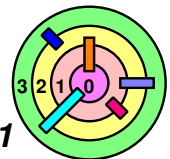


Kernel 1

Bill Cheng

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

- ➡ Prerequisite: a simple system (Ch 4)
- when you finish the prerequisite, please come back and review this material

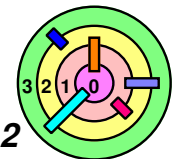


Boot Prestine Kernel

```
% script
Script started, file is typescript
reading .login (xterm) ...
% ./weenix -n
/usr/bin/qemu-system-i386
...
Not yet implemented: PROCS: bootstrap, file \
main/kmain.c, line 184
panic in main/kmain.c:186 bootstrap(): weenix \
returned to bootstrap()!!! BAD!!!
```

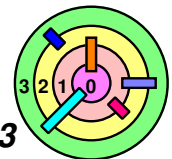
Kernel Halting.

```
^C
qemu: terminating on signal 2
% exit
Script done, file is typescript
% more typescript
```



Look At "typescript"

- ➡ Where is kernel's text segment?
 - ▬ `0xc0000000-0xc0039000` means `[0xc0000000, 0xc0039000)`
 - this interval is "closed" on the left (i.e., includes) and "open" on the right (i.e., excludes)
 - ◆ pretty much all intervals are denoted this way
- ➡ Where are the other segments?
 - ▬ data: `[0xc0039000-0xc0045b6a)`
 - ▬ bss: `[0xc0045b6a-0xc005b000)`
 - ▬ page system: `[0xc00a0000-0xcfe9d000)`
 - I think this is like the dynamic segment for the kernel (i.e., Buddy System)
- ➡ If your numbers are slightly different, it's probably okay



Look At "typescript"



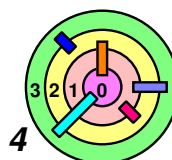
What else?

- kernel data structures are initialized (e.g., memory allocators)
- some hardware are initialized (e.g., PIC)
 - *don't need to understand EVERYTHING*
- if your kernel access anywhere outside this range, most likely you will get a **kernel page fault**
 - this will be followed by a **kernel panic**



VERY IMPORTANT: read kernel FAQ about *"how can I debug a page fault that caused kernel panic?"*

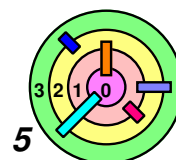
- whenever you get a kernel page fault, you must follow the steps there to figure out **EXACTLY** where your kernel crashed
 - I have no special power, I have no idea why your kernel crashed



Kernel 1

% make nyi

proc/kmutex.c:36	kmutex_init()	PROCS
proc/kmutex.c:48	kmutex_lock()	PROCS
proc/kmutex.c:58	kmutex_lock_cancellable()	PROCS
proc/kmutex.c:78	kmutex_unlock()	PROCS
proc/kthread.c:106	kthread_create()	PROCS
proc/kthread.c:127	kthread_cancel()	PROCS
proc/kthread.c:148	kthread_exit()	PROCS
proc/proc.c:223	proc_create()	PROCS
proc/proc.c:254	proc_cleanup()	PROCS
proc/proc.c:268	proc_kill()	PROCS
proc/proc.c:280	proc_kill_all()	PROCS
proc/proc.c:294	proc_thread_exited()	PROCS
proc/proc.c:315	do_waitpid()	PROCS
proc/proc.c:328	do_exit()	PROCS
proc/sched.c:121	sched_cancellable_sleep_on()	PROCS
proc/sched.c:137	sched_cancel()	PROCS
proc/sched.c:179	sched_switch()	PROCS
proc/sched.c:198	sched_make_runnable()	PROCS
proc/sched_helper.c:43	sched_sleep_on()	PROCS
proc/sched_helper.c:49	sched_wakeup_on()	PROCS
proc/sched_helper.c:56	sched_broadcast_on()	PROCS
main/kmain.c:184	bootstrap()	PROCS
main/kmain.c:279	initproc_create()	PROCS
main/kmain.c:298	initproc_run()	PROCS



