Lab 3 : Scheduling and Threading

Part 1 : Xv6 Scheduler

Exercise 1 : Laying scheduling groundwork

- `sched.h` : Created the header file. Defines `SCHED_OTHER` , `SCHED_RR` , `SCHED_FIFO` .
- `syscall.h` : added syscall number for the setscheduler syscall
- `syscall.c` : declared and added a new `sys_setscheduler` function to the `syscalls` array.
- `usys.S` : added `SYSCALL(setscheduler)`
- `user.h` : added `setscheduler` to the list of syscalls.
- `sysproc.c` : defined the `sys_setscheduler()` function.
- `proc.h` : added a `scheduler` field to the existing `stuct proc`
- `proc.c` : set default value of `scheduler` upon creation in `allocproc()`

Exercise 2 : Implementing SCHED_RR

- `proc.c` : refactored `scheduler()` . Defined a new function `rr_scheduler_lab3()` which implements the `SCHED_RR` scheduling policy and is now called by `scheduler()` . Also defined a new helper function `runproc_lab3()` .
- `defs.h` : added declaration of `rr_scheduler_lab3()` , `runproc_lab3()`
- `schedtest.c` : unit test for the round robin scheduler
- `Makefile` : added `_schedtest` to `UPROGS`

Test case : 4 child processes (numbered from 0 to 3 in the output), all `SCHED_RR` . *Note : The number used to identify the child process here and in subsequent tests is not its real process id, but it still uniquely identifies the process.*

Code :

```c
#include "types.h"
#include "user.h"


#define NB_FORKS 4
#define OCCUPY_LEN 20000000

#define NB_CHUNKS 10
#define CHUNK_LEN (OCCUPY_LEN / NB_CHUNKS)

void occupy(int id){
    for(int i = 0; i < OCCUPY_LEN; i++){
        if(i % CHUNK_LEN == 0){
            printf(1, "process : %d - done %d / %d\n", id, i/CHUNK_LEN + 1, NB_CHUNKS);
        }
    }
}

int main(void){
    int pid, i;
    for(i = 0; i < NB_FORKS; i++){
        if(fork() == 0){ //fork() returns 0 in child proc
            occupy(i);
            exit();
        }
    }

    for(i = 0; i < NB_FORKS; i++) //Wait for each child
        wait();

    exit();
}
```

Output :

```
$ schedtest
process : 0 - done 1 / 4
process : 1 - done 1 / 4
process : 2 - done 1 / 4
process : 3 - done 1 / 4
process : 0 - done 2 / 4
process : 1 - done 2 / 4
process : 2 - done 2 / 4
process : 3 - done 2 / 4
process : 1 - done 3 / 4
process : 2 - done 3 / 4
process : 3 - done 3 / 4
process : 0 - done 3 / 4
process : 1 - done 4 / 4
process : 2 - done 4 / 4
process : 3 - done 4 / 4
process : 0 - done 4 / 4
$
```

Note 2 : It may be the case that we switch while printing the feedback string – causing the console output to look a bit weird –

but this is expected behavior. (To get a clean output as above, I repeated the test a few times.)

Exercise 3 : Implement SCHED_FIFO

- `defs.h` : added declaration of `fifo_scheduler_lab3()`
- `proc.c` :
    - Modified `struct ptable` to hold a queue (implemented as linked list) of FIFO processes. Added both `fifo_head` and `fifo_tail` pointers (for efficiency). Also modified `struct proc` in `proc.h` to add a `next` pointer to allow making linked list out of processes. Added default initalization of `next` in `allocproc()` .
    - Restructured `scheduler()` to prioritize FIFO processes over RR.
    - Added definition of `fifo_scheduler_lab3()`
- `sysproc.c` :
    - modified `sys_setscheduler()` to add `SCHED_FIFO` processes to the FIFO queue and remove them if previous changing scheduler policy from `SCHED_FIFO` to something else. *Note : remove is O(n) because of singly linked list, but it was way simpler and I assumed we woudln't change the scheduler that often.*
    - I defined a new method `setscheduler_lab3()` in `proc.c` to do that (and added its declaration to `defs.h`).
    - Added functions `enqueue()` , `dequeue()` and `remove()` to manipulate the FIFO queue.
- `trap.c` : modified to avoid interrupting FIFO processes.
- `schedtest.c` : Modified to test FIFO scheduler.

