- 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

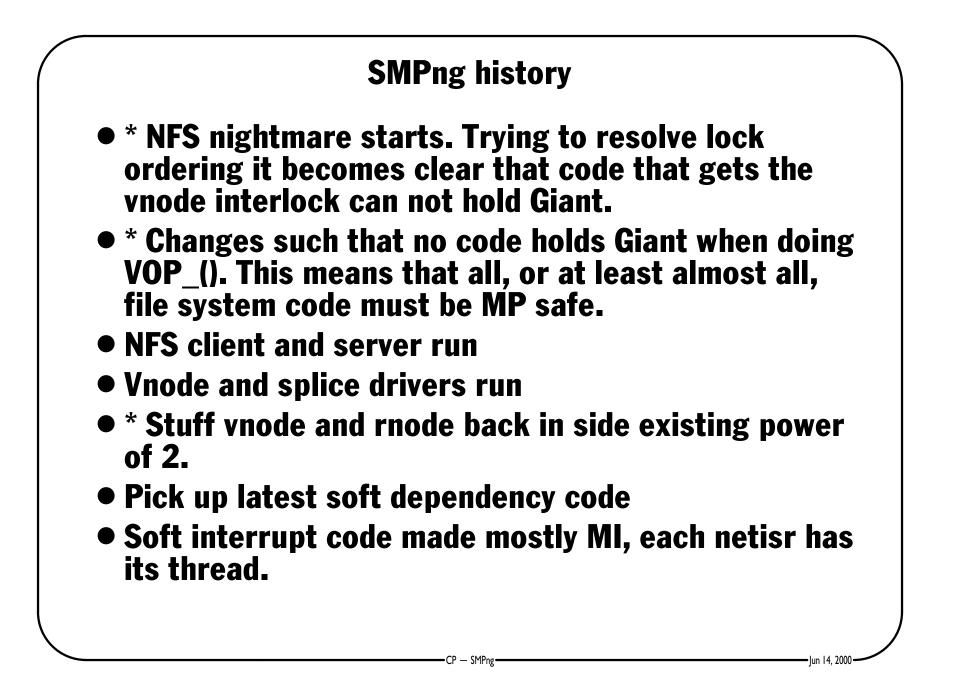
	BSD/0S 4.x
Tł	nree lock kernel
	klock, main kernel lock, acquired before processing
	○ Syscall
	o Trap
	○ Interrupt
	Scheduling lock
	• Protects run queues
	 Allows a context switch without holding klock
	pipending lock
	• Allows ipending to be set without holding klock
