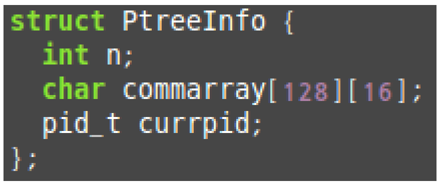
Kernel Driver Lab Report

2013-11422 Lee Eunha

Device driver allows the kernel to access hardware connected to the system. There are three categories in device driver-character device drivers, block device drivers, and network device drivers. In this lab, we only treated character device drivers.

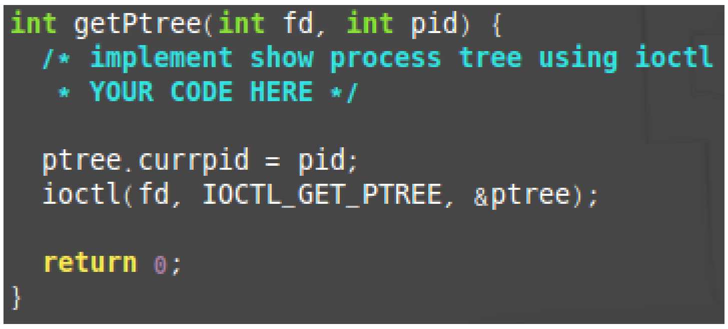
When we use device drivers, we should program them in kernel space. IOCTL is the function that send data from user space to kernel space. Printing and getting data was conducted in user space, and reading and writing in driver was conducted in kernel space.

Assignment 1 was to implement a kernel module that outputs the list of all parent processes up to the root when called through ioctl(). First I made PtreeInfo structure for process tree. It has current pid, processes' name array, and the number of child/parent processes. I thought the number of parents would be less than 128.



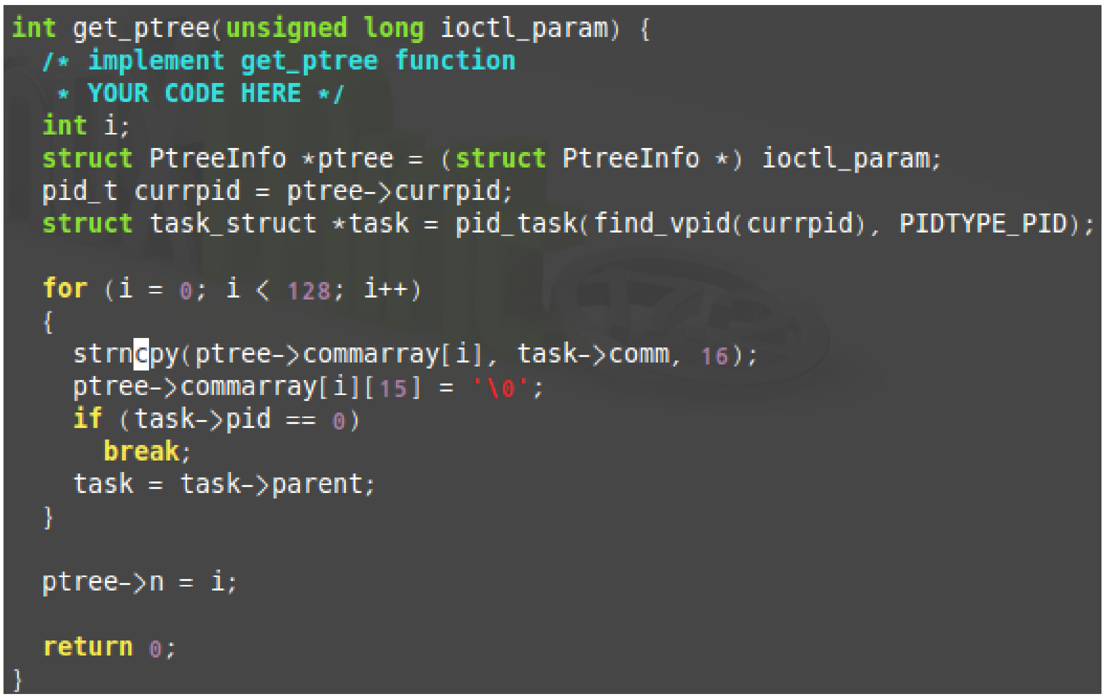
```
struct PtreeInfo {
  int n;
  char commarray[128][16];
  pid_t currpid;
};
```

Then I made function getPtree in profiler.c. It substitutes pid in the static variable PtreeInfo ptree. Next, it sends ptree to kernel level by ioctl.



