

lecture 20: 11/13/14

inter-process communication, pipes

major concepts

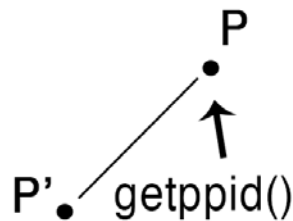
- wait
- race conditions
- blocking and polling

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*



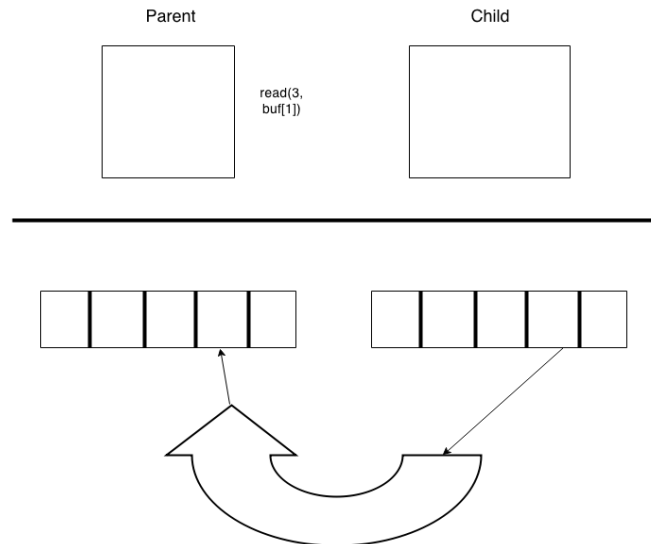
- P is parent, P' is child
- P knows P' pid through fork return value
- P' knows P pid through either:
 - making a copy of `getpid` before forking
 - calling `getppid` (`man getpid` for details)

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?

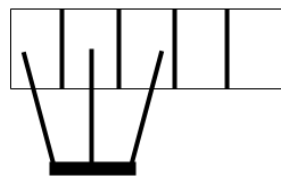
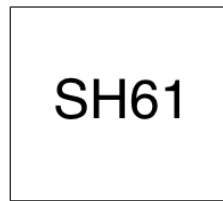
NO! LOGICAL FALLACY! counterexamples:

- 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"

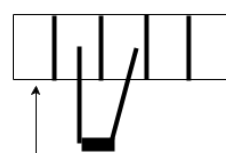
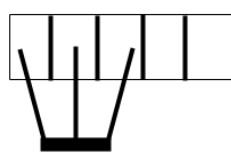
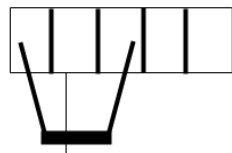
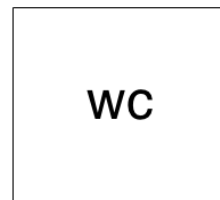
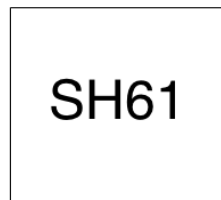
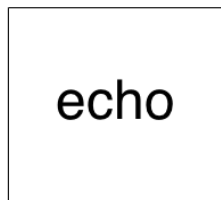
actual protip 2: when confused with pipe processes, draw pictures!

pipe process picture example: echo foo | wc -l

echo foo | wc -l



Initial State



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?

