```
From: John Baldwin <jhb@FreeBSD.org>
Date: Tue, 21 Nov 2000 13:10:15 -0800 (PST)

jhb         2000/11/21 13:10:15 PST

  Modified files:
    sys/kern             kern_ktr.c
  Log:
  Ahem, fix the disclaimer portion of the copyright so it disclaim's the
  voices in my head.  You can sue the voices in Bill Paul's head all you
  want.

  Noticed by:   jhb

  Revision  Changes    Path
  1.6       +3 -3      src/sys/kern/kern_ktr.c
```

From: John Baldwin <jhb@FreeBSD.org>

On 21-Nov-00 John Baldwin wrote:
> jhb         2000/11/21 13:10:15 PST
>
>   Modified files:
>     sys/kern             kern_ktr.c
>   Log:
>   Ahem, fix the disclaimer portion of the copyright so it disclaim's the
>   voices in my head.  You can sue the voices in Bill Paul's head all you
>   want.
>
>   Noticed by: jhb

Oh geez.  That should be 'Noticed by: jlemon'.  I guess the voices
are getting a bit too rambunctious.

From: Warner Losh <imp@village.org>

In message <XFMail.001121131818.jhb@FreeBSD.org> John Baldwin writes:
: Oh geez.  That should be 'Noticed by: jlemon'.  I guess the voices
: are getting a bit too rambunctious.

It could be worse.  You could be talking about yourself in the third
person.  Warner hates it when he does that.

From: John Baldwin <jhb@FreeBSD.ORG>

On 21-Nov-00 Warner Losh wrote:
> In message <XFMail.001121131818.jhb@FreeBSD.org> John Baldwin writes:
>: Oh geez.  That should be 'Noticed by: jlemon'.  I guess the voices are
>: getting
>: a bit too rambunctious.
>
> It could be worse.  You could be talking about yourself in the third
> person.  Warner hates it when he does that.

Well, I'm sure Warner will have a private discussion with Warner about
doing that in public.

I wonder how the voices do their locking...

```
From: Warner Losh <imp@village.org>

In message <XFMail.001121133952.jhb@FreeBSD.org> John Baldwin writes:
: On 21-Nov-00 Warner Losh wrote:
: > In message <XFMail.001121131818.jhb@FreeBSD.org> John Baldwin writes:
: >: Oh geez.  That should be 'Noticed by: jlemon'.  I guess the voices are
: >: getting
: >: a bit too rambunctious.
: >
: > It could be worse.  You could be talking about yourself in the third
: > person.  Warner hates it when he does that.
:
: Well, I'm sure Warner will have a private discussion with Warner about
: doing that in public.

Warner will do that only if Warner notices.

: I wonder how the voices do their locking...

Warner Speculates that Warner's voices don't do locking.
```

From: John Baldwin <jhb@FreeBSD.ORG>

On 21-Nov-00 Warner Losh wrote:
> In message <XFMail.001121133952.jhb@FreeBSD.org> John Baldwin writes:
>: Well, I'm sure Warner will have a private discussion with Warner about
>: doing that in public.
>
> Warner will do that only if Warner notices.
>
>: I wonder how the voices do their locking...
>
> Warner Speculates that Warner's voices don't do locking.

John thinJohn's voices are too inks that the consefficient to useole driver
doesn't ha sleep locks and endve any lock up sping yeinning a lott.
fatal double fault
eip = 0x000000
ebp = %62F k epomn e

