OS222: Assignment 2

Synchronization

28/4/2022

Responsible TAs: Hedi Zisling, Roee Weiss Lipshitz
Due Date: 15/5/2022 23:59

0 Don't Panic

In many of the more relaxed civilizations on the Outer Eastern Rim of the Galaxy, the Hitchhiker's Guide has already supplanted the great Encyclopaedia Galactica as the standard repository of all knowledge and wisdom, for though it has many omissions and contains much that is apocryphal, or at least wildly inaccurate, it scores over the older, more pedestrian work in two important respects.
First, it is slightly cheaper; and secondly it has the words DON'T PANIC inscribed in large friendly letters on its cover.

1 Introduction

The main goal of the assignment is to teach you about Xv6 synchronization mechanisms and process management. Initially, you will implement the atomic Compare And Swap (`CAS`) instruction, as you will need it for the next parts.
You will then alter the process scheduling behaviour to add processor affinity, namely, each CPU will have its own process list it can run, and ZOMBIE and SLEEPING processes as well as UNUSED entries will have their own lists as well. This will allow CPUs to manage process scheduling without the need to iterate over UNUSED entries, SLEEPING and ZOMBIE processes, and more importantly, without waiting for other CPUs that iterate over the processes, as each CPU will manage it's own processes, eliminating the need for locks in many critical points in the code. (note: we will not ask you to remove these locks, but your changes will in effect mitigate the waiting times for them)
Finally, you will optimize the process management by implementing load balancing for the system you built, more on that in part 4.
The assignment is composed of three main parts:

1. Add `CAS` support for Xv6.
2. Implement the new process scheduling behaviour. (process list per CPU + ZOMBIE, SLEEPING and UNUSED lists)
3. Implement load balancing.

Reminder: when we refer to the **process control block (PCB)** we mean the `struct proc` defined in `proc.h` of each process.

> Implement the assignment over an unmodified version of the Xv6 code. You can clone it from this git repository: https://github.com/mit-pdos/xv6-riscv

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

2 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** (this is refers to the general idea of `CAS`, however in your implementation in RISC-V, due to its specification, your `CAS` will return **false** when it succeeds, as shown in the code block below). 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 - 1;
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 Xv6 code 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 counter = 0 ;
2  int increment() {
3    int old;
4    do {
5      old = counter;
6    } while(cas(&counter, old, old+1)) ;
7    return old;
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 `CAS` is implemented in a much more efficient way than any lock. 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.

2.1 Implementing `CAS` in Xv6

In order to implement `CAS` as an atomic operation we will use the LR/SC risc-v assembly instruction. Since we want to use an instruction which is available from assembly code, you will need to add a new assembly file to the kernel folder.

2.1.1 tasks

1. create a function called `CAS` inside the `proc.c` file which has the following signature:

```
1    extern uint64 cas(volatile void *addr, int expected, int newval)
2
```

2. create a new `cas.S` file in the kernel folder, with the following format in it:

```
1    .global cas
2    cas:
3    // your code will be here
4
```

   Add the file `$K/cas.o` to the `OBJS` parameter in the makefile.

3. write assembly in order to implement `CAS` using the load-reserved (`lr`) and store-conditional (`sc`) instructions. You can use the example shown here.
   Note: in risc-v calling convention, the arguments are saved in registers rather then on stack (in particular on the registers `a0,a1, ...`). So, for example, to get the first argument, you need to read the value of register `a0`. In addition, say that the value of `a0` is a pointer, to read the value it points to we use parentheses like so: `(a0)`.
   (for more information about risc-v instructions you can turn to the official risc-v manual).
4. change the implementation of `allocpid` (inside `proc.c`) to use the `CAS` shared counter implementation instead of the existing spinlock based shared counter implementation.
5. **check that everythings works!** (this can easily be done by forking multiple times, and forking with the children, making sure no id is repeated)

 when you use your implementation of `CAS` in the following assignmant parts, remember to repeat task 1 as shown above for every file you use `CAS` in.

3 Enhancing Xv6's to add processor affinity functionality

Xv6 process management is built around the manipulation of a single process table (implemented as an array of `struct proc` and refered to as `proc`). 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 spinlocks placed inside each *PCB* (Process Control Block) referred to as `p.lock`. When managing processes (as you've seen in **assignment 1** in the context of scheduling) the CPUs iterate over the process table, acquiring each process's lock before they check and/or modify its fields. This operating method can create an undesired behaviour where multiple CPUs have to compete for locks as they iterate over all of the processes, which can hinder the system performance.

