OS202: Assignment 2

Signals and Synchronization

24/04/2020

Responsible TAs: Or Dinari, Linoy Barel
Due Date: 12/5/2020 23:59

1 Introduction

The main goal of the assignment is to teach you about Xv6 synchronization mechanisms and process management. Initially, you will implement a signal framework in Xv6, implementing the *sigaction*, *sigprocmask* and *sigret* system calls, and all the required infrastrcture, which is specified at section 2. Then you will implement the atomic Compare And Swap (CAS) instruction, and use it to replace Xv6 processes management synchronization mechanism with your own lock-free implementation (using CAS). You will write a sanity test for testing both parts of the assignment, in addition you will need to make sure that all of the usertests pass.

The assignment is composed of four main parts:

1. Implement a simple signals framework.
2. Using the signals framework, create a multi process application.
3. Add CAS support for Xv6.
4. Enhance the Xv6 process management mechanism via a lock-free synchronization algorithm.

Use the following git repository:
`http://www.cs.bgu.ac.il/~os202/git/Assignment2`
If you see *error: inflate: data stream error* or *error 302*, ignore it. You can verify the repo works by compiling and running usertests.

 Before writing any code, make sure you read the **whole** assignment.

2 Implementing Signals In Xv6

In practical session 3 you were introduced to the idea of signals. Using signals is a method of Inter-Process Communication (IPC). In this part of the assignment, you will add a framework that will enable the passing of signals from one process to another.

2.1 Creating the signal framework

2.1.1 Updating the process data structure

In order to implement signals, you will initialy have to enhance the process data structure (located at *proc.h*) to accomodate the required features:

- **Pending Signals-** 32bit array, stored as type *uint*.
- **Signal Mask-** 32bit array, stored as type *uint*.
- **Signal Handlers-** Array of size 32, of type *void**.
- **User Trap Frame Backup-** Pointer to a trapframe struct stored as *struct trapframe**.

In addition to modifying the *proc* data strcture, you will add the following macros:

```
#define SIG_DFL 0 /* default signal handling */
#define SIG_IGN 1 /* ignore signal */

#define SIGKILL     9
#define SIGSTOP    17
#define SIGCONT    19
```

2.1.2 Updating process creation behavior

Now you will need to modify the existing behavior in order to use signals:

1. **Default Handlers-** The default handler, for all signals, should be *SIG_DFL*.
2. **Process creation-** When a process is being created, using *fork()*, it will inherit the parent's signal mask and signal handlers, but will not inherit the pending signals.
3. **Executing a new process-** When using *exec*, we will return all custom signal handlers to the default, note that *SIG_IGN* and *SIG_DFL* should be kept.

2.1.3 Updating the process signal mask

You will implement a new system call:

```
uint sigprocmask(uint sigmask)
```

This will update the process signal mask, the return value should be the old mask.

2.1.4 Registering Signal Handlers

A process wishing to register a custom handler for a specific signal will use the following system call which you should add to Xv6:

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
```

This system call will register a new handler for a given signal number (signum). sigaction returns 0 on success, on error, -1 is returned. If oldact isn't NULL, it will recieve the old signal handler (sigaction struct). You will define the new struct required for sigaction.

```
struct sigaction {
    void    (*sa_handler)(int);
    uint    sigmask;
}
```

sa_handler is the signal handler, and *sig_mask* specifies a mask of signals which should be blocked during the execution of the signal handler. *SIG_IGN* , *SIGSTOP*, *SIGKILL*, *SIGCONT*, *SIG_DFL* are valid values for the signal handler field, in addition to a pointer for a user space signal handler. Any positive value is valid for sigmask, see PS3.In the sigaction struct *oldact* you will store the previous sigaction.

 Make sure that *SIGKILL* and *SIGSTOP* cannot be modified, blocked, or ignored! Attempting to modify them will results in a failed sigaction.

For additional info regarding the *sigaction* check the man.

2.1.5 The sigret system call

You will implement a new system call:

