apo042@uit.no

adi705
Aditya Ponnappan

Introduction:

This project implement a basic virtual memory system with the help of demand paging scheme and a usb stick used as swap are for the processes to be stored. Every user process has its own address space along with disabling access to kernel space for all the processes with the exception of threads as they share the address space with the kernel and uses it for memory accesses.

Technical background:

Virtual Memory: It is a property of an operating system through a system overcomes the memory shortage by having most of the parts of a running code in a disc storage while having a small part in the memory.

Page Fault: Every user process is split into chunks called 'pages' which for most part are present in the memory. But every once in a while, a requested page may be unavailable and thus has to be fetched from the secondary memory. This is called a page fault.

FIFO page replacement algorithm : As the name suggests It makes an assumption that the oldest page isn't recently used and thus ought be the one for replacement.

Random replacement algorithm: This is one of the simplest algorithms available. This follows the principle of deleting a page randomly and subsequently replacing it. Needless to say it is not the most optimal replacement algorithm.

Design:

Normally, the programs running on a computer systems tend to be larger than the specified RAM of the device. Thus only parts of a running program are actually present in RAM (Main Memory) at any point of time while the rest being stored in the disk. The locality of references ensures that this is a reasonable thing to do.

Implementation of a Virtual memory system overcomes the limitation of availability of RAM by effectively giving the programs an expanded address space with the OS having to do most of the heavy lifting. It implements a paging mechanism is used to achieve this with the help of data structures called page tables.

2 level paging design: Each process is divided into a fixed size 'pages' that are of 4096 bytes, that are defined by virtual addresses. Theoretically a memory management unit is responsible for mapping theses virtual address to a physical address in memory. A page table is a data structure that stores the mapping of these virtual memories. A page table could be thought of as an array whose index represents a page number and the content at the index is an address where the process is actually loaded in main memory.

The size of a page table is determined by the number of bits used for addressing index which in our project is 10, thus there exist 1024 page table entries in every page table.

Similarly a 2 level paging implies that the page table is further divided into pages whose addresses are stored in an outer page table known as page directory. Just like a page table, a page directory has 1024 entries each of which point to the address of a page table.

As mentioned earlier a process has its code & data segments an assigned virtual memory and each process in this project has identical starting virtual address.

A special data structure called page_status is maintained to keep track of the total available pages that can be allocated based on immediate requirement.

all_page_status: A user defined data structure with attributes that tells whether a page frame is occupied or available and if it is pinned or not. The purpose of having a page frame pinned reflects the fact that it is immutable, implying that an eviction algorithm cannot substitute the pages that are "pinned".

A page fault handler is responsible for trapping the page faults. The page fault in the context of our project can occur because:

1. A process tries to access a page for which it does not have the required permission, such as page in kernel space.

2. When the page is not present in the physical memory. Consequently the appropriate sector number is retrieved from the USB disk.

If a page is not in the page table then we first need to check if there is an available page frame in the designate physical memory.

A list keeps track of page occupancy for detecting an empty page. If a page frame is empty, a designated physical address is allocated for the page to be stored . The missing page itself is read in from the secondary memory and written to the newly allocated physical address.

In event of shortage of free pages, a page must be replaces. There are couple of efficient page replacement algorithms that can accomplish this by reducing the odds of reassessing the replaced page, thus leading to fewer page faults & lower latency. However for the sake of simplicity our algorithm replaces the first available page frame that is not pinned.

Implementation:

Physical memory allocation policy: A page_alloc function allocate size bytes of physical memory. This function is called by all the other associate functions to allocate memory for a page directory or a page table and by page fault handler to allocate memory for various segments of a process such as code & data segments or a stack segment . For the purposes of memory alignment to a page boundary , every byte of an allocated page is filled with a zero. This ensures every designated physical address is a multiple of a page size.

Memory initialisation: This function is responsible for setting up the kernel page directory and subsequently initialising a kernel page table. In order for the kernel to have memory access for all possible physical addresses, the mapping is identical. As both the kernel page directory and page table have a permanent presence in the RAM, the page frames that was allocated with the help of our user-defined function must be pinned. The user access bits in case of page table entries of a kernel page table is set to 0 as the user process does not have direct access to kernel space.

Setting a page table: A method sets up a page directory and page table for a new process or thread along with allocating a page table for its user stack. While a new thread uses the kernel page directory to access memory, a specialised page directory is created to store the addresses of process page tables and its a stack table. The first index of every process page directory points to the kernel page table with the difference of setting user access bit to 1 for every page table entry.

Page Fault Handler: As the name suggest, this function performs a sequence of actions on encountering a page fault which is primarily caused by the absence of a page frame in memory. The first step of figuring out which virtual address caused the page fault is stored as an attribute in the pcb data structure.

This method utilises this information to obtain the correct sector number in which the virtual address is stored in USB, thereby storing the page frame (8 sectors) into an array whose base address is allocated with the help of a page_alloc function.

Assuming a virtual address is valid, the handler tries to set up a free page frame. If no frames are free, the page replacement algorithm is run to remove a page. The contents of the substituted page is written to disk regardless of the fact that a page selected is dirty or not.

The fault address is used to retrieve the page directory index to check if a page table exists by extracting out the last bit. Depending on its presence (1 or 0), a page table is allocated with the help of a page_alloc function.

Discussion:

While everything may appear to work perfectly fine at first glimpse , things begin to take an unexpected turn when trying to load the last process. But the simulation is again error free if the last 3 or 4 processes are loaded. Thus it seems to be the case that as long as the number of process loaded into memory is not over a threshold, the OS simulates paging mechanism as per expectations. The error could perhaps be attributed to the lack of pagable pages as all the pages can end up being pinned leaving no memory for subsequent processes.

Resources:

https://www.geeksforgeeks.org/two-level-paging-and-multi-level-paging-in-os/

https://www.geeksforgeeks.org/page-fault-handling-in-operating-system/