**Memory Management**

Main Memory refers to a physical memory that is the internal memory to the computer. The word main is used to distinguish it from external mass storage devices such as disk drives. Main memory is also known as RAM. The computer is able to change only data that is in main memory. Therefore, every program we execute and every file we access must be copied from a storage device into main memory.

All the programs are loaded in the main memeory for execution. Sometimes complete program is loaded into the memory, but some times a certain part or routine of the program is loaded into the main memory only when it is called by the program, this mechanism is called **Dynamic Loading**, this enhance the performance.

Also, at times one program is dependent on some other program. In such a case, rather than loading all the dependent programs, CPU links the dependent programs to the main executing program when its required. This mechanism is known as **Dynamic Linking**.

**Swapping**

A process needs to be in memory for execution. But sometimes there is not enough main memory to hold all the currently active processes in a timesharing system. So, excess process are kept on disk and brought in to run dynamically. Swapping is the process of bringing in each process in main memory, running it for a while and then putting it back to the disk.

**Contiguous Memory Allocation**

In contiguous memory allocation each process is contained in a single contiguous block of memory. Memory is divided into several fixed size partitions. Each partition contains exactly one process. When a partition is free, a process is selected from the input queue and loaded into it. The free blocks of memory are known as *holes*. The set of holes is searched to determine which hole is best to allocate.

**Memory Protection**

Memory protection is a phenomenon by which we control memory access rights on a computer. The main aim of it is to prevent a process from accessing memory that has not been allocated to it. Hence prevents a bug within a process from affecting other processes, or the operating system itself, and instead results in a segmentation fault or storage violation exception being sent to the disturbing process, generally killing of process.

**Memory Allocation**

Memory allocation is a process by which computer programs are assigned memory or space. It is of three types :

1. **First Fit**

The first hole that is big enough is allocated to program.

1. **Best Fit**

The smallest hole that is big enough is allocated to program.

1. **Worst Fit**

The largest hole that is big enough is allocated to program.

**Fragmentation**

Fragmentation occurs in a dynamic memory allocation system when most of the free blocks are too small to satisfy any request. It is generally termed as inability to use the available memory.

In such situation processes are loaded and removed from the memory. As a result of this, free holes exists to satisfy a request but is non contiguous i.e. the memory is fragmented into large no. Of small holes. This phenomenon is known as **External Fragmentation.**

Also, at times the physical memory is broken into fixed size blocks and memory is allocated in unit of block sizes. The memory allocated to a space may be slightly larger than the requested memory. The difference between allocated and required memory is known as **Internal fragmentation** i.e. the memory that is internal to a partition but is of no use.

**Paging**

A solution to fragmentation problem is Paging. Paging is a memory management mechanism that allows the physical address space of a process to be non-contagious. Here physical memory is divided into blocks of equal size called **Pages**. The pages belonging to a certain process are loaded into available memory frames.

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous.

* Logical Address or Virtual Address (represented in bits): An address generated by the CPU
* Logical Address Space or Virtual Address Space( represented in words or bytes): The set of all logical addresses generated by a program
* Physical Address (represented in bits): An address actually available on memory unit
* Physical Address Space (represented in words or bytes): The set of all physical addresses corresponding to the logical addresses

The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.

* The Physical Address Space is conceptually divided into a number of fixed-size blocks, called **frames**.
* The Logical address Space is also splitted into fixed-size blocks, called **pages**.
* Page Size = Frame Size

Let us consider an example:

* Physical Address = 12 bits, then Physical Address Space = 4 K words
* Logical Address = 13 bits, then Logical Address Space = 8 K words
* Page size = frame size = 1 K words (assumption)

Address generated by CPU is divided into

* **Page number(p):** Number of bits required to represent the pages in Logical Address Space or Page number
* **Page offset(d):** Number of bits required to represent particular word in a page or page size of Logical Address Space or word number of a page or page offset.

Physical Address is divided into

* **Frame number(f):** Number of bits required to represent the frame of Physical Address Space or Frame number.
* **Frame offset(d):** Number of bits required to represent particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.

