Binary Exploit - III

u1f383 🎃

About

- u1f383 🎃
- Pwner
- Zoolab
- NCtfU / Xx TSJ xX / Goburin’

Outline

- FILE
  - Introduction
  - Background
  - Arbitrary read
  - Arbitrary write
  - Arbitrary execute

Introduction

$ Introduction

Process overhead

- CPU 執行 system call 後就會切換 register 以及 page table，這會造成很大的成本
- 為了執行速度，因此有些常用且無安全疑慮的 system call 可以直接在 userspace 的 vDSO 執行，像是 gettimeofday

```
[ DISASM ]
  0x555555555070 <gettimeofday@plt>     endbr64
  0x555555555074 <gettimeofday@plt+4>   bnd jmp qword ptr [rip + 0x2f55]     <gettimeofday>
   ↓
  0x7ffff7fc1bd0 <gettimeofday>         jmp    0x7ffff7fc1a20               <0x7ffff7fc1a20>
   ↓
► 0x7ffff7fc1a20                        push   rbp
  0x7ffff7fc1a21                        mov    r10, rdi
  0x7ffff7fc1a24                        mov    r11, rsi
  0x7ffff7fc1a27                        mov    rbp, rsp
  0x7ffff7fc1a2a                        push   r12
  0x7ffff7fc1a2c                        lea    r12, [rip - 0x49b3]
  0x7ffff7fc1a33                        push   rbx
  0x7ffff7fc1a34                        test   rdi, rdi
[ STACK ]
```

```
pwndbg> xinfo 0x7ffff7fc1a20
Extended information for virtual address 0x7ffff7fc1a20:

  Containing mapping:
    0x7ffff7fc1000     0x7ffff7fc3000 r-xp     2000 0      [vdso]

  Offset information:
        Mapped Area 0x7ffff7fc1a20 = 0x7ffff7fc1000 + 0xa20
```

$ Introduction

Process overhead

```
pwndbg> info register
rax            0x401745            4200261
rbx            0x7ffd8eb94ef8      140726997962488
rcx            0x1                 1
rdx            0x7ffd8eb94ef8      140726997962488
rsi            0x7ffd8eb94ee8      140726997962472
rdi            0x1                 1
rbp            0x7ffd8eb94d00      0x7ffd8eb94d00
rsp            0x7ffd8eb94d00      0x7ffd8eb94d00
r8             0x4c7d70            5012848
r9             0x4                 4
r10            0x80                128
r11            0x206               518
r12            0x1                 1
r13            0x7ffd8eb94ee8      140726997962472
r14            0x4c17d0            4986832
r15            0x1                 1
rip            0x40174d            0x40174d
eflags         0x246               [ IOPL=0 IF ZF PF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
fs_base        0x16853c0           23614400
gs_base        0x0                 0
k_gs_base      0xffffffff81a4b000  -2119913472
cr0            0x80050033          [ PG AM WP NE ET MP PE ]
cr2            0x4b4b20            4934432
cr3            0x1000fb000         [ PDBR=1048827 PCID=0 ]
cr4            0x3006b0            [ SMAP SMEP OSXMMEXCPT OSFXSR PGE PAE PSE ]
cr8            0x1                 1
efer           0xd01               [ NXE LMA LME SCE ]
```

```
► 0x40174d    syscall  <SYS_<unk_4200261>>
```

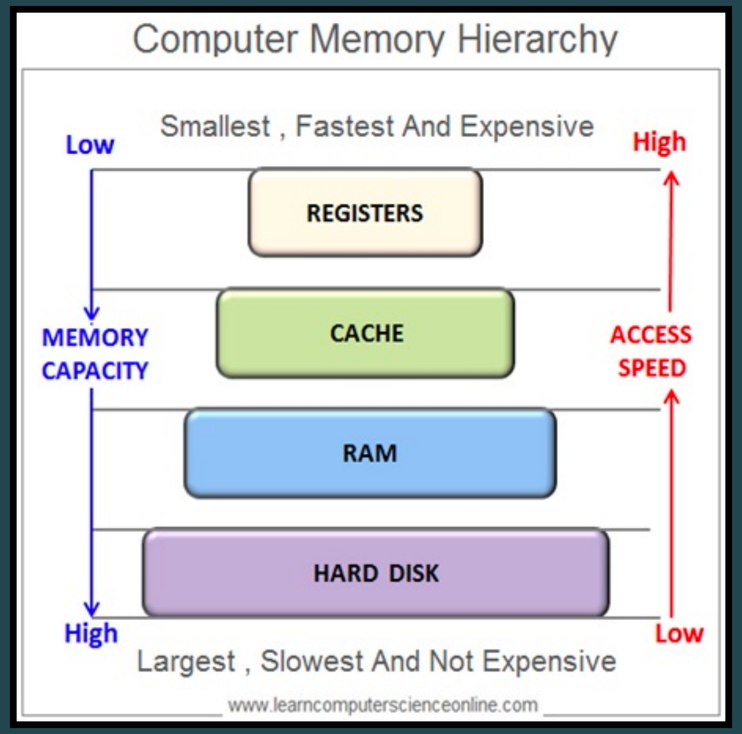
system call

```
pwndbg> info reg
rax            0x401745            4200261
rbx            0x7ffd8eb94ef8      140726997962488
rcx            0x40174f            4200271
rdx            0x7ffd8eb94ef8      140726997962488
rsi            0x7ffd8eb94ee8      140726997962472
rdi            0x1                 1
rbp            0x7ffd8eb94d00      0x7ffd8eb94d00
rsp            0x7ffd8eb94d00      0x7ffd8eb94d00
r8             0x4c7d70            5012848
r9             0x4                 4
r10            0x80                128
r11            0x246               582
r12            0x1                 1
r13            0x7ffd8eb94ee8      140726997962472
r14            0x4c17d0            4986832
r15            0x1                 1
rip            0xffffffff81400000  0xffffffff81400000 <entry_SYSCALL_64>
eflags         0x2                 [ IOPL=0 ]
cs             0x10                16
ss             0x18                24
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
fs_base        0x16853c0           23614400
gs_base        0x0                 0
k_gs_base      0xffffffff81a4b000  -2119913472
cr0            0x80050033          [ PG AM WP NE ET MP PE ]
cr2            0x4b4b20            4934432
cr3            0x1000fb000         [ PDBR=1048827 PCID=0 ]
cr4            0x3006b0            [ SMAP SMEP OSXMMEXCPT OSFXSR PGE PAE PSE ]
cr8            0x1                 1
efer           0xd01               [ NXE LMA LME SCE ]
```

$ Introduction

Process overhead

- 做 IO 操作時會需要與硬體裝置互動，最常見就是讀檔寫檔時會存取 disk
- CPU 對於外部元件的存取速度大概是： Register > RAM >>> disk / device



REF: https://www.learncomputerscienceonline.com/what-are-cpu-registers/

$ Introduction

Process overhead

- 讀寫檔案會需要呼叫 system call (read, write, …)，同時又是 IO 操作，因此會造成不小的 overhead
- 於是 glibc 實作 FILE 結構來包裝對檔案的存取，內部優化了檔案的操作，降低 system call 呼叫的次數

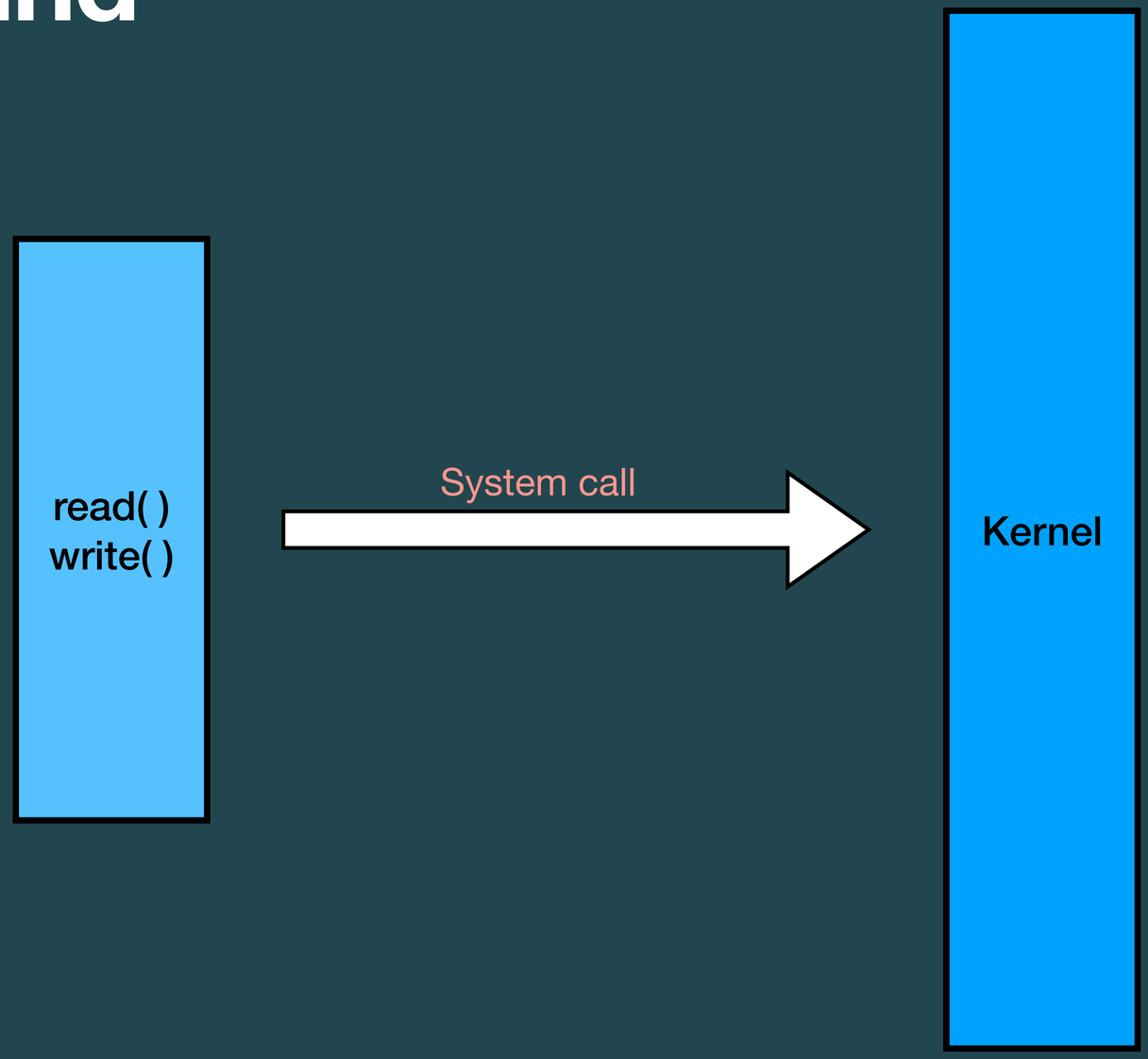
```c
int main()
{
    char buf[0x10];

    int fd = open("/tmp/meow", O_RDONLY);
    read(fd, buf, 0x10);
    close(fd);

    return 0;
}
```

```c
int main()
{
    char buf[0x10];

    FILE *fp = fopen("/tmp/meow", "r");
    fread(buf, 0x10, 1, fp);
    fclose(fp);

    return 0;
}
```

