BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-96-M3

"Checkpointing and Migration for Quahog"

by

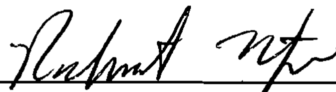
Alnoor Mamdani

Checkpointing and Migration
for Quahog

Alnoor Mamdani
Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for
the degree of Master of Science in the Department of
Computer Science at Brown University

September 1995



Robert H. B. Netzer
Advisor

Contents

1 Introduction **3**
1.1 Project Goals 4

2 Quahog **4**
2.1 Quahog Components 4
2.1.1 Quahog Daemons 4
2.1.2 Quahog Clients 5
2.1.3 Quahog State Server 6
2.1.4 Quahog Commands 6
2.2 Extending Quahog 7

3 Checkpoints **7**
3.1 Process Memory Layout 8
3.2 Signals 9
3.3 `setjmp()`, and `longjmp()` 9
3.4 Saving Checkpointing Information 9
3.4.1 Saving the Program Text 9
3.4.2 Saving the Data Segment 9
3.4.3 Saving the Stack 10
3.4.4 Saving the Process State 10
3.4.5 Saving Open File Information 10
3.5 Checkpointing and Recovery Mechanics 11
3.5.1 Setup and Initialization 11
3.5.2 Creating the Checkpoint 11
3.5.3 Recovery using the Checkpoint 14
3.5.4 Well Behaved Programs & Limitations 14

4 Improving Checkpointing Performance **15**
4.1 Incremental Checkpoints 15
4.2 Forked Checkpoints 16
4.3 Memory Exclusion 16
4.4 Synchronous Checkpoints 17

5 Project Summary **17**
5.1 Investigation Phase 17
5.2 Implementation Phase 17

5.2.1 Other Porting Issues 18
5.3 Project Status 19
5.4 Bugs 19

6 User's Guide **20**
6.1 Changes to User Sources 20
6.2 `.ckptrc` File 20
6.3 Quahog Requirements File 21
6.4 Location of Project Files 21

7 Further Work **21**

Master's Project: Checkpointing and Migration for Quahog

Alnoor Mamdani

September 8, 1995

1 Introduction

Networks of workstations offer computing resources that are often unutilized during some parts of the day, or underutilized depending on the type of user. Generally, in such a computing environment, a workstation is dedicated to a single user who may or may not be able to fully utilize all the power of the workstation.

Workstation users have been categorized in to three types [3]. *Type 1* users mostly use their workstation for word processing, sending and receiving mail. Type 1 user's machines are almost always underutilized. Software developers can be identified as *Type 2* users. Their machines are underutilized during code editing phase of their work, but more resources are required during the compilation, running, and debugging phases of their work. *Type 3* users need all the computing power of their workstations and usually more. Type 3 users who run large simulations or searches requiring more computing power than they have are always clamoring for more computing power. Furthermore, Type 3 users can keep their machines busy 24 hours a day, but Type 1 and Type 2 user's machines are completely unutilized when they go home. Quahog is a distributed process manager developed here at Brown that allows Type 3 users to take advantage of idle CPU cycles on workstations owned by Type 1 and Type 2 users.

1.1 Project Goals

The initial goal of this project was to make the Quahog system more *usable* by adding checkpointing and migration. We felt users would appreciate being able to submit long running compute intensive jobs to Quahog without the worry and irritation of their partially complete job being killed or suspended. This Master's project investigates checkpointing and migration of single threaded UNIX processes. We add limited migration functionality to the Quahog system, although much work still needs to be done. This document should serve a good starting point for another student who wishes to take this project to completion. The Quahog system, checkpointing, and the checkpointing library **libckpt** are described. Finally the current status of the project is described.

2 Quahog

Quahog is a distributed process manager running on 160 SPARC machines within the Computer Science Department at Brown University. Users can submit commands or batch files that will be run on idle machines.

Quahog has been developed as a *polite* system. That is, it minimally impacts the owner of workstations that will run Quahog jobs. When Quahog detects that the workstation is being used by its owner, the job is guaranteed to be killed within a fixed amount of time. The delay the workstation owner suffers in order to regain the resources of his machine is bounded.

2.1 Quahog Components

The Quahog system consists of three parts: daemons, clients, and a state server. See Figure 1.

