Virtual Memory

- So far when discussing memory addresses we haven't been especially realistic
- When code is compliled, the addresses of variables are hardcoded into the executable
- If these were the actual physical locations in memory this would make running two copies of the code at the same time difficult (impossible!)
- A value updated in one process would modify the physical memory which was also being used by the second copy
- Obviously, since we can run multiple copies of the same executable concurrently this is not what happens

Virtual Memory

- Virtual memory adds a layer of abstaction between the process and the hardware
- We translate the addresses in your code (virtual addresses) into addresses locating information in real memory (physical addresses)
- This mapping is different for each executing process
- The Memory Management Unit (MMU) performs these translations in a transparent way for the executing program

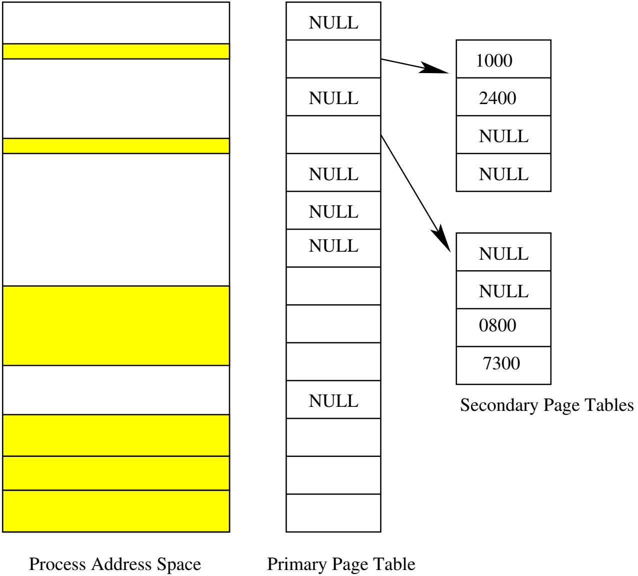
Memory Pages and Page Tables

- Physical memory is divided into fixed sized chunks called pages
- Size of pages depends on the hardware — typical sizes are 512 bytes to 4K
- All addressable locations in memory are in a page
- Therefore each location can be addressed by a page number and and offset in the page
- The page tables contains the mapping from virtual memory to physical memory
- Two separate page tables for kernel and user activities
- A set of page tables for each executing process

The Page Table

- Most modern page tables are arranged in a multi-level hierarchy
- This hierarchy saves spaces but has a small performance hit
- Useful because not every part of a processes address space needs to be mapped to physical memory
- Most accesses in memory are adjacent — on the heap, the stack (more on these later)
- In a 32-bit system you might have a primary table with 2^{12} entries each of which point to a secondary table with 2^{8} each pointing at a 2^{12} byte page of physical memory
- The page table also stores some meta-data about the page including clean/dirty, present, process id etc.

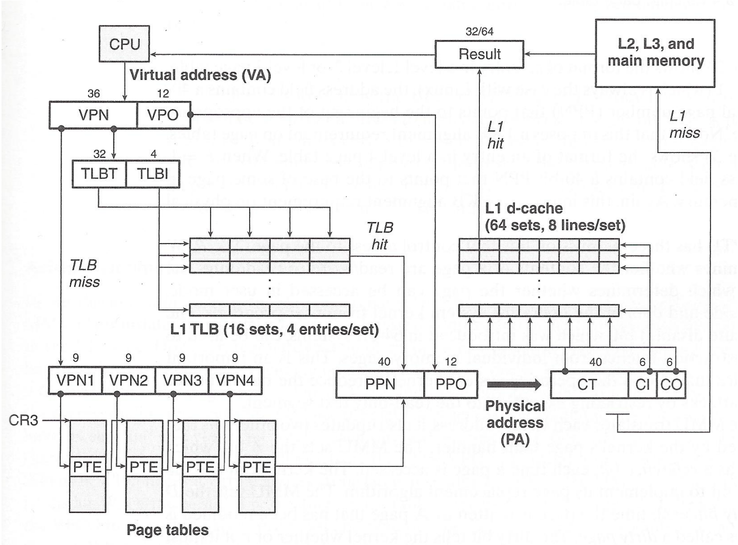
Page Tables


NULL
NULL
NULL
NULL
NULL
NULL
1000
2400
NULL
NULL
NULL
NULL
0800
7300
Secondary Page Tables
Process Address Space
Primary Page Table

Translation Lookaside Buffer

- To reduce the cost of lookups the translation lookaside buffer (TLB) stores recent page table translations
- Implemented in hardware (see over)
- Usually achieve hit rates of 95%
- Linear memory accesses are nice to the TLB. Exploit locality of reference yet again!

Translation Lookaside Buffer



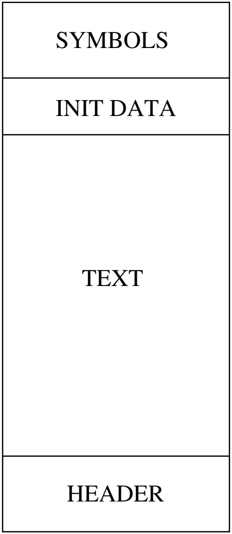
Stolen from Bryant and O'Hallaron

Processes

- A process is the execution of a program
- Consists of three types of data
  - *text* the CPU instructions
  - *data* where malloc'ed variables are kept
  - *stack* where local variables and other housekeeping is kept
  - Actually two stacks — user and kernel
- Operates in either user mode or kernel mode
- In kernel mode the process has full control of the system
- Can only get into kernel mode through well defined system calls