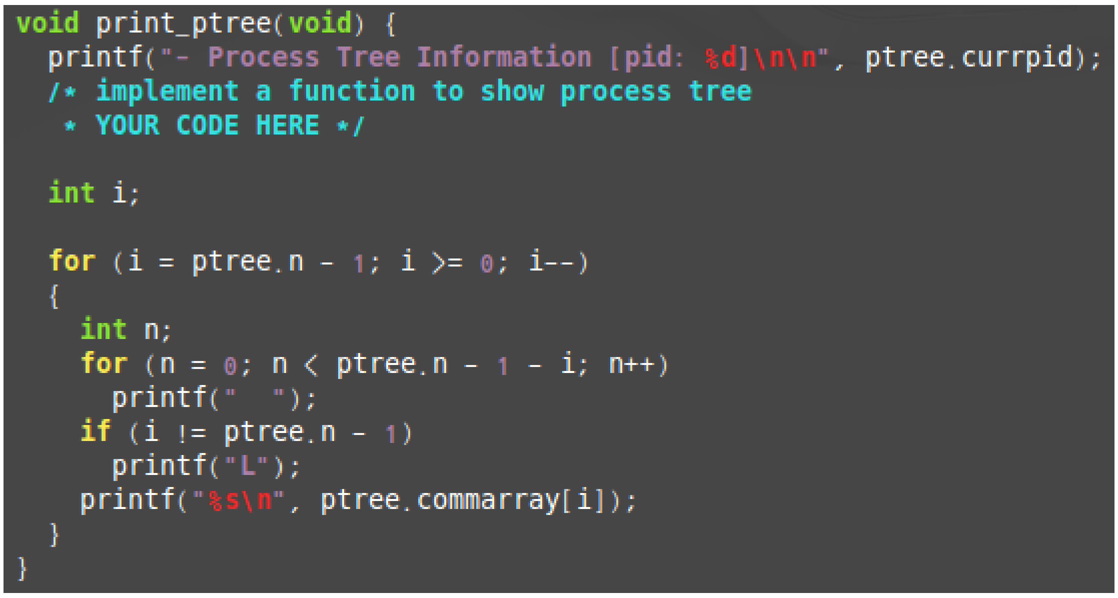
```
int getPtree(int fd, int pid) {
  /* implement show process tree using ioctl
   * YOUR CODE HERE */

  ptree.currpid = pid;
  ioctl(fd, IOCTL_GET_PTREE, &ptree);

  return 0;
}
```

The address of ptree revives in chardev.c. Using task_struct, we can take parent of current process, parent of parent process, and so on.



