20-EECE-4029 - Introduction to Operating Systems

Review Notes for Exam 2

March, 2013

Memory Management

Motivation

- Efficient use of memory requires support from many complex components

Memory Hierarchy

- Fast memory is expensive, cheap memory is slow.
- There are many types of memory that fall between the above extremes (discussed later):
  - ▷ Registers - there are relatively few of these, each might be at most 8 bytes long.
  - ▷ Hardware cache - cache lines of about 64 bytes each, total of about 64KB, save lookups to slower main memory
  - ▷ Translation Lookaside Buffers - page table caches of about 4MB each for quick translations from virtual to physical addresses
  - ▷ There might be two levels of the caches mentioned above
  - ▷ RAM operates more than 10 times slower than the caches in the processor but GBs of RAM may be available at reasonable cost.
  - ▷ DISK access time may be as long as .1msec
  - ▷ The cloud has gobs of space but uncertain reliability and access time, not to mention potential security problems.

Processes in Memory

- Each process occupies some space, possibly interspersed with free space
- There are usually many processes but few processors, so some scheduler must switch a processor to run on a collection of processes
- Filling registers on each context switch might take a lot of time
- Three memory segments associated with a process: heap, stack, text.

```
int main () {          // &main is where the text segment is
    int *s = (int*)malloc(sizeof(int)); // s is where the heap is
    int x = 3;         // &x is where the stack is
}
```

- The OS makes a process think it owns the machine but actually it shares the memory space with other processes via virtualization

Lazy Resolution of Function Addresses

Motivation

- addresses can be resolved at compile time, load time, or during execution. Maximum flexibility is obtained the later the bindings happen

Global Offset Table

- Converts position independent addresses to absolute virtual addresses
- A table of addresses - in the data section of a program
- An executing instruction gets virtual addresses from the GOT
- Supports dynamic linking to libraries with PLT

Procedure Linkage Table

- Converts position independent function call addresses to absolute addresses
- PLT entries contain:
  - a jump to a location in the GOT
  - setup code that changes the GOT entry so as not to recall the setup code
  - a call to the resolver
- PLT operates as follows to resolve `f@PLT[n]`:
  - `PLT[n]` is called and jumps to the address pointed to in `GOT[n]`
  - This address points into `PLT[n]` itself, to the setup code which prepares arguments for the resolver
  - The resolver is called
  - The resolver resolves the actual address of `f`, places the actual address into `GOT[n]` and calls `f`
  - `GOT[n]` then points to the actual `f` instead of back into the PLT
  - When `f` is called again `PLT[n]` is called and jumps to the address pointed to in `GOT[n]`
  - Since `GOT[n]` points to `f` this just transfers control to `f`, not the resolver

Virtualization:

- **Goals**:
  - **Transparency**: OS needs to set virtual addresses so that process thinks it owns the machine
  - **Efficiency**: OS needs to do this with minimum overhead (time & space)
  - **Protection**: OS needs to prevent unauthorized access of memory
- **Address Translation**: hardware changes virtual addresses to physical addresses
  - during instruction fetch, load, store
  - translation occurs on every memory reference
  - registers are provided to save a base address and a bounds limit
  - the physical address is the addition of the base to the virtual
  - efficient since this is done directly in hardware
- **Managed by the OS**:
  - keeps track of which addresses are free and which are owned by whom
  - multiple processes may share the same physical memory at the same time
  - OS only intervenes when a process is created, terminated, and switched
- **Segments**: portion of contiguous address space
  - OS satisfies requests for memory by dishing out segments
  - base and bounds registers - offset added to base for address, bounds register used to check whether access is legal
  - as memory is returned to the OS and other requests are satisfies, holes show up in free space and finding segments of the right size becomes hard
  - compaction will then be needed but this is difficult on varying sized segments
- **Allocation Algorithms**: for segments
  - **Best fit**: return memory from smallest chunk of free space that satisfies request (attempts to reduce wasted space)
  - **Worst fit**: return memory from largest chunk of free space that satisfies request (hopefully other requests can be satisfied from what's left)
  - **First fit**: return memory from first chunk satisfying request (low overhead - small free chunks develop near the front)
  - **Next fit**: same as first fit but take memory from the back (along with first fit can remove some of the splintering of first fit)

Paging:

- **Motivation**:
  - Since de-fragmentation is easy with fixed sized segments, use memory units that are all the same size
  - These units might be 4K in size (12 bits - offset)
- **Properties**:
  - Both physical and virtual address space are paged
  - A page of physical memory is called a page frame
  - The OS supports the abstraction of an address space effectively, regardless of how processes use the address space
  - No assumptions about how the heap and stack grow and how they are used need be made
  - An address space may have many pages and they do not have to be in address-increasing order during instruction fetch, load, store
- **Page Table**:
  - per-process page table in memory that maps address space pages to physical space frames
  - page table address is in a page table base address register (PTBR)
  - bits 12-31 index into the page table
  - page table entry has the following (32 bit addresses):
    - bits 12-31: page frame
    - bits 0-11: dirty bit, reference bit, valid bit, r/w bit, user/superuser bit plus others, plus a few are available for the OS
- **Address Translation Computation**:
  - Extract `VPN` from virtual address
  - Page table entry address = `PTBR`+`VPN`*sizeof(`PTE`)
  - Fetch `PTE`
  - Check whether process can access page (valid? legal?)
  - Mask virtual address for offset
  - Append offset to page frame number

Translation Look-aside Buffer:

- **Motivation**:
  - Address translation can be slow if done in software due to all the calculations for a single address
  - So cache the most important translations in hardware
- **Outline**:
  - TLB is a cache of popular virtual-to-physical address translations
  - On a virtual memory access, the hardware looks at the TLB first
  - depends on locality: spatial locality - if memory location x is accessed then it is likely that a location near x will be accessed soon; temporal locality - if memory location is accessed, it is likely that it will be accessed again soon (e.g. loop instructions)
  - Implemented as a hash table for fast access
  - May be only 4MB in size
  - Only applies to the running process
- **Performance**:
  - Lookups are only a couple of nanoseconds which is much less than RAM
  - Hit ratios are commonly 80-90%
- **Problems**:
  - Page tables can be large and there is one per process
  - 100 processes using 32 bits of address space costs 400MB
  - Large page size increases internal fragmentation
  - Multi-level page tables allows some of a process' page table to be placed in memory by paging the page table.
  - bits 22-31: index the directory table, bits 12-21: index the page table, bits 0-11: the offset into the table

Buddy Allocation:

- **Motivation**:
  - Low overhead mechanism for allocating pages
  - External fragmentation can be great but
  - Compaction is simple
- **Outline**:
  - Two halves of address space are buddies
  - Two halves of each half of address space are buddies
  - Two halves of each quarter of address space are buddies
  - On a request, find a segment of size that is a power of 2 that satisfies the request by splitting from a large segment - which segment, if there are many, depends on many factors, for example the need to retain some large segments
  - When memory is returned, if the buddy of the memory is free combine them into a larger segment
- **Performance**:
  - Worst case allocation/deallocation is slow: coalescence is too eager although the number of such ops is $\log_2 n$ where n is the number of primitive blocks
  - External fragmentation is low: small allocation holes are not persistent
  - High internal fragmentation: lots of memory is allocated that is not needed - a 66K request may be satisfied with 128K memory!
  - Compaction in the classical sense is not necessary
  - Small data structures are used to keep track of which buddies exist
  - Can be used as a course-grain allocator on top of a Slab allocator

Slab Allocation:

- **Motivation**:
  - Time to initialize, allocate, and destroy objects that get used over and over again is wastful - the idea is to allocate such objects one time
- **Components**:
  - **Slab**: a contiguous piece of memory.
  - **Cache**: a small amount of very fast memory consisting of one or more slabs - a slab is the amount a cache can grow or shrink by.
- **Operation**:
  - Initially, all slabs are empty and unmarked
  - OS tries to find and return an open slot in an unmarked slab
  - If the last open slot of a slab is taken, that slab becomes marked
  - If no open slots exist, the OS allocates a new slab from contiguous physical memory and assigns it to a cache and returns a slot from it
  - Slabs are obtained from a buddy allocator
- **Hardware Cache**:
  - To avoid CPU wait for instruction fetches/execs - not the TLB
  - A *line*: a few dozen bytes - transferred as a burst from DRAM
  - The cache: subdivided into many lines
  - *fully associative*: main memory line can go to any line in cache
  - *N-way*: a main memory line can go to one of N cache lines
- **Cache Operation**:
  - CPU looks at bits of physical address representing a reference to cache line - if the line matches the address there is a hit
  - if a `read` operation: data transferred from cache line to a register access to memory is avoided
  - if a `write` operation and is write-through: both RAM and cache line are written; otherwise (write-back) only the cache line is updated
  - on a cache miss: a cache line is selected to be written to memory (if dirty) the line is replaced by the needed one

