Felix Bueno 2020-05609
Ralph Diaz 2020-00356
James Adrian Perez 2020-05633

CS 140 Project 1: RSDL Variant Implementation

This documentation contains all the information regarding the project including but not limited to the list of changes made in the files/codes per phase as well as the implementation for the different requirements such as the schedlog, swapping of sets, and replenishment of quanta. This document also contains the link for the google drive which contains the document video to be passed.

Introduction

This project is similar to the lab exercise we did in the subject, mainly lab exercise 5 from which we got he schedlog code with minor changes as well as the idea for the project. Project 1 is mainly this exercise with added functions divided into 5 phases. Some of these functions are making the originally round-robin scheduler FIFO based, creation of two sets called active and expired which have levels, implementation for enqueuing and dequeuing of the processes as well as some rules for swapping of sets as well as replenishment of the quanta of both processes and level quantum. All of these will be discussed thoroughly throughout the project documentation.

Important Auxiliary Functions and Other Changes

There are several helper functions used throughout the project to help their implementation. Listed below are these functions as well as their code snippets and explanation.

rsdl.h

The file rsdl.h contains #define statements for constants that shall be used throughout the implementation of this project. Shown below is a code snippet that contains the said information:

```
1   #define RSDL_LEVELS 3
2   #define RSDL_STARTING_LEVEL 0
3   #define RSDL_PROC_QUANTUM 50
4   #define RSDL_LEVEL_QUANTUM 100
```

Figure i. Code Snippet of rsdl.h

The file defines the values of four constants. First is the RSDL_LEVELS which is set to 3. This corresponds to the number of priority levels as there are three levels for the active set and three levels for the expired set. The second constant defined is RSDL_STARTING_LEVEL which we have set to 0 since all processes will start at active level 0 (with the exception of those that called the priofork system call). The next constant defined is RSDL_PROC_QUANTUM which is the length of the quantum (in ticks) assigned by default to each process. We have set its value to 50 ticks. Lastly, the RSDL_LEVEL_QUANTUM is the length of the quantum (also in ticks) assigned by default to each level. We have set its value to 100 ticks.

queueclean

This helper function mainly cleans the queue for several factors. The code for this function is shown below. To do this, we first find the highest unused process. There are two parts and conditions for using queueclean. The first one is clearing up the queue in the active set after the processes goes on to the expired set. This allows the program to have the correct order in queue once the processes go back to the active set. To do this, we first check if the process is in the expired set as well as if this process is running and has no quantum left. If so, we use the for loop to clear the queue for all the processes that went to the expired set. Another time for using the queueclean if for when there are zombie and unused processes, processes that went down a level, as well as for finished processes. To do this, we once again check the conditionals and if so, we loop through to clean the queue.

```
412  void queueclean(void){
413    //Remove UNUSED, ZOMBIE, and PROC with 0 Quantum from the queue of both the active and expired set.
414    struct proc *pp;
415    struct proc *xp;
416    int temp_active = 0;
417    int highest_idx = -1;
418
419    //Identify highest idx
420    for (int k = 0; k < NPROC; k++) {
421          pp = &ptable.proc[k];
422          if (pp->state != UNUSED) highest_idx = k;
423    }
424    //Clear up Queue in the Active Set for processes done after a process went to expired
425    for (int k = 0; k <= highest_idx; k++) {
426          pp = &ptable.proc[k];
427          if(pp->state == RUNNING && pp->ticks_left == 0 && pp->set == 1 && pp->level == pp->starting_level){
428                  temp_active = pp->queue;
429                  pp->queue = -1;
430                  for (int i = 0; i<=highest_idx; i++){
431                          xp = &ptable.proc[i];
432                          if(temp_active < xp->queue && xp->set == 0 && xp->level == RSDL_LEVELS-1)
433                                  xp->queue = xp->queue - 1;
434                  }
435          }
436    }
```

Figure ii. Snippet Code of queueclean Function

```
438     //Clear up Queue in the Active Set for process done but went down a level and for zombie and unused processes
439     for (int j = 0; j < RSDL_LEVELS; j++){
440           for (int k = 0; k <= highest_idx; k++) {
441                   pp = &ptable.proc[k];
442                   if((((pp->state == UNUSED && pp->queue != -1) || (pp->state == ZOMBIE && pp->queue != -1)) && pp->set == 0 && pp->level == j) ||
443                   (pp->state == RUNNING && pp->ticks_left == 0 && pp->set == 0 && pp->level == j+1)){
444                           temp_active = pp->queue;
445                           pp->queue = -1;
446                           for (int i = 0; i<=highest_idx; i++){
447                                   xp = &ptable.proc[i];
448                                   if(temp_active < xp->queue && xp->set == 0 && xp->level == j)
449                                           xp->queue = xp->queue - 1;
450                           }
451                   }
452           }
453     }
454   }
```

Figure iii: Continuation of the Snippet Code of queueclean Function

queuesort

This is another helper function where its main purpose is to find the next queue which is concerned with the order of the processes. The code for this function is shown below. To do this, we first find the highest unused process as well as clean the queue using the auxiliary function queueclean. We then find the next queue by looping taking into account the sets and the levels. After finding the highest queue, we increment 1 to the value of it to get the value of the next queue.

```
482   int queuesort(int pset, int plevel){
483     //acquire(&ptable.lock);
484     //Find the next queue
485     struct proc *pp;
486     int nextqueue = -1;
487     int highest_idx = -1;
488
489     //Identify highest idx
490     for (int k = 0; k < NPROC; k++) {
491           pp = &ptable.proc[k];
492           if (pp->state != UNUSED) highest_idx = k;
493     }
494
495     //release(&ptable.lock);
496     //Call queueclean which cleans the queue of both active and expired sets
497     queueclean();
498     //UNUSED and ZOMBIE proccesses are clear. Find the highest in the queue
499     //Consider Set and Level
500     for (int k = 0; k <= highest_idx; k++) {
501     pp = &ptable.proc[k];
502           if((pp->state == SLEEPING || pp->state == RUNNING || pp->state == RUNNABLE) && pp->set == pset && pp->level == plevel){
503                   if(pp->queue > nextqueue)
504                           nextqueue = pp->queue;
505           }
506     }
507     return nextqueue+1;
508   }
```

