NachOS MP1

Trace code

(a) SC_Halt

Trace the SC_Halt system call to understand the implementation of a system call. (Sample code : halt.c)

void Halt();
userprog/syscall.h

trace code 的目標是 syscall.h 的 Halt()。

```
#define SC_Halt 0

void Halt();
```

```
/* test/start.s */

      .globl Halt
      .ent  Halt
Halt:
      addiu $2,$0,SC_Halt
      syscall
      j     $31
      .end Halt
```

執行指令 addiu，SC_Halt 是定義在 "syscall.h" 的常數，MIPS 架構的 register(0) 永遠是 0 ，將兩者相加後存到 register(2) 中，即將 SC_Halt 的值 assign 給 register(2)。

```
addiu $2,$0,SC_Halt
```

執行指令 syscall，進入Exception handler。

```
syscall
```

在 machine.h 有定義常數 RetAddrReg 的值是 31。跳到 register(31) 所儲存的位置，表示回到 frame 的上一層。

```
j     $31
```

1. void Machine::Run()
machine/mipssim.cc

程式開始執行，kernel 呼叫 Machine::Run()，Machine::Run() 在無窮迴圈呼叫 OneInstruction()，不停的執行指令。"mipssim.h" 定義了 MIPS 架構支援的 63 個 operation，OneInstruction() 負責處理這 63 個 operation 組成的指令。

```
void Machine::Run()
{
    for(;;)
    {
        OneInstruction(instr);
    }
}
```

2. void Machine::OneInstruction(Instruction *instr)
machine/mipssim.cc

machine.h 定義常數PCReg 值 34，register(34) 放有當前指令的位址。不同指令進入不同的 switch case，當指令是 addiu，不會發生 interrupt，但當指令是 syscall，interupt 發生，呼叫 RaiseException()。

```
Void Machine::OneInstruction(Instruction *instr)
{
    int raw;

    /* fetch instruction */
    if (!ReadMem(registers[PCReg], 4, &raw)) {
        return;
    }

    instr->value = raw;
    instr->Decode();

    switch (instr->opCode) {
    case OP_SYSCALL:
        RaiseException(SyscallException, 0);
        return;
    }
}
```

NachOS 定義了 9 種 exception 類型，在 case OP_SYSCALL 發生的是SyscallException 類型的 exception。

```
/* machine/machine.h */

enum ExceptionType {
    NoException,
    SyscallException,
    PageFaultException,
    ReadOnlyException,
    BusErrorException,
    AddressErrorException,
    OverflowException,
    IllegalInstrException,
    NumExceptionTypes
};
```

3. Void Machine::RaiseException(ExceptionType which, int badVAddr)

machine/machine.cc

把程式從 user mode 改成 kernel mode，呼叫 ExceptionHandler。

```
Void Machine::RaiseException(ExceptionType which, int badVAddr)
{
    kernel->interrupt->setStatus(SystemMode);
    ExceptionHandler(which);
    kernel->interrupt->setStatus(UserMode);
}
```

4. void ExceptionHandler(ExceptionType which)

userprog/exception .cc

ExceptionHandler() 讀取 register(2) 的值，先前已經把 SC_Halt 的值 assign 給 register(2)，因此進入 SC_Halt 的 switch case，呼叫 SysHalt()。

```
void ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
    switch(type) {
    case SC_Halt:
        SysHalt();
        break;
    }
}
```

5. void SysHalt()

userprog/ksyscall.h

```
void SysHalt()
{
    kernel->interrupt->Halt();
}
```

6. Void Interrupt::Halt()

machine/interrupt.cc

將物件 kernel 所佔用的記憶體釋放，即關機。

```
Void Interrupt::Halt()
{
    cout << "Machine halting!\n\n";
    cout << "This is halt\n";
    kernel->stats->Print();
    delete kernel;
}
```

(b) SC_Create

Trace the SC_Create system call to understand the basic operations and data structure in a file system. (Sample code : createFile.c)

void Create();
userprog/syscall.h

trace code 的目標是 syscall.h 的 Create()。

```
#define SC_Create 4

int Create(char *name);
```

```
/* test/start.s */

#include "syscall.h"

      .globl Create
      .ent  Create
Create:
      addiu $2,$0,SC_Create
      syscall
      j     $31
      .end Create
```