# The FreeBSD SMPng implementation

Greg Lehey
grog@FreeBSD.org
Adelaide, 25 November 2000

# Topics

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

# The UNIX kernel design

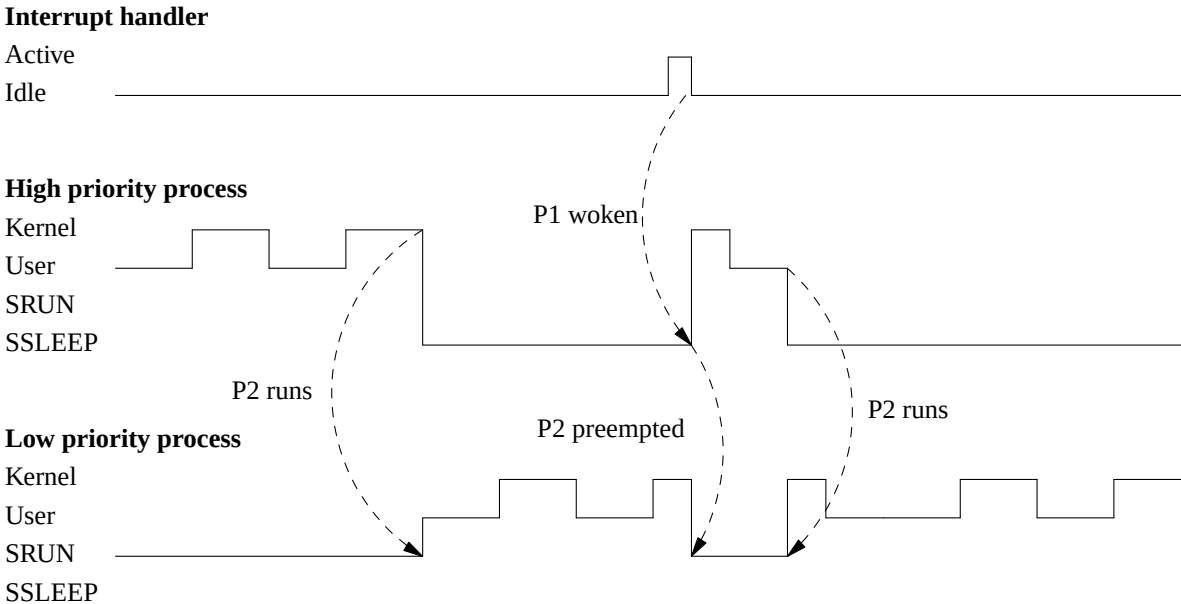
- One CPU
- Processes perform user functions.
- Interrupt handlers handle I/O.
- Interrupt handlers have priority over processes.

