Note: IPC - inter process communication - this applies to processes which can exchange data or synchronize execution flow - in certain scenarios you may also encounter threads and certain system level synchronization as well - in this context, focus is on the following:

- process to process synchronization via system space
- process to process synchronization via user space
- process to process data exchange via system space
- process to process data exchange via user-space
- related mechanisms and implementation details
- certain system calls may be required to use the above services, if needed by the developer !!!!

Note: most of what you study here and many process related concepts are applicable to threads/multithreading environments - however, there will be subtle changes as needed !!!

1. typically processes are independent and have independent address spaces !!! we are more concerned with data areas and still more concerned with read/write permissions to data areas - do not mix up this with code areas, when code is shared - meaning, process A cannot access process B's data area or information !!! here, the context is from system point of view - whenever we say that processes are independent , we mean from the system point of view, not user space point of view !!! later, when you do more productive work inside processes, you will understand that processes may not be independent, they may interact(more of user-space) - exchange data and synchronize with respect to each other !!! once again, not all processes need to interact with each other - some may have to - that is the discussion below !!!
   - from the system's perspective
     - micro level details and framework is in system space
     - you may see certain mechanisms, which have user space implementation details as well !!
   - from the developer's perspective
     - using certain system calls we can break or bend the microlevel framework in system space
     - this will provide us a set of skills !!!

2. if process A is interested in communicating with process B, there must be a mechanism supported by the underlying operating system(for instance, a shared kernel space) - typically operating systems can support several

mechanisms to achieve this - one such is a message queue -
one more is a pipe (unnamed or named) - one more is shared
memory - one more is sockets(used for networked I/O) - in addition,
operating system may also provide
synchronization mechanisms like semaphore or signals or
mutex or condition variable or any other such mechanism !!!
in this context, we will be looking more on semaphores -
signals were discussed, in earlier context !!!

- some mechanisms are data-exchange
- some mechanisms are locking/synchronization
- you may come across other synch. mechanisms - similar
  to semaphore, but implementation will be different ???
- in many real mechanisms, data exchange and synch/locking
  may go hand in hand - both mechanisms may work together !!!

3. let us assume process A wishes to pass data to process B,
   how can it be done ??

- one mechanism may be message passing
- another is using pipe mechanism
- depending upon context and operating system, you may
  have the choice of other mechanisms - accordingly, you can
  use them !!!

- why do we need the above mechanisms for this purpose ??
  - in this context, asynchronous mechanisms are assumed !!!
    - asynchronnous means independent operations on the
      mechanisms !!!
  - depending upon the platform, you may also encounter
    certain synchronous mechanisms - understand and use
    them accordingly !!!
  - since the processes do not see each others data
    spaces, system needs to store and forward data
    by receiving the data from one process and passing
    the data to another process !!!
    - how will process A send a message to the system ??
      a system call is needed
    - how will system maintain message(s) on behalf of
      processes ??
      a message queue data-structure is needed
    - how will process B receive a message from the system ??
      a system call is needed
    - what happens, if process B attempts to receive message

before process A sends a message ??
a waitqueue of pds is needed to be maintained
as part of message queue data structure

- what happens if process A sends a message before process B attempts to receive a message ??
  system has to maintain the message till process B attempts to receive a message - a message queue is maintained as part of message queue data structure !!
  - refer to chapter 6 of crowley - there is a section on message queue related system calls and their internal implementation !!!
- there will be a message queue objects array - this holds pointers
  to all message queue objects - there is one message queue object for each message queue mechanism instance, in the system !!!!no. of message queue instances/objects is as per the usage in the system !!!
- each message queue object maintains the following:
  - a list of message headers - each message header is used to manage one message stored in the message queue
  - messages are stored in the system buffers - messages can be of variable size !!1
  - size of the message is stored in the message header
  - normally, oldest message is at the head of the message queue and given to the receiver of message
  - in addition, a message queue object may also maintain a wait queue - normally, this wait queue will be empty - pd of a process is maintained in blocked state in the wq of a message queue object, if there is no message waiting in the message queue !!!
  - in certain systems, a message header may also

    maintain a message type - message type is a no. - a sending process and a receiving process can use a certain message type, if they agree !!!1
    - this type must be implemented in system calls and processes must understand and use the agreed type values between them !!! this is decided by the developer and implemented as needed !!
    - this is available in one implementation of message queues -
      in another implementation of message queues, priorites may be used - such message queues

are more common in RTOS - real time interfaces!!!

- each message queue object in the system will be storing an unique KEY value - this KEY uniquely identifies a message queue object and used by system call APIs !! due to practical reasons, apart from KEY, system also manages an id per message queue object !! in fact, this message queue id is more often used with system call APIs- KEY and id will be will clear when we discuss about system call APIs !!!
- internally, id may be mapped to the KEY value, if required - in any case, if you use the right system call APIs they will take care of this !!!
  - you may come across another set up in another OS or interface !!!

- before proceeding further, look into msg_server.c and msg_client.c examples - system call APIs and their parameters are explained !!!

- in most cases, system also is involved in synchronizing the activities of the processes that are involved in exchanging the data !! meaning, there will be an implicit synchronization between the processes !!!
  - if a process A is expecting data from another process B and initiating a receive message call from a message queue, the corresponding process A will be blocked in the wait queue of the corresponding message queue instance
  - it is the responsibility of the system to wake-up the blocked process A, receiving process, when the sending process B has sent a message to the message queue instance !!!more precisely, this synchronization is built into message sending system call is responsible !!! most of the intelligence is built into the operations of the IPC mechanisms !!!
    - synchronization typically involves co-ordination between processes and it may be achieved via operating system services explicitly or implicitly !!!
    - in explicit case, co-ordination is a deliberate act of the developer
    - in implicit case, communication/data exchange is the real intention and system adds co-ordination to ensure certain natural rules are satisfied - this can be convenient to the developer or may interfere with the developer's work !! you

must understand the working in each case !!!
- in certain cases, developer may influence the behaviour of system calls by using certain special flags/options !!! explore manual page of msgrcv() and msgsnd() for such special flags, if any !!!

- let us assume a message queue object is initially empty !!! meaning, no messages and wq is also empty!!
- let us assume process B invokes a receive system call API - what will happen ??
  - process B's contexts are saved
  - process B is added to wq of mq object - in blocked state
  - scheduler is invoked !!!
- let us assume that process A sends a message some time in the future !! what will happen ??
  - system call will add the message to the mq object
  - system call will scan the wq and if there is a process waiting, wake-up the process - meaning, change the state to ready state and and add pd to ready queue
- some time in the future, scheduler will schedule Process B - Process B will resume from the receive system call API - it will complete the receiving and return from system call API
- in this order, processes will continue using the message queue and exchanging messages, with the help of implicit synchronization implemented by the operating system !!
  - what is synchronization in this context ?? controlling execution of one process by another process via mq's wq mechanism and conditions !!
  - controlling a process from another
  - controlling cannot be direct, via some OS/lib mechanism - mechanisms are several !!!
  - will you accept, if there is no wq in the message queue ?? what will happen, if there is no wq, in the message queue ??

