MP2_Report_Team11

Team Member

- 資工大三 108062213 顏浩昀
- 電資大三 108061250 吳長錡

Contributions

Work	Member
Trace code	顏浩昀&吳長錡
Implement	顏浩昀&吳長錡
Debug	顏浩昀&吳長錡
report（TraceCode）	顏浩昀&吳長錡
report（Implement）	顏浩昀

PartII-1 Trace code

Kernel::Exec

```
int Kernel::Exec(char* name)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum-1;
```

從Kernel::Exec中我們可以知道要執行執行一個程式，我們需要

1. 新增一個Thread
2. 給一個AddrSpace
3. 透過Fork()載入要執行的程式碼
4. 紀錄Thread數量+1，並會傳現在是第幾個Thread

ForkExecute()

```
void ForkExecute(Thread *t)
{
    if ( !t->space->Load(t->getName()) ) {
        return;                 // executable not found
    }

    t->space->Execute(t->getName());

}
```

(於Kernel.cc中)
接著先看到 `Fork()` 中用到的 `ForkExecute()` ，它會呼
叫 `AddrSpace` 的 `Load()` 與 `Execute()` ，將要執行的程式載入，並使kernel中的Register與PageTable對應到現在要執行的程式，並在最後執行 `machine->Run()` 進行程式執行。

```
void AddrSpace::Execute(char* fileName)
{

    kernel->currentThread->space = this;
    this->InitRegisters();            // set the initial register values
    this->RestoreState();             // load page table register
    kernel->machine->Run();           // jump to the user progam
    ASSERTNOTREACHED();                       // machine->Run never returns;
                                      // the address space exits
                                      // by doing the syscall "exit"
}


void AddrSpace::InitRegisters()
{
    Machine *machine = kernel->machine;
    int i;
    for (i = 0; i < NumTotalRegs; i++)
        machine->WriteRegister(i, 0);
    machine->WriteRegister(PCReg, 0);
    machine->WriteRegister(NextPCReg, 4);
    machine->WriteRegister(StackReg, numPages * PageSize - 16);

}


void AddrSpace::RestoreState()
{
    kernel->machine->pageTable = pageTable;
    kernel->machine->pageTableSize = numPages;
}
```

Thread::Fork()

```
void
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " " << arg);
    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);    // ReadyToRun assumes that interrupts
                                    // are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
```

回到 Fork() 的部分，我們發現Fork還有執行 StackAllocate 、 SetLevel 與 ReadyToRun ，用來執行：

1. Allocate a excution stack and init it.
2. 將interrupt設為IntOff
3. 將要Thread放入readyList中

```
void Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

Kernel::ExecAll()

```
void Kernel::ExecAll()
{
    for (int i=1;i<=execfileNum;i++) {
        int a = Exec(execfile[i]);
    }
    currentThread->Finish();
    //Kernel::Exec();
}
```

在 `ExecAll()` 中就可以看到會一個一個去執行所有的程式，在執行完後執行 `Finish()`

```
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);
    Sleep(TRUE);                                // invokes SWITCH
    // not reached
}
```

於是在 `Finish()` 會呼叫 `Sleep()`

Thread::Sleep()

```
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;
  // 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
  }
  // returns when it's time for us to run
  kernel->scheduler->Run(nextThread, finishing);
}
```

`Sleep()` 要做的事就是要關掉現在這個thread，但因為直接關掉thread就會失去位置，因此先將現在的thread status設為BLOCKED，然後找尋nextThread，如果有ReadyList還不是empty，就會因為 `FindNextToRun()` 的關係將ReayList的front設為nextThread

```
Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    if (readyList->IsEmpty()) {
                return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

Scheduler::Run()

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;

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

    if (finishing) {    // mark that we need to delete current thread
        ASSERT(toBeDestroyed == NULL);
     toBeDestroyed = oldThread;
    }

    if (oldThread->space != NULL) { // if this thread is a user program,
        oldThread->SaveUserState();     // save the user's CPU registers
    oldThread->space->SaveState();
    }

    oldThread->CheckOverflow();         // check if the old thread
                                        // had an undetected stack overflow

    kernel->currentThread = nextThread;  // switch to the next thread
    nextThread->setStatus(RUNNING);      // nextThread is now running

    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".