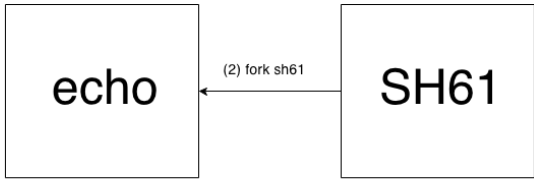
```

kshier@ubuntu: ~/csa1-lectures/72a
kshier@ubuntu: ~/csa1-lectures/72a
mmap2(NULL, 2097152, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb732f000
mmap2(NULL, 4096, PROT_READ, MAP_PRIVATE, 3, 0x2c5000) = 0xb76f7000
close(3) = 0
fstat64(1, {st_mode=S_IFIFO|0600, st_size=0, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb76f6000
write(1, "I love you\nI love you\nI love you"..., 4096) = 4096
write(1, "ve you\nI love you\nI love you\nI l"..., 4096) = -1 EPIPE (Broken pipe)
--- SIGPIPE {si_signo=SIGPIPE, si_code=SI_USER, si_pid=12740, si_uid=1000} ---
+++ killed by SIGPIPE +++
(END)

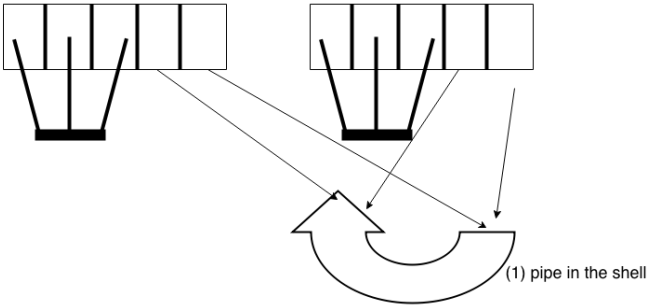
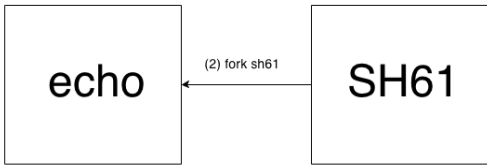
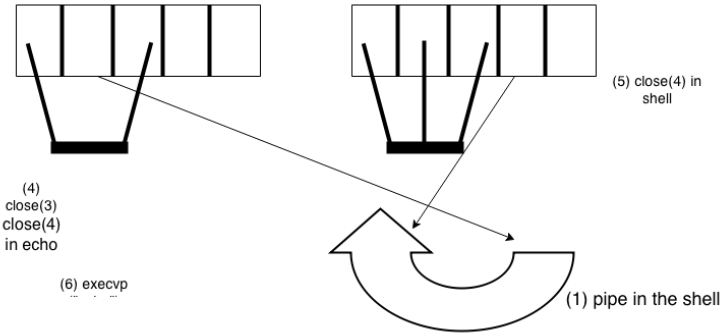
```

