ECE391
Computer System Engineering

Lecture 8

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Fall 2022

Lecture Topics

- Synch issues
- Conservative synchronization design
- Semaphores
- Reader/writer synchronization

Aministrivia

- **PS2** (Problem Set 2) posted
  - Due by Tuesday October 20

- **MP2** posted
  - **Checkpoint 1** due by Monday October 3
  - **Final Checkpoint** due by Monday October 10

ECE391 EXAM 1

- **EXAM 1 – September 27 (Tuesday);**
  - Time: 7:00pm to 9:00pm
- **Conflict Exam**
  - Deadline to request conflict exam: Thursday, September 22 (by email to: *kalbarcz@Illinois.edu*)
- **Exam 1 Review Session in collaboration with HKN**
  - Date (tentative): Saturday September 24;

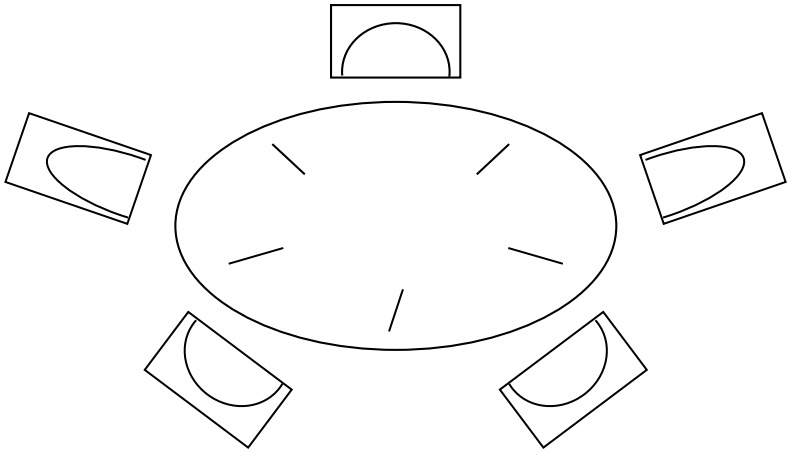
ECE391 EXAM 1

- **Topics covered by EXAM 1**
  - Material covered in lectures (Lecture1 – Lecture10)
    - x86 Assembly
    - C-Calling Convention
    - Synchronization
    - Interrupt control (using PIC)
  - Material covered in discussions
  - MP1

- **NO Lecture on Tuesday, September 27**

Another Philosophy Lesson

- Synchronization issues
  - five hungry philosophers
  - five chopsticks

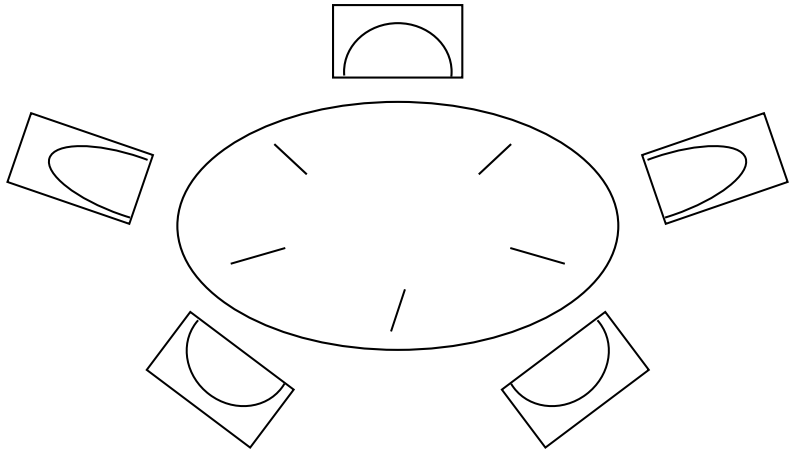


- Protocol
  - take left chopstick (or wait)
  - take right chopstick (or wait)
  - eat
  - release right chopstick
  - release left chopstick
  - digest
  - repeat

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

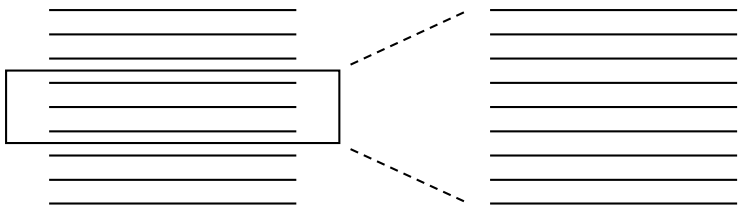
Another Philosophy Lesson (cont.)

