```
CS214 / 2021-03-10
==================

Project 1 notes
---------------

Your program does not need to worry about whether its input is a text file
- it should assume any file it is given is a text file
- it should assume that any (regular) file in a directory is a text file
- it is not your program's job to fix the user's mistakes

When we process a directory, we create new wrapped files
- these files are named by adding "wrap." to the start of the input file's name
-> what should happen if we call ww twice on the same directory?
    if "foo" and "wrap.foo" both exist, we don't want to wrap both
        wrapped version of "foo" will be written to "wrap.foo"
        but what happens to the previous "wrap.foo"?
        should we create "wrap.wrap.foo"?
    even if "wrap.foo" does not already exist, what stops us from encountering
    it later?

    - we avoid all these problems by skipping files that start with "wrap."
        -> we assume those have already been wrapped or were created by us
           previously
    - if the output file already existed, we overwrite it
        - that is, we open using O_WRONLY|O_TRUNC|O_CREAT

    -> note that this only applies when processing a directory!

- what permissions should we give the new files?
    - when we call open with O_CREAT, we must specify the mode (permissions)
        - open will assume the third parameter is there, and bad things will
          happen if we omit it
          - C is inherently unsafe! beware!
    - it is not super-important, since the user can always change it
    - there is no reason not to give the user read and write permission
    - there is no reason to give anyone executable permission

    there is a feature called "umask" which limits the permissions we can give
        - on the iLab, a program cannot give group- or world-write permission
    - So it is safe to give everyone read+write, or just the user

        Reasonable choices:
            0600   rw-------
            0644   rw-r--r--
            0666   rw-rw-rw-


filenames vs paths
------------------

Briefly: a file name is its entry in the directory listing (d_name)
    -> file names are only unique within a directory
    -> a file name by itself is useless unless we know what directory it is in

Most programs and libraries that work with file names actually work with paths
-> a path says how to find a file from a particular starting point

    Absolute paths begin with / (the root of the directory tree)
        /foo/bar/baz/quux

        says "quux" inside "baz" inside "baz" inside "foo" inside the root

        browse the root for foo, browse for for bar, etc.

    Absolute paths uniquely identify a file
        (an absolute path can only identify one file)

    Absolute paths can be long, and aren't necessarily convenient to work with

Thus we have the idea of the "working directory" or "current directory"
    relative paths are interpreted relative to the working directory

    "foo/bar/baz"
        "baz" inside "bar" inside "foo" inside the working directory

Shell commands that work with files use paths (relative or absolute)
    In the shell, pwd tells us the working directory
        cd changes the working directory

C library functions that work with files use paths (relative or absolute)
    getcwd tells us the working directory
    chdir changes the working directory
    When our program starts, the working directory is the same as our parent's
        (e.g., the shell)

Exercise: write a program that calls getcwd and prints out the working directory
    -> then call that program from different locations
    compare:
        $ foo/my_pwd         <- calling my_pwd while in its parent's directory
        $ ./my_pwd           <- calling my_pwd in the same directory
        $ ../my_pwd          <- calling my_pwd from a subdirectory

These functions are set up to do the "expected" thing
    if my program takes a file name as an argument, I pass that name unchanged to
        stat, open, opendir, etc.
        -> it does not matter if it is a relative path or an absolute path
        -> the functions will do the expected thing consistent with Unix conventions

What about if I want to open a file in a different directory?
    -> construct a path!
        concatenate the directory name and the file name, with a slash
            path_to_dir/file

            path_to_dir can be a bare directory name
                or a relative path
                or an absolute path
            it does not matter, because it will work the same in all cases

How do you concatenate strings?
