OS161

Running a user process

Loading a Program into an Address Space

- When the kernel creates a process to run a particular program, it must create an address space for the process, and load the program's code and data into that address space

- A program's code and data is described in an *executable file*, which is created when the program is compiled and linked
  - OS/161 (and other operating systems) expect executable files to be in `ELF(Executable and Linking Format)` format

- The OS/161 kernel menu creates a user proces in `common_prog`

```
int common_prog(int nargs, char **args);
```

- the OS/161 `fork` system call allows process creation by user process duplication (so keeping same executable)
-
```
pid fork(void);
```
- the OS/161 `execv` system call allows a user process to replace its executable file/ program (and address space)

```
int execv(const char *program, char **args);
```

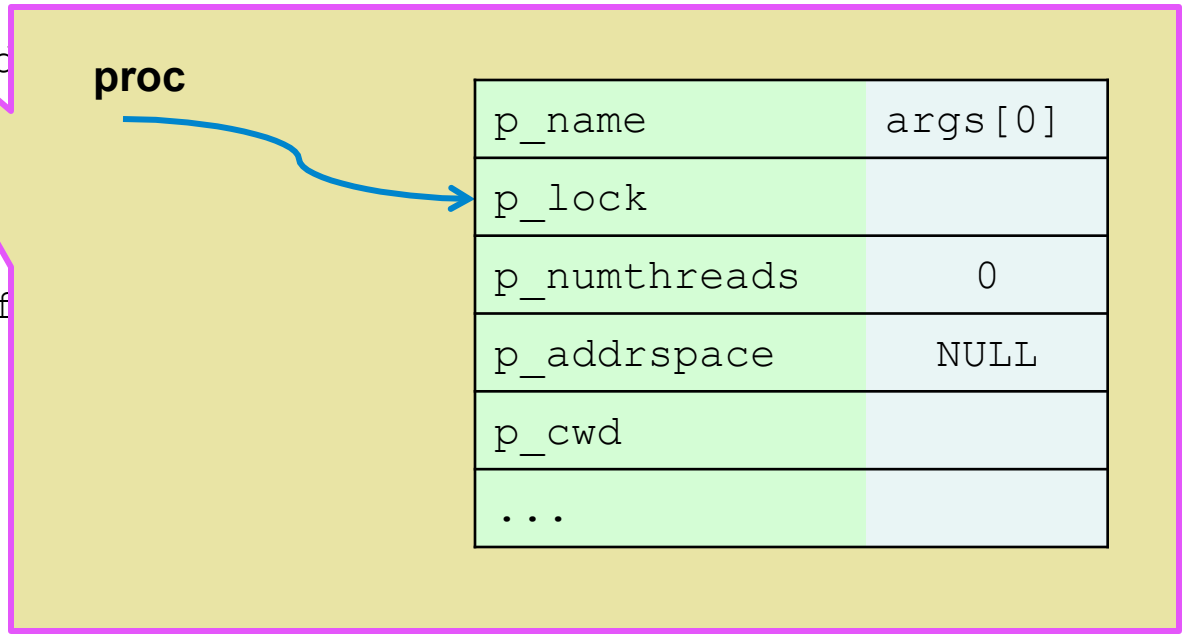
  - The program parameter of the `execv` system call should be the name of the `ELF` executable file for the program that is to be loaded into the address space.
  - A process needs an address space where to load the executable (`ELF`) file and run the process (in user mode).

runprogram

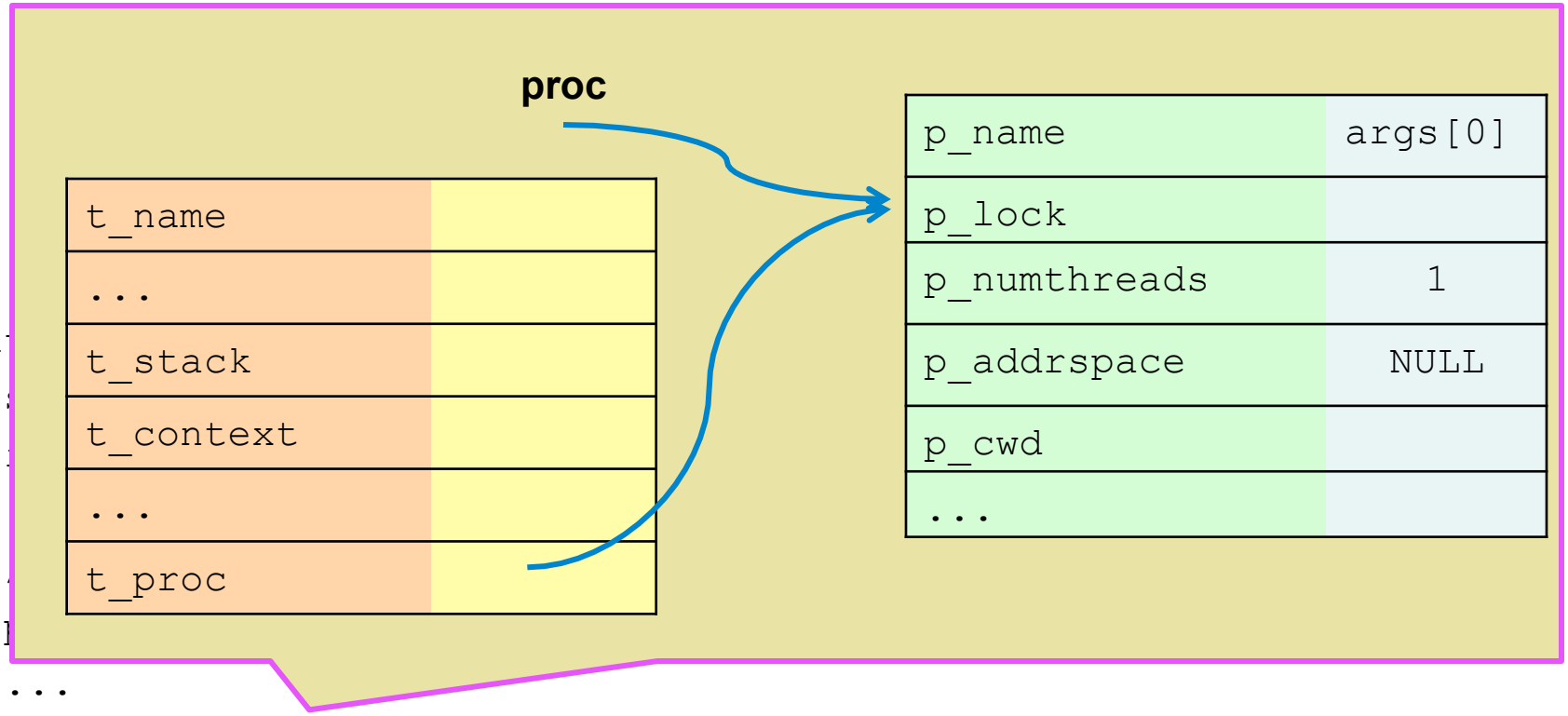
```
int common_prog (int nargs, char **args) {
  struct proc *proc;
  int result;

  /* Create a process for the new program to run in. */
  proc = proc_create_runprogram(args[0] /* name */);
  ...
  result = thread_fork(args[0] /* thread name */,
     proc /* new process */, cmd_progthread /* thread function */,
     args /* thread arg */, nargs /* thread arg */);
  ...
  /* TODO: wait for process termination */
  ...
  return 0;
}
```

common_prog

```
int common_prog (int nargs, char **args) {
  struct proc *proc;
  int result;

  /* Create a process for the new program to run in. */
  proc = proc_create_runprogram(args[0] /* name */);
  ...
  result = th ead
     proc /* n
     args /* th
  ...
  /* TODO: wait f
  ...
  return 0;
}
```

