COMP 321: Introduction to Computer Systems

Project 4: Checkpointing
Assigned: 3/27/25, **Due: 4/10/25, 11:55 PM**

Overview

You will write a checkpointing system that will periodically (or on demand) create a checkpoint of the current application state. These checkpoints can be used to restart the application from that point in time.

The goals of this project are as follows:

- Understand the use of `mprotect` and segmentation fault handling.
- Become more familiar with the address space layout of applications.
- Understand how to save and restore a complete context, including its entire memory state.

General Overview of Checkpointing

Checkpointing is one mechanism to provide *fault tolerance* for an application. The general idea is to provide snapshots (or "checkpoints") of an application's state so that the application can restart from that state in the case of a failure. This is useful for running programs that take a very long time (days, weeks, or months) to complete. If the system crashes, reboots, or otherwise fails during that time, you do not want to have to start over from the beginning. With checkpointing, you can instead restart from the last checkpoint, only losing any work that occurred after that last checkpoint.

Checkpoints can either be done "on demand", meaning the application requests checkpoints at specific points during its execution, or automatically, meaning the system checkpoints the application periodically on some sort of timer. Taking the actual checkpoint is the same, regardless of whether it is done automatically or on demand.

In order to create a checkpoint from which an application can restart, the *entire* state of the application must be saved. This includes the execution context, all of the application's memory, and the state of all files that are being accessed by the application.

You are familiar with the execution context from the Threads assignment. This includes the register contents, the signal mask, the location of the stack, etc. The memory state of the application requires understanding the memory map of the address space and ensuring that all of its contents are properly saved. Finally, ensuring that the state of all files are saved requires careful thought about how files may be written in order to ensure that their state is consistent with the rest of the checkpoint if you have to restart execution of the program from that point.

You will implement a complete checkpointing system that implements both on demand and automatic checkpointing and allows single-threaded applications to be restarted from any checkpoint taken by your system.

Your checkpointing system will have a simple API that consists of only a single function:

```
int checkpoint(void);
```

This `checkpoint` function can be called by the application for an on demand checkpoint, or the checkpointing system can automatically call it from within a `SIGPROF` signal handler. The system is initialized within a constructor, called `initialize_checkpointer`, and the behavior of the constructor can be controlled by four environment variables:

1. `CKPT_PREFIX`: this variable specifies the prefix for the names of the checkpoint files. This variable must exist and be set to some value.

2. `CKPT_RESTART`: if this variable exists, the program should restart from the last valid checkpoint using the files specified by `CKPT_PREFIX`. If this variable is set, `CKPT_PERIOD` will be ignored and the period specified when the program was first run will be used instead.

3. `CKPT_PERIOD`: if this variable exists, the value is the period at which checkpoints should be taken (in seconds). If the value is 0 or the variable does not exist, then no automatic checkpoints should be taken (only on demand checkpoints requested by the application will happen).

4. `CKPT_DEBUG`: if this variable exists, then debugging information should be printed. The use of this variable is up to you and you can define different values to print different debugging information.

Setting the `CKPT_PERIOD` variable to 0 is useful for debugging purposes, as it will allow you to create checkpoints at specific points and confirm that the checkpoint file contains the correct information. Furthermore, if `CKPT_PERIOD` is not set to 0, requests for manual checkpoints from the application should be ignored. This ensure that checkpoints are either taken periodically or by explicit calls from the application. This will make your system simpler to implement and debug.

Memory Protection

One way to checkpoint an application would be to save its entire state on every checkpoint. However, this is inefficient, as most of the application's memory is not likely to change between checkpoints. Instead, we can write protect all of the application's writable memory after each checkpoint. When the application writes to a memory page, the system would then record the page that has now been modified and update the memory protection on that page to allow that write (and future writes to the same page) to succeed.

