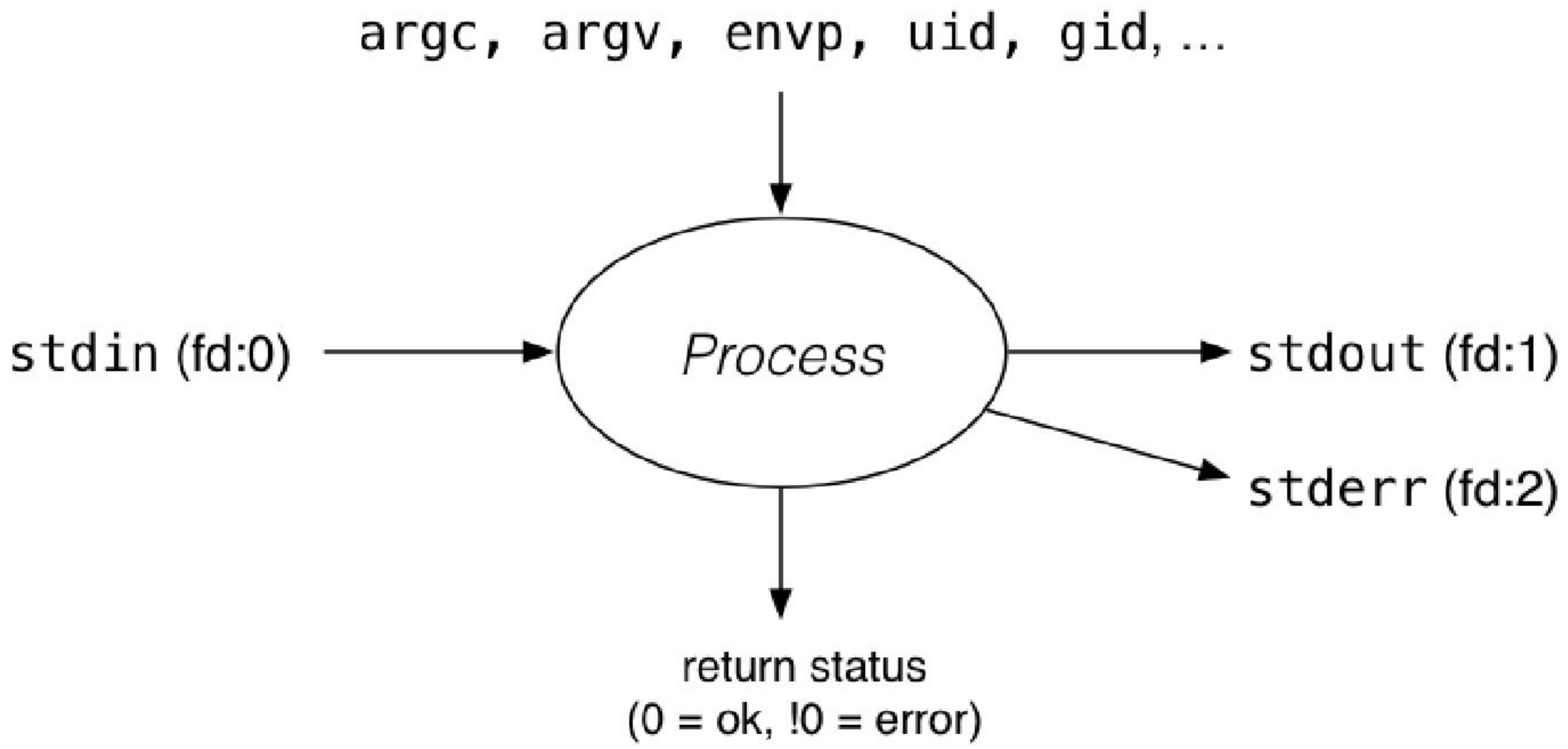
Question 3: What were some challenges with Extended ASCII code pages?

A computer process



- A process is a program running in an environment
- The operating system manages starting, stopping processes

Environment for Unix/Linux Processes



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)

Processes (cont.)

On Unix/Linux:

- each process has a unique process ID, or PID: a positive integer, type pid_t, defined in <unistd.h>
- PID 1: init, used to boot the system.
- low-numbered processes usually system-related, started at boot
    - … but PIDs are recycled, so this isn't always true
- some parts of the OS may appear to run as processes
    - many Unix-like systems use PID 0 for the operating system

Parent Processes

Each process has a parent process.