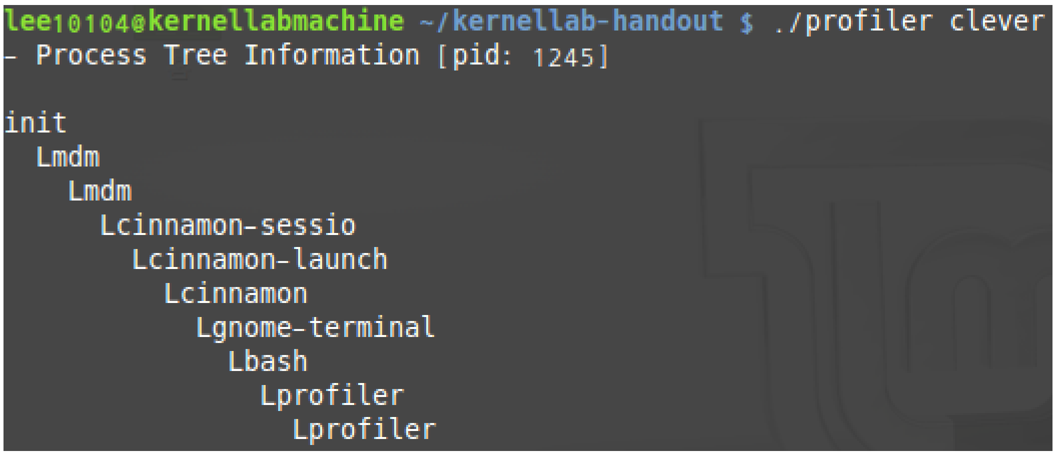
```c
int get_ptree(unsigned long ioctl_param) {
  /* implement get_ptree function
   * YOUR CODE HERE */
  int i;
  struct PtreeInfo *ptree = (struct PtreeInfo *) ioctl_param;
  pid_t currpid = ptree->currpid;
  struct task_struct *task = pid_task(find_vpid(currpid), PIDTYPE_PID);

  for (i = 0; i < 128; i++)
  {
    strncpy(ptree->commarray[i], task->comm, 16);
    ptree->commarray[i][15] = '\0';
    if (task->pid == 0)
      break;
    task = task->parent;
  }

  ptree->n = i;

  return 0;
}
```

Saved process array would be printed in profiler.c.



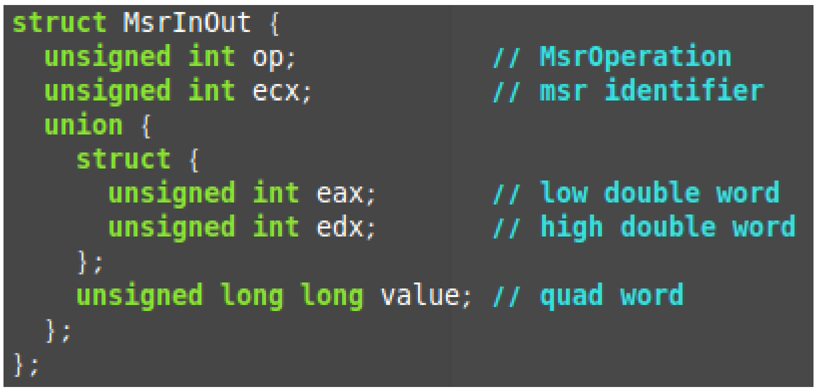
```c
void print_ptree(void) {
  printf("- Process Tree Information [pid: %d]\n\n", ptree.currpid);
  /* implement a function to show process tree
   * YOUR CODE HERE */

  int i;

  for (i = ptree.n - 1; i >= 0; i--)
  {
    int n;
    for (n = 0; n < ptree.n - 1 - i; n++)
      printf("  ");
    if (i != ptree.n - 1)
      printf("L");
    printf("%s\n", ptree.commarray[i]);
  }
}
```

The result shows like this.



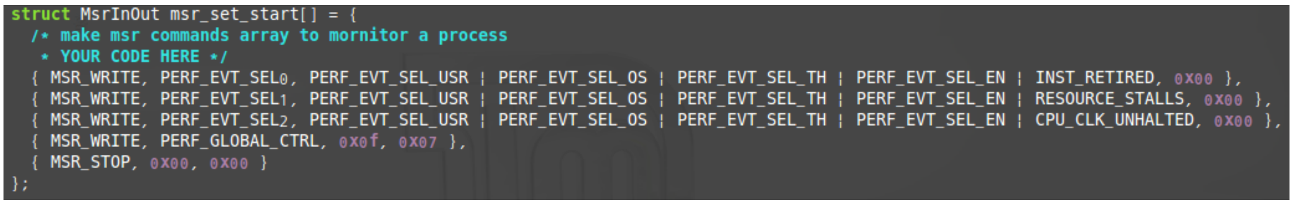
```
lee10104@kernellabmachine ~/kernellab-handout $ ./profiler clever
- Process Tree Information [pid: 1245]

init
  Lmdm
    Lmdm
      Lcinnamon-sessio
        Lcinnamon-launch
          Lcinnamon
            Lgnome-terminal
              Lbash
                Lprofiler
                  Lprofiler
```

Assignment 2 was to implement a kernel module to manage the processor's performance monitoring unit. We should use assembly to do this assignment, so we had structure MsrInOut what contains registers. MsrInOut also has operations what to do in kernel level.



