Platforms and Operating Systems

EHC LVL5 - S02 2021/2022

Toqa Mahmoud

CU1900305

The Knowledge Hub Universities

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Many separate users can run processes at what seems to be the same time over most modern operating systems. These processes are not executing at the same time if the operating system is hosted by a hardware system with a single Central Processing Unit (CPU); they merely appear to be. The operating system allots CPU and other system resources to the processes that use them, switching these resources from one executing program to another (preemptively). This resource swapping creates the impression that the programs are operating at the same time. Several current operating systems permit many threads to operate within each process in a similar manner. Again, the operating system is in charge of allocating CPU and other resources to particular threads and switching these deployments from one running thread to another. This gives the impression that both threads are active at the same time (Multi-threading and the Operating System, 2017).

This report describes the multi- process environment, concurrent process and synchronization of these processes as well as inter process communication and sharing data. The resource allocation and management for multi-process environment, allocation of files, memory, I/Os, and Processors to concurrent process. Moreover, the different security services provided by operating system to assure the secure use of multi-user/multi-process environment were precisely described. It is required to assure the authorized access by the suitable identity and integrity of data and results of this environment. Finally, a detailed complete comparison of these services in different operating system including, Linux, Unix, Windows, and Android is illustrated.

2 CONCURRENCY: MUTUAL EXCLUSION AND SYNCHRONIZATION

Concurrent processing is a computing approach in which numerous processors run instructions at the same time in order to improve performance. Concurrent means that something happens while something else is happening. The jobs are divided into sub-types, which are then allocated to individual processors to complete concurrently, rather than sequentially, as they would be if only one processor were available (Concurrent Processes in Operating System, 28). The central themes of operating system design are all concerned with the management of processes and threads:

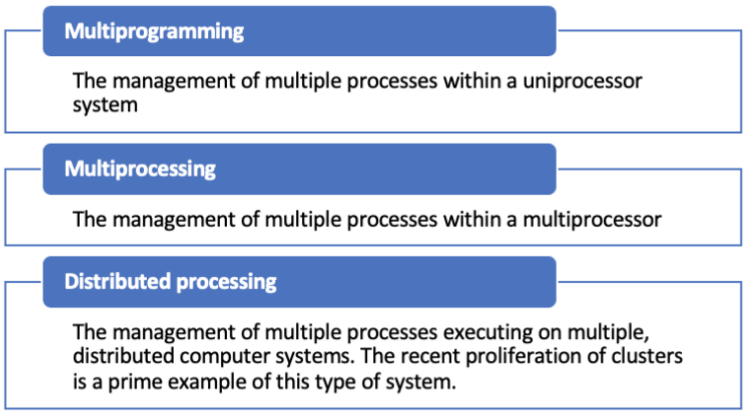

Multiprogramming
The management of multiple processes within a uniprocessor system
Multiprocessing
The management of multiple processes within a multiprocessor
Distributed processing
The management of multiple processes executing on multiple, distributed computer systems. The recent proliferation of clusters is a prime example of this type of system.

Figure 1 Process Types

Concurrency covers a wide range of development concerns, including process communication, resource sharing and competition (for memory, files, and I/O access), synchronization of many processes' activity, and processor time allocation. The report illustrates that these challenges exist in multiprocessing and distributed processing environments. Concurrency can be found in three separate situations as shown in figure 2 (Operating system design/concurrent, n.d.):

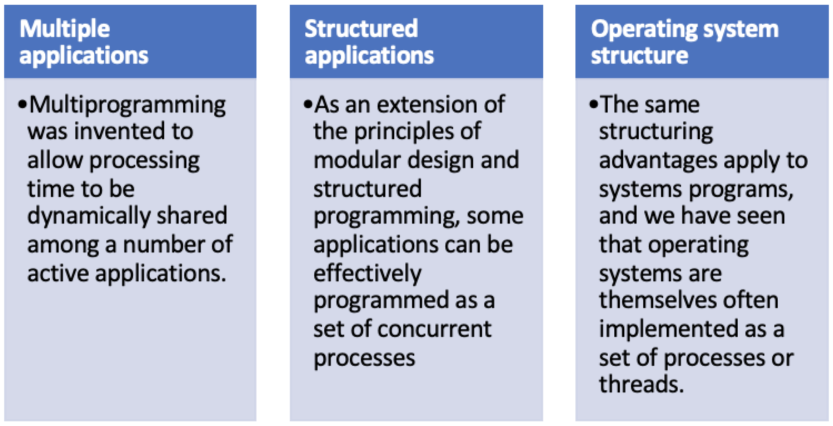

Multiple applications
•Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.
Structured applications
•As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes
Operating system structure
•The same structuring advantages apply to systems programs, and we have seen that operating systems are themselves often implemented as a set of processes or threads.

Figure 2 Concurrency Situations

2.1 PRINCIPLES OF CONCURRENCY

Multiple processes can not only be executed in parallel on a multiprocessor system, but they can also overlap. Both techniques might be considered instances of concurrent processing, and they both have the exact issues. The challenges with a uniprocessor originate from a fundamental feature of multiprogramming systems; The relative pace with which procedures are carried out can't be determined. It is dependent on the activity of other processes, the OS's handling of interrupts, and the OS's scheduling regulations. Figure 3 shows the difficulties that arise:

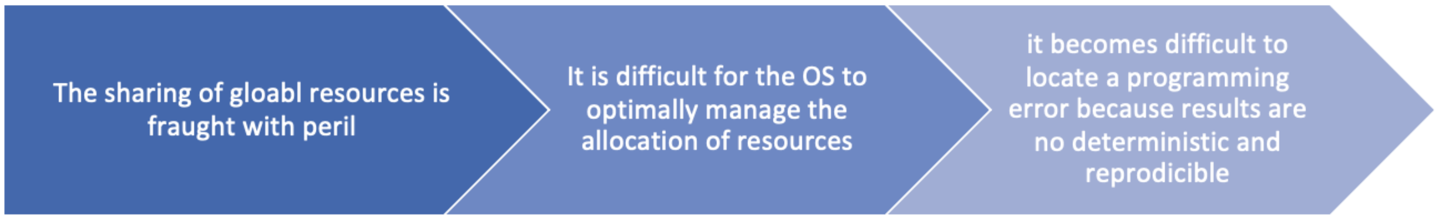

The sharing of gloabl resources is fraught with peril
It is difficult for the OS to optimally manage the allocation of resources
it becomes difficult to locate a programming error because results are no deterministic and reprodicible

Figure 3 Concurrency Difficulties

When many processes or threads read and write collected data under a race condition, the outcome is determined by the sequence in which the orders in the multiple processes are executed. Assume that the global variable **a** is shared by both processes, P1 and P2. P1 modifies **a** to the value 1 at a certain time during running, and P2 modifies **a** to the value 2 at a certain time during its operation. As a result, both tasks are competing to write variable **a**. The process that modifies last decides the final value of **a** in this case (Corbet, Kroah-Hartman, & Rubini, 2005). Figure 4 illustrates four points of OS concerns regarding concurrency.

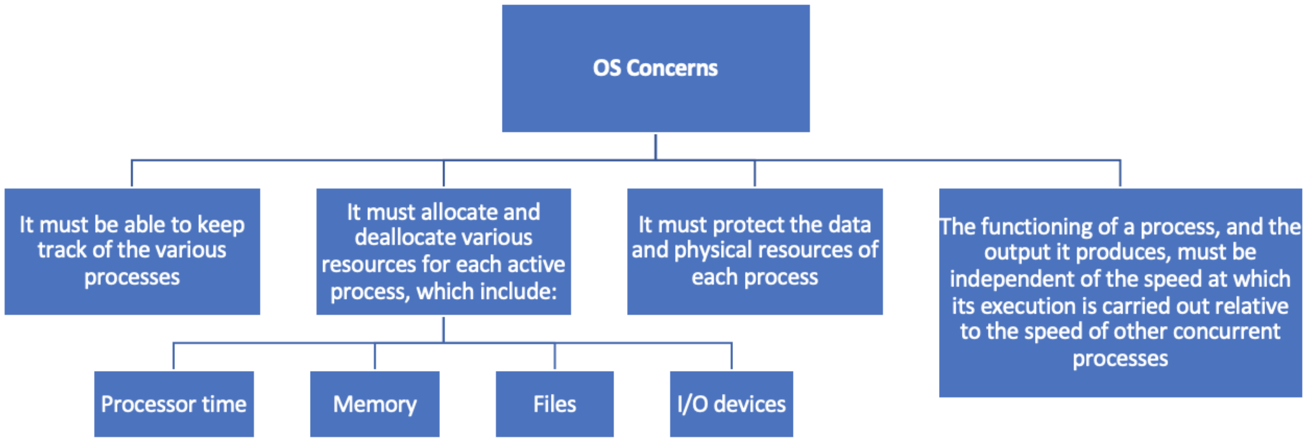

OS Concerns
It must be able to keep track of the various processes
It must allocate and deallocate various resources for each active process, which include:
It must protect the data and physical resources of each process
The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes
Processor time
Memory
Files
I/O devices

Figure 4 OS Concerns

2.1.1 PROCESS INTERACTION

The level to which processes are informed of other process's existence could be used to categorize the methods in which they interact. The three different levels of awareness and their implications are:

- **Processes that aren't aware of one another**: These are separate processes that aren't meant to interact. Despite the fact that the processes aren't cooperating, the operating system must be cautious about **competition** for resources.
- **Processes that are indirectly aware of one another**: These are processes that share access to a resource, such as an I/O buffer, but are not always aware of each other by their individual process IDs. These processes demonstrate **cooperation** in the sharing of a common resource.
- **Processes that are completely conscious of one another**: These are processes that can interact with one another via process ID and are designed to collaborate on a task. Such processes, yet again, demonstrate **cooperation**.

