

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

Máquinas virtuales e hilos

Virtualizar la máquina entera · Varios flujos en un proceso

Gerson Garrido

Semana 9

Hoy

Sesión A: máquinas virtuales

1. Un OS debajo de otro OS: trap-and-emulate
2. Memoria: tablas sombra y paginación anidada
3. Paravirtualización, KVM, QEMU y RISC-V
4. Contenedores: namespaces y cgroups

Sesión B: hilos

5. Hilo vs. proceso: qué se comparte
6. `counter++`, instrucción por instrucción
7. La API: `pthread_create`, `join`, `mutex`, `cond`
8. xv6: un hilo por proceso, un kernel concurrente

Lectura

OSTEP: diálogo y apéndice *Virtual Machine Monitors* (sesión A); cap. 25–27, *Concurrency: An Introduction* y *Interlude: Thread API* (sesión B).

SECCIÓN 01

Virtualizar la CPU

01



Un sistema operativo para sistemas operativos

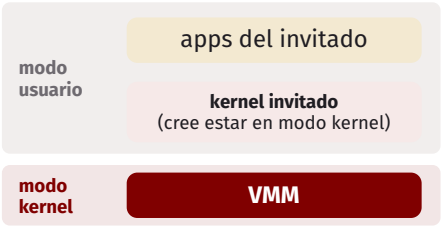




IBM System/370 modelo 145. Autor desconocido, CC0, Wikimedia Commons.

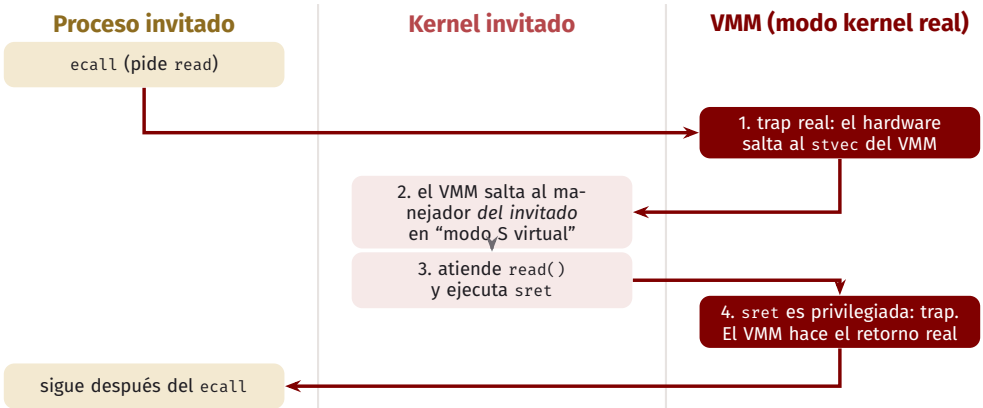
- El monitor de máquinas virtuales (VMM) o hipervisor *(hypervisor)* controla el hardware y lo reparte entre OS *invitados (guests)*.
- Cada invitado cree tener la máquina para él, sin ser modificado: **transparencia**.
- Para qué: consolidar servidores, aislar, probar, la nube. IBM VM/370 en los 70; Disco y VMware a fines de los 90.
- **Tipo 1**, sobre el hardware: Xen, ESXi, Hyper-V, Linux con KVM. **Tipo 2**, un programa sobre un OS anfitrión *(host)*: VirtualBox, QEMU.

La misma idea: ejecución directa limitada



- El invitado corre directamente en la CPU: sumas, saltos y cargas van a toda velocidad.
- Pero su kernel no puede correr en modo kernel *(kernel mode)*: controlaría la máquina. Corre *desprivilegiado*, en modo usuario *(user mode)*.
- El VMM recuerda, por cada CPU virtual, en qué modo *cree* estar el invitado.
- Cambiar de VM es como un cambio de contexto *(context switch)*, pero guarda también el estado privilegiado: `satp`, `stvec`, `sstatus`...

Trap-and-emulate: una llamada al sistema en una VM



Cada llamada al sistema *(system call)* cuesta dos traps extra al VMM.

