Operating Systems

Virtual Memory
Based on Ch. 9 of
OS Concepts by SGG

Need for Virtual Memory

- Most Important: sounds good!
- We can run many huge programs on systems with little memory (unusual on modern systems)
- We can avoid loading all the program if we need to use a small part
- We do not need to load the whole program to start execution and show the first response to the user.

Demand Paging

- In virtual memory systems we do not need to have every page available in the physical memory
  - The page table has a valid bit. If it is false the page is missing
- When we try to access a page that is missing because it is on the disk, we generate a page fault
  - The end result is to get the page we want

Page Faults

- What happens if a process tries to access a page that is not in the memory?
  - An interrupt is raised
  - System checks if it is really a segmentation violation (terminate?) or a simply a page fault
  - If a page fault, we find a free frame (or a victim and grab it)
  - Schedule a disk operation to bring it in, block the process, and let the scheduler take over.
  - When the page arrives, the process is put back on the ready queue with its state restored.

Two Issues with Page Faults

- How do we restart an instruction interrupted by a page fault.
  - We deal briefly with this.
- How do we select a victim so that
  - We minimize the number of page faults
  - Increase efficiency

Locality of reference

- Most real world programs at any time use a few pages only.
  - If only we knew which, we could keep the program optimally satisfied
- This is a common theme in cache memories, TLBs, etc that we exploit.

Hardware Support Needed

- Page system, with valid bit or equivalent
- Ability to restart after an interrupt as if nothing happened
- Swap space on a device like disk.

Restarting an Interrupted Instruction

- Most asynchronous interrupts can be serviced at the boundaries of instructions
  - That’s easy.
- A page fault happens in the middle of an instruction. We cannot wait till the end of the instruction
  - This keeps computer architects safely employed
- We let computer architects solve this problem.

Restarting an Interrupted Instruction

- Some instructions are easy to restart
  - Like: store reg. R3 to M[R4]
- Others are impossible:
  - Copy a long array to another array
    - Good luck if they overlap.
- Others are a bit tricky
  - The instruction refers to two addresses on two different pages and referencing the one causes the other to be kicked out
- Most such instructions have disappeared from modern ISAs

Free-Frame List

- Usually systems keep a list of unused frames
- When a page fault occurs the system wastes no time looking for a victim.
- The system should make sure the page is zeroed before given to a process (security!)
- Occasionally the system searches for rarely used pages and reclaims their frames.

Performance

- Disk access is around a few thousand times slower than main memory access
- So if *p* is the probability of a memory reference to result in a page fault, *t* is the time it takes for a main memory access and *T* is the time to swap in a page
  - $t\_eff = (1-p)*t + p*T$
- Probability *p* (aka page-fault rate) should be tiny

Copy-on-Write

- When we fork we create two identical processes.
  - If a huge process wants to create a little short-lived utility process it has to make a copy of itself! And then discard this copy almost immediately with exec().
- There is a little trick to avoid all this waste
  - Copy only the page tables and mark them copy-on-write.
- When a process writes anywhere on its address space it causes a new copy of that page to be created.
- There is a video regarding a bug with the copy-on-write: https://www.youtube.com/watch?v=CQcgz43MEZg

Page replacement

- Usually we allocate fewer frames to each process and not the whole address space.
- This allows more processes to be resident and ready to run
- A typical process needs a certain number of frames to run efficiently
- We have to optimize the whole page replacement process
  - If we split the frames among too many processes, none of them will have enough to run efficiently with few page faults
  - If we split them among too few, the probability that all of them are doing I/O or paging at the same time is large.

The Cost

- The cost of a page fault can be broken into four main components
  - Cost of servicing the interrupt
  - Cost of saving the victim
  - Cost of reading the new page
  - Cost of restarting the process
- The first and the last are usually much smaller if we do not get too ambitious.

Minimizing the cost

- The cost of swapping in a frame depends on the disk, the bus, etc.
- The cost of writing back a victim on the same things
  - But we do not always need to write it back
  - We keep a “dirty” bit on the page table to indicate if a page needs writing back
  - We prefer to swap a “clean” page out
- Occasionally we write back pages to “clean” them.

Reducing the Frequency

- We would like to reduce the page-fault rate
- If every process has the number of pages it needs it will create few page-faults
- The problem is we do not know how many these are
- And if we avoid paging out pages that will be needed soon, we further decrease page-faults.

Reference Strings

- Assume that we know how a process references pages.
  - We can emulate a program and record that info.
- We want to try a few algorithms and see how they behave

FIFO

- Replace the oldest page first
- Simple, and needs minimal hardware support
- We can emulate it with a random string like
  - 1,2,3,4,1,2,5,1,2,3,4,5
- And assume that we have either 2, 3, 4 or 5 frames available
- It exhibits Belady’s Anomaly

Optimal Page Replacement

- We hire a fortune teller or palm reader to tell us which page is going to be needed further into the future
- Works best in Harry Potter situations.
- But it is used to see how good is a new algorithm
  - If it is close to optimal, great!

LRU

- Like SJF we use the past as a guide to the future
- Least Recently Used is a good approximation to the optimal algorithm (in real life)
- It is hard to implement in an exact way
- We will talk about several approximations.

LRU Implementations

- Counters:
  - Increment the counter at every memory access
  - Every time a page is referenced we copy the counter to the page table entry for this page
  - The counter acts as a clock. Should be able to handle overflow.
- Stack:
  - Have all the pages in a linked list.
  - Every time we reference a page we move the page to the head of the list

LRU Implementations

- Both methods require constant updates to the memory
- Both methods require extra memory
- Both methods try to find an exact solution to an approximation
  - Remember: LRU is a real world approximation of the Optimal method.

Reference Bit

- We have a reference bit for every page in the page table (right next to the valid bit)
- If the reference bit is true when we reference a page we continue as usual
- If it is false, we set it to true.
- Periodically we clear all the reference bits
- If the bit is on it means that the page was referenced at least once since the last purge.

Additional Reference bits

- A single bit is kind of crude.
  - Immediately after a purge we do not know anything about which pages are in use
- Have a second bit
  - Periodically, copy the (first) reference bit onto the (second) history bit and clear the first.
  - This allows us to distinguish between recently and very recently referenced pages
- Can have more than two history bits

Second Chance

- AKA clock algorithm
- It is essentially a FIFO replacement algorithm.
- If we get a page fault we select a victim as in the usual FIFO page replacement algorithm
  - If the victim has the reference bit set, we clear it and find another victim
  - If the reference bit is clear, we have a victim.

Enhanced Second Chance

- Like the regular one (clock algorithm) but we also look at the dirty bit.
- Like before we make an initial selection FIFO style.
  - If both the reference and dirty bits are clear we have a victim
  - Otherwise we clear the reference bit and find next candidate victim.
  - If both bits are set in all pages we go back to the first candidate victim.
- This may need several passes over the page table.

Page Buffering

- Most systems need to maintain a set of free frames.
  - Allows a new page to be brought immediately
  - The free pages can be just the “clean” pages in the system
  - If we put a page on the death row and it is referenced before paged out, we can restore it immediately without any I/O
- This scheme can be used in systems without hardware support for reference bit

Applications with special needs

- Databases typically do their own paging
  - Very often they scan a huge file bringing in a page to use it only once.
- In most other applications if a page was used once, it will be used again soon.
- Databases often handle their filesystem themselves. The OS makes available a raw disk.

Frame Allocation

- Every process needs a minimum number of frames to work efficiently
  - If it has fewer than that it will have too many page faults
  - If it has more it wastes resources
- This number varies from process to process and from time to time for the same process
- The total number of frames cannot exceed the size of the main memory

Equal and Proportional Allocation

- Both very simple but almost useless
- Do not take into account the needs of the process
  - A process with large address space may use a tiny portion of it and the opposite

Global vs Local Allocation

- In global schemes any page in the system can be a victim
  - Allows balancing of resources
  - Can take into account priorities
  - Shared pages are a non issue
- In local schemes only a page that belongs to the process that raised the page fault can be a victim
  - Can guarantee the minimum number of frames
  - The page-fault rate does not depend (much) on the load of the system

Non-uniform Memory Access

- In NUMA systems every CPU has its own local RAM.
- All the CPUs talk to each other through a common bus.
- Accessing local memory is faster than non-local
- The memory management subsystem has to decide not only how many pages to allocate but where.
- Linux keeps threads from migrating to different NUMA nodes and keeps one free-frame list per node.
- Similar to Solaris.

