EE516 : Project 2

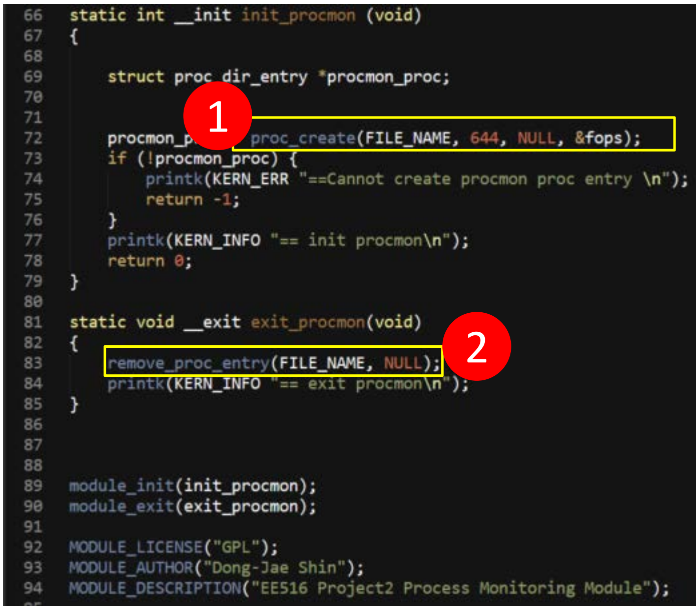
Gaurav Kalra

(Student ID: 2016 45 93)

gvkalra@kaist.ac.kr

Task 1: Analyze the given skeleton program

Find the functions that you do not know and explain the operations of those functions



```c
static int __init init_procmon (void)
{

    struct proc_dir_entry *procmon_proc;

    procmon_p  proc_create(FILE_NAME, 644, NULL, &fops);
    if (!procmon_proc) {
        printk(KERN_ERR "==Cannot create procmon proc entry \n");
        return -1;
    }
    printk(KERN_INFO "== init procmon\n");
    return 0;
}

static void __exit exit_procmon(void)
{
    remove_proc_entry(FILE_NAME, NULL);
    printk(KERN_INFO "== exit procmon\n");
}


module_init(init_procmon);
module_exit(exit_procmon);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dong-Jae Shin");
MODULE_DESCRIPTION("EE516 Project2 Process Monitoring Module");
```

proc_create(): creates a file entry in /proc directory

parameters:

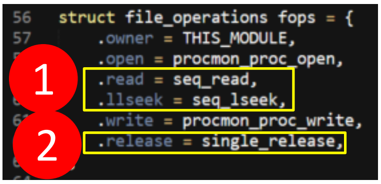
1. **name** – filename which will show up in proc directory
2. **mode** – file creation mode (w,r,x for user,group,others)
3. **parent directory** – pointer to parent directory in /proc where this file will be created (we used NULL to indicate /proc/ directory)
4. **file operations** – pointer to file operations structure

remove_proc_entry(): remove the file entry from /proc directory

parameters:

1. **name** – filename as in /proc
2. **parent** – parent dir pointer (same NULL here)

Figure 1: Creating / Removing a file from procfs



```c
struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = procmon_proc_open,
    .read = seq_read,
    .llseek = seq_lseek,
    .write = procmon_proc_write,
    .release = single_release,
```

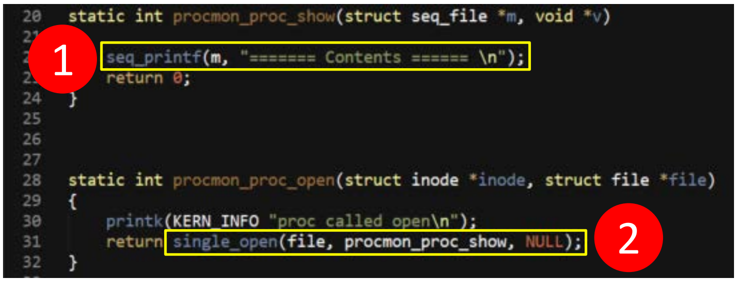
seq_read(): read method for sequential files

seq_lseek(): llseek method for sequential files

single_release(): free the structures associated with sequential files

The parameters for these APIs are compatible with respective VFS file operations. And thus can be used as drop-in replacement so that we do not need to write our own implementation.

Figure 2: Sequence File APIs for VFS file operations



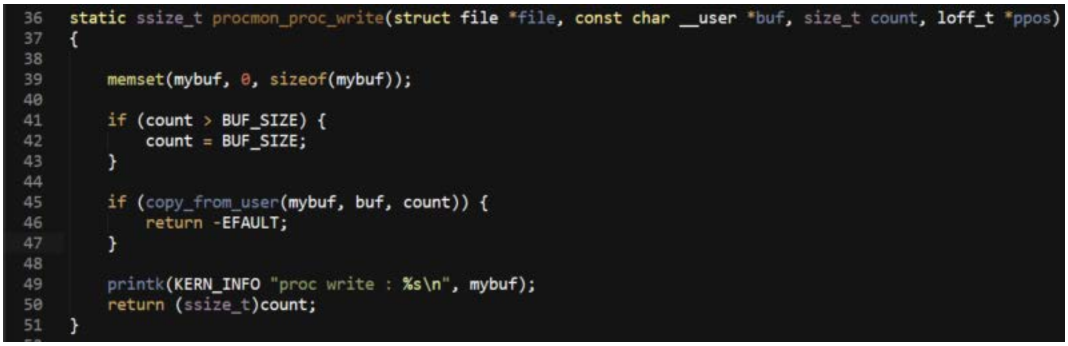
```c
static int procmon_proc_show(struct seq_file *m, void *v)

    seq_printf(m, "======= Contents ====== \n");
    return 0;
}



static int procmon_proc_open(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "proc called open\n");
    return single_open(file, procmon_proc_show, NULL);
}
```

seq_printf(): It works like printk() for proc files. The first argument must be the sequence file.

single_open(): It creates and opens a new file descriptor for sequence files.

parameters:

1. **file** – The file structure from VFS
2. **show** – Call-back for printing data in proc filesystem
3. **data** – Call-back data (NULL in our case)

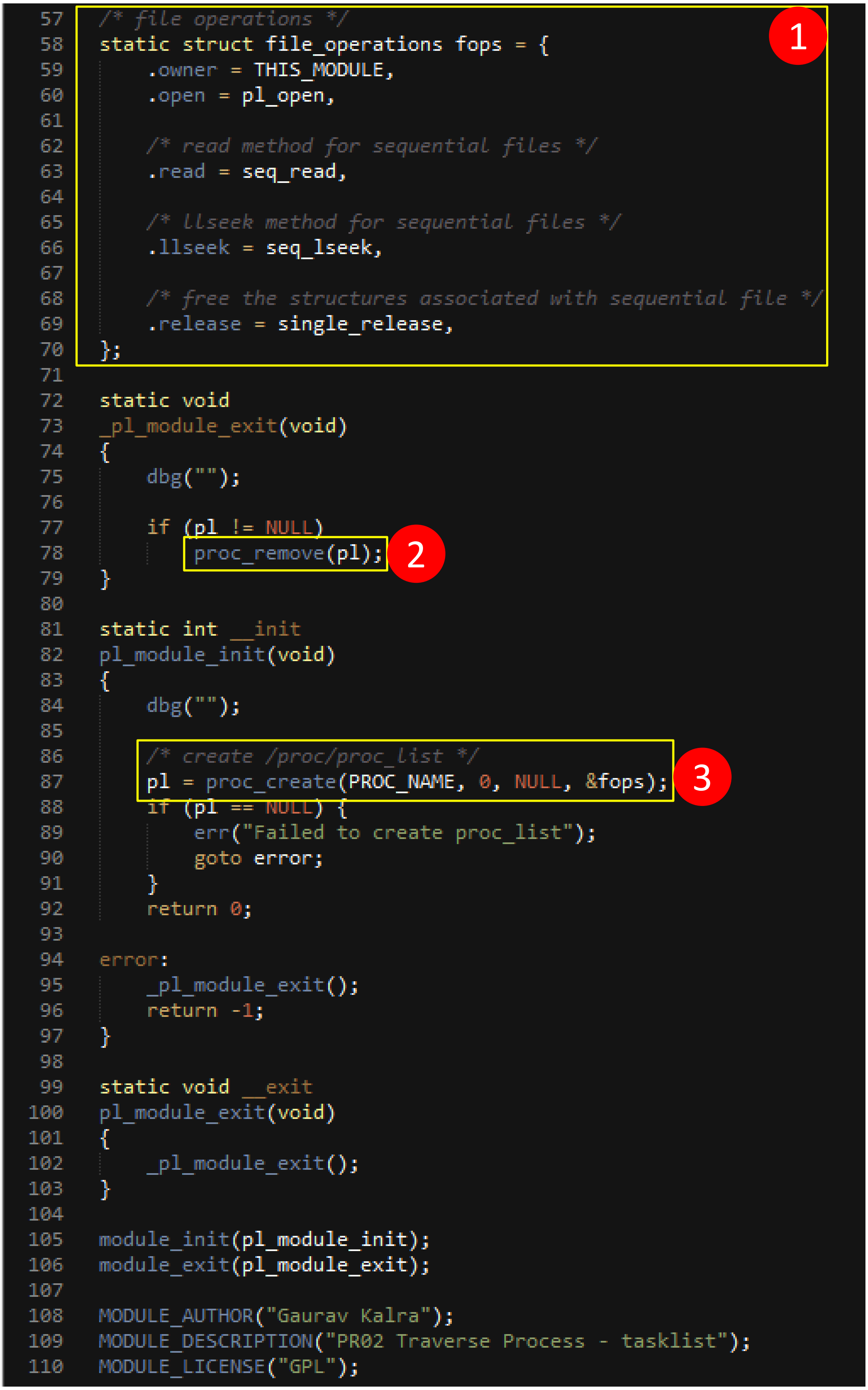


```c
static ssize_t procmon_proc_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
{

    memset(mybuf, 0, sizeof(mybuf));

    if (count > BUF_SIZE) {
        count = BUF_SIZE;
    }

    if (copy_from_user(mybuf, buf, count)) {
        return -EFAULT;
    }

    printk(KERN_INFO "proc write : %s\n", mybuf);
    return (ssize_t)count;
}
```

Figure 3: Using Sequence Files for displaying contents of procfs

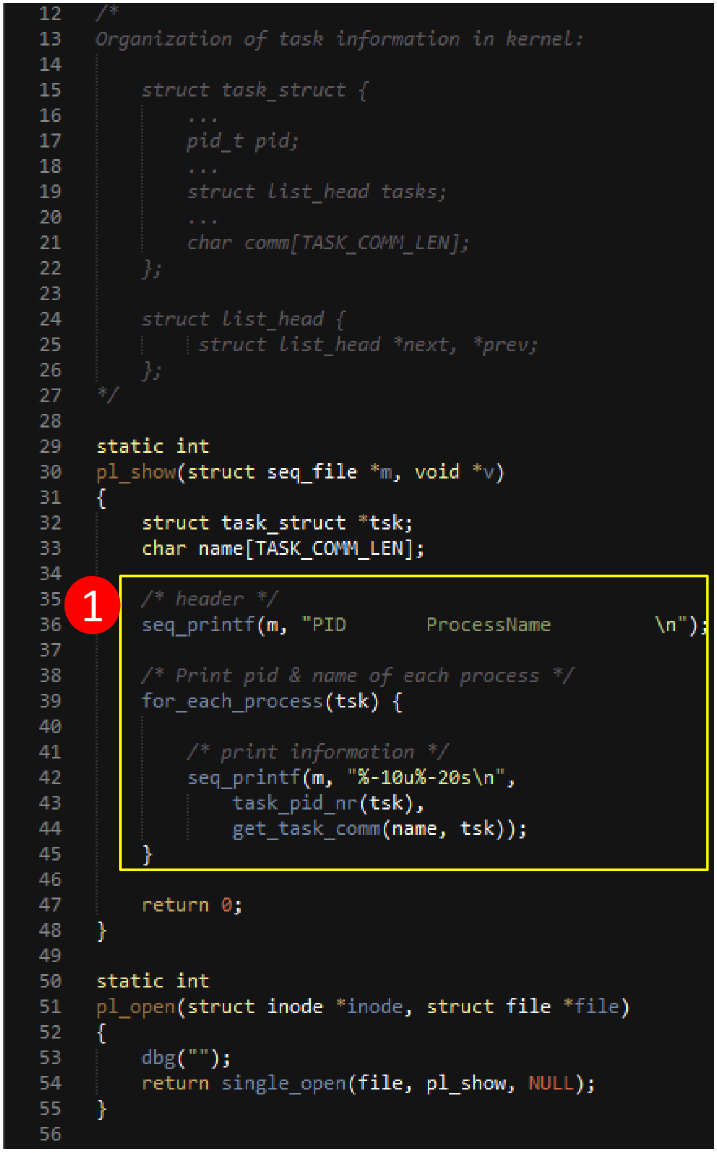
Task 2: Traverse Process - tasklist

Print every task’s **PID** and **Process Name** in your proc file system



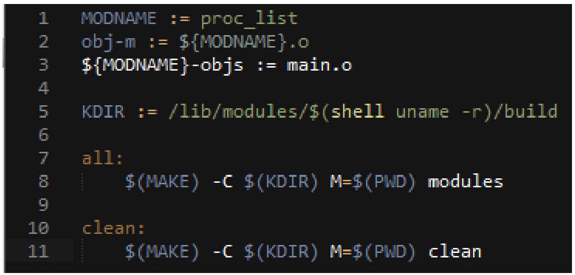
```
/* file operations */
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = pl_open,

    /* read method for sequential files */
    .read = seq_read,

    /* llseek method for sequential files */
    .llseek = seq_lseek,

    /* free the structures associated with sequential file */
    .release = single_release,
};

static void
_pl_module_exit(void)
{
    dbg("");

    if (pl != NULL)
        proc_remove(pl);
}

static int __init
pl_module_init(void)
{
    dbg("");

    /* create /proc/proc_list */
    pl = proc_create(PROC_NAME, 0, NULL, &fops);
    if (pl == NULL) {
        err("Failed to create proc_list");
        goto error;
    }
    return 0;

error:
    _pl_module_exit();
    return -1;
}

static void __exit
pl_module_exit(void)
{
    _pl_module_exit();
}

module_init(pl_module_init);
module_exit(pl_module_exit);

MODULE_AUTHOR("Gaurav Kalra");
MODULE_DESCRIPTION("PR02 Traverse Process - tasklist");
MODULE_LICENSE("GPL");
```