1. OneInstruction(Instruction *instr)
machine/mipssim

```
#include "machine.h"
#include "mipssim.h"

Void Machine::OneInstruction(Instruction *instr)
{
    switch (instr->opCode) {
      case OP_SYSCALL:
         RaiseException(SyscallException, 0);
         return;
    }
}
```

2. RaiseException(ExceptionType which, int badVAddr)

machine/machine.cc

```
#include "machine.h"

Void Machine::RaiseException(ExceptionType which, int badVAddr)
{
    kernel->interrupt->setStatus(SystemMode);
    ExceptionHandler(which);
    kernel->interrupt->setStatus(UserMode);
}
```

3. ExceptionHandler(ExceptionType which)

userprog/exception.cc

從 register(4) 讀出呼叫 Create 時傳入的參數，即 filename 的位置。Syscreate 回傳是否有成功的 create file，成功回傳 1，失敗回傳 0。完成 SysCreate 後必須修改 PCReg 讓 PCReg 指向下一個指令，如果沒有加上這一行，程式會不停的執行同一道指令。

```
#include "syscall.h"
#include "ksyscall.h"

void ExceptionHandler(ExceptionType which)
{
   int type = kernel->machine->ReadRegister(2);

   switch(type)
   {
       case SC_Create:
          val = kernel->machine->ReadRegister(4);
          char *filename = &(kernel->machine->mainMemory[val]);
          status = SysCreate(filename);
          if(status != -1) status = 1;
          kernel->machine->WriteRegister(2, (int) status);

          kernel->machine->WriteRegister(PrevPCReg,
                                kernel->machine->ReadRegister(PCReg));
          kernel->machine->WriteRegister(PCReg,
                                kernel->machine->ReadRegister(PCReg) + 4);
          kernel->machine->WriteRegister(NextPCReg,
                                kernel->machine->ReadRegister(PCReg)+4);
          return;
          ASSERTNOTREACHED();
          break;
   }
}
```

4. int SysCreate(char *filename)
userprog/ksyscall.h

```
int Interrupt::CreateFile(char *filename)
{
    return kernel->CreateFile(filename);
}
```

5. int Interrupt::CreateFile(char *filename)
machine/interrupt.cc

class FileSystem 是 NachOS 的 file system API。

```
int Kernel::CreateFile(char *filename)
{
    return fileSystem->Create(filename);
}
```

在 filesys.cc 的開頭有一段 #ifdef #else，有沒有 #define FILESYS_STUB 使用的是不同的檔案系統，如果有 #define FILESYS_STUB，並不是使用真正的 NachOS 檔案系統，只是借用了 linux 的檔案系統，如果沒有 #define FILESYS_STUB，才是使用 NachOS 的檔案系統。

```
#ifdef FILESYS_STUB
```

Makefile 裡有下 DFILESYS_STUB 的 flag，代表目前 NachOS 只是借用了 linux 的檔案系統。

```
DEFINES =  -DFILESYS_STUB -DRDATA -DSIM_FIX
```

6. bool Filesystem::OpenForWrite(char *name)
filesys/filesys.cc

```
#ifdef FILESYS_STUB

class FileSystem {
public:
    bool Create(char *name)
      {
            int fileDescriptor = OpenForWrite(name);

            if (fileDescriptor == -1) return FALSE;
            Close(fileDescriptor);
            return TRUE;
```

```
      }
}
```

7. int OpenForWrite(char *name)

lib/sysdep.c

目前的檔案系統是借用 linux 的檔案系統，Create 的實做是呼叫了 linux 提供的 open()，open() 失敗回傳 -1，成功回傳非負整數。NachOS 目前沒有真正的檔案系統，如果我們要使用真正的 NachOS 檔案系統，就要在 File System 的 API 之下，自己寫一個檔案系統。

```
#include <stdlib.h>
#include <sys/file.h>

int OpenForWrite(char *name)
{
    int fd = open(name, O_RDWR|O_CREAT|O_TRUNC, 0666);
    ASSERT(fd >= 0);
    return fd;
}
```

