CS 377 : Operating Systems Lab 06

Anurag Shirolkar (120050003)
Dheerendra Rathor (120050033)

Question 1

Function calls in main.c

Init_BSS()

Definition Source File: mem.c

- Initializes the .bss section of the memory image of the executable.
- Fills the .bss section with zeroes.

Init_Screen()

Definition Source File: screen.c

- Initializes the screen module.
- Sets the console state and clears the screen

Init_Mem(bootinfo)

Definition Source File: mem.c

- Initializes memory management data structures.
- looks through the bootInfo memory regions to find a region starting at 0x100000 and if that region is of type 1, set the bootInfo memsize as that region's length in KB + 0x1000.
- Initialises some other variables like number of pages, end of memory etc.
- Main function which allow page allocation

Init_CRC32()

Definition Source File: crc32.c

- Fills the crc_table

Init_TSS()

Definition Source File: tss.c

- Initializes the task state segment.
- TSS data structure contains the GPRs, the registers for stack, other registers.

lockKernel()

Definition Source File: smp.c

- Called whenever an interrupts occur in general.
- Holds a global lock so that no other processes can enter kernel mode
- Other user threads may run on concurrent processes.
- Other processes that require kernel are said to "spin in place" waiting for the global lock to be released.

Init_Interrupts()

Definition Source File: int.c

- It first initializes the Interrupt Descriptor Table.
- Installs a dummy interrupt handler for every entry in the handler table.

Init_SMP()

Definition Source File: smp.c

- Initializes the symmetric multiprocessing parameters. Like first allocating memory for stack and then alloting them to cpus.

TODO_P(PROJECT_VIRTUAL_MEMORY_A, "initialize virtual memory page tables.")

- Prints the “feature unimplemented” message if the first argument (in this case PROJECT_VIRTUAL_MEMORY) is true

Init_Scheduler()

Definition Source File: kthread.c

- Creates a kernel thread object and initializes it and makes it the main thread
- Adds created thread to the list of all threads.
- Allso starts idle and reaper threads.

Init_Traps()

Definition Source File: trap.c

- Initializes handlers for software interrupts.
- stack exception, general protection fault and syscall handlers are installed

Init_local_APIC()

Init_Timer()

Definition Source File: timer.c

- Installs interrupt handlers for timer interrupt request
- Initializes the timer interrupt.

Init_Keyboard()

Definition Source File: keyboard.c

- Disables the shift keys
- Empties the buffer
- Installs the interrupt handler for keyboard

Init_DMA()

Definition Source File: dma.c

- Resets the dma controller
- Clears registers of DMA controller

Init_IDE()

Definition Source File: ide.c

- Resets the ide controller, clears all its registers
- Probes and registers the ide drives
- Finally starts the ide request thread

Init_PFAT()

Definition Source File: pfat.c

- Registers pseudo FAT file system

Init_GFS2()

Definition Source File: gfs2.c

- Registers the global file system.
- Allows all nodes in the cluster to access the memory concurrently.

Init_GOSFS()

Definition Source File: gosfs.c

- Registers the geekOS file system.

Init_CFS()

Definition Source File: cfs.c

- Registers the clustered file system.
- Allows the file system to be mounted on multiple servers simultaneously.

Init_Alarm()

Definition Source File: alarm.c

- Starts a kernel thread with a function called alarm handler
- Starts with normal priority

Release_SMP()

Definition Source File: smp.c

- In case of multiple CPUs, sets the running of other CPUs as 1 and waits for running to become 2 in a loop.

Init_Network_Devices()

Definition Source File: net/net.c

- Registers various network devices
- Starts the receive kernel thread with normal priority

Init_ARP_Protocol()

Definition source file: in net/arp.c

- Initializes the arp table, sets its head and tail to zero and dispatches the table

Init_IP()

Definition Source File: net/ip.c

- For the devices in the network device list, register the device with an IP address.
- IP address is assigned on the basis of a base IP address and device's own address.

Init_Routing()

Definition Source File: net/routing.c

- Initializes the routing table with local routes
- Generates local routes for directly attached IP devices.

Init_Sockets()

Definition Source File: net/socket.c

- Prints message according to status of project sockets

Init_Sound_Devices

Definition Source File: include/geekos/projects.h

- calls a TODO_P function which should print "Initializing sound card".

Mount_Root_Filesystem

Definition Source File: in main.c

- Mounts the root filesystem

TODO_P(PROJECT_VIRTUAL_MEMORY_A, "initialize page file.");

Definition Source File: include/geekos/projects.h

- Prints “initialize page file” based on value of PROJECT_VIRTUAL_MEMORY_A

Set_Current_Attr(ATTRIB(BLACK, GREEN | BRIGHT));

Definition source file: screen.h

- Sets the font color to bright green
- Sets the background color as black

Print("Welcome to GeekOS!\n");

Definition Source file: conio.c

- Prints "Welcome to GeekOS!"

