Kernel and processes

Signals

Signals/ Events

- Process interact with kernel using system calls
- In computer system special events require the kernel to interact with processes
- In UNIX/LINUX these events are called signals
  - Exceptions
    - illegal memory access, division by zero, …
  - User generated
    - process abortion , Ctrl-C
  - generated by system calls
    - son process termination, timer, kill

How to handle signals

- Polling
  - All processes poll the kernel to existing events
  - Not efficient
  - Not correct
    - programmer should implement this “system”feature”
- kernel can manage notification (and scheduling)
  - of processes requesting notification

Upcall: User-level event delivery

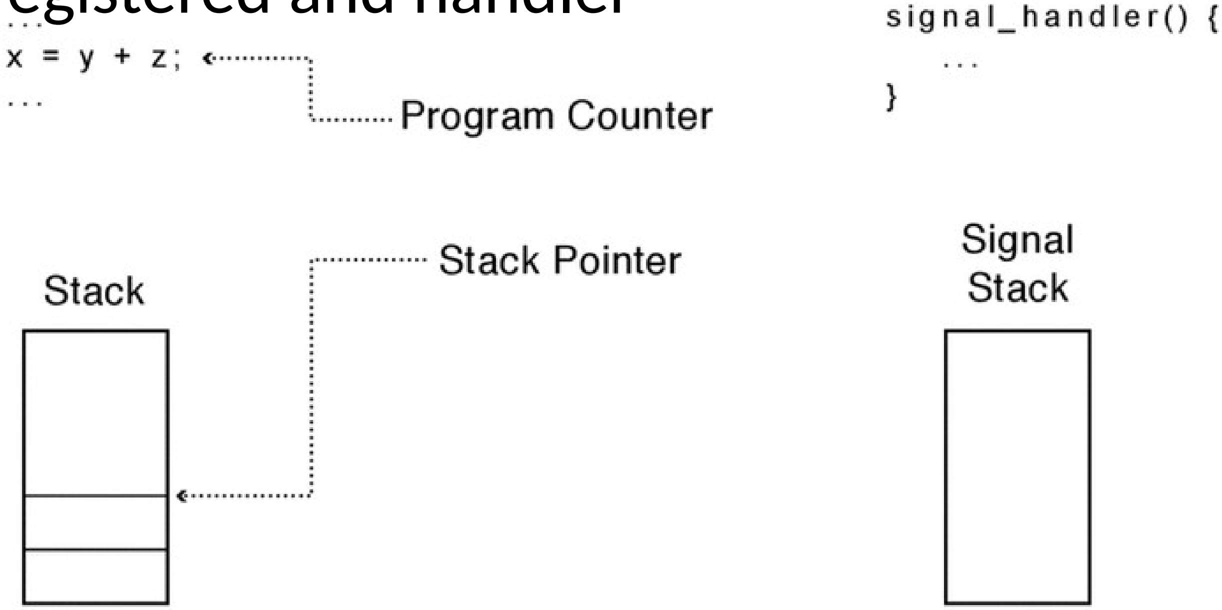
- Notify user process of some event that needs to be handled right away
  - Time expiration
    - Real-time user interface
    - Time-slice for user-level thread manager
  - Interrupt delivery for VM player
  - Asynchronous I/O completion (async/await)
- UNIX signal

Upcalls vs Interrupts

- Signal handlers = interrupt vector
- Signal stack = interrupt stack
- Automatic save/restore registers = transparent resume
- Signal masking: signals disabled while in signal handler

Upcall: Before

- Process previously register a signal handler
  - will execute in user level
  - kernel will redirect execution to it
- Kernel was notified of the event
  - verifies that process registered and handler

