Memory Management

Agenda

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

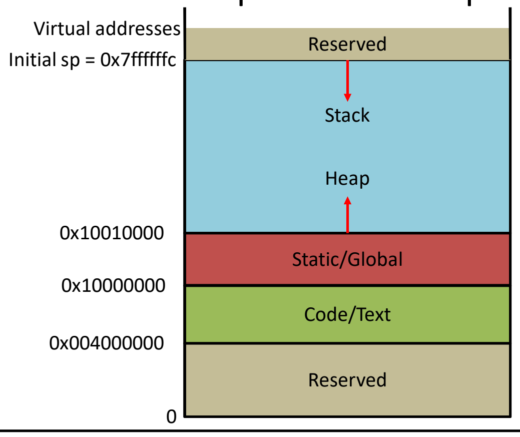
Address space

- 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 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 present bit serves the purpose of a valid bit as well
  - A page fault is handled by raising a restartable exception and the page fault handler of the OS handles the exception
    - After handling the exception, the process undergoing the page fault will restart from the same instruction that suffered from a page fault

Paging

- Page fault handling
  - Save context
  - 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
    - 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

- 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, and a process id (aka address space id)
    - A TLB miss leads to a page table access

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
    - We will discuss four 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

Enhanced second chance algorithm

- Suppose we categorize each page into the following four classes in increasing order of priority
  - (reference bit reset, dirty bit reset), (reference bit reset, dirty bit set), (reference bit set, dirty bit reset), (reference bit set, dirty bit set)
- The pages are replaced from the two lowest priority classes
  - The oldest page in the lowest non-empty priority class is replaced (only the lowest two classes are considered); why is the dirty bit important?
  - If both the lowest priority classes are empty, the LRU-CLOCK algorithm is run on the other two priority classes to populate the two lowest priority classes

Pseudo-LRU

- An O(log n) time algorithm requiring O(n) space for n page frames
  - Arrange the page frames logically at the leaves of a binary tree
  - Each internal node of the tree has a bit indicating which way to go from that node
  - On each access (including new allocations), a traversal is made from the accessed frame to the root changing the bits of the internal nodes encountered on the way to point in the other direction
  - A replacement candidate is found by starting a traversal from the root until a leaf is reached
    - The leaf is the replacement candidate

Page frame caches

- When replacing a dirty page, the replaced page must be written to the disk before the new page can be filled in
  - To reduce the critical path, a few frames are always kept in reserve so that a newly brought page can be filled into one of these without delay
  - When the dirty page is completely flushed out, the frame holding this page can be returned to the reserve pool (also known as the reserve frame cache)
  - The implication is that page replacement should be invoked as soon as the memory is full beyond a threshold
  - Recently replaced pages are usually kept in a page frame cache before dropping them

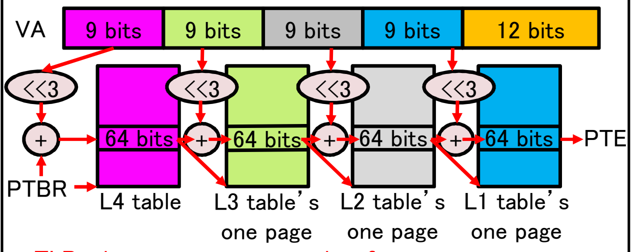
Design of page table

- 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 the page table is wasteful
  - The program doesn't need the entire page table at any point in time
  - The entire page table may not be needed for running the program
- The page table is usually paged
  - Need a "page directory table" to locate pages of the page table
  - Since the page directory table is large, 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 (older processors used software TLB miss handlers, but with multi–level page table, that is slow)
  - Step#3A: Look up PSCs in parallel and identify the lowest level PSC that returns a hit among L4, L3, L2; suppose n is the lowest level that returns hit (n=5 if no hit in any PSC)
  - Step#3B: start retrieving entries at levels n−1, n−2, … one at a time by first looking up the traditional CPU cache hierarchy and on a miss main memory; continue until PTE is obtained; insert each level entry in PSCs and CPU caches
  - Step#3C: insert PTE in both L1 and L2 TLBs

Design of page table

- Hierarchical page table
  - Suppose we have a virtual page with number N in the range [0, 2^{v-p})
  - The address space of the page table is [0, 2^{v-p+t}) if we assume that the page table starts at address zero
    - We will page this space
    - The PTE for the virtual page N is located at address 2^tN; we will view this as the logical address of the PTE
    - Once the page table is paged, this logical address will get located in some physical page frame
    - Let us apply the same mechanism for translating virtual addresses to physical addresses and see how the logical PTE address gets translated to the paged physical memory

Design of page table

- Hierarchical page table
  - The first step is to remove the page offset from the logical PTE address to get the PTE's virtual page number and this is $2^{t-p}N$
  - This virtual page number must be translated to a physical page frame number which will tell us where the page of the page table containing this PTE is located in physical memory
  - To do this, we use a page table with number of entries equal to 2^{v-2p+t}
    - Let us call it the L1 page table, which must be contiguous in physical memory
  - Let us assume that each entry of the L1 page table is of size 2^t bytes so that the total size in bytes is $2^{v-2p+2t}$

