Práctica 3: Señales

75.59 - Técnicas de Programación Concurrente I

Ejercicios

Las señales están relacionadas con las interrupciones al flujo regular de la ejecución de un programa, por ejemplo: error de punto flotante, falla de corriente, alarma, terminación de un proceso hijo, ctrl.-c, ctrl.-z, etc.

1. Estudiar el archivo `signal.h` y hacer una lista de las señales predefinidas indicando la macro, el número y la descripción de la señal.

2. Escribir un programa tal que el proceso correspondiente recibe una señal de alarma utilizando la llamada `alarm()`.

3. Modificar el programa del ejercicio anterior para que capture y procese la señal *SIGALRM*, de acuerdo con una lógica definida por el programador.

4. Escribir un programa que se proteja contra señales *SIGINT*.

5. Explicar cómo funciona la llamada al sistema: `int kill(int pid, int sigCod)`.

6. Documentar el siguiente código:

```
1  int delay;
2  manejarHijo ();
3
4  main ( int argc, char* argv[] ) {
5          int pid;
6          signal ( SIGCHLD,manejarHijo );
7          pid = fork();
8          if(pid == 0) {
9                  execvp (argv[2],&argv[2] );
10                 perror ( "limite" );
11         } else {
12                 sscanf ( argv[1],    % d  , &delay );
13                 sleep ( delay );
14                 printf ( "el Hijo %d excedio el limite y esta siendo eliminado\ n  , pid )
                       ;
15                 kill ( pid,SIGINT );
16         }
17 }
18
19 manejarHijo () {
20         int pidHijo, statusHijo;
21
22         pidHijo = wait ( &statusHijo );
23         printf ( "hijo % terminado dentro de %d segundos \n", pidHijo, delay );
24         exit ( 0 );
25 }
```

7. 1.Hacer un programa que:

   (a) Cree dos hijos tales que ambos entren en un lazo infinito y muestren un mensaje cada segundo

   (b) El programa principal espera 3 segundos y suspende al primer hijo

   (c) El segundo hijo continúa ejecutándose

(d) Luego de otros tres segundos, el padre recomienza al primer hijo, espera algo más y termina con ambos hijos

8. En el siguiente código, explicar cómo es afectado por un ctrl.- c, el proceso hijo:

```
manejarSigint ();

main () {
        int i;
        signal ( SIGINT, manejarSigint );
        if ( fork() == 0 )
                setpgrp ( 0,getpid() );
        printf ( "el proceso de PID %d y PGRP %d esta esperando\n",getpid(), getpgrp(0) );
        for ( i = 1; i <= 3; i++ ) {
                printf ( "El proceso %d sigue vivo\n",getpid() );
                sleep ( 1 );
        }
}


manejarSigint () {
        printf ( "el proceso %d obtuvo un SIGINT \n",getpid() );
        exit ( 1 );
}
```

9. Describir la cabecera `setjmp.h`.

10. Explicar las principales características de las señales.

11. Estudiar qué relación tiene `struct sigaction` con las señales en UNIX.

12. ¿Qué es la estructura de control de señales de un proceso Unix-Linux? Hacer una descripción.

Apuntes

Señales

75.59 - Técnicas de Programación Concurrente I

Facultad de Ingeniería - Universidad de Buenos Aires

Introducción
Origen de las señales
Envío de señales
Bloqueo y administración de señales
Asociación de handlers
Aplicación de patrones de diseño
Bibliografía

Resumen

Introducción (I)

- Son notificaciones que envía el kernel o un proceso a otros procesos, relacionadas con eventos asincrónicos que pueden afectar su comportamiento
- Son similares a las interrupciones que los dispositivos de hardware envían al sistema operativo
- Cada señal es un número entero con un nombre simbólico asociado
  - `/usr/include/linux/signal.h`
  - `kill -l`

Introducción (II)

- Cada señal puede tener una función que la administra llamada *signal handler*
- Cuando el proceso recibe la señal, el SO ejecutará el handler de manera asincrónica, al pasar de modo kernel a usuario.
  - Si el proceso está en espera y no bloqueó la señal, el SO cambia su estado a Running para que el scheduler le otorgue tiempo de uso de la CPU
- Cuando finaliza la ejecución del handler, el proceso continúa su ejecución desde el punto en que fue interrumpido

Introducción (III)

No todos los procesos pueden enviar señales a cualquier otro proceso:

- El kernel y los procesos que se ejecutan como *root* pueden enviar señales a cualquier otro proceso
- El resto de los procesos pueden enviar señales a otros procesos con el mismo *uid* real o efectivo

Ciclo de vida de las señales

1. Generación de la señal: evento que ocurre fuera del control del proceso (*asincrónico*)
2. Entrega de la señal: no es inmediata
3. Administración de la señal: el kernel puede
   - Ignorar la señal
   - Atrapar y manejar la señal
   - Ejecutar la acción *default* de la señal: *terminate*, *dump*, *ignore*, *stop* y *continue*

Origen de las señales

¿Qué eventos dan origen a señales?

