Covered in This Lecture:

- *Wait*
- *Race conditions*
- *Blocking & polling*
- *Signals*
- *Pipes*

Lets get to it!

- *Wait* (e.g. waitpid)
    - Parent process waits for **a change in the state of** the child process, for example:
        - Termination of a child process
        - A signal to stop a child process
        - Signals to resume a child process
    - ***Note:*** *only a parent can wait for a child; a child cannot wait for its parent*

- What if we implemented a wait function using pipes instead?...
    - Read end of the pipe can 'wait' for one of these to occur:
        - Child dies and the write-end of its pipe closes → parent starts reading out contents of pipe
        - Child writes a byte to the pipe → parent assumes that the child process is complete and starts reading
        - Child closes the pipe → parent assumes that the child process is complete

- **waitpid**(pid, &status, 0)
    - Blocks until pid change status, sets status, and returns 0
    - Parent processes must wait for one of these conditions to change before resuming

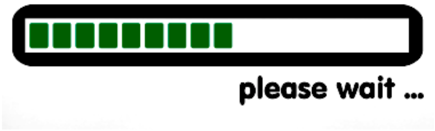
- Implementing a timeout
    - **Timeout**: can tell a process to wait for a specific amount of time, or until the child dies
    - *Example pseudocode for a 0.75 second timeout:*

```
while ( start_time + 0.75s  >=  timestamp ) {
        waitpid(p1, &status, WNOHANG)  ;
}
```

    - If child has exited, this will return 1
    - If there is a timeout, it will return 0

- *Blocking* system call
    - Waits for a single event, *will not return until state change*
    - Advantage – *good CPU utilization* (CPU can do other work in the meantime)
    - Examples – **usleep**(miliseconds), **select**(args)

- *Polling* system call
    - Returns immediately, and returns a different thing once state changes
    - Advantage – *greater control over when to stop waiting* (user can specify wakeup conditions)
    - Disadvantage – *poor CPU utilization*
    - Example: WNOHANG



- Signals
  - Interrupts
  - **usleep**() will end early if it receives a signal from the child
  - Ex: SIGCHILD can allow us to send a signal when child dies
  - Signal handlers
    - Should be prepared to handle immediately and at any time
    - Consequence → *should not make any long system calls* (e.g. a printf)
  - Example signal handler: handle_signal(SIGCHILD, handler);

- *Pipes*
  - Are inaccessible except to the parent and child processes
  - Can create memory leaks if you never close the read end of the pipe

- Example: yes “I love you” | head -n 4
  - Prints the first four lines of “I love you”
  - After the first 4 instances, the read end of the pipe closes and then the process is killed
  - How to make this happen (pseudocode version)
    - **pipe sh**
      - creates pipe and gives read and write ends to shell
    - **fork sh**
      - Now echo is connected to the same pipe on both the read and write ends (but via higher number page descriptors, not standard in and out)
    - **dup2**(4, 1) echo
      - 4 is original place in array that lead to the write end
      - 1 is standard output, where we want to move it
    - **close**(3) **close**(1) echo
      - Pipe hygiene!
    - **close**(4) sh
      - Pipe hygiene!
    - **execvp**(“echo”)
    - **fork sh**
      - Creates child process wc
    - **dup2**
      - Sets standard input of wc to be from the pipe
    - **close**(3)
      - Pipe hygiene!

- Outtakes & extras
  - Useful function: **getppid**
    - Allows child process to find its parent’s id (getpid for running process id)
  - Protip: Draw pictures to help envision a shell’s initial and final state
  - The world’s shortest *fork bomb* (is delicious evil)
    - : ( ) { : | : & } ; :
    - Halts system if run as root,
    - Try it for yourself!... or don’t…