

# Signals

## Topics

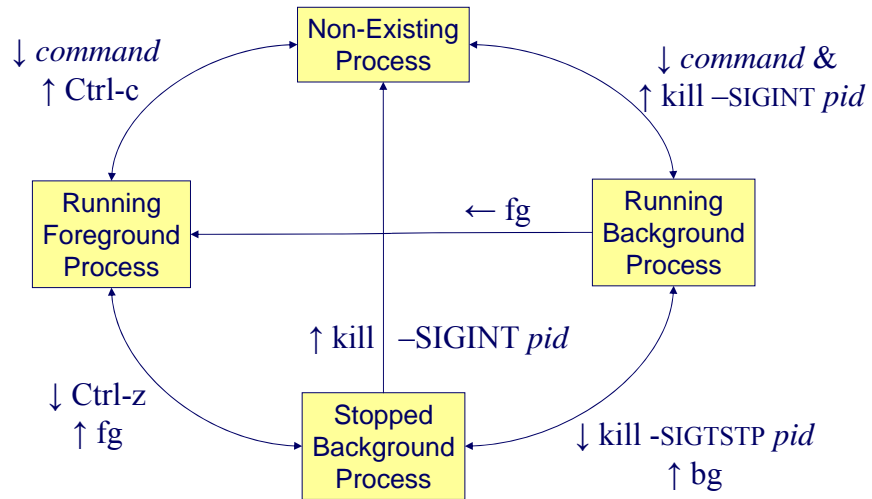
- Unix Process Control
- Sending Signals
- Receiving Signals
- Blocking Signals

## Two Fundamental Questions

Q1: How does the OS communicate to an application process?

Q2: How does an application process communicate to the OS?

## Unix Process Control



- 3 -

## Unix Process Control

[Demo of unix process control on tanner.]

- 4 -

## Exactly what happens when you:

### Type Ctrl-c?

- Keyboard sends hardware interrupt
- Hardware interrupt is handled by OS
- OS sends a 2/SIGINT **signal**

### Type Ctrl-z?

- Keyboard sends hardware interrupt
- Hardware interrupt is handled by OS
- OS sends a 20/SIGTSTP **signal**

### Issue a “kill *-sig pid*” command?

- OS sends a *sig* **signal** to the process whose id is *pid*

### Issue a “fg” or “bg” command?

- OS sends a 18/SIGCONT **signal**

- 5 -

## Signals

A *signal* is a software interrupt delivered to a process

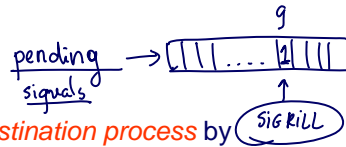
- Reports that an event of some type has occurred.
- Sent from the kernel (sometimes at the request of another process) to a process.
- Different signals are identified by small integer ID's
- The only information in a signal is its ID.

ID	Name	Default Action	Corresponding Event
2	<b>SIGINT</b>	Terminate	Interrupt from keyboard ( <i>ct1-c</i> )
9	<b>SIGKILL</b>	Terminate	Kill program (cannot override or ignore)
11	<b>SIGSEGV</b>	Terminate & Dump	Segmentation violation
14	<b>SIGALRM</b>	Terminate	Timer signal
17	<b>SIGCHLD</b>	Ignore	Child stopped or terminated

- 6 -

# Signal Concepts (1)

## Sending a signal



- Kernel **sends** (delivers) a signal to a **destination process** by updating some state in the context of the destination process.
- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)  $x/y$ 

```
int x=2;
int y=0;
```
  - Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process.

# Signal Concepts (2)

## Receiving a signal

- A destination process **receives** a signal when it is forced by the kernel to react in some way to the delivery of the signal.

- Three possible ways to react:
  1. ● Ignore the signal (do nothing)
  2. ● Terminate the process.
  3. ● Catch the signal by executing a user-level function called a signal handler.

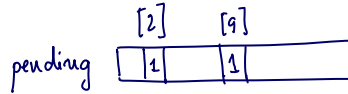
```

void catch_int (int sig)
{
    ≡
}

Example:
int main ()
{
    signal (SIGINT, catch_int);
}

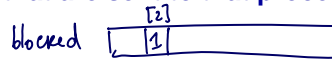
signal (SIGINT, SIG_IGN);
signal (SIGSTP, SIG_IGN);
    
```

## Signal Concepts (3)



A signal is **pending** if it has been sent but not yet received.