Background

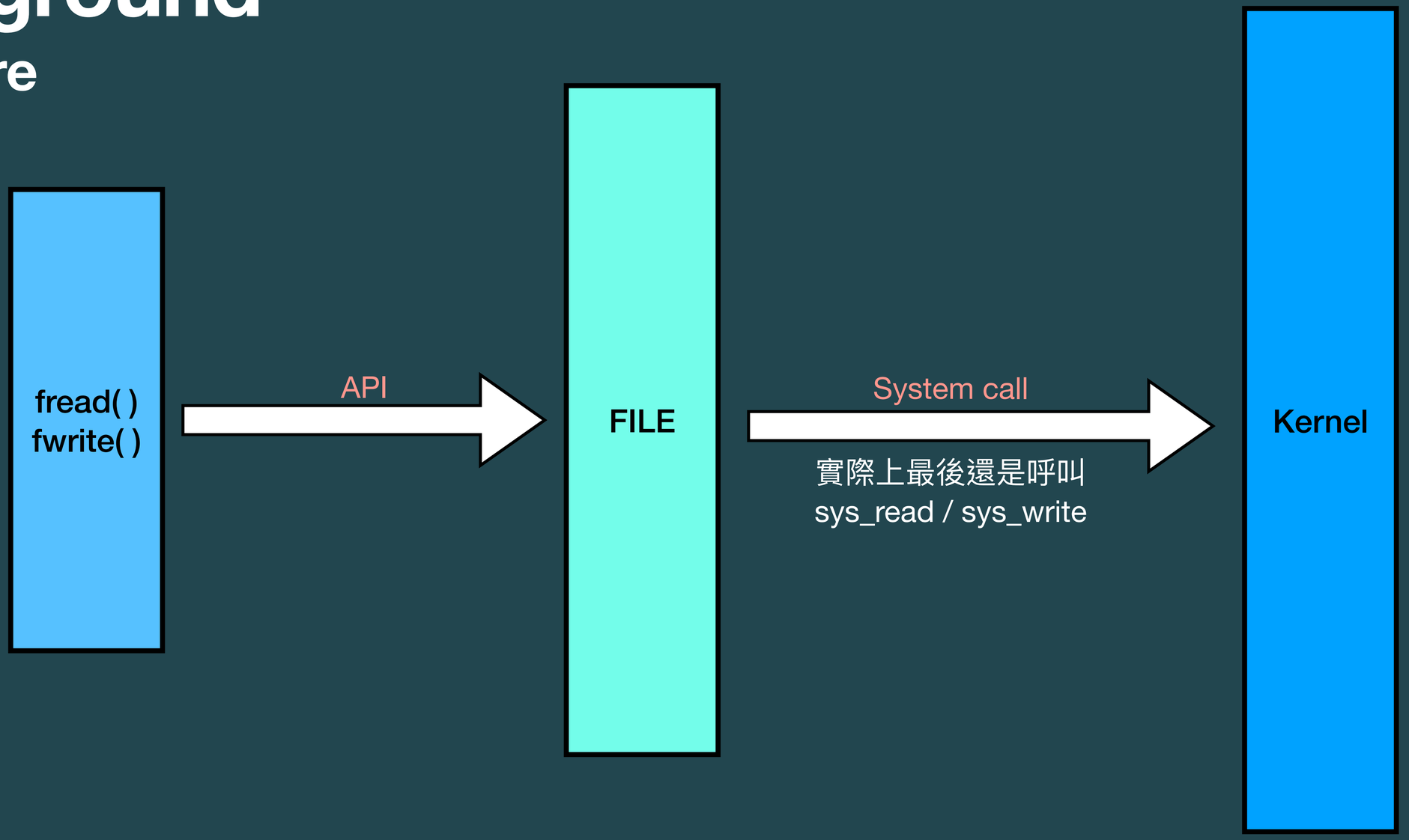
$ Background

Architecture



$ Background

Architecture



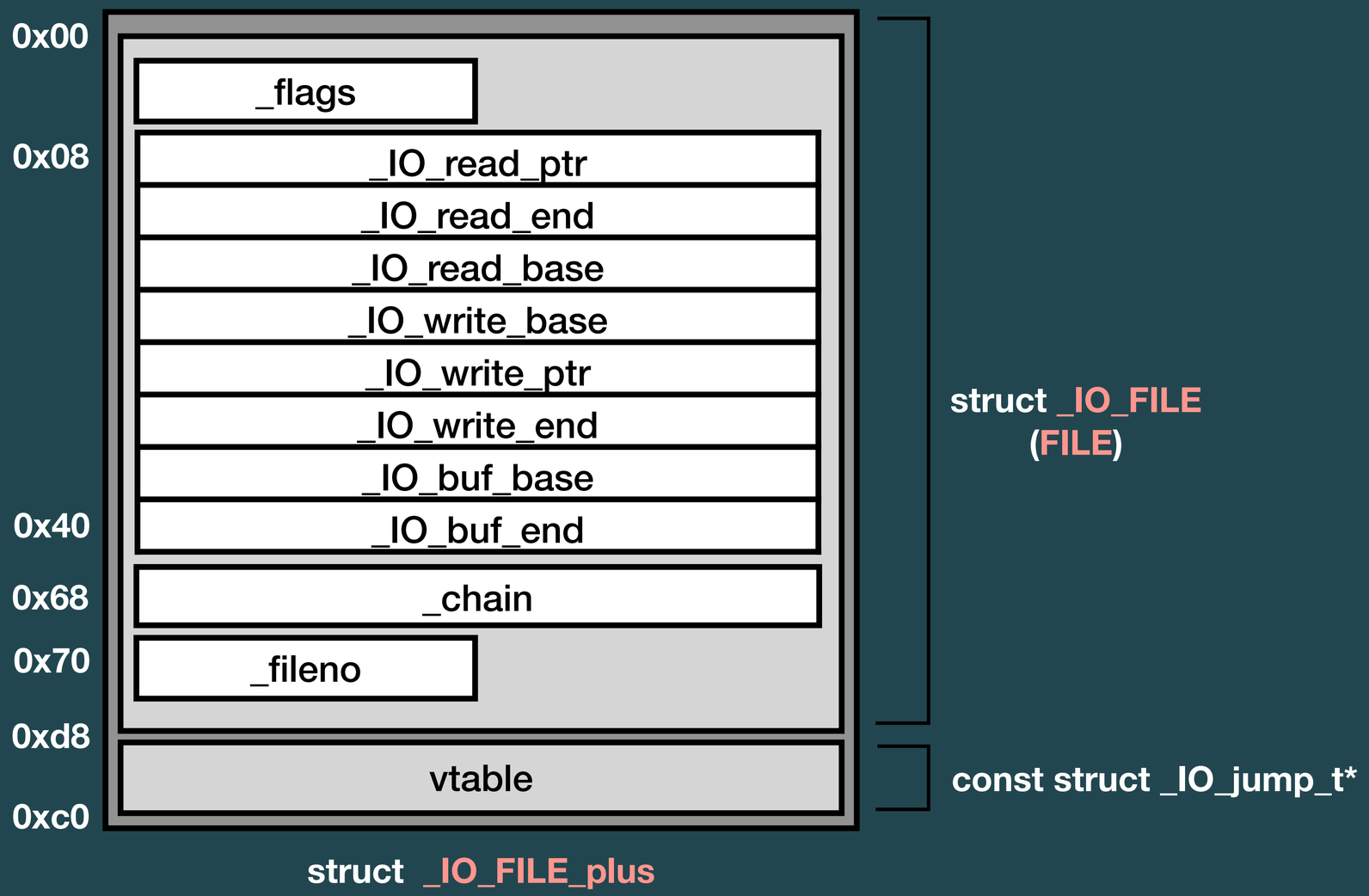
$ Background

Architecture

- Glibc FILE 具體做了哪些優化，包含但不限於：
  - Data buffer
  - Lock
  - Security check
- 在 userspace 層可以實作的操作，就盡量不透過 kernel 來做
- 在不影響使用者體驗的情況下，可以想辦法減少 system call 的次數

$ Background

Architecture



$ Background

Architecture

struct _IO_FILE_plus

決定 mode 是 “r” 或 “w”

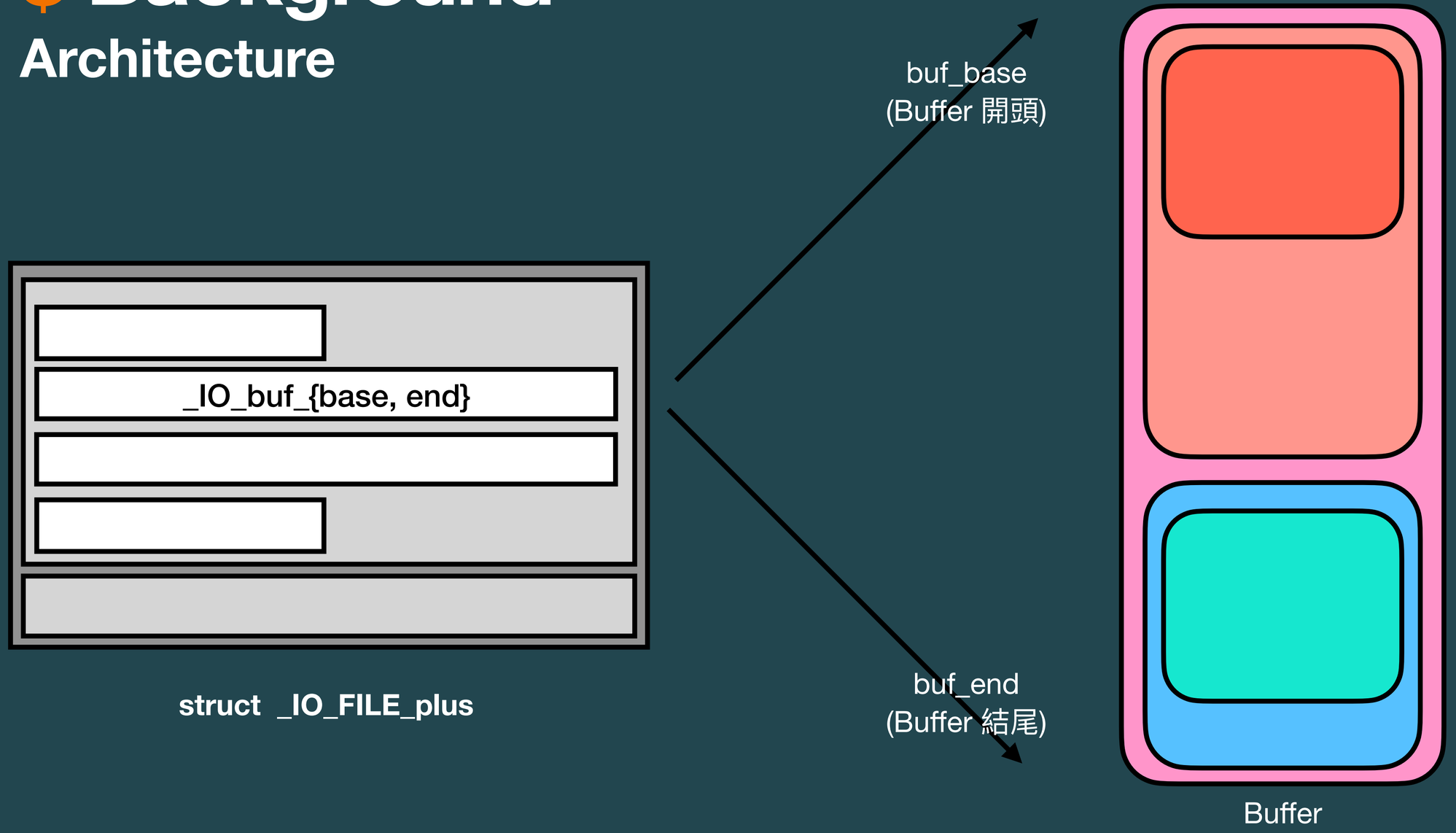
代表正在寫資料 (write)

Magic number

```
#define _IO_MAGIC          0xFBAD0000
#define _IO_MAGIC_MASK     0xFFFF0000
#define _IO_USER_BUF           0x0001
#define _IO_UNBUFFERED         0x0002
#define _IO_NO_READS           0x0004
#define _IO_NO_WRITES          0x0008
#define _IO_EOF_SEEN           0x0010
#define _IO_ERR_SEEN           0x0020
#define _IO_DELETE_DONT_CLOSE  0x0040
#define _IO_LINKED             0x0080
#define _IO_IN_BACKUP          0x0100
#define _IO_LINE_BUF           0x0200
#define _IO_TIED_PUT_GET       0x0400
#define _IO_CURRENTLY_PUTTING  0x0800
#define _IO_IS_APPENDING       0x1000
#define _IO_IS_FILEBUF         0x2000
                               /* 0x4000
#define _IO_USER_LOCK          0x8000
```