Figure 4: Module Initialization / Clean-up (refer task01 for details)

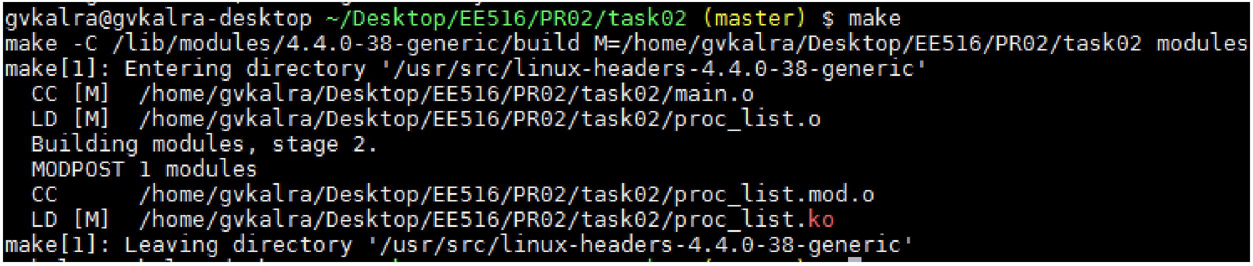


```
/*
Organization of task information in kernel:

    struct task_struct {
        ...
        pid_t pid;
        ...
        struct list_head tasks;
        ...
        char comm[TASK_COMM_LEN];
    };

    struct list_head {
        struct list_head *next, *prev;
    };
*/

static int
pl_show(struct seq_file *m, void *v)
{
    struct task_struct *tsk;
    char name[TASK_COMM_LEN];

    /* header */
    seq_printf(m, "PID       ProcessName         \n");

    /* Print pid & name of each process */
    for_each_process(tsk) {

        /* print information */
        seq_printf(m, "%-10u%-20s\n",
            task_pid_nr(tsk),
            get_task_comm(name, tsk));
    }

    return 0;
}

static int
pl_open(struct inode *inode, struct file *file)
{
    dbg("");
    return single_open(file, pl_show, NULL);
}

```

Figure 5: Logic for printing PID & Process Name

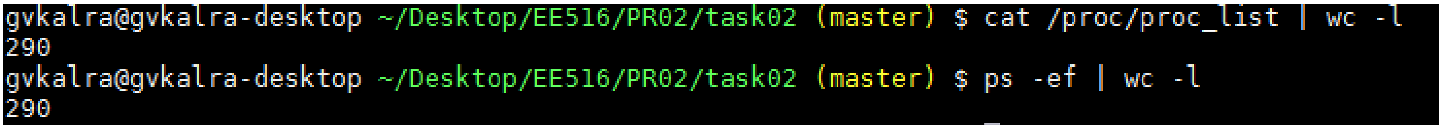


```
MODNAME := proc_list
obj-m := ${MODNAME}.o
${MODNAME}-objs := main.o

KDIR := /lib/modules/$(shell uname -r)/build

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
```



```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task02 (master) $ make
make -C /lib/modules/4.4.0-38-generic/build M=/home/gvkalra/Desktop/EE516/PR02/task02 modules
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-38-generic'
  CC [M]  /home/gvkalra/Desktop/EE516/PR02/task02/main.o
  LD [M]  /home/gvkalra/Desktop/EE516/PR02/task02/proc_list.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/gvkalra/Desktop/EE516/PR02/task02/proc_list.mod.o
  LD [M]  /home/gvkalra/Desktop/EE516/PR02/task02/proc_list.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-38-generic'
```

Figure 6: Makefile (no warnings)

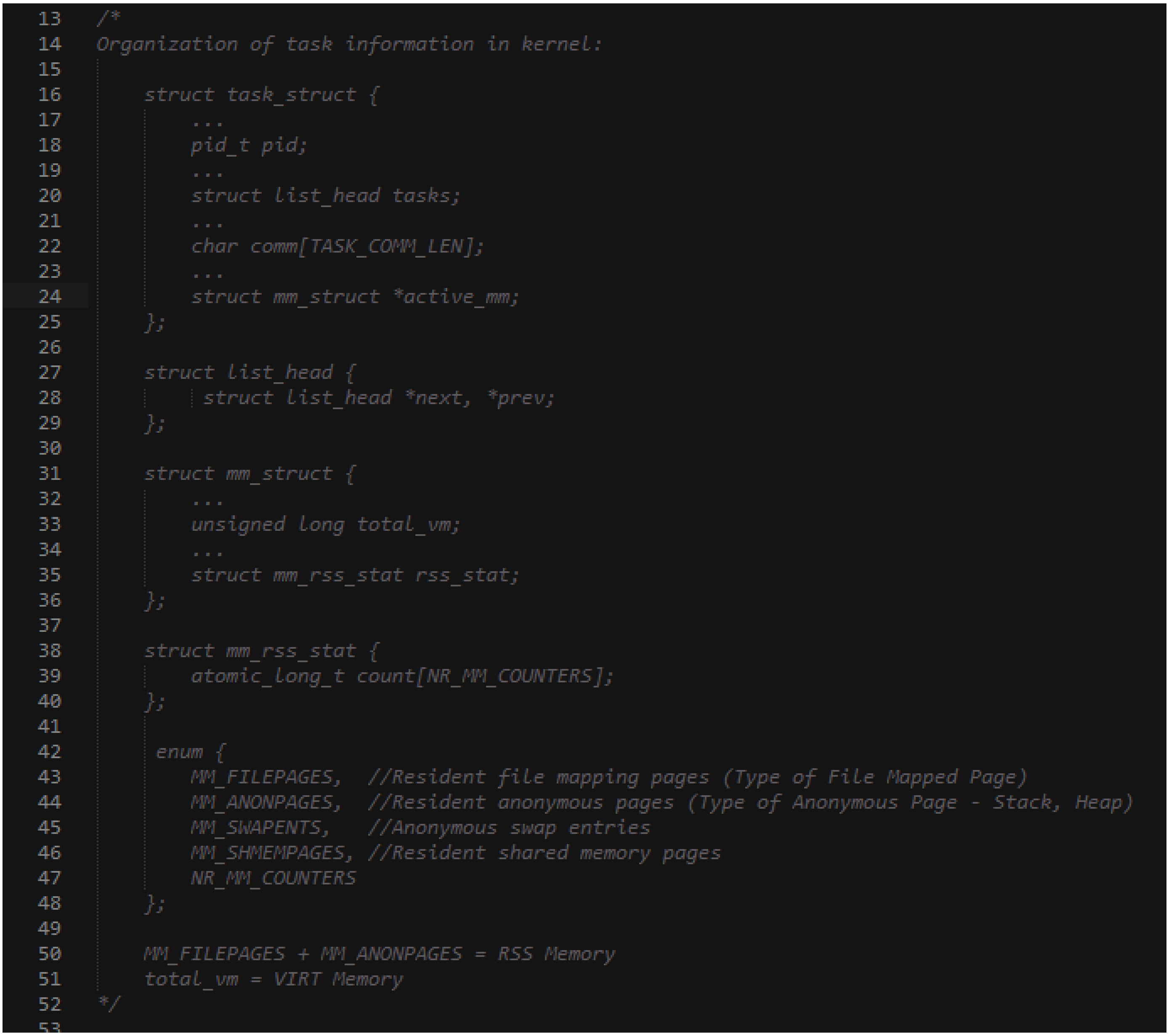


```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task02 (master) $ cat /proc/proc_list | wc -l
290
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task02 (master) $ ps -ef | wc -l
290
```

Figure 7: Verification (**proc_list & ps -ef both output 290 processes**)

