Memory Management

Agenda

- Basics: Address space
- Virtual memory
- Address translation
- Segmentation
- Paging
- Page replacement algorithms
- Design of page table
- Page faults and thrashing

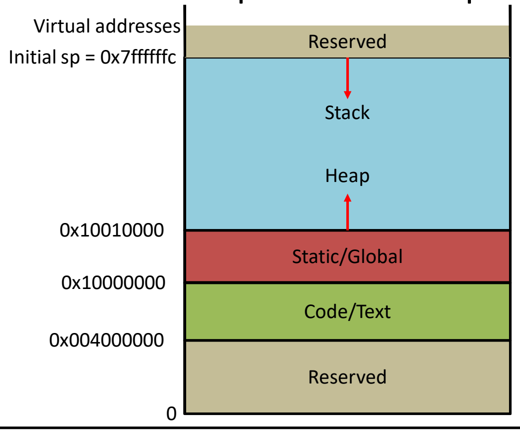
Address space

- Early computers ran one process at a time
  - The process can occupy whole physical memory except a small portion reserved for OS code and data
- With multiprogramming, there are two options
  - Give whole memory to the currently running process
    - Slow because we need to swap in the code and data of a scheduled process and swap out the code and data of a decheduled process
  - Leave the memory-resident code and data of the processes in memory as long as possible
    - Fast because need to save/restore only registers on context switch
    - How to assign addresses to processes?
    - What if memory is full? What about protection/isolation?

Address space

- Each process is assigned a distinct address space
  - Each address space starts at address zero and goes up to 2^v-1 where v is the width of virtual address
  - All addresses in an address space are virtual addresses
  - A program can generate only virtual addresses
  - In some OS, each address space is assigned a distinct ID, referred to as ASID
    - Each ASID has a one-to-one correspondence with PID
    - A process can have only one ASID
  - Address space of a process contains code (set of instructions), global static data, dynamic data (heap), and stack

Address space: An example



Address space

- Each process is assigned a distinct address space
  - The job of the OS is to map demanded portions of the address space of a process to free blobs of physical memory and maintain this mapping
    - Virtual to physical address mapping
    - All memory accesses go through a virtual to physical address translation
- Address space abstraction is needed for isolation
  - The job of the OS is to keep the code/data of a process isolated from other processes
    - Address space abstraction helps achieve this through address mapping
  - Central component of memory virtualization

Basics of memory management

- Basic requirement of memory management is two-fold: correctness and security
  - A process should read from or write to a piece of data if and only if the piece of data belongs to that process
- Memory management is done at two levels
  - By OS and by each process
    - OS manages the physical memory
    - Each process manages its own virtual memory with the help of OS

Process-level memory management

- Stack memory is managed automatically by compiled code
  - Compiler generates fills from and spills to stack
- Two operations for managing process heap: allocation and de-allocation
  - Allocations done through standard library functions such as malloc, calloc, realloc, etc. in C
  - Deallocation done through library function free
  - Different languages have different functions
- Allocation function invokes the brk or sbrk system call if needed

Process-level memory management

- Allocation functions invoke sbrk or brk system call only if needed
  - If the heap allocated so far needs to be extended, the system call is invoked to extend the "memory break" of the process
- Deallocation functions just return the freed memory to the heap of the process
- All allocated memory of a process is reclaimed by OS when the process exits
  - Therefore, not freeing memory for a short-lived process does not lead to any problem
- Not freeing memory that is no longer used by long-running processes leads to memory leak

Process-level memory management

- A process can also use mmap() to dynamically allocate memory
  - A wrapper around the mmap system call
  - mmap allocates memory in two possible ways
    - Maps a file (or a portion thereof) to memory and uses the file space to back up this memory space
    - Allocates memory backed by swap space (which doesn't belong to any file); this is referred to as anonymous mapping
  - mmap allows sharing the allocated memory across processes without requiring shmget/shmat calls

Physical address binding

- Every piece of data has an address and the address can be determined at three different points: compilation, loading, and execution
- If the compiler knows exactly where a data will be placed in memory, it can generate absolute addresses
  - Reduces flexibility in memory allocation
  - How to bind addresses for dynamically allocated data?
  - How to handle a process, size of which exceeds the size of the available memory?
  - How to handle multiple processes?