Emular una instrucción privilegiada: el caso de xv6

kernel/vm.c (sin comentarios)

```
1  void
2  kvminithart()
3  {
4    sfence_vma();
5    w_satp(MAKE_SATP(kernel_pagetable));
6    sfence_vma();
7  }
```

En `kernel/riscv.h`, `w_satp` es una sola instrucción:

kernel/riscv.h

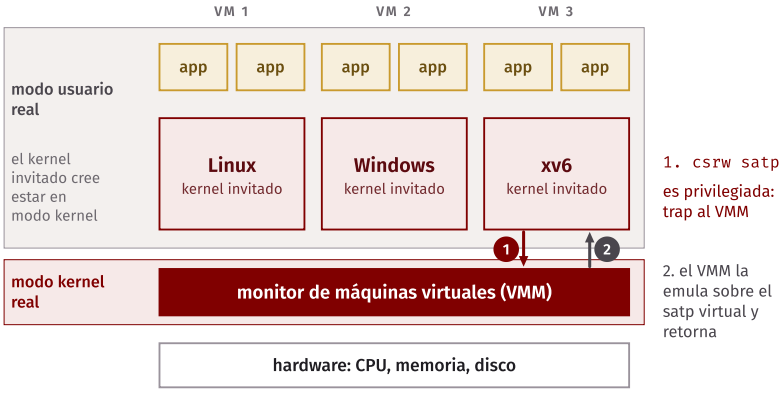
```
1  asm volatile("csrw satp, %0" : : "r"(x));
```

Si xv6 es un invitado, `csrw satp` se ejecuta en modo usuario real:

1. Excepción de instrucción ilegal: trap al VMM.
2. El VMM guarda el valor en el `satp` *virtual* de esa vCPU.
3. Prepara la traducción real (sección siguiente).
4. Avanza el PC del invitado 4 bytes y retorna.

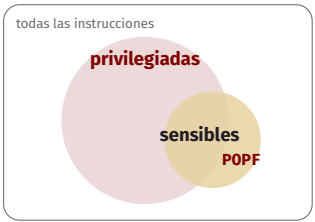
Emular: hacer en software lo que la instrucción haría en la máquina *virtual*.

Trap-and-emulate, de un vistazo



Solo el VMM corre en modo kernel real; cada instrucción privilegiada del invitado es un trap que el VMM emula.

¿Toda CPU se puede virtualizar así?



x86 de 32 bits: no se puede de la forma clásica.

- **Privilegiada**: en modo usuario provoca un trap.
- **Sensible**: lee o cambia el estado de la máquina: modo, `satp`, interrupciones *(interrupts)*.
- **Popek y Goldberg (1974)**: trap-and-emulate funciona si toda instrucción sensible es privilegiada.
- En x86, `POPF` en modo usuario ignora en silencio el bit de interrupciones. No hay trap: el VMM no se entera.

Tres salidas

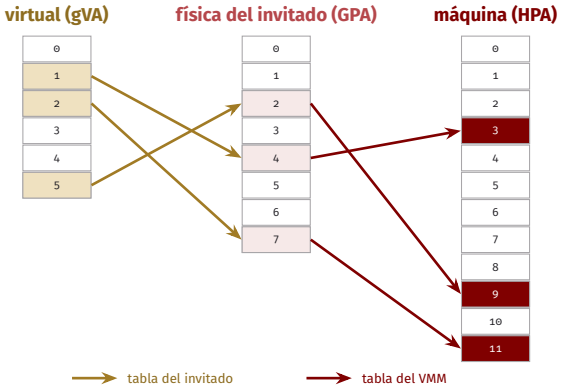
Traducción binaria del kernel invitado (VMware, 1999) · **paravirtualización** (Xen, 2003) · **soporte de hardware** (VT-x, AMD-V, 2005–2006): un modo para el invitado.

SECCIÓN 02

Virtualizar la memoria

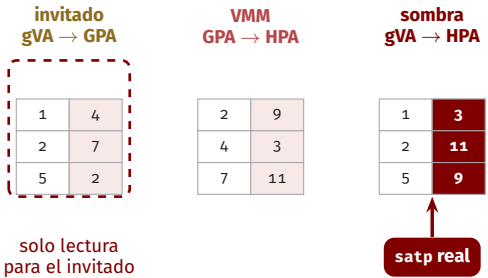
02

Tres tipos de dirección



- El kernel invitado traduce con *su* tabla de páginas *(page table)*.
- Pero su "memoria física" también es virtual: el VMM reparte la memoria real entre las VMs.
- El VMM tiene otra tabla por VM: física del invitado → máquina.
- La MMU sabe hacer *una* traducción. ¿Cómo llega de gVA a HPA?

