CS 140

Project 1

The Rotating Staircase Deadline Scheduler

Michael Sean Brian Omisol
Justin Jose Ruaya
Aaron Jude Tanael

Department of Computer Science
College of Engineering
University of the Philippines - Diliman

A.Y. 2022-2023, S1

Table of contents

1-4: Phases, References, Global variables, PCB

1. **Phases.** All branches (*phase1,phase2,phase3,phase4,phase5*) are eligible for checking. **phase5** is the highest phase this team was able to produce.
2. **References.**

[1] E.P. Quiwa. *Data structures*. Electronics Hobbyists Publishing House, 2007. URL: https://books.google.com.ph/books?id=LZCcswEACAAJ.

[2] W.M. Tan, A.A. Zabala, and J.M. Coronel. *Laboratory report 5: Process Scheduling*. 1st semester, A.Y. 2022-2023. CS 140: Operating Systems, 2022, pp. 1–4.

Note that Quiwa's book was used for implementing the process queue. More on that later. Laboratory 5 is used as basis for the `schedlog` system call.

3. **Global Variables**. We have 5 global variables that the system uses. There are three sets and two variables.
   - The `struct set active` denoting the *active set* (more on this later)
   - The `struct set expired` denoting the *expired set* (more on this later)
   - The `struct set temp` denoting the temporary set for swapping (more on this later)
   - `int schedlog_active = 0;` for the traditional schedlog system call; lab5
   - `int schedlog_lasttick = 0;` for the traditional schedlog systemcall; lab5
4. **Process Control Block**. We modify the PCB from `proc.h` as follows:

Code Block 1: PCB Modification in proc.h

```
// .... code from the original PCB ....
// Phase 5 Modification
int quantum_left;           // Quantum left for the process
int starting_level;         // Original priority level for the process
// ...
```

`quantum_left` is used for denoting how much quantum the process has on a specific level (from Lab 5). In this project, we introduce a new variable `starting_level`, which is used when a process migrates from active to expired set (as a result of depleted process-local quanta), or when the quantum of the last priority level gets depleted. In addition, this is where the `priofork` system call stores the custom starting value. Instead of the `RSDL_STARTING_LEVEL`, we use its argument when the systemcall is invoked.

5: Active set, levels, process queue

5 We actually use the same code for both active and expired set.

Code Block 2: Code for `struct set`

```
1  // Struct set for ACTIVE and EXPIRED sets
2  struct set
3  {
4    char name[16];
5    struct pq pq[RSDL_LEVELS];
6  };
```

In code block 2, it just houses the name of the set (in our case, "active"), which is invoked by `InitSet(&active, "active")` (in `userinit`) with its definition in code block 3:

Code Block 3: Implementation of `InitSet()`

```
1  // Initialize the active and expired sets using this function
2  void InitSet(struct set * set, char * name){
3    safestrcpy(set->name, name, sizeof(set->name));
4    for(int l = 0; l < RSDL_LEVELS; l++) {
5      InitQueue(&set->pq[l]);
6    }
7  }
```

Note that `userinit` is the first process. This is where we initialize the sets are initialized and this gets enqueued first in the queue.

Code block 3 sets the name of the set and initializes every single queue from level 0 to `RSDL_LEVELS-1`.

The other part of the set is the `struct pq pq[RSDL_LEVELS]`, which is just an array of `RSDL_LEVELS` process queues. This handles the **levels** of the set, with each level containing at most `NPROC = 64` processes. The definition is given on the next page.

5: Active set, levels, process queue

The process queue came from Quiwa[1]'s EASY definition of a circular queue, with slight modifications to accommodate processes. In our case, the circular queue is used to avoid re-indexing once all elements were populated until the end of the array. This makes computation much faster.

Code Block 4: Implementation of process queue

```
1  // Process queue struct
2  struct pq
3  {
4    int front;
5    int rear;
6    int quantum_left;
7    struct proc * proc[NPROC];
8  };
```

The array contains `NPROC` processes. The queue has a `front` and `rear` that dictates the bounds of the queue. It also has `quantum_left` for its level quantum. Since this is circular, we also have to define a remainder function.

Code Block 5: Getting the remainder of a modulo operation

```
1  int mod(int a, int b)
2  {
3      int r = a % b;
4      return r < 0 ? r + b : r;
5  }
```

The code below initializes the circular queue, as called in Code Block 3.

