More on Synchronization
Classic Problems and Deadlock

CPEN333 – System Software Engineering
2021 W2
University of British Columbia

Introduction

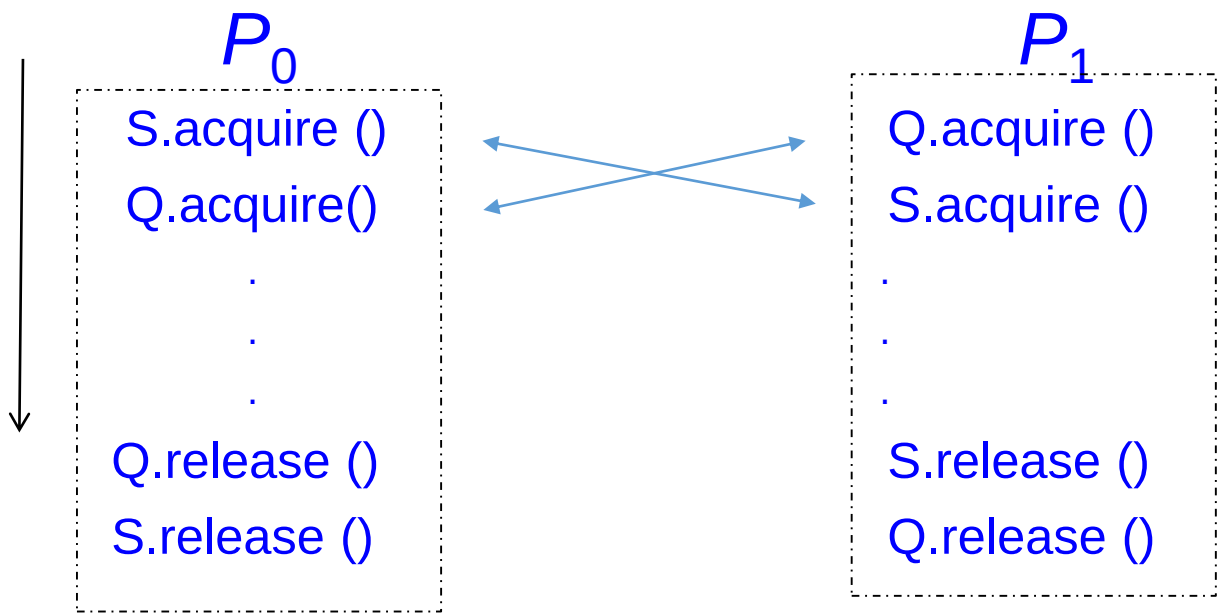
- We have already discussed the importance and application of synchronization among co-operating processes/threads.
- In this set of slides, we examine when deadlock can happen. We also discuss some classic synchronization problems.
- The discussed concepts may apply to either of process-based or thread-based multitasking systems.

Learning Objectives

- To introduce the concept of deadlock
- To examine classic problems of synchronization

Deadlock

- Deadlock happens when two or more tasks/entities (e.g. process or thread) are waiting indefinitely for an action/event that can be caused only by another waiting process, so neither one of them can proceed.
  - e.g. let S and Q be two semaphores initialized to 1



Deadlock is a possibility, if P0 acquires S, and P1 acquires Q, so P0 is waiting indefinitely to acquire Q and P1 is waiting indefinitely to acquire S.

Note

NOTE: we always want to avoid deadlock obviously, but to be able to demo the deadlock problem, in the next few examples, the code is written so that deadlock is possible or is likely or is guaranteed.