This can be accomplished using `mprotect` and a `SIGSEGV` signal handler. As a checkpoint is being written, `mprotect` should be used on any writable pages in the process address space to make them read-only. Then, if the application tries to write any of these pages before the next checkpoint, it will cause a segmentation fault and deliver a `SIGSEGV` signal. The signal handler then becomes responsible for updating the state of the checkpointing system and changing the protection on the memory page so that when the signal handler returns, the application's memory access can successfully complete.

Note the following:

1. The signal handler should only allow writes to the one 4KB page that includes the faulting memory address. The checkpointing system will need to save all of the modified memory and you want to limit what is saved to the smallest amount possible that encompasses the modified memory.

2. The signal handler should *not* store the contents of the memory page for multiple reasons. First, the page hasn't been modified yet. Second, the checkpointing system should capture the state of the memory at the time of the checkpoint, not when the application first accesses the memory. The application could make many modifications to memory in the page after the signal handler makes it writable and before the next checkpoint.

3. The signal handler should not allow writes to pages that are supposed to be read-only or to memory that is outside of the process' address space. Such writes should cause a segmentation violation as usual, because they are application errors.

The checkpointing system must take care to write protect *all* of the memory used by the application in order to ensure that the checkpoints capture the entire application state and can be used to restore that state. However, it also must take care *not* to write protect the memory used by the checkpointing system itself. First, you do not want to checkpoint the checkpointer. Second, and more importantly, if you write protect the checkpointer's memory, it will also cause segmentation violations when it accesses memory while processing the `SIGSEGV` signal and cause another `SIGSEGV` signal, which cannot be handled. This has several implications:

1. You must allocate a separate region of memory for the checkpointer and be careful not to change the protection or save any of this memory in the checkpoint files.

2. You will need to use a custom memory allocator that only allocates memory from this region for the checkpointer. The `mm_malloc` implementation from the previous assignment is suitable for this purpose.

3. The `SIGSEGV` signal handler *can not* run on the application's stack (or it will be checkpointed!), so you must use a separate signal stack which is also not checkpointed (more on this later).

4. When the application calls the `checkpoint` function to initiate an explicit, manual checkpoint, this also cannot run on the application's stack for the same reason the signal handler can not run on the application's stack. You will therefore need to use a separate context and stack to perform the checkpoint.

Beware: It is *very* easy to accidentally allocate memory outside of the checkpointer's stack and heap. This will create a recursive segmentation fault that will crash the program. Be very careful and intentional about how you allocate and access memory in your checkpointer.

Initialization

The constructor for the checkpointer must perform the following tasks (not necessarily in this order) in order to initialize the system. At a high level, the checkpointer must set up all of the state necessary for checkpointing, and register signal handlers so that the checkpointer will run at appropriate times in the future.

1. Read the `CKPT_*` environment variables.

2. Initialize a checkpointing heap.

3. Store any checkpointing state variables on that heap.

4. Allocate and register a stack for signal handlers.

5. Register a signal handler for `SIGSEGV`.

6. If automatic checkpointing is enabled, register a signal handler for `SIGPROF` and start a timer with the appropriate interval.

7. Create and store a context for use by the checkpointer.
8. Read the process address map.
9. Restore the appropriate memory if the process is being restarted.

These are the major tasks that need to be taken by the constructor, but there may be other minor things you need to do and it may be convenient to do these tasks in a different order. Note that if you are restarting the application from a checkpoint, it should also start creating new checkpoints as the application runs.

With automatic checkpointing, the checkpointer will always be initiated by signal delivery. With on demand checkpointing, the checkpointing process is initiated by a function call. It is therefore important to ensure that the signal handlers and on demand checkpointing all run in the checkpointer's context and only use the checkpointer's resources (other than to save, restore, protect, and unprotect the application's memory).

Ranges

The checkpointing system deals with *memory ranges*. A memory range is a contiguous sequence of pages. On CLEAR9, the page size is 4KB, however, your system should not make this assumption. Rather, you should get the page size by calling `sysconf(_SC_PAGE_SIZE)`.