	O Interrupts sometimes get delivered 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

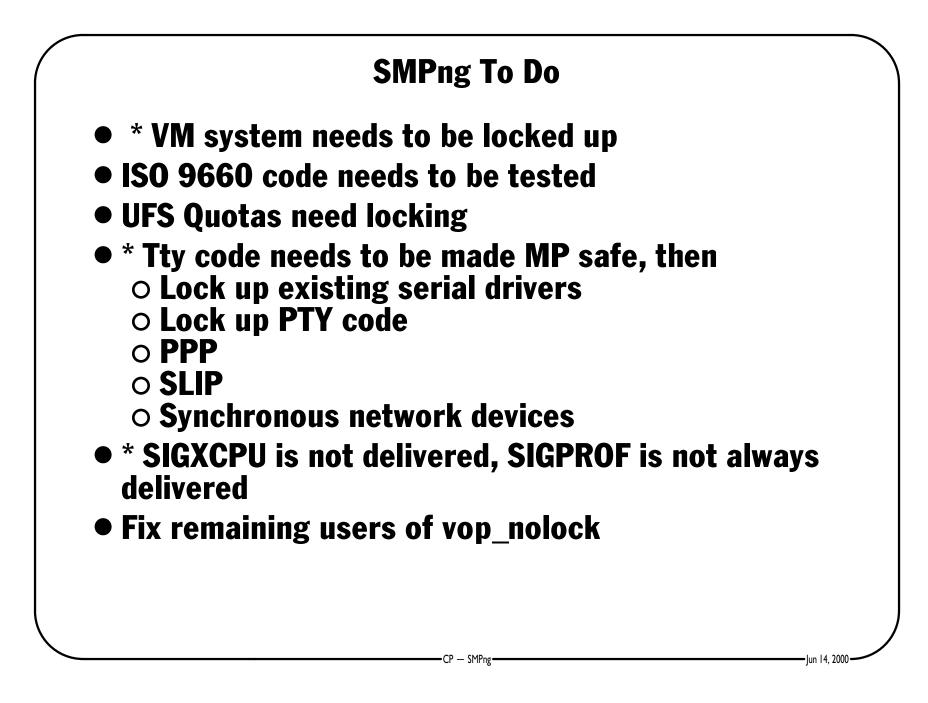
SMPng history

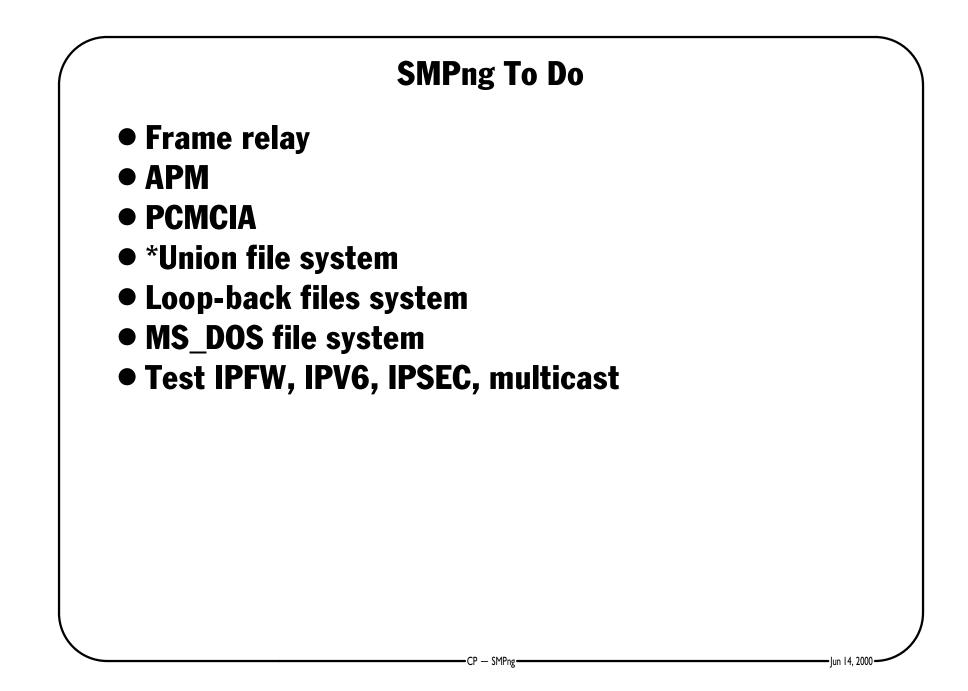
- * Rip out existing SMP code
- Install unsafe code mutex, Giant
- * Update interface to sleep/tsleep.
- Install scheduling mutex sched_lock.
 Kernel runs from here on
- Install updated interrupt code
- * Update scheduling code, AST changed to be per process.
