Sistemas Operativos
(75.08/95.03/TA043)

FIUBA

Temas administrativos
(la parte aburrida)

Checklist

- ✓ Página de la materia: fisop.github.io
  - Completar el formulario de alta!
  - Adquirir el calendario de la materia
- ✓ 1 lab + 3 TPs + 1 parcial
- ✓ **No** hay final
- ✓ Calendario con fechas importantes en la página
- ✓ **Discord de la materia!**
  - https://discord.gg/XsFGqyY7
  - Al unirse, cambiar el *alias* para que los podamos reconocer y agregarlos.
- ✓ Consultas administrativas a:
  - fisop-doc@googlegroups.com

Régimen de cursada

https://fisop.github.io/website/regimen/

Discord

Discord

IMPORTANTE: antes de realizar cualquier pregunta revisar el **histórico**
(muy probablemente ya esté respondida)

Intro a entorno Unix

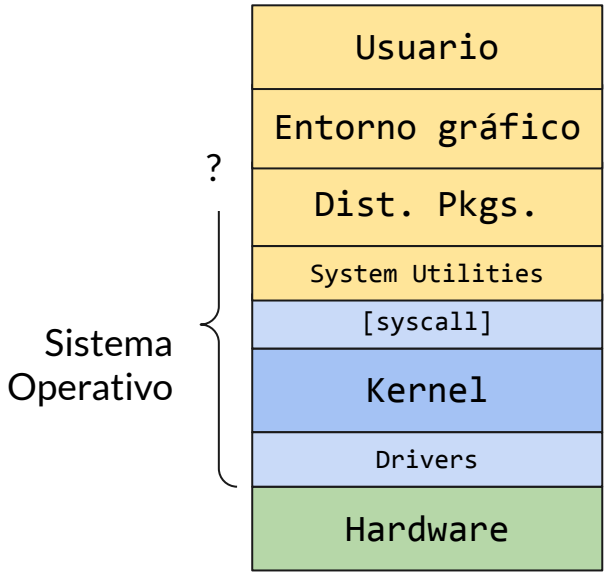
Un muy breve recorrido por la terminal

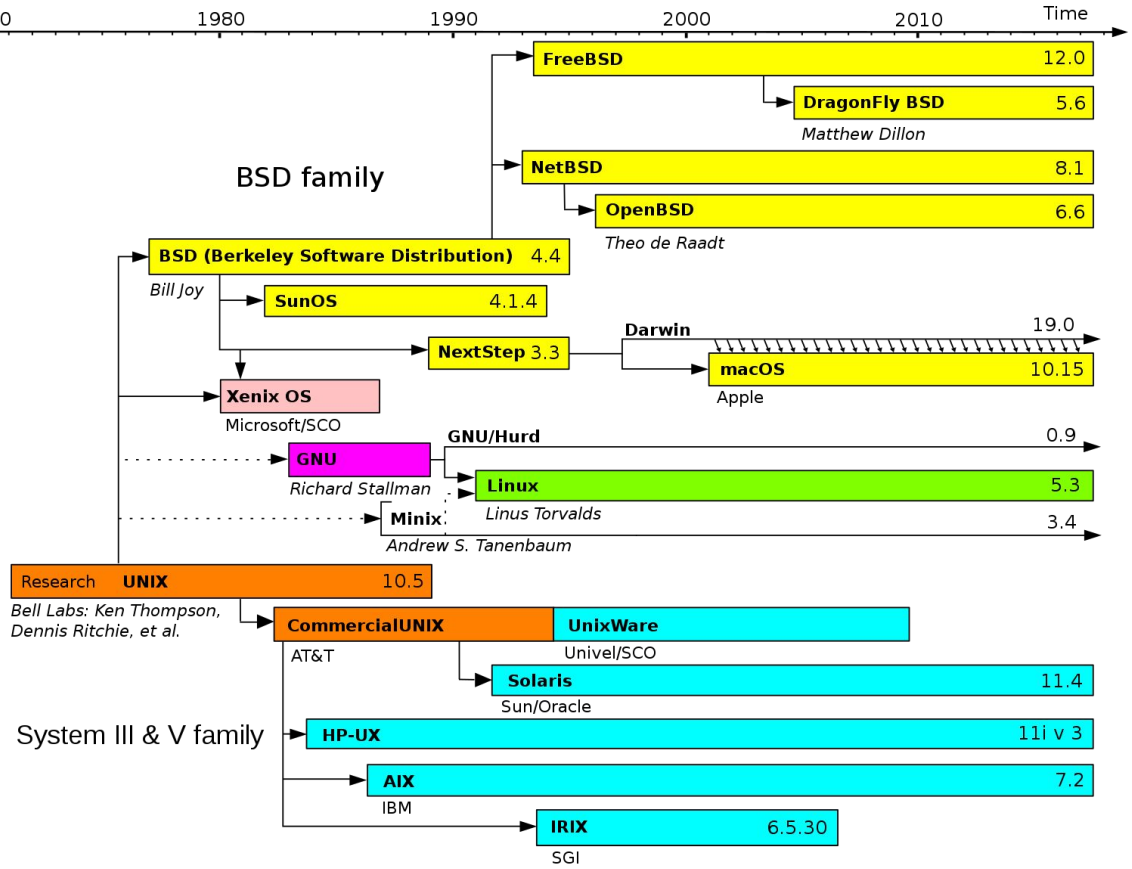
- Sistema Operativo
- Kernel
- UNIX
- Linux
- GNU
- BSD

- Ubuntu
- Debian
- Windows
- POSIX
- Línea de comandos/Shell
- Drivers

¿Qué quiere decir Unix?

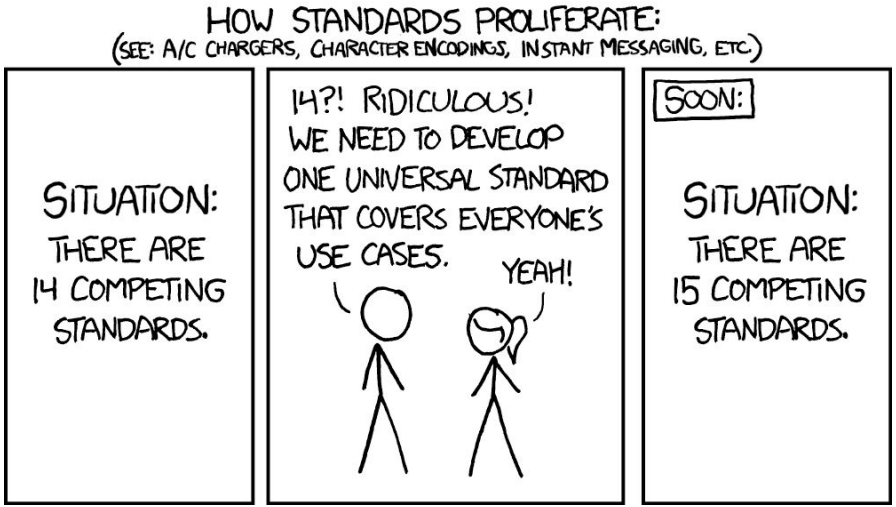
- Es una *familia* de sistemas operativos
  - **Descendientes de UNIX**
  - De propósito general
- Varios componentes:
  - Un kernel
  - Una línea de comandos (*shell*)
  - Librerías y *headers*
  - Utilidades del sistema: **ls, ps, find, grep, sed**
- Filosofía Unix




1970
1980
1990
2000
2010
Time
FreeBSD 12.0
DragonFly BSD 5.6
Matthew Dillon
BSD family
NetBSD 8.1
OpenBSD 6.6
Theo de Raadt
BSD (Berkeley Software Distribution) 4.4
Bill Joy
SunOS 4.1.4
Darwin
19.0
NextStep 3.3
macOS 10.15
Apple
Xenix OS
Microsoft/SCO
GNU/Hurd
0.9
GNU
Richard Stallman
Linux 5.3
Linus Torvalds
Minix
3.4
Andrew S. Tanenbaum
Research UNIX 10.5
Bell Labs: Ken Thompson,
Dennis Ritchie, et al.
CommercialUNIX
UnixWare
Univel/SCO
AT&T
Solaris 11.4
Sun/Oracle
System III & V family
HP-UX 11i v 3
AIX 7.2
IBM
IRIX 6.5.30
SGI

