TOSHIBA
Leading Innovation

多核Linux系统上的核心分区技术

Core Partitioning Technique on Multicore Linux Systems

Kouta Okamoto, TOSHIBA Corporation
Japan Technical Jamboree 63
Dec 1st, 2017

Agenda

- Background
- Core Partitioning for User Processes
- Core Partitioning for Interrupts
- Core Partitioning for Kernel Threads
- Executing a Realtime Application
- Evaluating latency with cyclictest

Agenda

- **Background**
- Core Partitioning for User Processes
- Core Partitioning for Interrupts
- Core Partitioning for Kernel Threads
- Executing a Realtime Application
- Evaluating latency with cyclictest

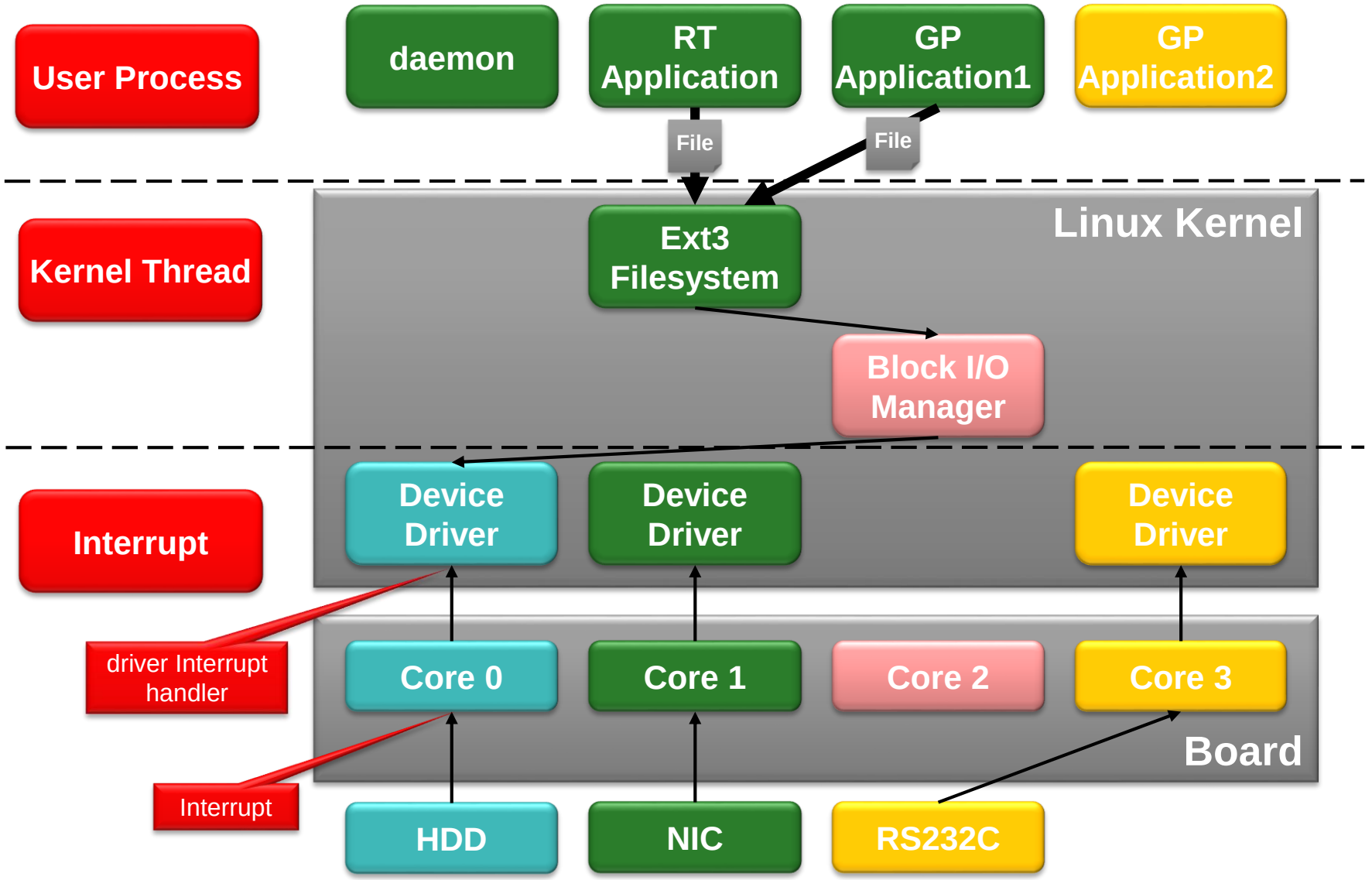
Background

- **Multicore CPUs became available for embedded systems**
  - Intel Apollolake
  - Rasphberry pi 3

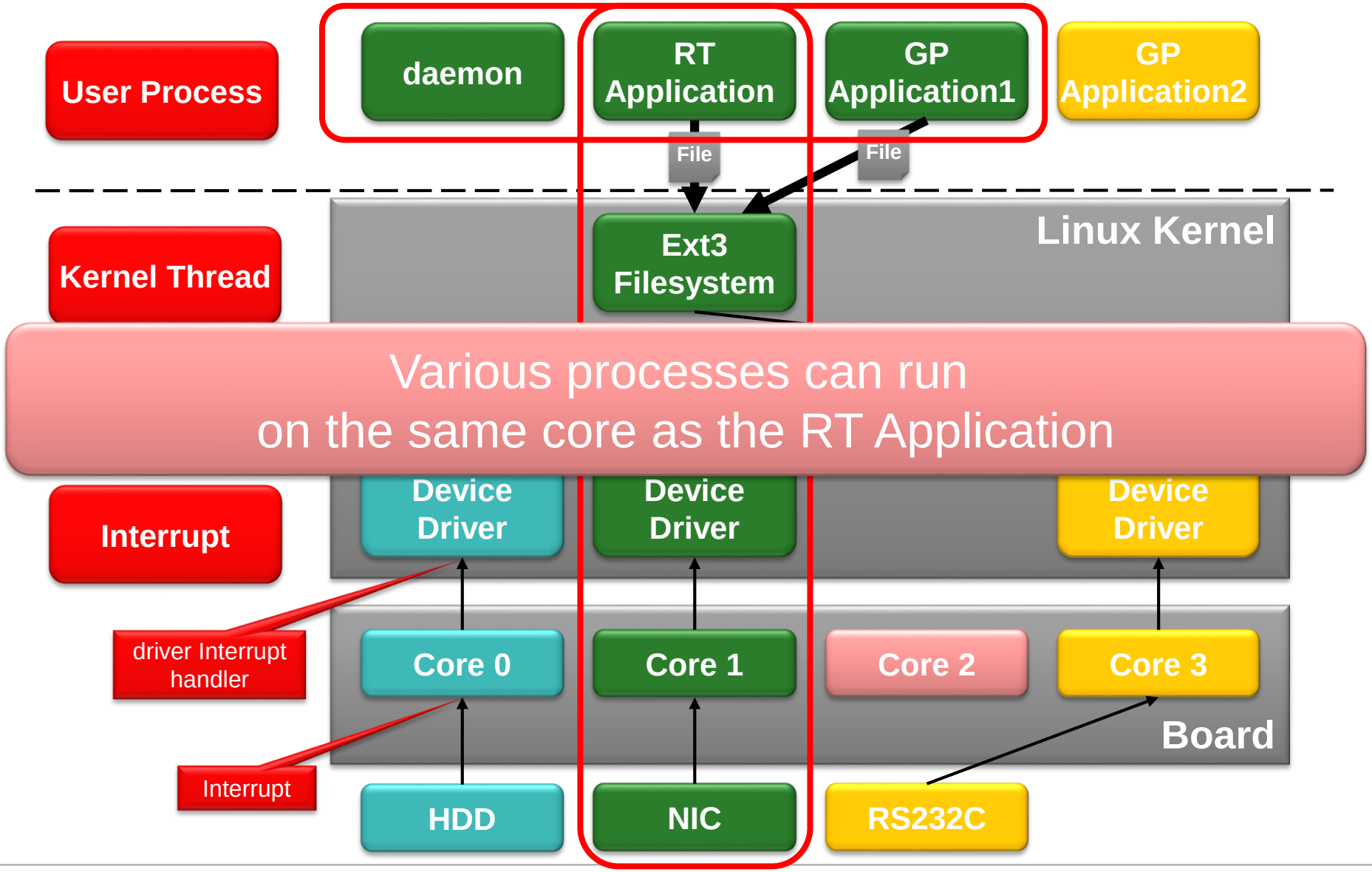
- **Advanced Requirments came up**
  - Realtime Applications
    - need to satisfy deadlines
    - E.g. controller
  - General Purpose Applications
    - provide additional value
    - E.g. http server