Figure iv: Code Snippet of queuesort Function

expired_check

This helper function's main purpose is to check if there is a process left in the specific level (set by the user) in the expired set. This works by looping through the maximum amount of processes (64) and returning 1 if there is a process found. We know this if the value of the set of the process is 1 and its level is the same as the parameter.

```
int expired_check(int lvl){
        struct proc *xp;
        for(int k = 0; k < NPROC; k++){
          xp = &ptable.proc[k];
          //Find if there is a process in the expired level set
          if(xp->set == 1 && xp->level == lvl) return 1;
        }
        return 0;
}
```

Figure v: Code for Expired_Check Function

active_check

Similar to the auxiliary function expired_check, this function checks on whether there are functions in the active set in the specific level set by the user. The conditionals for this are a bit different however since we want processes that are initialized and not killed. We also added the condition for the state of the process found to not be unused since this means they weren't initialized yet and that they are not zombies.

```
int active_check(int lvl){
        struct proc *xp;
        for(int k = 0; k < NPROC; k++){
          xp = &ptable.proc[k];
          //Find if there is a process in the active level set
          if(xp->set == 0 && xp->level == lvl && xp->pid != 0 && xp->state != UNUSED && xp->state != ZOMBIE) return 1;
        }
        return 0;
}
```

Figure vi: Code Snippet for Auxiliary Function active_check

sleepclean

Similar to the auxiliary function queueclean, this function searches for the passed process coming from the sleep function (identified by its pid, and sleeping state) and

effectively rearrange the queue order as the process who went to sleep will be reenqued to the back of the same level. This would place that process at the back.

```
456  void sleepclean(int spid){
457          struct proc *pp;
458          struct proc *xp;
459          int temp = 0;
460          int level = 0;
461          int highest_idx = -1;
462          for (int k = 0; k < NPROC; k++) {
463                  pp = &ptable.proc[k];
464                  if (pp->state != UNUSED) highest_idx = k;
465          }
466
467          for (int k = 0; k <= highest_idx; k++) {
468                  pp = &ptable.proc[k];
469                  if(pp->state == RUNNING && pp->pid == spid){
470                          temp = pp->queue;
471                          level = pp->level;
472                          pp->queue = -1;
473                          for (int i = 0; i<=highest_idx; i++){
474                                  xp = &ptable.proc[i];
475                                  if(temp < xp->queue && xp->set == 0 && xp->level == level)
476                                          xp->queue = xp->queue - 1;
477                          }
478                  }
479          }
480  }
```

Figure vii. Code Snippet for Auxiliary Function sleepclean

Note that along with this auxiliary function, we modified the sleep() call by adding additional lines to accommodate this edge case. In lines 860 to 865, we added the sleepclean function and the queuesort to accommodate the reenqueue of the process involve in the sleep process.

```
858      // Go to sleep.
859      p->chan = chan;
860      sleepclean(p->pid);
861      p->state = SLEEPING;
862
863      release(&ptable.lock);
864      p->queue = queuesort(p->set, p->level);
865      acquire(&ptable.lock);
866
867      sched();
868
869      // Tidy up.
870      p->chan = 0;
```

Figure viii. Changes Made in the Sleep Syscall

1. List of all Phases with Working Code

In total, there are 5 phases of the project all of which have added functions over the previous one. For this project, the group was able to make up to the last phase of the project, **phase 5**. Codes for the different phases were saved in different branches. Phase 1 was saved in the branch **phase1**, phase 2 was saved in the branch **phase2**, phase 3 was saved in the branch **phase3**, phase 4 was saved in the branch **phase4**, and lastly phase 5 was saved in the branch **phase5**. For this project, we will discuss the entirety of the project meaning phase 5 only. The discussion of the last phase should include the discussions of the lower phases as phase 5 encompasses all the phases.

2. All references used with purpose specified

There were **no external references** used for any of the phases done in this project. The only reference used was the lab exercise 5 we were given in the past which used a schedlog code as well as the user test code (test.c). The group also made it as the basis for the project.

3. List of All Global Variables Introduced

There is only one global variable added in during the several phases of the project. These variables are the following:

- **Struct plevel** - used for storing the necessary values for each level such as the level, set, and the level quantum as well as for implementing the levels in phases 2 - 5. Struct lvltable was used for this and will be further discussed in the implementation of the levels.

4. List of Changes Made to the Process Control Block

There are a total of five (5) additions to the process control block which are:

- **ticks_left** - which was used for the amount of quantum/ticks left needed before a process finishes.
- **queue** - used for the ordering or queuing of processes in the right order
- **set** - used for recording/manipulating the active and expired sets.
- **level** - used for recording/finding the values in the different levels of the program (i.e. level 0 - N-1)
- **starting_level** - used for storing the value of the starting level of the processes.

5. Code Representation of the Active Set and its Levels

The code representation of the active set as well as its corresponding levels can be found in a function called **levelinit** in the proc.c file. The code involved in this is seen in figure 1 below.

```
void levelinit(void){
        struct qlevel *rp;
        //Init active levels
        for(int i=0;i<RSDL_LEVELS;i++){
                rp = &plevel.qlevel[i];
                rp->set = 0;
                rp->level = i;
                rp->lvlquantum = RSDL_LEVEL_QUANTUM;
        }
```

Figure 1: Code Snippet for Active Set and its Levels

Before we discuss this, we must first discuss its corequisite found in the file proc.h which has a new addition of a struct called qlevel. This contains the necessary value for initializing the set and its levels. The code for this is shown below.

```
65   struct qlevel{
66           int level;
67           int set;
68           int lvlquantum;
69   };
```

Figure 2: qlevel Struct Added in Proc.h

To discuss the code in figure 1, we first initialize the variable *rp with the struct qlevel which means each rp will hold the value for level, set, and lvlquantum. The group then used a for loop which loops through number 0 to N-1 for the levels initialization. This allowed the program to have their own values stored in a struct for each level in the active set.

6. Implementation Explanation for Initial Enqueuing of New Processes

The implementation for the initial enqueuing of a new process is very similar to the initial enqueuing of a new process done in the lab exercise 5 wherein if we found an unused process in the process table then we will initialize its needed values as well as change its state to embryo. The codes for this can be found in the **allocproc** function in proc.c file. Since there were additional functions in the phases of the project then there will be additions to the initial enqueuing implementation as well. These changes are shown and further discussed below.

