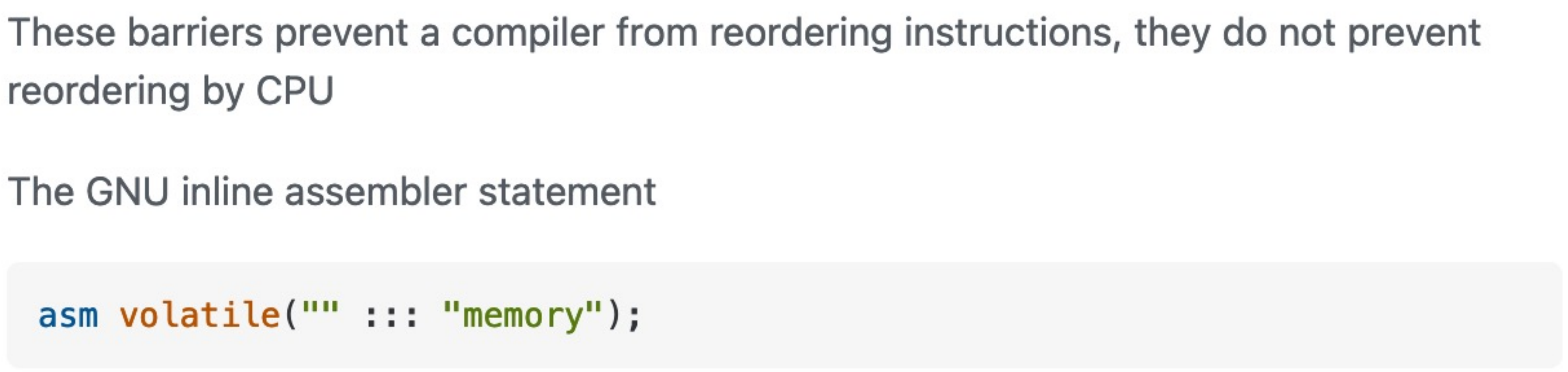
Summary of Lecture 5

- Three ways to get from user code to kernel code:
  - System calls, interrupts, exceptions
  - All mediated through the Interrupt Descriptor Table
- Interrupt handler and other code shares various stuff:
  - Register, EFLAGS (push on stack)
  - Memory (use a separate stack)
  - Shared state:
    - Data structures (linked list example)
    - Device state (VGA example)
    - Memory (is_device_locked)

Summary of Lecture 5

- Solving these issues:
  - CLI / STI – disable interrupts
  - Define shared variables as volatile
  - Prevent compiler and ISA reorderings
    - Google: “memory barrier” and “serializing instruction”



These barriers prevent a compiler from reordering instructions, they do not prevent reordering by CPU