    - responsibility is more on the application and application may have to find better implementation techniques !!!
- comments for message queue implementation as per ch6 / crowley :

- receive message system call system routine works
  as below:
    - r9 is used to pass user-space buffer ptr
    - r10 is used to pass message queue id
    - check whether id is valid - message queue
      is valid !!
    - check whether there are messages in mq -
      if no messages, block the current process
      , add it to wq and end up calling switchprocess()
    - switch process saves system context and
      ends up calling scheduler !!
- send message system call system routine works as
  below:
    - r9 contains ptr to a buffer which hold a
      message
    - r10 contains mq id
    - id is validated - if it is invalid, error is
      returned !!!
    - a system space buffer is allocated for a message
    - using a special system API, system call system
      routine copies message from user-space buffer to
      system space - adds this system space buffer to
      mq's message queue
    - scans wq for waiting processes - if any process,
      wake-up the appropriate process !!!
    - return from send system call

- receive process will wake up and resume as below:
    - resume execution from receive system call
      system routine after switchprocess()
    - as given in the code, it will remove a
      a message from mq's message queue and
      copy it to user-space buffer passed by
      receive system call API
    - eventually, return to user-space and
      process resumes !!!

- is the above message queue mechanism, unidirectional or
  bidirectional ???
    -
- what is the requirement, if a message queue is needed
  to support bi-directional communication between
  processes ???
    - what is the fundamental problem that you
      visualize ?? there is a possibility for
      a sender to receive its own message !!!

- how to overcome such a problem ???
  - maintain one message queue for process A to B and another message queue for process B to A !!! this may be used, if a system's message queue implementation does not support message type fields !!!
  - instead, we can settle for a single message queue and do the following - use type A for messages from process A to B and type B for messages from process A to B and type B for messages from process B to A !!!
    - this needs system's support for implementation of message type fields and system call APIs !!!

  - refer to class diagram for better clarity !!!
- in Linux and other platforms, you may encounter other type of message queues - depending upon their implementation, rules may vary, but basic remain the same - study the documentation as required !!! read manual pages for more implementation details - refer to Linux or RTOS manual pages !!!
- a typical downside of this mechanism is that several data exchanges are to be done between user space and system space - which means, lot of copying between user -space buffers and system space buffers !!! if amount of data copied is large and frequently copied, may increase latencies !!! in a typical RTOS, such large data copying may be unacceptable - in certain RTOS environments it may still be acceptable,with certain constraints !!! finally, it depends on the developer what to use and what not to use !!!
- when you study and use socket mechanisms, you will realize that socket mechanism extends mq mechanism between machines with the help of network stack / rules !!!

4. let us assume that process A wishes to share certain data region/space(virtual pages) with process B, how can it do so ?? this involves sharing page-frames via page-table/pte manipulations !!!this sharing must be done in user-space, not in system space !!!

   - unlike the message queue case, in this scenario, we will be not be passing data via system-space, certain page-frames of process A will be shared with process B, using a mechanism known as shared memory mechanism !!!

- immediate advantage of this mechanism is that
  copying data to system space and back is avoided-
  several system call APIs are avoided - so, faster
  and more efficient !!!this mechanism reduces no of
  system call APIs during data exchange !!! such
  ligter and faster mechansims are preferred, if
  performance is key !!!

- how is this achieved ? meaning, what is the set-up ??
  - certain ptes of process A and certain ptes of process
    B will be forced to point to the same set of page-frames
    by the system with the help of a set of system
    calls !!!
  - what is the difference between these shared page frames
    and the shared page frames associated with system space ???
    - u/s bit will be different - here it is set to 1
      (for system space page frames(shared/private) it
      it is set to 0)
    - shared user-space page frames can be accessed by
      application code and shared system space page frames
      can be accessed by system space code only !!!
    - shared user-space page frames can be accessed
      without system call APIs - however, shared system
      space page frames need system call APIs !!
    - user space shared memory can be accessed in user-space
      using simple pointers/virtual addresses/logical
      addresses, not system call APIs !!!
  - in short, a set of virtual pages of a process and a set
    of virtual pages of second process are set up such
    that they map to the same set of page-frames with
    u/s bit set to 1 - in addition, if process A and process
    B wish to exchange data, it is just a matter of writting
    to this shared set of virtual pages and reading from
    this shared set of virtual pages !!! shared virtual pages
    /associated page frames can be accessed using pointers -
    no need for system call APIs

- do you expect any problem due to concurrent /parallel
  executions of processes sharing certain system space
  IPC mechanism ?? meaning, processes share page frames
  of system space for data exchange via system space
  IPC mechanisms !!! if so, how these problems are
  taken care by the system space ???
    - possible race conditions and subsequent
      inconsistency in data exchange !!! there are

several scenarios - most of these race conditions occur due to user space preemptions/system space preemptions and multiprocessor, parallel executions - for all cases, the best approach is locking - some form of locking is used by the system in appropriate contexts !!!

- typical locks are semaphores, mutexes and spinlocks - they are used along with other mechanisms like preemption disabling and interrupt disabling !!!
- in most cases, you may not be able to escape locking - however, in some cases, with meticuluous implementations, you may escape locking !!!
- what do you think is the downside of locking ??
  - multitasking is minimized and one or more processes may be blocked and cannot progress - which means, latency and performance are affected!!!
- so such race-conditions/problems are common in system space and there are appropriate locking mechanisms in place by the system space developers - if you write system space code, you may need to implement appropriate locks as well !!!

note : refer to chapter 6 of crowley - race condition related diagrams - it is more relevant to multiprocessor scenarios !!!

- do you expect any problem due to concurrent execution of process A and process B sharing certain virtual pages in user-space - meaning, do we expect any problems ,when process A and process B are sharing memory via shared user-space virtual pages !!! yes !!!
  - due to multitasking, user-space preemption and multiprocessing/ parallel executions !!!

- the typical problem encountered, in the above case is known as race condition - race condition will lead to inconsistency in data and lead to inconsistent results - this is a very basic computing problem and operating system/hw provide mechanisms to overcome such problems - refer to read/write diagram in ch6 of crowley - this diagram explains race-conditions and

related inconsistencies - along with this diagram
read what is given in this notes - explanation is
given here !!!

- this diagram is more suitable for multiprocessor systems - so, be careful !!!
- this diagram may be interpreted for uniprocessor systems as well as multiprocessor systems - use it as per the context !!!
- we are in uniprocessor context and assuming that a process is accessing a shared page variable and there is always read, modify and update - there may be other cases - what we are looking at is the classical race condition in computing !!!

