Synchronization

CPEN333 – System Software Engineering

2021 W2

University of British Columbia

Introduction

- So far we have written process-based and thread-based multitasking programs, however, we have not allowed any shared data used between them.

- If a number of tasks (process or thread) have simultaneous access to shared data, data inconsistency problem may arise.

- In this set of slides, we examine related concepts of the critical section, race condition, and the synchronization concept to avoid data consistency issues.

Learning Objectives

- To introduce the race condition
- To introduce the critical-section problem
- To present both software and hardware solutions of the critical-section problem to ensure the consistency of shared data
  - Locks
  - Synchronization Hardware
  - Semaphores

NOTE

- The concepts we discuss here applies to any multi-tasking system regardless of being process-based or thread-based.

- For this general introduction to synchronization, we focus more on synchronization between cooperating processes.

Cooperating processes

- Processes executing concurrently in the OS may be either:
  - **Independent** process cannot affect or be affected by the execution of another process, or
  - **Cooperating** process can affect or be affected by the execution of another process

- Cooperating processes can either directly share a logical address or be allowed to share data through messages or files.
  - Concurrent access to shared data may result in data inconsistency
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

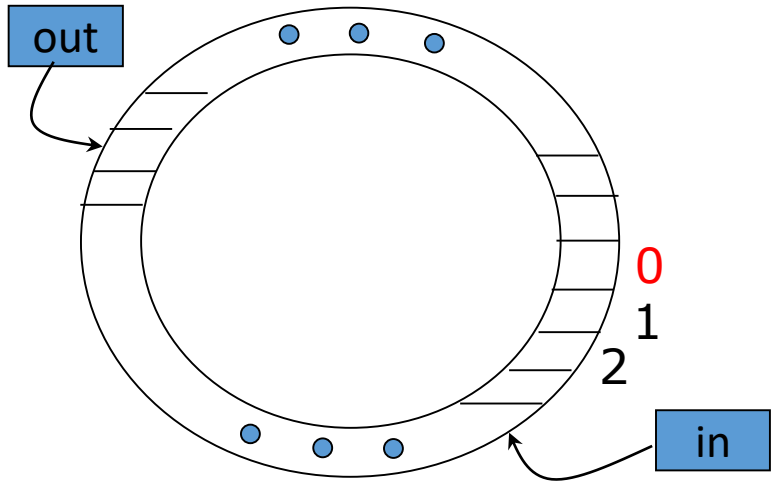
Example: Producer-Consumer Paradigm

- To describe the problem, let's look at an example.
- A producer process produces information that is consumed by a consumer process.
  - e.g. a compiler produces assembly code, which is consumed by the assembler.
- One solution to the producer-consumer problem uses shared memory
  - We must have available a buffer of items that can be filled by the producer and emptied by the consumer.
  - The producer and consumer must be synchronized
  - For practical reasons, we consider a *bounded-buffer* (a fixed buffer size).

Bounded-Buffer Shared-Memory Solution

- The following variables reside in a region of memory shared by the producer and consumer processes.

```
#define BUFFER_SIZE 10
typedef struct {
/* . . . */
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



- The above shared buffer is implemented as a circular array with two logical pointers: in and out.
- Bounded-buffer: The above is one correct solution, but it can only use BUFFER_SIZE-1 elements

Note: since the concept can be explained better with lower-level memory management, we are using C here.

Example: Producer-Consumer Paradigm

- Suppose that we want to provide a solution to the consumer-producer problem that fills all the buffers.

- We can do so by using an integer counter that keeps track of the number of full buffers.

- Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer-Consumer with *counter*

```c
while (true) {
        /*produce an item and put in nextProduced*/
         while (counter == BUFFER_SIZE)
                  ; /* do nothing */
          buffer[in] = nextProduced;
          in = (in + 1) % BUFFER_SIZE;
          counter++;
}
```

```c
while (true)  {
         while (counter == 0)
                 ; /* do nothing */
          nextConsumed =  buffer[out];
          out = (out + 1) % BUFFER_SIZE;
         counter--;
          /*  consume the item in nextConsumed */
}
```

C

```python
while True:
    #produce an item and put in nextProduced
    while counter == BUFFER_SIZE:
        pass # do nothing
    buffer[in] = nextProduced
    in = (in + 1) % BUFFER_SIZE
    counter += 1
```

Producer process

```python
while True:
    while counter == 0:
        pass #do nothing
    nextConsumed = buffer[out]
    out = (out + 1) % BUFFER_SIZE
    counter -= 1
    #consume the item in nextConsumed
```

Consumer process

Python

Race Condition

- We now show that the value of counter may be incorrect when both the producer and consumer execute concurrently, as follows:

`counter++` could be implemented as:

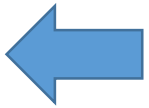
```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--` could be implemented as:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving, with "counter = 5" initially:

S0: producer execute register1 = counter {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = counter {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute counter = register1 {counter = 6 }
S5: consumer execute counter = register2 {counter = 4}



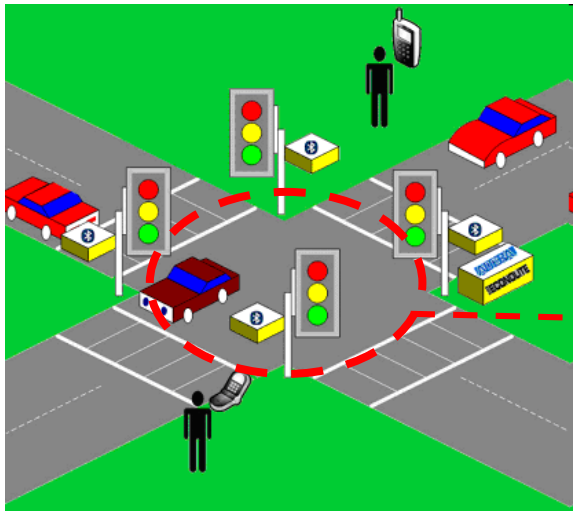
Not what we expect!

