Lecture 22 Scribe Notes
19 November 2013
Todd Lubin, Lydia Chen

Problem Set 5

Grammar notation

- PEMDAS: acronym for order of operations (parentheses, exponential, multiplication and division, addition and subtraction)
    - How do we let a computer know how to parse (evaluate) an expression like x * y + z / 4
        - We don't, for example, do it from left to right
        - A grammar expresses the rules for evaluation in a concise form. In EBNF grammar, we have two parts: a start symbol and a set of productions
            - Ignoring parentheses and exponentials:
                - An expression (expr) derives either
                    - term
                    - expr "+" term
                    - expr "-" term
                - A term derives either
                    - atom
                    - term "*" atom
                    - term "/" atom
                - An atom is either
                    - variable
                    - constant
            - You would also need to further define a variable and a constant. But for loose grammar (as in the pset), this is not necessary
            - The start symbol for this grammar is expr.
        - Ex: Convert x * y + z / 4 into expr
            - Match with expr + term
                - Expr must match x*y and term must match z/4
            - And so on
            - Notice this grammar gives us the grouping that we want. Plus and minus are closer to the start symbol and therefore less binding
        - How can we change this grammar to support parenthesis?
            - At the atom level, add an additional choice: '(' expr ')'
- The purpose of the grammar in the pset is to show that the & applies to the entire group
- In a sequences of commands with pipes, the pipes bind more than && or ||

High-level ideas

- **Signals**: Unix process-level abstraction for interrupts
    - Powerful and a little crazy
    - Every signal has a name and a number. Some are:

        - SIGKILL – 9 – Destroys the process without recourse
        - SIGALRM – Alarm
        - SIGCHLD – Child changed status
    - Default behavior of signal is to kill the process and then print a message to the console about why it died. Will have to handle the signal (see waitalarm.c function signal_handler)
    - Once a process receives a signal, it moves on. If it is in the middle of a system call, the system call will return early (stop blocking)
- **Race Condition**: a bug that is dependent on scheduling

Lecture 22 Repo, wait.c

- usleep(x) sleeps for x number of microseconds
- **What we want to do**: when using waitpid, implement a time out. Should wait at most $\frac{3}{4}$ second for the child process to end
    - Suggestion: enter a loop. When we fork, take the time and continually check to see if we've been waiting long enough
        - What happens: the child still exited before we returned (still waiting the amount of time in usleep)
        - But we still get the message that the parent was bored
        - If we give waitpid a last argument of 0, it blocks until something happens.
            - Other system calls that can block:
                - Read will block when reading from a pipe that has nothing in it, or when a disk file has not yet been brought into memory
                - Write can also block
                - Any atomic system call would block other processes
            - But we can also give the WNOHANG flag as the last argument, and then waitpid will not block
    - However, we still have a problem: if we now make the usleep time shorter than the maximum wait time, we don't break out of the loop that checks the wait time
        - Need another condition to break out of the loop
        - The loop doesn't block because we have no blocking system call (WNOHANG flag)
    - Another problem: using a lot of CPU by constantly checking to see if the child has exited, if the wait time is very long
        - How can we accomplish this without so much wasted CPU time?
        - Suggestion: we could sleep and check, rather than checking obsessively. Could probably get a fairly good decrease in CPU usage even if sleep is very short
            - Problems: this isn't completely blocking. Moreover, it's not quite as responsive as we might like (e.g. if the child exits while the parent process is sleeping)
        - Remember that a process is an abstraction of an entire computer.
            - Illusion: process is in charge of all of the computer's resources

- If we were the kernel, how would we solve this problem?
- Analogy: We have a program that is waiting for an arbitrary event that is coming from the outside world. What part of the computer models the event that might come from the outside world? → Interrupts.
    - The child exiting is essentially an "interrupt" because of process isolation
    - We can therefore use the "**signal**" model
- In the kernel we have timer interrupts. Is this possible in a process?
    - Use setitimer to set the alarm

- The waitblock.c solution uses the SIGCHLD interrupt signal to interrupt the parent process' sleep. (if sleep not interrupted, then the parent process exits after sleep ends)
    - But what would happen if the child exits right away? Depending on which process acts first, the parent might continue on to wait for .75 seconds, never receive the SIGCHLD interrupt, and assume the child never exited
    - This is called a **race condition**: a bug that is dependent on schedule
        - Problem may appear or disappear by tweaking the parent process' timing
    - We don't want to call usleep if the child has already exited
        - Set global variable to true if the signal has occurred. Sleep only if the signal has not occurred.
    - However, this is still not free of race conditions. There is a time between instructions that the signal could be delivered, causing the parent to wait too long.
        - There is a list in `man 7 signal` of functions that you may safely call in a signal handler
            - Functions NOT included: printf, malloc
            - You can read, write, and set global variables (and many other things)
- The plan:
    - The problem is that the signal handler is definitely executing when the child exits. But that signal handler is changing the process' memory in a non-atomic manner. The kernel, however, guarantees the atomicity of certain function calls, such as write and read
    - Create a pipe for the process (used only for the process to talk to itself). Instead of blocking on wait or usleep, we block on read. The signal handler writes 1 byte to the pipe. The handler for the SIGCHLD will write a byte, as will the SIGALRM. Whichever writes first will unblock read, which will read that byte
    - This is implemented in waitblocksig.c and has the desired behavior, free of race conditions