- refer to examples - prod_test.c and cons_test.c - in these examples, a shared memory area is manipulated and a specific variable is used as shared counter - prod_test.c increments the shared variable and cons_test.c decrements shared variable - if concurrency and preemption are applied, there are possibilities for race conditions and result in inconsistent data - which will result in inconsistent computing !!
- refer to race.c, which is specifically coded to generate race conditions, in uniprocessor and multiprocessor !!
- in these examples, if process A is reading and updating a shared variable, i, there are possibilities of race conditions, if another process B also reads and updates i - this is true in the case of uniprocessor systems with preemption and multiprocessor systems with or without preemption !!!

- one such case is described below:
  - let us assume process A reads value of i , internally updates the value of i in a temporary variable/register - before process A updates i, system may preempt process A and schedule process B - before rescheduling, execution context of process A is saved - process B will read pre-updated(old data) of i and also update it with a new value - some time

in the future, process A will be rescheduled and
it will update its new value saved in its context
into i - the outcome is that process B's change
is lost - only process A's change is updated -
- what is more important is that the final
result is inconsistent -
this is known as inconsistency in shared data
due to race conditions - this may occur once
in 10years or once in a second - still, unacceptable
for real world problems !!!
- many such problems and scenarios exist in
real world and operating system people analyse
and solve such problems - provide solutions
and mechanisms for such problems !!!
- for a typical developer, mechanisms already
exist, but finding the problems, sections of code
having problems and solving them are critical !!!

- a section of code of process A and a section of code
of process B that are involved in a race-condition
are said to be related critical sections !!!

- it is the responsibility of processes/applications
to prevent race-conditions between related critical
sections by using mechanisms like semaphores /mutexes !!!
  - critical sections may be protected by appropriate
    locking mechanism !!!

- you may encounter semaphores/mutexes directly or
indirectly in your future programming frame-works -
do not get attached to names and definitions - focus
on the functionality and purpose of such mechanisms !!

- let us understand race-condition using a classical
example of a single shared variable - whatever you
learn and understand in this context can be easily
extended to other complex contexts !!

- using the same single shared variable context, we
will also understand how to use a semaphore mechanism
to overcome race-conditions !!!

- of course, before doing so we may have to understand
the fundamental working / implementation of a semphore
by the operating system !!!

- once we do the above, we will be using semaphores in the context of shared memory between processes and enable two or more processes to share data consistently without race-conditions and with appropriate synchronization !!

- if 2 or more processes are sharing a shared variable(object/ data structure) and updating the variable such that there will be inconsistency in the results of the variable(objects/data structures), if a process is preempted while it is executing its critical section and the other process is scheduled to execute its critical section - meaning, updating the shared variable(objects/data structures) in the other process - such a problem is known as a race condition due to concurrency and preemption !!!

- in the above case, there can be problem if 2 or more processes are executing their related critical sections simultaneously, in a multiprocessor system !!! such a problem is also a race condition - this is due to multiprocessing and parallel scheduling of processes !!!

5. an example of a race condition !!!

- process A and process B share a variable i
- process A may access i using i++ operation
- process B may access i using i-- operation
- this is just an example - processes may access larger objects stored in shared memory or access data-structures stored in shared - memory !!
- in all the above cases, problem is the same - race-conditions will lead to inconsistent results !!!

- in the case of i++ / i-- access in 2 processes, if there is a preemption in the middle of i++ or i--, the results will be inconsistent - this is due to the fact that multiple machine instructions are used to implement i++ / i-- and the preemptive nature of operating system - this is a very basic problem and code like i++ / i-- on shared variable by multiple processes are known as related critical sections

- although i++/i-- are very primitive examples, in the real-world, we will be dealing with larger objects and data-structures - critical sections are longer and more troublesome !!!

- critical sections are typically said to non-atomic - meaning, they are not atomic - atomic in this context means indivisible - indivisible in this context means, cannot be preempted by the system and instructions from other processes cannot be interleaved !!!
  - instructions of a critical section and instructions of another related critical section must not be interleaved !!! if there is interleaving of instructions, we say that a critical section is non atomic !!
  - if related critical sections and their instructions are not interleaved, we say that they are atomic !!!
    - this atomicity must be true in uniprocessor and multiprocessor cases - refer to ch6 of crowley !!!

6. life of a semaphore - meaning, its life cycle and its usage :

note: semaphore operations are assumed to be atomic, with the help of operating system and hw !!!

- semaphore is a special system variable(a counter) managed by the operating system, specially - like any other shared variable, this will also suffer from race conditions and other issues discussed, above - system manages this special variable using certain sw and hw techniques - this enables this variable to behave as a super variable - see the discussion below !!!

- a semaphore variable is maintained as part of a semaphore object !!!normally, only one semaphore is maintained in a semaphore object!!!
  - a semaphore object is mostly maintained in system space
  - in many cases, semaphore object may be maintained partially in user-space and partially in system-space !!
  - if required, you may understand the underlying implementation !!
- for the discussion below, we are looking at a conventional semaphore - meaning, it is entirely maintained in system space !!

- a semaphore variable is constrained by certain rules - it can have value between 0 and a +ve no. decided by the system and the developer - typically semaphore

value cannot drop below 0 - cannot be -ve -
typically semaphore value is between 0 and 1, in
many cases - although, it can also be +ve(>=1), in many cases !!!

- max value of a semaphore is decided by operating system
- current max value for a given application is decided by developer
- binary semaphore has value between 0 and 1
- counting semaphore has value between 0 and a +ve (>1, decided by developer)
- in either case, semaphore variable is fundamentally a counter !!! how we use it depends on how we initialize - once again , application and developer decide how to use a semaphore !! however, rules are the same for binary as well as counting semaphores !!!

- system normally supports certain operations on a semaphore
  - creation, initialization, decrement , increment and destruction (a semaphore object/semaphore is a logical resource)
  - creation is supported by a system call - a semaphore object is created using this system call
  - initialization of a semaphore enables to initialize the semaphore value as per developer's requirement - it can be 0 , 1 or a +ve no. - this is done using another system call API !!!subject to the rules of semaphore ,operating system and application's requirement !!!
  - let us assume that initial value of semaphore is 1 (depending upon application's requirement, it can be different - we see more of this during examples / assignments)
    - there is no such default value for a semaphore !!
    - during assignments, try to set / initialize semaphores with large +ve values and -ve values - find the error using errno !!
  - decrement operation - decrement operation follows the rules below:(another system call API)
    - if the semaphore value is +ve, just decrement the value of semaphore by 1 and return success !!!
    - if the semaphore value is 0, do not decrement, change the state of the process to blocked and add the process descriptor to the wait-queue of the semaphore object - this leads to blocking the process that has attempted a decrement operation on the semaphore !!!in the case blocking, scheduler is invoked !!!
    - a blocked process, in the wq of a semaphore may be

woken-up by another process that executes increment operation on the respective semaphore
    - increment operation and decrement operation are connected , when respective processes use the operations !!!
- increment operation - increment operation follows the rules below:(another system call API)
    - if the semaphore's wq is empty, just increment the value of the semaphore by 1 and return success !!!
    - if the semaphore's wq is non-empty, do not increment the semaphore's value, but wake-up a process that may be blocked in the wq of the semaphore and return success !!!
    - based on the above decrement operation and this increment operation, we can understand that synchronization is managed by the semaphore !!!
        - processes may co-ordinate their execution with the help of a semaphore to achieve certain locking/counting/some other operation !!!
        - synchronization is the basic requirement for which semaphores are used - of course, based on the application's requirement, semaphore may take up additional responsibilies !!!
        - semaphores are used for explicit synchronization , in applications/system code !!!