Physical address binding

- Usually, the compiler generates relative addresses
  - All the addresses are relative to some address such as stack pointer, global pointer, etc.
  - Loader carries out the absolute address binding
  - Lots of flexibility in memory allocation, but needs an additional level of translation at run-time
  - How to handle a process, size of which exceeds the available memory size?
- Ideally, we want to load parts of the process as and when needed and do physical address binding at that time

Virtual memory

- With a 32-bit address, one can access 4 GB of memory
  - A process will never get the complete memory though
  - Seems enough for most day-to-day applications
  - However, it seems unfair to load the entire application executable at startup
    - A process will never require the complete executable at any point in time
    - Takes away memory from other processes and hurts the degree of multiprogramming
  - Virtual memory offers an address space and a translation mechanism in computer systems that help decouple the logical and physical views of memory

Virtual memory

- Virtual or logical memory consists of the address space that every process sees
  - This is the process's view of the memory and every process sees exactly the same address space
  - The size of this address space is determined by the instruction set architecture of the processor
    - Datapath width
  - In a 32-bit architecture, every process gets 4 GB virtual address space
    - Some of it is usually reserved for kernel use and the rest is given to the process
    - Text, global constants, heap, and stack reside in the virtual address space and get mapped dynamically to physical memory by the OS through address translation

Address translation

- Act of translating virtual addresses to physical addresses at run time
  - Each memory access requires this translation
  - Done in hardware for speed
  - Can be done using just two new hardware registers base address and bound, provided the address space of a process is mapped contiguously to physical memory
    - Just add the base register contents to virtual address to get the physical address; check that the generated physical address is within the bound
    - On a context switch, save and restore these two registers
    - Need kernel mode instructions to write to these registers

Address translation

- Act of translating virtual addresses to physical addresses at run time
  - Can be done using just two new hardware registers base address and bound, provided the address space of a process is mapped contiguously to physical memory
  - Unfortunately, mapping the address space of a process contiguously to physical memory poses significant challenges
    - What if there is no big enough free hole in physical memory?
    - What if the process needs to grow at run time?
    - Seems wasteful to reserve full address space at start of a process (the space between stack and heap is wasted)

Segmentation

- Wasted memory can be reduced by having a (base, bound) pair for each segment
  - One pair each for code, global, heap, and stack segments
  - OS can place a segment in any free hole of physical memory that is large enough to accommodate the segment
    - Needs to maintain a list of free physical memory holes
    - What if a segment grows too much to fill up the allocated hole or what if there isn't a big enough hole to begin with?
    - Still, there is wasted holes of small sizes: external fragmentation
  - Any access outside the bound of a segment leads to a segmentation fault; also needs rwx permission check
  - Segments can be identified by upper few bits of virtual address (e.g., 00, 01, 10, 11 for CS, GS, HS, SS)

Paging

- Segmentation requires the OS to be able to accommodate variable-sized segments some which can also grow
- This drawback is addressed by paged virtual memory
  - The process address space is divided into equally sized small units called pages
  - To get rid of external fragmentation, the physical memory is also divided into equal-sized units called page frames
  - A virtual page is loaded into a page frame selected at run-time only when the virtual page is needed
    - This is called demand paging (only internal fragmentation)
  - The CPU generates virtual addresses while executing
  - Even though the virtual pages of a process are contiguous, physical pages need not be
    - A virtual to physical page number translation is needed

Paging

- Example

  int *a = (int*)malloc(8000*sizeof(int));

  for (int i=0; i<8000; i++) a[i] = 0;
  - Virtual memory for array "a" is allocated by malloc
    - Virtual memory allocated = 32000 bytes (sizeof(int) is 4)
    - No physical memory is allocated yet
  - Assume the page size to be 4096 bytes
    - One page can hold 1024 contiguous elements of "a"
    - Assume that "a" occupies virtual pages 10 to 17
  - When a[0] is accessed in the first iteration of the loop, the first virtual page of "a" is accessed from physical memory, but the physical page is not there
    - OS allocates one free physical page frame (say PF#3) and remembers the mapping VPN 10 → PFN 3 in the page table of the process (each process has a page table)

