Persistent User Programs - A Checkpoint-Restart Framework

Team `SIGSYS(31)`: *Matthew Zhong and Savannah Amos*

Submission Date: December 14, 2020

Table of Contents

Abstract

Persistent user programs are widely desired, particularly for software which performs complex or lengthy computations. Introducing persistence in programs, however, raises several considerable challenges. Tracing execution is inherently a conflicting problem which requires a balance between usability and performance.

In this project, we **provide an application-transparent checkpoint-restart framework** which allows users to add persistence to user programs. Our system encompasses a non-volatile memory manager, memory region checkpointing semantics, thread-level checkpointing, and an after-crash resurrection procedure. **Core functionality works as intended**, but performance, while functional, could use future improvements.

The code for this project can be found at https://git.cs.vt.edu/amos3/capstone. *Please use the* ***`Final-Framework-Product`*** *tag and* ***not*** *the top commit in the* `master` *branch when evaluating this product.*

Introduction and Background

Under a traditional programming model, users design applications which run in a volatile state, computing their results and saving them at very infrequent intervals. *If a program is terminated mid-execution, users usually must restart the program from the very beginning.* To combat this, researchers have experimented with adding persistence to user programs. We define persistence as a program's ability to remember its current state of execution, even if terminated abruptly.

In our project proposal, we identified several current state-of-the-art solutions which add persistence to user programs. *Predominantly, most solutions which currently exist either use a checkpoint-restart strategy or non-volatile memory systems to implement user-level persistence.*

Checkpoint-restart is a strategy in which a program creates snapshots of its current execution state at selected checkpoints. *Upon abrupt termination, when the user restarts program execution, the program should continue running from the most recent checkpoint saved.* Examples of solutions using checkpoint-restart include NVThreads [1] and C/R for MPI [2].

Non-volatile memory management is a technology which allows programs to allocate memory which persists after the program has finished execution. While most developers are familiar with a standard filesystem in an operating system, researchers have also investigated a variety of other techniques, including EasyCrash [3] and NVMReconstruction [4]. Both of these solutions provide systems which provide programmatic data with persistence without requiring the developer to manually manage how their data is saved.

In our original research, we noticed that one major issue with most solutions enabling persistent programs was simply that many solutions were **difficult to understand and use**. From this, our main objective was to design and implement an **application-transparent, simple to use checkpoint-restart** framework.

Design and Implementation

Fundamentally, modern computers represent execution of an application using a call stack and a heap. *The technology behind our checkpoint-restart framework involves saving the call stack and heap into a non-volatile file upon checkpointing, followed by loading them back into volatile RAM upon restoration.* **Every other design decision stems from this strategy.**

The first component we built in this system is a **non-volatile memory manager**. Traditionally, application-level programs dynamically allocate memory with `malloc()` and `free()`. We needed functions which mimic their behavior but also persist their allocations across several executions.

Common approaches to solve this problem include:

1.) **Make the user specify when to write-through memory to non-volatile storage.** This approach is simpler and imposes less overhead, but is harder for users to understand how to use. Users would need to dramatically change their programming paradigms to suit our system, making their code less portable and harder to write.

2.) **Automatically detect and update which sections of memory were changed.** This approach is more complicated for us to implement and requires its own background service, but users use our versions of `malloc()` and `free()` in the way they originally should, with no change in semantics.

We ultimately chose the **second approach**. We can implement an application-silent memory manager with strategic use of `mmap()` and the `userfaultfd` system call. We can track when anything attempts to access sections of memory allocated by `mmap()` by registering them through `ioctl()`. After registration, we can monitor any memory accesses at a page-level granularity. This service is controlled on its own dedicated thread, since page fault resolutions block the thread which accesses a clean page. Observe the diagram below for an example.

Figure 1: Automatic Page Fault Resolution

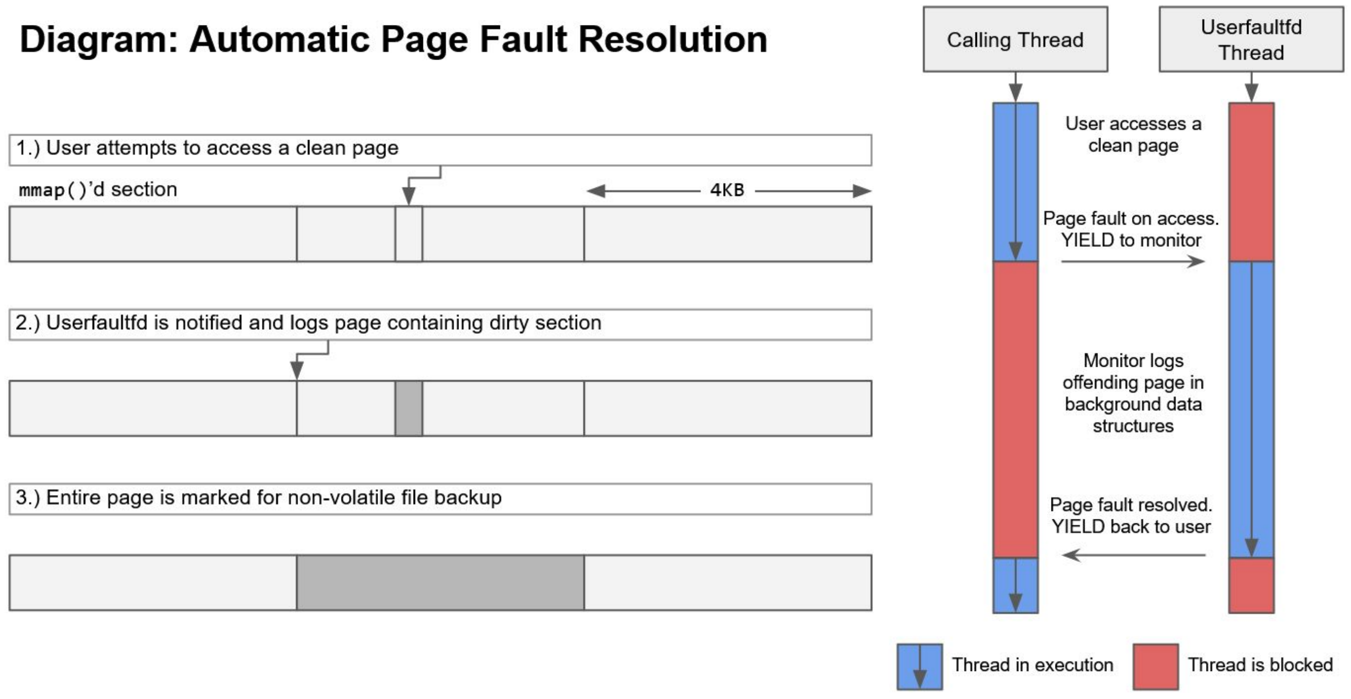
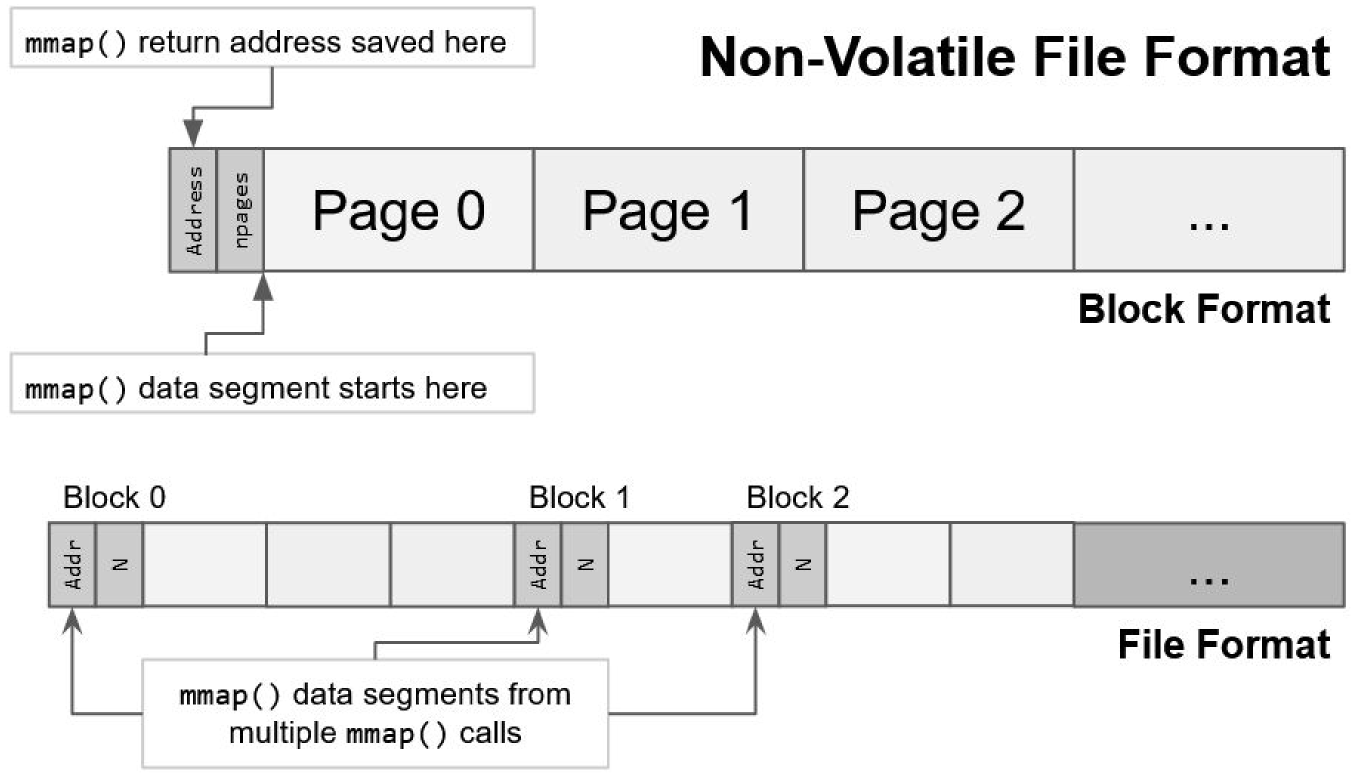