Virtual Memory

Motivation

- We want a process to think it has lots of space but we can't afford to give that much to it.
- So, we introduce the notion of virtual address space and a function that maps virtual addresses to physical addresses - physical space is allocated as needed.
- To support the sharing of libraries we introduce the Procedure Linkage Table and the Global Offset Table.
- We also segment the virtual address spaces to prevent alocating space that is not going to be used.
- But segmentation can be expensive because a translation function is required and because quite a bit of de-fragmentation may be necessary to satisfy requests for space.
- This last problem can be reduced by using fixed size segments called pages that are referenced through page tables.

Demand paging

- To get many more users in mainstore at once
- To get better utilization of hardware
- To handle jobs bigger than mainstore
- But it costs: performance & complexity
- Relies on: temporal locality - instructions in loops repeat often
- and spatial locality - instructions execute in sequence

Components

- Caches, particularly in hardware, such as the Translation look-ahead buffer (TLB) for faster page table access
- Slabs, for kernel objects of known size
- Multi-level page tables: cache the cache of page table entries
- Page allocator: Buddy system
- Swapper: lazy

Page fault

- If a page frame is not in memory a page fault is generated as follows:
  - ▷ an exception is raised - the OS handles it
  - ▷ the state of the process causing it is saved
  - ▷ it is determined that the exception was a page fault
  - ▷ it is determined that the address reference is valid
  - ▷ the location of the page on disk is found
  - ▷ the page is read from disk - this requires a wait to complete a seek plus some latency
  - ▷ the process that raised the exception is restarted because the CPU dropped it to act on another
  - ▷ the state of current I/O process is saved
  - ▷ the page table is updated
  - ▷ the state of process causing the fault is restored
  - ▷ the process is resumed
- Service time can be as high as 30 msec
- Memory access time is about 60 nsec - 6 orders of magnitude difference!!!
- For 10% impact on access speed we can can afford only 1 swap for every 2 million accesses
- If there is no free frame for the request, some extra tasks are added:
  - ▷ a “victim” page frame is found and written to disk (if it is modified)
  - ▷ the free-frame table is updated
  - ▷ the new page is read into the frame
  - ▷ the page table is updated
  - ▷ the process is restarted and resumed

Page replacement

- If a page must be swapped out to make room for a new frame, a victim frame must be chosen and swapped out. The following are strategies for choosing a victim:

  ▷ **FIFO**: remove the frame that has been resident the longest subject to an anomaly: increased frames may cause more faults!

  ▷ **OPT**: remove the frame that will be next accessed the furthest into the future. This cannot be achieved in any practical sense although it is OK for some applications. It is primarily used as a benchmark to test other strategies against

  ▷ **LRU**: remove a frame that has not been used for the longest period. LRU may be implemented using a stack: when a frame is accessed it is pulled from the stack and placed on top, the victim frame is the one on the bottom. But this is hard to implement efficiently so approximations are used

  ▷ **LRU***: approximate LRU with a single reference bit in the page table entry. The victim is the first page with a 0 reference bit. Perhaps 0 out reference bits every so often so a frequently accessed frame will always have its reference bit set and it will be less likely to be swapped out

  ▷ **LRU+**: the *second chance* algorithm. All pages are in a circular queue. If the cursor points to a page with reference bit set then reset it (giving it a second chance) and move the cursor to the next page to test again. If all reference bits are set the one the cursor pointed to originally is the victim. This is easily implemented in software

  ▷ **LRU#**: expand on LRU* by adding a dirty bit. Define four classes of page:

    1. dirty/ref = 0,0: pages of this class can be swapped out easily because it is not likely that a reference will be made soon and it is not necessary to write the page to disk
    2. dirty/ref = 0,1: is similar except a write is necessary
    3. dirty/ref = 1,0: probably should not be swapped out because a reference is likely
    4. dirty/ref = 1,1: certainly is last to be considered for swap

Frame allocation

- We should also be aware that every process should be granted some minimum number of page frames or else it will be competing with itself for free pages

- **Fixed Allocation**: all processes get the same minimum number of free pages Some processes get starved, some are too wealthy

- **Proportional Allocation**: processes get a number of pages that is proportional to either their total size or a mix of size and priority

