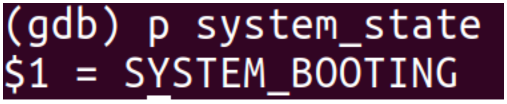
内核调试实验

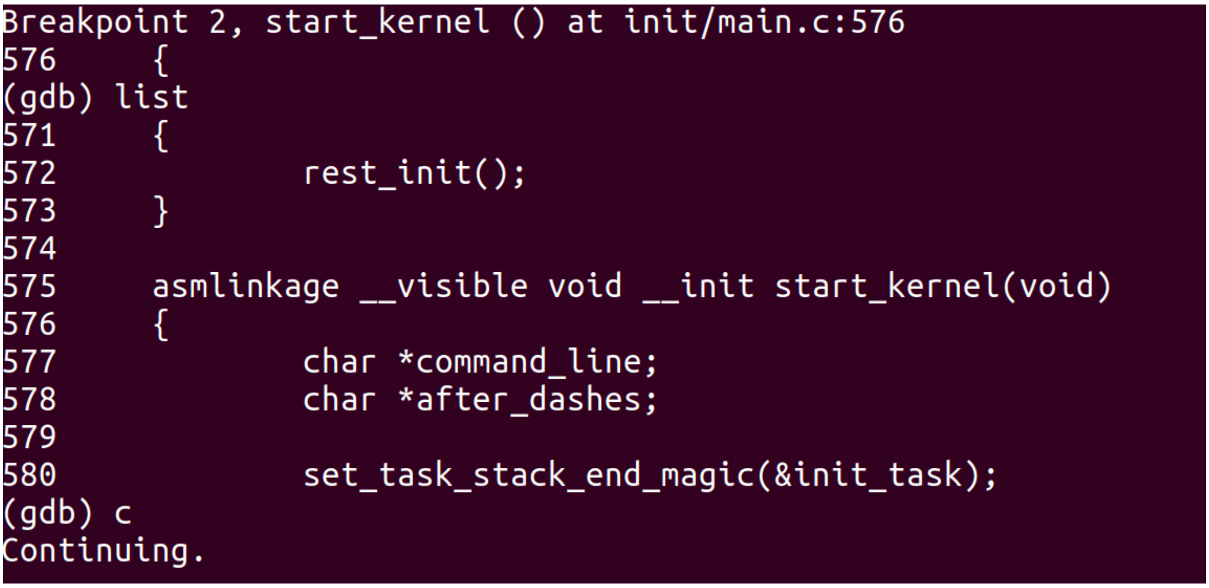
实验一：系统是如何从start_kernel开始一步步进入启动用户空间第一个程序的（提示）：

watch system_state ramdisk_execute_command
SYSTEM_BOOTING



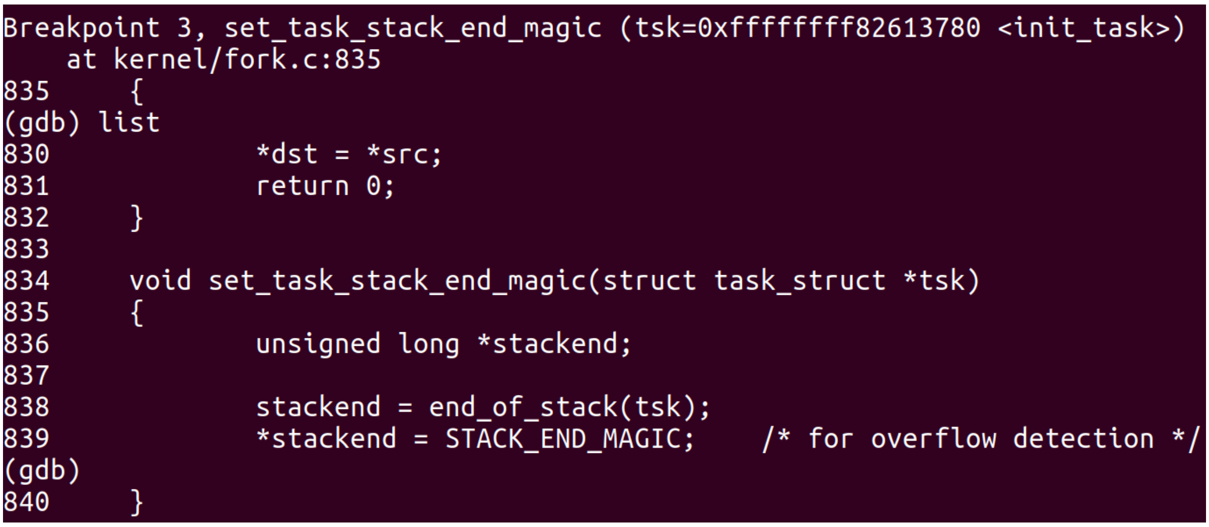
```
(gdb) p system_state
$1 = SYSTEM_BOOTING
```

