CSE2431 – Lecture Topic 7: Concurrency (part 4): Concurrency Bugs and Deadlock

THE OHIO STATE UNIVERSITY
COLLEGE OF ENGINEERING

Concurrency (Part 4)

Instructor: Luan Duong, Ph.D.

CSE 2431: Introduction to Operating Systems

Reading: **Chap. 32 [OSTEP]**

Outline: Concurrency (part 4)

- Critical Regions
- Synchronization via busy waiting (a.k.a. Locks)
- Improve busy waiting (reduce spinning, sleep and wakeup)
- Mutex Locks and condition Variables
- Semaphores – Monitors – Barriers
- Classic Synchronization Problems
- Concurrency Bugs
- Deadlocks

Revise basic rules

- If multiple threads share data, you probably need synchronization
  - Things get trickier if you're using libraries built by others.
  - Are `malloc/printf/open/read/write` thread-safe?
- Lock + condition variable can solve most problems.
- Correctness first, then performance
  - Think carefully before you write code
  - Fine-grained locking is often used to achieve more concurrency, but it easily introduces bugs.

Common bugs of lock

- Forget (or sometimes too lazy) to use a lock
  - Hmmm. “a=1” is probably atomic, so no need to use a lock? That’s wrong…..

- Forget to unlock

- Deadlock (we will talk in more details in this lecture)

Common bugs of condition variable

- We have seen plenty in previous lectures

- Remember the rules:
  - Always use a lock together with a condition variable
  - Always wait on some condition
  - Always use “while” instead of “if”
  - “broadcast” is usually correct, although not efficient. “signal” can have subtle problems.

Deadlock

- **Resources**
- Deadlock
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Deadlock Recovery

Resource (1)

- A **resource** is a commodity needed by a process.
- Resources can be either:
  - **Serially reusable:** e.g., CPU, memory, disk space, I/O devices, files.
    Acquire → use → release
  - **Consumable:** produced by a process, needed by a process; e.g., messages, buffers of information, interrupts.
    create → acquire → use
    Resource **ceases to exist** after it has been used, so it’s **not** released.

Resource (2)

- Resources can also be either:
  - **Preemptible:** e.g., CPU, central memory or
  - **Non-preemptible:** e.g., tape drives.
- And resources can be either:
  - Shared among several processes or
  - Dedicated exclusively to a single process.

Using Semaphores to Share Resources

```
Process P {
  ② down(&A);
  ③ down(&B);
     // use both resources
  ④ up(&B);
  ⑤ up(&A);
}
```

```
Process Q {
  ① // starts
  ⑥ down(&A);
  ⑦ down(&B);
     // use both resources
  ⑧ up(&B);
  ⑨ up(&A);
}
```

External semaphores A, B initialized to 1.

```
① A := 1, B := 1;
② A := 0, B := 1;
③ A := 0, B := 0;
④ A := 0, B := 1;
⑤ A := 1, B := 1; 👍
```

```
⑥ A := 0, B:= 1;
⑦ A := 0, B:= 0;
⑧ A := 0, B := 1;
⑨ A := 1, B := 1; 👍
```

But things can get tricky...

```
Process P() {
  ① // starts
  ② down(&A);
  ③ down(&B); // waiting…
  // use both resources
  up(&B);
  up(&A);
}
```

