

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

Espacios de direcciones, traducción y segmentación

¿Cómo comparten la memoria muchos procesos?

Gerson Garrido

Semana 5

Hoy

1. El espacio de direcciones: la memoria tal como la ve un proceso
2. La API de memoria: `malloc`, `free` y los errores que todos cometemos
3. Traducción de direcciones con base y límite
4. Segmentación: un par base/límite por cada segmento
5. Laboratorio: el planificador de xv6 por dentro

> **Lectura**
> OSTEP cap. 12–14 (sesión A) y cap. 15–16 (sesión B).

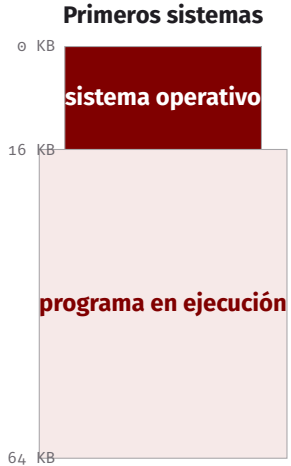
SECCIÓN 01

El espacio de direcciones

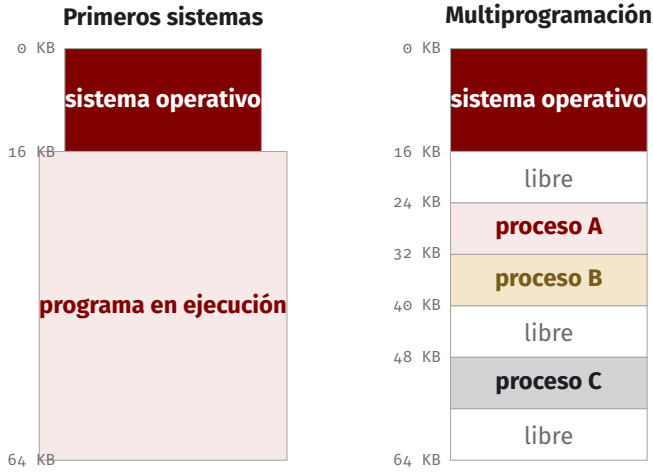
01



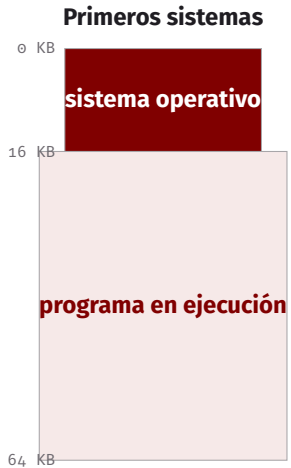
Al principio: un programa, toda la memoria

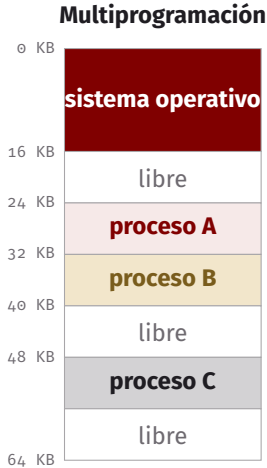


Al principio: un programa, toda la memoria



Al principio: un programa, toda la memoria





Varios procesos en memoria a la vez: mientras uno espera E/S, otro usa la CPU.

Problema: ¿cómo evitar que A lea o escriba la memoria de B?

La abstracción: el espacio de direcciones



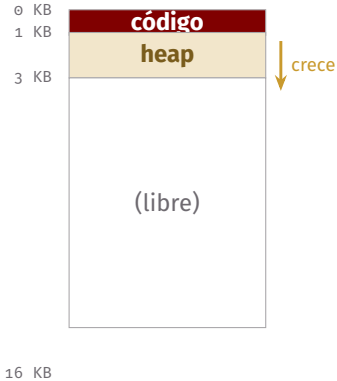
- **Código**: las instrucciones. Tamaño fijo.

La abstracción: el espacio de direcciones



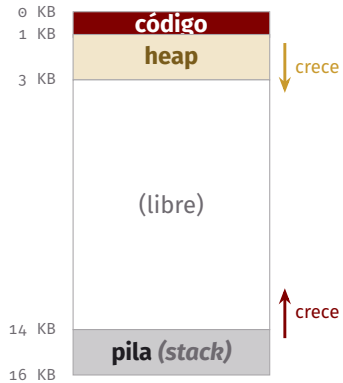
- **Código**: las instrucciones. Tamaño fijo.
- **Heap**: lo que se pide con `malloc`. Crece hacia direcciones altas.

La abstracción: el espacio de direcciones



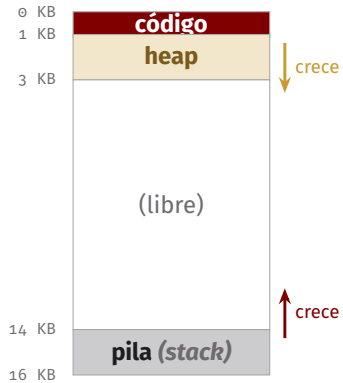
- **Código**: las instrucciones. Tamaño fijo.
- **Heap**: lo que se pide con `malloc`. Crece hacia direcciones altas.
- En medio, espacio libre para que heap y pila crezcan sin chocar.

La abstracción: el espacio de direcciones



- **Código**: las instrucciones. Tamaño fijo.
- **Heap**: lo que se pide con `malloc`. Crece hacia direcciones altas.
- En medio, espacio libre para que heap y pila crezcan sin chocar.
- **Pila**: variables locales y direcciones de retorno. Crece hacia abajo.

La abstracción: el espacio de direcciones



- **Código**: las instrucciones. Tamaño fijo.
- **Heap**: lo que se pide con `malloc`. Crece hacia direcciones altas.
- En medio, espacio libre para que heap y pila crezcan sin chocar.
- **Pila**: variables locales y direcciones de retorno. Crece hacia abajo.

Espacio de direcciones *(address space)*

La memoria tal como la ve un proceso: de 0 a un máximo, solo para él. Es una ilusión.

Tres metas de la memoria virtual

Transparencia

El programa no nota la virtualización: cree tener la memoria física solo para él.

Tres metas de la memoria virtual

Transparencia

El programa no nota la virtualización: cree tener la memoria física solo para él.

Eficiencia

Traducir no debe hacer lento cada acceso ni gastar mucha memoria en tablas: hace falta ayuda del hardware (la MMU y, más adelante, la TLB).

Tres metas de la memoria virtual

Transparencia

El programa no nota la virtualización: cree tener la memoria física solo para él.

Eficiencia

Traducir no debe hacer lento cada acceso ni gastar mucha memoria en tablas: hace falta ayuda del hardware (la MMU y, más adelante, la TLB).

Protección

Un proceso no puede tocar la memoria de otro ni la del kernel: aislamiento. Si un programa falla, falla solo él.

¿Dónde vive cada cosa? `layout.c`

code/week-05/layout.c

```c
int global_init = 42; // .data
int global_zero;      // .bss

int main(void) {
  int local = 7;
  int *small = malloc(16);
  int *big = malloc(1 << 20);
  void *brk_end = sbrk(0);
  printf("...%p\n", (void *)main);
  // ... una línea por región
}
```

bash

```
$ setarch -R ./layout
código  main()       0xaaaaaaaa0918
datos   global_init  0xaaaaaaac0010
bss     global_zero  0xaaaaaaac0018
heap    malloc(16)   0xaaaaaaac12a0
heap    sbrk(0)      0xaaaaaaae2000
mmap    malloc(1MiB) 0xfffff7cef010
pila    local        0xffffffffbcc
```

¿Dónde vive cada cosa? `layout.c`

code/week-05/layout.c

```c
int global_init = 42; // .data
int global_zero;      // .bss

int main(void) {
  int local = 7;
  int *small = malloc(16);
  int *big = malloc(1 << 20);
  void *brk_end = sbrk(0);
  printf("...%p\n", (void *)main);
  // ... una línea por región
}
```

bash

```
$ setarch -R ./layout
código  main()       0xaaaaaaaa0918
datos   global_init  0xaaaaaaac0010
bss     global_zero  0xaaaaaaac0018
heap    malloc(16)   0xaaaaaaac12a0
heap    sbrk(0)      0xaaaaaaae2000
mmap    malloc(1MiB) 0xfffff7cef010
pila    local        0xfffffffffbcc
```

El orden coincide con el dibujo: código abajo, pila arriba. El bloque de 1 MiB va a una región aparte, creada con `mmap`.

Toda dirección que ven es virtual

bash

```
$ ./layout | grep -E 'main|local'
código  main()       0xaaaabe4a0918
pila    local        0xffff929d1fc
$ ./layout | grep -E 'main|local'
código  main()       0xaaaac7280918
pila    local        0xffffd3370b6c
```

- Sin `setarch -R` cambian en cada ejecución: es la aleatorización del espacio de direcciones *(ASLR)*, una defensa contra ataques.

Toda dirección que ven es virtual

```
bash
$ ./layout | grep -E 'main|local'
código  main()       0xaaaabe4a0918
pila    local        0xffff929d1fc
$ ./layout | grep -E 'main|local'
código  main()       0xaaaac7280918
pila    local        0xffffd3370b6c
```

- Sin `setarch -R` cambian en cada ejecución: es la aleatorización del espacio de direcciones *(ASLR)*, una defensa contra ataques.
- `0xfffffffffbcc` está cerca de 256 TB: no es una posición real de la RAM.

Toda dirección que ven es virtual

```
bash
$ ./layout | grep -E 'main|local'
código  main()       0xaaaabe4a0918
pila    local        0xffff929d1fc
$ ./layout | grep -E 'main|local'
código  main()       0xaaaac7280918
pila    local        0xffffd3370b6c
```

- Sin `setarch -R` cambian en cada ejecución: es la aleatorización del espacio de direcciones *(ASLR)*, una defensa contra ataques.
- `0xfffffffffbcc` está cerca de 256 TB: no es una posición real de la RAM.
- El hardware y el sistema operativo traducen cada dirección virtual a una física.

Lo que viene

¿Cómo se traduce de forma rápida y segura? Primero, lo más simple: base y límite.

SECCIÓN 02

La API de memoria

02



Dos tipos de memoria en un programa en C

Pila: automática. El compilador la reserva al entrar a la función y la libera al salir.

pila

```c
void f(void) {
  int x = 3; // muere al salir
}
```

Dos tipos de memoria en un programa en C

Pila: automática. El compilador la reserva al entrar a la función y la libera al salir.

pila

```c
void f(void) {
  int x = 3; // muere al salir
}
```

Heap: manual. Ustedes la piden y ustedes la devuelven. Sobrevive a la función.

heap

```c
int *g(void) {
  int *x = malloc(sizeof(*x));
  return x;  // alguien hará free
}
```

Dos tipos de memoria en un programa en C

Pila: automática. El compilador la reserva al entrar a la función y la libera al salir.

pila

```c
void f(void) {
  int x = 3; // muere al salir
}
```

Heap: manual. Ustedes la piden y ustedes la devuelven. Sobrevive a la función.

heap

```c
int *g(void) {
  int *x = malloc(sizeof(*x));
  return x;  // alguien hará free
}
```

La regla

Todo `malloc` exitoso necesita *exactamente un* `free`, y después del `free` el puntero ya no se usa.

