Design Document - HW2

Student ID: B112040003
Department and Class: Computer Science 115
Name: 張景旭

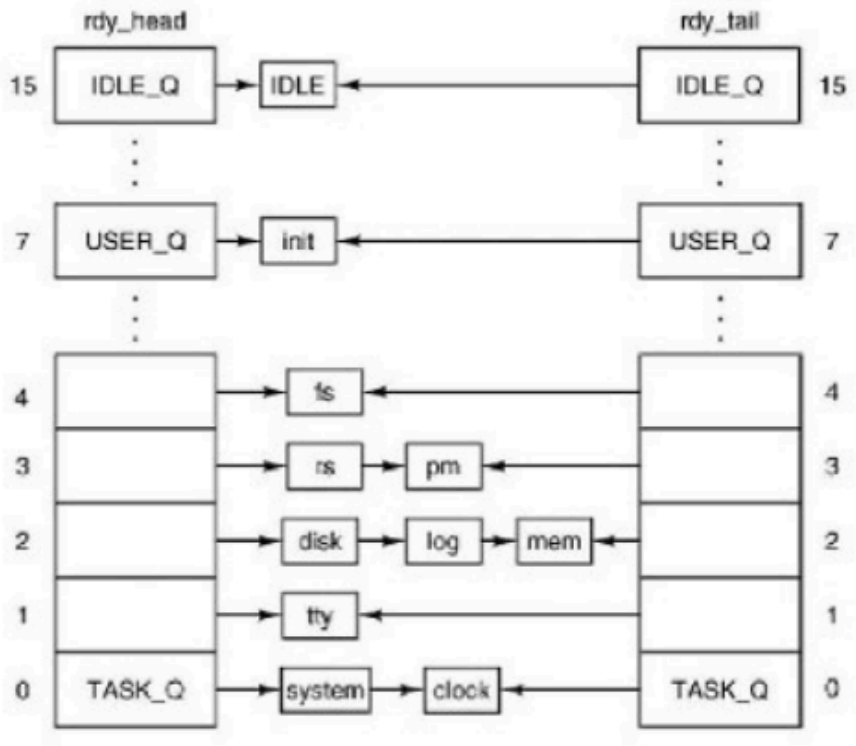
1. Overview

1. Kernel Process Table:

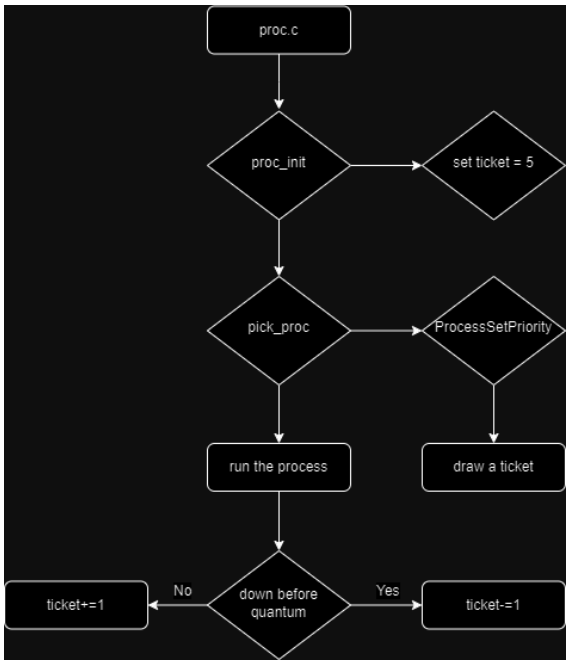
> It includes fields like p_max_priority for initial queue assignment, p_priority for dynamic queue adjustments, and p_nextready for linking processes on scheduler queues.

2. Scheduling Queues Management:

> Arrays rdy_head and rdy_tail maintain scheduling queues. The variable p_priv points to a privilege structure for IPC restrictions, with only system processes having private copies while user processes share the same copy.



How the different parts of the design interact together



Major algorithms

1. Initialization:
    - Initialize variables and data structures required for the scheduling algorithm.
    - set all the proceess ticket to five.
2. Queue Traversal and Ticket Assignment:
    - Traverse through all priority queues, assigning tickets to each process based on its priority.
    - Calculate the total number of tickets available in the system.
3. Lottery Selection:
    - Generate a random ticket number within the range of total tickets.
    - Traverse through the priority queues again, accumulating tickets until the generated ticket number is reached.
    - run the selected process
    - Adjust the priority and ticket count, if the process finish before qunatum, take a ticket away, else add a ticket unti reach the maxium.

