Midterm Exam

- Date: **10:45 am – 11:45 am, Oct 31, 2023, Tuesday**
- Classroom: CYCP1
- Format:
  - Open-book, but no computers or any electronics
  - About 5 questions (some are short questions)
  - Answer sheets will be provided
  - Calculators are allowed, but optional
- Content:
  - Everything so far cut at "Deadlock" (excluded)
- Absence Policy:
  - 10% marks automatically merged to Final Exam
- Solutions to PS#2 will be released in Monday midnight.

Deadlock

2023-2024 Fall COMP3230A

Contents

- What is deadlock?
- Four necessary conditions of deadlock
- Deadlock prevention
- Deadlock avoidance
- Detection and recovery

Related Learning Outcome

- ILO 2c - explain the underlying causes of deadlock issues and describe the principles and techniques used by OS to support concurrency control

Readings & References

- Required Reading
  - Chapter 32 – **Common Concurrency Problems**
    - http://pages.cs.wisc.edu/~remzi/OSTEP/threads-bugs.pdf

Producer/Consumer Problem: Problem?

```
sem_t empty, full, mutex;

sem_init(&empty, 0, MAX); // MAX buffers are empty initially
sem_init(&full, 0, 0); // 0 buffers are full
sem_init(&mutex, 0, 1); // mutex = 1 because it is a lock (binary semaphore)
```

```
Producer()
{
    while (1)
    {
        P(&mutex); /* lock */
        P(&empty); /* block if buffer unavail */

        // critical section
        <<< Put item into shared buffer >>>

        V(&full);
        V(&mutex);
    }
}
```

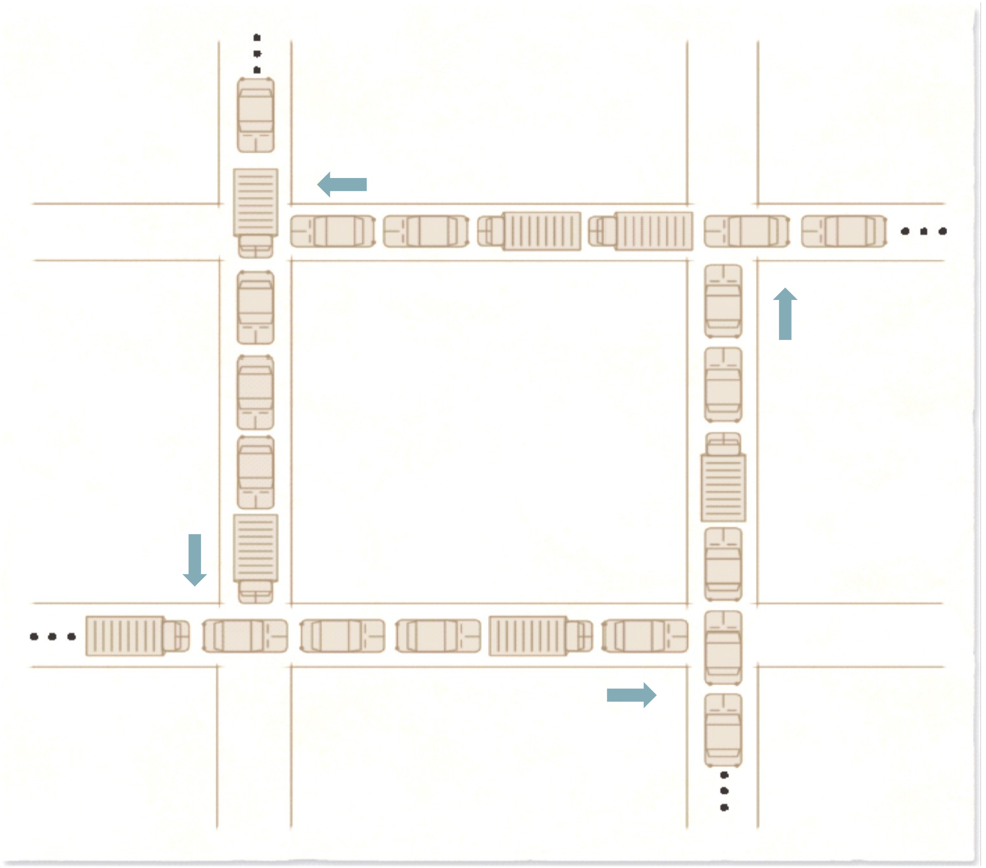
```
Consumer()
{
    while (1)
    {
        P(&mutex);
        P(&full);

        // critical section
        <<< Remove item from shared buffer >>>

        V(&empty);
        V(&mutex);
    }
}
```

