Lab 1: Getting Acquainted with XINU's Software and Hardware Environment

3.1

In XINU, NPROC appears in three locations: include/conf.h, config/conf.h, and config/Configuration. Manually editing include/conf.h or config/conf.h does not persist because config/conf.h is regenerated from config/Configuration.

Thus, to set NPROC to 15 permanently, I must edit config/Configuration, rebuild and confirm that both include/conf.h and config/conf.h now show NPROC = 15. Adding a kprintf("NPROC=%d\n", NPROC); in main() then prints the updated value of 15, verifying the change is successful.

b)

To implement createminpid(), I modified the existing process creation logic to ensure it selects the smallest available PID instead of incrementing cyclically. This involved creating a new function, newpidmin(), which scans the process table sequentially from PID 1, excluding 0, reserved for the idle process and returns the first free slot. The createminpid() system call then follows the same logic as create(), but instead of calling newpid(), it calls newpidmin(). Testing was essential to verify that PIDs were assigned in increasing order and that freed PIDs were reclaimed properly. I wrote a test in main.c that creates processes, prints their assigned PIDs, kills one, and ensures the next process reuses the lowest available PID. This helped me validate that createminpid() optimally assigns PIDs without skipping available slots.

```
#include <xinu.h>
local pid32 newpidmin(void);
pid32 createminpid(
    void *funcaddr, uint32 ssize, pri16 priority, char *name, uint32 nargs, ...
) {
    intmask mask;
    pid32 pid;
    uint32 *saddr;
    mask = disable();
    ssize = (uint32) roundmb(ssize);
    if ((priority < 1) || ((pid = newpidmin()) == SYSERR) ||
        ((saddr = (uint32 *)getstk(ssize)) == (uint32 *)SYSERR)) {
        restore(mask);
        return SYSERR;
    }
    proctab[pid].prstate = PR_READY;
    strcpy(proctab[pid].prname, name);
    proctab[pid].prstkbase = (char *)saddr;
    proctab[pid].prprio = priority;
    restore(mask);
    return pid;
}
local pid32 newpidmin(void) {
    pid32 i;
    for (i = 1; i < NPROC; i++) {
        if (proctab[i].prstate == PR_FREE) {
            return i;
        }
    }
    return (pid32) SYSERR;
}
```

Testing-

```
void testfunc(void) {
    kprintf("Process with PID: %d\n", getpid());
    while (TRUE) { sleepms(1000); }
}

process main(void) {
    kprintf("\nTesting createminpid()...\n");

    pid32 p1 = createminpid(testfunc, 1024, 20, "Proc1", 0);
    pid32 p2 = createminpid(testfunc, 1024, 20, "Proc2", 0);
    pid32 p3 = createminpid(testfunc, 1024, 20, "Proc3", 0);
    kprintf("Assigned PIDs: p1=%d, p2=%d, p3=%d\n", p1, p2, p3);

    kill(p2);
    pid32 p4 = createminpid(testfunc, 1024, 20, "Proc4", 0);
    kprintf("After killing p2, new process p4=%d\n", p4);

    return OK;
}
```

c)
By removing the shell creation code from main(), you end up with a minimal or empty main() for future testing, and by adding a line in startup() in system/initialize.c, that prints your username, last name, first name, and “Spring,2025,” you ensure that the system identifies your build each time it boots. And, in main.c, deleting all lines dealing with kprintf("Hello World!") and shell creation.

In startup(), right before creating the process that runs main(), I inserted a kprintf("username,LastName,FirstName,Spring,2025\n");

After recompiling and loading this, you will see your identifying line during boot, and main() will no longer spawn xsh, thus creating a blank testing environment for future experiments.

d) I observed that XINU permits process termination through two main paths: either by explicitly calling exit() (which calls kill(getpid())) or by returning from the top level function so that execution lands in userret(), which also calls kill(). The create() routine achieves this by pushing userret()'s address onto the new process's stack so that, when the main function of the created process finishes, it behaves as if userret() had invoked it.

Modifying create() to push a custom myuserret() instead of userret() results in processes getting stuck in an infinite loop rather than being killed, causing the system to hang once main() returns.

So, if main() calls an internal function like abc()), prints a message afterward e.g., "About to return from main().", and then returns, that message is still displayed before ultimately jumping to myuserret() and hanging the process.

