THE OPEN UNIVERSITY OF KENYA

DESIGN PLAN

Programme title	Bachelor of Science in Cybersecurity and Digital Forensics
Course title	Operating Systems
Learning Module number	06
Learning module title	Process Synchronization
Module Developer	Elisha Abade
Module duration in hours	8
Instructional Hour Equivalent (Divide duration by 2)	4
Reviewed by	
Vision	The innovative university for inclusive prosperity
Audience description	Learners of Cyber Security in first semester of second year
Instructions to learners ⓘ	In this course we shall be learning about the basic concepts of an Operating System. We'll begin by watching videos on Operating systems. You are encouraged to ensure that you have access to a reliable Internet and that your devices (computer, tablet or phone) have properly working multimedia systems. This module also presents a number of interactive and non-interactive activities. You will be required to complete all the activities.
Learning module description	This module aims to facilitate learners to have an understanding of the fundamental concepts of Operating Systems such as the history and evolution of Operating Systems as well as functions, structure and components of an Operating System.
Module objectives:	This module aims at facilitating learners to acquire knowledge about: 1. The definition of process synchronization 2. The critical section problem 3. Classical process synchronization challenges 4. Process scheduling algorithms
Module learning outcomes:	By the end of the module, you should be able to: 1. Explain what process synchronization means 2. Describe the critical section problem in process synchronization 3. Design solutions to critical section problem

	4. Appraise the various scheduling algorithms for concurrent process
Planned Learning Resources	
ACTIVITY 1: INTRODUCTION VIDEO 1: Pre-recorded lecture on topic emphasizing LEARNING OUTCOME 1: Factual knowledge. 	**Video 1: Basics of Process Synchronization (10 minutes)** Welcome to the first session of this module on process synchronization. In the previous session, we looked at what processes are, their life cycle and some of the management practices that an Operating System must perform in order for processes to achieve their intended goals. In this session, we shall be looking at the concepts behind the need for synchronization among processes. In most operating systems, processes are executed concurrently. Since there are many processes, a running process may be interrupted at any time before it completes execution. In the course of execution, processes also access shared data resources. Concurrent access to shared data may result in one process modifying data while another one wants to read the same data. This might lead to data inconsistency. In order to maintain consistency, there is a need to have mechanisms that ensure that there is an orderly execution of cooperating processes. **Illustration of the problem** In order to illustrate the situation described above, let's consider the classical producer-consumer problem. In this problem, we have two processes and a shared buffer. The role of the first process is to continuously put in values in the shared buffer while that of the second process is to continuously consume values from the shared buffer. So the producer continuously writes into the shared buffer while the consumer continuously reads from the shared buffer. Assume that we want to craft a solution to this problem through creating a process 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 process after it produces a new buffer and is decremented by the consumer after it consumes a buffer. Code snippets for these two processes can be given as follows: **Producer** ***while (true) { /* produce an item in next produced */*** ***while (counter == BUFFER_SIZE) ;*** ***/* do nothing */*** ***buffer[in] = next_produced;*** ***in = (in + 1) % BUFFER_SIZE;*** ***counter++;*** ***}***

Consumer

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

Since processes are independent, the two processes will run concurrently and there are chances that they will be "clashing" as they try to access the shared buffer. This will likely lead to a condition referred to as, "race condition".

Race Condition

This refers to an undesirable situation that occurs when two or more processes attempt to perform some operations at the same time but the nature of the operation requires that it is done in proper sequence in order for it to be done correctly.

Based on the example above:

1. counter++ could be implemented as:
   register1 = counter
   register1 = register1 + 1
   counter = register1
2. counter-- could be implemented as:
   register2 = counter
   register2 = register2 - 1
   counter = register2

Video 2: Critical Section Problem (8 minutes)

In this second session of this module, you will be learning about a critical section and its associated challenges.

A critical Section is a segment of code or the program that accesses or modifies the value of the variables in a shared resource. In a system of n processes {p0, p1, … pn-1}, each process has a critical section segment of code. When one process is executing a critical section, no other may be in its critical section.

The critical section problem is to design a protocol to solve this condition such that each process must ask for permission to enter critical section in entry section, may follow critical section with exit section, then remainder section.

Therefore the general structure of process P_i

```
do {
   Entry section
      Critical Section
   Exit Section
      Remainder section
}
```

The algorithm for Process Pi

```
do {
    while (turn == j);
          critical section
          turn = j;
         remainder section
} while (true);
```

Solution to Critical-Section Problem

There are a number of solutions that have been proposed to solve the critical section problem. These include:

1. **Mutual Exclusion** - If a process P_i 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.

