CS 140 Project 1 Documentation

xv6 Rotating Staircase Deadline Scheduler

Group Name: J. Batrina, A. Convento, Y. Hebron

Group Members:

- Jan Paul Batrina
- Angelo Convento
- Yenzy Hebron

Instructors:

- Sir Juan Felipe Coronel
- Sir Wilson Tan
- Ma’am Angela Zabala

Table of Contents

16. Implementation explanation for `priofork()`

Notion Link

Project 1 Documentation

View this in Notion: https://conventoangelo.notion.site/Project-1-Documentation-c0cb35d868c446eb9390d7710c403a94 Remind to change

https://conventoangelo.notion.site/Project-1-Documentation-c0cb35d868c446eb9390d7710c403a94

GitHub Link

https://github.com/UPD-CS140/cs140221project1-j-batrina-a-convento-y-hebron/compare/initial...phase5

Video Explanation - Google Drive Link

cs140project1.mp4

https://drive.google.com/file/d/1tz89OH9HZINDWIyBGx8JMmch4MLltlsT/view?usp=share_link

💡 **Remark:** For all code blocks to follow, code line numbers are based on the latest commit to the branch `phase5`.

1. List of all phases with working code

The list of all phases with working code are as follows:

- Phase 1 in branch `phase1`
- Phase 2 in branch `phase2`
- Phase 3 in branch `phase3`
- Phase 4 in branch `phase4`
- Phase 5 in branch `phase5`

---

2. All references used with purpose specified

The list of all references we used for implementation of Phases 1-5 are as follows:

1. Project 1 specifications - primary reference for the project.
2. Lab 2 specifications - for how the `ptable` is initialized. Specifically, `ptable` is initialized globally with all `NPROC` PCBs already being created when xv6 starts up, but these PCBs will have their states also initialized to `UNUSED`.
3. Lab 5 specifications - for inspiration for test programs and the `schedlog` syscall implementation. In particular, test.c saved as test_loop3.c and loop.c saved as test_loop.c were used as the initial basis of our test programs.

```
// test_loop3.c
#include "types.h"
#include "user.h"

int main() {
    schedlog(10000);

    for (int i = 0; i < 3; i++) {
        if (fork() == 0) {
            char *argv[] = {"test_loop", 0};
            exec("test_loop", argv);
        }
    }

    for (int i = 0; i < 3; i++) {
        wait();
    }

    shutdown();
}
```

```
// test_loop.c
#include "types.h"
#include "user.h"

int main() {
    int dummy = 0;
    for (unsigned int i = 0; i < 4e8; i++) {
        dummy += i;
    }

    exit();
}
```

4. Kolivas, Con. "Rotating Staircase Deadline CPU Scheduler Policy." Internet Archive, March 17, 2007. - for studying mechanics of original scheduler and reasoning behind design decisions.
5. corbet. "The Rotating Staircase Deadline Scheduler." LWN.net, n.d. https://lwn.net/Articles/224865/. - for assurance regarding the Round-Robin selection process usually maintained for RSDL for which we modeled our scheduling around.

---

3. List of all global variables introduced with purpose specified

We list only those that are not related to system calls.

- We have defined a `struct level_queue` (in proc.h) and added the member `struct level_quue *queue` to `struct cpu`. More information regarding `struct level_queue` in ITEM 11.

```
// proc.h: 1-11 & 23
#include "spinlock.h"

// NOTE: each level is represented as an array with NPROC elements
//       for simplicity (since the previous linke list approach had a lot of mysterious crashes)
struct level_queue {
```

```
  struct spinlock lock;
  // must only be modified by enqueue_proc and unqueue_proc
  int numproc;
  int ticks_left;
  struct proc *proc[NPROC];
};

struct cpu{
  ...
  struct level_queue *queue;   // level queue where proc can be found
};
```

```
// proc.h: 14-24
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
  struct level_queue *queue;   // level queue where proc can be found
};
```

- We have modified the anonymous global struct with declaration `ptable` to accommodate the new `struct level_queue`, as well as the pointers to the active and expired sets

```
// proc.c: 16-20
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  // pointers to start of active and expired sets
  // either active = &level[0] and expired &level[1] or vice versa
  // NOTE: Only active[0..RSDL_LEVELS-1] are correct access, ditto for expired
  struct level_queue *active;
  struct level_queue *expired;
  struct level_queue level[2][RSDL_LEVELS];
} ptable;
```

- As instructed in the Project specifications, we also created `rsdl.h` and included it in `param.h`

```
// param.h: 15
...
#define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
#define FSSIZE       2000 // size of file system in blocks

#include "rsdl.h" // For RSDL scheduler parameters
```

The `rsdl.h` header file defines the `RSDL_LEVELS`, `RSDL_STARTING_LEVEL`, `RSDL_PROC_QUANTUM`, and `RSDL_LEVEL_QUANTUM` in accordance with the sample parameters in section 3.2.

```
#define RSDL_LEVELS           3  // Number of priority levels
#define RSDL_STARTING_LEVEL   0  // Priority level all new processes will be assigned to  (must be from 0 
#define RSDL_PROC_QUANTUM    20  // Length of quantum (in ticks) assigned by default to each process
#define RSDL_LEVEL_QUANTUM  100  // Length of quantum (in ticks) assigned by default to each level
```

- For clarity in code, we also created the `NULL` symbolic constant which is used for indicating errors in functions that return a pointer (e.g. `find_available_queue()`

```
// proc.c: 10
...
#include "proc.h"
#include "spinlock.h"

#define NULL (void *) 0x0
```

- To prevent the contents of spinlock.h from being included more than once, we add header guards to it.

```
// spinlock.h 1, 2, & 13
#ifndef SPINLOCK_H
#define SPINLOCK_H
// Mutual exclusion lock
struct spinlock{
...
#endif
```

---

4. List of changes made to the PCB

The process control block (PCB) structure can be found in `proc.h` and is defined as `struct proc`. The changes we made to it are:

1. The addition of `int ticks_left` which tracks the remaining quantum (in ticks) of a process.
2. The addition of `int default_level` which overrides the `RSDL_STARTING_LEVEL` defined in `rsdl.h` and is utilized mainly by the `priofork()`.

```
// proc.h: 65 & 66
struct proc {
  ...
  char name[16];               // Process name (debugging)
  int ticks_left;              // Remaining process quantum (in ticks)
  int default_level;           // starting level for initial run and during swapping of sets
};
```

---

5. Code representation of the active set and its levels

Because of the intertwined nature of the implementation of both the active set and its levels, and the expired set and its levels, which we have done so in a twin-like, contiguous manner, we have decided to consolidate the explanation of both the code representation of the active set and its levels and the expired set and its levels to ITEM 11. As an appetizer, the code representation boils down to the following new members of proc.c's `ptable` structure:

```
struct level_queue *active;
struct level_queue *expired;
struct level_queue level[2][RSDL_LEVELS];
```

Wherein `active` and `expired` points to either `level[0]` or `level[1]` such that `active ≠ expired`. This representation greatly simplifies set swapping, as we'll see later in ITEM 13.

