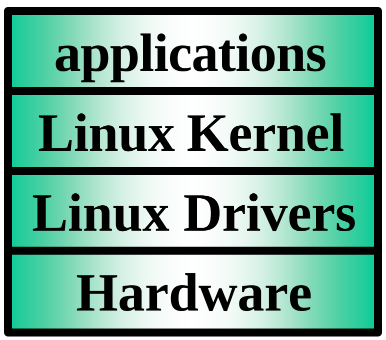
LINUX Kernel Preemption

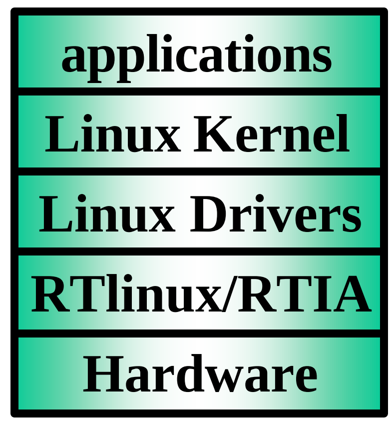
George Anzinger
Member Technical Staff
MontaVista Software, Inc.

Prior Art

MontaVista Software

- Other efforts to improve Linux latency
  - RTAI
  - Rtlinux



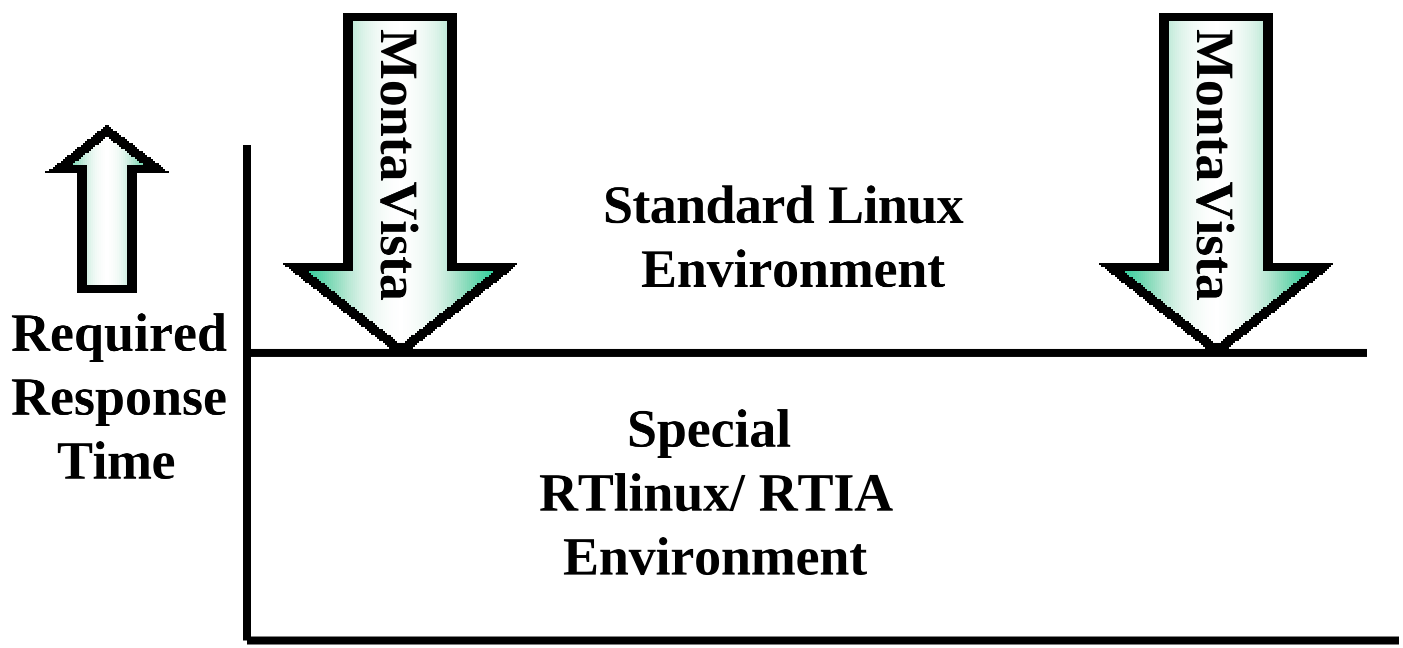


Prior Art

- Cost
  - Latency to services
  - Complexity
    - Hard to design
    - Non standard access to OS services
    - Hard to debug

Goals

- MontaVista software wants to move the standard Linux response times down



Goals

- To put numbers on the goal, we would like to see preemption times of:

  ~30 dispatch times,

  About 250 microseconds on a

  4-500mhz ix86 processor.

Assumptions

- The application is coded for RT:
  - Pre-allocates and locks down memory.
  - Allocates and controls task priorities using standard system primitives.

