OPERATING SYSTEMS – ASSIGNMENT 1

SYSTEM CALLS & SCHEDULING

Responsible TAs: Yarin Kuper and Tsahi Saporta

Introduction

Throughout this course we will be using a simple, UNIX like teaching operating system called xv6: https://pdos.csail.mit.edu/6.828/2018/xv6.html

The xv6 OS is simple enough to cover and understand within a few weeks yet it still contains the important concepts and organizational structure of UNIX. To run it, you will have to compile the source files and use the QEMU processor emulator (installed on all CS lab computers).

- xv6 was (and still is) developed as part of MIT's 6.828 Operating Systems Engineering course.
- You can find a lot of useful information and getting started tips here: https://pdos.csail.mit.edu/6.828/2018/overview.html
- xv6 has a very useful guide. It will greatly assist you throughout the course assignments: https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf
- You may also find the following useful: https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf

You can download xv6 sources for the current work by executing the following command:

`git clone` http://www.cs.bgu.ac.il/~os192/git/Assignment1

Task 0: Running xv6

Begin by downloading our revision of xv6, from our course Assignment repository:

- Open a shell, and traverse to a directory in your computer where you want to store the sources for the OS course. For example, in Linux:
  ➔ mkdir ~/os192
  ➔ cd ~/os192
- Execute the following command:
  ➔ `git clone` http://www.cs.bgu.ac.il/~os192/git/Assignment1
- Build xv6 by calling make
  ➔ make
- Run xv6 on top of QEMU by calling:
  ➔ make clean qemu

Task 1: Warm up ("HelloXV6")

This part of the assignment is aimed at getting you started. It includes small changes in the xv6 shell. Note that in terms of writing code, the current xv6 implementation is limited: it supports only a subset of the system calls you may use when using Linux and its standard library, and is quite limited.

1.1. Support the PATH environment variable:

When a program is executed, the shell seeks the appropriate binary file in the current working directory and executes it. If the desired file does not exist in the working directory, an error message is printed. Currently all the user programs are located inside the /bin folder, therefore in order to run a program you should add a prefix "/bin/" to it (for example to run "ls" you should type "/bin/ls").

"PATH" is an environment variable which specifies the list of directories where commonly used executables reside. If, upon typing a command, the required file is not found in the current working directory, the shell attempts to execute the file from one of the directories specified by the ***PATH*** variable. An error message is printed only if the required file was not found in the working directory or any of the directories listed in ***PATH***.

Your first task is to add support for the ***PATH*** environment variable. In order to simplify the support for environment variables we require that the value of the ***PATH*** environment variable will reside in file "`/path`". Namely, each time the shell needs to know the value of ***PATH***, it should read the content of the file "`/path`". This means that each change in the content of the file "`/path`" will cause an update of the value of the ***PATH*** variable. The value of the ***PATH*** variable consists of a list of directories where the shell should search for executables. Each directory name listed should be delimited by a colon ('`:`'). For example, if we want to add the *root directory* and the *bin directory* to the ***PATH*** variable, we can set the content of the "`/path`" file to:

```
/:/bin/:
```

Finally, the shell must be aware of the ***PATH*** environment variable when searching for a program. The first place to seek the binary of the executed command should be the current working directory, and only if the binary does not exist there, the shell must search it in directories defined by ***PATH***. The list of directories can be traversed in any order and must either execute the binary (if it is found in one of the directories) or print an error message in case the program is not found. Note that the user can execute a program by providing to the shell an *absolute path* (i.e., a path which has '/' as its first character) or a *relative path*. Test your implementation by executing a binary which does not reside in your current working directory but is pointed to by ***PATH***. The tests must include commands which use I/O redirection (i.e., file input/output and pipes).

Task 2: Wait, exit and detach systems calls

In most operation systems, the termination of a process is performed by calling an exit system call. The exit system call receives a single argument called "status", which can be collected by a parent process using the wait system call. If a process ends without an explicit call to exit, an implicit call to exit is performed with the status obtained from the return value of the main function. This is not the case in xv6:

- The exit system call does not receive a status and the wait system call does not return it. In addition, no implicit call to exit is performed. The following task will modify xv6 in order to support the common behavior.

In this part you are required to extend the current kernel functionality so as to maintain an exit status of a process and to endow the kernel with an ability to make an implicit system call exit when the process is done. You must add a field to the process control block PCB (see proc.h – the proc structure) in order to save an exit status of the terminated process. Then, you have to change all system calls affected by this change (i.e., exit and wait).

In addition, you will add the detach system call which detaches the child with the given pid from the calling process. After detaching a child process from its parent, the new parent of the child process will be the init process (global variable initproc in proc.c). The system call returns 0 whether the child was successfully detached or -1 when failed. It may fail if the parent process doesn't have a child with the given pid, or when this child has already been detached by its parent.