Solución por software: tablas de páginas sombra



- La MMU usa la sombra: *una* traducción, gVA → HPA.
- El invitado cree usar la suya: su `satp` es virtual.
- Su tabla queda protegida contra escritura: cada cambio a una PTE es un trap y el VMM actualiza la sombra.

Hardware al rescate: paginación anidada

El costo de las tablas sombra

- Cada escritura de una PTE del invitado es un trap: `fork`, `exec`, `sbrk`…
- Fallos de página *(page faults)* "ocultos" que el invitado no ve.
- Una sombra por proceso del invitado; los bits A y D, sincronizados a mano.

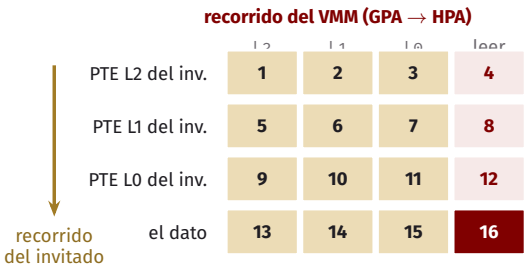
Paginación anidada *(nested paging)*

- La MMU conoce dos tablas y las recorre ella misma.
- El invitado cambia sus PTE y su `satp` sin traps.
- Intel EPT, AMD NPT (2007–2008); en RISC-V, la etapa G *(G-stage)*.



Registros de RISC-V. En Intel: el `CR3` del invitado y el puntero EPT.

El recorrido en dos dimensiones



La PTE L0 da la GPA del dato: 3 accesos más y, por fin, el dato.

Cuenta

n niveles en el invitado y m en el VMM: $n(m+1)+m$ accesos. Sv39: $3 \cdot 4 + 3 = 15$. x86-64: $4 \cdot 5 + 4 = 24$.

La TLB guarda gVA → HPA: el recorrido largo solo se paga en un fallo.

SECCIÓN 03

Máquinas virtuales hoy

03

Paravirtualización: un invitado que sabe que es invitado

Paravirtualización

Se modifica el kernel invitado para que pida al VMM lo que necesita, con hiperllamadas *(hypercalls)*, en vez de usar instrucciones que hay que atrapar.

- Xen (2003): Linux modificado, casi a velocidad nativa en x86.
- Cierra la brecha de información: el VMM no sabe si el invitado está ocioso o si va a borrar una página ya limpia.

Hoy se paravirtualiza sobre todo la E/S: virtio. Así arranca QEMU el disco de xv6:

Makefile de xv6 (recortado)

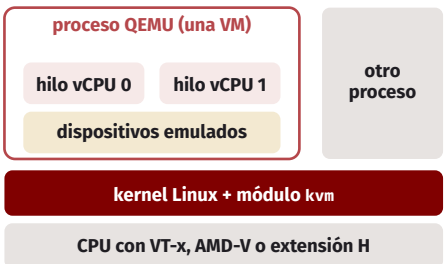
```
-device virtio-blk-device,drive=x0,...
```

y lo maneja `kernel/virtio_disk.c`.

xv6 ya es un invitado paravirtual

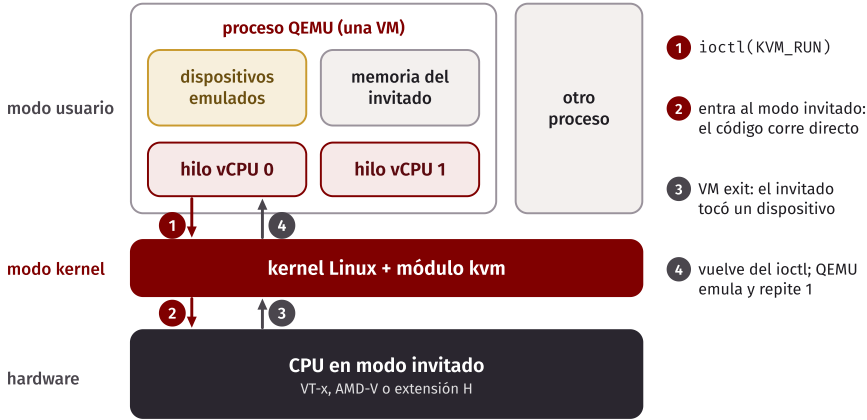
Su disco no imita uno real: es una cola de pedidos en memoria compartida con QEMU.

