

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

Introducción y procesos

¿Qué hace un sistema operativo?

Gerson Garrido

Semana 1

Hoy

1. El curso: cómo funciona y qué vamos a construir
2. ¿Qué es un sistema operativo?
3. Cuatro demostraciones: CPU, memoria, concurrencia y persistencia
4. Un poco de historia y cómo se organiza un kernel
5. La abstracción de proceso

> **Lectura**
> OSTEP cap. 1–2 (introducción) y cap. 4 *The Abstraction: The Process*.

SECCIÓN 01

El curso

01

Tres piezas fáciles

Virtualización

La CPU y la memoria parecen exclusivas de cada programa

Concurrencia

Muchas cosas pasan a la vez, sin romperse

Persistencia

Los datos sobreviven a un apagón

+ máquinas virtuales, sistemas distribuidos y seguridad

Cómo funciona el curso

- **Teoría** (4 h): dos sesiones por semana.
- **Laboratorio** (2 h): práctica guiada en el contenedor del curso.
- **Proyectos**: C en Linux y el kernel xv6 sobre RISC-V.
- **Exámenes**: parcial en la semana 8 y final al cierre del curso.
- **Lectura**: OSTEP, gratis en línea, y los apuntes del curso.

Todo está en el sitio del curso

Calendario, guías semanales, apuntes, diapositivas, código y enunciados.

Lo que van a construir

	Proyecto	Dónde
P0	Utilidades Unix: `wcat`, `wgrep`, `wzip`	C en Linux
P1	Una shell: `wish`	C en Linux
P2	Planificador por lotería	kernel xv6
P3	Tablas de páginas	kernel xv6
P4	Compresión en paralelo: `pzip`	C + hilos
P5	Archivos grandes y enlaces simbólicos	kernel xv6

Bono: `fork` con copy-on-write en xv6.

SECCIÓN 02

¿Qué es un sistema operativo?

02

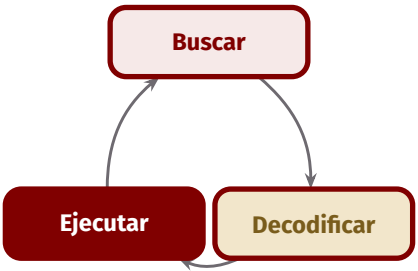


¿Qué pasa cuando un programa se ejecuta?

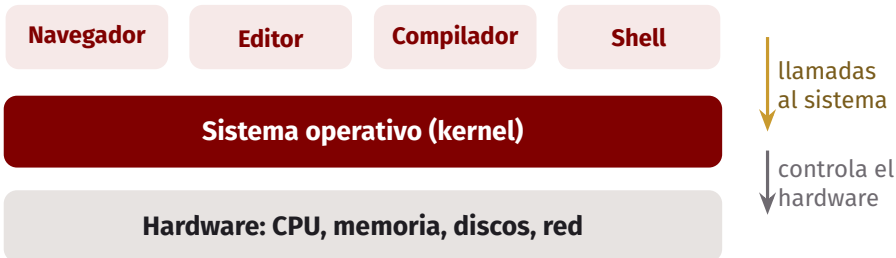
El procesador repite, miles de millones de veces por segundo:

1. Buscar la siguiente instrucción en memoria *(fetch)*
2. Decodificarla *(decode)*
3. Ejecutarla: sumar, comparar, saltar, leer o escribir memoria *(execute)*

Así funciona el modelo de Von Neumann. ¿Dónde entra el sistema operativo?



El sistema operativo es una capa



Los programas no tocan el hardware directamente: piden al kernel.

Tres trabajos del sistema operativo

1. Virtualizar

Convertir un recurso físico (CPU, memoria) en una versión virtual más fácil de usar, que parece exclusiva de cada programa.

2. Administrar recursos

Decidir quién usa la CPU, cuánta memoria recibe y en qué orden se atienden los discos.

3. Ofrecer una interfaz

Llamadas al sistema (`open`, `read`, `fork`…): la *biblioteca estándar* de la máquina.

