# Locking

**Youjip Won**

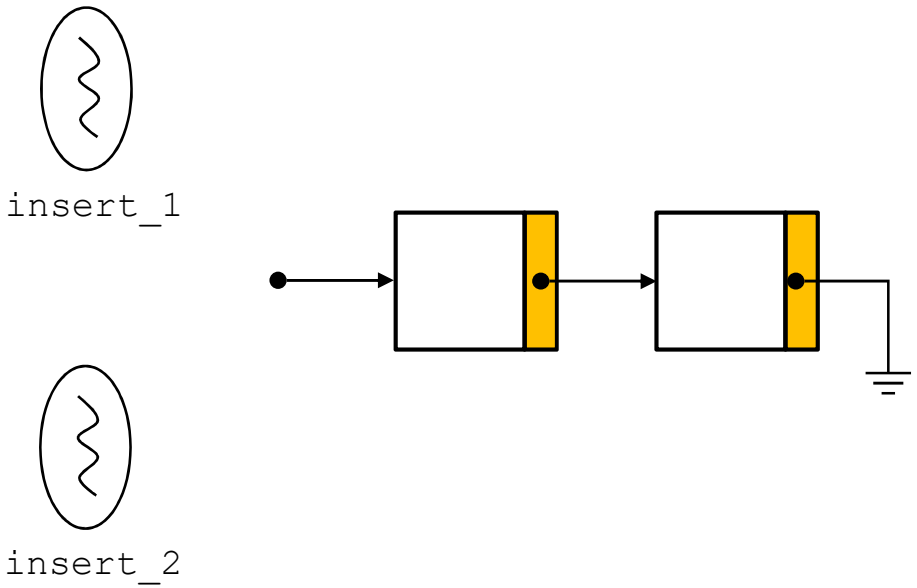
KAIST

# Contents

- Outline
- Race condition
- Spinlock
- Using lock
- Deadlock
- Interrupt handler
- __sync_synchronize
- Sleep Lock
- Limitations of locks

# Race condition

- Situation in which a memory location is accessed concurrently, and at least one access is a write

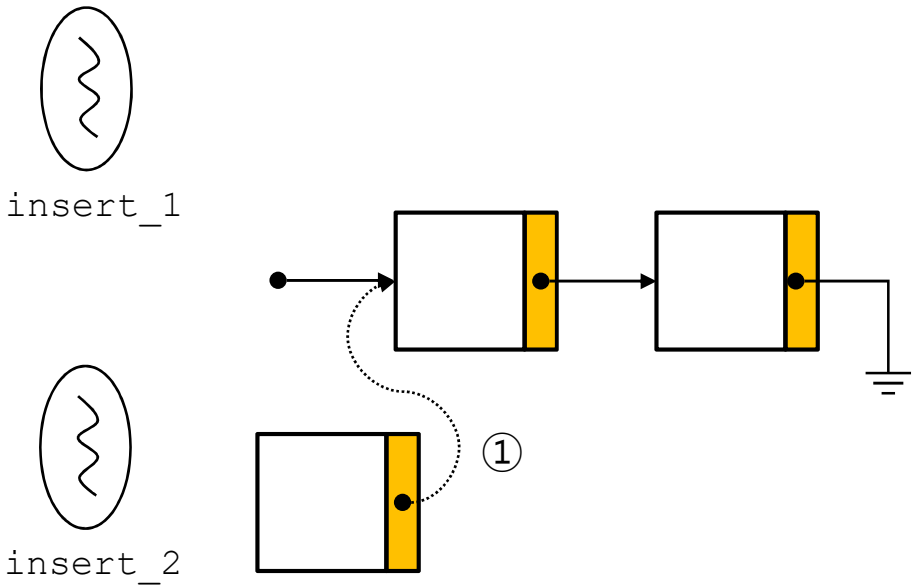



Concurrency in `insert` operation

```
struct list {
  int data;
  struct list *next;
};

struct list *list = 0;

void
insert(int data)
{
  struct list *l;

  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
}
```

# Race condition (Cont.)

- Situation in which a memory location is accessed concurrently, and at least one access is a write

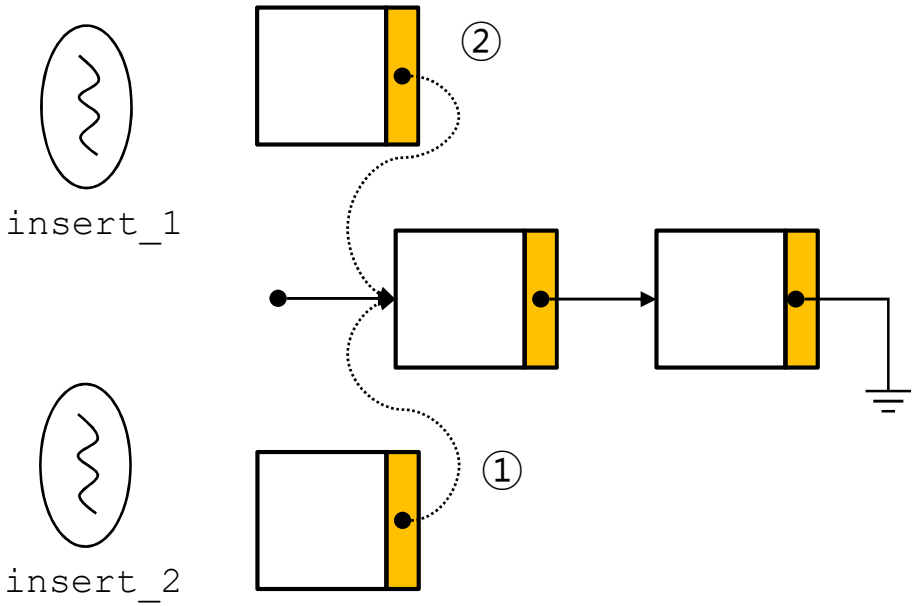



Concurrency in `insert` operation

```
struct list {
  int data;
  struct list *next;
};

struct list *list = 0;

void
insert(int data)
{
  struct list *l;

  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
}
```

# Race condition (Cont.)

- Situation in which a memory location is accessed concurrently, and at least one access is a write

