The FreeBSD SMP implementation

Greg Lehey
grog@FreeBSD.org
grog@au1.ibm.com
Boston, 29 June 2001

Topics

- How we got into this mess.
- Threaded interrupt handlers.
- Kinds of locks.
- Debugging.

The UNIX kernel design

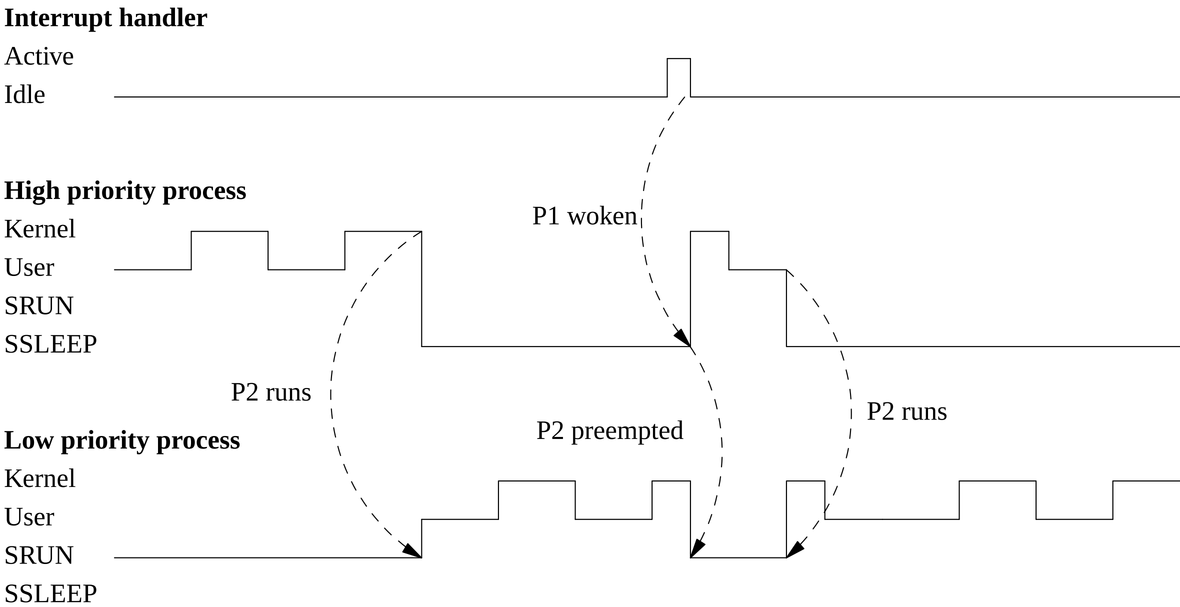
- One CPU
- Processes perform user functions.
- Interrupt handlers handle I/O.
- Interrupt handlers have priority over processes.

Processes

- One CPU
- Processes have different priorities.
- The scheduler chooses the highest priority process which is ready to run.
- The process can relinquish the CPU voluntarily (`tsleep`).
- The scheduler runs when the process finishes its time slice.
- Processes are not scheduled while running kernel code.

Interrupts

- Interrupts cannot be delayed until kernel is inactive.
- Different synchronization: block interrupts in critical kernel code.
- Finer grained locking: `splbio` for block I/O, `spltty` for serial I/O, `splnet` for network devices, etc.