Run them on one board

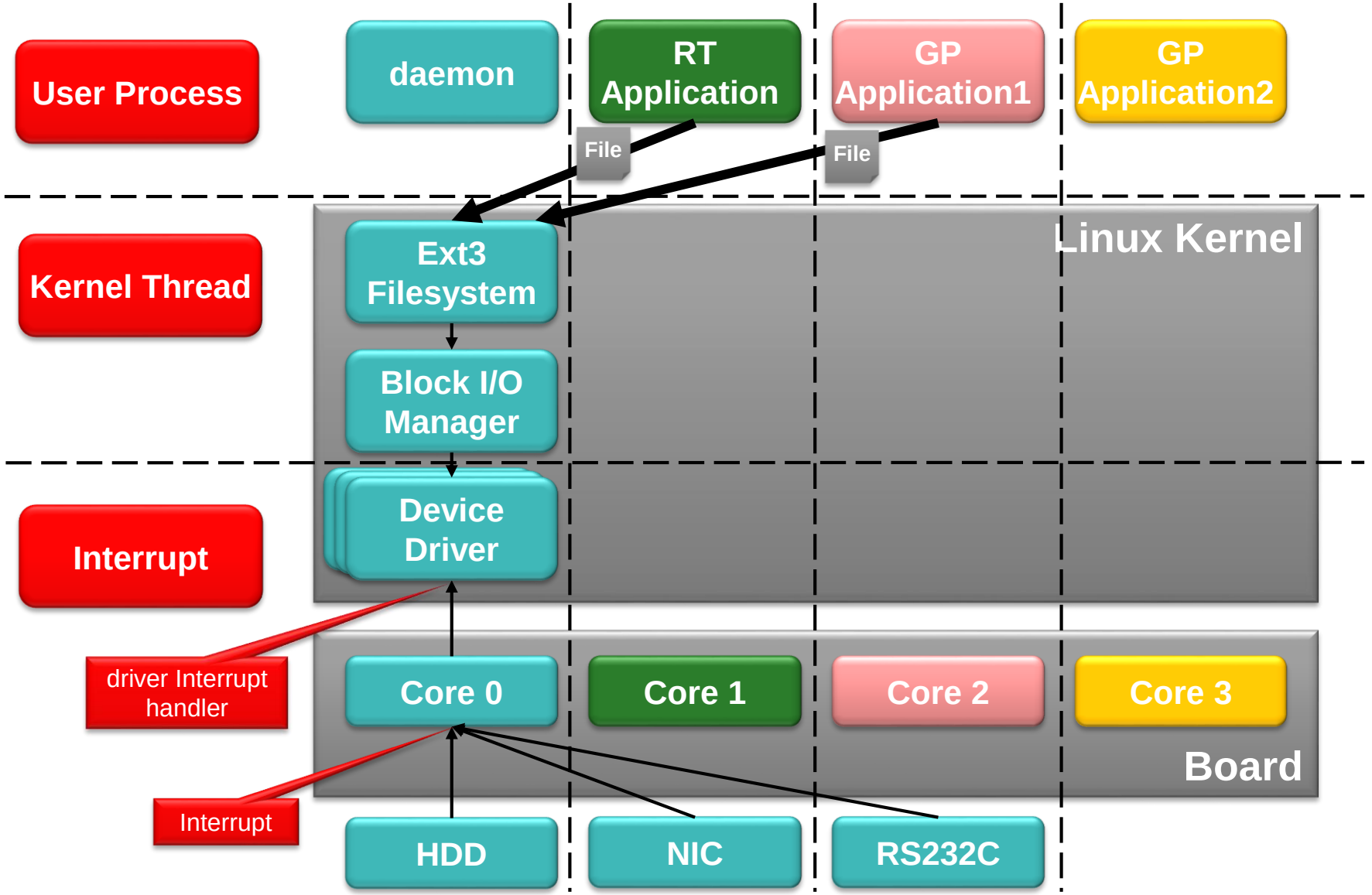
Example: a 4-core system on General Linux



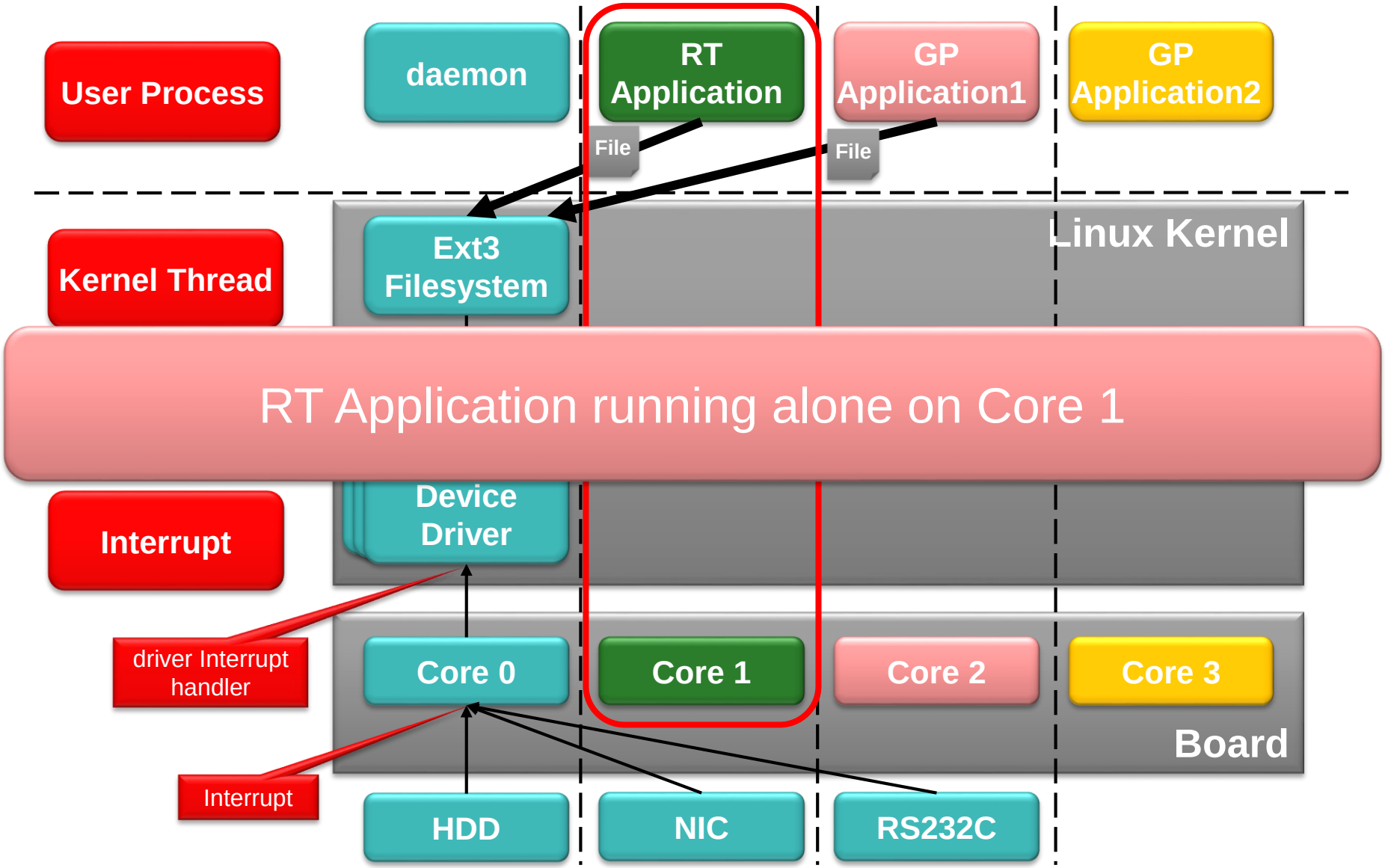
Example: a 4-core system on General Linux