Las funciones de la biblioteca

Función	Qué hace
`malloc(n)`	Reserva `n` bytes sin inicializar. Retorna `NULL` si no puede.
`calloc(k, n)`	Reserva `k` elementos de `n` bytes, todos en cero.
`realloc(p, n)`	Cambia el tamaño del bloque; puede moverlo a otra dirección.
`free(p)`	Devuelve el bloque. No recibe el tamaño: la biblioteca lo recuerda.

Las funciones de la biblioteca

Función	Qué hace
`malloc(n)`	Reserva `n` bytes sin inicializar. Retorna `NULL` si no puede.
`calloc(k, n)`	Reserva `k` elementos de `n` bytes, todos en cero.
`realloc(p, n)`	Cambia el tamaño del bloque; puede moverlo a otra dirección.
`free(p)`	Devuelve el bloque. No recibe el tamaño: la biblioteca lo recuerda.

- Usen `malloc(n * sizeof(int))` o `malloc(sizeof(*p))`, nunca un número fijo.

Las funciones de la biblioteca

Función	Qué hace
`malloc(n)`	Reserva `n` bytes sin inicializar. Retorna `NULL` si no puede.
`calloc(k, n)`	Reserva `k` elementos de `n` bytes, todos en cero.
`realloc(p, n)`	Cambia el tamaño del bloque; puede moverlo a otra dirección.
`free(p)`	Devuelve el bloque. No recibe el tamaño: la biblioteca lo recuerda.

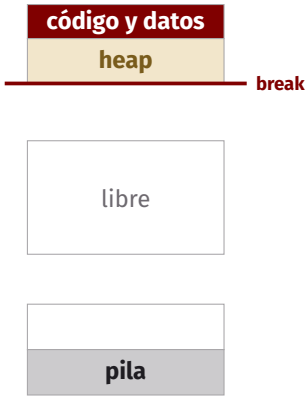
- Usen `malloc(n * sizeof(int))` o `malloc(sizeof(*p))`, nunca un número fijo.
- Para cadenas: `malloc(strlen(s) + 1)`, por el `'\0'` final.

Las funciones de la biblioteca

Función	Qué hace
`malloc(n)`	Reserva `n` bytes sin inicializar. Retorna `NULL` si no puede.
`calloc(k, n)`	Reserva `k` elementos de `n` bytes, todos en cero.
`realloc(p, n)`	Cambia el tamaño del bloque; puede moverlo a otra dirección.
`free(p)`	Devuelve el bloque. No recibe el tamaño: la biblioteca lo recuerda.

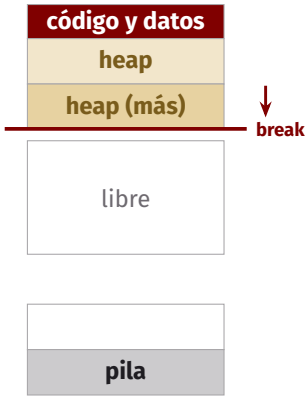
- Usen `malloc(n * sizeof(int))` o `malloc(sizeof(*p))`, nunca un número fijo.
- Para cadenas: `malloc(strlen(s) + 1)`, por el `'\0'` final.
- `malloc` y `free` no son llamadas al sistema: son funciones de la biblioteca de C que administran el heap.

Por debajo: `brk`/`sbrk` y `mmap`



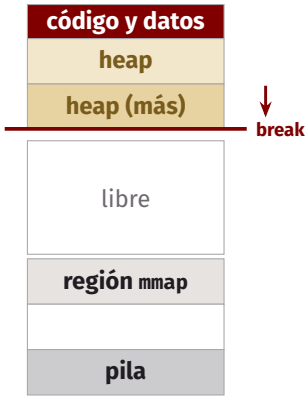
- El heap termina en el break del programa.

Por debajo: `brk`/`sbrk` y `mmap`



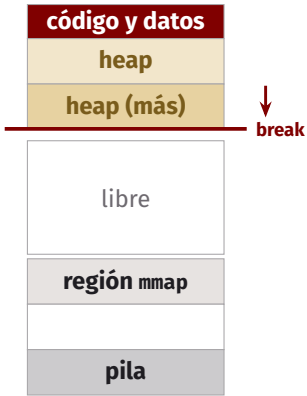
- El heap termina en el break del programa.
- Cuando `malloc` se queda sin espacio, llama a `brk`/`sbrk` para mover el break: el kernel agrega memoria al final del heap.

Por debajo: `brk`/`sbrk` y `mmap`



- El heap termina en el break del programa.
- Cuando `malloc` se queda sin espacio, llama a `brk`/`sbrk` para mover el break: el kernel agrega memoria al final del heap.
- Para bloques grandes (en glibc, desde 128 KB), `malloc` usa `mmap`: el kernel crea una región anónima nueva en otra parte.

Por debajo: `brk`/`sbrk` y `mmap`



- El heap termina en el break del programa.
- Cuando `malloc` se queda sin espacio, llama a `brk`/`sbrk` para mover el break: el kernel agrega memoria al final del heap.
- Para bloques grandes (en glibc, desde 128 KB), `malloc` usa `mmap`: el kernel crea una región anónima nueva en otra parte.
- No llamen a `brk` si usan `malloc`: los dos manejan el mismo break.

Errores típicos con la memoria

1. **Olvidar reservar**: `strcpy(d, s)` con `d` sin apuntar a nada.

Errores típicos con la memoria

1. **Olvidar reservar**: `strcpy(d, s)` con `d` sin apuntar a nada.
2. **Reservar de menos**: `malloc(strlen(s))` y luego `strcpy`: un byte fuera del bloque *(buffer overflow)*.

Errores típicos con la memoria

1. **Olvidar reservar**: `strcpy(d, s)` con `d` sin apuntar a nada.
2. **Reservar de menos**: `malloc(strlen(s))` y luego `strcpy`: un byte fuera del bloque *(buffer overflow)*.
3. **No inicializar**: leer basura de `malloc`.

Errores típicos con la memoria

1. **Olvidar reservar**: `strcpy(d, s)` con `d` sin apuntar a nada.
2. **Reservar de menos**: `malloc(strlen(s))` y luego `strcpy`: un byte fuera del bloque *(buffer overflow)*.
3. **No inicializar**: leer basura de `malloc`.
4. **Fuga de memoria** *(memory leak)*: nunca llamar a `free`.

Errores típicos con la memoria

1. **Olvidar reservar**: `strcpy(d, s)` con `d` sin apuntar a nada.
2. **Reservar de menos**: `malloc(strlen(s))` y luego `strcpy`: un byte fuera del bloque *(buffer overflow)*.
3. **No inicializar**: leer basura de `malloc`.
4. **Fuga de memoria** *(memory leak)*: nunca llamar a `free`.
5. **Puntero colgante** *(dangling pointer)*: usar la memoria después del `free`.

Errores típicos con la memoria

1. **Olvidar reservar**: `strcpy(d, s)` con `d` sin apuntar a nada.
2. **Reservar de menos**: `malloc(strlen(s))` y luego `strcpy`: un byte fuera del bloque *(buffer overflow)*.
3. **No inicializar**: leer basura de `malloc`.
4. **Fuga de memoria** *(memory leak)*: nunca llamar a `free`.
5. **Puntero colgante** *(dangling pointer)*: usar la memoria después del `free`.
6. **Doble `free`**: liberar dos veces el mismo bloque.

Errores típicos con la memoria

1. **Olvidar reservar**: `strcpy(d, s)` con `d` sin apuntar a nada.
2. **Reservar de menos**: `malloc(strlen(s))` y luego `strcpy`: un byte fuera del bloque *(buffer overflow)*.
3. **No inicializar**: leer basura de `malloc`.
4. **Fuga de memoria** *(memory leak)*: nunca llamar a `free`.
5. **Puntero colgante** *(dangling pointer)*: usar la memoria después del `free`.
6. **Doble `free`**: liberar dos veces el mismo bloque.
7. **`free` inválido**: un puntero que no vino de `malloc`.

Errores típicos con la memoria

1. **Olvidar reservar**: `strcpy(d, s)` con `d` sin apuntar a nada.
2. **Reservar de menos**: `malloc(strlen(s))` y luego `strcpy`: un byte fuera del bloque *(buffer overflow)*.
3. **No inicializar**: leer basura de `malloc`.
4. **Fuga de memoria** *(memory leak)*: nunca llamar a `free`.
5. **Puntero colgante** *(dangling pointer)*: usar la memoria después del `free`.
6. **Doble `free`**: liberar dos veces el mismo bloque.
7. **`free` inválido**: un puntero que no vino de `malloc`.

Compila no significa funciona

El compilador casi nunca los ve y muchos *parecen* funcionar: usen `valgrind` o `-fsanitize=address`.

Fuga de memoria: `leak.c`

code/week-05/leak.c

```c
static char *make_label(int i) {
  char *s = malloc(64); // línea 9
  // ... (verifica NULL, snprintf)
  return s;
}
int main(void) {
  for (int i = 0; i < 10; i++) {
    char *label =
        make_label(i);    // línea 18
    printf("%s\n", label);
  } // falta free(label)
}
```

bash

```
$ valgrind --leak-check=full ./leak
...
in use at exit: 640 bytes in 10 blocks
640 bytes in 10 blocks are definitely
    lost in loss record 1 of 1
  at malloc (vgpreload_memcheck...)
  by make_label (leak.c:9)
  by main (leak.c:18)
LEAK SUMMARY:
definitely lost: 640 bytes in 10 blocks
```

Fuga de memoria: `leak.c`

code/week-05/leak.c

```c
static char *make_label(int i) {
  char *s = malloc(64); // línea 9
  // ... (verifica NULL, snprintf)
  return s;
}
int main(void) {
  for (int i = 0; i < 10; i++) {
    char *label =
        make_label(i);    // línea 18
    printf("%s\n", label);
  } // falta free(label)
}
```

bash

```
$ valgrind --leak-check=full ./leak
...
in use at exit: 640 bytes in 10 blocks
640 bytes in 10 blocks are definitely
    lost in loss record 1 of 1
  at malloc (vgpreload_memcheck...)
  by make_label (leak.c:9)
  by main (leak.c:18)
LEAK SUMMARY:
definitely lost: 640 bytes in 10 blocks
```

valgrind dice dónde se reservaron los bloques (`leak.c:9`), no dónde falta el `free`.

Puntero colgante: `uaf.c`

code/week-05/uaf.c

```
1  static void close_account(
2      struct account *a) {
3    free(a);
4  }
5  int main(void) {
6    struct account *a =
7        malloc(sizeof(*a)); // l. 17
8    a->id = 1; a->balance = 100;
9    close_account(a);   // línea 22
10   printf("saldo: %d\n",
11          a->balance); // línea 23
12 }
```

bash

```
$ valgrind ./uaf
Invalid read of size 4
  at main (uaf.c:23)
Address 0x4a7e044 is 4 bytes inside
    a block of size 8 free'd
  at free (vgpreload_memcheck...)
  by close_account (uaf.c:13)
  by main (uaf.c:22)
Block was alloc'd at
  by main (uaf.c:17)
saldo: 100
```

Puntero colgante: `uaf.c`

code/week-05/uaf.c

```c
static void close_account(
    struct account *a) {
  free(a);
}
int main(void) {
  struct account *a =
      malloc(sizeof(*a)); // l. 17
  a->id = 1; a->balance = 100;
  close_account(a);   // línea 22
  printf("saldo: %d\n",
         a->balance); // línea 23
}
```

bash

```
$ valgrind ./uaf
Invalid read of size 4
  at main (uaf.c:23)
Address 0x4a7e044 is 4 bytes inside
    a block of size 8 free'd
  at free (vgpreload_memcheck...)
  by close_account (uaf.c:13)
  by main (uaf.c:22)
Block was alloc'd at
  by main (uaf.c:17)
saldo: 100
```

Imprimió 100: parece funcionar, hasta que otro `malloc` reutilice el bloque.

