CS307 PA4

NAME: UTKU

SURNAME: GENC

SUID: 30611

CONTENTS

What Is the Point Of This PA4?

The point of this PA4 is to learn about memory virtualization. Every application thinks that it owns the memory, and they are the only one. However this is not true. Behind the scenes, OS controls the real memory locations, so that applications does not have to think about it. To do that, inside the **PCB** (Process Control Block) it stores **page tables.**

What Is Virtual Memory?

Virtual memory is the address that is only sensible for the application. What is that mean? It means that application thinks it access a data in a memory location (0x3000 lets say), but in reality the address of stored data may be different. And application does not know that, and it does not have to.

When a data stored in the memory, it has an address. This real address only known by OS. The application only have an virtual address, and when it request something, it request via the virtual address. The OS takes this virtual address, looks at the **page table** of that application, and finds the real address. As you can see, appication thinks it has a linear and continuos memory, but in reality it does not have to. So that we can achieve virtualization.

How Did I Solve The Problem Of Virtualization Of Memory?

I will explain my solution function by function

- initOS()

```
-  void initOS() {
-    mem[0] = 0xffff;
-    mem[1] = 0x0000;
-    mem[2] = 0x0000;
-    mem[3] = 0b0001111111111111;
-    mem[4] = 0b1111111111111111;
-    return;
-  }
```

In initOS, i initialize the memory location for the OS. The details of what they are can be found in PA4 explaination.

- createProc(char *fname, char *hname)

```
-  int createProc(char *fname, char *hname) {
-    if ((mem[2] & 0b0000000000000001) == 1)
-    {
-      printf("The OS memory region is full. Cannot create a new PCB.\n");
-      running = 0;
-      return 0;
-    }
-
-    // mem[1] is total process count.
-    uint16_t index = mem[1];
-    if (index == 0) // This is the first program.
-    {
-
-      mem[12 + index * 3] = 0x0000;        // Initialize process id
-      mem[(12 + index * 3) + 1] = 0x3000; // Initialize PC
-      mem[(12 + index * 3) + 2] = 4096;    // Initialize page_table_entry
-
-    }
-    else
-    {
-      mem[12 + index * 3] = mem[12 + (index - 1) * 3] + 1;   // Initialize process id (+1 bigger)
-      mem[(12 + index * 3) + 1] = 0x3000;                    // Initialize PC
-      mem[(12 + index * 3) + 2] = 4096 + index * 32;         // Initialize page_table_entry
-    }
-
-    int16_t found_code1 = -1;
-    int16_t found_code2 = -1;
-    int16_t found_heap1 = -1;
-    int16_t found_heap2 = -1;
-
```

```
-     // Memory allocation is vpn.
-     found_code1 = allocMem(mem[(12 + index * 3) + 2], 6, 0xffff, 0x0000);
    // Try to allocate first available page for code.
-
-     if (found_code1 != -1)
-     {
-       found_code2 = allocMem(mem[(12 + index * 3) + 2], 7, 0xffff,
   0x0000);  // Try to allocate second available page for code.
-
-       if (found_code2 != -1)
-       {
-         found_heap1 = allocMem(mem[(12 + index * 3) + 2], 8, 0xffff,
   0xffff);
-         if (found_heap1 != -1)
-         {
-           found_heap2 = allocMem(mem[(12 + index * 3) + 2], 9, 0xffff,
   0xffff);
-           if (found_heap2 != -1)
-           {
-             uint16_t code_offsets[2] = {found_code1 * 2048, found_code2 *
   2048};  // Physical addresses.
-             uint16_t heap_offsets[2] = {found_heap1 * 2048, found_heap2 *
   2048};  // Physical addresses.
-             uint16_t code_size = get_file_size(fname);  // File size
-             uint16_t heap_size = get_file_size(hname);  // File size
-
-             ld_img(fname, code_offsets, code_size);
-             ld_img(hname, heap_offsets, heap_size);
-
-             mem[1] = mem[1] + 1;  // Increase process number.
-             return 1;
-           }
-         }
-       }
-     }
-
-     if ( (found_code1 == -1) || (found_code2 == -1) )
-     {
-       if (found_code1 != -1)
-       {
-         freeMem(6, mem[(12 + index * 3) + 2] = 0x0000);
-       }
-       printf("Cannot create code segment.\n");
-     }
-     if ( (found_heap1 == -1) || (found_heap2 == -1) )
-     {
-       freeMem(6, mem[(12 + index * 3) + 2] = 0x0000);
-       freeMem(7, mem[(12 + index * 3) + 2] = 0x0000);
-       if (found_heap1 != -1)
```

```
      {
        freeMem(8, mem[(12 + index * 3) + 2] = 0x0000);
      }
      printf("Cannot create heap segment.\n");
    }

    // Free the allocated memory.
    mem[12 + index * 3] = 0x0000;
    mem[(12 + index * 3) + 1] = 0x0000;
    mem[(12 + index * 3) + 2] = 0x0000;
    return 0;
  }
```

