Overview of VAX Virtual Memory Management

Jeff Brandon

April 15, 2014

CPS 470: Introduction to Operating Systems

Abstract

The goal of this paper is to summarize the report of Henry M. Levy and Peter H. Lipman on the virtual memory management system they helped design and implement for the VAX/VMS operating system. An overview is given of the virtual address space, pager and swapper functions, page replacement algorithms, and how disk access is minimized. By reading this document one should attain a general understanding of the virtual memory system implemented by Levy, and Lipman.

1 Introduction

Henry M. Levy and Peter H. Lipman wrote their seminal paper working for Digital Equipment Corporation or DEC. While working there, Levy was a member of a team responsible for designing and implementing VAX/VMS. Lipman was also a primary designer and implementer of the system and was more specifically responsible for the memory management of the VMS release 1. Their report is on their design decisions and their implementation methods and why certain techniques were chosen over others.

2 Virtual Address Space

The first thing to understand about the Vax virtual memory system is how the virtual address space is divided. Each virtual address is comprised of 32 bits [1, pp 35]. The page size is 512

bytes so the least significant 9 bits are used as the page offset. The following 21 bits are used to identify the page and are called the page number. This leaves the most significant 2 bits available for determining the region of the process. If the most significant bit is set, the referenced process is a kernel level process (also referred to as system level) and the second most significant bit can be ignored because it is unused. If the most significant bit is not set then the process being referenced is user level and the second most significant bit is used to determine what region of the process is being referenced. The two regions of user processes are referred to by Levy and Lipman as **P0** and **P1**.

2.1 Kernel Processes

The system level address space is shared by all user processes. While each process needs its own space for its execution, each process shares the same system section. This reduces overall memory needed for a process. This also means that when a context switch occurs, only the process specific space is changed. Within the system space exist pointers to system functions that user level processes may use to access 'executive routines' [1, pp 36]. Also stored here are executive procedures which user level procedures are not allowed direct access.

2.2 User Processes

As alluded to before the user process space is divided into two sections **P0** and **P1**. The **P0** region is used to store the users executable program and dynamically grows towards the higher addresses of the **P0** space. On the other hand the **P1** region is used to store process specific data and the program image used by the command interpreter. The process stack is located in the high end of the **P1** region and grows towards the lower addresses.

3 Pager and Swapper

The pager is an operating system procedure that executes as the result of a page fault. It executes in the context of the faulting process and is responsible for loading and removing process pages into and out of memory. The swapper is a separate process responsible for loading and removing entire processes into and out of memory.

3.1 Pager

Some systems use a global page replacement algorithm where a faulting processes may replace any page in memory regardless of which process that page belonged to [1, pp 37]. VAX/VMS uses a constrained system for page replacement in which a processes may only remove pages that it controls. This implementation was chosen because in the case of a heavily faulting process, the process will not cause other processes to fault more by removing pages that belong to them.

There is a maximum number of pages any one process may have in memory at one time. The set of pages currently in memory for a process is referred to as that process' *resident set*. When a process has filled its resident set and a page fault occurs, its resident set is used to determine a victim for replacement. The replacement algorithm used is first in first out because it is simple and operates in $O(1)$ time. At first this seems like a poor implementation because no matter how heavily used a page is, it can be replaced when it is the page that has been in a process' resident set the longest. In other words it does not take into account how recently a page has been used when a resident set eviction occurs. This is, however, a conscious decision on the designers part to reduce required cpu time when an eviction occurs. The reasoning behind this decision was that it is easier to add memory than it is to increase processor cycles (per time unit) so they favor using a more memory intensive solution compared to compute intensive.

3.1.1 Free and Modified Page Lists

When a page is no longer in a process' resident set it is a candidate for eviction from memory and is placed on one of two lists. If the page's modified bit is not set, it is placed on the tail of the *free page list*. Contrarily, if the page's modified bit is set it will be placed on the tail of the *modified page list*. These lists are maintained in memory and serve as a buffer for recently evicted pages. When a page fault occurs, the free and modified page lists are first checked, if the page exists in either list, the page is removed from the list and remains in its page frame (the page is no longer a candidate for replacement when a fault occurs). In this way the free and modified page lists serve as a memory cache for pages removed from a process' resident set.

The free page list also serves as a list of available page frames. When a page fault occurs, the new page is read into the page frame currently at the head of the free list. How long a page is cached on the list depends on how long the list is and how frequently page faults are occurring for currently executing processes. For this reason it is important that each list maintains a reasonable number of pages otherwise any benefit from checking for a faulting page in them will be negated. The main difference between the free and modified page lists is that the pages in the modified page list need to be written back to the disk before being removed from memory whereas the pages on the free list can be discarded any time because the disk copy is still up to date. This memory intensive addition to the first in first out page eviction algorithm yields performance relatively close to the more favorable, but more compute intensive, least recently used page replacement algorithm.

3.2 Swapper

The swapper has two primary goals. First to keep high priority processes resident in memory, and second to avoid high paging rates generated by resuming a process [1, pp 40]. Swapping is

handled by the swapper process, it runs when called by the operating system. When a process is being swapped out, its resident set is written to a swap file along with some other process specific data used by the operating system [1, pp 40]. Also stored in the swap file is the process' page table. When a process is being swapped in memory is allocated for the process' resident set and page table along with the operating system specific data. Once free pages have been allocated the swapper updates the page table for the process so that all virtual memory mappings are accurate. A process will not be swapped in unless there is sufficient memory available for its resident set [1, pp 40]. This way it is ensured that when the process resumes its resident set is in the exact same state it was in when the process was swapped out.

4 Disk Access Minimization

Hard disk access times are significantly higher than memory access latency. Therefore it makes sense to attempt to reduce the number of times the disk is read from or written to. Levy and Lipman handle this problem by using a technique known as read or write *clustering* [1, pp 38]. Essentially clustering in this context means performing several disk reads or writes during a single disk access. For reading from the disk it means that when a process is initially loaded from the disk, several pages are loaded into its resident set based on spatial locality of reference. Significant gains are realized here over zero demand paging schemes where pages are loaded into memory when they are first referenced, causing page faults with each new page. For writing clusters the modified page list is used for the source of all writes. By using a delayed write scheme, several optimizations are realized by Levy and Lipman. Because writes are delayed, many writes are avoided entirely because the page faults or the program terminates. Because the modified page list is used as a form of cache it is unwise to write all pages in the modified page list to the disk when a write is performed. It is important that the list maintains a certain

lower limit, between 20 and 30 pages, in order to perform well as a buffer for pages removed from their resident set. A write is performed when the modified page list becomes significantly large and reaches an upper limit. Pages are written from the head of the list until only the lower limit of pages are left. After the pages are written to the disk they are placed in the free list so that they may be eventually overwritten when another page fault is encountered.

References

[1] Henry M. Levy and Peter H. Lipman, *Virtual Memory Management in the VAX/VMS Operating System*. Computer, 1982