SEE ITEM 11

---

6. Implementation explanation for initial enqueuing of new processes

Whenever a new process arrives, we first search for a free PCB. Searching for a free PCB is separated from finding the correct level in the queue to place `proc`. It remains the same as in xv6 base.

```
//proc.c: 324-327
  ...
  for(p = &ptable.proc[0]; p < &ptable.proc[NPROC]; p++){
      if(p->state == UNUSED)
        goto found;
  }
```

Jumping to `found`, quantum is initialized to `RSDL_PROC_QUANTUM`. The `default_level` of the process is also set to `RSDL_STARTING_LEVEL`

```
// proc.c: 335 & 336
static struct proc*
allocproc(void)
{
  ...
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->ticks_left = RSDL_PROC_QUANTUM;
  p->default_level = RSDL_STARTING_LEVEL;
```

Since a process created via `allocproc` is still an `EMBRYO`, we defer enqueueing it when the caller of `allocproc()` sets its state to `RUNNABLE`. By doing a `grep` ping on the whole xv6 codebase, it shows that `allocproc()` is only called by `userinit()` and the `priofork()` (originally `fork()`) functions.

```
// proc.c: 397
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  ...
  p->state = RUNNABLE;
  // only enqueue here since we are sure that allocation is successful
  struct level_queue *q = find_available_queue(p->default_level, p->default_level);
  enqueue_proc(p, q);

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

```
// proc.c: 478
// Accessible as either fork(void) or priofork(int) syscalls
int
priofork(int default_level)
{
  ...
  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
  ...
  np->default_level = default_level;  // set priority level
  np->state = RUNNABLE;
   // only enqueue here since we are sure that allocation is successful
  struct level_queue *q = find_available_queue(np->default_level, np->default_level);
  enqueue_proc(np, q);

  release(&ptable.lock);

  return pid;
}
```

In both cases, we simply call `find_available_queue()` with the `default_level` of the process as its first and second arguments. As we will see in ITEM 12 below, this means that we will start searching for the `next_active_level()` starting from `active_start = np->default_level` and going down in each priority level. If no such available level is found, we search for the next available level in the expired set, starting from `expired_start` then downwards in decreasing level priority. The first of these two searches that will succeed returns a pointer to the corresponding level queue.

Finally, after `find_available_queue()` returns the pointer `q` to a level queue, we simply call `enqueue_proc(p,q)` or `enqueue_proc(np,q)` to enqueue the newly created process into the level.

```
// proc.c: 398-400
  p->state = RUNNABLE;
  // only enqueue here since we are sure that allocation is successful
  struct level_queue *q = find_available_queue(p->default_level, p->default_level);
```

```
  enqueue_proc(p, q);

// proc.c: 477-481
  np->default_level = default_level;  // set priority level
  np->state = RUNNABLE;
   // only enqueue here since we are sure that allocation is successful
  struct level_queue *q = find_available_queue(np->default_level, np->default_level);
  enqueue_proc(np, q);
```

`enqueue_proc` simply checks if either the process pointer `p` or the level queue pointer `q` are null, or if there is no space available to store a process in this level queue (`q->numproc >= NPROC`). Since these cases normally shouldn't happen, we call `panic()` with error messages so that such errors/bugs can be easily caught. Since we will be modifying the queue, we wrap the relevant code section acting on `q` with `acquire()` and `release()` of lock `q->lock`. Since our level queues use simple array representation, we simply enqueue the next proc `p` to `q->proc` at index `q->numproc`, then increment `q->numproc` after (which is why the post increment `++` is used).

```
// proc.c: 148-169
void
enqueue_proc(struct proc *p, struct level_queue *q)
{
  if (p == NULL) {
    panic("enqueue of NULL proc node");
    return;
  }

  if (q == NULL) {
    panic("enqueue in NULL queue");
    return;
  }

  acquire(&q->lock);
  if (q->numproc >= NPROC) {
    panic("enqueue in full level");
  } else {
    // enqueue *p and increment number of procs in this level
    q->proc[q->numproc++] = p;
  }
  release(&q->lock);
}
```

---

7. Implementation explanation for dequeuing of exiting processes

`exit()` simply sets the state of a process into `ZOMBIE` and jumps to scheduler. Before doing so, we first remove the process from all levels or unqueue it using `remove_proc_from_levels`.

> 💡 **Remark:** From here on, we say “**unqueue**” instead of “**dequeue**” since dequeque removes elements from the front end of the queue. Meanwhile, in the following implementation, we find the process/es to be removed in the front or somewhere in the middle of the queue (e.g. if state of process at front is set to `SLEEP`) which is why we say **unqueue** to imply removal in the queue but not in a dequeue fashion.

```
// proc.c: 536 in exit()
  // Process exited, remove from its queue
  remove_proc_from_levels(curproc);

  curproc->state = ZOMBIE;
  ...
```

`remove_proc_from_levels()` simply checks each level (of both active and expired) for the proc to be removed and unqueues the said proc via `try_unqueue_proc()` explained in detail later.

```
// proc.c: 232-255
int
remove_proc_from_levels(struct proc *p)
{
  struct level_queue *q;
  int found = 0;
  // Naive implementation: use linear search on each level to find level
  for (int s = 0; s < 2; ++s) {
    for (int k = 0; k < RSDL_LEVELS; ++k){
      q = &ptable.level[s][k];
      if (try_unqueue_proc(p, q) != -1) {
        found = 1;
        break;
      }
    }
    if (found)
      break;
  }

  if (!found) {
    panic("Proc not found in any level");
    return -1;
  }

  return 0;
}
```

To understand `try_unqueue_proc()`, we first define `unqueue_proc_full()`. This function does a series of checks with the existence of a process or a queue, as well as the number of processes in a queue (either `numproc < 0` (negative), `numproc > NPROC` (exceeding), or `numproc == 0` (empty/no processes)). We initialize `found` tag to 0. Because we will be modifying the level queue, we wrap the relevant code section with `acquire` and `release` with `q->lock`.

```c
// proc.c: 173-218
int
unqueue_proc_full(struct proc *p, struct level_queue *q, int isTry)
{
  ...
  if (q->numproc == 0) {
    if (!isTry) {
      panic("unqueue on empty level");
    }
    return -1;
  }

  ...
  int found = 0;
  int i, j;

  acquire(&q->lock);
  for (i = 0; i < q->numproc; ++i) {
    if (q->proc[i] == p) {
      // found proc, remove from current queue linked list
      found = 1;
      break;
    }
  }

  if (found) {
    // move succeeding procs up the queue
    for (j = i+1; j < q->numproc; ++j) {
      q->proc[j-1] = q->proc[j];
    }
    q->numproc--;   // decrement number of procs in this level
  }
  release(&q->lock);

  if (!found) {
    if (!isTry) {
      panic("unqueue of node not belonging to level");
    }
    return -1;
  }

  // we only reach here if unqueue is successful
  return i;
}
```

Since our `struct level_queue` uses a simple array representation, we find the process by looping over its `q->numproc` elements. Since `p` and the elements in `q->proc` are pointers to `struct proc`, we can simply compare them to check if they are the same entity. If we find a match, we set the `found` flag to `1` and break out of the loop.

Since we will remove the `struct proc` pointer at index `i`, we simply move the pointers up to queue by looping over each process pointer in the queue, starting at `i + 1`, and moving it to the position right before it. By doing so, we move the remaining parts of the queue (from index `i + 1` up to the tail of queue) up in the array, effectively leaving no empty space between the head and the tail of the queue in the memory. Consequentially, this overwrites the pointer of `p` in the queue, hence unqueueing it.

By setting the `isTry` (third argument of `unqueue_proc_full`) tag to 0 or 1, we can choose either to panic or not. `try_unqueue_proc()` is simply `unqueue_proc_full()` with `isTry` tag automatically set to 1, which means that we don't panic when we don't find the proc. `unqueue_proc()`, having its `isTry` tag set to 0, panics when proc is not found since normally we only call `unqueue_proc(p, q)` when we are sure that proc `p` is in level queue `q`.

```
// proc.c: 220-230
int
unqueue_proc(struct proc *p, struct level_queue *q)
{
  return unqueue_proc_full(p, q, 0);
}

int
try_unqueue_proc(struct proc *p, struct level_queue *q)
{
  return unqueue_proc_full(p, q, 1);
}
```

---

8. Implementation explanation for process-local quantum consumption

Inside the infinite loop in `scheduler()` (proc.c:615), we search for the next runnable process by looping over levels `k = 0` to `k = RSDL_LEVELS-1` of the active set, check each process in each queue/level, and break out of the two loops if we find the next runnable process (now stored in `p`). We set the `found` tag into 1 from its initialized 0 value. Since `k` is declared outside the initialize statement of the for loop, this also means that `k` now holds the level number in the active set where the currently active process `p` is enqueued.

```
// proc.c: 597-631
void
scheduler(void)
{
  struct proc *p = NULL;
  struct level_queue *q = NULL;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    int i, prev_idx, k, nk;
    int found = 0;
    struct proc *np;
    struct level_queue *nq;
    for (k = 0; k < RSDL_LEVELS; ++k) {
      q = &ptable.active[k];
      if (q->ticks_left <= 0)
        continue;
```

```
      acquire(&q->lock);
      for (i = 0; i < q->numproc; ++i ) {
        p = q->proc[i];
        if(p->state == RUNNABLE && p->ticks_left > 0) {
          found = 1;
          break;
        }
      }
      release(&q->lock);
      if (found)
        break;
    }
```

Like in **Lab 5**, we know that each tick triggers a trap with `trapno == T_IRQ0+IRQ_TIMER`. Thus, we consume the process-local quantum (`int proc_ticks`) or decrement it for each tick through `--myproc()->ticks_left;`. When the process-local quantum is depleted (i.e., `proc_ticks <=  0`), we `yield()` control back to the scheduler.

