

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

Lotería, multiprocesadores y primer contacto con xv6

Repartir la CPU en proporción, entre varios CPUs, y un primer vistazo al kernel

Gerson Garrido

Semana 4

Hoy

1. Planificación proporcional: lotería, moneda y transferencia de boletos
2. Stride y el planificador de Linux: CFS, `vruntime` y EEVDF
3. Varios CPUs: cachés, coherencia, afinidad, una cola o muchas
4. Primer contacto con xv6-riscv: cómo arranca y cómo viaja una llamada al sistema

> **Lectura**
>
> OSTEP cap. 9 *Proportional Share*, cap. 10 *Multiprocessor Scheduling* y cap. 11 (diálogo de cierre de la CPU).

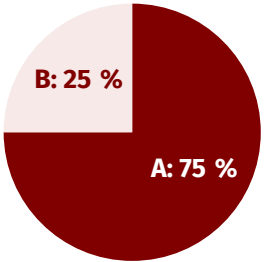
SECCIÓN 01

Planificación por lotería

01

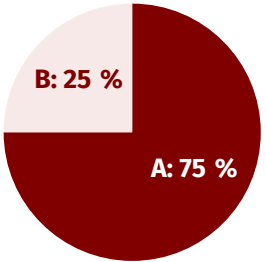
Otra meta: repartir en proporción

- MLFQ optimiza tiempo de retorno y de respuesta.



Otra meta: repartir en proporción

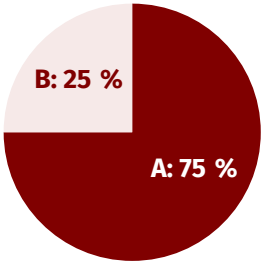
- MLFQ optimiza tiempo de retorno y de respuesta.
- A veces la meta es otra: que cada uno reciba una fracción fija de la CPU. Planificación proporcional *(proportional share)*.



tiempo de CPU a largo plazo

Otra meta: repartir en proporción

- MLFQ optimiza tiempo de retorno y de respuesta.
- A veces la meta es otra: que cada uno reciba una fracción fija de la CPU. Planificación proporcional *(proportional share)*.
- Ejemplos: dos máquinas virtuales que pagan 75 % y 25 % de un servidor; un usuario que no debe acaparar la máquina.

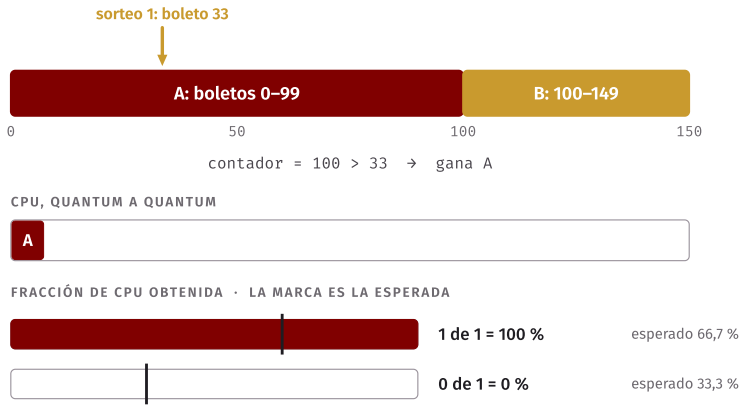


tiempo de CPU a largo plazo

Boletos *(tickets)*

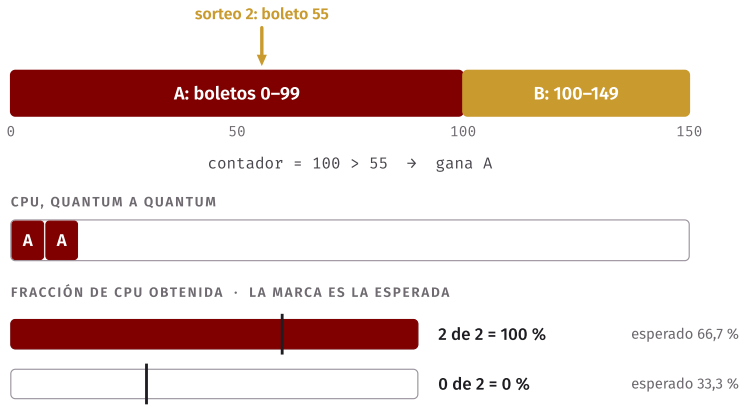
Cada proceso tiene un número de boletos. Su fracción de CPU es la fracción de boletos que tiene: con 300 y 100 boletos, A espera 75 % y B 25 %.

La lotería, sorteo a sorteo



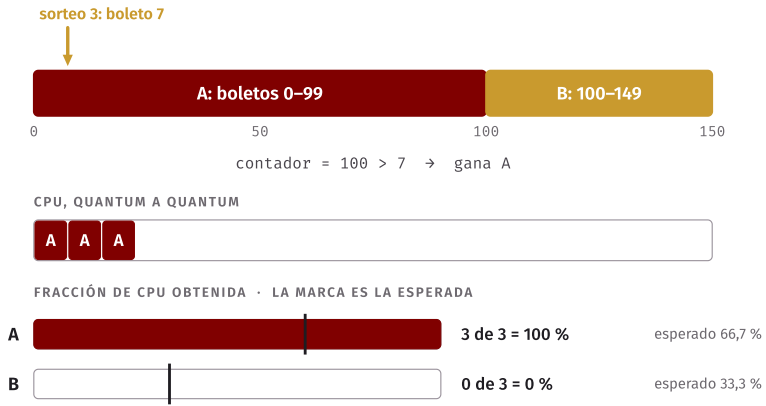
`./lottery 7 20 100 50`: cada quantum, un número entre 0 y 149. Cae en 33: corre A.

La lotería, sorteo a sorteo



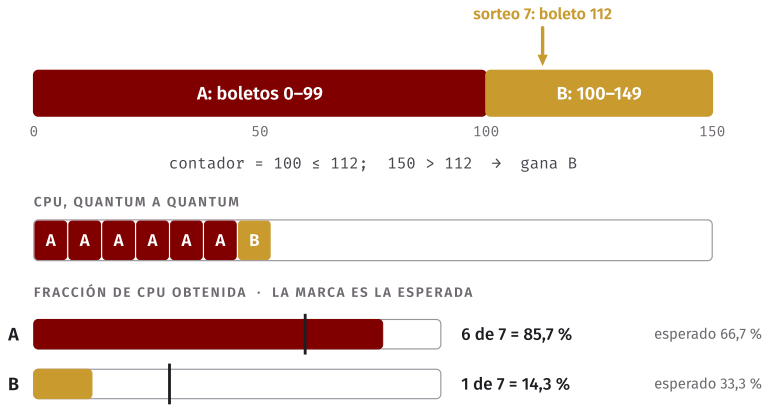
A tiene 2/3 de los boletos: en promedio gana 2 de cada 3 sorteos, pero nada lo garantiza.

La lotería, sorteo a sorteo



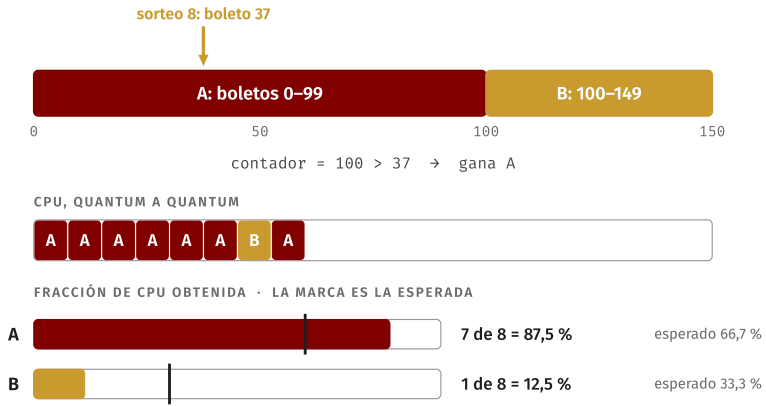
A tiene 2/3 de los boletos: en promedio gana 2 de cada 3 sorteos, pero nada lo garantiza.

La lotería, sorteo a sorteo



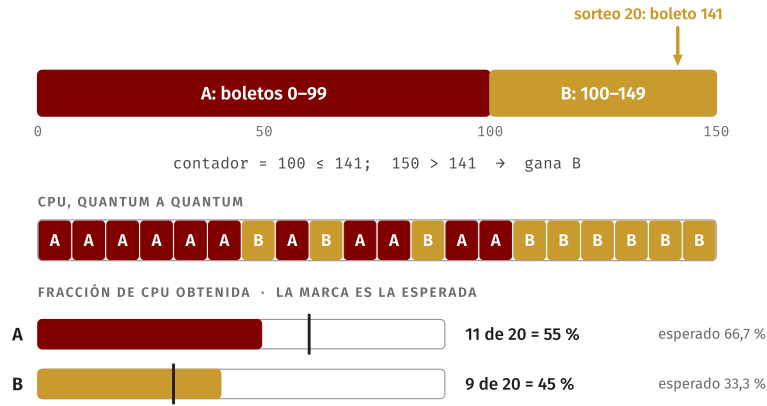
El sorteo 7 cae en 112: gana B por primera vez.

La lotería, sorteo a sorteo



El sorteo 7 cae en 112: gana B por primera vez.

La lotería, sorteo a sorteo



En 20 quanta A ganó solo el 55 %; en 10 000 quanta, el 66,9 %. La lotería es justa *en probabilidad*.

Implementación: una lista y un número aleatorio

lottery.c

```c
// Recorre la lista sumando boletos: O(n).
static int pick_winner(struct job *jobs,
                       int n, int winner) {
    int counter = 0;
    for (int i = 0; i < n; i++) {
        counter += jobs[i].tickets;
        if (counter > winner)
            return i;
    }
    return n - 1;  // no debería pasar
}
// en el bucle principal, cada quantum:
int ticket = next_random() % total;
int w = pick_winner(jobs, n, ticket);
```

- Sortear: un número en $[0, \text{total})$.
- Recorrer: sumar boletos hasta pasarlo.

Implementación: una lista y un número aleatorio

lottery.c

```c
// Recorre la lista sumando boletos: O(n).
static int pick_winner(struct job *jobs,
                       int n, int winner) {
    int counter = 0;
    for (int i = 0; i < n; i++) {
        counter += jobs[i].tickets;
        if (counter > winner)
            return i;
    }
    return n - 1;  // no debería pasar
}
// en el bucle principal, cada quantum:
int ticket = next_random() % total;
int w = pick_winner(jobs, n, ticket);
```

- Sortear: un número en $[0, \text{total})$.
- Recorrer: sumar boletos hasta pasarlo.
- Sin estado por proceso: solo boletos.

