Appendix E

GDB for μC++

The symbolic debugging tools (e.g., dbx, gdb) do not work well with μC++, because each coroutine and task has its own stack and the debugger does not know about these stacks. When a program terminates with an error, only the stack of the coroutine or task in execution at the time of the error is understood by the debugger. The following gdb and Python macros allow gdb to understand the μC++ runtime environment: tasks (see Section 2.13, p. 29), processors (see Section 8.4, p. 129), and clusters (see Section 8.3, p. 127).

These instructions assume the install <prefix> for μC++ is /usr/local/u++-7.0.0; if installed elsewhere, change <prefix>. Copy <prefix>/gdb/.gdbinit to your home directory or merge it into your existing .gdbinit file. If installed elsewhere, edit the <prefix> within the .gdbinit file to the install location. Thereafter, gdb automatically loads the .gdbinit file from the home directory at start up making the following new gdb commands available.

Debugging involves setting one or more breakpoints in a program. When a breakpoint is encountered, the entire concurrent program stops, i.e., all user and kernel threads. At this point, it is possible to examine the stacks of each μC++ task stack by listing all the tasks (command task), switching to an individual task (command task 2), and printing the back trace (command backtrace) for its stack. The brace trace shows where that task is executing, and by moving up and down the back trace, it is possible to examine the variables in each stack frame.

E.1 Clusters

Command	*Description*	*Example*
info clusters clusters (alias)	print list of clusters	info clus

E.2 Processors

Command	*Description*	*Example*
info vprocessors	print virtual processors in userCluster	info vproc
info vprocessors *clusterName*	print virtual processors in cluster *clusterName*	info vproc clusterA
vprocessors (alias)		

E.3 Tasks

Command	*Description*	*Example*
task	print tasks (ids) in userCluster, application tasks only	task
task *clusterName*	print tasks (ids) in *clusterName*, application tasks only	task clusterA
task all	print all clusters, all tasks	task all
task *id*	switch debugging stack to task *id* on userCluster	task 2
task 0x*address*	switch debugging stack to task 0x*address* on any cluster	task 0x80024875
task *id clusterName*	switch debugging stack to task *id* on cluster *clusterName*	task 2 clusterA
prevtask	switch debugging back to the previous task on the stack	prevtask

Note, in μC++, the name of the program main is changed to uCpp_main; hence, to set a break point at the start of the application main, do the following:

```
(gdb) break uCpp_main
```

During execution, the debugger cannot step through a context switch for either a coroutine or task. Therefore, it is necessary to put break points after the suspend/resume or signal/wait to acquire control, and just continue execution through the context switch. Once the breakpoint is reached, it is possible to next/step through the lines of the coroutine/task until the next context switch.

For debuggers that handle multiple kernel threads (corresponding to μC++ virtual processors), it is also possible to examine the active task running on each kernel thread.

```
(gdb) info threads      # list all kernel threads
(gdb) thread 2          # switch to kernel thread 2
```

Finally, it is necessary to tell the debugger that μC++ is handling UNIX signals SIGALRM and SIGUSR1 to perform pre-emptive scheduling. For gdb, the following debugger commands allows the application program to handle signal SIGALRM and SIGUSR1:

```
(gdb) handle SIGALRM nostop noprint pass
(gdb) handle SIGUSR1 nostop noprint pass
```

This step is handled automatically in the specified .gdbinit file.