Set_Current_Attr(ATTRIB(BLACK, GRAY));

Definition Source file: screen.h

- Sets font color to gray
- Sets the background color to black

TODO_P(PROJECT_SOUND, "play startup sound")

Definition Source file: include/geekos/projects.h

- Checks if the first argument is true or not, if yes it prints the second argument

Spawn_Init_Process

Definition Source file: main.c

- Creates a kernel_thread called initProcess.
- Calls Spawn_Foregroud function with the executable /c/shell.exe and initProcess.
- Waits for the initProcess to exit.

Hardware_Shutdown

Definition Source file: main.c.

- Sends shutdown code "Shutdown" for Qemu and Boch

Question 2:

The definition of Kernel_Thread is described below

```
struct Kernel_Thread {

	/* Save threads stack pointer when thread is suspended. */
	ulong_t esp;

	/* Counter for timer based preemption */
	volatile ulong_t numTicks;

	/* Total time used by thread till now */
	volatile ulong_t totalTime;

	/* Priority of thread. Used for prioroty based premption */
	int priority;

	/* Defines macro for the previous and next fields when this thread is in thread
queue */
	DEFINE_LINK(Thread_Queue, Kernel_Thread);

	/* Pointer to stack page of Kernel Thread */
	void *stackPage;

	/* Pointer to user context */
	struct User_Context *userContext;

	/* Pointer to parent thread of current thread */
	struct Kernel_Thread *owner;
```

```
        /* Prefered core for this thread. AFFINITY_ANY_CORE to run on any core */
        int affinity;


        int refCount;
        /* On being true, process doesn't wait to be reaped after Exit() */
        int detached;

        /* PID of the thread */
        int pid;

        /* Stores whether flag is alive or dead */
        bool alive;

        /* List of threads to be joined */
        struct Thread_Queue joinQueue;

        /* exit code (Return Status) of thread */
        int exitCode;

        /* Macro for linking previous and current thread */
        DEFINE_LINK(All_Thread_List, Kernel_Thread);

        /* Array availabe for thread local data */
        #define MAX_TLOCAL_KEYS 128
        const void *tlocalData[MAX_TLOCAL_KEYS];

        /* Name of thread */
        char threadName[20];
};
```

Question 3

Difference between User level thread and Kernel level thread

1. Scheduling of Kernel level threads are managed by scheduler of the kernel. Whereas the scheduling of the User level threads is managed by a thread library
2. The kernel is not aware of the User level threads

In the implementation of GeekOS there are two ways for creating a kernel_thread

1. `Start_Kernel_Thread`
2. `Start_User_Thread`

Both return pointer to an object kernel_thread. The only difference between those functions is that `Start_Kernel_Thread` return a thread that can run in kernel mode.
`Start_User_Thread` returns a thread that runs in user mode.
Both these functions append the newly created threads in the runnable queue by calling

```
Make_Runnable_Atomic(kthread);
```

This means that both the threads will be scheduled by the scheduler.
Hence the threads implemented in the GeekOS are all Kernel-level threads.

Question 4:

Keyboard input is handled via syscall **`SYS_GETKEY`** which in turn call **`Wait_For_Key(void)`** from **`keyboard.c`**
The function first checks if there is already a key present in the Pipe and return this key. Otherwise it disable Interrupts and put the keyboard buffer queue *s_waitQueue* in sleep until key arrives in buffer queue

The **`Wait(Thread_Queue)`** function puts the `thread_queue` into the `waitQueue` and call scheduler to schedule running processes.
The keyboard interrupts are handled by function **`Keyboard_Interrupt_Handler (Interrupt_State).`** It makes an interrupt request at port `KB_CMD (io port 0x64)` and read bytes at this port. After reading data, it pushes data into keyboard buffer queue and wake up the thread by **`Wake_Up`** which makes the thread runnable and available for scheduling

Question 5.

Implementation of Sys_Fork()

1. Define a new function `make_context_copy()` which makes a copy of context of the current thread and returns it.
2. Now store the copy of context in a variable
   `context context_copy = make_context_copy();`
3. Create a new child process using the above copy
   `child_thread = create_user_thread(context_copy);`
4. Now set `child_thread->owner = parent_process;`
5. Now change the value of `%eax` register which stores the returned value of the function. For this esp of `child_thread` is used.
6. The `%eax` can be accessed by dereferencing `(esp + 4 * 10)`. esp is child_thread->esp here.
7. For child_thread set %eax to 0 . This ensures that PID returned by fork to child process is 0 when this process is scheduled by the scheduler.
8. Use `Make_Runnable_Atomic(child_thread)` to put the child thread on the Runnable Queue.
9. Return `child_thread->pid` from the syscall.

Alternate Implementation of Sys_Fork

```
void sys_fork() {
      int parent_pid = sys_getPID();
      context context_copy = make_context_copy();
      kernel_thread child_thread = create_user_thread(context_copy);

      // At this point the context for the two process will be exactly same
      // including the `parent_pid` variable defined above.

      if (sys_getPID() == parent_pid){
            // Only the parent process will enter this loop
            Make_Runnable_Atomic(child_thread);
            return child->pid;
      }
      else {
```

```
            // Child thread will enter this
            // since parent_pid variable has pid of its parent
            return 0;
      }

}
```