- initially, the process that created it;
- if a process' parent terminates, its parent becomes init (PID 1)

A process may have child processes

- these are processes that it created
- a parent may later kill the child processes


FBI
Google
Killing a child
Google Search
I'm Feeling Lucky
FBI
Google
Killing a child process stackoverflow
Google Search
I'm Feeling Lucky

Unix tools

Unix provides a range of tools for manipulating processes

Commands:

- sh … creating processes via object-file name
- ps … showing process information
- w … showing per-user process information
- top … showing high-cpu-usage process information
- kill … sending a signal to a process

syscalls to get info about a process

`pid_t getpid()`

- requires `#include <sys/types.h>`
- returns the process ID of the current process

`pid_t getppid()`

- requires `#include <sys/types.h>`
- returns the parent process ID of the current process

For more details: man 2 getpid

Not used in this course: `getpgid()` ... get process group ID

Minimal example for getpid() and getppid():

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

int main(void){
    printf("My PID is (%d)\n", getpid());
    printf("My parent's PID is (%d)\n", getppid());
    return 0;
}
```

Environment variables

Unix-like shells have simple syntax to set environment variables

- common to set environment in startup files (e.g .profile)
- then passed to any programs they run
- Almost all program pass the environment variables they are given to any programs they run
    - perhaps add/edit the value of specific environment variables

Environment variables

Provides simple mechanism to pass settings to all programs, e.g

- timezone (TZ)
- user’s prefered language (LANG)
- directories to search for promrams (PATH)
- user’s home directory (HOME)

Environment variables: code

When run, a program is passed a set of environment variables:

- array of strings of the form name=value, terminated with NULL.
- access via global variable `environ`

```
// print all environment variables
extern char **environ;
for (int i = 0; environ[i] != NULL; i++) {
    printf("%s\n", environ[i]);
}
```

Demo: environ.c

Environment variables: better code

Many C implementations also provide as 3rd parameter to main:

```
int main(int argc, char *argv[], char *env[])
```

Best method: Access using **`getenv()`** and **`setenv()`**

setenv() - set an environment variable

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite);
```

- adds name=value to environment variable array
- if name in array, value changed if overwrite is non-zero

Returns 0 if success, or -1 if error (error stored in *errno*)

getenv() - get an environment variable

```c
#include <stdlib.h>
char *getenv(const char *name);
```

- Reads value from environment variable array by name
- if name not in array, returns NULL

Demo: get_status.c

Multi-Tasking

On a typical modern operating system…

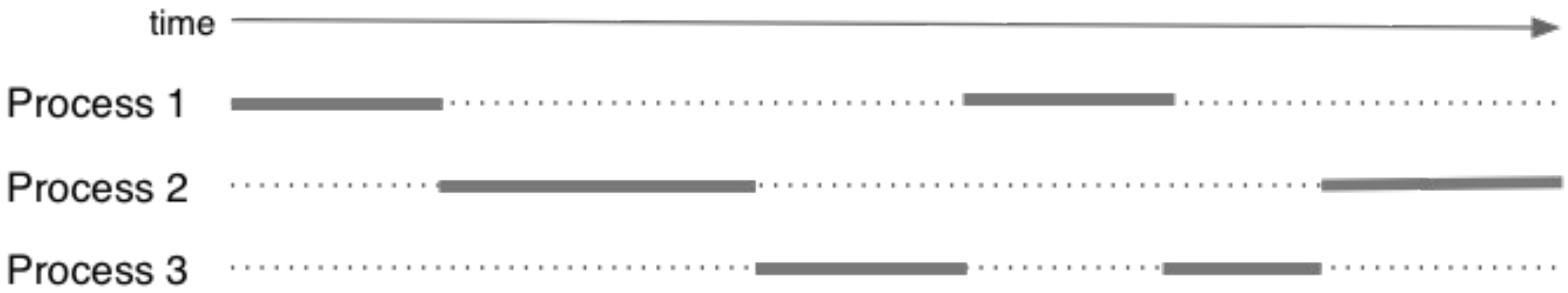
- multiple processes are active "simultaneously" (multi-tasking)
- operating systems provides a virtual machine to each process:
- each process executes as if it is the only process running
- e.g. each process has its own address space (N bytes, addressed 0..N-1)

Multi-Tasking (cont.)