Whenever the checkpointing system is managing memory, it should always do so using ranges, whether it is a single 4KB page or a 1GB memory region.

To specify ranges, we have provided two structures that are defined as follows:

```
struct range {
  uintptr_t start;
  uintptr_t end;
  uint32_t  crc32;
  bool      write_protect;
};

struct range_list {
  struct range       range;
  struct range_list *prev;
  struct range_list *next;
};
```

You should create an appropriate set of functions for manipulating range lists. A range list should be maintained in sorted order (meaning that the start addresses of the ranges increase as you traverse the list) and *not* include any overlapping ranges. This means that when you insert a new range, you may need to coalesce two or more adjacent ranges into a single, larger range. It also means that when you remove a range, you may need to split one or more ranges that are in the list. In other words, you can not assume that ranges that are being inserted or removed correspond exactly to ranges already in the list.

Segmentation Fault Signal Handler

When there is a segmentation fault, the operating system will deliver a `SIGSEGV` signal. The two main reasons why a segmentation fault can occur are that the program tried to access a memory address that is not

part of the process' address space or the program tried to access a memory address whose protection state does not allow the type of access.

You will need to distinguish between the two types of accesses in your handler. If the access is a protection fault (`SEGV_ACCESS`) and the address belongs to the process' address space, then this is a protection fault, whereby the application was trying to write to memory that you designated as read-only for checkpointing purposes. If that is the case, your handler should do two things:

1. Record the page as being updated so that it will be saved in the next checkpoint.
2. Use `mprotect(2)` to change the permissions of the page to read and write so that the program can continue.

Note that you only want to perform these actions at the granularity of a *single page*. You do not want to save more data in a checkpoint than necessary and the operating system allows you to set permissions on a per-page basis. So, you should only update the protection on the page that contains the faulting address (and record only that page as having been updated).

If the fault is not a protection fault or the faulting address is not an address that is part of the process' address mapping, then this is a true error in the program. In this case, there is nothing you can or should do to fix the situation, and the program should crash with a segmentation violation.

Checkpoint File Formats

Checkpoints consist of a single ".map" file and a numbered sequence of data files ".data.0", ".data.1", etc. All files have the same prefix for the filename, as specified by the `CKPT_PREFIX` environment variable.

Your checkpoint files **must** conform **exactly** to the formats described below. The files must contain all of the specified data and may not contain any extra data or empty space.

Map File

The map file contains a header that looks as follows:

```
struct {
        int         period;      // The period between checkpoints
        uint64_t    epoch;       // The epoch of this checkpoint
        uintptr_t   brk;         // The break pointer
        uintptr_t   stack;       // The stack pointer
        int         range_cnt;   // The number of active address ranges
        int         file_cnt;    // The number of open files
        int         maxfd;       // The largest open file descriptor
        ucontext_t  ctx;         // The application's execution context
};
```

After this structure, there should be data for `range_cnt` ranges, each of which looks as follows:

```
struct {
        uintptr_t start;         // Starting memory address of this range
        uintptr_t end;           // Ending memory address of this range
        uint32_t  crc32;         // CRC32 checksum of the data in this range
        bool      write_protect; // True if this range should be write protected
};
```

The size of the range (`end - start`) must be a multiple of the page size, which can be found with `sysconf(_SC_PAGE_SIZE)`. You may assume that the page size will be the same on the system when you restart from a checkpoint as it was when the checkpoint was taken.

After the ranges, there should be data for `file_cnt` open files, each of which looks as follows:

```
struct {
        char  name[MAXFILENAME]; // The name of the file
        int   flags;             // The flags they used to open the file
        int   fd;                // The open file's file descriptor
        off_t offset;            // The current offset in the file
};
```

Data Files

Each data file has a sequence of range data which contains all of the memory that was modified since the last checkpoint. The data file for the 0^{th} epoch will contain all of the write-enabled data within the application. The data file right after a restart will also contain all of the write-enabled data within the application. These complete memory checkpoints are required, as other libraries (particularly `libc`) might have constructors that run before your checkpoint constructor, so they could modify memory before the checkpointing system is able to write protect their memory. To simplify the design of your system, you should simply checkpoint the entire writable application state in those two circumstances.

