- **How SMPng got where it is**
- **SMPng high level design**
- **New Tools**
- **SMPng high mid/low design issues**
- **SMPng globally visible changes**
- **Talk about technology I believe FreeBSD ought to pick up from BSD/OS.**
- **Hopefully have set expectations, with regard to performance and amount of work**

BSD/OS 4.x

Three lock kernel

- klock, main kernel lock, acquired before processing
  - Syscall
  - Trap
  - Interrupt
- Scheduling lock
  - Protects run queues
  - Allows a context switch without holding klock
- ipending lock
  - Allows ipending to be set without holding klock
  - Interrupts sometimes get delivered to "wrong" processor
- Performance is in general good
  - Equivalent to 3.x on uniprocessor
  - Almost perfect on a cpu saturated system using less than 1 processor worth of system cpu time

SMPng history

- * Rip out existing SMP code
- Install unsafe code mutex, Giant
- * Update interface to sleep/tsleep.
- Install scheduling mutex sched_lock.
  - Kernel runs from here on
- * Install updated interrupt code
- * Update scheduling code, AST changed to be per process.
- Specfs made to understand safe and unsafe drivers
- SCSI subsystem made MP safe
- * Malloc gets its own mutex.
- * All entries into VM system acquire Giant

SMPng history

- * Add interposition code to handle safe and unsafe network device drivers.
- First cut of networking code
- * Witness code implemented.
- Proc layer made MP safe.
- Made buffer cache MP safe.
- UFS read/write path pulled out from under Giant.
- Soft dependencies made to work.
- Mfs daemon runs without Giant.

SMPng history

- * NFS nightmare starts. Trying to resolve lock ordering it becomes clear that code that gets the vnode interlock can not hold Giant.
- * Changes such that no code holds Giant when doing VOP_(). This means that all, or at least almost all, file system code must be MP safe.
- NFS client and server run
- Vnode and splice drivers run
- * Stuff vnode and rnode back in side existing power of 2.
- Pick up latest soft dependency code
- Soft interrupt code made mostly MI, each netisr has its thread.

SMPng To Do

- * VM system needs to be locked up
- ISO 9660 code needs to be tested
- UFS Quotas need locking
- * Tty code needs to be made MP safe, then
  - Lock up existing serial drivers
  - Lock up PTY code
  - PPP
  - SLIP
  - Synchronous network devices
- * SIGXCPU is not delivered, SIGPROF is not always delivered
- Fix remaining users of vop_nolock

SMPng To Do

- Frame relay
- APM
- PCMCIA
- *Union file system
- Loop-back files system
- MS_DOS file system
- Test IPFW, IPV6, IPSEC, multicast

SMPng design goals

- Simultaneous execution of multiple kernel threads
- Support loosely cohered memory
- Allow for various hardware interrupt control schemes
- Simple programming model
  - Reduce/eliminate complex lock interactions
- Reasonable uniprocessor performance
- Good interrupt response
- Respect scheduling priorities

SMPng major characteristics

- All data must be protected by a lock
  - All global data is effectively volatile
- No SPLS
- In general blocking on a lock causes a context switch
- Both top half and bottom half code are subject to preemption
- Interrupts are handled by interrupt threads
- Simple lock ordering
  - If lock "A" held while lock "B" is acquire then "B" can not be held while "A" is acquired
  - Further, if lock "C" is ever acquired while holding lock "B" then "A" can not be acquired after "C"

Tools - Tracing

- low overhead kernel execution tracing mechanism
- conventional stack back trace much less useful
- functionality similar to printf
- format string is not decoded at run time
- circular buffer per cpu
- trace points conditionally present
- run time choice of what gets traced
- examples:

```
CTR0(KTR_PROC, "wakeupend");
 CTR1(KTR_PROC, "remrq proc=%p", (void *)p);
 CTR2(KTR_PMAP,
      "pmap_enter: wiring change->%x p=%x ",
      wired, CURPROC);
```

Tools - Witness

- Lock ordering
- Verification of duplicate instances of a mutex being acquired
- Verification of mutexs held when sleeping
- Verification that M_SPIN and M_DEF are not mixed

Tools - mp_fixme

- A unified way of marking that can be found with cscope or glimpse.
- Compiler generated error if the marking string is wrong.
- Descriptive text is closely associated with the marking string.