The GNU inline assembler statement

```
asm volatile("" ::: "memory");
```

CPUID—CPU Identification

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	ZO	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.[1] The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-8 shows information returned, depending on the initial value loaded into the EAX register.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using some Intel processors, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* Returns Extended Topology Enumeration leaf. *)[2]
CPUID.EAX =1FH (* Returns V2 Extended Topology Enumeration leaf. *)[2]
CPUID.EAX = 80000008H (* Returns linear/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0BH. *)
```

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers.

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

MFENCE—Memory Fence

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE F0 MFENCE	ZO	V/V	SSE2	Serializes load and store operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction.[1] The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

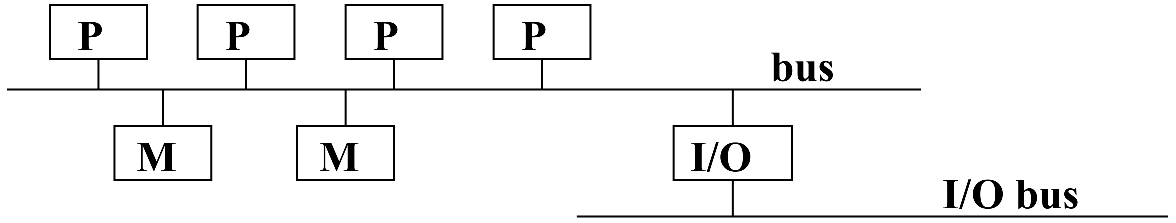
Specification of the instruction's opcode above indicates a ModR/M byte of F0. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, MFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 0-7.

Lecture Topics

- Multiprocessors and locks
- Spin locks

Multiprocessors and Locks

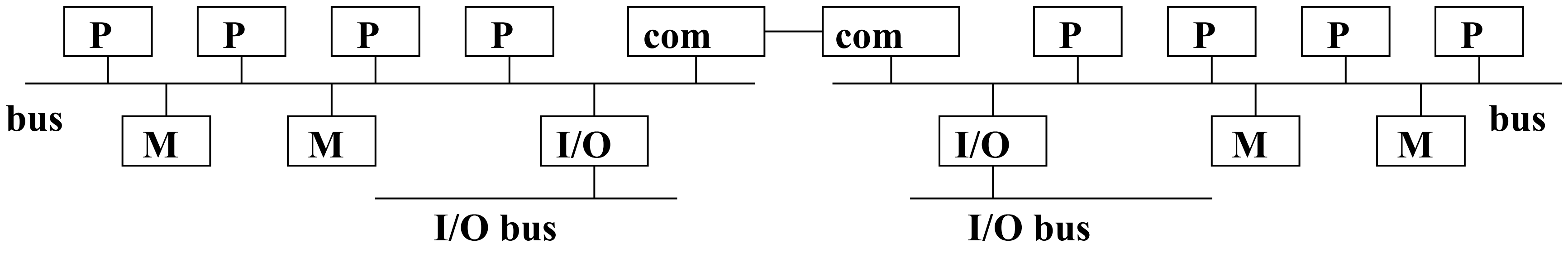
- We solved the critical section problem for uniprocessors
- What about multiprocessors?
  - *CLI … critical section …. STI*
- What is a multiprocessor?
  - usually a symmetric multiprocessor (SMP)



  - symmetric aspect: all processors have equal latency to all memory banks
  - multicore processors are similar from our perspective

Multiprocessors and Locks (cont.)

- Some non-uniform memory architecture (NUMA) machines were built

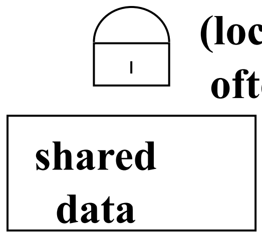


Multiprocessors and Locks (cont.)

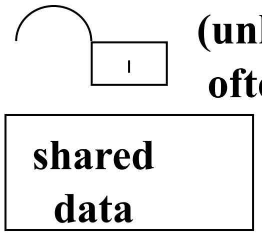
- Multithreaded code is not protected by ***IF***
- Why haven't we solved the atomicity problem on multiprocessors?
  - interrupts are masked if ***IF*** is cleared!
  - answer: ***IF*** is not cleared on other processors!
  - just tell other processors to clear IF, too?
    - too slow
    - requires an interrupt!
- We need to use shared memory to synchronize…

Multiprocessors and Locks (cont.)

- Logically, we use a lock
  - when we want to access a piece of shared data we first look at the lock
  - if it's locked, we wait until it's unlocked



(locked state is often represented as 1)

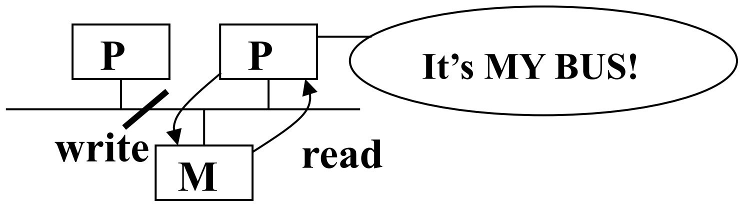


(unlocked state is often represented as 0)

  - once it's unlocked
    - we lock it
    - access the data
    - then unlock it

Multiprocessors and Locks (cont.)

- Locking must be atomic with respect to other processors!



INCL (%EAX)

↓

read EAX
add 1
put it back

LOCK: Prefix - execute following instruction with bus locked

LOCK INCL (%EAX)

Spin Locks

- The simplest lock
  - spin lock
  - lock op

