EMBEDDED OPERATING SYSTEMS

Embedded Linux on Beaglebone Black

Signal: What?

- Notification delivery mechanism
  - Notification to a process
    - That an event has occurred
- Akin to ‘software interrupts’
- Who can send signals?
  - Kernel
  - Any process with sufficient permissions
- Asynchronous in nature
  - Receiving process can not predict its arrival

Signals on Linux

- Linux defines 64 signals
  - Header file: ***#include <signal.h>***
- Each signal has an integer value
  - Numbered from 1 to 64
  - Also given names (SIGXXX)
    - Example: Signal 2 is called SIGINT (terminal interrupt signal)
- Categories of signals
  - Standard
    - Numbered 1 to 31
    - Used by kernel
  - User-defined
    - Numbered 32 to 64

Signals: Kernel generated

- Kernel generates and sends signals
  - To processes

- Examples:
  - SIGSEGV
    - Memory segment access violation
  - SIGINT
    - Terminal interrupt (User pressed CTRL-C)
  - SIGCHLD (to parent)
    - Child process terminated
  - SIGXCPU
    - Process exceeded (pre-defined) CPU usage

Signal: Terminology

- A signal is **generated** when an event occurs
- A signal is **delivered** to the process
  - Which could take some responsive **action**
- A signal is **pending**
  - Between generation and delivery
- A process can **block** a signal
  - By adding its entry to its **signal mask**
    - Signal mask: Set of signals whose delivery is blocked
  - Pending signal is delivered after it is **unblocked**

Signal: Actions (disposition)

- On receipt of a signal, a process can:
  - **Ignore**
    - Ignore the signal and go ahead
  - **Terminate**
    - The process receiving it is terminated / killed
  - **Core dump**
    - The process generates a core dump and terminates
  - **Stop**
    - The process execution is suspended
  - **Continue**
    - The process execution resumes from suspended point
- Default action for a signal is signal-specific

Standard Signals and Actions

Name	Description	Default
SIGABRT	Abort process	Core
SIGALRM	Real-time timer expiration	Term
SIGBUS	Memory access error	Core
SIGCHLD	Child stopped or terminated	Ignore
SIGCONT	Continue if stopped	Cont
SIGFPE	Arithmetic exception	Core
SIGHUP	Hangup	Term
SIGILL	Illegal Instruction	Core
SIGINT	Interrupt from keyboard	Term
SIGIO	I/O Possible	Term
SIGKILL	Sure kill	Term
SIGPIPE	Broken pipe	Term
SIGPROF	Profiling timer expired	Term
SIGPWR	Power about to fail	Term
SIGQUIT	Terminal quit	Core
SIGSEGV	Invalid memory reference	Core
SIGSTKFLT	Stack fault on coprocessor	Term
SIGSTOP	Sure stop	Stop
SIGSYS	Invalid system call	Core
SIGTERM	Terminate process	Term
SIGTRAP	Trace/breakpoint trap	Core
SIGTSTP	Terminal stop	Stop
SIGTTIN	Terminal input from background	Stop
SIGTTOU	Terminal output from background	Stop
SIGURG	Urgent data on socket	Ignore
SIGUSR1	User-defined signal 1	Term
SIGUSR2	User-defined signal 2	Term
SIGVTALRM	Virtual timer expired	Term
SIGWINCH	Terminal window size changed	Ignore
SIGXCPU	CPU time limit exceeded	Core
SIGXFSZ	File size limit exceeded	Core

Signals: Stop, Continue

- SIGTSTP
  - “Stop terminal” signal
    - Halts process execution and suspends it
    - Shell example: Pressing of **CTRL-Z**
- SIGSTOP
  - “Stop terminal surely” signal
    - Cannot be ignored
- SIGCONT
  - “Terminal continue” signal
    - Continue from the point of stoppage

Signal Handling

- For some signals, default action can be changed
  - By process
  - By defining a signal handler
    - Execute user-defined / process-defined function
    - Called “handling” / “catching” the signal
- Default actions cannot be changed for:
  - SIGKILL
  - SIGSTOP

Changing Signal Action (1/2)

- Signal changing API

  # include <signal.h>
  *int **sigaction** (int sig, const struct sigaction *new_act, struct sigaction *old_act);*

  sig: Number of signal to be changed
  new_act: Struct for new action (NULL for no change)
  old_act: Returns previous state (NULL if we don't care)

- Retrieving the current state:

  int sig_num = 12;
  struct sigaction old_act;
  sigaction(sig_num, NULL, &old_act);

Changing Signal Action (2/2)

- Definition of ***struct sigaction***

  *struct **sigaction** {*
  *void (*sa_handler)(int);*
  sigset_t sa_mask ;
  int sa_flags ; … };

  sa_handler:
  - address of signal handler
    - *note the API: **void** my_signal_handler**(int)***
  - SIG_IGN (for ignoring the signal)
  - SIG_DFL (for reverting to default action)

  sa_mask: signals to block, used through *sigsetopts()*
  sa_flags: bit mask of flags affecting handler invocation

