OS162: Assignment 2

Operating Systems Synchronization and Signals

TAs in charge

Vadim Levit levitv@post.bgu.ac.il
Benny Lutati bennyl@post.bgu.ac.il

Due Date: 30.04.2016

1 Introduction

The assignment main goal is to teach you about XV6 synchronization mechanisms and process management. First, you will learn and use the Compare And Swap (CAS) instruction by adding support for the use of the atomic CAS instruction. Next, you will add simple signals for XV6 that will also use CAS. You will use signals to create an application that coordinates multiple processes. In the last part of the assignment you will replace XV6's process management synchronization mechanism with your own lock-free implementation (using CAS). This will establish an environment that supports a multi-processor/core.

The assignment is composed of four main parts:

1. Add CAS support for XV6.
2. Implement a simple signals framework.
3. Using the signal framework, create a multi process application.
4. Enhance the XV6 process management mechanism via a lock-free synchronization algorithm.

It is very important to read the entire work before starting. Don't be lazy here, the work will be much easier if you read and understand the entire work in advance.

In addition note that this assignment requires that you will use a **new version of XV6**, its location can be found in the Submission Guidelines section.

2 Compare And Swap

Compare And Swap (CAS) is an atomic operation which receives 3 arguments *CAS(addr,expected,value)* and compares the content of *addr* to the content of *expected*. Only if they are the same, it modifies *addr* to be *value* and returns true. This is done as a single atomic operation. CAS is used for many modern synchronization algorithms. At first glance, it is not trivial to see why CAS can be used for synchronization purposes. We hope it is easy to understand by using the following simple example:

A multi-threaded shared counter is a simple data structure with a single operation:

```
int increase();
```

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

```
//shared global variable that will hold the number
//of times that 'increase' was called.
int counter;
int increase() {
  return counter++;
}
```

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 the *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 (due to race conditions). However, since there were two calls to the function *increase* 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:

```
int counter;
spinlock lock;
int increase() {
  int result;
  acquire(lock);
  result = counter++;
  release(lock);
  return result;
}
```

This will work. One drawback of this approach is that while one thread acquired the lock all other threads (that call the increase function) must wait before advancing into the function. In the new XV6 code attached to this assignment you can look at the proc.c file on the function increase_pid 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.

Another possible solution is to use CAS to solve this problem

```
int conter = 0;

int increase() {
  int old;
  do {
    old = counter;
  } while (!CAS(&counter, old, old+1));
  return old+1;
}
```

In this approach multiple threads can be inside the increase function simultanously, 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 summerize, when one is uses spinlocks it must follow the pattern:

1. lock using some variable or wait if already locked
2. enter critical section and perform operations
3. unlock the variable

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

1. copy shared resource locally (to the local thread/cpu stack)
2. change the locally copied resource as needed
3. check the shared resource - if same as originally copied - commit your changes to the shared resource otherwise retry the whole process.

This method actually reduce the "critical section" into a single line of code - 3, because only line 3 effects the shared resource, this single line of code is performed atomically using CAS and therefore no locking is needed.

2.1 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.

2.1.1 Tasks

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

```
static inline int cas(volatile int *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 increase_pid function (inside proc.c) to use the cas shared counter implementation instead of the existing spinlock based shared counter implementation.
4. **check that everythings works!**

2.2 Using CAS

The function *allocproc* in *proc.c* is responsible for allocating a new process (as a result from a call to fork). This function chooses an unused process from the ptable and prepares it for usage. The function uses the spinlock *ptable.lock* in order to make sure that two cpu's will not allocate the same process by accident (as a result of race conditions). 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. Note that you can implement a spinlock using CAS, you are forbidden to do so - i.e., a successfull call to cas should be the last operation in each critical section.

3 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 assignmnet you are requested to implement a basic signal framework which will allow you to send basic information (an integer) between processes. This implementation will support only a single signal handler but unlike the signals you learned about in the practical session, this signal handler can receive two numbers - the one is the pid of the sending processes and the other is a number that is sent by the sending process. The signal framework that you will create includes 4 system calls:

```
//decleration of a signal handler function
typedef void (*sig_handler)(int pid, int value);

