Sleep(),Finish(),Yield():Thread.cc

Accumulate time:在running state跑多久

Onetick():interrupt.cc

Ready to Run:

接受Terminal端的參數作為execfile,為每個execfile建立一個thread,宣告一個Address space.Fork這個procedure,其中會set register value和load page table register(Execute()),分配Stack空間(StackAllocate),把interrupt關掉後就把Thread放到Read to Run state.

完成Forked procedure,最後current->Finish(),call Sleep(),放棄CPU.

Run to Ready:

將User Program的一個指令抓過來Decode,執行OneTick(),將System跟User都tick++,CheckIfDue會檢查是否有到期的pending interrupt,Yield()找下一個在ready list的thread(FindNextToRun),並把現在這個執行的thread放回ready list重新排隊(ReadyToRun),並做context switch(Sceduler::Run)

Timer定期發出一個Interrupt來呼叫"YieldOnReturn()"函式

把yieldOnReturn設為True,OneTick看到這個flag為True就會context switch

Running to Waiting:

發生在I/O或system call.

Read a character typed at the keyboard, Semaphore::P():類似wait(),semaphore value--, ,會把Thread放進queue中(Append), 並call Sleep()把thread Block住,這時候CPU空掉了,要找下一個在readyqueue的thread(FindNextToRun),找不到就idle,最後分配CPU給下一個在ready queue的thread(Run())

Waiting->Ready:

Semaphore::V() 類似Signal() semaphore value++, 並且把Waiting queue的front thread取出來,放回去Ready queue(ReadyToRun())

Running to Terminating:

Exception Handler的SC\_EXIT 執行Finish(),他會call,Sleep(),並且把finishing設成True,這時候找ready queue最前面的thread(FindNextToRun()),最後分配CPU(Run()),前面的fininshing=True在Run()代表最後會刪除這個Thread

Running to Ready

從Ready queue挑選下一個thread(FindNextToRun()).

Switch.s:

Register value分別代表

eax:points to startup function對應ThreadBegin()會enable interrupt

edx:對應(void\*) arg

esi 對應(void) func也就是ForkExecute()

edi對應(void\*)ThreadFinish

esp存放新Thread的Program counter對應(void\*) ThreadRoot

T1 change to T2

t1把所有register相關的資訊儲存起來,t2從memory 把相關Register的資訊Load Register,到達ret時set CPU program counter to the memory address pointed by the value of register esp,也就是抓取ThreadRoot位置,ThreadRoot會call

StartupRC:ThreadBegin()

InitialPC ForkExecute()

WhenDonePC ThreadFinish().

Switch回來後,

如果thread1是被Block的(waiting state)且Finish=True,Run執行完後會把thread1 delete掉,並執行新的user program(execfile),若Finish=false,會挑選下一個在ready queue的thread.

Run():將instruction Decode,並且用OneTick模擬Clock執行時間

Implement:

2-2

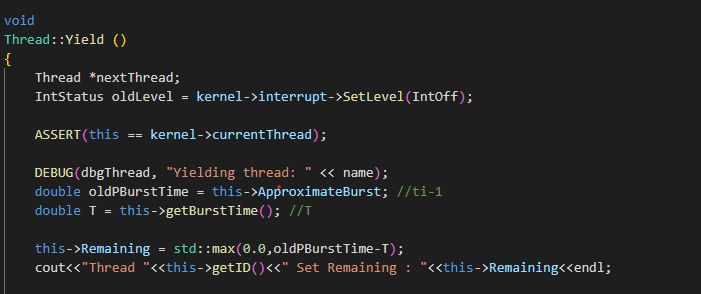
Running->Ready 計算時間放在Yield(),因為FindNextToRun已經在Ready

(a):放在ReadyToRun

(b):放在FindNextToRun

第一次I/O的時候T=0,approximtate = 275 所以max(0,275-0) =275 然後t1就會一直執行

當Running到Ready的時候算出Remaining



If running thread Remaining time < next thread Remaining time

static int

compareSR(Thread\* t1,Thread\* t2)

{

    //選最小Remainging time

        if(t1->Remaining > t2->Remaining) return 1;

        else if(t1->Remaining < t2->Remaining) return -1;

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

        return 0;

}

Scheduler::Scheduler()

{

    readyList = new List<Thread \*>;

    SRList = new SortedList<Thread \*>(compareSR);

    toBeDestroyed = NULL;

}

//----------------------------------------------------------------------

// Scheduler::~Scheduler

//  De-allocate the list of ready threads.