2.1.1.1 COMPETITION AMONG PROCESSES FOR RESOURCES

Three control issues must be addressed in the situation of competing processes. The first requirement is **mutual exclusion**. When multiple processes need accessibility to a particular non-shareable resource, it is critical that only one programme be permitted at a moment. Because the detailed criteria may not be evident, we cannot rely solely on the OS to comprehend and apply this limitation. Mutual exclusion, on the other hand, introduces two new control issues: **deadlock** and **starvation** (Stallings, 2018).

2.1.1.2 COOPERATION AMONG PROCESSES BY SHARING & COMMUNICATION

Processes that communicate with different processes without being directly notified of them are included in the case of **cooperation via sharing**. Multiple processes, for instance, may have access to the shared variables, files, or databases. Moreover, processes can consume and update shared data while not involving additional processes, but they should be aware that other processes can have accessibility to the exact data. As a result, the processes must work together to guarantee that the data they exchange is correctly handled by assuring data integrity. **Cooperation through communication**, on the other hand, refers to procedures that work together to achieve a single goal. The communication allows the various actions to be synchronized or coordinated. The programming language may have primitives for sending and receiving messages, or the OS kernel may offer them. Mutual exclusion is not really a control criteria for this type of cooperation because nothing is shared between processes while passing messages. However, the issues of deadlock and starvation continue to exist (Stallings, 2018).

2.1.1.3 MUTUAL EXCLUSION REQUIREMENTS

Any resource or capacity that will provide mutual exclusion assistance must have specific criteria including:

1. A single process is permitted inside the critical region at a time.
2. A noncritical component of a process must be stopped without affecting other operations.
3. A process getting access to a vital area must not be able to be postponed endlessly: no deadlock or starvation.
4. If no processes are in a critical part, whatever process that wants access to its critical section should be allowed.
5. There are no expectations about relative processor speeds or the quantity of processors.
6. A process typically spends a finite amount of time inside its crucial section (Stallings, 2018).

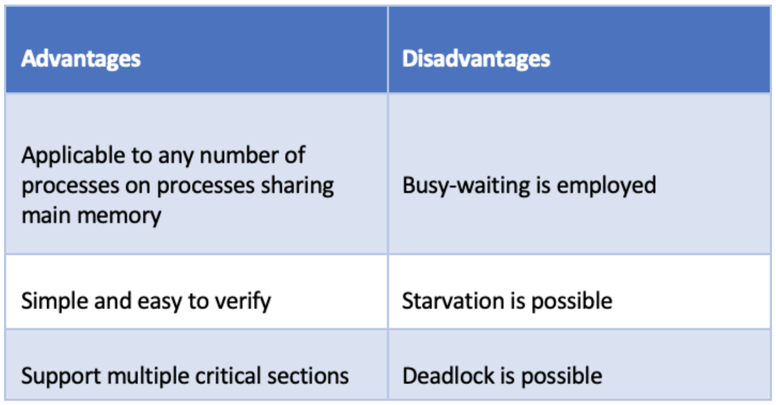
2.2 MUTUAL EXCLUSION: HARDWARE SUPPORT

Concurrent processes in a uniprocessor system could only be interleaved, not overlapped. Additionally, a process will keep running until it calls an OS service or is terminated. As a result, simply preventing a process from becoming interrupted is enough to ensure mutual exclusion. This functionality can be offered in the form of primitives for disabling and enabling interrupts specified by the OS kernel. Since the processor's capacity to interleave tasks is limited, the efficiency of execution could be considerably reduced. Another issue is that this strategy will not function in a multiprocessor environment.

2.2.1 SPECIAL MACHINE INSTRUCTIONS

Multiple processors share access to the same main memory in a multiprocessor system. In this situation, the processes act freely in a peer relationship. Mutual exclusion cannot be dependent on an interrupt mechanism between processors (Stallings, 2018). Advantages and disadvantages are illustrated in table 1.

Table 1 Advantages & Disadvantages of Special Machin Instructions



Advantages	Disadvantages
Applicable to any number of processes on processes sharing main memory	Busy-waiting is employed
Simple and easy to verify	Starvation is possible
Support multiple critical sections	Deadlock is possible

2.3 SEMAPHORES

A semaphore is an integer value that is used to communicate between processes. It can only be used for three procedures, which are all atomic: initialize, decrement, and increment. The decrement action may cause a process to become blocked, while the increment operation may cause a process to become unblocked. Also known as a **counting semaphore** or a **general semaphore**. It's impossible to predict whether a process would block or not when it decrements a semaphore. The two processes continue to run concurrently when one increments a semaphore and the other is woken up. On a uniprocessor system, there is no way of knowing whether process, will resume immediately. Because you can't determine whether the other processes is waiting once you signal a semaphore, the amount of unblocked processes could be zero or one.

The process which has been blocked the longest is the one to be freed from the queue; a **strong semaphore** is one whose definition includes this principle. A **weak semaphore** is one that doesn't specify the sequence in which processes are removed from the queue (Semaphores in Operating System, n.d.).

2.3.1 THE PRODUCER/CONSUMER PROBLEM

The producer/consumer issue is among the most typical issues encountered in concurrent processing. There are one or more producers who generate data (records, characters) and store it in a buffer. A single consumer is removing items from the buffer one by one. The system will be limited to avoid buffer operations from overlapping. That is, the buffer can only be accessed by one agent (producer or consumer) at a time. The issue is ensuring that the producer does not attempt to add data to a full buffer and that the consumer does not attempt to delete data from an empty buffer. To introduce a new and realistic constraint to the producer/consumer problem: the buffer must be finite. Because the buffer is viewed as a circular store, pointer values must be stated modulo the buffer size (Stallings, 2018).

2.3.2 IMPLEMENTATION OF SEMAPHORES

The semWait and semSignal actions must be implemented as atomic primitives. Implementing them in hardware or firmware is one option. If that fails, various schemes have been proposed. The issue is essentially one of mutual exclusion: Only a single process at a time can use the semWait or semSignal operations to manipulate a semaphore. Thus, any of the software approaches, such as Dekker's or Peterson's, might be utilised; however, this would result in a significant processing overhead. Another option is to employ one of the mutual exclusion strategies that are supported by hardware (Stallings, 2018).

2.4 MESSAGE PASSING

Two key conditions should be satisfied when processes communicate with each other: **synchronisation** and **communication**. To guarantee mutual exclusion, processes must be synchronised, and cooperating processes might have to communicate information. Message passing is one way to provide both features. The actual message passing function is usually implemented as a pair of primitives (send & receive), as shown in figure 5.

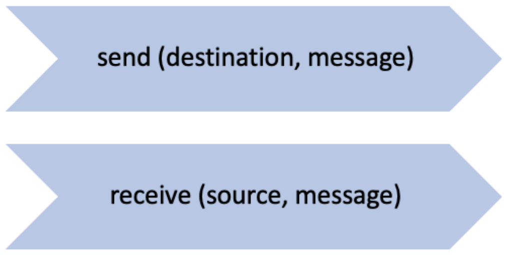


Figure 5 Message Passing

This is the bare minimum of operations required for processes to send messages. A process transmits data to another process indicated by a *destination* in the form of a *message*. By using the receive primitive and specifying the *source* and *message*, a process can receive data (Singh, n.d.).

2.4.1 SYNCHRONIZATION

The transmission of a message across two processes necessitates some degree of synchronisation: A message cannot be received until it has been transmitted by another process. Furthermore, we must define what happens to a process after it sends or receives a primitive. First, consider the send primitive. When a process sends out a receive primitive, it can do one of two things:

1. When a message has already been delivered, it is received and processing proceeds.
2. When there is no waiting message, the process is either (a) halted until one comes, or (b) continues to operate, abandoning the receiving attempt.

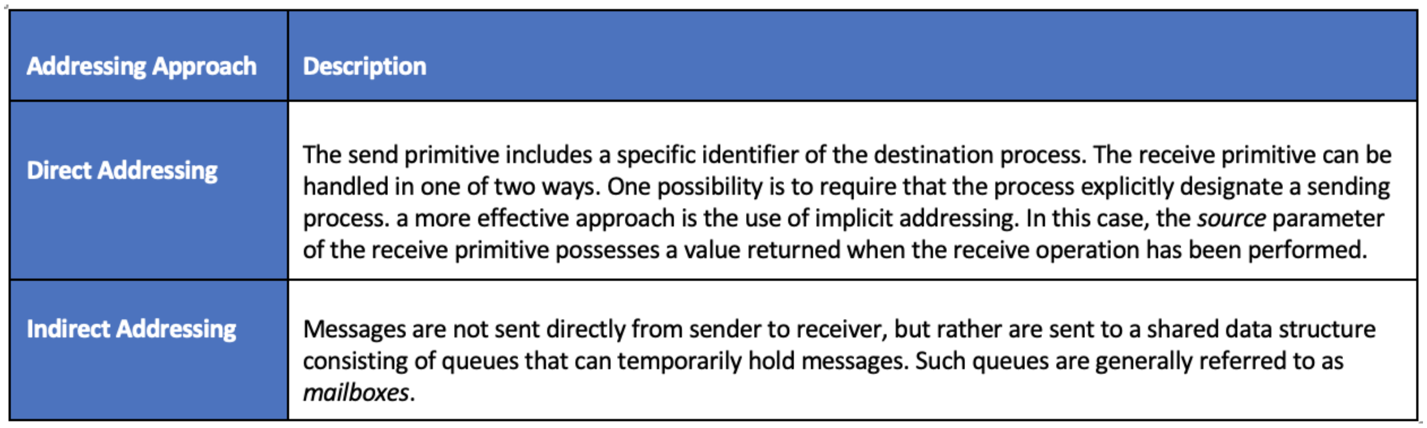
As a result, either the sender or the receiver can be blocking or nonblocking. There are three typical combinations:

1. **Blocking send and receive**: The sender and receiver are both blocked until the message is delivered; this is known as a rendezvous. This combination enables for tight process synchronisation.
2. **Only sending is nonblocking**: Receiving is blocked until the required message arrives. This has been the most beneficial combination. It enables a process to instantly transmit one or more messages to a range of destinations.
3. **Nonblocking send and receive**: Neither party is obligated to wait (Yadav, 2022).