//set the signal handler to be called when signals are sent
sig_handler sigset(sig_handler );

//send a signal with the given value to a process with pid dest_pid
int sigsend(int dest_pid, int value);

//complete the signal handling context (should not be called explicitly)
void sigret(void);

//suspend the process until a new signal is received
int sigpause(void);
```

The following subsection describes in detail the above system calls together with their implementation requirements. You are required to follow the instructions given and implement the requested data-structures and system calls.

3.1 Storing and Changing the signal handler

In order to store the signal handler, you will add a new field to *struct proc* (see proc.h). This field will hold a pointer to the current handler (or -1 if no handler is set). As described in practical session 3, both fork and exec system calls modify the signal handlers. We will copy the signal related behavior of fork and exec for our signals implementation.

- The *fork* system call will copy the parent process' signal handler to the newly created child process.
- The *exec* system call will reset the signal handler to be the default (-1).

The new *sigset* system call will replace the process signal handler with the provided one and return the previously stored signal handler.

3.2 Sending a signal to a process

The new *sigsend* system call sends a signal to a destination process. When a signal is sent to a process it is not handled instantly since the destination process may be already running or even blocked. This means that each process must store all the signals which were sent to it but still not handeled in a data structure that we will refer to as the *pending_signals* stack.

Since multiple processes can send signals to the same recipient at the same time, the recipient pending signals stack must support concurrent operations. You must implement a concurrent stack

for storing pending signals using the new cas operation you added in Part 1. As in Part 1, your implementation should not use spinlocks. Note again that one can implement a spinlock by using CAS, but you are forbidden to do so. Specifically, a successfull call to cas should be the last operation in each critical section. Your implementation will support the following operations:

Since we do not want to perform "unsafe" operations inside the kernel, you will not perform any dynamic memory allocations at the kernel. Therefore, the following cstack struct holds an array of preallocated cstackframes. You should use the *used* field in order to check if the cstackframe is already used. If all cstackframes are used, then the cstack is full and cannot accept more items.

```
// defines an element of the concurrent struct
struct cstackframe {
  int sender_pid;
  int recepient_pid;
  int value;
  int used;
  struct cstackframe *next;
};

// defines a concurrent stack
struct cstack {
  struct cstackframe frames[10];
  struct cstackframe *head;
};

// adds a new frame to the cstack which is initialized with values
// sender_pid, recepient_pid and value, then returns 1 on success and 0
// if the stack is full
int push(struct cstack *cstack, int sender_pid, int recepient_pid, int value);

