Signals (HD)

- Move from shell programming to network programming
- Waittimeout.c: first attempt to run a loop until time elapsed (.75 seconds) or exit loop, whichever comes first
- Flaw of code
  - Bad utilisation because it uses polling
- Utilisation is a general metric that can be applied to any system
  - Fraction of some resource devoted to useful work
  - Bigger numbers are better - want high utilisation
  - Utilisation = value between 0 and 1
  - Problem: who decides what's useful?
- CPU: 98% utilised
  - Who knows what’s useful?
    - Eddie: not useful because knows what code does (Tight loop)
    - Kernel: what is useful / not useful
      - Useful work is when process is running
      - Purpose of kernel: perform work on behalf of processes
      - Even though it may be using a lot of the computer, we may know it is not useful even though the kernel may think it is useful
      - Kernel normally doesn’t look into a process to determine if the work is useful
- Blocking: increase utilisation by getting rid of useless work
  - Introduces race conditions
- Waitblock: parent process deliver signal when child process dies
  - Signal: wake up any blocked system call (slow system call)
  - Better to sleep and then wait for an interrupt because sleep won’t use the CPU
  - Use system to call to do blocking, in this case, sleeping
  - wait for child to print status
  - waitblock not using CPU -> less useless work
  - Where is the race condition [bug] (scheduling leads to incorrect outcome) in the code?
    - child exits -> parent exits or child goes past x milliseconds, and parent exits at x milliseconds
    - A: child exits right before sleeping (sleep won’t be interrupted)
- Sleep-Wakeup Race
  - Wake up (stop blocking) when a signal arrives. However, if signal arrives before program sleeps, program will sleep for a very long time (race condition). Solution is to use sigblock, which blocks signals temporarily, then delivers them after process is unblocked. Leaves a race condition after unblocking, but a much less dangerous one.
- Atomic Code: executes indivisibly without interruption
Two things are atomic if they execute as a unit without interruption

- **What if signal runs and then process sleeps?**
  - can only send signal when process is asleep, how to fix?
    - however can’t do it with Linux. Kernel programming doesn’t let us disable interrupts forever
    - sigblock: blocks signals
      - *every process must be killed*
      - use: block, then unblock signal before we sleep
  - Fork only after wake
  - Sigblock(int mask): While a signal is block, it will not be delivered, when it is unblocked it will be delivered except every process can always be killed
  - select (slow system call): will get woken up after delivery of signal (pselect)
    - generic way to go to sleep for specific amount of time
    - Use pselect -> takes in a signal mask, atomically unblocks a set of signals and sleeps, combines checking for an event and going to sleep in a single operation
    - waitblock safe uses a pipe to solve this problem
    - takes in sig mask: unblock signals and sleeps
      - solves sleep wake up race by combining unblock (checking for event) and going to sleep into one atomic operation
      - can block until file system descriptor is readable
        - Bad code:
          
          ```
          if (event_not_happen)
              block forever();
          ```
        - Good code:
          
          ```
          set up foreground pipeline
          put fgpl in foreground;
          if (SIGINT)
              kill fg pipeline
          wait for fgpl
          ```
  - Sigwait
  - sigprocmask

- What’s a zombie process?
  - *nix guarantees that when a process dies, its parent can recover its final status (through waitpid). A zombie process has died, but its parent has not waited yet, so it’s resources are still taking up memory
  - Expect to collect zombie processes; call wait for every child
  - Shell forks child, and parent dies -> Somebody has to wait for the child
  - init(1) -> parent dies child becomes a child of init and it loops and waits for child
  - Pstree- > shows all processes and their parents
Networking: Client-Server Programming

- Message Sequence Diagram
  - time moves down
  - client = active party (sends request)
  - server = passive party (not actively contacting clients, waits for client contact)
  - client makes request -> server responds

- port 80: reserved for unencrypted web servers
- port 443: encrypted web servers
- Seems similar to a pipe but isn't a pipe
- For pipes, need parents to create parent shell before children are forked off
- To open communication, client & server need to
1. Each create socket
2. Client & server agree to create sockets (system call - connect)

- Telnet: 2 direction interactive text connection
- Need a file descriptor to represent a future channel: Socket
  - Creates an endpoint for communication
- A network file descriptor
- Client and server create sockets, and then connect binds them together
- Server must be a well known endpoint
  - To avoid race conditions, we do not use the listening socket for connections
  - Instead we use accept(listenfd)
    - Accept takes in a listenfd and returns a new fd for the connection
    - Connected socket is bound to the clients socket
  - Bad design of having serial number because when you change computers the number changes
- Telnet is very insecure, anyone can tell what you’re typing, uses it to keep open
- Denial of service attack: when a user tries to make a network resource unavailable to other users