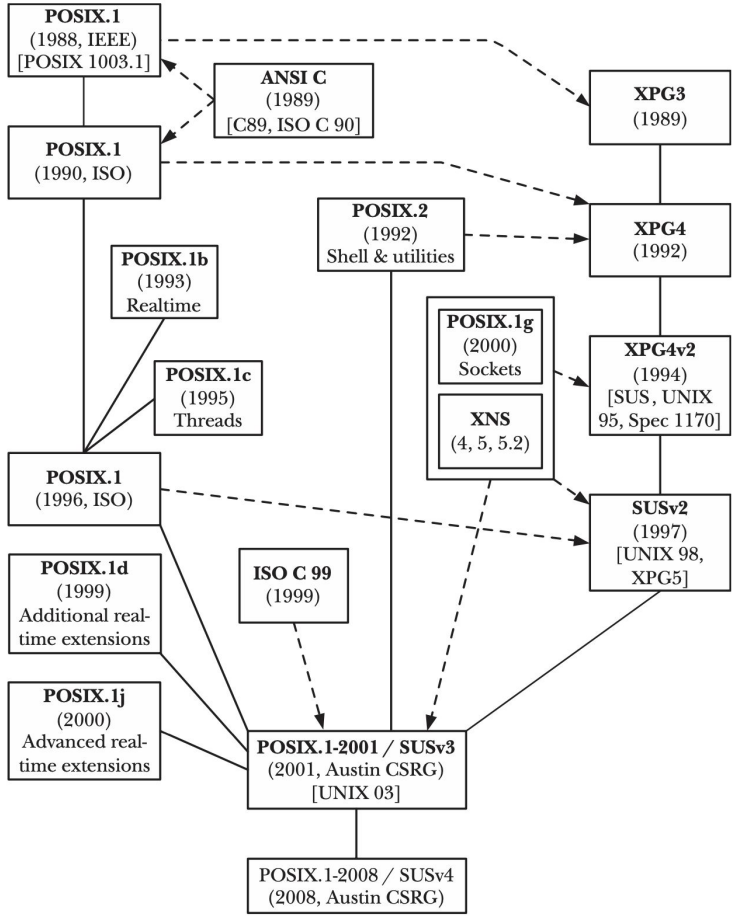
Estándares POSIX y SUSv4

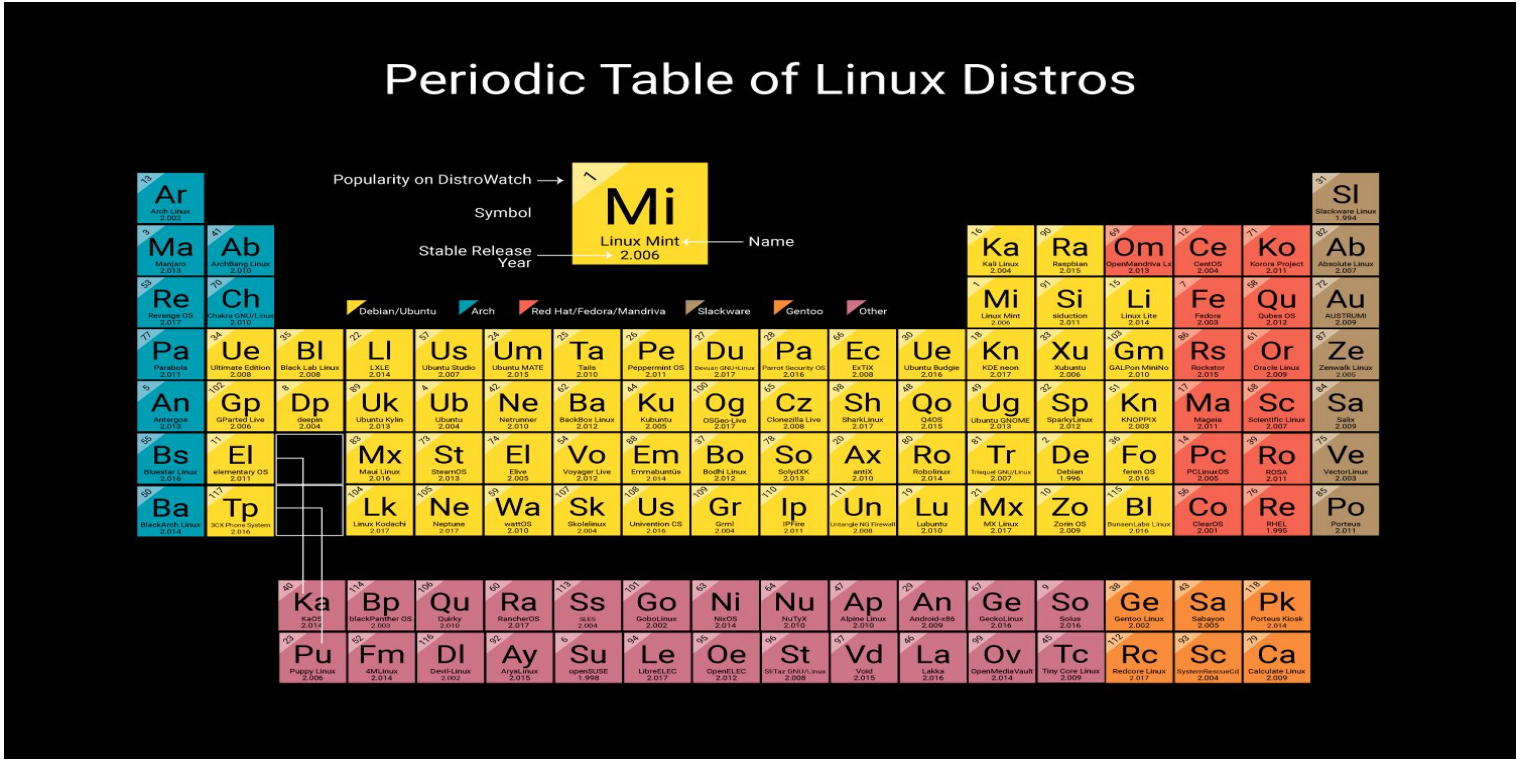
- Portable Operating System Interface y Single Unix Specification v4
- Principios
  - Application-Oriented
  - Interface, Not Implementation
  - **Source, Not Object, Portability**
  - Minimal Interface, Minimally Defined
- Unificados en el mismo estándar
  - SUSv4 es un superset de POSIX



POSIX y SUSv4

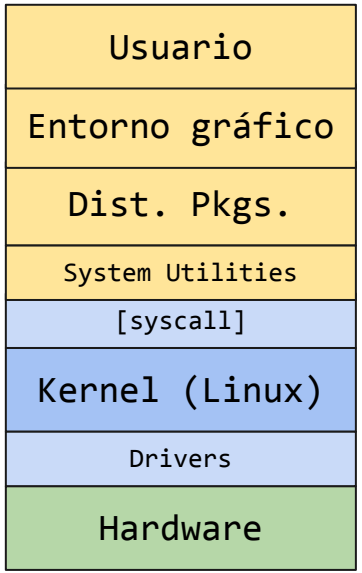
- Evolución de los estándares
- *Incorporan* el estándar de C
- *The Open Group* es el dueño del “nombre UNIX”




Periodic Table of Linux Distros
Popularity on DistroWatch
Symbol
Stable Release Year
Mi
Linux Mint
2.006
Name
Debian/Ubuntu
Arch
Red Hat/Fedora/Mandriva
Slackware
Gentoo
Other

Distribuciones basadas en Linux

- Basados en el **kernel de Linux**
  - Incluyendo utilidades en común
  - ***mayormente*** POSIX
- Diferentes aplicaciones
  - Manejadores de paquetes
  - Entornos gráficos
- Ej: Ubuntu, Debian, Fedora, RedHat, LinuxMint, y muchas más
- *If* Stallman *then* GNU/Linux



¿Por qué línea de comandos?