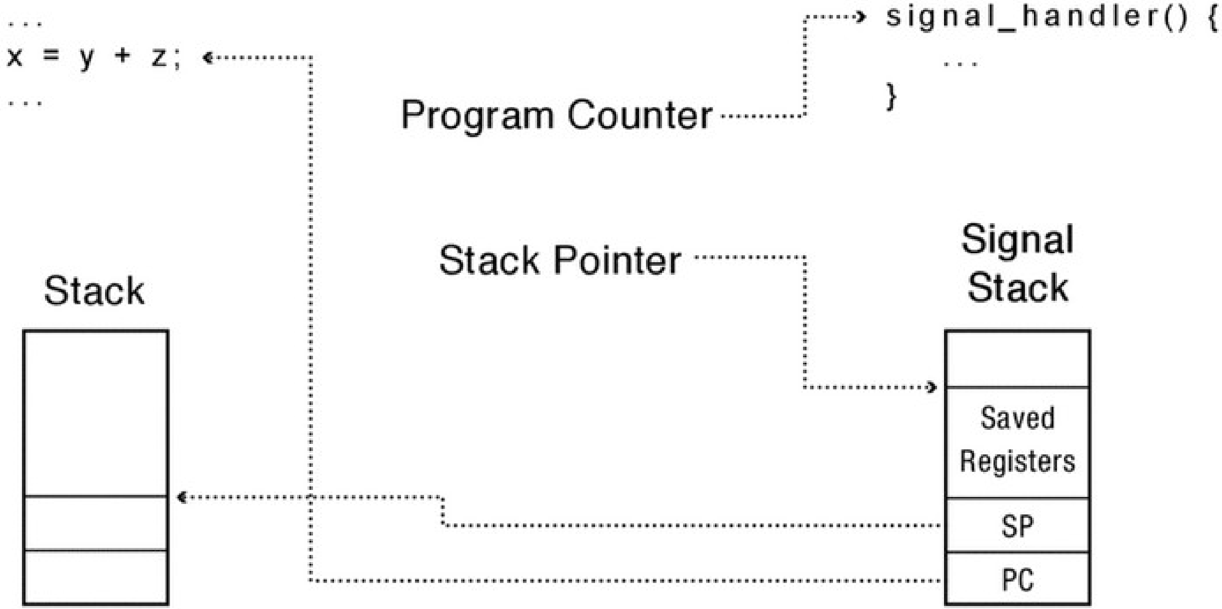


```
...
x = y + z;
...
```

```
signal_handler() {
    ...
}
```

Upcall: During

- In kernel mode
  - created signal stack
    - correct signal return
  - updated SP and PC
    - for process to execute handler
- In user mode
  - execution of handler
  - Return to resume



Signal handling in UNIX

- Default handling
  - Kernel handles the signal
  - Process is not notified
    - Most of times it is terminated
- Ignore the signal
- handle the event with a user level handler
- Two signals can not be handled or ignored:
  - SIGKILL
  - SIGSTOP

Unix signals

Signal	Code	Default action	Cause
SIGHUP	1	Terminates process	Terminal line hangup
SIGINT	**2**	**Terminates process**	**interrupt program (CTRL-C)**
SIGQUIT	3	Terminates process + core dumps	Quit program
SIGKILL	**9**	**Terminates process**	**Kill program**
SIGALRM	14	Terminates process	Timer expired
SIGCHLD	**20**	**Ignored**	**child status has changed**
SIGSTOP	17	Stop process	Stop signal generated from keyboard (CTRL-Z)
SIGSCONT	19	Continue if stoped	Continue after stop
SIGUSR1	31	Terminates process + core dumps	User defined signal 1

