M.I.T. Laboratory for Computer Science

Request for Comments No. 229
July 20, 1982

Presenting a Simple Multi-Tasked CLU

by Geoffrey Cooper

1. Introduction

This document describes an extension to the CLU language to provide a multitask environment. The aim in this extension was to design a simple multitask environment which will run within a single Unix process on the Vax CLU implementation, and which can be implemented with a minimum of hassle. Specifically, it was required that no modifications be made to the CLU compiler or linker. The environment does not attempt to be the last word in a concurrent CLU, but rather a quickly built and flexible first attempt to be used for experimental purposes. Our intent is to discard this implementation after a very short period of use.

Multitasking centers around a set of CLU clusters which are handed to the CLU linker along with the user's CLU programs. These clusters allow for the manipulations of tasks, timers, and unix signals. Some of the code that implements multitasking is written in assembly language, but most is written in CLU. A (very slightly) modified version of the CLU garbage collector must also be loaded. Since implementing a concurrent or incremental garbage collection scheme is a major undertaking, no attempt has been made to change the substance of the regular garbage collector. When the garbage collector is invoked, all tasks must wait until it is through before any can run.

The *task* cluster is used to create and manipulate tasks. It also contains the task scheduler. Scheduling is non-preemptive; tasks run until they voluntarily yield the processor by calling into the task cluster. The garbage collector, while non-concurrent, will not cause a task pre-emption. When the garbage collector is called during the execution of some task, that same task is resumed after garabage collection. An elementary mechanism, called *events*, allows tasks to send wakeups to other tasks.

The *unix_signal* cluster couples the task *event* mechanism with the unix operating system's *signal* mechanism. Each unix signal may be bound to a task-event pair, such that future occurances of the signal are translated into a setting of an event for the task.

2. Programmer Interface Specifications

A CLU program which wishes to run in a multitask environment must begin by intializing tasking with a call to *task$multitask*. When this procedure returns, its caller is running as the only task in a newly-multitasked environment. Presumably, this task will then create other tasks, using *task$create* before yielding control with *task$wait* or *task$let_anyone_else_run*.

A typical format for the "start_up" procedure (the procedure that is called by the CLU run time system to start things off) of a multitasked program is as follows:

```
start_up = proc()
  % begin multitasking, with self as first task
  task$multitask()
  for each task I want to start with do
    task$create( proc that runs it, some_stack_size )
    end
  task$exeunt() % I'm not needed anymore...
  end start_up
```

The start_up procedure will most likely create other tasks. This does not preclude tasks from being created by other procedures or tasks. Tasks may be created at any time and by any CLU module once tasking has been started using task$multitask.

There is currently no provision for handing arguments to tasks. Since there are no global variables in CLU. The solution to this problem is to use clusters as multi-process modules. First, several tasks may be encapsulated in a cluster, and may communicate one with the other using the cluster's own variables. Other processes may access such own variables using cluster operations as "in-calls."

The specifications of the task cluster are as follows:

```
data type task is multitask,
                  create,
                  wait,
                  let_anyone_else_run,
                  abort,
                  test_events,
                  set_event,
                  me,
                  exeunt
```

Auxiliary Information

The task cluster implements a multi-tasking function in CLU. Calls to several of the cluster's operations will cause the thread of control to shift in ways which are unlike anything in the CLU Reference Manual.

Operations

multitask = **proc**(stack_size: int)

effect Initializes and begins multitasking. This procedure causes the caller to run as the only task in a multitasked system. This task may (and presumably will) create other tasks before it exits. The *stack_size* argument is used as a hint to suggest the stack size that will be needed by this task.[1] If the caller returns before calling *task$abort* or *task$exeunt*, a failure results.

create = **proc**(body: proctype(), stack_size: int) **returns**(task)

effect A new task is created and enqueued for running. The task will run the procedure *body* until a call is made to task$exeunt. The *stack_size* argument is used as a hint to suggest the stack size that will be needed by this task. A failure occurs if *body* returns without calling *task$exeunt* or *task$abort*.

wait = **proc**(events: seq[string]) **returns**(string) **signals**(abort)

effect If one of the events named in *events* is already set when wait is called, the procedure returns immediately; otherwise it causes the current task to block until any of the named events is set for it. The procedure then non-deterministicly chooses any set event named in *events*, clears it, and returns its name. If *events* is empty, the effect is to wait for ***any*** event. If *task$abort* is called before this procedure would normally return, abort is signalled. This procedure has the effect of *task$let_anyone_else_run* when *task$abort* was called before the call to *task$wait*.

[1]The implementation on Unix takes this as more than a hint - be careful!

let_anyone_else_run = **proc**()

effect Causes any tasks that are ready to run to run, then resumes the calling task. This procedure is not intended for general use, but it does allow the normal function of the task scheduler to be subverted by any particular task.

abort = **proc**()

effect *This procedure never returns.* It causes every subsequent call to *task$wait* to signal abort. Its use is to allow all tasks to cleanup and terminate so that the multitasking may end.

set_event = **proc**(ts: task, event: string)

effect Sets the event *event* in the task *ts*.

test_events = **proc**(events: seq[string]) **returns**(string) **signals**(no_events_set)

effect If no events are set in the currently running task, signals no_events_set; otherwise, chooses any set event from *events*, clears it, and returns its name. If *events* is empty, the effect is to return the name of ***any*** set event.