Deadlock Problem in Real Life

It's a system-wide tangle of resource requests, but the system is in a state that all requests cannot be fulfilled, and **the whole system** comes in a **standstill**.



The Deadlock Problem

- A set of threads each holding some system resources and block waiting to acquire another system resource held by another thread in the set.
- Examples

T0	*T1*
wait (A);	*wait(B)*
wait (B);	*wait(A)*

T0	*T1*
lock (&lock1);	*lock (&lock1);*
lock(&lock2);	*lock(&lock2);*
while (wait) { wait(&cv, &lock2);}	*signal(&cv, &lock2);*
unlock(&lock2);	*unlock(&lock2);*
unlock(&lock1);	*unlock(&lock1);*

T0	*T1*
buffer1.put();	*buffer2.put();*
buffer1.put();	*buffer2.put();*
buffer2.get();	*buffer1.get();*
buffer2.get();	*buffer1.get();*

Why do Deadlocks occur?

- In previous examples, if we detect the situations, probably deadlock would not happen
- Unfortunately, sometimes we cannot see how threads use their locks
  - in large code bases with complex dependencies
  - in programs with **external library functions**, as with encapsulation that hides the details of implementation

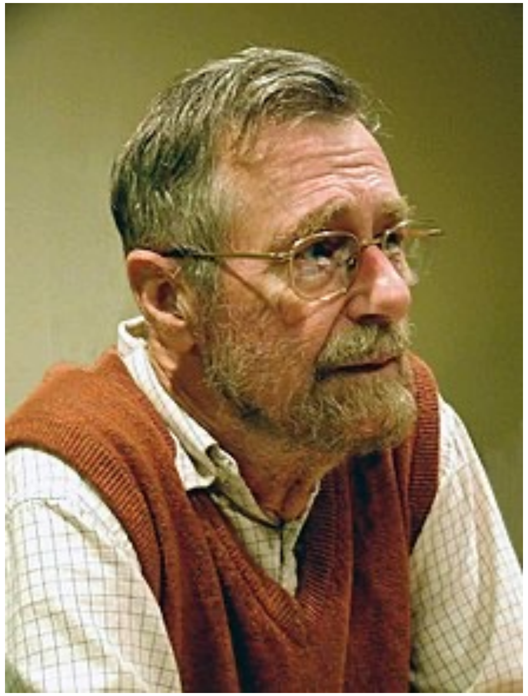
```
Vector vec1, vec2;

Thread1                  Thread2
   :                        :
   :                        :
vec1.AddAll(vec2);       vec2.AddAll(vec1);
   :                        :
                            :
```

To avoid race condition, AddAll() probably needs to lock both Vectors before performing the computation.