Tools - Kdebug

- Decoding and displaying system trace buffer
- Display various kernel and hardware variables
- Decode and display certain kernel data structures
- Display and modify 32 bit words or 8 bit bytes in physical or virtual memory
- Issue inb and outb instructions to I/O devices
- Use hardware breakpoint registers of CPU to implement four kernel breakpoints or watchpoints.
- Display stack back trace
- Start cross system KGDB
- Reboot the system immediately

SMPng synchronization mechanisms

- Mutexs
  - Context Switching
  - Spin
- Sleep/Wakeup
  - Sleep & tsleep have added argument
- Lock manager locks
  - Build on both mutexs and sleep/wakeup
  - Provides reader/writer locks

Synchronization mechanisms not present

- Low level reader/writer locks
  - No priority propagation
  - Slightly expensive
  - Implementation understood
- Counting semaphores

Mutex design goals

- Uncontested operations should be fast
- Support recursion
- Support priority propagation
- Reasonable debugging
- A function may not know type of mutex
- Don’t preclude mixed mode operation

Mutex (non-spin) details

- A mutex is acquire by setting a field to the proc which wants it
- Low order bits in the owner ship field are used as flags to prevent the compare and exchange from succeeding.
- The unowned value may not be zero.
- A linked list of mutexs which are owned by a thread and contested are kept for priority propagation.
- A linked list of processes blocked on a mutex are kept to allow a process to be put on the run queue when the mutex is released.

Mutex enter (non-spin) operation

- compare unowned and exchange proc pointer with field in mtx
- if successful done
- call mtx_enter_hard
- if recursion
  - set recurse bit
  - increment recursion count
  - return
- acquire sched_lock
- try to set contested bit on failure
  - release sched lock
  - logically start over
  - at this point uncontested release can not occur
- put our proc on list of procs blocked on mutex
- propagate priority as needed
- call cpu switch
- logically start over

Mutex exit (non-spin) operation

- compare proc pointer and exchange unowned with field in mtx
- if successful done
- call mtx_exit_hard
- if recursed
  - decrement recurse count if zero
    - clear recursed bit
  - return
- acquire sched_lock
- put blocked process on run queue
- set mutex to unowned
- if new process is higher priority
  - put self on run queue
  - call cpu_switch()
- return

Mutex Primitives

- mtx_enter(mtx_t *, int flag)
- mtx_try_enter(mtx_t *, int flag)
- mtx_exit(mtx_t *, int flag)
- mtx_init(mtx_t *, char *name, flag)
- mtx_owned(mtx_t *)
- mtx_destroy(mtx_t *)
- mtx_assert(mtx_t *, int what)

Flags used with Mutex Primitives

- M_DEF
- M_SPIN
- M_RLIKELY
- M_NORECURSE
- M_NOSPIN
- M_NOSWITCH
- M_FIRST
- M_TOPHALF

Hiding mutex primitives

```
#define MBUF_LOCK() mtx_enter(&mbuf_lock, M_DEF)
```

- Often gets in the way
  - What is underling mechanism
  - Exactly what lock is acquired
- Global changes easier
- Place to insert debugging code
- No always correct answer

Giant

- Used to protect data accessed from code which has not been converted.
- Can drop Giant in any function that can sleep
  - malloc(M_WAITOK)
- Ordering was/is a problem
  - malloc/kmem_malloc even with no sleep
  - sleep/tsleep assumes Giant before passed in mutex.
- DROP_GIANT() starts with "do {" so storage can be allocated
- PICKUP_GIANT() ends with "} while (0)"
- PARTIAL_PICKUP_GIANT same as PICKUP_GIANT without "}while (0)"

Idle Proc

- Idle proc for each processor
- Supplies initial context for interrupts
- Watches run queues and calls cpu_switch
- Never on a run queue
- Machine dependent
- Could do useful work

Run queues/process priority

- 32 user run queues
- 32 kernel run queues
- Run queue maps to hardware priority
- Priority set on entrance to kernel
- Exhaust kernel queues before returning to user

Interrupt Threads

- One for every interrupt source (level)
- ithrd is super set of proc
- Lightness comes from how they are started
- Can not call sleep
- Soft int are virtually identical to hard interrupts

Device driver interface

- Typically add mutex to softc
- Flag passed to intr_establish saying driver is MP safe
- Mutex typically gotten
  - Interrupt service routine
    - Safe to release and re-acquire mutex
  - Timeout functions
  - Top down entrances: open/close/strategy/ioctl/start
- Unexpected behavior mtx_exit() followed by mtx_enter()
- SCSI shares mutex between all layers
- Extra layers can be performance problem

