**OS HW04 GROUP 18**

**Part 1:Implementation Detail**

1. **Detail**

**threads/thread.h**

public:

void setBurstTime(double t) {burstTime = t;}

void setWaitingTime(double t) {waitingTime = t;}

void setExecutionTime(double t) {executionTime = t;}

void setPriority(double p) {priority = p;}

void setL3Time(double t){L3Time = t;}

void setStartexcution(double t){startexcution = t;}

double getBurstTime(){return (burstTime);}

double getWaitingTime(){return (waitingTime);}

double getExecutionTime(){return (executionTime);}

double getPriority(){return (priority);}

double getL3Time(){return (L3Time);}

double getStartexcution(){return startexcution;}

int aging\_flag;

int initial\_priority;

int first\_exe;

private:

double burstTime;

double waitingTime;

double executionTime;

double L3Time;

double startexcution;

在 Thread Class 裡面新增一些新的參數，用以做 process scheduling

burstTime : 用來記錄 thread 的 approximate burst Time 結果。

waitingTime : 用來記錄 thread 在 ready queue 裡等待的時間，用來做 aging。

executionTime : 用來記錄 thread 從 running 到 waiting 花費多少時間。

L3Time : 用來設定 RR 的 TQ。

startexcution : 用來記錄 thread 開始的時間點，與現在的執行時間相減即可獲得 executionTime。

而隨之的 function 即為這些 parameter 的setting function 以及 get value function。

而還有定義一些其他的參數用以實作 aging 和 update Approximation Burst Time

Aging\_flag : 若 thread 有因為 aging 的機制增加 priority 則此 flag = 1

Initial\_priority : aging 過的 thread 若被執行過了，則會回復到最初設定的 priority

first\_exe : 看助教提供的 log 檔裡面好像 thread 第一次 content switch 也會更新 approximation burstTime，故設定此 flag。

**threads/thread.cc**

void

Thread::Sleep (bool finishing){

Thread \*nextThread;

ASSERT(this == kernel->currentThread);

ASSERT(kernel->interrupt->getLevel() == IntOff);

DEBUG(dbgThread, "Sleeping thread: " << name);

status = BLOCKED;

if(kernel->stats->mainTicks == 0){

kernel->stats->mainTicks = kernel->stats->totalTicks;

this->setExecutionTime(kernel->stats->mainTicks);

}

else if(this->first\_exe==0){

kernel->scheduler->ApproximateBurstTime(this);

}

while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {

kernel->scheduler->ReadyToRun(kernel->currentThread);

kernel->interrupt->Idle(); // no one to run, wait for an interrupt

}

if(kernel->currentThread->getID() == 0 && finishing == 1){

}

kernel->scheduler->Run(nextThread, finishing);

}

在此 if(kernel->stats->mainTicks == 0) 中我們使用mainTicks 用來記錄一開始 main function 所執行的時間，當 main function 執行到一段落，我們會設定 main function 的 executionTime，用來做 DeBug 的輸出。

而若 if 不成立，即為 main function 的部分處理完，開始進入 thread 的 instruction，在這邊雖然助教說當 running 到 waiting 時才會更新 Approximation Burst Time，但是我們觀察到 log 檔裡第一次 switch 也會更新故在此利用 first\_exe 判斷是否要進入 *ApproximateBurstTime()* 來進行 burst time 的更新。

**threads/scheduler.h**

class Scheduler {

……

public:

void updatePriority();

void ApproximateBurstTime(Thread\* thread);

private:

SortedList<Thread \*> \*L2ReadyList;

SortedList<Thread \*> \*L1ReadyList;

List<Thread \*> \*L3ReadyList;

…….

}

在class Scheduler 裡新增了 *updatePriority()* 和 *ApproximateBurstTime()* 用來實作 aging 的 update priority 和計算 Approximation burst tim，以及因題目要求的L1，L2，L3 ReadyQueue，其中因 L3 為 RR 故不須使用 SortedList。

**threads/scheduler.cc**

設定 compareL1 , compareL2 function，用來給 SortedList 吃的 function pointer，可以決定如何 insert element 到 queue 裡面。

static int compareL2(Thread \*t1, Thread \*t2){

if(t1->getPriority()> t2->getPriority())

return -1;

else if(t1->getPriority() < t2->getPriority())

return 1;

else

return t1->getID() < t2->getID() ? -1:1;

return 0;

}

static int compareL1(Thread \*t1, Thread \*t2){

if(t1->getBurstTime()> t2->getBurstTime())

return 1;

else if(t1->getBurstTime() < t2->getBurstTime())

return -1;

else

return t1->getID() < t2->getID() ? -1:1;

return 0;

}

如果是compareL2，則 priority 比較大的 thread 會排在 queue 比較前面的地方，即可實現所要的 priority algorithm。

如果是compareL1，則 burst time 比較小的 thread 會排在 queue 比較前面的地方，即可實現所要的 SJF algorithm。

而L3 因為為 RR 故不用額外設定 function。

Scheduler::Scheduler()

{

L2ReadyList = new SortedList<Thread \*>(compareL2);

L1ReadyList = new SortedList<Thread \*>(compareL1);

L3ReadyList = new List<Thread \*>;

toBeDestroyed = NULL;

}

Scheduler::~Scheduler()

{

delete L2ReadyList;

delete L1ReadyList;

delete L3ReadyList;

}

利用 scheduler 的 constructor 初始化 List，和利用 Destructor 釋放記憶體。

void Scheduler::updatePriority()

{

ListIterator<Thread \*> \*iter1 = new ListIterator<Thread\*>(L1ReadyList);

ListIterator<Thread \*> \*iter2 = new ListIterator<Thread\*>(L2ReadyList);

ListIterator<Thread \*> \*iter3 = new ListIterator<Thread\*>(L3ReadyList);

Statistics \*stats = kernel->stats;

int oldPriority;

int newPriority;

for(;!iter1->IsDone();iter1->Next())