    SWITCH(oldThread, nextThread);

    // we're back, running oldThread

    // interrupts are off when we return from switch!
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

    CheckToBeDestroyed();       // check if thread we were running
                                // before this one has finished
                                // and needs to be cleaned up

    if (oldThread->space != NULL) {     // if there is an address space
        oldThread->RestoreUserState();     // to restore, do it.
    oldThread->space->RestoreState();
    }
}
```

1. Save the state of the old thread
2. Check if the old state had an undetected stack overflow
3. Dispatch the CPU to next thread

   > Set currentThread as nextThread
   > Set nextThread status as RUNNING

4. SWITCH
5. Clean up the thread has finished(old one)

```
void
Thread::CheckOverflow()
{
    if (stack != NULL) {
#ifdef HPUX                     // Stacks grow upward on the Snakes
        ASSERT(stack[StackSize - 1] == STACK_FENCEPOST);
#else
        ASSERT(*stack == STACK_FENCEPOST);
#endif
   }
}
```

```
void
Scheduler::CheckToBeDestroyed()
{
    if (toBeDestroyed != NULL) {
        delete toBeDestroyed;
        toBeDestroyed = NULL;
    }
}
```

PartII-2 Implement

1. 先實作PageTable的部分。

- 首先先在 `addrspace.h  Class AddrSpace` 中新增 `static bool usedPhysicalPage[NumPhysPages]` ，用來記錄被使用過的PhyscalPages。
- 接著在 `addrspace.cc  Load()` 中加入下方程式碼，用來紀錄virtual page與physical page的轉換，並同時**set up “valid, readOnly, use, and dirty” field for my pagetable**。

```
 1  for(int i = 0; i < numPages; i++){
 2          int j = 0;
 3          pageTable[i].virtualPage = i;
 4          while(j<NumPhysPages && AddrSpace::usedPhysicalPage[j])  j++;
 5
 6          AddrSpace::NumFreePage--;
 7          AddrSpace::usedPhysicalPage[j] = true;
 8          pageTable[i].physicalPage = j;
 9          pageTable[i].valid = true;
10          pageTable[i].use = false;
11          pageTable[i].dirty = false;
12          pageTable[i].readOnly = false;
13       }
```

2. 處理MLE的部分。

- 首先根據Spec提示至 `machine.h` 新增一個 ExceptionType `MemoryLimitException` 於 `NumExceptionTypes` 之前（index = 8）。
- 接著在 `addrspace.h` 再新增 `static int NumFreePage`
  初始值設為NumPhysPages（128），用來確定還有多少page可以使用。看到上方程式碼line 6的部分可以發現，當有一頁physcal page被用掉後，就會將NumFreePage-1。並在AddrSpace的Destructor加入下方程式碼，來增加可用的FreePage數。同時也會將usedPhysicalPage還原成可用的狀態。

```
AddrSpace::~AddrSpace()
{
    for(int i = 0; i < numPages; i++){
        AddrSpace::usedPhysicalPage[pageTable[i].physicalPage] = false;
        AddrSpace::NumFreePage++;
    }
   delete pageTable;
}
```

- 最後將原本在 `Load()` 裡的 `ASSERT(numPages<=NumPhysicalPages)` 改為 `if(numPages > NumFreePage) ExceptionHandler(MemoryLimitException)` ，這樣改之後能確定不會FreePage不夠的時候會進到ExceptionHandler，而不是避免單一個file的numPages太大。另外，如果只使用 `ASSERT()` 將只會有**Assertion fail及Aborted**兩行輸出結果，不符合Spec所說的三行輸出結果，因此需要Exceptionhandler case default裡的**Unexpected user mode exception**這句輸出。