Thrashing

- If the number of processes in a system is so high that there are not enough pages to satisfy the minimum requirement, the page faults increase
  - The fault rate can increase for other reasons too (change of locality). It is hard to detect it.
- Often the situation gets worse if the OS misdiagnoses the situation as not having enough processes.

Working Set Model

- We try to estimate the size of the working set of frames that will allow the process to run efficiently.
- We define parameter Δ which is the width of the working set window.
  - Typical values for Δ are 5,000 or 10,000, 20,000
  - The working set is the set of distinct pages we referenced in the past Δ memory references
- We only care for the size of the working set

Working Set Model

- We do not need much in terms of hardware support to approximate WSM
- At fixed intervals Δ we count the number of pages that have been referenced and clear the bits.
- This is a bit crude, so we use one or more history bits (copy and clear the reference bit onto the history bit) and copy and clear more frequently.
- So if a page fault occurs and we need to check if the number of frames we have is enough we count the number of pages that were referenced

Memory Mapped Files

- Every frame in the system has space on the disk (the swap space)
- We can copy a file (or part of it) into the memory.
  - This memory is not mapped on the swap space, but on the original file
- This can have many uses

Uses of Memory Mapped Files

- Easy programming: access files like arrays
  - No need to seek, read, seek again, etc
- Save disk space (if we read the file in the usual way, we will allocate swap space)
- Two processes can share this memory space if they can share the file.
- The file could be a device
  - Web cameras can work this way to minimize the number of times an image is copied or minimize the number of system calls needed to read an image.

Kernel Memory

- Kernel memory is often allocated from a different pool of free frames.
- There a few reasons for this:
  - Kernel data structures have varying sizes and often much longer life span.
  - They may not be subjected to the page system.
  - May need to be in contiguous physical space to interact with devices.

Kernel Memory Allocation

- Buddy system:
  - Only allocate memory in power of two chunks.
  - If we need less than half of the smallest chunk, we split it in two buddies.
  - If we free a chunk, and its buddy is also free, we coalesce them.
  - Has a rather big waste to internal fragmentation.

Kernel Memory Allocation

- Slab Allocation
  - A slab is a set of one or more physically contiguous pages
  - A cache is one or more slabs.
  - There is a single cache for each kind (size) kernel data structure
    - One cache for PCBs, one for open file records, one for system wide semaphores, etc
- Easy to allocate each size structure and free it without internal fragmentation.

Prepaging

- System tries to guess which pages will be needed next.
- Easy if we swap out a whole process (then bring back the pages that we swapped out)
- Not easy in general.
- Need to check the cost of prepaging as compared to the savings/speed up we achieve.

Page Size

- Small pages allow smaller internal fragmentation and better approximation of locality
- Large pages decrease I/O per byte brought in
  - Unless of course we bring in things that we do not need
- TLB reach is increased with larger page size
  - The bigger the TLB reach the higher the TLB hit ratio
- Large pages need less space for page tables

TLB Reach

- The other way to increase TLB reach is to increase the number of entries
  - This is obvious but costly.
- Or provide multiple page sizes
- Or take advantage of the contiguous pages:
  - It is rather common that a large sequence of contiguous pages maps to contiguous frames
  - These pages can be made to map to a single TLB entry (ARMv8)

I/O interlocking and Page Locking

- During I/O a page designated as an I/O buffer cannot be swapped out.
- There are two ways to enforce this:
  - Always do I/O to and from kernel memory
  - Allow pages to be locked in memory using a lock bit
    - A lock bit is needed to lock some kernel pages in.
    - Or lock database managed pages.
    - Or keep new pages from being swapped out before they are used even once