Unix signals

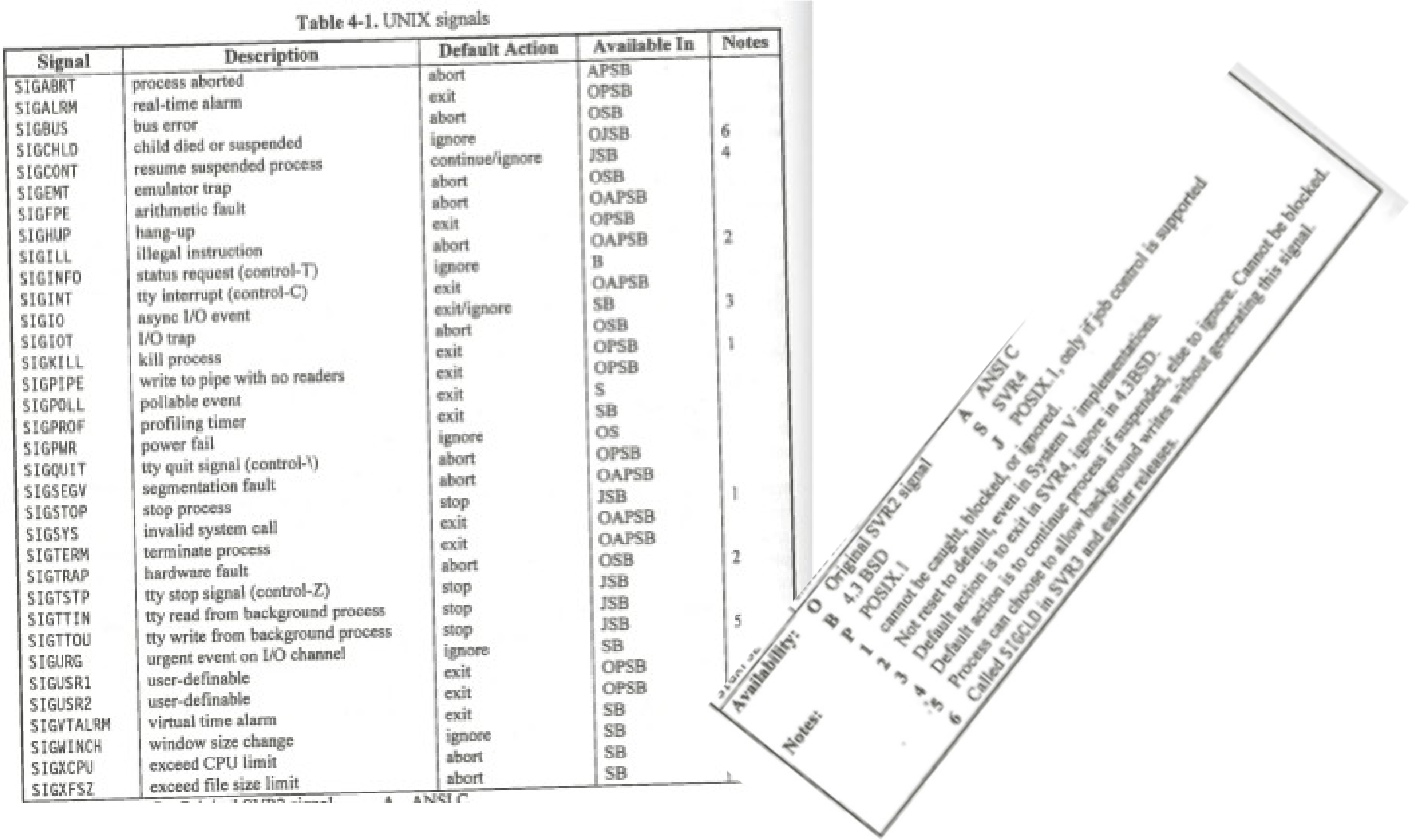


Table 4-1. UNIX signals

Signal	Description	Default Action	Available In	Notes
SIGABRT	process aborted	abort	APSB	
SIGALRM	real-time alarm	exit	OPSB	
SIGBUS	bus error	abort	OSB	
SIGCHLD	child died or suspended	ignore	OJSB	6
SIGCONT	resume suspended process	continue/ignore	JSB	4
SIGEMT	emulator trap	abort	OSB	
SIGFPE	arithmetic fault	abort	OAPSB	
SIGHUP	hang-up	exit	OPSB	
SIGILL	illegal instruction	abort	OAPSB	2
SIGINFO	status request (control-T)	ignore	B	
SIGINT	tty interrupt (control-C)	exit	OAPSB	
SIGIO	async I/O event	exit/ignore	SB	3
SIGIOT	I/O trap	abort	OSB	
SIGKILL	kill process	exit	OPSB	1
SIGPIPE	write to pipe with no readers	exit	OPSB	
SIGPOLL	pollable event	exit	S	
SIGPROF	profiling timer	exit	SB	
SIGPWR	power fail	ignore	OS	
SIGQUIT	tty quit signal (control-\\)	abort	OPSB	
SIGSEGV	segmentation fault	abort	OAPSB	
SIGSTOP	stop process	stop	JSB	1
SIGSYS	invalid system call	exit	OAPSB	
SIGTERM	terminate process	exit	OAPSB	
SIGTRAP	hardware fault	abort	OSB	2
SIGTSTP	tty stop signal (control-Z)	stop	JSB	
SIGTTIN	tty read from background process	stop	JSB	
SIGTTOU	tty write from background process	stop	JSB	5
SIGURG	urgent event on I/O channel	ignore	SB	
SIGUSR1	user-definable	exit	OPSB	
SIGUSR2	user-definable	exit	OPSB	
SIGVTALRM	virtual time alarm	exit	SB	
SIGWINCH	window size change	ignore	SB	
SIGXCPU	exceed CPU limit	abort	SB	
SIGXFSZ	exceed file size limit	abort	SB	

