CS 411 Operating Systems II Final Exam

Analysis of the Android x86 Build Tree

Li Li

1 Let's first talk about process scheduling.

In project 1, we implement sched.c and sched_rt.c just like what the official implementation in kernel.org.

1.1 linux kernel implementation

1.1.1 sched.c RT Policy

The below code was added to the function "rt_policy" in order to determine if a task was realtime (RT) or not.

```
if (unlikely(policy == SCHED_FIFO || policy == SCHED_RR))
        return 1;
```

1.1.2 sched.c Stop task

The below code was added to the function "sched_set_stop_task" in order to set a "stop task" which is a FIFO task at the highest priority, effectively stopping the CPU.

```
sched_setscheduler_nocheck(stop, SCHED_FIFO, &param);
```

1.1.3 sched.c Forking reset

The below code was added to the function "sched_fork" in order to reset a process's scheduling policy to normal after forking.

```
if (p->policy == SCHED_FIFO || p->policy == SCHED_RR) {
        p->policy = SCHED_NORMAL;
        p->normal_prio = p->static_prio;
}
```

1.1.4 sched.c Accept RT Policies

The below code was added to the function "__sched_setscheduler" in order for the setscheduler function to accept RT policy types.

```
if (policy != SCHED_FIFO && policy != SCHED_RR &&
        policy != SCHED_NORMAL && policy != SCHED_BATCH &&
```

1.1.5 sched.c Minimum and Maxiumn RT Priority

The below code was added in order to return the minimum or maximum RT policy for a task.

Maximum (sys_sched_get_priority_max):

```
case SCHED_FIFO:
case SCHED_RR:
        ret = MAX_USER_RT_PRIO-1;
        break;
```

Minimum (sys_sched_get_priority_min):

```
case SCHED_FIFO:
case SCHED_RR:
        ret = 1;
        break;
```

1.1.6 sched_rt.c Timeslice management

The below code was added to the function "task_tick_rt" in order to handle the action of managing timeslices by decrementing the current process's timeslice each tick. If the timeslice reaches 0, the process's time on the CPU has expired. So, the timeslice is reset, and then the current process is put to the end of the queue.

```
if (p->policy != SCHED_RR)
        return;

if (--p->rt.time_slice)
        return;

p->rt.time_slice = DEF_TIMESLICE;
```

1.1.7 sched_rt.c Get timeslice

The below code was added to the function "get_rr_interval_rt" in order to get the (default) timeslice for a RT process.

```
if (task->policy == SCHED_RR)
        return DEF_TIMESLICE;
```

1.2 Android's approach

1.2.1 nearly the same

Android's code is base on linux kernel, there are tiny difference. Below is diff in kernel/sched_rt.c

```
1041c1041
<             curr->prio < p->prio) &&
---
>             curr->prio <= p->prio) &&
1392a1393,1397
```

```
> #ifdef __ARCH_WANT_INTERRUPTS_ON_CTXSW
>        if (unlikely(task_running(rq, next_task)))
>                return 0;
> #endif
>
1572c1577
<            rq->curr->prio < p->prio))
---
>            rq->curr->prio <= p->prio))
1776c1781
<
---
>
```

There are little more diffin kernel/sched.c but most of them is just a update of function name in a newly kernel. However, there are these two function that is added. One is to show cpu frequency, another is show total power.

```
> static int cpuacct_cpufreq_show(struct cgroup *cgrp, struct cftype *cft,
>               struct cgroup_map_cb *cb)
> {
>       struct cpuacct *ca = cgroup_ca(cgrp);
>       if (ca->cpufreq_fn && ca->cpufreq_fn->cpufreq_show)
>               ca->cpufreq_fn->cpufreq_show(ca->cpuacct_data, cb);
>
>       return 0;
> }
>
> /* return total cpu power usage (milliWatt second) of a group */
> static u64 cpuacct_powerusage_read(struct cgroup *cgrp, struct cftype *cft)
> {
>       int i;
>       struct cpuacct *ca = cgroup_ca(cgrp);
>       u64 totalpower = 0;
>
>       if (ca->cpufreq_fn && ca->cpufreq_fn->power_usage)
>               for_each_present_cpu(i) {
>                       totalpower += ca->cpufreq_fn->power_usage(
>                                       ca->cpuacct_data);
>               }
>
>       return totalpower;
> }
```

1.3 Why android choose these approch?

Android is a lightweight smartphone os, there are not so many threads coming and need to determing the optimal solution by a very sophisticated method. Also the common threads in android are normal audio, display ,user-foreground thread. Not very complex, setting a default priority for them would be a quick and efficent solution. Round-robin and FIFO method should also work in Android environment.