1. allocate space for the concatenated string
2. copy data into the new string
    we could use strcat, but if we already know the lengths of the strings,
        just use memcpy twice


fork and exec
-------------

Where do processes come from?

Last time, we discussed how to create a process using fork

    pid_t fork();

        fork duplicates our process, or makes a copy of our process (*mostly)
        the new process that is created is running the same program
            and has the same contents of memory and registers, including the PC
        meaning that the child will behave as though it had been the parent all along

    the difference between the child and the parent:
        in the parent, fork will return the PID of the child
        in the child, fork will return 0

    pid_t p = getpid();
    pid_t c = fork();

    // check for failure
    if (c == -1) {
        perror("fork");
        abort();
    }

    if (c == 0) {
        // do the child thing
        printf("I am %d, my parent is %d\n", getpid(), p);
    } else {
        // do the parent thing
        printf("I am %d, my child is %d\n", p, c);

        ...
        wait(NULL);  // don't orphan our child!
    }



    pid_t getpid(void)
        return PID of the current process



This by itself is not extremely useful
    -> we can use this to take advantage of multiple processors
    -> or split up work that is IO bound

    -> this is called "multiprocessing" (because it involves multiple processes)

We can use functions like pipe to allow our processes to communicate

    int pipe(int pipefd[2]);

    -> creates two files (streams) and stores their file descriptors in the array

    int fd[2];
    pipe(fd);   // returns -1 on failure

        // anything written to fd[1] can be read from fd[0]

    If we call pipe before fork, then both processes will have access to both ends
        of the pipe

    int fd[2];
    pipe(fd);  // create a stream with two ends
        // fd[0] - read end
        // fd[1] - write end

    pid_t child = fork();

    if (child == 0) {
        close(fd[0]); // close the child's copy of the read end

        write(fd[1], "Hello!", 5);
        close(fd[1]);
        exit(EXIT_SUCCESS);  // stop here; don't do the stuff the parent will do
    }

    close(fd[1]); // close the parent's copy of the write end!
        // we won't get EOF on fd[0] until every copy of fd[1] has been closed

    char buf[100];
    int r;
    while ((r = read(fd[0], buf, 100)) > 0) {
        // do something
    }

    close(fd[0]);  // close read end
    wait(NULL);    // wait for child to exit (prevent zombie orphan)


-> for 2-way communication, we want two pipes
    -> having multiple processes writing to or reading from the same file can be
        unpredictable


But the main reason to use fork is to start a different program

    1. use fork to spawn a new process
    2. have the child process use exec to switch what program it is running


execl and execv are functions that change the current program
    -> the process stays the same, but its execution state is reset and
        its code changes to the new program

    -> this stops executing the current program and starts executing a different program
    -> most program attributes are preserved
        -> such as the list of open files
        -> this is also why standard input and standard output are shared with the shell
            (by default)
            that is, your program inherits files 0, 1, and 2 from the shell

-> this is the mechanism that the shell uses to start programs
-> this is how every program starts
    -> every program except init starts when its parent forks and execs

Both execl and execv specify the program file and the argument list

int execl(char *program, ... /* additional arguments followed by NULL */);

    execl("/bin/echo", "/bin/echo", "Hello", "world!", NULL);
        these will become  argv[0]  argv[1]   argv[2]

        this starts the program /bin/echo and passes 3 arguments (argc = 3)
        By convention, the first argument should be the same as the program file

    execl takes multiple arguments, each of which is a (terminated) string
        the last argument to execl must be NULL
        (this is how execl knows to stop looking for additional arguments)

        these populate argv

    We can use execl to start any program (that we have permission to execute)
        we can give execl any strings arguments, including spaces and special
            characters
        (this does not go through the shell!)