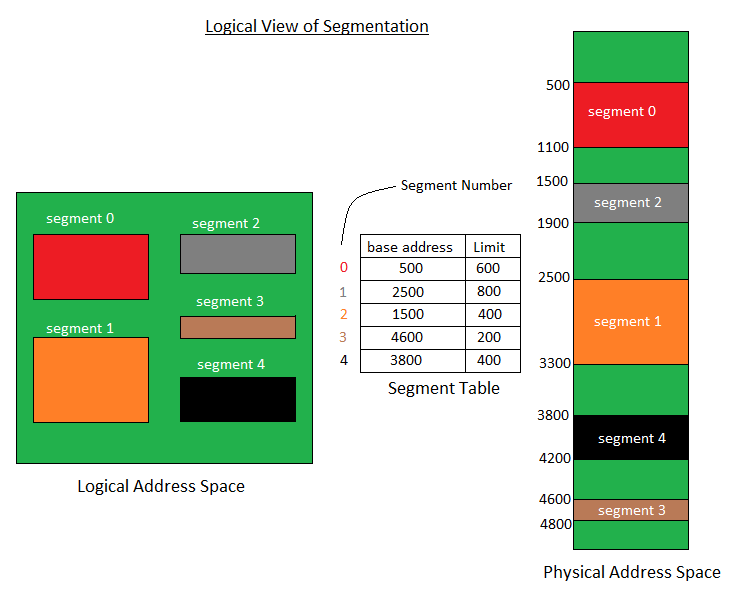
**Page Table**

A Page Table is the data structure used by a virtual memory system in a computer operating system to store the mapping between *virtual address* and *physical addresses.*

Virtual address is also known as Logical address and is generated by the CPU. While Physical address is the address that actually exists on memory.

**Segmentation**

Segmentation is another memory management scheme that supports the user-view of memory. Segmentation allows breaking of the virtual address space of a single process into segments that may be placed in non-contiguous areas of physical memory.



**Segmentation with Paging**

Both paging and segmentation have their advantages and disadvantages, it is better to combine these two schemes to improve on each. The combined scheme is known as 'Page the Elements'. Each segment in this scheme is divided into pages and each segment is maintained in a page table. So the logical address is divided into following 3 parts :

* Segment numbers(S)
* Page number (P)
* The displacement or offset number (D)

**Virtual Memory**

Virtual Memory is a space where large programs can store themselves in form of pages while their execution and only the required pages or portions of processes are loaded into the main memory. This technique is useful as large virtual memory is provided for user programs when a very small physical memory is there.

In real scenarios, most processes never need all their pages at once, for following reasons :

* Error handling code is not needed unless that specific error occurs, some of which are quite rare.
* Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
* Certain features of certain programs are rarely used.

**Benefits of having Virtual Memory :**

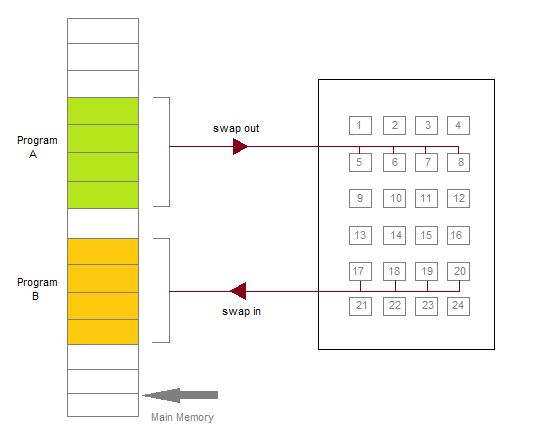
1. Large programs can be written, as virtual space available is huge compared to physical memory.
2. Less I/O required, leads to faster and easy swapping of processes.
3. More physical memory available, as programs are stored on virtual memory, so they occupy very less space on actual physical memory.

**Comparison Chart:** 

| Parameter | LOGICAL ADDRESS | PHYSICAL ADDRESS |
| --- | --- | --- |
| Basic | generated by CPU | location in a memory unit |
| Address Space | Logical Address Space is set of all logical addresses generated by CPU in reference to a program. | Physical Address is set of all physical addresses mapped to the corresponding logical addresses. |
| Visibility | User can view the logical address of a program. | User can never view physical address of program. |
| Generation | generated by the CPU | Computed by MMU |
| Access | The user can use the logical address to access the physical address. | The user can indirectly access physical address but not directly. |
| Editable | Logical address can be change. | Physical address will not change. |
| Also called | virtual address. | real address. |

**Demand Paging**

The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them(On demand). This is termed as lazy swapper, although a pager is a more accurate term.



Initially only those pages are loaded which will be required the process immediately.

