Lab 1: XV6 and UNIX Utilities - avarg116

SLEEP

The UNIX program sleep delays the execution of a process for a number of ticks. To implement sleep, I referenced kill, echo, etc programs that have similar command-line argument parsing.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char *argv[]) {
    if(argc < 2) {
        fprintf(2, "usage: sleep...\n");
        exit();
    }
    int interrupts= atoi(argv[1]);
    sleep(interrupts);

    exit();
}
```

First, there is an if statement that checks if there is at least 1 argument otherwise printing an error message. The “atoi” function converts the string argument to an int (num of ticks). Next, we call the sleep system call with the given number of interrupts and the program pauses.

In xv6, the sleep system call is implemented in the kernel’s sys_sleep function in sysproc.c. The user program calls the sleep function (user.h) prompting the system call. User switches to the kernel using a trap instruction in usys.S.

FIND

The UNIX program find searches recursively through directory trees for files and directories with specific names. The program ls lists the files in a working directory so I modified it to serve the functionality of find.

The fmtname function is unchanged. It formats a string by taking the end of the input path (file/directory name) and checks for fixed length.

```
void
find(char *path, char *filename) //filename added to argument
{
  char buf[512], *p;
  int fd;
  struct dirent de;
  struct stat st;

  if((fd = open(path, 0)) < 0){ //file descriptor
    fprintf(2, "find: cannot open %s\n", path);
    return;
  }

  if(fstat(fd, &st) < 0){ //entry type
    fprintf(2, "find: cannot stat %s\n", path);
    close(fd);
    return;
  }
```

find function: I added a second argument to the ls function. We need both the path to start searching and the filename to search for. The beginning part of the function checks that the file/directory can be opened and retrieves the status (stored in st struct). Otherwise, it prints an error message.

```
switch(st.type){
case T_FILE:
  if(strcmp(path, filename) == 0) { //if filename found
    printf("%s\n", path);
  }
  //printf("%s %d %d %l\n", fmtname(path), st.type, st.ino, st.size);
  break;

case T_DIR:
  if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
    printf("find: path too long\n");
    break;
  }
  strcpy(buf, path);
  p = buf+strlen(buf);
  *p++ = '/';
  while(read(fd, &de, sizeof(de)) == sizeof(de)){
    if(de.inum == 0) //deleted check
      continue;
    memmove(p, de.name, DIRSIZ); //copies name
    p[DIRSIZ] = 0;
    if(stat(buf, &st) < 0){
      printf("find: cannot stat %s\n", buf);
      continue;
    }
    if(st.type == T_DIR && strcmp(de.name, ".") != 0 && strcmp(de.name, "..") != 0) { //if directory
      find(buf, filename);
    }
    else if(st.type == T_FILE && strcmp(de.name, filename) == 0) { //if else file
      printf("%s\n", buf);
    }
    //printf("%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
  }
  break;
}
close(fd);
}
```

The T_DIR case (also in ls) handles a single directory (path). The first condition checks if the path is too long for the buf array (overflow). Next, the path is copied into buf and p points to the end of buf. The while loop checks for a valid entry (inum), copies the name to buf, and goes to the next entry.

T_FILE case: I added an if statement that checks if the path and filename are equal strings using the C library function “strcmp”.

T_DIR case: the if statement first checks if the directory entry is a directory and makes sure that it doesn’t recurse into the current (“.”) or parent (“..”) directories. This allows us to look into subdirectories. The else if checks if the entry is the matching filename.

Output:

```
$ echo > b
$ mkdir a
$ echo > a/b
$ find . b
./b
./a/b
$
```

XARGS

The UNIX program xargs converts input items to arguments and executes a command for each one. It is useful for performing repetitive tasks on large sets of data (e.g. renaming files). Of course, this is a very simplistic version. The real xargs in UNIX has more features like setting the max num of arguments per command, different input types, etc.

```
void
xargs(char *argv[])
{
  char buf[512], *args[MAXARG]; //input line and arguments
  int fd= 0; //standard input
  int num_char;

  while((num_char= read(fd, buf, sizeof(buf))) > 0) {
    for(int i= 0; i < num_char; i++) {
      if(buf[i] == '\n') {
        buf[i]= '\0'; //terminates string

        for(i= 0; i < 2; i++) {
          args[i]= argv[i+1]; //copy args after xargs
        }
        args[i]= buf; //add buf
        args[i+1]= 0;

        if(fork() == 0) { //child process
          exec(args[0], args);
          exit();
        }
        else {
          wait();
        }
      }
    }
  }
}
```

Setup: First, we initialize buf to store the input line and args to store the command and arguments. MAXARG is used to limit the number of arguments to help the kernel and resources. num_char is used to keep track of the characters read from the standard input.

Process: the while loop reads characters from input to buf. The for loop checks if a newline appears and replaces it with null to terminate. The next for loop copies the first 2 arguments from argv in args. Then, it adds buf as an argument to args.

Exec: forks a child process where the exec system call is used to run the command with the arguments. The child exits and the parent waits for it to finish. We use fork so the parent can continue processing input while the child transitions from xargs to execution.

Drawing

ex: echo hello too | xargs echo bye
[xargs, echo, bye]
read:
buf = [hello, too]
newline:
buf = [hello, too \0]
copying arguments:
args = [echo, bye]
add buf as arg:
args = [echo, bye, hello, too \0]
fork:
exec(echo, [echo, bye, hello, too \0])
bye hello too

Testing

```
$ sh < xargstest.sh
$ $ $ $ $ $ hello
hello
hello
$ $ QEMU: Terminated
[avarg116@sledge cs179f-fall23]$ echo hello too | xargs echo bye
bye hello too
[avarg116@sledge cs179f-fall23]$ echo -e "1\\n2" | xargs -n 1 echo line
line 1
line 2
[avarg116@sledge cs179f-fall23]$
```

```
sleep, no arguments:
$ make qemu-gdb
OK (5.1s)
sleep, returns:
$ make qemu-gdb
OK (0.9s)
sleep, makes syscall:
$ make qemu-gdb
OK (0.8s)
find, in current directory:
$ make qemu-gdb
OK (1.1s)
find, recursive:
$ make qemu-gdb
OK (1.4s)
xargs:
$ make qemu-gdb
OK (1.7s)
Score: 100/100
[avarg116@sledge cs179f-fall23]$
```