KVM + QEMU: cómo se virtualiza hoy en Linux



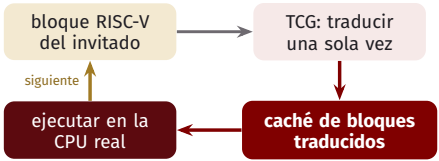
- **KVM**: módulo del kernel Linux (2007) que usa el modo invitado del hardware. Linux pasa a ser el hipervisor.
- Cada CPU virtual es un hilo de QEMU que llama `ioctl(KVM_RUN)`: el código del invitado corre *en el hardware*.
- Si el invitado toca un dispositivo, hay una salida *(VM exit)*: QEMU emula el dispositivo y vuelve.
- Así funcionan las VMs de la nube. Linux planifica las vCPU como a cualquier hilo.

El ciclo de una vCPU con KVM



La VM es un proceso más: Linux planifica sus hilos vCPU como a cualquier otro hilo.

Nuestro QEMU no usa KVM: traducción binaria



- Cada bloque básico se traduce a código del anfitrión y se guarda: no interpreta instrucción por instrucción.
- Cargas y guardados pasan por una TLB por software; `csrw`, `ecall` y `sret` llaman a funciones que emulan su efecto.

contenedor sistemas-operativos

```
$ uname -m
aarch64
$ qemu-system-riscv64 -accel help
Accelerators supported in QEMU binary:
tcg
$ ls /dev/kvm
ls: cannot access '/dev/kvm': No such ...
```

¿Por qué no KVM?

KVM corre el invitado *en* la CPU real: misma arquitectura o nada. xv6 es RISC-V; las laptops, no.

QEMU es un proceso con hilos

contenedor del curso, xv6 con -smp 3

```
$ ps -L -o pid,lwp,pcpu,comm -p 437
  PID   LWP %CPU COMMAND
  437   437  0.1 qemu-system-ris
  437   438  0.0 qemu-system-ris
  437   439  6.5 CPU 0/TCG
  437   440  6.6 CPU 1/TCG
  437   441  6.3 CPU 2/TCG
  437   442  0.0 worker
```

Mismas opciones que `make qemu`, más `-name xv6,debug-threads=on` para que QEMU ponga nombre a sus hilos.

- Un solo proceso (PID 437), seis hilos (LWP).
- Uno por CPU virtual: las tres harts de xv6 corren *en paralelo* en tres CPU reales.
- Los tres comparten la memoria del invitado: tienen que sincronizarse.

Puente a la sesión B

Un proceso, varios flujos de ejecución, una memoria: eso es un hilo.

RISC-V y la extensión de hipervisor (H)



- El modo S pasa a ser **HS** y aparecen dos modos virtuales: **VS** y **VU**.
- En VS, el `csrw satp` de xv6 escribe en `vsatp` sin trap: el hardware ya sabe que es un invitado.
- Traducción en dos etapas: `vsatp` (gVA → GPA) y `hgatp` (GPA → HPA), como EPT.
- Ratificada en 2021. xv6 no la usa; KVM sí, en anfitriones RISC-V.

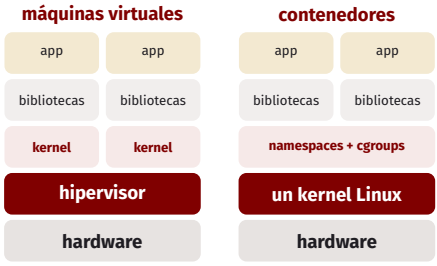
SECCIÓN 04

Contenedores

04



Contenedores vs. máquinas virtuales



	VM	Contenedor
Kernel	uno por VM	compartido
Arranque	segundos	milisegundos
Memoria extra	cientos de MB	casi nada
Otro OS	sí (Windows)	no
Aislamiento	hardware	kernel

Un contenedor *(container)* no es una VM

Es un proceso del kernel anfitrión, con una vista restringida del sistema. No hay hipervisor ni segundo kernel.

Las dos piezas: namespaces y cgroups

Namespace: qué puede *ver*

El kernel le da al proceso su propia copia de un recurso global.

`pid`	sus propios PIDs: el primero es el 1
`mnt`	su propio árbol de archivos
`net`	sus interfaces y puertos
`uts`	su propio `hostname`
`user`	sus propios UID (`root` adentro)
`cgroup`	su propia raíz de cgroups
`ipc, time`	…

cgroup: cuánto puede *usar*

Un grupo de procesos con límites y contabilidad de CPU, memoria y E/S.

- `docker run --memory=256m` escribe un límite en `memory.max`.
- `--cpus=1.5`: 150 ms de CPU cada 100 ms, en `cpu.max`.
- El planificador y la memoria virtual son los de siempre: un contenedor es un proceso más aislado.

