Operating Systems and Multiprogramming Exam

Nikolaj Dybdahl Rathcke (rfq695)

March 28, 2014

Contents

1 Practical Part

I use fyams.harddisk and my volume name is arkimedes. Tests used are `getclock` and `deadline`.

1.1 Obtaining the system time

To implement the system call `getclock`, it was sufficient to work in file `proc/syscall.c`. The wrapper to the syscall, `syscall_getclock()` is seen here.

```
int syscall_getclock()
{
    return rtc_get_msec();
}
```

Calling a function from `drivers/metadev.h` which returns the number of milliseconds since startup. We then add the syscall to the switch statement

```
void syscall_handle(context_t *user_context) {

  (..)

  case SYSCALL_GETCLOCK:
      user_context->cpu_regs[MIPS_REGISTER_V0] =
          syscall_getclock();
      break;

  (..)
}
```

The function requires no argument and simply puts the result in register V_0.

A simple test is made to see if this works is seen below

```
#include "tests/lib.h"

int main(void)
{
    int a = syscall_getclock();
```

```
    printf("Running time is %d ms\n", a);
    int b = syscall_getclock();
    while (b-a < 1000) {
      b = syscall_getclock();
    }
    printf("Running time is now %d ms\n", b);
    syscall_halt();

    return 0;
}
```

Which simply prints the running time, a, since startup, then checks running time minus a until this is 1000 ms later (waiting one second). It then prints the running time since startup again.
This printed, for me, 940 and then 1940, which is exactly what was expected since it should be 1000 ms later than a meaning the implementation was successful.

1.2 Deadline scheduling in Buenos

1.2.1 Kernel Implementation

The implementation is done in `kernel/thread.[ch]` and `kernel/scheduler.c`. Changes made to `thread.h` is to the data structure `thread_table_t` where following fields were added/modified.

```
typedef struct {
  (..)

  /* A pointer to the previous thread in list */
  /* Used to make scheduler_ready_to_run doubly-linked */
  TID_t previous;

  /* pad to 64 bytes */
  uint32_t dummy_alignment_fill[7];

  /* deadline */
  int deadline;
```

```
} thread_table_t;
```

An int `deadline` in order to use deadline scheduling.
A TID_t `previous` is added, since I discovered, during the implementation in `scheduler.c` that, in order to move an element from a singly linked list to the head, it was easier to keep track of the element before, so it could be linked with the element afterwards.
The amount that is filled is reduced by 2, because 2 new field has been introduced.

In `thread.c` the new field `deadline` is initialized to 0 (which is the lowest priority) and `previous` to -1 in the thread table.

```
void thread_table_init(void) {

  (..)

  thread_table[tid].previous   = -1;
  thread_table[i].deadline     = 0;

  (..)
}
```

And when creating a thread, `previous` is set to -1 and `deadline`, if given, is set to the given deadline plus the the running time since startup to give a determining value for the deadline. That is, so no thread can 'overtake' other threads just because it has a lower deadline, e.g. a thread with a deadline on 500 will run before a thread created 400 ms later but with a deadline on 200.
If no deadline is given it is initialized to 0.

```
TID_t thread_create(void (*func)(uint32_t), uint32_t arg, int deadline) {

  (..)

  thread_table[tid].previous     = -1;
  if (deadline > 0) {
    thread_table[tid].deadline   = deadline + rtc_get_msec();
  }
```

```
  else {
    thread_table[tid].deadline   = 0;
  }

  (..)
}
```

The implementation of deadline schedule is made in the function `scheduler_schedule` in the section before `scheduler_remove_first_ready` is called.
`scheduler_remove_first_ready` is the function that removes the head of the singly linked list `scheduler_ready_to_run` and then runs this thread.
A `TID_t`, `t` and `lowestDL`, is used in the implementation.

```
void scheduler_schedule(void)
{
  TID_t t;
  TID_t lowestDL = -1;

  (..)

  t = scheduler_ready_to_run.head;

  while (lowestDL == -1 && t != -1) {
    if (thread_table[t].deadline > 0) {
      lowestDL = t;
      break;
    }
    t = thread_table[t].next;
  }
```