Test case : Processes 0 and 2 are `SCHED_FIFO` and 1 and 3 are `SCHED_RR` in the output below.

Code :

```c
#include "types.h"
#include "user.h"
#include "sched.h"


#define NB_FORKS 4
#define OCCUPY_LEN 20000000

#define NB_CHUNKS 4
#define CHUNK_LEN (OCCUPY_LEN / NB_CHUNKS)

void occupy(int id){
    for(int i = 0; i < OCCUPY_LEN; i++){
        if(i % CHUNK_LEN == 0){ //print some feedback on progress
            printf(1, "process : %d - done %d / %d\n", id, i/CHUNK_LEN + 1, NB_CHUNKS);
        }
    }
}

int main(void){
    setscheduler(SCHED_FIFO); //We don't want the parent to be preempted until it has forked all children
    int pid, i;
    for(i = 0; i < NB_FORKS; i++){
        if(fork() == 0){ //fork() returns 0 in child proc

            setscheduler(SCHED_FIFO);
            if(i%2 == 1)
              setscheduler(SCHED_RR);

            occupy(i);
            exit();
        }
    }

    setscheduler(SCHED_RR); //Now the parent can be preempted
    for(i = 0; i < NB_FORKS; i++) //Wait for each child
        wait();

    exit();
}
```

Output :

```
$ schedtest
process : 0 - done 1 / 4
process : 0 - done 2 / 4
process : 0 - done 3 / 4
process : 0 - done 4 / 4
process : 2 - done 1 / 4
process : 2 - done 2 / 4
process : 2 - done 3 / 4
process : 2 - done 4 / 4
process : 1 - done 1 / 4
process : 3 - done 1 / 4
process : 1 - done 2 / 4
process : 3 - done 2 / 4
process : 1 - done 3 / 4
process : 3 - done 3 / 4
process : 1 - done 4 / 4
process : 3 - done 4 / 4
$
```

Exercise 4 : Implement Prioritization

- `proc.h` : Added a `priority` field to `struct proc`
- `sched.h` : Define `PRTY_DFLT` (default priority level) to be `0`
- `proc.c` :
  - Added default initialization of `priority` field in `allocproc()`
  - Added `rr_head` and `rr_tail` pointers.
  - Modified `enqueue()`, `dequeue()` and `remove()` to allow specifing a priority list (rr or fifo).
    Note : Adding a `policy` argument to `enqueue()` and `remove()` isn't strictly necessary, as one could instead require for the given `proc` argument to already have its `scheduler` field set to the required policy instead. I simply found it less error prone to have it as an argument.
  - Added a function `setqueueptrs()` which sets `head` and `tail` pointers to the right queue, depending on `policy`. This allows to not have to further differentiate `SCHED_RR` and `SCHED_FIFO` in `enqueue()`, `dequeue()` and `remove()` (and thus write "generic" code).
  - Modified `enqueue()` to enforce priorities by carefuly inserting a proc in the right place.
  - Modified `rr_scheduler_lab3()` and `fifo_scheduler_lab3()` : Since both are now implemented by with a priority queue, I moved the code to a new functions `scheduler_lab3()` which takes a `policy` as parameter and removed these two functions. I updated `defs.h` accordingly. Important note : `scheduler_lab3` will `dequeue()` a process, run it, and then `enqueue()` it again if it has not called `exit()`. This allows both RR and FIFO policies to work properly :
    - For `SCHED_RR` : Removing from the priority queue and reinserting will push the process back in the queue right behind processes with the same priority level.
    - For `SCHED_FIFO` : Since these processes aren't preempted, they won't get reinserted in the queue.
  - Modified the `setscheduler()` syscall to also take a priority level, which implied modifiying `sysproc.c`, `user.h`, `defs.h` and `proc.c`.
  - Modified `dequeue()` to only dequeue a process if it is `RUNNABLE`. I defined a function `findrunnable()` and modified `remove()` to only perform the pointer rewirings (no searching anymore, takes a `prev` argument instead).

