ECS 150: Project #3 - User-level thread library (part 2)

Prof. Joël Porquet-Lupine

UC Davis, Fall Quarter 2019

1 Changelog

*Note that the specifications for this project are subject to change at anytime for additional clarification. Make sure to always refer to the **latest** version.*

- v1: First publication

2 General information

- Due before **11:59 PM, Tuesday, November 12th, 2019**.
- You will be working with a partner for this project.
  - Remember, you cannot keep the same partner for more than 2 projects over the course of the quarter.
- The reference work environment is the CSIF.

3 Objectives of the project

The objectives of this programming project are:

- Implementing one of the most popular synchronization primitives (semaphores) as specified by a given API.
- Designing a mechanism that allows each thread to have its own private storage.
- Understanding critical sections and synchronization between threads.
- Learning how to test your code, by writing your own testers and maximizing the test coverage.
- Writing high-quality C code by following established industry standards.

4 Program description

4.1 Introduction

The goal of this project is to extend your understanding of threads, by implementing two independent parts:

1. Semaphores, for efficient thread synchronization by using waiting queues.
2. Per-thread protected memory regions. Threads will be able to create their own private storage (a.k.a. TPS or *Thread Private Storage*), transparently accessible by an API and protected from other threads.

Concerning the thread management, and in order for this project to be as independent as possible from the previous project, we will use the POSIX `pthreads` library.

For both parts, you are welcome to re-use the queue API that you developed for project 2, or use the pre-compiled object file provided as part of the skeleton code.

4.2 Constraints

Your library must be written in C, be compiled with GCC and only use the standard functions provided by the GNU C Library (https://www.gnu.org/software/libc/manual/) (aka `libc`). *All* the functions provided by the `libc` can be used, but your program cannot be linked to any other external libraries (apart from the pthread library).

Your source code should adopt a sane and consistent coding style and be properly commented when necessary. One good option is to follow the relevant parts of the Linux kernel coding style (https://www.kernel.org/doc/html/latest/process/coding-style.html).

4.3 Assessment

Your grade for this assignment will be broken down in two scores:

- Auto-grading: ~60% of grade

  Running an auto-grading script that tests your program and checks the output against various inputs

- Manual review: ~40% of grade

  The manual review is itself broken down into different rubrics:

  - Report file: ~50%
  - Submission : ~5%
  - Makefile: ~5%
  - Semaphore implementation: ~15%
  - TPS implementation and testing: ~20%
  - Code style: ~5%

5 Suggested work phases

5.1 Phase 0: Skeleton code

The skeleton code that you are expected to complete is available in `/home/cs150jp/public/p3/`. This code already defines most of the prototypes for the functions you must implement, as explained in the following sections.

```
$ cd /home/cs150jp/public/p3
$ tree
.
├── libuthread
│   ├── Makefile*
│   ├── queue.h
│   ├── queue.o
│   ├── sem.c*
│   ├── sem.h
│   ├── thread.h
│   ├── thread.o
│   ├── tps.c*
│   └── tps.h
└── test
    ├── Makefile
    ├── sem_buffer.c
    ├── sem_count.c
    ├── sem_prime.c
    └── tps.c
```

The code is organized in two parts. In subdirectory `test`, there are a few test applications which make use of the (improved) thread library. You can compile these applications and run them.

Subdirectory `libuthread` contains the files composing the thread library that you must complete. The files to complete are marked with a star (you should have **no** reason to touch any of the headers which are not marked with a star, even if you think you do...).

Copy the skeleton code to your account.

5.2 Part 1: semaphore API

The interface of the semaphore API is defined in `libuthread/sem.h` and your implementation should go in `libuthread/sem.c`.

Semaphores are a way to control the access to common resources by multiple threads. To keep track of the number of available resource, a semaphore maintains an internal count, which can already be initialized to a certain positive value when the semaphore is created.

Threads can then ask to grab a resource (known as “down” or “P” operation) or release a resource (known as “up” or “V” operation).

Trying to grab a resource when the count of a semaphore is down to 0 adds the requesting thread to the list of threads that are waiting for this resource. The thread is put in a blocked state and shouldn’t be eligible to scheduling.