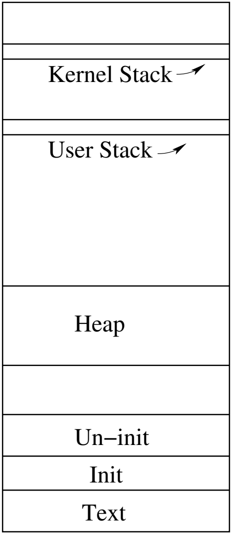
Executables

- Four regions
- Symbols — used for debugging
- Initialised data — global variables with values
- Text — the actual instructions to be executed
- Header — basic information needed to start the process



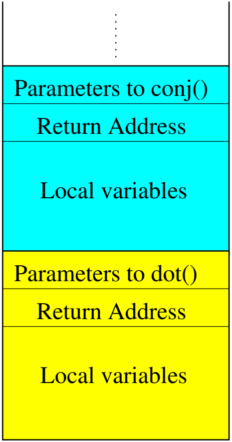
Processes

- Six regions
- Kernel Stack
- User Stack
- Heap
- Uninitialised data
- Initialised data
- Text



Stack Frames

- Each function that is called generates a new frame on the stack which contains
  - The return address from the function
  - Local variables
  - Parameters to the function
- Say we have a function conj() which calls dot()



The Stack

- The stack has a maximum size that is set as the program starts executing
- We can examine this using `ulimit -s`
- If we use more than this amount of memory our program will crash. This is called stack overflow
- Two main ways to do this
  - Declare large stack variables. Create a local array such as `double x[100000]`
  - Too many function calls. Each function call uses a small amount of memory for parameters, saved variables and a return value. Infinite recursion is usually to blame here.

Growing the heap

- When you allocate more memory using malloc() it appears on the heap
- The system uses brk() or sbrk() to extend the size of the heap
- brk() performs some checks and then calls growreg() to create new page tables for the region (if required)
- Passing sbrk() a negative argument decreases the size of the heap and releases page tables that are no longer required
- Can uses sbrk(0); to find out where the heap currently ends

Page Faults

- Page faults occur when the MMU finds the page table entry for the required page is invalid. This isn't always an error
- May have to
  - Load a page from disk
  - Zero the existing page
  - Kill the process
- Since a page fault involves talking to the kernel and a possible disk transfer, they are expensive

Page Faults

- Several types of page faults can occur
- DISK — need to look on disk at a given location for the data
- NULL — physical memory not yet allocated
- MEM — protection level fault
- IOP — wait for I/O to complete
- LOCK — trying to modify a locked page

VM Benefits

- Protection — by only mapping pages belong to a process you can prevent access to another's data
- Demand paging — loading an entire program at once is wasteful. Demand paging only loads parts of a program as needed
- Shared pages — two processes running the same program can make use of the same pages of an executable
- Memory Mapped Files — a file can be made to look like part of a process' address space. When a page is read it is first copied in from the file. Used for shared libraries

Compilation Process

- Generating an executable consists of four phases
  - Pre-processing
  - Compilation
  - Assembling
  - Linking

Compilation Process

- The pre-processing phase involves all the hash statements in the program - `#include`, `#define`, `#pragma`
- The output from this phase can be seen using `gcc -E`
- Possible to do clever things here with `#ifdef` statments
- Pass arguments to the compiler using `gcc -DFOO`

Compilation Process

- Compilation takes the output from the pre-processor and turns it into assembly code
- This itself is a five stage process
  1. Lexical Analysis
  2. Parsing
  3. Semantic Analysis
  4. Optimization
  5. Code Generation
- Assembly code can be generated by using `gcc -S`

Compilation Process

- Assembling turns the assembly code into machine instructions specific for the processor architecture
- Normally just a straight translation
- There might be some further minor changes to the code at this stage (reordering, padding)
- Object code is generated using `gcc -c`

Compilation Process

- The final stage of generating an executable is linking
- This allows multiple object files to be combined into a single program
- Linking uses the `ld` command, but most of the time we just let `gcc` handle it for us
- Linking is also the time when references to external libraries are resolved
- Two forms of linking
  - Static linking - all the object code and libraries are rolled into a single executable
  - Dynamic linking - the object code is merged but references to libraries are left as pointers

Shared Libraries

- Shared libraries are loaded at the start of execution of a program
- The contain the program code that is to be added to the executable
- Shared libraries live in pre-definied locations on the filesystem — check `/etc/ld.so.conf`
- In modern systems this file usually includes additional files from a sub-directory
- Can add additional locations by modifying the `LD_LIBRARY_PATH` environment variable

Shared Libraries

- Shared libraries have three 'names'
  - real name: libgmp.so.3.5.0
  - soname: libgmp.so.3
  - compiler name: gmp
- The soname is used by the linker to determine which version of a library to use
- The compiler name is what you use as part of your `gcc` command
- `gcc -o foo foo.c -lgmp`

Shared Libraries

- To make your own shared libraries you recompile your code to generate it as position independent (or relocatable) code.
- Use either `gcc -fpic` or `gcc -fPIC`
- The position of the code in your process is defined in the global offset table (GOT)
- `-fpic` uses a machine-specific maximum size for the GOT. If your library is large your code might exceed the GOT
- `-fPIC` ignores the maximum GOT size but ends up producing larger code than `-fpic`

Compilation Process

- Once an executable has been created you can examine what shared libraries it requires by running `ldd a.out`

```
[dfrost@chuck ~]$ ldd ./a.out
linux-vdso.so.1 =>  (0x00007fffe87c9000)
libc.so.6 => /lib64/libc.so.6 (0x0000003fe1600000)
/lib64/ld-linux-x86-64.so.2 (0x0000003fe1200000)
```

- Can replace these references at runtime using the `LD_PRELOAD` mechanism
- Allows a user to replace function calls from libraries with their own version