```
do {
    try to change lock variable from 0 to 1
} while (attempt failed);
```

  - other work to do? ignore it!
  - spin in a tight loop on the lock (hence the name)
- Once successful, program/interrupt handler owns the lock

Spin Locks (cont.)

- Only the owner can unlock
  - How?
    - (change lock variable to 0)
  - Need to be atomic?
    - (no, only owner can change when locked)
  - What about memory op reordering?
    - (must be avoided!)
- Why keep asking the last question?
  - reordering leads to subtle bugs
  - unlikely to happen often
  - unlikely to be repeatable
  - very hard to find!

Linux Spin Lock API

- Static initialization

```
static spinlock_t  a_lock = SPIN_LOCK_UNLOCKED;
```

- Dynamic initialization

```
spin_lock_init (&a_lock);
```

- When is dynamic initialization safe?
  - lock must not be in use (race condition!)
  - other synchronization method must prevent use

Linux Spin Lock API – Basic Functions

```
void spin_lock (spinlock_t* lock);

void spin_unlock (spinlock_t* lock);
```

Linux Spin Lock API – Testing Functions

```
int spin_is_locked (spinlock_t* lock);
```

returns 1 if held, 0 if not, but beware of races!

```
int spin_trylock (spinlock_t* lock);
```

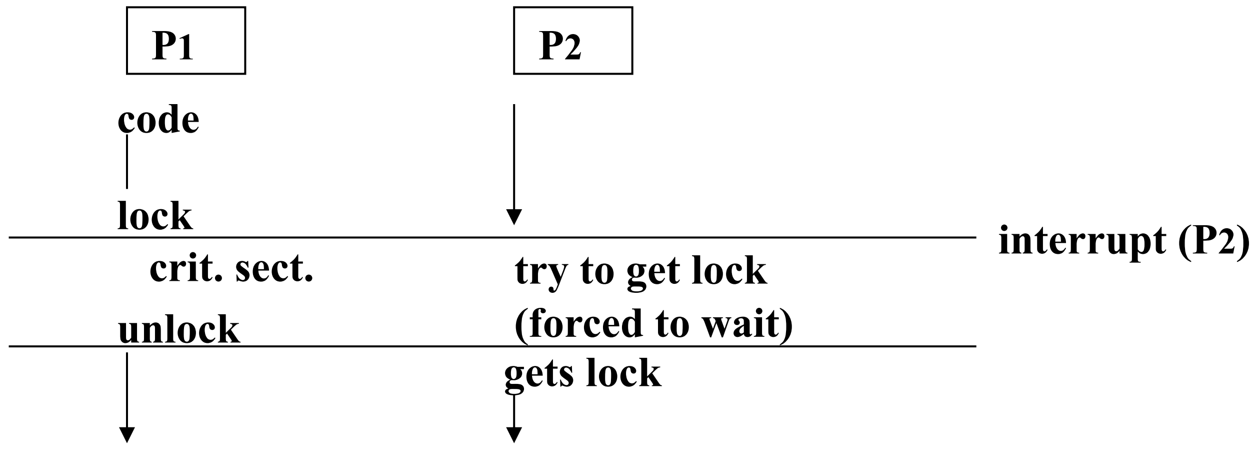
make one attempt; returns 1 on success, 0 on failure