# 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.



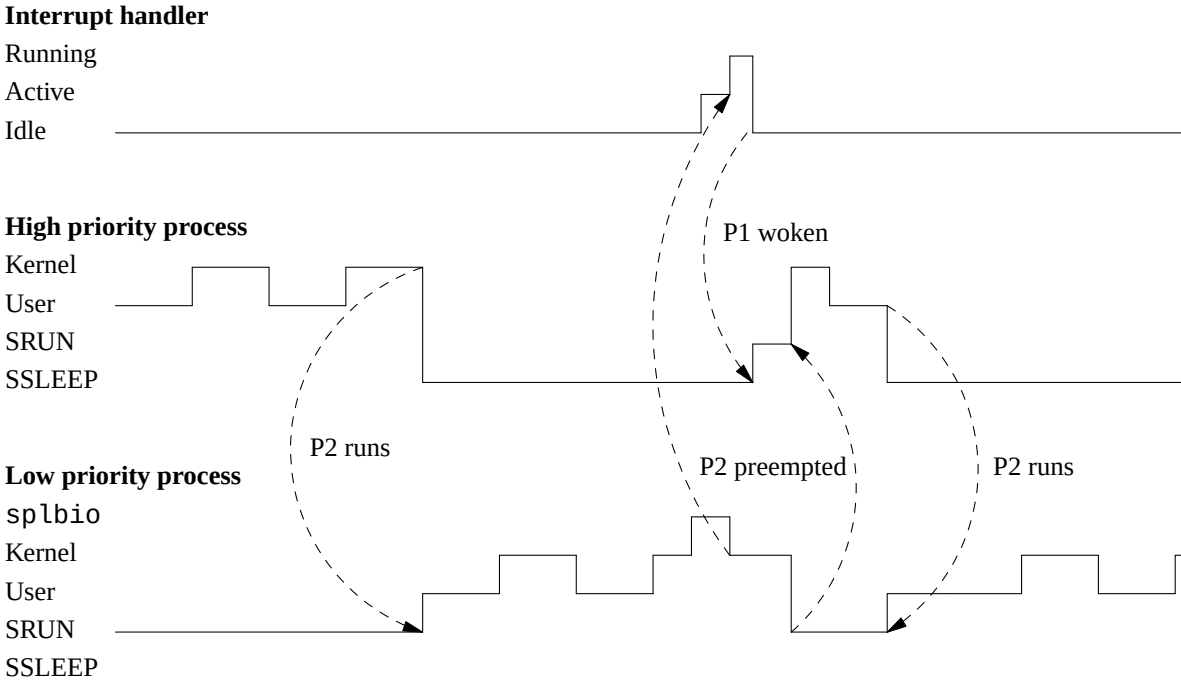

# Ideal single processor scheduling

# Problems with this approach

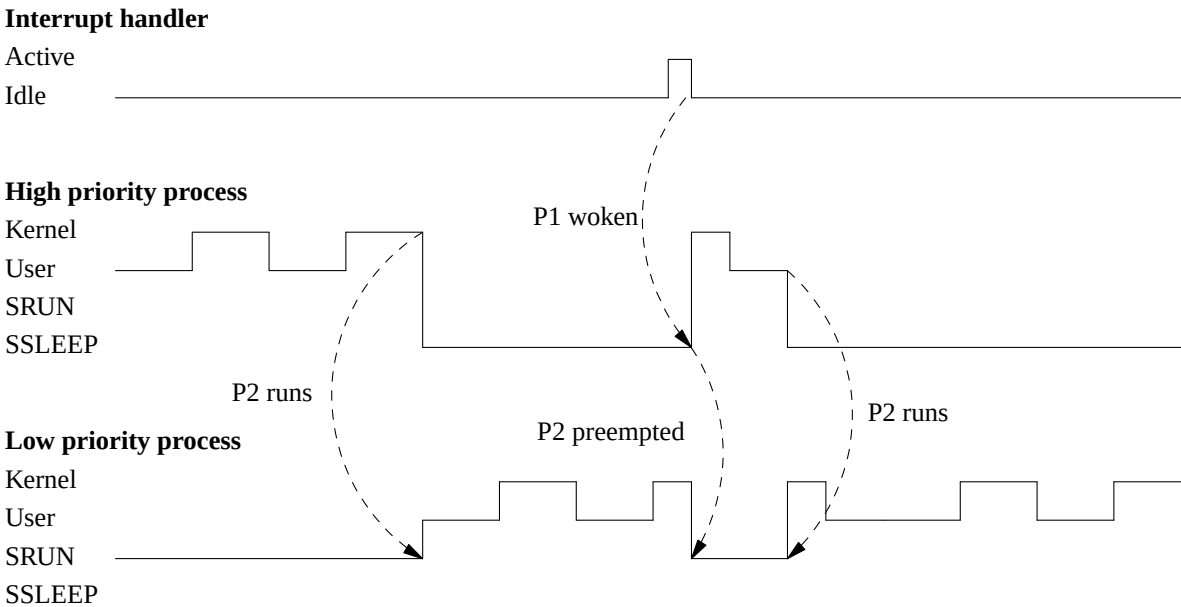
Kernel synchronization is inadequate. UNIX can't guarantee consistency if multiple processes can run in kernel mode at the same time.

**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.

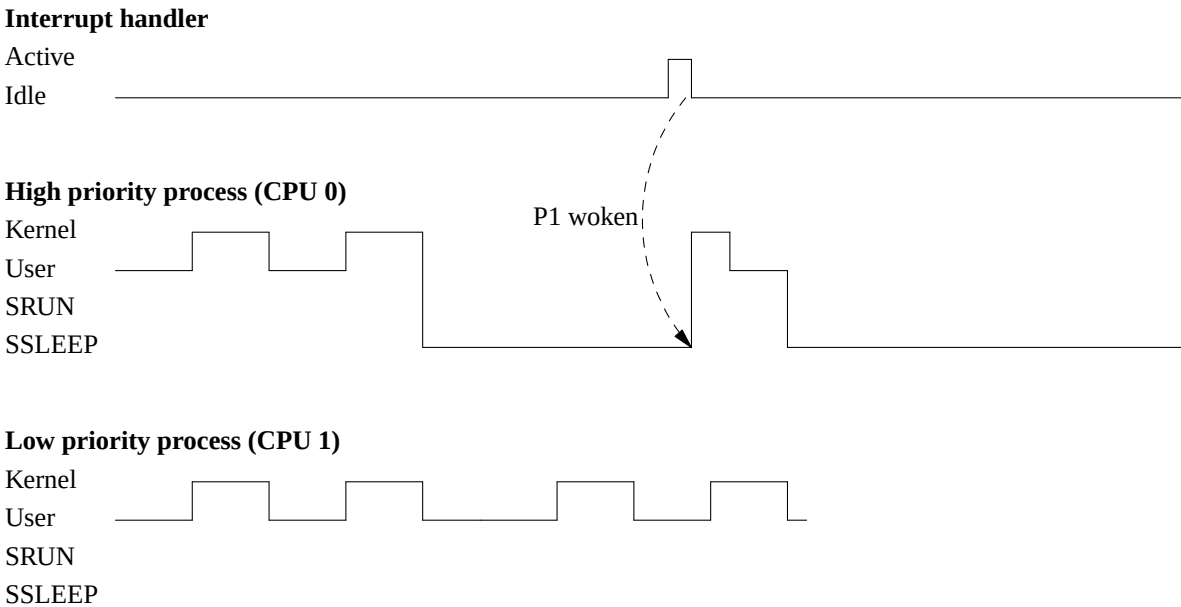
**Danger**: If a process does stay in the kernel for an extended period of time, it can cause significant performance degradation or even hangs.