- Specfs made to understand safe and unsafe 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 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

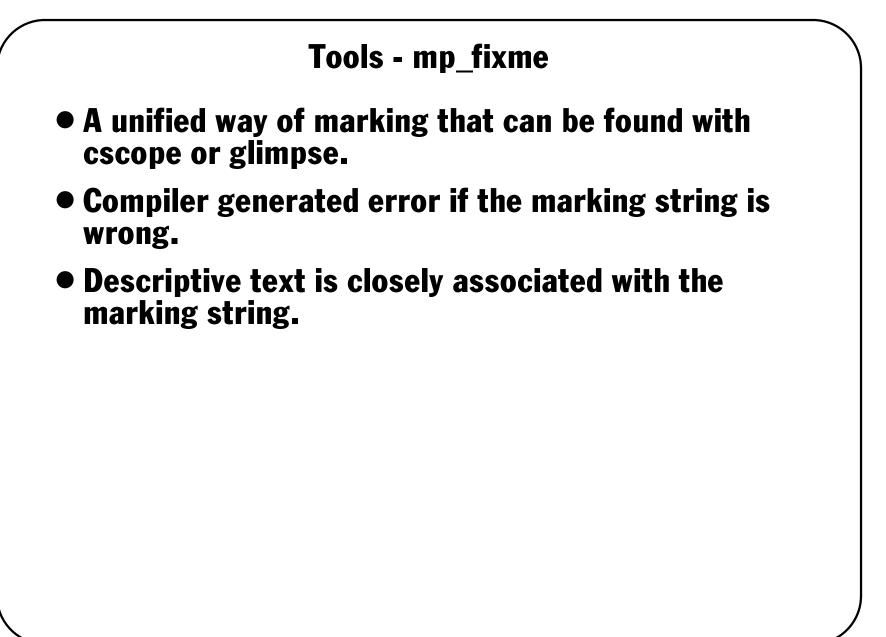
- Iow overhead kernel execution tracing mechanism
- conventional stack back trace much less useful
- functionality similar to printf
- format string is not decoded at run time
- circular buffer per cpu
- trace points conditionally present
- run time choice of what gets traced

```
• examples:
CTRO(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

un 14. 200



un 14. 200

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**
 - \circ 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
 - \circ 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

- \circ set recurse bit
- \circ increment recursion count
- \circ return

• acquire sched_lock

- try to set contested bit on failure
 - release sched lock
 - logically start over
 - $\boldsymbol{\circ}$ 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
- Iogically 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 	
 set indtex 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)

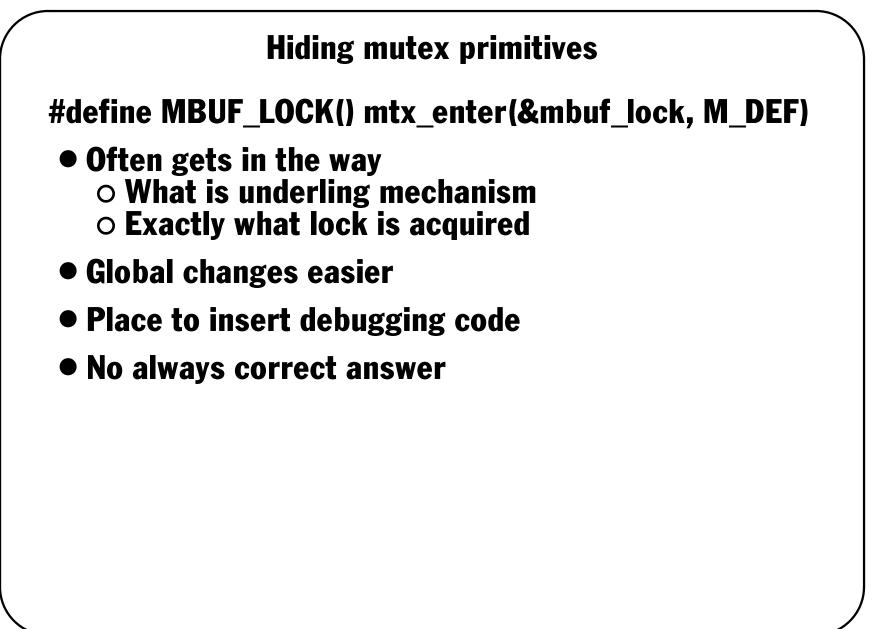
un 14, 200

Flags used with Mutex Primitives

• M_DEF

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

lun 14, 2000



P — SMPng-

	Giant
 Used to protect data not been converted 	ata accessed from code which has ed.
