The following paper was originally published in the
Proceedings of the 2nd USENIX Windows NT Symposium
Seattle, Washington, August 3–4, 1998

A Transparent Checkpoint Facility On NT

Johny Srouji, Paul Schuster, Maury Bach, and Yulik Kuzmin
Intel Corporation

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

A Transparent Checkpoint Facility On NT

Johny Srouji
Johny.Srouji@intel.com
Paul Schuster
Paul.Schuster@intel.com
Maury Bach
Maury.Bach@intel.com
Yulik Kuzmin
Yulik.Kuzmin@intel.com

Intel Corporation, Israel Design Center

Abstract

With the increased use of networks of NT workstations for long-running engineering applications, process checkpointing and process migration can avoid wasted computer cycles and improve system utilization. The problem we solve is how to capture and reconstruct process state transparently and efficiently without affecting the correctness of the application.

A checkpoint facility enables the intermediate state of a process to be saved to a file. Users can later resume execution of the process from the checkpoint file. This prevents the loss of data generated by long-running processes due to program or system failures, and it also facilitates debugging when the bug appears after the program has executed for a long time.

This paper describes the implementation of a checkpoint library that permits users to save temporary state of long-running multi-threaded programs on a Windows/NT system and to resume execution from the checkpointed state at a later time. Our Windows implementation is the first such implementations that we are aware of for this operating system. Our implementation is portable, maintains good performance, and is transparent.

The checkpoint facility is currently used in several major internal projects at Intel.

1. Introduction

This paper describes a checkpoint facility for long-running programs on Windows/NT. The checkpoint facility permits users to save the state of a running process at arbitrary points of execution and to resume execution from the saved state at a later time. This facility is important for long-running processes, so that users can capture intermediate results of processes that do not run to completion because of machine or application failure. If a long-running processes is interrupted before completion but after a checkpoint, the user can resume execution from an intermediate state instead of having to re-run the process from start. Checkpointing also gives developers great leverage when debugging long-running processes; they can debug from a point deep in the run or change input data after run time, rather than having to restart programs from the beginning. Finally, checkpointing is an important milestone in the development of a facility for process migration, whereby processes can be halted on one machine, moved to another, and continue execution transparently.

Our checkpoint facility is non-intrusive to user programs, in that the programmer need not change any code to save checkpoints during execution. Users can resume execution from the point of the checkpoint and receive the same results they would have received without checkpointing, subject to changes in the run-time environment that may have occurred before the program is resumed. Of course, processes can continue to execute after completing a checkpoint.

Process migration improves the overall utilization of resources to achieve high performance, and enhance fault tolerance by providing the capability to move work from a failing machine.

The checkpoint system currently works on Windows/NT and on UNIX (AIX and FreeBSD) systems for several long-running simulation applications, but nothing precludes use of the checkpoint facility in other environments. This paper describes our NT implementation.

2. Design Goals

To ensure application transparency of process checkpointing, it is necessary to capture the process state

and restore it later. Thus, the problem we solve is how to capture and reconstruct the process state efficiently without affecting the correctness of the application. Our implementation is portable, transparent to the user, and provides good performance.

The following high level design goals for the process checkpointing facility were followed:

- **Transparency**

  Our implementation does not require availability of user source to run the checkpoint utility. User applications need only link to the checkpoint library DLL, which will automatically change the startup routine and system call import table. Changing the startup routine allows us to inject optional checkpoint specific command line flags in the application and initialize checkpointing. Changing the system call import table allows us to wrap system API calls to preserve state across a checkpoint.

- **Correctness**

  Process execution gives the same results whether or not checkpoints are taken at runtime. Resuming a process from a checkpoint provides the same result as the original execution. We used a regression test suite that contains real application linkage as well as specific test cases to ensure that our checkpoint library is correct.

