Due: Monday, July 20th at 11:59pm in p3 of cs50 using handin.
There is no written homework, just programs. Names: strtok.csp, stacker.csp, ch8-11.csp

1) (10 points) Name: strtok.csp
Write a program that will call a function similar to C's strtok() function. The function will have the following Pascal declaration:

FUNCTION STRTOK(VAR STR : CHAR_ARRAY; LETTER : CHAR): INTEGER;

When this function is called with STR != 0 it will find the first instance of LETTER in STR and replace it with a $00 and return 0. Subsequent calls with STR == $00 will cause the function to return the index of the successor (next character) to the last token found, and find the next instance of LETTER from that point and replace it with a $00. LETTER need not be the same with each call. As in C, the end of the original string must be indicated by the character $00. If STRTOK is called when there are no characters left in the string it should return -1. Your program should prompt repeatedly prompt the user for a string and a letter, and then display the substring delimited by the letter. If the user enters nothing as the string then your function STRTOK() should be passed a $000 with the entered letter. The program should end when the user enters 'X' for the string. Here is a sample run of the program:

String: Hello there class!
Letter: h
Hello t
String: ere cla
Letter: o
s!
String:
Letter: s
String: Howdy
Letter: w
Ho
String: Whoopie!
Letter: i
Whoop
String: X
Done

2) (10 points) Name: ch8-11.csp
Do Problem 8-11 from the book.

3) (50 points) Name: stacker.csp
Write a program that emulates the game tetris. You will find my basic C version, tetris.out, centipede.out, centipede.c, centipede.csp, pong.c, pong.csp in ~ssdavis/50/p3.

Specifications:
1. Design
   1.1. Other than the initial set-up the ISRs, and the introduction screen explaining which options your program implemented, all code must be event driven by either the keyboard, or timer interrupts. You should have a main loop of a NOP.
   1.2. Your ISRs must make calls to separate functions to drop the block, move the block, display the score, and produce the game over screen.
   1.3. Though the initial ISRs by necessity must use global variables, all functions they call must be passed all needed (including global) variables as parameters.
   1.4. You must decide whether you wish to think of the CRT buffer as two dimensional, or one dimensional. There are advantages and disadvantages to both.
2. Basic Operation (35 points)
2.1. Program must halt when 'x' is pressed.
2.2. The exposed part of the block must never move outside the 6 x 14 playing area.
2.3. When the user presses the space bar and current block is above the bottom row, the current block must be edited so that only those bricks immediately above bricks on the row below will continue to exist, and a copy of the remaining block appears on the row above.
2.4. Every third drop of a block will cause the delay of the movement of the block to decrease to 75% of its current delay.
2.5. Whenever a block is dropped the score must be updated.
2.6. When the block has no bricks left, an intermission screen must be displayed for approximately three seconds before a new phase begins. The score should accumulate over the phases.

3. Optional Operations (5 points each, unless otherwise specified) You must select at least 15 points of these items to receive full credit. You may do up to five for ten points extra credit.
3.1. Add a gun?? (15 points)
3.2. Allow the user to change the number of bricks with which to start.
3.3. Moving block blinks.
3.4. Lost cube drops to bottom.
3.5. Next screen has fewer cubes to start with.
3.6. Increment score value of each level every three levels.
3.7. Gaps
3.8. Erratic speed.
3.9. Bottom block moves at a different rate.
3.10. Random bad block should not be put on tower.
3.11. Sudden death with timer displaying countdown. (10 points)
3.12. Grow down instead of up after reaching the top. (10 points)
3.13. Different characters in new moving block that must align with those in tower.
3.15. Two player. (15 points)
3.16. Two layers of moving bricks moving at different speeds. (10 points)
3.17. Keep track of high score for all runs of the program in a file named scores.txt. (You will need to access this with the tape system.)
3.18. Pressing 'p' pauses the game until the 'p' is pressed again.

4. Suggestions for development
4.1. Program when you are fresh, and undistracted! Finding bugs is incredibly time consuming.
4.2. Assign all constants names using .equ's.
4.3. Print out stacker.c
4.4. Keep a current version of stacker.lst open on the desktop to easily find addresses for breakpoints. Placing strategically placed breakpoints is key, and you’ll need the stacker.lst to find the right addresses.
4.5. Place your global variables starting at address $500. This assures that they won’t be overrun by your assembled code, and means that their addresses will remain the same throughout the development process which makes debugging much easier.
4.6. Write stubs for all the functions and ISRs with the appropriate BGN, and FIN. I’d also put psha, pshx, popx, and popa in every function and ISR. You can always remove them later, if you do not modify a register. Some functions in stacker.c have no corresponding functions in stacker.csp, e.g. init(). The two ISRs contain fragments of main(), moveBlock(), and getKey().
4.6.1. Always write BGN and FIN in pairs. Always write pushes and their pops or ADS's in pairs.
4.6.2. Changing the number of local variables after writing code can introduce many bugs, so allow for some slack by having BGN # a couple larger than needed initially. The inefficient use of the stack will have no effect on the program.
4.6.3. I will copy the prototypes from stacker.c above each jsr, and then make an .equ for each parameter by prepending part of the jsr’s name to the variable name from the C prototype. You might want to copy entire functions from the C code, and comment it out in CUSP. As you write above the C code, bring along side the CUSP code the C code that it addresses, e.g. have “; for(i = 0; i < n; i++)” next to the test of i in CUSP.
4.6.4. When I have completed and tested a function, or part of it, I draw line through it in my printout of stacker.c
4.7. After writing a function, assemble, and test run it to make sure it works properly before proceeding!!!!
4.8. My version of stacker.c has the following functions in alphabetical order. I suggest that you keep your CUSP functions in alphabetical order so it is easier to search for them.