Paging

- Example (continued)

  int *a = (int*)malloc(8000*sizeof(int));

  for (int i=0; i<8000; i++) a[i] = 0;
  - Second page of "a" gets accessed when loop reaches i=1024
    - OS allocates another free page frame (say, PF#7) and adds a new mapping VPN 11 → PFN 7 to the page table of the process
  - Next page frame gets allocated when i=2048 and so on
  - The allocated 8 page frames remain in memory until the OS is forced to swap out some of the pages due to shortage of physical memory

Paging

- Virtual to physical address translation
  - Every virtual address is divided into two parts: virtual page number and page offset
  - The page offset remains unchanged in the translation
  - The virtual page number is translated to a physical page frame number
  - The translation is maintained in a per-process page table
  - The first step in this translation is to access the page table (for now, let's assume a contiguous page table)
    - Every process has a page table base register (PTBR) loaded by the loader; it is a part of the process context and stores the starting physical address of the page table
    - Necessary offsets are added to the PTBR

Paging

- Page table entry and page faults
  - A page table entry (PTE) contains several pieces of information
    - A valid bit, present bit, a dirty bit, permission bits (rwx), a mode bit, a reference or accessed bit, and the much needed translation
    - The present bit indicates if the page is currently present in physical memory and if not, the result is a page fault
    - The valid bit indicates whether the page has a valid PTE
  - A page fault is handled by raising a restartable exception (which generates a trap) and the page fault handler of the OS handles the exception in software
    - After handling the exception, the process undergoing the page fault returns from trap and restarts from the same instruction that suffered from a page fault (contrast syscall)

Paging

- Page fault handling
  - Save context (same as done in any trap e.g., syscall)
  - Locate the needed page in the next level of storage
  - Find or create a free physical page frame to accommodate the newly brought in page
  - Start a copy operation via DMA and invoke the process scheduler to pick a new process to run
  - On DMA completion, make appropriate changes in the page table
  - Start a DMA operation to write the replaced page to the next level of storage, if the dirty bit is set

Paging

- Page fault handling
  - Since the next level of storage is disk, the swap-in and swap-out operations are slow
  - A part of the disk is maintained as a swap partition
    - This partition is maintained using lower-overhead techniques so that reading from and writing to this partition is fast
  - A page replaced from memory is kept in the swap partition until it has to be evicted (due to finite swap space) and moved to its original location in the file system (anonymous pages stay in swap partition)
    - The swap partition is a fast victim cache for the memory
    - The page replacement algorithm plays an important role; its goal is to minimize the number of page faults

Paging

- Who does page replacement and when?
  - Page replacement is done ahead of time by the swap daemon or a page daemon
    - A process that runs only in kernel mode
  - The page daemon usually sleeps and it is woken up only when the number of free pages reaches a certain threshold
    - This threshold is usually referred to as the low watermark
  - The page daemon selects pages to replace and continues running until the number of free pages reaches another threshold referred to as the high watermark
    - Number of free pages is kept between HWM and LWM

Paging

- Virtual to physical address translation
  - Once the physical page frame number is available, it is appended in front of the page offset to get the physical address, which can now be used to access the data item
  - Two memory accesses to get a data item: one to get the translation and another to get the data
  - Translation-related memory accesses can be reduced in number by caching a subset of the translations used recently inside the processor
    - This cache is called translation look-aside buffer (TLB), one for instruction and one for data
    - A TLB entry contains a tag (derived from the virtual page number), a PTE, an address space id, a valid bit, repl. bits
    - A TLB miss leads to a page table access

Paging

- TLB example
  - for (i=0; i<8000; i++) a[i] = 0;
  - Assume that each array element is four bytes
  - Assume that page size is 4096 bytes
  - How many TLB misses if TLB has 16 entries?
- How is a TLB miss handled?
  - In hardware or by software-based TLB miss exception handler (the latter case goes through a trap)
  - After the miss is handled, the instruction is re-executed
- What if the TLB is full and there is a new miss?
  - Need to replace an entry to accommodate a new one
  - Like cache replacement policies: LRU, random, etc.