To solve this problem, you will implement a processor affinity system. Instead of the single process list Xv6 uses, your new system will use a ready process list for every CPU, and 3 more process lists, UNUSED entries, SLEEPING and ZOMBIE:

- CPU ready lists: contain processes that are RUNNABLE.
- UNUSED list: contains all UNUSED process entries.
- SLEEPING list: contains all SLEEPING processes.
- ZOMBIE list: contains all ZOMBIE processes.

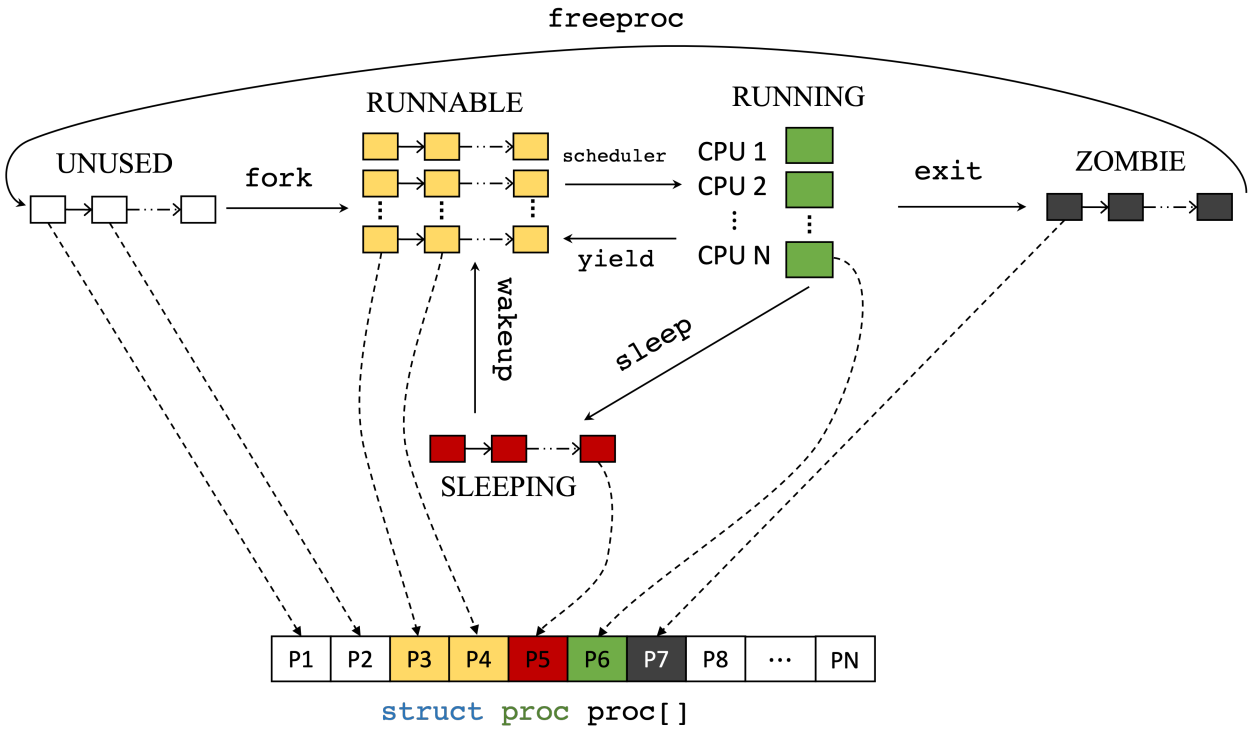
The new lists, with the behaviour specified below, will greatly reduce the number of critical sections where CPUs have to compete for locks, thus improving multi-programming performance.

3.0.1 Concerning list implementation

The simplest way to think about the lists is as linked lists. However, implementing linked lists as you normally would is difficult, as allocating memory in the kernel dynamically is not as trivial as in user space. As such, we suggest that any linked list implementation will hold its fields inside the `PCB` (which is allocated statically at system initialization time). In addition, you might find it simpler to use indices into the `proc[]` array to point to the next link in the list rather then using pointers. This could be easier for you to debug.

 We advise that you test your list implementation before moving forward. We also advice that you adhere to encapsulation rules in your implementation, and more specifically, that you make generic functions to manage (eg. adding and removing links) your list implementation.

