Project 3
Improving Process Management

Introduction

This project will modernize process management in `xv6`. All process management will be impacted. New console control sequences will be implemented to support testing and debugging of the new process management facility.

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) Transactional semantics for all process state transitions[1].
2. Expand the console debug facility with new control sequences.
3. Learn to implement complex concurrency control in the `xv6` kernel.

This project will use a new Makefile flag, `-DCS333_P3`, for conditional compilation. Set `CS333_PROJECT` to `3` in the `Makefile` to turn this flag on.

Scheduler

`xv6` already uses a per-CPU round-robin scheduler. It isn't obvious, but if you pay careful attention to how each CPU is managed, you can readily see that the routine `scheduler()` implements a round-robin algorithm, albeit inefficiently. You will be *improving* the efficiency of the scheduler, and making it globally round-robin, rather than per-CPU round-robin.

The round-robin scheduling algorithm is conceptually simple.

1. If there is a process at the head of the *ready list*
   (a) Remove it
   (b) Add it to the running list
   (c) Give it the CPU
2. When a process `yield()`s the CPU
   (a) Remove it from the running list
   (b) Determine next state/list
   (c) Add the process to this next list
   (d) Invoke the scheduler via `sched()`

Once you modify `scheduler()` to use state lists, the implementation will more closely match our high-level description of the round-robin algorithm. Note that the `scheduler()` routine does not perform step 2.

Changing Existing Routines for This and Future Assignments

This assignment requires modifications to several existing functions in `xv6`. For `exit()`, `wait()`, `scheduler()`, `yield()`, `sleep()`, `wakeup1()`, and `kill()`, you **are required** to create a new version of each, where the

[1] We define transactional semantics to be that a transaction executes atomically with respect to other code. The failure of such a transaction will leave the relevant part of the system in the *same state* as before the transaction.

new version is activated with the `CS333_P3` flag, while the old version will still be included in the absence of the new compiler flag. Start by copying the existing function to the new routine and then modifying the code to match the specification in this assignment. For example:

```
#ifdef CS333_P3
void
foo() {
  /* copy of foo() modified for P3 */
}
#else
void
foo() {
  /* original version of foo() */
#endif
```

If a function will vary significantly between projects, you may structure your code as follows (note that `#ifdef X` and `#if defined(X)` are equivalent and can be used interchangeably):

```
#if defined(CS333_P3)
void
foo() {
  /* copy of foo() modified for P3 */
}
#elif defined(CS333_P2)
void
foo() {
  /* copy of foo() modified for P2 */
}
#elif defined(CS333_P1)
void
foo() {
  /* copy of foo() modified for P1 */
}
#else
void
foo() {
  /* original version of foo() */
}
#endif
```

Note that the order matters in this structure, the implementations must be listed from most to least recent project. Otherwise, a prior project's code could be compiled instead of code for the most recent project.

For changes in other functions (typically smaller changes), you may choose to use the approach above, or you may just use conditional compilation within the body of the function. For example:

```
  #ifdef CS333_P3
    x = /* value x should have in P3 */
    y = /* function call we should make in P3 */
  #elif defined(CS333_P2)
    x = /* value x should have in P2 */
    y = /* function call we should make in P2 */
  #elif defined(CS333_P1)
    x = /* value x should have in P1 */
    y = /* function call we should make in P1 */
  #else
    x = /* value x originally had */
    y = /* function call we originally made */
```

```
#endif
```

This approach is most appropriate for routines where code is modified for each project, e.g., `procdump()`. Preprocessor conditionals can be nested, so it is fine to combine these approaches.

One benefit of conditionally compiling your code for Project 3 is that if you are having trouble with a specific function, you can "go back" to the old version of that one routine just by changing the conditional compilation flag enclosing that one 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 turn off certain functionality as there are dependencies.

State Lists

You will add `RUNNABLE`, `UNUSED`, `SLEEPING`, `ZOMBIE`, `RUNNING`, and `EMBRYO` lists to the current approach. Each list corresponds to a state of the `xv6` state transition model. Adding these lists to `xv6` **will not** require that you abandon the current array of processes that is created at boot time, but you will *hide* its use. Instead, you will add a new array `struct ptrs list` to store these lists in `struct ptable`, and add a new field `struct proc *next` to the `struct proc` in `proc.h` that points to the next process for whatever list a process is a member. You will have `statecount` (6) new pointers in `list`: `list[RUNNABLE].head`, `list[UNUSED].head`, `list[SLEEPING].head`, `list[ZOMBIE].head`, `list[RUNNING].head`, and `list[EMBRYO].head` to point to the first process, or head, of each list, or NULL if the list is empty. You will also have six new pointers that point to the tail of each list: `list[RUNNABLE].tail`, `list[UNUSED].tail`, `list[SLEEPING].tail`, `list[ZOMBIE].tail`, `list[RUNNING].tail`, and `list[EMBRYO].tail`. 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 new concurrency requirements when handling these new lists.* See the *State Lists and Transitions* and *State Transition Procedure* sections for more details about these concurrency requirements.

