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。

1. 每次隨機random一個數字，來決定執行哪個Process。
2. 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;

}