Design of page table

- Hierarchical page table
  - For the PTE we are looking for, the L1 page table will be looked up at offset $2^{2t-p}N$ from the start of the L1 page table
  - Let us suppose that this L1 page table entry provides the physical page frame f
    - This is the physical page frame of the main page table (we can call it the L2 page table) where the PTE is located
  - We append the last p bits of 2^tN after f to get the actual physical address where the PTE can be found
  - In summary, for each process, we need to keep the entire L1 page table in memory occupying a contiguous space of $2^{v-2p+2t}$ bytes and on demand bring in the pages of the L2 page table

Design of page table

- Hierarchical page table
  - In a fully paged physical memory, it is a little inconvenient to allocate a contiguous space the size of which exceeds the size of a page
    - May need moving data to create a big enough hole
  - Ideally, we want our L1 page table to be at most a page in size
  - The only way to achieve this is to introduce more levels in the hierarchy of page tables
    - Page the L1 page table
  - Notice that each new level reduces the L1 page table size by a factor of 2^{p-t}
    - The penalty comes in terms of the time taken in a translation if we miss the TLB

Design of page table

- Hierarchical page table
  - In an n-level page table hierarchy, the total size of the L1 page table is $2^{v-np+nt}$ bytes
  - We need to find the smallest +ve integer n such that $2^{v-np+nt} \leq 2^p$ or $v-(n+1)p+nt \leq 0$ i.e., $n \geq (v-p)/(p-t)$
    - For v=32, p=12, and t=3, we have $20-9n \leq 0$ i.e., n is at least 3 meaning that in this case we need a three-level page table hierarchy with L1, L2, and L3 tables; the L1 table is of size 32 bytes (four entries), which fits in a page; the L2 and L3 tables are paged, the pages are brought in as needed, and can also be swapped out; the total size of the L2 page table is four physical page frames i.e., 16 KB (2048 entries); the size of the L3 page table is 2048 page frames i.e., 8 MB (2^{20} entries, as required)
    - Notice that the total size of all the page tables has increased from 8 MB to 8 MB + 16 KB + 32 bytes

Design of page table

- Hierarchical page table
  - Starting from the virtual address how do we get the final PTE in an n-level hierarchy?
    - Derive the virtual page number by shifting the address to the right by p bit positions
    - Shift the virtual page number to the right by (n-1)(p-t) bit positions and the residual portion is the index into the L1 page table
    - Shift the L1 page table index to the left by t bits to get the byte offset which needs to be added to the starting address of the L1 page table to get the address of the L1 PTE
    - The L1 PTE provides the physical page frame number of the L2 PTE; the actual L2 PTE address is calculated by first shifting the next p-t bits of the virtual address to the left by t bit positions and adding this to the starting address of the page containing the L2 PTE and the lookups continue

Design of page table

- Lower bound on page table size
  - If the physical memory size is 2^m bytes, we only need to maintain the translation for 2^{m-p} pages leading to a page table size lower bound of 2^{m-p+t} bytes
  - However, the traditional way of maintaining translation requires page tables for each process and the aggregate storage of these can far exceed the lower bound
  - Why can't we maintain translations of only the 2^{m-p} resident pages?

Design of page table

- Lower bound on page table size
  - One possible implementation that achieves this bound is known as the inverted page table
  - The page table has exactly 2^{m-p} entries and each entry stores a pair: (pid, virtual page number)
  - When a process with pid P needs to translate a virtual page number N, it would search the inverted page table looking for the pair (P, N)
    - If found, the index of the entry is the desired physical page frame number; otherwise it results in a page fault
  - The TLB miss latency can be very large and the worst part is that it grows as more memory is installed
    - Could be greatly optimized with a space-bounded hash table i.e., the total number of hash elements is bounded

Design of page table

- Page replacement and inverted page table
  - Inverted page tables are not good for VA to PA translation, but are useful for PA to VA translation
  - The inverse translation (PA to VA) is needed at the time of replacing a page
    - The PTE of the replaced page needs to be looked up and updated, which requires the VA and the pid of the process owning the replaced page
    - The VA is also needed to flush the translation from the TLB and may be needed to flush the cache blocks mapping to the page from the processor caches
  - The inverse translation can be obtained in O(1) time from an inverted page table

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
  - In early operating systems, CPU utilization was used as an indicator for increasing or decreasing the degree of multiprogramming
    - Can get into a situation where thrashing keeps increasing

Page faults and thrashing

- Thrashing avoidance is a two-step algorithm
  - Working set estimation and pre-paging
  - Working set estimation involves remembering the list of unique pages accessed over a small window (e.g., the last five thousand references)
    - Too small a window fails to capture the full working set
    - Too large a window captures multiple working sets
    - The list of accessed pages can be generated by periodically collecting the reference bits
    - Once working sets of all active processes are collected, determining the optimal degree of multi-programming is equivalent to the bin-packing problem
    - The processes that cannot be accommodated are swapped out from memory and suspended until there is a space in memory

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
    - Pre-paging is used to solve this problem
  - The working set list 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
    - Downside: The working set list 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

Page faults and thrashing

- Even if the OS takes all measures to control thrashing, poor programming can lead to a very high volume of page faults
  - Consider the following two ways of initializing a two-dimensional array

```
for (i=0; i<n; i++) for (j=0; j<n; j++) a[j][i] = 0;


for (i=0; i<n; i++) for (j=0; j<n; j++) a[i][j] = 0;
```

  - This example shows that the OS must switch to a local frame allocation policy for the processes with high fault rates
    - Saves the other processes from thrashing