CC4302
Sistemas Operativos
Profesor: Luis Mateu

- Implementación de un núcleo de Unix
- Espacio de direcciones virtuales
- Peers (contrapartes)
- Relación Unix y nThreads
- Núcleo clásico y moderno
- Los spin-locks

Núcleos de Unix

- 1970: Primera versión pública de Unix, release 7, Bell Labs de la ATT
- Comienzos de los 80:
  - Unix System V release 3
  - Unix BSD de la Universidad de Berkeley
- Fines de los 80: Sun OS 4 de Sun Microsystems
- Comienzos de los 90: Linux 1

Son todos núcleos clásicos:

- Solo para monocore
- Interrupciones inhibidas dentro del núcleo
- No hay dataraces

- 1990: 1er. núcleo moderno, Unix System V rel. 4

  multicore, interrupciones en el núcleo, dataces
- Después: Windows NT, Solaris 2 de Sun, Linux 2

Núcleo de Unix

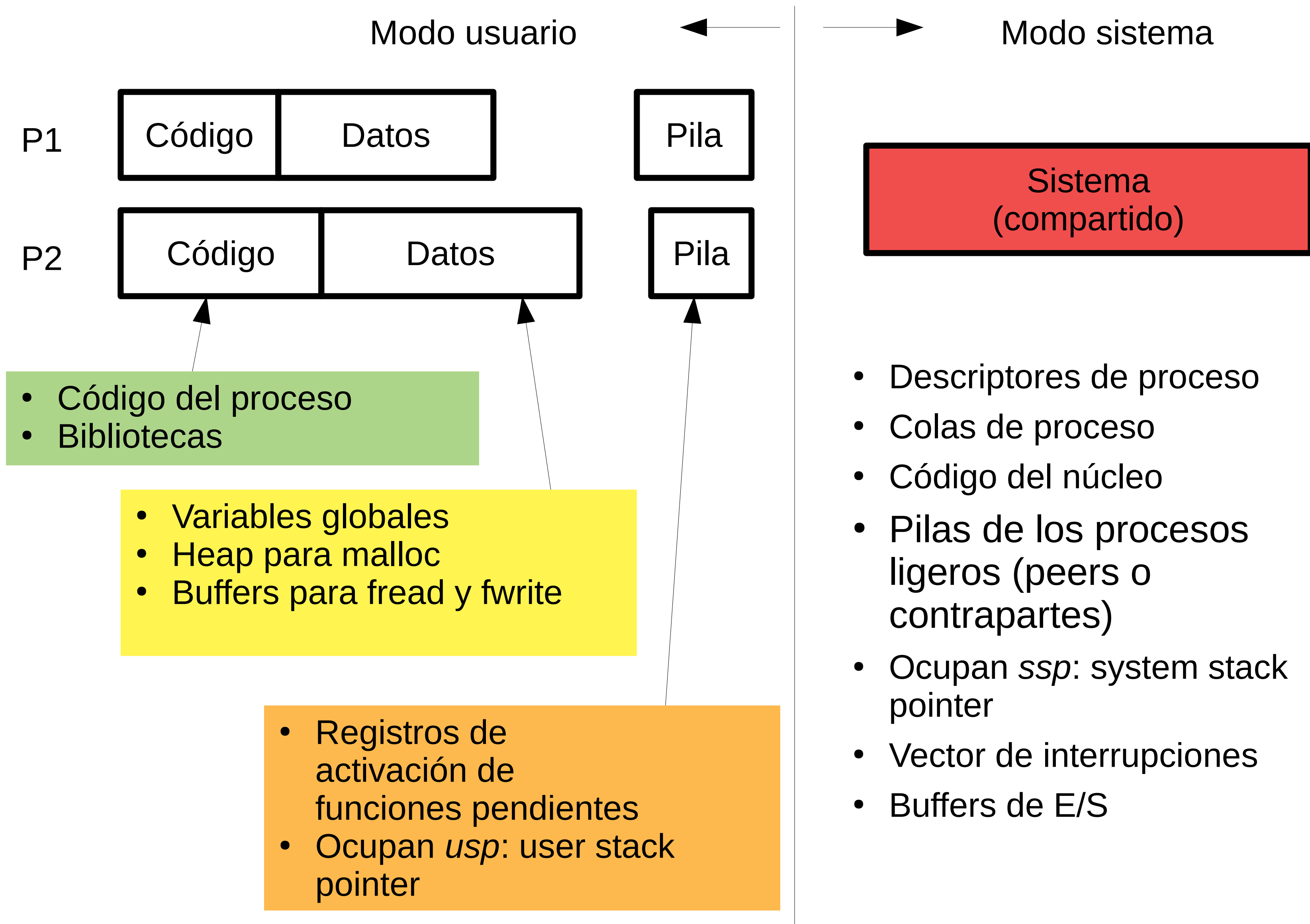
En modo usuario

- Segmento sistema inaccesible
- Ciertas instrucciones priveligiadas deshabilitadas como la que inhibe las interrupciones
- ***La única forma de pasar a modo sistema es por medio de una interrupción***
- Las llamadas a sistema (open, read, fork, exit) se implementan generando una interrupción
- Stack pointer para modo usuario

En modo sistema

- Puede ejecutar las instrucciones privilegiadas
- El código, datos y pila vive en el segmento sistema
- Stack pointer para modo sistema