{

ASSERT(iter1->Item()->getStatus()==READY);

iter1->Item()->setWaitingTime(iter1->Item()->getWaitingTime()+TimerTicks);

if(iter1->Item()->getWaitingTime() >=1500 && iter1->Item()->getID()>0)

{

oldPriority = iter1->Item()->getPriority();

newPriority = oldPriority + 10;

DEBUG('z',"[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << iter1->Item()->getID()<< "] changes its priority from [" << oldPriority << "] to ["<< newPriority << "]");

if(newPriority>149)

{

newPriority = 149;

}

iter1->Item()->aging\_flag = 1;

iter1->Item()->setPriority(newPriority);

iter1->Item()->setWaitingTime(0);

}

}

for(;!iter2->IsDone();iter2->Next())

{

ASSERT(iter2->Item()->getStatus()==READY);

iter2->Item()->setWaitingTime(iter2->Item()->getWaitingTime()+TimerTicks);

if(iter2->Item()->getWaitingTime() >=1500 && iter2->Item()->getID()>0)

        {

            oldPriority = iter2->Item()->getPriority();

            newPriority = oldPriority + 10;

            DEBUG('z',"[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << iter2->Item()->getID()<< "] changes its priority from [" << oldPriority << "] to ["<< newPriority << "]");

            if(newPriority>149)

            {

                newPriority = 149;

            }

            iter2->Item()->setPriority(newPriority);

            iter2->Item()->aging\_flag = 1;

            iter2->Item()->setWaitingTime(0);

            L2ReadyList->Remove(iter2->Item());

            ReadyToRun(iter2->Item());

        }

    }

    for(;!iter3->IsDone();iter3->Next())

    {

        ASSERT(iter3->Item()->getStatus()==READY);

        iter3->Item()->setWaitingTime(iter3->Item()->getWaitingTime()+TimerTicks);

        if(iter3->Item()->getWaitingTime() >=1500 && iter3->Item()->getID()>0)

        {

            oldPriority = iter3->Item()->getPriority();

            newPriority = oldPriority + 10;

            DEBUG('z',"[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << iter3->Item()->getID()<< "] changes its priority from [" << oldPriority << "] to ["<< newPriority << "]");

            if(newPriority>149)

            {

                newPriority = 149;

            }

            iter3->Item()->setPriority(newPriority);

            iter3->Item()->aging\_flag = 1;

            iter3->Item()->setWaitingTime(0);

            Thread\* tmpThread = iter3->Item();

            L3ReadyList->Remove(tmpThread);

            ReadyToRun(tmpThread);

        }

    }

}

在 *updatePriority()* 裡會實作 aging 的機制，首先我們會建立各個 queue 的 iterator，用來遍歷所有 queue 裡的 elements，我們會分成 L1 , L2 , L3 來做，但三個 queue 的實作方法都相同，首先因為我們的會先累加各個 thread 的 waitingTime，其中因為我們為每次 Alarm Timer 會呼叫一次 *updatePriority()*，故每次累加 TimerTicks，然後若 waitingTime 超過 1500 則會進行 aging 將 priority 加 10，然後將 agin\_flag 設為 1，並將 waitingTime 設為 0 使其重新累加 aging 的時間，接著 L1 和 L2 & L3 有著小小的不同，L2 因為 priority algorithm，故 thread 的 priority 改變後會將其拿出 List 中，並依據現有的 priority 重新排列( *ReadyToRun()* )，L3 則是有可能會上升到 L2 queue 中，故也要和 L2 的寫法一樣，然後在 L1 時，因為為 SJF 故改變 priority 部會影響 queue 中的排序。

void

Scheduler::ReadyToRun (Thread \*thread){

ASSERT(kernel->interrupt->getLevel() == IntOff);

DEBUG(dbgThread, "Putting thread on ready list: " << thread->getID());

thread->setStatus(READY);

if(thread->getPriority() >= 100 && thread->getPriority() <= 149){

if(!L1ReadyList->IsInList(thread)){

DEBUG('z' , "[A] Tick [" << kernel->stats->totalTicks << "]:Thread [" << thread->getID()<< "] is inserted into L1 queue");

L1ReadyList->Insert(thread);

}

}

else if(thread->getPriority() >= 50 && thread->getPriority() <= 99){

if(!L2ReadyList->IsInList(thread)){

DEBUG('z' , "[A] Tick [" << kernel->stats->totalTicks << "]:Thread [" << thread->getID()<< "] is inserted into L2 queue");

L2ReadyList->Insert(thread);

}

}

else{

if(!L3ReadyList->IsInList(thread))

{

DEBUG('z' , "[A] Tick [" << kernel->stats->totalTicks << "]:Thread [" << thread->getID()<< "] is inserted into L3 queue");

L3ReadyList->Append(thread);

}

}

}

**而***ReadytoRun()*，根據thread 不同的 priority 分配到不同的 queue L1，L2，L3，然後根據作業要求的 scheduling algorithm做Insert或Append的動作，同時也把state 設成 Ready。

void Scheduler::ApproximateBurstTime(Thread \*thread){

if(kernel->stats->mainTicks != 0){

double prevBurstTime = thread->getBurstTime();

thread->setExecutionTime( (kernel->stats->totalTicks) - (thread->getStartexcution()));

double newBurstTime = 0.5 \* prevBurstTime + 0.5 \* thread->getExecutionTime();

thread->setBurstTime(newBurstTime);

DEBUG('z',"[D] Tick [" << kernel->stats->totalTicks << "]: Thread ["<< thread->getID()

<< "] update approximate burst time, from: [" << prevBurstTime

<< "], add [" << thread->getExecutionTime() <<"], to ["<< newBurstTime <<"]" );

}

}

而這邊是我們定義出來計算Approximate Burst Time的function，我們按照功課提供的算式，0.5\* 先前的burst time + 0.5\*預測的burst time來算出新的burst time。