Goal of Core Partitioning



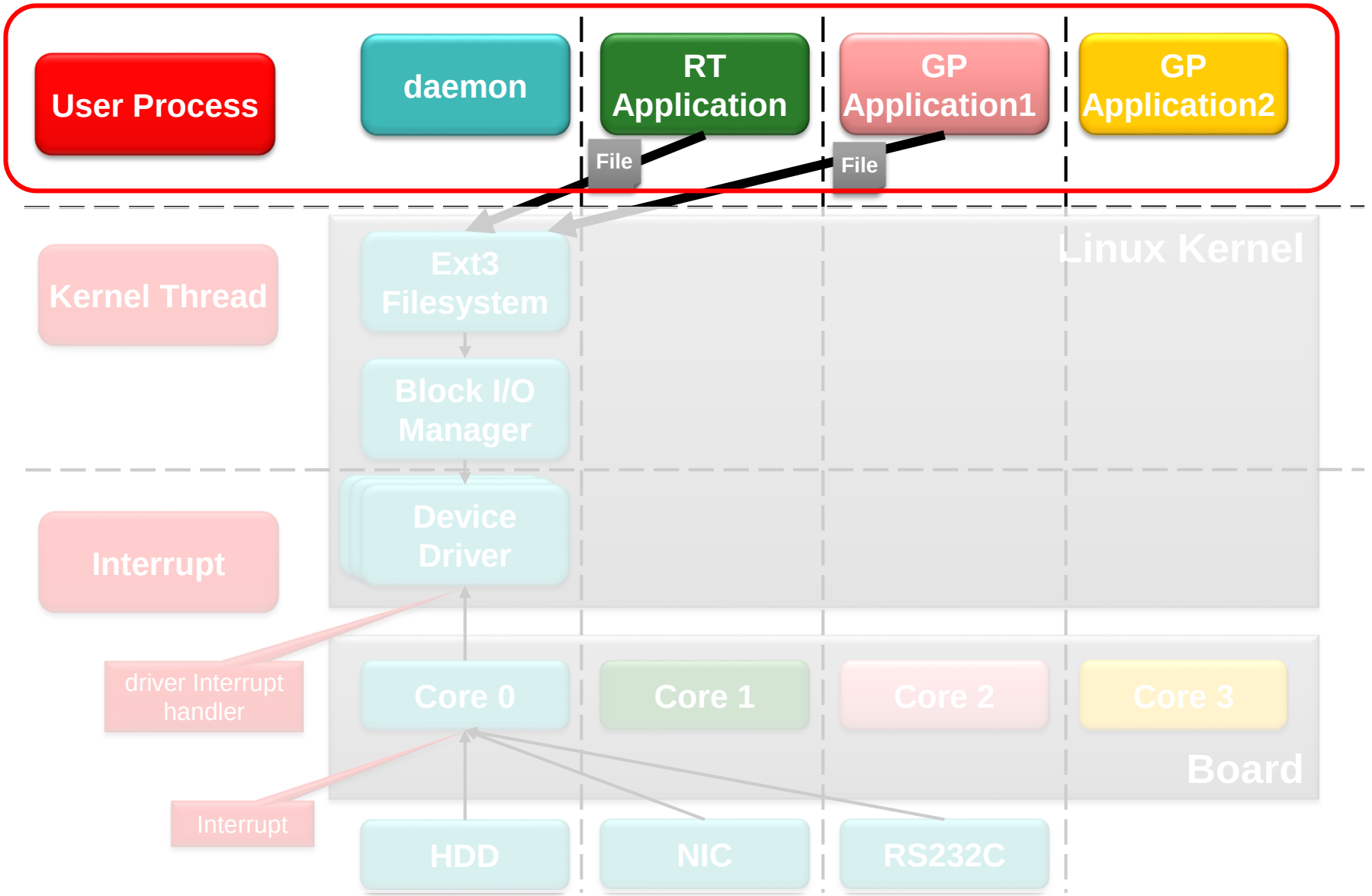
Goal of Core Partitioning



Agenda

- Background
- **Core Partitioning for User Processes**
- Core Partitioning for Interrupts
- Core Partitioning for Kernel Threads
- Executing a Realtime Application
- Evaluating latency with cyclictest

Core Partitioning for User Process



CPU affinity for user process

- **Default CPU affinity for user process**
  - Runnable on all CPU Core

taskset - 设置CPU亲和性

```
# taskset -p 1
pid 1's current affinity mask: f
```

- **Change default CPU affinity for user process**
  - kernel arguments to set user process CPU affinity to Core 0(avoid 1-3)

```
isolcpus=1-3
```

- **Check Result**
  - Only Runnable on Core 0

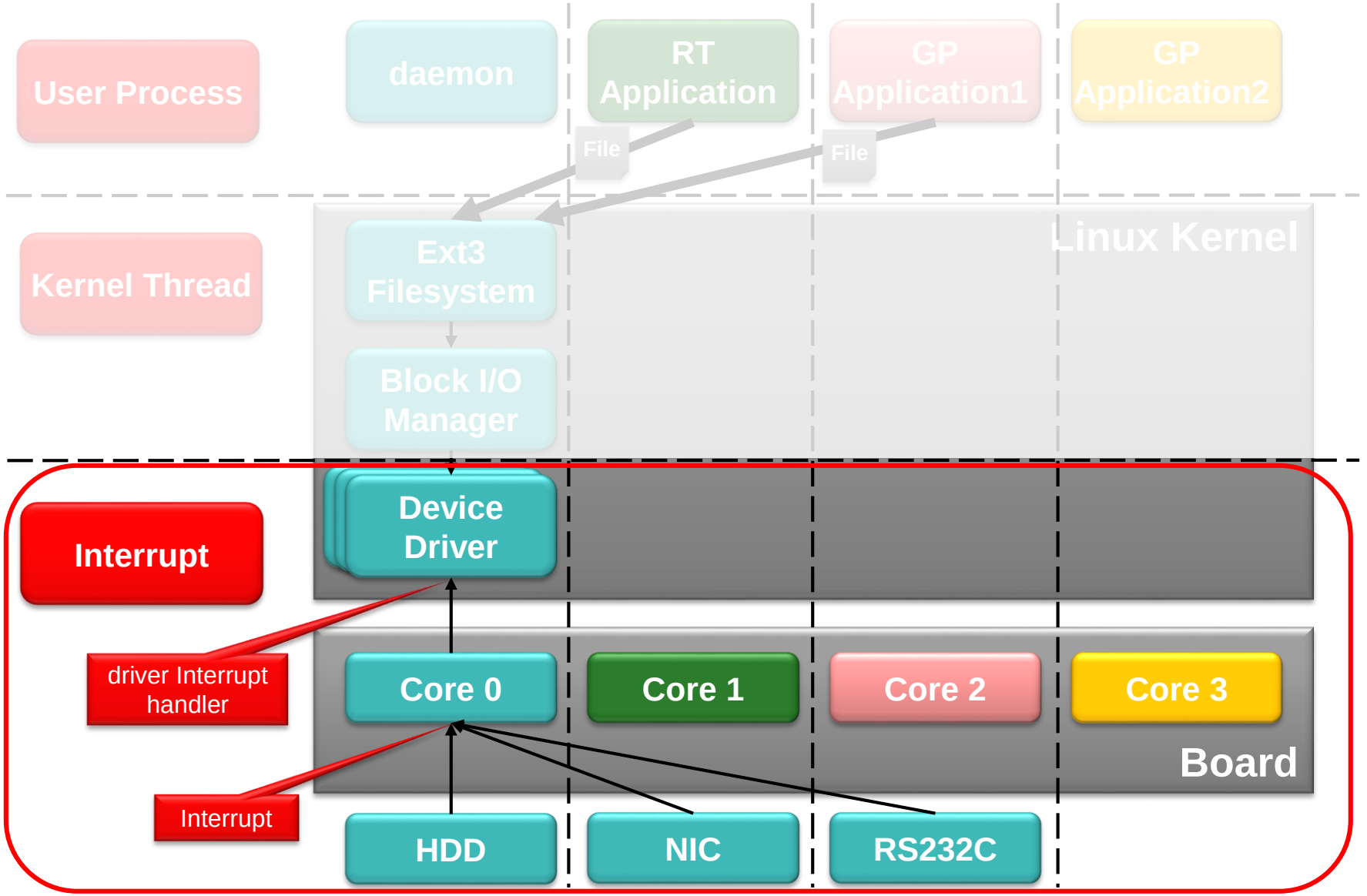
```
# taskset -p 1
pid 1's current affinity mask: 1
```

Does not affect to kernel thread!!

