ECE391
Computer System Engineering
Lecture 7

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Fall 2018

Lecture Topics

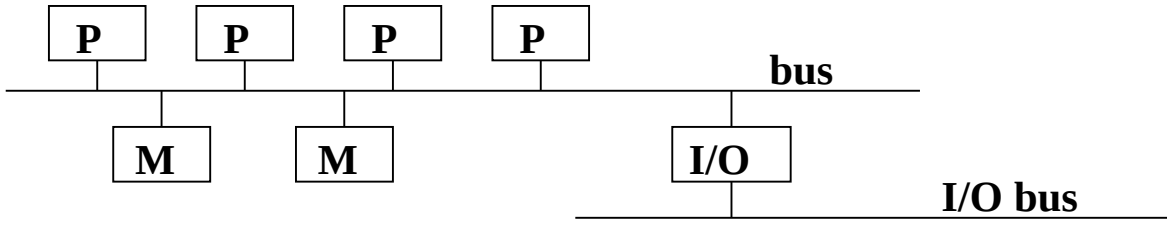
- Multiprocessors and locks
- Spin locks
- Synchronization issues

Aministrivia

- PS2 Posted (soon)

Multiprocessors and Locks

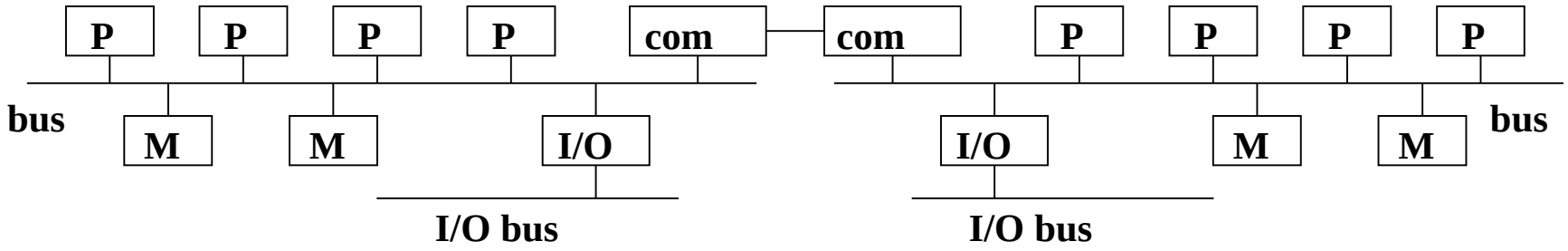
- We solved the critical section problem for uniprocessors
- What about multiprocessors?
  - *CLI … critical section …. STI*
- What is a multiprocessor?
  - usually a symmetric multiprocessor (SMP)



  - 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

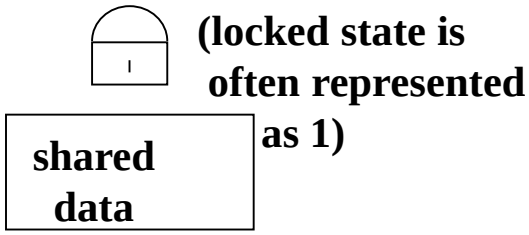


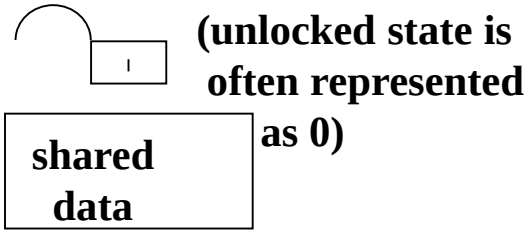
Multiprocessors and Locks (cont.)

