Nicholas Hemstreet
823 008 051
CSCE 313

Homework 4

1) Suppose a system uses 1KB blocks and 16-bit addresses. What is the largest possible file size for this file system for the following inode organizations?

a) The inode contains 12 pointers directly to data blocks.

Just points to the 12 1KB data blocks therefore the dtasize will be 12 KB.

b) The inode contains 12 pointers directly to data blocks and one indirect block. Assume the indirect data block uses all 1024 bytes to store pointer to data blocks.

The 12 pointers directly refer to the 1 KB blocks, but the indirect block will have 1024 bytes/ (2 bytes) where 2 bytes come from the 16-bit address to yield 512 references so 512 KB + 12 KB = 524 KB

c) The inode contains 12 direct pointers to data blocks, one indirect data block, and one doubly indirect data block.
Assuming that the indirect data block is the same as the first problem then it represents 512 KB of data and the 12 direct pointers reference 12 KB of data and the doubly indirect pointers reference 512 pointers to 512 datablocks of 1 KB so total referenced data = 12 + 512 + 512^2 = 262,144+ 524 = 262,668 KB of data referenced by this schema.

2) Problem 2

```
// Title:: main.c

#include <unistd.h>
#include <stdlib.h>

int main() {

  FILE* in_file = fopen("input.txt", "w");
  FILE* out_file = fopen("output.txt","w");

  pid_t pid = fork();
  if (pid == 0){
    // The output of the picodb function will be the input to this program
    dup2(fileno(in_file),1);
    // the input file to the picodb function is our output file from this program
    dup2(fileno(out_file),0);
    // Now that we have setup the files we execute the other program
    execl("./picodb","");
  }else if (pid > 0){
```

```
    // Parent process
    // To send data to the other function we just write to the in_file
    // To get data from the other function we read from the out_file
  }
  return 0;
}
```

3) Problem 3

Assuming that these files are writing to the device using some sort of file descriptor or piping (which can be turned into a file descriptor by the command line and thus any execl command) you can simply repalce the global variable with the name of the file which you are keeping all the other pieces of data. I would suggest, however, that you implement a data "staging" area that will look through the incoming data before depositing in the main stage to keep any negative or possibly harmful data from corrupting the log file. In thise case I would setup a monitor which wakes up intermittently, checks the staging area and then pipes it over to the final log file where we can keep all of the main data that has been approved.

4) Please see the problem_4.c file as well as the makefile for a proper show of the program. I've pasted the program below but it may be easier to read as a seperate text file
* I've included a compiled version which SHOULD work on your architecture. A lot of the gnu specific stuff is really distro dependent which is why I compiled the file with gnu99. Keep that in mind when you start compiling. If there are any problems please feel free to email me and I"ll make sure things work fine. Thank you.

problem_4.c

```
// Title :: Problem 4 Implementation
// Author:: Nicholas Hemstreet


#include <dirent.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

// Global variables
int fd[2], nbytes;

int search(char* s_dir,int level);
```

```
// fd[0] == read from this channel
// fd[1] == write to this channel


int main(int argc, char** argv){
  // Setup the file descriptors and output to
  // We exit if we don't get a directory name
  char* s_dir;
  if (argc < 2){
    s_dir = (char*)malloc(sizeof(char[2]));
    s_dir[0]= '.';
    s_dir[1]= '/';
  }else{
    s_dir = argv[1];
  }

  // Pipe File
  pipe(fd);
  pid_t pid = 0;
  pid = fork();
  if (pid == 0){
    close(fd[0]);
    // Child process, Search the past directory
    search(s_dir,1);
    // Reached the end of the file time to close up shop
    close(fd[0]);
    close(fd[1]);
    // we need to
    printf("Done Searching\n");
    exit(0);
  }else if (pid > 0){
    // Parent Process
    close(fd[1]);
    int total_count = 0;
    int bytes_to_read = 0;
    // Wait for the child process to complete it's stuff
    wait(NULL);
    ioctl(fd[0],FIONREAD,&bytes_to_read);
    char readbuffer[bytes_to_read];
    memset((void*)readbuffer,0,sizeof(char[bytes_to_read]));

    nbytes = read(fd[0], readbuffer,sizeof(readbuffer));
    // need to count the number of \ns in the readbuffer
    for (int i=0; i < nbytes; i++){
      if (readbuffer[i] == '\n'){
        total_count++;
      }
    }
```

```
    if (nbytes != bytes_to_read){
      printf("Tom Follery is afoot\n");
    }
    printf("Total Files and Directories = %d\n",total_count);
    close(fd[0]);
    close(fd[1]);
  }else{
    // Do nothing because it failed
  }

  return 0;
}

int search(char* s_dir, int level){
  DIR*    search_d;
  struct dirent *dir;
  search_d = opendir(s_dir);
  if (search_d==NULL){
    return 1;
  }
  char wr_val[strlen(s_dir)+2];
  memcpy ((void*)wr_val,s_dir,strlen(s_dir));
  wr_val[strlen(s_dir)] = '\n';
  wr_val[strlen(s_dir)+1] = '\x00';
  write(fd[1],s_dir,strlen(s_dir));
  printf("%s",s_dir);
  printf(" (directory)\n");
  while ( ( dir = readdir(search_d)) ) {
    // Loop through each file inside of this file
    if (strcmp(dir->d_name,"..") != 0 && strcmp(dir->d_name,".") != 0){
      if (dir->d_type == DT_DIR ){
        // if this is a directory we need to spawn a new thread and
        // create a new process
        // We need to create the new path to this directory
        char new_address[strlen(s_dir) + strlen(dir->d_name) + 2];
        memcpy((void*)new_address,(void*)s_dir,strlen(s_dir));
        // Now we need to build the new pointer for new_address
        char* d_addr = (char*)(&new_address[0] + strlen(s_dir));
        memcpy((void*)d_addr, (void*) dir->d_name, strlen(dir->d_name));
        new_address[strlen(s_dir) + strlen(dir->d_name)] = '/';
        new_address[strlen(s_dir) + strlen(dir->d_name)+1] = '\x00';
        // Make sure that there is a backslash to search recursively
        search((char*)new_address,level+1);
        // We should also count the directory
      }else {
        char output[strlen(dir->d_name)+1];
        memcpy((void*)output,(void*)dir->d_name,strlen(dir->d_name));
        output[strlen(dir->d_name)] = '\n';
        // Otherwise it's a boring ass file so we pipe it out to the filecounter
```

```
        write(fd[1],output,(strlen(output)));
        printf("|");
        for (int i=0; i < level; i++){
          printf("-");
        }
        printf(">");
        // Read out the file
        printf("%s \n",dir->d_name);
      }
    }
  }
  return 0;
}
```

makefile

```
all: problem_4.c
	gcc -std=gnu99 -g -Wall problem_4.c -o problem_4
```