2.1. Updating the exit system call:

Change the *exit* system call signature to `void exit(int status)`. The `exit` system call must act as previously defined (i.e., terminate the current process) but it must also store the exit status of the terminated process in the proc structure.

- In order to make the changes in the *exit* system call you must update the following files: `user.h, defs.h, sysproc.c, proc.c` and all the user space programs that use the `exit` system call.
- Note, you must change all the previously existing user space programs so that each call to exit will be called with a status equal to 0 (otherwise they will fail to compile).

2.2. Updating the wait system call:

Update the *wait* system call signature to `int wait(int *status)`. The `wait` system call must block the execution of the calling process until any of its child processes is terminated (if any exists) and return the terminated child exit status through the `status` argument.

- The system call must return the ***process id*** of the child that was terminated or ***-1*** if no child exists (or if an unexpected error occurred).
- Note that the `wait` system call can receive `null` (include the file types.h for using null) as an argument. In this case the child's exit status must be discarded.
- Note that like in task 2.1 (exit system call), you must change all the previously existing user space programs so that each call to wait will be called with a status equal to 0 (otherwise they will fail to compile).

Pay attention: when you add/change a system call, you must update both kernel sources and user-space program sources.

2.3. Add the detach system call:

Add the detach system call with the signature `int detach(int pid)`. It called by a parent process in order to transfer a child process with the given `pid` from the parent to the init process (global variable initproc in proc.c). This system call releases the parent process from the need to wait for the child to finish its execution. The detached child process will be cleaned up by the init process when it finishes its execution. If the child with the given `pid` exists, then the detach operation should succeed and returns 0. If the operation fails, detach returns -1 (because the parent has no child whose identifier is `pid`).

After a successful detach operation, once the detached child finishes running, the "zombie!" message should be printed to the console by the init process.

For example, if the current process preforms:

```
pid_t pid;
int first_status;
int second_status;
int third_status;

pid = fork(); // the child pid is 99
if(pid > 0) {
   first_status = detach(99);  // status = 0
   second_status = detach(99); // status = -1, because this process has already
                               // detached this child, and it doesn't have
                               // this child anymore.
   third_status = detach(77);  // status = -1, because this process doesn't
                               // have a child with this pid.
}
```

Write a user space program "sanity.c", test each of the testable cases:

1. Detach operation of an existing child succeeds.
2. Detach operation of a non-existing child returns -1.

Task 3: Scheduling Policies

Important: Read the entire task carefully before you start implement. Read the notes at the end of this task.

Scheduling is a basic and important service of any operating system. The scheduler aims to satisfy several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low-priority and high-priority processes, and so on. The set of rules used to determine when and how to select a new process to run is called a scheduling policy.

You first need to understand the current (e.g., existing) scheduling policy. Locate it in the code and try to answer the following questions: which process the policy chooses to run, what happens when a process returns from I/O, what happens when a new process is created and when/how often scheduling takes place.

In this assignment, you are required to replace the current scheduling policy of xv6 by testing each one of the following policies and measure the impact of these policies on the performance of the system. You will implement these policies using data structures that we have prepared for you, which are explained later in this document.

3.1. Round Robin Scheduling:

In this policy you should implement a round robin scheduling algorithm by using a special queue interface, which we explain later. Each runnable process becomes running when it is removed from the head of the queue. When it finishes its time quantum it is inserted to the tail of the queue.

3.2. Priority Scheduling:

This scheduling policy is based on accumulating values in a manner we explain soon. This accumulation takes process priorities into consideration. Whenever the scheduler needs to select the next process to execute, it will choose the process with the lowest accumulated number. To support this mechanism, you should add a field to the PCB (Process Control Block) data structure (defined in the proc.h file) that will store this accumulated value. For presentation simplicity, we name this field as `accumulator`, but you are free to call it by another name in your implementation. You should implement a new system call: `void priority(int)`, which can be used by a process to change its priority. The priority of a new processes is 5, the lowest priority is 10 and the highest priority is 1 (thus, lower values imply a higher priority). Each time a process finishes its time quantum (that is, each time it exhausts it and remains runnable), the system should add the process' priority to its `accumulator` field in the PCB. We advise you to define this field as type long long (yes, **long long**, this is not a mistake), because its value can grow to be very large.

Each time a new process is created or a process shifts from the blocked state to the runnable state, the system should set the value of its `accumulator` field to the minimum value of the `accumulator` fields of all the runnable / running processes. If it is the only runnable process in the system, the system should set its accumulator value to 0. This

gives high priority to new processes or to processes that are returning from I/O (particularly, in xv6, the call for "read", "write", "printf" and etc don't necessarily block a process. We advise you to use the system call "sleep", in you tests program, to simulate I/O blockings). Whenever the scheduler needs to select a new process to run, it selects a process with the lowest `accumulator` value.
Note that the `accumulator` fields of processes with high priorities (small priority values) grow more slowly, hence the scheduler will favor such processes.

