```
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
```