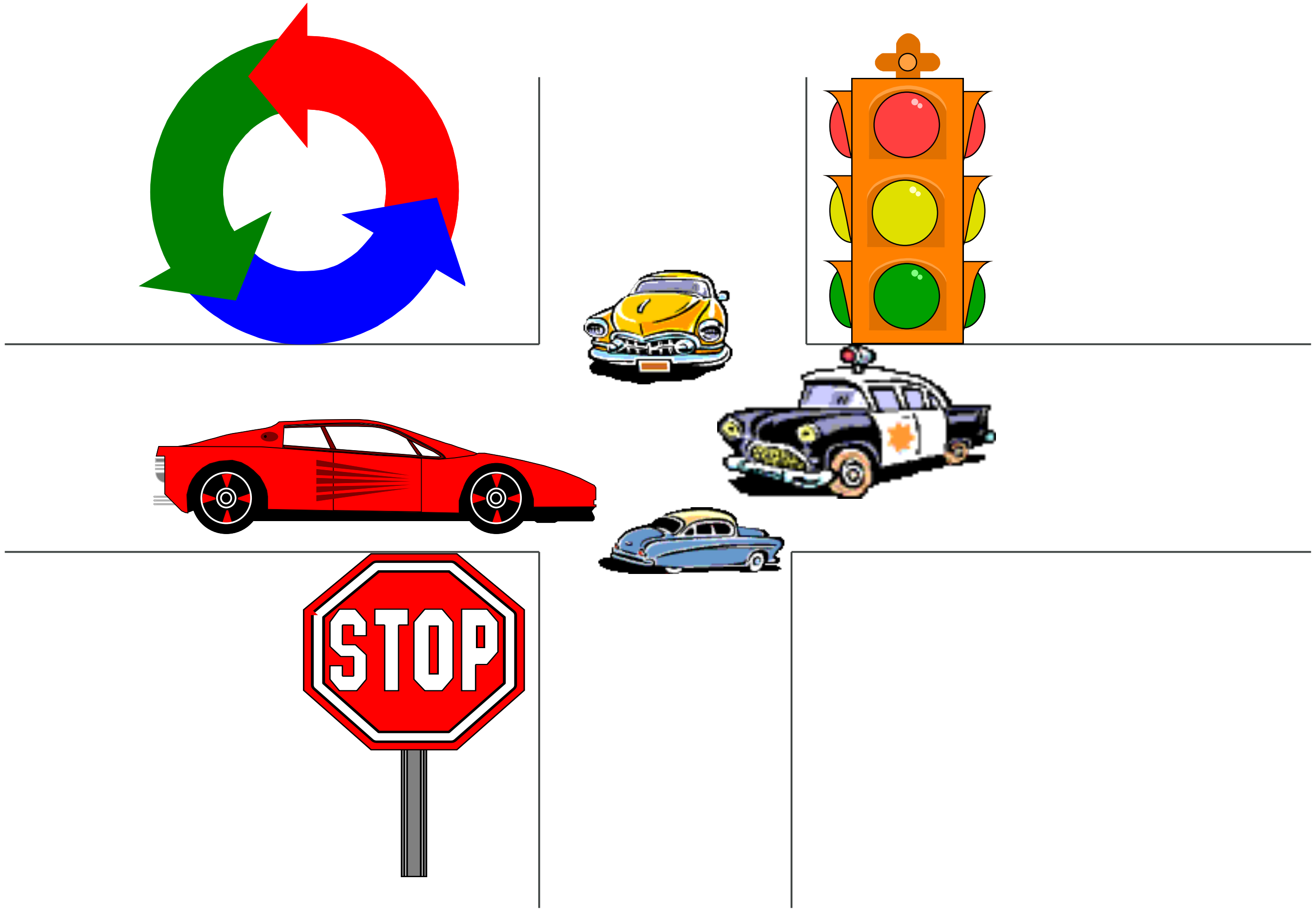
```
Process Q() {
  ④ down(&A);
  ⑤ down(&B); // waiting…
  // use both resources
  up(&A);
  up(&B);
}
```

External semaphores A, B initialized to 1.

```
① A := 1, B := 1;
② A := 0, B := 1;
③ A := 0, B := 0;
④ A := -1, B := 1;
⑤ A := -1, B := -1; 😱
```

DEADLOCK!

Deadlock in real life?



1. Roundabout
2. STOP sign at intersection
3. Traffic light?

Mechanisms for Deadlock Control

What is Deadlock?

