udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA

Kernel

Adín Ramírez
adin.ramirez@mail.udp.cl

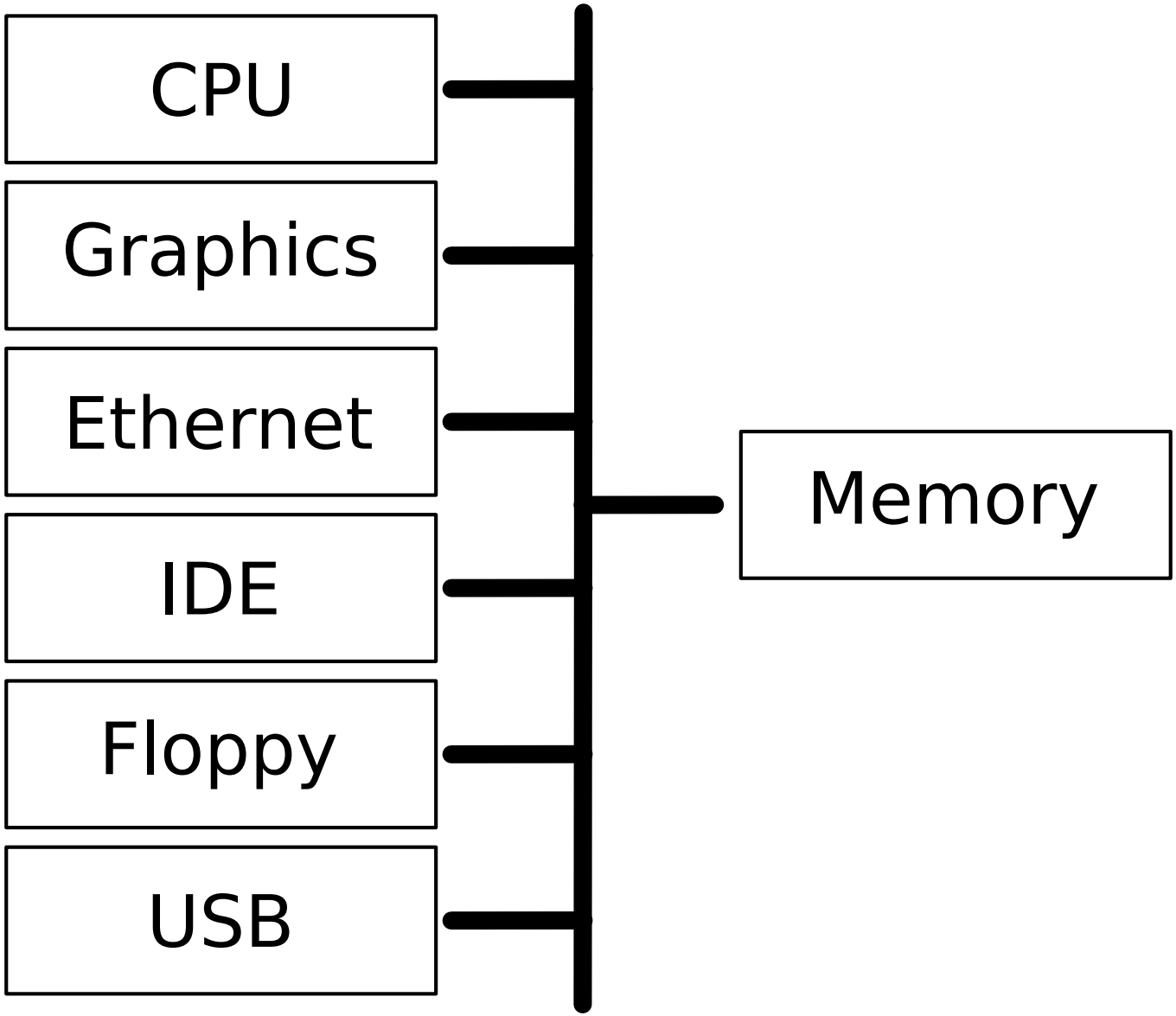
Sistemas Operativos (CIT2003-1)
1er. Semestre 2015