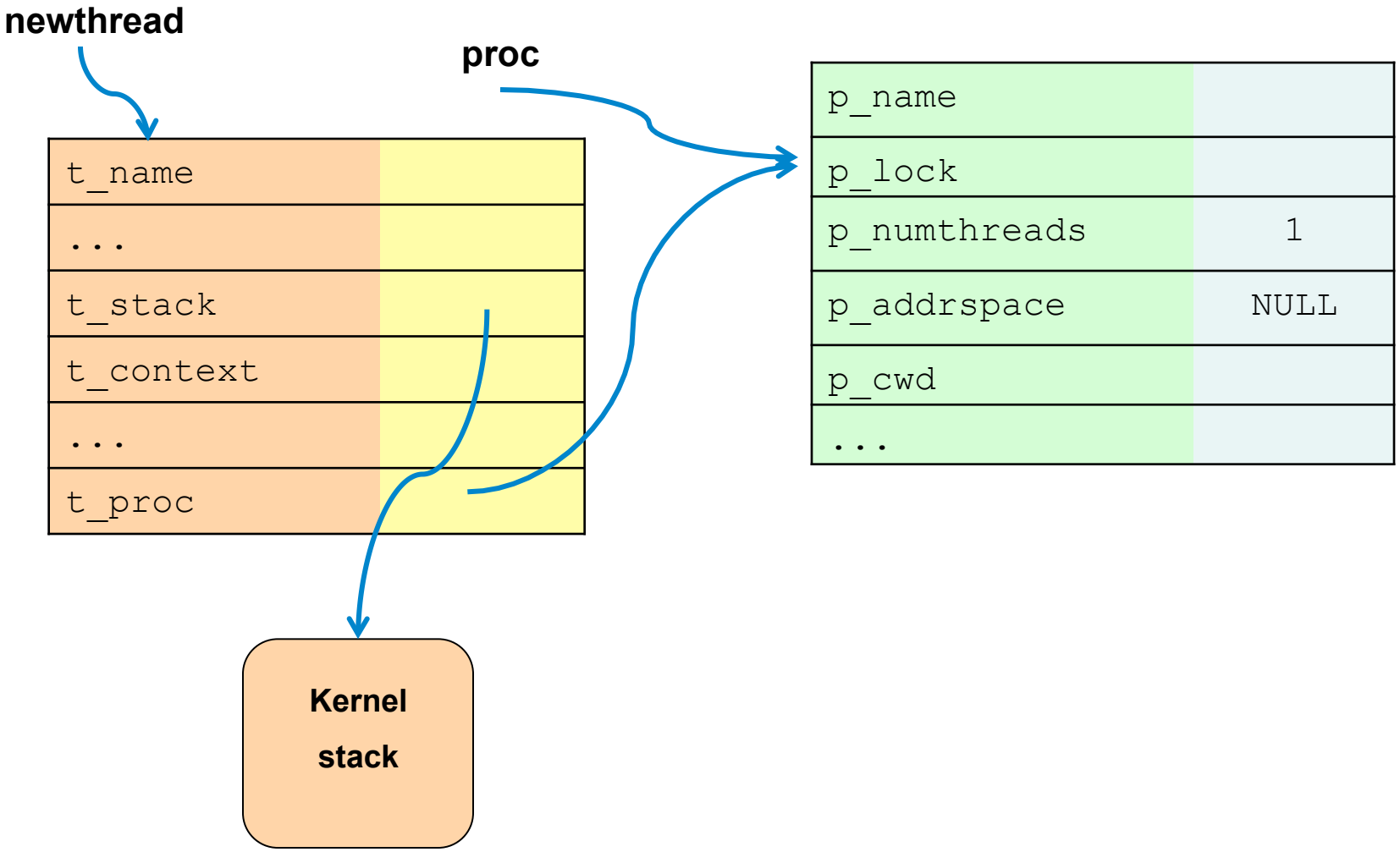


*/,

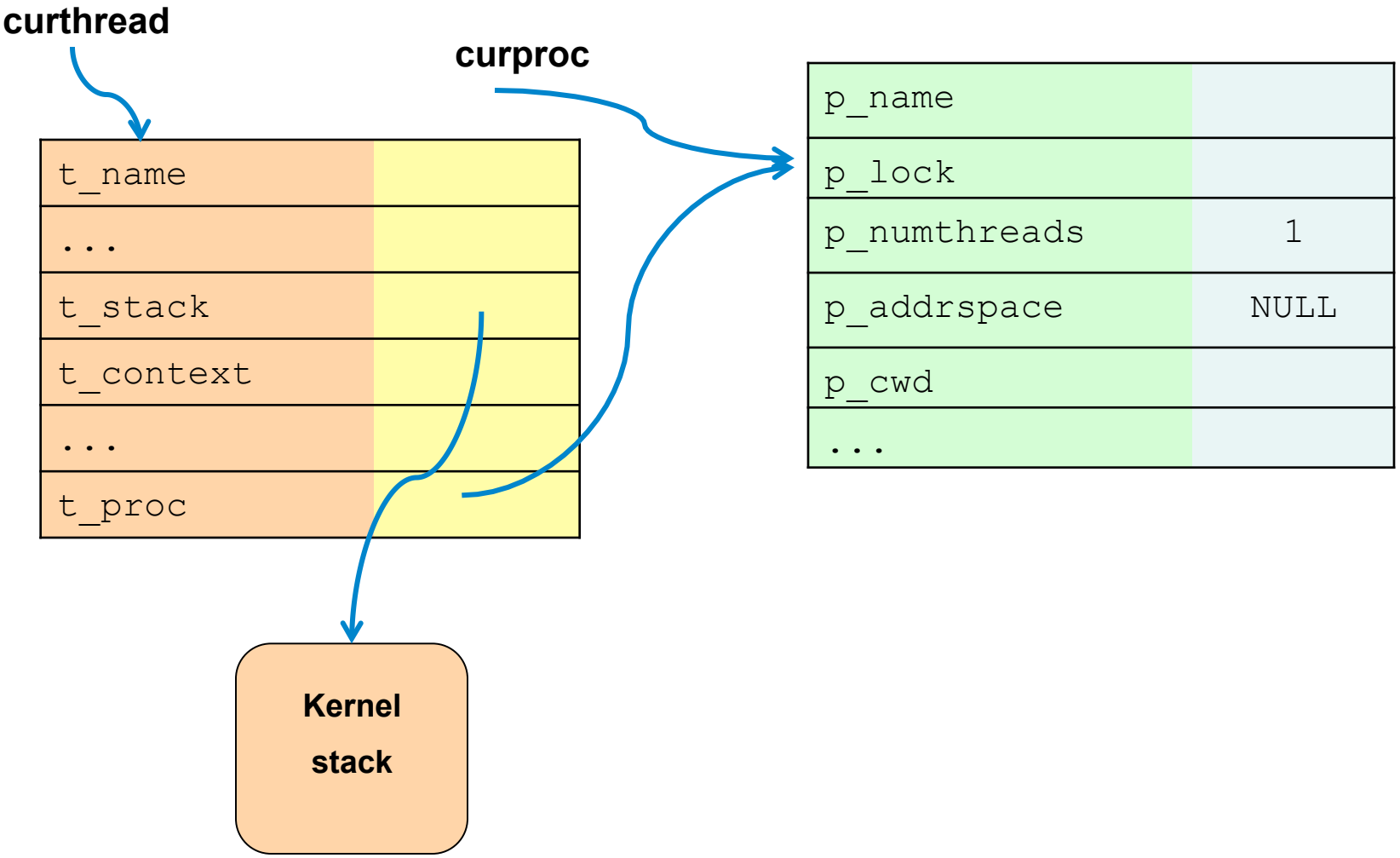



```
in
    ...
    result = thread_fork(args[0] /* thread name */,
        proc /* new process */, cmd_progthread /* thread function */,
        args /* thread arg */, nargs /* thread arg */);
    ...
    /* TODO: wait for process termination */
    ...
    return 0;
}
```

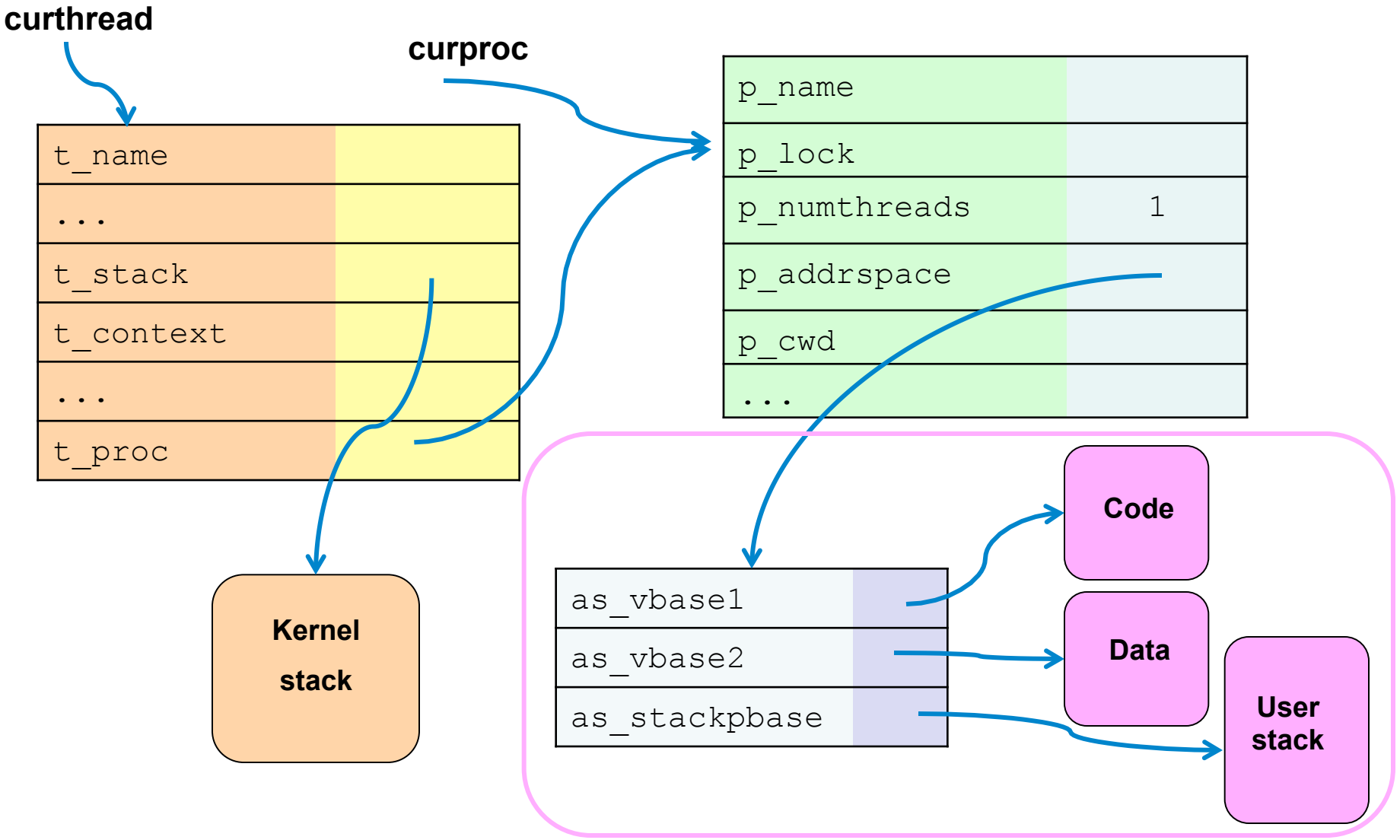
thread_fork (cmd_progthread)



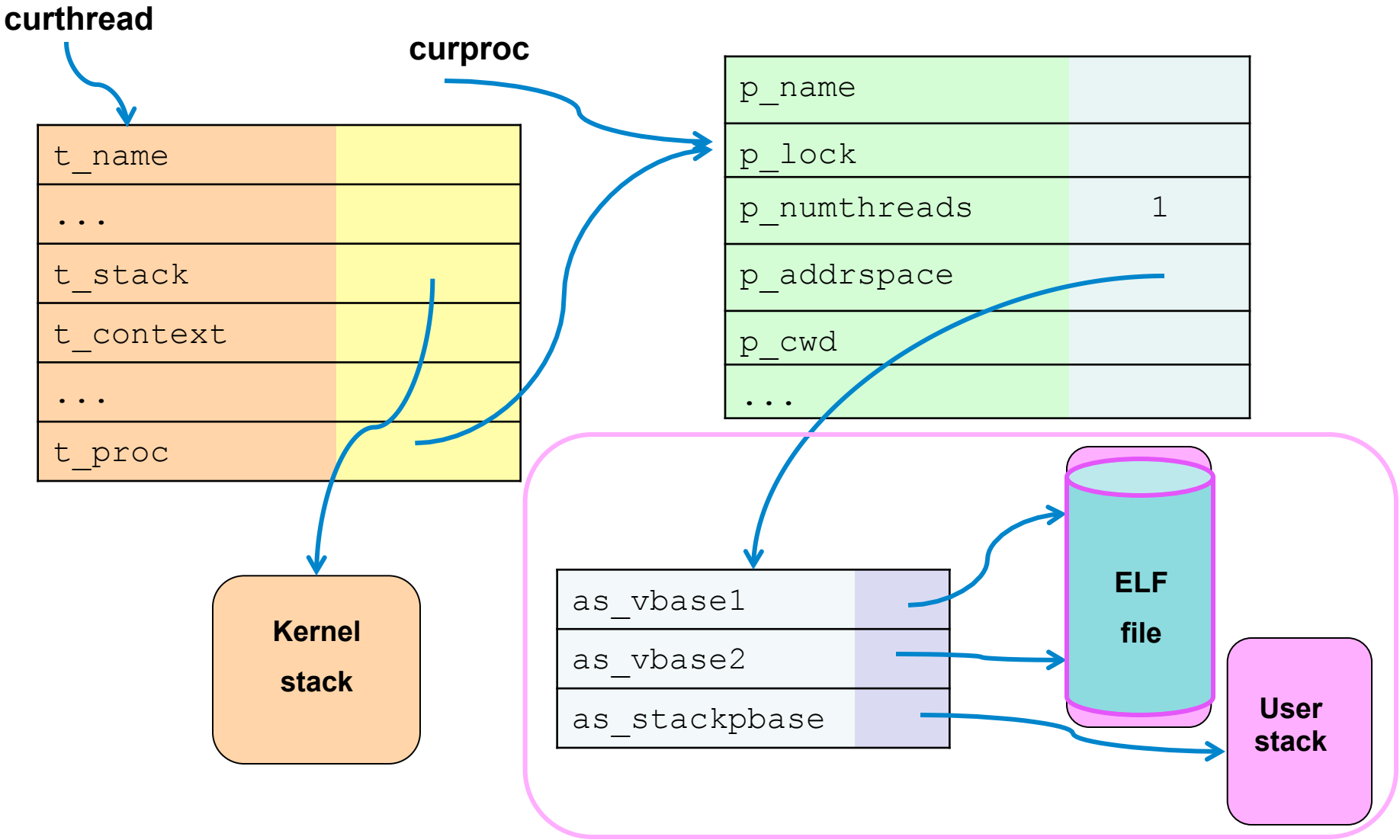
cmd_progthread



runprogram
(called by cmd_progthread)



runprogram
(called by cmd_progthread)



runprogram

```
runprogram(char *progname) {
  struct addrspace *as;
  struct vnode *v;
  vaddr_t entrypoint, stackptr;
  int result;

  /* Open the file. */
  result = vfs_open(progname, O_RDONLY, 0, &v);
  ...
  /* Create a new address space. */
  as = as_create();
  ...
  /* Switch to it and activate it. */
  proc_setas(as);
  as_activate();
```

