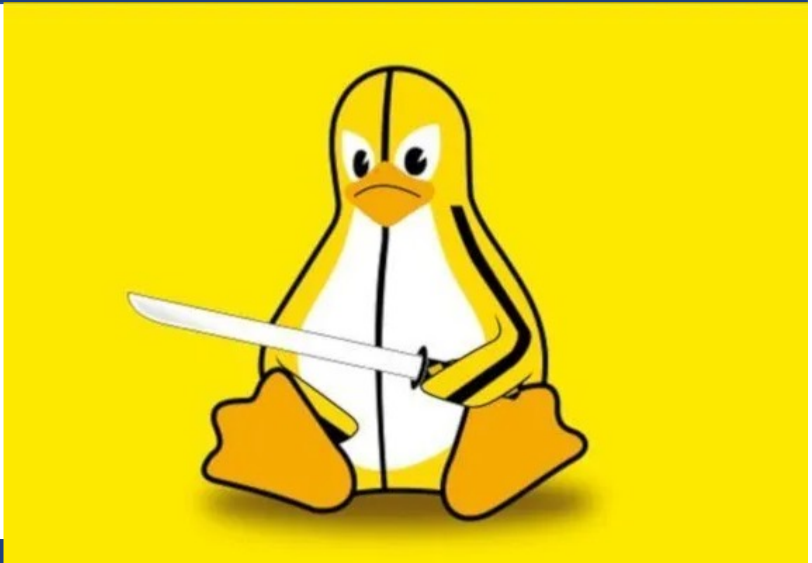
CMPSC 311 - Introduction to Systems Programming

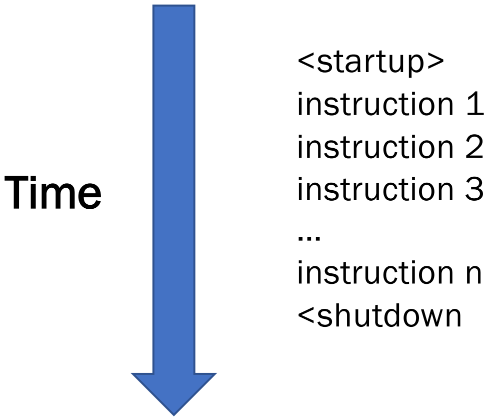


Signals

Professors:
Suman Saha
(Slides are mostly by *Professor Patrick McDaniel* and *Professor Abutalib Aghayev*)

Control Flow

- Processors do only one thing:
  - from startup to shutdown, a CPU simply reads and executes a sequence of instructions, one at a time
  - This sequence is the CPU’s control flow (or flow of control)