Thread \*

Scheduler::FindNextToRun (){

ASSERT(kernel->interrupt->getLevel() == IntOff);

if (!L1ReadyList->IsEmpty()){

Thread \*a = L1ReadyList->RemoveFront();

if(a->aging\_flag == 1){

a->setPriority(a->initial\_priority);

}

a->setStartexcution(kernel->stats->totalTicks);

DEBUG('z',"[B] Tick [" << kernel->stats->totalTicks << "]:Thread [" << a->getID()<< "] is removed from L1 queue");

return a;

}

else if(!L2ReadyList->IsEmpty()){

Thread \*a = L2ReadyList->RemoveFront();

if(a->aging\_flag == 1){

a->setPriority(a->initial\_priority);

}

DEBUG('z',"[B] Tick [" << kernel->stats->totalTicks << "]:Thread [" << a->getID()<< "] is removed from L2 queue");

return a;

}

else if(!L3ReadyList->IsEmpty()){

Thread \*a = L3ReadyList->RemoveFront();

if(a->aging\_flag == 1){

a->setPriority(a->initial\_priority);

}

DEBUG('z',"[B] Tick [" << kernel->stats->totalTicks << "]:Thread [" << a->getID()<< "] is removed from L3 queue");

return a;

}

else{

return NULL;

}

}

這裡的 FindNextToRun() 是在準備從 ready queue 挑選一個要執行的 thread 時，而這邊我們會判定他是否有被 aging 過 (aging\_flag == 1)，若挑到被 aging 過的 thread 則會將此 thread 的 priority 設定為原本 command 設定的 initial priority。

void

Thread::Yield (){

Thread \*nextThread;

IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

ASSERT(this == kernel->currentThread);

DEBUG(dbgThread, "Yielding thread: " << name);

kernel->scheduler->ReadyToRun(this);

nextThread = kernel->scheduler->FindNextToRun();

if(nextThread != NULL){

kernel->scheduler->Run(nextThread, FALSE);

}

(void) kernel->interrupt->SetLevel(oldLevel);

}

而這裡是將 **ReadyToRun()** 從迴圈中移出，放到 **FindNextToRun()** 之前，這樣可以避免在某些情況下發生 priority 比較高的 process 比 priority 低的 process 晚跑的情形。換句話說，這樣的調整能確保 process 的執行順序更加符合預期的優先權設定。

void

Alarm::CallBack()

{

Interrupt \*interrupt = kernel->interrupt;

MachineStatus status = interrupt->getStatus();

kernel->scheduler->updatePriority();

Thread \*thread = kernel->currentThread;

thread->setExecutionTime(thread->getExecutionTime()+ TimerTicks);

thread->setL3Time(thread->getL3Time()+ TimerTicks);

if(kernel->currentThread->getID()>0 && status != IdleMode && kernel->currentThread->getPriority() >=100)

{

interrupt->YieldOnReturn();

}

if (status != IdleMode && kernel->currentThread->getPriority() < 50) {

if(kernel->currentThread->getL3Time()>= 99)

interrupt->YieldOnReturn();

}

}

在每個 Timer Interrupt 之間進行 aging 的判斷、execution time 的累積，以及 L1 或 L3 的 preemptive 判斷。因此，這樣的設計能確保系統在中斷週期內動態管理 process 的優先級與執行時間，同時符合設計的 preemptive 行為。

**machine/console.cc**

void ConsoleOutput::PutInt(int value) {

ASSERT(putBusy == FALSE);

Thread \*thread = kernel->currentThread;

char \* printStr = (char\*)malloc(sizeof(char) \* 15);

sprintf(printStr, "%d\n\0", value);

WriteFile(writeFileNo, printStr, strlen(printStr) \* sizeof(char));

putBusy = TRUE;

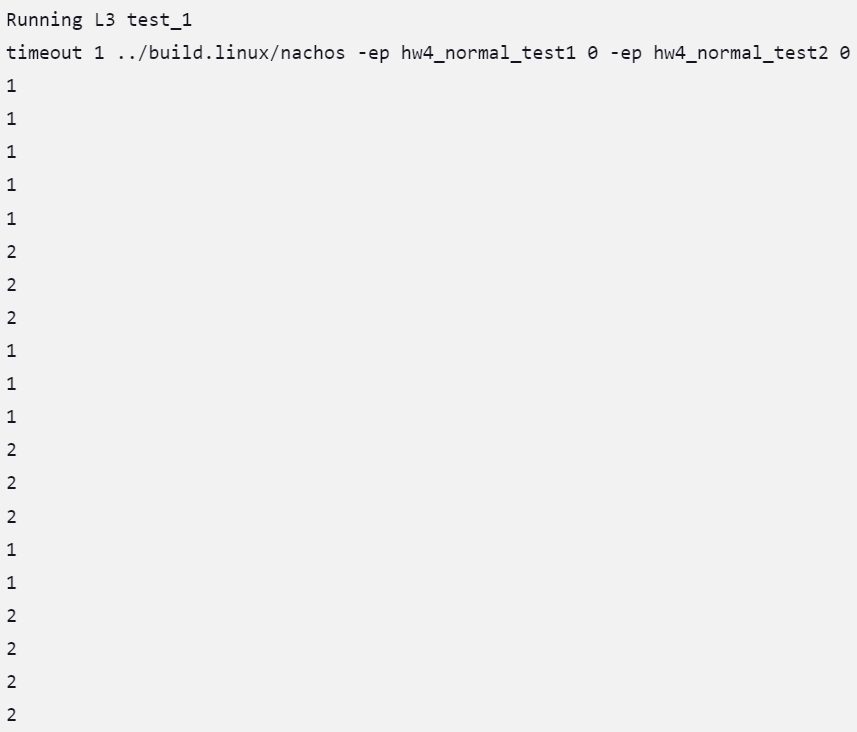
kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);