Critical-Section Handling in OS

There are two approaches adopted by Operating Systems when it comes to handling critical sections. These depend on whether the OS kernel is preemptive or non- preemptive.

Preemptive kernels allow for preemption when it is running in kernel mode while non-preemptive ones run until one of the following events occur:

1. They exit kernel mode,
2. They block, or
3. They voluntarily surrender the CPU

Video 3: Mutex Locks and Semaphores (12 minutes)

The previous solutions are complicated and are generally inaccessible to application programmers. This made the Operating System designers to build software tools to solve the critical section problem.

Mutex locks are one of the simplest tools for this solution. In this approach, when a process wants to enter the critical section, it must first acquire() a lock then after completing the execution of the critical section, explicitly release() the lock.

These functions can be represented in code as follows:

```
acquire() {
    while (!available)
        ; /* busy wait */
        available = false;;
}

release() {
    available = true;
}
```

They can then be invoked by a process as follows:

```
do {
    acquire lock
        critical section
    release lock
    remainder section
} while (true);
```

The acquire() and release() functions implement a boolean variable that indicates whether a lock is available or not. Calls to ***acquire()*** and ***release()*** must be atomic. This is usually implemented via hardware atomic instructions.

But this solution requires busy waiting hence this lock is commonly referred to as a spinlock. Weaknesses like this have led to adoption of other techniques such as Semaphores that we shall be looking at in the next session.

Semaphore

A semaphore is an integer variable that is used to enforce mutual exclusion, avoid race conditions and implement synchronization between processes through two atomic operations, wait and signal.

It therefore provides a more sophisticated way (than Mutex locks) for processes to synchronize their activities.

In this approach, a semaphore S is an integer variable that can only be accessed via two indivisible (atomic) operations, wait() and signal(). These were originally referred to as P() and V() respectively. These two functions are defined as follows:

Definition of the wait() operation

```
wait(S) {
    while (S <= 0)
     ; // busy wait
```

```
        S--;
    }
```

Definition of the signal() operation

```
    signal(S) {
      S++;
    }
```

Semaphores can be broadly described in two classes, namely:

1. Counting semaphore: Refer to integer values that can range over an unrestricted domain
2. Binary semaphore – Refers to integer values that can range only between 0 and 1. They are the same as mutex locks.

Semaphore Implementation

A proper implementation of a semaphore must guarantee that no two processes can execute the ***wait()*** and ***signal()*** calls on the same semaphore at the same time. Therefore the implementation becomes the critical section problem where the ***wait*** and ***signal*** code are placed in the critical section.

The definition of wait and signal functions described above could still have a busy waiting in critical section implementation but only relying on the fact that the implementation code is short hence only a little busy waiting may occur coupled with the fact that the critical section is rarely occupied.

In the real-world, applications may spend lots of time in critical sections hence this is not an optimal solution. This led to another approach to implementing semaphores with no busy waiting.

Semaphore Implementation with no Busy waiting

In this approach, each semaphore is associated with a waiting queue. Each entry in a waiting queue has two data items: value (of type integer) and a pointer to the next record in the list. This implementation has two operations:

1. **block**: This places the process invoking the operation on the appropriate waiting queue.
2. **wakeup**: This removes one of the processes in the waiting queue and places it in the ready queue.

These operations can be represented in code as follows:

```
    typedef struct{
        int value;
       struct process *list;
    } semaphore;

    wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
```

	add this process to S->list;
	block();
	}
	}
	signal(semaphore *S) {
	S->value++;
	if (S->value <= 0) {
	remove a process P from S->list;
	wakeup(P);
	}
	}
	
	Video 4 Deadlock and Starvation
	In this session, we shall be looking at the concepts of deadlocks and starvation.
	
	A deadlock occurs when two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
	For example, let S and Q be two semaphores initialized to 1 and we have two processes, P_0 P_1 Such that we have their wait() and signal() functions implemented as:
	wait(S); wait(Q);
	wait(Q); wait(S);
	... ...
	signal(S); signal(Q);
	signal(Q); signal(S);
	
	This leads to a deadlock condition.
	
	Starvation on ther other hand occurs when there is indefinite blocking. In this case, a process may never be removed from the semaphore queue in which it is suspended.
	