- There can be at most one pending signal of any particular type.
- Important: Signals are not queued
  - If a process has a pending signal of type  $k$ , then subsequent signals of type  $k$  that are sent to that process are discarded.



A process can **block** the receipt of certain signals.

- Blocked signals can be delivered, but will not be received until the signal is unblocked.

A pending signal is received at most once.

- 9 -

## Signal Concepts (4)

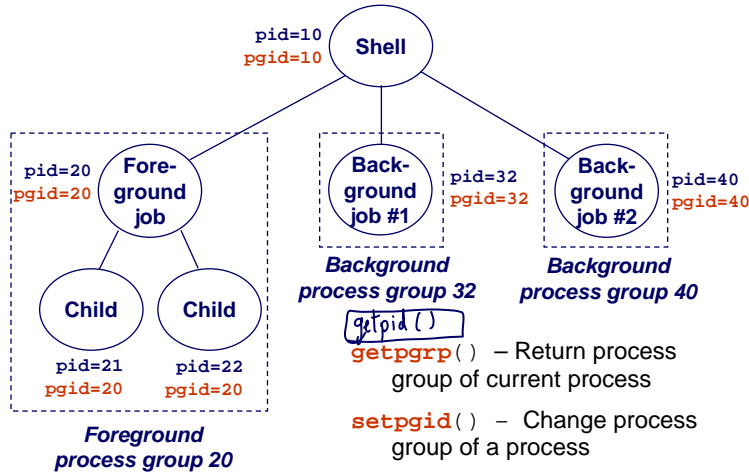
Kernel maintains **pending** and **blocked** bit vectors in the context of each process.

- **pending** – represents the set of pending signals
  - Kernel sets bit  $k$  in **pending** when a signal of type  $k$  is delivered.
  - Kernel clears bit  $k$  in **pending** when a signal of type  $k$  is received.
- **blocked** – represents the set of blocked signals
  - Can be set and cleared by the application using the **sigprocmask** function.

- 10 -

# Process Groups

Every process belongs to exactly one process group.



- 11 -

# Sending Signals

1. Using the kill program eg: `kill -SIGKILL pid`
2. From the keyboard eg: `CTRL-C, CTRL-Z`
3. Using the kill function → From within a program  
`kill(pid, SIGKILL);`

- 12 -

# 1. Sending Signals with kill Program

**kill** program sends arbitrary signal to a process or process group

Examples:

- `kill -9 24818`
  - Send SIGKILL to process 24818
- `kill -SIGKILL -24817`
  - Send SIGKILL to every process in group 24817 (observe that PID is negative here)
- `kill -SIGINT 1234`
  - Same as pressing Ctrl-c if process 1234 is running in foreground

No signal type name or number specified => sends 15/SIGTERM signal

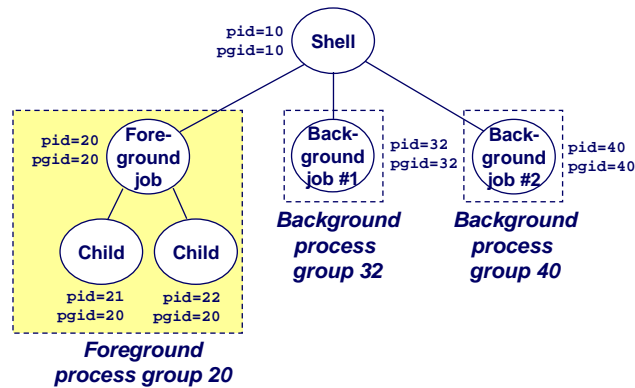
Comment: Better command name would be **sendsig**

- 13 -

# 2. Sending Signals from the Keyboard

Typing `ctrl-c` (`ctrl-z`) sends a SIGINT(SIGTSTP) to every job in the *foreground* process group.

