CS4231
Parallel and Distributed Algorithms

Lecture 1

Instructor: YU Haifeng

A bit of self-introduction…

- https://www.comp.nus.edu.sg/~yuhf

References

- Some of the materials in the slides of this course are adapted from
  - Various textbooks (such as “Distributed Algorithms: An Intuitive Approach” by Wan Fokkink, 2018, 2nd edition)
  - Various recent research papers to keep the materials up-to-date
  - Some materials and homework problems from “Concurrent and Distributed Computing in Java” by Vijay Garg

Module Overview

- Module homepage ready on Canvas
  - Check module homepage often!
- Polls on https://www.polleverywhere.com/home and PollEv.com/haifengyu229
- Prerequisite:
  - CS3230 Design and Analysis of Algorithms or CS3210 Parallel Computing --- CS3230 is more important
  - Can NOT be waived – will not honor request for waiver
- What is the module about:
  - Designing parallel/distributed algorithms, and proving their correctness/properties
  - This is a theory module, involving mostly with proofs and theorems

Is This Module Hard?

- This is an elective module catering to students with strong interests and background in the subject

- While having the same theoretical nature as the compulsory module CS3230, this module will be much harder than CS3230
  - Analogy:

    English modules taken by English major students

    vs.

    English modules taken by non-English-major students

  - This module will be taught in very different ways from CS3230

Is This Module Hard?

- This is a 4000-level module:
  - Not recommended for 1st, 2nd, or 3rd year students – but if you think you can handle this challenging module, you can still take it but you are encouraged to chat with me before enrolling

- Overall, this is designed to be a challenging module
  - If you did not enjoy CS3230, you will not enjoy this module
  - If you enjoyed CS3230, you may still not enjoy this module if your formal/mathematical skill and abstract thinking ability are not strong
  - In previous offerings, it was not unusual that some students who did well in CS3230 had difficulty handling this module

Why This Module Is Important

- This module is one of the most theoretical modules in SoC
  - Theoretical = impractical = useless?
  - No. Actually it directly relates to your career…
- Reason #1: Deep understanding of distributed algorithms distinguishes you from others
  - (Unfortunately) Everyone’s career has a lot to do with competing with other people
  - Good programming skills ensure that you don’t lose to others; deep understanding helps you to win

Why This Module Is Important

- Reason #2: Understanding foundational concepts enables you to learn new material in the future
  - Giving you some meat vs. giving you a hunting gun
  - Computer science is particularly fast-changing – don’t expect what you learn today can be directly applicable 10 years later

- Reason #3: Foundational concepts are harder to grasp
  - This can be your only opportunity to learn these concepts in your whole career

An Example

- Boss asks you to solve the following simple problem
  - Two nodes A and B, each has a starting value of 0 or 1
  - Each needs to output a single value 0 or 1
  - They can communicate but messages can be lost
  - Goal: A and B should output the same value. Specifically,
    - If A and B both start with 0, they should both output 0.
    - If A and B both start with 1 and if no messages are lost, they should both output 1.
    - If A and B both start with 1 and if some message is lost, they should **either** both output 1 **or** both output 0.
    - If A and B start with different values, they should **either** both output 1 **or** both output 0.

Quick Poll https://pollev.com/haifengyu229

To be, or not to be, that is the question

CS4231

on one hand:
challenging

on the other hand:
important

The decision is up to you…

What students in previous years say

- “A very difficult module and I found it very hard to understand”
- “An interesting module that touch on parallel algorithms and distributed system, but concepts are a little too difficult sometimes”
- “The concepts are interesting, but really difficult to digest.”
- “Better to prove the algorithm in a more straight forward way. Too many lemma used, which make student hard to follow when using the lemma.”

