UNIVERSITY OF COPENHAGEN

A checkpointing library in C/C++

for use in a server enviroment with a single address space

Frederik Leed Henriksen

Supervisor: Brian Vinter

18.05.2020

Abstract

Checkpointing is the discipline of enabling a application to save progress periodically during execution with the possibility of resuming the application to that point later. This can improve the applications fault tolerance along with supporting process migration and more.

A checkpointing solution is often made with a specific use-case in mind and vary greatly in transparency to the programmer and performance, with faster checkpoints often being a trade-off of greater memory usage.

Here I show a checkpointing library using an approach for application level checkpointing that hides the associated I/O of saving a checkpoint without using significant additional memory.

Benchmarking of the library under a near optimal scenario performs with an overhead of around 1.0%.

The findings show that a low overhead on application run-time is achievable without increased memory usage while also providing a functioning general purpose checkpointing library.

Abbreviations

- **OS**; operating system
- **I/O**; input/output
- **BSP**; bulk synchronous parallel
- **mutex**; mutually exclusive
- **MTTI**; mean time to interruption
- **MTTF**; mean time to failure
- **CPU**; central processing unit
- **syscall**; system call

List of Figures

Contents

Chapter 1

Introduction

There are indeed many ways an application can fail. It can be a silent error or one that causes a error/warning to be thrown. It can be an error originating from a human error when writing the code or a hardware error. In many production environments, if there is a shared computation resource available, there possibly is a maximum limit as to how long one job is allowed to run for[1]. Bottom line: there are many ways to crash an application or getting the computation cut short. Often scaling with the time required to computing the job, fault tolerance or the ability to break up computations in some interval becomes increasingly desirable.

Checkpointing is a convenient and approachable way to limit the amount of computation time lost if an error occurs or the computation is otherwise cut short. By periodically copying some/all of the applications memory to a safe location, checkpointing enables the user to use the computer resources better or more time efficient in those cases where the programmer has reason to believe the execution will be prematurely stopped. It does of course come with the added computational cost of I/O the data to/from the checkpoint and also potentially some extra code for the user to deal with.

This thesis aims to provide a ligthweigth and easily approachable checkpointing library. Specifically, this report details the design of an application level synchronous shared-memory checkpointing library written for C++ that I have simply chosen to call **checkpointlib**.

To download the repository containing **checkpointlib** along with unit-tests, code-examples and benchmark-code, visit this site: `https://gitlab.com/fumpen/checkpointlib`
Note that when there is referred to the "**checkpointlib** repository" in this thesis, it is a reference to this repository.
As an alternative, an identical repository is available here: `https://github.com/fumpen/checkpointlib.git`.

[1] sometimes refereed to as a walltime-limit

1.1 Research question of the thesis

The objective of this thesis is to create a checkpointing library in C/C++ for use in a server environment with a single address space.

When executing a simulation or a large calculation on a server that needs many hours, days, months or even years, it is often desirable to have certain points in the execution in which the program can be stopped. It could be if hardware needed change, an other program maybe need to be executed or something else. This brings us to the field of checkpointing, where this thesis seeks to solve this need with regards to programs in C/C++. I will construct a library that enables a user to make checkpoints and start code execution with the state and starting-point of a given checkpoint. The library will be set up to be functional in a server environment with a single address-space. If the time permits, it would be interesting to explore what the ideal way to save a checkpoint would be, both in what format might be ideal and also what should be saved. It could be that the entire state should be saved every time? It could also be an incremental approach or maybe a third option. It would also be interesting to see how flexible the library should be with regards to how it is determined what should be saved. Maybe it should be user-specified what variables should go into the checkpoint, it could possibly also be an automated process or maybe even a mix.

1.2 What is checkpointing

The basics of what checkpointing is and the terminology for this section is described here: [Sch11].

Checkpointing at it's base, is the act of enabling the programmer to resume the state of some code, which it previously had. It saves the state of the code in what is called a checkpoint and if at a later time the programmer wishes to continue from that checkpoint, he/she can "restart" the code from that point. The computations happening between two checkpoints is called an "epoch".

1.2.1 Application and system level checkpointing

Checkpointing can be implemented on two levels. On the application level and on the system level.

In a system level checkpoint, all the data that makes up the process is saved (even external resources such as open files). System level checkpointing can be implemented completely in the operating system, making it completely transparent[2]. Since it is implemented with an OS in mind, it is likely that the implementation of system level checkpointing is not easily made portable.

Application level checkpointing is a checkpointing approach in which the programmer has to incorporate the checkpointing in some way into the application, meaning at least some effort from the

[2] meaning the programmer does not have to modify the application in any way for it to make use of checkpointing

programmer is needed. The application level checkpointing is however potentially advantageous in that it is not blind to the program and can make more intelligent decisions in what is necessary to save in a given checkpoint, as to facilitate a restart.

In general, system level checkpointing has a great deal of transparency and at the cost of larger checkpoint file sizes and run-time overhead. Application level checkpointing is on the other hand an effort to apply for the programmer each time a new application is constructed, but allows for potentially smaller checkpoints and therefore a more effective implementation. Furthermore it is likely to be easier to make portable.

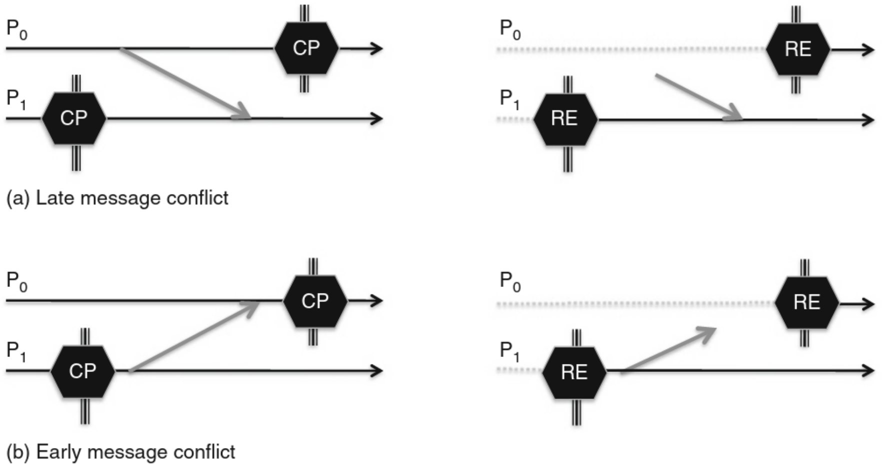
Finally, there are efforts to combine the two approaches where, through preprocessing, the compiler attempts to determine the important parts of the program. This would potentially combine the advantages of both approaches, making the checkpointing transparent and having a smaller file size requirement for the checkpoint itself.

1.2.2 Shared memory and message passing application checkpointing

Due to increased risk of failure and the increased amount of resources required for parallel programming, there is an increased incitement to use checkpoints in parallel programs.
It comes with the added challenge of finding the correct place to take the checkpoint, given that concurrently running code on two different runs, is likely not at the same place in execution relative to each other when reaching a checkpoint.

Shared memory checkpointing is checkpointing implemented on a parallel running code originating from the same process and all threads running in that process have access to the same memory. A common way of taking a coordinated checkpoint here would be to synchronize the threads at the time of taking a checkpoint using a barrier. Alternatively in the rare scenario where there is no shared memory between the threads, each thread can create a checkpoint in parallel.

Message passing application checkpointing is checkpointing implemented on concurrent running processes. When making checkpoints on multiple processes that message each other, there is the additional risk of causing what is referred to as a "late-message conflict" or an "early-message conflict".



(a) Late message conflict

(b) Early message conflict

Checkpointing. Fig. 3 Message conflicts caused by uncoordinated checkpoints (checkpoint shown *left*, restart shown *right*)

Figure 1.1: Early/late message conflict. Model found here: [Sch11]

This can be solved by coordinating the processes (e.g via a barrier synchronization) or by additionally saving the messages that are sent/ should be sent and thus avoiding any loss of messages on restart.

1.2.3 Storing a checkpoint

Write to disk

To perform a restart from a checkpoint, it needs to be stored somehow and somewhere.
The most straightforward way is to write all data connected to a checkpoint to disk, saving the checkpoint even if the application crashes or system unexpectedly shuts unexpectedly down.

To save a checkpoint in it's entirety can amount to a lot of memory and result in a lot of I/O needed. An alternative strategy is the incremental storage strategy, where each new checkpoint only contains the changes/additions in relation to the former checkpoint. Meaning a restart of the code can be performed at minimal memory cost, though some additional cost in the shape of reconstructing the checkpoint from all the previous checkpoints are required.

Diskless checkpointing

Often the greatest cost of checkpointing results from the I/O needed to commit a checkpoint to disk. Diskless checkpoints offers a way to remove I/O entirely. Instead of writing a checkpoint to disk, the checkpoint is stored in memory across multiple processors. This approach enables restart even if a number of processors crashes. What processor that crashes is important for the recovery. E.g the three methods mentioned in [PLP97] shows the decisions and trade offs involved in implementing a diskless solution. An example of the difference in fault tolerance using different methods is shown in figure 4.2, where "Raid level 5" would be unable to recover from just two simultaneous crashes, "Mirroring" would require both the processing and the checkpointing processor from the same pair to crash before recovery completed to be unable to recover and the "one-dimensional parity" method would depend on the size of the groups of processing processors per checkpointing processor

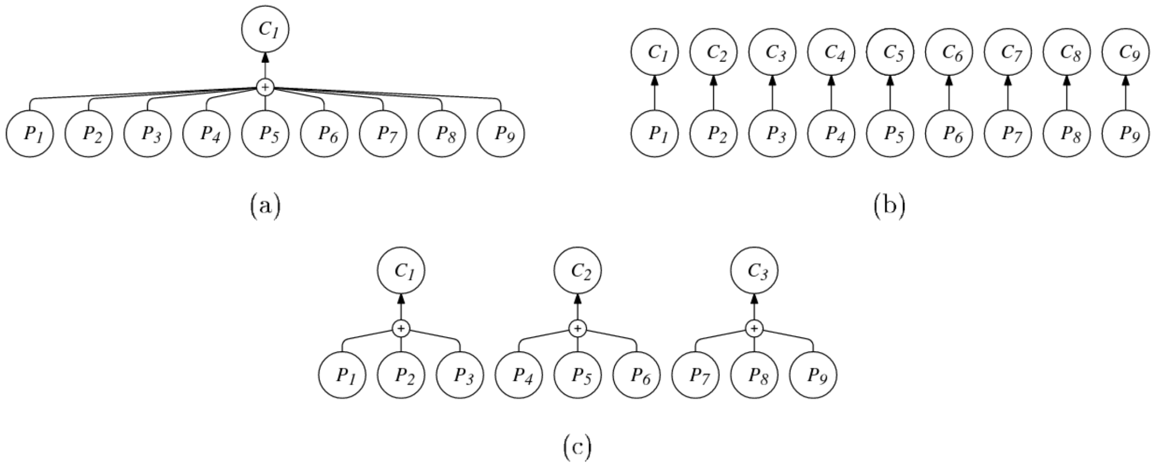

C1
P1 P2 P3 P4 P5 P6 P7 P8 P9
(a)
C1 C2 C3 C4 C5 C6 C7 C8 C9
P1 P2 P3 P4 P5 P6 P7 P8 P9
(b)
C1 C2 C3
P1 P2 P3 P4 P5 P6 P7 P8 P9
(c)

Figure 2: Encoding the checkpoints: (a) Raid Level 5, (b) Mirroring, (c) One-dimensional parity

Figure 1.2: Encoding examples of diskless checkpoints. Model found here: [PLP97].

Finally, regardless of method, the diskless approach does of course not persist when powering off the machine entirely, leading to some combinations of the diskless and the more traditional approach of writing to disk.

1.2.4 Why use checkpointing

A very common use for checkpointing is to provide fault-tolerance. Meaning if a thread/process crashes it is possible to restart from the latest checkpoint with limited computation time lost. It is also entirely possible to automate the restart in this case.

Time-sharing computer resources can be an advantage when some applications might take a long time to compute. Checkpointing here can offer a convenient way of segmenting an otherwise lengthy computation as to enabling e.g other applications time on the shared resources.

Say there is interest in the outcomes of some application and that the application always initiates the same way. A checkpoint after the initiation can potentially save a significant amount of time restarting from a checkpoint, especially since that checkpoint can be restarted from multiple threads/processes if needed.

Finally, depending on how the checkpointing is implemented, it is even possible to move a live process from one computer to another computer. Now there is of course some restrictions to this, e.g a system-level checkpoint probably require the sender and receiver to have a similar OS.

1.3 Related work

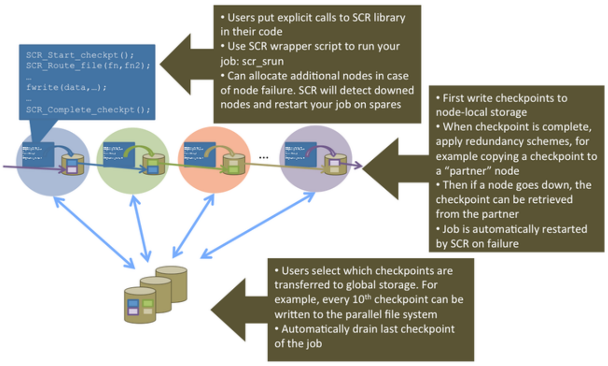
As there is a vast body of work concerning checkpointing and implementations thereof, this section is limited to libraries that represents an interesting or effective checkpointing related solution and which are significantly different from the other included libraries.

libckpt
An old and almost completely transparent library, libckpt ([Pla+95]) can with almost no change to the application code[3] enable regular periodical checkpoints to be made. As it is implemented to be adjustable with regards to frequency of taking checkpoint's.

VirtualBox and Docker
Perhaps the simplest way of applying checkpointing to some application is to run it through some of the virtual environment software like Docker ([Inc13]) and VirtualBox ([Cor07]). They are able to make checkpoints that can be run on basically every system that is capable of installing Docker/VirtualBox. All while being completely transparent to the application. Of course, since everything is saved, it is likely not worth it to implement regular checkpointing like this in a more demanding situation.

