Signals
(POSIX systems)

Patrício R. Domingues

Departamento de Eng Informática

ESTG/IPLeiria

POINTER TO FUNCTIONS

Pointer to functions (1)

- The C language has the concept of *pointer to a function*
  - The pointer holds the address of a function
  - The function can be called through the pointer
    - Call syntax is similar to a regular function call
- What's the datatype of a pointer to a function?
  - Datatype is function
  - A function has a signature defined by the following elements
    - Datatype of the return value
    - Datatype of the parameters (if any)

Pointer to functions (2)

- Examples of signatures of functions
  - `int F1(int a, int b);`
  - `int F2(int y, int z);`
    - F1 and F2 have the same signature (the name of the parameters is irrelevant)
  - `double F3 (int a, int b);`
    - F3 has a signature different from F1 (and obviously from F2)
  - `int *F4(int a, int b);`
    - F4 has a signature different from F1, F2 and F3

Pointer to functions (3)

- Declaration of a pointer to a function
  - Pointer points to the signature
- Example
  - int (***PtrF1**)(int , int);
    - PtrF1 can point to functions with signature "**int ... (int, int);**"
  - double (***PtrF2**)(double ,char *);
    - PtrF2 can point to functions with signature "**double ... (double,char*);**"

Pointer to functions (4)

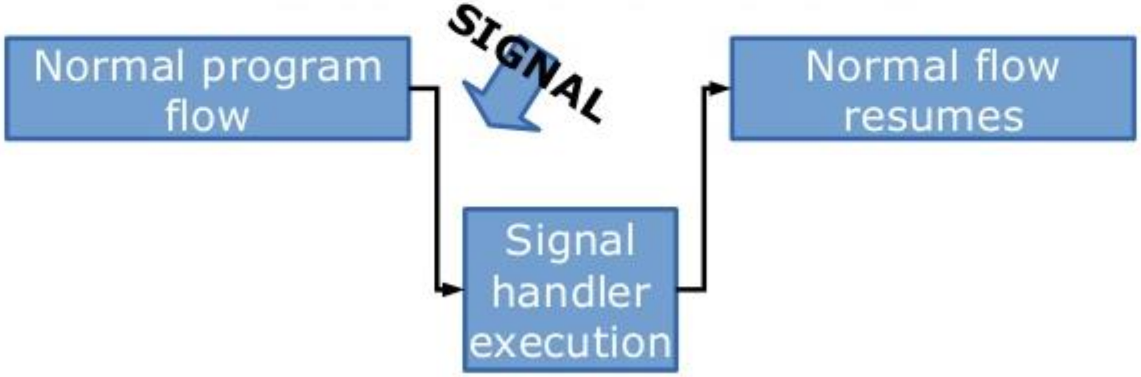
- How to get the address of a function?
  - simple
    - The name of the function holds the address of the function
    - Thus, the pointer only needs to be assigned the name of the function
    - This can also be done through “**&FunctionName**”
- Example

```
int F1(int a, int b){
    return a + b;
    }
int (*PtrF1)(int , int) = NULL;   /*PtrF1 declaration*/
int Result;
PtrF1 = F1;             /* PtrF1 points to the F1 function */
Result = PtrF1(10,30);       /* Call of F1 through PtrF1 */
Result = (*PtrF1)(10,30);  /* Same as previous line of code */
```

SIGNALS

What’s a *signal*? (1)

- Signal is an asynchronous notification delivered to a process
  - – Notification
    - Related to an event
      - – Example:



https://bit.ly/2wLTTca

- » control + C within the shell has the effect of delivering a signal (SIGINT) to the running process
- » Process tries to access an invalid memory address. SO delivers the SIGSEGV (segmentation violation) to the process.

  - – Asynchronous
    - The signal can happen at any time
      - – Therefore, the notification can be delivered at any time...

What’s a *signal*? (2)

- In UNIX, a signal is represented by an integer number
- Each signal has also a symbolic name
  - Easier for us humans to remember
  - Examples: SIGINT, SIGHUP, SIGSEGV, SIGPIPE, etc.
- There are around 40 different signals
  - man 7 signal
  - kill -l signal
    - List the available signals

JULIA EVANS
@b0rk

signals

drawings.jvns.ca

If you've ever used kill you've used signals

the Linux kernel sends your process signals in lots of situations

- your child terminated
- that pipe is closed
- illegal instruction
- the timer you set expired
- Segmentation fault

you can send signals yourself with the kill system call or command

SIGINT	Ctrl-C	various levels of "die"
SIGTERM	kill	
SIGKILL	kill -9	

SIGHUP kill -HUP

↑ often interpreted as "reload config", eg by nginx

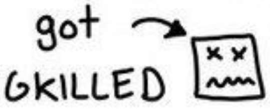
Every signal has a default action, one of:

- ignore
- kill process
- kill process AND make core dump file
- stop process
- resume process

Your program can set custom handlers for almost any signal

exceptions:

SIGSTOP & SIGKILL can't be ignored



signals can be hard to handle correctly since they can happen at ANY time

Sources of signals (#1)

- Signals come from many sources. Some possible sources:
  - Something executing the system call (“`kill(…)`”) that transmit a signal to a process
  - Sending a signal from the process unto itself
  - When a child process exits
  - When the parent process dies or *hangup*
  - When the program behaves incorrectly
  - Hardware failure

Source of signals

- Which elements can send a signal to a process?