Classical Synchronization Problem

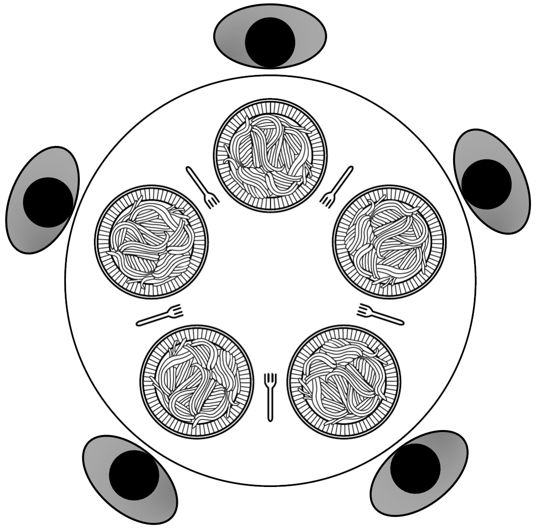
- Dining Philosophers
  - Five philosophers sit around a circular table. Each leads a simple life alternating between thinking and eating spaghetti. In front of each philosopher is a dish of spaghetti that is constantly replenished by a dedicated wait staff. There are exactly five forks on the table, one between each adjacent pair of philosophers. Eating spaghetti (in the most proper manner) requires that a philosopher uses both adjacent forks (simultaneously). Develop a concurrent program free of deadlock and indefinite postponement that models the activities of the philosophers.



Dining Philosophers

"The problem is famous because it is fun and somewhat intellectually interesting; however, its practical utility is low."

The key challenge is to show that your solution is without deadlock, no philosopher is being starved, and concurrency is high



```
philosopher

while (true) {
        think();
        eat();
}


eat() {
        pickupLeftFork();
        pickupRightFork();
        eatingSpaghetti();
        putdownRightFork();
        putdownLeftFork();
}
```

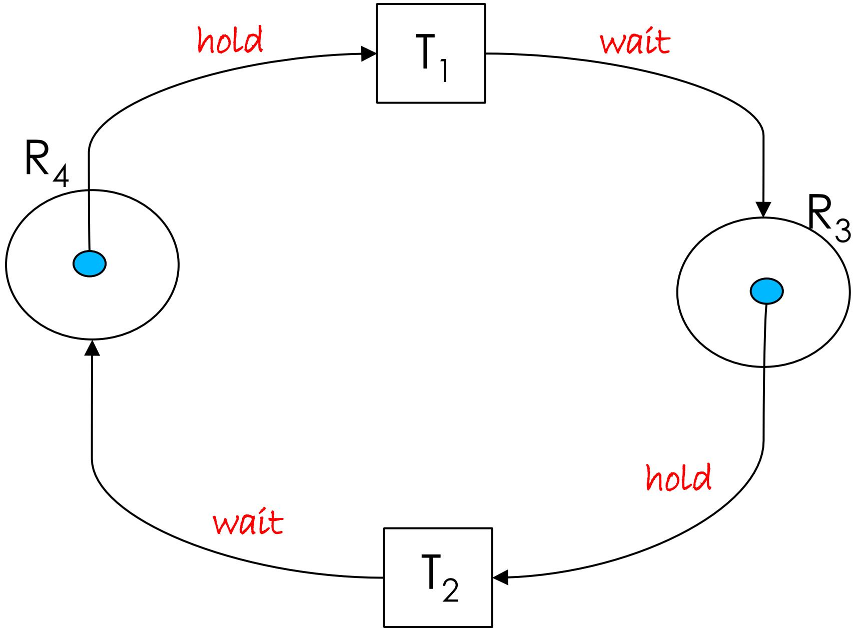
Is this solution working?

