請將sys_write追蹤完畢，說明從剛開始到「於terminal中印出字串」為止的函數。

這次的作業追蹤sys_write可以追到很深，過程中很多不確定的地方，經過努力上網爬文和自己不斷的嘗試之後，才追到BUILDIO。

sys_write

```
635 SYSCALL_DEFINE3(write, unsigned int, fd, const char __u
636         size_t, count)
637 {
638     struct fd f = fdget_pos(fd);
639     ssize_t ret = -EBADF;
640
641     if (f.file) {
642         loff_t pos = file_pos_read(f.file);
643         ret = vfs_write(f.file, buf, count, &pos);
644         if (ret >= 0)
645             file_pos_write(f.file, pos);
646         fdput_pos(f);
647     }
648
649     return ret;
650 }
```

對sys_write設立中斷點後，會停在635行，查看function裡的函數發現vfs_write 很可疑，所以對vfs_write 設立中斷點，並且往下追蹤。

vfs_write

```
576 ssize_t vfs_write(struct file *file, const char __use
577 {
578     ssize_t ret;
579
580     if (!(file->f_mode & FMODE_WRITE))
581         return -EBADF;
582     if (!(file->f_mode & FMODE_CAN_WRITE))
583         return -EINVAL;
584     if (unlikely(!access_ok(VERIFY_READ, buf, count)))
585         return -EFAULT;
586
587     ret = rw_verify_area(WRITE, file, pos, count);
588     if (ret >= 0) {
589         count = ret;
590         file_start_write(file);
591         if (file->f_op->write)
592             ret = file->f_op->write(file, buf, count,
593         else if (file->f_op->aio_write)
594             ret = do_sync_write(file, buf, count, pos
595         else
```

進入vfs_write，第591行的判斷需要依照file內定義的檔案來決定所呼叫的函數，因為所以對第592行下中斷點，進入redirected_tty_write。

sys_write和vfs_write是屬於tty核心(TTY_core)，對整個tty設備的抽象，對用戶提供統一的接口。

redirected_tty_write

```
1252 ssize_t redirected_tty_write(struct file *file, cons
1253                           size_t count, loff_t *ppos)
1254 {
1255     struct file *p = NULL;
1256
1257     spin_lock(&redirect_lock);
1258     if (redirect)
1259         p = get_file(redirect);
1260     spin_unlock(&redirect_lock);
1261
1262     if (p) {
1263         ssize_t res;
1264         res = vfs_write(p, buf, count, &p->f_pos);
1265         fput(p);
1266         return res;
1267     }
1268     return tty_write(file, buf, count, ppos);
1269 }
```

爬文發現file->f_op->write函數指的是redirected_tty_write。
在redirected_tty_write裡面調用了tty_write，對tty_write進行近一步的追蹤。

tty_write

```
1227 static ssize_t tty_write(struct file *file, const cha
1228                           size_t count, loff_t *ppos)
1229 {
1230     struct tty_struct *tty = file_tty(file);
1231     struct tty_ldisc *ld;
1232     ssize_t ret;
1233
1234     if (tty_paranoia_check(tty, file_inode(file), "tt
1235         return -EIO;
1236     if (!tty || !tty->ops->write ||
1237         (test_bit(TTY_IO_ERROR, &tty->flags)))
1238             return -EIO;
1239     /* Short term debug to catch buggy drivers */
1240     if (tty->ops->write_room == NULL)
1241         printk(KERN_ERR "tty driver %s lacks a write_
1242             tty->driver->name);
1243     ld = tty_ldisc_ref_wait(tty);
1244     if (!ld->ops->write)
1245         ret = -EIO;
1246     else
1247         ret = do_tty_write(ld->ops->write, tty, file,
```

進入tty_write，裡面調用了do_tty_write，對do_tty_write設定斷點，往下追蹤。

do_tty_write

```
1100 static inline ssize_t do_tty_write(
1101     ssize_t (*write)(struct tty_struct *, struct fi
1102     struct tty_struct *tty,
1103     struct file *file,
1104     const char __user *buf,
1105     size_t count)
1106 {
1107     ssize_t ret, written = 0;
1108     unsigned int chunk;
1109 
1110     ret = tty_write_lock(tty, file->f_flags & O_NDE
1111     if (ret < 0)
1112         return ret;
1113 
1114     /*
```

