Virtual Memory

Without virtual memory addresses in a program tell you where, physically, to find data you are interested in.

Virtual memory breaks this direct relationship allowing complex rules for translating addresses used by your program (virtual addresses) into addresses locating data in memory (physical addresses).

MMUs provide the ability to do these translations in a way that is transparent to running programs. Traditionally virtual memory has been much more under the control of the OS than caches have.

Pages and Segments

Memory is divided into chunks, on which translations are similar.

pages All of memory divided into of fix sized chunks (pages). A *page table* describes how virtual pages are mapped on to physical pages and other metadata (eg. writable, read by OS).

segments Divide memory up into variable sized chunks based on intended use. Typically the number of segments will be small (16?). Access to segments through special instructions, registers etc.

These can be combined (eg. pages within segments).

Translation & TLB

Address will usually divided into page number and offset. Page number looked up in page table to get address of physical page and offset added to give final address.

To reduce cost of lookup a *Translation Lookaside Buffer* stores recent translations. Local and linear memory access are likely to be kind to TLB.

To store the page table as a simple array can be quite space consuming, so sometimes inverted or multilevel page tables are used.

Page Faults

Page faults occur when the MMU finds that the page table entry for the desired page is invalid. This does not have to be an error, the OS must decide what to do.

This may result in a page being loaded from disk, zeroed or maybe just he page table being changed. If it is an error then the program may be killed or signaled.

As a page fault involves at least going into the kernel and at worst waiting for a disk transfer. This makes them expensive.

VM features

protection By mapping only pages belonging to a process you can prevent processes changing one another's data.

demand paging Loading an entire program at startup may be wasteful. Demand paging only loads parts of a program as it is needed.

shared pages Two copies running copies of the same program can use the same pages of an executable. Data can also be shared if pages can be made *copy-on-write*.

memory mapping of files A file can be made to look like part of a processes memory. When a page is read it is first

copied in from the file. Good for sparse access to structured files. Also used for shared libraries.

sparse memory mapping When you `malloc` space it may not be allocated until you first read or write it.

paging More memory can be allocated to processes than is available as ram. By unmapping pages and copying them to disk the OS can free ram for new data. If the data on disk is needed again a page fault occurs and the data can be fetched from disk.

swapping If the OS has run very short of memory whole processes may be copied to disk to free ram.

VM management

How VM is managed has a large effect on the performance of a busy computer. For many applications adding ram may have improve performance more than adding a faster CPU.

- Merged disk cache & VM system,
- better allocators,
- OPT, LRU, etc.
- read ahead,
- prepaging.

References

- http://www.inf.fu-berlin.de/lehre/WS94/RA/RISC-9.html
- http://www.eng.dmu.ac.uk/~pdn/UltraSPARC/ultra_arch_architecture.html
- Any OS book.
- http://www.backplane.com/FreeBSD/FreeBSDVM.txt
- http://www.backplane.com/FreeBSD/LinuxVM.txt

Assignment

Take your dot product program and split it into two files, one containing just the dot product function.

```
double dotprod(double *a,double*b,int len);
```

Use a `Makefile` to compile the files into the final program. Experiment with different levels of optimisation, and examine the outputted assembly for the dot product.

Optimisation: `-O1` ... `-O6`, Produce Assebley: `-S`, Produce Object File: `-c`.