SKILLS TEST TWO: Soln

Preliminary questions

1. **Why is it hard to have a correct clock source on multi-core systems?**
   Time is executed by a mechanic clock to which the CPUs are connected. Changes in CPU frequency or power (electric supply) makes the time returned per CPU different. Therefore making it difficult to have a correct clocksource on multi-core systems. Also, clocks may stop when the system goes to an idle state, or become out of sync when their CPUs enter energy saving states or perform speed- or frequency-scaling operations.

2. **What difference do you make between a MONOTONIC and REAL-TIME clock source?**
   REAL-TIME gives time passed since the linux epoch (January 1 1970), while MONOTONIC gives time passed since a fixed starting point usually since system boot.

First step: Build your own linux kernel

To do this, Simply follow instructions in doc and debug where necessary: had one major issue, that of certs.
Fix: Comment out the lines $CONFIG\_SYSTEM\_TRUSTED\_KEY$ and $CONFIG\_MODULE\_SIG\_KEY$.

I want the kernel to have my name

To do this, modify value of variable $EXTRAVERSION$ in Makefile

I shall be the time

3. **Build, and test if your new clock source does not crash the kernel.**
   Changes made to the file $arch/x86/kernel/tsc.c$ are as shown in the screenshots. Figure 1 illustrates the custom clocksource created (zcs).
   NB: Since most changes here were meant to be like tsc, some functions were also directly edited to have similar effect on our custom clocksource (zcs) Other changes that were recorded were as shown (figures, 2, 3 and 4):

I'm better than you IIIIIIIIIII

4. **Analyze the results obtained** Results obtained show an increasing straight line for all the clocksources (cf Figure 5).

5. **Analyze your results and comment on your clock synchronization accuracy** The results obtained show very little or unnoticeable (with the naked eye) changes from values for the various clocksources and threads. We therefore can say that the synchronization is very high and accurate. (cf Figure 6, 7, 8 and 9)

```
/// MY CUSTOM CLOCKSOURCE (zcs: zidane's clocksource) AND ITS FUNCTIONS:
// CLOCK_SOURCE_VALID_FOR_HRES flag removed so that zcs will not be used immediately

static u64 read_zcs(struct clocksource *cs)
{
    struct task_struct *p = current;
    u64 tsc_time  = (u64)rdtsc_ordered();
    return (tsc_time - p->se.statistics.sleep_max);
}

static struct clocksource clocksource_zcs = {
    .name                   = "zcs",
    .rating                 = 275,
    .read                   = read_zcs,
    .mask                   = CLOCKSOURCE_MASK(64),
    .flags                  = CLOCK_SOURCE_IS_CONTINUOUS |
              CLOCK_SOURCE_MUST_VERIFY,
    .archdata               = { .vclock_mode = VCLOCK_TSC },
    .resume         = tsc_resume,
    .mark_unstable     = tsc_cs_mark_unstable,
    .tick_stable       = tsc_cs_tick_stable,
    .list       = LIST_HEAD_INIT(clocksource_zcs.list),
};


// ENDS HERE ;-)
```

Figure 1: Structure of clocksource zcs (Zidane's clocksource)

```
static int __init init_tsc_clocksource(void)
{
    if (!boot_cpu_has(X86_FEATURE_TSC) || !tsc_khz)
        return 0;

    if (tsc_unstable)
        goto unreg;

    if (tsc_clocksource_reliable || no_tsc_watchdog) {
        clocksource_tsc.flags &= ~CLOCK_SOURCE_MUST_VERIFY;
        clocksource_zcs.flags &= ~CLOCK_SOURCE_MUST_VERIFY;
    }

    if (boot_cpu_has(X86_FEATURE_NONSTOP_TSC_S3)) {
        clocksource_tsc.flags |= CLOCK_SOURCE_SUSPEND_NONSTOP;
        clocksource_zcs.flags |= CLOCK_SOURCE_SUSPEND_NONSTOP;
    }

    /*
     * When TSC frequency is known (retrieved via MSR or CPUID), we skip
     * the refined calibration and directly register it as a clocksource.
     */
    if (boot_cpu_has(X86_FEATURE_TSC_KNOWN_FREQ)) {
        if (boot_cpu_has(X86_FEATURE_ART))
            art_related_clocksource = &clocksource_tsc;
        clocksource_register_khz(&clocksource_zcs, tsc_khz); // register my clocksource zcs first
        clocksource_register_khz(&clocksource_tsc, tsc_khz);
unreg:
        clocksource_unregister(&clocksource_tsc_early);
        return 0;
    }

    schedule_delayed_work(&tsc_irqwork, 0);
    return 0;
}
```