Diagram: Automatic Page Fault Resolution
1.) User attempts to access a clean page
mmap()'d section
4KB
2.) Userfaultfd is notified and logs page containing dirty section
3.) Entire page is marked for non-volatile file backup
Calling Thread
Userfaultfd Thread
User accesses a clean page
Page fault on access. YIELD to monitor
Monitor logs offending page in background data structures
Page fault resolved. YIELD back to user
Thread in execution
Thread is blocked

To manage the locations of these `mmap()`'d regions, we use block segments in a non-volatile file. Each `mmap()`'d section has a one-to-one representation in a shared, non-volatile file. At the beginning of program execution, we read through this file and remap the `mmap()`'d regions into RAM. Observe the diagram below for the file data layout. *Every other data structure we designed for the non-volatile memory manager revolves around fast, random access to these block sections.*

Figure 2: Non-Volatile File Format


mmap() return address saved here
Non-Volatile File Format
Address
npages
Page 0
Page 1
Page 2
...
Block Format
mmap() data segment starts here
Block 0
Block 1
Block 2
Addr
N
...
File Format
mmap() data segments from multiple mmap() calls

Once we established this file format and system workflow for the memory manager, we immediately recognized the importance of multiple crucial indexes. These indexes include:

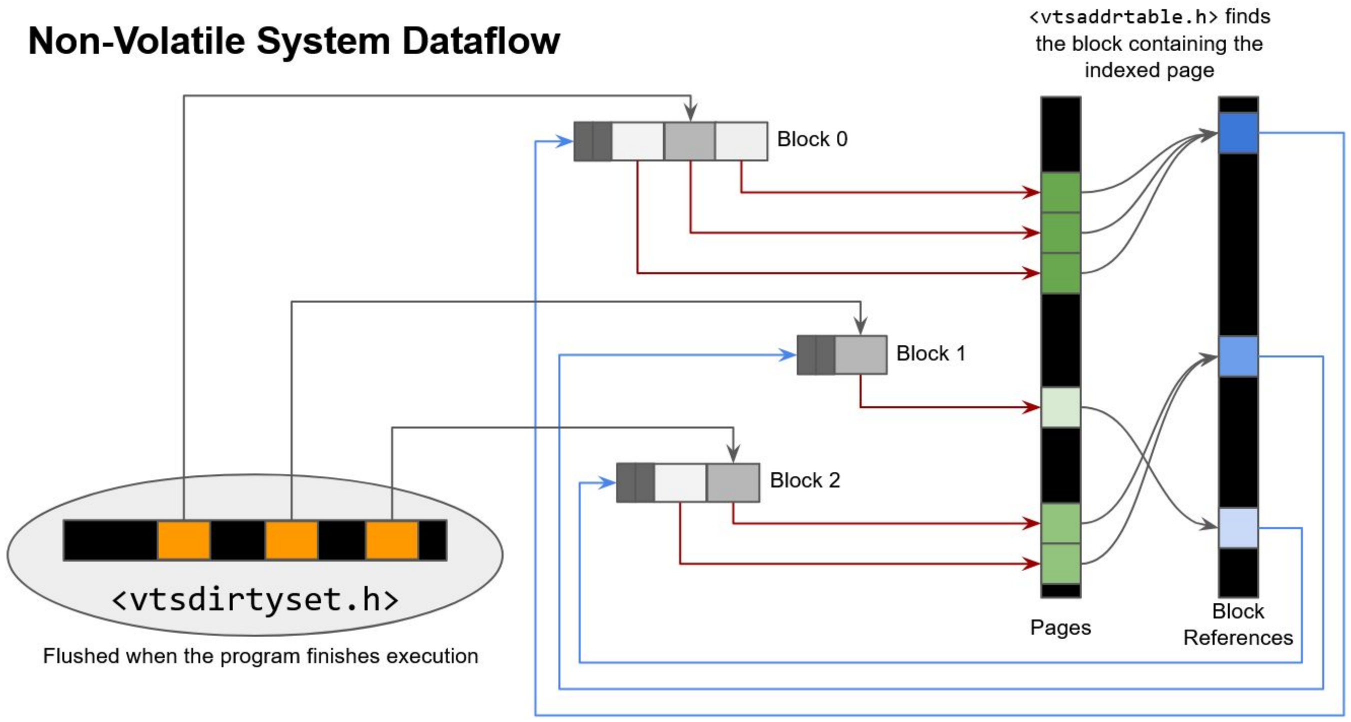
1.) **`<vds/vtsdirtyset.h>` A dirty set of addresses.** This is a hashset which holds the addresses of each location accessed which triggered a page fault. When performing page fault resolution, we log the offending addresses in this hashset.

2.) **`<util/list.h>` A list of blocks.** We used a linked list approach, but arrays would have also worked. Initialization and shutdown both require iteration through all allocated blocks, so this index facilitates this task.

3.) **`<vds/vtsaddrtable.h>` A mapping of pages to reference the block containing them.** To determine the proper file offset to which we flush dirty pages, we need the starting address of the `mmap()`'d section. Insertions to this mapping do not happen frequently, but lookups happen often, so we use a hashmap with a probing hash collision resolution policy. The hash key policy automatically converts any input address into the page address containing it, reducing the number of entries inserted per block.

With these data structures identified, our non-volatile memory manager now supports silent tracking for select `mmap()`'d sections of memory. At this point, users can allocate file-backed, persistent memory segments using the `void *nvstore_allocpage(size_t npages)` function. When the user requests for the whole heap to be flushed, the system removes addresses from the dirty set and writes their containing pages to the non-volatile file. For more information, examine the diagram below.

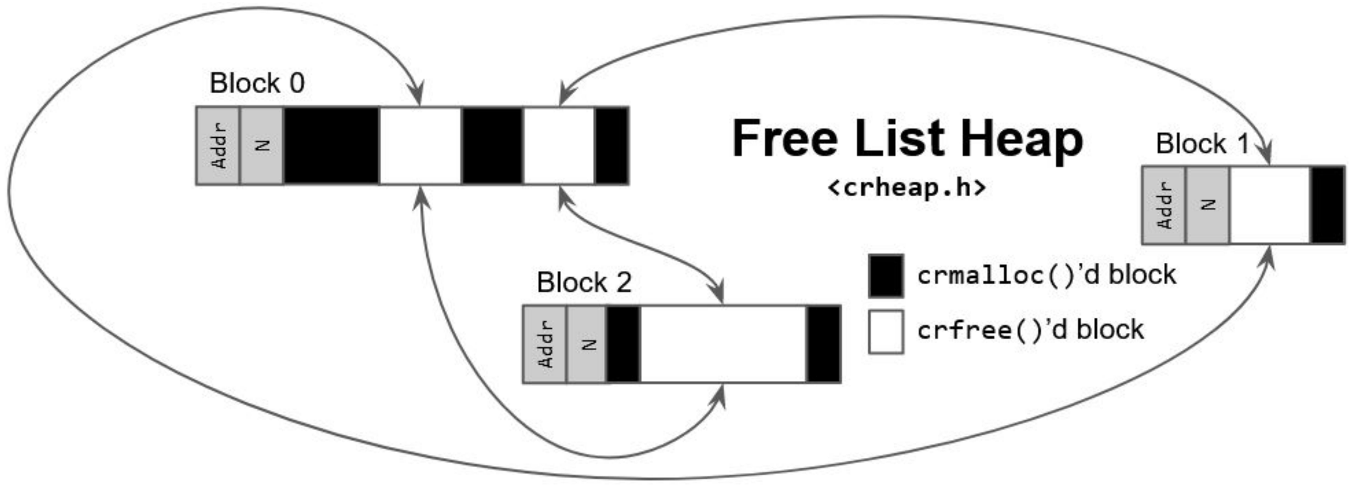
Figure 3: Non-Volatile System Dataflow



Non-Volatile System Dataflow
<vtsaddrtable.h> finds the block containing the indexed page
Block 0
Block 1
Block 2
<vtsdirtyset.h>
Flushed when the program finishes execution
Pages
Block References