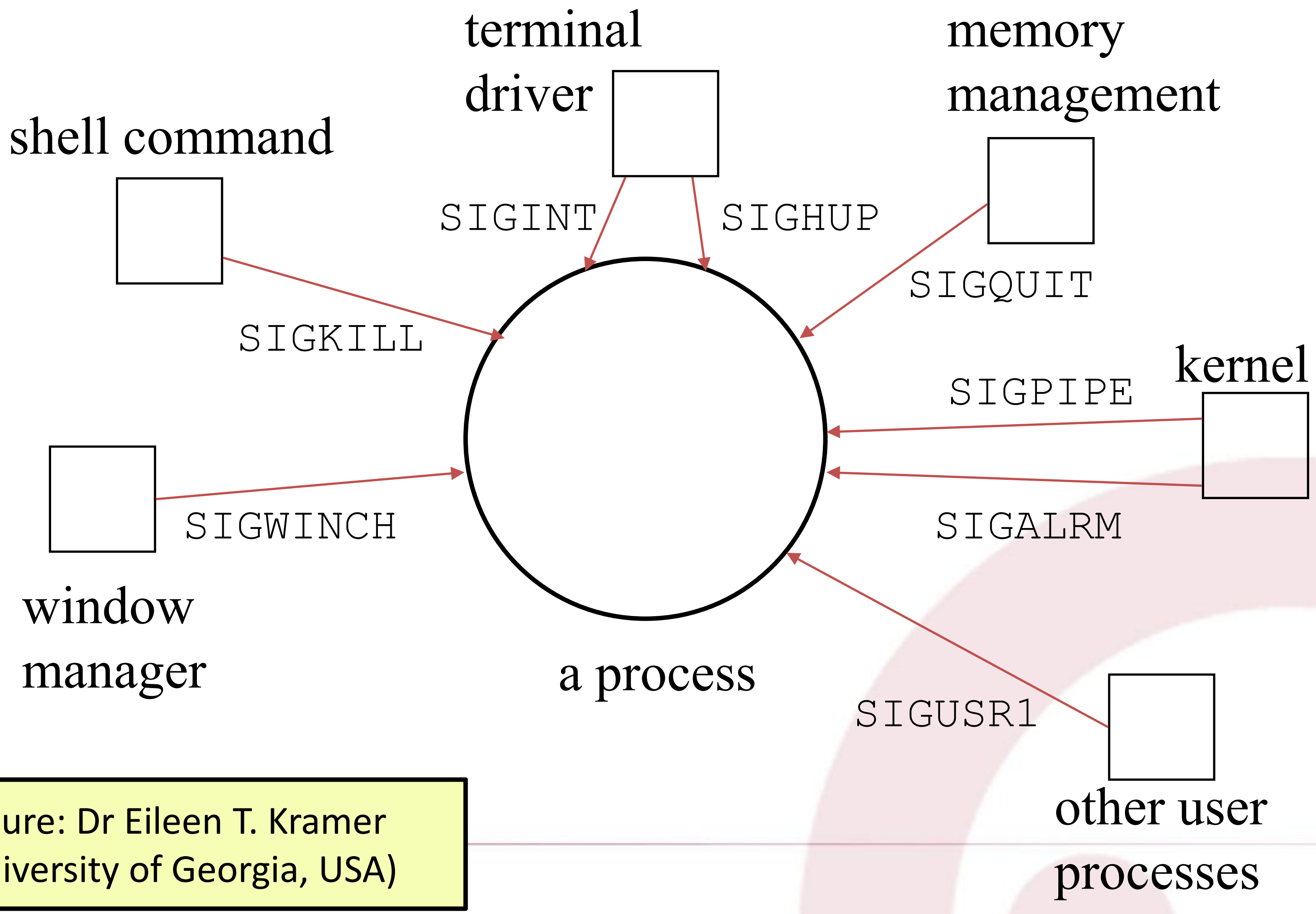


Figure: Dr Eileen T. Kramer
(University of Georgia, USA)

The *kill* command (1)

- kill command (tries to) deliver a signal to a destination process
  - Syntax
    - `kill [options]PID1 PID2  PID3...`
    - Send *signal* to processes with PID1, PID2, PID3, ...
  - Why the name *kill*?
    - By default, when a process receives a signal, it terminates
    - A process can be configured to have a different behavior
      - *Handling* the signal
  - More on the command kill
    - `man kill`

The *kill* command (2)

- Examples
  - `kill -SIGINT 1234`
    - Send the signal SIGINT to process with PID 1234
  - `kill -9 1234`
    - Send the signal 9 (**SIGKILL**) to processes 1234
  - `kill -SIGKILL 1234`
    - Same as above
  - `kill -kill $$`
    - Kills the shell
    - $$ is the PID of the shell

SIGKILL and SIGSTOP are the only signal that cannot be captured
(more on this later)

Sending a signal

- A regular usar can only send a signal to processes that she/he owns
  - kill -KILL 1

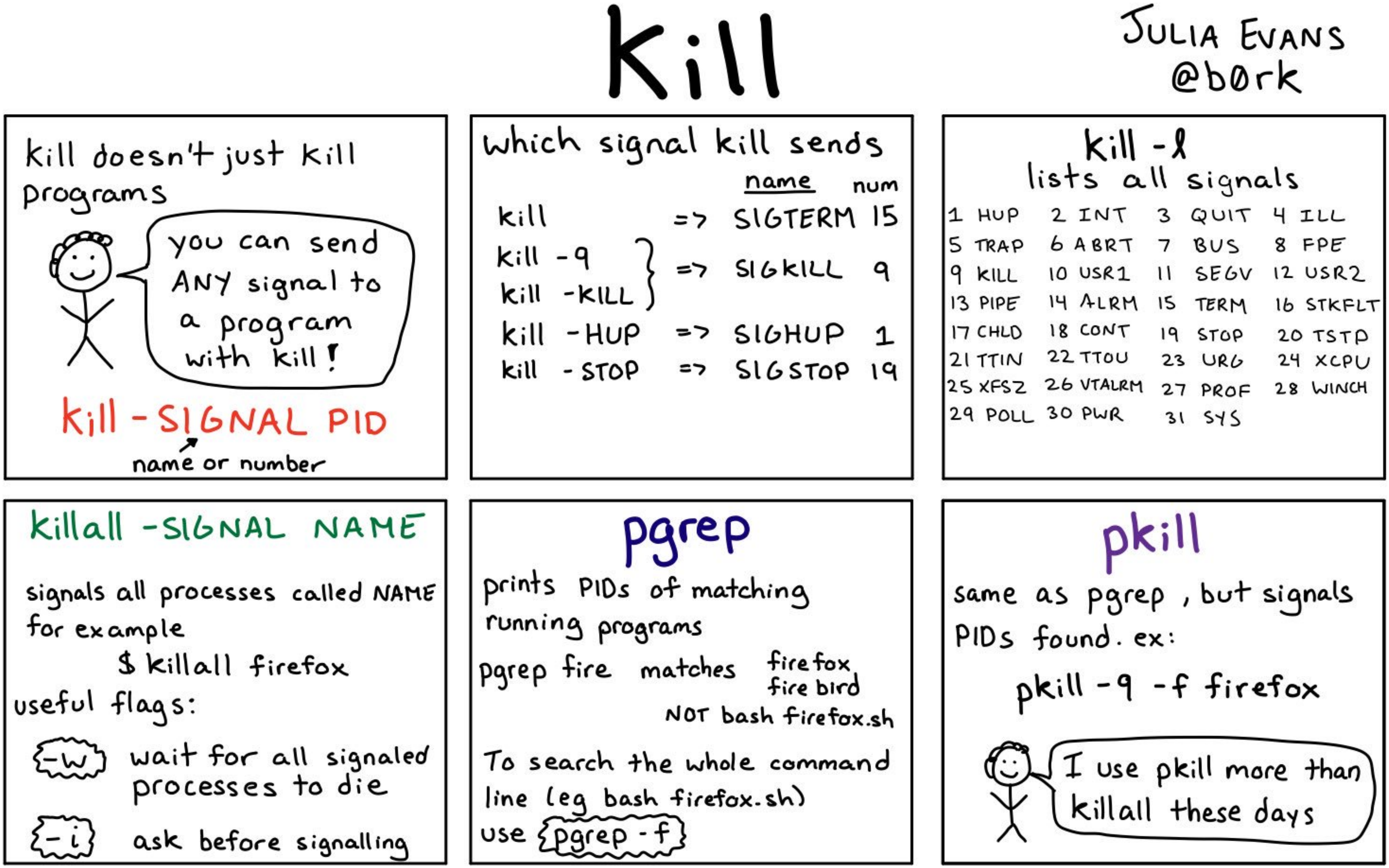
```
bash: kill: (1) - Operation not permitted
```