Memoria sin inicializar: `uninit.c`

code/week-05/uninit.c

```c
int *v = malloc(n * sizeof(int));
for (int i = 0; i < n - 1; i++)
  v[i] = i;     // falta v[n-1]
int sum = 0;
for (int i = 0; i < n; i++)
  sum += v[i];
if (sum > 28)   // línea 18
  printf("suma grande: %d\n", sum);
else
  printf("suma: %d\n", sum);
```

bash

```
$ valgrind --track-origins=yes ./uninit
Conditional jump or move depends
    on uninitialised value(s)
  at main (uninit.c:18)
Uninitialised value was created
    by a heap allocation
  at malloc (vgpreload_memcheck...)
  by main (uninit.c:9)
...
suma: 21
```

Memoria sin inicializar: `uninit.c`

code/week-05/uninit.c

```
1  int *v = malloc(n * sizeof(int));
2  for (int i = 0; i < n - 1; i++)
3    v[i] = i;      // falta v[n-1]
4  int sum = 0;
5  for (int i = 0; i < n; i++)
6    sum += v[i];
7  if (sum > 28)    // línea 18
8    printf("suma grande: %d\n", sum);
9  else
10   printf("suma: %d\n", sum);
```

bash

```
$ valgrind --track-origins=yes ./uninit
Conditional jump or move depends
    on uninitialised value(s)
  at main (uninit.c:18)
Uninitialised value was created
    by a heap allocation
  at malloc (vgpreload_memcheck...)
  by main (uninit.c:9)
...
suma: 21
```

valgrind avisa cuando una decisión depende de la basura (línea 18), no al copiarla.

Doble `free`: `doublefree.c`

code/week-05/doublefree.c

```c
static void cleanup(char *buf) {
  free(buf);          // línea 9
}
int main(void) {
  char *buf = malloc(32); // l. 13
  strcpy(buf, "hola");
  printf("%s\n", buf);
  cleanup(buf);       // línea 18
  free(buf);          // línea 19
}
```

bash

```
$ valgrind ./doublefree
Invalid free() / delete / delete[]
  at free (vgpreload_memcheck...)
  by main (doublefree.c:19)
Address 0x4a7e040 is 0 bytes inside
    a block of size 32 free'd
  by cleanup (doublefree.c:9)
  by main (doublefree.c:18)
```

Doble `free`: `doublefree.c`

code/week-05/doublefree.c

```c
static void cleanup(char *buf) {
  free(buf);          // línea 9
}
int main(void) {
  char *buf = malloc(32); // l. 13
  strcpy(buf, "hola");
  printf("%s\n", buf);
  cleanup(buf);       // línea 18
  free(buf);          // línea 19
}
```

bash

```
$ valgrind ./doublefree
Invalid free() / delete / delete[]
  at free (vgpreload_memcheck...)
  by main (doublefree.c:19)
Address 0x4a7e040 is 0 bytes inside
    a block of size 32 free'd
  by cleanup (doublefree.c:9)
  by main (doublefree.c:18)
```

Sin valgrind, glibc a veces lo detecta: `free(): double free detected in tcache 2` y el programa aborta. Otras veces corrompe el heap en silencio.

Cómo leer a valgrind

Mensaje	Error	Ejemplo
`definitely lost`	fuga de memoria	`leak.c`
`Invalid read/write ... free'd`	puntero colgante (use-after-free)	`uaf.c`
`Invalid write ... after a block`	escribir fuera del bloque	`malloc(strlen(s))`
`depends on uninitialised value(s)`	memoria sin inicializar	`uninit.c`
`Invalid free()`	doble `free` o `free` inválido	`doublefree.c`

El comando de la semana

```
gcc -Wall -Wextra -g prog.c -o prog
valgrind --leak-check=full --track-origins=yes ./prog
```

Con `-g`, valgrind muestra archivo y línea. En `code/week-05`: `make check`.

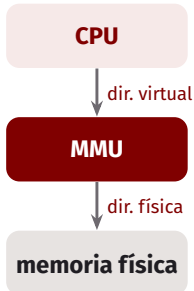
SECCIÓN 03

Base y límite

03

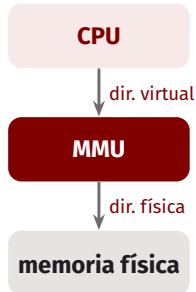
La idea: interponerse en cada acceso

- Cada búsqueda de instrucción, cada lectura y cada escritura usa una dirección virtual.



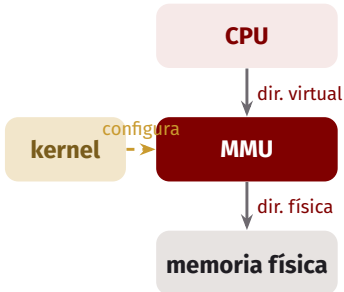
La idea: interponerse en cada acceso

- Cada búsqueda de instrucción, cada lectura y cada escritura usa una dirección virtual.
- El hardware la traduce a una dirección física *en cada acceso*: traducción de direcciones *(address translation)*.



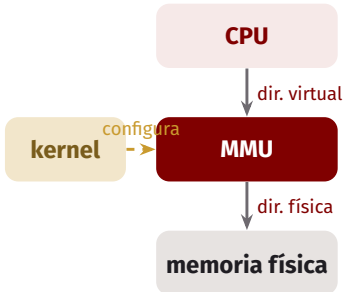
La idea: interponerse en cada acceso

- Cada búsqueda de instrucción, cada lectura y cada escritura usa una dirección virtual.
- El hardware la traduce a una dirección física *en cada acceso*: traducción de direcciones *(address translation)*.
- El sistema operativo configura el hardware y lleva la cuenta de qué memoria está libre.



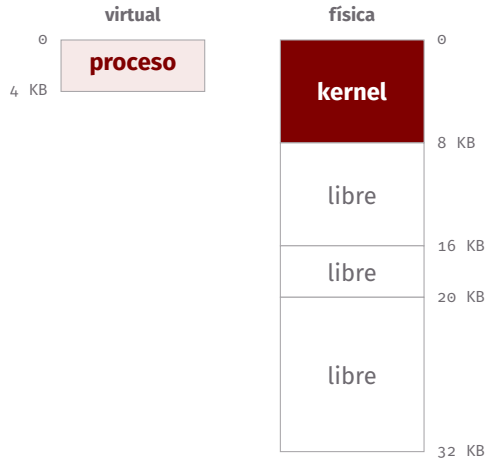
La idea: interponerse en cada acceso

- Cada búsqueda de instrucción, cada lectura y cada escritura usa una dirección virtual.
- El hardware la traduce a una dirección física *en cada acceso*: traducción de direcciones *(address translation)*.
- El sistema operativo configura el hardware y lleva la cuenta de qué memoria está libre.
- Mismo patrón que la ejecución directa limitada: el programa corre directo, el hardware lo limita y el kernel interviene solo en momentos clave.



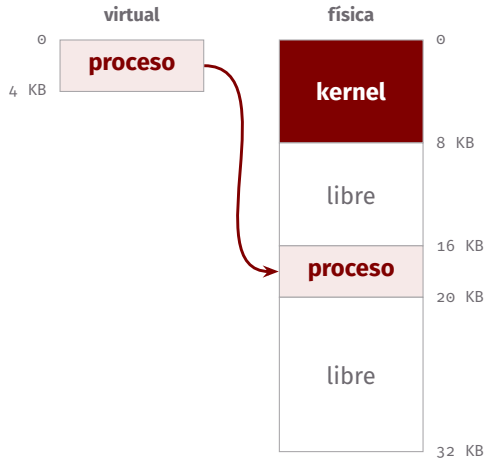
Relocalización: el proceso no vive en la dirección 0

- El proceso cree que su espacio de direcciones empieza en 0 y mide 4 KB.



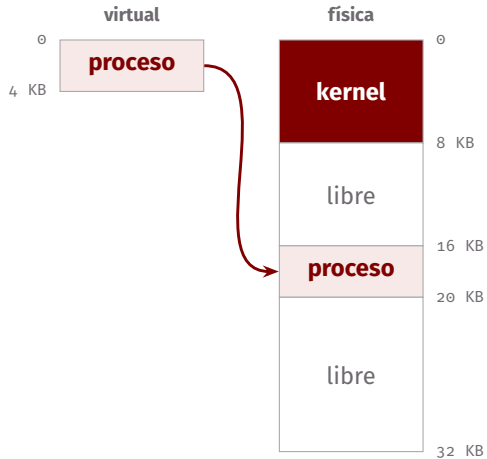
Relocalización: el proceso no vive en la dirección 0

- El proceso cree que su espacio de direcciones empieza en 0 y mide 4 KB.
- En la memoria física, el kernel lo colocó en 16 KB.



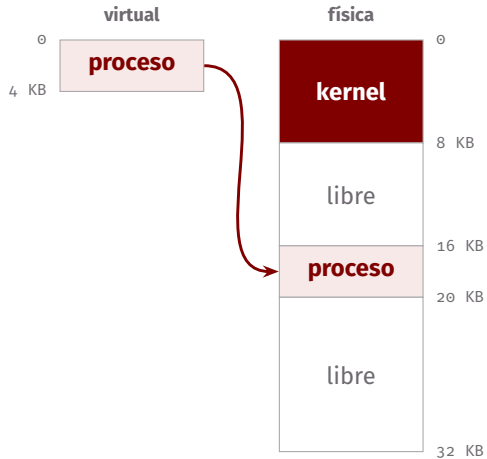
Relocalización: el proceso no vive en la dirección 0

- El proceso cree que su espacio de direcciones empieza en 0 y mide 4 KB.
- En la memoria física, el kernel lo colocó en 16 KB.
- Relocalización *estática* (reescribir las direcciones al cargar): no protege y no permite mover el proceso después.



Relocalización: el proceso no vive en la dirección 0

- El proceso cree que su espacio de direcciones empieza en 0 y mide 4 KB.
- En la memoria física, el kernel lo colocó en 16 KB.
- Relocalización *estática* (reescribir las direcciones al cargar): no protege y no permite mover el proceso después.
- Relocalización *dinámica* *(dynamic relocation)*: el hardware suma la base en cada acceso.