2.1.1 Quahog Daemons

A daemon runs on every Quahog machine that accepts or submits tasks for execution. Quahog daemons service local clients who wish to submit

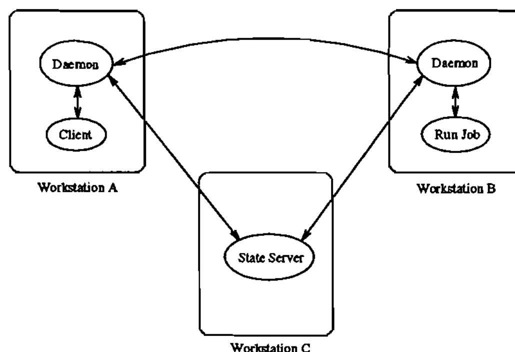


Figure 1: Quahog Components

remote jobs. They accept remote requests from other remote daemons to run a job locally. Finally, they keep track of local keyboard activity and load information. Quahog clients connect to the local Quahog daemon to submit a task for execution. The local daemon determines which remote machine to submit the task to by consulting a local database. This database contains information about every remote machine in the Quahog system, and is updated periodically from a the global state server. Once the local daemon decides the remote machine, it connects to the remote daemon to submit the job. The job can be rejected if state conditions have changed since the submitting machine's updating of its local information from the global state server. The daemon also determines whether a Quahog job can be run locally. The daemon continuously monitors system activity including the load, CPU usage, memory usage, and interactive users by checking keyboard activity. This information is sent by the daemon to the state server. The local Quahog daemon will reject a remote job if there are interactive users using the system. The other information is used by the remote machine to decide where the job should be run. See Figure 2.

2.1.2 Quahog Clients

A Quahog client is a program that submits jobs for remote execution. If the client is a program, then it communicates with Quahog through the Quahog API. Alternatively, jobs can be submitted to the Quahog system through the

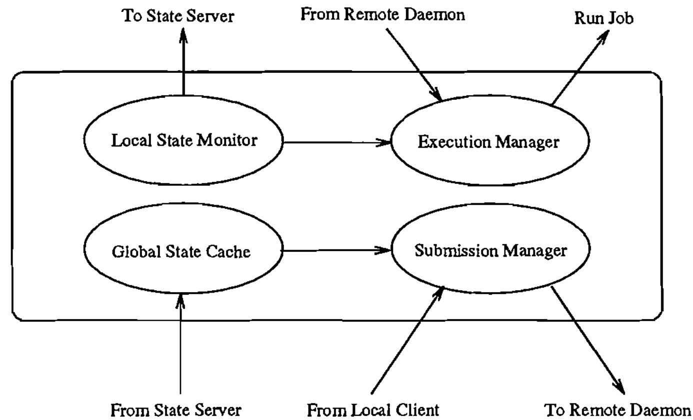


Figure 2: The Quahog Daemon

utility qrun.

2.1.3 Quahog State Server

The state server is a network-wide database containing the local state information of each workstation. The daemons on each machine periodically update the information in this database, and periodically update their local copies of this database. Workstation information such as the CPU load, the amount of memory, idle time, number of processors, and number of Quahog jobs running are used to determine which remote machine to submit the the job on.

2.1.4 Quahog Commands

- `qrun` allows the user to run a command or batch on a remote machine.
- `qstat` gives statistics on all machines running the Quahog daemon.
- `qjobs` lists all active Quahog jobs running on a system.
- `qkill` kills a Quahog job.

- The Quahog API [6] allows programs written in C to interact with the Quahog daemon.

2.2 Extending Quahog

The Quahog system does not guarantee that a job submitted will execute until completion. When Quahog determines that a Quahog user is interfering with the workstation owner, in the interest of politeness, Quahog must stop or kill the Quahog user's job. The Quahog user specifies how Quahog should handle user jobs once the system becomes unavailable due to interactive users. By default the job will be killed if the node is reclaimed by its owner, however the user can ask Quahog to attempt to suspend the job if it does not consume excessive resources. Killing the process is reasonable for short-lived processes. When Quahog stops a process, although no CPU resources are consumed, the job may still interfere with the workstation owner if excessive swap space is being consumed, and in those cases the process is also killed. This is not reasonable for long-lived processes. The unfortunate result of killing the process is that all work accomplished thus far is lost, and compute resources expended on that work were wasted. By adding checkpointing and migration to the Quahog system, the Quahog user's job can be migrated to other idle machines when the workstation owner needs his workstation back.