```
// trap.c: 109-117
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER){
  // level queue must be known
  // NOTE: struct cpu cpus[NCPUS] is a global variable, so all cpus[i].queue members are initialized to NULL
  if (!mycpu()->queue)
    panic("Running process located outside active/expired set.");

  int proc_ticks = --myproc()->ticks_left;
  int level_ticks = --mycpu()->queue->ticks_left;
  if (proc_ticks <= 0 || level_ticks <= 0){
    yield();
  }
}
```

At cases where the process-level quantum was not completely consumed, i.e., process still has quantum remaining when it yields, it is re-enqueued into the same level (`nk = k`). Otherwise, ITEM 9 shows how a process is downgraded to the next lower priority level (through `nk = k + 1`).

```
// proc.c: 685-692
// proc has given up control to scheduler
if (q->ticks_left <= 0) {
    // level-local quantum depleted, migrate all procs
    ...
} else {
    if (p->ticks_left <= 0) {
        // proc used up quantum: enqueue to lower priority
        p->ticks_left = RSDL_PROC_QUANTUM;
        nk = k + 1;
    } else {
        // proc yielded with remaining quantum: re-enqueue to same level
        nk = k;
    }

    // only try to re-enqueue proc if it was not removed before
    // e.g. when it calls exit() (state == ZOMBIE), it removes itself so no need to re-enqueue
    if (q->numproc > 0 && p->state != ZOMBIE) {
      prev_idx = unqueue_proc(p, q);
```

```
    ...
    // find vacant queue, starting from level nk as decided above
    // if no available level in active set, enqueue to original level in expired set
    nq = find_available_queue(nk, p->default_level);
    ...
    enqueue_proc(p, nq);
}
```

Note that this only happens, when level-local quantum is NOT depleted. A part of the events that occur when level-local quantum is consumed is that it also replenishes the quantum of **ALL** procs that are enqueued to it as they are re-enqueued to the next available lower priority level.

💡 **Remark on `k` and `q`:** Since `k` and `q` are going to be referenced a lot in the subsequent items, it is important that we have a nice grasp of them. Please observe the code snippet below, which corresponds to the “search for `RUNNABLE` process” portion of the scheduler:

```
for(;;){
    ...
    // i : just a loop control variable.
    // prev_idx : remnant of previous implementation
    int i, prev_idx, k, nk;
    int found = 0;
    struct proc *np;
    struct level_queue *nq;
    for (k = 0; k < RSDL_LEVELS; ++k) {
      q = &ptable.active[k];
      if (q->ticks_left <= 0)
        continue;

      acquire(&q->lock);
      for (i = 0; i < q->numproc; ++i ) {
        p = q->proc[i];
        if(p->state == RUNNABLE && p->ticks_left > 0) {
          found = 1;
          break;
        }
      }
      release(&q->lock);
      if (found) {
        if (schedlog_active) cprintf("%d|Scheduled PID %d in level %d\n", ticks, p->pid, k);
        break;
      }
    }
```

As can be seen, the outer loop, highlighted in red, uses `k` to traverse through the levels inside of the active set, with it being used as the index that assigns the current level being searched to `q` via `q = &ptable.active[k]`. Meanwhile, the inner loop, highlighted in orange, traverses the processes inside of the current level being searched. The point is that

1. If a `RUNNABLE` process is found in the active set, `k` eventually becomes the **index of the active set level that the selected process `p` resides in**. This specific `k` is then stashed for later use in the `if (found) {...}` case of the scheduler as a convenient value for generating the `active_start` argument for `find_available_queue.`
2. Else if no `RUNNABLE` process is found in the active set, the `k` variable is simply reused in the “no `RUNNABLE` process found” case of the scheduler as a similar but functionally unrelated variable to the one we’ve described above.

`q` works in tandem with `k`, such that if `k` corresponds to case 1 above, then `q` **also points at the level that the selected (or to be scheduled) process resides in**. If `k` corresponds to case 2, then `q` is also simply reused in the “no `RUNNABLE` process found” case of the scheduler (e.g. for set swaps).

9. Implementation explanation for process-local quantum replenishment

The process-local quantum represented as `p->ticks_left` is easily replenished by setting it to `RSDL_PROC_QUANTUM`.

When a proc yields due to depletion of its own quantum (`p->ticks_left <= 0`), it returns execution back to scheduler. Since it will be moved to a lower level (or expired set) by enqueueing it to the next available level starting at `nk = k + 1` (which is one level lower than the current), we need to replenish its quantum.

```
// proc.c: 650, in scheduler()
swtch(&(c->scheduler), p->context);
switchkvm();

// proc has given up control to scheduler
if (q->ticks_left <= 0) {
    // level-local quantum depleted, migrate all procs
    ...
} else {
    ...
    if (p->ticks_left <= 0) {
        // proc used up quantum: enqueue to lower priority
        p->ticks_left = RSDL_PROC_QUANTUM;
        nk = k + 1;
      ...
      // find vacant queue, starting from level nk as decided above
      // if no available level in active set, enqueue to original level in expired set
      nq = find_available_queue(nk, p->default_level);
      ...
      enqueue_proc(p, nq);
   }
```

If moving a process results to it being placed in the expired set, we also replenish its quantum.

```
// proc.c: 703-708
// find vacant queue, starting from level nk as decided above
// if no available level in active set, enqueue to original level in expired set
nq = find_available_queue(nk, p->default_level);
if (is_expired_set(nq)) {
    // proc quantum refresh case 2: proc moved to expired set
    p->ticks_left = RSDL_PROC_QUANTUM;
}
enqueue_proc(p, nq);
```

Also, note that when the level-local quantum is depleted, we move **ALL** processes to the lower level with available (level-local) quantum, or to expired set if none are available. In either case, when this happens we replenish the process-local quantum.

```
// proc.c: 656-659
if (q->ticks_left <= 0) {
    // level-local quantum depleted, migrate all procs
    while (q->numproc > 0) {
        np = q->proc[0];
        // moving to next level OR expired set, replenish quantum
```

```
        np->ticks_left = RSDL_PROC_QUANTUM;

        unqueue_proc(np, q);
        ...
        // move proc to next available level in active set
        // if none, enqueue to original level in expired set
        nq = find_available_queue(k+1, np->default_level);
        // re-enqueue to same level but in active set, or below
        enqueue_proc(np, nq);
    }
```

---

10. Implementation explanation for `schedlog`

Similar to **Lab 5**, we add the following variables in `proc.c`. They are used before indicating if and for how long `schedlog` output should be active. This makes the `schedlog(int n)` implementation simply setting `schedlog_active = 1` and `schedlog_lasttick = ticks + n`

```
// proc.c: 43-50
// Variables for scheduling logs. See schedlog() and scheduler() below
int schedlog_active = 0;
int schedlog_lasttick = 0;

void schedlog(int n) {
  schedlog_active = 1;
  schedlog_lasttick = ticks + n;
}
```

For each iteration in `scheduler()` 's infinite loop (which looks for the next process to run), we check if `schedlog` output is active and if `ticks > schedlog_lasttick` (i.e. `schedlog` output has been already active for the requested amount of ticks). If so, we set `schedlog_active = 0` to suppress its output.