  - Process with PID 1 is a system process (process *init*)
- A root account can send a signal to **any** process

The *killall* command

- Killall is a variant of the kill command
- `killall [options] name`
  - It sends the signal to all processes whose name is *name`

- `killall -INT bash`
  - *(tries) to send the SIGINT signal to all processes whose name is bash*
  - For a regular user, only the bash processes that belong to the user receive the signal



kill

JULIA EVANS
@b0rk

kill doesn't just kill programs

you can send ANY signal to a program with kill!

kill -SIGNAL PID

name or number

which signal kill sends

		name	num
kill	=>	SIGTERM	15
kill -9 kill -KILL	=>	SIGKILL	9
kill -HUP	=>	SIGHUP	1
kill -STOP	=>	SIGSTOP	19

kill -l

lists all signals

1 HUP	2 INT	3 QUIT	4 ILL
5 TRAP	6 ABRT	7 BUS	8 FPE
9 KILL	10 USR1	11 SEGV	12 USR2
13 PIPE	14 ALRM	15 TERM	16 STKFLT
17 CHLD	18 CONT	19 STOP	20 TSTP
21 TTIN	22 TTOU	23 URG	24 XCPU
25 XFSZ	26 VTALRM	27 PROF	28 WINCH
29 POLL	30 PWR	31 SYS	

killall -SIGNAL NAME

signals all processes called NAME for example

```
$ killall firefox
```

useful flags:

- -w wait for all signaled processes to die
- -i ask before signalling

pgrep

prints PIDs of matching running programs

pgrep fire matches firefox, fire bird

NOT bash firefox.sh

To search the whole command line (eg bash firefox.sh) use pgrep -f

pkill

same as pgrep, but signals PIDs found. ex:

```
pkill -9 -f firefox
```

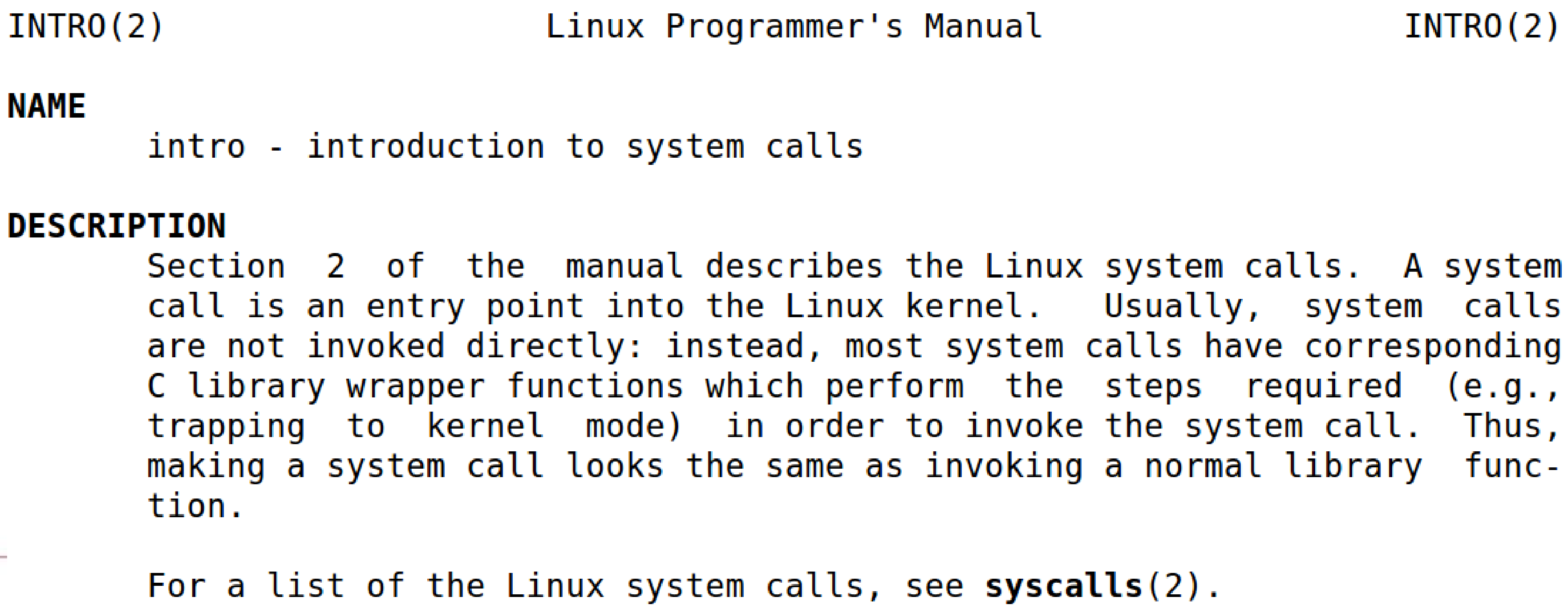
I use pkill more than killall these days

The kill() function (#1)

- How do we send a signal programmatically?
  - kill function
    - `int kill(pid_t pid, int sig);`
    - return -1 on error (setting *errno*), 0 on success
    - Interpretation of pid
      - *pid* > 0: send signal to process with PID *pid*
      - *pid* == 0: send signal to all processes whose group ID equals the sender
        - » Parent process sending a signal to all children
    - Interpretation of sig
      - Sig > 0: send signal sig
      - Sig==0: do not actually send a signal, but acts if has done so
        - » Use-case: checking if a given process exist

The kill() function (#2)

- `man 2 kill`
  - Need to specify the section 2 of the manual
  - Section 2 of man describes system calls
    - Kill, read, write, open…
  - man 2 intro



```
INTRO(2)                  Linux Programmer's Manual                  INTRO(2)

NAME
       intro - introduction to system calls

DESCRIPTION
       Section  2  of  the  manual describes the Linux system calls.  A system
       call is an entry point into the Linux kernel.   Usually,  system  calls
       are not invoked directly: instead, most system calls have corresponding
       C library wrapper functions which perform  the  steps  required  (e.g.,
       trapping  to  kernel  mode)  in order to invoke the system call.  Thus,
       making a system call looks the same as invoking a normal library  func-
       tion.

       For a list of the Linux system calls, see syscalls(2).
```