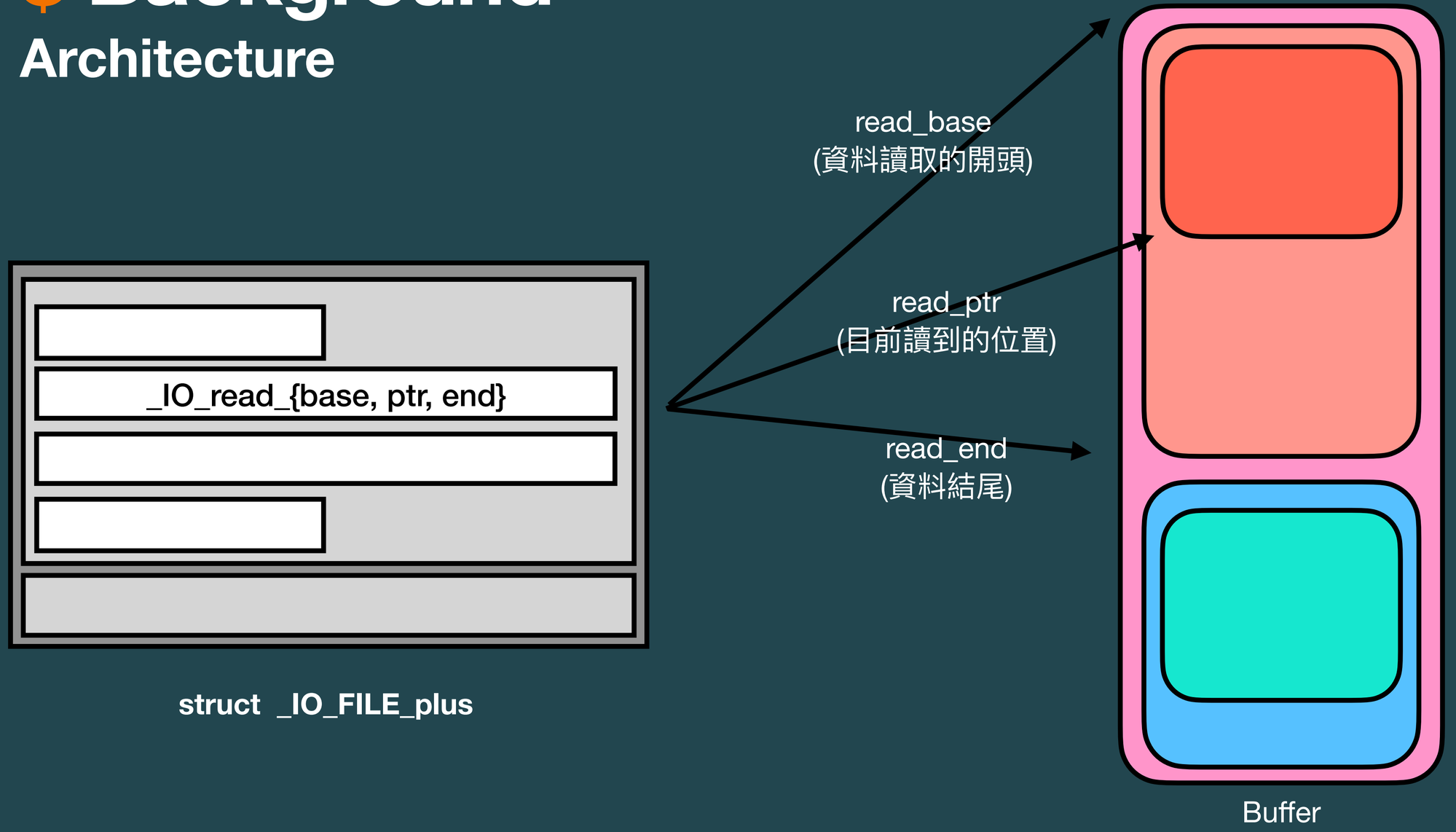
$ Background

Architecture



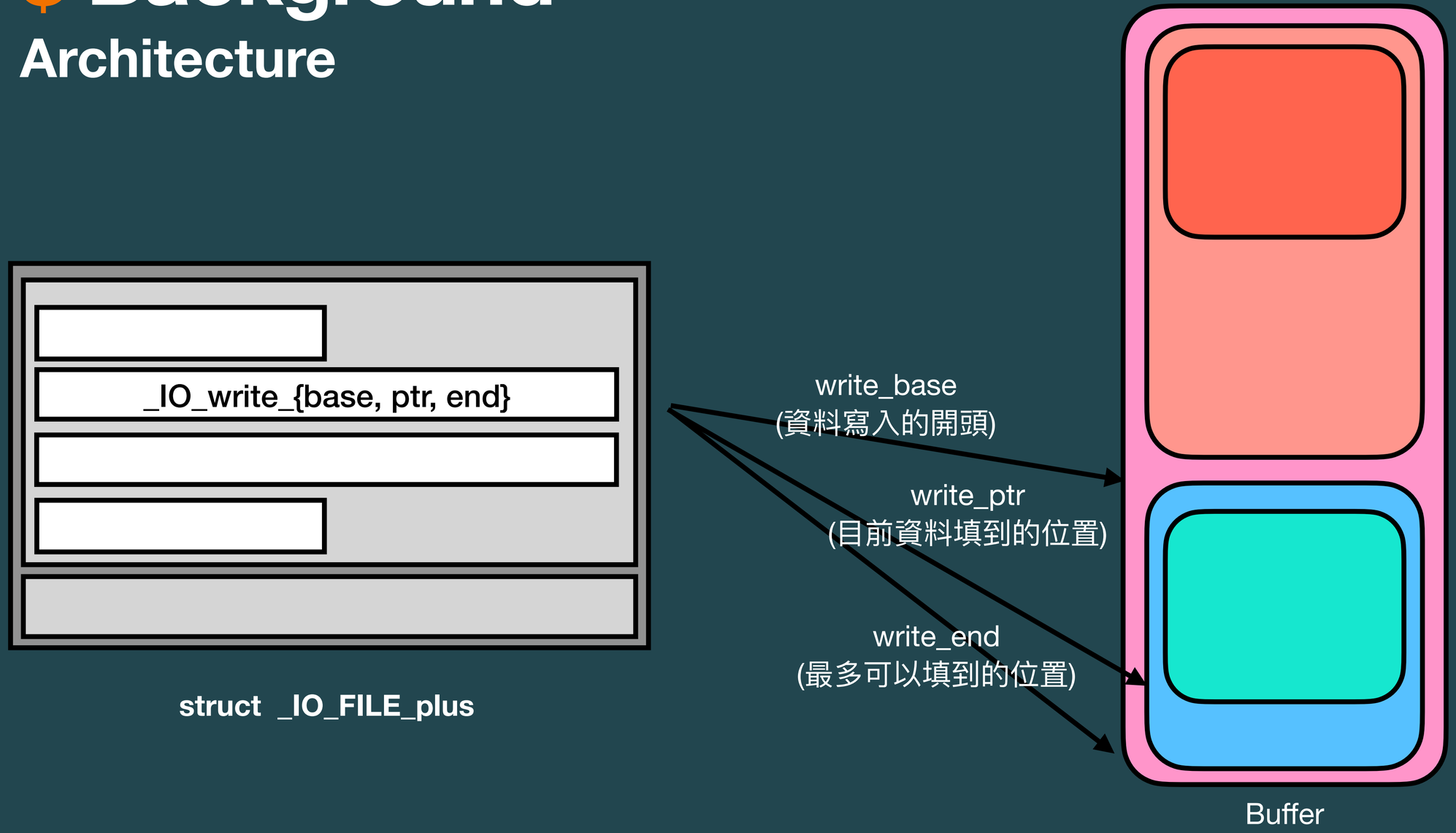
$ Background

Architecture



$ Background

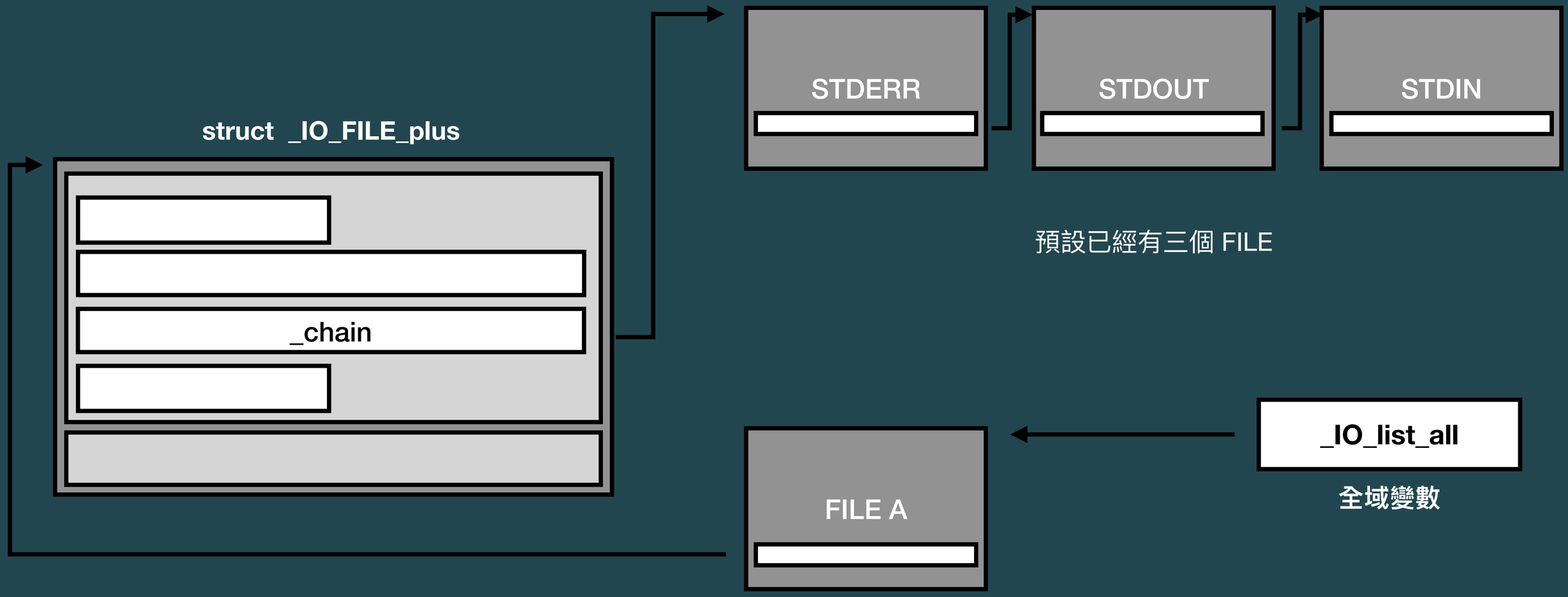
Architecture



struct _IO_FILE_plus

$ Background

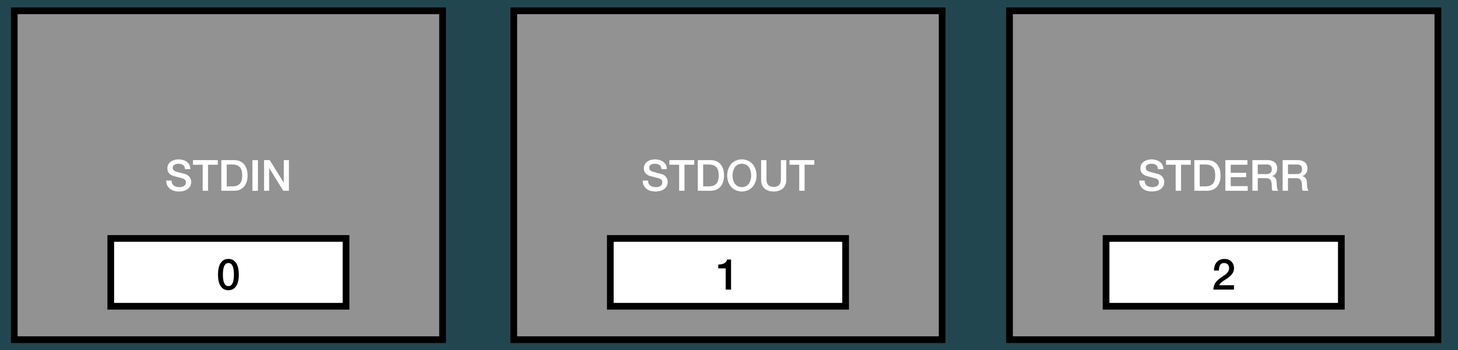
Architecture



$ Background

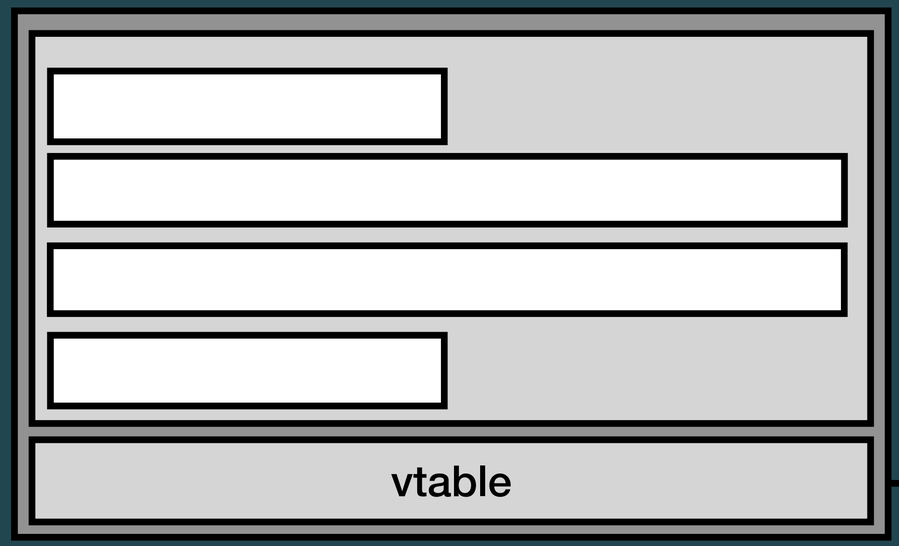
Architecture

struct _IO_FILE_plus



$ Background

Architecture



struct _IO_FILE_plus

預設指向
_IO_file_jumps

```
gef➤  p _IO_file_jumps
$1 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x7ffff7e84ff0 <_IO_new_file_finish>,
  __overflow = 0x7ffff7e85a00 <_IO_new_file_overflow>,
  __underflow = 0x7ffff7e856b0 <_IO_new_file_underflow>,
  __uflow = 0x7ffff7e869c0 <__GI__IO_default_uflow>,
  __pbackfail = 0x7ffff7e87d40 <__GI__IO_default_pbackfail>,
  __xsputn = 0x7ffff7e84be0 <_IO_new_file_xsputn>,
  __xsgetn = 0x7ffff7e847a0 <__GI__IO_file_xsgetn>,
  __seekoff = 0x7ffff7e84010 <_IO_new_file_seekoff>,
  __seekpos = 0x7ffff7e86d60 <_IO_default_seekpos>,
  __setbuf = 0x7ffff7e838f0 <_IO_new_file_setbuf>,
  __sync = 0x7ffff7e83780 <_IO_new_file_sync>,
  __doallocate = 0x7ffff7e783b0 <__GI__IO_file_doallocate>,
  __read = 0x7ffff7e84bb0 <__GI__IO_file_read>,
  __write = 0x7ffff7e845f0 <_IO_new_file_write>,
  __seek = 0x7ffff7e83d70 <__GI__IO_file_seek>,
  __close = 0x7ffff7e838e0 <__GI__IO_file_close>,
  __stat = 0x7ffff7e845d0 <__GI__IO_file_stat>,
  __showmanyc = 0x7ffff7e87ed0 <_IO_default_showmanyc>,
  __imbue = 0x7ffff7e87ee0 <_IO_default_imbue>
}
```

