Project 2: Multiprogramming System Calls

Overview

The Nachos code you have been given is capable of executing user application programs, but in an extremely limited way. In particular, at most one user application process can run at a time, and the only system call that has been implemented is the `halt` system call that shuts down Nachos. In this assignment, you will correct some of these deficiencies and turn Nachos into a multiprogramming operating system with a working set of basic system calls.
For this assignment, you are to modify Nachos so that it can run multiple user applications at once. That means that you have to implement the `Fork`, `Yield`, `Exit`, `Exec`, `Kill`, and `Join` system calls. The detailed specifications for these system calls are given below.
For this assignment, you will be working on the version of Nachos in the `userprog` directory. You will also need to write some simple user application programs, compile them using the MIPS cross compiler, and run them under Nachos to test that your modifications to Nachos actually work. User application programs are written in ANSI C. To write them, go to the `test` subdirectory of Nachos. Several sample applications are already provided in that directory. The only one that will run properly on an unmodified Nachos is the `halt` program. When you are in the `code` directory and type `make`, the programs in the `test` directory are cross compiled and executable Nachos user applications are created. Check out the `test` directory and note the source files such as `halt.c` with their corresponding executables such as `halt`. Then go to the `userprog` directory. Nachos should have been built already (i.e., there is a `nachos` executable in this directory). If not, type `make nachos`. Then, type `nachos -x ../test/halt`. This will start Nachos and ask it to load and run the `halt` program. You should see a message indicating that Nachos is halting at the request of the user program.
In brief, what happens when you type `nachos -x halt` is as follows:

- Nachos starts. The initial thread starts by running function `StartProcess()` in file `progtest.cc`.
- A new address space is allocated for the user process, and the contents of the executable file `halt` are loaded into that address space. This is accomplished by the constructor function `AddrSpace::AddrSpace()` in the file `addrspace.cc` called from `StartProcess()`.
- MIPS registers and memory management hardware are initialised for the new user process by the functions `AddrSpace::InitRegister()` and `AddrSpace::RestoreState()` in `addrspace.cc`.
- Control is transferred to user mode and the `halt` program begins running. This is accomplished by the function `Machine::Run()` in `machine/mipssim.cc`, which starts the MIPS emulator.
- The system call `Halt()` is executed from user mode (now running the program `halt`). This causes a trap back to the Nachos kernel via function `ExceptionHandler()` in file `exception.cc`.
- The exception handler determines that a `Halt()` system call was requested from user mode, and it halts Nachos by calling the function `Interrupt::Halt()` in `machine/interrupt.cc`.

Trace through the Nachos code until you think you understand how program `halt` is executed.

In this assignment, you will also need to know the object file formats for Nachos. This is how NOFF (Nachos Object File Format) looks like.

```
----------- | bss | segment ----------- | data | segment ----------- | code | segment
----------- | header | -----------
```

Noff-format files consist of four parts. The first part, the Noff header, describes the contents of the rest of the file, giving information about the program's instructions (code segment), initialised variables (data segment) and uninitialised variables (bss segment).

The Noff header resides at the very start of the file and contains pointers to the remaining sections. Specifically, the Noff header contains

```
-------------- |magic | 0xbadfad -------------- For each of the three segments ------
-------- |virtual addr| points to the location in virtual memory -------------- |in f
ile addr| points to a location within the NOFF file where section begins ------------
-- |size | size of the segment in bytes -------------
```

This information about the NOFF can be found in /bin/noff.h file.

When you create user programs and compile them using the MIPS compiler (cross compile), you get COFF (common object file format) files. This is a normal MIPS object (executable). For this file to be runnable under Nachos, it has to be turned into NOFF. This is done by using `bin/coff2noff`, the COFF to NOFF translator. Please check the Makefile in code/test directory to see how is this done. You will need to add code in start.s and userprog/syscall.h in order to add a new system call (Kill).