// removes and returns an element from the head of given cstack
// if the stack is empty, then return 0
struct cstackframe *pop(struct cstack *cstack);
```

The *sigsend* system call will add a record to the recipient pending signals stack. It will return 0 on success and -1 on failure (if pending signals stack is full).

3.3 Signal handling

When a process is about to return from kernel space to user space (using the function *trapret* which can be found at *trapasm.S*) it must check its pending signals stack. If a pending signal exists and **the process is not already handling a signal** (i.e., you should not support handling multiple signals at once) then the process must handle the signal. The signal handling can be either discarding the signal (if the signal handler is default) or executing a signal handler when it returns to user space.

In order to force the execution of the signal handler in user space you will have to modify the user space stack and the instruction pointer of the process. This requires knowledge of conventions of function call. You can refresh your your memory regarding function call conventions here. There are three major steps that must be performed in order to make a function call:

1. push the arguments for the called function on the stack
2. push the return address on the stack
3. and finally jump to the body of the called function

Pushing arguments and the return address on 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 finishes its run, the user space program should continue from the point it was stopped before execution of the signal handler. 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 registers 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 should create a new field inside *struct proc* that will hold the original registers values. When the signal handler finishes, your code must return to kernel space in order to restore them. This is the responsibility of the *sigret* system call, which will only restore the CPU registers values for the user space execution (you may backup the whole old trapframe in a new field inside *struct proc*).

The main problem here is that the signal handler can accidently 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 putting this code onto the user space stack and setting the return address from the signal handler to point to the beginning of the injected code. 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.

3.4 Waiting for signals

The final system call that you need to implement is the *sigpause* system call. The *sigpause* system call puts the calling process to sleep until it has a signal to handle. You can look at the *wait* function in *proc.c* for reference for putting processes to sleep. The *sigpause* system call does not really have to return a value, but since there are some code in XV6 that assumes that system calls allways return an int - you are required that it will return the constant 0.

3.5 Testing your signal framework

Once you completed the signal framework you are requested to write a user-level program called *primsrv* that will test your implementation.

primsrv should receive as input from the commandline a number - n and start n new processes that we will refer to as workers and print their pids to the stdout. Next, *primsrv* will read numbers from its stdin. For each number x that is not 0 *primsrv* will choose a worker that is not currently working, send x to it by using signals and continue reading numbers from the commandline. In case there are no idle workers, it should prompt "no idle workers" to its stdout. When a worker receives a number x it will find the first prime number that is larger than x, send it back to *primsrv* and wait for the next number. When *primsrv* receives a prime number p larger than x from a worker it

prints *worker <worker pid> returned <p> as a result for <x>* to its stdout and treat the sending worker as idle. Finally, when *primsrv* receives 0 it will send 0 to all its workers, wait for them to terminate and then exit. When a worker reveives 0 it prints *worker <pid> exit* to its stdout and then exits. Since signals will not get handled if a process is not runnable, if primsrv *primsrv* receives an empty line as input it must ignore it. We will use this behavior to wake up the process so that it will be able to collect signals (as you will see in the example below)

The following is an example for the usage of *primsrv*.

```
$> primesrv 2
workers pids:
4
5
please enter a number: 3
worker 4 returned 5 as a result for 3
please enter a number: 223456789
please enter a number: 123456789
please enter a number: 323456789
no idle workers
please enter a number: <pressing enter>
please enter a number: <pressing enter>
worker 5 returned 223456811 as a result for 223456789
please enter a number: <pressing enter>
worker 4 returned 123456791 as a result for 123456789
please enter a number: 0
worker 4 exit
worker 5 exit
primesrv exit
$>
```

4 Part 3: Enhancing XV6's process management synchronization

Please read this part completely before starting to code.

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*. A spinlock is a lock implementation which internally uses busy waiting. 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 find out 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 may start scheduling processes (while blocking other processors on the same 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.

When talking about 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 *EMBRIYO* state. In part 1 of this assignment you saw that such a transition happens when the kernel wants to allocate a new process structure. The ptable.lock that you removed in part 1 solved the race condition that can happen if two processes perform the system call *fork* concurrently and 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 the missing of 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 to do the same (even when their operations may not conflict or create a race condition). This, reduces the efficiency of the system.

Consider the case where one process p_1 executes the *fork* system call, this system call must allocate a new process structure. Choosing a new process structure is considered a critical section from the reasons listed above and therefore it was protected by the *ptable.lock*. Another process p_2, 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, p_1 may need to wait until p_2 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 requested to use CAS in order to solve the above problems. This will require you 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 and which we will cover here briefly.

When the kernel does it work on kernel space it can receive interrupts which will cause it to change the code it is executing. Therefore, when attempting to make the process state transitions atomic you must consider the following cases:

- atomic for the CPU that performs the transition i.e., the CPU code path cannot be changed externally while transitioning states.
- atomic for the CPUs that are different than the CPU which performs the transition, i.e., other CPUs cannot interfere with the state transition.

The solution for the first case 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 case is not so straightforward. In order to understand the difficulties that can arise due to the second case 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 problems with the atomicity of this transition since only the CPU that is currenltly running the process will perform such a transaction. 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 registeres 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 state $-X$ (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 ($-X$).

To summerize, 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 for all instances.

5. test your implementation: make sure that *usertests* is passed and that *primsrv* is working correctly with this implementation.

Hints:

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. A special protection needed (for multiple CPUs) for transitions of following states: RUNNABLE, SLEEPING ans ZOMBIE.

5 Submission Guidelines

You should download the XV6 code that belongs to this assignment here:

```
https://github.com/os162/xv6-a2.git
```

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
```