# Real single processor scheduling



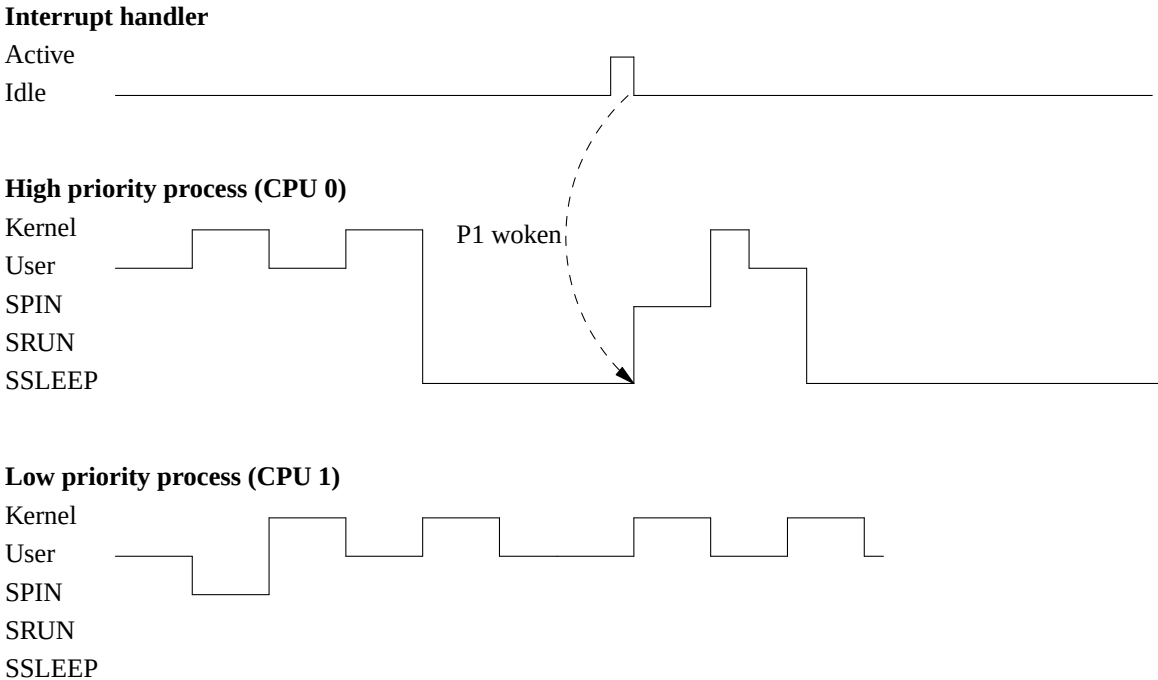

# Ideal single processor scheduling




# 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

**Process in CPU 0**

Kernel
User
SPIN

**Process in CPU 1**

Kernel
User
SPIN

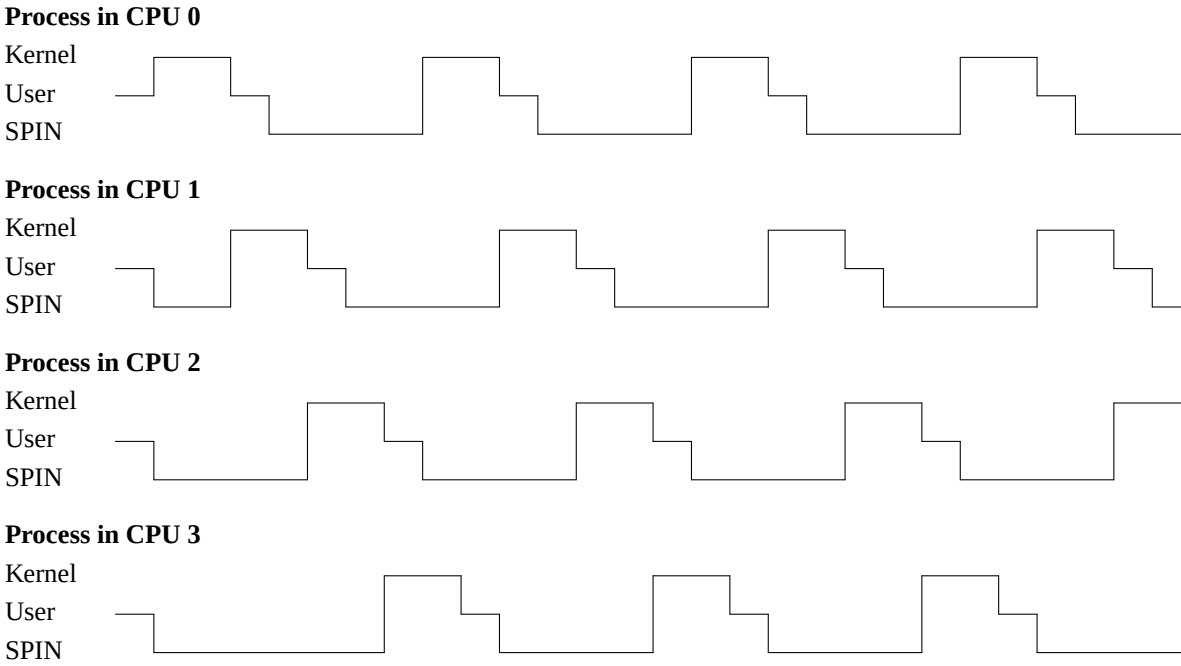
**Process in CPU 2**

Kernel
User
SPIN

**Process in CPU 3**

Kernel
User
SPIN

# 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 49.6  0.0      0    0  ??  RL    2:10PM   5:45.35  (idle: cpu1)
root      11 48.4  0.0      0    0  ??  WL    2:10PM   5:45.34  (idle: cpu0)
root      12  0.0  0.0      0    0  ??  WL    2:10PM   0:01.09  (softinterrupt)
root      13  0.0  0.0      0    0  ??  WL    2:10PM   0:00.00  (irq14: ata0)
root      14  0.0  0.0      0    0  ??  WL    2:10PM   0:00.00  (irq15: ata1)
root      15  0.0  0.0      0    0  ??  WL    2:10PM   0:00.05  (irq3: dc0)
root      16  0.0  0.0      0    0  ??  WL    2:10PM   0:00.05  (irq10: ahc0)
root      17  0.0  0.0      0    0  ??  WL    2:10PM   0:00.00  (irq11: atapci1+)
root      18  0.0  0.0      0    0  ??  WL    2:10PM   0:00.01  (irq1: atkbd0)
root      19  0.0  0.0      0    0  ??  WL    2:10PM   0:00.00  (irq12: psm0)
root      20  0.0  0.0      0    0  ??  WL    2:10PM   0:00.00  (irq7: ppc0)
root      21  0.0  0.0      0    0  ??  WL    2:10PM   0:01.44  (irq0: clk)
root      22  0.0  0.0      0    0  ??  WL    2:10PM   0:01.36  (irq8: rtc)
```

# 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 mutex

```
struct mtx {
        volatile u_int  mtx_lock;       /* lock owner/gate/flags */
        volatile u_short mtx_recurse;   /* number of recursive holds */
        u_short         mtx_f1;             /* flags */
        u_int           mtx_savefl;     /* saved flags (for spin locks) */
        char            *mtx_description;  /* name */
        TAILQ_HEAD(, proc) mtx_blocked; /* list of waiters */
        LIST_ENTRY(mtx) mtx_contested;
        struct mtx      *mtx_next;      /* all locks in system */
        struct mtx      *mtx_prev;
};
```

# mutex forms

- Described in *mutex(9)*
- Adaptive lock: Set flag `MTX_DEF` (default).
- Spin lock: Set flag `MTX_SPIN`.
- Sleep lock: Set flags `MTX_DEF` and `MTX_NOSPIN`.
- Many flag definitions taken from BSD/OS are currently unused.
- Currently no semaphores or read/write locks.

# Condition variables

- Currently no prototypical condition variables.
- Same functionality available from the `msleep` function: enter holding a mutex.
- The mutex will be released before sleeping and reacquired on wakeup.
- Similar to the behaviour of `tsleep` with `splx` functions.
- `tsleep` reimplemented as a macro calling `msleep` with null mutex.

# Original locks

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

# Current locks

- `clock_lock` protects low-level time manipulation routines.
- `random_reseed`, `random_harvest`. Both used by the kernel random number generator.
- `vm86pcb_lock`
- `malloc_mutex`
- `w_mtx`
- `eventhandler_mutex`
- `mmbfree.m_mtx`
- `mclfree.m_mtx`
- `mcntfree.m_mtx`

# 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

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

These slides are available at
`http://echunga.linuxcare.com.au/SMPng/`