    memory tip: execl takes a list of arguments in its argument list

execv takes a vector (array) of arguments

int execv(char *program, char **argv)

    argv is an array of pointers to terminated strings
    the last entry in argv must be NULL


    char *args[] = {"/bin/echo", "Hello", "world!", NULL};
    execv("/bin/echo", args);

        // args will become argv in the new program (argc is derived from it)

Choice between execl and execv is just convenience; no other difference

Note: execl and execv replace the current running program!
    These functions do not return (unless they failed)
    There is no way to resume the current process

    Like exit, exec does not return
    But fork returns "twice"
        -> so we can use fork to spawn a child and have the child exec the program we want



Typical scenario: run another program and wait for it to finish

    pid_d child = fork();
        // FIXME check for -1

    if (child == 0) {
        execv(program_path, args);

        // if we got here, then execv failed
        perror("exec");   // print a message indicating what went wrong
        abort();
    }

    int wstatus;
    wait(&wstatus);
        // FIXME should check return value

    // wait blocks until a child process halts
    // (wait returns -1 if there are no child processes or it had some other problem)
    // wait will write information about the halted process to wstatus

    if (WEXITSTATUS(wstatus) != 0) {
        // something went wrong with the child
        // WEXITSTATUS() is a macro that extracts the exit code
        // there is other information in wstatus, but we usually don't need it
    }




Common notes to put in comments
// FIXME
// TODO



How does the shell do pipes and redirects?
    - normally a process started from the shell gets the same stdin and stdout as
      the shell (e.g., the terminal)
    - but if we use file redirection or a pipe, the process gets different
        stdin and stdout

    - is this something we can do ourselves?
        - can we start a process and have its stdin read from a file?
        - can we start a process and read what it writes to stdout?


Recall:
    when we fork, the child gets copies of all the open file entries
        -> not copies of the files, just the file entry
        -> so the parent and child can read/write the same files
    When we exec, the new process will retain its open files (*usually)

If I want to start a process and read what it writes to stdout we can
    1. use fork to spawn a child process
    2. somehow change file descriptor 0
    3. exec the program

int dup2(int oldfd, int newfd);
    this duplicates an open file descriptor
    if it succeeds, oldfd and newfd will refer to the same file
        -> if newfd is already open, it closes it


    dup2(x, y);
        now x and y both refer to the same open file
        specifically, y now refers to the same file as x

1. use pipe to create a two-ended stream
2. fork
3. in the child, use dup2 to change stdout to be the write end of the pipe
4. in the child, exec the new process
5. in the parent, close the write end of the pipe & read from the read end

    // FIXME: should check for errors after all of these syscalls

    int fd[2];
    pid_t child;


    pipe(fd);
    child = fork();

    if (child == 0) {
        close(fd[0]);
        dup2(fd[1], 1);   // make stdout the same as the write end of the pipe
        execv(my_prog, my_args);
    }

    // in parent
    close(fd[1]);

    // now we can read from fd[0] whatever my_prog wrote to standard output

    ...
    close(fd[0]);
    wait(...);   // always wait once per fork!


---

pid_t p1 = fork();    // spawns first child
pid_t p2 = fork();    // spawns second child (from parent) and third child (from first child)

-> results in 4 processes
    the original parent
        two children
            one child of the first child

parent
    p1 identifies first child
    p2 identifies second child

first child
    p1 is 0
    p2 identifies third child

second child
    p1 identifies first child
    p2 is 0

third child
    p1 is 0
    p2 is 0


    parent (pid 10)
       |
     p1 = fork() ----------------------------> first child (pid 11)
       |                                           |
     p2 = fork() ----> second child (pid 12)     p2 = fork() -----> third child (pid 13)
       |                   |                       |                      |
     p1: 11            p1: 11                    p1: 0                  p1: 0
     p2: 12            p2: 0                     p2: 13                 p2: 0
```

