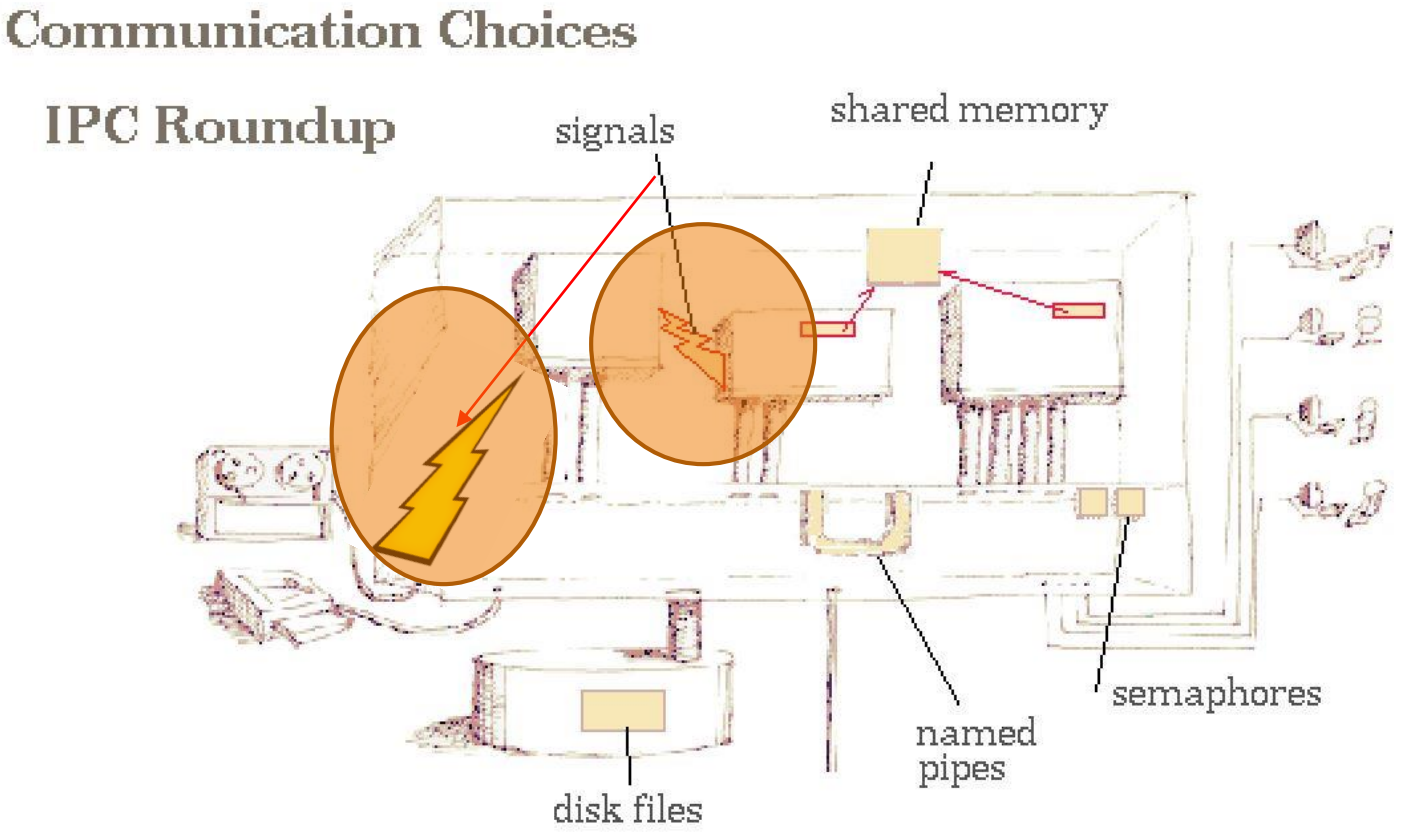
W10: IPC - SIGNALS

Spring 2017

Inter-Process Communication Landscape



- Rendering from Prof. Farrell (Kent State University)