Once implemented, every process will be on one of these six lists. Moreover, except as noted below, the code in `proc.c` will no longer reference the `ptable.proc` array.

Adding Lists to `xv6`

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

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

You will now use the data structure type declared just above the ptable definition:

```
#ifdef CS333_P3
struct ptrs {
  struct proc* head;
  struct proc* tail;
};
#endif
```

This just packages up the head and tail pointers in a struct, so we can put them in an array.

Next, add this constant definition,

```
#ifdef CS333_P3
#define statecount NELEM(states)
#endif
```

change the `ptable` structure to include our new array of `struct ptrs`,

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
#ifdef CS333_P3
  struct ptrs list[statecount];
#endif
} ptable;
```

and change the definition of `struct proc` in `proc.h` to have a pointer, called `next`, to the next process in each list. Each list corresponds to a *state* of the `xv6` state transition model as discussed in class.

To give an example of how these lists fit together, `ptable.list[RUNNING].head` will point to the first `struct proc` in the running list. From there, we can follow the chain of `next` pointers until we reach a `struct proc` with a `NULL next` pointer. This is the last node in the list, and `ptable.list[RUNNING].tail` will point to it. When a list is empty, its `head` and `tail` pointers will be `NULL`.

If you are unfamiliar or uncomfortable with using a tail pointer in a linked list, you can get a refresher over at A Comprehensive Guide to Implementation of Singly Linked Lists Using C or this lecture on Linked Lists with Tail Pointers.

State Lists and Transitions

The act of removing a process from a list and moving it 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`.

The new lists are part of the `ptable`. You must use atomic transactions using the `ptable.lock` to access and modify the lists. This means that all code that accesses the new lists can only perform that access while the `ptable` lock is held. Be sure to release the lock when you are finished reading and/or modifying the lists. As long as you are holding the `ptable` lock, no other thread can enter the critical section. Once the lock is released, another thread can acquire the lock and enter the critical section.

In the current kernel, 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, and the `state` field of struct proc will be accessed. This can help you find transitions, and other code that takes process state into account.

Initializing the Lists

The first `xv6` process is created in the routine `userinit()` in the file `proc.c`. This is where you will need to initialize the lists. Two functions have been provided for you to make initialization easier: `initProcessLists()` and `initFreeList()`.

You should initialize all the lists 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. You must hold the `ptable` lock when calling these functions, since they modify the state lists. Going forward, we'll generally assume that you know from the section above that you need to hold the `ptable` lock whenever you're reading or modifying any of the state lists.

New Console Control Sequences

You will add new console control sequences that will print information about certain lists. These sequences are helpful for debugging as you update `xv6` to use the state lists, so it makes sense to implement them as soon as the lists are initialized.

1. control−r: prints information about the `RUNNABLE` list

2. control – f: prints information about the `UNUSED` list
3. control – s: prints information about the `SLEEPING` list
4. control – z: prints information about the `ZOMBIE` list

Detection of control sequences occurs in the routine `consoleintr()` in `console.c`. Observe how control – p is handled as you will do something similar for the new control sequences.

Warning: Unlike `procdump()`, which implements control – p, the new control sequences must use correct concurrency control.

control – r

This control sequence will print the PIDs of all the processes that are currently on the `RUNNABLE` 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 `RUNNABLE` list (the head), and n is the PID of last process in the list (the tail).

control – f

This control sequence will print the number of processes that are on the `UNUSED` list. Your output will be

Free List Size: N processes

where N is the number of elements in the `UNUSED` list.

control – s

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

control – 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 control – r above:

Zombie List Processes:

(PID1, PPID1) → (PID2, PPID2) → (PID3, PPID3) ... → (PIDn, PPIDn)

where PID1 is the PID of the first process on the list (the head) and PPID1 is the PID of the parent of process 1. The PID of the last process on the list (the tail) is PIDn and PPIDn is the PID of parent process of process n.

List Management Functions

The following functions **must** be used for managing the state lists. This code is included in `proc.c`, conditionally compiled to first appear in Project 3.

