Mach Interface Proposals - Priorities, Handoff, Wiring

DRAFT of 13 August 1989

David L. Black
Carnegie Mellon University

Introduction

This proposal involves three sets of changes:

1. **Priority** - Export priorities on individual threads to users. Introduce notion of scheduling policies, and support for fixed-priority threads.
2. **Handoff -** Handoff scheduling (specify thread to run next) and related mechanisms.
3. **Wiring -** Lock data and threads into memory to prevent paging and swapping.

Priority

There are three goals behind this set of proposed changes:

1. Export priorities for individual threads to users.
2. Support fixed priorities for real-time and other uses (e.g. OLTP)
3. Produce a clean interface for users as well as kernel clients.

Under this new proposal threads have both a priority and a max priority (like memory protections); priority cannot exceed max priority, but the max priority can be reset by presenting the appropriate processor_set object port. Since the default processor set's object port is privileged, ordinary users who do not do their own processor allocation cannot raise thread priorities above their initial maximum. Initial priority is inherited from the task at creation (of both threads and tasks), and the initial max priority is inherited from the processor set at thread creation.

Scheduling policy can be set on a per thread basis; for now there are two policies, time sharing and fixed priority. For fixed priority, a quantum can be specified (again on a per thread basis); this is the quantum that the thread will receive before being eligible for preemption at the same priority. A processor set may forbid scheduling policies other than time sharing.

The ranges of priority will be changed to correspond more closely to scheduler internals, but Unix interfaces will continue to deal with Unix priorities. Two notions of priority will be supported; a base priority and a scheduled priority. The base priority ranges from 0 to 31 and is the priority assigned to the thread by the user or user-mode scheduler. The scheduled priority also has the same range; it is identical to the base priority for fixed priority threads, but for timesharing threads it is the base priority plus some increment derived from usage. In both cases low numbers are the highest priorities; changing this would just create confusion[1]. The traditional Unix priority (range 0-127) can be obtained from these numbers (range 0-31) by multiplying by 4. Additional levels of fixed priorities can be added by using negative priorities; some real-time investigators believe this is necessary to do a realistic deadline scheduler, but this is a topic for future work.

---

[1]Although we can still be convinced otherwise if enough people protest loudly enough

Preemption normally occurs when a thread becomes runnable at a higher priority than a currently running thread when there are no idle processors in the processor set. It may take as long as a (machine-dependent) clock interrupt period for a multiprocessor kernel to notice that preemption is needed (interprocessor interrupts are not used for preemption events). There is one major exception to this rule that delays preemption of time-sharing threads. The reason for this is that time-sharing threads have their priority recalculated as they run; if these new priorities were always put into effect immediately, the result would be additional (unneeded) context switches. To avoid this, running timesharing threads usually delay preemption due to this priority recalculation until the end of their current quantum (typically 1/10 of a second).

Note: in call specifications, a processor_set argument refers to the processor set's control port; the name port corresponds to a processor_set_name argument. The following new calls are proposed:

- kern_return_t **thread_priority**(thread, priority, set_max) - Set thread's priority. The priority must be no higher (lower number) than max. set_max argument is a boolean; the thread's maximum priority is reduced to the priority argument if set_max is TRUE.
- kern_return_t **thread_max_priority**(thread, processor_set, max_priority) - Change the maximum priority of a thread. The processor_set port is the control port, not the name port. This call fails if the thread and processor_set do not correspond.
- kern_return_t **thread_policy**(thread, policy, data) - Set thread's scheduling policy. The policy is either POLICY_TIMESHARE or POLICY_FIXEDPRI. The data argument is a policy specific int. For timesharing, data is ignored. For fixed priority data is an integer quantum in milliseconds.
- kern_return_t **task_priority**(task, priority, change_threads) - Set task priority, given to newly created threads. Also change thread priorities if change_threads is TRUE. In the latter case as many priorities as possible will be changed, but a failure code will be returned if any thread's max priority is violated (any such threads are unaffected).
- kern_return_t **processor_set_priority**(processor_set, max_priority, change_threads) - The max_priority argument sets the processor set's maximum priority; this applies to newly created or assigned threads. If the change_threads argument is true, any restrictions caused by this new maximum priority are applied to threads currently in the set.
- kern_return_t **processor_set_policy_enable**(processor_set, policy) - Indicate that the specified policy is permitted on the specified processor set. The policy must be a valid policy.
- kern_return_t **processor_set_policy_disable**(processor_set, policy, change_threads) - Indicate that the specified policy is not permitted on the processor set. The policy must be a valid policy. The change_threads argument indicates whether to apply the operation to threads currently in the set; if so, any threads with the forbidden policy are reset to time sharing. The timesharing policy may not be forbidden.

The policy for threads defaults to time sharing in the absence of explicit specification. Similarly, processor sets default to only allowing time sharing.

I/O drivers may prioritize requests placed by fixed priority threads according to the priorities of the threads; this behavior is machine-dependent. Any such prioritizing applies only to I/O explicitly requested by a fixed priority thread; implicit I/O (e.g. that requested by an external memory manager on behalf of a thread) will only be prioritized according to the thread that actually requested it (e.g. the thread in the external memory manager). As a result, systems that care about the priority of implicitly requested I/O should provide their own external memory managers to perform that I/O. The notion of priority may not be meaningful to some potential future policies (e.g. round-robin).

Timesharing may not be forbidden to simplify the semantics and implementation of **assign** operations. A thread whose policy is forbidden by the target processor set is still assigned successfully, but its policy is reset to timesharing; the policy for newly created threads similarly defaults to timesharing. This avoids the need to introduce a notion of default policy for processor sets and the additional logic needed to make sure the default policy is permitted and at least one policy is permitted at all times. Timesharing can still be avoided on user-created and managed processor sets by resetting the thread's policy after creation or assignment ; a real-time scheduler will surely want to do this in any case.

When a thread is assigned to a processor set, its priorities (both actual and max) will be reduced if either is above the target set's maximum priority. In addition its scheduling policy will be reset to time sharing if the target processor set does not permit its current policy. Default values of priority and max priority are 12 for historical reasons.

processor_set_priority may be used as a boot-time mechanism to clear out high-level priorities for a real-time subsystem by indicating the max_priority for time sharing threads and setting change_threads to TRUE.

Handoff Scheduling

Handoff scheduling refers to the technique of transferring the processor from one thread to another with as little operating system involvement as possible. The kernel already uses this technique for message operations; this section describes primitives that make it and related operations available to the user. These allow the user to take advantage of knowledge about which threads should or should not be run to influence operating system scheduling decisions without requiring the user to write a complete scheduler.

The proposed calls are independent of the above priority manipulation calls; they are motivated by situations encountered in doing locking on multiprogrammed systems. The first two calls are traps that operate on the thread that invokes them.

- kern_return_t **thread_switch**(new_thread, option, option_time) - Context switch to new_thread if possible. The new thread argument is a hint; it can be set to THREAD_NULL to just cause a context switch. Even if a hint is supplied but the new thread is not runnable or is already running, the call does not fail. Two options are supported:
    1. **WAIT** - option_time is a time period during which the current thread should be suspended internally. An internal mechanism will be used that cannot be released by thread_resume; thread_abort must be used to abort this wait.
    2. **DEPRESS** - option_time is a time_period during which the current thread's priority should be reduced to the lowest possible. suspend it due to coarse timing granularity.
- void **thread_restore_priority**() - Prematurely cancel effect of DEPRESS option to thread_switch().
- kern_return_t **thread_depress_abort**(thread). Identical to thread_restore_priority, but is an rpc to the kernel that operates on any thread.

thread_switch is designed to handle a variety of situations that occur in multi-threaded programming.

The specification of the next thread to run is designed to handle situations in which the user is keeping explicit or implicit scheduling information that can be taken advantage of by the kernel. An example of explicit information is tagging locks with the identity of the thread that holds them (this is easy to do if the hardware supports an atomic compare and swap if 0 operation). An example of implicit information is that in a functionally partitioned program with one thread per component, knowing the component that is being waited for immediately identified the thread that should run.

The **WAIT** option is designed for situations in which there is a significant minimum time for which the thread should be blocked.

The **DEPRESS** option handles all other situations. Experience with parallel applications at CMU indicates that a primitive that only context switches (and implicitly surrenders the remainder of the quantum) is not sufficient; two threads performing this operation can hog the processor to the exclusion of threads that should run. It is necessary to severely depress the priority of the threads that should not be run so that the other threads will be run even if their priority is below that of the waiting threads due to actions of the timesharing scheduler. The advantage of using depression over blocking is that an explicit unblock operation is not needed to resume the thread before the timeout expires. Implementors should be warned that large number of threads spinning through thread_switch or thread_switch with the DEPRESS option can effectively thrash a multiprocessor scheduler.

Interaction of quantum assignment and thread_switch: The current quantum is transferred to the new thread only if both thread and new_thread are time-sharing threads. If either or both are fixed-priority threads, the new_thread gets a new quantum.

Fixed priority threads do not get special treatment for any of these calls. This means that thread_switch will run the new thread regardless of its priority, and can be used to implement the implicit priority elevation required when a critical section is accessed by both low and high priority threads. Similarly, the calls involving depression operate identically on both fixed priority and time sharing threads.

The DEPRESS option is motivated in part by the lack of fine grain timing on many machines. If cheap fine-grain timing was available, the WAIT option might be adequate by itself. Extensive experience at CMU indicates that one of the two is necessary in a multi-threaded program where the user often does not know which thread is to run next.

Wiring

This section involves two calls to make resources non-pageable primarily for real-time experimentation.

kern_return_t **vm_wire**(host_priv, task, start, end, prot) - Wire down (i.e. make non-pageable) an address range in task’s map.

The prot argument indicates the types of memory access that must not fail so VM_PROT_NONE unwires. Implicit protection check is included. The system has an overall limit on wired down memory; this call fails if the requested wiring would exceed that limit. The limit and amount remaining will be available via a new call (vm_wire_statistics) because the existing vm_statistics kernel call is not extensible. This limit applies to address space discounted by explicit sharing. For the purposes of vm_wire, "explicit sharing" means:

- Sharing via a sharing map. (e.g. VM_INHERIT_SHARE) or
- Sharing by use of the same memory object.

[IMPLEMENTATION NOTE: The initial implementation will not discount for any sharing, so the limit will be an address space limit.]

There may be additional resources managed by pmap modules that limit wiring of memory. This is an issue to be confined to the pmap modules; such modules may export an interface to perform other wiring operations (e.g. wire down a thread context).

kern_return_t **thread_wire**(host_priv, thread, wired) - Make thread’s internal kernel state non-pageable so that thread will always be able to run immediately when run. This will consume some number of pages from the wired-down memory limit.