With the non-volatile filesystem implementation in place, we shift our focus to tackle **user-level memory allocation requests**. Our current approach implements a freelist which manages unused regions from each block. This system coalesces adjacent free blocks in order to reduce memory usage. We opted for a free list approach as opposed to a segmented list or tree-based approach due to time constraints, but further work into this project could potentially benchmark different heap data structures for performance. Because this heap is implemented directly on top of the non-volatile system, any calls to `crmalloc()` and `crfree()` can be used as persistent memory allocation calls.

Figure 4: Free List Heap



Now that we've built a non-volatile memory manager, we can use it to implement **non-volatile threads.** At its core, implementing persistent threads requires us to first devise semantics which allocate, save, and continue execution of non-volatile call stacks. Our design uses two stacks for each thread - a main stack holding thread execution, and a recovery stack intended for restoring the execution context when the thread is resurrected.

The basic procedure we use to checkpoint and restore threads uses the following approach:

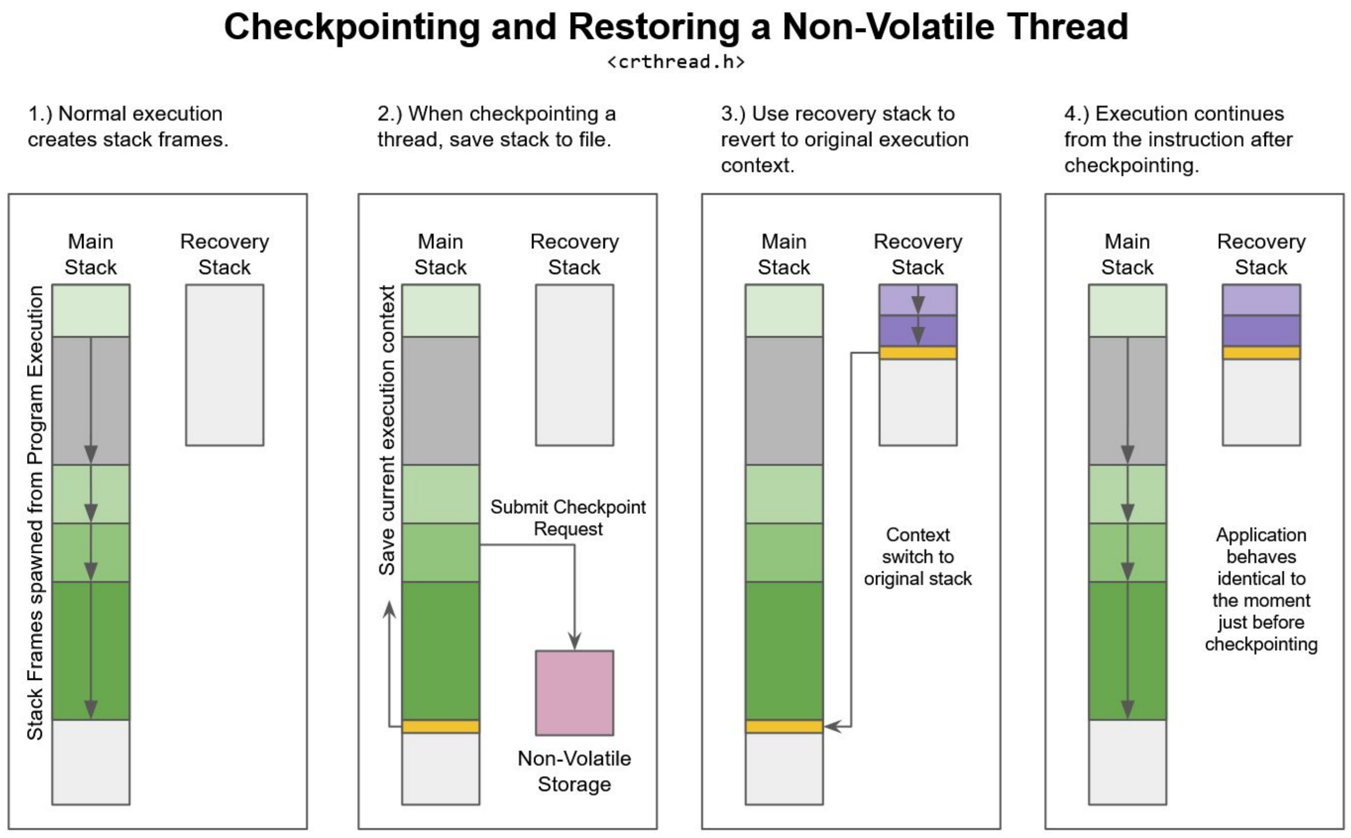
1.) **Spawn a new thread using non-volatile memory**. Use the `<pthread.h>` library to begin concurrent execution of the user-specified thread function on the main non-volatile stack. The thread should automatically create stack frames in its normal execution.

2.) **Perform a checkpoint, saving the current thread into non-volatile memory.** Because the calling thread crashes when it attempts to save its own stack, a background thread performs this operation. In doing this, we actually shutdown the `pthread_t` which backs our thread.

3.) **Perform a restoration.** Because only the execution context is saved, we create a new `pthread_t` which begins execution on a preallocated recovery stack. This thread then immediately performs a context switch to where the original thread created a checkpoint.

4.) **Continue execution as usual.** The new thread should continue on the instruction immediately after the call to a checkpoint, and all register variables should be restored.

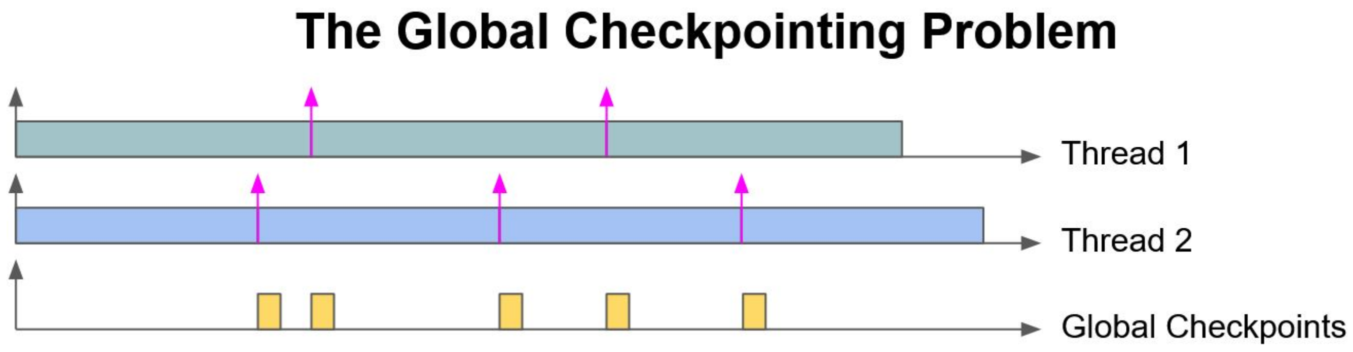
Figure 5: Checkpoint and Restoration of One Thread


Checkpointing and Restoring a Non-Volatile Thread
<crthread.h>
1.) Normal execution creates stack frames.
2.) When checkpointing a thread, save stack to file.
3.) Use recovery stack to revert to original execution context.
4.) Execution continues from the instruction after checkpointing.
Main Stack
Recovery Stack
Stack Frames spawned from Program Execution
Save current execution context
Submit Checkpoint Request
Non-Volatile Storage
Context switch to original stack
Application behaves identical to the moment just before checkpointing

In order to perform a context switch, we began with the POSIX-standard libraries `<setjmp.h>` and `<ucontext.h>`. Unfortunately, from our experience, both `setjmp()` and `setcontext()` have the same unavoidable issue - they both allocate stack frames of their own when invoked. They never worked in our project because we were specifically swapping contexts on different stacks and because we were jumping deeper into a stack, as opposed to escaping into shallower stack frames. *We ultimately created our own custom context switching mechanism in x86 assembly which did not allocate any stack frames and which used only registers to execute their algorithm.*

Implementing the threading subsystem revealed some inadequacies in how we defined our data checkpointing semantics. In a typical application, each thread modifies a specific subset of data, some of which is shared. As a result, a naive checkpoint which saves the entire application heap to a file may accidentally save data which one or more threads has yet to finish updating to a consistent state, exemplified in the diagram below.