(c) PrintInt

Trace the SC_PrintInt system call to understand how NachOS implements asynchronized I/O using CallBack functions and register schedule events. (Sample code : add.c)

void PrintInt(int number);
userprog/syscall.h

trace code 的目標是 syscall.h 的 PrintInt()。

```
#define SC_PrintInt 16

void PrintInt(int number);
```

```
/* test/start.s */

#include "syscall.h"

      .globl PrintInt
      .ent  PrintInt
PrintInt:
      addiu $2,$0,SC_PrintInt
      syscall
      j     $31
      .end PrintInt
```

1. OneInstruction(Instruction *instr)
machine/mipssim

```
Void Machine::OneInstruction(Instruction *instr)
{
    int raw;

    if (!ReadMem(registers[PCReg], 4, &raw))
      return;

    instr->value = raw;
    instr->Decode();

    switch (instr->opCode) {
    case OP_SYSCALL:
        RaiseException(SyscallException, 0);
        return;
    }
}
```

2. RaiseException(ExceptionType which, int badVAddr)

machine/machine.cc

```
Void Machine::RaiseException(ExceptionType which, int badVAddr)
{
    kernel->interrupt->setStatus(SystemMode);
    ExceptionHandler(which);
    kernel->interrupt->setStatus(UserMode);
}
```

3. ExceptionHandler(ExceptionType which)

userprog/exception .cc

```
#include "syscall.h"
#include "ksyscall.h"

void ExceptionHandler(ExceptionType which)
{
   int type = kernel->machine->ReadRegister(2);

   switch(type)
   {
       case SC_PrintInt:
          val = kernel->machine->ReadRegister(4);
          SysPrintInt(val);
          kernel->machine->WriteRegister(PrevPCReg,
                                kernel->machine->ReadRegister(PCReg));
          kernel->machine->WriteRegister(PCReg,
                                kernel->machine->ReadRegister(PCReg) + 4);
          kernel->machine->WriteRegister(NextPCReg,
                                kernel->machine->ReadRegister(PCReg)+4);
          break;
   }
}
```

4. SysPrintInt()

userprog/ksyscall.h

```
void SysPrintInt(int number)
{
    kernel->synchConsoleOut->PutInt(number);
}
```

5. void SynchConsoleOutput::PutInt(int number)

userprog/synchconsole.cc

只有一個 console，因此必須要 sync，輸出的過程是 critical section，一次只能有一個 thread 做 console output。使用 while 迴圈，將字串中的字元一個一個的印出。在此使用 do while 迴圈，因為第一個輸出的字元不需要等待，他既然可以成功 acquire 到 lock 代表目前沒有人在進行輸出，但接下來的字元輸出都要等前一個字元輸出完成才能進行。

```
void SynchConsoleOutput::PutInt(int value)
{
    char str[15];
    int idx=0;

    sprintf(str, "%d\n\0", value); /* convert int into string */
    lock->Acquire();  /* enter critical section */
    do{
        consoleOutput->PutChar(str[idx]);
        idx++;
        waitFor->P();
    } while(str[idx] != '\0');
    lock->Release(); /* leave critical section */
}
```

```
class SynchConsoleOutput : public CallBackObj {
private:
    ConsoleOutput *consoleOutput; /* the hardware display */
    Lock *lock; /* only one writer at a time */
    Semaphore *waitFor; /* wait for callBack */
    void CallBack(); /* called when more data can be written */
};
```

waitFor 是 class Semaphore 的物件，當呼叫 P() 時，若 waitFor.value < 0 ，thread 會被加入 waitFor.queue 中等待。

```
waitFor->P();
```

```
/* threads/synch.h */

class Semaphore {
public:
    void P(); /* waits until value > 0, then decrement */
    void V(); /* increment, waking up a thread waiting in P() if
               * necessary */
private:
    int value;
    List<Thread *> *queue; /* threads waiting in P() for the value to be
                            * > 0 */
};
```

