

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

API de procesos y ejecución directa limitada

Crear procesos y recuperar la CPU

Gerson Garrido

Semana 2

Hoy

1. `fork()`: duplicar un proceso
2. `wait()` y `exec()`: esperar y reemplazar
3. La shell: redirección, pipes y señales
4. Ejecución directa limitada: modos, traps, temporizador y cambio de contexto
5. El camino completo de una llamada al sistema en xv6-riscv

> **Lectura**
>
> OSTEP cap. 5 *Interlude: Process API* y cap. 6 *Mechanism: Limited Direct Execution*.
> Apuntes del curso: capítulo 2.

SECCIÓN 01

Duplicar: fork

01



Tres llamadas para manejar procesos

Llamada	¿Crea un proceso?	¿Cambia el programa?	Retorna
`fork()`	Sí	No	Dos veces

Tres llamadas para manejar procesos

Llamada	¿Crea un proceso?	¿Cambia el programa?	Retorna
`fork()`	Sí	No	Dos veces
`exec()`	No	Sí	Nunca, salvo error

Tres llamadas para manejar procesos

Llamada	¿Crea un proceso?	¿Cambia el programa?	Retorna
`fork()`	Sí	No	Dos veces
`exec()`	No	Sí	Nunca, salvo error
`wait()`	No	No	Cuando termina un hijo

Tres llamadas para manejar procesos

Llamada	¿Crea un proceso?	¿Cambia el programa?	Retorna
`fork()`	Sí	No	Dos veces
`exec()`	No	Sí	Nunca, salvo error
`wait()`	No	No	Cuando termina un hijo

Con estas tres se construye la shell

La que usan todos los días y la que van a escribir en el proyecto **P1** (`wish`).

`fork()`: se llama una vez, retorna dos veces

fork.c

```c
printf("hola, soy %d\n", getpid());

pid_t rc = fork();
if (rc < 0) {
  perror("fork");
  exit(1);
} else if (rc == 0) {
  printf("hijo: mi PID es %d, mi padre es %d\n",
         getpid(), getppid());
} else {
  printf("padre: creé al hijo %d\n", rc);
}
```

bash

```
$ ./fork
hola, soy 4127
padre: creé al hijo 4128
hijo: mi PID es 4128,
      mi padre es 4127
```

`fork()`: se llama una vez, retorna dos veces

fork.c

```c
printf("hola, soy %d\n", getpid());

pid_t rc = fork();
if (rc < 0) {
  perror("fork");
  exit(1);
} else if (rc == 0) {
  printf("hijo: mi PID es %d, mi padre es %d\n",
         getpid(), getppid());
} else {
  printf("padre: creé al hijo %d\n", rc);
}
```

bash

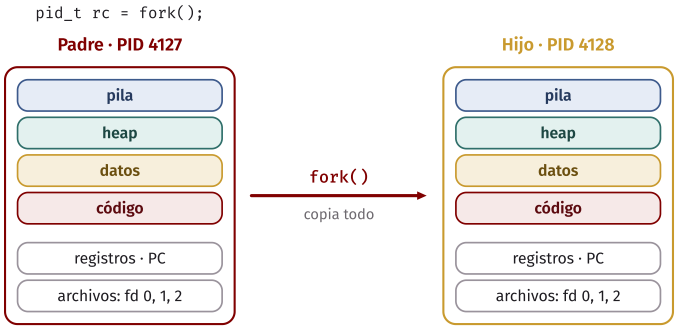
```
$ ./fork
hola, soy 4127
padre: creé al hijo 4128
hijo: mi PID es 4128,
      mi padre es 4127
```

- Padre: `rc` = PID del hijo.
- Hijo: `rc` = `0`.
- Error: `rc` = `-1`, no hay hijo.

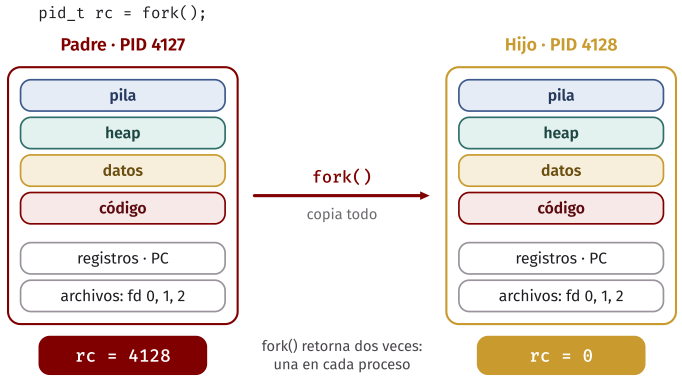
Qué copia `fork()`



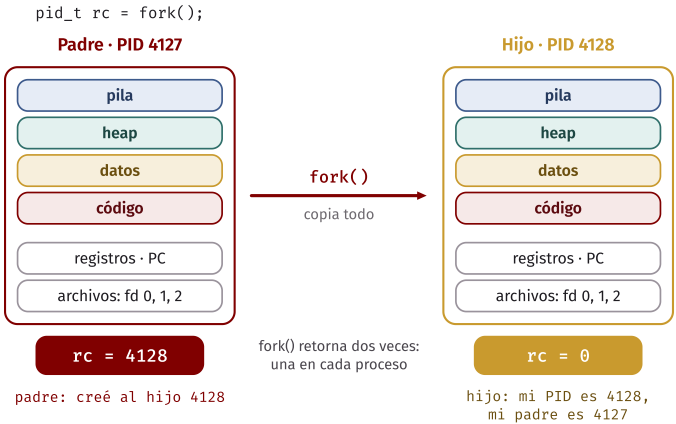
Qué copia `fork()`



Qué copia `fork()`



Qué copia `fork()`



En xv6: `kfork()` en `kernel/proc.c` (resumido)

kernel/proc.c

```
1  int kfork(void) {
2    if ((np = allocproc()) == 0)
3      return -1;
4    uvmcopy(p->pagetable, np->pagetable, p->sz);
5    *(np->trapframe) = *(p->trapframe);
6    np->trapframe->a0 = 0;   // 0 en el hijo
7    for (i = 0; i < NOFILE; i++)
8      if (p->ofile[i])
9        np->ofile[i] = filedup(p->ofile[i]);
10   // ...
11   np->parent = p;
12   np->state = RUNNABLE;
13   return pid;              // PID al padre
14 }
```

En xv6: `kfork()` en `kernel/proc.c` (resumido)

kernel/proc.c

```
1  int kfork(void) {
2    if ((np = allocproc()) == 0)
3      return -1;
4    uvmcopy(p->pagetable, np->pagetable, p->sz);
5    *(np->trapframe) = *(p->trapframe);
6    np->trapframe->a0 = 0;   // 0 en el hijo
7    for (i = 0; i < NOFILE; i++)
8      if (p->ofile[i])
9        np->ofile[i] = filedup(p->ofile[i]);
10   // ...
11   np->parent = p;
12   np->state = RUNNABLE;
13   return pid;              // PID al padre
14 }
```

- `uvmcopy`: copia la memoria del padre.

En xv6: `kfork()` en `kernel/proc.c` (resumido)

kernel/proc.c

```
1  int kfork(void) {
2    if ((np = allocproc()) == 0)
3      return -1;
4    uvmcopy(p->pagetable, np->pagetable, p->sz);
5    *(np->trapframe) = *(p->trapframe);
6    np->trapframe->a0 = 0;   // 0 en el hijo
7    for (i = 0; i < NOFILE; i++)
8      if (p->ofile[i])
9        np->ofile[i] = filedup(p->ofile[i]);
10   // ...
11   np->parent = p;
12   np->state = RUNNABLE;
13   return pid;              // PID al padre
14 }
```

- `uvmcopy`: copia la memoria del padre.
- Copia el `trapframe`: los registros de usuario, PC incluido.

En xv6: `kfork()` en `kernel/proc.c` (resumido)

kernel/proc.c

```
1  int kfork(void) {
2    if ((np = allocproc()) == 0)
3      return -1;
4    uvmcopy(p->pagetable, np->pagetable, p->sz);
5    *(np->trapframe) = *(p->trapframe);
6    np->trapframe->a0 = 0;   // 0 en el hijo
7    for (i = 0; i < NOFILE; i++)
8      if (p->ofile[i])
9        np->ofile[i] = filedup(p->ofile[i]);
10   // ...
11   np->parent = p;
12   np->state = RUNNABLE;
13   return pid;              // PID al padre
14 }
```

- `uvmcopy`: copia la memoria del padre.
- Copia el `trapframe`: los registros de usuario, PC incluido.
- `a0` es el registro de retorno en RISC-V: por eso el hijo ve `0`.

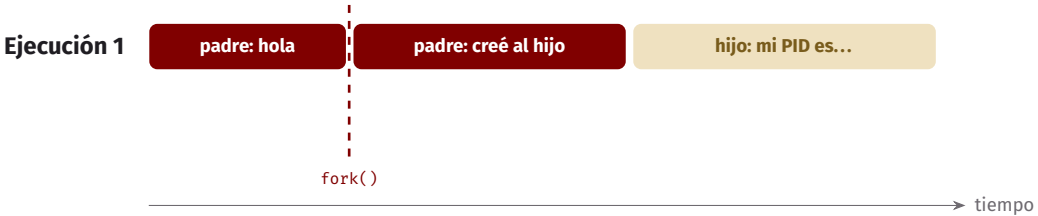
En xv6: `kfork()` en `kernel/proc.c` (resumido)

kernel/proc.c

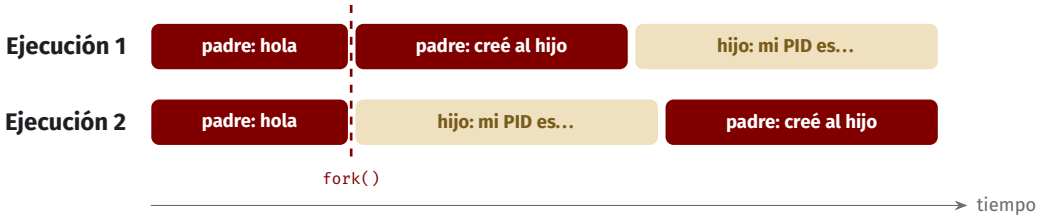
```
1  int kfork(void) {
2    if ((np = allocproc()) == 0)
3      return -1;
4    uvmcopy(p->pagetable, np->pagetable, p->sz);
5    *(np->trapframe) = *(p->trapframe);
6    np->trapframe->a0 = 0;   // 0 en el hijo
7    for (i = 0; i < NOFILE; i++)
8      if (p->ofile[i])
9        np->ofile[i] = filedup(p->ofile[i]);
10   // ...
11   np->parent = p;
12   np->state = RUNNABLE;
13   return pid;              // PID al padre
14 }
```

- `uvmcopy`: copia la memoria del padre.
- Copia el `trapframe`: los registros de usuario, PC incluido.
- `a0` es el registro de retorno en RISC-V: por eso el hijo ve `0`.
- El hijo queda `RUNNABLE`: listo, esperando al planificador.

