High-Performance-Multitasking in universellen Programmiersprachen als Basis für flexible Simulationssysteme

Thomas Wiedemann
wiedem@informatik.htw-dresden.de
HTW Dresden Fachbereich Informatik /Mathematik ,
F.-List-Platz 1 D-01069 Dresden, Germany

Kurzfassung

Trotz der in den letzten Jahren um Größenordnungen gestiegenen Taktraten der Personalcomputer wird immer noch über zu lange Laufzeiten von Simulationsexperimenten berichtet. Dadurch können aufsetzende Optimierungsläufe mit einigen Hundert Simulationsläufen nur schlecht oder gar nicht durchgeführt werden. Ein Grund für diese Situation ist die stark zunehmende Implementierung von Simulationssystemen auf der Basis interpretierter Sprachen wie Java oder die Verwendung wenig effizienter Prozess-Steuerungsalgorithmen.

Der vorliegende Artikel zeigt auf, wie durch eine Rückkehr zu Multitasking-Routinen in Assembler eine volle Ausnutzung der Leistungssteigerung der Prozessoren erreicht werden kann. Durch eine Integration kleiner Multitasking-Funktionen in Standard-Entwicklungsumgebungen wie Delphi oder Visual C++ können hochperformante Simulationsmodelle mit einem Maximum an Flexibilität generiert werden.

1 Einleitung

In jedem Jahr werden innerhalb der Simulations-Community neue Entwicklungssysteme vorgestellt. Als Ursache für diese ständigen Neuentwicklungen kann die allgemeine Unzufriedenheiten der Entwickler mit den verfügbaren Systemen bezüglich Leistung, Preis, Flexibilität und Performance gesehen werden. Dabei insbesondere fällt auf, daß neuere Simulationswerkzeuge in der Regel langsamer werden, da dem Performanceaspekt bei der Neuimplementierungen geringere Bedeutung als bei Vorgängersystemen beigemessen wird. Es kommt zum Paradoxon, daß sehr alte Simulationssprachen wie GPSS aufgrund ihrer deutlich besseren Laufzeiteigenschaften immer noch für Simulationsuntersuchungen eingesetzt werden [1].

Bei der Verwendung von Java als Entwicklungsbasis wird zum Umschalten der Simulationsprozesse meist der in Java implementierte Thread-Mechanismus verwendet. Untersuchungen von Kilgore [2] zeigten jedoch, daß dieser Mechanismus auf etwa 30.000 Prozesse begrenzt ist und zudem Instabilitäten und schlechte Laufzeiteigenschaften zeigt. Dies ist nicht verwunderlich, da Threads in der Regel für relativ große Prozesse vorgesehen und die sehr häufige Umschaltung sehr vieler kleiner Prozesse nicht in der Intention der Java-Entwickler bei SUN lag.

Es gibt allerdings auch bei in nativen Prozessorcode übersetzenden Systemen wie C++ oder Delphi Probleme bei der effizienten Entwicklung von Simulationsprogrammen. Eine Hauptursache ist die fehlende Koroutinenfähigkeit heutiger Standardcompiler.

Während diese Koroutinenfähigkeit bei SIMULA aus den 60er Jahren noch eine Standardfunktionalität war, ist bei heutigen Standardcompilern ein derartiges Umschalten der Programmausführung innerhalb einer Funktion A zu einem anderen Punkt innerhalb einer Funktion B nicht möglich. Als Alternative verbleibt nur das Umschalten von ganzen Funktionen über pointerbasierte Funktionsaufrufe. Da dies wiederum eine Art Interpreter erfordert, kommt es zu Leistungseinbussen und verschlechtert sich insbesondere die Lesbarkeit der Syntax bei der Modellierung. Ein entsprechendes System wurde durch den Autor seit 1997 implementiert [3]. Die als nicht optimal eingeschätzte Situation und fehlende Alternativen führten zur Frage, ob nicht durch einen unterhalb der Standard-C-Funktionen angesiedelten Mechanismus auf Prozessorebene eine Lösung denkbar wäre. Dazu sollen die konkreten Anforderungen heutiger, diskreter Simulationssystem kurz diskutiert werden.

2 Simulationsbezogenes Multitasking

Bei einer diskreten Simulation wird die in der Regel einmal vorliegende Modellbeschreibung eines Objektes in einigen Tausend bis Millionen Instanzen gestartet. Wenn ein aktives Objekt in der Simulation zeitlich verzögert wird, muß zum nächsten Objekt innerhalb der zeitlichen Prioritätsliste umgeschaltet werden. Wenn die zeitliche Verzögerung abgelaufen ist, muß genau zu diesem Objekt und genau einen Befehl nach dem Verzögerungsbefehl (z.B. advance) zurückgeschaltet werden. Bei mehrstufigen Fertigungsprozessen kann ein sehr häufiges Umschalten innerhalb eines Produktes erforderlich sein (vgl. Bild 1).

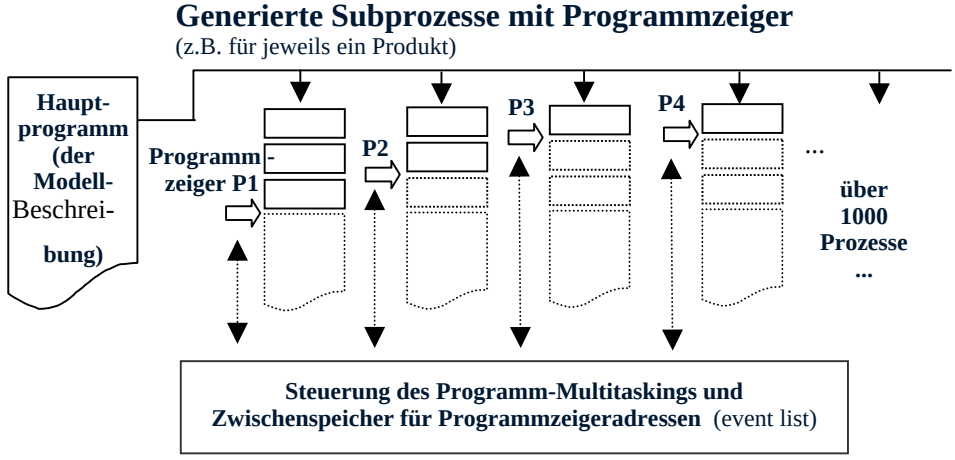

Bild 1: Multitasking in diskreten Simulationsmodellen

3. Lösungsoptionen für Multitasking und Scheduling

3.1 Extremes Multitasking auf Stackebene

Ausgehend von der Orientierung auf eine Trennung von Multitasking und Scheduling wird zuerst ein effizientes Multitasking diskutiert.
Die offene Schnittstelle zwischen Multitasking und Scheduling kann relativ einfach mit

switchprocesses (long ProcId , long NewProcID);

definiert werden. Die entsprechende Funktion schaltet den aktiven Prozess mit ProcID auf den neuen Prozess mit der ID = NewProcID um.

Die spezifischen Daten eines Simulationsobjektes werden im Regelfall als lokale Variable auf dem Stack gehalten oder über Pointer im globalen HEAP referenziert. Das Umschalten von einem Prozeß zum nächsten erfordert zwingend die Sicherung dieser Daten, da sonst keine korrekte Weiterarbeit nach einer Rückkehr zum Prozeß mehr möglich wäre. Im Normalfall werden die Daten in einen anderen Speicherbereich kopiert

Es gibt jedoch noch eine andere Option: Da lokale Daten auf dem Stack relativ zum aktuellen Stackpointer adressiert werden, ändert sich bei einer Änderung der Stackpointeradresse sofort auch der aktuelle Prozesskontext. Im Vergleich zum Kopieren von einigen Hundert Byte mit etwa einem Takt pro Kopieraktion ist das Umschalten der Stackadresse eine Operation auf dem Stackregister mit einem einzigem Takt oder ohne jeglichen Zeitverlust bei paralleler Ausführung zu einem anderem Befehl verbunden. Das einzige Problem bei dieser Vorgehensweise ist der Verbrauch an Speicherplatz, da für jeden Prozess ein eigener Stackspeicherbereich vorhalten werden muß. Angesichts von RAM-Ausstattungen normaler Rechner im Gigabytebereich erscheint dieser Ansatz jedoch vertretbar.

Es muß angemerkt werden, daß ein Modifizieren des Stackpointers eine relativ kritische Operation ist, da die Rücksprungadresse ebenfalls auf dem Stack abgelegt werden. Bereits ein Versatz um eine Adresszelle würde eine völlige falsche Adresse vom Stack in den Befehlszähler laden und damit das Programm im Regelfall zum Absturz bringen.

Anderseits wird genau diese Operation zum Umschalten der Prozesse benötigt. Aus der Sicht der zweier umzuschaltender Prozesse taucht der Prozessor im laufenden Prozeß in der speziellen switchprocesses – Funktion ab und taucht nach dem Modifizieren des Stackpointers im zweiten Prozeß wieder auf. Erstaunlicherweise akzeptiert bei dieser berechtigterweise als „extrem“ zu bezeichnenden Prozessumschaltung selbst der Delphi-Debugger im Schrittmodus die Vorgehensweise und führt das Umschalten korrekt aus.

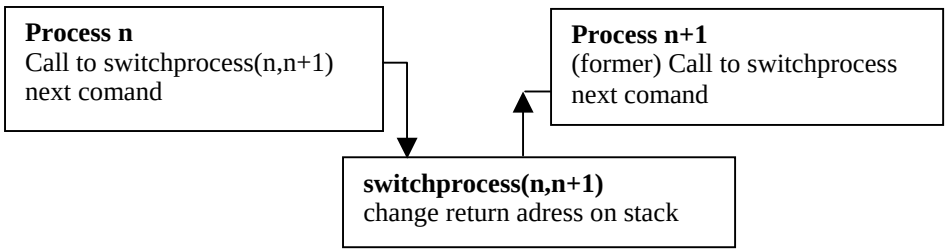

Bild 2 : Das Umschalten zwischen den Prozessen

Eine voll funktionsfähige Version dieser Multitaskingroutine zeigt Bild 3. Nach einem Sichern der möglicherweise belegten Prozessorregister wird aus einer Tabelle cal[] die Stackadresse cal[NewProcID] des neuen Prozesses geholt und nach einem Sichern der bisherigen Stackadresse in cal[OldProcId] auf das Stackpointerregiester geschrieben. Danach ist bereits die Stackumgebung des neuen Prozesses aktiv und es erfolgt nach einem Wiederherstellen der Registerinhalte der Rücksprung mit RET. Der Return-Befehl führt dabei das Wechseln der Programmablaufsteuerung zum neuen Prozeß aus.

```
procedure switchprocesses(OldProcId: integer; NewProcID:integer );
begin asm   push eax  // save calling environment
          push ebx
          push ecx
          push edi
          mov stackold,esp; end; // store old STACKP
       stacknew := cal[NewProcID];
       cal[OldProcId]:= stackold;
       asm mov  esp,stacknew; // get new STACKP
          pop edi
          pop ecx
          pop ebx
          pop eax  // get old environment
       end;
end; // RET = SWITCHING!
```

Bild 3: Eine lauffähige Multitasking-Routine im Assembler /Delphi -Mischcode

3.2 Ein einfacher, sequentieller Scheduler

Einen zum obigen Multitasking passfähigen Scheduler zeigt Bild 4. Es handelt sich dabei um die einfache Form der sequentiellen Umschaltung zu allen Prozessen. Durch die Trennung von Multitasking und Scheduling können bei gleichen Interface beliebige weitere Schedulingalgorithmen realisiert werden und wahlweise eingesetzt werden.

```
function scheduler_enumall(ProcId: integer;NewProcID:integer):longint;
begin newsimobid := actsimobid +1;
     if   newsimobid> SimobCount then    newsimobid :=1;
     if sobs[newsimobid].State <> Active then exit;
     actsimobid := newsimobid;
     actsimob   :=sobs[actsimobid];
     switchprocesses(0,newsimobid); // switch from MAIN to next
process
end;
```

Bild 4: Die einfachste Scheduling-Routine in Delphicode

3.3 Ein TREE/ARRAY-basierter Scheduler

Bei sehr vielen, weit auseinander liegenden Simulationsevents stößt der einfache sequentielle Scheduler an seine Grenzen. Einen Ausweg bietet ein Scheduler mit einer gemischten TREE/ARRAY-Verwaltung der Events. Basis des Ansatzes ist die Zerlegung der als 32-BIT-Ganzzahl abgelegten Simulationszeit in 4 einzelne Bytes. Jedes Byte dient als Offset in einen aus 256 Einträgen bestehenden Vektor von Pointern. Die Vektoren auf der 1. Ebene verweisen dann direkt auf die Simulationsevents oder auf eine einfache Liste von Events mit gleichem Zeitpunkt (vgl. Bild 5). Dem Vorteil einer schnellen Eintragungsroutine mit immer nur genau 4 Speicherzugriffen steht natürlich wieder ein erheblicher Speicherbedarf gegenüber. Allerdings werden die bei einer Vollbelegung theoretischen 4 Gigabyte nie erreicht, da typische Simulationen analog zu einer Bugwelle nur im Nahbereich entsprechende Events generieren und diese mit etwa 3 Megabyte auskommen.

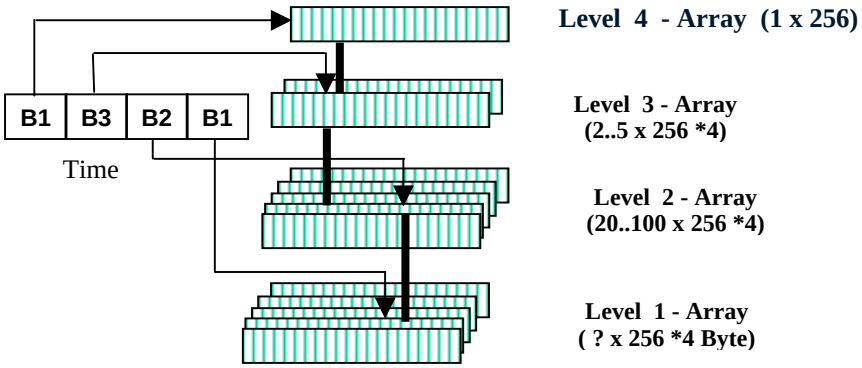

Bild 5: Der TREE/ARRAY-Scheduler

3.4 Das Extreme-Multitasking im Performancevergleich

Die in Abschnitt 3 vorgestellten Algorithmen für das Multitasking und Scheduling aus Assemblerbasis benötigen zusammen eine Umschaltzeit von etwa 20 Nanosekunden auf einem 1,5 GHz-Rechner. Dies entspricht ca. 30 Takten des Prozessors. Es dürfte selbst aus der Sicht der Assemblerprogrammierung kaum noch schneller zwischen Prozessen zu wechseln sein.

Selbst wenn ein komplexer Scheduling-Algorithmus diesen Turn-Around-Zyklus um den Faktor 5-10 verschlechtert, liegt die erreichte Umschaltfrequenz immer noch um den Faktor 10 bis 500 über dem komplexer Bausteinsysteme wie ED oder Arena. Lediglich SLX ist etwa gleichwertig bezüglich der Größenordnung.

Bei allen derartigen Performancemessungen ist der gravierende Anteil der Rechenzeit für die Darstellung von Ergebnissen auf den grafischen Bedienoberflächen zu beachten. Wie Messungen und Berechnungen zeigen, kann dieser Rechenzeitanteil bis zu 98% betragen. Somit wird in nur 2% der Zeit wirklich simuliert. Es daher sehr bedauerlich, daß sich bei nur wenigen Simulationssystemen die grafische Anzeige komplett abschalten läßt, wodurch Zeitmessungen kritisch sind und größere Optimierungsläufe behindert werden.

4. Zusammenfassung und Ausblick

Das diskutierte und realisierte Konzept für ein extrem schnelles Multitasking und Scheduling von Simulationsprozessen erlaubt neue Optionen bei der zukünftigen Umsetzung von Simulationsexperimenten :

- Der Programmcode ist mit nur einigen Hundert Zeilen überschaubar groß und lässt sich prinzipiell auf jedes nativ kompilierende Entwicklungssystem portieren. Aktuell sind Delphi und C/C++ als Prototypen getestet. Tests mit jedem zu verwendenden Compiler sind durchzuführen, da die Stackorganisation abweichen in Details kann. Die Routinen sind auch funktionsfähig unter Debuggingconditionen, d.h. es müssen keine Abstriche an Testoptionen gemacht werden.
- Nach einer Integration als Library in die jeweilige Programmiersprache kann die Standardsyntax der Sprache zur Modellierung verwendet werden. Auch eine komfortable GUI-Oberfläche und Datenbankfunktionen sind damit gegeben. Es sind keine speziellen und meist extrem langsamen Skriptsprachen notwendig. Als Ergebnis der Kompilierung entsteht ein ausführbares Programm, welches Lizenzkostenfrei ist und damit auf beliebig vielen Rechnern im Rahmen einer verteilten Simulation eingesetzt werden kann.
- Das Konzept erreicht eine maximale Performance durch ein Ausweichen auf die Verwendung von deutlich mehr Speicher pro Prozeß - „Performance wird bezahlt mit Speicher“. Selbst bei 1 Million parallel ablaufender Prozesse reichen jedoch etwa 256 MByte RAM. Bei größeren Modellen sind entsprechend 100 Euro für 1 Gigabyte RAM zu investieren.

Das vorgestellte Konzept bildet den Kernel für die im Oktober 2004 auf dem Europäischen Simulationssymposium in Budapest gegründete länderübergreifende Workgroup zur Entwicklung eines „EUROPEAN SIMULATION ENVIRONMENT“. Erste Ergebnisse sind abrufbar unter www.simsolution.net .

Literatur

[1] Kuljis, Jasna and Ray J. Paul,: A Review of web based simulation: whiter we wander?, *Proceedings of the 2000 Winter Simulation Conference*, Orlando Florida, page 1872-1881

[2] Kilgore, R. A.: Open source simulation modeling language (SML). In Proceedings of the 2001 Winter Simulation Conference, 2001

[3] Wiedemann. T.: VisualSLX – an open user shell for high-performance modeling and simulation, *Proceedings of the 2000 Winter Simulation Conference*, Orlando Florida, 1865-1871