在 value > 0 之前都會卡在迴圈裡，在前一個字元做輸出時，thread 會進入 Sleep() 中，scheduler 會安排正在readyList 的 thread 使用 CPU。

```
/* threads/synch.cc */

void Semaphore::P()
{
    while (value == 0) {
        queue->Append(currentThread);
        currentThread->Sleep(FALSE);
    }
    value--;
}
```

6. ConsoleOutput::PutChar()

machine/console.cc

呼叫 WriteFile() 對螢幕進行輸出，呼叫 Schedule() 模擬螢幕完成輸出後發出 interrupt。

```
void ConsoleOutput::PutChar(char ch)
{
    WriteFile(writeFileNo, &ch, sizeof(char));
    kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
}
```

WriteFile() 的實做是呼叫 linux 提供的 write()，fd 是被寫入檔案的 file descriptor，writeFileNo 的值在 console.cc 中被 assign 為 1。 在 linux 系統中，file descriptor 為 1 即是 stdout，字元會被輸出到螢幕。

```
/* lib/sysdep.cc */

int WriteFile(int fd, char *buffer, int nBytes)
{
    int retVal = write(fd, buffer, nBytes);
    return retVal;
}
```

一個真正的作業系統會在輸出完成的時候發出 interrupt，由於這不是真的作業系統，所以他只能預估輸出時間然後自己發出 interrupt 。NachOS 預估字元輸出需要 100 單位時間，因此安排在 100 個單位時間後發出 interrupt 。關鍵字 this 通常被用在一個 class 內部，this 是一個指標，指向正在被執行的 class 的物件，this 在此處是 ConsoleOutput。

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

```
/* machine/stats.h */

const int ConsoleTime = 100; /* time to read or write one character */
```

IntType 紀錄這個 interrupt 是由哪個硬體發出，NachOS支援六種硬體相關 interrupt。

```
/* machine/interrupt.h */

enum IntType { TimerInt, DiskInt, ConsoleWriteInt, ConsoleReadInt,
               NetworkSendInt, NetworkRecvInt};
```

writeFileNo 的值在 console.cc 中被 assign 為 1。

```
/* machine/console.cc */

ConsoleOutput::ConsoleOutput(char *writeFile, CallBackObj *toCall)
{
    if (writeFile == NULL)
        writeFileNo = 1;
    else
        writeFileNo = OpenForWrite(writeFile);
}
```

```
/* threads/kernel.cc */

synchConsoleOut = new SynchConsoleOutput(consoleOut);
```

```
/* threads/kernel.cc */

consoleOut = NULL;
```

7. void Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)
machine/interrupt.cc

把待發生的 interrupt 放進 pending，當排定的時間到達時才會發生。

```
void Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)
{
    int when = kernel->stats->totalTicks + fromNow;
    PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
    pending->Insert(toOccur);
}
```

```
class Interrupt {
private:
    SortedList<PendingInterrupt *> *pending;
};
```

當排定的時間到達，interrupt 發生，回報發出 interrupt 的物件，即 ConsoleOutput，讓他進行 interrupt 的處理。

```
PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
```

when 是 interrupt 排定要發生的時間，totalTicks 是當前時間，fromNow 的值為 ConsoleTime，即 100。

```
int when = kernel->stats->totalTicks + fromNow;
```

8. void Machine::Run()
machine/mipssim.cc

Onetick() 將系統時間推進一個單位，並且檢查是否有 interrupt 正要發生。

```
void Machine::Run()
{
    Instruction *instr = new Instruction;
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
    }
}
```

9. Interrupt::OneTick()
machine/interrupt.cc

```
void Interrupt::OneTick()
{
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    } else {
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
    }
    CheckIfDue(FALSE);
}
```

10. Interrupt::CheckIfDue
machine/interrupt.cc

如果沒有interrupt 要發生，回傳 false，若有 interrupt 要發生，執行並且回傳 true，若 advanceClock 為 true，表示目前已經沒有任何指令要執行，則直接執行一個 interrupt。

```
bool Interrupt::CheckIfDue(bool advanceClock)
{
    PendingInterrupt *next;
    Statistics *stats = kernel->stats;

    if (pending->IsEmpty()) {
      return FALSE;
    }
    next = pending->Front();

    if (next->when > stats->totalTicks) {
        if (!advanceClock) {
            return FALSE;
        }
        else {
          stats->idleTicks += (next->when - stats->totalTicks);
          stats->totalTicks = next->when;
      }
    }
    inHandler = TRUE;

    do {
        next = pending->RemoveFront();
        next->callOnInterrupt->CallBack();
        delete next;
    } while(!pending->IsEmpty()
            && (pending->Front()->when <= stats->totalTicks));
    inHandler = FALSE;
    return TRUE;
}
```

