CC4302
Sistemas Operativos
Profesor: Luis Mateu

Unidad 4: Entrada/Salida

- Motivación: interactuar con dispositivos como un botón, un led, un teclado y un disco
- Entrada/salida mapeada en memoria
- Interrupciones
- Dispositivos por caracteres y por bloques
- Canales de entrada/salida
- Controlador de pantalla (GPU)

Motivación

- Determinar si un botón está siendo presionado
- Encender o apagar un led (light emitter diode)

- Leer un teclado

- Leer/escribir en un disco

- Escribir en una pantalla

Entrada/salida mapeada en memoria

- Sería demasiado caro agregar instrucciones de máquina para interactuar con cada tipo de dispositivo
- Es más económico interactuar con un dispositivo como si fuese memoria mapeada en algunas direcciones atribuidas a ese dispositivo
- Por ejemplo para encender un led el driver escribe un 1 en la dirección 0xe0000000 y para apagarlo se escribe un 0 en la misma dirección
- Para determinar si un botón está presionado el driver lee la dirección 0xe0000000, si el primer bit está en 1 quiere decir que está presionado y si está en 0, no lo está
- A esto se llama *entrada/salida mapeada en memoria*
- La dirección 0xe0000000 se denomina *puerto de entrada/salida*
- Ventaja: no se necesitan instrucciones de máquina especiales para interactuar con los dispositivos

Uso del led/botón

- Se crea el dispositivo /dev/led-boton
- El usuario abre el dispositivo con:

```
int fdLB = open("/dev/led-boton", O_RDWR);
```

- Enciende o apaga el led escribiendo un 1 o 0:

```
char c= 1; // 1 para enceder, 0 para apagar
write(fdLB, &c, sizeof(c)); // sizeof(c)==1 siempre
```

- Leer el estado del botón:

```
read(fdLB, &c, sizeof(c));
if (c)
    printf("encendido\n");
else
    printf("apagado\n");
```

El driver para el led/botón

- Función que escribe (led_boton_write):

```
char *port= (char*)(intptr_t)0xe0000000;
char ledVal;
copy_from_user(&ledVal, buf, 1);
*port= ledVal;
…
```

- Función que lee (led_boton_read):

```
char *port= (char*)(intptr_t)0xe0000000;
char botonVal= *port & 1; // borra bits 7 a 1
copy_to_user(buf, &botonVal, 1);
…
return 1;
```

Leer un teclado

- Muchos dispositivos como un teclado se comunican con el computador por medio de una puerta serial (como USB)

- Cada vez que el usuario presiona una tecla, el teclado envía el código de la tecla en un byte por el cable serial, el que se almacena en un *buffer* en la puerta serial
- La puerta serial se mapea en memoria en un puerto de datos y un puerto de control
- El puerto de control sirve para determinar si el *buffer* tiene datos esperando ser leídos
- El puerto de datos sirve para recibir los datos almacenados en el *buffer*

Uso del teclado

- El teclado está asociado al dispositivo /dev/teclado
- Este programa lee el teclado y lo escribe en la salida estándar:

```
int fdTecl = open("/dev/teclado", O_RDONLY);
for (;;) {
    char buf[80];
    int rc= read(fdTecl, buf, 80); // rc es siempre 1
    write(1, buf, rc);
}
```

El driver del teclado

- En el driver, un thread consulta continuamente la puerta de control (0xe0000001) para determinar si llegó un byte del teclado, lo lee de la puerta de datos (0xe0000002) y lo deposita en un buffer compartido con la función de lectura del driver:

```
char *port_ctrl= (char*)(intptr_t)0xe0000001;
char *port_data= (char*)(intptr_t)0xe0000002;
for (;;) {  // busy-waiting!
   if (*port_ctrl & 1) {
      put(buf, *port_data);
      cnt++;
} }
```

- En la función de lectura del driver (teclado_read):

```
char tecladoVal= get(buf); // espera si no hay nada
// errata: en la clase la variable usrbuf se llamaba buf,
// igual que el buffer buf compartido con el thread
// ¡pero son variables distintas!
copy_to_user(usrbuf, &tecladoVal, 1);
cnt--;
…
return 1;
```

La función teclado_read completa (pero sin manejo de errores)

```
static  int  cnt= 0;

static  ssize_t teclado_read(struct file *filp, char *usrbuf,
                                        size_t count, loff_t *f_pos) {
      char tecladoVal= get(buf); // espera si no hay nada
      copy_to_user(usrbuf, &tecladoVal, 1);
      cnt--;
      return 1;
}
```