3 Checkpoints

By creating a checkpoint of the process before killing it, we save the necessary process information required to restart the process on another machine. In order to establish a *point of recovery*, we save the contents of memory, processor registers, and the status of open files so that the process can continue from the point of recovery.

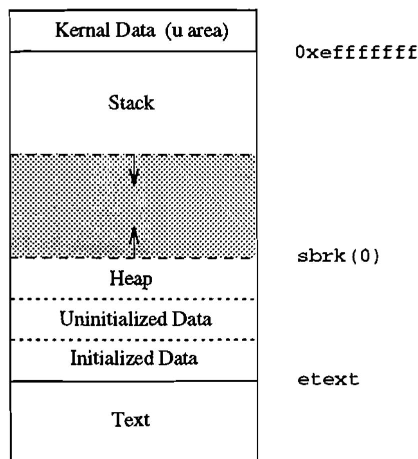


Figure 3: Solaris 5.4 Process Memory Layout

3.1 Process Memory Layout

The memory image of a UNIX process contains three major segments. See Figure 3.

Stack This is where the activation record for each function called is kept and automatic variables are stored. The size of the stack varies at run time depending on the number of functions currently called.

Text Program code executed by the CPU is kept in the text segment. The text segment contains routines written by the user as well as the routines that are part of the checkpointing library. The contents of the text segment do not change through the life of the process.

Data The data segment is divided into three parts. The initialized data segment contains static variables that have been assigned values by the programmer at compile time. The uninitialized data segment is the area where static variables that were not assigned values at compile time live. Variables in this area are initialized to 0 by the kernel when the program is loaded. Finally the heap is where dynamic memory allocation takes place. The heap grows as calls are made to the UNIX system calls `brk()` and `sbrk()` through the C function `malloc()`.

3.2 Signals

Signals are software interrupts that provide a way of handling asynchronous events. When a process receives a signal, the current state of the process is saved and control is transferred to a user defined signal handler function. When the signal handler returns, the process resumes execution at the point it was interrupted. The UNIX kernel keeps track of the information needed to restore the process. When creating a checkpoint file, we use the signal handling mechanism to help save the state of a process.

3.3 `setjmp()`, and `longjmp()`

`setjmp()` and `longjmp()` are C-library routines used for non-local branching. They provide a way to `goto` a label in a different function. We call `setjmp()` from the point in our code that we want a corresponding `longjmp()` to return to in later execution. `setjmp()` saves the stack environment, registers, and signal mask in a `jmp_buf` struct that is later used by `longjmp()`. During checkpointing, we call `setjmp()` to save our call environment; when recovering, we call `longjmp()` to restore the saved call environment.

3.4 Saving Checkpointing Information

When creating the checkpoint we must save the contents of the data and stack segments, process state, and open file information in order to establish a point of recovery.

3.4.1 Saving the Program Text

Since information in the text segment does not change, we get this information from the executable. Nothing needs to be saved.

3.4.2 Saving the Data Segment

In order to save the contents of the data segment we need its beginning and ending addresses. The beginning of the data segment is given by the system

variable `etext` which is the first address above the program text. We get the end address of the data segment by calling `sbrk(0)`. `sbrk(0)` returns the first address past the top of the heap. The memory contents between these two address are written to a disk file.

3.4.3 Saving the Stack

We save stack information in the following way. First we save the stack context by using `setjmp()`. The stack environment, including the stack pointer and the processor state, is saved in a `jmp_buf` struct in the data segment. However, a limitation of `setjmp()` is that it does not save the actual contents of the stack. Thus, we must also save the area where the activation records live. Again we need the beginning and ending addresses to write the contents to disk. The bottom of stack for Solaris 5.4 running on SPARC machines is `0xefffffff`, and we determine the top of stack programatically since it varies at runtime. The stack grows toward lower addresses.

3.4.4 Saving the Process State

The process state including processor registers is saved using `setjmp()` and stored in a `jmp_buf` struct in the data segment.