```
void spin_unlock_wait (spinlock_t* lock);
```

wait until available (race condition again!)

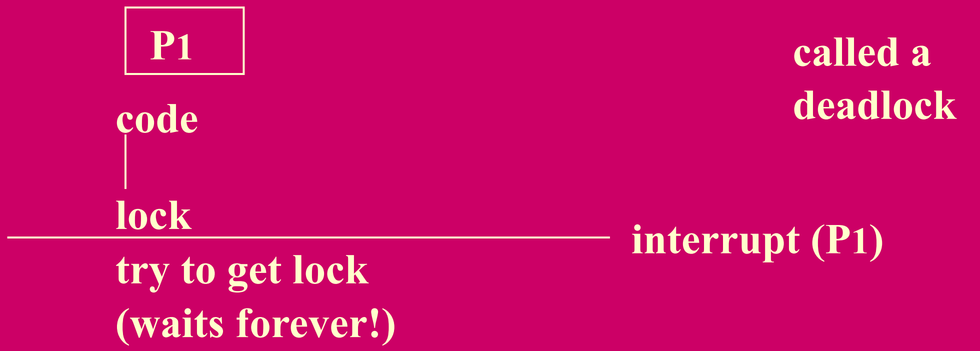
Linux Spin Lock API (cont.)

- Is spinlock enough to protect a critical section?



Linux Spin Lock API (cont.)

- What about ?



- Still need CLI/STI
- Which is first, CLI or lock?
  - CLI first
  - interrupt may occur between them, leading to scenario above

Linux' Lock/CLI Combo

```
static spinlock_t the_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;

spin_lock_irqsave (&the_lock, flags);
/* the critical section */
spin_unlock_irqrestore (&the_lock, flags);
```

```
asm volatile ("
        PUSHFL
        POPL %0
        CLI
" : "=g" (flags)      /* outputs  */
  :                   /* inputs   */
  : "memory"          /* clobbers */
);
spin_lock (&the_lock);
```

Linux' Lock/CLI Combo (cont)