 **Important: don't change the `proc[]` array located in `proc.c`, build the new system on top of it (as mentioned above). WE DO NOT WANT YOU TO MODIFY THE PROCESS MEMORY ALLOCATION IN THE SYSTEM! If you find yourself dealing with memory layout code YOU ARE DOING SOMETHING WRONG!**



The figure describes what your system should look like. The solid arrows labeled by `function` names describe the transitions from state to state as well as transitions between lists. The striped arrows represent the fact that the lists do not replace the `proc[]` table but are actually referencing entries in it.

3.1 Specifications

3.1.1 Concurrent linked list

In your linked list implementation you are allowed to use the `PCB` locks. This is due to the fact that implementing a lock-free concurrent linked list is a difficult task, not covered in the course material. However, it is relatively straightforward to implement a concurrent linked list that does not use a single lock for the entire list, but does use a lock for every link (the `PCB` locks). To implement that you can use solution described here. Specifically, the solution that uses double locking (of the removed link and it's predecessor) is an elegant and straightforward solution that you might find useful for removal. For insertion of links, it should be sufficient to lock the predecessor of the inserted link (ie. the link that will point to our inserted link).
If you are interested in reading more about a lock free solution (a solution that uses `CAS`) you can check out this paper.

3.1.2 Scheduling specifications

- Each CPU has its own ready (that are in RUNNABLE state) list, and it picks a process to run only from it. The picked process is always the first process in the ready list.
- When a CPU picks a process to run and changes its state to RUNNING, it also removes it from its ready list.
- When a CPU is done running a specific process, it admits it to the end of its ready list.
- Note: In effect, the scheduling algorithm your system is using is **round robin**, similarly to Xv6, but in a more explicit manner as you are using a real queue as opposed to iterating over a list in a circular manner.

3.1.3 Sleeping specifications

- When a process is blocked (goes to sleep), it is moved to the global SLEEPING list until it wakes up.
- When a process wakes up, it should be returned to the CPU's ready list it was last ran on. Thus, add a field to the *PCB* in `proc.h` specifying the number of the CPU the process is affiliated with.
- A process that goes to sleep is responsible for admitting itself to the SLEEPING list.
- A process that wakes up another process is responsible for admitting the woken up process into the correct CPU's ready list.

3.1.4 General specifications

- When creating a new process, the system will pick a process from the UNUSED list. It will then, after initialization, admit it to the end of the fathers CPU's process list (after its state is set to RUNNABLE).
- When a process is killed, it will be removed from its CPU's ready list and admitted to the global ZOMBIE list.
  The killed process is responsible for admitting itself to the ZOMBIE list.
- When a ZOMBIE process is collected, it is removed from the ZOMBIE list and its entry is admitted into the UNUSED list (thus completing the life cycle).
  The process that waited on the ZOMBIE is responsible for admitting the UNUSED entry into the UNUSED list.

3.1.5 Some system calls and functions you will need to add/change

- **Add** a system call to assign current process to a different CPU. The system call will have the following signature:

```
1    int set_cpu(int cpu_num)
2
```

  The system call will return the CPU number if it succeeds, otherwise a negative number. This system call is added for debugging purposes.
  Note: This system call should cause the process to yield the CPU. This is done so it won't keep running on the current CPU as it no longer belongs to its list.

- **Add** a system call to return the current CPU id. The system call will have the following signature:

```
1    int get_cpu()
2
```

  The system call will return the CPU number if it succeeds, otherwise a negative number.

- **Modify** `fork` to admit the new process to the father's current CPU's ready list.
- **Modify** `scheduler` to pick the first process from the correct CPU's list.
- **Modify** `exit` to admit the exiting process to the ZOMBIE list.
- **Modify** `freeproc` to remove the freed process from the ZOMBIE list and admit its entry to the UNUSED entry list.
  Note: it would make more sense to make the changes in the function `wait`, as it contains an iteration over the `proc[]` array, which we wanted to eliminate. However, due to difficulties in modifying the `wait` function, we only ask you to modify `freeproc` and leave `wait` as it is.
- **Modify** `sleep` to admit the calling process into the SLEEPING list.
- **Modify** `wakeup` to iterate over the SLEEPING list to find processes to wake up, remove woken up processes from the the SLEEPING list and admit them to the correct CPU's ready list.

