Boletín 5: Señales y el comando `bjobs`

Ampliación de Sistemas Operativos

Dpto. Ingeniería y Tecnología de Computadores (DITEC)

Universidad de Murcia

Curso 2017/2018

Índice

1. Señales

Señales

Definición

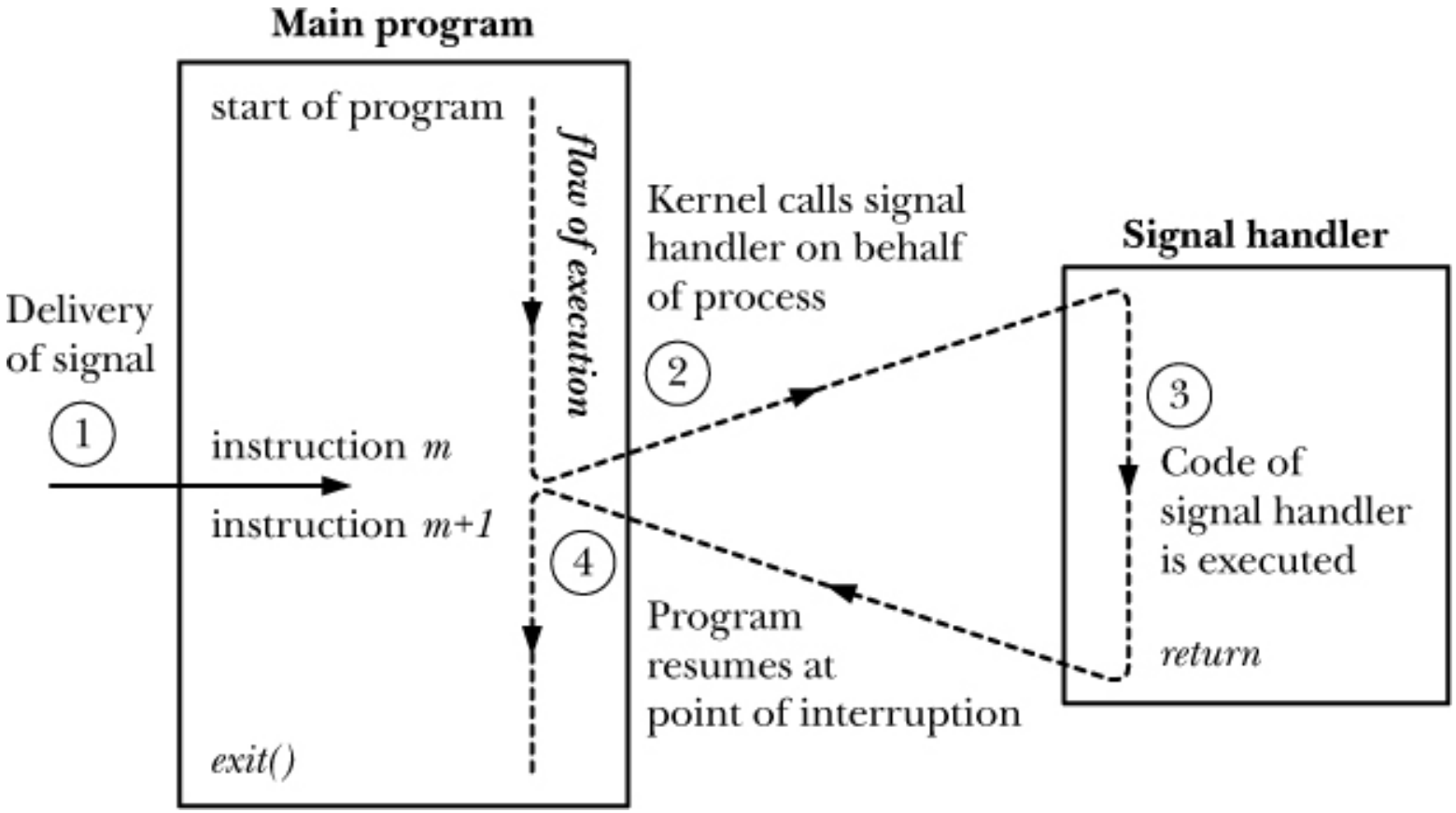
- Una señal es una notificación a un proceso cuando sucede un evento
  - Excepción hardware: Dirección de memoria inválida (`SIGSEGV`)
  - Evento software: Un proceso hijo ha terminado (`SIGCHLD`)
  - Notificación de E/S: Interrupción con CTRL+C (`SIGINT`)
- Un proceso puede recibir una señal enviada por otro o por el kernel
- Una señal es un entero ≥ 1 definida en `signal.h` como `SIGXXXX`
- Desde que una señal se genera hasta que se entrega está *pendiente*
- Una señal pendiente se entrega inmediatamente al proceso si se está ejecutando o tan pronto como reanude su ejecución en caso contrario
- No obstante, un proceso puede bloquear la entrega de una señal mediante una *máscara de señales* que la deja en estado *bloqueada*
- Referencia:
  `http://man7.org/linux/man-pages/man7/signal.7.html`

