Design Document

作者 or 組員：陳育霖 B113040052

目標：將 minix 的調度器修改為 Lottery Scheduling

修改的檔案有以下(Kernel 或 User 二選一)：

Kernel：Proc.c　Proc.h　User：schedule.c　schedproc.h　sys_schedule.c

檔案路徑如下：

Proc.c 與 Proc.h	/usr/src/minix/kernel
schedule.c 與 schedproc.h	/usr/src/minix/servers/sched
sys_schedule.c	/usr/src/minix/lib/libsys

要實施的功能：

1. 每個 Process 預設為 5 個 ticket。
2. 檢查 Process 是否在時間到達前就遭 block，如果是，則多加一張 ticket 如果否，則減少一張。每個 Process 最多 10 張，最少 1 張。
3. 每次隨機 random 一個數字，來決定執行哪個 Process。
4. 每次 interrupt 時，計算當下 Process 數量，並在 interrupt 100 次時，算出平均 Process 數量。

測試檔案有：

test.c　index.c　// test 為測試的 Process、index 為呼叫多個 test 的程式

編譯步驟如下：

cd /usr/src/minix/kernel

make clean

make kernel

cp /usr/sbin/kernel /boot/minix_latest/kernel

reboot

cd /usr/benchmarks/unixbench/pgms

./spawn 1000

透過 SSH 連接 minix 後輸入 top

測試步驟如下(以 Kernel 為主)：

clang test.c -o test

clang index.c -o index

透過 SSH 連接 minix 後輸入 top

./index　// 在 minix 內輸入，即可透過 top 觀察到 test1 ~ test10 的 Pri 狀態

測試結果(以 Kernel 為主)：

當被抽到時，Priority 減 1，當執行時間完成時，Priority 加 1。

結果：一開始 Process 落在 7~8，後面慢慢遞增到 15

當被抽到時，Priority 減 2，當執行時間完成時，Priority 加 1。

結果：一開始 Process 落在 7~8，後面慢慢遞增到 15 (速度較前者慢了些)

結論：以上過程中，每個 Process 的 Priority 都非常接近(只差 1 ~ 2)，導致每個任務完成的時間也非常接近。

Compare to Linux

Linux 的調度器預設為 Completely Fair Scheduler (CFS)，且 Linux 也有專門針對互動性和響應性進行最佳化的調度器，例如 Brain Fuck Scheduler（BFS）。

CFS 使用紅黑樹來實作，演算法時間複雜度為$O(\log n)$，而 Lottery Scheduling 則是使用隨機亂數，並配合 for 迴圈，從中找到被抽中的 Process，時間複雜度為$O(n)$。

CFS 的優先度是使用累積虛擬運行時間（vruntime）來找下一個 Process，對 CPU 的資源分配較為公平；而 Lottery 調度器則為隨機抽取，被抽取越多次，則再被抽中的機率較低，此演算法也可以較公平的分配 CPU 資源，但還是有可能發生極端的事件(連續抽中同個 Process)。

選擇 Kernel 要修改的部分為：

Proc.h

```
struct proc {
   int p_tickets;    // 記錄每個 Process 的 ticket
   ...   // 其他程式碼
}
```

