Nachos

Matthew Peters, Robert Hill, Shyam Pather

September 1, 1998

1 Introduction

Nachos is a multitasking operating system simulation that runs in a single Unix process. It implements a multi-process model by using user-level threads. Thus, each user-level thread is viewed as its own process under Nachos.

From figure 1 below you can see the overall structure of Nachos. Each thread (indicated by T1, T2, and so on) is an independent thread of control in the Nachos operating system. For the most part, they can be viewed as processes, only on a smaller scale. It should be noted that **all** of the internals of the Nachos program are invisible to Unix while running: to Unix, Nachos is just another process.

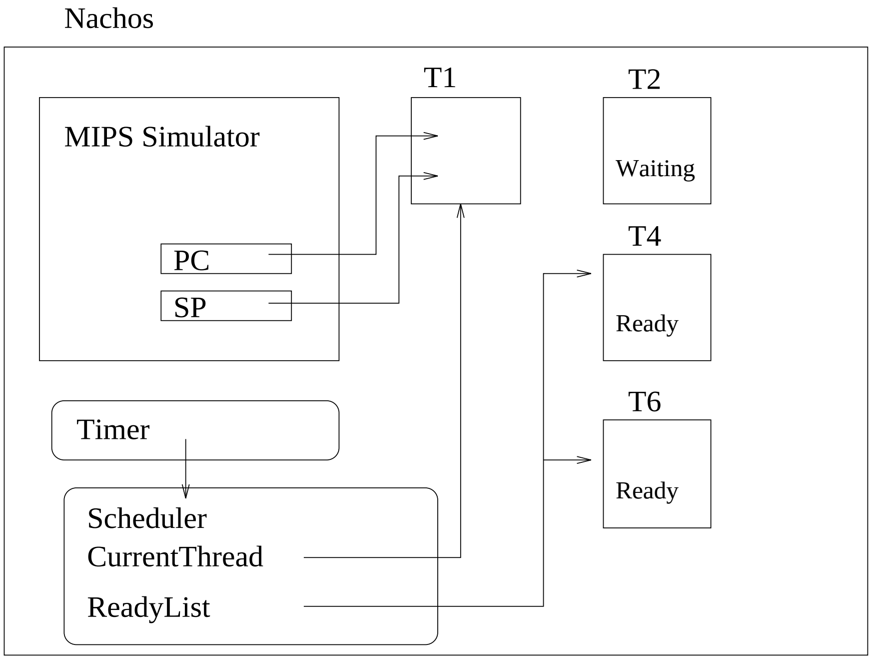

Figure 1: Structure of the Nachos Process

Unlike a real operating system, Nachos doesn't run on bare hardware. To simulate the low-level functions of interrupts, clocks and registers, the Nachos operating system simulates a MIPS processor. This processor is capable of performing the same jobs a normal processor would, including executing arbitrary instructions from a user program.

The Nachos system can run in two modes: User mode, and System mode. In user mode the MIPS simulator is spinning through a series of fetch-execute cycles on a user program. It does this until a system trap occurs, either as the result of an expired quantum, an illegal instruction, a page fault, or a system call. At this point it switches into system mode.

It should be noted that while user programs in Nachos are run on the simulated MIPS machine, System Mode transactions are performed directly on the host machine (in this case, an x86).

2 Machine Setup

The simulated MIPS processor consists of a set of registers (implemented in software as an array of integers), a bank of main memory, and a set of simple read and write instructions. A conceptual view of this is shown in Figure 2.