Example 1 (threading)

```
import threading

def worker1(lock1, lock2):
    lock1.acquire()
    print("DEBUG: lock1 acquired, lock2 is next")
    lock2.acquire()
    print("In critical section")
    lock2.release()
    lock1.release()

def worker2(lock1, lock2):
    lock2.acquire()
    print("DEBUG: lock2 acquired, lock1 is next")
    lock1.acquire()
    print("In critical section")
    lock1.release()
    lock2.release()

if __name__ == "__main__":
    mutex1 = threading.Lock()
    mutex2 = threading.Lock()

    t1=threading.Thread(target=worker1, args=(mutex1, mutex2))
    t1.start()
    t2=threading.Thread(target=worker2, args=(mutex1, mutex2))
    t2.start()
    t1.join()
    t2.join()
```

Note that if we run this code, deadlock is a possibility but we may run it many times and deadlock may never happen in any of those times.

Nevertheless, we must design our code so that it is guaranteed that deadlock does not happen.

Example 1b

```
import threading

def worker1(lock1, lock2):
    lock1.acquire()
    print("DEBUG: lock1 acquired, waiting for lock2")
    while not lock2.locked(): pass  #ensuring deadlock for demo
    lock2.acquire()
    print("In critical section")
    lock2.release()
    lock1.release()

def worker2(lock1, lock2):
    lock2.acquire()
    print("DEBUG: lock2 acquired, waiting for lock1")
    while not lock1.locked(): pass  #ensuring deadlock for demo
    lock1.acquire()
    print("In critical section")
    lock1.release()
    lock2.release()

if __name__ == "__main__":
    mutex1 = threading.Lock()
    mutex2 = threading.Lock()

    t1=threading.Thread(target=worker1, args=(mutex1, mutex2))
    t1.start()
    t2=threading.Thread(target=worker2, args=(mutex1, mutex2))
    t2.start()
    t1.join()
    t2.join()
```

output:

```
DEBUG: lock1 acquired, waiting for lock2
DEBUG: lock2 acquired, waiting for lock1
```

Here by using the `while`, we ensure that worker1 acquires lock1 and worker2 acquires lock2, so a deadlock is ensured.

This is just a demonstration for a case where deadlock happens.

Example 1c (using `with`)

- The example in the previous slide written using `with`:

```
import threading

def worker1(lock1, lock2):
    with lock1:
        print("DEBUG: lock1 acquired, waiting for lock2")
        while not lock2.locked(): pass  #ensuring deadlock for demo
        with lock2:
            print("In critical section")


def worker2(lock1, lock2):
    with lock2:
        print("DEBUG: lock2 acquired, waiting for lock1")
        while not lock1.locked(): pass  #ensuring deadlock for demo
        with lock1:
            print("In critical section")

if __name__ == "__main__":
    mutex1 = threading.Lock()
    mutex2 = threading.Lock()

    threading.Thread(target=worker1, args=(mutex1, mutex2)).start()
    threading.Thread(target=worker2, args=(mutex1, mutex2)).start()
    #join omitted to be concise
```

output:

```
DEBUG: lock1 acquired, waiting for lock2
DEBUG: lock2 acquired, waiting for lock1
```

Example 2 (multiprocessing)

```
import multiprocessing
import time, random

def worker1(lock1, lock2):
    lock1.acquire()
    print(f"DEBUG {multiprocessing.current_process().name}: lock1 acquired, waiting for lock2")
    time.sleep(random.random()) #increasing deadlock possibility (0 to 1 sec wait randomly)
    lock2.acquire()
    print("In critical section")
    lock2.release()
    lock1.release()

def worker2(lock1, lock2):
    lock2.acquire()
    print(f"DEBUG {multiprocessing.current_process().name}: lock2 acquired, waiting for lock1")
    time.sleep(random.random()) #increasing deadlock possibility (0 to 1 sec wait randomly)
    lock1.acquire()
    print("In critical section")
    lock1.release()
    lock2.release()

if __name__ == "__main__":
    mutex1 = multiprocessing.Lock()
    mutex2 = multiprocessing.Lock()

    t1=multiprocessing.Process(target=worker1, args=(mutex1, mutex2))
    t1.start()
    t2=multiprocessing.Process(target=worker2, args=(mutex1, mutex2))
    t2.start()
    t1.join()
    t2.join()
```

