OPERATIVSYSTEM 2011, ÖVNING 3, 24.11.2011, deadline 2.12.2011

1. I ett minnesallokeringssystem håller man reda på lediga minnesområden (hål) med hjälp av bitkartor alternativ länkade listor. Om allokeringsstorleken för bitkartsmetoden är *n* bytes, hur många ”hål” skall det finnas för att bitkartsrepresentationen är effektivare (dvs. använder mindre mängd minne)? Antag att den länkade listan behöver element med en 28-bitars adress, en 28-bitars längd samt en 16-bitars nästa-nod element. (2 p)

2. På sidan *http://www.cs.nmsu.edu/~pfeiffer/classes/473/notes/intelvm.html* (även som bilaga) visas hur man i en i386 arkitektur går från virtuellt minne till fysiskt minne. Vad är den fysiska adressen för den virtuella adressen 0x4580eb9c (2p) samt vilka egenskaper har den sida datan finns i, om man använder sig av samma minnesdata som i bilagan? (2p)

3. I ett system med sidindelat minne (demand-paging) används en sidstorlek på 1024 ord. Det externa minnet, dit sidor swappas ut, är ett skivminne som roterar 7200 varv per minut och har en överföringshastighet på 10 000 000 ord/sek. Genomsnittlig armryckningstid (*average seek time*) är 10 ms. Primärminnets accesstid är 50 ns och maskinen har en TLB (translation look-aside buffert) med 10 ns accesstid.
   Statistiska mätningar på systemet har gett följande resultat:

   • 0.2% av alla instruktioner refererade en annan sida än den aktiva (senast refererade) sidan.
   • 90% av dessa referenser var till sidor som redan fanns i primärminnet.
   • När sidor måste läsas in från sekundärminnet var den sida som ersattes modifierad 50% av gångerna.

   Beräkna den effektiva accesstiden för systemet om man antar att endast en process exekveras, och systemet alltså är outnyttjat under överföringar till/från sekundärminnet. (2p)

4. Skriv ett program som simulerar minnesallokering. Det skall använda sig av en bitkarta för representation av ledigt / upptaget minne. Allokeringsstorleken är 4 kB, och totalt antar man att det finns 80 MB minne tillgängligt. Följande funktioner skall implementeras (4p)

   ```
   int myalloc(unsigned int blocksize);
   int myfree(unsigned int pos);
   ```

   `myalloc` returnerar position där den nya minnesrymden börjar (om det lyckas, -1 om det inte går att allokera.. Blocksize anges i kB, likaså minnesrymden.
   `OBS. Som språk kan C alternativt Java användas.`

   a) Verifiera programmet genom sekvensen

   ```
   1: myalloc(36000);
   2: myalloc(8000);
   3: myallaoc(36000);
   4: myfree(36000);  //Dvs det block som allokerades på rad 2
   5: myalloc(6000);
   6: myalloc(6000); // Denna borde ge fel.
   ```

   b) Gör en simulering för fragmentering; allokera 20 minnesområden i slumpmässig (4 kB-2 MB) storlek, fortsätt och deallokera (tidigare allokerade) minnesområden, och reallokera i 100 omgångar. Hur fragmenterat blir minnet? (hur många fragment [minnesområden som har lediga utrymmen emellan] har vi)?

Intel Virtual Memory

Note: there is a figure that goes along with this web page; if you have popups disabled, it didn't appear. Here's a link to it.

In order to give an example of a practical, widely-used, and very nice multilevel paged virtual memory scheme, let's look at how Intel does it.

We'll discuss the following items:

1. Address Breakdown: how the address is divided for virtual memory lookups
2. Translation Algorithm: how a virtual address is translated into a physical address.
3. Format of Directory and Page Table Entries
4. Examples: one example of how the algorithm operates that I worked, and a second address for you to translate.