- 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

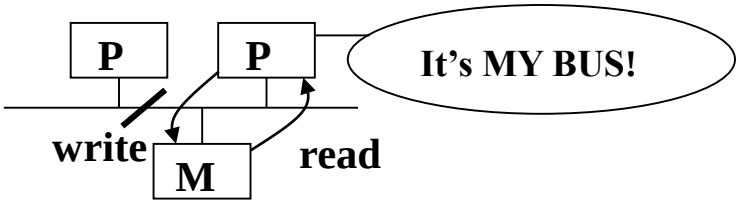




  - once it's unlocked
    - we lock it
    - access the data
    - then unlock it

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);
```

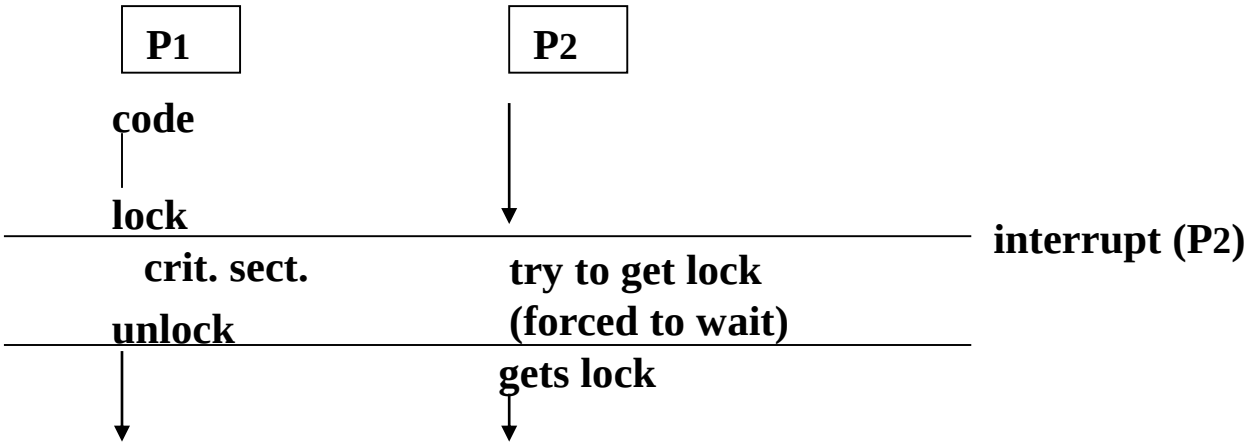
make one attempt; returns 1 on success, 0 on failure

```
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.)

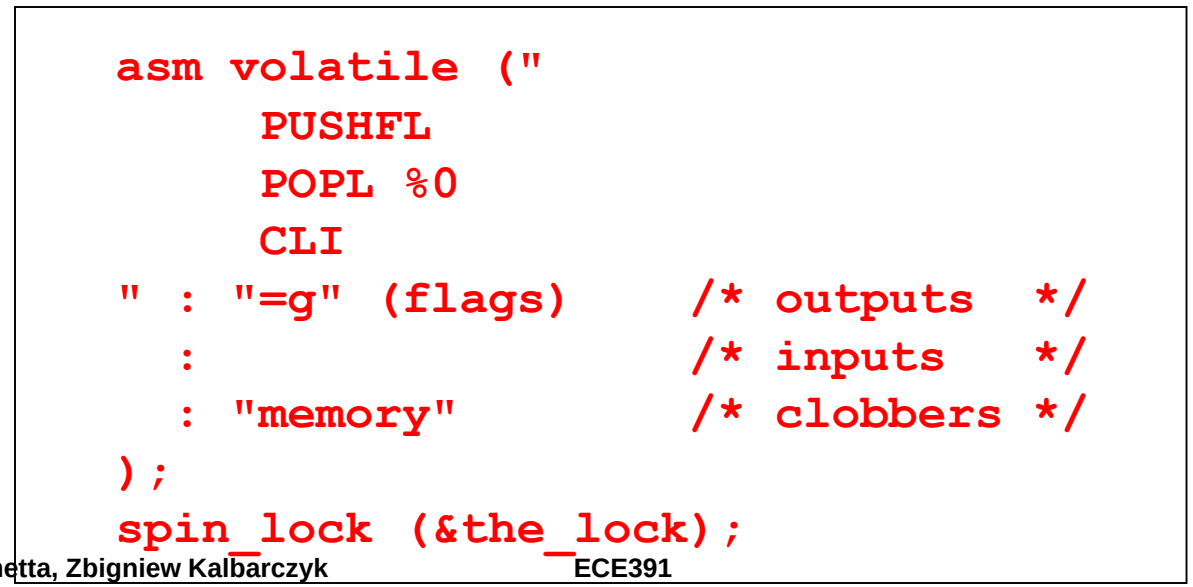
- What about ?