To start with, the general explanation for the implementation of the initial enqueuing is to look for an unused process, if found change its state to embryo (a new process) and initialize its values, run in the kernel else return 0. There were no changes to the allocation of a stack to the process since the changes only mainly involved the initial values of the processes since there were levels and quantums added. These changes can be found in the **found** section of the allocproc function.

```
90   found:
91     p->state = EMBRYO;
92     p->pid = nextpid++;
93     p->set = 0;
94
95     //Check if starting level is available
96     int result = searchlevel(RSDL_STARTING_LEVEL);
97     if(result != -1)
98             p->level = result;
99     else{
100            p->set = 1;
101            p->level = RSDL_STARTING_LEVEL;
102    }
103
104    p->starting_level = RSDL_STARTING_LEVEL;
105    p->ticks_left = RSDL_PROC_QUANTUM;
106    release(&ptable.lock);
107
108    temp = queuesort(p->set, p->level);
109
110    acquire(&ptable.lock);
111    p->queue = temp;
112    release(&ptable.lock);
```

Figure 3: Changes Done in allocproc Function for New Initial Enqueuing Implementation

Figure 3 shows the relevant additions to the initialization of a new process. If there is an unused process found, we proceed as normal in making its state into an embryo (new process) as well as add a unique pid to it. The change however is that we also add a value to its p->set to 0 since we want the new processes to go to the active set instead of the expired set if they are new. We also search for the lowest active level using the function **searchlevel**. Figure 4 shows the code for the function search level and what it mainly does is search for the available active levels meaning active levels whose quantum are not yet depleted. This was done since there can be a possibility that the starting level has no available quantum already when the new process gets enqueued so the process doesn't start in the starting level or that no such active level exists so it starts out in the starting level in the expired set. After setting the new process' current level to be in any active level or in the starting level of the expired set, we also initialize other important values such as its starting_level which will be used for the swapping as well as its current quantum using the variable ticks_left. We then get its queue using the queuesort function which both finds the next queue as well as rid of the queue of zombie processes, finished processes, and downleveled processes and other more. We do this since we follow a FIFO based round-robin scheduler which means that all new processes must be at the last of the queue.

```
400  //Search available active levels
401  int searchlevel(int start){
402          struct qlevel *rp;
403          for(int i=0;i<RSDL_LEVELS;i++){
404                  rp = &plevel.qlevel[i];
405                  if(rp->level == start && rp->lvlquantum != 0) return start;
406                  else if(rp->level > start && rp->lvlquantum != 0) return rp->level;
407          }
408          //Return -1 if there are no available active levels
409          return -1;
410  }
```

Figure 4: Code Snippet for searchlevel Function

7. Implementation Explanation for Dequeuing of Exiting Processes

The implementation for the dequeuing of exiting/finished processes is very simple. To account for the finished process we mainly use a part of the code in the **queueclean** function shown below. Notice how this code is used for both exiting processes as well as zombie and unused processes. The particular condition used for exiting processes is the line "**if pp->state == UNUSED && pp->queue != -1**" which means that a process only became unused since it exited successfully and we know this since its queue did not change. Notice in line 441 that we have the value of the queue of exiting processes to be

-1. This is how we know that the process we checked for has finished successfully and to dequeue this we need to change its queue value to -1. Doing so makes the program ignore this process in printing and other functions since it's no longer part of the queue.

```
438     //Clear up Queue in the Active Set for process done but went down a level and for zombie and unused processes
439     for (int j = 0; j < RSDL_LEVELS; j++){
440           for (int k = 0; k <= highest_idx; k++) {
441                   pp = &ptable.proc[k];
442                   if((((pp->state == UNUSED && pp->queue != -1) || (pp->state == ZOMBIE && pp->queue != -1)) && pp->set == 0 && pp->level == j) ||
443                   (pp->state == RUNNING && pp->ticks_left == 0 && pp->set == 0 && pp->level == j+1)){
444                           temp_active = pp->queue;
445                           pp->queue = -1;
446                           for (int i = 0; i<=highest_idx; i++){
447                                   xp = &ptable.proc[i];
448                                   if(temp_active < xp->queue && xp->set == 0 && xp->level == j)
449                                           xp->queue = xp->queue - 1;
450                           }
451                   }
452           }
453     }
454   }
```

Figure 5: Code Snippet for Implementation of Dequeuing of Exiting Processes

8. Implementation Explanation for Process-Local Quantum Consumption

Relevant codes for Process-local quantum consumption can be found in the trap.c file along with Level-local quantum consumption. Code snippet regarding this is shown in figure 6 below. What the code in the snippet does is it checks whether there is a process and whether it is running as well as check if its trapframe trap number is the same as the previous user space registers before its context switches. The important part here however is the code **--myproc()->ticks_left** which is the main code for the quantum consumption process-wise. What it does is for every tick, it decrements its quantum (the ticks_left variable) by 1.

```
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER){
      for(rp = plevel.qlevel; rp < &plevel.qlevel[RSDL_LEVELS*2]; rp++){
              if(rp->level == myproc()->level && rp->set == myproc()->set){
              //cprintf("Current Proc: PID-%d, Level-%d, Set-%d\n", myproc()->pid, myproc()->level, myproc()->set);
              //cprintf("Current RP: Level-%d, Set-%d\n", rp->level, rp->set);
                      break;}
      }
      --rp->lvlquantum;
      --myproc()->ticks_left;
```