The value of pid can be :

1. Less than -1 : Meaning wait for any child process whose process group ID is equal to the absolute value of pid.
2. Equal to -1 : Meaning wait for any child process.
3. Equal to 0 : Meaning wait for any child process whose process group ID is equal to that of the calling process.
4. Greater than 0 : Meaning wait for the child whose process ID is equal to the value of pid.

WIFEXITED and WEXITSTATUS are two of the options which can be used to know the exit status of the child.
WIFEXITED(status) : returns true if the child terminated normally.

WEXITSTATUS(status) : returns the exit status of the child. This macro should be employed only if WIFEXITED returned true.

main.c - Tutorial - Code - OSS

main.c

main.c > main(int, char * [])

RUN

VARIABLES

WATCH

```c
    if (pid == 0) {
        // Child process
        int file = open("pingResults.txt", O_WRONLY | O_CREAT, 0777);
        if (file == -1) {
            return 2;
        }
        int file2 = dup2(file, STDOUT_FILENO);


        int err = execlp("ping", "ping", "-c", "1", "google.com", NULL);
        if (err == -1) {
            printf("Could not find program to execute!\n");
            return 2;
```

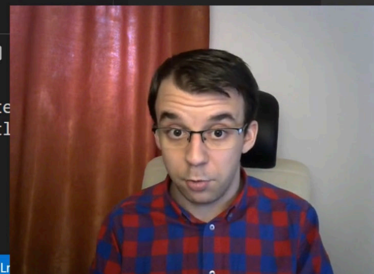
Could also not be one

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

2: cppdbg: main

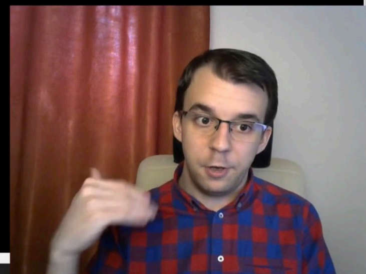
```
PING google.com(bud02s24-in-x0e.1e100.net (2a00:1450:400d:803::200e)) 56 data byte
64 bytes from bud02s24-in-x0e.1e100.net (2a00:1450:400d:803::200e): icmp_seq=1 ttl
e=8.16 ms

--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 8.162/8.162/8.162/0.000 ms
```

nas.local:1919 ⊗ 0 ⚠ 0 gcc - Build and debug active file (Tutorial)



Untitled - paint.net 4.2.10

file descriptors	file
0	STDIN
1	STDOUT
2	STDERR
3	pingResults.txt
4	



Drag the handle to move the text. Hold Ctrl to hide handle and cursor.

Example

The following example shows the usage of atexit() function.

Live Demo

```
#include <stdio.h>
#include <stdlib.h>

void functionA () {
   printf("This is functionA\n");
}

int main () {
   /* register the termination function */
   atexit(functionA );

   printf("Starting  main program...\n");

   printf("Exiting main program...\n");

   return(0);
}
```

Let us compile and run the above program that will produce the following result −

```
Starting main program...
Exiting main program...
This is functionA
```

Previous Page Print Page Next Page

```c
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

void onexit(){
     puts("hi iam called before termination");
     getch();
     }

int main(){
    int counter = 1;
    if(atexit(onexit) != 0){
                     puts("failed to register onexit as the termination function"
                     }
       while(1){
                printf("%d\n",counter);
                if(counter == 10){
                           exit(0);
                           }
                counter++;
                }


   return 0;
}
```

```c
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
/*
int atexit(void (*func)(void));
void exit(int exit_code);
void _Exit(int exit_code);

*/

void onexit(){
     puts("hi iam called before termination");
     getch();
     }

int main(){

    if(atexit(onexit) != 0){
                    puts("failed to register onexit as the t");
                    }

   return 0;
}