- What is a deadlock?
  - A process is **deadlocked** if it is waiting for an event that will never occur.
  - Typically, but not necessarily, more than one process will be involved together in a deadlock (the deadly embrace).
- Is deadlock the same as starvation (or infinitely postponed)?
  - A process is **infinitely postponed** if it is delayed repeatedly over a long period of time while the attention of the system is given to other processes (i.e., logically the process may proceed but the system never gives it resources).
- What **conditions** should exist in order to lead to a deadlock?

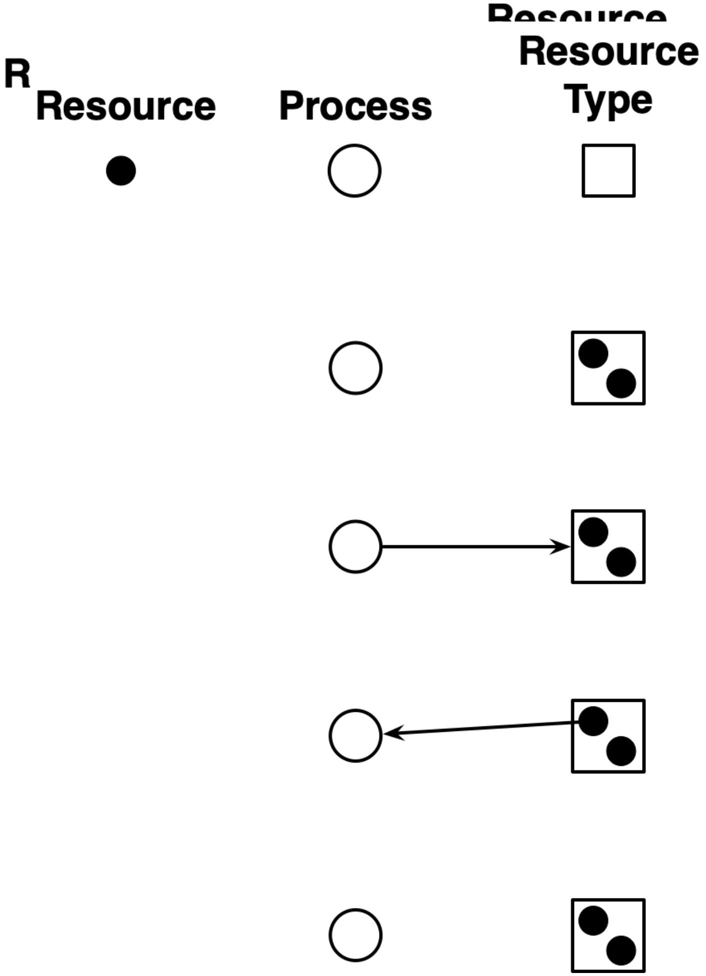
Conditions for Deadlock

- **Mutual exclusion:** Processes claim exclusive control of resources/locks
- **Hold-and-wait**: Processes will hold resources they already get while waiting for other resources/locks
- **No preemption:** OS cannot force a thread to release a resource/lock it holds
- **Circular wait:** a circular chain of threads such that each wait for the next one in the chain to release resource/lock

What is Deadlock?

- What is a deadlock?
  - A process is **deadlocked** if it is waiting for an event that will never occur.
  - Typically, but not necessarily, more than one process will be involved together in a deadlock (the deadly embrace).
- Is deadlock the same as starvation (or infinitely postponed)?
  - A process is **infinitely postponed** if it is delayed repeatedly over a long period of time while the attention of the system is given to other processes (i.e., logically the process may proceed but the system never gives it resources).
- What **conditions** should exist in order to lead to a deadlock?