```
// proc.c: 633-635, in scheduler()
for(;;){
    ...
    for (k = 0; k < RSDL_LEVELS; ++k) {
        ...
        if (found)
        break;
    }

    if (schedlog_active && ticks > schedlog_lasttick) {
        schedlog_active = 0;
    }

    if (found) {
```

Just like in **Lab 5**, we print the `schedlog` output when execution returns to `scheduler()` and the next runnable process is found. We check if schedlog output is active, and if it still has not reached the maximum number of ticks it is requested to be active in

```
// proc.c: 646-648, in scheduler()
if (found) {
      ...
      p->state = RUNNING;

      if (schedlog_active && ticks <= schedlog_lasttick) {
        print_schedlog();
      }
```

The actual printing is implemented in a separate `print_schedlog()` function which simply walks through each level (in active set first, then in expired set), prints the "header" for the `schedlog` output of this level, then walks through each process in the level and prints its information in the prescribed format for Phases 4-5. Instead of accessing `ptable.level` directly (which contains all `level_queue`), we create an array of pointers to `ptable.active` and `ptable.expired`, which is the same order we want to print the sets in. We loop over the (index) of each set, active first then expired

```
// proc.c: 52-73
void print_schedlog(void) {
  struct proc *pp;
  struct level_queue *qq;

  struct level_queue *set[] = {ptable.active, ptable.expired};
  for (int s = 0; s < 2; ++s) {
    char *set_name = (is_active_set(&set[s][0])) ? "active" : "expired";
    for (int k = 0; k < RSDL_LEVELS; ++k) {
      qq = &set[s][k];
      acquire(&qq->lock);
      cprintf("%d|%s|%d(%d)", ticks, set_name, k, qq->ticks_left);
      for(int i = 0; i < qq->numproc; ++i) {
        pp = qq->proc[i];
        if (pp->state == UNUSED) continue;
        else cprintf(",[%d]%s:%d(%d)", pp->pid, pp->name, pp->state, pp->ticks_left);
      }
      release(&qq->lock);

      cprintf("\n");
    }
  }
}
```

For each level, we `acquire` and `release` its associated lock so that no other processes (e.g. running in a different CPU) can modify the queue while we are printing it. `UNUSED` processes are skipped since normally they wouldn't be in the queue, or if they are they will probably be removed soon. Each level is printed in its own line, so we print newline at the end of the inner loop.

For printing the label/name of the set, we use the `is_active_set()` function to determine if the `set_name` should be `active` or `expired`. Its implementation is shown below:

```
// proc.c: 27-36
int
is_active_set(struct level_queue *q)
{
  // NOTE: assumes that &ptable.level[0][0] <= q < &ptable.level[1][RSDL_LEVELS]
```

```
  //       since each level queue is created in the contiguous ptable.level
  // q is in active set if its address is within ptable.active
  // otherwise since q is inside ptable.level, then it must be in ptable.expired
  return ptable.active <= q
         && q < &ptable.active[RSDL_LEVELS];
}
```

Since we allocated all `struct level_queue` at the start of the OS in `ptable.level`, then we can use simple address range checking to determine if the `struct level_queue *q` is in the active set. We simply check if it is without the lower bound (`ptable.active <= q`) and upper bound (`q < &ptable.active[RSDL_LEVELS]`) of the active set pointed to by `struct level_queue` array `ptable.active`, which has `RSDL_LEVELS` elements. Since `is_active_set` is only used for pointers pointing inside `ptable.level`, we can be sure that if `q` does not satisfy this bound, it is in the expired set. Hence, `is_expired_set()` is implemented simply as the negation of `is_active_set()`.

```
// proc.c: 38-41
int is_expired_set(struct level_queue *q)
{
  return !is_active_set(q);
}
```

Then, to make `schedlog()` a usable syscall, we add it to `syscall.h` with syscall number `24`.

```
// syscall.h: 25
#define SYS_schedlog  24
#define SYS_priofork  25
```

Then in `usys.S`, we use the `SYSCALL()` macro to generate the `int`errupt code for the `schedlog()` syscall.

```
// usys.S: 34
SYSCALL(schedlog)
SYSCALL(priofork)
```

In syscall.c, we declare the `sys_schedlog()` syscall entry point, which we associate with the syscall number `SYS_schedlog == 24` in the `syscalls` table:

```
// sycall.c: 108
extern int sys_schedlog(void);
extern int sys_priofork(void);
```

```
// sycall.c: 135
static int (*syscalls[])(void) = {
...
[SYS_schedlog] sys_schedlog,
[SYS_priofork] sys_priofork,
};
```

This entry point/syscall handler is implemented in `sysproc.c`, with code similar to `sys_kill()` which also takes an integer argument via `argint`.

```
// sysproc.c: 115-124
int sys_schedlog(void)
{
  int n;

  if(argint(0, &n) < 0)
    return -1;

  schedlog(n);
  return 0;
}
```

We then make this syscall accessible to other parts of the OS by including its function prototype in `defs.h`.

```
// defs.h: 117
void            procdump(void);
void            schedlog(int);
```

Finally, we make this syscall accessible to user programs by including its function prototype in `user.h`.

```
// user.h: 29
// system calls
...
int schedlog(int);
int priofork(int);
```

---

11. Code representation of the expired set and its levels

We present here how the active and expired sets and their respective levels are represented by code.

`ptable.active` and `ptable.expired` points to contiguous arrays with `RSDL_LEVELS` number of `struct level_queue` each, representing each level in the active set or the expired set.

```
// proc.c: 12-21
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  // pointers to start of active and expired sets
  // either active = &level[0] and expired &level[1] or vice versa
  // NOTE: Only active[0..RSDL_LEVELS-1] are correct access, ditto for expired
  struct level_queue *active;
  struct level_queue *expired;
  struct level_queue level[2][RSDL_LEVELS];
} ptable;
```

Each `struct level_queue` contains `ticks_left` for level-local quantum and `struct proc *proc[NPROC]` which is an array of struct proc pointers which point to the process PCBs stored in `ptable.proc`. Essentially, this means that our queues are implemented using simple array representation. Each level queue also has a `lock` which is `acquire`d and `release`d when reading/modifying the level queue to prevent race conditions, mainly in `enqueue_proc()` and `unqueue_proc()`. The `int numproc` member holds the number of processes enqueued to a level queue.

```
// proc.h: 1-11
#include "spinlock.h"

// NOTE: each level is represented as an array with NPROC elements
//       for simplicity (since the previous linked list approach had a lot of mysterious crashes)
struct level_queue {
  struct spinlock lock;
  // must only be modified by enqueue_proc and unqueue_proc
  int numproc;
  int ticks_left;
  struct proc *proc[NPROC];
};
```

The use of `struct proc *proc[NPROC]` is an important layer of abstraction so that the native functions in `proc.c` such as `allocproc()` and `wait()` can co-exist with the RSDL scheduler without much modification. This way, the RSDL scheduler can indirectly manipulate which level and which set each process belongs to via pointers to them without greatly impacting the native functions that have to access `ptable.proc` directly.

The address blocks of each `struct level_queue` is allocated at OS compile time in `ptable.level`. Since each set has `RSDL_LEVELS` queues or levels, the inner/second dimension is `[RSDL_LEVELS]`. Since we have two sets (active and expired), the outer/first dimension of the multi-dimensional array is `[2]`

```
//proc.c: 20
struct {
  ...
  struct level_queue level[2][RSDL_LEVELS];
} ptable;
```

In `pinit()`, we initialize each level queue by setting their `numproc` to `0`, initializing its `ticks_left` to `RSDL_LEVEL_QUANTUM`, and manually “erasing” its contents by filling its `proc` array with `NULL`.

```
// proc.c: 83-101 in pinit()
...
  struct level_queue *lq;
  initlock(&ptable.lock, "ptable");

  // To be sure, explicitly initialize all queues to empty
  acquire(&ptable.lock);
  for (int s = 0; s < 2; ++s) {
    for (int k = 0; k < RSDL_LEVELS; ++k){
      lq = &ptable.level[s][k];
      // NOTE: all queues will have same lock names
      initlock(&lq->lock, "level queue");
```

```
      acquire(&lq->lock);
      lq->numproc = 0;
      lq->ticks_left = RSDL_LEVEL_QUANTUM;
      for (int i = 0; i < NPROC; ++i){
        lq->proc[i] = NULL;
      }
      release(&lq->lock);
    }
  }
  ...
```

The `pinit()` modification, in short, initializes all queues to empty and thereafter set the initial values of `ptable.active` and `ptable.expired`. Hence, the only possible values of `ptable.active` and `ptable.expired` are its initial values (as shown in the modified `pinit()`) and possible state after swap:

```
// proc.c: 104-105
void
pinit(void)
{
  ...
  ptable.active = ptable.level[0];
  ptable.expired = ptable.level[1];
  release(&ptable.lock);
}
```

OR when the sets are swapped

```
ptable.active = ptable.level[1]
ptable.expired = ptable.level[0]
```

---

12. Implementation explanation for transferring a process from the active to expired

As we have seen in ITEMS 6, 8, 9, `find_available_queue()` is used to determine where to enqueue a process. We also note that those sections show that the `ticks_left` member of a process is replenished to `RSDL_PROC_QUANTUM` every time it is re-enqueued, including the case where the process moves from the active to the expired set. We will discuss how the `find_available_queue()` function works and how it can cause a process to move from the active set to the expired set.