Each range looks as follows:

```
struct {
        uintptr_t start;   // Starting memory address of this range
        uintptr_t end;     // Ending memory address of this range
        uint32_t  crc32;   // Not used
        bool      write_protect; // Not used
        byte      data[];  // Data in this range
};
```

The size of the range (`end - start`) must be a multiple of the page size. Furthermore, the size of the `data` array should be `end - start`.

Address Space Layout Randomization

Modern systems randomize the address space layout to provide some protection against attacks using a technique called address space layout randomization (ASLR). ASLR randomly selects the start of the stack, heap, libraries, and other components to prevent attackers from guessing the location of vulnerable data using repeatable addresses. This is not a perfect security measure, rather, it adds one more layer of complexity to thwart attackers.

Unfortunately, ASLR interferes with the checkpointing mechanism you will implement. In particular, if the address space layout of the restarting process is different than the address space layout of the process that was checkpointed, it would be difficult to successfully restart the application. While you can place the stack, heap, and other elements in their new locations, all pointers on the stack and heap would then point to the old locations.

Rather than try to fix all of the pointers in the application (which is effectively an impossible task in C), we will instead turn ASLR off for this assignment. In Linux, this is done by running a program as follows:

```
$ setarch --verbose --addr-no-randomize <program>
```

To set environment variables, you will need to use the `env(1)` program. For example, to run the program with `CKPT_PREFIX` and `CKPT_PERIOD` set, you would run the program as follows:

```
$ setarch --addr-no-randomize env CKPT_PREFIX=ckpt CKPT_PERIOD=0 <program>
```

Note that `--verbose` is optional, it just provides more information on what `setarch` is doing, which may be useful for debugging. The provided `Makefile` uses `setarch` for you. If you do not use the `Makefile` because you want to run a different command, for example, make sure that you use `setarch` to run your program or checkpointing will not work.

Restarting from a Checkpoint

To restart from a checkpoint, the system must first read the ".map" file and restore the checkpointing system's state. After that, the system must restore the files that were open at the time of the checkpoint, including the offset within those files, and the entire application memory at the time of the checkpoint.

To restore the open files, the system must reopen each file using the same flags as were originally used to open that file. Then, the current position in the file must be restored to where it was at the time of the checkpoint. Note that for files that were being written, any additional data beyond that position must be deleted. In order for this to work, files can only be *appended* to. If you allow arbitrary writes into the middle of files, then the program can modify the middle of a file after a checkpoint, forever losing the file's state at the time of the checkpoint. Therefore, you must wrap the `open(2)` system call and ensure that if a file is being opened for writing that it is always using "append" mode. For simplicity, your system does not need to support `fopen(3)`.

To restore the state of the memory, your system must work *backwards* through the ".data" files, starting at the epoch of the checkpoint which was stored in the map file. You must go backwards as that is the most recent state of the memory. Note also that a range (or part of a range) within a data file may have been de-allocated by the program after the checkpoint, so it is not necessarily still valid. So, when you see a page in the data file that you have not yet seen, you should do the following:

1. Check if you have already restored that page from a later checkpoint. If so, do nothing.
2. Check if the page is still part of a valid data range, as listed in the map file. If not, do nothing.
3. Copy the page to its correct location in memory.

Once you have restored the memory, you should first verify that the CRC32 value for each writable range matches the value recorded in the map file, and then increment the epoch, as future checkpoints should be in subsequent epochs.

Note that on restore, you are not write protecting the memory for the reasons listed above. At the first checkpoint after the restore, all memory will be saved. At that point, all writable memory should be write protected for future checkpoints.

Finally, you need to restart the program by resuming the execution context that was stored in the map file.