- 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

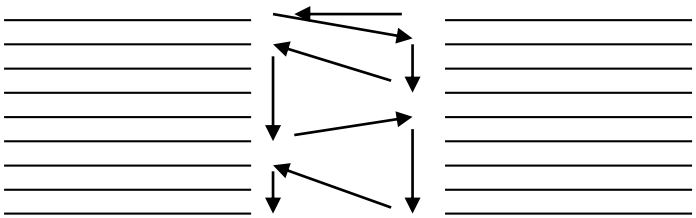
- Getting synchronization correct can be hard
  - it's the focus of several research communities

- 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 conservative but systematic definition

  if any data written by one piece of code

  are also read or written by another piece of code

  these two pieces must be atomic with respect to each other

Conservative Synchronization Design (cont)

not shared

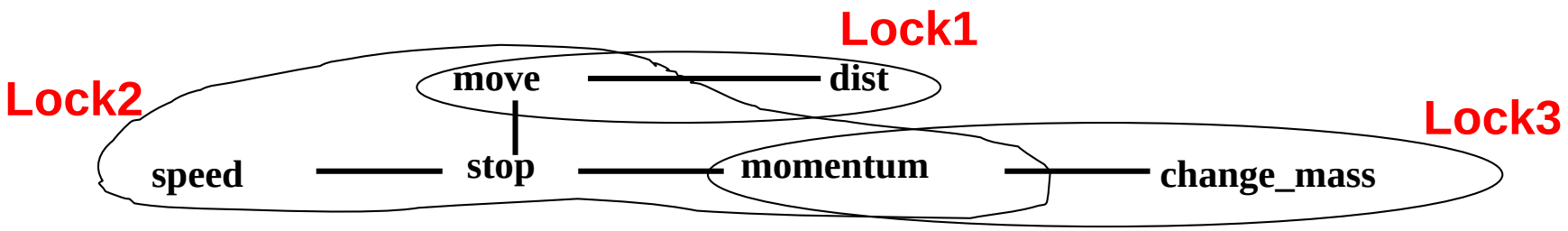
- What variables are shared?
- *step 0:* ignore the parts that don't touch shared data
- *step 1:* calculate read & write sets
- *step 2:* check for R/W, W/W relationships
  - **must be atomic!**
- *step 3:* add lock(s) to guarantee atomic execution (pick order if > 1 locks)
- *step 4:* optimize if desired

```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

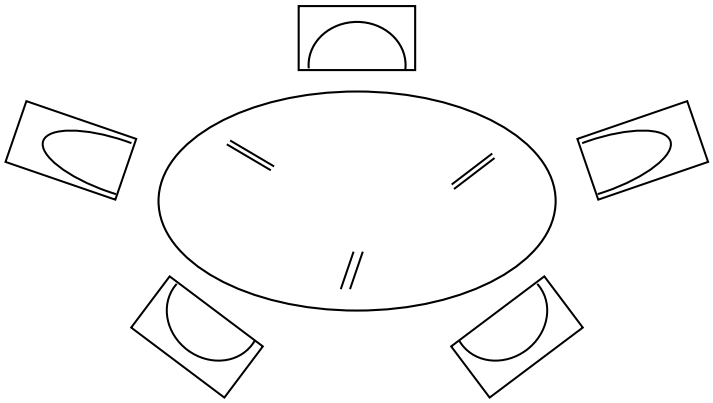
Conservative Synchronization Design – Example Code Analysis (cont)

- 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

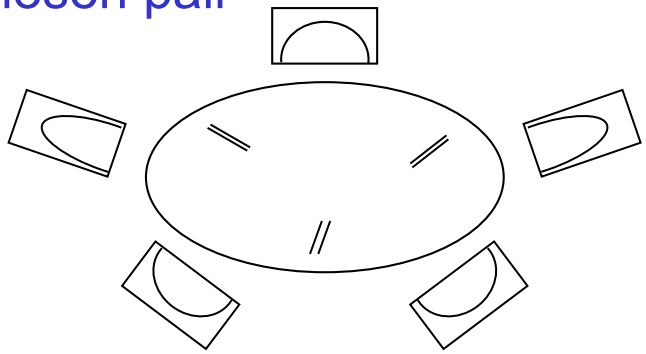
- Recall our philosophical friends
- Five philosophers
- Three pairs of chopsticks (one “lock” per pair)
- Problem: how do you get a pair?



- 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 must not 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)

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 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)
- What if newspaper is always being read? starvation!

Reader/Writer Locks in Linux

- 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)