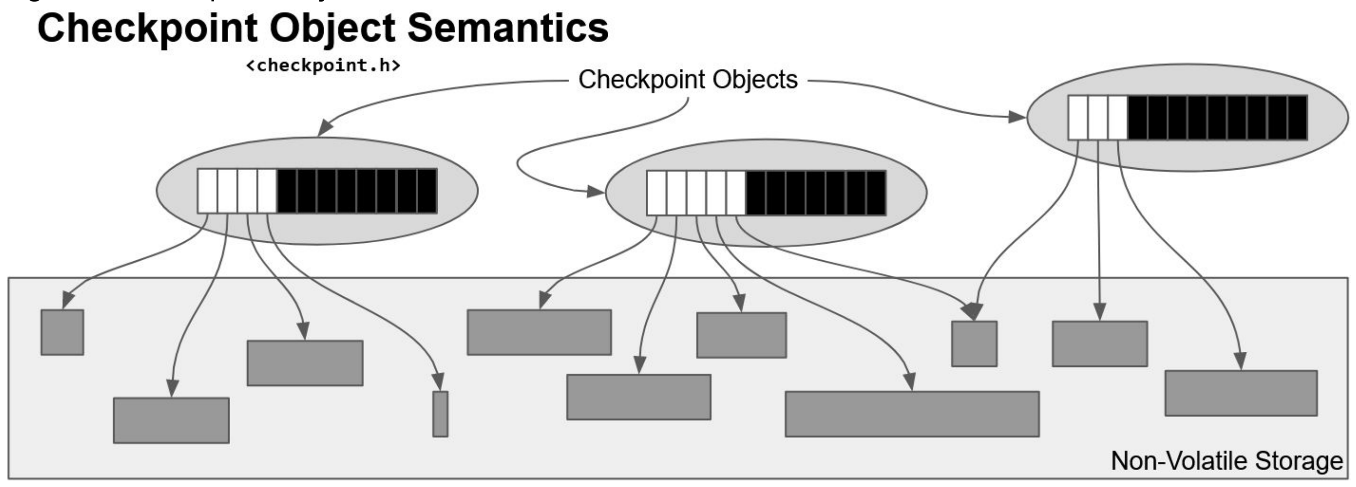
Figure 6: The Global Checkpointing Problem



Notice that a checkpoint invoked from Thread 1 would have saved Thread 2 in an inconsistent state, and that a checkpoint invoked from Thread 2 would have saved Thread 1 in an inconsistent state.

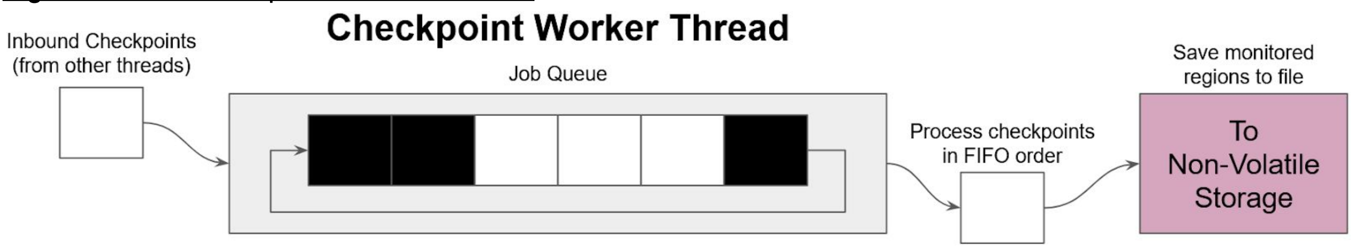
To solve this problem, we needed to revisit how a checkpoint was defined in our system. Rather than using a global checkpoint system, we instead defined new semantics for creating a checkpoint. Using checkpoint objects defined in `<checkpoint.h>`, we monitor user-defined subsets of non-volatile memory regions. The diagram below shows a simple example of this interface in action.

Figure 7: Checkpoint Object Semantics



Because of these new semantics, we also can offload the objective to save each checkpoint to a non-volatile file to a separate **checkpoint worker thread**. This thread accepts checkpoint jobs and continuously saves their monitored regions to a file. Not only does this solve the data race when two checkpoints attempt to be saved at once, this also removes some of the execution burden from the threads performing useful application computations. In the diagram below, we can see the design of this service in action.

Figure 8: The Checkpoint Worker Thread


Checkpoint Worker Thread
Inbound Checkpoints (from other threads)
Job Queue
Process checkpoints in FIFO order
Save monitored regions to file
To Non-Volatile Storage

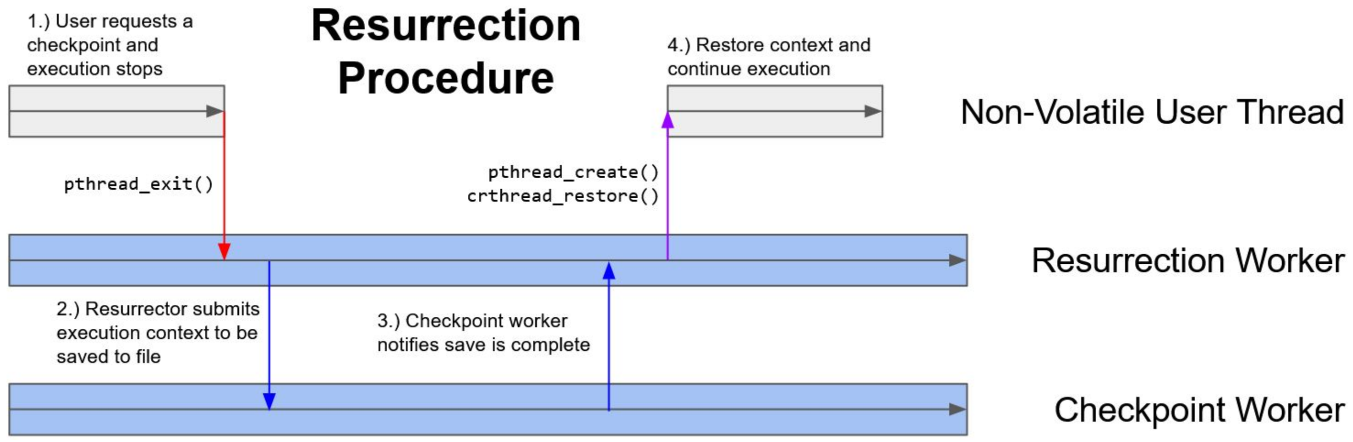
It also follows that, in the previous example, we solved the partial thread checkpoint problem by simply designating separate checkpoint objects monitoring the data for the different threads. Each thread, then, upon checkpointing, would only checkpoint its own thread-local data.

One issue that also arose in our stack saving mechanism was that attempting to copy stack data to a file while stack frames were either allocated or modified often caused data races and crashes which were impossible to trace. This essentially prevented any thread from checkpointing its own stack, meaning we needed a dedicated background service which:

1.) **Joins with the thread which requested a checkpoint.** The non-volatile thread stops execution using `pthread_exit()`.
2.) **Submits a checkpoint object which monitors the thread's current stack to be saved.** The background checkpoint saver thread handles the actual RAM-to-file save.
3.) **Resurrects the thread which was just exited.** This thread restores itself using the method as described in the paragraph discussing thread restoration.

We call this service the **resurrector thread**. Below, we present a diagram of how the individual threads interact with each other.