對write 函數設立中斷點，進入n_tty_write

redirected_tty_write、tty_write、do_tty_write 主要對傳輸的資料做格式化。

n_tty_write

```
2337 static ssize_t n_tty_write(struct tty_struct *tty, st
2338                 const unsigned char *buf, size_t nr)
2339 {
2340     const unsigned char *b = buf;
2341     DEFINE_WAIT_FUNC(wait, woken_wake_function);
2342     int c;
2343     ssize_t retval = 0;
2344 
2345     /* Job control check -- must be done at start (PO
2346     if (L_TOSTOP(tty) && file->f_op->write != redirec
2347         retval = tty_check_change(tty);
2348         if (retval)
2349             return retval;
2350     }
2351 
2352     down_read(&tty->termios_rwsem);
2353 
2354     /* Write out any echoed characters that are still
```

進入n_tty_write後，發現process_out_block很可疑，於是往下追蹤process_out_block。

process_output_block

```
547 static ssize_t process_output_block(struct tty_struct *
548                     const unsigned char *buf, unsigned
549 {
550     struct n_tty_data *ldata = tty->disc_data;
551     int space;
552     int i;
553     const unsigned char *cp;
554 
555     mutex_lock(&ldata->output_lock);
556 
557     space = tty_write_room(tty);
558     if (!space) {
559         mutex_unlock(&ldata->output_lock);
560         return 0;
561     }
562     if (nr > space)
563         nr = space;
```

進入process_out_block查看，對uart_write設立斷點。

uart_write

```
508 static int uart_write(struct tty_struct *tty,
509                        const unsigned char *buf, int co
510 {
511     struct uart_state *state = tty->driver_data;
512     struct uart_port *port;
513     struct circ_buf *circ;
514     unsigned long flags;
515     int c, ret = 0;
516 
517     /*
518      * This means you called this function _after_ t
519      * closed.  No cookie for you.
520      */
521     if (!state) {
522         WARN_ON(1);
523         return -EL3HLT;
524     }
```

查看uart_write，發現__uart_start很可疑，所以對__uart_start設立斷點，進行追蹤。

__uart_start

```
93  static void __uart_start(struct tty_struct *tty)
94  {
95      struct uart_state *state = tty->driver_data;
96      struct uart_port *port = state->uart_port;
97
98      if (!uart_tx_stopped(port))
99          port->ops->start_tx(port);
100 }
101
102 static void uart_start(struct tty_struct *tty)
103 {
104     struct uart_state *state = tty->driver_data;
105     struct uart_port *port = state->uart_port;
106     unsigned long flags;
107
108     spin_lock_irqsave(&port->lock, flags);
109     __uart_start(tty);
110     spin_unlock_irqrestore(&port->lock, flags);
111 }
```

進入__uart_start，對 port->ops->start_tx 追蹤

serial8250_start_tx

```
1374 static void serial8250_start_tx(struct uart_port *port)
1375 {
1376     struct uart_8250_port *up = up_to_u8250p(port);
1377
1378     serial8250_rpm_get_tx(up);
1379
1380     if (up->dma && !up->dma->tx_dma(up))
1381         return;
1382
1383     if (!(up->ier & UART_IER_THRI)) {
1384         up->ier |= UART_IER_THRI;
1385         serial_port_out(port, UART_IER, up->ier);
1386
1387         if (up->bugs & UART_BUG_TXEN) {
1388             unsigned char lsr;
1389             lsr = serial_in(up, UART_LSR);
1390             up->lsr_saved_flags |= lsr & LSR_SAVE_FLAGS
1391             if (lsr & UART_LSR_THRE)
1392                 serial8250_tx_chars(up);
```

port->ops->指的是serial8250所以進入 serial8250_start_tx。追蹤 serial_port_out。

uart_write、__uart_start、serial8250_start_tx是 tty 的硬體驅動，針對硬體取操作。

serial_port_out

```
255 static inline void serial_port_out(struct
256 {
257     up->serial_out(up, offset, value);
258 }
```

追蹤 io_serial_out

io_serial_out

```
448 static void io_serial_out(struct uart_
449 {
450     offset = offset << p->regshift;
451     outb(value, p->iobase + offset);
452 }
```

對outb進行追蹤

BUILDIO

```
313 BUILDIO(b, b, char)
314 BUILDIO(w, w, short)
315 BUILDIO(l, , int)
```

追不下去惹