Exercises

You are to implement the `Fork()`, `Yield()`, `Exit()`, `Exec()`, `Kill`, and `Join()` system calls. The function prototypes of the system calls are listed in `syscall.h` and act as follows:

- The `Fork(func)` system call creates a new user-level (child) process, whose address space starts out as an exact copy of that of the caller (the parent), but immediately the child abandons the program of the parent and starts executing the function supplied by the single argument. Fork should return pid of child process (`SpaceId`). Notice this definition is slightly different from the one in the syscall.h file in Nachos. Also, the semantics is not the same as Unix fork(). After forked function func finishes, the control should go back to the instruction after the initial system call Fork.
- The `Yield()` call is used by a process executing in user mode to temporarily relinquish the CPU to another process. The return value is undefined.
- The `Exit(int)` call takes a single argument, which is an integer status value as in Unix. The currently executing process is terminated. For now, you can just ignore the status value. You need to supply this value to parent process if and when it does a `Join()`.
- The `Exec(filename)` system replaces the current process state with a new process executing program from file . You can think as if the current process stops executing and the new program is loaded in its place. The new program uses the object code from the Nachos object file which

name is supplied as an argument to the call. It should return -1 to the parent if not successful. If successful, the parent process is replaced with the new program running from its beginning. The way to execute a new process without destroying the current one is to first Fork and then Exec.
- The `Kill(SpaceId)` kills a process with supplied SpaceId. It returns 0 if successful and -1 if not (for example SpaceId is not valid).
- The `Join(SpaceId)` call waits and returns only after a process with the specified SpaceID (supplied as an argument to the call) has finished. The return value of the Join call is the exit status of the process for which it was waiting or -1 in the case of error (for example, invalid SpaceId).

Test your code by creating several user programs that exercise the various system calls. Be sure to test each of the system calls, and to try forking up to three processes (since each has a 1024 byte stack, that's all that will fit in Nachos' 4K byte physical memory right now) and have them yield back and forth for a while to make sure everything is working. Since the facility for I/O from user program will be implemented in your next assignment, you may *initially* have to rely on using debugging printout in the kernel to track what is happening. Use the `DEBUG` macro for this, and make sure that debugging printout is disabled by default when you submit your code for grading.

Required Output Prints

In order for us to see how your program works, some debugging information must be added in your code. You should print out the following information:

1. Which system call is called? Whenever a system call is invoked, print:
   `System Call: [pid] invoked [call]`
   where [pid] is the identifier of the process (SpaceID) and [call] is the name of one of the system calls that you had to implement. Just give the name without parentheses (e.g., Fork, Create, Exit).
2. How many pages are allocated to the user program when it is loaded? The following line should be printed when a program is loaded:
   `Loaded Program: [x] code | [y] data | [z] bss`
   where [x], [y] and [z] are the sizes of the code, (initialized) data and bss (uninitialized) data segments in bytes.
3. Which process has been forked and how many pages are allocated to the forked process? Whenever a new process is forked, print the following line:
   `Process [pid] Fork: start at address [addr] with [numPage] pages memory`
   where [pid] is the process identifier (SpaceID) of the parent process, [addr] is the virtual address (in hexadecimal format) of the function that the new (child) process starts to execute and [numPage] is the number of pages that the new process gets.
4. What is the file name in the Exec system call? When a new process should be executed, print the following line:
   `Exec Program: [pid] loading [name]`
   where [pid] is the identifier of the process that executes an Exec system call and [name] is the name of the executable file to be loaded

5. When a process exits, print the following line:
   `Process [pid] exits with [status]`
   where [pid] is the identifier of the exiting process and [status] is the exit code.
6. When a process is killed, print the following line:
   `Process [pid] killed process [killed-pid]`
   where [pid] is the identifier of the process calling Kill system call and [killed-pid] is the pid of killed process. If the call is unsuccessful, print:
   `Process [pid] cannot kill process [killed-pid]: doesn't exist`

Grading

System calls Fork and Exec are the most complex assignments and they are 35% of the total grade each (70% for both). Yield, Exit, Join, and Kill are the remaining 30%.

If something is not precisely specified, we expect you to take a reasonable assumption, clearly explain it in report, and proceed with your implementation.

What to Submit

You should include a file `under code/reports/project2.txt` that *explains how you design the code* and *how your code works*, and *how to run tests*. You should also *indicate what does not work*and *explain your efforts* in order to get partial credits. Also, do not forget to put the *name of your group* in the writeup file. Submit the tgz archive of your code directory as specified in the instructions below. As always, you can submit multiple times. We will use the last version you submit for grading. However, make sure that you keep the name of the turnin file as proj2.tgz for all your uploads.

Turnin Instructions

- Put your report in `code/reports/project2.txt` It should be in plain ascii. In other words, DO NOT use any fancy document writers (e.g. MS Word). You may use vi or emacs or other basic editors.
- Archive and compress the compiled nachosdir (top-level) directory structure (This directory structure should also contain your report, project2.txt). Run the following command:

```
tar c nachosdir | gzip > proj2.tgz
```

- Upload proj2.tgz. The name of the file you will upload **must be**proj2.tgz
- All Done!