- **Minimal Performance Impact**

  The checkpoint facility writes the various memory segments of the application to a checkpoint file. Elapsed time is therefore directly proportional to process size. Figure 1 shows benchmark for a typical application sizes:

Process Size	Checkpoint File	Time to Checkpoint
10 MB	10648 KB	4 sec
20 MB	20888 KB	9 sec
30 MB	31128 KB	15 sec
50 MB	51608 KB	21 sec

Figure 1 - Checkpoint Performance

In our implementation, we wrap certain system and library API calls so that we can save state information. This adds a minimal overhead to the application. Figure 2 shows the average overhead for typical wrapped calls for 1 million consecutive calls of each function:

Library Call	Overhead
`CreateFile,CloseHandle`	6.8×10^{-5} sec
`WriteFile`	1.5×10^{-5} sec
`malloc`, `free` sequence	0 sec

Figure 2 - Checkpoint Overhead

- **Portability**

  Our checkpoint facility runs on Windows/NT, AIX and FreeBSD UNIX systems. We use a similar user-level methodology on all OS implementations, which has proved to be easily portable, but of course there are some code differences over OS implementations. Our NT implementation contains a few, small assembly routines written on Intel architecture, which are easy to port to other NT platforms. No application modifications are required.

- **Multiple Thread Support**

  Our solution supports checkpointing of multi-threaded Windows applications.

3. Architecture

The following block diagram provides a high level architecture of the checkpoint library.

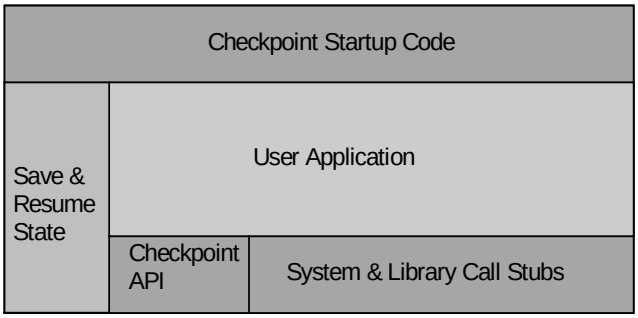

Figure 3 - Architecture

The library is implemented with two components:

1. Loader - a program that loads the user program into the operating system. We use this program because NT's virtual addresses allocations are affected by the length of the command line arguments. We need to ensure that the same virtual addressing is used at process resume regardless of the command line.

2. Checkpoint DLL - the main program that sets up wrappers to system and API calls in the user's inary, dumps the process state to a checkpoint file, and resumes execution of a process from a checkpoint file.

System and library API calls made by the application are redirected to versions provided by the checkpoint library. This allows system state held by the operating system on behalf of the process (such as open file handles) to be saved and recreated over a checkpoint. We use this "wrapper" method of saving library and system call state information, as it is portable between different operating systems.

When the operating system completes initialization of a user application, it loads the checkpoint DLL that changes the entry point of the application, so that checkpoint initialization routines will be called instead of the user program. The checkpoint DLL checks whether a process dump or resume is required. For a process dump, the checkpoint DLL updates the process import table to inject the API wrappers. Then the DLL creates a new thread for the program, which will be responsible for dumping the process and for timer notifications for periodic checkpointing. When all initialization procedures are completed, control is passed to the original user code entry point. For a process resume, the checkpoint DLL restores the state of the application when it was dumped, including threads, memory and system objects. If further checkpointing is required, the DLL proceeds as above; otherwise control is simply given to the resumed user code. Figure 4 shows a high level outline of the implementation. The checkpoint-specific arguments provide user run-time control over the checkpointing.

```
int
startup_algorithm ( )
{
   process_chkpt_args();


   if (checkpointing) {
      if (chkpt_periodic)
           set_chkpt_periodic(period);
       if (chkpt_on_signal)
           set_chkpt_on_signal(signum);
   }
   else if (resume)
      _chkpt_resume(dumpfile);

   return((int)
         _chkpt_user_main(argc,argv,envp));
}
```

Figure 4 - Startup Code

Checkpoint API calls are also available to the application programmer so that checkpoints can be requested at critical points in the application code.

As described above, linking an application to the checkpoint DLL library involves changing the application entry point. There is no need to recompile the user application or make any changes to the code.

4. Interface

The checkpoint facility provides a user-level interface and an API to the checkpoint capabilities.

4.1 Users Interface

The build process replaces the application's startup function with the checkpoint DLL startup code. As we have described, this startup code adds checkpoint specific options to the application so the user can control checkpoint behavior at run time. Regular application options are added after the checkpoint options.

The following checkpoint command line options are supported:.

