ECE391
Computer System Engineering

Lecture 24

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Spring 2021

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 software (kernel or 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 (Portable Operating System Interface) model (as opposed to traditional model)
    - new real-time signals are queued (one invocation per signal sent)
    - can be sent to threads or processes
      (latter delivered to any thread in a process)
    - *siginfo* structure contains additional information about the signal (e.g., address of exception)

List of Signals and Behaviors

name	default 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 SIGIOT	dump core	abort signal from `abort()` IOT trap; a synonym for SIGABRT
SIGBUS	dump core	bus error (bad memory access)
SIGFPE	dump core	floating point exception
SIGKILL	terminate (always)	kill signal
SIGUSR1	terminate	user-defined signal 1
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 SIGCLD	ignore	child stopped or terminated synonym for SIGCHLD
SIGCONT	continue	continue if stopped
SIGSTOP	stop (always)	stop process
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 SIGPOLL	terminate	I/O now possible pollable event; synonym for SIGIO
SIGPWR	terminate	power failure
SIGSYS	dump core	bad system call

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

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

traditional

POSIX

don't use/set both!

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
SA_NOMASK	allow signal to interrupt itself
SA_SIGINFO	send more info to handler

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)

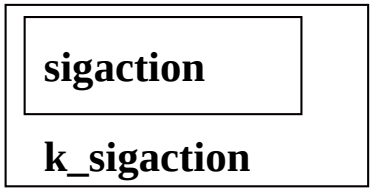
```
int sigsuspend (sigset_t* mask);
```

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



- **`sighand_struct`** struct (linux/sched.h)

  count — **count of processes using this structure (can be shared)**

  k_sigaction[64] — **array of 64 signal behaviors (one per signal)**

  spin lock

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.h
  - linux/signal.h
  - asm/signal.h
  - kernel/signal.c
  - arch/i386/kernel/signal.c
  - linux/signalfd.h
  - asm-generic/siginfo.h

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

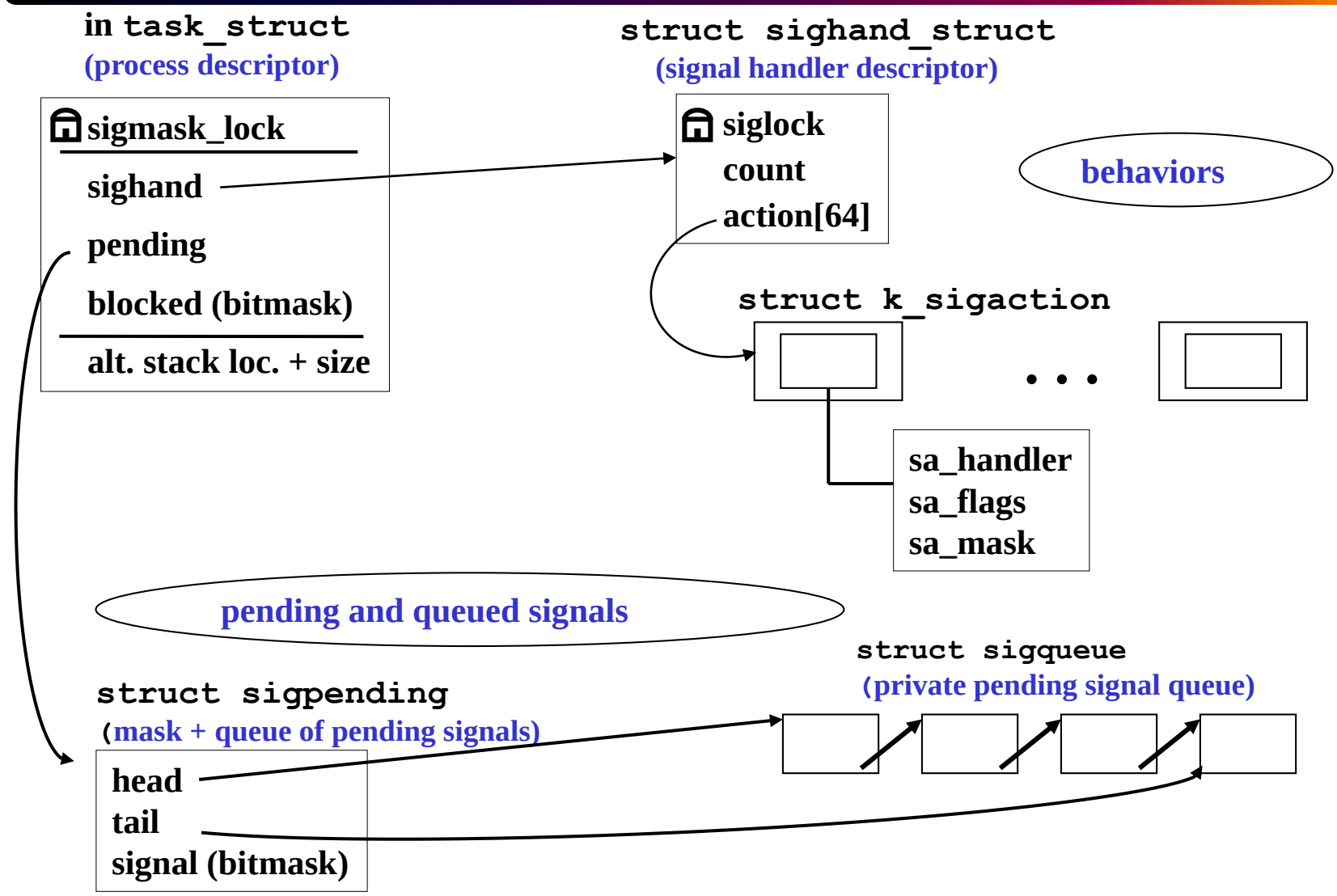
```
struct sigpending pending;
```
mask + queue of pending signals