Implementación: una lista y un número aleatorio

lottery.c

```c
// Recorre la lista sumando boletos: O(n).
static int pick_winner(struct job *jobs,
                       int n, int winner) {
    int counter = 0;
    for (int i = 0; i < n; i++) {
        counter += jobs[i].tickets;
        if (counter > winner)
            return i;
    }
    return n - 1;  // no debería pasar
}
// en el bucle principal, cada quantum:
int ticket = next_random() % total;
int w = pick_winner(jobs, n, ticket);
```

- Sortear: un número en $[0, \text{total})$.
- Recorrer: sumar boletos hasta pasarlo.
- Sin estado por proceso: solo boletos.
- Ordenar la lista de mayor a menor acorta el recorrido.

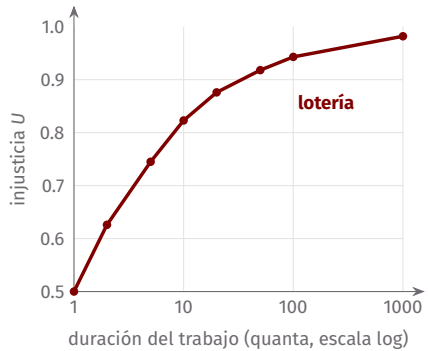
Implementación: una lista y un número aleatorio

lottery.c

```c
// Recorre la lista sumando boletos: O(n).
static int pick_winner(struct job *jobs,
                       int n, int winner) {
    int counter = 0;
    for (int i = 0; i < n; i++) {
        counter += jobs[i].tickets;
        if (counter > winner)
            return i;
    }
    return n - 1;  // no debería pasar
}
// en el bucle principal, cada quantum:
int ticket = next_random() % total;
int w = pick_winner(jobs, n, ticket);
```

- Sortear: un número en $[0, \text{total})$.
- Recorrer: sumar boletos hasta pasarlo.
- Sin estado por proceso: solo boletos.
- Ordenar la lista de mayor a menor acorta el recorrido.
- Proceso nuevo: sumar sus boletos.

¿Qué tan justa es en el corto plazo?



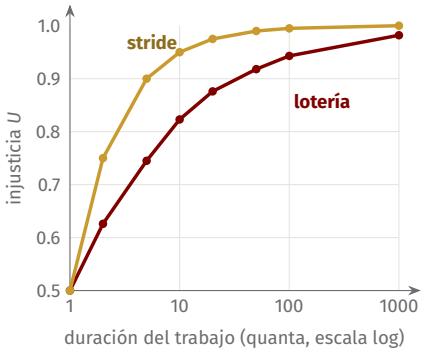
Dos trabajos iguales, mismos boletos. Medimos

$$U = \frac{\text{fin del primero}}{\text{fin del segundo}}$$

- $U = 1$: terminan juntos (justo).
- Con trabajos de 10 quanta, la lotería da $U \approx 0{,}82$.

Datos: `./fairness 2000` (promedio de 2000 ensayos por punto).

¿Qué tan justa es en el corto plazo?



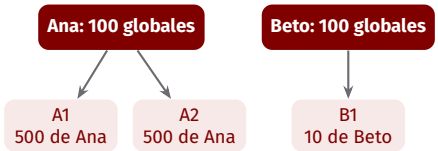
Dos trabajos iguales, mismos boletos. Medimos

$$U = \frac{\text{fin del primero}}{\text{fin del segundo}}$$

- $U = 1$: terminan juntos (justo).
- Con trabajos de 10 quanta, la lotería da $U \approx 0{,}82$.
- Stride (próxima sección) es casi perfecto desde el inicio.

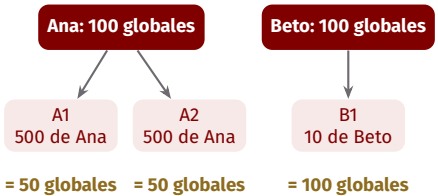
Datos: `./fairness 2000` (promedio de 2000 ensayos por punto).

Moneda, transferencia e inflación



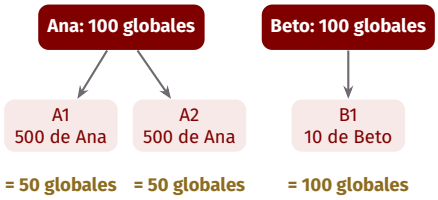
Moneda *(ticket currency)*: cada usuario reparte *su* moneda entre sus procesos; el sistema la convierte a boletos globales.

Moneda, transferencia e inflación



Moneda *(ticket currency)*: cada usuario reparte *su* moneda entre sus procesos; el sistema la convierte a boletos globales.

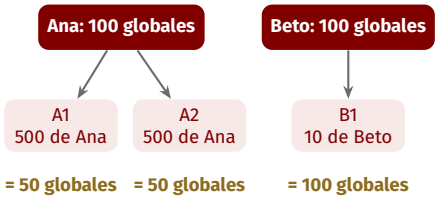
Moneda, transferencia e inflación



Moneda *(ticket currency)*: cada usuario reparte *su* moneda entre sus procesos; el sistema la convierte a boletos globales.

- **Transferencia**: un cliente que espera a un servidor le presta sus boletos, para que lo atienda rápido; al terminar, los recupera.

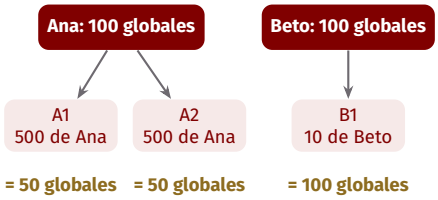
Moneda, transferencia e inflación



Moneda *(ticket currency)*: cada usuario reparte *su* moneda entre sus procesos; el sistema la convierte a boletos globales.

- **Transferencia**: un cliente que espera a un servidor le presta sus boletos, para que lo atienda rápido; al terminar, los recupera.
- **Inflación**: un proceso sube o baja sus propios boletos. Solo tiene sentido entre procesos que confían entre sí.

Moneda, transferencia e inflación



Moneda *(ticket currency)*: cada usuario reparte *su* moneda entre sus procesos; el sistema la convierte a boletos globales.

- **Transferencia**: un cliente que espera a un servidor le presta sus boletos, para que lo atienda rápido; al terminar, los recupera.
- **Inflación**: un proceso sube o baja sus propios boletos. Solo tiene sentido entre procesos que confían entre sí.
- ¿Cuántos boletos le doy a cada uno? La lotería no lo responde: es el problema abierto de la proporcionalidad.

SECCIÓN 02

Stride y el planificador de Linux

02



Stride: la lotería sin azar

Cada trabajo tiene stride $= L/$boletos y un contador `pass` en 0. En cada quantum corre el de menor `pass`, que luego suma su stride. En empate, el de menor índice.

Con $L = 60$ y boletos 3, 2, 1: strides **20**, **30**, **60**.



quantum	pass A	pass B	pass C	corre
1	0	0	0	A

Traza de `./stride 60 6 3 2 1`.

Stride: la lotería sin azar

Cada trabajo tiene stride $= L/$boletos y un contador `pass` en 0. En cada quantum corre el de menor `pass`, que luego suma su stride. En empate, el de menor índice.

Con $L = 60$ y boletos 3, 2, 1: strides **20**, **30**, **60**.



quantum	pass A	pass B	pass C	corre
1	0	0	0	A
2	20	0	0	B

Traza de `./stride 60 6 3 2 1`.

Stride: la lotería sin azar

Cada trabajo tiene stride $= L/$boletos y un contador `pass` en 0. En cada quantum corre el de menor `pass`, que luego suma su stride. En empate, el de menor índice.

Con $L = 60$ y boletos 3, 2, 1: strides **20**, **30**, **60**.



quantum	pass A	pass B	pass C	corre
1	0	0	0	A
2	20	0	0	B
3	20	30	0	C

Traza de `./stride 60 6 3 2 1`.

Stride: la lotería sin azar

Cada trabajo tiene stride $= L/$boletos y un contador `pass` en 0. En cada quantum corre el de menor `pass`, que luego suma su stride. En empate, el de menor índice.

Con $L = 60$ y boletos 3, 2, 1: strides **20**, **30**, **60**.



quantum	pass A	pass B	pass C	corre
1	0	0	0	A
2	20	0	0	B
3	20	30	0	C
4	20	30	60	A

Traza de `./stride 60 6 3 2 1`.

Stride: la lotería sin azar

Cada trabajo tiene stride $= L/$boletos y un contador `pass` en 0. En cada quantum corre el de menor `pass`, que luego suma su stride. En empate, el de menor índice.

Con $L = 60$ y boletos 3, 2, 1: strides **20**, **30**, **60**.



quantum	pass A	pass B	pass C	corre
1	0	0	0	A
2	20	0	0	B
3	20	30	0	C
4	20	30	60	A
5	40	30	60	B

Traza de `./stride 60 6 3 2 1`.

Stride: la lotería sin azar

Cada trabajo tiene stride $= L/$boletos y un contador `pass` en 0. En cada quantum corre el de menor `pass`, que luego suma su stride. En empate, el de menor índice.

Con $L = 60$ y boletos 3, 2, 1: strides **20**, **30**, **60**.



quantum	pass A	pass B	pass C	corre
1	0	0	0	A
2	20	0	0	B
3	20	30	0	C
4	20	30	60	A
5	40	30	60	B
6	40	60	60	A

Traza de `./stride 60 6 3 2 1`.

Stride: la lotería sin azar

Cada trabajo tiene stride $= L/$boletos y un contador `pass` en 0. En cada quantum corre el de menor `pass`, que luego suma su stride. En empate, el de menor índice.

Con $L = 60$ y boletos 3, 2, 1: strides **20**, **30**, **60**.



quantum	pass A	pass B	pass C	corre
1	0	0	0	A
2	20	0	0	B
3	20	30	0	C
4	20	30	60	A
5	40	30	60	B
6	40	60	60	A
7	60	60	60	…

Tras 6 quanta: A 3, B 2, C 1. Exacto, y todos los `pass` vuelven a igualarse.

Traza de `./stride 60 6 3 2 1`.