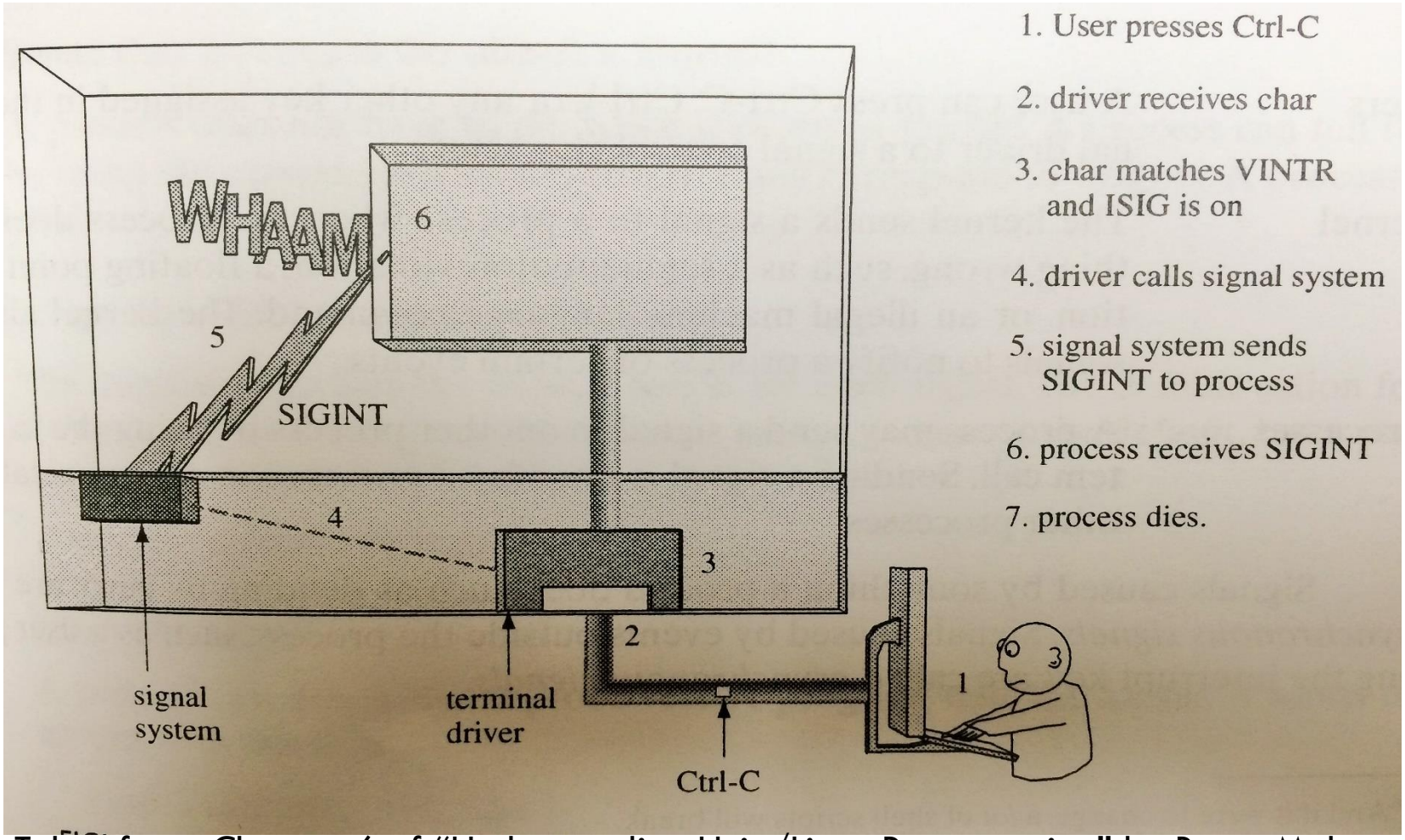
Inter-Process Communication

- IPC classes
  - Pipes and FIFO
  - Signals
  - Message Passing
  - Shared Memory
  - Semaphore Sets
- References:
  - Baseline slides: CSCE-313 Spring'14 Bettati & Gu
  - Advanced Programming in the UNIX Environment, Third Edition, W. Richard Stevens and Stephen A. Rago, Addison-Wesley Professional Computing Series, 2013. Chapters 10, 15
  - Understanding Unix/Linux Programming, Bruce Molay, Chapter 6
  - Advanced Linux Programming Ch 5
  - Some material also directly taken or adapted with changes from Illinois course in System Programming (Prof. Angrave), UCSD (Prof. Snoeren), and USNA (Prof. Brown)