- `loader <progname> -chkpt_P <period>`

checkpoint the process every *period* seconds

- `loader <progname> -chkpt_R <file>`

resumes process execution from the state previously checkpointed in *file*

- `loader <progname> -chkpt_D <dir>`

write checkpoint files to directory *dir*

- `loader <progname> -chkpt_X <file>`

extract header information from checkpoint *file*

4.2 Developer Interface

The checkpoint facility defines the following API call:

```
_chkpt_now (char *filename)
```

This function checkpoints process state to a file using the given *filename*. If passed a NULL string a standard sequential naming convention is used. In case of error, an external variable _chkpt_errno is set to the corresponding error number and may be used by the calling function.

5. Process State

Checkpointing and resuming a process should be transparent so that, as far as the application code is concerned, the checkpoint never happened.

To achieve transparency, the entire process state must be captured at checkpoint time and fully restored when the

process is resumed. Figure 6 shows the components of typical process state, which we explain from the bottom up.

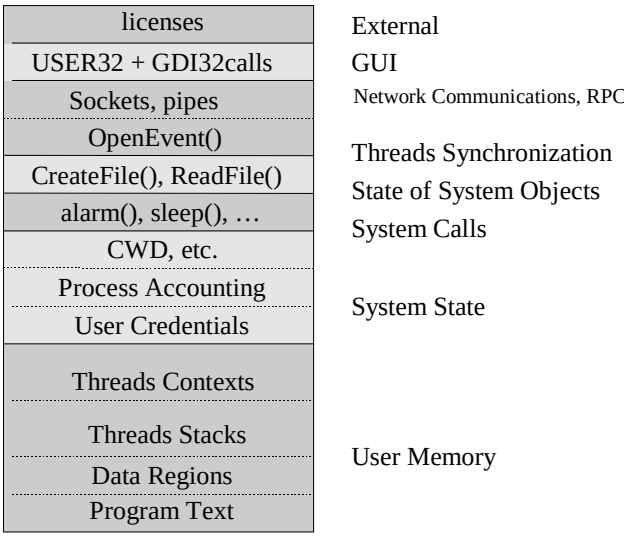

Figure 6 - Process State

Program text is the application object code, which is typically memory mapped from the executable file on disk. Data regions include statically and dynamically allocated data, for example calls to `malloc` or `new`. Program stack and the value of registers, stack pointer and program counter complete the memory components of the process state.

System state is held by the operating system on behalf of the process. It includes user credentials such as the process ID and process owner, various accounting data (for example cumulative CPU usage), and other miscellaneous state such as the current working directory.

At runtime, the process may use system calls that maintain state, for example `Sleep` suspends a thread for a specified time interval. If in the meantime the process checkpoints, the checkpoint DLL must capture this state. An interesting problem with `Sleep` is whether resume should honor the original request with respect to real time or process run time at resume. Our implementation assumes process run time, although the true definition of this API call is real time.

A process may have open files and inter-process communication channels with data in transit at checkpoint time. Graphics applications will have additional state; for example a graphic application may have called the Win32 API `GetWindowDC()` to obtain a device context. Finally some processes such as those that may have requested a floating license will have state held in some external entity.

As we move up the layers of Figure 6, process state becomes progressively harder to capture. A basic checkpoint implementation can be built by capturing only the lower user memory layers. However, a practical checkpoint facility must capture at least parts of the system state, system calls and open files.

6. Implementation

We now describe the implementation of the process checkpoint facility.

6.1 Checkpoint Initialization

The checkpoint library first needs to assume control of the program at execution time to check for any specific command line arguments and to create a special checkpoint control thread.

We use a self-modifying code technique to change the executable's entry point to gain control transparently at program startup. When a program is executed in NT, the system library calls each DLL's initialization routine (**DllMain**) before passing control to the program's entry point. The checkpoint DLL's initialization routine locates the program entry point, saves and then overwrites the existing instructions with a `jmp` instruction to a checkpoint specific startup function. The entry point is easily extracted from the in-memory process image header which can be located using the `GetModuleHandle()` Win32 API.

The checkpoint startup function runs as the program entry point and parses the command line using the global `__argc`, `__argv` symbols. If checkpoint or resume mode is requested a special checkpoint control thread is created. When running in checkpoint mode, the saved program entry point instructions are restored, checkpoint command flags are stripped and another `jmp` instruction is executed back to the original program entry point. Control passes to the user code.