Resource Allocation Graph (RAG)

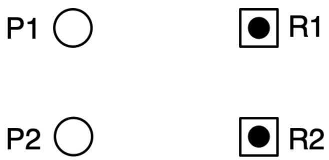


Scenario

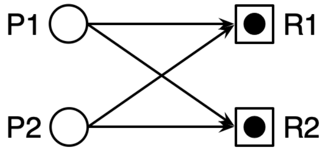
1. 1 process, 2 resources of the same type (e.g., Blu-Ray drives)
2. Process requests resource
3. Process is assigned resource
4. Process releases resource

Deadlock Model (RAG)

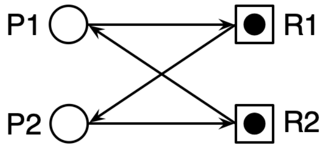
Scenario



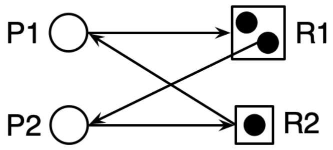
1(a). We have two processes, P1 and P2; there is one type of each resource (resource types R1 and R2).



1(b). Both P1 and P2 request one of R1 and one of R2.



1(c). We allocate R1 to P2 and R2 to P1. There's a cycle in the resource allocation graph (RAG), hence DEADLOCK. (For **one** resource of each type, a cycle in the RAG $\Longleftrightarrow$ a deadlock in the system.)



2. Suppose there are **two** instances of R1 and we have the same allocation as in 1(c). Deadlock does **not** occur in this case. (For **multiple** resources of each type, deadlock in the system $\Longrightarrow$ a cycle in the RAG.)

How to Deal with Deadlock?

- Resources
- Deadlock
- **Deadlock Prevention**
- **Deadlock Avoidance**
- **Deadlock Detection**
- **Deadlock Recovery**

How to Deal with Deadlock?

- **Prevention:** Design a system such that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Avoidance:** Impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.
- **Detection:** In a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery:** After a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and restarting them.

Deadlock Prevention and Avoidance

- Prevention:
  - Break circular wait
  - Break hold-and-wait
  - Allow preemption
  - Break mutual exclusion

- Avoidance:
  - Smart scheduling
  - Detect and Recover

Deadlock Prevention: The Ostrich Algo

- Guess: What is implemented in Linux for Deadlock Prevention?
- Do not do anything, simply restart the system (stick your head into the sand, pretend there is no problem at all)
- Rationale:
  - Make the common path faster and more reliable
  - Deadlock prevention, avoidance or detection/recovery algorithms are expensive
  - If deadlock occurs only rarely, it is not worth the overhead to implement any of these algorithms.

Deadlock Prevention: Havender’s Algorithms

- Break one of the deadlock conditions:
  - **Mutual exclusion**
    - **Solution:** exclusive use of resources is an important feature, but for some resources (virtual memory, CPU), it is possible.
  - **Hold-and-Wait condition**
    - **Solution:** Force each process to request all required resources at once. It cannot proceed until all resources have been acquired.
  - **No preemption condition**
    - **Solution:** Forcibly take away the resources assigned to the process due to lack of other requested resources.
  - **Circular wait condition**
    - **Solution:** Number all resource types; processes must request resources in numerical order