```
// proc.c: 290-309
struct level_queue*
find_available_queue(int active_start, int expired_start)
{
  int level = next_active_level(active_start);
  if (level == -1) {  // no lower prio level available
    // re-enqueue in expired set instead, starting at expired_set
    level = next_expired_level(expired_start);
    if (level == -1) {
      // NOTE: shouldn't happen normally
      panic("No free level in expired and active set, too many procs");
```

```
      return NULL;
    }

    // We reach here if we found available queue in expired set
    return &ptable.expired[level];
  }

  // We reach here if we found available queue in active set
  return &ptable.active[level];
}
```

Here we can see that the first argument is `active_start` and the second argument is `expired_start`. We search for the next available level starting from `active_start` and going down in each priority level. If found, we simply return the corresponding `struct level_queue *` for the level found in the active set, (`return &ptable.active[level]`). If no such available level is found (`level == 1`), we search for the `next_expired_level()` starting from `` `expired_start` `` then downwards in decreasing level priority. This is how a process moves from the active to the expired set. Similarly if an available level is found we simply return the corresponding `struct level_queue *` for the level found in the active set, (`return &ptable.expired[level]`). Otherwise, this means that we couldn't find any available level in the active nor the expired set, which normally should not happen. We call `panic()` with an error message in such cases so that they are easier to catch and debug.

```
// proc.c: 278-288
int
next_active_level(int start)
{
  return next_level(start, 0);
}

int
next_expired_level(int start)
{
  return next_level(start, 1);
}
```

From the above, `next_active_level` and `next_expired_level` simply call `next_level` with `start` as its first argument. The only difference is their 2nd argument

```
// proc.c: 257-276
int
next_level(int start, int use_expired)
{
  const struct level_queue *set = (use_expired) ? ptable.expired : ptable.active;
  if (start < 0)
    return -1;

  int k = start;
  for ( ; k < RSDL_LEVELS; ++k) {
    if (set[k].ticks_left > 0 && set[k].numproc < NPROC) {
      break;
    }
  }

  if (k < RSDL_LEVELS) {
```

```
        return k;
    } else {
        return -1;
    }
}
```

As we can see, the second argument is actually treated as a Boolean (actually `int`) flag `use_expired`, which is why `next_active_level` supplies `0` while `next_expired_level` supplies `1`. This is done because these two functions perform very similar things, just using different `set` as determined in the first line of `next_level`. We also ensure that the supplied `start` level is non-negative since levels start at `0`

We then simply check the next available level in the `set` by walking through each level (via a for loop), checking if it still has remaining level-local quantum (`set[k].ticks_left > 0`) and if it still has space to store another `struct proc` pointer (`set[k].numproc < NPROC`). If such a level is found, we `break` out of the loop, with the index of the current level stored in `k`. Otherwise, the loop will continue and if `k == RSDL_LEVELS`, then this means that we have checked all levels below `start` and none of them are available. This is why we only return `k` if `k < RSDL_LEVELS` and return `-1` otherwise to indicate that there is no next available level in the set.

---

13. Implementation explanation for swapping of sets

First, we keep in mind the following reminders from the Project Guide with respect to swapping of sets:

- Set swapping happens every time the active set has no more processes that are `RUNNABLE`.
- The *old* expired set becomes the *new* active set, and the *old* active set becomes the *new* expired set.
- Processes still in the *old* active set (now the *new* expired set) are moved to their original or **default priority levels** in the *new* active set.
- On moving of processes, their FIFO order in their respective levels are preserved before the swap. This was discussed in ITEM 12.

Now for the implementation:

All modifications to implement set swapping can be found in proc.c, and these modifications primarily work hand-in-hand with the ones previously discussed in ITEM 5 (for the active set), ITEM 11 (for the expired set), and ITEM 12 (for moving processes from active to expired) of this documentation.

Every time a process yields control back to the `scheduler`, we check each level in the active set for the next `RUNNABLE` process. If no such process is found, then we must swap the active and expired sets (meaning: set active to expired, and expired to active). In our case, what we're essentially swapping are the global `ptable` attributes `active` and `expired` of type `struct level_queue *`. As explained before, `ptable.active` and `ptable.expired` point to the base address of the currently active set and the currently expired set respectively. Sets are implemented as twin arrays of `struct level_queue` elements which are stored in the `ptable` attribute `struct level_queue level[2][RSDL_LEVELS]`, hence the need for `ptable.active` and `ptable.expired` to be of type `struct level_queue *`.

With these in mind, to swap the active and expired sets when no `RUNNABLE` process is found, we simply swap `ptable.active` and `ptable.expired` with the help of a temporary variable. This is implemented as follows

```
// proc.c: 717-721, in scheduler()
for (k = 0; k < RSDL_LEVELS; ++k) {
    q = &ptable.active[k];
    if (q->ticks_left <= 0)
        continue;

    acquire(&q->lock);
    for (i = 0; i < q->numproc; ++i ) {
        p = q->proc[i];
        if(p->state == RUNNABLE && p->ticks_left > 0) {
          found = 1;
          break;
        }
      }
    release(&q->lock)
    if (found)
        break;
}
...
if (found) {
    ...
} else {
    // No RUNNABLE proc found; Can happen before initcode runs, all procs sleeping but will return after n ms,
    // Since there are no procs ready in active set, we swap sets
    nq = ptable.active;
    ptable.active = ptable.expired;
    ptable.expired = nq;
    ...
```

This is considered a swap because as explained previously:

- Whenever we look for `RUNNABLE` processes in the active set levels, we use `ptable.active`, implemented as `q = &ptable.active[k];` in the snippet above.
- Once a `RUNNABLE` process is found, the active set level wherein it was found is pointed to by `q`, which is where we operate for the case of found processes and “yielding” processes with respect to enqueuing and unqueuing them.

Hence, whichever of the twin sets `ptable.active` is pointing to shall be considered the active set.

Swapping of sets also necessitates that we migrate processes left in the old active set (now the expired set) to the new active set. We move said processes to their respective default priority levels in the expired set. Default priority level is defined as

- `RSDL_STARTING_LEVEL` for processes created using `fork(void)` which wraps `priofork(int default_level)` with `default_level = RSDL_STARTING_LEVEL`, and
- The variable argument `default_level` for processes created using a direct call to `priofork(int default_level)`.

This migration to the new active set is done while preserving the FIFO order of the processes from their respective levels in the old active set. We do this by looping over each old active set level `0 <= k < RSDL_LEVELS` for as long as level `k` still contains a PCB pointer (`while (q->numproc > 0)`) and unqueuing the

PCB pointers from there. We only unqueue at the head of level `k`, and this convenient approach is made possible by the fact that our `unqueue_proc` implementation moves processes up the level queue to fill the vacant slot left by the unqueued process. As the processes are moved, we also also replenish their process-local quanta. And as each level is explored, we also explore their level local quanta. These are shown in the code below:

```
// proc.c: 717-740, in scheduler()
    } else {
      // No RUNNABLE proc found; Can happen before initcode runs, all procs sleeping but will return after n ms
      // Since there are no procs ready in active set, we swap sets
      nq = ptable.active;
      ptable.active = ptable.expired;
      ptable.expired = nq;

      // re-enqueue procs in old active set (expired set) to new active set
      for (k = 0; k < RSDL_LEVELS; ++k) {
        q = &ptable.expired[k];
        q->ticks_left = RSDL_LEVEL_QUANTUM; // replenish level-local quantum
        while (q->numproc > 0) {
          p = q->proc[0];
          // proc will be re-enqueued to new level, replenish quantum
          p->ticks_left = RSDL_PROC_QUANTUM;
          unqueue_proc(p, q);