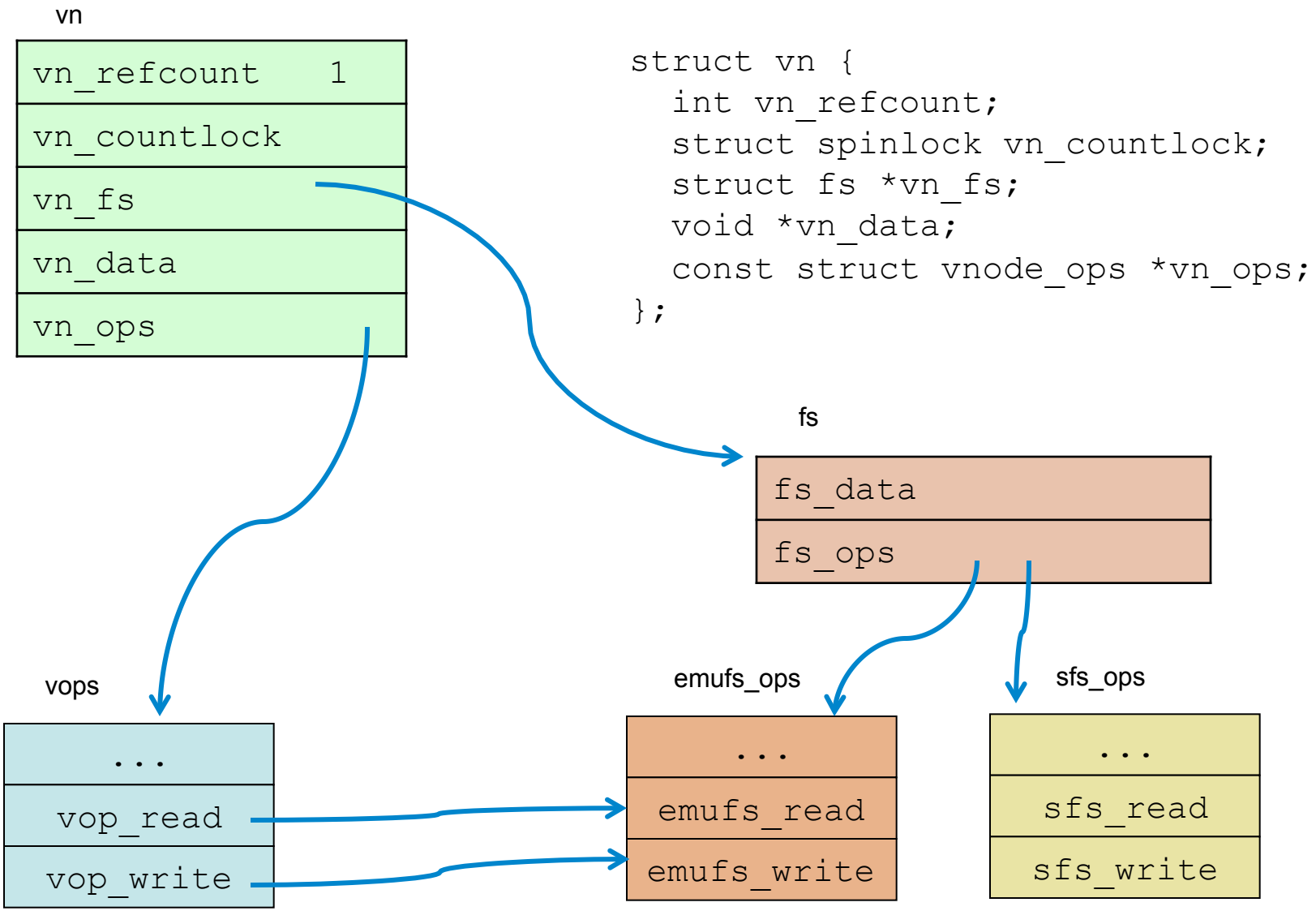
runprogram

```
  /* Load the executable. */
  result = load_elf(v, &entrypoint);
  ...
  /* Done with the file now. */
  vfs_close(v);

  /* Define the user stack in the address space */
  result = as_define_stack(as, &stackptr);
  ...
  /* Warp to user mode. */
  enter_new_process(0/*argc*/, NULL/*userspace addr of argv*/,
            NULL /*userspace addr of environment*/,
            stackptr, entrypoint);
  /* enter_new_process does not return. */
  panic("enter_new_process returned\n");
  return EINVAL;
}
```

load_elf (read exec hdr)

```
load_elf(struct vnode *v, vaddr_t *entrypoint) {
  Elf_Ehdr eh;   /* Executable header */
  Elf_Phdr ph;   /* "Program header" = segment header */
  int result, i;
  struct iovec iov;
  struct uio ku;
  struct addrspace *as;

  as = proc_getas();

  /* Read the executable header from offset 0 in the file. */
  uio_kinit(&iov, &ku, &eh, sizeof(eh), 0, UIO_READ);
  result = VOP_READ(v, &ku);
  …
```

Struct vnode



load_elf (read segment hdrs)

```
/*Go through the list of segments and set up the address space.*/
for (i=0; i<eh.e_phnum; i++) {
    off_t offset = eh.e_phoff + i*eh.e_phentsize;
    uio_kinit(&iov, &ku, &ph, sizeof(ph), offset, UIO_READ);
    result = VOP_READ(v, &ku);
    ...
    result = as_define_region(as,
                 ph.p_vaddr, ph.p_memsz,
                 ph.p_flags & PF_R,
                 ph.p_flags & PF_W,
                 ph.p_flags & PF_X);
}

result = as_prepare_load(as);
```

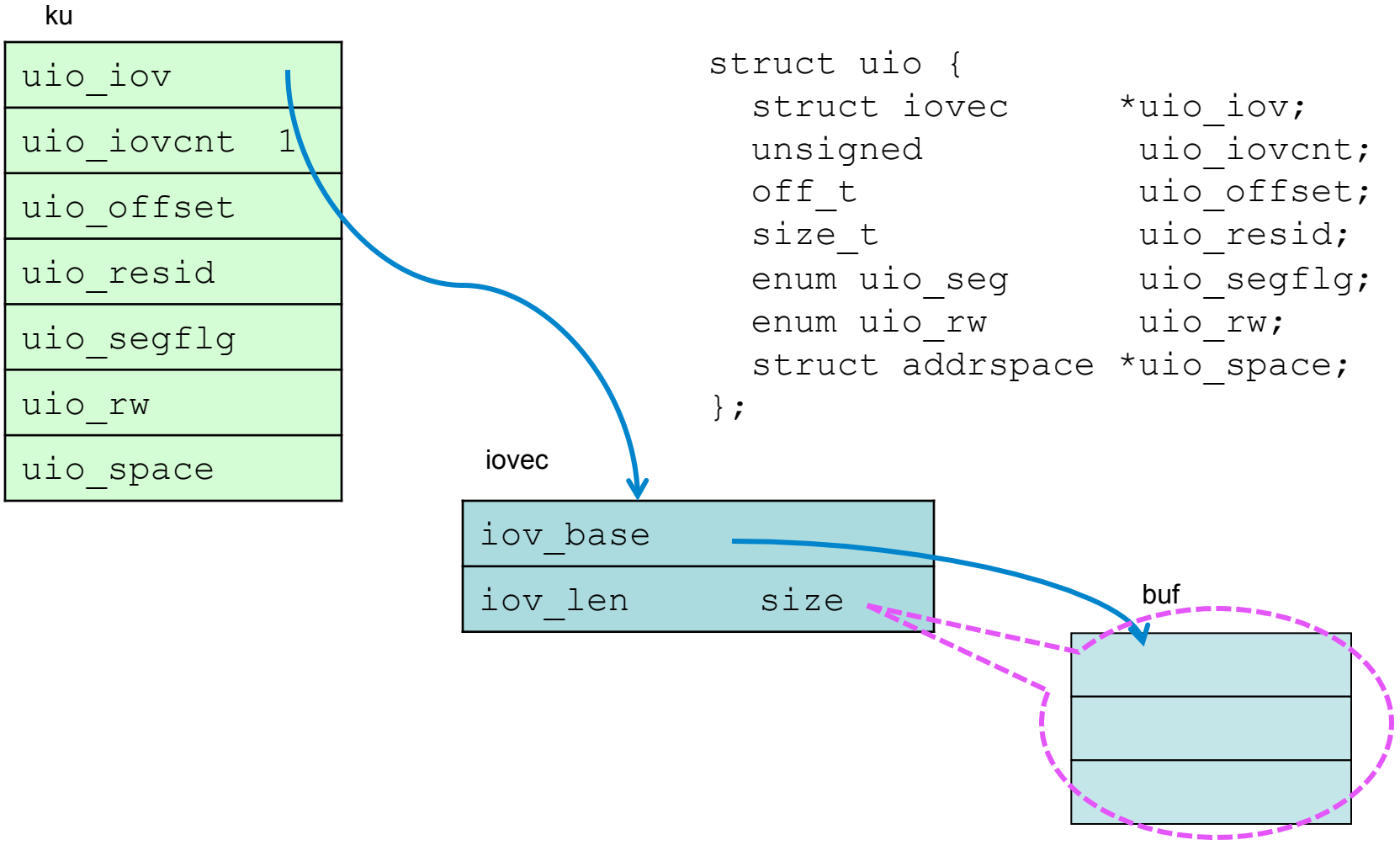
load_elf (read segments)

```
  /* Now actually load each segment. */
  for (i=0; i<eh.e_phnum; i++) {
      off_t offset = eh.e_phoff + i*eh.e_phentsize;
      uio_kinit(&iov, &ku, &ph, sizeof(ph), offset, UIO_READ);
      result = VOP_READ(v, &ku);

      result = load_segment(as, v, ph.p_offset, ph.p_vaddr,
                            ph.p_memsz, ph.p_filesz,
                            ph.p_flags & PF_X);
  }

  result = as_complete_load(as);

  *entrypoint = eh.e_entry;
  return 0;
}
```