Problems

- Long or indeterminate algorithms used on the schedule path.

- Long preemption latencies.

- Too much work at the driver level.

Outline of the Talk

- We will examine the run list management and the time slice computation code.
  - We will take a look at MontaVista's solutions.
- We will take a brief look at the history of Linux to see how we got here.
- We will discuss the various problems and MontaVista's solutions.
  - Spinlocks.
  - Interrupts.
  - Traps.
  - Task state.

Outline Continued

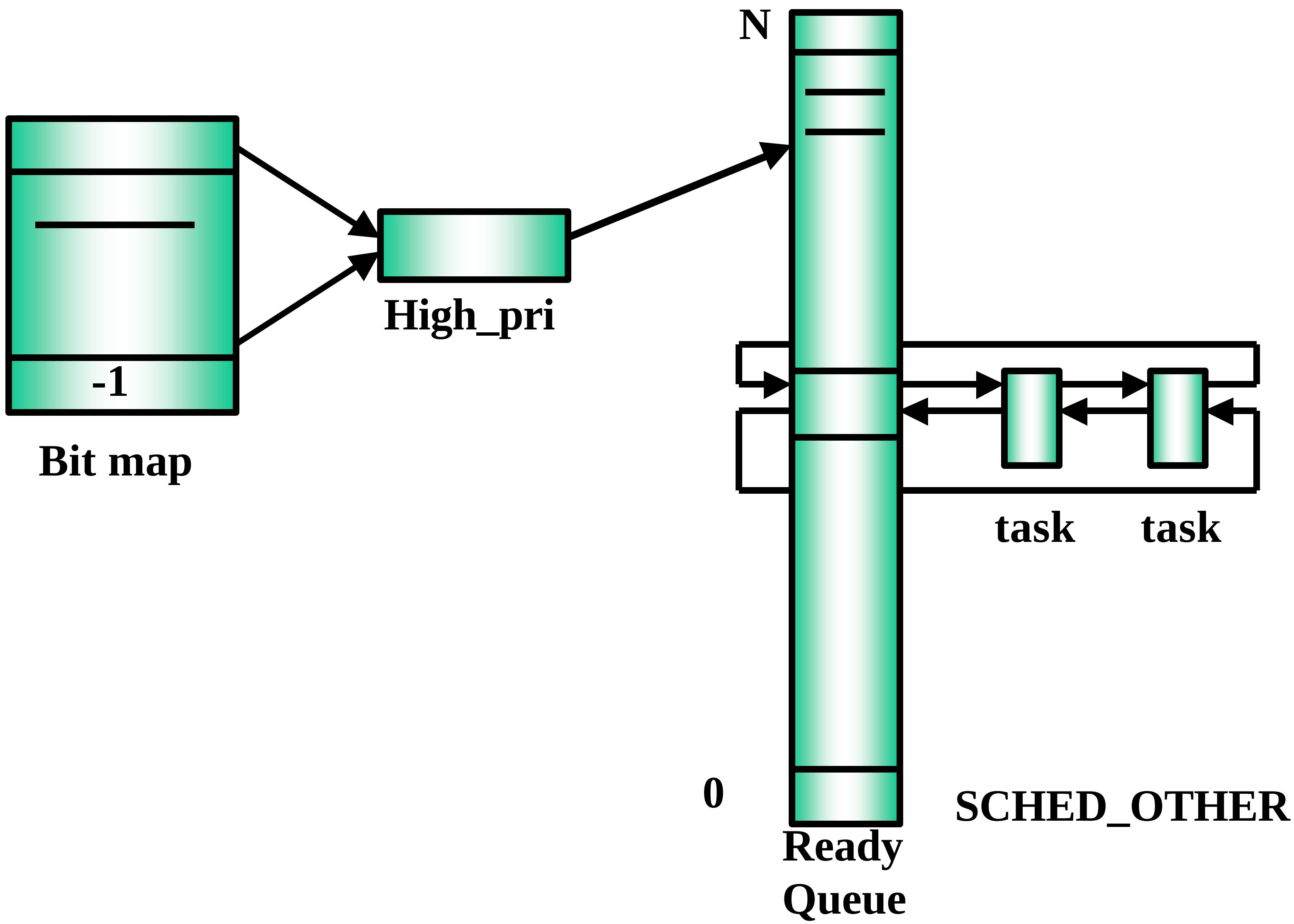
- MontaVista's solutions:
  - Preemption count.
  - Spinlocks -> pi mutex.
    - Just what the heck is a pi mutex anyway?
  - Big kernel lock.
    - What it is.
    - How we address it.
- Finally, we will look at:
  - Where we are.
  - What the results are to this point.