```

Learning Lad

```
CS 214 / 2021-03-24
===================

Assignment 3 preview:

You will need to
    - fork a child process
    - use execl or execv to run another program
    - hard mode:
        - redirect the program's standard output to a pipe
        - that is, the parent must read the output of the child

Simple way to send input from one fd to another
    int infd, outfd, bytes;
    char buf[SIZE];

    while ((bytes = read(infd, buf, SIZE)) > 0) {
        write(outfd, buf, bytes);
    }

----

recap:
    traps / hardware interrupts
        - a mechanism that allows the CPU to react to something immediately
        - suspends the current process and switches to some designated OS code
        - when the trap handler completes, OS can resume the suspended process
        - not something that user programs deal with directly

related idea: signals
- mechanism for communicating with a running process
- signals are sent to a process (from OS or other processes or the same process)
- they start out as "pending"
- normally, after some short period, they are "delivered" to the process

- for each signal, we declare a "disposition"
    - block the signal (leaves it pending; may get delivered later if signal is unblock)
    - ignore the signal
    - terminate process
    - execute a signal handler

a signal handler is a function that will be called when the signal is delivered

We can declare our own signal handlers using signal

       #include <signal.h>

       typedef void (*sighandler_t)(int);

       sighandler_t signal(int signum, sighandler_t handler);

manual for signal function:    man 2 signal
manual for signals in general: man 7 signal

signal registers a signal handler
    - There is a table somewhere that holds our process's disposition for each
        signal
    - When the OS delivers a signal, it calls the appropriate signal handler

signal(int signal_number, sighandler_t signal_handler)

    sighandler_t is usually a void function that takes an int
    or it could be:
        SIG_DFL  - the default handler for this signal
        SIG_IGN  - ignore this signal

When our process starts, it has a default disposition for every signal
    - what that will do depends on the specific signal
        - some terminate the process
        - some terminate the process and create a core file
        - some stop the process (but it can be resumed later)
        - some are ignored

    man 7 signal lists the default behaviors


To register a signal handler:

    sighandler_t prev = signal(SIGINT, interrupt_handler);

    prev is the previous signal handler, or SIG_ERR

    -> any time you register a signal, you should check for SIG_ERR

To ignore ^C

    if (signal(SIGINT, SIG_IGN) == SIG_ERR) {
        // didn't work
    }

    // now we are immune to ^C
    // not usually a good idea (makes it harder for user to stop the program)
    // if we block SIGINT, then users will have to use SIGKILL to stop our program
    //      SIGKILL is what kill -9  or kill -KILL sends
    // we cannot set a handler for SIGKILL, so we can't do any cleanup if we
    //    receive it

-> common technique is to intercept SIGINT, set a global variable, and then
    shut down cleanly

notes on signal handlers
-> signal handlers are functions that may be called from anywhere
    - normally, the current function is interrupted and the signal handler
        is added to the call stack as though it had been called
    - it is a function call that could happen at any time

-> it generally isn't a good idea to do a lot of work in a signal handler
    - not all library functions are safe to call from a signal handler
        - e.g., you might get the signal in the middle of calling printf
            calling printf again may cause problems
    - you may get another signal while you are executing the signal handler
    - normally, a signal handler will not be interrupted by itself

signal does not behave consistently across Posix implementations
    - the main difference has to do with what happens if you receive the same
        signal while the signal handler is running
        possibility 1: use the default handler for that signal
        possibility 2: block the signal until the handler completes
    - portable code cannot assume which of these is being used
    - GCC on the iLab uses #2

For portable code, use sigaction
    - Posix (may not be available on non-Posix systems)
    - more powerful and flexible
    - more work to use

GCC Manual discussing signals
    https://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html


Blocking signals
- in addition to signal handlers, our process has a "signal mask"
    - for each signal, the mask says that signal is blocked
    - a blocked signal stays pending; it does not get delivered
    - we can change the mask while we are running
    - if we unblock a signal, and there is a pending message for that signal,
        we receive it at that time

Waiting for signals
    int pause(void);

    pause suspends the current program until a signal is received
    it returns the signal that was received, or -1 on error

    e.g., if we didn't have sleep, we could implement our own using
        alarm(some_time);  // set an alarm -> receive SIGALRM when it is up