3.3. Extended Priority Scheduling:

Consider what may happen if we also allow priority of value 0.

In this policy we want to extend the range of priorities that can be provided to the `priority(int)` function to 0-10 (instead of 1-10). Scheduling should be implemented similarly to the previous policy (priority scheduling) except for the following changes. First, note that after a process sets its priority to zero, the previous algorithm will add zero to its accumulator when it finishes its time quantum. Hence, the accumulator value of this process will not change. In this case, the same process may be chosen by the scheduler again and again forever and other processes may starve. In the extended priority scheduling policy, we would like to solve this starvation problem.

One possible solution is to maintain a global variable (of type long long) that counts the number of time quantums that have expired. Each time this counter modulo 100 equals 0, we want to choose another runnable process (if one exists) that did not execute for the longest time. If a few such processes exist, you can choose to schedule any of them. To implement this solution, you would need to add new fields to the PCB (in the proc.h file).

3.4. Changing the scheduling policy:

In order to be able to select a desired scheduling policy, you are required to implement a system call: `void policy(int)`. This system call receives a policy identifier (1 – for Round Robin Scheduling, 2 – for Priority Scheduling and 3 – for Extended Priority Scheduling) as an argument and changes the currently used policy. This system call should retain the priority values and the accumulated values of all existing processes when policy 3 (Extended Priority Scheduling) is selected. When policy 1 (Round Robin Scheduling) is selected, all accumulated values should be initialized to 0, and priority values remain unchanged. When policy 2 (Priority Scheduling) is selected, you should set the priority value 1 to all of the processes having priority 0. The accumulated value remains unchanged. This means that when changing the policy from 2 to 3 and then back again, priorities of 0 are not maintained. Additionally, create a user space program called "policy.c" that receives a single argument, which is the code of the new policy, and performs a call to the `policy` system call.

- The default scheduling policy should be Round Robin Scheduling.

3.5. Measuring the performance of scheduling policies

In class, you learned about several quality measures for scheduling policies. In this task you are required to implement some of them and measure your new scheduling policies performance. The first step is to extend the proc struct (see proc.h) by adding the following fields to it:

- ctime – process creation time.
- ttime – process termination time.
- stime – the total time the process spent in the SLEEPING state.
- retime – the total time the process spent in the READY state.
- rutime – the total time the process spent in the RUNNING state.

These fields retain sufficient information to calculate the turnaround time and the waiting time of each process.

Upon the creation of a new process the kernel will update the process' creation time. The fields (for each process state) should be updated for all processes whenever a clock tick occurs (see trap.c) *(*you can assume that the process' state is SLEEPING only when the process is calling the system call sleep). Finally, care should be taken in marking the termination time of the process (note: a process may stay in the 'ZOMBIE' state for an arbitrary length of time. Naturally, this should not affect the process' turnaround time, wait time, etc.).

Since all this information is retained by the kernel, we are left with the task of extracting this information and presenting it to the user. To do so, create a new system called `wait_stat`, which extends the wait system call:

`int wait_stat(int* status, struct perf * performance)`, where the second argument is a pointer to the following structure:

```
struct perf {
    int ctime;
    int ttime;
    int stime;
    int retime;
    int rutime;
};
```

The `wait_stat` function will return the pid of the terminated child process or -1 upon failure.

Notes:

- **Inside the code of the scheduler function you are not allowed to iterate over the processes' table (ptable.proc) neither directly nor indirectly. The only case you are allowed to iterate over the processes' table (and apparently should) is the one when policy 3 is active - once every 100 time qunatums** (for solving the starvation problem described above). Instead, you are provided with an interface of a few data structures that support the operations you need to perform in each policy. This interface is documented in the file schedulinginterface.h and is also described in an appendix in this document.
- Both policies 2 and 3 use the same data-structure – the priority queue. In policy 3, once every 100 time quantums, you need to iterate over the ptable.proc for selecting the process that did not run for the longest time. After selecting it, you should remove it from the priority queue. This operation should be done inside the scheduler's function code. That is the only case you are allowed to iterate over the process table inside that function.
- The operations on this interface's data structures are asynchronous. You should hold the ptable.lock before you perform such operations on these data structures.

Your user space program, sanity, should also tests your implementation and the performance of the currently selected scheduling policy. Make sure to print the performance measure collected during the test.

Submission Guidelines:

Make sure that your Makefile is properly updated and that your code compiles with no warnings whatsoever. We strongly recommend documenting your code changes with comments – these are often handy when discussing your code with the graders.

Due to our constrained resources, assignments are only allowed in pairs. Please note this important point and try to match up with a partner as soon as possible.