6.2 Checkpoint Control Thread

The special checkpoint thread is responsible for controlling the process checkpoint (dump) and resume. This thread continuously scans its APC (Asynchronous Procedure Call) queue waiting for dump or resume requests to arrive. APC queue scanning is implemented in the OS core and does not take additional application cycles.

The library supports two types of checkpointing: periodic and on-demand (through a checkpoint library API call). In periodic mode a waitable object is created and each time

the wait period elapses, the dump APC is placed on the checkpoint thread's APC queue. With on-demand mode, the checkpoint API `_chkpt_now()` function call creates a dump APC with 0 wait period.

6.3 Process Dump

On receipt of a dump APC call, the checkpoint thread suspends all other program threads and saves their context by calling `GetThreadContext()`, then opens a new checkpoint file. A naming sequence is used to ensure that filenames are unique, and that the checkpoint order can be determined even when a checkpoint is taken in a process that was itself resumed from a checkpoint.

A header is written to the file with global information about the process such as the process name, current working directory, time of checkpoint and the command line arguments. Process state is saved to the file as described below. The checkpoint thread then resumes all suspended threads simultaneously before putting itself into a blocking wait on the next APC queue event.

The following sections describe how we capture state of the individual components of the process.

- **Text Segment**

We assume the original object file is available at resume, and so the program text segment does not need to be saved. We check time stamps on process resume to verify that the original object file has not changed.

- **Data Segments**

The checkpoint routines write the contents of the process data segments to the checkpoint file. The difficulty is to identify the start and end virtual addresses for each segment.

Static data regions are allocated when the process image is loaded. The base address of the process image is located using the `GetModuleHandle()` Win32 API. The static data regions can be located from this base address with sequential calls to the `VirtualQuery()` Win32 API. All writeable regions that have the same base allocation address contain static data. We prefer this technique over the simpler method of extracting location information on static data regions from the image header, because it is more reliable: A developer can add and change static region names at link time, making them impossible to identify.

We make an extra check, so that we do not include the static import table data, as this has been modified by our checkpoint DLL (see Section 6.4.2) The import table address and size can be extracted from the in-memory process image header.

Dynamic data segments are allocated in several ways. Heap space is allocated with `HeapAlloc()`, `GlobalAlloc()` and `LocalAlloc()` Win32 API functions. Heap allocations can be located using the `GetProcessHeaps()` Win32 `API`. For efficiency and to avoid problems at resume, all heaps expect for that used by the checkpoint DLL (allocated by the CRT library) are saved. The checkpoint DLL heap can be identified using the address of the global symbol `_crtheap`.

Dynamic memory allocations through calls to the `VirtualAlloc()` Win32 API are redirected to a checkpoint wrapper stub, which saves all address allocation information.

- **Thread Execution Context and Stack Segment**

The execution context of each thread was saved at the beginning of the dump sequence and is simply written to the checkpoint file along with the contents of each thread's stack.

The end address of each stack is located using the stack pointer (ESP) contained in the thread execution context. Appropriate calls to `VirtualQuery()` are used to locate the start address and size of the stack.

- **System State**

Information needed to reconstruct system state changed by Win32 API calls is written to the checkpoint file. This information is captured through our technique of redirecting Win32 API calls through wrapper functions discussed in Section 6.5. For each wrapped API we save the call parameters, thread ID and any call specific information.

6.4 Process Resume

An application resumes by reconstructing its state, using a previously created checkpoint file. At startup the checkpoint DLL checks for the resume command line parameter. If present, a resume APC is placed on the checkpoint thread's APC queue.

The checkpoint thread opens the file that contains the process checkpoint data and examines the file header to ensure the checkpoint was created by a prior invocation of the currently running object code. It then reads data from the checkpoint file and reinstates system state including thread creation, execution context, stack and data segments. After state has been restored, all threads are simultaneously resumed and the process continues from where it was checkpointed. We discuss these steps in more detail below.

- **Process and Thread Creation**

Process creation is implemented by invoking the original application with the `-chkpt_R` argument, which restores the original text segment and transfers control to the resume APC in the checkpoint thread.

The checkpoint thread reads saved Win32 API call state information from the file and uses the information on calls to the `CreateThread()` Win32 API routine to create a new set of threads. New and old thread ID's are saved in a special association table, allowing for translation by subsequent calls through Win32 API function wrappers.

