Operating Systems Course WS07/08
Team: FtpOs

Friedle Karin, Pöhr Florian & Taferl Johann

February 15, 2008

Abstract

This document was created within an operating systems-course held by Prof. Kirsch at the university of Salzburg during the winter semester 07/08. Our task was to implement an operating system with the focus on fundamental concepts such as support for concurrency, memory management, synchronization, scheduling, virtual memory and file system.

Contents

1 Project - Introduction

We decided to use GeekOS 0.3.0 in order to implement an operating system. GeekOS is an open source tiny operating system kernel. For emulating an x86 machine we have chosen the Bochs emulator 2.3. As GeekOS code is written in C we had to adapt ourself and use C as our programming language. In general we followed the instructions of the GeekOS handbook and realized the several projects step-by-step. This document just describes our contributions to the existing GeekOS code and not the whole concept of GeekOS.

2 Milestones

The table below shows the milestones we had to achieve on the way to our operating system and how we separated the work:

handling the ELF-format	Pöhr Florian
adding processes - support for user mode	Taferl Johann
memory management	Team FtpOS
thread synchronization - semaphores	Friedle Karin
scheduling	Friedle Karin
virtual memory - paging	Pöhr Florian, Taferl Johann
file system	Team FtpOS

We all met each other up to two times a week in order to discuss essential themes, next steps and the operating system architecture. This also included many times of pair-programming especially if one of us had a problem or different parts of code had to be tested together. In order to keep our sources up to date for each member, we used subversion as revision control system.

3 ELF Executable File

After fulfilling this project we were able to load ELF executable files from disk into memory. Our job was to parse an executable file and fill in appropriate structures so that the GeekOS-loader knows about the program's sections and can lay out segments in memory.

3.1 ELF Format

Informations about the ELF format which are worth to know:
An ELF executable file is divided into sections: a text section for the code, and the data section for global variables. Moreover the ELF file contains headers that describe how these sections should be stored in our memory. By using these headers we determined what the executable image should be for the loaded program so that the loader can load and execute the program correctly.

3.2 Approach

- First of all we had to read the ELF Header, which is always at the beginning of the file. The header gave us the information about how the rest of the file is laid out.
- Then we searched for the Program Headers in order to know where in the file we find the text and data sections and where they should end up in the executable image.
- GeekOS offered us data types for structures which we needed to match the format of the ELF and program headers.
- The following function parses the ELF headers and fills out the Exe_Format structure:

```
/**
 * @param exeFileData buffer containing the executable file
 * @param exeFileLength length of the executable file in bytes
 * @param exeFormat structure describing the executable's segments
 *   and entry address; to be filled in
 */

int Parse_ELF_Executable(
        char *exeFileData,
        ulong_t exeFileLength,
        struct Exe_Format *exeFormat)
```

3.3 Implementation-Steps

1. mapping the elfHeader to the exeFileData

```
typedef struct {
```

```
    unsigned  char ident[16];
    unsigned  short type;
    unsigned  short machine;
    unsigned  int version;
    unsigned  int entry;
    unsigned  int phoff;
    unsigned  int sphoff;
    unsigned  int flags;
    unsigned  short ehsize;
    unsigned  short phentsize;
    unsigned  short phnum;
    unsigned  short shentsize;
    unsigned  short shnum;
    unsigned  short shstrndx;
} elfHeader;

 elfHeader* eh = (elfHeader*) exeFileData
```

2. identifying the file as an ELF object file

```
eh->ident[0] == 0x7F &&
eh->ident[1] == 'E' &&
eh->ident[2] == 'L' &&
eh->ident[3] == 'F'
```

3. checking the file's class and capacity

```
0 ... invalid class
2 ... 64-bit objects

eh->ident[4] != 0 ||
eh->ident[4] != 2
```

4. identifying the object file type (executable file)

```
#define ET_EXEC 2

eh->type == ET_EXEC
```

5. identifying the correct machine type (Intel architecture)

```
#define EM_386 3

eh->machine == EM_386
```

6. setting the number of exeSegments, phnum holds the number of entries in the program header table

```
struct Exe_Format {
    struct Exe_Segment segmentList[EXE_MAX_SEGMENTS];
    int numSegments;
    ulong_t entryAddr;
};

exeFormat->numSegments = eh->phnum
```

7. setting the entry address, entry gives the virtual address to which the system first transfers control, thus starting the process.

```
exeFormat->entryAddr = eh->entry
```

8. checking if the file has a program header table.

```
eh->phoff != 0
```

9. setting the nextprogramHeaderOffset, phoff holds the program header table's file offset in bytes.

```
int nextProgramHeaderOffset = (eh->phoff)
```

10. looping over all exeSegments

    (a) the number of segments must be less the maximum numbers set for the exeFormat

    ```
    #define EXE_MAX_SEGMENTS 3

    i < EXE_MAX_SEGMENTS
    ```

(b) mapping the programHeader to the exeFileData

```
typedef struct {
    unsigned  int   type;
    unsigned  int   offset;
    unsigned  int   vaddr;
    unsigned  int   paddr;
    unsigned  int   fileSize;
    unsigned  int   memSize;
    unsigned  int   flags;
    unsigned  int   alignment;
} programHeader;

programHeader* ph =
      (programHeader*) (exeFileData+nextProgramHeaderOffset)
```

(c) filling in the exeFormat, offset gives the offset from the beginning of the file at which the first byte of the segment resides, fileSize gives the number of bytes in the file image of the segment, vaddr gives the virtual address at which the first byte of the segment resides in memory, memSize gives the number of bytes in the memory image of the segment, flags gives flags relevant to the segment.

```
(exeFormat->segmentList[i]).offsetInFile = ph->offset
(exeFormat->segmentList[i]).lengthInFile = ph->fileSize
(exeFormat->segmentList[i]).startAddress = ph->vaddr
(exeFormat->segmentList[i]).sizeInMemory = ph->memSize
(exeFormat->segmentList[i]).protFlags  = ph->flags
```

(d) getting the address of the next programHeader, phentsize holds the size in bytes of one entry in the file's program header table. All entries are the same size.

```
nextProgramHeaderOffset += eh->phentsize
```

4 User Mode Support

We controll the parts of memory that can be accessed when user code is running and we limit the set of machine operations that the user code can execute. A program that is running in this sort of controlled way is said to be running in user mode.

