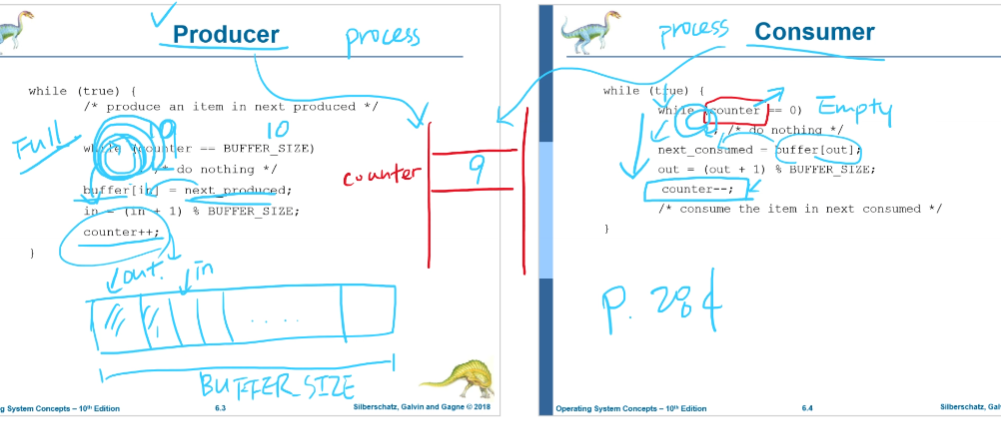
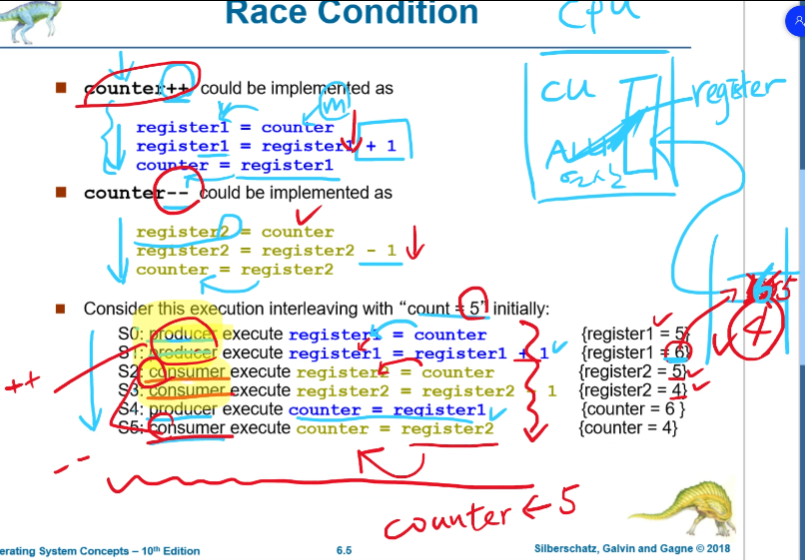
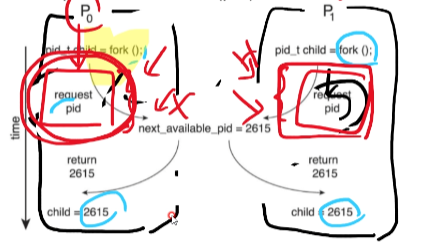
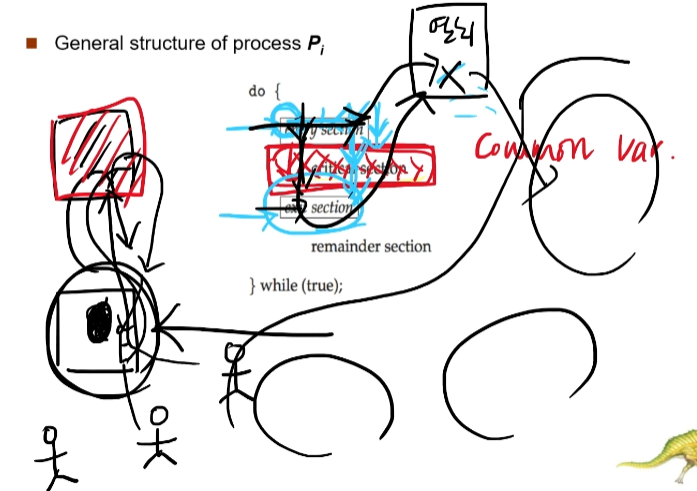
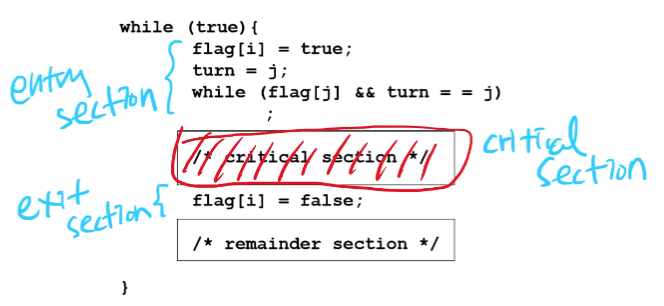