- destruction operation - destruction operation simply frees the semaphore object !!! the semaphore object and corresponding semaphore are no longer accessible !!!
    - in addition, during freeing of a semaphore/semaphore object, currently blocked processes will be forcibly woken-up - however, this is an error condition, not a normal wakeup - it is the headache of the developer to check for error conditions and take the next action !!!
    - during implementation, there can be subtle changes in the behaviour - you must read the respective documentation !!! in these cases, apart from destruction, system may take further actions - refer to manual pages of different implementations !!!

- in reality, a semaphore is maintained as part of a semaphore object array / table !!!
- each semaphore object contains a semaphore variable, certain credentials and a wait queue for maintaining

blocked process descriptors that attempted decrement operation on this semaphore !! in addition, a lock variable may be maintained in a sem object !! this lock variable works with hw mechanisms and is responsible for consistent behaviour of semphore/semaphore operations !!

7. let us assume that we are using a semaphore/semaphore object to implement critical section of i++/i-- in 2 processes - meaning, a semaphore is used to provide atomicity to the critical sections such that when a process A executing in the critical section is preempted and process B will be blocked, if it attempts to enter its critical section !!! the same applies vice-versa, if process B is preempted in its critical section !!!

- in fact, this solution using semaphores is known as mutual exclusion technique !!
- in the above case, the semaphore is already created and initial value of the semaphore is set to 1 and we are operating on the semaphore !!

- let us assume process A(P1) is scheduled first - P1 will attempt to decrement the semaphore value and semaphore will become 0 - in addition, P1 will continue executing its critical section !!!

- let us assume P1 is preempted in the middle of its critical section - P1 will be preempted and P2 may be scheduled - P2 will attempt to decrement the semaphore value and P2 will be blocked in the wait queue of the semaphore - this ensures that P2 does not enter into its critical section, when P1 is in the middle of its critical section !!!this ensures that P1's critical section is atomic due to the use of a semaphore !! meaning, P1's critical section is atomic with respect to P2's critical section - no more !!!

- sometime in the future, P1 will be rescheduled and it will complete its critical section and increment the semaphore - since P2 is blocked in the wq of the semaphore, when P1 increments the semaphore value, semaphore value is not incremented, but P2 will be woken-up - what is the current value of the semaphore after the increment operation and wake up of the P2 process , in this context ??? before P2 is rescheduled by the scheduler ??? during this increment operation, semaphore value is not incremented and remains 0 !!!

- when P2 is woken-up and rescheduled in the future, P2 will resume its execution from from decrement system call,complete the system call execution, return from system call execution and enter its critical section - the semaphore value is still maintained as 0, due to a tricky mechanism !!! described above !!! once the critical section of P2 is completed, it will increment the semaphore - semaphore value will change from 0 to 1

- in the above sequence of execution , semaphore/semaphore operations ensure that a set of instructions in a set of critical sections are executed atomically with respect to the other section !!!! meaning, instructions of a related critical section and instructions from another related critical section are not interleaved !!! this is achieved by using semaphores as described in the above section and in the class diagram !!!
    - semaphores ensure critical sections are executed atomically with respect to each other
    - due to this race conditions are prevented
    - if race conditions are prevented,inconsistencies of shared memory access are prevented !!! in short, this is what we have achieved using locks !!!

- to understand the working of semaphores, do the following:
    - read the above notes
    - refer to chapter 8 of crowley - there is a good discussion on semaphores - system call system routines of semaphore decrement and increment operations are illustrated !!!
    - refer to ch6 of crowley - there is a good discussion on blocking and waking up processes - the exact saving of context and restoration of context is discussed - also, you will understand how a process resumes its execution after it blocks inside a system call and it is woken up - so a combination of ch8 and ch6 will give you more clarity
    - in addition, also refer to class room diagram, which explains system space working of semaphores operations !!!
    - in addition, we will use prod_test.c, prod_1_1.c , cons_test.c and cons_1_1.c for a more practical understanding and usage of semaphores !!!

Note: for semaphores or similar locks/synchronization mechanisms
to work properly, there
is an assumption - assumption is that the operations
/system calls used for the semaphore/lock operations
are atomic in nature - which means, if a process is
currently executing a semaphore operation's system call
, another process must not be allowed to enter/execute
any other semaphore operation's system call on the
same semaphore/semaphore object that is used by the former
process !!!

meaning, they do not face any
race conditions due to implementations !! the below
discussion explains how such atomicity is achieved
in operating systems - semaphore operations are one set
of examples - there are many such os implementations that
are useful in providing user space services !!!1

8. system call system routines implementing semaphore
operations may face race-conditions, if the system
supports system-space preemption and/or system supports
multi-processing - in these cases, semaphore value/
semaphore object will encounter inconsistency - such
in consistency cannot be accepted as this will
lead to inconsistencies in applications !!!
- are you able to visualize the above problems
- can you vis. problems in semaphore operations
, in system space along with preemption and uniprocessor conditions ??
- if there are race conditions in the semaphore operations,
how can they be fixed ?? any comments !!!

- we may disable hw interrupts before a semaphore operation
, in system space and enable hw interrupts after a semaphore
operation, in system space - typically,we cannot do such
activities in user-space !!!
this will decrease the responsiveness of
the system - normally, a system's responsiveness is tightly
clubbed with I/O responsiveness !!! in addition,
as we will see it below,
this solution may not work on multiprocessor systems !!!this
may work in uniprocessor systems only !!!
- in the uniprocessor system, disabling preemption is a
better solution than disabling hw interrupts - meaning,

disabling preemption is more efficient in terms of
I/O responsiveness than disabling interrupts !!!
- what is the meaning of disabling preemption,
  in system space ???
  - a special variable(not a semaphore) is
    set such that scheduler does not reschedule
    in the system space, when this flag is set !!!
  - the scheduler will resume it normal working
    , when this flag is reset !!!
  - we will see more of this during system space
    coding !!!