- SIGINT – default action is to **terminate** each process
- SIGTSTP – default action is to **stop (suspend)** each process



- 14 -

## Example of ctrl-c and ctrl-z

```
bash$ ./forkex
Child: pid=24868 pgrp=24867
Parent: pid=24867 pgrp=24867
<typed ctrl-z>
Suspended
bash$ ps -a
  PID TTY          STAT       TIME COMMAND
 24788 pts/2    S           0:00 -bash
 24867 pts/2    T           0:01 ./forkex → Stopped
 24868 pts/2    T           0:01 ./forkex
 24869 pts/2    R           0:00 ps a
bash$ fg
./forkex
<typed ctrl-c> → sigINT is sent to
bash$ ps -a
  PID TTY          STAT       TIME COMMAND
 24788 pts/2    S           0:00 -bash
 24870 pts/2    R           0:00 ps a
```

- 15 -

## 3. Sending Signals with kill Function

### kill(...)

```
int kill(pid_t pid, int sig);
```

- Sends a sig signal to the process whose id is pid
- Comment: Better function name would be sendsig()

### Example

```
pid_t pid = getpid(); /* Process gets its id.*/
kill(pid, SIGINT); /* Process sends itself a
                    SIGINT signal (commits
                    suicide?) */
```

- 16 -

## kill Function Example

```
void fork12()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

*N processes are forked  
Each process runs an infinite loop.*

*Parent terminates all children*

- 17 -

## Receiving Signals

1. Default Actions
2. Signal Handlers

- 18 -

## Receiving Signals

Suppose kernel is returning from exception handler and is ready to pass control to process  $p$ .

Kernel computes  $\text{pnb} = \text{pending} \& \sim\text{blocked}$

- The set of pending nonblocked signals for process  $p$

If ( $\text{pnb} == 0$ )

- Pass control to next instruction in the logical flow for  $p$ .

Else

- Choose least nonzero bit  $k$  in  $\text{pnb}$  and force process  $p$  to receive signal  $k$ .
- The receipt of the signal triggers some *action* by  $p$
- Repeat for all nonzero  $k$  in  $\text{pnb}$ .
- Pass control to next instruction in logical flow for  $p$ .

- 19 -

## Default Actions

Each signal type has a predefined *default action*, which is one of:

- The process terminates
- The process terminates and dumps core.
- The process stops until restarted by a SIGCONT signal.
- The process ignores the signal.

- 20 -

## Installing Signal Handlers

The `signal` function modifies the default action associated with the receipt of signal `signum`:

```
■ handler_t *signal(int signum, handler_t *handler)
```

Different values for `handler`:

- SIG\_IGN: ignore signals of type `signum`
- SIG\_DFL: revert to default action on receipt of signals of type `signum`.
- Otherwise, handler is the address of a *signal handler*
  - Called when process receives signal of type `signum`
  - Referred to as “*installing*” the handler.
  - Executing handler is called “*catching*” or “*handling*” the signal.
  - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.

- 21 -

## Predefined Signal Handler: SIG\_IGN

Can install to ignore signals

```
int main(void) {  
    signal(SIGINT, SIG_IGN);  
    ...  
}
```

Subsequently, process will ignore SIGINT signals

- 22 -

## Predefined Signal Handler: SIG\_DFL

Can install to restore default signal handler

```
int main(void) {  
    signal(SIGINT, SIG_IGN);  
    ...  
    signal(SIGINT, SIG_DFL);  
}
```

Subsequently, process will handle 2/SIGINT signals using the default handler for 2/SIGINT signals

- 23 -

## Signal Handling Exceptions

A program *cannot* install its own handler for signals:

- 9/SIGKILL
  - Default handler exits the process
  - Catchable termination signal is 15/SIGTERM
  
- 19/SIGSTOP
  - Default handler suspends the process
  - Can resume the process with signal 18/SIGCONT
  - Catchable suspension signal is 20/SIGTSTP

- 24 -

## Signal Handling Example

```
void int_handler(int sig)
{
    printf("Process %d received signal %d\n",
           getpid(), sig);
    exit(0);
}

void fork13()
{
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);