Relation with CS3211

- CS3211
  - PARALLEL AND CONCURRENT PROGRAMMING

- CS3211 is more programming oriented
- CS4231 is more about designing algorithms and proving their correctness/properties
- Overlap is small

Teaching Format

- Wednesday 6:30pm-9:30pm (not 6:30pm-8:30pm) in every lecturing week: Lecturing + tutorial (break in the middle)
  - Face-to-face

- By NUS university guideline
  1. One-hour lesson: 45-minute teaching + concludes 15 minutes before the end of the hour
  2. Two-hour lesson: 90-minute teaching + 5-minute break in the middle + concludes 25 minutes before the end of the second hour
  3. Three-hour lesson: 135-minute teaching + 20-minute break in the middle (potentially split into two smaller breaks) + concludes 25 minutes before the end of the third hour

- You should read the materials to be covered before each lecture – Otherwise you won't follow

Course Format

- Office hours:
  - Monday 4:00pm to 5:00pm every lecturing week in my office COM2-04-25
  - You are also welcome to approach me at other times or email me
- Do not email my via Canvas – please directly email haifeng@comp.nus.edu.sg to get faster response

- Weekly homework – Use of AI not allowed
- No systematic programming homework/exercise
  - To avoid overlapping with CS3211
  - To avoid excessive workload in this module
  - But you are still encouraged to implement algorithms learned

Teaching Materials

- All materials taught in class and all materials in lecture notes are compulsory materials
  - These are kept up-to-date
- Why no textbooks?
  - Existing textbooks on the subject can be slightly old (as compared to the lecture notes)
  - Theoretical and foundational materials do not actually change so often, so people do not write a lot of new textbooks on such materials…
  - Some similar courses in other universities do not use textbooks either.

Grading Policy

- 30% mid-term exam and 70% final exam
  - Both exams cover whatever have been taught by the time of the exam
  - Mid-term: Closed book, face-to-face
  - Final: Open book, face-to-face
- Homeworks do not directly contribute to final score
  - BUT some exam questions will be variants of homework questions
- Cheating ABSOLUTELY not tolerated and will be reported

Mid-term Exam Date

- 19:00pm-20:00pm on Wed 04 Mar 2026 (students must arrive by 18:30pm)
  - Venue will be announce later
  - If venue unavailable, then it will be 19:00pm-20:00pm on Wed 11 Mar 2026

- Same policy as the final exam:
  - Not showing up for the mid-term exam = zero mark
  - Showing up later for the mid-term exam = less time to work on the exam (no extra time will be given)

You MUST physically attend the (face-to-face) mid-term exam

- **Being able to physically attend the (face-to-face) mid-term exam is <u>prerequisite</u> for taking this module.**

- If you feel you have trouble showing up for the mid-term exam, let me know now…

Class Participation and Penalty for Non-participation

- The School and also myself encourage class participation
  - But hard to incorporate into assessment (fairness issues)

- My approach to encourage participation:
  - If you miss any important discussions/announcements I made during lecture – you pay the price yourself
  - There will be important things that are only discussed during lecture, and not via email or Canvas or other venues

- Examples of questions that I will not answer:
  - I was at a party last night so I didn't attend the lecture, did you say anything important in the lecture?

~~Rescheduling of Class~~

- ~~18 Feb 2026 is a public holiday, so we need to reschedule the class on that day…~~
  - ~~To 2:30pm-5:30pm on Saturday 24 Jan 2026~~

Material Covered Today and Next Week

- Today: “Mutual Exclusion”
  - No tutorial today

- Next week:
  - “Synchronization Primitives”
  - Read slides before you come to class next week

Break

Mutual Exclusion Problem

- Context: Shared memory systems
  - Multi-processor computers
  - Multi-threaded programs
- Shared variable x
  - Initial value 0
- Program $x = x+1$

process 0	*process 1*
read x into a register (value read: 0)	
increment the register (1)	
write value in register back to x (1)	
	read x into a register (value read: 1)
	increment the register (2)
	write value in register back to x (2)