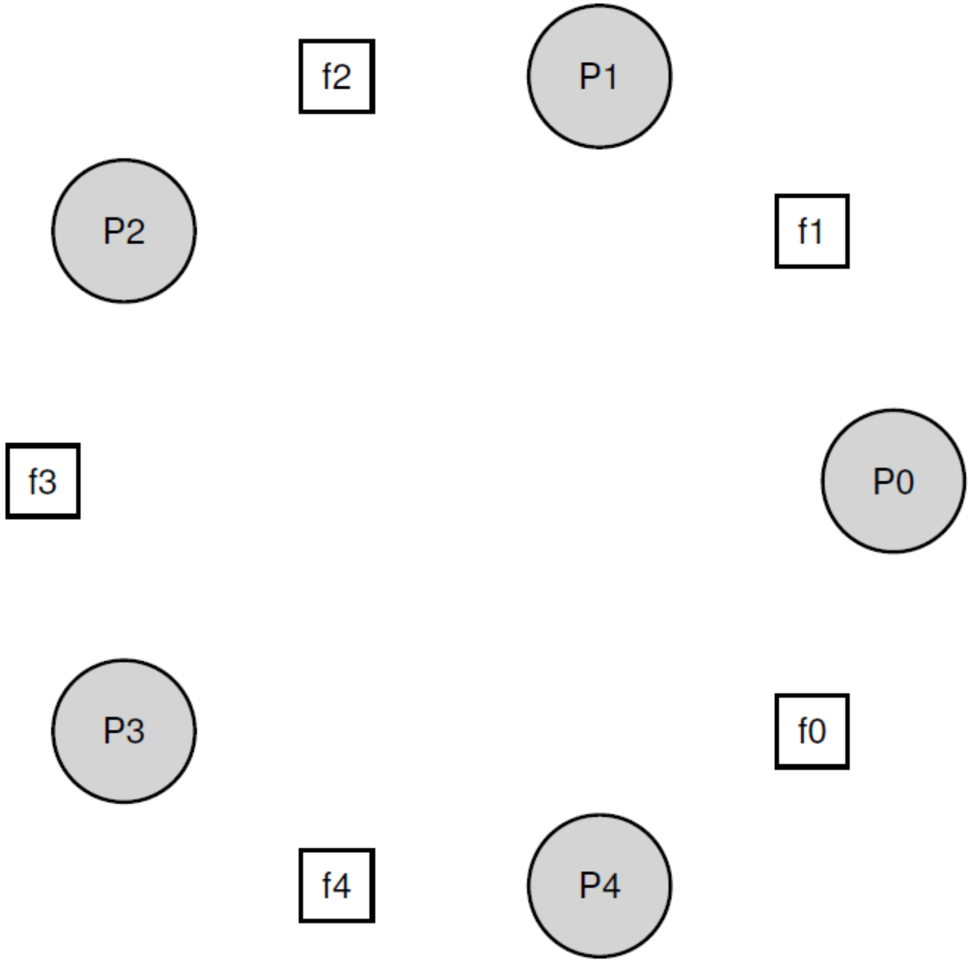
Two-Phase Locking

- Phase One
  - process tries to lock all records it needs, one at a time
  - If needed record found locked, start over
  - (No real work done in phase one)
- If phase one succeeds, start Phase Two
  - Perform updates
  - Release locks
- Note similarity to requesting all resources at once

Breaking circular wait

- Request one resource at a time.
- If some threads need to acquire more than one locks, always acquire locks in a predefined order
  - E.g. You can give a number to each lock; if a thread needs to acquire multiple locks, always acquire smaller locks first
- Programmers (you) need to do this.
- Try this rule on the dining philosopher's problem

The Dining Philosophers



Rule: each philosopher needs to grab the smaller fork first

Result:
P0-P3 will grab the left fork first; P4 will grab the right fork first; this breaks circular wait.

Breaking hold-and-wait

- Acquire all resources/locks together

```
pthread_mutex_lock(prevention);   // begin lock acquistion
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

- Problem: in many cases, a thread does not know which locks to acquire at the beginning of execution

Allow preemption

- If a thread finds it cannot acquire the next resource, it can release the previous resources it already gets