Véanlo en nuestro contenedor

contenedor del curso, en macOS

```
$ uname -r
6.12.72-linuxkit
$ ls -l /proc/self/ns      # recortado
mnt -> mnt:[4026533186]
net -> net:[4026533199]
pid -> pid:[4026533193]
user -> user:[4026531837]
uts -> uts:[4026533189]
$ cat /proc/self/cgroup
0::/
$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
   11 ?        00:00:00 ps
```

- `uname -r` no dice “Darwin”: el kernel es Linux. Docker Desktop corre una VM Linux liviana y los contenedores viven *dentro* de ella.
- Cada número es un namespace. Dos procesos con el mismo número comparten ese namespace.
- `bash` es el PID 1: en su namespace de PID no hay nadie más.
- `0::/`: el contenedor ve la raíz de *su* cgroup, no la del anfitrión.

Dos contenedores, un kernel

	contenedor A	contenedor B	VM Linux
pid	4026533193	4026533470	4026531836
net	4026533199	4026533472	4026531840
uts	4026533189	4026533468	4026531838
user	4026531837	4026531837	4026531837

un contenedor C que corre sleep 60

```
$ docker exec C ps -eo pid,comm
  PID COMMAND
    1 sleep
$ ps -eo pid,comm | grep -w sleep   # en la VM
49346 sleep
```

- A y B, vivos a la vez, tienen namespaces distintos entre sí y de la VM.
- `user` es el mismo: Docker no crea uno por defecto. El `root` del contenedor es `root` para el kernel.
- Un proceso, dos PIDs: 1 adentro, 49346 afuera.
- Con `--memory=256m`, `memory.max` vale `268435456`.

Las capas del entorno del curso



- Una VM con virtualización por hardware.
- Un contenedor, que *no* agrega kernel.
- Una máquina RISC-V emulada por traducción binaria.
- Arriba, el kernel que ustedes modifican.

En Linux nativo

Sin VM de por medio: `uname -r` muestra el kernel de su computadora.

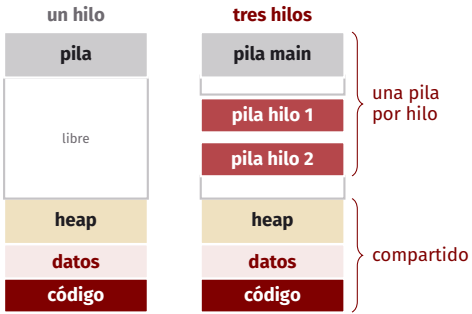
SECCIÓN 05

Hilos

05



Un proceso, varios hilos



Hilo ***(thread)***

Un flujo de ejecución dentro de un proceso: su propio PC, registros y pila *(stack)*, y el mismo espacio de direcciones *(address space)* que los demás hilos.

- Su estado va en un TCB *(thread control block)*, como el PCB de un proceso.
- Cambiar de hilo *no* cambia la tabla de páginas: es más barato.
- Para qué: paralelismo y no bloquear todo el programa en una E/S.

¿Qué se comparte y qué no?

Compartido	Propio de cada hilo
código	PC y registros
variables globales	pila: variables locales
heap	identificador (TID)
archivos abiertos	`errno`
PID, directorio actual	máscara de señales

- Lo compartido es lo que facilita cooperar: un hilo escribe en el heap y los demás lo ven, sin pipes ni copias.
- Lo compartido también es lo peligroso: dos hilos pueden modificar lo mismo a la vez.
- "Propio" no es "protegido": un hilo puede leer la pila de otro si tiene un puntero a ella.

Comparen con `fork`

Después de `fork`, padre e hijo tienen *copias* de todo. Con hilos, una sola copia de todo menos la pila y los registros.