The Real Time Scheduler

MONTAVISTA SOFTWARE

- Problems with scheduler:
  - Each call, it scans a run list containing all ready or executing tasks.
    - Looks for:
      - Real time, highest priority task.
      - Non real time, highest slice (with fuzz).
        - » What is fuzz?
- All that just takes too long.

Scheduler Solutions



Scheduler Ready Queue

MontaVista Software

- Each priority has its own list.
- SCHED_OTHER is priority 0.
- High_pri is updated when a higher priority task comes into the ready queue.
- The bit map keeps track of which priorities have tasks on them.
- The bit map is only searched when high_pri is found to point at an empty list.
- The –1 is a stop bit. It stops the bit map search at priority –1.

Linux Time Slicing

- Unix uses nice() to adjust task priority.
  - Nice() takes values from –20 (high priority) to 19 (low priority).
  - Linux uses this to define a slice of time:

```
p->counter = (p->counter >>1) +
                    NICE_TO_TICKS(p->nice);
```

  - This value declines as p->counter-- each jiffy that the task is found executing.
  - When zero the task is preempted and will not be reschedule until it gets another slice.

Linux Time Slicing

Recalculating the Slices

- Recalculation is done when all slices in the run list are zero, -- nothing to run.
- At this time a new slice is calculated for ALL TASKS ON THE SYSTEM.
- Results in dormant tasks accumulating up to twice their “normal” slice value.
  - Tops out at about 6 recalculates.
  - Means waking task will likely get the processor.
- Problems:
  - Doing all tasks on the system takes too long.
  - Making two passes thru the run list and one thru all tasks takes WAY too long.

Squeezing Time Out of Recalculate

- First, why:
  - When recalculating the scheduler can not dispatch ANY thing. ⇨ Long schedule latencies.
- How:
  - Keep track of total of slice time available in the ready queue.
    - Requires that running tasks not be in the ready queue.
  - If zero time in ready queue, do one pass to update and select a new task.
  - Don’t update other tasks until they come on the ready queue.