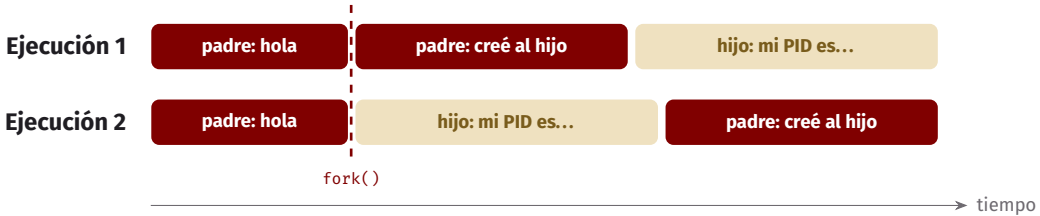
¿Quién imprime primero?



¿Quién imprime primero?



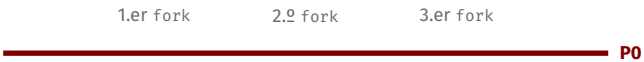
¿Quién imprime primero?



No hay un orden garantizado

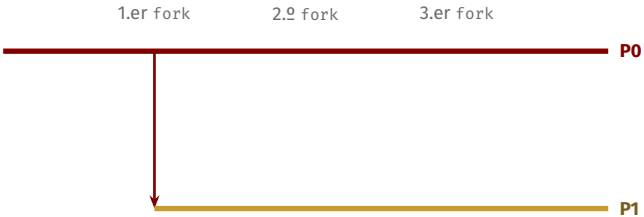
Después de `fork()` hay dos procesos *listos* y decide el planificador. Con varios núcleos pueden correr al mismo tiempo. Primer encuentro con el **no determinismo**.

¿Cuántos procesos crea `fork(); fork(); fork();`?



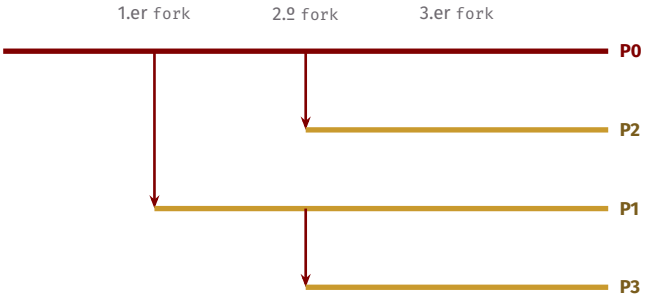
Cada `fork` duplica *a todos* los procesos que existen en ese momento.

¿Cuántos procesos crea `fork(); fork(); fork();`?



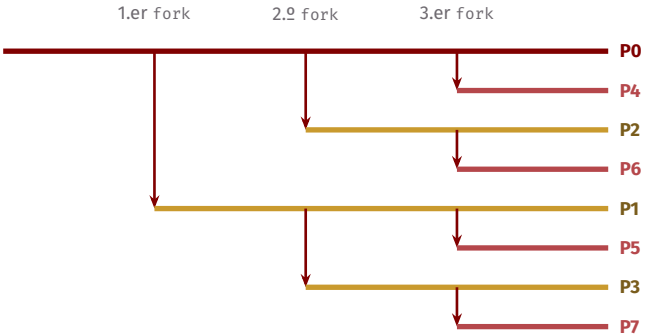
Tras el 1.er `fork`: **2** procesos.

¿Cuántos procesos crea `fork(); fork(); fork();`?



Tras el 2.º: **4**. P0 y P1 vuelven a hacer `fork`.

¿Cuántos procesos crea `fork(); fork(); fork();`?



Tras el 3.^{er}: $2^3 = \mathbf{8}$ procesos, 7 de ellos nuevos.

Un misterio: “hola” aparece dos veces

en una terminal

```
$ ./fork
hola, soy 28
padre: creé al hijo 29
hijo: mi PID es 29, ...
```

salida a un pipe

```
$ ./fork | cat
hola, soy 28
hijo: mi PID es 29, ...
hola, soy 28
padre: creé al hijo 29
```

Un misterio: “hola” aparece dos veces

en una terminal

```
$ ./fork
hola, soy 28
padre: creé al hijo 29
hijo: mi PID es 29, ...
```

salida a un pipe

```
$ ./fork | cat
hola, soy 28
hijo: mi PID es 29, ...
hola, soy 28
padre: creé al hijo 29
```

El `printf` está *antes* del `fork()`

Solo el padre lo ejecuta. ¿Cómo puede salir dos veces? La respuesta no está en el kernel, sino en la biblioteca de C.

La explicación: el buffer de `stdout`



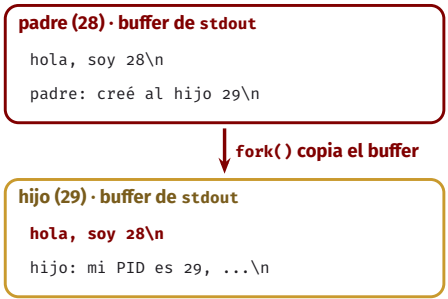
`printf` no llama a `write` de inmediato: guarda el texto en un buffer del proceso. Hacia un pipe, el buffer se vacía cuando se llena o al terminar.

La explicación: el buffer de `stdout`



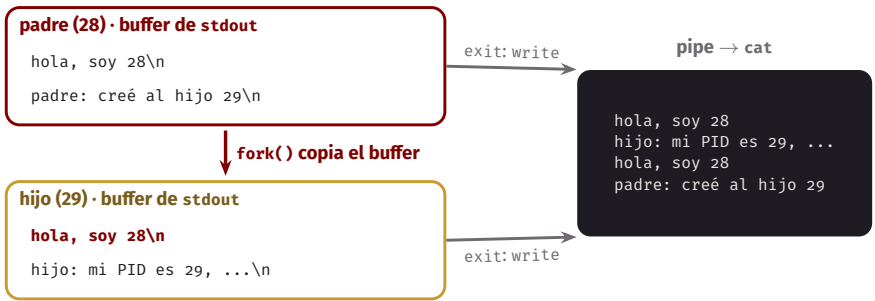
`fork()` copia *toda* la memoria, buffer incluido: el hijo hereda el texto pendiente.

La explicación: el buffer de `stdout`



Cada proceso agrega su propia línea al buffer.

La explicación: el buffer de `stdout`



Al salir, los dos vacían su buffer. En una terminal se vacía en cada `\n`: por eso allí no pasa.

Solución: `fflush(stdout)` antes de `fork()`.

SECCIÓN 02

wait y exec

02



`wait()`: el padre espera al hijo

wait.c

```c
pid_t rc = fork();
if (rc == 0) {
  printf("hijo %d trabajando...\n", getpid());
  sleep(1);
  exit(42);
}

int status;
pid_t done = waitpid(rc, &status, 0);
if (WIFEXITED(status))
  printf("padre: el hijo %d terminó con código %d\n",
         done, WEXITSTATUS(status));
```

bash

```
$ ./wait
hijo 34 trabajando...
padre: el hijo 34
terminó con código 42
```

`wait()`: el padre espera al hijo

wait.c

```c
pid_t rc = fork();
if (rc == 0) {
  printf("hijo %d trabajando...\n", getpid());
  sleep(1);
  exit(42);
}

int status;
pid_t done = waitpid(rc, &status, 0);
if (WIFEXITED(status))
  printf("padre: el hijo %d terminó con código %d\n",
         done, WEXITSTATUS(status));
```

bash

```
$ ./wait
hijo 34 trabajando...
padre: el hijo 34
terminó con código 42
```

- Ahora la salida es determinista.
- `status` empaqueta varios datos: se lee con `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`...

Zombis y huérfanos

Zombi *(zombie)*

padre
vivo, sin `wait`

Zombis y huérfanos



Zombis y huérfanos



Ya no ejecuta, pero ocupa su entrada en la tabla de procesos hasta que el padre llame a `wait()`.

Zombis y huérfanos

Zombi *(zombie)*



Ya no ejecuta, pero ocupa su entrada en la tabla de procesos hasta que el padre llame a `wait()`.

Huérfano *(orphan)*



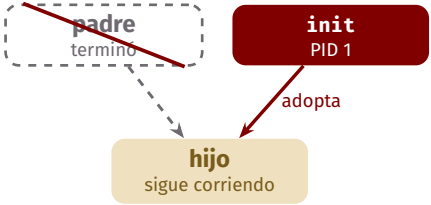
Zombis y huérfanos

Zombi *(zombie)*



Ya no ejecuta, pero ocupa su entrada en la tabla de procesos hasta que el padre llame a `wait()`.

Huérfano *(orphan)*



`init` lo adopta y hace `wait()` por él. En xv6: `reparent()` en `kernel/proc.c`.

exec( ): reemplazar el programa

exec.c

```c
pid_t rc = fork();
if (rc == 0) {
  char *args[] = {"wc", "-l", (char *)file, NULL};
  execvp(args[0], args);
  perror("execvp");   // solo si exec falla
  exit(1);
}

waitpid(rc, NULL, 0);
printf("padre: wc terminó\n");
```

bash

```
$ ./exec
25 exec.c
padre: wc terminó
```

exec( ): reemplazar el programa

exec.c

```c
pid_t rc = fork();
if (rc == 0) {
  char *args[] = {"wc", "-l", (char *)file, NULL};
  execvp(args[0], args);
  perror("execvp");   // solo si exec falla
  exit(1);
}

waitpid(rc, NULL, 0);
printf("padre: wc terminó\n");
```

bash

```
$ ./exec
25 exec.c
padre: wc terminó
```

- Familia: `execl`, `execlp`, `execv`, `execvp`, `execve`... Cambia solo cómo se pasan los argumentos.

exec( ): reemplazar el programa

exec.c

```c
pid_t rc = fork();
if (rc == 0) {
  char *args[] = {"wc", "-l", (char *)file, NULL};
  execvp(args[0], args);
  perror("execvp");   // solo si exec falla
  exit(1);
}

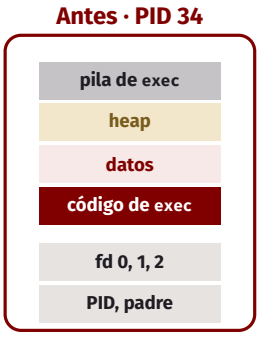
waitpid(rc, NULL, 0);
printf("padre: wc terminó\n");
```

bash

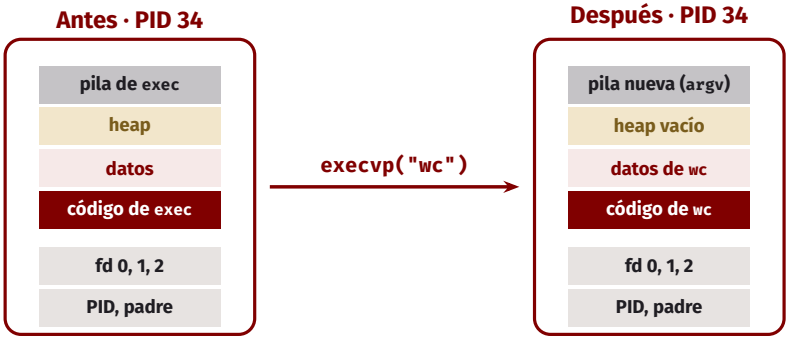
```
$ ./exec
25 exec.c
padre: wc terminó
```

- Familia: `execl`, `execlp`, `execv`, `execvp`, `execve`… Cambia solo cómo se pasan los argumentos.
- Si funciona, nunca retorna: el código que la llamó ya no existe.