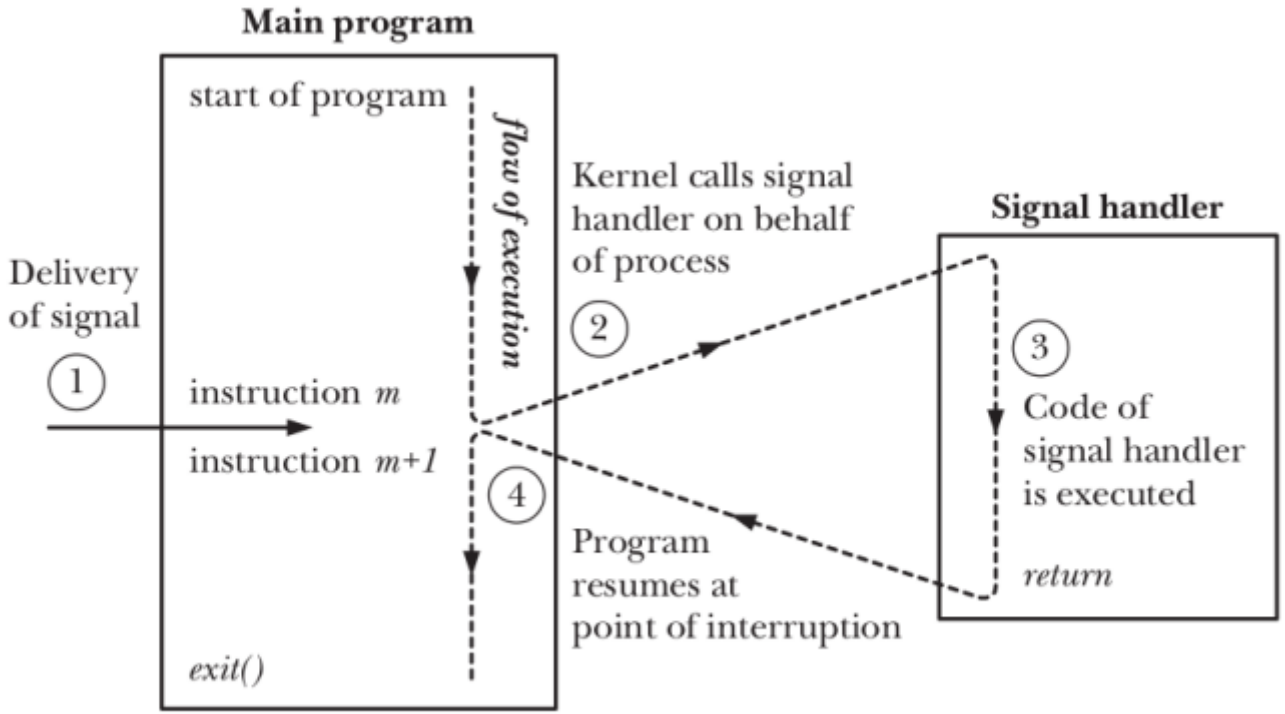
Ignoring a signal

- Setting up function code to ignore a signal

```
int ignore_signal (int sig) {
    struct sigaction sa;

    sa.sa_handler = SIG_IGN;
    sa.sa_flags = 0;
    sigemptyset (&sa.sa_mask );
    return sigaction (sig, &sa, NULL );
    }
```

Signal Handler Invocation



Printing Signal Descriptions

- API to print the string value of a signal

  #define _GNU_SOURCE

  #include <string.h>

  *char ****strsignal*** *(int sig);*

  - **NSIG** is an internally defined constant
  - Equal to 1 greater than maximum signal number

- Similar to ***strerror()***

Signals exercise

- ***catch-ctrlc.c***
  - Catch the SIGINT (CTRL-C for terminal)
  - Assign a user-defined handler for this signal
  - Compile and run the code
  - Observe what happens
    - When you do not press CTRL-C at all
    - When you press CTRL-C

- ***forking-signals.c***
  - Here, parent kills the child process it created
    - After ***parent_sleep*** reduces to 0
  - Notice how parent sends a ***kill(SIGINT)*** to child
  - Notice how parent now comes to know how child was killed
    - Through SIGCHLD handler
    - Also by probing child's return status

Pipes

- Pipes are ways to communicate
  - Between processes
  - More secure compared to shared memory
- Usage:
  - 1 process creates/opens the pipe
    - And starts writing to the pipe
  - Another process opens this pipe as a file
    - And starts reading from the pipe
- Only 1-way communication possible between processes
  - Unidirectional
- Data is process persistent

Pipes and FIFOs

- Pipes are of 2 types in Linux
  - Unnamed (without backing file)
  - Named (with backing file – called FIFOs)

- Pipes can be used:
  - From the shell
  - From C programs using pipe API

Pipe usage from Shell

- The ‘|’ character denotes an unnamed pipe
  - It should be placed between 2 shell commands
  - The command to the left of ‘|’ is the writer
  - The command to the right of ‘|’ is the reader

- Example:

  $ ls / | sort –r

  - ***ls /*** is the writer – it writes to the unnamed pipe
  - ***sort –r*** is the reader – it reads from the unnamed pipe