For Inquiring Minds

```
if ((diff = runq.recalc - p->counter_recalc) != 0) {
        p->counter_recalc = runq.recalc;
        c = NICE_TO_TICKS(p->nice) << 1;
        p->counter = diff > 8 ? c - 1 :  /* max priority */
                            c + ((p->counter - c) >> diff);
}
```

runq.recalc is a system wide global, number of recalcs done

counter_recalc is number of recalcs done for this task

Scheduler SMP Issues

- Current scheduler keeps a list of CPUs and their tasks.
- RT scheduler adds:
  - Priority.
  - Makes this a doubly linked list by priority.
- Why:
  - When a new task comes on the RQ need only test against one CPU.
    - The one doing the lowest priority thing.

Preemption How We Got Here

- The first Linux system ran all system services to completion or till they blocked.
- Next it was expanded to SMP.
  - A lock was put on the kernel code to prevent more than one CPU at a time in the kernel.
- Over time finer grained locking has been used more and more.
- It seems the kernel lock is on the way out and may well be gone by 2.6.
- The kernel is still not preemptable except for selected explicit calls to schedule().

Preemption
How We Are Doing It

- The kernel has SMP locking protecting all the needed areas.
- Expand the SMP lock macros to indicate areas that must not be preempted.
- In addition, protect:
  - All interrupt code.
  - All trap code.
  - The scheduler itself.

Preemption Counter

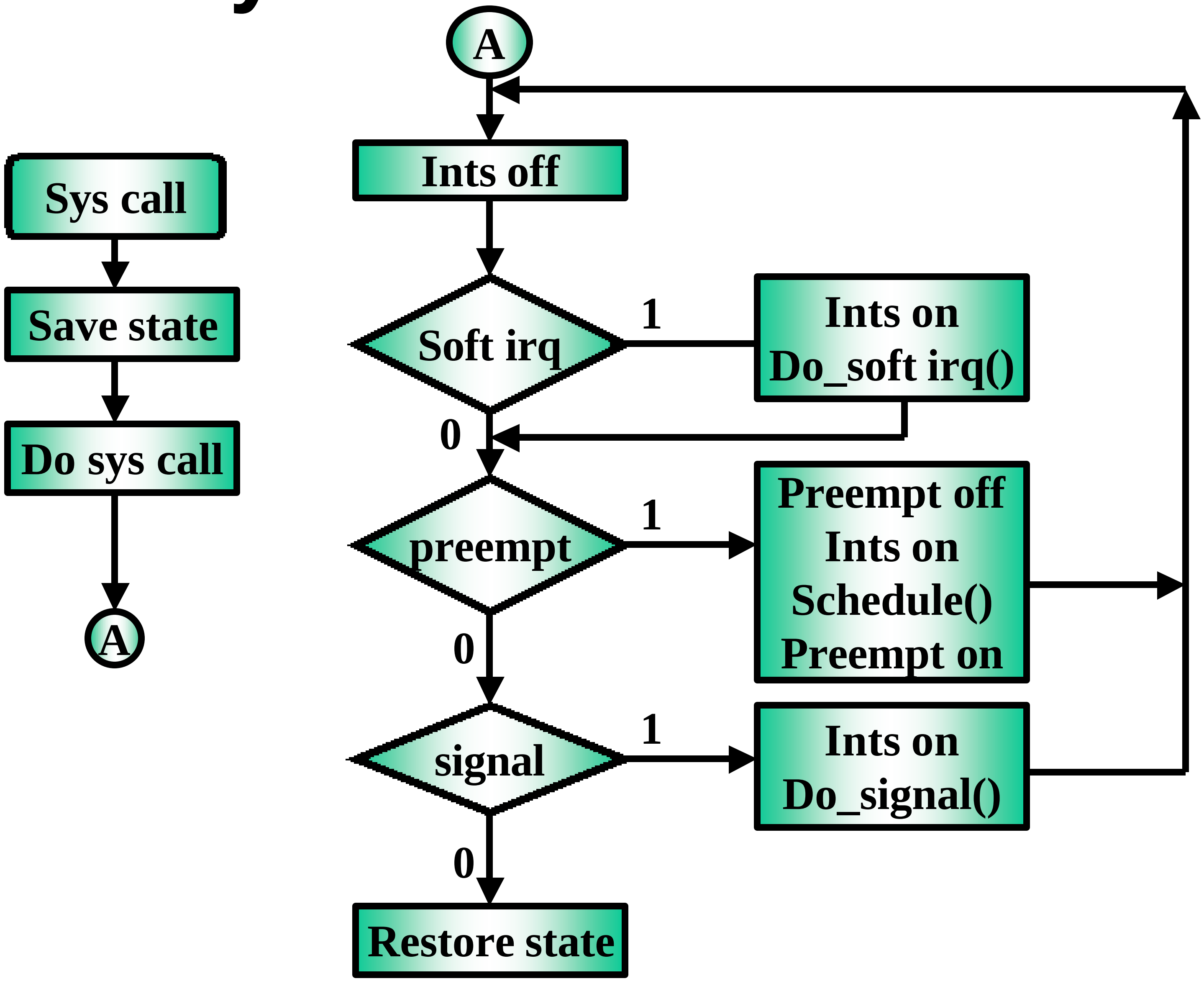
- All of that is a lot of stuff to test to see if it is ok to preempt so:
  - We created a preemption counter to count the reasons why we could not preempt.
  - Inc the counter on a condition,
  - Dec the counter when the condition clears.
- Test is reduced to:

If (!preempt_count && need_resched)
preempt_schedule();

Changes to Entry.S

- Manage the preemption count on trap entry and trap and interrupt exit.
- Insure that the exit sequence will not miss an allowed preemption.

Entry.S Flow Chart



Run State Issue

- How Linux sleeps a task.
  - Set the task state to something other than running.
  - Set up the wake up.
  - Call schedule().
- Schedule() will take a task that is not in the run state out of the run list.
- Consider preemption after the state change but before the wake up set up.

Run State Solution

- For preemption we call preempt_schedule().
- In preempt_schedule() we:

```
current->state |= TASK_PREEMPTING;
schedule();
```

- In schedule() we treat a task with the TASK_PREEMPTING flag as running.
  – It stays in the run list.
  – Gets rescheduled just like a running task.
  – Schedule() clears the flag.

Priority Inversion Explained

Given mutex X and 3 tasks:

- L at low priority.
- M at medium priority.
- H at high priority.

Suppose L locks mutex X.

L is preempted by M, a long running task.

M is then preempted by H which wants X.

M is running instead of H!

The priority is inverted!

The Pi Mutex Solution

- Pi stands for priority inherit.
- A pi mutex gives the owner of a mutex the priority of the highest priority waiter.
- The priority reverts when the owner exits.
- In our example, L would run at H’s priority until L released the mutex.

Pi Mutex Structures

```
struct pi_mutex {
    volatile struct task_struct *owner;
    struct list_head wait_list;
    struct list_head owner_list;
};

struct __pi_mutex_wait_queue {
    struct task_struct *task;
    struct pi_mutex *mutex;
    struct list_head task_list;
};
```

Pi Mutex Structures

MONTAVISTA SOFTWARE

- In addition the task_struct is modified:
  - A: int effprio; /*Effective priority of task */.
  - B: struct __pi_mutex_wait_queue *pi_mutex;
  - C: struct list_head owned_pi_mutex;
- The owned_pi_mutex list is a list of all the contested pi_mutexs held by the task.
- The wait list, headed at the pi_mutex, is a list of all waiters for the mutex.
- Both lists are in priority order.
  - The priority of a mutex is the priority of the highest priority waiter.

Managing Priority Lists

MONTAVISTA SOFTWARE

- For the RT scheduler, we have one or two (if SMP) priority lists a task can be in.
- Adding the pi_mutex, adds another priority list.
- Each of these lists needs a management function to change priority.
- Problem:
  - How do we know which list, if any, a task is in so we know what function to call.

