Problem 3:

1:

If we consider the small program to add two numbers:

```
Push ebp
Mov ebp, esp
Mov eax, 3
Mov ebx, 4
Add eax, ebx
Pop ebp
Ret
```

As we can see there are no privileged instructions in the above code so there will be no switching from user mode to kernel mode and guest kernel will be running in user mode. Therefore, there will be no overhead in the above-mentioned code without privileged instructions.

No, There will be not much overhead when syscalls are invoked and no privileged instructions are executed in guest kernel because hypervisor is not involved and it is a jump in user mode itself from app to guest kernel. When we consider getpid() it is a just a table lookup call and it does not require hypervisor, therefore will not have any system call.

2:

Privileged instructions require hypervisor to execute as privileged instruction needs data. Yes, the overhead occurs when there is execution of privileged instructions. Suppose our guest kernel executes cli/sti , then it goes to hypervisor as it is a trap. Then hypervisor catch it and runs it code which disables global interrupt until cli of the guest kernel is released. Therefore, to protect disabling of global interrupt, hypervisor use sandboxing. And only interrupts that affects a particular guest kernel (who executed cli) will only be disabled not for others. We make virtual cli to perform this action. When guest kernel executes privileged instruction, it passes it to hypervisor. Hypervisor makes sure that these guest kernels are not buggy so it takes it as a user mode and put sandboxing around each guest kernel and there apps from other guest kernels and apps. Sandboxing is provided around the processes and kernels individually. Hypervisor provides sandboxing to each guest kernel and process together also.

3:

Sensitive instructions are like popf, when we context switch between two process and we save the state of current process and load the state of other process and run it. There are flags register like eflags: IF – cli,sti and IOPL-2bits where hardware compare the current privileged level. When current privileged instruction <= IOPL then instruction go through. By default CPL is 11 and IOPL 00. Popf is sensitive instruction stores the value of registers in Eflags when we context switch. Eflags are stored in User space. It does not let the user change IOPL value to 11. If popf is executed as a privileged instruction in user mode by a guest kernel then it make system buggy and system will crash. Popf decides whether it should be run by user

mode or kernel mode then it does not work in full virtualization as it is supposed to be only for privileged or non-privileged instructions and not sensitive instructions. To solve this problem VMware came up with a technique JIT translation, it is like an interpreter, doesn't precompile the code, it compiles it at runtime

4:
When the code is compiled then JVM translates the code into byte code that is kept in cache and code has locality reference. The locality reference brings overhead as it takes time to convert to byte code first then to cache it and remove sensitive instructions and change it into only privileged and non-privileged instructions. But Overhead can be overcome only if same code is run again and again and it turns out to give amortized cost of overall translation-locality reference. It differs from full virtualization as in full virtualization we have only 2 types of instructions: privileged and non-privileged instructions and not sensitive instructions hence it brings less overhead in the system than para-virtualization which interprets the code first and then it compiles at run time. Overhead in full virtualization comes only when privileged instructions are executed which requires the involvement of hypervisor.

For example the following java code:

```
public class BasicJavaProgram
{

    public static void main(String[] args)
    {
        int number1 = 1;
        int number2 = 5;

        //calculating number1 + number2;
        int sum = number1 + number2;

    }
}
```

So to compile the code we need to interpret the code and change it into byte code with help of JVM using *javac BasicJavaProgram.java* and then compiling the code at run time using *java BasicJavaProgram*

Problem 4:

1. When running the instructions, yes the 'P' characters printed by the parent process were interleaved by outputs from its children. The output is:

```
P
11
11
11
11
11
P
22
22
22
```

```
22
22
P
33
33
33
33
33
P
44
44
44
44
44
```

Hence, it also shows that xinu implements static scheduling because it did not change the priority of any of the process throughout the execution.

2. Yes, by changing the INITPRIO to 35, it changes the priority of main to 35 which now has the highest priority so it executes first before executing all it's children. After it finishes printing all P's then children are executed.
3. Now, first 3 children has same priority as of parent and 4th child has 50 as a priority because of which all three child run but as soon as 4th child executes it is always assigned priority of 50 and it does not let any other process run until it finishes. So when it finishes then other children are allowed to execute.
4. When the experiment is done using sleepms(1) then when process sleeps for 1ms it then process sleeps and NULL process runs. With sleepms(0) it shows that NULL process is always running as it has priority of 0. When processes completed executing then NULL process starts executing until it a process of higher priority comes.
5. To implement CPU time for a process, it is incremented every millisecond whenever clkhandler.c executes. For every process, prcputime is maintained and when a clkhandler() runs for a process it's prcputime is recorded by incrementing prcputime in clkhandler. It keeps the value of cpu time taken by an individual process in millisecond.