Code Block 6: process queue initialization

```
1  void InitQueue(struct pq *Q){
2    Q->front = 0;
3    Q->rear = 0;
4    Q->quantum_left = RSDL_LEVEL_QUANTUM;
5  }
```

This just sets the default values. Other methods will be introduced when necessary.

5: Active set, levels, process queue

We also define the checkers if the process is empty or full:

Code Block 7: IsEmptyQueue() and IsFullQueue() implementation

```
1   // Check if queue is empty
2   int IsEmptyQueue(struct pq *Q)
3   {
4     // cprintf("IE: %d\n", Q->front == Q->rear);
5     return(Q->front == Q->rear);
6   }
7
8   // Check if queue is full
9   int IsFullQueue(struct pq *Q)
10  {
11    // cprintf("IE: %d\n", Q->front == Q->rear);
12    return(Q->front == mod(Q->rear + 1, NPROC));
13  }
14
```

6: Initial enqueuing of new processes

6 Similar to previous items, enqueueing a new process comes from Quiwa's EASY definition implementation. The following code shows how it is done: by adding the incoming process to the rear of the queue. The if-else statement covers a how a process replenishes its quantum, which will be discussed at a later section.

Code Block 8: Enqueueing incoming process

```
1  void ENQUEUE(struct pq *Q, struct proc * x, int quantum)
2  {
3    if (IsFullQueue(Q)) panic("queue overflow"); // Do not enqueue process beyond NPROC!
4    x->quantum_left = quantum; // default quantum level
5    Q->rear = mod((Q->rear + 1), NPROC); // updates rear index
6    Q->proc[Q->rear] = x;
7  }
```

To add incoming processes, we call the function ENQUEUE inside the lock environment in priofork(). More on the logic of priofork() later, but to spoil we refactored the fork() logic into priofork() and then fork just calls priofork with the default starting level. It adds the new process to the active set, with its corresponding starting level (RSDL_STARTING_LEVEL) and the process itself (np).

Code Block 9: Addition to the locked environment in fork logic in proc.c

```
1  acquire(&ptable.lock); // vanilla xv6 code
2
3  np->state = RUNNABLE;  // vanilla xv6 code
4  np->starting_level = n; // For fork() this is just RSDL_STARTING_LEVEL
5  ENQUEUE(&active.pq[NEWLEVEL(np->starting_level)], np, RSDL_PROC_QUANTUM);
6
7  release(&ptable.lock);
```

6: Initial enqueuing of new processes

We also enqueue `userinit` since it is the very first process to run. This is enqueued in the default `RSDL_STARTING_LEVEL`.

Code Block 10: Enqueueing the init process (userinit edit)

```
1   acquire(&ptable.lock); // vanilla xv6 code
2
3   p->state = RUNNABLE; // vanilla xv6 code
4   p->starting_level = RSDL_STARTING_LEVEL;
5   ENQUEUE(&active.pq[NEWLEVEL(p->starting_level)], p, RSDL_PROC_QUANTUM); // level quanta might deplete right
    ↪ before enqueue
6
7   release(&ptable.lock);  // vanilla xv6 code
```

For processes waking up, we also re-enqueue it, taking into account the **quantum_left** (not replenishing):

Code Block 11: Modification of sleep() system call

```
1   p->state = SLEEPING; // vanilla xv6 code
2
3   // Re-enqueue the process to sleep
4   int level = GETLEVEL(p);
5   REMOVE(&active.pq[level], p);
6   ENQUEUE(&active.pq[NEWLEVEL(level)], p, p->quantum_left);
7
8   sched(); // vanilla xv6 code
```

7: Dequeueing exiting processes.

7 exit() handles any exiting processes, thus they are dequeued via the REMOVE function, also an implementation from QUIWA's easy definition.

Code Block 12: Removing a process from a queue

```
1   void REMOVE(struct pq *Q, struct proc * x)
2   {
3     int k = mod(Q->front + 1, NPROC); // initializes an integer k to the value of the front of the
    ↪  queue
4     // finds the process to be removed from the given queue
5     while(k != mod(Q->rear + 1, NPROC)){
6       if(Q->proc[k]->pid == x->pid) break;
7       k = mod((k + 1), NPROC);
8     }
9     // pid found - REMOVE asserts that the element x exists in the queue
10    // afterwards, shift the positions of the rest of the elements of the queue by 1
11    while(k != mod(Q->rear + 1, NPROC)){
12      Q->proc[k] = Q->proc[mod((k + 1), NPROC)];
13      k = mod((k + 1), NPROC);
14    }
15    // moves the rear since we removed an element in the queue
16    Q->rear = mod((Q->rear-1), NPROC);
17  }
```