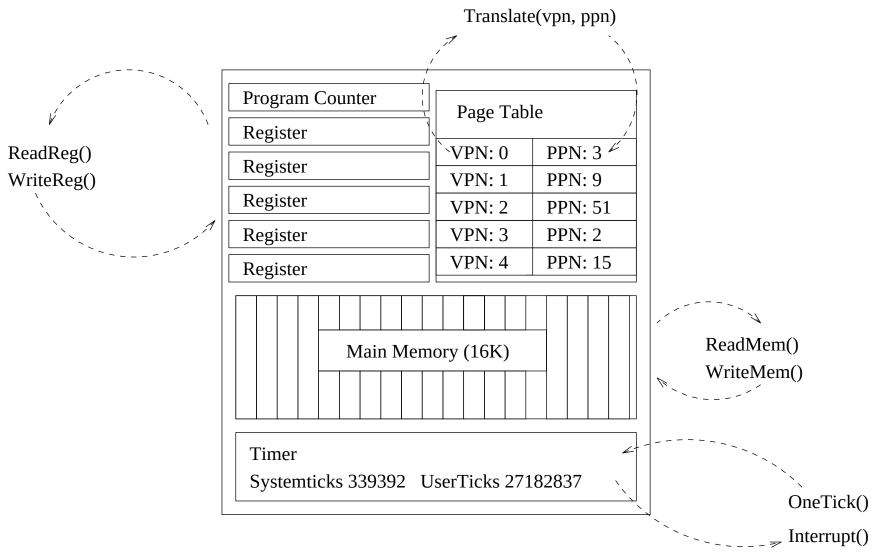

Figure 2: Machine Functions and Structure

When a program is loaded into memory the Nachos operating system reads the contents of the program's executable file from disk, then writes those contents into main memory. This is done one byte at a time. Once the contents of the executable are in memory and the program counter is set, the machine begins execution by calling *Machine::Run()* (located in *machine/mipssim.cc*).

The structure of the MIPS machine can be seen below:

```
class Machine {
  public:
    void Run();
    int ReadRegister(int num);
    void WriteRegister(int num, int value);
```

```
    void OneInstruction(Instruction *instr);
    bool ReadMem(int addr, int size, int* value);
    bool WriteMem(int addr, int size, int value);
    ExceptionType Translate(int virtAddr, int* physAddr, int size);
    void RaiseException(ExceptionType which, int badVAddr);

    char *mainMemory;
    int registers[NumTotalRegs];
};
```

Machine::Run() gets the next instruction (from the position indicated by the program counter) and decodes it. It then uses a switch statement containing one case of each of the instructions in the machine's instruction set, in order to evaluate the instruction. Part of this switch statement is shown below:

```
    switch(instr->opCode) {
...
...
...
      case OP\_SYSCALL:
        RaiseException(SyscallException, 0);
        break;

      case OP\_XOR:
        registers[instr->rd] = registers[instr->rs] ^ ...
        break;

      case OP\_XORI:
        registers[instr->rt] = registers[instr->rs] ^ ...
        break;

      case OP\_UNIMP:
        RaiseException(IllegalInstrException, 0);
        return;
       ...
...
...
    }
```

Note that in the case of an OP_SYSCALL the machine calls *RaiseException*, which is the equivalent of a trap into kernel space in a normal operating system. The path of a system call trap into kernel space can be seen below:

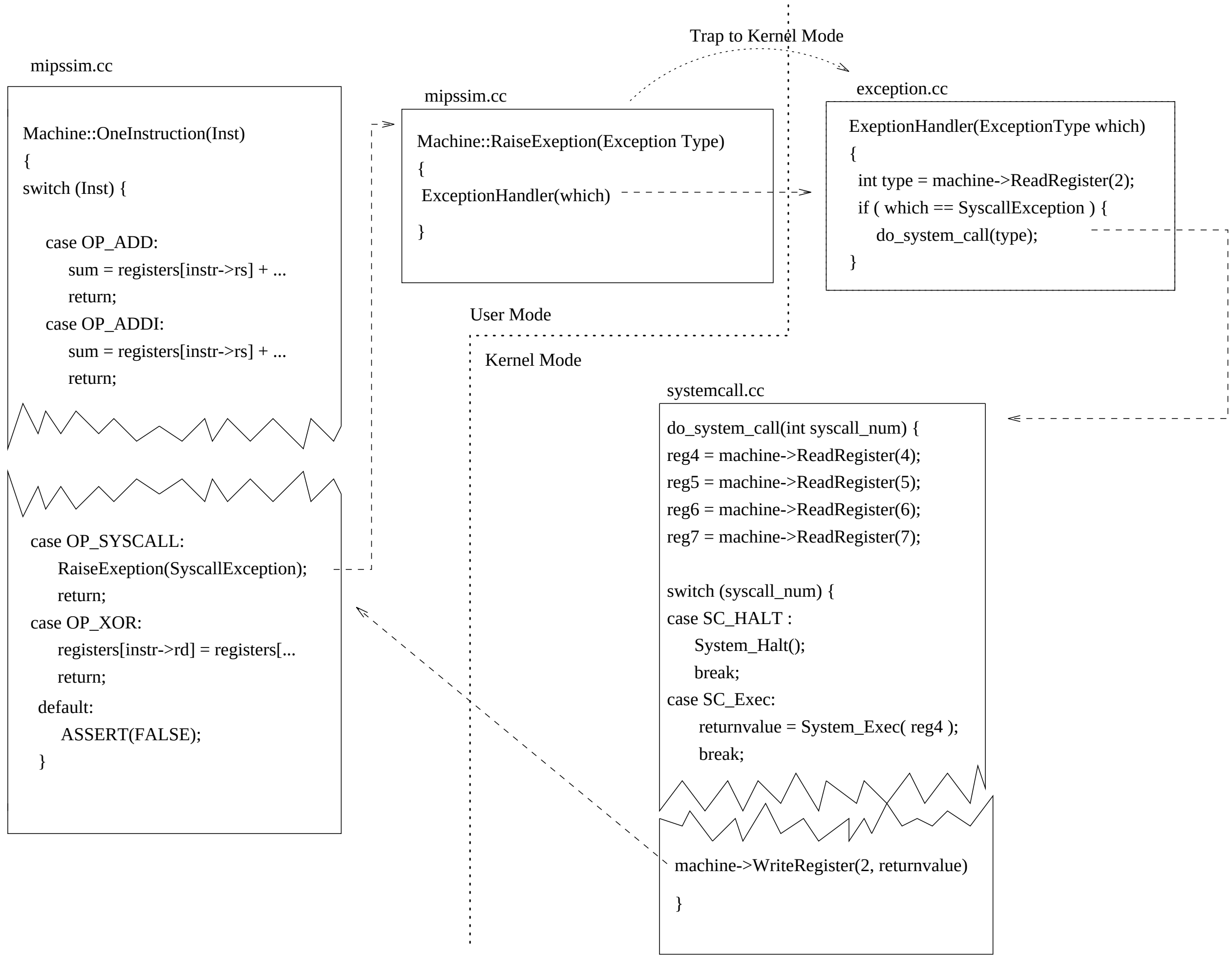

Figure 3: Trap into Kernel Mode

3 Main Memory

In the MIPS simulator, main memory is implemented as an array of bytes. This array is written to by the instruction *WriteMem()*, and is read by the instruction *ReadMem()*. Inside the machine object is a pointer to a page table, which is a table of translations from virtual addresses to physical addresses. An executing instruction which references a virtual address must first have that virtual address translated into a physical address via *Translate()* (which refers to the page table) before it can be executed.

This translation step during the fetch-execute cycle is shown below:

4 The Nachos Thread

A Nachos thread can be viewed as a diminutive process. Threads are created and destroyed with the *Fork* and *Exit* system calls, respectively. The thread object also contains a set of structures similar to a Unix processes PCB. The thread object is pictured below:

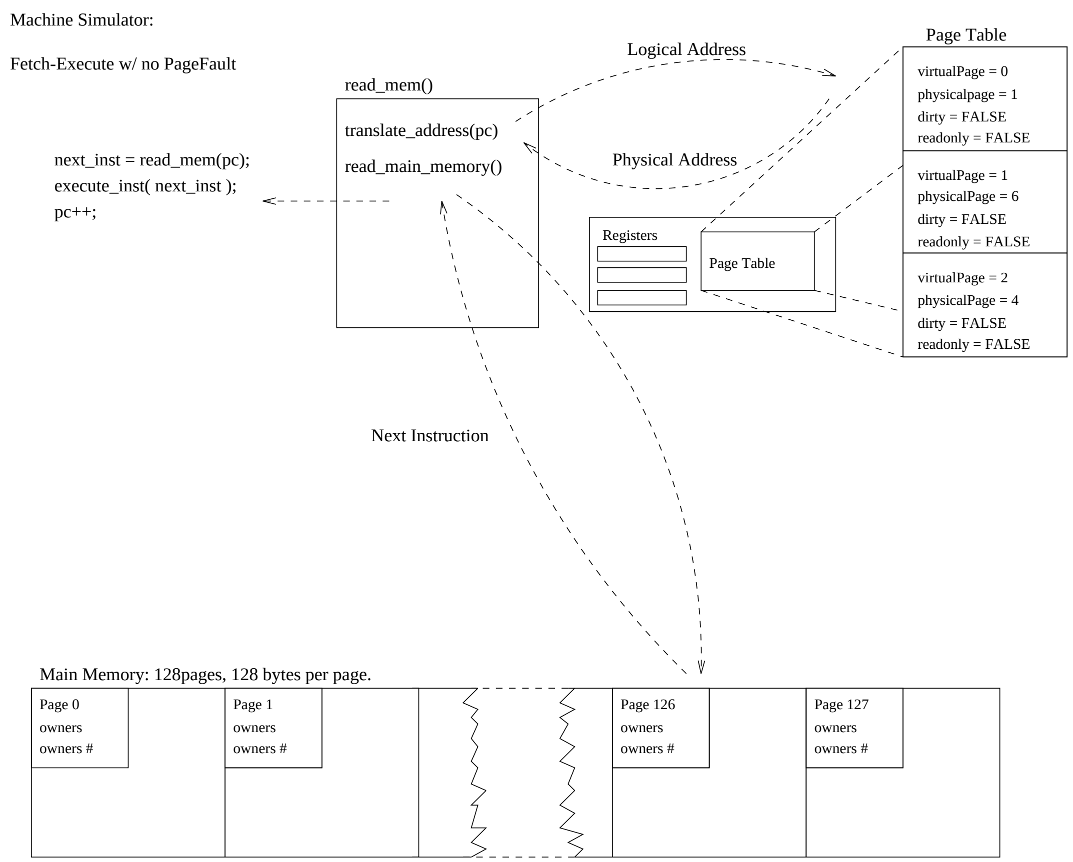

Figure 4: Address Translation

```
From: code/threads/thread.h (not complete)

class Thread {
public:
    FDTEntry *FDTable [MAX\_FD];
    int ThreadID;
    ThreadStatus status;

    unsigned int* stack;
    char name [MAXFILENAMELENGTH + 1];

    int userRegisters[NumTotalRegs];
    void SaveUserState();
    void RestoreUserState();
    AddrSpace *space;
    FDTEntry *getFD (int fd);
private:
    unsigned int machineState[MachineStateSize];
};
```

5 State/Context Switching

As we can see, the thread contains many of the same structures as a Unix PCB: a file descriptor table, an Id, and a current status.

The primary difference bewteen a Nachos thread and a Unix process is that a Nachos thread contains **two** sets of registers and two stack pointers. This is because the Nachos user-program runs on a MIPS processor, while the kernel runs on a host processor (in this case, an x86). For this reason, the individual thread needs **two** sets of everything to save its context (i.e. it needs to know where it was in it's user program **and** where it was in the kernel). It should be noted that this roughly corresponds to Unix, which maintains a user-space stack and a kernel stack for each process.

6 Multi-Programming

Just as a Unix user program executes inside a process, a Nachos user program executes inside a thread. The following sections describe the way in which nachos manipulates threads while in system mode to produce multi-programming and virtual memory.

Nachos uses a fixed quantum, round-robin scheduling algorithm. Thus, when one thread has been executing on the MIPS machine and its quantum expires, an interrupt is triggered (a simulated interrupt on the simulated machine), which causes a trap into system mode. In kernel space, the scheduler then forces the currently running thread to yield, and saves its state. The scheduler then finds the next available thread on the ready list, restores its state, and lets the simulated MIPS processor run again. This process is displayed in Figure 5, below.