SCR (scalable checkpoint/restart)
The SCR library [Lab07] started as an attempt at addressing the problem presented in [Sch05] and designing a solution. In [Sch05] the argument goes that is is more likely that a single thread fails rather than a system-wide failure. To mitigate this, a two-level recovery scheme is proposed. the first level here is where threads save in memory (i.e the diskless approach) as to enable recovery if a single processor fails. Level two consists of periodically saving a checkpoint to disk (e.g every fifth checkpoint) to enable recovery from more severe failures.
SCR is an application level checkpointing scheme that combines techniques of diskless checkpointing and the conventional save to disk. In SCR each thread saves it's checkpoint locally before continuing the computation. After the local save it saves the checkpoint in another thread, as to always have a a backup if a single thread fails and enabling recovery immediately. For more severe failures there is a second backup where the programmer can decide how often a checkpoint should be written to disk.

[3] literally just change one line, regardless of the application's size



```
SCR_Start_checkpt();
SCR_Route_file(fn,fn2);
…
fwrite(data,…);
…
SCR_Complete_checkpt();
```

- Users put explicit calls to SCR library in their code
- Use SCR wrapper script to run your job: scr_srun
- Can allocate additional nodes in case of node failure. SCR will detect downed nodes and restart your job on spares

- First write checkpoints to node-local storage
- When checkpoint is complete, apply redundancy schemes, for example copying a checkpoint to a "partner" node
- Then if a node goes down, the checkpoint can be retrieved from the partner
- Job is automatically restarted by SCR on failure

- Users select which checkpoints are transferred to global storage. For example, every 10th checkpoint can be written to the parallel file system
- Automatically drain last checkpoint of the job

Figure 1.3: Representation of the SCR workflow, found here(2020-05-10): `https://computing.llnl.gov/projects/scalable-checkpoint-restart-for-mpi`

CRAFT

CRAFT ([Sha+17]) is a newer library that comes with support for asynchronous checkpointing and supports SCR. In addition it comes with an option for the programmer to define custom data types to be committed to checkpoint, by wrapping the custom class in another class with suitable read and write functions.

CRAK

There is a great deal of system level checkpointing libraries[4]. CRAK ([ZN01]) is one such example. It is completely transparent to the programmer and focuses on facilitating process migrations and enabling administrators of shared computer resources to manage applications without the need to contact the ones responsible for the running applications.

1.4 Proposed solution

The proposed library **checkpointlib** is an application level library on a Linux OS and based on the BSP model. The library is intended for use on multi-threaded applications that uses only one process (i.e the treads share memory).

The programmer is free to decide how often to checkpoint (i.e the length of each epoch), how many threads to start and the size of the buffer in memory that is to be made into a checkpoint.

The library allocates a block of memory during initiation (like one would use malloc), starts threads much like pthreads([Ker19a]) and synchronizes the threads upon each checkpoint.

This library is implemented as to minimize the cost of I/O, but not by using extra memory like diskless checkpointing. It uses an approach that imposes write-protection on the memory intended to be written to disk. A thread will write the data to disk in the background and gradually removes

[4] quite a few are discussed in this article [Gio+12]

the write-protection, enabling the application to continue its computation after synchronizing but potentially without the need to wait for I/O.

The library is intended to put as small an overhead on the application as possible. With the exception of the thread responsible for I/O, there is no thread or process initiated from the library. The data structures used require little extra memory as more threads are added. It is only depending on standard libraries and requires a small effort for the programmer to integrate.

It has, as far as I can tell, not been attempted to create a checkpointing library using this approach (i.e the segfault handler and write protection combination).

In the remainder of the thesis, the block of memory allocated by **checkpointlib** during initiation will be referred to as "checkpoint memory".
The files created to receive the checkpoint memory by **checkpointlib** will be referred to as "checkpoint files".
The thread that writes checkpoint memory to a checkpoint file will be referred to as a checkpoint thread.

1.5 Outline

Here is the overview of the remainder of the thesis:
Chapter two and three covers theory that was relevant for the construction of the library.
Chapter four investigates what directions a checkpoint library design can take anlong with the direction this thesis ended up following.
Chapter five covers the design, implementation and performance of the proposed library.
Chapter six mentiones what areas would be interesting to improve in the library.

Chapter 2

Memory Management

When a programmer is constructing an application using checkpointing, chances are, there is some memory which the programmer needs to save/restore at the given checkpoint. In the proposed library, the programmer is given the option to allocate such an area in memory which is usable by the library. This however also gives rise to the question of how this allocated memory is managed. Memory management, in this thesis, refers to the the management of one or more blocks of memory[1]. Concerning memory allocation, it seems relevant to look at the allocation itself, the logic of finding and freeing memory blocks and the problem of memory fragmentation.

2.1 Allocating memory

It is important that each block of memory discussed in this thesis is continuous. I.e if a block of 20 bytes in a byte-size-indexed address space is allocated starting at address 0 in memory, that block will span addresses 0-20. In a Linux system, such allocation is often achieved by the mmap function([Ker19b]) which is also how it is allocated in C/C++'s 'malloc' on larger allocations.

2.2 Fitting allocation to available memory

Say the programmer have reserved a block of memory to be saved by the library. While it would be the simplest approach and perfectly viable to simply let the programmer use this block as desired, it is not very user-friendly. In this library the option to allocate through the library and letting the programmer keep the usual coding pattern of malloc/free from C/C++ will hopefully make the library more approachable. This however requires the library to manage the programmer's allocations in a sensible way.

2.2.1 Bin packing problem

What is called the one dimensional bin packing problem is at the base of seemingly all memory management tasks. The problem in it's classical form, a version of which is mentioned in [JG85] and [Kor02], is as such:

Given a list **L** of items, each with a size in the interval $(0, 1]$ and a sequence of bins B_1, B_2.... each

[1] A "block" of memory here referring to a continuous location in memory of some size

with a capacity of 1. The goal is now to “pack” as many items into each bin as possible without their total size surpassing 1, leaving **L** empty and as few bin’s full as possible.

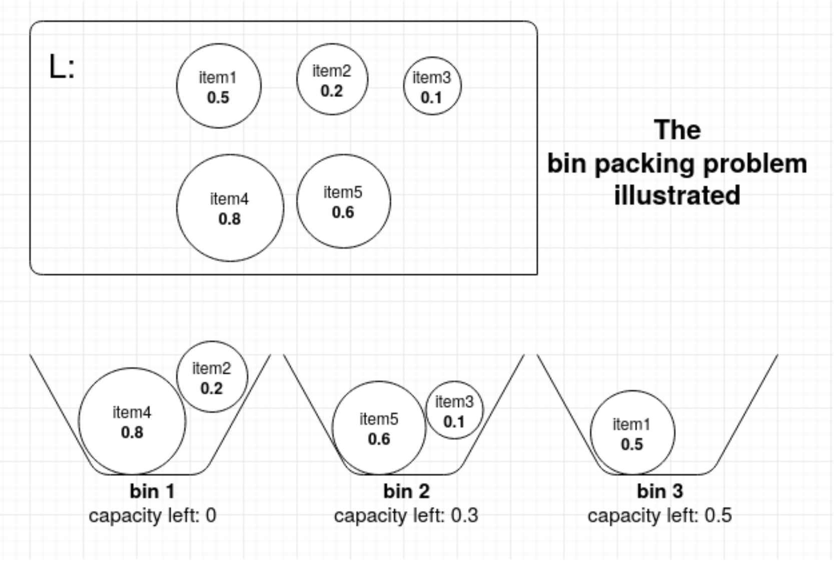


Figure 2.1: An illustration of the bin packing problem, where it is necessary to use 3 bin’s

Now in this thesis, the bin is a block of memory available to the programmer and the items are allocations made into the bin’s. The bin packaging problem is a NP-hard problem, therefore many heuristic methods have been developed and refined to tackle the challenges of memory management as efficiently as possible (one such heuristic will also be presented in this thesis).

2.2.2 Dynamic allocations and it’s problems

Naturally it is restrictive to have all bin’s of the same size and of course one would not necessarily know all allocations/deallocations at any one time. Let’s look a bit closer at the problems that are presented for dynamic allocations (some mentioned in [Knu07]).

First consider that **L** is not known at the beginning of execution. This means that whenever a new item is added to **L**, it needs to be put into a bin before knowing the final set of **L** and therefore the bin that the new item is put into might not be the optimal bin. For example, if at time $T1$ $L = \{0.8, 0.8, 0.1, 0, 1\}$, then $B_1 = \{0.8, 0.1\}$ $B_2 = \{0.8, 0.1\}$ would be an optimal solution. If however at time $T2$ $L = \{0.8, 0.8, 0.1, 0, 1, 0.2\}$, then either the items need to be reassigned to another bin[2] or a new bin needs to be put to use. So an optimal assigning of items to bin’s at one time, might not hold at a later time.

Second, note the freedom to choose the size of the bin’s ,I.E all bin’s do not necessarily have identical capacity. To keep the abstraction of bin’s and items, think of this as the bin with the largest capacity as being empty and the remaining bins as being partially filled with “placeholder items”.

[2] I.E defragmented

Third, consider the option of removing an item from **L**(essentially freeing an allocated block). Now we have to stray from the idea of bin's and items a bit. In memory, each allocation (item) is put at a specific address and fills some bytes following this address. This means that in reality an item is not simply put into a bin, but is actually put into a specific position in that bin. This has significance. For instance, if a bin with capacity of 1 has the items in the following order in it $[0.1, 0.2, 0.5, 0.2]$ it is at max capacity. Now say the two items of size 0.2 is removed, the bin now have an excess capacity of 0.4. In an ideal situation, an item of size 0.4 should now be able to fit into the bin, but since the position matters, the item of size 0.5 is "in the way" and as a result, only items of size $0, 2$ or less can be assigned to this bin. This problem is usually referred to as memory fragmentation.

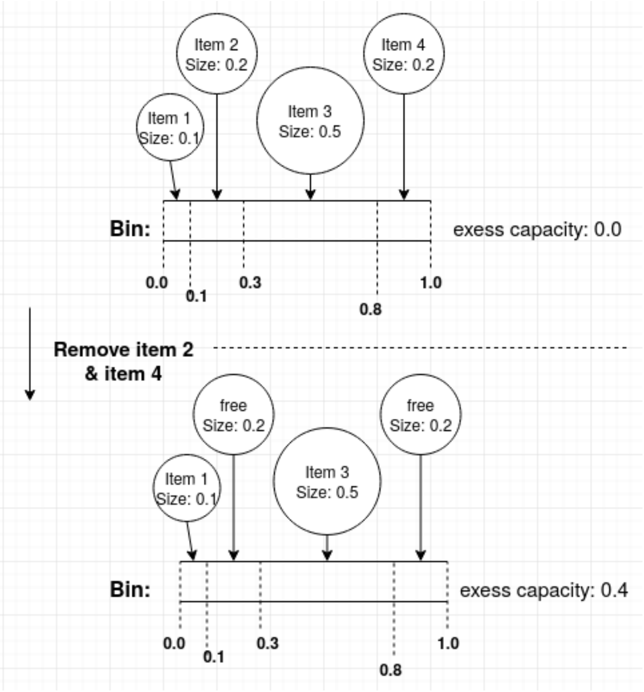


Figure 2.2: An illustration of fragmented memory, using the metaphor of bin's and items

The bin packing problem with the additional points above are some of the challenges that e.g an OS contends with when running processes. The solution often employed is a heuristic method that strikes some balance between how fast an item should be assigned to a bin and how many bins (in exes of the optimal solution) the OS is willing to use.

2.2.3 An algorithm that fits

As mentioned, many heuristics exists that solves the problem of allocating memory with regards to either speed, minimum of allocated space required or some other priority. The possibly best known method is the first fit algorithm (e.g described here [**p437**; Knu07]).

Algorithm 1: First Fit - a simple implementation

```
Result: some pointer if size n was successfully allocated, 0 otherwise
With P and Q being pointers, some size n that is to be allocated, AVAIL being the first
  element in a linked list of free memory and each pointer having a size (denoted P.size)
  and a pointer to the next element (denoted P.next). Also assume that the last element in
  the list points to nothing (denoted by 0), so E.G an empty list is AVAIL = 0. Start by
  setting ;
P = AVAIL;
Q = AVAIL ;
while P != 0 do
    if P.size >= n then
        if P.size > n then
            P.size -= n ;
            P += n ;
            return P - n;
        else
            Q.next = P.next ;
            return P;
        end
    else
        Q = P ;
        P = P.next;
    end
end
Return 0;
```

There are a great many strategies for allocating memory, best fit, worst fit and next fit to mention a few. There are also methods like the buddy system that does not even optimize for dynamic allocations and presumably many more. In this thesis, I will focus on the first fit algorithm, because deciding what heuristic is the best suitable for this library is worthy of a thesis on it's own[3].

The library needs to be able to handle a bit more than what algorithm 1 is able to handle in its current state however. It is still not able to handle all that was described previously. First of all, a way to free allocated memory is needed. Again, there are multiple valid ways of doing this, I will be using a algorithm much like the one described in [**p440**; Knu07].
The tricky part of freeing a block, is that if it borders a block of free memory on either side(or both sides) a "freeing" algorithm should ensure to merge the freed block into the bordering free block(s) to ensure the largest possible block is available to new allocations. For now, lets change the name of "AVAIL" to "MEMMANAGER", since all blocks in it are not necessarily available memory and lets assume that each block in AVAIL also have the attribute "P.is_allocated" that is true if a block is allocated memory. However having allocated blocks tracked does slow things down a bit, since discovering the bordering blocks potentially will require to look through the

[3] slight changes in the implementation, e.g how to save the nodes of free/allocated memory or when in the program the allocations happen might also have an impact on what is optimal, making development slower than needed.

entire list of MEMMANAGER. It can however easily be improved a bit.

Lets assume that MEMMANAGER is put into order of increasing memory address, i.e the lowest address in MEMMANAGER P_0 is the first element and $P_0.next$ points to the second lowest member of MEMMANAGER. This ordering is not necessary for correctness, but guarantees that when the correct point in MEMMANAGER is found in a “free” operation, the point immediately before and after in the list is the relevant block’s of memory to potentially merge. If the list was not ordered, both the block before and the block after would have to be identified and with no order they could be anywhere in MEMMANAGER, effectively increasing the chance that it is necessary to look through the entire list.

There are many things that could be done to possibly improve the above algorithm. Some honorable mentions:

- for allocation and free efficiency: Add a minimum size to each allocation, as to stop small blocks of free memory that will probably never be used to clutter the allocation process. This is simply implemented by setting a minimum size, and if after an allocation the remainder of the block is below this size, silently add the remainder to the new allocation.
- for allocation efficiency: Keep a pointer to where the last search stopped and start searching for a new allocation from said pointer (this is the next-fit approach and all that separates this from first-fit is a single pointer). This should mitigate the concentration of small (useless) allocations at the start of the MEMMANAGER list.
- for free efficiency: knowing the size of the entire address space in MEMMANAGER, if an address to be freed is closer to the end, rather than the beginning of the list, the elements could be made into a double linked list with a pointer to the last element in MEMMANAGER made available as to go from the bottom up. Were all allocations roughly of same size, this could potentially half the time of a “free” operation

But in the end, the above mentioned all make assumptions about the size and distributions of the allocations, so that will be left as potential future work.