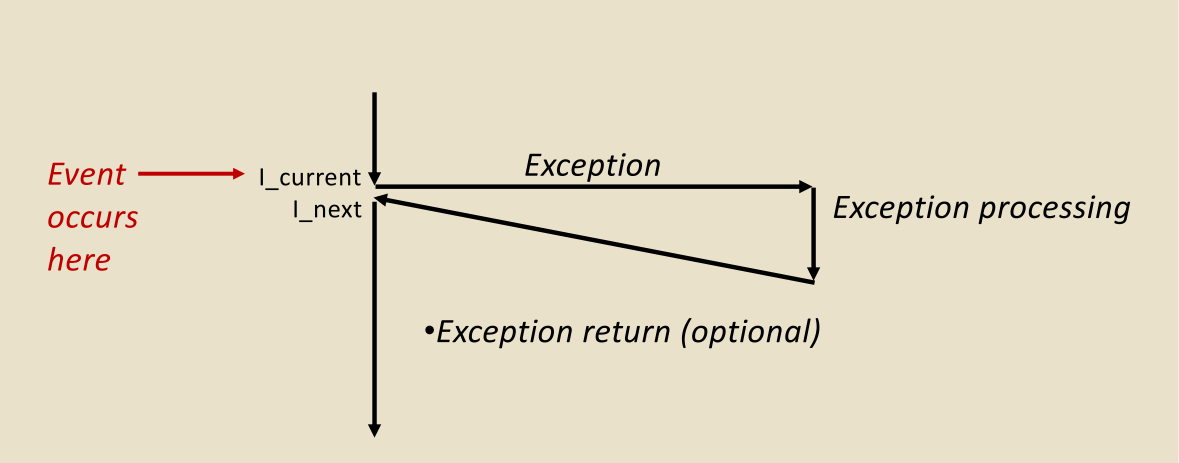


Time

<startup>
instruction 1
instruction 2
instruction 3
…
instruction n
<shutdown

Exceptional Control Flow

- Exceptional control flow enables a system to react to an event



Exceptional Control Flow

PennState

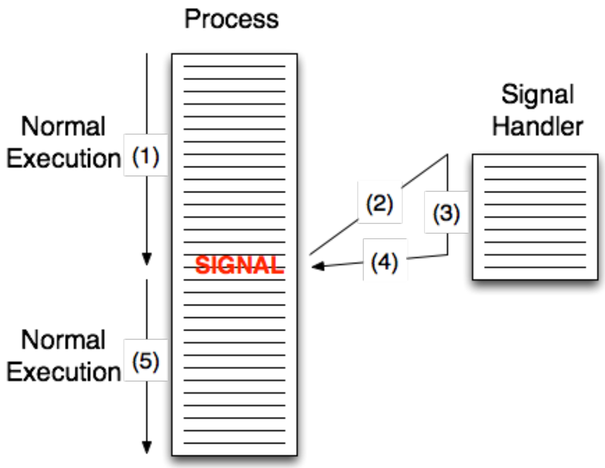
- Mechanisms exists at all levels of a computer system for exceptional control
- Low-level mechanisms
    - Exceptions
        - Examples: interrupts, traps, faults, and aborts
    - Implemented using combination of hardware and OS software
- High-level mechanisms
    - Process context switch (implemented by OS software and hardware timer)
    - Signals (implemented by OS software)
    - Nonlocal jumps: setjmp() and longjmp() (implemented by C runtime library)

UNIX Signals

- A signal is a special message sent through the OS to tell a process (or thread) of some command or event

- The process execution stops and special "signal handler" code runs.

- The process can resume operation after the signal handling is complete.



Signal types (abbreviated)

```
/* Signals */
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).  */
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGFPE          8       /* Floating-point exception (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGSEGV         11      /* Segmentation violation (ANSI).  */
#define SIGTERM         15      /* Termination (ANSI).  */
#define SIGSTKFLT       16      /* Stack fault.  */
#define SIGCHLD         17      /* Child status has changed (POSIX).  */
#define SIGCONT         18      /* Continue (POSIX).  */
#define SIGSYS          31      /* Bad system call.  */
```

Signals as process control

- The operating system use signals to control process behavior
  - Signals are sent on errors

```
#define SIGILL          4       /* Illegal instruction (ANSI).  */
#define SIGTRAP         5       /* Trace trap (POSIX).  */
#define SIGIOT          6       /* IOT trap (4.2 BSD).  */
#define SIGBUS          7       /* BUS error (4.2 BSD).  */
#define SIGFPE          8       /* Floating-point exception (ANSI).  */
#define SIGSEGV         11      /* Segmentation violation (ANSI).  */
```