Pros:

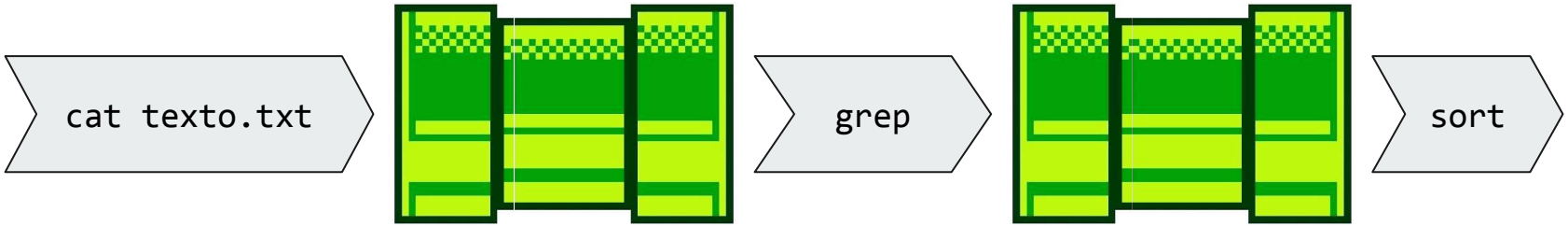
- Es **estándar** en cualquier sistema Unix-like (POSIX)
- Facilita **scripting y automatización**
- Muchas utilidades
  - *“do one thing and do it well”*
- Más **rápido** con menos recursos
- Extensible y configurable

Cons:

- Curva de aprendizaje
- No se puede hacer todo
  - Browsers, IDE

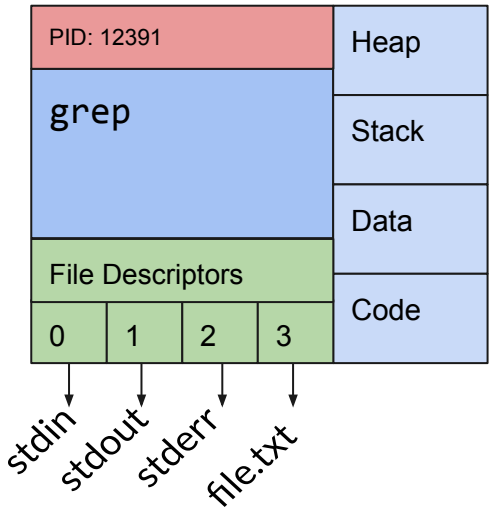
Filosofía Unix

- Utilidades **pequeñas** que realizan tareas **específicas**
- **Piping**: La **salida** de un programa es la **entrada** de otro
- Una **línea de comandos** que **crea procesos**



El proceso

- Una “instancia” de un programa
- Tiene su propia memoria
  - Heap, stack, code, etc
- Tiene un identificador único
  - PID
- Tiene un conjunto de “archivos abiertos”
  - Descriptores de archivo



¿Dónde estoy? ¿Quién soy?

`pwd`	Muestra directorio actual de trabajo
`whoami`	Muestra el usuario actual
`uname`	Muestra información del sistema operativo
`whereis`	Encuentra dónde reside un programa
`who`	Muestra quién está logueado

Navegación y archivos

`ls`	Muestra contenidos del directorio actual
`cd`	Cambia el directorio actual
`mkdir`	Crea un directorio
`touch`	Crea un archivo
`file`	Determina el tipo de un archivo
`mv`	Mueve o renombra archivos (y directorios)

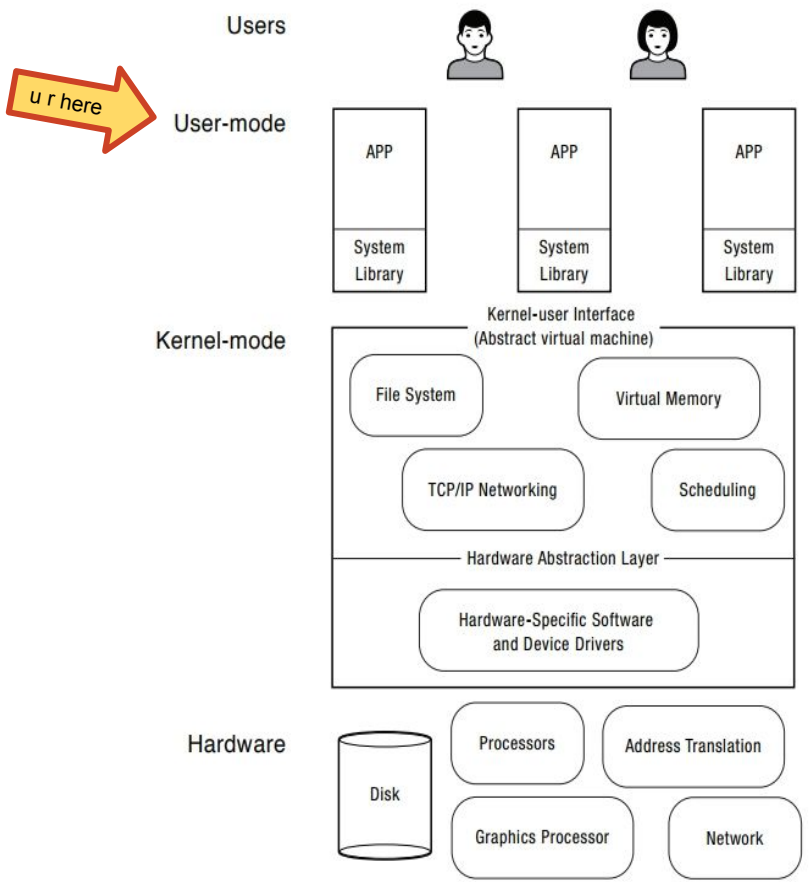
¡Ayuda!

`man`	Páginas de manual (referencia)
`info`	Páginas de info (más completas)
`whatis`	Breve descripción de lo que hace un comando
`apropos`	Búsqueda de páginas de manual por palabras clave
`history`	Muestra historial de comandos

Lab fork

Lab fork

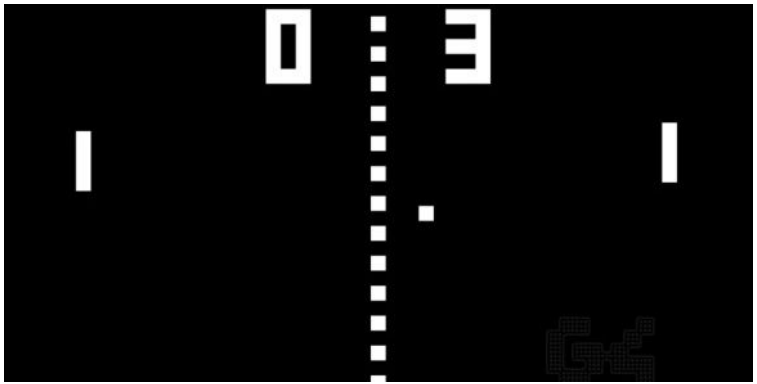
- Punto de vista de un usuario (user-mode)
- Utilizaremos **funcionalidades provistas por el kernel**



ping pong

ping-pong

- Dos procesos: padre e hijo
- Proceso padre envía un mensaje al proceso hijo
- Proceso hijo recibe y contesta
- Ambos procesos terminan



- ¿Cómo se crea un nuevo proceso?
- ¿Cómo se comunican?

fork(2)

- Crea un nuevo proceso
- Idéntico **en todo sentido** al proceso que llama a fork
- **Excepto** por el valor que **retorna fork**
- Puede fallar!