When a thread releases a semaphore which count was 0, it checks whether some other threads were currently waiting on it. In such case, the first thread of the waiting list can be unblocked, and be eligible to run later.

The semaphore implementation should make use of the functions defined in the thread API, as described in the `thread.h` header. The implementation of this API is provided to you already compiled, as an object file, in `thread.o`. You will probably need to slightly modify your Makefile to not try to recreate this object file when compiling, or remove it when cleaning.

5.2.1 Testing

Three testing programs are available in order to test your semaphore implementation:

- `sem_count`: simple test with two threads and two semaphores
- `sem_buffer`: producer/consumer exchanging data in a buffer
- `sem_prime`: prime sieve implemented with a growing pipeline of threads

It is recommended to look at the code of these testers and understand how they work in order to see how they will stress your semaphore implementation.

5.2.2 Corner case

There is a very specific corner case that you need to consider in order to implement your semaphore correctly. Here is the scenario:

- Thread A calls `down()` on a semaphore with a count of 0, and gets blocked.
- Thread B calls `up()` on the same semaphore, and gets thread A to be awaken
- Before thread A can run again, thread C calls `down()` on the semaphore and snatch the newly available resource.

There are two difficulties with this scenario:

1. The semaphore should make sure that thread A will handle the situation correctly when finally resuming its execution. Theoretically, it should go back to sleep if the resource is not longer available by the time it gets to run. If the thread proceeds anyway, then the semaphore implementation is incorrect.
2. If this keeps happening, thread A will eventually be starving (i.e., it never gets access to the resource that it needs to proceed). Ideally, the semaphore implementation should prevent starvation from happening.

Think about your implementation to at least make it correct for that corner case, and ideally, find a way to avoid any starvation.

5.3 Part 2: TPS API

As you know, threads of a same process all share the same memory address space. In general, this is a great behavior because it allows threads to easily share information. However, it can also be an issue when, typically due to programming bugs, one thread accidentally modifies values that another thread was using. To protect data from being overwritten by other threads, it would be convenient for each thread to possess a protected storage area that only this thread could read from and write to. For this project, we call such memory area, *Thread Private Storage*.

The goal of the TPS API is to provide a single private and protected memory

page (i.e. 4096 bytes) to each thread that requires it.

The interface of the TPS API is defined in `libuthread/tps.h` and your implementation should go in `libuthread/tps.c`.

Note that your TPS implementation should make use of the critical section helpers defined in the thread API, as described in the `thread.h` header, when manipulating shared variables.

5.3.1 Phase 2.1: Unprotected TPS with naive cloning

For this phase, you will need to implement a first version of the API. Below is a few indications to help you started.

In `tps_init()`, you can initialize your API the way you want. Depending on your implementation, it is not necessary that this function contains any code at this point. But if you need some internal objects to be initialized before any TPS can be created, this function is where it should go. For this phase, you can ignore the parameter `segv`.

In `tps_create()`, you must create a TPS for the thread that requires it. As you can notice, this function doesn't return any object. It means that the TPS object must be kept inside the library and shall not be exposed to the client. Each time the client will subsequently interact with its TPS, via `tps_read()` or `tps_write()`, it is up to the library to find the corresponding TPS and operate on it.

> Hint: you can identify client threads by getting their Thread ID with `pthread_self()`.

The page of memory associated to a TPS should be allocated using the C library function `mmap()`. Because `mmap()` allocates memory at the granularity of pages, it will be possible in the next phase to protect these pages by setting no read/write permissions. `mmap()` is very versatile and you will need to study the documentation in order to understand how to allocate a memory page that is private and anonymous, and can be accessed in reading and writing.

In `tps_clone()`, the calling thread is requesting its own TPS whose content must be a copy of the target thread's TPS. For this first phase, you can create a new memory page and simply copy the content from the target thread's TPS with `memcpy()`.

Useful resources for this phase:

- https://www.gnu.org/software/libc/manual/html_mono/libc.html#Memory_002dmapped-I_002fO (https://www.gnu.org/software/libc/manual/html_mono/libc.html#Memory_002dmapped-I_002fO)

5.3.1.1 Testing

One simple testing program, `tps.c`, is available in order to test your first implementation.

