An Implementation and Application of the Real-Time Producer/Consumer Paradigm*

Dan Poirier
Kevin Jeffay

University of North Carolina at Chapel Hill
Department of Computer Science
Chapel Hill, NC 27599-3175
{poirier,jeffay}@cs.unc.edu

Technical Report 90-038
October 1990

Abstract: We have created an efficient prototype implementation of the Real-Time Producer/Consumer Paradigm, allowing us to build multiprograms within a Unix[1] process. We present a brief introduction to the RTP/C paradigm and the design of a kernel that supports a concurrent programming system based on the RTP/C paradigm. We then discuss the results of adapting an interactive graphics program to use the RTP/C programming model.[2]

1. Introduction

The Real-Time Producer/Consumer Paradigm (RTP/C) is a paradigm for the design and analysis of real-time systems [Jeffay 89]. Associated with the RTP/C paradigm is a programming model, Communicating Real-Time Processes (CRTP). This paper describes a prototype implementation of an operating system kernel to support systems constructed using the RTP/C paradigm and CRTP, and some experience using this kernel with an interactive graphics application.

We will assert that the RTP/C paradigm is helpful in working with real programs and provides real benefits compared to traditional approaches. For example, we can determine bounds on response time for a thread of execution on the critical path through the program. We explain how we adapted a program to use the RTP/C paradigm and CRTP, and the insights we gained as a result.

---

* Supported in part by a Digital Faculty Program Grant from the Digital Equipment Corporation.

1 Unix is a trademark of AT&T Bell Laboratories.

2 Head-mounted Display support provided by ONR Contract #N00014-86-K-0680, NIH Grant #5-R24-RR02170, and DARPA-NSF Contract #DAEA 18-90-C-0044.

We will show that the RTP/C paradigm and CRTP can be implemented efficiently by describing our prototype kernel. This kernel implements light-weight tasks within a Unix process, which communicate via message passing. While the use of Unix prevents us from offering any rigorous response-time guarantees, it does enable us to test the suitability of the RTP/C paradigm and CRTP for actual programs.

Some features of the implementation include:

- Use of a single stack by all tasks to decrease context-switching overhead and improve memory utilization.
- Optimized scheduling so that tasks may be split into smaller tasks for modularity without incurring excessive overhead or lengthening the time it takes a message to propagate through the system.
- Use of ordinary Unix signals as input for the system.
- Built-in instrumentation to record how often tasks are executed.

Sections 2 and 3 will describe the RTP/C paradigm and the CRTP programming model. Section 4 gives an overview of our kernel design, with special attention to properties of the model that help to create an efficient implementation. Section 5 then gives details of our adaptation of a program to use RTP/C, and the results of that work. Finally, the appendix provides more details of our prototype kernel and may serve as a programmer's guide.

2. The Real-Time Producer/Consumer Paradigm

The Real-Time Producer/Consumer (RTP/C) paradigm was developed as part of a framework for the design and analysis of real-time systems. It defines a real-time semantics of inter-process communication.

As the name implies, the Real-Time Producer/Consumer paradigm focuses on a producer-consumer relationship between processes, in which one process, the producer, is generating data items and sending them to another process, the consumer. It assumes that the consumer cannot control the producer. The rate r at which data items arrive at the consumer can be measured in terms of the worst-case minimum inter-arrival time, p_{min}: $r = 1/p_{min}$. (See Figure 1.)

If the output rate can be realized over an arbitrarily long interval of time, and the consumer does not consume data items at rate r, then no amount of data buffering will suffice to make the system behave correctly (*i.e.*, without losing data items). Therefore, in such a system, the consumer must consume data items at the rate at which they are produced. In other

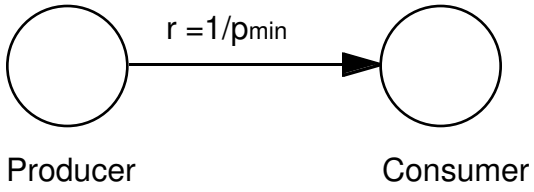

Figure 1: The Real-Time Producer/Consumer Paradigm

words, the RTP/C paradigm requires that the i^{th} output of the producer be consumed before the $(i + 1)^{st}$ output is produced.

For more information about the RTP/C paradigm, see [Jeffay 89, Section 2.2].

3. The Communicating Real-Time Processes Programming Model

The Communicating Real-Time Processes programming model (CRTP) allows us to apply the RTP/C paradigm to the analysis of systems larger than two processes. CRTP includes three abstractions: processes, data repositories, and communication channels.

CRTP views a program as a directed graph, in which nodes are processes and edges are unidirectional communication channels. Processes send messages along communication channels. Processes may be either sequential programs that execute on a single processor or physical processes in the environment external to the processor (such as input devices) that communicate with internal processes by interrupting the processor.

We postulate that each communication channel has a period p, which is the minimum inter-arrival time of messages on that channel. The maximum rate r at which messages are transmitted on a channel is therefore $1/p$. (See Figure 2.)

For simplification, we assume that the system has no global variables, and that tasks do not maintain state between execution requests. Therefore, an abstraction called a data

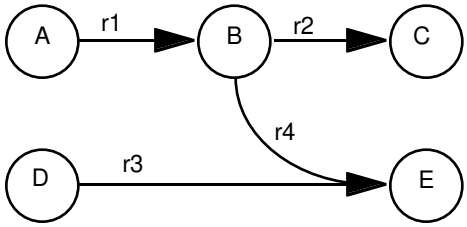

Figure 2: Communicating Real-Time Processes

repository models any state that needs to be preserved between task executions or shared among different processes. Tasks may store data in, and retrieve data from, these repositories.

The RTP/C paradigm and the CRTP programming model, and the analyses they make possible, are described in greater detail by Jeffay [Jeffay 89].

4. Implementation

We have implemented the programming model inside a single Unix process, using the C programming language. While the use of Unix precludes getting true real-time results, it is good enough as a prototype to investigate the usefulness of the model for real applications.