The pages that are not moved into the memory, are marked as invalid in the page table. For an invalid entry the rest of the table is empty. In case of pages that are loaded in the memory, they are marked as valid along with the information about where to find the swapped out page.

When the process requires any of the page that is not loaded into the memory, a page fault trap is triggered and following steps are followed,

1. The memory address which is requested by the process is first checked, to verify the request made by the process.
2. If its found to be invalid, the process is terminated.
3. In case the request by the process is valid, a free frame is located, possibly from a free-frame list, where the required page will be moved.
4. A new operation is scheduled to move the necessary page from disk to the specified memory location. ( This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime. )
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to valid.
6. The instruction that caused the page fault must now be restarted from the beginning.

There are cases when no pages are loaded into the memory initially, pages are only loaded when demanded by the process by generating page faults. This is called **Pure Demand Paging**.

The only major issue with Demand Paging is, after a new page is loaded, the process starts execution from the beginning. Its is not a big issue for small programs, but for larger programs it affects performance drastically.

**Page Replacement**

As studied in Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more number of processes into the memory at the same time. but what happens when a process requests for more pages and no free memory is available to bring them in. Following steps can be taken to deal with this problem :

1. Put the process in the wait queue, until any other process finishes its execution thereby freeing frames.
2. Or, remove some other process completely from the memory to free frames.
3. Or, find some pages that are not being used right now, move them to the disk to get free frames. This technique is called **Page replacement** and is most commonly used. We have some great algorithms to carry on page replacement efficiently.

**Basic Page Replacement**

* Find the location of the page requested by ongoing process on the disk.
* Find a free frame. If there is a free frame, use it. If there is no free frame, use a page-replacement algorithm to select any existing frame to be replaced, such frame is known as **victim frame**.
* Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
* Move the required page and store it in the frame. Adjust all related page and frame tables to indicate the change.
* Restart the process that was waiting for this page.

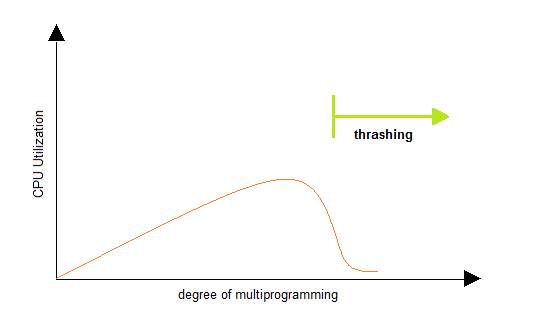
**FIFO Page Replacement**

* A very simple way of Page replacement is FIFO (First in First Out)
* As new pages are requested and are swapped in, they are added to tail of a queue and the page which is at the head becomes the victim.
* Its not an effective way of page replacement but can be used for small systems.

#### Thrashing

A process that is spending more time paging than executing is said to be thrashing. In other words it means, that the process doesn't have enough frames to hold all the pages for its execution, so it is swapping pages in and out very frequently to keep executing. Sometimes, the pages which will be required in the near future have to be swapped out.

Initially when the CPU utilization is low, the process scheduling mechanism, to increase the level of multiprogramming loads multiple processes into the memory at the same time, allocating a limited amount of frames to each process. As the memory fills up, process starts to spend a lot of time for the required pages to be swapped in, again leading to low CPU utilization because most of the proccesses are waiting for pages. Hence the scheduler loads more processes to increase CPU utilization, as this continues at a point of time the complete system comes to a stop.



To prevent thrashing we must provide processes with as many frames as they really need "right now".

**file System**

A file can be "free formed", indexed or structured collection of related bytes having meaning only to the one who created it. Or in other words an entry in a directory is the file. The file may have attributes like name, creator, date, type, permissions etc.

**File Structure**

A file has various kinds of structure. Some of them can be :

* **Simple Record Structure** with lines of fixed or variable lengths.
* **Complex Structures** like formatted document or reloadable load files.
* **No Definite Structure** like sequence of words and bytes etc.

**Attributes of a File**

Following are some of the attributes of a file :

* **Name**. It is the only information which is in human-readable form.
* **Identifier**. The file is identified by a unique tag(number) within file system.
* **Type**. It is needed for systems that support different types of files.
* **Location**. Pointer to file location on device.
* **Size**. The current size of the file.
* **Protection**. This controls and assigns the power of reading, writing, executing.
* **Time, date, and user identification**. This is the data for protection, security, and usage monitoring.