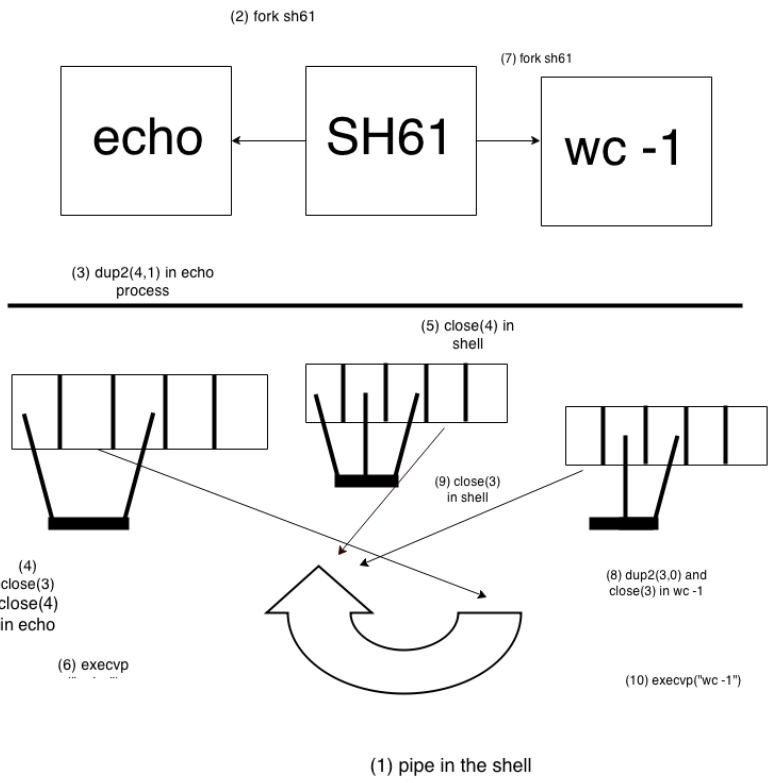
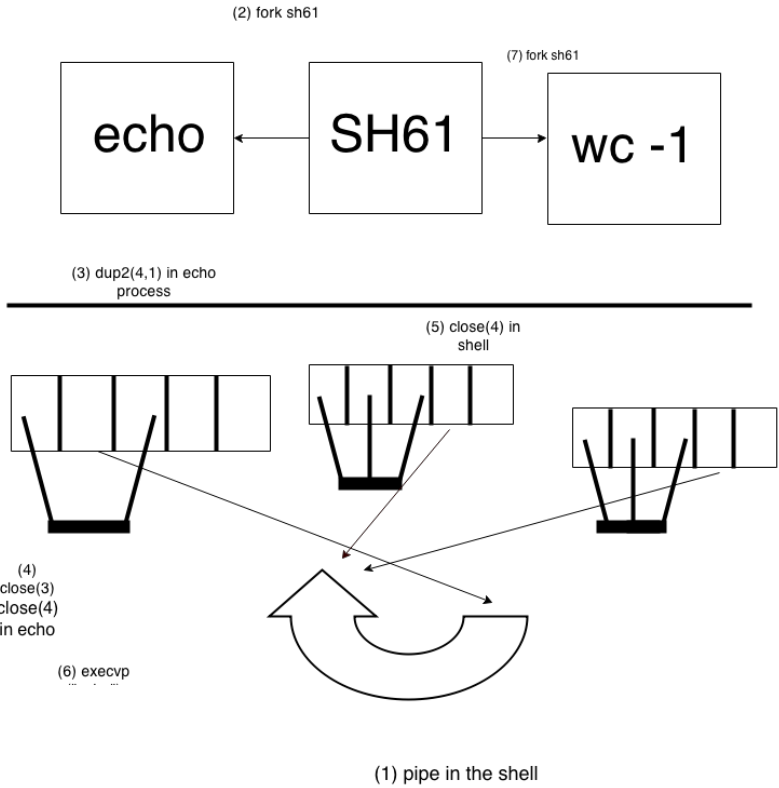
- we attempted to write to a closed read end pipe
 - EPIPE → error message: no one would hear the love
 - SIGPIPE → killing signal
 - cs is depressing sometimes
- signals: software model of hardware interrupts
- conclusion: if pipe kept read-end open...
 - yes would stick around forever
 - (it might *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