We call our implementation a "kernel," although it might be more appropriate to describe it as a library of support routines.

4.1 Tasks and scheduling

While the programming model deals with processes, the implementation consists of tasks. A process graph designed using the CRTP programming model is implemented by a set of sporadic tasks, one per process. Sporadic tasks are tasks that are invoked repeatedly, with some minimum time between execution requests. This minimum time is also called a period p.

In this case, when a process receives a message, the corresponding task makes an execution request. Since the tasks have minimum times between execution requests (determined by the minimum time between message arrivals), but may wait arbitrarily long between requests, they are called sporadic. Sporadic tasks are a generalization of periodic tasks, which are tasks that, once started, request execution at precise intervals. The first execution request of any task is called its release time.

When a message arrives, a deadline d is assigned to the task to ensure that the task processes the message before another one arrives on that channel. Each task must complete execution before its deadline. The minimum time before another message arrives is the channel's period, so the deadline is the arrival time of the message plus the channel's period, $d = t_{arrival} + p_{channel}$.

Scheduling requires deciding which task to execute next. Scheduling in the kernel follows a non-preemptive earliest deadline first (EDF) policy. In EDF scheduling, the task with the earliest deadline is always run.

Non-preemptive EDF is an optimal scheduling discipline[3] for sporadic tasks with arbitrary release times [Jeffay *et al.* 90]. In other words, if it is possible to schedule such a set of tasks without any of them missing a deadline, EDF will do so. We have also developed an algorithm to determine, *a priori*, whether it will be possible to schedule such a set of tasks [Jeffay *et al.* 90].

Our ability to schedule a collection of tasks is a function of the execution costs and periods of the tasks, and not of the topology of the process graph (although we use the graph to derive the periods using the RTP/C paradigm). This is one reason for distinguishing between processes and tasks. One consequence of this is that our decision procedure assumes the tasks are independent, and in particular that the worst case interleaving of execution requests can occur. But execution requests in fact correspond to message arrivals. In a process graph the tasks are decidedly not independent, and the worst case sequence of message arrivals may not occur. The decision procedure does not take this into account and thus is overly conservative. A set of tasks that is schedulable may be rejected by the decision procedure.

On the other hand, if the analysis decides that the tasks are schedulable, then we know that they are schedulable for any interleaving of requests, that is, any pattern of message arrivals in the program. This allows us to optimize our implementation to prevent poor performance that might occur when messages are passed along a string of tasks in series, due to the effects of propagation delay of such messages.

For example, suppose we have processes A, B, and C. Task A receives a message and sends another message to B, which then sends a message to C. We arrange that the tasks implementing processes A, B, and C all request execution when a message is received by process A, with scheduling ties broken in the order the tasks are connected (A before B before C). Thus, the deadline of the last task (task C) is determined by the message arrival time at the first (task A), and there is no delay added by a message traversing a string of

---

[3] With respect to scheduling disciplines that do not insert idle time.

such tasks. Showing that this is correct (and advantageous) is beyond the scope of this paper.

4.2 Unix signals

Unix signals are a class of software interrupts within Unix for which application programs can specify special handling. Programs can also request delivery of particular signals in certain circumstances. For example, Unix can interrupt a program with the signal SIGIO when specified I/O channels are ready for reading or writing. Unix can also interrupt a program at (roughly) regular intervals with the signal SIGALRM.

Our implementation allows programs to be driven by Unix signals. A task may be associated with each signal, so that delivery of that signal is treated as the arrival of a message for that task, and the task is scheduled for execution. The use of the SIGIO signal allows asynchronous input, while SIGALRM permits a task to act as a clock for the system.

4.3 Versions of the kernel

There are two versions of the kernel. They provide the same features, but differ in approach.

4.3.1 Initial version

The first version of the kernel implemented multiple threads of control within a Unix process. Each task consisted of some C code and its own execution stack. This required the use of in-line assembly code to perform context switches. It was hard to get the kernel working right, and it was also difficult to debug applications that used the kernel, since the typical Unix debuggers could not cope with an application that switched stacks.

This was in some sense a more faithful implementation of the CRTP model, since it provided actual multiple tasks. Each task consisted of an infinite loop which was blocked until a message arrived and the task was scheduled to execute. Then that task's context was restored and the task ran until it was ready to receive another message, whereupon it blocked again. A special "ACCEPT" call was used to block the task until a message was received and the task was scheduled to run again.

A typical task in the first version looked like this:

```
void task()
{
```

```
  loop
     Accept message (block until message received)
     Compute, possibly emit messages
  end loop

  /* never exits */
}
```

Interrupts (Unix signals) were handled by preempting the current task, placing an interrupt-handling task on the ready queue, and then continuing the interrupted task. This required mutual exclusion around the queue-manipulating code.

This first version of the kernel consisted of about 1800 lines of C, including comments.

4.3.2 Second version

The reason for the threads implementation was to allow each task to maintain its context between invocations. However, a closer examination of the CRTP model suggests that this is not necessary, since tasks do not maintain state, and our scheduling is non-preemptive. We therefore designed a new implementation without a separate stack for each task.

In the second version, each task is a C function. The scheduler runs a task by calling its function using a normal procedure call, with the received message as argument. The function returns when the task is complete. The only stack used is that of the main program; in fact, the whole system is in reality a sequential program. Scheduling is non-preemptive (each task runs until it completes), and tasks do not save state between invocations (except in data repositories), so letting all the tasks share one stack avoids messy and potentially expensive context-saving and stack-switching. No special statements are necessary for blocking tasks or receiving messages.

A task in the second version looks like this:

```
void task(message)
{
    /* Compute based on message */
    /* Optionally send messages */

    return;
}
```

When an interrupt occurs, a flag is set and execution of the current task continues immediately. Only when it is safe to do so does the kernel check the flag and add the interrupt handler to the ready queue. No explicit mutual exclusion is necessary.