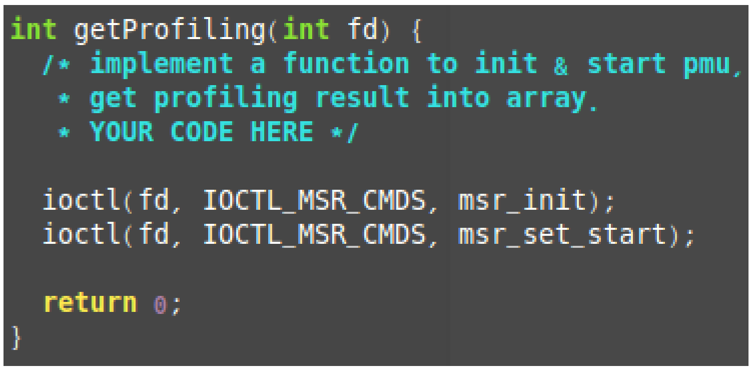
```
struct MsrInOut {
  unsigned int op;             // MsrOperation
  unsigned int ecx;            // msr identifier
  union {
    struct {
      unsigned int eax;        // low double word
      unsigned int edx;        // high double word
    };
    unsigned long long value;  // quad word
  };
};
```

In profiler.c, there are MsrInOut structure arrays that initialize, start writing, stop and read MSRs. I decided to put profiling data in PMC0, PMC1, PMC2.



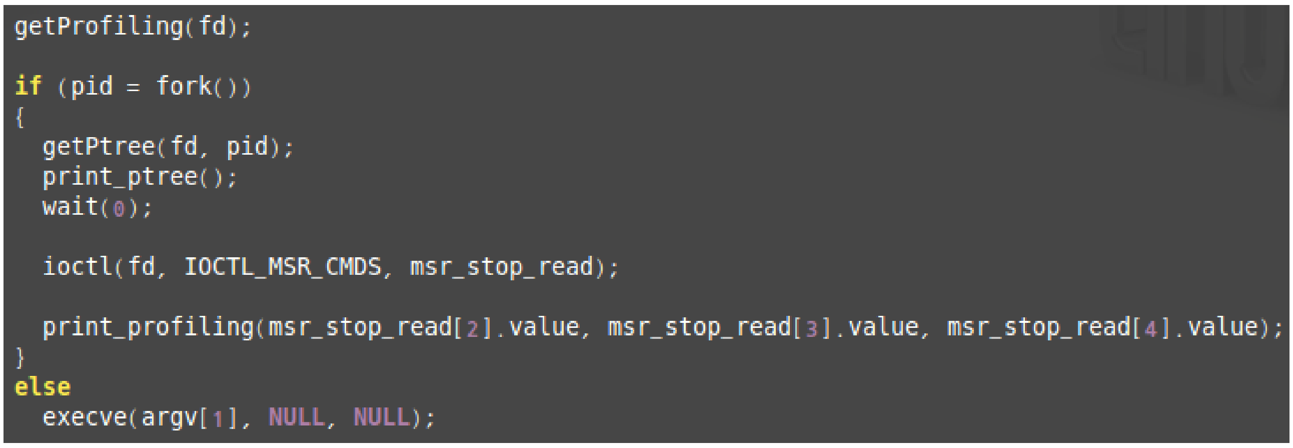
```
struct MsrInOut msr_set_start[] = {
  /* make msr commands array to mornitor a process
   * YOUR CODE HERE */
  { MSR_WRITE, PERF_EVT_SEL0, PERF_EVT_SEL_USR | PERF_EVT_SEL_OS | PERF_EVT_SEL_TH | PERF_EVT_SEL_EN | INST_RETIRED, 0x00 },
  { MSR_WRITE, PERF_EVT_SEL1, PERF_EVT_SEL_USR | PERF_EVT_SEL_OS | PERF_EVT_SEL_TH | PERF_EVT_SEL_EN | RESOURCE_STALLS, 0x00 },
  { MSR_WRITE, PERF_EVT_SEL2, PERF_EVT_SEL_USR | PERF_EVT_SEL_OS | PERF_EVT_SEL_TH | PERF_EVT_SEL_EN | CPU_CLK_UNHALTED, 0x00 },
  { MSR_WRITE, PERF_GLOBAL_CTRL, 0x0f, 0x07 },
  { MSR_STOP, 0x00, 0x00 }
};
```

Now we are ready to put data in registers. getProfiling starts initializing and gets ready for writing by taking data to kernel level.