  - Signals can be used by other applications too

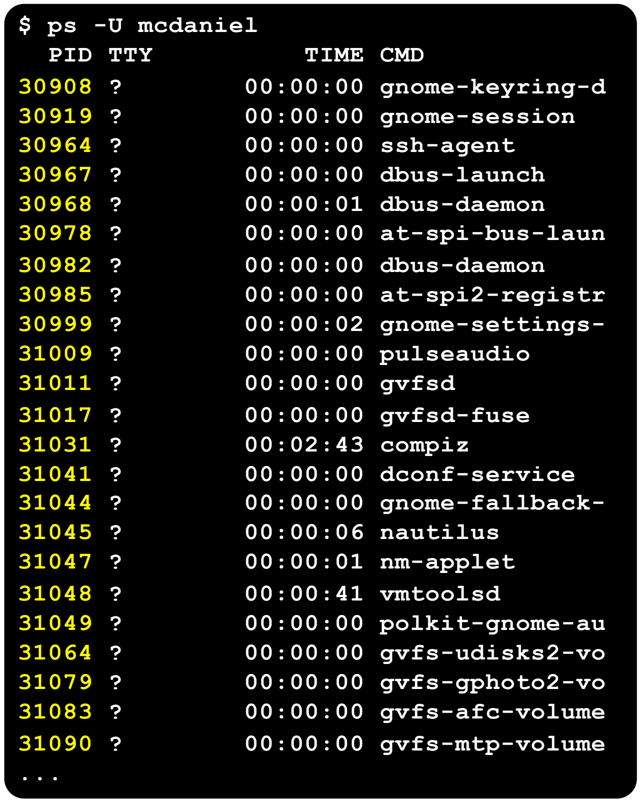
```
#define SIGUSR1         10      /* User-defined signal 1 (POSIX).  */
#define SIGUSR2         12      /* User-defined signal 2 (POSIX).  */
```

  - Control the process execution

```
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGCONT         18      /* Continue (POSIX).  */
#define SIGSTOP         19      /* Stop, unblockable (POSIX).  */
```

Process IDs

- Every process running on the OS is given a unique process ID (PID)
  - This is what is used in the OS and for process control to reference that specific running program instance.
- To find a process ID for a program, use the ps utility
  - The ps stands for “process status”



```
$ ps -U mcdaniel
  PID TTY          TIME CMD
30908 ?        00:00:00 gnome-keyring-d
30919 ?        00:00:00 gnome-session
30964 ?        00:00:00 ssh-agent
30967 ?        00:00:00 dbus-launch
30968 ?        00:00:01 dbus-daemon
30978 ?        00:00:00 at-spi-bus-laun
30982 ?        00:00:00 dbus-daemon
30985 ?        00:00:00 at-spi2-registr
30999 ?        00:00:02 gnome-settings-
31009 ?        00:00:00 pulseaudio
31011 ?        00:00:00 gvfsd
31017 ?        00:00:00 gvfsd-fuse
31031 ?        00:02:43 compiz
31041 ?        00:00:00 dconf-service
31044 ?        00:00:00 gnome-fallback-
31045 ?        00:00:06 nautilus
31047 ?        00:00:01 nm-applet
31048 ?        00:00:41 vmtoolsd
31049 ?        00:00:00 polkit-gnome-au
31064 ?        00:00:00 gvfs-udisks2-vo
31079 ?        00:00:00 gvfs-gphoto2-vo
31083 ?        00:00:00 gvfs-afc-volume
31090 ?        00:00:00 gvfs-mtp-volume
...
```

kill

- Kill is a program than sends signals to processes.

```
kill [-<sig>] <pid>
```

- Where <sig> is the signal number and <pid> is the process ID of the running program you want to send the signal.
  - If no SIGNUM is given, then SIGTERM is used by default.

PID

program

```
$ ps -U mcdaniel
57613 pts/4    00:00:00 signals
$ kill -1 57613
$ kill -2 57613
$ kill -9 57613
```

```
$ ./signals
Sleeping ...zzzzz ....
Signal handler got a SIGHUP!
Signals received : 1
Woken up!!
Sleeping ...zzzzz ....
Signal handler got a SIGNINT!
Signals received : 2
Woken up!!
Sleeping ...zzzzz ....
Killed
```

<sig> { 1 : SIGHUP
2: SIGINT
9: SIGKILL

SIGTERM vs. SIGKILL

- `SIGTERM` interrupts the program and asks it to shut down, which it should.
  - Sometimes this does not work (for instance when the process is in a locked state)
  - It is often desirable to add a signal handler to handle the SIGTERM, so that it can gracefully shut down the process, cleanup memory, close files, etc.
- `SIGKILL` kills the process
  - Can lead to inconsistent state, because there is no opportunity to gracefully shutdown the process.