- disabling preemption is not allowed in user-space - there
  is no such service !!!
- there are special hw machine instructions that may be used
  to implement hw supported locks and these locks can in turn
  be used to implement atomic operations for synchronization
  mechanisms like semaphore !!! there are legacy machine instructions
  and there are modern machine instructions - legacy machine
  instructions are said inefficient compared to recent
  implementations - read the related ARM manuals for legacy
  and recent implementations !!! recent implementations are
  more efficient in multiprocessor systems !!!

- let us assume that we are using a legacy machine instruction
  - say, swap instruction known as swp, in ARM architecture !!!
    - this instruction is said to be an exchange instruction
      that can exchange data in a register with data in
      a memory location, atomically at the hw level !!!
      - during the machine instruction, following
        is true as per the technical info. available:
        - hw interrupts are disabled, in local processor !!
        - no other process/processor can access
          memory (locations) !!!
        - this is what is the speciality of the swp
          machine instruction and internally, this
          is how hw level atomicity is achieved !!!
  - with the help of swp instruction, we can implement
    a hw supported lock, which will atomically attempt
    to check the lock variable's(part of a semaphore
    object) state and set to busy -
    if the return value of the atomic operation reflects
    that the lock variable's state was busy, our lock
    implementation will retry and continue to spin,
    until the previous lock variable's state is not
    busy !!! such a lock implementation is known as

spin lock !!!(this is as per OS conventions -
hw conventions may differ - however,
OS conventions are final !!!)

- another efficient solution may be provided using a lock variable and certain atomic machine instructions !!! h/w normally provides such atomic machine instructions !!!
  - to understand this, we will be using a lock variable inside a semaphore object !!!
  - operations on this lock variable are done using atomic machine instructions - one such is exchange machine instruction
  - it can read the previous value of the lock variable and set the current value of the lock variable to 1, atomically !!! during this machine instruction, no hw interrupt will be serviced - meaning, this machine instruction cannot be interrupted !!!
  - r1 is always set to 1, before invoking lock(sema->lock)
  - following pseudo code is used to implement

```
Lock(sema->lock) (ch8 of crowley )
{

  //while(test_and_set(sema->lock))
  while(swp(r1,sema->lock))
    do nothing
  endwhile
}
```

  - unLock(sema->lock)

```
{
   sema->lock = 0;
}
```

  - in the above cases, if the lock variable's value is 0 means lock is available and 1 means, lock is busy

  - in the above cases, if the lock is available, it is locked and Lock() code just returns !!!
  - in the above cases, if the lock is not available, the Lock() code spins until lock variable is free !!!
  - such a lock variable and its operations are together are known as spinlock !!!! these locks are spinning locks and not blocking locks - semaphore

mechanisms are known as blocking locks !!!
- in the uniprocessor context, is the usage of spinlock variable to protect semaphore operations effective ??? meaning, have we eliminated the race conditions in the semaphore operations ???
  - analyse using diagrams and preemption in system space semaphore operations - you will be able to visualize the problems !!!
  - this mechanism has eliminated the race condition , but wastes cpu cycles in certain cases and in certain cases, may lead to a type of dead lock !!!
    - analyse for timeslicing/time sharing cases
    - analyse for priority based scheduling cases !!!
- such a solution is unacceptable and a slightly modified solution is provided !!! before the lock is acquired, preemption is disabled in the system space - if this preemption is disabled in the system space, no preemption can occur - meaning, scheduler will never reschedule another process - in this context, there should not be any such wastage of cpu cycles/no dead-locks !!!
- preemption disabling means, no rescheduling will be allowed, but hw interrupts are allowed - meaning, hw interrupts will not end up recheduling - this is achieved by atomically setting a variable, which controls whether preemption will be allowed or not
  - such an atomic operation on the special variable must be completed before acquiring the spin lock controlling the respective semaphore, inside the semaphore operation code !!!

- it is a combination of preemption disabling at the scheduler level and also acquiring a spinlock - this combination works and works efficiently !!!
- the above solution using preemption disabling and spinlock works for uniprocessor - does the same solution work for multiprocessor systems - meaning, process1 and process2 may be scheduled on different processors as per the systems' load balancing on a MP system ???

- in uniprocessor
  - can you visualize this problem !!!

yes !!! there is a problem in uniprocessor
system also !!!
- how to overcome such a problem ???
  - see the solution above

- in multiprocessor
  - can you visualize this problem !!!
    yes !!! there is parallel execution and
    that leads to race-condition in system space !!!
  - how to overcome such a problem ???
    - the above solution also works for
      multiprocessor scenarios, but spinning of
      a process in system space, while another
      process is holding the spinlock cannot be
      avoided - however, if system space code
      is written properly, such spinning can
      be minimized !!!

- in both the above cases, following is the real - problem:
  - read - modify - write must be atomic
  - in our discussion, read - modify -write of one
    related critical section and read-modify-write
    of another related critical section must not
    be interleaved !!!
  - it does not really matter, we are in uniprocessor
    or multiprocessor !!!
  - however, uniprocessor scenario and multiprocessor
    have different race conditions and scenarios -
    they need different set of approaches - let us
    see how they work ??

in order to solve these race conditions, system supports
a special lock known as spinlock - it works as mentioned below:

- it is a special variable in the system - space - it
  can hold one of the two values - 0 or 1 - 0 means lock
  is available and 1 means lock is not available - this is
  a convention that is mostly followed

- spinlock()(lock()) operation will attempt to atomically
  set the value to 1 and read the previous value(this is
  a read- modify- write case for a spinlock() operation) !!!
  if the previous value was 0, lock is said to be obtained
  and spinlock() will return !!! if the spinlock()(lock())

operation finds
that the previous value of the lock was 1, spinlock() will continue spinning / busy waiting for the lock's value to be set to 0 - this is the reason why such a lock is given the name spinlock!!!!

- spinunlock()(unlock()) operation will just set the value to 0 - unlocked state - there is no great mechanism involved in this !!!

- a lock of this type does not have a waitqueue - it does not block the process, if the lock is not available - instead, it allows the process to busy-wait or spin !!!

- spinlocks are special locks used to implement semaphores and other ipc mechanisms !!!

- spinlocks use special h/w, atomic instructions to implement atomic locking, which helps spinlock implementation - in turn, semaphores and other mechanisms benefit from spinlocks

- one major shortcoming of a spinlock is it does not support, waitqueues - meaning, blocking !!!

- spinlocks are useful if the critical sections locked by spinlocks that are short !!! meaning, for longer critical sections, spinlocks tend to be inefficient when a spinlock is not available and a process is attempting to lock the spinlock !!! this is one of the major reasons why spinlock is not so popular, in user-space - it is still popular in system space - in system space, critical sections can be better controlled and there are certain scenarios, where semaphores or other locks cannot be used !!!
  - in specialized systems, user space developers may also be allowed to use spinlocks - in these systems,user-space developers are also highly skilled !!!!