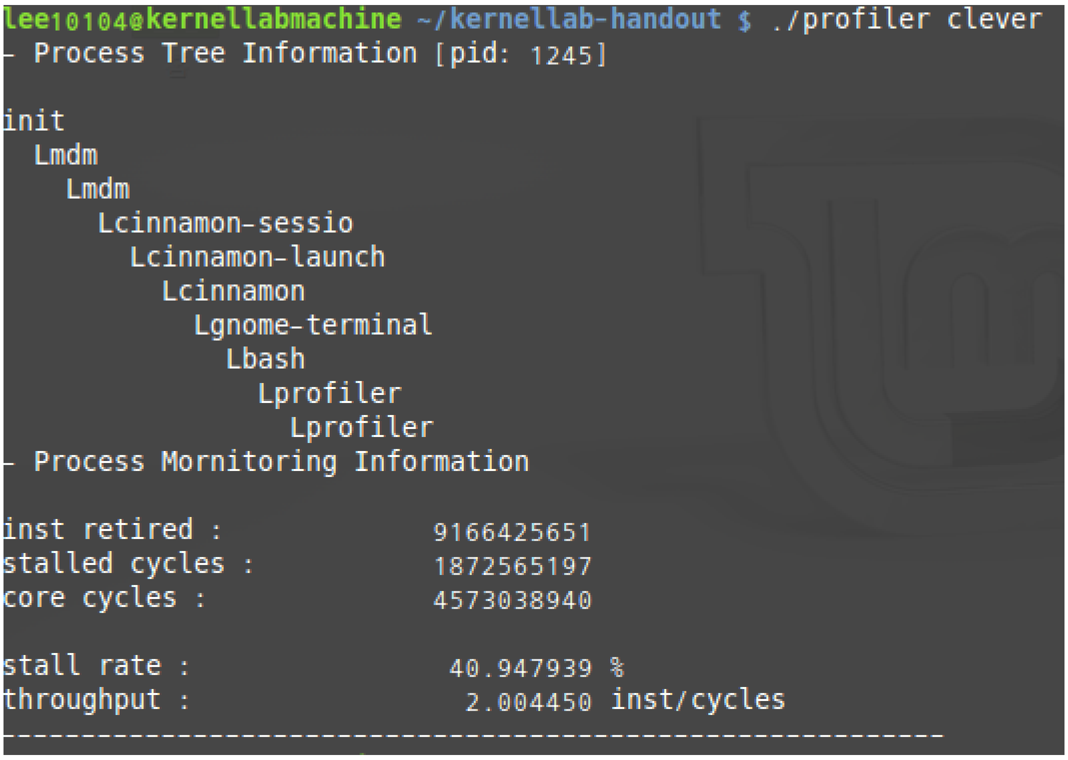
```
int getProfiling(int fd) {
  /* implement a function to init & start pmu,
   * get profiling result into array.
   * YOUR CODE HERE */

  ioctl(fd, IOCTL_MSR_CMDS, msr_init);
  ioctl(fd, IOCTL_MSR_CMDS, msr_set_start);

  return 0;
}
```

When the process forks, it will count the event of child process and save in MSRs. So we have to wait until child process terminates. Next we can read saved data.