Signals sent by functions

- Some system functions send a signal
  - **`abort()`** sends the SIGABRT signal to the calling process
    - `void abort(void);`
  - **`alarm()`** schedule the delivery of the SIGALRM signal to the calling process
    - Appropriate for setting timeouts
    - `unsigned int alarm(unsigned int seconds);`

Responding to a *signal*

- For given signal, a process can be configured to...
  - Execute the default action
    - The default action for many (not all!) signals is to terminate the process
    - SIGKILL and SIGSTOP always results in their default actions
      - SIGKILL: terminates the process
      - SIGSTOP: stops the process (same as ctrl+Z on the shell)
  - Ignore the signal
    - Not available for SIGKILL and SIGSTOP
  - Launch a signal *handler*
    - *Signal handler:* function called whenever a signal is received

Setting a *signal handler*

- The `sigaction` library call
  - Installs a signal handler for a given signal

```
#include <signal.h>
int sigaction(int signum, const struct
    sigaction *act, struct sigaction *oldact);
```

- `signum`: signal number to be configured
- `act`: pointer to `sigaction struct` that contains the configuration to use
- `oldact`: sigaction struct filled with the previous configuration
- Question
  - Why `const struct sigaction *act` vs. `struct sigaction *oldact`?

struct sigaction (#1)

- The *struct sigaction*

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

sa_handler:

- pointer to the function that will handle the signal. The function should have one integer parameter and does not return anything
  - The integer parameter corresponds to the signal number that triggered the handler
- SIG_DFL to restore the default behavior
- SIG_IGN to set the process to ignore the signal

struct sigaction (#2)

- The *struct sigaction*

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

`sa_sigaction:`

- If `sa_flags` is `SA_SIGINFO`, `sa_sigaction` specifies the signal handling function
- This has improved functionalties for the signal handler function, namely a `siginfo_t` structure (see next slide)
- `sa_sigaction` is not compatible with `sa_handler`. Use one or the other, but not both!

struct sigaction (#3)

- The *struct siginfo_t*

```
 struct siginfo_t {
   int si_signo;    /* Signal number */
   int si_errno;    /* An errno value */
   int si_code;     /* Signal code */
   pid_t si_pid;      /* Sending process ID */
   uid_t si_uid;      /* Real user ID of sending process */
   int   si_status;   /* Exit value or signal */
   clock_t si_utime;    /* User time consumed */
   clock_t si_stime;    /* System time consumed */
(...)
}
```

- The `struct siginfo_t` returns a large number of data regarding the signal and the context of the call

struct sigaction (#4)

- The *struct sigaction*

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

`sa_mask:`

- Set of signal to be blocked during the call of a signal handler
  - Avoid race conditions
  - During the handler execution, the signal being processed by the handler is blocked by default
  - Regarding sisget_t , see **`man 3 sigsetops`**

struct sigaction (#5)

- The *struct sigaction*

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

sa_flags:

- specifies a set of flags which modify the behavior of the signal.
- It can comprise several flags (through bitwise OR)
- Example
  - SA_NOCLDSTOP | SA_ONSTACK | …
  - SA_SIGINFO to use the field sa_sigaction as handler
  - | is the bitwise OR operator
  - See **man 3 sigaction**

Example with *sigaction*

```
int main(void)   {
  struct sigaction act;
  act.sa_handler = process_signal;
  sigemptyset( &act.sa_mask );
  act.sa_flags = 0;
  sigaction( SIGINT, &act, 0 );
  while(1)
         {
         printf("waiting one second - press CTRL+C :)\n");
         sleep(1);
         }
  return 0;
  }
void process_signal(int signum){
  printf("Capturing %d (SIGINT=%d)\n",signum, SIGINT);
}
```

The program will not terminate with CTRL+C (SIGINT)
However, it will terminate with CTRL+\ (SIGQUIT)

The printf function is **not** an async-signal-safe (it can lead to race conditions)
The “write” function is async-signal-safe (see man 7 signal-safety)

- SIGINT can be sent to a foreground process with CTRL+C

Example with siginfo_t (1)

```
/* Setting sigaction() to use siginfo_t. */
static void hdl(int sig,siginfo_t *siginfo,void *context){
      printf ("Sending PID: %ld, UID: %ld\n",
       (long)siginfo->si_pid, (long)siginfo->si_uid);
}
 int main (int argc, char *argv[]){
      struct sigaction act;
      memset (&act, '\0', sizeof(act));
      /* Use sa_sigaction field: handle has
 two additional parameters */
      act.sa_sigaction = &hdl;
```

http://www.linuxprogrammingblog.com/code-examples/sigaction

Example with siginfo_t (2)

```
/* SA_SIGINFO flag tells sigaction() to use the sa_sigaction
field, not sa_handler. */
        act.sa_flags = SA_SIGINFO;
        if (sigaction(SIGTERM, &act, NULL) < 0){
                perror ("sigaction");
                return 1;
        }
        while (1){
                sleep(10);
        }
        return 0;
}
```

