Betriebssysteme Assignment 2

Design Document 2

Group 13

Werner Schuster, Andreas Rath, Kerstin Pötsch, Wolfgang Pototschnik

Version 1.0, 22.11.2002

Contents

Chapter 1

Overview

1.1 Overview

The assignment consists of 2 major sets of changes concerning:

- implementation of the NACHOS syscalls (see section 3.1 for the implementation details), (Task 1 and 3)
- adding multiprogramming with time slices (see section 3.2) (Task 2)

With these adaptations NACHOS offers a the programming interface described in chapter 2.

To provide a stable OS, all kinds of potential error conditions must be considered and dealt with (as far as possible). Details about the measures taken can be found in section 4.1. This section describes how dangerous input data must be handled in the OS code and the necessary checks that are required, so the description of these checks won't be repeated in the implementation details (eg. of the methods handling the syscalls,...).

To assure a correct implementation of these changes, a suite of tests is provided. An overview can be seen in 4.2.

Chapter 2

Application Programming Interface

2.1 Libraries

Alway include the `syscall.h` file for necessary definitions of types (SpaceID, OpenFileID) and for the API declarations (functions, constants for errorcodes,...). To Write to and Read from the console, use the STDOUT and STDIN file handles, also defined in `syscall.h`.

2.2 Syscalls

2.2.1 Syscall details

SpaceID Exec(char* appName)

Creates a new child process and launches the program that is referenced in the appName string. This string contains the name of the executable file (Caution: if the Nachos versions uses the filesystem of the host OS, the filename must be relative to the directory where you launched Nachos). The name can be followed by an unlimited number of arguments (seperated by the space character). These arguments will be passed to the launched programs **main()** function as the **argv**[] array. Eg. you can launch a program like this: `Exec("progname arg1 arg2 arg3")`, "progname" being the name of the executable, the rest are strings making up the arguments.
Errorcodes (you can compare the returned SpaceID to these constants)

- EXEC_FILE_NOT_FOUND
  the executable was not found
- EXEC_NOT_ENOUGH_MEMORY
  there was not enough userspace memory to launch the program

Exit(int exitCode)

Exit the current process. The exitCode can be read by the parent process, and will be stored until the parent calls Exits as well.

int Join(SpaceID process)

Blocks the current process until the process with the given SpaceID has exited. This function returns the exitcode of the joined process, or zero if the SpaceID was invalid.

int Create(char* filename)

Tries to create the file with the name filename. If the file can be created, the return value is 1, otherwise the return value contains an error value.
Errorcodes:

- CREATE_OK
  the file was created
- CREATE_INVALID_FILENAME
  the filename given as argument was not a valid name for a file
- CREATE_INVALID_STRING
  the argument string was invalid (eg. null pointer,...)

OpenFileID Open(char* filename)

Tries to open the file with the name filename. If the file can be opened, the OpenFileID (which is simply a typedef'ed int) is returned. Otherwise the return value contains an error value.
Errorcodes (you can compare the returned OpenFileID to these constants):

- OPEN_INVALID_FILENAME
  the filename given as argument was not a valid name for a file
- OPEN_NO_SUCH_FILE
  the filesystem contains no file by this name
- OPEN_INVALID_STRING
  the argument string was invalid (eg. null pointer,...)

int Write(char* buffer, int size, OpenFileID file)

Writes size bytes from from buffer into file (which must be open). This syscall returns the amount of bytes that were written (which is not necessarily the same as size).
Errorcodes:

- OPEN_INVALID_FILE_ID
  the OpenFileID given as argument was not a valid (eg. there exists no such ID for this process)
- OPEN_INVALID_BUFFER
  the buffer was invalid (eg. null pointer,...)

int Read(char* buffer, int size, OpenFileID file)

Reads size bytes from from file (which must be open) and stores them in buffer. This syscall returns the amount of bytes that were read (which is not necessarily the same as size).
Errorcodes:

- OPEN_INVALID_FILE_ID
  the OpenFileID given as argument was not a valid (eg. there exists no such ID for this process)
- OPEN_INVALID_BUFFER
  the buffer was invalid (eg. null pointer,...)

int Close(OpenFileID file)

Closes the file with this OpenFileID. .
Errorcodes:

- OPEN_INVALID_FILE_ID
  the OpenFileID given as argument was not a valid (eg. there exists no such ID for this process)

Yield(void)

Yield the control of the CPU to some other process. Use this call (if the program is not busy) instead of using up all your time slice.

