CS 140 Project 1: The Rotating Staircase Deadline Scheduler Documentation

Members:
Bugaoisan, Denver Earl Paul S.
Nicolas, Jack Vincent
Ragunton, Carl David B.

Gdrive link for the video documentation:
https://drive.google.com/file/d/1TCENNM3x2FPPSFAOd5CRaKpITJ4kf9gw/view?usp=share_link

—--

1. [1,2,3,4,5] List of all phases with working code (i.e., which branches are eligible for checking)

The branches that our group was able to do are the following:

-phase1
-phase2
-phase3
-phase4
-phase5

2. [1,2,3,4,5] All references used with purpose specified (if any; otherwise, explicitly state that you did not use any external resource)

The only reference that our group used for this project is the “CS 140 Laboratory 5: Process Scheduling” from https://uvle.upd.edu.ph/pluginfile.php/767851/mod_resource/content/4/CS_140_22_1_LAB_5.pdf. The purpose of using this reference is to gain ideas and to replicate how the Round Robin Scheduler works in the lab. That will serve as the base of how the whole project works. The syscall schedlog is also mainly from this activity.

3. [1,2,3,4,5] List of all global variables introduced with purpose specified (only those not related to system calls)

`int num` in `proc.c`

The purpose of this global variable is to assign an ascending number to all processes. Once a process is made, the num increments and it is assigned to the process. The same idea applies when a process is used. This way, we can know which process is more recently enqueued and which ones are there first. This variable ensures the application of FIFO in the scheduler.

```
int level_quantum_active [RSDL_LEVELS];
int level_quantum_expired [RSDL_LEVELS];
```

in proc.c

The purpose of these global arrays is to store the remaining quantum per level. The level_quantum_active is responsible for the levels in the active set, while the level_quantum_expired is responsible for the levels in the expired set.

4. [1,2,3,4,5] List of changes made to the process control block

There are several variables that were added in the PCB:

```
int ticks_left;
int queue_num;
int active;
int level;
int quantum_level;
int reset;
int prio;
```

Variables	**Description**
`int ticks_left;`	The quantum of the process
`int queue_num;`	The queue number in local-level
`int active;`	Whether the process is in the active set (`active = 1`) or expired set (`active = 0`)
`int level;`	The priority level of the process
`int quantum_level;`	This determines whether the quantum for the current level is depleted.
`int reset;`	This dictates whether the quantums per level are set to `RSDL_LEVEL_QUANTUM (reset = 1)`. Otherwise, continue.
`int prio;`	Starting level if `priofork` syscall is called.

5. [1,2,3,4,5] Code representation of the active set and its levels.

First, it is important to take note of the PCB. The highlighted line shows the int variable active. This determines whether the process is in the active set (when active = 1) or not (when active = 0). This is in proc.h.

```c
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int ticks_left;
  int queue_num;
  int active;
  int level;
  int quantum_level;
  int reset;
  int prio;
};
```

This is the code for the active set and its levels. This is located in proc.c, under the scheduler function.

```c
for (int l = 0; l < RSDL_LEVELS; l++){
  cprintf("%d|active|%d(%d)", ticks, l, level_quantum_active[l]);

  struct proc *pp;

  char* initname="init";
  for (int k = 0; k <NPROC; k++) {
    pp = &ptable.proc[sorted[k]];
    if (pp->level != l) continue;
    if (pp->pid == 0 && pp->name != initname) continue;
    if (pp->state == 5 || pp->state == EMBRYO || pp->state == UNUSED) continue;

    if (pp->state == RUNNING && pp->active == 1) cprintf(",[%d]%s:%d(%d)",pp->pid, pp->name, pp->state,pp->ticks_left);
    else if(pp->active == 1) cprintf(",[%d]%s:%d(%d)",pp->pid, pp->name, pp->state,pp->ticks_left);

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

The starting for loop is responsible for the different levels that will be printed out in the active set. This is followed by a cprintf which prints out the tick, level and quantum in that level.

The levels are checked whether the processes are supposed to be printed on the current level or not, and if so, we just skip and loop again. The pid of the processes are also checked to skip when it is 0 and the name is not equal to the initname, which means that they are the killed processes.

The nested for loop is used to iterate through the active processes (when {process} -> active = 1) and print them out for the current level as long as their proc_state is not UNUSED, EMBRYO or ZOMBIE.