	Another situation that is closely related to these two conditions is **Priority Inversion.** This refers to a scheduling problem in which a lower-priority process holds a lock that is needed by a higher-priority process. This problem is usually solved by priority-inheritance protocol.

ACTIVITY 2: READING READING MATERIAL 1 	In this section you have been provided with additional information on where you can get additional reading materials. Please go through them so as to allow you to have a full grasp of the contents of this module.

	a. Process Management Andrew S Tanenbaum. (2016). Modern Operating Systems Paperback. Pearson. pp 85 - 165; https://www.amazon.com/Modern-Operating-Systems-Andrew-Tanenbaum/dp/9332575770#detailBullets_feature_div
ACTIVITY 3: Comprehension questions: 	1) Define the two-phase locking protocol. 2) What three conditions must be satisfied in order to solve the critical section problem? 3) Explain two general approaches to handle critical sections in operating systems. 4) Explain the process of starvation and how aging can be used to prevent it. 5) How could an operating system that can disable interrupts implement semaphores? 6) Consider the following solution to the mutual-exclusion problem involving two processes P0 and P1. Assume that the variable turn is initialized to 0. Process P0's code is presented below. */* Other code */* *while (turn != 0) { } /* Do nothing and wait. */* *Cr itical Section /* . . . */* *turn = 0;* */* Other code */* For process P1, replace 0 by 1 in above code. Determine if the solution meets all the required conditions for a correct mutual-exclusion solution. 7) Write two short functions that implement the simple semaphore wait() and signal() operations on global variable S. 8) Explain the difference between the first readers–writers problem and the second readers–-writers problem. 9) Describe the dining-philosophers problem and how it relates to operating systems. 10) How can write-ahead logging ensure atomicity despite the possibility of failures within a computer system? 11) What overheads are reduced by the introduction of checkpoints in the log recovery system? 12) Describe the turnstile structure used by Solaris for synchronization.

LEARNING OUTCOME 2: Conceptual knowledge ACTIVITY 4: Video to be used.	
CASE 1:  	Deep in the high waters of Lake Victoria lies a stretch off Ringiti island at the border of Kenya and Uganda that is known to be a rich fishing ground. Fishing boats and ships from both countries share a common passage along this stretch. Unfortunately, the Kenyan and Ugandan fishing boats occasionally collide here when entering the shared stretch. Since the stretch is along the international waters, fishermen from both countries often claim the right to the use the passage. The two countries agreed on the following collision prevention protocol. A large floater was setup at the entrance of the common passage at Ringiti island. Before entering the passage, a captain must stop his ship/boat and use a smaller speed boat to reach the floater to see if it has a flag on it. If the floater is empty, the captain puts his country's flag indicating that his ship/boat is entering the stretch. His fishing boat then enters the stretch and once his fishing boat has cleared the pass, he uses the smaller speed boat to get back to the floater at the entrance to pick his flag, thus indicating that the shared stretch is free for use. If an arriving captain finds a flag on the floater, he leaves the flag there and repeatedly takes a siesta and rechecks the floater until it is found to be empty so that he can put his country's flag in it. 1) A fishing boat captain from Kenya claimed that the protocol could block the channel from access by other ships/boats forever. Discuss. 2) A Ugandan captain who heard of the claim from their Kenyan counterpart disputed this assertion. His argument was that this has never happened ever since the protocol was designed. Discuss. 3) Unfortunately, one day two fishing ships from both countries collided on this stretch. Explain the cause of the crash. 4) Following the crash, you have been approached to review the protocol and explain what could be wrong with it. Give an improvement to this protocol so as to avoid the incidents of boats ever colliding.

ACTIVITY 5: READING MATERIAL 	In this section you are expected to do further reading in the area of process synchronization mechanisms with specific focus on monitors and message passing in processes. 1) Message Passing a) https://www.geeksforgeeks.org/monitors-in-process-synchronization/ b) https://www.tutorialspoint.com/what-is-message-passing-technique-in-os c) https://www.cs.princeton.edu/courses/archive/fall20/cos318/lectures/9.MessagePassing.pdf 2) Monitors a) https://www.cs.princeton.edu/courses/archive/fall98/cs441/Lectures/Lec24.html b) https://jcsites.juniata.edu/faculty/rhodes/os/ch5c.htm c) https://www.isy.liu.se/edu/kurs/TSEA81/pdf/lecture_monitor_messages.pdf a) Having read the above articles, you are required to write a blog in the LMS with focus on monitors and message passing as process synchronization mechanisms.
ACTIVITY 6: ONLINE DISCUSSION 	Activities based on reading Material 5. Your course instructor will create a discussion forum in the LMS to facilitate online group discussions. You are required to read the discussion topic and give comments. You are also encouraged to comment on contributions from at least three members of your group. You can use the LMS platform to send questions to your instructor on the discussion topics that he/she has posted on the LMS. The group discussion will be graded based on a weight that will be indicated on the LMS.
LEARNING OUTCOME 3: PRACTICAL SKILLS VIDEO 3: 	In this section you will watch videos that present the classical problems of process synchronization. Click on these links to watch the videos on: 1. Bounded-Buffer Problem (15:48 minutes) 2. Readers and Writers Problem (15:31 minutes) 3. Dining-Philosophers Problem (20 minutes)

