5/3/24

Signals & User level Threads

Signal: a type of IPC, also known as software exceptions

a pre-defined set of events that processes can use to communicate.

Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from abort(3)
SIGALRM	P1990	Term	Timer signal from alarm(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	-	Ign	A synonym for SIGCHLD
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	-	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	-		A synonym for SIGPWR
SIGINT	P1990	Term	Interrupt from keyboard
SIGIO	-	Term	I/O now possible (4.2BSD)
SIGIOT	-	Core	IOT trap. A synonym for SIGABRT
SIGKILL	P1990	Term	Kill signal
SIGLOST	-	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGPOLL	P2001	Term	Pollable event (Sys V); synonym for SIGIO
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	-	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4); see also seccomp(2)
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGUNUSED	-	Core	Synonymous with SIGSYS
SIGURG	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	P2001	Core	CPU time limit exceeded (4.2BSD); see setrlimit(2)
SIGXFSZ	P2001	Core	File size limit exceeded (4.2BSD); see setrlimit(2)
SIGWINCH	-	Ign	Window resize signal (4.3BSD, Sun)

Sending Signals

- → syscall: kill(pid, signal#)
- → kernel also forwards some exceptions
  - → SIGSEGV, SIGFPE, SIGILL
  - → application might be able to handle some & recover from these faults
- → how to implement this?
  - → track a pending set of signals per process
  - → enforce some sender/receiver restriction
    - → processes from the same user can send signals to each other freely
    - → privileged process (root)

Receiving Signals

- signal delivery is implicit to the receiver (no action required)
  - pending signals tracked as a set, multiple sends of the same signal result in a single delivery; once the signal is handled, can be delivered again

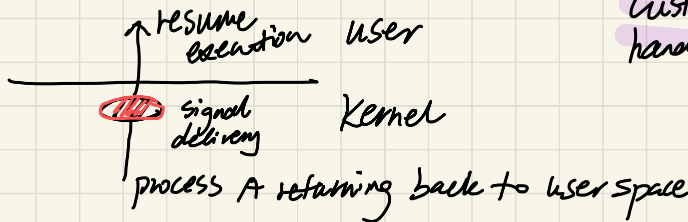
upon a signal delivery, if no custom handler, execute default action, otherwise, execute the custom handler

- Kernel defines default actions for all signals
- a process can define custom handlers for most signals (except SIGSTOP & SIGKILL)

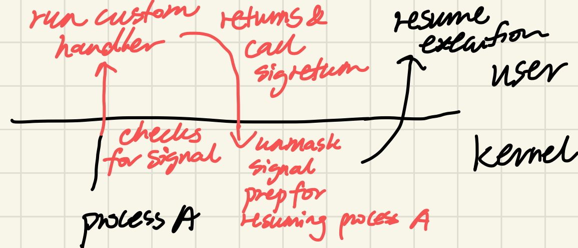
  via syscall signal(sig#, handler_func) → lives in user memory.

- How to implement signal delivery?
  - a process must be in the kernel for kernel to deliver a signal
  - deliver a signal while a process is in the kernel for whatever reason: context switch, syscall, interrupts, exceptions

Default action



Custom handler



More on signal delivery

- a process can choose to block until a specific signal is delivered
  - helpful for synchronizing states across processes (SIGUSR1, SIGUSR2)
- a process can mask certain signals, preventing them from being delivered.
- custom signal handlers are fully defined by the process, meaning they don't have to return (can jump out of the execution)

User Level Threads

→ So far we've only seen kernel level threads

managed & scheduled by the kernel, → TCB, kernel stack
context switch requires mode switch

→ threads are good for abstracting independent tasks

→ but often the task granularity doesn't justify the cost of creating & scheduling a kernel level thread ⁈

→ User level threads

managed & scheduled by user libraries & runtime
kernel is unaware of their existence

→ mid end laptop can run 50-60 million user level threads.