Porting SIMLIB/C++ to 64-bit Platform

Peringer Petr*
peringer@fit.vutbr.cz

Abstract: This article deals with porting of SIMLIB/C++ (SIMulation LIBrary for C++) from 32-bit to 64-bit environment. SIMLIB/C++ is a freely available open source simulation software tool usable for education and research. Main problem of porting the code was the non-portable implementation of cooperative threads (sometimes called ”coroutines”), which we use for parallel process modelling. The article contains basic performance comparison of 32-bit and 64-bit code using simple discrete and continuous simulation models. The results show we can achieve significantly better performance of simulation tools in 64-bit environment.

Keywords: simulation tool, SIMLIB/C++, process, coroutine, user-level thread, setjmp, longjmp, x86-64, 64-bit code performance

1 Introduction

The modelling and simulation practice depends on availability of suitable simulation software. There is a place not only for complex simulation environments like DYMOLA or SIMULINK, but also for simple simulation tools. Our small tool is SIMLIB/C++ — simple simulation library usable for discrete and continuous simulation with some extensions for simulation of special kinds of models (3D, fuzzy).

In this article we describe the implementation changes needed for using SIMLIB/C++ in 64-bit environment. We also show the results of basic performance testing in 32-bit and 64-bit environments.

2 Motivation and Current State of Development

The development of SIMLIB/C++ started in 1991 on 16-bit platform (MSDOS); later it was ported to 32-bit (Linux, Windows 95) environment. The original code designed during early 90's (before C++ standard was published) is now outdated. We want to improve overall library design, bring some new concepts of C++ programming, and release new version of library, which will solve most of the problems accumulated over time.

The 64 bit computers are becoming the main line of personal computing. Almost all currently sold PCs are equipped with 64-bit processors. Because of not very good 32-bit processor architecture traditionally used in PCs (`x86`) there are performance advantages gained by moving to 64-bit code. The main advantages of 64 bit platform (`x86-64`, also called `AMD64` or `EM64T`) from scientific computing point of view are:

- `x86-64` processor has more registers available (16 general purpose registers and 16 SSE registers). This allows the compiler to do better optimization of code.

* Department of Intelligent Systems, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 612 66 Brno–Královo Pole, Czech Republic

- Program can access more memory directly (pointers are 64-bit, possible address space is up to 2^{64} bytes in comparison with 2^{32} in 32-bit systems).

The process of porting to 64 bit architecture is described in many publications (for example [1] or [5]). Basically we should:

- check all data types which changed size (for example pointers in C++ programs are 32-bit in old code and 64-bit now) and alignment rules in the new environment,
- use `size_t` instead `int` or `long` for storing the size of objects in bytes,
- rewrite all processor dependent code (for example inline assembly code),
- test and debug the new code.

Porting of clean code written in high level languages is not big problem – only small parts of specialized code should be corrected or rewritten, which is the case of process switching code in SIMLIB/C++. This part of code is not portable, because it uses some inline assembly language code and direct access to stack contents.

There are many implementations of user-level threads [3] available as libraries. They use various methods of switching the stack, but all are more complex and use more memory per thread than our implementation.

3 Class Process Implementation

The implementation of specialized threads for simulation purposes can be simpler than general user-level threads implementations. Simulation processes can be implemented using cooperative multi-threading, which eliminates many problems. Switching process context in simulation software can be done in three possible ways:

- Stack switching – usually using `setcontext`/`getcontext` family of system calls which conforms to the *Single Unix Specification* standard [7]. More information can be found in [3]. This approach is currently not used in SIMLIB, but we plan to use this as alternative implementation for improving portability.
- Use operating system provided threads – this approach is the most portable, but slower and less memory efficient. This implementation can allow parallel execution of simulation models on multiprocessor machines, but the implementation of simulation tools this way is complex. We plan to try boost::threads [2] in SIMLIB in the future because it is possible, that boost::threads will be part of the next C++ standard.
- Save/restore stack – copy stack contents of interrupted thread into heap-allocated storage and restore it back before it will continue. Processor context can be read/changed using standard `setjmp`/`longjmp` functions. This approach is not portable because we need small inline assembly code for getting and setting of stack pointer. This is the main problem of porting SIMLIB/C++ to 64-bit platform.

Current version of SIMLIB/C++ uses ”save/restore stack” approach. It was used in initial MS-DOS version because of small memory requirements. The typical amount of memory needed for storing the stack contents is hundreds of bytes per running instance of class Process. Low memory overhead is the main advantage of this approach – we store only currently used data on stack, which is different from all other methods which use fixed stack size for each thread. The main disadvantages of this approach are:

- Non-portable implementation – we need stack manipulations (see Fig. 1), which can cause implementation problems on some platforms.