Submissions are only allowed through the submission system. To avoid submitting a large number of xv6 builds you are required to submit a patch (i.e. a file which patches the original xv6 and applies all your changes). You may use the following instructions to guide you through the process:

Back-up your work before proceeding!

Before creating the patch review the change list and make sure it contains all the changes that you applied and noting more.

Execute the command:

```
> make clean
```

Modified files are automatically detected by git but new files must be added explicitly with the '*git add*' command:

```
> git add . -Av; git commit -m "commit message"
```

At this point you may examine the differences (the patch):

```
> git diff origin
```

Once you are ready to create a patch simply make sure the output is redirected to the patch file:

```
> git diff origin > ID1_ID2.patch
```

- Tip: Although grades will only apply your latest patch, the submission system supports multiple uploads. Use this feature often and make sure you upload patches of your current work even if you haven't completed the assignment.

Finally, you should note that the graders are instructed to examine your code on lab computers only!

We advise you to test your code on lab computers prior to submission, and in addition after submission to download your assignment, apply the patch, compile it, and make sure everything runs and works.

Tips and getting started

Take a deep breath. You are about to delve into the code of an operating system that already contains thousands of code lines. BE PATIENT. This takes time!

Debugging

You can try to debug xv6's kernel with gdb (gdb/ddd is even more convenient). You can read more about this here: http://zoo.cs.yale.edu/classes/cs422/2011/lec/l2-hw

Appendix

types.h file:

```
#pragma once

typedef unsigned int      uint;
typedef unsigned short  ushort;
typedef unsigned char    uchar;
typedef uint              pde_t;
typedef int            boolean;

#define null 0

#ifndef __cplusplus

#define false 0
#define true 1


#endif
```

schedulinginterface.h file:

```
#pragma once

#include "types.h"

struct proc;

/**** The implementation of this is found in the ass1ds.cpp file. ****/

//The following c-structs are holding pointers to functions as fields, to make the usage
//easier, like in java.
//For example, suppose we have an instance "pq" of type "PriorityQueue", to invoke the
//isEmpty method just write:
//  boolean ans = pq.isEmpty();

//This structure holds the RUNNABLE processes - Policies 2 & 3
typedef struct PriorityQueue {
         //Checks whether this queue is empty
         boolean (*isEmpty)();

         //This function puts the given process to the priority queue.
         //It returns true if the operation succeeds. This operation may fail if you didn't
         //manage the structures correctly.
         boolean (*put)(struct proc* p);

         //Stores the value of the minimum accumulator inside the given accumulator pointer.
         //Returns true iff the queue isn't empty
         boolean (*getMinAccumulator)(long long* accumulator);

         //Extract a process with the minimum accumulator from the priority queue.
         //If this queue is empty it returns null.
         struct proc* (*extractMin)();

         //Call this function when you need to switch between policies.
         //This function transfers all the mapped process to the RoundRobinQueue.
         //It returns true if the operation succeeds. This operation may fail if you didn't
         //manage the structures correctly.
         boolean (*switchToRoundRobinPolicy)();

         //Extracts a specific process from the queue.
         //Use this function in policy 3 (Extended priority) once every 100 time quanta.
         //This function returns true if it succeeded to extract the given process,
         //it may fail if you didn't manage the data structures correctly.
         boolean (*extractProc)(struct proc* p);
} PriorityQueue;
```

```
//This structure holds the RUNNABLE processes - Policy 1
typedef struct RoundRobinQueue {
	//Checks whether this queue is empty.
	boolean (*isEmpty)();

	//Enqueue the given process to the queue - FIFO manner.
	//It returns true if the operation succeeds. This operation may fail if you didn't
	//manage the structures correctly.
	boolean (*enqueue)(struct proc* p);

	//Removes the first process from the queue and returns it - FIFO manner.
	//If the queue is empty it returns null.
	struct proc* (*dequeue)();

	//Call this function when you need to switch between policies.
	//This function transfers all the mapped process to the PriorityQueue.
	//It returns true if the operation succeeded. This operation may fail if you didn't
	//manage the structures correctly.
	boolean (*switchToPriorityQueuePolicy)();
} RoundRobinQueue;


//This structure holds the RUNNING processes
typedef struct RunningProcessesHolder {
	//Checks whether this structure is empty.
	boolean (*isEmpty)();

	//Adds a process to the structure.
	//It returns true if the operation succeeded. This operation may fail if you didn't
	//manage the structures correctly.
	boolean (*add)(struct proc* p);

	//Removes the given process from this structure.
	//Returns true iff the process was in the structure.
	boolean (*remove)(struct proc* p);

	//Stores the value of the minimum accumulator inside the given accumulator pointer.
	//Returns true iff the structure isn't empty.
	boolean (*getMinAccumulator)(long long *accumulator);
} RunningProcessesHolder;
```

Enjoy !!!