VE482 Lab 8 Report

Lan Wang 519370910084

Kaiwen Zhang 519370910188

2 Memory in Minix 3

2.1 Memory management at kernel level

- **What does `vm` stands for? (Hint: in this context the answer is not virtual machine)**

  Virtual Memory

- **Find the page table definition and search what fields each entry contain?**

  Page table is defined in `servers/vm/pt.h`:

```
/* A pagetable. */
typedef struct {
    /* Directory entries in VM addr space - root of page table.  */
    u32_t *pt_dir;      /* page aligned (ARCH_VM_DIR_ENTRIES) */
    u32_t pt_dir_phys;  /* physical address of pt_dir */

    /* Pointers to page tables in VM address space. */
    u32_t *pt_pt[ARCH_VM_DIR_ENTRIES];

    /* When looking for a hole in virtual address space, start
     * looking here. This is in linear addresses, i.e.,
     * not as the process sees it but the position in the page
     * page table. This is just a hint.
     */
    u32_t pt_virtop;
} pt_t;
```

- **What basic functions are used to handle virtual memory?**

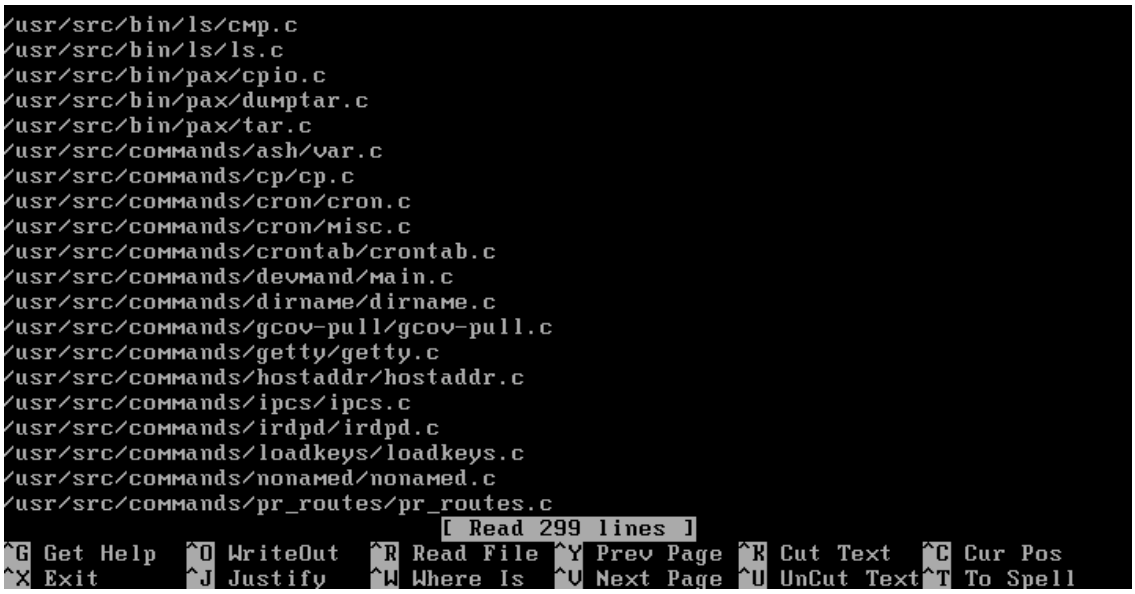
  To see the functions used to handle VM, we can check the main function of VM. Some basic functions are listed below:

```
/* Initialize system so that all processes are runnable */
void init_vm(void);

/* Register init callbacks. */
void sef_setcb_init_restart(sef_cb_init_t cb);
void sef_setcb_signal_handler(sef_cb_signal_handler_t cb);

/* Let SEF perform startup. */
void sef_startup();

/* Check status */
int sef_receive_status(endpoint_t src, message *m_ptr, int *status_ptr);
int vm_isokendpt(endpoint_t endpoint, int *proc);
static int vm_acl_ok(endpoint_t caller, int call);
```

```
/* Map all the services in the boot image and return a result which is used
to send reply message */
static int do_rs_init(message *m);

/* Handle pagefault */
void do_pagefaults(message *m);
void pt_clearmapcache(void);
```

- **Find all the places where the `vm` used inside the kernel, Why does it appear in so many different places?**

  `find /usr/src -name "*.c" | xargs grep -l "vm" > usage`



  The file contains 299 lines, indicating 299 files using virtual memory.