Chapter 3

Concurrency

In this chapter I discuss some of the options and problems of having a checkpoint library running on concurrent code.
While it is the hope to impose as little run-time overhead to a user of the checkpoint library as possible, there is still a need for serialization (paralelle applications being forced to run sequentially for some part of their run-time) and mutual exclusion (two events are guaranteed not to be run simultaneously). This chapter will serve to discuss some techniques which exist to provide synchronization and the cost of their employment.

In this section it is nice to remember that a "thread" ([Ker19a]) in a UNIX system, is a piece of code which the OS scheduler can choose to run just as it would any process. The difference between a thread and a process is that a process can live on its own, but a thread must be created by a process and is bound by it's lifespan. Furthermore however many threads a process creates, they all share the same memory belonging to that process.

3.1 The Bulk Synchronous Parallel model

In designing **checkpointlib**, a model was needed which offers as small an amount of necessary synchronization as possible. The BSP model is widely known and offers a simple approach to design applications. It offers a framework for running parallel programs while taking into account the need for synchronization. Thus, it seemed a good choice of inspiration for the library.

The bulk synchronous parallel model (BSP), described here [Val90], is a model attempting to ease the development of parallel systems. It does so by describing tree attributes of a machine which software and hardware developers can follow:

- Components whitch carry out computations and/or operate on memory.
- A router that can deliver a message between pairs of components.
- facilities that can synchronise all or part of the components.

The attributes are mentioned to describe what capabilities the hard/software should provide, not that they necessarily have to be separate attributes.

Each component has a number of tasks to complete and in between a set number of tasks, referred to as a "superstep", all components are synchronized. Each superstep is the expression of all the work the component have to do between global synchronization. A task can include the sending/receiving of messages to/from other components, but note that each component does not have a guarantee of having received the messages it expects in the current superstep before the global checkpoint have occurred. Another way of putting it would be that each component expect to complete the current superstep only with the information from other components that were available from the previous superstep.
Each global synchronization happens at an interval of **L**[1] times some time unit. If one/multiple component's have not finished their current superstep at the time of global synchronization, an additional period of **L** is reserved for the task to finish[2].

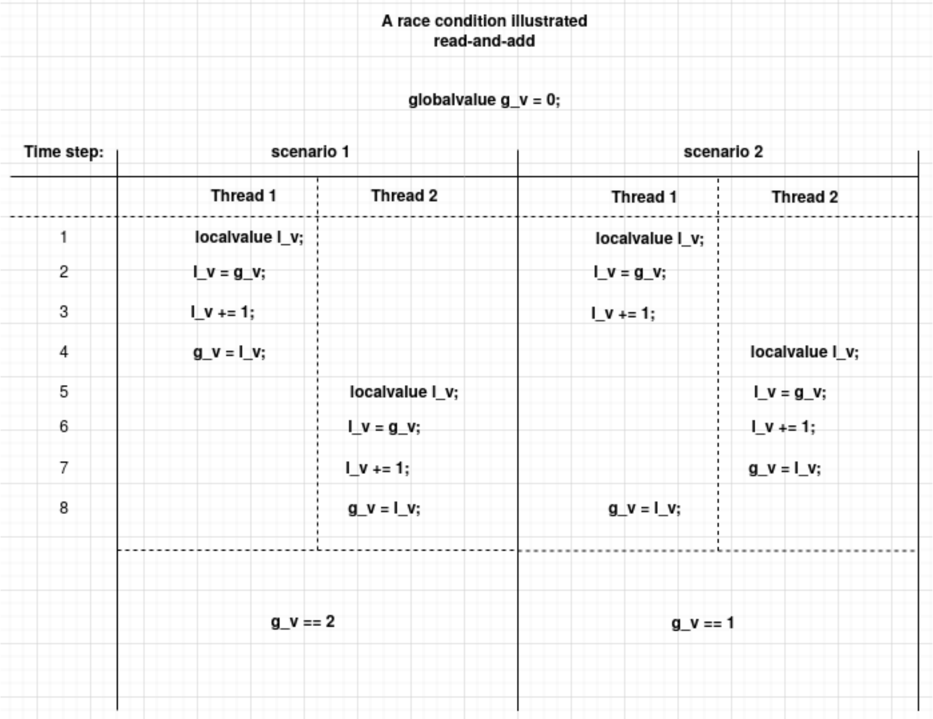
With regard to applying BSP to the checkpoint library, a "component" is analogous to a thread. The "router" can, in regard to the checkpointing library, be regarded as the shared memory of the process or a signal, e.g from a semaphore ([Dow16]). The "synchronization facilities" is a combination of shared variables and lock's known collectively as a barrier lock. Furthermore, think of the "superstep" as an epoch.

3.2 An incentive for synchronization

Before getting into the lock and barrier synchronization, consider the problem in parallel computing that makes the lock relevant.
When coding in parallel there is almost always the risk of encountering what is often referred to as a race condition ([NM92]). A race condition is basically any part of the code that can be influenced by the order that the scheduler runs the code. A race condition can be completely harmless, it can be a used as an malicious tool for hacking a system or it can simply be an oversight by the programmer leading to unwanted behavior.
With regards to the checkpointing library, the interesting case is that of code that affect a shared memory location. The results can vary greatly if no synchronization measures are taken. A popular example is the read-and-add example:

[1] Some parameter which optimal value would depend on both hardware and software. Therefore whatever program is running have a potential individual optimal value.

[2] potentially multiple periods of **L** time units are allocated if one is not sufficient for a component to finish it's superstep.



A race condition illustrated
read-and-add

globalvalue g_v = 0;

Time step:	scenario 1		scenario 2	
	Thread 1	Thread 2	Thread 1	Thread 2
1	localvalue l_v;		localvalue l_v;	
2	l_v = g_v;		l_v = g_v;	
3	l_v += 1;		l_v += 1;	
4	g_v = l_v;			localvalue l_v;
5		localvalue l_v;		l_v = g_v;
6		l_v = g_v;		l_v += 1;
7		l_v += 1;		g_v = l_v;
8		g_v = l_v;	g_v = l_v;	
	g_v == 2		g_v == 1	

Figure 3.1: An illustration of a race condition, emphasizing the importance of knowing in what order code is run.

Because of race conditions like the one above, the need for so-called "critical sections" in code become apparent. Here a critical section in code is a section of a parallel program where that section is required to run sequentially regardless of what order the scheduler runs the threads. For instance in the example 3.1, if the entire read and add section was deemed critical, this would mean that scenario two would be invalid.

3.3 Lock synchronization

The lock is a powerful and simple tool, widely used to synchronise threads or ensuring some order in way of accessing a critical section of code. Though the term "lock" covers many techniques/implementations/concepts, they all provide a framework that introduces some control over what code is run when, despite what order the scheduler decides.

Any lock is likely to be based on a "test-and-set" call ([AA03, Chapter 28]) or something equivalent[3]. A test-and-set call writes a new value to memory and returns the old value. This call works atomically, meaning that the writing of a value and return of the old one should be viewed as a single instruction. Because of the atomic attribute, test-and-set enables the mutual exclusion that is needed for a lock.

A basic lock have two functionalities, namely that it can be locked/held/acquired and unlocked/released. Depending on the type of lock and settings it is subjected to, it varies how many threads

[3] The one called "compare-and-swap" comes to mind.

can hold the lock at any time. But common for all is that if a thread is unable to hold the lock, it will not continue execution until it can be acquired. How the thread waits for the lock is a matter of the individual implementation.
Another common trait is that any thread that acquire a lock must also release the lock after use. If a lock is not released after the critical section, one risks to encounter what is often referred to as a "deadlock". A deadlock is simply a situation where a thread is waiting for a lock that is never released, thus that thread waits indefinitely.

As said there are different types of locks. An example is a read/write lock, where the critical section covers some variable that can be both written and read. Here the same lock can be acquired as a "read" lock or a "write" lock. This is advantageous to have such a lock if in a situation where many threads needs to read the variable, since the read lock can be shared and it only becomes necessary to exclude all other threads when a thread wishes to write.

In this thesis however, I'll focus on what is referred to as a "mutex" (described here [AA03, Chapter 28]). A mutex is a lock that can only be held by a single thread at any time. All other threads will wait until the lock is released and can be acquired anew by another thread. Note that the algorithms presented later does not have to be implemented as a mutex-type lock.

3.3.1 Drawbacks of using a mutex

While the mutex provides a way to safely operate within critical sections, it does come with a set of potential pitfalls. The ones that are relevant to the library are briefly mentioned here.
As previously mentioned, the deadlock where a thread could wait for ever on a mutex that would never be released. For a mutex that is not carefully implemented, there are many ways to encounter this case. A thread could encounter an error while holding the lock, exit and never releasing the mutex, two threads could each hold a mutex and try to acquire the one held by the other thread or the programmer could simply forget to release the mutex.
Another risk is "starvation". This occurs where multiple threads attempts to hold the same mutex. Here, if the implementation of the mutex or the programmer does not anticipate this, it is basically left up to luck what thread acquires the lock. If say all threads continuously iterates over the same mutex, there is a real chance that a thread will be continuously "unlucky", never/rarely acquiring the lock and essentially be a waste of processing power.
Closely related to starvation is "fairness". The idea is that the thread that arrives at the mutex first, acquires it first, the thread that arrives second gets to hold it next and so on... Again this depends on the scheduler and the "luck" of each thread.
Finally there's "performance". Now this is a given. Imposing any synchronization on concurrent code is almost guaranteed to impact the performance negatively. Further more, the larger a critical section is, the greater the negative impact on the run-time is likely to be.

3.3.2 Implementing a mutex

Using a test-and-set call is by itself almost enough to implement a simple "spin-lock" ([AA03, figure 28.3]). A spin-lock being a mutex that continually checks whether it is available. Increasing the chance of getting the mutex when it's released but at the cost of running the same check exclusively until it succeeds.

The simplest of algorithms to avoid wasting computations spinning on a lock is to check just once if a lock is available and if not, simply yield the remaining scheduled time. Then repeat the check the next time the scheduler puts the thread on ([AA03, figure 28.8]). This is an especially effective approach in cases where the system running the program only have access to one CPU (in this case, a spin-lock would be especially wasteful).

An alternative algorithm is the test-and-set mutex with exponential back-off that puts a thread to sleep (called "pause" in algorithm 2) [4]. Lessening the amount of processing required to acquire the mutex and thus improving the performance ([MS02]). Pseudo-code follows:

Algorithm 2: Test-and-set mutex with exponential back-off, modified to C like syntax from [MS02, fig 1]

```
type lock = (unlocked, locked);

void acquire_lock (lock L) ;
    int delay = 1;
    while test_and_set(L) == locked do
        pause(delay);
        delay = delay * 2;

end

void release_lock (lock L);
    L = unlocked;
```

If the possibility of starvation is an unacceptable risk, the ticket-lock is an option ([MS02, fig 2]). Here two counters keep track of the number of threads that wish to enter and the number they have accepted. Each thread gets a number when trying to acquire the lock and is thus put in a queue. With an ordering like this, there is no possibility of starvation and fairness is probably as high as it is possible to get. Since a delay for one thread in the front of the queue is propagated throughout the queue, the use of exponential back-off is not ideal with this lock, rather a static back-off time unit can be employed.

Now the area of mutex'es and locks in general is of course much larger than represented in this chapter, but with the above described challenges and possible solutions to protecting a critical section, there is a decent base to discuss how to implement a barrier and what to be aware of in the implementation of the library itself.

3.4 Barrier lock synchronization

A barrier lock is a way of having multiple threads synchronize at a specified point in each of their execution. For many types of barrier synchronization's this involves a critical section of sorts,

[4] here "sleep" refers to requesting the scheduler not to grant a thread any execution time in a certain period of time.

meaning some sort of locks are often relevant in constructing a barrier.
Besides what was previously mentioned regarding the drawbacks of locks/mutex'es, one additional drawback is relevant. While a barrier will only continue when it has stopped the number of threads, not all barriers prevent threads from entering a new barrier while there are still threads in the previous barrier. If the same barrier is reused (e.g for synchronization at the end of a loop), this can lead to a dead-lock.

The "two-lock" barrier ([Axe05]) or the essentially same "centralized barrier" ([MS02, fig 8]) are some of the simplest barriers that lead all threads through one or more centralized memory locations (e.g a counter). The two-lock barrier uses the extra lock to ensure that all threads have left the previous barrier, whereas the sense-reversing barrier uses a flag to the same effect.

Algorithm 3: sense-reversing centralized barrier from [MS02, fig 8]

```
shared count : integer := p;
shared sense : Boolean := true ;
processor private local_sense : Boolean := true;

procedure central_barrier
    local_sense := not local_sense;
    if fetch_and_decrement(&count) = 1 then
        count := p;
        sense := local_sense ;
else
        repeat until sense = local_sense ;
end
```

Be it due to waiting on the same lock or spinning on some shared variable ([AC04], the barrier designs above will likely not scale well with an increasing number of threads, since the barriers are designed to lead all threads through one shared point of memory.
I was unable to find any barrier design that removed the diminishing return of adding an increasing number of threads, but there are designs that diminish the negative effects.

To combat the scalability problems, many barrier algorithms employ some tree structure, wherein the main idea is to gather subsets of threads as to spread out the number threads per lock, thus decreasing the likelihood of a queue of processes forming behind any single lock. An example of such an approach is the "butterfly-barrier".
The butterfly-barrier([III12]) deals with the potential bottleneck by increasing the number of lock's used. Here each synchronization between each thread has it's own lock, meaning no two threads will wait on the same lock. There is still an increase in time needed to synchronize as the number of threads increase. Each thread must either via direct synchronization or another thread it previously synchronized with, reach all threads in the barrier. As illustrated here however, the number of steps needed is $log_2(T)$ where T is the number threads in the barrier.

