LAB 1

Anirudh Pal (pal5)

Part 3.1: Alternate mechanism for CPU idling would be to use an infinite while loop with a halt instruction in the loop. This would suspend the CPU till the next clock interrupt. After applying the change it didn't change its behaviour and operated as normal. It might have saved some power by not exhaustively running an empty loop.

Code:

```
while (TRUE) {
     __asm__("hlt;");
}
```

Part 4.1: Here is the output when the sleep() time is 123 seconds.

Output:

```
Going to sleep for 123 to test xuptime().
upmin: 2
upsec: 3
```

Part 5.1: Here is the output when you try to change byte order of 0x12ABCDEF or 313249263.

Output:

```
Attempting to Reverse byte order for 0x12ABCDEF.
revbyteorder_asm(): 0xEFCDAB12
revbyteorder_inline(): 0xEFCDAB12
revbyteorder(): 0xEFCDAB12
revbyteorder_gcc(): = 0xEFCDAB12
```

Part 5.2: Here is the output from the call of printsegaddress().

Output:

```
Calling printsegaddress().
End of Text at 0x0011BB10.
End of Data at 0x001202DF.
End of Bss at 0x001402C7.
Printing 6 bytes before 0x0011BB10:
0x11bb0f: a
0x11bb0e: 0
0x11bb0d: 20
0x11bb0c: 29
0x11bb0b: 73
0x11bb0a: 28
Printing 6 bytes before 0x001202DF:
0x1202de: 11
0x1202dd: b4
0x1202dc: 1e
0x1202db: 0
0x1202da: 11
0x1202d9: b4
Printing 6 bytes before 0x001402C7:
0x1402c6: 0
0x1402c5: 0
0x1402c4: 0
0x1402c3: 0
0x1402c2: 0
0x1402c1: 0
Printing 6 bytes after 0x0011BB10:
0x11bb11: 66
0x11bb12: 90
0x11bb13: 66
```

```
0x11bb14: 90
0x11bb15: 66
0x11bb16: 90
Printing 6 bytes after 0x001202DF:
0x1202e0: 0
0x1202e1: 0
0x1202e2: 0
0x1202e3: 0
0x1202e4: 0
0x1202e5: 0
Printing 6 bytes after 0x001402C7:
0x1402c8: 55
0x1402c9: 55
0x1402ca: aa
0x1402cb: aa
0x1402cc: 55
0x1402cd: 55
```

Part 5.3: Here is the test code output. (Commented out in main.c since files change.)

Output:

```
Run Time Stack Top Tests.
Before creating myprogA().
Address: 0x0EFC8FC0, Value: 0x00000000
Before calling myfuncA().
Address: 0x0FDEFFDC, Value: 0x00000000
In myfuncA().
After myfuncA() returns.
Address: 0x0FDEFFDC, Value: 0x00000000
After completion of myprogA()
Address: 0x0EFC8FC0, Value: 0x00000000
```

The first address is the main() process runtime stack top. The seconds address is the myprogA() process runtime stack top which is in a different frame compared to main() since it is a process. The third address is the same as the second one since the runtime stack resizes down to original after calling and getting a return from myfuncA(). The last address is the main() process runtime stack top which became bigger after resume() and create() call but downsized back to its original size after they returned.

Part 5.4: Here is the test code output. (Commented out in main.c since files change.)

Output:

```
Comparison of Run Time Stack.
Before calling myfuncA().
Stack Base: 0x0FDEFFFC, Stack Size: 1024, Stack Limit:
0x0FDEFEFC, Stack Pointer: 0x0FDEFFBC, PID: 4, Parent PID:
3
In myfuncA().
Stack Base: 0x0FDEFFFC, Stack Size: 1024, Stack Limit:
0x0FDEFEFC, Stack Pointer: 0x0FDEFF6C, PID: 4, Parent PID:
3
After myfuncA() returns.
Stack Base: 0x0FDEFFFC, Stack Size: 1024, Stack Limit:
0x0FDEFEFC, Stack Pointer: 0x0FDEFFBC, PID: 4, Parent PID:
3
In myfuncA().
Stack Base: 0x0FDEFFFC, Stack Size: 1024, Stack Limit:
0x0FDEFEFC, Stack Pointer: 0x0FDEFFB8, PID: 5, Parent PID:
3
```

Part 6: Here is the output before and after enabling the malware code.

Output (Before):

```
Attempting Hijack.
```

```
Before calling myfuncA().
Stack Base: 0x0FDEFFFC, Stack Size: 1024, Stack Limit:
0x0FDEFEFC, Stack Pointer: 0x0FDEFFBC, PID: 4, Parent PID:
3
In myfuncA().
Stack Base: 0x0FDEFBFC, Stack Size: 1024, Stack Limit:
0x0FDEFAFC, Stack Pointer: 0x0FDEFBB8, PID: 5, Parent PID:
4
After myfuncA() returns.
Stack Base: 0x0FDEFFFC, Stack Size: 1024, Stack Limit:
0x0FDEFEFC, Stack Pointer: 0x0FDEFFBC, PID: 4, Parent PID:
-1
```

Output (After):

```
Attempting Hijack.
Before calling myfuncA().
Stack Base: 0x0FDEFFFC, Stack Size: 1024, Stack Limit:
0x0FDEFEFC, Stack Pointer: 0x0FDEFFBC, PID: 4, Parent PID:
3
In myfuncA().
Stack Base: 0x0FDEFBFC, Stack Size: 1024, Stack Limit:
0x0FDEFAFC, Stack Pointer: 0x0FDEFBB8, PID: 5, Parent PID:
4
hijack succeeded
/="-=L[j&;
```

As you can see that after the hijacking code is executed, myprogA() never gets an opportunity to finish as its context has been hijacked and terminated with malwareA(). Thus it doesn't print the third output.

The strategy used to find the return address for sleepms() call was to take the process stack base from process table and then print the entire stack. Then you look for 3000 since it is an argument for sleepms() and the return address is next location down (CDECL). All we need to do is change that return address.

Code:

```
// Attack Code
uint32* base = (uint32*)proctab[getppid()].prstkbase;
int i;
for(i = 0; i < (proctab[getppid()].prstklen / 4); i++)
     kprintf("base - %d: %d\n", i, *(base - i));
*(base - 21) = (uint32)&malwareA;
```