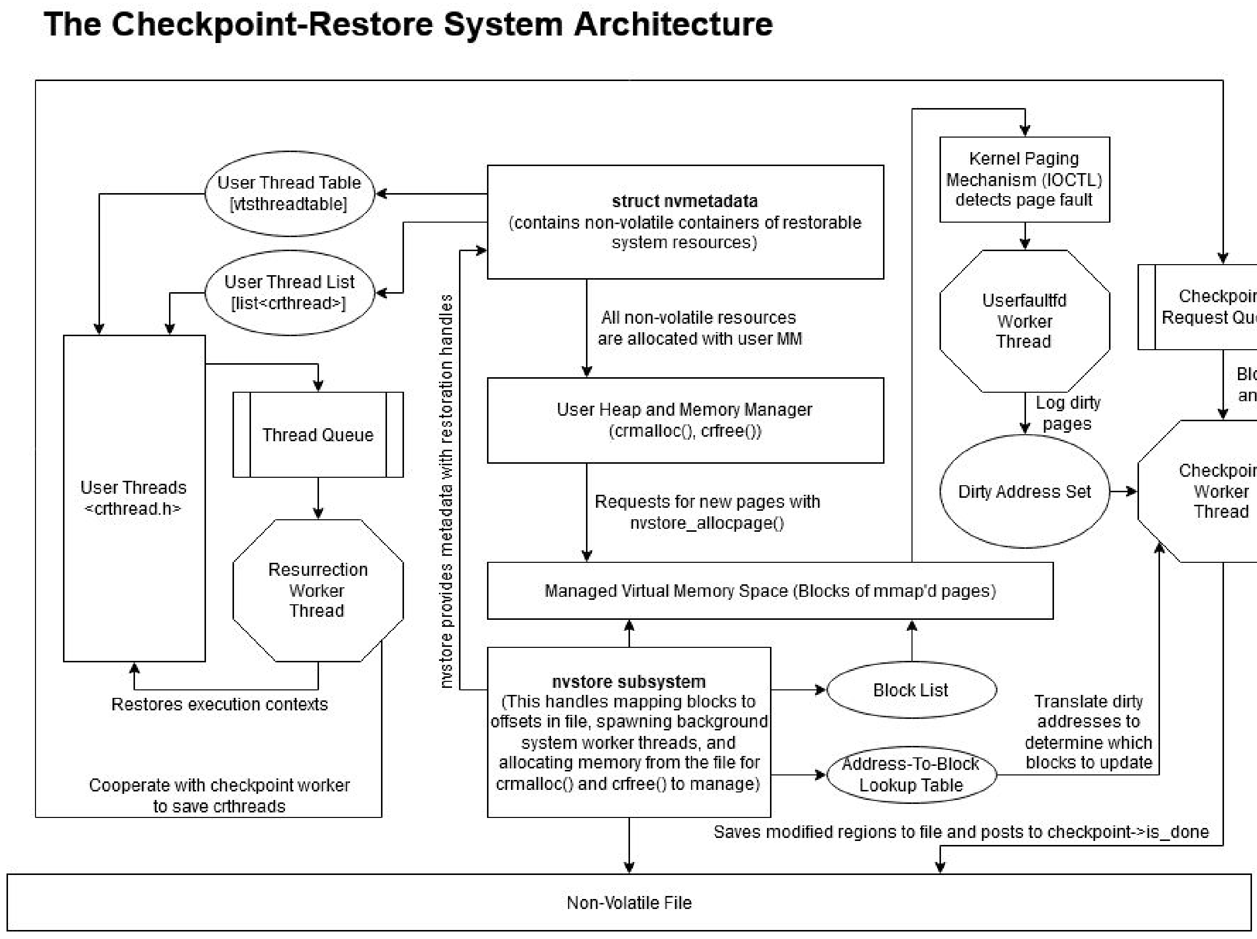
Figure 9: The Resurrection Process


Resurrection Procedure
1.) User requests a checkpoint and execution stops
pthread_exit()
4.) Restore context and continue execution
pthread_create()
crthread_restore()
Non-Volatile User Thread
Resurrection Worker
2.) Resurrector submits execution context to be saved to file
3.) Checkpoint worker notifies save is complete
Checkpoint Worker

After shutting down the main thread, the resurrector passes the thread handle into the checkpoint worker thread, which then saves the thread's stack and key metadata into a file. The thread is then respawned, performing the context switch to continue execution from where it requested for a checkpoint. *If the system is currently restarting from file (i.e. if the system crashed or terminated irregularly), the same process outlined above is used.*

We now have discussed all of the individual major components within our system, meaning we can now provide an **overview of the system as a whole**. We use a non-volatile memory manager to create threads in a non-volatile context. We manage these threads under our checkpoint semantics, addressing the global checkpointing problem. Finally, we resurrect these threads using our resurrection system. With each of these components, the **system is complete.** The program state is completely saved to a file upon each successful checkpoint. *If the user wishes to restart an application at the beginning, scrapping all checkpoint progress, they need only to delete the heap file (`*.heap`) created by this framework.*

Figure 10: The Full System


The Checkpoint-Restore System Architecture
User Thread Table [vtsthreadtable]
struct nvmetadata (contains non-volatile containers of restorable system resources)
Kernel Paging Mechanism (IOCTL) detects page fault
User Thread List [list<crthread>]
Userfaultfd Worker Thread
Checkpoint Request Queue
All non-volatile resources are allocated with user MM
nvstore provides metadata with restoration handles
Blocking Pop and Process
Log dirty pages
Thread Queue
User Heap and Memory Manager (crmalloc(), crfree())
User Threads <crthread.h>
Dirty Address Set
Checkpoint Worker Thread
Requests for new pages with nvstore_allocpage()
Resurrection Worker Thread
Managed Virtual Memory Space (Blocks of mmap'd pages)
nvstore subsystem (This handles mapping blocks to offsets in file, spawning background system worker threads, and allocating memory from the file for crmalloc() and crfree() to manage)
Block List
Restores execution contexts
Translate dirty addresses to determine which blocks to update
Address-To-Block Lookup Table
Cooperate with checkpoint worker to save crthreads
Saves modified regions to file and posts to checkpoint->is_done
Non-Volatile File

A Note about the Development and Execution Environment

Our system was developed and tested on Fedora 32 using Linux Kernel 5.6. We used the GCC and Makefile toolchain to implement our design. Unit tests, coverage tests, and some integration tests were simply Makefile targets in our build process with predefined symbols to enable certain features. We also used ***no additional libraries*** *outside of the POSIX library and the standard C library. As of now, this project only runs on a Linux OS with support for `userfaultfd`.*

Testing and Evaluation

Our testing strategy for this project consists of several major components:

1.) **Unit Tests.** We have designed our own custom unit testing interface which determines accuracy of multiple subsystems. This includes partial coverage testing.

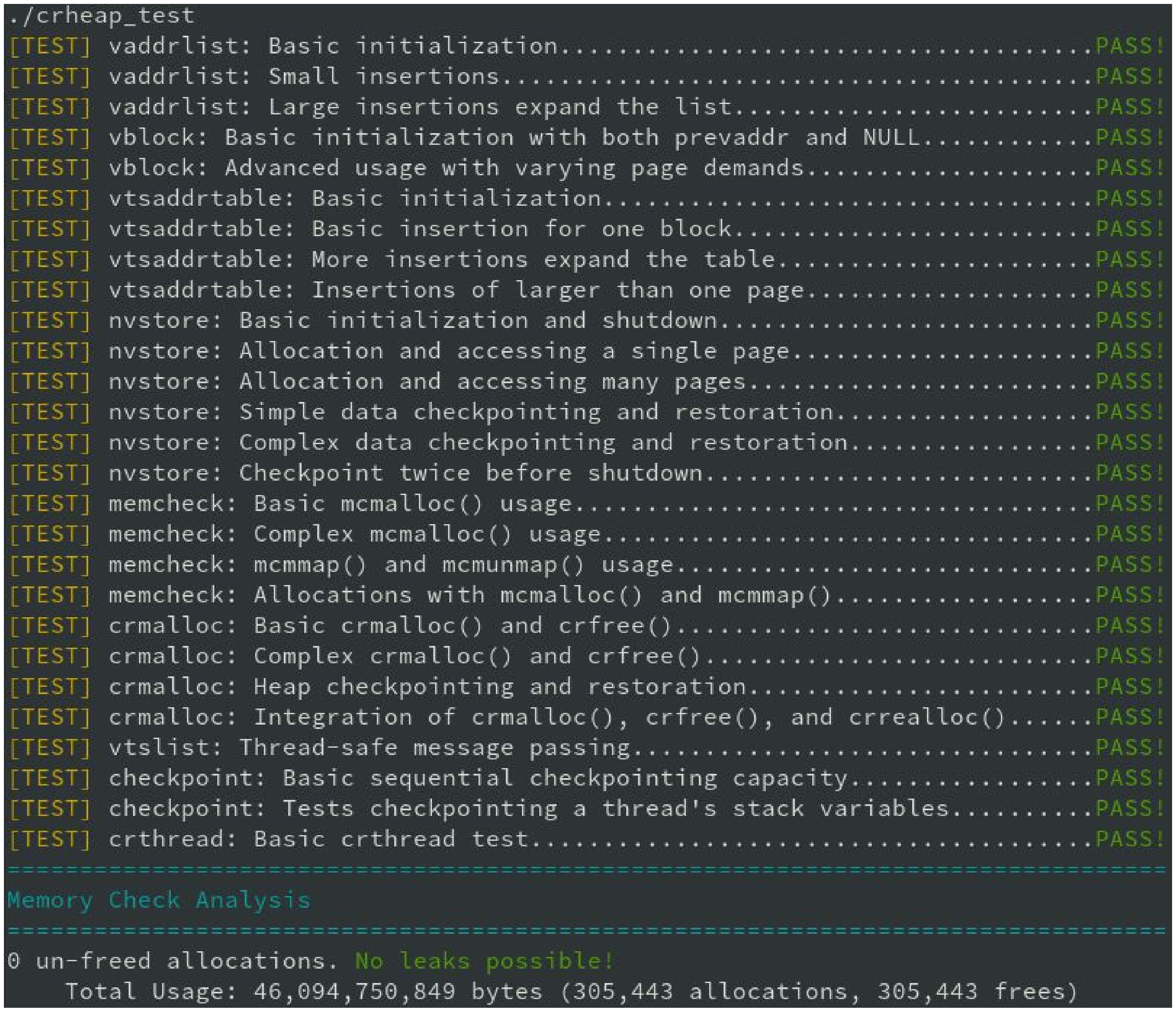
2.) **Memory Tests.** Because of flaws with Valgrind, we have also created our own memory tracers to analyze memory leaks.

