1. Multi-Processing in Quintus Prolog

The primary goal in our development of a multi-processing environment for Quintus Prolog is to concurrently run several light-weight Prolog processes on standard hardware (e.g. Sun workstations). These processes will share a common internal Prolog database and will communicate with each other via Prolog primitives.

A secondary goal is that this environment is portable. It should be designed to run on a variety of different machine types while retaining the same user interface and the same semantics.

The current design is based upon a Unix system with 1 processor and 1 memory unit. The memory management design for a system with N processors and 1 memory unit translates directly, with no further work required. A system with N processors and N local memories will require reorganization of both the memory management and process communications design.

1.1. Current Status

Currently, all stack group manager routines has been implemented and tested. This task required 2 man-months work. It remains to identify all process-specific data embedded in C and implement the context switching routines. This task is estimated to require 3 man-months work.

1.2. Manipulating Prolog Processes

The following are the user-level primitives which are available to manipulate lightweight Prolog processes.

1.2.1. Starting a process: process_start(+Goal, -Handle)

Goal will be executed in a new, separate Prolog process.

Upon successful creation of the process, Handle will be unified with a unique term, of the form '$process'(C,T), which will be used to identify this new process. All subsequent manipulation of this process will require Handle as an identifier. Once the process is created, process_start/2 immediately succeeds.

When Goal terminates, no indication is given to the calling process of whether Goal has succeeded or failed. This protocol can be implemented using the rendezvous/2 primitive between the parent and the child. For example,

```
process_start(Goal, Handle) :-
        gensym(Label),
        process_start(
            ( call(Goal) , rendezvous(Label, 1)
                         | rendezvous(Label, 0) ), Handle),
        rendezvous(Label, 1).
```

Any free variables in Goal will not be instantiated by running Goal as a separate process.

1.2.2. Terminating a process: process_terminate(+Handle)

The process corresponding to Handle will be terminated and all its space will be reclaimed. If Handle does not correspond to a current process then an error indication is given.

1.2.3. Suspending a process: process_stop(+Handle)

The process corresponding to Handle will be suspended and remain so until either the process it restarted (using process_continue/1) or terminated (using process_terminate/1). If Handle does not correspond to a current process then an error indication is given. If the process is already suspended, this predicate will succeed.

1.2.4. Restarting a stopped process: process_continue(+Handle)

The process corresponding to Handle will be restarted and remain so until either the process is stopped (using process_stop/1), terminated (using process_terminate/1) or waiting for a message (using rendezvous/2 or receive/2). If Handle does not correspond to a current process then an error indication is given. If the process is already running, this predicate will succeed.

Note that initially no parent/sibling specific operations are defined, but they can be programmed using the given communication primitives.

1.3. Communicating Between Prolog Processes

The following are the predicates used to communicate between Prolog processes:

1.3.1. Communication Via Blocking Write: rendezvous(+Label, ?Message)

If another process has already issued a rendezvous/2 with using Label, then the Messages of the rendezvous/2 pair are unified. If a rendezvous/2 has not been issued for Label then this process blocks until one does.

1.3.2. Communication Via Non-Blocking Write: send(+Label, +Message) and receive(+Label, ?Message)

send/2 adds Message to the message queue for Label. It never blocks. receive/2 unifies the front message of the queue for Label with Message and blocks if the message queue for Label is empty.

2. Basic Implementation Issues

To support multiple light-weight Prolog processes we must retain multiple copies of the Prolog execution state while retaining a single copy of the internal Prolog database, where the term, database, is extended to mean not only Prolog clauses but also the C data structures which contain Prolog state.

Since this Prolog database is shared amongst different Prolog execution states (each of which can change the database state independently) there are concurrency issues which must be resolved. The current design does not resolve these issues.

Each of these processes needs to record its own execution state. A process's execution state is represented as a complete set of Warren Abstract Machine (WAM) stacks and registers plus a small set of state variables which exists both as Prolog facts and C data structures.

2.1. Memory Management in a Multi-Processing Prolog

The current memory organization in (uni-processing) Quintus Prolog is as follows:

- C code space
- C data space
- Prolog code space
- Prolog data space
- Heap Stack
- Pdl Stack
- Trail Stack
- Local Stack
- Garbage Collection work space
- C stack

The current memory organization in the design for a multi-processing Quintus Prolog is as follows:

- C code space
- C data space /* global data */
- Prolog code space
- Prolog data space /* global and process-specific data */
- Stack Group space /* contains stack groups and free blocks */
- C stack

where each stack group consists of the following areas:

- C data space /* process-specific data */
- Heap Stack
- Pdl Stack
- Trail Stack
- Local Stack
- Garbage Collection work space

The following is a list of assumptions from which the current stack group memory manager was designed:

- A single stack group must be in contiguous memory
- Stack groups are of varying sizes (even starting at creation)
- The order of stack groups in memory is irrelevant (it may not even be chronological)
- No stack group can have pointers into another stack group
- A stack group may expand and/or contract many times during its existence
- Prolog code space may expand and/or contract many times during the multi-processing session
- Data movement must be kept to a minimum

- Memory fragmentation must be avoided
- For efficiency, no bounds checking will be done for memory references from C which are outside the bounds of a stack group

The same memory management done in the current uni-processing Prolog environment will be done in each stack group. The difference is that when the uni-processing system would otherwise ask for/return memory from/to the operating system, the multi-processing Prolog will issue a command to a stack group manager to perform the stack group expansion/contraction in the context of other stack groups.

