A Method of Implementing Paged, Segmented Virtual Memories on Microprogrammable Computers

Tanenbaum, A.S.

published in
ACM SIGOPS Operating Systems Review
1979

document version
Publisher's PDF, also known as Version of record

A Method for Implementing Paged, Segmented Virtual Memories on Microprogrammable Computers

by

Andrew S. Tanenbaum
Computer Science Group
Vrije Universiteit
Amsterdam, The Netherlands

1. INTRODUCTION.

A large, segmented, paged, virtual memory [1-4] makes the programming of certain applications easier. There are several reasons why such a two dimensional address space is helpful. First, the programmer need not waste any effort trying to cram a large program into a small memory. Second, if one procedure is changed, it is only necessary to relink the segment to which it belongs, and not any other segments. Third a protection system such as the MULTICS ring structure [5] can be implemented to increase security and trap certain kinds of errors. In addition to the above reasons, which apply primarily to the program, there are also advantages that accrue from having many large data segments as well.

A virtual address on a computer with a segmented virtual memory consists of two parts: the segment number and the address within the segment. If the segments are large, it will be impossible to keep an entire segment in memory at one time, so the segment will have to be divided into pages. Note that paging is an implementation convenience with which the programmer need not be concerned, whereas segmentation is a logical partitioning of the address space that is visible to the programmer. Each virtual address implicitly specifies a segment number, a page number, and an offset within the page.

In the usual implementation, the hardware recognizes a special descriptor segment, each of whose entries points to the page table for the corresponding segment. Each page table tells where in primary memory the pages comprising that segment can be found. Of course, not all pages and page tables need be in primary memory at all times. When the hardware is given the virtual address of a word (or byte) to be read, the following steps occur (conceptually). The segment number is used as an index into the descriptor segment to find the page table. Then the (virtual) page number is used as an index into the page table to fetch the (physical) address of the page frame where the requested page is located. From the physical address where the page frame starts and the offset within the page, the physical address of the needed word can be computed. The physical address is then fed to the memory subsystem to read the word. Many of the irrelevant details have been omitted here; see [4] for the full story.

The process of converting a virtual address to a physical one is complicated, as shown in Fig. 1. Remember that every instruction requires at least one read (to fetch the instruction itself) and possibly 1 or more reads and/or writes for data as well. On a microprogrammed computer the virtual memory could be implemented by simply putting the function virtualtophysical in the control store, and having it carried out step by step on every memory access.

If the virtual memory actually were implemented in firmware as suggested by Fig. 1, the machine would be horribly slow. To achieve speed, computers with paged, segmented virtual memories are equipped with an (expensive) associative memory that remembers the location of the last n pages referenced. Given a (segment, page) pair, it can very quickly find the page origin if the page happens to be among the n most recently referenced pages. If the page has not been used recently, virtualtophysical must be invoked.

In summary, if the computer does not have an associative memory it will be slow, and if it does have an associative memory it will be expensive. For applications where a microprogrammable mini- or microcomputer has been chosen due to its excellent price performance ratio, neither of these alternatives (low performance or high price) is attractive. Nevertheless, having a large segmented virtual memory is desirable, and it would be nice if a method could be found to implement it without any special hardware, and at a performance equal to or only slightly less than the same machine without the virtual memory feature. Such a method is described in the following section.

2. THE ALGORITHM

At any instant in time, the program counter points to the next instruction word to be fetched. The page in which this instruction resides will be called the current code page. Since most instructions are not jumps, the program counter usually advances up the page more or less linearly, increasing by one instruction length per instruction execution time. Consequently, once a page has become the current code page, it tends to remain so for a number of consecutive instructions.

Eventually, however, a new page will become the current code page. We will call this a page change. Note carefully that a page change is quite different from a page fault. The former occurs when the program counter moves from one page to another; the latter occurs when a needed page, be it a program page or a data page, is not in primary memory and must be fetched from secondary memory. A page change may cause a page fault, but it need not, since the new page may already be in primary memory.

Page changes can be initiated for a variety of reasons: the program counter can simply advance off the end of the page; an off page procedure can be called or returned from; a jump address may be off page; finally a

trap or interrupt may cause the program counter to suddenly take on a distant value. The key to the method lies in the fact that the number of instructions executed between page changes is relatively large. In other words most instructions do not cause a page change.

The basic idea is as follows. The microprogram maintains three variables in its scratchpad memory: pc (the physical address of the next instruction word to be fetched), pclimit (the physical address of the last word on the current code page) and pclimitvirtual (the virtual address corresponding to the last word on the current code page). Since pc is a physical and not a virtual address, the next instruction word can be fetched by simply delivering pc to the memory hardware. It is not necessary to invoke virtualtophysical. So far, at least, there is no penalty at all for having a virtual memory.

