Pintos System Calls

File System

Recap

- System calls you have implemented so far
  - exit
  - exec
  - wait
  - write (only for stdout)

Assignment 6

- Implement the system calls to
  - create, remove, open, close files
  - sequential read and write
  - get and change file position (seek, tell)
  - get size of a file
  - halt system call

Assignment 6

- File system is already implemented
- See `filesys/filesys.h` and `filesys/file.h`
- You will find all the functions you need for working with files (create, remove, read, write ...)

File descriptors (fds)

- A file descriptor (fd) is an integer value mapped to a file handle in the kernel
- Open system call returns an fd

```
int open (const char* filename)
```

- System calls that use files take the fd as argument

```
int write (int fd, void* buffer, int size)
```

Example

- Example program that writes to a file

```
int main (int argc, char const *argv[])
{
  char text [] = {"hello!"};
  create ("myfile.txt", strlen (text));
  int fd = open ("myfile.txt");
  write(fd, text, strlen (text));

  return 0;
}
```

Handling fds

- In the kernel, files are represented by a `struct file`
- In `file.h`, function `filesys_open` returns a pointer to a `struct file`, not an integer (fd)

```
struct file* filesys_open (const char* name)
```

- How can we map files to FDs?

Handling fds

- Keep a hash table that maps ints (fds) to file pointers
  - Either a table per thread or just a big global one
- Whenever a file is opened
  - create a unique integer for the FD
  - keep track of which fd maps to which file pointer using the hashtable

Pintos hash tables

- See `lib/kernel/hash.h`
- Usage is similar to lists

Hash table example

```
struct item {
    int item_id;
    // other fields here
    struct hash_elem elem;
};

// the hash table requires a function that returns a hash code for any item
unsigned item_hash (const struct hash_elem * e, void * aux) {
    struct item * i = hash_entry (e, struct item, elem);
    return hash_int (i->item_id);
}

// the hash table also needs a function to compare items
bool item_compare (cons struct hash_elem * a, cons struct hash_elem * b, void * aux) {
    struct item *i_a = hash_entry (a, struct item, elem);
    struct item *i_b = hash_entry (b, struct item, elem);
    return i_a->item_id < i_b->item_id;
}

// initializing the hash table
struct hash items_table;
hash_init (&items_table, item_hash, item_compare, NULL);
```

Hash table example

```
// insert
struct item * i = malloc(sizeof(struct item));
i->item_id = 1;
hash_insert(&items_table, &i->elem)

// lookup: to search in the hash table for an item with
// a certain id, you have to create a struct with such id
struct item i;
i.item_id = wanted_id;
struct hash_elem * e = hash_find(&items_table, &i.elem);
```

Remarks

- Synchronization
  - only one thread at a time can manipulate the filesystem (use a semaphore or a lock)
- Halt system call
  - just call `shutdown_power_off()`

Tests

All tests should pass, except for:

- sc-bad-sp
- sc-bad-arg
- rox-simple
- rox-child
- rox-multichild
- multi-oom
- wait-killed

Readings

- Section 3.3.4 covers the system calls you have to implement
- Appendix A.8 covers hash tables