Lotería o stride

	Lotería	Stride
Proporción	Correcta en promedio	Exacta en cada ciclo

Lotería o stride

	Lotería	Stride
Proporción	Correcta en promedio	Exacta en cada ciclo
Estado global	Ninguno: solo boletos	Un `pass` por proceso

Lotería o stride

	Lotería	Stride
Proporción	Correcta en promedio	Exacta en cada ciclo
Estado global	Ninguno: solo boletos	Un `pass` por proceso
Proceso nuevo	Sumar sus boletos	¿Con qué `pass`? Si empieza en 0, acapara la CPU

Lotería o stride

	Lotería	**Stride**
Proporción	Correcta en promedio	Exacta en cada ciclo
Estado global	Ninguno: solo boletos	Un `pass` por proceso
Proceso nuevo	Sumar sus boletos	¿Con qué `pass`? Si empieza en 0, acapara la CPU
Costo de elegir	$O(n)$ con lista	$O(\log n)$ con cola de prioridad

Lotería o stride

	Lotería	Stride
Proporción	Correcta en promedio	Exacta en cada ciclo
Estado global	Ninguno: solo boletos	Un `pass` por proceso
Proceso nuevo	Sumar sus boletos	¿Con qué `pass`? Si empieza en 0, acapara la CPU
Costo de elegir	$O(n)$ con lista	$O(\log n)$ con cola de prioridad

La salida de stride

Al nuevo proceso se le da el `pass` mínimo actual. Linux hace lo mismo con su `vruntime`, como veremos ahora.

CFS: el planificador “completamente justo” de Linux

- Linux 2.6.23 (2007) a 6.5. Cada proceso acumula `vruntime`: su tiempo de CPU, en tiempo virtual.



CFS: el planificador “completamente justo” de Linux

- Linux 2.6.23 (2007) a 6.5. Cada proceso acumula `vruntime`: su tiempo de CPU, en tiempo virtual.
- Corre el de **menor** `vruntime`: el que menos recibió.

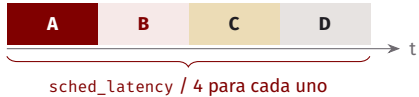


Es stride con otros nombres

`vruntime` hace de `pass`; el peso (`nice`) hace de boletos.

CFS: el planificador “completamente justo” de Linux

- Linux 2.6.23 (2007) a 6.5. Cada proceso acumula `vruntime`: su tiempo de CPU, en tiempo virtual.
- Corre el de **menor** `vruntime`: el que menos recibió.
- Slice: `sched_latency` repartido entre los *n* listos, con un mínimo (`min_granularity`).



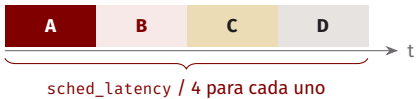
OSTEP usa 48 ms y 6 ms como ejemplo. En Linux real dependían del número de CPUs.

Es stride con otros nombres

`vruntime` hace de `pass`; el peso (`nice`) hace de boletos.

CFS: el planificador “completamente justo” de Linux

- Linux 2.6.23 (2007) a 6.5. Cada proceso acumula `vruntime`: su tiempo de CPU, en tiempo virtual.
- Corre el de **menor** `vruntime`: el que menos recibió.
- Slice: `sched_latency` repartido entre los *n* listos, con un mínimo (`min_granularity`).
- Al despertar: máx(su vruntime, mínimo). Dormir no da ventaja.



OSTEP usa 48 ms y 6 ms como ejemplo. En Linux real dependían del número de CPUs.

Es stride con otros nombres

`vruntime` hace de `pass`; el peso (`nice`) hace de boletos.

Pesos y `nice`: un ejemplo con números

`nice` va de -20 a 19 y se traduce en un peso. Cada nivel cambia el peso en un factor de $\approx 1{,}25$:

`nice`	-5	0	1
peso	3121	**1024**	820
`nice`	5	10	19
peso	335	110	15

$$\texttt{vruntime}_i \mathrel{+}= \text{runtime}_i \cdot \frac{1024}{\text{peso}_i}$$

A con `nice` 0 (1024), B con `nice` 1 (820). Periodo de 48 ms.

Pesos: `sched_prio_to_weight` en `kernel/sched/core.c` de Linux.

Pesos y `nice`: un ejemplo con números

`nice` va de -20 a 19 y se traduce en un peso. Cada nivel cambia el peso en un factor de $\approx 1{,}25$:

`nice`	-5	0	1
peso	3121	**1024**	820
`nice`	5	10	19
peso	335	110	15

$$\texttt{vruntime}_i \mathrel{+}= \text{runtime}_i \cdot \frac{1024}{\text{peso}_i}$$

A con `nice` 0 (1024), B con `nice` 1 (820). Periodo de 48 ms.

1. Porción de A: $48 \cdot \frac{1024}{1844} = 26{,}7$ ms (55,5 %).

Pesos: `sched_prio_to_weight` en `kernel/sched/core.c` de Linux.

Pesos y `nice`: un ejemplo con números

`nice` va de -20 a 19 y se traduce en un peso. Cada nivel cambia el peso en un factor de $\approx 1{,}25$:

`nice`	-5	0	1
peso	3121	**1024**	820

`nice`	5	10	19
peso	335	110	15

$$\texttt{vruntime}_i \mathrel{+}= \text{runtime}_i \cdot \frac{1024}{\text{peso}_i}$$

A con `nice` 0 (1024), B con `nice` 1 (820). Periodo de 48 ms.

1. Porción de A: $48 \cdot \frac{1024}{1844} = 26{,}7$ ms (55,5 %).
2. Porción de B: $48 \cdot \frac{820}{1844} = 21{,}3$ ms (44,5 %).

Pesos: `sched_prio_to_weight` en `kernel/sched/core.c` de Linux.

Pesos y `nice`: un ejemplo con números

`nice` va de -20 a 19 y se traduce en un peso. Cada nivel cambia el peso en un factor de $\approx 1{,}25$:

`nice`	-5	0	1
peso	3121	**1024**	820
`nice`	5	10	19
peso	335	110	15

$$\texttt{vruntime}_i \mathrel{+}= \text{runtime}_i \cdot \frac{1024}{\text{peso}_i}$$

A con `nice` 0 (1024), B con `nice` 1 (820). Periodo de 48 ms.

1. Porción de A: $48 \cdot \frac{1024}{1844} = 26{,}7$ ms (55,5 %).
2. Porción de B: $48 \cdot \frac{820}{1844} = 21{,}3$ ms (44,5 %).
3. `vruntime` de A: $26{,}7 \cdot \frac{1024}{1024} = 26{,}7$.

Pesos: `sched_prio_to_weight` en `kernel/sched/core.c` de Linux.

Pesos y `nice`: un ejemplo con números

`nice` va de -20 a 19 y se traduce en un peso. Cada nivel cambia el peso en un factor de $\approx 1{,}25$:

`nice`	-5	0	1
peso	3121	**1024**	820

`nice`	5	10	19
peso	335	110	15

$$\texttt{vruntime}_i \mathrel{+}= \text{runtime}_i \cdot \frac{1024}{\text{peso}_i}$$

A con `nice` 0 (1024), B con `nice` 1 (820). Periodo de 48 ms.

1. Porción de A: $48 \cdot \frac{1024}{1844} = 26{,}7$ ms (55,5 %).
2. Porción de B: $48 \cdot \frac{820}{1844} = 21{,}3$ ms (44,5 %).
3. `vruntime` de A: $26{,}7 \cdot \frac{1024}{1024} = 26{,}7$.
4. `vruntime` de B: $21{,}3 \cdot \frac{1024}{820} = 26{,}7$.

Pesos: `sched_prio_to_weight` en `kernel/sched/core.c` de Linux.

Pesos y `nice`: un ejemplo con números

`nice` va de -20 a 19 y se traduce en un peso. Cada nivel cambia el peso en un factor de $\approx 1{,}25$:

`nice`	-5	0	1
peso	3121	**1024**	820
`nice`	5	10	19
peso	335	110	15

$$\texttt{vruntime}_i \mathrel{+}= \text{runtime}_i \cdot \frac{1024}{\text{peso}_i}$$

A con `nice` 0 (1024), B con `nice` 1 (820). Periodo de 48 ms.

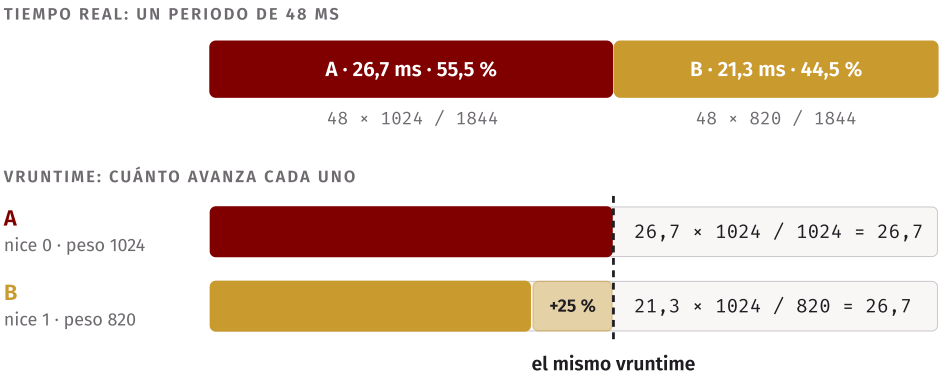
1. Porción de A: $48 \cdot \frac{1024}{1844} = 26{,}7$ ms (55,5 %).
2. Porción de B: $48 \cdot \frac{820}{1844} = 21{,}3$ ms (44,5 %).
3. `vruntime` de A: $26{,}7 \cdot \frac{1024}{1024} = 26{,}7$.
4. `vruntime` de B: $21{,}3 \cdot \frac{1024}{820} = 26{,}7$.

Lo que se iguala es el `vruntime`

El tiempo real se reparte según los pesos, pero el tiempo virtual avanza igual para todos.

Pesos: `sched_prio_to_weight` en `kernel/sched/core.c` de Linux.

Tiempo real y tiempo virtual



El tiempo real se reparte según los pesos; el vruntime avanza igual para los dos.