Definition: the term *graceful shutdown* refers to the proper and complete sync with secondary storage, disposal of resources, and normal termination.

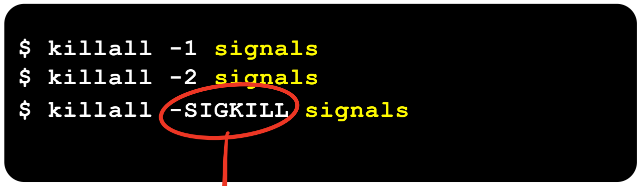
send signals to all instances of a patricular program

killall

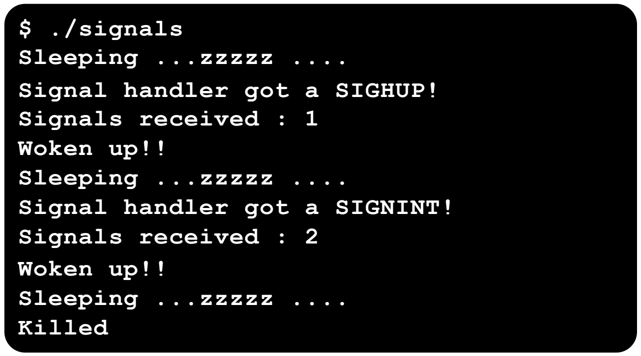
- Killall is a program than sends signals to all instances of a particular progam.

killall [-<sig>] <name>

- Where <sig> is the signal number and <name> is the name of running program you want to send the signal.
  - If no SIGNUM is given, then SIGTERM is used by default.



```
$ killall -1 signals
$ killall -2 signals
$ killall -SIGKILL signals
```



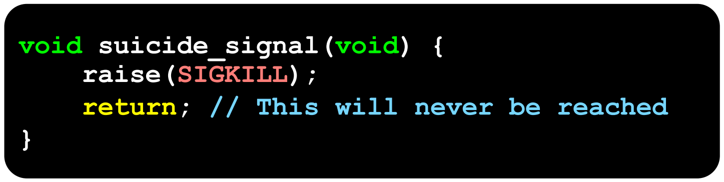
```
$ ./signals
Sleeping ...zzzzz ....
Signal handler got a SIGHUP!
Signals received : 1
Woken up!!
Sleeping ...zzzzz ....
Signal handler got a SIGNINT!
Signals received : 2
Woken up!!
Sleeping ...zzzzz ....
Killed
```

raise()

- raise allows a process to send signals to itself.

`int raise(int sig);`

- There are a range of reasons why a process might want to do this.
  - Suspend itself (`SIGSTOP`)
  - Kill itself (`SIGKILL`)
  - Reset its configuation (`SIGHUP`)
  - User defined signals (`SIGUSR1`..)



```
void suicide_signal(void) {
    raise(SIGKILL);
    return; // This will never be reached
}
```

User-defined signal handlers

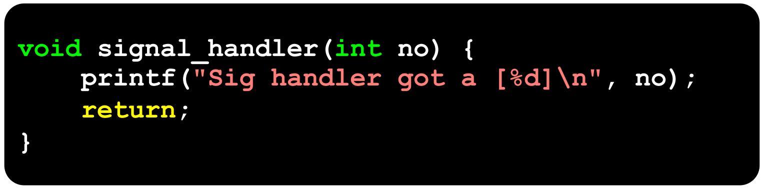
- You can create your own signal handlers simply by creating a function

```
void <fname>( int <var name> )
```

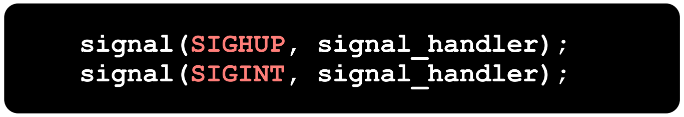
- and passing a function pointer to the function

```
sighandler_t signal(int signum, sighandler_t handler);
```