Code Block 13: Edit of `exit()`

```
1     int level = GETLEVEL(curproc);
2     REMOVE(&active.pq[level], curproc);
3
4     // Jump into the scheduler, never to return.
5     curproc->state = ZOMBIE; // vanilla xv6 code
```

Code Block 14: Gets the level of a process p

```
1   // Returns the level of the process in the active set.
2   int GETLEVEL(struct proc *p){
3     for(int l = 0; l < RSDL_LEVELS; l++) { // iterates from 0 to RSDL_LEVELS-1
4       if(IsEmptyQueue(&active.pq[l])){ // ignore empty queues
5         continue;
6       }
7       int k = mod(active.pq[l].front + 1, NPROC); // iterate from front to rear UNTIL we have
8       while(k != mod(active.pq[l].rear + 1, NPROC)){ // found the process
9         if(active.pq[l].proc[k]->pid == p->pid) return l;
10        k = mod((k + 1), NPROC);
11      }
12    }
13    return -1; // the process should be in the other set
14  }
```

8-9: Process-local quantum consumption and replenishment

8 **Process-local quantum consumption**. The process-level quanta works similar to Lab 5's modifications. Take note that the modified PCB (code block 1) has `quantum_left` (see line 4 of code block 13). The consumption of quantum happens in trap.c.

Code Block 15: Modification of `trap.c` to accommodate process-level quantum consumption

```
1     if(myproc() && myproc()->state == RUNNING &&
2        tf->trapno == T_IRQ0+IRQ_TIMER){
3
4       int proc_q = DEC_PQ(); // decrease process quanta
5       int level_q = DEC_LQ(); // decrease level quanta -- must be simultaneous with DEC_PQ
6
7       // if level quantum is 0, empty the level and enqueue to the next available level with quanta
8       // if no level with quanta is available, enqueue to the expired set based on starting level
9       // this can be simultaneous with process quantum being 0
10      if(level_q == 0){
11        ALTERLEVEL();
12        yield();
13      }
14      // if process quantum is 0, enqueue the process to the next priority level
15      // if there is no available next priority level, enqueue to the expired set based on starting
    ↪   level
16      else if(proc_q == 0){ // <---------- FOCUS HERE FOR THIS ITEM
17        ALTERPROC();
18        yield();
19      }
20    }
```

In here, it calls `DEC_PQ()`, which is just

Code Block 16: Definition of `DEC_PQ()` in proc.c

```
1   // Decrement and return the process quantum.
2   int DEC_PQ(void){
3     return --myproc()->quantum_left; // Process-local quanta
4   }
```

It also calls `ALTERPROC()` which is defined on the next page.

8-9: Process-local quantum consumption and replenishment

Code Block 17: Definition of `ALTERPROC()`

```
1   // Dequeues the process from the current level and enqueue to the next available level
2   void ALTERPROC(void){
3     acquire(&ptable.lock);
4     int level = GETLEVEL(myproc());
5     int nextlevel = NEXTLEVEL(level+1); // find the next available level for the dequeued process
6     REMOVE(&active.pq[level], myproc());
7     if(nextlevel < RSDL_LEVELS){
8       ENQUEUE(&active.pq[nextlevel], myproc(), RSDL_PROC_QUANTUM);
9       release(&ptable.lock);
10      return;
11    }
12    ENQUEUE(&expired.pq[myproc()->starting_level], myproc(), RSDL_PROC_QUANTUM);
13    release(&ptable.lock);
14    return;
15  }
```

The other conditions are logic for level-local quantum consumption (ALTERLEVEL) and what happens if the quantum of a process gets depleted. These will be discussed on the next parts.

9 **Process-local quantum replenishment**. This is straightforward, where it is embedded into `void ENQUEUE(struct pq *Q, struct proc * x)` itself:

Code Block 18: Enqueue again, but highlighted the process-local quantum replenishment

```
1   if (IsFullQueue(Q)) panic("queue overflow");
2   x->quantum_left = quantum; // for replenishment, we just call ENQUEUE(queue, process,
    ↪  RSDL_PROC_QUANTUM
3   Q->rear = mod((Q->rear + 1), NPROC);
```

Code block 15 in ALTERPROC() calls ENQUEUE() with `RSDL_PROC_QUANTUM`. Lines 8 and 12 replenishes the process-local quantum. Code block 22 also calls the REMOVE() and ENQUEUE() cases which replenishes the quanta of the processes.