```
146|active|0(100),[1]init:2(9),[2]sh:2(10),[3]test:2(9)
146|active|1(100)
146|active|2(100),[4]test:4(10)
146|active|3(100),[5]test:2(9)
146|active|4(100),[6]test:2(9)
```

6. [1,2,3,4,5] Implementation explanation for initial enqueuing of new processes.

Our group did not use literal queues to create a queue. Instead, we use values and conditions to replicate the queue structure.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int ticks_left;
  int queue_num;
```

We added queue_num to the PCB. This will represent when a process "enters the imaginary queue." The higher the queue_num, the more recent a process enters the queue. This ensures the FIFO.

```
int num = 0;//phase1
```

In proc.c, we added a global variable. This will give the queue_num of the processes their values.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->queue_num = num;//phase1
  num += 1;
  p->active = 1;//phase2
  p->ticks_left = RSDL_PROC_QUANTUM;
  p->level = RSDL_STARTING_LEVEL;
```

This is where the initial enqueueing of processes happens. When a process is created, it's queue->num becomes num. Then, num is incremented to let the next process have a higher value. That means the lower the queue_num, the deeper it is in the queue.

For multiple levels, we just set conditions on which level will be used.

7. [1,2,3,4,5] Implementation explanation for dequeuing of exiting processes.

First, we look into the proc.h and find the PCB with a variable queue_num. This represents when a process “enters the imaginary queue.” The higher the queue_num, the more recent a process enters the queue. This ensures the FIFO.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int ticks_left;
  int queue_num;
  int active;
  int level;
  int quantum_level;
  int reset;
  int prio;
};
```

Meanwhile, in proc.c, we have a global variable called num. This will give the queue_num of the processes their values.

```
int num = 0;
```

Whenever, these pairs come up in the code, it dequeues the exiting process. When num is incremented it enables the next process to have a higher value.This is an instantaneous dequeue and enqueue. As the value of queue_num goes up, the process is dequeued from its original position and enqueued to the end of the “queue”.

```
num+=1;
p->queue_num = num;
```

8. [1,2,3,4,5] Implementation explanation for process-local quantum consumption.

First, the process-local quantum is initialized inside the for (;;) in the scheduler function under proc.c. Each process is loaded with the process quantum of `RSDL_PROC_QUANTUM`.

```
if (p->active == 1) p->ticks_left = RSDL_PROC_QUANTUM;/
```

Now, the main code responsible for the quantum consumption in the local level is located in trap.c. In the trap function, when the process is running, the ticks_left is decremented for each tick that it runs. When, the ticks_left == 0 (this means that the process-local quantum is fully depleted), then we yield.

```
if(myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0+IRQ_TIMER){
    --myproc()->quantum_level;
    --myproc()->ticks_left;
  if (myproc()->quantum_level <= 0 || myproc()->ticks_left == 0){
    yield();
  }}
```

9. [1,2,3,4,5] Implementation explanation for process-local quantum replenishment.

First, the process-local quantum is initialized inside the for (;;) in the scheduler function under proc.c. Each process is loaded with the process quantum of `RSDL_PROC_QUANTUM`.

```
if (p->active == 1) p->ticks_left = RSDL_PROC_QUANTUM;/
```

The replenishment of the process-local quantum can be found in proc.c under the scheduler function. It mainly occurs when the ticks_left of the process has been depleted. It is replenished with a value depending on the `RSDL_PROC_QUANTUM`.

```
else if (p->ticks_left == 0 && p->active == 1){
  num+=1;
  p->queue_num = num;
  p->level++;
  p->ticks_left=RSDL_PROC_QUANTUM;
  if (p->level == RSDL_LEVELS) {
    if (p->prio != -1) p->level = p->prio;
    else p->level=RSDL_STARTING_LEVEL;
    p->active=0;
    p->ticks_left=RSDL_PROC_QUANTUM;
  }
}
```