$ Background

Architecture

- Buffer 一共分成 full buffer、line buffer 以及 no buffer，介紹會以 full buffer 為主
- 分析以下列常用的 function 來了解 FILE 的使用：
  - fopen( ) - 開啟檔案並且初始化 FILE 結構
  - fread( ) - 從 file descriptor 讀取資料
  - fwrite( ) - 寫入資料到 file descriptor
  - fclose( ) - 關閉檔案並且釋放 FILE 結構

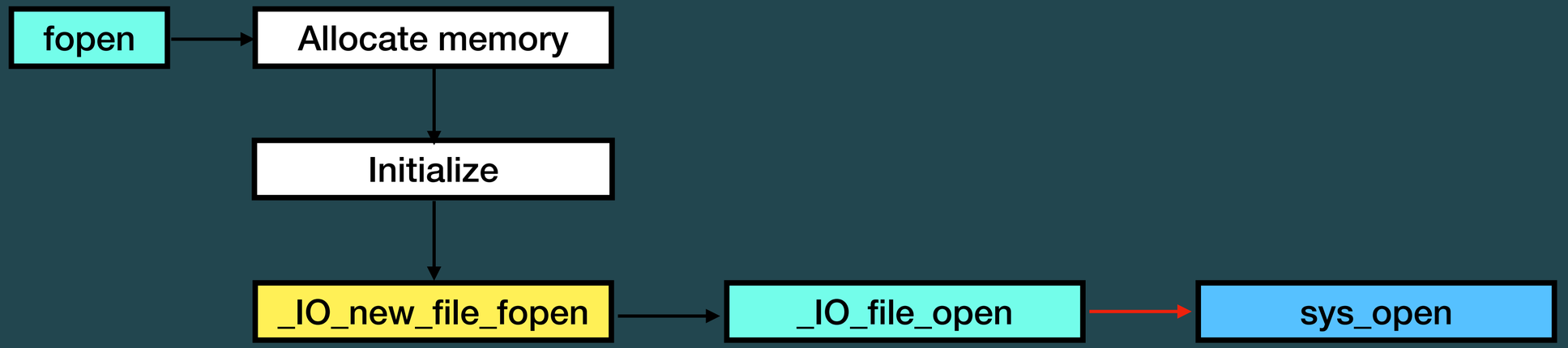
$ Background

Fopen

- Buffer 一共分成 full buffer、line buffer 以及 no buffer，介紹會以 full buffer 為主
- 分析以下列常用的 function 來了解 FILE 的使用：
  - fopen( ) - 開啟檔案並且初始化 FILE 結構
  - fread( ) - 從 file descriptor 讀取資料
  - fwrite( ) - 寫入資料到 file descriptor
  - fclose( ) - 關閉檔案並且釋放 FILE 結構

$ Background

Fopen



$ Background

Fopen

- fopen —> _IO_new_fopen —> __fopen_internal

```
FILE *
__fopen_internal(const char *filename, const char *mode, int is32)
{
  struct locked_FILE
  {
    struct _IO_FILE_plus fp;
    struct _IO_wide_data wd;
  } *new_f = (struct locked_FILE *)malloc(sizeof(struct locked_FILE));

  if (new_f == NULL)
    return NULL;
  _IO_no_init(&new_f->fp.file, 0, 0, &new_f->wd, &_IO_wfile_jumps);
  _IO_JUMPS(&new_f->fp) = &_IO_file_jumps;
  _IO_new_file_init_internal(&new_f->fp);
  if (_IO_file_fopen((FILE *)new_f, filename, mode, is32) != NULL)
    return __fopen_maybe_mmap(&new_f->fp.file);

  _IO_un_link(&new_f->fp);
  free(new_f);
  return NULL;
}
```

1. 分配記憶體

2. 初始化結構成員

3. 開啟檔案取得 fd

$ Background

Fopen

```
$4 = {
  fp = {
    file = {
      _flags = 0x0,
      _IO_read_ptr = 0x0,
      _IO_read_end = 0x0,
      _IO_read_base = 0x0,
      _IO_write_base = 0x0,
      _IO_write_ptr = 0x0,
      _IO_write_end = 0x0,
      _IO_buf_base = 0x0,
      _IO_buf_end = 0x0,
      _IO_save_base = 0x0,
      _IO_backup_base = 0x0,
      _IO_save_end = 0x0,
      _markers = 0x0,
      _chain = 0x0,
      _fileno = 0x0,
      _flags2 = 0x0,
      _old_offset = 0x0,
      _cur_column = 0x0,
      _vtable_offset = 0x0,
      _shortbuf = "",
      _lock = 0x0,
      _offset = 0x0,
      _codecvt = 0x0,
      _wide_data = 0x0,
      _freeres_list = 0x0,
      _freeres_buf = 0x0,
      __pad5 = 0x0,
      _mode = 0x0,
      _unused2 = '\000' <repeats 19 times>
    },
    vtable = 0x0
```

Uninitialized

_IO_MAGIC (0xfbad0000) - 一定要設
_IO_LINKED (0x80) - 有在 linked list 當中
_IO_NO_READS (0x4) - 呼叫時傳入 “w” 模式

```
$8 = {
  fp = {
    file = {
      _flags = 0xfbad2484,
      _IO_read_ptr = 0x0,
      _IO_read_end = 0x0,
      _IO_read_base = 0x0,
      _IO_write_base = 0x0,
      _IO_write_ptr = 0x0,
      _IO_write_end = 0x0,
      _IO_buf_base = 0x0,
      _IO_buf_end = 0x0,
      _IO_save_base = 0x0,
      _IO_backup_base = 0x0,
      _IO_save_end = 0x0,
      _markers = 0x0,
      _chain = 0x7ffff7fc15c0 <_IO_2_1_stderr_>,
      _fileno = 0x3,
      _flags2 = 0x0,
      _old_offset = 0x0,
      _cur_column = 0x0,
      _vtable_offset = 0x0,
      _shortbuf = "",
      _lock = 0x555555559380,
      _offset = 0xffffffffffffffff,
      _codecvt = 0x0,
      _wide_data = 0x555555559390,
      _freeres_list = 0x0,
      _freeres_buf = 0x0,
      __pad5 = 0x0,
      _mode = 0x0,
      _unused2 = '\000' <repeats 19 times>
    },
    vtable = 0x7ffff7fc24a0 <__GI__IO_file_jumps>
```

