3.1. (a) There are three places in the source code where the CPP directive #define is used to set the value of NPROC: include/conf.h, config/conf.h, config/Configuration where Configuration is a text file. Change NPROC to 15 in include/conf.h and config/conf.h. Modify main.c in system/ after the call to kprintf() outputting "...creating a shell" and before calling recvclr() to print the value of NPROC. What do you observe? (NPROC=100) Check the value of NPROC in include/conf.h and config/conf.h. What are their values? (NPROC=100) Check the last modified timestamp of the two files and use the diff command to compare their content. What do you find? (They have the same last modified timestamp, which is the time when they were compiled instead of when manually modified) Based on the discussion in class, what is the correct method for modifying the system parameter NPROC that sets the size of the kernel's process table? (Modifying the NPROC value in config/Configuration to 15) Perform this modification and verify that main() outputs 15 when printing NPROC.

(b) To test our createminpid implementation, we create a process, kill it, then create another process. If implemented correctly, we should have the same PID for these two processes, while the original create implementation would assign the next PID for the second process. See the test result in Figure 1.

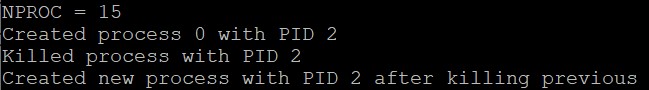


Figure 1

(d) Inspect the code of create() and specify in lab1ans.pdf which statement of create() helps implement this illusion. (\*--saddr = (long)INITRET;) Additional information pushed onto the stack after the address of userret() relates to context-switching which will be discussed later. To check that this is the case, code a function, void myuserret(void), in system/myuserret.c, that instead of calling kill() calls kprintf() to print the message "In myuserret()" then enters into an infinite while-loop. Modify create() so that the newly created function jumps to myuserret(), not userret(), upon executing ret. Specify in lab1ans.pdf how you accomplish this task. (In the file myuserret.c, I simply create a function doing two tasks, namely outputting “In myuserret” and entering an infinite loop) Using main() as your test function, verify that the system hangs when main() completes execution.

Suppose main() calls a function, int abc(void), that prints message "In abc." then returns to its caller. In your test case, suppose main() prints "About to return from main()." after calling abc(). Will your modification of create() allow "About to return from main()." to be output, or will abc() returning cause the system to hang? (It will allow “About to return from main().” To be output) Test this scenario and explain in lab1ans.pdf the result you observe. (XINU is a multi-process system with its scheduler, the infinite loop in the process does not freeze the entire system. Other processes continue to run. As a result, we still see output from the system even though the test process is stuck in myuserret())

(e) Parent process ID. create() remembers the PID of the process that created a new process in the latter's process table field, pid32 prparent. Perform a quick search of the header files in include/ to determine the C type of pid32 and note it in lab1ans.pdf. (The C type of pid32 is int32) Code a new XINU system call, pid32 getppid(pid32), in system/getppid.c that takes the PID of a process as argument, finds its parent's PID and returns the value. Since the argument is not guaranteed to be a valid PID, sanity checks must be performed which incurs overhead but is critical for system calls which cannot behave unexpectedly no matter what arguments are passed. That is, a system call running in kernel mode behaving erratically compromises the entire system. To determine if the passed PID is valid, check its entry in the process table data structure, proctab, where the process state field prstate has value PR\_FREE if the PID is invalid (i.e., process does not exist). getppid() returns SYSERR if the argument is not a valid PID. Is this the only sanity check that needs to be carried out? Explain your reasoning in lab1ans.pdf and modify getppid() accordingly. (No, we still need to check if the parent PID is valid, since it might have terminated and is no longer valid)

When extending a kernel by making changes such as introducing new system calls, compatibility with the legacy kernel and any unintended side effects need to be examined. create() populates the prparent field of a newly created process correctly. This is not the case for a NULL/idle process where inspecting nulluser() in initialize.c shows that the prparent field is not explicitly set. Before implementing additional kernel modifications to make getppid() backward compatible, we need to consider what options there are to assign meaningful behavior when getppid() is called with argument 0. Discuss in lab1ans.pdf two options, significantly distinct from each other, choose an option and modify XINU (e.g., getppid(), functions in initialize.c) accordingly. Ignoring the issue will not be considered a meaningful option. (Option 1: Return 0 to indicate that the idle process has no valid parent. Option 2: Modify system initialization in nulluser() to assign a valid parent to the idle process, so that it returns a valid value. Option 1 was used in my implementation)

3.2. Test your XINU code on a backend and describe what you find in lab1ans.pdf. Explain the results based on our discussion of how fixed priority scheduling works in XINU. (I saw mostly ‘A’ and ‘C’ and few ‘B’ in the output, since outputting ‘A’ and ‘C’ has the same priority and outputting ‘B’ has lower priority)

Second, repeat the above with the priority of the child process executing sndA set to 30. Explain the output behavior in lab1ans.pdf. (I saw mostly ‘A’ and few ‘B’ or ‘C’ in the output, since outputting ‘A’ has the highest priority and outputting ‘B’ and ‘C’ has lower priorities)