3.) **Fork-Join Checkpoint Tests.** Testing checkpoint and restoration is inherently challenging because it involves intermittent termination of the whole system. We have a strategy to induce crashing and demonstrate the system resurrects as intended.

4.) **Performance Tests.** We have some simple benchmarks for frequencies of checkpointing and a demo program which executes a few interruptible algorithms.

We have written a set of comprehensive unit tests which encompass every major subcomponent of this system. These tests use `gcov` to test coverage. We can see a sample unit test run in the figure below.

Figure 11: Sample Unit Testing Execution

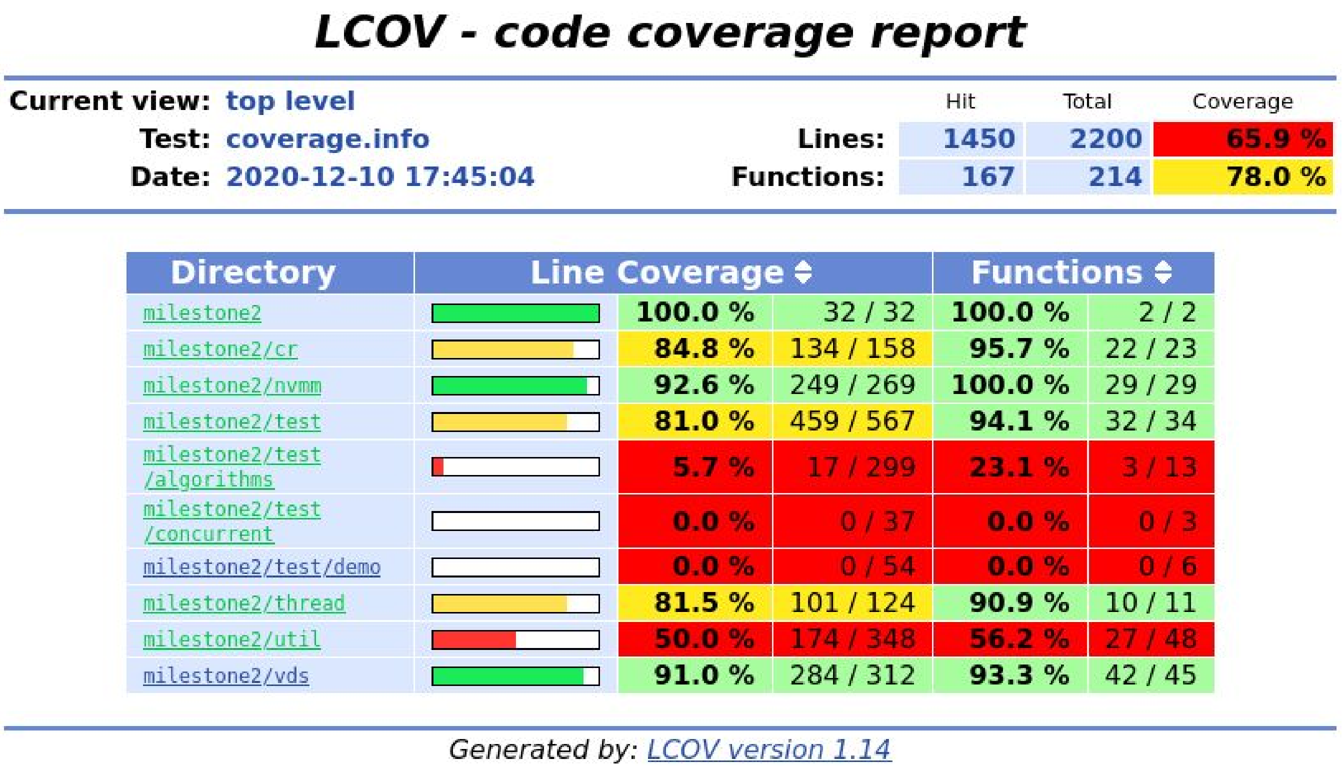


```
./crheap_test
[TEST] vaddrlist: Basic initialization....................................PASS!
[TEST] vaddrlist: Small insertions........................................PASS!
[TEST] vaddrlist: Large insertions expand the list........................PASS!
[TEST] vblock: Basic initialization with both prevaddr and NULL...........PASS!
[TEST] vblock: Advanced usage with varying page demands...................PASS!
[TEST] vtsaddrtable: Basic initialization.................................PASS!
[TEST] vtsaddrtable: Basic insertion for one block........................PASS!
[TEST] vtsaddrtable: More insertions expand the table.....................PASS!
[TEST] vtsaddrtable: Insertions of larger than one page...................PASS!
[TEST] nvstore: Basic initialization and shutdown.........................PASS!
[TEST] nvstore: Allocation and accessing a single page....................PASS!
[TEST] nvstore: Allocation and accessing many pages.......................PASS!
[TEST] nvstore: Simple data checkpointing and restoration.................PASS!
[TEST] nvstore: Complex data checkpointing and restoration................PASS!
[TEST] nvstore: Checkpoint twice before shutdown..........................PASS!
[TEST] memcheck: Basic mcmalloc() usage...................................PASS!
[TEST] memcheck: Complex mcmalloc() usage.................................PASS!
[TEST] memcheck: mcmmap() and mcmunmap() usage............................PASS!
[TEST] memcheck: Allocations with mcmalloc() and mcmmap().................PASS!
[TEST] crmalloc: Basic crmalloc() and crfree()............................PASS!
[TEST] crmalloc: Complex crmalloc() and crfree()..........................PASS!
[TEST] crmalloc: Heap checkpointing and restoration.......................PASS!
[TEST] crmalloc: Integration of crmalloc(), crfree(), and crrealloc()......PASS!
[TEST] vtslist: Thread-safe message passing...............................PASS!
[TEST] checkpoint: Basic sequential checkpointing capacity................PASS!
[TEST] checkpoint: Tests checkpointing a thread's stack variables.........PASS!
[TEST] crthread: Basic crthread test......................................PASS!
================================================================================
Memory Check Analysis
================================================================================
0 un-freed allocations. No leaks possible!
    Total Usage: 46,094,750,849 bytes (305,443 allocations, 305,443 frees)
```

You may notice that we have **a simple memory leak analysis tool** at the end of this suite of unit tests. Originally, we planned to simply use Valgrind to check for memory leaks. However, because Valgrind does not currently have support for the `userfaultfd` syscall, we needed to create our own custom memory checker. Our solution was to simply wrap `malloc()`, `free()`, and `mmap()` calls, using a simple chained hashmap to track allocations. In this way, we can verify we have patched all memory leaks in our system without the use of Valgrind.