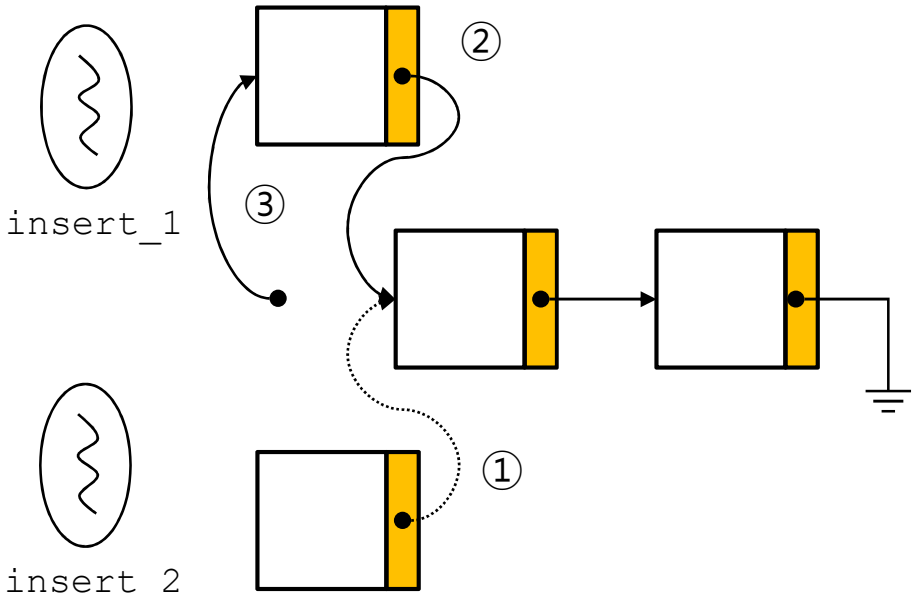



Concurrency in `insert` operation

```
struct list {
  int data;
  struct list *next;
};

struct list *list = 0;

void
insert(int data)
{
  struct list *l;

  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
}
```

# Race condition (Cont.)

- Situation in which a memory location is accessed concurrently, and at least one access is a write

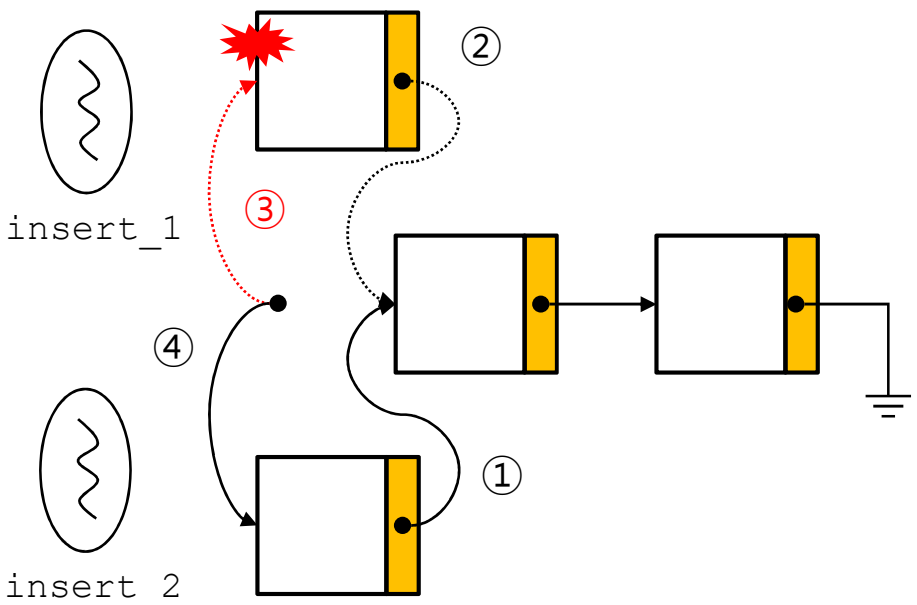



Concurrency in `insert` operation

```
struct list {
  int data;
  struct list *next;
};

struct list *list = 0;

void
insert(int data)
{
  struct list *l;

  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
}
```

# Race condition (Cont.)

- Situation in which a memory location is accessed concurrently, and at least one access is a write



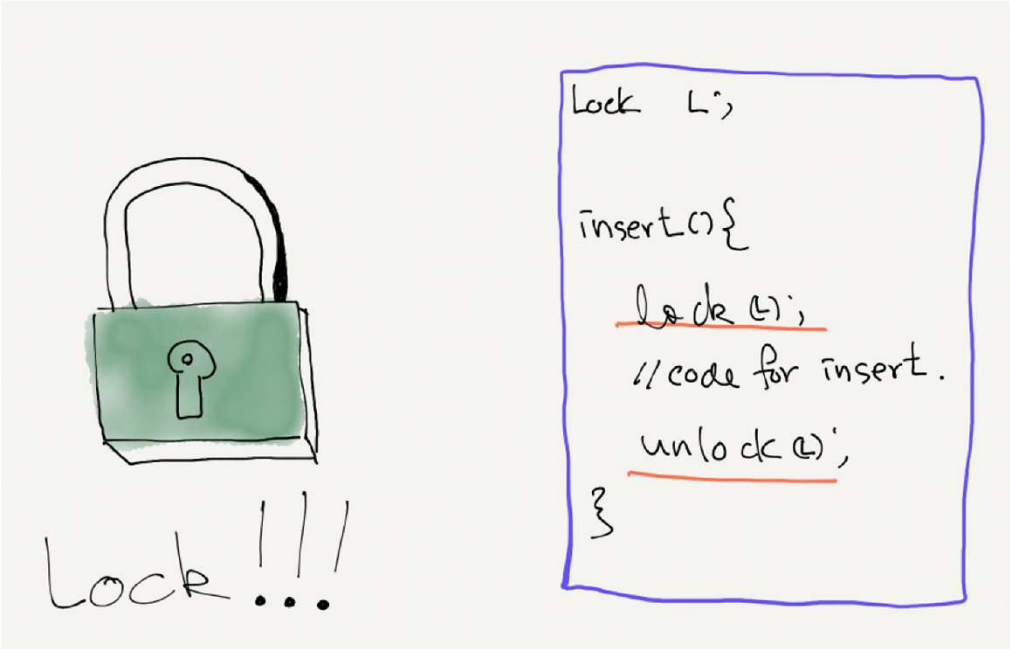

Concurrency in `insert` operation

```
struct list {
  int data;
  struct list *next;
};

struct list *list = 0;

void
insert(int data)
{
  struct list *l;

  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
}
```