        pause();           // wait until a signal is received

    -> be sure to override the default behavior for SIGALRM first!

Termination signals
Different signals are used to terminate our process in different circumstances
SIGHUP - "hang up"; sent if the shell that started our process ends
            (e.g., we closed the window or logged out)
SIGINT - "interrupt from keyboard" - user typed ^C
SIGTERM - "terminate", default signal for kill
SIGQUIT - "terminate and dump core"; used for debugging (type ^\)
SIGKILL - "kill", terminate without cleanup
            processes cannot handle or ignore SIGKILL; it always terminates

Stop and continue
SIGSTOP - stop signal; cannot be handled or ignored
SIGTSTP - "typed stop"; sent when user types ^Z; can be handled
SIGCONT - continue signal; sent when resuming a stopped process

Many other signals terminate by default
- usually because our program hit an error and can't safely proceed
    - division by zero
    - illegal memory access
    - malformed instruction
    - others (e.g., arithmetic errors)

- be careful when writing a handler for error condition
    - the error condition will still be there if the handler resumes
    - the only safe thing you can do is terminate the process or jump to somewhere
        else in the program (siglongjmp)

Aside: stopping and restarting processes in the shell

    Use ^Z to stop the current process
        Usually prints a message like: [1] Stopped  your_program
        1 is the "job number"
            use jobs to get the current list of jobs
            processes started by the shell have job numbers
            all processes have PIDs
        Stopped is the current state of the process

    To restart the process, use fg or bg
    fg [job number]
        resumes process in the foreground
    bg [job number]
        resumes process in the background (concurrent with shell)

    Either give job number, or leave it out
        defaults to the most recent job

    To start a process in the background, put & at the end of a command

        $ ./long_process_to_finish &
        [1] Running  ./long_process_to_finish
        $

    kill sends a signal to a process

        kill PID   - send SIGTERM to process PID
        kill -KILL PID - send SIGKILL to process PID

        Use %N to get the PID of job N

            $ ./long_process &
            [1] 10234
            $ kill %1
            $
            [1]+ Done    ./long_process

-----

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

volatile int signo = 0;
    // marked volatile because it may change asynchronously
    // that is, signals may be received at any time


// very simple signal handler
// note that it returns normally, so we can only use it with non-error signals
void handler(int signum)
{
    signo = signum;
}

// a very simple exit handler
void make_a_note(void)
{
    puts("We are in the exit handler");
}

int main(int argc, char **argv)
{
    // register an exit handler: make_a_note will be called after main returns
    atexit(make_a_note);

    // register some signal handlers
    // we can reuse handler because it will receive the signal
    signal(SIGHUP, handler);
    signal(SIGINT, handler);
    signal(SIGTERM, handler);
    signal(SIGCONT, handler);

    pause();  // stops the process until a signal is received
    // alternative:
    // while (signo == 0) { puts("Waiting"); sleep(1); }

    if (signo > 0) psignal(signo, "caught signal");

    return EXIT_SUCCESS;
}
```

```
CS 214 / 2021-03-22
===================

Recap: fork and exec

- fork "returns twice" in two different processes
    fork creates a duplicate of the current process
        the duplicate is called a child, the original is the parent
        both start with the same contents of memory, program counter, etc.
        as though both processes had been running since the start and
            behaving identically
    in the child, fork returns 0
    in the parent, fork returns the PID of the child

    the parent must use wait to clean up the child's PCB
        this also tells the parent how the child exited
            - what was the exit status
            - was the child terminated by a signal

- execl and execv change what program a process is executing
    specify file name for program we want to start executing
    provide arguments explicitly
    much of our other process information is preserved
        - e.g., open files

    exec *changes* the current program
        what we are currently doing is not preserved
        exec does not return (unless it failed)

we can take advantage of this to do things like
    - spawn a process and read what it writes to standard output

- use pipe to create a pair of file descriptors (read end and write end)
- use fork to start a child process
    - the pipe is now shared between the parent and child
- in the child
    - use dup2 to set the write end of the pipe to be standard output
    - use execl or execv to start our new program
    -> anything the new program writes will get sent to the pipe
- in the parent
    - read from the pipe to see the child program's output
    - wait for the child once we are done reading

- for two-way communication, use two pipes


recap: wait
    wait pauses the process until a child process ends