2.4.2 ADDRESSING

There are two types of approaches for defining processes in send and receive primitives. Table 2 summarizes these two approaches.

Table 2 Addressing Approaches



Addressing Approach	Description
Direct Addressing	The send primitive includes a specific identifier of the destination process. The receive primitive can be handled in one of two ways. One possibility is to require that the process explicitly designate a sending process. a more effective approach is the use of implicit addressing. In this case, the *source* parameter of the receive primitive possesses a value returned when the receive operation has been performed.
Indirect Addressing	Messages are not sent directly from sender to receiver, but rather are sent to a shared data structure consisting of queues that can temporarily hold messages. Such queues are generally referred to as *mailboxes*.

The relationship between senders and receivers in indirect addressing could be one-to-one, many-to-one, one-to-many, or many-to-many. Figure 6 summarizes their properties while figure 7 visualizes them.

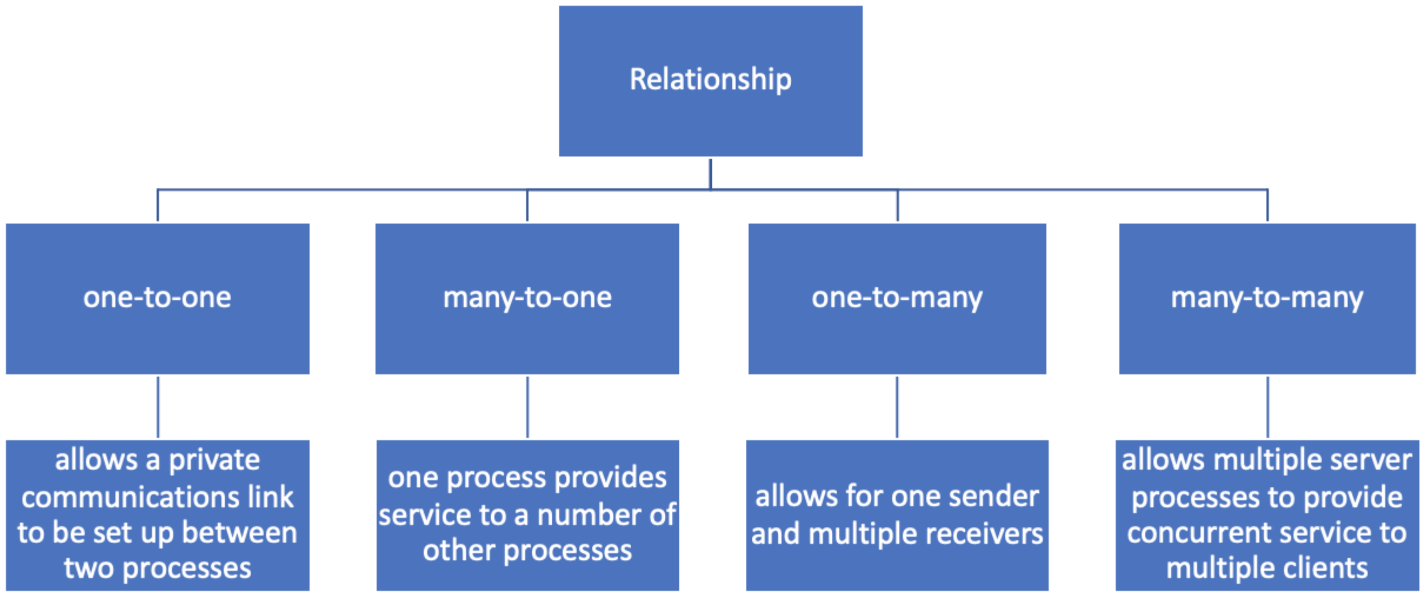

Relationship
one-to-one
many-to-one
one-to-many
many-to-many
allows a private communications link to be set up between two processes
one process provides service to a number of other processes
allows for one sender and multiple receivers
allows multiple server processes to provide concurrent service to multiple clients

Figure 6 Indirect Addressing Relationships 1

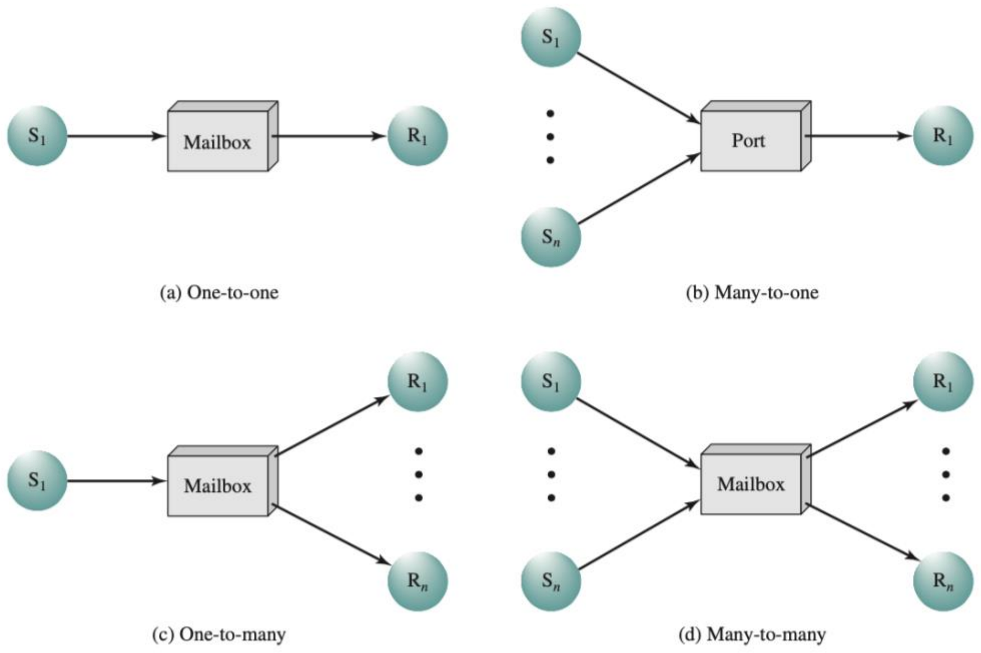


Figure 7 Indirect Addressing Relationship 2 (Stallings, 2018)

2.4.3 MESSAGE FORMAT

A basic message format for operating systems to allow variable-length messages is shown in Figure 8. The message is broken into two parts: a **header** that carries message information, and a **body** that includes the message's exact contents. The header may include information about the message's source and planned destination, as well as a length field and a type field to distinguish between different sorts of messages.