Priority List Solutions

- A location in the task structure holds a pointer to the function.
  - This pointer is set when the task is put in a priority queue.
- A priority change for one task implies that other tasks may also need to change.
  - This implies recursion in the priority change function.
- So all priority change functions are called under a spin lock that must be recursive.

For Inquiring Minds

```
static struct task_struct *newprio_inuse;
static int newprio_inuse_count;

void set_newprio(struct task_struct * tptr, int newprio)
{
        if ( newprio_inuse != current){
            spin_lock_irq(&runqueue_lock);
            newprio_inuse = current;
        }
        newprio_inuse_count++;
```

For Inquiring Minds - Continued

```
if (! tptr->newprio ) {
        tptr->effprio = newprio;
    }else if ( tptr->effprio != newprio) {
        tptr->newprio(tptr,newprio);
    }
    if ( ! --newprio_inuse_count ){
        newprio_inuse = 0;
        spin_unlock_irq(&runqueue_lock);
    }
}
```

Pi Mutex Overhead

- The entry code depends on (in ix86 land) the cmpxchg instruction:

```
int cmpxchg(int *x,int new, int old)
{
        if (*x == old) {
                *x = new;
                return old;
        }
        return *x;
}
```

Where We Use Pi Mutex

- We are replacing long held spin locks with the pi mutex.
- If preemption is not configured, the code will generate the standard spin lock.
- SPINTEX is the name we are currently using.
- Short held spin locks will use either interrupt off or preempt count protection.
  - Type of protection to be chosen by architecture.

The Pi Mutex Entry

- Entry is:

```
if (old = cmpxchg(&mutex,current,0))
            pi_mutex_entry_failed(&x,old);
```

- Exit is:

```
if ( ! cmpxchg(&mutex,0,current) == current)
            pi_mutex_exit_failed(&x);
```

Pi Mutex Notes

- None of the lists are needed or constructed unless there is contention.
- The contender builds the lists, etc.
- The exit code tears them down.
- On exit, the owner:
  - Assigns the mutex to the first waiter.
  - Wakes up the new owner.
  - Readjusts his own priority.
- For UP systems we do not need “lock”ed access.
  - Contending only with an interrupt.

Pi Mutex Overhead Vs Spin Lock

- There is the cmpxchg overhead on UP systems.
  - Hard to beat “;”
- For SMP entry should be about the same.
  - Both spin lock and pi mutex use “lock”ed memory access which is most of the overhead.
- Exit cost more.
  - Spin lock exit is a un”lock”ed store byte.
  - Pi mutex uses the “lock”ed cmpxchg.
- But, spin lock stalls the contender, pi mutex allows contender to do other work.

The Big Kernel Lock (BKL)

- As we said before this is on the way out, however, it is still used and is a problem.
- Converting the BKL to a pi mutex presents problems:
  - It is used prior to the scheduler init code during bring up.
  - It is released and reacquired by schedule() when schedule() is called while the BKL is held.
    - Implies recursion.

BKL Solutions

- Convert pi mutex to spin:

  while(pi_mutex_tryenter(&mutex));

- Schedule() must not release the BKL if entered for preemption.
  - Use the TASK_PTEEMPTING flag.
  - Must not try to reacquire if it is already held.
    - Easy to do with pi mutex (see owner field).
- With care, schedule() can recur to wait for this lock.
  - Must not attempt to release BKL here.
    - Same as preemption entry.

The Latest Patch Is:

- The MontaVista web site is at:
  – www.mvista.com

- Latest patch will be found at:

ftp://ftp.mvista.com/pub/Area51/preemptable_kernel/

Status

- Most of the code is written.
- Conversion of the BKL will happen once SMP runs reliably.
- Current numbers are:
  - Change in time to do a kernel compile:
  - Max measured preemption delay: