



# Sistemas Operativos (75.08)

Primer cuatrimestre 2022 - FIUBA



# Checklist

- ✓ Página de la materia: [fisop.github.io](https://fisop.github.io)
  - Completar el formulario de alta!
- ✓ Lista de correo:
  - [fisop-consultas@googlegroups.com](mailto:fisop-consultas@googlegroups.com)
- ✓ Consultas administrativas a:
  - [fisop-doc@googlegroups.com](mailto:fisop-doc@googlegroups.com)
- ✓ **Discord de la materia!**
  - Al unirse, cambiar el nick para que los podamos reconocer y agregarlos.



# ¿Cómo usar el repositorio?

- Para el primer lab
- La [página de entregas](#) explica integraciones
- Crear un *Pull Request* con todos los cambios

```
// 0) Clonar en un directorio local (e.g. mylabs)
$ git clone git@github.com:fiubatps/sisop_2022a_fresia mylabs

// 1) Agregar el repositorio remoto con los esqueletos
$ cd mylabs
$ git remote add catedra https://github.com/fisop/labs

// 2) Creación de la rama base para el lab fork
$ git checkout -b base_fork
$ git push -u origin base_fork

// 3) Integración del "esqueleto" lab fork
$ git fetch --all
$ git checkout base_fork
$ git merge catedra/fork
$ git push origin base_fork

// 4) Creación de la rama entrega para el lab fork
$ git checkout -b entrega_fork
$ git push -u origin entrega_fork

// Asegurarse de siempre commitear y pushear los cambios
// en el branch entrega_fork
```



## ¿Cómo crear el PR?

- Se debe crear un PR
  - base\_fork como base
  - entrega\_fork como target
- Sólo se deberían mostrar sus cambios

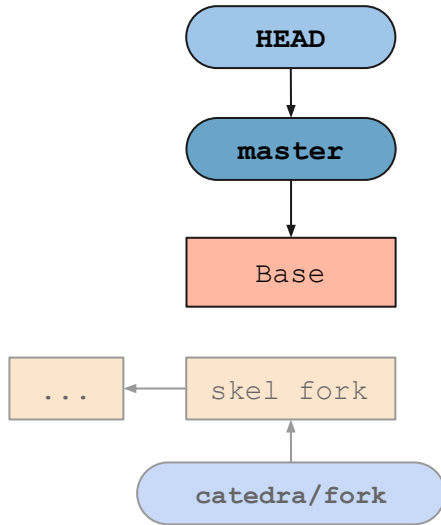
```
// Para ver el branch actual
$ git branch

// 5) Trabajar en los ejercicios
$ git add primes.c
$ git commit -m "Resuelvo primes"
$ git push origin entrega_fork

// 6) Cuando estén todos los cambios, crear el PR desde la UI
```

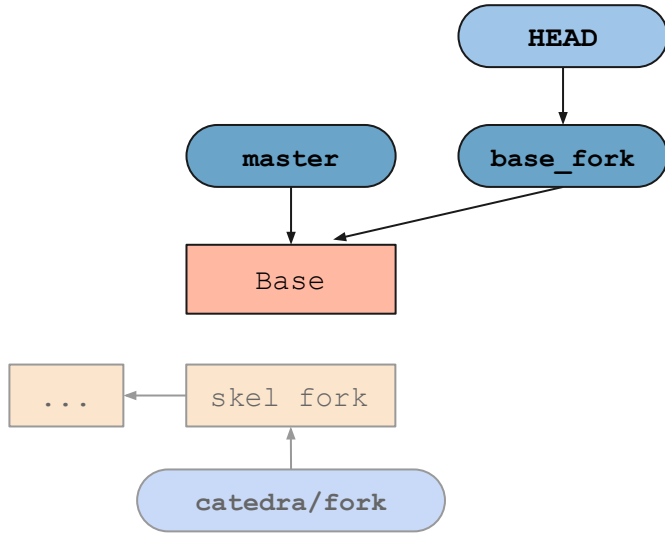
# Ejemplo de historia de git: (1)

git clone ...  
git remote add ...

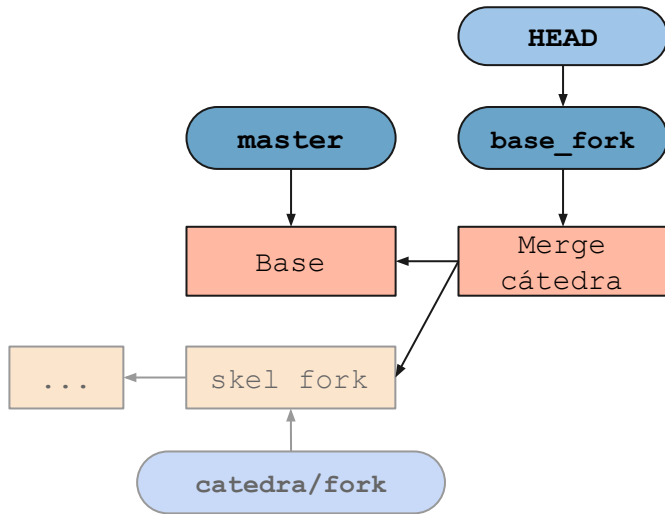


## Ejemplo de historia de git: (2)

```
git checkout -b base_fork  
git push -u origin base_fork
```



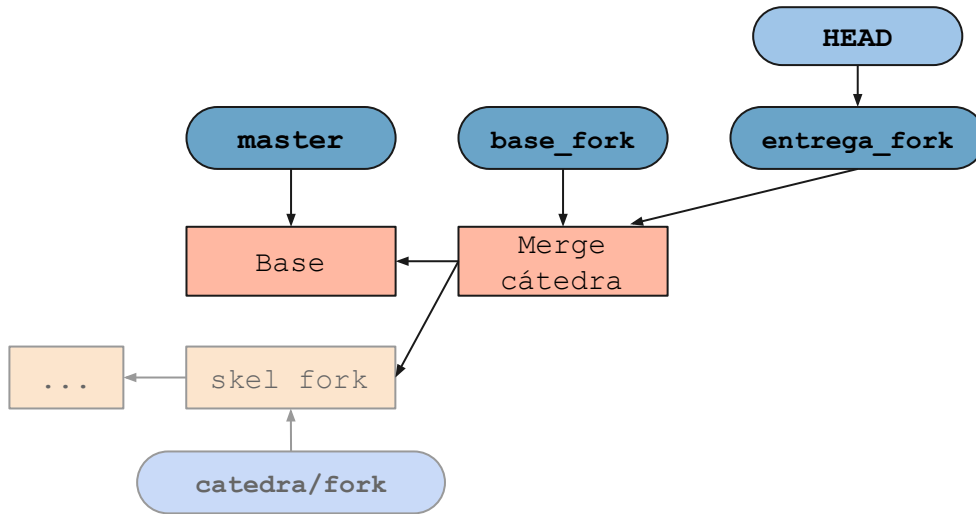
## Ejemplo de historia de git: (3)



```
git fetch --all  
git checkout base_fork  
git merge catedra/fork  
git push origin base_fork
```

## Ejemplo de historia de git: (4)

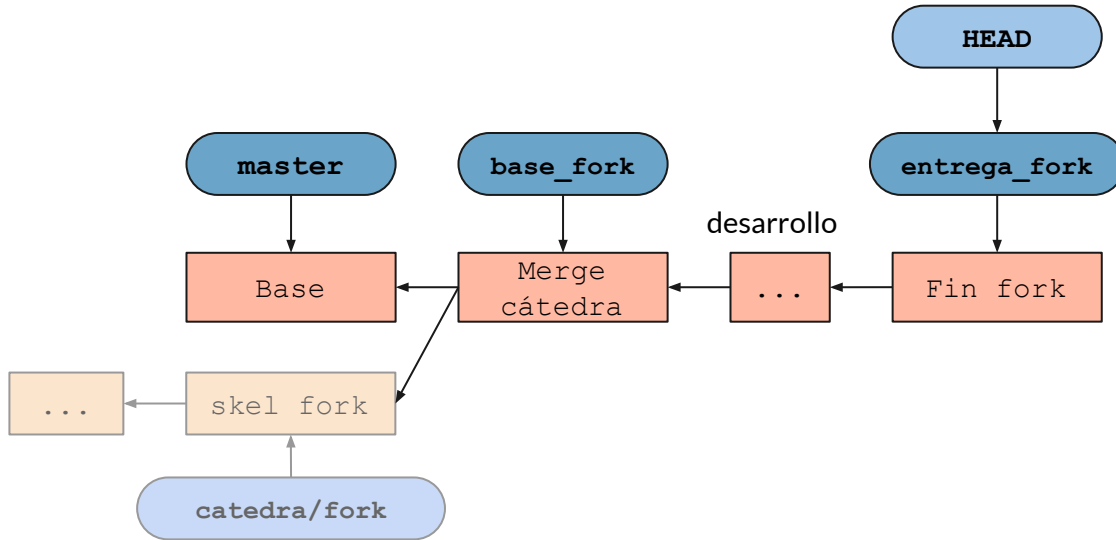
`git checkout -b entrega_fork`  
`git push -u origin entrega_fork`





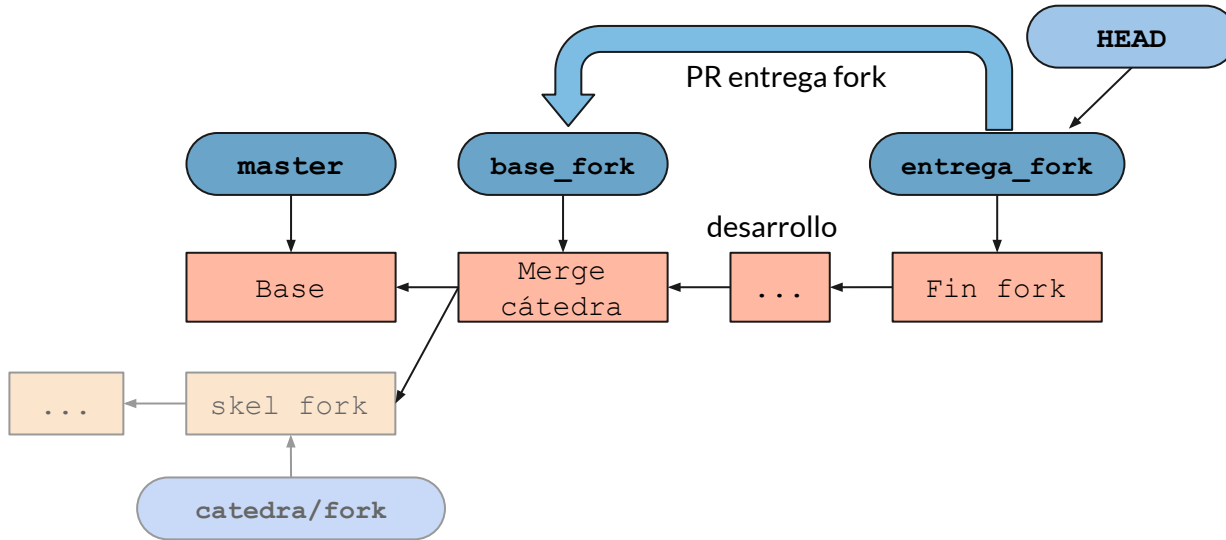
## Ejemplo de historia de git: (5)