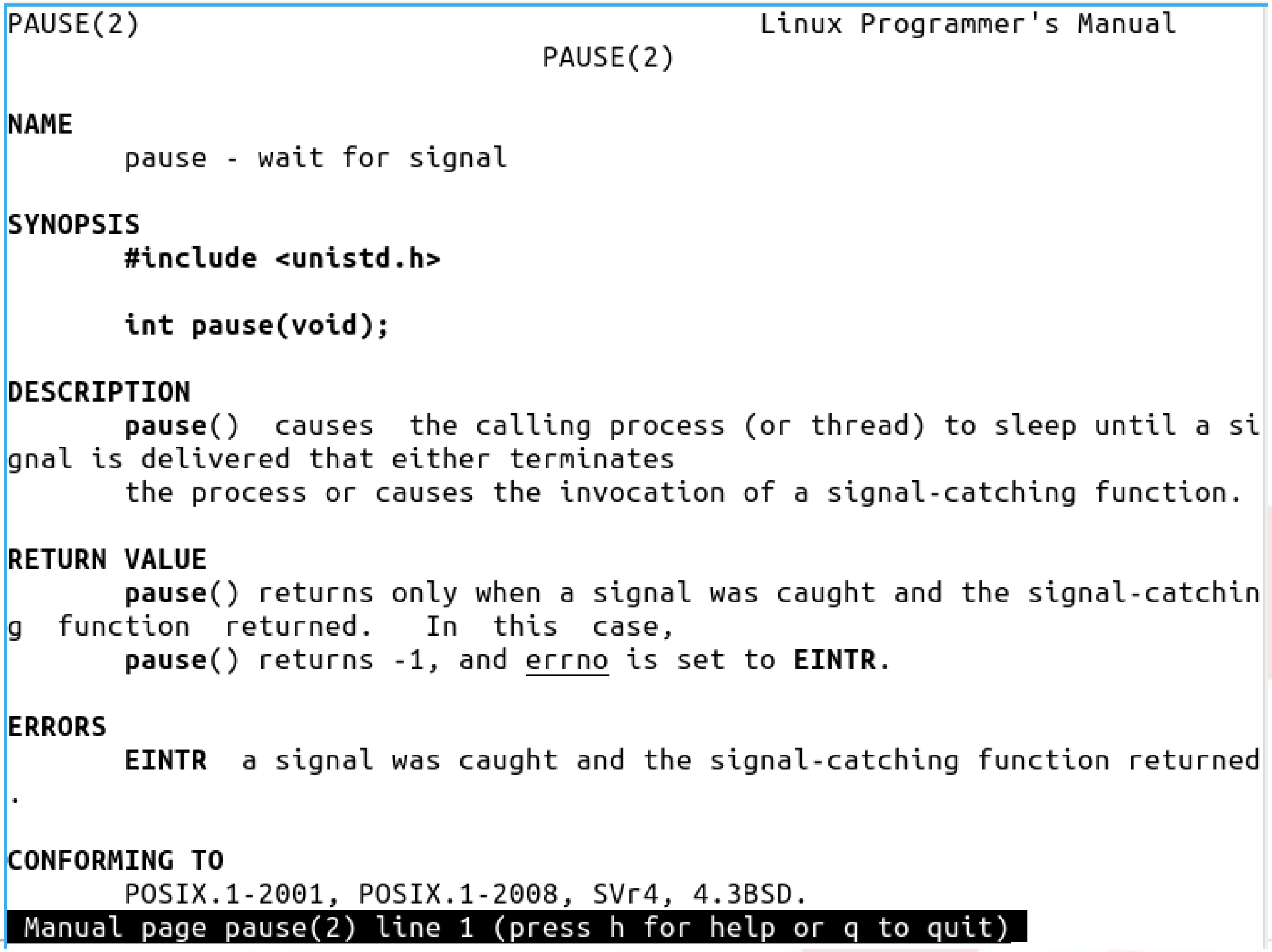
http://www.linuxprogrammingblog.com/code-examples/sigaction

System calls and signals

- When a system call (e.g. `read()`) is interrupted by a signal…
  1) The signal handler is called
  2) The signal handler terminates and thus returns the control back to the system call
     - On UNIX , slow system calls do not resume.
     - Whenever a signal is received within a system call, the system call returns an error and sets `errno` to EINTR
- What is a *slow system call*?
  – System calls that perform I/O operations on devices that can block the caller *forever*
    - sockets (networks), pipes
    - The pause( ) system call
      – Blocks the process until it receives a signal
- Use `SA_RESTART` to recover slow system calls automatically

pause function

- The pause(2) function
  - Suspends execution until any signal is caught.

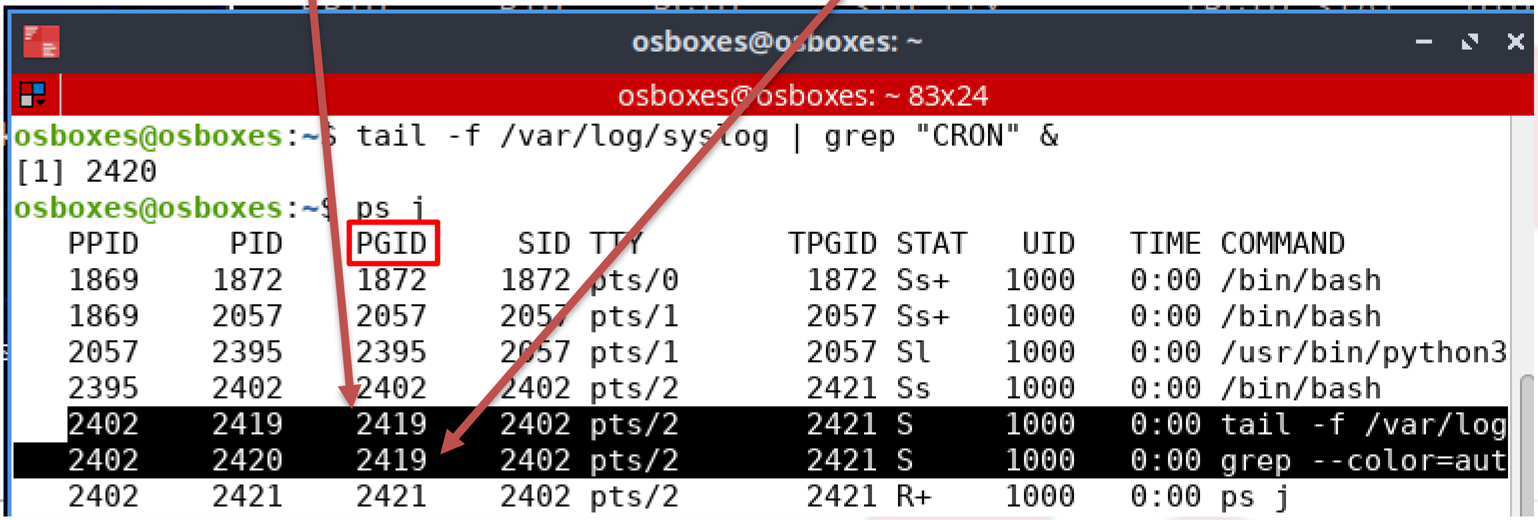


```
PAUSE(2)                                      Linux Programmer's Manual
                                PAUSE(2)

NAME
       pause - wait for signal

SYNOPSIS
       #include <unistd.h>

       int pause(void);

DESCRIPTION
       pause()  causes  the calling process (or thread) to sleep until a si
gnal is delivered that either terminates
       the process or causes the invocation of a signal-catching function.

RETURN VALUE
       pause() returns only when a signal was caught and the signal-catchin
g  function  returned.   In  this  case,
       pause() returns -1, and errno is set to EINTR.

ERRORS
       EINTR  a signal was caught and the signal-catching function returned
.

CONFORMING TO
       POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.
 Manual page pause(2) line 1 (press h for help or q to quit)
```

