lecture 20: 11/13/14

inter-process communication, pipes

major concepts

- wait
- race conditions
- blocking and polling

wait

- “the most fundamental inter-process communication mechanism that there is”
- “allows one process, the parent process, to discover that its child has died” :(

semantics of wait

- man wait for details
  - protip: avoid naming your functions wait, wait3, and wait4 like mac does
- actual protip: use waitpid rather than wait. why?
  - options allow exploration of race conditions, blocking, and polling
  - allows observation of state change info (death, signal interrupts, etc.)
  - zomg zombies

```
P
/  
P'
  ▲
  getppid()
```

- P is parent, P’ is child
- P knows P’ pid through fork return value
- P’ knows P pid through either:
  - making a copy of getpid before forking
  - calling getppid (man getpid for details)

process hierarchy

- permission requirements
- design restriction: “only a parent can wait for a child process”
- when do we want to wait for a child to complete?
  - shell waits for process to finish
  - program parallelizes task across cores and combines completed outputs
  - main process relies on helper processes
**how to implement wait... without wait? wait what?**

- proclaim the power of pipes!
- **key idea**: parent *(shell)* blocks *(stops running)* until child *(process)* completes

---

when will `read(3, buf, 1)` return?

- if child writes byte into pipe
- if child closes write end of pipe (child is only process that has it open)
- -1 if call interrupted

from this, a proposal:

- if child dies, write end is closed. specifically:
  - fd table is destroyed
  - write end of pipe dereferenced
  - write pipe closed
- *then* `read(3, buf, 1) will return 0 (EOF)`
- so child dies → read returns. does read returns → child dies?

NO! LOGICAL FALLACY! counterexamples:

- child can **close pipe** (which also causes read to return) without dying
- child can **write byte** (which also causes read to return) without dying

proposal is “sufficient” but not “consistent”

---

actual protip 2: when confused with pipe processes, draw pictures!
pipe process picture example: `echo foo | wc -l`

- `echo foo | wc -l`

```
SH61
```

---

- **Initial State**

- `echo`
  - `SH61`
  - `wc`

---

- **Final State**

**yes command example**

- what it does: print out a string to standard output repeatedly until killed
- e.g., `yes “I love you”` (for those lonely moments):
  - I love you
  - I love you
  - I love you
  - I love you
  - I love you
- I love you
- ...
- ^C
- </3

- try: yes "I love you" | head -n 4 (head -n K prints first K lines of output)
  - I love you
  - I love you
  - I love you
  - I love you

- now try: strace -o strace.txt yes "I love you" | head -n 4
  - what do you see?

- we attempted to write to a closed read end pipe
  - EPIPE → error message: no one would hear the love
  - SIGPIPE → killing signal
  - cs is depressing sometimes

- signals: software model of hardware interrupts

- conclusion: if pipe kept read-end open...
  - yes would stick around forever
  - (it might eventually block, but we'd still have a memory leak)
  - actual protip 3: have good pipe hygiene! close your pipe ends when done

pipe process picture example: echo foo | wc -l revisited
how do we get from initial to final state?
(3) dup2(4,1) in echo process

(4) close(3) close(4) in echo

(5) close(4) in shell

(6) execvp

(1) pipe in the shell

(1) pipe in the shell
problems with using pipe for wait
- even if process dies, pipe can remain alive
- process can write to pipe, fooling parent into thinking process had exited
- (recall logical fallacy above)

more robust solution: waitpid
- example usage: waitpid(pid, &status, 0)
- blocks until process with id “pid” changes status
- sets status, returns 0 (or child pid)

coding time: waitdemo.c
- read summary for details
- new idea: implement timeout so that we wait for min(0.75 s, time for child to die)
- man waitpid and examining options:
  - WNOHANG - return immediately if no child has exited
  - with WNOHANG set, waitpid will return 0 until child state has changed

```c
double start_time = timestamp();

// Wait for the child and print its status
// Wait for at most 0.75 sec, or child to die, whichever first
int status = -1;
while (start_time + 0.75 >= timestamp()) {
    pid_t exited_pid = waitpid(p1, &status, WNOHANG);
    assert(exited_pid == p1 || exited_pid == 0);
    if (exited_pid == p1)
        break;
}
```
- set status to initial value (results in abnormal exiting instead of nasal demons)
- waitdemo takes out CPU (claims to be 100% utilized). why?
  - doing work in silly loop
  - WNOHANG is polling, not blocking
“blocking vs polling: a great systems conundrum”