10: schedlog system call

10 The header files and c files are the same from the laboratory 5 schedlog[2] (next page). These are as follows: `user.h`, `usys.S`, `syscall.h` (syscall 24), `sysproc.c`, `proc.h`, and defs.h. The only difference is that we modify the logic for the schedlog found in `void scheduler(void)`

Code Block 19: Schedlog syscall from scheduler()

```
1   // Syscall modification
2   if (schedlog_active) { // schedlog_active
3     if (ticks > schedlog_lasttick) { // ticks > schedlog_lasttick
4       schedlog_active = 0;
5     } else {
6       // Schedlog for active set, but with levels
7       for(int l = 0; l < RSDL_LEVELS; l++) {
8         cprintf("%d|%s|%d(%d)", ticks, active.name, l, active.pq[l].quantum_left); //
    ↪  <tick>|<set>|<level>(<quantum left>) for phase 4
9
10        k = mod(active.pq[l].front+1, NPROC);
11        while (k != mod((active.pq[l].rear + 1), NPROC)) {
12          pp = active.pq[l].proc[k];
13          cprintf(",[%d]%s:%d(%d)", pp->pid, pp->name, pp->state, pp->quantum_left); //
    ↪  ,[<PID>]<process name>:<state number>(<quantum left>) for phase 4
14          k = mod((k + 1), NPROC);
15        }
16        cprintf("\n");
17      }
18      // Schedlog for expired set, but with levels
19      for(int l = 0; l < RSDL_LEVELS; l++) {
20        cprintf("%d|%s|%d(%d)", ticks, expired.name, l, expired.pq[l].quantum_left); //
    ↪  <tick>|<set>|<level>(<quantum left>) for phase 4
21
22        k = mod(expired.pq[l].front+1, NPROC);
23        while (k != mod((expired.pq[l].rear + 1), NPROC)) {
24          pp = expired.pq[l].proc[k];
25          cprintf(",[%d]%s:%d(%d)", pp->pid, pp->name, pp->state, pp->quantum_left); //
    ↪  ,[<PID>]<process name>:<state number>(<quantum left>) for phase 4
26          k = mod((k + 1), NPROC);
27        }
28        cprintf("\n");
29      }
30    }
31  }
```

The scheduler checks every level. If there are no running processes or there is no level-local quantum left (more on this later), then it just moves on to the next level .As seen here, it loops another time for `RSDL_LEVELS` times, where for each level `l`, it gets *k*, which is the index of our circular queue. Remember that the starting index of the queue is **not** at index 0, but rather at `front`. This loops through the queued processes that gets printed according to the format presented in the project specs. For the expired set, it is actually the **same**! The only difference is that we change from `active`→`expired`.

10: schedlog system call

Here is the change before the scheduler() function (laboratory report 5)

Code Block 20: Laboratory report 5 schedlog modifications

```
1  // Schedlog variables
2  int schedlog_active = 0;
3  int schedlog_lasttick = 0;
4
5  void schedlog(int n) {
6    schedlog_active = 1;
7    schedlog_lasttick = ticks + n;
8  }
```

11-12: Expired set, transfer from active→expired

11 **Expired Set**. Honestly, the expired set implementation is just the same as the active set. This is due to the `struct set` that we have defined in code block 2 and 3. But for completeness, we just put its initialization:

Code Block 21: Initialization of the expired set in userinit

```
1     InitSet(&active, "active"); //active set
2     InitSet(&expired, "expired"); // <---- expired set!
3     InitSet(&temp, "temp"); // <-- for swapping
4
5     p = allocproc(); // vanilla xv6 code
```

Line 33 in `proc.c` has `struct set expired;` which is uninitialized at first. This was initialized above.

12 **Transferring process from the active set to the expired set.** `ALTERPROC` dequeues the process from the current level and enqueues it to the next. However, should the process hit the final (RSDL_LEVELS-1) level, it will be dequeued and enqueued to the expired set level where it started from.

It will ignore the `if` condition if it is currently on the final level.

Code Block 22: Corresponding code regarding the downgrading of a process from active to expired set.

```
1   // Dequeues the process from the current level and enqueue to the next available level
2   void ALTERPROC(void){
3     acquire(&ptable.lock);
4     int level = GETLEVEL(myproc());
5     int nextlevel = NEWLEVEL(level+1); // find the next available level for the dequeued process
6     REMOVE(&active.pq[level], myproc());
7     if(nextlevel < RSDL_LEVELS){
8       ... // Since we are putting the process from active -> expired, we ignore the if condition
9     }
10    ENQUEUE(&expired.pq[myproc()->starting_level], myproc(), RSDL_PROC_QUANTUM);
11    release(&ptable.lock);
12    return;
13  }
```

We see that line 10 enqueues it to the expired set, with **replenished quantum**.