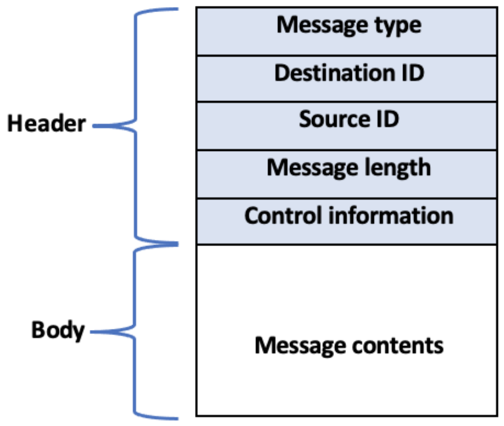


Figure 8 General Message Format

2.5 READERS/WRITERS PROBLEMS

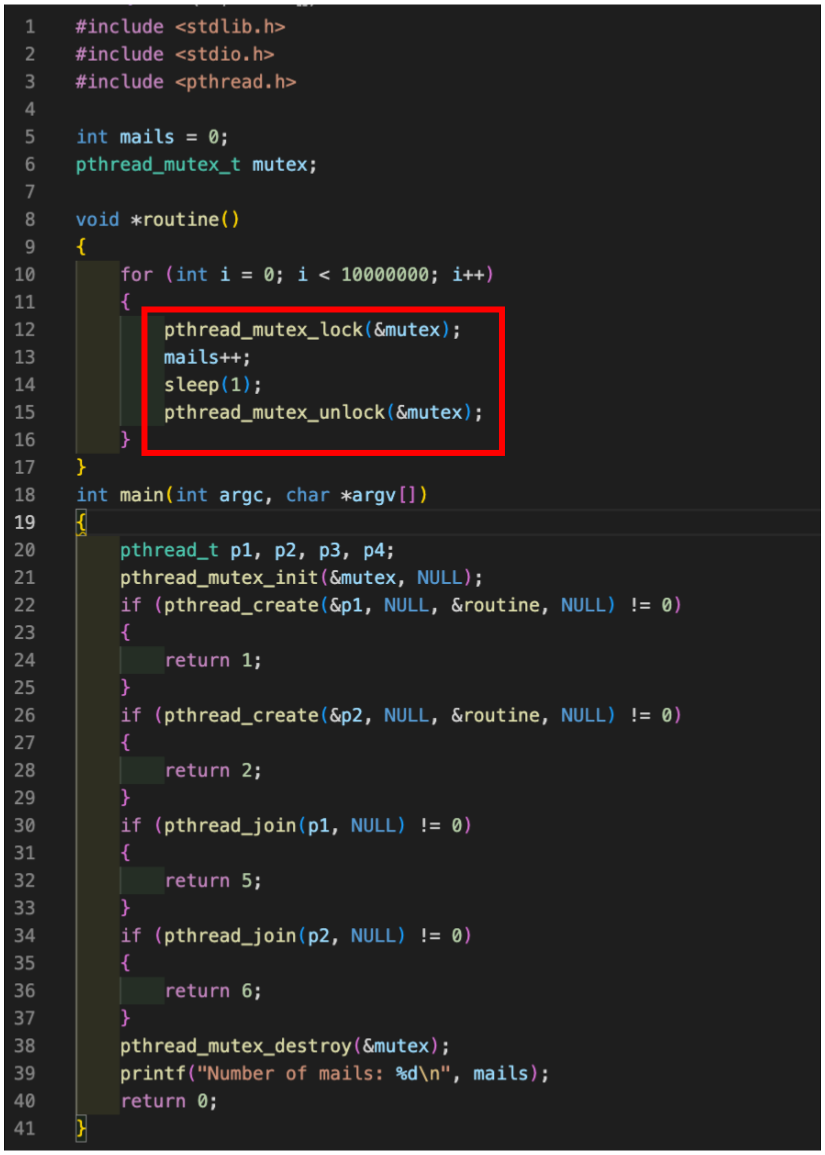
The difficulty of readers/writers is as follows: There is a data area that is shared by several processes. A file, a block of main memory, or even a bank of CPU registers could be used as the data area. There are certain processes that only read data from the data area (readers) and others that only write data to the data area (writers). The following are the requirements that should be met:

1. The file can be read by any number of people at the same time.
2. Only one writer can write to the file at a moment.
3. No one may view the file while it is being written to.

2.6 MUTEX IN C

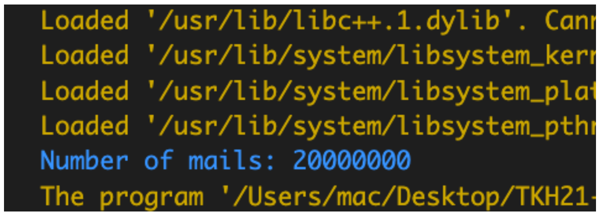
The code in figure 9a illustrates mutual exclusion in C programming language. The code increments mails 10000000 times by using 2 processes. The goal is to reach 20000000 mails, but this is not possible without mutex. Mutex locks a section of code until it is done execution. In the following code each process is locked using *pthread_mutex_lock()* and is unlocked by *pthread_mutex_unlock()* once it is done. As you can see in figure 9b, that the mails successfully incremented to 20000000. However, when mutex is not used it is impossible to reach the goal as shown in figure 9c. The number of mails were 10664511.

Figure 9 Mutex Code



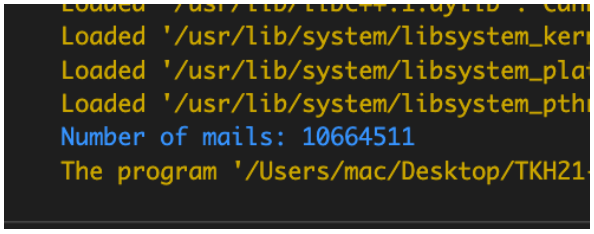
```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

int mails = 0;
pthread_mutex_t mutex;

void *routine()
{
    for (int i = 0; i < 10000000; i++)
    {
        pthread_mutex_lock(&mutex);
        mails++;
        sleep(1);
        pthread_mutex_unlock(&mutex);
    }
}
int main(int argc, char *argv[])
{
    pthread_t p1, p2, p3, p4;
    pthread_mutex_init(&mutex, NULL);
    if (pthread_create(&p1, NULL, &routine, NULL) != 0)
    {
        return 1;
    }
    if (pthread_create(&p2, NULL, &routine, NULL) != 0)
    {
        return 2;
    }
    if (pthread_join(p1, NULL) != 0)
    {
        return 5;
    }
    if (pthread_join(p2, NULL) != 0)
    {
        return 6;
    }
    pthread_mutex_destroy(&mutex);
    printf("Number of mails: %d\n", mails);
    return 0;
}
```

(a)



```
Loaded '/usr/lib/libc++.1.dylib'. Cann
Loaded '/usr/lib/system/libsystem_kern
Loaded '/usr/lib/system/libsystem_plat
Loaded '/usr/lib/system/libsystem_pthr
Number of mails: 20000000
The program '/Users/mac/Desktop/TKH21-
```

(b)



```
Loaded '/usr/lib/system/libsystem_kern
Loaded '/usr/lib/system/libsystem_plat
Loaded '/usr/lib/system/libsystem_pthr
Number of mails: 10664511
The program '/Users/mac/Desktop/TKH21-
```

(c)

Moreover, even though multiple processes exist only one mail is being processed per second because of **concurrency**. The next section discusses concurrency in more details.

3 CONCURRENCY: DEADLOCK AND STARVATION

This section illustrates the two main issues that affect concurrent processing: **deadlock** and **starvation**. The fundamental concepts of deadlock and the starvation issues are addressed first. Then the three approaches of dealing with deadlock, which are prevention, detection, and avoidance, will be discussed.

3.1 PRINCIPLES OF DEADLOCK

Deadlock is the state of a group of processes that are either competing for system resources or communicating with one another. When each process in a set is blocked expecting an event, which can only be activated by another blocked process in the set, the set is said to be deadlocked. Since none of the actions are ever triggered, the deadlock is permanent. In the general instance, unlike other problems in concurrent process management, there is no effective approach. **Reusable** and **consumable** resources are the two types of resources that can be identified (Eskicioglu & Marsland, 2001).

3.1.1 REUSABLE RESOURCES

A reusable resource is one that can be used by only one process at a time without being exhausted. Resources are obtained by processes, which are then released for reuse by other processes. Processors, I/O channels, main and secondary memory, devices, and data structures are instances of reusable resources. Take into consideration two processes competing for exclusive access to a disc file D and a tape drive T as an example of reusable resource deadlock. Table 3 depicts the actions carried out by the programmes. When each process has one resource and wants the other, a deadlock arises:

Process P		Process Q	
Step	Action	Step	Action
p0	Request (D)	q0	Request (T)
p1	Lock (D)	q1	Lock (T)
p2	Request (T)	q2	Request (D)
p3	Lock (T)	q3	Lock (D)
p4	Perform Function	q4	Perform Function
p5	Unlock (D)	q5	Unlock (T)
p6	Unlock (T)	q6	Unlock (D)

Table 3 Two Processes Competing for Reusable Resources (Stallings, 2018)

When the two processes proceed to their second request, a deadlock develops. It is challenging to deal with that sort of deadlock using system design limitations when the quantity of memory to be demanded is unknown earlier. The easiest method to cope with this issue is to use virtual memory to effectively eliminate the possibility (Stallings, 2018).

3.1.2 CONSUMABLE RESOURCES

A consumable resource is one that can be produced and used (consumed). There is usually no limit to the quantity of consumable resources of a given category. Any amount of these resources can be created by an unblocked production process. If a consuming process obtains a resource, then the resource is no longer available. Interrupts, signals, messages, and information in I/O buffers are types of consumable resources. Figure 10 shows an example of a deadlock using consumable resources, every process tries to receive a message from another process before sending a message to another process:

```
         P1                       P2
…                          …
Receive (P2);              Receive (P1);
…                          …
Send (P2, M1);             Send (P1, M2)
```

Figure 10 Consumable Resource Deadlock (Stallings, 2018)

If the receive is blocking, a deadlock develops. The deadlock is caused by a design flaw. Such mistakes can be minor and challenging to spot. Furthermore, the deadlock may be caused by an uncommon set of events, thus a software could be in use for months or even years before the deadlock happens. There is no one effective technique for dealing with every sort of deadlock, however section 3.2 discusses three common approaches shown in figure 11 (Stallings, 2018):

Deadlock prevention	Deadlock avoidance	Deadlock detection
Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening.	Do not grant a resource request if this allocation might lead to deadlock.	Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover.

Figure 11 Deadlock Approaches

3.1.3 RESOURCE ALLOCATION GRAPHS

The resource allocation graph, proposed by Holt, is a valuable tool for defining the allocation of resources to processes. Each process and resource is depicted by a node in the resource allocation graph, which is a guided graph that shows the state of the system of resources and processes. A graph edge pointing from a process to a resource denotes a resource that the process has demanded but not yet received, as shown in figure 12a. Every instance of a resource is represented by a dot inside a resource node. I/O devices assigned by a resource management module in the OS are an illustration of resource types that can have many copies. A graph edge pointing from a reusable resource node dot to a process shows that the request was approved, as shown in figure 12b; that is, the process was given one unit of that resource. A graph edge connecting a consumable resource node dot to a process denotes that the process is the resource's producer. A deadlock is depicted in figure 12c. Each of the resources Ra and Rb has only one unit. P1 has Rb and requests Ra, whereas P2 has Ra but requests Rb. Figure 12d has the same architecture as Figure 12c, but there is no deadlock because each resource is available in numerous units (Holt, 1972.).