# Race condition (Cont.)

- Race conditions can be solved by locking

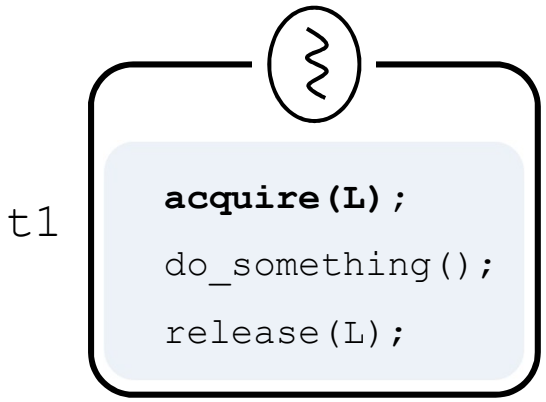
```
struct list *list = 0;
struct lock listlock;

void
insert(int data)
{
  struct list *l;

  acquire(&listlock);
  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
  release(&listlock);
}
```

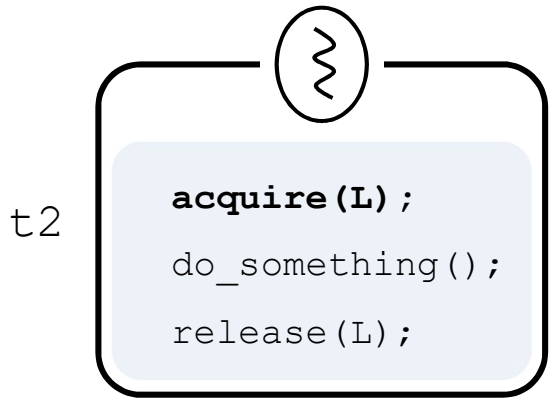


# Sleep lock vs. spin lock

```
void acquire(struct spinlock *lk) {
  for(;;) {
    if(!lk->locked) {
      lk->locked = 1;
      break;
    }
  }
}
```

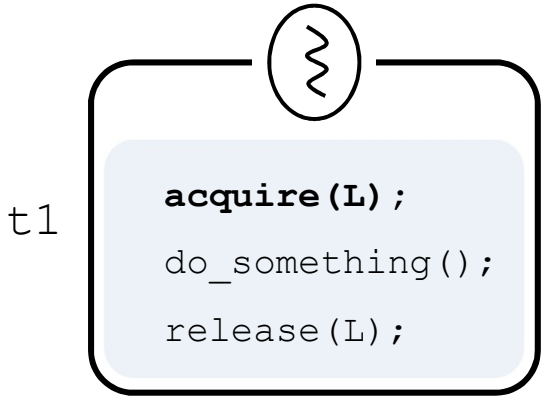
should be atomic..!

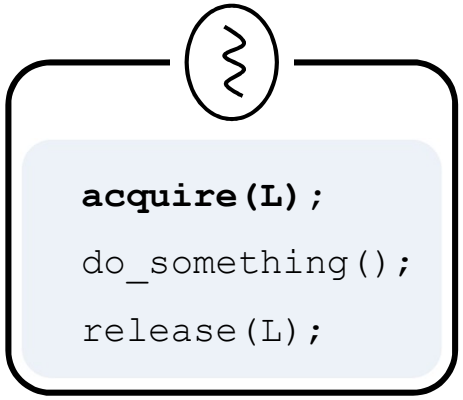



```
void acquire(struct spinlock *lk) {
    for (;;) {
        if (!lk->locked) {
            lk->locked = 1;
            break;
        }
    }
}
```

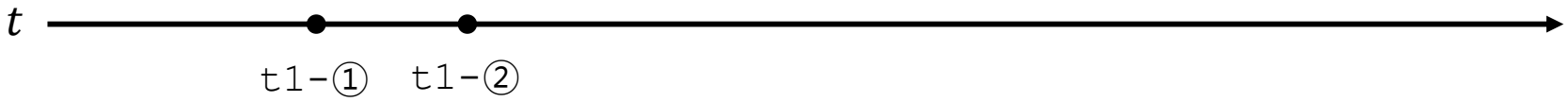
```
void acquire(struct spinlock *lk) {
    for (;;) {
        if (!lk->locked) {
            lk->locked = 1;
            break;
        }
    }
}
```

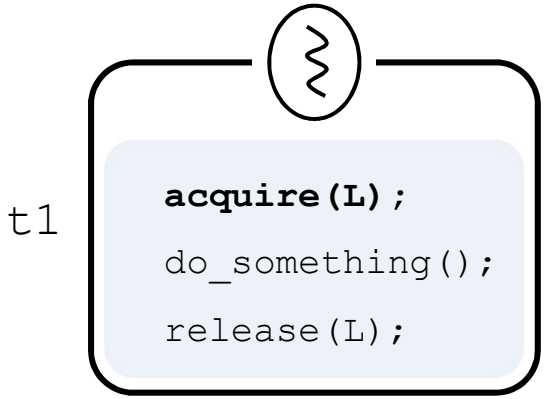


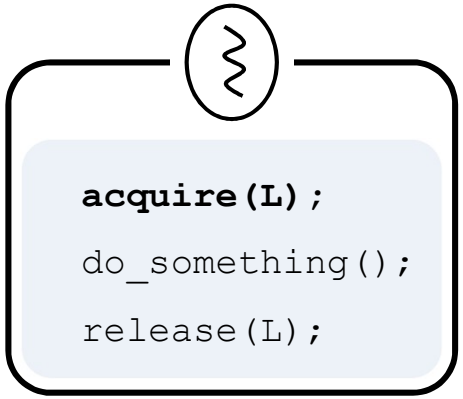




```
void acquire(struct spinlock *lk) {
    ① for (;;) {
    ② if (!lk->locked) {
            lk->locked = 1;
            break;
        }
    }
}
```

```
void acquire(struct spinlock *lk) {
    for (;;) {
        if (!lk->locked) {
            lk->locked = 1;
            break;
        }
    }
}
```

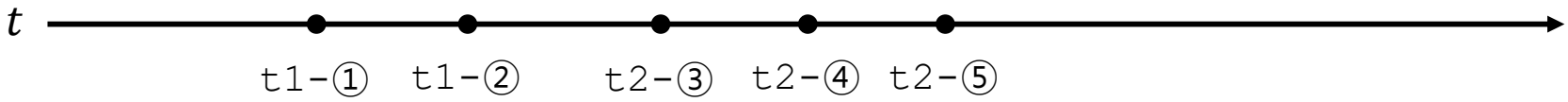





```
void acquire(struct spinlock *lk) {
  ① for (;;) {
      ② if (!lk->locked) {
            lk->locked = 1;
            break;
        }
    }
}
```

```
void acquire(struct spinlock *lk) {
  ③ for (;;) {
      ④ if (!lk->locked) {
          ⑤ lk->locked = 1;
            break;
        }
    }
}
```



t1

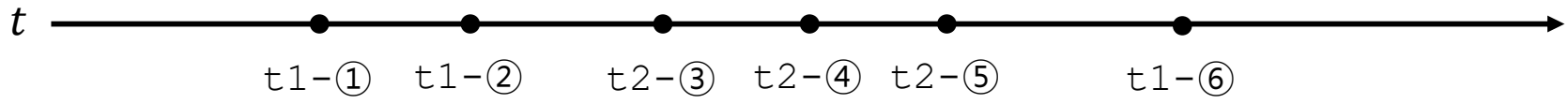
```
acquire(L);
do_something();
release(L);
```

t2

```
acquire(L);
do_something();
release(L);
```

```
void acquire(struct spinlock *lk) {
  ① for (;;) {
      ② if (!lk->locked) {
          ⑥ lk->locked = 1;
            break;
        }
    }
}
```