El mismo ejemplo, simulado

bash

```
$ ./cfs 30 0 1
ms  1: corre A   vruntime:  A=1.00  B=0.00
ms  2: corre B   vruntime:  A=1.00  B=1.25
ms  3: corre A   vruntime:  A=2.00  B=1.25
...
ms 11: corre A   vruntime:  A=6.00  B=6.24
ms 12: corre A   vruntime:  A=7.00  B=6.24
$ ./cfs 1000 0 5 | tail -2
   A       0   1024     75.3%     75.3%
   B       5    335     24.7%     24.7%
```

- Quantum fijo de 1 ms y búsqueda lineal: una simplificación.

El mismo ejemplo, simulado

bash

```
$ ./cfs 30 0 1
ms  1: corre A   vruntime:  A=1.00  B=0.00
ms  2: corre B   vruntime:  A=1.00  B=1.25
ms  3: corre A   vruntime:  A=2.00  B=1.25
...
ms 11: corre A   vruntime:  A=6.00  B=6.24
ms 12: corre A   vruntime:  A=7.00  B=6.24
$ ./cfs 1000 0 5 | tail -2
   A       0   1024     75.3%     75.3%
   B       5    335     24.7%     24.7%
```

- Quantum fijo de 1 ms y búsqueda lineal: una simplificación.
- B suma 1,25 por ms; A suma 1. Por eso A corre dos veces seguidas de vez en cuando.

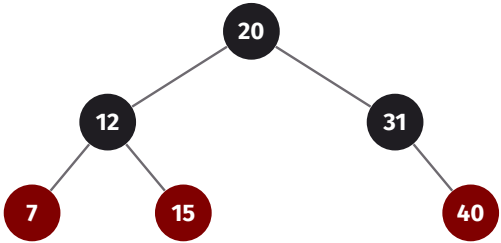
El mismo ejemplo, simulado

bash

```
$ ./cfs 30 0 1
ms  1: corre A   vruntime:  A=1.00  B=0.00
ms  2: corre B   vruntime:  A=1.00  B=1.25
ms  3: corre A   vruntime:  A=2.00  B=1.25
...
ms 11: corre A   vruntime:  A=6.00  B=6.24
ms 12: corre A   vruntime:  A=7.00  B=6.24
$ ./cfs 1000 0 5 | tail -2
   A       0   1024     75.3%     75.3%
   B       5    335     24.7%     24.7%
```

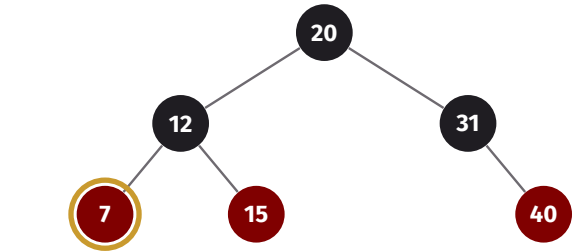
- Quantum fijo de 1 ms y búsqueda lineal: una simplificación.
- B suma 1,25 por ms; A suma 1. Por eso A corre dos veces seguidas de vez en cuando.
- Con `nice` 5, B recibe la cuarta parte.

¿Cómo encontrar el menor rápido? Un árbol rojo-negro



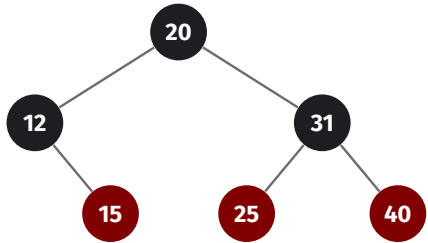
- Los procesos *listos* se ordenan por `vruntime` (los que duermen no están).

¿Cómo encontrar el menor rápido? Un árbol rojo-negro



- Los procesos *listos* se ordenan por `vruntime` (los que duermen no están).
- El próximo es el nodo de más a la izquierda (Linux lo guarda en caché).

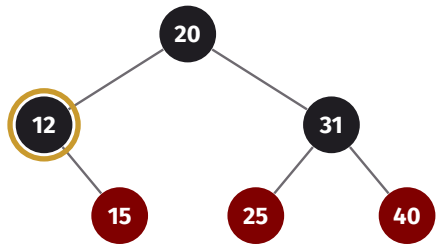
¿Cómo encontrar el menor rápido? Un árbol rojo-negro



vuelve con 25

- Los procesos *listos* se ordenan por `vruntime` (los que duermen no están).
- El próximo es el nodo de más a la izquierda (Linux lo guarda en caché).
- Corre, su `vruntime` sube de 7 a 25 y se reinserta: $O(\log n)$.

¿Cómo encontrar el menor rápido? Un árbol rojo-negro



vuelve con 25

- Los procesos *listos* se ordenan por `vruntime` (los que duermen no están).
- El próximo es el nodo de más a la izquierda (Linux lo guarda en caché).
- Corre, su `vruntime` sube de 7 a 25 y se reinserta: $O(\log n)$.
- El siguiente es 12.

Desde Linux 6.6: EEVDF

- Desde **6.6** (octubre de 2023), la clase *fair* (`SCHED_NORMAL`) elige con EEVDF: *Earliest Eligible Virtual Deadline First* (Stoica, 1995).

`kernel/sched/fair.c` de Linux; documentación `scheduler/sched-eevdf`.

Desde Linux 6.6: EEVDF

- Desde **6.6** (octubre de 2023), la clase *fair* (`SCHED_NORMAL`) elige con EEVDF: *Earliest Eligible Virtual Deadline First* (Stoica, 1995).
- Se conservan pesos, `nice`, `vruntime` y el árbol rojo-negro.

`kernel/sched/fair.c` de Linux; documentación `scheduler/sched-eevdf`.

Desde Linux 6.6: EEVDF

- Desde **6.6** (octubre de 2023), la clase *fair* (`SCHED_NORMAL`) elige con EEVDF: *Earliest Eligible Virtual Deadline First* (Stoica, 1995).
- Se conservan pesos, `nice`, `vruntime` y el árbol rojo-negro.
- **Elegible**: su *lag* (lo que se le debe) es ≥ 0; su `vruntime` no supera el promedio ponderado.

`kernel/sched/fair.c` de Linux; documentación `scheduler/sched-eevdf`.

Desde Linux 6.6: EEVDF

- Desde **6.6** (octubre de 2023), la clase *fair* (`SCHED_NORMAL`) elige con EEVDF: *Earliest Eligible Virtual Deadline First* (Stoica, 1995).
- Se conservan pesos, `nice`, `vruntime` y el árbol rojo-negro.
- **Elegible**: su *lag* (lo que se le debe) es ≥ 0; su `vruntime` no supera el promedio ponderado.
- **Deadline virtual** $=$ `vruntime` $+$ slice $\cdot\, 1024/$peso. Corre el elegible con deadline más temprano.

`kernel/sched/fair.c` de Linux; documentación `scheduler/sched-eevdf`.

Desde Linux 6.6: EEVDF

- Desde **6.6** (octubre de 2023), la clase *fair* (`SCHED_NORMAL`) elige con EEVDF: *Earliest Eligible Virtual Deadline First* (Stoica, 1995).
- Se conservan pesos, `nice`, `vruntime` y el árbol rojo-negro.
- **Elegible**: su *lag* (lo que se le debe) es ≥ 0; su `vruntime` no supera el promedio ponderado.
- **Deadline virtual** $=$ `vruntime` $+$ slice $\cdot\, 1024/\text{peso}$. Corre el elegible con deadline más temprano.

Qué cambió

Ya no hay `sched_latency` ni `min_granularity`: queda un slice base ($\approx 0{,}7$ ms, escalado por el número de CPUs). En versiones posteriores, un proceso puede pedir un slice más corto con `sched_setattr()`.

`kernel/sched/fair.c` de Linux; documentación `scheduler/sched-eevdf`.

Desde Linux 6.6: EEVDF

- Desde **6.6** (octubre de 2023), la clase *fair* (`SCHED_NORMAL`) elige con EEVDF: *Earliest Eligible Virtual Deadline First* (Stoica, 1995).
- Se conservan pesos, `nice`, `vruntime` y el árbol rojo-negro.
- **Elegible**: su *lag* (lo que se le debe) es ≥ 0; su `vruntime` no supera el promedio ponderado.
- **Deadline virtual** $=$ `vruntime` $+$ slice $\cdot\, 1024/\text{peso}$. Corre el elegible con deadline más temprano.

Qué cambió

Ya no hay `sched_latency` ni `min_granularity`: queda un slice base ($\approx 0{,}7$ ms, escalado por el número de CPUs). En versiones posteriores, un proceso puede pedir un slice más corto con `sched_setattr()`.

CFS sigue siendo el modelo para entender la idea: justicia por tiempo virtual ponderado.

`kernel/sched/fair.c` de Linux; documentación `scheduler/sched-eevdf`.

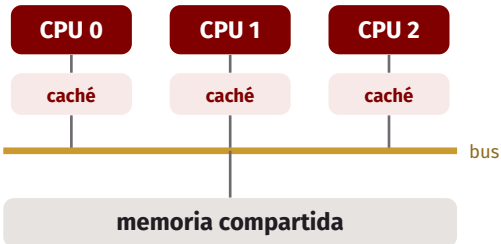
SECCIÓN 03

Planificación con varios CPUs

03

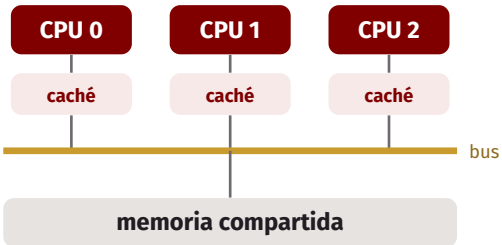


Varios CPUs, cada uno con su caché



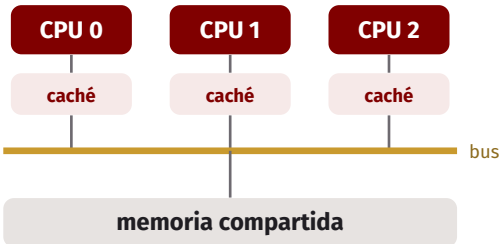
- La caché es pequeña y rápida (unos pocos ns); la memoria, grande y lenta (decenas de ns o más).

Varios CPUs, cada uno con su caché



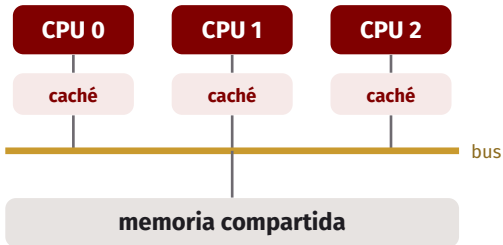
- La caché es pequeña y rápida (unos pocos ns); la memoria, grande y lenta (decenas de ns o más).
- Funciona por localidad: temporal (vuelvo a usar el mismo dato) y espacial (uso el dato vecino).