  Basically, when RAM is not enough, virtual memory will be used. Also, in MINIX3, process manager is split into process management and memory management. So functions like `fork()` and `exit()` are implemented in VM, leading to so many usages of VM.

- **How is memory allocated within the kernel? Why are not `malloc` and `calloc` used?**

  By using `kmalloc()` or `vmalloc()` (allocate virtual memory).

  `malloc()` and `calloc()` are defined in user-space, so they cannot be used in kernel space (kernel can only use functions defined in kernel space).

- **While allocating memory, how does the functions in kernel space switch back and forth between user and kernel spaces? How is that boundary crossed? How good or bad it is to put `vm` in userspace?**

  When `malloc()` is called in user space, it will see if there's a piece of unused memory that satisfy the allocation requirements. If not, it will call the kernel function `mmap()` to ask for some memory. Kernel will then allocates memory and allow the userspace process to map it into its address space with `mmap()`. User space process will then use what returned from `mmap()` to access that memory.

  I think it's a bad idea. Since the memory management needs to be done in kernel space, regrading to technique reasons and safety reasons, if virtual memory is moved to user space, this means that the boundary between kernel space and user space will be crossed frequently, which is not a good idea.

- **How are pagefaults handled?** [1] [2] [3]

Kernel sends the pagefault message through `pagefault()`. When VM receives the message in its `main()` function, it will call `do_pagefaults()` to handle it:

```
...
if(msg.m_type == RS_INIT && msg.m_source == RS_PROC_NR) {
    result = do_rs_init(&msg);
} else if (msg.m_type == VM_PAGEFAULT) {
    if (!IPC_STATUS_FLAGS_TEST(rcv_sts, IPC_FLG_MSG_FROM_KERNEL)) {
        printf("VM: process %d faked VM_PAGEFAULT "
                    "message!\n", msg.m_source);
    }
    do_pagefaults(&msg);
    pt_clearmapcache();
    ...
```

In `do_pagefaults()`, it checks whether the address is valid. If the process is writing, it will also check whether the memory is writable. After that, it sends signal back to kernel through `sys_vmctl()`.

After kernel receives this signal, it calls `do_vmctl()` to handle it. It checks `VMCTL_CLEAR_PAGEFAULT` to ensure that the pagefault is already handled:

```
...
switch(m_ptr->SVMCTL_PARAM) {
    case VMCTL_CLEAR_PAGEFAULT:
        assert(RTS_ISSET(p,RTS_PAGEFAULT));
        RTS_UNSET(p, RTS_PAGEFAULT);
        return OK;
    ...
```

2.2 Mum's Really Unfair!

- **What algorithm is used by default in Minix 3 to handle pagefault? Find its implementation and study it closely.**

  Just as hinted by the introduction part, Minix3 use **LRU** algorithm to handle pagefault. The implementation is written in `/servers/vm/pagefault.c` in `Minix 3.2.1`.

  - `pf_errstr`: a function used to transfer the detected `err` to `char*`
  - `do_pagefaults`:
    - this function first decode the input `message` to get the `endpoint`, `addr` & `err`
    - Then it first check whether the `endpoint` (the sender of the message) is valid.
    - After check `endpoint`, it checks whether the memory is valid, assert error if needed.
    - After checking whether memory is valid, if the process wants to `write` to the memory. Check whether the memory is writtable.
    - Then calculate the offset of memory.
    - After all these, handle the page fault accordingly and reactivate the process.
  - `do_memory`: parse the memory request. Check whether valid and call `handle_memory` if necessary.
  - `handle_memory`:
    - calculate the needed `page_size`
    - check the existence and the write access of the target memory.