Since there is no logical concurrency, understanding the system is much easier. The implementation also becomes more efficient due to the lack of mutual exclusion, since blocking and unblocking interrupts required the overhead of a Unix system call. Debugging under Unix is also easier.

The second implementation is simpler than the first. It consists of only about 1300 lines of C, including comments.

5. An Application of the RTP/C

The RTP/C concept has been tested by applying it to a real application, a program used by the Head-Mounted Display project in our department [Chung *et al.* 89].

5.1 Description of problem

The Head-Mounted Display project creates a virtual world by using special input and display devices to logically immerse the user in a computer-controlled reality. Two television screens are mounted on a helmet worn by the user, so that the user sees only what is displayed on the screens. Other hardware senses the position and orientation of the user's head, so that the appropriate part of the virtual world can be displayed to the user as he or she turns his or her head or moves around the room.

Another sensor detects the position of a hand-held ball, which acts as a three-dimensional mouse (complete with buttons), allowing the user to interact with the artificial reality.

5.1.1 Head-mounted Display System Design

The head-mounted display system consists of a VAX host computer running Unix, some position sensing hardware, and a special-purpose computer for graphics output. The sensing hardware is called a Polhemus[4]. The graphics computer is Pixel-Planes 4 [Eyles *et al.* 87, Fuchs & Poulton 81], a special purpose high performance SIMD computer developed at UNC for interactive graphics work. (See Figure 3.)

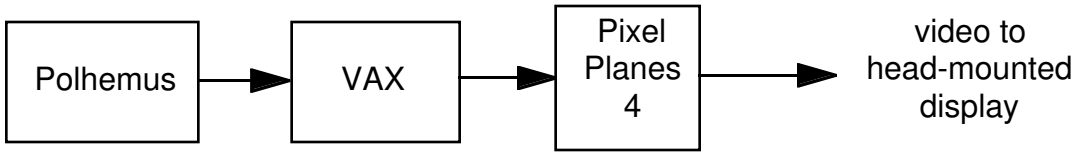

Figure 3: The head-mounted display system

[4] A 3SPACE Tracker, made by Polhemus Navigation Sciences.

In operation, an application program (running on the host computer) sends a request to the Polhemus. The Polhemus determines the current position and orientation of the user's head and the mouse, and sends a report back to the application. Based on the new information, the application sends requests to Pixel-Planes 4 to update its internal model of the virtual world and display a new picture for each of the user's eyes (slightly different pictures are displayed for each eye to provide stereo vision).

5.1.2 Time and the Head-mounted Display

The head-mounted display team has two time-related concerns in making their virtual world seem real. First, they wish to

minimize lag time, the time that elapses between movement of the user's head (or the mouse) and a display update reflecting the motion. Second, they wish the time between new pictures to be fast enough to create the illusion of smooth motion in the virtual world.

The first problem involves sensing the current position, communicating that position through the operating system to the application, and displaying a new picture based on that position. The hardware and operating system components of the lag time are out of our control.

The second problem is largely a graphics problem, depending on the speed of our application and Pixel-Planes 4. Again, the speed of the hardware is out of our control.

There are no hard and fast rules on what the times should be. A lag time of even 5 ms may be noticeable, and even that is very difficult to achieve. The head-mounted display team currently has a goal of 30 ms [Robinett 90].

For the appearance of smooth motion, 10 updates per second is too slow, while motion pictures use 24 updates per second, and television 30 [Robinett 90]. It has also been observed that usually, 12 new frames per second is indistinguishable from 24, except in certain circumstances such as moving point-of-view with changing perspective [Azuma 90]. In this application, the hardware imposes a maximum rate of 30 frames per second, since output is via NTSC standard video.

We were interested in studying lag-time in the head-mounted display system. Several questions arise: how is lag time defined? how much is there in our system? where does it come from? and what can we do to reduce it?. We weren't sure if we'd be able to reduce the lag time, but we hoped to get some idea of where it came from.

5.2 Applying CRTP and the RTP/C paradigm to the Head-mounted display

In order to test the applicability of CRTP and the RTP/C paradigm to real applications, we decided to adapt one of the programs used with the head-mounted display to the CRTP programming model, and see what analysis could reveal about the real-time behavior of the head-mounted display system.

5.2.1 The original program

The head-mounted display project has developed a variety of applications. We chose a simple one for our use, which displays a box floating in space. When a button on the mouse is pressed, the sides of the box move apart, revealing a white sphere inside, and a yellow square moving back and forth while spinning.

Most head-mount programs consist of a single control loop, as shown below. At the beginning of the loop, the program reads a report from the Polhemus to determine where the user's head and the mouse are. The rest of the loop computes and displays the new picture, and then everything repeats.

However, instead of requesting the report and waiting for it (it takes some time for the Polhemus to determine the positions of its sensors), the next report is requested immediately after receiving the previous one:

```
request a report
loop
   wait until report is ready
   read report
   request another report
   compute
   display
end loop.
```

Since there is a delay between requesting a report from the Polhemus and getting it, this refinement overlaps I/O with graphics processing. The application does computation while the Polhemus is preparing the next report, allowing more frequent reports and display updates. Even so, the update rate is fundamentally tied to the rate of Polhemus reports.

5.2.2 Process Graph

We implemented the floating box virtual world using CRTP. Our program consists of seven processes and two data repositories (Figure 4). We make heavy use of Unix signals (see Section 4.2).