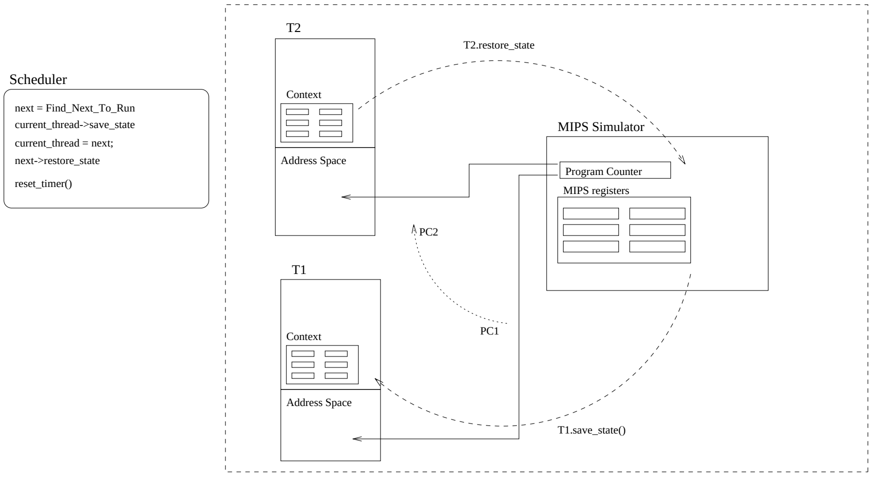

Figure 5: Context Switching

As described in the previous section, when a Nachos thread does a context switch it must save two sets of registers. This can be seen in the code for *Scheduler::Run()*, seen below. This function is usually called when the current thread's quantum has expired, and a new thread needs to be switched in.

```
void
Scheduler::Run (Thread *nextThread)
{
    Thread *oldThread = currentThread;

    if (currentThread->space != NULL) {
        currentThread->SaveUserState();
        currentThread->space->SaveState();
    }

    currentThread = nextThread;
    currentThread->setStatus(RUNNING);

    SWITCH(oldThread, nextThread);

    if (threadToBeDestroyed != NULL) {
        delete threadToBeDestroyed;
        threadToBeDestroyed = NULL;
    }

    if (currentThread->space != NULL) {
        currentThread->RestoreUserState();
        currentThread->space->RestoreState();
    }
}
```

This function is called with a pointer to the next thread to run. The scheduler then saves the current threads state (there are two calls here, the first one is for the user-state MIPS register, the second is to save the page table from the machine). We then set the new thread's status to running and call a machine language routine *SWITCH*. This saves the host machine registers associated with the first thread, and then restores the registers associated with the new thread.

It should be noted that, on return from *SWITCH*, a **different thread** is executing. If the switch is from Thread 1 to Thread 2, going into *SWITCH* Thread 1 is running, coming out of *SWITCH*, Thread 2 is now running, Thread 1 has been put back on the ready list, for resumption later (note: Thread 1's host PC still points to *SWITCH*, so that when it resumes later it will resume from there).

7 Running User Programs

A Nachos thread exists primarily to run user programs. To do this, the thread object has a pointer to an address space object, which has a page table inside it.

To run a user program, the thread must have control of the MIPS simulator. When this is the case, the thread's user program will be resident in main memory and the machine's program counter will be pointing to an instruction somewhere in that memory. Then the machine invokes Run(), which simply loops through a fetch-and-execute cycle, reading instructions from main memory and executing them until the threads quantum expires and the scheduler swaps it out. The user program execution process is diagramed below:

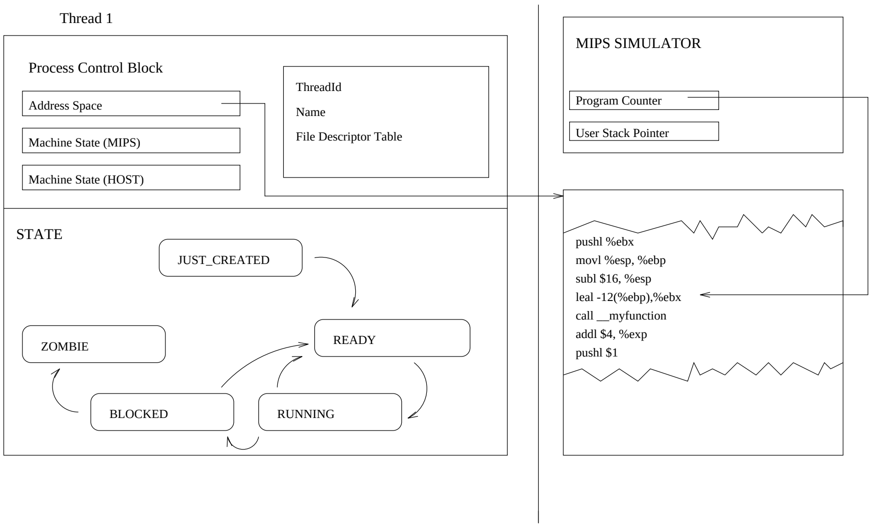

Figure 6: Executing a User Process

8 Memory Management

Nachos uses paging to give its threads contiguous logical address spaces. These logical address spaces are 128 kilobytes in size, even though the simulated MIPS processor on which they run has only 16 kilobytes of main memory. In order to accomplish this, a virtual memory scheme is used. In this scheme, pages of memory that are not in use are swapped out to a disk file, from which they are read back into memory when they are needed.

As with most modern computer systems, paging is achieved through the use of memory management unit (MMU). The *MemoryManager* and *Machine* objects together serve as the MMU of the nachos system. The *MemoryManager* object keeps track of which physical memory pages are free, which are used, and the owners of the used pages. As such, it is primarily involved in memory resource allocation. The *Machine* object, among other things, holds the page table of the currently running Nachos thread.