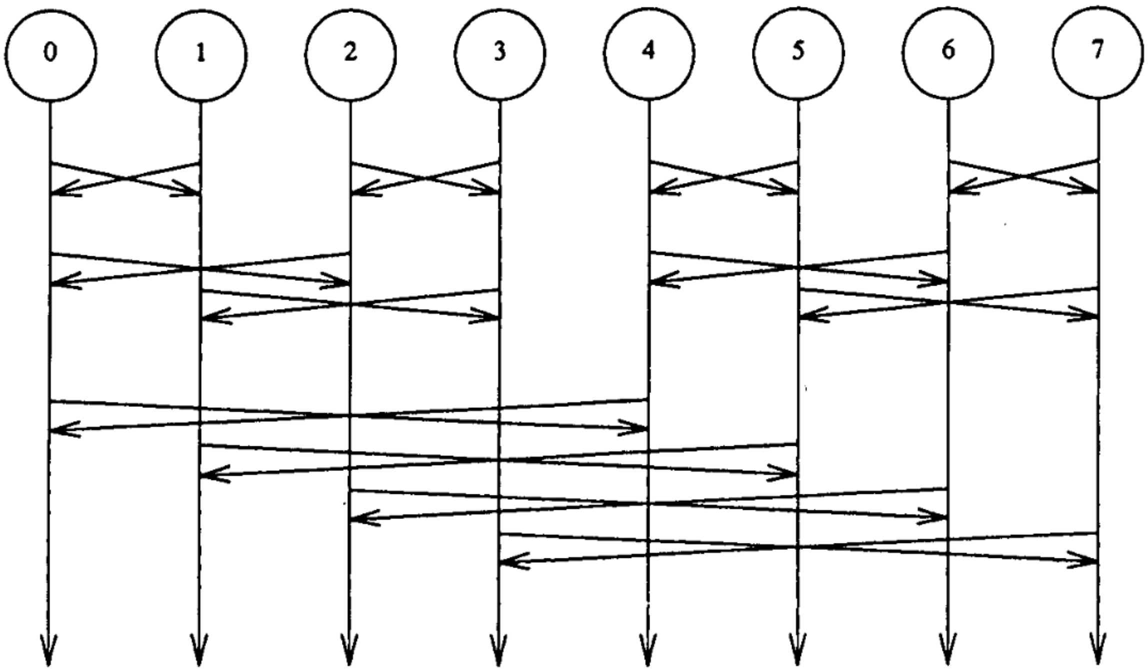


Fig. 3. Interaction pattern for the butterfly barrier. Time proceeds downwards.

Figure 3.2: A butterfly barrier, Figure found here: [Axe05]

Chapter 4

Analysis

As described in the introduction, there are no single correct approach for constructing a checkpointing library. How well it performs depends on what it is optimized for. A fully transparent checkpointing library will likely perform very different from a not-so-transparent library geared towards fast checkpoint construction and both methods are likely not as portable as a library geared for process migrations.
Thus, some specifying are needed to construct a checkpointing library. That is the aim of this chapter.

4.1 System or application level checkpointing

If a checkpointing library is implemented on the system level, it will require more space to store and impose a higher overhead on the application. In return, the user does not have to do anything as the solution is completely transparent.

The article [SS03] has looked into this trade-off. According to that, there is an overhead both in checkpoint file size required and in application run-time of roughly 20%-90%.
The overhead that comes with having complete transparency thus seems considerably.

4.2 Resuming code on restart

The intuitive thing to expect when restarting from a checkpoint, is that execution starts exactly from the point in the application that the checkpoint is made. For system level checkpointing, this is likely also the case, but not for application level checkpointing.

Application level checkpointing commonly relies on the initiation of the application for the initiation of threads and the allocation of memory, making this a necessary part of the restart.
Next, an application level checkpointing approach must consider when and how to resume the checkpoint. While it is not impossible to jump to instructions in C/C++ with calls like *goto*, *switch* and *if*, it is often left to the programmer to skip irrelevant sections on restart. Being partly responsible for skipping code on restart does come with a responsibility for the programmer.

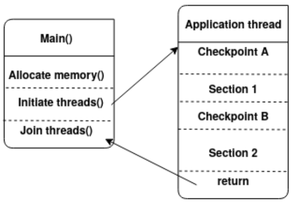


Figure 4.1: Example of a restarting application

Consider figure 4.1 with the following scenario: A previous run of the application created a checkpoint at **checkpoint B** and the application is not constructed to skip **checkpoint A** and **section 1**.
The best case scenario here is that the library is set up to recognize the existing checkpoint and simply does nothing on **checkpoint A**. It still have to recompute **section 1**, which is entirely wasted computations, since it already was stored in **checkpoint B**.
The worst case scenario here is that the library does not realize it is loading an unintended checkpoint, adding the checkpoint reading time on top of the best case scenario.

4.3 Constructing a checkpoint

Common for practically all papers proposing checkpointing approaches and checkpointing libraries encountered during the writing of this thesis, the approach chosen to construct the checkpoint has a large impact on the overhead checkpointing imposes on an application.

4.3.1 The practicalities of copying data

Data takes time to write to disk. An increasing amount of data requires a proportional amount of time to write. The following graph illustrates a small test to see if there are any difference in what library to use for copying data.

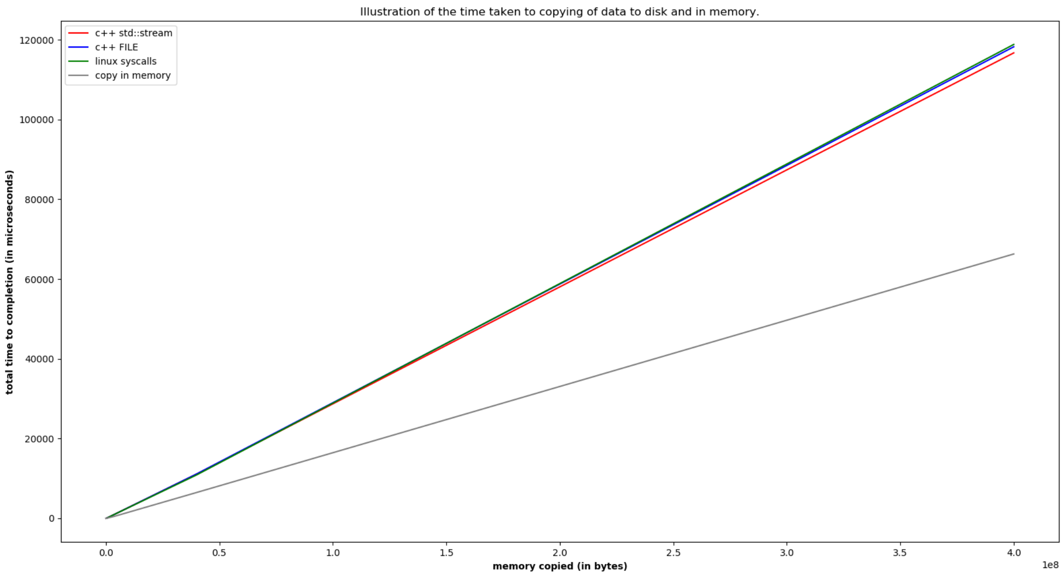


Figure 4.2: Timed copying of data from buffer to disk and in memory. Performed on an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz (2 cores). The test code can be found in the **checkpointlib** repository under benchmarks/IO_illustrated

Here the files and buffers were pre-allocated, meaning only the process of writing were measured. The three methods of writing to disk are more or less identical (stands to reason that FILE and stream implements the same syscall "write"), so there does not seem to be a difference.
Copying in memory (here using the function *memcpy*) is of course clearly the fastest approach and exemplifies why diskless checkpointing is an alluring approach.

4.3.2 Checkpoint file format

A checkpoint file can be saved either in it's entirety or as an incrementing model. Saving the entire checkpointing model is a simple approach, where all data linked to the checkpoint is saved at every checkpoint. An advantage of this approach is the freedom to either make a new checkpoint or continuously overwrite the same checkpoint as the application goes on, thus taking up a constant amount of space.

The other possible approach is the incremental approach. Here a checkpoint in an application is essentially the sum of every checkpoint taken up till that point. The main advantage here is that only the changed part of the checkpoint needs to be saved when at a checkpoint. The overheads of this approach that is hard to avoid is the additional computation time needed to restart a checkpoint. A challenge when dealing with incremental checkpointing, is "granularity" or minimum size to consider when tracking what memory to write to disk.
An example hereof is the solution proposed in [Tak+17]. A table of checkpoint memory is maintained. Each entry is marked if it was written to during an epoch and is saved on encountering the next checkpoint which resets all flags and they choose the granularity to be that of an OS memory page. They track memory that has been written to, using write protection and page faults.

4.3.3 Data structures to save/restore

The data structure pointer is problematic for checkpointing. Many data structures in C/C++ depends on pointers. It is one of the simple data types the language is build upon. It simply denotes an address in the process address space, which is often the beginning of a block of data (e.g an integer, making the pointer "a pointer to an integer").
The important thing to realize is that this address space is not necessarily the same. Even an application that is executed two times in succession is not guaranteed to have the same addresses. A pointers address is determined at run-time.

This proves problematic when constructing a checkpoint. There are approaches for saving such data structures like the previously mentioned library CRAFT ([Sha+17]) or the widely used Boost[1] library "Serialization". Seemingly a wrapper or dedicated method to save the pointer is needed to support the saving of pointers, meaning that a programmer would likely have to modify each class/data structure that is a part of the checkpoint as to enable this.

The trade off seems to be, that if an application level library should support inclusion of pointers in a checkpoint, the programmer must be made to do some customization to the application.

As it is basically the same problem, other system resources must be mentioned. For instance, an object representing an open file could also be relevant to include in a checkpoint. Considering these in addition to pointers seem to widen the scope of the thesis too much, so besides pointers, no saving of system resources will be considered in this report.

4.3.4 Shared mmap

A possible way to hide the costs of I/O is the shared memory map ([Ker19b]), where the OS synchronizes the changes done in memory with the underlying file. The synchronization is handled by the operating system and it makes for very efficient writing of data to disk.
A shared mmap is an intriguing possibility, since it almost ensures the checkpoint is continually updated with the latest computations. In practice upon an interruption of an application there would be no loss of computations between the latest checkpoint and the crash upon restart.
The downside is the small chance of error in the checkpoint if the crash happens during a write. The other downside is that even though the OS hides the writes and do them very efficient, they still do happen and there's probably a lot more of them than if it was just one write to disk per some epoc, so the OS will probably end up using more resources than would be necessary with a conventional checkpoint.

4.3.5 Two stage checkpointing

A way of hiding the cost of writing to disk is to split the checkpointing part into two stages. In the approach presented here [Sha+15], the first stage constructs the checkpoint in memory. The

[1] A very large collection of libraries for C++.

second stage then have another thread writing the checkpoint to disk. This approach enables the application to only having to wait on a checkpoint being constructed in memory.

This style of checkpointing is sometimes used in diskless checkpointing and offers a quick construction of the checkpoint in exchange for an increased memory usage.

4.3.6 When to save a checkpoint

Be it by setting an interval or placing explicit checkpoints directly in the application, there often seem to be the freedom for the programmer to decide how often a checkpoint is written to disk. There are models that can predict the optimal interval in which to save a checkpoint for a given application very reasonably ([Dal03]). If the MTTI (mean time to interrupt)[2] is known to the programmer for a given system, along with the expected[3] run-time, the "optimal" interval for checkpoints could even be computed at run-time ([FRL12]).

4.3.7 Corrupted checkpoints

To have a thread or indeed the entire application crash while writing a checkpoint to the disk is a possibility that any checkpointing library needs to consider.

The simplest of safeguards against such an eventuality is to simply create a new file every time a checkpoint is written to disk. In case of a corrupted checkpoint, it is still possible to restart from the previous one and only loose one epoch of computations.

Finally, there are tools and techniques used to limit the impact of corrupted checkpoints in checkpointing libraries, e.g the tool "ReCov" ([Li+15]). As the **CheckpointLib** ended up relying on the "simple safeguard", exploring these tools and techniques is left as potential future work.

4.4 Formulating the aspirations for checkpointlib

Considering the research question for this thesis along with advantages and disadvantages of this chapter, the following will be the criteria I've chosen to pursue:

- Use as few system resources as possible, both memory and disk space.
- As few lines of code or effort by the programmer needed as possible.
- Optimize the library for saving a checkpoint.

4.4.1 The ideal application for checkpointlib

While the library is a general purpose checkpointing library, it is designed fo fit especially well with applications that are more or less built up like this:

[2] sometimes also called "MTTF" (mean time to failure).

[3] here "expected" refers to a run without any crashes or interruptions of the application.

- **Initiation:**
    - Allocate necessary memory for execution.
    - Initiate threads, all with the same application, but different arguments supplied. E.g each thread is trying to find prime numbers, but is initiated with different seeds.
- **During Execution:**
    - Each thread's work is easily divided into roughly equal sized "sections" of executions. E.g a loop that attempts a new prime number each iteration.
    - Each thread has roughly the same number of "sections".
    - Each thread has roughly the same amount of work/execution time.
- **Post execution:**
    - Join all threads.
    - Either end computation here or start a new cycle of execution.

This application structure has two advantages:
It keeps memory allocations outside the parallel section, since restarting with memory allocated inside threads is messy.
If each thread is divided into roughly equal sections of work, e.g via a loop, it is very simple for the programmer to save the loop iterating variable and thereby resuming execution exactly where the previous run let off on a restart.

4.4.2 Limitations

Since checkpointing covers many areas of research and there are multiple ways of implementing it, some limitations to the library is in order as to focus the scope of the thesis and the coding needed in the library.

The library is written for a Linux OS. First reason being that the library relies on system calls, e.g *memprotect* ([Ker19c]), for central functionalities of the library and it would not be trivial to implement in another system. Furthermore, no guarantees are made that the **checkpointlib** can be implemented on other types of OS.

The library will only allocate memory in a single continuous block. E.g if there is technically available space on a system for the memory requested by an application, but the needed memory cannot be found in a single block of memory, the library will not make the allocation. This limitation is set to limit the amount of code needed in the library as well as making the save/restart simpler.

The library does not offer support if the programmer wishes to save a C/C++ pointer type variable. Since the library's checkpoint-data can be filled with anything the programmer wishes, nothing is stopping a pointer from being written there, but in case of a restart, it is likely that the process running the application has been assigned a new offset in the address space, making the saved pointer invalid.

When describing recovery from errors in this thesis, only errors of the type that causes the entire process to exit is considered (e.g the process encountered a problem that caused it to stop itself, a hardware problem occurred or something third).

Chapter 5

Implementation

checkpointlib is an application level library that has a near constant memory and disk-space usage over the course of executing an application. It maintains two checkpoint files, one for redundancy if a crash occurs during writing to disk and only supports of restart to the latest created checkpoint. It is left to the programmer to place checkpoints in the application along with making the application suitable for restart[1]. The checkpoints will synchronize, but not block on the creation of a new checkpoint and it will synchronize and block on restart.
Due to the way that C/C++ handles overwriting of the segfault handler, there should only be one instance of **checkpointlib** active in a process at any time, otherwise the behavior is undefined.