- Error en el programa: p.ej. división por cero o acceso a una dirección de memoria no válida
- Petición del usuario: `ctrl+z` o `ctrl+c`
- Finalización de un proceso hijo
- Finalización de un *timer* o alarma
- Llamada a `kill()` o `raise()` en el mismo proceso
- Llamada a `kill()` en otro proceso

Envío de señales

¿Cómo se envía una señal a un proceso?

- Función *kill()*

```
int kill ( pid_t pid,int sig );
```

  - Parámetros:
    - pid: ID del proceso destino
    - sig: señal que se quiere enviar
  - Retorna:
    - 0 en caso de éxito; -1 en caso de error, seteando *errno*
- Función *raise()*

```
int raise ( int sig );
```

  - Parámetros:
    - sig: señal que se quiere enviar
  - Retorna:
    - 0 en caso de éxito; un número distinto de cero en caso de error, seteando *errno*

Bloqueo de señales (I)

Un proceso puede bloquear señales en forma temporal (se retrasa la entrega)

- *Signal mask*: pertenece al proceso e indica qué señales se bloquean; se hereda del padre
- Función *sigprocmask()*:

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

- Parámetros:
  - how: SIG_BLOCK, SIG_UNBLOCK o SIG_SETMASK
  - set: conjunto de señales a bloquear / desbloquear
  - oldset: se utiliza para retornar el valor anterior de *signal mask*

Bloqueo de señales (II)

Señales que no se pueden bloquear

- SIGKILL: finaliza inmediatamente la ejecución del proceso
- SIGSTOP: cambia el estado del proceso a *Stopped* y se ejecuta el scheduler para que elija al siguiente proceso (el proceso queda detenido y sólo continúa si se envía la señal *SIGCONT*)

Administración de señales

Un proceso que recibe una señal puede:

- Ignorar la señal (salvo las excepciones mencionadas anteriormente)
- Definir un *handler* específico para esa señal
- Ejecutar la acción *default* de la señal

Si no se definió una acción específica para una señal, se ejecuta la acción *default*

Asociación de *handlers* (I)

Primera forma: función *signal ()*

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

- Parámetros:
  - signum: número de la señal a la cual se quiere asociar el *handler*
  - handler: puntero a la función *handler*, SIG_IGN para ignorar la señal recibida, o bien SIG_DFL para asignar el *handler* default (ejemplo: SIGINT ejecuta `exit()`)
- Retorna:
  - El puntero al *handler* anterior en caso de éxito
  - SIG_ERR en caso de error

Asociación de *handlers* (II)

Segunda forma: función *sigaction()*: permite un mayor control que *signal()*

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

- Parámetros:
  - signum: número de la señal al cual se quiere asociar el *handler*
  - act: puntero a la acción que se quiere realizar
  - oldact: buffer que se utiliza para retornar el puntero a la acción anterior, generalmente se deja NULL
- Retorna:
  - 0 en caso de éxito
  - -1 en caso de error

Asociación de *handlers* (III)

Estructura `struct sigaction`

```
struct sigaction {
    void        (*sa_handler)(int);
    void        (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t    sa_mask;
    int         sa_flags;
    void        (*sa_restorer)(void);
};
```

- sa_handler: puntero a la función *handler*
- sa_mask: señales que se quieren bloquear durante la ejecución del handler
- sa_flags: opciones relacionadas con la administración de la señal (consultar bibliografía)

Asociación de *handlers* (IV)

Para registrar una función como *signal handler*:

- Definir las señales que se quieren bloquear durante la ejecución del signal handler

```
struct sigaction sa;
sigemptyset ( &sa.sa_mask );
sigaddset ( &sa.sa_mask,SIGINT );
sigaddset ( &sa.sa_mask,SIGQUIT );
```

- Definir el signal handler

```
sa.sa_handler = funcion
```

- Cambiar la acción de la señal

```
sigaction ( signum,&sa,0 );
```


Ejemplo

```
sig_atomic_t graceful_quit = 0;

void SIGINT_handler ( int signum ) {
  assert ( signum == SIGINT );
  graceful_quit = 1;
}

void RegistrarFuncion () {
  struct sigaction sa;
  sigemptyset ( &sa.sa_mask );
  sa.sa_handler = SIGINT_handler;
  sigaction ( SIGINT,&sa,0 );
}

int main ( void ) {
  RegistrarFuncion ();

  while ( graceful_quit == 0 ) {
     cout << "Soy el proceso " << getpid() << endl;
     sleep ( 2 );
  }

  cout << "Fin del proceso" << endl;
  return 0;
}
```