Paging

- Demand paging poses a performance problem with fork()
  - Before starting the child, all pages of the parent need to copied into child's physical page frames leading to a large number of page faults
  - Usually, a copy-on-write policy is followed
    - Unless the child or the parent writes to a page after fork(), it is not copied
  - This read-only pages are shared between parent and child
  - On a fork() call, the parent's page table is copied into the child's; in both page tables the data pages are set to read-only permission and code pages are set to execute-only permission

Page replacement algorithms

- A page replacement algorithm is an online algorithm
  - Takes a sequence of accesses in the form (VA, pid) and returns (fault, PPFN) or (no fault, PPFN)
    - Invoked on all accesses, but replaces a page only on a page fault provided all physical page frames are occupied; otherwise there is no need for page replacement
    - In the case of a replacement, selects one of the physical page frames and assigns it to the current access (VA, pid) that has suffered from a page fault
    - The goal of a page replacement algorithm is to minimize the number of page faults
    - The biggest challenge in these algorithms is that the full input sequence is not known at any point in time as the future accesses are unknown

Page replacement algorithms

- We will transform the input sequence before applying the algorithm
  - The actual virtual addresses will be transformed to the corresponding virtual page numbers and the sequence of virtual page numbers forms the input to the algorithm
    - Consider a page size of 100 bytes and the virtual address sequence (100, 432, 101, 612, 102, 103, 104, 101, 611, 102, 103, 104, 101, 610, 102, 103, 104, 101, 609, 102, 105), which will be transformed to (1, 4, 1, 6, 1, 1, 1, 1, 6, 1, 1, 1, 1, 6, 1, 1, 1, 1, 6, 1, 1) and we will deal with this transformed sequence only

Page replacement algorithms

- The most popular deterministic replacement algorithms include FIFO, LRU, LFU, and a large number of approximations of LRU
  - The FIFO policy replaces the oldest page provided all physical page frames are occupied
    - Consider a memory with three physical page frames and the access sequence (7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1); this sequence experiences fifteen page faults with FIFO replacement
    - It may not be always true that having more page frames reduces the number of page faults with FIFO replacement
    - Consider the sequence (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5) and compare the number of faults with three and four page frames when using FIFO replacement

Page replacement algorithms

- The LRU replacement algorithm replaces the least recently used page
  - Needs to maintain the order of accesses to the pages
  - Every page access needs to update this order
  - The sequence on which FIFO has fifteen page faults, LRU has twelve page faults
  - The motivation for LRU policy comes from the optimal algorithm
    - An algorithm that replaces the page with the furthest access in the future is provably optimal (due to Laszlo Belady, 1966); also known as the longest forward distance (LFD) replacement algorithm
    - This algorithm cannot be implemented due to dependence on future
    - LRU is a crude approximation of the optimal algorithm

Implementing LRU

- Two ways of implementing LRU: counter-based and list-based (usually known as stack-based)
  - A counter-based implementation attaches a time-of-access field with each page frame
    - On each access, this field is updated with the current hardware clock tick; the page with the smallest tick is replaced
    - Three drawbacks: the size of this field must be equal to the size of the hardware clock register, handling wrap-around is difficult, replacement requires a find-min operation
  - Possible to maintain relative time instead of absolute time
    - On an access to a page frame, its time field is reset to zero and all others' time fields are incremented by one; the page with the largest count is replaced

Implementing LRU

- List-based implementation
  - Maintains a doubly-linked list of page frame ids
  - The head of the list is the MRU frame and the tail is the LRU frame
  - On each access, the accessed frame is delinked and made the head of the list
    - Requires O(1) pointer operations
  - Does not suffer from the drawbacks of the counter-based implementations
  - Still requires O(nlog n) space to store the recency list for n physical page frames
  - In reality, some approximation of LRU is implemented that needs only O(n) space
    - Let us discuss such approximate schemes

Reference bit algorithm

- For each page frame, there is a reference bit and a k-bit register
  - k is usually a small constant
  - The reference bit is set on each access to the frame
  - Periodically, all the registers are shifted to right by one bit and the reference bits are copied to the most significant position of the register; at this time all the reference bits are cleared
  - The page frame with the smallest register value is replaced; requires a find-min operation, which is O(n)
  - The register value of a frame is the history of accesses to the page frame during the last k periods