/* note that these are not protected by **`sigmask_lock`** */

```
unsigned long sas_ss_sp;
```
alternate stack base pointer

```
size_t sas_ss_size;
```
alternate stack 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

  `sigqueue* head;` linked list of queued signals

  `sigqueue* tail;` with tail pointer for fast append

  `sigset_t signal;` mask of all pending signals

  (non-RT appear here only; RT also in queue)
- **`sigqueue`** structure

  `sigqueue* next;` link to next queued signal

  `siginfo_t info;` info for delivery to handle

Signal Generation

- 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)

```
int send_sig_info (int sig, struct
                   siginfo* info, task_t* t);
```

Signal Generation

- 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);`**

Signal Generation

- Second internal interface to force signal generation

```
int force_sig_info (int sig,
                    struct siginfo* info, task_t* t);

void force_sig (int sig, task_t* t);
```

  - 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.

Signal Generation

- 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)

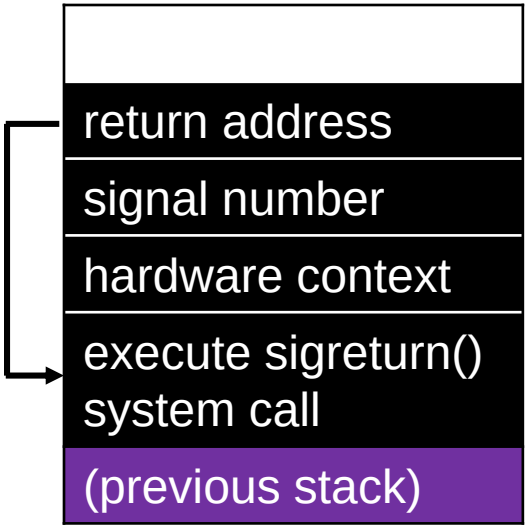
Signal Delivery

- 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

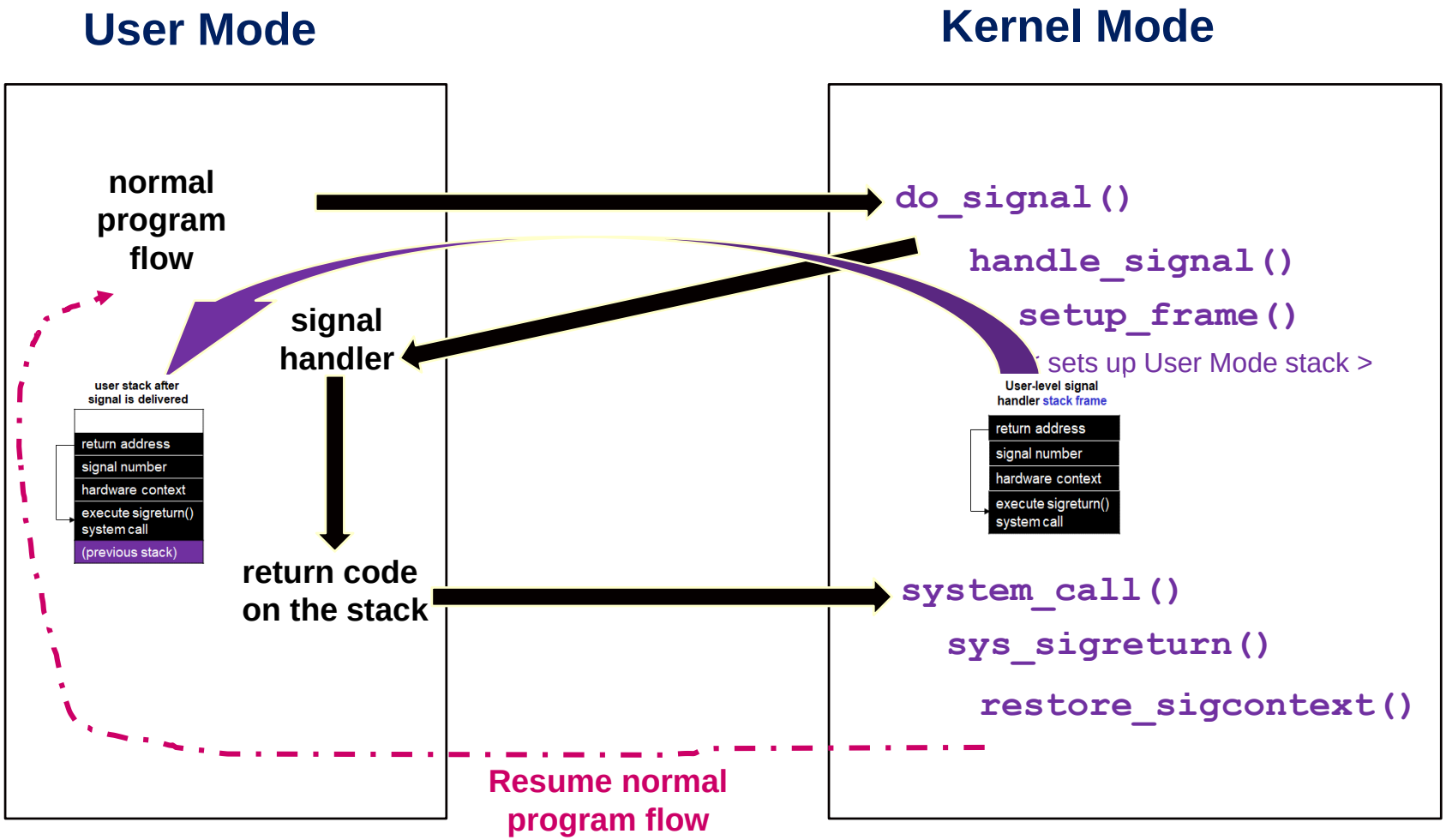
Signal Delivery

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

User stack after signal is delivered



Catching the Signal: Execute Program Defined Handler



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