3.4.5 Saving Open File Information

Part of the system state that we save includes the list of open file descriptors and file positions. Each time the user opens a file, we record file information such as the file path, file attributes, and the file descriptor in a table. When creating the checkpoint, for each open file we record the position of the file pointer. During recovery we used the stored information to reopen each file and reposition the file pointer using `lseek()`.

3.5 Checkpointing and Recovery Mechanics

3.5.1 Setup and Initialization

The Quahog user must link their program with the checkpointing library (`libckpt.a`) for Quahog to restart the process on a new machine. To link with **libckpt**, the user must rename their `main()` to `ckpt_target()`. **libckpt** defines a `main()` function that will do some required setup and initialization before calling the user's `ckpt_target()` function.

Part of the initialization is to set up a timer and signal handler for `SIGALRM`. Each time the process receives a `SIGALRM`, it checkpoints itself. When a checkpoint signal is delivered, the signal handler will save the data segment, the stack contents, and the stack context by calling `setjmp()`.

Another part of the initialization for **libckpt** is to read a configuration file `.ckptrc` that lets the user change default checkpointing options. For example, the user can change how often they wishe their process to be checkpointed.

3.5.2 Creating the Checkpoint

When the user process receives a `SIGALRM`, the signal handler will do the following:

1. Save process state information in the `jmp_buf` struct by calling `setjmp()`.
2. Record positions of open files.
3. Save the contents of the data segment (including the `jmp_buf` struct) to a disk file.
4. Save the contents of the stack to a disk file.

In Figure 4a we see the memory contents of a typical process while it is executing function `foo()`. Figure 4b shows the contents of memory after the process has received the `SIGALRM` signal and called `setjmp()`.