Dispositivos de salida como una impresora

- El caso de una impresora es similar al de un teclado, solo que se envían datos a la impresora en vez de recibirlos
- No hay que esperar a que lleguen los datos
- El problema es que el buffer de la puerta de comunicaciones es de tamaño limitado y se llena con facilidad
- La puerta de control indica cuando el buffer está lleno
- En el driver, un thread consulta periódicamente si hay datos que enviar y si el buffer no está lleno
- Si está lleno, espera
- Además, la impresora podría no ser suficientemente rápida para imprimir lo que le envían
- La impresora también posee un buffer que se puede llenar
- El driver debe consultar cuanto espacio queda en el buffer de la impresora antes de enviar nuevos datos
- Por lo tanto también se reciben datos de la impresora

Interrupciones

- El problema del driver anteriores es que el ciclo de busy-waiting consume el 100% de un core
- Para evitar esto, se configura la puerta serial para que gatille una interrupción cada vez que reciba un byte
- Ya no se necesita un thread para recibir los bytes, pero sí se requiere una rutina de atención de interrupciones que se invoca cada vez que la puerta a serial recibe un byte desde el teclado
- La rutina de interrupción ejecutará:

```
char *port_ctrl= (char*)(intptr_t)0xe0000001;
char *port_data= (char*)(intptr_t)0xe0000002;
while (*port_ctrl & 1   &&   cnt<bufsize) {
        put(buf, *port_data);
        cnt++;
}
```

- Las señales de PSS son la versión virtualizada de las interrupciones

El vector de interrupciones

- El vector de interrupciones es un arreglo de punteros a las rutinas de atención de interrupciones para cada fuente de interrupción
- Cuando se gatilla una interrupción, se determina un entero que identifica la fuente de la interrupción y se usa como índice en el vector de interrupciones para seleccionar la rutina de atención que se necesita invocar
- Luego se elige cualquier core que no tenga las interrupciones inhibidas, se suspende el proceso o thread que está ejecutando y se invoca la rutina de atención seleccionada en ese core
- Cuando la rutina de interrupción termina, se retoma el proceso o thread de manera transparente
- Si no hay ningún core con las interrupciones habilitadas se espera hasta que se habiliten en algún core
- La rutina de atención podría activar un proceso en estado de espera, en cuyo caso el scheduler podría elegir ceder el core a ese proceso

Dispositivos de acceso por caracteres y por bloques

- Los dispositivos botón, led, teclado, mouse y otros son *character devices*: se leen y escriben de a un solo carácter
- Un disco es un *block* device: se lee y escribe por bloques de 512 bytes (o 4096 bytes en el caso de discos de al menos 2 TB)
- Cuando el disco gatilla una interrupción, quiere decir que hay *n* * 512 bytes listos para ser leídos o escritos
- Un disco nunca se lee o escribe de a un solo byte, siempre en múltiplos de 512 bytes (o 4096)
- El resto es bastante parecido a un dispositivo de caracteres: también se mapea en memoria, tiene puertos de datos y de control
- Se podría decir que el caracter de un disco corresponde a 512 bytes

Canales de acceso directo a memoria

- En dispositivos veloces como un disco o la red, se puede consumir bastante tiempo de CPU leyendo o escribiendo datos desde o hacia el controlador de comunicaciones, porque leer un puerto es considerablemente más lento que leer la memoria
- Para evitar este consumo de CPU los procesadores modernos usan canales de acceso directo a memoria (*direct access memory* o *DMA*)
- Un canal DMA es un circuito especializado en transferir datos entre controlador de comunicaciones (como USB o SATA) y la memoria
- Cuando se van a leer *n* bloques de datos del disco, se configura el canal para que haga la transferencia y el canal se encarga de transferir los datos que vienen por la línea de comunicación directamente a la memoria
- Cuando la transferencia termina, se gatilla la interrupción pero los datos ya están en memoria
- Solo falta activar el proceso que esperaba esos datos

Controlador de pantalla (GPU)

- Una pantalla tiene su propia memoria, llamada *frame buffer*, para almacenar la imagen de la pantalla
- El frame buffer almacena cada pixel de la pantalla en al menos 3 bytes: uno describe la intensidad del verde, otro el rojo y otro el azul
- Las pantallas suelen ser de 1920x1080 (full hd) hasta 3840x2160 pixeles (4K o ultra hd)
- El driver de la pantalla es un proceso que tiene acceso al *frame buffer*, como si fuese memoria normal, solo que escribir en él significa modificar el contenido de la pantalla
- En Unix el driver se llama servidor de X-Windows y no es parte del núcleo
- El controlador de la pantalla, lee la imagen almacenada en el frame buffer 60 veces por segundo y lo envía a la pantalla LCD a través de un cable digital HDMI o DisplayPort
- En notebooks y celulares, el controlador de pantalla es una GPU (graphics processing unit) especializada en operaciones gráfica y un porcentaje de la memoria se destina a frame buffer
- En los computadores para *gamers* la GPU posee su propia memoria para el frame buffer y puede ser de varios GB