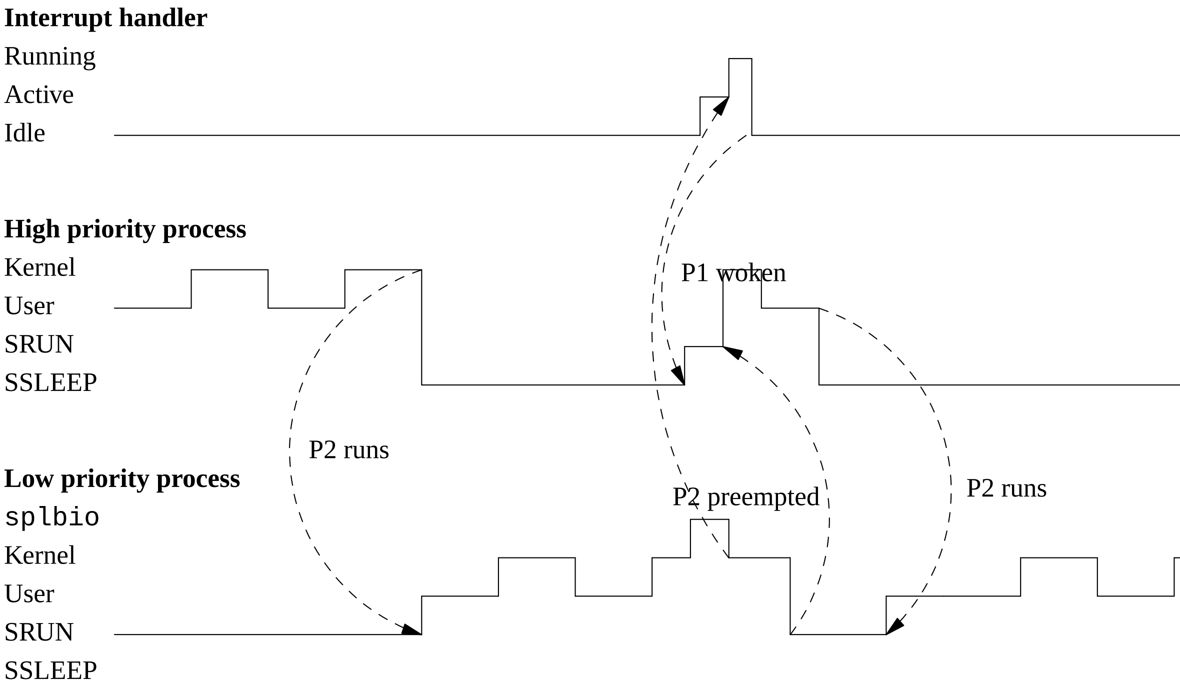
Ideal single processor scheduling

Problems with this approach

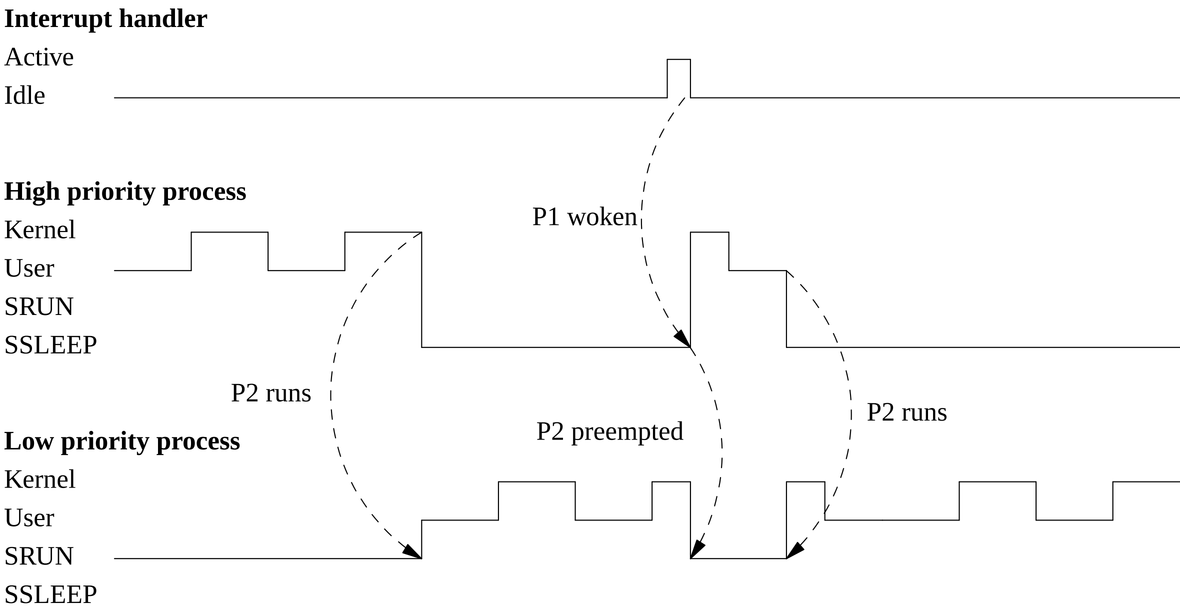
Kernel synchronization is inadequate. UNIX can’t guarantee consistency if multiple processes can run in kernel mode at the same time.