Aplicación de patrones de diseño (I)

La implementación tradicional de señales presenta varios problemas:

- Obliga a la declaración de variables globales
- Solución poco elegante
- No orientada a objetos

Para evitar estos problemas se aplicarán patrones de diseño

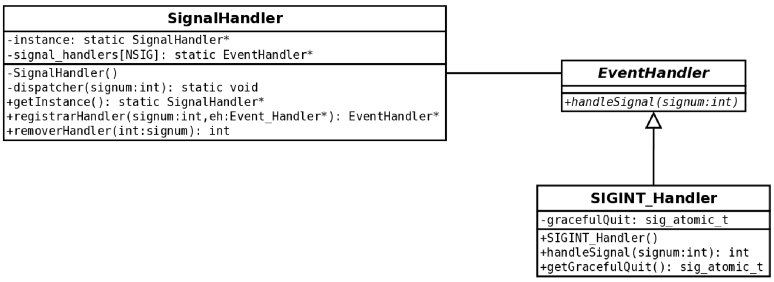
- Patrón de diseño: describe un problema de diseño particular y recurrente que aparece en contextos de diseño específicos, y presenta un esquema genérico para su solución

Aplicación de patrones de diseño (II)

Los patrones que se utilizarán son:

- Patrón Singleton: se aplica porque existe una única tabla de mapeo de señales en el proceso
- Patrón Adapter: transforma una interface en otra. Se aplica para adaptar el header de la función handler a métodos de C++
- Patrón Hook Method: desacopla la registración del handler de su implementación. El método hook es abstracto y debe implementarse en cada handler

Implementación de la solución (I)



Implementación de la solución (II)

```
SignalHandler* SignalHandler :: getInstance () {
    if ( instance == NULL )
        instance = new SignalHandler ();

    return instance;
}

EventHandler* SignalHandler :: registrarHandler(int signum,EventHandler* eh) {
    EventHandler* old_eh = SignalHandler :: signal_handlers [ signum ];
    SignalHandler :: signal_handlers [ signum ] = eh;

    struct sigaction sa;
    sa.sa_handler = SignalHandler :: dispatcher;
    sigemptyset ( &sa.sa_mask );
    sigaddset ( &sa.sa_mask,signum );
    sigaction ( signum,&sa,0 );

    return old_eh;
}

void SignalHandler :: dispatcher ( int signum ) {
    if ( SignalHandler :: signal_handlers[signum] != 0 )
        SignalHandler :: signal_handlers[signum]->handleSignal ( signum );
}
```

Ejemplo (I)

Se hereda la clase SIGINT_Handler de EventHandler

```
class SIGINT_Handler : public EventHandler {

    private:
        sig_atomic_t gracefulQuit;

    public:
        SIGINT_Handler () {
            this->gracefulQuit = 0;
        }

        virtual int handleSignal ( int signum ) {
            assert ( signum == SIGINT );
            this->gracefulQuit = 1;
            return 0;
        }

        sig_atomic_t getGracefulQuit () {
            return this->gracefulQuit;
        }
  };
```


Ejemplo (II)

El main del programa queda de la siguiente manera:

```
int main ( void ) {

    SIGINT_Handler sigint_handler;
    SignalHandler :: getInstance()->registrarHandler(SIGINT,&sigint_handler);

    while ( sigint_handler.getGracefulQuit() == 0 ) {
        cout << "Soy el proceso " << getpid() << endl;
        sleep ( 2 );
    }

    SignalHandler :: destruir ();
    cout << "Termino el proceso" << endl;
    return 0;
}
```

Bibliografía

- Manuales del sistema operativo
- *The Design of the Unix Operating System*, Maurice Bach
- *Understanding the Linux Kernel*, Daniel Bovet y Marco Cesati, tercera edición
- *Applying Design Patterns to Simplify Signal Handling*, Douglas Schmidt, http://www.cs.wustl.edu/~schmidt/signal-patterns.html
- *ACE Framework (ADAPTIVE Communication Environment)*, http://www.cs.wustl.edu/~schmidt/ACE.html
- *Design Patterns*, E. Gamma, R. Helm, R. Johnson & J. Vlissides