若目前有要發生的 interrupt，把 interrupt 從 pending 中取出，呼叫發出此 interrupt 物件的 callBack()，在 interrupt 處理完之後，回傳 true。

```
do {
    next = pending->RemoveFront();
    next->callOnInterrupt->CallBack();
    delete next;
} while(!pending->IsEmpty()
        && (pending->Front()->when <= stats->totalTicks));
```

11. void ConsoleOutput::CallBack()
machine/console.cc

當螢幕輸出完成，ConsoleOutput 呼叫 SynchConsoleOutput物件的 callBack()。

```
void ConsoleOutput::CallBack()
{
    putBusy = FALSE;
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();
}
```

consoleOutput 的 callWhenDone 是 SynchConsoleOutput。

```
/* machine/console.h */

class ConsoleOutput : public CallBackObj {
private:
    CallBackObj *callWhenDone;
};
```

```
/* machine/console.cc */

ConsoleOutput::ConsoleOutput(char *writeFile, CallBackObj *toCall)
{
    callWhenDone = toCall;
}
```

```
/* userprog/synchconsole.cc */

SynchConsoleOutput::SynchConsoleOutput(char *outputFile)
{
    consoleOutput = new ConsoleOutput(outputFile, this);
}
```

12. SynchConsoleOutput::CallBack()

userprog/synchconsole.cc

```
void SynchConsoleOutput::CallBack()
{
    waitFor->V();
}
```

V() 釋放一個 semaphore 資源，並且 pop 出下一個在 semaphore.queue 的 thread 放進 readyList，等到 scheduler 安排此 thread 可以使用 CPU 時，才能輸出下一個字元。

```
/* threads/synch.cc */

void Semaphore::V()
{
    if (!queue->IsEmpty()) {
      kernel->scheduler->ReadyToRun(queue->RemoveFront());
    }
    value++;
}
```

Implementation

Implement four I/O system calls in NachOS
- OpenFileId Open(char *name);
- int Write(char *buffer, int size, OpenFileId id);
- int Read(char *buffer, int size, OpenFileId id);
- int Close(OpenFileId id);

Open / Close / Read / Write 四者非常像，因此我以 Write 為例子進行實做解說，唯有比較需要說明的部份會將 Open / Close / Read 一起說明。

int Write(char *buffer, int size, OpenFileId id);
userprog/syscall.h

```
#define SC_Write 8

int Write(char *buffer, int size, OpenFileId id);
```

```
/* test/start.s */

#include "syscall.h"

      .globl Write
      .ent  Write
Write:
      addiu $2,$0,SC_Write
      syscall
      j     $31
      .end Write
```

1. Void Machine::OneInstruction(Instruction *instr)
machine/mipssim

```
#include "machine.h"
#include "mipssim.h"

Void Machine::OneInstruction(Instruction *instr)
{
    switch (instr->opCode) {
    case OP_SYSCALL:
        RaiseException(SyscallException, 0);
        return;
    }
}
```

2. RaiseException(ExceptionType which, int badVAddr)

machine/machine.cc

```
#include "machine.h"

Void Machine::RaiseException(ExceptionType which, int badVAddr)
{
    kernel->interrupt->setStatus(SystemMode);
    ExceptionHandler(which);
    kernel->interrupt->setStatus(UserMode);
}
```