Figure 6: Code Snippet for Process-Local Quantum Consumption

9. Implementation Explanation for Process-Local Quantum Replenishment

Similarly to the process-local quantum consumption, the replenishment of the quantum process-wise is replenished every time the process is enqueued into a new level as well as every time a process goes to the expired set. Since there are several conditions for replenishment of the process-local quantum, there are several conditionals which check for all these situations. The code snippet regarding this is shown below.

```
if (rp->lvlquantum == 0 || myproc()->ticks_left == 0){
    if (rp->lvlquantum == 0 && myproc()->level != RSDL_LEVELS - 1 && myproc()->ticks_left == 0){
            myproc()->ticks_left = RSDL_PROC_QUANTUM;
            leveldequeue(myproc()->level);
            yield();
    }

    else if (rp->lvlquantum == 0 && myproc()->level != RSDL_LEVELS - 1 && myproc()->ticks_left > 0){
            leveldequeue(myproc()->level);
            yield();
    }

    else if (rp->lvlquantum == 0 && myproc()->level == RSDL_LEVELS - 1 && myproc()->ticks_left > 0){
            leveldequeue(myproc()->level);
            yield();
    }

    else if (rp->lvlquantum != 0 && myproc()->level != RSDL_LEVELS - 1 && myproc()->ticks_left == 0){
            int result = searchlevel(myproc()->level+1);
            if(result != -1)
                    myproc()->level = result;
            else{
                    myproc()->set = 1;
                    myproc()->level = myproc()->starting_level;
            }
            myproc()->queue = queuesort(myproc()->set, myproc()->level);
            myproc()->ticks_left = RSDL_PROC_QUANTUM;
            yield();
    }
    else if (rp->lvlquantum != 0 && myproc()->level == RSDL_LEVELS - 1 && myproc()->ticks_left == 0){
            myproc()->set = 1;
            myproc()->level = myproc()->starting_level;
            myproc()->queue = queuesort(myproc()->set, myproc()->level);
            myproc()->ticks_left = RSDL_PROC_QUANTUM;
            yield();
    }
    else if (rp->lvlquantum == 0 && myproc()->level == RSDL_LEVELS - 1 && myproc()->ticks_left == 0){
            myproc()->ticks_left = RSDL_PROC_QUANTUM;
            leveldequeue(myproc()->level);
            yield();
```

Figure 7: Code Snippet Relevant to Process-Local Quantum Replenishment

The codes we are interested in are the conditionals that have the code **myproc()->ticks_left = RSDL_PROC_QUANTUM** which are highlighted in figure 7. The general conditional checks on whether the quantum of the level is 0 since this causes all the processes to get enqueued into a new level thus replenishing their quantum or whether the quantum of the process is emptied out or reaches 0 which causes it either to get enqueued into a lower priority level or into the expired set. The conditional for the first box checks whether the level reaches 0 quantum, whether it's not the lowest priority level, and if the process quantum reaches 0 as well. This condition causes the process to get enqueued into a lower priority level thus replenishing its quantum by setting its quantum (ticks_left) to the original quantum (RSDL_PROC_QUANTUM). The second box checks for the condition wherein the level quantum still exists but the process quantum reaches 0 meaning it needs to be dequeued and enqueued into a lower priority level. Similarly to the first box, it replenishes its quantum by setting the variable of the ticks_left to the original process quantum. Lastly, the third box checks for the condition for the level quantum as well as the process quantum reaching 0 and the level being the lowest priority level in which case it goes to the expired set and replenishes its quantum through the same process.

Additionally, we have also process replenishment for processes who went down a level because of their level-local quantum being exhausted. This is seen in the leveldequeue function and switch_set function. Note that the process in *qq* is the last active process and should be dequeued last and their quantum to be replenished.

```
for(int k = 0; k < NPROC; k++){
        xp = &ptable.proc[k];
        if(xp->state == RUNNING && xp->level == lvl && xp->set == 0 && xp->state != UNUSED && xp->state != ZOMBIE)
                qq = xp;
        else if(xp->level == lvl && xp->set == 0 && xp->state != UNUSED && xp->state != ZOMBIE){
                if(result != -1)
                        xp->level = result;
                else{
                        xp->set = 1;
                        xp->level = xp->starting_level;
                }
                xp->ticks_left = RSDL_PROC_QUANTUM;
                xp->queue = queuesort(xp->set, xp->level);
        }
}

if(qq){
        if(result != -1){
                qq->level = result;
        }
        else{
                qq->set = 1;
                qq->level = qq->starting_level;
        }
        qq->ticks_left = RSDL_PROC_QUANTUM;
        qq->queue = queuesort(qq->set, qq->level);
```

```
for (int i = 0; i <= maxqueue; i++){
        for (int k = 0; k < NPROC; k++) {
                pp = &ptable.proc[k];
                if (pp->set == 0 && (pp->state != UNUSED && pp->state != ZOMBIE) && pp->queue == i){
                        acquire(&ptable.lock);
                        pp->queue = -1;
                        pp->set = 1;
                        pp->level = pp->starting_level;
                        release(&ptable.lock);
                        pp->queue = queuesort(pp->set, pp->level);
                }
        }
}

acquire(&ptable.lock);
for (int k = 0; k < NPROC; k++) {
                pp = &ptable.proc[k];
                if (pp->state != UNUSED && pp->state != ZOMBIE){
                        pp->ticks_left = RSDL_PROC_QUANTUM;
                        pp->set = 0;
                }
        }
```

Figure 8: Code Snippets Relevant to Process-Local Quantum Replenishment

10. Implementation Explanation for Schedlog

Similar to lab exercise 5, we were required to make a schedlog syscall. As it is fairly similar, we made the schedlog syscall in the lab exercise as our base code and made changes from thereon. The following steps are very important so that the system will recognize our implemented syscalls when in both user and kernel interface. To implement the schedlog syscall there were several steps.

1. Add int schedlog(int) in user.h

```
44   int schedlog(int);
```

2. Add SYSCALL(schedlog) is usys.S

```
34   SYSCALL(schedlog)
```

3. Add #define SYS schedlog 24 in syscall.h

```
25   #define SYS_schedlog  24
```

4. Add [SYS schedlog] sys schedlog, and extern int sys schedlog(void); in syscall.c

```
108   extern int sys_schedlog(void);
135   [SYS_schedlog] sys_schedlog,
```

5. Add the code block below in sysproc.c. The codeblock takes in an integer input (returns -1 if the argument pass has failed) and calls the schedlog function in the proc.c, passing the maximum number of ticks that the schedlog will print.

```
107   int sys_schedlog(void) {
108           int n;
109           if(argint(0, &n) < 0)
110                   return -1;
111           schedlog(n);
112           return 0;
113   }
```

6. Add the code block below which forms the schedlog function to be used in the scheduler in proc.c before the scheduler function. This is used to flag that the schedlog is active and tracks how long (in ticks) will the schedlog be active.

```
645   void schedlog(int n) {
646           schedlog_active = 1;
647           schedlog_lasttick = ticks + n;
648   }
```

7. Add void schedlog(int); in defs.h

```
193   void                    schedlog(int);
```

Recall that although the codes from the lab exercise can mostly be copied, the format for the schedlog in the project changes according to the phase. For phase 5, the format is shown below in figure 9.