```
static void initProcessLists(void);
static void initFreeList(void);
static void stateListAdd(struct ptrs* list, struct proc* p);
static int  stateListRemove(struct ptrs* list, struct proc* p);
```

To add `p` to a list (let's use the running list as an example), where `*ptable.list[RUNNING].head` points to the head and `*ptable.list[RUNNING].tail` points to the tail of the list, write

```
stateListAdd(&ptable.list[RUNNING], p);
```

To remove `p` from this same list, write

```
int rc = stateListRemove(&ptable.list[RUNNING], p);
```

where "rc" is the return code that indicates success or failure, *w*hich you need to check.

These helpers are examples of how a suitable abstraction can help to make your code easier to maintain and less prone to errors.

Invariants

We call these bullets the *list invariants*:

- The `RUNNABLE` list is all the `RUNNABLE` processes.
- The `UNUSED` list is all the `UNUSED` processes.
- The `SLEEPING` list is all the `SLEEPING` processes.
- The `ZOMBIE` list is all the `ZOMBIE` processes.
- The `RUNNING` list is all the `RUNNING` processes.
- The `EMBRYO` list is all the `EMBRYO` processes.

Because a process is only in one state at a time, if the bullets above are true, we know that **a process must be on one, and only one, list at a time.**

The bullets about the state lists above, and the consequent guarantee that a process exists on one and only one list at a time, are examples of **invariants**. An invariant is a property that appears to remain unchanged

during execution[2]. The reality is that there will be times when a process will not be on a list – namely, while it is transitioning from one state to the next. This "in between" time, when one of the list invariants is broken, must be carefully hidden from all threads of control except the one manipulating the process using transactional semantics. Be careful to not violate invariants when manipulating processes; you risk introducing errors that can lead to a panic, system hang, or corrupted state within `xv6`. As a precaution, in each atomic transaction, after a process has been removed from its list, the transaction is required to use `assertState()` (see below) to ensure that the process removed is actually in the correct state. This check is required to be performed while the process is not on any list. If the process is not in the correct state, the code is required to panic with an appropriate error message. E.g., a process removed from the `RUNNING` list must be checked to ensure that its state at the time of removal is `RUNNING`. This check will find a process that was placed on an incorrect list, and prevent you from transitioning the process as if it were in the state for the incorrect list, rather than the process' actual state.

Checking Process State

We have provided code that checks the state of a process (usually one we have just removed from a state list) and panics the kernel if the process is not in the state that we expected.