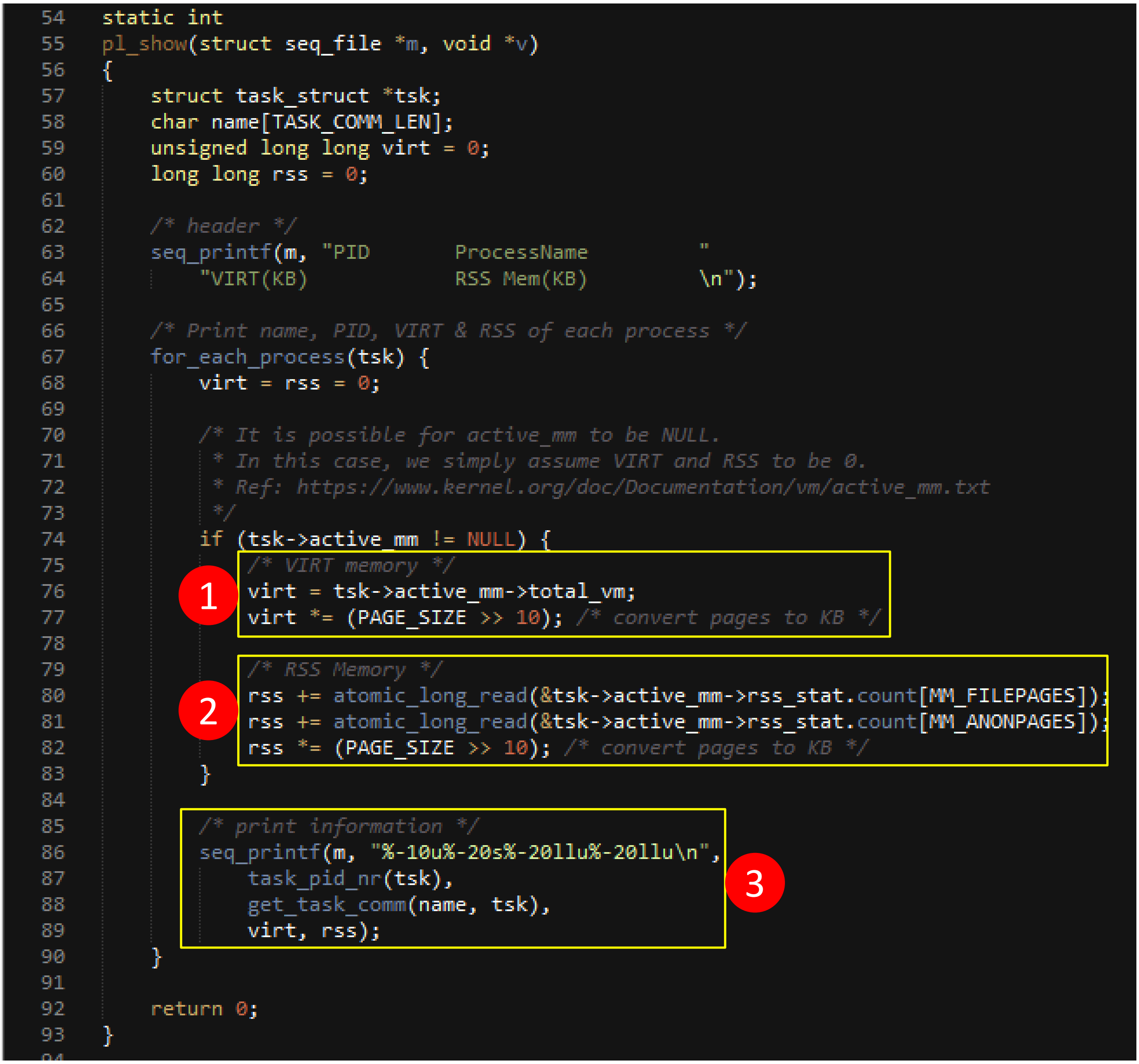
Task 3: per Process Memory Usage

Print every task’s **VIRT** and **RSS Memory** Size



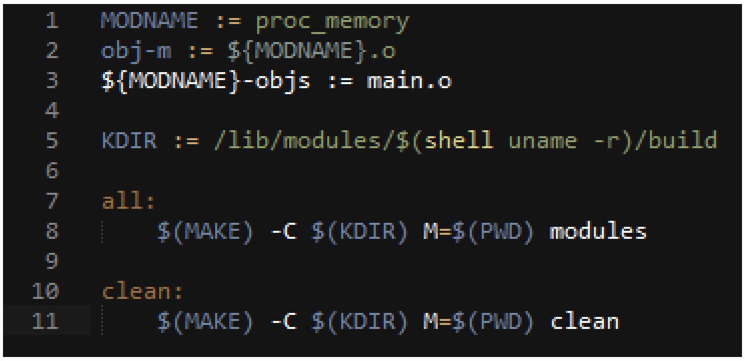
```
/*
Organization of task information in kernel:

    struct task_struct {
        ...
        pid_t pid;
        ...
        struct list_head tasks;
        ...
        char comm[TASK_COMM_LEN];
        ...
        struct mm_struct *active_mm;
    };

    struct list_head {
         struct list_head *next, *prev;
    };

    struct mm_struct {
        ...
        unsigned long total_vm;
        ...
        struct mm_rss_stat rss_stat;
    };

    struct mm_rss_stat {
        atomic_long_t count[NR_MM_COUNTERS];
    };

     enum {
        MM_FILEPAGES,  //Resident file mapping pages (Type of File Mapped Page)
        MM_ANONPAGES,  //Resident anonymous pages (Type of Anonymous Page - Stack, Heap)
        MM_SWAPENTS,   //Anonymous swap entries
        MM_SHMEMPAGES, //Resident shared memory pages
        NR_MM_COUNTERS
    };

    MM_FILEPAGES + MM_ANONPAGES = RSS Memory
    total_vm = VIRT Memory
*/
```

Figure 8: Organization of **VIRT** & **RSS** in task_struct



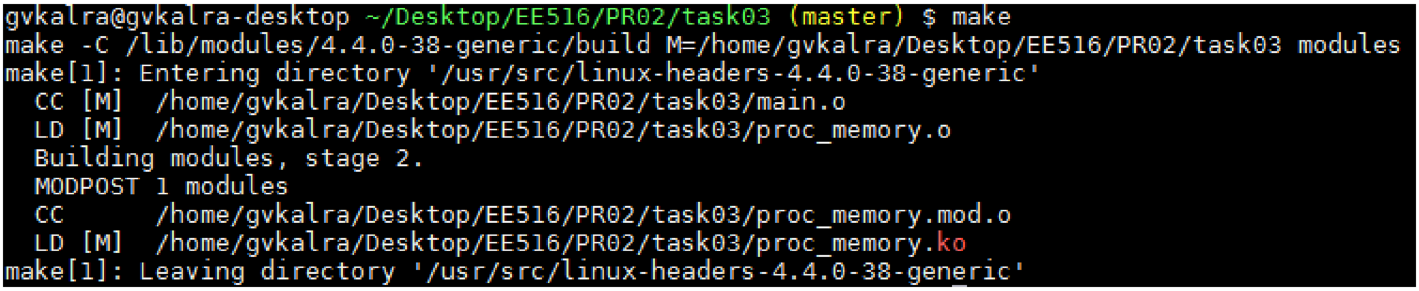
```c
static int
pl_show(struct seq_file *m, void *v)
{
    struct task_struct *tsk;
    char name[TASK_COMM_LEN];
    unsigned long long virt = 0;
    long long rss = 0;

    /* header */
    seq_printf(m, "PID      ProcessName         "
        "VIRT(KB)            RSS Mem(KB)         \n");

    /* Print name, PID, VIRT & RSS of each process */
    for_each_process(tsk) {
        virt = rss = 0;

        /* It is possible for active_mm to be NULL.
         * In this case, we simply assume VIRT and RSS to be 0.
         * Ref: https://www.kernel.org/doc/Documentation/vm/active_mm.txt
         */
        if (tsk->active_mm != NULL) {
            /* VIRT memory */
            virt = tsk->active_mm->total_vm;
            virt *= (PAGE_SIZE >> 10); /* convert pages to KB */

            /* RSS Memory */
            rss += atomic_long_read(&tsk->active_mm->rss_stat.count[MM_FILEPAGES]);
            rss += atomic_long_read(&tsk->active_mm->rss_stat.count[MM_ANONPAGES]);
            rss *= (PAGE_SIZE >> 10); /* convert pages to KB */
        }

        /* print information */
        seq_printf(m, "%-10u%-20s%-20llu%-20llu\n",
            task_pid_nr(tsk),
            get_task_comm(name, tsk),
            virt, rss);
    }

    return 0;
}
```

Figure 9: Logic for calculating and printing VIRT, RSS

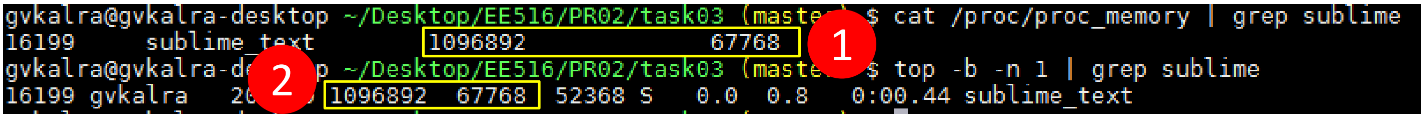


```
MODNAME := proc_memory
obj-m := ${MODNAME}.o
${MODNAME}-objs := main.o

KDIR := /lib/modules/$(shell uname -r)/build

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Figure 10: Makefile (no warnings)

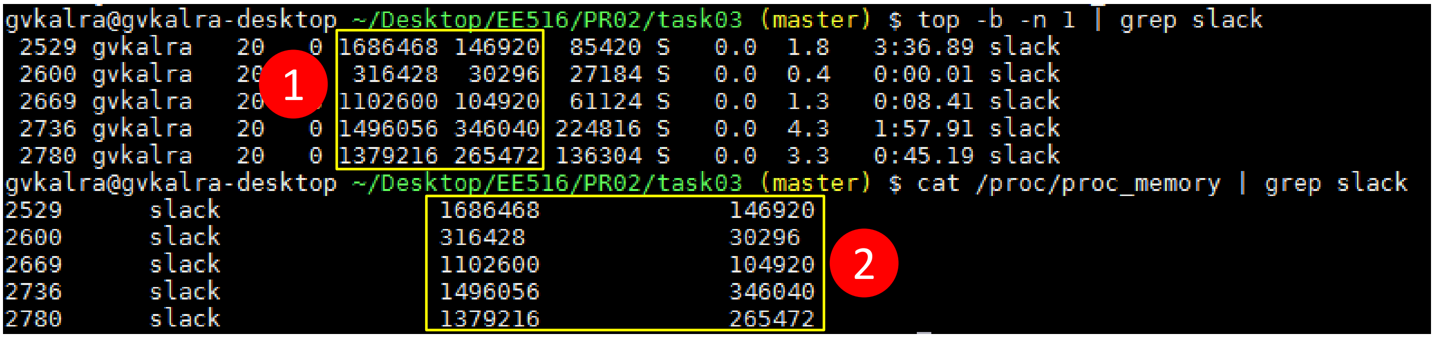


```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task03 (master) $ make
make -C /lib/modules/4.4.0-38-generic/build M=/home/gvkalra/Desktop/EE516/PR02/task03 modules
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-38-generic'
  CC [M]  /home/gvkalra/Desktop/EE516/PR02/task03/main.o
  LD [M]  /home/gvkalra/Desktop/EE516/PR02/task03/proc_memory.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/gvkalra/Desktop/EE516/PR02/task03/proc_memory.mod.o
  LD [M]  /home/gvkalra/Desktop/EE516/PR02/task03/proc_memory.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-38-generic'
```



```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task03 (master) $ cat /proc/proc_memory | grep sublime
16199      sublime_text          1096892                 67768
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task03 (master) $ top -b -n 1 | grep sublime
16199 gvkalra   20   0 1096892  67768  52368 S   0.0  0.8   0:00.44 sublime_text
```

Figure 11: Verification (sublime_text)

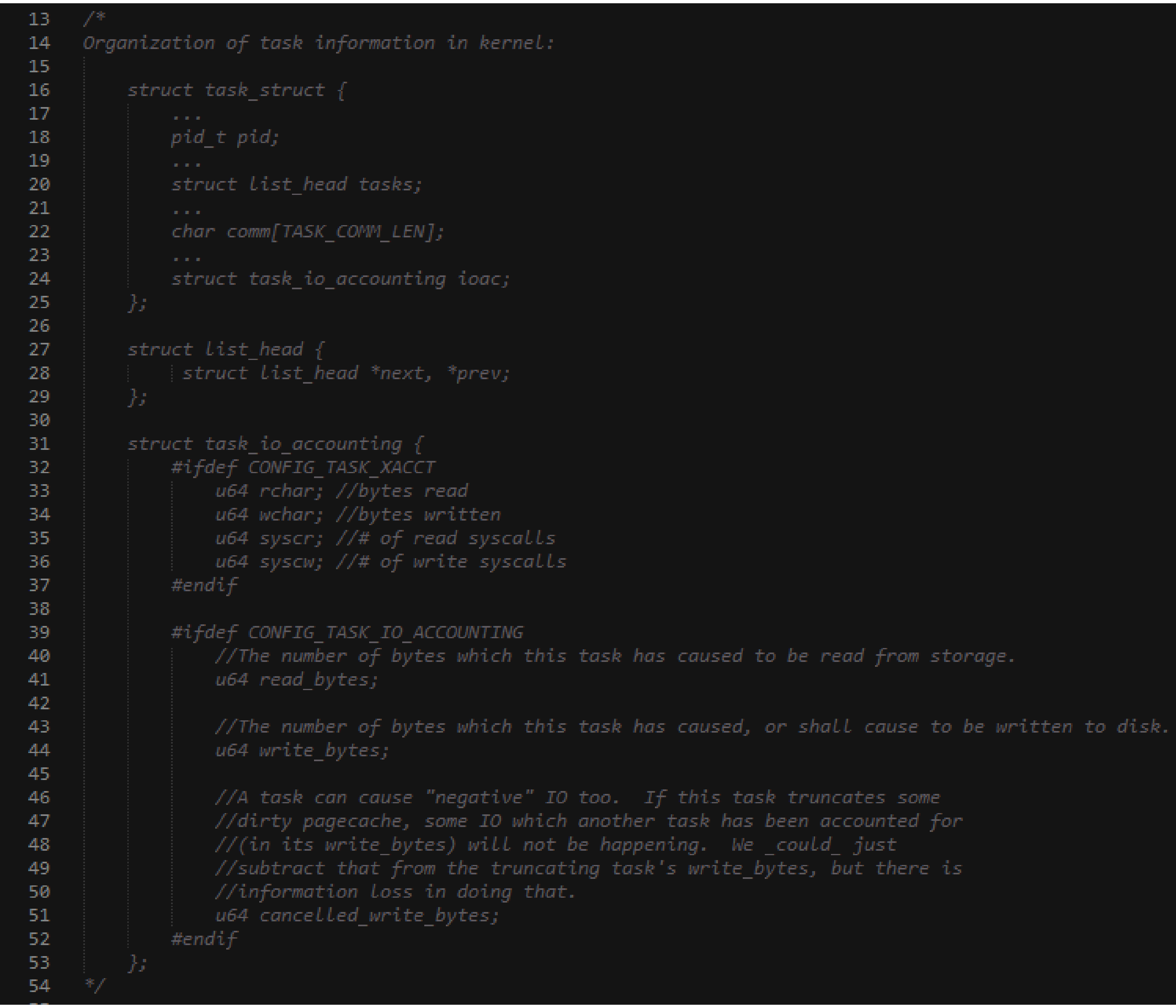