**File Access Methods**

The way that files are accessed and read into memory is determined by Access methods. Usually a single access method is supported by systems while there are OS's that support multiple access methods.

**Sequential Access**

* Data is accessed one record right after another is an order.
* Read command cause a pointer to be moved ahead by one.
* Write command allocate space for the record and move the pointer to the new End Of File.
* Such a method is reasonable for tape.

**Direct Access**

* This method is useful for disks.
* The file is viewed as a numbered sequence of blocks or records.
* There are no restrictions on which blocks are read/written, it can be dobe in any order.
* User now says "read n" rather than "read next".
* "n" is a number relative to the beginning of file, not relative to an absolute physical disk location.

**Indexed Sequential Access**

* It is built on top of Sequential access.
* It uses an Index to control the pointer while accessing files.

**What is a Directory?**

Information about files is maintained by Directories. A directory can contain multiple files. It can even have directories inside of them. In Windows we also call these directories as folders.

Following is the information maintained in a directory :

* **Name** : The name visible to user.
* **Type** : Type of the directory.
* **Location** : Device and location on the device where the file header is located.
* **Size** : Number of bytes/words/blocks in the file.
* **Position** : Current next-read/next-write pointers.
* **Protection** : Access control on read/write/execute/delete.
* **Usage** : Time of creation, access, modification etc.
* **Mounting** : When the root of one file system is "grafted" into the existing tree of another file system its called Mounting.

**Banker's Algorithm**

Banker's algorithm is a deadlock avoidance algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Everytime a loan has to be granted by the bank, it subtracts the loan amount from the total money the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Banker's algorithm works in a similar way in computers. Whenever a new process is created, it must exactly specify the maximum instances of each resource type that it needs.

Let us assume that there are **n** processes and **m** resource types. Some data structures are used to implement the banker's algorithm. They are:

* Available: It is an array of length **m**. It represents the number of available resources of each type. If Available[j] = k, then there are **k** instances available, of resource type **Rj**.
* Max: It is an **n x m** matrix which represents the maximum number of instances of each resource that a process can request. If Max[i][j] = k, then the process **Pi** can request atmost **k** instances of resource type **Rj**.
* Allocation: It is an **n x m** matrix which represents the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process **Pi** is currently allocated **k** instances of resource type **Rj**.
* Need: It is an **n x m** matrix which indicates the remaining resource needs of each process. If Need[i][j] = k, then process **Pi** may need **k** more instances of resource type **Rj** to complete its task.

Need[i][j] = Max[i][j] - Allocation [i][j]

**Resource Request Algorithm:**

This describes the behavior of the system when a process makes a resource request in the form of a request matrix. The steps are:

1. If number of requested instances of each resource is less than the need (which was declared previously by the process), go to step 2.
2. If number of requested instances of each resource type is less than the available resources of each type, go to step 3. If not, the process has to wait because sufficient resources are not available yet.
3. Now, assume that the resources have been allocated. Accordingly do,

Available = Available - Requesti

Allocationi = Allocationi + Requesti

Needi = Needi - Requesti

This step is done because the system needs to assume that resources have been allocated. So there will be less resources available after allocation. The number of allocated instances will increase. The need of the resources by the process will reduce. That's what is represented by the above three operations.

After completing the above three steps, check if the system is in safe state by applying the safety algorithm. If it is in safe state, proceed to allocate the requested resources. Else, the process has to wait longer.

**Safety Algorithm:**

1. Let Work and Finish be vectors of length **m** and **n**, respectively. Initially,
2. Work = Available
3. Finish[i] =false for i = 0, 1, ... , n - 1.

This means, initially, no process has finished and the number of available resources is represented by the **Available** array.

1. Find an index **i** such that both
2. Finish[i] ==false
3. Needi <= Work

If there is no such i present, then proceed to step 4.

It means, we need to find an unfinished process whose need can be satisfied by the available resources. If no such process exists, just go to step 4.

1. Perform the following:
2. Work = Work + Allocation;
3. Finish[i] = true;

Go to step 2.

When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.

1. If Finish[i] == true for all i, then the system is in a safe state.

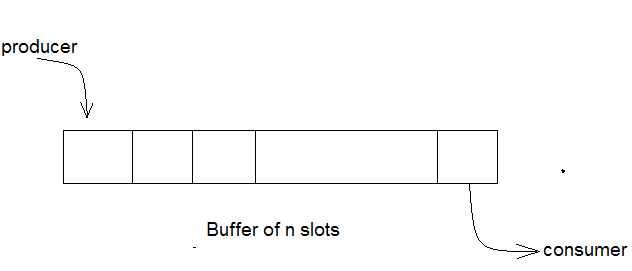
That means if all processes are finished, then the system is in safe state.

