1 Introduction

This is a specification of a simple scheduler and assembler. The system contains a set of registers and a block of memory. Processes can be created, with each containing a sequence on instructions that are executed on the system. The instruction format is a simplified format of the Intel x86 architecture. Processes are scheduled based on the credit system that is found in the Linux 2.0 kernel.

2 Stack

This specification was written as a test spec for the CZT project. As a result, there are parts that may appear to be specified in a strange way - this is to test out the tools on a large set of Z.

section *Stack* **parents** *standard_toolkit*

A generic stack.

$$Stack[X] == [stack : \operatorname{seq} X]$$
$$InitStack[X] == [Stack[X] \mid stack = \varnothing]$$

$$\begin{array}{|l} \Delta Stack[X] \\ x? : X \\ \hline stack' = stack \frown \langle x? \rangle \end{array}$$

$$\begin{array}{|l} \Delta Stack[X] \\ x! : X \\ \hline stack' \frown \langle x! \rangle = stack \end{array}$$

3 Definitions

section *Definitions* **parents** *standard_toolkit*

Firstly, we define some basic types and functions that are used throughout the specification.

Declarations	This section	Globally
Unboxed items	3	3
Axiomatic definitions	0	0
Generic axiomatic defs.	0	0
Schemas	0	0
Generic schemas	2	2
Total	**5**	**5**

Table 1: Summary of Z declarations for Section 2.

singleton is the set of all sets whose size is less than or equal to 1. This is included only to have a generic axiom definition.

$$\mathbf{relation}(singleton\ \_)$$

$$[X]$$

$$singleton\ \_ : \mathbb{P}(\mathbb{P}\ X)$$

$$\forall\ s : \mathbb{P}\ X \bullet singleton\ s \Leftrightarrow \#\ s \leq 1$$

The basic type of this system is a word, which specifically, is an unsigned octet. An unsigned word is used so references to memory etc a 1-relative.

$$WORD == 0\,..\,255$$

Then, we define the size of the memory block, and give it a value for animation purposes.

$$mem\_size : WORD$$

$$mem\_size = 100$$

A *LABEL* is used to label instructions for *jump* instructions etc, although 'jump' hasn't been specified yet.

$$[LABEL]$$

Now we define the different instructions, as well as their operands. A *CONSTANT* is used both as a constant value, as well as a memory reference for load and store instructions.

$$INST\_NAME ::= add \mid sub \mid divide \mid mult \mid push \mid pop \mid load \mid store \mid loadConst \mid print$$
$$OPERAND ::= AX \mid BX \mid CX \mid DX \mid constant\langle\langle WORD\rangle\rangle$$
$$REGISTER == \{AX, BX, CX, DX\}$$
$$CONSTANT == OPERAND \setminus REGISTER$$

An instruction is specified as a instruction name, a sequence of operands, and optionally, a label.

Instruction

$$label : \mathbb{P}\, LABEL$$
$$name : INST\_NAME$$
$$params : \text{seq}\, OPERAND$$

$$singleton\ label$$

Declarations	**This section**	**Globally**
Unboxed items	9	12
Axiomatic definitions	1	1
Generic axiomatic defs.	1	1
Schemas	1	1
Generic schemas	0	2
Total	**12**	**17**

Table 2: Summary of Z declarations for Section 3.

4 System

section *System* **parents** *Definitions*, *Stack*

The system consists of a set of registers, and a block of memory. There is also a buffer for displaying output.

$$REGISTERS == REGISTER \rightarrow OPERAND$$
$$MEMORY == 1 \,..\, mem\_size \nrightarrow WORD$$

System

$$
\begin{array}{l}
registers : REGISTERS \\
memory : MEMORY \\
output : \mathrm{seq}\ WORD
\end{array}
$$

Initially, all registers and memory hold the minimum *WORD* value. The output buffer is empty.