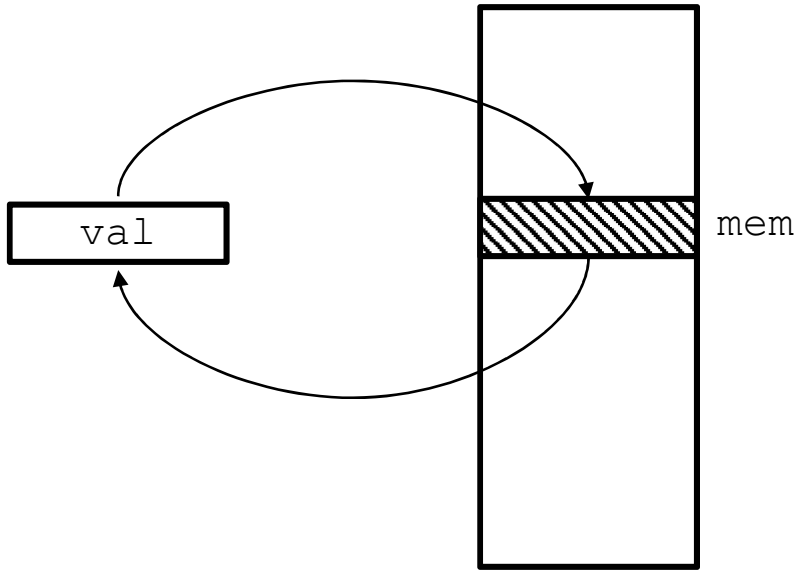
```
void acquire(struct spinlock *lk) {
  ③ for (;;) {
      ④ if (!lk->locked) {
          ⑤ lk->locked = 1;
            break;
        }
    }
}
```



# Atomic Instruction: `xchg`

```
xchg(mem, val);
```

1. Exchange
2. Return the value that was in  the memory



# Spinlock

- The xv6 has 2 types of locks: spinlock and sleep-lock
- Spinlock structure
- primitives
  - `acquire(struct spinlock *lk)`
  - `release(struct spinlock *lk)`

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of the lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

# `acquire` in spinlock

```
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor not to move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

# Spinlock (Cont.)

- `getcallerpcs(void *v, uint pcs[])`

```
void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp;
  int i;

  ebp = (uint*)v - 2;
  for(i = 0; i < 10; i++){
    if(ebp == 0 || ebp < (uint*)KERNBASE
        || ebp == (uint*)0xffffffff)
      break;
    pcs[i] = ebp[1];     // saved %eip
    ebp = (uint*)ebp[0]; // saved %ebp
  }
  for(; i < 10; i++)
    pcs[i] = 0;
}
```

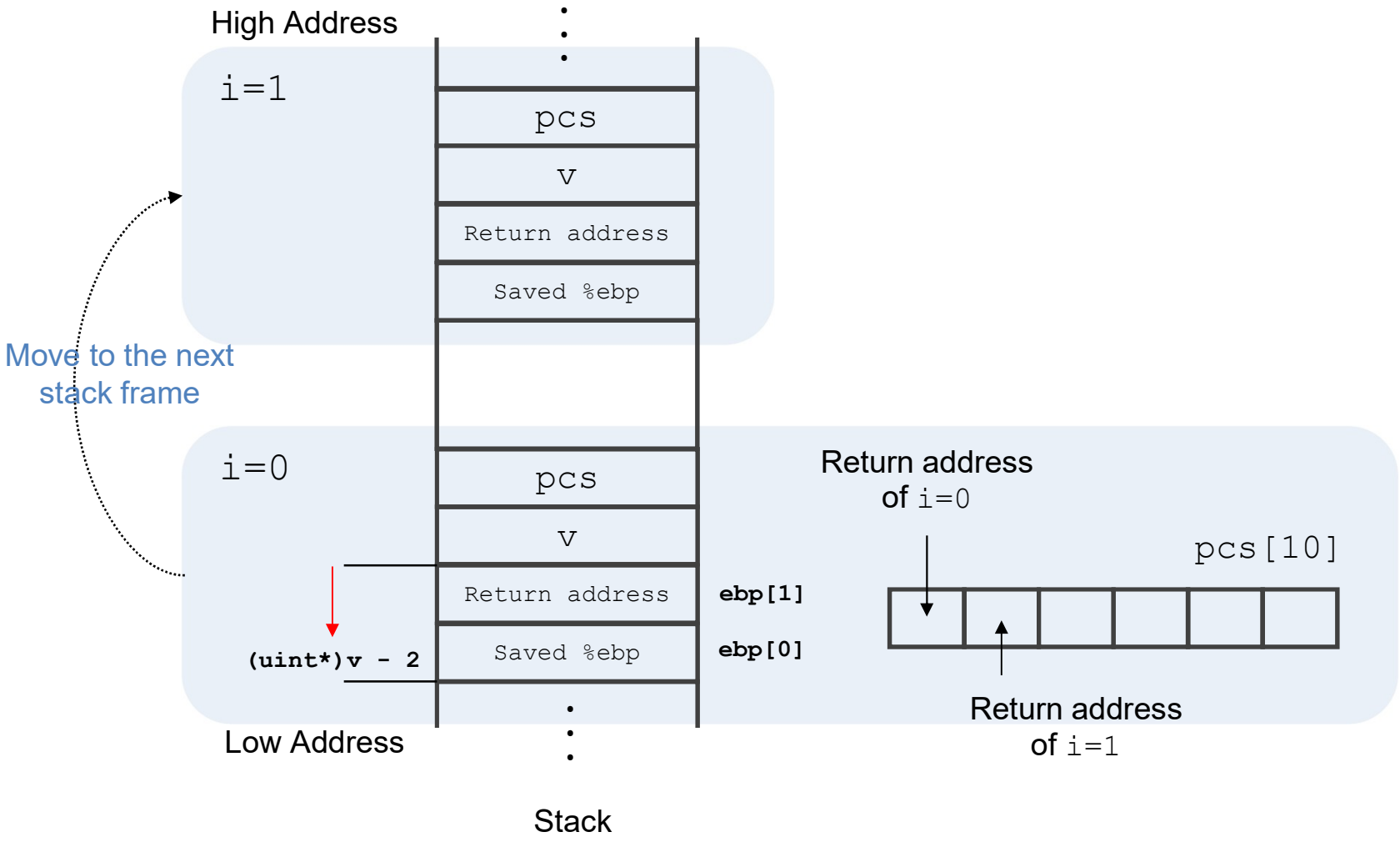
# Spinlock (Cont.)

- `getcallerpcs(void *v, uint pcs[])`

- `getcallerpcs()` follows the latest ten stack frames, and records the caller address (return address) to `pcs`.
- XV6 uses `getcallerpcs()` for debugging purpose.
- Unrecorded elements of `pcs` array are set to `0`.

# Spinlock (Cont.)

- `getcallerpcs (void *v, uint pcs[])`



# `release ()` in spinlock

