System project,
M1, 2011-2012

NACHOS STEP 4 : VIRTUAL MEMORY

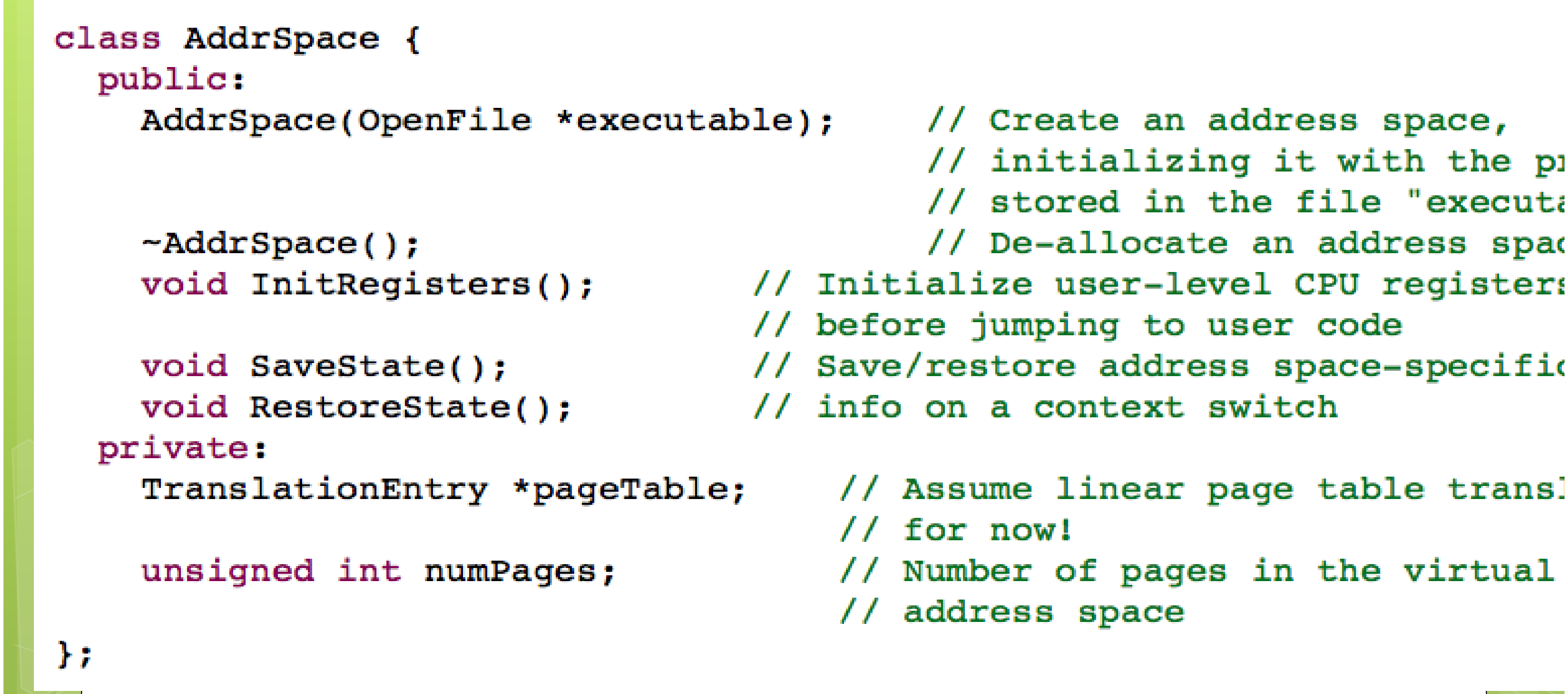
V.Marangozova, V.Danjean

Virtual Addresses

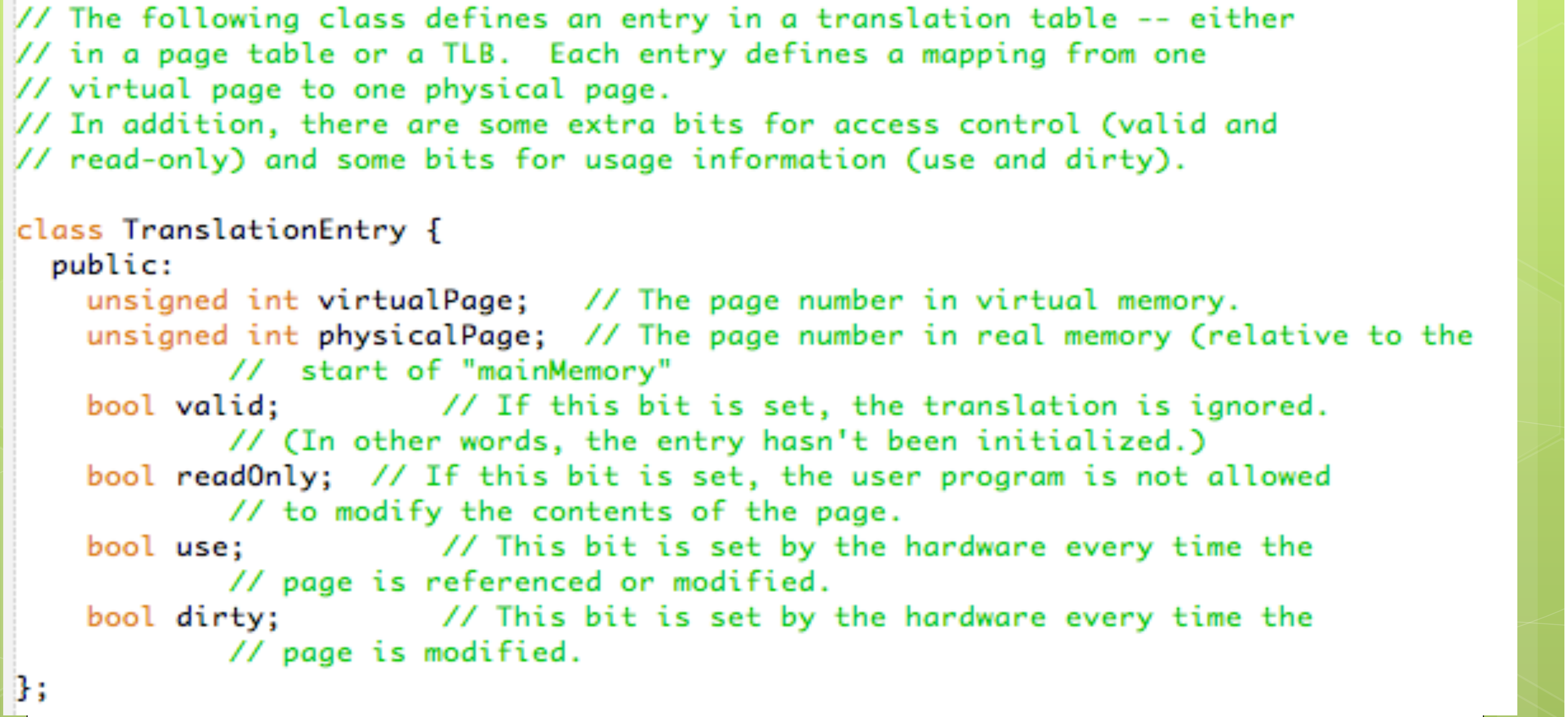
- The AddressSpace of a process work with virtual addresses
  - From 0 to size (size of the address space)
  - The address space of a process is lso called its « virtal memory »
  - When the process is loaded into memory, the virtual addresses are to be translated into physical addresses
    - The translation is done during execution
    - When there is a memory access, the MMU trabslates it into the corresponding physical address
    - The translation depends on the memory management

NACHOS : MIPS Virtual Memory

- MIPS uses one of these two
  - Page table
  - TLB
- In our project, we will work only with the page table
- Do you remember what paging is and how it works?