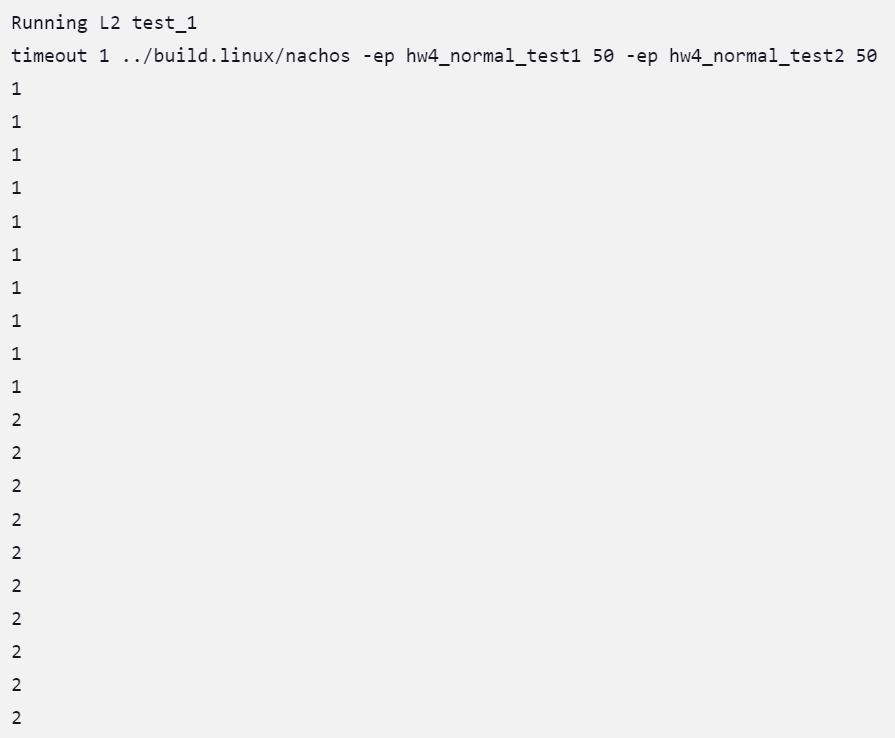
kernel->scheduler->ApproximateBurstTime(thread);

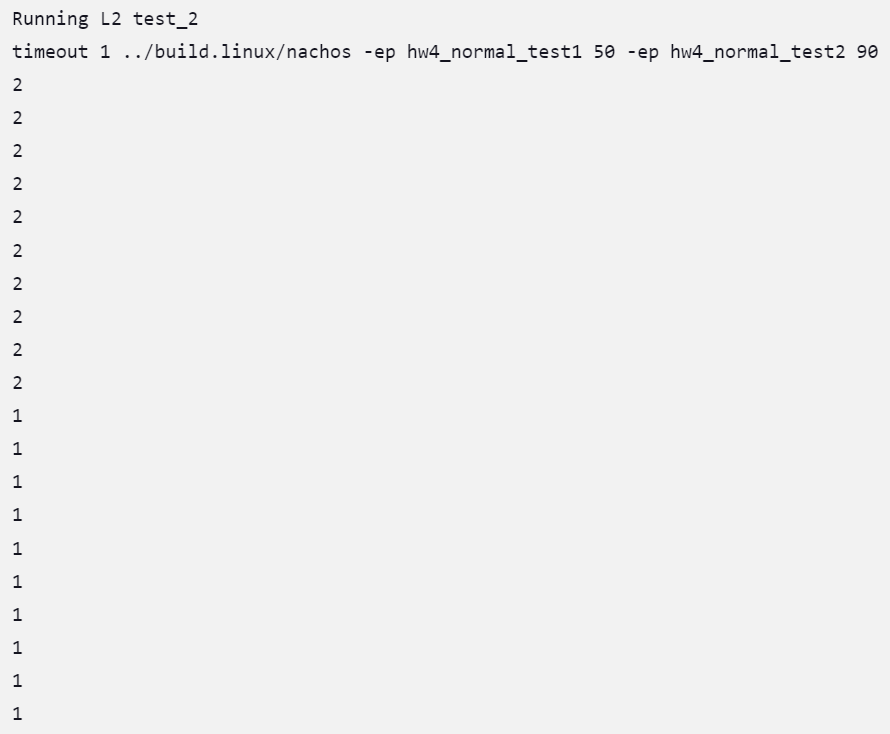
}

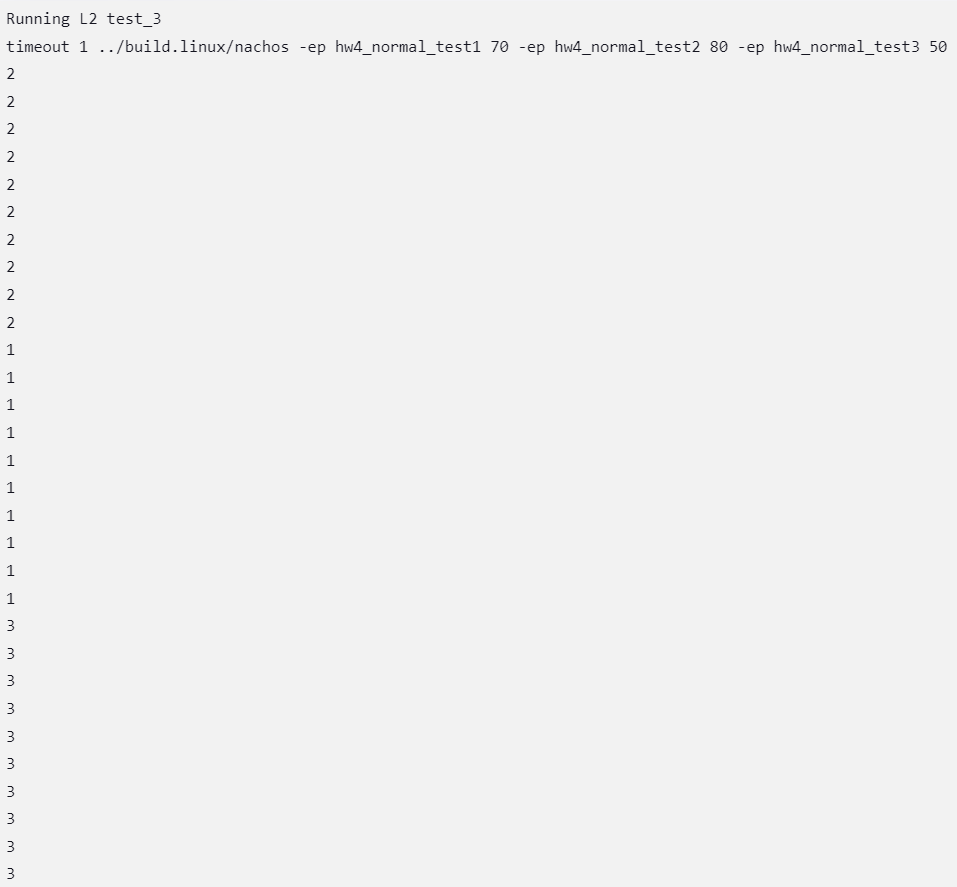
因為作業要求為 running 到 waiting 時要 update Approximation burst time，故在輸出 PutInt() 時跑 *ApproximateBurstTime()* function。

