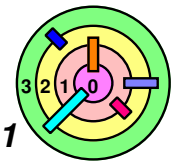
Kernel 3

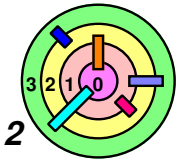
Bill Cheng

http://merlot.usc.edu/william/usc/



New Things in Kernel 3

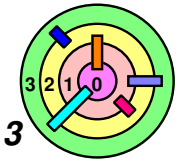
- Polymorphism go recursive!
  - fortunately, any recursion can be turned into a loop
    - so, think about the recursion as a loop
    - when you traverse a linked-list of shadow objects, you will eventually reach *bottom object* (and the recursion must stop)
- Address space
  - address space is implemented with *virtual memory maps*
    - review Ch 7 slides and *read kernel 3 FAQ*
- Handle page faults
  - keep Ch 7 slides in mind
- System call and system call support
  - `fork()` - where everything comes together
- Debugging
  - you may have to single-step machine instructions to find bugs



Kernel 3

```
% make nyi
```

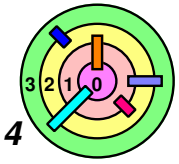
mm/pframe.c:359	pframe_get()	VM
mm/pframe.c:379	pframe_pin()	VM
mm/pframe.c:395	pframe_unpin()	VM
api/syscall.c:77	sys_read()	VM
api/syscall.c:87	sys_write()	VM
api/syscall.c:103	sys_getdents()	VM
api/access.c:144	addr_perm()	VM
api/access.c:160	range_perm()	VM
proc/fork.c:77	do_fork()	VM
vm/pagefault.c:72	handle_pagefault()	VM
vm/shadow.c:73	shadow_init()	VM
vm/shadow.c:85	shadow_create()	VM
vm/shadow.c:97	shadow_ref()	VM
vm/shadow.c:111	shadow_put()	VM
vm/shadow.c:126	shadow_lookuppage()	VM
vm/shadow.c:144	shadow_fillpage()	VM
vm/shadow.c:153	shadow_dirtypage()	VM
vm/shadow.c:160	shadow_cleanpage()	VM
proc/kthread.c:161	kthread_clone()	VM
fs/vn_special.c:142	special_file_mmap()	VM
fs/vn_special.c:155	special_file_fillpage()	VM
fs/vn_special.c:167	special_file_dirtypage()	VM
fs/vn_special.c:179	special_file_cleanpage()	VM



Kernel 3

```
% make nyi
```

```
vm/vmmap.c:129          vmmap_create()          VM
vm/vmmap.c:138         vmmap_destroy()          VM
vm/vmmap.c:148          vmmap_insert()          VM
vm/vmmap.c:161       vmmap_find_range()         VM
vm/vmmap.c:171          vmmap_lookup()          VM
vm/vmmap.c:182          vmmap_clone()           VM
vm/vmmap.c:215            vmmap_map()           VM
vm/vmmap.c:251         vmmap_remove()           VM
vm/vmmap.c:262   vmmap_is_range_empty()         VM
vm/vmmap.c:277          vmmap_read()            VM
vm/vmmap.c:292         vmmap_write()            VM
   vm/brk.c:76               do_brk()           VM
  vm/anon.c:60           anon_init()             VM
  vm/anon.c:72         anon_create()             VM
  vm/anon.c:84            anon_ref()             VM
  vm/anon.c:98            anon_put()             VM
 vm/anon.c:106     anon_lookuppage()             VM
 vm/anon.c:115       anon_fillpage()             VM
 vm/anon.c:122      anon_dirtypage()             VM
 vm/anon.c:129      anon_cleanpage()             VM
  vm/mmap.c:55             do_mmap()             VM
  vm/mmap.c:70           do_munmap()             VM
```



/usr/bin/hello

Page frame management

mm/pframe.c:359	pframe_get()	VM
mm/pframe.c:379	pframe_pin()	VM
mm/pframe.c:395	pframe_unpin()	VM