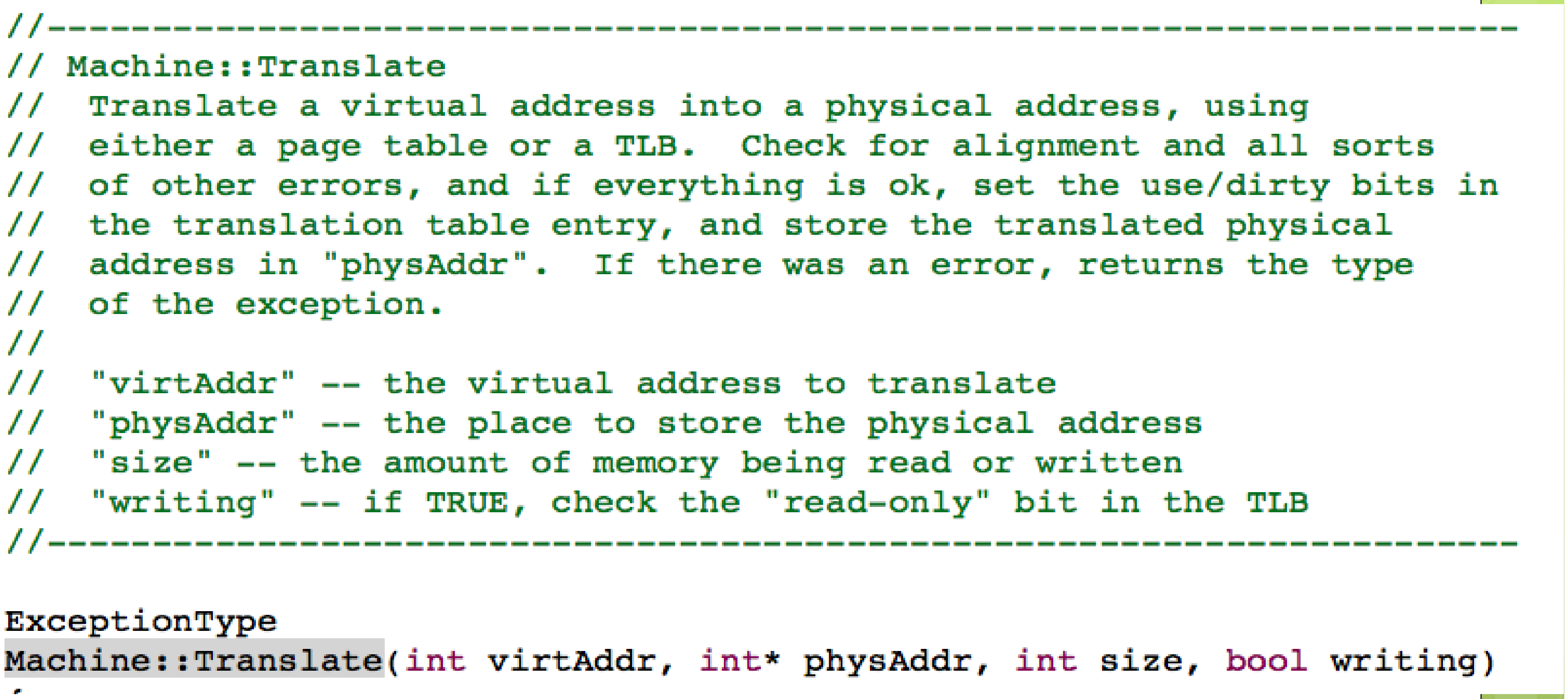
```
class AddrSpace {
  public:
    AddrSpace(OpenFile *executable);    // Create an address space,
                                        // initializing it with the pr
                                        // stored in the file "executa
    ~AddrSpace();                       // De-allocate an address spac
    void InitRegisters();       // Initialize user-level CPU register
                                // before jumping to user code
    void SaveState();           // Save/restore address space-specifi
    void RestoreState();        // info on a context switch
  private:
    TranslationEntry *pageTable;    // Assume linear page table transl
                                    // for now!
    unsigned int numPages;          // Number of pages in the virtual
                                    // address space
};
```



```
// The following class defines an entry in a translation table -- either
// in a page table or a TLB.  Each entry defines a mapping from one
// virtual page to one physical page.
// In addition, there are some extra bits for access control (valid and
// read-only) and some bits for usage information (use and dirty).

class TranslationEntry {
  public:
    unsigned int virtualPage;   // The page number in virtual memory.
    unsigned int physicalPage;  // The page number in real memory (relative to the
            //  start of "mainMemory"
    bool valid;             // If this bit is set, the translation is ignored.
            // (In other words, the entry hasn't been initialized.)
    bool readOnly;  // If this bit is set, the user program is not allowed
            // to modify the contents of the page.
    bool use;               // This bit is set by the hardware every time the
            // page is referenced or modified.
    bool dirty;             // This bit is set by the hardware every time the
            // page is modified.
};
```