Varios CPUs, cada uno con su caché



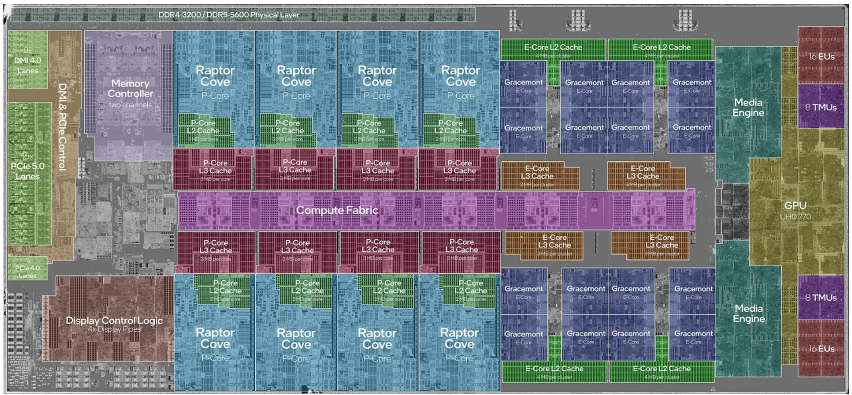
- La caché es pequeña y rápida (unos pocos ns); la memoria, grande y lenta (decenas de ns o más).
- Funciona por localidad: temporal (vuelvo a usar el mismo dato) y espacial (uso el dato vecino).
- Se mueve en **líneas** de 64 bytes, no en bytes sueltos.

Varios CPUs, cada uno con su caché



- La caché es pequeña y rápida (unos pocos ns); la memoria, grande y lenta (decenas de ns o más).
- Funciona por localidad: temporal (vuelvo a usar el mismo dato) y espacial (uso el dato vecino).
- Se mueve en **líneas** de 64 bytes, no en bytes sueltos.
- Problema nuevo: el mismo dato puede estar copiado en *varias* cachés.

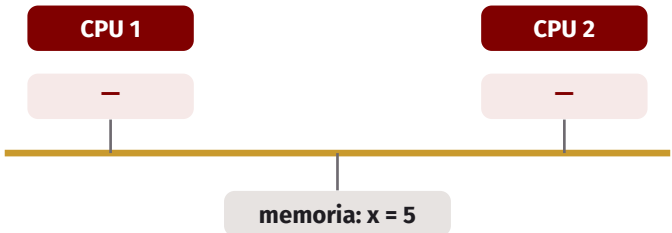
Así se ve un chip real



Intel Core i9-13900K. Fritzchens Fritz, rótulos de JmsDoug, CC0, Wikimedia Commons.

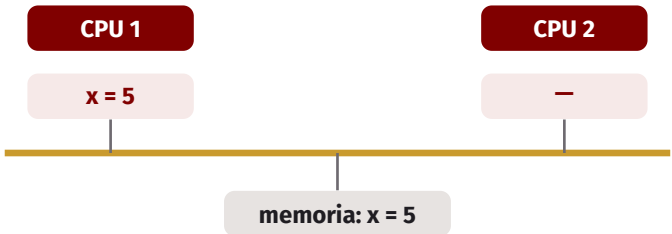
Cada núcleo P (azul) tiene su caché L2 (verde); la L3 está repartida en slices junto a la red central.

Coherencia de caché, paso a paso



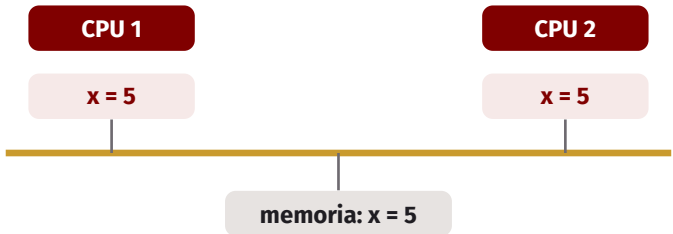
Inicialmente `x = 5` está solo en memoria.

Coherencia de caché, paso a paso



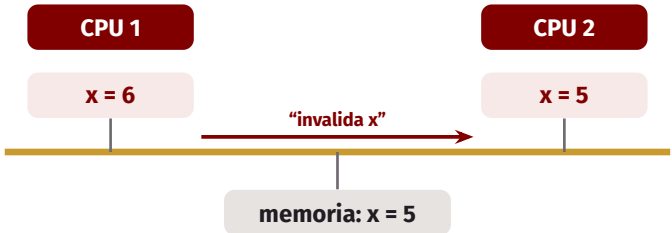
CPU 1 lee x: una copia queda en su caché.

Coherencia de caché, paso a paso



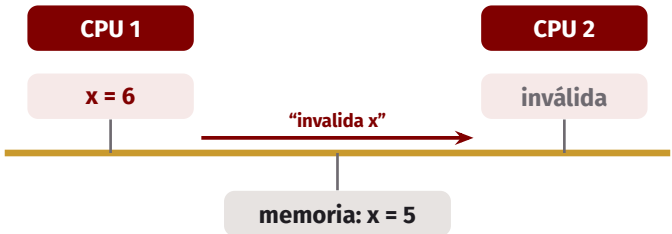
CPU 2 lee x: ahora hay dos copias. Mientras solo se lea, no hay problema.

Coherencia de caché, paso a paso



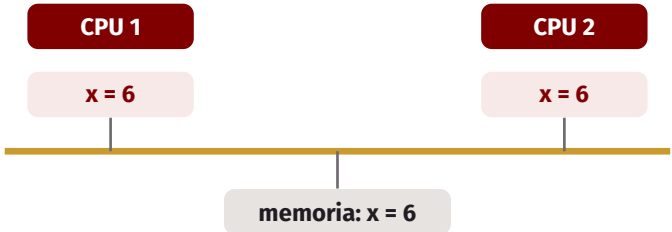
CPU 1 escribe `x = 6`. Si nadie avisa, CPU 2 seguiría leyendo 5.

Coherencia de caché, paso a paso



Las cachés espían el bus *(bus snooping)*: CPU 2 ve la escritura e invalida su copia.

Coherencia de caché, paso a paso



Cuando CPU 2 vuelve a leer, falla en su caché y obtiene el valor nuevo. El hardware garantiza la coherencia.

La coherencia no es gratis

falseshare.c

```c
// ALIGNED alinea a LINE = 64.
// Juntos: en la misma línea de caché.
static struct {
    long a;
    long b;
} ALIGNED together;

// Separados: cada uno en su línea.
static struct {
    long a;
    char pad[LINE - sizeof(long)];
    long b;
} ALIGNED apart;
```

bash

```
$ ./falseshare 100
misma línea:      2.156 s
líneas distintas: 0.248 s
relación:         8.7x
```

Dos hilos suman cada uno a *su* contador. Comparten la línea, no los datos: cada suma invalida la copia del otro. Es false sharing.

(Varía con la carga: medimos de 1,4x a 8,7x.)

La coherencia no es gratis

falseshare.c

```c
// ALIGNED alinea a LINE = 64.
// Juntos: en la misma línea de caché.
static struct {
    long a;
    long b;
} ALIGNED together;

// Separados: cada uno en su línea.
static struct {
    long a;
    char pad[LINE - sizeof(long)];
    long b;
} ALIGNED apart;
```

bash

```
$ ./falseshare 100
misma línea:      2.156 s
líneas distintas: 0.248 s
relación:         8.7x
```

Dos hilos suman cada uno a *su* contador. Comparten la línea, no los datos: cada suma invalida la copia del otro. Es false sharing.

(Varía con la carga: medimos de 1,4x a 8,7x.)

Si comparten el dato, la coherencia no basta: hace falta un lock o una instrucción atómica.

Afinidad de caché

- Un proceso que corrió en CPU 2 dejó *su* estado en la caché de CPU 2.

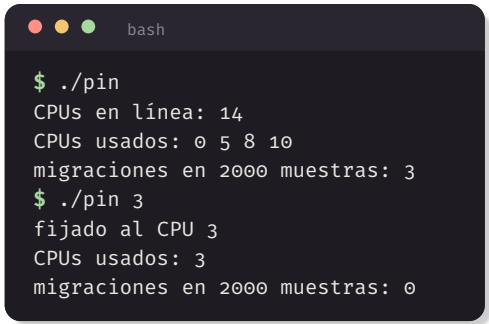
bash

```
$ ./pin
CPUs en línea: 14
CPUs usados: 0 5 8 10
migraciones en 2000 muestras: 3
$ ./pin 3
fijado al CPU 3
CPUs usados: 3
migraciones en 2000 muestras: 0
```

28 hilos de relleno compiten por 14 CPUs durante 1 s.

Afinidad de caché

- Un proceso que corrió en CPU 2 dejó *su* estado en la caché de CPU 2.
- Si lo muevo a CPU 0, no hay que copiar nada… pero arranca con la caché fría: cada acceso falla hasta recargarla.



28 hilos de relleno compiten por 14 CPUs durante 1 s.

Afinidad de caché

- Un proceso que corrió en CPU 2 dejó *su* estado en la caché de CPU 2.
- Si lo muevo a CPU 0, no hay que copiar nada… pero arranca con la caché fría: cada acceso falla hasta recargarla.
- Afinidad *(cache affinity)*: preferir el CPU donde corrió antes.

bash

```
$ ./pin
CPUs en línea: 14
CPUs usados: 0 5 8 10
migraciones en 2000 muestras: 3
$ ./pin 3
fijado al CPU 3
CPUs usados: 3
migraciones en 2000 muestras: 0
```

28 hilos de relleno compiten por 14 CPUs durante 1 s.

Afinidad de caché

- Un proceso que corrió en CPU 2 dejó *su* estado en la caché de CPU 2.
- Si lo muevo a CPU 0, no hay que copiar nada… pero arranca con la caché fría: cada acceso falla hasta recargarla.
- Afinidad *(cache affinity)*: preferir el CPU donde corrió antes.
- Linux deja fijarla a mano: `sched_setaffinity()` o `taskset`.

bash

```
$ ./pin
CPUs en línea: 14
CPUs usados: 0 5 8 10
migraciones en 2000 muestras: 3
$ ./pin 3
fijado al CPU 3
CPUs usados: 3
migraciones en 2000 muestras: 0
```

28 hilos de relleno compiten por 14 CPUs durante 1 s.