<table>
<thead>
<tr>
<th></th>
<th>blocking</th>
<th>polling</th>
</tr>
</thead>
<tbody>
<tr>
<td>when does it return?</td>
<td>not until state changes</td>
<td>immediately</td>
</tr>
<tr>
<td>why is it useful/bad?</td>
<td>• allows CPU to do other work (e.g., read)</td>
<td>• gives more control about when we wake up (build conditionals)</td>
</tr>
<tr>
<td></td>
<td>• better utilization*</td>
<td>• terrible utilization*</td>
</tr>
</tbody>
</table>

*utilization*: doing “useful” work (but who decides utility? stay tuned for future lecture!)

how can we use blocking instead of polling?

- idea 1
  - usleep(750000) at end of while loop (usleep is blocking)
  - problem: still waits 0.75 s, regardless of when child died
  - usleep blocks for amount of time passed in!
  - idea 1.5
    - usleep for small amount of time
    - okay, but let's try to use just one blocking call

- idea 2
  - use signals (software interrupts)
    - man usleep: ERRORS has EINTR
    - man waitpid: ERRORS has EINTR
    - EINTR can return value for every system call that can block
    - we can use this to wake our sleeping
  - man 7 signal
    - SIGALRM: timer signal
    - SIGCHLD: signal if child stopped/terminated
      - usually signal ignored (assumes we don't like interruptions)
      - this time, we can explicitly make handler for signal
warning: do not put something like `fprintf(stderr, “I love you”)` in your handler!
- signals can be delivered anytime, even in middle of printing
- keep as simple as possible
- only use for waking up system

protip 2: run the world’s shortest fork bomb
- `:(){ :||:& };:`
- protip 2.5: run this as root

now that we’ve created a handler, use it to handle signal:
result: appears to work properly

- child sleeping for 0.5 s → exit after 0.5 s
- child sleeping for 500000 s → exit after 0.75 s
- strace reveals that very few system calls were made
- however...

race conditions

detail:

race condition bug in this example! how to induce:

- get rid of sleep and fprintf in child
- replace fork with nfork (nondeterministically run either parent or child first)
- note: luck might not be on your side, so also add small sleep (5 us) to parent
  - note: this is totally valid for system to do, too

what happens in race condition?

- child died right away, but parent was too busy sleeping (in 5 us) to notice :
- parent then waited 0.75 s even after child died right away - bad!

solution attempts

- move handle_signal to before nfork, so every usleep should be woken up
  - problem still exists! why?
  - child exited before we even started sleeping
- almost fix - use global variable
  - first change handler to change global variable when signal received
- then only sleep if global variable not changed (no signal received)
- race condition bug with very tiny probability, but still exists
- possible to receive signal between conditional jump and usleep!
- solve once and for all with select
  - “fundamental ‘wait for multiple things’ system call”
  - blocks until something happens, generally one of following:
    - data appears to be read
    - data appears to be written
    - data appears to be exceptfds (← no one actually knows what this is)
    - data appears to be timed out
  - here, we can wait for byte to be readable or timeout to occur
- add writing to pipe in signal handler

```c
#include "helpers.h"
int signalpipe[2];

void signal_handler(int signal) {
    (void) signal;
    ssize_t r = write(signalpipe[1], "1", 1);
    (void) r;
}

int main(void) {
    int r = pipe(signalpipe);
    assert(r >= 0);
    r = handle_signal(SIGCHLD, signal_handler);
    assert(r >= 0);
}
```

- then set up timeout, readfds, and let select solve race conditions!

```c
int status;
pid_t exited_pid = waitpid(p1, &status, WNOHANG);
assert(exited_pid == 0 || exited_pid == p1);
if (exited_pid == 0)
    fprintf(stderr, "Child timed out\n");
else if (WIFEXITED(status))
    fprintf(stderr, "Child exited with status %d after %g sec\n",
    WEXITSTATUS(status), timestamp() - start_time);
```

- proclaim the power of pipes!