Availability:	O	Original SVR2 signal	A	ANSI C
	B	4.3 BSD	S	SVR4
	P	POSIX.1	J	POSIX.1, only if job control is supported

Notes:
1 cannot be caught, blocked, or ignored.
2 Not reset to default, even in System V implementations.
3 Default action is to exit in SVR4, ignore in 4.3BSD.
4 Default action is to continue process if suspended, else to ignore. Cannot be blocked.
5 Process can choose to allow background writes without generating this signal.
6 Called SIGCLD in SVR3 and earlier releases.

Signal Generation

- Exceptions
  - When an exception occurs in the process (for instance, an attempt to execute an illegal instruction), the kernel notifies the process by sending it a signal.
- Other processes
  - A process may send a signal to another process, or set of processes, through the **kill** or **sigsend** system calls.
  - A process may even send a signal to itself.
- Terminal Interrupts
  - Certain keyboard characters, such as control-C or control-\, send signals to the foreground process on that terminal.
  - The **stty** command allows the user to bind each terminal-generated signa! to a specific key.
- Job control
  - Background processes that try to read or write to the terminal are sent job control signals.
  - Job control shells such as csh and ksh use signals to manipulate foreground and background processes.
  - When a process terminates or is suspended, the kernel notifies its parent via a signal.

Signal Generation

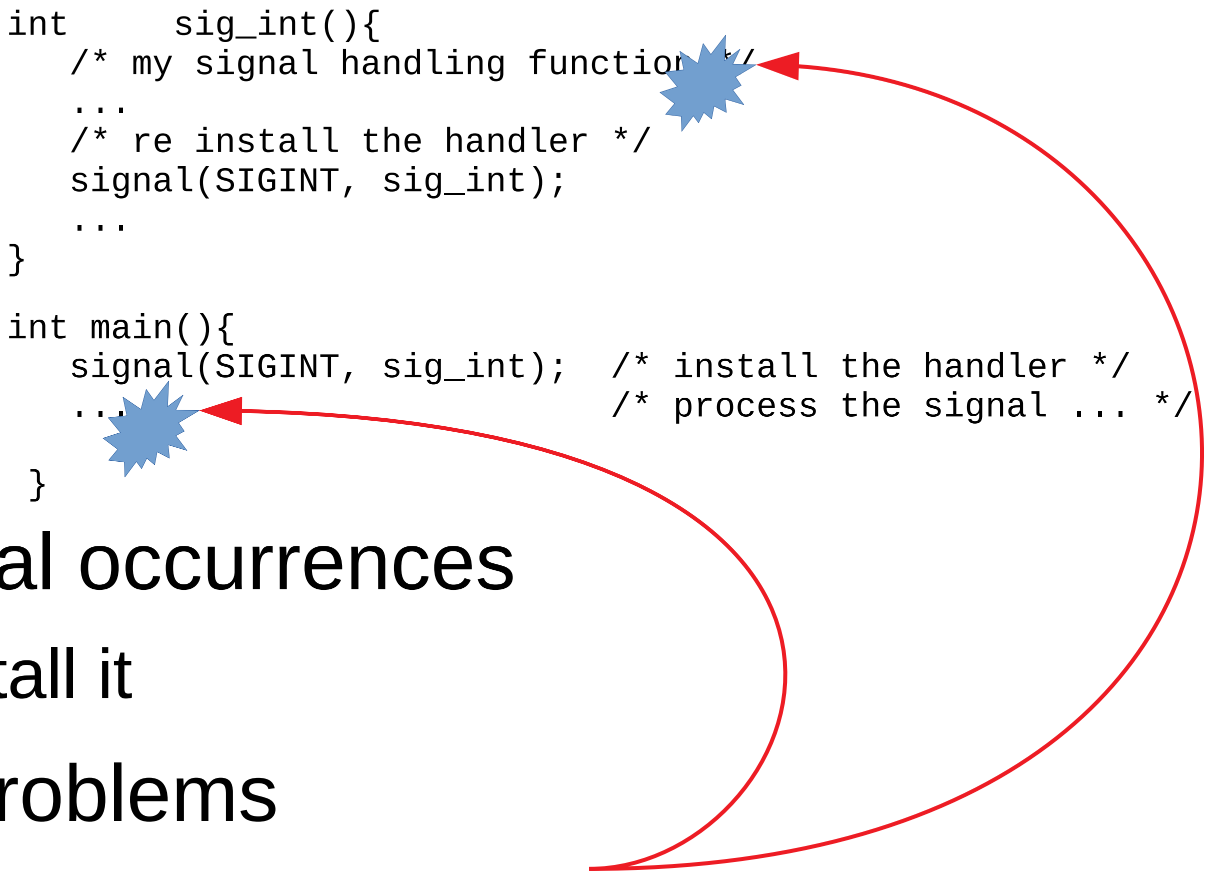
- Quotas
  - When a process exceeds its CPU or file size limits, the kernel sends a signal to the process.
- Notifications
  - A process may request notification of certain events, such as a de- vice being ready for I/O.
  - The kernel informs the process via a signal.
- Alarms
  - A process may set an alarm for a certain time;
  - when it expires, the kernel notifies the process through a signal