Unfortunately there is no such thing as a free lunch. Once in a while pc will stray off the current code page. If we did not check for this condition, the program would execute incorrectly, since there is no particular relation between virtual pages and physical page frames; the next highest virtual page is generally not located in the next page frame. To detect the page change condition, the main loop of the microprogram (the one that fetches the instructions, increments the program counter, and then dispatches to the execution routines for the various opcodes) should begin with this statement:

```
if pc > pclimit then pagechange;
```

Since pc and pclimit are both kept in microprogram scratchpad registers, the needed statement amounts to comparing two internal registers, branching if the first is larger than the second. Usually one or two micro instructions should be sufficient, which is fortunate, since the test has to be made every time an instruction word is fetched. If an instruction occupies multiple words, the test must be repeated before each one is fetched.

Now let us consider how the microprocedure pagechange works. For the case of pc merely crawling across the upper page boundary, the virtual program counter has the value pc - pclimit + pclimitvirtual. Starting from this virtual address and the known current segment number (which is not changed by crawling over a page boundary), virtualtophysical is called to compute the physical address where the pc should actually be pointing.

If the new code page is in primary memory, pclimit and pclimitvirtual can be quickly updated and execution can continue on the new page. If the new page is not in primary memory, the page fault sequence is initiated, and the operating system is started to handle the fault. Notice that we have managed to reduce calls to virtualtophysical from one per instruction word, to one per page change.

For the case that pc does not crawl over the edge, but rather changes suddenly via a jump, call, trap etc., we proceed as follows. First

consider the subcase of instructions that stay within the current segment and specify the jump or call address by giving the target distance relative to the current instruction, rather than by giving an absolute address. If the distance, d, jumped is positive, it can merely be added to pc and the main loop restarted. Whether or not a page change is needed, the test in the main loop will work correctly.

If, however, the jump is backwards (d < 0) the microprogram must check to see if pc + d is within one page of pclimit. If so, there is no page change and the execution is the same as for forward jumps, calls, etc. If there is a page change, the jump, call etc. microcode must itself compute the new virtual address and then update the three registers, presumably by calling pagechange. Note that although machine instructions generated from structured programs may contain many short forward jumps (from the if statements) backward jumps only result from loops, with only one per loop, and from calls, returns, traps and interrupts.

Now we must consider the worst case: a discontinuous change in pc caused by an instruction specifying an explicit (i.e. absolute rather than relative) address. The only alternative is to compute the physical address using virtualtophysical and then to update all three variables.

It should be clear by now that the choice of instructions in the target machine's repertoire has a significant effect on efficiency. Both jumps and calls should use relative rather than absolute addressing (the PDP-11, for example allows all addresses to be relative). Furthermore, having distinct instructions for forward and backward eliminates the need to make a run time test on the sign of the distance, d.

3. VIRTUAL MEMORY FOR DATA

Up until now we have only considered the virtual memory for program segments. The method described above works because programs have the property that page changes are relatively infrequent. With data references the situation can be radically different. Depending on the language being used and the target machine architecture, it is quite possible that data references are spread out over a substantial number of pages. If this is true, the only way out is to separate the instruction space from the data space (as on the PDP-11/45) and provide a segmented virtual memory for instruction space, and an unpaged, unsegmented, permanently "wired down" address space for the data. Although not perfect, it is still better than forcing both the program and the data into a single, tiny address space.

However, for one particular class of target machine architecture, there is still hope: stack machines (e.g. Burroughs B6700). On a stack machine, at any given instant the stack pointer points to a word on a page we will call the current stack page. The machine may well use different stacks in different segments, at different times (e.g. for multiprogramming), but once a given stack page becomes current, it tends to remain so

for a substantial number of machine instructions. Under these conditions we can proceed in a manner analogous to the algorithm given for program pages.

When a stack machine is used to run programs written in block structured languages (e.g. PASCAL, ALGOL 68, ALGOL 60, PL/I, etc.) every time a procedure (or block) is entered, space is reserved on the stack for its actual parameters and local variables. This is accomplished by simply advancing the stack pointer, sp, by the appropriate amount. When the procedure is exited, the stack pointer is decremented again. Parameters and local variables are addressed by giving their offset from the local base register, lb, which points to the first word allocated to the procedure.

Arithmetic is performed on stack machines by first pushing the operands on to the top of the stack (above all the local variables). The arithmetic instructions fetch their operands from the top of the stack, and place their results where the operands previously had been. Most of the details of how stack machines work are irrelevant for our purposes except the observation that most data references will either be offset a small, positive amount from lb (parameters and locals), or be in the general vicinity of sp (arithmetic operands).