- **Global**: a victim frame may be selected from any frame, allowing one process to "steal" from another. Performance is on a single process is non-deterministic (unpredictable)

- **Local**: a victim frame may be selected only from the set belonging to the evicting process. Results in more predictable performance but throughput suffers

Thrashing:

- system condition where processes are spending more time on page faults than on useful work
- cause: processes have not been given enough frames there are too many processes running
- cure: based on locality
  - ▷ define *working-set-window* (WSW) as some number of instructions and define *working-set* as all pages accessed within the last WSW. If the sum of all WS sizes is greater than number of pages available then thrashing results. In that case, suspend some jobs as needed to keep the sum less. A working set changes slowly with time. A crude way to keep up with changes is to look at a few bits that are updated at regular intervals and remove pages from the working set whose bits have fallen off the table
  - ▷ decide on minimum and maximum allowable page fault rates per process. If a rate falls below the minimum, remove allocated pages from the process. If the rate rises above the maximum, add free pages to the working set
- control: page size
  - ▷ if page size is increased there will be fewer page faults which reduces time overhead spent waiting for I/O to disk. But internal fragmentation will be increased. However, smaller page size improves locality (matches locality of the process) which reduces I/O to disk. A larger page size means smaller page tables.

Performance enhancers:

- **I/O interlock (locked pages)**: for code required for I/O fault handling, buffers, partially updated pages, critical kernel code, performance critical data. If, for example, an I/O process is given an address from which to take or put data, it must make sure the data at that address does not change for the long period of time that the I/O process takes - note the process should itself be interruptible.
- **Inverted page table**: a single page table replaces one page table per process. Implemented as a hash table, indexed on process and page number. These can be a lot smaller than a collection of many page tables.
- **Demand segmentation**: old processors do not have hardware support for demand paging and use demand segmentation instead
- **TLB reach**: the total memory that the TLB can cover is (TLB size)*(page size). If the reach is smaller than the working set size page faults increase
  - ▷ solution 1: increase TLB size. But that could go out of control
  - ▷ solution 2: increase page size. But that has problems noted above
  - ▷ solution 3: allow many different page sizes. But TLBs currently are designed to accomodate one size so this has to be worked out with chip manufacturers. Perhaps this is why linux only has sizes 4K and 4M - the latter maintained in software

Performance improvements:

- **Copy-on-write**: if multiple processes request resources which are *initially indistinguishable* they can all be given pointers to the same resource. Then if a process tries to modify its "copy" of the resource, a separate (private) copy is made for that process to prevent its changes from becoming visible to all others. If no process ever makes any modifications, no private copy need ever be created.
  - ▷ **Main use**: when a process creates a copy of itself, the pages in memory that might be modified by either the process or its copy are marked copy-on-write. When one process modifies the memory, the kernel intercepts the operation and copies the memory so that changes in one process's memory are not visible to the other.
  - ▷ **Other use**: calloc returns 0'ed memory. Initial calls return pointers to the same page. Then, as writes are made, copies are made and pointers changed. This is done for large callocs
  - ▷ **Implementation**: the MMU is notified that certain pages in the process's address space are read-only. When data is written to these pages, the MMU raises an exception and the handler allocates new space in physical memory and makes the page being written correspond to that new location in physical memory.
  - ▷ **Advantage**: the ability to use memory sparsely. Usage of physical memory only increases as data is stored in it. Efficient hash tables can be implemented which only use little more physical memory than is necessary to store the objects they contain.
  - ▷ **Problem**: such programs run the risk of running out of virtual address space. Virtual pages unused by the hash table cannot be used by other parts of the program.
  - ▷ **Problem**: complexity. When the kernel writes to pages, it must copy any such pages marked copy-on-write.
- **Memory Mapped Files**: a resource is read into memory using demand paging. Reads and write are treated as ordinary memory reads and writes. Speeds up access to the resource. Allows several processes to share access to the resource.
  - ▷ **Problem**: a 5KB resource maps to two 4KB pages, so there is some waste
  - ▷ **Problem**: when a block of data is loaded in page cache, but is not yet mapped into the process's virtual memory space, page faults may occur
  - ▷ **Problem**: a file larger than the addressable space can have only portions mapped at a time, complicating reading it. An IOMMU can remedy this situation.
  - ▷ **Advantage**: a system call takes orders of magnitude longer than a memory access (e.g. seek time and latency are eliminated)
  - ▷ **Advantage**: the mapping can be to the kernel's page cache and therefore not take away from user pages