problems with using pipe for wait

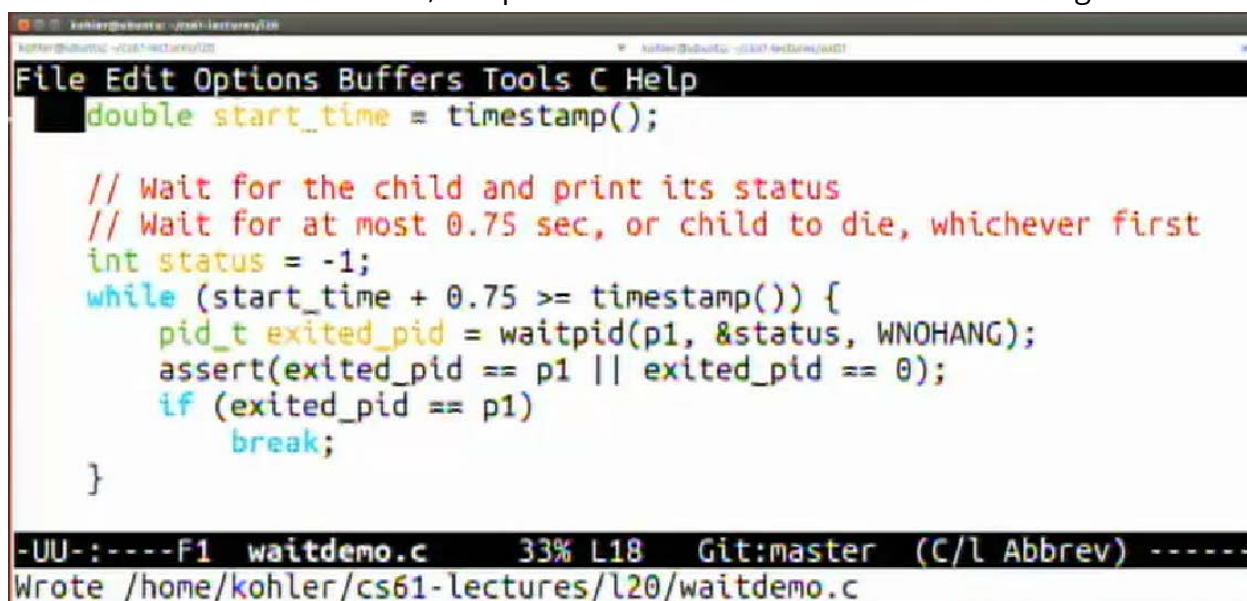
- even if process dies, pipe can remain alive
- process can write to pipe, fooling parent into thinking process had exited
- (recall logical fallacy above)

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



```
File Edit Options Buffers Tools C Help
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;
}

-UU-:----F1 waitdemo.c 33% L18 Git:master (C/l Abbrev) -----
Wrote /home/kohler/cs61-lectures/l20/waitdemo.c
```

- 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”

	blocking	polling
when does it return?	not until state changes	immediately
why is it useful/bad?	<ul style="list-style-type: none">• allows CPU to do other work (e.g., read)• better utilization*	<ul style="list-style-type: none">• gives more control about when we wake up (build conditionals)• terrible utilization*

***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


```
File Edit Options Buffers Tools C Help
#include "helpers.h"

sig_atomic_t variable;
void handler(int s) {
    (void) s;
    variable = 1;
}

int main(void) {
    fprintf(stderr, "Hello from parent pid %d\n", getpid());

    // Start a child
-UU-:***-F1 waitdemo.c Top L6 Git:master (C/l Abbrev) -----
```

- 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:

```
File Edit Options Buffers Tools C Help
double start_time = timestamp();

handle_signal(SIGCHLD, handler);

// Wait for at most 0.75 sec, or child to die, whichever first
usleep(750000);

// Wait for the child and print its status
int status = -1;
pid_t exited_pid = waitpid(p1, &status, WNOHANG);
assert(exited_pid == p1 || exited_pid == 0);
-UU-:----F1 waitdemo.c 36% L25 Git:master (C/l Abbrev) -----
Wrote /home/kohler/cs61-lectures/l20/waitdemo.c
```

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

race condition bug in this example! how to induce:

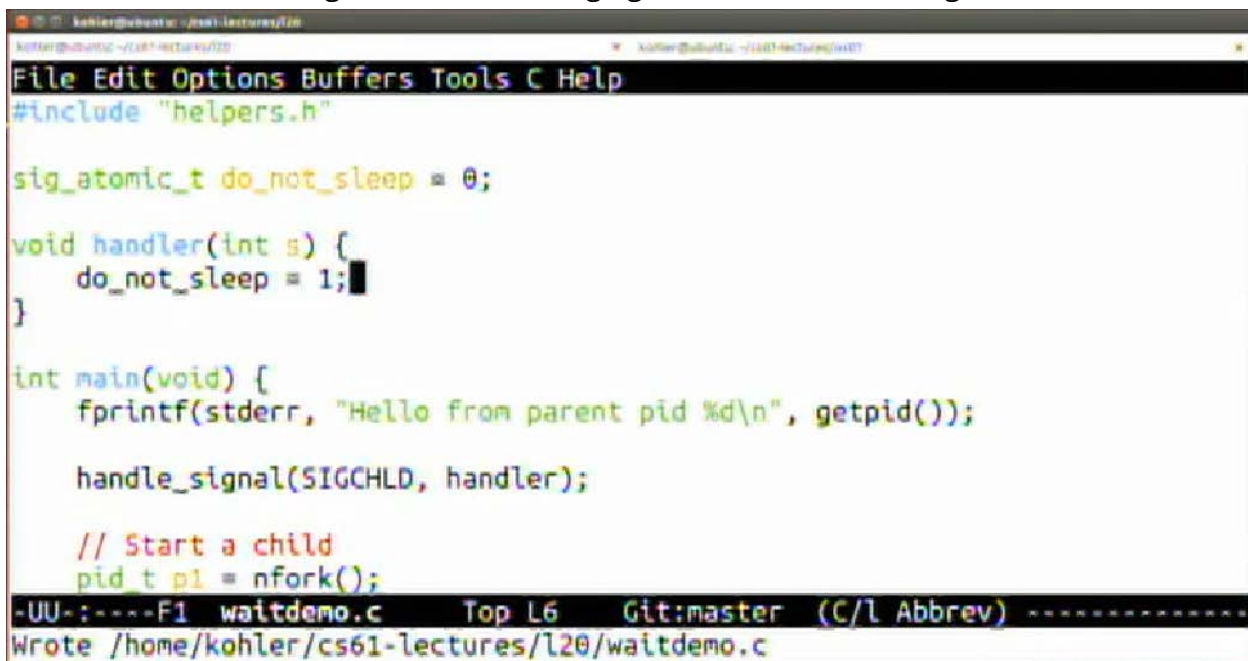
- get rid of sleep and fprintf in child
- replace fork with nfork (nondeterministically run either parent or child first)
- note: luck might not be on your side, so also add small sleep (5 us) to parent
 - note: this is totally valid for *system* to do, too

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



```
File Edit Options Buffers Tools C Help
#include "helpers.h"

sig_atomic_t do_not_sleep = 0;

void handler(int s) {
    do_not_sleep = 1;
}

int main(void) {
    fprintf(stderr, "Hello from parent pid %d\n", getpid());

    handle_signal(SIGCHLD, handler);

    // Start a child
    pid_t p1 = nfork();
-UU-:----F1 waitdemo.c Top L6 Git:master (C/l Abbrev) -----
Wrote /home/kohler/cs61-lectures/l20/waitdemo.c
```

- then only sleep if global variable not changed (no signal received)

```
File Edit Options Buffers Tools C Help
//fprintf(stderr, "Goodbye from child pid %d\n", getpid());
exit(0);
} else
    usleep(5);
double start_time = timestamp();

// Wait for at most 0.75 sec, or child to die, whichever first
if (!do_not_sleep)
    usleep(750000);

// Wait for the child and print its status
int status = -1;
pid_t exited_pid = waitpid(p1, &status, WNOHANG);
assert(exited_pid == p1 || exited_pid == 0);

-UU-:----F1 waitdemo.c 32% L27 Git:master (C/L Abbrev) -----
Wrote /home/kohler/cs61-lectures/l20/waitdemo.c
```

- 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

```
File Edit Options Buffers Tools C Help
#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);
}

-UU-:----F1 waitblocksafe.c Top L1 Git-master (C/l Abbrev) -----
```

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

```
File Edit Options Buffers Tools C Help
// Wait for 0.75 sec, or until a byte is written to `signalpipe`,
// whichever happens first
struct timeval timeout = { 0, 750000 };
fd_set readfds;
FD_SET(signalpipe[0], &readfds);
r = select(signalpipe[0] + 1, &readfds, NULL, NULL, &timeout);

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);

-UU-:----F1 waitblocksafe.c 45% L28 Git-master (C/l Abbrev) -----
```

- proclaim the power of pipes!