```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task03 (master) $ top -b -n 1 | grep slack
 2529 gvkalra   20   0 1686468 146920  85420 S   0.0  1.8   3:36.89 slack
 2600 gvkalra   20   0  316428  30296  27184 S   0.0  0.4   0:00.01 slack
 2669 gvkalra   20   0 1102600 104920  61124 S   0.0  1.3   0:08.41 slack
 2736 gvkalra   20   0 1496056 346040 224816 S   0.0  4.3   1:57.91 slack
 2780 gvkalra   20   0 1379216 265472 136304 S   0.0  3.3   0:45.19 slack
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task03 (master) $ cat /proc/proc_memory | grep slack
2529       slack                 1686468                 146920
2600       slack                 316428                  30296
2669       slack                 1102600                 104920
2736       slack                 1496056                 346040
2780       slack                 1379216                 265472
```

Figure 12: Verification (slack)

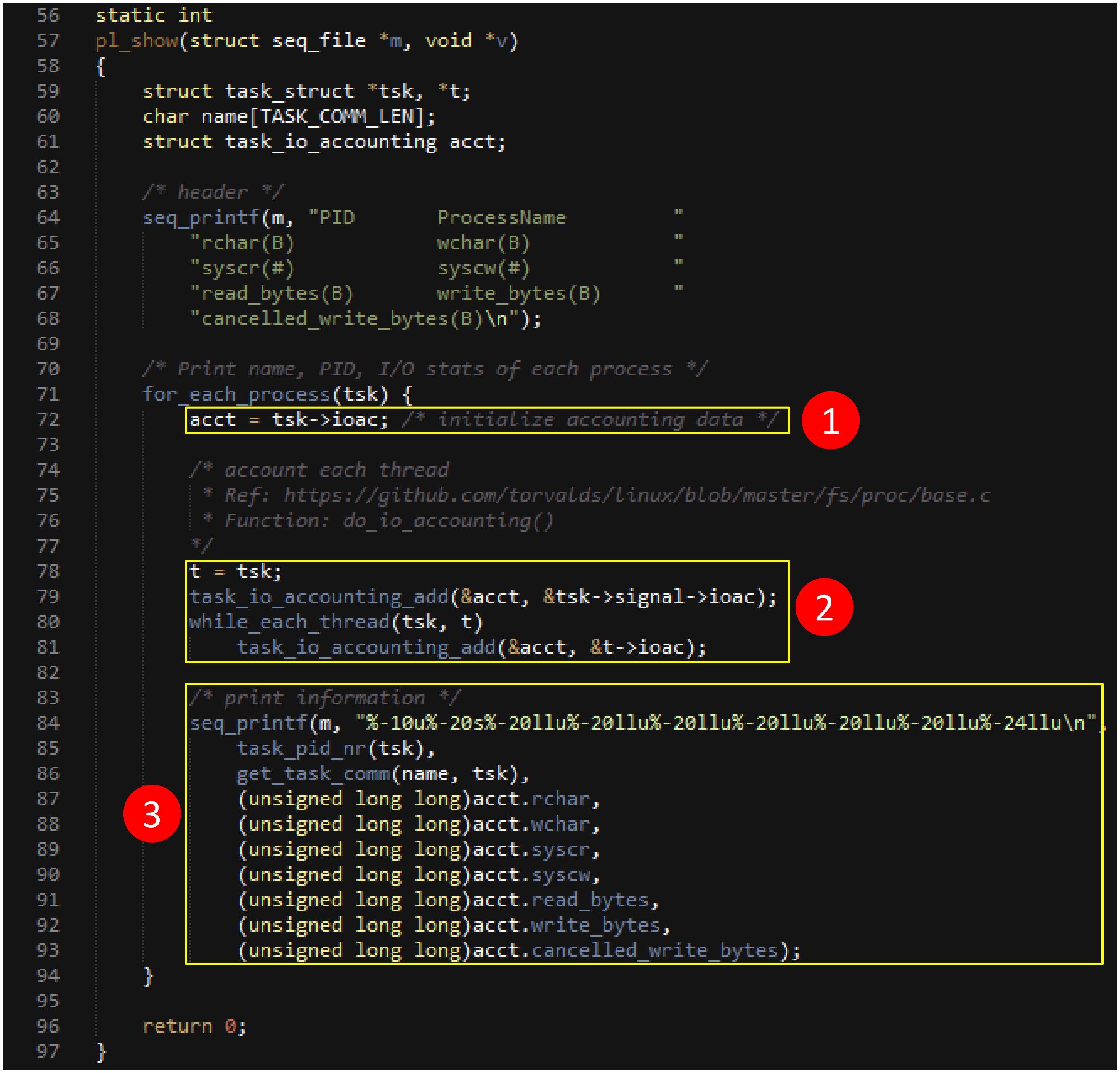
Task 4: per Process I/O Usage

Print every task’s I/O usage



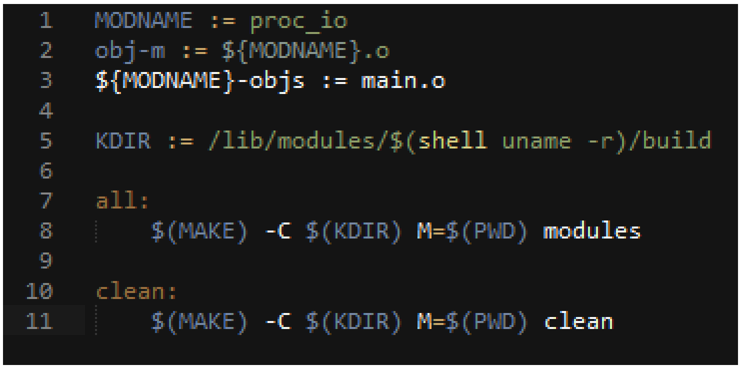
```
/*
Organization of task information in kernel:

    struct task_struct {
        ...
        pid_t pid;
        ...
        struct list_head tasks;
        ...
        char comm[TASK_COMM_LEN];
        ...
        struct task_io_accounting ioac;
    };

    struct list_head {
         struct list_head *next, *prev;
    };

    struct task_io_accounting {
        #ifdef CONFIG_TASK_XACCT
            u64 rchar; //bytes read
            u64 wchar; //bytes written
            u64 syscr; //# of read syscalls
            u64 syscw; //# of write syscalls
        #endif

        #ifdef CONFIG_TASK_IO_ACCOUNTING
            //The number of bytes which this task has caused to be read from storage.
            u64 read_bytes;

            //The number of bytes which this task has caused, or shall cause to be written to disk.
            u64 write_bytes;

            //A task can cause "negative" IO too.  If this task truncates some
            //dirty pagecache, some IO which another task has been accounted for
            //(in its write_bytes) will not be happening.  We _could_ just
            //subtract that from the truncating task's write_bytes, but there is
            //information loss in doing that.
            u64 cancelled_write_bytes;
        #endif
    };
*/
```

Figure 13: Organization of **I/O data** in task_struct



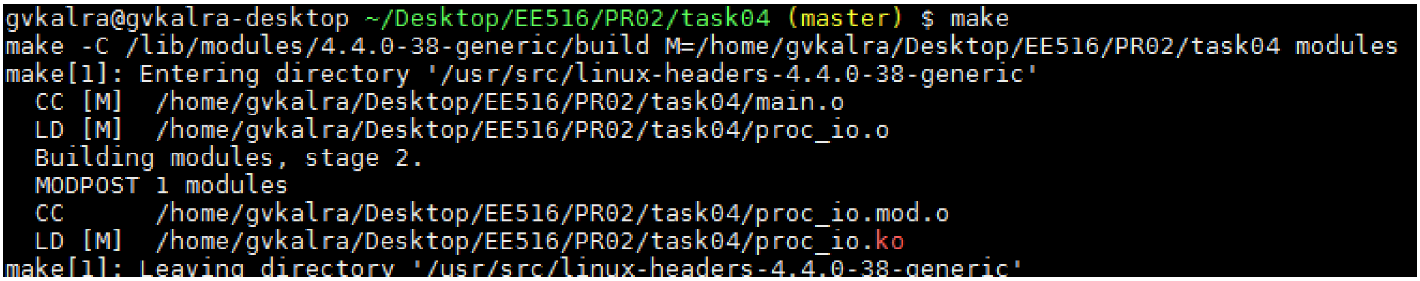
```
static int
pl_show(struct seq_file *m, void *v)
{
    struct task_struct *tsk, *t;
    char name[TASK_COMM_LEN];
    struct task_io_accounting acct;

    /* header */
    seq_printf(m, "PID       ProcessName         "
        "rchar(B)            wchar(B)            "
        "syscr(#)            syscw(#)            "
        "read_bytes(B)       write_bytes(B)      "
        "cancelled_write_bytes(B)\n");

    /* Print name, PID, I/O stats of each process */
    for_each_process(tsk) {
        acct = tsk->ioac; /* initialize accounting data */

        /* account each thread
         * Ref: https://github.com/torvalds/linux/blob/master/fs/proc/base.c
         * Function: do_io_accounting()
        */
        t = tsk;
        task_io_accounting_add(&acct, &tsk->signal->ioac);
        while_each_thread(tsk, t)
            task_io_accounting_add(&acct, &t->ioac);

        /* print information */
        seq_printf(m, "%-10u%-20s%-20llu%-20llu%-20llu%-20llu%-20llu%-20llu%-24llu\n",
            task_pid_nr(tsk),
            get_task_comm(name, tsk),
            (unsigned long long)acct.rchar,
            (unsigned long long)acct.wchar,
            (unsigned long long)acct.syscr,
            (unsigned long long)acct.syscw,
            (unsigned long long)acct.read_bytes,
            (unsigned long long)acct.write_bytes,
            (unsigned long long)acct.cancelled_write_bytes);
    }

    return 0;
}
```

Figure 14: Logic for calculating and printing I/O information

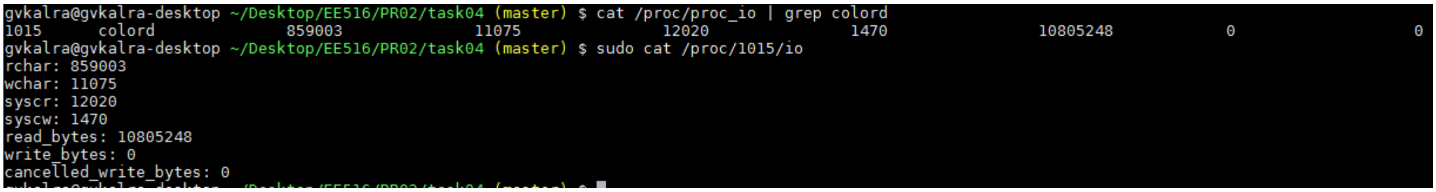


```
MODNAME := proc_io
obj-m := ${MODNAME}.o
${MODNAME}-objs := main.o

KDIR := /lib/modules/$(shell uname -r)/build

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Figure 15: Makefile (no warnings)



```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task04 (master) $ make
make -C /lib/modules/4.4.0-38-generic/build M=/home/gvkalra/Desktop/EE516/PR02/task04 modules
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-38-generic'
  CC [M]  /home/gvkalra/Desktop/EE516/PR02/task04/main.o
  LD [M]  /home/gvkalra/Desktop/EE516/PR02/task04/proc_io.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/gvkalra/Desktop/EE516/PR02/task04/proc_io.mod.o
  LD [M]  /home/gvkalra/Desktop/EE516/PR02/task04/proc_io.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-38-generic'
```

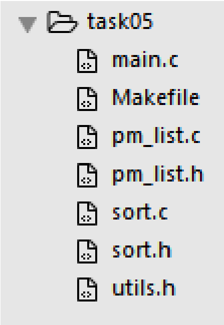


```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task04 (master) $ cat /proc/proc_io | grep colord
1015      colord              859003              11075               12020               1470                10805248            0                   0
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task04 (master) $ sudo cat /proc/1015/io
rchar: 859003
wchar: 11075
syscr: 12020
syscw: 1470
read_bytes: 10805248
write_bytes: 0
cancelled_write_bytes: 0
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task04 (master) $
```

Figure 16: Verification (colord)