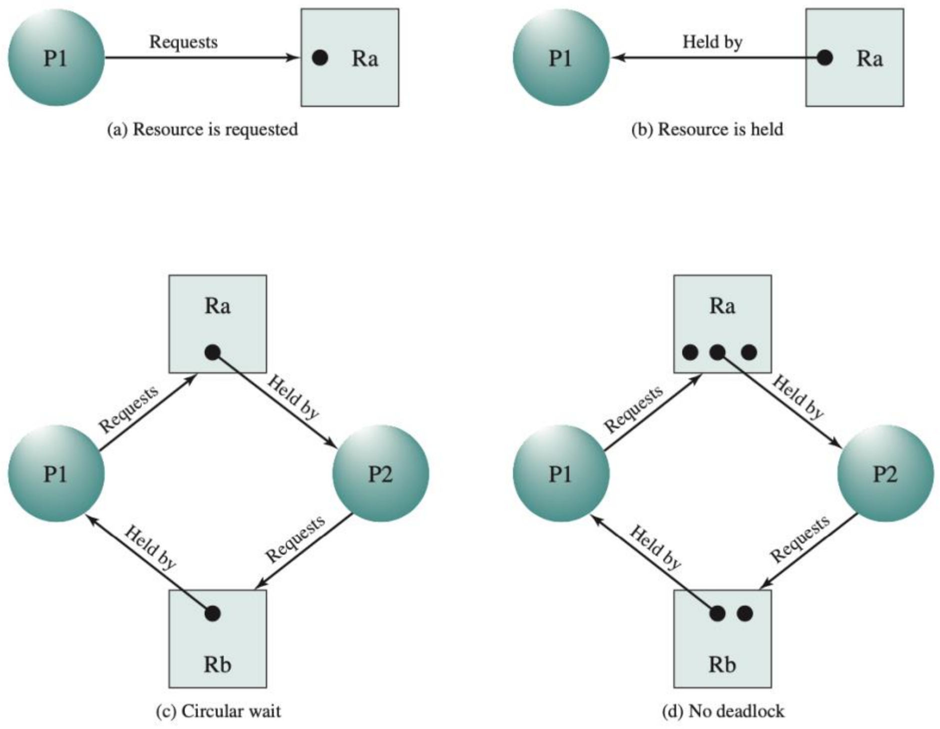


Figure 12 Resource Allocation Graphs (Stallings, 2018)

3.1.4 THE CONDITIONS FOR DEADLOCK

Three policy requirements should occur for a deadlock to happen:

1. **Mutual exclusion**: At any given time, a resource could only be used by single process. Any process cannot be permitted a resource unit it has been assigned to another process.
2. **Hold and wait**: A process may keep assigned resources while waiting for more to be appointed.
3. **No preemption**: No resource can be taken from a process that was consuming it without warning.
4. **Circular wait**: A closed chain of processes in which every process demands at least single resource from the preceding process, as shown in figure 12c (Handling Deadlocks, 2022).

3.2 DEADLOCK PREVENTION

Deadlock prevention is the method of designing a system to eliminate the potential of a deadlock. Techniques for avoiding, can be divided into two categories. Preventing the occurrence of one of the first three essential circumstances previously described is an **indirect technique** of deadlock prevention. Preventing the occurrence of a circular wait is a **direct technique** of deadlock prevention.

Mutual Exclusion The mutual exclusion, in general, cannot be denied. If accessibility to a resource needs mutual exclusion, the OS should allow mutual exclusion. Certain resources, including files, can offer multiple read access but only a single write access. Even if multiple processes involve write permission, a deadlock can arise.

Hold and Wait The hold-and-wait condition can be avoided by forcing a process to request the required resources at once and then blocking it when all requests are fulfilled. This strategy is ineffective in two respects. A process may be stalled for an extended period of time while waiting for all of its resource demands to be fulfilled when it might have done so with only a portion of the resources. Second, resources granted to a process could go underutilised for a long time, preventing them from being utilised by other processes. Another issue is that a process might not be aware of the resources it will need ahead of time.

No Preemption There are numerous strategies to avoid this condition. Firstly, if a process that would be holding certain resources is disallowed a new request, the process should release those resources and, if necessary, request them along with the new resource. If a process demands a resource that is already being used by another process, the OS can interrupt the second process and force it to release its resources. Only if no two processes had the same priority would this technique prevent deadlock.

Circular Wait By establishing a linear ordering of resource classes, the circular wait situation can be avoided. If a process is given resources of type R, it can only demand resources of types that come after R in the ordering. Circular wait prevention, like hold-and-wait prevention, may be unproductive, because it will slow down processes and limit resource access.

3.3 DEADLOCK AVOIDANCE

Deadlock avoidance permits the three necessary criteria to exist, but makes careful decisions to avoid reaching the deadlock point. As a result, avoidance allows for more concurrent behaviour than prevention. With deadlock avoidance, a dynamic decision is performed to see if the present resource allocation request will cause a deadlock if granted. To avoid a deadlock, you must be aware of future process resource requirements, figure 13 illustrates the two approaches of deadlock avoidance.

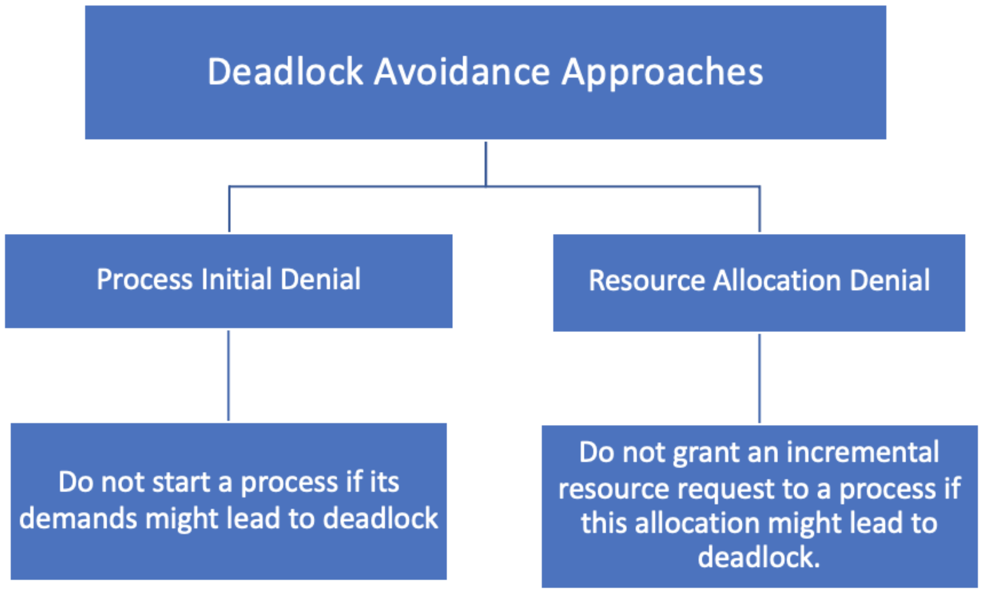

Deadlock Avoidance Approaches
Process Initial Denial
Resource Allocation Denial
Do not start a process if its demands might lead to deadlock
Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

Figure 13 Deadlock Avoidance Approaches

Deadlock avoidance has the benefit of not requiring preemption and rollback processes, as is the case with deadlock detection, and is less restrictive than deadlock prevention. However, there are certain limitations to its application:

- Each process' maximum resource need must be defined in advance.
- There must be a set number of resources to allot, and the sequence the processes run in must be unrestrained by any synchronisation needs.
- No process can leave while resources are being held.

3.4 DEADLOCK DETECTION

Deadlock detection solutions do not impose any restrictions on resource accessibility or process activity. Demanded resources are given to processes anytime possible. The operating system runs an algorithm to perceive the circular wait state on a regular basis.

3.4.1 DEADLOCK DETECTION ALGORITHM

Deadlock can be checked as frequently as every resource request. Scanning each resource request has two benefits: it allows for early detection and the technique is reasonably easy because it is based on gradual changes in the system's state. However, regular inspections eat up a lot of processing time.

3.4.2 RECOVERY

Once a deadlock has been identified, a recovery plan is required. The below are some conceivable ways, in order of increasing complexity:

1. Stop any processes that are stuck.
2. Restart all processes after backing up each blocked process to a previously determined stage.
3. Terminate stuck processes one by one until the deadlock is gone.
4. Circumvent resources one by one until deadlock is eliminated.