Major data structures

> proc
> queue
> link list

2. In-depth analysis and implementation

The functions to be implemented

- ProcessSetPriority

  > return a randon number represents the ticket be drawn

The existing functions to be modified

- proc.c - void proc_init(void)

  > initailize the ticket

  ```
  1  rp->p_tickets = 5 ;
  2  rp->p_pritickets = 5 ;
  3  rp->p_runtimes = 0 ;
  ```

- main.c - static void announce(void)

  > set random base and print out the author

```
1  srandom(777);
2  printf("============================================================\n");
3  printf("Programming Assignment 2: Priority Scheduling                \n");
4  printf("Department of Computer Science and Engineering, NSYSU B112040003\n");
5  printf("============================================================\n");
```

- proc.c - static struct proc * pick_proc(void)

    1. draw a tick and check the range, if winner greater then the drawn number, run the process

```
Set tickets_num = ProcessSetPriority(total_tickets)
Set winner = 0
For q = 0 to 14:
    Set aux = rdy_head[q]
    While aux is not null:
        Set winner = winner + aux->p_tickets
        If winner > tickets_num:
            Assert(proc_is_runnable(aux))
            If priv(aux)->s_flags && BILLABLE != 0:
                run aux
            End If
            Set aux->p_runtimes to 1
            Return aux
        End If
        Set aux = aux->p_nextready
    End While
Next q
```

    2. if process have be runned before, modify lottery scheduling to decrease a process's priority by 1 each time it receives a full quantum, and increase its priority by 1

```
For q = 0 to 14:
    Set aux = rdy_head[q]
    While aux is not null:
        Set winner = winner + aux->p_tickets
        If winner >= tickets_num and setflagticket == 1:
            If aux->p_pritickets is greater than 1:
                Set aux->p_pritickets = aux->p_pritickets
                Set setflagticket = 2
        Else:
            If aux->p_pritickets < maxtickets and not proc_no_quantum(aux) :
                Set aux->p_pritickets = aux->p_pritickets + 1
                Set aux->p_runtimes = 0
            End If
            If aux->p_pritickets > mintickets and proc_no_quantum(aux):
                Set aux->p_pritickets = aux->p_pritickets - 1
                Set aux->p_runtimes = 0
            End If
        End If
        Set aux = aux->p_nextready
    End While
Next q
```

3. Risks

Stravation

> If the lottery keep choose the same process, or there might be process didn't be choosed. The correctness of such an algorithm is crucial as it directly affects the execution order of processes in the system, potentially impacting system performance and stability. If there are errors in the lottery logic, it may result in processes not being selected for execution as expected, or even lead to scheduling chaos, causing system crashes or process starvation.

4. Assignment questions

1. Basic lottery scheduling.

> **implemented in proc.c static struct proc * pick_proc(void) line 1730-1764, 1797-1814**

```
register struct proc *aux;
int total_tickets = 0;
rdy_head = get_cpulocal_var(run_q_head);


for (q=0; q <= 14; q++) {
aux = rdy_head[q];
while(aux) {

   if (aux->p_runtimes!=1) aux->p_runtimes =0;
   if ( aux->p_pritickets < 1 ){
       aux->p_tickets = 5;
       aux->p_pritickets = aux->p_tickets;
   }
   else  {
      aux->p_tickets = aux->p_pritickets;
   }
    total_tickets += aux->p_tickets;
    aux = aux->p_nextready;
}
}
winner = 0;
for (q=0; q <= 14; q++) {
    aux = rdy_head[q];
    while(aux) {
        winner += aux->p_tickets;
        if(winner >= tickets_num) {
            assert(proc_is_runnable(aux));
            if (priv(aux)->s_flags & BILLABLE){

                get_cpulocal_var(bill_ptr) = aux;
            }
            aux->p_runtimes=1;
            return aux;
        }
        aux = aux->p_nextready;
    }
}
}
```

2. Lottery scheduling with dynamic priorities

> **implemented in proc.c static struct proc * pick_proc(void) line 1766-1796**

```
int tickets_num = ProcessSetPriority(total_tickets);
int winner = 0;

setflagticket = 1;
for (q=0; q <= 14; q++) {
    aux = rdy_head[q];
    while(aux) {
        winner += aux->p_tickets;
        if(winner >= tickets_num && setflagticket == 1  ) {

            if (aux->p_pritickets > 1)
               aux->p_pritickets=aux->p_pritickets;
               setflagticket = 2;
        }
        else {
            if (aux->p_pritickets < maxtickets
                && !proc_no_quantum(aux) && aux->p_runtimes>0 ){

               aux->p_pritickets=aux->p_pritickets+1;
               aux->p_runtimes=0;
            }
            if (aux->p_pritickets > mintickets
                && proc_no_quantum(aux) && aux->p_runtimes>0 ){
               aux->p_pritickets=aux->p_pritickets-1;
               aux->p_runtimes=0;
            }
        }
        aux = aux->p_nextready;
    }
}
```

3. Profile your scheduler

> by running test1.c, we can see the order process runned

> The process in the default sceduler runs in a certain order.
> However, in the lottery scheduler, the process run in random order.