Adaptado de Dave Eckhardt

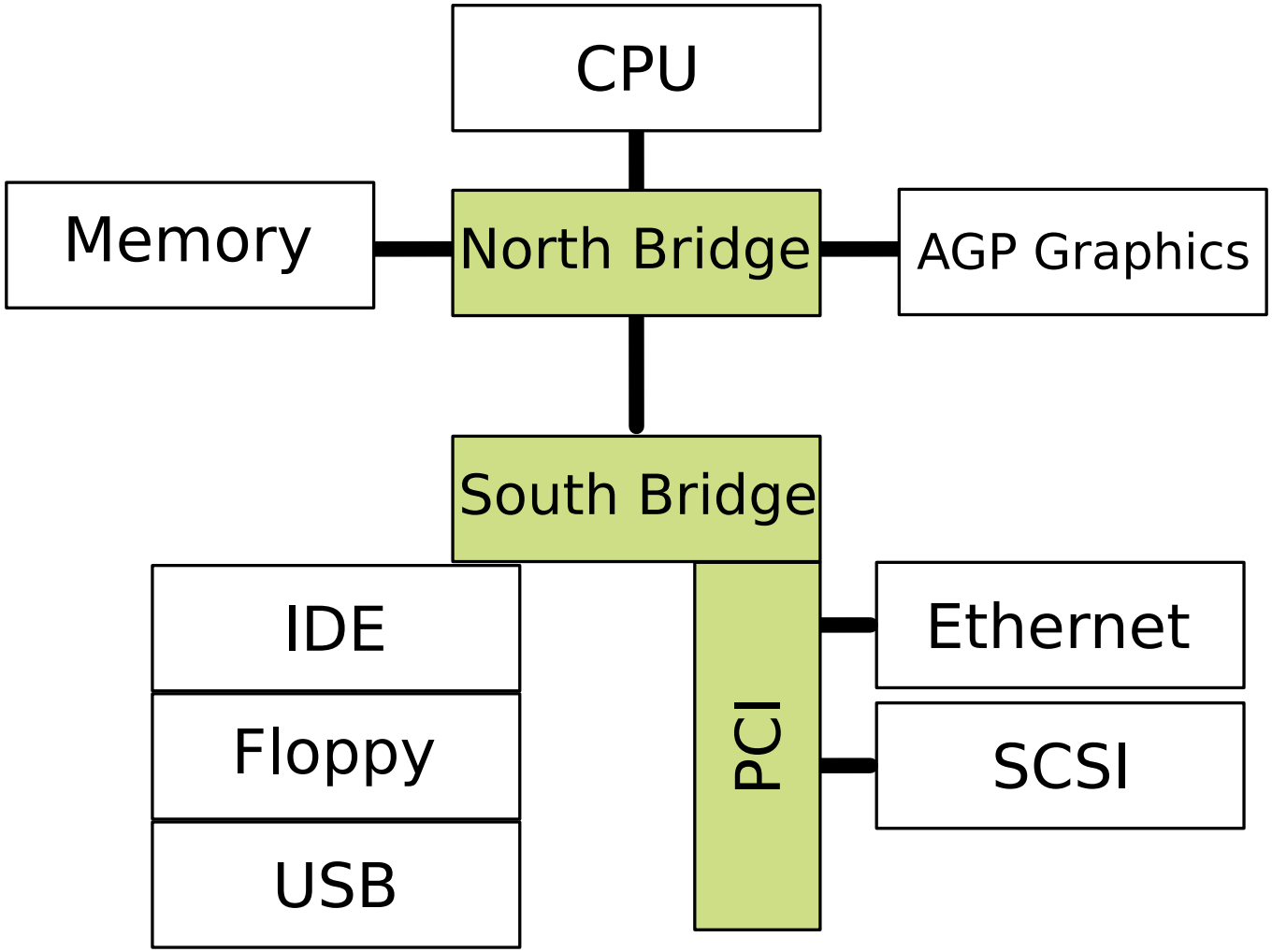
Objetivos de la clase

- Hardware
- Estado del CPU
- Llamadas del sistema
- CPU context switch
- Interrupciones
- Condiciones de carrera
- Enmascarado de interrupciones
- Ejemplos de hardware

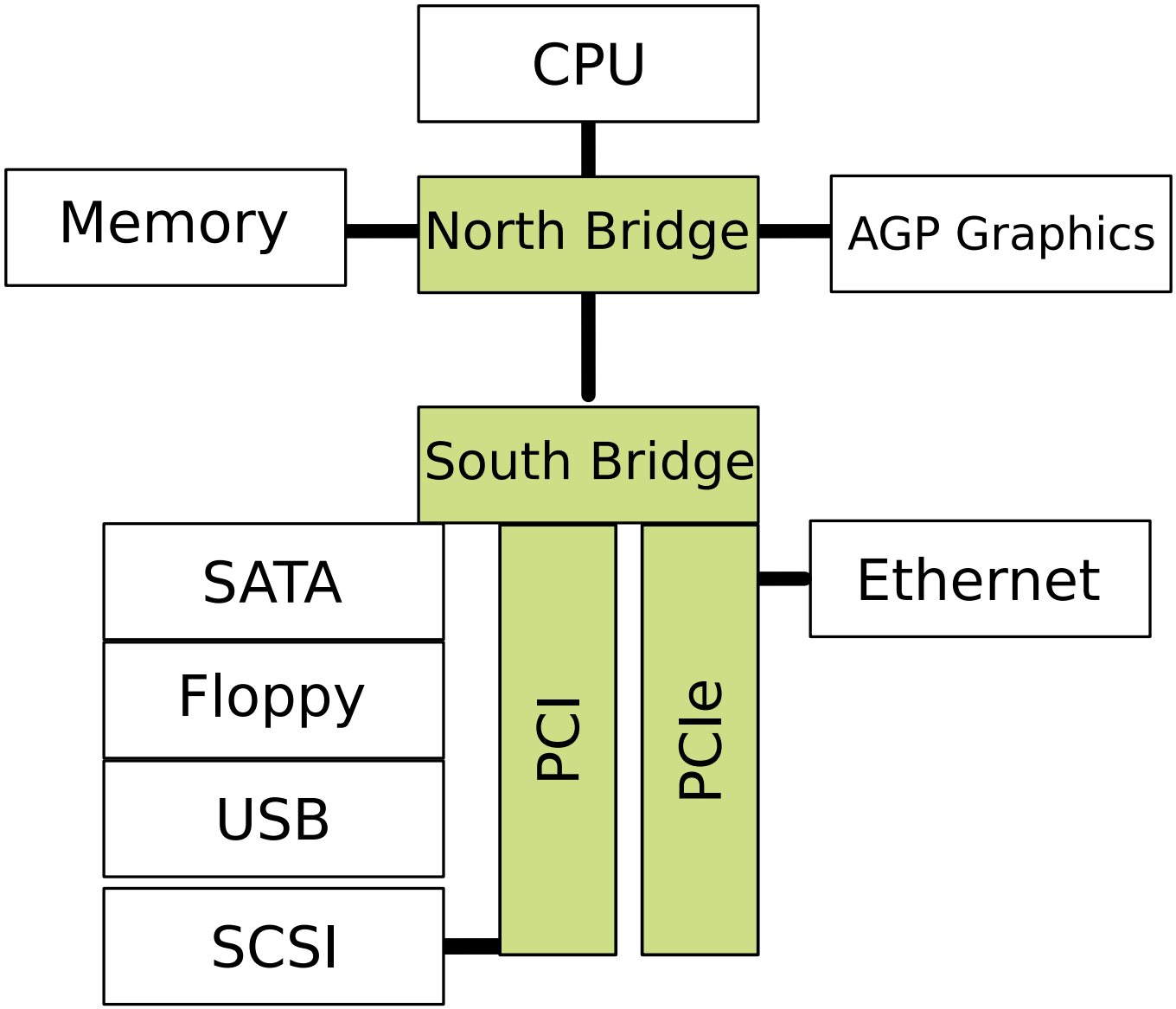
Dentro de la computadora: histórico



Dentro de la computadora: 1997–2004



Dentro de la computadora: 2004–



CPU

- Registros de usario (en la Planet IA32)
  - Propósito general: `%eax`, `%ebx`, `%ecx`, `%edx`
  - Stack pointer: `%esp`
  - Frame (base) pointer: `%ebp`
  - Registros misteriosos de strings: `%esi`, `%edi`

CPU

- Registros que no son del usuario: del estado procesador
  - Modo actual: código de usuario o de kernel
  - Interrupciones: on/off
  - Memoria virtual: on/off
  - Modelo de memoria
    - pequeño, mediano, grande, etc.

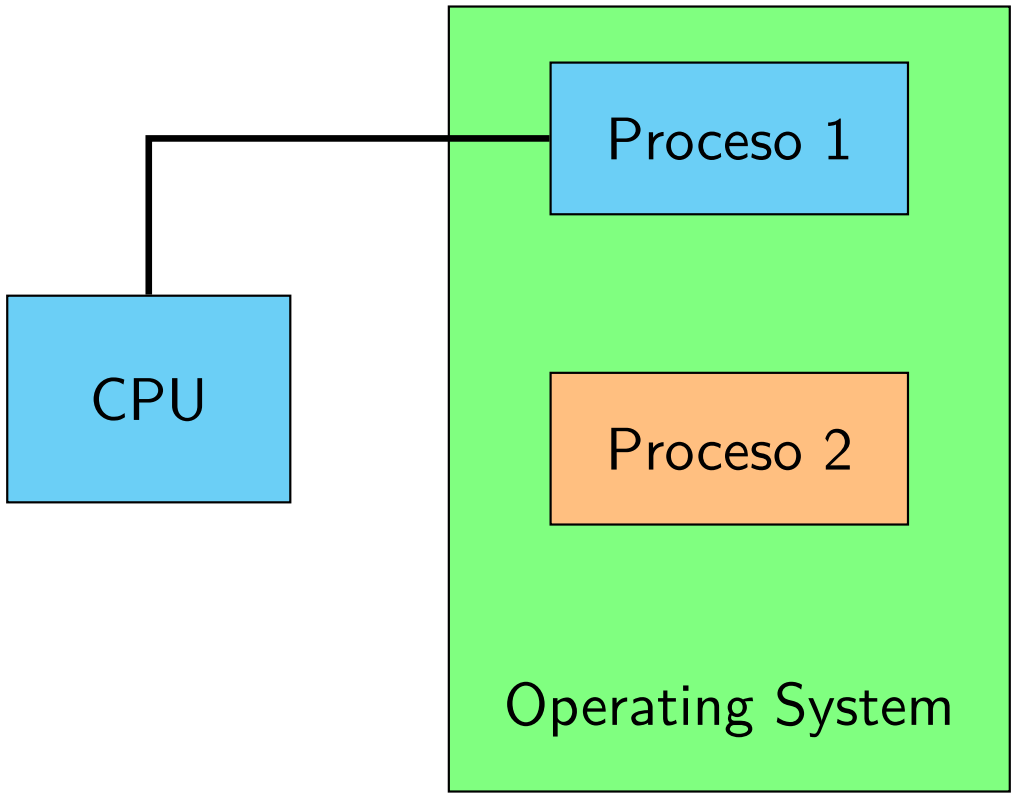
CPU

- Registros de números de punto flotante
  - Parte lógica de los *registros de usuario*
  - A veces, son registros especiales
    - Algunas máquinas no tienen operaciones de punto flotante
    - Algunos procesos no usan punto flotante

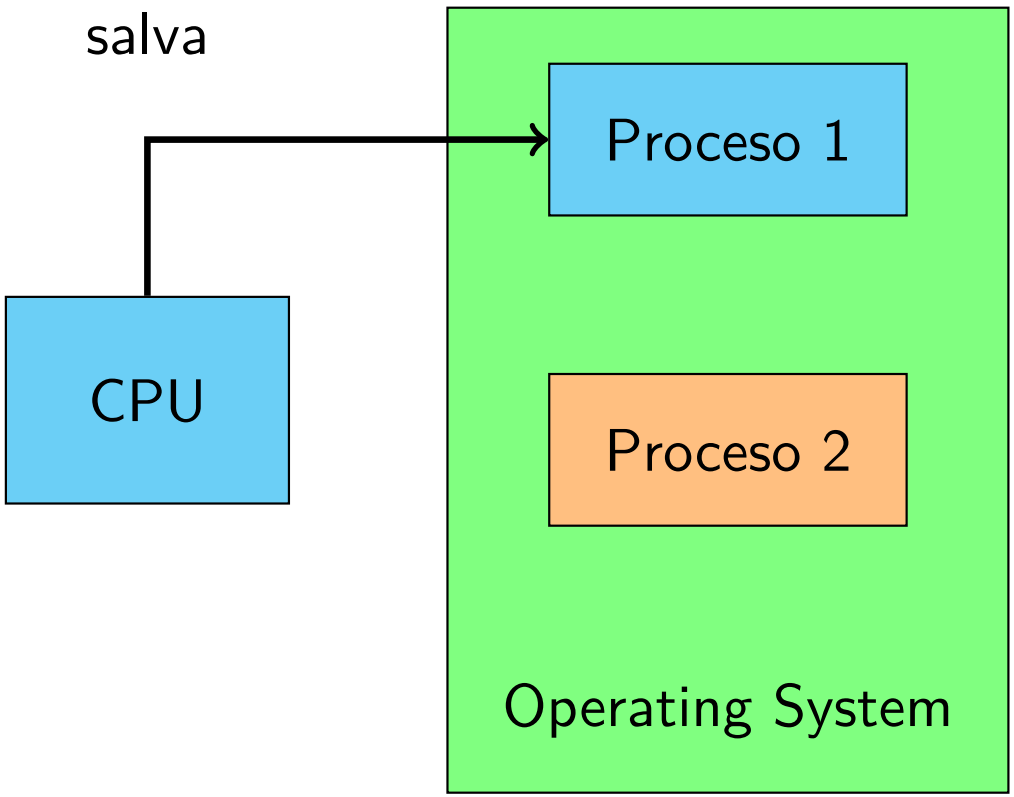
getpid()

- El proceso del usuario está ejecutandose
  - Proceso del usuario ejecuta `getpid()` (de la librería)
  - La librería ejecuta `TRAP $314159`
    - En los Intel, `TRAP` es llamado `INT`, pero no es una interrupción
- El mundo cambia
  - Algunos registros se bajan a memoria en alguna parte
  - Algunos registros se cargan a memoria de alguna parte
- El procesador ha entrado al modo de kernel

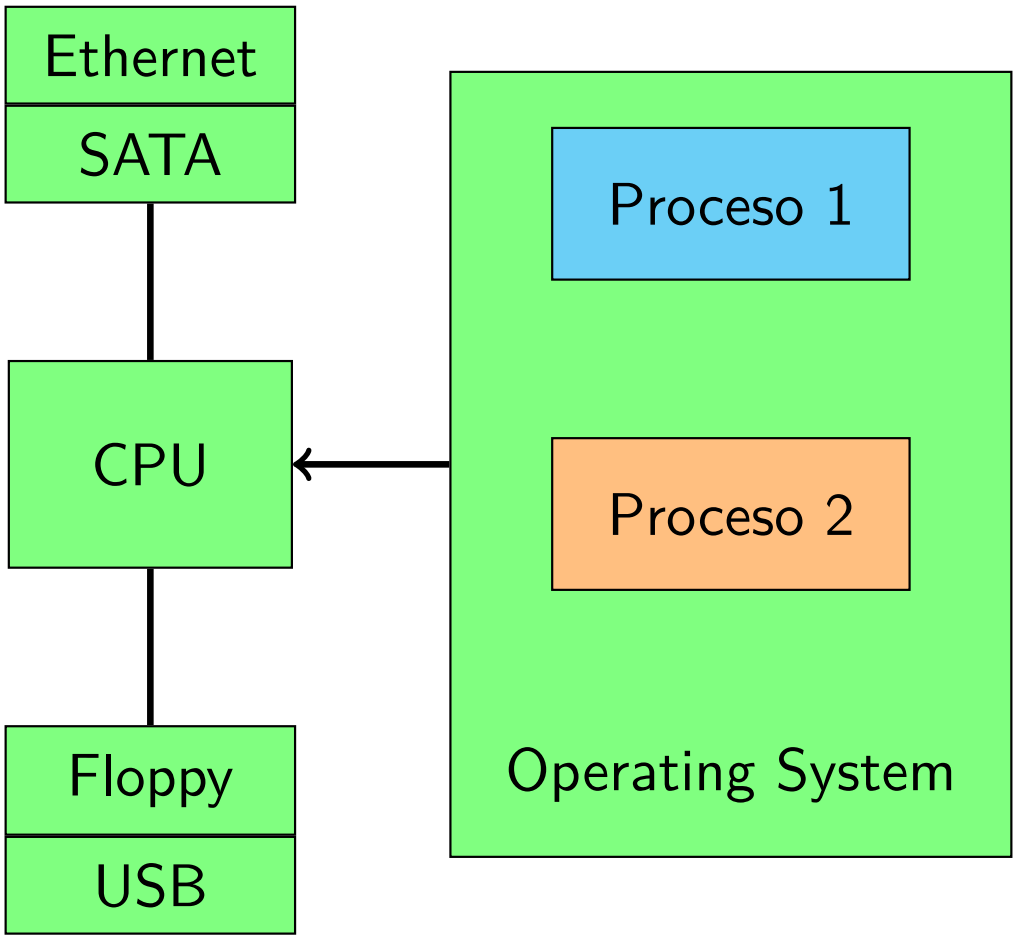
Modo de usuario



Entrando en modo de kernel



Entrando en modo de kernel



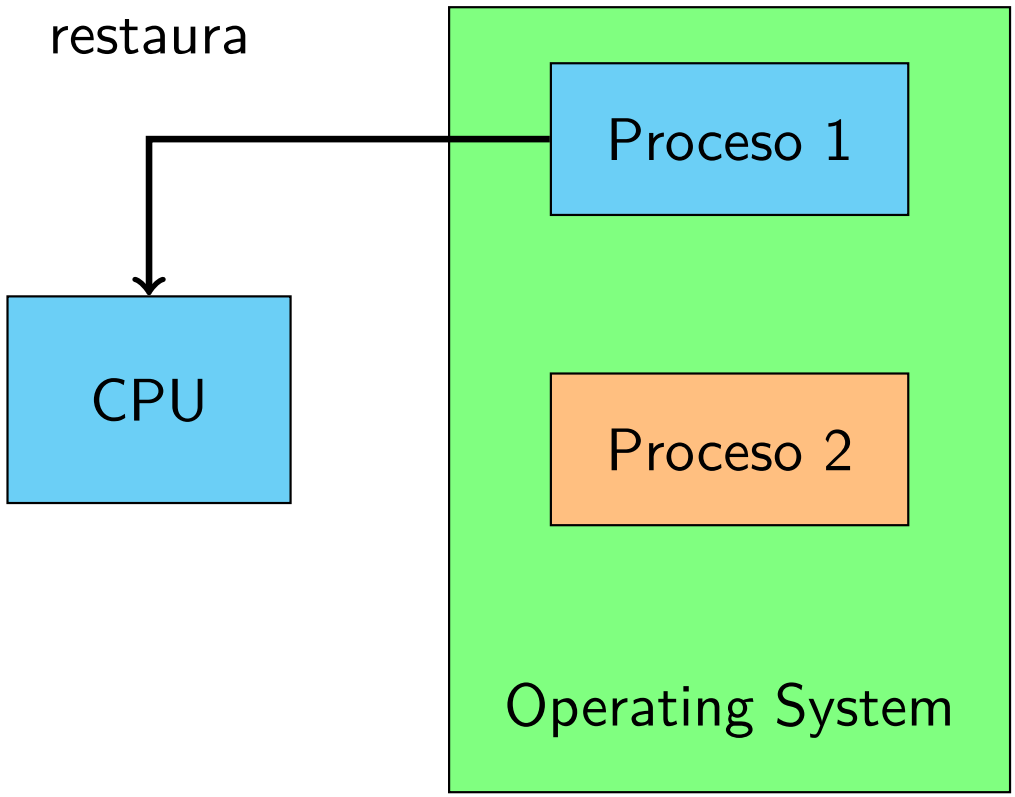
Ambiente de ejecución del kernel

- Los lenguajes de tiempo de ejecución difieren
  - ML: puede no tener stack (solo heap)
  - C: basado en un stack
- El procesador es más o menos agnóstico
  - Algunos asumen o requieren un stack
- El manejador de trap construye el ambiente de ejecución del kernel
  - Dependiendo del procesador
    - Cambia al stack correcto
    - Salva los registros
    - Enciende la memoria virtual
    - Limpia los caches

La historia de `getpid()`

- El proceso se ejecuta en modo de kernel
  - `running->u_reg[R_EAX] = running->u_pid;`
- Retorna de la interrupción
  - El estado del procesador es restaurado a modo de usuario
- El proceso de usuario continua ejecutándose
  - La librería retorna `%eax` cómo el valor de `getpid()`

Regresa a modo de usuario



getpid()

- ¿Qué es la llamada del sistema `getpid()`?
  - Una función de `C` que podemos llamar para obtener el identificado del proceso
  - Una instrucción simple que modifica `%eax`
  - Código privilegiado que puede accesar al estado interno del sistema operativo

read()

- El proceso de usuario está ejecutandose
  - `count = read(7, buf, sizeof(buf));`
- El proceso de usuario se detiene (pausa)
- El sistema operativo solicita una lectura de disco
- El tiempo pasa
- El sistema operativo copia datos para el buffer del usuario
- El proceso del usuario regresa a ejecutarse

`read()`, pero con detalles

P1: `read()`

- Trap para el modo de kernel

Kernel: le dice al disco *lee el sector 2781828*

Kernel: cambia a ejecutar P2

- Regresa a modo de usuario —pero ejecuta P2, no P1
- P1 está bloqueado en una *llamada a sistema*
  - El `%eip` de P1 está en alguna parte del kernel
- Marcado como *no puede ejectuar más instrucciones*

P2: calcula 1/3 del minado de una bitcoin

read(), pero con detalles

Disco: terminé de leer!

- Envía una señal de *petición de interrupción*
- CPU detiene la ejecución de P2
- Interrumpe para modo de kernel
- Ejecuta el código del *manejador de interrupción de disco*

Kernel: cambia la ejecución a P1

- Regresa de la interrupción —a P1, P2 sigue detenido
- P2 puede ejecutar instrucciones, pero no lo hace
  - P2 no se está ejecutando
  - Pero no está bloqueado
  - Su estado es *ejecutable*, a diferencia del P1 antes de que el disco terminara

Tabla de vectores de interrupciones

- ¿Cómo sabe el CPU como manejar cada interrupción?
  - Interrupción de disco ⇒ invocar al driver del disco
  - Interrupción del mouse ⇒ invocar al driver del mouse
- Necesita saber
  - Donde almacenar los registros
    - Frecuentemente, propiedad del proceso actual, no de la interrupción
  - Cargar valores nuevos al CPU
    - Clave: nuevo contador de programa, nuevo registro de estado
    - Estos definen el nuevo ambiente de ejecución

Envío de interrupciones

- Lookup table
  - El controlador de interrupciones dice: esta es la fuente de la interrupción #3
  - CPU va a traer la entrada #3 de la tabla
    - La tabla basada en punteros es construida cuando el SO arranca
    - El tamaño de la tabla se define por HW
- Almacenar el estado del procesador
- Modificar el estado del CPU de acuerdo a la entrada de la tabla
- Iniciar la ejecución del manejador de la interrupción

Retorno de la interrupción

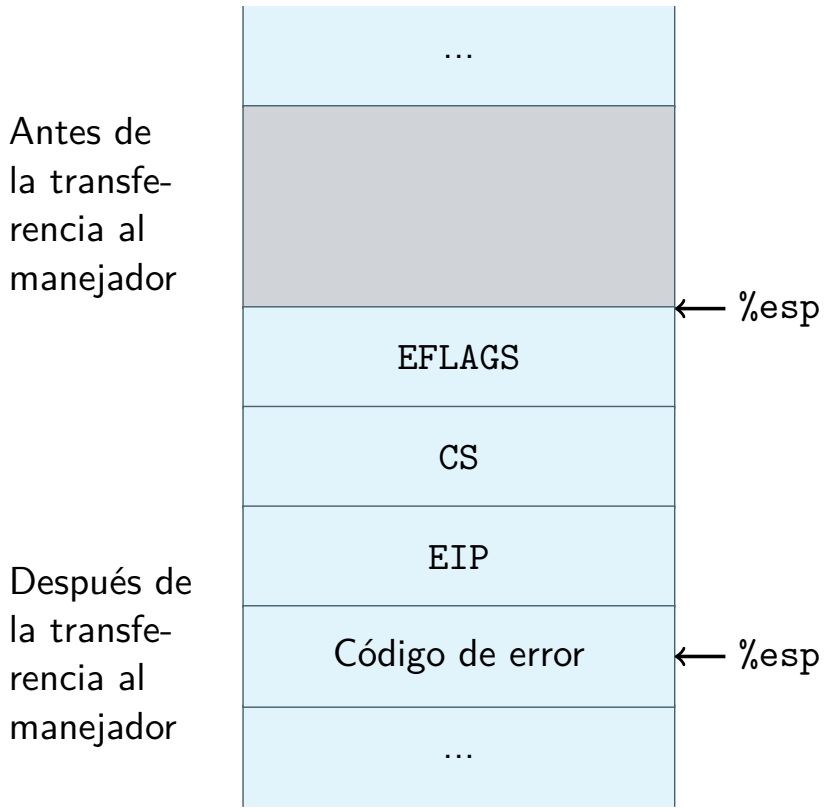
- Operación de retorno de una interrupción
  - Cargar el estado del procesador a los registros
  - Restaurar el contador del programa reactiva el código antiguo
  - Las instrucciones de hardware restauran alguna parte del estado
  - El kernel debe de restaurar el resto

x86/IA32

- El CPU guarda el estado del procesador
  - Almacenado en el stack del kernel
- El CPU modifica el estado de acuerdo a la tabla de entradas
  - Carga información privilegiada, el contador del programa
- Inicia el manejador de la interrupción
  - Usa el stack del kernel para sus operaciones
- Se termina el manejador de la interrupción
  - Vacia el stack a su estado original
  - Invoca el retorno de interrupción (`iret`)
    - Los registros se cargan del stack del kernel
    - El modo puede cambiar de kernel a usuario
    - Puede que el código se mantenga en modo de kernel