内核进入c语言阶段，会执行start_kernel 576 初始化内核，首先完成一些核心环境的初始化



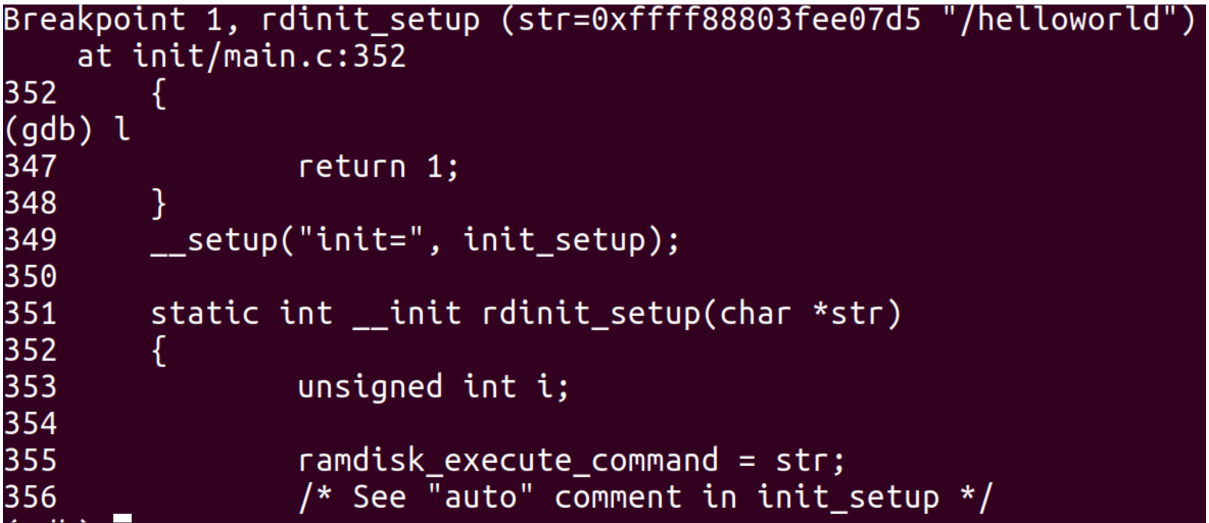
```
Breakpoint 2, start_kernel () at init/main.c:576
576     {
(gdb) list
571     {
572             rest_init();
573     }
574
575     asmlinkage __visible void __init start_kernel(void)
576     {
577             char *command_line;
578             char *after_dashes;
579
580             set_task_stack_end_magic(&init_task);
(gdb) c
Continuing.
```

set_task_stack_end_magic(人工0号进程) kernel/fork.c:786

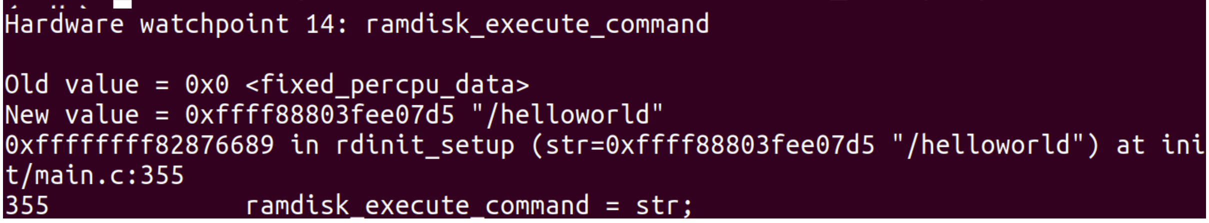


```
Breakpoint 3, set_task_stack_end_magic (tsk=0xffffffff82613780 <init_task>)
    at kernel/fork.c:835
835     {
(gdb) list
830             *dst = *src;
831             return 0;
832     }
833
834     void set_task_stack_end_magic(struct task_struct *tsk)
835     {
836             unsigned long *stackend;
837
838             stackend = end_of_stack(tsk);
839             *stackend = STACK_END_MAGIC;    /* for overflow detection */
(gdb)
840     }
```

rdinit_setup 352 如果bootargs设置了rdinit，那么内核在启动阶段会解析并赋给ramdisk_execute_command

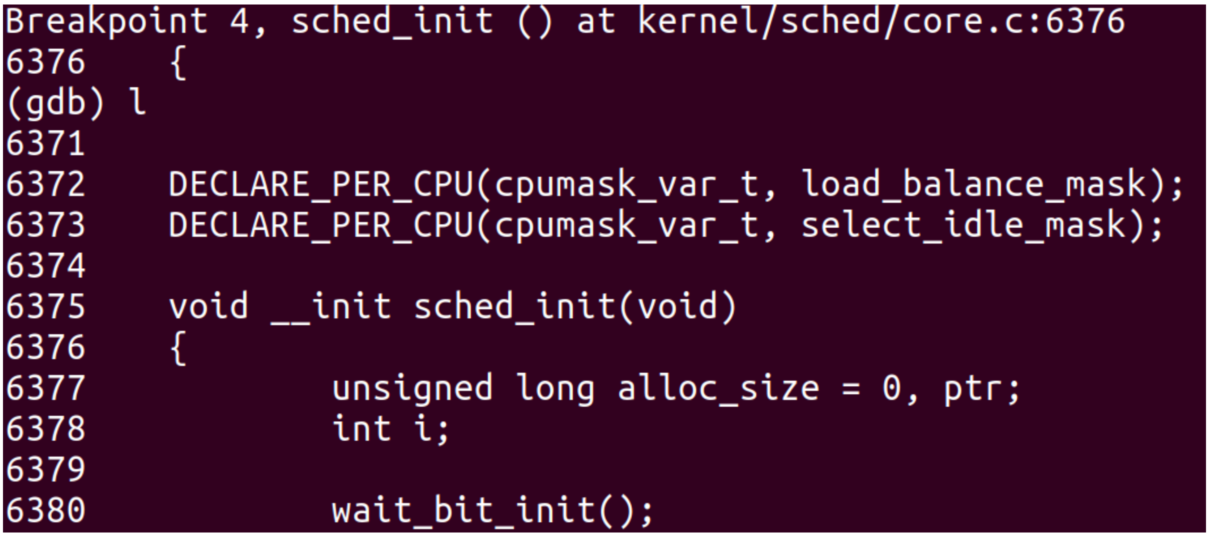


```
Breakpoint 1, rdinit_setup (str=0xffff88803fee07d5 "/helloworld")
    at init/main.c:352
352     {
(gdb) l
347             return 1;
348     }
349     __setup("init=", init_setup);
350
351     static int __init rdinit_setup(char *str)
352     {
353             unsigned int i;
354
355             ramdisk_execute_command = str;
356             /* See "auto" comment in init_setup */
```