1. default scheduler(RR)

```
# ./a.out
proc1 set success
proc2 set success
proc3 set success
# prc3 heart beat 1
prc1 heart beat 1
prc2 heart beat 1
prc3 heart beat 2
prc1 heart beat 2
prc2 heart beat 2
prc3 heart beat 3
prc1 heart beat 3
prc2 heart beat 3
prc3 heart beat 4
prc1 heart beat 4
prc2 heart beat 4
prc3 heart beat 5
Change proc1 Run
prc2 heart beat 5
prc3 heart beat 6
prc1 heart beat 5
prc2 heart beat 6
prc3 heart beat 7
prc1 heart beat 6
prc2 heart beat 7
prc3 heart beat 8
prc1 heart beat 7
prc2 heart beat 8
prc3 heart beat 9
prc1 heart beat 8
Change proc3 Run
prc2 heart beat 9
prc1 heart beat 9
prc3 heart beat 10
prc2 heart beat 10
prc1 heart beat 10
prc3 heart beat 11
prc2 heart beat 11
prc1 heart beat 11
prc3 heart beat 12
prc2 heart beat 12
prc1 heart beat 12
prc3 heart beat 13
prc2 heart beat 13
prc1 heart beat 13
prc3 heart beat 14
prc2 heart beat 14
prc1 heart beat 14
prc3 heart beat 15
prc2 heart beat 15
prc1 heart beat 15
prc3 heart beat 16
```

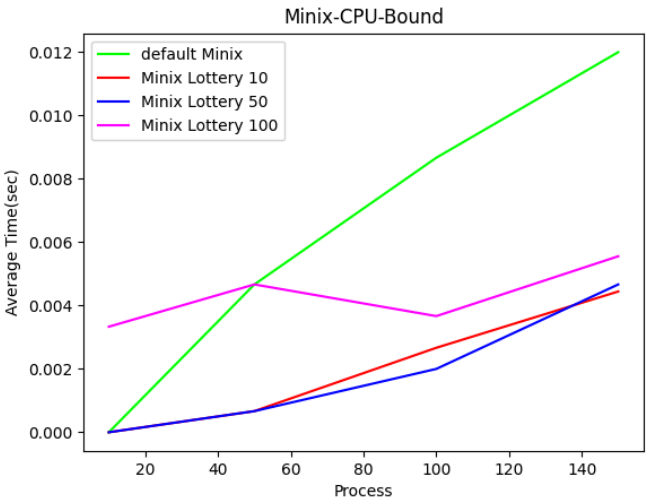
2. lottery scheduler

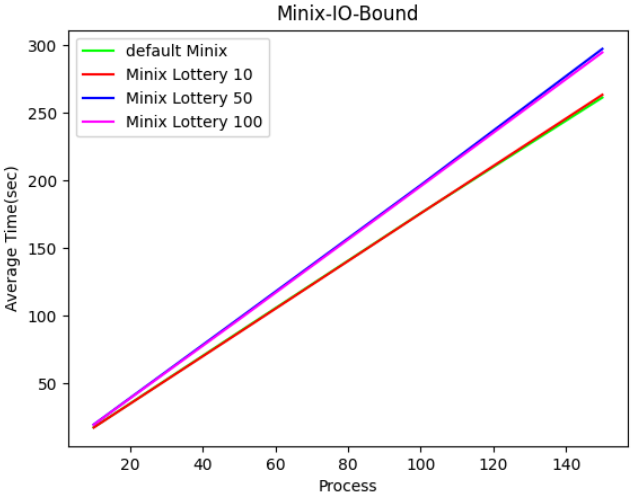
```
# ./a.out
proc1 set success
proc2 set success
proc3 set success
# prc3 heart beat 1
prc2 heart beat 1
prc1 heart beat 1
prc3 heart beat 2
prc2 heart beat 2
prc3 heart beat 3
prc1 heart beat 2
prc1 heart beat 3
prc2 heart beat 3
prc3 heart beat 4
prc2 heart beat 4
prc1 heart beat 4
prc3 heart beat 5
Change proc1 Run
prc1 heart beat 5
prc3 heart beat 6
prc2 heart beat 5
prc1 heart beat 6
prc2 heart beat 6
prc3 heart beat 7
prc1 heart beat 7
prc2 heart beat 7
prc3 heart beat 8
prc1 heart beat 8
prc2 heart beat 8
prc3 heart beat 9
Change proc3 Run
prc2 heart beat 9
prc1 heart beat 9
prc3 heart beat 10
prc1 heart beat 10
prc3 heart beat 11
prc2 heart beat 10
prc2 heart beat 11
prc3 heart beat 12
prc1 heart beat 11
prc2 heart beat 12
prc1 heart beat 12
prc3 heart beat 13
prc1 heart beat 13
prc3 heart beat 14
prc2 heart beat 13
prc2 heart beat 14
prc3 heart beat 15
prc1 heart beat 14
prc2 heart beat 15
prc3 heart beat 16
```