- 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)

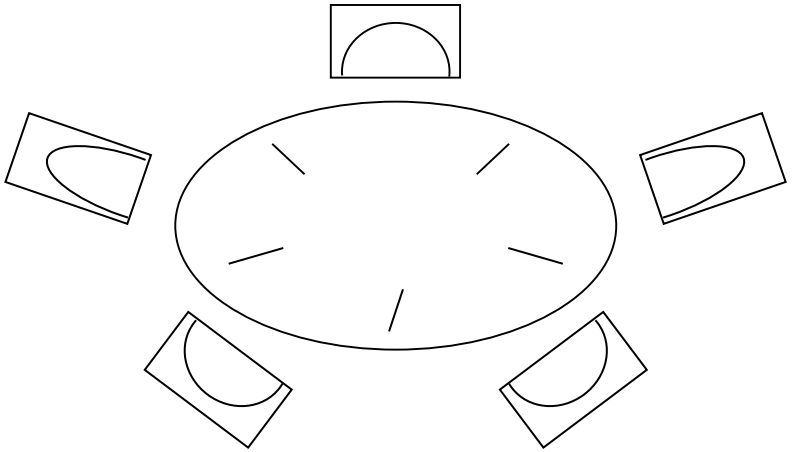
```
spin_unlock (&the_lock);
asm volatile ("
     PUSHL %0
     POPFL
" :                         /* outputs  */
  : "g" (flags)             /* inputs   */
  : "memory", "cc"          /* clobbers */
);
```

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

Another Philosophy Lesson

- Synchronization issues
  - five hungry philosophers
  - five chopsticks

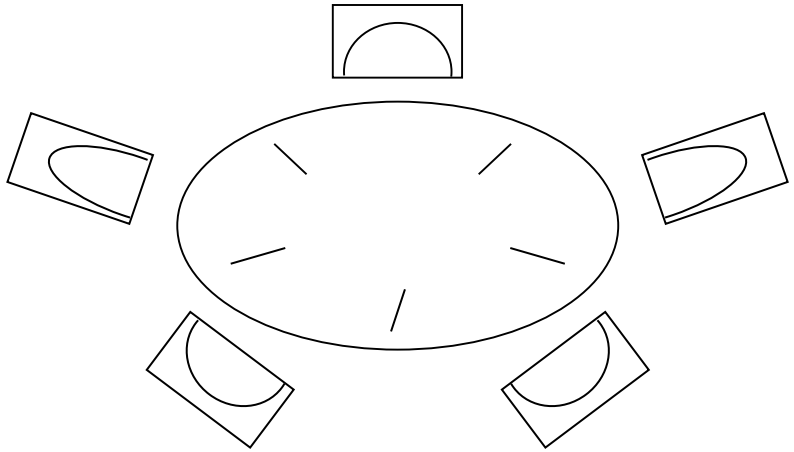


- Protocol
  - take left chopstick (or wait)
  - take right chopstick (or wait)
  - eat
  - release right chopstick
  - release left chopstick
  - digest
  - repeat

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

Example of Code Understanding

- Given code in left column
  - Which other columns are compatible (i.e., can't lead to deadlock)?

#0	#1	#2	#3
lock A	lock B		
lock B	lock A	lock A	lock B
.	.	.	.
.	.	.	.
.	.	.	.
unlock B	unlock A	unlock A	unlock B
unlock A	unlock B		

  - Columns 2 and 3 are compatible with column 0