[7] Feldmeir, D.C., “Multiplexing Issues in Communication System Design,” *Proc. ACM Sigcomm*, October 1990, pp. 209-219.

[8] Gopalakrishnan, R., Parulkar, G.M., “Efficient Quality of Service Support in Multimedia Computer Operating Systems,” Technical Report WUCS-94-26, Washington University, September 1994.

[9] Gopalakrishnan, R., Parulkar, G.M., “Application Level Protocol Implementations to Provide QoS Guarantees at Endsystems,” *Proceedings of the Ninth IEEE Workshop on Computer Communications*, Duck Key, Florida, October 1994.

[10] Gopalakrishnan, R., Parulkar, G.M., “RMDP-A Real-time CPU scheduling Algorithm to Provide QoS Guarantees for Protocol Processing,” *IEEE Real-time Technology and Applications Symposium*, (Short Paper) Chicago, May 1995.

[11] Govindan, R., Anderson, D.P., “Scheduling and IPC Mechanisms for Continuous Media,” *13th ACM Symposium on Operating Systems Principles*, 1991.

[12] Katcher, D., Arakawa, H., Strosnider, J.K., “Engineering and Analysis of Fixed Priority Schedulers,” *IEEE Transactions on Software Engineering*, October 1993.

[13] Khanna, S., et. al., “Realtime Scheduling in SunOS5.0,” *USENIX*, Winter 1992, pp. 375-390.

[14] Leffler, S.J., McKusick, M.K., Karels, M.J, Quarterman, J.S., “The Design and Implementation of the 4.3BSD UNIX Operating System,” Addison Wesley Publishers, 1989.

[15] Liu, C.L., Layland, J.W., “Scheduling Algorithms for multiprogramming in a Hard-Real-time Environment,” *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, January 1973, pp.46-61.

[16] Maeda, C., Bershad, B. N., “Protocol service decomposition for high performance networking,” *14th ACM Symposium on Operating Systems Principles*, Dec 1993. pp.244-255.

[17] Marsh, B.D., et. al., “First Class User Level Threads,” *ACM Symposium on Operating Systems Principles,*, Oct. 1991, pp.110-121.

[18] Pingali, S., “Protocol and Real Time Scheduling Issues for Multimedia Applications,” Ph.D dissertation, Dept. of Elec. and Comp. Engg., University of Massachusetts, Amherst, September 1994.

[19] Robin Philippe, et al., “Implementing a QoS Controlled ATM Based Communications System in Chorus,” Internal Report MPG-94-05, Department of Computing, Lancaster University, March 1994.

[20] Sha, L., Rajkumar, R., Lehoczky, J.P., “Priority Inheritance Protocols: An Approach to Real-Time Synchronization,” *IEEE Transactions on Computers*, Vol.39, No.9, September 1990, pp. 1175-1185.

[21] Sharma, R., Keshav, S., “Signaling and Operating System Support for Native-Mode ATM Applications,” *Proc. ACM Sigcomm*, August 1994, pp. 149-157.

[22] Thekkath, C.A., Nguyen T.D., Moy Evelyn, Lazowska, E.D., “Implementing Network Protocols at User Level,” *Proc. ACM Sigcomm*, Ithaca, NY, September 1993, pp.64-72.

[23] Tokuda, H., Nakajima, T., Rao, P., “Real-Time Mach: Towards predictable real-time systems,” *Proceedings, USENIX 1990 Mach Workshop*, Oct 1990.

[24] Zhang, L., Deering, S., Estrin, D., Shenker, S., Zappala, D., “RSVP: A New Resource Reservation Protocol,” *IEEE Network*, September 1993, pp. 8-18.

need to lock variables shared between RTUs thus simplifying the task of the programmer. We have derived an improved schedulability test for our scheduling scheme that allows us to achieve a higher system utilization than can be provided by existing results. We have implemented our scheme within the NetBSD operating system. Our experiments indicate that our system is robust, behaves consistently, and has met our design objectives successfully. This facility is the first step in our long term effort to implement efficient user level protocols with QoS guarantees. We are planning to extend the RTU facility in the following ways—

- *Virtual Time Upcalls* (*VTUs*): This is motivated by the need to support protocols in user space that do not require QoS guarantees. A VTU is similar to an RTU except that it does not have a period nor a real-time priority. Unlike an RTU whose delivery is triggered by the callout mechanism every period, an active VTU is delivered whenever the scheduler runs the process that contains the VTU. This is determined by the process scheduling policy. The advantage of using VTUs is that we can still retain the locking benefits of using the RTU pre-emption scheme in which the running handler is not pre-empted by another in the same process. Another general idea that VTUs exploit is a closer integration of *scheduling* and *protocol processing*. This is a useful feature for user level implementations of a protocol such as TCP that must deal with the case when one of the peer processes terminates abruptly due to a local error (bus error, keyboard interrupts etc.). In such a situation, the scheduler must not deallocate the terminating process until the protocol has had a chance to perform any cleanup operations with its peer. If the scheduler knows about active VTUs then it can deliver them a failure indication and wait until the protocols have cleaned up their state. Without this integration, other techniques such as migrating state to a trusted server or back to the kernel have to be adopted to deal with local failures. We will be exploring the ramifications of integrating scheduling and protocol processing in the near future.
- *Other Enhancements*: We are considering several enhancements to the RTU facility. Currently we have a restriction that a process should not make system call that may block indefinitely if it has active RTUs because upcalls are invoked only if the process is runnable. Thus a process must use the *rts_sleep* call to wait for its RTUs to complete before it makes a blocking system call. We can get around this restriction by putting *wrappers* around system calls that allow a system call to be restarted if it is interrupted by an RTU invocation[5]. We also have plans to allow RTUs to be created with any period. Currently it is restricted to multiples of 10 msec. This will involve coming up with an efficient run queue data structure.

We are building a protocol architecture around the RTU facility and will be developing multimedia applications that will have demanding bandwidth and processing requirements. We plan to use as our testbed other in-house efforts underway to develop high speed ATM switches, ATM host interfaces, video storage servers, and multimedia transport protocols.

References

[1] Clark, D.D., “The Structuring of Protocols using Upcalls,” *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, 1985, pp.171-180.

[2] Clark, D. D., Jacobson, V., Romkey, J., Salwen, H., “An analysis of TCP processing overhead,” *IEEE Communications Magazine*, 27(6) 1989, pg. 23-29.

[3] Clark, D.D., Shenker, S., Zhang, L., “Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism,” *ACM Sigcomm*, August 1992, pp.14-26.

[4] Dannenberg, R.B., et. al., “Performance Measurements of the Multimedia Testbed on Real-Time Mach,” Technical Report CMU-CS-94-141, School of Computer Science, Carnegie Mellon University, April 1994.

[5] Edwards, A., Muir, S., “Experiences Implementing a high performance TCP in user-space,” *Proc. ACM SIGCOMM*, 1995, pp. 196-205.

[6] Eicken, T.V., et. al., “U-Net: A User-level Network Interface for Parallel and Distributed Computing,” *Proc. of the 15th ACM Symposium on Operating Systems Principles*, 1995.

effect on UDP throughput when both the client and server are implemented using the base UNIX process mechanism. When the server alone is implemented as an RTU, the throughput is much higher. For example, with 8KB packets, the throughput for the RTU based server is 5 times that for the normal server. In addition, the throughput of the former remains almost constant over time. For the normal server, even with a socket buffer size of 192 KB, packets were lost consistently due to overflow. This is because the server was not getting scheduled frequently enough to be able to drain its socket buffer. With the RTU based receiver there was no socket buffer overflows. Thus the throughput was much higher.
When the client was modified to send data within the RTU handler, the maximum throughput was obtained (almost double for 8KB packets) and remained unaffected by background load. For 8KB packets the client–server pair can sustain 80 Mbps each (160 Mbps aggregate throughput) without any packets being dropped due to socket buffer overruns. This shows conclusively that the RTU mechanism not only provides high throughput but maintains it over long durations and in the presence of background load.

8. Related Work

We have already made a detailed comparison of real-time upcalls with real-time thread facilities as provided in RT-Mach and Solaris. To summarize, RTUs take advantage of the iterative nature of handlers to schedule them non-preemptively from one iteration to the next. As a result, a handler need not lock shared data within an iteration. In addition, this scheme reduces the number of context switches since an RTU is allowed to complete a minimum number of iterations every time it is invoked. General purpose RT-Threads do not provide this flexibility and are therefore a less efficient mechanism for protocol processing. We review some related work that either complement our work, or have some features similar to it.