        - allocate the memory and update necessary informations.
- **Use the top command to keep track of your used memory and cache, then run `time grep -r "mum" /usr/src`. Run the command again. What do you notice?**
    - First time: 8.01 real 0.25 user 5.00 sys
    - Second time: 2.45 real 0.73 user 1.71 sys
    - Notice: The `real` time & `sys` time drops dramatically while the `usr` time increases. As the search has just been carried out, due to `LRU` algorithm, it will replace an oldest page and when we carried out the second search, as it was cached, the search speed will be much faster and no need to handle more `page fault`, system time decreases.
- **Adjust the implementation of LRU into MRU and recompile the kernel.**
    - Replace almost all the `oldest` with `youngest` in `free_yielded`

```
vir_bytes free_yielded(vir_bytes max_bytes)
{

/* PRIVATE yielded_t *lru_youngest = NULL, *lru_oldest = NULL; */
    vir_bytes freed = 0;
    int blocks = 0;

    while(freed < max_bytes && lru_youngest) {
        SLABSANE(lru_youngest);
        freed += freeyieldednode(lru_youngest, 1);
        blocks++;
    }

    return freed;
}
```

- In `lmfs_get_block`, replace `front` with `rear`

```
struct buf *lmfs_get_block(
  register dev_t dev,       /* on which device is the block? */
  register block_t block,   /* which block is wanted? */
  int only_search       /* if NO_READ, don't read, else act normal */
)
{
/* Check to see if the requested block is in the block cache.  If so, return
 * a pointer to it.  If not, evict some other block and fetch it (unless
 * 'only_search' is 1).  All the blocks in the cache that are not in use
 * are linked together in a chain, with 'front' pointing to the least recently
 * used block and 'rear' to the most recently used block.  If 'only_search' is
 * 1, the block being requested will be overwritten in its entirety, so it is
 * only necessary to see if it is in the cache; if it is not, any free buffer
 * will do.  It is not necessary to actually read the block in from disk.
 * If 'only_search' is PREFETCH, the block need not be read from the disk,
 * and the device is not to be marked on the block, so callers can tell if
 * the block returned is valid.
 * In addition to the LRU chain, there is also a hash chain to link together
 * blocks whose block numbers end with the same bit strings, for fast lookup.
 */

  int b;
  static struct buf *bp, *prev_ptr;
  u64_t yieldid = VM_BLOCKID_NONE, getid = make64(dev, block);
```

```
assert(buf_hash);
assert(buf);
assert(nr_bufs > 0);

ASSERT(fs_block_size > 0);

assert(dev != NO_DEV);

/* Search the hash chain for (dev, block). Do_read() can use
 * lmfs_get_block(NO_DEV ...) to get an unnamed block to fill with zeros when
 * someone wants to read from a hole in a file, in which case this search
 * is skipped
 */
b = BUFHASH(block);
bp = buf_hash[b];
while (bp != NULL) {
  if (bp->lmfs_blocknr == block && bp->lmfs_dev == dev) {
      /* Block needed has been found. */
      if (bp->lmfs_count == 0) rm_lru(bp);
      bp->lmfs_count++;   /* record that block is in use */
      ASSERT(bp->lmfs_bytes == fs_block_size);
      ASSERT(bp->lmfs_dev == dev);
      ASSERT(bp->lmfs_dev != NO_DEV);
      ASSERT(bp->data);
      return(bp);
  } else {
      /* This block is not the one sought. */
      bp = bp->lmfs_hash; /* move to next block on hash chain */
  }
}

/* Desired block is not on available chain.  Take oldest block ('front'). */
if ((bp = rear) == NULL) panic("all buffers in use: %d", nr_bufs);

if(bp->lmfs_bytes < fs_block_size) {
  ASSERT(!bp->data);
  ASSERT(bp->lmfs_bytes == 0);
  if(!(bp->data = alloc_contig( (size_t) fs_block_size, 0, NULL))) {
      printf("fs cache: couldn't allocate a new block.\n");
      for(bp = rear;
          bp && bp->lmfs_bytes < fs_block_size; bp = bp->lmfs_next)