9. although semaphore is typically treated as a lock and books describe semaphores using critical sections, semaphores are not just locks - they can be used for counting resources and for typical synchronization - both counting resources and synchronization can be understood using practical scenarios - synchronization may be explained theoretically, in this context :
   - synchronization is controlling execution of a process by another process via an operating system mechanism - one such popular synchronization mechanism is semaphore - in many mechanisms, synchronization is implicit !!!

- if a process attempts to decrement a semaphore whose current value is 0, corresponding process will be added to the wq of the semaphore and state of the process is changed to blocked !!!
- if another process executes increment on the same semaphore due an event, increment system call will wake-up the blocked process
- the above is a clear example for synchronization, where a process is controlled by another process via a semaphore !!
- in the above example, there is no critical section and semaphore is not used as a lock
- even if a semaphore is used as a lock, still it uses synchronization to control a process by another process during locking / unlocking operations !!!

- can the above implementation of semaphore operations work consistently in uniprocessor and multiprocessor environments ???
  - preemption is not disabled in user-space - whatever is discussed is only for system space issues - whenever a process resumes in user-space, preemption is immediately restored to original state !!! preemption is always enabled in user-space and hw interrupts are always enabled in user-space !!!
  - in uniprocessor, using the spinlock along with disabling preemption is redundant - find out how the real implementations are ?? for the class room, this conclusion is ok !!!
  - in multiprocessor, using the spinlock along with preemption disabling is not redundant - it is a must !!! why so ???
    - in this case, process1 and process2 may execute system space critical sections of semaphore operations, simultaneously, on different processors !!
    - what happens, if both processes access the spinlock locking simultaneously ???
      - refer to ch6 of crowley - there is a diagram on read-modify-write for all cases !!!

  - based on the above discussions, we can trust that the semaphore operations are atomic and they can be used in our critical section code - similarly, os implements many locks and operations on these locks are said to be atomic-

we must trust and use them - no need to suspect
their behaviour !!!

- however, you will still face problems and this
is common in system space locks and their implementations!!1

Note: in every process, certain virtual pages/virtual addresses are reserved for system space usage - meaning, system space code, system space data, system space dynamic memory and so on - in addition, in every process, such reserved virtual pages are managed by system and mapped to the same set of page frames - these page frames hold the system space code/data/dynamic memory and so on - since, such reserved pages in every process are mapped to the same set of system's page frames, such virtual pages of every process are said to be shared with respect to corresponding reserved virtual pages in other processes !!! these are also known as shared virtual pages, in system space !!

10. shared memory related system call APIs and their functionalities:

- how is shared virtual pages / shared memory regions created between 2 or more processes, in user-space ??? meaning, what is the underlying setup that is needed to accomplish this ??? refer to point no. 4, above and then, resume with the discussion below :

- shmid = shmget(param1,param2,param3);
  - param1 is the key value - which is unique no. identifying a particular shared memory object in the system - this is chosen by the developer !!!
  - param2 is the size of the shared memory region managed by the shared memory object, in the context !! size of the shared memory region must be a multiple of page size !!!
    - if we do not provide a size that is not a multiple of page size, system will round it up to next nearest multiple !!! resizing is disallowed !!
  - param3 provides a set of flags, which we will understand as needed - let us use default set of flag values - later we will tune it as needed !!!
  - if shmget() is successful, it will create a shared memory

object in the system-space and return the corresponding id for the shared memory object - further system call APIs are expected to use this id as their parameter to access this shared memory object

- when shmget is invoked, a shared memory object may be created, if it does not exist - if the shared memory object does exit already, system will just return its corresponding id - when you use shmget(), be aware of these rules !!!
- still refer to man 2 shmget() for more details !!!
- shared memory object also maintains an array of shared page frame base addresses - this is not same as page tables - this array only contains base addreses of page frames that are used for the shared memory region managed by this shared memory object !!! shared virtual pages mapping a shared memory region of one or more processes will be using these shared page frames for their ptes !!!
    - how this is achieved is discussed below !!!
- the no. of elements in the array is dependent on the size of the shared memory region - shared memory region size is always a multiple of page-frame size !!!
- when shmget() is invoked and shared memory object is newly created, the elements of the shared page frames array are initialized to 0 - meaning, no shared page frames are allocated for a shared memory region, when shmget() creates shared memory object - this is based on demand paging principle of virtual memory !!
    - meaning, page frames allocation for shared memory regions are deferred !!!

- can we access the shared memory area managed by the shared memory object from a process ?? meaning, from the process that has created the shared memory object !!!
- if a process is interested in using a shared memory region associated with a shared memory object, the process must attach//associate itself to the shared memory object using shmat() system call API - shmat() does the following:
    - creates a new VAD for the current process, sets special shared flag in in the VAD,shmid of the associated shared memory object is also stored into the new VAD !!!
    - in addition, corresponding page table entires

may be created and initialized to 0 !!!
- shmat(param1,param2,param3) - param1 is the
  id of the shared memory object - param2 may be used
  to tell the system the starting virtual address
  to be used in the new VAD - if 0 is mentioned
  as param2, system will assign a new set of
  virtual addresses to be used with this new VAD -
  0 is a preferred option - param3 is to pass flags-
  normally, flags are not needed - 0 is commonly used !!!
  refer to man page of shmat() to understand more
  on flags!!!
- if shmat() is successful, in addition to what was
  described earlier, it will return the first virtual
  address associated with the new VAD - in short,
  these virtual addresses starting from the returned
  virtual address may be used to access shared memory
  region and associated page frames !!!
- corresponding to the shared memory VAD, certain
  secondary page tables are created for this
  process and the secondary ptes are set to invalid !!!
- each such shared memory related VAD is special -
  it will have a special shared flag set and also
  the id of the shared memory object is stored in it !!!
- let us assume all the above and most of what is
  discusssed below are in P1 (we see P2 after this !!!)
- eventually, a process associated with a shared memory
  region will attempt to access the shared memory region
  via new set of virtual addresses - when a process
  attempts to access a new virtual address corresponding
  to a virtual page of the shared memory region, a page
  fault exception will be generated - page fault
  exception handler will be invoked - as discussed during
  virtual memory management, most steps are the same -
  however, there are changes - we will discuss the changes
  only
- after the faulting virtual address is verified with
  the available VADs of a process, system does the following:
  - checks whether the VAD has the shared flag set
    (in the case of normal VADs, shared flag is not set !!)
  - if true(shared memory case only), uses the
    corresponding shared memory object id stored in
    the VAD to access the associated shared memory
    object(VAD and shared memory object are connected)
  - after accessing the shared memory object, checks
    the appropriate entry in the shared page-frames

array for this particular shared virtual page -
let us assume the shared memory region has
3 virtual pages and 3 elements in the shared
page frames array !!! how are they connected !!!
- shared virtual page0 is mapped to page frame base
  address array[0] , shared virtual page 1 is mapped
  to page frame base address array[1] and so on !!!
if the mapped array element(can be 0th element,
1st element or ith element or n-1th element)
contains 0, allocates a new shared page frame,
stores its base address in
the particular entry of the shared page frames
array, uses the page frame base address to
set up the current process's shared virtual page's
pte entry - current process is restarted to
resume from faulting virtual address !!
  - in the above case, base address of
    a shared pageframe may be non-zero,
    in certain cases , if required !!!
  - in the case of a normal page fault,
    a new page frame will be allocated
    immediately - in the shared memory virtual address
    page fault case, a new page
    frame must be allocated via shared memory
    object mechanism and its rules !!!