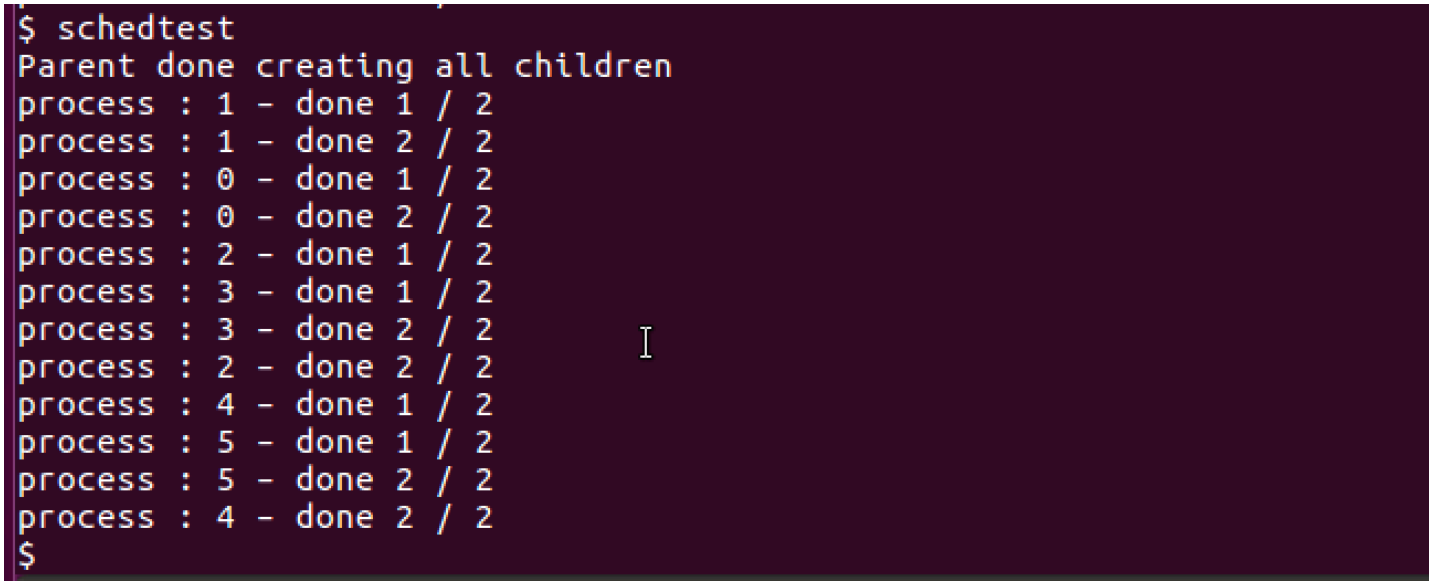
- `schedtest.c` : Changed the test priorites as well as `SCHED_FIFO` and `SCHED_RR` behaviors.
- `proc.h` : Added `proc.c` local functions' signatures

Test case : The one described in the assignment *(process ids shifted by 1)*, namely :

- Proc. 0 : `SCHED_FIFO` , priority 3
- Proc. 1 : `SCHED_FIFO` , priority 4
- Proc. 2 : `SCHED_RR` , priority 3
- Proc. 3 : `SCHED_RR` , priority 3
- Proc. 4 : `SCHED_RR` , priority 2
- Proc. 5 : `SCHED_RR` , priority 2

Code : See `schedtest.c`

Output :



Note : I make use of `sleep()` , which "does the job" even though technically incorrect. This is the solution endorsed by Prof. Devecsery, see Piazza @293.

Question :

> Given this information, write down a command that would cause a number of processes (e.g., 3) to run in reverse order of PID assuming that the child processes' PIDs increase as they are created and that they do not start counting until they are all created.

Such a command could be : `sudo ./count 3 1000 "chrt -f -p 1" "chrt -f -p 2" "chrt -f -p 3"`

Suppose that the child processes' PIDs are 101, 102 and 103. Then we have :

- Process 101, priority 1
- Process 102, priority 2
- Process 103, priority 3

And they'll run in priority order `3 -> 2 -> 1` , ie. PIDs `103 -> 102 -> 101` which is what we wanted.

Bonus

Suboptimal behaviors in my implementation :

- `setscheduler_lab3()` yields even if the next process to be executed is the currently running process. Could perform some checks to determine in `setscheduler_lab3()` to avoid that (e.g. if new priority level is higher than previous priority level).
- Processes get removed from their priority list when they start to execute and put back in when they get preempted / yield (if they haven't yet exited). That simplifies the implementation of the `SCHED_RR` policy, but isn't very efficient since the reinsertion goes over processes which were closer to the head of the priority list already (and we know the reinsert process only needs to be "pushed back" by a few places, namely right after other processes having the same priority level. A more efficient way would be to remember the position of the process in the priority list, and start from here when reinserting.
- An even more efficient way would be to have pointers for each of the priority level (in an array), allowing constant time insertion of a process in a priority list.