As for replenishment of the process-local quanta in the expired set, they are replenished when they are moved from the active set to the expired set.

```
if (p->level == RSDL_LEVELS) {
  if (p->prio != -1) p->level = p->prio;
  else p->level=RSDL_STARTING_LEVEL;
  p->active=0;
  p->ticks_left=RSDL_PROC_QUANTUM;
}
```

10. [1,2,3,4,5] Implementation explanation for schedlog.

The `schedlog` works in a similar fashion to the `schedlog` in Lab 5. As stated in Lab 5, `schedlog` takes in an integer that corresponds to the length of time (measured in ticks) to output scheduling information. The scheduler was modified to print out the process table before it switches to a new scheduled process if `schedlog` is enabled, or disable `schedlog` if it has reached the specified number of ticks.

```c
int schedlog_active = 0;
int schedlog_lasttick = 0;

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

If the schedlog is active, the process table will be first sorted based on their queue number (queue_num). The PCB of each process has its own queue number. The sorting algorithm is implemented this way:

Queue numbers of each process is stored in the integer array called "array". There is also an integer array called "sorted" that is used to store the indices of the elements in "array" after they are sorted. The outer for loop with the variable "k" iterates through each element in the "array" and "sorted" arrays. The inner for loop with the variable "j" compares the element at index "k" in the "array" to each element with an index greater than "k". If the element at index "k" is greater than the element at index "j", the elements at these indices are swapped in both the "array" and "sorted" arrays. This process is repeated until all of the elements in the array are in the desired order. After the nested for loops complete, the "array" will be sorted in ascending order and the "sorted" array will contain the indices of the elements in the original "array" after they were sorted.

```
int sorted[NPROC];
int array[NPROC];
for (int k = 0; k < NPROC; k++) {
  sorted[k]=k;
  struct proc *pp;
  pp = &ptable.proc[k];
  array[k]=pp->queue_num;
}

for (int k = 0; k < NPROC; k++) {
  for (int j = k+1; j < NPROC; j++) {
    if(array[k] > array[j]) {
      int temp = sorted[k];
      sorted[k]=sorted[j];
      sorted[j]=temp;


      int temp2 = array[k];
      array[k] = array[j];
      array[j] = temp2;
    }
  }
}
```

In order to print the scheduling information:
The priority levels are represented by the integer value of the "level" field of each process. The outer for loop with the variable "l" iterates through each priority level. It prints first the global ticks, set (active or expired), levels and the quantum of each level. The inner for loop with the variable "k" iterates through each process in the priority queue. For each process, the code checks if the process is at the current priority level, and if the process is in a valid state (not EMBRYO, UNUSED, or ZOMBIE). If these conditions are met, another if condition is used to check if the state of the process is running and active = 1. If true, the code prints out information about the process, including its PID, name, state, and the number of ticks remaining. Printing for the expired set works in a similar fashion with active = 0.

```
        for (int l = 0; l < RSDL_LEVELS; l++){
          cprintf("%d|active|%d(%d)", ticks, l, level_quantum_active[l]);

          struct proc *pp;

          char* initname="init";
          for (int k = 0; k <NPROC; k++) {
            pp = &ptable.proc[sorted[k]];
            if (pp->level != l) continue;
            if (pp->pid == 0 && pp->name != initname) continue;
            if (pp->state == 5 || pp->state == EMBRYO || pp->state == UNUSED) continue;

            if (pp->state == RUNNING && pp->active == 1) cprintf(",[%d]%s:%d(%d)",pp->pid,
            pp->name, pp->state,pp->ticks_left);
            else if(pp->active == 1) cprintf(",[%d]%s:%d(%d)",pp->pid, pp->name, pp->state,
            pp->ticks_left);

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

11. [..2,3,4,5] Code representation of the expired set and its levels.