指向下個 FILE

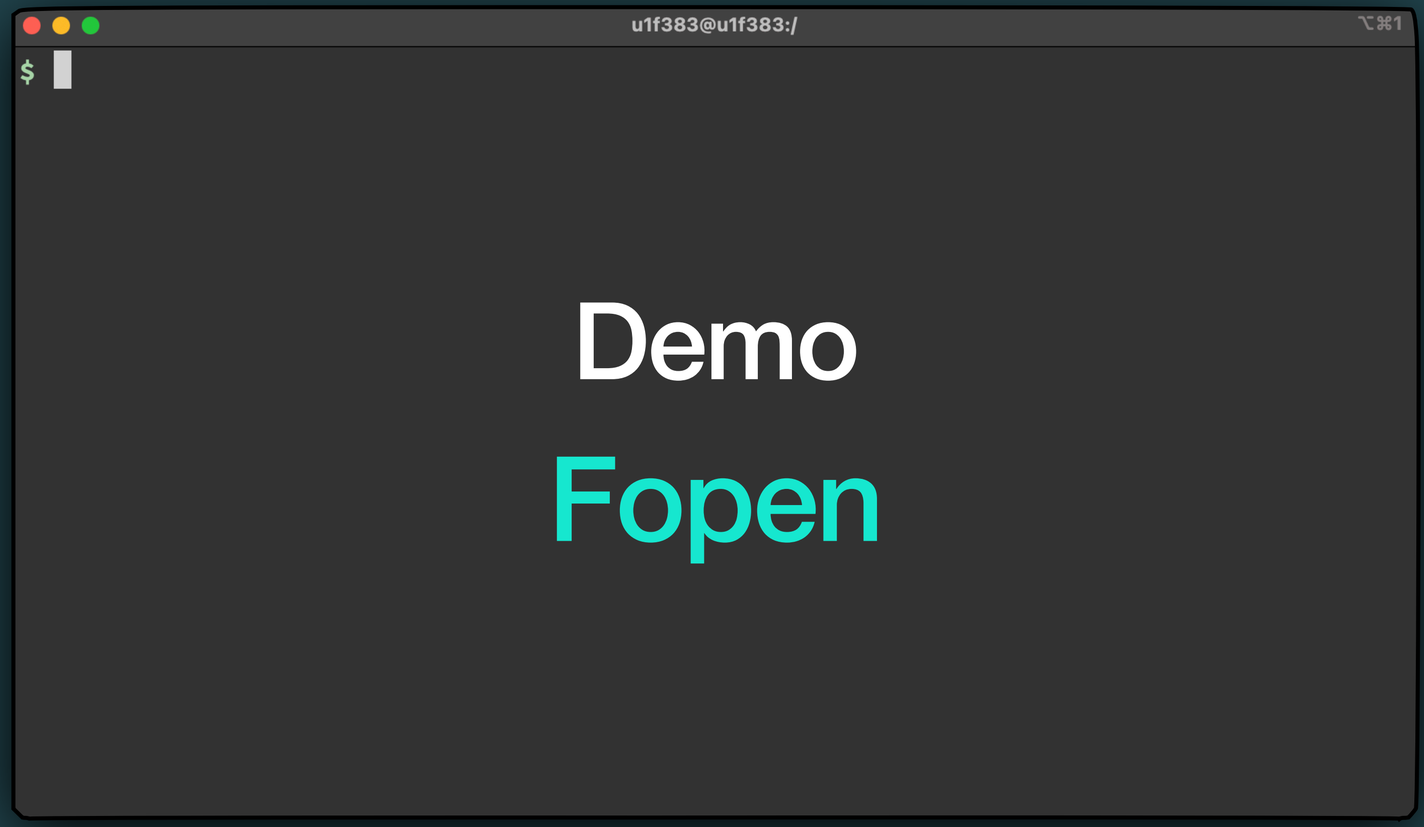
sys_open 回傳的 fd

指向 _IO_file_jumps function table

Initialized

$ Background

Fopen



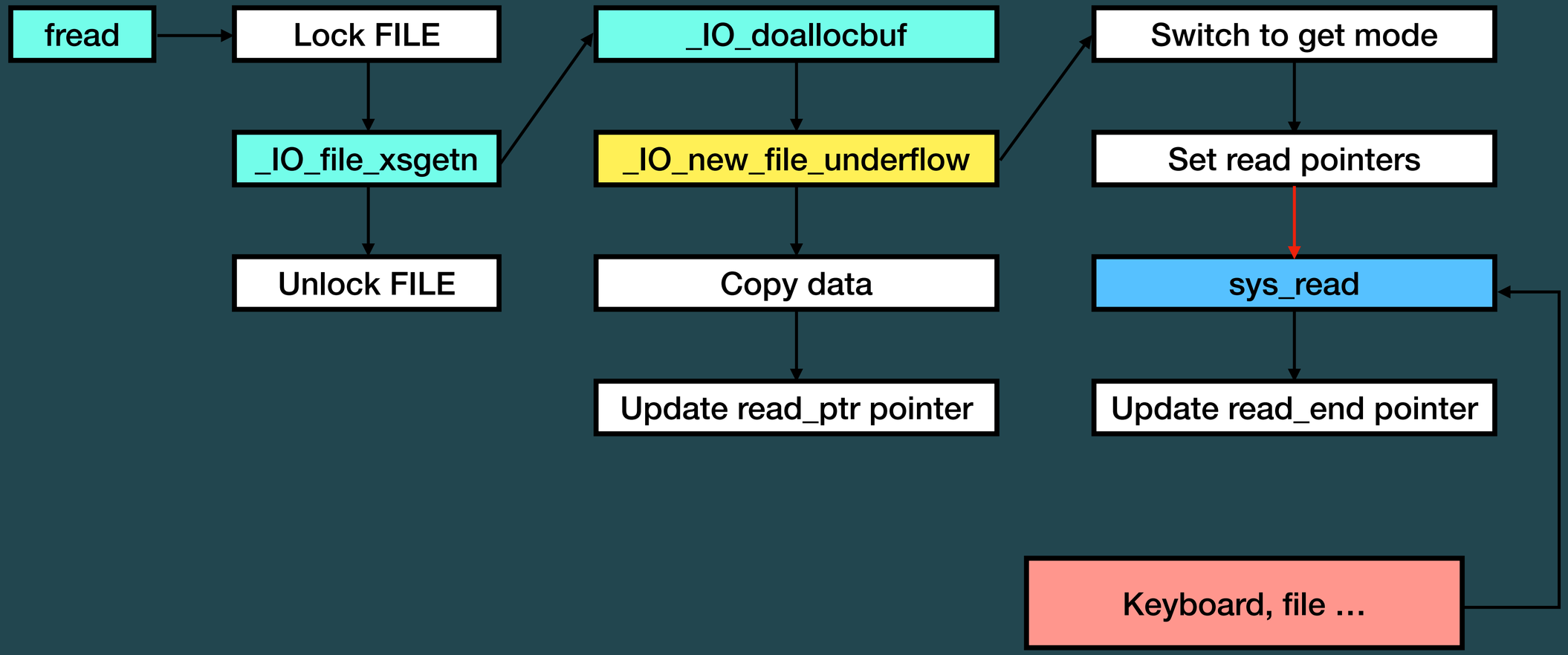
$ Background

Fread

- Buffer 一共分成 full buffer、line buffer 以及 no buffer，介紹會以 full buffer 為主
- 分析以下列常用的 function 來了解 FILE 的使用：
  - fopen( ) - 開啟檔案並且初始化 FILE 結構
  - fread( ) - 從 file descriptor 讀取資料
  - fwrite( ) - 寫入資料到 file descriptor
  - fclose( ) - 關閉檔案並且釋放 FILE 結構

$ Background

Fread



$ Background

Fread

- fread (= _IO_fread)

```
size_t
_IO_fread (void *buf, size_t size, size_t count, FILE *fp)
{
  size_t bytes_requested = size * count;
  size_t bytes_read;
  CHECK_FILE (fp, 0);
  if (bytes_requested == 0)
    return 0;
  _IO_acquire_lock (fp);
  bytes_read = _IO_sgetn (fp, (char *) buf, bytes_requested);
  _IO_release_lock (fp);
  return bytes_requested == bytes_read ? count : bytes_read / size;
}
```

1. 上鎖

2. 呼叫 _IO_sgetn (= _IO_file_xsgetn)

3. 解鎖

$ Background

Fread

- _IO_file_xsgetn

```
size_t
_IO_file_xsgetn(FILE *fp, void *data, size_t n)
{
  size_t want, have;
  ssize_t count;
  char *s = data;

  want = n;

  if (fp->_IO_buf_base == NULL)
  {
    _IO_doallocbuf(fp);
  }

  while (want > 0)
  {
    have = fp->_IO_read_end - fp->_IO_read_ptr;
    if (want <= have)
    {
      memcpy(s, fp->_IO_read_ptr, want);
      fp->_IO_read_ptr += want;
      want = 0;
    }
    else
    {
      if (have > 0)
      {
        s = __mempcpy(s, fp->_IO_read_ptr, have);
        want -= have;
        fp->_IO_read_ptr += have;
      }
      if (fp->_IO_buf_base && want < (size_t)(fp->_IO_buf_end - fp->_IO_buf_base))
      {
        if (__underflow(fp) == EOF)
          break;
```

1. 初始化 buffer 與 pointer

3. 從 buffer 複製資料 (這裡是步驟三，因為在執行 underflow 後 buffer 才會有資料)

4. 更新 read_ptr

2. 呼叫 _IO_new_file_underflow 讀取資料

$ Background

Fread

- _IO_file_xsgetn

```
    _IO_setg(fp, fp->_IO_buf_base, fp->_IO_buf_base, fp->_IO_buf_base);
    _IO_setp(fp, fp->_IO_buf_base, fp->_IO_buf_base);

    count = want;
    if (fp->_IO_buf_base)
    {
      size_t block_size = fp->_IO_buf_end - fp->_IO_buf_base;
      if (block_size >= 128)
        count -= want % block_size;
    }

    count = _IO_SYSREAD(fp, s, count);
    if (count <= 0)
    {
      if (count == 0)
        fp->_flags |= _IO_EOF_SEEN;
      else
        fp->_flags |= _IO_ERR_SEEN;

      break;
    }

    s += count;
    want -= count;
    if (fp->_offset != _IO_pos_BAD)
      _IO_pos_adjust(fp->_offset, count);
  }
}

return n - want;
```

處理超過 buffer 大小的請求，
會直接使用 sys_read

$ Background

Fread

- _IO_new_file_underflow

```
int _IO_new_file_underflow(FILE *fp)
{
  ssize_t count;

  if (fp->_flags & _IO_EOF_SEEN)
    return EOF;

  if (fp->_IO_read_ptr < fp->_IO_read_end)
    return *(unsigned char *)fp->_IO_read_ptr;

  if (fp->_IO_buf_base == NULL)
  {
    _IO_doallocbuf(fp);
  }

  _IO_switch_to_get_mode(fp);

  fp->_IO_read_base = fp->_IO_read_ptr = fp->_IO_buf_base;
  fp->_IO_read_end = fp->_IO_buf_base;
  fp->_IO_write_base = fp->_IO_write_ptr = fp->_IO_write_end = fp->_IO_buf_base;

  count = _IO_SYSREAD(fp, fp->_IO_buf_base,
                      fp->_IO_buf_end - fp->_IO_buf_base);
  if (count <= 0)
  {
    if (count == 0)
      fp->_flags |= _IO_EOF_SEEN;
    else
      fp->_flags |= _IO_ERR_SEEN, count = 0;
  }
  fp->_IO_read_end += count;
  if (count == 0)
  {
    fp->_offset = _IO_pos_BAD;
    return EOF;
  }
```

1. 如果原本在寫資料，就會先做 flush 並 unset putting flag

2. 將所有 pointer 設為 buf_base

3. 呼叫 sys_read

4. 更新 read_end

因為會多讀 (buf_end - buf_base) - want 的資料，因此下次讀取時就不一定需要呼叫 sys_read

$ Background

Fread

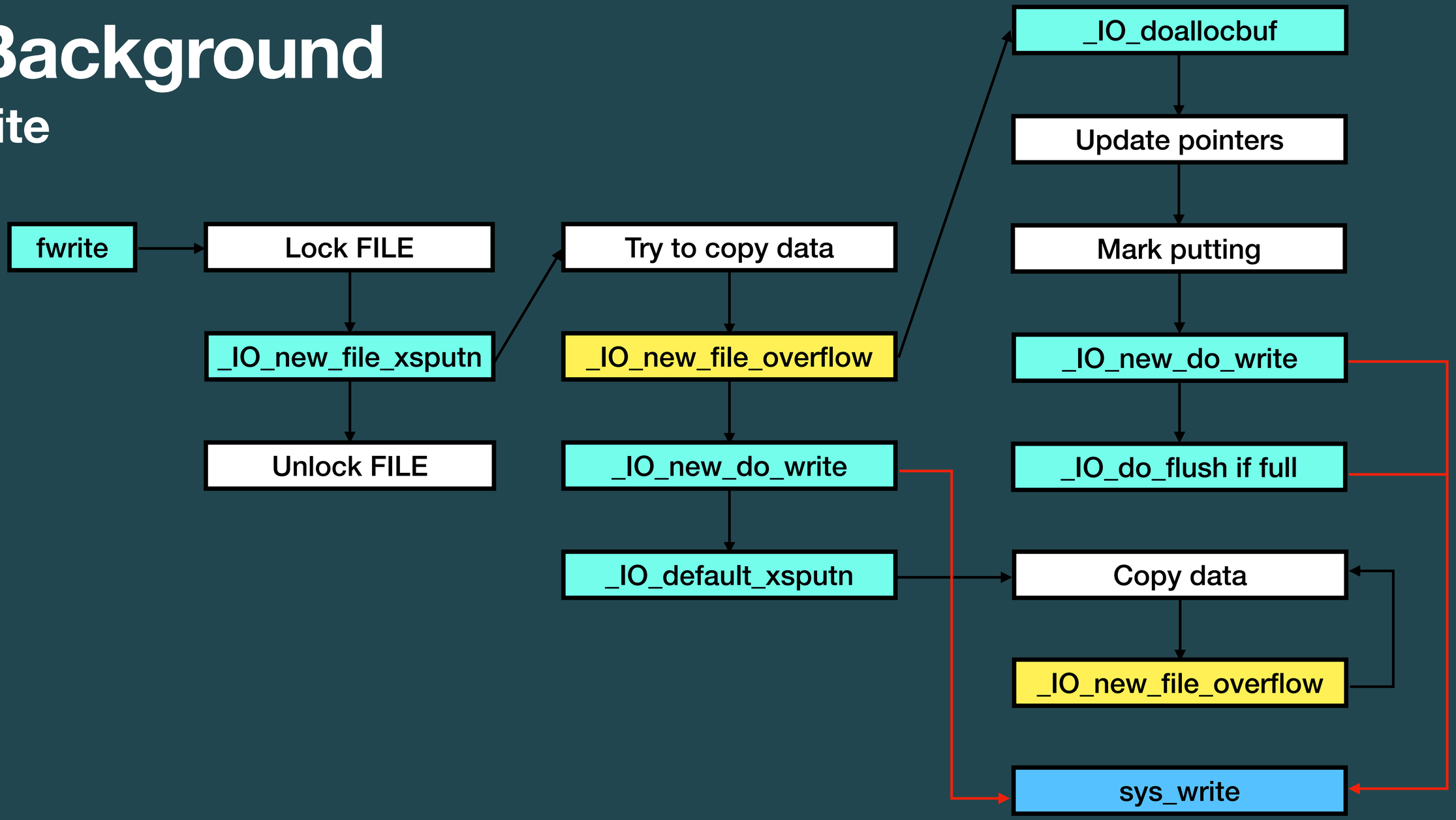
$ Background

Fwrite

- Buffer 一共分成 full buffer、line buffer 以及 no buffer，介紹會以 full buffer 為主
- 分析以下列常用的 function 來了解 FILE 的使用：
  - fopen( ) - 開啟檔案並且初始化 FILE 結構
  - fread( ) - 從 file descriptor 讀取資料
  - fwrite( ) - 寫入資料到 file descriptor
  - fclose( ) - 關閉檔案並且釋放 FILE 結構

$ Background

Fwrite



$ Background

Fwrite

- fwrite (= _IO_fwrite)

```
size_t
_IO_fwrite (const void *buf, size_t size, size_t count, FILE *fp)
{
  size_t request = size * count;
  size_t written = 0;
  CHECK_FILE (fp, 0);
  if (request == 0)
    return 0;
  _IO_acquire_lock (fp);
  if (_IO_vtable_offset (fp) != 0 || _IO_fwide (fp, -1) == -1)
    written = _IO_sputn (fp, (const char *) buf, request);
  _IO_release_lock (fp);
  if (written == request |
    return count;
  else
    return written / size;
}
```

1. 上鎖

3. 解鎖

2. 呼叫 _IO_sputs (= _IO_new_file_xsputn)

$ Background

Fwrite

- _IO_new_file_xsputn

```
size_t
_IO_new_file_xsputn(FILE *f, const void *data, size_t n)
{
  const char *s = (const char *)data;
  size_t to_do = n;
  int must_flush = 0;
  size_t count = 0;

  if (n <= 0)
    return 0;
  count = f->_IO_write_end - f->_IO_write_ptr;

  if (count > 0)
  {
    if (count > to_do)
      count = to_do;
    f->_IO_write_ptr = __mempcpy(f->_IO_write_ptr, s, count);
    s += count;
    to_do -= count;
  }
  if (to_do + must_flush > 0)
  {
    size_t block_size, do_write;
    if (_IO_OVERFLOW(f, EOF) == EOF)
      return to_do == 0 ? EOF : n - to_do;

    block_size = f->_IO_buf_end - f->_IO_buf_base;
    do_write = to_do - (block_size >= 128 ? to_do % block_size : 0);

    if (do_write)
    {
      count = new_do_write(f, s, do_write);
      to_do -= count;
      if (count < do_write)
        return n - to_do;
    }

    if (to_do)
      to_do -= _IO_default_xsputn(f, s + do_write, to_do);
  }
  return n - to_do;
}
```