Solution: Ensure that a process leaves kernel mode before preempting it. Since processes do not execute kernel code for very long, this causes only minimal problems.

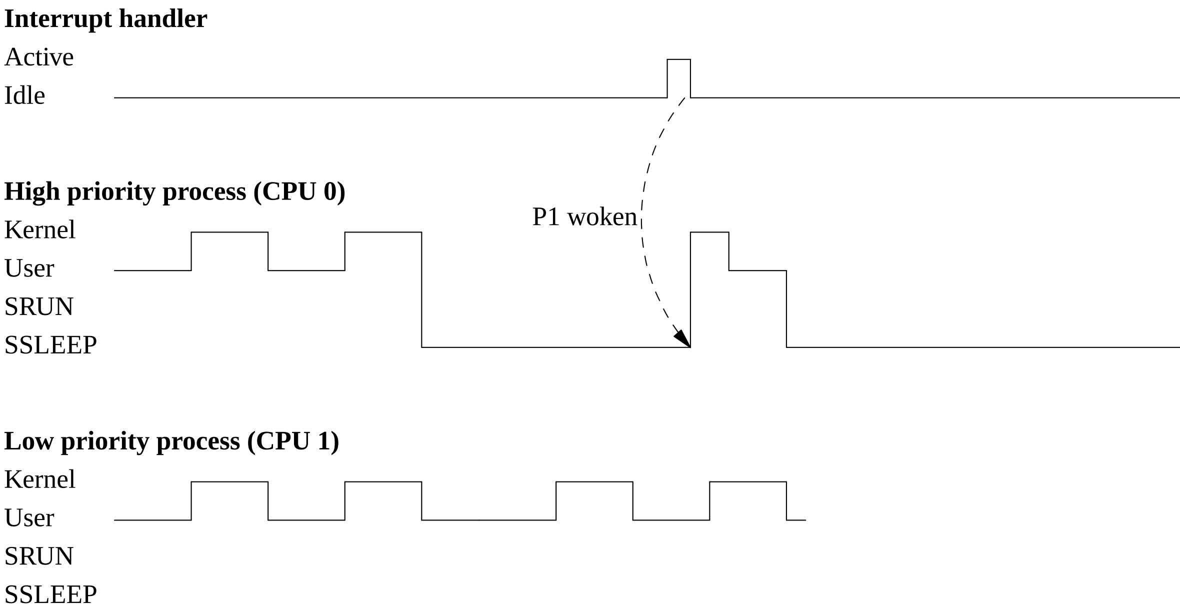
Danger: If a process does stay in the kernel for an extended period of time, it can cause significant performance degradation or even hangs.