Base y límite: la MMU más simple

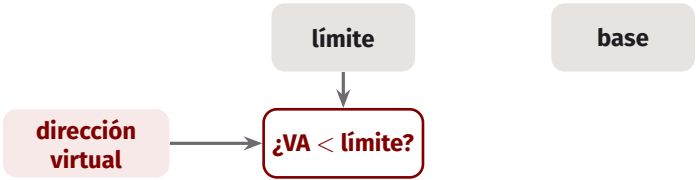
límite

base

dirección virtual

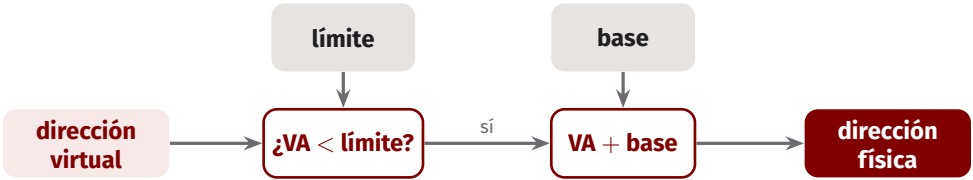
Dos registros por CPU, dentro de la MMU. Solo el kernel puede escribirlos.

Base y límite: la MMU más simple



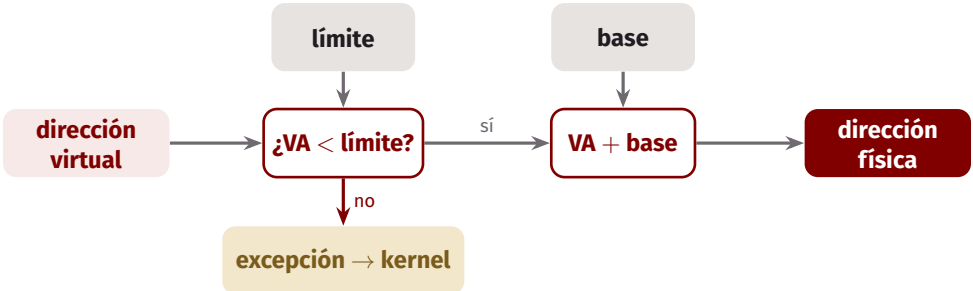
Dos registros por CPU, dentro de la MMU. Solo el kernel puede escribirlos.

Base y límite: la MMU más simple



Dos registros por CPU, dentro de la MMU. Solo el kernel puede escribirlos.

Base y límite: la MMU más simple



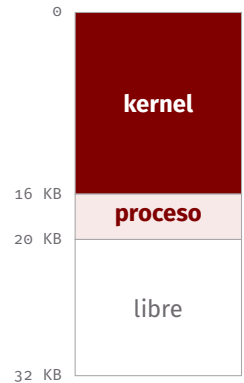
física = virtual + base, si virtual < límite

Aquí el límite es el *tamaño* del espacio de direcciones: se compara con la dirección virtual, antes de sumar la base.

Ejemplo resuelto 1: base = 16 KB, límite = 4 KB

Registros: `base = 0x4000` (16 384) y `límite = 0x1000` (4096).

Dirección virtual 1 KB = `0x0400` = 1024

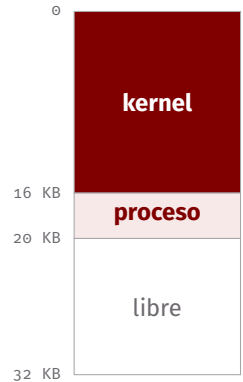


Ejemplo resuelto 1: base $=$ 16 KB, límite $=$ 4 KB

Registros: `base = 0x4000` (16 384) y `límite = 0x1000` (4096).

Dirección virtual 1 KB $=$ `0x0400` $=$ 1024

1. ¿`0x0400` $<$ `0x1000`? Sí: 1024 $<$ 4096.



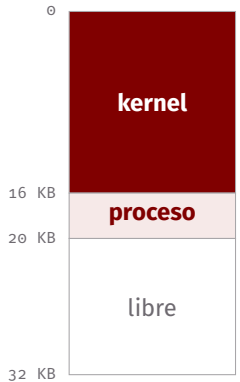
Ejemplo resuelto 1: base $=$ 16 KB, límite $=$ 4 KB

Registros: `base = 0x4000` (16 384) y `límite = 0x1000` (4096).

Dirección virtual 1 KB $=$ `0x0400` $=$ 1024

1. ¿`0x0400` $<$ `0x1000`? Sí: 1024 $<$ 4096.
2. Sumar la base, en binario:

```
  0000 0100 0000 0000    0x0400
+ 0100 0000 0000 0000    0x4000
```



Ejemplo resuelto 1: base = 16 KB, límite = 4 KB

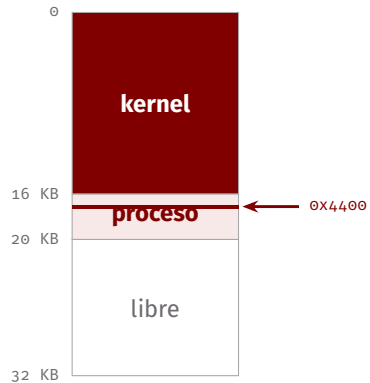
Registros: `base = 0x4000` (16 384) y `límite = 0x1000` (4096).

Dirección virtual 1 KB = `0x0400` = 1024

1. ¿`0x0400` < `0x1000`? Sí: 1024 < 4096.
2. Sumar la base, en binario:

```
  0000 0100 0000 0000    0x0400
+ 0100 0000 0000 0000    0x4000
-------------------------------
  0100 0100 0000 0000    0x4400
```

Dirección física `0x4400` = 17 408 = 17 KB.



Ejemplo resuelto 1: base = 16 KB, límite = 4 KB

Registros: `base = 0x4000` (16 384) y `límite = 0x1000` (4096).

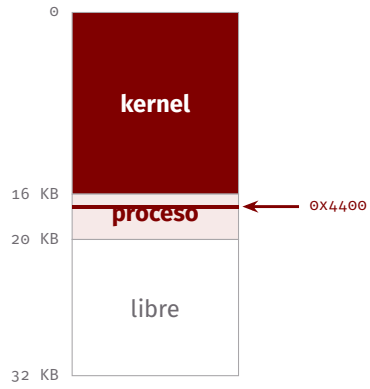
Dirección virtual 1 KB = `0x0400` = 1024

1. ¿`0x0400` < `0x1000`? Sí: 1024 < 4096.
2. Sumar la base, en binario:

```
  0000 0100 0000 0000    0x0400
+ 0100 0000 0000 0000    0x4000
-------------------------------
  0100 0100 0000 0000    0x4400
```

Dirección física `0x4400` = 17 408 = 17 KB.

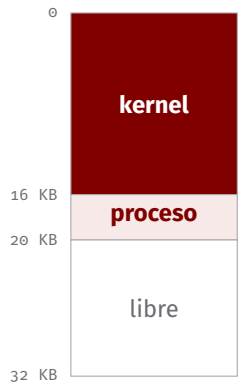
No hay acarreos: la base solo “enciende” el bit 14.



Ejemplo resuelto 2: fuera del límite

Mismos registros: `base = 0x4000`, `límite = 0x1000`.

Dirección virtual 5 KB $=$ `0x1400` $=$ 5120

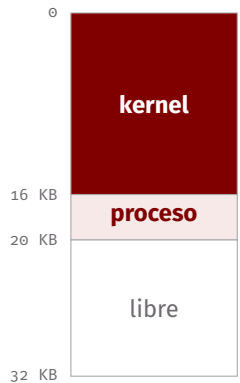


Ejemplo resuelto 2: fuera del límite

Mismos registros: `base = 0x4000`, `límite = 0x1000`.

Dirección virtual 5 KB $=$ `0x1400` $=$ 5120

1. ¿`0x1400` $<$ `0x1000`? No: 5120 $\geq$ 4096.

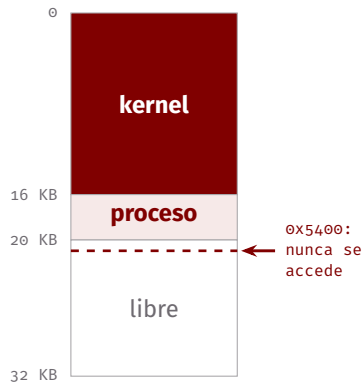


Ejemplo resuelto 2: fuera del límite

Mismos registros: `base = 0x4000`, `límite = 0x1000`.

Dirección virtual 5 KB $=$ `0x1400` $=$ 5120

1. ¿`0x1400` $<$ `0x1000`? No: 5120 $\geq$ 4096.
2. La MMU no suma: lanza una excepción y salta al manejador del kernel.

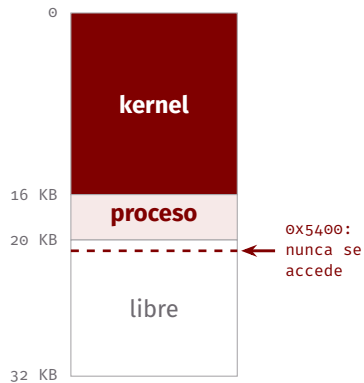


Ejemplo resuelto 2: fuera del límite

Mismos registros: `base = 0x4000`, `límite = 0x1000`.

Dirección virtual 5 KB = `0x1400` = 5120

1. ¿`0x1400` $<$ `0x1000`? No: 5120 $\geq$ 4096.
2. La MMU no suma: lanza una excepción y salta al manejador del kernel.
3. El kernel termina el proceso (en Linux, señal `SIGSEGV`).



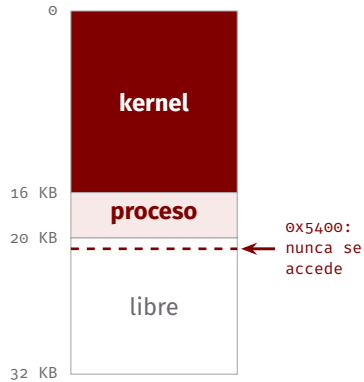
Ejemplo resuelto 2: fuera del límite

Mismos registros: `base = 0x4000`, `límite = 0x1000`.

Dirección virtual 5 KB $=$ `0x1400` $=$ 5120

1. ¿`0x1400` $<$ `0x1000`? No: 5120 $\geq$ 4096.
2. La MMU no suma: lanza una excepción y salta al manejador del kernel.
3. El kernel termina el proceso (en Linux, señal `SIGSEGV`).

¿Y la dirección `0x1000` $=$ 4096?



Ejemplo resuelto 2: fuera del límite

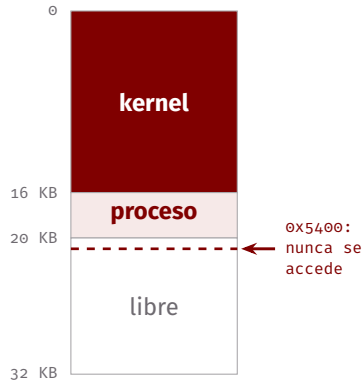
Mismos registros: `base = 0x4000`, `límite = 0x1000`.