Kernel 1

➡ Kernel thread creation, cancellation, and destruction

proc/kthread.c:106	kthread_create()	PROCS
proc/kthread.c:127	kthread_cancel()	PROCS
proc/kthread.c:148	kthread_exit()	PROCS

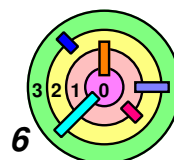
— please keep MPT=0 in Config.mk and **only** implement *single thread processes*

➡ Kernel scheduler

proc/sched.c:121	sched_cancellable_sleep_on()	PROCS
proc/sched.c:137	sched_cancel()	PROCS
proc/sched.c:179	sched_switch()	PROCS
proc/sched.c:198	sched_make_runnable()	PROCS
proc/sched_helper.c:43	sched_sleep_on()	PROCS
proc/sched_helper.c:49	sched_wakeup_on()	PROCS
proc/sched_helper.c:56	sched_broadcast_on()	PROCS

➡ Kernel process creation, waiting, and destruction

proc/proc.c:223	proc_create()	PROCS
proc/proc.c:254	proc_cleanup()	PROCS
proc/proc.c:268	proc_kill()	PROCS
proc/proc.c:280	proc_kill_all()	PROCS
proc/proc.c:294	proc_thread_exited()	PROCS
proc/proc.c:315	do_waitpid()	PROCS
proc/proc.c:328	do_exit()	PROCS



Kernel 1

➡ Kernel mutex

proc/kmutex.c:36	kmutex_init()	PROCS
proc/kmutex.c:48	kmutex_lock()	PROCS
proc/kmutex.c:58	kmutex_lock_cancellable()	PROCS
proc/kmutex.c:78	kmutex_unlock()	PROCS

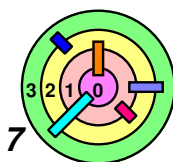
➡ Kernel startup

main/kmain.c:184	bootstrap()	PROCS
main/kmain.c:279	initproc_create()	PROCS
main/kmain.c:298	initproc_run()	PROCS

➡ Read the *comment blocks* to figure out what these functions suppose to do and how they are related to each other

- use "grep" to see how they are called
- feel free to discuss this in the class Google Group
- you can talk about code that came with the prestine kernel
 - do *not* post code/pseudo-code you are planning to write

➡ When a thread gives up the CPU, you must make sure that all *global variables* and *data structures* are in a *consisten state*

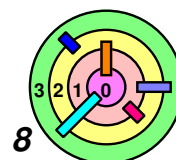


How To Start?



I would recommend doing this first:

- phase 1: get the kernel to simply start and quit with `DRIVERS=0`
 - in `bootstrap()`, create IDLE proc and switch to IDLE proc
 - ◆ first procedure of IDLE proc is written for you already, don't change anything
 - ◆ *Hint*: look at the code in `faber_thread_test()` to see how to create a process, create a thread in it, run it
 - ◆ Note: code in `faber_thread_test()` is running in *thread context*, you are *not* in a thread context in `bootstrap()`
 - in `initproc_create()`, create INIT proc and INIT thread
 - don't write code in `initproc_run()`, which is the first procedure of INIT proc
 - ◆ it should self-terminate
 - ◆ single-step to see where it goes
 - this should wake up the IDLE proc, which will turn off the machine



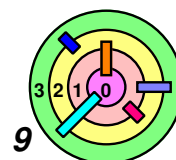
How To Start?