4.1 Running User Processes

Function:
int Spawn(const char *program, const char *command, struct Kernel_Thread **pThread)
is for launching user programs:

- program - is the full path of the program executable file
- command - is the command, including name of program and arguments
- pThread - reference to Kernel_Thread pointer where a pointer to the newly created user mode thread (process) is stored

1. We load the entire executable into a memory buffer:

   ```
   Read_Fully(program, &buf, &fileLen);
   ```

2. We parse the executable in order to get an Exe_Format data structure describing how the executable should be loaded:

   ```
   Parse_ELF_Executable(buf, fileLen, &exeFormat);
   ```

3. We prepare a new address space for the program and create a User_Context with the loaded program:

   ```
   Load_User_Program(buf, fileLen, &exeFormat, command, &pUserContext);
   ```

4. We start a new User Thread with the User Context:

   ```
   Start_User_Thread(pUserContext, 0);
   ```

5. We return the process-id of the new thread:

   ```
   (*pThread)->pid;
   ```

4.2 Load User Program

This theme is described in the paging section.

4.3 Creating a User-Context

The User-Context structure stores all of the information that we need to setup and run a user thread.
Function: Create_User_Context(ulong_t size)

1. We allocate size memory:

```
context = (struct User_Context*)Malloc(sizeof(struct User_Context));
context->refCount = 0;
```

2. We create a local descriptor table for the process and create a selector that contains the location of the LDT descriptor within the GDTthe selector:

```
context->ldtDescriptor = Allocate_Segment_Descriptor();
context->ldtSelector = Selector(KERNEL_PRIVILEGE,
         true, //segmentIsInGDT
         Get_Descriptor_Index(
         context->ldtDescriptor));
```

3. We initialize the LDT in the GDT:

```
Init_LDT_Descriptor( context->ldtDescriptor,
context->ldt,
         NUM_USER_LDT_ENTRIES);
```

4. We create descriptor for the code segment of the user program and add the descriptor to the LDT. Moreover we create a selector that contains the location of the descriptor within the LDT:

```
Init_Code_Segment_Descriptor(&context->ldt[0], //segment descriptor
         USER_VM_START, //baseAddr
         0x80000, //numPages
         USER_PRIVILEGE //privilegeLevel
);
context->csSelector = Selector(USER_PRIVILEGE,
false, //segmentIsNotInGDT
         0 //index of the segment descriptor
);
```

5. Same procedure for the data segment:

```
Init_Data_Segment_Descriptor(&context->ldt[1],
         USER_VM_START,
         0x80000,
         USER_PRIVILEGE
);
context->dsSelector = Selector(USER_PRIVILEGE,
         false,
         1
);
```

4.4 Start User Thread

The functions for the user thread are similar to the functions for the kernel thread.
We just present the user part in this section.
Function:
struct Kernel_Thread* Start_User_Thread(struct User_Context* userContext, bool detached)

1. We create a new thread: kthread = Create_Thread(PRIORITY_NORMAL, detached);
   a) We allocate one page for the thread context object and one page for its stack.
   b) We initialize the stack pointer of the new thread and account info:
   `Init_Thread(kthread, stackPage, priority, detached);`

```
static int nextFreePid = 1;
struct Kernel_Thread* owner =
              detached ? (struct Kernel_Thread*)0 : g_currentThread;
memset(kthread, '\0', sizeof(*kthread));
kthread->stackPage = stackPage;
kthread->esp = ((ulong_t) kthread->stackPage) + PAGE_SIZE;
kthread->numTicks = 0;
kthread->priority = priority;
kthread->userContext = 0;
kthread->owner = owner;

/*
 * The thread has an implicit self-reference.
 * If the thread is not detached, then its owner
```

```
         * also has a reference to it.
         */
        kthread->refCount = detached ? 1 : 2;

        kthread->alive = true;
        Clear_Thread_Queue(&kthread->joinQueue);
        kthread->pid = nextFreePid++;

        if(kthread->priority == PRIORITY_IDLE)
          kthread->currentReadyQueue = MAX_QUEUE_LEVEL-1;
        else
          kthread->currentReadyQueue = 0;

        kthread->blocked = false;

        kthread->semaphorBitset = Create_Bit_Set(NR_OF_SEMAPHORS);
    }
```

c) We add the thread to the list of all threads in the system.

```
Add_To_Back_Of_All_Thread_List(&s_allThreadList, kthread);
```

2. We get the thread ready to execute in user mode:
   Setup_User_Thread(kthread, userContext);

   a) We attach the user context to the kernel thread:
      Attach_User_Context(kthread, userContext);

   b) We set up the initial thread stack.

3. In order to schedule the process for execution:
   `Make_Runnable_Atomic(kthread);`

```
int currentQ = kthread->currentReadyQueue;
kthread->blocked = false;
Enqueue_Thread(&s_runQueue[currentQ], kthread);
```

4.5 Context-Switching

To give the impression of processes running concurrently, the kernel switches contexts among processes, thus giving each process a small time quantum to run. The kernel threads have a

NULL userContext, whereas for user threads, this context is non-NULL. For every context switch, the kernel calls `Switch_To_User_Context()`:

1. We switch to a new address space by loading this processes LDT:

   ```
   Switch_To_Address_Space(kthread->userContext);
   ```

   a) We switch to the LDT of the new thread:

   ```
   Load_LDTR(userContext->ldtSelector);
   Set_PDBR(userContext->pageDir);
   ```

2. We move the stack pointer up one page:

   ```
   Set_Kernel_Stack_Pointer((ulong_t)kthread->stackPage + PAGE_SIZE);
   ```

5 Scheduling

We added three scheduling algorithms to our operating system in order to handle the threads:

- Round Robin algorithm (RR)
- Multilevel Feedback algorithm (MLF)
- priority based algorithm (Prio)

5.1 Round Robin

The Round Robin algorithm holds a FIFO queue where all threads sit. If a new thread is created or the thread's quantum of time is over it will be added to the end of the queue.

5.2 Multilevel Feedback

Instead of having only one queue we have 4 queues here. Each queue represents one priority level and all threads within this queue have the same priority. The priorities range from 0 (highest priority) to 3 (lowest priority).

5.2.1 Approach

- A new thread is placed on the highest priority queue (0) But the Idle Thread is placed on the lowest level queue and is never permitted to move out of that level.
- Each time a thread completes a full quantum, it is placed on the run queue with the next lowest priority level, until it reaches priority level 3, at which point it cannot go any lower.
- When a thread becomes blocked, it has to go into a higher priority queue, until it reaches priority level 0, at which point it cannot go any higher.