Una sola cola: SQMS

Single-queue multiprocessor scheduling: todos los CPUs toman trabajo de una misma cola. Cuatro trabajos, tres CPUs:



- Simple y balancea solo.

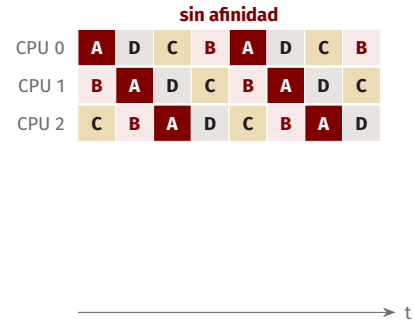


Una sola cola: SQMS

Single-queue multiprocessor scheduling: todos los CPUs toman trabajo de una misma cola. Cuatro trabajos, tres CPUs:



- Simple y balancea solo.
- Pero la cola necesita un lock: con muchos CPUs, todos esperan por él.
- Y los trabajos saltan de CPU en CPU: sin afinidad.
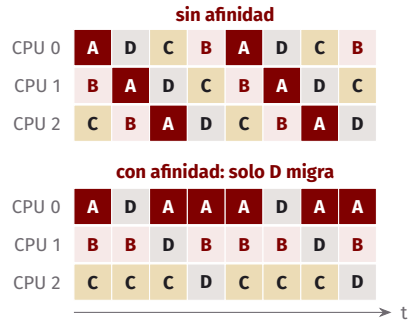


Una sola cola: SQMS

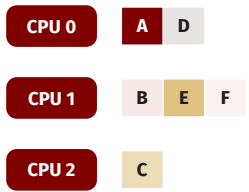
Single-queue multiprocessor scheduling: todos los CPUs toman trabajo de una misma cola. Cuatro trabajos, tres CPUs:



- Simple y balancea solo.
- Pero la cola necesita un lock: con muchos CPUs, todos esperan por él.
- Y los trabajos saltan de CPU en CPU: sin afinidad.
- Arreglo parcial: mantener a casi todos fijos y mover solo a uno.

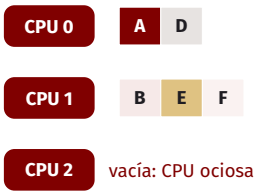


Una cola por CPU: MQMS



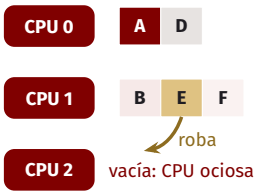
- Cada CPU tiene su cola y su lock: escala y respeta la afinidad.

Una cola por CPU: MQMS



- Cada CPU tiene su cola y su lock: escala y respeta la afinidad.
- Problema: desbalance. Termina C y CPU 2 queda sin trabajo mientras CPU 1 tiene tres.

Una cola por CPU: MQMS



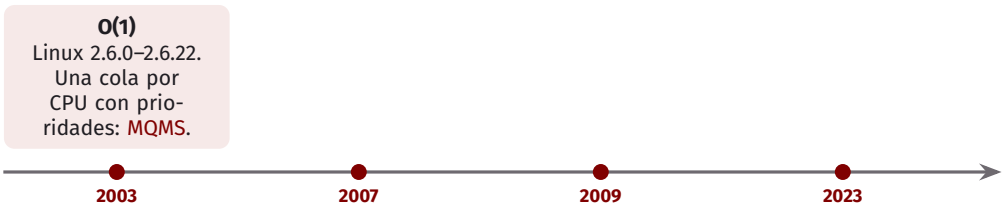
- Cada CPU tiene su cola y su lock: escala y respeta la afinidad.
- Problema: desbalance. Termina C y CPU 2 queda sin trabajo mientras CPU 1 tiene tres.
- Solución: **migrar**. En el robo de trabajo *(work stealing)*, la cola con poco trabajo mira a otra y se lleva uno.

Una cola por CPU: MQMS

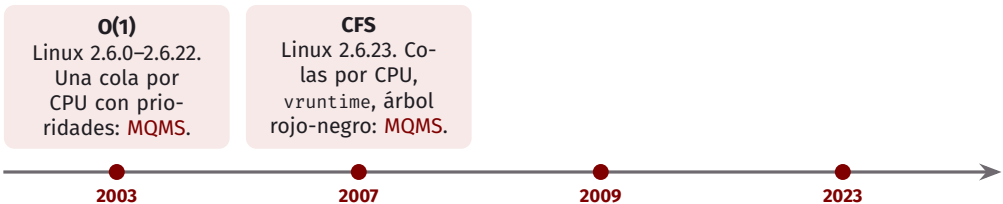


- Cada CPU tiene su cola y su lock: escala y respeta la afinidad.
- Problema: desbalance. Termina C y CPU 2 queda sin trabajo mientras CPU 1 tiene tres.
- Solución: **migrar**. En el robo de trabajo *(work stealing)*, la cola con poco trabajo mira a otra y se lleva uno.
- ¿Cada cuánto mirar? Mucho: overhead. Poco: desbalance.

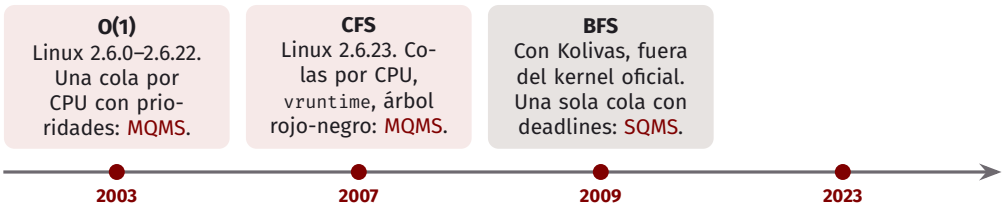
Los planificadores de Linux



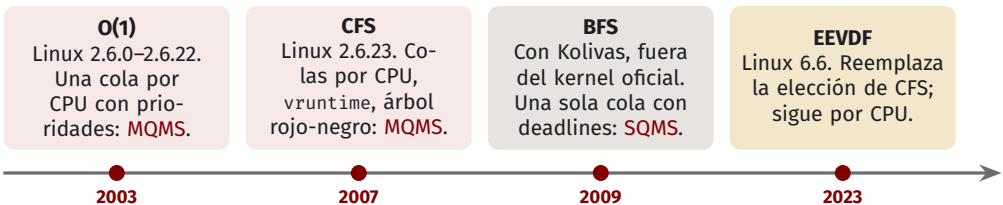
Los planificadores de Linux



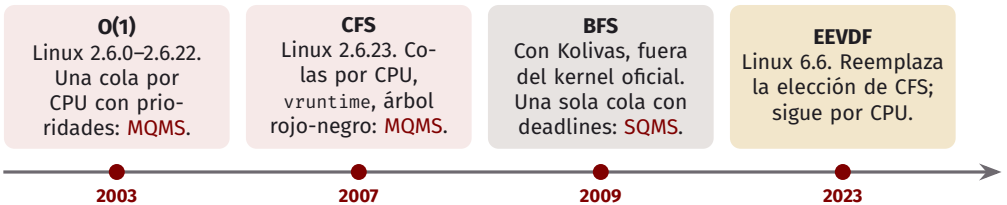
Los planificadores de Linux



Los planificadores de Linux



Los planificadores de Linux



No hay un ganador universal: Linux equilibra las colas por CPU con migraciones periódicas entre dominios (núcleos, sockets).

Cierre de la virtualización de la CPU

Mecanismos (semanas 1–2)

Procesos, modo usuario y kernel, traps, interrupción del temporizador, cambio de contexto. Así el kernel *puede* quitar la CPU.

Políticas (semanas 3–4)

FIFO, SJF, STCF, RR, MLFQ, lotería, stride, CFS, y su versión con varios CPUs. Así el kernel *decide* a quién dársela.

Cierre de la virtualización de la CPU

Mecanismos (semanas 1–2)

Procesos, modo usuario y kernel, traps, interrupción del temporizador, cambio de contexto. Así el kernel *puede* quitar la CPU.

Políticas (semanas 3–4)

FIFO, SJF, STCF, RR, MLFQ, lotería, stride, CFS, y su versión con varios CPUs. Así el kernel *decide* a quién dársela.

La lección del diálogo (cap. 11)

No hay política perfecta: cada una optimiza una métrica (retorno, respuesta, proporción) a costa de otra. Y el kernel vive en un mundo de casos reales, no de supuestos limpios.

SECCIÓN 04

Primer contacto con xv6-riscv

04

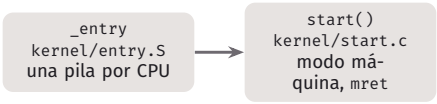


Cómo arranca xv6

`_entry`
`kernel/entry.S`
una pila por CPU

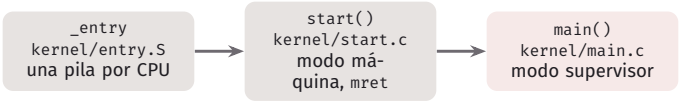
QEMU carga el kernel en `0x80000000` y *todos* los CPUs (harts) saltan ahí.

Cómo arranca xv6



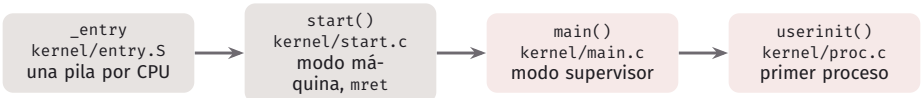
En modo máquina: delega traps al supervisor, programa el temporizador y “retorna” a `main` con `mret`.

Cómo arranca xv6



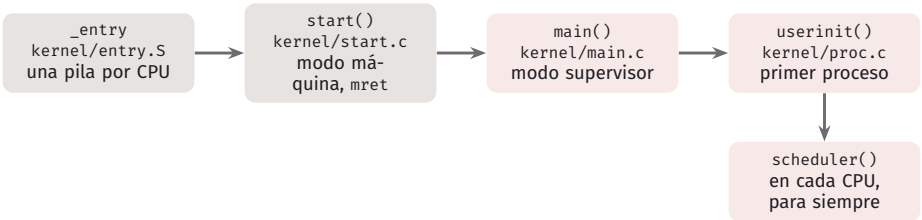
El CPU 0 inicializa memoria, tablas de páginas, procesos, disco… Los demás esperan la bandera `started`.