Una pila por hilo: medido

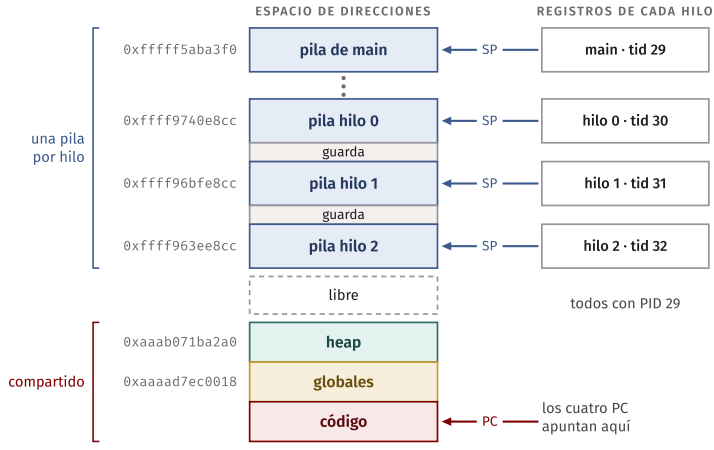
stacks.c: cada hilo anota la dirección de una variable local

```
$ ./stacks 3
global       0xaaaad7ec0018
heap         0xaaab071ba2a0
main  local  0xffff5aba3f0   pid 29  tid 29
hilo 0 local  0xffff9740e8cc   pid 29  tid 30
hilo 1 local  0xffff96bfe8cc   pid 29  tid 31   (a 8256 KiB del hilo 0)
hilo 2 local  0xffff963ee8cc   pid 29  tid 32   (a 8256 KiB del hilo 1)
```

- `global` y `heap` son una sola dirección para todos.
- Cada `local` vive en una pila distinta, separadas por unos 8 MiB (`ulimit -s`) más una zona de guarda. En su máquina el número puede variar.
- Mismo PID, distinto TID: para Linux, cada hilo es una tarea que se planifica aparte.

Los hilos esperan en una barrera para estar vivos a la vez: si uno termina antes, la biblioteca puede reutilizar su pila para el siguiente.

Cuatro pilas, un espacio de direcciones



Crear y esperar hilos

pthread.h

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
int pthread_join(pthread_t thread, void **value_ptr);
```

- `thread`: dónde guardar el identificador. `attr`: casi siempre `NULL`.
- `start_routine`: la función del hilo. Recibe y devuelve `void *`: así se pasa *cualquier* tipo.
- `pthread_join`: espera a que el hilo termine y recibe lo que devolvió. Es el `wait()` de los hilos. Se compila con `gcc -pthread`.

El orden no está garantizado

¿Corre primero el hilo nuevo o `main`? Lo decide el planificador (*scheduler*).

Pasar argumentos y recibir resultados

args.c (recortado)

```c
typedef struct { int id; long from, to; } task_t;
typedef struct { long sum; } result_t;

void *sum_range(void *arg) {
    task_t *t = arg;            // void * -> task_t *
    result_t *r = malloc(sizeof(*r));  // heap
    r->sum = 0;
    for (long i = t->from; i <= t->to; i++)
        r->sum += i;
    return r;
}
// main: un task_t por hilo
pthread_create(&th[i], NULL, sum_range, &tasks[i]);
void *ret;
pthread_join(th[i], &ret);
total += ((result_t *)ret)->sum;
free(ret);
```

args.c

```
$ ./args 100 3
hilo 0: [1, 33] -> 561
hilo 1: [34, 66] -> 1650
hilo 2: [67, 100] -> 2839
total = 5050   (N(N+1)/2 = 5050)
```

- Cada hilo recibe su propio `task_t`.
- El resultado va en el heap: sobrevive al hilo. Lo libera quien hace `join`.

Dos errores clásicos

1. Devolver un puntero a la pila

mal.c

```c
void *worker(void *arg) {
    result_t r = { .sum = 42 };
    return &r;     // MAL
}
```

Al terminar el hilo, su pila se libera o se reutiliza: `join` recibe un puntero a basura. Usen `malloc`.

2. Pasar la dirección de `i`

mal.c

```c
for (int i = 0; i < n; i++)
    pthread_create(&th[i], NULL,
                   worker, &i);  // MAL
```

Todos reciben la misma dirección; cuando la leen, `i` quizás ya cambió. ¡Una condición de carrera! Usen un argumento por hilo.

Regla: lo que se pasa a un hilo, o lo que este devuelve, debe seguir vivo mientras alguien lo use.

SECCIÓN 06

Condiciones de carrera

06

El problema: `counter++` con varios hilos

race.c

```c
static volatile long counter = 0;

void *worker_nolock(void *arg) {
    for (long i = 0; i < loops; i++)
        counter++;
    return NULL;
}
```

contenedor sistemas-operativos, 14 CPU

```
$ ./race nolock 1 1000000
... esperado=1000000  obtenido=1000000  OK
$ ./race nolock 4 1000000
... esperado=4000000  obtenido=1928101  MAL
```

counter-rv.s (make asm)