Real single processor scheduling



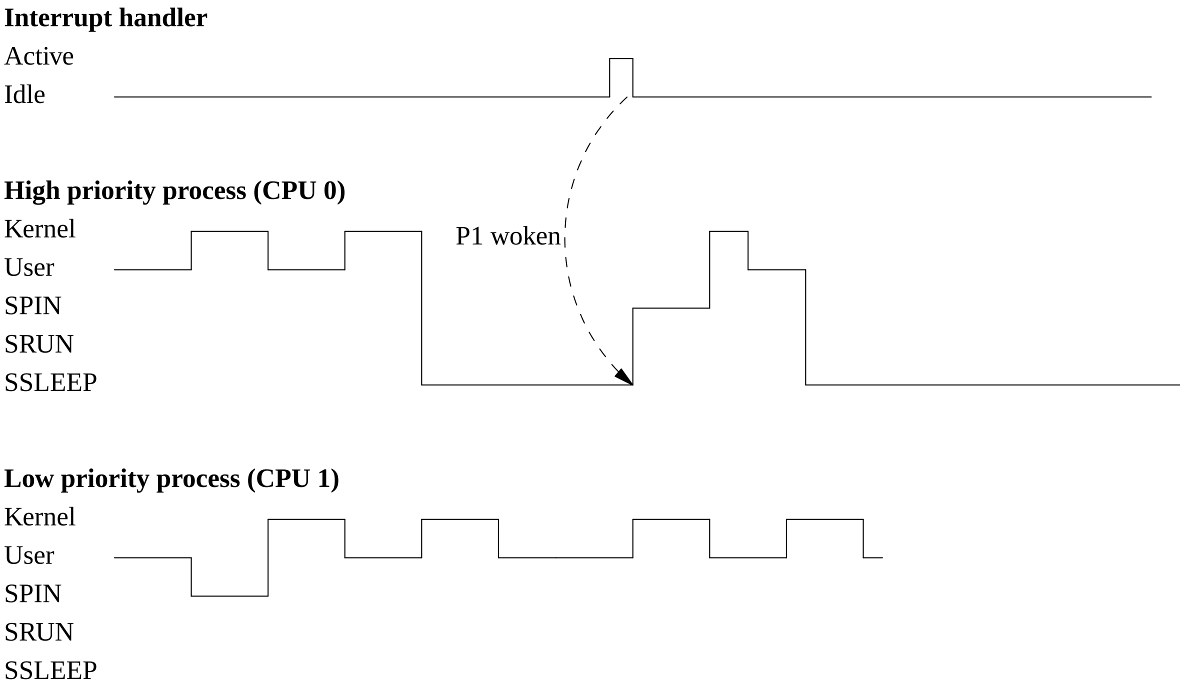
Ideal single processor scheduling



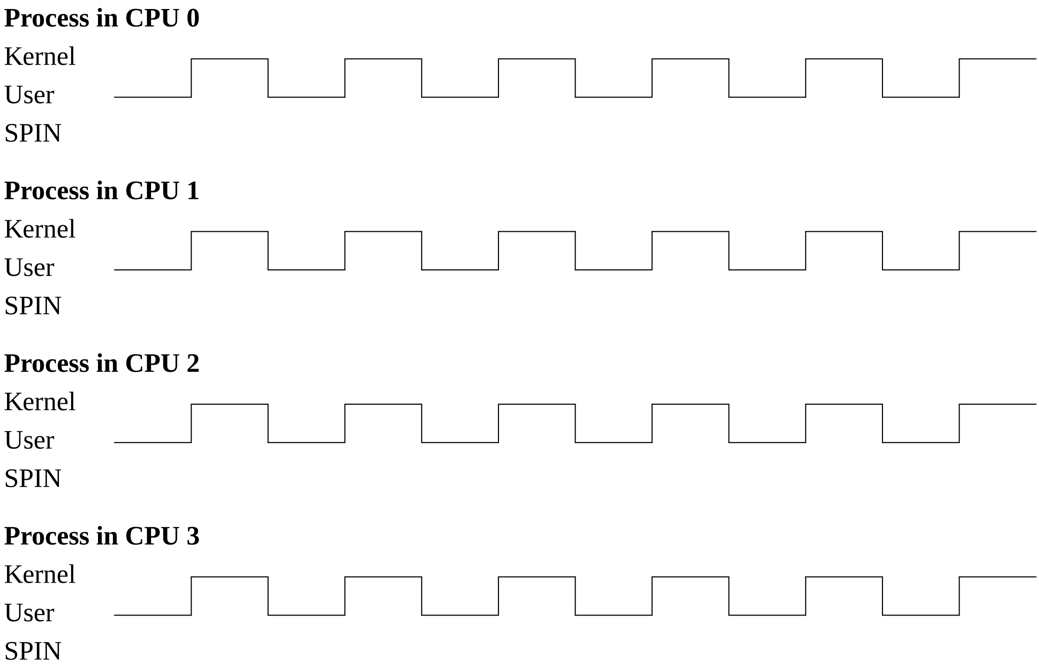
Ideal dual processor scheduling

Problems with ideal view

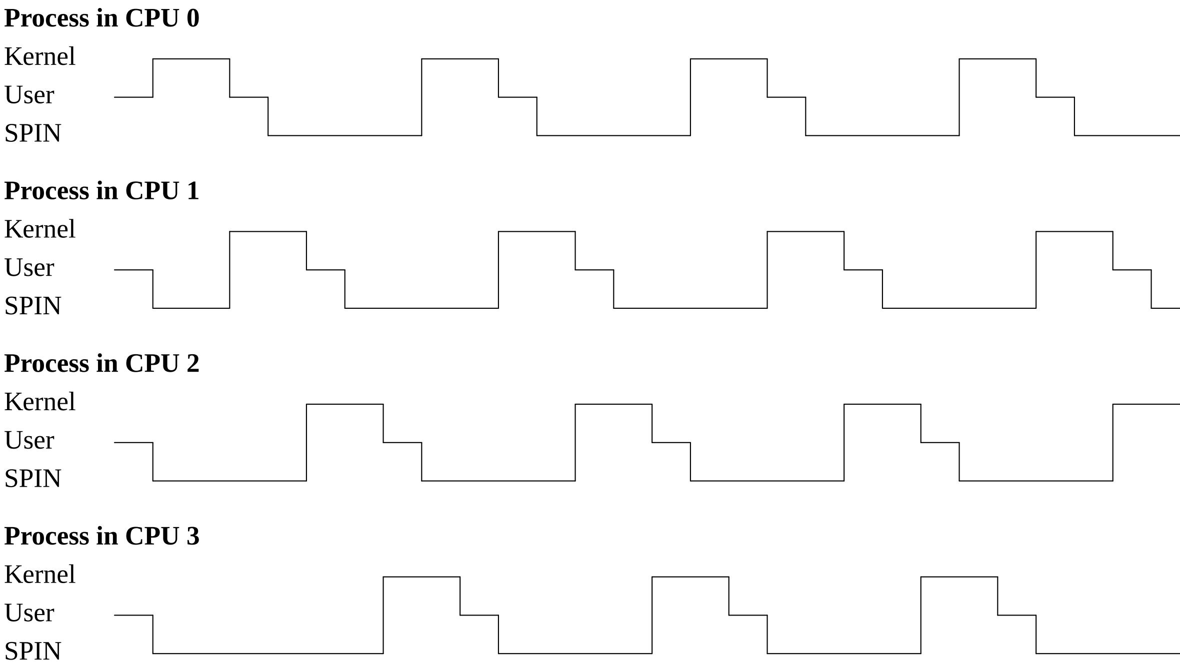
- Can’t have more than one process running in kernel mode.
- “Solution”: introduce Big Kernel Lock. Spin (loop) waiting for this lock if it’s taken.
- Disadvantage: much CPU time may be lost.



Real dual processor scheduling



Extreme quad processor scheduling: ideal



Extreme quad processor scheduling: real

Limiting the delays

- Create ‘‘fine-grained’’ locking: lock only small parts of the kernel.
- If resource is not available, block, don’t spin.
- Problem: interrupt handlers can’t block.
- Solution: let them block, then.

