CIS520 – Operating Systems
Handout 10
Issues in Paging and Virtual Memory

- Page Table Structure. Where to store page tables, issues in page table design.
- In a real machine, page tables stored in physical memory. Several issues arise:
  - How much memory does the page table take up?
  - How to manage the page table memory. Contiguous allocation? Blocked allocation? What about paging the page table?
- On TLB misses, OS must access page tables. Issue: how the page table design affects the TLB miss penalty.
- Real operating systems provide the abstraction of sparse address spaces. Issue: how well does a particular page table design support sparse address spaces?
- Linear Page Table.
  - OS now faces variable-sized allocation problem for the page table storage.
  - Page table may occupy significant amount of physical memory. Page table for 32-bit address space with 4K byte pages has $2^{32}/2^{12} = 2^{20}$ entries. If each entry is 32 bits, need 4M bytes of memory to store page table.
  - Does not support sparse address spaces well - too many wasted entries.
  - TLB miss handler is very simple - just index the page table.
- Two-Level Page Table. Page Table itself is broken up into pages. An outer page table indexes pages of page table. Assuming 4K byte pages and 32 byte page table entries, each page can hold $2^{12}/4 = 2^{10}$ entries. There are 10 bits left in virtual address for index of outer page table. Virtual address now has 10 bit outer page table index, 10 bit inner page table offset and 12 bit page offset.
- Page table lookup for two level page table.
  - Find physical page containing outer page table for process.
  - Extract top 10 bits of address.
  - Index outer page table using extracted bits to get physical page number of page table page.
  - Extract next 10 bits of address.
  - Index page table page using extracted bits to get physical page number of accessed page.
  - Extract final 12 bits of address - the page offset.
  - Index accessed physical page using 12 bit page offset.
- Evaluation of two level scheme.
  - Eliminates variable-sized allocation problem for page tables. Have one page for outer page table, and rest of page table is allocated in page-size chunks.
  - Have internal fragmentation - both for last page table page and for outer page table page.
  - If page table takes up too much memory, can page the pages of the page table. Question: is there anything that OS MUST keep in physical memory?

  - Supports sparse address spaces - can eliminate page table pages if corresponding parts of address space are not valid.
  - Increases TLB miss time. Have to perform two table lookups instead of one.

- Three level scheme. Like two level scheme, only with one more level. May become scheme of choice for machines with 64 bit address space. On such machines the outer page table can become much larger than one page. SPARC uses three level page tables.

- Primary job of page table: doing TLB reload. So why map entire address space? Instead, just maintain mappings for pages resident in physical memory.

- Inverted Page Table. Has one entry for each physical page frame specifying process that owns the page and the virtual address of page frame.

- On a TLB miss, search inverted page table data structure to find physical page frame for virtual address of process generating the access. Speed up the lookup by:

  - Hashing
  - Associative table of recently accessed entries.

  IBM machines (RS/6000, RT, System 38) and HP Spectrum machines use this scheme.

- What if accessed page is not in memory? Must look up the disk location in a data structure that looks much like a standard page table. Since this data structure should not be accessed very often, it can be paged.

- All of these schemes have advantages and disadvantages. Which one should the hardware implement? Answer: hardware designer does not have to decide! Most modern machines handle TLB misses in software, so the OS can use whatever page table scheme it wants. Hardware only "knows" about the TLB.

- Sharing code and data. If two page table entries in different processes point to same physical page, the processes share the memory. If one process writes the data, other process will see the changes. Is a very efficient way to communicate.

- Can also share code. For example, only one copy of editor or compiler code can be kept in memory, and all editor or compiler processes can execute that one copy of the code. Helps memory utilization.

- Concept of reentrant code. Reentrant code cannot modify itself and must make sure that it has a separate copy of per-process global variables. All of your Nachos kernel code should be reentrant.

- Complications with sharing code: virtually indexed caches and inverted page tables.

- Virtual Memory. Basic idea: main memory used as a cache for backing store. Usual solution: demand paging. A page can be resident either on disk or in main memory.

- First extension for demand paging: the valid bit. Each page table or TLB entry has a valid bit. If the valid bit is set, the page is in physical memory. In a simple system, page is on disk if valid bit is not set.

- The operating system manages the transfer of pages to and from the backing store. It manages the valid bit and the page table setup.

- What does OS do on a page fault?