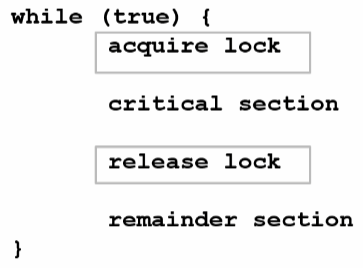
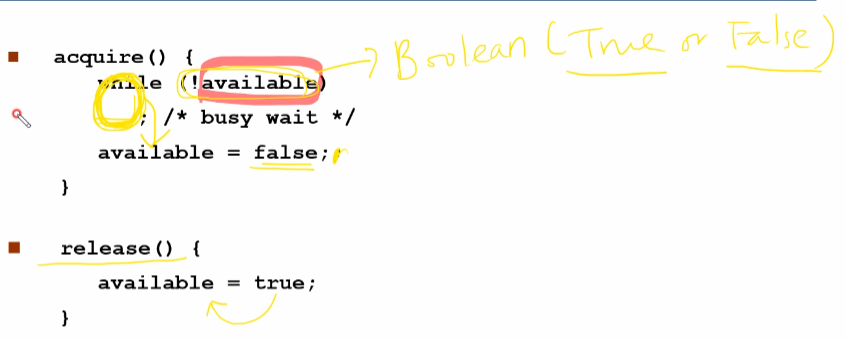
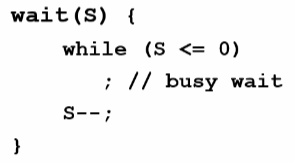
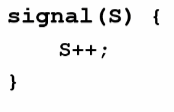
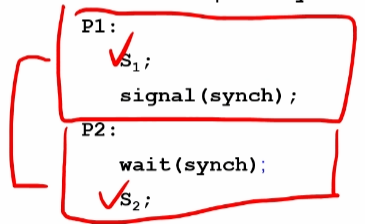