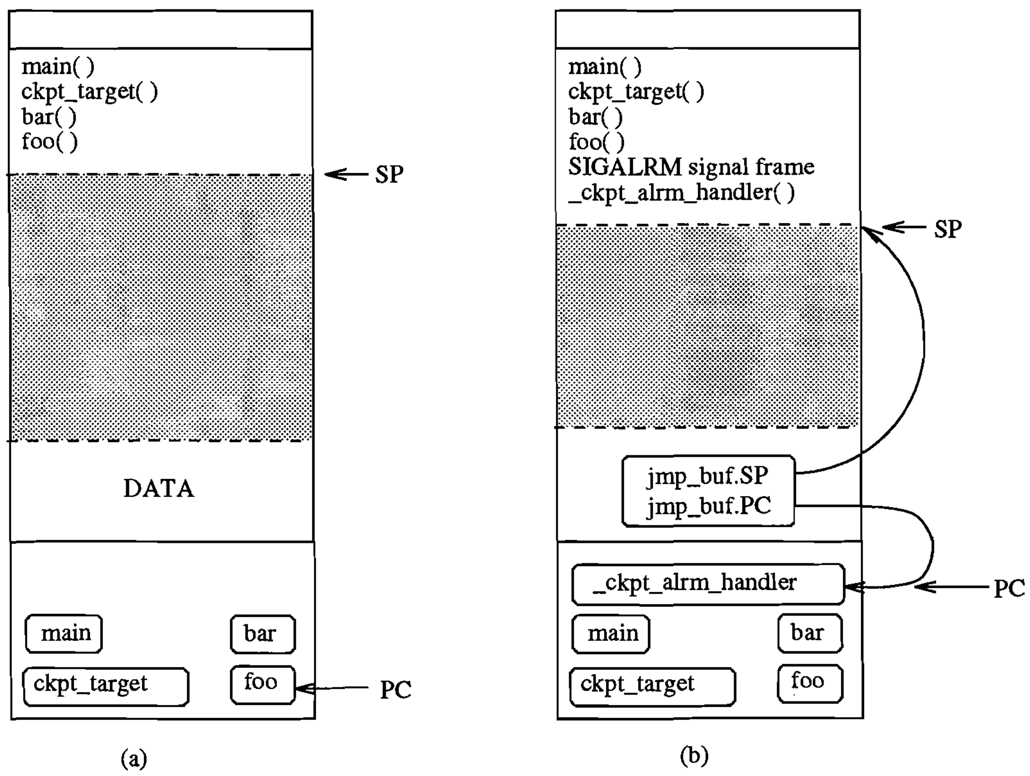


Figure 4: Creating the Checkpoint

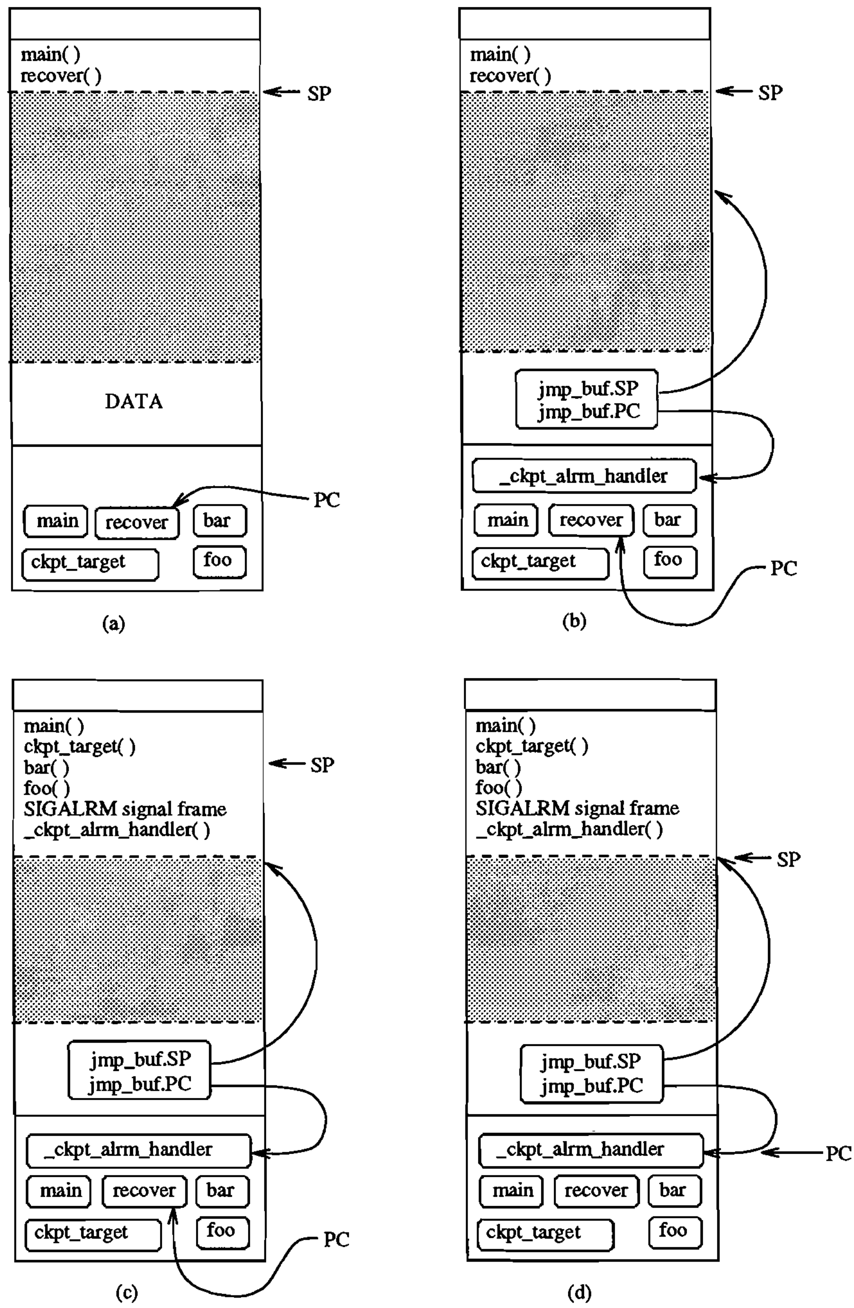


Figure 5: Recovery from Checkpoint

3.5.3 Recovery using the Checkpoint

Recovery proceeds in the following way:

1. Quahog will restart the command with a command line argument `=recover`. The text segment is automatically restored when the process is started, but stack and data segment contents need to be restored to their values at the time of the checkpoint.

2. **Libckpt** detects the `=recover` flag and restores the contents of the data and stack segments by reading the disk files containing their contents at the time of the last checkpoint.