**Bounded Buffer Problem**

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization.

**Problem Statement:**

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



**Bounded Buffer Problem**

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

**Solution:**

One solution of this problem is to use semaphores. The semaphores which will be used here are:

* **m**, a binary semaphore which is used to acquire and release the lock.
* **empty**, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
* **full**, a counting semaphore whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

**Producer Operation:**

The pseudocode of the producer function looks like this:

do {

wait(empty); // wait until empty>0 and then decrement ‘empty’

wait(mutex); // acquire lock

/\* perform the insert operation in a slot \*/

signal(mutex); // release lock

signal(full); // increment ‘full’

} while(TRUE)

* Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.
* Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
* Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
* After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

**Consumer Operation:**

The pseudocode of the consumer function looks like this:

do {

wait(full); // wait until full>0 and then decrement ‘full’

wait(mutex); // acquire the lock

/\* perform the remove operation

in a slot \*/

signal(mutex); // release the lock

signal(empty); // increment ‘empty’

} while(TRUE);

* The consumer waits until there is atleast one full slot in the buffer.
* Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
* After that, the consumer acquires lock on the buffer.
* Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
* Then, the consumer releases the lock.
* Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

**Comparison of Scheduling Algorithms**

By now, you must have understood how CPU can apply different scheduling algorithms to schedule processes. Now, let us examine the advantages and disadvantages of each scheduling algorithm.

**First Come First Serve (FCFS)**

**Advantages:**

* FCFS algorithm doesn't include any complex logic, it just puts the process requests in a queue and executes it one by one.
* Hence, FCFS is pretty simple and easy to implement.
* Eventually, every process will get a chance to run, so starvation doesn't occur.

**Disadvantages:**

* There is no option for pre-emption of a process. If a process is started, then CPU executes the process until it ends.
* Because there is no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed.

**Shortest Job First (SJF)**

**Advantages:**

* According to the definition, short processes are executed first and then followed by longer processes.
* The throughput is increased because more processes can be executed in less amount of time.

**Disadvantages:**

* The time taken by a process must be known by the CPU beforehand, which is not possible.
* Longer processes will have more waiting time, eventually they'll suffer starvation.

**Note:** Preemptive Shortest Job First scheduling will have the same advantages and disadvantages as those for SJF.

**Round Robin (RR)**

**Advantages:**

* Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority.
* Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind.

**Disadvantages:**

* The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behavior as FCFS.
* If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

**Priority based Scheduling**

**Advantages:**

* The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high end game will have better graphics, that means the process which updates the screen in a game will have higher priority so as to achieve better graphics performance.

**Disadvantages:**

* A second scheduling algorithm is required to schedule the processes which have same priority.
* In preemptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

**Usage of Scheduling Algorithms in Different Situations:**

Every scheduling algorithm has a type of a situation where it is the best choice. Let's look at different such situations:

**Situation 1:**

The incoming processes are short and there is no need for the processes to execute in a specific order.

In this case, FCFS works best when compared to SJF and RR because the processes are short which means that no process will wait for a longer time. When each process is executed one by one, every process will be executed eventually.

**Situation 2:**

The processes are a mix of long and short processes and the task will only be completed if all the processes are executed successfully in a given time.

Round Robin scheduling works efficiently here because it does not cause starvation and also gives equal time quantum for each process.

**Situation 3:**

The processes are a mix of user based and kernel based processes.

Priority based scheduling works efficiently in this case because generally kernel based processes have higher priority when compared to user based processes.

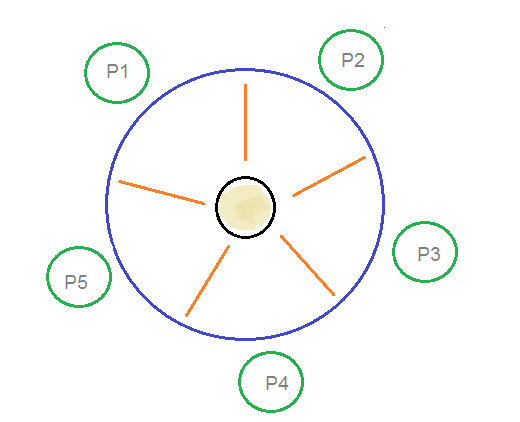
For example, the scheduler itself is a kernel based process, it should run first so that it can schedule other processes.