git add ...  
git commit ...  
git push origin entrega\_fork

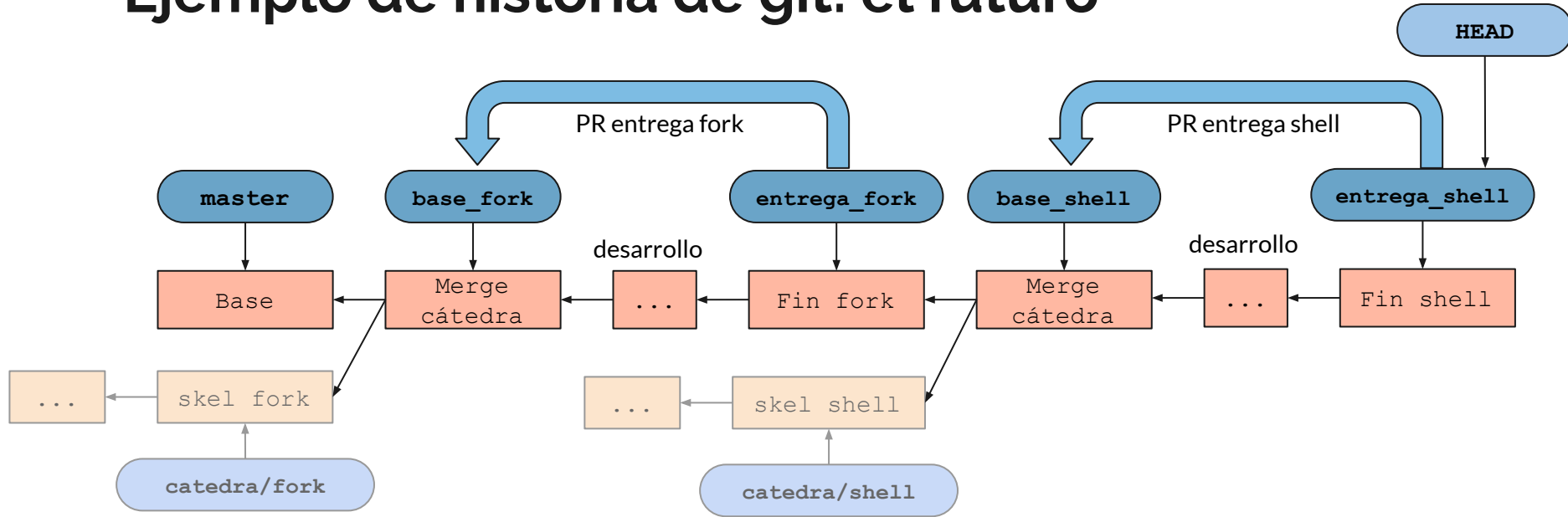


## Ejemplo de historia de git: (6)

PR desde UI github  
entrega\_fork -> base\_fork



# Ejemplo de historia de git: el futuro



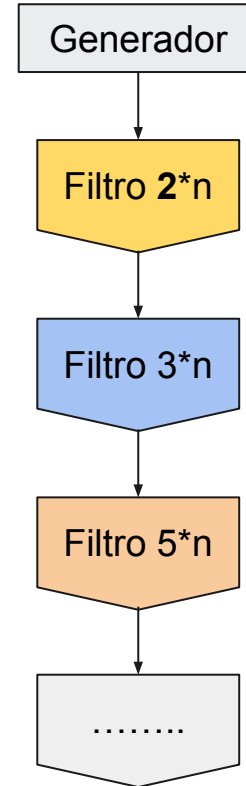
---

# Lab fork

## Primera parte - cont.

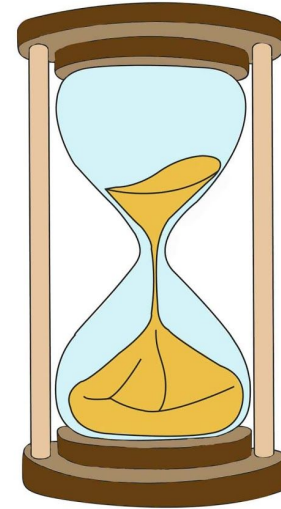
# Tarea: primes

- ¿Progreso?
- ¿Cómo esperar a que un programa termine?



## wait(2)

- Espera a que **algún proceso** hijo termine
- Es **bloqueante** si hay procesos que esperar
- Devuelve el **pid** del hijo que terminó
  - Es posible obtener el exit code





## wait(2) - ejemplo 0

- ¿En qué orden se imprimen los mensajes?
- ¿Qué *exit code* tiene el proceso hijo?  
¿Y el proceso padre?
- ¿Cuál es el parámetro de *wait*?

```
int main(int argc, char* argv[]) {
    int i = fork();

    if (i == 0) {
        printf("Soy el proceso hijo y mi pid es: %d\n", getpid());
        sleep(2);
        printf("Proceso hijo termina (%d)\n", getpid());
        exit(17);
    } else {
        printf("Soy el proceso padre y mi pid es: %d\n", getpid());

        int ret = wait(NULL);
        printf("PID %d terminó\n", ret);
        printf("Proceso padre termina (%d)\n", getpid());
    }
}
```



## wait(2) - ejemplo 1

- Recuperando el exit code del proceso hijo
- Macro *WEXITSTATUS*

```
int main(int argc, char* argv[]) {
    int i = fork();

    if (i == 0) {
        printf("Soy el proceso hijo y mi pid es: %d\n", getpid());
        sleep(2);
        printf("Proceso hijo termina (%d)\n", getpid());
        exit(17);
    } else {
        printf("Soy el proceso padre y mi pid es: %d\n", getpid());
        int wstatus;
        int ret = wait(&wstatus);
        printf("PID %d terminó con %d\n", ret, WEXITSTATUS(wstatus));
        printf("Proceso padre termina (%d)\n", getpid());
    }
}
```





## wait(2) - macros útiles

- **WIFEXITED(*wstatus*)**: returns true if the child terminated normally
- **WEXITSTATUS(*wstatus*)**: returns the exit status of the child
- **WIFSIGNALED(*wstatus*)**: returns true if the child process was terminated by a signal
- **WTERMSIG(*wstatus*)**: returns true if the child process was terminated by a signal