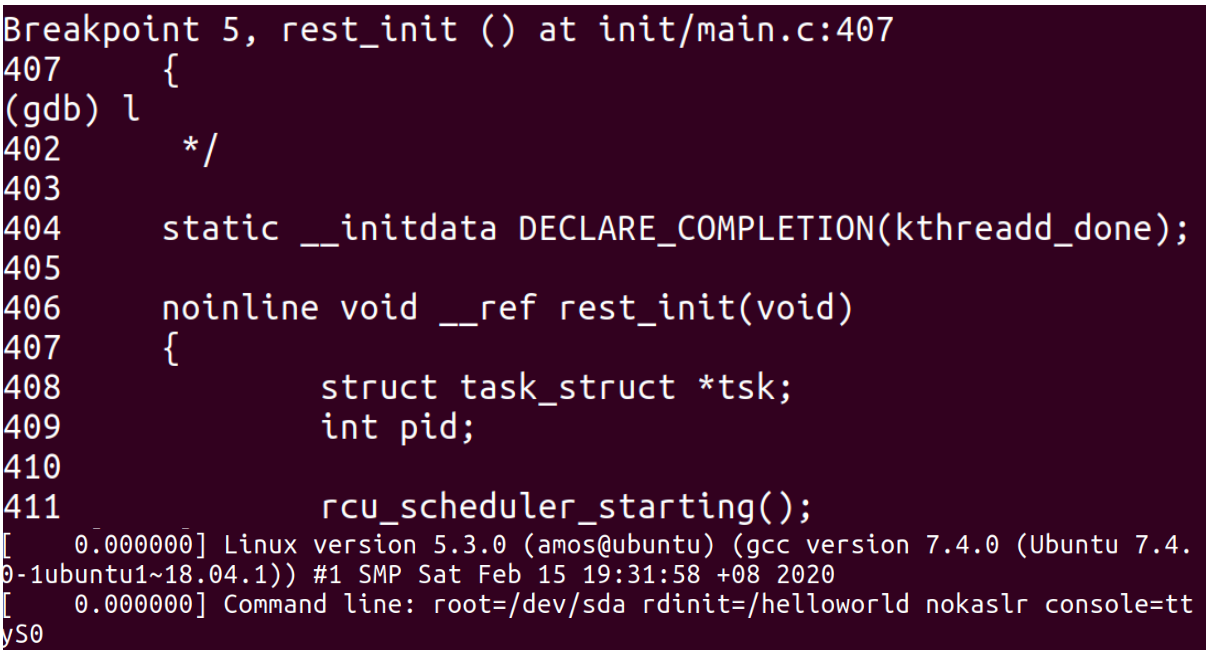
```
Hardware watchpoint 14: ramdisk_execute_command

Old value = 0x0 <fixed_percpu_data>
New value = 0xffff88803fee07d5 "/helloworld"
0xffffffff82876689 in rdinit_setup (str=0xffff88803fee07d5 "/helloworld") at ini
t/main.c:355
355             ramdisk_execute_command = str;
```

sched_init kernel/sched/core.c:6376 进程调度相关初始化



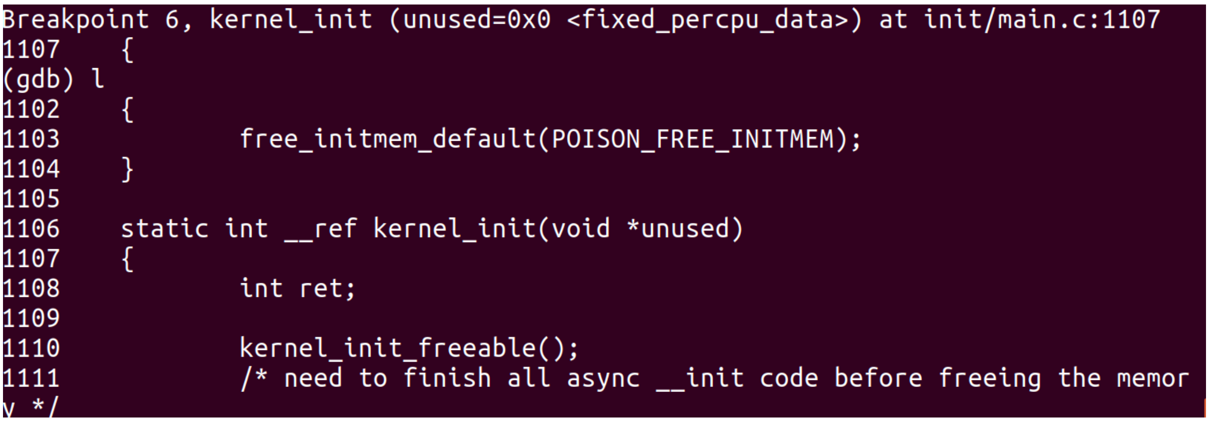
```
Breakpoint 4, sched_init () at kernel/sched/core.c:6376
6376    {
(gdb) l
6371
6372    DECLARE_PER_CPU(cpumask_var_t, load_balance_mask);
6373    DECLARE_PER_CPU(cpumask_var_t, select_idle_mask);
6374
6375    void __init sched_init(void)
6376    {
6377            unsigned long alloc_size = 0, ptr;
6378            int i;
6379
6380            wait_bit_init();
```