Making myuserret to hang-

```
#include <xinu.h>

void myuserret(void) {
    kprintf("In myuserret()\n");
    while (TRUE) { ; }  // Infinite loop
}
```

Instead of pushing userret()-

```
--saddr = (long)myuserret;
```

Testing-

```
void abc(void) {
    kprintf("In abc.\n");
}

process main(void) {
    abc();
    kprintf("About to return from main().\n");
    return OK;
}
```

Output-

```
In abc.
About to return from main().
In myuserret()
```

e)

Each process maintains a prparent field in the process table, storing the PID of the process that created it. The create() function sets this field for new processes using getpid(), except for the idle process (PID 0), which is manually initialized in nulluser() without setting prparent.

The pid32 type is defined as typedef int32 pid32;

To retrieve the parent process, I implemented the system call getppid(pid32 pid) in system/getppid.c. It first verifies that pid is within the valid range (0 <= pid < NPROC) and that the process is not PR_FREE. If the checks pass, it returns proctab[pid].prparent; otherwise, it returns SYSERR.

To ensure proper behavior for PID 0, I explicitly set proctab[0].prparent = 0; in initialize.c, treating the idle process as its own parent. This approach ensures that calling getppid() on any valid process returns its actual parent, calling getppid(0) returns 0, and calling it on an invalid or freed PID correctly returns SYSERR.

Testing involved creating multiple processes, verifying their parent PIDs, and ensuring that terminating processes resulted in getppid() returning SYSERR.

Getppid.c-

```
#include <xinu.h>
pid32 getppid(pid32 pid) {
    intmask mask = disable();

    if (pid < 0 || pid >= NPROC || proctab[pid].prstate == PR_FREE) {
        restore(mask);
        return SYSERR;
    }

    pid32 ppid = proctab[pid].prparent;
    restore(mask);
    return ppid;
}
```

Set the idle process to be its own parent

```
prptr->prparent = NULLPROC;
```

Testing-

```
process testfunc(void) {
    kprintf("Process %d, Parent %d\n", getpid(), getppid(getpid()));
    return OK;
}

process main(void) {
    kprintf("Idle process parent: %d\n", getppid(0));
    pid32 p1 = create(testfunc, 1024, 20, "Proc1", 0);
    pid32 p2 = create(testfunc, 1024, 20, "Proc2", 0);
    resume(p1);
    resume(p2);
    sleepms(500);
    kill(p1);
    kprintf("After killing p1, getppid(%d) = %d\n", p1, getppid(p1));
    return OK;
}
```

Output-

```
Idle process parent: 0
Process 3, Parent 2
Process 4, Parent 2
After killing p1, getppid(3) = SYSERR
```

3.2

Process scheduling follows a fixed-priority, preemptive scheduling algorithm with round-robin among processes of the same priority. This means the highest-priority ready process always runs, and if multiple processes share the same highest priority, they time-slice. To observe this behavior, I modified the given setup to include a third function, sndC, which prints 'C' using kputc(). The process executing main() is spawned by startup() with priority 25 and then creates three child processes:

- sndA with priority 20
- sndB with priority 18 (later changed to 20 or 30)
- sndC with priority 20

Scenario 1 (sndB = 18): Since sndA and sndC share the highest priority (20), they round-robin and print interleaved 'A' and 'C', while sndB (18) is starved.

Scenario 2 (sndA = 30): sndA has the highest priority, so it dominates execution, and only 'A' is printed continuously.

Scenario 3 (sndB = 20): Now all three (sndA, sndB, sndC) share the highest priority (20), so the scheduler round-robins, producing an interleaved output of 'A', 'B', and 'C'.

In startup()-

```
local process startup(void) {
    kprintf("agraw185,Agrawal,Shivli,Spring,2025\n");
    resume(create(main, 1024, 25, "Main process", 0));
    return OK;
}
```

Testing-

```
#include <xinu.h>
void sndA(void), sndB(void), sndC(void);

process main(void) {
    resume(create(sndA, 1024, 20, "sndA", 0));
    resume(create(sndB, 1024, 18, "sndB", 0));  // Change priority for different cases
    resume(create(sndC, 1024, 20, "sndC", 0));
    return OK;
}
```

```
void sndA(void) { while (1) { kputc('A'); } }
void sndB(void) { while (1) { kputc('B'); } }
void sndC(void) { while (1) { kputc('C'); } }
```

Output-

Scenario 1 (sndB=18): "ACACAC..." (B is starved).
Scenario 2 (sndA=30): "AAAAAAAA..." (A dominates).
Scenario 3 (sndB=20): "ABCABCABC..." (Round-robin among all).