The new memory management facility that is provided in the multi-processing system is the stack group manager. It manipulates stack groups as if they were atomic objects; it does not care what the internal structure of a stack group is. It just requires two hooks into the lower level memory manager for the stack group internals:

- shift_stack_group(+stack_group, +bytes)
- initialize_stack_group(+stack_group, +bytes, +goal_handle)

These functions are,

- expand_code(+min_size, +ideal_size)
- contract_code(+ideal_size)
- allocate_stack_group(+min_size, +ideal_size, -stack_group)
- expand_stack_group(+stack_group, +min_size, +ideal_size)
- contract_stack_group(+stack_group, +ideal_size)
- deallocate_stack_group(+stack_group)

2.2. Stack Group Manager Primitives

The following are specifications for the C primitives provided by the stack group manager. All of these routines are currently implemented and debugged on a prototype version of the system. They are not yet integrated with the two internal stack group memory management routines mentioned above (shift_stack_group() and initialize_stack_group()). The debugging of these routines involved handling a set of randomly generated memory management requests.

Stack group space consists of variable-length memory blocks, some allocated to active stack groups and some available for allocation (i.e. free). All of these memory blocks form a doubly-linked list in which each block has a pointer to both the block adjacent to it on the right and on the left. With this scheme it only takes constant time to coalesce adjacent free blocks and this happens as each new free block is created.

2.2.1. expand_code(+min_size, +ideal_size)

Try to expand the code space gap by ideal_size bytes. It is acceptable to expand by more or less than ideal_size bytes to reduce or avoid memory fragmentation or due to constraints on memory availability as long as it is expanded by least min_size bytes. This call may fail if there is not enough memory available. This call may fragment an existing free block.

2.2.2. contract_code(+ideal_size)

Contract the code space gap by ideal_size bytes. The space deallocated will be a new free block which will be used by the stack group manager. It is acceptable to contract by less than ideal_size bytes to reduce or avoid memory fragmentation. This call may create a new free block or expand an old free block.

2.2.3. allocate_stack_group(+min_size, +ideal_size, -stack_group)

Try to allocate a stack group which is ideal_size bytes. It is acceptable to allocate a stack group which is more or less than ideal_size bytes to reduce or avoid memory fragmentation or due to constraints on memory availability as long as it is allocated at least min_size bytes. This call may fail if there is not enough memory available. This call may fragment an existing free block.

2.2.4. expand_stack_group(+stack_group, +min_size, +ideal_size)

Try to expand stack_group by ideal_size bytes. It is acceptable to expand by more or less than ideal_size bytes to reduce or avoid memory fragmentation or due to constraints on memory availability as long as it is expanded by least min_size bytes. This call may fail if there is not enough memory available. This call may fragment an existing free block. This call may result in the movement of multiple stack groups.

2.2.5. contract_stack_group(+stack_group, +ideal_size)

Contract stack_group by ideal_size bytes. The space deallocated will be a new free block which will be used by the stack group manager. It is acceptable to contract by less than ideal_size bytes to reduce or avoid memory fragmentation. This call may create a new free block or expand an old free block.

2.2.6. deallocate_stack_group(+stack_group)

Deallocates the stack group. This call may create a new free block or expand an old free block.

3. Context Switching

Context switching will entail calling a C routine (at which time all of the prolog registers are saved in a global memory block) this C routine copies the global memory block into a stack group memory block and changes the global memory block to another context (a copy of a different stack group memory block or a new memory block, created for a new process) and returning from the C procedure (which will restore the prolog registers, which reflect the new context, from the global memory block).

The current design does not handle timer interrupts, rather a process must explicitly give up control or be aborted from the top level. This initial design models the Xerox 1100 family of workstations.

3.1. Determining Process-Specific Data

A problem arises when trying to categorize memory into global and process-specific. A Quintus Prolog execution state consists of code space, WAM stacks and C execution state (because of Quintus' general foreign language interface). It is easy to partition WAM stacks into stack groups so that one set of stacks can be active while all others can be waiting, however there is no portable and efficient way to have C stack exist in each stack group (the C stack can only exist in one place). Various solutions were reviewed:

- save/restore the entire C stack in the stack group when a context switch occurs
- enforce the restriction that the entire chain of C function calls from Prolog to the point in C code where a context switch will occur must be reentrant. This is so that when a process which has been stopped in C code it can be restarted by reinvoking the entire chain of C function calls without any side effects. Since there are no timer interrupts in the current design, the places in the C code which can cause context switches are well-defined (e.g. I/O requests)

To avoid the high cost of data copying a every context switch the second solution was chosen.

3.2. Process-Specific Data In Prolog Database

There is some process-specific data which is stored as meta-facts (facts about the Prolog execution state) in the Prolog database. Our original intent was to share all the Prolog facts and rules amongst all processes in the system. However, these meta-facts violate this intent. Two solutions,

- provide (only to the Prolog system programmer) a local_assert and local_retract which manipulates Prolog facts and rules in a small code space allocated in each stack group
- provide (only to the Prolog system programmer) a local_assert and local_retract which uses the global Prolog database but associates a process id with the asserted fact and only allows retracting facts which belong the process executing the local_retract

The first solution provides automatic reclamation of locally asserted facts when a process is terminated. However, the second solution requires a garbage collector pass to reclaim all of the clauses which were not explicitly retracted by the user before the process was terminated.

The second solution was decided upon since it did not require re-implementing the code space manager to handle local code spaces.