```
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```

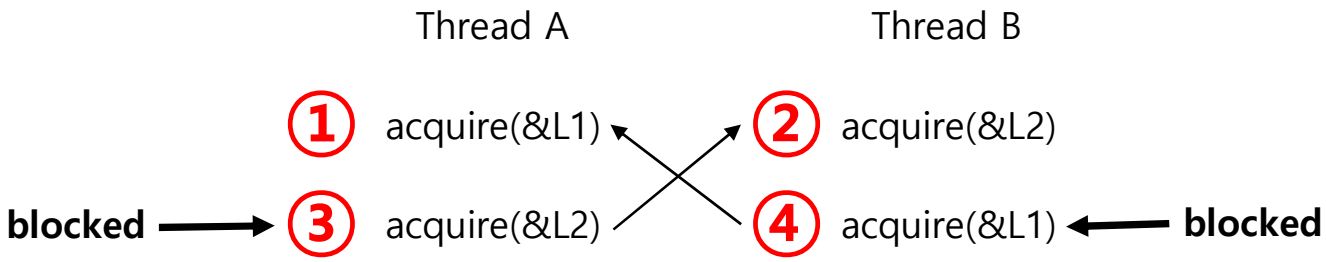
# Caution

- Locking should be used properly, not too much.
- Too many locks reduce parallelism.
- xv6 uses a few data-structure specific locks.

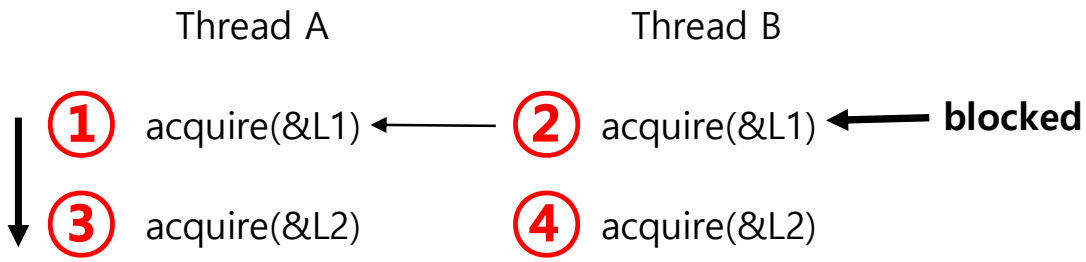
| Lock | Description |
|---|---|
| bcache.lock | Protects allocation of block buffer cache entries |
| cons.lock | Serializes access to console hardware, avoids intermixed output |
| ftable.lock | Serializes allocation of a struct file in file table |
| icache.lock | Protects allocation of inode cache entries |
| idelock | Serializes access to disk hardware and disk queue |
| kmem.lock | Serializes allocation of memory |
| log.lock | Serializes operations on the transaction log |
| pipe's p->lock | Serializes operations on each pipe |
| ptable.lock | Serializes context switching, and operations on proc->state and proctable |
| tickslock | Serializes operations on the ticks counter |
| inode's ip->lock | Serializes operations on each inode and its content |
| buf's b->lock | Serializes operations on each block buffer |

# Deadlock

- Deadlock: The situation that two threads require each other's lock and wait indefinitely.

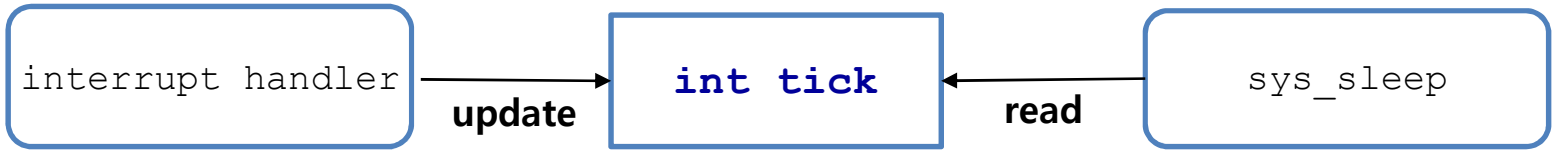



- To avoid deadlock, the callers must invoke functions in a way that causes locks to be acquired in the agreed-on order.



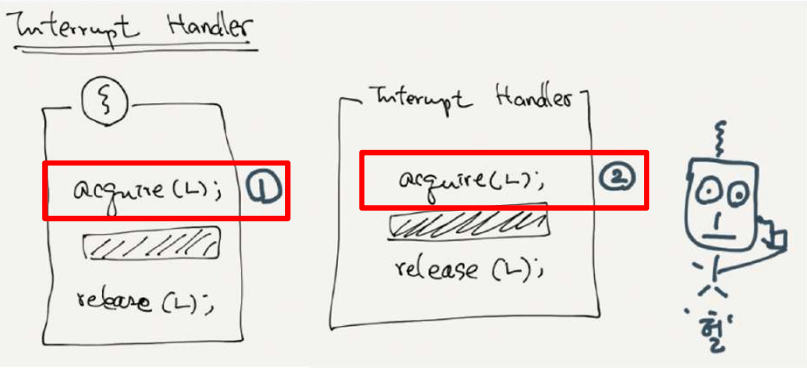
# Interrupt and lock

- Spinlocks are used by the interrupt handler as well as the user thread.