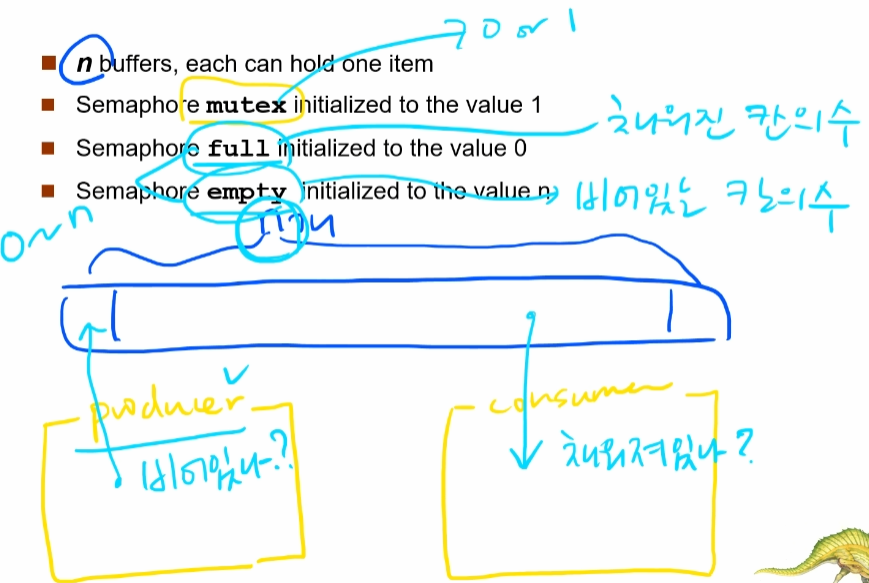
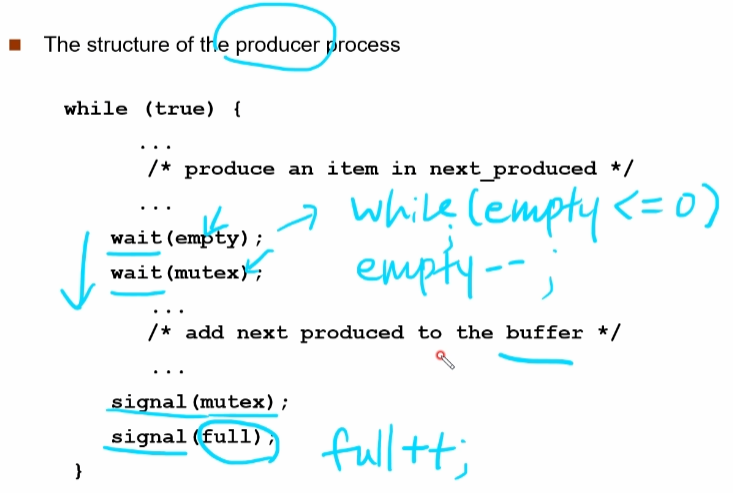
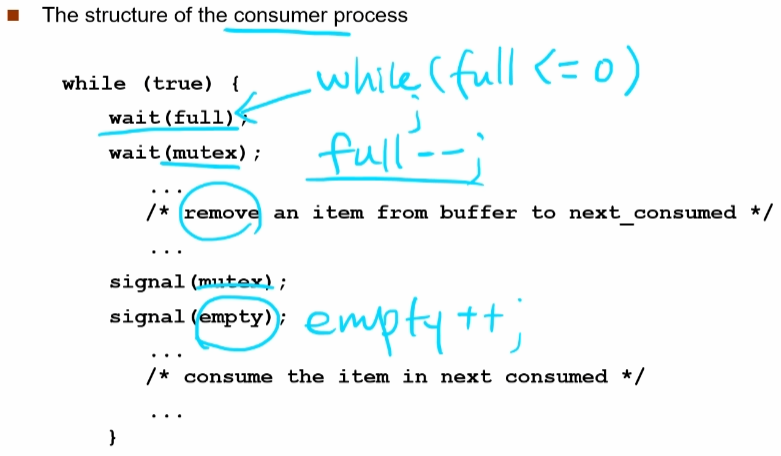
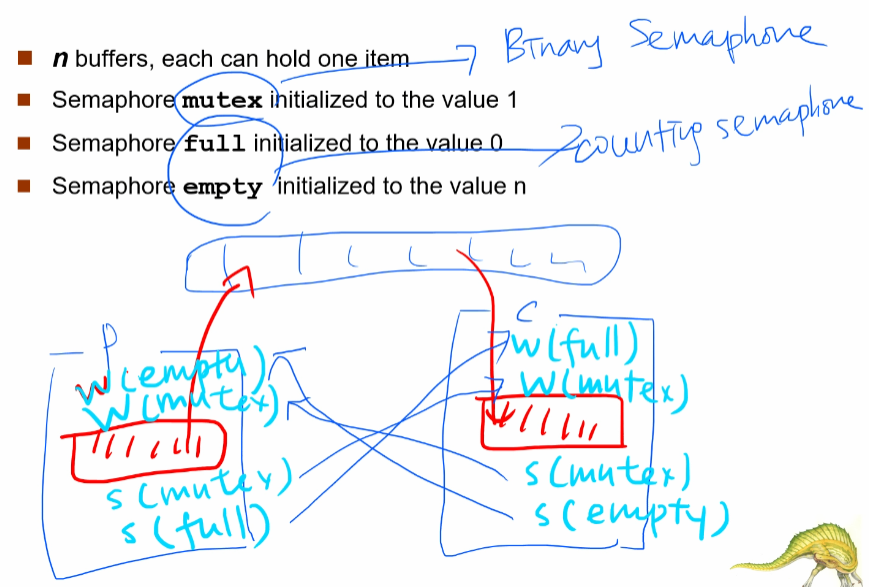
Chapter 6: Process Synchronization; 프로세스 동기화