This procedure is intended for use where a task has useful tasks to perform, but wishes to give priority to any events pending before performing them. It is also useful for debugging purposes to determine if any uncaught events are set. Since it does not check for a prior call to *task$abort*, it provides a means for a task to check signals after *task$abort* has been called

me = **proc**() **returns**(task)

effect Returns the task object associated with the currently running task.

exeunt = **proc**()

effect *This procedure never returns.* Terminates the currently running task. A task which returns before calling *task$abort* or *task$exeunt* causes a failure.

end task

Timer facilities are provided by the *timer* cluster using the event facility of the task package. The timer cluster is as follows:

data type timer **is** set, advance, defer, clear

Operations

set = **proc**(ts: task, event: string, when: int)
effect Causes the event *event* to be set in task *ts* in *when* milliseconds.[2] This procedure always causes a new timer to be set. Each timer will cause the event to be set in the task *ts*, regardless of whether the event has been previously set or not.

advance = **proc**(ts: task, event: string, when: int)
effect If no timer for event *event* is set for task *ts*, this procedure works exactly as timer$set. When a timer is set for the event for *ts*, the timer is reset to go off in *when* milliseconds *if this time is sooner than the time already set.*

defer = **proc**(ts: task, event: string, when: int)
effect This procedure works exactly as timer$advance, except that it will only reset a timer to a *later* time than is already set.

clear = **proc**(ts: task, event: string) **signals**(not_set)
effect If no timer for event *event* is set for task *ts*, this procedure signals not_set. Otherwise, all such timers are cleared.

end timer

An interface to Unix signals is provided, so that tasks may respond to external events. The approach taken is to allow a task to associate an event with a particular Unix signal. When a signal goes off, it is translated to the specified event in the specified task. Note that this mechanism does not allow a task to tell if a signal has happened a number of times. Hopefully tasks will switch frequently enough so that this will not be a problem.

Unix signals are bound to task-event pairs using the unix_signal cluster:

data type unix_signal **is** bind, unbind

Auxiliary Information
This cluster provides an interface between the task-event mechanism and unix signals

Operations

[2]Note: The resolution of this timer is *one second*!! Thank you, Unix.

bind = **proc**(ts: task, event: string, ux_sig: int) **signals**(no_such_signal, already_bound)
effect Causes the event *event* to be set for task *ts* whenever the unix signal numbered ux_sig is signalled. Signals no_such_signal if there is no unix signal numbered ux_sig; signals already_bound if the signal is already bound to some event-task pair.

unbind = **proc**(ts: task, event: string, ux_sig: int) **signals**(not_bound, no_such_signal)
effect Causes the binding that was created by a previous call to unix_signal$bind with the same arguments to be undone. Signals: not_bound if the binding does not exist; no_such_signal if there is no unix signal numbered ux_sig.

3. Using Multitasking

The appropriate binary files are available on MIT-AJAX in the directory /user/lwa/geof/task. Specifications of the task, timer, and unix_signal clusters may be loaded into the CLU compiler by merging the file "/usr/lwa/geof/task/task.lib". Programs should be linked using the following syntax: when the CLU debugger is used:

```
    % link <listOfMyFiles>,@/usr/lwa/geof/task/dtask
```

when a production program is being created:

```
    % link <listOfMyFiles>,@/usr/lwa/geof/task/task
```

4. Caveats

This section mentions a number of implementation restrictions which users of the tasking package should be careful to not abuse. They are listed here for convenience:

1. *Stack Size:* The stack_size arguments to task$multitask and task$create are interpreted not as hints but as firm limits on the size of the stack allocated to the task. Failures occur if there is not enough room left in the stack area to create a task or begin multitasking. No additional checking is performed, although it would not be difficult to modify the task scheduler to check for stack overflows. A stack overflow will in all likelihood crash the run time environment of the CLU system.

   Note that, for production programs, the linker allows the maximum stack size to be set. Tasking programs may need to be linked with a larger stack. The debugger gives a (non-changeable) stack size of 12K bytes. This may not be sufficient for all multi-tasking programs.

2. *Dynamic Task Creation:* Stacks of terminated tasks are garbage collected,[3] but existing stacks may not be compacted for technical reasons. This means that fragmentation of

[3]This is not to say that they are in the heap. Stack space is garbaged collected by specialized routines which are invoked at task creation and exeunt time.

stack space will tend to occur is some programs. Causing stack sizes to be integer multiples of some number will alleviate this somewhat but not entirely (since only adjacent stacks may be merged).

3. *Wierd Restriction:* The procedure task$multitask must ***never*** be called when there is an iterator on the stack (even in some calling routine). It is probably a good idea to call it as the first thing in a CLU program.

4. *Inaccurate Timers:* Due to deficiencies of the Unix operating system, the resolution of the timers provided by the timer cluster is one second. A special version of the timer code has been created which busy waits, calling task$let_anyone_else_run and checking the system time (which is resolved to a much finer grain) rather than setting operating system alarms. Thus it is more accurate but may be less efficient. It may be loaded by using the file /usr/lwa/geof/task/task_bw (and dtask_bw, for debugging) instead of the usual command file.