rest_init init/main.c:407 完成最后的初始化工作



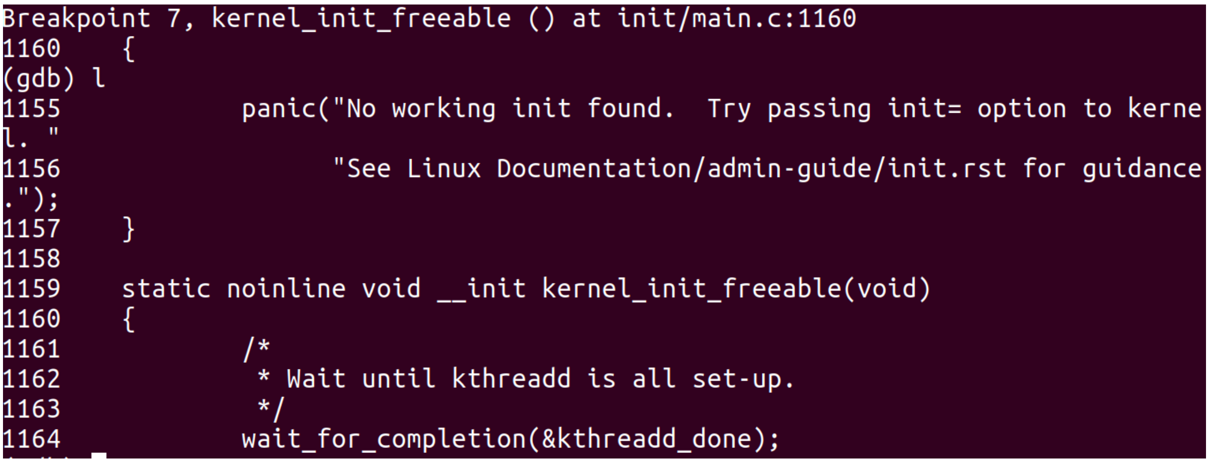
```
Breakpoint 5, rest_init () at init/main.c:407
407     {
(gdb) l
402      */
403
404     static __initdata DECLARE_COMPLETION(kthreadd_done);
405
406     noinline void __ref rest_init(void)
407     {
408             struct task_struct *tsk;
409             int pid;
410
411             rcu_scheduler_starting();
[    0.000000] Linux version 5.3.0 (amos@ubuntu) (gcc version 7.4.0 (Ubuntu 7.4.
0-1ubuntu1~18.04.1)) #1 SMP Sat Feb 15 19:31:58 +08 2020
[    0.000000] Command line: root=/dev/sda rdinit=/helloworld nokaslr console=tt
yS0
```

kernel_init(->1号进程，先创建，但是要等待2号进程已创建完成，化身为用户进程祖先) 1107



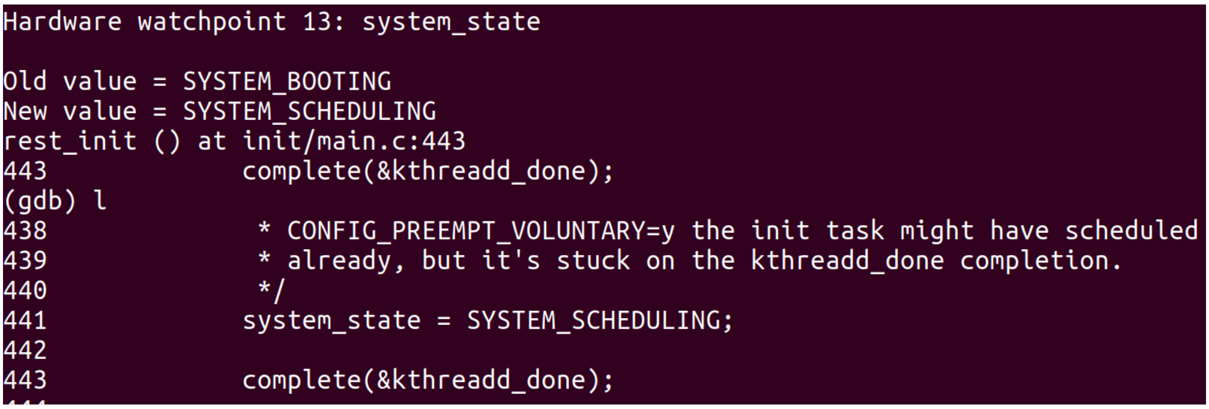
```
Breakpoint 6, kernel_init (unused=0x0 <fixed_percpu_data>) at init/main.c:1107
1107    {
(gdb) l
1102    {
1103            free_initmem_default(POISON_FREE_INITMEM);
1104    }
1105
1106    static int __ref kernel_init(void *unused)
1107    {
1108            int ret;
1109
1110            kernel_init_freeable();
1111            /* need to finish all async __init code before freeing the memor
y */
```

kernel_init_freeable 1160 -> do_basic_setup 1001 -> driver_init 注册内核驱动模块



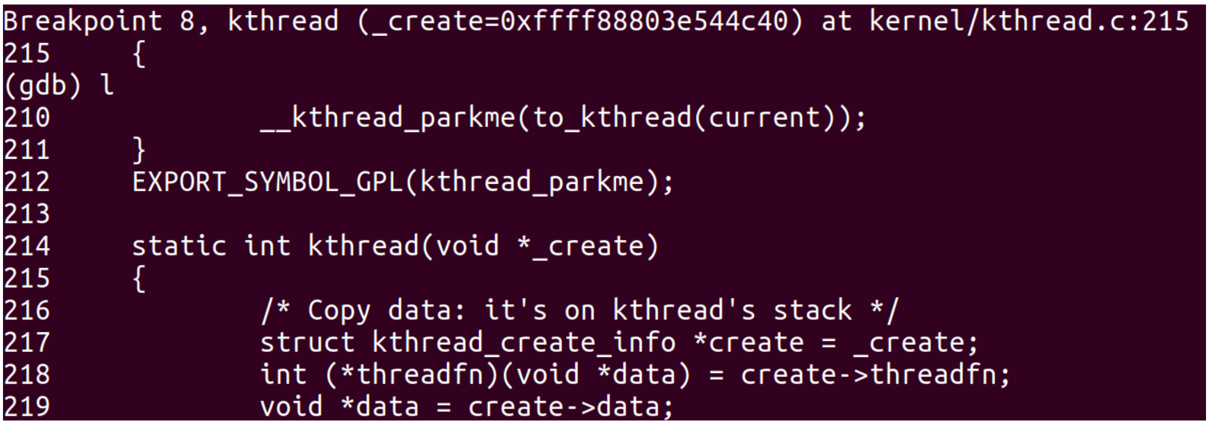
```
Breakpoint 7, kernel_init_freeable () at init/main.c:1160
1160    {
(gdb) l
1155            panic("No working init found.  Try passing init= option to kerne
l. "
1156                  "See Linux Documentation/admin-guide/init.rst for guidance
.");
1157    }
1158
1159    static noinline void __init kernel_init_freeable(void)
1160    {
1161            /*
1162             * Wait until kthreadd is all set-up.
1163             */
1164            wait_for_completion(&kthreadd_done);
```

SYSTEM_SCHEDULING此时系统已经初始化完毕，但是必要的内核线程还没有运行起来，此时0号和1号进程卡在这里等待2号进程以及它的一系列必要子进程创建完成



```
Hardware watchpoint 13: system_state

Old value = SYSTEM_BOOTING
New value = SYSTEM_SCHEDULING
rest_init () at init/main.c:443
443             complete(&kthreadd_done);
(gdb) l
438              * CONFIG_PREEMPT_VOLUNTARY=y the init task might have scheduled
439              * already, but it's stuck on the kthreadd_done completion.
440              */
441             system_state = SYSTEM_SCHEDULING;
442
443             complete(&kthreadd_done);
```

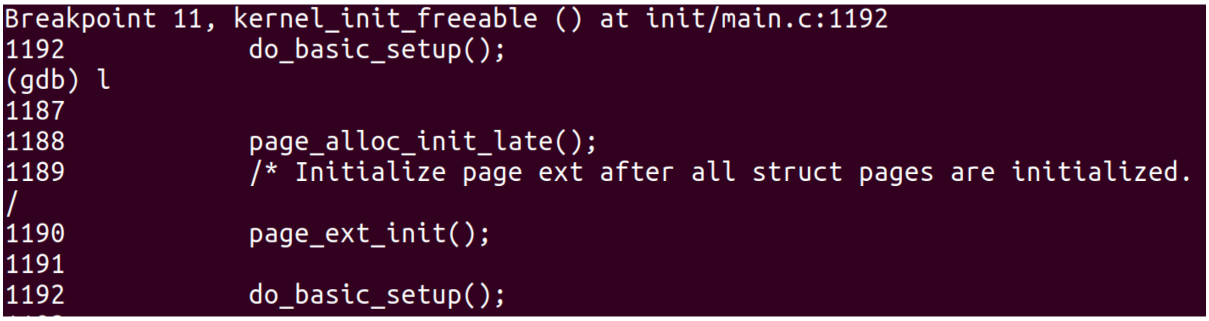
kthreadd(->2号进程，所有内核线程的祖先) kernel/kthread.c:215



```
Breakpoint 8, kthread (_create=0xffff88803e544c40) at kernel/kthread.c:215
215     {
(gdb) l
210             __kthread_parkme(to_kthread(current));
211     }
212     EXPORT_SYMBOL_GPL(kthread_parkme);
213
214     static int kthread(void *_create)
215     {
216             /* Copy data: it's on kthread's stack */
217             struct kthread_create_info *create = _create;
218             int (*threadfn)(void *data) = create->threadfn;
219             void *data = create->data;
```

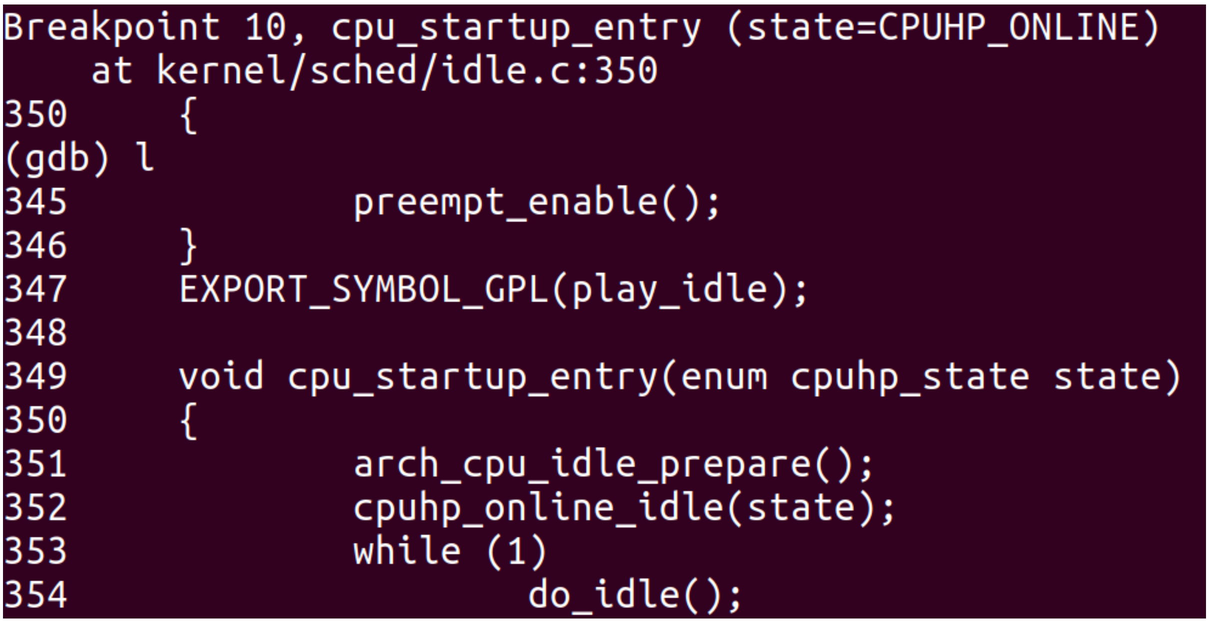
kgdboc调试断在这里。

内核线程初始化之后，1号进程可以进行do_basic_setup如driver_init等 完成设备、驱动等初始化



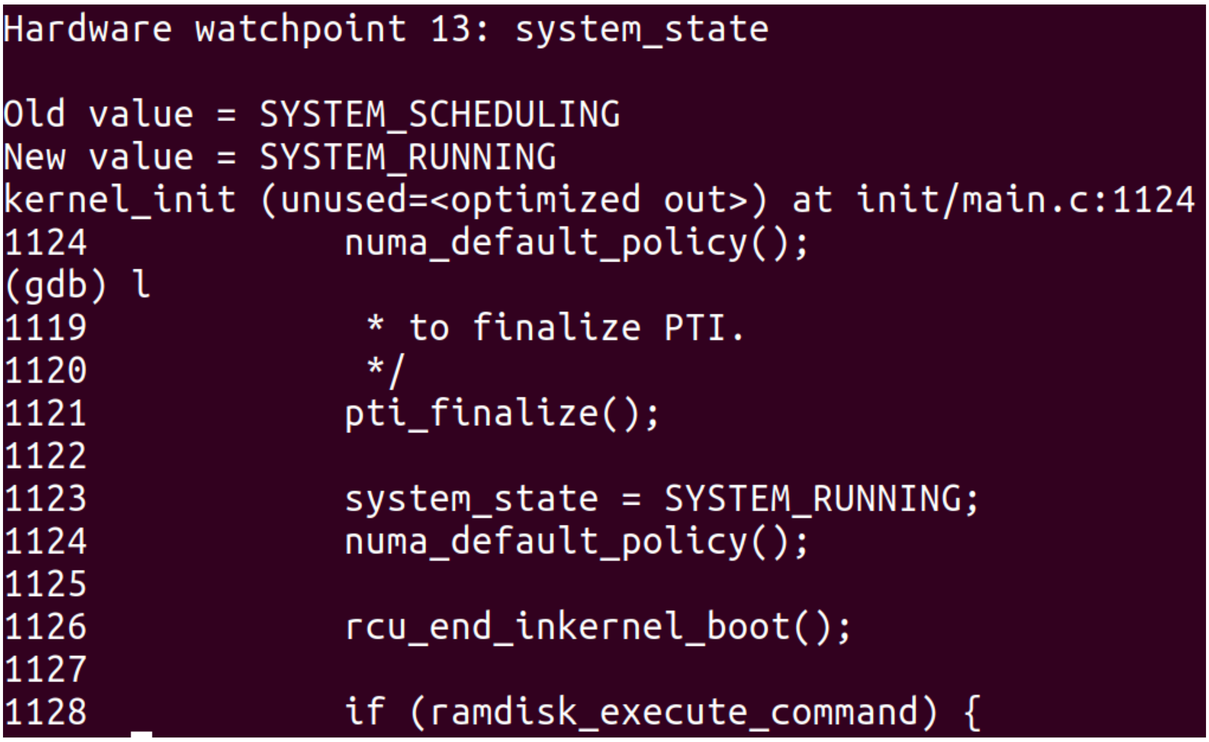
```
Breakpoint 11, kernel_init_freeable () at init/main.c:1192
1192            do_basic_setup();
(gdb) l
1187
1188            page_alloc_init_late();
1189            /* Initialize page ext after all struct pages are initialized.
/
1190            page_ext_init();
1191
1192            do_basic_setup();
```

cpu_startup_entry(0号进程) kernel/sched/idle.c:350 在完成全部的启动后，进入idle循环，化身为idle进程



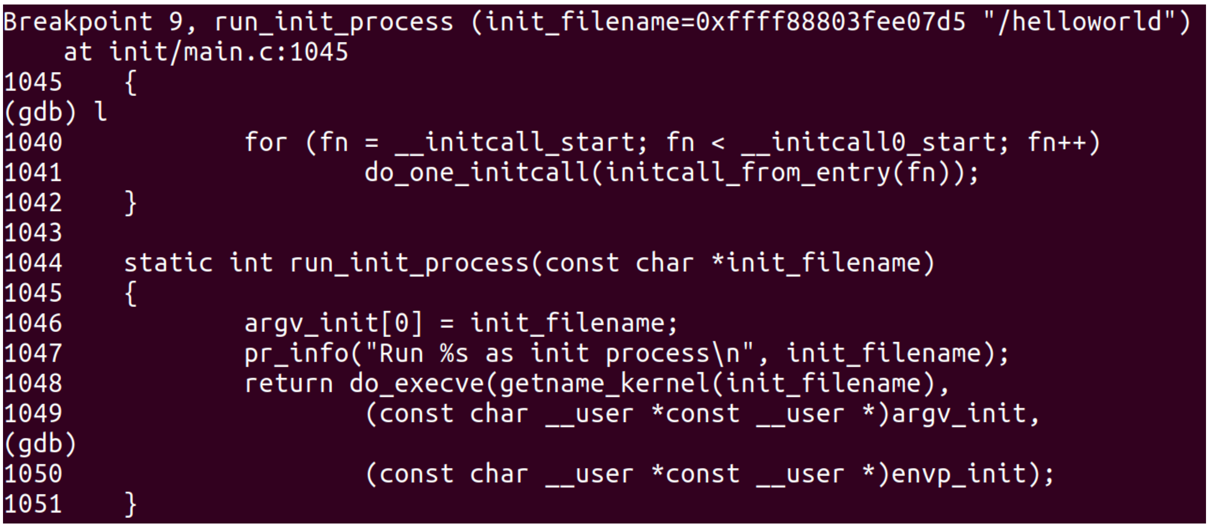
```
Breakpoint 10, cpu_startup_entry (state=CPUHP_ONLINE)
    at kernel/sched/idle.c:350
350     {
(gdb) l
345             preempt_enable();
346     }
347     EXPORT_SYMBOL_GPL(play_idle);
348
349     void cpu_startup_entry(enum cpuhp_state state)
350     {
351             arch_cpu_idle_prepare();
352             cpuhp_online_idle(state);
353             while (1)
354                     do_idle();
```

SYSTEM_RUNNING此时系统才算运行起来



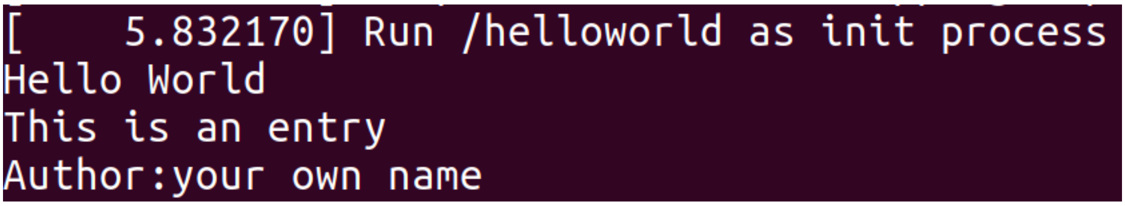
```
Hardware watchpoint 13: system_state

Old value = SYSTEM_SCHEDULING
New value = SYSTEM_RUNNING
kernel_init (unused=<optimized out>) at init/main.c:1124
1124            numa_default_policy();
(gdb) l
1119             * to finalize PTI.
1120             */
1121            pti_finalize();
1122
1123            system_state = SYSTEM_RUNNING;
1124            numa_default_policy();
1125
1126            rcu_end_inkernel_boot();
1127
1128            if (ramdisk_execute_command) {
```

run_init_process 1045 kernel_init作为1号进程化身为用户空间祖先



```
Breakpoint 9, run_init_process (init_filename=0xffff88803fee07d5 "/helloworld")
    at init/main.c:1045
1045    {
(gdb) l
1040            for (fn = __initcall_start; fn < __initcall0_start; fn++)
1041                    do_one_initcall(initcall_from_entry(fn));
1042    }
1043
1044    static int run_init_process(const char *init_filename)
1045    {
1046            argv_init[0] = init_filename;
1047            pr_info("Run %s as init process\n", init_filename);
1048            return do_execve(getname_kernel(init_filename),
1049                    (const char __user *const __user *)argv_init,
(gdb)
1050                    (const char __user *const __user *)envp_init);
1051    }
```

进入系统



```
[    5.832170] Run /helloworld as init process
Hello World
This is an entry
Author:your own name
```

实验二（提示）：在qemu里调试模块

1.将模块添加入到制作的busybox.img中的/lib目录下

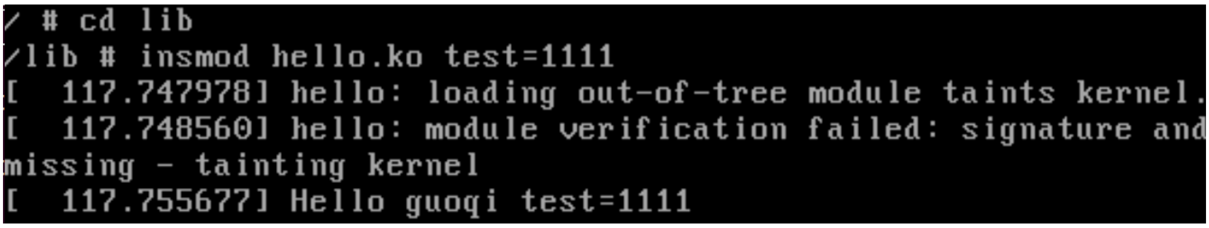
```
>> sudo mount -o loop ~/kDebug/busybox.img /mnt/disk
>> sudo cp ~/kDebug/hello.ko /mnt/disk/lib
>> sudo umount /mnt/disk
```

2.sudo qemu-system-x86_64 -kernel bzImage -initrd initrd.img-5.3.0 -append "root=/dev/sda nokaslr" -boot c -hda busybox.img -k en-us -m 1024 -serial tcp::4321,server

不加kgdbwait使能参数

开发机gdb连接

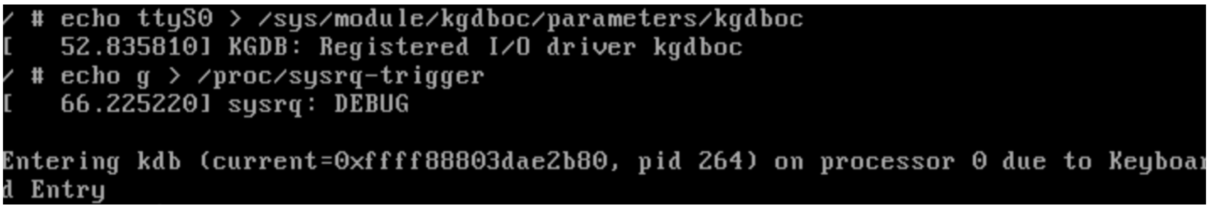
3.启动qemu之后insmod hello



```
/ # cd lib
/lib # insmod hello.ko test=1111
[  117.747978] hello: loading out-of-tree module taints kernel.
[  117.748560] hello: module verification failed: signature and
missing - tainting kernel
[  117.755677] Hello guoqi test=1111
```

4.使能kgdb，echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc 注册kgdb

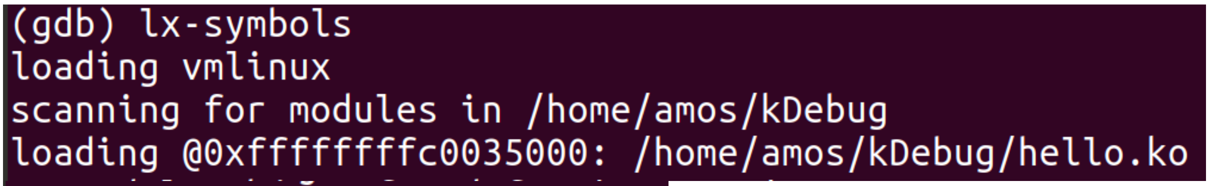
然后触发断点echo g > /proc/sysrq-trigger



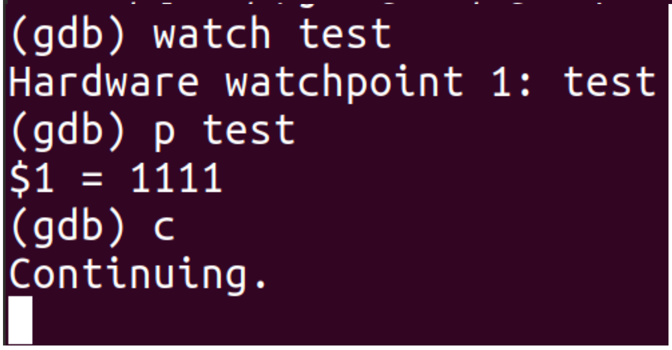
```
/ # echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
[   52.835810] KGDB: Registered I/O driver kgdboc
/ # echo g > /proc/sysrq-trigger
[   66.225220] sysrq: DEBUG

Entering kdb (current=0xffff88803dae2b80, pid 264) on processor 0 due to Keyboar
d Entry
```

这时开发机gdb再连接，会成功断在kgdb_breakpoint()这里

5.开发机lx-symbols把hello.ko模块中的符号加入进来，这时可设置hello里的函数断点、watch变量，并print 变量



```
(gdb) lx-symbols
loading vmlinux
scanning for modules in /home/amos/kDebug
loading @0xffffffffc0035000: /home/amos/kDebug/hello.ko
```



```
(gdb) watch test
Hardware watchpoint 1: test
(gdb) p test
$1 = 1111
(gdb) c
Continuing.
```

6.按c把主动权回到调试机

reference:
https://01.org/linuxgraphics/gfx-docs/drm/dev-tools/gdb-kernel-debugging.html