The split scheduling scheme[11] resembles the RMDP scheduling policy in that it has a user level scheduler (ULS) and kernel level scheduler that communicate using shared memory. A ULS in a process decides which of its threads has the highest priority, and based on this information from all processes, the kernel scheduler determines which process has the highest priorities. The main advantage of this split scheme is that context switches within a process occur without kernel intervention. It makes no attempt to reduce the number of inter-domain context switches as RMDP does with its deferred pre-emption scheme. Furthermore, it does not address the issues of real-time locking and in this respect suffers the drawbacks of other fully pre-emptive schemes. One drawback of the scheme is that it requires a high resolution timer to be mapped to every process and this is not always possible.

We also look at a user level TCP implementation[5] that describes some of the problems in moving protocols to user space. In this scheme, a single TCP connection was implemented with two processes–the “upper” one responded to user requests for sending and receiving and the “lower” half processed network events such as ACKs and timer expirations. Because these two halves could pre-empt each other locking of shared variables was needed. Furthermore, when the “lower” half was made real-time, it would pre-empt the “upper” half that was currently holding a lock. The “lower” half would fail to obtain the lock yielding the CPU, causing two unnecessary context switches. If RTU handlers are used instead of processes, then both these problems can be avoided.

Finally we take a look at the U–Net implementation[6]. In U–Net user processes can directly send and receive data over the network interface without kernel intervening in the data path. Our work can benefit from this facility, and can build on top of it by providing scheduling support for QoS guarantees. U–Net (and in the future APIC) provides an efficient way to move data between process and network adaptor whereas the RTU facility ensures that the data gets processed at the required rate within the endsystem. Thus our work is orthogonal and complementary to these efforts.

9. Conclusions and Future Work

We have presented a simple design for providing QoS guarantees for processing within the endsystem. Our design does not require a kernel level thread facility to be available and is therefore more portable. It also retains all the benefits of existing real-time thread facilities such as implemented in systems such as RT-Mach. We have also been able to take advantage of the repetitive nature of protocol processing to implement a novel scheduling mechanism. Our scheduling mechanism allows us to reduce the number of expensive context switch operations. It also eliminates the

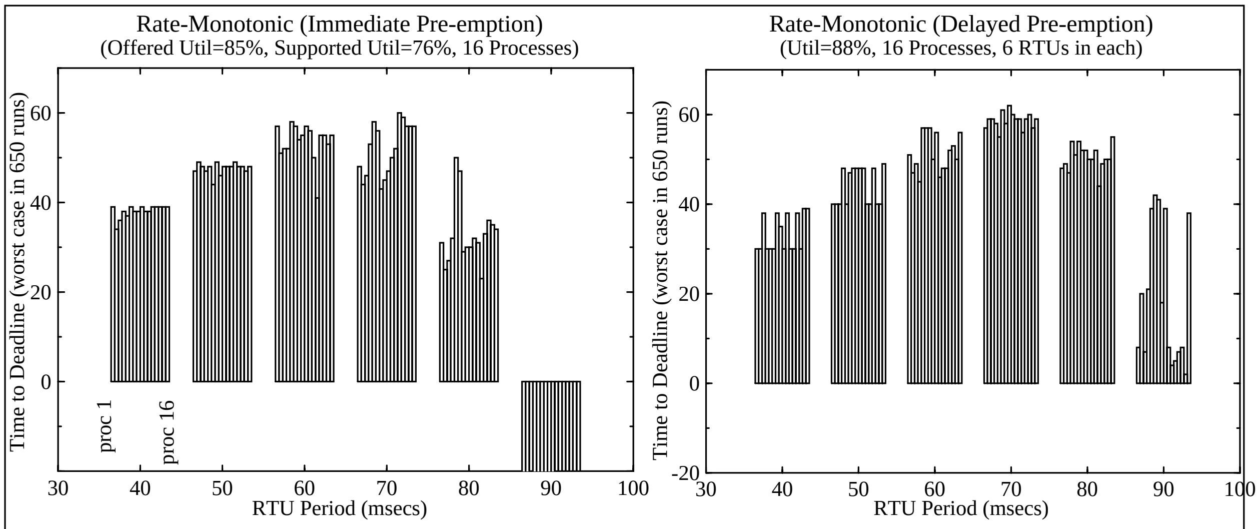

FIGURE 8. Effect of Pre-emption Delays

7.3 Experiment 3: UDP Performance with RTUs

In this experiment we implemented a client process that sends data packets to a server process in the same machine (loopback). The packet sizes ranged from 1/2 KByte to 8 KB. The first client-server pair was implemented in the usual way where the client program calls the `send`() system call in a loop in the `main`() function, and the server calls the `recv`() system call until all the packets have been received. The second client-server pair was identical to the first except that RTU handlers of 20 msecs and 10 msecs were created in the client and server programs respectively. The RTU handler in the client does one batch of `send`() calls in each period and returns. The RTU handler in the server calls `recv`() in a loop until it has read one batch of packets (or there are no more packets waiting to be read) and returns. We measured the throughput at the receiver for both client-server pairs for different number of packets sent. These are shown in Figure 9 for the two cases. UDP was in the kernel in both experiments.

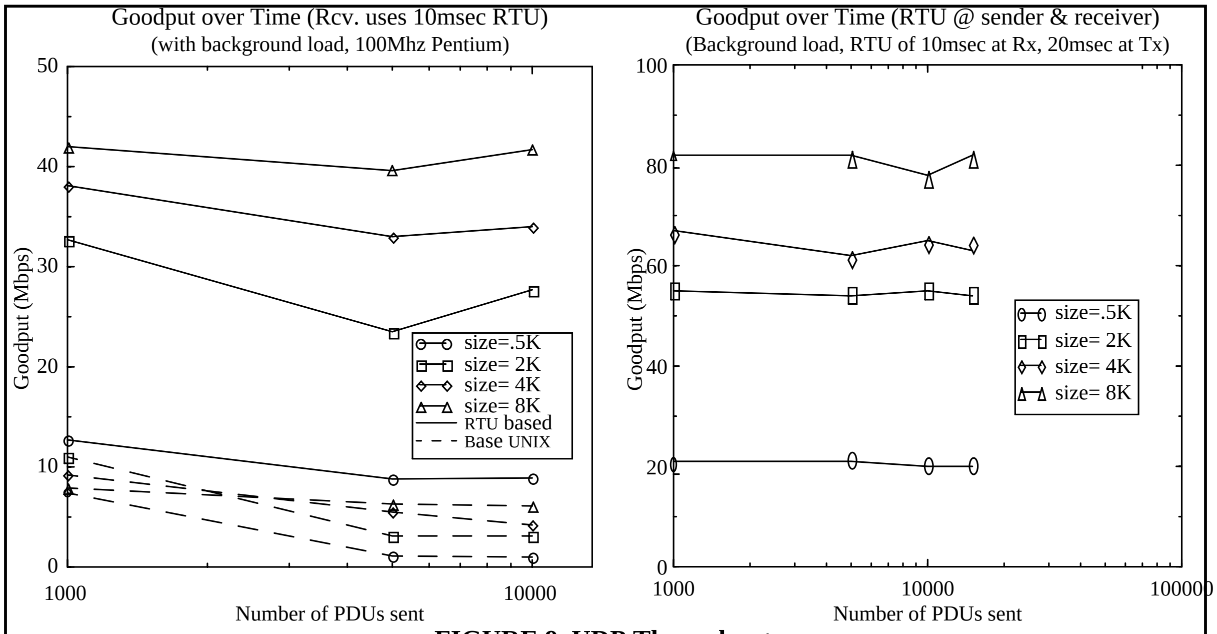

FIGURE 9. UDP Throughput

Both plots show that throughput increases with increasing packet sizes. This is due to the fact that larger chunks of data are moved across the user–kernel boundary. The first plot shows that background load on the CPU has an adverse

