Memory Management in Real-time Multiprocessors

Kasi Anantha, David Tangman and Daniel Gold
Department of Mathematical Sciences
San Diego State University San Diego, CA 92182

Abstract

In multiprocessor systems, a common programming error occurs when one of the processors accidently modifies the instructions or data of another processor. These errors can be extremely difficult to detect, but can be eliminated through the use of memory management hardware. The KUMARAN II architecture is a multiprocessor architecture (based on seven MC68030 processors) targeted for real-time applications. The basic philosophy of the KUMARAN II architecture is that the real-time system developer should be primarily concerned with the application and not with the internal hardware. Thus, the architecture has been designed to automatically implement and enforce the user-entered, high-level memory definitions and specifications. This paper describes the specification and implementation of user-interface to KUMARAN II memory management.

1 Introduction

Real-time programmers usually avoid memory management; the overhead of translating virtual addresses to physical addresses, as well as the penalty for accessing secondary storage when servicing a page fault, makes memory management unattractive for many real-time applications. Thus, real-time applications are generally memory-resident, and it is up to the programmer to carefully design and manage memory resources. However, in a multiprocessor system with shared memory space, this technique can induce extremely subtle programming errors, as illegal accesses by an errant processor are often difficult to detect. Memory management in Kumaran II has been designed specifically to solve these problems.

Kumaran II contains seven MC68030 processors and 32 Megabytes of physical memory. Kumaran II enforces a strict memory management discipline which partitions the 32-Megabyte memory into seven, 4-Megabyte local memories (one for each processor) and one, 4-Megabyte shared memory area. Furthermore, Kumaran II prevents processes from writing to local and shared memory pages which are marked as read-only.

1.1 Previous Work

CHIMERA[1] is a real time computing environment developed at Carnegie-Mellon University for the Reconfigurable Modular Manipulator System (RMMS) project. CHIMERA consists of one or more Ironics 68020 CPU boards, acting as real-time computing engines, residing on the VME bus of a Sun 3 workstation. This computing environment stressed the following two aspects:

1. High-level computing, performed on a general purpose processor, for real-time program development and offline storage and analysis of experimental results.

2. High performance numerical computing on dedicated real-time processors to implement the control law, and sample and buffer data for later storage on the general purpose system.

The CONDOR[2] system on the other hand was developed at MIT for controlling the Utah-MIT hand and other complex robots. In the second version of the CONDOR hardware a Sun-3 forms the main development machine while the real time controller utilizes Ironics Corporation 16 MHz Motorola 68020 based single board computers equipped with Motorola 68881 floating point processors. The control processors are linked using a VME bus and are coupled to the development host through a VME bus to VME bus adaptor.

The Kumaran[3,4,5] multiprocessor system used Motorola MVME136A boards based on MC68020 and MC68881 processors, and AT&T System V on a Motorola MVME147 board for program development and data storage and analysis. The real time CPU boards (MVME136A) and the development CPU (MVME147) were kept on the same VME bus chassis. Real time CPUs communicated with the Unix CPU via the VME bus using special device drivers.
The Kumaran II multiprocessor system replaced the Motorola MVME136A boards with faster MVME143-2 boards containing an MC68030 at 33 MHz and MC68882 floating point units. Later the Unix CPU was moved to its own chassis and connected to the real time CPUs using a VME bus to VME bus adaptor.

It was noticed during the software development and experimentation that the multiprocessors occasionally overwrote the instructions and data space of another processor resulting in a program crash. The system was left in a state from which debugging was extremely difficult. The destructive interference of these processors could be alleviated by the use of the virtual memory system provided by the MC68030. In addition, the use of virtual memory further improved the development of application programs by eliminating the need for individual process to know the physical address used by shared variables. Each process can access a shared variable using its own virtual address and the MMU will map the virtual address to the physical address automatically.

1.2 System Design

Kumaran II provides the application designer with a distributed, real-time memory management environment which partitions the VME bus address space into eight sections: an I/O space (low memory), a separate 4MB processor-local memory space for each of the 6 real-time Kumaran II multiprocessors, and a 4MB common memory space. By default, every Kumaran II process has access to the I/O space and its own processor-local RAM, but must use the facilities provided by the memory management executive calls to establish and manipulate access to the common memory partition. This result is achieved through a built-in Kumaran II process construction procedure, a set of callable executive routines, and the internal MMU of each of the MC68030 processors within Kumaran II.