  - Trap to OS.
  - Save user registers and process state.
  - Determine that exeception was page fault.
  - Check that reference was legal and find page on disk.
  - Find a free page frame.
  - Issue read from disk to free page frame.

    - Queue up for disk.
    - Program disk controller to read page.
    - Wait for seek and latency.
    - Transfer page into memory.
  - As soon as program controller, allocate CPU to another process. Must schedule the CPU, restore process state.
  - Take disk transfer completed interrupt.
  - Save user registers and process state.
  - Determine that interrupt was a disk interrupt.
  - Find process and update page tables.
  - Reschedule CPU.
  - Restore process state and resume execution.

- How well do systems perform under demand paging? Compute the effective access time as in OSC Sec. 9.3. p is proportion of memory accesses that generate a page fault ($0 \leq p \leq 1$). What is effective acess time? If data in memory, between 10 and 200 nanoseconds, depending on if it is cached or not. Call it 100 nanoseconds for purposes of argument. Retrieving page from disk may take 25 milliseconds - latency of 8 milliseconds, seek of 15 milliseconds and transfer of 1 millisecond. Add on OS time for page fault handling, and get 25 milliseconds.

- Effective access time $= (1 - p) * 100 + p * 25 * 10^6$. If we want overall effective access time to be 110, less than one memory reference out of $2.5 * 10^6$ can fault.

- In the future, difference between local accesses and faulting accesses will only get worse. In practice people simply do not run computations with a lot of paging - they just buy more memory or run smaller computations.

- Where to swap. Swap space - a part of disk dedicated to paging. Can usually make swap space accesses go faster than normal file accesses because avoid the overheads associated with a normal file system.

- On the other hand, using the file system immediately makes the paging system work with any device that the file system is implemented on. So, can page remotely on a diskless workstation using file system.

- May not always use backing store for all of process's data.
  - Executable code. Can just use executable file on disk as backing store. (Problem: recompilation).
  - Unreferenced pages in uninitialized data segment. Just zero the page on first access - no need to access backing store. Called zero on demand paging.

- To get a free page, may need to write a page out to backing store. When write a page out, need to clear the valid bit for the corresponding page table entry. A core map helps this process along. A core map records, for each physical page frame, which process and virtual page occupy that page frame. Core maps will be useful for other operations.

- When invalidate a page, must also clear the TLB to avoid having a stale entry cached.

- Page replacement algorithms. Which page to swap out? Two considerations:
  - A page that will not be accessed for a long time.
  - A clean page that does not have to be written back to the backing store.

- Hardware provides two bits to help the OS develop a reasonable page replacement policy.
  - Use bit. Set every time page accessed.
  - Dirty bit. Set every time page written.

Hardware with software-managed TLBs only set the bits in the TLB. So, TLB fault handlers must keep TLB entries coherent with page table entries when eject TLB entries. There is another way to synthesize these bits in software that makes TLB reload faster.

- How to evaluate a given policy? Consider how well it works with strings of page references.
- FIFO page replacement. Do Figure 9.8 in OSC. What is disadvantage of FIFO? May eject a heavily used page.
- Belady's anomaly - adding more physical memory may actually make paging algorithm behave worse! See OSC Fig. 9.9.
- LRU - eject least recently used page. Do OSC Fig. 9.11. Problem: how to implement LRU.
- Two strategies:
  - Build a clock and mark each page with the time every time it is accessed.
  - Move page to front of list every time it is accessed.

  Both strategies are totally impractical on a modern computer - overhead is too large.
- So, implement an approximation to LRU. One version is equivalent to LRU with a clock that ticks very slowly. So, many pages may be marked with the same time. Have a low-resolution LRU.
- Implement using use bits. Periodically, OS goes through all pages in physical memory and shifts the use bit into a history register for the page. It then clears the use bit. When read in a new page, clear page's history register.
- FIFO with second chance. Basically, use a FIFO page ordering. Keep a FIFO queue of pages to be paged out. But, if page at front of FIFO list has its use bit on when it is due to be paged out, clear the use bit and put page at end of FIFO queue.
- Can enhance FIFO with second chance to take into account four levels of page replacement desirability:
  - use = 0, dirty = 0: Best page to replace.
  - use = 0, dirty = 1: Next best - has not been recently used.
  - use = 1, dirty = 0: Next best - don't have to write out.
  - use = 1, dirty = 1: Worst.

