Jednym z czynników decydujących o stabilności systemu operacyjnego jest, jak wiemy już, odizolowanie od siebie procesów. Zdarzają się jednak sytuacje, gdy wymagana jest komunikacja między jednym a drugim procesem. Na przykład, gdy mamy dwa osobne programy operujące na wspólnym zestawie danych, wówczas jeden z procesów może blokować drugi, aby nie zapisywać danych jednocześnie, albo informować o tym, że są nowe dane do przetworzenia. W Unixie przewidziano **mechanizm komunikacji między procesami IPC** (Inter Process Communication). Oparty jest on na **sygnałach**, czyli powiadomieniach wysyłanych z jednego procesu do drugiego. Proces zawsze odbiera sygnały, ale jak na nie zareaguje to już kwestia implementacji, tego jak program został napisany, co zależy od konkretnych potrzeb (są od tej zasady 2 wyjątki, ale o tym później).

Sygnały mają przypisane numery oraz nazwy symboliczne. Listę wszystkich sygnałów możemy zobaczyć używając polecenia `kill -l`

```
canopus$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
 9) SIGKILL     10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGUSR1
17) SIGUSR2     18) SIGCHLD     19) SIGPWR      20) SIGWINCH
21) SIGURG      22) SIGIO       23) SIGSTOP     24) SIGTSTP
25) SIGCONT     26) SIGTTIN     27) SIGTTOU     28) SIGVTALRM
29) SIGPROF     30) SIGXCPU     31) SIGXFSZ     32) SIGWAITING
33) SIGLWP      34) SIGFREEZE   35) SIGTHAW     36) SIGCANCEL
37) SIGLOST     41) SIGRTMIN    42) SIGRTMIN+1  43) SIGRTMIN+2
44) SIGRTMIN+3  45) SIGRTMAX-3  46) SIGRTMAX-2  47) SIGRTMAX-1
48) SIGRTMAX
```

Konkretne definicje stałych odpowiadających sygnałom można znaleźć, przynajmniej w Solaris, w pliku `/usr/include/sys/iso/signal_iso.h`:

```
#define SIGHUP  1       /* hangup */
#define SIGINT  2       /* interrupt (rubout) */
#define SIGQUIT 3       /* quit (ASCII FS) */
#define SIGILL  4       /* illegal instruction (not reset when caught) */
#define SIGTRAP 5       /* trace trap (not reset when caught) */
#define SIGIOT  6       /* IOT instruction */
#define SIGABRT 6       /* used by abort, replace SIGIOT in the future */
#define SIGEMT  7       /* EMT instruction */
#define SIGFPE  8       /* floating point exception */
#define SIGKILL 9       /* kill (cannot be caught or ignored) */
#define SIGBUS  10      /* bus error */
#define SIGSEGV 11      /* segmentation violation */
#define SIGSYS  12      /* bad argument to system call */
#define SIGPIPE 13      /* write on a pipe with no one to read it */
#define SIGALRM 14      /* alarm clock */
#define SIGTERM 15      /* software termination signal from kill */
#define SIGUSR1 16      /* user defined signal 1 */
#define SIGUSR2 17      /* user defined signal 2 */
#define SIGCLD  18      /* child status change */
#define SIGCHLD 18      /* child status change alias (POSIX) */
#define SIGPWR  19      /* power-fail restart */
#define SIGWINCH 20     /* window size change */
#define SIGURG  21      /* urgent socket condition */
#define SIGPOLL 22      /* pollable event occured */
#define SIGIO   SIGPOLL /* socket I/O possible (SIGPOLL alias) */
#define SIGSTOP 23      /* stop (cannot be caught or ignored) */
#define SIGTSTP 24      /* user stop requested from tty */
```

```
#define SIGCONT 25      /* stopped process has been continued */
#define SIGTTIN 26      /* background tty read attempted */
#define SIGTTOU 27      /* background tty write attempted */
#define SIGVTALRM 28    /* virtual timer expired */
#define SIGPROF 29      /* profiling timer expired */
#define SIGXCPU 30      /* exceeded cpu limit */
#define SIGXFSZ 31      /* exceeded file size limit */
#define SIGWAITING 32   /* reserved signal no longer used */
#define SIGLWP  33      /* reserved signal no longer used */
#define SIGFREEZE 34    /* special signal used by CPR */
#define SIGTHAW 35      /* special signal used by CPR */
#define SIGCANCEL 36    /* reserved signal for thread cancellation */
#define SIGLOST 37      /* resource lost (eg, record-lock lost) */
#define SIGXRES 38      /* resource control exceeded */
#define SIGJVM1 39      /* reserved signal for Java Virtual Machine */
#define SIGJVM2 40      /* reserved signal for Java Virtual Machine */
```

Polecenie `kill` zaprojektowano do wysyłania sygnałów, choć nazwa sugeruje coś trochę innego. Sygnał określamy na dwa sposoby: podając wartość numeryczną lub symboliczną. Wartość sygnału poprzedzamy znakiem - albo używamy opcji -s. Drugi konieczny argument to identyfikator PID procesu, do którego chcemy wysłać sygnał. Na przykład, gdy chcemy wysłać sygnał SIGALRM do procesu o identyfikatorze 2918 to możemy to zrobić w jeden z następujących sposobów:

```
canopus$ kill -SIGALRM 2918
canopus$ kill -ALRM 2918
canopus$ kill -14 2918
canopus$ kill -s SIGALRM 2918
canopus$ kill -s ALRM 2918
canopus$ kill -s 14 2918
```