 Can drop Giant in malloc(M_WAI1 	any function that can sleep FOK)
 sleep/tsleep as mutex. DROP_GIANT() sta allocated PICKUP_GIANT() e 	malloc even with no sleep soumes Giant before passed in arts with "do {" so storage can be ends with "} while (0)" _GIANT same as PICKUP_GIANT

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

un 14, 200

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

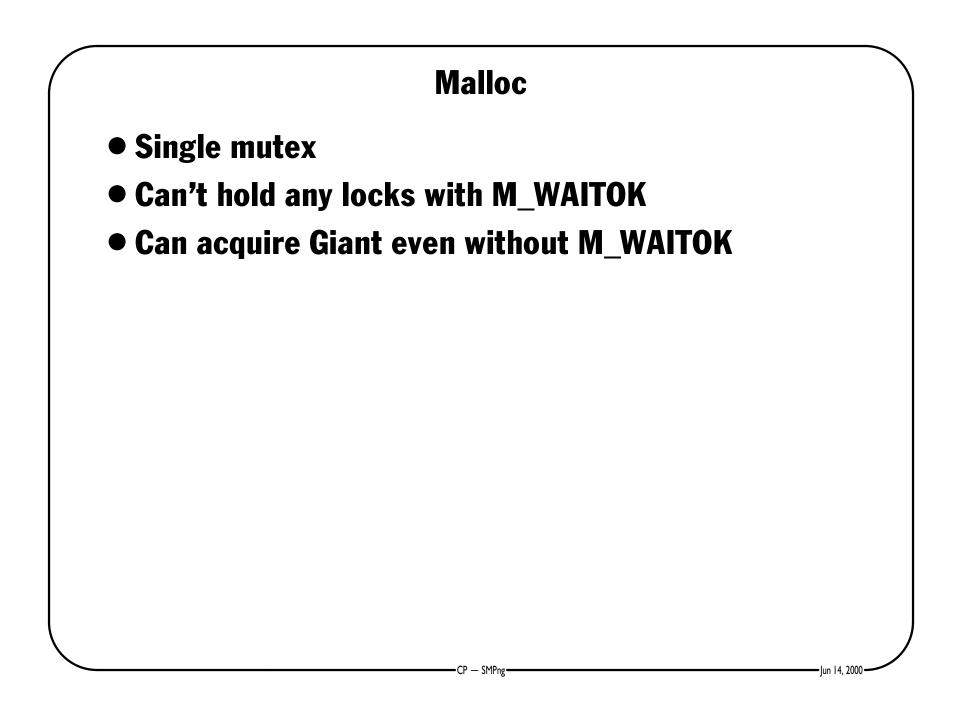
- One for every interrupt source (level)
- ithrd is super set of proc