Dirección virtual 5 KB $=$ `0x1400` $=$ 5120

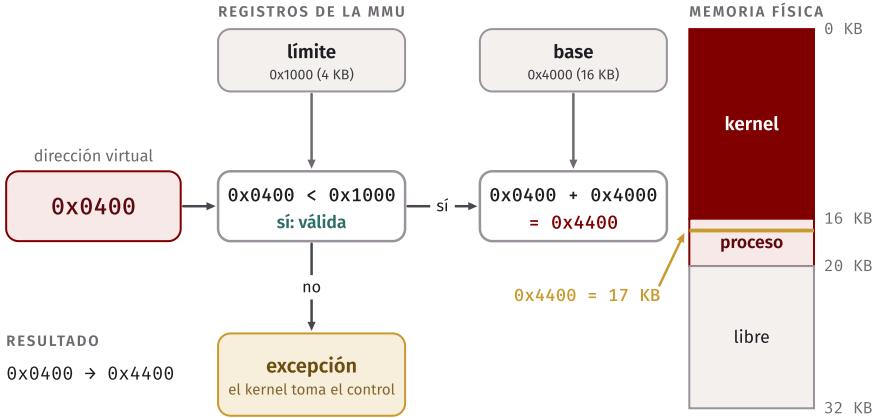
1. ¿`0x1400` $<$ `0x1000`? No: 5120 $\geq$ 4096.
2. La MMU no suma: lanza una excepción y salta al manejador del kernel.
3. El kernel termina el proceso (en Linux, señal `SIGSEGV`).

¿Y la dirección `0x1000` $=$ 4096?

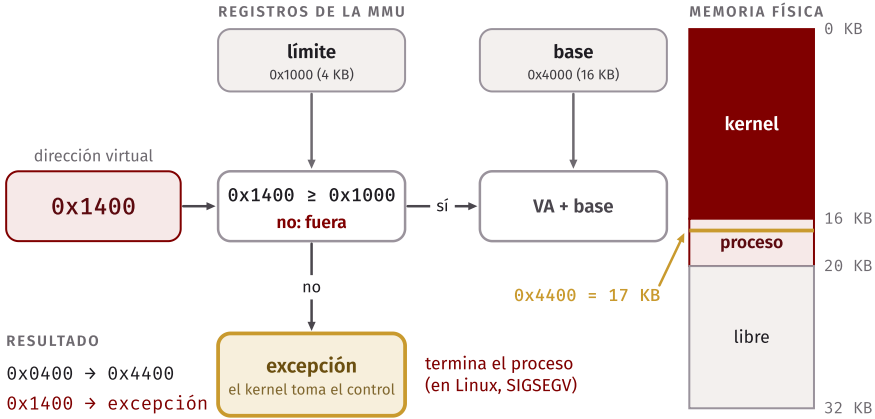
4. ¿`0x1000` $<$ `0x1000`? No: el último byte válido es `0x0FFF`. Una sola unidad de más ya es un error.



Los dos ejemplos, dentro de la MMU



Los dos ejemplos, dentro de la MMU



¿Qué necesita el hardware?

Requisito	Para qué
Modo kernel y modo usuario	Que solo el kernel cambie la configuración.

¿Qué necesita el hardware?

Requisito	Para qué
Modo kernel y modo usuario	Que solo el kernel cambie la configuración.
Registros base y límite	Un par por CPU, dentro de la MMU.

¿Qué necesita el hardware?

Requisito	Para qué
Modo kernel y modo usuario	Que solo el kernel cambie la configuración.
Registros base y límite	Un par por CPU, dentro de la MMU.
Sumador y comparador	Traducir y verificar cada acceso sin ayuda del software.

¿Qué necesita el hardware?

Requisito	Para qué
Modo kernel y modo usuario	Que solo el kernel cambie la configuración.
Registros base y límite	Un par por CPU, dentro de la MMU.
Sumador y comparador	Traducir y verificar cada acceso sin ayuda del software.
Instrucciones privilegiadas	Escribir base y límite (desde modo usuario: excepción).

¿Qué necesita el hardware?

Requisito	Para qué
Modo kernel y modo usuario	Que solo el kernel cambie la configuración.
Registros base y límite	Un par por CPU, dentro de la MMU.
Sumador y comparador	Traducir y verificar cada acceso sin ayuda del software.
Instrucciones privilegiadas	Escribir base y límite (desde modo usuario: excepción).
Excepciones	Saltar al kernel si un acceso está fuera de límite.

¿Qué necesita el hardware?

Requisito	Para qué
Modo kernel y modo usuario	Que solo el kernel cambie la configuración.
Registros base y límite	Un par por CPU, dentro de la MMU.
Sumador y comparador	Traducir y verificar cada acceso sin ayuda del software.
Instrucciones privilegiadas	Escribir base y límite (desde modo usuario: excepción).
Excepciones	Saltar al kernel si un acceso está fuera de límite.

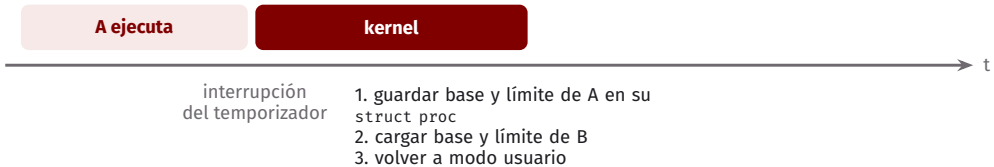
Es rápido

Una comparación y una suma se hacen en hardware, en paralelo con el resto de la instrucción.

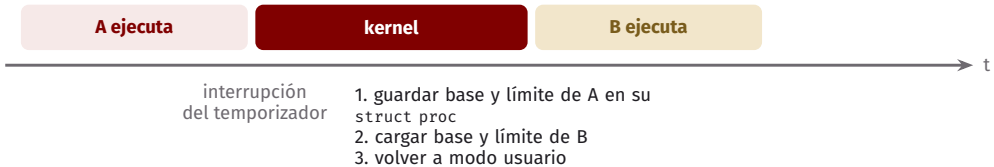
El papel del sistema operativo



El papel del sistema operativo

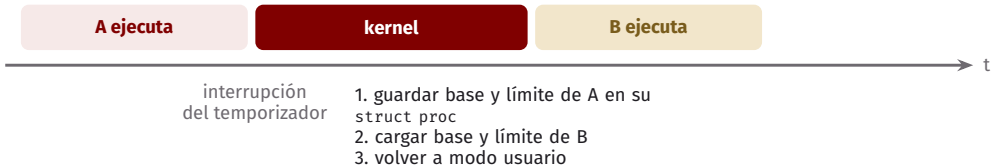


El papel del sistema operativo



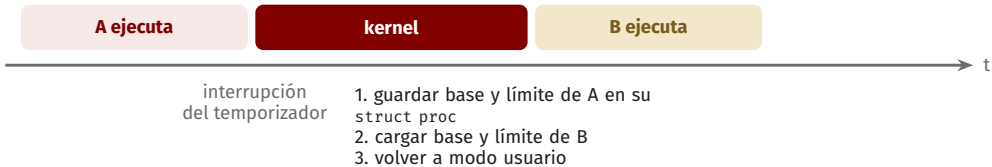
- **En cada cambio de contexto**: guardar y restaurar base y límite.

El papel del sistema operativo



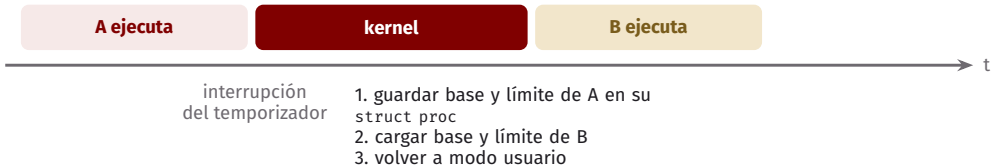
- **Al crear un proceso**: buscar un hueco en la lista libre *(free list)* y marcarlo usado.
- **En cada cambio de contexto**: guardar y restaurar base y límite.

El papel del sistema operativo



- **Al crear un proceso**: buscar un hueco en la lista libre *(free list)* y marcarlo usado.
- **Al terminar**: devolver su memoria a la lista libre.
- **En cada cambio de contexto**: guardar y restaurar base y límite.

El papel del sistema operativo



- **Al crear un proceso**: buscar un hueco en la lista libre *(free list)* y marcarlo usado.
- **Al terminar**: devolver su memoria a la lista libre.
- **En cada cambio de contexto**: guardar y restaurar base y límite.
- **Excepciones**: instalar los manejadores al arrancar; terminar al proceso que se sale de su límite.

El problema de base y límite



- El espacio de direcciones entero se coloca contiguo en la memoria física.

El problema de base y límite



- El espacio de direcciones entero se coloca contiguo en la memoria física.
- El hueco entre heap y pila ocupa memoria física aunque nadie lo use: fragmentación interna *(internal fragmentation)*.

El problema de base y límite



- El espacio de direcciones entero se coloca contiguo en la memoria física.
- El hueco entre heap y pila ocupa memoria física aunque nadie lo use: fragmentación interna *(internal fragmentation)*.
- Con espacios de direcciones enormes (los 256 TB de `layout.c`) es imposible.

El problema de base y límite



- El espacio de direcciones entero se coloca contiguo en la memoria física.
- El hueco entre heap y pila ocupa memoria física aunque nadie lo use: fragmentación interna *(internal fragmentation)*.
- Con espacios de direcciones enormes (los 256 TB de `layout.c`) es imposible.

La siguiente idea

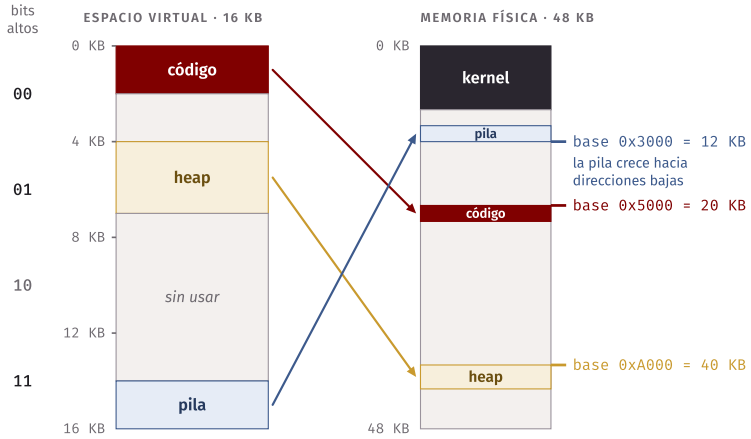
¿Y si cada *parte* del espacio de direcciones tuviera su propio par base/límite?

SECCIÓN 04

Segmentación

04

Un par base/límite por segmento



La tabla de segmentos

Segmento	Bits	Base	Tamaño	Crece hacia
Código	`00`	`0x5000` (20 KB)	`0x800` (2 KB)	+
Heap	`01`	`0xA000` (40 KB)	`0xC00` (3 KB)	+
Pila	`11`	`0x3000` (12 KB)	`0x800` (2 KB)	−

La tabla de segmentos

Segmento	Bits	Base	Tamaño	Crece hacia
Código	`00`	`0x5000` (20 KB)	`0x800` (2 KB)	+
Heap	`01`	`0xA000` (40 KB)	`0xC00` (3 KB)	+
Pila	`11`	`0x3000` (12 KB)	`0x800` (2 KB)	−

- Espacio virtual de 16 KB: direcciones de 14 bits.