## wait(2) - ejemplo 2

- ¿Cómo podríamos forzar cada una de las ramas del *if*?
- ¿Cómo se ve desde la shell cada código de salida?

```
int main(int argc, char* argv[]) {
    int i = fork();

    if (i == 0) {
        printf("Soy el proceso hijo y mi pid es: %d\n", getpid());
        sleep(2);
        printf("Proceso hijo termina (%d)\n", getpid());
        exit(17);
    } else {
        printf("Soy el proceso padre y mi pid es: %d\n", getpid());

        int wstatus;
        int ret = wait(&wstatus);
        if (WIFEXITED(wstatus)) {
            printf("PID %d terminó con %d\n", ret, WEXITSTATUS(wstatus));
        } else if (WIFSIGNALED(wstatus)) {
            printf("PID %d fue terminado con %d\n", ret, WTERMSIG(wstatus));
        }

        printf("Proceso padre termina (%d)\n", getpid());
    }
}
```



## getppid(2) y waitpid(2)

- **getppid(2)**: returns the *parent process PID*
- **waitpid(2)**: allows to wait *a specific child process*
  - ¿Qué pasa si el hijo no terminó?
  - ¿Qué pasa si trato de esperar un pid que no es mi proceso hijo?
- Ver **WNOWAIT** y **WNOHANG**

```
int main(int argc, char* argv[]) {
    int i = fork();

    if (i == 0) {
        printf("[hijo] Mi pid es: %d\n", getpid());
        printf("[hijo] Mi ppid es: %d\n", getppid());
        printf("[hijo] Termina\n");
        _exit(17);
    } else {
        printf("[padre] Mi pid es: %d\n", getpid());
        int r = waitpid(i, NULL, 0);
        if (r >= 0) {
            printf("[padre] Hijo %d ha terminado\n", r);
        }
        r = waitpid(100, NULL, 0);
        if (r < 0) {
            perror("[padre] waitpid");
        }
        printf("[padre] Termina\n");
    }
}
```



## wait(2) - huérfanos y zombies

- ¿Qué pasa si el proceso padre termina *sin hacer wait* a un proceso hijo?
- ¿Qué pasa si el proceso hijo termina, pero el proceso padre no hace *wait*?



## wait(2) - huérfanos

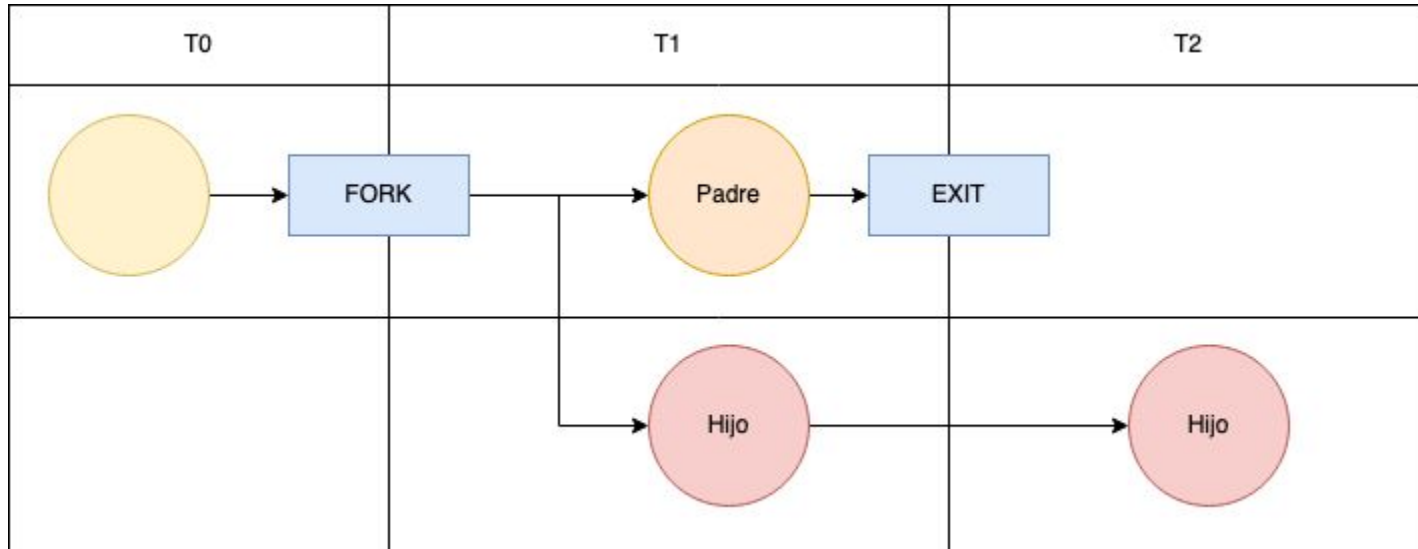
- ¿Qué pasa si el proceso padre termina *sin* hacer *wait* a un proceso hijo?
- ¿Qué imprime el proceso hijo?
- Ver notas en **man 2 wait**
- ¿Qué es un *subreaper*?

```
int main(int argc, char* argv[]) {
    int fds[2];
    pipe(fds);
    int msg;
    int i = fork();

    if (i == 0) {
        close(fds[1]);
        printf("[hijo] Mi pid es: %d\n", getpid());
        printf("[hijo] Mi ppid es: %d\n", getppid());
        read(fds[0], &msg, sizeof(msg));
        sleep(2); // Wait for parent to die

        printf("[hijo] Mi ppid es: %d\n", getppid());
        printf("[hijo] Termina\n");
        close(fds[0]);
        _exit(17);
    } else {
        close(fds[0]);
        printf("[padre] Mi pid es: %d\n", getpid());
        printf("[padre] Terminas sin esperar\n");
        close(fds[1]);
    }
}
```

