Lecture 12: Virtual Memory

Adam Hawley

February 26, 2019

Contents

1 Demand Paging

1.1 Introduction

Program is permanently stored in a backing store and is swapped in as needed. The backing store is also split into storage units (called **blocks**), which are the same size as the frame and pages. The prgram ends up with a small set of pages loaded in main memory — a *working set*. Programs are executed (more or less) sequentially and coverage of the program is generally small (many functions are seldom used e.g. error handling routines, mutually exclusive modules, maintenance etc.).

1.2 Advantages of Demand Paging

Demand paging allows to run programs which require much more memory than physically available (limited by secondary storage or addressing space).

1.3 Requirements of Demand Paging

- Fast secondary storage device: DMA
- More decisions to be taken by the OS:
    - What to do when all memory is full and a new page is needed? - Which page is replaced?

2 Virtual Memory

2.1 Introduction

Separation of user logical memory from physical memory. Logical address space can be much larger than physical address space since only part of the program eeds to be in memory for execution. More performance and resource efficiency since less I/O needed to load or swap processes and allows for more efficient process creation.

2.2 Virtual Address Space

Usually design logical address space for stack to start at Max logical address and grow *down* while heap grows *up*. This maximises space use by leaving unused address space between the two, this means that no physical memory is needed until the heap or stack grows to a given new page. It also enables sparse address spaces with holes left for growth, dynamically linked libraries etc. System libraries shared via mapping into virtual address space. Shared memory by mapping pages read-write into virtual address space. Pages can be shared during `fork()`, speeding process creation.

3 Demand Paging Mechanism

Similar to paging system with swapping. Pager is a swapper that deals with pages that only loads the needed pages. A page is said to be needed when there is a reference to it (read/write to its address range).

- Invalid reference $\rightarrow$ Abort

- Not-in-memory $\rightarrow$ Bring to memory

When a process is to be swapped in, the pager predicts which pages will be used before the process is swapped out again. So instead of swapping in a whole process, the pager brings into memory only its estimate of the working set. The OS must distinguish between the pages that are in memory and the pages that are on the disk (to do this, usually a valid-invalid scheme is used). If pages needed are already **memory resident** then there is no difference from regular paging. If the page needed is not **memory resident** then there is a **page fault** and the OS needs to detect it and load the page into memory from storage without changing the program behaviour and without the programmer needing to change code.

3.1 Handling Page Faults

A page faut is an interrupt so a context switch ensues. This means that the process state is saved and the OS is enabled to restart the instruction that caused the page fault, as the CPU will be in exactly the same state as prior to the memory reference. The OS will do the following upon page fault:

1. Find a free frame
2. Swap page into frame via scheduled I/O operation
3. Reset tables to indicate page now in memory (set validation bit equal to valid)
4. Restart the instruction that caused the page fault

In the extreme case, a process may be started with none of its pages in memory. The solution is *pure demand paging.*

1. OS sets instruction-pointer to the first instruction of the process, non-memory-resident.
2. Page fault signalled and recovered
3. Same for every other process pages on first access.

A single instruction can access multiple pages and cause multiple page faults. For example, fetch and decode of an instruction which adds 2 numbers from memory and stores the result back to memory. The two numbers may reside in two different pages.

3.2 Demand Paging Mechanism Performance

There are three major activities:

1. Service the interrupt — few hundred instructions
2. Read in the page — lots of time
3. Restart the process — few hundred instructions

Page Fault Rate The liklihood of a memory access to be a page fault:

- If $p = 0 \rightarrow$ no page faults
- If $p = 1 \rightarrow$ every reference is a fault

Effective Access Time (EAT) `EAT = (1 - p) * memory access time + p(page fault overhead + swap page out + swap page in )`

3.3 Demand Paging Mechanism Optimisation

- Swap space I/O faster than file system I/O even if on the same device. (The swap is allocated in larger chunks so there is less management needed than file system)
- Copy entire process image to swap space at process load time and then page in and out of swap space. (Need to remember the big initial copy cost)
- Demand page from program binary on disk, but discard rather than paging out when freeing frame.
  - Used in Solaris and current BSD
  - Still need to write to swap space
- Copy-On-Write (COW) allows both parent and child processes to initially share the same pages in memory (if either process modifies a shared page, only then will a page be copied). This allows for more efficient process creation.
- Variation on `fork()` system call has parent suspended and child using address space of parent (`vfork()` on Linux and designed to have the child make an `exec()` call). This is very efficient.