Blocking interrupt handlers

- Interrupt handlers get a process context.
- Short term: normal processes, involve scheduler overhead on every invocation.
- Longer term: “light weight interrupt threads”, scheduled only when conflicts occur.
- Choice dictated by stability requirements during changeover.
- Resurrect the idle process, which gives a process context to each interrupt process.

Blocking interrupt handlers

```
USER      PID %CPU %MEM   VSZ  RSS  TT  STAT STARTED      TIME COMMAND
root       10 98.1  0.0     0    0  ??  RWL    2:54PM   4:41.65  (idle: cpu1)
root       11 98.1  0.0     0    0  ??  RWL    2:54PM   4:41.73  (idle: cpu0)
root       13  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.63  (swi6: tty:sio+)
root       14  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.00  (swi4: vm)
root       15  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.00  (swi5: task queue)
root       16  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.00  (swi2: camnet)
root       17  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.01  (swi3: cambio)
root       18  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.03  (irq14: ata0)
root       19  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.00  (irq15: ata1)
root       20  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.01  (irq3: dc0)
root       21  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.01  (irq10: ahc0)
root       22  0.0  0.0     0    0  ??  WWL    2:54PM   0:03.13  (irq11: atapci1+)
root       23  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.00  (irq1: atkbd0)
root       24  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.00  (swi0: tty:sio)
root       25  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.00  (irq4: sio0)
root       26  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.00  (irq7: ppc0)
root       27  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.00  (irq0: clk)
root       28  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.00  (irq8: rtc)
root       12  0.0  0.0     0    0  ??  WWL    2:54PM   0:00.02  (swi1: net)
```

Types of locking constructs

- Semaphores.
- Spin locks.
- Adaptive locks.
- Blocking locks.
- Condition variables.
- Read-write locks.

Locking constructs are also called *mutexes* .

Semaphores

- Oldest synchronization primitive.
- Include a *count* variable which defines how many processes may access the resource in parallel.
- No concept of ownership.
- The process that releases a semaphore may not be the process which last acquired it.
- Waiting is done by blocking (scheduling).
- Traditionally used for synchronization between processes.

Spin locks

- Controls a single resource: only one process may own it.
- ‘‘busy wait’’ when lock is not available.
- May be of use where the delay is short (less than the overhead to run the scheduler).
- Can be very wasteful for longer delays.
- The only primitive that can be used if there is no process context (traditional interrupt handlers).
- May have an *owner*, which is useful for consistency checking and debugging.

Blocking lock

- Controls a single resource: only one process may own it.
- Runs the scheduler when lock is not available.
- Generally usable where process context is available.
- May be less efficient than spin locks where the delay is short (less than the overhead to run the scheduler).
- Can only be used if there is a process context.
- May have an *owner*, which is useful for consistency checking and debugging.

Adaptive lock

- Combination of spin lock and blocking lock.
- When lock is not available, spin for a period of time, then block if still not available.
- Can only be used if there is a process context.
- May have an *owner*, which is useful for consistency checking and debugging.

Condition variable

- Tests an external condition, blocks if it is not met.
- When the condition is met, all processes sleeping on the wait queue are woken.
- Similar to *tsleep* /*wakeup* synchronization.

Read-write lock

- Allows multiple readers or alternatively one writer.

Comparing locks

Lock type	Multiple resources	owner	requires context
Semaphore	yes	no	yes
Spin lock	no	yes	no
Blocking lock	no	yes	yes
Adaptive lock	no	yes	yes
Condition variable	yes	no	yes
Read-write lock	yes	no	yes

Recursion

- What do we do if a process tries to take a mutex it already has?
- Could be indicative of poor code structure.
- In the short term, it's very likely.
- Solaris does not allow recursion, and this has caused many problems.
- Currently FreeBSD allows recursion. Discussion is still intense.