Inside createProc, we try to allocate memory. First we check whether the OS region is full or not. If full, we terminate.

Than we get the total process count and initialize the PCB of the process. After that we try to allocate memory. If for any reason the memory could not be allocated, we free the allocated memory and PCB, and give the error print.

- loadProc(uint16_t pid)

```
void loadProc(uint16_t pid) {
    mem[0] = pid;                            // Set current process.
    reg[PTBR] = mem[(12 + pid * 3) + 2];   // Set page_table_base_register.
    reg[RPC] = mem[(12 + pid * 3) + 1];    // Set PC for this process.
  }
```

This function takes a pid and loads it. With pid we can easily reach the PCB. In PCB we take the page table and process counter. Set the register.

- allocMem(uint16_t ptbr, uint16_t vpn, uint16_t read, uint16_t write)

```
uint16_t allocMem(uint16_t ptbr, uint16_t vpn, uint16_t read, uint16_t write) {

    // Check wheter is it allocated or not.
    if ( (mem[ptbr + vpn] & 0x0001) == 1)
    {
      printf("Cannot allocate memory for page %d of pid %d since it is already allocated.\n", vpn, mem[0]);
      return 0;
    }


    int16_t found = -1;
    uint16_t mask = 0b1000000000000000;

    for (int i = 0; i < 16; i++)
    {
      // mem[3] is free page map.
      if ( ((mem[3] & mask) >> (15 - i)) == 1)
      {
```

```
-       found = i;
-       mem[3] = mem[3] & ~(0b0000000000000001 << (15 - i)); // Update the free page.
-       break;
-     }
-     else
-     {
-       mask = mask >> 1; // Look next bit.
-     }
-   }
-
-   if (found == -1) // There is no free space in the first 16 pages.
-   {
-
-     mask = 0b1000000000000000;  // Set the mask.
-     for (int i = 0; i < 16; i++)
-     {
-       if ( ((mem[4] & mask) >> (15 - i)) == 1)
-       {
-         found = i + 16;
-         mem[4] = mem[4] & ~(0b0000000000000001 << (15 - i));
-         break;
-       }
-       else
-       {
-         mask = mask >> 1;
-       }
-     }
-   }
-
-   if (found != -1)
-   {
-
-     if (read == UINT16_MAX)
-     {
-       if (write == UINT16_MAX)
-       {
-          mem[ptbr + vpn] = (found << 11) + 0b0000000000000111;
-          //printf("Page table entry is written with this value (WR): %d\n", mem[ptbr + vpn]);
-       }
-       else
-       {
-         mem[ptbr + vpn] = (found << 11) + 0b0000000000000011;
-         //printf("Page table entry is written with this value (R): %d\n", mem[ptbr + vpn]);
-       }
-     }
-     else
```

```
-       {
-         if (write == UINT16_MAX)
-         {
-            mem[ptbr + vpn] = (found << 11) + 0b0000000000000101;
-            //printf("Page table entry is written with this value (W):
   %d\n", mem[ptbr + vpn]);
-         }
-         else
-         {
-           mem[ptbr + vpn] = (found << 11) + 0b0000000000000000;
-           //printf("Page table entry is written with this value (INV):
   %d\n", mem[ptbr + vpn]);
-         }
-       }
-
-       // If all pages are allocated: Set the OS status to 1.
-       if (found == 31)
-       {
-         mem[2] = 0x0001;
-       }
-
-       // Return the PF
-       return (found);
-     }
-
-     // There is no free space.
-     printf("Cannot allocate more space for pid %d since there is no free
   page frames.\n", mem[0]);
-     return 0;
-   }
```