La tabla de segmentos

Segmento	Bits	Base	Tamaño	Crece hacia
Código	`00`	`0x5000` (20 KB)	`0x800` (2 KB)	+
Heap	`01`	`0xA000` (40 KB)	`0xC00` (3 KB)	+
Pila	`11`	`0x3000` (12 KB)	`0x800` (2 KB)	−

- Espacio virtual de 16 KB: direcciones de 14 bits.
- Los 2 bits altos eligen el segmento; los 12 bajos son el offset (hasta 4 KB).

La tabla de segmentos

Segmento	Bits	Base	Tamaño	Crece hacia
Código	`00`	`0x5000` (20 KB)	`0x800` (2 KB)	+
Heap	`01`	`0xA000` (40 KB)	`0xC00` (3 KB)	+
Pila	`11`	`0x3000` (12 KB)	`0x800` (2 KB)	−

- Espacio virtual de 16 KB: direcciones de 14 bits.
- Los 2 bits altos eligen el segmento; los 12 bajos son el offset (hasta 4 KB).
- La pila crece hacia abajo: su base es el *tope* de su región (ocupa de 10 KB a 12 KB).

La tabla de segmentos

Segmento	Bits	Base	Tamaño	Crece hacia
Código	`00`	`0x5000` (20 KB)	`0x800` (2 KB)	+
Heap	`01`	`0xA000` (40 KB)	`0xC00` (3 KB)	+
Pila	`11`	`0x3000` (12 KB)	`0x800` (2 KB)	−

- Espacio virtual de 16 KB: direcciones de 14 bits.
- Los 2 bits altos eligen el segmento; los 12 bajos son el offset (hasta 4 KB).
- La pila crece hacia abajo: su base es el *tope* de su región (ocupa de 10 KB a 12 KB).
- El kernel guarda la tabla en el `struct proc` y la carga en cada cambio de contexto.

Ejemplo resuelto 3: una dirección del heap

Dirección virtual `0x1068` $= 4200$.

Ejemplo resuelto 3: una dirección del heap

Dirección virtual `0x1068` = 4200.



1. `0x1068` = `01` `0000 0110 1000`.

Ejemplo resuelto 3: una dirección del heap

Dirección virtual `0x1068` = 4200.



1. `0x1068` = `01` `0000` `0110` `1000`.
2. Segmento `01` → heap: base `0xA000`, tamaño `0xC00`.

Ejemplo resuelto 3: una dirección del heap

Dirección virtual `0x1068` = 4200.



1. `0x1068` = `01` `0000` `0110` `1000`.
2. Segmento `01` → heap: base `0xA000`, tamaño `0xC00`.
3. Offset = `0x1068 & 0xFFF` = `0x068` = 104.

Ejemplo resuelto 3: una dirección del heap

Dirección virtual `0x1068` = 4200.



1. `0x1068` = `01` `0000 0110 1000`.
2. Segmento `01` → heap: base `0xA000`, tamaño `0xC00`.
3. Offset = `0x1068 & 0xFFF` = `0x068` = 104.
4. ¿`0x068` < `0xC00`? Sí (104 < 3072).

Ejemplo resuelto 3: una dirección del heap

Dirección virtual `0x1068` = 4200.



1. `0x1068` = `01` `0000 0110 1000`.
2. Segmento `01` → heap: base `0xA000`, tamaño `0xC00`.
3. Offset = `0x1068 & 0xFFF` = `0x068` = 104.
4. ¿`0x068` < `0xC00`? Sí (104 < 3072).
5. Física = `0xA000` + `0x068` = `0xA068` = 41 064.

Ejemplo resuelto 3: una dirección del heap

Dirección virtual `0x1068` = 4200.



1. `0x1068` = `01` `0000` `0110` `1000`.
2. Segmento `01` → heap: base `0xA000`, tamaño `0xC00`.
3. Offset = `0x1068 & 0xFFF` = `0x068` = 104.
4. ¿`0x068` < `0xC00`? Sí (104 < 3072).
5. Física = `0xA000` + `0x068` = `0xA068` = 41 064.

¿Y `0x1E00`?

Segmento `01`, offset `0xE00` = 3584 ≥ 3072: fuera del heap. El hardware lanza una excepción: de ahí el nombre `segmentation fault`.

Ejemplo resuelto 4: la pila crece hacia abajo

Dirección virtual `0x3C00` = 15 360.



Ejemplo resuelto 4: la pila crece hacia abajo

Dirección virtual `0x3C00` = 15 360.

1	1	1	1	0	0	0	0	0	0	0	0	0	0

1. Segmento `11` → pila: base `0x3000`, tamaño `0x800`, crece hacia −.



Ejemplo resuelto 4: la pila crece hacia abajo

Dirección virtual `0x3C00` = 15 360.

1	1	1	1	0	0	0	0	0	0	0	0	0	0

1. Segmento `11` → pila: base `0x3000`, tamaño `0x800`, crece hacia −.
2. Offset = `0xC00` = 3072.



Ejemplo resuelto 4: la pila crece hacia abajo

Dirección virtual `0x3C00` = 15 360.

1	1	1	1	0	0	0	0	0	0	0	0	0	0

1. Segmento `11` → pila: base `0x3000`, tamaño `0x800`, crece hacia −.
2. Offset = `0xC00` = 3072.
3. Offset negativo = offset − tamaño máximo (4 KB): `0xC00` − `0x1000` = −`0x400`.



Ejemplo resuelto 4: la pila crece hacia abajo

Dirección virtual `0x3C00` = 15 360.

1	1	1	1	0	0	0	0	0	0	0	0	0	0

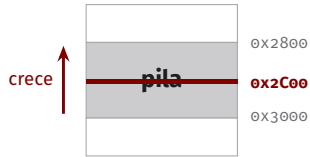
1. Segmento `11` → pila: base `0x3000`, tamaño `0x800`, crece hacia −.
2. Offset = `0xC00` = 3072.
3. Offset negativo = offset − tamaño máximo (4 KB): `0xC00` − `0x1000` = −`0x400`.
4. ¿$|-1024| \leq 2048$? Sí: válido.



Ejemplo resuelto 4: la pila crece hacia abajo

Dirección virtual `0x3C00` = 15 360.

1	1	1	1	0	0	0	0	0	0	0	0	0	0



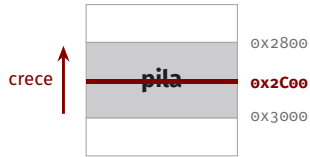
1. Segmento `11` → pila: base `0x3000`, tamaño `0x800`, crece hacia −.
2. Offset = `0xC00` = 3072.
3. Offset negativo = offset − tamaño máximo (4 KB): `0xC00` − `0x1000` = −`0x400`.
4. ¿$|-1024| \leq 2048$? Sí: válido.
5. Física = `0x3000` − `0x400` = `0x2C00` = 11 264.

Ejemplo resuelto 4: la pila crece hacia abajo

Dirección virtual `0x3C00` = 15 360.

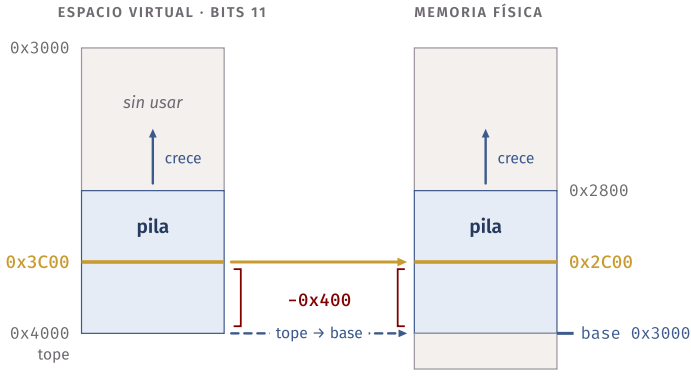
1	1	1	1	0	0	0	0	0	0	0	0	0	0

1. Segmento `11` → pila: base `0x3000`, tamaño `0x800`, crece hacia −.
2. Offset = `0xC00` = 3072.
3. Offset negativo = offset − tamaño máximo (4 KB): `0xC00` − `0x1000` = −`0x400`.
4. ¿$|-1024| \leq 2048$? Sí: válido.
5. Física = `0x3000` − `0x400` = `0x2C00` = 11 264.



El hardware necesita un bit más por segmento: ¿crece hacia arriba o hacia abajo?

La pila en dibujo: medir desde el tope



`offset = 0xC00; la pila crece hacia abajo: 0xC00 - 0x1000 = -0x400`

`física = base + offset = 0x3000 - 0x400 = 0x2C00`

Lo que hace el hardware en cada acceso

pseudocódigo de la MMU

```
seg    = (va >> 12) & 0x3;  // 2 bits altos
offset = va & 0xFFF;        // 12 bits bajos
if (grows_neg[seg]) {
  offset = offset - 0x1000; // negativo
  bad = -offset > size[seg];
} else
  bad = offset >= size[seg];
if (bad || !allowed(prot[seg], op))
  raise_exception(SEGFAULT);
pa = base[seg] + offset;
```

- Desplazar y enmascarar: operaciones baratas para el hardware.

Lo que hace el hardware en cada acceso

pseudocódigo de la MMU

```
seg    = (va >> 12) & 0x3;  // 2 bits altos
offset = va & 0xFFF;        // 12 bits bajos
if (grows_neg[seg]) {
  offset = offset - 0x1000; // negativo
  bad = -offset > size[seg];
} else
  bad = offset >= size[seg];
if (bad || !allowed(prot[seg], op))
  raise_exception(SEGFAULT);
pa = base[seg] + offset;
```

- Desplazar y enmascarar: operaciones baratas para el hardware.
- Igual que antes, una comparación y una suma; ahora con una tabla de 3 o 4 entradas.

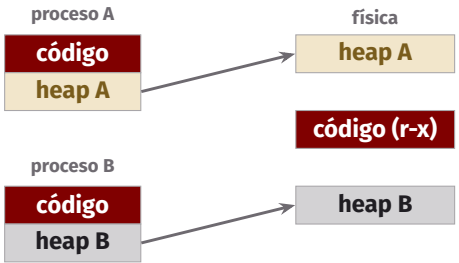
Lo que hace el hardware en cada acceso

pseudocódigo de la MMU

```
seg    = (va >> 12) & 0x3;  // 2 bits altos
offset = va & 0xFFF;        // 12 bits bajos
if (grows_neg[seg]) {
  offset = offset - 0x1000; // negativo
  bad = -offset > size[seg];
} else
  bad = offset >= size[seg];
if (bad || !allowed(prot[seg], op))
  raise_exception(SEGFAULT);
pa = base[seg] + offset;
```

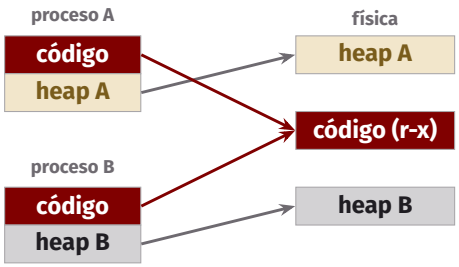
- Desplazar y enmascarar: operaciones baratas para el hardware.
- Igual que antes, una comparación y una suma; ahora con una tabla de 3 o 4 entradas.
- Con 2 bits y 3 segmentos, un cuarto del espacio (`10`) queda sin usar.