Figure 2: Changes done to the *init_tsc_clocksource* function

```
out:
    if (tsc_unstable)
        goto unreg;

    if (boot_cpu_has(X86_FEATURE_ART))
        art_related_clocksource = &clocksource_tsc;
    clocksource_register_khz(&clocksource_zcs, tsc_khz); // REGISTER MY CLOCKSOURCE ZCS FIRST
    clocksource_register_khz(&clocksource_tsc, tsc_khz);
unreg:
    clocksource_unregister(&clocksource_tsc_early);
}
```

Figure 3: Change done to the *tsc_refine_calibration_work* function

```
void mark_tsc_unstable(char *reason)
{
    if (tsc_unstable)
        return;

    tsc_unstable = 1;
    if (using_native_sched_clock())
        clear_sched_clock_stable();
    disable_sched_clock_irqtime();
    pr_info("Marking TSC unstable due to %s\n", reason)

    clocksource_mark_unstable(&clocksource_tsc_early);
    clocksource_mark_unstable(&clocksource_zcs); // mar
    clocksource_mark_unstable(&clocksource_tsc);
}
```

Figure 4: Change done to the *mark_tsc_unstable* function

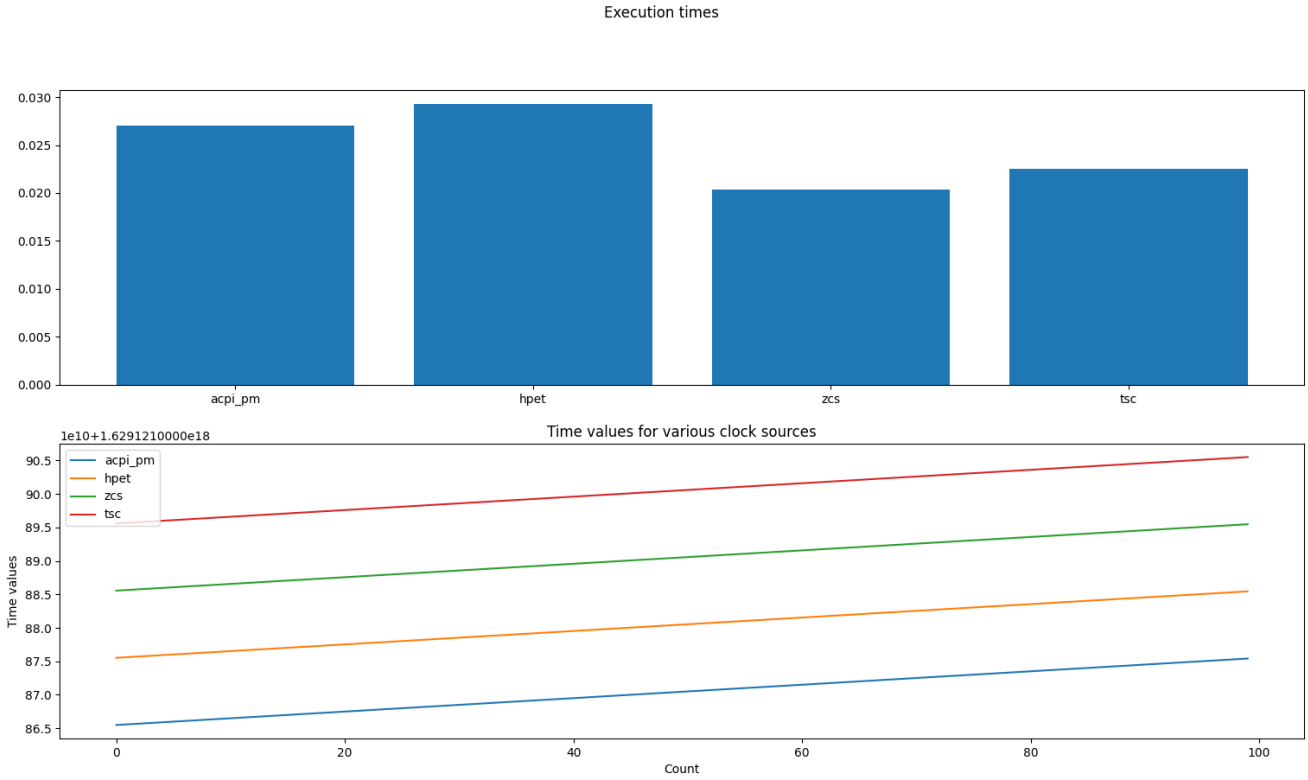

