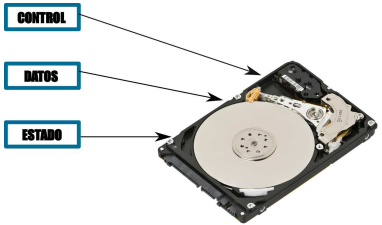
Entrada/Salida, Interrupciones, Protección

Arquitectura del Computador

Entrada Salida

- A nivel de controlador la entrada salida se realiza a través de registros.
- En general los dispositivos tiene varias clases de registros, de control, de estado y de datos.



- Un caso de uso típico para escribir un datos es por ejemplo:
  - Escribir el dato en el resgistro de datos del dispositivo.
  - Enviar el comando de “escribir” en el registro de control.
  - Esperar que el control de estado indique que se ha concluido la escritura.

Entrada Salida

Detrás de la cortina...

- Hemos utilizado el SO para realizar entrada salida (`printf`, `scanf`), escribir a disco, abrir conexiones de red, etc.
- El SO abstrae los dispositivos de hardware para mostrar una visión simplificada y unificada.
- Pero cómo realiza el SO la entrada salida?

A través de la historia distintas arquitecturas lo han solucionado de distintas formas:

- Instrucciones especiales.
- BIOS
- Espacio de direcciones de Puertos.
- Dispositivos mapeados en memoria.

Basic Input Output System

- El BIOS es un firmware en ROM que viene en la placa madre de la PC.
- Incluye por ejemplo el código para "bootear" la máquina e inicializar todos los dispositivos.
- Ofrece el programador varios servicios para realizar I/O (leer del disco, escribir en pantalla).
- Desventaja: Es lento, no son "re-entrantes" y deben ser accedidos en modo real.

Espacio de direcciones de Puertos

- Aquí la idea es tener OTRO Espacio de direcciones, distinto al de las direcciones de memoria.
- Dentro de ese nuevo espacio cada dispositivo tiene un rango asignado donde puede ubicar sus registros.
- La arquitectura incluye dos instrucciones, una para leer `in` y otra para escribir `out`.
  Por ejemplo para leer un byte `inb $0x60, %al`. Notar acá que esta instrucción no accede a la memoria sino que está utilizando el espacio de direcciones de puertos.
- Es más flexible que la BIOS.
- Desventaja: Tener que utilizar instrucciones especiales.
- Esta forma de realizar I/O es una de las que utiliza x86_64.

Dispositivos mapeados en memoria

- En este caso los registros de los dispositivos son “replicados” en la memoria principal en alguna dirección conocida.
- De esta forma leer/escribir en un registro es sólo leer/escribir en la memoria. Por ejemplo `movb $'a', 0xb800`
- Esta es otra forma de realizar I/O que utiliza x86_64.
- Ventaja: No necesitamos instrucciones especiales ni otro espacio de direcciones.
- Desventaja: Ocupamos ciertas direcciones de memoria. Problemas con cache.

Interrupciones

Supongamos que queremos leer un sector del disco:

```
read:
    movq $0x20, %al     # 0x20 = leer un sector
    outb %al, 0x1f0     # 0x1f0 es el registro de control
 w: inb 0x1f1, %al      # 0x1f1 indica cuando la lect. se completa
    cmpb $0, %al        # si es cero leer de nuevo
    je w
    ret
```

Problema?

¿Qué problemas tiene esto en un contexto multi-programado?

Interrupciones

- Las Interrupciones surgen para poder realizar I/O asíncronamente, esto es, el hardware avisa cuando ha ocurrido un evento, por ejemplo se completó la lectura del bloque.
- De esta forma en un caso como el anterior los sistemas multi-programados pueden detener el proceso que quiere leer el disco, dar el procesador a otro proceso, y luego cuando el hardware notifique que se completó la lectura continuar con el proceso detenido.
- A nivel de hardware las interrupciones siempre detienen la ejecución del proceso y comienzan a ejecutar un código especial llamado “manejador de interrupción”. Cada dispositivo tiene asociado un manejador.