`t` is set to head. Because of the way the implementation works, `lowestDL` needs to be set to a `TID_t` whose deadline is larger than 0 if possible. This is seen later why this is necessary. The while-loop runs through the entire linked list and if such a deadline exists, `lowestDL` is set to the `TID_t` with a deadline and the while loop is broken out of.

```
  t = scheduler_ready_to_run.head;

  if (lowestDL != -1) {
```

```
while (t != -1) {
  if (thread_table[t].deadline < thread_table[lowestDL].deadline &&
      thread_table[t].deadline > 0) {
    lowestDL = t;
  }
  t = thread_table[t].next;
}
```

Now, `t` is reset to be the head of the list, so we can traverse through the entire list again.
A check to see if a `TID_t` with a deadline larger than 0 is made. If it does not exist, it skips the list modification and simply runs the head, since all deadlines must be 0 (low priority).
If it does exist, the entire list is traversed to find the thread with the lowest deadline. This is why, `lowestDL` had to be initialized to a `TID_t` with a deadline larger than 0, since otherwise this deadline could not be 'beaten' ad deadlines have values larger than 0.

```
if (lowestDL == scheduler_ready_to_run.tail &&
      lowestDL != scheduler_ready_to_run.head) {
  scheduler_ready_to_run.tail = thread_table[lowestDL].previous;
  thread_table[scheduler_ready_to_run.tail].next = -1;
  thread_table[lowestDL].next = scheduler_ready_to_run.head;
  thread_table[scheduler_ready_to_run.head].previous = lowestDL;
  thread_table[lowestDL].previous = -1;
  scheduler_ready_to_run.head = lowestDL;
}
```

Now when the thread with the lowest deadline is found, it needs to be inserted in the head. We have to check in the case that it lies in the tail first. If it does *and* it is not the head, since it would be a list consisting of one element and we do not want to link it with itself, we make the proper changes to the list. This is, setting a new tail, the previous element, with a new `next` to -1 (end of list) and inserts it in the head by linking it together with the old head and setting its `previous` to -1.

```
else if (lowestDL != scheduler_ready_to_run.head) {
  thread_table[thread_table[lowestDL].previous].next =
```

```
                                        thread_table[lowestDL].next;
      thread_table[thread_table[lowestDL].next].previous =
                                        thread_table[lowestDL].previous;
      thread_table[lowestDL].next = scheduler_ready_to_run.head;
      thread_table[scheduler_ready_to_run.head].previous = lowestDL;
      thread_table[lowestDL].previous = -1;
      scheduler_ready_to_run.head = lowestDL;
    }
  }
```

In the case where it is not the tail or the head, it removes this element by linking its next element and previous element together and the inserts it in the head in the same manner as above.
This ensures that we now have the thread with the lowest deadline or one with no deadline if no thread with a deadline exists in the head of the list and we can run this thread.

```
  t = scheduler_remove_first_ready();
  thread_table[t].state = THREAD_RUNNING;

  (..)
}
```

Note: A spinlock is acquired before making these changes since we are accessing the thread table.

1.2.2 Userland API with Deadlines

This sort of merges with task (a). However it is simply giving the exec call a new argument, an `int deadline`.
To follow this, the function `process_spawn` requires the deadline. When this function creates a thread, it gives it deadline with it, and as such the thread inherits the deadline given when `syscall_exec` is called.
All other references for `process_spawn` are also given a deadline with it. In `syscall.c`, the `SYSCALL_EXEC` is modified to take an extra argument

```
  case SYSCALL_EXEC:
      user_context->cpu_regs[MIPS_REGISTER_V0] =
         syscall_exec((char *)A1, A2);
```

Since the deadline is an `int` it is not necessary to cast it to anything.