Cómo arranca xv6



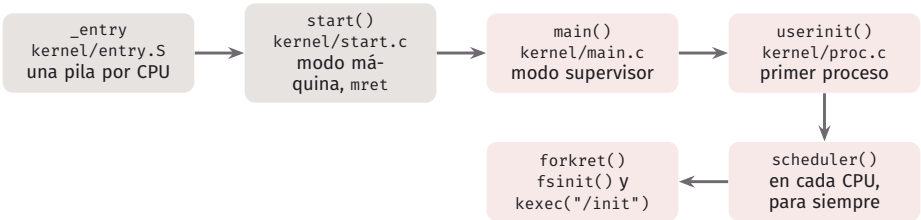
Crea el primer proceso, sin programa todavía, y lo marca `RUNNABLE`.

Cómo arranca xv6



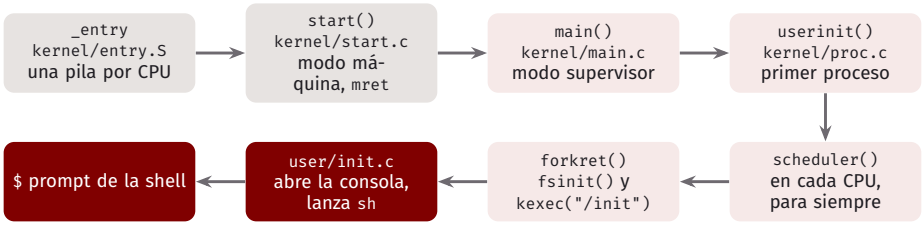
Cada CPU entra a su bucle de planificación (Round Robin sobre `proc[]`).

Cómo arranca xv6



La primera vez que ese proceso corre, monta el sistema de archivos y carga `/init`.

Cómo arranca xv6



Desde aquí, todo es código de usuario. Lo verán con `make qemu`.

El viaje de una llamada al sistema

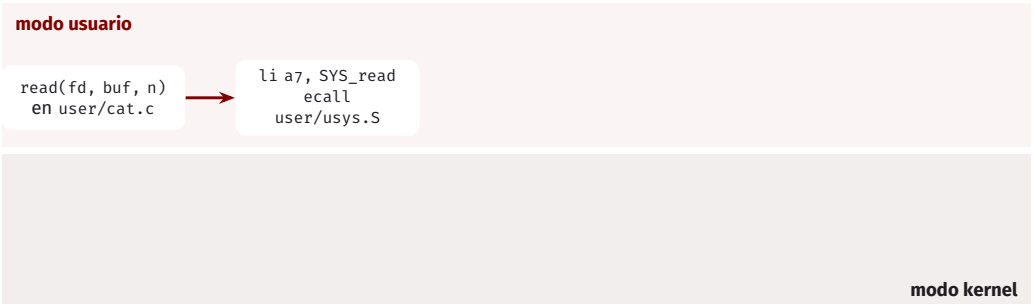
modo usuario

`read(fd, buf, n)` en `user/cat.c`

modo kernel

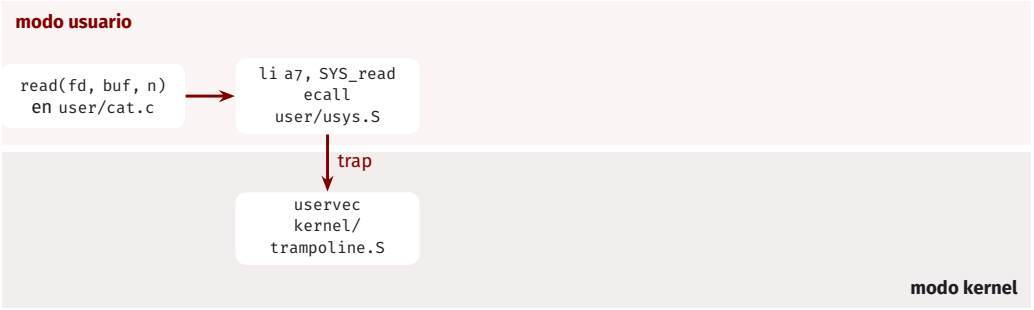
Un programa llama a `read()` como si fuera una función más.

El viaje de una llamada al sistema



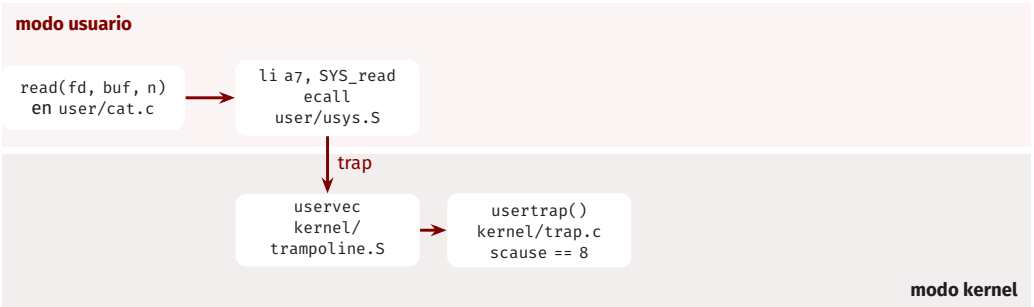
El stub generado pone el número de la llamada en `a7` y ejecuta `ecall`.

El viaje de una llamada al sistema



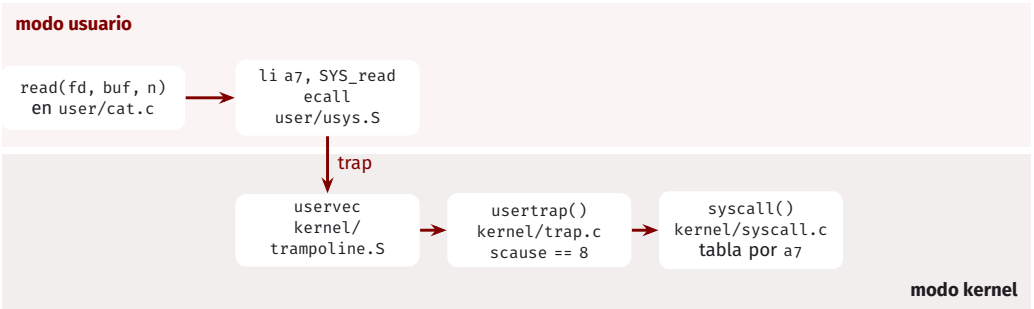
El hardware salta a la dirección de `stvec`, que apunta a `uservec`: guarda los registros en el *trapframe*.

El viaje de una llamada al sistema



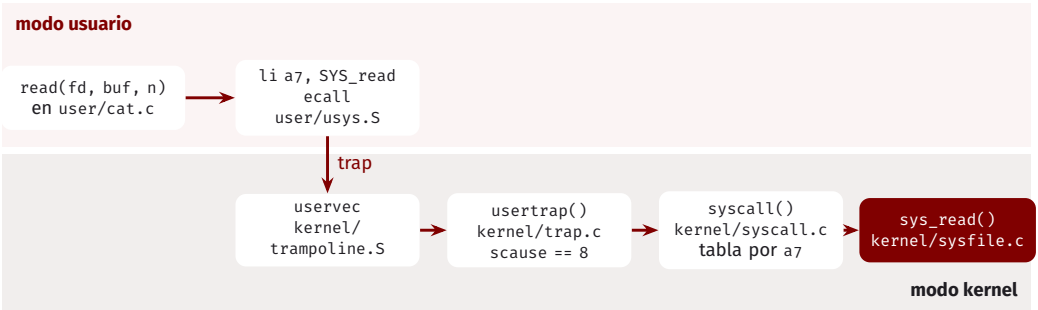
Ya en C: ¿fue una llamada al sistema, una interrupción o un fallo de página?

El viaje de una llamada al sistema



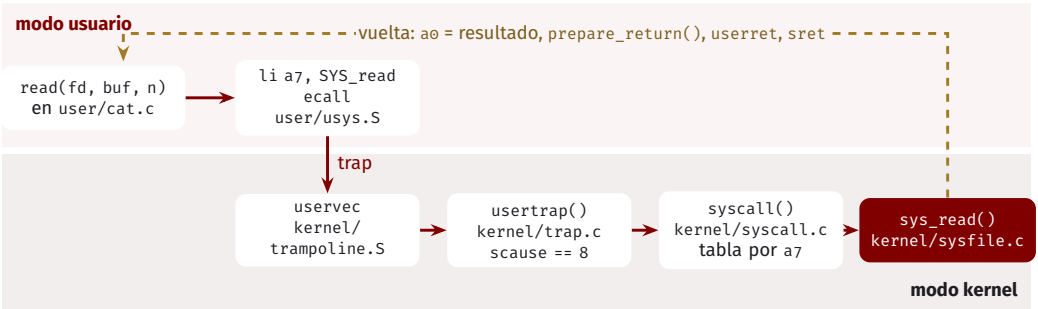
Lee `a7` del trapframe y busca la función en el arreglo `syscalls[]`.

El viaje de una llamada al sistema



La función real lee sus argumentos del trapframe (`argint`, `argaddr`).

El viaje de una llamada al sistema



El resultado viaja en `a0` del trapframe; `sret` vuelve a la instrucción siguiente al `ecall`.

Del lado del usuario: `user/usys.pl`

user/usys.pl

```
1  sub entry {
2      my $prefix = "sys_";
3      my $name = shift;
4      # ... (caso especial de sbrk)
5      print ".global $name\n";
6      print "$name:\n";
7      print " li a7, SYS_${name}\n";
8      print " ecall\n";
9      print " ret\n";
10 }
11
12 entry("fork");
13 entry("exit");
14 # ... una línea por llamada
```

Un script en Perl genera `user/usys.S`, con un stub de tres instrucciones por llamada:

user/usys.S (generado)

```
1  .global read
2  read:
3   li a7, SYS_read
4   ecall
5   ret
```

`SYS_read` vale 5 (`kernel/syscall.h`). Los argumentos ya están en `a0`–`a2` por la convención de llamadas de RISC-V.

Entrar al kernel: uservec

kernel/trampoline.S

```
1  uservec:
2    csrw sscratch, a0
3    li a0, TRAPFRAME
4    sd ra, 40(a0)
5    sd sp, 48(a0)
6    # ... los otros registros ...
7    ld sp, 8(a0)   # kernel_sp
8    ld tp, 32(a0)  # kernel_hartid
9    ld t0, 16(a0)  # usertrap
10   ld t1, 0(a0)   # kernel_satp
11   sfence.vma zero, zero
12   csrw satp, t1  # tabla del kernel
13   sfence.vma zero, zero
14   jalr t0        # usertrap()
```

- El trapframe es una página por proceso, mapeada en la misma dirección (TRAPFRAME) en cada espacio de usuario.