4.8.1. void dropBlock(Block *block, int *score, int *delay);  // called from keyboard ISR
4.8.2. char getKey();  // replaced by keyboard ISR, so not needed.
4.8.3. void init();  // not needed in CUSP
4.8.4. void intermission();
4.8.5. void msleep(unsigned long milisec);  // not needed in CUSP
4.8.6. void moveBlock(Block *block);  // called from Timer ISR
4.8.7. void printBlock(const Block *block, char letter);
4.8.8. void putChar(char c, int x, int y);  // not needed in CUSP
4.8.9. void showIntro();
4.8.10. void showScore(int score);
4.8.11. void startNewScreen(int score, Block *block, int *delay);

4.9. msleep(), init(), and putChar() are used with ncurses, and have no corresponding functions in the CUSP program.

5. Suggested order for development based on stacker.c.

5.1. Use .blkw to create a Block struct similar to the Tetro struct in tetris.csp.
5.2. Create new .equis for all consts at the top of stacker.c that are not already in tetris.csp.
5.3. The CUSP “main” should call showIntro(), and startNewScreen() and set-up the ISRs (but don’t enable them yet!). There no need to ask for the speed from the user. You should set-up your Timer ISR delay to move the block about twice a second when the Delay scroll bar is in the middle position.
5.4. showIntro() displays your options. This gives you a chance to get use to interact with the keyboard without using an ISR. You will need to access the keyboard data register to await an Enter key press in a loop. This is quite similar to the showIntro() in tetris.csp
5.5. startNewScreen(). Besides setting up the new screen, this set the delay to PLAY_DELAY, and update TIM_VALUE. You should ignore the crt[] array throughout the C code. It was needed to allow access to the values in the display grid. Your CUSP code has direct access to these values using the CRT ports. There is no need to print out the long “-----…-“ string. That was in the C code to demark the CUSP display vertical limits. However, you will need to print the vertical lines. This is similar to startNewScreen() in tetris.csp
5.6. showScore() This is similar to showScore() in tetris except that the score has only two digits.
5.7. printBlock() The only way this will work properly is if you use the memory mapped approach to writing to the display using the CRT_BUFFER, and not CRT_DATA! This will require you to compute a location in a two dimensional array, and transferring the value to XR so you can use index addressing mode with the CRT_BUFFER.
5.8. keyISR() has the part of main() dealing with the space key being pressed. Once you have it written you can enable the Keyboard interrupt. The Quit should work, but the space bar should just cause showScore(), and the dropBlock() stub to be called, and thus do nothing visible.
5.9. dropBlock() is the longest function, so you really must provide good commenting from the corresponding C code, or you will never be able to debug it. I put the C else lines in a comment as the only thing on their corresponding labelled line that is used for their conditional jump, e.g.

```
dropBlock2: ; else // i greater than LEFT_BORDER
```

To emulate reading from the crt array, you can treat the CRT_BUFFER as a two dimensional array and use index addressing to read from it. Beware that the indexing system of the CRT starts at 1, not zero! Note that changing the delay at this point will no effect since the TimerISR has not been enabled yet. When done, each time you press the space bar an X should appear on the left edge on the row above the last one.

5.10. intermission() will change the timer value so the screen will appear for three seconds, temporarily disable interrupts, and simply poll the timer status until the ready bit is set.
5.11. TimerISR() copies delay into TIM_VALUE, and calls moveBlock(). The tetris.csp TimerISR is much more complex, and will mislead you.
5.12. moveBlock() is a bit long, but straightforward.
EQU KBD_CNTL, $000
EQU KBD_STAT, $000
EQU KBD_DATA, $001
EQU KBD_FLUSH, $40
EQU TIM_CNTL, $030
EQU TIM_STAT, $030
EQU TIM_VALUE, $031
EQU TIM_COUNT, $034
EQU INTERRUPT_ENA, $80
EQU RESET_READY_BIT, $40
EQU START_AFTER_LOAD, $10
EQU ENABLE_RESET_AND_START, $D0
EQU ENABLE_AND_RESET, $C0

EQU CRT_BUFFER, $100
EQU BOTTOM_RIGHT, $313
EQU CRT_XREG, $314
EQU CRT_YREG, $315
EQU CRT_CNTL, $316
EQU CRT_DATA, $317
EQU CRT_ROWS, 14
EQU CRT_COLS, 38
EQU CLEAR_DISPLAY, $01

EQU PUT_NUM, $E00 ; MINI_OS JSRS
EQU GET_NUM, $E01
EQU GET_STR, $E04
EQU PUT_STR, $E05
EQU PUT_NL, $E06
EQU PUT_NUM2, $E07
EQU PUT_CHR, $E08

.equ BOTTOM, 14
.equ START_LENGTH, 3
.equ LEFT_BORDER, 15
.equ RIGHT_BORDER, 22
.equ BRICK, 'X'
equ TO_RIGHT, 0
.equ TO_LEFT, 1
equ DROP, ''
equ SPACE, ''
equ QUIT, 'x'
equ START, 13 ; Enter key
equ LENGTH, 0
equ ROW, 1
equ COL, 2
equ DIRECTION, 3