As stated above, we use `gcov` for coverage testing. Below is a screenshot of our code coverage statistics. While it may seem that our testing coverage is lower than expected, this is mostly because gcov also attempts to track coverage for code containing unit tests. When disregarding any code in the `/test/*` folders, our code coverage is at an acceptable level, **ranging between 80 and 90 percent coverage.**

Figure 12: gcov Statistics



LCOV - code coverage report

Current view: top level
Test: coverage.info
Date: 2020-12-10 17:45:04

	Hit	Total	Coverage
Lines:	1450	2200	65.9 %
Functions:	167	214	78.0 %

Directory	Line Coverage			Functions	
milestone2		100.0 %	32 / 32	100.0 %	2 / 2
milestone2/cr		84.8 %	134 / 158	95.7 %	22 / 23
milestone2/nvmm		92.6 %	249 / 269	100.0 %	29 / 29
milestone2/test		81.0 %	459 / 567	94.1 %	32 / 34
milestone2/test/algorithms		5.7 %	17 / 299	23.1 %	3 / 13
milestone2/test/concurrent		0.0 %	0 / 37	0.0 %	0 / 3
milestone2/test/demo		0.0 %	0 / 54	0.0 %	0 / 6
milestone2/thread		81.5 %	101 / 124	90.9 %	10 / 11
milestone2/util		50.0 %	174 / 348	56.2 %	27 / 48
milestone2/vds		91.0 %	284 / 312	93.3 %	42 / 45

Generated by: LCOV version 1.14

While we do have a robust unit testing strategy, some features of this system do not lend themselves to easy unit testing. Since a unit test is just a function which is intended to run a targeted segment of code and check for its output, the only thing a unit test can really verify is whether results of a certain function are accurate. **Because of this, testing the system's ability to restore itself after a crash is almost impossible through unit testing alone.**

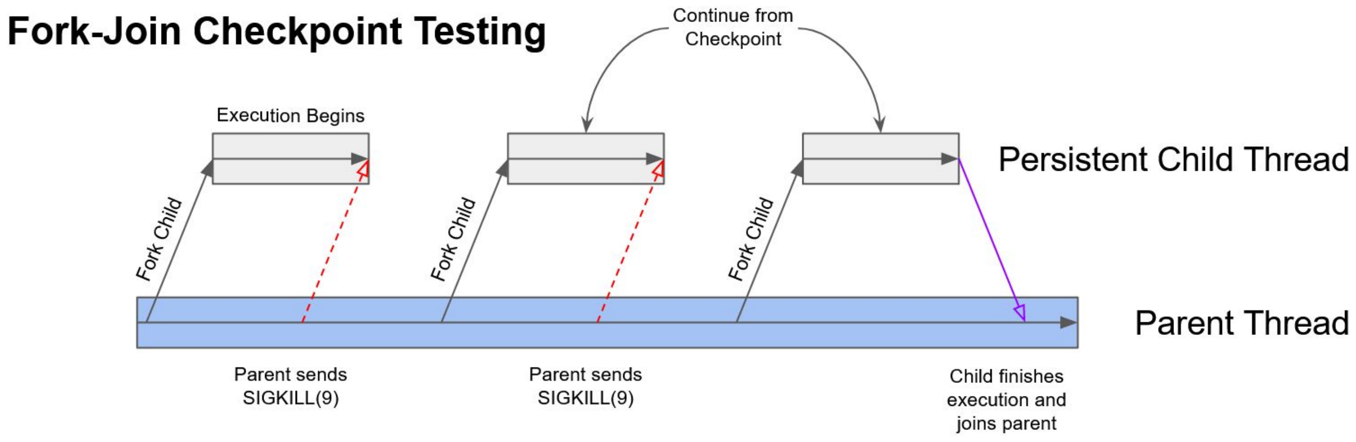
To mitigate this issue, we introduce a system of **fork-join, multi-process tests**. These tests essentially execute the following algorithm:

1.) **Fork off a child process.** The child process begins executing a toy program in a persistent state which is written with our checkpoint-restore library. *We are specifically testing whether this child process can be killed and restored correctly*, as per the objective of this framework.

2.) **Wait a few seconds and then send `SIGKILL(9)` to the child.** Our crash simulation strictly involves using `SIGKILL(9)`, since it forces immediate termination of the child program.
3.) **Fork a new child which executes the same program in (1).** Ideally, this child should continue at the last checkpoint before the previous child was killed.
4.) **Continue this procedure** of waiting a few seconds, then killing and re-forking the child with the program under test until the child returns.

For a more detailed visualization of how the threads are running, consult the diagram below.

Figure 13: Fork-Join Checkpoint Testing



Now that we have verified the functionality of all critical portions of our system, we perform some basic performance tests. Due to time constraints, we mostly resorted to testing checkpoint frequencies. To formulate a test which was composed of a subproblem which we could checkpoint at multiple intervals, we inserted checkpoints at multiple phases of a recursive `mergesort` algorithm. The mergesort subproblem into which we place checkpoints follows this pseudocode:

Figure 14: `Mergesort` Algorithm with Subproblem Checkpointing

```
void Mergesort(array, low, high)
{
    mid = (low + high) / 2;
    checkpoint_this_thread();         # CHECKPOINT_4

    Mergesort(array, low, mid);
    checkpoint_this_thread();         # CHECKPOINT_3

    Mergesort(array, mid + 1, high);
    checkpoint_this_thread();         # CHECKPOINT_2

    Merge(array, low, mid, high);
    checkpoint_this_thread();         # CHECKPOINT_1
}
```

To increase the frequency of checkpoints, we insert a select number of checkpoints after select steps. We test this algorithm under an array of `8192` integers. We outline some examples for where we activate checkpoints for various trials of this test case:

- For **one (1)** checkpoint per subproblem, we only request a checkpoint at `CHECKPOINT_1`.
- For **two (2)** checkpoints per subproblem, we request a checkpoint at `CHECKPOINT_2` **and** `CHECKPOINT_1`.
- For **three (3)** checkpoints per subproblem, we request a checkpoint at `CHECKPOINT_3`, `CHECKPOINT_2`, **and** `CHECKPOINT_1`.
- For **four (4)** checkpoints per subproblem, we request a checkpoint at all locations: `CHECKPOINT_4`, `CHECKPOINT_3`, `CHECKPOINT_2`, **and** `CHECKPOINT_1`.

Using the `time` command in `bash`, we ran each configuration of number of checkpoints for five trials per configuration. In the graphs below, we can observe the effects of frequent checkpointing. Note that we include both user and kernel execution times - both seem to scale linearly, meaning that the strongest contributor to the long execution times is largely due to algorithmic and design complexity. *For each count of checkpoints per subproblem, we conducted **five trials**. The graphs we present show the average, minimum, and maximum execution times of each checkpoint count.*

We can clearly see that checkpointing has a **significant execution burden** on the `mergesort` itself. Increasing the number of checkpoints per problem drastically increases the execution time linearly. While our checkpoint-restart program scales linearly for each checkpoint requested, the actual execution cost per checkpoint is fairly significant. We do not currently have benchmarks demonstrating why so much execution time is consumed due to time restraints; however, we theorize that one potential reason is because **file I/O is a substantial bottleneck**. Because the whole program is, in some way, saved to a file, this bottleneck may explain the considerable amount of time consumed.