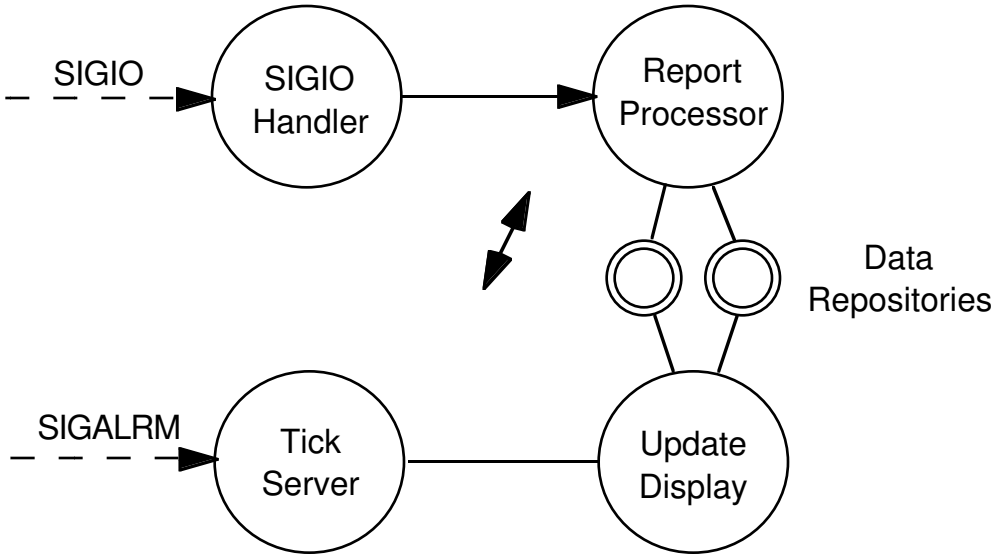

Figure 4: Our system

In the original program, the update rate was tied strictly to the rate at which Polhemus reports were received. This would be fine if the view changed only when we found out that the user or the mouse had moved, but our program included some animation (the square moves and spins, and when a button is pressed, the sides of the box move apart). Many head-mount programs include more detailed animation, which helps to make the virtual world seem real. By separating the display update rate from the Polhemus report rate, we could update the display more frequently to get smoother animation.

In our program, we arrange for the Unix signal SIGIO to be delivered whenever input is available from the Polhemus. A task called SigioHandler requests execution whenever we receive a SIGIO. When run, this task determines if a complete report has arrived yet. If so, it reads it and sends the report in a message to a second task, ReportProcessor. This task interprets the report to determine the position and orientation of the user's head and the mouse. It then stores the information in two data repositories.

Meanwhile, we have arranged for the Unix signal SIGALRM to be delivered periodically. A task called TickServer is scheduled whenever the SIGALRM occurs. This task counts SIGALRM's and regularly sends a message to the fourth task, UpdateDisplay. It reads the position and orientation information from the data repositories, and begins the processing needed to update the display, which is continued in tasks Display2, Display3, and Display4 by sending messages from each task to the next one. The result is a regular, periodic new display, at a rate unrelated to the input rate from the Polhemus.

5.2.3 Evolution of the program

In our original design, the UpdateDisplay task did all of the display processing. The display processing takes far longer than any other part of the system (see Section 5.3), and it must run to completion whenever scheduled (since our scheduler is non-preemptive). The execution time of the display processing task, UpdateDisplay, turned out to be longer than the period of many of the other tasks, so that it was impossible for them to run as frequently as they needed to. Breaking the task UpdateDisplay into four pieces lessened the delays in scheduling tasks with shorter periods, in effect providing explicit preemption points in our non-preemptive system.

We also changed the way we requested new reports from the Polhemus. A new request needed to be made only when the previous report had been completely received. Since the ReportProcessor task was invoked whenever a complete report was received, it was natural in our first design to request a new report in ReportProcessor. The alternative, to have the SigioHandler task request a new report immediately whenever a complete report was received, would have increased the maximum computation cost of SigioHandler even though this code was seldom needed (SigioHandler is invoked many times for each complete report received).

However, the delay between SigioHandler reading a complete report and ReportProcessor being scheduled to process it (and request a new report) was imposing a needless restriction on the rate at which we could receive Polhemus reports. We therefore made the SigioHandler task request a new report whenever a complete report was received.

In our present design, the TickServer task is unnecessary, since we could schedule UpdateDisplay directly when SIGALRM's occur. TickServer remains part of the design for generality. We may decide to schedule a task to read the Polhemus periodically instead of in response to SIGIO's. TickServer could count SIGALRM's and send messages to the tasks at different intervals to simulate multiple clocks. The execution of TickServer costs about 30 ms at each SIGALRM, but this is smaller than any other task by an order of magnitude.

5.3 Data

To assess real-time performance, we need computation times and periods of each task. We timed the original program and our version on a lightly loaded system (we were the only active user). For measuring programs and program fragments, the best clock resolution we could come up with was Unix's *gettimeofday* call, with a 10 ms resolution on our VAX.

Wherever possible we made repeated runs and divided the total time by the number of runs. Thus, all times given are average elapsed time, rather than the worst-case processor time we might have preferred to measure. Given the uniformity of the code and light system load the average case is probably little different from worst case (or best case).

5.3.1 Original program

In a test program we found that just reading Polhemus reports in a tight loop took about 67 ms per report. The entire original program, which looked like

```
request report
loop
   wait for report
   read report
   request next report
   update display
endloop,
```

also took 67 ms per iteration. It was immediately obvious that the program was bound by the Polhemus report rate. Indeed, updating the display only took about 28 ms, and the rest of each loop was spent waiting for the next Polhemus report. Thus, the original program had an update rate of about 15 updates/second.

This is a little misleading. Our program had a pretty simple virtual world to display. A more involved program would take more than 28 ms to calculate each update. However, any program that took less than 67 ms to display could benefit from dissociating the display update rate from the Polhemus report rate, and even slower programs would become more flexible, allowing lag-time to be traded off against update rate by varying the priorities of the tasks that read the Polhemus and update the display.

5.3.2 Adapted program

Task	Time (ms)
SigioHandler	0.720
ReportProcessor	0.778
TickServer	0.030
UpdateDisplay	8.280
Display2	5.900
Display3	8.020
Display4	8.060
UpdateDisplay—Display4	27.840

The execution times for our tasks are shown above. In our first design, the tasks UpdateDisplay and Display2 through Display4 were combined into a single task, shown here as UpdateDisplay – Display4.