InitSystem

$$
\begin{array}{l}
System \\
\hline
registers = \{r : REGISTER \bullet r \mapsto constant(min(WORD))\} \\
memory = \{m : 1 \mathinner{..} mem\_size \bullet m \mapsto min(WORD)\} \\
output = \langle\rangle
\end{array}
$$

The system can have arithmetic and memory instructions.

$$
\begin{array}{l}
Arith\_Inst == [Instruction \mid \#params = 2 \land params(1) \in REGISTER] \\
Add\_Inst == [Arith\_Inst \mid name = add] \\
Sub\_Inst == [Arith\_Inst \mid name = sub] \\
Mult\_Inst == [Arith\_Inst \mid name = mult] \\
Div\_Inst == [Arith\_Inst \mid name = divide]
\end{array}
$$

$$
\begin{array}{l}
Memory\_Inst == [Instruction \mid \#params = 2 \land params(1) \in REGISTER \\
\quad \land params(2) \in CONSTANT] \\
Load\_Inst == [Memory\_Inst \mid name = load] \\
LoadConst\_Inst == [Memory\_Inst \mid name = loadConst] \\
Store\_Inst == [Memory\_Inst \mid name = store]
\end{array}
$$

A print instruction prints the value of a register.

$$Print\_Inst == [Instruction \mid \#params = 1]$$

val maps constants to their value, and *dereference* dereferences the value of a register, transitively if required.

$$
\begin{array}{l}
val : CONSTANT \to WORD \\
dereference : OPERAND \times REGISTERS \to WORD \\
\hline
\forall c : CONSTANT \bullet \\
\quad (\exists n : WORD \bullet c = constant(n) \land val(c) = n) \\
\forall a : OPERAND;\ r : REGISTERS \bullet \\
\quad dereference(a, r) = \\
\qquad \textbf{if}\ a \in REGISTER\ \textbf{then}\ dereference(r(a), r)\ \textbf{else}\ val(a)
\end{array}
$$

The specification of the arithmetic instructions.

Add

$\Delta System$
Add_Inst

$\exists\, o_1 == dereference(params(1), registers);$
$\quad o_2 == dereference(params(2), registers) \bullet$
$\quad registers' = registers \oplus \{params(1) \mapsto constant(o_1 + o_2)\}$
$memory' = memory$
$output' = output$

Sub

$\Delta System$
Sub_Inst

$\exists\, o_1 == dereference(params(1), registers);$
$\quad o_2 == dereference(params(2), registers) \bullet$
$\quad registers' = registers \oplus \{params(1) \mapsto constant(o_1 - o_2)\}$
$memory' = memory$
$output' = output$

Mult

$\Delta System$
$Mult\_Inst$

$\exists\, o_1 == dereference(params(1), registers);$
$\quad o_2 == dereference(params(2), registers) \bullet$
$\quad registers' = registers \oplus \{params(1) \mapsto constant(o_1 * o_2)\}$
$memory' = memory$
$output' = output$

Div

$\Delta System$
Div_Inst

$\exists\, o_1 == dereference(params(1), registers);$
$\quad o_2 == dereference(params(2), registers) \bullet$
$\quad registers' = registers \oplus \{params(1) \mapsto constant(o_1 \text{ div } o_2)\}$
$memory' = memory$
$output' = output$

The `load` operation loads a constant from memory. The second parameter is an index to the memory location from which the constant is loaded.

Load

$$\Delta System$$
$$Load\_Inst$$

$$\exists\, o_2 == val(params(2)) \bullet$$
$$\quad registers' = registers \oplus$$
$$\quad\quad \{params(1) \mapsto constant(memory(o_2))\}$$
$$memory' = memory$$
$$output' = output$$

`loadConst` loads a constant into a register. The second parameter the constant to be loaded.

Load_Const

$$\Delta System$$
$$LoadConst\_Inst$$

$$\exists\, o_2 == val(params(2)) \bullet$$
$$\quad registers' = registers \oplus \{params(1) \mapsto constant(o_2)\}$$
$$memory' = memory$$
$$output' = output$$

Store the value of a register in memory.