Checkpointing Library

Your checkpointing system will be compiled as a library that can be linked with other programs. To use the checkpointing system, an application must be linked with your library and the appropriate environment variables must be set to control its behavior.

You can view the `Makefile` to see how this is done. Effectively, to compile an application to use your checkpointing library, you would do so as follows:

```
$ gcc -o app app.o -L. -lcheckpoint
```

The `-L` tells `gcc` to look for libraries in the current directory and `-lcheckpoint` tells it to link with the library `libcheckpoint.so`. Remember, to run the program you **must** use `setarch`, as previously described, in order to disable ASLR. With `setarch`, you will need to use `env` to set the appropriate environment variables. You can look at the `Makefile` to see how this is done.

Getting Started

Please visit the web page at `https://classroom.github.com/a/jzYrl7IU`. (If you are copying this URL, do not include the period at the end of the sentence, as it is not part of the URL.) This page should say "RICE-COMP321-S25-Classroom" and "Accept the assignment — Checkpointing". Moreover, it should have a green button labeled "Accept this assignment"[1]. Please accept the assignment.

Upon accepting the assignment, you will be redirected to another web page. This page will confirm that you have accepted the assignment, and it will eventually (after you click refresh) provide you with a link to your personal repository for the assignment. Click this link to go to your personal repository.

The web page for your personal repository has a green button labeled "Code". Click this button. You should now see a text field with a URL. Copy or remember this URL.

Login to the `CLEAR9` system if you have not already done so. Type the following:

`git clone` [Copy the URL for your repo here]

You will be prompted for your `github` username and password.

Once the clone operation is complete, you will have a directory named

`checkpointing-`[YOUR `github` ID]

Please `cd` into this directory, and do the following:

- Type a header comment at the top of `checkpoint.c` with your name and NetID.
- Type the command `make` to compile and link a skeleton checkpointing library and some simple programs you can run to test it.

The provided code includes the following source files for the `checkpointing` library:

1. `checkpoint.c/h`
2. `checkpoint_internal.h`
3. `mm.c/h`

[1] You may have to login to GitHub to see this page. If so, you will be prompted for your GitHub username and password.

4. `rio.c/h`
5. `sio.c/h`
6. `file.c`

The `checkpoint.c` file is the heart of the checkpointing library. It implements all of the checkpointing functionality. You will need to complete this file to create a functioning checkpointing library.

The `mm.c` file is a complete malloc implementation (including `mm_init`, `mm_malloc`, `mm_free`, and `mm_realloc`). It is very slightly modified from the provided code from the malloc assignment. You should *always* use this allocator within your checkpointing system. Otherwise, you will be checkpointing the checkpointer which will cause you serious problems.

The `sio.c` file provides signal-safe printing functions. You should use *always* use these functions when printing from your checkpointing system, whether you are in a signal handler or not. Otherwise, you will mix your checkpointing system's I/O with the application's I/O and again, potentially cause yourself serious problems. The `rio.c` file provides robust I/O functions that are mostly useful for the `rio_readlineb` function when reading lines from the process address space file.

Finally, the `file.c` file contains wrappers around `open(2)`, `close(2)`, and `lseek(2)`. These will allow you to track the files that are open in the system and prevent unsupported behaviors.

Checking Your Work

First and foremost, you should make sure to take advantage of the `CKPT_DEBUG` environment variable. When it is set, you can use the `sio` functions to print out debugging information from your checkpointing system. As you develop, feel free to modify the `Makefile` to assign values to the `CKPT_DEBUG` variable and use those values to control which debugging printouts are active. We will always run your code *without* defining `CKPT_DEBUG`, so you may use this however you see fit.

Second, you should develop your checkpointing system *incrementally*. We have provided a reference solution that you can run as described below. You should first get checkpointing working and then confirm you can restart from your checkpoint using the reference library. You should then work on restart and confirm you can restart with your library from a checkpoint generated by the reference library. Then, and only then, should you try to create a checkpoint with your library that can be restarted with your library.

