Creating a Simple Virtual Machine in C

(With paging of course)

Goal:

Today's society is complex and fast and dare I say overwhelming. In times like these it is normal to yearn simpler times. Maybe for this reason our 4th homework is to implement a virtual machine with its own "operating system". The basic functionalities are similar to a 16 bit VM made by **Andrei Ciobanu**.

The homework is to add paging for memory virtualization. This is needed to have more than one program in our virtual machine, not that far simpler times. The functions to fill were provided, the job of this vm is to virtualize the memory for processes. That is almost all of it.

Helper Functions And Definitions

```c
// YOUR CODE STARTS HERE
size_t getFileSize(FILE *file) {
    fseek(file, 0, SEEK_END); // Move to the end of the file
    size_t size = ftell(file); // Get the position (file size)
    rewind(file); // Reset the file pointer to the beginning
    return size / sizeof(uint16_t); // Return size in terms of 16-bit words
}
#define VALID_MASK (0x0001)
#define WRITE_MASK (0x0004)
#define PTE_SHIFT (11)
#define PCB_START (12)
uint16_t translateAddress(uint16_t va);


uint16_t getPCB(uint16_t pid)
{
  uint16_t pcb = PCB_START + pid * PCB_SIZE;
  return pcb;
}
uint16_t getPageTableStart(uint16_t pid)
{
  //they start from 3rd table
  uint16_t _pageTableStart = PAGE_SIZE;
  return (_pageTableStart + pid * 32);
}
```

getFileSize is a simple function that returns the size of a file by going to the end of the file, getting that position returning the total number of byte in function, resetting the file pointer and than returning how many 16bits the file is by dividing the size by sizeof(uint16_t). This is needed for the load image provided. By knowing the file size it will write to the position other allocMem function chooses.

getPCB function returns the start of Process Control Block that belongs to the specified pid.

getPageTableStart will find the Page Table belonging to pid. The page tables start from the 3rd page and are reserved 32 per process.

```c
uint16_t getPTE(uint16_t va)
{
  uint16_t pcb_address = getPCB(mem[Cur_Proc_ID]);
  uint16_t pte_start = mem[pcb_address + PTBR_PCB];
  uint16_t vpn = va >> PTE_SHIFT; // Extract VPN

  return mem[(pte_start + vpn)];
}
uint16_t translateAddress(uint16_t va) {
    uint16_t pcb_address = getPCB(mem[Cur_Proc_ID]);
    uint16_t pte_start = mem[pcb_address + PTBR_PCB];

    uint16_t vpn = va >> PTE_SHIFT; // Extract VPN
    uint16_t offset = va & 0x07FF; // Offset within the page

    //printf("ptbr is %d vpn is ")

    //printf("va is %d and pte_start is %d and vpn is %d",va, pte_start, vpn );
    uint16_t pte = pte_start + vpn;

    if (!(mem[pte] & VALID_MASK)) { // Check if the page is valid
    //printf("vpn is %d\n", vpn);
        fprintf(stderr, "Invalid page table entry for pte: 0x%04X\n", pte); //vpn 0dan başlıyo
        exit(1); // Terminate to prevent further invalid accesses
    }
    pte = mem[pte];
    uint16_t phys_frame = pte >> PTE_SHIFT;
    //printf("translated number is %d \n", ((phys_frame << PTE_SHIFT) + offset));
    return (phys_frame << PTE_SHIFT) + offset;
}
```

getPTE returns the page table entry. First finds the process control block, then the page table starting point, and returns the page table entry of the virtual address.

translateAddress get a virtual address. Finds the virtual page number and offset inside the page. We find the pte and check if it is valid. İf it is we get the physical page number from it. And return the correct physical memory location.

Definitions

In page table entries the least significant bit is the valid bit. Since I have seen many times in slides the use of macro MASKS in these type of low levelish stuff I preferred a definition for it.

Again for writing permission bit is the 3rd from right. Sice 0x0004 is 0000 0000 0000 0100 it will be more readable for these checks.

Page table entries keep the page num in first 5 bits. To access them we have to shift to right 11 times. Since it is cooler to have a macro, I opted for it.

Process Control Block starts at 12th word, however it isn't provided in macros. This one is for ease of remembering.

Main functions

İnitOs

Most of this is what is specified in the homework. The ideas are, if CurProcId is 0xFFFF it means no active process. There are 0 processes yet so the count is zero. Os_Status aims to signal the memory filling, that is not the case at the start. There are two freebitmaps holding 32 pages the system has. First 3 is full because they are reserved for system. Hence first free bitmap is 0001 1111 1111 1111.

```
void initOS() {
    mem[Cur_Proc_ID] = 0xFFFF;
    mem[Proc_Count] = 0x0000;
    mem[OS_STATUS] = 0x0000;
    mem[OS_FREE_BITMAP] = 0x1FFF; //first three pages are full
    mem[OS_FREE_BITMAP+1] = 0xFFFF;
    return;
}
```

CreateProc

Aim is to create process out of the given code and heap parts of the process.

```c
int createProc(char *fname, char *hname) {
  if (mem[OS_STATUS] & VALID_MASK){
    //os_stat is not zero, it is full
    printf("The OS memory region is full. Cannot create a new PCB.\n");
    return -1;
  }
  uint16_t pid = mem[Proc_Count];
  mem[Proc_Count] += 1;
  if (mem[Proc_Count] == 1361)
  {
    mem[OS_STATUS] = 0xFFFF;
  }
  uint16_t self_pcb = getPCB(pid);
  mem[self_pcb + PID_PCB] = pid;
  mem[self_pcb + PC_PCB] = 0x3000; //processes start from 0x3000
  mem[self_pcb + PTBR_PCB] = getPageTableStart(pid);
  FILE *code_file = fopen(fname, "r");
  if (!code_file){
      fprintf(stderr, "Cannot read code file: %s\n", fname);
      return -1;
  }
  uint16_t offsets[2];
  for (int i = 0; i < CODE_SIZE; i++){
    uint16_t worked = allocMem(mem[self_pcb + PTBR_PCB], i + 6, 0xFFFF, 0);
    if (worked){
      offsets[i] = (mem[mem[self_pcb + PTBR_PCB] + i + 6] >> PTE_SHIFT) << PTE_SHIFT;
    }
    else{
      printf("Cannot create code segment.\n");
      return -1;
    }
  }
  size_t writ = getFileSize(code_file);
  fclose(code_file);
  ld_img(fname, offsets,writ);
  for (int i = 0; i < HEAP_INIT_SIZE; i++){
```

This function is called by the main.c, for every given input from terminal. The user enters the name of the code followed by the name of the heap. These make up a process. We first check OS_Status. And then if this is the final possible process set the Os_Status. We fill out the process control block for this new process. Open the file for code, and allocate CODE_Size(2) pages. Allocation takes the start of page table registery, virtual page number, read permission and write permission. Virtual pages start from 6 because of the way they are implemented.Processes think the first ever part of their code starts at 0x3000. İf it allocates we keep record of it for the load image function.Load the image for code, and then do the same to heap.

Load Proc and AllocMem

```c
void loadProc(uint16_t pid) {

  mem[Cur_Proc_ID] = pid;

  uint16_t pcb = getPCB(pid);
  reg[RPC] =  mem[pcb + PC_PCB];
  reg[PTBR] = mem[pcb + PTBR_PCB];

}
//for example the first code starts with 0x3000 shifted 11 is 6, it is the first though. VPN should be 5 less than first 5 bits
//pre: ptbr is base register for page table, vpn is the first 5 bit of virt address
uint16_t allocMem(uint16_t ptbr, uint16_t vpn, uint16_t read, uint16_t write) {
    for (uint16_t i = 0; i < 32; i++) {
        uint16_t *bitmap = (i < 16) ? &mem[OS_FREE_BITMAP] : &mem[OS_FREE_BITMAP + 1];
        uint16_t bit = 0x8000 >> (i % 16); // Mask for current page frame

        // Check if the page is free
        if (*bitmap & bit) {
            *bitmap &= ~bit; // Mark page as used in the bitmap

            // Calculate and store the PTE in the page table
            //printf("write is %d", write);
            uint16_t pte = (i << PTE_SHIFT) | ((read == 0xFFFF) << 1) | ((write == 0xFFFF) << 2) | VALID_MASK;
            //printf("new pte is 0x%04X\n", pte);
            mem[ptbr + vpn] = pte; // Write PTE directly at ptbr + vpn
            return 1; // Allocation successful
        }
    }
    fprintf(stderr, "No free page available.\n");
    return 0; // Failure
}
```

