> Name:
>
> Liqin Zhang 517370910123
>
> Siwei Ye 517370910122

VE482 Lab8

Memory management at kernel level

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

  `vm` stands for virtual memory.

- Find all the places where the vm used inside the kernel. Why does it appear in so many different

  places?

  We find that the structure used by VM to pass data to the kernel while enabling paging is defined in `/include/arch/arm/include/vm.h`. It also includes the memory part of PM. (`fork()`, `exit()`)

  All files under `/servers/vm/` use `vm`. It is used when we need to run one or several programs using large amount of mem. Since there is not enough space in RAM, we can use VM to let the program imagine they have enough memory (in fact we are partially using disk space), thus helping the program to complete.

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

  We use `kmalloc` or `vmalloc` in kernel, `calloc` and `malloc` both requires the package `stdlib.h`, which can only be accessed from the user space.

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

  There are two ways. Message passing(mainly) and memory grants (for transferring larger amount of data).

  The message is mainly composed of

  - Endpoint: who is sending the message? e.g. `VM_PROC_NR` .
  - Type: What is the message about? e.g. `VM_PAGEFAULT` .
  - Other data

  There are three basic APIs:

  1. SEND: a message is sent, the sender is blocked until the message is delivered.
  2. RECEIVE: the process is blocked until a message is delivered to them,
  3. SENDEREC: a message is sent, the sneder is blocked until reply from the receiver

  The boundary is crossed using `brk` system call. It is bad consider putting `vm` in user space, as the idea of creating virtual space (usually hard disk) for the memory, each time we need to call `vm`, we have to cross the boundary, which is very resource consuming and dangerous considering the malicious attack from outside sources.

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

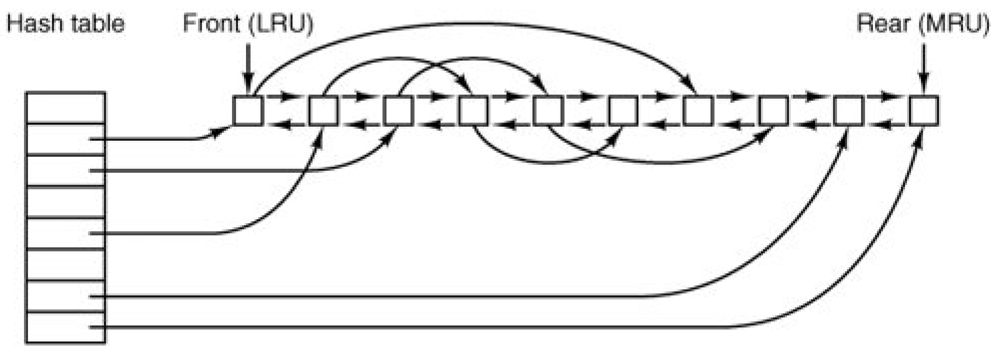
  1. The computer hardware traps to the kernel and program counter (PC) is saved on the stack. Current instruction state information is saved in CPU registers.

2. An assembly program is started to save the general registers and other volatile information to keep the OS from destroying it.
3. Operating system finds that a page fault has occurred and tries to find out which virtual page is needed. Sometimes hardware register contains this required information. If not, the operating system must retrieve PC, fetch instruction and find out what it was doing when the fault occurred.
4. Once virtual address caused page fault is known, system checks to see if address is valid and checks if there is no protection access problem.
5. If the virtual address is valid, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to remove a page.
6. If frame selected is dirty, page is scheduled for transfer to disk, context switch takes place, fault process is suspended and another process is made to run until disk transfer is completed.
7. As soon as page frame is clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
8. When disk interrupt indicates page has arrived, page tables are updated to reflect its position, and frame marked as being in normal state.
9. Faulting instruction is backed up to state it had when it began and PC is reset. Faulting is scheduled, operating system returns to routine that called it.
10. Assembly Routine reloads register and other state information, returns to user space to continue execution.

Mum's really unfair

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

  `LRU` implemented in `/server/vm/region.c`. Notice that it uses a LRU list to store various blocks, and always maintain a pointer to the youngest block and oldest block. The LRU list is a double-linked list, like what is shown below:

  When the pagefault occurs, the oldest page will be removed and the pointer will be point to the next element in the list.

```
static void lrucheck(void)
{
    yielded_t *list;

    /* list is empty and ok if both ends point to null. */
    if(!lru_youngest && !lru_oldest)
        return;

    /* if not, both should point to something. */
    SLABSANE(lru_youngest);
    SLABSANE(lru_oldest);
```

```
    assert(!lru_youngest->younger);
    assert(!lru_oldest->older);

    for(list = lru_youngest; list; list = list->older) {
        SLABSANE(list);
        if(list->younger) {
            SLABSANE(list->younger);
            assert(list->younger->older == list);
        } else  assert(list == lru_youngest);
        if(list->older) {
            SLABSANE(list->older);
            assert(list->older->younger == list);
        } else  assert(list == lru_oldest);
    }
}
```

- 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?

  Before running `time grep -r "mum" /usr/src`.

```
load averages:  0.00, 0.01, 0.01
45 processes: 1 running, 44 sleeping
main memory: 523836K total, 302252K free, 287004K contig free, 151844K cached
CPU states:   0.12% user,   1.14% system,   0.65% kernel,  98.09% idle
CPU time displayed ('t' to cycle): user ;  sort order ('o' to cycle): cpu

  PID USERNAME PRI NICE    SIZE STATE   TIME     CPU COMMAND
   -1 root       0        2613K         0:00   0.65% kernel
   10 root       1    0    228K         0:00   0.42% tty
    7 root       5    0   1052K         0:00   0.26% vfs
   12 root       2    0   6164K         0:00   0.20% vm
   89 service    7    0    120K         0:00   0.10% random
   27 root       7    0    836K   RUN   0:00   0.09% procfs
  165 root       7    0    580K         0:00   0.05% top
   36 service    5    0   6216K         0:00   0.04% mfs
   62 root       7    0    192K         0:00   0.04% devman
    5 root       4    0    248K         0:00   0.03% pm
   83 root       7    0    220K         0:00   0.03% devmand
  118 root       7    0    100K         0:00   0.00% vbox
    4 root       4    0   1372K         0:00   0.00% rs
    8 root       3    0    104K         0:00   0.00% memory
    9 root       2    0    144K         0:00   0.00% log
    3 root       4    0    164K         0:00   0.00% ds
```

  After running `time grep -r "mum" /usr/src.`

```
load averages:  0.30, 0.00, 0.01
47 processes: 2 running, 45 sleeping
main memory: 523836K total, 301676K free, 287004K contig free, 151844K cached
CPU states:  17.71% user,  17.31% system,   9.58% kernel,  55.40% idle
CPU time displayed ('t' to cycle): user ;  sort order ('o' to cycle): cpu

  PID USERNAME PRI NICE    SIZE STATE   TIME     CPU COMMAND
  167 root       8    0    356K   RUN   0:00  16.52% grep
   -1 root       0        2613K         0:00   9.58% kernel
    7 root       5    0   1052K         0:01   8.26% vfs
   56 service    5    0  37668K         0:00   3.46% mfs
   12 root       2    0   6196K         0:00   3.14% vm
   10 root       1    0    228K         0:01   1.26% tty
   36 service    5    0   6216K         0:00   0.78% mfs
  154 root       7    0   2396K         0:00   0.62% sshd
  108 service    7    0   1064K         0:00   0.40% inet
    5 root       4    0    248K         0:00   0.24% pm
   89 service    7    0    120K         0:00   0.08% random
   27 root       7    0    836K   RUN   0:00   0.08% procfs
  165 root       7    0    580K         0:00   0.05% top
  104 root       7    0    112K         0:00   0.05% lance
   62 root       7    0    192K         0:00   0.03% devman
   83 root       7    0    220K         0:00   0.03% devmand
```

It can detected that when running `time grep -r "mum" /usr/src`, the `grep` command takes 356K memory and its CPU usage is 16.52%.

- Adjust the implementation of LRU into MRU and recompile the kernel.

  To replace `LRU` with `MRU`, the function `free_yielded` in `region.c` is modified. I.e. free youngest node instead of oldest node in the list.

```
/*===========================================================================*
 *              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);
-   while(freed < max_bytes && lru_oldest) {
-       SLABSANE(lru_oldest);
-       freed += freeyieldednode(lru_oldest, 1);
        blocks++;
    }

    return freed;
}
```

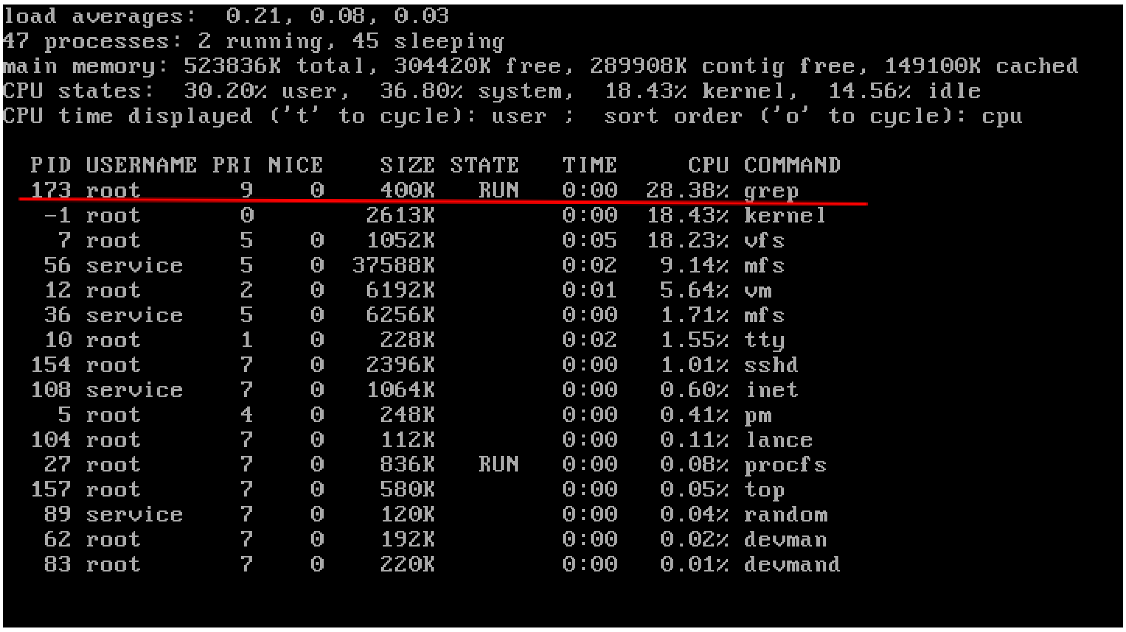
  Then recompile the kernel:

```
su
cd /usr/src
make build
```

- 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?

  After recompilation, the result when running `time grep -r "mum" /usr/src` is shown below:



```
load averages:  0.21, 0.08, 0.03
47 processes: 2 running, 45 sleeping
main memory: 523836K total, 304420K free, 289908K contig free, 149100K cached
CPU states:  30.20% user,  36.80% system,  18.43% kernel,  14.56% idle
CPU time displayed ('t' to cycle): user ;  sort order ('o' to cycle): cpu

  PID USERNAME PRI NICE    SIZE STATE   TIME     CPU COMMAND
  173 root       9    0    400K   RUN   0:00  28.38% grep
   -1 root       0        2613K         0:00  18.43% kernel
    7 root       5    0   1052K         0:05  18.23% vfs
   56 service    5    0  37588K         0:02   9.14% mfs
   12 root       2    0   6192K         0:01   5.64% vm
   36 service    5    0   6256K         0:00   1.71% mfs
   10 root       1    0    228K         0:02   1.55% tty
  154 root       7    0   2396K         0:00   1.01% sshd
  108 service    7    0   1064K         0:00   0.60% inet
    5 root       4    0    248K         0:00   0.41% pm
  104 root       7    0    112K         0:00   0.11% lance
   27 root       7    0    836K   RUN   0:00   0.08% procfs
  157 root       7    0    580K         0:00   0.05% top
   89 service    7    0    120K         0:00   0.04% random
   62 root       7    0    192K         0:00   0.02% devman
   83 root       7    0    220K         0:00   0.01% devmand
```

It can be detected that this time `grep` is taking more memory (400K) and more CPU usage (28.38%) .

- Discuss the different behaviours of LRU and MRU as well as the consequences for the users. Can

  you think of any situation where MRU would be better than LRU?

  In our above test, `MRU` is worse than `LRU` because `grep` is recursively searching the contents in directory `/usr/src`. The memory accessed are close to each other in this command, which means during the search the page accessed are very likely to be accessed again in the future. If we use `MRU`, we freed the most recently accessed memory (youngest), then the frequency of page fault rapidly increases.

  `MRU` would be better than `LRU` when the process the working on something that does not contain so much repeating.

---

1. cs.uttyler.edu↩
2. professormerwyn.wordpress.com↩