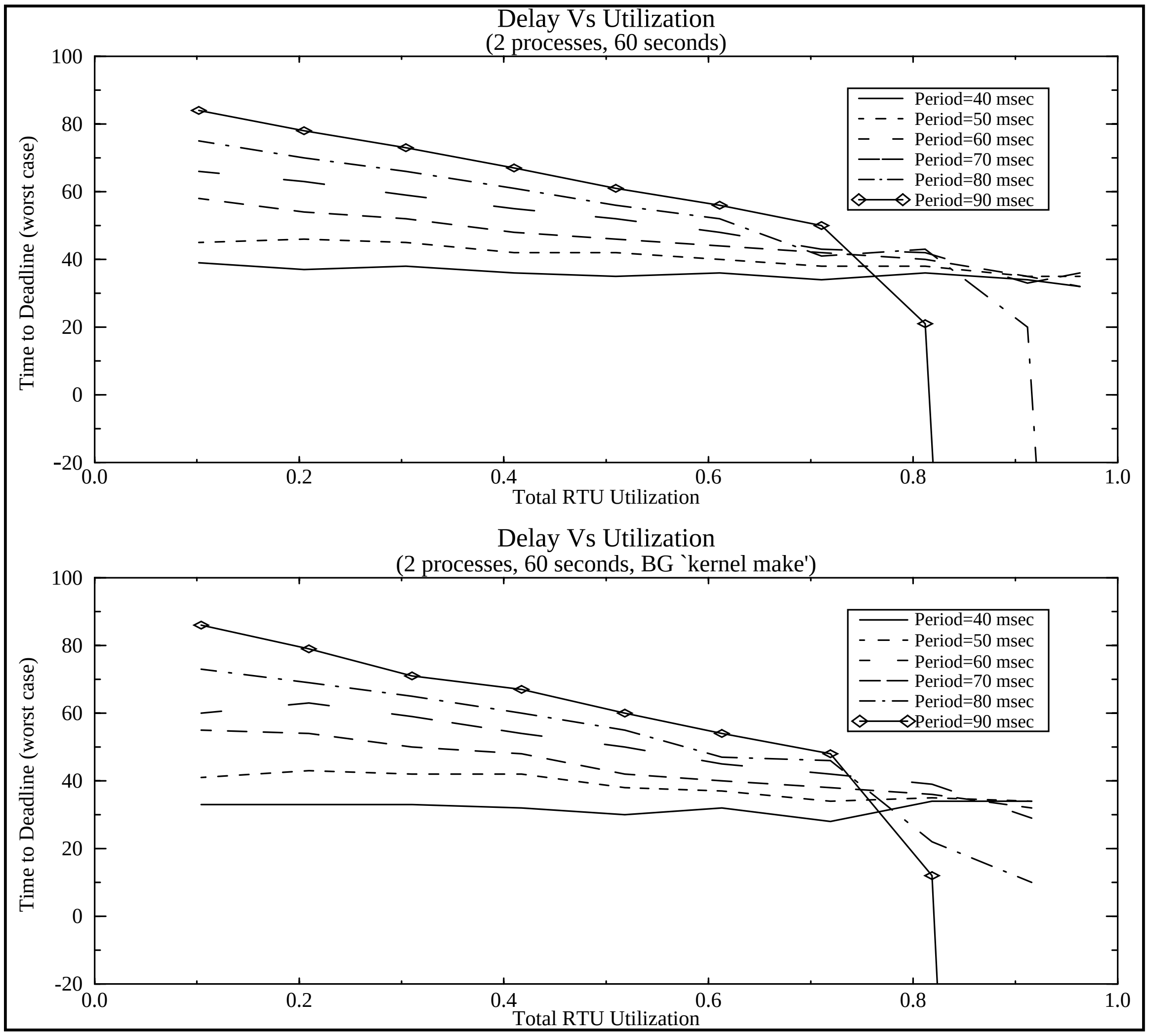

FIGURE 7. RTU Utilization Plots

7.2 Experiment 2: Reduction in Context Switches

The setup was the same as for the previous experiment except that we created 16 processes each with 6 RTUs. We compared two cases, one in which pre-emption was immediate and the other in which the RTU defers yielding the CPU until it completes certain number of iterations (usually half or all). For each RTU we measured the worst case time to deadline over time and plotted it against the RTU periods. We also verified that no deadlines were missed on account of delayed pre-emption. Figure 8 shows the cases without and with pre-emption delays.

We notice that the 90 msec RTU in all the 16 processes missed its deadlines when pre-emption was immediate. With delayed pre-emption the 90 msec RTU could be accommodated. The possible reason for this is that , there must have been at least one invocation of the 90 msec RTU in the first case that must have been pre-empted often enough to have missed its deadline. In the latter case, the 90 msec RTU always was able to complete its requirement before yielding without affecting the smaller period RTUs. We observed a 35% reduction in context switches overall when we used delayed pre-emption. The gain in useful utilization using delayed pre-emption is around 12%. This is significant considering that the maximum possible utilization cannot be higher than 100%.

cess structure. A side benefit is that this structure is in kernel space and so there can be no page fault associated with accessing it.

6.1.4 UNIX Signal Handling

Handling UNIX signals while in the context of a RTU handler is an important issue. Debuggers such as `gdb` depend on the STOP and CONT signals to set breakpoints and resume the target process. When a process gets a STOP signal while running, the RTU run queue must be purged of any pending RTUs for this process. This is done exactly as in the case when the process goes to sleep. When a CONT signal is received for a stopped process, the RTUs are put back in the run queue. By handling STOP and CONT in this way, RTU handlers can be debugged using a debugger just as any other user program. Other UNIX signal handlers are handled in keeping with UNIX semantics. Thus, the <ctrl-c> signal from the tty causes the program to terminate (if it is not caught) even if an RTU is running at that time.

7. Experimental Results

Our initial implementation was in the NetBSD kernel on a Sun 4/360 machine based on the Sparc-1 processor clocked at 25Mhz. We were able to port it to a Gateway 2000 PC with a 100Mhz Pentium processor in a matter of a week. We consider this to be a strong indication that the RTU facility is easily portable across CPU platforms. The aim of our experiments was to verify if the RTU facility meets its design goals of *timeliness* and *guarantees*, and is unaffected by background loads. We also measured the efficiency gains from the reduction in context switching resulting from the RMDP scheduling policy. Finally, we used RTUs to program a client-server application to verify if it can provide throughput guarantees in the face of background loads. All results are from experiments conducted on the 100 MHz Pentium platform. Utilization experiments were run for a duration equal to the least common multiple of the upcall periods since the load pattern repeats after this duration. We set the number of clock ticks per second to be 1000 which means that the clock ticks every 1 millisecond (msec).

7.1 Experiment 1:Ability to Meet Deadlines

We created 6 RTUs in each of two processes with periods ranging from 40 msecs to 90 msecs in increments of 10 msec. We varied the total CPU utilization consumed by the 12 RTU handlers and measured how close each invocation came to its deadline. The utilization for a handler was obtained by measuring the time it took to complete under idealized conditions when there were no page faults or pre-emptions, and dividing this time by its period. For each invocation of an RTU, we noted its completion time and subtracted it from its deadline to obtain the time remaining for its deadline. We then took the minimum of these remainders (which represents the worst case) among the 40 invocations (which represents the worst case) and plotted it against each utilization value as shown in Figure 7. In this experiment, no pre-emption delay was introduced by the handlers, which means that they yielded as soon as a higher priority RTU became ready. We repeated this experiment with programs such as `primes`, `netscape`, and `make` running concurrently. The first thing we notice is that in both cases all deadlines were met when utilization was below 0.8.As can be seen, the completion times for RTUs came closer to their deadlines with increasing utilization. This is due to the fact that at higher utilizations, it is more likely that an RTU is running when another RTU becomes runnable. Thus lower priority RTUs have to wait for the running RTU to complete before they can run. Similarly, higher priority RTUs have to wait until the running RTU is switched out before they can run. Furthermore, we notice that the difference between the completion time and deadline for the highest priority RTU did not change much with increasing utilization, but for lower priority RTUs the change was more pronounced. This is only to be expected because higher priority RTUs should not be affected by lower priority ones. The first deadline miss occurs at a utilization above 80% which is much higher than the theoretical worst case bound. This shows that OS overheads do not significantly affect schedulability. We would like to point out that the NetBSD kernel is non-preemptible, and so a handler may have to wait for the time that a system call executes in the worst case, before it can run. Thus background load can affect an RTU only for the duration of the longest system call and our results indicate that this worst case does not affect performance.

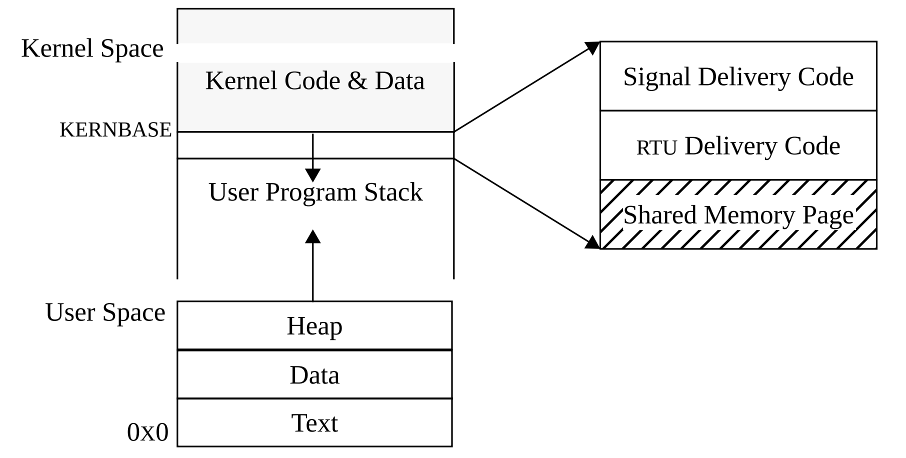

FIGURE 6. A Unix Process

mechanism is self-perpetuating as long as the RTU is active. The function also places the RTU in the priority run queue as shown in Figure 4. If the priority of the new RTU is higher than that of a running RTU (if any), the *yield_request* field is set in the shared memory of the running RTU. After this function runs, the system returns from the interrupt routine to user mode. Just before giving control to a user process, the RTU run queue is examined and the appropriate RTU handler is invoked as described in Section 7.1.3.