1. 如果先前已經有讀資料到 buffer，就直接 append

2. 嘗試透過 _IO_new_file_overflow 寫入先前的資料

3. 當資料過多，會先透過 sys_write 寫 data % buffer_size 的資料

4. 剩下的資料在用 _IO_default_xputn 寫

$ Background

Fwrite

- _IO_new_file_overflow

1. 第一次執行會初始化 buffer

2. 初始化 pointers 成 buf_base

3. 標記成已經有被使用 (putting)

4. 當 ch == EOF，代表要做 flush

5. 當 buffer 滿了也需要做 flush

```
int _IO_new_file_overflow(FILE *f, int ch)
{
  if ((f->_flags & _IO_CURRENTLY_PUTTING) == 0 || f->_IO_write_base == NULL)
  {
    if (f->_IO_write_base == NULL)
    {
      _IO_doallocbuf(f);
      _IO_setg(f, f->_IO_buf_base, f->_IO_buf_base, f->_IO_buf_base);
    }

    if (f->_IO_read_ptr == f->_IO_buf_end)
      f->_IO_read_end = f->_IO_read_ptr = f->_IO_buf_base;
    f->_IO_write_ptr = f->_IO_read_ptr;
    f->_IO_write_base = f->_IO_write_ptr;
    f->_IO_write_end = f->_IO_buf_end;
    f->_IO_read_base = f->_IO_read_ptr = f->_IO_read_end;

    f->_flags |= _IO_CURRENTLY_PUTTING;
    if (f->_mode <= 0 && f->_flags & (_IO_LINE_BUF | _IO_UNBUFFERED))
      f->_IO_write_end = f->_IO_write_ptr;
  }

  if (ch == EOF)
    return _IO_do_write(f, f->_IO_write_base,
                        f->_IO_write_ptr - f->_IO_write_base);

  if (f->_IO_write_ptr == f->_IO_buf_end) // buffer full
    if (_IO_do_flush(f) == EOF)
      return EOF;

  *f->_IO_write_ptr++ = ch;
  return (unsigned char)ch;
}
```

$ Background

Fwrite

- _IO_default_xsputs

```
size_t
_IO_default_xsputn(FILE *f, const void *data, size_t n)
{
  const char *s = (char *)data;
  size_t more = n;
  if (more <= 0)
    return 0;
  for (;;)
  {
    if (f->_IO_write_ptr < f->_IO_write_end)
    {
      size_t count = f->_IO_write_end - f->_IO_write_ptr;
      if (count > more)
        count = more;
      if (count > 20)
      {
        f->_IO_write_ptr = __mempcpy(f->_IO_write_ptr, s, count);
        s += count;
      }
      else if (count)
      {
        char *p = f->_IO_write_ptr;
        ssize_t i;
        for (i = count; --i >= 0;)
          *p++ = *s++;
        f->_IO_write_ptr = p;
      }
      more -= count;
    }
    if (more == 0 || _IO_OVERFLOW(f, (unsigned char)*s++) == EOF)
      break;
    more--;
  }
  return n - more;
}
```

1. 檢查是否有空間，不夠的話會先把資料給寫入 fd

2.1 小請求用 memcpy 複製到 buffer 當中

2.2 大請求用 for loop 複製

把資料複製到 buffer 資料，等到夠多或讀到換行時再一次執行 sys_write

$ Background

Fwrite

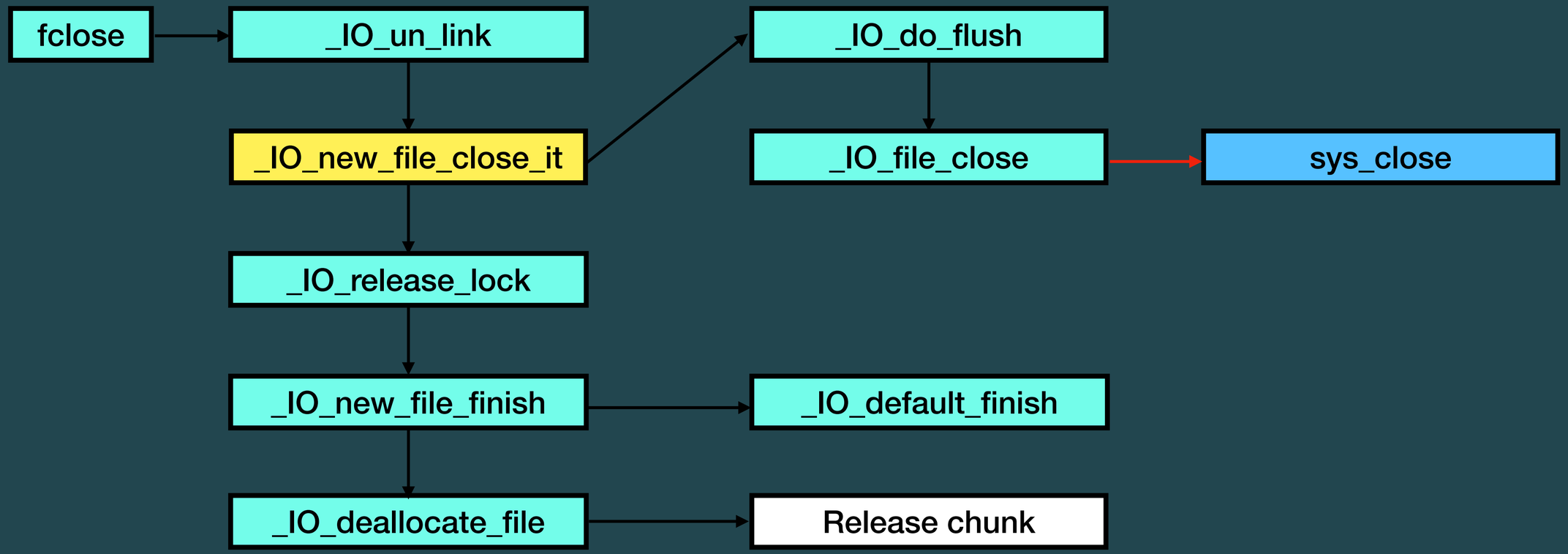
$ Background

Fclose

- Buffer 一共分成 full buffer、line buffer 以及 no buffer，介紹會以 full buffer 為主
- 分析以下列常用的 function 來了解 FILE 的使用：
  - fopen( ) - 開啟檔案並且初始化 FILE 結構
  - fread( ) - 從 file descriptor 讀取資料
  - fwrite( ) - 寫入資料到 file descriptor
  - fclose( ) - 關閉檔案並且釋放 FILE 結構

$ Background

Fclose



$ Background

Fclose

- fclose (= _IO_new_fclose)

```
int _IO_new_fclose(FILE *fp)
{
  int status;

  CHECK_FILE(fp, EOF);
  if (fp->_flags & _IO_IS_FILEBUF)
    _IO_un_link((struct _IO_FILE_plus *)fp);

  _IO_acquire_lock(fp);
  if (fp->_flags & _IO_IS_FILEBUF)
    status = _IO_file_close_it(fp);
  else
    status = fp->_flags & _IO_ERR_SEEN ? -1 : 0;
  _IO_release_lock(fp);
  _IO_FINISH(fp);
  if (_IO_have_backup(fp))
    _IO_free_backup_area(fp);
  _IO_deallocate_file(fp);
  return status;
}
```

1. 從 _IO_list_all 移除
2. 將資料寫入後關閉 fd
3. 重置部分結構內容
4. 釋放結構記憶體

$ Background

Fclose

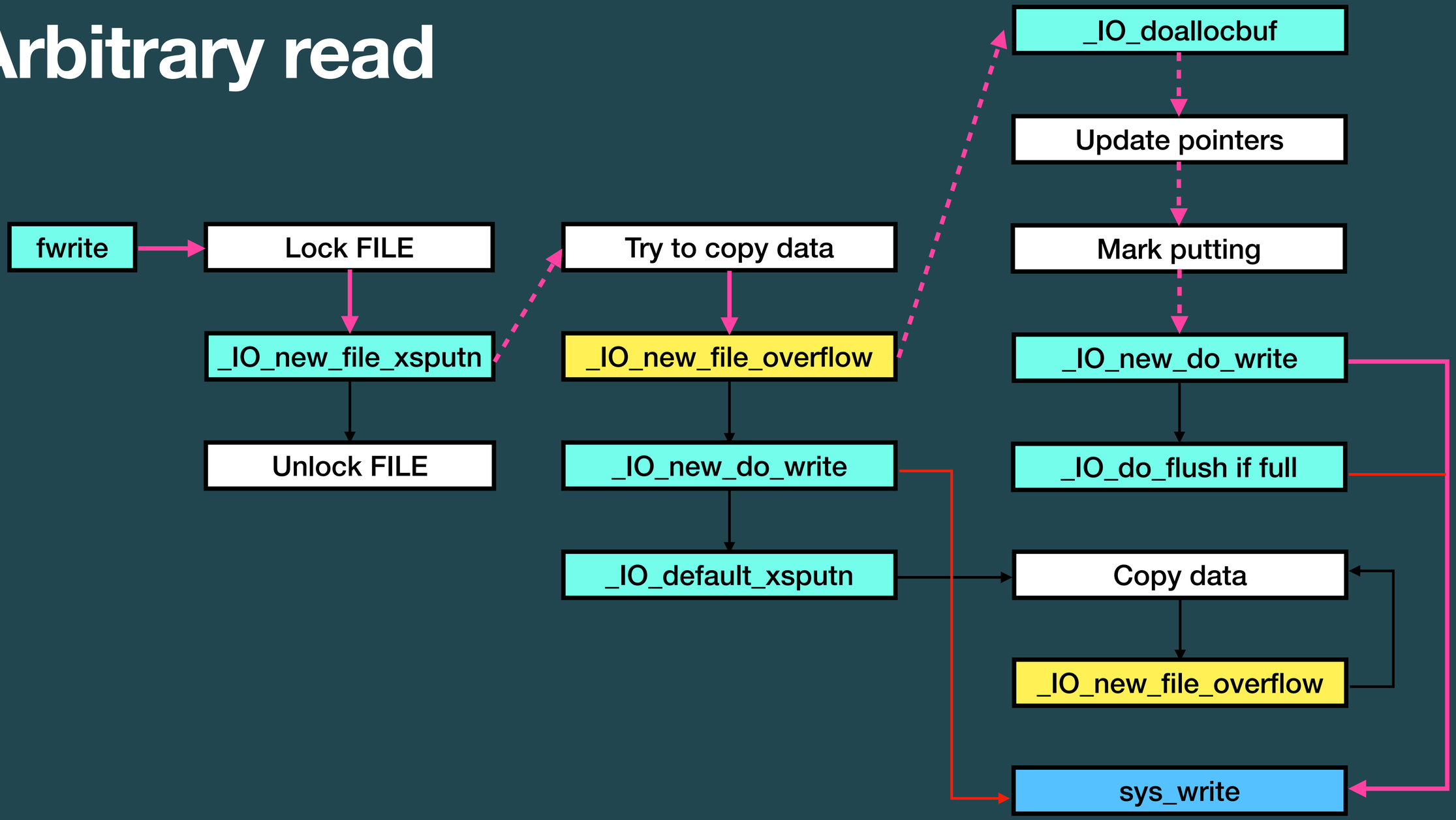
Arbitrary read

$ Arbitrary read

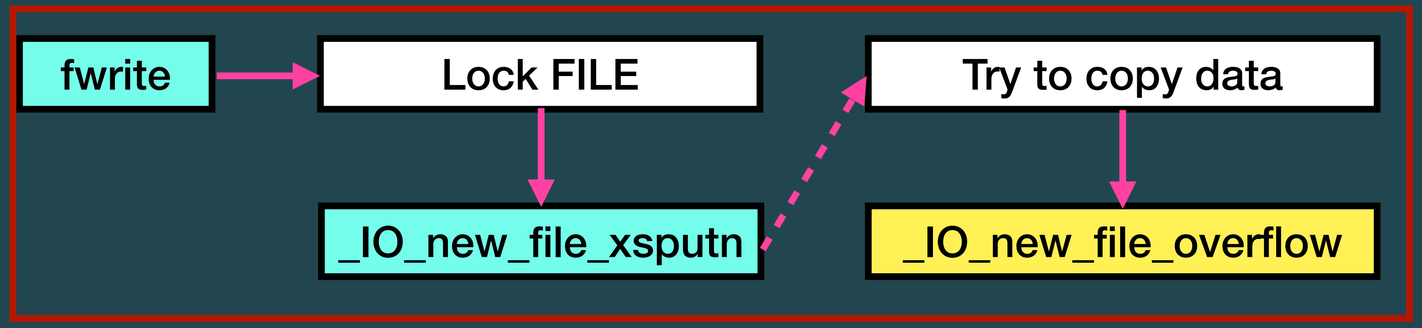
Concept

- 任意讀代表是將資料印出到 stdout (or stderr)，所以要控制的是 fwrite 的執行流程
- 設置：
  - flags &= ~NO_WRITES
  - flags |= (MAGIC | CURRENTLY_PUTTING)
  - fileno = 1 (輸出到 stdout)
  - write_end = 0 (滿足 write_end <= write_ptr)
  - read_end = write_base = target_address
  - write_ptr = target_address + target_size