Demostración 1: virtualizar la CPU

cpu.c

```c
int main(int argc, char *argv[]) {
  for (int i = 0; i < 5; i++) {
    spin(1);
    printf("%s\n", argv[1]);
  }
  return 0;
}
```

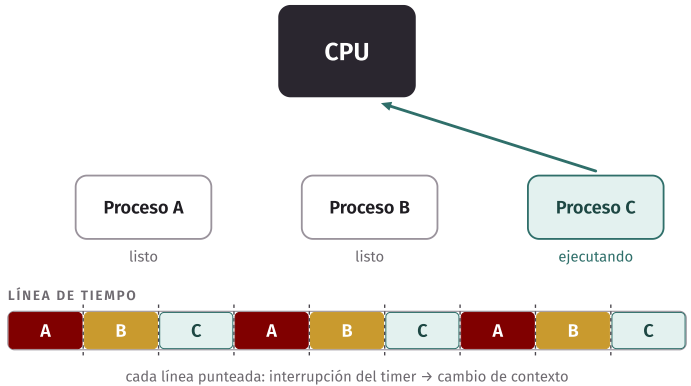
bash

```
$ ./cpu A & ./cpu B &
A
B
B
A
...
```

La ilusión

Cada programa cree tener una CPU para él: virtualización de la CPU.

La ilusión: tiempo compartido



cada línea punteada: interrupción del timer → cambio de contexto

Demostración 2: virtualizar la memoria

mem.c

```c
int *p = malloc(sizeof(int));
*p = atoi(argv[1]);
printf("dirección: %p\n", p);
for (int i = 0; i < 4; i++) {
  spin(1);
  *p = *p + 1;
  printf("*p = %d\n", *p);
}
```

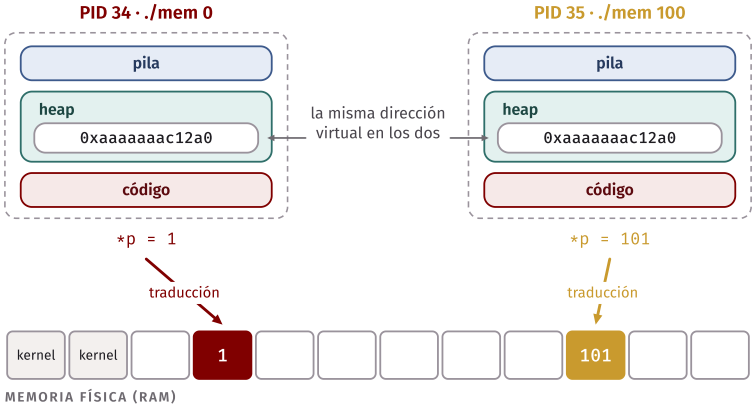
bash

```
$ setarch -R ./mem 0 &
$ setarch -R ./mem 100 &
(34) dirección: 0xaaaaaaaac12a0
(35) dirección: 0xaaaaaaaac12a0
(34) *p = 1
(35) *p = 101
```

Misma dirección, valores distintos

Cada proceso tiene su propio espacio de direcciones: las direcciones son *virtuales*.

Misma dirección virtual, distinta memoria física



el sistema operativo y el hardware (MMU) llevan cada dirección virtual a un marco físico distinto

Demostración 3: concurrencia

threads.c

```c
static volatile long counter = 0;

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

bash

```
$ ./threads 1000
esperado: 2000
obtenido: 2000
$ ./threads 1000000
esperado: 2000000
obtenido: 1080962
```

¿Se perdieron casi un millón de sumas?

`counter++` es *leer*, *sumar* y *escribir*: si dos hilos se intercalan, se pierden sumas.

Demostración 4: persistencia

io.c