- Lightness comes from how they are started
- Can not call sleep
- Soft int are virtually identical to hard interrupts

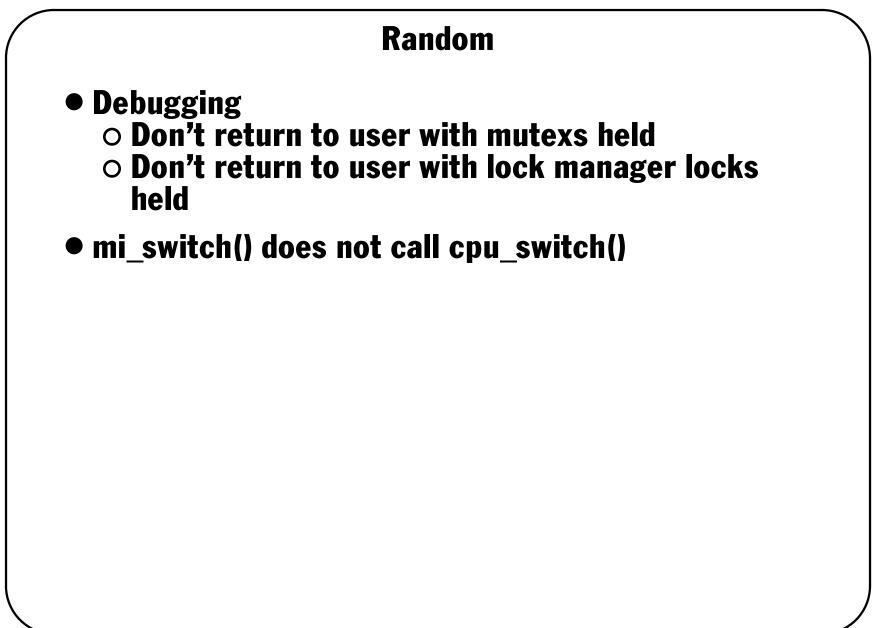
un 14, 200

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





un 14. 200

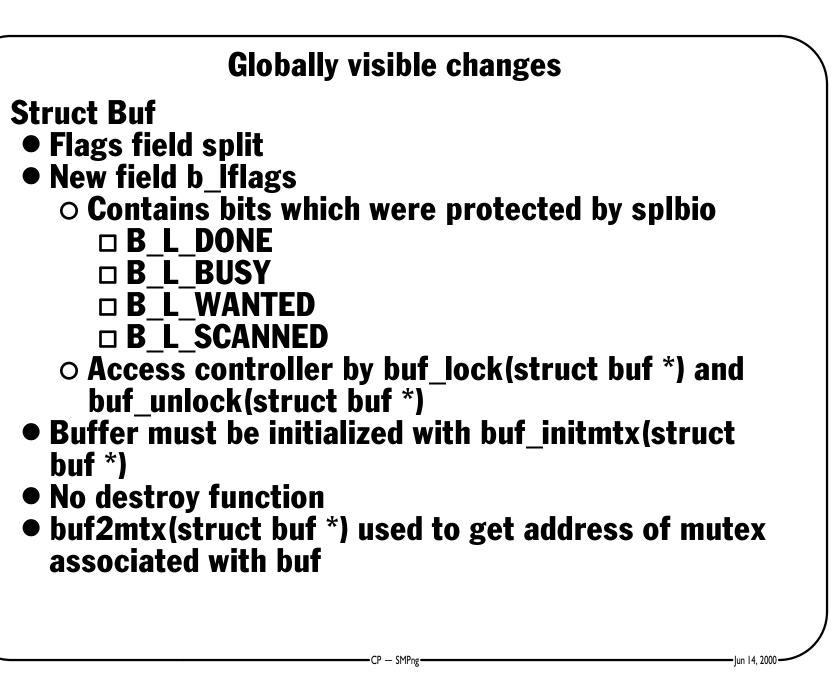
Globally visible changes

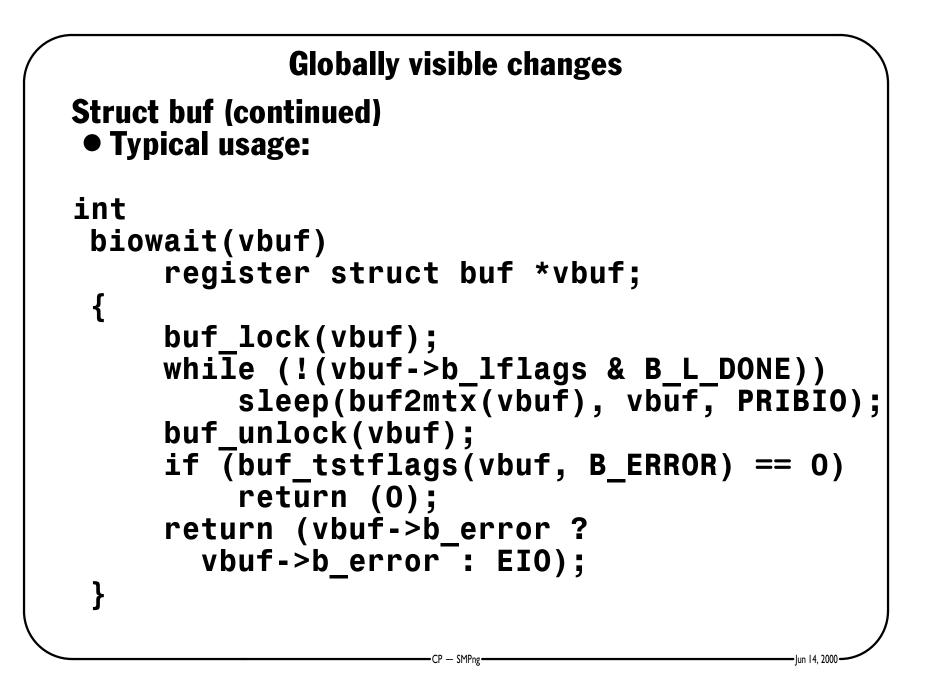
Sleep, tsleep

- 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

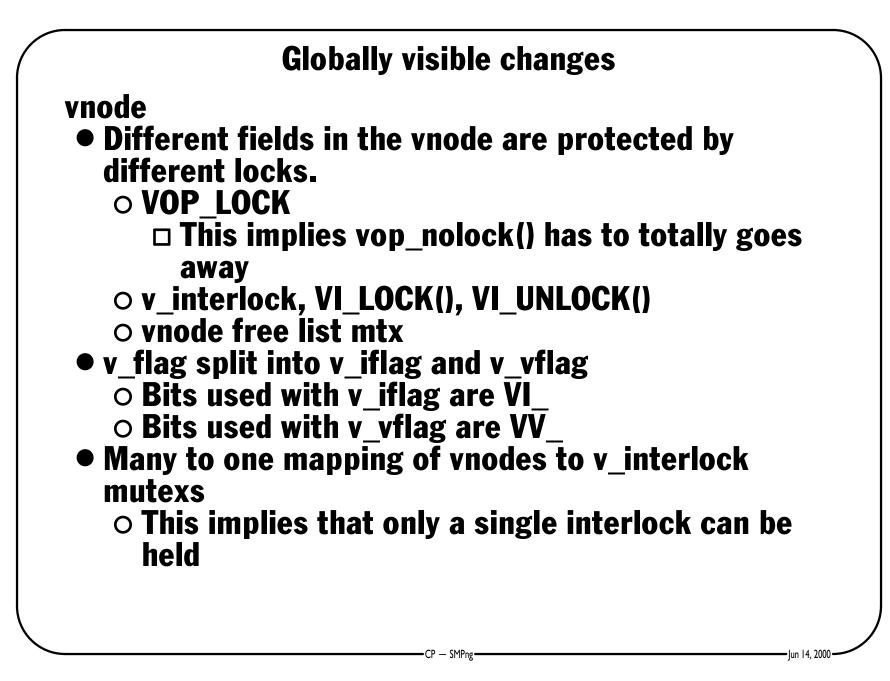




Globally visible changes