## wait(2) - huérfanos (diag.)





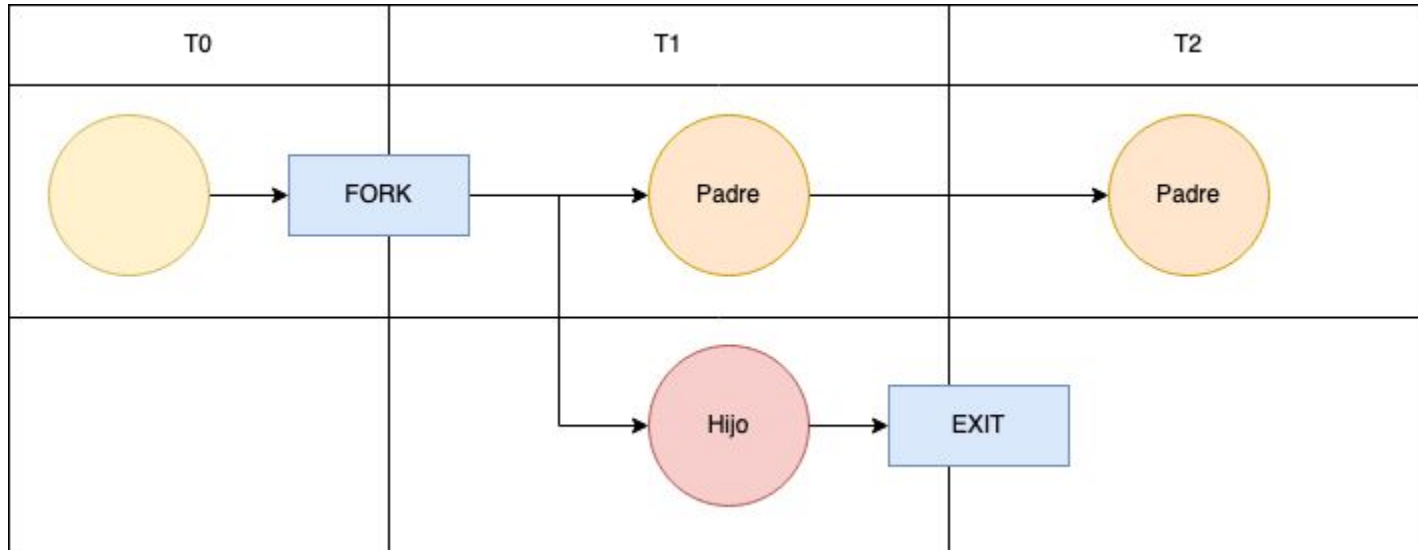
## wait(2) - zombies

- ¿Qué pasa si el proceso hijo termina, pero el proceso padre no hace *wait*?
- ¿Sigue existiendo el proceso? ¿Cómo se ve desde *ps*?
- ¿Qué pasa con los *zombies huérfanos*?

```
int main(int argc, char* argv[]) {
    int i = fork();

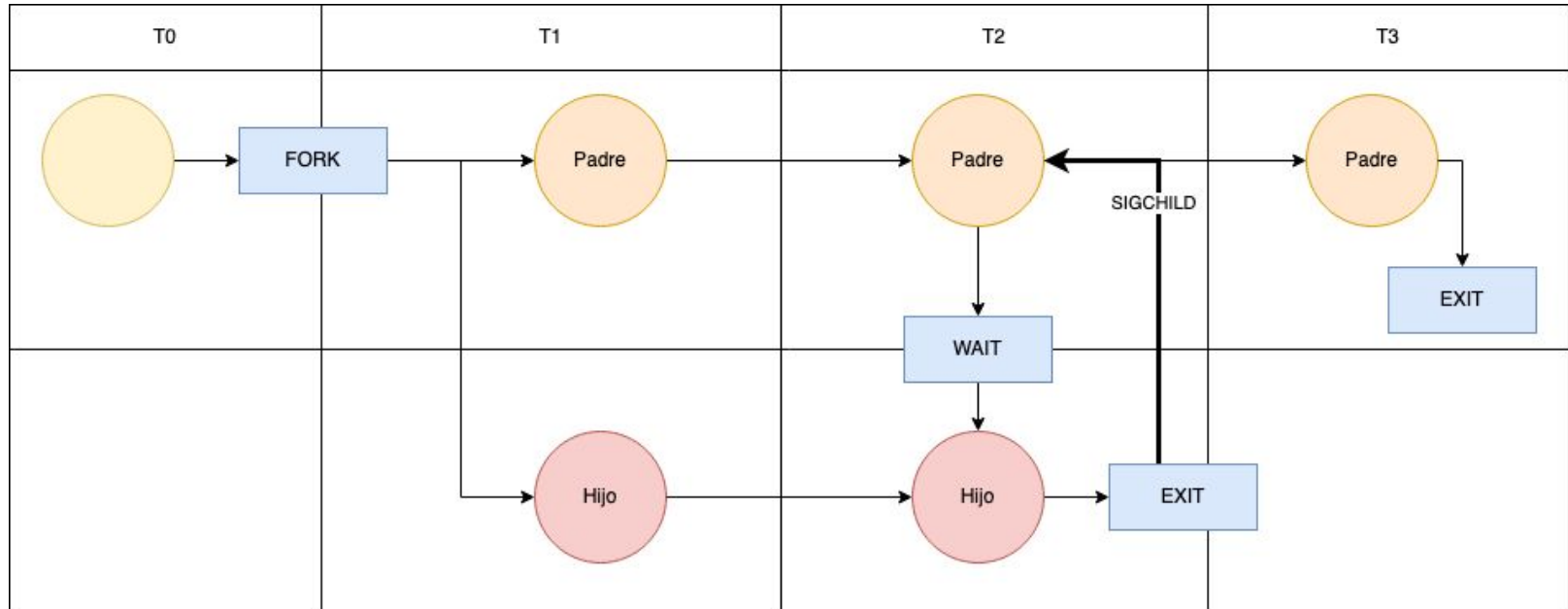
    if (i == 0) {
        printf("[hijo] Mi pid es: %d\n", getpid());
        printf("[hijo] Mi ppid es: %d\n", getppid());
        printf("[hijo] Termina\n");
        _exit(17);
    } else {
        printf("[padre] Mi pid es: %d\n", getpid());
        // Simulamos que el padre hace
        /// otras tareas sin hacer wait
        while (1) {}
        printf("[padre] Terminas sin esperar\n");
    }
}
```