Address Breakdown

A 32 bit virtual address is divided into a ten bit page table number, a ten bit page number, and a twelve bit offset into a page:

Page Table Number	Page Number	Offset
31-22	21-12	11-0

Translation Algorithm

The "short form" of the translation algorithm is:

1. Use the page table number to index a directory, getting the address (in physical memory) of a page table. Notice that a process may have 1024 page tables.
2. Use the page number to index the page table, getting the address in physical memory of the page containing the data. Each page table also has 1024 entries.
3. Use the offset to index the page, getting the actual data.

In the "long form" of the algorithm, the numbers following correspond to the numbers in the figure in the popup:

1. The incoming virtual address is divided into a page table number, a page number, and an offset.
2. The process desriptor base register (`PDBR`) in the CPU tells where the directory starts.
3. The page table number is multiplied by four to use as an offset into the directory, and the directory entry is looked up.

4. The directory entry contains the address of the page table, and validity and protection information. If this information says that either the page table isn't present in memory or the protections aren't OK, the translation stops and an exception is raised.
5. The page number is multiplied by four to use as an offset into the page table, and the page table entry is looked up.
6. The page table entry contains the address of the page, and validity and protection information. If this information says that either the page isn't present in memory or the protections aren't OK, the translation stops and an exception is raised.
7. The offset is used as an index into the page.
8. The data is at the address finally arrived at.

Directory and Page Table Entry Format

Directory entries and page table entries share a common format:

PFA	Avail	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

Bits	Name	Interpretation
31-12	PFA	page frame address
11-9	Avail	available to OS
8	0	must be 0
7	L	PTE -- Must be 0. Dir Entry -- 4MB page
6	D	dirty (PTE only -- documented as undefined in directory entry)
5	A	accessed
4	PCD	page cache disable (can't cache data on this page)
3	PWT	page write transparent (tell external cache to use write-through strategy for this page)
2	U	user accessible
1	W	writeable
0	P	present

Note: "present" is actually checked first. If it's not present, the entire remainder of the PTE (or directory entry) is available.

Intel VM Translation Examples

Assume the following partial contents of memory for both examples. PDBR contains 001b3000.

Address	Contents
0x0001a038	0x000b4045
0x000b4b9c	0x236b12c1
0x000b91a0	0x1b9d8fc5
0x001b31cc	0x003a9067
0x001b3458	0x0001a067
0x003a9054	0x000b9067

Example 1: VM address 1cc151a0

1. Dividing the VM address up into two ten bit fields and a twelve bit field gives the following:
   Page Table Number: `0x073`
   Page Number: `0x015`
   Offset: `0x1a0`
2. According to the `PDBR`, the directory starts at `0x001b3000`.
3. We get the offset into the directory by multiplying the page table number by four (equivalently, left-shifting two places), giving `0x001b3000` + `0x1cc` = `0x001b31cc`.
4. This address contains `0x003a9067`. Decoding gives us:

Bit(s)	Contains	Means
31-12	`0x003a9`	Page Table starts at `0x003a9000`
11-7	`0x00`	Nothing relevant
6	`1`	Nothing in directory
5	`1`	Page has been accessed (lately)
4-3	`0`	Nothing relevant
2	`1`	User accessible
1	`1`	Writeable
0	`1`	Present

5. Adding the address of the page table (`0x003a9000`) to the offset into the page table (`0x0015` * 4 = `0x0x054`) gives `0x003a9054` as the address of the page table entry.
6. This address contains `0x000b9067`. Decoding gives us:

Bit(s)	Contains	Means
31-12	`0x000b9`	Page starts at `0x000b9000`
11-7	`0x00`	Nothing relevant
6	`1`	Page is dirty
5	`1`	Page has been accessed (lately)
4-3	`0`	Nothing relevant
2	`1`	User accessible
1	`1`	Writeable
0	`1`	Present

7. Adding the address of the start of the page (`0x000b9000`) to the offset (`0x1a0`) tells us the data is at `0x000b91a0` in physical memory.
8. The data at that address is `0x1b9d8fc5`