- **Phase 4 and 5:** `<tick>|<set>|<level>(<quantum left>),[<PID>]<process name>:<state number>(<quantum left>),...`

Figure 9: Schedlog Format for Phase 5

Recall that the main function of the schedlog syscall is to print out all the relevant information before each context switch. To generate the format shown in figure 9, we adjusted the code provided in lab exercise 5. Below is the relevant code snippet for this format. We separated the printing into two parts, mainly the general information such as the tick, set, and level and process information where we printed down all the necessary processes in that specific set and level as well as its information such as the process-local quanta. The code for printing the general information is shown in figure 8, the 1st part of the code snippet. Notice the conditionals if, else if, and else which are used for printing out the general information for different conditions of the processes. The first conditional is for checking if the process is in the expired set as well as checking if there is a process in the expired set using the function expired_check. The second conditional is for checking for active sets and whether there are processes in the active set. The else conditionals is for situations wherein the level has no processes enqueued into them. It's also worth mentioning that the outer loop i is for looping through the two sets (active and expired) while the loop for j is for looping through the 0 to N-1 levels.

```
if (schedlog_active) {
        if (ticks > schedlog_lasttick) {
                schedlog_active = 0;
        } else {
                for (int i = 0; i < 2; i++){
                        for (int j = 0; j < RSDL_LEVELS; j++){
                                //Track the highest queue
                                struct proc *pp;
                                int highest_idx = -1;

                                //DEBUG
                                /*
                                struct qlevel *rrp;
                                for(rrp = plevel.qlevel; rrp < &plevel.qlevel[RSDL_LEVELS*2]; rrp++){
                                        cprintf("Set=%d, Level=%d, Quantum=%d\n", rrp->set, rrp->level, rrp->lvlquantum);
                                }
                                */

                                struct qlevel *rp;
                                for(rp = plevel.qlevel; rp < &plevel.qlevel[RSDL_LEVELS*2]; rp++){
                                        if(rp->level == j && rp->set == i)
                                                break;
                                }

                                if (i == 1 && expired_check(j)) cprintf("%d|%s|%d(%d),", ticks, curr_set[i], j, rp->lvlquantum);
                                else if (i == 0 && active_check(j)) cprintf("%d|%s|%d(%d),", ticks, curr_set[i], j, rp->lvlquantum);
                                else cprintf("%d|%s|%d(%d)", ticks, curr_set[i], j, rp->lvlquantum);

                                for (int k = 0; k < NPROC; k++) {
                                        pp = &ptable.proc[k];
                                        if (pp->queue > highest_idx && pp->set == i && pp->level == j) {
                                                highest_idx = pp->queue;
                                        }
                                }
```

Figure 10: Part 1 of 2 of the Code Snippet for Schedlog

For the second part of the implementation, we can now iterate through the *ptable* as we know the number of current process in the current level and set. We will use this information to print the queue. Meaning the queue should be equal to *highest_idx*. While iterating through the *ptable*, the process with current queue number, its state being unused and adheres to the current set and level will be printed by this loop. This would print the queue processes in order along with their proper information. Lastly, the last for loop looks for the last process in the queue and avoids printing “,” to comply with the formatting stated.

```
for (int x = 0; x < highest_idx; x++){
        for (int k = 0; k <= NPROC; k++){
                pp = &ptable.proc[k];
                if (x == pp->queue && pp->state != UNUSED && pp->set == i && pp->level == j){
                        //cprintf("[%d]%s:%d(%d), Q=%d set=%d level=%d ", pp->pid, pp->name, pp->state, pp->ticks_left, pp->queue,

                        cprintf("[%d]%s:%d(%d),", pp->pid, pp->name, pp->state, pp->ticks_left);
                        break;
                }
        }
}
for (int k = 0; k <= NPROC; k++) {
        pp = &ptable.proc[k];
        if (highest_idx == pp->queue && pp->state != UNUSED && pp->set == i && pp->level == j && pp->queue >= 0){
                        //cprintf("[%d]%s:%d(%d) Q=%d set=%d level=%d ", pp->pid, pp->name, pp->state, pp->ticks_left, pp->queue, pp-

                        cprintf("[%d]%s:%d(%d)", pp->pid, pp->name, pp->state, pp->ticks_left);
                        break;
        }
}
```

Figure 11: Part 2 of 2 of the Code Snippet for Schedlog

11. Code Representation of the Expired Set and its Levels

The code representation for the expired set and its levels is found in the proc.c file inside the **levelinit** function. This function contains the code representation of both the active and the expired set. The code for this is shown below inside the box. Similar to the active set, we initialize the expired set using the struct qlevel which includes the values for the current level (level), its set (active or expired using 0 or 1), as well as level quantum (quantum of the specific level). To initialize the expired set, the group used a for loop to loop through all the N-1. So as to not overwrite the values of the active level, the group used its initial level for the expired set (0 the highest priority level) as N and set its maximum to N * 2. This allowed us to have both the active set and expired set without overwriting their values. For each iteration, we then initialized the values of the level being i - RSDL_LEVELS since we still need its levels to be labeled as 0 to N-1, the value of the set being 1 which corresponds to it being the expired set as well as the quantum level set to the original value which is the RSDL_LEVEL_QUANTUM.

```
void levelinit(void){
        struct qlevel *rp;
        //Init active levels
        for(int i=0;i<RSDL_LEVELS;i++){
                rp = &plevel.qlevel[i];
                rp->set = 0;
                rp->level = i;
                rp->lvlquantum = RSDL_LEVEL_QUANTUM;
        }
        //Init expired levels
        for(int i=RSDL_LEVELS;i<RSDL_LEVELS*2;i++){
                rp = &plevel.qlevel[i];
                rp->set = 1;
                rp->level = i - RSDL_LEVELS;
                rp->lvlquantum = RSDL_LEVEL_QUANTUM;
        }
}
```