Señales

Acciones por defecto

- Cuando un proceso recibe una señal, ejecuta la acción por defecto:
  - La señal es ignorada
  - El proceso es matado (*killed*)
  - Se genera un fichero *core* y el proceso es matado
  - El proceso se detiene (*stopped*)
  - El proceso reanuda su ejecución (*resumed*)
- Un proceso puede cambiar la acción por defecto:
  - Ignorando la señal
  - Ejecutando un manejador de señales (*signal handler*)
  - Restaurando la acción por defecto

- Un manejador de señales es una función definida por el usuario que lleva a cabo acciones apropiadas en respuesta a una señal concreta



Servicios POSIX para señales

- Consulta y modificación de la acción por defecto:

```
1  #include <signal.h> /* POSIX */
   int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);
3
   struct sigaction {
5      void (*sa_handler)(int);    /* Address of handler */
       sigset_t sa_mask;           /* Signals blocked during handler invocation */
7      int sa_flags;               /* Flags controlling handler invocation */
       void (*sa_restorer)(void); /* Not for application use */
9  };
```

  - `sig` es la señal para la cual se realiza la consulta o modificación
  - `sa_handler` se refiere al manejador de la señal o a las constantes `SIG_IGN` (ignorar la señal) o `SIG_DFL` (restaurar la acción por defecto)
  - `sa_mask` define qué señales se bloquearán durante la ejecución del manejador de la señal además de las ya bloqueadas en la máscara de señales del proceso con `sigprocmask()` (incluida la señal `sig`)
  - `sa_flags` es una máscara de bits con opciones que afectan a la ejecución del manejador de la señal, por ejemplo, `SA_RESETHAND` restablece la acción por defecto cuando se ejecuta el manejador

Servicios POSIX para señales

- Consulta y modificación de la máscara de señales:

```
1   #include <signal.h> /* POSIX */
    int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

  - Si `set` es `NULL`, `sigprocmask()` devuelve la máscara de señales del proceso en `oldset`
  - En caso contrario, `sigprocmask()` modifica la máscara de señales del proceso en función de `how`:
    - `SIG_BLOCK` añade `set` a la máscara de señales del proceso
    - `SIG_UNBLOCK` elimina `set` de la máscara de señales del proceso
    - `SIG_SETMASK` establece la máscara de señales del proceso a `set`

    `sigset_t` es un conjunto de señales gestionado con `sigemptyset()`, `sigfillset()`, `sigaddset()`, `sigdelset()` y `sigismember()`
  - `SIGKILL` y `SIGSTOP` no se pueden capturar, bloquear o ignorar

Servicios POSIX para señales

- Código: 

```
int main(int argc, char *argv[])
{
    /* Block signal SIGSEGV */
    sigset_t blocked_signals;
    sigemptyset(&blocked_signals);
    sigaddset(&blocked_signals, SIGSEGV);
    if (sigprocmask(SIG_BLOCK, &blocked_signals, NULL) == -1) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }

    /* Establish signal handler for SIGINT */
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa)); /* SIGSEGV!!! */
    sa.sa_handler = signal_handler;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction 1");
        exit(EXIT_FAILURE);
    }

    while(1)      /* Loop forever, waiting for signals */
    {
        pause(); /* Block until a signal is caught */
        printf("Caught SIGINT %d times\n", shc);
    }

    /* The program will never get here! */
    return EXIT_SUCCESS;
}
```

Servicios POSIX para señales

- Código (cont.):

```
1   static int shc = 0; /* signal handler counter */

3   static void signal_handler(int sig)
    {
5       if (sig == SIGINT) shc++; /* SIGINT (^C): increase the counter */
    }
```

Señales

- (0.5p) **EJERCICIO 1**: Tratamiento de las señales `SIGINT` y `SIGQUIT`:
  - `simplesh` debe bloquear la señal `SIGINT` (CTRL+C)
  - `simplesh` debe bloquear la señal `SIGQUIT` (CTRL+\)
- Para ello, únicamente se pueden usar llamadas al sistema POSIX o funciones de la biblioteca estándar de C (C11)
- Llamadas `POSIX` y/o `glibc` a considerar:
  - `sigemptyset()`
  - `sigaddset()`
  - `sigprocmask()`

Señales

- (1.0p) **EJERCICIO 2**: Identifica y soluciona las defiencias en la implementación de los comandos en segundo plano en `simplesh`