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



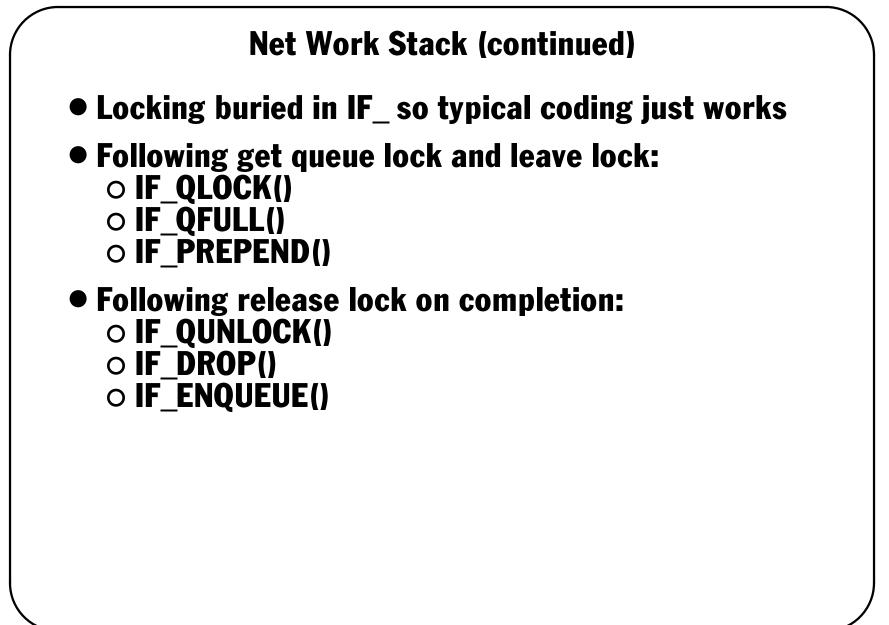
Globally visible changes

proc

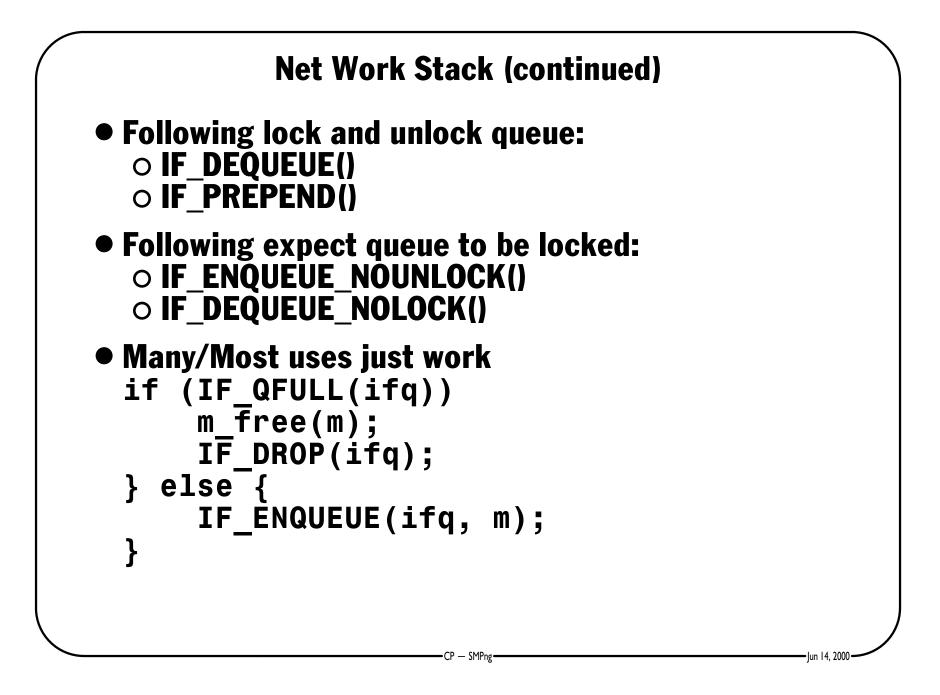
- Likely most difficult area
- Many different locks for different fields
 no lock, doesn't change after creation
 - mutex associated with proc
 - \circ all proc mutex
 - proc tree
 - process group mutex
 - \circ pid hash table mtx
 - \mathbf{O} sched lock
 - proc lock in attach proc or attaching proc 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