```
static void
assertState(struct proc *p, enum procstate state, const char * func, int line)
```

Every time you remove a process from a state list, use `assertState()` to check that the process was in the state corresponding to that list.

`assertState()` asserts `p->state` is `state` and panics the kernel if the assertion is false, providing a useful message in its call to panic that uses the `states[]` array to print out the name of the required process state, and uses the provided function name and line number to help you figure out which call to `assertState()` failed. The preprocessor will replace `__FUNCTION__` and `__LINE__` with the current enclosing function name and line number, so you can simply include them as parameters as shown below.

For example, to check that a removed process `p` is in the `RUNNABLE` state, call

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

You would make this call after removing `p` from the `RUNNABLE` list.

See the *State Transition Procedure* section below for details on how to check process state during state transitions.

Doing the Project Step-by-Step

You are welcome to take any approach to finishing this project. We suggest the following approach because it will enable you to make a relatively minor change, build and test your code, and then move on to another relatively minor change. Your code will be graded in its final state, so breaking up the changes in a different way, or performing all changes at once, are also viable options.

It is possible to incrementally change `xv6` to use the state lists in `ptable.list[]` one state at a time. For a given state, you can modify all the code in `proc.c` that transitions processes into or out of that state to use that state list. At this point, you should be able to build and run `xv6` and run some tests to make sure that everything is still working. This way, you have a better chance of finding problems quickly, compared to a less incremental approach.

As you modify functions in `proc.c` to use the state lists, make sure that your changes don't accidentally alter the behavior of those functions in unintended ways. Your goal is to generally keep these functions working as before, but extend them to use the state lists for all transitions.

[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)

One important complication you need to address is that some transitions may fail. If a transition fails (for example, when `allocproc()` fails to allocate a kernel stack for the new process `p`), you may need to move transitioning processes back to their original state (and the corresponding list) or do other cleanup, to match the existing behavior of `xv6`. These failure transitions are not depicted on the `xv6` state transition diagram, but you can find them by carefully reading the functions you modify in proc.c.

While Not All Lists Are Used

While you are using some (but not all) state lists, you need to keep track of which state lists are currently in use, and which you haven't gotten to yet. Wherever you are transitioning to a state whose list is in use, you need to add the transitioning process to that list. Wherever you are transitioning out of a state whose list is in use, you need to remove the transitioning process from that list.

On the other hand, wherever you transition into a state whose list is not currently in use, you do *not yet* need to add the process to that list. Similarly, when you transition out of a state whose state list isn't currently in use, you do *not yet* need to remove the process from that list.

Once you move on to using the next new state list, you may need to revisit functions you have previously modified that transition processes into or out of that list's state. When lists are in use for the states at either end of a transition, that transition's `stateListRemove()` should have a corresponding `stateListAdd()`.

State Transition Procedure

To atomically transition one or more processes from some state `A` to some other state `B`, follow this procedure, replacing `A`, `B`, and `p` in the example code as needed:

1. Acquire the ptable lock (if you do not already hold it):

   ```
   acquire(&ptable.lock);
   ```

2. For as many processes as needed:

   (a) Find a process to transition from `A` to `B`, often by searching one or more lists. If you already know which process(es) you're transitioning[3], skip this step.

   (b) If you are using state list `A`: Remove the process from state list `A` (assuming a `struct proc *` called `p`)

   ```
   if (stateListRemove(&ptable.list[A], p) == -1) {
     panic("TODO write a helpful error message here!");
   }
   ```

   (c) Assert that the process was in the correct state for that list

   ```
   assertState(p, A, __FUNCTION__, __LINE__);
   ```

   (d) Update the process' `state` field to `B`

   ```
   p->state = B;
   ```

   (e) If you are using state list `B`: Add the process to state list `B`

   ```
   stateListAdd(&ptable.list[B], p);
   ```

[3]For example, because a process is transitioning its own state by making a system call, or you acquired exclusive ownership of the process in a previous atomic transaction protected by the `ptable` lock

3. Finally, after transitioning all of the processes that are part of this atomic transaction, release the `ptable` lock (if you aren't already doing so later)

   ```
   release(&ptable.lock);
   ```

Here is how this procedure preserves the list invariants, provided both lists `A` and `B` are in use:

- The list invariants should hold for lists `A` and `B` at step 1, at the beginning of the atomic transaction.
- When you remove a process from list `A` in 2(b), the invariant for list `A` is temporarily broken, because `p` is in state `A`, but not on list `A`. You hold the `ptable` lock, so `p` is the only process out of place.
- You change `p`'s state to `B` in 2(d). Now the invariant for list `A` is restored: List `A` is just all of the remaining processes in state `A`, which no longer include `p`, whose state is now `B`. However, now the invariant for list `B` is temporarily broken, because `p` is not on list `B`.
- In 2(e), you restore the list `B` invariant by adding `p` to list `B`. Now both list invariants are restored.
- By the end of the atomic transaction in step 3, both list invariants hold, since they were restored for every process you transitioned.