- **Important!!!**
  - You have to disable the interrupt when you hold the spinlock that is used by the interrupt handler.
  - It is difficult to determine whether the given lock is used by the interrupt handler or not.
  - XV6 blindly disables the interrupt when you hold the spinlock.
  - **Turn off interrupt before you acquire spinlock.**

# Interrupt and lock




```
void
iderw(struct buf *b)
{
  struct buf **pp;

  acquire(&idelock);  //DOC:acquire-lock

  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
      sleep(b, &idelock);

  release(&idelock);
```

```
void
ideintr(void)
{
  struct buf *b;

  // First queued buffer is the active request.
  acquire(&idelock);
```

# Interrupt handler

```
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
```

Disable interrupt.

Enable interrupt.

# Disable and enable interrupt

- `pushcli/popcli` are like `cli/sti` except that they are matched: it takes two `popcli` to undo two `pushcli`.
- If interrupts are off, then `pushcli`, popcli leaves them off.

```
void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

```
#define FL_IF           0x00000200      // Interrupt Enable
```

# Instructions and Memory Ordering

```
i = 1;
j = j * 2 ;
```

```
j = j * 2;
i = 1;
```

- Compilers may reorder instructions.
- Many complier and processor execute code out of order to achieve higher performance.

# Instructions and Memory Ordering (Cont.)

- There is bad example. Line 6 can runs before Line 3, 4, 5 by other cores.

```
acquire(&listlock);
l = malloc(sizeof *l);
l->data = data;
l->next = list;
list = l;
release(&listlock);
```

# __sync_synchronize

- `__sync_synchronize` tell the hardware and compiler not to perform re-orderings.
- `__sync_synchronize` can guarantees execution order in critical section by being called in acquire and release.

```
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();
```

# __sync_synchronize (Cont.)

```
acquire(&listlock);
----------------------
l = malloc(sizeof *l);

l->data = data;

l->next = list;

list = l;
----------------------
release(&listlock);
```

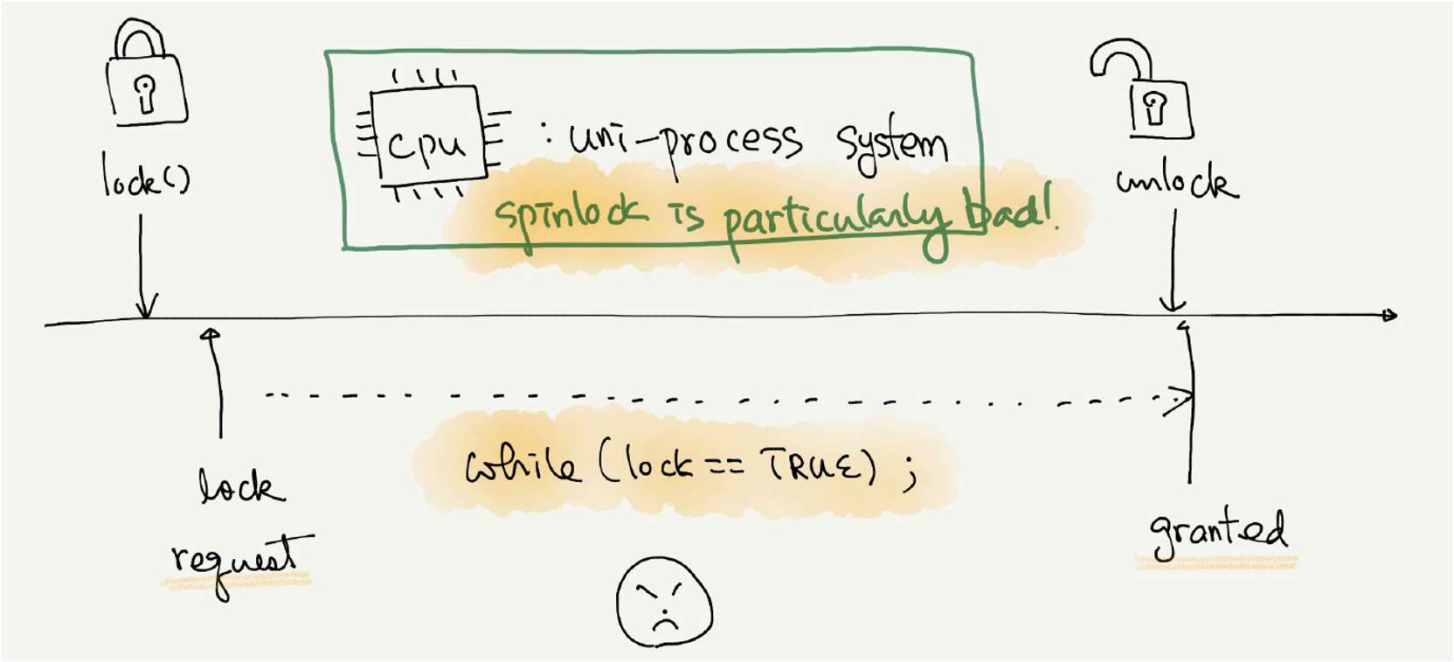
__sync_synchronize()

__sync_synchronize()

# Sleep lock

- Sometimes a process needs to hold a lock for a long time.
- When a thread that wants to hold lock is looping, another thread that is holding lock cannot release lock.

- Sleep lock: yield CPU to others while waiting for the lock!



# Data structure and API's for sleep lock

```
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};
```

For debugging

- lock: put itself into sleep.
- unlock: wake up one of the processes that have been waiting for the lock.

# process structure for the sleep lock

- specify the lock which it is waiting for. : called **sleep channel**.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

# sleep lock

- `acquiresleep`

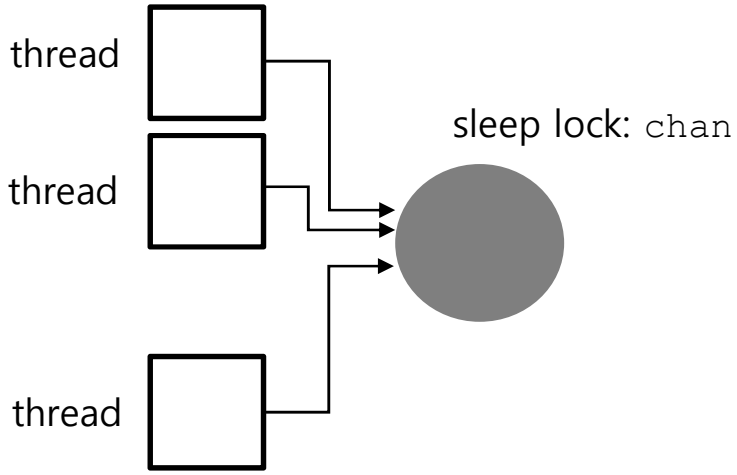
going to sleep with holding a lock?