3. The processor state is restored by calling `longjmp()` using the saved `jmp_buf` struct. The processor registers including the stack pointer and the program counter are restored. At this point, execution resumes in **libckpt** code where `setjmp()` was called.

4. Finally, files that were open at checkpoint time are now opened and their file pointers repositioned.

In Figure 5a we see the memory contents of the restarted process executing the `recover()` function. At this point only the text segment has been restored. When **libckpt** detects the `=recover` flag it calls the `recover()` function. Figure 5b shows the memory contents after the data segment has been restored, and Figure 5c shows the contents after the stack has been restored. Finally, Figure 5d shows the contents after the call to `longjmp()` and the process state restored. At this point the process is ready to resume execution.

3.5.4 Well Behaved Programs & Limitations

Not all types of programs may be checkpointed. The following are limitations of jobs that can be checkpointed:

1. Currently **libckpt** can only support a single process job. Programs that make calls to `fork()` and `exec()` cannot be checkpointed.

2. Programs that use signals and signal handlers cannot be checkpointed. A signal could be delivered during the generation of the checkpoint and

corrupt the contents of the checkpoint files if the system state has been changed.

3. **Libckpt** cannot restore the system state of processes that communicate with other processes through the use of sockets or pipes since it cannot ensure that peer processes are extant at the time of recovery.

4. File operations must be idempotent. Read-only and write-only file accesses work correctly, but programs that read-write the same file may not.

5. It is assumed that the machine that the process is restarted on has access to the same file system.

4 Improving Checkpointing Performance

The most straightforward way to checkpoint a process is to write the entire contents of the data and stack segments to disk files. These types of checkpoints are called *sequential* because execution of the process being checkpointed is suspended while the checkpoint files are written to disk. Disk transfers are not interleaved with program execution. Four optimization techniques have been implemented in **libckpt** that can offer improved performance over vanilla sequential checkpointing. These optimizations have been implemented by the authors of **libckpt** [5].

4.1 Incremental Checkpoints

Incremental checkpoints save only the information that has changed since the last checkpoint. Since the amount of information in the checkpoint file is reduced, the size of the file and checkpointing overhead are also reduced.

Page protection is used to implement incremental checkpoints. During the initialization phase, the `mprotect()` call is used to mark each page in the data segment as read-only. In addition, signal handlers for `SIGSEGV` and `SIGBUS` are installed. When the process tries to change a data item, a `SEGV` signal is generated, and the signal handler is invoked. The signal

handler marks the page dirty, and changes the protection for that page to read-write. When the process receives a checkpoint signal, only the dirty pages are written to the checkpoint file.

Non-incremental checkpoint files can overwrite previous versions since all information required for recovery are in the new checkpoint files. Incremental checkpoint files, however, cannot be deleted since they contain necessary information required to restore the process. The system state is restored by using the first checkpoint, and then adding the information in each subsequent checkpoint file incrementally. To keep a bound on the number of checkpoint files, **libckpt** will coalesce incremental checkpoint files into a single file when a user defined threshold is reached.

4.2 Forked Checkpoints

Forked checkpoints improve performance by interleaving application program execution with the creation of the checkpoint file. When forked checkpointing is specified, **libckpt** creates an asynchronously executing thread of control to write the checkpoint file using the `fork()` primitive. The `fork()` system call creates a new child process with a copy of the parents data space to write the checkpoint file, and lets the parent resume. Writing of the checkpoint file by the child is interleaved with the parent application program execution.

4.3 Memory Exclusion

Memory exclusion allows the user to explicitly mark areas of memory that do not need to be saved to the checkpoint file. These areas can be categorized as either *clean* or *dead*. Dead locations are areas of memory that will never be read nor written and thus do not need to be saved. Clean locations are areas of memory that exist in a previous checkpoint and have not changed. These locations do not need to be written to the checkpoint file.

Memory exclusion can be used in the following way to improve performance. Suppose the user allocates a large temporary array to make a computation. Each time the user makes the calculation the old values in the array are written over. Thus, the values in the array are dead between calculations and can be safely excluded from a checkpoint.