In allocMem we try to allocate memory. First we check wheter is it already allocated by the process. If it is we return 0. Else we continue.

We try to find a free page first. When we find we write it to the memory with corresponding page and vpn. Also we consider write and read parameters.

If there is no free space in the memory. We print error message.

- freeMem(uint16_t vpn, uint16_t ptbr)

```
-   int freeMem(uint16_t vpn, uint16_t ptbr) {
-     // The page is already freed.
-     if ((mem[ptbr + vpn] & 0x0001) == 0)
-     {
-       return 0;
-     }
-     else
-     {
-       uint16_t temp = mem[ptbr + vpn] >> 11; // physical page number;
-       if (temp >= 16)
```

```
-       {
-         temp = temp - 16;
-         mem[4] = mem[4] ^ (0b0000000000000001 << (15 - temp));
-       }
-       else
-       {
-         mem[3] = mem[3] ^ (0b0000000000000001 << (15 - temp));
-       }
-       // Update the page entry INVALID.
-       mem[ptbr + vpn] = mem[ptbr + vpn] & 0b1111111111111110;
-
-       // Set the OSstatus to 0 again. (A page is freed)
-       mem[2] = 0x0000;
-       return 1;
-     }
-   }
```
We free the memory allocated by alloc memory.We take the vpn and page table base register and calculate the real address. Then free it: making the valid bit 0.

- Tbrk()

```
-   static inline void tbrk() {
-     uint16_t address = reg[R0];
-
-     uint16_t vpn = address >> 11;
-     uint16_t request = address & 0x0001;                // If 1
    allocate, if 0 free the vpn.
-     uint16_t read = (address & 0b0000000000000010) >> 1;  // If 1, read,
    else no read.
-     uint16_t write = (address & 0b0000000000000100) >> 2;
-     if (read == 1) {read = 0xffff;}
-     else read = 0x0000;
-
-     if (write ==1 ) write = 0xffff;
-     else write = 0x0000;
-
-     // Write and read it set.
-     if (request == 1) // Allocate memory
-     {
-       printf("Heap increase requested by process %d.\n", mem[0]);
-       allocMem(reg[PTBR], vpn, read, write);
-     }
-     else // Free memory
-     {
-       printf("Heap decrease requested by process %d.\n", mem[0]);
-       uint16_t success = freeMem(vpn, reg[PTBR]);
-       if (success == 0)
-       {
```

```
-         printf("Cannot free memory of page %d of pid %d since it is not allocated.\n", vpn, mem[0]);
-       }
-     }
-   }
```

This function requests new pages or visa versa.

- Yield()

```
- static inline void tyld() {
-   uint16_t current = mem[0];
-   uint16_t old = mem[0];
-
-   // Save the current PC.
-   mem[(12 + current * 3) + 1] = reg[RPC];
-
-   // Go to next process.
-   current = (current + 1) % mem[1];
-
-   while ((mem[(12 + current * 3)] != mem[0]) && (mem[(12 + current * 3)] == 0xffff))
-   {
-     current = (current + 1) % mem[1];
-   }
-
-   // Set the current process.
-   mem[0] = current;
-
-   // Retrive the PC and Page_table_base_register
-   reg[PTBR] = mem[(12 + current * 3) + 2];
-   reg[RPC] = mem[(12 + current * 3) + 1];
-
-   // Inform the switch.
-   printf("We are switching from process %d to %d.\n", old, mem[0]);
- }
```

This function yields the control the another process. It updates the registers for this. Then print the message that it is changing processes.