Part 2 : Kernel threads

Exercise 1 : Demonstrate user-level thread limitations

- `uthread.c` : Added the file from Homework 5.
- `uthread_switch.S` : Added the file from Homework 5.
- `Makefile` : Modified to add `uthread` user program (and be able to make).

All threads block when one thread blocks (use I/O or something else to force a block)

- `uthread.c` : Modified `mythread()` to block on I/O.

Test case : Two threads. They loop and print some id when they execute. One thread eventually reaches a blocking line (a `read` which waits on `stdin` input).

Code :

```
/*Rest of the file not shown here*/

static void
mythread(void)
{
  int i;
  printf(1, "my thread running\n");
  for (i = 0; i < 100; i++) {
    printf(1, "my thread 0x%x\n", (int) current_thread);
    thread_yield();
  }

  //Block on I/O
  printf("About to block.\n");
  char buf[10];
  read(0, buf, 10); //Read 10 bytes from stdin. Will block until 10 bytes given.
  printf("Not reaching here due to blocking I/O\n");


  printf(1, "my thread: exit\n");
  current_thread->state = FREE;
  thread_schedule();
}


int
main(int argc, char *argv[])
{
  thread_init();
  thread_create(mythread);
  thread_create(mythread);
  thread_schedule();
  return 0;
}
```

Output :

```
my thread 0x2E08
my thread 0x4E10
my thread 0x2E08
my thread 0x4E10
my thread 0x2E08
my thread 0x4E10
my thread 0x2E08
my thread 0x4E10
About to block.
```

We see that no progress is made by either of the threads once one of the thread blocks. See next section *"Threads are locked to same CPU"* for a more detailled explanation / reasoning.

Threads are locked to same CPU (no performance gain)

- Adding the `getcpu()` syscall : Modified `syscall.h` , `syscall.c` , `usys.S` , `user.h` , `sysproc.c` in a similar way as Part 1 Exercise 1 above.
- `uthread.c` : Added the cpu number to the printf.

Code :

Only modified the printed string in `mythread()` : (in diff format)

```
-    printf(1, "my thread 0x%x\n", (int) current_thread);
+    printf(1, "my thread 0x%x on cpu %d\n", (int) current_thread, getcpu());
```

Also, I ran `make qemu-nox-gdb` **without** `CPUS = 1` .

Output :

```
my thread 0x4E30 on cpu 0
my thread 0x2E28 on cpu 0
my thread 0x4E30 on cpu 0
my thread 0x2E28 on cpu 0
my thread 0x4E30 on cpu 0
my thread 0x2E28 on cpu 0
my thread 0x4E30 on cpu 0
About to block.
```

They are locked on the same CPU since they clearly executed on the same CPU already, and nothing gets printed out to the console after one thread blocks. In particular, the non-blocking thread doesn't reach the `printf("About to block")` line ("About to block." only appears once in the output), which shows that it cannot make any progress (and thus allows us to rule out the possibility that both threads reached the blocking I/O line).

Exercise 2 : Implement Kernel threads

Implement `clone()`

- Adding the `clone()` syscall : Modified `syscall.h` , `syscall.c` , `usys.S` , `user.h` , `sysproc.c` in a similar way as Part 1 Exercise 1 above.
- `proc.h` : Added a `clone` field to `struct proc`
- `proc.c` :
    - Added a `clone_lab3()` function which does the actual work of the `clone()` syscall. *Note : Most of it is borrowed from `fork()` .*
    - Initializied `clone` field in `allocproc()`
    - Set `clone` field in `clone()`
    - Modified `wait()` to only call `freevm()` for child processes which weren't created via `clone()` (avoids freeing parent memory if it's still running, double freeing, …)
- `defs.h` : Added `clone_lab3()` declaration.
- `kthread.c` : Added a new file and test program.
- `Makefile` : added `_kthread` to `UPROGS` .

Test case : The parent clones a child, then waits for it. The child prints a string and exits.