## wait(2) - zombies (diag.)





## wait(2) - flujo normal (diag.)



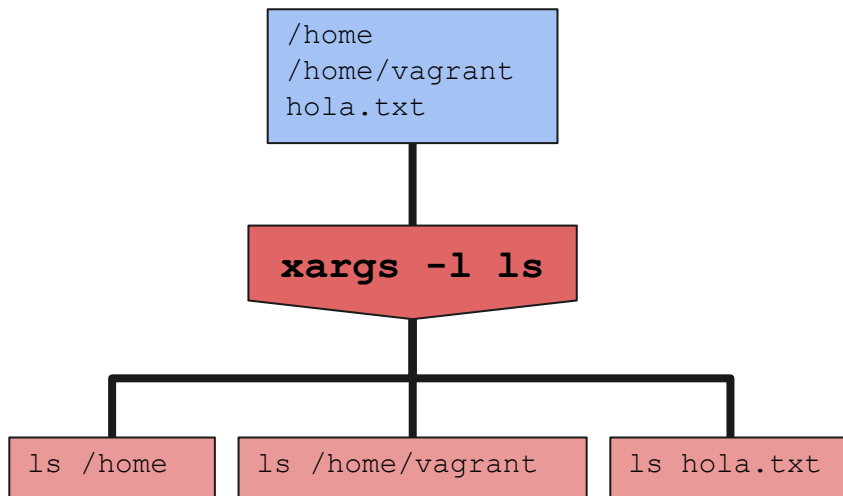
---

# Lab fork

## Segunda parte

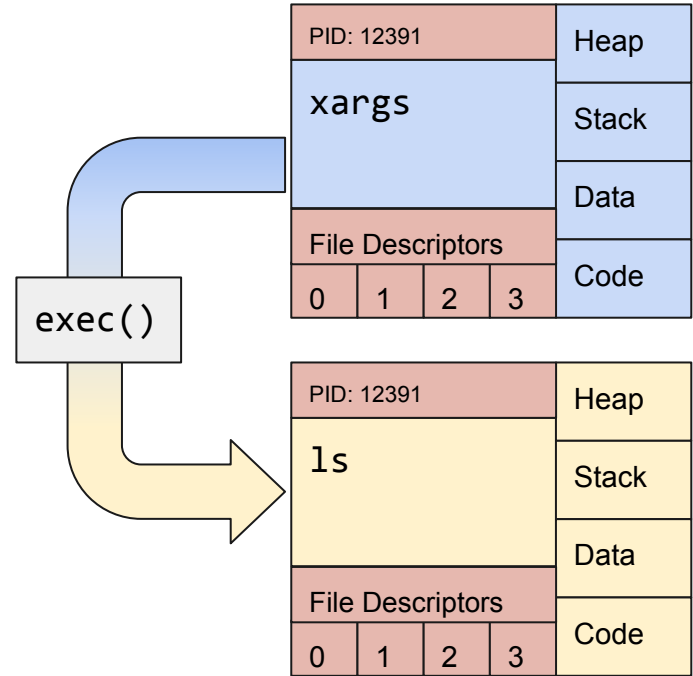
# Tarea: xargs

- Herramienta versátil para ejecutar un **programa** repetidas veces sobre varios inputs
- Lee **stdin** y usa esos valores como argumentos para un **comando**
- Ejecuta un binario arbitrario!!
  - ¿Cómo?



# execve(2)

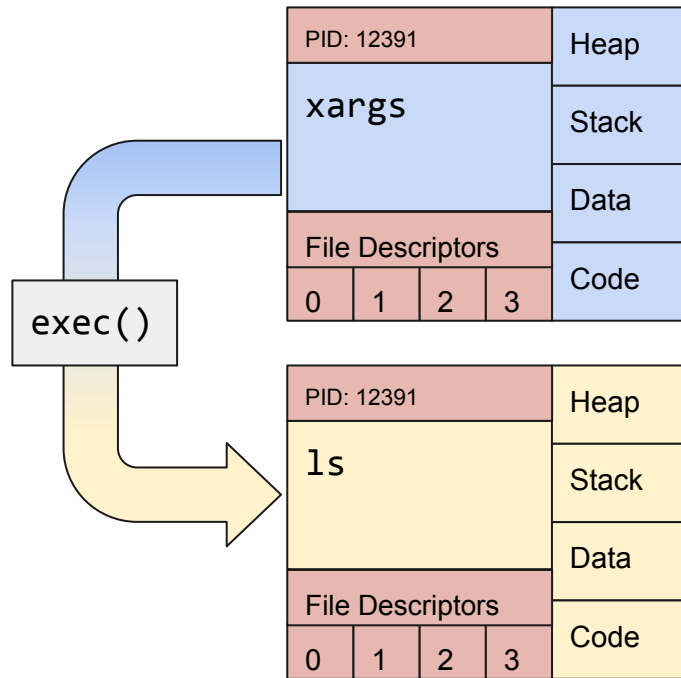
- Cambia la **imagen** de un proceso
  - Memoria virtual, entorno, argumentos
- Mantiene todo lo demás
  - fd, pid, ppid, etc
- Única **syscall**