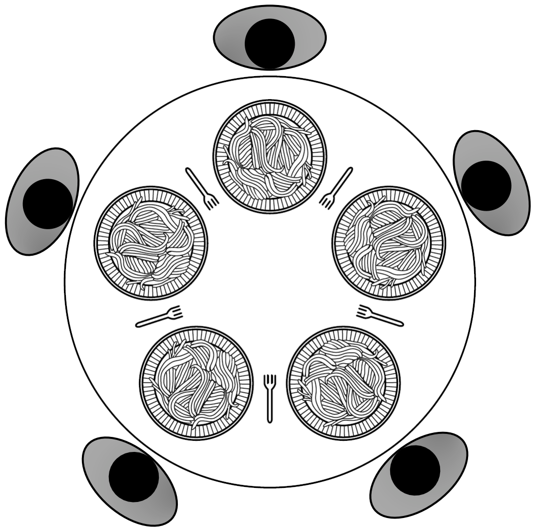
Necessary Conditions for Deadlock

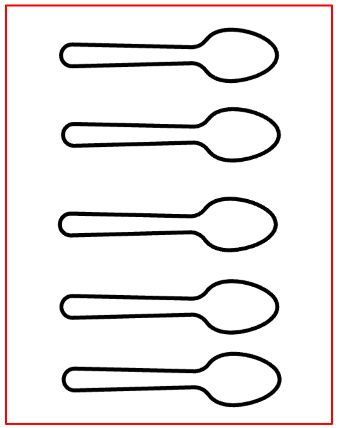
- Four conditions must be held for a deadlock to occur; if any of these are not met, deadlock CANNOT occur
  - ***Mutual exclusion condition***
    - Allow only one thread to have exclusive access to a resource
  - Wait-for condition (***hold-and-wait condition***)
    - A thread may hold some resources while awaiting assignment of additional resources
  - ***No-preemption condition***
    - No resource can be forcibly removed from a thread that holding it
  - ***Circular-wait condition***
    - Two or more threads are locked in a "circular chain" in which each thread is waiting for one or more resources that the next thread in the chain is holding
- The four conditions are **necessary but not sufficient**

Circular-Wait Condition

- Thread T1 has been allocated resource R4 that is being requested by thread T2 that has been allocated resource R3 that is being requested by T1







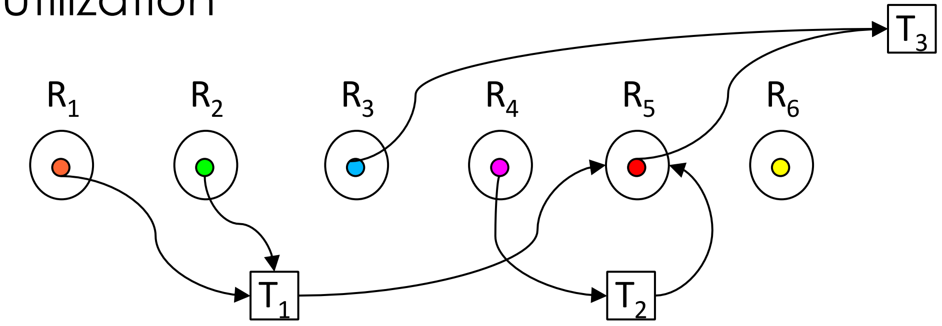
Deadlock vs. Starvation

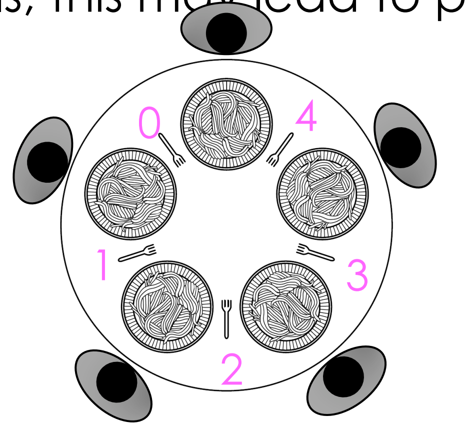
- **Starvation**: A thread fails to make progress for an indefinite period
- **Deadlock**: a group of threads forms a cycle where none make progress because each is waiting for some others to act
- **Deadlock → Starvation; Starvation !→ Deadlock**
  - E.g., Readers-Writers problem: Starving/waiting writers are waiting for active readers to finish
  - Any active thread waiting on one or more waiting threads?
- A thread subject to starvation/deadlock does not always have starvation/deadlock

Deadlock Prevention

- By using restrictive policy in allocation of resources to remove any one of the four necessary conditions, deadlock cannot happen