Code :

```
int main(void)
{
  setscheduler(SCHED_FIFO, 0); //Don't want the parent to be interupted by child

  char* stack;
  if((stack = malloc(PGSIZE)) == (void*)0){
      printf(2, "malloc failed\n");
      exit();
  }

  printf(1, "Parent calls clone\n");
  if(clone(stack, PGSIZE) == 0){ //Returns 0 in the child process
    printf(1, "Hello from cloned child !\n");
    printf(1, "Child exiting\n\n");
    exit();
  }


  printf(1, "Parent will wait\n\n");
  wait();
  printf(1, "Parent exiting.\n");
  exit();
}
```

Output:

```
$ kthread
Parent calls clone
Parent will wait

Hello from cloned child !
Child exiting

Parent exiting.
$
```

Questions :

> Do you need a new kernel stack? How is this handled?

We do need a new kernel stack since each kernel threads should be able to execute concurrently, but this is handled for us by `allocproc()` .

> The new process should share memory with the parent process (how does `copyuvm()` change?)

We don't use `copyuvm()` anymore, since both processes share memory. Instead, we just make the `pgdir` pointer of the child point to the same top-level page table as its parent.

Implement simple user threading API and supporting functions

- `kthread.c` :
  - Implemented `thread_create()` . Required a new function `free_stack_and_exit()` , since we cannot call `exit()` after having freed the stack. *Note : see Piazza @337*
    - Created a new header file `free_stack_and_exit.h` and copied the given code.
    - Created a new asm file `free_stack_and_exit.S` and copied the given code.
  - Implemented `thread_join()` , which is a simple wrapper for `wait()` .

Exercise 3 : Demonstrate and Compare

Show two threads with one blocking on I/O

- `kthread.c` :
  - Added two routines `blockingIO()` and `occupy()`
  - Wrote a simple test in `main()` .

Test case : Create two threads, one blocking and one printing progress status.

Code :

```c
/**
 * @brief blocks waiting for stdin input
 * @param unused unused
 */
void* blockingIO(void* unused){
  //Block on I/O
  printf(1, "Thread %d about to block.\n", getpid());
  char buf[10];
  read(0, buf, 5); //Read 5 bytes from stdin. Will block until 5 bytes given.
  printf(1, "Got input. Not blocking anymore.\n");

  return (void*)0;
}

/**
 * @brief Loops for some time, printing status at regular interval
 * @param unused unused
 */
void* occupy(void* unused){
  for(int i = 0; i < 10000; i++){
    if(i % 2000 == 0) //print some feedback on progress
      printf(1, "Thread %d - done %d / 5\n", getpid(), i/2000 + 1);
  }

  return (void*)0;
}

int main(void)
{
  setscheduler(SCHED_FIFO, 0); //Don't want the parent to be interupted by child

  thread_create(occupy, 0);
  thread_create(blockingIO, 0);

  thread_join();
  thread_join();

  exit();
}
```

Output :

```
$ kthread
Thread 7 - done 1 / 5
Thread 7 - done 2 / 5
Thread 7 - done 3 / 5
Thread 8 about to block.
Thread 7 - done 4 / 5
Thread 7 - done 5 / 5
asdf
Got input. Not blocking anymore.
$
```

We see that thread 7 continues to make progress whilst thread 8 is blocking. I (typed `"asdf"` and) pressed `enter` to unblock thread 8, which then completes and exits cleanly.

Show two threads running on separate CPUs at same time

- `kthread.c` :
  - Modified `occupy()` to print cpu current thread is running on.
  - Modified the test.

Test case : Create two threads, both with routine `occupy()` .

Code : Only minor modifications (here in git diff format)

```
occupy :

- for(int i = 0; i < 10000; i++){
-   if(i % 2000 == 0) //print some feedback on progress
-     printf(1, "Thread %d - done %d / 5\n", getpid(), i/2000 + 1);
+ for(int i = 0; i < 500000; i++){
+   if(i % 100000 == 0) //print some feedback on progress
+     printf(1, "Thread %d on cpu %d - done %d / 5\n", getpid(), getcpu(), i/100000 + 1);

main :

- thread_create(blockingIO, 0);
+ thread_create(occupy, 0);
```

Output :

```
$ kthread
Thread 7 on cpu 0 - done 1 / 5
Thread 8 on cpu 1 - done 1 / 5
Thread 8 on cpu 1 - done 2 / 5
Thread 7 on cpu 0 - done 2 / 5
Thread 8 on cpu 1 - done 3 / 5
Thread 8 on cpu 1 - done 4 / 5
Thread 8 on cpu 1 - done 5 / 5
Thread 7 on cpu 0 - done 3 / 5
Thread 7 on cpu 0 - done 4 / 5
Thread 7 on cpu 0 - done 5 / 5
$
```

Exercise 4 : Integrate your scheduler

- `kthread.c` :
  - Added a new function `thread_setscheduler()` which serves as a wrapper for the `setscheduler()` syscall and allows setting a scheduling policy as well as a priority level for the currently running thread.
  - Modified the test case to mix `fork()` , `clone()` and various priority levels / scheduling policies.