Jak widać przedrostek `SIG` w nazwie symbolicznej sygnału można pominąć. Polecenie `pkill` działa tak samo tyle, że zamiast PID procesu podajemy nazwę programu - podobnie jak w `pgrep`. Na przykład:

```
canopus$ pkill -SIGALRM moj_program
canopus$ pkill -ALRM moj_program
canopus$ pkill -14 moj_program
canopus$ pkill -s SIGALRM moj_program
canopus$ pkill -s ALRM moj_program
canopus$ pkill -s 14 moj_program
```

Jeśli nie podamy jaki sygnał wysyłamy poprzez `kill` lub `pkill`, to wysłany zostanie domyślny sygnał `SIGTERM` zakończenia programu. Nazwa programu `kill` wzięła się pewnie stąd, że najczęściej używany sygnał jaki wysyłamy przy jego pomocy to `SIGKILL` aby wymusić zakończenie procesu.

Dwa sygnały są wyjątkowe:

- `SIGKILL` 9 : bezwarunkowe zakończenie procesu (kill)
- `SIGSTOP` 23 : wstrzymanie wykonania procesu (stop)

Nie mogą one być ani przechwycone, ani zignorowane. Pozostałe sygnały można oprogramować. Polega to na tym, że piszemy funkcję w programie (może to być C albo skrypt shellowy) a potem w odpowiedni sposób kojarzymy tę funkcję z sygnałem jaki nas interesuje. Do przechwytywania sygnałów w shellu mamy wbudowaną funkcję `trap`, a w bibliotece C mamy następujące funkcje:

```
#include <signal.h>

void (*signal(int sig, void (*disp)(int)))(int);
```

```
void (*sigset(int sig, void (*disp)(int)))(int);
int sighold(int sig);
int sigrelse(int sig);
int sigignore(int sig);
int sigpause(int sig);
```

W załączniku jest skrypt `sig.sh`.

```
#!/bin/sh

hello () {
    echo "Hello!"
}

surprise () {
    echo "Surprise!"
}

trykill () {
    echo "Trying to kill"
}

trystop () {
    echo "Trying to stop"
}

trap hello INT
trap surprise ALRM
trap trykill KILL
trap trystop STOP

i=0
while true
do
    echo $i
    sleep 1
    i=`expr $i + 1`
done
```

Wprawdzie skrypty będziemy pisać nieco później, ale składnia i sens tego małego skryptu są chyba dość czytelne. Mamy tam zaimplementowane 4 funkcje:

- `hello()`,
- `surprise()`,
- `trykill()` oraz
- `trystop()`.

Wykonanie `trap` powoduje skojarzenie wskazanej funkcji z podanym sygnałem. Sygnał określamy symbolicznie lub numerycznie. Dalej mamy nieskończoną pętlę, która w odstępach co 1 sekundę wypisuje kolejne liczby naturalne od 0. Skrypt uruchamiamy wołając program powłoki `sh`, albo wpisując jego nazwę, ale drugi sposób wymaga, aby wcześniej zmienić uprawnienia dodając sobie prawo uruchamiania `x`. Najprościej w katalogu, gdzie jest skrypt

```
canopus$ sh sig.sh
```

Z drugiego okienka terminala spróbujemy wysyłać sygnały do nowo utworzonego procesu. Najpierw jednak należy ustalić jego PID:

```
canopus$ ps -ef | fgrep sig
```

```
 mariusz  7605  2335   0 12:18:23 pts/7       0:00 sh sig.sh
 mariusz  7618  2412   0 12:18:27 pts/8       0:00 fgrep sig
```

Polecenie `pgrep` nam nie zadziała tak jak byśmy chcieli, bo programem wykonywanym w tym wypadku jest `sh` a `sig.sh` to tylko jego argument. Dostajemy tutaj 2 odpowiedzi, bo jednym z procesów pasujących do naszego polecenia filtrującego `fgrep sig` jest sam on sam. Tutaj 7605 to interesujący nas PID. Wyślijmy kilka sygnałów obserwując jednocześnie w oknie obok co wypisuje nasz skrypt:

```
canopus$ kill -2 7605
canopus$ kill -INT 7605
canopus$ kill -14 7605
canopus$ kill -ALRM 7605
canopus$ kill -9 7605
canopus$ kill -23 7605
```

Przy okazji. Sygnał `SIGINT` to sygnał wysyłany przez naciśnięcie CTRL+C w tym samym terminalu co uruchomiony skrypt. Podobnie `SIGSTOP` to CTRL+Z. Co się dzieje gdy naciskamy te kombinacje klawiszy? Co się dzieje gdy wysyłamy sygnały używając polecenie `kill`?

Jak zatrzymać nasz proces, skoro typowe przerwanie CTRL+C nie robi tego co trzeba? Wyślijmy do naszego procesu `SIGKILL`:

```
canopus$ kill -9 7605
```