13: Swapping of set

13 Swapping sets occurs in the `scheduler()`. A temporary set is used. It goes through three procedures: (1) dequeueing the active set and enqueueing dequeued processes to a temporary set; (2) dequeueing expired set and enqueueing dequeued processes to the active set, and; (3) enqueueing proccesses from the temporary set to the active set, if non-empty.

These three procedures check through all levels and check which queues are non-empty to follow the procedure, using a for-loop for iteration and a while-loop to check for non-empty cases. The code block below shows how it is done. We also defined a set `temp` in code block 20.

Code Block 23: Swapping of sets, via a temporary set

```
1   if(swap){
2     // refresh quanta for active and expired sets
3     for(int l = 0; l < RSDL_LEVELS; l++) {
4       active.pq[l].quantum_left = RSDL_LEVEL_QUANTUM;
5       expired.pq[l].quantum_left = RSDL_LEVEL_QUANTUM;
6     }
7
8     // empty active set first before swap, then enqueue into temp
9     for(int l = 0; l < RSDL_LEVELS; l++) {
10      while(!IsEmptyQueue(&active.pq[l])){
11        DEQUEUE(&active.pq[l], &pp);
12        ENQUEUE(&temp.pq[l], pp, RSDL_PROC_QUANTUM);
13      }
14    }
15    // empty expired set but put it to active
16    for(int l = 0; l < RSDL_LEVELS; l++) {
17      while(!IsEmptyQueue(&expired.pq[l])){
18        DEQUEUE(&expired.pq[l], &pp);
19        ENQUEUE(&active.pq[l], pp, RSDL_PROC_QUANTUM);
20      }
21    }
22    // if temp (old active set) is nonempty, enqueue the elements into active set based on their
      ↪ starting levels
23    for(int l = 0; l < RSDL_LEVELS; l++) {
24      while(!IsEmptyQueue(&temp.pq[l])){
25        DEQUEUE(&temp.pq[l], &pp);
26        ENQUEUE(&active.pq[pp->starting_level], pp, RSDL_PROC_QUANTUM);
27      }
28    }
29  }
30  swap = 1;
```

14: Downgrading of process levels

14 As stated in the documentation, RSDL downgrades processes should two conditions occur: (1) a process consumes its own quantum and; (2) the level where the process runs has ran out of quantum.

For (1), `ALTERPROC()` handles process downgrading, while (2) has `ALTERLEVEL()`, which handles for level quantum cases. We've already defined `ALTERPROC()`, so we define `ALTERLEVEL()` as follows:

Code Block 24: Process and level downgrade

```
1   // Empties the current level and enqueue to the next available level
2   void ALTERLEVEL(void){
3     acquire(&ptable.lock);
4     struct proc * pp; // pointer placeholder
5     int level = GETLEVEL(myproc()); // gets the level of current process
6     int nextlevel = NEWLEVEL(level+1); // demote and find the next available level for the dequeued
    ↪  process
7     REMOVE(&active.pq[level], myproc()); // remove first the active process - this will be the last
    ↪  process to be enqueued
8     if(nextlevel < RSDL_LEVELS){ // this condition holds if there are still available undepleted levels
    ↪  to be enqueued
9       // transfers all processes from the current level to the next enqueueable level in the active set
10      while(!IsEmptyQueue(&active.pq[level])){
11        DEQUEUE(&active.pq[level], &pp);
12        ENQUEUE(&active.pq[nextlevel], pp, RSDL_PROC_QUANTUM);
13      }
14      // enqueue the interrupted process to the tail-end of the queue
15      ENQUEUE(&active.pq[nextlevel], myproc(), RSDL_PROC_QUANTUM);
16      release(&ptable.lock);
17      return;
18    }
19  // else, this runs if there are no remaining enqueueable levels below the current level
20    // transfers all processes from the current level to the expired set based on their starting levels
21    while(!IsEmptyQueue(&active.pq[level])){
22      DEQUEUE(&active.pq[level], &pp);
23      ENQUEUE(&expired.pq[pp->starting_level], pp, RSDL_PROC_QUANTUM);
24    }
25    // enqueue the interrupted process to the tail-end of the queue
26    ENQUEUE(&expired.pq[myproc()->starting_level], myproc(), RSDL_PROC_QUANTUM);
27    release(&ptable.lock);
28    return;
29  }
```

Similarly, this is also called in `trap.c` which is elaborated on the next page.

