Announcements

- Program #3 due today 8pm
- Exam #2
  - one week from today
  - April 12
  - same room as the 1st exam (ARM 0131)
  - time as the 1st exam (6:00-7:15)
  - minimally comprehensive
- Reading
  - for today's lecture
    - The remaining portions of Chapter 13
  - for Thursday's lecture
    - notes

Command Line Arguments

- int main(void){

- int main(int argc, char *argv[]){
  - argc - number of arguments (including program name)
  - argv - array of command line arguments
    - argv[0] - command invoked to start program
    - argv[n] - n^{th} command line argument
- Example:

  % args -file myfile -help
  - argv[0] = “args"
  - argv[1] = "-file"
  - argv[2] = "myfile"
  - argv[3] = "-help"

Programming Command Line Arguments

- Generally, want command line portions to be accepted in any order but options may have argument which needs to immediately follow
  - args -file myfile -help
  - args -help -file myfile
- One Solution:
  - Table driven command line parsing
    - Example of a common pattern, table drive parsing
  - Declare a structure
    - fields describe each valid option

```
typedef enum { Flag, StrParam, IntParam } optionTypes;
typedef struct {
     char *name;
     optionTypes options;
     void *parameter;
} option;
```

Command Line Arguments -- Table Example

```
int debugFlag, limit;
char *inputFileName;
option   optionTable[] = {
        { "-debug", Flag, &debugFlag },
        { "-file", StrParam, &inputFileName },
        { "-limit", IntParam, &limit },
          /* possible other options go here */
 };

int optionCount = sizeof(optionTable)/sizeof(option);
```

Project 1B Using Table Driven Parsing

```
typedef struct {
    char *name;
    int opCode;
    int numRegisters;
    int usesMem;
} opInfo;
```

```
static opInfo opTable[] = {
    { "Load",   1, 1, 1 },
    { "Move",   2, 2, 0 },
    { "Store",  3, 1, 1 },
    { "Add",    4, 3, 0 },
    { "Halt",   5, 0, 0 },
    { "Negate", 6, 1, 0 },
    { "Branch", 7, 2, 1 },
    { "Bnn",    8, 1, 1 },
    { "Input",  10, 1, 0 },
    { "Output", 11, 1, 0 },
    { "Data",   -1, 0, 1 },
};
```

Environment Variables

- Examples:
  - PATH - used by shell to find programs
  - PRINTER - default printer to use
- Passed as the third parameter to main
  - int main(int argc, char *argv[], char *envp[])
  - consists of a null terminated array of characters
    - KEYWORD=VALUE stored in each one
  - where KEYWORK is the variable name
- C helper functions:
  - char *getenv(const char *keyword);
  - int putenv(const char *string);
    - where keyword is the variable name or string is KEYWORD=VALUE
- Shell Commands to Setup Environement:
  - C-shell and T-shell: setenv FOO_CONFIG ~/foo
  - Bash and Bourne shell: FOO_CONFIG=~/foo

Creating New Processes

- Use fork system call
  - UNIX (and LINUX) specific
  - creates a new copy of the current process
- Function Prototype **`int fork();`**
  - returns twice
    - once from initial call
    - second time in a new process
  - return value indicates which process is which
    - 0 the "child" (new) process
    - > 0 the "parent" (original) process
    - < 0 the "parent", but not child was created (error case)
- ps
  - UNIX command to see the current list of processes

Learning About Other Processes

- Wait and waitpid system call
  - **`#include <sys/types.h>`**
  - **`#include <sys/wait.h>`**
  - **`pid_t wait(int *status);`**
    - wait until a child process terminates
  - **`pid_t waitpid(pid_t pid, int *status, int options);`**
    - wait until the passed process terminates
  - return is the id the the terminated process
  - status fills out a status variable
    - WIFEXITED(status) - true if the child terminated normally
    - WEXITSTATUS(status) - low 8 bits of parameter to exit
    - WTERMSIG(status) - signal number that terminated process

Invoking a new program

- exec system calls
  - int execv(const char *prog, char *const argv[]);
    - run the program in the passed prog parameter
    - pass the new program argv
  - int execvp(const char *file, char *const argv[]);
    - like execv, but used the PATH environment variable
- On success,
  - new program launched, the system call does not return to caller
- On failure,
  - returns -1, sets global **`errno`** to indicate cause of the error

Additional I/O system Calls

- Sometimes processes want to communicate
  - abstraction: pipe - one process writes, other reads what was written
  - shell example: **`ls | wc -l`**
- int pipe(int filedes[2]);
  - create a pipe between two processes
  - write and read system call can use these
- Standard I/O
  - filedescriptor 0 is standard input to a process
  - filedescriptor 1 is standard output from a process
  - filedescriptor 2 is standard error output from
- int dup2(int oldfd, int newfd);
  - change output of one filedescriptor to another
  - dup2(fd, myfd)
    - now all standard input comes from myfd

A Simple Shell Program

```
while (1)  {
      ret = fgets(line, sizeof(line), stdin);
      …
      /* convert line into array of arguments */
      pid = fork();
      if (pid == 0) {
            ret = execvp(args[0], args);
          if (ret) {
              printf("Command not found\n");
              exit(-1);
          }
      } else if (pid > 0) {
           ret = waitpid(pid, &status, NULL);
      } else { … }
}
```