```
increment:
    lla   a4,.LANCHOR0   # &counter
    ld    a5,0(a4)       # leer
    addi  a5,a5,1        # sumar
    sd    a5,0(a4)       # escribir
    ret
```

- Compilado con el `riscv64-linux-gnu-gcc` del contenedor: tres instrucciones, no una.
- Entre ellas puede llegar una interrupción, u otra CPU puede tocar `counter`.

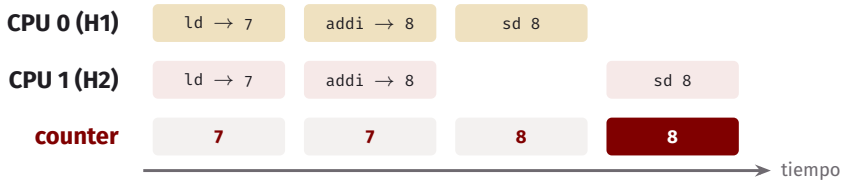
Una actualización perdida, instrucción por instrucción

paso	hilo 1	a5 (H1)	hilo 2	a5 (H2)	counter
		estado inicial: counter = 7			7
1	`ld a5,0(a4)`	7			7
2	`addi a5,a5,1`	8			7
	— interrupción del temporizador: el kernel guarda a5 = 8 de H1 y pasa a H2 —				
3			`ld a5,0(a4)`	7	7
4			`addi a5,a5,1`	8	7
5			`sd a5,0(a4)`	8	**8**
	— interrupción: vuelve H1 con a5 = 8 restaurado —				
6	`sd a5,0(a4)`	8			**8**

Dos incrementos, el contador subió solo 1

Se esperaba 9 y quedó 8. Depende de cuándo llega la interrupción: **no determinista**.

Con dos CPU no hace falta ni una interrupción



- Los dos hilos corren a la vez y leen el mismo valor.
- Otra vez 8 en lugar de 9. Con 4 hilos se perdió la mitad de las sumas.

`volatile` no lo arregla

Solo obliga al compilador a leer y escribir la memoria cada vez. Siguen siendo tres instrucciones. Lo que falta es **atomicidad**.

Vocabulario

Condición de carrera *(race condition)*

El resultado depende del orden en que se intercalan los hilos.

Exclusión mutua *(mutual exclusion)*

Si un hilo está en la sección crítica, ningún otro entra. Se logra con un lock.

Sección crítica *(critical section)*

Código que accede a un recurso compartido y que no deben ejecutar dos hilos a la vez.

Atomicidad

Un grupo de operaciones ocurre “todo o nada”: nadie ve un estado intermedio.

Dos problemas distintos: **atomicidad**, que nadie se meta en medio (mutex), y **esperar** a que algo pase primero (variables de condición *(condition variables)*).

SECCIÓN 07

La API de pthreads

07



Mutex: exclusión mutua

race.c

```c
pthread_mutex_t lock =
    PTHREAD_MUTEX_INITIALIZER;

void *worker_mutex(void *arg) {
    for (long i = 0; i < loops; i++) {
        pthread_mutex_lock(&lock);
        counter++;      // sección crítica
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

Si otro hilo tiene el lock, `pthread_mutex_lock` espera hasta que lo suelte.

modo	obtenido	tiempo
`nolock`	12 621 040	0.039 s
`mutex`	40 000 000	1.342 s
`local`	40 000 000	0.007 s

`./race <modo> 4 10000000`: esperado 40 000 000.

- Correcto, pero 35 veces más lento: los 4 hilos se pelean por el lock en cada vuelta.
- `local`: cada hilo suma en su pila y toma el lock *una* vez. Más sobre esto en la semana 10.

Variables de condición: esperar a que algo pase

cond.c (recortado)

```c
// hilo que espera (main)
pthread_mutex_lock(&lock);
while (!ready)
    pthread_cond_wait(&ready_cv, &lock);
printf("main: listo, data = %d\n", data);
pthread_mutex_unlock(&lock);

// hilo que avisa (producer)
pthread_mutex_lock(&lock);
data = 42;
ready = 1;
pthread_cond_signal(&ready_cv);
pthread_mutex_unlock(&lock);
```

- `wait` suelta el lock y duerme; al despertar, lo vuelve a tomar.
- Lo que importa es el estado (`ready`); la variable solo sirve para dormir y despertar.
- `while`, no `if`: al despertar, volver a verificar.
- Cambiar el estado y avisar con el lock tomado.

cond.c

```
$ ./cond
main: esperando...
main: listo, data = 42
```

Errores comunes con la API

Mutex

- No inicializarlo (`PTHREAD_MUTEX_INITIALIZER`).
- Un `return` dentro de la sección crítica, sin `unlock`.
- Proteger el mismo dato con locks distintos.
- Ignorar los códigos de retorno.