fork(2) - ejemplo 0

- ¿Qué imprime este programa?
- ¿Qué *includes* son necesarios?

```
int main(int argc, char* argv[]) {
  printf("Mi PID es: %d\n", getpid());

  int i = fork();

  if (i < 0) {
    printf("Error en fork! %d\n", i);
    exit(-1);
  }

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

  printf("Terminando\n");

  exit(0);
}
```

fork(2) - ejemplo 1

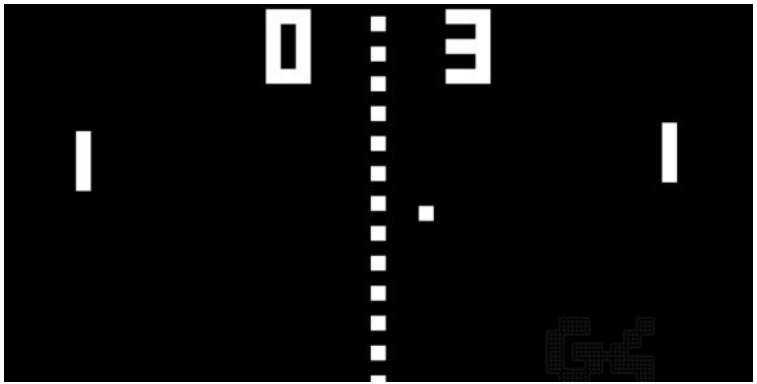
- ¿Qué imprime este programa?
- ¿Cuál es el valor de a *“a”* lo largo de la ejecución?
- ¿Y si imprimimos la dirección de la variable *“a”*?
- Ejercicio: hacer *malloc* y comparar punteros

```
int main(int argc, char* argv[]) {
  int a = 4;

  int i = fork();

  a = 5;

  if (i < 0) {
    printf("Error en fork! %d\n", i);
    exit(-1);
  }

  if (i == 0) {
    printf("[hijo] mi pid es: %d\n", getpid());
    printf("[hijo] a=%d\n", a);
  } else {
    a = 6;
    printf("[padre] mi pid es: %d\n", getpid());
    printf("[padre] a=%d\n", a);
  }

  printf("Terminando\n");
  exit(0);
}
```

ping-pong

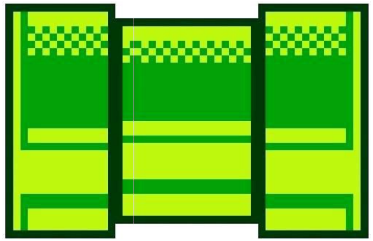
✅ Dos procesos: padre e hijo

- Proceso padre envía un mensaje al proceso hijo
- Proceso hijo recibe y contesta
- Ambos procesos terminan



- ~~¿Cómo se crea un nuevo proceso?~~
- ¿Cómo se comunican?

pipe(2)



- Crea un par de **descriptores de archivos** que están **conectados**
  - Archivo “virtual”
- El pipe es **unidireccional**
- La **lectura bloquea** hasta que haya algo que leer
- La **escritura bloquea** hasta que se pueda escribir (e.g. el pipe está lleno!)

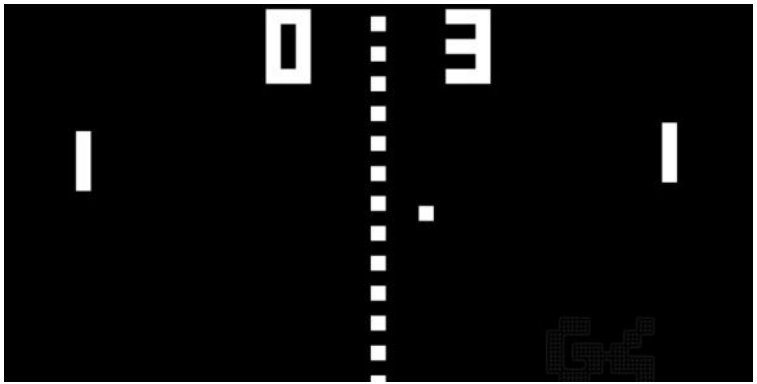
pipe(2) - ejemplo 0

- ¿Qué hace este programa?
- ¿Qué ocurrirá si descomentamos el primer *read*?
- ¿Qué valores imprime el primer *print*? ¿Serán siempre los mismos?

```
int main(int argc, char* argv[]) {
  int fds[2];
  int msg = 42;

  int r = pipe(fds);

  if (r < 0) {
    perror("Error en pipe");
    exit(-1);
  }

  printf("Lectura: %d, Escritura: %d\n", fds[0], fds[1]);

  // read(fds[0], &msg, sizeof(msg)); // ????

  // Escribo en el pipe
  write(fds[1], &msg, sizeof(msg));

  int recibido = 0;
  read(fds[0], &recibido, sizeof(recibido));
  printf("Recibi: %d\n", recibido);

  close(fds[0]);
  close(fds[1]);
}
```

ping-pong

- ✅ Dos procesos: padre e hijo
- ✅ Proceso padre envía un mensaje al proceso hijo
- ✅ Proceso hijo recibe y contesta
- ✅ Ambos procesos terminan



- ~~¿Cómo se crea un nuevo proceso?~~
- ~~¿Cómo se comunican?~~

Tarea: primes

La criba de eratóstenes - calculando primos

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

La criba de eratóstenes - calculando primos

1	**2**	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

La criba de eratóstenes - calculando primos

1	**2**	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

La criba de eratóstenes - calculando primos

1	**2**	**3**	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

La criba de eratóstenes - calculando primos

1	**2**	**3**	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

La criba de eratóstenes - calculando primos

1	**2**	**3**	4	**5**	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

La criba de eratóstenes - calculando primos

1	**2**	**3**	4	**5**	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

La criba de eratóstenes - calculando primos

1	**2**	**3**	4	**5**	6	**7**	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

La criba de eratóstenes - calculando primos

1	**2**	**3**	4	**5**	6	**7**	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

La criba de eratóstenes - calculando primos

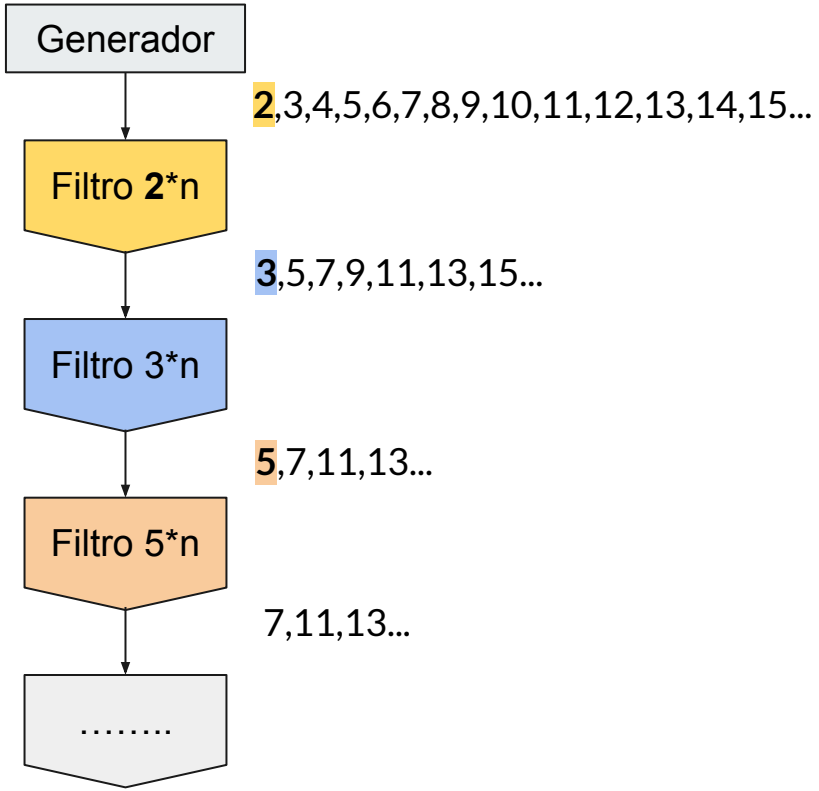
1	**2**	**3**	4	**5**	6	**7**	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

La criba de eratóstenes - calculando primos

1	**2**	**3**	4	**5**	6	**7**	8	9	10
11	12	**13**	14	15	16	**17**	18	**19**	20
21	22	**23**	24	25	26	27	28	**29**	30
31	32	33	34	35	36	**37**	38	39	40
41	42	**43**	44	45	46	**47**	48	49	50
51	52	**53**	54	55	56	57	58	**59**	60
61	62	63	64	65	66	**67**	68	69	70