Mutual Exclusion Problem

- Context: Shared memory systems
  - Multi-processor computers
  - Multi-threaded programs
- Shared variable *x*
  - Initial value 0
- Program *x* = *x*+1
- Quick Poll https://pollev.com/haifengyu229

process 0	*process 1*
read *x* into a register (value read: 0)	
increment the register (1)	
	read *x* into a register (value read: 0)
	increment the register (1)
write value in register back to *x* (1)	
	write value in register back to *x* (1)

Critical Section

Critical Section

(also called Critical Region)

RequestCS(int processId)

Read *x* into a register

Increment the register

Write the register value back to *x*

ReleaseCS(int processId)

Roadmap

- Software solutions
  - Unsuccessful attempts
  - Peterson’s algorithm
  - Bakery algorithm

- Hardware solutions
  - Disabling interrupts to prevent context switch
  - Special machine-level instructions

Attempt 1

```
Shared boolean variable openDoor;
    //whether door is open
RequestCS(int processId) {
    while (openDoor == false) {};
    openDoor = false;  // close door behind me
}

ReleaseCS(int processId) {
    openDoor = true;  // open door to let other people in
}
```

Both process may see openDoor as true

Violate mutual exclusion: Two processes in critical region

Attempt 2

Shared boolean variable wantCS[0], wantCS[1] initialized to false

wantCS[0] = wantCS[1] = true

Process 0	*Process 1*
RequestCS(0) {	RequestCS(1) {
wantCS[0] = true;	wantCS[1] = true;
while (wantCS[1] == true) {};	while (wantCS[0] == true) {};
}	}
ReleaseCS(0) {	ReleaseCS(1) {
wantCS[0] = false;	wantCS[1] = false;
}	}

No progress: No one can enter critical region

Attempt 3

Shared int turn initialized to 0

Process 0

```
RequestCS(0) {
    while (turn == 1) {};
}

ReleaseCS(0) {
    turn = 1;
}
```

Process 1

```
RequestCS(1) {
    while (turn == 0) {};
}

ReleaseCS(1) {
    turn = 0;
}
```

Starvation: Process 0 may never enter critical region again

(There are other kinds of starvation…)

Properties Needed

1. Mutual exclusion: No more than one process in the critical section
2. Progress: If one or more process wants to enter and if no one is in the critical section, then one of them can eventually enter the critical section
   - Intuitively, we want to ensure that "resource" is fully utilized
3. No starvation: If a process wants to enter, it eventually can always enter
   - No starvation implies Progress

- Always need to consider the worst-case schedule.
- If a process is in critical section, we always assume that it will eventually exit the critical section.

Peterson's Algorithm

Shared bool wantCS[0] = false, bool wantCS[1] = false, int turn = 0;

Process 0

```
RequestCS(0) {
    wantCS[0] = true;
    turn = 1;
    while (wantCS[1] == true &&
              turn == 1) {};
}

ReleaseCS(0) {
    wantCS[0] = false;
}
```

Process 1

```
RequestCS(1) {
    wantCS[1] = true;
    turn = 0;
    while (wantCS[0] == true &&
              turn == 0) {};
}

ReleaseCS(1) {
    wantCS[1] = false;
}
```

Correctness Proof for Peterson's Alg.

Process 0

```
RequestCS(0) {
    wantCS[0] = true;
    turn = 1;
    while (wantCS[1] == true &&
              turn == 1) {};
}
```

Process 1

```
RequestCS(1) {
    wantCS[1] = true;
    turn = 0;
    while (wantCS[0] == true &&
              turn == 0) {};
}
```

Mutual exclusion: Proof by contradiction.

Case 1: turn == 0 when P0 and P1 are both in critical section.

Then P0 executed "turn = 1" before P1 executed "turn = 0".

Hence wantCS[0] == false as seen by P1.

But wantCS[0] set to true by Process 0.

Case 2: turn == 1. Symmetric – complete yourself…

Correctness Proof for Peterson's Alg.

