Operating Systems
2010/2011

Signals

Johan Lukkien
Signals

• A signal is a software generated event
  – notification of state change
    • encapsulation of physical event
  – usually: sent from a process to a process
    • in POSIX, not between threads!
    • the sender can also be the kernel

• Two views:
  – asynchronous message transfer
    • process will handle these messages by a dedicated message handler, when it wants to
    • however, signals usually cannot carry much data (hence, not much content)
  – asynchronous function call
    • process execution interrupted at an arbitrary moment
    • in fact, “concurrent” call into the process space (by an implicit new thread)
  – both views abstract from signals and their handling as if another thread executes the handler function concurrently with the other process code
POSIX 1003.1 signals

- Software exception/interrupt/trap for
  - event notification (including synchronization)
    - timer, message arrival, error conditions, ....
- Asynchronous
  - interrupt current execution
  - can realize ‘poor man’s concurrency’
- Handler function
  - invoked upon receipt of signal: in fact an asynchronous call into a process
- Masking
  - to block specific signals temporarily
- Performance & use
  - fairly bad performance, in general, but can be optimized
    - signal delivery is a complicated operation
  - few distinct signal types
  - signals carry no or little data
  - there is no signal precedence
  - signals may be lost
Using signals

- Using signals, with Posix functions
  - setting up a handler for a particular signal: `signal()`
  - sending a signal: `kill()`
  - waiting for signals to come in: `sigwait()`
  - ignoring signals: `sigmask()`
  - which thread is going to execute it?
    - not well resolved in general in Posix
    - can associate it with a new thread, i.e., let the handler function be started as a new thread
Example: Linux

```c
#include <signal.h>
/* defines, among others, constants for types of signals */
#define SIGHUP          1       /* Hangup (POSIX). */
#define SIGINT          2       /* Interrupt (ANSI). */
#define SIGQUIT         3       /* Quit (POSIX). */
#define SIGILL          4       /* Illegal instruction (ANSI). */
#define SIGTRAP         5       /* Trace trap (POSIX). */
#define SIGABRT         6       /* Abort (ANSI). */
#define SIGIOT          6       /* IOT trap (4.2 BSD). */
#define SIGBUS          7       /* BUS error (4.2 BSD). */
#define SIGFPE          8       /* Floating-point exception (ANSI). */
#define SIGILL          9       /* Kill, unblockable (POSIX). */
#define SIGUSR1        10       /* User-defined signal 1 (POSIX). */
#define SIGSEGV        11       /* Segmentation violation (ANSI). */
#define SIGUSR2        12       /* User-defined signal 2 (POSIX). */
#define SIGPIPE        13       /* Broken pipe (POSIX). */

typedef void (*sighandler_t)(int); /* handler function prototype */
int kill(pid_t pid, int sig);    /* send signal to process */
sighandler_t signal(int signum, *struct sigaction *act, struct sigaction *oldact);
    /* associate handler with signal */
    /* set mask, to indicate which are caught */
    /* and ignored */
int sigsuspend (mask);           /* pause and wait for signal in mask */
```
Example

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#define true 1

void SigHandler (signum) {
  switch (signum) {
      case SIGINT:  printf ("You typed ^C\n");
                      break;
      case SIGALRM: printf ("Received Alarm call - exiting\n");
                      default:      exit (0);
  } }

void main () {
  alarm (2);

  signal (SIGALRM, SigHandler);
  signal (SIGINT, SigHandler);
  /* do the normal useful stuff */
  while (true) sleep (1);
}
```

2R320/programs> ./a.out
You typed ^C
You typed ^C
Received Alarm call - exiting

- `alarm` sets a `SIGALRM` signal to happen after a specified number of seconds
- Function `signal()` associates a handler with a signal
- Typing `^C` generates a `SIGINT` signal.
Real-time signals (POSIX 1003.1b)

- Real-time signals
  - flexible signal identification
    - many signals types, signals carry data
    - can be used as an asynchronous message passing facility
  - signals can be queued and have precedence
    - lowest number first
  - support for efficient signal catching

- Using signals:
  - setting up a handler for a particular signal: `sigaction()`
  - sending a signal: `sigqueue()`
  - setting up an event (e.g. timer event) that is sent upon generation of a signal: `sigevent` structure – some payload is admitted
  - waiting for signals to come in: `sigwaitinfo()` – more as asynchronous messages
    - more efficient than using handler functions
Summary: Communication & synchronization primitives

- **Shared memory [with multi-threading]**
  - just shared variables, e.g., event flags
  - semaphores, mutexes
  - condition variables
  - readers/writers locks
  - monitors

- **Message passing**
  - streaming: pipes, fifos, sockets
  - structured: message queues
  - asynchronous, buffered or synchronous

- **Signals**
  - asynchronous messages, i.e., interrupt the flow of control
  - may regard the *signal handler* as a concurrent thread, that is (synchronously) waiting for an event
Synchronization primitives in systems

- **Posix:**
  - Semaphore (counting)
  - Mutex (between threads only; two states)
  - Condition variables with Wait/Signal

- **Win 32**
  - Semaphore and Mutex (synchronization object)
    - WaitForSingleObject
    - ReleaseSemaphore
  - Critical Section (= mutex between threads of a process)
    - EnterCriticalSection
    - LeaveCriticalSection
  - Event
    - WaitForSingleObject
    - SetEvent
    - ResetEvent
    - PulseEvent (~buffersize=0)

- **Java object**
  - Monitor (“synchronized”)
  - notify; wait (like Posix conditions)