Project 3
Improved Process Management

Introduction

This project will enhance process management in xv6. Impacted areas include process scheduling, process allocation, sleep/blocked process management, and zombie processing. New console control sequences will be implemented in support of testing and debugging process management.

Increasing the efficiency of process management will be the principle focus of this project. The current mechanisms are inefficient in that they traverse a single array (`ptable.proc[]`) that contains all processes, regardless of state. In this project, you will:

1. Implement a new approach to process management with improved efficiency. The new approach will encompass
   (a) Process scheduling
   (b) Process allocation and deallocation.
   (c) State transitions for active processes.
   (d) Zombie processing.
2. Expand the console debug facility with new control sequences.
3. Learn about practical issues related to implementing atomicity in managing this new approach.

This project will use a new Makefile flag, `-DCS333_P3P4`, for conditional compilation. This same flag will also be used in project 4, hence the name.

New Lists

You will add "ready", "free", "sleep", "zombie", "running", and "embryo" lists to the current approach. Adding these lists to xv6 **will not** require that you abandon the current array of processes that is created at boot time. Instead, you will create a new structure `struct StateLists` to store these lists, add a field of that structure to `struct ptable`, and add a new field to the `struct proc` in `proc.h` that points to the next process of whatever list a process is a member of. You will have six new pointers in `struct StateLists`: `ready`, `free`, `sleep`, `zombie`, `running`, and `embryo` to point to the first process, or head, of each list when the list is not empty. In this way, you will use these pointers to more efficiently traverse the existing process array when looking for processes in each of the six possible states. You must take into account concurrency when handling these new lists.

Once implemented, every process will be on one of these six lists.

The current process table, `ptable`, is defined at the beginning of `proc.c`:

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

You will add this new structure just above the ptable definition:

```
struct StateLists {
  struct proc* ready;
  struct proc* free;
  struct proc* sleep;
  struct proc* zombie;
  struct proc* running;
  struct proc* embryo;
};
```

and then change the `ptable` structure to:

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  struct StateLists pLists;
} ptable;
```

and change the definition of `struct proc` in `proc.h` to have a pointer, called `next`, to the next item in each list.

Invariants

All processes on the ready list are in the `RUNNABLE` state; able to be scheduled on a processor. All processes on the free list are in the `UNUSED` state; able to be allocated to a new process. All processes on the sleep list are in the `SLEEPING` state; waiting on some future event. All processes on the zombie list are in the `ZOMBIE` state; waiting to be "reaped" by their parent process or the `init` process. All processes on the running list are in the `RUNNING` state; executing on a CPU [1]. All processes on the embryo list are in the `EMBRYO` state; being initialized.

A process must be on one, and only one, list at a time.

The guarantee that a processes exist only on a specific list at a specific time, reflecting its current state, is an **invariant**, or property that remains unchanged during execution[2]. The reality is that there will be times when a process in one of these states will not be on a list. This "in between" time must be carefully hidden from all threads of control except the one manipulating the process. Be *very* careful to not violate invariants when manipulating processes; you risk introducing errors that can lead to a panic or corrupted state within xv6. As a precaution, your code for removing an item from one of the lists is *required* to check to ensure that the process removed is actually in the correct state (i.e., not put there in error) and panic with an appropriate error message if it is not.

Initializing the Lists

The first process under xv6 is created in the routine `userinit()` in the file `proc.c`. This is where you will need to initialize both the free and ready lists.

You should initialize the free list when you first enter this routine. This is because the routine will then call `allocproc()` which is where a process will be removed from the free list and allocated.

At the very end of `userinit()` you will have a `RUNNABLE` process. This needs to be put on the ready list. Since you know that it is the very first process, you can just have `ptable.pLists.ready` point to this process. Do not forget to set the `next` pointer of the process to 0 (NULL).

[1] The maximum number of elements in this list is NCPU, the number of CPUs in xv6 (see `param.h`)

[2] For our purposes, the definition of invariant is *a quantity or expression that is constant throughout a certain range of conditions.* (from dictionary.com)

Be sure to initialize the remaining lists to be empty. While the list pointers will already be set to zero, doing so here makes the requirement that these lists are empty explicit without unduly impacting performance.

List Management

The act of removing a process from a list and moving it to to another list is a manifestation of a *state transition*. All code for managing state transitions is in the file `proc.c`. It is important that you understand *list invariants* and *concurrency* before making these changes to xv6. Be particularly careful to handle the cases where a state transition fails; you will need to ensure that the process goes back to the list that it was removed from in order to maintain the invariant. There are several locations in `proc.c` where a state transition can fail; you will need to update this processing to the new approach. *Not all state transitions are listed here*; there are others that you will need to locate and change as appropriate.

For the scheduler, you will use the "stub" `scheduler()` routine that already exists in `proc.c` and is activated with the new compiler flag. Start by copying the existing scheduler to the new routine and then modifying the code to correctly use the new ready list. This is where a process will transition from the `RUNNABLE` state to the `RUNNING` state. You will also need list management where the state for a process transitions to the `RUNNABLE` state. Be sure to add the scheduled process to the running list when its state transitions accordingly in the *new* `scheduler()`, and remove it from this list in `sched()` when it is about to exit the CPU. A stub routine for `sched()` has been provided.

You will modify the `allocproc()` routine to use the free list when searching for a process in the `UNUSED` state. You will also need to manage the free list where the state for a process transitions to the `UNUSED` state. Be sure to add the process to the embryo list in the routine `allocproc()`. The process will transition from the `EMBRYO` to `RUNNABLE` state in the `userinit()` and `fork()` routines.

Use the "stub" `wakeup1()` routine that already exists in `proc.c` and is activated with the new compiler flag to improve the existing `wakeup1()` routine using the sleep list — be careful, more than one process can be woken here! Start by copying the existing `wakeup1()` routine to the new routine, and then modifying the code to correctly use the new sleep list. You will also need to manage the sleep list where the state for a process transitions to the `SLEEPING` state.

Similarly, modify the "stub" `exit()`, `kill()`, and `wait()` routines to use the `ready`, `sleep`, `zombie`, and `running` lists to remove traversals of the `ptable.proc[]` structure while having the same functionality as the existing code. Once at a time, copy the existing `exit()`, `wait()` and `kill()` routines to their stub routines and modify the code to correctly use the four lists.

Keep in mind that all code for managing processes is in the file `proc.c`. As a further hint, any time that the state of a process is checked or changed, one of the values from `enum procstate` in proc.h will be used. This is a very good use of the C enumerator type.

It should be clear that these lists will improve certain performance aspects of xv6. If this isn't clear to you, be sure to ask as it is rather important.

List Manipulation

You will need some basic knowledge of linked list manipulation (CS 163) to successfully complete this project. Think very carefully about how you will be inserting into/removing from each list. You will partly be graded on the efficiency of your code.

Note: Since all processes on the free list are equivalent and any one can be used to satisfy an allocation request, efficient management of this list will be different from the other lists. The free list can be managed with a complexity of $O(1)$ rather than $O(n)$. Insertion into the sleep, zombie, running and embryo lists can also be accomplished in $O(1)$ time. Finally, `wakeup1()` can be improved to $O(n)$ time using the sleep list.

Hints/Suggestions

Implement one list at a time; implementing the next list only when you are sure that the current list is correctly working. Course staff recommend that you implement the free, ready, and sleeping lists first. Creating helper functions will likely aid greatly in the organization of your code. Code for state transitions not yet converted to use the new lists can continue to use the `ptable.proc[]` structure without issue.

You might find that much of your list manipulation will be similar across some of the lists. You can achieve the equivalent effect of pass by reference in `C++` using pointers in C, being sure to dereference the pointer whenever you're accessing the field. That way, you can write more generic list manipulation functions to avoid copy-pasting code across methods. For example, you can write the following, generic method