If a process goes to sleep when an RTU handler is running due to a page fault for example, all the RTUs belonging to the process that are in the run queue have to be purged. Later, when the process is woken up, these saved RTUs are put back in the run queue in the *wakeup* routine. If a higher priority RTU of the woken up process has become active while the process was sleeping, then the system ensures that it is invoked only after the handler that was interrupted is resumed first. This is necessary because our semantics dictate that a running handler cannot be pre-empted in the middle of an iteration. When an RTU in one process is blocked, RTUs in other processes may run. Currently we do not check whether a handler has missed its deadline before upcalling it.

6.1.3 RTU Invocation

If the RTU run queue is not empty when the system returns to user mode, then the RTU invocation and restoration routines (shown in Figure 4) get control. Invoking an RTU handler involves switching (if necessary) to the process that contains the handler, giving control to the handler function after saving process state, and restoring this state after the handler returns. Most of these operations are performed in the *userret* function that is invoked just before returning to user mode after a system call or an interrupt. *Userret* checks the RTU run queue and dequeues the highest priority RTU. It then checks if the currently running process is the one that contains the handler. If not, it does a context switch operation to the appropriate process. Once the correct process starts running, the kernel arranges for the handler to be called as soon as the system returns to user mode.

Giving control to a handler involves saving the state to which the process would have returned had the handler not become ready, and branching to the RTU *trampoline* code as shown in Figure 6. The trampoline code is passed the address of the handler and it invokes the handler function. After the handler function returns, the trampoline code makes a system call to the kernel so that the saved process state can be restored. The trampoline code is required because one cannot depend on the programmer who wrote the handler to make the system call. The trampoline code is copied on the user program stack when the process is *execed*. The trampoline idea is borrowed from the NetBSD kernel which uses it to deliver conventional signals.

The fact that RTU handlers are not fully pre-emptible makes invoking an RTU even more efficient than delivering a normal signal. Because normal signals are pre-emptible, a potentially large amount of the process state has to be saved on the user stack. However, to deliver an RTU we need only limited user state which can be stored in a per pro-

2. *rts_run* : This call tells the kernel to place an RTU in an internal run queue to be delivered to the process periodically. After this call the RTU, is considered to be in the active state.
3. *rts_suspend* : This call has the opposite effect of *rts_run* and is used to suspend the current and future delivery of the RTU. On suspension, any pending call to the handler (if any) in the internal run queue is removed, and any entry in the internal ready queue that triggers the next delivery is also removed.
4. *rts_close* : This call suspends delivery of the RTU and frees all internal resources and state associated with the RTU. It is similar to a socket close and can be called explicitly by the process or occurs implicitly when the process exits.
5. *rts_sleep* : The *rts_sleep* call is provided so that a process can wait for all its active RTUs to finish their work. This call resembles the UNIX *select* call. The call takes a bitmask of RTU descriptors that the process is waiting on as its argument. When the *rts_sleep* call is made, it checks to see which of the RTUs in its mask are no longer in the active state (i.e in the suspended state). If there are any, it sets the corresponding bits in a bitmask and returns to the caller. If no RTU has become inactive, the process goes to sleep waiting for an RTU to become runnable. As soon as an RTU becomes runnable, the process is woken up. The handler is then run, and the *rts_sleep* system call is *restarted* when the handler returns. Once all RTUs have completed their tasks, the main program may continue.

TABLE 1. API Functions

Name	Arguments	Remarks
rts_creat	timing attributes handler address and arguments shared memory value-result	Used to create an RTU Returns a descriptor
rts_run	RTU descriptor	Puts RTU in run queue for periodic delivery
rts_suspend	RTU descriptor	Suspends future delivery of RTU
rts_close	RTU descriptor	Closes and frees an RTU
rts_sleep	mask of RTU descriptors	Used to wait for RTUs to finish running

6.1 RTU Internals

There are three main internal components in the RTU facility. The first is the implementation of the shared, non-pageable data structures between each RTU and the kernel. The second is the internal routines that run off the clock interrupt and determine when RTUs become runnable, and which handlers to invoke according to the RMDP policy. The third is the operations performed to deliver an RTU and to clean up state after its handler returns.

6.1.1 Shared Memory

Figure 6 shows the structure of a UNIX process. The user program stack begins at KERNBASE and grows downwards. We allocate a page of memory on the user stack, to hold the data structures for RTUs in the process. Currently a page can hold 128 such structures. The page is allocated when a process is *execed*, and is marked as non-pageable with the *vm_map_pageable* call. This allows the kernel to directly read/write fields in the shared memory without having to use *copyin/copyout* functions. This saves the extra work and potential delays due to page faults. When an RTU is created, a structure is allocated to it from the shared page. Because the virtual address of the shared page is the same in all processes, the kernel need not store any state information on a per process basis.

6.1.2 Clock Based Operations

The *rts_run* API function uses the *timeout* routine[14] provided in the kernel to make an active RTU runnable. It calls *timeout* with three arguments—a kernel function to be called, the number of clock ticks after which the function must be called, and an argument to pass to the function. *Timeout* arranges for the function be called after the specified number of ticks. Once the function is called, it schedules itself in the future using the *timeout* routine as before. Thus the where B_i is the worst case blocking time of task *i* due to lower priority tasks. We could have used the above analysis by assuming the worst case blocking time B_i of an RTU *i* to be the maximum of the durations for which lower priority RTUs could delay yielding the CPU. In other words we could set the blocking time of task *i* to[1]

$$B_i = \max_{j > i}\left(C_j \cdot \frac{b_{p,j}}{B_{p,j}} \right)$$

Here B_i is obtained by calculating the time that is taken by each RTU *j* with priority lower than that of RTU *i* to process its minimum number of PDUs $b_{p,j}$, and choosing the maximum among these. However this result is still conservative because it does not take advantage of the fact that the blocking (or delay) time is used by the lower priority RTU to do useful work. In our analysis we divide the computation time C_i into quanta c_i where a quantum is the time to process the minimum number of PDUs for RTU *i*. We then obtain the following result which states that the task set is schedulable under the RMDP scheme if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + c_{max} \cdot (1/T_1 - 1/T_n) \le n\left(2^{1/n} - 1\right)$$

where c_{max} is the largest quantum in the set of RTU, and T_n is the maximum period among the RTUs. The improvement is captured by the term $(-c_{max} \cdot 1/T_n)$. The details of the proof are beyond the scope of this paper and may be found in [10]. If the range of periods is small, then our result gives a much higher utilization bound compared to[20]. Ours is a worst case result, and when we simulated a standard multimedia task set taken from [18], we observed that the available utilization under RMDP was almost the same as for RM. More importantly, we saw a 30% reduction in number of pre-emptions under RMDP and more details can be found in [10]. Our analysis of delayed pre-emption is independent of the RM priority scheme itself and so is applicable to other real-time scheduling algorithms as well. We have used RM mainly because of its ease of implementation.

5.4 Security and Availability

A point of concern is security and availability. Operating systems such as Solaris restrict the use of real-time facilities to supervisory processes. The rationale is that an ordinary user could consume all the utilization making the system unavailable for others. We too could impose this restriction because we can then assume that the handler can be trusted to yield the CPU according to the pre-emption scheme. However, it is strictly not necessary to make this assumption in our implementation because we can detect if a malicious user is violating the scheme by checking if the current invocation is still running when the next invocation becomes ready. In such a case, several recovery options are available. An exception could be raised which could terminate the process if it is not caught. Other options such as user quotas are also under consideration. We believe that bugs in the protocol library code itself that could cause the RMDP scheme to be violated would be weeded out before being released for use.

6. RTU Implementation Details

We first describe the system calls that are provided to create and manage RTUs as shown in Table 1, "API Functions," on page 12.