- Unnamed shell pipes can be chained

  $ ls / | sort –r | uniq

  - Here, there are 2 pipes and 3 commands
  - Chained together from left to right through pipes

Pipe API in C

- We need 2 file descriptors (fd's) to create a pipe
  - The first is called the "read" end
  - The second is called the "write" end
  - Example:

    int pipefds[2];

- This will be the input for the pipe() API command

  *int **pipe**(int pipefds[2]);*
  - Return values:
    - 0: Success
    - -1: Error, errno will contain the error number

- Once created, *pipefds[]* is the pipe
  - We can use open(), read(), write() and close() APIs

Pipe exercise

- ***unnamed-pipe.c***
  - We use a parent – child process combination
    - Parent creates the pipe
      - It opens “WriteEnd” and closes “ReadEnd”
      - And writes a message to pipe (like a file)
    - Child reads from the pipe
      - It opens “ReadEnd” and closes “WriteEnd”
      - And reads from the pipe (like a file)
  - Notice the printing sequence of the messages
  - Notice where the pipe is created
    - In system memory
    - By the OS / kernel
  - The pipe resources are reclaimed by kernel
    - When all connected processes terminate

FIFOs

- FIFOs in Linux are named pipes
  - They have a backing file on the file system
- Behaviour is identical to pipes
  - 1-way communication from writer to reader
- Again, FIFOs can be used
  - From the shell
  - From C programs using the FIFO API

FIFO usage from shell

- We use the ***mkfifo*** command
  - To create the FIFO
- Write:
  - We could use a write command
    - Like **echo** or **cat** to write to the FIFO
- Read:
  - We could use a **cat** command
    - To read from the FIFO and type to stdout

- Example:

  $ mkfifo myfifo-001
  $ cat > myfio-001
  Hello world <ENTER>
  CTRL-C
  $ unlink mfifo-001

  $ cat myfifo-001 (2nd terminal)

FIFO API in C

- FIFOs are created by the mkfifo() command

  #include <sys/types.h>
  #include <sys/stat.h>
  *int **mkfifo**(const char *pathname, mode_t mode);*

  - Return values:
    - 0: Success
    - -1: Failure, errno will be set accordingly
  - After creation, the FIFO is treated like a file
    - *open(), close(), read(), write()*

- When done, we call ***unlink(pathname)*** to clean up

FIFO exercise

- ***fifo-writer.c***
  - Creates the FIFO (./***myfifo-001***)
  - Opens it like a file
  - In a loop
    - After random sleeps (within 5 seconds max)
      - Writes a large random integer to the FIFO
    - Write is in binary format
  - Waits for reader to pick it up
  - Finally, unlinks the FIFO

- ***fifo-reader.c***
  - Opens the already created FIFO
  - Reads the integer from FIFO
    - Read is in binary format
    - Computes whether it is prime
  - Ends when it reaches the end-of-stream
    - When ***read()*** outputs a 0
  - Prints the final output

Message Queues

- Pipes and FIFO's are strictly FIFO
  - First In First Out order maintained
- What if we wanted to read messages
  - Out of order? Perhaps in some order of priority?

- Message queues
  - Treat messages like packets
    - With a "priority" and a "payload"
  - Reading can be sequenced using the "priority"
    - 0 is lowest priority

POSIX Message Queue API

- Create/open an MQ

  #include <fcntl.h> / For O_* constants */*
  #include <sys/stat.h> / For mode constants */*
  #include <mqueue.h>
  *mqd_t **mq_open**(const char *name, int oflag, mode_t mode, struct mq_attr *attr);*

- Send to an MQ

  *int **mq_send**(mqd_t mqdes, const char msg_ptr[.msg_len], size_t msg_len, unsigned int msg_prio);*

- Receive from an MQ

  *ssize_t **mq_receive**(mqd_t mqdes, char msg_ptr[.msg_len], size_t msg_len, unsigned int *msg_prio);*

- Close an MQ

  *int **mq_close**(mqd_t mqdes);*

- Unlink the MQ

  *int **mq_unlink**(const char *name);*

POSIX MQ: mq_attr

- POSIX MQ's are defined by my_attr

```
struct mq_attr {
    long mq_flags; // Message queue description flags
    long mq_maxmsg; // Maximum number of messages on queue
    long mq_msgsize; // Maximum message size (in bytes)
    long mq_curmsgs; // Number of messages currently in queue
    };
```

- Usually set when the MQ is opened
  - By ***mq_open()*** call
  - Later, by ***mq_setattr()***
  - Obtained by ***mq_getattr()***

MQ exercise

- ***mq-writer.c***
  - Implements the MQ writer
  - Creates an MQ and opens it
  - Sends messages to MQ
    - With differing **priorities**
  - Closes the MQ

- ***mq-reader.c***
  - Implements the MQ reader
  - Opens the already created MQ
  - Receives messages from MQ
    - In **decreasing priority order**!
    - Till it runs empty
      - Notice O_NONBLOCK flag and EAGAIN check
  - Closes the MQ and unlinks the MQ handle

THANK YOU!