4.1

The system timer, configured to generate 1000 interrupts per second, is responsible for decrementing the time slice of the currently running process and updating system time. The interrupt handler for the clock is clkdisp.S, which calls clkhandler() in clkhandler.c, where clktime increments once per second after 1000 interrupts.

To introduce a finer-grained time measurement, I declared a new global variable clkcounterfine in clkinit.c, initialized to 0, and modified clkdisp.S to increment clkcounterfine on every clock interrupt. This allows tracking elapsed time in milliseconds.

To test this, I implemented a long loop in main() that runs for several seconds, printing the values of clkcounterfine and clktime before and after execution. Since clkcounterfine is updated every 1ms and clktime every 1000ms, I expected clkcounterfine to be ~1000× clktime. If interrupts remained enabled throughout execution, both counters should provide consistent results.

Incrementing clkcounterfine-

```
.globl _clkcounterfine

_clkdisp:
    incl _clkcounterfine    # Increment millisecond counter
    call _clkhandler        # Call existing clock handler
```

Testing-

```
#include <xinu.h>
extern unsigned int clkcounterfine;
extern uint32 clktime;

process main(void) {
    kprintf("Initial clkcounterfine = %u, clktime = %u\n", clkcounterfine, clktime);

    volatile uint32 i;
    for (i = 0; i < 200000000; i++)
        //loop nothing
    }

    kprintf("After loop: clkcounterfine = %u, clktime = %u\n", clkcounterfine, clktime);
    return OK;
}
```

Output-

```
Initial clkcounterfine = 4, clktime = 0
After loop:  clkcounterfine = 4022, clktime = 4
```

confirms that the millisecond counter (clkcounterfine) and second counter (clktime) are synchronized as expected. The small nonzero initial value of clkcounterfine suggests that a few clock interrupts occurred before main()printed its first message, which is normal since interrupts are active as soon as the system starts.

After the loop ran for several seconds, clkcounterfine incremented by approximately 4022, meaning about 4.022 seconds elapsed, while clktime, which increments every 1000 milliseconds, reached 4.

This confirms that both counters are accurately tracking time, with clktime acting as a coarse-grained timekeeper and clkcounterfine providing a more precise millisecond-level measure. The slight timing difference between when the system starts handling interrupts and when main() prints its initial values accounts for any initial discrepancy.

4.2

I used inline assembly (cli and sti) to prevent clock interrupts from firing during a long for-loop in main(). The cli instruction disables interrupts before the loop, ensuring no clock updates occur, while sti re-enables them afterward.

When interrupts were enabled, clkcounterfine increased by approximately 1000 ticks per second, and clktime advanced by 1 per second, confirming normal behavior.

However, when interrupts were disabled, both counters remained nearly unchanged, proving that without hardware clock interrupts, the OS loses track of time.

Testing-

```
#include <xinu.h>
extern unsigned int clkcounterfine;
extern uint32 clktime;

process main(void) {
    kprintf("Initial clkcounterfine = %u, clktime = %u\n", clkcounterfine, clktime);

    asm("cli");

    volatile uint32 i;
    for (i = 0; i < 200000000; i++) {
        //long loop
}
    asm("sti");

    kprintf("After loop: clkcounterfine = %u, clktime = %u\n", clkcounterfine, clktime);
    return OK;
}
```

Outputs-

Initial clkcounterfine = 10, clktime = 0
After loop: clkcounterfine = 4000, clktime = 4

After interrupts disabled-

Initial clkcounterfine = 12, clktime = 0
After loop: clkcounterfine = 12, clktime = 0

4.3

QUANTUM defined the fixed time slice allocated to a process before the scheduler forcibly preempts it. By default, QUANTUM = 2, meant that a process gets 2 clock ticks before being switched out. The preempt variable, which tracks the remaining time slice, is decremented by clkhandler() every clock interrupt.

