Constitution Day

Sep 17th

Rationale

- Constitution Day in the US is September 17th
- Signing of the Constitution was Sept 17, 1787
- 2004 Senator Byrd's proposal changes "Citizenship Day" to "Constitution and Citizenship Day"
  - "The second is that each educational institution which receives Federal funds should hold a program for students every September 17th."
- [source]

National Constitution Center

- 1st amendment videos (yes, both of them)
  - https://constitutioncenter.org/learn/constitutional-exchanges/day-three-speech
  - Hopefully we have audio in this room…
- More than just a rehash of the same basic definition you've heard, but history as well
  - Law and interpretations evolve
  - Big tech gets mentioned a couple times, tangentially relevant to net prog?
  - Policy changes based on user opinion, realistic?

Signals

Signals

- A way for the OS (or other processes) to deliver notifications to processes
- Every signal has a number (see "man 7 signal")
- We will use names, SIGKILL, SIGINT, etc.
- Sometimes called "software interrupts"

Catching Signals

- When a signal arrives, a *signal handler* can capture the signal and react
- The handler is a function which gets added to the top of the stack
- Behaviors may be OS-dependent
  - For example, if a call was blocking, do we restart it or not? (more on this later)

Signal Handlers

- void handlername(int signo);
- No return values!
- Only one argument, the signal
- We may have no choice but to use global variables
- For now our handler is pretty simple, calls sigaction() without much setup
  - sigaction() is POSIX compliant but gross to use

Signal Disposition

- Most signals can have disposition SIG_IGN (ignore this signal)
- We can also use SIG_DFL to use the default.
  - Usually results in the process terminating
  - Some signals are ignored by default
- We can't catch SIGKILL or SIGSTOP
- SIGKILL and SIGSTOP can't be ignored either
  - This is why “kill -9” works

signal()

- man 2 signal shows

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

- Unfortunately, if not in GNU C

```
void ( *signal(int signum, void
(*handler)(int)) ) (int);
```

- unp.h uses "Sigfunc"
  - Works out that the signal handler returns void and takes one int argument
- Let's look at some excerpts from the man page

Portability 1/3

- " The behavior of signal() varies across UNIX versions, and has also varied historically across different versions of Linux. Avoid its use: use sigaction(2) instead. See Portability below."

Portability 2/3

- "The only portable use of signal() is to set a signal's disposition to SIG_DFL or SIG_IGN. The semantics when using signal() to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); do not use it for this purpose."

Portability 3/3

- "POSIX.1 solved the portability mess by specifying sigaction(2)"

Signal Set Fuctions

- `int sigemptyset(sigset_t *set);`
- `int sigfillset(sigset_t *set);`

- `int sigaddset(sigset_t *set, int signum);`
- `int sigdelset(sigset_t *set, int signum);`

- `int sigismember(const sigset_t *set, int signum);`

- Lets us manipulate signal sets (e.g. sa_mask)
- Need to initialize first using sigemptyset or sigfillset

Signal Handler Installation

- sigaction() registers our action and puts the old actions in the third argument (&oact for us)
- The book's signal() implementation returns the old action
- In older systems, you would have to "reinstall" the handler every time it was triggered
  - Modern systems don't have this problem

More On Handlers

- When a signal handler registered for signal X is active, that signal is blocked (we cannot catch another X while in X).
- If any signals are in the sa_mask set associated with signal X, they are also blocked while we handle X.
- As we'll see shortly, if signal X is thrown several times before we can catch it, we get one combined signal X instead of a queue of several signal X.

Figure 5.6 from UNP

```
Sigfunc *
signal(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef  SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;    /* SunOS 4.x */
#endif
    } else {
#ifdef  SA_RESTART
        act.sa_flags |= SA_RESTART;      /* SVR4, 44BSD */
#endif
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
/* end signal */
```

Server child terminates?

- Does the system reclaim all the child resources?
- Nope!*

Why zombies?

- The parent needs the chance to review what happened to the child!
- Until this takes place, the child is now a zombie process
- Let’s try not to pollute our system with zombie processes
- Running "ps -t" we can see each process
- Status S = sleeping, Z = zombie, R = running, etc. ("man ps")

Handling SIGCHLD

```
void sig_child(int signo) {
    pid_t pid;
    int stat;

    pid = wait(&stat);
    printf("child %d terminated\n", pid);
}

int main() {
    Signal(SIGCHLD, sig_child);

    int n = fork();
    if (n == 0) // child
        return 0;
    else        // parent
        sleep(2);

    return 0;
}
```

Handling SIGCHLD better

- wait() has problems! We might not catch all of our children!
- Why? POSIX doesn't require that signals are queued
- If we have multiple children terminating around the same time, we may miss some signals

Use waitpid() (Figure 5.11)

- Use -1 to wait for any children
- Use a while loop and the WNOHANG option

```
void
sig_chld(int signo)
{
    pid_t   pid;
    int     stat;

    while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
        printf("child %d terminated\n", pid);
    return;
}
```

"Slow" system calls

- Some calls are "slow", basically synonymous with "blocking"
- When a signal arrives this will usually interrupt the call, unless it has a SA_RESTART
  - Interrupting means run the handler, then stop the syscall and set errno to EINTR
- read(), write(), etc. from terminal, pipe or network, sleep(), wait() will not restart
  - However read(), write() etc. to file I/O aren't as "slow" and may auto-restart after an interrupt!

EINTR

- Some system calls are *slow* i.e., may never return
- Sometimes they return an errno value of EINTR, or interrupted
- You probably just want to try making the same call again
- Using *goto* is fine in this case

goto/EINTR example

```
//some code goes up here
...
//code that may block
do_read:
if((nbytes = read(fd,&buf,len))<0){
     if(errno == EINTR)
          goto do_read;
     else
          err_sys("read error"); //book func
}
//more code below
...
```

alarm() / SIGALRM

- Sometimes we want to interrupt calls (such as…?) after a certain amount of time
- We can use alarm(seconds) to schedule a SIGALRM signal.
  - If there is an existing alarm it is overwritten
  - If seconds is 0, this cancels the alarm, it does not send a SIGALRM.
- Recall the book's signal() implementation does not set SA_RESTART for SIGALRM… why?

SIGPIPE

- We haven't discussed connection-oriented sockets
- When we do have one (such as TCP), it is possible that the receiver will have closed (or crashed) but the sender sends more data
- Normally, there's a reset signal sent to show the socket is dead
- If the client ignores this and keeps writing, a broken pipe signal (SIGPIPE) is raised.
- Default behavior is to return EPIPE.