2 Then, I think memory management in both OS are worth to talk about.

2.1 Well, this part is our best-fit slob.c implementation in linux

2.1.1 struct best_slob

We first implemented a struct called best_slob which helps keep track of the best fit slob/page. The struct page holds the best fit page. The pointers prev and next hold the previous and next blocks, respectively. The pointer slob holds the best fit block. The size variable holds the size of the block.

2.1.2 best_fit()

The best_fit function searches for the best fit block in the list head. If the best fit block is found then the function returns a 1, otherwise it returns 0.

2.1.3 slob_page_alloc()

We have added code to the existing slob_page_alloc() as well as adding two extra parameters to add the best_slob block and the previous slob block to the function. If the block with the best fit is found then the current block pointer is set to point to that block. The existing code will continue to search for the first fit block starting with the current pointer to see if the block fits the size that is needed. Since the current pointer is pointing to the best fit block the code returns that block.

2.1.4 slob_alloc()

We added some code to this function for finding the best fit block. First we initialize our struct, the best_slob and the prev_slob. Then we run the best_fit() function to see if there is a block that best fits what is needed. If the best_page is not found then we will have to create a new page. If best_page exists then it is set to a variable and the function's normal operation continues.

2.2 Then let's see what android does.

To be frank, android did not change anything of linux's standard of memory management for this praticular slob.c. However, it adds ashmem.c and pmem.c.

Ashmem - Android shared memory is implemented in mm/ashmem.c. According to the Kconfig help "The ashmem subsystem is a new shared memory allocator, similar to POSIX SHM but with different behavior and sporting a simpler file-based API." Apparently it better-supports low

memory devices, because it can discard shared memory units under memory pressure. To use this, programs open /dev/ashmem, use mmap() on it, and can perform one or more of the following ioctls:

- **ASHMEM_SET_NAME**
- **ASHMEM_GET_NAME**
- **ASHMEM_SET_SIZE**
- **ASHMEM_SET_PROT_MASK**
- **ASHMEM_GET_PROT_MASK**
- **ASHMEM_PIN**
- **ASHMEM_UNPIN**
- **ASHMEM_GET_PIN_STATUS**
- **ASHMEM_PURGE_ALL_CACHES**

From a thread on android-platform source, you can create a shared memory segment using:

```
fd = ashmem_create_region("my_shm_region", size);
  if(fd < 0)
      return -1;
  data = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  if(data == MAP_FAILED)
      goto out;
```

In the second process, instead of opening the region using the same name, for security reasons the file descriptor is passed to the other process via binder IPC.
The libcutils interface for ashmem consists of the following calls: (found in system/core/include/-cutils/ashmem.h)

- int ashmem_create_region(const char *name, size_t size);
- int ashmem_set_prot_region(int fd, int prot);
- int ashmem_pin_region(int fd, size_t offset, size_t len);
- int ashmem_unpin_region(int fd, size_t offset, size_t len);
- int ashmem_get_size_region(int fd);

PMEM - Process memory allocator, implementation at: drivers/misc/pmem.c with include file at: include/linux/android_pmem.h. The pmem driver is used to manage large (1-16+MB) physically contiguous regions of memory shared between userspace and kernel drivers (dsp, gpu, etc). It was written specifically to deal with hardware limitations of the MSM7201A, but could be used for other chipsets as well. For now, you're safe to turn it off on x86.

Ashmem and pmem are very similar. Both are used for sharing memory between processes.

ashmem uses virtual memory, whereas pmem uses physically contiguous memory. One big difference is that with ashmem, you have a ref-counted object that can be shared equally between processes. For example, if two processes are sharing an ashmem memory buffer, the buffer reference goes away when both process have removed all their references by closing all their file descriptors. pmem doesn't work that way because it needs to maintain a physical to virtual mapping. This requires the process that allocates a pmem heap to hold the file descriptor until all the other references are closed.

You have the right idea for using shared memory. The choice between ashmem and pmem depends on whether you need physically contiguous buffers. In the case of the G1, we use the hardware 2D engine to do scaling, rotation, and color conversion, so we use pmem heaps. The emulator doesn't have a pmem driver and doesn't really need one, so we use ashmem in the emulator. If you use ashmem on the G1, you lose the hardware 2D engine capability, so SurfaceFlinger falls back to its software renderer which does not do color conversion, which is why you see the monochrome image.

2.3 Why android does it?

First, to follow the standardized memory management in linux kernel. It do not need any changes, since it works fine. Also the featured idea to add an ashmem memory allocation method is a plus. Sharing anonymous memory may help caching and similar resource management that efficiently integrates with kernel memory management. The use of pmem may for the reason that kernel drivers (dsp, gpu, etc) is so widely used, it need a contiguous regions of memory to boost the performance.