```
pthread_mutex_lock(L1);
if (pthread_mutex_trylock(L2) != 0) {
  pthread_mutex_unlock(L1);
  goto top;
}
```

- Problem: live lock
  - T1 gets lock1; T2 gets lock2; T1 wants to get lock2 and T2 wants to get lock1; they both quit; ……
  - If those threads try multiple times, they may get through, but there is no guarantee that they will get through.

Breaking mutual exclusion

- Lock-free data structures:
  - Recall that the synchronization problem comes from the fact that instructions may not be executed atomically
  - If hardware can provide atomicity guarantee, then we don’t need locks
  - Modern hardware does provide atomicity for a few instructions, and people have developed data structures with these instructions

- It is, in general, harder to reason about than the locking approach
  - Read the book if you are interested in it
  - I would not suggest you use lock-free approaches at this stage. Of course you can use libraries built by others.

Deadlock Prevention: Summary

Condition	How to Break It
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Deadlock Avoidance

- Look at the previous example again

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

- If OS is clever enough, it should realize that if T1 already acquired L1, then it should not allow Thread 2 to acquire L2, until T1 acquired L2.
- Banker's algorithm (by Dijkastra) can find when it is safe for a thread to acquire a lock.

Deadlock Avoidance

- The system needs to know resources needed ahead of time
- Banker's algorithm (**Dijkstra, 1965**)
  - Each customer tells banker the max. number of resources needed
  - Customer borrows resources from banker
  - Customer returns resources to banker
  - Customer eventually pays back loan
  - Banker only lends resources if the system will be in a safe state after the loan
- **Safe state:** there is a lending sequence where all customers can take out a loan
- **Unsafe state:** there is a possibility of deadlock

Banker's Algorithm

- Before any thread acquires a lock, the algorithm will check whether it is possible that allowing the thread to acquire the lock will create a deadlock in the future.
- Problem:
  - It is conservative. Sometimes it will report unsafe even if it is actually impossible to create a deadlock. This hurts performance.
  - It requires threads to know which locks they will acquire in the future.
- Although interesting in theory, it is rarely used in practice
  - Read the book yourself if you are interested.

Detect and Recover

- Detection: if your program freezes for a while, then it is probably a sign that your program actually experiences a deadlock.
  - Accurate detection requires building a wait-for graph and checking whether there is a cycle in it.

- Recovery: modern database systems can rollback the execution of some threads and retry them.
  - Unfortunately, modern OSes do not provide this functionality, because of its overhead.

Your Responsibility

- Since modern OSes don't provide any mechanisms to prevent or avoid deadlocks, it is your responsibility to ensure that your program does not have a deadlock in it.
  - You can use rules like "always grab locks in predefined order"
- There are tools to help you understand how a deadlock actually happened.
  - pstack (for C program) and jstack (for Java program)
  - Let's see an example

Synchronization in Java

- This is not required content. Will not appear in exams and hws.

- I teach this because Java is so popular today

Synchronization in Java

- In Java, an Object can serve as both a lock and a condition variable
- Use an Object as a lock: use the “synchronized” keyword
- Use an Object as a condition variable: an Object has three functions ： wait, notify, notifyAll. They are equivalent to pthread_cond_wait, pthread_cond_signal, and pthread_cond_broadcast.

Lock in Java

```
final Object lock = new Object();


synchronized(lock){
    ....
    ....
}
```

You can also define a function as “synchronized”. This is equivalent by putting all code in this function into a “synchronized(this)” block

Condition Variable in Java

```
final Object lock = new Object();

synchronized(lock){
    while(...)
        lock.wait()
    ....
}

synchronized(lock){
    ....
    lock.notify()/notifyAll()
}
```

Synchronization in Java

- Besides, Java provides many thread-safe data structures in `java.util.concurrent`
- Two very frequently use data structures:
  - LinkedBlockingQueue: this is Java’s version of bounded buffer
  - ConcurrentHashMap

Summary: Important Notes

- Synchronization is very important in OS when **accessing kernel data structures**
- System performance **may vary considerably**, depending on kind of sync. primitive selected
- **Rule of thumb adopted by kernel devs:** Always maximize system concurrency level
- Concurrency level **depends on two factors**:
  - Number of I/O devices that operate concurrently
  - Number of CPUs that do productive work
- To maximize I/O throughput, interrupts should be **disabled for short times**
- To use CPUs efficiently, sync primitives based on **spinlocks should be avoided** whenever possible
- **Choice of sync primitives** depends on kernel control flows, access data structures