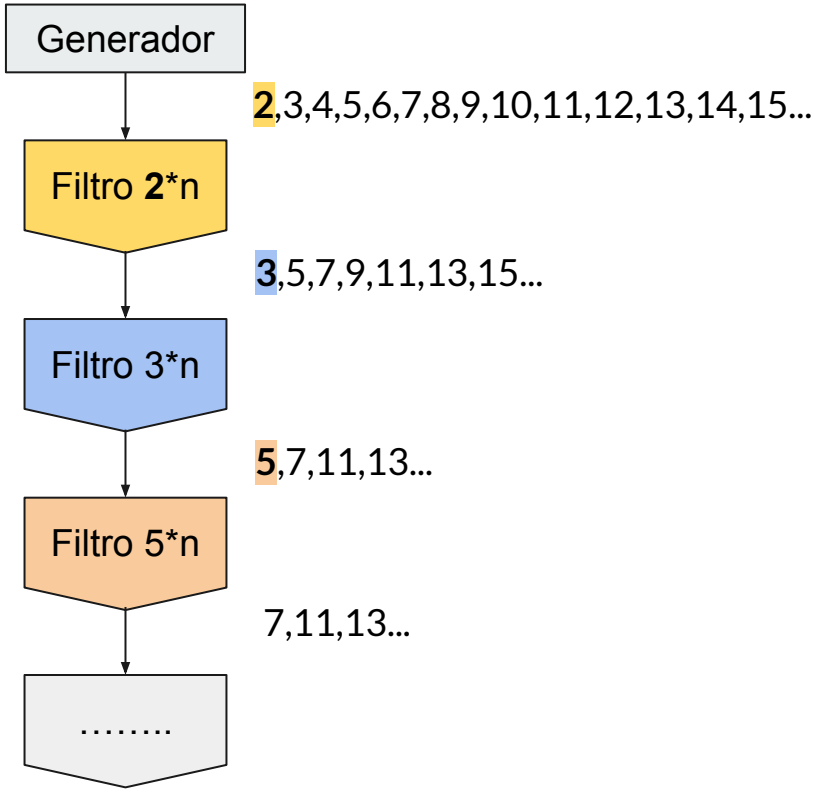
Tarea: primes

- Imprime los números primos menores o iguales a N
- El primer proceso genera una lista de números consecutivos
  - Se los envía a un proceso "**filtro**"
- Cada **filtro** toma el primer valor que recibe, y filtra los múltiplos



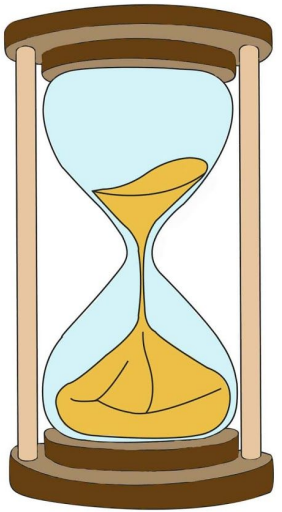
Tarea: primes

- ¿Cómo esperar a que un proceso termine?
- **Tiene que escalar con el valor de N y no leakear file descriptors**



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) - huérfanos y zombies

- ¿Qué pasa si el proceso padre termina *sin hacer wait* a un proceso hijo?

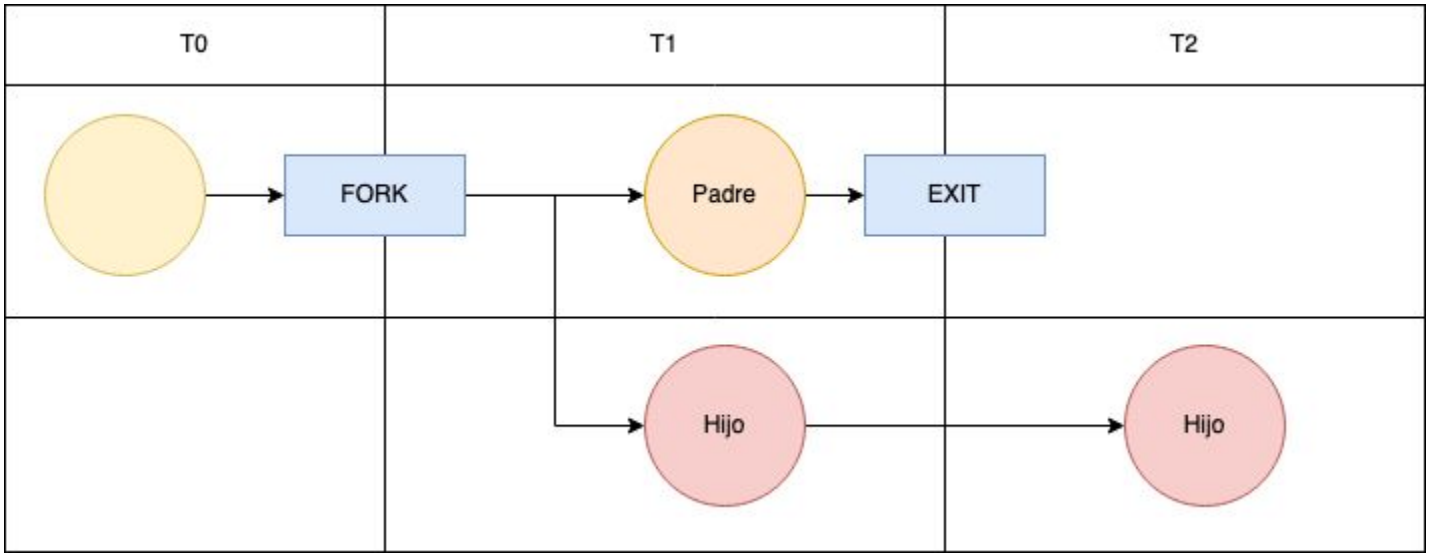
- ¿Qué pasa si el proceso hijo termina, pero el proceso padre no hace *wait*?

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.)