//----------------------------------------------------------------------

Scheduler::~Scheduler()

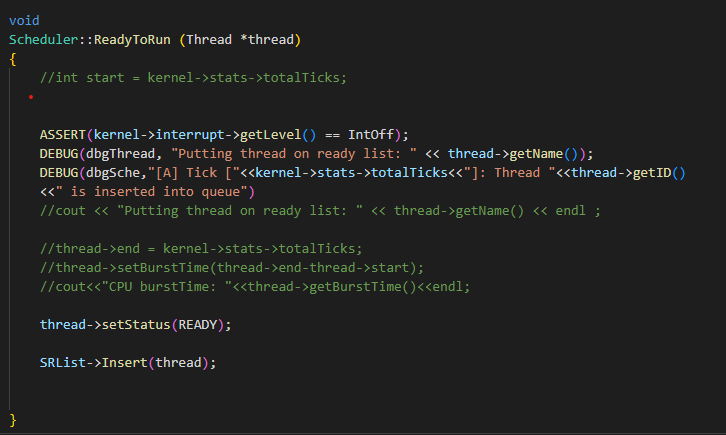
{

    delete readyList;

    delete SRList;

}

新增 SRList的一個SortedList,以Remaining作為排序,小的會在Front.



在ReadyToRun中

依照Spec,印出Debug訊息,並把下一個要run的thread放進SRList

Thread \*

Scheduler::FindNextToRun ()

{

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

    if (SRList->IsEmpty()) {

        return NULL;

    } else {

        Thread\* frontThread  = SRList->RemoveFront();

        DEBUG(dbgSche,"[B] Tick ["<<kernel->stats->totalTicks<<"]: Thread "<<frontThread->getID()

    <<" is removed from queue")

        if(kernel->currentThread->Remaining<frontThread->Remaining){

            SRList->Insert(frontThread);

        }

        return frontThread ;

    }

}

FindNextToRun中,印出Debug訊息,如果現在的Running Thread的Remain burst time比SortedList的frontThread還小,就把前面Remove的thread再插回去,現在Running thread繼續跑.

void

Thread::Yield ()

{

    Thread \*nextThread;

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

    ASSERT(this == kernel->currentThread);

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

    double oldPBurstTime = this->ApproximateBurst; //ti-1

    double T = this->getBurstTime(); //T

    this->Remaining = std::max(0.0,oldPBurstTime-T);

    //cout<<"Thread "<<this->getID()<<" Set Remaining : "<<this->Remaining<<endl;

    nextThread = kernel->scheduler->FindNextToRun(); //這個接下來變Running

    if (nextThread != NULL ) {

    // [E]

    DEBUG(dbgSche,"[E] Tick ["<<kernel->stats->totalTicks<<"]: Thread "<<nextThread->getID()<<

    " is now selected for execution, thread ["<<this->getID()<<"] is preempted"<<

    ", and it has executed ["<<this->getBurstTime()<<"] ticks");

    if(kernel->currentThread->Remaining>nextThread->Remaining){

        kernel->scheduler->ReadyToRun(this);  //把這個thread從Running到Ready

    }

    nextThread->start = kernel->stats->totalTicks;

    kernel->scheduler->Run(nextThread, FALSE); //把readylist的thread送去running

    //nextThread->end= kernel->stats->totalTicks;

    }

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

}

在Yield funtion中,每次從Running回到Ready都必須要設置RemainingTime,如果現在thread 的Remaining比較大,就insert到SRList

void

Thread::Sleep (bool finishing)

{

    Thread \*nextThread;

    ASSERT(this == kernel->currentThread);

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

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

    DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);

    status = BLOCKED;

    this->end = kernel->stats->totalTicks;

    this->setBurstTime((this->end-this->start)+this->getBurstTime());

    double BurstTime = this->getBurstTime();   //T

    double PreApproximateBurst = this->ApproximateBurst; //ti-1

    this->ApproximateBurst = 0.5\*BurstTime + 0.5\*PreApproximateBurst; //ti //更新

    //[C]

    DEBUG(dbgSche,"[C] Tick ["<<kernel->stats->totalTicks<<"]: Thread "<<kernel->currentThread->getID()<<

    " update approximate burst time, from: "<<0.5\*PreApproximateBurst<<" add "<<0.5\*BurstTime<<

    " , to "<<this->ApproximateBurst);

    this->setBurstTime(0);  //Burst Time = 0

    //cout << "debug Thread::Sleep " << name << "wait for Idle\n";

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

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

    }

    //[D]

    DEBUG(dbgSche,"[D] Tick ["<<kernel->stats->totalTicks<<"]: Thread "<<nextThread->getID()<<

    " is now selected for execution, thread ["<<this->getID()<<"] starts IO"<<

    ", and it has executed ["<<BurstTime<<"] ticks");

    // returns when it's time for us to run

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

}

在Yield的Run()中,設置一個計時start等到被Block時進入Sleep,設置一個end.相減之後再加上在Running跑的時間.就是總共的BurstTime,最後把這個BurstTime歸零,並印出所有Debug訊息