# Familia de exec(3)

execl, execlp, execl, execv, execvp, execvpe

- **vector vs list:** si recibe un array de cadenas como argumento, o una lista
- **environment:** permite sobrescribir el entorno
- **path:** facilita la búsqueda de binarios en **PATH**
- Se recomienda **execvp()**





## execvp(3) - ejemplos 0 y 1

- ¿Qué hacen estos programas?
- ¿Se ejecuta en algún momento el print “Terminado”?

```
int main(int argc, char* argv[]) {  
    char *args[] = {"echo", "hello world!", NULL};  
    execvp("echo", args);  
  
    printf("Terminando: %d\n", getpid());  
}
```

```
int main(int argc, char* argv[]) {  
    // Notar que argv+1 es lo mismo que  
    // {argv[1], argv[2], ..., argv[argc-1], NULL}  
    execvp(argv[1], argv+1);  
  
    printf("Terminando: %d\n", getpid());  
}
```



## execvp(3) - ejemplo 2

- ¿Qué hace este programa?
- ¿Qué efecto tiene *close(1)*?
- ¿Les suena a algo que hayan usado?
- Extra: investigar la función **dup(2)**

```
int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Se necesita un argumento\n");
        exit(-1);
    }

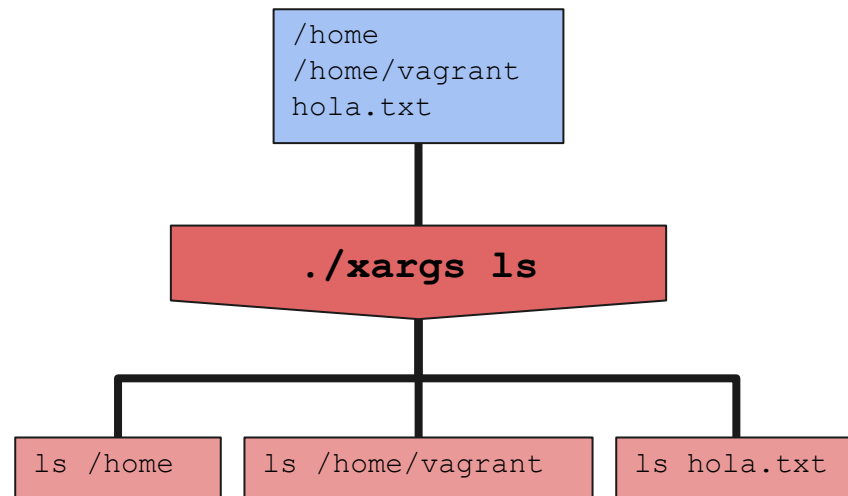
    close(1);
    int fd = open("hola.txt", O_CREAT | O_RDWR, 0644);
    printf("Archivo abierto en %d\n", fd);

    // Notar que argv+1 es lo mismo que
    // {argv[1], argv[2], ..., argv[argc-1], NULL}
    execvp(argv[1], argv+1);

    printf("Terminando: %d\n", getpid());
}
```

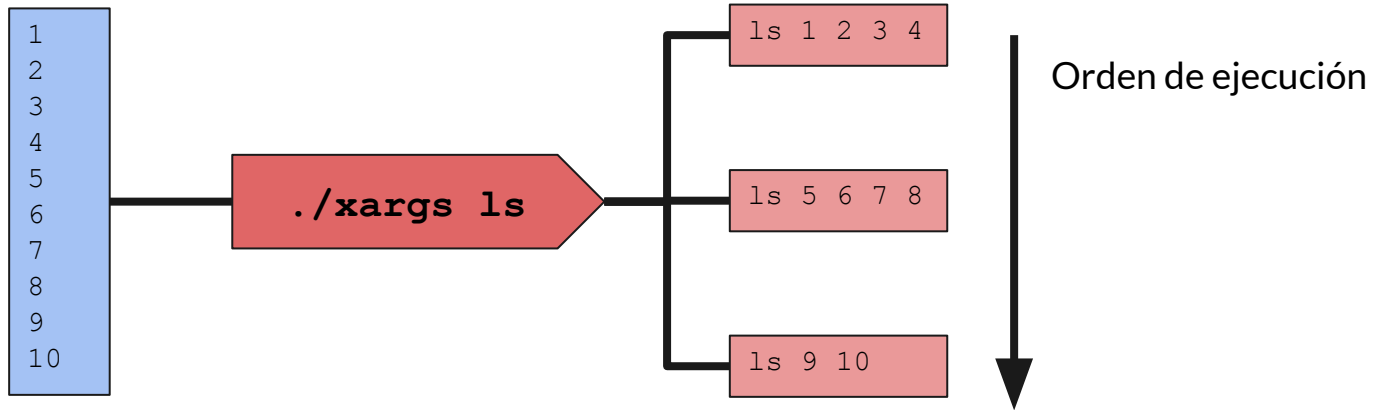
# Tarea: xargs

- Leer línea a línea, y pasar **NARGS** líneas leídas como parámetros al comando
  - equivalente a `xargs -n NARGS <cmd>`
- Los argumentos son las líneas leídas sin el `\n`
- Las ejecuciones son **secuenciales**
  - Usar **wait** para garantizar una ejecución a la vez
- Challenge: permitir **hasta 4** ejecuciones paralelas con un flag **-P**





## Tarea: xargs *NARGS* = 4





## Otras syscalls útiles

- *open(2)*: apertura de archivos
- *read(2)* y *write(2)*: escritura/lectura de archivos. Familiarizarse con errores relacionados.
- *dup(2)*: duplicación de file descriptors (muy útil para shell)
- *kill(2)*: útil para *enviar señales* (e.g. *SIGTERM* o *SIGSTOP*)
- *sigaction(2)*: permite definir respuestas a señales (e.g. *SIGCHLD*)
- *syscalls(7)*: para la lista completa

# Tarea: find

- Utilidad que encuentra archivos
  - Nombre, tipo, metadata
- Implementación de una versión reducida
  - Busca por **subcadenas** en el nombre del archivo
  - Flag para indicar sensibilidad