Figure 12: Code Representation for the Expired Set and its Levels

12. Implementation Explanation for Transferring a Process from Active to Expired

A process will only go to the expired set when its current level is the lowest priority level (N-1) and either the process quantum or the level quantum reaches 0 during which they will then go to the expired set. The implementation for the transfer then starts from the trap.c file where we first check if the quantum reaches 0. This can be seen in the code snippet shown below. Figure 13 shows the condition wherein the level quantum reaches 0 but the current running process still has quantum. When this happens, all the processes from that level will be dequeued and enqueued into the expired set since the current level is RSDL_LEVELS - 1 (N - 1). Figure 14 shows two conditionals. The first conditional shows when the process in the lowest priority level runs out of quantum wherein only that process will be enqueued into the expired set on the original priority level by calling leveldequeue. We also redo the value of the level to the starting level since we want the process to be enqueued into the original priority level. We also call queuesort() to make sure that we remove the process in the active queue using queueclean as well as to ensure that the process gets enqueued into the last index since we follow the FIFO standard. We also replenish the process' quantum since we have the rule that the quantum gets replenished when we transfer a process from the active to the expired. The last conditional is for when both the process quantum as well as the level quantum reaches 0. For conditionals 1 and 3 we call on level dequeue since we dequeue all processes from that specific level whenever our level quantum reaches 0.

```
else if (rp->lvlquantum == 0 && myproc()->level == RSDL_LEVELS - 1 && myproc()->ticks_left > 0){
        leveldequeue(myproc()->level);
        yield();
}
```

Figure 13: First Conditional for Transferring Processes from Active to Expired

```
else if (rp->lvlquantum != 0 && myproc()->level == RSDL_LEVELS - 1 && myproc()->ticks_left == 0){
        myproc()->set = 1;
        myproc()->level = myproc()->starting_level;
        myproc()->queue = queuesort(myproc()->set, myproc()->level);
        myproc()->ticks_left = RSDL_PROC_QUANTUM;
        yield();
}
else if (rp->lvlquantum == 0 && myproc()->level == RSDL_LEVELS - 1 && myproc()->ticks_left == 0){
        myproc()->ticks_left = RSDL_PROC_QUANTUM;
        leveldequeue(myproc()->level);
        yield();
}
```

Figure 14: Conditionals for Transferring Processes from Active to Expired

We then discuss the relevant code in the leveldequeue function for which the processes in the lowest priority level all go to the expired set. The relevant code snippet for this is shown below. With the help of the searchlevel function, we can determine if there are no available levels left in the active set. In this case, we transfer the processes in that level by setting their set values to 1 keeping in mind that level becomes the original priority level.

```
void leveldequeue(int lvl){
        // Search for processes in the same level, subtract their queue
        // Assign them to another level, Last active goes to the back of the queue
        // Check flag if the level is the last level in the set
        struct proc *xp;
        struct proc *qq;

        int result = searchlevel(lvl);

        for(int k = 0; k < NPROC; k++){
                xp = &ptable.proc[k];
                if(xp->state == RUNNING && xp->level == lvl && xp->set == 0 && xp->state != UNUSED && xp->state != ZOMBIE)
                        qq = xp;
                else if(xp->level == lvl && xp->set == 0 && xp->state != UNUSED && xp->state != ZOMBIE){
                        if(result != -1)
                                xp->level = result;
                        else{
                                xp->set = 1;
                                xp->level = xp->starting_level;
                        }
                        xp->ticks_left = RSDL_PROC_QUANTUM;
                        xp->queue = queuesort(xp->set, xp->level);
                }
        }

        if(qq){
                if(result != -1){
                        qq->level = result;
                }
                else{
                        qq->set = 1;
                        qq->level = qq->starting_level;
                }
                qq->ticks_left = RSDL_PROC_QUANTUM;
                qq->queue = queuesort(qq->set, qq->level);
        }
        return;
}
```

Figure 15: Relevant Code Snippet in leveldequeue Function for Transferring Processes from Active to Expired

13. Implementation Explanation for Swapping Sets

For the implementation of swapping sets, most of the added/modified code is inside the proc.c file. First, we need to know when we should swap the sets or to be more specific, we need to know when there are no more processes that are ready to execute in the active set. This happens inside the scheduler function in proc.c. There is first a nested for-loop that goes through the queues of each level in the active set looking for the process ready to execute with the lowest queue index. This process is stored to the variable choice, however, if no such process is found, then the value of choice stays at -1. After the nested for-loop ends, we now enter an if-statement, it checks whether the value of choice is either -1 or not. If its value is -1, then there are no more processes that are ready to execute in the active set, which means we shall check if the expired set has a process and if it has, then we shall swap the sets, hence we call the switch_set function (more on this later). This happens in a for-loop that calls the expired_check function (discussed at the auxiliary functions portion of the document) for each level in the expired set. Of course, once we have confirmed that there is a process in the expired set, we shall break out of the for-loop. Below is a code snippet that contains what has been mentioned so far:

```
667         struct proc *xp;
668         int temp = 2147483647;
669         int choice = -1;
670         for(int j = 0; j < RSDL_LEVELS; j++){
671                 for(int k = 0; k < NPROC; k++){
672                         xp = &ptable.proc[k];
673                         //Find the lowest RUNNABLE process in Queue starting from first level and should not be in the expired set
674                         if(xp->state == RUNNABLE && xp->queue < temp && xp->set != 1 && xp->level == j){
675                                 temp = xp->queue;
676                                 choice = k;
677                         }
678                 }
679                 if(choice != -1) break;
680     }
681
682         //If no RUNNABLE process are available
683         if (choice == -1){
684                 // Check if expired set has processes and switch if a process exists.
685                 for(int j = 0; j < RSDL_LEVELS; j++){
686                         int check = expired_check(j);
687                         if (check) {
688                                 release(&ptable.lock);
689                                 switch_set();
690                                 acquire(&ptable.lock);
691                                 break;
692                         }
693                 }
694                         goto done;
695
696         }
```