Figure 15: User Execution Time

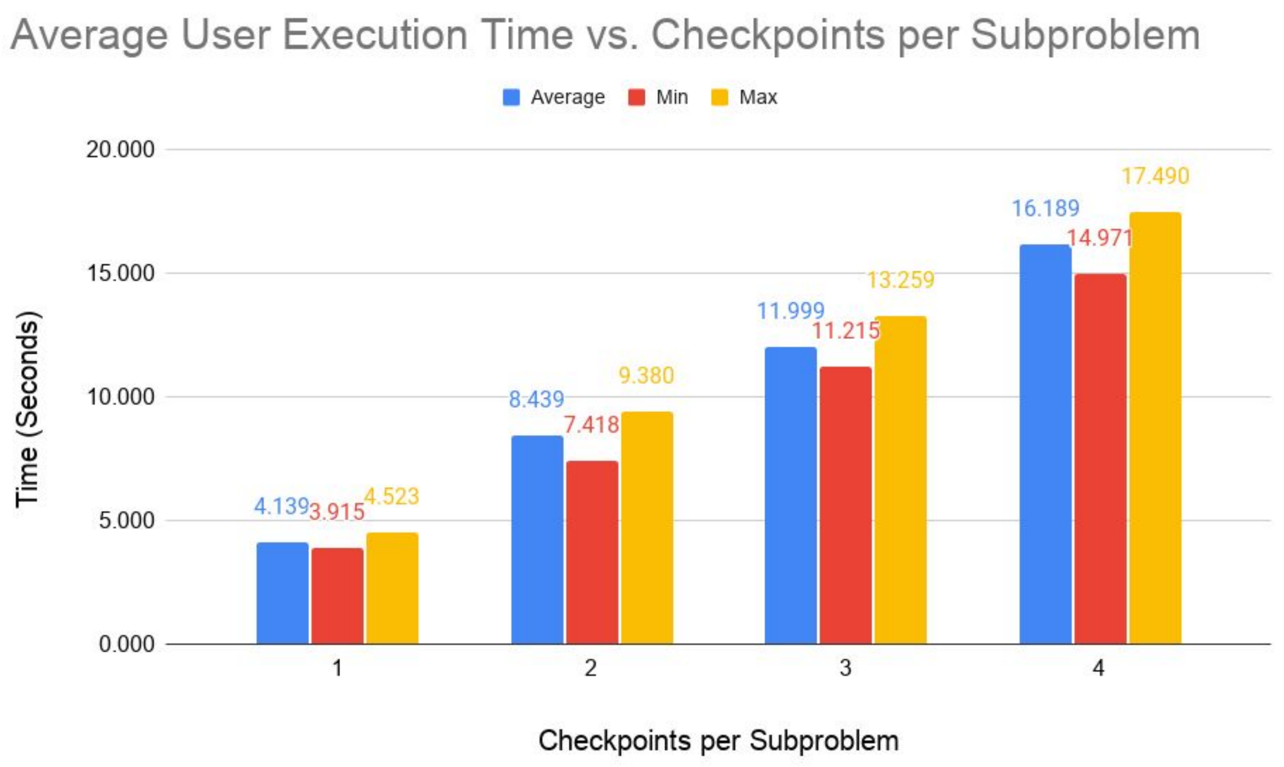
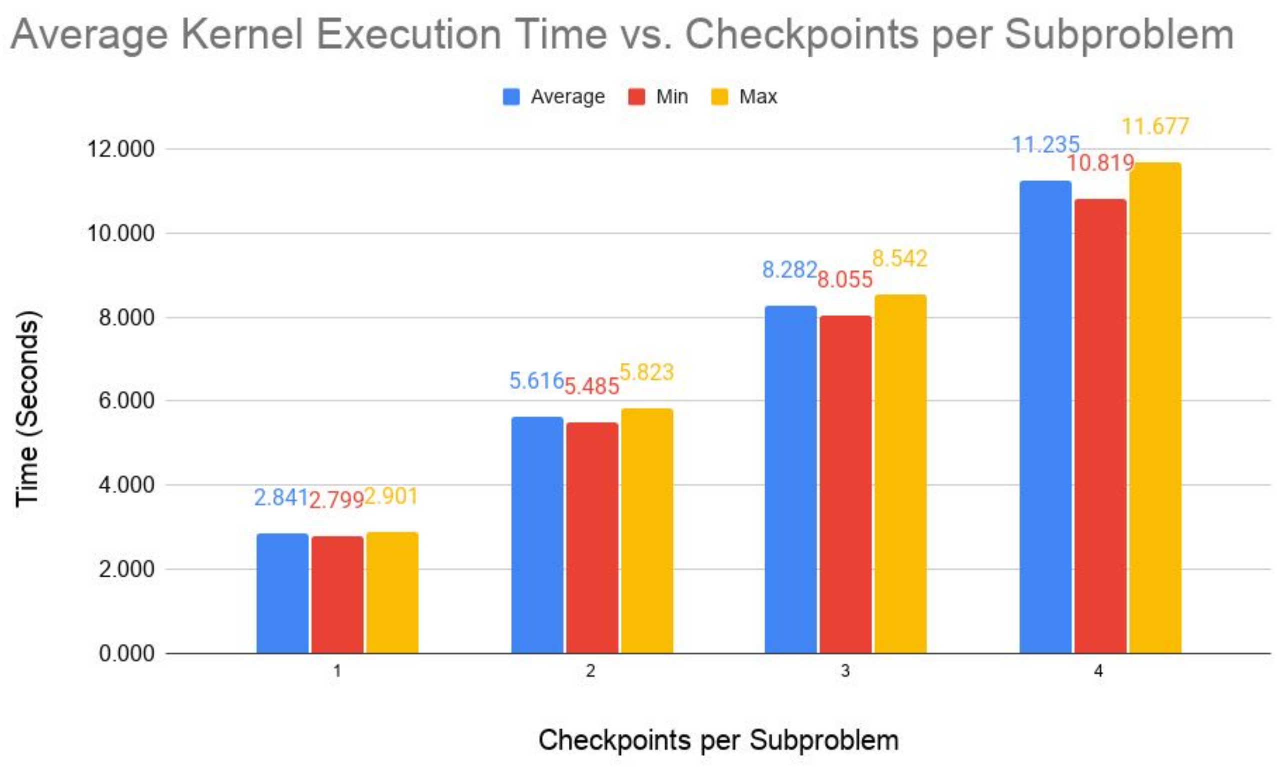


Figure 16: Kernel Execution Time



Conclusions, Lessons Learned, and Further Work

Implementing a checkpoint-restart framework proved to be a significant undertaking in this project. We encountered many, **many** roadblocks in our efforts. Not only was the initial system decomposition sometimes confusing to decompose, *many of the integration elements were also significantly harder to implement than we had expected.* We did not expect the need for **so many extra background threads** in order to implement something as conceptually simple as spawning a thread, and we certainly did not expect to need to **write assembly which handles manual context switching**.

Many tools which we had relied on in previous courses also broke on us in a multitude of ways. **Valgrind did not execute our project**, meaning we needed to write our own memory leak tracing tools. Unit tests were often challenging to write, since we needed to formulate many of the tests beyond the traditional CS approach of testing per-function. **We have crashed GDB within the process of debugging this framework**, particularly when we were attempting to implement context switching. Stepping through the program, line by line, was a feature we had taken for granted which was now unreliable.

We believe most of these struggles arose in large part because the system we were attempting to create **functioned at an almost-invisible level** to the application developer. We could have definitely reduced our work by redefining the semantics of checkpointing to require application-level registration and file management, as well as changed execution semantics to require users to log their logic branches manually. *The core functionality of this framework in providing users an almost-invisible experience which allows them to implement programs with traditional execution branch control was the root of almost all of this project's biggest challenges.*

Despite all of this, we believe that the final product we have produced **accurately reflects our knowledge of systems computing** as a small team consisting of just an ECE student and a CS student. The final product, while not identical to our original project proposal, *successfully implements a checkpoint-restore framework which executes user-level functions with minimal application interaction required.* While the system's performance is not necessarily optimal, we believe that the core functionality is complete, and that this project serves as a strong foundation for polish and future work.

Plenty of potential exists to extend our project beyond what we have written this semester. Performance analysis shows that checkpointing in this project can be fairly slow, and analyzing how to reduce file I/O could yield a significant speedup. **Persistent system resources** were a major feature which we lacked the time to implement, and the spawning of multiple threads should probably be revisited. We could potentially **increase the throughput of the free-list heap** by switching to more effective data structures. **Kernel modules** which assist in execution by offloading some of the checkpointing responsibility into kernel space, and the *entire checkpointing strategy* could even be revisited. Perhaps, an alternative approach requiring more manual user input may prove to be a better solution.

References

[1] Terry Ching-Hsiang Hsu, Kimberly Keeton, Helge Brugner, Patrick Eugster, and Indrajit Roy. 2017. NVThreads: Practical Persistence for Multi-threaded Applications. In *EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems.* Belgrade, Serbia. ACM, Inc., New York, NY. http://dx.doi.org/10.1145/3064176.3064204

[4] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-Transparent Checkpoint/Restart for MPI Programs over Infiniband. In *2006 International Conference on Parallel Processing (ICPP '06).* Columbus, Ohio. IEEE. https://ieeexplore.ieee.org/document/1690651

[3] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-Oriented Recovery for Non-volatile Memory. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 153 (November 2018), 22 pages. https://doi.org/10.1145/3276523

[4] Jie Ren, Kai Wu, and Dong Li. 2019. EasyCrash: Exploring Non-Volatility of Non-Volatile Memory for High Performance Computing Under Failures. arXiv:1906.10081. Retrieved from: https://arxiv.org/abs/1906.10081