

6.828 2014 Lecture 7: using virtual memory ==

- plan: cool things you can do with vm
 - kernel tricks (e.g., one zero-filled page)
 - faster system calls (e.g., copy-on-write fork)
 - new features (e.g., memory-mapped files)
 - JOS and VM
 - ideas for last lab (final project)
- virtual memory: several views
 - primary purpose: isolation each process has its own address space
 - Virtual memory provides a level-of-indirection provides kernel with opportunity to do cool stuff
- lazy/on-demand page allocation
 - sbrk() is old fashioned; it asks application to "predict" how much memory they need difficult for applications to predict how much memory they need in advance sbrk allocates memory that may never be used.
 - moderns OSes allocate memory lazily allocate physical memory when application needs it
 - HW solution
- share kernel page tables in xv6
 - observation: kvmalloc() allocates new pages for kernel page table for each process but all processes have the same kernel page table
 - idea: modify kvmalloc()/freem() to share kernel page table
- guard page to protect against stack overflow
 - put a non-mapped page below user stack if stack overflows, application will see page fault
 - allocate more stack when application runs off stack into guard page
- one zero-filled page
 - kernel often fills a page with zeros
 - idea: memset *one* page with zeros map that page copy-on-write when kernel needs zero-filled page on write make copy of page and map it read/write in app address space
- copy-on-write fork
 - observation: xv6 fork copies all pages from parent (see fork()) but fork is often immediately followed by exec
 - idea: share address space between parent and child modify fork() to map pages copy-on-write (use extra available system bits in PTEs and PDEs) on page fault, make copy of page and map it read/write
- demand paging
 - observation: exec loads the complete file into memory (see exec.c) expensive: takes time to do so (e.g., file is stored on a slow disk) unnecessary: maybe not the whole file will be used
 - idea: load pages from the file on demand allocate page table entries, but mark them on-demand on fault, read the page in from the file and update page table entry
 - challenge: file larger than physical memory (see next idea)
- use virtual memory larger than physical memory
 - observation: application may need more memory than there is physical memory
 - idea: store less-frequently used parts of the address space on disk page-in and page-out

- pages of the address address space transparently
 - works when working sets fits in physical memory
- memory-mapped files
 - idea: allow access to files using load and store can easily read and writes part of a file e.g., don't have to change offset using lseek system call
 - page-in pages of a file on demand when memory is full, page-out pages of a file that are not frequently used
- shared virtual memory
 - idea: allow processes on different machines to share virtual memory gives the illusion of physical shared memory, across a network
 - replicate pages that are only read
 - invalidate copies on write
- JOS and virtual memory
 - layout: [picture](#)

- UVPT trick (lab 4) recursively map PD at 0x3BD virtual address of PD is $(0x3BD \ll 22) | (0x3BD \ll 12)$ if we want to find pte for virtual page n, compute

```
pde_t uvpt[n], where uvpt is (0x3BD << 22)
= uvpt + n * 4 (because pde is a word)
= (0x3BD << 22) | (top 10 bits of n) | (bottom 10 bits of n) << 2
= 10 | 10 | 12
```

for example, uvpt[0] is address $(0x3BD \ll 22)$, following the pointers gives us the first entry in the page directory, which points to the first page table, which we index with 0, which gives us pte 0

simpler(?) than pgdirwalk()

- user-level copy-on-write fork (lab4) JOS propagates page faults to user space user programs can play similar VM tricks as kernel! you will do user-level copy-on-write fork Edit By [MaHua](#)