```
1  void sigret(void);
```

This system call will be called implicitly when returning from user space after handling a signal. This system call is responsible for restoring the process to its original workflow, its purpose will be clear after section 2.4.

2.2 Sending a Signal

2.2.1 Updating the kill system call

So far, we described how a process should register new handlers. Next we will add the ability to send a signal to a process. You will modify the existing *kill* system call, and change it to the standard linux kill system call:

```
1  int kill(int pid, int signum);
```

Given a process id (pid) and a signal number (signum), the kill system call will send the desired signal to the process pid. Upon successful completion, 0 will be returned. A -1 value will be returned in case of a failure (think about the cases where kill can fail).

2.2.2 Fixing current kill occurrences

Now you will make sure kill is always used properly, there are not many calls to it in Xv6 code.
When finishing section 2.4 *usertests* should pass, and in addition the program *'kill'* will work, thus enabling you to write in shell *'kill* $\langle pid \rangle$ $\langle signal \rangle$*'* in order to send signals to processes.

2.3 Implement kernel space signals

Not all signals are being handled in the user space, specifically, *SIGSTOP*, *SIGKILL*, *SIGCONT*, are all handled in the kernel space. You will now implement those signals:

- **SIGKILL-** Will cause the process to be killed, similar to the original Xv6 *kill*.
- **SIGSTOP-** Will cause the process to freeze until *SIGCONT* is received.
- **SIGCONT-** Will cause a process to resume, if freezed by a previous *SIGSTOP*.

After a process handles *SIGSTOP* it will become suspended (frozen), only when in this mode *SIGCONT* should be handled, at all other cases it will be ignored.

2.3.1 SIGSTOP and SIGCONT

When handling *SIGSTOP* the process will be freezed, until a *SIGCONT* is recieved, this cannot be done by simply blocking the process, as then it will not recieve any CPU time, and will not check for pending signals. In order to overcome this, whenever a process is being stopped its state will remain ready but whenever it receives CPU time, the system will check for a pending *SIGCONT* signal. if none are recieved it will yield the CPU back to the scheduler.

2.4 Handling Signals

When a process is about to return from kernel space to user space (using the function *trapret* which can be found at *trapasm.S*) its pending signals must be checked by the kernel. If a pending signal exists and it is not blocked by the process signal mask then the process must handle the signal. The signal handling can be done by either discarding the signal (if the signal handler is SIG_IGN) or by executing a signal handler.
Executing a kernel space signal handler is straightforward, and should be done before returning to user space. If the signal handler is defined to be SIG_DFL, it will attempt to execute the proper default action, for $SIGSTOP$, $SIGCONT$ and $SIGKILL$ it will attempt to execute the behavior you implemented at 2.3, for all other signals the default should be the $SIGKILL$ behavior.
User space handlers (as can be defined by the *signal* function) are more difficult to execute:
Initialy, you must create a backup of the user current trap frame, backing up all relevant registers. You should use the field you added earlier to *struct proc* for holding the trapframe backup.
After backing up the user trap frame, you will have to modify the user space stack and the instruction pointer of the process, in order to execute the user-space handler. This requires knowledge of conventions of function call. You can refresh your memory regarding function call conventions here. There are three major steps that must be performed in order to make a function call:

- Push the arguments for the called function to the stack (in our case, the signal number)
- Push the return address to the stack
- Finally jump to the address of the called function

Pushing arguments and the return address to the user space stack is a straightforward operation over the process trapframe once you understand the function call conventions. In order to execute the body of the signal handler upon return to user space, one must update the instruction pointer's value to be the address of the desired function. When the signal handler returns, the user space program should continue from the point it was stopped before. Thus, one can naturally think that the return address that should be placed on the stack as the return address of the signal handler should be the previous instruction pointer (before changing it to point to the signal handler). However, this will not work. Since the signal handler can change the CPU register values this can cause unpredictable behavior of the user space program once jumping back to the original code. In order to solve this problem, you must save the CPU registers values before the execution of the signal handler and restore them after the execution of the signal handler finishes. You will use the field you created in *struct proc* for holding the trapframe backup. When the signal handler finishes, your code must return to kernel space in order to restore the original trapframe. This is the responsibility of the *sigret* system call, which will only restore the original trapframe.
The main problem here is that the signal handler can accidentally not call the sigret system call on exit and this may cause an unpredictable behavior in the user code. To solve this problem, you need to "inject" an implicit call to the sigret system call after the call to the signal handler. This can be done by modifying the user space stack directly, creating an explicit call to sigret in the user stack, and pointing the return address of the signal handler to the sigret call. You can do so by using the memmove function which is located at string.c. You learned how to create a function and copy the compiled code to another location in the memory in the architecture and splab courses you already took.

Note that you should not support nested user-level signal handling. That is, once a user-level signal handler starts executing, it will execute until it terminates. Just before a signal handler starts executing, the process sigmask should be replaced by the sigaction mask corresponding to the signal. Once the signal handler returns, the value of the process sigmask should be restored to its previous value.

2.5 Testing your signal framework

You should write a user program to test all relevant parts of the framework, the test should include spawning of multiple processes, modifying the handlers using sigaction, restoring previous handlers using the sigaction *oldact*, blocking signals, all possible actions should be tested, and user-space signals as well.

You should update the existing *kill* user program to support signal sending, as in Linux kill shell command.

3 Compare And Swap

Compare And Swap (CAS) is an atomic operation which receives 3 arguments CAS(addr,expected,new) and compares the content of addr to the expected value. Only if they are the same, it modifies addr to be new and returns true. Otherwise, addr's value is unchanged and the CAS operation returns false. This is done as a single atomic operation. At first glance, it is not trivial to see why CAS can be used for synchronization purposes. We hope the following simple example will help you understand this: A multi-threaded shared counter is a simple data structure with a single operation:

```
1  int increment();
```

The *increment* operation can be called by many threads concurrently and must return the number of times that increment was called. The following naive implementation does not work:

```
1  // shared global variable that will hold the number
2  // of times that increment was called.
3  int counter;
4  int increment() {
5    return counter++;
6  }
```

This implementation does not work when it is called from multiple threads (or by multiple CPUs which have access to the same memory space). This is due to the fact that the *counter++* operation is actually a composition of three operations:

1. fetch the value of the global counter variable from the memory and put it in a cpu local register
2. increment the value of the register by one
3. store the value of the register back to the memory address refered to by the counter variable

Let us assume that the value of the counter at some point in time is C and that two threads attempt to perform *counter++* at the same time. Both of the threads can fetch the current value of the counter (C) into their own local register, increment it and then store the value C+1 back to the counter address. However, since there were two calls to the function increment this means that we missed one of them (i.e., we should have had counter=C+2). One can solve the shared counter problem using spin locks (or any other locks) in the following way:

```
1  int counter;
2  spinlock lock;
3  int increment() {
4    int result;
5    acquire(lock);
6    result = counter++;
7    release(lock);
8    return result;
9  }
```

This will work. One drawback of this approach is that while one thread acquires the lock all other threads (that call increment simultaneously) must wait before advancing into the function. In the new Xv6 code attached to this assignment you can read the function *allocpid* that is called from allocproc in order to get a new pid number to a newly created process. This function is actualy a shared counter implementation. An alternative solution is to use CAS to solve this problem:

```
1  int conter = 0 ;
2  int increment() {
3    int old;
4    do {
5      old = counter;
6    } while(!CAS(&counter, old, old+1)) ;
7    return old+1;
8  }
```

In this approach multiple threads can be inside the increment function simultaneously, in each call to CAS **exactly one** of the threads will exit the loop and the others will retry updating the value of the counter. In many cases this approach results in better performance than the lock based approach - as you will see in the next part of this assignmnet. So to summarize, when using spinlocks the following pattern is used:

1. capture a lock or wait if already locked
2. enter critical section and perform operations
3. release the lock

If the critical section is long - threads/cpus are going to wait a long time. Using CAS you can follow a different design pattern (one that is in use in many high performance systems):

1. copy shared variable locally (to the local thread/cpu stack)
2. change the local copy of the variable as needed
3. read the shared variable - if its value is the same as that previously read from it, the new value is written to the variable, otherwise retry the whole process. This method actually reduces the "critical section" into a single line of code - 6, because only line 6 can change the shared variable, this single line of code is performed atomically using CAS and therefore no locking is needed.

3.1 Implementing CAS in Xv6

In order to implement CAS as an atomic operation we will use the cmpxchg x86 assembly instruction with the lock prefix (which will force cmpxchg to be atomic). Since we want to use an instruction which is available from assembly code, in this part of the assignment you are required to learn to use gcc-inline assembly. You can take a look at many examples of using inline assembly inside the Xv6 code itself, especially in the file x86.h.

 GCC inline assembly uses AT&T style assembly syntax which is different than the one you learned in the architecture course. The differences in the syntax are described in the provided link above

3.1.1 tasks

1. create a function called cas inside the x86.h file which has the following signature:

```
1    static inline int cas(volatile void *addr, int expected, int newval)
```

2. use inline assembly in order to implement cas using the cmpxchg instruction (note that in order to access the ZF (zero-flag) you can use the pushfl assembly instruction in order to access the FLAGS register).
3. change the implementation of the *allocpid* (inside proc.c) to use the cas shared counter implementation instead of the existing spinlock based shared counter implementation.
4. **check that everythings works!**

3.2 Using CAS

The function *allocproc* in *proc.c* is responsible for allocating an entry in the process table for the new process (as a result of a call to fork). This function chooses an unused entry from the ptable and prepares it for usage. The function uses the spinlock ptable.lock in order to make sure that two CPUs will not allocate the same entry by accident (as a result of race condition). Your task is to remove the usage of *ptable.lock* from *allocproc* and use the new cas function in order to avoid synchronization issues. Your implementation should not use spinlocks.

4 Enhancing Xv6's process management synchronization

Xv6 process management is built around the manipulation of a single process table (implemented as an array of *struct proc* and refered to as *ptable*). In a multi processors/core environment Xv6 must make sure that no two processors will manipulate (either execute or modify) the same process or its table entry - it does so by using a spinlock which is refered to as the *ptable.lock*. The usage of busy waiting may seem strange at first, especially in a performance critical code like the OS kernel, but in the kernel level if a processor finds it has to wait in order to schedule a process - there is nothing else it can do but re-attempt to schedule the process. In the user level on the other hand, busy waiting is wasteful since the processor can actually run another process instead of waiting in a loop. Busy waiting means that a processor will loop while checking an exit condition again and again. Only when the exit condition is satisfied, the processor will exit the loop. In this part you are requested to change the current implementation to use the atomic CAS operation in order to maintain the consistency of the ptable. In the cotext of process management, the main purpose of the synchronization mechanism (ptable.lock) is to protect the transitions between different process states, in other words, to ensure that the transitions between different process states are atomic. For example, consider the transition from the UNUSED state to the EMBRYO state. The ptable lock that you removed earlier solved the race condition that can happen if two processes perform the system call fork concurrently, enter the allocproc function and choose the same proc structure as their new proc. An additional example is the transition from the SLEEPING state to the RUNNABLE state. In this case, a synchronization mechanism is needed in order to prevent the "loss" of a "wakeup" event. At the time the process changes its state to SLEEPING the spinlock ptable.lock is locked and the same spinlock is needed in order to wake the process. This prevents missing the event that wakes up the process. One downside of using ptable.lock as it was used in Xv6 comes from the fact that one CPU that is reading or writing any information about the ptable prevents any other CPU from doing the same (even when their operations do not conflict or create a race condition). This reduces the efficiency of the system. Consider the case where one process p1 executes the fork system call. This system call must allocate a new process structure. Choosing a new process structure must be done while avoiding race conditions for the reasons listed above and therefore it was protected by the ptable.lock. Another process p2, at the same time, may perform the wait system call which also contains a critical section that is also protected by ptable.lock. In such a case, p1 may need to wait until p2 will release ptable.lock. The main issue here is that both of the processes could actually execute their code simultaneously since they cannot affect one another in this specific case. In this part of the assignment you are required to use CAS in order to solve the above problems. Specifically, you need to use CAS in order to make the transitions among the different process states atomic. Before attempting to do so, there are several issues you need to consider which we describe here:

- When the kernel does its work in kernel space it can receive interrupts which will start executing interrupt-handling code. Therefore, when attempting to make the process state transitions the action should be atomic, i.e, the CPU should not be interruped and change its current code path.
- Other CPU's must not interfere with the state transition.

The solution for the first problem is straightforward: if one disables interrupts during the transitions between process states, then handling each transition becomes atomic for the CPU which performs the transition.
The solution for the second problem is not so straightforward. In order to understand why, let us consider the following problematic transitions:

1. Transition from RUNNABLE state to RUNNING state: for such a transition one must ensure that only a single CPU executes the code of a selected process at a given time. Otherwise, two CPUs may execute the same process.
2. Transition from RUNNING state to RUNNABLE state: at a first glance there is no problem with the atomicity of this transition since only the CPU that is currenltly running the process will perform such a transition. But if you examine this transition more carefully, you will find that the first stage of this transition is to change the state of the current process from RUNNING to RUNNABLE and only then to perform context switch to the scheduler. Once the process state is changed to be RUNNABLE, a

different CPU can immediately choose it to be executed. This is problematic because the process is not fully transitioned yet. For example, the registers may not be saved yet which means that the process context is not ready for execution. This means that the state of the process should not be changed from RUNNING to RUNNABLE at this stage but only after it is actually ready to be executed. But, what is the state of a process that is between RUNNING and RUNNABLE? It seems that we need more process states. We will use the negative state value $-X$ (X can be any of the existing states) in order to express: "this process is transitioning to X". Therefore, a RUNNING process will not immediately transition to RUNNABLE, but instead to -RUNNABLE. It will only get changed to RUNNABLE when the transition is completed (i.e., after the context switch).

3. Transition from RUNNING state to SLEEPING.: This transition is similar to transition 2 and therefore can be solved using the new transition states ($-X$).
4. Transition from RUNNING state to ZOMBIE.: This transition is similar to transition 2 and therefore can be solved using the new transition states ($-X$).
5. Transition from SLEEPING state to RUNNABLE.: This transition is similar to transition 2 and therefore can be solved using the new transition states ($-X$).

To summarize, in this part you should only change the files proc.c and proc.h. The tasks in this part are:

1. Remove ptable.lock completely from proc.c.
2. Since acquiring ptable.lock also disables interrupts and releasing the ptable.lock enables the interrupts you need to replace each of these calls with their corresponding pushcli or popcli calls.
3. Examine each state transition and decide its start and end locations. When the transition from X to Y starts, switch atomically the process state to $-Y$ and when it ends atomically switch $-Y$ to Y .
4. The different functions in proc.c may check the different process states and decide on their actions according to the state. As a rule of thumb most such checks should be replaced with a check of the absolute value of the state (e.g., if a function originally checked whether the state of a process p is RUNNING then it should now check if the state of p is RUNNING or -RUNNING). This is not true in all cases.
5. Test your implementation: make sure that *usertests* is passed and create a sanity file that tests the signal framework and synchronization rigorously, you are recommended to expand the file you wrote at section 2.5 for this purpose, spawn multiple processes, each with its own intensive computation task, and see that all processes finish execution.

 Test your implementation with different CPU's count, this can be specified in the makefile.

1. Pay attention that a context switch can be started only from two functions (1) scheduler and (2) sched. A context switch that starts at sched will end on scheduler and a context switch that starts at scheduler will end on any of two functions: (1)*sched* or (2)$forkret$ - make sure you understand why.

2. Special protection is needed (for multiple CPUs) when transitioning into the following states: RUNNABLE, SLEEPING ans ZOMBIE.

5 Submission Guidelines

You should download the Xv6 code that belongs to this assignment here:

```
http://www.cs.bgu.ac.il/~os202/git/Assignment2
```

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:

1. Backup your work before proceeding!
2. Before creating the patch review the change list and make sure it contains all the changes that you applied and nothing more. 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"
   ```

3. At this point you may examine the differences (the patch)

   ```
   > git diff origin
   ```

4. 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
   ```

5. 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
6. 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, create a clean Xv6 folder (by using the git clone command), apply the patch, compile it, and make sure everything runs and works. The following command will be used by the testers to apply the patch:

   ```
   > patch -p1 < ID1_ID2.patch
   ```