5.2.2 Scheduling a Thread

1. Look at the highest priority queue.
2. If there exists threads, choose the highest priority (highest numbered) process in the queue.

```
#define PRIORITY_IDLE    0
#define PRIORITY_USER    1
#define PRIORITY_LOW     2
#define PRIORITY_NORMAL  5
#define PRIORITY_HIGH   10
```

3. If there are no threads there, go to the next lowest priority queue, and keep repeating until you find a thread.

5.3 Prio

The algorithm just chooses the thread with the highest priority number no matter in which queue it is.

5.4 Switching the Scheduler

- We can switch the used scheduling algorithm via a system call
  a) 0 ... RR
  b) 1 ... MLF

c) 2 ... Prio

```
static int Sys_SetSchedulingPolicy(struct Interrupt_State* state)
{
  if(state->ecx < 2 || state->ecx > 100)
  {
    Print("quantum of ticks out of bounds (2 <= %d <= 100)\n", state->ecx);
    return EINVALID;
  }
  if(state->ebx >= 0 && state->ebx <=2)
  {
    setSchedulingPolicy(state->ebx);
  }
  g_Quantum = state->ecx;
  return 0;
}
```

- MLF is the initial scheduler.

5.5 Implementation of Scheduler

- s_runQueue is an array of structs, one for each priority level
- fields in the struct Kernel_Thread which are important for the scheduler:

```
struct Kernel_Thread {
  int priority;
  // is needed to control for how long the thread is running
  volatile ulong_t numTicks;
  /*
  * The run queue level that the thread should be put on
  * when it is restarted.
  */
  int currentReadyQueue;
  bool blocked;

  struct Bit_Set *semaphorBitset;
};
```

- When a thread becomes blocked in Wait(), it has to go into a higher priority queue this means currentReadyQueue gets decremented.

```
void Wait(struct Thread_Queue* waitQueue)
{
  struct Kernel_Thread* current = g_currentThread;

  if(schedPolicy == SCHED_RR)
  { //RR
    current->blocked = true;
    current->currentReadyQueue = 0;
    Add_To_Back_Of_Thread_Queue(waitQueue, current);
  }
  else
  { //MLF and PRIO (= others)
    /* put thread into queue with next higher priority. */
    if(current->currentReadyQueue > 0 && current->priority > PRIORITY_IDLE)
      current->currentReadyQueue --;

    /* Add the thread to the wait queue. */
    current->blocked = true;
    Enqueue_Thread(waitQueue, current);
  }

  /* Find another thread to run. */
  Schedule();
}
```

- function Scheduler(void) finds the next thread to run:

```
void Schedule(void)
{
  struct Kernel_Thread* runnable;

  /* Get next thread to run from the run queue */
  runnable = Get_Next_Runnable();

  /* Activate the new thread, saving the context of the current thread.
```

```
  Switch_To_Thread(runnable);
}
```

- function Get_Next_Runnable(void) represents the real scheduler:

```
/*
 * Number of ready queue levels.
 */
#define MAX_QUEUE_LEVEL 4

...

/*
* Get the next runnable thread from the run queue.
* This is the scheduler.
*/
struct Kernel_Thread* Get_Next_Runnable(void)
{
  int i = 0;
  int bestQueue = -1;
  struct Kernel_Thread* best = NULL;

  if(schedPolicy == SCHED_MLF)
  {
    // Find the best thread from the highest-priority run queue
    for(i=0; i<MAX_QUEUE_LEVEL; i++)
    {
      struct Kernel_Thread* tmp = NULL;
      /*
      * Find the best (highest priority) thread in given
      * thread queue.
      */
      tmp = Find_Best(&s_runQueue[i]);

      if(tmp != NULL && (best == NULL || (best->priority == PRIORITY_IDLE)))
      {
        best = tmp;
        bestQueue = i;
```

```
      }
      if(best == 0 && tmp != 0)
      {
        best = tmp;
        bestQueue = i;
      }
    }
  }
  else if(schedPolicy == SCHED_RR)
  {
    best = (&s_runQueue[0])->head;
    bestQueue = 0;
  }
  else
  { //PRIO
    struct Kernel_Thread* tmp = NULL;

    for(i=0; i<MAX_QUEUE_LEVEL; i++)
    {
      tmp = Find_Best(&s_runQueue[i]);

      if(tmp != NULL && (best == NULL || tmp->priority > best->priority))
      {
        best = tmp;
        bestQueue = i;
      }
    }
  }

  //remove thread from queue
  Remove_From_Thread_Queue(&s_runQueue[bestQueue], best);

  return best;
}
```

- system call for switching the algorithm: int Set_Scheduling_Policy(int policy, int quantum)

a) the quantum of time can be set at runtime and is identical for all threads in all four queues.

b) quantum = number of ticks that a user thread may run before getting removed from the processor

c) integers must be from the interval [2,100]

d) If a thread runs out of time, we inform the interrupt return code that we want to choose a new thread:

```
static void Timer_Interrupt_Handler(struct Interrupt_State* state)
{
 struct Kernel_Thread* current = g_currentThread;

 if (current->numTicks >= g_Quantum)
 {
   g_needReschedule = true;
   /*
   * The current process is moved to a lower priority queue,
   * since it consumed a full quantum.
   */
   if (current->currentReadyQueue < (MAX_QUEUE_LEVEL - 1))
   {
     if(getSchedulingPolicy() == SCHED_RR)
       current->currentReadyQueue = 0;
     else
       current->currentReadyQueue ++;
   }
 }
 End_IRQ(state);
}
```

6 Synchronization

In order to enable thread sychronization among different threads we added four new system calls that provide user programs with semaphores:

```
static int Sys_CreateSemaphore(struct Interrupt_State* state)
{
```

```
    return sem_create(state->ebx, state->ecx, state->edx, g_currentThread);
}

static int Sys_P(struct Interrupt_State* state)
{
    return sem_p(state->ebx, g_currentThread);
}

static int Sys_V(struct Interrupt_State* state)
{
    return sem_v(state->ebx, g_currentThread);
}

static int Sys_DestroySemaphore(struct Interrupt_State* state)
{
    return sem_destroy(state->ebx, g_currentThread);
}
```