Read from file

```
struct iovec iov;
struct uio ku;
struct addrspace *as;
Elf_Phdr ph;   /* "Program header" = segment header */
off_t offset = ...;

uio_kinit(&iov, &ku, &ph, sizeof(ph), offset, UIO_READ);
result = VOP_READ(v, &ku);
```

struct uio


ku
uio_iov
uio_iovcnt 1
uio_offset
uio_resid
uio_segflg
uio_rw
uio_space
struct uio {
struct iovec *uio_iov;
unsigned uio_iovcnt;
off_t uio_offset;
size_t uio_resid;
enum uio_seg uio_segflg;
enum uio_rw uio_rw;
struct addrspace *uio_space;
};
iovec
iov_base
iov_len size
buf

struct uio (kernel space)

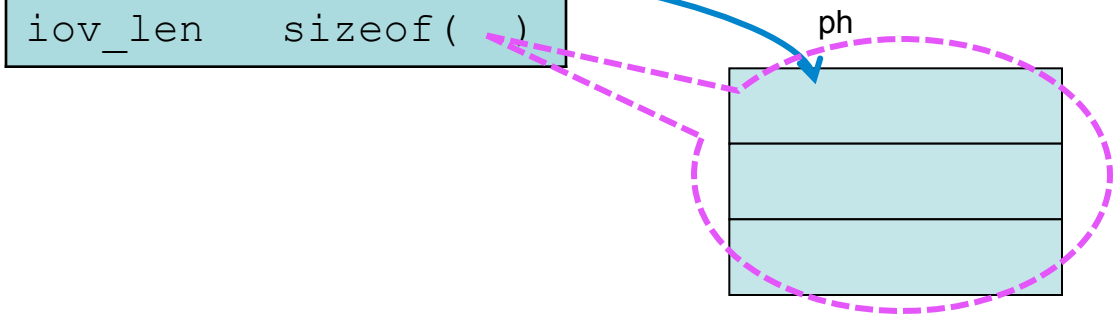
ku

uio_iov	
uio_iovcnt	1
uio_offset	
uio_resid	
uio_segflg	UIO_SYSSPACE
uio_rw	UIO_READ
uio_space	NULL

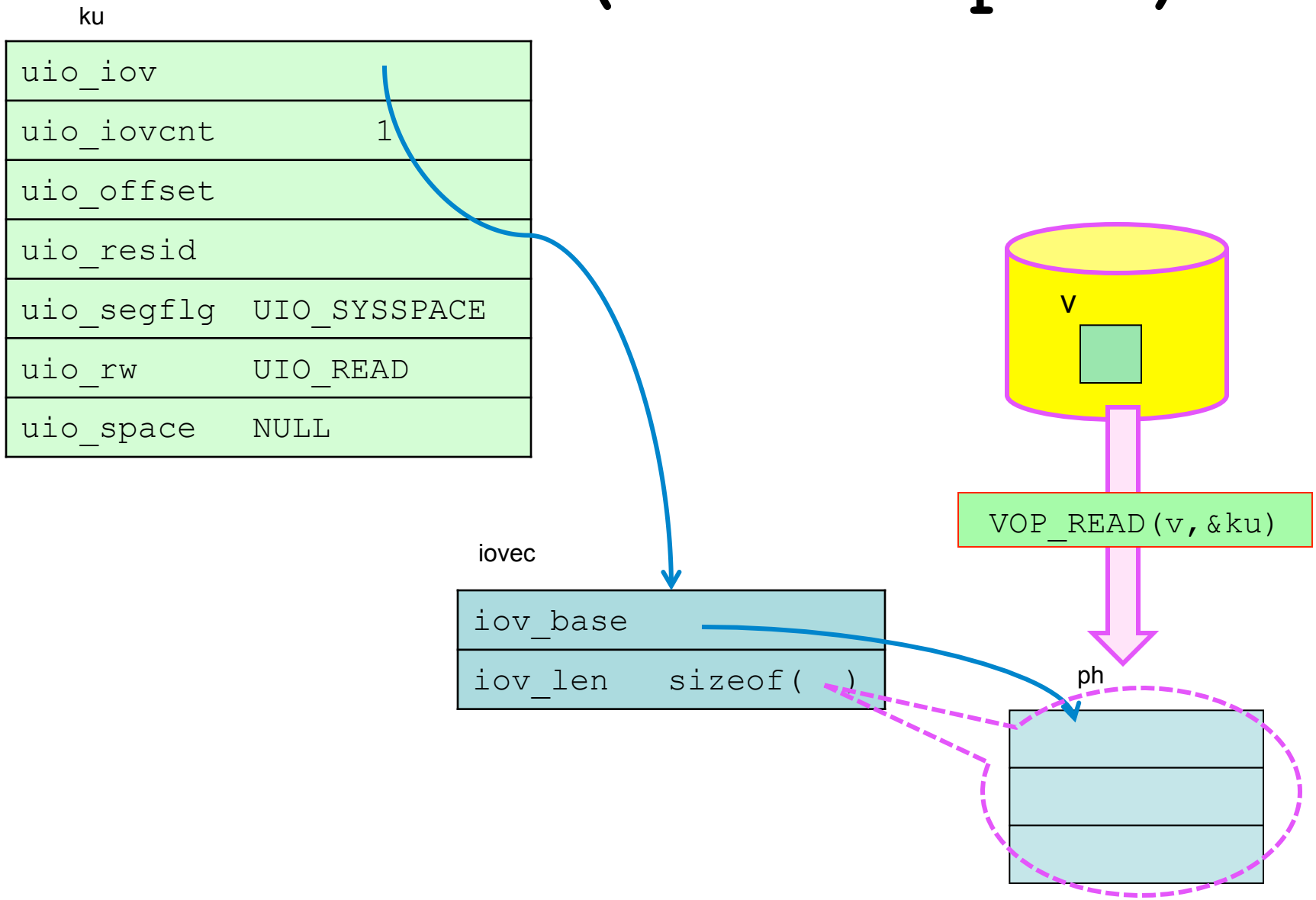
```
struct uio {
  struct iovec     *uio_iov;
  unsigned          uio_iovcnt;
  off_t             uio_offset;
  size_t            uio_resid;
  enum uio_seg      uio_segflg;
  enum uio_rw       uio_rw;
  struct addrspace *uio_space;
};
```

iovec

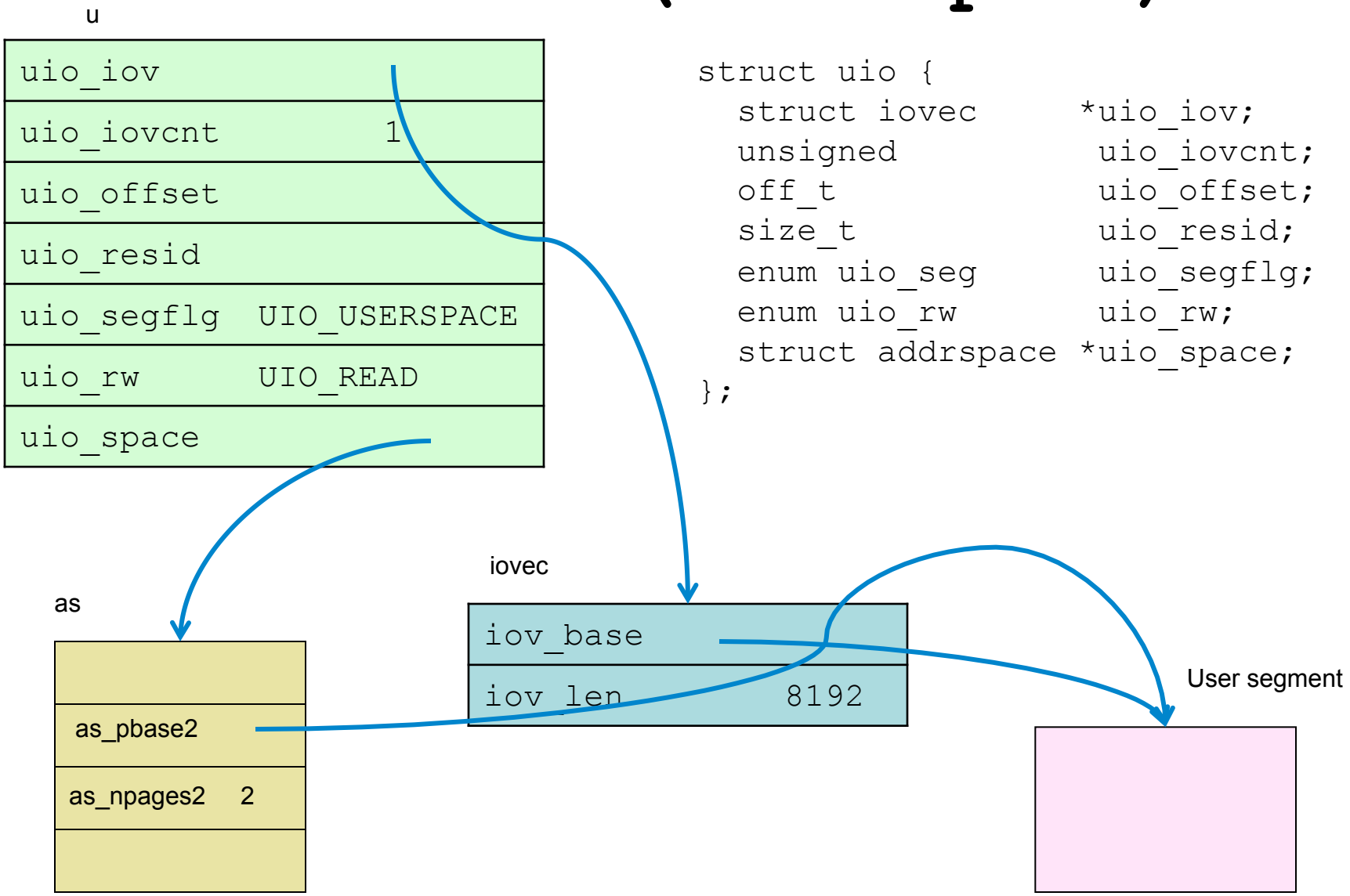
iov_base
iov_len sizeof( )