- Prevent Circular-wait condition
  - **Imposes a total ordering** of all resource types, and requires that each threads **requests resources in** an increasing **order** of enumeration
  - Disadvantage:
    - Not all programs using resources in that order, but you are required to hold resources of smaller labels before granting resources of larger labels; this may lead to poor resource utilization





Deadlock Prevention

- Prevent Hold-and-wait condition
  - At start, thread gets all needed resources all at once or nothing
  - So the hold-and-wait condition is never satisfied
  - Disadvantages:
    - Low resource utilization
    - Starvation possible
      - a process requests many resources may have to wait for a longer time as this strategy favors waiting processes with small resource needs

- The Mutual Exclusion condition
  - Sharable resources, if allow non-mutually exclusive access, do not result in deadlock
    - Unfortunately, most sharable resources don't support or work properly under non-mutual exclusive access

Deadlock Prevention

- Denying No-preemption condition
  - If a process that is holding some resources requests another resource but not immediately available, must **release all holding resources**
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
  - Disadvantages:
    - This can lead to substantial overhead
      - when a process releases resources, it may lose all of its work to that point; or it may have to undone all previous work before going to restart
    - possibility of starvation

Deadlock Avoidance

- Does not preventively remove one of the conditions for deadlock; instead, the system tries to avoid deadlock if it knows ahead of time all the resources requests associated with each of the threads

- Avoidance **requires some global knowledge** of which locks/resources various threads might grab during their execution
- Subsequently schedules the threads in a way as to guarantee no deadlock can occur

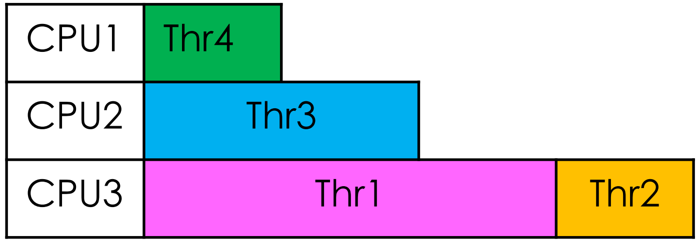
- Two approaches
  - Avoidance by scheduling
  - Banker's algorithm

Avoidance by Scheduling

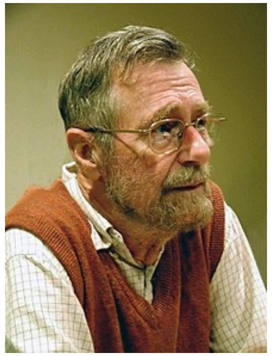
- Given the resources needs of different concurrent threads, the system looks at their dependency and use it as a guideline in scheduling the threads to avoid deadlock

- Example

	Thr1	Thr2	Thr3	Thr4
R1	need	need	no	no
R2	need	need	need	no



  - If the system allows Thr1 & Thr2 to execute concurrently by different CPUs, there is a possibility of having deadlock
  - This approach is a bit conservative and may result in under utilization of the resources



Banker's Algorithm

- Threads are allowed to hold locks/resources while requesting additional locks/resources

- System only grants the allocation of additional resources to a thread when the allocation **will not** result in an **unsafe state**
  - i.e., the system estimates that deadlock would not happen with the remaining resources
  - by means of checking whether with the remaining resources, there still exists a feasible allocation solution to satisfy all future demands from all threads that could lead to successful termination of all threads

- It has a number of weaknesses, such as
  - requiring to know ahead of time the resource needs as well as only works with a fixed number of threads and resources
    - e.g., if a device breaks and not available, the algorithm won't work as this may cause the state to turn to unsafe

Banker's Algorithm

Declared maximum usage

Current usage

Future requests

	max()	loan()	claim()
T1	4	1	3
T2	6	4	2
T3	8	5	3
total=12		available=2	

Total unit of resources

Remain available

In this example, we only have one type of resource

- Safe, unsafe, deadlocked states

If T1 asks for 1 unit, should the system grant this request?