PROCESS GROUPS, SESSIONS & SIGNALS

Process group

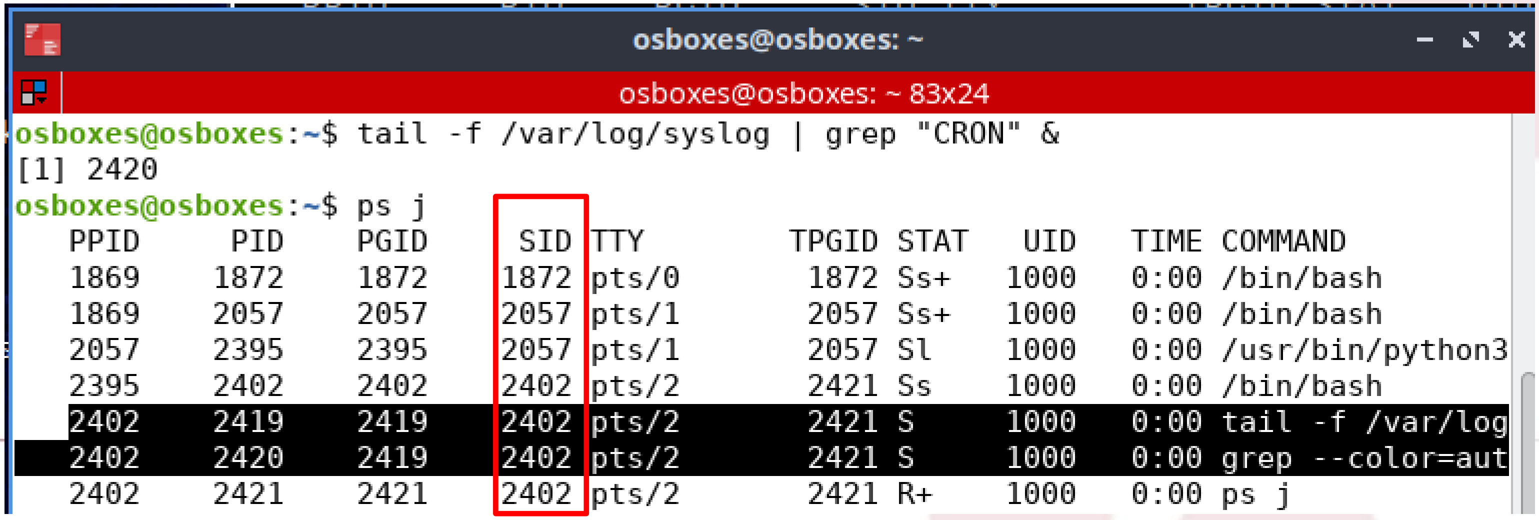
- In addition to the parent/child process association, there are two more associations:
  - Process group: PGID
  - Sessions: SID
- bash creates a process group for each command line that has pipes
- Each process group has a unique PGID
- Command "ps j" lists the processes with the indication of the PGID and SID
- Example: **tail -f /var/log/syslog | grep "CRON" &**



```
osboxes@osboxes:~$ tail -f /var/log/syslog | grep "CRON" &
[1] 2420
osboxes@osboxes:~$ ps j
   PPID     PID    PGID     SID TTY        TPGID STAT   UID   TIME COMMAND
   1869    1872    1872    1872 pts/0       1872 Ss+   1000   0:00 /bin/bash
   1869    2057    2057    2057 pts/1       2057 Ss+   1000   0:00 /bin/bash
   2057    2395    2395    2057 pts/1       2057 Sl    1000   0:00 /usr/bin/python3
   2395    2402    2402    2402 pts/2       2421 Ss    1000   0:00 /bin/bash
   2402    2419    2419    2402 pts/2       2421 S     1000   0:00 tail -f /var/log
   2402    2420    2419    2402 pts/2       2421 S     1000   0:00 grep --color=aut
   2402    2421    2421    2402 pts/2       2421 R+    1000   0:00 ps j
```

Sessions (SID)

- Session is a Process group set
  - Usually associated with:
    - A control terminal
    - A process designated as a session leader
  - If a session has a control terminal, it will have a single foreground process group, and all other process groups in the session will be background groups
    - **Example**: BASH with PID 2402 is session leader SID=2402 which also contains tail (PID=2419) and grep (PID=2420)



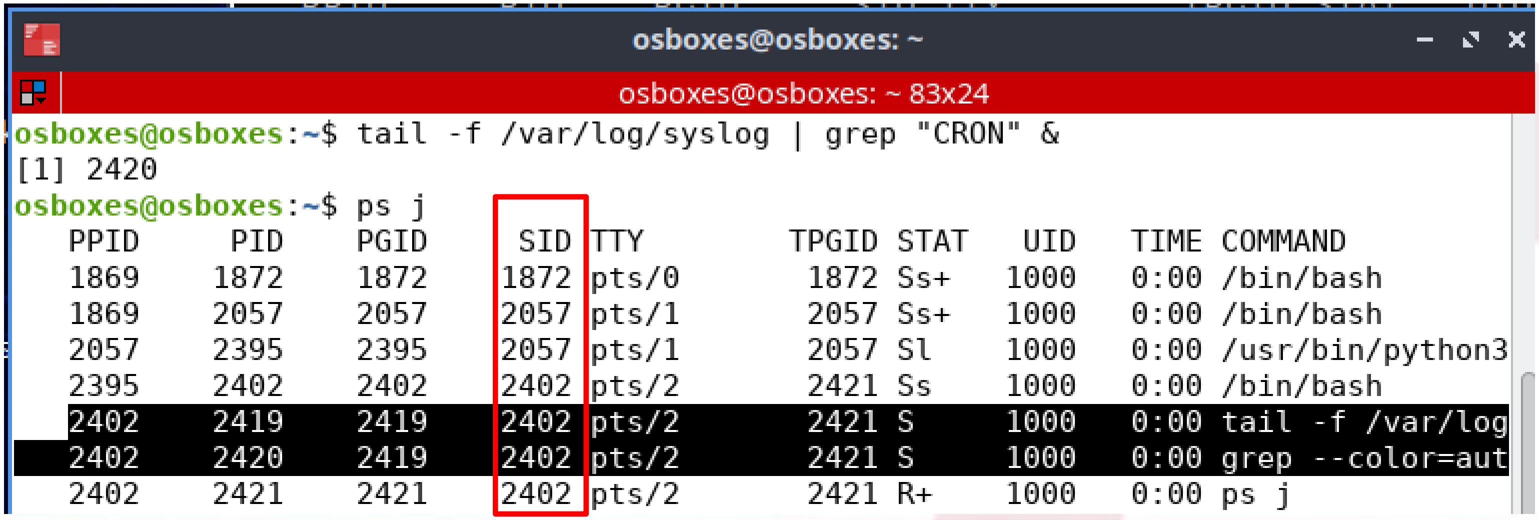
```
osboxes@osboxes:~$ tail -f /var/log/syslog | grep "CRON" &
[1] 2420
osboxes@osboxes:~$ ps j
   PPID     PID    PGID     SID TTY        TPGID STAT   UID   TIME COMMAND
   1869    1872    1872    1872 pts/0       1872 Ss+   1000   0:00 /bin/bash
   1869    2057    2057    2057 pts/1       2057 Ss+   1000   0:00 /bin/bash
   2057    2395    2395    2057 pts/1       2057 Sl    1000   0:00 /usr/bin/python3
   2395    2402    2402    2402 pts/2       2421 Ss    1000   0:00 /bin/bash
   2402    2419    2419    2402 pts/2       2421 S     1000   0:00 tail -f /var/log
   2402    2420    2419    2402 pts/2       2421 S     1000   0:00 grep --color=aut
   2402    2421    2421    2402 pts/2       2421 R+    1000   0:00 ps j
```