Background: User-Mode Exceptional Flow

- So far exceptional control flow features have been usable only by the operating system
  - Exceptions:
    - Synchronous: faults, traps, aborts
    - Asynchronous: interrupts
- All exception handlers run in **protected (KERNEL) mode**
- Would like similar capabilities for **user mode code**
  - Inter-Process communication to facilitate 'exceptional control flow' is subject of today's discussion
  - We will discuss them through "Signals" mechanism
    - ……but also in a broader context beyond just IPC

Example: What does CTRL-C do?



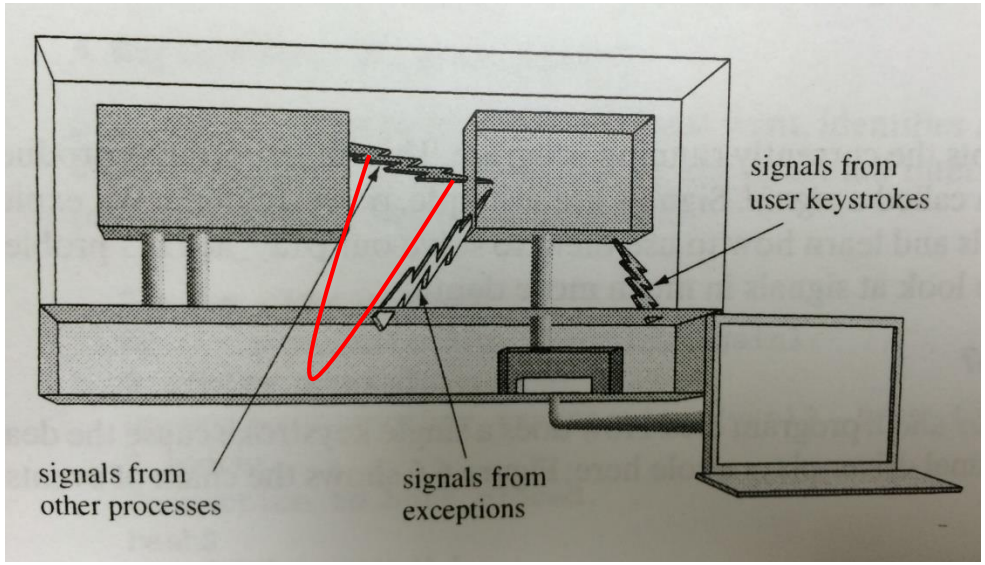
Taken from: Chapter 6 of "Understanding Unix/Linux Programming" by Bruce Molay

What is a Signal?

- A Signal is a one-word message
  - A Green Light is a signal, A Referee's whistle is a signal
  - These items and events do not contain messages, they are messages!
- Each Signal has a numerical code
- So when we press CTRL-C key we ask the Kernel to send the interrupt signal to the currently running process

Where do Signals come from?

- Today we'll look at Signals in a broader context
  - IPC is one of the contexts
  - Others are facilitated by **Users** and **Kernel**
    - [Users] Signals generated by external Input devices
    - [Kernel] Exceptions



Taken from: Chapter 6 of "Understanding Unix/Linux Programming" by Bruce Molay

Where do Signals come from?

(USER) Terminal-generated signals: triggered when user presses certain key on terminal. (e.g. ^C)

(KERNEL) Exception-generated signals: CPU execution detects condition and notifies kernel. (e.g. **`SIGFPE`** divide by 0, **`SIGSEGV`** invalid memory reference)

`kill(2)` function: Sends any signal to another process.

`kill(1)` command: The command-line interface to `kill(2)`.