FreeBSD locks

```
struct lock_object {
        struct lock_class *lo_class;
        const  char *lo_name;
        const  char *lo_file;       /* File and line of last acquire. */
        int    lo_line;
        u_int  lo_flags;
        STAILQ_ENTRY(lock_object) lo_list; /* List of all locks in system. */
        struct witness *lo_witness;
};

#define LO_INITIALIZED  0x00010000       /* Lock has been initialized. */
#define LO_WITNESS      0x00020000      /* witness this lock. */
#define LO_QUIET        0x00040000      /* Don't log locking operations. */
#define LO_RECURSABLE   0x00080000      /* Lock may recurse. */
#define LO_SLEEPABLE    0x00100000      /* Lock may be held when sleeping */
#define LO_LOCKED       0x01000000      /* Someone holds this lock. */
#define LO_RECURSED     0x02000000      /* Someone has recursed this lock */
```

FreeBSD mutex

```
struct mtx {
        struct lock_object mtx_object;      /* Common lock properties. */
        volatile uintptr_t mtx_lock;/* owner (and state for sleep locks) */
        volatile u_int mtx_recurse; /* number of recursive holds */
        critical_t mtx_savecrit;    /* saved flags (for spin locks) */
        TAILQ_HEAD(, proc) mtx_blocked;     /* threads blocked on this lock */
        LIST_ENTRY(mtx)      mtx_contested;/* list of all contested locks */
};

#defineMTX_DEF       0x00000000    /* DEFAULT (sleep) lock */
#define MTX_SPIN     0x00000001    /* Spin lock (disables interrupts) */
#define MTX_RECURSE  0x00000004    /* Option: lock allowed to recurse */
#defineMTX_NOWITNESS 0x00000008    /* Don’t do any witness checking. */
#defineMTX_SLEEPABLE 0x00000010    /* We can sleep with this lock. */
```

Condition variables

```
struct cv {
        struct cv_waitq      cv_waitq;     /* Queue of condition waiters. */
        struct mtx    *cv_mtx;      /*
                                     * Mutex passed in by cv_*wait*(),
                                     * currently only used for CV_DEBUG.
                                     */
        const char    *cv_description;
};
```

Condition variables

- Acquire a condition variable with `cv_wait()`, `cv_wait_sig()`, `cv_timedwait()` or `cv_timedwait_sig()`.
- Before acquiring the condition variable, the associated mutex must be held. The mutex will be released before sleeping and reacquired on wakeup.
- Unblock one waiter with `cv_signal()`.
- Unblock all waiters with `cv_broadcast()`.
- Wait for queue empty with `cv_waitq_empty`.
- Same functionality available from the `msleep` function.

msleep

- A version of `tsleep` which takes a mutex parameter.
- The mutex will be released before sleeping and reacquired on wakeup.
- Similar to the behaviour of `tsleep` with `splx` functions in traditional UNIX.
- `tsleep` reimplemented as a macro calling `msleep` with null mutex.
- Functionality equivalent to condition variables, which should be used for new code.

Shared/exclusive locks

Another name for reader/writer locks.

```
struct sx {
        struct lock_object sx_object;   /* Common lock properties. */
        struct mtx      sx_lock;        /* General protection lock. */
        int             sx_cnt;         /* -1: xlock, > 0: slock count. */
        struct cv       sx_shrd_cv;     /* slock waiters. */
        int             sx_shrd_wcnt;   /* Number of slock waiters. */
        struct cv       sx_excl_cv;     /* xlock waiters. */
        int             sx_excl_wcnt;   /* Number of xlock waiters. */
        struct proc     *sx_xholder;    /* Thread presently holding xlock. */
};
```

Shared/exclusive locks

- More expensive than mutexes, should only be used where very few write (exclusive) accesses occur.
- All functions require a pointer to a user-allocated `struct sx`.
- Create an sx lock with `sx_init()`.
- Attain a read (shared) lock with `sx_slock()` and release it with `sx_sunlock()`.
- Attain a write (exclusive) lock with `sx_xlock()` and release it with `sx_xunlock()`.
- Destroy an sx lock with `sx_destroy`.

Original locks

- `Giant`: protects the kernel.
- `sched_lock`: protects the scheduler.