Testing

Due to the limitations of our interfaces (namely, only being able to set the scheduler / priority level for the currently running

process), I had to use quite convoluted ways to allow me to setup a relatively good test. In particular, note that I reused the fact that making processes sleep for a long enough time allowed me to have all threads reach the `setscheduler()` syscall (as done for testing of part 1).

- `kthread.c` :
    - Defined a new `struct test_struct` which wraps both a routine to execute and its argument, and a scheduling policy and priority level.
    - Defined a new function `setsched_sleep_do()` which takes a `struct test_struct` (pointer) and :
        1. Calls `thread_setscheduler()` to set the scheduling policy / priority level
        2. Sleeps for a given amount of time
        3. Executes the actual routine.

Using this function as a "wrapper" for the actual routine we want to execute, it is possible to set the policy / priority level for a thread, make it sleep (to allow other threads to do the same), then execute the routine.

Test case :

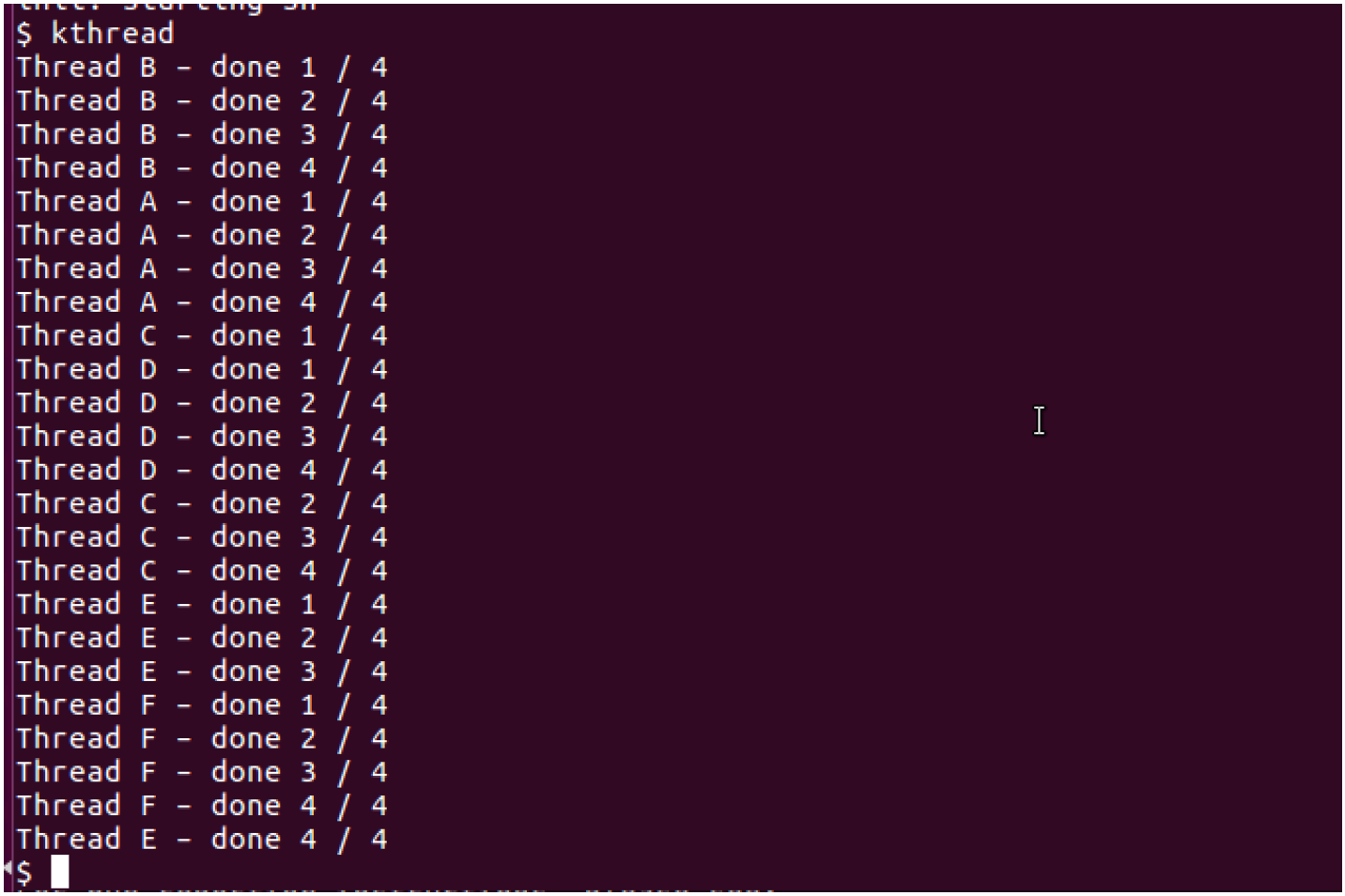
The parent spawns 2 threads A and B and forks two children C and D. The first of the children spawns two threads E and F too.