Second chance algorithm

- We have just the reference bit per page frame
- The idea is to replace the page in the FIFO order that has its reference bit reset
  - If at the time of replacement, the oldest page has its reference bit set, it is skipped but its reference bit is reset i.e., it is given a second chance to get accessed
  - LRU-CLOCK is one of the simplest implementations
    - The page frames are organized in a circular FIFO queue with a pointer pointing to the next replacement candidate (this is like a clock hand)
    - If the replacement candidate has its reference bit set, the reference bit is reset, the pointer is moved to the next entry, and the process continues until a replacement candidate is found; its reference bit is set and the pointer is moved to the next frame

Design of page table

- Organization of page table
  - The page table access algorithm implicitly assumes that the page table is stored in a contiguous portion of physical memory
    - Let's see if it is practical by computing how large the page table is
    - Notice that all translations resident in the page table are not needed simultaneously with high probability
    - Assume that the page size is 2^p bytes, the virtual address space is 2^v bytes, and the PTE size is 2^t bytes; this leads to a page table size of 2^{v-p+t} bytes which is 8 MB for v=32, p=12, and t=3; this is just for one process
  - To conserve memory, page tables are also paged just like the processes
    - Called hierarchical paging and affects how a PTE is located

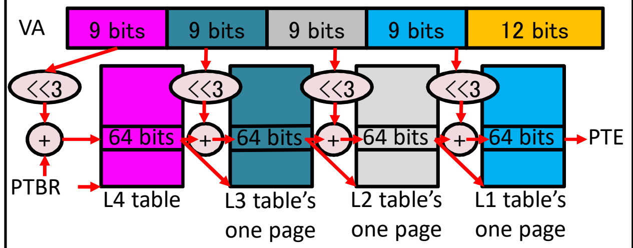
Design of page table

- Reserving a large contiguous memory space for a linear page table is wasteful
  - The program doesn't need the entire page table at any point in time; many page table entries may not be needed while running
- Larger page size reduces page table size, but increases internal fragmentation
- Can use segmentation and paging together to reduce the page table size
  - Maintain one linear page table per segment
    - Base and bound registers of a segment maintain the starting address and size of that segment's page table

Design of page table

- Paged segments avoid allocating page table entries for unused parts of a segment
  - External fragmentation in allocation of page table
  - Growing segments may need page tables to grow
- Popular solution: page the page table
  - Avoids allocating pages of the page table (PT) that have only invalid PTEs; reduces PT size significantly
  - Need a "page directory table" to locate pages of PT
  - The page directory table is large, so that is also paged
  - General principle: any level of page table that is larger than a page is paged to avoid reserving contiguous memory larger than a page

VA to PA translation: An example



TLB misses are very expensive: four memory accesses

Two levels of TLB with very large L2 TLB

Small page structure caches (PSCs) or page walk caches (PWCs) to cache recently used L4, L3, L2 table's entries

VA to PA translation: An example

- Step#1: Look up L1 TLB; on a hit, return PTE
- Step#2: Look up L2 TLB; on a hit, insert PTE in L1 TLB and return PTE
- Step#3: Invoke hardware page walker to handle TLB miss (some processors use software TLB miss handlers, but with multi-level page table, that is slow)
  - Step#3A: Look up the PSCs in parallel and retrieve entries that are not found in the PSCs one at a time from physical memory; continue until PTE is obtained; insert each level entry in PSCs
  - Step#3B: insert PTE in both L1 and L2 TLBs

Page faults and thrashing

- A process uses a set of pages quite frequently before moving on to another set of pages
  - The set of pages accessed over a time window (representing a locale of the program) is called the working set of the process
    - The working set keeps changing with time
  - A process is said to be thrashing if it is suffering from an excessive volume of page faults
    - Happens if the process fails to get enough page frames to accommodate its working set

Page faults and thrashing

- Pre-paging
  - When a new process is swapped in, thrashing may occur until the process has its working set in the memory (lot of page faults in the beginning)
    - Pre-paging or prefetching is used to solve this problem
  - The last seen working set of a process is stored along with its context and the OS swaps in these pages before scheduling the process
    - This is called pre-paging or prefetching
    - Downside: The working set may not be accurate and some of the swapped in pages may not be used
  - Thrashing can also be controlled by increasing or decreasing the degree of multi-programming depending on the page fault count
    - Referred to as admission control algorithm