6.1 Creating a Semaphore

The current thread's request for using a semaphore is handled by sem_create(). A thread can not call sem_p() or sem_v() unless it calls sem_create(). The user provides a name for the semaphore, as well as the semaphore's initial value, and will get a free semaphore ID returned, an integer between 0 and N - 1. Our operating system handles at least 20 (thus N = 20) semaphores whose names may be up to 25 characters long.

```
struct semaphor
{
  int semCounter;
  int threadCounter;
  char* semName;
  struct Thread_Queue waitingThreads;
};

#define NR_OF_SEMAPHORS 20
#define MAX_SEM_NAME_LENGTH 25

int sem_create(uint_t name, uint_t nameLength,
```

```
                uint_t initCount,
                struct Kernel_Thread* caller)
{
  int i;
  int freeSem = -1;

  //copy name from user space to kernel space
  char* semName = Malloc(nameLength + 1);
  Copy_From_User(semName, name, nameLength);
  *(semName + nameLength) = 0;

  //search for semaphor (and save the next free semaphor, if sem does not exists)
  for(i=0; i< NR_OF_SEMAPHORS; i++)
  {
    if(semaphors[i].semName == NULL)
    {
      if(freeSem < 0) //save the first free semaphor, not the last
      freeSem = i;
    }
    else if(!strncmp(semaphors[i].semName, semName, MAX_SEM_NAME_LENGTH))
    {
      if(!Is_Bit_Set(caller->semaphorBitset, i))
      {
        semaphors[i].threadCounter ++;
        Set_Bit(caller->semaphorBitset, i);
      }
      Free(semName);
      return i;
    }
  }
  //semaphor with given name not found; create new
  if(freeSem >= 0)
  {
    semaphors[freeSem].semName = semName;
    semaphors[freeSem].semCounter = initCount;
    semaphors[freeSem].threadCounter = 1;
    Set_Bit(caller->semaphorBitset, freeSem);
  }
```

```
  return freeSem;
}
```

We check if another thread has made this system call with the same name. If so, we return the semaphore ID associated with this name. The returned SID value will allow the calling thread to tell the kernel which semaphore it wants to use later. We also add this SID to the list of semaphores the calling thread can use, as well as increment the count of registered threads which are permitted to use the semaphore.
If this is the first time Create_Semaphore has been called by the name passed in, then we search for an unused SID, and initialize the value of the semaphore variable to the given value. Again, we add the SID to the list of semaphores the current thread can use, as well as incrementing the semaphore's count of authorized threads.

6.2 Wait() and Signal()

Whenever a thread calls sem_p() or sem_v(), we check if the thread has permission to make this call. This is done by checking if the thread has the SID in its list of SIDs that it can access. If it is there, it will be allowed to execute sem_p() or sem_v().
When waiting on a semaphore operation, the thread may not use a busy wait. Instead, to block a thread, we use the Wait function. We have a thread queue for threads blocked on semaphore operations. To wakeup one thread/all threads waiting on a given semaphore, we use the function Wake_Up().

```
int sem_p(int semId, struct Kernel_Thread* caller)
{
  if(semaphors[semId].semName == NULL || !Is_Bit_Set(caller->semaphorBitset, semId))
  {
    return EINVALID;
  }
  while(true)
  {
    if(semaphors[semId].semCounter > 0)
    {
      semaphors[semId].semCounter --;
      break;
    }
    Wait(&semaphors[semId].waitingThreads);
  }
  return 0;
```

```
}

int sem_v(int semId, struct Kernel_Thread* caller)
{
  if(semaphors[semId].semName == NULL || !Is_Bit_Set(caller->semaphorBitset, semId))
  {
    return EINVALID;
  }
  semaphors[semId].semCounter ++;
  Wake_Up(&semaphors[semId].waitingThreads);
  return 0;
}
```

6.3 Destroying a Semaphore

Destroy_Semaphore(int sem) removes the passed semaphore from the list of semaphores the calling thread is allowed to use. It also keeps track of how many threads have references to the semaphore, and delete the semaphore from the table when the last thread that can access this semaphore calls Destroy_Semaphore().
When a thread exits, the kernel automatically calls Destroy_Semaphore() on behalf of this thread, for all the semaphores it has in its list.

```
int sem_destroy(int semId, struct Kernel_Thread* caller)
{
  if(semId < 0 || semId >= NR_OF_SEMAPHORS || semaphors[semId].semName == NULL
     || !Is_Bit_Set(caller->semaphorBitset, semId))
  {
    return EINVALID;
  }
  Clear_Bit(caller->semaphorBitset, semId);
  semaphors[semId].threadCounter --;

  if(semaphors[semId].threadCounter == 0)
  {
    Free(semaphors[semId].semName);
    semaphors[semId].semName = NULL;
  }
  return 0;
```

```
}
```

6.4 Testing

6.4.1 Get System Time

One way to compare scheduling algorithms is to see how long it takes a process to complete from the time of creation to the termination of the process. Get_Time_Of_Day() syscall returns the value of the kernel global variable g_numTicks.
We call it once at the beginning of the thread (in the user code) and once at the end. Then we can calculate how long it took the thread to run.

6.4.2 Scheduler

The file workload.c is used for testing spawn with arguments and the scheduling algorithm:

```
/c/workload.exe [algorithm] [quantum]
```

The algorithm can take any of the values rr,mlf or prio and quantum is the scheduling algorithm's quantum.

6.4.3 Producer-Consumer Problem

Launching the files ping.c and pong.c concurrently creates the well known producer-consumer problem:

```
/c/ping.exe &
/c/pong.exe
```

7 Paged Virtual Memory

The purpose of paging is to provide a mapping from a user address to a physical address. Paging is similar to segmentation because it allows each process to run in its own memory space.

7.1 Concept

- In order to add paging to our operating system we implemented a two-level paging system that includes page tables and directories.

- Each page directory and page table are also pages in the system.
- Each page directory/page table entry is 4 bytes long and each table contains 1024 entries.
- Each page directory contains a pointer to a page table which in turn contains pointers to the physical memory.

7.2 Memory Layout

We have a single page table for all kernel only threads, and a page table for each user process. In addition, the page tables for user mode processes also contain entries to address the kernel mode memory.

- Kernel memory starts at: VA 0x0000 0000
- Data/Text start at page: VA 0x8000 0000
- Initial stack at top of page: VA 0xFFFF E000
- Args in page: VA 0xFFFF F000
- Memory space ends at: VA 0xFFFF FFFF
- So we have 2 GB for Kernel Space and 2 GB for User Space.