4.4 Synchronous Checkpoints

Synchronous checkpointing allows the user the specify the points in the code where a checkpoint can be taken. These checkpoints are called synchronous because they are not initiated by asynchronous timer interrupts. The overhead of taking a checkpoint depends on the amount of data that must be written to disk. If the user can force a checkpoint when the size of the stack is small, and when the amount of excluded memory is high, then performance will improve.

5 Project Summary

This section describes some of the approaches used to investigate the problem, the porting issues, and the current status of the project.

5.1 Investigation Phase

Initially I started with a list of projects that are implementing recoverable processes for PVM. I started poking around the web sites on the list to find out about checkpointing in general. One of these sites had a link to the Condor Web Page at the University of Wisconsin-Madison. Initially, this was probably the most helpful resource I came across since they had a few papers that describe the basics of checkpointing a UNIX process [3]. In a paper at another web site, I also found a reference to the Litzkow [4] paper that explained the fundamental concepts of checkpointing I needed to know. Finally, Juan Leon at CMU sent me a pointer to a checkpointing web site `http://warp.dcs.st-and.ac.uk/warp/systems/checkpoint` where I found **libckpt**.

5.2 Implementation Phase

To understand the methods used to checkpoint UNIX processes, I had to first consult a few texts [1, 7] on the UNIX operating system. Before starting my graduate studies at Brown, all of my software experience was in the PC world

on single user systems such as MS Windows and MSDOS. UNIX processes, signals, and checkpointing were all new concepts for me. I also spent much time reviewing the Sun *Linker and Libraries* manual and the *Solaris 1.x to Solaris 2.x Transition Guide.*

Initially I started looking at the Condor checkpointing code for Solaris 1.x. I planned to extract the checkpointing code from Condor and then port it to Solaris 2.x. During this effort I found a generic checkpointing library **libckpt** at the University of Tennessee, Knoxville, and scraped the work I had done with the Condor code.

Initially, when porting **libckpt** I tried to use the compatibility libraries and compilers that allow developers to create executables for Solaris 2.x from code written for Solaris 1.x. After adjusting the makefiles and compiling, the executables produced strange results even before any checkpointing code was reached. Weird behavior like pointer alignment problems while reading the `.ckptrc` file convinced me to quit using the compatibility libraries and do a full port to Solaris 2.x.

5.2.1 Other Porting Issues

The major changes required to port **libckpt** to Solaris 2.x included the following:

1. Compiler. I changed the compiler from GNU C to the Sun C compiler in order to take advantage of all the developer tools for the Sun environment. In addition, I was more familiar with the Sun compiler. This required adding prototypes and changing some types since the Sun compiler is a little more picky.

2. Signals. In Solaris 2.x, in order to get extra information to the signal handler via the `siginfo_t` structure, the `sigaction()` call should be used instead of the `signal()` call.

3. File IO. The `syscall()` system call is not available in Solaris 2.x. **Libckpt** used `syscall()` to trap file system calls such as `open()` and `read()` so it could reopen and reposition files open at the time of a checkpoint. I renamed **libckpt** file system calls to begin with the prefix `libckpt_`. So `open()` becomes `libckpt_open()`, etc. Unfortunately, the user is forced to change his source to take advantage of this feature

of **libckpt**. However, he can easily include `#define`'s in his headers to rename `open()` in his source to `libckpt_open()`.

4. Incremental Checkpoints. There were some bugs in the function `protect_all()` that is called after an incremental checkpoint is written to disk. This function remarks every page in the data segment as read only. It was incorrectly computing the starting and stopping addresses for the data segment. This was a difficult bug to find since the error returned by `mprotect()` was not being checked and there was no indication of error.

5. Address Alignment. The start address data segment needed to be aligned with the beginning of the page. The system variable `etext` gives the first address past the text segment, but it is not page aligned.

5.3 Project Status

Currently we have basic checkpointing working. The user can restart his process with the `=recover` flag. Forked and incremental checkpoints are also working, but have not been rigorously tested.

When a job has been checkpointed, it cannot be restarted on just any machine in the department. The hardware, eg. SPARC10 or SPARC2, of the machines must be the same, and also the OS version must be identical, eg. Generic_101945-13. The number of processors for multiprocessor systems does not matter.