## find

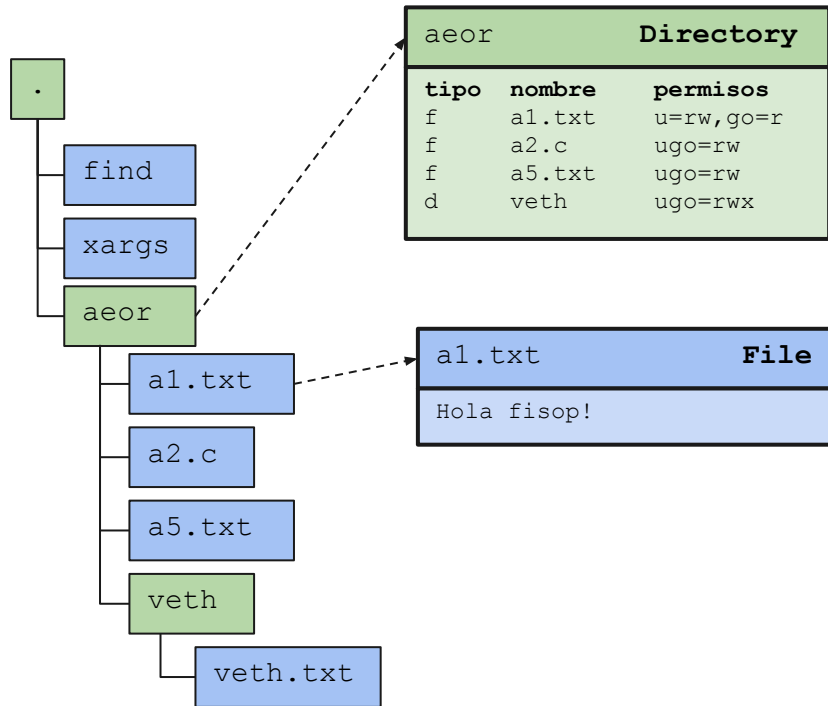
JULIA EVANS  
@b0rk

<p>find searches a directory for files</p> <p>find /tmp -type d -print</p> <p>↑            ↑            ↑</p> <p>directory to search    which files    action to do with the files</p> <p>here are my favourite find arguments!</p>	<p><b>-name</b></p> <p>the filename! eg</p> <p>-name '*.txt'</p>	<p><b>-type [TYPE]</b></p> <p>f: regular file    l: symlink</p> <p>d: directory    + more!</p>
<p><b>-mtime NUM</b></p> <p>files that were modified at most NUM days in the past (also ctime, atime)</p>	<p><b>-path</b></p> <p>search the full path!</p> <p>-path '/home/*/*.go'</p>	<p><b>-maxdepth NUM</b></p> <p>only descend NUM levels when searching a directory</p>
<p><b>-exec COMMAND</b></p> <p>action: run COMMAND on every file found</p>	<p><b>-print</b></p> <p>action: print filename of files found. The default. Use -print0 with xargs -0!</p>	<p><b>locate</b></p> <p>The locate command searches a database of every file on your system.</p> <p>good: faster than find</p> <p>bad: can get out of date</p> <p><b>\$sudo updatedb</b></p> <p>updates the database</p>
	<p><b>-delete</b></p> <p>action: delete all files found</p>	

- ¿Cómo acceder a directorios?
- ¿Cómo filtrar nombres?

# Filesystem overview

- Existen **Directorios** y **Archivos**
- Un **Directorio** contiene una referencia a otros **Directorios** y **Archivos**
  - Incluido nombre, permisos, mt, at, etc.
- Un **Archivo** contiene solo datos






## opendir(3) y readdir(3)

- No son syscalls
- Estructuras **DIR\*** y **struct dirent\***
- Distintos tipos de archivo

```
DIR *opendir(const char *name);

struct dirent *readdir(DIR *dirp);

struct dirent {
    ino_t      d_ino;      /* Inode number */
    off_t      d_off;      /* Not an offset; see below */
    unsigned short d_reclen; /* Length of this record */
    unsigned char d_type;   /* Type of file; not supported
                             by all filesystem types */
    char        d_name[256]; /* Null-terminated filename */
};
```



## opendir(3) - ejemplo

- ¿Qué hace este programa?
- ¿De dónde sale cada una de las líneas que imprime?
- ¿En algún caso imprimirá *tipo desconocido*?

```
int main(int argc, char* argv[]) {
    char *path = "/home/juan"
    DIR *directory = opendir(path);
    if (directory == NULL) {
        perror("error con opendir");
        exit(-1);
    }

    struct dirent* entry;
    while (entry = readdir(directory)) {
        if (entry->d_type == DT_DIR) {
            printf("%s es un directorio\n", entry->d_name);
            continue;
        }

        if (entry->d_type == DT_REG) {
            printf("%s es un archivo regular\n", entry->d_name);
            continue;
        }

        printf("%s es de tipo desconocido\n", entry->d_name);
    }
}
```



## openat(2) - dirfd(3) - fdopendir(3)

- ¿En qué se diferencian?
- Facilitan la operación con directorios y path relativos

```
int openat(int dirfd, const char *pathname, int flags);  
  
int dirfd(DIR *dirp);  
  
DIR *fdopendir(int fd);
```



## strstr(3) y strcasestr(3)

- Versión sencilla para buscar patrones en cadenas
- strstr: Buscar la **aguja** en el pajar
  - case sensitive
- strcasestr: buscar la **AgUJa** en el pajar
  - case insensitive

```
char *strstr(const char *haystack, const char *needle);  
char *strcasestr(const char *haystack, const char *needle);
```



# ¿Dónde aprender más?

- Páginas de manual (man)
- *The Linux Programming Interface*  
de Michael Kerrisk
- *Advanced Programming in the UNIX Environment*  
de Stevens & Rago