Store

$$\Delta System$$
$$Store\_Inst$$

$$\exists\, o_1 == dereference(params(1), registers);$$
$$\quad o_2 == val(params(2)) \bullet$$
$$\quad\quad memory' = memory \oplus \{o_2 \mapsto o_1\}$$
$$registers' = registers$$
$$output' = output$$

Print

$$\Xi System$$
$$Print\_Inst$$

$$output' = output \frown \langle dereference(params(1), registers)\rangle$$
$$registers' = registers$$
$$memory' = memory$$

$$Stack\_Inst == [Instruction \mid \# params = 1]$$
$$Push\_Inst == [Stack\_Inst \mid name = push]$$
$$Pop\_Inst == [Stack\_Inst \mid name = pop]$$

The specification of the stack instructions on the system.

*Push*0

$$\Xi System$$
$$PushStack[WORD]$$
$$Push\_Inst$$

$$x? = dereference(params(1), registers)$$

*Pop*0

$$\Delta System$$
$$PopStack[WORD]$$
$$Pop\_Inst$$

$$registers' = registers \oplus \{params(1) \mapsto constant(x!)\}$$
$$memory' = memory$$
$$output' = output$$

$$Push == Push0 \upharpoonright [System;\ Stack[WORD]]$$
$$Pop == Pop0 \upharpoonright [System;\ Stack[WORD]]$$

This executes an instruction on the on the system. *inst*? is the instruction to execute, and *base*? is the base memory value of the executing process. If the instruction is a `load` or `store` instruction, the memory reference must offset using the base value.

$$
\begin{schema}{exec\_inst}
\Delta System \\
inst? : Instruction \\
base? : 1 \mathinner{..} mem\_size
\where
\exists\, label : \mathbb{P}\, LABEL;\ name : INST\_NAME;\ params : \mathrm{seq}\, OPERAND \mid \\
\quad label = inst?.label \land name = inst?.name \land \\
\quad params = inst?.params \bullet \\
\qquad Add \lor Sub \lor Mult \lor Div \lor \\
\qquad Print \lor Load\_Const \lor \\
\qquad name \in \{load, store\} \Rightarrow (\exists\, p : \mathrm{seq}\, OPERAND \mid \\
\qquad\quad p = \langle params(1), \\
\qquad\qquad constant(val(params(2)) + base?)\rangle \bullet \\
\qquad\quad Load[p/params] \lor Store[p/params])
\end{schema}
$$

Declarations	This section	Globally
Unboxed items	19	31
Axiomatic definitions	1	2
Generic axiomatic defs.	0	1
Schemas	13	14
Generic schemas	0	2
Total	**33**	**50**

Table 3: Summary of Z declarations for Section 4.

5 Scheduler

section *Scheduler* **parents** *System*

This part of the specification is the scheduler.

Here, we declare the set of process IDs, the priority values, and the default number of credits a process receives when it is created.

$$Pid == \mathbb{N}$$
$$Priority == -19 \mathinner{..} 19$$
$$Default\_Credits == 10$$

The possible status that a process can hold.

$$Status ::= pWaiting \mid pReady \mid pRunning$$

A process consists of a process ID, a status, a number of credits, and a priority. Each process has a sequence of instructions to be executed on the assembler, with a pointer to the current instruction. The memory that a process can occupy is between a base and limit value. Instructions must only access memory with a value less than the limit, but they know nothing about the base value - this is added onto the memory index provided by the instruction when an instruction is executed. Each procss also contains a stack and values for all registers, which are used to store values when the process is suspended.

Processes

$$
\begin{array}{l}
pids : \mathbb{P}\, Pid \\
status : Pid \pfun Status \\
credits : Pid \pfun \mathbb{N} \\
priority : Pid \pfun Priority \\
instructions : Pid \pfun (\mathrm{seq}\, Instruction) \\
inst\_pointer : Pid \pfun \mathbb{N}_1 \\
base, limit : Pid \pfun WORD \\
pregisters : Pid \pfun REGISTERS \\
pstack : Pid \pfun Stack[WORD] \\
\hline
pids = \mathrm{dom}(status) = \mathrm{dom}(credits) = \mathrm{dom}(priority) = \\
\quad \mathrm{dom}(instructions) = \mathrm{dom}(inst\_pointer) = \mathrm{dom}(base) = \\
\quad \mathrm{dom}(limit) = \mathrm{dom}(pstack) \\
\forall\, pid : pids \bullet inst\_pointer(pid) \leq \#(instructions(pid)) \\
\forall\, pid : pids \bullet base(pid) + limit(pid) \leq mem\_size
\end{array}
$$

The *sort* function takes the credits and priorities of all processes, and returns a sequence of process IDS sorted firstly by their credits (the more credits a process has, the higher preference they get), and if the credits are equal, then their priority. If the priority is equal, then the order is non-deterministic.

$$
\begin{array}{l}
sort : (Pid \pfun \mathbb{N}) \times (Pid \pfun Priority) \pfun \mathrm{iseq}\, Pid \\
\hline
sort = (\lambda\, credits : (Pid \pfun \mathbb{N});\ priority : (Pid \pfun Priority) \mid \\
\qquad \mathrm{dom}(credits) = \mathrm{dom}(priority) \bullet \\
\qquad (\mu\, s : \mathrm{iseq}\, Pid \mid \mathrm{ran}(s) = \mathrm{dom}(credits) \land \\
\qquad\quad (\forall\, i : 1 \,..\, \#\, s - 1 \bullet \\
\qquad\qquad credits(s(i)) > credits(s(i+1)) \lor \\
\qquad\qquad (credits(s(i)) = credits(s(i+1)) \land \\
\qquad\qquad\ priority(s(i)) > priority(s(i)))) \bullet s))
\end{array}
$$

To interrupt a process during execution, the kernel must be in *kernel* mode.

$$Mode ::= user \mid kernel$$

For the scheduler, we track which mode the operating system is in, as well as declaring three "secondary" variables, *waiting*, *running*, and *ready*, to keep the sets of waiting running, and ready variables respecitvely. In fact, *ready* is a sequence, and is ordered based on the credits that each process has. A process with more credits will have a higher priority. This is fair scheduling, because at each timer interrupt (the *tick* operation specified below), the current process losses one credit, therefore, process spending a lot of time executing will eventually have a low priority.

Scheduler

$$
\begin{array}{l}
Processes \\
System \\
Stack[WORD] \\
mode : Mode \\
waiting, running : \mathbb{P}\, Pid \\
ready : \mathrm{iseq}\, Pid \\
\hline
\# running \leq 1 \\
waiting \cap running \cap \mathrm{ran} ready = \varnothing \\
waiting \cup running \cup \mathrm{ran} ready = pids \\
waiting = \{p : pids \mid (status^{\sim})(pWaiting) = p\} \\
running = \{p : pids \mid (status^{\sim})(pRunning) = p\} \\
ready = sort((waiting \cup running) \lhd credits, \\
\quad (waiting \cup running) \lhd priority) \\
\forall r : \mathrm{ran}(ready) \bullet status(r) = pReady \\
\forall r : running \bullet credits(r) > 0
\end{array}
$$

This uses semicolons as conjunctions for predicates, which conforms to the grammar in the ISO standard, but according to the list of differences between ZRM and ISO Z on Ian Toyn's website, semicolons can no longer be used to conjoin predicates.

InitScheduler

$$
\begin{array}{l}
Scheduler \\
InitStack[WORD] \\
InitSystem \\
\hline
pids = \varnothing \,;\; status = \varnothing \,;\; priority = \varnothing \\
credits = \varnothing \,;\; instructions = \varnothing \,;\; inst\_pointer = \varnothing \\
waiting = \varnothing \,;\; running = \varnothing \,;\; ready = \langle\rangle \\
base = \{\} \,;\; limit = \{\} \,;\; pregisters = \{\} \\
mode = user
\end{array}
$$

newProcess creates a new process with a unique process ID and a specified priority, and places this new process on the ready queue.

create_new_process

$$
\begin{array}{l}
\Delta Scheduler \\
\Xi System \\
priority? : Priority \\
instructions? : \mathrm{seq}\, Instruction \\
base?, limit? : WORD \\
pid! : Pid \\
\hline
pid! \notin pids \\
status' = status \cup \{pid! \mapsto pReady\} \\
credits' = credits \cup \{pid! \mapsto Default\_Credits\} \\
priority' = priority \cup \{pid! \mapsto priority?\} \\
instructions' = instructions \cup \{pid! \mapsto instructions?\} \\
inst\_pointer' = inst\_pointer \cup \{pid! \mapsto 1\} \\
base' = base \cup \{pid! \mapsto base?\} \\
limit' = limit \cup \{pid! \mapsto limit?\} \\
pregisters' = \\
\quad pregisters \cup \{pid! \mapsto \{r : REGISTER \bullet \\
\quad\quad r \mapsto constant(min(WORD))\}\} \\
pstack' = pstack \cup \{pid! \mapsto (\langle\!| stack == \langle\rangle |\!\rangle)\} \\
pids' = pids \cup \{pid!\}
\end{array}
$$

We define a schema that contains only the variables that do not change when a reschedule occurs.

$$
\begin{array}{l}
RescheduleChange == \\
\quad Scheduler \setminus (status, running, ready, waiting, credits)
\end{array}
$$

A reschedule occurs when all ready processes have no credits. Every process, not just the ready processes, have their credits re-calculated using the formula $credits = credits/2 + priority$. This implies that the ready process with the highest priority will be the next process executed.

reschedule

$$
\begin{array}{l}
\Delta Scheduler \\
\Xi RescheduleChange \\
\hline
ready \neq \varnothing \\
\forall r : \mathrm{ran}(ready) \bullet credits(r) = 0 \Rightarrow \\
\quad credits' = \{p : pids \bullet p \mapsto (credits(p) \operatorname{div} 2) + priority(p)\} \land \\
\quad status' = status \\
\neg\, (\forall r : \mathrm{ran}(ready) \bullet credits(r) = 0) \Rightarrow \\
\quad status' = status \oplus \{head(ready) \mapsto pRunning\} \land \\
\quad credits' = credits
\end{array}
$$

We declare a new schema that contains only the state variables that do not change when a status change occurs.

$$StatusChange == Scheduler \setminus (status, running, waiting, ready, registers, pregisters, pstack)$$

Interrupts the currently executing process if the new process is of a higher priority then the current process and the kernel is in *kernel* mode.

interrupt

$$\begin{array}{l}
\Delta Scheduler \\
\Xi StatusChange \\
create_new_process \\
\hline
mode = kernel \\
running = \varnothing \lor (\exists\, p : running \bullet priority? \geq priority(p)) \\
\exists\, r : running \bullet \\
\quad status' = status \oplus \{pid! \mapsto pRunning, r \mapsto pReady\} \land \\
\quad pregisters' = pregisters \oplus \{r \mapsto registers\} \land \\
\quad \theta\, Stack' = pstack(r) \\
registers' = pregisters(pid!)
\end{array}$$

Remove the currently running process and put it back in the ready queue.

remove_running_process

$$\begin{array}{l}
\Delta Scheduler \\
\Xi StatusChange \\
\hline
\exists\, pid == (\mu\, r : running) \bullet \\
\quad status' = status \oplus \{pid \mapsto pReady\} \land \\
\quad pregisters' = pregisters \oplus \{pid \mapsto registers\} \land \\
\quad pstack' = pstack \oplus \{pid \mapsto \theta\, Stack'\}
\end{array}$$

A process becomes blocked if it is waiting on a resource such a an IO device, or waiting on another process

$$block\_process == remove\_running\_process \fatsemi reschedule$$

We declare a schema containing only the variables that change for an unblock.

$$UnblockProcessChange == Scheduler \setminus (status, running, ready, waiting)$$

Unblocks a process that is blocked by another process.

$$
\begin{array}{|l}
\textit{unblock\_process} \\
\Delta Scheduler \\
\Xi UnblockProcessChange \\
pid? : Pid \\
\hline
pid? \in pids \\
status(pid?) = pWaiting \\
running = \varnothing \Leftrightarrow status' = status \oplus \{pid? \mapsto pRunning\} \\
running \neq \varnothing \Leftrightarrow status' = status \oplus \{pid? \mapsto pReady\} \\
\hline
\end{array}
$$

Remove a process from the system

$$
\begin{array}{|l}
\textit{remove\_process} \\
\Delta Scheduler \\
\Xi Stack[WORD] \\
\Xi System \\
pid? : Pid \\
\hline
pid? \in pids \\
pids' = pids \setminus \{pid?\} \\
status' = \{pid?\} \lhd status \\
credits' = \{pid?\} \lhd credits \\
priority' = \{pid?\} \lhd priority \\
instructions' = \{pid?\} \lhd instructions \\
inst\_pointer' = \{pid?\} \lhd inst\_pointer \\
base' = \{pid?\} \lhd base \\
limit' = \{pid?\} \lhd limit \\
pregisters' = \{pid?\} \lhd pregisters \\
pstack' = \{pid?\} \lhd pstack \\
\hline
\end{array}
$$

Update the details in the process table when each instruction is executed, as well as communicate the current instruction and the base value for the current process.

$$ChangeInstPointer == Scheduler \setminus (inst\_pointer)$$

update_process_table

$$
\begin{array}{l}
\Delta Scheduler \\
inst! : Instruction \\
base! : WORD \\
\hline
running \neq \varnothing \\
(\exists\, pid == (\mu\, r : running) \bullet \\
\quad inst! = head(instructions(pid)) \land \\
\quad base! = base(pid) \land \\
\quad (inst\_pointer(pid) = \#(instructions(pid)) \Rightarrow \\
\qquad remove\_process[pid/pid?]) \land \\
\quad inst\_pointer(pid) < \#(instructions(pid)) \Rightarrow \\
\qquad inst\_pointer' = \\
\qquad\quad inst\_pointer \oplus \{pid \mapsto inst\_pointer(pid) + 1\}) \\
\theta\, ChangeInstPointer = \theta\, ChangeInstPointer'
\end{array}
$$

$$
next == exec\_inst \gg update\_process\_table \fatsemi \\
([\Delta Scheduler \mid running = \varnothing] \land reschedule) \lor \\
([\Xi Scheduler \mid running \neq \varnothing])
$$

$$
idle0 == \neg\ \mathbf{pre}\ next
$$

idle

$$
\begin{array}{l}
\Xi Scheduler \\
inst? : Instruction \\
base? : WORD \\
\hline
idle0
\end{array}
$$

$$
tick == next \lor idle
$$

$$
\vdash ?(\forall\, n : \mathbb{N}_1 \bullet n > 0)
$$

$$
[X] \vdash ? \forall\, x : \mathbb{P}\, X \bullet \# x \leq 1 \Leftrightarrow singleton\, x
$$

theorem PreconditionCheck

$$
\forall\, Scheduler \bullet \mathbf{pre}\ update\_process\_table
$$

Declarations	**This section**	**Globally**
Unboxed items	21	52
Axiomatic definitions	1	3
Generic axiomatic defs.	0	1
Schemas	11	25
Generic schemas	0	2
Total	**33**	**83**

Table 4: Summary of Z declarations for Section 5.