Signal handler

- `void (*sighandler_t)(int);`
- Registration of signal handler
  - `#include <signal.h>`
  - `sighandler_t signal(int, sighandler_t);`
  - 1st parameter
    - signal code (SIGXXX)
  - 2nd parameter
    - poiter to function
    - or SIG_IGN
    - or SIG_DFL
- Returns the previous signal handler
- Different OS implement signals in different ways
  - signal handler should be re-assigned

Unreliable Signais

- Bigeste concern
  - Signal delivery



```
int     sig_int(){
   /* my signal handling function */
   ...
   /* re install the handler */
   signal(SIGINT, sig_int);
   ...
}

int main(){
   signal(SIGINT, sig_int);  /* install the handler */
   ...                        /* process the signal ... */

 }
```

- To catch new signal occurrences
  - Users must re-install it
- Synchronization problems
  - Signal generated in fast succession?
- No mechanisms to temporary block a signal….

Reliable signals

- Solves previou sproblems
  - BSD + POSIX
- Persistent handlers
  - Signal handlers remain installed
  - No windows between signal catch and re-install
- Masking
  - A signal can be masked/blocked temporarly
    - Kernel keeps signal until it is unblocked
- Unblock and wait
  - A process can be blocked until it receives a signal

Signal handler

- int sigaction(int, const struct sigaction * newhandler, struct sigaction * old);
  - System call to configure system handling
  - Same behavior in all systems
  - 1st parameter : specifies a signal number.
  - 2nd parameter: new handler
  - 3rd parameter: old handler
- Signal handler configuration - struct sigaction
  - void (*sa_handler)()
    - signal handler function
  - void (*sa_sigaction)(int signum, siginfo_t *siginfo, void *uctx);
    - alternative signalnal handler function
  - sigset_t sa_mask
    - signals that shpuld be blocked during signal handler
  - int as_flags
    - signal handling configuration
  - void (*sa_restorer)();
-

User Generated signals

- int kill(pid_t pid, int sig);
  - #include <sys/types.h>
  - #include <signal.h>
  - 1st parameter
    - pid of process to receive signal
    - -1 all processes
  - 2nd parameter
    - Signal number
  - Returns 0 in case of success
    - -1

Signal Management API

- kill
  - sends signal to process
- typedef void (*sighandler_t)(int);
  - Signal handler
- Signal() / sigaction()
  - sets the disposition of the a signal a handler
  - Can also disable or reset
- If signal is not handled will probably terminate process