Project 2 Overview

Task 1 : Implement file system calls

- *Create, open, read, write, close* and *unlink*
- See *test/syscall.h* and *UserProcess.handleSyscall*
- See the code for *halt* and skeleton code for *exit* in *UserProcess.java*

Task 1 : Implement file system calls

- File system calls are implemented in the *UserProcess.handleSyscall(...)* method which takes in an *int* as the syscall number and *int* syscall arguments depending on the type of syscall invoked.
- A stub file system interface is already provided as well as methods to read/write virtual memory (within *UserProcess.java*). So these should be used by the file system calls.
- The *OpenFile* class should be treated as an interface which syscalls use to interact with files.
- All files accessed by processes are stored as *OpenFile* objects in an array.

Task 1 : Implement file system calls

The general requirements for implementation are:

- Each program should be able to handle at least 16 concurrently open files
- Each program assigns a unique nonnegative integer to every open file to serve as a file descriptor for indexing purposes (reuse of indexes once a file is closed is allowed)

  Should be stored as a data structure of OpenFile objects
- Returns from system calls are int values which depend on syscall success (various) or failure (-1)

Task 1 : Implement file system calls

- Create vs open: The purpose of open is to find a named file and open it into that process (returning its file descriptor). If the file does not exist, open will fail. Create will work like open, except if the file does not exist, create will create the named file and open it into that process (returning its file descriptor).

- Write vs. read: write will take a virtual address location in the process and read its contents into a byte array. Write will then write those contents into an open file in the process. Read works similarly, however it will read the contents of the file in the process, and then write those contents to the virtual address provided.

Task 1 : Implement file system calls

- Close vs. Unlink: close only closes the active file in the process. Unlink deletes the file from the file system (regardless of whether it is being used in the file system).

Task 2: Multiprogramming

- Make the nachos code work for multiple user processes.
- Modify
  - *UserProcess.loadSections()*
  - *UserProcess.readVirtualMemory ()*
  - *UserProcess.writeVirtualMemory ()*

Task 2: Multiprogramming

- Make the nachos code work for multiple user processes.
- Modify
  - *UserProcess.loadSections()*
  - *UserProcess.readVirtualMemory*
  - *UserProcess.writeVirtualMemory*

Task 2: Multiprogramming

- All free memory in the system can be accessed using calls to the machine processor and the number of physical pages is also a known entity.

- The free pages in the system are accounted for in linked list within the *UserKernel* class. When a new process is created in the user kernel, allocate a sufficient number of pages of memory for it.

Task 2: Multiprogramming

LoadSections()

- Allocate the number of pages that the process needs depending on the size.
- A pageTable was added so that the process is loaded into the correct physical memory pages.
- Check if there is enough pages available for the process to use. If not, an error should be returned.

Task 2: Multiprogramming

ReadVirtualMemory

- The readVirtualMemory method will transfer data from the byte[] array to the process's virtual memory. From there it will return the number of bytes that were successfully copied if there were any bytes copied.

WriteVirtualMemory

- The writeVirtualMemory method take in the same parameters as readVirtualMemory and have the same structure.
- Check if the page is read only before any bytes start to get copied. If the page is read only then no bytes will be copied.
- The chunk of memory that is copied into the array is different.

Task 3: Implement the system calls

- *Exec*, *join*, and *exit* (documented in *syscall.h*)

  Exec:

  - Spawn a new process
  - Give each new process a unique process ID and remember the parent process (which spawned the current process)
  - The parent process should somehow know all its child processes

  Join:

  - The functionality is the same as the *KThread.join,* but here we join a UserProcess
  - Only parent processes are allowed to join to their chirdren.

Task 3: Implement the system calls

- *Exec*, *join*, and *exit* (documented in *syscall.h*)

  Exit:

  - Terminate processes and free up resources they previously possessed
  - Close all the open files before terminating
  - The last process to call exit should also directly cause the machine to halt by calling Kernel.kernel.terminate().

Suggestions

- Read the code that is handed out with the assignment first, until you pretty much understand it. Start with the interface documentation.

- Don't code until you understand what you are doing. Design, design, and design first. Only then split up what each of you will code.

- If you get stuck, talk to your teammates, TA and instructor.

Suggestions

- Communication and cooperation will be essential
  - Regular meetings
  - Slack/Messenger/whatever doesn’t replace face-to-face!
- Everyone should do work and have clear responsibilities
  - You will evaluate your teammates at the end of each project.
  - Dividing up by Task is not a good approach. Work as a team.