```
spin_unlock (&the_lock);
asm volatile ("
    PUSHL %0
    POPFL
" :                    /* outputs   */
  : "g" (flags)        /* inputs    */
  : "memory", "cc"     /* clobbers */
);
```

Comments on code

- Notice that *spin_lock_irqsave* changes the flags argument (it's a macro)
- The "memory" argument
  - tells compiler that all memory is written by assembly block
  - prevents compiler from moving memory ops across assembly
- The "cc" argument
  - condition codes change
  - can lead to subtle bugs if left out!
- *spin_lock* and *spin_unlock* calls
  - become NOPs on uniprocessors
  - in that case, calls just change IF
- Restore rationale: may have had IF=0 on entry; if so, STI is unsafe

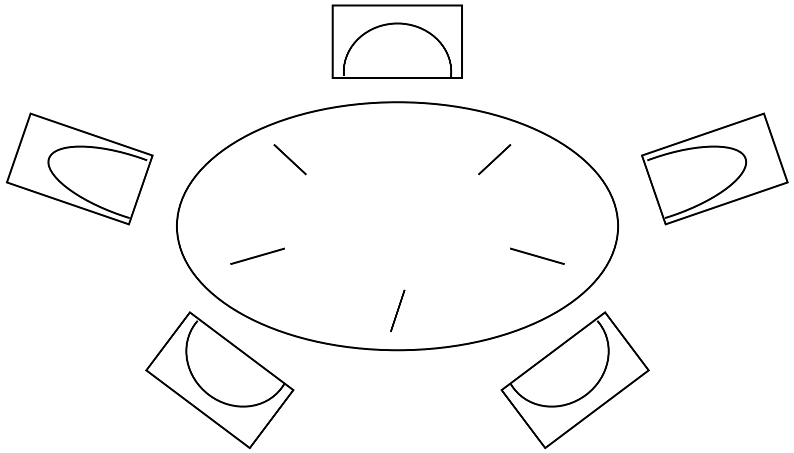
Lecture Topics

- Synch issues
- Conservative synchronization design
- Semaphores
- Reader/writer synchronization
- Selecting a synchronization mechanism

Another Philosophy Lesson

- Synchronization issues
  - five hungry philosophers
  - five chopsticks

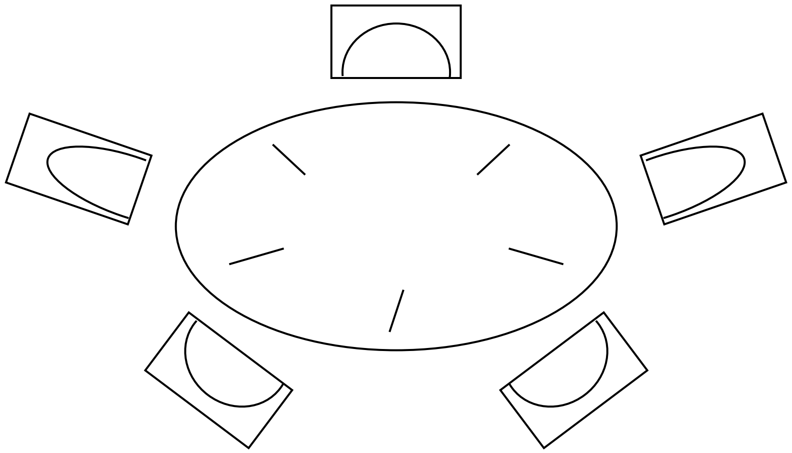
- Protocol
  - take left chopstick (or wait)
  - take right chopstick (or wait)
  - eat
  - release right chopstick
  - release left chopstick
  - digest
  - repeat



problems? deadlock!

Another Philosophy Lesson (cont.)

- How about the following protocol?
  - take left chopstick (or wait)
  - if right chopstick is free, take it
  - else release left chopstick and start over
  - eat
  - release right
  - release left
  - digest
  - repeat



- Does this work?

Another Philosophy Lesson (cont.)

- What if all philosophers act in lock-step (same speed)?

left	left	left	left	left
release	release	release	release	release
left	left	left	left	left
release	release	release	release	release