```
static int
removeFromStateList(struct proc** sList, struct proc* p)
```

to remove `p` from the state list `sList`, where `*sList` would denote the head. If we want to remove a process `p` from the running list using this method, for example, we would write

```
int rc = removeFromStateList(&ptable.pLists.running, p);
```

where "rc" is the return code that indicates success or failure. You will *not* be graded on whether or not you use this suggestion when writing your code.

Similarly, you might find that you are also writing the same code when checking the state of removed process and possibly panicking the kernel. This is where the `enum` representation of process states come in handy. You can write a separate method

```
static void
assertState(struct proc* p, enum procstate state)
```

that asserts `p->state` is `state`, panicking the kernel if the assertion is false. The `states[]` array can then be used to print out the name of the required process state in the panic message. For example if you want to check that a removed process `p` is in the `RUNNABLE` state, you can call

```
assertState(p, RUNNABLE);
```

Again, you will *not* be graded on whether or not you use this suggestion when writing your code.

As a final example, we can apply the ideas presented above to write the prototype of a generic removal method that removes a process `p` from the given state list `sList` and then asserts that the state of the removed process is `state`. The prototype would be

```
static int
removeFromStateList(struct proc** sList, \
            enum procstate state, struct proc* p)
```

For example, if you wanted to remove a process `p` from the `running` list and make sure it is in the `RUNNING` state, you can call