Race Condition (cont.)

- A situation like this is called a race condition,
  - where several processes access and manipulate the same data concurrently and
  - the outcome of the execution depends on the particular order in which the access is taken place.

Critical Section

- The critical-section problem is to design a protocol that the processes can use to cooperate.

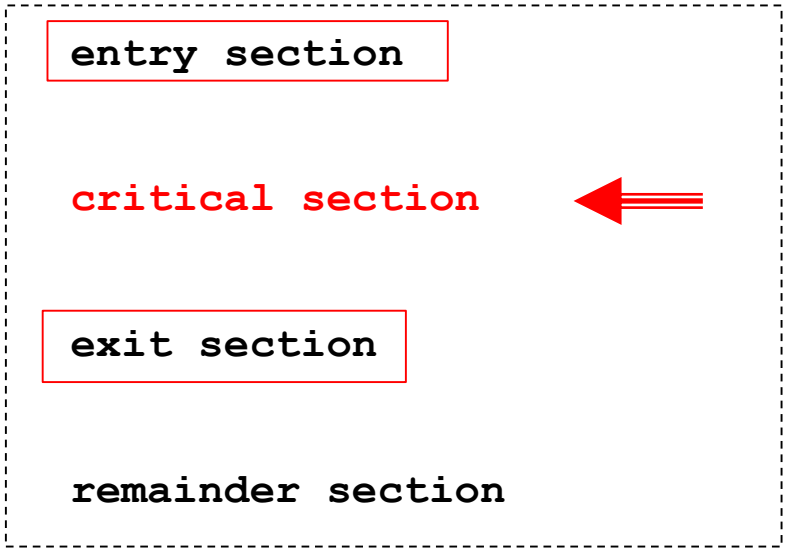


An good analogy for a critical section

- Consider a system consisting of *n* processes $\{P_0, P_1, \ldots, P_{n-1}\}$
  - Each process has a segment of code called a *critical section*.
  - When one process is executing in its critical section, no other process is allowed to execute in its critical section.
  - Each process must request permission to enter its critical section.

General Structure

- Each process must somehow request permission to enter its critical section.
- The entry section is the section of code that implements this request.
- The critical section may be followed by an exit section.
- The remaining code is the remainder section.



Solution to Critical-Section Problem

- A solution to the critical section problem must satisfy three requirements:
  1. Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
  2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
  3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
     - Assume that each process executes at a nonzero speed,
     - however, no assumption concerning relative speed of the n processes is made

Lock

- We can generally state that any solution to the critical-section requires a simple tool, a lock.

- The mutex lock (mutual exclusion) is the simplest of such tools.

```
acquire lock
critical section
release lock
remainder section
```

Synchronization Hardware

- Uniprocessor system: the critical-section problem could be solved if we could disable interrupts (e.g. in non-preemptive kernels)
  - since current sequence of instructions would execute without preemption
  - this approach is generally too inefficient on multiprocessor systems
- Modern computer systems provide special hardware instructions that allow us to atomically:
  - either test/modify the content of a word
  - or to swap the contents of two words
  - Atomic = as one non-interruptable or indivisible unit
- Example: Intel instruction set has atomic increment (with LOCK prefix):

```
LOCK INC [value] ;increment atomically
```

See details here: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

Semaphore

- A semaphore is one of the oldest synchronization primitives.
  - A semaphore s is an integer variable
  - It is used to control access to common resources
  - S can be accessed only through two standard atomic operations:
    - `acquire_lowlevel()` and `release_lowlevel()`

```
void acquire_lowlevel (S)
{
    while (S <= 0)
        ; /* no-op */
     S--;
}
```

```
void release_lowlevel (S)
{
        S++;
}
```

    - All modifications to the semaphore in these operations must be executed indivisibly
    - These two operations are also called wait() and signal().
      - They were originally called P() and V().

Semaphore

- There are two types of semaphores: **counting** and **binary**

- The value of a counting semaphore can range over an unrestricted domain
  - e.g. set to the # of resources available
- The value of a binary semaphore can range only between 0 and 1;
  - binary semaphores behave like (and on most systems are also known as) mutex locks (***mut***ual ***ex***clusion),
  - and are simpler to implement

- Semaphores can be used to solve various synchronization problems.

Semaphore Implementation

- The main disadvantage of the previous semaphore definition is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously.
  - This type of semaphore is also called a spinlock because the process spins while waiting for the lock.
- This continuous looping is clearly a problem as it wastes CPU cycles that some other process might be able to use productively.
  - On the other hand, spinlocks do have an advantage in that no context switch is required when the process must wait on a lock,
  - so, spinlocks are useful, when locks are expected to be held for short times.

Semaphore Implementation with no Busy waiting

- To overcome the need for busy waiting, we can modify **`acquire`** and **`release`** operations.
  - Rather than engaging in busy waiting, the process can *block* itself.

- The following two operations (provided by OS as basic system calls) are used:
  - **`block`** – The ***`block()`*** operation suspends the process invoking the operation and places it on the appropriate waiting queue.
  - **`wakeup`** – The ***`wakeup()`*** operation removes one of processes in the waiting queue and place it in the ready queue (i.e. resuming the operation of a blocked process).

References

- Some sections from chapter five 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).