We have provided you with some test programs to help you check your work (all of which should be compiled and run by using `make`). You can run any of the tests with manual checkpointing or periodic checkpointing and you can restart them using the following commands, where `<testname>` is the name of the test.

- `make run-periodic-<testname>`: This builds and runs the test in `test_<testname>.c` using periodic checkpointing with a period of 1 second.
- `make run-manual-<testname>`: This builds and runs the test in `test_<testname>.c` using explicit calls to the `checkpoint` function.
- `make restart-<testname>`: This restarts the test in `test_<testname>.c` from the most recent checkpoint stored on disk.

The C file for each test is named `test_<testname>.c`. You can also run the tests linked with the reference library using `make runref-periodic-<testname>`, `make runref-manual-<testname>`, and `make restrartref-<testname>`. These work

exactly the same, except the tests use the reference library. You can checkpoint using your library (for example, by running `make run-manual-<testname>`) and then restart using the reference library (for example, by then running `make restartref-<testname>`, with the same `<testname>`), and vice versa.

To test restarting, you need to run the test (using either manual or periodic checkpointing) and then hit `Ctrl-C` to stop the program after a checkpoint, but before it completes. You can then try to restart the program from that checkpoint. Do *not* run make clean or otherwise modify with the checkpoint files between hitting `Ctrl-C` and restarting.

Many of these programs are too short to work with periodic checkpointing with a period of 1 second. They therefore have calls to `delay` in them to slow them down. The `delay` function is simply a spin loop that does nothing but waste CPU time, allowing a checkpoint to be taken. This also allows you to use `Ctrl-C` to stop the program after such a checkpoint so you can test restarting from that checkpoint.

The provided tests are as follows:

1. readonly_write

   This tests that a write to a readonly region causes the program to terminate with a segmentation violation.

   Run `make run-manual-readonly_write`, and verify that there is a segmentation fault. This test is too short to induce any checkpoints.

2. unmapped_write

   This tests that a write to an unmapped region causes the program to terminate with a segmentation violation.

   Run `make run-manual-unmapped_write`, and verify that there is a segmentation fault. This test is too short to induce any checkpoints.

3. fib

   This tests that the checkpointer properly checkpoints the stack.

   When execution completes (directly or after a checkpoint/restart), the output should be `fib(50) = 12586269025`.

4. fib2

   This tests that the checkpointer properly checkpoints the stack when it grows.

   When execution completes (directly or after a checkpoint/restart), the output should be `fib(50) = 12586269025`.

5. heap

   This tests that the checkpointer properly checkpoints the heap.

   When execution completes (directly or after a checkpoint/restart) the output should be:

   ```
   start array: 0123456789
   end array:   5678901234
   ```

   The reasoning for these is that 0-9 should be the first 10 digits in this program, and that it counts to 9 like this up until $2^{28} - 2$ (null term and starting 0) = 268435454, which ends with a four.

6. read_file

   This tests that the checkpointer properly tracks a user's file opened for reading.

   When execution completes (directly or after a checkpoint/restart) the output should be:

   ```
   before checkpoint
   after checkpoint
   ```

   A manual checkpoint and delay occurs between these statements (which are the result of reading the file `test_file_1.txt`). If you stop the program after the checkpoint (during the delay). It should *only* print "before checkpoint" before you stop the program and "after checkpoint" on a restart.

7. write_file

   This tests that the checkpointer properly tracks a user's file opened for writing.

   Important: Make sure you do not have a non-empty file called `write_file_1.txt` in your directory.

   When execution completes (directly or after a checkpoint/restart) the file `write_file_1.txt` should contain the following:

   ```
   before checkpoint
   after checkpoint
   ```

   A manual checkpoint and delay occurs between writing these statements to the file. If you stop the program after the checkpoint (during the delay). It should *only* write "before checkpoint" to the file and add "after checkpoint" on a restart.

8. multi_file

   This tests that the checkpointer can properly track files concurrently being read and written (different files).

   Important: Make sure you do not have a non-empty file called `write_file_2.txt` in your directory.