5.1 How to run checkpointlib

There is next to no preparation needed to use **checkpointlib**. The essentials is written here, for the full documentation, see appendix **A** or the **checkpointlib** repository.
For examples, benchmark-applications and unit tests for the entire library, see the **checkpointlib** repository.
The library itself is in the **checkpointlib** repository under lib/.
checkpointlib has been developed and tested using the C++ compiler *g++*.
In case a programmer wants to use **checkpointlib** in a file called "main.cpp", it could as minimum include the following line:

```
#include "checkpoint.hpp"
```

"main.cpp" can then be compiled using the following line in a shell:

```
g++ main.cpp -std=c++11 -lpthread
```

Nothing more is required.

5.2 General work flow

Following is an explanation of how the parts of **checkpointlib** is implemented and how to make the most of it.

[1] Here referring to what was mentioned in chap 4.2.

5.2.1 The checkpoint file layout

The presence of two checkpoint files is ensured by the **Checkpointlib** in the initiation phase of the application. The reason for two checkpoints is to guard against a crash during writing, i.e the checkpoint files with the oldest checkpoint stored is the next file to be written to.
The following is an explanation of how **Checkpointlib** manages checkpoint files.

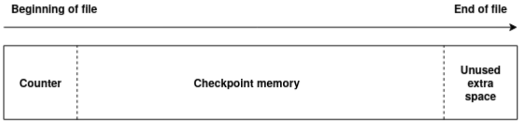


Figure 5.1: A example of the layout of a checkpoint file generated by **checkpointlib**

At the beginning of a checkpoint file is the counter. The size of the counter depends on how large the data type *size_t*[2] is defined as on the system that generated the checkpoint file. The counter denotes which of the checkpoint files has the most recent checkpoint stored and each time a checkpoint finishes the old checkpoint file's counter is incremented by two, making it the newest. To ensure that a frequently used checkpoint file does not exceed the capabilities of *size_t* (though it is hard to imagine), the checkpoint file's counter are reset during each initiation phase. I.e the checkpoint file with the higher counter gets set to one and the other file's counter is set to zero.

The section of the checkpoint file belonging to the checkpoint memory is exactly the size reserved by the programmer during initiation.

If one or both of the file-paths provided to **checkpointlib** points to a preexisting file which size exceeds what the library needs (i.e *sizeof(size_t + checkpoint memory)*), that space is left untouched. The library will never downscale a checkpoint file, only expand it if it is not sufficiently large.

As a consequence of the above, if attempting a process migration, the programmer should take care not to mix checkpoint files from different runs. It would be regrettable if the desired checkpoint file had a lower counter than the other one.

5.2.2 The checkpoint memory layout

Recall algorithm 1, which is the method used for managing the checkpoint memory. Much of the manager is implemented directly into the memory itself. This has the disadvantage that the programmer need to ask for more memory than is strictly needed and the advantage that the manager takes up less memory than if it was a separate structure.
The following is an example of how the memory manager works

[2] *size_t* is commonly used to contain largest possible unsigned integers supported by the OS in that implementation of C/C++.

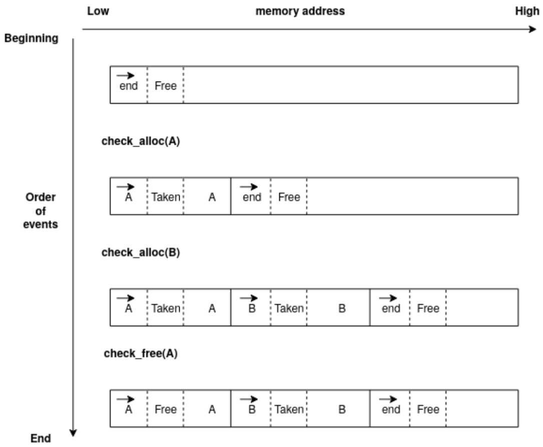


Figure 5.2: An example of the layout of the checkpoint memory

The memory is divided into blocks. The first part of a block consists of a number, which in algorithm 1 was a pointer, but in this application is an offset (where offset=0 is the start of the checkpointing memory)[3]. This offset denotes the start of the next block and the offset is in figure 5.2 denoted by an arrow atop a descriptor of where the next block begins.
The next part of the block is a flag denoting whether the block is free or taken, i.e if a block is free, it can be allocated from. The remainder of a block is the actual usable memory of the block.

Before the first allocation, the entire memory is free. Note the dotted lines marks the separation of values within the same block. There is only one block in the beginning, the offset points to the end of checkpoint memory and the block is marked as free.

When a memory chunk of size **A** is allocated, the first(and only) block that has memory necessary, marks the block and makes a new block consisting of the remaining memory after **A** is subtracted.

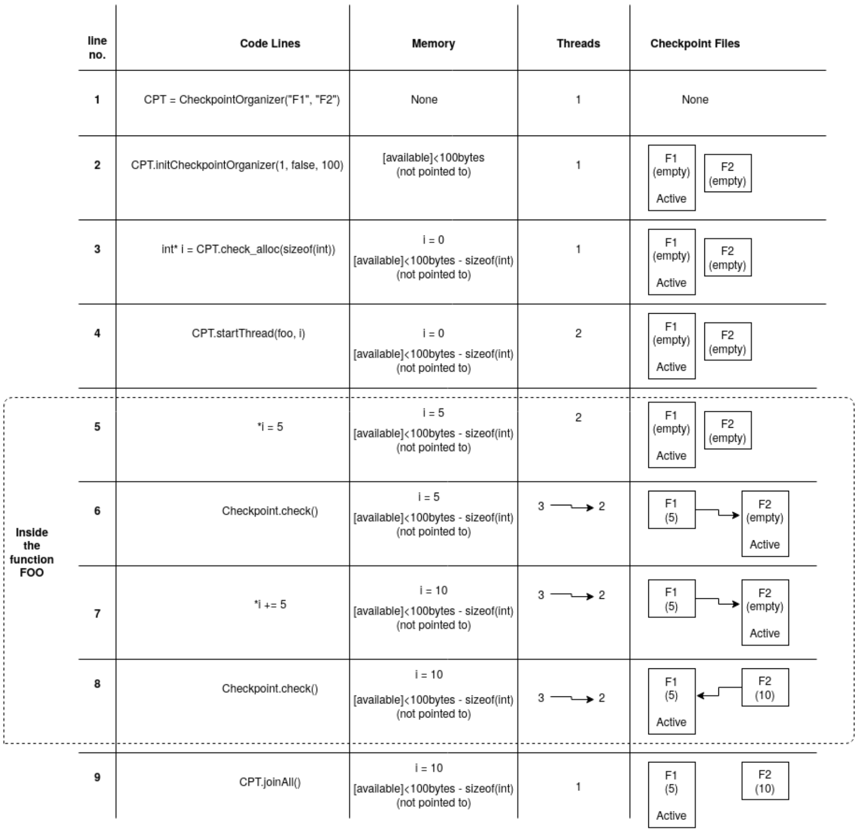
After the chunk **B** is allocated same as before, the block containing **A** is now freed. Note that freeing **A** does not mean that the block disappears, but rather is marked as free again. recall segmented memory from chapter 2.

As shown, there is an overhead every time a new allocation is made. The overhead is precisely *sizeof(size_t) + 1* bytes per allocation. If a block that is being freed borders other free blocks, two blocks can melt into one, releasing the memory used for offset and the flag. It is not much extra memory per allocation, but a programmer with a very memory intensive application might want to keep this overhead in mind.

[3] This can work with offset instead of pointers, exactly because the checkpoint memory is always a single unbroken block of memory

5.2.3 Running checkpointlib, what resources are used when?

An illustration of when the **Checkpointlib** uses what resources and an explanation follows



line no.	Code Lines	Memory	Threads	Checkpoint Files
1	CPT = CheckpointOrganizer("F1", "F2")	None	1	None
2	CPT.initCheckpointOrganizer(1, false, 100)	[available]<100bytes (not pointed to)	1	F1 (empty) Active F2 (empty)
3	int* i = CPT.check_alloc(sizeof(int))	i = 0 [available]<100bytes - sizeof(int) (not pointed to)	1	F1 (empty) Active F2 (empty)
4	CPT.startThread(foo, i)	i = 0 [available]<100bytes - sizeof(int) (not pointed to)	2	F1 (empty) Active F2 (empty)
5	*i = 5	i = 5 [available]<100bytes - sizeof(int) (not pointed to)	2	F1 (empty) Active F2 (empty)
6	Checkpoint.check()	i = 5 [available]<100bytes - sizeof(int) (not pointed to)	3 → 2	F1 (5) → F2 (empty) Active
7	*i += 5	i = 10 [available]<100bytes - sizeof(int) (not pointed to)	3 → 2	F1 (5) → F2 (empty) Active
8	Checkpoint.check()	i = 10 [available]<100bytes - sizeof(int) (not pointed to)	3 → 2	F1 (5) Active ← F2 (10)
9	CPT.joinAll()	i = 10 [available]<100bytes - sizeof(int) (not pointed to)	1	F1 (5) Active F2 (10)

Figure 5.3: A step-by-step tour through the libraries use of system resources on a small example

Here is a walk through of figure 5.3 line-by-line to explain what is going on.

1. Here the *CheckpointOrganizer* class is initialized. It is initialized with paths to where the checkpoint files should be stored (in this case in the directory of the calling process) and what to name them (here they would be named "F1" and "F2"). The only thread running is that of the process itself.

2. The call to *initCheckpointOrganizer* allocates the requested amount of memory (here 100 bytes). As explained previously, not all 100 bytes are available to the programmer, but close. In this example, no checkpoint files existed previously, so the files "F1" and "F2" are created. Also, since no checkpoint files previously existed, "F1" is selected to be the first file to copy the checkpoint memory to ("F1" is marked *active* to signal this).
   Finally, note that the argument for the total number of threads is being set to one.

3. *check_alloc* here allocates the space required for an integer. To simplify the model it is typecast to an int pointer instead of the usual void pointer. Note the pool of available memory for *check_alloc* has shrunk.

4. *startThread* starts a new thread. In this case it starts the function "foo" with the integer that was previously allocated as an argument. Note there are now two threads, the main thread and the one running "foo".

5. Here inside the function "foo" a value is being written to the variable previously allocated.

6. The class **Checkpoint**, automatically passed to "foo" when it was called with *startThread*, calls it's only function **check**. Here it would have synchronized with other threads started by *startThread*, but in this example it is the only one. The call to **check** starts a thread that copies all the checkpoint memory to a checkpoint file. Note the arrows in figure 5.3 denotes what state it was when the checkpoint-thread started and in what state it will be when it has completed it's task.
   There are three threads now, but there will be two again as soon as the checkpoint-thread is done. "F2" will be the active checkpoint file as soon as the checkpoint memory is written to "F1".

7. Another change in checkpoint memory. Note that the checkpointing-thread is not guaranteed to be done before writes to checkpoint memory can occur. If the area pointed to by the variable "i" had yet to be written, the thread running foo would be put to sleep until it had been written.

8. Another call to **check**. The previous call to **check** could have been running parallel with the thread running "foo" till this point, but **check** is guaranteed to block until the previous checkpoint-thread has finished it's job. Therefore, though it looks the same, it is a new checkpoint-thread that is running after this call. Also note the active checkpoint file has shifted back to "F1", so were **check** to be called again, the content of that checkpoint file would be overwritten.

9. Returning to the main thread that has been sleeping while the thread running "foo". **joinAll** joins all the threads started by *startThread* back into the main thread.

5.2.4 How checkpointlib writes a checkpoint to disk without blocking the application

While a checkpoint is being written to disk, all threads of the application are free to continue computing. The **checkpointlib** makes this possible by replacing the process segfault handler[4].
If a segfault happens outside the address space that makes up the checkpoint memory, the segfault handler acts as the default handler would. If the segfault is raised on a memory address that is within the checkpoint memory, the thread is put to sleep for a short while and will then attempt to re-run the instruction that made it segfault. In practice, this means that the thread will repeat that instruction forever, unless it is at some point allowed to write to that address.

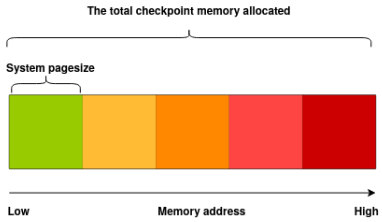


Figure 5.4: The checkpoint memory illustrated during a checkpoint being written to disk

[4] A segfault, or segmentation fault, is a type of error raised when a thread/process attempts to access an address in memory for an action it is not allowed to.

When the writing of a checkpoint to disk is being initiated, the first step by the **checkpointlib** is to add write-protection to the entire checkpoint memory. I.e if the application attempt to alter the checkpoint memory, it triggers a segfault.
Note in figure 5.4 that checkpoint memory is a multiple of the system specific variable *pagesize* ([Ker19d]). Specifically, *pagesize* is used as the multiple for the size of the checkpoint memory, because the syscall used to set the protection of the checkpoint memory (found here: [Ker19c]) sets memory protection in chunks of "*pagesize*" size.

The **Checkpointlib** then periodically takes a chunk of size *pagesize* and writes it to disk, starting from the lowest address. In figure 5.4, this is depicted as the green chunk being the first one and then chronologically going through to the dark-red one.
After a chunk has been written to disk, the write protection for that chunk is lifted.

As a consequence of the behavior described above, then for a programmer to utilize the library to the greatest efficiency, it is beneficial to plan out what part of the checkpoint memory is used when. In the span of a epoch, writing to the green area is less likely to put the thread to sleep, than a write to the red area. In general, the more writes to checkpoint memory that can be placed in the end of an epoch, the less likely it is that the checkpointing thread has not yet written that chunk to disk and therefore that the thread is put to sleep.

Regardless of how many chunks have been written to disk, the application always has read access to the checkpointing memory. That is in the beginning of an epoch and all through to the end.

5.3 Core API

5.3.1 CheckpointOrganizer

The core of the library, the CheckpointOrganizer class takes two system paths as arguments. These paths can point to none, one or two checkpoint files. If there is no such file where the path points, this is where the new checkpoint file will be created.
There is no requirement for the checkpoints to be made on the machine running the application. E.g to migrate a checkpoint from another machine, the file can be copied to the current machine and no other steps are needed.
This should always successfully return an instance of the 'CheckpointOrganizer' class.

initCheckpointOrganizer

This call sets up the behavior of the library.
It takes as argument the number of threads that the programmer intents to start. As previously discussed, this library relies on barrier synchronization, meaning that if fewer threads are initiated, the application will deadlock.
It accepts an argument that decides whether the application should commence with a restart or only create new checkpoints in that run.