(PROCESSES) Software-condition generated signals: Triggered by software event (e.g. **`SIGPIPE`** by broken pipe)

Signals and the Kernel - Modes

- Many of the signals are generated by the KERNEL in response to events and exceptions received
  - SIGALRM – timer interrupt
  - SIGFPE – FP exception
  - SIGILL – Illegal instruction
  - SigSEGV – Segment Violation
- Many others are routed through the Kernel, if not originating from the Kernel itself
  - E.g. SIGHUP (terminal hang), SIGINT (CTRL-C keyboard), SIGTSTP (CTRL-Z), SIGKILL (KILL)
- As with SYSTEM calls, the kernel receives the signals from hardware and other processes on behalf of a process
  - Then Kernel forwards the signal to the appropriate process

Generating Signals: `kill(2)` and `raise(3)`


```
#include <signal.h>

int kill(pid_t pid, int sig);
   /* send signal 'sig' to process 'pid' */
```

```
/* example: send signal SIGUSR1 to process 1234 */
if (kill(1234, SIGUSR1) == -1)
  perror("Failed to send SIGUSR1 signal");

/* example: kill parent process */
if (kill(getppid(), SIGTERM) == -1)
  perror("Failed to kill parent");
```

```
#include <signal.h>

int raise(int sig);
   /* Sends signal 'sig' to itself.
      Part of ANSI C library! */
```

Raise sends a signal to the executing process
Kill sends a signal to the specified process

Where can I find a list of Signals?

- Unix provides Signals
  - Location: /usr/include/signal.h
- Some example signals along with default behaviors

ID	Name	Description	Default Action
1	SIGHUP	Terminal line hangup	Terminate
2	SIGINT	Keyboard interrupt (Ctrl-C)	Terminate
3	SIGQUIT	Quit from keyboard (Ctrl-\)	Terminate
4	SIGILL	Illegal instruction	Terminate
8	SIGFPE	Floating-point exception	Terminate + dump core
9	SIGKILL	Kill program	Terminate
11	SIGSEGV	Invalid memory access	Terminate + dump core
14	SIGALRM	Timer signal from alarm function	Terminate
10	SIGUSR1	User-defined signal 1	Terminate
12	SIGUSR2	User-defined signal 2	Terminate
20	SIGTSTP	Stop from keyboard (Ctrl-Z)	Suspend until SIGCONT received

What can a Process do about a Signal?

Tell the kernel what to do with a signal:

1. **Accept Default action.** All signals have a default action

   signal (SIGINT, SIG_DFL)

2. **Ignore the signal.** Works for most signals

   signal (SIGINT, SIG_IGN)

   cannot ignore SIGKILL and SIGSTOP; also unwise to ignore hardware exception signals

3. **Catch the signal (call a function).** Tell the kernel to invoke a given function (signal handler) whenever signal occurs.

   signal (SIGINT, foo)