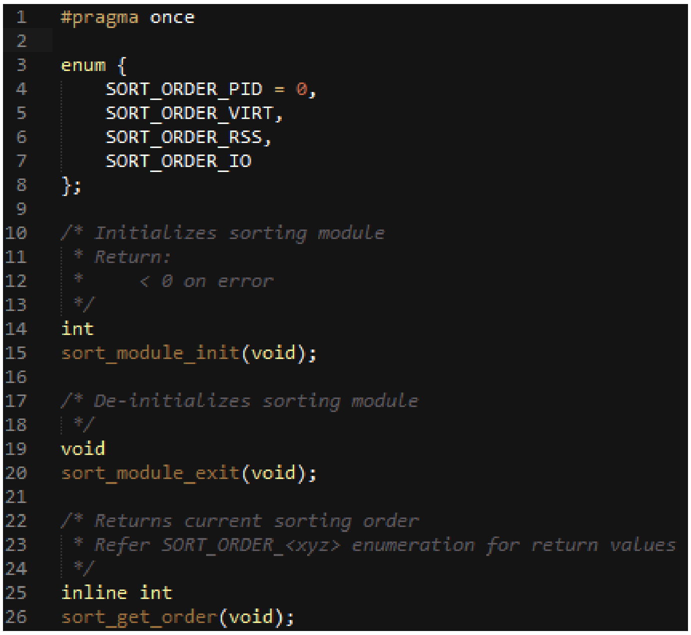
Task 5: Sorting Features

Create a process monitoring tool with sorting feature

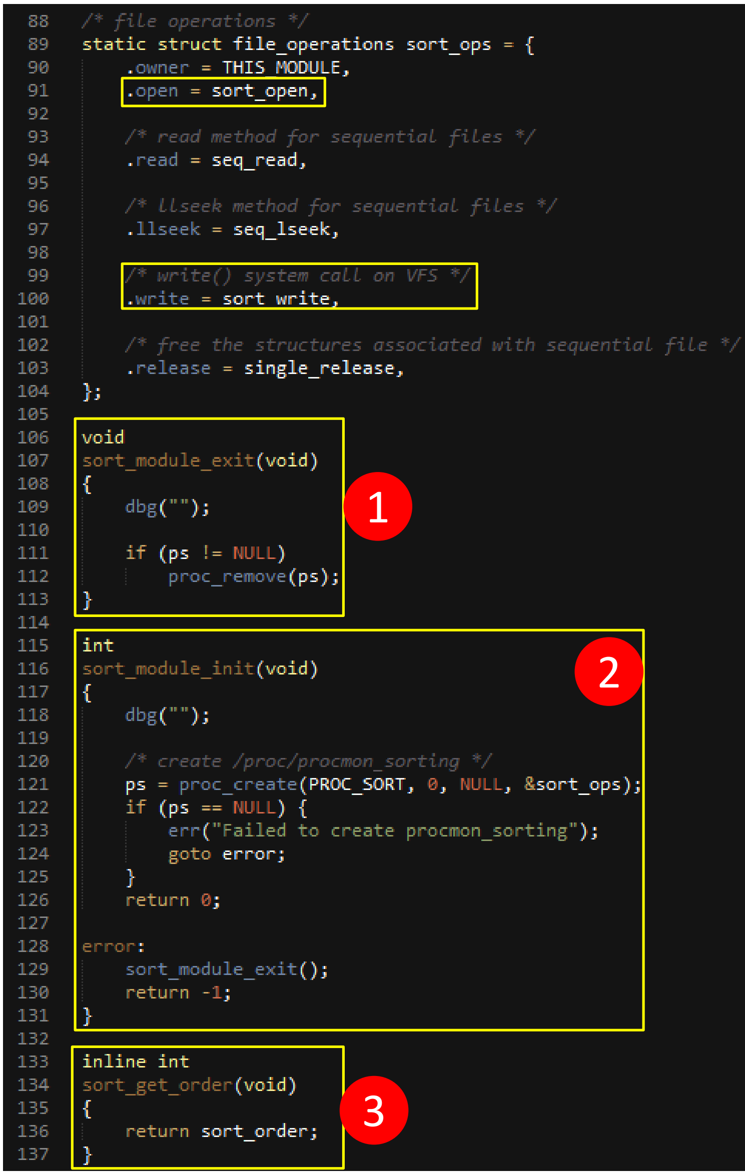


The task is divided into 2 modules:

1. **Sorting Module** – It is used to view or update current sorting order
2. **PM (procmon) Module** – It is used to create process monitoring

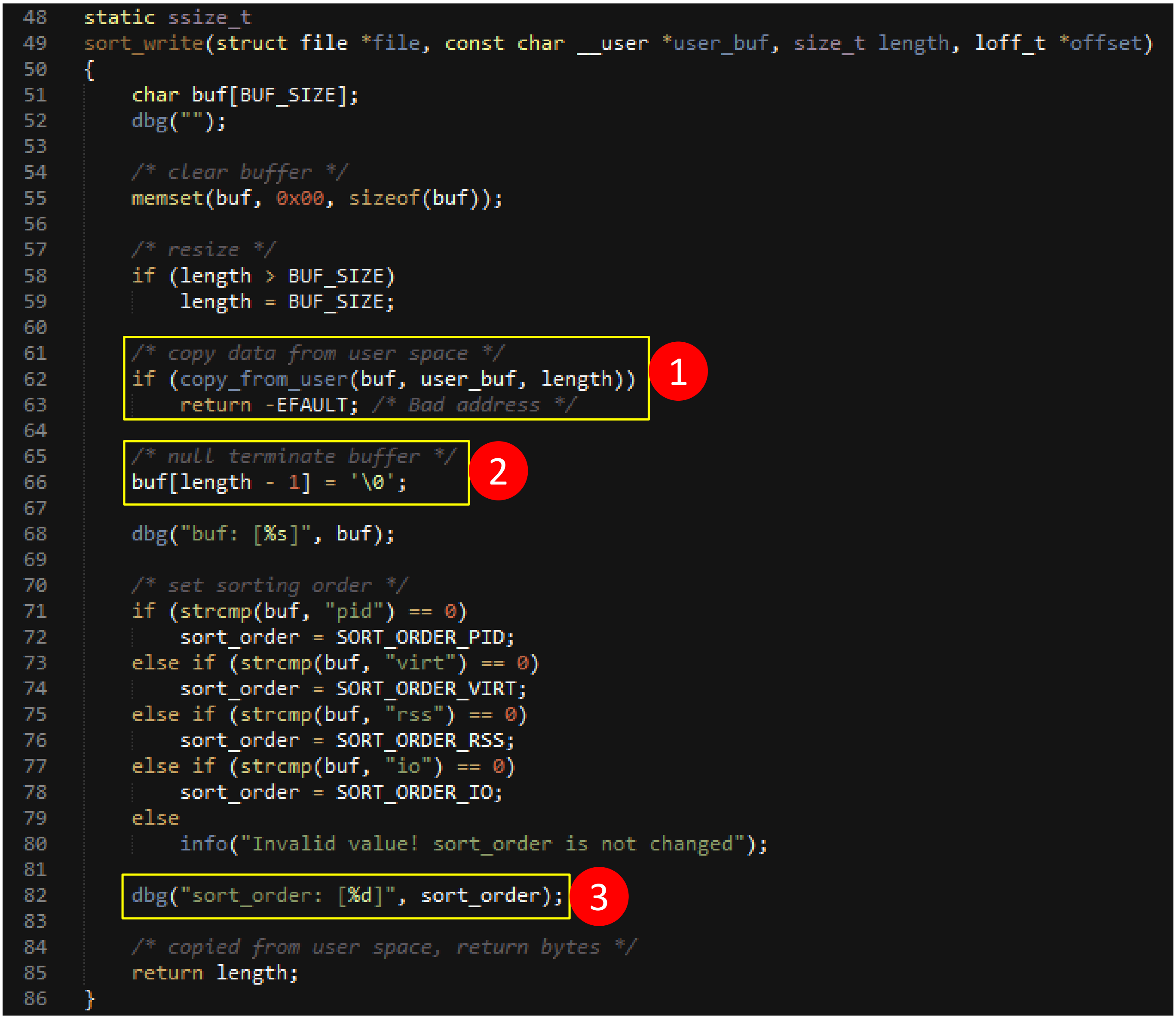


```
#pragma once

enum {
    SORT_ORDER_PID = 0,
    SORT_ORDER_VIRT,
    SORT_ORDER_RSS,
    SORT_ORDER_IO
};

/* Initializes sorting module
 * Return:
 *     < 0 on error
 */
int
sort_module_init(void);

/* De-initializes sorting module
 */
void
sort_module_exit(void);

/* Returns current sorting order
 * Refer SORT_ORDER_<xyz> enumeration for return values
 */
inline int
sort_get_order(void);
```



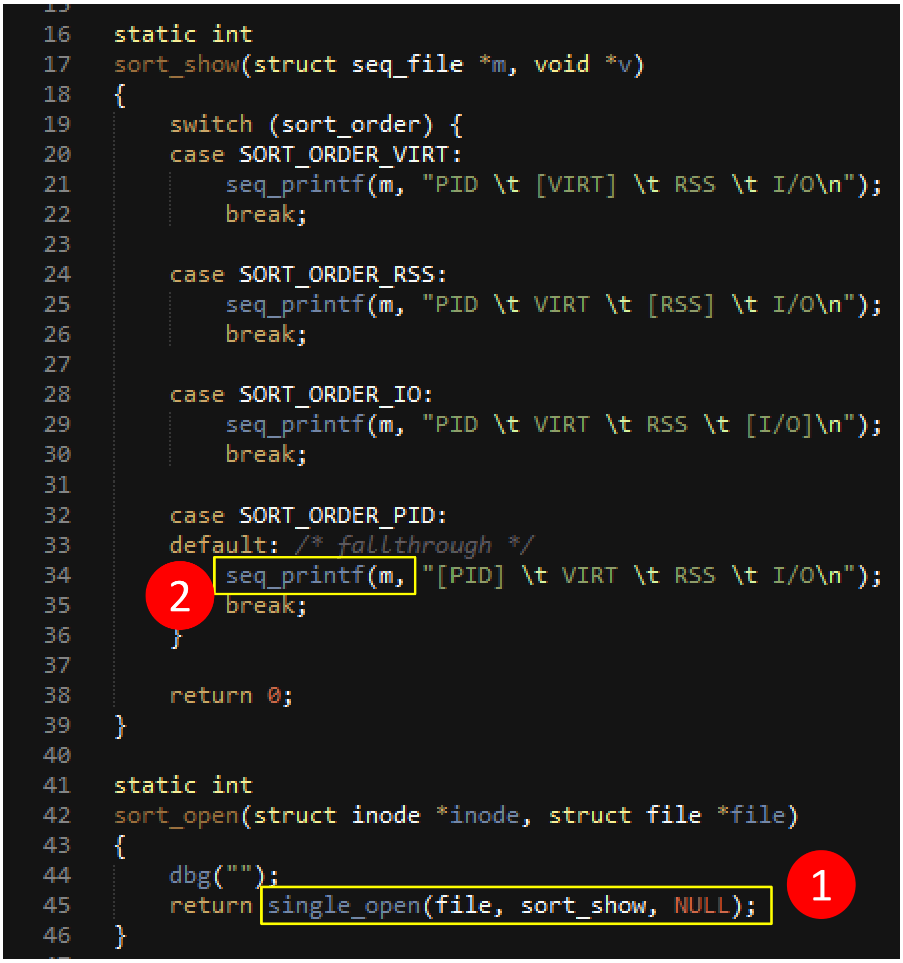
```
/* file operations */
static struct file_operations sort_ops = {
    .owner = THIS_MODULE,
    .open = sort_open,

    /* read method for sequential files */
    .read = seq_read,

    /* llseek method for sequential files */
    .llseek = seq_lseek,

    /* write() system call on VFS */
    .write = sort_write,

    /* free the structures associated with sequential file */
    .release = single_release,
};

void
sort_module_exit(void)
{
    dbg("");

    if (ps != NULL)
        proc_remove(ps);
}

int
sort_module_init(void)
{
    dbg("");

    /* create /proc/procmon_sorting */
    ps = proc_create(PROC_SORT, 0, NULL, &sort_ops);
    if (ps == NULL) {
        err("Failed to create procmon_sorting");
        goto error;
    }
    return 0;

error:
    sort_module_exit();
    return -1;
}

inline int
sort_get_order(void)
{
    return sort_order;
}
```