    ... /* same as before */
}
```

```
bash$ ./fork13
Killing process 24973
Killing process 24974
Killing process 24975
Killing process 24976
Killing process 24977
Process 24977 received signal 2
Child 24977 terminated with exit status 0
Process 24976 received signal 2
Child 24976 terminated with exit status 0
Process 24975 received signal 2
Child 24975 terminated with exit status 0
Process 24974 received signal 2
Child 24974 terminated with exit status 0
Process 24973 received signal 2
Child 24973 terminated with exit status 0
Bash$
```

- 25 -

## Signal Handler Funkiness

### Pending signals are not queued

```
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    printf("Received signal %d from process %d\n",
           sig, pid);
}

void fork14()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Exit */
            exit(0);
        }
    while (ccount > 0)
        pause(); /* Suspend until signal occurs */
}
```

- For each signal type, just have single bit indicating whether or not signal is pending
- Even if multiple processes have sent this signal

- 26 -

## Living With Nonqueuing Signals

Must check for all terminated jobs

- Typically loop with `wait`

```
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    while ((pid = wait(&child_status)) > 0) {
        ccount--;
        printf("Received signal %d from process %d\n", sig, pid);
    }
}

void fork15()
{
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
}
```

- 27 -

## Externally Generated Events (ctrl-c)

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
    printf("You think hitting ctrl-c will stop the bomb?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler); /* installs ctrl-c handler */
    while(1);
}
```

- 28 -

## Internally Generated Events

```
#include <stdio.h>
#include <signal.h>

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
    printf("BEEP\n");
    fflush(stdout);

    if (++beeps < 5)
        alarm(1);
    else {
        printf("BOOM!\n");
        exit(0);
    }
}
```

```
main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in
              1 second */

    while (1) {
        /* handler returns here */
    }
}
```

```
bash$ ./a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bash$
```

- 29 -

## Signal Blocking

1. Race Conditions
2. Blocking Signals

- 30 -

## Race Conditions in Signal Handlers

A **race condition** is a flaw in a program whereby the correctness of the program is critically dependent on the sequence or timing of other events.

We have seen that race conditions can occur in threads ...

Race conditions can also occur in signal handlers...

- 31 -

## Race Condition Example

```
void addSalaryToSavings(int sig) {  
    int iTemp;  
    iTemp = iSavingsBalance;  
    iTemp += iMonthlySalary;  
    iSavingsBalance = iTemp;  
}
```

Handler for hypothetical  
"update monthly salary" signal

- 32 -

## Race Condition Example (1)

(1) Signal arrives; handler begins executing

```
void addSalaryToSavings(int iSig) {  
    int iTemp;  
    ↓ iTemp = iSavingsBalance;           2000  
    iTemp += iMonthlySalary;  
    iSavingsBalance = iTemp;  
}
```

- 33 -

## Race Condition Example (2)

(2) Another signal arrives; first instance of handler is interrupted; second instance of handler begins executing

```
void addSalaryToSavings(int iSig) {  
    ... int iTemp;  
    ↓ iTemp = iSavingsBalance;           2000  
    iTemp += iMonthlySalary;  
    iSavingsBalance = iTemp;  
}
```

```
void addSalaryToSavings(int iSig) {  
    int iTemp;  
    ↓ iTemp = iSavingsBalance;           2000  
    iTemp += iMonthlySalary;  
    iSavingsBalance = iTemp;  
}
```

- 34 -

## Race Condition Example (3)

(3) Second instance executes to completion

```
void addSalaryToSavings(int iSig) {  
    ... int iTemp;  
    ↓ iTemp = iSavingsBalance;           2000  
    iTemp += iMonthlySalary;  
    iSavingsBalance = iTemp;  
}
```

```
void addSalaryToSavings(int iSig) {  
    int iTemp;  
    iTemp = iSavingsBalance;           2000  
    iTemp += iMonthlySalary;          2050  
    ↓ iSavingsBalance = iTemp;         2050  
}
```

- 35 -

## Race Condition Example (4)

(4) Control returns to first instance, which executes to completion

```
void addSalaryToSavings(int iSig) {  
    ... int iTemp;  
    ↓ iTemp = iSavingsBalance;           2000  
    iTemp += iMonthlySalary;          2050  
    ↓ iSavingsBalance = iTemp;         2050  
}
```

Lost 50 !!!