This example is different in two aspects:

1. It is using multiprocessing
2. deadlock is quite likely but not guaranteed (we may need to run it a few times for deadlock to happen, as we are using random delays).

output possibility 1 (deadlock):

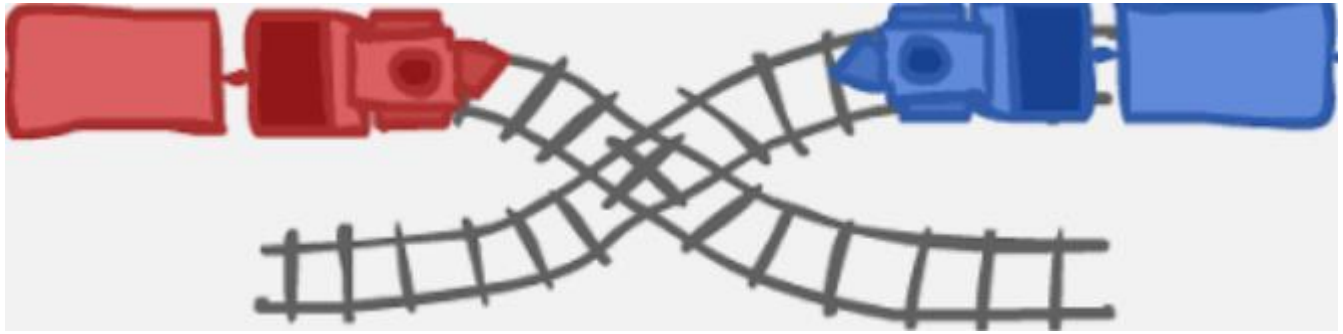
```
DEBUG Process-1: lock1 acquired, waiting for lock2
DEBUG Process-2: lock2 acquired, waiting for lock1
```

output possibility 2 (process-2 happens to be able to acquire both locks first, so no deadlock):

```
DEBUG Process-2: lock2 acquired, waiting for lock1
In critical section
DEBUG Process-1: lock1 acquired, waiting for lock2
In critical section
```

An old train safety law

- A 20th century Kansas legislation stipulates: "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone"



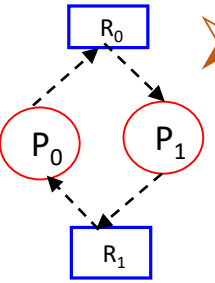
An example of a deadlock
(as per above)

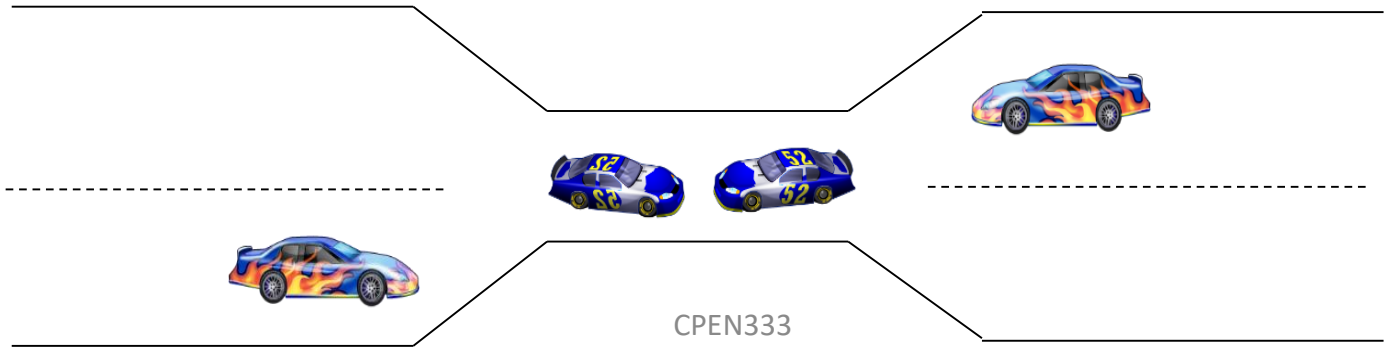
Deadlock conditions

- A deadlock situation can arise if these four conditions hold simultaneously:
  - **Mutual exclusion**
  - **Hold and wait**
  - **No preemption**
  - **Circular wait**

Deadlock conditions

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes (similarly threads) such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.