Agenda

- Background
- Core Partitioning for User Processes
- **Core Partitioning for Interrupts**
- Core Partitioning for Kernel Threads
- Executing a Realtime Application
- Evaluating latency with cyclictest

Core Partitioning for Interrupt



CPU affinity for interrupt

- **Default CPU affinity for interrupt**
  - Interruptible on all CPU

```
# cat /proc/irq/0/smp_affinity
f
```

为中断的CPU亲和性

- **Change CPU affinity for interrupt**
  - Change each interrupt CPU affinity to Core 0

```
# for file in `find /proc/irq -name "smp_affinity_list"`; do ¥
    echo 0 > ${file} 2>/dev/null; ¥
  done
```

  - Change default CPU affinity for interrupt to Core 0

```
# echo 1 > /proc/irq/default_smp_affinity
```

- **Check Result**
  - Only Interruptible on Core 0

```
# cat /proc/irq/0/smp_affinity
1
```

CPU affinity for driver interrupt handler

- **Driver interrupt handler**
  - In default, driver interrupt handler will be executed in irq context.
  - It may cause amount of latency, because irq context is not preemptible.

- **Change driver interrupt handler to kernel thread**
  - kernel argument to change the way to execute interrupt handler from irq context to kernel thread.

```
threadirqs
```

- **Check Result**
  - The interrupt kernel thread will be created such as “irq/<interrupt number>-<driver name>”

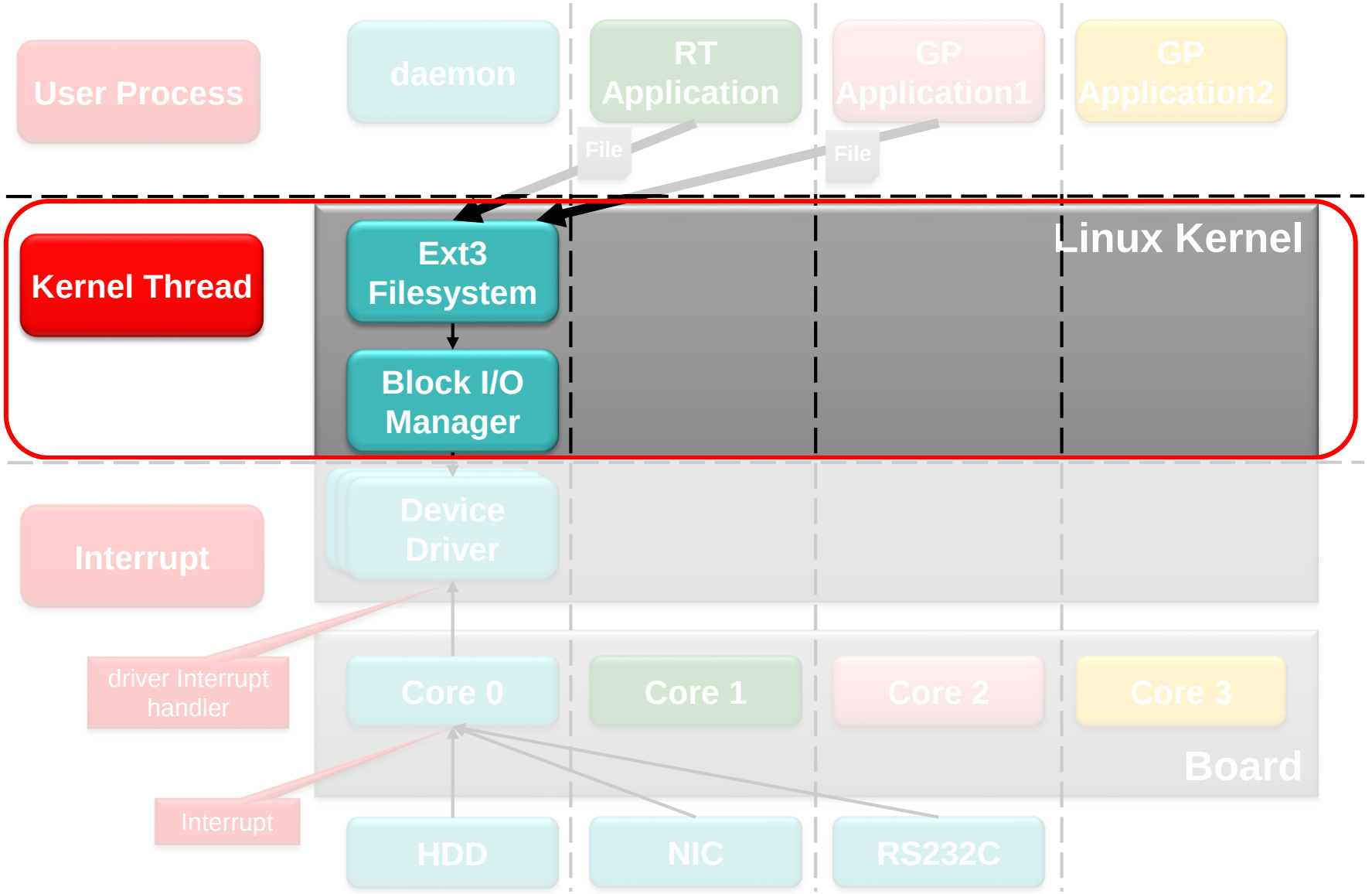
```
# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.1  28988  5272 ?        Ss   13:40   0:00 /sbin/init
...
root       206  0.0  0.0      0     0 ?        S    13:40   0:00 [irq/16-ehci_hcd]
root       207  0.0  0.0      0     0 ?        S    13:40   0:00 [irq/23-ehci_hcd]
root       209  0.0  0.0      0     0 ?        S    13:40   0:00 [irq/12-i8042]
```

irq handler already executed by kernel thread.
CPU affinity of them can be changed. See after next slide.

Agenda

- Background
- Core Partitioning for User Processes
- Core Partitioning for Interrupts
- **Core Partitioning for Kernel Threads**
- Executing a Realtime Application
- Evaluating latency with cyclictest

Core Partitioning for Kernel Thread



CPU affinity for kernel thread

- **Default CPU affinity for kernel thread**
  - Almost kernel thread are runnable on all CPU Core
    - CPU affinity for these thread can be changed by some way
      - taskset
      - cgroup <- select

- **The way to use cgroup**
  1. direct access to cgroup filesystem
  2. use libcgroup package
  3. use cpuset package <- select
     - The reason of this select is simplicity of cset command

- **Change CPU affinity for kernel thread**
  - Following command create grpup “cpu0” to run on Core0, and move all thread which include not only user process but also kernel thread to “cpu0” group.

```
# cset set -s cpu0 -c 0
# cset proc -m -k --force -f root -t cpu0
```

  - NOTE: init process should be on root group. Changing cgroup for init process cause wrong affect for container tool such as lxc.

```
# cset proc -m -p 1 -f cpu0 -t root
# taskset -p 1 1
```

CPU affinity for kernel thread

- **Limitation of kernel thread CPU affinity**
  - CPU bound kernel threads
    - Some kernel threads are bound to specified CPU cores. The CPU affinity of these threads can't be changed.
    - e.g.
      - CPU bound kernel thread is named such as "<name>/<core number>"

```
# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  29460  5472 ?        Ss   09:25   0:01 /sbin/init
...
root         9  0.0  0.0      0     0 ?        S    09:25   0:00 [migration/0]
root        10  0.0  0.0      0     0 ?        S    09:25   0:00 [watchdog/0]
root        11  0.0  0.0      0     0 ?        S    09:25   0:01 [watchdog/1]
root        12  0.0  0.0      0     0 ?        S    09:25   0:00 [migration/1]
root        13  0.8  0.0      0     0 ?        S    09:25   3:19 [ksoftirqd/1]
```

  - Dynamically created kernel thread
    - Some kernel thread are dynamically created on demand. If these thread will be created after setting of change kernel thread CPU affinity, it can be run at all CPU.
    - e.g.
      - kjournald will be created at the time of mount ext4 filesystem.