1. *rts_creat* : This system call creates an RTU and returns a *RTU descriptor* to the caller. Currently the RTU descriptor is the same as a file descriptor but this is strictly not necessary. Subsequent operations on the RTU can be performed by referring to the descriptor. Several attributes for an RTU can be passed as arguments to the *rts_creat* call. One set of attributes are the timing information such as the period, the computation time for a PDU, and the number of PDUs to be processed per period. The other argument contains the address of the upcall handler, and the pointer to a protocol control block (PCB) allocated by the program. If the schedulability test is satisfied, the call returns successfully. The caller also gets the pointer to the shared memory data structure.

1. We assume that higher subscript tasks have lower priorities

5.2 The RTU Handler

The job of the RTU handler is to process PDUs for a particular protocol session. The handler communicates with the scheduler through the shared memory to participate in the scheduling scheme. The handler is declared in the user program as follows:

```
void SigHandler(arg)
   struct rts_funargs *arg;
   {
      struct mypcb *proto=arg->pcb; /* protocol control block */
      struct rts_smstr *shmptr=arg->smstr; /* shared memory structure */
      /* Do Protocol Processing here */
   }
```

The handler is called with an argument that contains the address of the protocol control block (PCB) and the address of the shared memory structure that it uses to communicate with the kernel. Typically the PCB would be registered with the kernel during RTU creation time. For example, if the handler was implementing the TCP protocol, the PCB would be the *tcpcb* structure that is used by the NetBSD implementation. This structure would contain all the state information pertaining to the connection and serves to isolate different protocol sessions in a process. Since the PCB must persist across multiple calls to the handler it cannot be a local variable in the handler function because it would get deallocated when the handler returns. It is currently allocated on the heap but it is possible to allocate it in non-pageable memory to prevent page faults on access. The handler runs at the lowest processor level and normal interrupts and traps can occur while it runs. The handler can make system calls just like other functions in the program.

A crucial aspect that a programmer should be aware of is that an RTU is delivered to a process only if the process is in the runnable state. If a process activates one or more RTUs and gets blocked in the kernel before the RTUs have completed their task, then these RTUs will not be delivered until the process becomes runnable again. This is because if a process is sleeping in a system call, invoking the handler would cause the sleep to be interrupted prematurely and the process state to be overwritten when the handler returns. In order to allow a process to synchronize with its RTUs, the *rts_sleep* system call is provided (see Section 6). Note that page faults can occur while a handler is running because of which the process can be put to sleep. However, involuntary blocking of this sort is always temporary and our experiments have not found this to be a problem.

5.3 Schedulability Test

In order to guarantee that each RTU meets its deadline it is necessary to derive a schedulability test for the RMDP scheme. We have modified the basic result for the RM algorithm by incorporating the effect of delayed pre-emption. The basic result [15] states that a set of n periodic tasks scheduled by the RM algorithm always meet their deadlines if

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \le n\left(2^{1/n} - 1\right)$$

where C_i and T_i are the execution time and period of the i^{th} task respectively and where U is the total utilization. It turns out that the maximum utilization that can be supported in RMDP is less than that in the RM scheme because a higher priority RTU can experience *blocking* due to a lower priority RTU that is running. There are a lot of results that have analyzed the effect of blocking on schedulability. These results characterize the blocking effects due to OS mechanisms such as context switch time, interrupt overhead and so on[12]. However, they are unsuitable for our purpose because they view blocking as a system overhead and assume blocking times are more or less constant for a given system. A result that is of more relevance to our work is the one by Sha[20] et.al., who analyze the effect of the blocking that results when a lower priority task is in a non-preemptible critical section, and a higher priority task becomes ready to run. They provide a mechanism to bound the blocking time, and prove that using their mechanism a set of n periodic tasks can be scheduled by the RM algorithm if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + max\left(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \le n\left(2^{1/n} - 1\right)$$

in the process is invoked with the correct arguments, and the system state is restored after an RTU completes its work. Before we move on to describe the details of the system in Section 6, we present the RMDP scheduling policy which is the most crucial component of the implementation.

5. Overview of RMDP

The basic idea behind the RMDP scheme is that an RTU handler is allowed to do some minimum amount of work every time it is run. This work is measured in terms of number of PDUs processed. Therefore, in addition to the batch size B_p which is the maximum number of PDUs that the RTU can process each period, we associate a value b_p which is the number of PDUs that the RTU is allowed to process (i.e one iteration of the handler) without interruption from higher priority RTUs. Thus higher priority runnable RTUs have to wait until the running handler completes its current iteration. Since a handler completes at least one iteration every time it runs, the number of context switches is reduced. In addition, if variables shared with other RTUs are accessed only inside an iteration there is no need to lock them. In the next section we show how RMDP can be implemented using shared memory between each RTU and the scheduler.

5.1 The RMDP Scheme

The RMDP algorithm is shown in Figure 5. The data structure shown on the right is placed in a memory region that is shared by an RTU and the scheduler. Each time a running RTU handler generates/consumes a PDU, it increments the *pdu_count* field. It then checks if the number of PDUs processed so far i.e the value of *pdu_count*, equals the batch size B_p. If so, it yields the CPU since it has obtained its processing requirement for the current period. This gives control to the RMDP scheduler which schedules the next RTU (if any). If *pdu_count* is less than B_p, then the RTU handler goes on to process the next b_p PDUs (i.e the next iteration). Before going back however, it checks to see if the *yield_request* field in the shared memory structure is set. The *yield_request* field gets set in the kernel clock interrupt routine when an RTU of priority higher than the running RTU becomes runnable. If the handler notices this field to be set, it yields the processor and it is put back in the run queue. The count of PDUs that remain to be processed for the current period is stored in the shared memory (not shown), and when the handler is resumed it needs to process only these many PDUs. Other systems[11,17] have used shared memory to allow the user level scheduler and the kernel level scheduler to communicate, but their objectives and their algorithms are different.

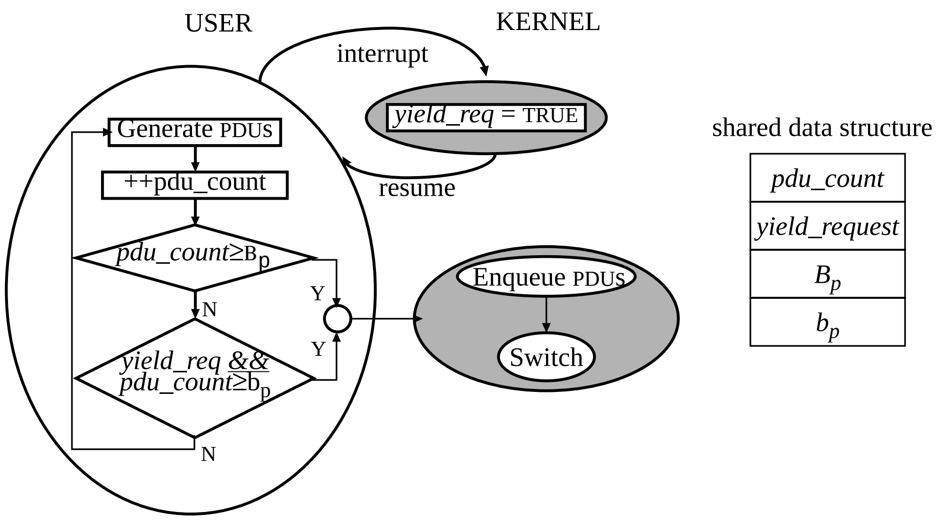

FIGURE 5. RMDP Scheme

cedure to execute the protocol code. An *upcall* is a well known mechanism[1] to structure layered protocol code. Protocol code in a user process can register an upcall with the kernel and associate a handler function with each upcall. The kernel can then arrange to have the handler invoked periodically. The *signal* mechanism is a somewhat similar mechanism that is provided by UNIX. For example, the combination of the *alarm* and *signal* facilities in UNIX can be used to setup a handler to be called periodically. One limitation with the existing facility is that only one such alarm can be setup per process. Another drawback is that there could be arbitrary delays before the alarm signal actually gets delivered and so real-time behavior cannot be ensured. The third major limitation is that the scheduler cannot guarantee that all signal handlers will get their requested amount of processing time. Finally, signals have traditionally been used only for exception handling and are inappropriate as a processing abstraction.