Synchronization Tolls; 동기화 도구들

* Processes can execute concurrently  
  프로세스는 동시에 실행할 수 있다
  + May be interrupted at any time, partially completing execution  
    언제든지 중단될 수 있으며 부분적으로 완료될 수 있다
* Concurrent access to shared data may result in data inconsistency  
  공유 데이터 동시 접속 시 데이터 불일치가 발생할 수 있다
* Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes  
  데이터의 일관성을 유지하기 위해서는 협력 프로세스의 질서 있는 실행을 보장하는 메커니즘이 필요하다
* Illustration of the 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. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

1. Background; 배경  
     
   
2. The Critical-Section Problem; 임계구역 문제
   1. Process P0 and P1 are creating child process using the fork() system call
   2. Race condition on kernel variable next\_available\_pid which represents the next available process identifier (pid)  
      
   3. Unless there is mutual exclusion, the same pid could be assigned to two different processes!  
      상호 배제가 없는 한, 동일한 pid를 두 개의 다른 프로세스에 할당할 수 없다
   4. Consider system of n processes {P0, P1, Pn-1}
   5. Each process has critical section segment of code  
      각 프로세스의 코드에는 크리티컬 섹션이 있다
      1. Process may be changing common variables, updating table, writing file, etc  
         공통변수 변경, 표 갱신, 파일 작성 등
      2. When one process in critical section, no other may be in its critical section  
         한 프로세스가 자기의 크리티컬 섹션에 들어가 있다면 다른 프로세스는 아무도 크리티컬 섹션에 들어갈 수 없다
   6. Critical section problem is to design protocol to solve this
   7. Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section  
      각 프로세스는 엔트리 섹션에서 크리티컬 섹션으로 진입하려면 허가를 요청해야한다. 크리티컬 섹션 뒤에는 엑시트 섹션이 따라올 수 있고 코드의 나머지 부분들은 총칭하여 리메인더 섹션이라고 부른다  
      
   8. Solution to Critical-Section Problem
      1. Mutual Exclusion – If process Pi is executing in its critical section, then no other processes can be executing in their critical sections  
         상호 배제 – 만약 프로세스가 크리티컬 섹션에서 실행 중이라면, 어떤 프로세스든 그들의 크리티컬 섹션에서 실행될 수 없다
      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  
         진행 – 만약 크리티컬 섹션에서 실행 중인 프로세스가 없고 몇몇 프로세스들이 크리티컬 섹션에 존재하려고 한다면, 다음으로 크리티컬 섹션에 진입할 프로세스의 선정을 무기한 연기할 수 없다
      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  
         한정된 대기 – 프로세스가 자신의 크리티컬 섹션에 진입하려는 요청을 한 후부터 해당 요청이 허가되기까지 다른 프로세스가 해당 크리티컬 섹션에 진입할 수 있는 횟수에 한계가 있어야 한다
         1. Assume that each process executes at a nonzero speed  
            각 프로세스의 실행 속도가 0이 아닌 것으로 가정한다
         2. No assumption concerning relative speed on the n processes  
            n개 프로세스의 상대 속도에 대한 가정은 없다
3. Peterson’s Solution; Peterson의 해결안
   1. Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)  
      현대 구조에서 올바르게 작동한다고 보장할 수 없다
   2. Two process solution
   3. Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted  
      load와 store 같은 기본적인 기계어를 수행하는 방식이 아토믹하다고 가정; 중단될 수 없다
   4. The two processes share two variables:
      1. int turn;
      2. Boolean flag[2]
   5. The variable turn indicates whose turn it is to enter the critical section
   6. 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!  
      