$ Arbitrary read



$ Arbitrary read

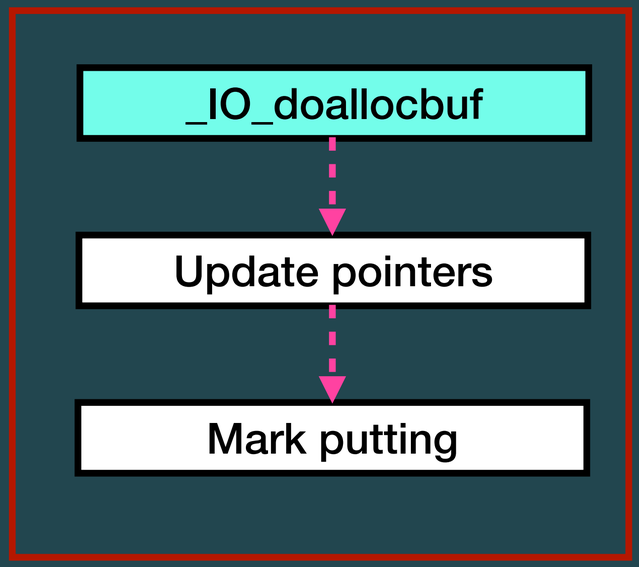


- 1. flags 需 & ~NO_WRITES，代表可寫
- 2. 避免複製，因此 write_end <= write_ptr
- 3. 呼叫 _IO_new_file_overflow

```
size_t
_IO_new_file_xsputn(FILE *f, const void *data, size_t n)
{
  const char *s = (const char *)data;
  size_t to_do = n;
  int must_flush = 0;
  size_t count = 0;

  if (n <= 0)
    return 0;

  ...
  else if (f->_IO_write_end > f->_IO_write_ptr)
    count = f->_IO_write_end - f->_IO_write_ptr;

  if (count > 0)
  {
    if (count > to_do)
      count = to_do;
    f->_IO_write_ptr = __mempcpy(f->_IO_write_ptr, s, count);
    s += count;
    to_do -= count;
  }
  if (to_do + must_flush > 0)
  {
    size_t block_size, do_write;
    if (_IO_OVERFLOW(f, EOF) == EOF)
      return to_do == 0 ? EOF : n - to_do;
```

$ Arbitrary read

```
int _IO_new_file_overflow(FILE *f, int ch)
{
  if ((f->_flags & _IO_CURRENTLY_PUTTING) == 0 | f->_IO_write_base == NULL)
  {
    if (f->_IO_write_base == NULL)
    {
      _IO_doallocbuf(f);
      _IO_setg(f, f->_IO_buf_base, f->_IO_buf_base, f->_IO_buf_base);
    }

    if (f->_IO_read_ptr == f->_IO_buf_end)
      f->_IO_read_end = f->_IO_read_ptr = f->_IO_buf_base;
    f->_IO_write_ptr = f->_IO_read_ptr;
    f->_IO_write_base = f->_IO_write_ptr;
    f->_IO_write_end = f->_IO_buf_end;
    f->_IO_read_base = f->_IO_read_ptr = f->_IO_read_end;

    f->_flags |= _IO_CURRENTLY_PUTTING;
    if (f->_mode <= 0 && f->_flags & (_IO_LINE_BUF | _IO_UNBUFFERED))
      f->_IO_write_end = f->_IO_write_ptr;
  }

  if (ch == EOF)
    return _IO_do_write(f, f->_IO_write_base,
                        f->_IO_write_ptr - f->_IO_write_base);
```

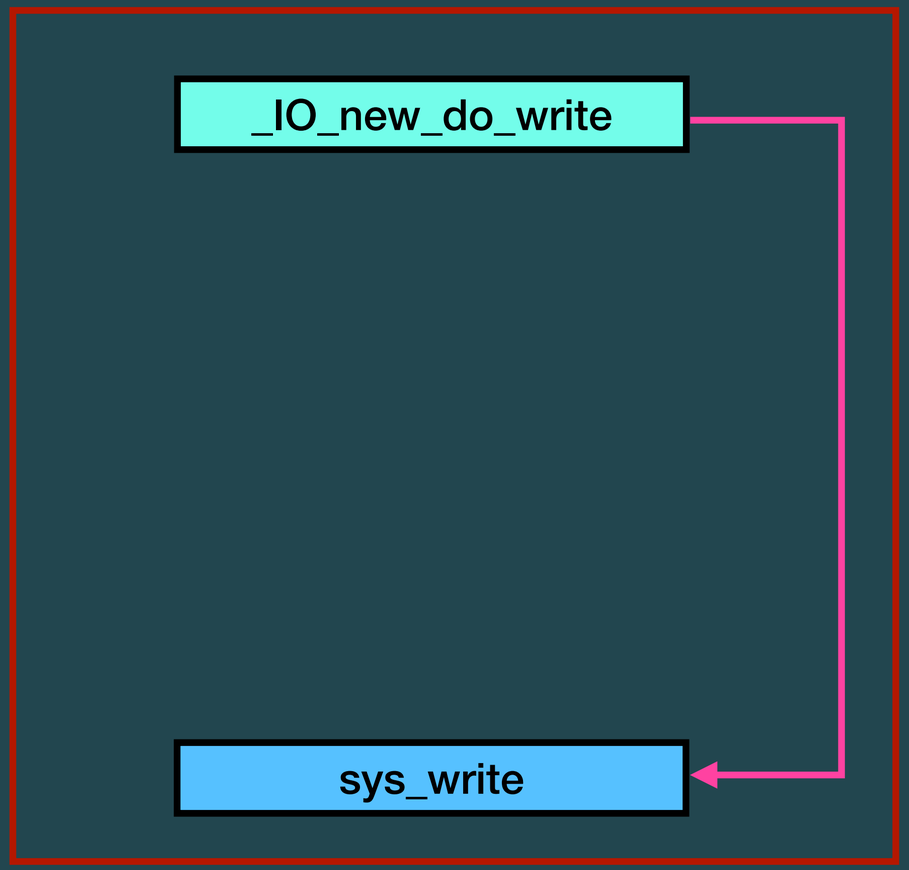


- 1. 避免初始化，設置 flags CURRENTLY_PUTTING
- 2. write_base 指向目標位址，write_ptr 設為 write_base + 資料大小

$ Arbitrary read

```
static size_t
new_do_write(FILE *fp, const char *data, size_t to_do)
{
  size_t count;
  if (fp->_flags & _IO_IS_APPENDING)
    fp->_offset = _IO_pos_BAD;
  else if (fp->_IO_read_end != fp->_IO_write_base)
  {
    off64_t new_pos = _IO_SYSSEEK(fp, fp->_IO_write_base -
    if (new_pos == _IO_pos_BAD)
      return 0;
    fp->_offset = new_pos;
  }
  count = _IO_SYSWRITE(fp, data, to_do);
```

- 1. 避免執行 sys_seek，設置 read_end == write_base
- 2. 最後呼叫 sys_write 印出目標位址的資料



$ Arbitrary read

Arbitrary write

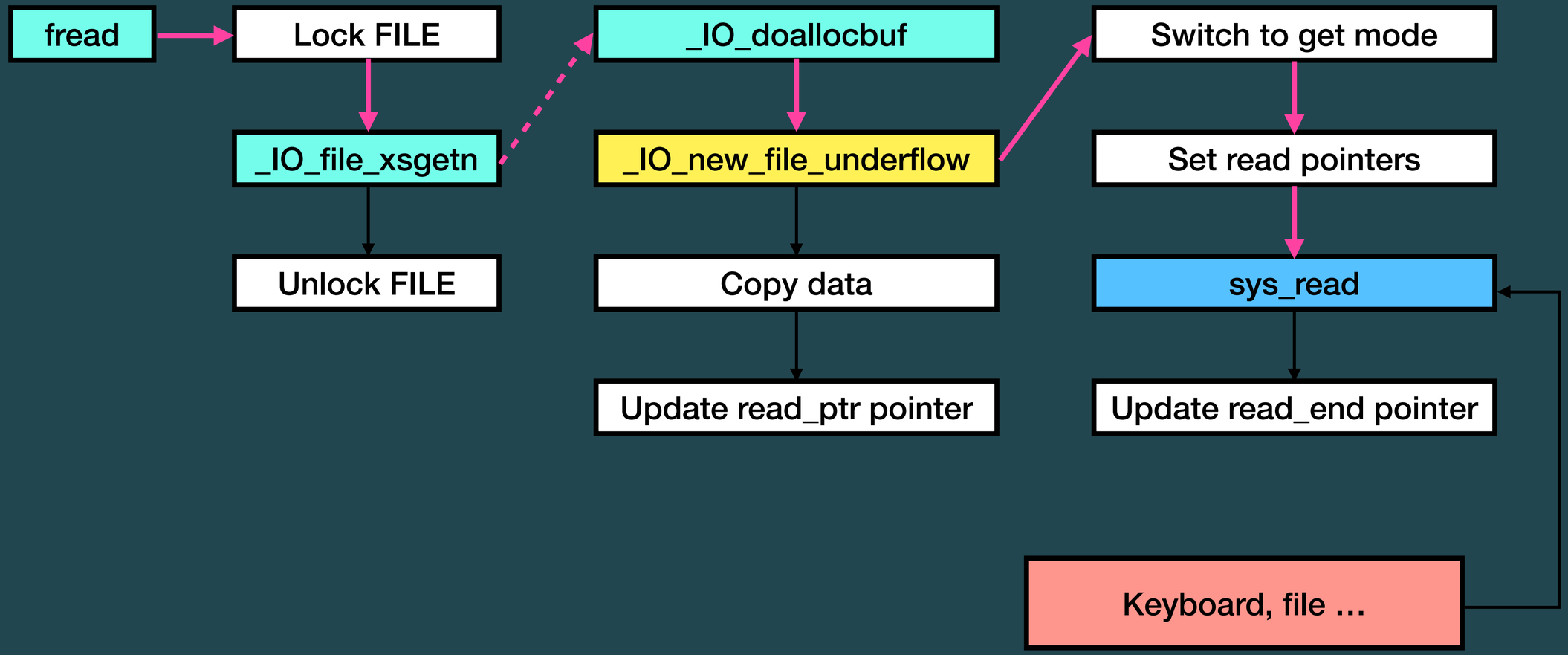
$ Arbitrary write

Concept

- 任意寫代表是把從 stdin 讀取的資料寫到特定位址，所以要控制的是 fread 的執行流程
- 設置：
  - flags &= ~(NO_READ | EOF_SEEN)
  - flags |= MAGIC
  - fileno = 0 (從 stdin 讀取)
  - read_end = read_ptr = 0 (滿足 read_end == read_ptr)
  - buf_base = target_address
  - buf_end = target_address + large value