Simple Signal Handling: Example

linux2.cse.tamu.edu - PuTTY

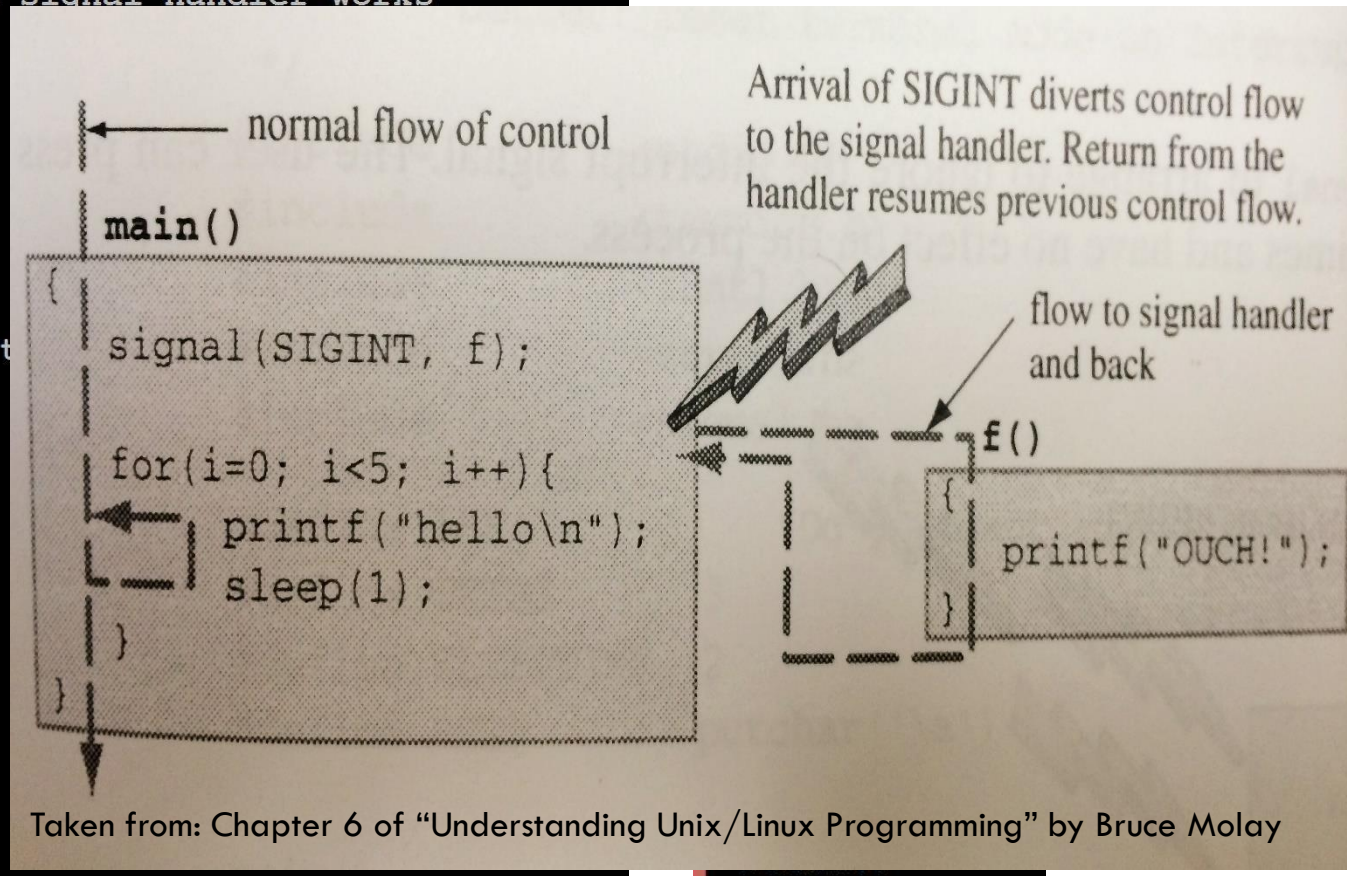
```
:: more sigdemo.c
/* sigdemo.c - shows how a signal handler works
 * credit: Bruce Molay
 */

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

main()
{
  void f(int); /* declare t
  int i;

  signal(SIGINT, f);
  for (i=0; i<5; i++) {
    printf("hello\n");
    sleep(1);
  }
}

void f(int signum)
{
  printf("OUCH!\n");
}
```



Taken from: Chapter 6 of "Understanding Unix/Linux Programming" by Bruce Molay

Signals Terminology

- A signal is **generated** for a process when event that causes the signal occurs. (Hardware exception, software condition, etc.)
- A signal is **delivered** when action for a signal is taken.
- During the time between generation and delivery, signal is **pending**.
- A process has the option of **blocking** the delivery of a signal.
  - Signal remains blocked until process either (a) unblocks the signal, or (b) changes the action to ignore the signal

Signals Terminology

- The system determines what to do with a blocked signal when the signal is delivered, not when it is generated.
- What happens when blocked signal is generated more than once? (If system delivers the signal more than once, the signal is **queued**)
- **signal mask** is the mechanism to define set of signals that are blocked from delivery.