          ;
      if(!bp) {
          panic("no buffer available");
      }
  } else {
      bp->lmfs_bytes = fs_block_size;
  }
}

ASSERT(bp);
ASSERT(bp->data);
ASSERT(bp->lmfs_bytes == fs_block_size);
ASSERT(bp->lmfs_count == 0);

rm_lru(bp);
```

```
/* Remove the block that was just taken from its hash chain. */
b = BUFHASH(bp->lmfs_blocknr);
prev_ptr = buf_hash[b];
if (prev_ptr == bp) {
  buf_hash[b] = bp->lmfs_hash;
} else {
  /* The block just taken is not on the front of its hash chain. */
  while (prev_ptr->lmfs_hash != NULL)
      if (prev_ptr->lmfs_hash == bp) {
          prev_ptr->lmfs_hash = bp->lmfs_hash;    /* found it */
          break;
      } else {
          prev_ptr = prev_ptr->lmfs_hash; /* keep looking */
      }
}

/* If the block taken is dirty, make it clean by writing it to the disk.
 * Avoid hysteresis by flushing all other dirty blocks for the same device.
 */
if (bp->lmfs_dev != NO_DEV) {
  if (bp->lmfs_dirt == BP_DIRTY) flushall(bp->lmfs_dev);

  /* Are we throwing out a block that contained something?
   * Give it to VM for the second-layer cache.
   */
  yieldid = make64(bp->lmfs_dev, bp->lmfs_blocknr);
  assert(bp->lmfs_bytes == fs_block_size);
  bp->lmfs_dev = NO_DEV;
}

/* Fill in block's parameters and add it to the hash chain where it goes. */
MARKCLEAN(bp);        /* NO_DEV blocks may be marked dirty */
bp->lmfs_dev = dev;       /* fill in device number */
bp->lmfs_blocknr = block; /* fill in block number */
bp->lmfs_count++;     /* record that block is being used */
b = BUFHASH(bp->lmfs_blocknr);
bp->lmfs_hash = buf_hash[b];

buf_hash[b] = bp;     /* add to hash list */

assert(dev != NO_DEV);

/* Go get the requested block unless searching or prefetching. */
if(only_search == PREFETCH || only_search == NORMAL) {
  /* Block is not found in our cache, but we do want it
   * if it's in the vm cache.
   */
  if(vmcache) {
      /* If we can satisfy the PREFETCH or NORMAL request
       * from the vm cache, work is done.
       */
      if(vm_yield_block_get_block(yieldid, getid,
          bp->data, fs_block_size) == OK) {
          return bp;
      }
  }
}
```

```
  if(only_search == PREFETCH) {
    /* PREFETCH: don't do i/o. */
    bp->lmfs_dev = NO_DEV;
  } else if (only_search == NORMAL) {
    read_block(bp);
  } else if(only_search == NO_READ) {
    /* we want this block, but its contents
     * will be overwritten. VM has to forget
     * about it.
     */
    if(vmcache) {
        vm_forgetblock(getid);
    }
  } else
    panic("unexpected only_search value: %d", only_search);

  assert(bp->data);

  return(bp);           /* return the newly acquired block */
}
```

- recompile the kernel

  ```
  cd /usr/src
  make build
  reboot
  ```

- **Use the top command to keep track of your used memory and cache, then run `time grep -r "mum" /usr/src`. Run the command again. What do you notice?**
  - First time: 32.41 real 0.21 user 13.03 sys
  - Second time: 29.56 real 0.58 user 11.76 sys
  - Notice: The First time search is very very very slow compared to the `LRU` implementation. In the Second time, the running speed speed up to a very limited sense due to the property of `MRU` will replace the most recently updated page. As the most recently used page are replaced, doing repeated tasks won't increase.
- **Discuss the different behaviours of LRU and MRU as well as the consequences**
  - `LRU` replaces the least-recently-used pages while `MRU` will replace the most-recently-used pages. In most situation, due to locality and spatial locality, `LRU` will be better. But under the following situation, I think `MRU` will be better:
  - In video website, after the user have finished one video, it is less likely for them to see it again in a short time. Then `MRU` will be better. [4]