Since each Nachos thread is viewed as a process with its own contiguous logical address space, each thread needs its own page table. In Unix, a pointer to a process's page table is stored in its PCB. As described earlier, the Nachos equivalent of the PCB is the *Thread* object, which maintains all the information that the kernel needs to know about a thread. One of the items in the *Thread* object is a pointer to an *AddrSpace* object. This object is simply an abstraction of the thread's logical address space, and contains a pointer to the thread's page table. When the CPU is given to the thread, the page table in the *Machine* object is loaded using this pointer.

Figure 7 shows how page tables are used to give threads contiguous logical address spaces.

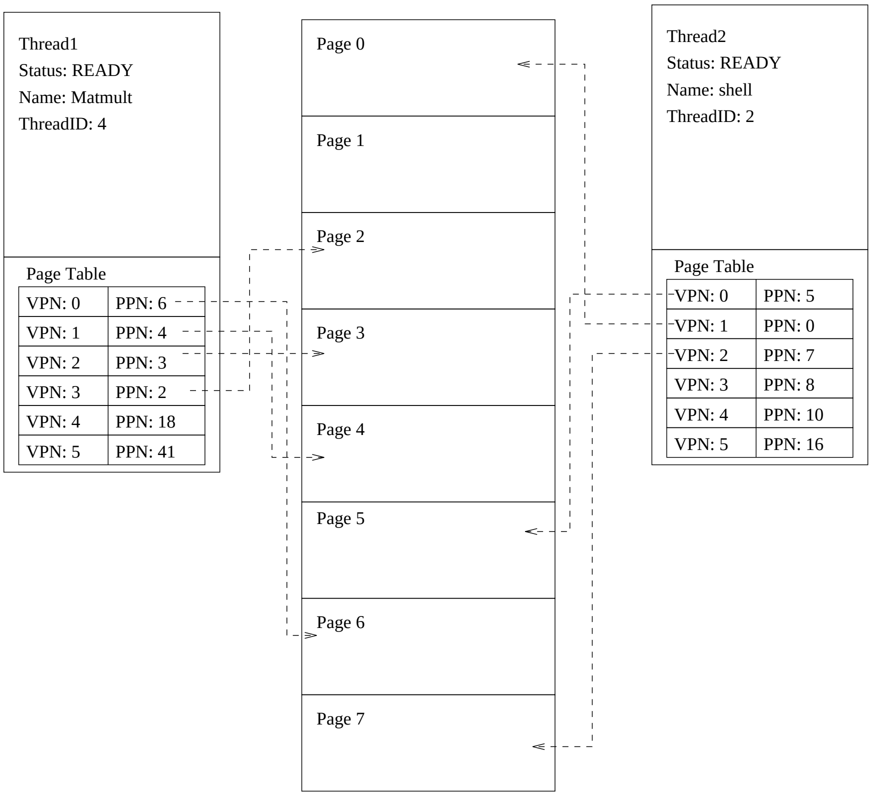

Figure 7: Non-contiguous Memory Allocation

The thread's page table is implemented as an array of *TranslationEntry* objects, with one *TranslationEntry* object for each page of the thread's logical address space. The *TranslationEntry* class is declared as follows:

```
class TranslationEntry {
  public:
    unsigned int virtualPage;
    unsigned int physicalPage;
    bool valid;
    bool readOnly;
    bool use;
    bool dirty;
    OpenFile* File;
    size\_t offset;
    bool zero;
    bool cow;           // Copy on write
};
```

The most obvious function of the *TranslationEntry* object is to hold the virtual to physical page mapping of a page. In addition, it also has a *valid* member whose value indicates whether the page is "valid" (i.e. has not been swapped out of main memory). It also contains members to indicate the read-only status of the page (program text pages are usually considered read-only), use of the page (set every time the page is referenced by the hardware), and whether or not the page has been modified (whether or not it is "dirty").

The **File** field points to a physical (Unix) file which contains the program text. This may be the swapfile, in the case of a program that has been paged, or the executable file from which the program came. **offset** then, refers to the page's offset within the file.

When a program makes a reference to a logical memory address, the machine invokes the *Translate()* function to get the corresponding physical address. *Translate()* finds the page table entry for the page containing the logical memory address, and checks is *valid* member. If the page is "valid", then *Translate* simply computes the physical address corresponding to the logical address it was passed, and this is then used to complete the original memory reference.If the page is found to not be "valid" (because it is not currently in main memory), then a page fault exception is generated. The exception handler handles the page fault by swapping the desired page back into main memory. The instruction that originally performed the memory reference is then restarted. This process is illustrated in Figure 8.