Process 0

```
RequestCS(0) {
    wantCS[0] = true;
    turn = 1;
    while (wantCS[1] == true &&
                turn == 1) {};
}
```

Process 1

```
RequestCS(1) {
    wantCS[1] = true;
    turn = 0;
    while (wantCS[0] == true &&
                turn == 0) {};
}
```

Progress: Proof by contradiction and consider the value of turn when both P0 and P1 are waiting.

Case 1: turn == 0. Then P0 can enter.

Case 2: turn == 1. Symmetric – complete yourself…

Correctness Proof for Peterson's Alg.

Process 0

```
RequestCS(0) {
    wantCS[0] = true;
    turn = 1;
    while (wantCS[1] == true &&
      turn == 1) {};
}
```

Process 1

```
RequestCS(1) {
    wantCS[1] = true;
    turn = 0;
    while (wantCS[0] == true &&
      turn == 0) {};
}
ReleaseCS(1) {
    wantCS[1] = false;
}
```

No starvation: Proof by contradiction.

Case 1: If P0 waiting, then wantCS[1] = true and turn = 1.

P1 in critical region -- will exit and set wantCS[1] to false.

What if P1 wants to enter again immediately?

Case 2: P1 is waiting. Symmetric – complete yourself…

Lamport's Bakery Algorithm

- For n processes
  - Get a number first
  - Get served when all people with lower number have been served

- Two shared arrays of n elements
  - boolean choosing[i] = false; // process i is trying to get a number
  - int number[i] = 0; // the number got by process i;
    // "0" means process i not interested in being served

```
ReleaseCS(int myid) {
    number[myid] = 0;
}

// a utility function
boolean Smaller(int number1, int id1, int number2, int id2) {
    if (number1 < number 2) return true;
    if (number1 == number2) {
            if (id1 < id2) return true; else return false;
    }
    if (number 1 > number2) return false;
}
```

get a number

```
RequestCS(int myid) {
  choosing[myid] = true;
  for (int j = 0; j < n; j++)
    tmp = number[ j ];
    if (tmp > number[myid]) number[myid] = tmp;
  number[myid]++;
  choosing[myid] = false;

  for (int  j = 0; j < n; j++) {
    while (choosing[ j ] == true);
    while (number[ j ] != 0 &&
              Smaller(number[ j ],  j, number[myid], myid));
  }
}
```

wait for people ahead of me

```
choosing[myid] = true;
for (int j = 0; j < n; j ++)
    tmp = number[ j ];
    if (tmp > number[myid])
        number[myid] = tmp;
number[myid]++;
choosing[myid] = false;

for (int  j = 0; j < n; j ++) {
    while (choosing[ j ] == true);
    while (number[ j ] != 0 &&
        Smaller(number[ j ],  j,
        number[myid], myid));
}
```

- Progress: Proof by contradiction. Consider any set of processes that wants to enter the CS but no one can make progress. Each process is guaranteed to get a queue #. Let process *i* be the one with the smallest queue number. Consider where process *i* can be blocked:
  - Case 1:

    Process *j* will eventually set choosing[*j*] to false

    Process *j* will then block (otherwise there is progress already!)
  - Case 2: Impossible since process *i* has the smallest queue number
- No starvation: Can be similarly shown…work it out yourself...

Mutual exclusion: Suppose i and k both in critical section.

At T1, process i is here

```
choosing[myid] = true;
for (int j = 0; j < n; j ++)
    tmp = number[ j ];
    if (tmp > number[myid])
        number[myid] = tmp;
number[myid]++;
choosing[myid] = false;

for (int  j = 0; j < n; j ++) {
    while (choosing[ j ] == true);
    while (number[ j ] != 0 &&
        Smaller(number[ j ],  j,
        number[myid], myid));
}
```

At T1, process k is here

- W.l.o.g, assume Smaller(number[i], i, number[k], k) after they are in the critical sec
- Process k must see number[i] == 0 at that time T1: We want to know where process i is at time T1.
  - Case 1: Process i has not executed "tmp = number[k]".Then eventually number[i] > number[k]. Impossible.
  - Case 2: Has executed "tmp = number[k]"
    - Subcase 2.1: Process i has executed "number[myid]++;" -- impossible since number[i] == 0
    - Subcase 2.2: Has not executed "number[myid]++;" -- This is the only possible case.