Signals/ GPID / SID

- Sending a signal to a process group
  - Kill command
  - Syntax: **`kill -SIGTERM -PGID`**
  - Example: **`kill -SIGTERM -2402`**

Indicate minus sign

- Sending a signal to a SID session
  - **pkill** Command
  - Syntax: **`pkill -s SID`**
  - Example: **`pkill -s 2402`**

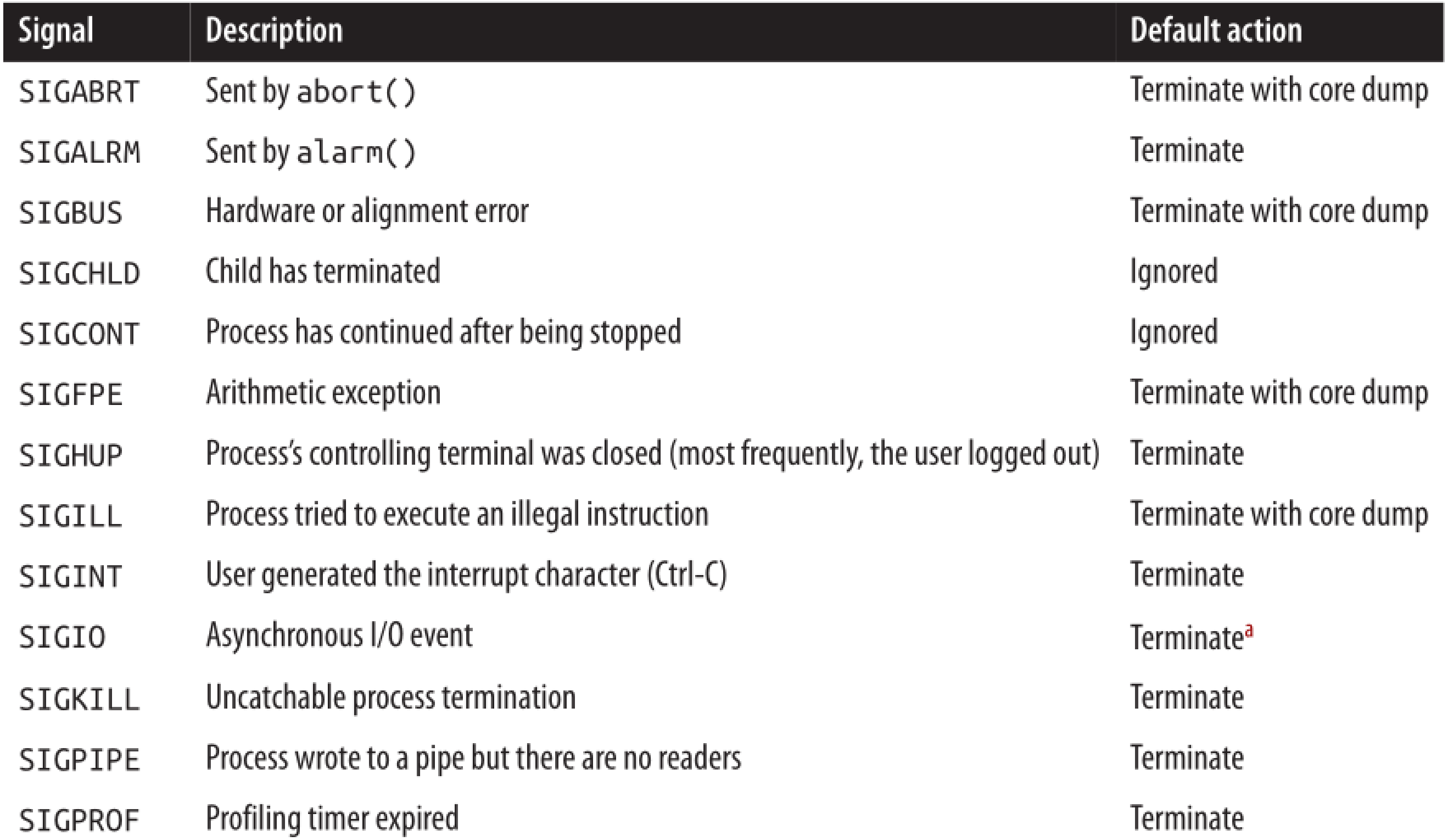


```
osboxes@osboxes:~$ tail -f /var/log/syslog | grep "CRON" &
[1] 2420
osboxes@osboxes:~$ ps j
   PPID     PID    PGID     SID TTY        TPGID STAT   UID   TIME COMMAND
   1869    1872    1872    1872 pts/0       1872 Ss+   1000   0:00 /bin/bash
   1869    2057    2057    2057 pts/1       2057 Ss+   1000   0:00 /bin/bash
   2057    2395    2395    2057 pts/1       2057 Sl    1000   0:00 /usr/bin/python3
   2395    2402    2402    2402 pts/2       2421 Ss    1000   0:00 /bin/bash
   2402    2419    2419    2402 pts/2       2421 S     1000   0:00 tail -f /var/log
   2402    2420    2419    2402 pts/2       2421 S     1000   0:00 grep --color=aut
   2402    2421    2421    2402 pts/2       2421 R+    1000   0:00 ps j
```

SIGNALS IN LINUX

Signals in Linux (#1)

