Linux Kernel
Process Management Experiment

518030910031 陳韋廷

1. Modification

Linux Kernel version: 5.10.31

a. <include/linux/sched.h>: line 651

```
...
volatile long       state;

/* Added by Nickchen Nick */
int ctx; // ctx will be initialized as 0 and increases per call.

/*
 * This begins the randomizable portion of task_struct. Only
 * scheduling-critical items should be added above here.
 */
randomized_struct_fields_start
...
```

b. <kernel/fork.c>: pid_t kernel_clone()

```
...
p = copy_process(NULL, trace, NUMA_NO_NODE, args);
add_latent_entropy();

if (IS_ERR(p))
  return PTR_ERR(p);

/* Added by Nickchen Nick */
p->ctx = 0; // initialize ctx here

/*
 * Do this prior waking up the new thread - the thread pointer
 * might get invalid after that point, if the thread exits quickly.
 */
trace_sched_process_fork(current, p);
...
```

c. <kernel/sched/core.c>: void activate_task()

```
void activate_task(struct rq *rq, struct task_struct *p, int flags)
{
  enqueue_task(rq, p, flags);

  p->on_rq = TASK_ON_RQ_QUEUED;

  /* Added by Nickchen Nick */
  p->ctx = p->ctx + 1; // increases ctx by 1 everytime it is activated
}
```

d. <fs/proc/base.c>

i. read function

```
static int my_ctx_read(struct seq_file *m, void *v) {
  struct inode *inode = m->private;
  struct task_struct *p;

  p = get_proc_task(inode);
  if (!p) return -ESRCH;

  task_lock(p);
  seq_printf(m, "%u\n", p->ctx);
  task_unlock(p);

  return 0;
}
```

ii. open function

```
static int my_ctx_open(struct inode *inode, struct file *flip) {
  return single_open(flip, my_ctx_read, inode);
}
```

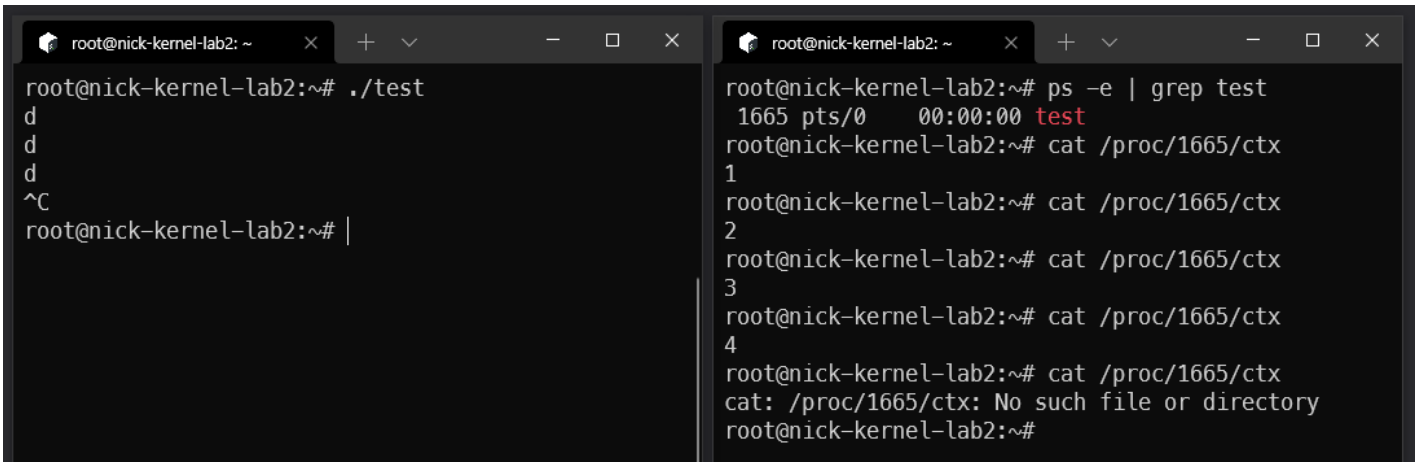
iii. file_operations

```
static const struct file_operations my_ctx_ops = {
  .open = my_ctx_open,
  .read = seq_read,
  .llseek = seq_lseek,
  .release = single_release,
}
```

iv. add a new entry in tgid_base_stuff[]

```
...
#ifdef CONFIG_PROC_PID_ARCH_STATUS
  ONE("arch_status", S_IRUGO, proc_pid_arch_status),
#endif
  /* Added by Nickchen Nick */
  REG("ctx", S_IRUSR, my_ctx_ops),
};
...
```

2. Result



3. Final Thoughts

Compiling the kernel is scary (?) because it takes a lot of time and anything might break during compilation. Also if tmux is used, one won't be able to see the errors after the compilation stops because there are some steps that will occur after the failure of one step. One technique I use is to redirect the output to a log file, i.e., something like make -j 2 > kernel_compile.log. When the compilation stops for whatever reason, I can simply cat kernel_compile.log | tail --lines=20 to see what happened during the last moment. But anyway it's a fun experiment!