From virtual to physical addresses

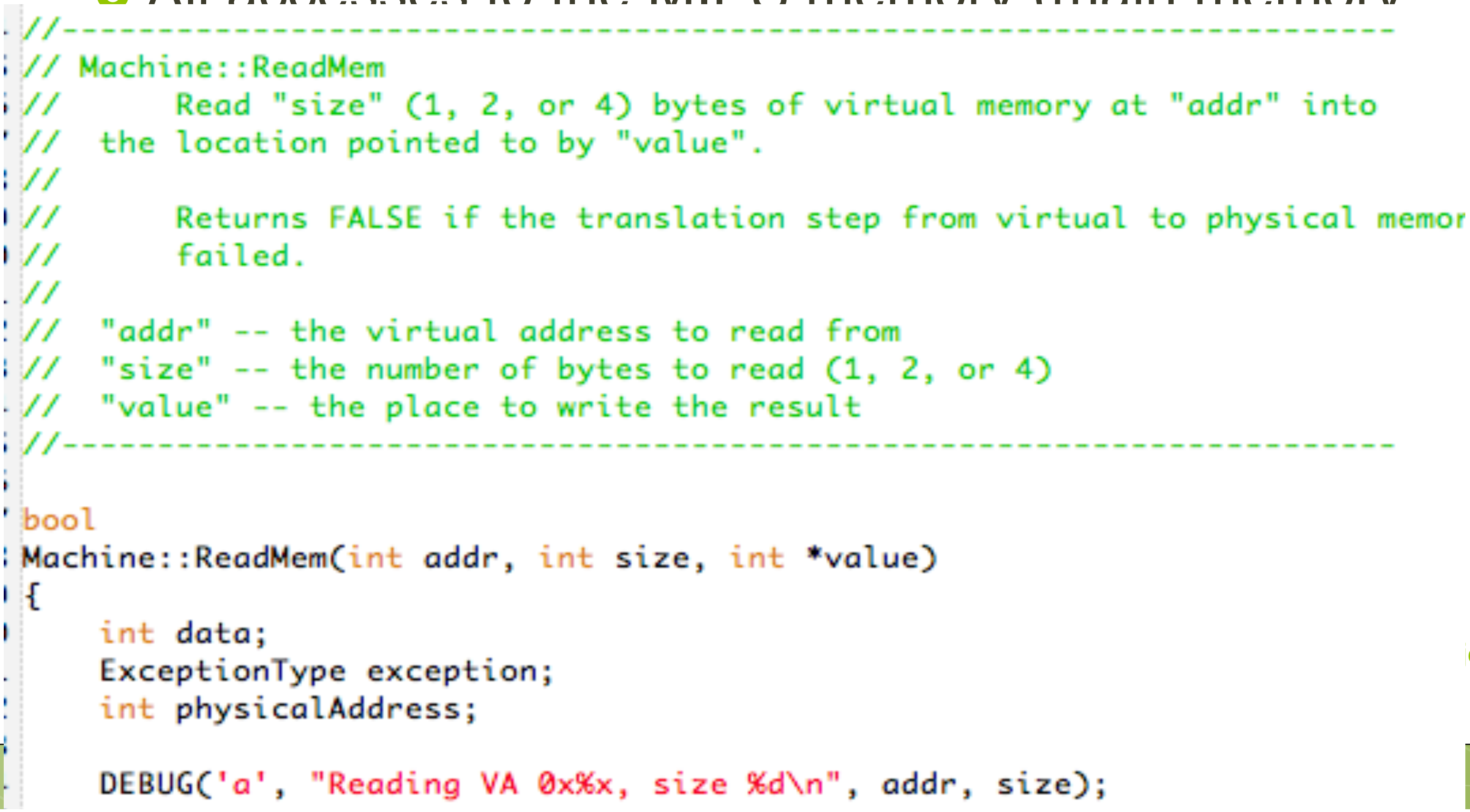
- What is the address format?
- How do we calculate?
- Implemented in `machine/translate.cc`
  -
```
Translate(   int virtAddr,
             int* physAddr,
             int size,
             bool writing)
```



```
//----------------------------------------------------------------------
// Machine::Translate
//  Translate a virtual address into a physical address, using
//  either a page table or a TLB.  Check for alignment and all sorts
//  of other errors, and if everything is ok, set the use/dirty bits in
//  the translation table entry, and store the translated physical
//  address in "physAddr".  If there was an error, returns the type
//  of the exception.
//
//  "virtAddr" -- the virtual address to translate
//  "physAddr" -- the place to store the physical address
//  "size" -- the amount of memory being read or written
//  "writing" -- if TRUE, check the "read-only" bit in the TLB
//----------------------------------------------------------------------

ExceptionType
Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
```