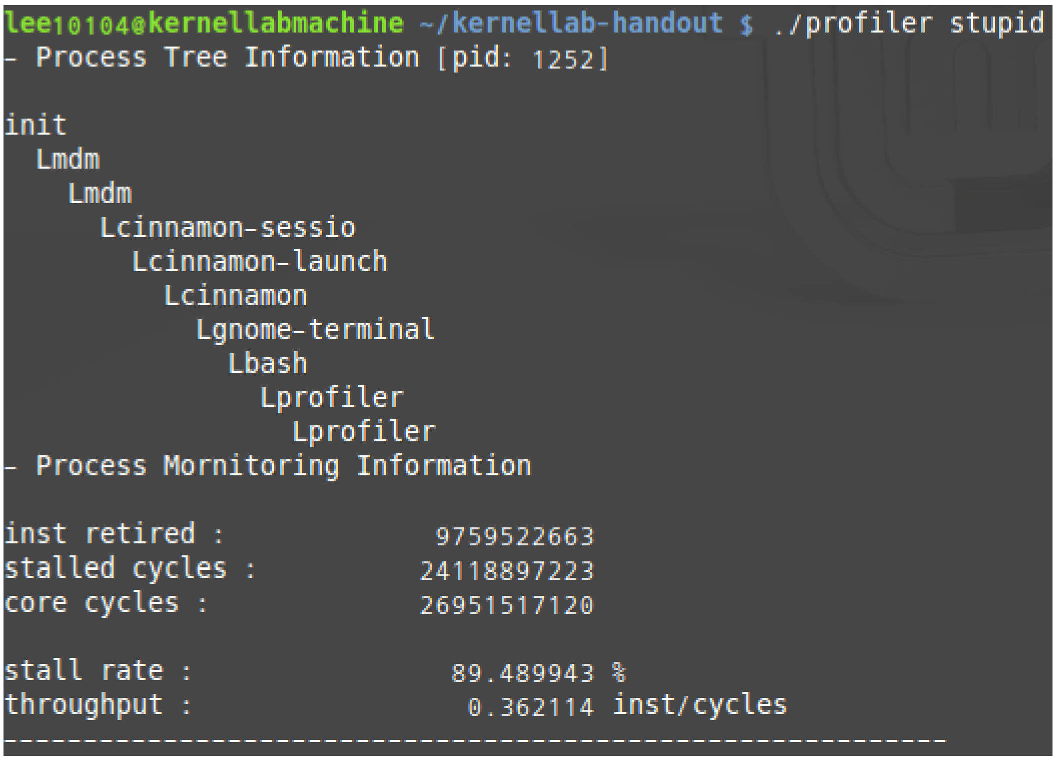


```
getProfiling(fd);

if (pid = fork())
{
  getPtree(fd, pid);
  print_ptree();
  wait(0);

  ioctl(fd, IOCTL_MSR_CMDS, msr_stop_read);

  print_profiling(msr_stop_read[2].value, msr_stop_read[3].value, msr_stop_read[4].value);
}
else
  execve(argv[1], NULL, NULL);
```

The result shows like this.



```
lee10104@kernellabmachine ~/kernellab-handout $ ./profiler clever
- Process Tree Information [pid: 1245]

init
  Lmdm
    Lmdm
      Lcinnamon-sessio
        Lcinnamon-launch
          Lcinnamon
            Lgnome-terminal
              Lbash
                Lprofiler
                  Lprofiler
- Process Mornitoring Information

inst retired :            9166425651
stalled cycles :          1872565197
core cycles :             4573038940

stall rate :               40.947939 %
throughput :                2.004450 inst/cycles
------------------------------------------------------------
```



```
lee10104@kernellabmachine ~/kernellab-handout $ ./profiler stupid
- Process Tree Information [pid: 1252]

init
  Lmdm
    Lmdm
      Lcinnamon-sessio
        Lcinnamon-launch
          Lcinnamon
            Lgnome-terminal
              Lbash
                Lprofiler
                  Lprofiler
- Process Mornitoring Information

inst retired :            9759522663
stalled cycles :         24118897223
core cycles :            26951517120

stall rate :               89.489943 %
throughput :                0.362114 inst/cycles
-----------------------------------------------------------
```

It was difficult to find what to do in kernel driver lab, especially assignment 2. I think if I didn't receive skeleton code and references, I would be failed to finish this lab. I could complete it by reading reference, but I still don't completely understand about kernel and MSR. I only wish next labs not to be difficult like this time.