3.4.3 AN INTEGRATED DEADLOCK STRATEGY

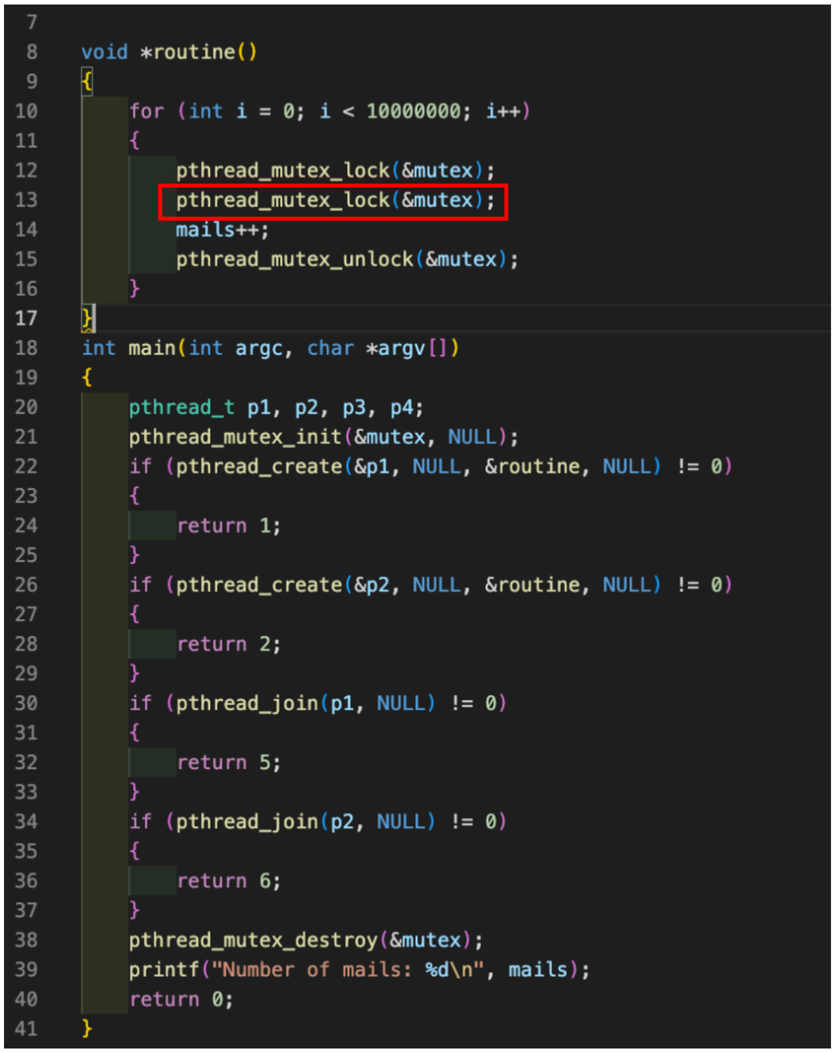
All of the ways for dealing with deadlock have advantages and disadvantages. Using diverse tactics in various situations is more efficient. Howard offers the following strategy:

- Divide resources into a variety of resource categories.
- To avoid deadlocks across resource classes, use the linear ordering method to prevent circular waits.
- Use the best suited algorithm Inside a resource class (Howard, 1973).

3.5 DEADLOCK IN C

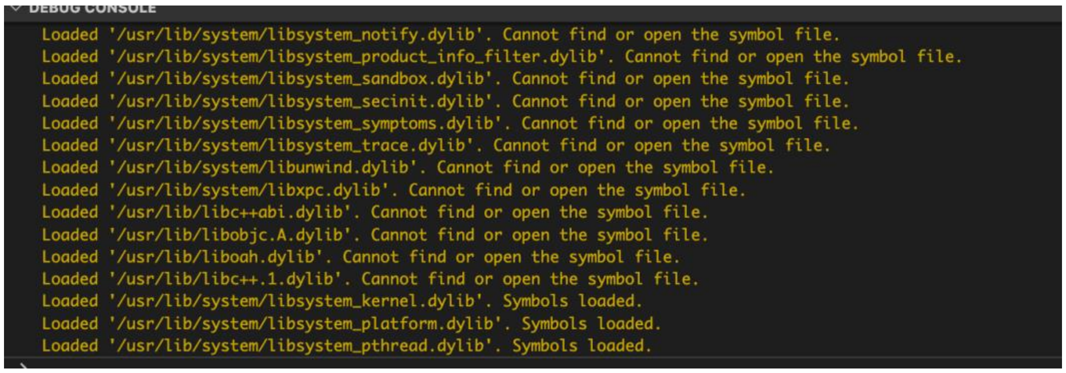
Figure 14 illustrates a simple deadlock code. Using the same code as the one in figure 9, I added another pthread_mutex_lock() in figure 14a. This results to a deadlock because a mutex was locked twice on the same thread, as shown in figure 14b the program never finishes execution and it is stuck forever. This is because the lock has already been taken and it is trying to take it again. Therefore, locking a mutex a second time in a thread is going to block everything.

Figure 14 Deadlock Code



```c

void *routine()
{
    for (int i = 0; i < 10000000; i++)
    {
        pthread_mutex_lock(&mutex);
        pthread_mutex_lock(&mutex);
        mails++;
        pthread_mutex_unlock(&mutex);
    }
}
int main(int argc, char *argv[])
{
    pthread_t p1, p2, p3, p4;
    pthread_mutex_init(&mutex, NULL);
    if (pthread_create(&p1, NULL, &routine, NULL) != 0)
    {
        return 1;
    }
    if (pthread_create(&p2, NULL, &routine, NULL) != 0)
    {
        return 2;
    }
    if (pthread_join(p1, NULL) != 0)
    {
        return 5;
    }
    if (pthread_join(p2, NULL) != 0)
    {
        return 6;
    }
    pthread_mutex_destroy(&mutex);
    printf("Number of mails: %d\n", mails);
    return 0;
}
```

(a)



```
DEBUG CONSOLE
Loaded '/usr/lib/system/libsystem_notify.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/system/libsystem_product_info_filter.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/system/libsystem_sandbox.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/system/libsystem_secinit.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/system/libsystem_symptoms.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/system/libsystem_trace.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/system/libunwind.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/system/libxpc.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/libc++abi.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/libobjc.A.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/liboah.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/libc++.1.dylib'. Cannot find or open the symbol file.
Loaded '/usr/lib/system/libsystem_kernel.dylib'. Symbols loaded.
Loaded '/usr/lib/system/libsystem_platform.dylib'. Symbols loaded.
Loaded '/usr/lib/system/libsystem_pthread.dylib'. Symbols loaded.
```

(b)

4 OPERATING SYSTEM SECURITY

4.1 INTRUDERS AND MALICIOUS SOFTWARE

The main security concern in the architecture of any operating system is to prevent or detect attempts by a user or malicious software (malware) to gain unauthorised rights on the system, particularly root access. The dangers and solutions connected to this security vulnerability are briefly summarised in this section. **Intruders** and **malicious** software are the two types of system access threats. (Silberschatz, Gagne, & Baer , 2005).

4.1.1 INTRUDERS

Intruders are known to be one of the main known threats. Table 4 summarizes Anderson's identifications of the three categories of intruders (James & Anderson , 1980).

Table 4 Intruder Classes

Intruder Class	Description
Masquerader	A person who is unauthorized to use a computer and who exploits a legal user's account by breaching a system's access rules.
Misfeasor	A legal user who gains access to data, programmes, or resources to which he or she is not permitted, or who is granted such access but abuses it.
Clandestine user	An individual who confiscates supervisory control of a system and employs it to circumvent auditing and access controls or suppress audit collection.

4.1.2 MALICIOUS

Programs that target flaws in computing systems are among the most advanced sorts of attacks to computer systems**. Malicious software,** or **malware**, is the term for such dangers. The risks can reach application programmes, utility programmes like editors and compilers, and kernel-level applications in this environment. There are two types of malicious software: **parasitic** and **independent**. Parasitic software requires a host programme to run. Figure 15 summarizes both types.

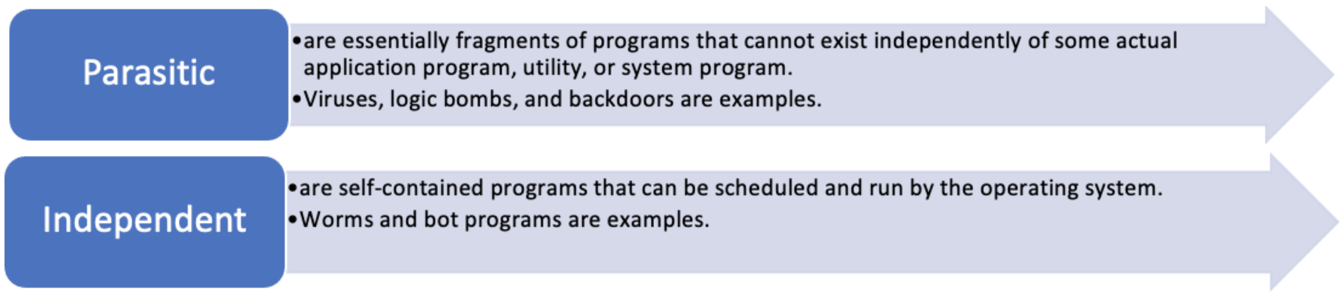

Parasitic
•are essentially fragments of programs that cannot exist independently of some actual application program, utility, or system program.
•Viruses, logic bombs, and backdoors are examples.
Independent
•are self-contained programs that can be scheduled and run by the operating system.
•Worms and bot programs are examples.