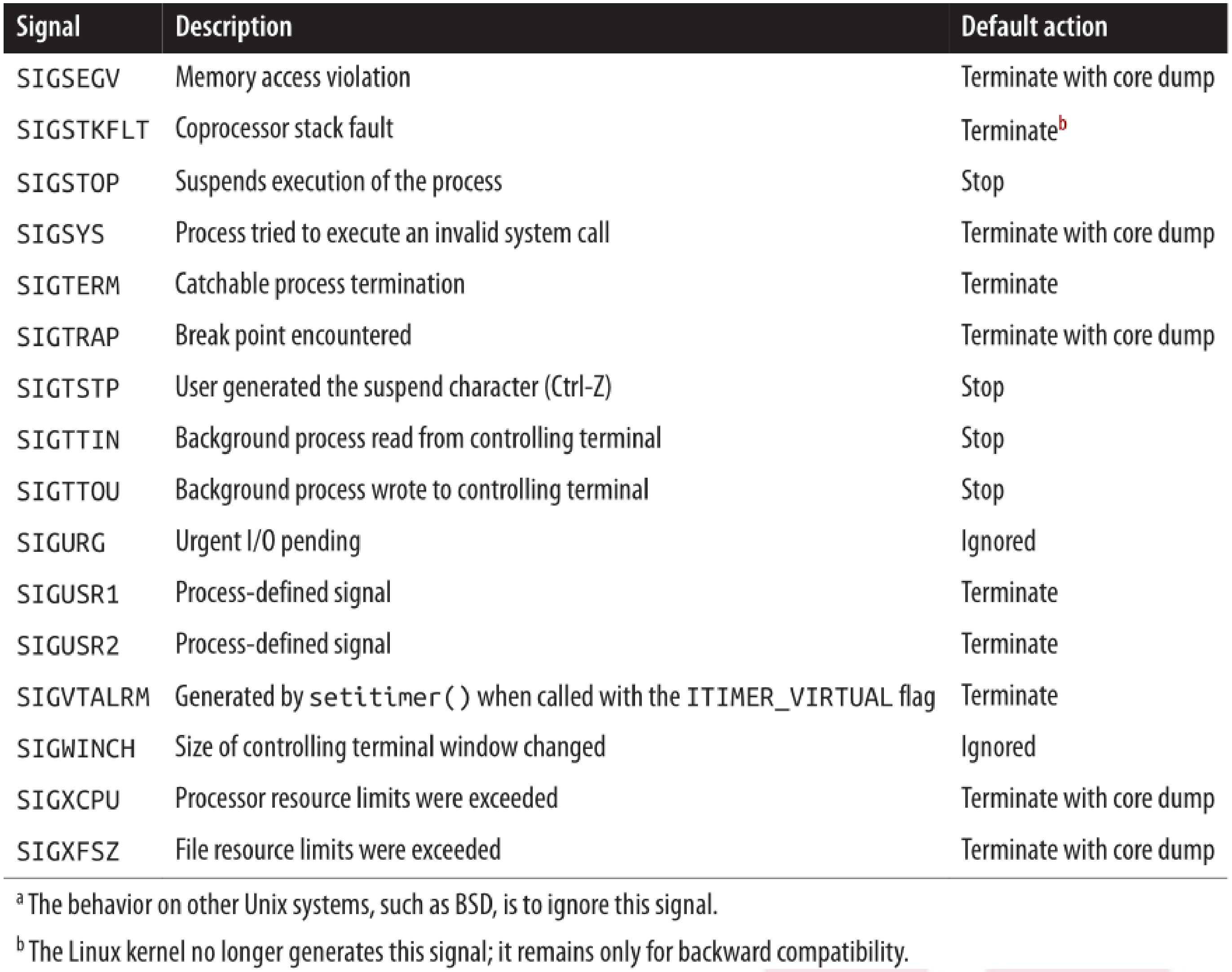
- Source: *"Signals", Chapter 10 - Linux System Programming,* Robert Love, 2nd Edition, O'Reilly, 2013



Signal	Description	Default action
`SIGABRT`	Sent by `abort()`	Terminate with core dump
`SIGALRM`	Sent by `alarm()`	Terminate
`SIGBUS`	Hardware or alignment error	Terminate with core dump
`SIGCHLD`	Child has terminated	Ignored
`SIGCONT`	Process has continued after being stopped	Ignored
`SIGFPE`	Arithmetic exception	Terminate with core dump
`SIGHUP`	Process's controlling terminal was closed (most frequently, the user logged out)	Terminate
`SIGILL`	Process tried to execute an illegal instruction	Terminate with core dump
`SIGINT`	User generated the interrupt character (Ctrl-C)	Terminate
`SIGIO`	Asynchronous I/O event	Terminate[a]
`SIGKILL`	Uncatchable process termination	Terminate
`SIGPIPE`	Process wrote to a pipe but there are no readers	Terminate
`SIGPROF`	Profiling timer expired	Terminate

Signals in Linux (#2)

- Source: *"Signals", Chapter 10 - Linux System Programming,* Robert Love, 2nd Edition, O'Reilly, 2013



Signal	Description	Default action
SIGSEGV	Memory access violation	Terminate with core dump
SIGSTKFLT	Coprocessor stack fault	Terminate[b]
SIGSTOP	Suspends execution of the process	Stop
SIGSYS	Process tried to execute an invalid system call	Terminate with core dump
SIGTERM	Catchable process termination	Terminate
SIGTRAP	Break point encountered	Terminate with core dump
SIGTSTP	User generated the suspend character (Ctrl-Z)	Stop
SIGTTIN	Background process read from controlling terminal	Stop
SIGTTOU	Background process wrote to controlling terminal	Stop
SIGURG	Urgent I/O pending	Ignored
SIGUSR1	Process-defined signal	Terminate
SIGUSR2	Process-defined signal	Terminate
SIGVTALRM	Generated by `setitimer()` when called with the `ITIMER_VIRTUAL` flag	Terminate
SIGWINCH	Size of controlling terminal window changed	Ignored
SIGXCPU	Processor resource limits were exceeded	Terminate with core dump
SIGXFSZ	File resource limits were exceeded	Terminate with core dump

[a] The behavior on other Unix systems, such as BSD, is to ignore this signal.

[b] The Linux kernel no longer generates this signal; it remains only for backward compatibility.

GETTING A SIGNAL’S NAME

Signal name in Linux

- The system-defined `sys_siglist[...]` vector of strings holds names of each signal
  - `extern const char *const sys_siglist[];`
- The name of signal is also available with:
  - `char *strsignal(int sig);`
- Example:

```
#include <signal.h>
for(i=0;i<20;i++){
   printf("signal %d => '%s'\n",i,sys_siglist[i]);
 }
```

References

- *"Signals", Chapter 10 - Linux System Programming,* Robert Love, 2nd Edition, O'Reilly, 2013
- "UNIX SIGNALS", Overview of signals
  - https://venam.nixers.net/blog/unix/2016/10/21/unix-signals.html
    - Audio: https://nixers.net/showthread.php?tid=2003
- Sigaction
  - http://www.linuxprogrammingblog.com/code-examples/sigaction