However this tester is not nearly enough to really challenge your TPS API, especially for the following phases, 2.2 and 2.3. It is expected that you should write your own comprehensive tester that checks your API, as well as its corner cases. Such tester should be described and discussed in your report, and its code submitted as part of your bundle.

5.3.2 Phase 2.2: Protected TPS

In this phase, you need to modify the previous phase by adding TPS protection.

In `tps_create()`, the memory page should have no read/write permission by default. It's only in `tps_read()` that the page should temporarily become readable, and in `tps_write()` that the page should temporarily become writable.

At this stage, you might now encounter two types of segmentation faults. In addition to the usual programming errors (e.g. dereferencing a NULL pointer), accessing a protected TPS area will also cause a segmentation fault.

A good way to distinguish between these two types of faults is to set up our own segmentation fault handler. Modify `tps_init()` in order to associate a function handler to the signals of type `SIGSEGV` and `SIGBUS`:

```
int tps_init(int segv)
{
    ...
    if (segv) {
        struct sigaction sa;

        sigemptyset(&sa.sa_mask);
        sa.sa_flags = SA_SIGINFO;
        sa.sa_sigaction = segv_handler;
        sigaction(SIGBUS, &sa, NULL);
        sigaction(SIGSEGV, &sa, NULL);
    }
    ...
}
```

Write the signal handler that will distinguish between real segmentation faults and TPS protection faults. Here is a skeleton that needs to be completed:

```
static void segv_handler(int sig, siginfo_t *si, void *context)
{
    /*
     * Get the address corresponding to the beginning of the
page where the
     * fault occurred
     */
    void *p_fault = (void*)((uintptr_t)si->si_addr & ~(TPS_SIZE
- 1));

    /*
     * Iterate through all the TPS areas and find if p_fault
matches one of them
     */
    ...
    if (/* There is a match */)
        /* Printf the following error message */
        fprintf(stderr, "TPS protection error!\n");

    /* In any case, restore the default signal handlers */
    signal(SIGSEGV, SIG_DFL);
    signal(SIGBUS, SIG_DFL);
    /* And transmit the signal again in order to cause the
program to crash */
    raise(sig);
}
```

Useful resources for this phase:

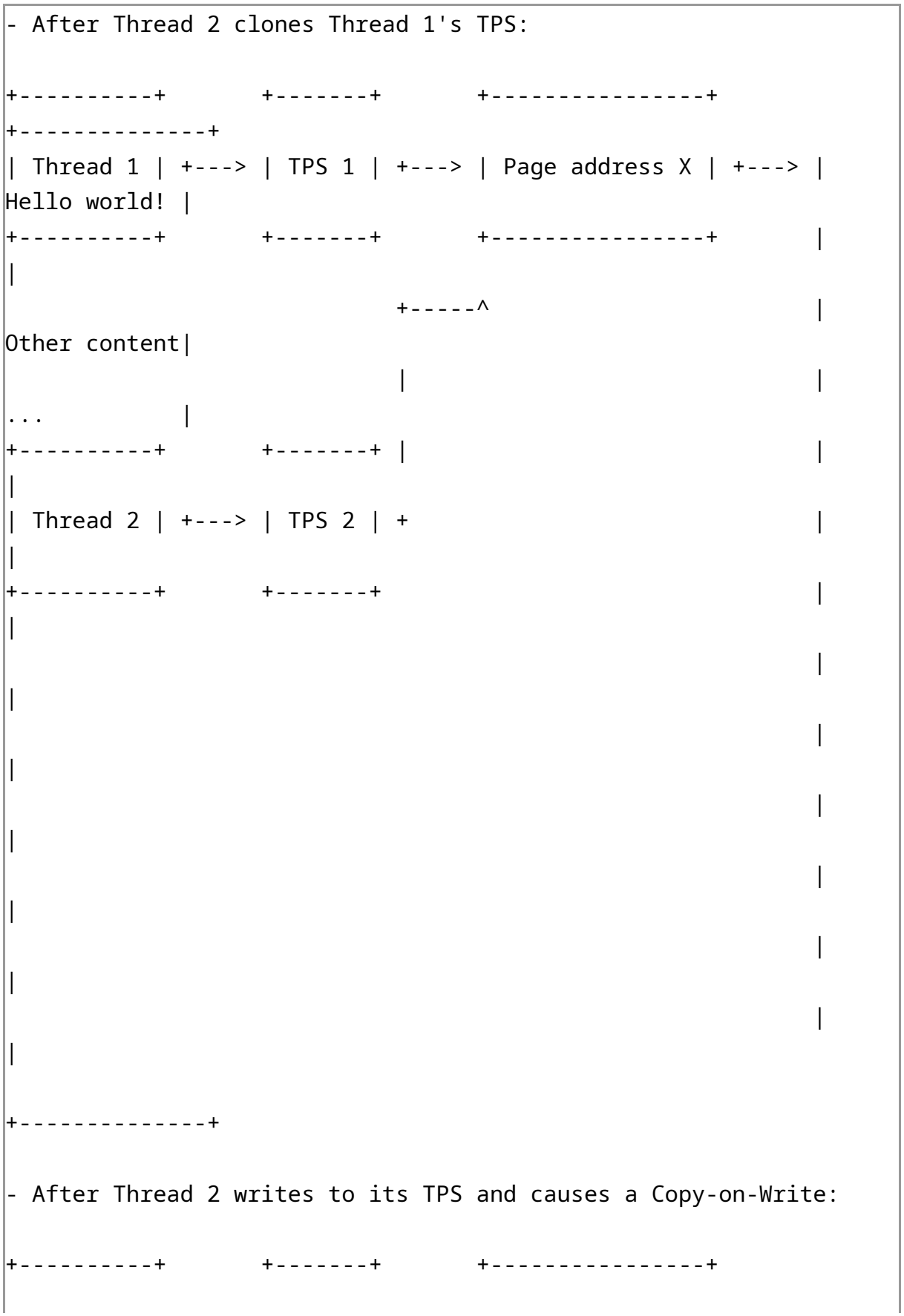
- https://linux.die.net/man/2/mprotect (https://linux.die.net/man/2/mprotect)
- http://pubs.opengroup.org/onlinepubs/7908799/xsh/sigaction.html (http://pubs.opengroup.org/onlinepubs/7908799/xsh/sigaction.html)

5.3.3 Phase 2.3: Copy-on-Write cloning

With the naive cloning strategy adopted so far, the memory page of the target thread is directly copied when being cloned. But assuming that the new TPS

won’t be modified right away, it would be possible to delay both the allocation of a new page and the copy of the content as long as the shared memory page is only read from by either one of the threads sharing it. This approach is called CoW (Copy-on-Write), which saves memory space and avoid unnecessary allocation/copy operations until required.