- 36 -

## Blocking Signals in Handlers

### Blocking signals

- To **block** a signal is to **save** it for delivery at a later time

### Why block signals when handler is executing?

- Avoid race conditions when another signal of type x occurs while the handler for type x is executing

### How to block signals when handler is executing?

- **Automatic** during execution of signal handler!!!
- Previous sequence **cannot happen!!!**
- While executing a handler for a signal of type x, all signals of type x are blocked
- When/if signal handler returns, block is removed

- 37 -

## Blocking Signals in General

Race conditions can occur elsewhere too

```
int iFlag = 0;

void myHandler(int iSig) {
    iFlag = 1;
}

int main(void) {
    if (iFlag == 0) {
        /* Do something */
    }
}
```

Problem: myflag might become 1 just after the comparison!

Must make sure that critical sections of code are not interrupted.

- 38 -

## Blocking Signals in General

### POSIX standard

- Defines `sigprocmask()` and `sigaction()` func.
- Provides mechanism to block signals in general

### Each process has a signal mask in the kernel

- OS uses the mask to decide which signals to deliver
- User program can modify mask with `sigprocmask()`

### Functions for constructing signal sets

- `sigemptyset()`, `sigaddset()`, ...

- 39 -

## Blocking Signals - `sigprocmask`

### `sigprocmask()`

```
int sigprocmask(int iHow,
                const sigset_t *psSet,
                sigset_t *psOldSet);
```

- `psSet`: Pointer to a signal set
- `psOldSet`: (Irrelevant for our purposes)
- `iHow`: How to modify the signal mask
  - `SIG_BLOCK`: Add `psSet` to the current mask
  - `SIG_UNBLOCK`: Remove `psSet` from the current mask
  - `SIG_SETMASK`: Install `psSet` as the signal mask
- Returns 0 iff successful

- 40 -

## Blocking Signals Example

```
sigset_t sSet;
int main(void) {
    int iRet;
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGINT);
    iRet = sigprocmask(SIG_BLOCK, &sSet, NULL);
    assert(iRet == 0);
    if (iFlag == 0) {
        /* Do something */
    }
    iRet = sigprocmask(SIG_UNBLOCK, &sSet, NULL);
    assert(iRet == 0);
    ...
}
```

- 41 -

## Blocking Signals in Handlers

Signals of type x automatically are blocked when executing handler for signals of type x

Additional signal types to be blocked can be defined at time of handler installation...

- 42 -

## Installing a Signal Handler

### `sigaction()`

```
int sigaction(int iSig,  
              const struct sigaction *psAction,  
              struct sigaction *psOldAction);
```

- `iSig`: The type of signal to be affected
- `psAction`: Pointer to a structure containing instructions on how to handle signals of type `iSig`, including signal handler name and which signal types should be blocked
- `psOldAction`: (Irrelevant for our purposes)
- Installs an appropriate handler
- Automatically blocks signals of type `iSig`
- Returns 0 iff successful

- 43 -

## Installing a Handler Example

Program `testsigaction.c`:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <signal.h>  
  
void myHandler(int iSig) {  
    printf("In myHandler with argument %d\n", iSig);  
}  
...
```

- 44 -

## Installing a Handler Example (contd.)

Program testsigaction.c (contd.):

```
...
int main(void) {
    int iRet;
    struct sigaction sAction;
    sAction.sa_flags = 0;
    sAction.sa_handler = myHandler;
    sigemptyset(&sAction.sa_mask);
    iRet = sigaction(SIGINT, &sAction, NULL);
    assert(iRet == 0);

    printf("Entering an infinite loop\n");
    while (1);
    return 0;
}
```

- 45 -

## Installing a Handler Example (contd.)

[Demo of testsigaction.c]

- 46 -

## Predefined Signals

List of the predefined signals:

```
bash$ kill -l
```

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGIOT	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	

Applications can define their own signals

- An application can define signals with unused values

- 47 -

## Summary

A signal is an asynchronous event mechanism

- Can generate from user programs
- Can define effect by defining signal handler

Signals don't have queues

- Just one bit for each pending signal type

Beware of race conditions

- Signals of type x automatically are blocked while handler for type x signals is running
- POSIX `sigprocmask()` blocks signals in any critical section of code

- 48 -