The approach we have taken is to provide an upcall facility with real-time support to structure protocol processing. A process can create multiple RTUs. Each RTU is delivered periodically in real-time, and there is an admission control operation to ensure that each RTU gets its requested time to run in each period. RTUs are less expensive to implement than real-time threads. An RTU runs on the program stack and the kernel needs to maintain very little state information per RTU. RTUs are local to a process and typically synchronization between RTUs in different processes is not necessary (nor supported) because they would be associated with different protocol sessions. Another feature of the RTU abstraction is that a running handler will not be pre-empted by a higher priority handler in an uncontrolled manner. Since protocol processing is iterative, we have implemented pre-emption as completion of the current iteration (which might be processing the current PDU) and returning from the handler routine. The obvious advantage of this scheme is that there is no need to save the activation record and register context of the handler across pre-emptions. The other advantage is that two (or more) RTUs in a process that share common variables need not lock them before access. Since simplicity often translates to greater efficiency, we believe RTUs are an optimal mechanism to organize protocol processing in user space with QoS guarantees. The overall organization of the RTU facility is shown in Figure 4.

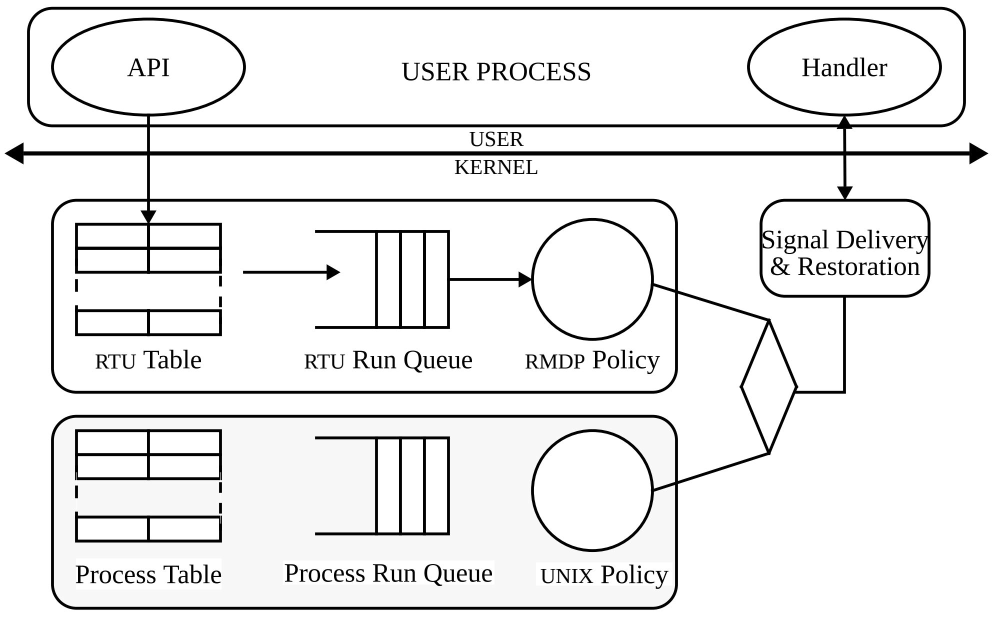

FIGURE 4. RTU Organization

The RTU facility is layered on top of the normal UNIX process scheduling mechanism. RTUs are placed in the run queue according to their specified periods, and they are scheduled according to a modified rate-monotonic scheme called RMDP (RM with delayed pre-emption). As can be seen we have not modified the basic UNIX scheduler or its policies in any way. Instead, the RMDP scheduler sits on top of the existing UNIX scheduler and makes the final scheduling decision. The policy determines which RTU handler (if any) should be run, and arranges for the process that contains it to be made the currently running process. The RTU delivery and restoration routines ensure that the handler threads. A periodic thread has a time period and a processing time associated with it. The scheduler invokes the thread at the start of every period. In each invocation, the thread function does the required processing, and returns. A thread can be pre-empted by higher priority threads while it is running.

The second aspect is the mechanisms within the kernel that support the user level abstractions. For each thread, the kernel maintains scheduling attributes such as period, computation time required, and the amount of time it has run. The scheduler could use one of several pre-emptive, priority based scheduling policies to schedule a thread. An example of such a policy is the *rate monotonic* (RM) scheme[15]. In the RM scheme, each thread has a priority that is proportional to its rate. Given two threads with different periods, the one with the smaller period (higher rate) has the higher priority. The priority is used to decide if a running thread must be pre-empted when another thread becomes ready to run. RM priorities are static because they only depend on the period. Other schemes such as *earliest deadline first*[15] are dynamic priority schemes and are more complex to implement. Before creating a new thread, the scheduler performs a *schedulability test* to determine if all threads will be able to meet their deadlines. This guarantees that once an thread is created, it will get its requested processing time.The kernel also implements the context switching mechanism to switch the CPU between different threads.

3.1 Limitations of RT-Mach

One of the first options we considered for implementing processing guarantees for protocols in NetBSD was the real-time thread approach. A closer examination however reveals several drawbacks. These are listed below.

- Implementation complexity: As mentioned before, systems such as Mach (Chorus and Solaris) layer the real-time support over a complex thread mechanism supported by the kernel. Since NetBSD does not provide kernel support for threads, we would have to port one of these thread facilities before real-time scheduling policies could be implemented. This is a non-trivial exercise. Thread *id*s have to be managed globally. Support must also be provided for threads in different domains to synchronize. However these features would hardly ever be used by a thread that just needs to do protocol processing but must be provided for reasons of compatibility and generality. We concluded therefore that this was an unnecessary overhead.
- Locking: The real-time nature of threads coupled with the fully pre-emptible scheduling mechanism used in most systems, introduces additional complexity to support locking of shared variables. For example *priority inheritance* is implemented in RT-Mach[4, 20] and in Solaris[13] to prevent unbounded priority inversions when a real-time thread locks[1] a shared resource. Locking introduces run-time overheads[13] because a system call is needed to setup state so that priority can be inherited later if required. In addition, two unnecessary context switches occur when a thread pre-empts a running low priority thread only to find that the lock is held. An example that motivates the need for locking is a protocol session that is supported by two threads, one of which sends data packets and the other processes acknowledgments. A shared variable that tracks the number of unacknowledged bytes would be modified by both threads, and must therefore be locked before access. We concluded that a fully pre-emptive scheduling mechanism was expensive to implement and to use.
- Context Switching Overhead: Context switching costs are known to dominate PDU processing costs[2]. Hence the number of context switches must be kept to a minimum. However a fully pre-emptive real-time scheduling policy can lead to excessive context switching compared to quantum based time sharing schemes thereby reducing effective utilization. We wanted a scheduling scheme that would allow the number of pre-emptions to be controlled.

The RTU mechanism has been designed keeping the above points in mind and is described next.

4. The Real-time Upcall (RTU) Approach

We have seen that processing requirements can be expressed using parameters that include a time period, a computation time requirement in each period (which can also be expressed as the number of PDUs to be processed), and a pro-

1. We assume that real-time threads would use blocking locks (or semaphores) rather than spin-locks.

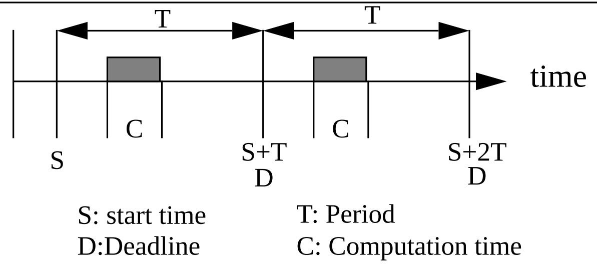

FIGURE 2. Periodic Thread/Signal Model

made for the data transfer phase of the protocol by an appropriate choice of operating system mechanisms for scheduling protocol processing.

It must be noted that the existing UNIX scheduling mechanism cannot meet the above requirements. For one thing, since multiple sessions with different periods can be active in a process a single priority per process is not enough for priority based schedulers. In the rest of this paper we explore the OS principles and mechanisms that are needed to support this requirement. We would again like to point out that this method is general and protocol processing is an example of its use. We begin with the description of how things are done in a representative OS such as RT-Mach.

3. QoS Support in RT-Mach

We identify two components in the implementation of OS support for periodic processing. The first is the mechanism used to structure the periodic processing activity within a process. The second is the mechanism used within the OS to schedule these activities in different processes. These two aspects are shown in Figure 3. Note that this organization is shared by other systems as well, such as Solaris[13] and Chorus[19] for example.