Note that the combined times of UpdateDisplay and the three Display tasks run separately are greater than the time of the four tasks run together. This is a measurement anomaly that probably resulted from running the display tasks separately for timing. We split our original single display task, UpdateDisplay, at three completely arbitrary points. This made no different to Pixel-Planes 4 as long as we ran the four parts in sequence.

However, it happens that the graphics processing done by the first two tasks is handled by Pixel-Planes 4's front-end floating point processors, while the processing in the last two tasks occurs on the smart frame buffer. When run in sequence, the tasks are somewhat faster due to the multiprocessing between the parts of Pixel-Planes 4, but that advantage is lost when we run the same task over and over [Ellsworth 90].

We were hoping to continue getting Polhemus reports at close to 15 per second (apparently the maximum rate the Polhemus will support in our configuration), while increasing the update rate. Thus, we assigned periods to our tasks as follows (in ms):

	Updates/second		
Period (ms)	10	20	30
SigioHandler	7	7	7
ReportProcessor	67	67	67
TickServer	100	50	33.3
UpdateDisplay	100	50	33.3
Display2	100	50	33.3
Display3	100	50	33.3
Display4	100	50	33.3

The period for the TickServer and display update tasks is set to 1/(*the desired update rate*), which we varied from 10--30 per second. The 7 ms period for SigioHandler was determined empirically from measurements of a test program (SIGIO's were delivered about every 7 ms).

It is clear at this point that it is not possible for our implementation to guarantee that all tasks meet their deadlines. We would like to execute SigioHandler every 7 ms, but 3 of

our tasks take over 8 ms to execute and will block SigioHandler for too long to achieve our desired rate. However, SigioHandler is potentially called every time Unix receives another byte of input from the Polhemus. As long as we manage to execute SigioHandler once per complete report (30 bytes), we won't miss anything. Thus, missing a few deadlines for SigioHandler isn't critical. Unix ensures that we receive all incoming data, thus satisfying the spirit of the RTP/C paradigm, that no data is lost.

We ran our program at various (attempted) update rates to see what actual rates we observed, and whether the failure to guarantee meeting deadlines caused problems. Here are some of our results. The nominal update rate is the rate at which we requested SIGALRM signals (to trigger display updates) and the rate upon which we based the periods of the update tasks.

	Nominal updates/sec		
	10	20	30
Observed updates/sec	9.98	19.92	27.15
Polhemus reports/sec	14.84	14.84	11.59
Missed reports/sec	0	0.016	1.15

Missed reports occur when the SigioHandler sends a report to the ReportProcessor but the ReportProcessor has not yet been scheduled to process the previous report.

Looking first at the rate of polhemus reports, we sustained about the original 15 reports/second at 10 and 20 updates/second, but slowed to about 11 reports/second at the fastest update rate of 30 updates/second. This happens partly because the processor is becoming saturated and we take longer to request a new report after receiving one, and partly because we're missing some reports altogether.

With our clock set at 30 ticks per second, we managed to get update rates of over 27 updates per second, which is enough for smooth animation.

We had no easy way to measure lag time, but it was still noticeably slow in our program, just as it had been in the original. Since it takes 67 ms for each Polhemus report in our system, it's possible that up to 67 ms of lag-time is caused by the Polhemus, which we can't do anything about.

5.4 Analysis

We have no control over the time it takes the Polhemus to deliver a report (once requested), so for purposes of our analysis we will define response time as the time from our receipt of a Polhemus report to the display of a new image reflecting that report.

Although we could not guarantee that our tasks met their deadlines, we will make the analysis with the assumption that we could, to show the guarantees possible with analysis using the RTP/C paradigm. See Figure 5. This figure shows the worst case occurrence of execution requests for the tasks SigioHandler, ReportProcessor, and UpdateDisplay starting from the arrival of a Polhemus report. The line for each task represents the time from the execution request to the deadline of the task.

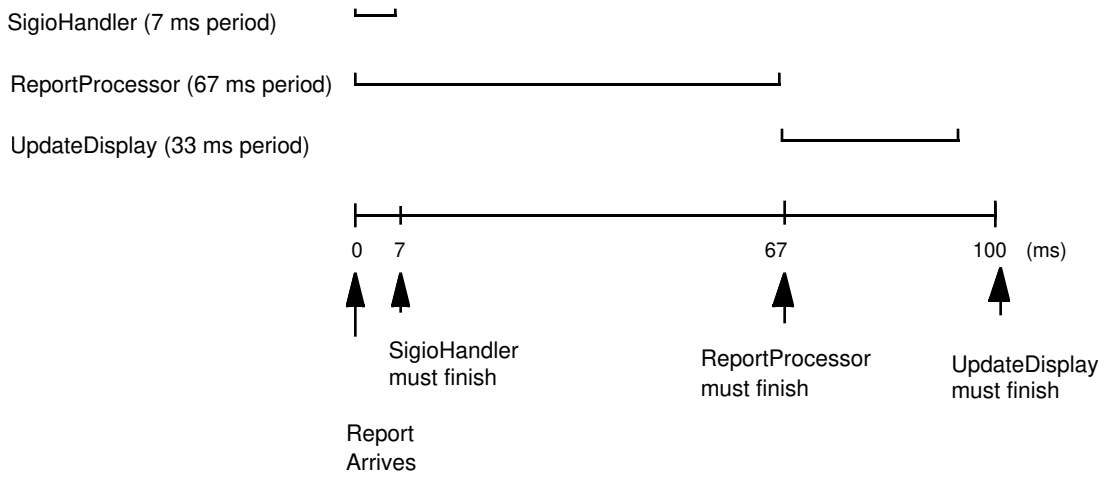

Figure 5: Analysis of response time

Starting our analysis from the arrival of a Polhemus report, the first thing that happens is that a SIGIO signal is delivered and we schedule the task SigioHandler. It reads the report and sends a message to ReportProcessor. We apply our scheduling optimization to SigioHandler and ReportProcessor, scheduling ReportProcessor as though it made its execution request at the same time as SigioHandler (at the arrival of the SIGIO). When ReportProcessor executes, it computes the positions of the user's head and the mouse and stores them in data repositories. Since the period of ReportProcessor is 67 ms, and it makes its execution request at the time of the original SIGIO, we can guarantee that the task will complete and the data will be placed in the data repositories within 67 ms of the report's arrival (assuming the tasks were feasible). In the worst case, it takes a full 67 ms.