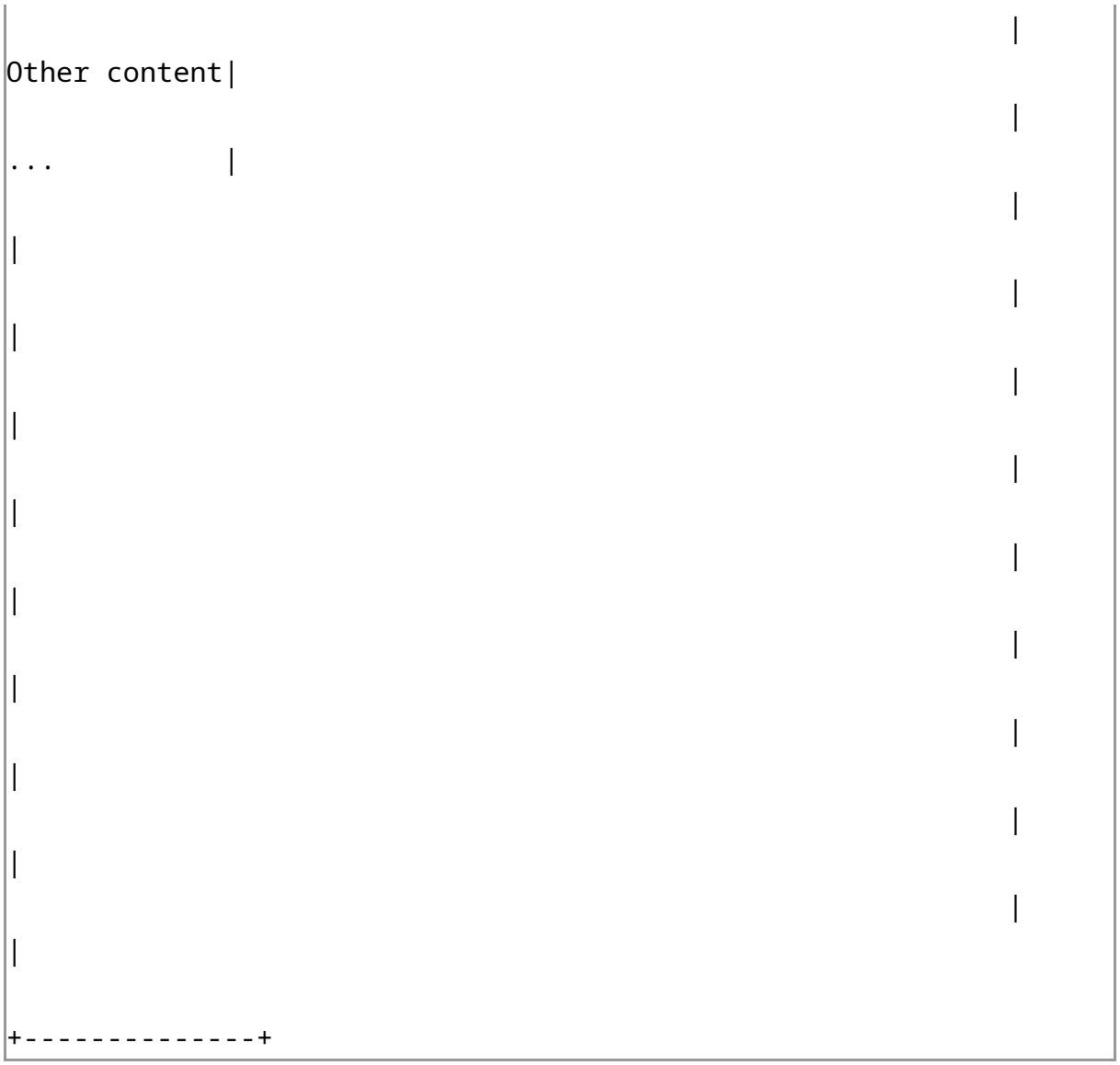
The cloning operation should therefore only create a TPS object that refers to the same page. And it’s only when one of the threads sharing the same page actually calls `tps_write()` that a new page is effectively allocated and filled with the content copied from the original memory page.


- After Thread 2 clones Thread 1's TPS:
Thread 1
TPS 1
Page address X
Hello world!
Other content
...
Thread 2
TPS 2
- After Thread 2 writes to its TPS and causes a Copy-on-Write:

```
+--------------+
| Thread 1 | +---> | TPS 1 | +---> | Page address X | +---> |
Hello world! |
+----------+      +-------+      +----------------+      |
|
                                                          |
Other content|
                                                          |
...         |
                                                          |
|
                                                          |
|
                                                          |
|
                                                          |
|
                                                          |
|
                                                          |
|
                                                          |
|
                                                          |
|
                                                          |
|

+--------------+

+----------+      +-------+      +----------------+
+--------------+
| Thread 2 | +---> | TPS 2 | +---> | Page address Y | +---> |
Hello davis! |
+----------+      +-------+      +----------------+      |
|
```



In order to implement such cloning strategy, you will need to perform a few modifications to your existing implementation:

1. You need to dissociate the TPS object from the memory page. Before you probably had the address of the TPS's memory page as part of the TPS object, now you need an extra level of indirection. You need a page structure containing the memory page's address, and TPS can only point to such page structures. This way, two or more TPSes can point to the same page structure. Then, in order to keep track of how many TPSes are currently "sharing" the same memory page, the struct page must also contain a reference counter.
2. In `tps_clone()`, you need to allocate a new TPS object for the calling

thread, but have the page structure be shared with the target thread and the reference counter updated accordingly.
3. In `tps_write()`, writing to a page that has a reference count superior to 1 should causes a new memory page to be allocated first. This memory page should be the identical copy of the original one, and should now become the private copy of the calling thread, while the original page's reference counter should be decremented. Also, protection just be properly adjusted for both pages: the original page should become protected, while the new page should become temporarily writable during the ongoing writing operation.