Figure 17: Sorting Module APIs exposed to PM module



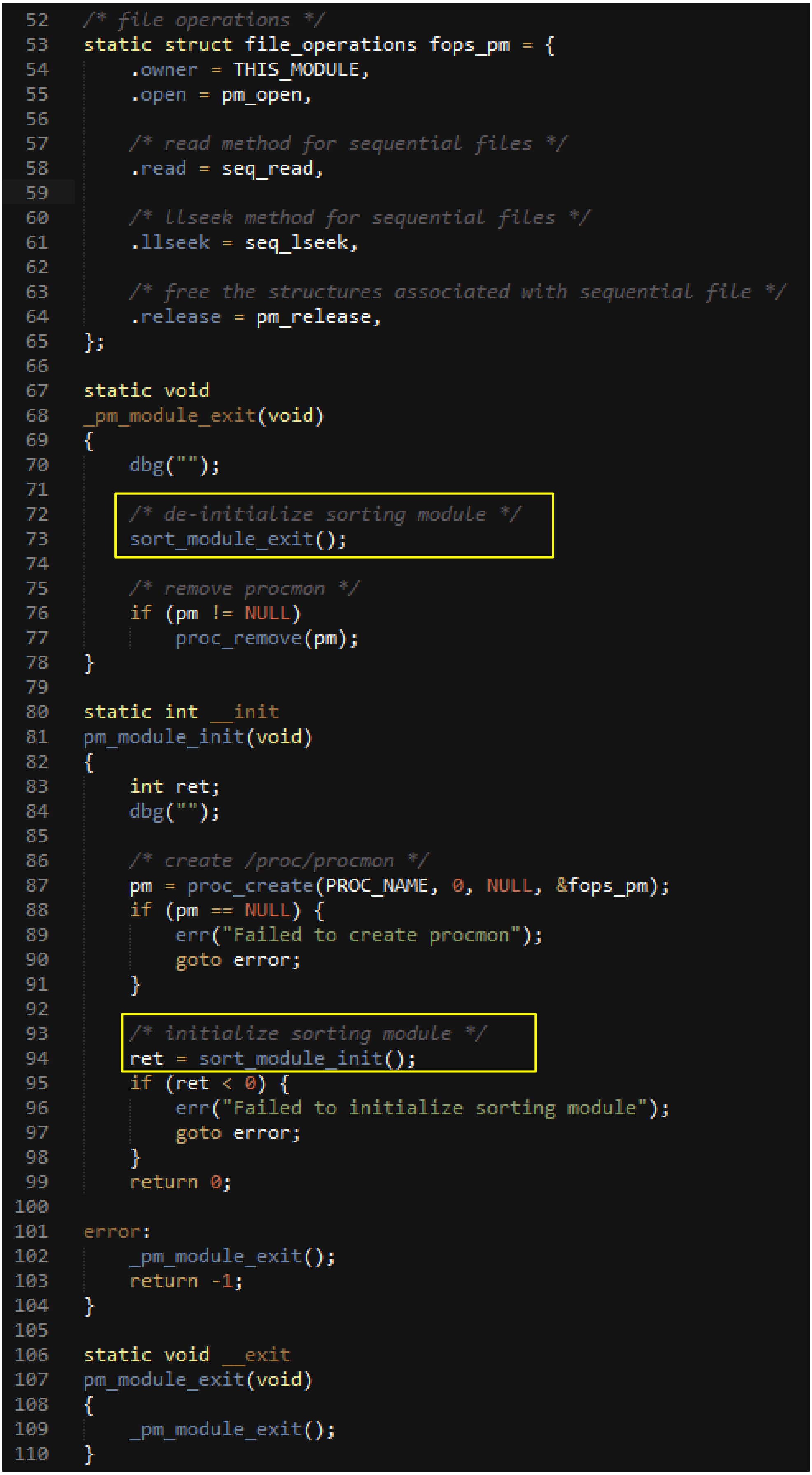
```
static ssize_t
sort_write(struct file *file, const char __user *user_buf, size_t length, loff_t *offset)
{
    char buf[BUF_SIZE];
    dbg("");

    /* clear buffer */
    memset(buf, 0x00, sizeof(buf));

    /* resize */
    if (length > BUF_SIZE)
        length = BUF_SIZE;

    /* copy data from user space */
    if (copy_from_user(buf, user_buf, length))
        return -EFAULT; /* Bad address */

    /* null terminate buffer */
    buf[length - 1] = '\0';

    dbg("buf: [%s]", buf);

    /* set sorting order */
    if (strcmp(buf, "pid") == 0)
        sort_order = SORT_ORDER_PID;
    else if (strcmp(buf, "virt") == 0)
        sort_order = SORT_ORDER_VIRT;
    else if (strcmp(buf, "rss") == 0)
        sort_order = SORT_ORDER_RSS;
    else if (strcmp(buf, "io") == 0)
        sort_order = SORT_ORDER_IO;
    else
        info("Invalid value! sort_order is not changed");

    dbg("sort_order: [%d]", sort_order);

    /* copied from user space, return bytes */
    return length;
}
```

Figure 18: Sorting Module write() file operation to update sort_order



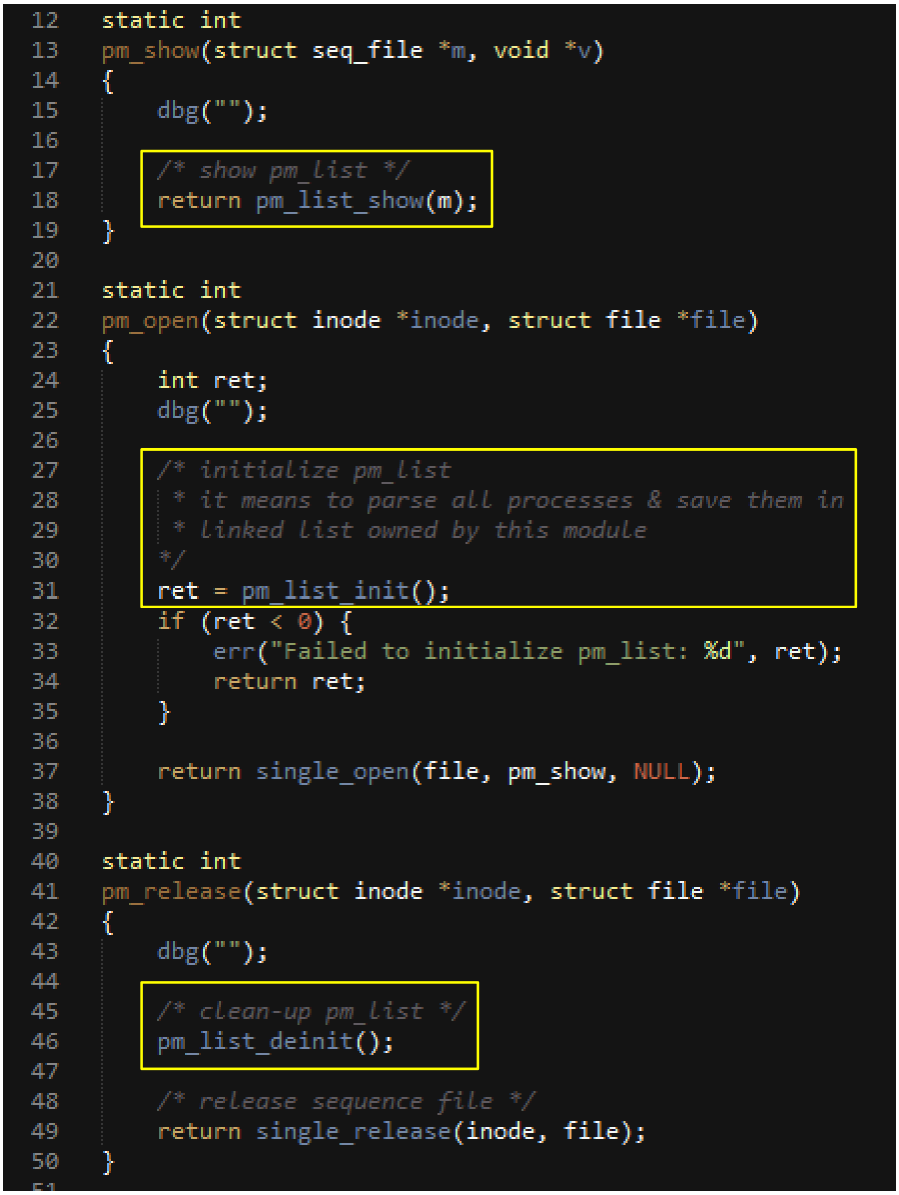
```
static int
sort_show(struct seq_file *m, void *v)
{
    switch (sort_order) {
    case SORT_ORDER_VIRT:
        seq_printf(m, "PID \t [VIRT] \t RSS \t I/O\n");
        break;

    case SORT_ORDER_RSS:
        seq_printf(m, "PID \t VIRT \t [RSS] \t I/O\n");
        break;

    case SORT_ORDER_IO:
        seq_printf(m, "PID \t VIRT \t RSS \t [I/O]\n");
        break;

    case SORT_ORDER_PID:
    default: /* fallthrough */
        seq_printf(m, "[PID] \t VIRT \t RSS \t I/O\n");
        break;
    }

    return 0;
}

static int
sort_open(struct inode *inode, struct file *file)
{
    dbg("");
    return single_open(file, sort_show, NULL);
}
```

Figure 19: Sorting Module open() file operation



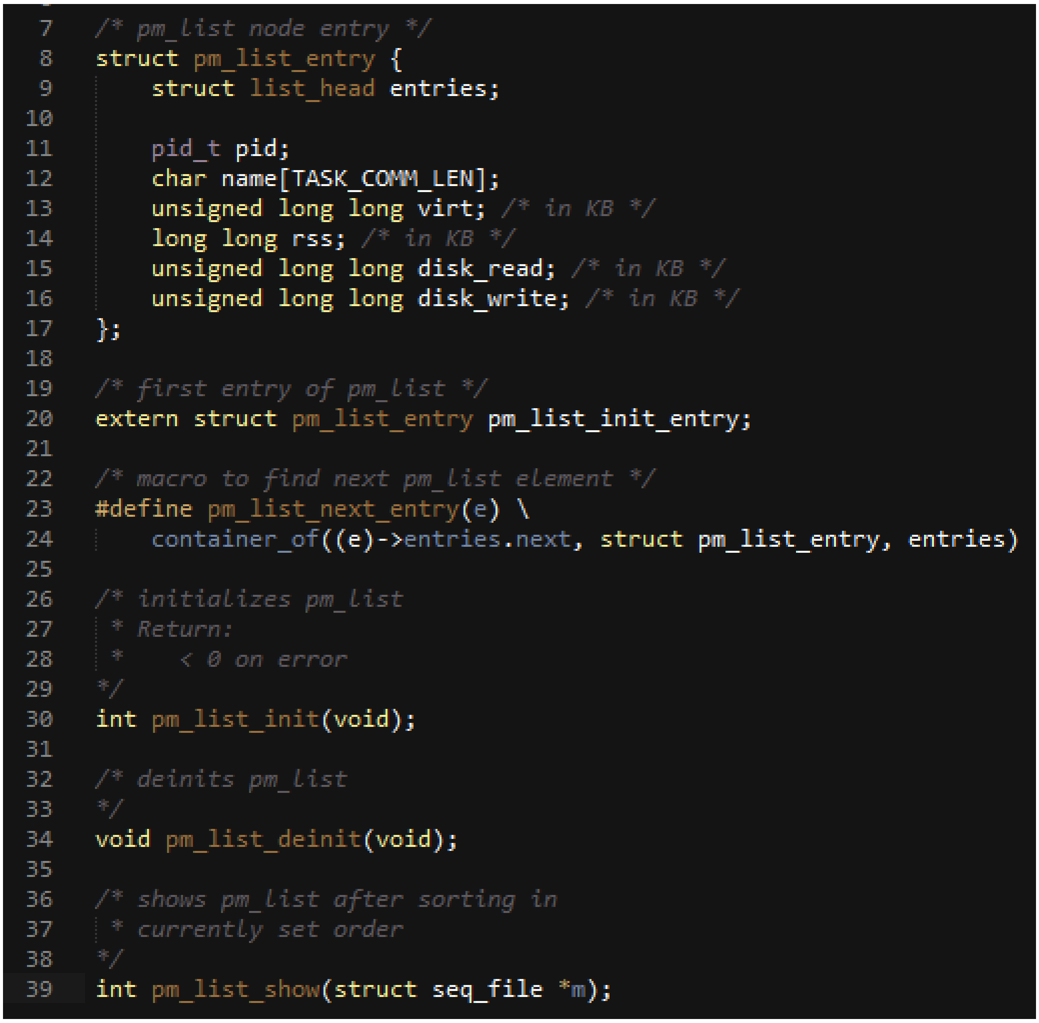
```
/* file operations */
static struct file_operations fops_pm = {
    .owner = THIS_MODULE,
    .open = pm_open,

    /* read method for sequential files */
    .read = seq_read,

    /* llseek method for sequential files */
    .llseek = seq_lseek,

    /* free the structures associated with sequential file */
    .release = pm_release,
};

static void
_pm_module_exit(void)
{
    dbg("");

    /* de-initialize sorting module */
    sort_module_exit();

    /* remove procmon */
    if (pm != NULL)
        proc_remove(pm);
}

static int __init
pm_module_init(void)
{
    int ret;
    dbg("");

    /* create /proc/procmon */
    pm = proc_create(PROC_NAME, 0, NULL, &fops_pm);
    if (pm == NULL) {
        err("Failed to create procmon");
        goto error;
    }

    /* initialize sorting module */
    ret = sort_module_init();
    if (ret < 0) {
        err("Failed to initialize sorting module");
        goto error;
    }
    return 0;

error:
    _pm_module_exit();
    return -1;
}

static void __exit
pm_module_exit(void)
{
    _pm_module_exit();
}
```