Getting Started

If you have not already done so, make conditionally compiled Project 3 copies of the `exit()`, `wait()`, `scheduler()`, `yield()`, `sleep()`, `wakeup1()`, and `kill()` routines. As you modify your P3 copies, it can be helpful to look back at the original versions of these functions, and compare what your P3 version does to what the original version does. Also, remember that if you are modifying a function with no separate P3 version, you still need to use conditional compilation.

One List At a Time

Following the incremental strategy above, you can start using state lists in any order. We'll present one possible order below.

A reasonable place to start is the `EMBRYO` state, near the beginning of the process lifecycle. You can then walk through the `xv6` state transition diagram one-state-at-a-time. Check the diagram and investigate `proc.c` to figure out which functions correspond to transitions into and out of a given state. Not all transitions are depicted in the diagram, so you'll need to inspect the code to find some of them.

- `EMBRYO` – Add to `EMBRYO` in `allocproc()`. Remove from `EMBRYO` in `fork()`, `userinit()`.
- `RUNNABLE` – Add to `RUNNABLE` in `fork()`, `userinit()`, `kill()`, `yield()`, and `wakeup1()`. Remove from `RUNNABLE` in `scheduler()`.
- `RUNNING` – Add to `RUNNING` in `scheduler()`. Remove from `RUNNING` in `sleep()`, `yield()`, and `exit()`.
- `SLEEPING` – Add to `SLEEPING` in `sleep()`. Remove from `SLEEPING` in `wakeup1()` and `kill()`.
- `ZOMBIE` – Add to `ZOMBIE` in `exit()`. Remove from `ZOMBIE` in `wait()`.
- `UNUSED` – Add to `UNUSED` in `wait()`, `allocproc()` (transition back from `EMBRYO` on kernel stack allocation failure), `fork()` (transition back from `EMBRYO` on page directory allocation failure). Remove from `UNUSED` in `allocproc()`.

At this point, if all lists are in use for all transitions and `xv6` continues working correctly, you have a guarantee that each transition is mediated through the appropriate state lists. Now, all of the list invariants should hold whenever we aren't in the middle of the atomic transaction for a state transition. However, the code in `proc.c` that searches through processes is still scanning the entire `ptable.proc` array. Your next job is to modify this code to take advantage of the state lists.

Replacing `ptable.proc` loops

Now, it's time to use this guarantee, that every process is on the list for its state, to optimize `xv6` and gain some ordering guarantees, by changing loops that currently scan through the entire `ptable.proc` array (all processes) to only scan through processes in certain states. You now have handy state lists that identify which processes are in each state, so only scanning processes in certain states is easy. Further, in some cases you know that the process you want is at the head of a state list, so you can access it in constant time. After completing each optimization, the optimized function *should no longer loop through* the `ptable.proc` array. Be sure to think through how you expect these changes to affect performance, and ordering guarantees among threads.

Note that a mistake in one of the steps above (modifying `xv6` to use the state lists) could cause these optimizations to break `xv6`. So, if you run into problems as you add optimizations, you need to check both the code for that optimization and all of the other code that uses the state lists that optimization depends on.

You can implement the optimizations in any order, so long as the state list(s) a given optimization depends on are currently in use and are implemented correctly. Whenever you are using a state list, remember that it could be empty, and your code needs to handle that case correctly. As was the case while you were implementing state lists, your kernel should continue to pass all tests after you implement each optimization.

- `scheduler()` – Depends on the `RUNNABLE` list. Find the next `RUNNABLE` process using the `RUNNABLE` list. This also guarantees that `xv6` schedules processes round-robin globally, instead of round-robin per CPU.
- `allocproc()` – Depends on the `UNUSED` list. Find the next `UNUSED` process using the `UNUSED` list instead of by scanning `ptable.proc`.
- `wakeup1()` – Depends on the `SLEEPING` list. Scan through sleeping processes using the `SLEEPING` list. Take care to ensure that your `wakeup1()` continues to check all sleeping processes, even after removing one from the `SLEEPING` list.
- `exit()` – Depends on the `EMBRYO`, `SLEEPING`, `RUNNABLE`, `RUNNING`, and `ZOMBIE` lists. Scan through these lists instead of the entire `ptable.proc` array when searching for the `exit()`ing process' children, taking advantage of the fact that `UNUSED` processes can't be children, and so can be safely skipped.
- `kill()` – Depends on the `EMBRYO`, `SLEEPING`, `RUNNABLE`, `RUNNING`, and `ZOMBIE` lists. Scan through these lists instead of the entire `ptable.proc` array when searching for the process to kill, taking advantage of the fact that `UNUSED` processes don't have PIDs and can't be killed, and so can be safely skipped.
- `wait()` – Depends on the `EMBRYO`, `SLEEPING`, `RUNNABLE`, `RUNNING`, and `ZOMBIE` lists. Scan through these lists instead of the entire `ptable.proc` array when searching for the `wait()`ing process' children, taking advantage of the fact that `UNUSED` processes can't be children, and so can be safely skipped.

Once these changes are completed, the only remaining code in `proc.c` compiled for Project 3 that traverses the `ptable.proc` array should be in `procdump()`, your `getprocs()` helper, and `initFreeList()`.

Required Tests

The list of required tests is in the project rubric.

Before You Submit

As described in the *Survival Guide*, you are required to test that your submission works properly before submission. There is a `Makefile` target to help you. You are **required** to run "`make dist-test`" to verify that your kernel compiles and runs correctly, including all tests. A common error is to not put the name of

new files into `runoff.list`. Once you have verified that the submission is correct, use "`make tar`" to build the archive file that you will submit. Submit your project report as a separate PDF document. **Failure to submit your report in PDF format will result in a grade of zero for the report. No exceptions.**