Compartir segmentos y bits de protección



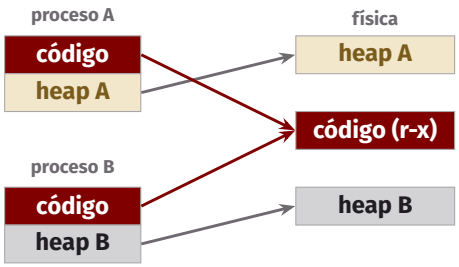
- Dos procesos del mismo programa tienen el mismo código.

Compartir segmentos y bits de protección



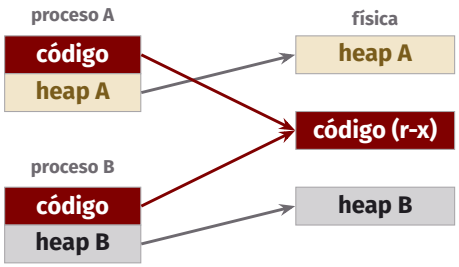
- Dos procesos del mismo programa tienen el mismo código.
- Sus segmentos de código pueden apuntar a la misma memoria física: una sola copia.

Compartir segmentos y bits de protección



- Dos procesos del mismo programa tienen el mismo código.
- Sus segmentos de código pueden apuntar a la misma memoria física: una sola copia.
- Para que sea seguro, cada segmento tiene bits de protección: lectura, escritura, ejecución.

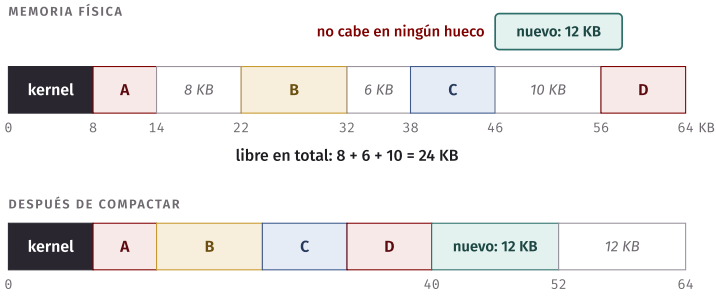
Compartir segmentos y bits de protección



- Dos procesos del mismo programa tienen el mismo código.
- Sus segmentos de código pueden apuntar a la misma memoria física: una sola copia.
- Para que sea seguro, cada segmento tiene bits de protección: lectura, escritura, ejecución.

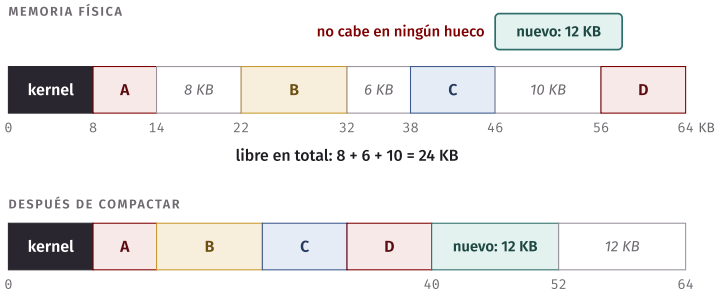
Seg.	Base	Prot.
Código	`0x5000`	`r-x`
Heap	`0xA000`	`rw-`
Pila	`0x3000`	`rw-`

Fragmentación externa



los 24 KB libres quedan contiguos y el segmento cabe; copiar todo es caro

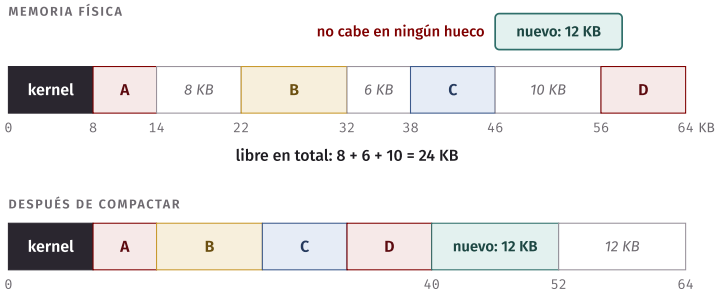
Fragmentación externa



los 24 KB libres quedan contiguos y el segmento cabe; copiar todo es caro

- Segmentos de tamaños distintos dejan la memoria libre en huecos: es la fragmentación externa *(external fragmentation)*.

Fragmentación externa



los 24 KB libres quedan contiguos y el segmento cabe; copiar todo es caro

- Segmentos de tamaños distintos dejan la memoria libre en huecos: es la fragmentación externa *(external fragmentation)*.
- Compactar (abajo) funciona, pero copiar segmentos y actualizar sus bases es muy caro. La otra salida, elegir bien el hueco (best fit, first fit, buddy…), es la semana que viene.

Balance de la segmentación

Resuelve

- El hueco entre heap y pila ya no gasta memoria física.
- Espacios de direcciones dispersos *(sparse)*.
- Compartir código entre procesos.

Balance de la segmentación

Resuelve

- El hueco entre heap y pila ya no gasta memoria física.
- Espacios de direcciones dispersos *(sparse)*.
- Compartir código entre procesos.

No resuelve

- La fragmentación externa.
- Un heap grande y poco usado sigue ocupando toda su memoria física.

Balance de la segmentación

Resuelve

- El hueco entre heap y pila ya no gasta memoria física.
- Espacios de direcciones dispersos *(sparse)*.
- Compartir código entre procesos.

No resuelve

- La fragmentación externa.
- Un heap grande y poco usado sigue ocupando toda su memoria física.

Hacia la paginación

El problema son los trozos de *tamaño variable*. Con trozos del mismo tamaño (páginas), la fragmentación externa desaparece. Semana 6.

SECCIÓN 05

Laboratorio: el planificador de xv6

05



Del temporizador al planificador

interrupción
del temporizador

Del temporizador al planificador



- `devintr()` (`kernel/trap.c`) ve que la causa es el temporizador (∼0,1 s) y retorna 2.

Del temporizador al planificador



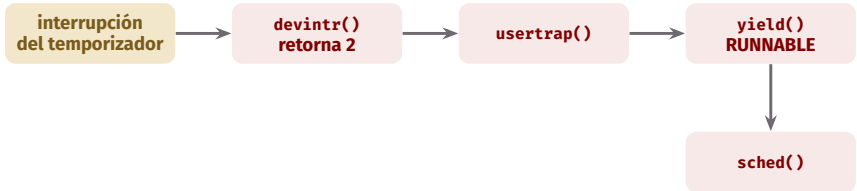
- `devintr()` (`kernel/trap.c`) ve que la causa es el temporizador (∼0,1 s) y retorna 2.

Del temporizador al planificador



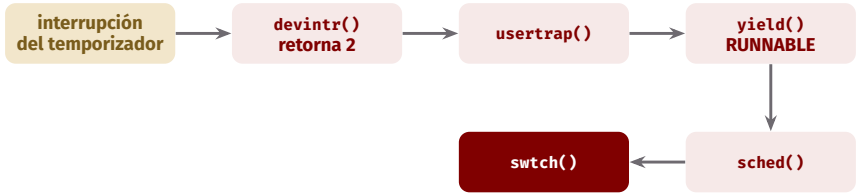
- `devintr()` (`kernel/trap.c`) ve que la causa es el temporizador (∼0,1 s) y retorna 2.
- `usertrap()` llama a `yield()`, que marca al proceso `RUNNABLE`.

Del temporizador al planificador



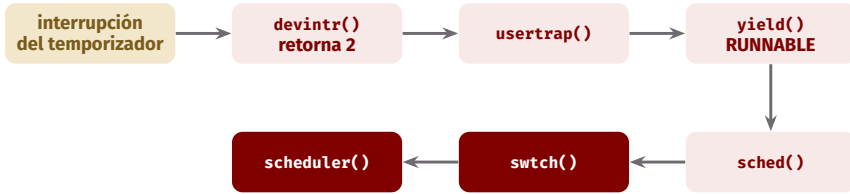
- `devintr()` (`kernel/trap.c`) ve que la causa es el temporizador (∼0,1 s) y retorna 2.
- `usertrap()` llama a `yield()`, que marca al proceso `RUNNABLE`.

Del temporizador al planificador



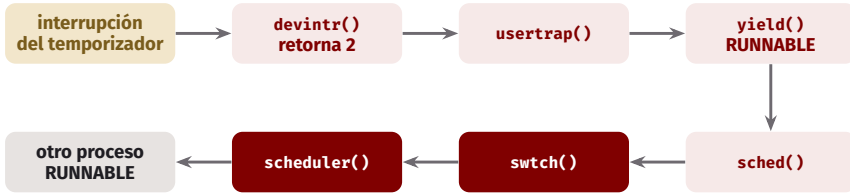
- `devintr()` (`kernel/trap.c`) ve que la causa es el temporizador (∼0,1 s) y retorna 2.
- `usertrap()` llama a `yield()`, que marca al proceso `RUNNABLE`.
- `swtch()` (`kernel/swtch.S`) guarda los registros del proceso y carga los del planificador.

Del temporizador al planificador



- `devintr()` (`kernel/trap.c`) ve que la causa es el temporizador (∼0,1 s) y retorna 2.
- `usertrap()` llama a `yield()`, que marca al proceso `RUNNABLE`.
- `swtch()` (`kernel/swtch.S`) guarda los registros del proceso y carga los del planificador.
- Cada CPU tiene su hilo `scheduler()`: elige al siguiente `RUNNABLE`. Esto es lo que van a cambiar en P2.

Del temporizador al planificador



- `devintr()` (`kernel/trap.c`) ve que la causa es el temporizador (∼0,1 s) y retorna 2.
- `usertrap()` llama a `yield()`, que marca al proceso `RUNNABLE`.
- `swtch()` (`kernel/swtch.S`) guarda los registros del proceso y carga los del planificador.
- Cada CPU tiene su hilo `scheduler()`: elige al siguiente `RUNNABLE`. Esto es lo que van a cambiar en P2.

`scheduler()`: round robin sobre la tabla

kernel/proc.c

```
1  for (;;) { // ... intr_on(); intr_off();
2    int found = 0;
3    for (p = proc; p < &proc[NPROC]; p++) {
4      acquire(&p->lock);
5      if (p->state == RUNNABLE) {
6        p->state = RUNNING; c->proc = p;
7        swtch(&c->context, &p->context);
8        // ... c->proc = 0;
9        found = 1;
10     }
11     release(&p->lock);
12   }
13   if (found == 0) asm volatile("wfi");
14 }
```

- Recorre `proc[NPROC]` en orden y corre a *cada* proceso `RUNNABLE`: round robin.

`scheduler()`: round robin sobre la tabla

kernel/proc.c

```
1  for (;;) { // ... intr_on(); intr_off();
2    int found = 0;
3    for (p = proc; p < &proc[NPROC]; p++) {
4      acquire(&p->lock);
5      if (p->state == RUNNABLE) {
6        p->state = RUNNING; c->proc = p;
7        swtch(&c->context, &p->context);
8        // ... c->proc = 0;
9        found = 1;
10     }
11     release(&p->lock);
12   }
13   if (found == 0) asm volatile("wfi");
14 }
```

- Recorre `proc[NPROC]` en orden y corre a *cada* proceso `RUNNABLE`: round robin.
- `swtch` retorna recién cuando el proceso cede la CPU con `sched()`.