4. Mutex Locks (Mutual Exclusion, 상호 배제)  
   Previous solutions are complicated and generally inaccessible to application programmers  
   이전의 솔루션들은 복잡하고 일반적으로 프로그래머가 어플리케이션으로 액세스 할 수 없음  
   OS designers build software tools to solve critical section problem  
   OS 설계자들이 소프트웨어 도구들을 크리티컬 섹션 문제를 해결할 수 있게 개발함  
   Simplest is mutex lock  
   가장 간단한 것은 상호 배제 잠금  
   Protect a critical section by first acquire() a lock then release() the lock  
   크리티컬 섹션을 보호하는 게 먼저, acquire (취득하다) release (풀어주다)  
    ・Boolean variable indicating if lock is available or not  
    부울 변수가 잠금이 가능한지 아닌지 나타낸다  
   Calls to acquire() and release() must be atomic  
   acquire() release() 호출은 원자적이어야 한다  
    ・Usually implemented via hardware atomic instructions such as compare-and-swap  
    보통 비교-스왑과 같은 하드웨어 원자적 명령을 통해 구현됨  
   But this solution requires busy waiting  
   그러나 이 솔루션은 바쁜 대기를 요구함  
    ・This lock therefore called a spinlock  
    이 잠금은 그러므로 스핀록이라 불림  
   Solution to Critical-section Problem Using Locks  
     
   Mutex Lock Definitions  
   
5. Semaphore; 세마포 (Dijkstra)  
   Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities  
   Mutex locks보다 더 정교한 프로세스 작업 동기화 방법을 제공하는 동기화 도구  
   Semaphore S – integer variable  
   Can only be accessed via two indivisible (atomic) operations  
   2개의 분할할 수 없는(원자) 연산을 통해서만 액세스할 수 있음  
    ・wait() and signal()  
    -> (Originally called P() and V())  
   Definition of the wait() operation  
     
   Definition of the signal() operation  
   
   1. Semaphore Usage; 세마포 사용법
      1. Counting semaphore – integer value can range over an unrestricted  
         카운팅 세마포 – 정수 값의 범위는 제한되지 않음
      2. Binary semaphore – integer value can range only between 0 and 1  
         이진 세마포 – 정수 값은 0과 1 사이의 값만 가능
         1. Same as a mutex lock  
            mutex lock과 같다
      3. Consider P1 and P2 that require S1 to happen before S2  
         S1 작업이 S2보다 먼저 실행되길 바랄 때 프로세스 P1과 P2  
         Create a semaphore “synch” initialized to 0  
         synch를 0으로 초기화  
          signal()의 작업이 뭔지, wait()의 작업이 뭔지 알기
      4. Can implement a counting semaphore S as a binary semaphore
      5. Classical problems used to test newly/proposed synchronization schemes
         1. Bounded-Buffer Problem; Producer-Consumer
         2. Readers and Writers Problem
         3. Dining-Philosophers Problem
   2. Bounded-Buffer Problem (320P)  
        
        
        
      
   3. Reader-Writer Problem (321P)
      1. A data set is shared among a number of concurrent processes  
         데이터 셋은 동시에 발생하는 다수의 프로세스들 간에 공유됨
         1. Readers – only read the data set; they do not perform any updates  
            데이터베이스의 내용을 읽기만 함
         2. Writers – can both read and write  
            갱신(읽고 쓰기)을 할 수 있음
      2. Problem – allow multiple readers to read at the same time  
         문제 – 여러 reader가 동시에 읽을 수 있도록 허락해야 함
         1. Only one single writer can access the shared data at the same time  
            하나의 writer만이 공유 데이터에 액세스할 수 있음 (동시 X)
      3. Several variations of how readers and writers are considered – all involve some form of priorities  
         readers와 writers 문제의 여러 변형 – 모두 어떤 형태의 우선 순위를 포함한다
      4. Shared Data
         1. Data set
         2. Semaphore rw\_mutex initialized to 1
         3. Semaphore mutex initialized to 1
         4. Integer read\_count initialized to 0 – 현재 읽고 있는 reader 프로세스의 수  
            