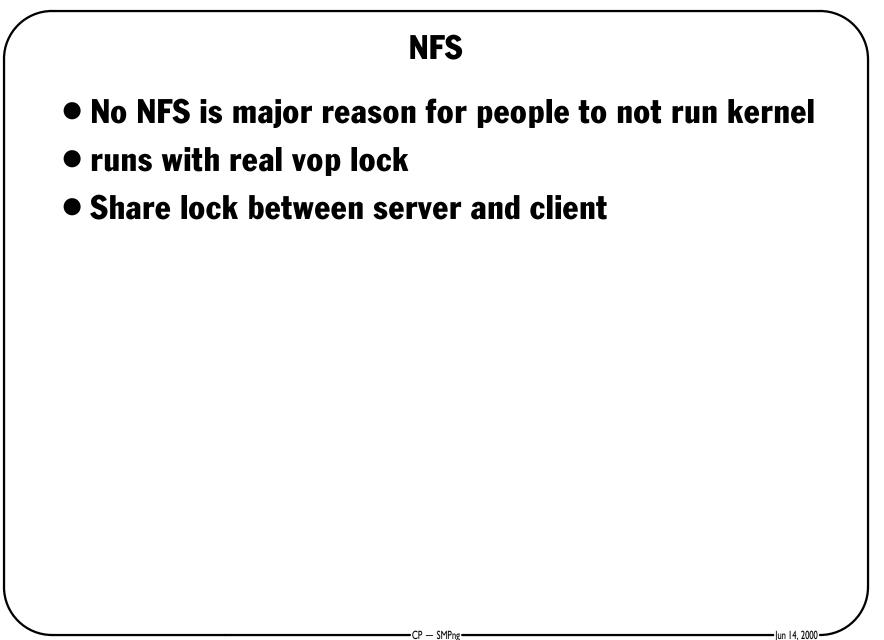
lun 14. 2000

Net	
Interface queues are leaf locks	
 In general locking optimized for interrup acquire pcb head acquire pcb generally drop pcb head socket send/recv queue 	t threads
 Have to drop and re-acquire lock from to drop send/recv queue acquire pcb head acquire pcb generally drop pcb head acquire send/recv queue) p
• Net graphs may pose performance probl	em



un 14, 200





Ktrace

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

lun 14, 2000