Figure 15 Malicious Software Categories

4.2 COUNTERMEASURES

4.2.1 INTRUSION DETECTION

Intrusion detection is a security service that detects and examines event logs in order to detect and notify of attempts to access system resources in an illegal manner in almost real time. Intrusion detection systems (IDSs) are divided into two categories as shown in table 5:

Table 5 Intrusion detection systems

Intrusion detection systems (IDSs)	Description
Host-based IDS	Monitors the characteristics of a single host and the events occurring within that host for suspicious activity
Network-based IDS	Monitors network traffic for particular network segments or devices and analyzes network, transport, and application protocols to identify suspicious activity

Three logical components that make up an IDS:

- **Sensors**: Sensors are in charge of gathering data. Any portion of a system that potentially hold evidence of an attack could be the sensor's input.
- **Analyzers**: Analyzers are devices that receive data from one or even more sensors or other analysis tools. The analyzer is in charge of detecting whether or not an intrusion has occurred. This component's output is a notification when an attack has occurred.
- **User interface**: An IDS's user interface allows a user to monitor output from the system and manage its actions.

Intrusion detection systems are often intended to identify both human and malicious software intruder activities.

4.2.2 AUTHENTICATION

User authentication is the key component and main line of defence in most computer security scenarios. Most kinds of access control and user accountability are based on user authentication. User authentication is the process of authenticating an identity asserted by or for a system entity. There are two steps in the authentication process:

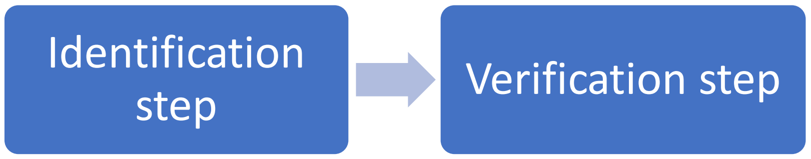


Figure 16 Authentication Process

The method of confirming the authenticity is through user authentication. There are four basic methods for verifying a user's identity that can be used separately or in combination:

1. **A thing the person knows**: a password, PIN, or the answers to a set of predetermined questions.
2. **A thing the person owns**: e-keycards, smart cards, and physical keys. A token is the name for this type of authenticator.
3. **A thing about the person (static biometrics):** fingerprint, retina, and face recognition.
4. **A feature of the person (dynamic biometrics):** speech pattern recognition, handwriting traits, and typing rhythm.

4.2.3 ACCESS CONTROL

Access control establishes a security policy that determines who or what could have accessibility to each specific system resource, as well as the sort of access allowed in each case. It connects a user to system resources such as programs, operating systems, firewalls, routers, files, and databases. A user requesting permission must first be authenticated by the system. The authentication function usually determines whether a user is allowed to access the system in anyway. The access control function then decides if the user's specific request for access is allowed. An authorization database is maintained by a security administrator that indicates what sort of access to which resources this user has. This database is used by the access control function to determine whether or not to give access.

4.2.4 FIREWALLS

Firewalls can safeguard a local system or network of systems from network-based security risks while still allowing contact with the outside world through wide area networks and the Internet. Typically, a firewall is a specialised computer that connects with computers outside of a network and has extra security features to safeguard important files on network computers. It's used to support external networks, particularly Internet connections and dial-in lines. Personal firewalls, which are focused on one workstation or PC and are deployed in hardware or software, are also widespread. The following firewall design goals must be met:

1. Restricting accessibility to the local network to the firewall.
2. Only authorised traffic will be permitted.
3. The firewall is impenetrable by itself.

4.3 ACCESS CONTROL

Access control can be handled by the operating system, the file system, or both. At both levels, the concepts that have generally been implemented are the constant. In this section, examining access control from the perspective of file access control is discussed, then the subject is broaden to include policies that apply to a number of system resources.

4.3.1 FILE SYSTEM ACCESS CONTROL

A user can be recognized by the system using the user-access control technique. A identity that determines permitted operations and file accesses can be connected with each user. The operating system could then use the user profile to enforce policies. Access to individual entries or even parts of information must be controlled by the database management system.

4.3.2 ACCESS CONTROL POLICIES

An access control policy specifies the sorts of access that are allowed, under what conditions, and by whom. The following are the different types of access control policies:

- **Discretionary access control (DAC):** Limits access depending on the requestor's identification and authorizations that specify what requestors can and cannot do.
- **Mandatory access control (MAC):** This system restricts access by correlating security labels to security clearances.
- **Role-based access control (RBAC):** Controls access depending on users' roles in the system and rules defining which accesses are granted to users in specific roles.
- **Attribute-based access control (ABAC):** Limits access depending on the user's characteristics, the resource being accessed, and the current environment.

4.4 OPERATING SYSTEMS HARDENING

For most major operating systems, proper security setup guidelines and checklists exist, and as such must be examined, although constantly guided by the individual demands of each organisation and their systems. Automated tools may be offered in some circumstances to aid in the security of the configuration. To protect an operating system, the National Institute of Standards and Technology recommends taking the following basic measures (Guide to General Server Security, 2008):

4.4.1 OPERATING SYSTEM INSTALLATION: INITIAL SETUP AND PATCHING

The deployment of the operating system is the first step in ensuring system security. An unpatched network-connected device is susceptible to attack during setup or usage. As a result, it's critical that the system isn't disclosed when it's in this vulnerable state. New devices should ideally be built on a secure network. Extra software packages must be installed only if they are essential for the system's function during the initial setup. The complete boot process must be protected as well. Additional device driver code must also be selected and installed with caution, as it runs with full kernel privileges and is frequently provided by a third party.

4.4.2 REMOVE UNNECESSARY SERVICES, APPLICATION, AND PROTOCOLS

Since any software running on a system may include software vulnerabilities, the threat is clearly decreased if less software packages are able to run. To increase security, the system design process must determine what is actually necessary for a specific system, so that an appropriate degree of capability is given while removing any software that is not needed. It is recommended that only the needed packages be installed during the initial installation.

4.4.3 CONFIGURE USERS, GROUPS, AND AUTHENTICATION

The sorts of users on the system, their privileges, the pieces of data they could access, and where and how they are created and authorised should all be considered during the system planning process. Furthermore, it is extremely desired that these users only have enhanced rights when they are required to complete a task. Any default accounts created as part of the system installation should also be secured at this time. Those that aren't needed should be eliminated or at the very least disabled. System accounts that handle system services must be disabled from being used for interactive logins. Any default passwords should be changed to new passwords with adequate security. Any policy relating to authentication credentials, particularly password security, is also set up.

4.4.4 CONFIGURE RESOURCE CONTROLS

When the users and their related groups have been identified, permissions on data and resources can be adjusted to fit the policy. This could be used to restrict who can run specific applications, particularly ones that affect the system state, or to restrict who could read or write data in certain directory trees. Many security-hardening guides include lists of modifications to the default access configuration that should be made to enhance security..

4.4.5 INSTALL ADDITIONAL SECURITY CONTROLS

Additional security technologies, such as antivirus software, host-based firewalls, IDS or IPS software, or application white-listing, may be installed and configured to increase security. Some of them may be pre-installed with the operating system, but they are not configured or activated by default. Others are purchased and used third-party products. Antivirus software is a key security element for many systems, given the increasing presence of malware. To detect and even react to some forms of attacks, IDS and IPS software may integrate additional techniques such as traffic monitoring or file integrity checks.

4.4.6 TEST THE SYSTEM SECURITY

Security testing is the last step along the way of safeguarding the core operating system. The purpose is to confirm that the security management setup stages are followed correctly, as well as to detect any potential vulnerabilities that need to be addressed. Many security-hardening manuals provide appropriate checklists. There are additional programmes that analyse a system to verify that it complies basic security criteria, as well as check for common vulnerabilities and poor configuration practises. This should be done once the system has been hardened, and then repeated on a regular basis as part of the security maintenance program.

4.5 SECURITY MAINTAINANCE

The procedure of preserving security is ongoing once the system has been properly constructed, secured, and deployed. This is due to the continually evolving environment, new vulnerabilities being discovered, and thus new threats being exposed. According to the National Institute of Standards and Technology, this security maintenance process should also include the following steps:

- Analyzing and tracking logging data.
- Making routine backups.
- Recovering from data security breaches.
- Periodic system security testing
- Applying proper software maintenance procedures to patch and upgrade any key software, as well as monitor and change configuration as required.

4.5.1 LOGGING

Effective logging allows system administrators to rapidly and correctly determine what occurred in the case of a system breach or failure, allowing them to concentrate their remediation and recovery efforts more efficiently. The system, network, and applications can all create logging data. Because it is dependent on the server's security requirements and information sensitivity, the scope of logging data acquired should be defined during the system planning phase. Logging can generate a substantial amount of data. To help manage the total size of the logging data, a proper automatic log rotation and archive scheme should be setup. Automated analysis is recommended since it is more likely to detect anomalous behaviour.

4.5.2 DATA BACKUP AND ARCHIVE

Another key measure that aids in protecting the integrity of the system and user data is creating regular backups of data on the system. **Backup** is the procedure of generating regular copies of information in order to restore deleted or damaged data in a short period of time, usually a several hours to several weeks. **Archive** is the process of archiving data over a long period of time, such as months or years, in order to meet legal and operational requirements for accessing historical data. Although they meet different purposes, these procedures are frequently integrated and handled jointly. During the system planning stage, identify the backup and archive requirements and policies.