Pending Signals

- For each process, the Kernel manages two bookkeeping variables for signal handling
  - **Pending** – a bit vector of signals that are currently pending for the process
    - These signals have been sent to the process, but haven't yet been handled by the process
  - **Each kind of signal has one bit assigned to it**
    - If a particular signal type is already pending and then is sent again, the second signal is dropped

Blocked Signals

- The Kernel also keeps a **blocked bit-vector** for each process
  - Each type of signal has a bit assigned to it
  - If a particular type of signal is blocked it will not be delivered to the process
- When the Kernel calls a signal handler on a process, that type of signal is **automatically blocked**
  - Generally signal handlers don't need to worry about being interrupted by the same kind of signal again

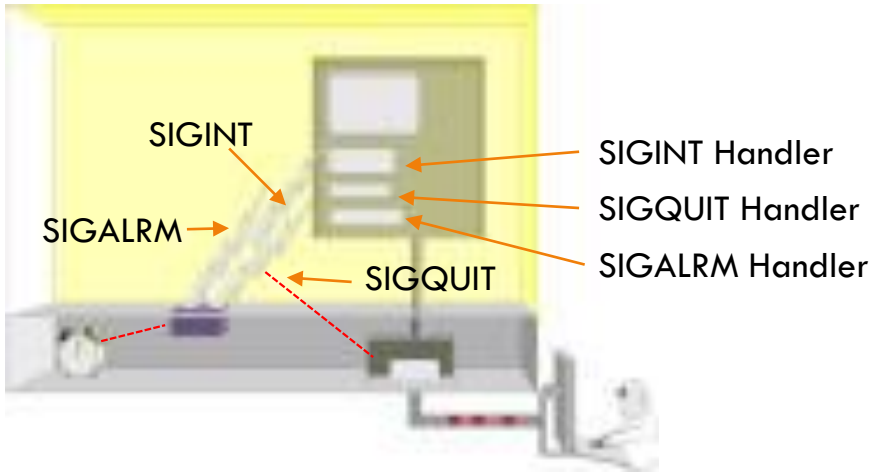
Blocked Signals (2)

- When a signal handler returns, the blocked signal type is automatically unblocked
  - When handler returns, signals of that type can start to be delivered again
  - In case of a blocked pending SIGINT, it will subsequently be delivered to the process
- Several functions for manipulating these signal bit vectors
  - **sigpending** (returns current pending signals for the process)
  - **sigprocmask** (manipulate the set of blocked signals for the process)

Blocked Signals - Example

- Example: A process with a SIGINT handler
  - First SIGINT received causes the SIGINT handler to be called
  - Also causes SIGINT to be blocked for the process
  - If another SIGINT occurs during Handler execution, it is recorded in pending bit vector but not delivered

Practice: Multiple Signals Handling….Process



- A Process receives multiple signals
  - What happens if SIGY is generated while the process is in SIGX handler?
  - What happens if a second SIGX is generated while the process is still in SIGX handler? Or a third SIGX?
  - What happens if a signal is generated while the program is blocking on input?

Signals Concept Refresh

- Where do signals come from?
  - User, Kernel, Process
- What can a process tell the Kernel to do about a Signal?
  - Accept Default, Ignore, Catch
- A Signal is “Generated” when ……
  - event that causes the signal occurs
- A Signal is “Delivered” when ……
  - action for signal is taken

Signals Concept Refresh

- A Signal is “Blocked” when …..
  - it is not allowed to be delivered
- **signal mask** can be used to control ……
  - set of signals that are blocked from delivery
- When the Kernel calls a signal handler on a process, that type of signal is ……
  - **automatically blocked**
- When a signal handler returns, the blocked signal type is …….
  - automatically unblocked

SUMMARY: Signals

- Signals share many common traits with hardware exception handling
  - A user-mode version of hardware exceptions
  - When a signal handler is invoked that type of signal is blocked until the handler returns
    - Very similar to H/W interrupts
- Signals allow us to leverage exceptional control flow in user programs
  - Enables powerful techniques in server programming
  - Are used in most widely used server programs
    - Web servers, email servers, DNS servers, Databases, etc.