Figure 20: PM Module init / exit
(Sorting module is init/exit by PM module)



```c
static int
pm_show(struct seq_file *m, void *v)
{
    dbg("");

    /* show pm_list */
    return pm_list_show(m);
}

static int
pm_open(struct inode *inode, struct file *file)
{
    int ret;
    dbg("");

    /* initialize pm_list
     * it means to parse all processes & save them in
     * linked list owned by this module
     */
    ret = pm_list_init();
    if (ret < 0) {
        err("Failed to initialize pm_list: %d", ret);
        return ret;
    }

    return single_open(file, pm_show, NULL);
}

static int
pm_release(struct inode *inode, struct file *file)
{
    dbg("");

    /* clean-up pm_list */
    pm_list_deinit();

    /* release sequence file */
    return single_release(inode, file);
}
```

Figure 21: PM Module open() and release()
(**pm_list** is PM Module’s data structure discussed in next page)



```c
/* pm_list node entry */
struct pm_list_entry {
    struct list_head entries;

    pid_t pid;
    char name[TASK_COMM_LEN];
    unsigned long long virt; /* in KB */
    long long rss; /* in KB */
    unsigned long long disk_read; /* in KB */
    unsigned long long disk_write; /* in KB */
};

/* first entry of pm_list */
extern struct pm_list_entry pm_list_init_entry;

/* macro to find next pm_list element */
#define pm_list_next_entry(e) \
    container_of((e)->entries.next, struct pm_list_entry, entries)

/* initializes pm_list
 * Return:
 *    < 0 on error
*/
int pm_list_init(void);

/* deinits pm_list
*/
void pm_list_deinit(void);

/* shows pm_list after sorting in
 * currently set order
*/
int pm_list_show(struct seq_file *m);
```

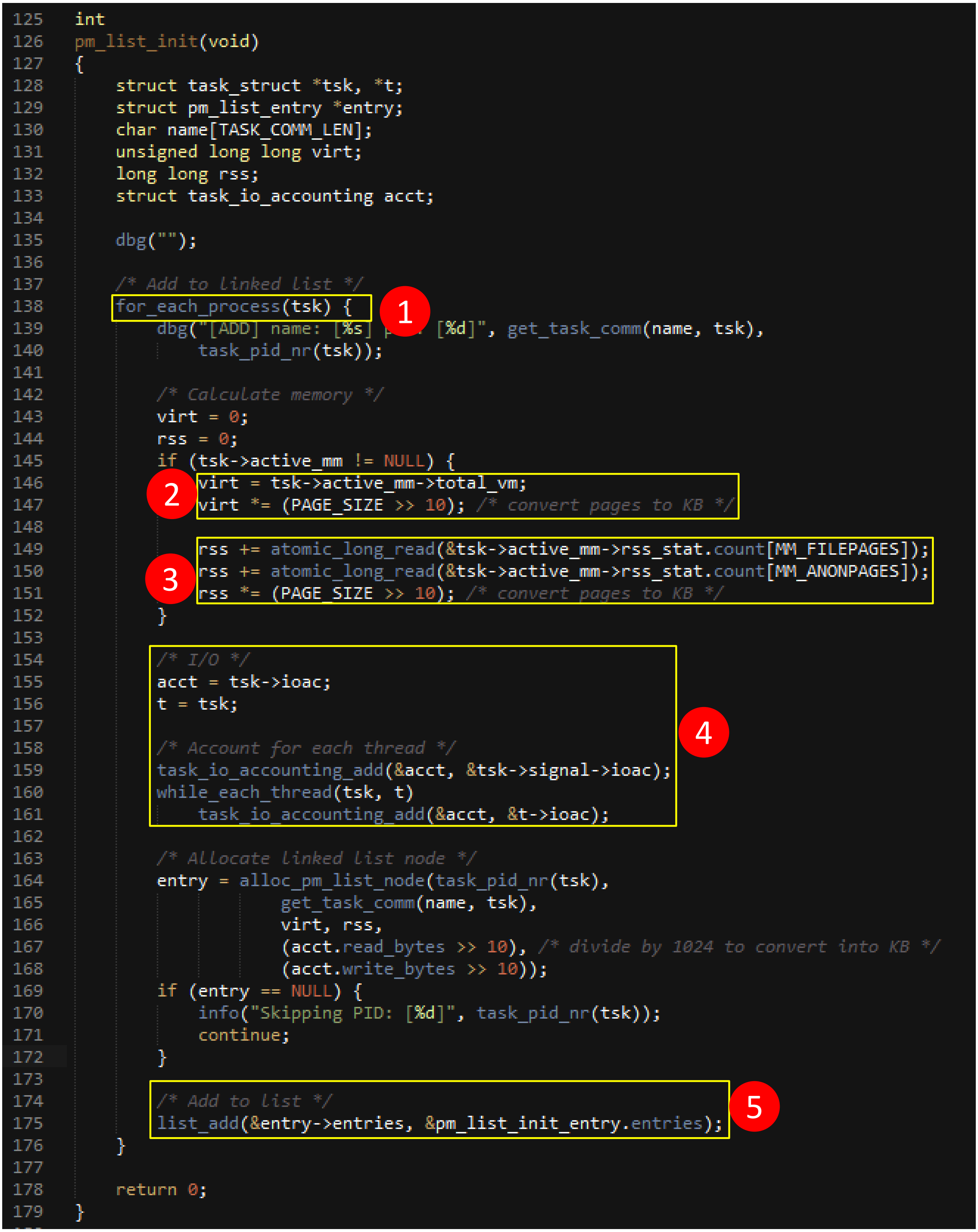
Figure 22: **pm_list** data structure & APIs.
(Each API is discussed next)

Notes:

1. The design of this data structure closely resembles task_struct from the Linux Kernel.

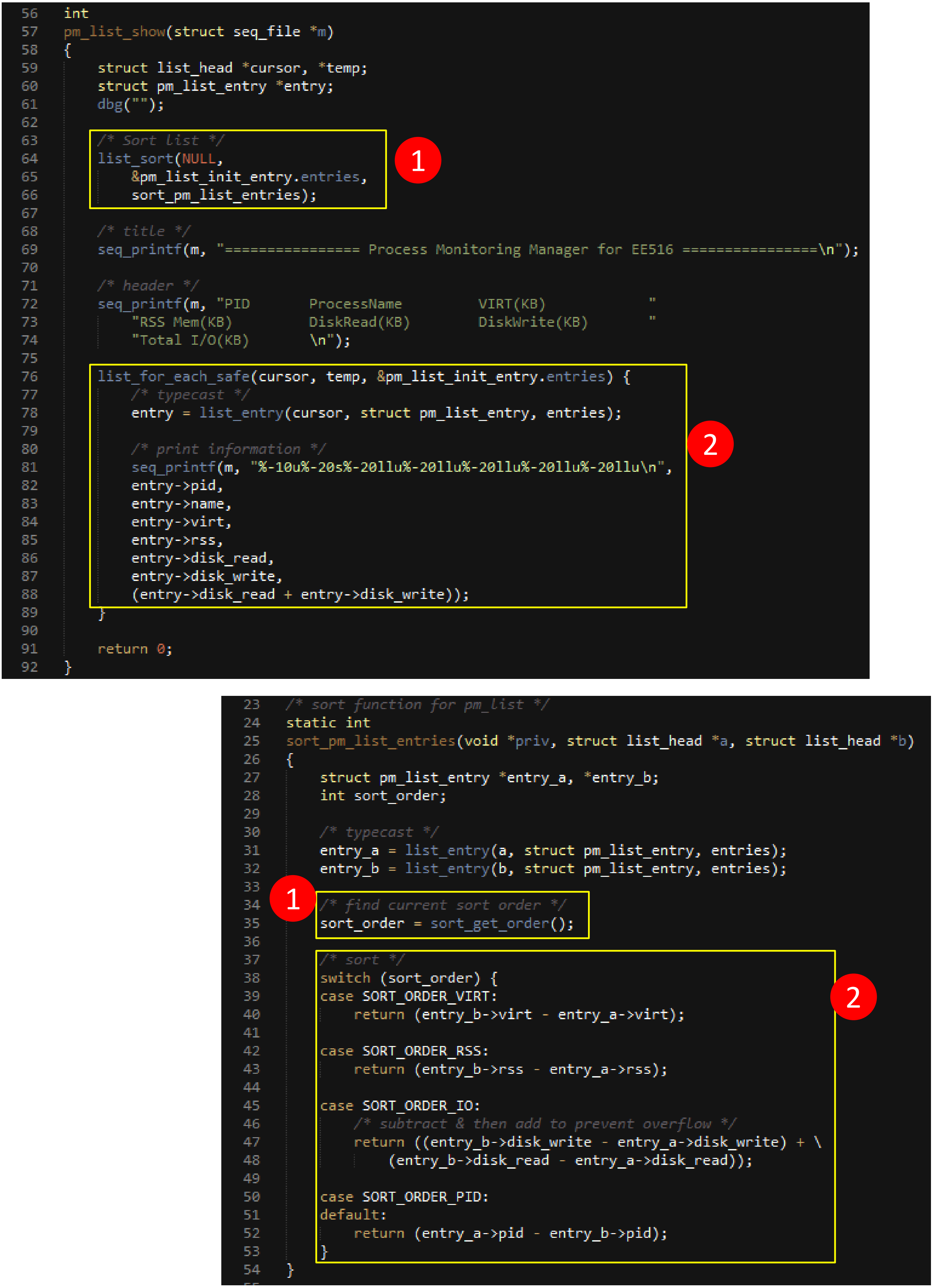
2. Reference:
https://github.com/torvalds/linux/blob/master/init/init_task.c

3. “struct list_head” is the kernel way of providing linked lists, which has been referred from
https://github.com/torvalds/linux/blob/master/include/linux/list.h



```c
int
pm_list_init(void)
{
    struct task_struct *tsk, *t;
    struct pm_list_entry *entry;
    char name[TASK_COMM_LEN];
    unsigned long long virt;
    long long rss;
    struct task_io_accounting acct;

    dbg("");

    /* Add to linked list */
    for_each_process(tsk) {
        dbg("[ADD] name: [%s] pid: [%d]", get_task_comm(name, tsk),
            task_pid_nr(tsk));

        /* Calculate memory */
        virt = 0;
        rss = 0;
        if (tsk->active_mm != NULL) {
            virt = tsk->active_mm->total_vm;
            virt *= (PAGE_SIZE >> 10); /* convert pages to KB */

            rss += atomic_long_read(&tsk->active_mm->rss_stat.count[MM_FILEPAGES]);
            rss += atomic_long_read(&tsk->active_mm->rss_stat.count[MM_ANONPAGES]);
            rss *= (PAGE_SIZE >> 10); /* convert pages to KB */
        }

        /* I/O */
        acct = tsk->ioac;
        t = tsk;

        /* Account for each thread */
        task_io_accounting_add(&acct, &tsk->signal->ioac);
        while_each_thread(tsk, t)
            task_io_accounting_add(&acct, &t->ioac);

        /* Allocate linked list node */
        entry = alloc_pm_list_node(task_pid_nr(tsk),
                    get_task_comm(name, tsk),
                    virt, rss,
                    (acct.read_bytes >> 10), /* divide by 1024 to convert into KB */
                    (acct.write_bytes >> 10));
        if (entry == NULL) {
            info("Skipping PID: [%d]", task_pid_nr(tsk));
            continue;
        }

        /* Add to list */
        list_add(&entry->entries, &pm_list_init_entry.entries);
    }

    return 0;
}
```