Load process is simple. Since the system remembers the process by keeping two of its properties in registers(our array based pseudo register) and the id of process in OS. We set them according to the given id.

In allocMem arguments are start of the page table registry, virtual page number, permission for read and write.

First find a place in freebitmaps that keep track of it. Since there are two words that keep track of it according to which physical page we choose we take the first bitmap or second. And the bit we will use is 1000 0000 0000 0000 shifted right according to order.

We check until we find a free one. After that we create the page table entry. First five bits is the physical page number. And then we put the bits to their respective places. For example if read is permitted the variable is 0xFFFF. İf it is shift 1 left and add it to the pte. Put that pte to its memory spot.

Edit Selection View Go Run Terminal Help ege.balyali [SSH: 10.3.0.91]

vm.c differences2.txt

pa4 > C vm.c

```c
static inline void tbrk() {
  uint16_t vpn = reg[R0] >> PTE_SHIFT;
  uint16_t pcb_start = getPCB(mem[Cur_Proc_ID]);
  uint16_t pte_start = mem[pcb_start + PTBR_PCB];

  if ((mem[pte_start + vpn] & VALID_MASK) == 0) {
    //not allocated
    printf("Heap increase requested by process %d.\n", mem[Cur_Proc_ID]);
    //printf("regr0 is %d", reg[R0]);
    allocMem(mem[pcb_start + PTBR_PCB], vpn, (reg[R0] & 0x0002 ? UINT16_MAX : 0), (reg[R0] & 0x0004 ? UINT16_MAX : 0));
  }
  else{
    freeMem(vpn, mem[pcb_start + PTBR_PCB]);
  }
}
```

Tbrk is called to increase or decrease the heap. How do we know which one? The r0 will keep the page specified by the process. If it is already allocated deallocate it, else if it is new allocate it.

```c
330 static inline void tyld() {
331   uint16_t pcb_start = getPCB(mem[Cur_Proc_ID]);
332   mem[pcb_start + PC_PCB] = reg[RPC];
333   mem[pcb_start + PTBR_PCB] = reg[PTBR];
334   uint16_t original_start = pcb_start;
335   pcb_start += PCB_SIZE;
336   while(pcb_start != original_start)
337   {
338     if((pcb_start + PID_PCB) == PCB_START && mem[pcb_start + PID_PCB] != 0xFFFF){
339       //printf("line331We are switching from process %d to %d.\n",mem[Cur_Proc_ID],mem[pcb_start + PID_PCB] );
340       printf("We are switching from process %d to %d.\n",mem[Cur_Proc_ID],mem[pcb_start + PID_PCB] );
341       loadProc(mem[pcb_start + PID_PCB]);
342       return;
343     }
344     if (mem[pcb_start + PID_PCB] != 0 && mem[pcb_start + PID_PCB] != 0xFFFF)
345     {
346       printf("We are switching from process %d to %d.\n",mem[Cur_Proc_ID],mem[pcb_start + PID_PCB] );
347       loadProc(mem[pcb_start + PID_PCB]);
348       return;
349     }
350     pcb_start = (pcb_start + PCB_SIZE) % (1360 * PCB_SIZE);
351     if (pcb_start == 0)
352       pcb_start = PCB_START;
353   }
354   //printf("came to line 345, it will call itself");
355   loadProc(mem[original_start + PID_PCB]);
356 }
```

Tyld is called when the process wants to give other processes a cahance to run. If there is none it calls itself. We get the process control block of the current process. And increase it to access the next process control block until another one is found. At line 350 we ensure the wrap around by taking mod of it with the process control block size times amount of processes that can exist(almost 1360).