```c
int flags = O_WRONLY | O_CREAT | O_TRUNC;
int fd = open("hola.txt", flags, 0644);
write(fd, "hola, disco\n", 12);
fsync(fd);  // forzar al disco
close(fd);
```

- La memoria se borra al apagar la computadora; el disco no.
- El sistema de archivos decide dónde van los datos en el disco.
- ¿Y si se va la luz *a mitad* de la escritura?

Persistencia

Guardar datos de forma confiable y eficiente, aunque el sistema falle: parte III del curso.

Metas de diseño

- **Abstracciones** que hagan fácil usar la máquina.
- **Rendimiento**: el sistema operativo debe costar poco.
- **Protección** y aislamiento entre programas.
- **Confiabilidad**: si el kernel falla, falla todo.
- **Eficiencia energética**: celulares, laptops, centros de datos.
- **Seguridad** frente a programas maliciosos.

Un tema que se repite

Casi todas las decisiones de diseño son un compromiso: más protección cuesta rendimiento, más rendimiento cuesta simplicidad.

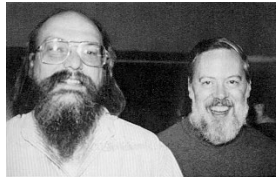
SECCIÓN 03

Un poco de historia

03

De las bibliotecas a los sistemas multiusuario

- **1950s**: el “sistema operativo” es una biblioteca de rutinas de E/S. Un operador carga trabajos por lotes *(batch)*.
- **1960s**: aparece la protección: llamadas al sistema, modo kernel. Multiprogramación: mientras un trabajo espera E/S, corre otro.
- **1969**: Ken Thompson y Dennis Ritchie crean Unix en Bell Labs. Ideas simples y combinables: procesos, archivos, pipes, la shell.



Thompson y Ritchie, 1973. Dominio público, Wikimedia Commons.



PDP-11/40. Stefan Kögl, CC BY-SA 3.0, Wikimedia Commons.

De la PC al celular

- **1980s**: computadoras personales. Los primeros sistemas (DOS, Mac OS clásico) olvidan la protección… y se cuelgan.
- **1991**: Linus Torvalds publica Linux, un kernel tipo Unix libre.
- **Hoy**: Linux en servidores, Android, routers y supercomputadoras. macOS e iOS también descienden de Unix. Las ideas de 1970 siguen vigentes.



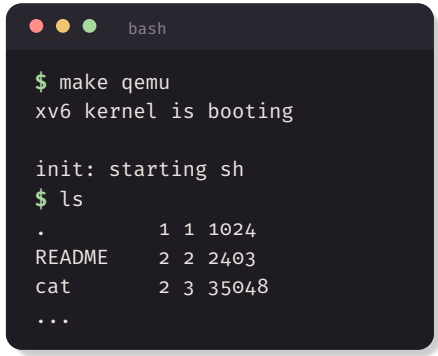
IBM PC XT. Ruben de Rijcke, CC BY 3.0, Wikimedia Commons.



Linus Torvalds. CC BY-SA 3.0, Wikimedia Commons.

xv6: un Unix para estudiar

- Reimplementación moderna de Unix v6, escrita en el MIT para enseñar.
- Corre sobre RISC-V, la arquitectura abierta que se usa hoy en investigación, en chips embebidos y cada vez más en la industria.
- Menos de 7000 líneas de C: se puede leer completo en un semestre.
- Lo ejecutamos en QEMU, dentro del contenedor del curso.

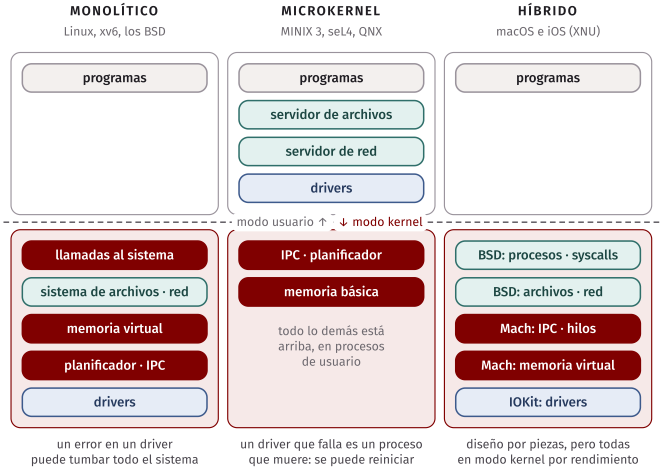


SECCIÓN 04

Cómo se organiza un kernel

04

¿Qué corre en modo kernel?



- **Monolítico**: todo el sistema operativo es un solo programa en modo kernel.
- **Microkernel**: adentro solo lo mínimo; archivos, red y drivers son procesos de usuario que se hablan por mensajes *(IPC)*.
- **Híbrido**: piezas de microkernel, pero todas adentro.

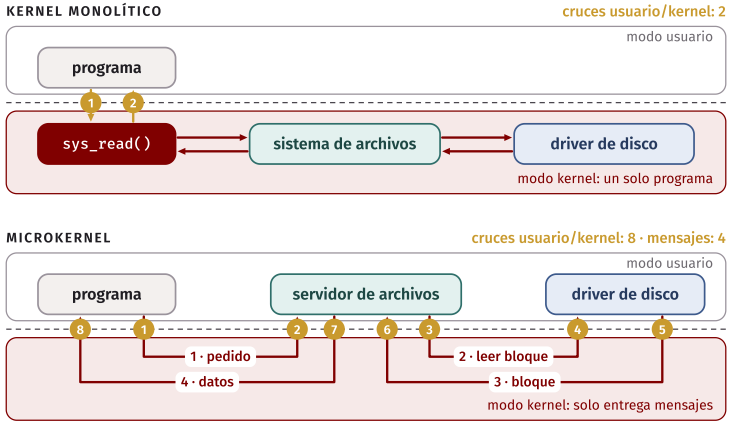
xv6 es monolítico

bash · clon de xv6