        -> if a child process has already ended, it returns immediately
        -> if there are no child processes, it returns an error
        -> if there is more than one child, it waits for the first one to end

    related:
    - timedwait if you don't want to wait more than some amount of time
    - waitpid if you want to wait for a specific child


{
    pid_t child1 = fork();
    if (child1 == 0) { ... execv(program_1, p1args); ... }

    pid_t child2 = fork();
    if (child2 == 0) { ... execv(program_2, p2args); ...}

    // at this point, we have two child processes (assuming no errors from fork)
    // we have to call wait twice to clean up both children

    // the two children run concurrently with us

    pid_t finished[2];

    finished[0] = wait(NULL); // wait for one child to exit
    finished[1] = wait(NULL); // wait for other child to exit

    // finished tells us which child ended first
    if (finished[0] == child1) {
        puts("Child 1 was faster");
    }
}


    pid_t wait(int *wstatus);

        return value -> PID of next child to terminate (or -1)
        exit information will be written to wstatus
            use macros like WEXITSTATUS to get information from wstatus

reminder: fork returns the PID of the child process
    to get the exit status of the child, we must use wait

    if (fork() == 0) {   // FIXME we should also check for -1
        // do child stuff (e.g., exec)
        exit(1); // just to make sure we terminated (shouldn't be necessary)
    }

    int wstatus;
    pid_t finished = wait(&wstatus);  // wait for child to finish
        // FIXME: should be checking for error return value

    // finished contains PID of child
    // WEXITSTATUS(wstatus) is the exit status of the child

What happens if we call fork multiple times?

    fork();
    fork();
    fork();
    fork();

    -> this would result in 16 processes


        parent
        |-------------------------
        parent                   1
        |------------            |------------
        parent      2            1           1.1
        |-----      |------      |------     |-------
        parent      2     2.1    1     1.2   1.1    1.1.1
        |----       |--   |---   |--   |--   |--    |--

danger: beware the "forkbomb"
    don't create infinitely many child processes

    while (1) fork();   // never a good idea

    exhausting the open process table can cripple the OS
    iLab has protection against users starting too many processes

-----

How can we have multiple processes on a single-processor system?
- "time sharing"; some method for switching between running processes
    - cooperative multitasking
        ("task" means "process" in this context)
        each process runs for a bit and then yields control to the OS
        the OS then resumes another process
        problem: uncooperative processes can monopolize the CPU

    - preemptive multitasking
        OS sets up a timer
        each process runs for a short period of time
        CPU interrupts program and returns control to OS
        OS lets the next process have a slice of time

essential difference is who controls when we switch processes
- which is better?
    - cooperative multitasking is vulnerable to bad programs and bugs
        one infinite loop can lock up the whole computer
    - preemptive multitasking requires more hardware support
        preemption may occur at awkward times
        e.g., a real-time system can't predict when it will be preempted

How does preemption occur?

-> Hardware interrupts or "traps"
    there are a bunch of related/similar ideas that have used different
    terms in different contexts, or used the same term in different ways

Basic idea: something happens where the CPU needs to respond to it immediately
    - e.g., data from an IO device arrives
    - run-time exception (division by zero, bad memory access)
    - attempt to execute ill-formed/invalid instruction

If something happens, the CPU may interrupt the current process and
transfer control to the OS
    - current process state is saved
    - control switches to OS code at a specified address (a "trap handler")
        -> trap handler will do something in response
        - copy data from IO device to a buffer in memory
        - terminate process with error condition
        - do nothing and resume process

For example, if we try to dereference a bad pointer (e.g., NULL)
    the CPU notices the attempt to read from an invalid address
    it switches to the trap handler for bad address errors
    the trap handler terminates our process with a SEGV signal

Typically, we (user programs) do not ever see traps or set trap handlers
    -> this is reserved for the OS

The OS can set alarms that will trap after some amount of time
    -> e.g, after 200 ms
    the trap handler can suspend the current process and have the scheduler
        resume another process

    -> thus, preemptive multitasking

    preemptive multitasking requires some hardware support, and is a bit
    more work than cooperative multitasking, but it is generally safer and
    more predictable


next time: signals
    signals are a way to interrupt a process
    processes can designate signal handlers that deal with signals when they arrive

start of multithreading
```