Each newly created thread is put in a mode similar to the checkpoint thread, continuously waiting for APC's to execute.

- **System State**

For each saved Win32 API call, we use the thread association table to map the original calling threadID to a new thread. The original parameters and name of the API function to call are sent to the thread using `QueueUserAPC()`. The thread executes the appropriate Win32 API function. API calls are sent to the threads in the same order they were originally called.

After all saved Win32 API calls have been re-invoked, a special APC is sent to each thread, which respond by executing `WaitForSingleObject()` on a common event object. This object is used to wake all threads simultaneously once all process state has been recovered.

- **Data Segment and Thread Context Resume**

The checkpoint thread now restores data and stack segments. Data is read from the checkpoint file and written directly to memory to the saved addresses of each process region. When all memory related data is recovered, thread contexts are restored using the `SetThreadContext()` API.

- **Process Control Resume**

At this point all state has been restored. All threads are waiting on a single event object and their contexts, including their instruction pointer, are set to the checkpointed values. The checkpoint thread sets the common event object and all the threads simultaneously resume from where they were suspended by the original dump APC. Finally, the checkpoint thread puts itself to sleep, and waits for further Dump APC calls.

6.5 System Calls

Some Win32 API calls such as `CreateThread()`, `CreateFile()`, and `CreateSemaphore()` change system state for a process. These API calls present a problem for checkpointing, since they represent state that is held within the operating system on behalf of a process. Without privileged access to kernel data space, this state is difficult to capture and restore.

To remove the need for privileged kernel access, we adopt a technique of redirecting certain Win32 API calls through function wrappers. The wrappers save enough call information that we can recreate state upon resume. For example knowing the name of an open file and the current offset, we can reopen the file during resume, seek to the saved offset, and craft the original file handle to correctly access the reopened file in all subsequent Win32 API calls.

It is important to do this at the Win32 API interface (KERNEL32.DLL) boundary so that we do not need to deal individually with the thousands of library functions provided by the NT programming environment. For example by wrapping the `VirtualAlloc()` Win32 API call we do not need to deal with the `malloc` family of library functions.

6.5.1 Redirecting Win32 API Calls

In Windows NT, each API call is redirected by the linker to the IAT (Import Address Table), from where another jump is taken to reach the real API function handler. The basic technique of wrapping Win32 API calls is to change the addresses in the IAT, so the second jump will lead to the appropriate checkpoint wrapper, which will collect necessary information about the call before the real API is executed.

One method to do this would be to simply redirect the call to our routine by changing the symbol. Our routine could then call the Win32 API directly. However this method does not work because the checkpoint routine adds a frame to the stack, and the real API will get an incorrect stack pointer. Registers may also be changed by the checkpoint wrapper routine. To avoid this problem, we again used a self-modifying code technique. For each wrapped Win32 API, a small code fragment is created at run-time, where we save the registers and execute the corresponding checkpoint wrapper function. After the wrapper returns, the registers are restored, and control is passed to the real API handler through a `jmp`.

Since the real Win32 API function is now called from the same stack frame as if it was called directly, the checkpoint wrapper redirect is transparent to the user code and the Win32 API function.

6.5.2 Supported System Calls

We support the following important Win32 API functions in the current version of the checkpoint library:

EnterCriticalSection, InitializeCriticalSection, LeaveCriticalSection
CreateThread, GetStdHandle
HeapCreate, HeapAlloc, VirtualAlloc
CreateFile, ReadFile, WriteFile, CloseHandle

Figure 7 – Checkpointed Win32 API Functions

Another 26 functions are tested, and could be used with the checkpoint facility.

7. Comparison to Similar Work

There are several existing solutions to the checkpoint and process migration problems on UNIX, and we choose to discuss a few of them. We could not find any similar work on NT.

7.1 MPVM

MPVM, an extension of *PVM*, is a research project at Oak Ridge National Laboratory that allows parts of a parallel computation to be suspended and subsequently resumed on other workstations by migrating process state from one machine to another. Migration transparency is addressed by modifying PVM libraries and daemons and by providing wrapper functions to certain system calls so that the migration occurs without modifying the application code. The migration mechanism is implemented at user level. *MPVM* has the following limitations:

- The developer must explicitly create executable files that are statically linked to support shared libraries.
- *MPVM* assumes use of a global file system.