Ps(void)

Shows a list with details about all the processes on the system to the console (stdout of the calling process).

2.2.2 Overview

syscall	**arg1**	**arg2**	**arg3**	**arg4**	**return value**
Exec	char * (zero terminated string)				SpaceID
Exit	int, (exit code)				
Join	PID				int (exit status of joined process)
Create	char * (filename)				
Open	char * (filename)				OpenFileID
Write	char * (data to write)	int (number of bytes to write)	OpenFileID (the file to write to		int (bytes written)
Read	char * (buffer to read into)	int (number of bytes to read)	OpenFileID (the file to read from)		int (bytes read)
Close	OpenFileID (the file to close)				
Yield					
Ps					

Chapter 3

Implementation

3.1 Syscalls

3.1.1 New syscall: PS

For this new syscall, a new constant `SC_PS` (for its number) and function prototype `Ps()` must be added to `syscall.h`.

This code is implemented in the file `exception.cc`.

3.1.2 ExceptionHandler

The code of the ExceptionHandler function is altered to contain a switch statement that decides which syscall handler function to call depending on the the number of the syscall which is stored in register number 2 of the MIPS CPU. The function is then called with the arguments taken from registers numbers 4 to 7 from the MIPS CPU.
All the syscall handler functions return an integer, which the ExceptionHandler function stores in the register number 2 of the MIPS CPU (as defined).
These functions return errorcodes which are defined in `syscall.h`, and can be seen in detail in chapter refAPI, so they won't be repeated here.

3.1.3 Syscall handler functions

int doExec(int, int, int, int)

Reads the arguments and tries to open the executable. If that succeded, try to create a new Thread, and an AddrSpace for the new program. If all that has succeded, assign the AddrSpace to the Thread, and fork the Thread using the ProcessExecute function (also defined in this file), before that, the registers number 4 and 5 must be set to point to the arguments of the userspace main function. The new Thread must be appended to the ProcessTable, to get the new SpaceID, which is this syscalls return value. The ProcessExecute simply inits the process and calls machine->Run() on it, to start executing the programs instructions.

int doExit(int, int, int, int)

Calls Thread->Finish() which cleans up (frees all allocated memory, calls AddrSpace destructor, removes child processes from ProcessTable...) and sets the state of the process (in the ProcessTable) to DEAD.

int doJoin(int, int, int, int)

Gets the Thread object for the SpaceID (using the ProcessTable), and if it is available, calls currentThread->Join(Thread).

int doCreate(int, int, int, int)

Checks the parameters and tries to create the file, using the filesystem.

int doOpen(int, int, int, int)

Checks the parameters and tries to open the file, using the filesystem.

int doWrite(int, int, int, int)

Checks the parameters and tries to write from the file, using the filesystem.

int doRead(int, int, int, int)

Checks the parameters and tries to read from the file, using the filesystem.

int doClose(int, int, int, int)

Checks if the OpenFileID is valid and then tries to close the file using the filesystem.

int doYield(int, int, int, int)

Calls currentThread->Yield()

int doPs(int, int, int, int)

Writes a list of processes using doWrite(...) with the STDOUT OpenFileID. The format consists of "PS" followed by a newline, and then followed by the header of the table: "SpaceID alloc. Bytes alloc. Pages commandLine" followed by a line (one for each process) that shows the appropiate info for each of these columns.
The output string is not accumulated and then written to the STDOUT, but instead each line gets composed and immediately written, thus only a constant amount of memory is needed instead of an amount proportional to the amount of processes on the system.