Figure 16. Code Snippet for Determining When to Swap Sets

Now, to explain the swapping of sets, we examine the code snippet for the switch_set function:

```
537  void switch_set(void){
538          //Keep the queue from the old active set
539          //Set the queue of all proc in the old active to -1
540          //Call queue sort for the new sort in the expired set
541          //Set all the processes set to be active
542          struct proc *pp;
543          struct qlevel *rp;
544          int maxqueue = 0;
545
546          queueclean();
547
548          for (int k = 0; k <= NPROC; k++) {
549                  pp = &ptable.proc[k];
550                  if((pp->state == SLEEPING || pp->state == RUNNING || pp->state == RUNNABLE) && pp->set == 0){
551                          if(pp->queue > maxqueue) maxqueue = pp->queue;
552                  }
553          }
554
555          for (int i = 0; i <= maxqueue; i++){
556                  for (int k = 0; k < NPROC; k++) {
557                          pp = &ptable.proc[k];
558                          if (pp->set == 0 && (pp->state != UNUSED && pp->state != ZOMBIE) && pp->queue == i){
559                                  acquire(&ptable.lock);
560                                  pp->queue = -1;
561                                  pp->set = 1;
562                                  pp->level = pp->starting_level;
563                                  release(&ptable.lock);
564                                  pp->queue = queuesort(pp->set, pp->level);
565                          }
566                  }
567          }
568
569          acquire(&ptable.lock);
570          for (int k = 0; k < NPROC; k++) {
571                          pp = &ptable.proc[k];
572                          if (pp->state != UNUSED && pp->state != ZOMBIE){
573                                  pp->ticks_left = RSDL_PROC_QUANTUM;
574                                  pp->set = 0;
575                          }
576                  }
577          release(&ptable.lock);
578
579          //Refill the quantum in plevel
580          for(int i=0;i<RSDL_LEVELS*2;i++){
581                  rp = &plevel.qlevel[i];
582                  rp->lvlquantum = RSDL_LEVEL_QUANTUM;
583          }
```

Figure 17. Code Snippet of switch_set Function

We first call the queueclean function (explained earlier in the document) to make sure that the queues in each level are in the most recent correct order. The for-loop that starts in line 548 essentially gets the process with the highest index present in any queue

in the old active set. This process's index is stored in the maxqueue variable. Next, for the nested for-loop that starts at line 555, the outer loop iterates from zero to the value in maxqueue to make sure we check all processes in the old active set. Next, we loop through all the processes and check if they are in the old active set and are not an unused or a zombie process. If they satisfy these conditions, then we move them to the expired set and set their level to the starting level. Next, we call the queuesort function for each of these processes so that we are sure that the queue in the new expired set has the correct order.

Now, we are done putting all the processes from the old active set to the old expired set. We now need to put all these processes to the new active set. This happens in the for-loop that starts in line 570, wherein it loops through all the processes and checks if they are not unused and not zombies, and if they are indeed not, we move them to the active set and replenish all their quantum. Note that the processes here are already in a queue so there is no need to check for proper ordering this time.

Lastly, swapping of sets shall replenish the level-local quantum of all the levels. Thus, starting from line 580, we enter a for-loop that iterates through all the levels, both active and expired, and replenishes their quantum.

14. Implementation Explanation for Downgrading of Process Levels

Processes will only go down a level every time its own quantum the process-local quantum is depleted as well as the level-local quantum it's on. The two conditions however are different because the process-local quantum being depleted would mean that only that process will go down a level while depletion of the level-local quantum will make all the process on that level go down a level. The relevant codes for these are found in trap.c where we have the conditional on finding out when we need to downgrade a level. The downgrade per level is found however in proc.c inside the leveldequeue function.

We already discussed the implementation for the transfer of the processes from the active to the expired processes so we will no longer discuss that in this section. We will focus on the downgrading of process levels in the active set. There are 3 conditionals shown in the snippet below that we are interested in. These codes are highlighted by a box and will be discussed one-by-one. The code in the first box is for the condition when both the process-local and the level-local quantum is depleted in which all of the processes in that level will be downgraded using the leveldequeue function to be discussed below as well as replenish the quantum of the current process since its quantum got depleted.. The second box takes into account the condition when only the level-local quantum is depleted during which the process that made it get depleted will not replenish its quantum and all of the processes on that level will be downgraded by a level using the leveldequeue function. The third condition factors out the scenario wherein only the process-local quantum gets depleted in which only that process will go down a level. During this time, we also replenish its quantum by setting the value of the ticks_left variable by the original quantum set in the rsdl.h. We downgrade it by a level adding 1 to its current level since 0 is always the highest priority level.

The code we are interested in the leveldequeue function is shown in figure 19. How this works is first we loop through all the processes and check if they are in the active set as well as if their process states are viable (i.e. not zombie and unused). We also check if the level they are on is the last level and if they are we don't do anything. The second conditional does the same but instead checks if the process isn't in the last level during which we then downgrade this process to the next level by searching for the appropriate value of level as well as sort out the queue since we will be enqueuing and dequeuing. Note that process who went down a level has their quantum replenish.

```
if (rp->lvlquantum == 0 || myproc()->ticks_left == 0){
        if (rp->lvlquantum == 0 && myproc()->level != RSDL_LEVELS - 1 && myproc()->ticks_left == 0){
                myproc()->ticks_left = RSDL_PROC_QUANTUM;
                leveldequeue(myproc()->level);
                yield();
        }
        else if (rp->lvlquantum == 0 && myproc()->level != RSDL_LEVELS - 1 && myproc()->ticks_left > 0){
                leveldequeue(myproc()->level);
                yield();
        }
        else if (rp->lvlquantum == 0 && myproc()->level == RSDL_LEVELS - 1 && myproc()->ticks_left > 0){
                leveldequeue(myproc()->level);
                yield();
        }
        else if (rp->lvlquantum != 0 && myproc()->level != RSDL_LEVELS - 1 && myproc()->ticks_left == 0){
                int result = searchlevel(myproc()->level+1);
                if(result != -1)
                        myproc()->level = result;
                else{
                        myproc()->set = 1;
                        myproc()->level = myproc()->starting_level;
                }
                myproc()->queue = queuesort(myproc()->set, myproc()->level);
                myproc()->ticks_left = RSDL_PROC_QUANTUM;
                yield();
        }
        else if (rp->lvlquantum != 0 && myproc()->level == RSDL_LEVELS - 1 && myproc()->ticks_left == 0){
                myproc()->set = 1;
                myproc()->level = myproc()->starting_level;
                myproc()->queue = queuesort(myproc()->set, myproc()->level);
                myproc()->ticks_left = RSDL_PROC_QUANTUM;
                yield();
        }
        else if (rp->lvlquantum == 0 && myproc()->level == RSDL_LEVELS - 1 && myproc()->ticks_left == 0){
                myproc()->ticks_left = RSDL_PROC_QUANTUM;
                leveldequeue(myproc()->level);
                yield();
```