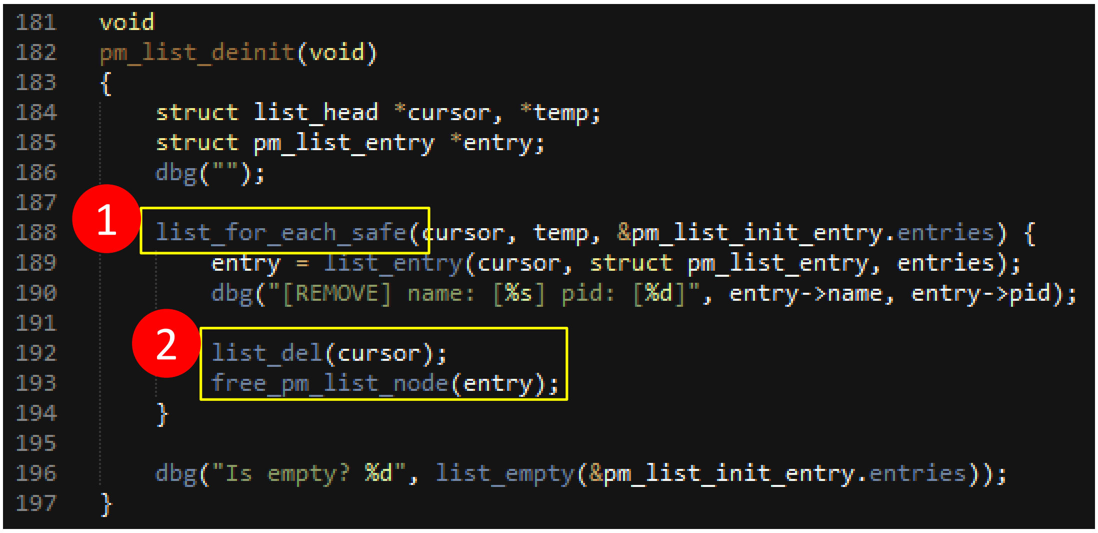
Figure 23: pm_list_init() – API for creating & initializing a PM list



```c
int
pm_list_show(struct seq_file *m)
{
    struct list_head *cursor, *temp;
    struct pm_list_entry *entry;
    dbg("");

    /* Sort list */
    list_sort(NULL,
        &pm_list_init_entry.entries,
        sort_pm_list_entries);

    /* title */
    seq_printf(m, "================ Process Monitoring Manager for EE516 ================\n");

    /* header */
    seq_printf(m, "PID       ProcessName         VIRT(KB)            "
        "RSS Mem(KB)        DiskRead(KB)        DiskWrite(KB)       "
        "Total I/O(KB)      \n");

    list_for_each_safe(cursor, temp, &pm_list_init_entry.entries) {
        /* typecast */
        entry = list_entry(cursor, struct pm_list_entry, entries);

        /* print information */
        seq_printf(m, "%-10u%-20s%-20llu%-20llu%-20llu%-20llu%-20llu\n",
        entry->pid,
        entry->name,
        entry->virt,
        entry->rss,
        entry->disk_read,
        entry->disk_write,
        (entry->disk_read + entry->disk_write));
    }

    return 0;
}
```

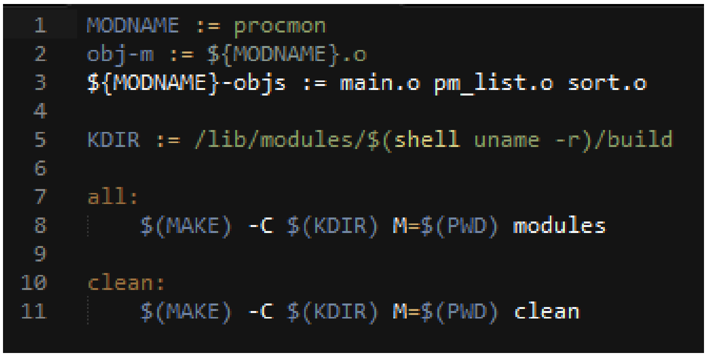
```c
/* sort function for pm_list */
static int
sort_pm_list_entries(void *priv, struct list_head *a, struct list_head *b)
{
    struct pm_list_entry *entry_a, *entry_b;
    int sort_order;

    /* typecast */
    entry_a = list_entry(a, struct pm_list_entry, entries);
    entry_b = list_entry(b, struct pm_list_entry, entries);

    /* find current sort order */
    sort_order = sort_get_order();

    /* sort */
    switch (sort_order) {
    case SORT_ORDER_VIRT:
        return (entry_b->virt - entry_a->virt);

    case SORT_ORDER_RSS:
        return (entry_b->rss - entry_a->rss);

    case SORT_ORDER_IO:
        /* subtract & then add to prevent overflow */
        return ((entry_b->disk_write - entry_a->disk_write) + \
            (entry_b->disk_read - entry_a->disk_read));

    case SORT_ORDER_PID:
    default:
        return (entry_a->pid - entry_b->pid);
    }
}
```

Figure 24: pm_list_show() – API for displaying PM list in sorted order



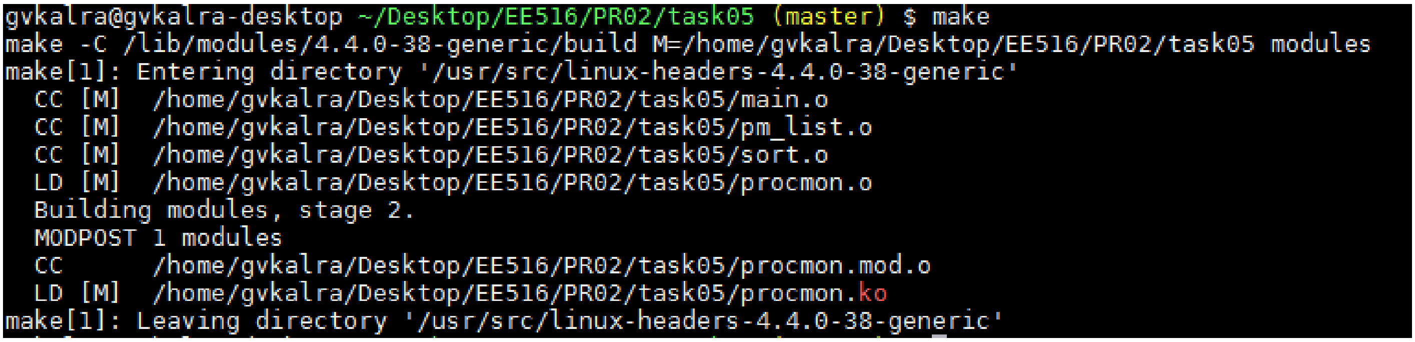
```
void
pm_list_deinit(void)
{
    struct list_head *cursor, *temp;
    struct pm_list_entry *entry;
    dbg("");

    list_for_each_safe(cursor, temp, &pm_list_init_entry.entries) {
        entry = list_entry(cursor, struct pm_list_entry, entries);
        dbg("[REMOVE] name: [%s] pid: [%d]", entry->name, entry->pid);

        list_del(cursor);
        free_pm_list_node(entry);
    }

    dbg("Is empty? %d", list_empty(&pm_list_init_entry.entries));
}
```

Figure 25: pm_list_deinit() – API for freeing a PM list

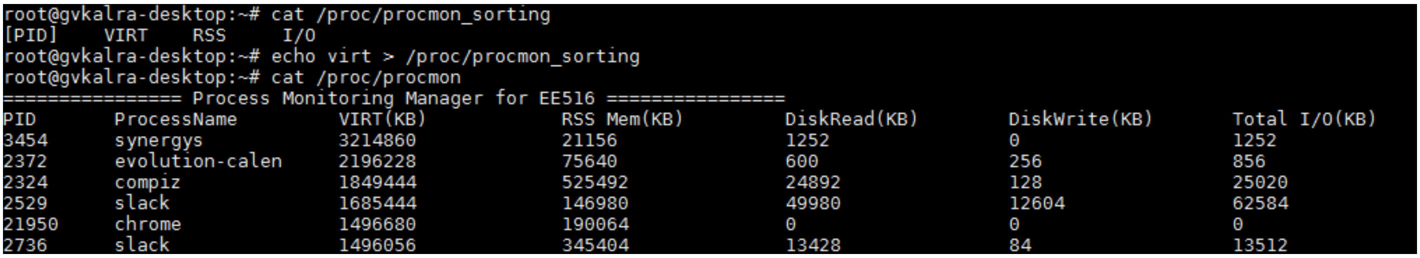


```
MODNAME := procmon
obj-m := ${MODNAME}.o
${MODNAME}-objs := main.o pm_list.o sort.o

KDIR := /lib/modules/$(shell uname -r)/build

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Figure 26: Makefile (no warnings)

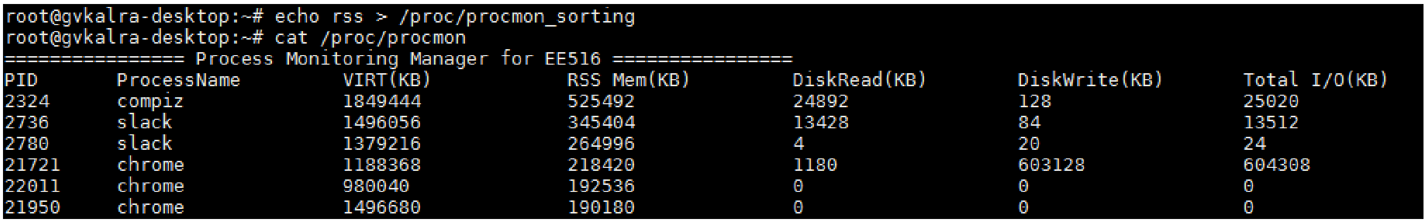


```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR02/task05 (master) $ make
make -C /lib/modules/4.4.0-38-generic/build M=/home/gvkalra/Desktop/EE516/PR02/task05 modules
make[1]: Entering directory '/usr/src/linux-headers-4.4.0-38-generic'
  CC [M]  /home/gvkalra/Desktop/EE516/PR02/task05/main.o
  CC [M]  /home/gvkalra/Desktop/EE516/PR02/task05/pm_list.o
  CC [M]  /home/gvkalra/Desktop/EE516/PR02/task05/sort.o
  LD [M]  /home/gvkalra/Desktop/EE516/PR02/task05/procmon.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/gvkalra/Desktop/EE516/PR02/task05/procmon.mod.o
  LD [M]  /home/gvkalra/Desktop/EE516/PR02/task05/procmon.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-38-generic'
```



```
root@gvkalra-desktop:~# cat /proc/procmon_sorting
[PID]    VIRT    RSS    I/O
root@gvkalra-desktop:~# echo virt > /proc/procmon_sorting
root@gvkalra-desktop:~# cat /proc/procmon
================ Process Monitoring Manager for EE516 ================
PID         ProcessName         VIRT(KB)            RSS Mem(KB)         DiskRead(KB)        DiskWrite(KB)       Total I/O(KB)
3454        synergys            3214860             21156               1252                0                   1252
2372        evolution-calen     2196228             75640               600                 256                 856
2324        compiz              1849444             525492              24892               128                 25020
2529        slack               1685444             146980              49980               12604               62584
21950       chrome              1496680             190064              0                   0                   0
2736        slack               1496056             345404              13428               84                  13512
```

Figure 27: Verification (VIRT)



```
root@gvkalra-desktop:~# echo rss > /proc/procmon_sorting
root@gvkalra-desktop:~# cat /proc/procmon
================ Process Monitoring Manager for EE516 ================
PID         ProcessName         VIRT(KB)            RSS Mem(KB)         DiskRead(KB)        DiskWrite(KB)       Total I/O(KB)
2324        compiz              1849444             525492              24892               128                 25020
2736        slack               1496056             345404              13428               84                  13512
2780        slack               1379216             264996              4                   20                  24
21721       chrome              1188368             218420              1180                603128              604308
22011       chrome              980040              192536              0                   0                   0
21950       chrome              1496680             190180              0                   0                   0
```

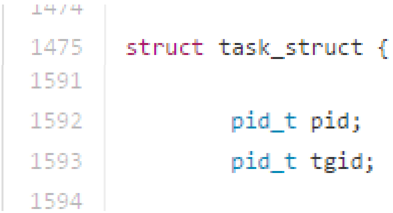
Figure 28: Verification (RSS)

Problem 1
Describe similarity and difference Process (Thread Group) and Thread in the Linux

In Linux, Processes & Threads are all represented by task_struct. The pid field of task_struct is unique for every process & thread.

However, since POSIX mandates that all threads of a process should have the same ProcessID (as seen by the user), Linux uses “Thread Group ID” (TGID) to satisfy POSIX mandate.

A TGID is the PID of the thread that started the whole process.



```
struct task_struct {

        pid_t pid;
        pid_t tgid;

```

Figure 29: pid/tgid in task_struct.
Source: https://github.com/torvalds/linux/blob/master/include/linux/sched.h

In Linux, both processes & threads are created using clone() system call. The ‘flags’ passed to clone() specify the degree of data sharing. Threads in Linux have it’s own stack but shares Heap, BSS, Data and Text with other threads having same TGID.

The CFS (Completely Fair Scheduler) as used in Kernel doesn’t discriminate between a thread & a process. It works on “tasks” and is not aware of thread & process abstractions.

Problem 2

Use of copy_to_user / copy_from_user functions in proc filesystem

In Linux, user-space is separated from kernel-space in all aspects. To arbitrate data between user / kernel space, Linux provides two functions:

1. **copy_to_user**: Copy data from kernel space to user space
2. **copy_from_user**: Copy data from user space to kernel space

A proc file system (procfs) is just another type of file system on top of Virtual File System (VFS) & thus applies the same rule of data exchange between user & kernel space.