Chapter 5 summary

- Processes can execute concurrently.
    - May be interrupted at any time, partially executed.
- Concurrent access to shared data may result in **data inconsistency**.
- Maintaining data consistency requires mechanisms to ensure orderly execution of cooperating processes.
- Consumer-producer problem:
    - Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers.
    - We can do so by having an **integer counter** that keeps track of the number of full buffers.
    - Initially, counter is set to **0** and is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.
- A **race condition** is a situation that may occur inside a **critical section**. This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute.
- Each process has critical section segment of code.
    - Process may be changing common variables, updating table, writing file, etc.
    - When one process in critical section, no other may be in its critical section.
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**.
- Solution to Critical-Section Problem:
    - **Mutual Exclusion** - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
    - **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.
    - **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.
        - No assumption concerning relative speed of the n processes.

- Peterson's solution:
  - **Two** process solution.
  - Assume that the **load** and **store** instructions are **atomic**; that is, cannot be interrupted.
  - The two processes share two variables:

    int **turn**;

    Boolean **flag[2]**
  - The variable **turn** indicates whose turn it is to enter the critical section.
  - The **flag** array is used to indicate if a process is ready to enter the critical section.

    flag[i] = true implies that process Pi is ready!
- Synchronization hardware:
  - Many systems provide hardware support for critical section code.
  - All solutions below based on idea of **locking**.
    - Protecting critical regions via locks.
  - Uniprocessors – could disable interrupts.
    - Currently running code would execute without preemption.
    - Generally, too inefficient on multiprocessor systems.
      - Operating systems using this not broadly scalable.
  - Modern machines provide special atomic (non-interruptible) hardware instructions:
    - Either test memory word and set value.
    - Or **swap** contents of two memory words.
- Mutex Locks:
  - Protect critical regions with it by first **acquire()** a lock then **release()** it.
  - Boolean variable indicating if lock is available.
  - Calls to **acquire()** and **release()** must be **atomic**.
    - Usually implemented via hardware atomic instructions.
  - But this solution requires **busy waiting**.
    - This lock therefore called a **spinlock**.
- Semaphore:
  - Synchronization tool that does not require **busy waiting**.
  - Semaphore **S** – integer variable
    - Two standard operations modify **S**:**wait()** and **signal()**
      - Originally called **P()** and **V()**
  - Less complicated.
  - **Counting semaphore** – integer value can range over an unrestricted domain.
  - **Binary semaphore** – integer value can range only between 0 and 1.
    - Like a mutex lock.

- Semaphore implementation:
    - Must guarantee that no two processes can execute wait() and signal() on the same semaphore at the same time.
    - Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
        - Could now have **busy waiting** in critical section implementation.
            - But implementation code is short.
            - Little busy waiting if critical section rarely occupied.
    - Note that applications may spend lots of time in critical sections and therefore this is not a good solution.
- Semaphore implementation with no busy waiting:
    - With each semaphore there is an associated **waiting queue**.
    - Each entry in a waiting queue has two data items:
        - value (of type integer).
        - pointer to next record in the list.
    - Two operations:
        - **block** – place the process invoking the operation on the appropriate waiting queue.
        - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- **Starvation** – **indefinite blocking** – A process may never be removed from the semaphore queue in which it is suspended.
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process.
- Bounded-Buffer Problem:
    - **n** buffers, each can hold one item.
    - Semaphore **mutex** initialized to the value **1**.
    - Semaphore **full** initialized to the value **0**.
    - Semaphore **empty** initialized to the value **n**.
- Readers-Writers Problem:
    - A data set is shared among a number of concurrent processes.
        - **Readers** – only read data set and do not perform any updates.
        - **Writers** – can both read and write.
    - Problem:
        - Allow **multiple readers** to read at the same time.
        - Only one **single writer** can access the shared data at the same time.

- Readers-Writers Problem Variations:
    - **First variation** – no reader kept waiting unless writer has permission to use shared object.
    - **Second variation** – once writer is ready, it performs write ASAP.
    - Both may have **starvation** leading to even more variations.
    - Problem is solved on some systems by kernel providing **reader-writer locks**.
- Dining-Philosophers Problem:
    - The dining-philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begin thinking again.
    - Deadlock and starvation are possible.
    - Incorrect use of semaphore operations:
        - signal (mutex) …. wait (mutex)
        - wait (mutex) … wait (mutex)
        - Omitting of wait (mutex) or signal (mutex) (or both)