CPU affinity for worker thread

- **What is a worker thread.**
  - Workqueue is a delayed processing framework in Linux kernel. Worker threads have the responsibility to execute delayed callback handlers.
  - Woker thread have also two type, CPU bound and CPU unbound. CPU bound thread is named such as “kworker/<core number>:<id>”. CPU unbound thread is named such as “kworker/u<pool number>:<id>”.
  - e.g.
    - [kworker/0:1] is CPU bound worker thread.
    - [kworker/u8:2] is CPU unbound worker thread.

```
# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  29460  5472 ?        Ss   09:25   0:01 /sbin/init
...
root        16  0.0  0.0      0     0 ?        S    09:25   0:05 [kworker/0:1]
...
root       114  0.0  0.0      0     0 ?        S    09:25   0:00 [kworker/u8:2]
```

    - [kworker/0:1] is runnable on CPU Core 0

```
# taskset -p 16
pid 16's current affinity mask: 1
```

    - [kworker/u8:2] is runnable on All CPU Core

```
# taskset -p 114
pid 114's current affinity mask: f
```

CPU affinity for worker thread

- **Change CPU affinity for workqueue**
  - Change workqueue CPU affinity to Core 0
    - NOTE: In my machine, workqueue which have controllable cpumask is only “writeback”.

```
# for file in `find /sys/devices/virtual/workqueue "cpumask"`; do ¥
    echo 1 > ${file} 2>/dev/null; ¥
  done
```

  - Change default CPU affinity for worker thread to Core 0

```
# echo 1 > /sys/devices/virtual/workqueue/cpumask
```

- **Check Result**
  - new kworker kernel thread which have responsibility for above workqueue will be created.
    - [kworker/u9:0] is new worker thread.

```
# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  29460  5472 ?        Ss   09:25   0:01 /sbin/init
...
root        16  0.0  0.0      0     0 ?        S    09:25   0:05 [kworker/0:1]
...
root       114  0.0  0.0      0     0 ?        S    09:25   0:00 [kworker/u8:2]
...
root      1014  0.0  0.0      0     0 ?        S    09:25   0:00 [kworker/u9:0]
```

    - [kworker/u9:2] is runnable on CPU Core 0

```
# taskset -p 1014
pid 1014's current affinity mask: 1
```

Default CPU affinity for kernel thread

- **Default CPU affinity for kernel thread**
  - It can’t be changed in current Linux kernel
  - This is limitation for dynamically created kernel thread
    - We need to care such kernel thread like kjournald.

We attempt to create patch to change default CPU affinity for kernel thread

Patch for changing default CPU affinity for kernel thread(1/2)

```
diff --git a/kernel/kthread.c b/kernel/kthread.c
index 760e86d..2396194 100644
--- a/kernel/kthread.c
+++ b/kernel/kthread.c
@@ -23,6 +23,8 @@
 static DEFINE_SPINLOCK(kthread_create_lock);
 static LIST_HEAD(kthread_create_list);
 struct task_struct *kthreadd_task;
+static int enable_kthread_default_cpumask = 0;
+static struct cpumask kthread_default_cpumask;

 struct kthread_create_info
 {
@@ -282,7 +284,11 @@ struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),
                             * The kernel thread should not inherit these properties.
                             */
                            sched_setscheduler_nocheck(create.result, SCHED_NORMAL, &param);
-                           set_cpus_allowed_ptr(create.result, cpu_all_mask);
+                           if (enable_kthread_default_cpumask) {
+                                       set_cpus_allowed_ptr(create.result, &kthread_default_cpumask);
+                           } else {
+                                       set_cpus_allowed_ptr(create.result, cpu_all_mask);
+                           }
                }
                return create.result;
 }
@@ -450,7 +456,11 @@ int kthreadd(void *unused)
                /* Setup a clean context for our children to inherit. */
                set_task_comm(tsk, "kthreadd");
                ignore_signals(tsk);
```

Patch for changing default CPU affinity for kernel thread(2/2)

```
-               set_cpus_allowed_ptr(tsk, cpu_all_mask);
+               if (enable_kthread_default_cpumask) {
+                               set_cpus_allowed_ptr(tsk, &kthread_default_cpumask);
+               } else {
+                               set_cpus_allowed_ptr(tsk, cpu_all_mask);
+               }
                set_mems_allowed(node_states[N_MEMORY]);

                current->flags |= PF_NOFREEZE;
@@ -653,3 +663,16 @@ void flush_kthread_worker(struct kthread_worker *worker)
                wait_for_completion(&fwork.done);
 }
 EXPORT_SYMBOL_GPL(flush_kthread_worker);
+
+static int __init kthread_default_cpumask_setup(char *str)
+{
+               int ret;
+
+               ret = cpumask_parse(str, &kthread_default_cpumask);
+               if (!ret)
+                               enable_kthread_default_cpumask = 1;
+
+               return 1;
+}
+
+__setup("kthread_default_cpumask=", kthread_default_cpumask_setup);
```

kernel argument “kthread_default_cpumask=” can makes change default CPU affinity for kernel thread. It looks no problem for now.

Agenda

- Background
- Core Partitioning for User Processes
- Core Partitioning for Interrupts
- Core Partitioning for Kernel Threads
- **Executing a Realtime Application**
- Evaluating latency with cyclictest

Execute Application process by cgroup

- **All user process are running on Core 0**
  - Already default CPU affinity is changed
  - We hope that our application run on Core 1.

- **Cgroups enable executing our application on a specified Core**
  - We use cset command like kernel thread CPU affinity settings.

- **Example**
  - Do command such as following

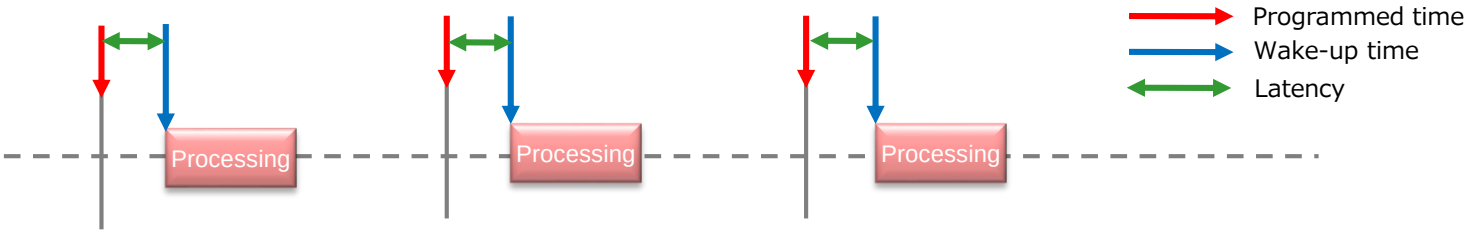
```
# cset set -s cpu1 -c 1
# cset proc -s cpu1 -e -- <command>
```

Agenda

- Background
- Core Partitioning for User Processes
- Core Partitioning for Interrupts
- Core Partitioning for Kernel Threads
- Executing a Realtime Application
- **Evaluating latency with cyclictest**

Evaluate latency by cyclictest

- **What is Cyclictest?**
  - Benchmark tool for interval timer latency.
  - Cyclictest thread is woken up periodically with a defined interval by an expiring timer. Calcurate difference between the programmed and the effective wake-up time. This time called “Latency”.
    - Refer: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest



- **Cyclictest argument**
  - Run cyclictest wit following arguments.

Interval	300us, 500us, 1000us
task priority	FIFO 98
sample number	1000000

  - Get latency histogram by following command.

```
# cyclictest -q -m -i399 -p98 -l1000000 -h1000
```

Evaluate latency by cyclictest

- **Environment**
  - Evaluate machine spec.