Excepciones

- Las excepciones son similares a las Interrupciones sólo que su causa son generados por algún problema en el programa o procesador.
- Las posibles causas son:
  - División por cero.
  - Error de direccionamiento (Segmentation Fault).
  - Instrucción ilegal.
- Al igual que las interrupciones las excepciones detienen la ejecución del proceso y comienzan a ejecutar el “manejador de excepción" correspondiente.
- En Linux las excepciones en general terminan el proceso en ejecución (si no es recuperable).

Protección y multi-programación

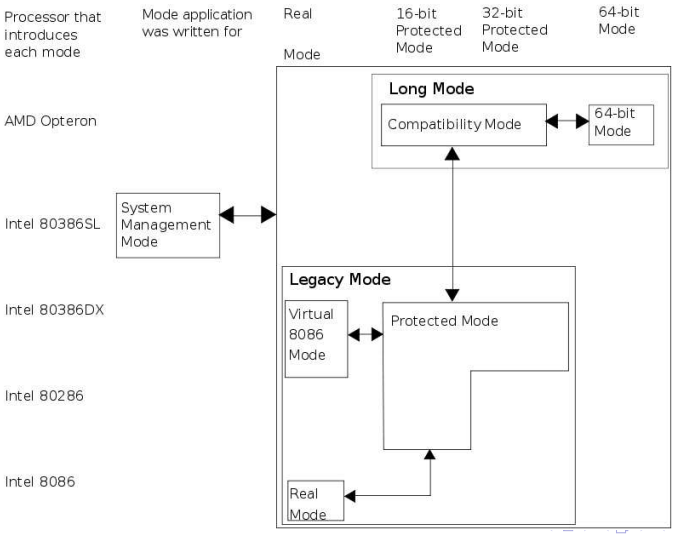
- Al tener más de un programa corriendo a la vez se debe cuidar que los programas no interfieran entre sí.
- Un programa maligno o erróneo podría:
  - Acceder y modificar los datos de otro programa $\Longrightarrow$ Memoria Virtual.
  - Realizando I/O borrar datos del disco de otro usuario, cambiar tablas de paginación, etc.
- Para resolver esto la arquitectura x86_64 incluye distintos modos de ejecución.

Modos de ejecución

Modos de ejecución

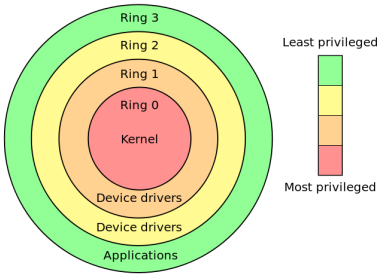
- Modo Real (Sólo por compatibilidad): La PC está en este modo luego de iniciarse. No hay paginación, sólo segmentación. Direcciones de 20 bits y datos de 16 bits. I/O sin protección.
- Modo Protegido (Bit PE en CR0): Incluye paginación, direcciones y datos de 32 bits. Protección para I/O e instrucciones privilegiadas.
- Modo Largo: Simil modo protegido pero direcciones y datos de 64 bits.

Modos de ejecución



Protección y multi-programación

- En modo protegido el procesador puede correr en distintos anillos (0, 1, 2, 3) siendo 0 el más privilegiado y 3 el menos.
- Las instrucciones privilegiadas y de I/O en el anillo 3 generan una excepción (será atendida por el SO).
- El SO ejecuta sobre el anillo 0 llamado “modo kernel”.
- Las aplicaciones (navegador, gcc, etc) ejecutan sobre el anillo 3 llamado “modo usuario”.



Entrada y salida en modo Protegido

- Como vimos un programa corriendo en modo usuario no puede ejecutar instrucciones de I/O.
- Pero `printf` imprime en pantalla, cómo lo hace?
- Lo hace a través del SO invocando una `syscall`.

```
movl $1, %eax # syscall deseada
syscall
```

- La instrucción `syscall` cambia de modo usuario a modo kernel.
- Allí el SO, corriendo en el anillo 0 realiza las operaciones de I/O.