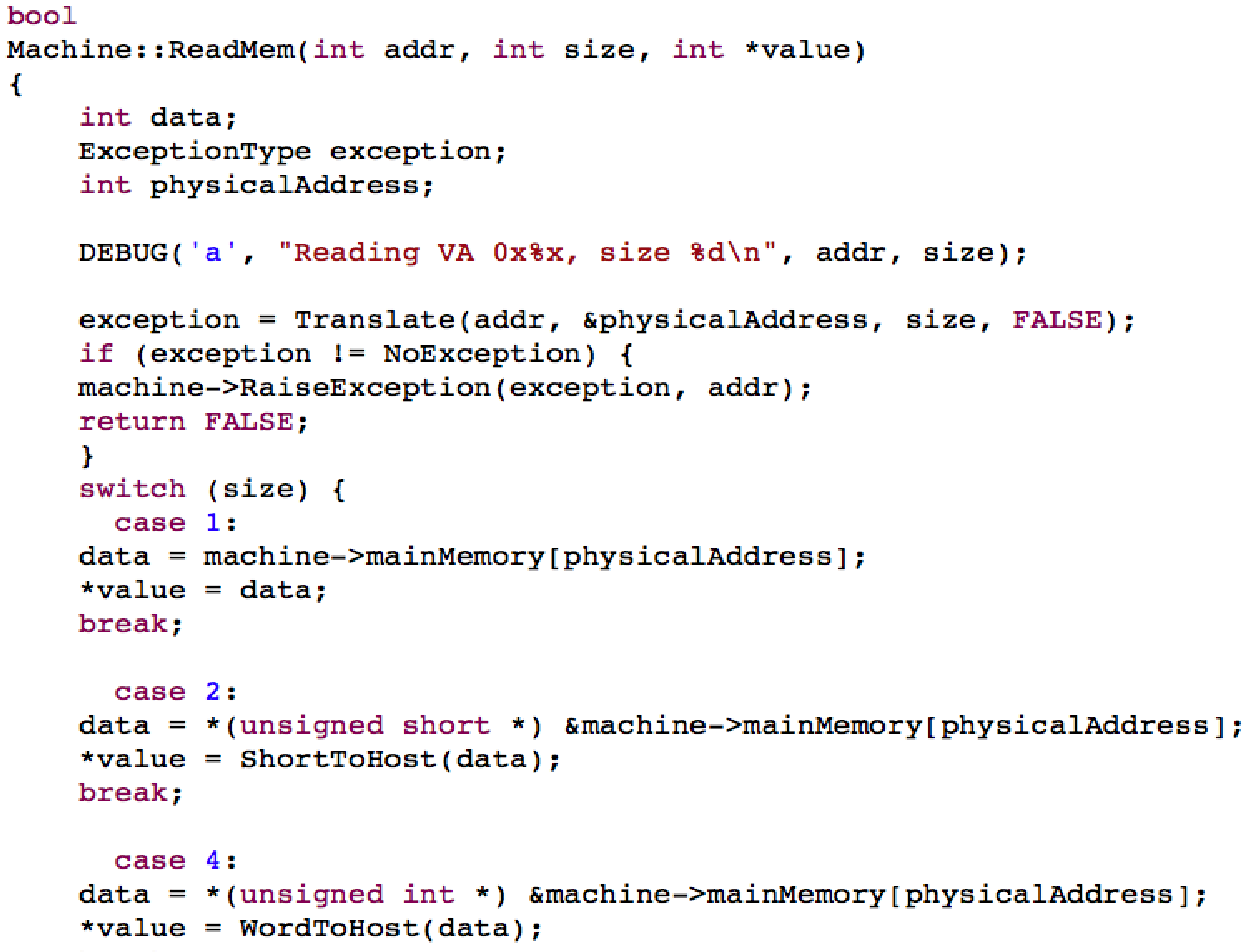
Memory Access

- All accesses to the MIPS memory (main memory



```
//----------------------------------------------------------------------
// Machine::ReadMem
//      Read "size" (1, 2, or 4) bytes of virtual memory at "addr" into
//  the location pointed to by "value".
//
//      Returns FALSE if the translation step from virtual to physical memor
//      failed.
//
//  "addr" -- the virtual address to read from
//  "size" -- the number of bytes to read (1, 2, or 4)
//  "value" -- the place to write the result
//----------------------------------------------------------------------

bool
Machine::ReadMem(int addr, int size, int *value)
{
    int data;
    ExceptionType exception;
    int physicalAddress;

    DEBUG('a', "Reading VA 0x%x, size %d\n", addr, size);
```