3. CPU-bound and IO bound

> test2.c can test how long does a process be done, and colloect the ouput of test2.c to a excel to draw the following figure.

> The following figure show the averagetime to run the IO/CPU-bound process, and the number follow by the Minix lottery means the max ticket of the lottery schedule.





> it is obvious that the performance of lottery scheduling
> is betther than the default RR scheduling, while the max ticket should not be too big, 10~50 is the best max ticket number

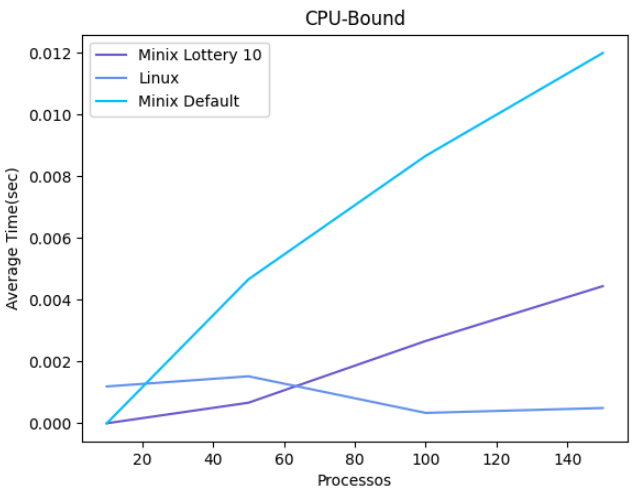
> the performance of lottery sceduling with 10 max ticket is similar to the default scheduling, while larger max ticket perform worse.

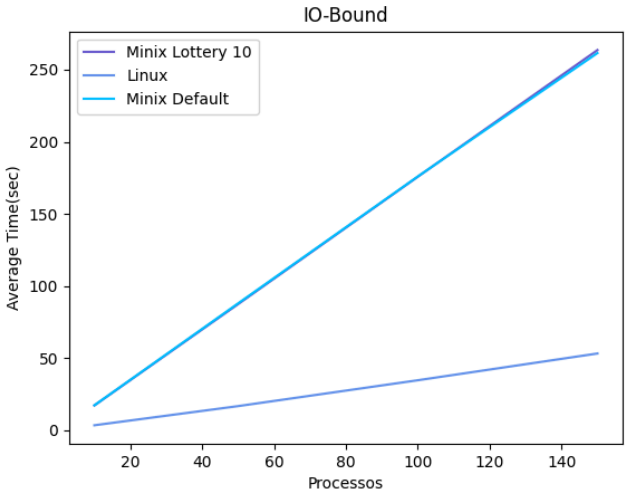
> In conculsion, the lottery scheduling with 10 max ticket perfom the best in the test result.

4. Compare to Linux

> test2.c can test how long does a process be done, and colloect the ouput of test2.c to a excel to draw the following figure.

> The following figure show the averagetime to run the IO/CPU-bound process. Respectly running in Mininx and Linix, however, the performance of linix is too good. So the time of Linix has be mutiplied by 50 to be show on the figure





> Since Linux version 2.6.23, a new scheduling mechanism called the Completely Fair Scheduler (CFS) has been introduced. It aims to allocate processor time as fairly as possible to tasks of different priorities. It utilizes a red-black tree to increase the likelihood of selecting tasks that have been executed less frequently. This helps prevent tasks with lower priorities from experiencing longer delays before being executed by the processor.

> Linux 6.6 introduces the EEVDF scheduler, finally lands Intel Shadow Stack support, adds to the Nouveau DRM driver the necessary user-space API additions for the Mesa NVK Vulkan driver, continues enabling upcoming Intel and AMD platforms, there's a boat load of other driver improvements, and also some nice performance optimizations. See the Linux 6.6 feature list for a more comprehensive overview of all the great changes in Linux 6.6