A flag `-m` has been added to the `qrun` Quahog utility that will migrate the process when Quahog kills the job. When this flag is specified and the job has been linked with the checkpointing library, Quahog will automatically restart the job on another idle machine.

5.4 Bugs

There are a number of known bugs:

1. Quahog requirements file. A Quahog user can specify a requirements file with the `qrun` utility. However, `qrun` currently ignores the field

Q_OSVERSION so the job may be run on a system with a different version and the restart will fail.

2. Incremental checkpoints. When coalescing incremental checkpoints on new machine, a segfault is generated. Incremental coalescing works however on the same machine.

3. File repositioning has not been tested.

6 User's Guide

6.1 Changes to User Sources

Libckpt users must include the file `checkpoint.h` and rename their `main()` to `ckpt_target()`. Optionally, they can append the prefix `libckpt_` to the following system calls if they wish to have open files repositioned on recovery: `open()`, `read()`, `write()`, `close()`, `dup()`, and `dup2()`.

`exclude_bytes()` and `include_bytes()` are used to exclude and include memory regions from the checkpoint file.

`checkpoint_here()` is used to generate user-directed checkpoints within the code.

6.2 .ckptrc File

A `.ckptrc` can be placed in either the users home directory or the directory containing the executable. It can contain the following entries:

`checkpointing <on|off>` turns checkpointing on or off.

`incremental <on|off>` turns incremental checkpoints on or off.

`fork <on|off>` turns forked checkpointing on or off.

`maxtime <seconds>` defines the interval between checkpoints.

`mintime <seconds>` defines the minimum time between checkpoints. This is useful when both user-directed and asynchronous checkpoints are used. If the amount of time specified has not elapsed, then a checkpoint will not be generated.

`maxfiles <n>` is the maximum number of incremental checkpoint files generated before they are coalesced.

`directory <dir>` specifies directory that checkpoint files will be found in.

`verbose <on|off>` turns debugging information on or off.

6.3 Quahog Requirements File

The Quahog requirements file specifies requirements for the machine the job will be run on. It is important to specify the hardware and OS version for the job to be restarted correctly.

```
Q_MACHINEHW=sun4m
Q_OSVERSION=Generic_101945-13
```

6.4 Location of Project Files

The sources for **libckpt** can be found `/pro/quahog/src/migration`. See the `README` file.

7 Further Work

As it stands currently, the system is still incomplete. There is work to finish in fixing known bugs, and better testing of the system. Performance testing and monitoring are areas that can also be explored.

- Bug Fixes and Testing. Not much work has been done in testing the system. File repositioning is an area that has not been tested at all.

- Checkpoint Naming Scheme. Currently the file names given to checkpoint disk files are static. Only one copy of a program can be concurrently checkpointed since multiple concurrently running copies of the same executable would overwrite each others checkpoint files. By creating checkpoint files with unique names, multiple copies of the same executable can be checkpointed and migrated.

- Performance testing. By putting hooks in the library we can measure the overhead due to checkpointing, and the amount of time it takes to migrate a process. These times will depend on the type of application being checkpointed, the settings in the `.ckptrc` file, network traffic, and the number of other Quahog jobs running.

- Monitoring. It would be nice to graphically view Quahog usage across the network. We could look at how much time a workstation is used by its owner, and how much time is used by Quahog. Also, we could track long running jobs, and follow how much time they spend at each node.

References

[1] Maurice Bach, *The Design of the UNIX Operating System*, Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[2] John Bazik, "Quahog: Polite Remote Processing", Computer Science Department, Brown University.

[3] Allan Bricker, Michael Litzkow, Miron Livny, "Condor Technical Summary", Computer Science Department, University of Wisconsin - Madison, 10/9/91.

[4] Micheal Litzkow and Marvin Solomon, "Supporting checkpointing and process migration outside the UNIX kernel", *Proceedings of the USENIX Winter Conference*, San Francisco, CA, January 1992, pp. 283-290.

[5] James Plank, Micah Beck, Gerry Kingsley, Kai Li, "Libckpt: Trasparent Checkpointing under UNIX", *Proceedings of the USENIX Winter Conference*, New Orleans, Louisiana, January 1995.

[6] Kevin Regan, "Quahog API", Computer Science Department, Brown University.

[7] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, Reading, Massachusetts, 1992.