If at any point the new process control block has a working pid we load that process with loadProc. İf none is found and the loop comes full circle meaning there is no other active process, vm loads the original process.

thalt

```c
static inline void thalt() {
  uint16_t pcb_start = getPCB(mem[Cur_Proc_ID]);
  mem[pcb_start + PID_PCB] = 0xFFFF;
  uint16_t pte_start = mem[pcb_start + PTBR_PCB];
  //printf("pte location %d and pte is %d\n", (pte_start + i), mem[pte_start + i]);
  for ( int i = 0;i < 32; i++){
    if ((mem[pte_start + i] & VALID_MASK) == 1){
      freeMem(i, mem[(pcb_start + PTBR_PCB)]);
    }
  }
  uint16_t i = PCB_START;
  uint16_t no_process_left = 1;
  if (mem[PCB_START + PID_PCB] != 0xFFFF){
    no_process_left = 0;
  }
  while(no_process_left && (i < (PCB_SIZE * 32 + PCB_START)))
  {
    if (mem[i + PID_PCB] != 0x0000 && mem[i + PID_PCB] != 0xFFFF){
      no_process_left = 0;
      break;
    }
    i += PCB_SIZE;
  }
  if (no_process_left)
  {running = false;}
  else{
    changeProcAfterHalt();
  }
}
```

Thalt is called by processes when they are finished. We changed that processes id to 1111 1111 1111 1111 to indicate it is finished. For every page it has we free the memory. Than we need to find if all processes ended. First we make no_process_left true. İf any process is found that has a legit id no_process_left becomes false. İf there aren't any process left vm raises the flag running=false and the vm stops. Else we change the process. I used to call tyield since the job is similar. However, this shouldn't print so another similar function is called.

```c
static inline void changeProcAfterHalt() {
  //there isn't wrap around the pte table
  uint16_t pcb_start = getPCB(mem[Cur_Proc_ID]);
  uint16_t original_start = pcb_start;
  pcb_start += PCB_SIZE;

  while(pcb_start != original_start)
  {
    //printf("original_start is %d pcb_start is at %d ",original_start,pcb_start);
    if((pcb_start + PID_PCB) == PCB_START && mem[pcb_start + PID_PCB] != 0xFFFF)
    {
      loadProc(mem[pcb_start + PID_PCB]);
      return;
    }
    if (mem[pcb_start + PID_PCB] != 0 && mem[pcb_start + PID_PCB] != 0xFFFF)
    {
      loadProc(mem[pcb_start + PID_PCB]);
      return;
    }
    pcb_start = (pcb_start + PCB_SIZE) % (1360 * PCB_SIZE);
    if (pcb_start == 0)
      pcb_start = PCB_START;
  }
  //printf("came to line 345, it will call itself");
  loadProc(mem[original_start + PID_PCB]);
}
```

This is the function called by thalt when there are other processes The idea is the same as tyield.

Mr and Mw

```c

static inline uint16_t mr(uint16_t address) {
  //printf("mr calls translate with %d", address);
  uint16_t phys_address = translateAddress(address);
  return mem[phys_address];
}

static inline void mw(uint16_t address, uint16_t val) {
  //printf("mw calls translate with %d", address);

  if ((getPTE(address) & WRITE_MASK) == 0)
  {
    printf("Cannot write to a read-only page.\n");
    exit(1);
  }
  uint16_t phys_address = translateAddress(address);
  mem[phys_address] = val;
}

// YOUR CODE ENDS HERE
```

Memory read translates virtual address and returns the intended information in physical memory.

Memory write is similar with the difference of checking if it is a writable page.