left	left	left	left	left

(ad infinitum)

- Called a livelock

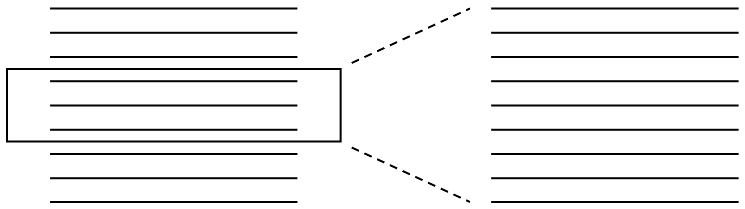
Another Philosophy Lesson (cont.)

- To solve the problem, need (partial) lock ordering
  - e.g., call chopsticks #1 through #5
  - protocol: take lower-numbered, then take higher-numbered
  - two philosophers try to get #1 first
  - can’t form a loop of waiting philosophers
  - thus someone will be able to eat

Conservative Synchronization Design

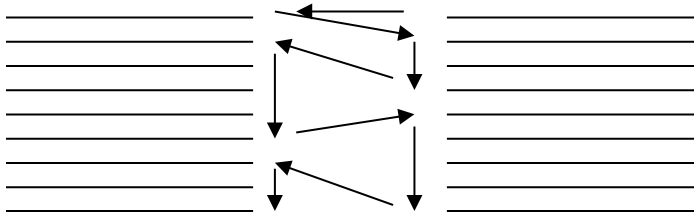
- Getting synchronization correct can be hard
  - it's the focus of several research communities

- On uniprocessor
  - mentally insert handler code between every pair of adjacent instructions
  - ask whether anything bad can happen
  - if so, prevent with CLI/STI



Conservative Synchronization Design (cont)

- On a multiprocessor
  - consider all possible interleavings of instructions
  - amongst all pieces of (possibly concurrent) code
  - ask whether anything bad can happen
  - if so, use a lock to force serialization
  - good luck!



Conservative Synchronization Design (cont)

- What does "bad" mean, anyway?

- A <u>conservative</u> but <u>systematic</u> definition

  if any data written by one piece of code

  are also read <u>or</u> written by another piece of code

  these two pieces <u>must be atomic</u> with respect to each other

Conservative Synchronization Design (cont)

not shared

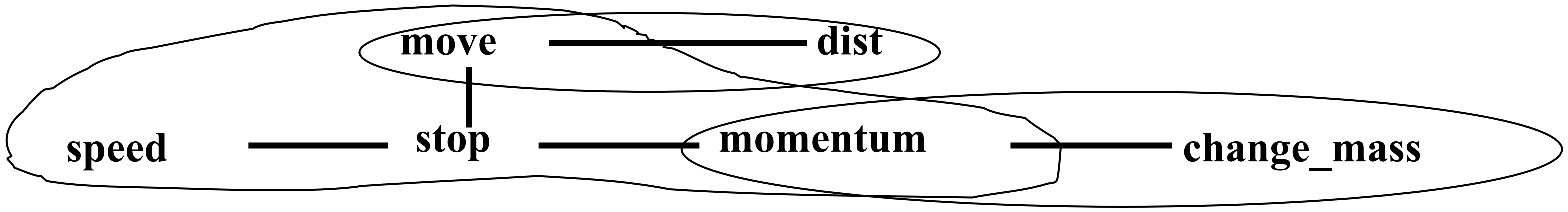
- What variables are shared?
- *step 0:* ignore the parts that don't touch shared data
- *step 1:* calculate read & write sets
- *step 2:* check for R/W, W/W relationships
  - **must be atomic!**
- *step 3:* add lock(s) to guarantee atomic execution (pick order if > 1 locks)
- *step 4:* optimize if desired