7.2 Condor

Condor is a batch facility running on UNIX systems that allocates processes to idle work stations. It performs process migration by checkpointing a process to a file, transferring the file, and then restoring the process from the checkpoint file. No special programming is required, but user applications need to be re-linked with *Condor* libraries.

Condor has several limitations:

- Condor does not support all system calls and library calls. Signals and signal handlers are not supported, and *popen* is not supported..
- All file operations must be idempotent - read only and write only file accesses work correctly, but programs which read and write a same file may not checkpoint transparently.

7.3 Libchkpt

Libckpt shares many goals of our checkpoint facility, but they do not support a complete set of system wrappers. They support file system calls such as open, close, read and write, but they do not support popen, or signals. Our UNIX version of the checkpoint facility does support popen and signals.

7.4 MOSIX

MOSIX is a multi-computer operating system that supports transparent, preemptive process migration and load balancing for efficient utilization of overall resources and to balance work distribution. The MOSIX enhancements are implemented at the operating system kernel level without changing the UNIX interface, and therefore it is completely transparent to the application level. It uses *PVM* as the distribution engine. The process migration in MOSIX is dynamic and preemptive, that is it responds to variations in workstation load by migrating processes from one node to another at any stage of the life cycle of a process. The granularity of the work distribution in MOSIX is the UNIX process. The processors must be homogeneous (from the same family) to allow process migration. MOSIX has the following limitations:

- It is implemented at the kernel level and therefore it is more complex and requires source code availability.
- It is a preemptive system that does not provide the capability to dump/resume user processes at arbitrary times.

8. User Experience

Many internal Intel simulators and program environments use the checkpoint facility on Windows NT and UNIX (AIX and FreeBSD) systems. Developers typically checkpoint their work every hour, permitting them to focus on a problem area that is discovered deep into a run. When a bug is discovered, they resume execution from the nearest checkpoint, sometimes set up finer-grained checkpoints, and have been able to debug their programs with greater ease than was the case before use of the checkpoint facility.

9. Limitations

The following limitations exist in our current checkpoint implementation.

9.1 External File Persistence

Where the runtime environment of a program depends on some external data such as a file or network connection, the checkpoint facility cannot correctly restore state if the corresponding media is unavailable or modified at process

resume. For example if some input file is deleted after a checkpoint is taken, the resume process will be unable to reopen the file and recreate the file handle that may be necessary at a later stage in the process execution.

9.2 Direct System Object Access

The current design of the checkpoint utility cannot save the state of system objects that were changed in a way other than calling Win32 system APIs (provided by KERNEL32.DLL). For example:

- The checkpoint utility can't set API wrappers on dynamically loaded system APIs that are called using the pointer returned by `GetProcAddress()`. Those calls don't go through the import table, and so our current method of wrapping calls will not work.
- Checkpoints may fail if a user program calls undocumented NTDLL.DLL services or calls "`int 2Eh`" to get system services.
- We cannot checkpoint kernel mode drivers or programs that dynamically modify kernel drivers.

9.3 Direct System Data Access

The checkpoint facility assumes that user programs do not manually modify system data that is maintained in virtual space of the program by system DLLs and the operating system. For example: the data segment of KERNEL.32 DLL. Such modified data will not be restored.

10. Future Development Plans

10.1 Optimization

The current version of the checkpoint system dumps the entire process state for each checkpoint call. This requires time proportional to the size of the process, and can clearly be an expensive operation for large processes. The developers of the Libchkpt checkpoint system [4] note that it is possible to speed up checkpointing procedures by dumping only those pages whose data has changed since the previous checkpoint call. We may optimize our checkpoint facility to dump incremental changes to the process state by using the `VirtualProtect` API to write-protecting pages using the GUARD_PAGE facility to mark the page as accessed, then writing only marked pages at the next checkpoint call.

10.2 Multithreaded API Call Support

As discussed in section 6.5, we support checkpoint of applications that use system API calls by call redirection through our code, which captures calling thread ID, call parameters and return values. We then use this data to reconstruct system state by re-calling the system API functions from the context of the original thread at process resume. At checkpoint, all threads are suspended regardless of whether or not they are in the middle of a system API call.