          // re-enqueue to original level in active set
          // if no available level in active set, enqueue to original level in expired set
          nk = p->default_level;
          nq = find_available_queue(nk, nk);
          enqueue_proc(p, nq);
        }
      }
    }
```

Specifically, while each queue is not empty:

- Before the PCB pointers (treated as the processes themselves) that the current old active set level (`q = &ptable.expired[k];`) possibly contains are migrated, we first replenish their quantum with `q->ticks_left = RSDL_LEVEL_QUANTUM`.
- We then unqueue the process `p` at the front/head of the queue and replenish its `p->ticks_left` to `RSDL_PROC_QUANTUM`. Note that this operation decrements `q->numproc` by 1.
- Technically, we then find the next level in the new active set with remaining level-local quantum and space for procs, starting at `p->default_level` to respect the possibility of prioforked processes. We say technically because that's what our `nq=find_available_queue(nk, nk)` call does for each process being moved (where `nk=p->default_level;`). When that available new active set level is found, that's where we enqueue the process being moved to.
    - But in practice, since there can only be `NPROC == 64` processes in xv6 and since the level-local quantum of the old active set (now expired set) are always replenished every swap, we can safely assume that the respective default level for a process being migrated from the old active set to the new active set will always be available.

- Hence, the use of `find_available_queue(nk, nk)` here is merely a matter of convenience.

- Finally, the previously unqueued process `p` from their previous old active set level is assigned to the current tail of their default level in the new active set using `enqueue_proc(p, nq)`, completing the migration. Note that this operation increments `nq->numproc` by 1.

---

14. Implementation explanation for downgrading of process levels

SEE ITEM 9.

As previously explained in ITEM 8, the `int` variable `k` inside the infinite loop of the scheduler holds the level index in the active set where the currently active process `p` is located until before the next schedulable process is found and set to `RUNNABLE`.

When a process depletes its quantum, it yields control back to the scheduler, which resumes exactly after the `swtch(&(c->scheduler), p->context);` call. If the active set level `q = &ptable.active[k]` currently containing the yielding process `p` still has remaining quantum when `p` yielded (keeping in mind that in this case, `p` has already depleted its quantum), we unqueue `p` from `q` (`unqueue_proc(p,q)`), then re-enqueue it to one of the following:

- An available active set level of lower priority than `q` which is searched for starting from `k+1` until `RSDL_LEVELS-1`. *Available* here means “having level-local quantum left” and “having room for another process” (the latter not really a big deal).
- The default priority level of the process in the expired set given by `p->default_level`. The use of `fork` and the direct use of `priofork` play a big role in this.

For this, we use `nq = find_available_queue(nk, p->default_level);` to determine the new level queue `nq` that the unqueued `p` will be enqueued to. `nk` in this “yielding process used up quantum” case is simply set to `k + 1` (re-enqueue to the nearest active set lower priority level, if none, enqueue to the process’ expired set default level, as explained later on), in contrast to the “yielding process still have remaining quantum” that we tackled in ITEM 8 wherein `nk = k` (re-enqueue to the same level). This corresponds to the following code:

```
// proc.c: 650 in scheduler()
swtch(&(c->scheduler), p->context);
switchkvm();

// proc has given up control to scheduler
if (q->ticks_left <= 0) {
    // level-local quantum depleted, migrate all procs
    ...
// proc.c: 680 in scheduler()
} else {
    ...
    if (p->ticks_left <= 0) {
        // proc used up quantum: enqueue to lower priority
        // We're referring to this mainly
        p->ticks_left = RSDL_PROC_QUANTUM;
        nk = k + 1;
```

```
    } else {
        // proc yielded with remaining quantum: re-enqueue to same level
        nk = k;
    }
```

As we may recall from ITEM 12, `find_available_queue` is prototyped as `find_available_queue(int active_start, int expired_start)`, thus, `nq = find_available_queue(nk=k+1, p->default_level)` means that the routine will look for the next available active set level of lower priority than `k` (starting from `k + 1`), and if no such active set level is found, the process will be enqueued to its `default_level` in the expired set which is always available for it as discussed in ITEM 13.

- **Caveat on the Expired Set:** Note that the “finding of available queue” in the expired set is merely a technical remnant from a previous interpretation of required behavior for the expired set.
- Also note that yielding processes being “downgraded” to the expired set also have their process-local quantum replenished.

This is demonstrated in the following code (note the parts highlighted in yellow):

```
// proc.c: 694-709
      // only try to re-enqueue proc if it was not removed before
       // e.g. when it calls exit() (state == ZOMBIE), it removes itself so no need to re-enqueue
       if (q->numproc > 0 && p->state != ZOMBIE) {
         prev_idx = unqueue_proc(p, q);
         if (prev_idx == -1) {
           panic("re-enqueue of proc failed");
         }
         // find vacant queue, starting from level nk as decided above
         // if no available level in active set, enqueue to original level in expired set
         nq = find_available_queue(nk, p->default_level);
         if (is_expired_set(nq)) {
           // proc quantum refresh case 2: proc moved to expired set
           p->ticks_left = RSDL_PROC_QUANTUM;
         }
         enqueue_proc(p, nq);
       }
     }
```

As discussed in ITEM 7, processes calling `exit()` immediately remove themselves from the their own level/queue. Since execution control just returned to the `scheduler()`, this can mean that the process `p` called `exit()` and removed itself from the queue, which we can determine by checking if `p->state == ZOMBIE`. Thus, we only unqueue and re-enqueue a process if its `p->state != ZOMBIE`.

Additionally as will be elaborated in **ITEM 15**, the depletion of a level’s quantum results to the migration of the processes it contains to the next available lower priority level in the active set, or in their respective default levels in the expired set. Again, these cases are responsible for the downgrading of process levels, as can be seen by the use of `find_available_queue(k+1, np->default_level)` and `find_available_queue(k+1, p->default_level)`.

```
// proc.c: 669 & 677 in scheduler()
if (q->ticks_left <= 0) {
      // level-local quantum depleted, migrate all procs
```

```
    while (q->numproc > 0) {
      np = q->proc[0];
      // moving to next level OR expired set, replenish quantum
      np->ticks_left = RSDL_PROC_QUANTUM;

      unqueue_proc(np, q);
      // Section 2.4: The active process should be enqueued last
      if (np == p) {
        continue;
      }

      // move proc to next available level in active set
      // if none, enqueue to original level in expired set
      nq = find_available_queue(k+1, np->default_level);
      // re-enqueue to same level but in active set, or below
      enqueue_proc(np, nq);
    }

    // If proc called exit, it already unqueued itself; no need to re-enqueue
    if (p->state != ZOMBIE) {
        // active process is the last process to be enqueued
        nq = find_available_queue(k+1, p->default_level);
        enqueue_proc(p, nq);
   }
```

As one may have noticed, “downgrading of process levels” has been interpreted to apply to three cases:

1. Process depletes its quantum and gets re-enqueued to the next available lower priority level in the active set.
2. Process depletes its quantum and doesn’t find an available active set level lower than its current level, so it gets re-enqueued to the expired set.
3. Level depletes its quantum, so the processes it contains are downgraded to the next available lower priority level in the active set. If none is found, the processes are all downgraded to their respective default levels in the expired set.

---

15. Implementation explanation for level-local quantum consumption

Recall the `int ticks_left` attribute of `struct level_queue` as described in ITEM 3. We want this attribute of each `struct level_queue` in the active set (specifically the `level_queue` of the currently `RUNNING` process) to be accessible from within **trap.c** the same way `myproc()→ticks_left` is accessible in there for the two values (process-local quantum and level-local quantum) to be simultaneously decremented every tick.

Then, note the `struct level_queue *queue` attribute we have added to `struct cpu` in proc.h as described earlier in ITEM 3.