The scheduling policies / priorities are as follows (same as the final test of part 1) :

- A : `SCHED_FIFO` , priority 3
- B : `SCHED_FIFO` , priority 4
- C : `SCHED_RR` , priority 3
- D : `SCHED_RR` , priority 3
- E : `SCHED_RR` , priority 2
- F : `SCHED_RR` , priority 2

Code : See `kthread.c`

Output with `CPUS = 1` :



Threads execute in the expected order, as described in the Lab 3 Part 1 assignment.

Output with multiple CPUs :

```
$ kthread
B 0-1/4
B 0-2/4
B 0-3/4
A 1-1/4
B 0-4/4
C 0-1/4
C 0-2/4
A 1-2/4
A 1-3/4
A 1-4/4
D 0-1/4
D 0-2/4
D 0-3/4
C 0-3/4
C 1-4/4
D 0-4/4
E 0-1/4
E 0-2/4
F 1-1/4
F 1-2/4
F 1-3/4
F 1-4/4
E 0-3/4
E 0-4/4
$
```

With multiple CPUs, it got way harder to get a clean output whilst still printing long strings, so I had to condensate the information. In the output above, one line must be interpreted as :
`thread_name` *on* `cpu` – `work_done` / `total_work`

The output still makes sense :

- B runs first on CPU 0, A runs concurrently on CPU 1.
- C and D alternate or run concurently on different CPUs.
- E and F alternate on CPU 0, and they run after C and D.

Question

I'm copy-pasting the code for `push()` *here to get line numbers to refer to.*

```
void push(int key, int value)
{
   elem_t *e = malloc(sizeof(*e));
   e->next = head;
   e->key = key;
   e->value = value;
   head = e; // Put it on the stack
}
```

Example of a non-benign race

Let the stack look like this :

```
head -> (k, v) -> ...
```

where `head ->` means "head points to" and `(k, v)` is an element, and `...` the rest of the list (if any).

Suppose two threads A and B called `push()` at roughly the same time, such that they both execute line 4 before any of them reaches line 7.

At that point, both thread set their respective `e->next` pointer to point to the element at `head`.

Then suppose thread A reaches line 7 first and executes it *(WLOG)*. The stack will look like this :

```
head -> (ka, va) -> (k, v) -> ...
```

Thread B finally reaches line 7 and rewires the `head` pointer to point at is own pushed element. After executing it, the stack will look like :

```
                  (ka, va)
                      |
                      v
head -> (kb, vb) -> (k, v) -> ...
```

And the element pushed by thread A will be "lost", in the sense that it won't be accessible anymore through the `head` pointer of the stack.

Sequence of read and writes demonstrating this race

Assume an initially empty stack.

```
Thread A : executes up to line 4 (included)
Thread B : executes up to line 4 (included)
Thread A : executes up to line 7 (included)
Thread B : executes up to line 7 (included)

Thread B : executes pop, fully
Thread A : executes pop, fully
```

In this sequence and as described above, only the element pushed by B will actually be put on the stack and the other one is lost. As a result, `Thread B : pop` will get it element back, but `Thread A : pop` will get a `null` back, instead of the element it previously pushed.