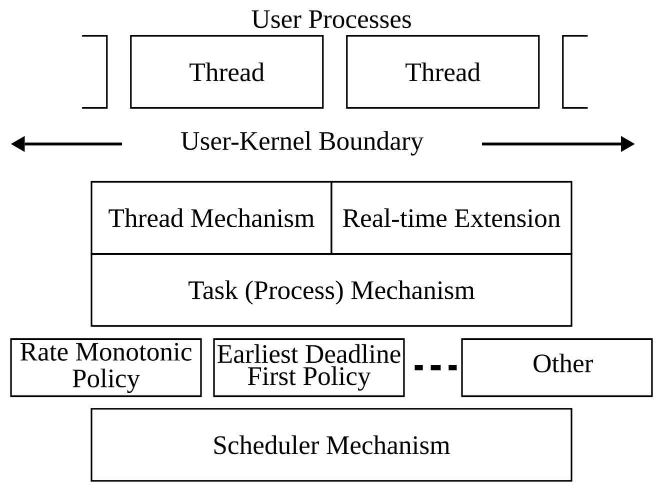

FIGURE 3. Existing OS Mechanisms

The first aspect is the abstractions provided by the OS to structure independently schedulable threads of control at the user level. In RT-Mach, this corresponds to the thread mechanism to structure the processing code. This corresponds to a light weight process in Solaris[13]. The thread abstraction provides isolation between several concurrent and asynchronous activities thus allowing a programmer to use a synchronous programming style for each. A thread is created with an entry point which is a function in the program. The OS scheduler invokes the function when a thread is run. A thread has an *id* that can be used to control it, and has its own activation record, stack area, message queues, priority, rights and so on. The thread model has been extended to do periodic processing by introducing the notion of *periodic*

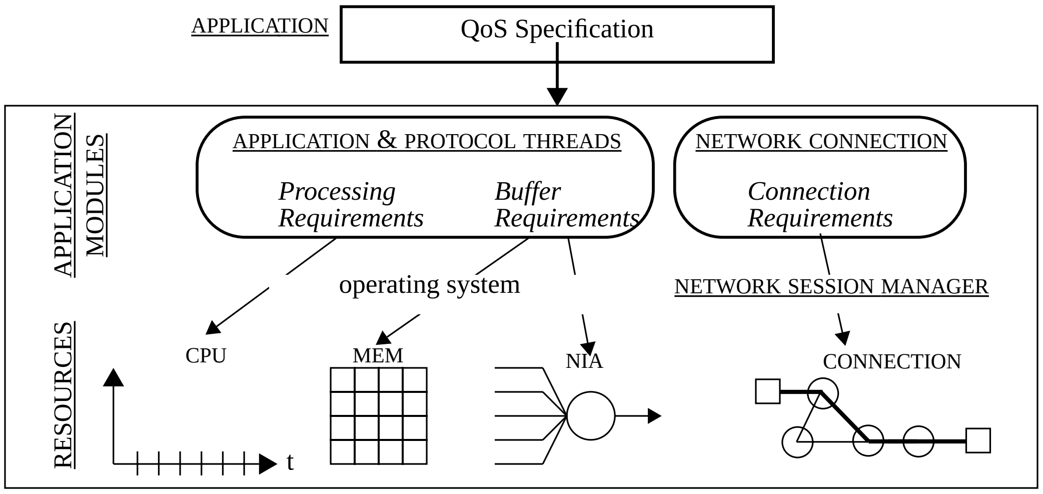

FIGURE 1. QoS Mapping

ing that largely determines QoS for a session executes in the process and it uses the real-time processing facility provided by the OS. The ALP model is well suited to native mode ATM implementations[21] where PDUs of each protocol session are carried on a private network connection with appropriate QoS parameters that fit the requirements of the session. Since ATM cells carry the connection identifier(VCI/VPI), the adaptor can demultiplex incoming data to the appropriate protocol session in the destination process. Likewise outgoing data can move directly between user and the VCI without OS intervention[6]. Thus the OS is involved mainly in scheduling the processing of network data in the different processes. The RTU approach for QoS guarantees can also be extended to protocols implemented in the kernel.

In the next section we look at how the QoS mapping operation determines processing requirements for a session.

2.1 Deriving Processing Requirements

We very briefly consider the QoS mapping operation for a continuous media stream such as compressed video. We interpret the delay parameter specified by the application as the maximum time that can elapse between generation of a frame at the source and its display at the receiver. The delay along with the estimate of compressed frame sizes can be used to derive the connection bandwidth. The bandwidth of the connection, and the size of a PDU determine the time taken by the adaptor to drain a PDU. In order to keep up with the adaptor, protocol processing at the sender must occur at a rate so that at least one PDU is generated and enqueued at the adaptor in every such duration. Equivalently, we could require that a batch of B_p PDUs be generated in every interval of length T that is B_p times the original duration. This technique is called *batching*, and is useful because if protocol sessions are active in different processes, it is expensive to switch between processes after each processed PDU. Other trade-offs involved in the choice of batch size are discussed in [8]. Depending on the platform and the amount of processing involved, the computation time C required to process B_p PDUs can be determined. Thus we arrive at a periodic processing model as shown in Figure 2. The period is T, and the requested computation time is C. The deadline D is the time at which the next period begins. The utilization of a session is equal to C/T and the sum of the utilizations of all the sessions is the total utilization. The periodic model has been used before in the real-time operating systems domain[23] to express processing requirements and in addition closely mirrors the periodic nature of continuous media data.

Therefore specifying two quantities namely, the size of a batch, and the period between batches characterizes the processing requirements for a protocol session. The same quantities hold for the receiving application process at the receiving host. For transfer of bulk data and continuous media streams that extend over durations of a few seconds or longer, this characterization is sufficient. For these sessions, the values of the batch size, the period, PDU size, and PDU processing time can be used to reserve CPU capacity during connection setup. Thus processing guarantees can be

Apart from the protocol code whose processing requirements which can be expressed as RTU parameters, if the processing requirements of application specific code can also be expressed in a similar fashion, then truly end-to-end guarantees can be made. This is indeed our final objective, and there are indications that other components of the operating system such as disk I/O subsystem for example are being designed so that parts of the application that do I/O can also obtain guaranteed real-time response. In the rest of the paper we discuss the provision of guarantees only for the protocol processing aspects of applications.

The outline of this paper is as follows. Section 2 describes the overall QoS framework that we have developed and shows how the processing requirements for a protocol session can be derived. Section 3 describes the thread based approach such as used by RT-Mach and similar systems, and identifies their limitations. Section 4 presents the RTU mechanism as an alternative to real-time threads for implementing network protocols in user space. Section 5 describes our scheduling scheme and presents its schedulability test. Section 6 contains important implementation details. Section 8 presents experimental results and we end with our conclusions.

2. The QoS Framework

We have developed a QoS framework[8] from the point of view of application processes that run on the endsystem. We identify four dimensions to the problem of providing QoS guarantees for protocol processing. These are:

1. *QoS Specification:* Specification of QoS requirements is essential to provide performance guarantees. Since applications can have widely varying requirements, a structured and general way to specify QoS is necessary. A typical approach is to identify a few application classes that can capture the needs of most applications, and to define parameters within each class using which users can specify their requirements. We have identified four application classes that encompass continuous media, bulk data, low bandwidth transaction messages, and high bandwidth message streams. It is important to note that applications can only specify high level parameters, and other low level parameters must be derived automatically.
2. *QoS Mapping:* QoS specifications referred to above are at the application level. Since several resources (such as CPU, memory, and network connections) are involved in communication, the specifications must be mapped to resource requirements. For example, network connection bandwidth must be determined from the specified QoS parameters while setting up the network connection. Likewise, the amount of processing required must be determined so that the CPU capacity can be allocated to ensure that protocol data units (PDUs) are processed at the desired rate. The operation of deriving resource requirements from QoS specifications is referred to as *QoS mapping*, and is illustrated in Figure 1. From the specification for a data stream, the mapping operation derives network connection attributes such as bandwidth and cell delay, pacing parameters to determine the rate at which the network interface must transmit cells, processing attributes for the protocol code, and memory requirements. We have worked out details of the mapping operation for the resources mentioned above[8].
3. *QoS Enforcement:* The mapping operation as mentioned above derives resource requirements, that are allocated by the operating system to each application during the setup phase. During the data transfer phase, the operating system implements the *Qos enforcement* function, which involves scheduling various shared resources to satisfy these allocations. In particular, the CPU scheduling policy of the operating system largely determines how the aggregate processing capacity of the endsystem is shared between different network sessions and is therefore crucial in determining the QoS provided. We therefore focus on CPU scheduling mechanisms that can be implemented efficiently, and integrated into protocol code.
4. *Protocol Implementation Model:* Given these solutions for QoS specification, mapping and enforcement, protocol code has to be structured to take advantage of these facilities. The protocol implementation model facilitates the mapping of protocol services to appropriate implementation components provided in the framework. An important objective of the model is to provide efficient OS mechanisms to improve the efficiency of protocol implementations by reducing the overhead associated with data movement and context-switching operations that dominate protocol processing costs[2].