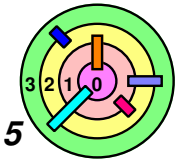
User address space implementation

vm/vmmap.c:129	vmmap_create()	VM
vm/vmmap.c:138	vmmap_destroy()	VM
vm/vmmap.c:148	vmmap_insert()	VM
vm/vmmap.c:161	vmmap_find_range()	VM
vm/vmmap.c:171	vmmap_lookup()	VM
vm/vmmap.c:182	vmmap_clone()	VM
vm/vmmap.c:215	vmmap_map()	VM
vm/vmmap.c:251	vmmap_remove()	VM
vm/vmmap.c:262	vmmap_is_range_empty()	VM
vm/vmmap.c:277	vmmap_read()	VM
vm/vmmap.c:292	vmmap_write()	VM

- some of them are needed by the loader in "`api/elf32.c`" since the loader creates an address space for a user program

Handle page faults

vm/pagefault.c:72	handle_pagefault()	VM



/usr/bin/hello

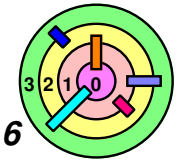
Basic system calls and system call support

api/syscall.c:77	sys_read()	VM
api/syscall.c:87	sys_write()	VM
api/syscall.c:103	sys_getdents()	VM
api/access.c:144	addr_perm()	VM
api/access.c:160	range_perm()	VM

Memory management (mmobj)

vm/anon.c:60	anon_init()	VM
vm/anon.c:72	anon_create()	VM
vm/anon.c:84	anon_ref()	VM
vm/anon.c:98	anon_put()	VM
vm/anon.c:106	anon_lookuppage()	VM
vm/anon.c:115	anon_fillpage()	VM
vm/anon.c:122	anon_dirtypage()	VM
vm/anon.c:129	anon_cleanpage()	VM

- recall that there are 3 types of `mmobj`
  - one lives inside the `vnode`
  - anonymous object
  - shadow object



/usr/bin/hello

- this may not be the complete list (I was told that these are what you need)

Basic system calls and system call support

api/syscall.c:77	sys_read()	
api/syscall.c:87	sys_write()	
api/syscall.c:103	sys_getdents()	
api/access.c:144	addr_perm()	VM
api/access.c:160	range_perm()	VM

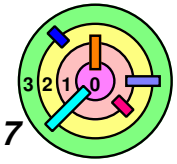
Memory management (mmobj)

"hello"

vm/anon.c:60	anon_init()	VM
vm/anon.c:72	anon_create()	VM
vm/anon.c:84	anon_ref()	VM
vm/anon.c:98	anon_put()	VM
vm/anon.c:106	anon_lookuppage()	VM
vm/anon.c:115	anon_fillpage()	VM
vm/anon.c:122	anon_dirtypage()	VM
vm/anon.c:129	anon_cleanpage()	VM

- recall that there are 3 types of `mmobj`
  - one lives inside the `vnode`
  - anonymous object
  - shadow object

To get "`hello`" to work, you need to get all of the above to work (except for `sys_read()` and `sys_getdents()`)



Beyond `/usr/bin/hello`

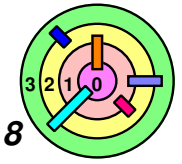
More memory management (mmobj)

vm/shadow.c:73	shadow_init()	VM
vm/shadow.c:85	shadow_create()	VM
vm/shadow.c:97	shadow_ref()	VM
vm/shadow.c:111	shadow_put()	VM
vm/shadow.c:126	shadow_lookuppage()	VM
vm/shadow.c:144	shadow_fillpage()	VM
vm/shadow.c:153	shadow_dirtypage()	VM
vm/shadow.c:160	shadow_cleanpage()	VM

- shadow object is complicated
  - only really need this when you get to run "`fork-and-wait`"