```
simplesh> sleep 1 &
/* Se muestra de nuevo el prompt inmediatamente */
simplesh> ls
simplesh> Makefile simplesh simplesh.c simplesh.o
/* No se muestra el prompt tras el listado de ficheros */
```

  - La ejecución de comandos en segundo plano no debe interferir con la ejecución de los comandos en primer plano en ningún caso
  - Cuando un comando en segundo plano se ejecuta o cuando termina, se debe enviar su `[PID]` a `stdout`
  - Cuando un comando en segundo plano termina, `simplesh` debe evitar que se convierta en un proceso *zombie*

```
simplesh> sleep 1 &
[PID]             /* Se muestra de nuevo el prompt inmediatamente */
simplesh> [PID]   /* Transcurrido 1 segundo... */
                  /* Intro */
simplesh> ls
Makefile simplesh simplesh.c simplesh.o
```

- Para ello, únicamente se pueden usar llamadas al sistema POSIX o funciones de la biblioteca estándar de C (C11)
- Llamadas `POSIX` y/o `glibc` a considerar: `waitpid()` y `sigaction()`

2. El **comando** `bjobs`

El comando bjobs

- (0.5p) **EJERCICIO 3**: Implementa el comando interno `bjobs`:

```
    simplesh> sleep 5 &
 2  [PID0]
    simplesh> sleep 10 &
 4  [PID1]
    simplesh> bjobs      /* Antes de que transcurran 5 segundos... */
 6  [PID0]
    [PID1]
 8  simplesh> [PID0]     /* Transcurridos 5 segundos */
                         /* Intro */
10  simplesh> bjobs      /* Transcurridos entre 5 y 10 segundos... */
    [PID0]
12  simplesh> [PID1]     /* Transcurridos 10 segundos */
```

- Para ello, únicamente se pueden usar llamadas al sistema POSIX o funciones de la biblioteca estándar de C (C11)
- Llamadas `POSIX` y/o `glibc` a considerar:
  - `waitpid()`
  - `sigaction()`
  - `kill()`
- Para implementar `bjobs`, escribe la función `run_bjobs()` e inserta llamadas a la misma en `simplesh.c` donde sea necesario

El comando bjobs

- Si se proporciona el parámetro `-h`, `bjobs` debe enviar a `stdout`:

```
simplesh> bjobs -h
Uso: bjobs [-k] [-h]
            Opciones:
            -k Mata todos los procesos en segundo plano
            -h help
```

- (0.5p) **Opcional**: El parámetro `-k` mata todos los procesos en segundo plano:

```
simplesh> sleep 10 &
[PID0]
simplesh> sleep 10 &
[PID1]
simplesh> bjobs     /* Antes de que transcurran 10 segundos... */
[PID0]
[PID1]
simplesh> bjobs -k  /* Antes de que transcurran 10 segundos... */
simplesh> [PID0]
[PID1]

simplesh> bjobs     /* Antes de que transcurran 10 segundos... */
```

El comando `bjobs`

- Implementación:
  1. Bloquear las señales `SIGINT` y `SIGQUIT` antes de `parse_args()`
  2. Modificar `run_cmd()` para que la ejecución de los comandos en segundo plano no interfiera con la de los comandos en primer plano
  3. Modificar `run_cmd()` para que la ejecución de los comandos en segundo plano envíe `[PID]` a `stdout`
  4. Implementar un manejador de señales para la señal `SIGCHLD`
     - El manejador debe enviar `[PID]` a `stdout` SÓLO cuando termine la ejecución de un proceso creado para un comando en segundo plano
     - El manejador tiene que evitar que los procesos creados para comandos en segundo plano se conviertan en procesos *zombies* al terminar
     - **Referencia**: *Reap zombie processes using a SIGCHLD handler*
  5. Procesar los parámetros de `bjobs` con `getopt()` (`man 3 getopt`)
     - **Nota**: Antes de volver a usar `getopt()`, se debe inicializar `optind` a `1`
  6. Implementar la función `run_bjobs()` e insertar llamadas a la misma igual que para las funciones `run_*()` de los boletines anteriores
     Importante: Para resolver los ejercicios 2 y 3 se necesita seguir la pista a los procesos en segundo plano *activos* en cada momento