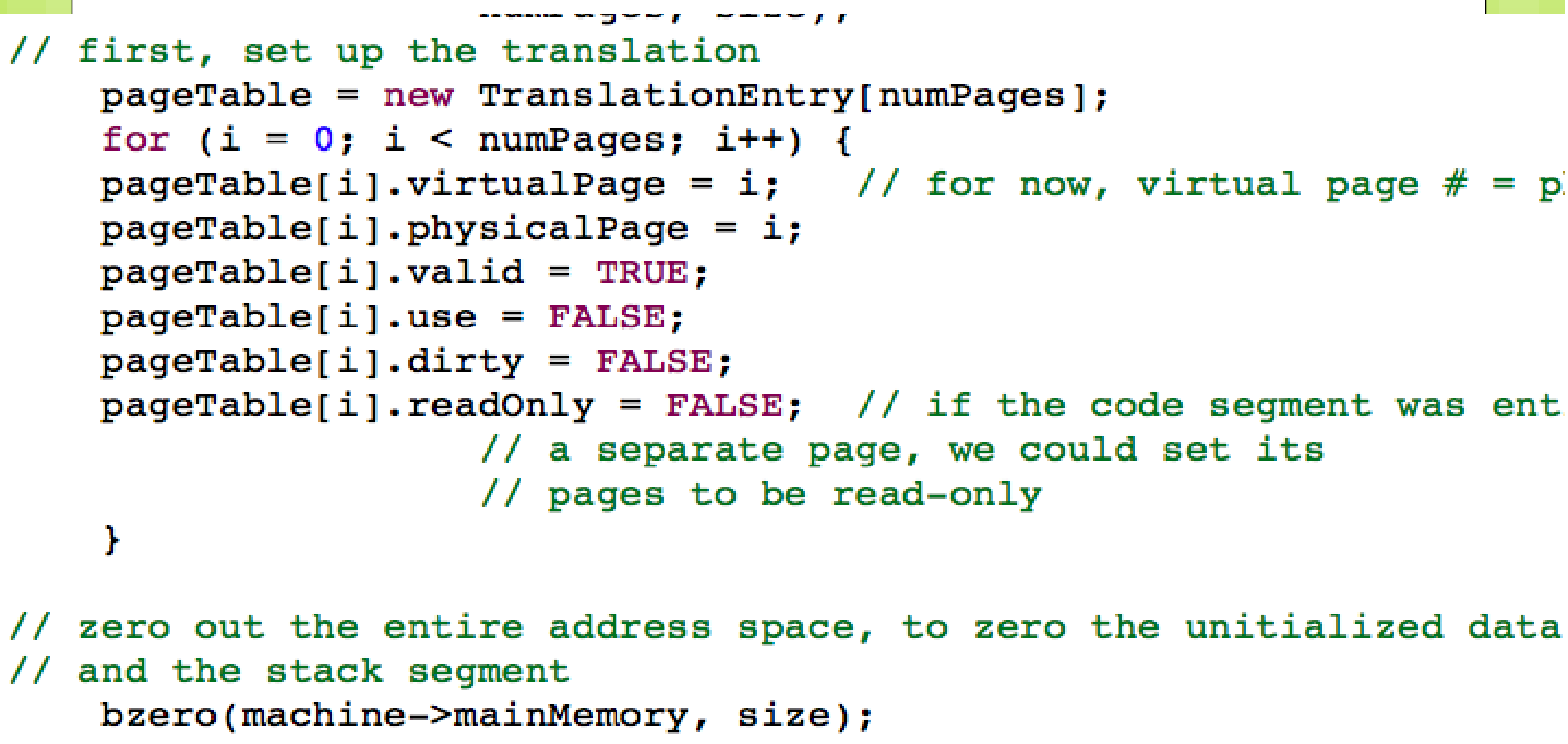
ean



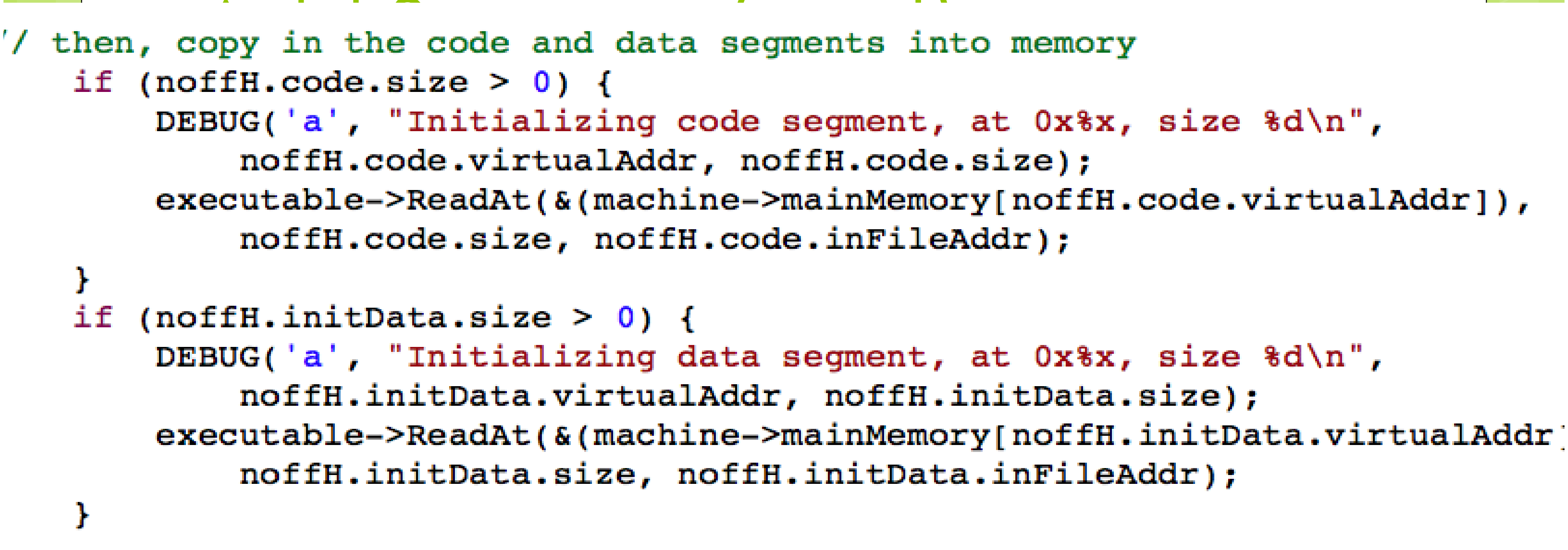
```
bool
Machine::ReadMem(int addr, int size, int *value)
{
    int data;
    ExceptionType exception;
    int physicalAddress;

    DEBUG('a', "Reading VA 0x%x, size %d\n", addr, size);

    exception = Translate(addr, &physicalAddress, size, FALSE);
    if (exception != NoException) {
    machine->RaiseException(exception, addr);
    return FALSE;
    }
    switch (size) {
      case 1:
    data = machine->mainMemory[physicalAddress];
    *value = data;
    break;

      case 2:
    data = *(unsigned short *) &machine->mainMemory[physicalAddress];
    *value = ShortToHost(data);
    break;

      case 4:
    data = *(unsigned int *) &machine->mainMemory[physicalAddress];
    *value = WordToHost(data);
```