This call also allocates all memory, creates/expands checkpoint files if needed and runs many sanity checks. If this call returns successfully, the programmer can be fairly confident that the library runs correctly.

check_alloc

Functions like malloc from C/C++, except, this call is limited to the block of memory allocated in "initCheckpointOrganizer". returns a pointer to a chunk of memory of the requested size.

startThread

Is called to start a thread. It is best thought of as a wrapper for std::thread from C++ and all threads that are to be a part of the checkpointing must be started with this function. The number of threads started with this function should match the number supplied in "initCheckpointOrganizer".

5.3.2 Checkpoint

With each thread started by "startThread", a member of this class is passed along.

check

Each time this is called, the library either starts to save or restore a checkpoint. Note that the number of calls to "check" must be the same for all threads started by that instance of 'CheckpointOrganizer' or a deadlock could occur.

5.4 Use-case examples

5.4.1 Basic example: Generate prime numbers

In this example, two threads are started, each tasked with finding 100 random primes. Note that whether the **Checkpointlib** should restart is implemented as an argument passed to the function when the process starts.

```
#include "checkpoint.hpp"
#include <stdlib.h>
#include <iostream>
#include <unistd.h>
#include <time.h>

int main(int argc, char** argv){
  if(argc != 2) return 0;
  bool restart = 0;
  if(atoi(argv[1])) restart = 1;
  unsigned int tot_num_threads = 2;
  CheckpointOrganizer cptOrg("tmp1.
    chpt", "tmp2.chpt");

  cptOrg.initCheckpointOrganizer(
    tot_num_threads, restart, sizeof(
    pBuff) * 2 + 100);

  void (*foo)(void *, Checkpoint*);
  foo = &gen_prime;

  pBuff* var1 = (pBuff*)cptOrg.
    check_alloc(sizeof(pBuff));
  var1->count = 0;
  pBuff* var2 = (pBuff*)cptOrg.
    check_alloc(sizeof(pBuff));
  var2->count = 0;
  cptOrg.startThread(foo, var1);
  cptOrg.startThread(foo, var2);

  cptOrg.joinAll();
  std::cout << "Process ran to finish
    \n";
  return 0;
}
```

Listing 5.1: gen_primes.cpp - part 1/2

```
struct pBuff{
  unsigned int count;
  int prime[100];
};

void gen_prime(void* raw_data,
    Checkpoint* cpt){
  pBuff* lBuff = (pBuff*) raw_data;
  srand(time(0));
  int tmp, halfway;
  unsigned int i;
  bool flag;
  while(100 > lBuff->count){
    tmp = rand();
    flag = true;
    halfway = tmp / 2;
    for(i = 2; i <= halfway; i++){
      if((tmp % i) == 0){
  flag = false;
  break;
      }
    }
    if(flag){
      lBuff->prime[lBuff->count] =
    tmp;
      lBuff->count += 1;
      if((lBuff->count % 10) == 0){
  std::cout << "@ checkpoint\n";
  cpt->check();
      }
    }
  }
}
```

Listing 5.2: gen_primes.cpp - part 2/2

The function "gen_prime" has here been set up as to take a checkpoint for each ten prime numbers it finds. Note that the variable "count" is stored in the checkpoint-memory. Say a checkpoint exists that has found 60 prime numbers and the application is run with "restart=true". This implementation would need to find ten primes, go to "cpt->check()", overwrite both the current count and the ten discovered primes with the stored 60 primes and "count" from the checkpoint and at the start of the next iteration of the loop, "count=60".

It is also easily spotted, that were the call to "cpt->check()" encountered at "count=0", there would hardly be any wasted computation on restart. The downside here being a wasted checkpoint in the beginning of a run without a preexisting checkpoint.
While worst case for careless placement of checkpoints probably only being the loss of one epoch, it can accumulate (e.g if the application needs frequent restarts) and it is something for the programmer to be mindful of.

The example can be found in the **checkpointlib** repository under examples/gen_primes

5.4.2 Recovery example: counting numbers, automatic recovery

In this example, two threads are started, each tasked with counting up and is set to exit (i.e kill the process) just shy of half the needed iterations. Note that the option to restart is passed as an argument like the previous example.

```
#include "lib/checkpoint.hpp"
#include <stdlib.h>
#include <iostream>
#include <unistd.h>


void fail_halfway(void* raw_data,
    Checkpoint* cpt){
  std::thread::id this_id = std::
    this_thread::get_id();
  unsigned int mean_time_to_failure =
     5;
  unsigned int count_to_mttf = 0;
  int* data = (int*)raw_data;
  for(int* i=&data[0]; (*i)<10000; (*
    i)++){
     data[1] += 2;
     if((*i % 1000) == 0) {
       std::cout << "thread_id: " <<
    this_id
    << " - iteration: "
    << *i << "\n";
       cpt->check();
       count_to_mttf += 1;
       if(mean_time_to_failure <=
    count_to_mttf){
  _exit(10);
       }
     }
  }
}
```

Listing 5.3: application.cpp - part 1/2

```
int main(int argc, char** argv){
  if(argc != 2) return 0;
  bool restart = 0;
  if(atoi(argv[1])) restart = 1;
  unsigned int tot_num_threads = 2;
  CheckpointOrganizer cptOrg("tmp1.
    chpt", "tmp2.chpt");
  cptOrg.initCheckpointOrganizer(
    tot_num_threads, (bool)restart,
    1000);

  void (*foo)(void *, Checkpoint*);
  foo = &fail_halfway;

  int* var1 = (int*)cptOrg.
    check_alloc(2*sizeof(int));
  var1[0] = 0;
  var1[1] = 0;
  int* var2 = (int*)cptOrg.
    check_alloc(2*sizeof(int));
  var2[0] = 0;
  var2[1] = 0;
  cptOrg.startThread(foo, var1);
  cptOrg.startThread(foo, var2);

  cptOrg.joinAll();
  std::cout << "Process finished\n";
  return 0;
}
```

Listing 5.4: application.cpp - part 2/2

```
#include <stdio.h>
#include <string>
#include <iostream>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(){
  std::string cmd_no_restart = "./testApp 0";
  std::string cmd_with_restart = "./testApp 1";

  FILE* recoverable_process = popen(cmd_no_restart.c_str(), "r");
  bool continue_execution = true;
  int status;
  int exit_status;
  while(continue_execution){
    wait(&status);

    if(WIFEXITED(status)){
      exit_status = WEXITSTATUS(status);
      if(exit_status){ // if exit status is non-zero => error occurred
  std::cout << "restart was called \n";
  pclose(recoverable_process);
  recoverable_process = popen(cmd_with_restart.c_str(), "r");
      }else{
  continue_execution = false;
      }
    }
  }
  pclose(recoverable_process);
  std::cout << "success\n";
  return 0;
}
```

Listing 5.5: main.cpp

In this example, there is a local variable in "fail_halfway" called "mean_time_to_failure" and is meant to simulate what it is to face a task where one or more failures are expected before the application is finished.

Here application.cpp is compiled to the binary named "testApp". What is needed for automatic recovery is a process that monitors how the application is faring. "main.cpp" is possibly the simplest structure that achieves this. Leveraging that any other status code than zero[5] means the application wasn't finished and should be restarted.
Since the iterator in the application is stored in checkpoint memory, it can then be restarted the number of times needed to reach the desired count, increments beyond the MTTF limit. Note that the line "wait(&status);" puts the calling process to sleep, so the overhead of using this strategy is practically nothing.

While this approach is meant to demonstrate automatic crash recovery, it should also be noted that this approach is essentially also a simple implementation of what in the introduction was described as "message passing application checkpointing". The **checkpointlib** is meant to be a shared memory checkpointing library running in a single process, but can be expanded to be a simple asynchronous message passing checkpointing model.

The example can be found in the **checkpointlib** repository under examples/recovery.

[5] Standard convention in C/C++ and what Linux OS'es expect processes to exit with.

5.4.3 Process migration

To migrate a process from one machine to another, simply copy the checkpoint file. The only requirement is that the system variables between the machines are the same size, e.g *sizeof(std::size_T)* are the same on both machines.

5.5 Benchmarks

In performing benchmarks for **checkpointlib**, note that what operations the applications performs is not of any importance. It all boils down to the epoch time and the size of checkpoint memory.

5.5.1 Framework

All benchmarks were made on a machine with the following specs:
Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz (2 cores).
The operating system is Ubuntu, version 18.04
8GB of available memory.

5.5.2 Performance in a favorable scenario

Included in appendix B is the applications there were tested[6] the relevant ones for this benchmark being *array_test_checkpoint()* and *array_test_control()*.
In short: Using four threads, 100x200000 2D array of integers and a checkpoint-memory array of 200000 integers per thread. The functions simply had to generate a random array of int's, by picking different multiples of itself and multiplying anew.
This is an illustration of a near optimal use case for the library since the writing to the checkpoint-memory is set to just before the checkpoint and immediately after the checkpoint, a large read-section follows without writes to checkpoint memory until the end of the epoch.
With 1000 trials, this resulted in an average time of 10.8333 seconds without the checkpoint library and an average time of 10.9753 seconds with the library. Meaning about a 1.0% overhead of using the library.

5.5.3 Illustration of inefficient scenarios

As mentioned, this library writes checkpoint memory to disk by writing a section at the time and freeing the read-only restriction on that memory upon completed write. Recall from earlier that when a write happens to an address that is yet to be written to disk, the thread is put in a sleep loop until the write is successful. As a consequence, the programmer can expect a performance difference between writing to the beginning of the checkpoint-memory versus the end.

[6] The full test code can be found in the **checkpointlib** under /benchmarks/random_array

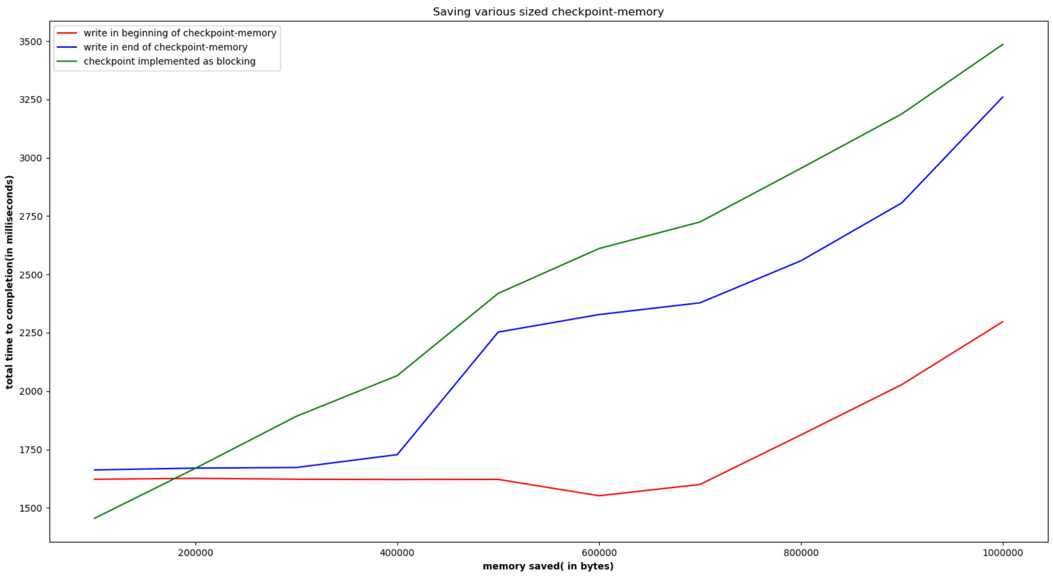


Figure 5.5: An illustration of the difference between taking advantage of the libraries background saving feature or not, using a function that only writes in the beginning of the checkpoint memory, one that writes in the end and a thread started by a modified version of the **checkpointlib** that blocks on each checkpoint until it is committed to memory. Four threads were running in this test. The experiment was repeated 100 times. The test code can be found in the **checkpointlib** under /benchmarks/blocking_vs_optimal

Note that even for an application that takes full advantage of **checkpointlib**'s method of saving checkpoints, there is a potential inefficiency to consider:
The applications that the threads displayed in the graph are running, does a set amount of writes to the checkpoint memory, regardless of the size of the checkpoint memory. I.e the checkpoint memory increases, but the length of the epoch stays the same. It can be seen that even if the optimized application can use more of the epoch to write to disk, it is ultimately limited by the length of the epoch and the size of the checkpoint memory.

5.5.4 Cost of initiating a checkpoint

The following benchmark was taken on a application that simply inverts an array of integers. the array resides in checkpoint memory. the application resumes from a premade checkpoint at the halfway point of the list inversion and then returns as not to measure the application itself (i.e the run-time of the application after restart is completed). This benchmark was taken using four threads and illustrates the increase in time needed for restart as checkpoint memory grows.

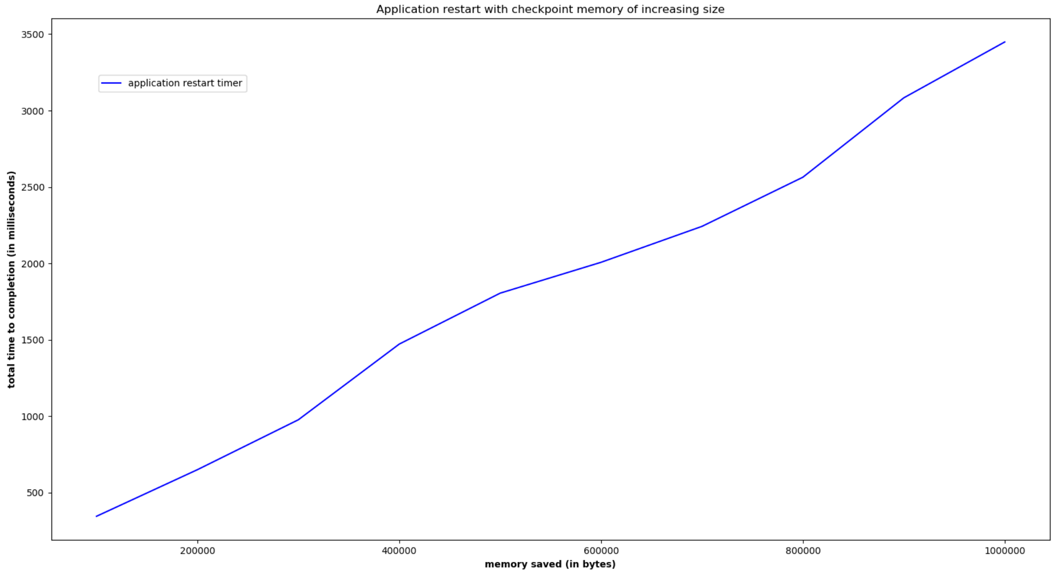


Figure 5.6: Graph illustrating the increased time taken to restart a application as checkpoint memory increases. 100 trials were run. The test code can be found in the **checkpointlib** under /benchmarks/restart_benchmark

5.6 Discussion

In the favorable scenario, The library caused an overhead of 1.0%. The meaning of favorable here should be understood as in that the library did not have to perform any of it's segfault-stalling. High precision was was especially important in this benchmark, but as the nature of this benchmark is to see the performance in a near ideal scenario, the epoch has intentionally been made rather lengthy. making an increased number of trials even more time-consuming. In this benchmark, 1000 trials were possibly not high enough.
Regardless, all of the I/O were in this case "hidden" and the overhead that remains probably comes from the following three sources.