1.2.3 Testing and Discussion

To test the implementation, a testfile `tests/deadline.c` was made. It makes a call for 7 other testfiles called deadlinetest[1-7] (resides in `tests` folder). These tests **only** prints their deadline. These child processes are then joined and the `syscall_halt()` is called before terminating.

```
#include "tests/lib.h"

static const char prog1[] = "[arkimedes]dtest1";
static const char prog2[] = "[arkimedes]dtest2";
static const char prog3[] = "[arkimedes]dtest3";
static const char prog4[] = "[arkimedes]dtest4";
static const char prog5[] = "[arkimedes]dtest5";
static const char prog6[] = "[arkimedes]dtest6";
static const char prog7[] = "[arkimedes]dtest7";

int main(void)
{
  uint32_t child1;
  uint32_t child2;
  uint32_t child3;
  uint32_t child4;
  uint32_t child5;
  uint32_t child6;
  uint32_t child7;

  child1 = syscall_exec(prog1, 15000);
  child2 = syscall_exec(prog2, 0);
  child3 = syscall_exec(prog3, 1000);
  child4 = syscall_exec(prog4, 22500);
  child5 = syscall_exec(prog5, 0);
  child6 = syscall_exec(prog6, 7500);
  child7 = syscall_exec(prog7, 4000);

  syscall_join(child1);
  syscall_join(child2);
  syscall_join(child3);
  syscall_join(child4);
```

```
  syscall_join(child5);
  syscall_join(child6);
  syscall_join(child7);

  syscall_halt();
  return 0;
}
```

This was expected to return the print statement in the following order: (1000, 4000, 7500, 15000, 22500, 0, 0). This was the result when running it with a single CPU.
However when running it with multiple CPUs it would still print the low priorities last, but the order of thread with priorities would be scrambled (at least 4000 is the last occuring before the 0's in this scenario, but the rest is ordered). I was not able to identify or resolve this issue.
I used several orders of my calls to ensure that, on one CPU, that it was not just a 'coincidence' that the order was correct.

1.3 Named pipes - Buenos Process Communication

1.3.1 Initialization and mounting at startup

To do this, the function `pipe_init()` is called in `init/main.c`

```
void init(void)
{
  (..)

  kprintf("Initializing Pipe Filesystem\n");
  fs_t* pipe = pipe_init();
  vfs_mount(pipe, pipe->volume_name);

  (..)
}
```

After this, it is mounted by use of function `vfs_mount` where volume name is given with the `fs_t` from calling `pipe_init()`.
In `fs/pipe.c`, two structures are modified.

```
typedef struct {
```

```
  /* Data needed */
} pipe_t;

typedef struct {
  semaphore_t *lock;
  pipe_t pipes[16];
} pipefs_t;
```

We have a fixed array with a new type `pipe_t` which has some data. There is no data in the structure since implementation of (3b) has not been done.

1.3.2 Implement VFS methods

Not implemented.

1.3.3 Test and discuss your implementation

Due to (3b) has not been implemented, this task is not completed.

2 Theoretical Part

2.1 Semaphores

2.1.1 a

Devise a synchronization of these delivery processes using semaphores. Explain for each semaphore how it is initialized and which role it plays. Then insert calls to P and V into the code for the three activities.

Start by initializing the semaphores.
One is needed for how much space there is in the factory, *space*, initialized to 12.

```
createSemaphore(space, 12);
```

One is needed for the amount of chairs, *chair*, initialized to 0.

```
createSemaphore(chair, 0);
```

One is needed for the amount of tables, *table*, initialized to 0.

```
createSemaphore(table, 0);
```

There is also need for a semaphore to check if the platform is free or not, $free$, initialized to 1.

```
createSemaphore(free, 1);
```

Now the three activities are rewrote to use the semaphores.

```
void deliverTable() {
  while (true) {
    P(table);
    P(free);
    approach(platform);
    load(table);
    leave(platform);
    deliver(table);
    V(space);
    V(free);
  }
```

```
}

void deliverChair() {
  while (true) {
    P(chair);
    P(free);
    approach(platform);
    load(chair);
    leave(platform);
    deliver(chair);
    V(space);
    V(free);
  }
}

void fromFactory() {
  while (true) {
    P(space);
    P(space);
    P(space);
    P(free);
    approach(platform);
    unload(table);
    unload(chair);
    unload(chair);
    leave(platform);
    V(table);
    V(chair);
    V(chair);
    V(free);
  }
}
```

2.1.2 b

Argue that your solution synchronizes correctly and that deadlocks cannot happen. Can several vans and lorries be used?