CPU	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz 4core
Memory	4GB
OS	Debian GNU/Linux 8.8.0(jessie)

- **Load program**
  - Run load program in background such as following.

stresss-no	no load program

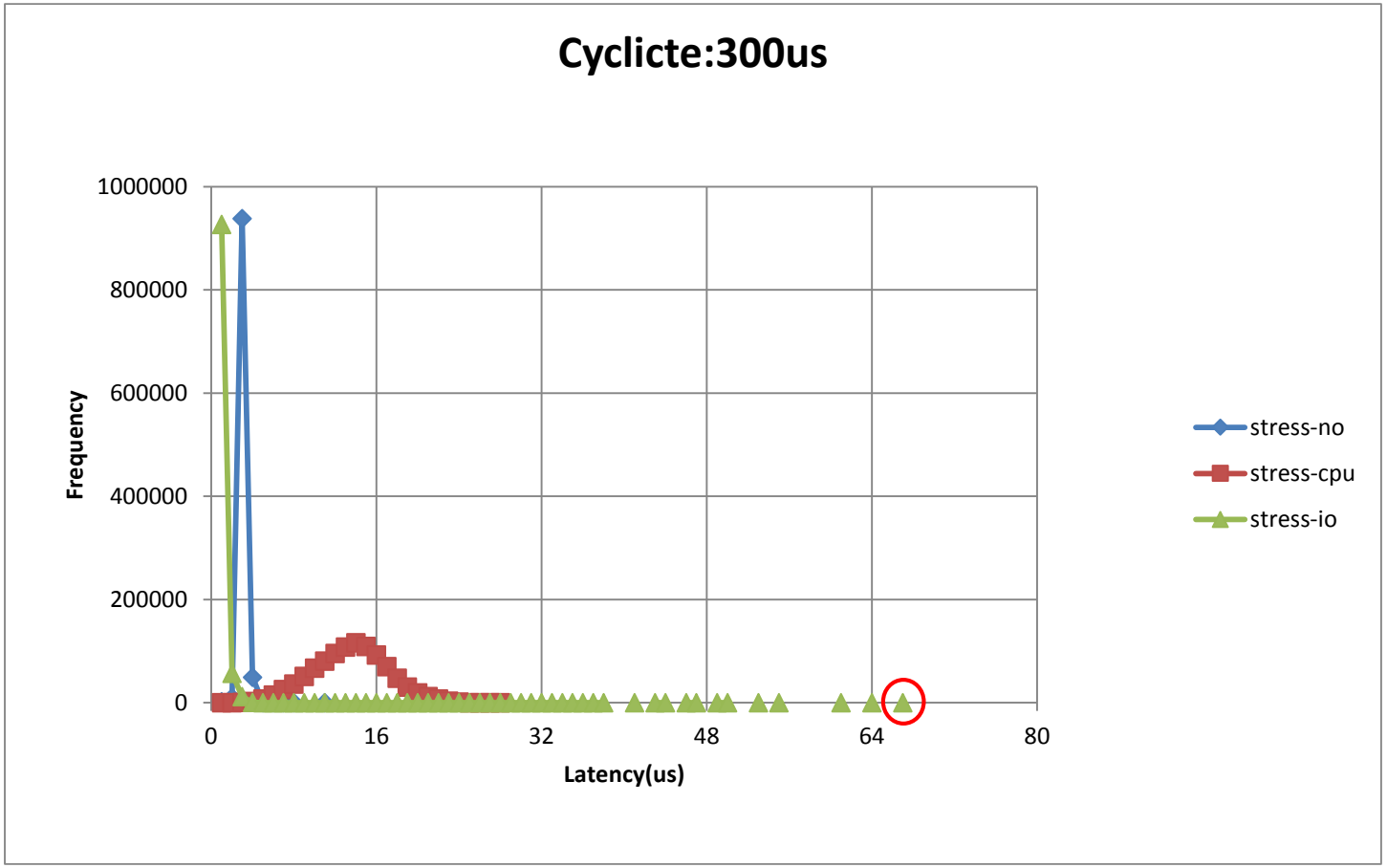
stress-cpu	150 thread execute such as following.
	while (1) {
	  usleep(1);
	}

stress-io	150 thread execute such as following.
	while (1) {
	  write(file) with O_SYNC;
	  uspeep(1);
	}

- **Core Partitioning**
  - OFF
    - All CPU affinity settings are default.
  - ON
    - CPU affinity of cyclictest is on Core 1.
    - CPU affinity of all other processes is on Core 0.