Deadlock Avoidance

- One way to avoid deadlock is to ensure that at least one of the necessary conditions for deadlock cannot occur.

- Some examples:
  - for Hold and Wait: e.g. must guarantee that whenever a process requests a resource, it does not hold any other resources
  - for No Preemption: e.g. if a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - for No Circular Wait: e.g. impose a total ordering of all resource types
  - …

Problems with Semaphores

- Similarly, using semaphores incorrectly can result in timing errors that are difficult to detect
  - these errors happen if some particular execution sequence take place

- Some possible incorrect use of semaphore operations:
  - incorrect order
    - e.g. in this example, mutual exclusion is violated

```
semaphore1.release()
    …
semaphore1.acquire()
```

  - incorrect use
    - e.g. in this example, deadlock will occur

```
semaphore1.acquire()
    …
semaphore1.acqurie()
```

  - Omitting **`acquire()`** or **`release()`** (or both)
    - either mutual exclusion is violated or deadlock will occur

Starvation

- In a deadlock, processes never finish executing, and system resources are possibly tied up, preventing other jobs from starting.
  - e.g. a process may never be removed from the semaphore queue in which it is suspended and is blocked indefinitely.

- Starvation then can happen: A task (e.g. process or thread) is denied resources it needs to proceed or to work with.

Classical Problems of Synchronization

- We are going to discuss a number of synchronization problems that are used for testing newly proposed synchronization schemes:
  - A consumer-producer (bounded-buffer) Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

A producer-consumer problem

- As discussed before, the buffer has *N* buffers (buffer spaces), each can hold one item
- The buffer is bounded, e.g. circular buffer.

- We present a solution using one binary and two counting semaphores.
- semaphore mutex, provides mutual exclusion (initialized to the value 1)
- Semaphore full, counts the # of full buffers (initialized to the value 0)
- Semaphore empty, counts the # of empty buffers (initialized to the value N)

- `mutex` is used to guard the critical section to ensure mutual exclusion (note that alternatively, we could use a mutex Lock instead)
- Here we can consider `full` and `empty` as counters with system-supported atomic operations (`acquire` and `release`) used to synchronize the use of the bounded-buffer.

Producer-consumer (cont.)

- An implementation:

The structure of the producer process:

```
#  produce an item in nextp
empty.acquire();
mutex.acquire();
#  add the item in nextp to the buffer
mutex.release();
full.release();
```

The structure of the consumer process:

```
full.acquire();
mutex.acquire();
# remove an item from buffer to nextc
mutex.release();
empty.release();
# consume the item in nextc
```

Note: We can use **`with`** for `mutex`, but not `full` and `empty`

- Recall the low-level implementation of the `acquire` and `release` methods of a semaphore we discussed previously:
  - `acquire` checks and decrements its value (under the hood) and `release` increments it (again under the hood).

Readers-Writers Problem

- Suppose a dataset is to be shared among a number of concurrent processes
  - *Readers* are the processes that only read the dataset; they do **not** perform any updates
  - *Writers* are the processes that can both read and write

- Readers-writers synchronization problem:
  - allows multiple readers to read at the same time
  - only one single writer can access the shared data at any time

- So not allowing “a writer and a writer” or “a writer and a reader” process to access at the same time

Readers-writers synchronization problem

- Let's consider a problem in which requires that no reader be kept waiting unless a writer already obtained permission to use the shared object.