The barrier lock itself that is used by the library is bound to be responsible for some of the overhead. The library uses a two-lock barrier ([Axe05]), which as previously discussed will decrease in efficiency as the number of threads increases, but for this case with only four threads, the effect is probably limited.

The library initiates a thread each time a checkpoint is written to memory. The initiation is bound to impact the overhead, not only by the instruction itself, but because it happens inside the critical section of the barrier. As previously mentioned, the critical sections should be kept as small as possible.

During the writing of the checkpoint, the OS is sharing the CPU with an extra thread that is constantly working, meaning there is simply less time for the remaining threads. While the previous two reasons are interesting due to possibly alternative implementation strategies, this reason likely makes up the minimum overhead possible for this library.

The attempt of measuring the effect of the barrier lock and the cost of starting a new thread were actually attempted, the applications also being included in appendix **B**. But when running a 1000 trials, the differences proved too small to measure adequately. Those are therefore not included. A better benchmark application and more iterations are likely needed.

Looking at the inefficient scenario; If the programmer does not make use of the attributes of **checkpointlib**, it essentially blocks each checkpoint until all is written to disk and the overhead imposed by the library becomes limited by the speed of writing to disk.
The reason as to why the test case that writes to the high end of the checkpoint memory is a bit more efficient is likely that it's threads are out of the lock when they start to sleep, whereas the blocking test does it's writing inside the critical section of the barrier.

Even the test that writes in the beginning of the checkpoint memory, has increased overhead as the checkpoint memory increases. It seems, that knowing the time it takes to write data to disk and the time it takes to complete an epoch is key to utilize a library like **checkpointlib** to it's fullest.

Finally, relating to the benchmark for restart. On restart, **checkpointlib** simply blocks until all data is read from the checkpoint file and into the checkpoint memory. As expected, the time taken to restart a checkpoint grows proportionally with the size of the checkpoint memory.

Chapter 6

Future Works

6.1 A new barrier lock implementation

The barrier lock that is currently implemented is the two-lock model, which is likely the simplest barrier lock there is. With few threads it is not an issue, but for it to be better suited to a larger number of threads, a barrier lock such as the butterfly barrier ([III12]) from chapter 3 should replace the "two-lock" barrier.

6.2 An appropriate sleep model for threads caugth in the segfault mechanic

As is, **checkpointlib** has a rather naive approach to the threads that enters the segfault-sleep loop. The thread is put to sleep for a static amount of time and this is repeated untill the address has it's write-protection lifted.
It is likely not trivial to determine the best way to minimize the amount of time that threads recides in the loop. A priority queue that is updated as threads enters the segfault loop seemes a possibillity, as does experimenting with exponential backoff timers in the loop itself like in the 2 algorithm.
Regardless, more sofisticated test-cases is needed to test this, that was provided in this report.

6.3 Expanding the memoryManager

As is now, there is little reason for the library to contain logic that manages the checkpoint memory, nor is there any note-worthy disadvantages. The intend back in the beginning of developing the **checkpointlib** was to support the allocation and freeing of checkpoint memory at any point during the application, even in the threaded part of execution. While this turned out to be less descireable largely due to the responsibillity it would put on the programmer[1].
The support of allocations of multiple blocks of memory as checkpointing memory, lifting the requirement of a single block of continuous memory in the system and opholding the illusion of continuity through the memoryManager would be a welcome addition to this library.

[1] The initiation, especially during a restart where there may be skipped section of code, can become very difficult to get right.

6.4 Incorporating segfault and memory protection to enable non-blocking restart

It seems a interesting to look into the possibillity of **Checkpointlib** using the same approach it uses for creating checkpoint files, to restart applications. It would have to be read-protected memory instead of write protected memory, which the syscall mprotect ([Ker19c]) does not support on all system architectures[2].

[2] e.g using the syscall to set write-protection is synonomous with setting read-protection on architectures like i386.

Chapter 7

Conclusion

For this thesis, the objective was to create a checkpointing library in C/C++ to be used in a single address space. The proposed answer to that objective is **checkpointlib**, a checkpointing library written for C++ and a Linux OS.

The research field of checkpointing is wide and depending on which priority one finds more interesting, the resulting library from the same research question could very reasonably had been completely different. The priority of space and memory efficiency drove this thesis down the path of application level checkpointing.
checkpointlib maintains exactly two checkpoint files during the run of a application for near constant disk-space-usage, it hardly has an increased memory usage during a checkpoint creation and the library's API is intended as to be a one-to-one replacement of the code needed to run a corresponding application without checkpointing.

During research and implementation of the library, there never came a point where it became important that the library should function specifically on a server environment, though it was specified in the original research question. As such, the library functions on practically all Linux based machines.

A programmer using **checkpointlib** has great freedom to choose how much checkpoint memory is needed and how often to take a checkpoint. As have been shown, this freedom can impact the performance greatly and the responsibility to make the most of the library is left to the programmer.

checkpointlib is geared towards minimizing an overhead on checkpoint creation in a application. Considering the checkpoint memory is written to disk, the imposed overhead on checkpoint creation ended up being very small in favorable scenarios and the overhead of restarting a checkpoint ended up being proportional the the checkpoint memory as one would expect.

While the library is perfectly capable of functioning as is presented in this thesis, some areas of interest appeared that could very well improve the usability and efficiency of **Checkpointlib** like:

- Using a more sensible barrier design (e.g the "butterfly" barrier).
- Expanding the memory manager as to enable utilization of more than one unbroken block of memory at the time.
- Investigate the impact of both the currently used and different approaches of threads being caught in the library's segfault handler.

In conclusion. **Checkpointlib** is a fully capable application level checkpointing library.
This thesis has used the approach of hiding I/O costs with write-protected memory and a segfault handler, shown it to be efficient library with a viable method for creating checkpoints. Furthermore, there is still potential and room for improvement.

Bibliography

[AA03] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems. Three Easy Pieces.* Version 0.09. 2015-03.

[AC04] Anant Agarwal and Mathews Cherian. “Adaptive backoff Synchronization Teckniques”. In: *ACM SIGARCH Computer Architecture News* (1989-04). URL: https://www.sciencedirect.com/science/article/pii/016781918690030X (visited on 05/06/2020).

[Axe05] Tim S. Axelrod. “Effects of synchroization barriers on multiprocessor performance *”. In: *Parallel Computing 3* (1986-05). URL: https://www.sciencedirect.com/science/article/pii/016781918690030X (visited on 05/05/2020).

[Cor07] Oracle Corporation, ed. *VirtualBox*. 2007. URL: https://www.virtualbox.org/ (visited on 05/10/2020).

[Dal03] John Daly. “A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps”. In: *Computational Science – ICCS 2003* (2003). URL: https://link.springer.com/chapter/10.1007/3-540-44864-0_1 (visited on 05/10/2020).

[Dow16] Allen B. Downey. *The Little Book Of Semaphores. Version 2.2.1.* 2016. URL: http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf (visited on 05/04/2020).

[FRL12] Leonardo Fialho, Dolores Rexachs, and Emilio Luque. “On the Calculation of the Checkpoint Interval in Run-Time for Parallel Applications”. In: (Apr. 2, 2012). URL: https://www.researchgate.net/publication/266223822_On_the_Calculation_of_the_Checkpoint_Interval_in_Run-Time_for_Parallel_Applications/link/54b3ef460cf28ebe92e43d38/download (visited on 05/11/2020).

[Gio+12] Roberto Gioiosa, José Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. “Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers”. In: *IEEE Xplore* (2005-12). URL: https://www.researchgate.net/publication/4204631_Transparent_Incremental_Checkpointing_at_Kernel_Level_a_Foundation_for_Fault_Tolerance_for_Parallel_Computers (visited on 05/10/2020).

[III12] Eugene D. Brooks III. “The Butterfly Barrier”. In: *International Journal of Paralelle Programming, vol 15, no 4* (1986-12). URL: https://link.springer.com/content/pdf/10.1007/BF01407877.pdf (visited on 05/06/2020).

[Inc13] Docker Inc. *Docker*. 2013. URL: https://www.docker.com/ (visited on 05/10/2020).

[JG85] David S. Johnson and Michael R. Garey. “A 71/60 Theorem for Bin Packing*”. In: *JURNAL OF COMPLEXITY 1, 65-106(1985)* (1985). URL: https://reader.elsevier.com/reader/sd/pii/0885064X85900226?token=17134E773251338C6B021ACB1EA0AFC493E43BF3A7 (visited on 02/26/2020).

[Ker19a] Michael Kerrisk, ed. *Linux man pages online. PTHREADS(7)*. Oct. 10, 2019. URL: http://man7.org/linux/man-pages/man7/pthreads.7.html (visited on 05/02/2020).

[Ker19b] Michael Kerrisk, ed. *Linux man pages online. MMAP(2)*. Oct. 10, 2019. URL: http://man7.org/linux/man-pages/man2/mmap.2.html (visited on 02/25/2020).

[Ker19c] Michael Kerrisk, ed. *Linux man pages online. MPROTECT()*. Oct. 10, 2019. URL: http://man7.org/linux/man-pages/man2/mprotect.2.html (visited on 05/11/2020).

[Ker19d] Michael Kerrisk, ed. *Linux man pages online. GETPAGESIZE(2)*. Oct. 10, 2019. URL: http://man7.org/linux/man-pages/man2/getpagesize.2.html (visited on 05/14/2020).

[Knu07] Donald Ervin Knut. *THE ART OF COMPUTER PROGRAMMING. volume 1/Fundamental Algorithms*. 3 EDITION. 1997-07. ISBN: 0-201-89683-4. URL: http://broiler.astrometry.net/~kilian/The_Art_of_Computer_Programming%20-%20Vol%201.pdf (visited on 02/26/2020).

[Kor02] Richard E. Korf. “A New Algorithm for Optimal Bin Packing”. In: *AAAI-02 Proceedings* (2002). URL: https://www.aaai.org/Papers/AAAI/2002/AAAI02-110.pdf (visited on 02/27/2020).

[Lab07] Lawrence Livermore National Laboratory, ed. *SCR*. 2007. URL: https://github.com/LLNL/scr (visited on 05/10/2020).

[Li+15] Guanpeng Li, Karthik Pattabiraman, Chen-Yong Cher, and Pradip Bose. “Experience report: An application-specific checkpointing technique for minimizing checkpoint corruption”. In: *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)* (2015). URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7381808&casa_token=BrJ5Vb4meY4AAAAA:k2ciXIuYH3e7nkBvm4Fqg5aRDWOeTbKTLwgZzQ
tag=1 (visited on 05/11/2020).

[MS02] John M. Mellor-Crummey and Michael L. Schott. “Algorithms for scalable Syncronization on Shared-Memory Multiprocessors”. In: *ACM Transactions on Computer Systems, vol 9, No 1* (1991-02). URL: https://dl.acm.org/doi/abs/10.1145/103727.103729 (visited on 05/05/2020).

[NM92] Robert H. B. Netzer and Barton P. Miller. “What Are Race Conditions? Some Issues and Formalization”. In: *ACM Letters on Programming Languages and SystemsMarch 1992* (1992). URL: https://dl.acm.org/doi/abs/10.1145/130616.130623 (visited on 05/01/2020).

[Pla+95] James S. Plank, Micha Beck, Gerry Kingsley, and Kai Li, eds. *Libckpt: Transparent Checkpointing under Unix*. 1995. URL: http://web.eecs.utk.edu/~jplank/plank/papers/USENIX-95W.pdf (visited on 05/10/2020).

[PLP97] James S. Plank, Kai Li, and Michael Puening. “Diskless Checkpointing”. In: *CORRECT ME* (Dec. 12, 1997). URL: http://web.eecs.utk.edu/~jplank/plank/papers/CS-97-380.pdf (visited on 05/09/2020).

[Sch05] Martin Schulz. “A case for two-level distributed recovery schemes”. In: *ACM SIGMETRICS Performance Evaluation Review* (1995-05). URL: https://dl.acm.org/doi/pdf/10.1145/223586.223596 (visited on 05/07/2020).

[Sch11] Martin Schulz. "Checkpointing". In: *Encyclopedia of Paralelle Computing, 2011 edition* (2011). URL: https://link.springer.com/content/pdf/10.1007%2F978-0-387-09766-4_62.pdf (visited on 05/07/2020).

[Sha+15] Faisal Shahzad, Markus Wittmann, Thomas Zeiser, and Gerhard Wellein. "Asynchronous Checkpointing by Dedicated Checkpoint Threads". In: *Recent Advances in the Message Passing Interface. EuroMPI 2012* (2015). URL: https://link.springer.com/content/pdf/10.1007%2F978-3-642-33518-1.pdf (visited on 05/12/2020).

[Sha+17] Faisal Shahzad, Jonas Thies, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein, eds. *CRAFT: A library for easier application-level Checkpoint/Restart andAutomatic Fault Tolerance.* 2017. URL: http://web.eecs.utk.edu/~jplank/plank/papers/USENIX-95W.pdf (visited on 05/10/2020).

[SS03] Luís Moura Silva and João Gabriel Silva. "System-Level versus User-Defined Checkpointing". In: *Proceedings of the IEEE Symposium on Reliable Distributed Systems* (2000-03). URL: https://www.researchgate.net/publication/2644469_System-Level_versus_User-Defined_Checkpointing (visited on 05/15/2020).