Segmentation allows the user space process to think that its virtual location 0 is the 2GB point in the page layout.

7.3 Approach

Each memory access by a user process involves two steps:

1. The base address of the user process will be added to the user memory address.
2. The resulting linear address will be mapped to the physical address using the page directory and page tables.

Mapping: Each memory address in the x86 system is 32 bits long. When the system sees an access to a virtual address, it will first find the page directory currently in use. It will then use the most significant 10 bits to index into the page directory. The page directory entry is used to find the appropriate page table. The next 10 bits are used to index the page table. Finally, the page table entry is used to find the base of the physical memory page. The last 12 bits of the virtual memory address are used to index into the physical page.

7.4 Kernel Memory Mapping

The kernel memory is a one to one mapping of all of the physical memory in the processor. The page table entries for this memory are marked so that this memory is only accessible from kernel mode (this means that the flags bit in the page directory and page table do not contain VM_USER).

7.4.1 Initialization of the Virtual Memory

Init_VM(struct Boot_Info *bootInfo):

1. We build the kernel page directory and the page tables; memSizeKB/4 calculates the number of needed pages:

```
typedef struct {
    uint_t present:1;
    uint_t flags:4;
    uint_t accesed:1;
    uint_t reserved:1;
    uint_t largePages:1;
    uint_t globalPage:1;
    uint_t kernelInfo:3;
    uint_t pageTableBaseAddr:20;
} pde_t;

pageDir = (pde_t*) initPageDir(bootInfo->memSizeKB/4);
```

uint_t initPageDir(uint_t nrOfPages)

a) We calculate the needed number of entries for the page directory; the max. number of entries for a page table is 1024:
`nrOfPageDirEntries = ((nrOfPages-1) / 1024) + 1;`
If the number of pages is less then 1024 we would get 0 as result for the number of entries in the page directory, so we have to calclulate the result plus 1.
If we have 1024 pages we just need one entry, so we calculate the pages minus 1 in order to get just one entry as result.

b) We allocate a page for the page directory and get the physical address back:
pageDir = (ulong_t)Alloc_Page();

c) We set the first entry in the page directory to the address of the page directory:
`pageDirEntry = (pde_t*)pageDir;`

d) For all needed entries in the page directory, we set the following fields:

```
for(i=0; i<nrOfPageDirEntries; i++)
{
  *((uint_t*)pageDirEntry) = 0;

  pageDirEntry->present = 1;
  pageDirEntry->flags = VM_WRITE;
  pageDirEntry->globalPage = 1;
  pageDirEntry->pageTableBaseAddr =
    initPageTable(&pageAddr, min(NUM_PAGE_TABLE_ENTRIES, nrOfPages)) >> 12;
  nrOfPages -= NUM_PAGE_TABLE_ENTRIES;
  pageDirEntry ++;
}
```

uint_t initPageTable(uint_t* pageAddr, uint_t nrOfPages)

I We allocate a page for the page table and get the physical address back:
`pageTable = (ulong_t) Alloc_Page();`

II We set the first entry in the page table to the address of the page table:

```
typedef struct {
    uint_t present:1;
    uint_t flags:4;
    uint_t accesed:1;
    uint_t dirty:1;
    uint_t pteAttribute:1;
    uint_t globalPage:1;
```

```
    uint_t kernelInfo:3;
    uint_t pageBaseAddr:20;
} pte_t;

pageTableEntry = (pte\_t*) pageTable;
```

III For all needed entries in the page table we set the following fields:

```
for(i=0; i<nrOfPages; i++)
{
  *((uint_t*)pageTableEntry) = 0;

  pageTableEntry->present = 1;
  pageTableEntry->flags = VM_WRITE | VM_NOCACHE;
  pageTableEntry->globalPage = 1;
  pageTableEntry->pageBaseAddr = (*pageAddr) >> 12;
  pageTableEntry ++;
  (*pageAddr) += PAGE_SIZE;
}
```

IV If there are page table entries left, we set them to 0.

V We return the page table.

e) If there are page directory entries left, we set them to 0.

f) We return the page directory.

2. We do not map a page at address 0, so we can detect null pointer exceptions:

```
pte_t* pageTable = (pte_t*)(((pde_t*)pageDir)->pageTableBaseAddr << 12);
pageTable->present = 0;
```

3. To enable paging for the first time, we call Enable_Paging(pageDir) with the kernel page directory.

4. We install an interrupt handler for the page fault (interrupt 14):
Install_Interrupt_Handler(14, Page_Fault_Handler);

7.5 User Memory Mapping

This is done when we load the user program into address space:

```
int Load_User_Program(char *exeFileData, ulong_t exeFileLength,
                      struct Exe_Format *exeFormat, const char *command,
                      struct User_Context **pUserContext)
```

1. We copy the first half of the kernel page table into the user page table. This is done to use the same pageDirectory for kernel mode and for user mode.

   ```
   ulong_t pageDir = (ulong_t)Alloc_Page();

   //create userContext and set relevant fields
   *pUserContext =  Create_User_Context();
   (*pUserContext)->pageDir = (void*)pageDir;
   (*pUserContext)->size = USER_VM_SIZE;
   (*pUserContext)->entryAddr = exeFormat->entryAddr;
   //copy the kernel stuff to the user page directory
   memcpy((void*)pageDir, (void*)getKernelPageDir(), PAGE_SIZE >> 1);
   //clear the rest of the user directory
   memset((void*)pageDir + (NUM_PAGE_DIR_ENTRIES >> 1), 0, PAGE_SIZE >> 1);
   ```

2. We copy the segments to the pages:

   ```
   for(i=0; i<exeFormat->numSegments; i++)
   {
     //get the fields of one segment
     ulong_t fileOffset = exeFormat->segmentList[i].offsetInFile;
     ulong_t fileLength = exeFormat->segmentList[i].lengthInFile;
     ulong_t memLength  = exeFormat->segmentList[i].sizeInMemory;
     ulong_t memOffset  = exeFormat->segmentList[i].startAddress;

     //allocate the needed pages;
     allocPages(pageDir,
                memOffset + USER_VM_START,
                memLength
     );
     //copy segment to the pages
     copyToPages(pageDir,
                 memOffset + USER_VM_START,
                 exeFileData + fileOffset,
   ```

```
                fileLength
  );
}
```