IA32 modo de tarea única

- Hardware inserta los registros en el stack actual (no cambia el stack)
  - `EFLAGS` (estado del procesador)
  - `CS`/`EIP` (dirección de retorno)
  - Código de error (algunas interrupciones/errores, no todas: consultar arquitectura)
  - `iret` restaura el estado de `EIP`, `CS`, `EFLAGS`



Condiciones de carrera

- Dos actividades concurrentes
  - Programa de computadora, disco
- Varias secuencias de ejecución producen varias respuestas
  - ¿El disco interrumpe antes o después de la llamada?
- La secuencia de ejecución no está controlada
  - Cualquier resultado es posible aleatoriamente
- El sistema produce respuestas aleatorias
  - Una respuesta u otra gana la carrera

Condiciones de carrera

Driver del disco

- Una parte quiere lanzar peticiones de I/O al disco
  - Si el disco está desocupado, envía la petición
  - Si el disco está ocupado, encola la petición para después
- La acción del manejador de la interrupción depende del estado de la cola
  - Trabajo en cola $\Rightarrow$ enviar la siguiente petición al disco
  - Cola vacia $\Rightarrow$ deja el disco ocioso
- Varios ordenes de ejecución posibles
  - La interrupción de disco antes o después de la prueba *disco está desocupado*
- El sistema produce respuestas aleatorias
- Trabajo en cola, entonces transmitamos la siguiente petición (bien)
- Trabajo en cola, dejemos descansar el disco (*say what!?*)

Condiciones de carrera

Esqueleto del driver

```
dev_start (request) {
  if (device_idle) {
    device_idel = 0;
    send_device (request);
  } else {
    enqueue(request);
  }
}

dev_intr () {
  // finish up previous request code
  if (new_request = head()) {
    send_device(new_request);
  } else {
    device_idle = 1;
  }
}
```

Buen caso

Proceso de usuario	Manejador de la interrupción
`if (device_idle)`	
no, entonces...	
`enqueue(request)`	
	Interrupción
	hacemos trabajo
	`new = 0x80102044;`
	`send_device(new);`
	Retornamos de la interrupción

Mal caso

Proceso de usuario	Manejador de la interrupción
`if (device_idle)`	
no, entonces...	
	Interrupción
	hacemos trabajo
	`new = 0;`
	`device_idle=1;`
	Retornamos de la interrupción
`enqueue(request)`	

¿Qué fallo?

- Se ejecuta el algoritmo
  - Se examina el estado
  - Se lleva a cabo una acción (según el estado)
- El manejador de la interrupción ejecuta su algoritmo
  - Se examina el estado
  - Se lleva a cabo una acción (según el estado)
- Varios posibles estados de término
  - Dependen de cuando se ejecute el código del manejador de la interrupción
- El sistema produce varias salidas "aleatorias"
  - Estudien la condición, y eviten este problema en sus proyectos

¿Qué podemos hacer?

- Dos soluciones
  - Temporalmente suspender/enmascarar/diferir las interrupciones de los dispositivos mientras se verifica y encola
  - Utilizar una estructura de datos que sea libre de bloqueo
- Considerar
  - Evitar bloquear todas las interrupciones
  - Evtar bloquear por mucho tiempo

Temporizador (timer)

- Comportamiento sencillo
  - Cuenta algo: ciclos de CPU, ciclos de bus, microsegundos
  - Cuando se llega al límite, emite una interrupción
  - Reestablece el contador al valor inicial
    - Lo hace a nivel de HW, en el fondo, no necesita esperar al SW para reestablecerse
- Resumen
  - No hay peticiones, no hay resultados
  - Un flujo constante de eventos distribuidos equitativamente en el tiempo

¿Por qué necesitamos un timer?

- ¿Por qué detener una perfecta ejecución?
- Evitamos que las aplicaciones acaparen el procesador
  `while(1) continue;`
- Mantenemos la hora del día
  - Calendario respaldado por batería cuenta solo los segundos (no muy correctamente)
- Interrupción de doble propósito
  - Mantiene el tiempo: `++ticks_since_boot;`
  - Evita acaparación del CPU: forza el cambio de procesos

Puntos importantes

- Una abstracción del hardware (detalles en Arquitectura de Computadores)
- Modos de ejecución: kernel y de usuario
  - Memoria virtual
  - Código privilegiado
- Ejemplos de llamadas a sistema
- Interrupciones
  - Trampas: síncronas
    - Excepciones: errores en código, programas malignos, o benignos (debugger)
    - Llamadas al sistema
  - Interrupciones: asíncronas
    - Interrupciones por tiempo
    - Dispositivos
- Condiciones de carrera
- Enmascarado de las interrupciones