- Thalt()

```
- static inline void thalt() {
-
-   uint16_t current = mem[0];
-   uint16_t old = mem[0];
-
-   // Go to the next process
-   current = (current + 1) % mem[1];
-
-   while ((mem[(12 + current * 3)] != mem[0]) && (mem[(12 + current * 3)] == 0xffff))
-   {
-     current = (current + 1) % mem[1];
```

```
-     }
-
-     if (current == old) // If the next process is the same.
-     {
-       // Set it to 0xffff
-       mem[12 + current * 3] = 0xffff;
-
-       // For everypage that is allocated, free the page.
-       uint16_t current_page_entry = mem[(12 + current * 3) + 2];
-       for (int i = 0; i < 32; i++)
-       {
-         if (mem[current_page_entry] != 0)
-         {
-           freeMem(i, mem[(12 + current * 3) + 2]);
-         }
-         current_page_entry++;
-       }
-
-       // Stop the machine.
-       running = false;
-     }
-     else
-     {
-       // There is still process to run.
-       // Set the PCB to 0xffff;
-       mem[12 + old * 3] = 0xffff;
-
-       uint16_t current_page_entry = mem[(12 + old * 3) + 2];
-       for (int i = 0; i < 32; i++)
-       {
-         if (mem[current_page_entry] != 0)
-         {
-
-           freeMem(i, mem[(12 + old * 3) + 2]);
-         }
-         current_page_entry++;
-       }
-
-
-       mem[0] = current;       // Set the new process
-
-
-       // Restore PC and page_table_base_register.
-       reg[PTBR] = mem[(12 + current * 3) + 2];
-       reg[RPC] = mem[(12 + current * 3) + 1];
-     }
-   }
```

This is used to stop a process. It basically free all the memory allocated by the process. Than set the registers to next available process. If there is no any other process, it simply shut down the VM.

- Mr(uint16_t address)

```
static inline uint16_t mr(uint16_t address) {
  uint16_t vpn = address >> 11;

  if (vpn <= 5)
  {
    printf("Segmentation fault.\n");
    running = 0;
    return -1;
  }
  if ((mem[reg[PTBR] + vpn] & 0b0000000000000001) == 0)
  {
    printf("Segmentation fault inside free space.\n");
    running = 0;
    return -1;
  }
  if ((mem[reg[PTBR] + vpn] & 0b0000000000000010) == 0)
  {
    printf("Cannot read from a write-only page.\n");
    running = 0;
    return - 1;
  }
  uint16_t offset = address & 0b0000011111111111;
  uint16_t PF = mem[reg[PTBR] + vpn] >> 11;

  return mem[(PF << 11) + offset];
}
```

The address in here is virtual. It hase a page number and an offset. First we calculate the vpn. And go the the corresponding page table to find real address. Then read the memory if it is allowed.

- Mw(uint16_t address, uint16_t val)

```
static inline void mw(uint16_t address, uint16_t val) {
  // Address is virtual.
  // Take the page table.
  uint16_t vpn = address >> 11;
  //printf("This is the page table register value and vpn: %d, %d\n", reg[PTBR], vpn);
  //printf("This is the page table entry: %d.\n", mem[reg[PTBR] + vpn]);
  if (vpn <= 5)
  {
    printf("Segmentation fault.\n");
    running = 0;
    return;
  }
  if ((mem[reg[PTBR] + vpn] & 0b0000000000000001) == 0)
```

```
-     {
-       printf("Segmentation fault inside free space.\n");
-       running = 0;
-       return;
-     }
-
-     if ((mem[reg[PTBR] + vpn] & 0b0000000000000100) == 0)
-     {
-       printf("Cannot write to a read-only page.\n");
-       running = 0;
-       return;
-     }
-     uint16_t offset = address & 0b0000011111111111;
-     uint16_t PF = mem[reg[PTBR] + vpn] >> 11;
-
-     mem[(PF << 11) + offset] = val;
-   }
```
The same thing (almost) with memory read)