Core partitioning off

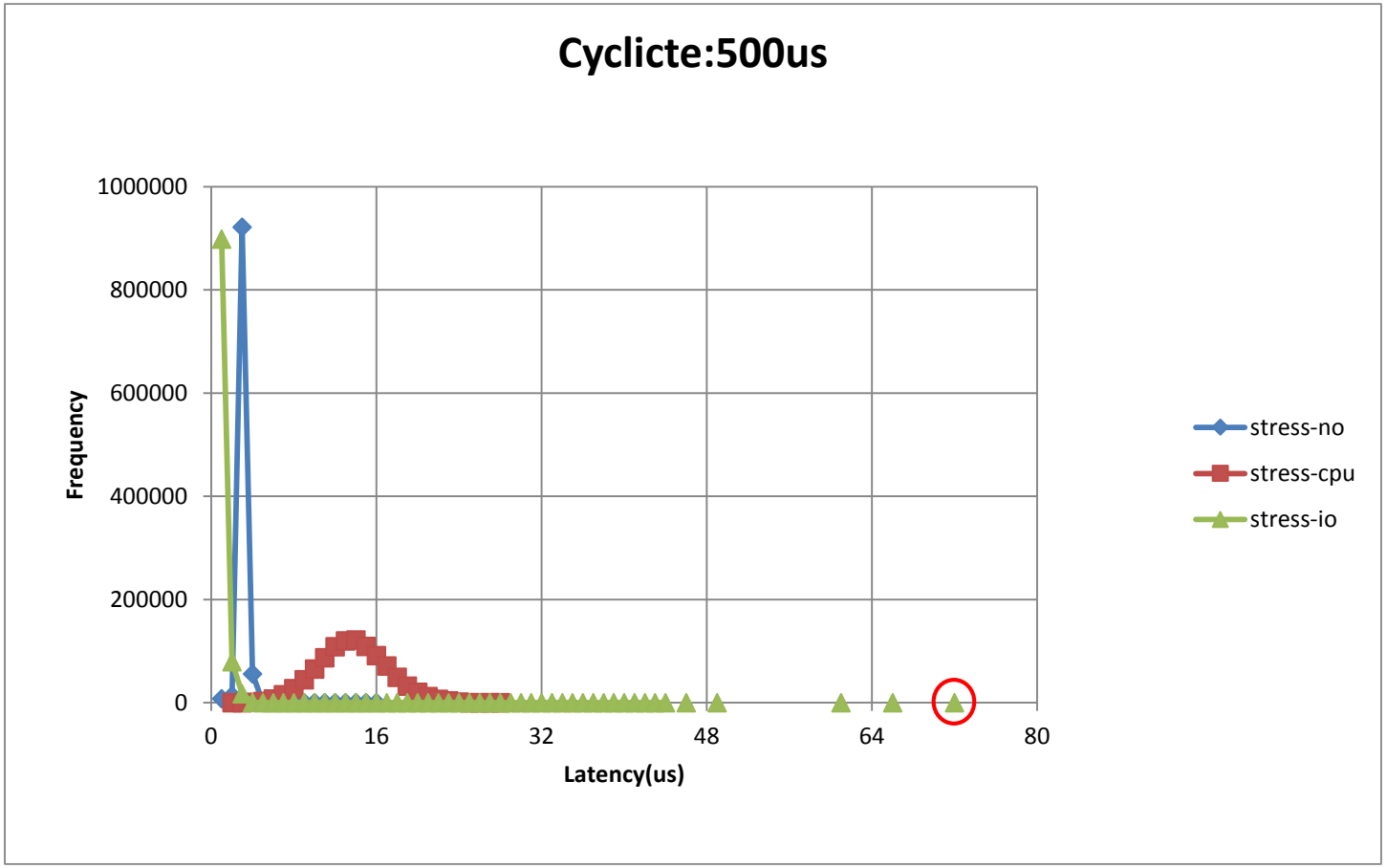
- 300us periodic



	stress-no	stress-cpu	stress-io
MAX	11	28	67
MIN	1	1	1
AVG	3	13	1

Core partitioning off

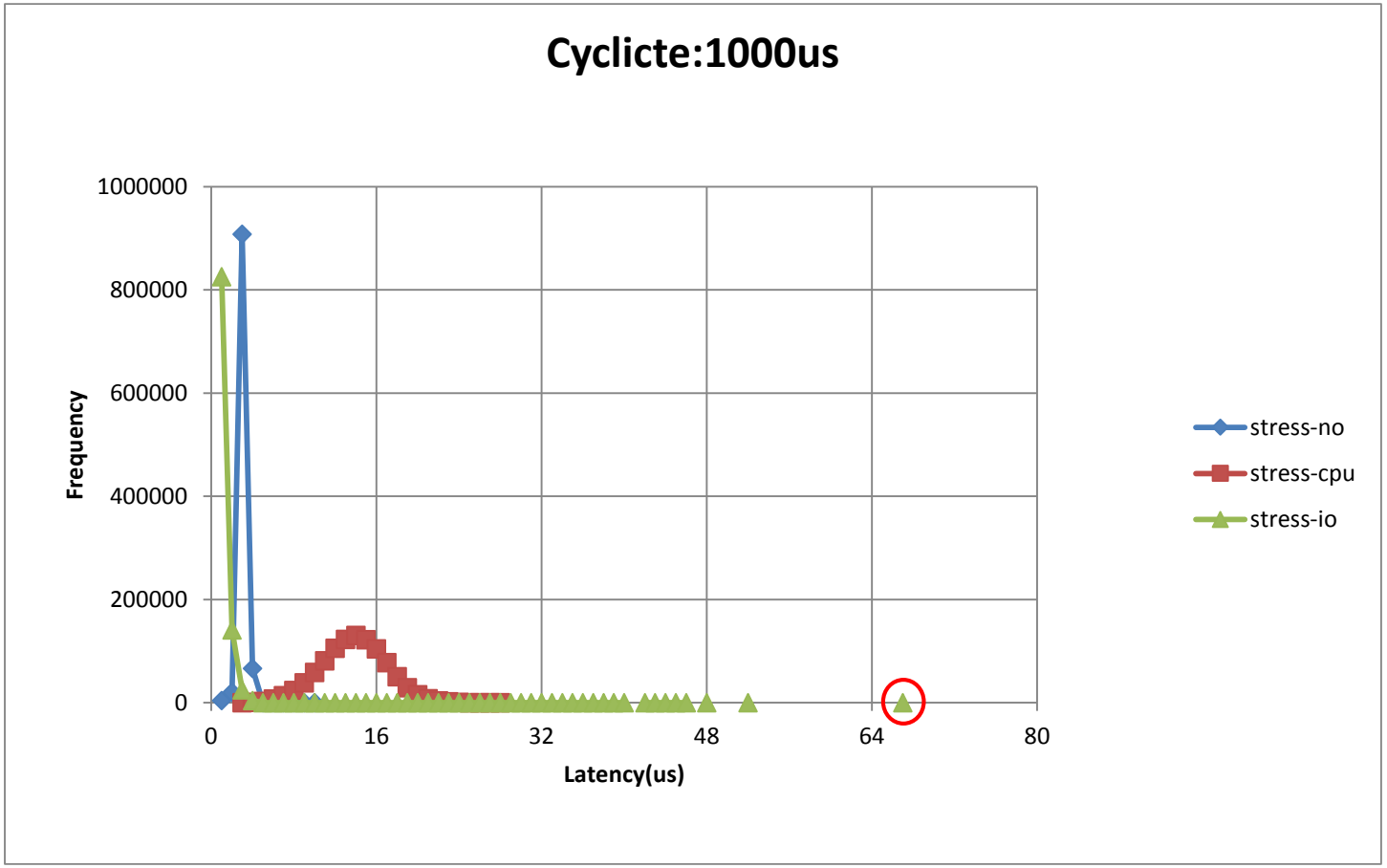
- 500us periodic



	stress-no	stress-cpu	stress-io
MAX	16	28	72
MIN	1	2	1
AVG	3	13	1

Core partitioning off

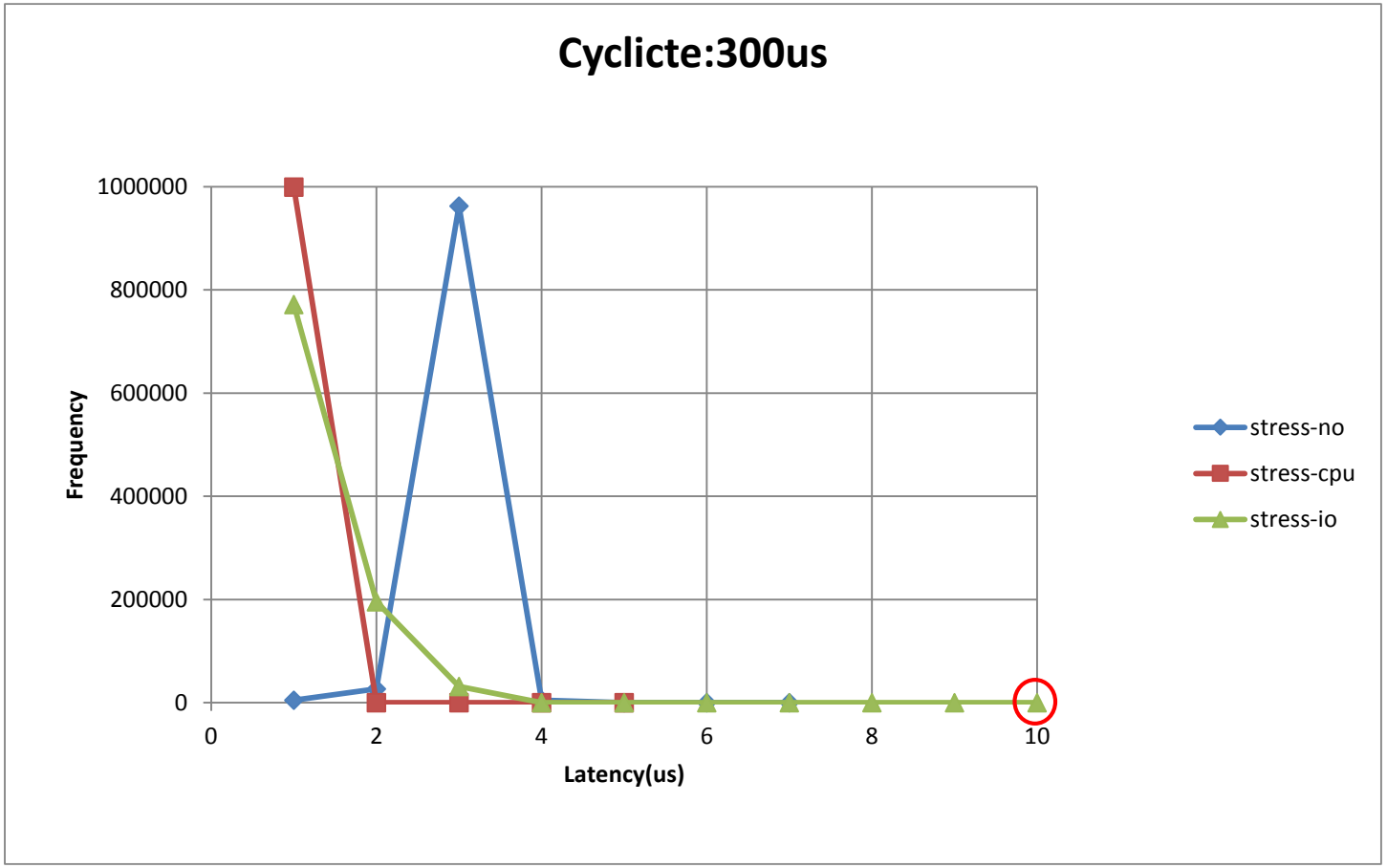
- **1000us periodic**



	stress-no	stress-cpu	stress-io
MAX	10	28	67
MIN	1	3	1
AVG	3	13	1

Core partitioning on

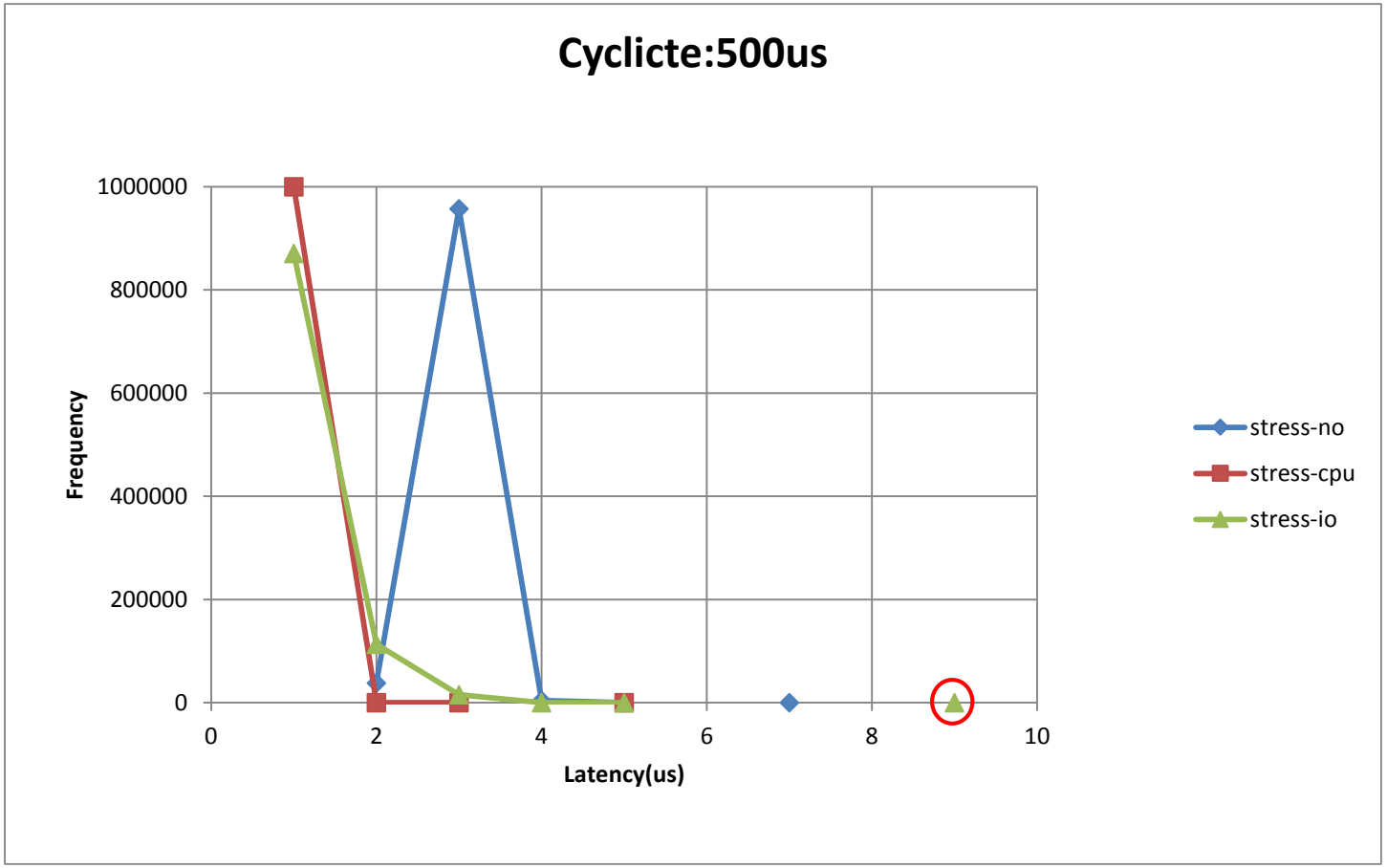