Entrar al kernel: uservec

kernel/trampoline.S

```
1  uservec:
2    csrw sscratch, a0
3    li a0, TRAPFRAME
4    sd ra, 40(a0)
5    sd sp, 48(a0)
6    # ... los otros registros ...
7    ld sp, 8(a0)   # kernel_sp
8    ld tp, 32(a0)  # kernel_hartid
9    ld t0, 16(a0)  # usertrap
10   ld t1, 0(a0)   # kernel_satp
11   sfence.vma zero, zero
12   csrw satp, t1  # tabla del kernel
13   sfence.vma zero, zero
14   jalr t0        # usertrap()
```

- El trapframe es una página por proceso, mapeada en la misma dirección (TRAPFRAME) en cada espacio de usuario.
- Se cambia a la pila del kernel y a la tabla de páginas del kernel.

Entrar al kernel: uservec

kernel/trampoline.S

```
1  uservec:
2    csrw sscratch, a0
3    li a0, TRAPFRAME
4    sd ra, 40(a0)
5    sd sp, 48(a0)
6    # ... los otros registros ...
7    ld sp, 8(a0)   # kernel_sp
8    ld tp, 32(a0)  # kernel_hartid
9    ld t0, 16(a0)  # usertrap
10   ld t1, 0(a0)   # kernel_satp
11   sfence.vma zero, zero
12   csrw satp, t1  # tabla del kernel
13   sfence.vma zero, zero
14   jalr t0        # usertrap()
```

- El trapframe es una página por proceso, mapeada en la misma dirección (TRAPFRAME) en cada espacio de usuario.
- Se cambia a la pila del kernel y a la tabla de páginas del kernel.
- Por eso este código vive en la página *trampolín*: está en la misma dirección virtual en el espacio del usuario y en el del kernel.

En C: `usertrap()` y `syscall()`

kernel/trap.c

```
1  uint64
2  usertrap(void)
3  {
4    // ...
5    p->trapframe->epc = r_sepc();
6    if (r_scause() == 8) {
7      // system call
8      if (killed(p))
9        kexit(-1);
10     p->trapframe->epc += 4;
11     intr_on();
12     syscall();
13   } // else: interrupción o fallo
```

kernel/syscall.c

```
1  void
2  syscall(void)
3  {
4    int num;
5    struct proc *p = myproc();
6    num = p->trapframe->a7;
7    if (num > 0 && num < NELEM(syscalls)
8        && syscalls[num]) {
9      p->trapframe->a0 = syscalls[num]();
10   } else {
11     // printk("... unknown sys call")
12     p->trapframe->a0 = -1;
13   }
14 }
```

Verlo con gdb

bash

```
$ make qemu-gdb CPUS=1         # terminal 1
$ gdb-multiarch kernel/kernel  # terminal 2
(gdb) break syscall
(gdb) continue
syscall () at kernel/syscall.c:140
(gdb) next 3
(gdb) p p->trapframe->a7
= 15
(gdb) bt
#0  syscall () at kernel/syscall.c:145
#1  usertrap () at kernel/trap.c:68
```

- La primera llamada al sistema de todo xv6: `init` abre la consola.

Salida abreviada: gdb imprime `Breakpoint 1`, y `$1 = 15`.

Verlo con gdb

bash

```
$ make qemu-gdb CPUS=1          # terminal 1
$ gdb-multiarch kernel/kernel  # terminal 2
(gdb) break syscall
(gdb) continue
syscall () at kernel/syscall.c:140
(gdb) next 3
(gdb) p p->trapframe->a7
= 15
(gdb) bt
#0  syscall () at kernel/syscall.c:145
#1  usertrap () at kernel/trap.c:68
```

- La primera llamada al sistema de todo xv6: `init` abre la consola.
- 15 es `SYS_open` en `kernel/syscall.h`.

Salida abreviada: gdb imprime `Breakpoint 1,` y `$1 = 15`.

Verlo con gdb

bash

```
$ make qemu-gdb CPUS=1          # terminal 1
$ gdb-multiarch kernel/kernel  # terminal 2
(gdb) break syscall
(gdb) continue
syscall () at kernel/syscall.c:140
(gdb) next 3
(gdb) p p->trapframe->a7
= 15
(gdb) bt
#0  syscall () at kernel/syscall.c:145
#1  usertrap () at kernel/trap.c:68
```

- La primera llamada al sistema de todo xv6: `init` abre la consola.
- 15 es `SYS_open` en `kernel/syscall.h`.
- `CPUS=1` evita que otros CPUs se crucen mientras depuran.

Salida abreviada: gdb imprime `Breakpoint 1,` y `$1 = 15`.

Agregar `getreadcount()`: el kernel

kernel/sysfile.c

```
+// Llamadas a read() desde el arranque.
+uint64 readcount = 0;
+
 uint64
 sys_read(void)
 {
   ...
+  // Varios CPUs a la vez: suma atómica
     .
+  __atomic_fetch_add(&readcount, 1,
+                     __ATOMIC_RELAXED);
   argaddr(1, &p);
```

kernel/sysproc.c

```
+extern uint64 readcount;
+
+uint64
+sys_getreadcount(void)
+{
+  // Un solo ld de 64 bits.
+  return readcount;
+}
```

Agregar `getreadcount()`: el kernel

kernel/sysfile.c

```
+// Llamadas a read() desde el arranque.
+uint64 readcount = 0;
+
 uint64
 sys_read(void)
 {
   ...
+  // Varios CPUs a la vez: suma atómica
     .
+  __atomic_fetch_add(&readcount, 1,
+                     __ATOMIC_RELAXED);
   argaddr(1, &p);
```

kernel/sysproc.c

```
+extern uint64 readcount;
+
+uint64
+sys_getreadcount(void)
+{
+  // Un solo ld de 64 bits.
+  return readcount;
+}
```

¿Por qué atómica? `make qemu` arranca 3 CPUs: con `readcount++` se pueden perder sumas, como en `threads.c` (semana 1).

Agregar `getreadcount()`: el registro

el resto del parche

```
@@ kernel/syscall.h
+#define SYS_getreadcount 23
@@ kernel/syscall.c
+extern uint64 sys_getreadcount(void);
   [SYS_sync]    = sys_sync,
+  [SYS_getreadcount] = sys_getreadcount,
@@ user/usys.pl
+entry("getreadcount");
@@ user/user.h
+int getreadcount(void);
@@ Makefile
+   $U/_readcount\
```

Siete archivos en total:

1. `kernel/syscall.h`: el número.
2. `kernel/syscall.c`: la tabla.
3. `kernel/sysfile.c`: el contador.
4. `kernel/sysproc.c`: la función.
5. `user/usys.pl`: el stub.
6. `user/user.h`: el prototipo.
7. `Makefile`: el programa en `UPROGS`.

Más el programa de prueba `user/readcount.c`.

Probarlo

user/readcount.c

```c
int
main(void)
{
  char buf[64];
  int before = getreadcount();
  int fd = open("README", O_RDONLY);
  while (read(fd, buf, sizeof(buf)) > 0)
    ;
  close(fd);
  int after = getreadcount();
  printf("antes %d, despues %d\n",
         before, after);
  exit(0);
}
```

bash

```
$ readcount
antes 10, despues 50
$ wc README
48 336 2441 README
$ readcount
antes 76, despues 116
```

Probarlo

user/readcount.c

```c
int
main(void)
{
  char buf[64];
  int before = getreadcount();
  int fd = open("README", O_RDONLY);
  while (read(fd, buf, sizeof(buf)) > 0)
    ;
  close(fd);
  int after = getreadcount();
  printf("antes %d, despues %d\n",
         before, after);
  exit(0);
}
```

bash

```
$ readcount
antes 10, despues 50
$ wc README
48 336 2441 README
$ readcount
antes 76, despues 116
```

- $50 - 10 = 40$: 39 lecturas con datos (2441/64) y una que retorna 0.

Probarlo

user/readcount.c

```c
int
main(void)
{
  char buf[64];
  int before = getreadcount();
  int fd = open("README", O_RDONLY);
  while (read(fd, buf, sizeof(buf)) > 0)
    ;
  close(fd);
  int after = getreadcount();
  printf("antes %d, despues %d\n",
         before, after);
  exit(0);
}
```

bash

```
$ readcount
antes 10, despues 50
$ wc README
48 336 2441 README
$ readcount
antes 76, despues 116
```

- $50 - 10 = 40$: 39 lecturas con datos ($2441/64$) y una que retorna 0.
- El 10: la shell lee byte por byte `readcount` y el Enter.

Resumen

- La **lotería** reparte la CPU en proporción a los boletos, sin estado global; es justa en promedio, no en el corto plazo. Moneda, transferencia e inflación ajustan los boletos.
- **Stride** logra la misma proporción de forma determinista con `pass` y stride.
- **CFS** es stride con pesos: corre el de menor `vruntime`, guardado en un árbol rojo-negro. Desde Linux 6.6 elige con **EEVDF** (elegibilidad y deadline virtual).
- Con varios CPUs importan la **coherencia** (y su costo), la **afinidad** y el diseño de colas: **SQMS** balancea solo, **MQMS** escala pero necesita migrar (work stealing).
- En xv6, una llamada al sistema va de `ecall` a `uservec`, `usertrap()`, `syscall()` y la función `sys_*`; agregar una nueva toca siete archivos.

Para la próxima clase

- Leer OSTEP cap. 12–14: espacios de direcciones y API de memoria.
- **Laboratorio xv6**: `make qemu` (salir con `Ctrl-a x`), depurar `syscall()` con gdb y agregar `getreadcount()`. El parche de referencia está en `code/week-04`.
- **P1** (`wish`) se entrega en la semana 6.
- **P2**, planificador por lotería en xv6, empieza la semana 5: hoy vieron su base.

Preguntas para pensar

1. Con boletos 100 y 50, ¿qué fracción espera cada uno? ¿Por qué varía en corridas cortas?
2. ¿Por qué migrar un proceso entre CPUs cuesta, aunque no se copie nada?
3. ¿Qué archivos de xv6 tocarían para agregar `getpriority()`?