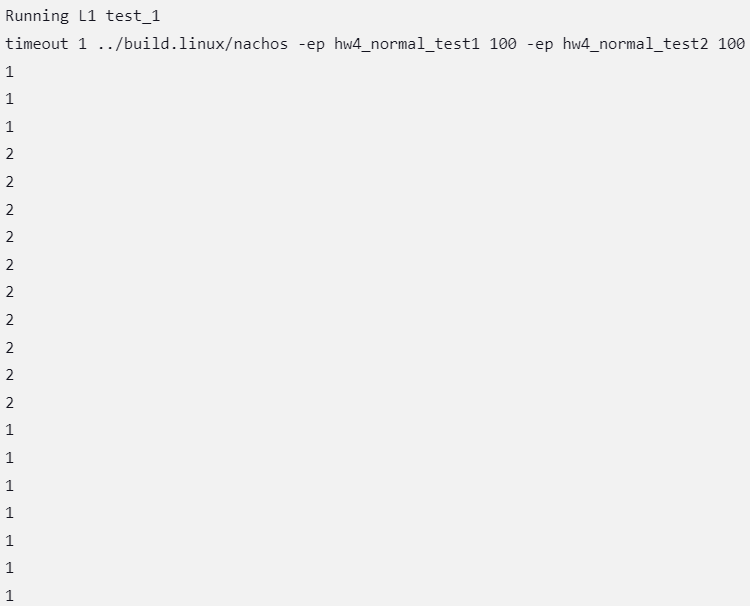
**(b) result**

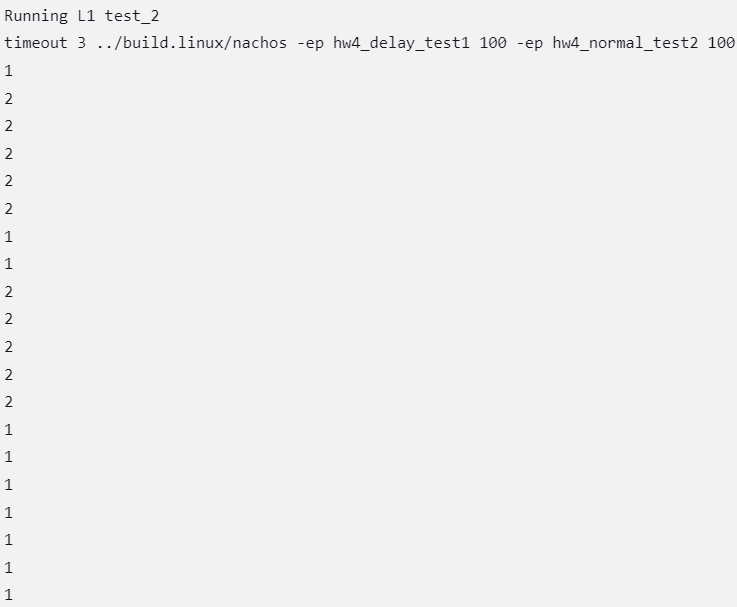
****

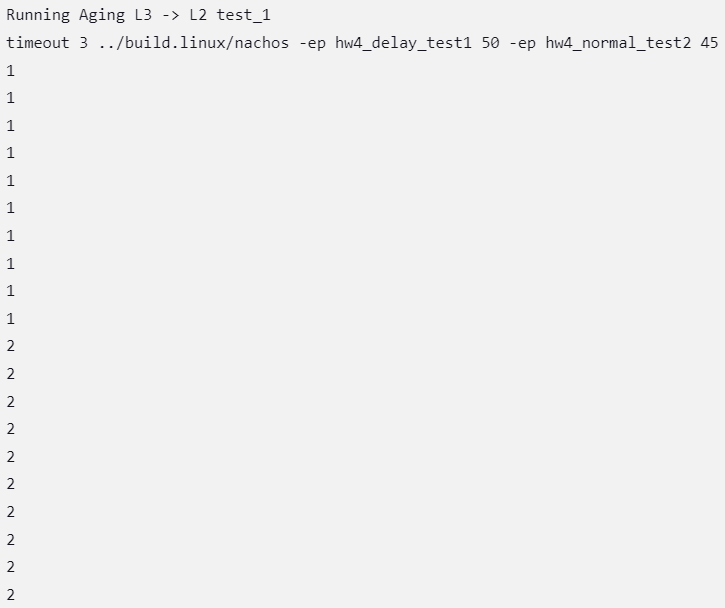
****

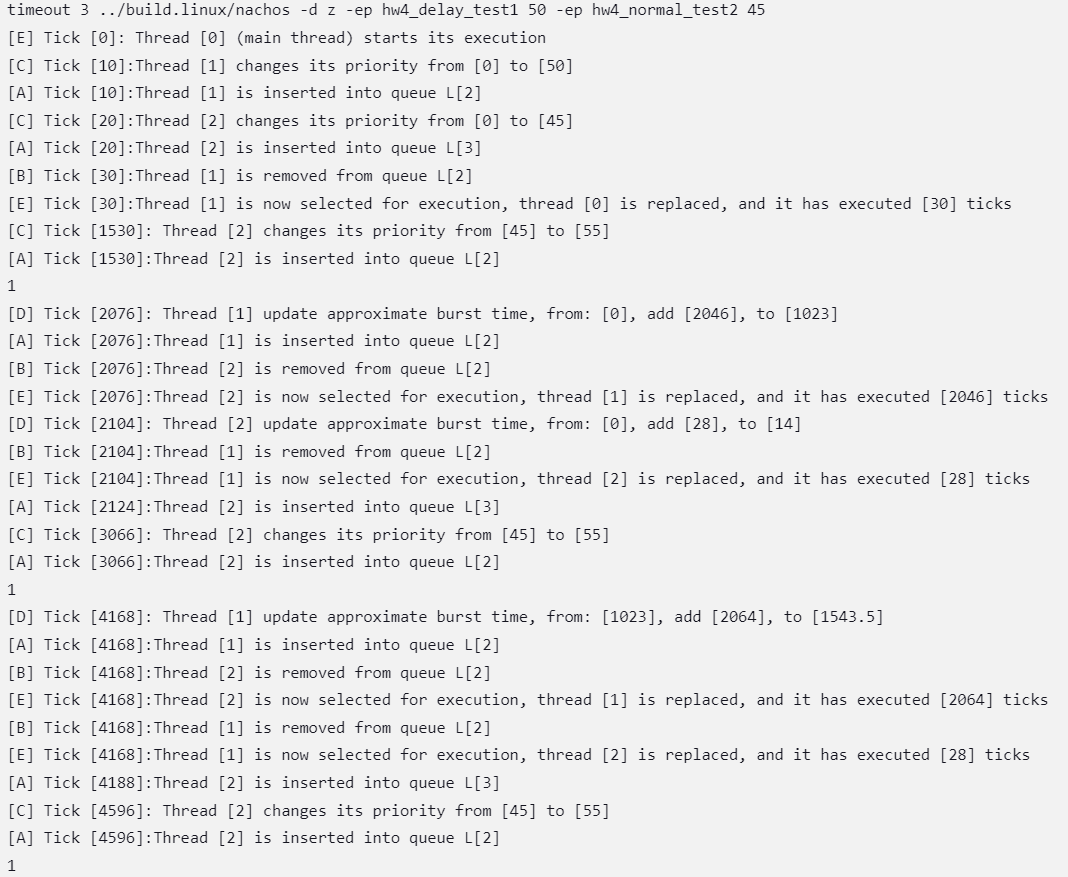