- 300us periodic



	stress-no	stress-cpu	stress-io
MAX	7	5	10
MIN	1	1	1
AVG	2	1	1

Core partitioning on

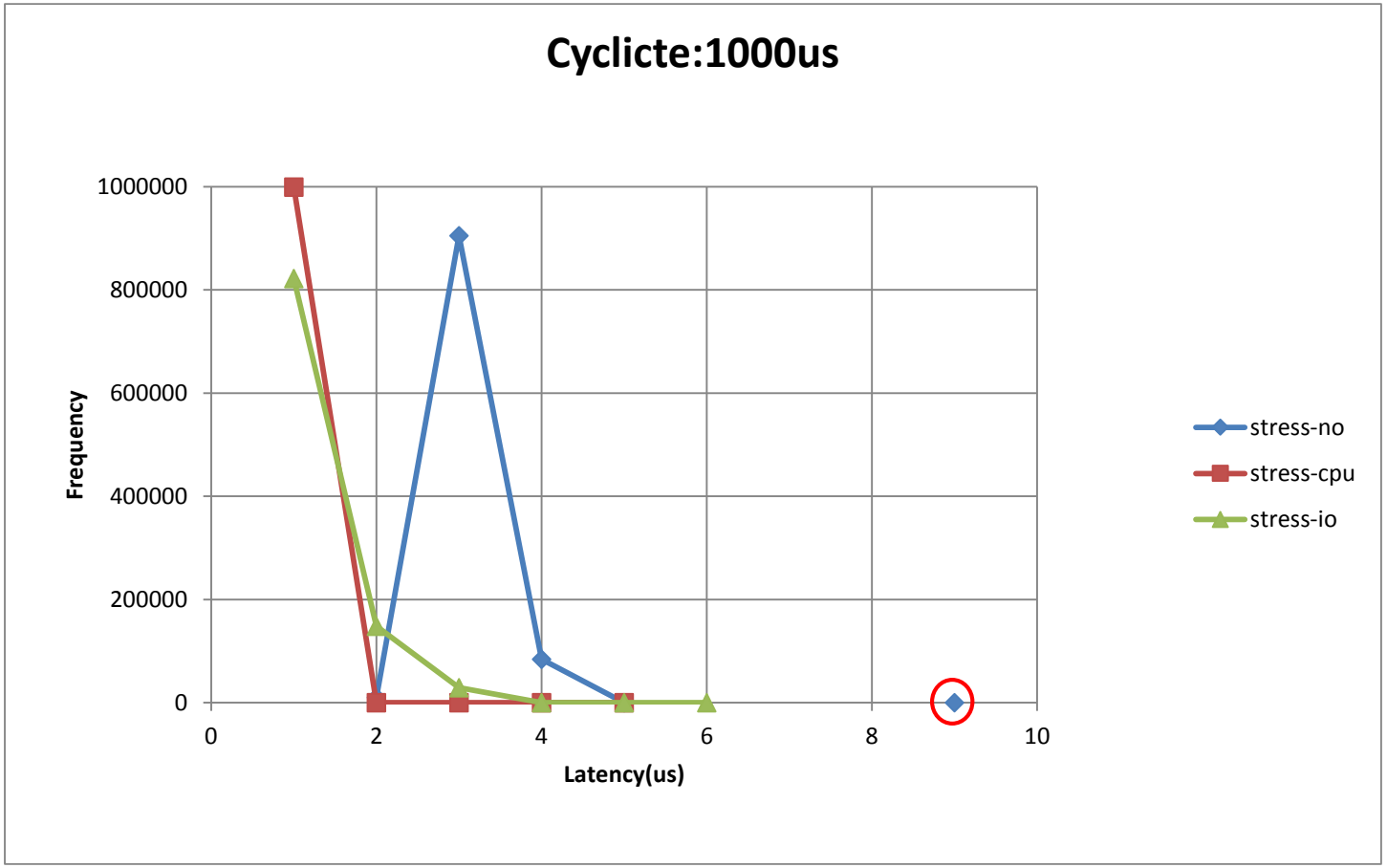
- 500us periodic



	stress-no	stress-cpu	stress-io
MAX	7	5	9
MIN	2	1	1
AVG	2	1	1

Core partitioning on

- 1000us periodic



	stress-no	stress-cpu	stress-io
MAX	9	5	6
MIN	2	1	1
AVG	3	1	1

Results

- **Core Partition off**
  - MAX 72us latency

- **Core Partition on**
  - MAX 10 us latency

We can keep low latency for realtime application by Core Partitioning!!


TOSHIBA
Leading Innovation