   When execution completes (directly or after a checkpoint/restart) the file `write_file_2.txt` should contain the following:

   ```
   before checkpoint
   after checkpoint
   ```

   A manual checkpoint and delay occurs between writing these statements (which are the result of reading the file `test_file_1.txt`) to the file. If you stop the program after the checkpoint (during the delay). It should *only* write "before checkpoint" to the file and add "after checkpoint" on a restart.

9. rle

   This tests that the checkpointer can properly track files concurrently being read and written along with concurrent computations on the contents of the files. This reads the `data.txt` file and produces a `data.rle` using run-length encoding to compress the file.

   Important: Make sure you do not have a non-empty file called `data.rle` in your directory when you run this test.

   When execution completes (directly or after a checkpoint/restart) the contents of the `data.rle` file should match the contents of the `correctdata.rle` file. You can check this by running `diff data.rle correctdata.rle`.

Notes

- Be sure to familiarize yourself with the provided code before starting.
- Develop your code incrementally. Do not try to make everything work all at once without testing anything. This is a recipe for failure.
- The signal handlers must use an alternate stack that is not write protected. If they did not, when you receive a `SIGSEGV`, the signal handler would be processing the protection fault on the application's stack. If the signal handler were to access a portion of the stact that is write protected, it would then cause another protection fault, crashing the program.
- Even though the alternate signal stack is only being used by your library, it still needs to be saved in the checkpoint. That is because when a `SIGPROF` is delivered, there is information on the stack that is needed to restart the saved execution context for the application on a restart from a checkpoint. Because it can't be write protected (see previous bullet), the signal stack needs special handling to make sure it is always checkpointed.
- Do *not* modify the provided code. This includes the structure definitions, file layout, etc. There are many nuances in checkpointing the entire process state which are handled by the provided code. The fact that you don't understand the provided code, think you can write it better, or can't get it to work with your code are not justifications for modifying it. If you do so, you are not likely to get the project working.
- The alternate signal stack must be aligned in order to properly save and restore it. You therefore must use `mmap(2)` to allocate it.
- The heap used by the provided `mm` files (which come from the provided code of the malloc assignment) should not be checkpointed. This is for internal use only within the library. You never want to write protect the region used by this internal heap, so it must be special cased within your library, to make sure that your library does not cause protection faults while it is checkpointing. This also means you should *always* use `mm_malloc` to allocate memory within your library.
- As with the heap used by your library, the stacks for any contexts that you use in your checkpointing system should be allocated using `mm_malloc`, not `mmap(2)`. These stacks do not need to be (and should not be) checkpointed. As with the signal handlers, you will also have a problem if you write protect their memory and cause protection faults within your checkpointing library.

- Whenever you print anything from your library, you should always use the `sio` functions to do so. If you use any functions from `stdio` they may cause protection faults, may interfere with the application, and will likely cause other issues with checkpointing.
- Remember, if you do anything in your library that causes a protection fault, it will not work. This has been said multiple times in this document, but it is important. Think very carefully about what functions you can use in your code.

Turning in Your Assignment

To turn in your code, you ***must*** use `git push` to copy your work to the GitHub remote repository. We will *only* look at the last version that you `push`ed before the deadline. As a precaution against accidental loss of your code, we encourage you to `push` periodically. Please note, the *only* file that you need to turn in is `checkpoint.c`. **You should not modify or add any other files in your repository.** In other words, this is the only file on which you should ever perform `git add`. **Do not ignore this instruction.**

For grading your submission, we will use the `Makefile` that was originally provided to compile your code. Therefore, your code should not rely on any modifications to the `Makefile` for correct compilation.

As a sanity check, you should use your web browser to visit your assignment repo on GitHub. Make sure that what you see in the browser is consistent what you think you have pushed.

Evaluation

The project will be graded as follows:

- Style: 20%
- Correctness: 80%

Your solution will be tested for correctness on a CLEAR9 machine.