If T2 asks for 1 unit, should the system grant this request?

Detection & Recovery

- Allow deadlocks to occasionally occur
    - "If a bad thing happens rarely, certainly one should not spend a great deal of effort to prevent it, particularly if the cost of the bad thing occurring is small"
- System takes action periodically to check whether deadlock has happened
    - Identifies processes and resources involved in the deadlock
    - Usually focus on determining if a circular wait exists
        - One technique for detecting deadlocks involves building a resource-allocation graph and looking for cycles (necessary but not sufficient if some resources have multiple instances)
- Recovery
    - Solution 0: Process without the resource
    - Solution 1: Abort all deadlocked processes
    - Solution 2: Abort one process at a time until the deadlock cycle is eliminated.
        - Successively preempt resources until deadlock no longer exists
        - Transactions: rollback and retry
    - No matter what, some processes become victims
        - Removal generally requires that the process be restarted from beginning or from a previous checkpoint

Summary

- Deadlock is a serious issue commonly found in concurrent programs, such as OS, multithreaded programs, and highly parallel programs.
- When we identify a deadlock scenario, we always find the four necessary conditions appeared in that scenario
- Solutions to deadlock
  - Prevention – use restrictive rules or guidelines to deny one of the necessary conditions
    - it is the responsibility of the programmers to apply the rules / guidelines in their programs
  - Avoidance – require to have the global knowledge of locks / resources usage amongst all threads, and use these as the hint to allocate locks / resources to threads
    - Programmers are free to arrange their logic in resource acquisition; it is the responsibility of the systems to apply the strategy in avoiding deadlock
  - Detection and Recovery – a pragmatic solution with no much overhead most of the time, but need more effort in recovering from deadlock

Non-Deadlock Concurrency Issues

- Atomicity violation
  - “The desired serializability among multiple memory accesses is violated.”
- Order violation
  - “The desired order between two (groups of) memory accesses is flipped.”

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: **Bugs In Modern Applications**

[L+08] “Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics” by Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou. ASPLOS ’08, March 2008, Seattle, Washington.

Operating Systems

- **Virtualization**
  - CPU Virtualization
    - Process Abstract
      - Address space
      - Process states
      - Process control block
      - Process operations API
      - Signals
    - Limited Direct Execution
      - System calls
      - Context switch
      - Interrupts
    - Scheduling
      - Scheduling metrics
      - FIFO, SJF, HRRN, STCF, RR, MLFQ
      - Multi-core scheduling, Linux CFS
  - Memory Virtualization
    - Address space
    - Address translation: dynamic relocation
    - Segmentation
    - Paging
    - TLB
    - Multi-level paging
    - Inverted page table
    - Swap space
    - Page replacement policy: FIFO, LFR, LRU, Clock
    - Thrashing

- **Concurrency**
  - Thread
    - POSIX threads (pthreads)
    - Race conditions, critical sections, mutual exclusion, atomic operations, synchronization
  - Locks
    - Atomic instructions: test-and-set, compare-and-swap
    - Mutex locks
  - Condition Variables
    - Pthread CVs
    - Producer-Consumer problem
  - Semaphores
    - Binary Semaphores
    - Counting Semaphores
    - Ordering
    - Readers-Writers problem
  - Deadlock
    - Dining philosophers' problem
    - Four necessary conditions
    - Deadlock prevention, avoidance, detection&recovery

- **Persistence**
  - I/O devices (HDD, SSD)
  - Files and Directories
    - Inode
    - File descriptor
    - Hard/Symbolic links
  - File System Implementation
    - On-disk data structure
      - Superblock, Bitmap, Inodes, Data blocks
    - Free space management
      - Bitmap, linked-list, block-list
    - Caching and buffering
    - Access control and protection
    - Journaling file system
      - Data journaling
      - Metadata journaling
- ***Advanced Topics***