- Thereafter, whenever a signal of the type `signo` is raised. your program is called instead of the default handler.



```
void signal_handler(int no) {
    printf("Sig handler got a [%d]\n", no);
    return;
}
```



```
signal(SIGHUP, signal_handler);
signal(SIGINT, signal_handler);
```

Function pointers

- A function pointer is a pointer to a function that can be assigned, passed as parameters, and called

`<return> (*<var>)(<params>);`

  - `<return>` is the return type of the function
  - `<var>` is the variable names
  - `<params >` are the parameters , separated by commas

```
int myfunc(int i) {
  printf("Got into function with %d\n", i);
  return 0;
}

int main( void ) {
  int (*func)(int);
  func = myfunc; //set variable name to function
  func(7);
  return 0;
}
```

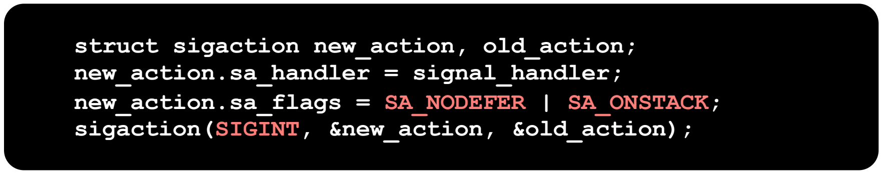
```
$ ./signals
Got into function with 7
$
```

An alternate approach

- The `sigaction()` system call changes the action taken by a process on receipt of a specific signal.

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- Where:
  - `signnum` - is the signal number to be handled
  - `act` - is a structure containing information about the new handler, NULL means ignore the signal
  - `oldact` - is a pointer to the previously assigned handler, as assigned in call to function



```
struct sigaction new_action, old_action;
new_action.sa_handler = signal_handler;
new_action.sa_flags = SA_NODEFER | SA_ONSTACK;
sigaction(SIGINT, &new_action, &old_action);
```

```
signalhandler_t signal (int signum, signalhandler_t handler);
int sigaction (int signum, const struct sigaction * new_act,
                                    struct sigaction * old_act);
```

Why another API?

PennState

- Many argue that the `sigaction` function is better:
  - The signal() function does not block other signals from arriving while the current handler is executing; sigaction() can block other signals until the current handler returns.
  - The signal() function resets the signal action back to SIG_DFL (default) for almost all signals.
  - Better tuning of signals/controls of process through flags
    - SA_NODEFER - don’t suspend signals while in handler
    - SA_ONSTACK - provide alternate stack for signal handler
    - SA_RESETHAND - Restore the signal action to the default upon entry to the signal handler.

Note: *In general*, `sigaction` is preferred over signal.

Putting it all together ...

```c
void signal_handler(int no) {
    printf(”Signal received : %d\n”, no);
    if (no == SIGHUP) {
        printf("Signal handler got a SIGHUP!\n");
    } else if (no == SIGINT) {
        printf("Signal handler got a SIGNINT!\n");
    }
    return;
}
void cleanup_handler(int no) {
    printf("Killed”);
    exit(0);
}
int main(void) {
    struct sigaction new_action, old_action; // Setup the signal actions
    new_action.sa_handler = signal_handler;
    new_action.sa_flags = SA_NODEFER | SA_ONSTACK;
    sigaction( SIGINT, &new_action, &old_action );
    signal( SIGHUP, signal_handler );      // Setup the signal handlers
    signal( SIGTERM, cleanup_handler );

    while (1) {
        printf( "Sleeping ...zzzzz ....\n" );
        select( 0, NULL, NULL, NULL, NULL );
        printf( "Woken up!!\n" );
    }

    // Return successfully
    return 0;
}
```

```
$ ./signals
Sleeping ...zzzzz ....
Signal received : 1
Signal handler got a SIGHUP!
Woken up!!
Sleeping ...zzzzz ....
Signal received : 2
Signal handler got a SIGNINT!
Woken up!!
Sleeping ...zzzzz ....
Killed
```