At T1, process *i* is here

```
choosing[myid] = true;
for (int j = 0; j < n; j ++)
  tmp = number[ j ];
  if (tmp > number[myid])
    number[myid] = tmp;
number[myid]++;
choosing[myid] = false;

for (int  j = 0; j < n; j ++) {
  while (choosing[ j ] == true);
  while (number[ j ] != 0 &&
    Smaller(number[ j ],  j,
     number[myid], myid));
}
```

At T2, process *k* is here

At T1, process *k* is here

- Now continue and consider the time T2 when process *k* invoked "while (choosing[*i*] = true);" and passed that statement.
- We want to see where process *i* is at T2. Since choosing[*i*] = false, process *i* must either have finished choosing its queue number or have not started choosing:
  - Case 1: process *i* has finished choosing and has executed choosing[*i*] = false; Impossible since T2 < T1.
  - Case 2: process *i* has not started choosing and has not executed choosing[*i*] = true. But then number[*i*] will be larger than number[*k*]. Contradiction.

Roadmap

- Software solutions
  - Unsuccessful attempts
  - Peterson's algorithm
  - Bakery algorithm

- Hardware solutions
  - Disabling interrupts to prevent context switch
  - Special machine-level instructions

Disable Interrupts

Do not allow context switch here

process 1	process 2
read *x* into a register (value read: 0)	
increment the register (1)	
	read *x* into a register (value read: 0)
	increment the register (1)
write value in register back to *x* (1)	
	write value in register back to *x* (1)

Special Machine-level Instructions

```
boolean TestAndSet(Boolean openDoor, boolean newValue) {
    boolean tmp = openDoor.getValue();
    openDoor.setValue(newValue);
    return tmp;
}
```

Executed atomically

```
shared Boolean variable openDoor initialized to true;
RequestCS(process_id) {
    while (TestAndSet(openDoor, false) == false) {};
}
ReleaseCS(process_id) { openDoor.setValue(true); }
```

Summary

- Module overview & policy

- Mutual exclusion problem in shared-memory systems
- Software solutions
  - Unsuccessful attempts
  - Peterson’s algorithm
  - Bakery algorithm
- Hardware solutions
  - Disabling interrupts to prevent context switch
  - Special machine-level instructions

Homework Assignment (on this and next few slides)

- Devise a mutual exclusion algorithm for *n* processes by using Peterson's 2-mutual-exclusion algorithm as a black-box
- Construct two interleaving executions to show that each of the following modifications to Peterson's algorithm will make it incorrect:
  - A process sets the turn variable to itself instead of to the other process.
  - A process sets the turn variable before setting the wantCS variable.
- Construct an interleaving executions to show that the bakery algorithm becomes incorrect, if we do not use the choosing variable

- Does the following algorithm for 2 processes guarantee: Mutual exclusion? Progress? No starvation? For each of these 3 properties, either give a proof or construct a problematic scenario as counter-example.

Shared boolean wantCS[0] = false, boolean wantCS[1] = false, int turn = 1;

Process 0

```
RequestCS(0) {
    wantCS[0] = true;
    while (wantCS[1]) {
      if (turn == 1) {
          wantCS[0] = false;
          while (turn == 1) { ;}
          wantCS[0] = true;
      }
    }
}
ReleaseCS(0) { turn = 1; wantCS[0] = false; }
```

Process 1

```
RequestCS(1) {
    wantCS[1] = true;
    while (wantCS[0]) {
      if (turn == 0) {
          wantCS[1] = false;
          while (turn == 0) { ;}
          wantCS[1] = true;
      }
    }
}
ReleaseCS(1) { turn = 0; wantCS[1] = false; }
```

- Bring you completed homework to class next week