ECE391 Computer System Engineering Lecture 24

Dr. Zbigniew Kalbarczyk
University of Illinois at Urbana- Champaign

Fall 2018

Lecture Topics

Signals

Introduction to Signals

- Signals are the user-level analogue of interrupts
- Similarities
 - asynchronous w.r.t. program execution
 - not queued like a physical line; sending twice may cause handler to execute once or twice
 - can be ignored, blocked, or caught
 - but has some NMIs (SIGKILL, SIGSTOP)
 - handler can be changed from default
 - only data given to handler is signal # (traditional model)
 - signal is (by default) blocked while handler executes

Introduction to Signals

Differences

- generated by <u>software</u> (kernel <u>or</u> a program via a system call)
- no "device" associated with signal; only software with permission can send a signal (single permission allows any signal to be sent)
- further differences with POSIX model (as opposed to traditional model)
 - new real-time signals are queued (one invocation per signal sent)
 - can be sent to threads <u>or</u> processes
 (latter delivered to any thread in a process)
 - siginfo structure contains additional information about the signal (e.g., address of exception)

	default	
name	action	description
SIGHUP	terminate	hangup detected on controlling terminal, or death of controlling process
SIGINT	terminate	interrupt (CTRL-C) from keyboard
SIGQUIT	dump core	quit (CTRL-\) from keyboard
SIGILL	dump core	illegal instruction
SIGTRAP	dump core	trace/breakpoint trap
SIGABRT	dump core	abort signal from abort ()
SIGIOT		IOT trap; a synonym for SIGABRT
SIGBUS	dump core	bus error (bad memory access)
SIGFPE	dump core	floating point exception
SIGKILL	terminate	kill signal
	(always)	
SIGUSR1	terminate	user-defined signal l
SIGSEGV	dump core	invalid memory reference
SIGUSR2	terminate	user-defined signal 2
SIGPIPE	terminate	broken pipe: write to pipe with no readers
SIGALRM	terminate	timer signal from alarm()
SIGTERM	terminate	termination signal
SIGSTKFLT	terminate	stack fault on coprocessor
SIGCHLD	ignore	child stopped or terminated
SIGCLD		synonym for SIGCHLD
SIGCONT	continue	continue if stopped
SIGSTOP	stop	stop process
	(always)	
SIGTSTP	stop	stop (CTRL-Z) from keyboard
SIGTTIN	stop	keyboard (tty) input for background process
SIGTTOU	stop	console (tty) output for background process
SIGURG	ignore	urgent condition on socket
SIGXCPU	dump core	CPU time limit exceeded
SIGXFSZ	dump core	file size limit exceeded
SIGVTALRM	terminate	virtual alarm clock
SIGPROF	terminate	profiling timer expired
SIGWINCH	ignore	window resize signal
SIGIO	terminate	I/O now possible
SIGPOLL		pollable event; synonym for SIGIO
SIGPWR	terminate	power failure
SIGSYS	dump core	bad system callSteven Lumetta, Zbigniew Kalbarczyk. ECE3

List of Signals and Behaviors

Signal Behavior

- Set to default by kernel
- Controlled by user program
- Stored in kernel (data struct sigaction)
- Used by kernel

Signal Behavior

From /usr/include/bits/sigaction.h

```
traditional
 struct sigaction {
    void (*sa handler)(int);
    void (*sa sigaction)(int, siginfo_t*, void*)
                                                                     don't use/set both!
    sigset t sa mask;
                                           POSIX
    int sa flags;
                                  signals to mask while executing
                                  (allows arbitrary prioritization)
 };
SA NOCLDSTOP
                 don't send signal when children stop
                  (used for SIGCHLD only)
SA RESETHAND
                 reset handler to default when signal delivered
SA ONSTACK
                  use alternate stack (see sigaltstack)
SA RESTART
                  restart system calls if possible
                  allow signal to interrupt itself
SA NOMASK
SA SIGINFO
                  Send@Steven htmettahZhighlew Kalbarczyk.
                                                    ECE391
```

Signal Behavior Control API

```
int sigaction (int signum, struct sigaction* newact,
                    struct sigaction* oldact);
                   read/replace signal behavior
                                                 Handler values when setting/reading
                                                 signal behavior
                                                     SIG IGN (ignore)
                                                     SIG_DFL (default behavior)
                                                     pointer to function
int sigprocmask (int how, sigset t* set, sigset_t* oldset);
                   set bit vector of blocked signals
                                          sigprocmask "how" options
                                              SIG BLOCK
                                                              add to mask
                                              SIG UNBLOCK
                                                              remove from mask
                                              SIG SETMASK
                                                              set mask
```

Signal Behavior Control API

```
int sigpending (sigset_t* set);
```

read bit vector of pending signals (these signals must be blocked/masked or they'd have been delivered already)

```
int sigsuspend (sigset t* mask);
```

temporarily set mask, then stop process until signal arrives

Kernel-side Data Structures for Signals asm/signal.h

- k_sigaction (a wrapper for sigaction)
- kernel sigaction
 - slightly different than user-level
 - puts mask last for extensibility

sigaction
k sigaction

sighand_struct struct (linux/sched.h)

```
count of processes using this structure (can be shared)

k_sigaction[64]

array of 64 signal behaviors (one per signal)
```

Kernel-side Data Structures for Signals

 Kernel must also track blocked signals, pending signals, etc.