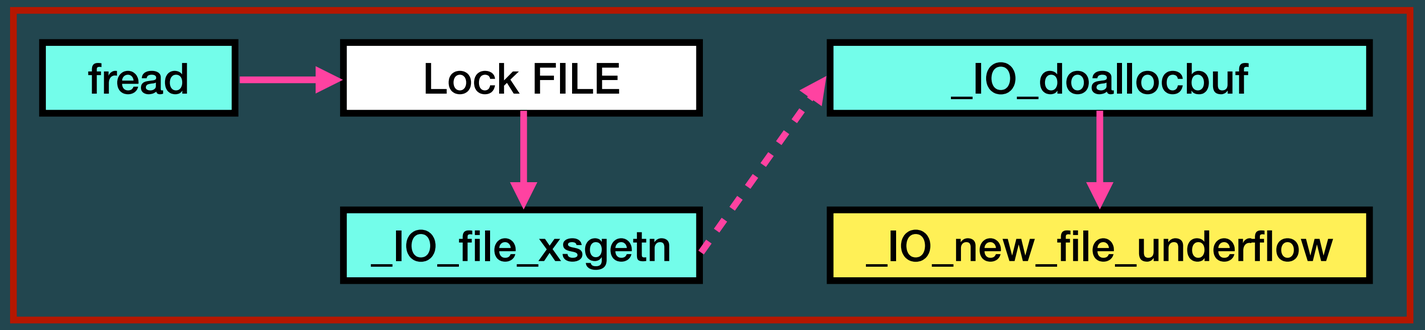
$ Arbitrary write

Concept



$ Arbitrary write

Concept



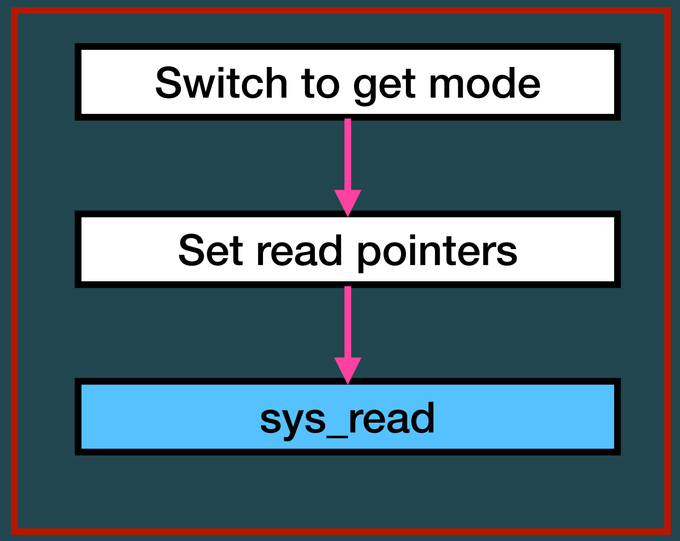
- 1. flags 需 & ~NO_READS，代表可讀
- 2. 避免複製，因此 read_end == read_ptr
- 3. 此條件與後續在 _IO_new_file_underflow 的限制相互呼應
- 4. 呼叫 _IO_new_file_underflow

```
size_t
_IO_file_xsgetn(FILE *fp, void *data, size_t n)
{
  size_t want, have;
  ssize_t count;
  char *s = data;

  want = n;

  if (fp->_IO_buf_base == NULL)
  {
    _IO_doallocbuf(fp);
  }

  while (want > 0)
  {
    have = fp->_IO_read_end - fp->_IO_read_ptr;
    if (want <= have)
    {
      memcpy(s, fp->_IO_read_ptr, want);
      fp->_IO_read_ptr += want;
      want = 0;
    }
    else
    {
      if (have > 0)
      {
        s = __mempcpy(s, fp->_IO_read_ptr, have);
        want -= have;
        fp->_IO_read_ptr += have;
      }
      if (fp->_IO_buf_base &&
          want < (size_t)(fp->_IO_buf_end - fp->_IO_buf_base))
      {
        if (__underflow(fp) == EOF)
          break;
```

$ Arbitrary write

Concept

```
int _IO_new_file_underflow(FILE *fp)
{
  ssize_t count;

  if (fp->_flags & _IO_EOF_SEEN)
    return EOF;

  if (fp->_IO_read_ptr < fp->_IO_read_end)
    return *(unsigned char *)fp->_IO_read_ptr;

  if (fp->_IO_buf_base == NULL)
  {
    _IO_doallocbuf(fp);
  }

  _IO_switch_to_get_mode(fp);

  fp->_IO_read_base = fp->_IO_read_ptr = fp->_IO_buf_base;
  fp->_IO_read_end = fp->_IO_buf_base;
  fp->_IO_write_base = fp->_IO_write_ptr = fp->_IO_write_end = fp->_IO_buf_base;

  count = _IO_SYSREAD(fp, fp->_IO_buf_base,
                      fp->_IO_buf_end - fp->_IO_buf_base);
```



- 1. 避免被視為 EOF，設置 flags & ~EOF_SEEN
- 2. buf_base 指向目標位址，buf_end 設為 buf_base + 夠大的值 (至少 >= want)

$ Arbitrary write

Arbitrary execute

$ Arbitrary execute

Concept

- FILE operation 會透過 function table (vtable) 來執行 overflow、underflow 等處理
- 以此為基礎，控制執行流程大致上有兩種方法，
  - 方法一： 竄改 function table，使其指向攻擊者可控的記憶體區塊
  - 方法二： 竄改 function table entry，使其呼叫到攻擊者指定的 function
- 過去兩種方法皆可，不過現在 FILE 在使用 function table 前多了一些檢查
- ****最新版的 glibc 把 function table 放在 read-only segment，因此不能改寫 entry****

$ Arbitrary execute

Concept - Method 1

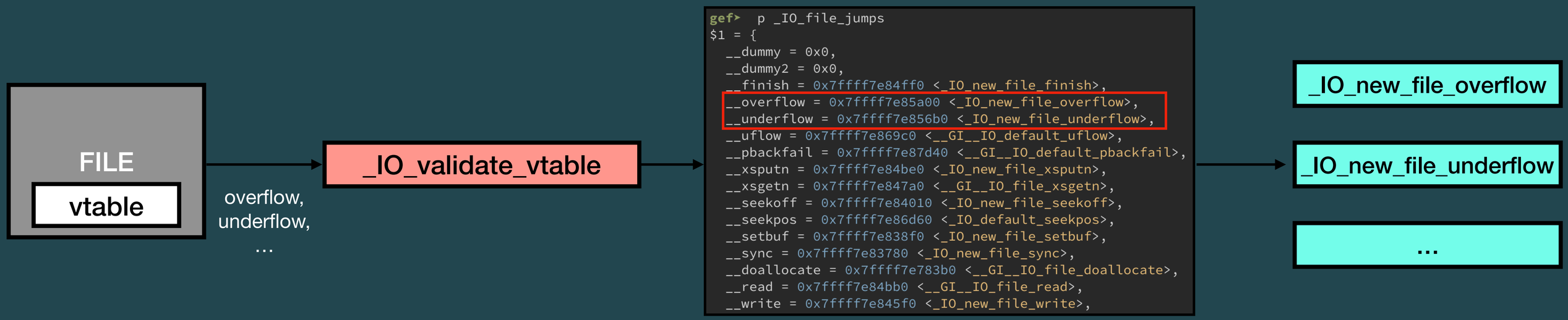
大寫的 function call 實際上是一個 macro，
會先檢查 vtable 合法性後才呼叫對應的 entry

- 1. 檢查 FILE vtable pointer 是否落在合法範圍
- 2. 不合法的情況會請 linker 幫忙檢查，如果失敗程式就會終止

不好繞

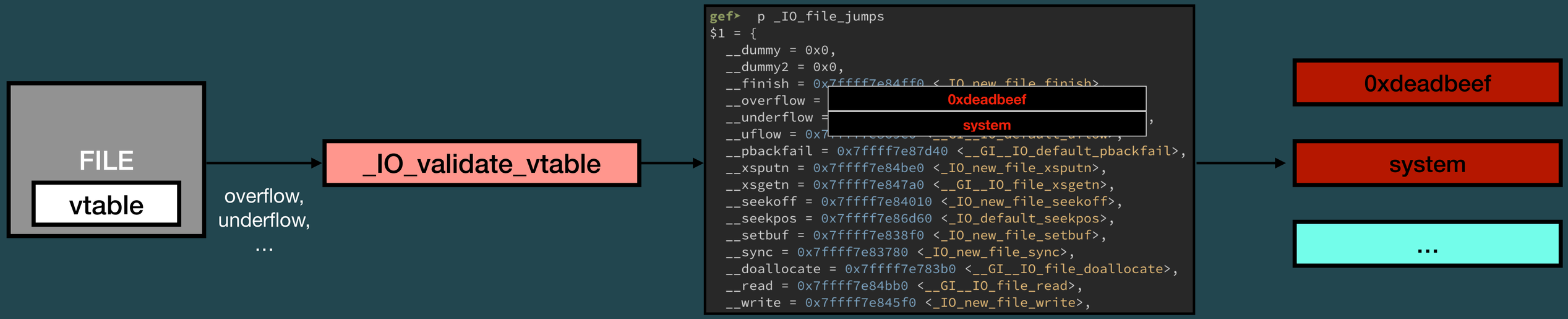
$ Arbitrary execute

Concept - Method 2



$ Arbitrary execute

Concept - Method 2



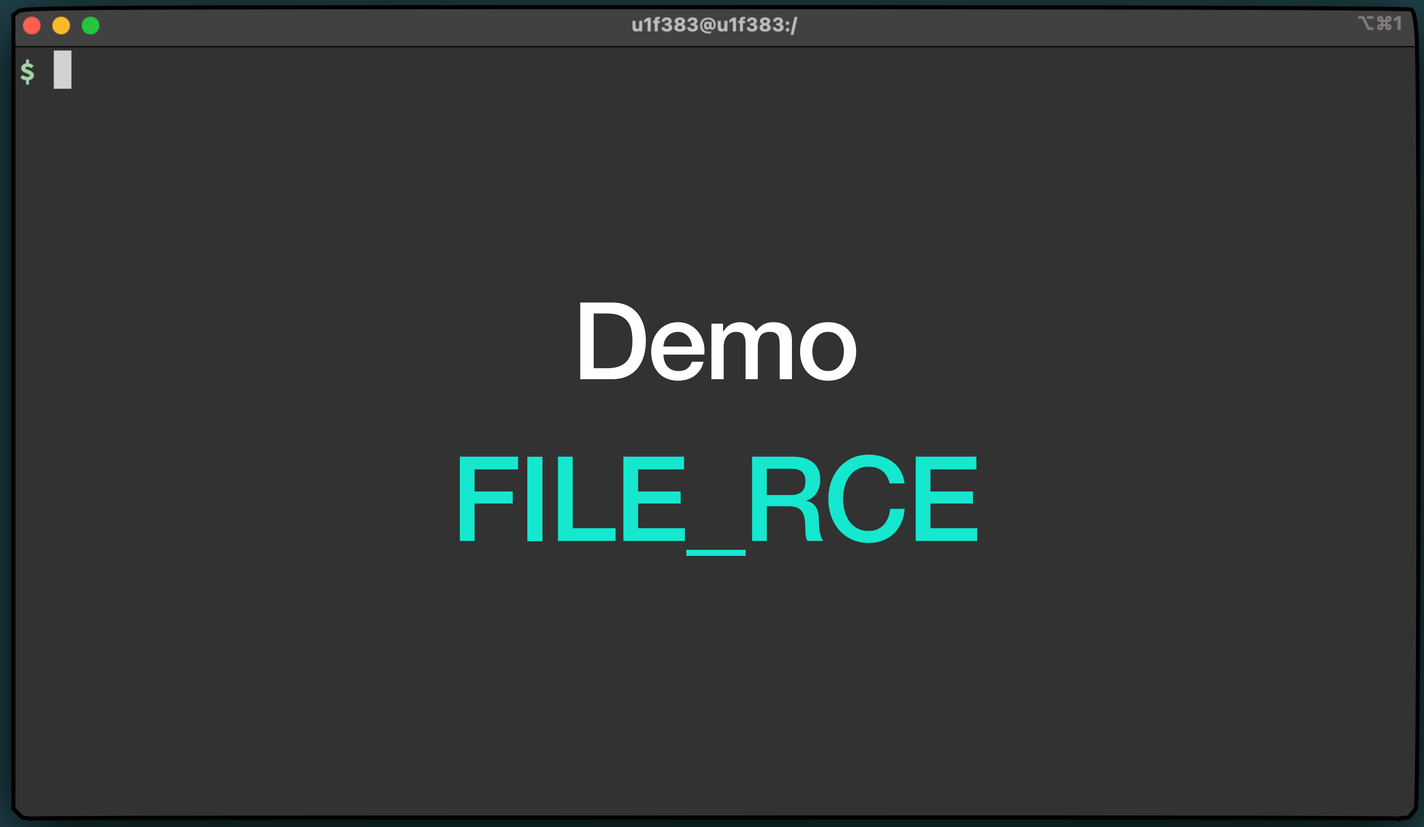
$ Arbitrary execute

Conclusion

- 結論
  - 方法一： 檢查機制嚴苛，不太可行
  - 方法二： 只要 function table 可寫就可行，而實際上 function table 落於 rw- page 當中

```
gef➤  xinfo &_IO_file_jumps
──────────────────────────────────────────────────────────────────────────────── xinfo: 0x7ffff7fc24a0 ─
Page: 0x007ffff7fc0000  →  0x007ffff7fc3000 (size=0x3000)
Permissions: rw-
Pathname: /usr/src/glibc/glibc_dbg/libc.so
Offset (from page): 0x24a0
Inode: 7235805
Segment: __libc_IO_vtables (0x007ffff7fc18a0-0x007ffff7fc2608)
Offset (from segment): 0xc00
Symbol: __GI__IO_file_jumps
```

$ Arbitrary execute



$ Summary

- 目前 FILE 結構主要還是拿來做任意讀/寫，直接控制執行流程就比較少
- 像 House of XXX 系列這種進階的 heap 攻擊手法，有些會配合著 FILE 做利用


u1f383@u1f383:/
⌥⌘1
$
$ Thanks for listening 🎃