Figure 18: Code Snippet for Downgrading Process Levels in trap.c

```
int result = searchlevel(lvl);

for(int k = 0; k < NPROC; k++){
        xp = &ptable.proc[k];
        if(xp->state == RUNNING && xp->level == lvl && xp->set == 0 && xp->state != UNUSED && xp->state != ZOMBIE)
                qq = xp;
        else if(xp->level == lvl && xp->set == 0 && xp->state != UNUSED && xp->state != ZOMBIE){
                if(result != -1)
                        xp->level = result;
                else{
                        xp->set = 1;
                        xp->level = xp->starting_level;
                }
                xp->ticks_left = RSDL_PROC_QUANTUM;
                xp->queue = queuesort(xp->set, xp->level);
        }
}

if(qq){
        if(result != -1){
                qq->level = result;
        }
        else{
                qq->set = 1;
                qq->level = qq->starting_level;
        }
        qq->ticks_left = RSDL_PROC_QUANTUM;
        qq->queue = queuesort(qq->set, qq->level);
```

Figure 19: Code Snippet for Downgrading Process Levels in proc.c

15. Implementation Explanation for Level-local Quantum Consumption

The implementation for local-level quantum consumption is primarily located in the trap.c file. The relevant code for this is shown in figure 20 below. As previously mentioned in the process-local quantum consumption, the program first checks whether there is a process and if its state is set to RUNNING. It also checks if the trapframe trap number is consistent with the previous user space registers before its context switches. If all of these conditions are satisfied then we know that there is a process currently running. This means that as the process runs, the quantum of the level it is currently at should be decremented / consumed. But first, we need to know which level should have its quantum consumed. Since the local-level quantum is saved in the qlevel struct, we need to make sure that the *rp->level* and *rp->set* is consistent with the current process' level and set (hence the for-loop at lines 112-117). Once we have determined the correct level, we can now decrement its level-local quantum every tick by 1. This is done via **--rp->lvlquantum** as seen in line 118.

```
111     if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER){
112           for(rp = plevel.qlevel; rp < &plevel.qlevel[RSDL_LEVELS*2]; rp++){
113                   if(rp->level == myproc()->level && rp->set == myproc()->set){
114                   //cprintf("Current Proc: PID-%d, Level-%d, Set-%d\n", myproc()->pid, myproc()->level, myproc()->set);
115                   //cprintf("Current RP: Level-%d, Set-%d\n", rp->level, rp->set);
116                           break;}
117           }
118           --rp->lvlquantum;
119           --myproc()->ticks_left;
```

Figure 20: Code Snippet for Level-Local Quantum Consumption

16. Implementation Explanation for Prio-Fork Syscall

The implementation for the Prio-Fork Syscall involves several steps. Firstly, we have to implement the correct system call as stated in the document. This is important so that both user and kernel interfaces will recognize our system call. This is done similarly to the implementation of the *schedlog* syscall. The following are the steps.
Similarly to lab exercise 5, we were required to make a schedlog syscall. As it is fairly similar, we made the schedlog syscall in the lab exercise as our base code and made changes from thereon. The following steps are very important so that the system will recognize our implemented syscalls when in both user and kernel interface. To implement the schedlog syscall there were several steps.

1. Add int priofork(int) in user.h

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

2. Add SYSCALL(priofork) is usys.S

```
35   SYSCALL(priofork)
```

3. Add #define SYS_priofork 25 in syscall.h

```
26   #define SYS_priofork  25
```

4. Add [SYS_priofork] sys_priofork, and extern int sys_priofork(void) in syscall.c

```
109   extern int sys_priofork(void);

136   [SYS_priofork] sys_priofork,
```

5. Add the code block below in sysproc.c. The codeblock takes in an integer input (returns -1 if the argument pass has failed) and return the call for priofork function in the proc.c, passing the priority level of the forked process.

```
115   int sys_priofork(void){
116           int n;
117           if(argint(0, &n) < 0 || n >= RSDL_LEVELS)
118                   return -1;
119           return priofork(n);
120           //return 0;
121   }
```

6. Add void priofork(int) in defs.h

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

7. Add the code block below in proc.c. The codeblock is the implementation of the priofork. The code has copied lines of code from the fork syscall with just a little

modification for both the starting_level and level. If the allocproc succeeds, line 256-257 replaces immediately replaces the process' level and starting_level to the passed argument respectively. This would effectively set the priority level of the process to whatever the int argument is and remembers its starting_level whenever we have a set swap. Note that we made changes to the variables compared to the original fork function.

```
245  int priofork(int n){
246          int j, pid;
247          struct proc *pfp;
248          struct proc *cur = myproc();
249
250          // Allocate process.
251          if((pfp = allocproc()) == 0){
252                  return -1;
253          }
254
255          // Allocate starting level
256          pfp->level = n;
257          pfp->starting_level = n;
258
259          // Copy process state from proc.
260          if((pfp->pgdir = copyuvm(cur->pgdir, cur->sz)) == 0){
261                  kfree(pfp->kstack);
262                  pfp->kstack = 0;
263                  pfp->state = UNUSED;
264                  return -1;
265          }
266          pfp->sz = cur->sz;
267          pfp->parent = cur;
268          *pfp->tf = *cur->tf;
269
270          // Clear %eax so that fork returns 0 in the child.
271          pfp->tf->eax = 0;
272
273          for(j = 0; j < NOFILE; j++)
274                  if(cur->ofile[j])
275                          pfp->ofile[j] = filedup(cur->ofile[j]);
276          pfp->cwd = idup(cur->cwd);
277
278          safestrcpy(pfp->name, cur->name, sizeof(cur->name));
279
280          pid = pfp->pid;
281
282          acquire(&ptable.lock);
283
284          pfp->state = RUNNABLE;
285
286          release(&ptable.lock);
287
288          return pid;
289  }
```

Figure 21: Code Snippet for Prio-Fork Syscall Implementation

VIDEO DOCUMENTATION LINK:

https://drive.google.com/file/d/1lasl4i7iWbzAsCYAY9C4nNdLQkLyQZWq/view?usp=share_link