```
$ riscv64-linux-gnu-nm kernel/kernel |
    grep -wE 'sys_read|fileread|readi|bread|virtio_disk_rw'
0000000080002b66 T bread
000000008000424c T fileread
000000008000360a T readi
0000000080004d62 T sys_read
00000000800058d8 T virtio_disk_rw
```

- Los 27 `.o` de `OBJS` se enlazan en un solo ejecutable, con todo el camino de un `read()`: de `sys_read` al driver.
- Rápido: del sistema de archivos al driver hay una simple llamada a función.
- Sin paredes: un error en un driver puede tumbar todo el sistema.

Microkernel: el camino de un `read()`



La espera del disco es la misma en los dos casos; aquí solo se cuentan las entradas y salidas del kernel.

Gana aislamiento (MINIX 3) y un kernel verificable (seL4); paga en mensajes.

¿Cuánto cuesta la estructura?

Supuestos redondos: llamada al sistema **0,1 µs**; un mensaje entre procesos **1 µs**. La semana 2 miden los de su máquina.

	Monolítico	Microkernel	Diferencia
Costo de la estructura	0,1 µs	$4 \times 1 = 4$ µs	3,9 µs
Con un SSD (80 µs)	80,1 µs	84 µs	4,9 % más lento
Con un disco duro (5 ms)	5000,1 µs	5004 µs	0,08 % más lento

Lo que importa es el costo del mensaje frente al trabajo útil

En operaciones cortas, 4 µs de mensajes pueden duplicar el tiempo. Mach (años 80) tenía IPC lento y le dio mala fama a la idea; L4 y seL4 lo hicieron mucho más barato.

Términos medios y un debate famoso

- **Módulos de Linux**: drivers `.ko` que se cargan en caliente (`modprobe`). Flexibilidad, no aislamiento.
- **Híbridos**: XNU (macOS) junta Mach, BSD e IOKit en modo kernel; Windows NT fue metiendo partes al kernel.
- **Capas**: THE (Dijkstra, 1968). **Exokernel**: solo reparte el hardware. **Unikernel**: una app + lo justo del OS.

Tanenbaum contra Torvalds (1992)

El autor de MINIX publica *"LINUX is obsolete"*: un kernel monolítico en 1991 es un paso atrás. Torvalds responde que Linux ya funciona.

Treinta años después

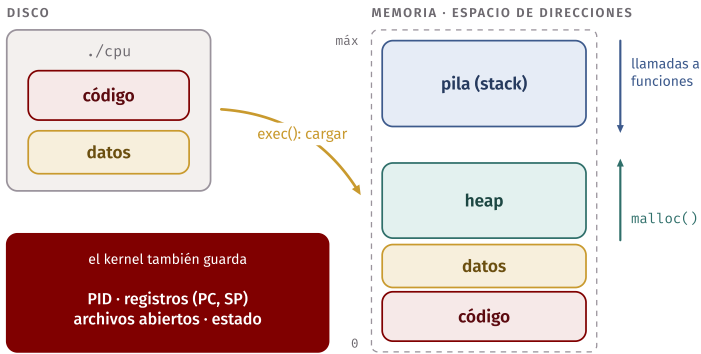
Linux domina servidores y celulares; los microkernels, autos y aviones. Otro compromiso.

SECCIÓN 05

La abstracción de proceso

05

Programa y proceso



Un **proceso** es un programa en ejecución, con todo el estado que lo acompaña.

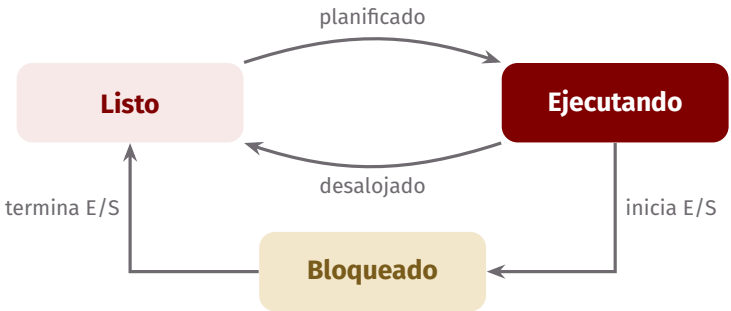
Crear un proceso, paso a paso

1. Cargar el código y los datos estáticos desde el ejecutable. (Los sistemas modernos cargan *perezosamente*: solo lo que se usa.)
2. Reservar la **pila**, con `argc` y `argv` de `main`.
3. Preparar el **heap** para `malloc`.
4. Abrir los descriptores de archivo 0, 1 y 2: entrada, salida y error estándar.
5. Saltar a `main` y ceder la CPU al programa.

API de procesos

Todo sistema operativo ofrece, de alguna forma: **crear**, **destruir**, **esperar**, **controlar** (suspender, reanudar) y **consultar el estado**. La próxima semana: `fork`, `exec` y `wait`.

Estados de un proceso



Listo *(ready)* · Ejecutando *(running)* · Bloqueado *(blocked)*

Traza: dos procesos, uno hace E/S

Tiempo	Proceso 0	Proceso 1	Nota
1	Ejecutando	Listo	
2	Ejecutando	Listo	
3	Ejecutando	Listo	P0 inicia una lectura de disco
4	Bloqueado	Ejecutando	P0 espera; P1 aprovecha la CPU
5	Bloqueado	Ejecutando	
6	Listo	Ejecutando	Termina la E/S de P0
7	Listo	Ejecutando	P1 termina
8	Ejecutando	—	

¿Debía P0 volver a correr apenas terminó su E/S? Lo decide el planificador *(scheduler)*: semanas 3 y 4.

Aprovechar la espera: 11 contra 8



la lectura de disco dura lo mismo; lo que cambia es qué hace la CPU mientras tanto

¿Cómo lo guarda el kernel? `struct proc` en xv6

kernel/proc.h