- For the solution, the processes share the following:
  - Semaphore wrt (initialized to 1)
    - used to ensure mutual exclusion for the writers.
    - also used by the first or the last reader that enters or exits the critical section.
  - Semaphore mutex (initialized to 1)
    - used by readers
    - used to ensure mutual exclusion when variable readCount is checked/updated.
  - Integer readCount (initialized to 0)
    - used to keep track of how many processes/threads are currently reading

Readers-Writers Problem (cont.)

- The following code segments present a solution to the readers-writers problem stated in the previous slide:

The structure of a writer process:

```
while True:
    wrt.acquire()
    # writing is performed
    wrt.release()
```

The structure of a reader process:

```
while True:
    mutex.acquire()
    readCount += 1
    if readCount == 1:
        wrt.acquire()
    mutex.release()
    # reading is performed
    mutex.acquire()
    readCount -= 1
    if readCount == 0:
        wrt.release()
    mutex.release()
```

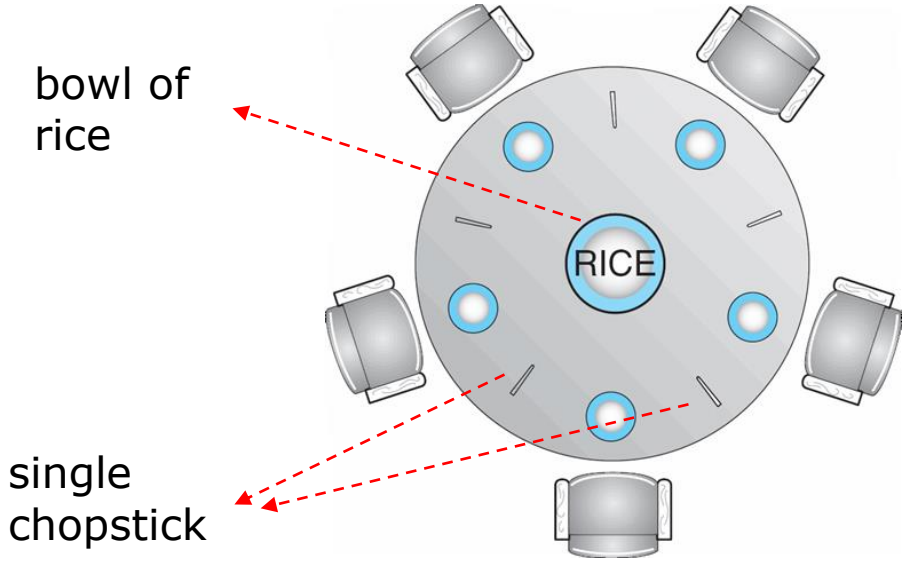
Readers-Writers Problem (cont.)

- Q: May the discussed solution result in starvation?
  - Yes, for example writers may starve.

- There are different variants of the problem.
  - For example, a *second variant of the* problem may require that once a writer is waiting to access the object, no new readers may start reading.

Dining-Philosophers Problem

- The dining philosophers problem is considered a classic synchronization problem
- Consider five philosophers, who spend their lives thinking and eating:



- One simple solution starts with representing each chopstick with a semaphore
  - Five Semaphores store in the list chopstick (each initialized to 1)

Dining-Philosophers Problem (Cont.)

The structure of Philosopher `i`:

```
chopstick[i].acquire()

chopstick[(i + 1) % 5].acquire()

#  eat

chopstick[i].release()

chopstick[ (i + 1) % 5].release()

#  think
```

- The above solution though may create a deadlock. To remedy that, we may:
  - allow at most four philosophers to be sitting simultaneously, or
  - allow a philosopher to pick up her chopsticks only if both chopsticks are available, or
  - use an asymmetric solution, e.g. an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right first and then her left chopstick.

References

- Some sections of chapter 5 and some sections of chapter 7 of Operating Systems Concepts

Acknowledgement: This set of slides is partly based on the PPTs provided by the Wiley's companion website for the operating system concepts book (including textbook images, when not explicitly mentioned/referenced).