3.2 Multiprogramming

3.2.1 New subsystems/classes

The following subsystems are initialized in `Initialize` and are stored in global variables. (While this might not be an elegant solution, no better one was found before the deadline for this document).

ProcessTable

An instance of this is stored in the global variable `processTable`. The ProcessTable holds a lists of all the processes on this system. A process can either be **ALIVE** or **DEAD**.
If a process is ALIVE, that means it is running or will run again. For a process that is ALIVE, the ProcessTable returns the corresponding Nachos Thread object, which can be used to access the AddrSpace object (which offers access to all information about the process).
If a process is DEAD, that means it is not running and won't run again. This means, that all the data about this process (Thread, AddrSpace,... objects) have been deleted. The reason why this process (and its SpaceID) is still in the ProcessTable, is that someone (its parent process) might want to access the exit code of the DEAD process by way of the Join syscall.
If a process is finished, the Thread::Finish method must must remove all its child processes from the ProcessTable so that these SpaceIDs are freed.
The ProcessTable internally uses a List to store the information about the processes.

Methods:

- `int getProcessState(SpaceID id)`
  returns the status (ALIVE or DEAD, which are int contants) for the process with the SpaceID id. If there is no such SpaceID, the return value is NO_SUCH_SPACE_ID.

- `SpaceID appendProcess(Thread thr)`
  adds a new process into the ProcessTable. A unique SpaceID is generated and returned.

- `Thread* getThreadBySpaceID(SpaceID id)`
  returns the Thread corresponding to the process with the SpaceID id. If the SpaceID does not exist OR if the process is DEAD, the methode returns NULL.

- `int getExitCodeBySpaceID(SpaceID id)`
  returns the exit code of the process with the SpaceID id. CAUTION: If the process is still ALIVE (which means that there is no exit code), this method returns zero;

- `killProcess(SpaceID id)`
  this causes the process with SpaceID id to finish. After that, the process has

the state DEAD and still remains in the ProcessTable. This is called by the Exit syscall.

- `removeProcess(SpaceID)`
  this removes a DEAD process from the ProcessTable, but has no effect on a process that is still ALIVE. A parent process/thread has to call this on all its child processes if the parent process is killed (eg. by the Exit syscall).

- `int getNumberOfProcesses()`
  returns the number of processes stored in the ProcessTable.

MemoryManager

An instance of this is stored in the global variable `userspaceMemoryManager`. It manages the physical userspace memory, ie. if a new physical memory (pages) must be allocated for a user space process, the MemoryManager must be used. The methods of the MemoryManager must also be threadsafe to avoid race conditions.
Methods:

- `int getFreeMemory()`
  returns number of bytes of non-allocated physical userspace memory

- `int getNumberOfFreePages()`
  returns number of pages of non-allocated physical userspace memory

- `int[] allocateMemory(int num)`
  allocates `num` pages of physical userspace memory and returns an array containing the numbers of the allocated pages.

- `void releasePages(int[] pages)`
  releases the pages with the numbers contained in the pages[] argument.

3.2.2 Changes in existing classes/files

AddrSpace

- TranslationEntry *pageTable

- List fileHandles
  contains the OpenFileID for the process; numbers 0 and 1 are reserved for STDIN, STDOUT.

- int[] pages
  array containing the numbers of physical pages for this virtual address space

- char* commandLine
  the char* buffer that this process was started with

- SpaceID PID

- SpaceID getSpaceID()
  returns the SpaceID for this AddrSpaces process

- Extend the constructor
  The constructor must read the information stored in the executable to gather information about the new process; especially the required amount of memory. If that has been calculated, it tries to allocate the required amount of pages using the MemoryManager. If enough pages are free, the page table can now be filled. The address space can be initialized (code, static data, arguments, heap) and the stack prepared (the pointer to the stack top is .stored in register number 29).