3. ExceptionHandler(ExceptionType which)

userprog/exception.cc

依照我們傳入參數的順序 (char *buffer, int size, OpenFileId id) 依序從 register $a0 $a1 $a2，即從 register(4) register(5) register(6) 讀出呼叫 system call Write 時所傳入的參數。SysWrite 會回傳寫入是否成功。依照 MIPS 慣例，函式回傳值應放在 register $v0，即 register(4)。完成 SysWrite 之後要修改 PCReg 讓 PCReg 指向下一個指令，如果沒有加上這一行的話，程式會不停的執行當前指令。

```
#include "syscall.h"
#include "ksyscall.h"

void ExceptionHandler(ExceptionType which)
{
   int type = kernel->machine->ReadRegister(2);

      case SC_Write:
          val = kernel->machine->ReadRegister(4);
          char *buffer = &(kernel->machine->mainMemory[val]);
          size = kernel->machine->ReadRegister(5);
          id = kernel->machine->ReadRegister(6);
          status = SysWrite(buffer, size, id);
          kernel->machine->WriteRegister(2, (int) status);
          kernel->machine->WriteRegister(PrevPCReg,
                                kernel->machine->ReadRegister(PCReg));
          kernel->machine->WriteRegister(PCReg,
                                kernel->machine->ReadRegister(PCReg) + 4);
          kernel->machine->WriteRegister(NextPCReg,
          kernel->machine->ReadRegister(PCReg)+4);
          return;
          ASSERTNOTREACHED();
          break;
}
```

4. int SysWrite(char *buffer, int size, int id)
userprog/ksyscall.h

```
int SysWrite(char *buffer, int size, int id)
{
      return kernel->WriteFile(buffer, size, id);
}
```

5. int Kernel::WriteFile(char *buffer, int size, int id)
machine/interrupt.cc

class FileSystem 是 NachOS 的 file system API。

```
int Kernel::WriteFile(char *buffer, int size, int id)
{
      return fileSystem->WriteF(buffer, size, id);
}
```

6. int Filesystem::WriteField(char *name)
filesys/filesys.h

```
#ifdef FILESYS_STUB

class FileSystem {
    int WriteF(char *buffer, int size, int id)
    {
        int status = WriteFile(id, buffer, size);
        return status;
    }
}
```

7_1. int WriteFile(int fd, char *buffer, int nBytes)
lib/sysdep.cc

class FileSystem 是 NachOS 的檔案系統 API，由於目前的檔案系統是依賴 linux 的檔案系統，system call Write 的實做是呼叫 linux 提供的 write()，write() 失敗回傳 -1，成功回傳成功寫入的字元數。

```
int WriteFile(int fd, char *buffer, int nBytes)
{
    int retVal = write(fd, buffer, nBytes);
    ASSERT(retVal == nBytes);
    return retVal;
}
```

7_2. int OpenForReadWrite(char *name, bool crashOnError)

lib/sysdep.cc

system call Open 的實做是呼叫 linux 提供的 open()，open() 失敗回傳 -1，成功回傳 file description。需要注意的是，如果使用 fopen() 開啟檔案，他回傳的是 FILE*，是一個指標，而 open() 回傳的是一個整數。

```
int OpenForReadWrite(char *name, bool crashOnError)
{
    int fd = open(name, O_RDWR, 0);
    ASSERT(!crashOnError || fd >= 0);
    return fd;
}
```

7_3. int Read(int fd, char *buffer, int nBytes)

lib/sysdep.cc

system call Read 的實做是呼叫 linux 提供的 read()，read() 失敗回傳 -1，成功回傳成功讀出的字元數。

```
int Read(int fd, char *buffer, int nBytes)
{
    int retVal = read(fd, buffer, nBytes);
    ASSERT(retVal == nBytes);
    return retVal;
}
```

7_4. int Close(int fd)

lib/sysdep.cc

system call Close 的實做是呼叫 linux 提供的 read()，read() 失敗回傳 -1，成功回傳 0。

```
int Close(int fd)
{
    int retVal = close(fd);
    ASSERT(retVal >= 0);
    return retVal;
}
```

Command

Compile / Rebuild NachOS

```
> cd NachOS-4.0_MP1/code/build.linux
> make clean
> make depend
> make
```

Test NachOS

```
> cd NachOS-4.0_MP1/code/test
> make clean
> make halt
> ../build.linux/nachos -e halt
```

Reference

[1] shawn2000100/10810CS_342301_OperatingSystem
https://github.com/shawn2000100/10810CS_342301_OperatingSystem