```
// proc.h: 14-24
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
```

```
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
  struct level_queue *queue;   // level queue where proc can be found
};

extern struct cpu cpus[NCPU];
extern int ncpu;
```

We have chosen this approach owing to the convenience of `cpu` (array of NCPU `struct cpu`) already being accessible in trap.c as `mycpu()` through `#include proc.h` and `#include defs.h`, and `struct cpu *c = mycpu()` already being accessible in the scheduler (even in the vanilla implementation). Hence, if we want to access the active set level queue that the currently `RUNNING` process belongs to, we use `mycpu()->queue->ticks_left`, piggy-backing on the new `queue` attribute of `cpu`.

To be more precise, in `scheduler()`, before switching control to process (`swtch(&(c->scheduler), p->context);` in proc.c: 650), we remember the level queue `q` (recall `q = &ptable.active[k]` in proc.c: 616) it is currently in by storing it in the `cpu` (`c->queue = q` in proc.c: 642). Through this we can directly access the `ticks_left` member of the level queue which represents its level-local quantum from `trap.c`. Note that after the process finished running, we no longer need to remember the queue the process is stored in which is why we set `c->queue = NULL`.

These are demonstrated in the code below:

```
// proc.c: 642-715 in scheduler()
    if(found) {
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      c->queue = q; // REMEMBER which queue p is in!
      switchuvm(p);
      p->state = RUNNING;

      ...

      swtch(&(c->scheduler), p->context);
      switchkvm();

      ...
      c->queue = NULL; // forget which queue p is in
```

After decrementing the process-local quantum (`int proc_ticks`) for each tick through `--myproc()->ticks_left;` as seen in ITEM 8, we also do the same *quantum consumption* for the level-local quantum (`level_ticks`) with `--mycpu()->queue->ticks_left;`. Similar to depleting the process-local quantum, when the level-local quantum is depleted, we also `yield()` control back to the scheduler, hence the condition `proc_ticks <= 0 || level_ticks <= 0`. Use of `<=` instead of `==` merely acts as sanity check against "negative ticks" (not encountered though). This portion is demonstrated in the trap.c code below:

```
// trap.c: 109-117 in trap()
// hardware-generated timer interrupt
```

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER){
  // level queue must be known
  // NOTE: struct cpu cpus[NCPUS] is a global variable, so all cpus[i].queue members are initialized to NULL
  if (!mycpu()->queue)
    panic("Running process located outside active/expired set.");

  int proc_ticks = --myproc()->ticks_left;
  int level_ticks = --mycpu()->queue->ticks_left;
  if (proc_ticks <= 0 || level_ticks <= 0){
    yield();
  }
}
```

After control of execution goes back to `scheduler()`, we reset `c->queue` to NULL in preparation for choosing the next process to run in the next iteration of the scheduler's infinite loop, as shown below:

```
// proc.c: 715 in scheduler()
if (found) {
      ...
      swtch(&(c->scheduler), p->context);
      switchkvm();
      ...

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
      c->queue = NULL;
```

Now, when level-local quantum is fully consumed, all processes still in the depleted level are unqueued and enqueued into the next available lower priority level in the active set (i.e. level that still has quantum remaining), and if there's no available lower priority level in the active set, the processes will have to be enqueued in the expired set.

The procedure for this is as follows:

- While the depleted level `q` still has processes in it (`q->numproc > 0`): For each process `np` in the level queue, we unqueue the process from the head of the level queue. Note that this operation decrements `q->numproc` by 1 and moves the remaining processes there up the queue (there's never an empty slot between the "head" and "tail" of a queue as discussed in ITEM 7). This is highly similar to what we did for the process migration during set swapping, but localized in a level.
- If it is the process `p` which just ran and returned control of execution to the `scheduler()` (i.e. `np == p`), we defer enqueueing it until the other processes have already been enqueued so it is the last process enqueued. Otherwise, we call `nq = find_available_queue(k+1, np->default_level)` to determine which level `nq` in the active set or possibly the expired set they properly belong to next.
- We then immediately call `enqueue_proc(np, nq)` to re-enqueue the process `np` to `nq`. One there's no more processes left in `q`, this process migration is complete.

Note that this process migration is accompanied with replenishment of process-local quantum, as required. These are implemented in the code below:

```
// proc.c: 669 - 677
if (q->ticks_left <= 0) {
      // level-local quantum depleted, migrate all procs
      while (q->numproc > 0) {
        np = q->proc[0];
        // moving to next level OR expired set, replenish quantum
        np->ticks_left = RSDL_PROC_QUANTUM;

        unqueue_proc(np, q);
        // Section 2.4: The active process should be enqueued last
        if (np == p) {
          continue;
        }

        // move proc to next available level in active set
        // if none, enqueue to original level in expired set
        nq = find_available_queue(k+1, np->default_level);
        // re-enqueue to same level but in active set, or below
        enqueue_proc(np, nq);
      }

      // If proc called exit, it already unqueued itself; no need to re-enqueue
      if (p->state != ZOMBIE) {
          // active process is the last process to be enqueued
          nq = find_available_queue(k+1, p->default_level);
          enqueue_proc(p, nq);
     }
```

When all other processes in the level have been re-enqueued, we proceed with re-enqueueing the previously active process `p`. Note that if `p->state == ZOMBIE` after it returns control to `scheduler()`, we know that the process called `exit()` and already removed itself from the level queues. Hence, we avoid re-enqueuing such processes by having the `if (p->state != ZOMBIE)` condition when re-enqueueing proc `p.` This is shown in the code snippet above, highlighted in blue.

Interestingly, we have discussed **three variants of process migration** (into a lower active set level or their default priority level in the expired set):

1. Process-local quantum depletion,
2. Level-local quantum depletion, and
3. Set swapping.

---

16. Implementation explanation for `priofork()`

First, we add an `int default_level` member to the PCB definition `struct proc` in `proc.h`.

```
// proc.h: 66
// Per-process state
struct proc {
  ...
  int ticks_left;              // Remaining process quantum (in ticks)
  int default_level;           // starting level for initial run and during swapping of sets
};
```

This attribute will contain the **per-process default level** as provided by `priofork(int default_level)` . Yes, just `priofork`. But what about `fork`? As you'll see later, `fork` now basically acts as a wrapper function that calls `priofork(int default_level)` with `default_level = RSDL_STARTING_LEVEL`. It's direct calls to `priofork` that **truly** prioforks a process.

Next, when a process is allocated an `UNUSED` PCB in the process table via `allocproc()`, we set its `default_level` member to `RSDL_STARTING_LEVEL`. As we'll find out later, this `default_level` is usually overwritten with the **correct** `default_level` at the tail-end of the proc.c `priofork()` code. We just maintain the setting of process `default_level` to `RSDL_STARTING_LEVEL` in `allocproc()` to respect the needs of `userinit()`. This is now shown below:

```
// proc.c: 336 in allocproc()
static struct proc*
allocproc(void)
{
  ...
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->ticks_left = RSDL_PROC_QUANTUM;
  p->default_level = RSDL_STARTING_LEVEL;
```

As we can see in the snippets below, `p->default_level` is used by `find_available_queue()` as the starting level in both the active set and the expired set (see `find_available_queue()` discussion in ITEM 12, and "expired set" caveat in ITEM 14) for which to begin look for an available level to enqueue to…

- When first enqueueing init via `userinit()`.

  ```
  // proc.c: 399 in userinit()
  void
  userinit(void)
  {
    ...
    p = allocproc();  // allocproc sets p->default_level of init
                      // to RSDL_STARTING_LEVEL
    ...
    p->state = RUNNABLE;
    // only enqueue here since we are sure that allocation is successful
    struct level_queue *q = find_available_queue(p->default_level, p->default_level);
    enqueue_proc(p, q);

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

- Process creation: When calling the refactored `fork()` who calls `priofork()`, and when directly calling `priofork()` (explained in more detail in a short while).
- When moving a process from the old active set (now expired) to to the new active set during set swapping, wherein `find_available_queue` is done for each process being migrated since they may have varying `p->default_level`:

```
// proc.c: 736, in scheduler()
      // re-enqueue procs in old active set (expired set) to new active set
      for (k = 0; k < RSDL_LEVELS; ++k) {
          q = &ptable.expired[k];
          q->ticks_left = RSDL_LEVEL_QUANTUM; // replenish level-local quantum
          while (q->numproc > 0) {
            ...
            // re-enqueue to original level in active set
            // if no available level in active set, enqueue to original level in expired set
            nk = p->default_level;
            nq = find_available_queue(nk, nk);
            enqueue_proc(p, nq);
          }
        }
```

- Or when re-enqueuing a process to the expired set in case there are no available levels in the active set. This time, only the "expired set starting" argument in `find_available_queue` is set to `p->default_level` (for re-enqueuing of curproc, preserving "last to be enqueued" property) or `np->default_level` (when re-enqueueing other processes than curproc) because transfer of a process from active set to the expired set can be seen as another "initial enqueue" to a set, as required. Meanwhile, the "active set starting" argument depends on discussions featured in Items 8, 12, 13, 14, 15).