Figure 5: Benchmark 1. results

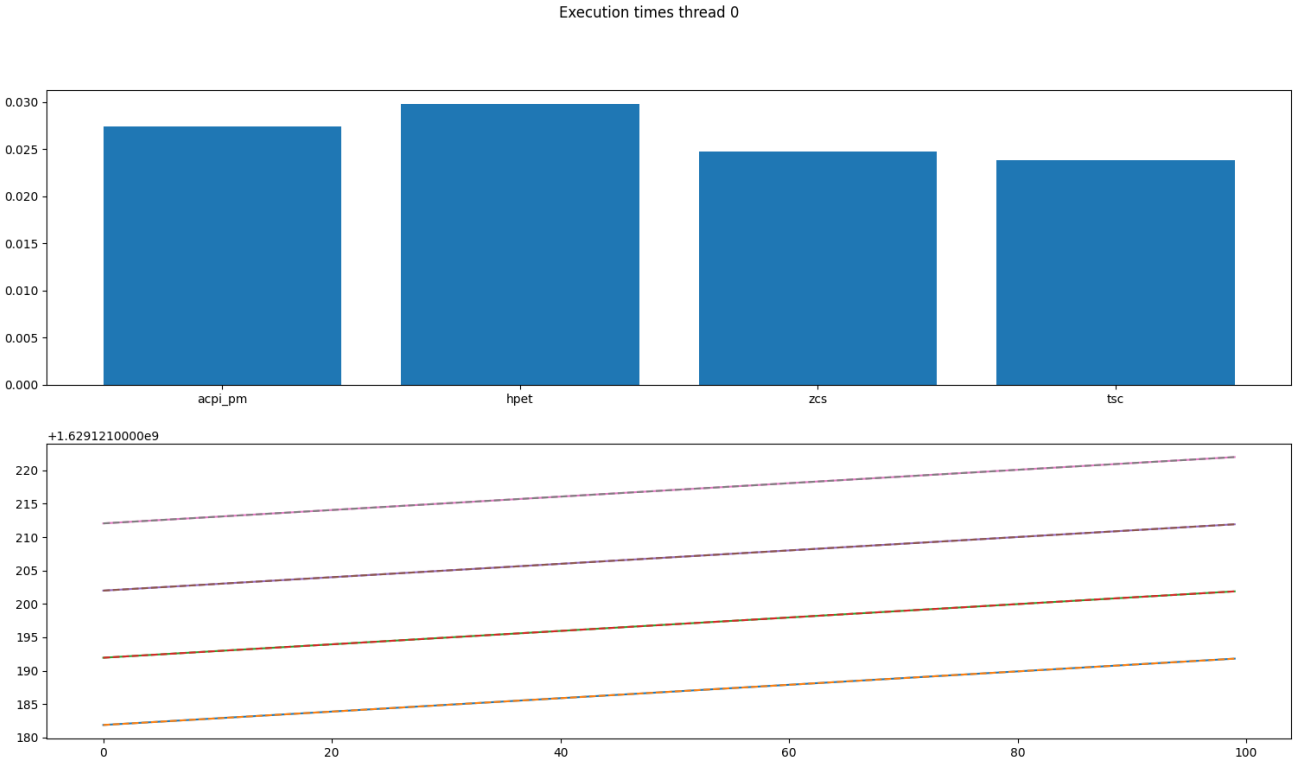

Figure 6: Benchmark 2, results thread 0

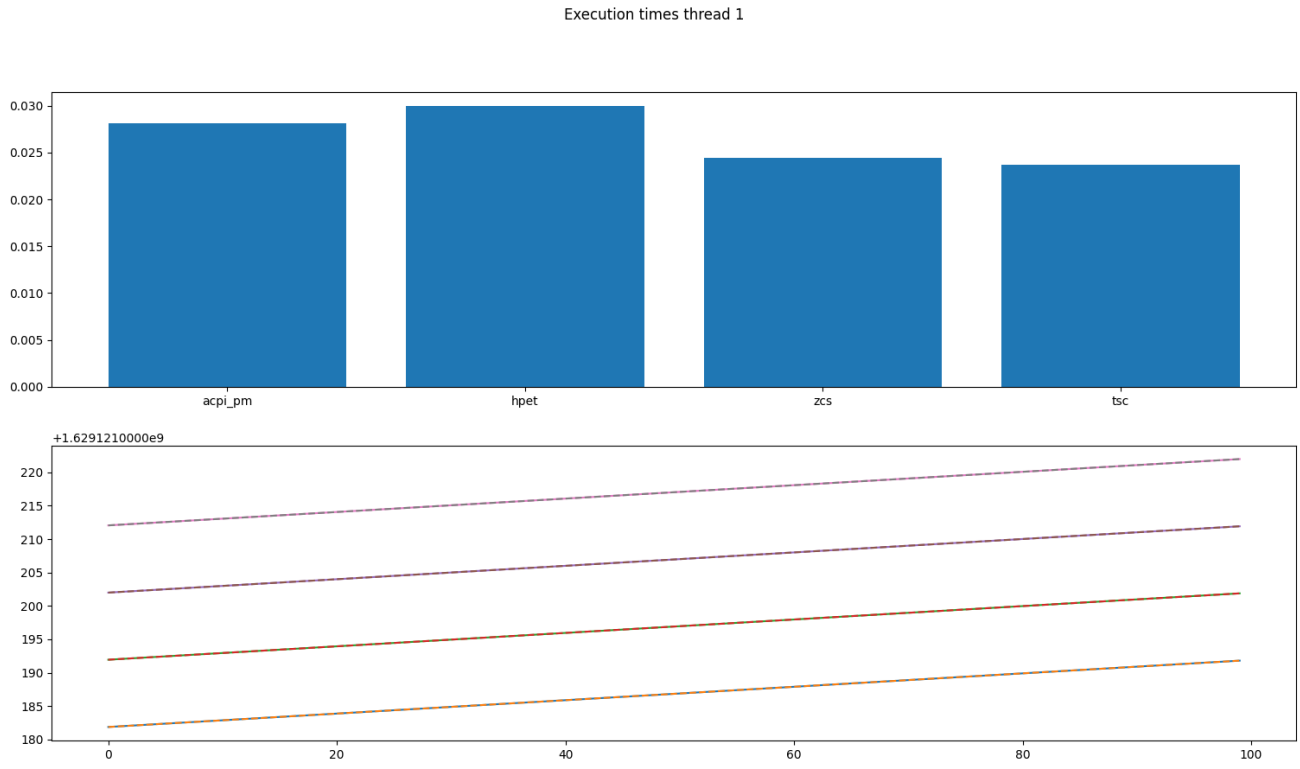

Figure 7: Benchmark 2, results thread 1

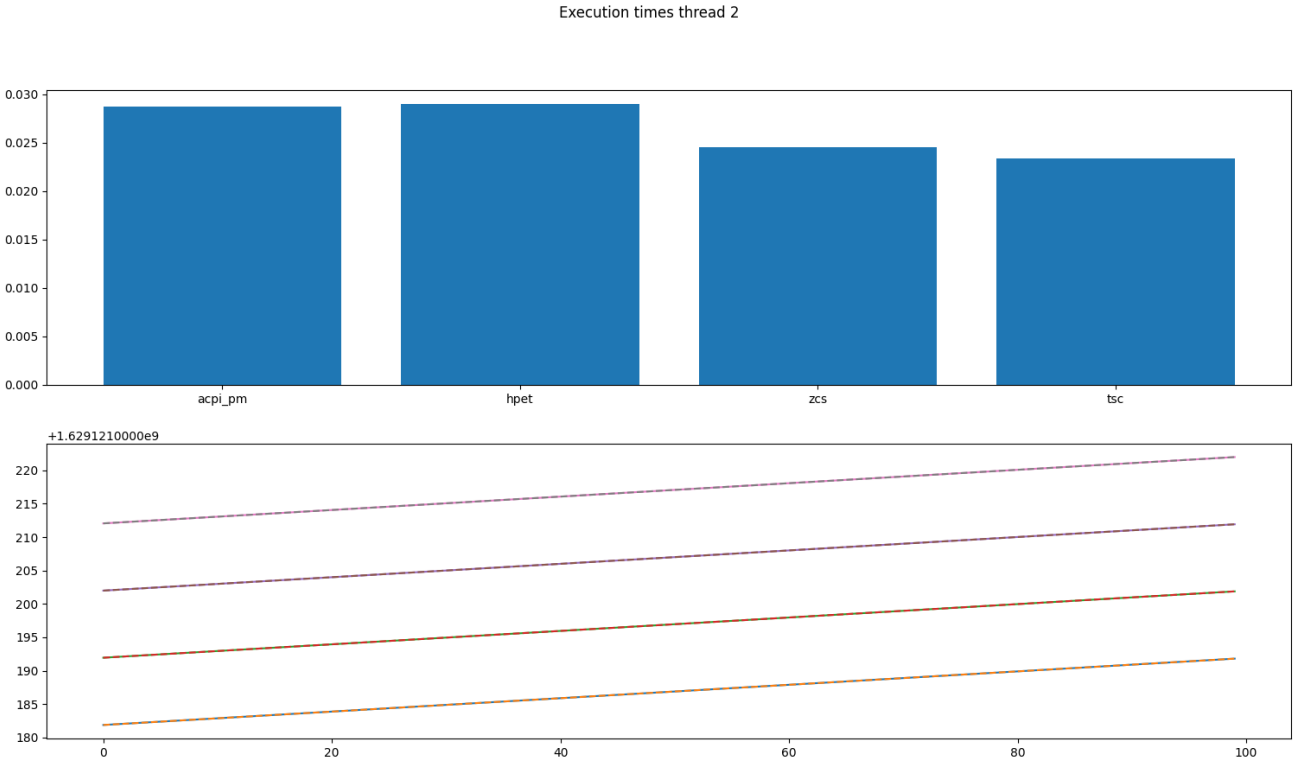

Figure 8: Benchmark 2, results thread 2

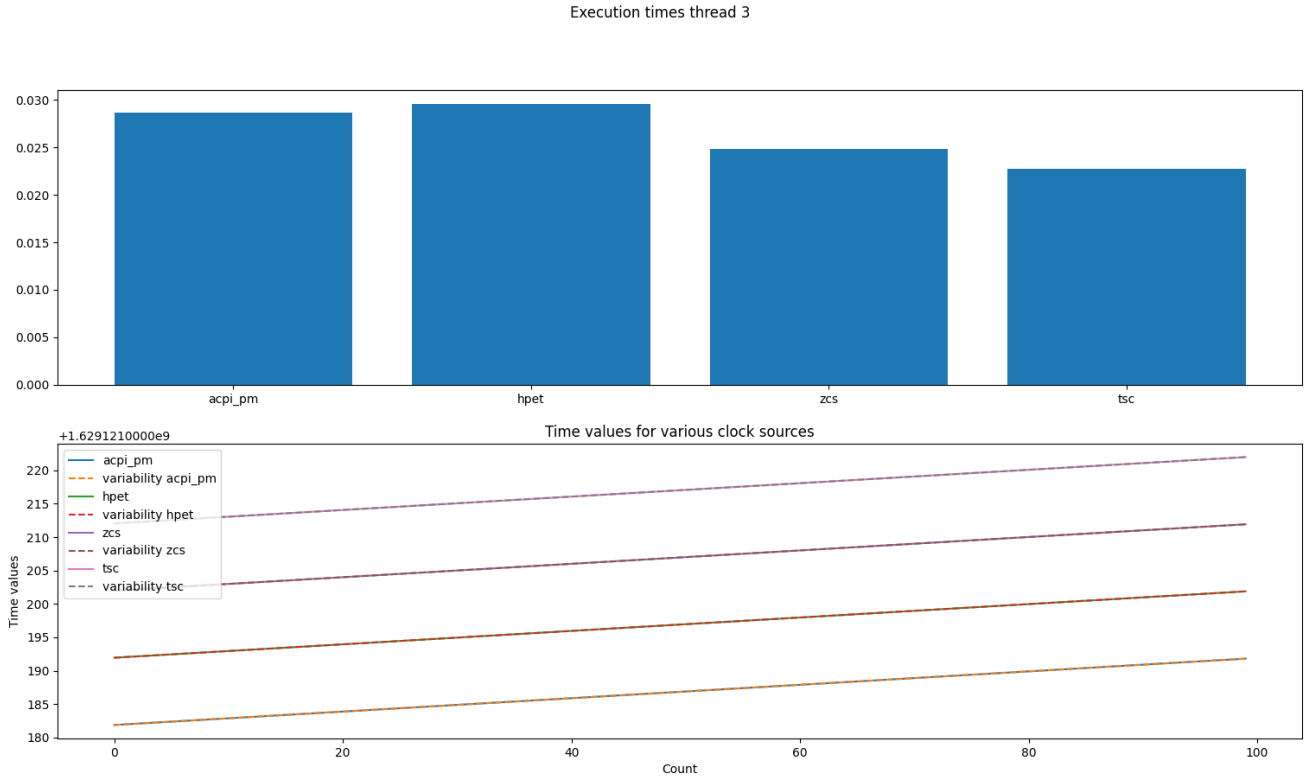

Figure 9: Benchmark 2, results thread 3