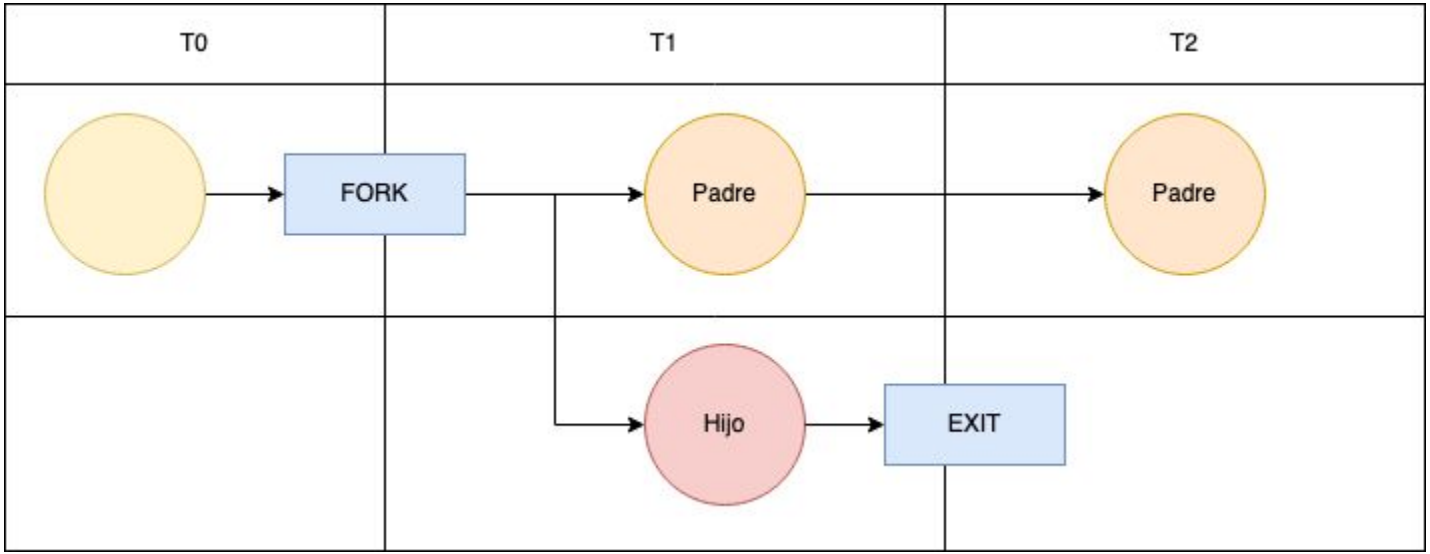


wait(2) - zombies

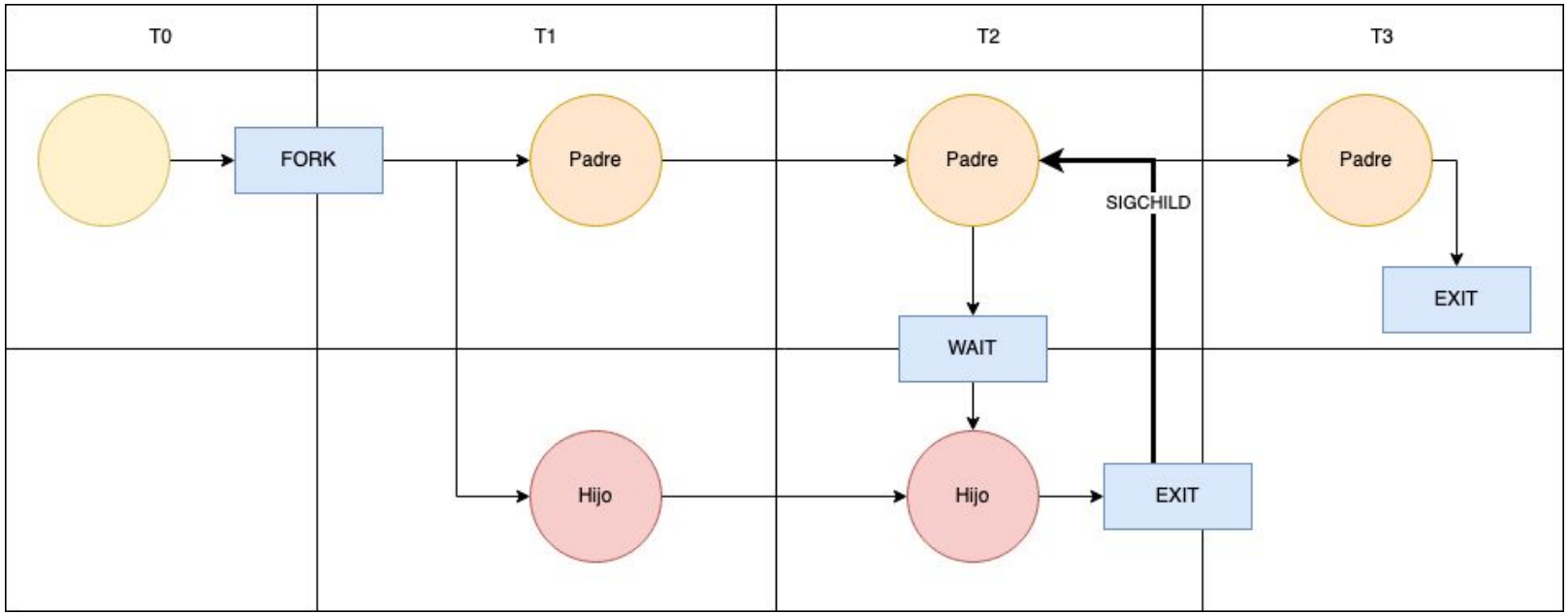
- ¿Qué pasa si el proceso hijo termina, pero el proceso padre no hace *wait*?
- ¿Sigue existiendo el proceso? ¿Cómo se ve desde **ps**?
- ¿Qué pasa con los *zombies huérfanos*?

```
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.)



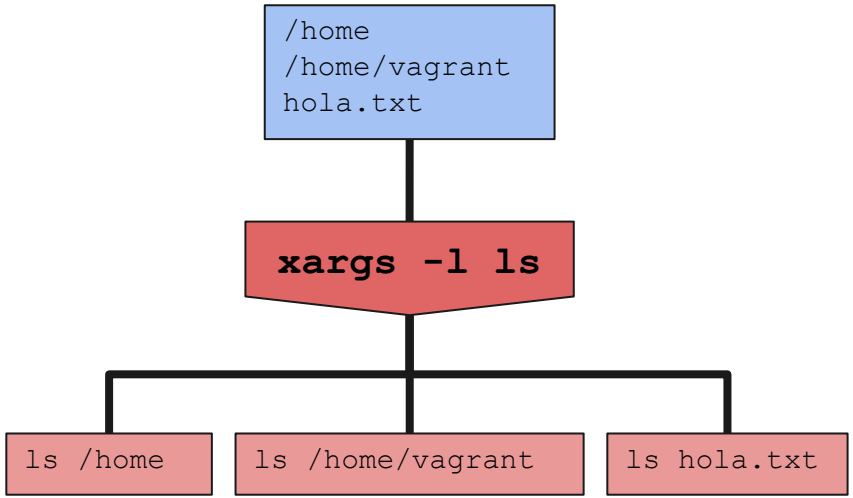
wait(2) - flujo normal (diag.)



Tarea: xargs

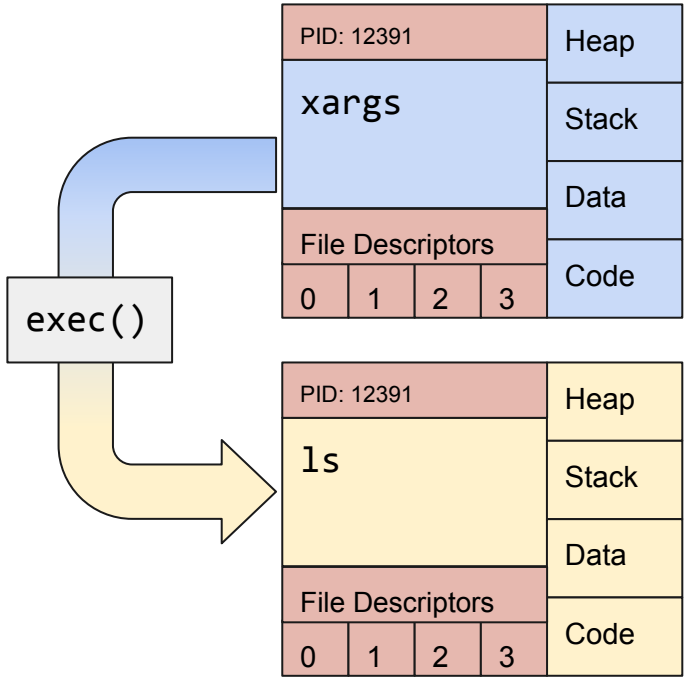
Tarea: xargs

- 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)

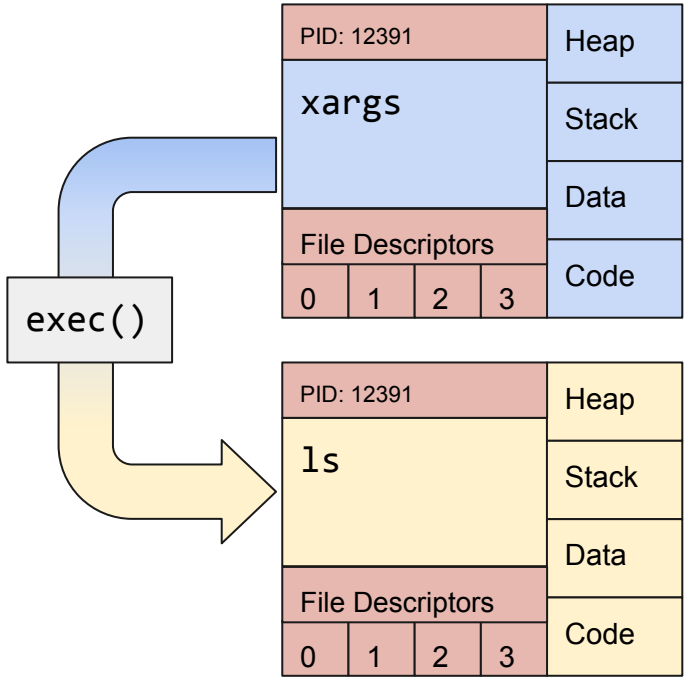
- Cambia **la imagen** de un proceso
  - Memoria virtual, entorno, argumentos
- Mantiene todo lo demás
  - fd, pid, ppid, etc
- Única **syscall**



Familia de exec(3)

execl, execlp, execle, execv, execvp, execvpe

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



execvp(3) - ejemplos 0 y 1

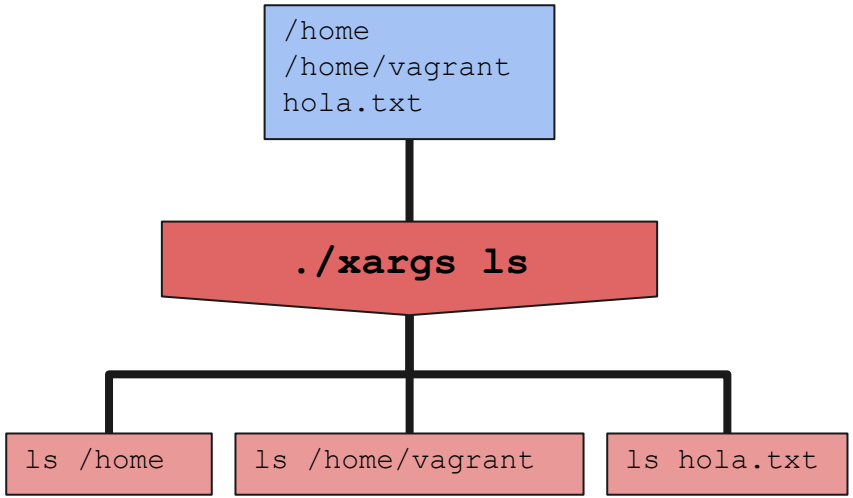
- ¿Qué hacen estos programas?
- ¿Se ejecuta en algún momento el print “Terminado”?

```
int main(int argc, char* argv[]) {
	char *args[] = {"echo", "hello world!", NULL};
	execvp("echo", args);

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

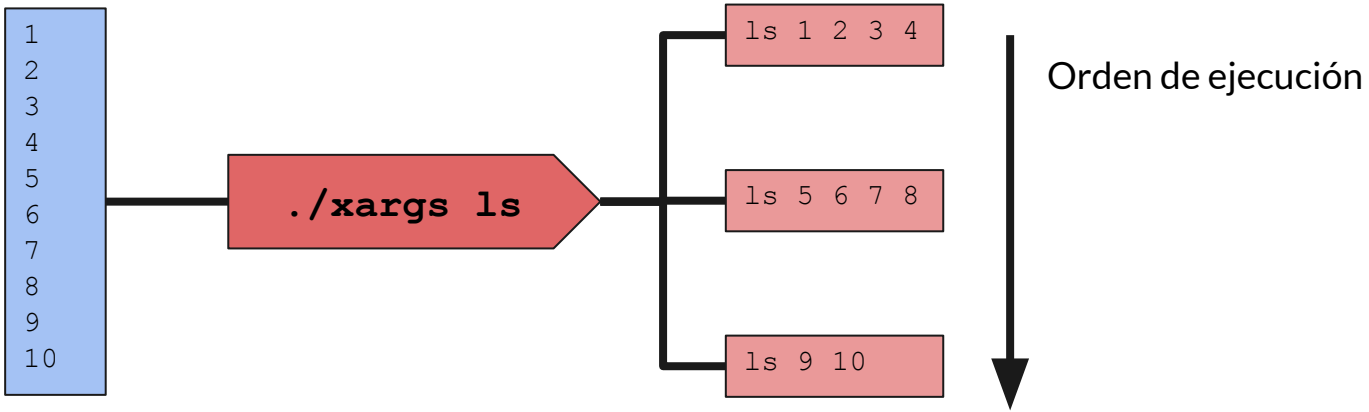
```
int main(int argc, char* argv[]) {
	// Notar que argv+1 es lo mismo que
	// {argv[1], argv[2], ..., argv[argc-1], NULL}
	execvp(argv[1], argv+1);

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

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*



Repositorio (git/github)

¿Cómo usar el repositorio?

- Para el lab
- La página de entregas explica integraciones
- Crear un *Pull Request* con todos los cambios

```
// 0) Clonar en un directorio local (e.g. fork)
$ git clone git@github.com:fiubatps/sisop_2022b_jfresia fork
$ cd fork

// 1) Agregar el repositorio remoto con elesqueleto
$ git remote add catedra https://github.com/fisop/fork

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

// 3) Integración del "esqueleto"
$ git fetch --all
$ git checkout base_fork
$ git merge catedra/main --allow-unrelated-histories
$ git push origin base_fork

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

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

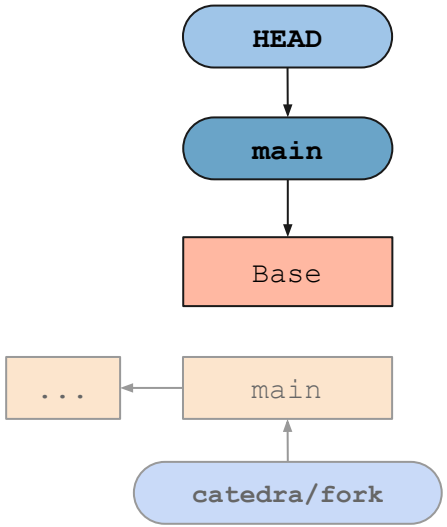
¿Cómo crear el PR?

- Se debe crear un PR
  - **base_fork** como base
  - **entrega_fork** como compare
- 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
```

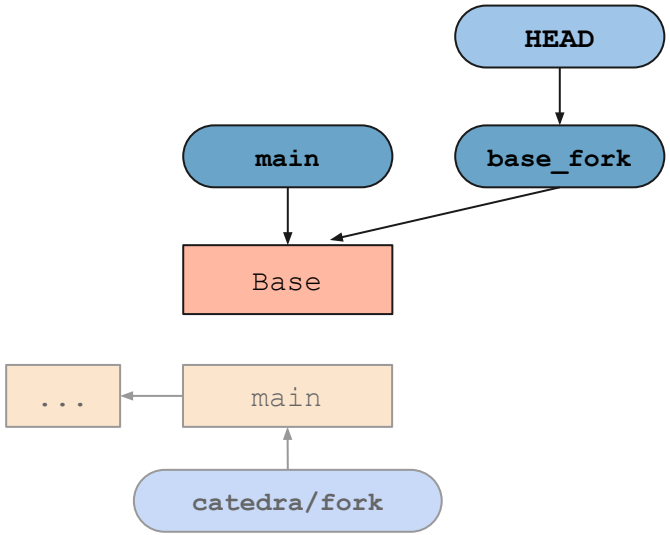
Ejemplo de historia de git: (1)

git clone ...
git remote add ...



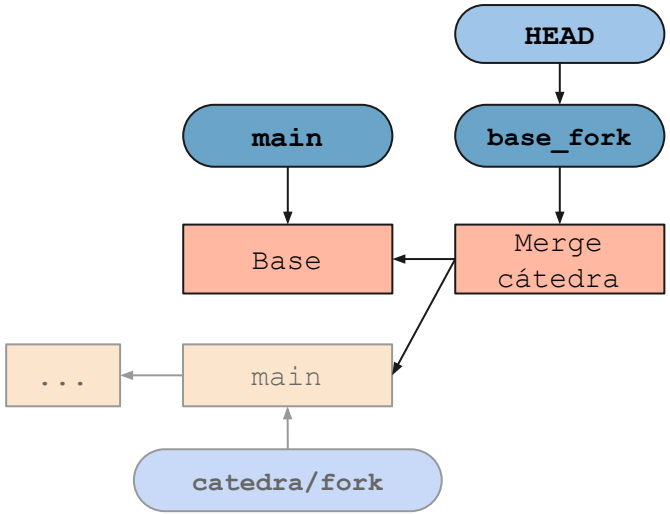
Ejemplo de historia de git: (2)

git checkout -b base_fork
git push -u origin base_fork



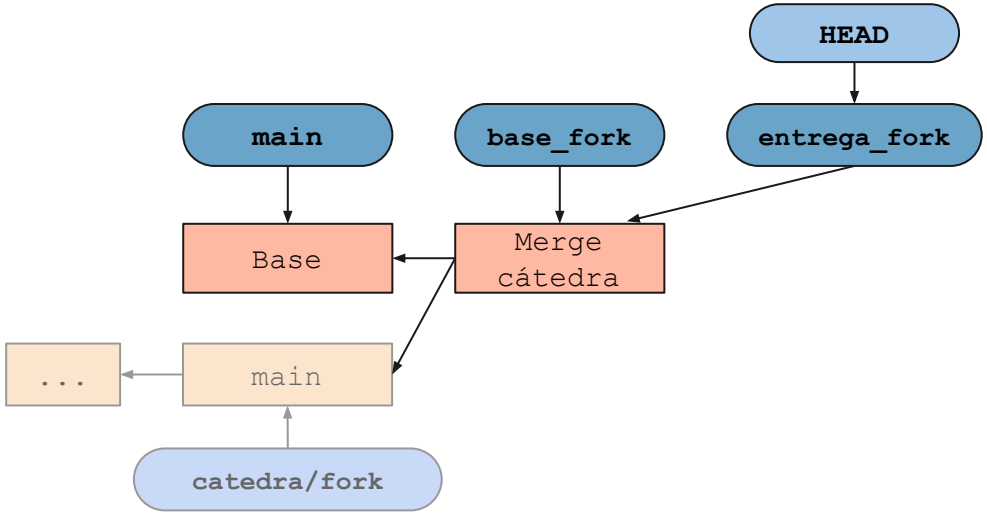
Ejemplo de historia de git: (3)

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



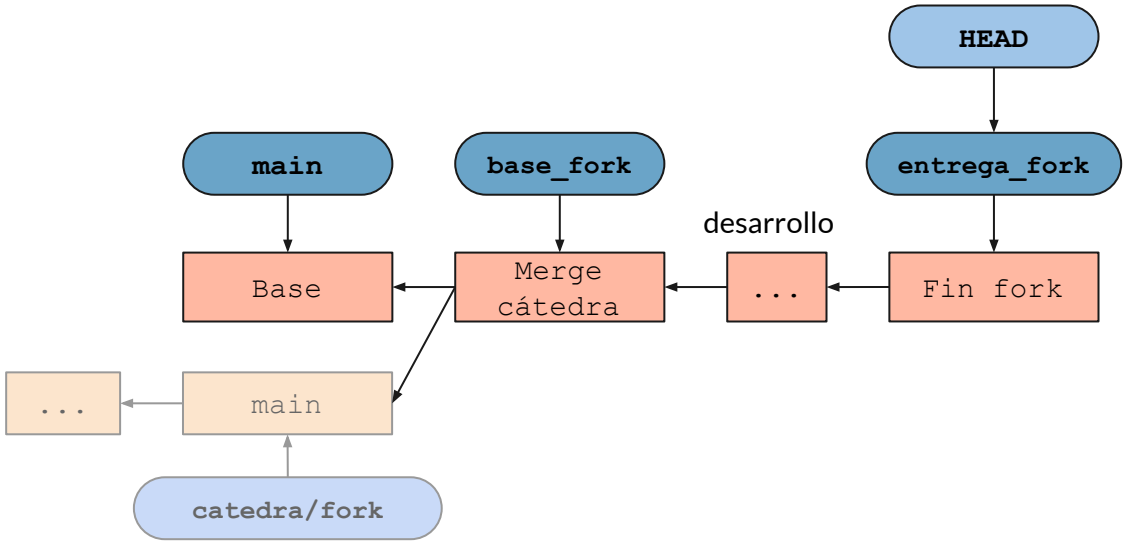
Ejemplo de historia de git: (4)

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



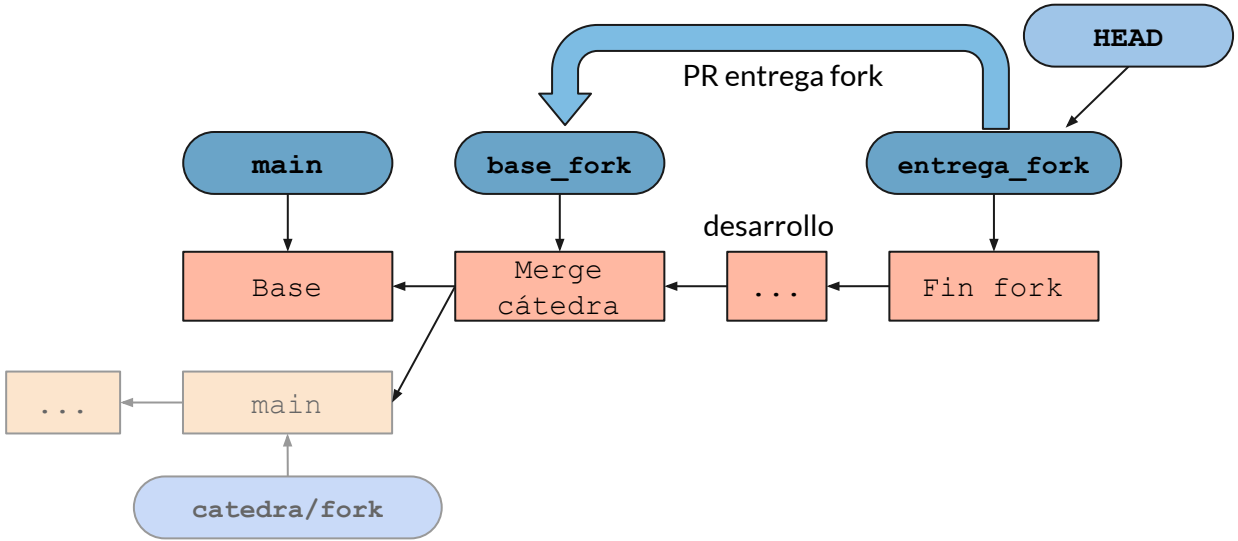
Ejemplo de historia de git: (5)

git add ...
git commit ...
git push origin entrega_fork



Ejemplo de historia de git: (6)

PR desde UI github
entrega_fork -> base_fork



Ejemplos

fork(2) - ejemplo 2

- ¿Qué hace este programa?
- ¿Queda algún archivo abierto?
  - ¿Se “filtra” algún file descriptor?
- ¿Podría el proceso padre escribir luego de que el proceso hijo llame a *close*?

```
int main(int argc, char* argv[]) {
  char* msg = "fisop\n";

  // Abro un archivo y si no existe lo creo
  int fd = open("hola.txt", O_CREAT | O_RDWR, 0644);
  int i = fork();

  if (i < 0) {
    printf("Error en fork! %d\n", i);
    exit(-1);
  }

  if (i == 0) {
    printf("[hijo] mi pid es: %d\n", getpid());
    write(fd, msg, 6);
    close(fd);
  } else {
    printf("[padre] mi pid es: %d\n", getpid());
  }

  printf("Terminando\n");
  exit(0);
}
```

fork(2) - ejemplo 3

- ¿Cuántos procesos en total genera este código?
- ¿Fallará **fork()** en algún momento?

```
int main(int argc, char* argv[]) {
  printf("Mi PID es: %d\n", getpid());

  for (int i = 0; i < 12; i++) {
    int r = fork();

    if (r < 0) {
      perror("Error en fork");
      exit(-1);
    }

    printf("[%d] Hola!\n", getpid());
  }

  printf("Terminando\n");
  exit(0);
}
```

pipe(2) - ejemplo 1

- ¿Qué hace este programa?
- ¿Por qué se cierran algunos fds?
- ¿Qué ocurre si el proceso hijo llega a llamar a *read* antes que el padre llame a *write*?
- ¿Y al revés?

```
int main(int argc, char* argv[]) {
  int fds[2];
  int msg = 42;

  pipe(fds);
  int i = fork();

  if (i == 0) {
    printf("[hijo] mi pid es: %d\n", getpid());
    // El hijo no va a escribir
    close(fds[1]);

    int recv = 0;
    read(fds[0], &recv, sizeof(recv));
    printf("[hijo] lei: %d\n", recv);

    close(fds[0]);
  } else {
    printf("[padre] mi pid es: %d\n", getpid());
    // El padre no va a leer
    close(fds[0]);

    // Esperamos dos segundos, el hijo no debería seguir
    sleep(2);
    write(fds[1], &msg, sizeof(msg));

    close(fds[1]);
  }
}
```

pipe(2) - ejemplo 2

- ¿Qué hace este programa?
- ¿Termina el algún momento? ¿Cómo?
- ¿Que contiene **pipe(7)**?

```
int main(int argc, char* argv[]) {
  int fds[2];

  pipe(fds);

  printf("Lectura: %d, Escritura: %d\n", fds[0], fds[1]);

  int msg = 42;
  int escritos = 0;
  while (1) {
    r = write(fds[1], &msg, sizeof(msg));
    if (r >= 0) {
      printf("Total escrito: %d\n", r, escritos);
      escritos += sizeof(msg);
    } else {
      printf("write fallo con %d\n", r);
      printf("errno was: %d\n", errno);
      perror("perror en write");
      break;
    }
  };

  close(fds[0]);
  close(fds[1]);
}
```

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 terminócon %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");
  }
}
```

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());
}
```

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(2)***: para la lista completa

Más recursos

- Páginas de manual (`man`)
- *The Linux Programming Interface*
  de Michael Kerrisk
- [The missing semester]
  Mastering the tools
- [Linux Journey]