The built-in Kumaran II process construction procedure is implemented as a Korn-shell script called "cmmake" which accepts a user-specified "buildfile" as input, and produces a UNIX "makefile" as output. Example 2(a) describes the general format for a cmmake buildfile. Note that white space characters may be freely used within a buildfile, and that the ".ext" file extension for object files within a "OBJECTS/ENDOBJECTS" block must either be "sa" (MC68030 assembly language sources) or "c" (standard C source files). The program name parameter is the name to assign to the executable image (COFF format) that will be produced by the make utility. For example, suppose an application (to be named exampleprog) consists of 5 source files: top.c, sub1.c, sub2.c, sub3.c, and drivers.sa (MC68030 assembly language source). The application designer would create a buildfile (named "mybuild") similar to that of example 2(b), and invoke the cmmake script through the Korn-shell.

If cmmake detects no errors either in the format of the buildfile or in the sources that are specified, a file named "makefile", acceptable as input to the UNIX make utility, will be created in the current directory path. Example 3 displays the makefile that would be produced from the buildfile for exampleprog. The programmer can further customize the makefile (e.g. add dependencies) if so desired, but unless the program name or objectX.o files which occur in the buildfile are changed, there is no reason to re-run cmmake.

Notice that in example 3, the reference to "$(CM)/header.o" and "$(CM)/cm.o" objects. The header.o object is the first object linked into the executable image, and contains the memory-management initialization code. After header.o has initialized the run-time memory environment for the MC68030 processor, it calls the "start" entry point into the application, as Kumaran II requires the programmer to name the top-level routine of the application "start" instead of "main". The cm.o object contains the common memory, memory management executive calls which Kumaran II provides to its applications.

Kumaran II memory management segments the 4MB common-memory partition into 128, 32KB blocks. The Kumaran II memory management executive calls (provided in the cm.o object) allow a process to dynamically manipulate access and rights to these 32KB blocks of common memory. A brief description of these calls is given below.

\begin{verbatim}
int address; /* VME bus address */
int pagnum; /* Common memory page number 0..127 */
int rights; /* 0=none; 1=read only; 2=read/write */
int size; /* # of bytes requested */

(1) int cmset( pagnum, rights, size )
\end{verbatim}

This function sets the access 'rights' to the common memory page(s) required for a block of 'size' bytes. This is accom-
plished by identifying and modifying the common memory page descriptors according to the requested access rights. If the ‘size’ parameter is greater than 32KB, then the access rights for multiple, contiguous, common memory pages will be set.

RETURNS:
  0: ok
  -1: illegal page number
  -2: illegal rights
  -3: illegal size

(2) char *cmlink( pagnum )

This function will return a pointer to the requested common memory block. It is up to the user to type cast the returned pointer to the appropriate data structure type.

RETURNS:
  -1: illegal page number
  0: no access rights
  > 0: pointer to first byte of the common page.

(3) int cmtest( pagnum )

This function will return the access rights that have been established for a common memory block.

RETURNS:
  -1: illegal page number
  0: no access rights
  1: read-only access
  2: read/write access

Example 1. memory management calls

The header.o object file implements the Kumaran II virtual memory model directly in the MC68030 MMU. The transparent translation (TT0) registers of the MC68030 MMU are initialized to allow for transparent translation of all I/O space addresses, and the address translation tree entirely in processor-local memory space. The entire VME bus address space is partitioned into 32KB pages, and the high-order 6 bits all virtual addresses will be ignored (thus, physical memory is limited to 64MB).

The CPU root pointer (CRP) points to the first-level of the address translation tree. The TIA field of the translation control register is set to 4 bits, and the TIB field is set to 7 bits. Thus, the highest-order 4 bits of a virtual address (bits 22-25) partition the virtual address space into sixteen segments. The 7-bit TIB field further divides each of the 16 4MB segments into 128, 32KB pages. Since the 4MB of local-memory addresses range from 0 to 400000(hex) and the 4MB of common-memory occupy the highest 4MB of the virtual address space, 2800000(hex) to 3200000(hex), all of the descriptors in the first-level translation table are invalid, except for the first and the tenth (which contain pointers to the second-level page descriptor tables for the local-space and common-space). If a Kumaran II process attempts to access an address that is described by one of the invalid descriptors in the first-level table, a bus-error exception is raised and the access is disallowed. The second-level tables for the first and the tenth entries in the first-level table (the local-memory and common-memory segments respectively) each contain 128 page descriptors which directly map to the 128, 32KB pages of physical memory at those locations. Once the internal tree traversal algorithm of the MC68030 MMU locates a page descriptor in the second-level table, the translation is immediately cached in the internal Address Translation Cache (ATC) of the MC68030.