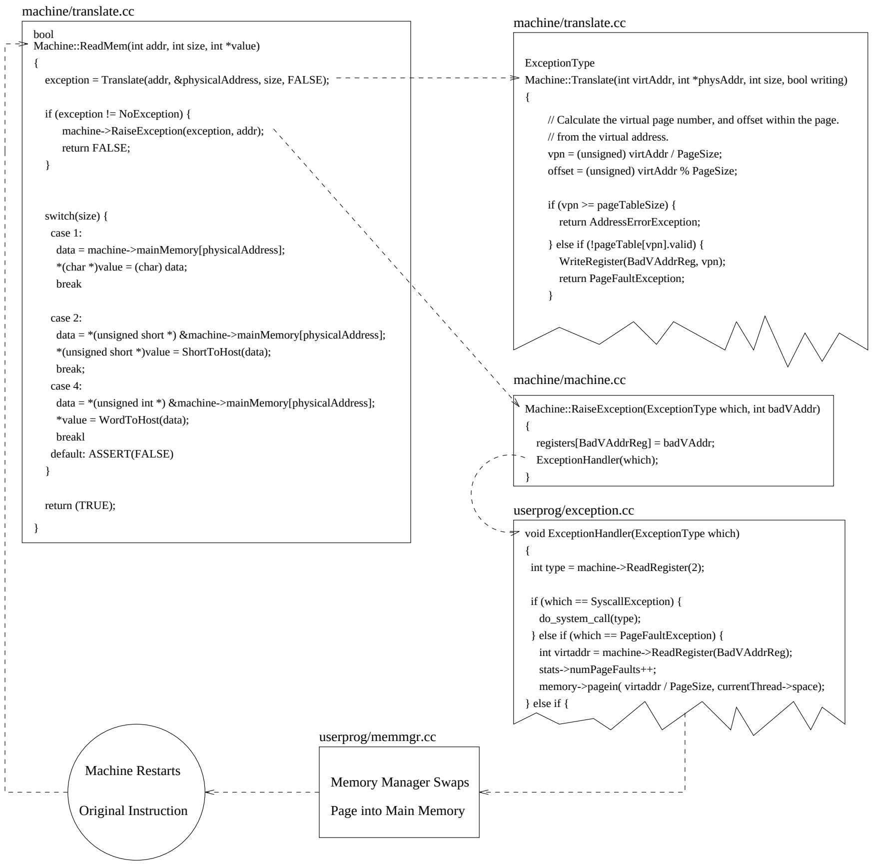

Figure 8: A Page Fault

9 Demand Paging

When a Nachos program is loaded, only its first and last pages are brought into main memory. The first page will be the first page of the program's text segment, and the last page will be its stack. Even though only two pages of the program are loaded into memory, page table entries are set up for all the program's pages. However, only two of these entries (the first and the last) will contain valid logical to physical page mappings. The others will all have their "valid" bits set to false.

Thus, when the program makes a reference to a page that has not been loaded, it will find the valid bit of its page table entry set to false. A page fault exception will be generated as described in the previous section.

Nachos handles page faults by loading the referenced page into main memory, and restarting the instruction that generated the page fault. Pages are swapped in using the *MemoryManager::pagein()* function. In order to swap a page in, this function may need to swap another page out (if there are no free pages). This is done using the *MemoryManager::pageout()* function.

The file from which a page is read when being swapped into memory will vary depending on the nature of the page. The *TranslationEntry* object of each page contains a pointer to an *OpenFile* object. This member points to the file from which the page can be read if it gets swapped out disk and has to be swapped back into main memory. This will point to the swap file, if the page had been modified before it was swapped out, or the original executable file if it has not been modified. If the page contained was contained only uninitialized data, the pointer to the *OpenFile* object is set to NULL (since the page would contain only zeros, and would not need to be read from a disk file when swapped back into main memory).

10 Concluding Remarks

As of this writing, Nachos incorporates most of the standard operating system features such as multi-tasking, preemptive scheduling and virtual memory. While these systems are functional and reasonably stable, they are certainly not optimal. During most of our coding, we have tried to give clarity a higher priority than optimal performance. We hope this will aid you in your Nachos programming endeavors.