- shared memory objects and shared memory regions
  are useless, if only one process is associated with
  them - meaning, two or more processes must be
  associated with a shared memory object/shared memory
  region for this mechanism to be useful !!!
- let us assume that another process involved
  in sharing a shared memory region with the
  earlier process - it has to do the following
  actions - the discussion below is about P2 :
  - use shmget() with the same KEY value
    as first process such that the second
    process can access the same shared memory object
  - in addition, second process must invoke
    shmat() to associate itself with the
    shared memory object/shared memory region !!!
    - when the second process attaches itself
      to the shared memory region via shmat()
      system call API, it will be provided
      its own shared memory VAD and connection

to the shared memory object - this new VAD
is also treated specially !!!!it has its
own ptes in a secondary page table of Process p2-
these ptes are initialized to 0, as per
virtual memory convention !!!

- after the above steps, if the second process
attempts to access a shared virtual page
allocated to it, following actions will be
taken:
  - a page fault exception will be generated
    for this second process
  - after the faulting virtual address is verified
    with the available VADs of a process,
    system does the following:
    - checks whether the VAD has the shared flag set
    - if true(shared memory case only), uses the
      corresponding shared memory object id stored in
      the VAD to access the associated shared memory
      object
    - after accessing the shared memory object, checks
      the appropriate entry in the shared page-frames
      array for this particular shared virtual page -
      if the array element is 0, allocates a new
      shared page frame, stores its base address in
      the particular entry of the shared page frames
      array, uses the page frame base address to
      set up the current process's virtual page's
      pte entry - current process is restarted to
      resume from faulting virtual address !!
      - in our current discussion, there is a
      high possibility that corresponding page
      frame base address will be non-zero !!

- in the above cases, if a process using a shared page/page
  frame has encountered a page fault and a new page frame
  is allocated, that shared page frame's base address
  is maintained in the shared memory object's shared page
  frames base address array - if another process also
  is attached to the same shared memory object and also
  attempts to access the same shared page frame via
  its own shared virtual page, the stored base address
  of the shared page frame will provided to the second
  process as well - this is the principle of shared
  memory region via shared memory object !!!
- the above actions will be repeated for all the shared
  virtual pages and corresponding shared page frames !!1

- this is how, the system shares page frames between interested processes via specially setup VADs and shared memory objects !!!

11. unix/linux sempahore system call APIs and their working :
   - a semaphore object and one or more semaphores may be created using semget()
   - ret = semget(KEY,param2,param3) - param1 is KEY - as mentioned in shared memory object, a semaphore object is uniquely identified using a KEY value - param2 decides whether a single semaphore will be managed by a semaphore object or multiple semaphores will be managed by a semaphore object !!! param 2 can be 1, in which case a single semaphore is managed by a semaphore object - param2 can be >1, in which case several semaphores can be managed using a single semaphore object !!!
     param3 is similar to what we discussed for a shared memory object - meaning, it passes required flags - a default set of flags !!!
   - normally, a semaphore object will contain a single semaphore - a unix/linux semaphore object may contain a single semaphore or multiple semaphores !!! it is the requirement of the developer that decides whether a single semaphore is needed or multiple semaphores are needed !!!
   - if a semget() is successful in creating a new semaphore object and associated semaphore(s), it will return appropriate semaphore id - this semaphore id may be used in further system call APIs !!!

   - after a semaphore object with semaphore(s) is created, semaphores must be initialized - to initialize a semaphore in a semaphore object, semctl() is used - semctl(param1,param2,param3,param4) - param1 is the id of the semaphore object - param2 is the index of the semaphore in the semaphore array of the semaphore object- param3 is the command to the semctl() - semctl() has many functionalities - initialization is one of its functionalities- SETVAL command is used to initialize a semaphore using semctl() - param4 is an union which has several fields - a field in the union is used depending upon the param3 - in our case, SETVAL uses val field of the union - val field of the union decides the initial value of the semaphore that will be initialized by semctl()

- once a semaphore is initialized as per our requirements, we can operate on the semaphore using semop() system call API - semop() system call API can be used for decrement operation as well as increment operation !!!

- semop(param1,param2,param3) - param1 is the id of the semaphore - param2 is address of an array - elements of this array are of type struct sembuf { } - param2 can point to an array, which contains one or more struct sembuf { } elements !! param3 indicates the no. of elements in the array pointed to by param2 that must be used by semop - meaning, param3 may be less than or equal to the total no of elements in the array - refer to examples -
  based on parameters passed to semop(), decrement operation or increment operation may be done on appropriate semaphore !!!

- struct sembuf { } elements are as below :

    - struct sembuf sb[3];

    - sb[0].sem_num is filled with the index of a semaphore in the semaphore object
    - sb[0].sem_op is filled with the appropriate operation - +1 for increment operation and -1 for decrement operation
    - sb[0].sem_flg is the flags field and typically set to 0

- for an illustration, let us a few examples below:

```
- sb[0].sem_num = 0;  //index is set to 0
  sb[0].sem_op  = +1; //increment operation
  sb[0].sem_flg = 0;  //just set to 0
  semop(id1,sb, 1); //what does this semop() do ??
  //the current value of semaphore with index == 0 is
  //incremented !!!
- sb[0].sem_num = 0;
  sb[0].sem_op  = -1;
  sb[0].sem_flg = 0;

  sb[1].sem_num = 1;
  sb[1].sem_op  = -1;
  sb[1].sem_flg = 0;
  semop(id1,sb, 2);
```

- you may operate on one semaphore at a time, in a semop() API and invoke semop() several times for each operation !!!
- or, you may operate on several semaphores at a time, in a single semop(), without invoking semop() several times !!!
- apart from programming aspects, can you mention some advantage when we use several semaphore operations in one semop() ??
  - when you do several operations in one semop(), these operations are handled atomically - meaning, if both operations can be completed, they will be or if either operation cannot be completed, both will not be completed!!!
    - meaning, if one of the operations is not successful, no operation is completed and current process is blocked !!
    - meaning, both operations will be successful or none will be successful !!!
    - refer to chapter 8 of charles crowley - there is a section on semaphores and dead-locks - in that, there is a section on deadlock prevention and semaphores - you will understand the importance of doing several semaphore operations atomically using a single semop() !!!
    - do read this section and try to figure out the importance of semaphores and their implementation details !!!