3. Determine the amount of memory required for the argument block, and determine how many arguments there are.

   ```
   Get_Argument_Block_Size(command, &numArgs, &argBlockSize);
   ```

4. We allocate pages for the stack:

   ```
   ulong_t destAddr =
     USER_VM_SIZE - Round_Up_To_Page(argBlockSize) - DEFAULT_USER_STACK_SIZE;
   (*pUserContext)->stackPointerAddr = destAddr + DEFAULT_USER_STACK_SIZE;
   allocPages(pageDir, destAddr + USER_VM_START, DEFAULT_USER_STACK_SIZE);
   ```

5. We allocate pages for the argument block

   ```
   destAddr = Round_Down_To_Page(USER_VM_SIZE - argBlockSize);
   (*pUserContext)->argBlockAddr = destAddr;
   allocPages(pageDir, destAddr + USER_VM_START, argBlockSize);
   char* argBlockAddr = Malloc(argBlockSize);

   //Format a user process command line argument block.
   Format_Argument_Block(
   argBlockAddr, //kernel addres of formated arg block
   numArgs,      //number of command line arguments
   destAddr,     //user address of argument block
   command       //the command that called the programm
   );

   copyToPages(pageDir, destAddr + USER_VM_START, argBlockAddr, argBlockSize);
   Free(argBlockAddr);
   ```

7.6 Page-Fault-Handler

Reasons for a Page Fault:

- accessing a page, which is not allocated (write error)

- accessing a page, which is paged out (read error)

void Page_Fault_Handler(struct Interrupt_State* state)

1. Update the clock of the current thread (because of LRU algorithm)
2. Get the address that caused the page fault:
   address = Get_Page_Fault_Address();
3. Get the fault code:

   ```
   faultCode = *((faultcode\_t *) &(state->errorCode));
   ```

4. Decide if faultCode.writeFault or faultCode.readFault.

7.7 WriteFault

1. We try to allocate new pages:

   ```
   allocPages((ulong_t)userContext->pageDir,
              Round_Down_To_Page(address),
              PAGE_SIZE
             );
   ```

2. If we are not successful then we get an out-of-memory return value.
   This exceptions already indicate, that there are no free pages on disk to page out a page from memory.

7.7.1 Writing to the Paging File

void Write_To_Paging_File(void *paddr, ulong_t vaddr, int pagefileIndex)

1. Get the page: `page = Get_Page((ulong_t) paddr);`
2. Loop over the sectors:
   `SECTORS_PER_PAGE = PAGE_SIZE / SECTOR_SIZE`

   a) Write the block for each sector:

   ```
   Block_Write( pd->dev,
                pagefileIndex*SECTORS_PER_PAGE + i + (pd->startSector),
                paddr+i*SECTOR_SIZE
              );
   ```

3. Set the bit for the page: Set_Bit(bitmap,pagefileIndex);

7.8 ReadFault

int PageFault(pde_t * pagedir, ulong_t vaddr)

1. Get the entry in the pagetable:
   pageTableEntry = getPageTableEntry(pagedir, vaddr);
2. The page is not present and its content is in the paging file that means:

   ```
   pageTableEntry->kernelInfo == KINFO_PAGE_ON_DISK
   ```

3. Get the address of the page (index in the page file):

   ```
   pageFileIndex = pageTableEntry->pageBaseAddr;
   ```

4. Get the physical address of a new page:
   `paddr = Alloc_Pageable_Page(pageTableEntry, Round_Down_To_Page(vaddr));`
5. Set the base address of the page:

   ```
   pageTableEntry->pageBaseAddr = (ulong_t) paddr >> 12;
   ```

6. Read from the disk:
   a) enable interrupts
   b) reading: Read_From_Paging_File(paddr, Round_Down_To_Page(vaddr), pageFileIndex);
   c) disable interrupts
7. Free the disk space: Clear_Bit(bitmap, pagefileIndex);

7.8.1 Reading from the Paging File

void Read_From_Paging_File(void *paddr, ulong_t vaddr, int pagefileIndex)

1. Get the wanted page: `page = Get_Page((ulong_t) paddr);`
2. Set the page as not pageable (so it can't be stolen):

   ```
   page->flags = page->flags & ~PAGE_PAGEABLE;
   ```

3. Loop over the sectors:
   `SECTORS_PER_PAGE = PAGE_SIZE / SECTOR_SIZE`

   a) Read the block for each sector:

   ```
   Block_Read(pd->dev,
              pagefileIndex*SECTORS_PER_PAGE + i + (pd->startSector),
              paddr+i*SECTOR_SIZE
              );
   ```

4. Clear the bit for the page: Clear_Bit(bitmap,pagefileIndex);

5. Set the virtual address of the page:

   ```
   page->vaddr = vaddr;
   ```

6. Get the entry in the page table:

   ```
   page->entry = getPageTableEntry(g_currentThread->userContext->pageDir, vaddr);
   ```

7. Set the page present:

   ```
   page->entry->present=1;
   ```

8. Set the page back as pageable:

   ```
   page->flags |= PAGE_PAGEABLE;
   ```

7.9 Allocating Pages

int allocPages(ulong_t pageDir, ulong_t destAddr, ulong_t size)

1. We split the destination address for pagedirectory-entry and pagetable-entry:
   `pageDirAddr, pageTabAddr, offsetAddr.`

2. Get the pagedirectory-entry: `pde_t *pageDirEntry = (pde_t*) pageDir + pageDirAddr;`

3. Get the number of pages to allocate:

   ```
   uint_t nrOfPages = ((size + offsetAddr - 1) / PAGE_SIZE) + 1;
   ```

4. Loop over the entries in the page table:

   a) If the entry in the page directory is present, then we create the entry in the page table:

   ```
   pageTabEntry = (pte_t*)(pageDirEntry->pageTableBaseAddr << 12);
   ```

   b) Otherwise we allocate a page, delete the content, format the entry in the page directory:

   ```
   pageTabEntry = (pte_t*)Alloc_Page();
   memset(pageTabEntry, 0, PAGE_SIZE);

   pageDirEntry->present = 1;
   pageDirEntry->flags = VM_USER | VM_WRITE | VM_READ;
   pageDirEntry->globalPage = 0;
   pageDirEntry->pageTableBaseAddr = (ulong_t)pageTabEntry >> 12;
   ```

   c) If the address of the page table is bigger then 1024, we switch to the next entry in the page directory:

   ```
   pageDirAddr ++;
   pageTabAddr = 0;
   pageDirEntry = (pde_t*) pageDir + pageDirAddr;
   pageTabEntry = (pte_t*)(pageDirEntry->pageTableBaseAddr << 12);
   ```

   d) If the entry in the page table is not present, we allocate a pageable page and get an address of a page:

   ```
   pageAddr = (ulong_t)Alloc_Pageable_Page(pageTabEntry,
              (pageDirAddr << 22) | (pageTabAddr << 12));
   *((uint_t*)pageTabEntry) = 0;
   pageTabEntry->present = 1;
   //VM_WRITE=1 ... Memory is writable
   //VM_USER=2 ... Memory is accessible to user code
   //VM_READ=0 ... Memory can be read (ignored for x86)
   pageTabEntry->flags = VM_USER | VM_WRITE | VM_READ;
   pageTabEntry->globalPage = 0;
   pageTabEntry->pageBaseAddr = pageAddr >> 12;
   ```

7.10 Alloacting a Pageable Page

void* Alloc_Pageable_Page(pte_t *entry, ulong_t vaddr)

1. We try to get a page: paddr = Alloc_Page();
   a) If the physical address is not null we get the page: `page = Get_Page((ulong_t) paddr);`
   b) Otherwise we select a page to steal (can be from an other process):

```
// Select a page to steal (e.g. from another process)
page = Find_Page_To_Page_Out();
paddr = (void*) Get_Page_Address(page);

// Find a place on disk for it
pagefileIndex = Find_Space_On_Paging_File();

// Make the page temporarily unpageable (can't let another process steal it)
page->flags &= ~(PAGE_PAGEABLE);

// Lock the page so it cannot be freed while we're writing
page->flags |= PAGE_LOCKED;

// Write the page to disk. Interrupts are enabled, since the I/O may block.
Enable_Interrupts();
Write_To_Paging_File(paddr, page->vaddr, pagefileIndex);
Disable_Interrupts();

// While we were writing got notification this page isn't even needed anymore
if (page->flags & PAGE_ALLOCATED)
{
  page->entry->present = 0;
  page->entry->kernelInfo = KINFO_PAGE_ON_DISK;
  page->entry->pageBaseAddr = pagefileIndex; // Remember where it is located!
}
else
{
  // The page got freed, don't need ????bookeeping or it on disk
  Free_Space_On_Paging_File(pagefileIndex);
```

```
        // It is still allocated though to us now
        page->flags |= PAGE_ALLOCATED;
      }

      // Unlock the page
      page->flags &= ~(PAGE_LOCKED);

      // XXX - flush TLB should only flush the one page
      Flush_TLB();
```

c) Set the fields for the page:

```
page->flags |= PAGE_PAGEABLE;
page->entry = entry;
page->entry->kernelInfo = 0;
page->vaddr = vaddr;
```

7.11 Finding Page to page out

We are using the LRU algorithm in order to find a page to page out.
static struct Page *Find_Page_To_Page_Out()

```
for (i=0; i < s_numPages; i++)
{
  if ((g_pageList[i].flags & PAGE_PAGEABLE) &&
  (g_pageList[i].flags & PAGE_ALLOCATED))
  {
    if (!best)
      best = &g_pageList[i];
    curr = &g_pageList[i];
    if ((curr->clock < best->clock) && (curr->flags & PAGE_PAGEABLE))
      best = curr;
  }
}
return best;
```

8 File System

We implemented a filesystem called geekOS filesystem (GOSFS). It is capable of multiple directories and long file names. In order to verify that our code is correct we used the test methods provided by geekOS.

8.1 Disk and Superblock Layout

The disk is divided in blocks of the same size. In our case the block size is 4 KB and stored in GOSFS_FS_BLOCK_SIZE. The first block on disk is reserved for the superblock. It containes the following attributes:

- magic: identifies the disk to be formatted with gosfs. We set the value to GOSFS_MAGIC_STRING "FTP"
- rootDirPointer: a pointer to the block the root directory is stored in
- size: the size of the disk in blocks
- bitmap: a bitmap to manage the allocation of blocks

8.2 Directory Layout

A directory is an array of struct GOSFS_Dir_Entry. Unlike files it has a fixed size of one block.
One directory entry contains the following attributes:

- size: the size of the file or directory
- flags: we used flags to determine whether
  this directory entry is used (GOSFS_DIRENTRY_USED) and if
  the entry describes another directory (GOSFS_DIRENTRY_ISDIRECTORY)
- filename: the name of the file or directory
- blockList: contains the blocks which belong to this file or directory
- acl: this field is not used by our implementation of the geekOS filesystem

8.3 Virtual File System Layer

As geekOS supports two filesystems (PFAT and GOSFS) it needs to be able to distinguish between those filesystems. This is where the virtual file system layer comes in place. It takes the system calls and forwards it to the correct filesystem implementation (ex. GOSFS_Open for Open()). This layer was provided for us.

8.4 Implementation

In this chapter we will have a look at some implementational details of our operating system. We will look at the basic funtions GOSFS_Format and GOSFS_Mount.

8.4.1 Format

The GOSFS_Format function takes a block device and formats it. This means setting up the super block and root directory so that the device can be mounted.
Function: static int GOSFS_Format(struct Block_Device *blockDev)

1. Create a directory and make sure it is clean

   ```
   struct GOSFS_Dir_Entry rootDirectory[GOSFS_DIR_ENTRIES_PER_BLOCK];
   memset( rootDirectory, '\0', sizeof(rootDirectory));
   ```

2. Get the number of blocks of the block device

   ```
   int numBlocks;
   numBlocks = Get_Num_Blocks(blockDev);
   int returnCode = 0;;
   ```

3. Get a temporary buffer cache and buffer to write root directory to disk

   ```
   bufferCache = Create_FS_Buffer_Cache(blockDev, GOSFS_FS_BLOCK_SIZE);

   if((returnCode =
       Get_FS_Buffer(bufferCache, GOSFS_ROOTDIR_BLOCKNR, &rootDirBuffer)) != 0)
       return returnCode;
   ```

4. Write root directory to disk:
   In our approach we always synchronise the disk with the buffers immediatly in order to keep the possibility of corruption as small as possible.

```
memcpy(rootDirBuffer->data, rootDirectory, sizeof(rootDirectory));
Modify_FS_Buffer(bufferCache, rootDirBuffer);
if((returnCode = Sync_FS_Buffer(bufferCache, rootDirBuffer)) != 0)
    return returnCode;
if((returnCode = Release_FS_Buffer(bufferCache, rootDirBuffer)) != 0)
    return returnCode;
```