14: Calls in trap.c

ALTERPROC() takes in the process and obtains its current level, before finding the next available level for it. It is then removed, and will be enqueued back to the active set only if it has not reached the lowest level (RSDL_LEVELS) before level downgrade. Otherwise, it will be enqueued to the expired set, set on the level where it has originally started.
ALTERLEVEL() takes a similar approach: obtain the process' level and the level where it will be enqueued. It will then remove the running process, before following up with the rest of the processes inside the active level to be downgraded. If there is no level below in the active set, all processes will be dequeued and enqueued in the expired set where they started. The running process will then be enqueued soon after.
Below are the changes in trap.c, which is just a repaste from the previous items.

Code Block 25: Changes in trap.c to accomodate downgrade

```
1   // Force process to give up CPU on clock tick.
2   // If interrupts were on while locks held, would need to check nlock.
3   if(myproc() && myproc()->state == RUNNING &&
4      tf->trapno == T_IRQ0+IRQ_TIMER){
5
6     int proc_q = DEC_PQ(); // decrease process quanta
7     int level_q = DEC_LQ(); // decrease level quanta -- must be simultaneous with DEC_PQ
8
9     // if level quantum is 0, empty the level and enqueue to the next available level with quanta
10    // if no level with quanta is available, enqueue to the expired set based on starting level
11    // this can be simultaneous with process quantum being 0
12    if(level_q == 0){
13      ALTERLEVEL();
14      yield();
15    }
16    // if process quantum is 0, enqueue the process to the next priority level
17    // if there is no available next priority level, enqueue to the expired set based on starting level
18    else if(proc_q == 0){
19      ALTERPROC();
20      yield();
21    }
22  }
```

The two following processes are called when the system calls for trap. As the state in running, it will check if the process quantum and the level quantum has decreased to 0 (via two variables proc_q and level_q). ALTERPROC() is called when proc_q is 0, the same for ALTERLEVEL() and level_q.

15: Level-local quantum consumption

15 To decrement the quantum of the level where the process is running, `DEC_LQ` is used. It returns the decreased quantum of the currently active level in the set.

Code Block 26: Function that decreases quantum pertaining to the currently active level.

```
1  // Decrement and return level quantum of a process.
2  // Only happens in the active set.
3  int DEC_LQ(void){
4    return --active.pq[GETLEVEL(myproc())].quantum_left; // Level quanta
5  }
```

It is always called in `trap()`, where in the previous section is also used to verify when to yield a process as seen in Code Block 24.

16: priofork systemcall

16 Priofork is easy to implement. We edited the same system call changes as in laboratory[2] 5. These are as follows: `user.h`, `usys.S`, `syscall.h` (syscall 25), `sysproc.c`, `proc.h`, and defs.h. In addition, we added extra code:

The gist is that instead of `void fork(void)` invoking the forking logic, we moved it to `int priofork(int n)`. This means that the vanilla `fork()` just calls `priofork(RSDL_STARTING_LEVEL)`. In fact, this is the new code for fork:

Code Block 27: Modified fork()

```
1  int
2  fork(void)
3  {
4    return priofork(RSDL_STARTING_LEVEL);
5  }
```

For priofork, we just added a single line:

Code Block 28: `priofork()` implementation

```
1  // Create a new process copying p as the parent.
2  // Sets up stack to return as if from system call.
3  // Caller must set state of returned proc to RUNNABLE.
4  int
5  priofork(int n)
6  {
7    int i, pid;  // vanilla xv6 code
8    struct proc *np;  // vanilla xv6 code
9    struct proc *curproc = myproc(); // vanilla xv6 code
10
11   // Allocate process.
12   // ...
13   // ... redacted vanilla xv6 code ...
14
15   acquire(&ptable.lock);  // vanilla xv6 code
16
17   np->state = RUNNABLE;  // vanilla xv6 code
18   np->starting_level = n;
19   ENQUEUE(&active.pq[NEXTLEVEL(np->starting_level)], np, RSDL_PROC_QUANTUM);
20
21   release(&ptable.lock);  // vanilla xv6 code
22
23   return pid;  // vanilla xv6 code
24 }
```

In here, it just enqueues the `np` process with quantum `RSDL_PROC_QUANTUM`. Line 18 sets the starting level to the syscall parameter `n`.

End of Project 1 Documentation

Thank you! ♡

Figure: </3