```
void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
```

# sleep()

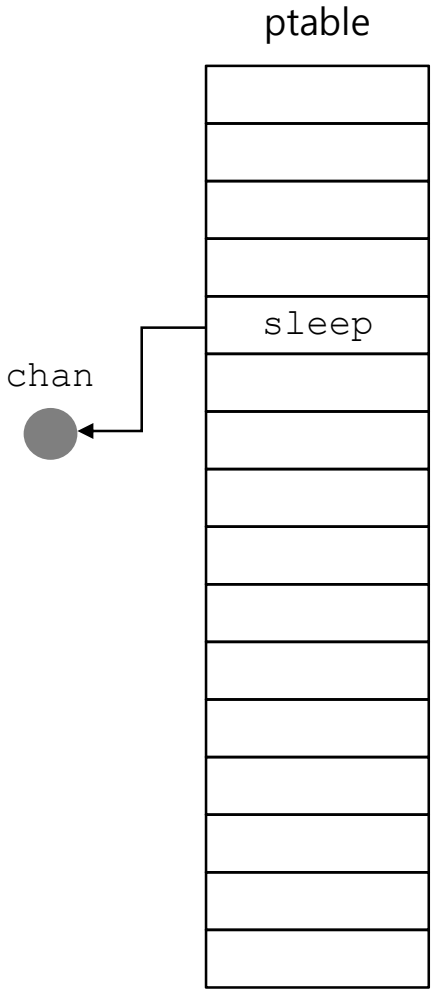
```
void sleep(void *chan, struct spinlock *lk)
```

Place the caller to `chan`, release `lk` and put the state of the caller process to sleep.



# peek on `sleep()`

```
void sleep(void *chan, struct spinlock *lk) {
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");
 if(lk == 0)
    panic("sleep without lk");
 if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  // Lock the page table.
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;
  sched();
  p->chan = 0;
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```



# sleep unlock

- releasesleep
  - Clears the sleeplock lk.
  - Wakes up all sleeping processes by calling wakeup().

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

# wakeup1 vs. wakeup

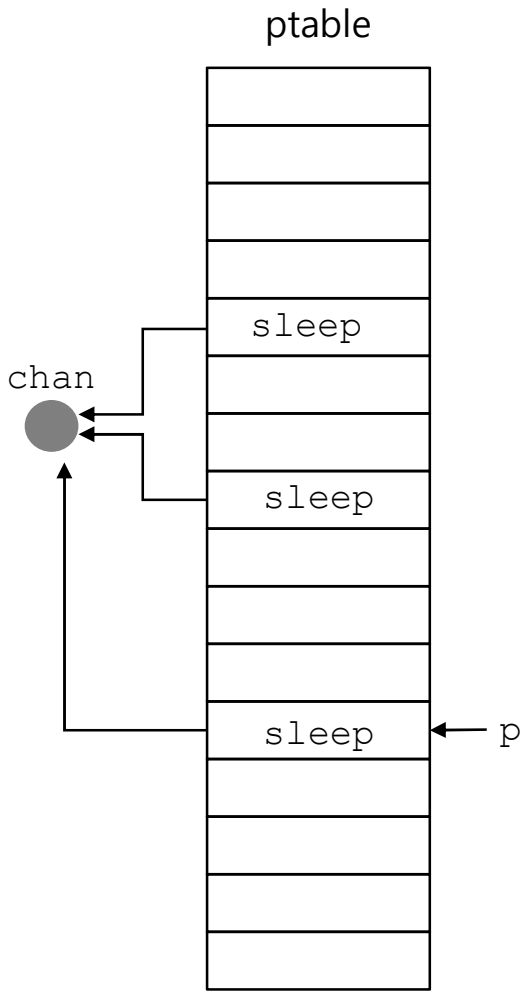
```
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

Lock the process table and call `wakeup1()`.

After `wakeup1()`, unlock the process table.

# wakeup1 vs. wakeup

```
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;                                          O(n)!!!
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}
```

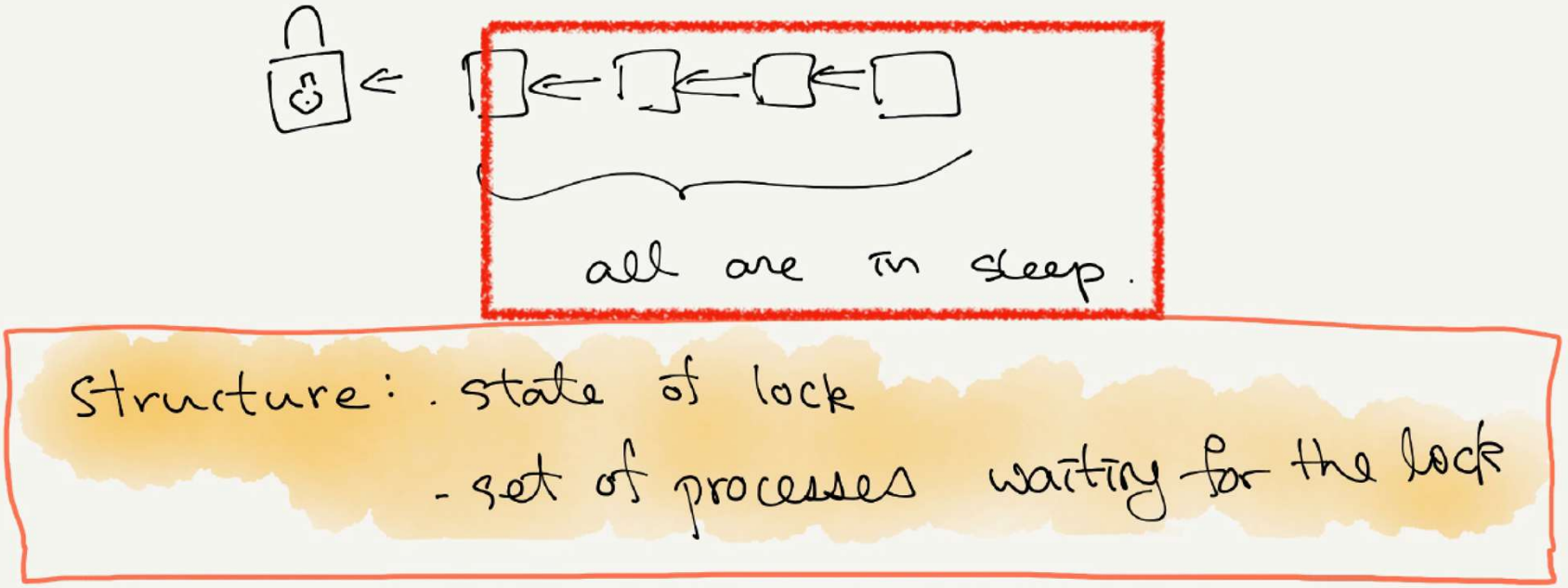



Wake up all processes that wait for the channel.

All are put into ready state. Only one of them actually runs.

Why do we wake up all processes that wait for the channel?

# Better Sleep lock design




- lock: If lock is not available, insert itself to the linked list and 'sleep'.
- unlock: scan the linked list and wake one up.

Sleeplock inside spinlock: No!