****

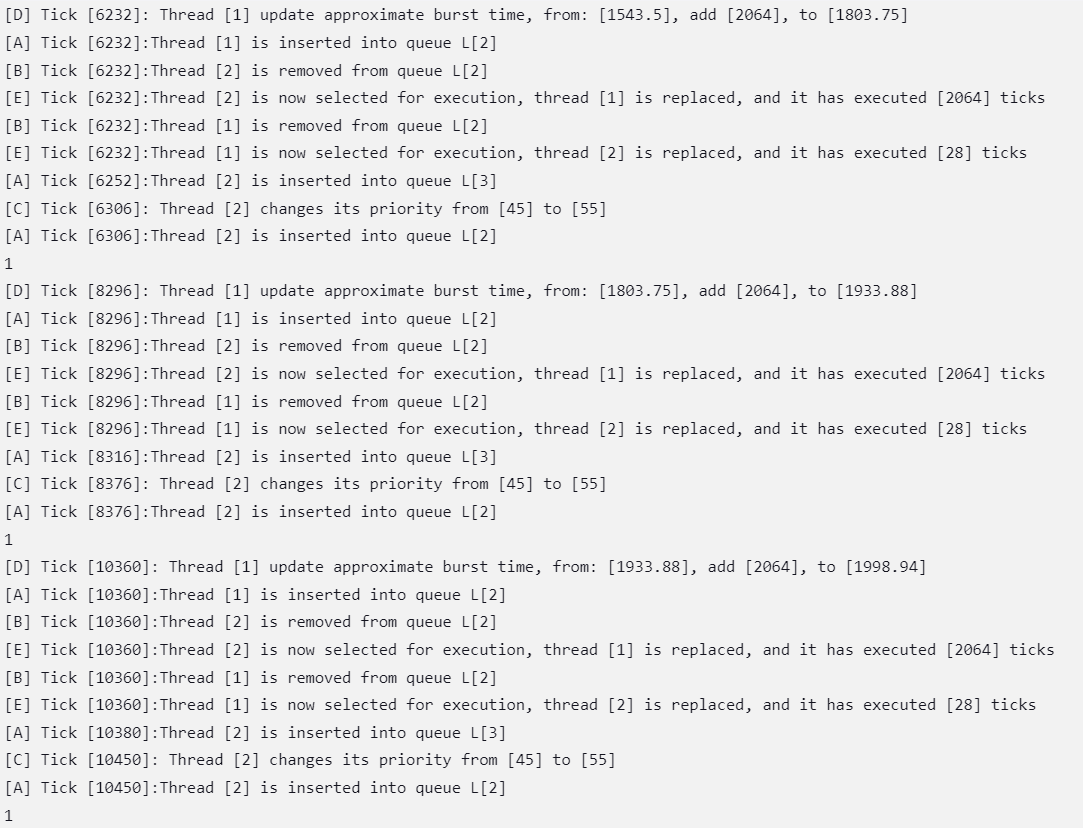
****

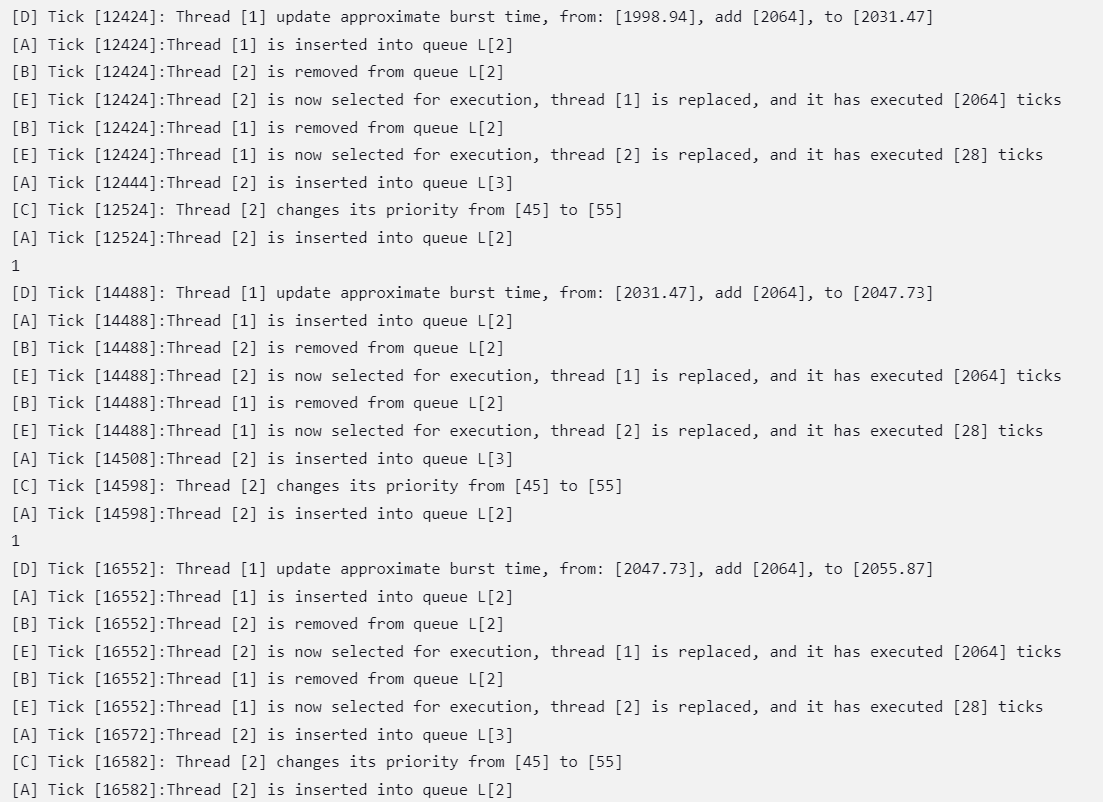
****

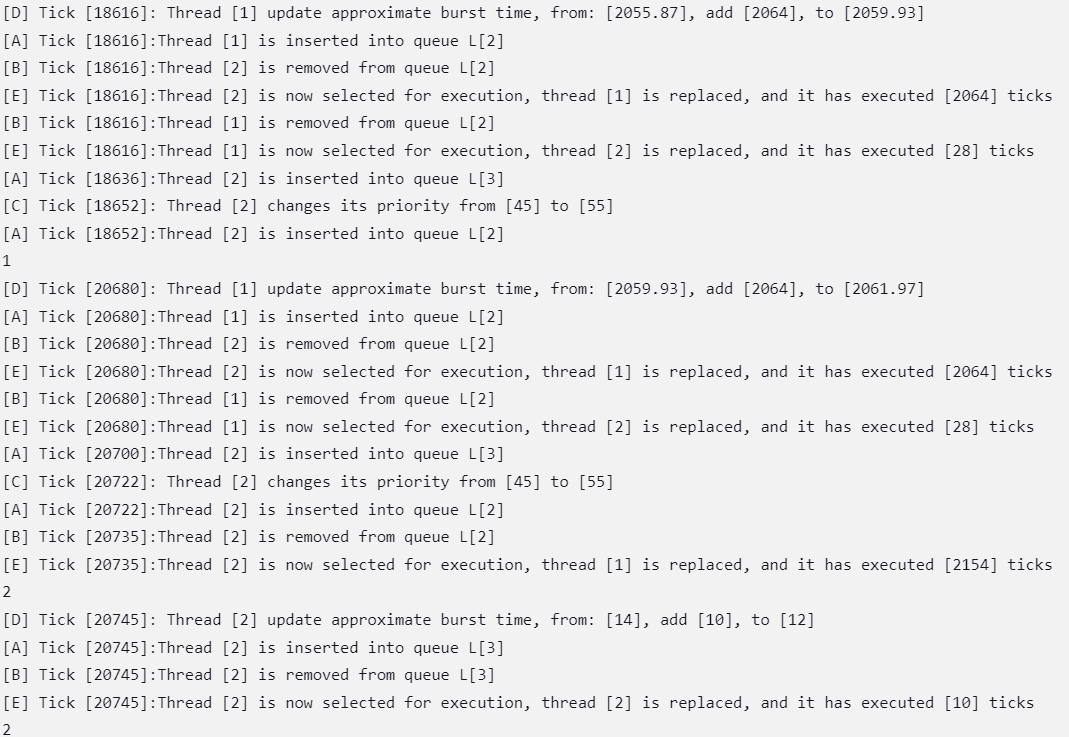
****

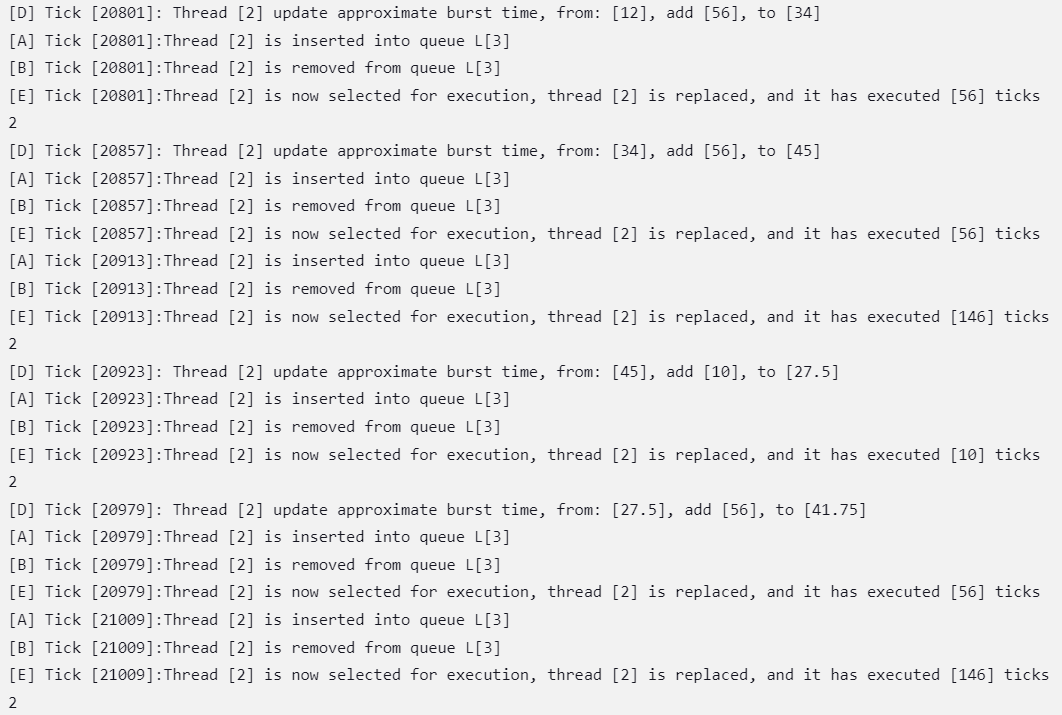