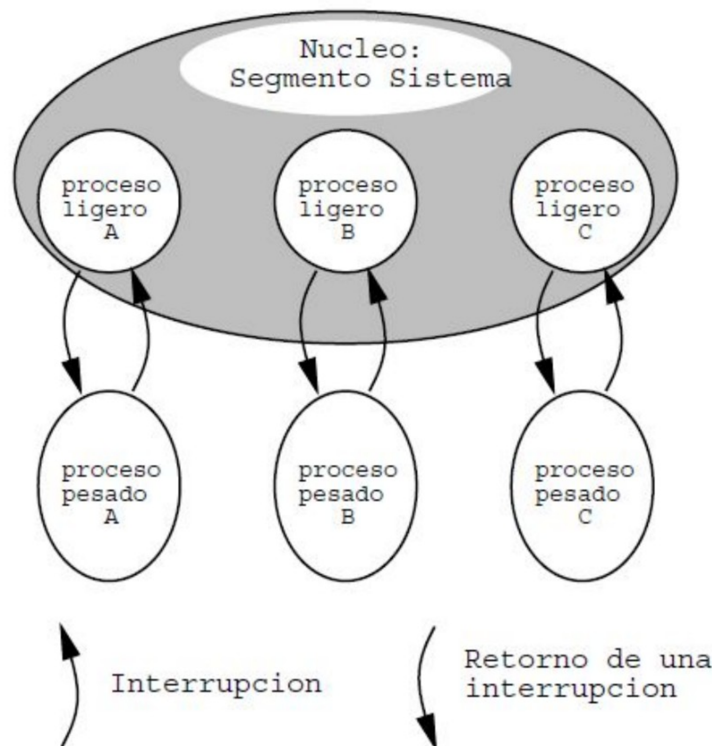
Espacio de direcciones virtuales



(hasta que aparecieron Meltdown y Spectre en 2018)

Procesos ligeros: *peers*

- Un proceso ligero por cada thread del usuario
- Se denomina *peer* o contraparte
- Atiende las interrupciones que ocurran mientras se ejecuta su contraparte en modo usuario



Relación Unix y nThreads

Unix	***nThreads***
Proceso pesado o pthread	***nthread de la aplicación***
Interrupción del timer	Señales SIGALRM y SIGVTALRM
spin-lock	LLMutex
inhibir interrupciones	inhibir señales
instrucción halt	sigsuspend
Espacios de direcciones virtuales	No disponible

Todos los threads de los procesos pesados se convierten en un proceso ligero dentro del núcleo

- En una llamada a sistema (interrupción)
- Una interrupción de un dispositivo

⟶ La implementación de un núcleo de Unix es similar a la de nThreads

Unix en multiprocesadores

- Varios cores físicos, cada uno con su propia memoria cache, pero compartiendo la memoria principal y los dispositivos
- Un proceso pesado es un computador virtual en el que múltiples cores virtuales (los threads) comparten la memoria (el espacio de direcciones virtuales)
- Los *pthreads* aparecen recién en ~ 2005, con la aparición de los primeros chips multicores para computadores personales

Scheduling

- Un cronómetro regresivo físico por core
- Cola de procesos ready compartida por todos los cores
- ¿Y si 2 cores extraen un proceso al mismo tiempo? Datarace
- Exclusión mutua: inhibir las interrupciones no es suficiente

Estrategias de implementación

- Núcleo clásico: Se admite un solo core trabajando dentro del núcleo
  - Ventaja: fácil de implementar en un núcleo clásico
  - Desventaja: no hay paralelismo en el núcleo
- Núcleo moderno de Unix: se garantiza la exclusión mutua a nivel de cada estructura de datos compartida en el núcleo
  - Ventaja: sí hay paralelismo en el núcleo
  - Desventaja: mucho, mucho más complejo
  - Problema: ¿Cómo se implementa la exclusión mutua?
- Solución: spin locks
  - Es un semáforo binario que se implementa con busy-waiting

Implementación incorrecta de spin-locks

```
void spinLock(int *plock) {
  while (*plock==CLOSED)
    ;  // ¡Incorrecta!
  *plock=CLOSED;
}
void spinUnlock(int *plock) {
  *plock= OPEN;
}
```

- Se necesita ayuda del hardware para poder implementar correctamente un spin-lock
- Todo los procesadores poseen una instrucción swap que intercambia ***atómicamente*** el valor almacenado en un registro por el valor almacenado en una dirección de memoria.

La instrucción swap

En assembler: swap [*R1*], *R2*

- R1 contiene la dirección de entero en memoria
- Ese entero puede contener OPEN o CLOSED
- R2 contiene el valor CLOSED
- Swap intercambia los valores almacenados en R2 y el entero en memoria
- Si el entero en memoria contenía OPEN, queda en CLOSED y R2 queda en OPEN
- Considere una función *swap* que hace lo mismo que esta otra función *swap* escrita en C:

```
int swap(int *psl, int val) {
   int ret= *psl;
   *psl= val;
   return ret;
}
```

- Pero está escrita en assembler para usar la instrucción de máquina swap y por lo tanto **es atómica**

Implementación ineficiente de spin-locks

```
void spinLock(int *psl) {
  while (swap(psl, CLOSED)==CLOSED)
    ;
}

void spinUnlock(int *psl) {
  *psl= OPEN;
}
```

- Esta implementación es correcta
- Pero ineficiente porque puede producir exceso de tráfico en el bus de datos que es compartido por todos los procesadores para comunicarse entre sí y para llegar a la memoria
- No hay problema si se pide el spin-lock y está libre
- Tampoco hay problema si un solo procesador pide el spin-lock que está siendo ocupado por otro procesador
- El procesador en espera hará busy-waiting llamando a swap hasta que se invoque *spinUnlock*
- El acceso a la memoria de *swap* ocurrirá en la memoria cache del procesador en espera, sin interferir con la ejecución de otros procesadores