6 Submission

Since we will use auto-grading scripts in order to test your code, make sure that it strictly follows the specified output format. More specifically, your code should not print anything (include error messages) that has not been explicitly specified in this document.

6.1 Content

Your submission should contain, besides your source code (library and tester(s)), the following files:

- `AUTHORS`: student ID of each partner, one entry per line. For example:

```
$ cat AUTHORS
00010001
00010002
$
```

- `IGRADING`: **only if your group has been selected for interactive grading for this project**, the interactive grading time slot you registered for.
    - If your group has been selected for interactive grading for this project, this file must contain exactly one line describing your time slot with

the format: `%m/%d/%y - %I:%M %p` (see `man date` for details). For example, an appointment on Monday April 20th at 2:10pm would be transcribed as `04/20/19 - 02:10 PM`.

```
$ cat IGRADING
04/20/19 - 02:10 PM
$
```

- `REPORT.md`: a description of your submission. Your report must respect the following rules:
  - It must be formatted in markdown language as described in this Markdown-Cheatsheet (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet).
  - It should contain no more than 200 lines and the maximum width for each line should be 80 characters (check your editor's settings to configure it automatically –please spare yourself and do not do the formatting manually).
  - It should explain your high-level design choices, details about the relevant parts of your implementation, how you tested your project, the sources that you may have used to complete this project, and any other information that can help understanding your code.
  - Keep in mind that the goal of this report is not to paraphrase the assignment, but to explain **how** you implemented it.
- `libuthread/Makefile`: a Makefile that compiles your source code without any errors or warnings (on the CSIF computers), and builds a static library named `libuthread.a`.

  The compiler should be run with the options `-Wall -Werror`.

  There should also be a `clean` rule that removes generated files and puts the directory back in its original state.

  The Makefile should use all the advanced mechanisms presented in class (variables, pattern rules, automatic dependency tracking, etc.)

Your submission should be empty of any clutter files (such as executable files, core dumps, backup files, `.DS_Store` files, and so on).

6.2 Git bundle

Your submission must be under the shape of a Git bundle. In your git repository, type in the following command (your work must be in the branch `master`):

```
$ git bundle create p3.bundle master
```

It should create the file `p3.bundle` that you will submit via `handin`.

Before submitting, **do make sure** that your bundle was properly been packaged by extracting it in another directory and verifying the log:

```
$ cd /path/to/tmp/dir
$ git clone /path/to/p3.bundle -b master p3
$ cd p3
$ ls
AUTHORS IGRADING libuthread REPORT.md test
$ git log
...
```

6.3 Handin

Your Git bundle, as created above, is to be submitted with `handin` from one of the CSIF computers by **only one person of your group**:

```
$ handin cs150jp p3 p3.bundle
Submitting p3.bundle... ok
$
```

You can verify that the bundle has been properly submitted:

```
$ handin cs150jp p3
The following input files have been received:
...
$
```

7 Academic integrity

You are expected to write this project **from scratch**. Therefore, you cannot use any existing source code available on the Internet, or even reuse your own code if you took this class before.

You are also expected to write this project **yourself**. Asking anyone someone else to write your code (e.g., a friend, or a "tutor" on a website such as Chegg.com) is not acceptable and will result in severe sanctions.

You must specify in your report any sources that you have viewed to help you complete this assignment. All of the submissions will be compared with MOSS to determine if students have excessively collaborated, or have used the work of past students.

Any failure to respect the class rules, as explained above or in the syllabus, or the UC Davis Code of Conduct (http://sja.ucdavis.edu/cac.html) will automatically result in the matter being transferred to Student Judicial Affairs.

UC DAVIS
COMPUTER SCIENCE