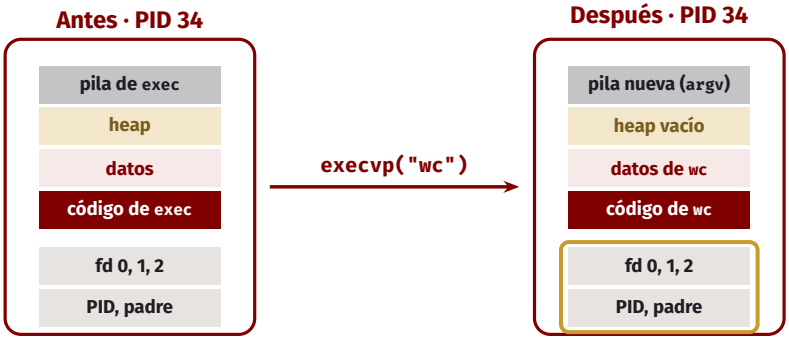
Qué conserva y qué reemplaza `exec()`



Qué conserva y qué reemplaza `exec()`



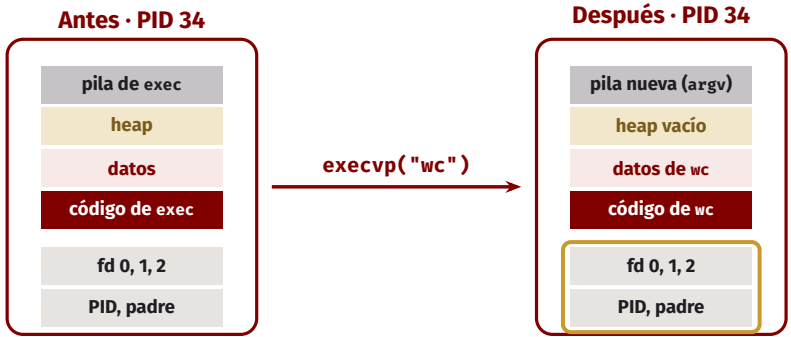
Qué conserva y qué reemplaza `exec()`



Se reemplaza: todo el espacio de direcciones.

Se conservan: el PID y los descriptores abiertos.

Qué conserva y qué reemplaza `exec()`



Se reemplaza: todo el espacio de direcciones.

Se conservan: el PID y los descriptores abiertos.

`exec()` **no crea** un proceso: transforma el que ya existe.

SECCIÓN 03

La shell y las señales

03



Por qué `fork` y `exec` están separados

El ciclo de una shell:

1. Leer una línea y separarla en palabras.

user/sh.c (xv6)

```
1  while (getcmd(buf, sizeof(buf)) >= 0) {
2    // ...
3    if (cmd[0] == 'c' && cmd[1] == 'd' &&
4        cmd[2] == ' ') {
5      // cd lo debe ejecutar el padre
6      if (chdir(cmd + 3) < 0)
7        fprintf(2, "cannot cd %s\n", cmd + 3);
8    } else {
9      if (fork1() == 0)
10       runcmd(parsecmd(cmd));
11     wait(0);
12   }
13 }
```

Por qué `fork` y `exec` están separados

El ciclo de una shell:

1. Leer una línea y separarla en palabras.
2. `fork()`.

user/sh.c (xv6)

```
1  while (getcmd(buf, sizeof(buf)) >= 0) {
2    // ...
3    if (cmd[0] == 'c' && cmd[1] == 'd' &&
4        cmd[2] == ' ') {
5      // cd lo debe ejecutar el padre
6      if (chdir(cmd + 3) < 0)
7        fprintf(2, "cannot cd %s\n", cmd + 3);
8    } else {
9      if (fork1() == 0)
10       runcmd(parsecmd(cmd));
11     wait(0);
12   }
13 }
```

Por qué `fork` y `exec` están separados

El ciclo de una shell:

1. Leer una línea y separarla en palabras.
2. `fork()`.
3. Hijo: preparar redirecciones *(redirection)*, luego `exec()`.

user/sh.c (xv6)

```
1  while (getcmd(buf, sizeof(buf)) >= 0) {
2    // ...
3    if (cmd[0] == 'c' && cmd[1] == 'd' &&
4        cmd[2] == ' ') {
5      // cd lo debe ejecutar el padre
6      if (chdir(cmd + 3) < 0)
7        fprintf(2, "cannot cd %s\n", cmd + 3);
8    } else {
9      if (fork1() == 0)
10       runcmd(parsecmd(cmd));
11     wait(0);
12   }
13 }
```

Por qué `fork` y `exec` están separados

El ciclo de una shell:

1. Leer una línea y separarla en palabras.
2. `fork()`.
3. Hijo: preparar redirecciones *(redirection)*, luego `exec()`.
4. Padre: `wait()` al hijo (salvo con `&`).

user/sh.c (xv6)

```
1  while (getcmd(buf, sizeof(buf)) >= 0) {
2    // ...
3    if (cmd[0] == 'c' && cmd[1] == 'd' &&
4        cmd[2] == ' ') {
5      // cd lo debe ejecutar el padre
6      if (chdir(cmd + 3) < 0)
7        fprintf(2, "cannot cd %s\n", cmd + 3);
8    } else {
9      if (fork1() == 0)
10       runcmd(parsecmd(cmd));
11     wait(0);
12   }
13 }
```

Por qué `fork` y `exec` están separados

El ciclo de una shell:

1. Leer una línea y separarla en palabras.
2. `fork()`.
3. Hijo: preparar redirecciones *(redirection)*, luego `exec()`.
4. Padre: `wait()` al hijo (salvo con `&`).
5. Volver al paso 1.

user/sh.c (xv6)

```
1  while (getcmd(buf, sizeof(buf)) >= 0) {
2    // ...
3    if (cmd[0] == 'c' && cmd[1] == 'd' &&
4        cmd[2] == ' ') {
5      // cd lo debe ejecutar el padre
6      if (chdir(cmd + 3) < 0)
7        fprintf(2, "cannot cd %s\n", cmd + 3);
8    } else {
9      if (fork1() == 0)
10       runcmd(parsecmd(cmd));
11     wait(0);
12   }
13 }
```

Por qué `fork` y `exec` están separados

El ciclo de una shell:

1. Leer una línea y separarla en palabras.
2. `fork()`.
3. Hijo: preparar redirecciones *(redirection)*, luego `exec()`.
4. Padre: `wait()` al hijo (salvo con `&`).
5. Volver al paso 1.

Entre `fork` y `exec` el hijo ejecuta código normal y puede cambiar su propio entorno. Windows, con `CreateProcess`, no tiene ese hueco.

user/sh.c (xv6)

```
1  while (getcmd(buf, sizeof(buf)) >= 0) {
2    // ...
3    if (cmd[0] == 'c' && cmd[1] == 'd' &&
4        cmd[2] == ' ') {
5      // cd lo debe ejecutar el padre
6      if (chdir(cmd + 3) < 0)
7        fprintf(2, "cannot cd %s\n", cmd + 3);
8    } else {
9      if (fork1() == 0)
10       runcmd(parsecmd(cmd));
11     wait(0);
12   }
13 }
```

Redirección: `wc -l redirect.c > salida.txt`

redirect.c

```c
pid_t rc = fork();
if (rc == 0) {
  int fd = open("salida.txt",
      O_CREAT | O_WRONLY | O_TRUNC, 0644);
  dup2(fd, STDOUT_FILENO); // stdout = archivo
  close(fd);

  char *args[] = {"wc", "-l", "redirect.c", NULL};
  execvp(args[0], args);
  perror("execvp");
  exit(1);
}
waitpid(rc, NULL, 0);
```

bash

```
$ ./redirect
listo: ver salida.txt
$ cat salida.txt
33 redirect.c
```

Redirección: `wc -l redirect.c > salida.txt`

redirect.c

```c
pid_t rc = fork();
if (rc == 0) {
  int fd = open("salida.txt",
      O_CREAT | O_WRONLY | O_TRUNC, 0644);
  dup2(fd, STDOUT_FILENO); // stdout = archivo
  close(fd);

  char *args[] = {"wc", "-l", "redirect.c", NULL};
  execvp(args[0], args);
  perror("execvp");
  exit(1);
}
waitpid(rc, NULL, 0);
```

bash

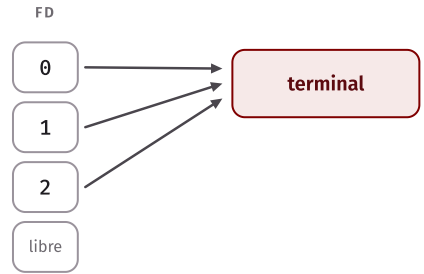
```
$ ./redirect
listo: ver salida.txt
$ cat salida.txt
33 redirect.c
```

`wc` no sabe nada de redirecciones: escribe en el descriptor 1, como siempre.

dup2: la tabla de descriptores de archivo *(file descriptors)*

CÓDIGO DEL HIJO

```
// hijo, recién creado por fork()
fd = open("salida.txt", ...);
dup2(fd, STDOUT_FILENO);
close(fd);
execvp("wc", args);
```