More system calls and system call support

proc/fork.c:77	do_fork()	VM
proc/kthread.c:161	kthread_clone()	VM
vm/mmap.c:55	do_mmap()	VM
vm/mmap.c:70	do_munmap()	VM
vm/brk.c:76	do_brk()	VM
fs/vn_special.c:142	special_file_mmap()	VM
fs/vn_special.c:155	special_file_fillpage()	VM
fs/vn_special.c:167	special_file_dirtypage()	VM
fs/vn_special.c:179	special_file_cleanpage()	VM



Beyond /usr/bin/hello

More memory management (mmobj)

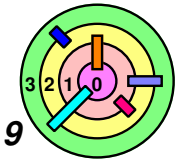
vm/shadow.c:73	shadow_init()	VM
vm/shadow.c:85	shadow_create()	VM
vm/shadow.c:97	shadow_ref()	VM
vm/shadow.c:111	shadow_put()	VM
vm/shadow.c:126	shadow_lookuppage()	VM
vm/shadow.c:144	shadow_fillpage()	VM
vm/shadow.c:153	shadow_dirtypage()	VM
vm/shadow.c:160	shadow_cleanpage()	VM

- shadow object is complicated
  - only really need this when you get to run "fork-and-wait"

More system calls and system call support

proc/fork.c:77	do_fork()	VM
proc/kthread.c:161	kthread_clone()	VM

vm/mmap.c:55	do_mmap()	VM
vm/mmap.c:70	do_munmap()	VM
vm/brk.c:76	do_brk()	VM
fs/vn_special.c:142	special_file_mmap()	VM
fs/vn_special.c:155	special_file_fillpage()	VM
fs/vn_special.c:167	special_file_dirtypage()	VM
fs/vn_special.c:179	special_file_cleanpage()	VM



Beyond `/usr/bin/hello`

More memory management (mmobj)

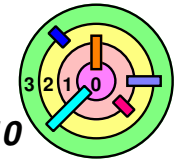
vm/shadow.c:73	shadow_init()	VM
vm/shadow.c:85	shadow_create()	VM
vm/shadow.c:97	shadow_ref()	VM
vm/shadow.c:111	shadow_put()	VM
vm/shadow.c:126	shadow_lookuppage()	VM
vm/shadow.c:144	shadow_fillpage()	VM
vm/shadow.c:153	shadow_dirtypage()	VM
vm/shadow.c:160	shadow_cleanpage()	VM

- shadow object is complicated
  - only really need this when you get to run "`fork-and-wait`"

More system calls and system call support

"`/sbin/init`"

proc/fork.c:77	do_fork()	VM
proc/kthread.c:161	kthread_clone()	VM
vm/mmap.c:55	do_mmap()	VM
vm/mmap.c:70	do_munmap()	VM
vm/brk.c:76	do_brk()	VM
fs/vn_special.c:142	special_file_mmap()	VM
fs/vn_special.c:155	special_file_fillpage()	VM
fs/vn_special.c:167	special_file_dirtypage()	VM
fs/vn_special.c:179	special_file_cleanpage()	VM

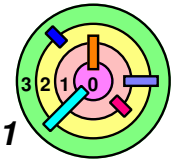


Strategy

- You must implement page frame management first

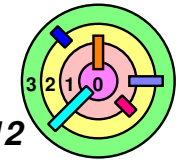
`mm/pframe.c:359`	`pframe_get()`	`VM`
`mm/pframe.c:379`	`pframe_pin()`	`VM`
`mm/pframe.c:395`	`pframe_unpin()`	`VM`

  - set `S5FS=1` but keep `VM=0` in `Config.mk` (even though this is kernel 3)
    - `VM=1` means that you are running user-space programs
  - the re-run `vfstest` and make sure everything works
    - `s5fs` has a real disk and `ramfs` didn't
  - don't bother with tests in "`faber_fs_test.c`" because they were designed for kernel 2 only
  - you should get this done by *end of week 12*
    - if code is mostly working but cannot past all `vfstest`, split up your team
      - one to debug code in "`pframe.c`", the rest move forward