First we will show how pushing and popping operands from the top of the stack can be handled, and then we will deal with parameters, locals and other variables. The microprogram needs 4 internal scratchpad variables: sp (the physical address of the top of the stack), splimit (the physical address of the last word on sp's page), splimitvirtual (the virtual address of splimit), and splolomit (the physical address of the first word of the current stack page). The first three of these are analogous to their pc counterparts.

Any instruction that pushes an operand onto the (upward growing) stack must check for the condition sp > splimit. Likewise, any instruction that pops an operand off the stack must check for sp < splolimit. Push type instructions need not check splolimit since stack contraction off the bottom of the page is impossible; pop type instructions need not check splimit since stack expansion off the top of the page is impossible. Only one register-register comparison and conditional branch is needed per stack reference. Stack page changes are handled in exactly the same way as code page changes.

References to parameters, which, in fact, are very much like initialized local variables, and local variables themselves, are handled in a similar way, with variables lb, lblimit, and lblimitvirtual. When the machine tries to access local variable n, the value of lb+n is computed and compared to lblimit. If it is greater page change is not invoked, (lb has not moved off page) but instead virtualtophysical is called. Note that lb page changes can only occur as a result of procedure calls, returns, traps, and interrupts, but not as a result of push, pop, branch or arithmetic instructions. The fact that all parameters and locals have offsets that are

positive with respect to lb means that nothing analogous to splolimit is needed.

References to variables declared in outer lexicographical levels cannot be handled so easily. It may be possible to use an ad hoc solution in some cases, such as setting a certain flag whenever the entire stack is on the sp page. In other cases, virtualtophysical will have to be called. Fortunately, references to variables declared at intermediate lexicographical levels are relatively rare. If many references to variables declared at the outermost scope level are to be expected (see [6] for why this is to be discouraged), the microprogrammer might even consider having a fourth special page for the bottom of the stack.

It is quite clear that the technique is less satisfactory for the data than for the program. This is simply a reflection of the fact that data references have more scatter than program references. Nevertheless just as the working set concept [7] is on paper problematical, but in practice quite useful, if programs are well behaved, we believe this technique will also be useful for a substantial class of programs.

It is probably worth pointing out that this technique is not restricted to so called "microprogrammable" computers, but may also be useful in other situations in which one machine performs an instruction by instruction interpretation of another machine's instructions.

4. REFERENCES

[1] Corbato, F.J. and Vyssotsky, V.A.: "Introduction and Overview of the MULTICS system," FJCC 27, Spartan Books, pp. 185-197, 1965.

[2] Bensoussan, A., Clingen, C.T., and Daley, R.C.: "The MULTICS Virtual Memory: concepts and Design," CACM 15, pp. 308-315, May 1972.

[3] Daley, R.C., and Dennis, J.B.: "Virtual Memory, Processes, and sharing in MULTICS," CACM 11, pp. 306-312, May 1968.

[4] Organick, E.I.: The MULTICS System, Cambridge, Mass., MIT Press 1972.

[5] Schroeder, M.D., and Saltzer, J.H.: "A hardware Architecture for Implementing Protection Rings," CACM 15, pp. 157-170, March 1972.

[6] Wulf, W. and Shaw, M.: "Global Variable Considered Harmful," SIGPLAN Notices 8, pp. 28-34, February 1973.

[7] Denning, P.J.: "The Working Set Model For Program Behavior," CACM 11, pp. 323-333, May 1968.

```
type segnr = 0..highseg;                {highseg is largest segment number}
     pagenr = 0..highpage;              {highpage is number of pages per segment-1}
     offsetnr = 0..highoffset;          {highoffset is size of page-1}
     physicaladdress = 0..highcore; {highcore is largest address in memory}

     dsentry = record                   {descriptor segment entry}
                  pt: ↑pagetable;
                  highestpage: pagenr;
                  pagetableabsent, accessprohibited: boolean
               end;

     ptentry = record                   {page table entry}
                  frameorigin: physicaladdress;
                  pageabsent: boolean
               end;

      descriptorsegment = array [segnr] of dsentry;
      pagetable = array [pagenr] of ptentry;
      word = {primitive machine word};

var ds: descriptorsegment;
    memory: array [physicaladdress] of word; {primary memory}

function virtualtophysical (seg:segnr; page:pagenr; offset:offsetnr):
                                                          physicaladdress;
var d: dsentry
    p: ptentry;
begin {return the physical address corresponding to a virtual one}
   d := ds[seg];                        {fetch descriptor for this segment}
   if d.accessprohibited                {check for protection violation}
      then protectionfault
      else if page > d.highestpage  {check for length violation}
              then lengthfault
              else begin
                   if d.pagetableabsent then fixpagetablefault;
                   p := d.pt↑ [page]; {p is page table entry}
                   if p.pageabsent then fixpagefault;
                   virtualtophysical := p.frameorigin + offset
                end
end;
```

Fig. 1. An outline of how virtual addresses are converted to physical ones.