```
int rc = removeFromStateList(&ptable.pLists.running, RUNNING, p);
```

Again, you will *not* be graded on whether or not you use this suggestion when writing your code.

Conditional Compilation Comment

In this project, you are required to implement alternate versions of entire functions in `proc.c`. The reason for this approach is that if you are having trouble with an implementation, you can "go back" to the old version just by changing the conditional compilation flag (in the case of a single routine). For the next project, it isn't critical that all lists use this new approach, so it will be possible to turn off certain lists that are not working correctly and not delay work on the next assignment. Speak with the course staff if you find yourself needing to use this technique.

New Console Control Sequences

You will add new console control sequences — ctrl−r, ctrl−f, ctrl−s and ctrl−z — that will print information about the ready, free, sleep, and zombie lists, respectively. All the existing control sequences are found in the routine `consoleintr()` in `console.c`. Observe how ctrl−p is handled — you will be doing something similar for ctrl−r, ctrl−f, ctrl−s, and ctrl−z.

ctrl−r

This control sequence will print the PIDs of all the processes that are currently on the ready list. Your output should look something like below:

Ready List Processes:
$1 \rightarrow 2 \rightarrow 3 \ldots \rightarrow n$

where 1 is the PID of the first process in the ready list (the head), and n is the PID of last one process in the list (the tail). The arrow denotes "is linked to".

ctrl−f

This control sequence will print the number of processes that are on the free list. Your output should look something like below:

Free List Size: N processes

where N is the number of elements in the free list.

ctrl−s

This control sequence will print the PIDs of all the processes that are currently on the sleep list. Its format will be identical to ctrl−r above, except that you will title it as *Sleep List Processes* instead of *Ready List Processes*.

ctrl−z

This control sequence will print the PIDs of all the processes that are currently on the zombie list as well as their parent PID. Its format will be similar to ctrl−r above:

Zombie List Processes:
(1, PPID1) $\rightarrow$ (2, PPID2) $\rightarrow$ (3, PPID3) $\ldots \rightarrow$ (n, PPIDn)

where 1 is the PID of the first process in the list (the head) and PPID1 is the PID of the parent of process 1. The PID of the last process in the list (the tail) is n and PPIDn is the PID of parent process of process n.

Required Tests

Your tests must, at a minimum,

1. Demonstrate that the free list is correctly initialized when xv6 is booted. Note that `init` and `sh` should be the only two active processes immediately after boot while the rest are unused. Recall that the `NPROC` variable represents the maximum number of processes in xv6.

2. Demonstrate that the free list is correctly updated when a new process is allocated (state transitions from `UNUSED`) and when a process is deallocated (state transitions to `UNUSED`).

3. Demonstrate the `kill` shell command causes a process to correctly transition to the `ZOMBIE` and then `UNUSED` states.

4. Demonstrate that round-robin scheduling is enforced. Specifically, the processes that are already in the ready list are scheduled before processes added afterwards; any process transitioning to the `RUNNING` state are removed from the front of the ready list. Processes transitioning to the `RUNNABLE` state must be added at the back of the ready list.

5. Demonstrate that the sleep list is correctly updated when a process sleeps (state transitions to `SLEEPING`) and when processes are woken (state transitions from `SLEEPING`).

6. Demonstrate that the zombie list is correctly updated when a process exits (state transitions to `ZOMBIE`) and when a process is reaped (state transitions from `ZOMBIE`).

7. Demonstrate that output for the console commands ctrl−r, ctrl−f, ctrl−s and ctrl−z is correct.

Discussion

Some of the features optimized in this project depend on each other to function correctly. One example is process deallocation and reaping zombie processes. In xv6, a process is deallocated after it has been reaped when in the ZOMBIE state. For this project, it means that the process is managed by the zombie list before it is managed by the free list. Hence, the zombie list *must* work correctly for the free list to be correct. This means you can test that Requirements 1, 2 and 6 are met using a single test command. One possible approach in the implementation of this command, call it `zombieFree` for the sake of discussion, is as follows:

1. Fork a child process until `fork()` fails, storing each child's PID and the total number of children created.

2. When `fork()` fails, print out the total number of children created and then put the parent process to sleep for a few seconds.

3. After waking up from sleep, use the stored child PIDs to kill off all of the children using the `kill()` system call. Only call `kill()` for the n^{th} child after the $n-1^{th}$ child has been deallocated — you can check this by looping on the `wait()` system call until it returns the correct PID, instead of −1.

4. After all of the children have been killed off, exit the test program.

You are *not* required to use `zombieFree` to test Requirements 1, 2 and 6. The command and above discussion are only intended to facilitate your thinking in writing the necessary tests. If you do want to use `zombieFree`, however, think about how you can use ctrl−f and ctrl−z to give you the information you need. Also note that it is possible to use `zombieFree` to meet additional requirements.

Finally, if you have a correct implementation of the elapsed CPU time functionality in Project 2, you can write a simple test that will verify the correctness of Requirement 3 using ctrl−p and several infinitely looping processes.

Voilà If you now look through the code in `proc.c` you will notice that **there are only three locations where the `ptable.proc[]` structure is used**: 1) `userinit()`; 2) `dumpproc()`; and 3) `getprocs()`, although you can update `getprocs()` if you so desire. **All other references to the array have been removed.** You have now *isolated* (or hidden) how processes are *stored* from how they are *used*. Adding this level of abstraction/indirection is very powerful as it allows a wide variety of approaches for handling

storage for processes, even dynamic creation and destruction! Congratulations, you just made process management for xv6 much more like operating systems such as Linux.