Proc.c

```
int process_sum = 0;    //   紀錄全部 Process 的 Priority
void record_process_priority() {
        register struct proc *rp;                /* process to run */
        struct proc **rdy_head;
        int q;                        /* iterate over queues */
        int total_tickets = 0, lottery = 0;
        for (q=0; q < NR_SCHED_QUEUES; q++) {
        if(!(rp = rdy_head[q])) {
                TRACE(VF_PICKPROC, printf("cpu %d queue %d empty\n", cpuid, q););
                continue;
        }
        while(rp != 0) {
                        process_sum += rp -> p_priority;    // 加總優先度
                        rp = rp -> p_nextready;
        }
        }
}
int timer_interrupt_count = 0;    //   紀錄 interrupt 幾次
void timer_interrupt_handler() {
        record_process_priority();    //   呼叫計算優先度的函式
        timer_interrupt_count++;    //   增加 interrupt 次數
        if (timer_interrupt_count % 100 == 0) {    //   每 100 次執行一次
                timer_interrupt_count = 0;
                printf("%.2f\n", (float)process_sum / 100.0);
        }
}
```

```
void ProcessSetPriority(struct proc *p, int n) {    //   設定 ticket 數量的函式
        p -> p_tickets += n;
}

void proc_init(void)
{
        ...   // 其他程式碼
        for (rp = BEG_PROC_ADDR, i = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++i) {
                rp->p_rts_flags = RTS_SLOT_FREE;/* initialize free slot */
                rp->p_magic = PMAGIC;
                rp->p_nr = i;                   /* proc number from ptr */
                rp->p_endpoint = _ENDPOINT(0, rp->p_nr); /* generation no. 0 */
                rp->p_scheduler = NULL;                 /* no user space scheduler */
                rp->p_priority = 0;             /* no priority */
                rp->p_quantum_size_ms = 0; /* no quantum size */
                ProcessSetPriority(rp, 5);        /*初始 5 張*/
                /* arch-specific initialization */
                arch_proc_reset(rp);
}
```

```
static struct proc * pick_proc(void)
{
    ... // 其他程式碼
    int total_tickets = 0, lottery = 0;    // 紀錄總 ticket 數量與抽到的數字
    for (q=0; q < NR_SCHED_QUEUES; q++) {
        ... // 其他程式碼
        while(rp != 0) {    // 計算 ticket 的總和
                    total_tickets += rp -> p_tickets;
                    rp = rp -> p_nextready;
        }
}
lottery = 1 + random() % total_tickets;    // 從中隨機抽一個 ticket
    total_tickets = 0;
    rdy_head = get_cpulocal_var(run_q_head);
    ... // 其他程式碼
        while(rp != 0) {    // 尋找 lottery 落在哪個 Process 之間
                    total_tickets += rp -> p_tickets;
                    if(total_tickets >= lottery)
                        break;
                    rp = rp -> p_nextready;
        }
        rp->p_priority--;    // 抽到的 Process 優先度增加
        assert(proc_is_runnable(rp));
        if (priv(rp)->s_flags & BILLABLE)
            get_cpulocal_var(bill_ptr) = rp; /* bill for system time */
        return rp;    // 回傳抽中的 Process
    }
    return NULL;
}
```

```
not_runnable_pick_new:
if (proc_is_preempted(p)) {    // 如果 Process 被占用
        p->p_rts_flags &= ~RTS_PREEMPTED;
        if (proc_is_runnable(p)) {    // 而 Process 還可運行
            if (p->p_cpu_time_left) {    // 以及 Process 還有時間
                if (p -> p_tickets < 10) // 且 Process 的 ticket 未超過 10 張
                    p_tickets++;
                enqueue_head(p);
            }
            else
                enqueue(p);
        }
}
void proc_no_time(struct proc * p)
{
    if (!proc_kernel_scheduler(p) && priv(p)->s_flags & PREEMPTIBLE) {
    // 當 Process 因時間到而結束時
        if(p -> p_tickets > 2) {    // 當 Process 完成前沒有被 Block
            p -> p_priority++;    //  優先度減少
            ProcessSetPriority(p, -1);    // 且 ticket 大於 2 時，會少一張
        }    /* this dequeues the process */
        notify_scheduler(p);
    }
    else {
        p->p_cpu_time_left = ms_2_cpu_time(p->p_quantum_size_ms);
#if DEBUG_RACE
        RTS_SET(p, RTS_PREEMPTED);
        RTS_UNSET(p, RTS_PREEMPTED);
#endif
    }
}
```

```
void record_process_priority() {    // 每次 interrupt 時，計算當下的 Process 數量
        register struct proc *rp;                /* process to run */
        struct proc **rdy_head;
        int q;                        /* iterate over queues */
        int total_tickets = 0, lottery = 0;
        for (q=0; q < NR_SCHED_QUEUES; q++) {
                if(!(rp = rdy_head[q])) {
                        TRACE(VF_PICKPROC, printf("cpu %d queue %d empty\n", cpuid, q););
                        continue;
                }
                while(rp != 0) {    // 當 rp 不是空的或無
                                process_sum++;    // Process 的數量+1
                                rp = rp -> p_nextready;
                }
        }
}

int timer_interrupt_count = 0;    // 紀錄 interrupt 次數

void timer_interrupt_handler() {    // 每次 interrupt 時，呼叫
        record_process_priority();
        timer_interrupt_count++;
        if (timer_interrupt_count % 100 == 0) {    // 當 interrupt 滿 100 次時
                timer_interrupt_count = 0;
                printf("%.2f\n", (float)process_sum / 100.0);    // 輸出平均 Process 數量
        }
}
```