For now, everything happens in physical memory directly…

- Let's go and analyse AddressSpace



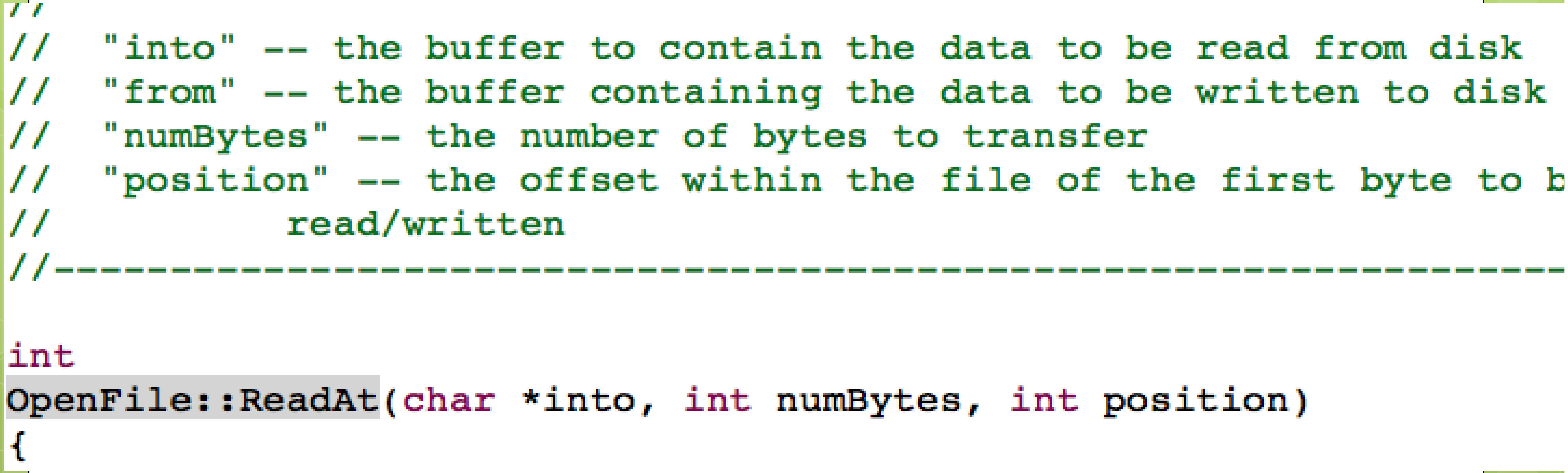
```
                    numPages, size);
// first, set up the translation
    pageTable = new TranslationEntry[numPages];
    for (i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;   // for now, virtual page # = p
    pageTable[i].physicalPage = i;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;  // if the code segment was ent
                    // a separate page, we could set its
                    // pages to be read-only
    }

// zero out the entire address space, to zero the unitialized data
// and the stack segment
    bzero(machine->mainMemory, size);
```



```
'/ then, copy in the code and data segments into memory
    if (noffH.code.size > 0) {
        DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
            noffH.code.virtualAddr, noffH.code.size);
        executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
            noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
            noffH.initData.virtualAddr, noffH.initData.size);
        executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]
            noffH.initData.size, noffH.initData.inFileAddr);
    }
```

What does ReadAt?

- `AddrSpace::AddrSpace` in `userprog/addrspace.cc`



```
//
//   "into" -- the buffer to contain the data to be read from disk
//   "from" -- the buffer containing the data to be written to disk
//   "numBytes" -- the number of bytes to transfer
//   "position" -- the offset within the file of the first byte to b
//           read/written
//----------------------------------------------------------------------

int
OpenFile::ReadAt(char *into, int numBytes, int position)
{
```

One Subtask

-
  ```
  static void ReadAtVirtual(
      OpenFile *executable,
      int virtualaddr,
      int numBytes,
      int position,
      TranslationEntry *pageTable,
      unsigned numPages)
  ```
- Same as ReadAt but using virtual memory(pageTable and numPages).
- You could use a buffer, filled with ReadAt and then copied in memory using WriteMem

FrameProvider

- You should allocate physical pages for your processes
  - Implement FrameProvider.
  - Use de BitMap
- Bzero the memory in this class

Multiprogramming

- Syscall
  - int ForkExec(char *s)
  - Executable as parameter,
  - Creates a nachos thread (a main of another process),
  - in parallel with the current thread
  - Note that the new process could create threads!
- What about AddrSpace?

Points to treat…

- If there are multiple processes, the machine should not Halt at the end of the first process that terminates
- You should take care to free the memory
- mini shell