- semctl() is a versatile system call supporting different commands - one such is SETALL - to use SETALL, following is the syntax - semctl(id1,0, SETALL, u1) - in this context, param1 is the id of the semaphore object - param2 is ignored/ unused - normally, we set it to 0 - param3 is SETALL - param4 is the union - in this case of SETALL, array field of union is used - array field is initialized to point to an array of unsigned short elements - no of elements in this array is equal to the no of elements in the semaphore object !!! each element is used to fill the initial value of the corresponding semaphore in the semaphore array maintained in the semaphore object - if there are 2 elements in the semaphore object, we need an array of 2 unsigned short elements - if we have n semaphores in a semaphore object, we need n elements in the array - best way understand these aspects is to look into a sample code and refer to manual pages !!

Note: also refer to chapter 8 of crowley for examples/illustrations on semaphore implementations and their usage !!!

12. if you have analysed prod_1_2.c and cons_1_2.c, what is the advantage of using free slots counting semaphore and filled slots counting semaphore ???

- let us assume producer is scheduled first and continues executing - it will continue using up free slots and decrementing free slots semaphore count - in addition, it will also keep incrementing filled slot semaphore's count- as per our assumption, consumer is not scheduled - will scheduled in the future !!!(analyse for other scenarios !!!)
  - in the above case, what will happen to the producer, after all the free slots are consumed ??

    the producer
    will be blocked after all the free slots are used
    and free slots counting semaphore value drops to 0 !!
    - well, this is the purpose of the free slots counting semaphore - they maintain the resource count and they also manage synchronization of the producer and consumer - consumer part will be clear soon !!!

- using counting semaphores enables proper blocking and wake-up of producer and consumer as needed, with respect to resources, not critical sections !!
- this involves synchronization as well as resource counting - in fact, synchronization is a by product of resource counting, in this case !!!
    - what is synchronization here ??
        - blocking and wakeup of consumer with the help of filled slot counting semaphore via semop() operations is synchronization !!!
        - in the same way, producer is also synchronized using free slots counting semaphore by blocking and waking up as needed using semop() operations !!!!these semop() operations are implemented as system call APIs !!!
        - in synchronization, with the help of semaphore operations, one process can control another process's execution !!!
        - there are other synchronization mechanisms other than semaphores !!!

- resource counting is obvious - see the prod_1_2.c and cons_1_2.c !!!

- is it possible to eliminate using counting semaphores in this context ?? if so, what problem do we face, in the consumer ??
  - consumer may end up using garbage data - you may control the consumer using good, defensive programming, but you cannot eliminate busy waiting !!! busy-waiting in these cases may lead to inefficiency in multi-tasking !!!

- if you can use resource semaphores accordingly, the above problems will be solved, but any race-conditions will still persist !!!

  - to solve race conditions, you must introduce binary /critical section semaphores along with counting/resource semaphores !!!
  - appropriate combination depends on application and coding !! you must analyse concurrency, resources and race conditions of your processes/threads - use appropriate semaphores and operations !!!

13. if a parent process creates shared memory object and attaches to it, VADs are created for parent process with appropriate attributes and page tables are created !!
  - what happens, if a child process is created for such a parent process ?? particularly, what happens to shared memory related aspects for the child process ???
    - child gets duplicate VADs and in the case of shared memory VAD, page frames are shared forever - meaning, for reading and writing - this is not the case for other data pages/pageframes , which are treated as per copy-on-write rules !!!
  - in this case, parent uses shmget() and shmat() before calling fork() - after fork(), child inherits the shared memory segments of parent - we do not explicitly use shmget() and shmat(), in the child process !!!
  - in this case, parent and child processes will see the same set of virtual addresses for shared memory segments !!!

14. what happens, if shmget() and shmat() are used in parent
as well as in child process ???

- we may use shmget() and shmat()
  (- or, we may just use shmat(), only - in this case,
  id is the duplicated id in the child process !!!)

- in this case, parent and child may or may not
  see the same set of virtual addresses - this
  is due to certain implementation reasons in
  modern systems !!!

- whether parent and child use the same set of
  virtual addresses or not, their respective
  shared memory VADs will point to the same set
  of shared memory objects - which means, they will
  end up sharing the same set of page frames via
  the same set of shared page frames array stored
  in common shared memory objects !!!

- also note that the above case of different virtual addresses
  will also occur in the case of 2 different,unrelated processes that
  are working on the same shared memory region /share memory object
  as well - the reasoning is the same - even if 2 unrelated processes
  are attached to 2 different set of virtual addresses, their
  VADs are still pointing to the same shared memory object
  and in turn will end up sharing the same set of page frames !!!

15. what happens, if shmdt() is invoked in a process ?

- connection between the current process and the
  shm object is destroyed !!!
  - shmdt() also destroys the VAD associated
    with the shared memory region for this
    process - in short, any relationship
    with the shared memory region is destroyed
    for this process !!!
- if shmdt() is not invoked in a process, it is
  invoked by the system, when process terminates !!!
- when shmdt() is invoked, shm object is not destroyed !!1

- shm obj is destroyed, when a process invokes
  shmctl() !!!

- if a process is still attached to a shm obj and another process invokes shmctl() to destroy the shm object, system will mark the shm obj for future destruction and when all processess attached to the shm object have detached, actual destruction will occur !!!

- like the above, there are several rules governing different objects of the operating system and best way to learn these is as we work !!!

16. in the case of semaphores, following are the observations:

- in one of the assignments, initial value of the semaphore is set to 0
- child process decrements
- parent process increments
- in both child and parent, print semaphore values before and after semaphore operations - ideally, you must have seen that all values are printed as 0s !!!
- in some cases, initial value was set to 8 and tested- in this case, changes were apparent !!!

- in another assignment, 2 semaphores are used as below:
    - one semaphore is a critical section semaphore
        - initial value is set to 1
    - another semaphore is a binary semaphore- used as synchronization semaphore !!!
        - initial value is 0

    - why use synch. semaphore ???
        - it ensures that the data is valid for the reading process - otherwise, it may dealing with stale data - you can explore and understand more !!!
    - in this case, the synchronization semaphore may be incremented beyond the system's limit and after that, any operation is invalid !!!
        - in this context, that is all that is !!!
        - in realistic scenarios, we must code such that this scenario must not occur - it is the responsibility of the developer !!!