- **Modify** `userinit` to admit the *init* process (the first process in the OS) to the first CPU's list.
- **Modify** `procinit` to admit all UNUSED process entries (which should, at this point, be all the entries in `proc`) to the UNUSED entry list.
- **Modify** `allocproc` to choose the new process entry to initialize from the UNUSED entry list.
- **Modify** `yield` to admit the yielding process to its CPU's ready list.

All the above mentioned additions and changes should be done in the `proc.c` file.

 Test your implementation with different CPU counts, this can be specified in the makefile by changing the parameter `CPUS`.
Note: there is a parameter in `param.h` named `NCPU`. This parameter specifies the maximum number of CPUs and is **not** related to the current number of simulated cores. There is **no need** to change this parameter.

4 Load balancing for the process management system

Now you will implement load balancing mechanisms for the aforementioned system. Initially you will implement auto-balancing at process admittance time, then you will implement another mechanism to balance idle time for CPUs.

4.1 Activation

To turn on/off the mechanisms in part 4, create the compiler flag `BLNCFLG` similarly to how you did it in **assignment 1**, with the options `BLNCFLG=ON`/`BLNCFLG=OFF`.

4.2 Auto-balancing at admittance time

In your current implementation of fork, the new child processes are added to the process queue of their father's CPU. Furthermore, the processes never get a chance to transfer to another CPU (barring the use of the system call you implemented, which is meant to be used for debugging purposes only). This implementation will result in many processes running on a single CPU (for example, all processes created by the shell will run only on the shell's CPU), which is very inefficient, and will greatly decrease CPU utilization and throughput as well as other metrics. To avoid that, you will implement a mechanism to automatically balance the assignment of processes to CPUs at creation and wakeup times.

4.2.1 Specification

- Every CPU process queue will have a counter for the number of processes that were admitted to it since the system started running. Your counters should be of type `uint64`. Assume you will never pass the value of max `uint64`. You can define those counters wherever you see fit, but it is advised they be defined somewhere in `proc.c` or `proc.h`.
- When a process is **created** or is **woken up** it is to be admitted to the least used CPU, that is, the CPU with the lowest counter value. Naturally, this will require changes to `fork` and `wakeup`.
- When a process is newly admitted to a CPU process queue, the counter is incremented using `CAS`. You must **not** use locks when updating the counter.
- There is no need to synchronize the reading of the counter values. Assume that when a process reads the value it is approximately correct.
- There is no need to update the counter when a running process `yield`s and returns to it's processor's queue.

To check your implementation, add a system call to return the counter of a given CPU. The system call should have the following signature:

```
int cpu_process_count(int cpu_num)
```

The system call should return the value of the counter you added if it succeeds, otherwise a negative number.

Note: this method will not guarantee total balance of the system, but rather will promote average balance over time with possible spikes of CPU loads. However, as in many cases in computer science, **good** functionality **most** of the time is sufficient for us.

4.3 Reducing 'idle' time for CPUs

The former enhancement addressed balancing work load of CPUs in the moment of process creation. This however does not address the case of CPUs 'idling' when they have no process on their queue, while other CPUs have many ready processes in their queue. To address this issue, you will implement a mechanism to allow idle CPUs to 'steal' processes from other CPUs.

4.3.1 Specifications

- During scheduling, when a CPU sees that its process queue is empty, it will attempt to steal a RUNNABLE process from another CPU's process queue, and then **run** it.
- To pick a process to steal, the CPU will iterate over other CPUs process queues and steal the first process available by removing it from the list.
- The stolen process will immediately **run** on the stealing CPU, and when it `yield`s it will be admitted to the stealing CPU's ready queue. In other words, the stolen process's affinity is switched to the stealing CPU.
- When stealing a process, the CPU should increment its process count, again using `CAS`.
- To implement this mechanism you again **must not use locks** to maintain lists integrity, only `CAS` operations.

5 Submission Guidelines

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

```
https://github.com/mit-pdos/xv6-riscv
```

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 Moodle course website. 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. Compress the patch file (`ID1_ID2.patch`) to `ID1_ID2.zip`.

6. Go the the Moodle course website, to the **Assignment 2** section, and upload your `ID1_ID2.zip` file. Only one submission is needed (by one of the partners).

7. Finally, you should note that the graders are instructed to examine your code on VM. We advise you to test your code on VM 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
```