Third, repeat the first scenario with the priority of the child process executing sndB() set to 20. Explain your finding. (I saw similar number of ‘A’, ‘B’ and ‘C’ in the console, since they have the same priority)

4.1. clkcounterfine is a software clock driven by a hardware clock. It does not get updated if clkdisp.S is not executed which can happen if XINU's hardware clock is temporarily ignored. That is, software can command the the hardware to silence or ignore interrupts. Some interrupts on x86 cannot be disabled, called NMIs (non-maskable interrupts), which are generated when catastropic hardware failures are detected. The more XINU disables clock interrupts, for whatever reason, the more inaccurate clkcounterfine becomes. Legacy XINU uses a global variable, uint32 clktime, that is updated in clkhandler() to monitor how many seconds have elapsed since a backend was bootloaded. Test and assess if both clkcounterfine and clktime provide consistent time using test code in main(). Make main() execute a lengthy for-loop before printing clkcounterfine and clktime. Experiment with loop bound (in the millions) until clktime outputs several seconds. Describe your finding in lab1ans.pdf. (I made the main function run an empty loop for 10^8 times. The value of clkcounterfine is 2263, and clktime shows 2 seconds. They provided consistent time. See the results in Figure 2)



Figure 2

4.2. As noted in 4.1, software clocks such as counters clkcounterfine and clktime become inaccurate when hardware clock interrupts are disabled. In x86, clock interrupts can be disabled by executing the cli assembly instruction. One method for embedding assembly code within C code is through inline assembly. By adding the statement asm("cli") before the for-loop in main(), you instruct gcc to embed cli before the assembly code for the for-loop that gcc generates (which then gets translated into machine code). Re-enabling the clock interrupt can be done by executing the assembly instruction sti. As a variation of 4.2, encapsulate your for-loop in main() with inline assembly code that execute cli and sti so that while the for-loop is executing clock interrupts are disabled. Compare the values of clkcounterfine and clktime against their values from 4.1. Discuss your finding in lab1ans.pdf. (After disabling clock interrupts, the value of clkcounterfine becomes 5, and clktime shows 0 seconds, which are inaccurate. See the results in Figure 3)



Figure 3

4.3. An important task carried out by XINU's clock interrupt handler is keeping track of how much of a process's time budget (i.e., time slice or quantum) has been expended. If the time slice remaining reaches 0, clkhandler() calls XINU's scheduler, resched() in system/resched.c, to determine which process to execute next on Galileo's x86 CPU. When a process runs for the first time after creation, its time slice is set to QUANTUM which is defined in one of the header files in include/. Its default value of 2 is on the small side. Unused time slice of the current process is maintained in the global variable, uint32 preempt, which is decremented by clkhandler() each time it is invoked by clkdisp.S. If preempt reaches 0, XINU's scheduler is called.

Rerun the third scenario of 3.2 where the fixed time slice of XINU's round robin scheduler, QUANTUM (defined in a header file in include/), is increased from 2 to 8. Compare the two results and discuss your finding. (After modifying QUANTUM from 2 to 8, the length of consecutive ‘A’, ‘B’ and ‘C’ in the output increases, meaning that a single process is executed for a longer period of time)

5.1. The code of addfour() in addfour.S modifies the content of EAX, EBX, ECX, EDX. In x86 CDECL the calling function is responsible for saving and restoring the content of registers EAX, ECX, EDX. Hence the assembly code generated by gcc when compiling the call to addfour() from main() will save the content of EAX, ECX, EDX before calling addfour() and restore their original values after addfour() returns. Per CDECL saving and restoring EBX is the callee's (i.e., addfour()) responsibility. Although calling addfour() from main() as is may work and return the correct addition result, addfour() contains a bug that can surface if main() has been using EBX whose original value is needed after the call to addfour() returns. Modify addfour() in addfour.S to correct the above bug. Explain in lab1ans.pdf the detailed logic behind your fix. Verify that your modified code works correctly. (I pushed EBX into the stack to save its value and pop it after the addition to restore its value. See the test results in Figure 4)



Figure 4

5.2. Since the return value is communicated from callee to caller through register EAX in CDECL, gcc will ensure that addfourC() puts its result in EAX before returning to its caller testaddfourC(). Your assembly code testaddfourC.S need not touch EAX so that the result returned by testaddfourC() is propagated back to the caller of testaddfourC() (i.e., main()). If your testaddfourC() modifies EBX then, as with 5.1, ensure that its original value is restored by testaddfourC() before returning to main(). If EBX is not modified there is no need to do so. Test and verify that your implementation works correctly. (See the test results in Figure 5)



Figure 5

Bonus. Implement a version of create(), int32 creates(), in system/creates.c, where the arguments are the same as create() but for stack size, priority, process name which are omitted. Stack size is set to a new system parameter, PROCSTACKSZ, defined as 8192 in include/process.h, priority is set to the parent's priority plus 1, and name is set to "NONAME". Test and verify that creates() works correctly, then rerun the third scenario of 3.2. Discuss your finding in lab1ans.pdf. (I can only see ‘A’ outputting if we do not sleep during the output, since the processing speed is too fast. However, we can see ‘A’, ‘B’ and ‘C’ if we add a short sleep in the output function)