For both `deliverChair()` and `deliverTable()` it waits for a table or chair is ready to be loaded before checking if the platform is free. This is because otherwise one could come to the platform and no chair or table is ready to be loaded causing a deadlock. However the way it's implemented makes sure they can only come to the platform if something is ready to be loaded.
If something is loaded, then `space` is increased once and platform is freed.

For `fromFactory()` it waits until `space` has at least 3 free items before taking the platform, since if it did it the other way around it would be the same case as above, where a deadlock might happen if there is no space.
Otherwise the unloading is done and `chair` and `table` is incremented twice and once, respectively, and the platform is freed.

Since `space` is only incremented once every time P is called on it, this ensures `space` is between 0 and 12 and if it is between 0 and 2, then either `deliverChair()` or `deliverTable()` is being used.

With this implementation, several vans and lorries *can not* be used, since a deadlock can happen if multiple lorries calls P on `space` only one time. If 12 lorries do this, nothing will ever be delivered.
However, by ensuring that only 1 lorry can call P on `space`, it is possible to have multiple lorries (and vans). This is completed by creating another semaphore, *access*, initialized to 1.

```
createSemaphore(access, 1);
```

and then rewriting the activity `fromFactory` as seen below

```
void fromFactory() {
  while (true) {
    P(access);
    P(space);
    P(space);
    P(space);
    V(access);
    P(free);
    approach(platform);
    unload(table);
    unload(chair);
```

```
    unload(chair);
    leave(platform);
    V(table);
    V(chair);
    V(chair);
    V(free);
  }
}
```

where the semaphore `access` is called P on before lorries are able to call P on `space`, thus ensuring that the deadlock described above can not happen, since just one 'lorry' can call P on `space`. This makes it possible to have several lorries and vans.

2.2 Paging

k denotes the amount of frames available.
ρ denotes the request sequence of pages.
$cost_{k,A}(\rho)$ denotes the number of pagefaults made by an algorithm A.

In following cases, determine if the claims are true or false.

2.2.1 a

There exists a request sequence ρ_1 for which $cost_{3,LRU}(\rho_1) < cost_{3,FIFO}(\rho_1)$. What is $cost_{3,MIN}(\rho_1)$.

This is true. To justify, an example is created where this is the case. The sequence (0,1,2) has already been put into the frames (since this is 3 pagefaults for both algorithms). The full sequence ρ_1 is (0,1,2,0,3,0)

FIFO example for the sequence (0,3,0) following (0,1,2).

0	3	0
0 1 2	3 1 2	3 0 2
No pagefault	Pagefault	Pagefault

LRU example for the sequence (0,3,0) following (0,1,2).

0	3	0
0 1 2	0 3 2	0 3 2
No pagefault	Pagefault	No Pagefault

In this case LRU has less pagefaults than FIFO. This is because FIFO does not update the "queue" when there is a hit.

For the sequence $\rho_1 = (0, 1, 2, 0, 3, 0)$, $cost_{3,MIN}(\rho_1)$ is 4 total (or 1 extra) where removing either page 1 or 2 in the second (or fifth for the full sequence) request does not matter.

2.2.2 b

There exists a request sequence ρ_2 for which $cost_{3,FIFO}(\rho_2) < cost_{3,LRU}(\rho_2)$.

This is true. Again this is justified with an example. Consider the sequence $\rho_2 = (0, 1, 2, 0, 3, 1)$.

FIFO example for the sequence (0,3,1) following (0,1,2).

0	3	1
0 1 2	3 1 2	3 1 2
No pagefault	Pagefault	No pagefault

LRU example for the sequence (0,3,1) following (0,1,2).

0	3	1
0 1 2	0 3 2	0 3 2
No pagefault	Pagefault	Pagefault

Where FIFO only has one (extra) pagefault compared to LRU's (extra) 2 pagefaults.

2.2.3 c

For some k, there exists a request sequence ρ_3 for which $cost_{3,MFU}(\rho_3)$ is significantly higher than $cost_{3,MIN}(\rho_3)$.