3 Also, I want to share some findings in I/O scheduler of both OSes

3.1 Here is basicly what I did in project2. SSTF for the linux kernel

In order to implement a SSTF IO scheduler, we first had to look at noop-iosched.c and elevator.c. Since the assignment is based off the implementation of noop scheduler we can reuse many of the functions within noop-iosched.c. In order for SSTF to function properly direction of head travel must be known as well as the sector location of all IO operations that are to be dispatched. We will start in one direction with a doubly linked circular list sorted in ascending order, traversing until we reach the sentinel(end of head travel), then reverse direction and until the sentinel is reached again. This operation is similar to the elevator scheduler.

Then there is this *dispatch*. Dispatch checks the direction of the sstf data struct to determine if the sectors are increasing or decreasing to determine in which direction a request should be sought. Once a request in the direction with a sector higher/lower that te last request is found, a printk statement indicates which request is being dispatched. When no new requests are in the current direction of travel, the direction is reversed and requests are dispatched in the opposite order. In addition to the previous printk statement, the sector difference between the request being dispatched sector and the previous sector is displayed to ensure the correct the traversal is in the correct direction.

After that is *Add Request*Add request inserts requests into a list sorted ascending order. A printk statement was added to give the sector number of the request being added.

3.2 Then let's see some android changes.

The android kernel provided have nothing changed from the original noop-iosched.c. It is a FIFO like mechanism.

So I want talk about a 3.0.x android kernel which has add this sio-iosched.c that I did not see in linux kernel. It is recommend as android default I/O scheduler in many online communities. Simple IO scheduler is Based on Noop, Deadline and V(R) IO schedulers. Let's see its implementation. Here is the merge function,basicly what it does is if next request expires before current request, assign its expire time to current request and move into next position in fifo list. And delete next request.:

```
static void
sio_merged_requests(struct request_queue *q, struct request *rq,
                    struct request *next)
{

        if (!list_empty(&rq->queuelist) && !list_empty(&next->queuelist)) {
                if (time_before(rq_fifo_time(next), rq_fifo_time(rq))) {
                        list_move(&rq->queuelist, &next->queuelist);
                        rq_set_fifo_time(rq, rq_fifo_time(next));
                }
        }

        rq_fifo_clear(next);
}
```

sio_add_request add request to the proper fifo list and set its expire time. *sio_queue_empty* check if fifo lists are empty. *sio_expired_request* retrieve request and check if it has expired and return it or else return null. *sio_choose_expired_request* check expired requests. Asynchronous requests have priority over synchronous. Write requests have priority over read. *sio_choose_request* Retrieve request from available fifo list. Synchronous requests have priority over asynchronous. Read requests have priority over write. In *sio_init_queue* It allocates structure, initialize fifo lists and initializes data. At last, *sio_exit_queue* will free all structures.

3.3 Why this sio I/O scheduler?

SIO is based on the deadline scheduler but it's more like a mix between no-op and deadline. SIO is like a lighter version of deadline but it doesn't do any kind of sorting, so it's aimed mainly for random-access devices (like SSD hard disks) where request sorting is no needed (as any sector can be accesed in a constant time, regardless of its physical location). For Android device, it random access devices would be much popular, so they made this scheduler. And it seems works quite good.

4 Addendum: Group evaluation

There are four members in our group. I think all of us did a good job in these particular class.

First, Josh Little he did most of the implementaions. Designing and direction were all most hold by him. Furthermore, he is just in his junior year. So I personal think he did a great job. As a group member, he is very helpful and he taught me a lot on materials I do not understood.

Also, Nicholas Kelly did half of the test code and some project implementations. He is also very hard working on almost all projects, expect for project 4 because of senior design. He is also very friendly to works with.

Then ,there is Chavis Sabin, he made all most all our writeups. They are great writeups which give us a good group score. But I think our group demo is also great.

For me, Li Li. Because of some unfarmiliar on language, I seems to talk not very much in the group. But I did individully about half of the test codes project1, project2, project3. Because Josh and Nick mostly did the coding. So most of the time watch their codes and try to give my attempt solution to them. So some part is original come up with me but implement by Josh. Among these five projects, I remember the most of the time was spent on project4. The best-fit slob.c, I worked with Josh Little the whole weekend and Monday which is also Memorial day. It cost me about 30 hours to debug. That time, we just try to build a whole new slob.c, new structs ,new functions. But we encountered kernel panic and a big problem with syscall from user space. The output is unusual large number or negtive number. It seem ed to be overflowed and underflowed. But we did not figure out. Until we swept all most everything and started from the beginning.

All in all, it is a hardworking class. We spent much much more time in project meetings than the actually lectural classes.