Unix Signals

- A signal is a notification to a process that an event has occurred.
- Signals can be sent by one process to another (or to itself) or by the kernel to a process.
- The signal() system call provides a mechanism for user programs to react to signals by associating a function called a signal handler with a specific signal.

signal() system call

```c
void (*signal (int sig, void (*func)(int)))(int);
```

- Example:
  ```c
  signal(SIGUSR1, myfunc);
  ```
  This would tell the kernel to call the user defined function myfunc() whenever the signal SIGUSR1 is received.
- There are many different signals - each has a name specified in the include file signal.h.
POSIX Signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>Abnormal termination</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>Timeout</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>Erroneous arithmetic operation</td>
</tr>
<tr>
<td>SIGINT</td>
<td>Hangup</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>Interactive attention</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>Termination signal</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>Invalid memory reference</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>Interactive Stop (^Z)</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>Continued</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>Child process has terminated</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>User defined</td>
</tr>
</tbody>
</table>

Sources of Signals

- The kill() system call allows a process to send a signal to another process.
  
  ```c
  int kill( int pid, int sig );
  ```

- A signal can be sent via the kill system call only if the effective UID of the sending process is 0 (root) or matches the effective UID of the receiving process.

Other sources of signals

- From shell: e.g. `kill -9 <pid>`
- Certain terminal characters generate signals, for example \^C.
- Hardware conditions can generate signals (division by zero, memory violation). These signals are passed to the process from the kernel.
- Some software related conditions can cause the kernel to send a signal (for example the receipt of out-of-band data).
Concurrent Servers

- Motivations for Concurrency
  - better response time
  - I/O (and CPU) overlapping
  - multiprocessor systems
  - Nonblocking I/O chew up all processor time

- (Common) Concurrent Server Architectures
  - Multi-process servers
  - I/O Multiplexing servers
  - Multithreaded servers

Multi-process Concurrent Servers

- Design:
  - Master/slave model
    - Master: always running, manage but does not communicate back
    - Slave: does the actual work (processing + comm)

- Algorithm for Connection-oriented servers:
  1. M: Create a socket & Bind to a port
  2. M: Make it passive! (listen)
  3. M: Accept connections & create a slave (S)
  4. M: Go to 3
  5. S: Read, process and reply
  6. S: Close and exit

Multiprocess Concurrent Servers (cont.)

- Algorithm for Connectionless servers:
  1. M: Create a socket & Bind to a port
  2. M: Receive a request & create a slave (S)
  3. M: Go to 2
  4. S: process and reply
  5. S: Close and exit

- A slave per connection Vs. per request
- few connectionless concurrent servers exist!
- Used mainly when slaves are independent (why?)
I/O Multiplexing Concurrent Servers

- **Design:**
  - Single process (apparent concurrency)
  - `select()`: the kernel support for asynchronous I/O multiplexing

- **Algorithm for connection-oriented servers:**
  1. Create a socket (m) & Bind to a port. Add m to `socklist`.
  2. use `select` {blocking, unblocking, timeout}
  3. Accept connections when m is “ready”, and add the new socket to `socklist`
  4. Read/write from/to a “ready” socket
  5. Go to 2

- **Motivations:**
  - No `fork()` overhead
  - Connections share same state or coordination
  - Client requests are very interleaved (eg. X clients: xclock, xbuf, xterm, xemacs ..etc)
  - Supporting multi-protocol single server
  - Serializability
  - Simpler!

- **Issues:**
  - Programmer must be aware of this concurrency
  - Fairness: short requests/service time
  - Multiprocessor machine does not help!

---

Select()

- The `select()` system call allows us to use blocking I/O on a set of descriptors (file, socket, …).
- For example, we can ask `select` to notify us when data is available for reading on either `STDIN` or a TCP socket.
Select()

```
int select(int maxfd,
           fd_set *readset,
           fd_set *writeset,
           fd_set *excepset,
           const struct timeval *timeout);
```

- **maxfd**: highest number assigned to a descriptor.
- **readset**: set of descriptors we want to read from.
- **writeset**: set of descriptors we want to write to.
- **excepset**: set of descriptors to watch for exceptions.
- **timeout**: maximum time select should wait.

```
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
}
```

```
struct timeval max = {1,0};
```

- Implementation is not important.
- Operations to use with `fd_set`:

```
void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
```
Using `select()`

- Create `fd_set`
- clear the whole thing with `FD_ZERO`
- Add each descriptor you want to watch using `FD_SET`.
- Call `select`
- When `select` returns, use `FD_ISSET` to see if I/O is possible on each descriptor.

Multithreaded Concurrent Servers

- Design: (detailed discussion later)
  - Master/slave model with stronger coordination
    - Master: always running, manage but does not communicate back
    - Slave: does the actual work (processing + comm)
- Algorithm for connection-oriented servers:
  1. MT: Create a socket & Bind to a port
  2. MT: Make it passive! (listen)
  3. MT: Accept & create a thread (ST)
  4. MT: Go to 3
  5. ST: Read, process, reply and coordinate
  6. ST: go to 5 or close and exit-thread

Servers Threat

- Resources consumption
  - rapid connections during 2*MSL
  - too many client crashes
- Reliability problems
  - Erroneous message
  - requests storm
- Server deadlock
  - client connects but does not send
  - client send but not receive
  - UDP server sends a request and expects a reply but the request is lost