Fuentes de los ejemplos

Listado 1: Programa principal

```
#include <iostream>
#include <unistd.h>

#include "SIGINT_Handler.h"
#include "SignalHandler.h"

void bloquearSigint ();

using namespace std;

int main () {

        // event handler para la senial SIGINT (-2)
        SIGINT_Handler sigint_handler;

        // se registra el event handler declarado antes
        SignalHandler :: getInstance()->registrarHandler ( SIGINT,&sigint_handler );

        // mientras no se reciba la senial SIGINT, el proceso realiza su trabajo
        while ( sigint_handler.getGracefulQuit() == 0 ) {
                cout << "Soy el proceso " << getpid() << endl;
                sleep ( 2 );
        }

        // se recibio la senial SIGINT, el proceso termina
        SignalHandler :: destruir ();
        cout << "Termino el proceso" << endl;
        return 0;
}

void bloquearSigint () {
        sigset_t sa;
        sigemptyset ( &sa );
        sigaddset ( &sa,SIGINT );
        sigprocmask ( SIG_BLOCK,&sa,NULL );
}
```

Listado 2: Clase EventHandler

```
#ifndef EVENTHANDLER_H_
#define EVENTHANDLER_H_

class EventHandler {

public:
        virtual int handleSignal ( int signum ) = 0;
        virtual ~EventHandler () {};
};

#endif /* EVENTHANDLER_H_ */
```

Listado 3: Clase SIGINT_Handler

```
#ifndef SIGINT_HANDLER_H_
#define SIGINT_HANDLER_H_

#include <signal.h>
#include <assert.h>

#include "EventHandler.h"

class SIGINT_Handler : public EventHandler {

        private:
                sig_atomic_t gracefulQuit;

        public:

                SIGINT_Handler () : gracefulQuit(0) {
                }

                ~SIGINT_Handler () {
                }

                virtual int handleSignal ( int signum ) {
```

```
23                          assert ( signum == SIGINT );
24                          this->gracefulQuit = 1;
25                          return 0;
26                  }
27
28                  sig_atomic_t getGracefulQuit () const {
29                          return this->gracefulQuit;
30                  }
31
32  };
33
34  #endif /* SIGINT_HANDLER_H_ */
```

Listado 4: Clase SignalHandler

```
1   #ifndef SIGNALHANDLER_H_
2   #define SIGNALHANDLER_H_
3
4   #include <signal.h>
5   #include <stdio.h>
6   #include <memory.h>
7
8   #include "EventHandler.h"
9
10  class SignalHandler {
11
12          private:
13                  static SignalHandler* instance;
14                  static EventHandler* signal_handlers [ NSIG ];
15
16                  SignalHandler ( void );
17                  static void dispatcher ( int signum );
18
19          public:
20                  static SignalHandler* getInstance ();
21                  static void destruir ();
22                  EventHandler* registrarHandler ( int signum,EventHandler* eh );
23                  int removerHandler ( int signum );
24
25  };
26
27  #endif /* SIGNALHANDLER_H_ */
```

Listado 5: Clase SignalHandler

```
1   #include "SignalHandler.h"
2
3   SignalHandler* SignalHandler :: instance = NULL;
4   EventHandler* SignalHandler :: signal_handlers [ NSIG ];
5
6   SignalHandler :: SignalHandler () {
7   }
8
9   SignalHandler* SignalHandler :: getInstance () {
10
11          if ( instance == NULL )
12                  instance = new SignalHandler ();
13
14          return instance;
15  }
16
17  void SignalHandler :: destruir () {
18          if ( instance != NULL ) {
19                  delete ( instance );
20                  instance = NULL;
21          }
22  }
23
24  EventHandler* SignalHandler :: registrarHandler ( int signum,EventHandler* eh ) {
25
26          EventHandler* old_eh = SignalHandler :: signal_handlers [ signum ];
27          SignalHandler :: signal_handlers [ signum ] = eh;
28
29          struct sigaction sa;
30          memset(&sa, 0, sizeof(sa));
31          sa.sa_handler = SignalHandler :: dispatcher;
32          sigemptyset ( &sa.sa_mask );    // inicializa la mascara de seniales a bloquear durante
                la ejecucion del handler como vacio
33          sigaddset ( &sa.sa_mask,signum );
34          sigaction ( signum,&sa,0 );     // cambiar accion de la senial
```

```
35
36          return old_eh;
37  }
38
39  void SignalHandler :: dispatcher ( int signum ) {
40
41          if ( SignalHandler :: signal_handlers [ signum ] != 0 )
42                  SignalHandler :: signal_handlers [ signum ]->handleSignal ( signum );
43  }
44
45  int SignalHandler :: removerHandler ( int signum ) {
46
47          SignalHandler :: signal_handlers [ signum ] = NULL;
48          return 0;
49  }
```