```c
typedef struct {
    double mass;
    double x, y, z;      /* position */
    double vx, vy, vz;   /* velocity */
} thing_t;

void move(thing_t* t)
{
    t->x += t->vx;
    t->y += t->vy;
    t->z += t->vz;
}

double dist(thing_t* t)
{
    return sqrt(t->x * t->x +
                t->y * t->y +
                t->z * t->z);
}

double speed(thing_t* t)
{
    return sqrt(t->vx * t->vx +
                t->vy * t->vy +
                t->vz * t->vz);
}

double momentum(thing_t* t)
{
    double tmp = t->mass;
    return tmp * speed(t);
}

void stop(thing_t* t)
{
    t->vx = t->vy = t->vz = 0;
}

void change_mass(thing_t* t, double new_mass)
{
    t->mass = new_mass;
}
```

```c
#include<stdio.h>

typedef struct person_t person_t;
struct person_t {
    char*      name;
    int        age;
    person_t*  next;
};

static person_t* group;

void birthday(person_t* p)
{
    p->age++;
}

void list_people(void)
{
    person_t* p;
    int num;

    for (p=group, num=0; NULL != p; p=p->next, num++)
    {
        if (10 > num)
            printf("%s %d\n", p->name, p->age);
        else
            printf("%s\n", p->name);
    }
}
```

Conservative Synchronization Design – Example Code Analysis

- Read/write sets
  - move
  - dist
  - speed
  - momentum
  - stop
  - change_mass



- Edges in graph imply need for atomicity

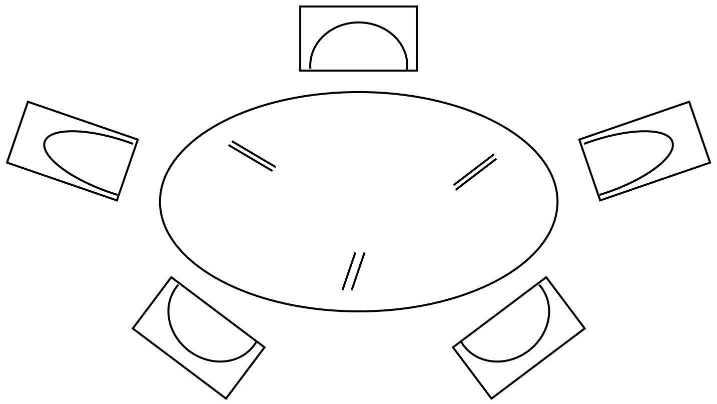
Conservative Synchronization Design – Example Code Analysis (cont)

- Each lock = circle in graph
- All edges must be contained in some circle

- One lock suffices, but prevents parallelism (performance)
- Could use three (as shown above);
  then MUST pick a lock order!

Role of Semaphores

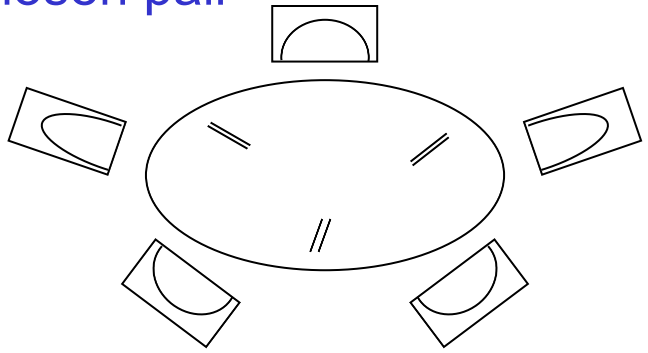
- Recall our philosophical friends
- Five philosophers
- Three pairs of chopsticks (one "lock" per pair)
- Problem: how do you get a pair?



- Option 1: walk around the table until you find a pair free
  - lots of walking
  - other people may cut in front of you

Role of Semaphores

- Option 2: pick a target pair and wait for it to be free
  - other pairs may be on the table
  - but you’re still waiting hungrily for your chosen pair



- Instead, use a semaphore!
  - an atomic counter
  - proberen (P for short, meaning test/decrement)
  - verhogen (V for short, meaning increment)
  - Dutch courtesy of E. Dijkstra

Semaphores

- When are semaphores useful?
- Fixed number of resources to be allocated dynamically
  - physical devices
  - virtual devices
  - entries in a static array
  - logical channels in a network
- Linux semaphores have a critical difference from Linux spin locks
  - can block (sleep) and let other programs run (spin locks do not block)
  - thus <u>must not</u> be used by interrupt handlers
  - used for system calls, particularly with long critical sections

Linux Semaphore API

```
void sema_init (struct semaphore* sem, int val);
```

Initialize dynamically to a specified value

```
void init_MUTEX (struct semaphore* sem);
```

Initialize dynamically to value one (mutual exclusion)

```
void down (struct semaphore* sem);
```

Wait on the semaphore (P)

```
void up (struct semaphore* sem);
```

Signal the semaphore (V)

Linux Semaphore API (cont.)

- If critical section needs both semaphores and spin locks
  - Get all semaphores first!
  - Linux expects <u>not</u> to be preempted while holding spin locks
  - Semaphore code <u>voluntarily</u> relinquishes processor

Reader/Writer Problem: The Philosophers and Their Newspaper

- Philosophers like to read newspaper
  - each philosopher reads frequently, taking short breaks between
  - multiple philosophers may read at same time
    - different sections
    - or just over another's shoulder
- Paper carrier delivers new paper
  - once per day (infrequently)
  - must change all sections at once
- Reader/writer synchronization supports this style
  - allows many (in theory infinite) readers simultaneously
  - at most one writer (and never at same time as readers)

Reader/Writer Problem: The Philosophers and Their Newspaper (cont)

- What if newspaper is always being read? starvation!
- Linux provides two types of reader/writer synchronization
  - reader/writer spin locks (rwlock_t)
    - extension of spin locks
    - use for short critical sections
    - ok to use in interrupt handlers
    - admit writer starvation (you must ensure that this not happen)
  - reader/writer semaphores (struct rw_semaphore)
    - extension of semaphores
    - use only in system calls
    - do not admit writer starvation
      (new readers wait if writer is waiting)

Selecting a Synchronization Mechanism

- General questions for synchronization of shared resources
  - What are the operations?
  - What are their frequencies?
  - example:
    - walk list and find min. age?
    - update min. age every time list changes?
  - What are the shared data?
- Goals
  - short critical sections
  - low contention for data

Selecting a Synchronization Mechanism (cont)

- Safe selection (when in doubt…)

	mutual exclusion	reader/writer