The Kumaran II programmer constructs the application around a distributed processing model that consists of six MC68030 processors (each of which have exclusive access to a local 4MB RAM and shared access to a 4MB RAM) and the set of Kumaran II common memory executive calls. Next, the programmer creates a buildfile which lists each source file in the application, as well as assigns a name for the process, and runs the cmmake Korn-shell script to construct the makefile for the target Kumaran II process. Finally the programmer invokes the UNIX make utility, and if no compilation or linking errors are detected, an executable Kumaran II image (COFF format) is produced. Using the “loadunix” utility described in [6], the programmer can then load the executable image onto a selected MC68030 processing node.

(a) PROGRAM exampleprog

OBJECTS
  top.c
  sub1.c
  sub2.c
  sub3.c
  drivers.sa
ENDOBJECTS

(b) PROGRAM <program_name>
Example 2. buildfile format and example

CHOBJS = $(CM)/header.o $(CM)/cm.o
OBJJS = top.o sub1.o sub2.o sub3.o drivers.o
exampleprog: $(OBJJS)
  ld -o exampleprog $(CHOBJS) $(OBJJS) \$(CM)/mv143.1d -m -lc > exampleprog.map

top.o: top.c
  gcc -c -W -I$(CM) top.c
sub1.o: sub1.c
  gcc -c -W -I$(CM) sub1.c
sub2.o: sub2.c
  gcc -c -W -I$(CM) sub2.c
sub3.o: sub3.c
  gcc -c -W -I$(CM) sub3.c
drivers.o: drivers.ro
cvtu -tO drivers.ro
asm drivers.sa -w +p=68030

Example 3. makefile for exampleprog

1.3 Results

The virtual memory implementation was applied to three application programs. We briefly describe those applications here.

First the simulation was run using one processor and then on multiple processors after loading the trace code in common global memory. The time taken to run the simulation was reduced by approximately 1/N where N was the number of processors used. Contrary to initial expectation the VME bus contention did not affect the results.

The results are presented in table 1 and table 2.

The second application used Kumaran II for fast parallel training of a neural network[8]. The most time consuming part of any neural network model is in the training of the network and finding the optimal parameters for the weights. We reduced the search time by running the model on all six processors of Kumaran II. Each processor was started with a slightly different normal perturbations of the weights with minimum squared error obtained from the previous runs. The minimum squared error after 10,000 iterations is plotted in Figure 1 as a function of number of iterations. While slow convergence is evident, the results are better than the uniprocessor case plotted in Figure 2.

For the third application the swept sine control system[5] was recoded to implement the virtual memory[8].

1.4 Conclusions

Recent trend in embedded control is to avoid virtual memory. Our experience has found that using virtual memory eliminates the chance of processors modifying the instructions and data of another processor and their own code. This in turn has reduced the amount of time spent in debugging multiprocessor applications. By avoiding paging to a secondary storage device and multi-level page tables, the normal overhead found in standard virtual memory implementations has been eliminated.

Acknowledgements

The authors would like to thank Bob Hilchey for hardware support in maintaining the Kumaran II multiprocessor system and Arul Anantha for system software support.

References


<table>
<thead>
<tr>
<th>Cache size</th>
<th>Instruction Cache Hit Rate</th>
<th>Data Cache Hit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 B</td>
<td>89.9</td>
<td>55.9</td>
</tr>
<tr>
<td>1 KB</td>
<td>92.7</td>
<td>66.0</td>
</tr>
<tr>
<td>4 KB</td>
<td>96.3</td>
<td>74.4</td>
</tr>
<tr>
<td>16 KB</td>
<td>98.4</td>
<td>75.9</td>
</tr>
</tbody>
</table>

Table 1

<table>
<thead>
<tr>
<th>Number of CPU</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27 secs</td>
</tr>
<tr>
<td>2</td>
<td>14 secs</td>
</tr>
<tr>
<td>4</td>
<td>7 secs</td>
</tr>
<tr>
<td>6</td>
<td>4.5 secs</td>
</tr>
</tbody>
</table>

Table 2

Execution time versus number of processors