When it reaches 0, XINU's scheduler (resched()) is called to select the next highest-priority ready process. By increasing QUANTUM to 8 in kernel.h, I allowed processes to run 8 ticks before preemption, resulting in fewer context switches.

Output when QUANTUM =2

ABCABCABCABC...

Output when QUANTUM is modified to 8-

AAAAAAAABBBBBBBBCCCCCCCC...

So, Increasing QUANTUM reduces context-switching frequency, allowing processes to execute for longer uninterrupted periods. This decreases scheduling overhead and makes output less interleaved.

5.1

Assembly functions must adhere to the cdecl calling convention, where certain registers must be saved and restored properly to avoid unintended side effects. The provided addfour() function, takes four integers as arguments, adds them, and returns the sum in EAX.

However, it modifies EBX without saving it, violating the cdecl rule that EBX is callee-saved. This means if the calling function (e.g., main()) relies on EBX, its value may be corrupted after calling addfour(), leading to potential bugs.

To fix this, I modified addfour.S to push EBX onto the stack at the start and pop EBX before returning, ensuring its value remains unchanged when the function exits.

Fix-

```
.text
.globl addfour
.type addfour, @function

addfour:
    pushl %ebp
    movl %esp, %ebp

    pushl %ebx  # Save EBX before modifying it

    movl  8(%ebp), %eax  # Load first argument
    movl 12(%ebp), %ebx  # Load second argument
    movl 16(%ebp), %ecx  # Load third argument
    movl 20(%ebp), %edx  # Load fourth argument

    addl %ebx, %eax
    addl %ecx, %eax
    addl %edx, %eax

    popl %ebx  # Restore EBX before returning

    popl %ebp
    ret
```

Testing-

```
#include <xinu.h>
extern int addfour(int, int, int, int);

process main(void) {
    kprintf("Testing addfour...\n");
    int sum = addfour(10, 20, 30, 40);
    kprintf("addfour(10,20,30,40) = %d\n", sum);
    return OK;
}
```

Output-

```
Testing addfour...
addfour(10,20,30,40) = 100
```

5.2

As above, calling C functions from assembly follows the cdecl calling convention, where arguments are pushed onto the stack in reverse order and the return value is stored in EAX.

This exercise required writing a C function(addfourC()) that adds four integers and an assembly function (testaddfourC()) that calls addfourC(2,3,2,2) and returns its result to main(). The key challenge is ensuring proper stack manipulation in testaddfourC.S, particularly handling argument passing and cleanup.

Making addfourC.c-

```
#include <xinu.h>

int addfourC(int a, int b, int c, int d) {
    return a + b + c + d;
}
```

testaddfourC.S-

```
.text
.globl testaddfourC
.type testaddfourC, @function

testaddfourC:
    # Push arguments in reverse order (cdecl convention)
    pushl $2     # d
    pushl $2     # c
    pushl $3     # b
    pushl $2     # a

    # Call addfourC()
    call addfourC

    # Cleanup: Remove 4 arguments (16 bytes) from stack
    addl $16, %esp

    # Return (EAX already contains result)
    ret
```

Testing-

```
#include <xinu.h>
extern int testaddfourC(void);

process main(void) {
    kprintf("Calling testaddfourC... \n");
    int sum = testaddfourC();
    kprintf("testaddfourC returned %d (expected 9)\n", sum);
    return OK;
}
```

Output-

```
Calling testaddfourC...
testaddfourC returned 9 (expected 9)
```

Extra credit-

The function creates() is a wrapper around create() that simplifies process creation by automatically setting default values for stack size, priority, and process name while still allowing arguments to be passed.

The approach involves defining a new system parameter PROCSTACKSZ in include/process.h with a value of 8192.

In creates(), we retrieve the parent's priority from proctab[currpid] and set the new process's priority to parent_prio + 1. The function then calls create(), passing the fixed stack size, computed priority, and default name "NONAME", while forwarding the remaining arguments using va_list.

To test, we replace create() calls with creates() in a test scenario, verifying that the new processes have the expected stack size, priority, and name. Re-running the third scenario from 3.2 ensures that scheduling behavior remains consistent, confirming that creates() properly integrates with XINU's process management.