   Since RTU handlers run in user space, we use the application (or user) level protocol (ALP) model[16, 22]. In an ALP, boundary operations such as connection setup and release are implemented in the kernel because they involve setting up of resources that are shared among all application processes. The data transfer protocol process-

- QoS Mapping: Deriving various resource requirements from the QoS specification
- QoS Enforcement: Scheduling mechanisms within the endsystem to ensure processing requirements of an application are met
- Protocol Implementation: Implementing protocols using the components provided in the framework

This paper reports on a QoS enforcement facility that we have designed, analyzed, implemented, and experimented with. It is called the RTU (real-time upcall) facility and has been integrated into the NetBSD operating system. A fairly well known representative of similar work is RT-Mach[23] and Solaris[[13]. RT-Mach provides real-time periodic threads that can be scheduled according to several real-time scheduling policies. However, this approach relies on a kernel level thread facility which is not widely available. Moreover the full generality of real-time threads is not necessary for protocol implementations, and they can benefit from a more specialized mechanism tailored to their special needs. The RTU approach is based on the following claims—

- We have observed that the processing that is needed for implementing network protocols can just as easily be structured as upcalls in user programs. For example, the code that implements the output and input routines of protocols in the NetBSD kernel resemble upcalls to a large extent, and may be invoked as a software interrupt. Upcall facilities are simple to implement and are more readily available compared to a kernel level thread facility.
- Protocol processing has the characteristic that it is repetitive in nature especially for bulk data transfer and continuous media applications, and we have used this to our advantage. For example, given a large chunk of data to be sent, the protocol processes it in terms of protocol data units (PDUs) usually of some fixed size. For each PDU it executes the same code. This make it easy to express the processing requirements in terms of the number of PDUs to be processed within a given time period, and the procedure (upcall handler) that iterates over the PDUs to perform protocol processing. While we intend to use this facility mainly for implementing network protocols, it is general enough to support any processing activity that is repetitive in nature. For example it could be in used in the image processing domain in which a sequence of images are generated, and for each image the same code is run on its smaller chunks.
- We have obtained significant efficiency benefits by viewing protocol processing as iterative. For example in a fully pre-emptive scheduling scheme such as implemented in RT-Mach and Solaris, a thread can be pre-empted at any point in execution. This requires threads to lock shared variables. Locking mechanisms for real-time threads must prevent unbounded priority inversions, and have to be implemented in the kernel. Thus real-time threads have to make system calls to manipulate locks leading to considerable overhead. However we have exploited the iterative nature of protocol processing to implement pre-emption by allowing a RTU to complete its current iteration (i.e processing the current PDU) and then switching to the higher priority RTU. This scheme eliminates the locking operations (system calls) and avoids situations in which a thread is switched in only to yield immediately because the lock is held by a lower priority thread. Apart from these two efficiency benefits, it saves on system complexity, and frees the protocol programmer from having to manipulate locks.
- Despite our departure from a fully pre-emptible scheme, we can closely approximate a number of well known pre-emptive real-time scheduling algorithms whose properties are well understood and have been established analytically. Furthermore, no changes were required to the existing process scheduling algorithm and the priority structure supported by the kernel. Thus multiple RTUs in a process, each with its own priority are managed independent of the process priority itself. This feature makes the RTU facility highly portable compared to real-time threads in the kernel.

Our experience with the RTU facility shows that it is effective even with background loads. While a UDP client and server programmed using the process abstraction suffer upto 90% reduction in throughput under background load, no measurable degradation was observed when the same functionality was implemented with RTUs. Furthermore all the handler invocations occur periodically and are completed before their deadlines despite the fact that page faults do occur within an RTU handler. We have not found the need to completely eliminate faults mainly because a typical handler has a small code size and therefore exhibits spatial locality. Also since it is written to execute in an iterative fashion, it also has temporal locality. The RTU scheduling scheme also reduces context switching by as much as 30% compared to a fully pre-emptive scheduler. This results in upto a 12% increase in useful utilization. With the elimination of locking and context switches due to lock unavailability, we expect even more savings.

Real-time Upcalls: A Mechanism to Provide Real-time Processing Guarantees

Abstract

Real-time upcalls (RTUs) are an operating systems mechanism that can be used by applications to efficiently schedule code segments (or *handlers*) that must execute periodically. While the mechanism was conceived to support protocol processing with quality-of-service guarantees for networked multimedia applications it is general enough to be applicable in other domains like real-time image processing. Until now real-time threads have been the only mechanism for implementing protocols in user space with QoS guarantees. The RTU mechanism avoids the implementation complexity of the thread based approach while retaining its ability to ensure real-time behavior. In addition, our design simplifies protocol code, improves performance, and can be ported to most systems. A key feature of RTU scheduling is the pre-emption scheme that exploits the iterative nature of protocol processing so that an RTU yields the CPU by returning from its handler. This obviates the need for RTU handlers to lock shared variables thereby eliminating the system calls needed for real-time locking, and the context switches that result when an RTU is run only to yield at once due to a lock being held by a lower priority RTU. Our new schedulability test for RTU scheduling improves upon existing results. Our implementation did not require changing the existing process scheduling mechanism, and has been ported to Sparc-1 and Pentium CPUs in the NetBSD kernel. Our experimental results with the UDP protocol show that the RTU facility provides QoS guarantees under all background processing loads. It also improves the useful utilization by 12% by reducing context switches by upto 35%.

1. Introduction

There is a growing need to provide support for multimedia processing within computer operating systems thereby enabling a variety of exciting applications such as interactive video, customized news services, virtual shopping malls and many more. A large fraction of data handled by these applications will be of the continuous media(CM) type. The transfer of CM data over the network, and its processing at the endsystem, must be in such a way that its periodic nature is preserved. This requirement is also referred to as the need to provide “real-time” guarantees for data transfer and processing. Emerging networks such as ATM, and the proposed integrated services internet[3] with reservation schemes such as RSVP[24], can provide guarantees for data transfer by managing network resources appropriately. Similarly the operating system (OS) must manage endsystem resources so that processing needs implied by the bandwidth and delay requirements of each connection are satisfied. This will complement the guarantees provided by the network for data transfer and build upon the increasing processing power of the endsystem hardware.

Providing processing guarantees for multimedia data at the endsystem is important, and for a good reason. Without these guarantees one can never be sure that two or more applications will not interact in an adverse manner. The situation would be somewhat similar to going to a ballgame in which there is no guarantee that only as many tickets as there are seats have been sold. In addition to the aspect of providing *guarantees*, the scheduling mechanism must ensure *timeliness*, which means that the data units of each connection must be presented at the right instant in real-time after the necessary processing. For multimedia applications to become ubiquitous these two aspects must be enforced so that users are assured that the operating system will preserve the periodic and continuous nature of the data streams over arbitrarily long durations no matter what other processing activity occurs during this time. In the networking domain, this is referred to as the need to provide *quality-of-service* (or QoS) guarantees. Until now researchers have been concentrating on how to provide QoS guarantees within the network. We have developed a QoS framework for the endsystem. Important aspects of this framework are–

- QoS Specification: Ways to allow an application to specify the QoS it requires for each data stream

Real-time Upcalls: A Mechanism to Provide Real-time Processing Guarantees

Raman Gopalakrishna

Guru M. Parulkar

WUCS-95-06

September 21, 1995

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St.Louis, MO 63130-4899

Abstract

Real-time upcalls (RTUs) are an operating systems mechanism that can be used by applications to efficiently schedule code segments (or *handlers*) that must execute periodically. While the mechanism was conceived to support protocol processing with quality-of-service guarantees for networked multimedia applications it is general enough to be applicable in other domains like real-time image processing. Until now real-time threads have been the only mechanism for implementing protocols in user space with QoS guarantees. The RTU mechanism avoids the implementation complexity of the thread based approach while retaining its ability to ensure real-time behavior. In addition, our design simplifies protocol code, improves performance, and can be ported to most systems. A key feature of RTU scheduling is the pre-emption scheme that exploits the iterative nature of protocol processing by allowing an RTU to yield the CPU by returning from the invocation. This obviates the need for RTU handlers to lock shared variables thereby eliminating system calls needed for real-time locking, and reducing unnecessary context switches. Our schedulability test for RTU scheduling improves upon existing results. Our implementation did not require changing the existing process scheduling mechanism, and has been ported to Sparc-1 and Pentium CPUs in the NetBSD kernel. Our experimental results with the UDP protocol show that the RTU facility provides QoS guarantees under all background processing loads. It also improves the useful utilization by 12% by reducing context switches by upto 35%.