First, it is important to take note of the PCB. The highlighted line shows the int variable active. This determines whether the process is in the expired set (when active = 0) or not (when active = 1). This is in proc.h.

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int ticks_left;
  int queue_num;
  int active;
  int level;
  int quantum_level;
  int reset;
  int prio;
};
```

This is the code for the expired set and its levels. This is located in proc.c, under the scheduler function.

```
for (int l = 0; l < RSDL_LEVELS; l++){
  cprintf("%d|expired|%d(%d)", ticks,l, level_quantum_expired[l]);
  struct proc *pp;

  char* initname="init";
  for (int k = 0; k <NPROC; k++) {
    pp = &ptable.proc[sorted[k]];
    if (pp->level != l) continue;
    if (pp->pid == 0 && pp->name != initname) continue;
    if (pp->state == 5 || pp->state == EMBRYO || pp->state == UNUSED) continue;

    if (pp->state == RUNNING && pp->active == 0)cprintf(",[%d]%s:%d(%d)",pp->pid, pp->name, pp->state,pp->ticks_left);
    else if(pp->active == 0) cprintf(",[%d]%s:%d(%d)",pp->pid, pp->name, pp->state,pp->ticks_left);

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

The starting for loop is responsible for the different levels that will be printed out in the expired set. This is followed by a cprintf which prints out the tick, level and quantum in that level.

The levels are checked whether the processes are supposed to be printed on the current level or not, and if so, we just skip and loop again. The pid of the processes are also checked to skip when it is 0 and the name is not equal to the initname, which means that they are the killed processes.

The nested for loop is used to iterate through the expired processes (when {process}->active = 0) and print them out for the current level as long as their proc_state is not UNUSED, EMBRYO or ZOMBIE.

```
146|expired|0(100)
146|expired|1(100)
146|expired|2(100)
146|expired|3(100)
146|expired|4(100)
```

12. [..2,3,4,5] Implementation explanation for transferring a process from the active to expired

The PCB of each process has `ticks_left` and `active` variables as shown below. `ticks_left` acts as the quantum for the process and `active` acts if the process is in the active set or expired set.

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int ticks_left;       OozoraCielo, 2 weeks ago • temp commit
  int queue_num;
  int active;      OozoraCielo, 2 days ago • temp commit
  int level;
  int quantum_level;
  int reset;
  int prio;
```

If the process is still active (`active = 1`); quantum of the process is depleted (`p->ticks_left = 0`); and it is in the last level of the active set (`p->level=RSDL_LEVELS`), the process will be transferred to the expired set by setting `active=0` as shown in the code below.

```
        if (p->ticks_left == 0 && p->active == 1){
          num+=1;
          p->queue_num = num;
          p->level++;
          p->ticks_left=RSDL_PROC_QUANTUM;
        };
        if (p->level == RSDL_LEVELS) {
            p->level=RSDL_STARTING_LEVEL;
            p->active=0;
            p->ticks_left=RSDL_PROC_QUANTUM;
        }
```

13. [..2,3,4,5] Implementation explanation for swapping of sets.

The scheduler will check if all the runnable processes are still active. As shown in the code below, if the variable `all_expired` is equal to 0, a runnable process is still active (`ppp->active = 1`). If all the processes are not active, `all_expired` is equal to 1.

```
      int all_expired = 1;
      for (int k = 0; k < NPROC; k++) {
        ppp = &ptable.proc[k];
        if (ppp->level < currlevel) continue;

        if (ppp->state != RUNNABLE) continue;
        else if (ppp->active == 1){
          all_expired = 0;
        }
      }
```

If `all_expired` is equal to 1, `reset` will toggle and the processes are arranged in FIFO manner. A loop is also used to set the unrunnable processes to 0, set their quantum to `RSDL_PROC_QUANTUM` and arrange them in FIFO manner.

```
      if (all_expired == 1){
        p->reset = 1;
        num += 1;//phase1F
        p->queue_num = num;//phase1

        for (int k = 0; k < NPROC; k++) {
          ppp = &ptable.proc[k];
          if (ppp->level != currlevel) continue;
          if (ppp->active == 1){
            ppp->active = 0;
            ppp->ticks_left=RSDL_PROC_QUANTUM;
            num += 1;//phase1F
            ppp->queue_num = num;//phase1
          }
        }
```

Lastly, since all processes now are not active, they are set to active again by another for loop.

```
        for (int k = 0; k < NPROC; k++) {
          ppp = &ptable.proc[k];
          ppp->active = 1;
        }
      }
```

14. [....3,4,5] Implementation explanation for downgrading of process levels

The PCB of each process has a variable level.

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int ticks_left;
  int queue_num;
  int active;
  int level;          OozoraCielo, yesterday • temp commit …
  int quantum_level;
  int reset;
  int prio;
};
```

Downgrading the level of each process is based on the `ticks_left` of the certain process; `ticks_left` also serves as the quantum for each process. If `ticks_left` is equal to 0 or the quantum is depleted, the priority level of the process will downgrade through `p->level++` in a FIFO manner and its quantum will also be replenished. If the level of the process is equal to `RSDL_LEVELS`, the process will be sent to the expired set based on `RSDL_STARTING_LEVEL`.

```
      if (p->quantum_level<=0){
        for (int k = 0; k < NPROC; k++){
            struct proc *ppp;
            ppp = &ptable.proc[sorted[k]];
            if (ppp->level != currlevel) continue;
            num+=1;
            ppp->queue_num = num;
            ppp->level += 1;
            ppp->ticks_left=RSDL_PROC_QUANTUM;
            if (ppp->level >= RSDL_LEVELS) {
              if (ppp->prio != -1) ppp->level = ppp->prio;
              else ppp->level=RSDL_STARTING_LEVEL;
              ppp->active=0;
              ppp->ticks_left=RSDL_PROC_QUANTUM;
            }
          }
          num+=1;
          p->queue_num = num;
      }
      else if (p->ticks_left == 0 && p->active == 1){
        num+=1;
        p->queue_num = num;
        p->level++;
        p->ticks_left=RSDL_PROC_QUANTUM;
        if (p->level == RSDL_LEVELS) {
          if (p->prio != -1) p->level = p->prio;
          else p->level=RSDL_STARTING_LEVEL;
          p->active=0;
          p->ticks_left=RSDL_PROC_QUANTUM;
        }
      }       github-classroom[bot], 2 weeks ago • Initial c
```

15. [......4,5] Implementation explanation for level-local quantum consumption

To implement this part, 2 changes are made to the PCB.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int ticks_left;
  int queue_num;
  int active;
  int level;
  int quantum_level;
  int reset;
  int prio;
};
```

They are the quantum_level and reset. quantum_level will let the program know what is the quantum of the current level of the process. reset will signal if the quantums of the levels should be reset.

In order to make the decrement of ticks of the quantum_level similar to the process of the ticks_left, the trap.c has also been changed.

```
103    // Force process to give up CPU on clock tick.
104    // If interrupts were on while locks held, would need to check nlock.
105    if(myproc() && myproc()->state == RUNNING &&
106       tf->trapno == T_IRQ0+IRQ_TIMER){
107       --myproc()->quantum_level;
108       --myproc()->ticks_left;
109      if (myproc()->quantum_level <= 0 || myproc()->ticks_left == 0){
110        yield();
111      }}
```

This causes the quantum level to decrement every tick just like how ticks_left works. Also, on line 109, I included an OR statement inside the if condition. Thus, whenever one of the quantum_level or the ticks_left are 0, the process will be forced to yield the control of the cpu.

On the proc.c, 2 global arrays has been added:

```c
int level_quantum_active [RSDL_LEVELS]; // phase 4
int level_quantum_expired [RSDL_LEVELS];
```

As their name suggests, level_quantum_active will contain all the quantum for active level sets and level_quantum_expired is for the quantum of expired level sets.

```c
for (int i = 0; i<RSDL_LEVELS; i++){
  level_quantum_active[i] = RSDL_LEVEL_QUANTUM;
  level_quantum_expired[i] = RSDL_LEVEL_QUANTUM;
}
for (int k = 0; k < NPROC; k++) {
  struct proc *ppp;
  ppp = &ptable.proc[k];
  if (ppp->active == 1){
    ppp->quantum_level = level_quantum_active[ppp->level];
  }
  else{
    ppp->quantum_level = level_quantum_expired[ppp->level];
  }
}
```

The first for loop here initializes the value of all the elements of the 2 arrays into the value of RSDL_LEVEL_QUANTUM. For the second loop, it assigns the quantum_level of each process to the correct one, depending if it is active or expired.

```c
if (p->quantum_level < 0) p->quantum_level=0;
level_quantum_active[p->level]=p->quantum_level;
```

Inside the code for schedlog, it is made sure that the values never go down zero. The level_quantum_active is also updated here to let the other processes later to be able to access their correct quantum_level.

```c
for (int l = 0; l < RSDL_LEVELS; l++){
  cprintf("%d|active|%d(%d)", ticks, l, level_quantum_active[l]);
```

```c
for (int l = 0; l < RSDL_LEVELS; l++){
  cprintf("%d|expired|%d(%d)", ticks,l, level_quantum_expired[l]);
```

The part of the code that prints the quantum of the level is above separated for the active and expired sets. They just use l since that is their correct level.

```c
int sorted[NPROC];
    int array[NPROC];
    for (int k = 0; k < NPROC; k++) {
        sorted[k]=k;
        struct proc *pp;
        pp = &ptable.proc[k];
        array[k]=pp->queue_num;

    }

    for (int k = 0; k < NPROC; k++) {
        for (int j = k+1; j < NPROC; j++) {
            if(array[k] > array[j]) {
                int temp = sorted[k];
                sorted[k]=sorted[j];
                sorted[j]=temp;


                int temp2 = array[k];
                array[k] = array[j];
                array[j] = temp2;
            }
        }
    }
```

After the part for schedlog, we do a imaginary sort again to be able to use the correct index later.

```c
int currlevel = p->level;
if (p->quantum_level < 0) p->quantum_level=0;
level_quantum_active[currlevel]=p->quantum_level;
```

Here, we just make sure that the level quantum will never go negative and update the level_quantum_active again.

```c
int currlevel = p->level;
if (p->quantum_level < 0) p->quantum_level=0;
level_quantum_active[currlevel]=p->quantum_level;

if (p->quantum_level<=0){
  for (int k = 0; k < NPROC; k++){
      struct proc *ppp;
      ppp = &ptable.proc[sorted[k]];
      if (ppp->level != currlevel) continue;
      num+=1;
      ppp->queue_num = num;
      ppp->level += 1;
      ppp->ticks_left=RSDL_PROC_QUANTUM;
      if (ppp->level >= RSDL_LEVELS) {
        if (ppp->prio != -1) ppp->level = ppp->prio;
        else ppp->level=RSDL_STARTING_LEVEL;
        ppp->active=0;
        ppp->ticks_left=RSDL_PROC_QUANTUM;
      }
    }
    num+=1;
    ppp->queue_num = num;
}
else if (p->ticks_left == 0 && p->active == 1){
  num+=1;
  p->queue_num = num;
  p->level++;
  p->ticks_left=RSDL_PROC_QUANTUM;
  if (p->level == RSDL_LEVELS) {
    if (p->prio != -1) p->level = p->prio;
    else p->level=RSDL_STARTING_LEVEL;
    p->active=0;
    p->ticks_left=RSDL_PROC_QUANTUM;
  }
}
```

From the last phase, this part is just a single if statement. But we can't combine the scenario of the quantum_level being zero and the ticks_left being zero into one because the p->level of the current process will go up twice. Thus, we first check for the quantum_level having a value of 0. If it is, all processes in the current level will go to the next level. Since, the queue here is just a visualization, we can just additionally increase the queue_num of the current process to let it be the last process to be enqueued.

The else if statement below is just the same from the last phase.