Esperar

- Girar sobre una bandera sin lock: gasta CPU y puede fallar.
- `if` en lugar de `while` alrededor de `wait`.
- `signal` sin cambiar antes el estado: si nadie espera, el aviso se pierde.

helgrind detecta la carrera

```
$ valgrind --tool=helgrind ./race nolock 2 1000
==13== Possible data race during read of size 8 at 0x128018 by thread #3
==13==    at 0x108BB0: worker_nolock (race.c:23)
==13== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 1 from 1)
```

Salida recortada. Con `mutex`: `ERROR SUMMARY: 0 errors`.

SECCIÓN 08

Concurrencia en xv6

08



xv6: un hilo por proceso, un kernel concurrente

- xv6 no tiene hilos de usuario. Cada `struct proc` tiene un solo `trapframe`, un `context` y una pila de kernel (`kstack`).
- Pero corre con `-smp 3`: tres harts ejecutan el kernel *a la vez*, sobre los mismos datos.

```
make qemu

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
```

kernel/proc.c

```
1  int
2  allocpid()
3  {
4    int pid;
5
6    acquire(&pid_lock);
7    pid = nextpid;
8    nextpid = nextpid + 1;
9    release(&pid_lock);
10
11   return pid;
12 }
```

¡Es nuestro `counter++`! Sin el lock, dos `fork` en dos harts podrían recibir el mismo PID.

Un adelanto: el spinlock de xv6

kernel/spinlock.c (recortado)

```
1  void
2  acquire(struct spinlock *lk)
3  {
4    push_off(); // disable interrupts to avoid deadlock.
5    // ...
6    while (__atomic_exchange_n(&lk->locked, 1,
7             __ATOMIC_ACQUIRE) != 0)
8      ;
9    lk->cpu = mycpu();
10 }
```

`release()` hace lo inverso: `__atomic_store_n(&lk->locked, 0, __ATOMIC_RELEASE)` y `pop_off()`.

- Se compila a `amoswap.w.aq`: escribe 1 y devuelve el valor anterior, en una instrucción atómica.
- Si era 1, otro tiene el lock: girar *(spin)*.
- `push_off`: sin interrupciones en esta CPU mientras se tiene el lock.
- Detalles: semana 10.

Resumen

- Un **VMM** usa ejecución directa limitada: el kernel invitado corre desprivilegiado y sus instrucciones privilegiadas se atrapan y **emulan**. x86 necesitó traducción binaria, paravirtualización o VT-x/AMD-V.
- Memoria: gVA $\rightarrow$ GPA $\rightarrow$ HPA. **Tablas sombra** por software, o **paginación anidada** (EPT/NPT, `hgatp`): hasta 15 accesos con Sv39.
- KVM usa el hardware; nuestro QEMU usa **TCG** (traducción binaria) porque xv6 es RISC-V. Un **contenedor** es un proceso con namespaces y cgroups: un solo kernel.
- Un **hilo** tiene su PC, registros y pila, y comparte el espacio de direcciones del proceso.
- `counter++` son tres instrucciones: sin **exclusión mutua** se pierden actualizaciones. Mutex para la atomicidad; variables de condición, con `while`, para esperar.
- xv6 no tiene hilos de usuario, pero su kernel es concurrente y usa spinlocks.

Para la próxima clase

- Leer OSTEP cap. 28: *Locks*.
- Laboratorio: `x86.py` (preguntas 1–8 del cap. 26) con `loop.s`, `looping-race-nolock.s` y `wait-for-me.s`. Usa instrucciones al estilo x86 (`mov`, `add`): son los mismos tres pasos.
- `main-*.c` del cap. 27 con `helgrind`: encontrar carreras y deadlocks.
- Prueben `code/week-09`: `make demo` y `./stacks 4`.

Entregas

P3 (lab `pgtbl`) se entrega la próxima semana. En la semana 10 se publica P4, `pzip`: compresión en paralelo con hilos.

Pregunta para pensar

Dos hilos ejecutan `counter++` 3 veces cada uno, empezando en 0. ¿Cuál es el valor final más *pequeño* posible? (No es 3.)