ACTIVITY 7: Learner practice sessions	In this session, you are required to do a "lightning talk" focusing on "*Critical Section and Process Synchronization*". In the "lightning talk", use your smartphone or any other video camera to record yourself in not more than "30 seconds" while explaining how the Operating System handles "***the critical section***". Note: 1. Upload your video with the captions <*fname_lname_talk6*>. where "*fname*" is your first name and "*lname*" is your last name (or surname). The video must not be more than 30 seconds long.
ASSESSMENT OF PRACTICAL SKILL:	In the above activity, you uploaded your video, <***fname_lname_talk6***>. It will be assessed by the instructor by looking at among others: a) Accuracy of the assertions you have made (5 Marks) b) Degree of completeness of your response to the task (3 Marks) c) Adherence to the requirements with regards to topic and length of the video. (2 Marks)
LEARNING OUTCOME 4: KEY/TRANSFERABLE SKILLS	In this section, you are required to watch a series of videos and also read from online resources that will help you to further your understanding of specific topics in process synchronization as follows: 1) Videos on selected topics a) Process synchronization b) Critical section c) Peterson's solution d) Test and lock e) Semaphores f) Disadvantages of Semaphores 2) Online reading resources In this section of further reading, you are required to use the provide resources to learn more about deadlock and starvation. These include: a) https://www.tutorialspoint.com/starvation-and-deadlock b) https://techdifferences.com/difference-between-deadlock-and-starvation-in-os.html c) https://www.tutorialspoint.com/difference-between-deadlock-and-starvation-in-os d) https://www.geeksforgeeks.org/difference-between-deadlock-and-starvation-in-os/

ACTIVITY 8	In this module, you have learnt about process synchronization in Operating Systems. You are required to write a two page essay on deadlocks and starvation and the Operating System mechanisms that are used to address these two issues. Your instructor will create an activity in the LMS that will allow you to submit this essay for assessment. The essay will be marked out of 15 Marks. Some of the guidelines to success in this activity include: a) Originality (avoid copying from the Internet and other sources) (5 Marks) b) Level of accuracy of the essay content (5 Marks) c) Completeness of content (3 Marks) d) Sticking to length (number of pages) requirements (1 Mark) e) Keeping to the theme (1 Mark)
QUIZZ: 	1. ..... refers to the ability of multiple process (or threads) to share code, resources or data in such a way that only one process has access to shared object at a time. a) Synchronization b) Mutual Exclusion c) Dead lock d) Starvation 2. ..... is the ability of multiple process to co-ordinate their activities by exchange of information a) Synchronization b) Mutual Exclusion c) Dead lock d) Starvation 3. ..... refers to a situation in which a process is ready to execute but is continuously denied access to a processor in deference to other processes. a) Synchronization b) Mutual Exclusion c) Dead lock d) Starvation 4. Which of the following is not the approach to dealing with deadlock? a) Prevention b) Avoidance c) Detection d) Deletion

	Answers 1. b) Mutual Exclusion 2. a) Synchronization 3. d) Starvation 4. d) Deletion
TAKE HOME MESSAGE	Your course instructor will create a feedback section in the LMS to facilitate provision of your take home message. You are required to give a brief description of what you have learnt in this module in not more than half a page (typed) in the feedback section provided.
Reference list	1. Andrew S Tanenbaum. (2016). *Modern Operating Systems Paperback, 5th Edition*. Pearson. 2. Silberschatz A., Galvin P. B. and Gagne G. (2008). *Operating System Concepts*, 8th Edition. Wiley. ISBN: 9780470128725 3. Meyers, M. (2016). *CompTIA A+ Certification Guide*. McGraw-Hill Education