`scheduler()`: round robin sobre la tabla

kernel/proc.c

```
1  for (;;) { // ... intr_on(); intr_off();
2    int found = 0;
3    for (p = proc; p < &proc[NPROC]; p++) {
4      acquire(&p->lock);
5      if (p->state == RUNNABLE) {
6        p->state = RUNNING; c->proc = p;
7        swtch(&c->context, &p->context);
8        // ... c->proc = 0;
9        found = 1;
10     }
11     release(&p->lock);
12   }
13   if (found == 0) asm volatile("wfi");
14 }
```

- Recorre `proc[NPROC]` en orden y corre a *cada* proceso `RUNNABLE`: round robin.
- `swtch` retorna recién cuando el proceso cede la CPU con `sched()`.
- Si nadie está listo, `wfi` duerme la CPU hasta la próxima interrupción.

`scheduler()`: round robin sobre la tabla

kernel/proc.c

```
1  for (;;) { // ... intr_on(); intr_off();
2    int found = 0;
3    for (p = proc; p < &proc[NPROC]; p++) {
4      acquire(&p->lock);
5      if (p->state == RUNNABLE) {
6        p->state = RUNNING; c->proc = p;
7        swtch(&c->context, &p->context);
8        // ... c->proc = 0;
9        found = 1;
10     }
11     release(&p->lock);
12   }
13   if (found == 0) asm volatile("wfi");
14 }
```

- Recorre `proc[NPROC]` en orden y corre a *cada* proceso `RUNNABLE`: round robin.
- `swtch` retorna recién cuando el proceso cede la CPU con `sched()`.
- Si nadie está listo, `wfi` duerme la CPU hasta la próxima interrupción.
- En P2, el bucle elige un proceso por sorteo, según sus boletos.

`yield()`, `sched()` y la llamada desde `usertrap()`

kernel/proc.c

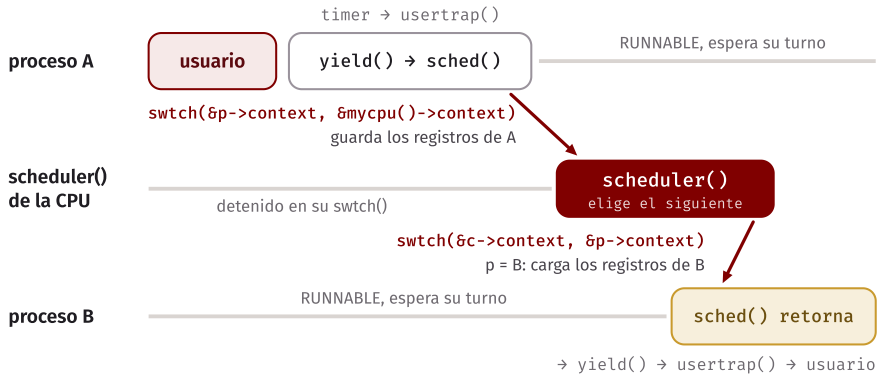
```
1  void yield(void) {
2    struct proc *p = myproc();
3    acquire(&p->lock);
4    p->state = RUNNABLE;
5    sched();
6    release(&p->lock);
7  }
8  void sched(void) {
9    // ... verificaciones con panic()
10   intena = mycpu()->intena;
11   swtch(&p->context, &mycpu()->context);
12   mycpu()->intena = intena;
13 }
```

kernel/trap.c

```
1  if (killed(p))
2    kexit(-1);
3
4  // give up the CPU if this
5  // is a timer interrupt.
6  if (which_dev == 2)
7    yield();
```

`which_dev` viene de `devintr()`. `kerneltrap()` hace lo mismo si la interrupción llega con el kernel corriendo.

Dos `swtch()` por cada cambio de proceso



A no salta directo a B: pasa por el hilo `scheduler()` de la CPU.

`kfork()`: cómo nace un proceso `RUNNABLE`

kernel/proc.c

```
1  if ((np = allocproc()) == 0)
2    return -1;
3  // copiar la memoria del padre
4  if (uvmcopy(p->pagetable, np->pagetable,
5              p->sz) < 0) { ... }
6  np->sz = p->sz;
7  *(np->trapframe) = *(p->trapframe);
8  np->trapframe->a0 = 0; // hijo: 0
9  // ... archivos, cwd, nombre, padre
10 acquire(&np->lock);
11 np->state = RUNNABLE;
12 release(&np->lock);
13 return pid;
```

- `allocproc()` toma una entrada `UNUSED` de la tabla.

`kfork()`: cómo nace un proceso `RUNNABLE`

kernel/proc.c

```
1  if ((np = allocproc()) == 0)
2    return -1;
3  // copiar la memoria del padre
4  if (uvmcopy(p->pagetable, np->pagetable,
5              p->sz) < 0) { ... }
6  np->sz = p->sz;
7  *(np->trapframe) = *(p->trapframe);
8  np->trapframe->a0 = 0; // hijo: 0
9  // ... archivos, cwd, nombre, padre
10 acquire(&np->lock);
11 np->state = RUNNABLE;
12 release(&np->lock);
13 return pid;
```

- `allocproc()` toma una entrada `UNUSED` de la tabla.
- `p->sz` es el tamaño de la memoria de usuario: lo que crece con `sbrk`.

`kfork()`: cómo nace un proceso `RUNNABLE`

kernel/proc.c

```
1  if ((np = allocproc()) == 0)
2    return -1;
3  // copiar la memoria del padre
4  if (uvmcopy(p->pagetable, np->pagetable,
5              p->sz) < 0) { ... }
6  np->sz = p->sz;
7  *(np->trapframe) = *(p->trapframe);
8  np->trapframe->a0 = 0; // hijo: 0
9  // ... archivos, cwd, nombre, padre
10 acquire(&np->lock);
11 np->state = RUNNABLE;
12 release(&np->lock);
13 return pid;
```

- `allocproc()` toma una entrada `UNUSED` de la tabla.
- `p->sz` es el tamaño de la memoria de usuario: lo que crece con `sbrk`.
- El hijo queda `RUNNABLE`: el planificador lo encontrará en su próxima vuelta.

`kfork()`: cómo nace un proceso `RUNNABLE`

kernel/proc.c

```
1  if ((np = allocproc()) == 0)
2    return -1;
3  // copiar la memoria del padre
4  if (uvmcopy(p->pagetable, np->pagetable,
5              p->sz) < 0) { ... }
6  np->sz = p->sz;
7  *(np->trapframe) = *(p->trapframe);
8  np->trapframe->a0 = 0; // hijo: 0
9  // ... archivos, cwd, nombre, padre
10 acquire(&np->lock);
11 np->state = RUNNABLE;
12 release(&np->lock);
13 return pid;
```

- `allocproc()` toma una entrada `UNUSED` de la tabla.
- `p->sz` es el tamaño de la memoria de usuario: lo que crece con `sbrk`.
- El hijo queda `RUNNABLE`: el planificador lo encontrará en su próxima vuelta.
- En P2, el hijo debe heredar los boletos del padre: es aquí.

Actividad: ver al planificador trabajar

kernel/proc.c: scheduler()

```
1  if (p->state == RUNNABLE) {
2    printk("[%d %s] ", p->pid, p->name)
       ;
3    // Switch to chosen process. ...
```

make qemu CPUS=1

```
init: starting sh
[2 init] [2 init] ... $ echo hola
[2 sh] [3 sh] [3 sh] [3 sh] [3 sh]
[3 sh] [3 sh] hola
[2 sh] $
```

1. En xv6-riscv se imprime desde el kernel con `printk`.

Actividad: ver al planificador trabajar

kernel/proc.c: scheduler()

```
1  if (p->state == RUNNABLE) {
2    printk("[%d %s] ", p->pid, p->name)
       ;
3    // Switch to chosen process. ...
```

make qemu CPUS=1

```
init: starting sh
[2 init] [2 init] ... $ echo hola
[2 sh] [3 sh] [3 sh] [3 sh] [3 sh]
[3 sh] [3 sh] hola
[2 sh] $
```

1. En xv6-riscv se imprime desde el kernel con `printk`.
2. La shell (PID 2) hace `fork`; el hijo (PID 3) aún se llama `sh` mientras carga `echo`.

Actividad: ver al planificador trabajar

kernel/proc.c: scheduler()

```
1  if (p->state == RUNNABLE) {
2    printk("[%d %s] ", p->pid, p->name)
       ;
3    // Switch to chosen process. ...
```

make qemu CPUS=1

```
init: starting sh
[2 init] [2 init] ... $ echo hola
[2 sh] [3 sh] [3 sh] [3 sh] [3 sh]
[3 sh] [3 sh] hola
[2 sh] $
```

1. En xv6-riscv se imprime desde el kernel con `printk`.
2. La shell (PID 2) hace `fork`; el hijo (PID 3) aún se llama `sh` mientras carga `echo`.
3. Cada `[3 sh]` es una vuelta del planificador: el hijo se duerme esperando al disco y vuelve a estar `RUNNABLE`.

Actividad: ver al planificador trabajar

kernel/proc.c: scheduler()

```
1  if (p->state == RUNNABLE) {
2    printk("[%d %s] ", p->pid, p->name)
       ;
3    // Switch to chosen process. ...
```

```
make qemu CPUS=1

init: starting sh
[2 init] [2 init] ... $ echo hola
[2 sh] [3 sh] [3 sh] [3 sh] [3 sh]
[3 sh] [3 sh] hola
[2 sh] $
```

1. En xv6-riscv se imprime desde el kernel con `printk`.
2. La shell (PID 2) hace `fork`; el hijo (PID 3) aún se llama `sh` mientras carga `echo`.
3. Cada `[3 sh]` es una vuelta del planificador: el hijo se duerme esperando al disco y vuelve a estar `RUNNABLE`.
4. Prueben con `CPUS=3` y con dos procesos en bucle: ¿se alternan?

Resumen

- El **espacio de direcciones** es la ilusión de una memoria privada: código, heap y pila. Toda dirección que ve un programa es *virtual*.
- `malloc`/`free` administran el heap sobre `brk` y `mmap`. Los errores de memoria rara vez fallan de inmediato: usen `valgrind`.
- Con **base y límite**, la MMU calcula $\textit{física} = \textit{virtual} + \textit{base}$ y verifica el límite en cada acceso. El kernel guarda y restaura los registros en cada cambio de contexto.
- La **segmentación** usa un par base/límite por segmento: no gasta memoria en huecos, permite compartir y proteger, pero sufre **fragmentación externa**.
- En xv6, la interrupción del temporizador lleva a `yield()` $\rightarrow$ `sched()` $\rightarrow$ `scheduler()`, que elige al siguiente proceso.

Para la próxima clase

- Leer OSTEP cap. 17 y 18.
- Laboratorio: `kernel/proc.c`, `kernel/trap.c` y un `printk` en `scheduler()`.
- `lottery.py` y `multi.py`: preguntas de los cap. 9 y 10.
- `make check` en `code/week-05`: expliquen cada error.

P2 publicado

Planificador por lotería en xv6-riscv. Se entrega en la semana 8. Examen parcial: semana 8, sesión B.

Pregunta para pensar

Si la pila de un proceso segmentado necesita crecer más allá de su tamaño, ¿quién se entera y qué puede hacer?