CS3204 Operating Systems - Spring 2001
Instructor: Dr. Craig A. Struble
Homework 2

Assigned: Thursday, Feb. 8, 2001 **Due:** Friday, Feb. 16, 2001

---

1. *[5 pts.]* Exercise 1 in Nutt, Chapter 3.

---

2. *[5 pts.]* Exercise 4 in Nutt, Chapter 3. By shared memory, the problem means a memory abstraction that allows processes on the same machine or different machines to store and retrieve information shared between processes in a transparent way. For example, a variable `x` that processes running on `foo.xyz.x` and `bar.xyz.x` use to store and retrieve an integer value. Be sure to discuss how message passing is used the applications to allocate and deallocate shared memory.

---

3. *[10 pts.]* This question deals with the abstractions provided by the runtime support in high level programming languages. Consider a simple C or C++ program for calculating the transpose of a matrix. The input matrix is contained in a file and the output is also written to a file. The names of these two files are entered by the user when prompted by the program. Identify ALL POSSIBLE instances when system calls are made.

---

4. *[5 pts.]* This question is regarding the booting process in Linux. After the hardware is initialized (`/usr/src/linux/arch/i386/kernel/head.S`), `start_kernel()` is the first C function to be called. Print out this function (in `/usr/src/linux/init/main.c`) and identify the routines called for initializing the following kernel structures

   1. trap handler table,
   2. IRQ (interrupt) handler table,
   3. kernel modules,
   4. file table,
   5. memory.

---

5. *[5 pts.]* Exercise 2 in Nutt, Chapter 4.

---

6. *[5 pts.]* Exercise 3 in Nutt, Chapter 4.

---

7. *[5 pts.]* Exercise 6 in Nutt, Chapter 4.

---

8. *[10 pts.]* Interrupt handlers often disable interrupts when called to prevent another interrupt from occuring while one is being handled. Frequently, only a portion of the interrupt handler needs to be executed immediately, while the rest can be deferred. Modern operating systems provide a mechanism for deferring non-critical interrupt handling operations to avoid disabling interrupts for too long.

Linux uses *bottom halves* for deferring interrupt handler operations. After every (slow) interrupt if no further interrupt is running, a list of bottom halves is scanned and if they

are marked active they are carried out in turn. Bottom halves are atomically executed, that is, interrupts are disabled when bottom halves are executed.

Discuss how Linux implements the scanning and activation of bottom halves. Refer to the function `void run_bottom_halves(void)` in `/usr/src/linux/kernel/softirq.c`. Identify any limitations built into the implementation. Why do the limitations exist? What OS design factors were used to motivate the limitations in the implementation? Useful information is also contained in `/usr/src/linux/include/asm/softirq.h`.