[Tak+17] Hiroyuki Takizawa, Muhammad Alfian Amrizal, Kazuhiko Komatsu, and Ryusuke Egawa. "An Application-Level Incremental Checkpointing Mechanism with Automatic Parameter Tuning". In: *2017 Fifth International Symposium on Computing and Networking (CANDAR)* (Nov. 19, 2017). URL: https://www.researchgate.net/publication/324787662_An_Application-Level_Incremental_Checkpointing_Mechanism_with_Automatic_Parameter_Tuning/link/5b98f15d92851c4ba813cbc4/download (visited on 05/15/2020).

[Val90] Leslie G Valiant. "A Bridging Model for paralelle Computation". In: *Communications of the ACM, August 1990* (Aug. 1990). URL: https://dl.acm.org/doi/10.1145/79173.79181 (visited on 04/02/2020).

[ZN01] Hua Zhong and Jason Nieh, eds. *CRAK: Linux Checkpoint/Restart As a Kernel Module.* Jan. 11, 2001. URL: http://systems.cs.columbia.edu/files/wpid-cucs-014-01.pdf (visited on 05/10/2020).

Appendices

Appendix A

Library documentation

```

    Documentation for the C++ header-only library CheckpointLib.
    An application-level checkpointing library for Linux-based systems.

    CheckpointLib consists of the files: checkpoint.hpp, fileManager.hpp,
    first_fit.hpp and memManager.hpp and expects all files to be in the same
    directory.

To use CheckpointLib, include the line:
  #include "checkpoint.hpp"
in the appliaction. While all public members and functions is declared in this doc,
checkpointlib is only designed to be used via the classes defined in checkpoint.hpp

To compile a application using CheckpointLib using g++,
if the application in question is "main.cpp" type the following:
g++ -std=c++11 main.cpp -lpthread



***fileManager.hpp:
Designed to manage the two checkpoint files and largely implemented
using linux system calls.

->FileManager(std::string init_fileNameA, std::string init_fileNameB)
Description:
Initiates an instance of the class "FileManager". It's objective
is to manage the interactions with the file system.
Arguments:
#init_fileNameA: either path and name of a file. If the file does not exist,
a file will be created here later
#init_fileNameB: either path and name of a file. If the file does not exist,
a file will be created here later. Should not be identical to init_fileNameA.
Returns:
  An instance of FileManager.


->int initFileManager(std::size_t size)
Description:
creates or expands the filepaths provided to FileManager.
Adjust or creates a counter in each file to trach which one
```

```
was written to the last time. This call will never shrink a file.
Arguments:
  #size: size (in bytes) that the files should be at least.
Returns:
   zero on sucess, else -1.

->int expand_file(std::size_t new_size)
Description:
Expands the filepaths provided to FielManager. This call will
never shrink a file.
Arguments:
  #size: size (in bytes) that the files should be at least.
Returns:
   zero on sucess, else -1.

->int load_checkpoint(void* buff, std::size_t size)
Description:
Reads the content of the file that has been written to most
recently into memory.
Arguments:
  #size: size (in bytes) to read into buffer.
  #buff: buffer in memory of at least #size bytes.
Returns:
   zero on sucess, else -1.


->int save_checkpoint_fd()
Description:
   Returns a file descriptor for the that of the two files that has the lowest
     counter.
Arguments:
Returns:
   a file descriptor, else -1.

->int save_checkpoint_finalize_save(int fd)
Description:
   Closes a filedescriptor and increments the counter of that file. Meant to close a
     file descriptor returned by save_checkpoint_fd().
Arguments:
  #fd: a file descriptor
Returns:
   Zero on sucess, else -1.


***first_fit.hpp:
Implements the first-fit memory management approach in a pre-allocated buffer of
     memory.

->FirstFit(void* buff_addr, std::size_t init_totalSize)
Description:
   Initiates the first-fit algorithm in the provided memory.
Arguments:
  #buff_addr: address pointing to the beginning of the allocated memory.
  #init_totalSize: the size (in bytes) starting at #buff_addr that should be
     administrated by FirstFit.
```

```
Returns:
  An instance of FirstFit.


->void* ff_allocate(std::size_t alloc_size)
Description:
  allocates a chunk of the buffer.
Arguments:
  #alloc_size: The size (in bytes) to allocate from the buffer.
Returns:
  A pointer on sucess, else a null-pointer.


->int ff_free(void* freeMe)
Description:
  frees a chunk allocated by ff_allocate(), makeing it free to be reallocated,
    possibly as smaller chunke or as part of a even bigger allocation.
Arguments:
  #freeMe: The address pointing to the start of the allocation to be freed..
Returns:
  Zero on sucess, else -1.

->int ff_adjust_allocation_size(std::size_t new_size)
Description:
  Adjust the FirstFit instance's perseption of the buffer size. This call is
    intended to make a runtime expansion of a existing memory mapping possible, but
    is in it's current state not very safe to call.
Arguments:
  #new_size: the new size (in bytes) that the instance of FirstFit will percieve the
    buffer to have.
Returns:
  This function will always return Zero.


***memManager.hpp:
Responsible for the allocation of memory along with the management of FileManager
    and FirstFit.

!!!Globally defined variables and functions start !!!
->void* _protected_mem_startAddr
Descritption:
  Globally defined pointer defining the first address to be included in the segfault
    handler.

->void* _protected_mem_stopAddr
Descritption:
  Globally defined pointer defining the last address to be included in the segfault
    handler.

->void segfault_sigaction(int signal, siginfo_t *si, void *arg)
Description:
  The function responsible for differentiating between segfaults in memory that
    CheckpointLib manages and any other segfault. If segfault happens in a address
    spanning between _protected_mem_startAddr and _protected_mem_stopAddr, the
    thread is put to sleep and afterwards returned to the line that caused the
    segfault. Otherwise the memory address that caused the segfault is printed and
    the entire process exits.
Arguments:
  #signal: standart argumment required by <signal.h>. Is ignored.
  #*si: contains the address of memory that triggered the segfault.
  #*arg: standart argumment required by <signal.h>. Is ignored.
Returns:
  Nothing.


->void _save_checkpoint_gradually(std::mutex* lockCheck, std::uint8_t* barrier_flag,
            void* memAddr, size_t memSize, FileManager* FM)
Description:
  This function writes a buffer of memory to a file dictated by FileManager. It
    imposes read protection on the buffer and lifts the restriction as the data is
    written(one system-defined page-size at a time).
Arguments:
  #*lockCheck: A lock that is held to the end of execution, preventing threads from
    starting a new checkpoint as long as the old one is being written.
  #*barrier_flag: Designed for the other threads to spin on this flag while the
    function sets up the initial read protection on the memory. The flag is set to
    zero when the buffer is protected.
  #*memAddr: Start address of the buffer to be written to file.
  #memSize: size (in bytes) of the buffer at #memAddr
  #*FM: pointer to a instance of FileManager.
Returns:
  Nothing.


void _load_checkpoint(void* memAddr, size_t memSize, FileManager* FM)
Description:
  This function is a wrapper for the FileManager function load_checkpoint().
Arguments:
  #memAddr: start of buffer, passed to load_checkpoint().
  #memSize: size of buffer(in bytes), passed to load_checkpoint()
  #*FM: pointer to the FileManager that facilitates the read.
Returns:
  Nothing.


!!!Globally defined variables and functions stop !!!

->MemManager(std::string init_fileNameA, std::string init_fileNameB)
Description:
  Creates the MemManager instance. MemManager acts as a wrapper for a instance of
    FileManager and FirstFit as well.
Arguments:
  #init_fileNameA: argument passed to FileManager.
  #init_fileNameB: argument passed to FileManager.
Returns:
  Instance of MemManager.


->int initMemManager(std::size_t expected_size)
Description:
  Asserts and allocates memory, initiates FirstFit and acts as Wrapper for initFileManager().
Arguments:
  #expected_size: The size of memory (in bytes) to allocate.
Returns:
  zero on sucess, -1 on failure.


->void* allocate_memory(size_t requested_size)
Description:
  See ff_allocate()


->int free_memory(void* addr)
Description:
  See ff_free()


->std::thread start_save_checkpoint(std::mutex* lockCheck, std::uint8_t* barrier_flag)
Description:
  Starts a new thread, running the function _save_checkpoint_gradually(). passes the entire memory buffer MemManager manages along to the function as well as its instance of FileManager.
Arguments:
  #*lockCheck: passed as argument to _save_checkpoint_gradually().
  #barrier_flag: passed as argument to _save_checkpoint_gradually().
Returns:
  Returns the thread control object.

->void start_load_checkpoint()
Description:
    see _load_checkpoint() (is called with arguments from the calling MemManager);


->void* ret_mm()
Description:
  Returns pointer to beginning of allocated memory buffer. Included for unit testing .
Returns:
  A pointer.

->size_t ret_memSize()
Description:
  Returns size (in bytes) of allocated memory buffer. Included for unit testing.
Returns:
  size of buffer.


***checkpoint.hpp:
The overall wrapper for the CheckpointLib.

->CheckpointOrganizer(std::string init_fileNameA, std::string init_fileNameB)
Description:
  Initiates an instance of the class "CheckpointOrganizer".
```

```
Arguments:
  #init_fileNameA: passed as argument to memmManager().
  #init_fileNameB: passed as argument to memmManager().
Returns:
  An instance of CheckpointOrganizer.


->int initCheckpointOrganizer(std::uint8_t init_threadTotal, bool init_load_first,
    std::size_t expected_size)
Description:
  Primarily acts as a wrapper for initMemManager().
Arguments:
  #init_threadTotal: The number of threads this instance of CheckpointOrganizer
    expects to be initiated.
  #init_load_first: If true, the application will perform a restart at first
    checkpoint.
  #expected_size: passed as argument  to initMemManager()
Returns:
  See initMemManager().


->std::thread* startThread(void (*f)(void *, Checkpoint*), void* init_data)
Description:
  Starts a thread that is able to be checkpointed by this instance of
    CheckpointOrganizer.
Arguments:
  #f: The function that is to run in the thread, note it must accept a instance of
    Checkpoint.
  #*init_data: argument passed to #f.
Returns:
  Pointer to the control object of the thread.


->int joinAll()
Description:
  Shortcut for joining all threads currently running that was initiated by
    startThread().
Returns:
  will always return 0.


->void* check_alloc(size_t requested_size)
Description:
    See allocate_memory().


->int check_free(void* addr)
Description:
    See free_memory().


->Checkpoint(std::uint8_t init_threadTotal, std::uint8_t init_myId, std::mutex*
    init_lockCheck_in, std::mutex* init_lockCheck_out, std::uint8_t*
    init_barrier_flag_in, std::uint8_t* init_barrier_flag_out, bool* init_load_first
    , MemManager* init_mem)
```

```
Description:
   A new instance of Checkpoint is automatically initialized and passe along to threads started by startThread().
Arguments:
   #init_threadTotal: the amount of threads to wait for when barrier synchronizing.
   #init_myId: A id for the thread.
   #*init_lockCheck_in: part of the barrier synchronization.
   #*init_lockCheck_out: part of the barrier synchronization.
   #*init_barrier_flag_in: part of the barrier synchronization.
   #*init_barrier_flag_out: part of the barrier synchronization.
   #*init_load_first: if true first checkpoint will act as restart.
   #*init_mem: pointer to the shared memManager
Returns:
   Instance of the Checkpoint class.


->void check()
Description:
   Will synchronize all threads started by startThread(). Depending on the parameter "init_load_first" from Checkpoint(), this call either restarts the memory from a checkpoint file or makes a new write. Note that if "init_load_first" is set to true, after the first encounter with this call, it is set to false.
Returns:
   Nothing.
```

Appendix B

Benchmark functions

Here is the functions used for benchmark'ing the overhead of the library. One is a application without checkpointing and one is the same application using **CheckpointLib**. In addition two more applicatoins is included: one is the application using only the barrier implementation that is used in **CheckpointLib** and the one application which starts a thread on each run. The last two will hopefully give an idea of where the overhead in the **CheckpointLib** is.
For the full setup, Look in the **CheckpointLib** repository under benchmarks/random_array

```

struct test_thread_struct{
  arrTestDat _arrTest;
  bool initiate_thread = false;
};

void almost_empty_function(int iterations){
  for(int i=0; i<iterations; i++){
    // do nothing...
  }
}

// only one of the threads has tmp_struct.initiate_thread==true, simulating checkpointlib
// where only one thread would start a checkpoint thread
void array_test_start_thread(void* data){
  srand(time(NULL));
  std::thread tmp_thread;
  test_thread_struct* tmp_struct = (test_thread_struct*) data;
  arrTestDat my_dat = tmp_struct->_arrTest;
  add_multiples_to_all(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
  refresh_prime(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);

  if(tmp_struct->initiate_thread) tmp_thread = std::thread(almost_empty_function, 5);



  add_multiples_to_all(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
  refresh_prime(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);

  if(tmp_struct->initiate_thread) tmp_thread.join();
}

std::chrono::milliseconds global_PauseThread(100);
std::uint8_t global_total_threads = 0;
std::uint8_t global_flag_in = 0;
std::uint8_t global_flag_out = 0;
std::mutex global_lock_in;
std::mutex global_lock_out;

void mock_barrier(){
  while(global_flag_out){
    std::this_thread::sleep_for(global_PauseThread);
  }
  global_lock_in.lock();
  global_flag_in += 1;
  if(global_flag_in >= global_total_threads){ // start of critical section
    global_flag_out = global_total_threads;
    global_flag_in = 0;
  }
  global_lock_in.unlock();

  while(global_flag_in){
    std::this_thread::sleep_for(global_PauseThread);
  }
  global_lock_out.lock();
  global_flag_out -= 1;
  global_lock_out.unlock();
}



void array_test_lock(void* data){
  srand(time(NULL));
  arrTestDat my_dat = *((arrTestDat*) data);
  add_multiples_to_all(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
  refresh_prime(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
  mock_barrier();
  add_multiples_to_all(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
  refresh_prime(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
}


void array_test_checkpoint(void* data, Checkpoint* check){
  srand(time(NULL));
  arrTestDat my_dat = *((arrTestDat*) data);
  add_multiples_to_all(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
  refresh_prime(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
  check->check();
  add_multiples_to_all(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
  refresh_prime(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
}
```

```

void array_test_control(void* data){
  srand(time(NULL));
  arrTestDat my_dat = *((arrTestDat*) data);
  add_multiples_to_all(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments
    );
  refresh_prime(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);

  add_multiples_to_all(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments
    );
  refresh_prime(my_dat.prime, my_dat.mem, my_dat.segment, my_dat.num_segments);
}

```