When there are multiple processes running on the machine,

- a process uses the CPU, until it is preempted or exits;
- then, another process uses the CPU, until it too is preempted.
- eventually, the first process will get another run on the CPU.

Multi-Tasking (cont.) (cont.)



Overall impression: three programs running simultaneously.

(In practice, these time divisions are imperceptibly small!)

Preemption — When? How?

What can cause a process to be preempted?

- it ran “long enough”, and the OS replaces it by a waiting process
- it needs to wait for input, output, or other some other operation

On preemption…

- the process's entire state is saved
- the new process's state is restored
- this change is called a context switch
- context switches are very expensive!

Which process runs next?

The *scheduler answers this.

The operating system's process scheduling attempts to:

- fairly sharing the CPU(s) among competing processes,
- minimize response delays (lagginess) for interactive users,
- meet other real-time requirements (e.g. self-driving car),
- minimize number of expensive context switches

Process-related Unix/Linux Functions/syscalls

Creating processes:

- `system()`, `popen()` … create a new process via a shell
    - convenient but major security risk
- `posix_spawn()` … create a new process.
- `fork()` `vfork()` … duplicate current process.

  (actually, “modern” `fork()` is actually `clone()`… sshhhh)
- `exec()` family … replace current process.

Process-related Unix/Linux Functions/syscalls

Destroying processes:

- **`exit()`** ... terminate current process, see also
- **`_exit()`** ... terminate immediately

  (**`atexit`** functions not called, stdio buffers not flushed)
- **`kill()`** ... send signal to a process

Monitoring changes:

- **`waitpid()`** ... wait for state change in child process

exec() family - replace yourself

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

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
- Most of the current process is re-initialized:
- e.g. new address space is created - all variables lost

exec() family - replace yourself

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

- open file descriptors survive
- e.g, stdin & stdout remain the same
- PID unchanged
- if successful, exec does not return … where would it return to?
- on error, returns -1 and sets errno

Example: using exec()

```
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

fork() — clone yourself

```
#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);
}
```

```
$ dcc fork.c
$ a.out
I am the parent because fork() returned 2884551.
I am the child because fork() returned 0.
$
```

Demo: fork.c

waitpid() — wait for process to change state

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

`status` 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`**.

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

```c
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("execvpe"); // 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 has some dangers, e.g. a fork bomb

```c
#include <stdio.h>
#include <unistd.h>
int main(void) {
    // creates 2 ** 10 = 1024 processes
    // which all print fork bomb then exit
    for (int i = 0; i < 10; i++) {
        fork();
    }
    printf("fork bomb\n");
    return 0;
}
```

Demo: fork_bomb.c

system() — run another program

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

Runs command via /bin/sh.

Waits for command to finish and returns exit status

system() — convenient but risky

Convenient ... but extremely dangerous —

very brittle; highly vulnerable to security exploits

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 **exec()**

- **fork()** duplicates the current process (parent+child)
- **exec()** “overwrites” the current process (run by child)

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 - 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

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);
```

Example: posix_spawn() versus system()

Running ls -ld via posix_spawn()

```
char *ls_argv[2] = {"/bin/ls", "-ld", NULL};
pid_t pid; int ret;
extern char **environ;
if((ret = posix_spawn(&pid, "/bin/ls", NULL, NULL, ls_argv, environ)) != 0) {
    errno = ret; perror("spawn"); exit(1);
}
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); exit(1);
}
```

Example: posix_spawn() versus system()

Running ls -ld via system()

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

Setting environment var for child process

```c
// 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);
```

Aside: Zombie Processes (advanced)

Aside: Zombie Processes (advanced)

A process cannot terminate until its parent is notified.

- notification is via wait/waitpid or SIGCHLD signal

Zombie process = exiting process waiting for parent to handle notification

- parent processes which don't handle notifications create long-term zombie processes
- wastes some operating system resources

Orphan process = a process whose parent has exited

- when parent exits, orphan assigned PID 1 (init) as its parent
- init always accepts notifications of child terminations

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 atexit

```c
#include <stdlib.h>

void _exit(int status);
```

- terminates current process without triggering functions registered as atexit()
- stdio buffers not flushed
- usually used by children of fork() when exiting

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

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()**

popen() — convenient way to set up pipe

```
#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() — a bit 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()

```c
// 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()

```
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