Several pieces of task state pertaining to signals;
 see the following files

linux/sched.hlinux/signalfd.h

linux/signal.h

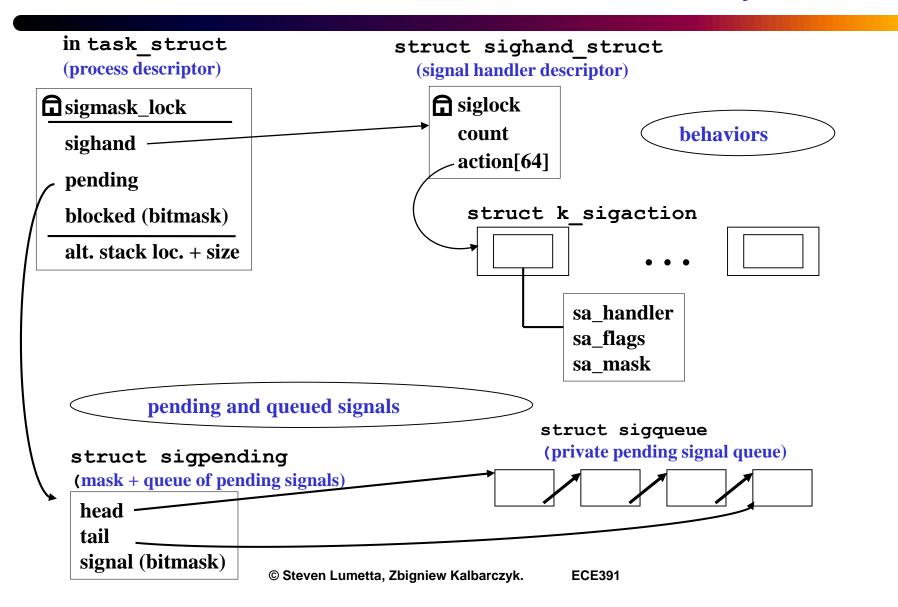
asm/signal.hasm-generic/siginfo.h

- kernel/signal.c
- arch/i386/kernel/signal.c

Task State Pertaining to Signals

```
int sigpending;
                           1 if any signal is pending (set to 0 if
                           blocked; recalculated if mask changes)
spinlock t sigmask lock; (protects indented fields)
struct sighand struct* sighand;
                                            signal behaviors (can
                                            be shared)
sigset t blocked;
                                     mask of blocked signals
struct sigpending pending;
                                     mask + queue of pending signals
/* note that these are not protected by sigmask lock */
unsigned long sas ss sp;
                                   alternate stack base pointer
                                   alternate stack size
size t sas ss size;
```

Signal-related Data Maintained by Kernel



Signal-related data maintained by kernel

Bit vectors use sigset_t; - unsigned long sig[2]; (64 bits) sigpending structure linked list of queued signals sigqueue* head; with tail pointer for fast append sigqueue* tail; mask of all pending signals sigset t signal; (non-RT appear here only; RT also in queue) sigqueue structure link to next queued signal sigqueue* next; siginfo t info; info for delivery to handle

Outside kernel (user-level interface)

```
int kill (pid t pid, int signal);
```

- arguments are target process and signal #
- returns 0 on success, -1 on failure
- Inside kernel (linux/sched.h)

- arguments are signal #, info structure pointer, and target task
- returns 0 on success, negative value on failure
- info structure pointer can be
 - an actual pointer
 - (void*)0 for traditional signals sent from users
 - (void*)1 for traditional signals sent from within the kernel
 - (void*)2 for forced signals (ignore masking)
- wrapper function for traditional signals
 - int send_sig (int sig, task_t* t, int priv);

Second internal interface to force signal generation

- what does "force" mean?
 - forcibly reset an ignored signal to default behavior
 - unblock the signal
 - then call send_sig_info
- forced signals are always privileged (only generated in kernel)
- force_sig is a wrapper function

Signal Generation – Question?

- What's the point of forcing signals?
- Why allow a program to block an asynchronous event and then override the block?

Answer

- used to deliver exceptions, for example
- What would happen if user program blocked signal from an exception?
 - program can't execute next instruction (it causes an exception)
 - kernel can't deliver signal (signal blocked)
 - deadlock!
- Instead of just letting the program hang, try to do something slightly more useful.

- Permission check for signal generation
 - sysadmin (or kernel, or privileged call) can always send
 - process with same user id can always send
 - process with same login session can send SIGCONT
- If generated signal is not being ignored
 - it is added to pending signals
 - a sleeping recipient is then woken up (kicked out of wait queues, for example)

Signal Delivery

- When are signals delivered? (see entry.S)
 - check sigpending (in task structure) when returning from
 - any interrupt
 - any exception
 - any system call
 - only deliver to currently executing process

Signal Delivery

- Action taken on delivery depends on current sigaction for the signal
 - defaults handled by kernel directly
 - handlers are executed by
 - adding a new stack frame to the user stack
 - transferring machine state context to the user stack
 - returning to user level

User Stack after signal is delivered

return address
signal number
siginfo pointer
hardware context pointer
siginfo structure
hardware context
execute sigreturn (sys call)
previous
stack

User Stack after Signal is Delivered

- Why bother copying h/w context (registers, etc.) from kernel stack?
 - switch threads in signal handler by swapping context with that on stack
 - avoid kernel stack overflow due to malicious programs
 - allow user to modify context

User Stack after Signal is Delivered

- Note: in Linux 2.6
 - sigreturn code still written to stack, but not used on most systems
 - instead, process has one copy of sigreturn system call in its address space
 - rather than creating a copy for each delivery
 - avoids stack execution, which is pathway for buffer overrun attacks

Returning from Signal Handlers

- On call to sigreturn
 - dump stack frame
 - check if signal was delivered during system call
 - if so, checks whether system call should be restarted
 - if not, returns –EINTR
 - if so, sets EAX to previous value (sys call #) and changes PC to re-execute INT x80 instruction
 - if not, simply continue the process
- What happens if sigreturn not called (possible security hole?)
 - no state left on kernel stack, so has no impact on kernel
 - can only affect user-level stack for that process