  Go through the FIFO list several times. Each time, look for next highest level. Stop when find first suitable page.
- Most paging algorithms try to free up pages in advance. So, don't have to write ejected page out when the fault actually happens.
- Keep a list of modified pages, and write pages out to disk whenever paging device is idle. Then clear the dirty bits. Increases probability that page will be clean when it is ejected, so don't have to write page out.
- Keep a pool of free frames, but remember which virtual pages they contain. If get a reference for one of these virtual pages, retrieve from the free frame pool. VAX/VMS uses this with a FIFO replacement policy - use bit didn't work on early versions of VAX!
- What if hardware does not implement use or dirty bits. Can the OS? Yes, if hardware has a valid and readonly bit.
- Hardware traps if valid bit is not set in a page table or TLB entry. Hardware also traps if readonly bit is set and reference is a write.
- To implement a use bit, OS clears valid bit every time it clears use bit. OS keeps track of true state of page in different data structure. When the first reference to the page traps, OS figures out that page is really resident, and sets use and valid bits.

- To implement dirty bit, OS sets readonly bit on clean pages. OS keeps track of true state of page in different data structure. When the first write traps, OS sets dirty bit and clears readonly bit.
- Many systems use this scheme even if TLB has dirty and use bits - with this scheme, don't have to rewrite page table entries when eject an entry from TLB.
- Concept of a working set. Each process has a set of pages that it frequently accesses. For example, if the process is doing a grid relaxation algorithm, the working set would include the pages storing the grd and the pages storing the grid relaxation code.
- Working set may change over time. If the process finishes the relaxing one grid and starts relaxing another, the pages for the old grid drop out of the working set and the pages for the new grid come into the working set.
- Discussion so far focussed on running program. Two complications: loading a program and extending address space.
- Invariant: must reserve space in backing store for all pages of a running process. If don't, may get in a situation when need to eject a page, but backing store is full.
- When load a process, must reserve space in backing store. Note: do not have to actually write pages to backing store - can just load into memory. Makes startup time faster.
- What needs to be initialized? Only init data segment, in principle.
  - Can use zero on demand pages for uninit data segment.
  - Can use executable file to hold code.

  Makes sense to preload much of code segment to make startup go faster. Of course, must be prepared to page even during startup - what if init data segment does not fit in available physical memory?
- Must also allocate more backing store when extend address space via something like malloc.
- What is involved to allocate backing store pages? Just manipulate data structure in kernel memory that maintains state for backing store page allocation.
- Thrashing. A system thrashes if physical memory is too small to hold the working sets of all the processes running. The upshot of thrashing is that pages always spend their time waiting for the backing store to fetch their pages.
- Thrashing is a discrete phenonmenon. Usually, there is a phase transition from no thrashing to thrashing.
- Typical way thrashing came about in early batch systems. Scheduler brought in new process whenever CPU utilization goes down. Eventually the size of the working sets become larger than physical memory, and processes start to page. The CPU utilization drops even further, so more processes come in, and the system starts to thrash. Throughput drops like crazy.
- Eliminating thrashing. Must drop degree of multiprogramming. So, swap all of a process out to backing store and suspend process.
- Come up with a Page Fault Frequency based solution. Basic idea: a process should have an ideal page fault frequency. If it faults too often, need to give it more physical page frames. If it faults too infrequently, it has too many physical page frames and need to take some from it and give to other process. When all processes fault too frequently, choose one to swap out.
- Another swapping algorithm - keep a free list threshold. When processes demand free pages at a high rate, free list will be consumed faster than pages can be swapped out to fill it. When amount of free memory goes above threshold, system starts swapping out processes. They start coming back in when free memory drops back below threshold.
- Page size. How big are current page sizes? 4K bytes or 8K bytes. Why aren't they bigger?

  - Tradition and existing code.
  - Increased fragmentation.

  Why aren't they smaller?

  - Smaller page size has more page faults for same working set size.
  - More TLB entries required.
  - Larger page tables required.

  Page sizes have increased as memory has become cheaper and page faults more relatively expensive. Will probably increase in the future.

- Pinning pages. Some pages cannot or should not be paged out.
  - There is some data that OS must always have resident to operate correctly.
  - Some memory accessed very frequently - disk buffers.
  - Sometimes DMA into memory - DMA device usually works with physical addresses, so must lock corresponding page into memory until DMA finishes.