This technique can be problematic as demonstrated in the following example. Suppose a program thread calls the `HeapAlloc()`API to allocate heap memory. This API will reserve a memory segment for the user data and write some meta-data structures to memory for managing the allocation. If we suspend the thread in the middle of this API, the checkpoint thread will have recorded that there was a call to `HeapAlloc()` and the checkpoint routine that saves memory segments may write the allocated segments to the checkpoint file before the `HeapAlloc()` API has finished writing the meta-data.

Upon resume, we will restart all the API calls, and `HeapAlloc()` will be called and setup the correct memory segment and meta-data structures. However the checkpoint code will then restore memory segments, which will overwrite the good meta-data structures with the in-complete saved structures.

We need to get a better solution to restore threads that were check-pointed part way through system API calls. One possible direction is to write an NT kernel driver which would be able to save the exact instruction at the time of checkpoint and at resume place a breakpoint at the same instruction. When all state has been restored and the program threads are resumed, they would continue at the exact point where they were check-pointed.

10.3 Enhanced System Calls Support

The type of applications the checkpoint library supports are heavily dependent upon the range of system API calls that can be supported over a checkpoint resume operation. In our work we support system calls used by a number of internal applications.

Of particular interest, but highly difficult to implement, is support for system API calls that involve multiple processes such as those used for named pipes, Windows sockets, RPC, and COM Interfaces. It is difficult to checkpoint several processes simultaneously and to capture data that may be in transit between processes (such as on a network), and later restore state transparently. Some distributed process checkpoint research has been done, but is generally application specific.

11. Conclusion

Intel engineers run many simulators for research, specification and validation of new chips. Simulators

typically run a long time, sometimes for weeks, to produce results. It is not uncommon for simulations to fail to complete because of system crashes, environmental failures such as electricity outages or, last but not least, programmer bugs. Employment of checkpoint procedures minimizes the costs of these failures by retaining results of a run until the last good state before failure, by permitting execution to restart from an advanced point of execution rather than from the beginning of the run, and by providing an advanced state for debugging that permits programmers to correct their programs more easily for future long-run use.

This paper described the implementation of a checkpoint facility that is being used in applications that run on NT systems. The checkpoint facility is a general purpose library that can be linked and used with many applications, saving developers the need to develop ad hoc solutions to checkpoint their programs. The checkpoint facility runs transparently to the application, and programmers do not have to change source code to obtain process checkpoints. Users have great flexibility in choosing the names and locations of checkpoint files and the frequency and circumstances under which checkpoints are created.

Our users have reported great success with our checkpoint facility, primarily in debugging long-running processes.

Acknowledgment

We would like to thank those who supported and used our development of checkpoint facility: Avi Giora, Shalom Goldenberg, Ariel Berkovits, Yosi Mor, Amit Dagan, Yaron Sheffer and Eric Koldinger.

References

[1] Jeremy Casas, et al. MPVM - A Migration Transparent Version of PVM. Computing Systems, vol. 8, no. 2, pp. 171-216, Spring 1995.

[2] Allan Bricker, Michael Litzkow, and Miron Livny: Condor Technical Summary, Version 4.1b, *University of Wisconsin - Madison, 1991*.

[3] Barak A., Braverman A., Gilderman I. and La'adan O., Performance of PVM with the MOSIX Preemptive Process Migration, *Proc. 7th Israeli Conf. on Computer Systems and Software Engineering, Herzliya, pp. 38-45, June 1996*.

[4] J.S. Plank, M. Beck, and G. Kingsley, Libckpt: Transparent Checkpointing Under Unix, *1995 Usenix Conferencee*.

[5] M.J. Litzkow. Remote UNIX: Turning Idle Workstations into Cycle Servers. In *Proc. USENIX summer '87, Phoenix, Arizona, June 1987.*

[6] K.I. Mandelberg and E. Sunderam. Process Migration in UNIX Networks. In *Proc. USENIX Winter '88, Dallas, Texas, February 1988.*

[7] R. Alonso and K. Kyrimis. A Process Migration Implementation for a UNIX System. In *Proc. USENIX Winter '88, Dallas, Texas, February 1988.*

[8] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In Proc. 8th Int. Conf. On Distributed Computing Systems, San Jose, California, June 1988.

[9] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under UNIX. In *Proc. USENIX Winter 1995, New Orleans, Louisiana, January 1995.*