28/3 - Señales

Una señal es una notifiación que se envía de forma puntual y unidireccional. Se utiliza para informar sobre eventos asincrónicos, ya que como las señales llegan de otros procesos no tenemos control de cuándo ocurrirán.

Algo similar ocurre con las interrupciones del kernel, como las de Entrada y Salida. Cuando ocurren el kernel guarda el contexto del proceso al momento de ser interrumpido y se pone a manejar él mismo la interrupción.

Las señales, en cambio, se manejan a nivel proceso. Se identifican con un número entero. `signal.h` tiene la lista. Con kill -l se ve la lista de señales.

Handlers

Un handler es una función que adminsitra una señal. Desde que se emite una señal ocurren tres procesos: *lanzamiento*, *notificación* y *administracioń*, que no ocurren en secuencia inmediata. Algunos detalles en orden:

1. La ejecución del proceso es interrumpido por una señal de otro.
2. El proceso continua (se pasa de modo kernel a usuario).
3. Se ejecuta el handler.
4. El proceso prosigue ejecutándose desde donde había sido interrumpido. Si había sido interrumpido en un `sleep`, ese sleep se da por terminado. Si estaba en espera se ejecuta cuando se pasa a running. Si no, en el momento.

El kernel y los procesos de root pueden enviar señales a cualquier proceso. Los otros solo a los de mismo *uid real*.

Opciones ante una señal

- Ignorar/bloquear.
- Atrapar y manejar. Se puede definir para un proceso el handler para cada señal.
- Ejecutar acción default.

Origenes

- Errores como div 0
- Petición ctrl-z o ctrl-c. SIGINT (c) tiene como handler default terminal el proceso.
- Fin proceso hijo.

- Fin timer.
- Kill entre procesos o Raise en el mismo proceso. Envían cualquier señal (el nombre puede ser engañoso).

`int kill (pid_t pid, int sig)` es la generalización de `raise` para permitir mandar señales entre procesos.

Señal Básica

```
typedef void ( * sighandler_t)(int);
sighandler_t signal (int signum, sighandler_t handler);
```

`sighandler_t` es un puntero a función void que toma un int como parámetro. Justamente eso es un puntero al handler, que no puede devolver nada y recibe el número de señal.

Esta función lo que hace es cambiar la disposición de la señal signum al handler especificado.

Bloqueo de señales.

Para cualquier proceso puede bloquearse el manejo de una señal: cuando llegue será anotada como pendiente. El kernel mantiene por proceso dos máscaras de bits:

- Señales Bloqueadas (*Signal Mask*).
- Señales Pendientes. Si enra una señal bloqueada se activará en esta máscara.

Podemos cambiar las señales bloqueadas con:

```
int sigprocmask (int how, const sigset_t * set, sigset_t * oldset);
```

- how: La acción a tomar: SIG_BLOCK, SIG_UNBLOCK o SIG_SETMASK.
- set: sobre qué señales debe tomarse la acción.
- oldset: "devuelve" el viejo valor de la máscara, para analizar algo, deshacer el cambio, etc. Es opcional: si no se usa, se le pone `NULL`.

Señales *inbloqueables*:

- SIGKILL. Produce la inmediata finalización del proceso.
- SIGSTOP. Cambia el estado del proceso a stopped hasta que se reciba la señal SIGCONT.

Para qué querríamos bloquear una señal? Por ejemplo, durante la ejecución de un handler. Una función es reentrante cuando es segura de ser llamada a si misma. Sería bueno bloquear los handlers que accedan al mismo espacio de memoria, por ejemplo, con memoria compartida.

Bloqueo durante handlers

Existe una forma más detallada de administrar señales:

```
int sigaction (int signum, const struct sigaction * act, struct sigaction * oldact);
```

- signum: señal sobre la que se aplican los cambios.
- act: acción o handler para signum, con su configuración detallada.
- oldact: la configuración anterior.

Uno de los campos que tiene el `struct sigaction` es `sa_mask`, la máscara de bits con las señales que deben ser bloqueadas durante el manejo de la señal `signum`.

Aquí hay un ejemplo, donde además se usan algunas funciones que trabajan sobre esas bitmasks. Se asigna un handler y se bloquea la señal `SIGINT` para la señal `signum`.

```
struct sigaction sa; // acciones a ejecutar.
sigemptyset (&sa.sa_mask) // Borra la máscara.
sigaddset (&sa.sa_mask, SIGINT) // bloquea sigint.
sa.sa_handler = funcion; // asigno un handler.
sigaction (signum, &sa, 0);
```

¿Y si recibimos un `SIGINT` durante el handler? Se prende como pendiente y se entregará la señal más tarde. ¿Y si recibimos otro? La máscara es un mapa de bits, no un contador, así que no se acumulan. Queda un solo `SIGINT` pendiente.

Patrones

Un ejemplo típico de uso de señales es el gracefull quit. Para evitar el while true. Se ejecutará hasta que se reciba la señal, se saldrá del while y se saldrá del programa elegantemente. Pero usar variables globales no está bueno. Cualquier parte del programa podría romperlo. Hay patrones de diseño para atacar esto.

Singleton: tendrá la tabla de las señales y sus handlers. Adapter: transformaba una interfaz en otra. Transforma el header de la fucnión handler para transformarlo en métodos de C++. La tabla es un vector de EventHandlers (hooks). No olvidarse de destruir al singleton manualmente.

Hook: desacopla la registración de la implementación. Es abstracto. El dispatcher es el hook. Cada handler tendrá que tener su propio hook.

Adapter.

Un SIGINT_Handler por ejemplo es una especialización de EventHandler, que tiene que implementar el handleSignal (handler). Además tendrá, por ejemplo, el famoso graceful quit como atributo. No se puede asignar, salvo en el handler, que también es interno al objeto.

Finalmente el while usará el getter.

REVISAR EL DISPACHER.

Ejemplo

Si aparece una señal se entra al handler. El estado del proceso será **running**: *se anula en efecto el sleep o wait o lo que sea.*

Ejercicio

Escribir un programa que registre un programa con handler SIGINT. fork e imprimir pid del hijo. enviar una señal con `kill -2` desde otra consola al hijo. Verificar si el handler del padre se ejecutó. Después lo mismo con exec.