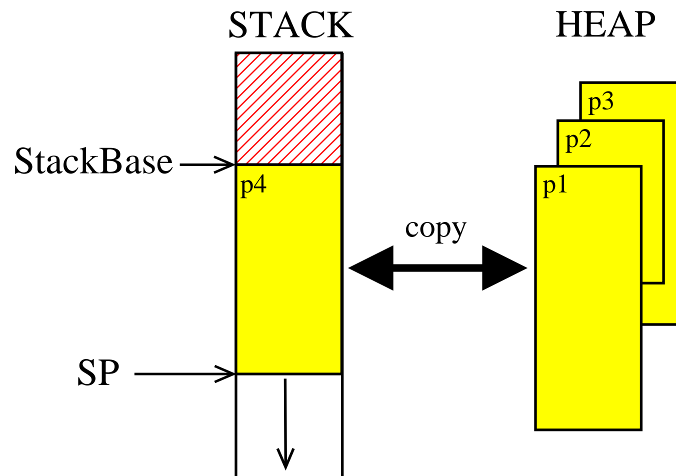

Fig. 1: Save/Restore stack contents of threads

- Speed – this implementation can be slower, because of memory copying. The amount of memory depends on amount of local variables in method `Behavior()`[1]. It is not recommended to use big local variables, best is to store such data into objects as attributes. The benchmark results later in this article show the dependence of speed on the size of local data.

3.1 Process dispatcher

The code of simulation system consists of simulation-control algorithm (we use Next-Event) and special function for process dispatch, which starts/continues behavior of active processes. The pseudo-code explains the function:

```
Dispatch:
1    Stack_Base = SP;     // save/check current stack pointer
2    if(setjmp(Base_context)==0)
         if(Current process has non-empty process context)
3            Restore stack contents
4            longjmp(Proc_context, 1); // go back to Behavior()
         else
5            Behavior(); // start of process behavior description
     else
6        // after Behavior() was interrupted
```

The only place, which can start/continue process behavior is the dispatcher. The reason for this is the `Stack_Base` variable, which should be constant during simulation. This variable contains (1) start of stack area to save/restore (stack usually grows down, see figure 1).

First call of `setjmp(Base_context)` stores (2) the processor context[2] into global variable `Base_context` and returns zero. After setting the `Base_context` we test whether cur-

[1] `Behavior()` is pure virtual method of class `Process`, which should be defined in each derived class. The method defines behavior of processes used in discrete simulation.

[2] The stored context is important for jump back from `Behavior()` as we will show later. After this jump it seems like `setjmp` returned again, but now it returns non-zero value, and we can distinguish those two returns.

rent process has the stored context. If not (5), it should start `Behavior()`, if there is the stored context (3) we should restore stack using `memcpy` and processor context (4) using `longjmp`. There are some implementation problems:

1. We should shift stack pointer (SP) outside overwritten area before copying the stored data back into stack, because the call of function `memcpy` uses stack for arguments and return address, and those should not be overwritten.
2. We can not use local variables in this area of code, because they can be referenced using stack pointer we change (it depends on compiler, level of optimization, and platform).

Call of `longjmp` never returns back — it continues with new context. If the call of `setjmp` returns with non-zero value (6) it is the case of process behavior interrupted by calling a special method.

3.2 Interrupting the Behavior() method

The Behavior() method can call other special methods able to interrupt running function and later come back. We show the implementation of `Wait()` method as the typical example:

```
Wait(dt):
1    Calendar.Schedule(this process, at Time + dt);

2    Compute size of stack data (from current SP to Stack_Base)
3    Allocate new process context
4    Save stack (from current SP to Stack_Base)
5    if(setjmp(Proc_context)==0)
6        longjmp(Base_context,1); // go back to dispatcher
     else // coming back from dispatcher
7        delete process context
```

The code for storing process context (2-7) is the same in all functions which can interrupt running `Behavior()`. First it reads the stack pointer `SP` and computes (2) size of stack area to save (`size = Stack_Base - SP`). Then (3) it allocates memory and does the copy (4) of stack contents to allocated memory. The pointer to the data is saved into process context in the object attributes.

Then the `setjmp` is called (5) and processor context saved. This call to `setjmp` returns zero. Then (6) the call to `longjmp` jumps back to dispatcher using `Base_context`. The process state is stored and can be used for restoring the state of process at the time of its re-activation by dispatcher. The call of `longjmp` by dispatcher leads to second return (5) from `setjmp` in `Wait()`, but now with non-zero return value. Then is the current process context removed (7), because is not valid anymore. The temporarily interrupted `Behavior()` method continues after return from `Wait()`.

This implementation is simple and works well at PC architecture, but there are other architectures, on which this implementation can be a problem. There is room for improvements and optimizations – for example frequent allocation and deallocation of process context can be improved by caching allocated blocks and reusing suitable sized blocks instead of new allocation.

4 Experimental results

We measured performance of the code in various benchmarks. For all benchmarks we used the PC class computer with Athlon64/3000+ and 512MiB of memory with Debian/GNU Linux version 4.0/Etch (Linux kernel 2.6.18-4-amd64). The compiler used is GCC version 4.1.2 with optimization level `-O2`.

The methodology we used is very simple: run programs 10 times, remove two slowest and two fastest measurements and average the remaining 6 values. The absolute numbers are not very important, because we want only relative performance comparison of 32-bit vs 64-bit environments.

4.1 Performance of process switching

First benchmark measures the time of one million context-switches[3]. Results of experiments are in the table (lower time is better):

Local data size	time [s]		difference
[Bytes]	32-bit code	64-bit code	[%]
0	1.02	0.71	-30
10	1.02	0.70	-31
100	1.08	0.76	-30
1000	1.57	1.05	-33
10000	7.31	4.14	-43
0 (Events)	0.14	0.08	-43

The table shows at least 30 percent improvement of process switching and scheduling performance in 64-bit environment. For comparison we measured equivalent code using events (without process switching code overhead). The use of events leads to more complex model description, but as results show, it is significantly faster.

4.2 Memory requirements

The amount of memory used by processes is tested by program, which generates and activates N simple processes without local variables at the same time. Then each process starts its `Behavior()` method, does `Wait(1)` operation and ends. There is N interrupted processes with saved context at once during this test. We estimated the possible maximum number of processes running simultaneously in memory limited to 400MiB (using command `ulimit -v` at Linux). Then we measured run-time of simultaneous creation, activation, `Wait(1)` and deletion of half million processes:

Benchmark	32-bit code	64-bit code	difference [%]
Number of processes created in 400MiB	1 100 000	880 000	-20
Average amount of memory per process [B]	381	476	+24
Time [s] for running 500 000 processes	1.19	0.84	-29

The results show increased memory consumption and faster execution of code in 64-bit environment.

[3] Process1 is interrupted, reactivation scheduled, context1 saved, found next Process2 record in calendar, context2 restored, Process2 can continue.

4.3 Performance of numerical methods

The performance of numerical methods code used by continuous simulation also improved in 64-bit implementation. Our simple benchmark uses continuous system simulation solving the differential equation $y'' = -y$, with initial conditions $y = 0$, $y' = 1$ (so-called ”circle test”). In the table is the time spent by one million steps using various numerical integration methods (lower time is better):

Integration method set using SetMethod(”name”)	time [s]		difference
	32-bit code	64-bit code	[%]
abm4	0.66	0.41	-38
euler	0.78	0.53	-32
rke (default)	1.70	1.12	-34
rkf8	2.43	1.56	-36

Table shows that our code compiled for 64-bit platform is at least 30 percent faster. The reason is that 64-bit processor allows the compiler better optimization of code containing floating-point instructions.

5 Conclusions

As we have seen in this article, there are some problems associated with conversion of 32-bit code to 64-bit computing environment, but we get better performance (at least with the code used in simulation). The conversion of our code to `x86-64` architecture was simple, except process switching code which needed some changes. Our simple benchmarks show, that 64-bit code can be up to 40 percent faster than 32-bit one on the same machine.

All the work described here is only small part of bigger effort which should result in new version of SIMLIB/C++. The process switching code will be implemented in at least two ways in final version to achieve better portability. We also plan the optional use of GSL (GNU Scientific Library [4]) as numerical methods backend, which should significantly improve the usability of SIMLIB/C++.

This work has been supported by the Research Plan No. MSM0021630528 ”Security-Oriented Research in Information Technology”.

Bibliography

1. 64-bit development resources, `http://www.viva64.com/links.php` (Jun 2007)
2. Boost WWW page: `http://www.boost.org/` (July 2007)
3. Engelschall, Ralf S.: *Portable Multithreading-The Signal Stack Trick for User-Space Tliread Creation*, In Proceedings of the USENIX Annual Technical Conference, San Diego, California, pages 239–250, June 2000
4. GNU Scientific Library WWW page: `http://www.gnu.org/software/gsl/` (July 2007)
5. Porting Linux applications to 64-bit systems `http://www.ibm.com/developerworks/library/l-port64.html` (Jun 2007)
6. SIMLIB/C++ WWW page: `http://www.fit.vutbr.cz/~peringer/SIMLIB/` (2007)
7. The Open Group: The Single UNIX Specification, Version 3, `http://www.unix.org/version3/` (Jun 2007)