****

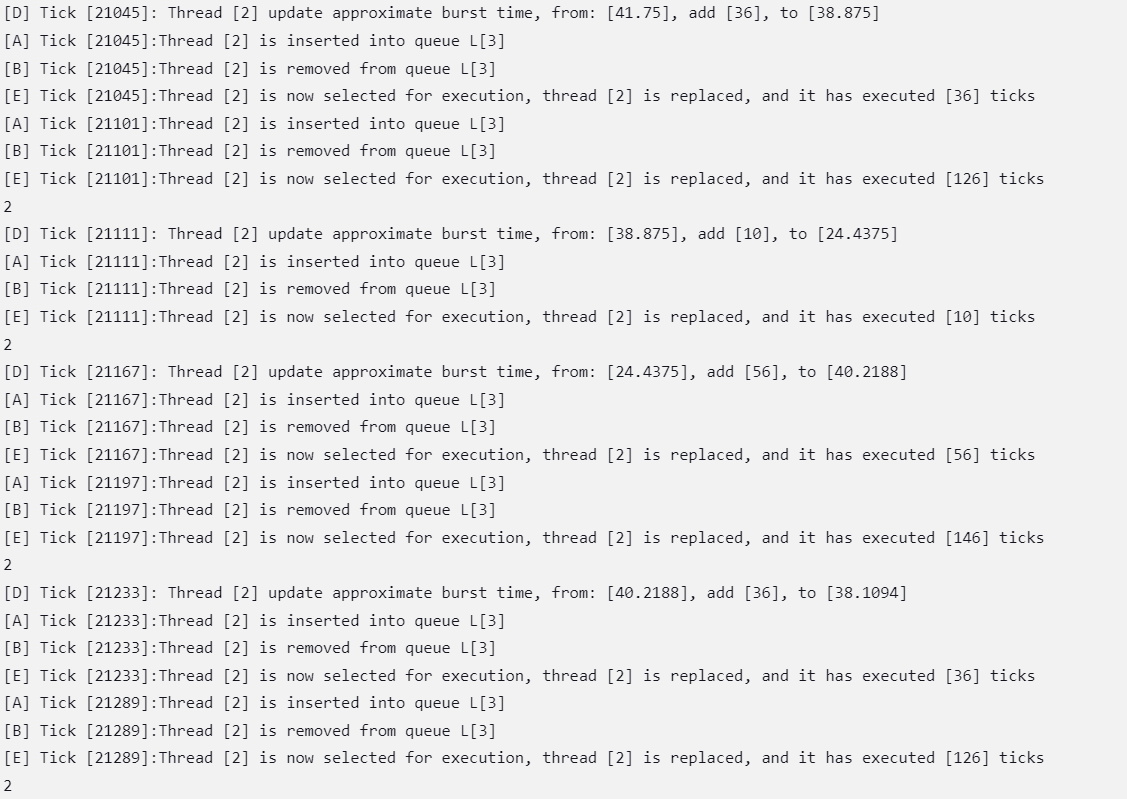
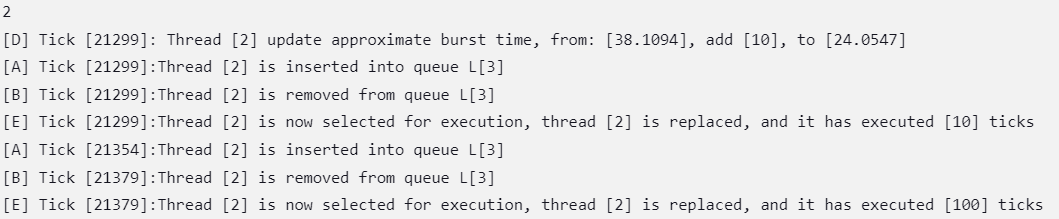
****

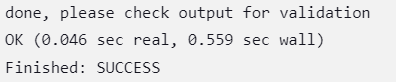
****

****

****

****

****

****

**Part 2:Contribution**

1. **Describe details and percentage of each member’s contribution.**

|  |  |  |
| --- | --- | --- |
|  | 吳孟儒 | 張世傑 |
| **Part1.** | 50 % | 50 % |
| **Part2.** | 50 % | 50 % |