5 OS COMPARISONS

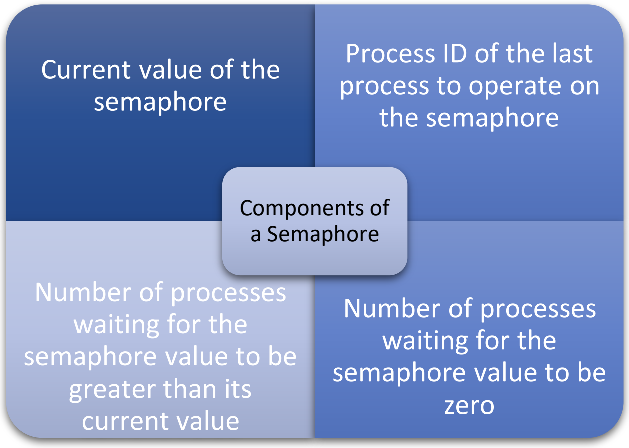
5.1 UNIX

5.1.1 CONCURRENCY MECHANISMS

UNIX has several ways for interprocessor communication and synchronisation, including:

- Pipes
    - A pipe is a circular buffer that allows two processes to interact. As a result, it's a first-in-first-out queue, with one process writing and another reading.
- Messages
    - A message is a set of bytes with a type associated to it. For processes to participate in message passing, UNIX offers the msgsnd and msgrcv system functions.
- Shared memory
    - In UNIX, shared memory is the quickest type of interprocess interaction. This is a virtual memory block that numerous programmes share.
- Semaphores
    - In UNIX System V, the semaphore system calls are a generalisation of the semWait and semSignal primitives. Figure 17 shows components of a semaphore.
- Signals
    - A signal is a software technique that notifies a process when asynchronous events occur. It is comparable to a hardware interrupt, except that it does not use priority.

Figure 17 Component of a Semaphore (UNIX)


Current value of the semaphore
Process ID of the last process to operate on the semaphore
Components of a Semaphore
Number of processes waiting for the semaphore value to be greater than its current value
Number of processes waiting for the semaphore value to be zero

Information can be transferred across processes through pipes, messages, and shared memory, while semaphores and signals are used to initiate actions by other processes (Stallings, 2018).

5.1.2 SECURITY

A unique user identification number is given to each UNIX user (user ID). A user belongs to a primary group and maybe several secondary groups, each of which has its own group ID. A file is specified as owned by a specific user and labelled with that user's ID when it is generated. It also belongs to a certain group, being either its creator's primary group or the group of its parent directory if SetGID permissions are enabled on that directory. A set of 12 protection bits is connected with each file. The file's inode contains the owner ID, group ID, and protection bits. The file owner, members of the group to which this file belongs, and all other users have read, write, and execute authority for nine of the protection bits. Access control lists are supported by several modern UNIX and UNIX-based operating systems, including FreeBSD, OpenBSD, Linux, and Solaris. (Stallings, 2018).

5.2 LINUX

5.2.1 CONCURRENCY MECHANISMS

All concurrency features that are available in UNIX systems including pipes, messages, shared memory, and signals, are included in Linux. It also supports real-time (RT) signals, which are a unique sort of signaling. Linux additionally comes with a number of concurrency methods designed expressly for usage when a thread is running in kernel mode. These are mechanisms utilised within the kernel to support concurrency during kernel code execution (Stallings, 2018).

5.2.2 SECURITY

The security paradigm of Linux is quite similar to UNIX security measures. Security concerns can be divided into two categories:

1) **Authentication**: Ensures that no one can use the system without first demonstrating that they have access rights.
    - This traditionally been accomplished by using a publicly accessible password file. The password of a user is paired with a random "salt" value, and the output is encoded using a one-way transformation function and saved in the password file.
2) **Access control**: Provided a system for determining if a user has the authority to access a certain item and limiting access to objects when necessary
    - This is accomplished via unique numeric identifiers. A user identifier (UID) is a unique number that identifies a specific user or set of access permissions. A GID is an additional identifier that can be used to identify privileges that belong to multiple users. Different objects in the system are subject to access control. (Silberschatz, Gagne, & Baer , 2005).

5.3 WINDOWS

5.3.1 CONCURRENCY MECHANISMS

As part of the object architecture, Windows enable thread synchronisation. Executive dispatcher objects, user-mode critical sections, slim reader–writer locks, condition variables, and lock-free operations are by far the most essential synchronisation approaches. Wait functions are used by dispatcher objects to permit a thread to block its execution. The family of dispatcher objects employed by the Windows Executive to provide synchronisation features includes the Notification event, Synchronization event, Mutex, Semaphore, Waitable timer, File, Process, and Thread. The first five object kinds were created with synchronisation in mind. The remaining object kinds serve a variety of purposes, including synchronisation (Stallings, 2018).

5.3.2 SECURITY

Processes, threads, files, semaphores, windows, and other objects all have a standard access control mechanism in Windows. An access token associated to every process and a security descriptor connected with every object for which interprocess access is permitted govern access control. **Security ID, Group SIDs, Privileges, Default owner,** and **Default ACL** make up the basic structure of an access token. Furthermore, Windows uses a name/password approach to authenticate users when they log on to a Windows system (Stallings, 2018).

5.4 ANDROID

5.4.1 CONCURRENCY MECHANISMS

Binder, a lightweight remote procedure call (RPC) feature that is effective in respect of both memory and processing demands and is perfectly matched to the needs of an embedded system, is added to the kernel by Android. All interactions between two processes are mediated by the Binder. A call is made by a component in one process (the client). This call is directed to the kernel's Binder, which forwards it to the destination item in the destination process (the service). The return from the destination is sent to the caller element in the calling process via the Binder (Stallings, 2018).

5.4.2 SECURITY

Android has a set of security features built into the Linux kernel. It gives the operating system a user-based permissions model, process isolation, a secure IPC method, and the option to delete any kernel components that are superfluous or possibly risky. It also functions to prevent numerous system users from accessing and draining each other's resources. Android may construct a "Application Sandbox" using this user-based protection. Each Android app is given an own user ID and runs as its own process. As a result, the Linux kernel enforces each application at the process level, preventing applications from interacting with one another and granting them only restricted access to the Android operating system. This offers the user access control based on permissions. The same is true with filesystem permissions: every application (or user) has its own files, and unless a developer specifically discloses files to another Android app, files created by one app cannot be read or modified by another. (Android Open Source Project., 2021).

6 CONCLUSION

In conclusion, multiprogramming, multiprocessing, and distributed processing are fundamental elements in current operating systems. Concurrency is crucial to these concepts, as well as the technology of OS architecture. When numerous processes are running at the same time, whether physically in a multiprocessor system or virtually in a single-processor multiprogramming system, conflict and cooperation difficulties develop.

Concurrent processes can work together in a variety of ways. Even if processes are ignorant of one another, they may compete for resources like CPU time or access to I/O devices. Since they share accessibility to a shared object, like a block of main memory or a file, processes could be indirectly aware of each other. Furthermore, processes can be directly knowledgeable of one another and cooperate via information exchange. Mutual exclusion and deadlock are two major challenges that develop in these communications.

Mitigating threat from intruders and malicious software is the most pressing challenge for OS security. Intruders try to get unauthorised access to system resources, while malicious software is developed to get past system defences and run on target computers. Intrusion detection systems, authentication methods, access control mechanisms, and firewalls are all defenses for both sorts of threats.

7 REFERENCES

Corbet, J., Kroah-Hartman, G., & Rubini, A. (2005). *Linux Device Drivers, 3rd Edition.* O'Reilly.

Operating system design/concurrent. (n.d.). Retrieved May, 2022 from Wikibooks, open books for an open world: https://en,wikibooks.org/wiki/Operating_System_Design/Concurrent

Stallings, W. (2018). *Operating Systems: Internals and Design Principles.* Pearson-Prentice Hall.

Semaphores in Operating System. (n.d.). From Tutorialspoint: https://www.tutorialspoint.com/semaphores-in-operating-system#:~:text=Semaphores%20are%20integer%20variables%20that,are%20used%20for%20process%20synchronization.&text=The%20wait%20operation%20decrements%20the,S%2C%20if%20it%20is%20positive.

Singh, J. P. (n.d.). *COS 318: Operating Systems Message Passing*. From Princeton: https://www.cs.princeton.edu/courses/archive/fall16/cos318/lectures/11.MessagePassing.pdf

Yadav, B. (2022, March 2). *Process Synchronization in OS*. From Scaler: https://www.scaler.com/topics/operating-system/process-synchronization-in-os/

Eskicioglu, & Marsland. (2001). *Deadlcok*. From ualberta: https://webdocs.cs.ualberta.ca/~tony/C379/Notes/PDF/06.4.pdf

Howard, J. (1973). *“Mixed Solutions for the Deadlock Problem.”* . Communications of the ACM.

Handling Deadlocks. (2022, May 5). From geeksforgeeks: https://www.geeksforgeeks.org/handling-deadlocks/

Silberschatz, A., Gagne, G., & Baer , P. G. (2005). *"Operating System Concepts, Seventh Edition ".* Johm Willey & Sons.

James , P., & Anderson , C. (1980). *Computer Security Threat Monitoring and Surveillance.* Fort Washington.

Guide to General Server Security. (2008). National Institute of Standards and Technology.

Android Open Source Project. (2021, September 27). From Security Overview. : https://source.android.com/security

Holt, R. (1972.). *Some Deadlock Properties of Computer Systems.* Computing Surveys.

Concurrent Processes in Operating System. (28, April 2020). From GeeksforGeeks: https://www.geeksforgeeks.org/concurrent-processes-in-operating-system/

Multi-threading and the Operating System. (2017). From Microfocus: https://www.microfocus.com/documentation/visual-cobol/vc40pu8/VS2017/BKMTMTINTRS001.html