選擇 User 要修改的部分為：

schedproc.h

```
EXTERN struct schedproc {    // 在 Process 的結構中增加 ticket...等
   ... // 其他程式碼
   unsigned tickets;
   unsigned torpil;
   unsigned is_sys_proc;
   unsigned cpu;
   ... // 其他程式碼
}
```

sys_schedule.c

```
#include "syslib.h"

int sys_schedule(endpoint_t proc_ep, int priority, int quantum, int
cpu, int niced)
{
        message m;
        m.m_lsys_krn_schedule.endpoint = proc_ep;
        m.m_lsys_krn_schedule.priority = priority;
        m.m_lsys_krn_schedule.quantum  = quantum;
        m.m_lsys_krn_schedule.cpu = cpu;
        m.m_lsys_krn_schedule.niced = niced;
        return(_kernel_call(SYS_SCHEDULE, &m));
}
```

schedule.c

```
int DYNAMIC = 0, MAX_TICKETS = 10, MIN_TICKETS = 1;    // 紀錄邊界值
void ProcessSetPriority(register struct schedproc *rmp, int n) {    // 更新 ticket 的值
    rmp += n;
}

int lottery(){
    struct schedproc *rmp;
    int winner; int proc_nr; int total_tickets = 0, int rv;
    for(proc_nr = 0, rmp = schedproc; proc_nr < NR_PROCS; proc_nr++,rmp++){
        if(rmp->torpil == 1 && rmp->flags & IN_USE ) {
            rmp->priority = 14;    // 將 process 的 priority 設為 14
            schedule_process_local(rmp);
        }
        if((rmp->torpil == 0 && rmp->flags & IN_USE && rmp->is_sys_proc != 1 ) && rmp-
>priority == 13 ){
            total_tickets += rmp->tickets;    // 計算總 tickets 數量
        }
    }
    printf("%d\n", total_tickets);
        if(total_tickets > 0){
            winner = random()%total_tickets + 1 ;    // 隨機抽取 Process
            for(proc_nr =0, rmp = schedproc; proc_nr <NR_PROCS; proc_nr++,rmp++){
                if(rmp->flags & IN_USE && rmp->torpil == 1){
                    rmp->priority = 14;    // 將之前抽到的 Process 的 Priority 改回 14
                }
        else if((rmp->flags & IN_USE && rmp->is_sys_proc != 1) && rmp->priority == 13){
                if(winner > 0){
                    winner -= rmp->tickets;
                    if(winner <= 0){    // 找到被抽到的 Process
                        rmp->priority--;    // 提高被抽到的 Process 的 Priority
                        if((rv = schedule_process_local(rmp))!=OK){
                            printf("error while scheduling...\n");
```

```
                    return rv;
                    }
                }
                }
            }
        }
    }
    return OK;
}
```