Malloc

- Single mutex
- Can’t hold any locks with M_WAITOK
- Can acquire Giant even without M_WAITOK

Random

- Debugging
  - Don’t return to user with mutexs held
  - Don’t return to user with lock manager locks held
- mi_switch() does not call cpu_switch()

Globally visible changes

Sleep, tsleep

- Optional mutex passed in
- Giant is invisible argument
- Unless a timer is running some mutex must be used
- Typical usage:

```
mtx_enter(vbuf->b_mtxp, M_DEF)
while (!(vbuf->b_lflags & B_L_DONE))
    sleep(vbuf->b_mtxp, vbuf, PRIBIO);
mtx_exit(vbuf->b_mtxp, M_DEF)
```

- Sets processor priority
- Natural locking order for wakeup() less than perfect

Globally visible changes

Struct Buf

- Flags field split
- New field b_lflags
  - Contains bits which were protected by splbio
    - B_L_DONE
    - B_L_BUSY
    - B_L_WANTED
    - B_L_SCANNED
  - Access controller by buf_lock(struct buf *) and buf_unlock(struct buf *)
- Buffer must be initialized with buf_initmtx(struct buf *)
- No destroy function
- buf2mtx(struct buf *) used to get address of mutex associated with buf

Globally visible changes

Struct buf (continued)

- **Typical usage:**

```
int
 biowait(vbuf)
     register struct buf *vbuf;
 {
     buf_lock(vbuf);
     while (!(vbuf->b_lflags & B_L_DONE))
         sleep(buf2mtx(vbuf), vbuf, PRIBIO);
     buf_unlock(vbuf);
     if (buf_tstflags(vbuf, B_ERROR) == 0)
         return (0);
     return (vbuf->b_error ?
       vbuf->b_error : EIO);
 }
```

Globally visible changes

timeout / untimeout

- timeout() changed and renamed to _timeout()
- timeout() calls _timeout() with unsafe flag set()
- mp_timeout() calls _timeout() without unsafe flag set()
- spin_timeout() calls _timeout() with spin held flag set()
- untimeout() can fail

Globally visible changes

vnode

- **Different fields in the vnode are protected by different locks.**
  - **VOP_LOCK**
    - **This implies vop_nolock() has to totally goes away**
  - **v_interlock, VI_LOCK(), VI_UNLOCK()**
  - **vnode free list mtx**
- **v_flag split into v_iflag and v_vflag**
  - **Bits used with v_iflag are VI_**
  - **Bits used with v_vflag are VV_**
- **Many to one mapping of vnodes to v_interlock mutexs**
  - **This implies that only a single interlock can be held**

Globally visible changes

proc

- Likely most difficult area
- Many different locks for different fields
  - no lock, doesn't change after creation
  - mutex associated with proc
  - all proc mutex
  - proc tree
  - process group mutex
  - pid hash table mtx
  - sched lock
  - proc lock in attach proc or attaching proc parent
  - time lock
- Typically only lock one proc at a time
- When multiple order is child then parent
- Re-parenting is biggest problem
- all proc chain protected by reader/writer lock
- overall hierarchy protected by reader/writer lock

Net

- Interface queues are leaf locks
- In general locking optimized for interrupt threads
  - acquire pcb head
  - acquire pcb
  - generally drop pcb head
  - socket send/recv queue
- Have to drop and re-acquire lock from top
  - drop send/recv queue
  - acquire pcb head
  - acquire pcb
  - generally drop pcb head
  - acquire send/recv queue
- Net graphs may pose performance problem

Net Work Stack (continued)

- Locking buried in IF_ so typical coding just works
- Following get queue lock and leave lock:
  - IF_QLOCK()
  - IF_QFULL()
  - IF_PREPEND()
- Following release lock on completion:
  - IF_QUNLOCK()
  - IF_DROP()
  - IF_ENQUEUE()

Net Work Stack (continued)

- Following lock and unlock queue:
  - IF_DEQUEUE()
  - IF_PREPEND()
- Following expect queue to be locked:
  - IF_ENQUEUE_NOUNLOCK()
  - IF_DEQUEUE_NOLOCK()
- Many/Most uses just work

```
if (IF_QFULL(ifq))
        m_free(m);
        IF_DROP(ifq);
} else {
        IF_ENQUEUE(ifq, m);
}
```

NFS

- No NFS is major reason for people to not run kernel
- runs with real vop lock
- Share lock between server and client

Ktrace

- Broken in existing release
- Lock ordering problems
- Forks process to write data

What should FreeBSD pickup?