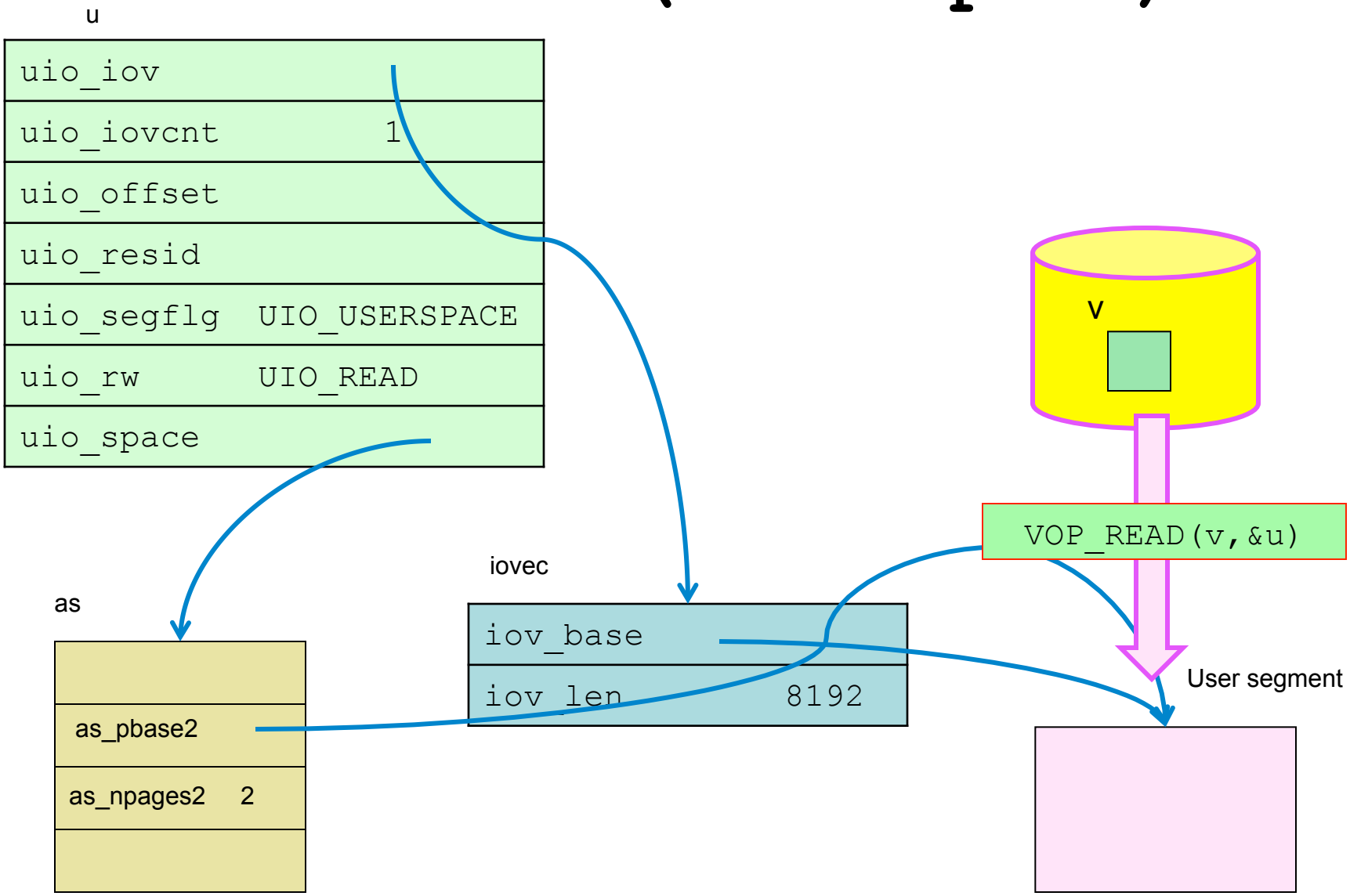
struct uio (kernel space)



struct uio (user space)



struct uio (user space)



vop_read
(emufs_read -> emu_read -> emu_doread)

```
int emu_doread(struct emu_softc *sc, uint32_t handle, uint32_t len,
       uint32_t op, struct uio *uio) {
    ...
    lock_acquire(sc->e_lock);
    emu_wreg(sc, REG_HANDLE, handle);
    emu_wreg(sc, REG_IOLEN, len);
    emu_wreg(sc, REG_OFFSET, uio->uio_offset);
    emu_wreg(sc, REG_OPER, op);
    result = emu_waitdone(sc);
    if (result) { goto out; }
    membar_load_load();
    result = uiomove(sc->e_iobuf, emu_rreg(sc, REG_IOLEN), uio);
    uio->uio_offset = emu_rreg(sc, REG_OFFSET);
 out: lock_release(sc->e_lock);
      return result;
}
```

vop_read
(sfs_read -> sfs_io -> sfs_blockio)

```
sfs_blockio(struct sfs_vnode *sv, struct uio *uio) {
    struct sfs_fs *sfs = sv->sv_absvn.vn_fs->fs_data;
    ...
    /* Get the block number within the file */
    fileblock = uio->uio_offset / SFS_BLOCKSIZE;
    /* Look up the disk block number */
    result = sfs_bmap(sv, fileblock, doalloc, &diskblock);
    if (result) {
        return result;
    }
    /* Do the I/O  */
    ...
    result = sfs_rwblock(sfs, uio);
    ...
    return result;
}
```

ELF Files

- ELF files contain address space segment descriptions, which are useful to the kernel when it is loading a new address space
- the ELF file identifies the (virtual) address of the program's first instruction
- the ELF file also contains lots of other information (e.g., section descriptors,symbol tables) that is useful to compilers, linkers, debuggers, loaders and other tools used to build programs

Address Space Segments in ELF Files

- Each ELF segment describes a contiguous region of the virtual address space.
- For each segment, the ELF file includes a segment *image* and a header, which describes:

- the virtual address of the start of the segment
- the length of the segment in the virtual address space
- the location of the start of the image in the ELF file
- the length of the image in the ELF file

- the image is an exact copy of the binary data that should be loaded into the specified portion of the virtual address space
- the image may be smaller than the address space segment, in which case the rest of the address space segment is expected to be zero-filled

To initialize an address space, the kernel copies images from the ELF file to the specifed portions of the virtual address space

ELF Files and OS/161

- OS/161's dumbvm implementation assumes that an ELF file contains two segments:

- a *text segment*, containing the program code and any read-only data
- a *data segment*, containing any other global program data

- the ELF file does not describe the stack (why not?)

- dumbvm creates a *stack segment* for each process. It is 12 pages long, ending at virtual address 0x7fffffff

Look at kern/userprog/loadelf.c to see how OS/161 loads segments from ELF files

ELF Sections and Segments

- In the ELF file, a program's code and data are grouped together into *sections*, based on their properties. Some sections:
  .text: program code
  .rodata: read-only global data
  .data: initialized global data
  .bss: uninitialized global data (Block Started by Symbol)
  .sbss: small uninitialized global data

- not all of these sections are present in every ELF file

- normally
  - the .text and .rodata sections together form the text segment
  - the .data, .bss and .sbss sections together form the data segement

- space for *local* program variables is allocated on the stack when the program runs

The segments.c Example Program (1 of 2)↰

```
#include <unistd.h>

#define N   (200)

int x = 0xdeadbeef;
int y1;
int y2;
int y3;
int array[4096];
char const *str = "Hello World\n";
const int z = 0xabcddcba;

struct example {
  int ypos;
  int xpos;
};
```

The segments.c Example Program (2 of 2)↰

```
int
main()
{
  int count = 0;
  const int value = 1;
  y1 = N;
  y2 = 2;
  count = x + y1;
  y2 = z + y2 + value;

  reboot(RB_POWEROFF);
  return 0; /* avoid compiler warnings */
}
```

ELF Sections for the Example Program

```
Section Headers:
  [Nr] Name      Type            Addr      Off     Size    ES  Flg
  [ 0]           NULL            00000000  000000  000000  00
  [ 1] .reginfo  MIPS_REGINFO    00400094  000094  000018  18    A
  [ 2] .text     PROGBITS        004000b0  0000b0  000200  00   AX
  [ 3] .rodata   PROGBITS        004002b0  0002b0  000020  00    A
  [ 4] .data     PROGBITS        10000000  001000  000010  00   WA
  [ 5] .sbss     NOBITS          10000010  001010  000014  00  WAp
  [ 6] .bss      NOBITS          10000030  00101c  004000  00   WA
  [ 7] .comment  PROGBITS        00000000  00101c  000036  00
...
Flags: W (write), A (alloc), X (execute), p (processor specific)

## Size = number of bytes (e.g., .text is 0x200 = 512 bytes
## Off  = offset into the ELF file
## Addr = virtual address
```

The mips-harvard-os161-readelf program can be used to inspect OS161 MIPS ELF files: mips-harvard-os161-readelf -a segments

ELF Segments for the Example Program

```
Program Headers:
Type    Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
REGINFO 0x000094 0x00400094 0x00400094 0x00018 0x00018 R   0x4
LOAD    0x000000 0x00400000 0x00400000 0x002d0 0x002d0 R E 0x1000
LOAD    0x001000 0x10000000 0x10000000 0x00010 0x04030 RW  0x1000
```

- segment info, like section info, can be inspected using the cs350-readelf program
- the REGINFO section is not used
- the first LOAD segment includes the .text and .rodata sections
- the second LOAD segment includes .data, .sbss, and .bss

Contents of the Example Program's .text Section

```
Contents of section .text:
 4000b0 3c1c1001 279c8000 3c08ffff 3508fff8 <...'...<...5...
...
## Decoding 3c1c1001 to determine instruction
## 0x3c1c1001 = binary 11110000011100000100000000001
## 0011 1100 0001 1100 0001 0000 0000 0001
## instr  | rs    | rt    | immediate
## 6 bits | 5 bits| 5 bits| 16 bits
## 001111 | 00000 | 11100 | 0001 0000 0000 0001
## LUI    | 0     | reg 28| 0x1001
## LUI    | unused| reg 28| 0x1001
## Load unsigned immediate into rt (register target)¶
## lui gp, 0x1001
```

The mips-harvard-os161-objdump program can be used to inspect OS161 MIPSELF file section contents:

```
mips-harvard-os161-objdump -s segments
```

Contents of the Example Program's .rodata Section

```
Contents of section .rodata:
 4002b0 48656c6c 6f20576f 726c640a 00000000  Hello World.....
 4002c0 abcddcba 00000000 00000000 00000000  ................
...
## 0x48 = 'H' 0x65 = 'e' 0x0a = '\n' 0x00 = '\0'
## Align next int to 4 byte boundary
## const int z = 0xabcddcba
## If compiler doesn't prevent z from being written,
##  then the hardware could
## Size = 0x20 = 32 bytes "Hello World\n\0" = 13 + 3 padding = 16
##               + const int z = 4 = 20
## Then align to the next 16 byte boundry at 32 bytes.
```

The .rodata section contains the "Hello World" string literal and the constant integer variable z.

Contents of the Example Program's .data Section

```
Contents of section .data:
 10000000 deadbeef 004002b0 00000000 00000000 .....@.....
```

```
...
## Size = 0x10 bytes = 16 bytes
## int x = deadbeef (4 bytes)↰
## char const *str = "Hello World\n"; (4 bytes)↰
## value stored in str = 0x004002b0.
## NOTE: this is the address of the start
## of the string literal in the .rodata section
```

The .data section contains the initialized global variables str and x.

Contents of the Example Program's .bss and .sbss Sections

```
...
10000010 A __bss_start
10000010 A _edata
10000010 A _fbss
10000010 S y3   ## S indicates sbss section
10000014 S y2
10000018 S y1
1000001c S errno
10000020 S __argv
...
10000030 B array  ## B indicates bss section
10004030 A _end
```

The y1, y2, and y3 variables are in the .sbss section. The array variable is in the .bss section. There are no values for these variables in the ELF file, as they are uninitialized. The mips-harvard-os161-nm program can be used to inspect symbols defined in ELF files: mips-harvard-os161-nm -b segments

System Calls

- Programming interface, for user programs, to the services offered by the system (Silberschatz, Galvin, Gagne ch. 2)

- User program view: an API (calls to functions):

```
int open(const char *path, int oflag, ...);
ssize_t read(int fd, void *buf, size_t nbyte);
ssize_t write(int fd, const void *buf, size_t nbyte);
int close(int fildes);
```

- Kernel view: software interrupt

```
void syscall(struct trapframe *tf) {
    int callno = tf->tf_v0;
    switch (callno) {
       ...
    }
}
```

syscall()
(kern/arch/mips/syscall/syscall.c)

```
void syscall(struct trapframe *tf) {
    int callno = tf->tf_v0;
    int32_t retval = 0;
    int err;

    switch (callno) {
        case SYS_reboot:
        err = sys_reboot(tf->tf_a0);
        break;

        case SYS___time:
        err = sys___time((userptr_t)tf->tf_a0,
                  (userptr_t)tf->tf_a1);
        break;

        /* Add stuff here */
```

syscall()
(kern/arch/mips/syscall/syscall.c)

```
switch (callno) {
        ...
        /* Add stuff here */

        default:
        kprintf("Unknown syscall %d\n", callno);
        err = ENOSYS;
        break;
}
if (err) {
        tf->tf_v0 = err;
        tf->tf_a3 = 1;      /* signal an error */
} else {
        /* Success. */
        tf->tf_v0 = retval;
        tf->tf_a3 = 0;      /* signal no error */
}
```

Adding a new call

```
void syscall(struct trapframe *tf) {

    switch (callno) {
        case SYS_reboot:
        err = sys_reboot(tf->tf_a0);
        break;
        /* Add stuff here */
    case SYS_read:
        retval = sys_read((int)tf->tf_a0,
                  (userptr_t)tf->tf_a1,
                  (int)tf->tf_a2);
        if (retval<0) {
          err = 1;
        }
        else err = 0;
        break;
```

Adding a new call

```
...
case SYS_write:
     retval = sys_write((int)tf->tf_a0,
                        (userptr_t)tf->tf_a1,
                        (int)tf->tf_a2);
     if (retval<0) {
       err = 1;
     }
     else err = 0;
     break;
case SYS__exit:
     sys__exit((int)tf->tf_a0)
     break;
...
```

sys_read()

```
// partial/temporary: just 1 byte from stdin

int sys_read(int fd, userptr_t buf, int nbyte) {
  if (fd == STDIN_FILENO && nbyte == 1) {
    char localBuf[2];
    kgets(localBuf,1);
    *(char *)buf = localBuf[0];
    return 1;
  }
  else {
    kprintf("sys_read implemented only on stdio (1 char)
\n");
    return -1;
  }
}
```

- \

sys_read()

```
// partial/temporary: just 1 byte from stdin (with getch)

int sys_read(int fd, userptr_t buf, int size) {
  if (fd == STDIN_FILENO && nbyte == 1) {
    *(char *)buf = getch();
    return 1;
  }
  else {
    kprintf(”sys_read implemented only on stdio (1 char)\n”);
    return -1;
  }
}
```

- \

sys_read()

```
// partial/temporary: just from stdin

int sys_read(int fd, userptr_t buf, int size) {
  if (fd == STDIN_FILENO) {
    char *p = (char *)buf;
    int i;
    for (i=0; i<size; i++) {
      int c = getch();
      if (c<0) return i; /* EOF */
      p[i] = (char)c;
    }
    return size;
  } else {
    kprintf("not yet implemented\n");
    return -1;
  }
}
```

sys_write()

```
// partial/temporary: just 1 byte to stdout

int sys_write(int fd, userptr_t buf, int size) {
  if (fd == STDOUT_FILENO && size == 1) {
    return kprintf("%c", ((char *)buf)[0]);
  }
  else {
    kprintf(”not yet implemented\n”);
    return -1;
  }
}
```

sys_write()

```
// partial/temporary: just 1 byte to stdout (putch)

int sys_write(int fd, userptr_t buf, int size) {
  if (fd == STDOUT_FILENO && size == 1) {
    putch(((char *)buf)[0]);
  }
  else {
    kprintf(”not yet implemented\n”);
    return -1;
  }
}
```

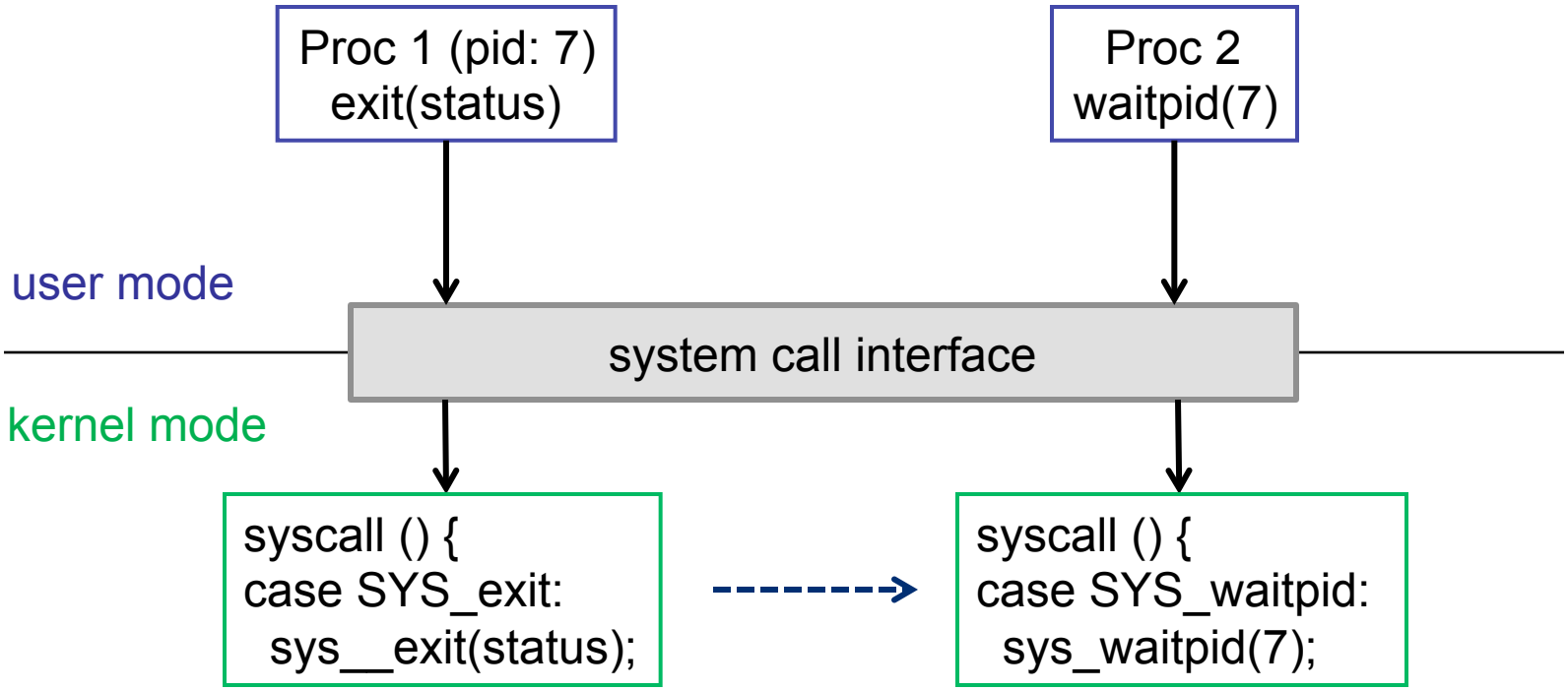
sys_write()

```
// partial/temporary: just to stdout

int sys_write(int fd, userptr_t buf, int size) {
  if (fd == STDOUT_FILENO) {
    int i;
    for (i=0; i<size; i++) {
      putch((char *)buf)[i]);
    }
    return size;
  }
  else {
    kprintf(”not yet implemented\n”);
    return -1;
  }
}
```

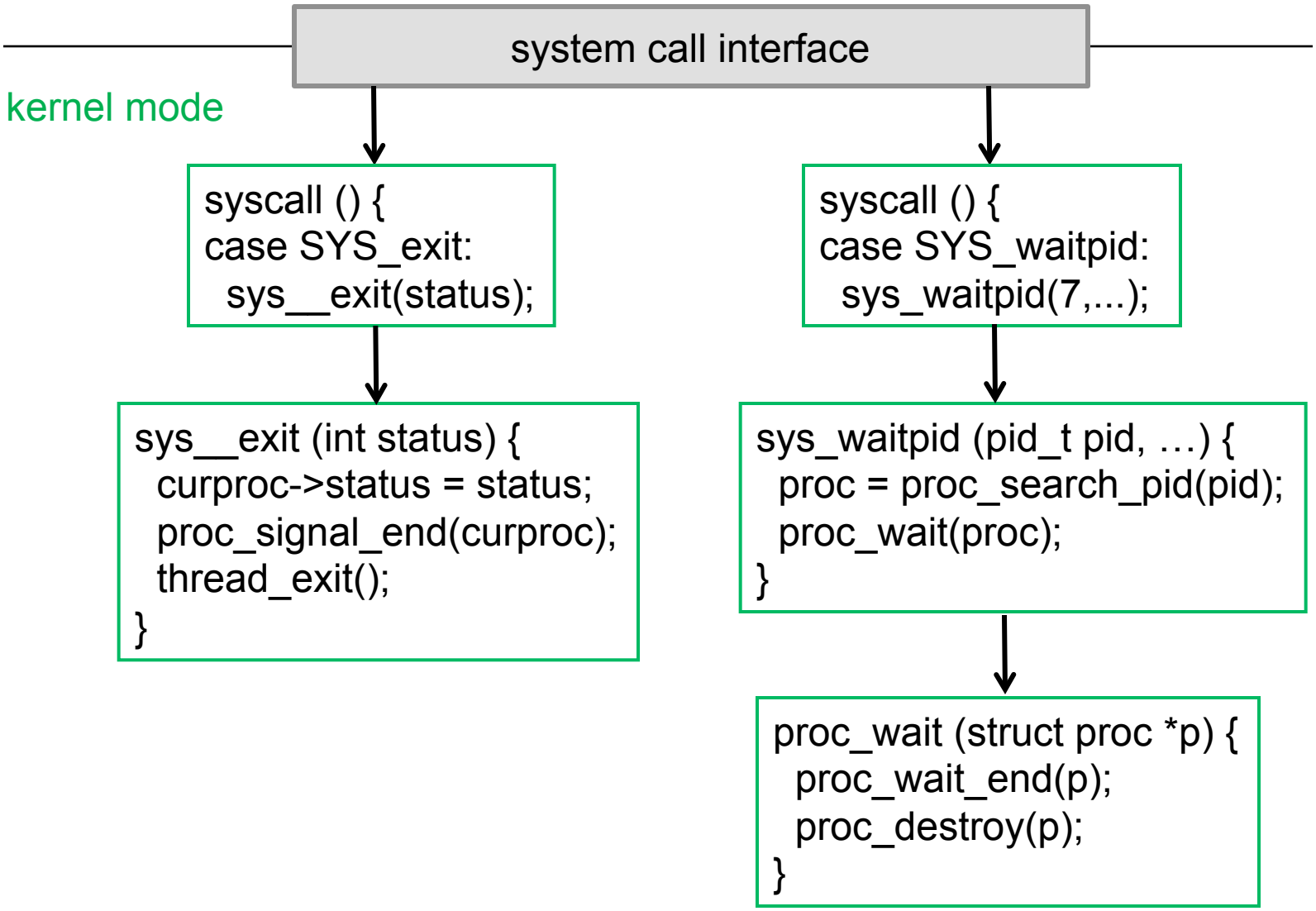
sys_exit()

```
// partial/temporary: (1) record status, (2) release
address space, (3) exit thread

int sys__exit(int status) {

  curproc->exit_status = status;
  as_destroy(curproc->p_addrspace);
  thread_exit();

}
```

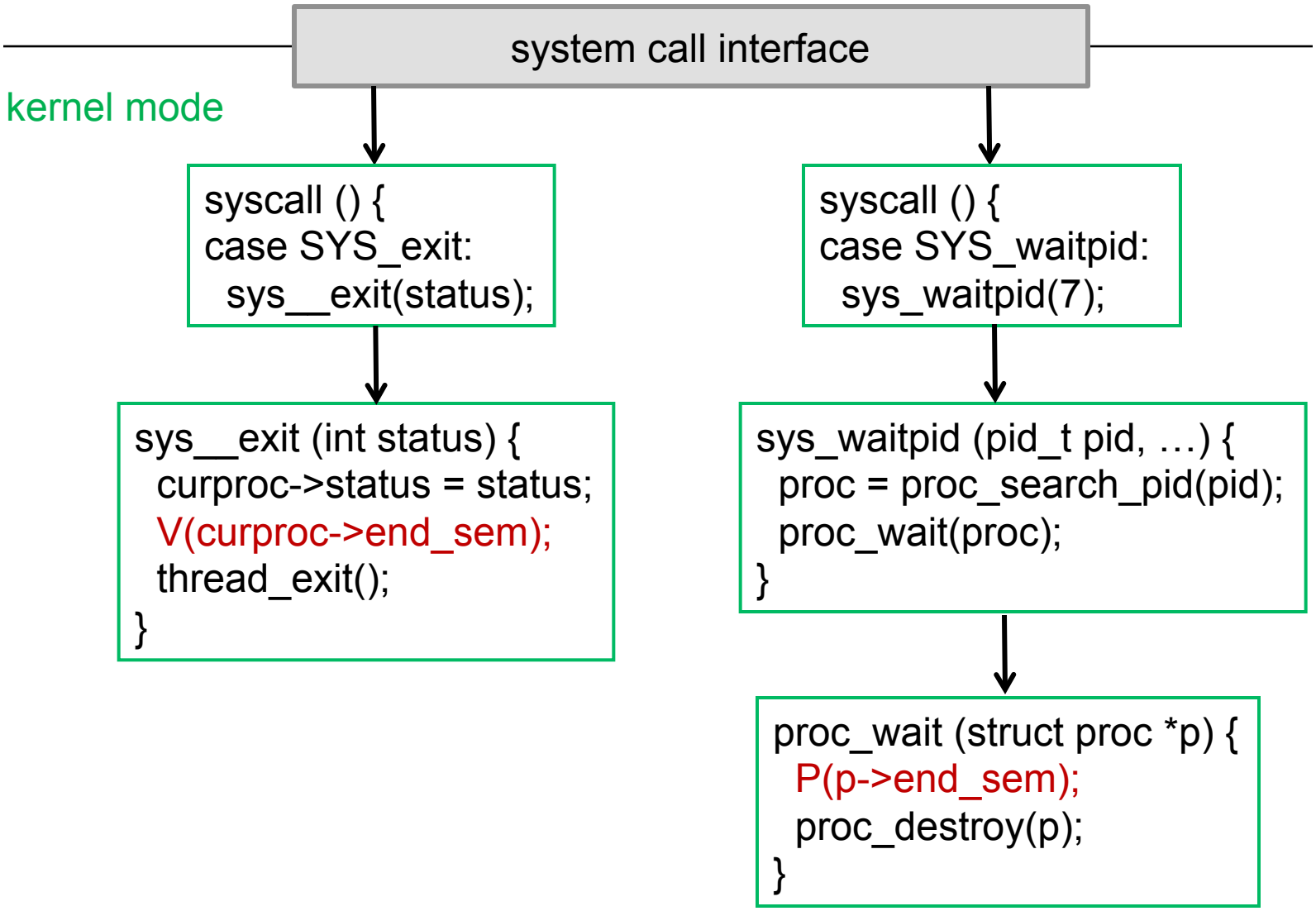
sys_exit (with synch)



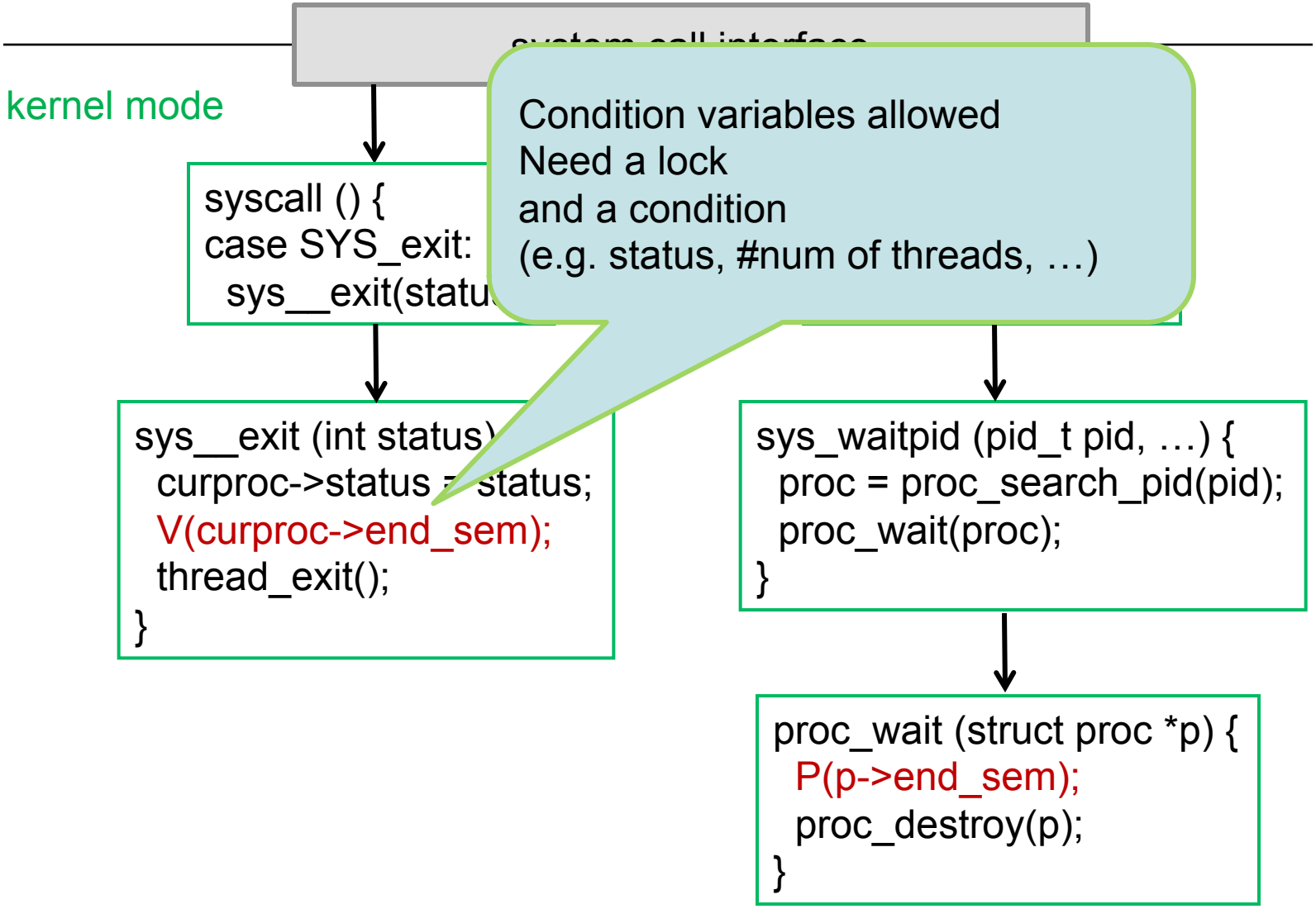
sys_exit (with synch)



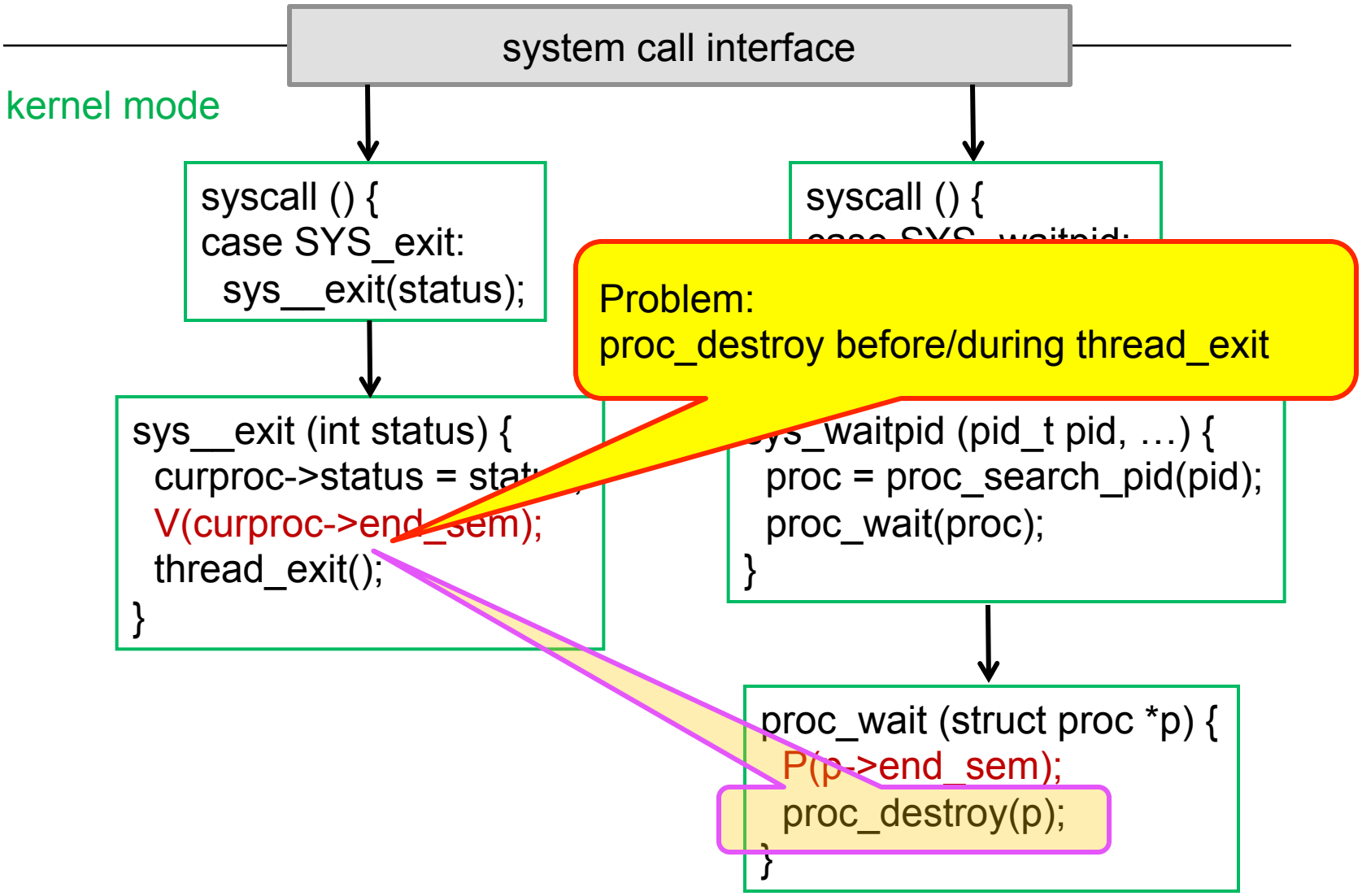
sys_exit (with semaphores)



sys_exit (with semaphores)



sys_exit (with synch)



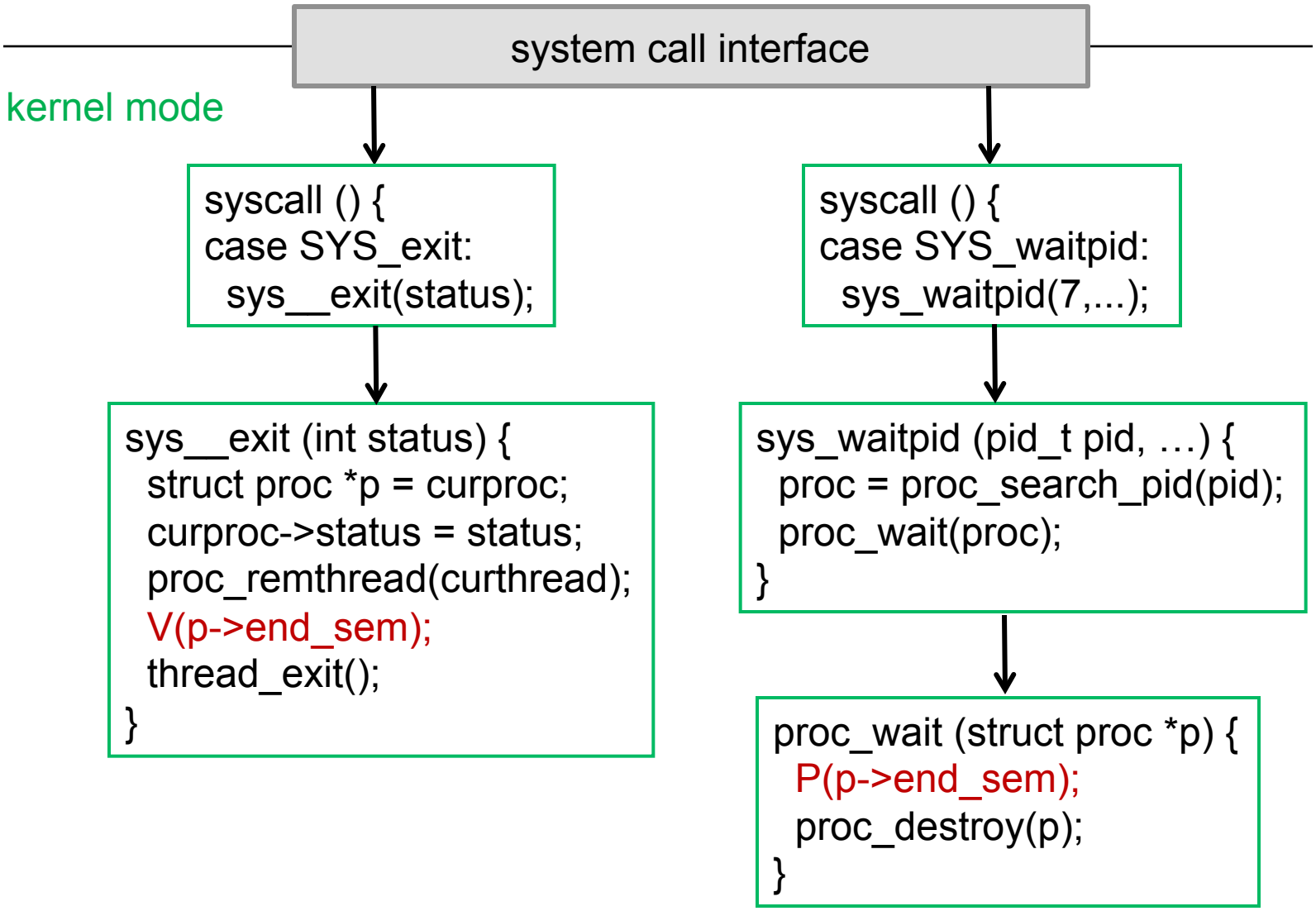
proc_destroy and thread_exit

- Need to do `proc_destroy` after `thread_exit` detaches thread from proc

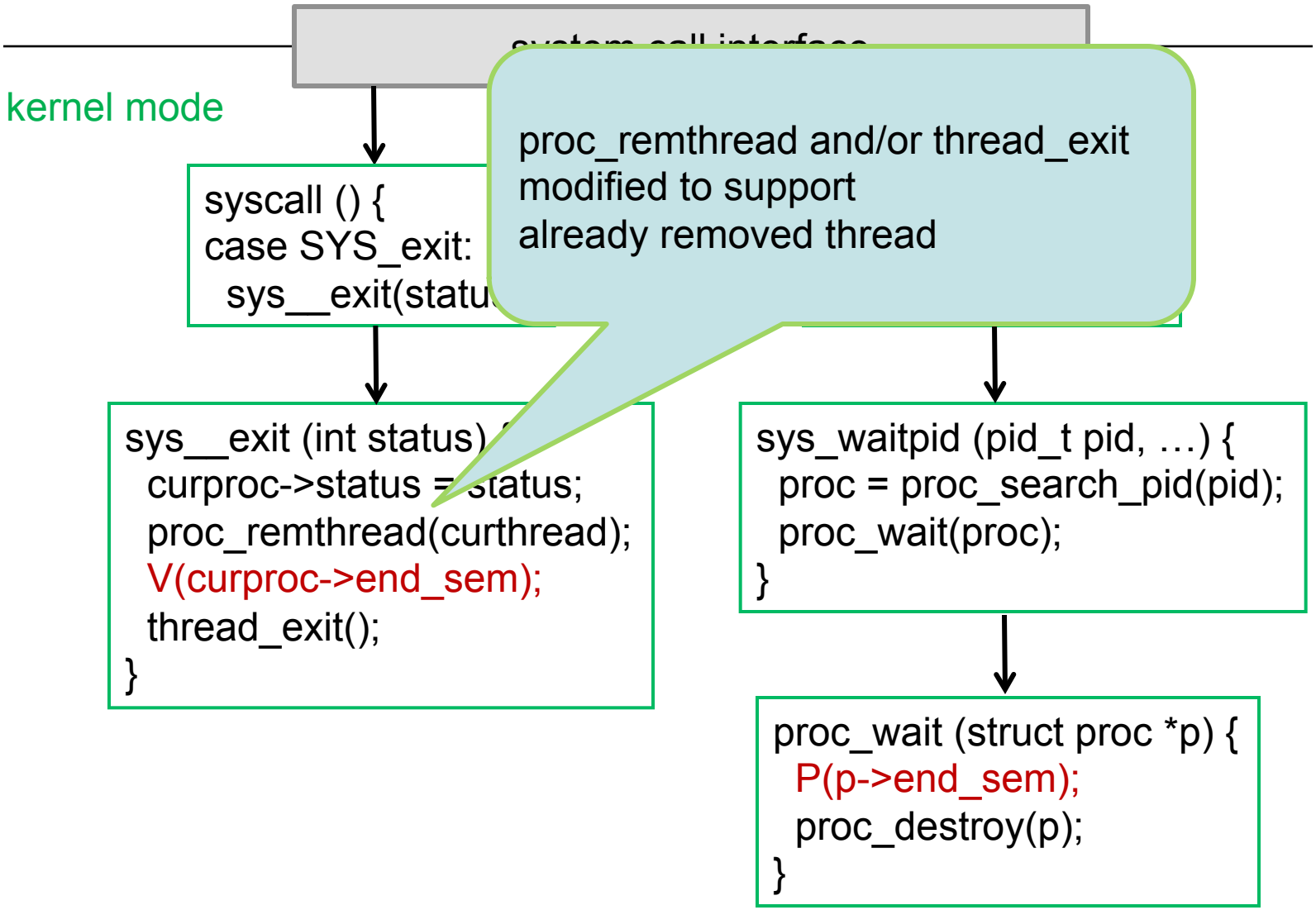
  `proc->p_numthreads == 0`

- Possible solutions
  - Sleep and/or polling before calling `proc_destroy`
  - Detach thread before signal. Modify `thread_exit` or `proc_remthread` (detach a thread only if still attached to proc)

sys_exit (with synch)



sys_exit (with synch)



proc_search_pid

- Need array to support conversion from pid to pointer
- Array in proc.c
- Direct access:
  - `struct proc *proc[PID_MAX+1];`
- If limit on max number of allowed processes MAXPROC << PID_MAX
  - `struct proc *proc[MAXPROC];`
- Alternative (kmalloc)