```
1  enum procstate { UNUSED, USED, SLEEPING,
2                   RUNNABLE, RUNNING, ZOMBIE };
3
4  struct proc {
5    struct spinlock lock;
6    enum procstate state;     // estado
7    int pid;                  // identificador
8    struct proc *parent;      // proceso padre
9    pagetable_t pagetable;    // memoria
10   struct context context;   // registros
11   struct file *ofile[NOFILE]; // archivos
12   char name[16];
13   // ...
14 };
```

- La lista de procesos es un arreglo: `struct proc proc[NPROC]`.
- `SLEEPING` es el estado *bloqueado*; `RUNNABLE`, *listo*.
- `ZOMBIE`: terminó, pero el padre aún no recogió su resultado.
- `context` guarda los registros para el cambio de contexto.

Mecanismo y política

Mecanismo: *cómo*

Cómo se hace algo. Por ejemplo, el cambio de contexto: guardar los registros de un proceso y cargar los de otro.

Política: *qué*

Qué decisión tomar. Por ejemplo, a qué proceso darle la CPU ahora.

Por qué separarlos

El mismo mecanismo sirve para muchas políticas. Se puede cambiar el planificador sin reescribir el cambio de contexto. Este curso alterna entre los dos: *cómo* funciona y *qué* conviene.

Resumen

- El sistema operativo **virtualiza** recursos, los **administra** y ofrece una **interfaz** de llamadas al sistema.
- Tres piezas: virtualización, concurrencia y persistencia.
- Un **proceso** es un programa en ejecución: memoria, registros y recursos abiertos.
- Un proceso está *listo*, *ejecutando* o *bloqueado*; el kernel lo guarda en una estructura como `struct proc`.
- Separar **mecanismo** y **política** es una idea que veremos todo el semestre.

Para la próxima clase

- Leer OSTEP cap. 5: *Process API*.
- Laboratorio: instalar Docker y levantar el contenedor del curso (guía en el sitio).
- Simulador `process-run.py`: preguntas 1–4 del cap. 4.
- Empezar el proyecto **P0**: utilidades Unix.

Pregunta para pensar

Si un proceso hace E/S y se bloquea, ¿qué debería hacer el sistema operativo con la CPU? ¿Y si hay diez procesos listos?