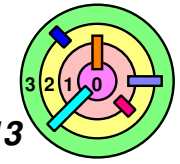
Strategy

- In the *first week* of kernel 3, you should get the *first user-space program* (i.e., "`hello`") to run to run
  - in `initproc_run()`, run "`/usr/bin/hello`" using `kernel_execve()`
    - the kernel's `INIT` process becomes the user-space "`hello`" process
    - you have to be able to *load* the program into memory
    - you have to be able to *build and manipulate* the *address space*
    - you have to be able to *handle page faults*
    - you have to get some *system call* (such as `write()`) to work
  - that's a lot of stuff to get working just to run "hello"
    - please do *not* even attempt to run "`/sbin/init`"
  - would be good to get this done by the *end of week 1 of kernel 3*



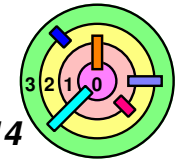
Strategy

- In the *2nd week* of kernel 3, you should get all the user-space programs in section (B) of the grading guidelines running from `initproc_run()`
  - the last one is "`/usr/bin/fork_and_wait`"
    - you need to get `fork()` to work
  - by the end of the weekend, you should get "`/sbin/init`" to run from `initproc_run()`
    - the kernel's `INIT` process becomes the user-space `INIT` process
      - the user-space `INIT` process spawn child processes to run user-space shell ("`/bin/sh`")
      - from the user-space shell ("`/bin/sh`"), you should be able to re-run all the section (B) user programs
  - would be good to get this done by the *end of weekend of the 2nd week of kernel 3*
- In the *last week* of kernel 3, get everything else to work



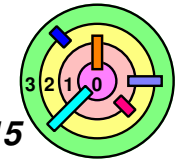
pframe.c

```
/*
 * Find and return the pframe representing the page identified by the
 * object and page number. If the page is already resident in memory,
 * then we return the existing page. Otherwise, we allocate a new page
 * and fill it (in which case this routine may block). Before allocating
 * the new pframe, we check to see if we need to call pageoutd and wake
 * it up if necessary.
 *
 * If the page is found (resident) but busy, then we will wait for it to
 * become unbusy and then try again (since it may have been freed after
 * that). Thus, as long as this routine returns successfully, the returned
 * page will be a non-busy page that will be guaranteed to remain resident
 * until the calling context blocks without first pinning the page.
 *
 * This routine may block at the mmobj operation level.
 *
 * @param o the parent object of the page
 * @param pagenum the page number of this page in the object
 * @param result used to return the pframe (NULL if there's an error)
 * @return 0 on success, < 0 on failure.
 */
int
pframe_get(struct mmobj *o, uint32_t pagenum, pframe_t **result)
{
    NOT_YET_IMPLEMENTED("VM: pframe_get");
    return 0;
}
```



pframe.c

```
/*
 * Increases the pin count on this page. Pages with a pin count > 0 will
 * not be paged out by pageoutd, so this ensures that the page will remain
 * resident until the pin count is decreased.
 *
 * If the pframe has not yet been pinned, remove this pframe's list link
 * from the allocated list and add it to the pinned list.  Be sure to
 * decrement nallocated and increment npinned.
 *
 * In either case, increment the pf_pincount.
 *
 * @param pf the page to pin
 */
int
pframe_pin(pframe_t *pf)
{
    NOT_YET_IMPLEMENTED("VM: pframe_pin");
    return 0;
}
```



pframe.c

```
/*
 * Decreases the pin count on a page. If the pin count reaches zero,
 * then the page could be paged out any time after the calling context
 * blocks.
 *
 * If the pin count reaches zero, move the pframe's list link from the
 * pinned list to the allocated list.  Be sure to correctly update
 * npinned and nallocated
 *
 * @param pf a pinned page (a page with a positive pin count)
 */
int
pframe_unpin(pframe_t *pf)
{
    NOT_YET_IMPLEMENTED("VM: pframe_unpin");
    return 0;
}
```