```
// proc.c: 669, 677, 703 in scheduler()
    if (q->ticks_left <= 0) {
        // level-local quantum depleted, migrate all procs
        while (q->numproc > 0) {
          np = q->proc[0];
          // moving to next level OR expired set, replenish quantum
          np->ticks_left = RSDL_PROC_QUANTUM;

          unqueue_proc(np, q);
          // Section 2.4: The active process should be enqueued last
          if (np == p) {
            continue;
          }

          // move proc to next available level in active set
          // if none, enqueue to original level in expired set
          nq = find_available_queue(k+1, np->default_level);
          // re-enqueue to same level but in active set, or below
          enqueue_proc(np, nq);
        }

        // If proc called exit, it already unqueued itself; no need to re-enqueue
        if (p->state != ZOMBIE) {
          // active process is the last process to be enqueued
          nq = find_available_queue(k+1, p->default_level);
          enqueue_proc(p, nq);
        }
    } else {
        ...
        // only try to re-enqueue proc if it was not removed before
        // e.g. when it calls exit() (state == ZOMBIE), it removes itself so no need to re-enqueue
        if (q->numproc > 0 && p->state != ZOMBIE) {
          prev_idx = unqueue_proc(p, q);
          if (prev_idx == -1) {
```

```
            panic("re-enqueue of proc failed");
          }
          // find vacant queue, starting from level nk as decided above
          // if no available level in active set, enqueue to original level in expired set
          nq = find_available_queue(nk, p->default_level);
          if (is_expired_set(nq)) {
              // proc quantum refresh case 2: proc moved to expired set
              p->ticks_left = RSDL_PROC_QUANTUM;
            }
            enqueue_proc(p, nq);
          }
```

These are the cases that necessitate consideration of `p->default_level` on re-enqueue. And as shown, the modifications applied above now makes the system compatible with the following priofork implementation.

Now, the only thing `int priofork(int default_level)` needs to do is to simply set a newly-initialized process' `p->default_level` to level `default_level` (the level index passed as the `priofork` argument).

For that, we refactor the original `fork()` implementation to `priofork()`. The main modification to the refactored fork being that we now set `np->default_level = default_level`. If the `default_level` is outside of bounds (`default_level < 0` or `default_level >= RSDL_LEVELS`), we indicate an error by returning a `-1`. We also now initially enqueue a process in the `default_level` when forking succeeds.

```
// proc.c: 429-486
// Accessible as either fork(void) or priofork(int) syscalls
int
priofork(int default_level)
{
  ...
  // default_level too large
  if (default_level >= RSDL_LEVELS) {
    return -1;
  }

  // default_level negative
  if (default_level < 0) {
    return -1;
  }
  ...
  np->default_level = default_level;  // set priority level
  np->state = RUNNABLE;
   // only enqueue here since we are sure that allocation is successful
  // the following find_available_queue is very important, observe how
  // for the initial enqueue it takes np->default_level for both of its parameters
  struct level_queue *q = find_available_queue(np->default_level, np->default_level);
  enqueue_proc(np, q);

  release(&ptable.lock);

  return pid;
}
```

To avoid code duplication, the original `fork()` implementation now calls `priofork()` with the `RSDL_STARTING_LEVEL` as argument, which is the original default priority level for processes. This makes `fork()` and `priofork()` work together in a simple manner.

```
// proc.c: 488-493
// original fork() call
int
fork(void)
{
  return priofork(RSDL_STARTING_LEVEL);
}
```

Auxiliary requirements for the priofork() system call 25

Then to make `priofork()` a usable syscall, we add it to `syscall.h` with syscall number `25`.

```
// syscall.h: 25
#define SYS_schedlog  24
#define SYS_priofork  25
```

Then, in `usys.S`, we use the `SYSCALL()` macro to generate the .asm interrupt handler code for the `priofork()` syscall.

```
// usys.S: 35
SYSCALL(schedlog)
SYSCALL(priofork)
```

In syscall.c, we declare the `sys_priofork()` syscall entry point, which we associate with the syscall number `SYS_priofork == 25` in the `syscalls` table:

```
// sycall.c: 109
extern int sys_schedlog(void);
extern int sys_priofork(void);
```

```
// sycall.c: 136
static int (*syscalls[])(void) = {
...
[SYS_schedlog] sys_schedlog,
[SYS_priofork] sys_priofork,
};
```

This entry point/syscall handler is implemented in `sysproc.c`, with code similar to `sys_kill()` which also takes an integer argument via `argint`.

```
// sysproc.c: 16-23
int
sys_priofork(void)
{
  int default_level;
  if(argint(0, &default_level) < 0)
    return -1;
```

```
  return priofork(default_level);
}
```

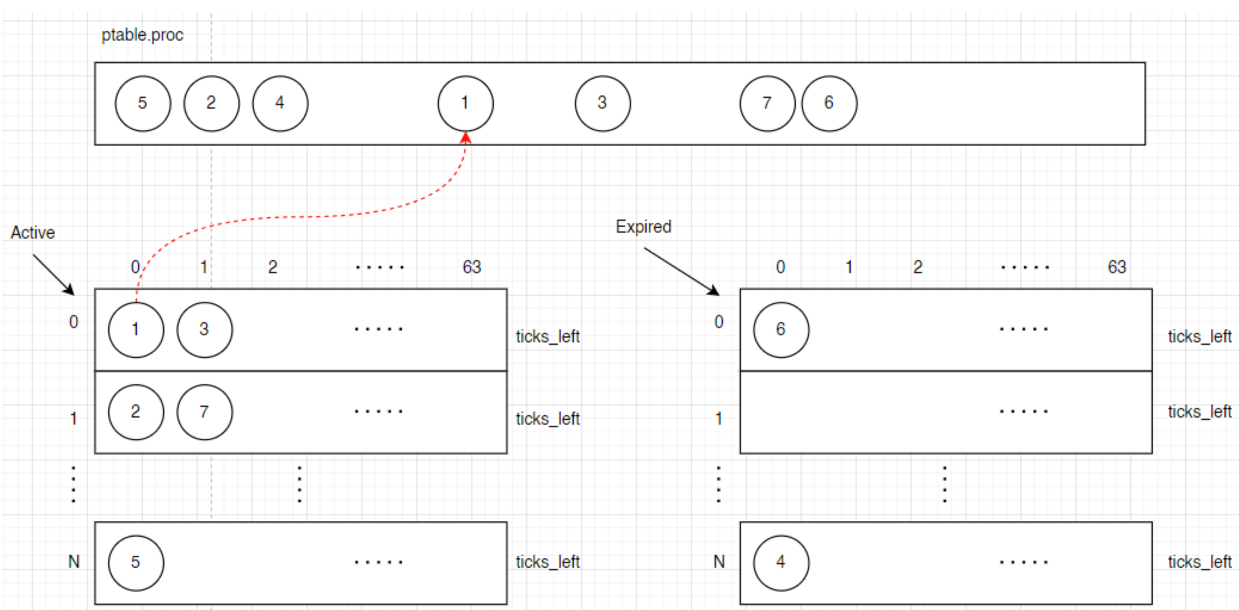
We then make this syscall accessible to other parts of the OS by including its function prototype in `defs.h`.

```
// defs.h: 110
int             fork(void);
int             priofork(int);
```

Finally, we make this syscall accessible to user programs by including its function prototype in `user.h`.

```
// user.h: 30
// system calls
...
int schedlog(int);
int priofork(int);
```

Appendix A: High-level view of an RSDL implementation. Process PCBs are still stored in `ptable.proc`. The "process" entries stored in the level queues in the `ptable.active` and the `ptable.expired` sets are essentially just pointers to their corresponding PCBs in `ptable.proc`, and it is on these sets that RSDL selects for processes to schedule.



END