- extend the destructor
  To close all open files, release all allocated Memory using the MemoryManager

Thread

- AddrSpace* space
  pointer to the AddrSpace of this process/thread

- change Finish to clean up
  call the AddrSpace destructor, remove child processes from ProcessTable

3.2.3 Other changes

Activate the Timer

In the fiely userprog/system.cc always start the Timer which on each Timer interrupt calls interrupt->YieldOnReturn(), causing the scheduler to run. The code for this is already in there, but turned of by the `if (randomYield)`. Also set the Timer to cause the event after a fixed amount of time, instead of randomly (as it is now, set with randomYield in the Timer constructor).

Chapter 4

Quality assurance

4.1 Error handling

4.1.1 General policy

Null Pointers

If a syscall takes a pointer as an argument, it has to be checked that the pointer != NULL.

IDs as index into internal table

IDs (SpaceID, OpenFileID,...) which are used as index into internal lits, must be stored as safe data structure (List,...) so that errors (like negative values or a value¿lenght of the List) can not cause unwanted effects.

4.1.2 Error situations

Non-zero-terminated strings

If a string is (by mistake) not zero-terminated, this can cause problems with string functions and can generate hard-to-find bugs.
Despite that, a userspace program that uses such a faulty string as an argument to a syscall will not cause stability problems for Nachos. The string will go on until the first memory adress that contains a zero or until the end of the user spaces adressspace. If this string is to be copied to another processes userspace, but proves to bee too big, the syscall will simply fail. On the other hand, if the string is to be copied to user space (eg. as the name of an executable,...), the OS tries to allocate enough memory for the string, if that fails, the syscall with that string will also fail.

Out of memory

It is possible that a memory allocation fails, because there is not enough memory left.

- User space
  This cannot happen in userspace except for the initial allocation of memory in the process creation (Exec syscall), since that is the only time when memory is allocated in userspace. If the creation of a new process requires more memory than is physically available, the creation (= the Exec syscall) will simply fail.

- Kernel space
  This situation is currently not handled, because the small physical userspace memory does not allow enough data and/or processes to be able to fill up the much bigger amount of data available to the host OS. If this (against all expectations) happens, the only possible solution would be to kill processes to free memory allocated for their data structures (Thread, AddrSpace, ... objects). Since it is difficult to decide which process can safely be killed, the Ostrich-Algorithm (ie. ignore the problem) was found to be a appropiate solution for the time.

Exceptions

The MIPS CPU generates several kinds of exception, of which only the SyscallException is currently handled. The other Exceptions do not pose a stability threat for the OS, and can for now be ignored (the memory related Exceptions will be handled in a future assignment anyway).

4.2 Tests

To make sure the implementation has succeded and works correctly, all the new features are tested.

These tests are all userspace programs, that check the correct implementation of the defined syscall interface, and also check if the stability of the OS by attempting to execute syscalls with invalid arguments.
The multiprogramming functionality is tested by the Exec, Exit,... syscall tests (ie. if those tests work correctly, multiprogramming works).

4.2.1 Filesystem syscalls

Implemented in `test/filesys.c`, this test checks Create, Open, Write, Read, Close. If all these steps have completed correctly, the filesystem syscalls should work. The program tries to

- Create a file
- Open the file
- Write some bytes into the file
- Close the file
- Open the same file again

- Read the contents of the file and check if they are the same that were written before
- Close the file again.

4.2.2 Multiprogramming syscalls

Implemented in `test/exectest.c, test/childNoArgs.c, test/childArgs.c, childJoin.c, yield.c`, and tests checks Exec (with and without arguments), Exit, Join, Yield, Ps.
If all these steps have completed correctly, multiprogramming and the corresponding syscalls should work.

Exec

- tries to launch the childNoArgs test program (test/childNoArgs)
- tries Exec with arguments (test/childArgs)
- executes the test/childJoin and joins it.
- executes test/yield, calls Yield several times. If Yield works, test/yield has time to create a file yield.test. The test then checks if this file exists, if it does, the Yield works. (CAUTION: Yield should be called at least 10 times, to make sure the other process can finish the filesystem operation)
- calls Ps. Since there would be no proper way to check the correctness properly, this test must be observed by a human tester. [The Ps test is still under construction].