Current situation

- `Giant` still protects most of the kernel, but is being weakened.
- `softclock` and signal handling are now MP-safe and do not require `Giant`.
- Individual components protected by leaf node mutexes.
- Many device drivers now converted.
- Choice of construct often left to individual developer.
- Few mid-range locking constructs.

Debugging

- Based on BSD/OS work.
- *ktr* maintains a kernel trace buffer.
- *witness* code debugs mutex use.

ktr

- Traces programmer-specified events.
- Multiple classes, e.g.

```
#define KTR_GEN         0x00000001              /* General (TR) */
#define KTR_NET         0x00000002              /* Network */
#define KTR_DEV         0x00000004              /* Device driver */
#define KTR_LOCK        0x00000008              /* MP locking */
#define KTR_SMP         0x00000010              /* MP general */
#define KTR_FS          0x00000020              /* Filesystem */
```

- Code only generated if class bit is set in kernel option `KTR_COMPILE`.
- Code only executed if class bit is set in variable `ktr_mask`, initially set from kernel option `KTR_MASK`.

ktr (continued)

- Stores trace information in fixed-size entries in a circular buffer.
- Low overhead trace stores pointers to format strings and decodes them via *tdump(8)*.
- *tdump(8)* has not yet been ported to FreeBSD.
- High-overhead trace enabled with kernel option `KTR_EXTEND`.
- Trace entries include complete formatted data.
- Suitable for use during intensive debug.
- Orders of magnitude slower than default ‘‘low-overhead’’ trace.

ktr (continued)

Sample call (*i386/isa/ithread.c*):

```
void
sched_ithd(void *cookie)
...

        CTR3(KTR_INTR, "sched_ithd pid %d(%s) need=%d",
                ir->it_proc->p_pid, ir->it_proc->p_comm, ir->it_need);
...
                CTR1(KTR_INTR, "sched_ithd: setrunqueue %d",
                        ir->it_proc->p_pid);
...
void
ithd_loop(void *dummy)

...
                CTR3(KTR_INTR, "ithd_loop pid %d(%s) need=%d",
                     me->it_proc->p_pid, me->it_proc->p_comm, me->it_need);
...
```

Sample ktr output

```
138 0:034559493 cpu0 machine/mutex.h.510
        REL sched lock [0xfffffc00006662d0] at ../../kern/kern_synch.c:813 r=0
137 0:034508805 cpu0 machine/mutex.h.471
        GOT sched lock [0xfffffc00006662d0] at ../../kern/kern_synch.c:785 r=0
136 0:032610555 cpu0 machine/mutex.h.471
        GOT Giant [0xfffffc00006664a0] at ../../kern/kern_synch.c:958 r=0
135 0:032560177 cpu0 machine/mutex.h.510
        REL Giant [0xfffffc00006664a0] at ../../alpha/alpha/interrupt.c:123 r=0
134 0:032509499 cpu0 machine/mutex.h.471
        GOT Giant [0xfffffc00006664a0] at ../../alpha/alpha/interrupt.c:121 r=0
133 0:032504810 cpu0 ../../alpha/alpha/interrupt.c.115
        clock interrupt
132 0:032450423 cpu0 machine/mutex.h.510
        REL sched lock [0xfffffc00006662d0] at ../../kern/kern_synch.c:956 r=1
```

Debugger extensions

- FreeBSD has a different kernel debugger from BSD/OS, no import of functionality.
- Macros for *gdb* : Display *ktr* information.

The way ahead

- Gradually weaken `Giant`.
- Convert interrupt handlers to use mutexes.
- Maintain discipline: we can expect chaos as `Giant` loses its strength.
- Particular challenge for an ‘‘Open Source’’ project.

Further information

The FreeBSD SMPng is looking for clever hackers. Join in!

Web page `http://www.FreeBSD.org/smp/`

Mailing list `FreeBSD-smp@FreeBSD.org`. Use *majordomo* to sign up.