Once the data is in the data repositories, the next execution of UpdateDisplay will use the new data. The worst case for this occurs when UpdateDisplay makes a new execution request exactly at the end of ReportProcessor's 67 ms request interval, and does not execute until the end of its request interval. If it made an execution request any earlier, it would execute earlier (not worst case); if its request were later, its previous request would have had a later deadline than ReportProcessor, would have executed after ReportProcessor, and used the new data, and we'd be considering that execution request instead of this one.

With an update rate of 30 updates/second, UpdateDisplay has a period of 33 ms. Our worst case delay is therefore about 67 ms (from ReportProcessor) + 33 ms (from UpdateDisplay), or 100 ms. Again, this assumes that the tasks are feasible and we can guarantee these deadlines will be met.

5.4.1 Improving performance guarantees

How could we improve the system's response time? One way is to look more carefully at the Polhemus report. It consists of two parts, one describing the head position and the other the mouse position. If we process each piece separately when it arrives, we will double the rate of ReportProcessor and halve its execution cost. Its period will be 33.5 ms (half of 67 ms). We can then guarantee that our response time from the arrival of a head report is 33.5 + 33 ms, or 66.7 ms. The response time from the arrival of a mouse report is the same. Unfortunately, the response time from arrival of a particular report (head or mouse) is not visible external to the system.

We might also notice the lack of synchronization between ReportProcessor and UpdateDisplay. ReportProcessor could send a message to UpdateDisplay when new data was ready. Then we would schedule UpdateDisplay as though it made an execution request at the same time as ReportProcessor, and assign it the same deadline, so we could guarantee a response time of only 67 ms. Unfortunately, this too has a drawback. UpdateDisplay is already executing with a period of 33 ms when the update rate is 30 updates/second. If we added another execution request every 67 ms, we would increase UpdateDisplay's CPU utilization by 50%. Its utilization is already 28 ms out of every 33 ms (at 30 updates/second), so a 50% increase would overload the processor even if there were no other tasks. Using RT-P/C analysis, we could experiment with slower update rates and determine how slowly we would have to run for our system to be feasible again.

Note that this change would make our system more closely resemble the original program, in which display updates were tied directly to Polhemus reports. The difference is that our system allows us to schedule more frequent display updates independent of the Polhemus report rate, to achieve smoother animation if desired.

Finally, we could increase the priority of ReportProcessor by making its period artificially short. If we had a faster processor so that we could meet our tasks' deadlines, this would be an entirely reasonable way to proceed. A shorter period for ReportProcessor would directly affect the response time we could guarantee.

6. Conclusions

- The RTP/C paradigm and the CRTP programming model are useful for real programs. We were able to adapt an interactive program with real-time requirements without too much trouble.
- Splitting the program into separate tasks allowed us to vary the update rate independent of the rate of Polhemus reports. RTP/C allowed us to analyze the system at various update rates, and determine what response time we could expect. Given a faster processor, we could have traded update rate for response time, using RTP/C to *a priori* assess the feasibility of different configurations.
- The CRTP programming model can be implemented efficiently. The state-less nature of tasks, together with the decision to use non-preemptive scheduling, allows low-cost "context switches" on a single stack. We only save state when we need to, by using data repositories.

The appendix gives further details of programming using our kernel.

7. Acknowledgments

We would like to thank the many people at UNC who helped us with this project. Warren Robinett, Jim Chung and Rich Holloway of the UNC Headmounted Display project were patient in explaining their work, answering our questions, and letting us borrow their equipment from time to time. Ron Azuma described Disney's experience with animation rates. Brice Tebbs and David Ellsworth explained how Pixel-Planes 4 is connected to the host computer and communicates with Unix.

8. References

[Azuma 90] Azuma, R., Personal communication, August 1990.

[Chung et al. 89] Chung, J.C., Harris, M.R., Brooks, F.P., Fuchs, H., Kelley, M.T., Hughes, J., Ouh-young, M., Cheung, C., Holloway, R.L., and Pique, M., Exploring virtual worlds with head-mounted displays, *Non-*

Holographic True 3-Dimensional Display Technologies, SPIE Proceedings, volume 1083, 1989.

[Ellsworth 90] Ellsworth, D., Personal communication, October 1990.

[Eyles et al. 87] Eyles, J.G., Austin, J.D., Fuchs, H., Greer III, T.H., and Poulton, J.W., Pixel-planes 4: A summary, *Advances in Graphics Hardware 2: Proc. 1987 Eurographics Workshop on Graphics Hardware*, 1987.

[Fuchs & Poulton 81] Fuchs, H. and Poulton, J., Pixel-Planes: A VLSI-oriented design for a raster graphics engine, *VLSI Design*, pages 20--28, Third Quarter 1981.

[Jeffay 89] Jeffay, K., *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations*, PhD thesis, University of Washington, September 1989, Technical Report #89-09-15.

[Jeffay et al. 90] Jeffay, K., Anderson, R., and Martel, C.U., On optimal, non-preemptive scheduling of periodic and sporadic tasks., Technical Report 90-019, University of North Carolina at Chapel Hill, March 1990. (Submitted for publication.).

[Robinett 90] Robinett, W., Personal communication, August 1990.

Appendices

A.1 Usage

This section will begin with some additional details about the programming model. It will then give instructions and a reference for using the kernel.

A.1.1 More on Communicating Real-Time Processes

The high-level description of CRTP in Section 3 left out some details that are only relevant when actually designing a system.

Tasks exist only while executing and therefore cannot maintain state from one invocation to the next, nor can they share data with other tasks (thus the need for data repositories). This property makes a simpler implementation possible (see Section 4.3.2).

A process may only have one connection to each other process, and may only send one message on each channel during an execution (although that message may contain many logical messages). Since tasks have minimum periods between invocations, and each channel is associated with a particular sending task, each channel has a minimum period between messages. A programmer specifies this period when building the system.

Input to the system is through special processes called input devices, which execute in response to external events and emit messages into the system. They have no incoming channels, but have handlers which interface directly with the kernel as described later. There may also be output devices, which are processes with no outgoing channels.

Communication between a process and a data repository is synchronous; *i.e.*, the process sends a message to the repository and waits for a reply. Data repositories and their connections to processes form a part of the process graph.

The rest of this appendix will consist of a short program example using our implementation of CRTP, then a section giving some description of how to use the features of CRTP, and finally references for the header files, data types, and functions provided.

A.2 Example

Here is an example program that uses two tasks, a data repository, and a signal handler. Task1 is invoked by the Unix signal SIGALRM (details of setting up SIGALRM have been

omitted; see your Unix manual). Task1 keeps a count of the number of times it has been executed using a data repository Repos, and sends this count as a message to task2, which prints it out.

```
#include <stdio.h>
#include <signal.h>

#include <sys/types.h>
#include "kernel_public.h"

struct repos_msg {
    int what;  /* 0 to read, 1 to write */
#define READ 0
#define WRITE 1
    int value;
};

Channel channel;           /* Channel declared here so it's in
                              scope for task1 */

/*
 * Repos
 *
 * A data repository.  Stores an integer value, which
 * can be retrieved or updated depending on the message
 * sent to it.
 */
MessageP
Repos(message)
    MessageP message;
{
    struct repos_msg *r;
    static int stored_value = 0;

    r = (struct repos_msg *) message;

    switch(r->what) {
    case WRITE:
        stored_value = r->value;
        break;
    case READ:
        r->value = stored_value;
        break;
    }
    return (MessageP) r;
}

/*
 * task1
 *
 * Invoked by SIGALRM, increments the count in Repos
 * and sends it to task2.
 */
void
task1(message)
    MessageP message;
{
```

```
    int execution_count;
    struct repos_msg rp;
    struct repos_msg *answer;


    rp.what = READ;                    /* Read number of
                                          previous executions */
    answer = SynchEmit(&rp);
    execution_count = answer->value;

    ++execution_count;

    rp.what = WRITE;                   /* Save new count */
    rp.value = execution_count;
    SynchEmit(&rp);

    Emit(channel, (MessageP) execution_count);
}

/*
 * task2
 *
 * Prints the value sent to it by task1.
 */
void
task2(message)
    MessageP message;
{
    int execution_count;

    execution_count = (int) message;
    printf("%d\n", execution_count);
}

main()
{
    TaskP Task1, Task2;

    /* Declare tasks */
    Task1 = CreateTask("Task 1", task1);
    Task2 = CreateTask("Task 2", task2);

    /* Create channel to task 2 with period 10 ms */
    BindChannel(&channel, Task2, 10);

    /* Arrange for SIGALRM to invoke task 1,
       with period 10 ms */
    AssignSignalHandler(SIGALRM, Task1, 10);

    /* Details omitted of arranging for SIGALRM to occur */

    /* Start system */
    MainLoop();
    /*NOTREACHED*/  /* Tell lint this is the end */
}
```

A.3 Using the kernel

This section will give instructions for what needs to be done in a program using the kernel. Refer back to Section A.2 for the example program.

A.3.1 Defining tasks

For each task, write a function of the form

```
void
function(message)
        MessageP message;
{
        /* code */
}
```

Then call *CreateTask*, passing it a name for the function and the function address. *CreateTask* will return a pointer of type *TaskP*.

A.3.2 Defining channels

For each communication channel, define a variable of type *Channel*. Call *BindChannel* with the address of the channel, the task pointer to the receiving task, and the period (in ms) for the channel.

If several tasks will send messages to the same receiving task, a channel must be set up for each sending task, to establish the proper buffering. Otherwise, messages can (and probably will) be lost.

A.3.3 Defining Data Repositories

For each data repository, write a function of the form

```
MessageP
function(message)
       MessageP message;
{
       /* code */
       return answer;
}
```

where the answer returned is a MessageP.

A.3.4 Sending Messages Between Tasks

To send a message to another task, call *Emit* with the address of the appropriate channel and a message pointer. You may only send one message to each task during an execution,

so if you need to communicate several pieces of data you must bundle them up into one message.

A.3.5 Using Data Repositories

To use a data repository, send it a message by calling *SynchEmit* with the address of the data repository function and a message pointer. *SynchEmit* will return the message pointer returned by the data repository.

A.3.6 Using Signals

To arrange for a task to be scheduled when a Unix signal occurs, call *AssignSignalHandler* with the signal number, the task pointer (returned by *CreateTask*), and the period (in ms) to be used in scheduling the task.

To return the signal to its normal handling (whatever it was before calling *AssignSignalHandler*), call *RevokeSignalHandler* with the signal number.

A.3.7 Starting the System

To start the system, call *MainLoop*. This function will not return.

A.3.8 Statistics

If compiled with the option *DSTATISTICS*, the kernel will keep count of various interesting statistics and print them on demand. To use this facility, you should arrange for your system to stop at some point and immediately call *StopTime* to record the elapsed time when the system stops. You should then disable any signals or other input. Finally, call *PrintStatistics* to print out the collected statistics.

The figures kept include, for tasks:

- Number of times scheduled (placed on ready queue)
- Number of times executed
- Number of times scheduled but not executed
- Rate of execution (number of times executed/elapsed time)
- Number of messages that were sent to the task but not delivered because the previous message had not been processed yet

For signals, the kernel records

- Number of times the signal occurred
- Number of times a task was run in response to the signal
- Number of times a task was not run because a previous signal was still outstanding (lost signals)
- Total of run and lost signals
- Rate (occurrences/second)

A.4 Kernel interface reference

A.4.1 Header files

The header files needed for using the kernel are

- *types.h* A few generic types used in the kernel
- *public.h* Partial definitions of types specific to the kernel, and declarations of all kernel functions
- *channel.h* Full type definition for channels
- *kernel.h* Type definitions needed by the kernel itself
- *message.h* Type definition for messages
- *sig.h* Type definitions for signal handling
- *task.h* Type definitions for tasks

You do not need to include all these files. Instead, you can just include *sys/types.h* (a system header file) and *kernel_public.h*, which includes all the other files.

On our system, these files are placed in the directory */currituck/csystemsD/rtpc/CRTP*.

A.4.2 Types

Public type names in the kernel are capitalized (each word is capitalized if the name has several words). A capital "P" is appended to pointer type names. Thus, *Task* is the type that describes a task, while *TaskP* is a pointer to an instance of *Task*.

Most of these names are typedefs defined in *public.h* with an incomplete structure definition; the complete definition is given in another file. For example, in *public.h* you will find

```
typedef struct task Task, *TaskP;
```

```
while in task.h there is a full definition of struct task:

struct task {
   ...
};
```

In the list below, if two files are listed, the first is *public.h* where the incomplete type definition occurs, and the second is the file with the full definition. You can take advantage of this information hiding by only including the header files necessary for your application.

- **Boolean** (*types.h*) Values of *FALSE* and *TRUE* are defined for this type. It is used in the kernel, and available for convenience.
- **Channel** (*public.h*, channel.h) Information about the connection between two tasks (sender and receiver).
- **ChannelP** (*public.h*) A pointer to a *Channel*.
- **MessageP** (*public.h*) There is no *Message* type. A *MessageP* is a "generic" pointer, currently of type *(void *)*, which is used by tasks to point to whatever type of message they wish to send. The kernel imposes no restrictions on the message. It just hands the pointer to the receiving task.
- **Task** (*public.h*, *task.h*) A data structure used by the kernel to keep track of information about tasks.
- **TaskFuncP** (*public.h*) A pointer to a function implementing a task. Such a function should be declared as follows:

  ```
  void task_function(message)
          MessageP message;
  {
     ...
  }
  ```

  For maximum portability, the argument should be declared of type *MessageP* and then cast to a pointer to the message type actually used.
- **TaskP** (*public.h*) A pointer to a *Task*.
- **Time** (*types.h*) A time value in milliseconds.

A.4.3 Functions

```
TaskP CreateTask( name, entry)
   char *name;
   TaskFuncP entry;
```

- *name* A name used to describe the task in kernel output.
- *entry* A pointer to the function that implements the task.

The *CreateTask* function declares a new task. The *name* is only used in informational messages and does not need to be unique or even meaningful. If *name* is NULL, a meaningless name will be assigned.

```
void BindChannel}( channel, receiver, period)
   ChannelP channel;
   TaskP receiver;
   Time period;
```

- **channel** A pointer to a *Channel* data structure that can be used to store information about this channel. The user program must provide this structure. It will also be used when sending messages, so having the user provide it gives more options for making it available to the tasks.
- **receiver** The task pointer returned by *CreateTask* for the task that will receive messages on this channel.
- **period** The minimum time in milliseconds between messages sent on this channel.

The *BindChannel* function sets up a channel that can be used to send messages to a task. See sections A.3.2 and A.3.4 for more information about creating and using channels.

```
void Emit( channel, message)
   ChannelP channel;
   MessageP message;
```

- **channel** A pointer to a *Channel* data structure that has been initialized for a receiving task by *BindChannel*.
- **message** A pointer to some data.

Emit sends a message from the currently executing task, along the specified channel, to the task that was specified in *BindChannel* for that channel. The "message" sent by *Emit* is just a generic pointer. You may pass a pointer to any data structure you like, remembering that when the receiver accesses it, the current task will no longer be active (so the data should be static or global).

When *Emit* sends a message to a task, the schedule calculates the task's deadline and places it on the run queue in earliest deadline first order.

```
void NullTask(message)
   MessageP message;
```

The null task is run whenever no other task has an outstanding execution request. The default null task does nothing. You may provide a replacement by writing your own *NullTask* and linking it into your program before the kernel library.

```
void AssignSignalHandler(signal, task, period)
```

```
int signal;
TaskP task;
Time period;
```

- **signal** A UNIX signal number.
- **task** A pointer returned by *CreateTask*.
- **period** A time in milliseconds.

The kernel can arrange to trap UNIX signals and run a task when they occur. This function will schedule the given task whenever the specified signal occurs, with a deadline based on the time of the signal and the period given. The signal number will be passed as the message (cast to a *MessageP*, of course).

```
void RevokeSignalHandler( signal)
   int signal;
```

- **signal** A UNIX signal number.

This function returns handling of the given signal to its status before calling *AssignSignalHandler*. We don't use this function, but it's available for completeness.

```
MessageP SynchEmit(repository, message)
   MessageP (*repository)();
   MessageP message;
```

- **repository** A pointer to a function implementing a data repository.
- **message** A pointer to a message.

This function (actually a macro defined in *public.h*) passes *message* as an argument to the repository function, and returns the *MessageP* returned by the repository. In the older, partially-preemptive version of the kernel, this function implemented mutual exclusion on data repositories. It is kept primarily to make repository access look like message sending (instead of the function call used to implement it).

```
void MainLoop()
```

MainLoop is the main loop of a system. It does not ordinarily return. It repeatedly runs the task at the head of the ready queue by calling the associated function with the appropriate message as an argument.