➡ I would recommend doing this first:

- phase 1: get the kernel to simply start and quit with DRIVERS=0
 - make sure the *kernel halts cleanly*
 - these functions are involved (the list may *not* be complete):

proc/kthread.c:106	kthread_create()	PROCS
proc/kthread.c:148	kthread_exit()	PROCS
proc/proc.c:223	proc_create()	PROCS
proc/proc.c:254	proc_cleanup()	PROCS
proc/proc.c:294	proc_thread_exited()	PROCS
proc/proc.c:315	do_waitpid()	PROCS
proc/sched.c:121	sched_cancellable_sleep_on()	PROCS
proc/sched.c:179	sched_switch()	PROCS
proc/sched.c:198	sched_make_runnable()	PROCS
proc/sched_helper.c:43	sched_sleep_on()	PROCS
proc/sched_helper.c:49	sched_wakeup_on()	PROCS
proc/sched_helper.c:56	sched_broadcast_on()	PROCS
main/kmain.c:184	bootstrap()	PROCS
main/kmain.c:279	initproc_create()	PROCS

- yes, it's a lot of code to get working just for phase 1!
- in a way, phase 1 is the most important step
 - ◆ if you do it the wrong way, you will have to come back and fix your code

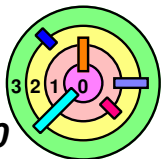


How To Start?



I would recommend doing this first:

- phase 2: get the kernel to simply start and quit *cleanly*
 - call `faber_thread_test()` from `initproc_run()`
 - ◆ start with `CS402TESTS=1` in `Config.mk`
 - ◆ then set `CS402TESTS=2, 3`, and so on, up to 8
 - always make sure the *kernel halts cleanly*
- phase 3: set `DRIVERS=1` in `Config.mk`
 - run `kshell` in `initproc_run()`
 - ◆ "help", "echo" and "exit" `kshell` commands should work
 - add `kshell` commands to invoke any test function in grading guidelines and your README (see rules about "SELF-checks")
 - ◆ for each `kshell` command, you need to *create a child process* and set the test function as the *first procedure* of the thread in the child process



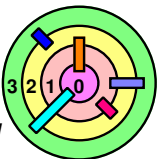
How To Start?

➡ Please remember that the teaching staff *cannot tell you what code to write*

➡ Here's what's appropriate to talk about in class Google Group

- 1) the spec
- 2) the kernel FAQ
- 3) the grading guidelines
- 4) test code

- kernel 1: `faber_thread_test()`, `sunghan_test()`, `sunghan_deadlock_test()`
 - ◆ they are mentioned in the grading guidelines
 - ◆ your kernel needs to work with these test code perfectly



Hints?



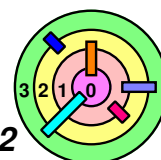
Hints are all over the place!

- documentation
- spec
- comment in code
- kernel code itself
- kernel FAQ



For example, the spec says:

- the kernel is very very powerful
 - if there is a bug, it's *your* bug!
- the weenix kernel is *non-preemptive*
 - *non-preemptive* means that a thread *cannot be preempted* by *another thread*
 - ◆ it can be interrupted to *service an interrupt*, then goes back to what it was doing before
 - ◆ if a kernel thread *does not want to be cancelled*, there is *no way* to kill it
- we are *not* implementing multiple threads per process, i.e., MTP=0 in Config.mk



"kmain.c"

kmain()

```
context_setup(&bootstrap_context, bootstrap, 0, NULL, bstack,  
             PAGE_SIZE, bpdire);  
context_make_active(&bootstrap_context);  
panic("\nReturned to kmain()!!!\n");
```

bootstrap()

```
NOT_YET_IMPLEMENTED("PROCS: bootstrap");  
panic("weenix returned to bootstrap()!!! BAD!!!\n");
```

idleproc_run()

```
kthread_t *initthr = initproc_create();  
init_call_all();  
GDB_CALL_HOOK(initialized);  
intr_enable();  
sched_make_runnable(initthr);  
child = do_waitpid(-1, 0, &status);
```

initproc_create()

```
NOT_YET_IMPLEMENTED("PROCS: initproc_create");
```

initproc_run()

```
NOT_YET_IMPLEMENTED("PROCS: initproc_run");
```