5. Set up the super block

```
memcpy(superBlock.magic, GOSFS_MAGIC_STRING, 4);
superBlock.rootDirPointer = GOSFS_ROOTDIR_BLOCKNR;
superBlock.size = numBlocks;
superBlock.blockBitmap = Create_Bit_Set(numBlocks);
Set_Bit(superBlock.blockBitmap, GOSFS_ROOTDIR_BLOCKNR);
Set_Bit(superBlock.blockBitmap, GOSFS_SUPERBLOCK_BLOCKNR);
```

6. Get the buffer for the super block and write it to disk

```
if((returnCode =
Get_FS_Buffer(bufferCache, GOSFS_SUPERBLOCK_BLOCKNR, &superBlockBuffer)) != 0)
    return returnCode;

memcpy(superBlockBuffer->data, &superBlock, sizeof(superBlock));
Modify_FS_Buffer(bufferCache, superBlockBuffer);
if((returnCode = Sync_FS_Buffer(bufferCache, superBlockBuffer)) != 0)
    return returnCode;
if((returnCode = Release_FS_Buffer(bufferCache, superBlockBuffer)) != 0)
    return returnCode;
```

7. Finally destroy the buffer cache and we are finished

```
if((returnCode = Destroy_FS_Buffer_Cache(bufferCache)) != 0)
    return returnCode;

return returnCode;
```

8.4.2 Mount

The mount function sets up a mount point and makes a block device usable.
Function: static int GOSFS_Mount(struct Mount_Point *mountPoint)

1. Allocate an instance which will contain filesystem specific data

```
instance = (struct GOSFS_Instance*) Malloc(sizeof(*instance));
if (instance == 0)
{
return ENOMEM;
}
memset(instance, '\0', sizeof(*instance));
```

2. Store the buffer cache in our instance

```
returnCode = Get_FS_Buffer(instance->bufferCache,
                           GOSFS_SUPERBLOCK_BLOCKNR,
                           &superBlockBuffer);
if(returnCode != 0)
{
 Free(instance);
 return returnCode;
}
```

3. Check the super block

```
superBlock = (struct GOSFS_SuperBlock*)Malloc(sizeof(*superBlock));
memcpy(superBlock, superBlockBuffer->data, sizeof(*superBlock));
if(strcmp(superBlock->magic, GOSFS_MAGIC_STRING))
{
  [...]
  return EINVALIDFS;
}
```

4. Add the super block to our instance

```
instance->superBlock = superBlock;
```

5. Set up mount point

```
mountPoint->fsData = instance;
mountPoint->ops = &s_gosfsMountPointOps;
```

6. Free the buffer and we are done

```
Release_FS_Buffer(instance->bufferCache, superBlockBuffer);
return 0;
```

8.4.3 Additional functions

After a formatted device is mounted you can work with it. As the other functions are much bigger than those described above we will just describe them briefly.

GOSFS_Close()	Takes an open file and closes it
GOSFS_Close_Directory()	Takes an open directory and closes it
GOSFS_Create_Directory()	Creates a directory at the given path
GOSFS_Lookup()	Returns the block number of the directory of the given path
GOSFS_Open()	Opens a file at the given path.If the file doesn‘t exist it creates it
GOSFS_Open_Directory()	Opens a directory
GOSFS_Read_Entry()	Reads a directory entry from an open directory
GOSFS_Stat()	Returns the metadata of the file spezified by the given path
GOSFS_Write()	Writes data to the current position in the file
GOSFS_Read()	Reads data from the current position in the file

9 System Calls

9.1 Supported Calls

Our kernel supports the following system calls:

Call	Effect
Sys_Null	Does nothing except immediately return control
-	to the interrupted user program
Sys_Exit	Destroys the thread and frees associated memory
Sys_PrintString	Prints the string passed as an argument
Sys_GetKey	Gets a single key press from the console
-	Suspends the user process until a key press is available
Sys_SetAttr	Sets the current text attributes
Sys_GetCursor	Gets the current cursor position
Sys_PutCursor	Sets the current cursor position
Sys_Spawn	Creates a new user process
Sys_Wait	Waits for a process to exit
Sys_GetPID	Returns the PID of the current thread
Sys_SetSchedulingPolicy	Sets the scheduling policy
Sys_GetTimeOfDay	Gets the time of day
Sys_CreateSemaphore	Creates a semaphore
Sys_P	Acquires a semaphore
Sys_V	Releases a semaphore
Sys_DestroySemaphore	Destroys a semaphore
Sys_Close	Closes an open file or directory
Sys_CreateDir	Creates directory
Sys_Format	Formats a device
Sys_FStat	Gets metadata of an open file
Sys_Mount	Mounts a filesystem
Sys_Open	Opens a file
Sys_OpenDirectory	Opens a directory
Sys_Read	Reads from an open file
Sys_ReadEntry	Reads a directory entry from an open directory handle
Sys_Stat	Gets file metadata
Sys_Write	Writes to an open file

- SYS_PRINTSTRING: we must ensure that the supplied string is not larger than 1024

characters

- SYS_SPAWN: we must ensure that the supplied string is not larger than VFS_MAX_PATH_LEN

9.2 Copy_From_User()

In order to be able to access the strings passed as arguments to the functions SYS_PRINTSTRING and SYS_SPAWN, we had to implement the function:

```
bool Copy_From_User(void* destInKernel, ulong_t srcInUser, ulong_t bufSize)
```

1. Validating if srcInUser is illegal (out of bound):

   ```
   srcInUser += USER_VM_START;
   if(!Validate_User_Memory(uc, srcInUser, numBytes))
     return false;
   ```

2. Copying user data to kernel buffer: memcpy(destInKernel, (void*)srcInUser, numBytes);

9.3 Copy_To_User()

We implemented this function in order to be able to acces data which flows from kernel to user space. This is needed when we store data in user space (SYS_GETCURSOR). The implementation is the same as in Copy_From_User, just the other way around.

```
bool Copy_To_User(ulong_t destInUser, void* srcInKernel, ulong_t bufSize)
```

10 Sources

- GeekOS documentation
- http://www.cs.umd.edu/class/spring2005/cmsc412/projects.html
- http://www.cs.iitm.ernet.in/ cs313/projects.html
- ELF Specification
- The IA-32 Intel(R) Architecture Software Developer's Manual