```c
if (all_expired == 1){
  p->reset = 1;
  num += 1;//phase1F
  p->queue_num = num;//phase1
```

Here is when the reset of the PCB comes in. If all the runnable processes are in the expired set, we need to reset the quantum of all the levels because a swap will be happening.

```c
if (p->reset == 1){
  for (int i = 0; i < RSDL_LEVELS; i++){

    level_quantum_active[i] = RSDL_LEVEL_QUANTUM;
  }
  p->reset = 0;
}
```

If a swap is to be done, a reset should also be done. We just initialize the quantum of all the levels of active sets to be RSDL_LEVEL_QUANTUM again. Then, the reset should stay 0 agin till another reset will be done.

16. [........5] Implementation explanation for priofork

To implement priofork, we changed the required files to create a syscall.
user.h

```
int priofork(int);
```

usys.S

```
SYSCALL(priofork)
```

syscall.h

```
#define SYS_priofork  25
```

Syscall.c

```
extern int sys_priofork(void);
```

```
[SYS_priofork] sys_priofork,
```

Sysproc.c

```
int
sys_priofork(void)
{
  int priority;

  if(argint(0, &priority) < 0)
    return -1;
  return priofork(priority);
}
```

This will pass the value of the priofork called into the function of priofork.

Defs.h

```
int             priofork(int);
```

Aside from those, there is also a change in the PCB in proc.h.

```
  int prio;
```

int prio has been added to it. This will tell us later whether fork or priofork has been used to create a process alongside with the value used by priofork.

The remaining changes are in the proc.c.

Our implementation of priofork is just the same as fork. We just duplicated the fork function in proc.c to create a function named priofork. The difference is that priofork has an int as a parameter.

```
np->state = RUNNABLE;
np->prio = -1;
```

This is in the fork function. We just set np->prio to -1 which will be passed in the ptable. If prio is -1, it means that the process is created by fork.

```c
int
priofork(int priority)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

```

```
acquire(&ptable.lock);

np->state = RUNNABLE;
np->prio = priority;
np->level = priority;

release(&ptable.lock);

return pid;
}
```

This is the whole priofork function. It only has small changes from fork.

```
np->state = RUNNABLE;
np->prio = priority;
np->level = priority;
```

When priofork is called, np->prio will become the value used by priofork. We also set the level of the process to it. This will override the previous level at this point which is the RSDL_STARTING_LEVEL.

```c
if (p->quantum_level<=0){
    for (int k = 0; k < NPROC; k++){
            struct proc *ppp;
            ppp = &ptable.proc[sorted[k]];
            if (ppp->level != currlevel) continue;
            num+=1;
            ppp->queue_num = num;
            ppp->level += 1;
            ppp->ticks_left=RSDL_PROC_QUANTUM;
            if (ppp->level >= RSDL_LEVELS) {
                if (ppp->prio != -1) ppp->level = ppp->prio;
                else ppp->level=RSDL_STARTING_LEVEL;
                ppp->active=0;
                ppp->ticks_left=RSDL_PROC_QUANTUM;
            }
        }
        num+=1;
        ppp->queue_num = num;
}
else if (p->ticks_left == 0 && p->active == 1){
    num+=1;
    p->queue_num = num;
    p->level++;
    p->ticks_left=RSDL_PROC_QUANTUM;
    if (p->level == RSDL_LEVELS) {
        if (p->prio != -1) p->level = p->prio;
        else p->level=RSDL_STARTING_LEVEL;
        p->active=0;
        p->ticks_left=RSDL_PROC_QUANTUM;
    }
}
```

The last changes are here. When a process is to exceed the max level, the prio is checked whether it is created by for or priofork. If it is created by priofork, instead of resetting its level to the RSDL_STARTING_LEVEL, we set it to its prio.