This is true. MFU removes the page request that is most frequent. Considering a case where the same page, called p, is requested almost all the time and the rest of the pages are only requested *once*. Call this arbitrary page for q. This means that every time a replacement is needed, p is replaced, and then p is requested again. This takes 2 replacements whereas if one of the other pages had been replaced, it would only take 1 replacement.
For clearance, take the sequence (p, p, q, .., p, p, q) where all k frames are used. Every time a new q is requested a replacement is needed. This means that p is replaced and inserted again in the request right after, taking 2 replacements. If another arbitrary page q had been replaced, it would only take 1 replacement, thus making MFU making twice the amount of pagefaults which is significantly higher.

2.2.4 d

LRU cannot exhibit Belady's anomaly.

This is true. According to (SGG 8th edition, page 377), LRU does not suffer from Belady's anomaly due to the fact that it is a stack algorithm. A stack algorithm is shown to never exhibit Belady's anomaly.
From (SGG 8th edition, page 377):

> For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory.

So LRU cannnot exhibit Belady's anomaly.

2.2.5 e

For no request sequence, the cost of a premature page-replacement algorithm can be smaller than the cost of MIN.

This is false. Since page replacements has no cost, but only misses has, then theoretically, if a premature page-replacement algorithm works the same way as MIN (that is, knowing what page requests will come), that means it can simply make the replacement instead of actually making a miss first.
Therefore a premature page-replacement algorithm *can* be smaller since there are no page faults at all.
However, if the page requests are unknown, then it would be true because it is not possible.

2.3 Memory Management

2.3.1 a

The buddy system is a way to allocate memory where the segments have size equal to the powers of 2. So a request to allocate n units will use the nearest power of 2 (rounded up) segment.
The main advantage of this is that two segments quickly can be merged to former larger segments. Another advantage is that every segment is at an address that is divisible with its size.
The datastructure used to handle this is a binary tree since it 'splits' in two for each step.

To implement `malloc()` and `free()` with the buddy system and make it run in O(lg N) time, we have a list for each block size that contains free blocks, starting with one large block containing the entire memory. If we want to allocate b bytes, we round it up to the nearest power of 2 and look through the list which contains the blocks of this size. If there is one free, we can allocate this.
If not, we multiply this by 2 (to get the next block size) and check if there are any free blocks here. This works recursively up the binary tree until a free block is found.
If a free block is found that is larger than b rounded up to the next power of 2, we remove this block from its 'free' list, add one half of the block to their corresponding free list, return the other half and keep doing this until

we have the initial block size we wanted. This is then returned to the process requesting it.
If there is no free block available, this will return `NULL`.
To free the memory again, we look for the buddy block and if it is free, merge them into one block that is double the size. We keep doing this until the blocks can no longer can be merged.

2.3.2 b

Using the buddy system can lose up to 50% of the bytes that needs to be allocated since it is powers of 2. So if there is a need to allocate $2^n + 1$ bytes, then there will be allocated 2^{n+1} bytes. The overhead of 8 bytes is not counted towards the loss from internal fragmentation (otherwise, the worst case would be $50\% + 7$ bytes, where it is 50% of the bytes allocated).

External fragmentation occurs when a small segment in memory is not used between two segments of allocated memory. Observe the following scenario, a small portion of memory, where a process, $p1$, allocates 1 KB, and another process, $p2$, allocates 2 KB. In the buddy system it will be allocated as seen here

1 KB	1 KB	1 KB	1 KB
p1	unused	p2	

In the middle 1 KB is 'wasted'. Considering this happens several places in memory, these *could* have been combined to create a larger segment to be allocated. For example, if this happened two places in memory and a process $p3$ requested 2 KB to be allocated, these could have been merged to fulfill this request.

2.3.3 c

Explain how realloc can be optimized to do better than this in terms of memory consumption and performance. Which conditions must be fulfilled for the optimization to be possible?

One optimization would be to check if there is enough room adjacent to the memory already allocated. If there is, simply allocate the adjacent space in memory, otherwise `realloc` can work as in the simple implementation.