```
func1() {
    spinlock(L);
    func2();
    spinunlock(L);
}
```

interrupt ⇒ disabled

```
func2() {
    sleeplock(L2);
    ttt;
    sleepunlock(L2);
}
```


# lock ordering (mm/filemap.c)

```
/*
 * Lock ordering:
 *
 *  ->i_mmap_rwsem      (truncate_pagecache)
 *    ->private_lock        (__free_pte->__set_page_dirty_buffers)
 *      ->swap_lock         (exclusive_swap_page, others)
 *        ->i_pages lock
 *
 *  ->i_mutex
 *    ->i_mmap_rwsem        (truncate->unmap_mapping_range)
 *
 *  ->mmap_sem
 *    ->i_mmap_rwsem
 *      ->page_table_lock or pte_lock (various, mainly in memory.c)
 *        ->i_pages lock    (arch-dependent flush_dcache_mmap_lock)
 *
 *  ->mmap_sem
 *    ->lock_page       (access_process_vm)
 *
 *  ->page_table_lock or pte_lock
 *    ->swap_lock       (try_to_unmap_one)
 *    ->private_lock        (try_to_unmap_one)
 *    ->i_pages lock        (try_to_unmap_one)
 *    ->zone_lru_lock(zone)     (follow_page->mark_page_accessed)
 *    ->zone_lru_lock(zone)     (check_pte_range->isolate_lru_page)
 *    ->private_lock        (page_remove_rmap->set_page_dirty)
 *    ->i_pages lock        (page_remove_rmap->set_page_dirty)
 *    bdi.wb->list_lock         (page_remove_rmap->set_page_dirty)
 *    ->inode->i_lock       (page_remove_rmap->set_page_dirty)
 *    ->memcg->move_lock    (page_remove_rmap->lock_page_memcg)
 *    bdi.wb->list_lock         (zap_pte_range->set_page_dirty)
 *    ->inode->i_lock       (zap_pte_range->set_page_dirty)
 *    ->private_lock        (zap_pte_range->__set_page_dirty_buffers)
```

# Limitations of locks

- Function uses a data that needs to be protected by a lock.
  - The caller may hold a lock.
  - The caller may not hold a lock.
- Solution
  - Function with a lock and function without a lock: wakeup1 and wakeup
  - Ask caller to hold lock always.: sched
  - Recursive lock: the caller hold the lock and callee re-acquire the lock.

# sched()

A process calls `sched()` to yield CPU.

Hand CPU over to other processes.

Put itself to the ready queue.

Cases

- A process uses up its time quantum.
- A process goes to sleep.
- A process exits.

# peek on sched()

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1) {
    cprintf("panic! ncli: %d\n", mycpu()->ncli);
    panic("sched locks");
  }
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

# Importance of system software and os

## Toyota's killer firmware: Bad design and its consequences

Michael Dunn -October 28, 2013

129 Comments

- Toyota's electronic throttle control system (ETCS) source code is of unreasonable quality.
- Toyota's source code is defective and contains bugs, including bugs that can cause unintended acceleration (UA).
- Code-quality metrics predict presence of additional bugs.
- Toyota's fail safes are defective and inadequate (referring to them as a *"house of cards" safety architecture).*
- Misbehaviors of Toyota's ETCS are a cause of UA.


FUEL CELL CARS

## Toyota calls back all the Mirais for software bug

It only affects about 2,800 vehicles around the world.

BY ANDREW KROK | FEBRUARY 16, 2017 11:19 AM PST

## What is the programming language u jets?

Ad by JetBrains

**Level up your code with IntelliJ IDEA.**

An IDE built for professional development. Make developing enjo

Free trial at jetbrains.com

### 6 Answers

Ajay Pandey, Equal Opportunity Skull Buster

Answered Jun 23 2018 · Author has **87** answers and **304.8k** answe

I am saddened reading this answer from a narrative builder what
experts in Quora. I can't resist to write a counter instead of an ans

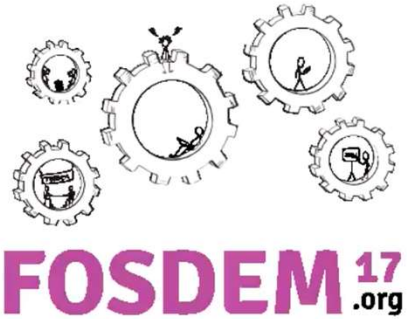
Programming language doesn't really matter. Giving emphasis to
language is just a non-expert attempt to add credibility. I am not saying C++ is not powerful. I myself is a C++ programmer for 24 years and saw the evolution from C to C++98 up to all modern standards of C++11/14/17.

Domain driven and specially designed propriety languages starts taking over any complex domain. Specially when you can meet the program-ability requirement of hardware by porting/cross-compilation or automatic generation of code. I myself programmed in Matlab and designed systems in Simulink and auto generated C++ code to feed into Hardware; once simulated system succeeds.

The series of false claims is revealing that the poster appears don't have any experience in basic Embedded systems. Expecting aviation and defense specific expertise is way too much.

Here is small list from Ahmed Siddique's answer:

1. ***Most of the fighters jets use Ada.*** - Not true, initial libraries were developed in Ada hence leading defense vendors carried it as far as possible. Most Non-western systems are not Ada unless they purchased Ada component libraries under ToT agreement.
2. ***Pakistan air force made a unique decision. Instead of using ADA programming language they used C++.*** - Not a unique decision, C++ is widely used language for interaction with Hardware even in defense applications.
3. ***Because there were literally tens of thousands of young highly***




Using Linux in Air Traffic Control

Hardware and Operating System Platforms

Gerolf Ziegenhain

Janson

2017-02-05


London City Airport and NATS to introduce the UK's first digital air traffic control tower