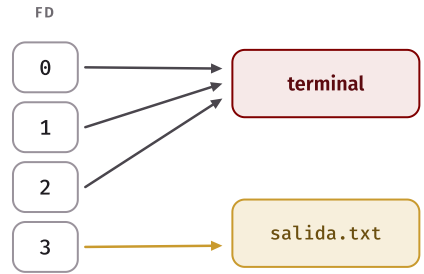
proceso: hijo de la shell

Igual que en el padre: 0, 1 y 2 apuntan a la terminal.

dup2: la tabla de descriptores de archivo *(file descriptors)*

CÓDIGO DEL HIJO

```
// hijo, recién creado por fork()
fd = open("salida.txt", ...);
dup2(fd, STDOUT_FILENO);
close(fd);
execvp("wc", args);
```



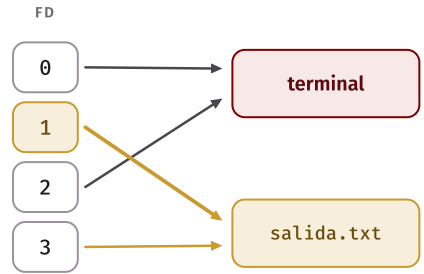
proceso: hijo de la shell

open() usa el menor descriptor libre: el 3.

dup2: la tabla de descriptores de archivo *(file descriptors)*

CÓDIGO DEL HIJO

```
// hijo, recién creado por fork()
fd = open("salida.txt", ...);
dup2(fd, STDOUT_FILENO);
close(fd);
execvp("wc", args);
```



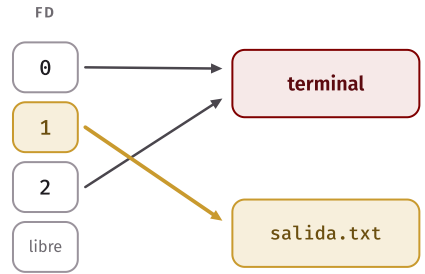
proceso: hijo de la shell

dup2(3, 1): el 1 pasa a apuntar a donde apunta el 3.

dup2: la tabla de descriptores de archivo *(file descriptors)*

CÓDIGO DEL HIJO

```
// hijo, recién creado por fork()
fd = open("salida.txt", ...);
dup2(fd, STDOUT_FILENO);
close(fd);
execvp("wc", args);
```



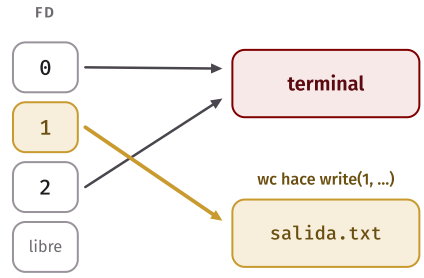
proceso: hijo de la shell

close(3): el 3 ya no hace falta.

dup2: la tabla de descriptores de archivo *(file descriptors)*

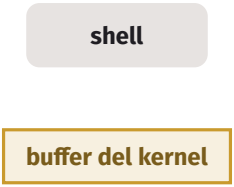
CÓDIGO DEL HIJO

```
// hijo, recién creado por fork()
fd = open("salida.txt", ...);
dup2(fd, STDOUT_FILENO);
close(fd);
execvp("wc", args);
```



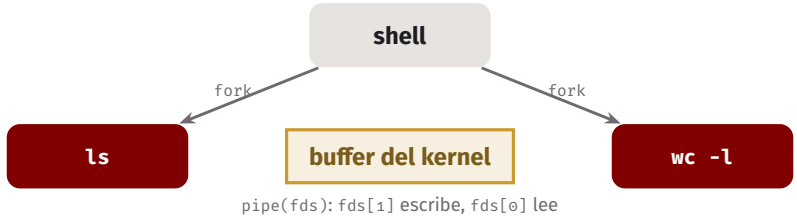
exec conserva la tabla: wc escribe en el 1 y llega al archivo.

Pipes: `ls | wc -l`



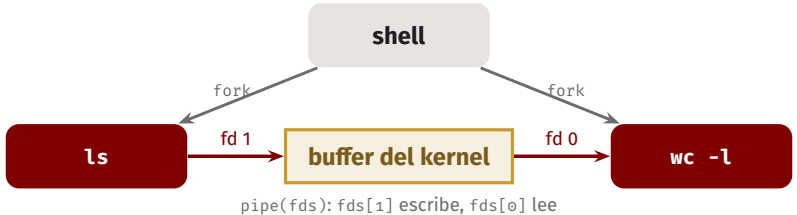
1. La shell crea el pipe *antes* de los `fork`: los hijos lo heredan.

Pipes: `ls | wc -l`



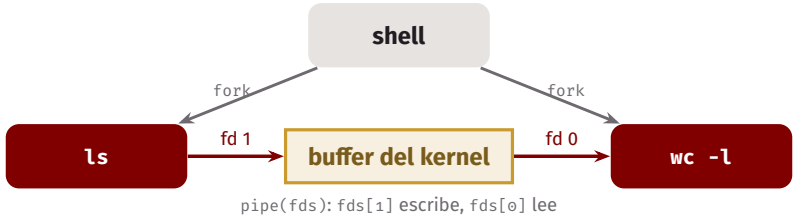
1. La shell crea el pipe *antes* de los `fork`: los hijos lo heredan.

Pipes: `ls | wc -l`



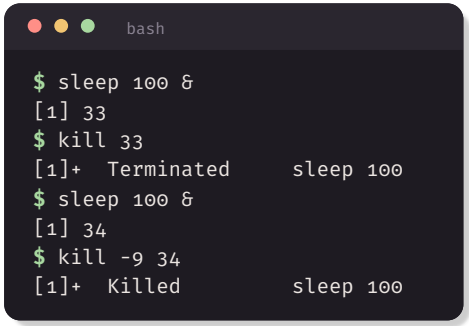
1. La shell crea el pipe *antes* de los `fork`: los hijos lo heredan.
2. Hijo izquierdo: `dup2(fds[1], 1)`; hijo derecho: `dup2(fds[0], 0)`. Luego `exec`.

Pipes: `ls | wc -l`



1. La shell crea el pipe *antes* de los `fork`: los hijos lo heredan.
2. Hijo izquierdo: `dup2(fds[1], 1)`; hijo derecho: `dup2(fds[0], 0)`. Luego `exec`.
3. Mismo truco que la redirección. Es lo que van a implementar en `wish` (P1).

Señales *(signals)*: avisarle algo a un proceso



Señal	Cuándo
`SIGINT`	Ctrl-C en la terminal
`SIGTERM`	`kill PID`: pedir que termine
`SIGKILL`	`kill -9`: no se puede capturar
`SIGTSTP`	Ctrl-Z: suspender
`SIGCONT`	`fg` / `bg`: reanudar
`SIGCHLD`	Un hijo terminó

Señales *(signals)*: avisarle algo a un proceso

```
bash
$ sleep 100 &
[1] 33
$ kill 33
[1]+  Terminated      sleep 100
$ sleep 100 &
[1] 34
$ kill -9 34
[1]+  Killed          sleep 100
```

Señal	Cuándo
`SIGINT`	Ctrl-C en la terminal
`SIGTERM`	`kill PID`: pedir que termine
`SIGKILL`	`kill -9`: no se puede capturar
`SIGTSTP`	Ctrl-Z: suspender
`SIGCONT`	`fg` / `bg`: reanudar
`SIGCHLD`	Un hijo terminó

Una interrupción para procesos

`kill(pid, sig)` es una llamada al sistema. Con `sigaction()` se instala un manejador; si no, la acción por defecto.

Usuarios y permisos sobre procesos

- Cada proceso pertenece a un usuario (UID), heredado del padre en `fork`.

como el usuario ana

```
$ id
uid=1001(ana) gid=1001(ana)
$ kill -9 1
bash: kill: (1) - Operation
  not permitted
```

Usuarios y permisos sobre procesos

- Cada proceso pertenece a un usuario (UID), heredado del padre en `fork`.
- Un usuario solo puede enviar señales a *sus* procesos.

como el usuario ana

```
$ id
uid=1001(ana) gid=1001(ana)
$ kill -9 1
bash: kill: (1) - Operation
  not permitted
```

Usuarios y permisos sobre procesos

- Cada proceso pertenece a un usuario (UID), heredado del padre en `fork`.
- Un usuario solo puede enviar señales a *sus* procesos.
- El superusuario (`root`, UID 0) puede controlar todos.

como el usuario ana

```
$ id
uid=1001(ana) gid=1001(ana)
$ kill -9 1
bash: kill: (1) - Operation
  not permitted
```

Usuarios y permisos sobre procesos

- Cada proceso pertenece a un usuario (UID), heredado del padre en `fork`.
- Un usuario solo puede enviar señales a *sus* procesos.
- El superusuario (`root`, UID 0) puede controlar todos.

como el usuario ana

```
$ id
uid=1001(ana) gid=1001(ana)
$ kill -9 1
bash: kill: (1) - Operation
  not permitted
```

En xv6 no hay usuarios

`kill(pid)` llama a `kkill()` (`kernel/proc.c`), que solo marca `p->killed = 1`. El proceso muere la próxima vez que entra al kernel: `usertrap()` revisa `killed(p)` y llama a `kexit(-1)`.

SECCIÓN 04

Ejecución directa limitada

04

¿Cómo virtualizar la CPU sin perder velocidad?

Ejecución directa

Lo más rápido: cargar el programa y saltar a `main`. Corre directo sobre la CPU, sin intérprete.

¿Cómo virtualizar la CPU sin perder velocidad?

Ejecución directa

Lo más rápido: cargar el programa y saltar a `main`. Corre directo sobre la CPU, sin intérprete.

Pero entonces…

1. ¿Cómo evitamos que el programa haga algo prohibido (leer el disco de otro usuario)?

2. Mientras el programa corre, el kernel *no corre*. ¿Cómo recupera la CPU?

¿Cómo virtualizar la CPU sin perder velocidad?

Ejecución directa

Lo más rápido: cargar el programa y saltar a `main`. Corre directo sobre la CPU, sin intérprete.

Pero entonces…

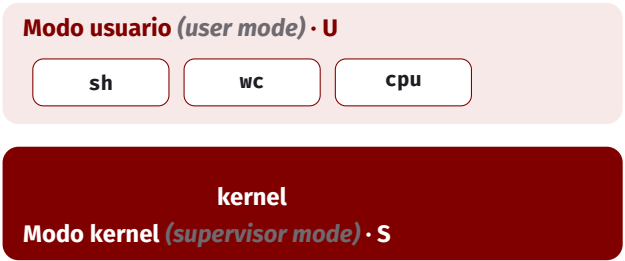
1. ¿Cómo evitamos que el programa haga algo prohibido (leer el disco de otro usuario)?

2. Mientras el programa corre, el kernel *no corre*. ¿Cómo recupera la CPU?

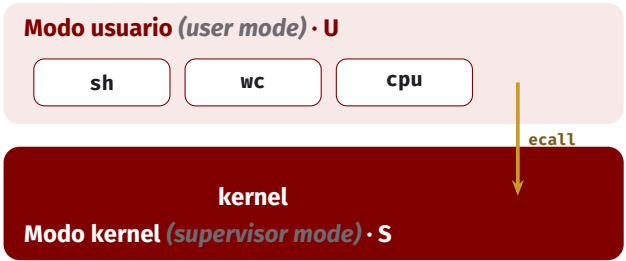
Ejecución directa *limitada* (*limited direct execution*)

El programa corre directo en el hardware, pero con límites que impone el propio hardware: modos de ejecución, traps e interrupción del temporizador.

Problema 1: modo usuario y modo kernel

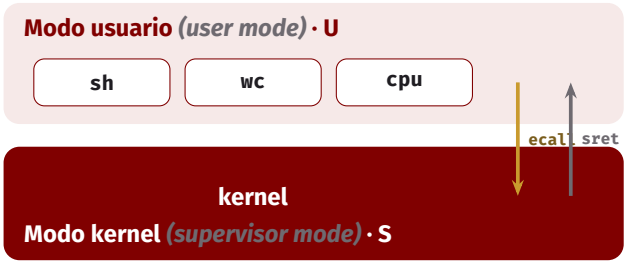


Problema 1: modo usuario y modo kernel



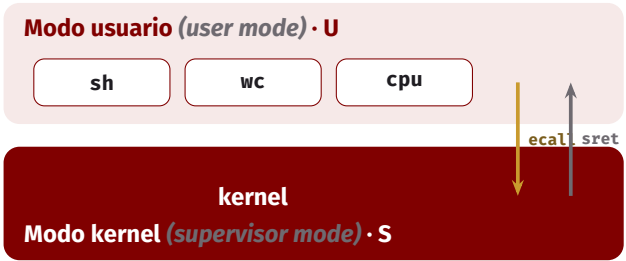
Para pedir algo al kernel, el programa ejecuta una instrucción de **trap**: `ecall` en RISC-V, `syscall` en x86-64.

Problema 1: modo usuario y modo kernel



Para pedir algo al kernel, el programa ejecuta una instrucción de **trap**: `ecall` en RISC-V, `syscall` en x86-64.

Problema 1: modo usuario y modo kernel



En modo U es ilegal:

· acceder a dispositivos
· cambiar la tabla de páginas (`satp`)
· desactivar interrupciones
· escribir `stvec`

Si lo intenta: excepción, y el kernel lo mata.

Para pedir algo al kernel, el programa ejecuta una instrucción de **trap**: `ecall` en RISC-V, `syscall` en x86-64.

El protocolo de un trap en RISC-V

Programa (modo U)	Hardware	Kernel (modo S)
		Antes de pasar a modo U: `stvec` ← manejador de traps

El protocolo de un trap en RISC-V

Programa (modo U)	Hardware	Kernel (modo S)
		Antes de pasar a modo U: `stvec` ← manejador de traps
`a7` ← nro. de llamada; `ecall`		

El protocolo de un trap en RISC-V

Programa (modo U)	Hardware	Kernel (modo S)
		Antes de pasar a modo U: `stvec` ← manejador de traps
`a7` ← nro. de llamada; `ecall`		
	`sepc` ← PC; `scause` ← 8; modo ← S; PC ← `stvec`	

El protocolo de un trap en RISC-V

Programa (modo U)	Hardware	Kernel (modo S)
		Antes de pasar a modo U: `stvec` ← manejador de traps
`a7` ← nro. de llamada; `ecall`		
	`sepc` ← PC; `scause` ← 8; modo ← S; PC ← `stvec`	
		Guarda los registros; `sepc += 4`; atiende la llamada; resultado en `a0`

El protocolo de un trap en RISC-V

Programa (modo U)	Hardware	Kernel (modo S)
		Antes de pasar a modo U: `stvec` ← manejador de traps
`a7` ← nro. de llamada; `ecall`		
	`sepc` ← PC; `scause` ← 8; modo ← S; PC ← `stvec`	
		Guarda los registros; `sepc += 4`; atiende la llamada; resultado en `a0`
		`sret`

El protocolo de un trap en RISC-V

Programa (modo U)	Hardware	Kernel (modo S)
		Antes de pasar a modo U: `stvec` ← manejador de traps
`a7` ← nro. de llamada; `ecall`		
	`sepc` ← PC; `scause` ← 8; modo ← S; PC ← `stvec`	
		Guarda los registros; `sepc += 4`; atiende la llamada; resultado en `a0`
		`sret`
	modo ← U; PC ← `sepc`	

El protocolo de un trap en RISC-V

Programa (modo U)	Hardware	Kernel (modo S)
		Antes de pasar a modo U: `stvec` ← manejador de traps
`a7` ← nro. de llamada; `ecall`		
	`sepc` ← PC; `scause` ← 8; modo ← S; PC ← `stvec`	
		Guarda los registros; `sepc += 4`; atiende la llamada; resultado en `a0`
		`sret`
	modo ← U; PC ← `sepc`	
Sigue después del `ecall`		

El protocolo de un trap en RISC-V

Programa (modo U)	Hardware	Kernel (modo S)
		Antes de pasar a modo U: `stvec` ← manejador de traps
`a7` ← nro. de llamada; `ecall`		
	`sepc` ← PC; `scause` ← 8; modo ← S; PC ← `stvec`	
		Guarda los registros; `sepc += 4`; atiende la llamada; resultado en `a0`
		`sret`
	modo ← U; PC ← `sepc`	
Sigue después del `ecall`		

El programa no elige a dónde salta: solo el kernel puede escribir `stvec`. En x86 el equivalente es la *tabla de traps (trap table)*.

Problema 2: recuperar la CPU

Enfoque cooperativo

El kernel espera a que el programa devuelva el control: al hacer una llamada al sistema, al fallar, o con un `yield()` explícito.

Problema 2: recuperar la CPU

Enfoque cooperativo

El kernel espera a que el programa devuelva el control: al hacer una llamada al sistema, al fallar, o con un `yield()` explícito.

Un bucle infinito cuelga todo

`while (1) ;` nunca llama al sistema. Solo queda reiniciar la máquina.

Problema 2: recuperar la CPU

Enfoque cooperativo

El kernel espera a que el programa devuelva el control: al hacer una llamada al sistema, al fallar, o con un `yield()` explícito.

Un bucle infinito cuelga todo

`while (1) ;` nunca llama al sistema. Solo queda reiniciar la máquina.

Enfoque no cooperativo

Un temporizador *(timer)* de hardware interrumpe a la CPU cada pocos milisegundos. El hardware hace lo mismo que con un trap: guarda el PC y salta a `stvec`.

Problema 2: recuperar la CPU

Enfoque cooperativo

El kernel espera a que el programa devuelva el control: al hacer una llamada al sistema, al fallar, o con un `yield()` explícito.

Un bucle infinito cuelga todo

`while (1) ;` nunca llama al sistema. Solo queda reiniciar la máquina.

Enfoque no cooperativo

Un temporizador *(timer)* de hardware interrumpe a la CPU cada pocos milisegundos. El hardware hace lo mismo que con un trap: guarda el PC y salta a `stvec`.

El kernel siempre vuelve

Haga lo que haga el programa. Solo el kernel puede programar el temporizador.

La interrupción del temporizador en xv6

kernel/trap.c

```
1  void clockintr() {
2    if (cpuid() == 0) {
3      acquire(&tickslock);
4      ticks++;
5      wakeup(&ticks);
6      release(&tickslock);
7    }
8    // siguiente interrupción: ~0.1 s
9    w_stimecmp(r_time() + 1000000);
10 }
11
12 // en usertrap(), al final:
13 if (which_dev == 2)   // temporizador
14   yield();            // ceder la CPU
```

- El hardware compara el reloj `time` con `stimecmp`; al alcanzarlo, interrumpe con `scause = 0x8000000000000005`.

La interrupción del temporizador en xv6

kernel/trap.c

```
1  void clockintr() {
2    if (cpuid() == 0) {
3      acquire(&tickslock);
4      ticks++;
5      wakeup(&ticks);
6      release(&tickslock);
7    }
8    // siguiente interrupción: ~0.1 s
9    w_stimecmp(r_time() + 1000000);
10 }
11
12 // en usertrap(), al final:
13 if (which_dev == 2)   // temporizador
14   yield();            // ceder la CPU
```

- El hardware compara el reloj `time` con `stimecmp`; al alcanzarlo, interrumpe con `scause = 0x8000000000000005`.
- `devintr()` lo reconoce y llama a `clockintr()`, que programa la *siguiente* interrupción.

La interrupción del temporizador en xv6

kernel/trap.c

```
1  void clockintr() {
2    if (cpuid() == 0) {
3      acquire(&tickslock);
4      ticks++;
5      wakeup(&ticks);
6      release(&tickslock);
7    }
8    // siguiente interrupción: ~0.1 s
9    w_stimecmp(r_time() + 1000000);
10 }
11
12 // en usertrap(), al final:
13 if (which_dev == 2)   // temporizador
14   yield();            // ceder la CPU
```

- El hardware compara el reloj `time` con `stimecmp`; al alcanzarlo, interrumpe con `scause = 0x8000000000000005`.
- `devintr()` lo reconoce y llama a `clockintr()`, que programa la *siguiente* interrupción.
- `yield()` pone al proceso en `RUNNABLE` y llama al planificador.

Cambio de contexto *(context switch)*: `kernel/swtch.S`

`void swtch(struct context *old, struct context *new);`

kernel/swtch.S

```
1  swtch:
2      sd ra, 0(a0)
3      sd sp, 8(a0)
4      sd s0, 16(a0)
5      # ... s1 a s10
6      sd s11, 104(a0)
7
8      ld ra, 0(a1)
9      ld sp, 8(a1)
10     ld s0, 16(a1)
11     # ... s1 a s10
12     ld s11, 104(a1)
13     ret
```

kernel/proc.h

```
1  struct context {
2    uint64 ra;    // dirección de retorno
3    uint64 sp;    // puntero de pila
4    uint64 s0;    // callee-saved: s0 a s11
5    // ...
6    uint64 s11;
7  };
```

Cambio de contexto *(context switch)*: `kernel/swtch.S`

`void swtch(struct context *old, struct context *new);`

kernel/swtch.S

```
1  swtch:
2      sd ra, 0(a0)
3      sd sp, 8(a0)
4      sd s0, 16(a0)
5      # ... s1 a s10
6      sd s11, 104(a0)
7
8      ld ra, 0(a1)
9      ld sp, 8(a1)
10     ld s0, 16(a1)
11     # ... s1 a s10
12     ld s11, 104(a1)
13     ret
```

kernel/proc.h

```
1  struct context {
2    uint64 ra;    // dirección de retorno
3    uint64 sp;    // puntero de pila
4    uint64 s0;    // callee-saved: s0 a s11
5    // ...
6    uint64 s11;
7  };
```

- `sd` guarda en `old` (`a0`); `ld` carga de `new` (`a1`).

Cambio de contexto *(context switch)*: `kernel/swtch.S`

`void swtch(struct context *old, struct context *new);`

kernel/swtch.S

```
1  swtch:
2      sd ra, 0(a0)
3      sd sp, 8(a0)
4      sd s0, 16(a0)
5      # ... s1 a s10
6      sd s11, 104(a0)
7
8      ld ra, 0(a1)
9      ld sp, 8(a1)
10     ld s0, 16(a1)
11     # ... s1 a s10
12     ld s11, 104(a1)
13     ret
```

kernel/proc.h

```
1  struct context {
2    uint64 ra;    // dirección de retorno
3    uint64 sp;    // puntero de pila
4    uint64 s0;    // callee-saved: s0 a s11
5    // ...
6    uint64 s11;
7  };
```

- `sd` guarda en `old` (`a0`); `ld` carga de `new` (`a1`).
- Solo 14 registros: el resto ya lo guardó el compilador en la pila.

Cambio de contexto *(context switch)*: `kernel/swtch.S`

`kernel/swtch.S`

```
1  swtch:
2      sd ra, 0(a0)
3      sd sp, 8(a0)
4      sd s0, 16(a0)
5      # ... s1 a s10
6      sd s11, 104(a0)
7
8      ld ra, 0(a1)
9      ld sp, 8(a1)
10     ld s0, 16(a1)
11     # ... s1 a s10
12     ld s11, 104(a1)
13     ret
```

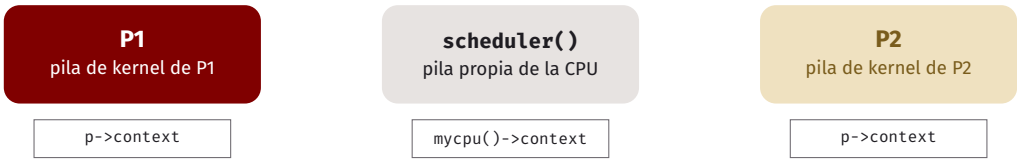
```
void swtch(struct context *old, struct context *new);
```

`kernel/proc.h`

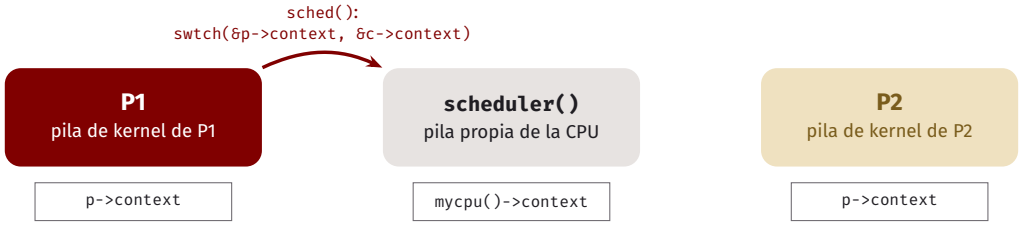
```
1  struct context {
2    uint64 ra;    // dirección de retorno
3    uint64 sp;    // puntero de pila
4    uint64 s0;    // callee-saved: s0 a s11
5    // ...
6    uint64 s11;
7  };
```

- `sd` guarda en `old` (`a0`); `ld` carga de `new` (`a1`).
- Solo 14 registros: el resto ya lo guardó el compilador en la pila.
- Con otros `sp` y `ra`, `ret` vuelve a otro hilo del kernel.

En xv6, un cambio de contexto son *dos* `swtch`

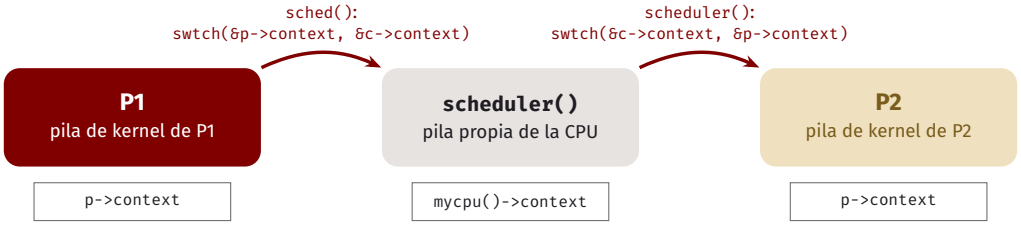


En xv6, un cambio de contexto son *dos* `swtch`



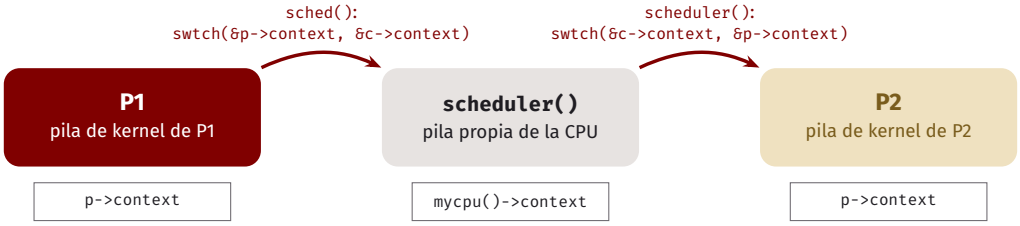
1. P1 (en `yield`) guarda sus registros en *su* `context` y salta al planificador.

En xv6, un cambio de contexto son *dos* `swtch`



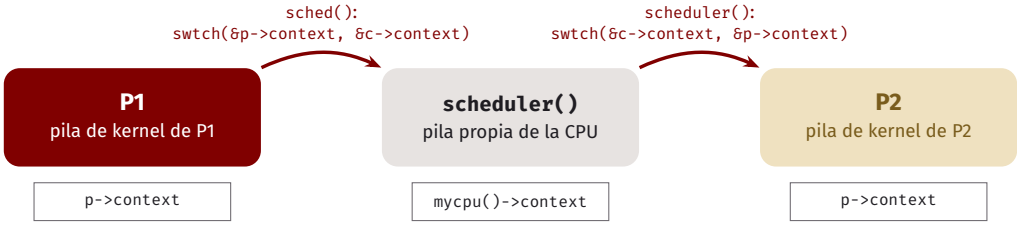
1. P1 (en `yield`) guarda sus registros en *su* `context` y salta al planificador.
2. El planificador elige al siguiente `RUNNABLE` y carga *su* `context`.

En xv6, un cambio de contexto son *dos* `swtch`



1. P1 (en `yield`) guarda sus registros en *su* `context` y salta al planificador.
2. El planificador elige al siguiente `RUNNABLE` y carga *su* `context`.
3. P2 continúa dentro de *su propio* `yield()`, que había quedado a medias. Vuelve a modo usuario.

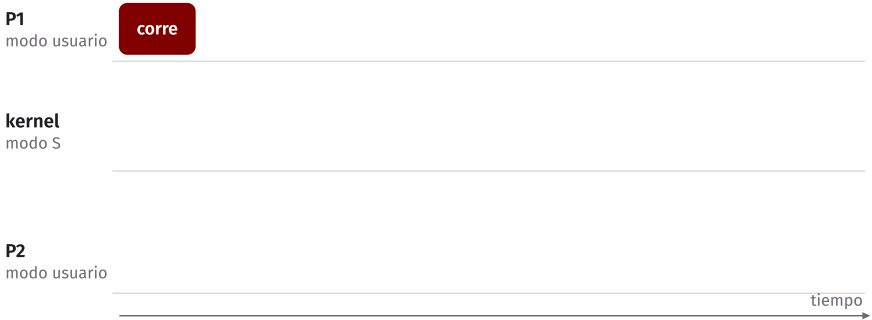
En xv6, un cambio de contexto son *dos* `swtch`



1. P1 (en `yield`) guarda sus registros en *su* `context` y salta al planificador.
2. El planificador elige al siguiente `RUNNABLE` y carga *su* `context`.
3. P2 continúa dentro de *su propio* `yield()`, que había quedado a medias. Vuelve a modo usuario.

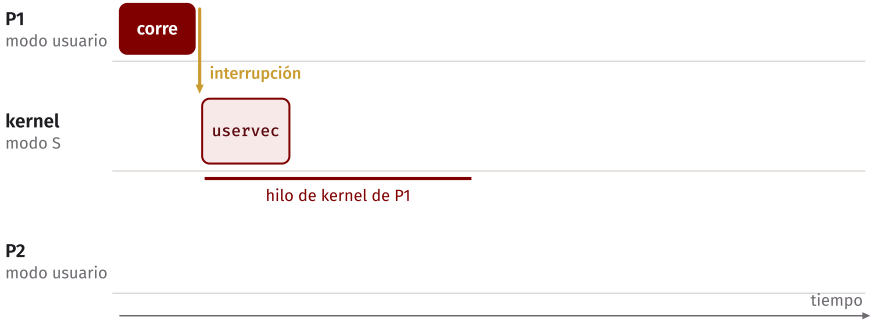
Mecanismo: `swtch`. **Política**: el bucle de `scheduler()`.

Línea de tiempo: del temporizador a otro proceso



P1 corre en modo usuario. El kernel no está ejecutando.

Línea de tiempo: del temporizador a otro proceso



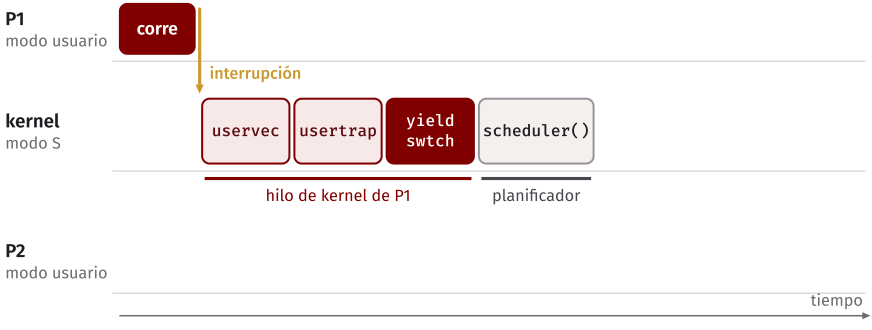
Interrupción del reloj: el hardware salta a uservec, que guarda los registros de P1.

Línea de tiempo: del temporizador a otro proceso



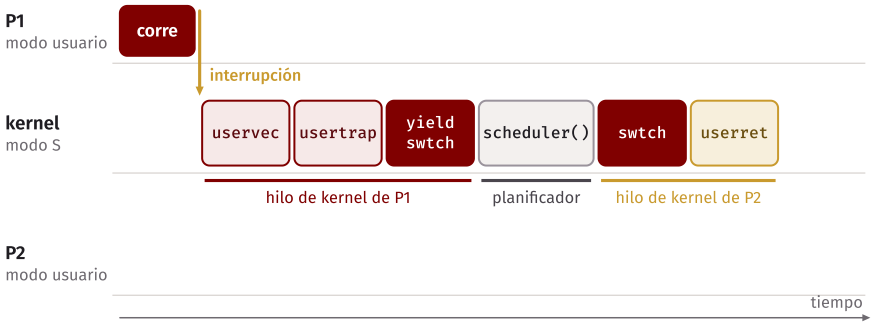
usertrap() ve que la causa es el temporizador (which_dev == 2) y llama a yield().

Línea de tiempo: del temporizador a otro proceso



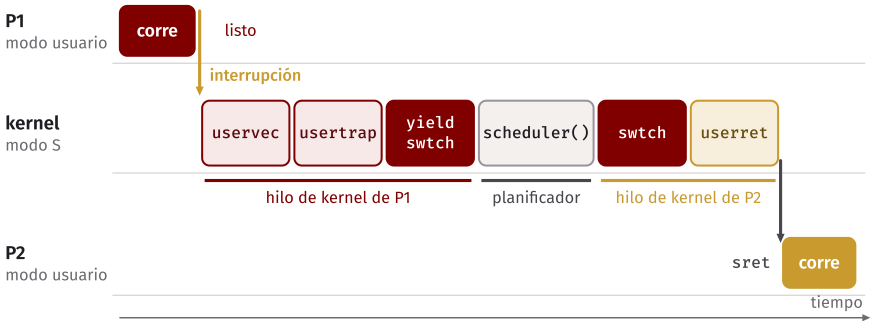
Primer swtch: del hilo de kernel de P1 al planificador, que elige a P2.

Línea de tiempo: del temporizador a otro proceso



Segundo swtch: P2 sigue en su propio yield(); userret restaura sus registros.

Línea de tiempo: del temporizador a otro proceso



sret: P2 vuelve a modo usuario como si nada hubiera pasado. P1 queda listo.

SECCIÓN 05

Una llamada al sistema en xv6

05



Lado usuario: `user/usys.pl`

user/usys.pl

```
sub entry {
    my $prefix = "sys_";
    my $name = shift;
    # ... (caso especial: sbrk)
    print ".global $name\n";
    print "$name:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}

entry("fork");
# ... exit, wait, pipe, read...
```

Al compilar, genera un `user/usys.S` con una función por llamada:

user/usys.S (generado)

```
.global fork
fork:
 li a7, SYS_fork
 ecall
 ret
```

Lado usuario: `user/usys.pl`

user/usys.pl

```
sub entry {
    my $prefix = "sys_";
    my $name = shift;
    # ... (caso especial: sbrk)
    print ".global $name\n";
    print "$name:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}

entry("fork");
# ... exit, wait, pipe, read...
```

Al compilar, genera un `user/usys.S` con una función por llamada:

user/usys.S (generado)

```
.global fork
fork:
 li a7, SYS_fork
 ecall
 ret
```

- `SYS_fork` vale 1 (`kernel/syscall.h`).

Lado usuario: `user/usys.pl`

user/usys.pl

```
sub entry {
    my $prefix = "sys_";
    my $name = shift;
    # ... (caso especial: sbrk)
    print ".global $name\n";
    print "$name:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}

entry("fork");
# ... exit, wait, pipe, read...
```

Al compilar, genera un `user/usys.S` con una función por llamada:

user/usys.S (generado)

```
.global fork
fork:
 li a7, SYS_fork
 ecall
 ret
```

- `SYS_fork` vale 1 (`kernel/syscall.h`).
- Los argumentos ya están en `a0`, `a1`… por la convención de llamada de C.

Entrada al kernel: `uservec` en `kernel/trampoline.S`

kernel/trampoline.S (resumido)

```
1  uservec:
2      csrw sscratch, a0  # liberar a0
3      li a0, TRAPFRAME
4      sd ra, 40(a0)      # guardar registros
5      sd sp, 48(a0)      #   de usuario
6      # ...
7      sd t6, 280(a0)
8
9      ld sp, 8(a0)       # pila de kernel
10     ld t0, 16(a0)      # &usertrap
11     ld t1, 0(a0)       # tabla de páginas
12     csrw satp, t1      #   del kernel
13     jalr t0            # usertrap()
```

- El `trapframe` es una página por proceso donde se guardan los registros de usuario.

Entrada al kernel: `uservec` en `kernel/trampoline.S`

kernel/trampoline.S (resumido)

```
1  uservec:
2      csrw sscratch, a0   # liberar a0
3      li a0, TRAPFRAME
4      sd ra, 40(a0)       # guardar registros
5      sd sp, 48(a0)       #   de usuario
6      # ...
7      sd t6, 280(a0)
8
9      ld sp, 8(a0)        # pila de kernel
10     ld t0, 16(a0)       # &usertrap
11     ld t1, 0(a0)        # tabla de páginas
12     csrw satp, t1       #   del kernel
13     jalr t0             # usertrap()
```

- El `trapframe` es una página por proceso donde se guardan los registros de usuario.
- `uservec` está en la página *trampoline*, mapeada en la misma dirección en las tablas de usuario y de kernel: sigue funcionando al cambiar `satp`.

Entrada al kernel: `uservec` en `kernel/trampoline.S`

kernel/trampoline.S (resumido)

```
1  uservec:
2      csrw sscratch, a0   # liberar a0
3      li a0, TRAPFRAME
4      sd ra, 40(a0)       # guardar registros
5      sd sp, 48(a0)       #   de usuario
6      # ...
7      sd t6, 280(a0)
8
9      ld sp, 8(a0)        # pila de kernel
10     ld t0, 16(a0)       # &usertrap
11     ld t1, 0(a0)        # tabla de páginas
12     csrw satp, t1       #   del kernel
13     jalr t0             # usertrap()
```

- El `trapframe` es una página por proceso donde se guardan los registros de usuario.
- `uservec` está en la página *trampoline*, mapeada en la misma dirección en las tablas de usuario y de kernel: sigue funcionando al cambiar `satp`.
- Lo hace en ensamblador porque C necesitaría registros que todavía no se guardaron.

`usertrap()` en `kernel/trap.c` (resumido)

kernel/trap.c

```
1  w_stvec((uint64)kernelvec);
2  p->trapframe->epc = r_sepc();
3  if (r_scause() == 8) {      // system call
4    if (killed(p))
5      kexit(-1);
6    p->trapframe->epc += 4;
7    syscall();
8  } else if ((which_dev = devintr()) != 0) {
9    // ok
10 } // ... fallos de página, errores
11 if (which_dev == 2)
12   yield();
13 prepare_return();
```

`usertrap()` en `kernel/trap.c` (resumido)

kernel/trap.c

```
1  w_stvec((uint64)kernelvec);
2  p->trapframe->epc = r_sepc();
3  if (r_scause() == 8) {      // system call
4    if (killed(p))
5      kexit(-1);
6    p->trapframe->epc += 4;
7    syscall();
8  } else if ((which_dev = devintr()) != 0) {
9    // ok
10 } // ... fallos de página, errores
11 if (which_dev == 2)
12   yield();
13 prepare_return();
```

- `scause` dice por qué se entró: 8 = `ecall` desde modo U.

`usertrap()` en `kernel/trap.c` (resumido)

kernel/trap.c

```
1  w_stvec((uint64)kernelvec);
2  p->trapframe->epc = r_sepc();
3  if (r_scause() == 8) {      // system call
4    if (killed(p))
5      kexit(-1);
6    p->trapframe->epc += 4;
7    syscall();
8  } else if ((which_dev = devintr()) != 0) {
9    // ok
10 } // ... fallos de página, errores
11 if (which_dev == 2)
12   yield();
13 prepare_return();
```

- `scause` dice por qué se entró: 8 = `ecall` desde modo U.
- `epc += 4`: al volver, seguir en la instrucción *siguiente*.

`usertrap()` en `kernel/trap.c` (resumido)

kernel/trap.c

```
1  w_stvec((uint64)kernelvec);
2  p->trapframe->epc = r_sepc();
3  if (r_scause() == 8) {      // system call
4    if (killed(p))
5      kexit(-1);
6    p->trapframe->epc += 4;
7    syscall();
8  } else if ((which_dev = devintr()) != 0) {
9    // ok
10 } // ... fallos de página, errores
11 if (which_dev == 2)
12   yield();
13 prepare_return();
```

- `scause` dice por qué se entró: 8 = `ecall` desde modo U.
- `epc += 4`: al volver, seguir en la instrucción *siguiente*.
- `prepare_return()`: `stvec` ← `uservec`, `sepc` ← `epc`, modo previo ← U.

`usertrap()` en `kernel/trap.c` (resumido)

kernel/trap.c

```
1  w_stvec((uint64)kernelvec);
2  p->trapframe->epc = r_sepc();
3  if (r_scause() == 8) {      // system call
4    if (killed(p))
5      kexit(-1);
6    p->trapframe->epc += 4;
7    syscall();
8  } else if ((which_dev = devintr()) != 0) {
9    // ok
10 } // ... fallos de página, errores
11 if (which_dev == 2)
12   yield();
13 prepare_return();
```

- `scause` dice por qué se entró: 8 = `ecall` desde modo U.
- `epc += 4`: al volver, seguir en la instrucción *siguiente*.
- `prepare_return()`: `stvec` ← `uservec`, `sepc` ← `epc`, modo previo ← U.
- Retorna (con la tabla de páginas de usuario) a `userret`, que restaura los registros y hace `sret`.

`syscall()`: despachar por número

kernel/syscall.c

```
1  static uint64 (*syscalls[])(void) = {
2    [SYS_fork]    = sys_fork,
3    // ...
4  };
5  void syscall(void) {
6    // ...
7    num = p->trapframe->a7;
8    if (num > 0 && num < NELEM(syscalls) &&
9        syscalls[num]) {
10     p->trapframe->a0 = syscalls[num]();
11   } else {
12     p->trapframe->a0 = -1;  // + printk
13   }
14 }
```

kernel/sysproc.c

```
1  uint64
2  sys_fork(void)
3  {
4    return kfork();
5  }
```

`syscall()`: despachar por número

kernel/syscall.c

```
1  static uint64 (*syscalls[])(void) = {
2    [SYS_fork]    = sys_fork,
3    // ...
4  };
5  void syscall(void) {
6    // ...
7    num = p->trapframe->a7;
8    if (num > 0 && num < NELEM(syscalls) &&
9        syscalls[num]) {
10     p->trapframe->a0 = syscalls[num]();
11   } else {
12     p->trapframe->a0 = -1;  // + printk
13   }
14 }
```

kernel/sysproc.c

```
1  uint64
2  sys_fork(void)
3  {
4    return kfork();
5  }
```

- El número sale de `a7`, guardado en el trapframe.

`syscall()`: despachar por número

kernel/syscall.c

```
1  static uint64 (*syscalls[])(void) = {
2    [SYS_fork]    = sys_fork,
3    // ...
4  };
5  void syscall(void) {
6    // ...
7    num = p->trapframe->a7;
8    if (num > 0 && num < NELEM(syscalls) &&
9        syscalls[num]) {
10     p->trapframe->a0 = syscalls[num]();
11   } else {
12     p->trapframe->a0 = -1;  // + printk
13   }
14 }
```

kernel/sysproc.c

```
1  uint64
2  sys_fork(void)
3  {
4    return kfork();
5  }
```

- El número sale de `a7`, guardado en el trapframe.
- El resultado va a `trapframe->a0`: al restaurar, `fork()` lo ve como valor de retorno.

El camino completo de `fork()` en xv6-riscv

`fork()` en `user/usys.S`: `li a7, 1; ecall`

modo usuario

modo kernel

El camino completo de `fork()` en xv6-riscv



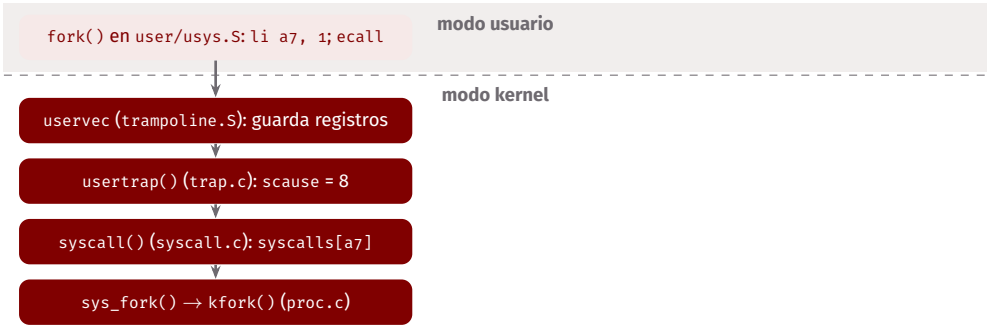
El camino completo de `fork()` en xv6-riscv



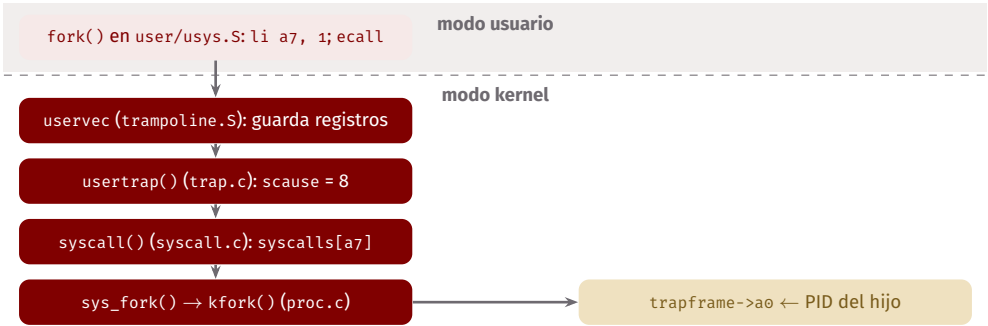
El camino completo de `fork()` en xv6-riscv



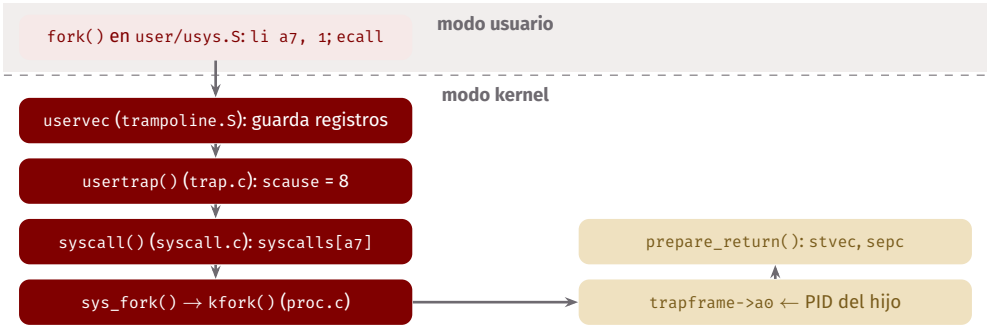
El camino completo de `fork()` en xv6-riscv



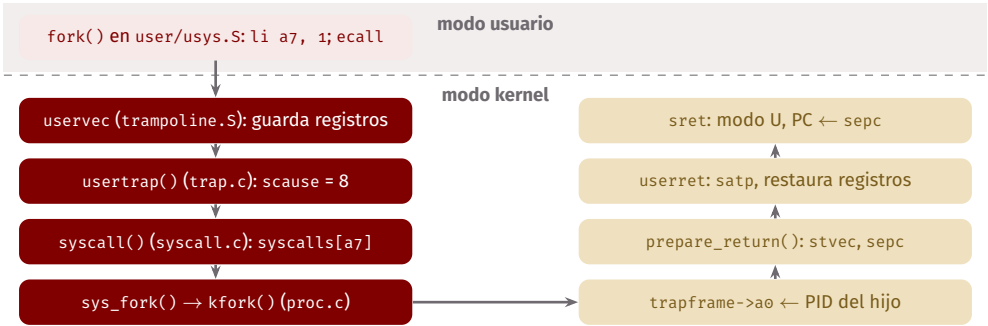
El camino completo de `fork()` en xv6-riscv



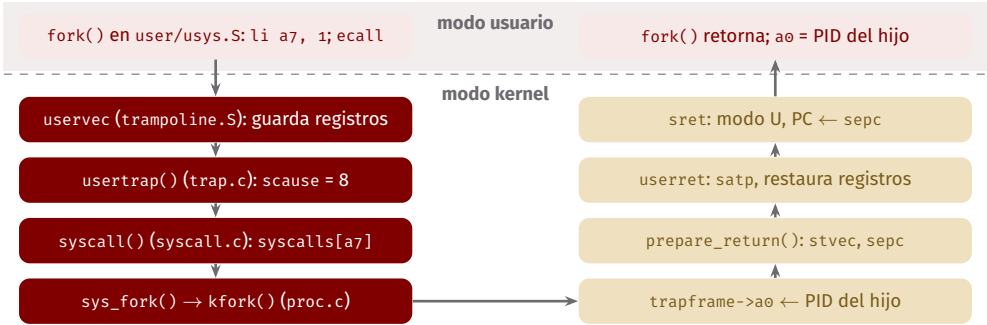
El camino completo de `fork()` en xv6-riscv



El camino completo de `fork()` en xv6-riscv



El camino completo de `fork()` en xv6-riscv



El hijo, en cambio, empieza en `forkret()` y sale por el mismo `prepare_return` + `userret`, con `a0 = 0`. Lo seguiremos con gdb en la semana 4.

¿Cuánto cuesta todo esto?

Una llamada al sistema

Trap, guardar y restaurar registros, cambiar de tabla de páginas, volver.

Medición: un millón de `read(fd, buf, 0)` entre dos `clock_gettime`; dividir.

¿Cuánto cuesta todo esto?

Una llamada al sistema

Trap, guardar y restaurar registros, cambiar de tabla de páginas, volver.

Medición: un millón de `read(fd, buf, 0)` entre dos `clock_gettime`; dividir.

Un cambio de contexto

Entrar al kernel, planificador, `swtch`, volver. Y la caché y la TLB quedan "frías".

Medición: dos procesos se pasan un byte por dos pipes, en un solo núcleo (`taskset`).

¿Cuánto cuesta todo esto?

Una llamada al sistema

Trap, guardar y restaurar registros, cambiar de tabla de páginas, volver.

Medición: un millón de `read(fd, buf, 0)` entre dos `clock_gettime`; dividir.

Un cambio de contexto

Entrar al kernel, planificador, `swtch`, volver. Y la caché y la TLB quedan "frías".

Medición: dos procesos se pasan un byte por dos pipes, en un solo núcleo (`taskset`).

Laboratorio

Midan los dos en el contenedor. ¿Cuántas llamadas a función "valen" una llamada al sistema?

Resumen

- `fork()` copia el proceso y retorna dos veces; el orden posterior no está garantizado. Copia también los buffers de `stdio`.
- `wait()` recoge al hijo; hasta entonces es un **zombi**. Los huérfanos los adopta `init`.
- `exec()` reemplaza el programa, conserva PID y descriptores. Separarlo de `fork` permite redirección y pipes con `dup2`.
- Las señales avisan eventos a un proceso; solo el owner (o `root`) puede enviarlas.
- **Ejecución directa limitada**: modo U/S, `ecall`/`stvec`/`sret`, interrupción del temporizador y `swtch`.
- Una llamada al sistema en xv6: `usys.S` → `uservec` → `usertrap` → `syscall` → `sys_*` → `userret`.

Para la próxima clase

- Leer OSTEP cap. 7: *Scheduling: Introduction*.
- Laboratorio: tarea de código del cap. 5 (preguntas 1–8); `strace -f` sobre `fork`/`exec`; medir el costo de una llamada al sistema y de un cambio de contexto (cap. 6).
- Proyecto **P0** (utilidades Unix): se entrega la **semana 3**.
- La semana 3 se publica **P1**: la shell `wish`.

Pregunta para pensar

El temporizador le devuelve la CPU al kernel cada cierto tiempo. ¿A qué proceso le conviene dársela? ¿Al que llegó primero, al más corto, a todos por turnos?