**Dining Philosophers Problem**

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

**Problem Statement:**

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



**Dining Philosophers Problem**

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

**Solution:**

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, **stick[5]**, for each of the five chopsticks.

The code for each philosopher looks like:

while(TRUE) {

wait(stick[i]);

wait(stick[(i+1) % 5]); // mod is used because if i=5, next

// chopstick is 1 (dining table is circular)

/\* eat \*/

signal(stick[i]);

signal(stick[(i+1) % 5]);

/\* think \*/

}

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

* A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
* Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

**Readers Writer Problem**

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

**Problem Statement:**

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer**is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource.

**Solution:**

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one mutex **m** and a semaphore **w**. An integer variable **read\_count** is used to maintain the number of readers currently accessing the resource. The variable **read\_count** is initialized to 0. A value of 1 is given initially to **m** and **w**.

Instead of having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating the **read\_count** variable.

The code for the writer process looks like this:

while(TRUE) {

wait(w);

/\*perform the

write operation \*/

signal(w);

}

The code for the reader process looks like this:

while(TRUE) {

wait(m); //acquire lock

read\_count++;

if(read\_count == 1)

wait(w);

signal(m); //release lock

/\* perform the

reading operation \*/

wait(m); // acquire lock

read\_count--;

if(read\_count == 0)

signal(w);

signal(m); // release lock

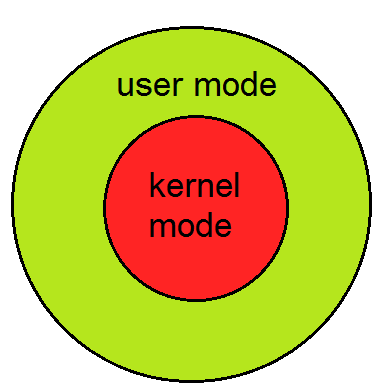
}

**Code Explained:**

* As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
* After performing the write operation, it increments **w** so that the next writer can access the resource.
* On the other hand, in the code for the reader, the lock is acquired whenever the **read\_count** is updated by a process.
* When a reader wants to access the resource, first it increments the **read\_count** value, then accesses the resource and then decrements the **read\_count** value.
* The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
* The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
* Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

**System Calls**

To understand system calls, first one needs to understand the difference between **kernel mode** and **user mode** of a CPU. Every modern operating system supports these two modes.



**Modes supported by the operating system**

**Kernel Mode**

* When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
* Hence kernel mode is a very privileged and powerful mode.
* If a program crashes in kernel mode, the entire system will be halted.

**User Mode**

* When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
* In user mode, if any program crashes, only that particular program is halted.
* That means the system will be in a safe state even if a program in user mode crashes.
* Hence, most programs in an OS run in user mode.

**System Call**

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.

Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

* Creating, opening, closing and deleting files in the file system.
* Creating and managing new processes.
* Creating a connection in the network, sending and receiving packets.
* Requesting access to a hardware device, like a mouse or a printer.

In a typical UNIX system, there are around 300 system calls. Some of them which are important ones in this context, are described below.

**Fork()**

The fork() system call is used to create processes. When a process (a program in execution) makes a fork() call, an exact copy of the process is created. Now there are two processes, one being the **parent** process and the other being the **child** process.

The process which called the fork() call is the **parent** process and the process which is created newly is called the **child** process. The child process will be exactly the same as the parent. Note that the process state of the parent i.e., the address space, variables, open files etc. is copied into the child process. This means that the parent and child processes have identical but physically different address spaces. The change of values in parent process doesn't affect the child and vice versa is true too.

Both processes start execution from the next line of code i.e., the line after the fork() call. Let's look at an example:

//example.c

#include <stdio.h>

void main() {

int val;

val = fork(); // line A

printf("%d",val); // line B

}

When the above example code is executed, when **line A** is executed, a child process is created. Now both processes start execution from **line B**. To differentiate between the child process and the parent process, we need to look at the value returned by the fork() call.

The difference is that, in the parent process, fork() returns a value which represents the **process ID** of the child process. But in the child process, fork() returns the value 0.

This means that according to the above program, the output of parent process will be the **process ID**of the child process and the output of the child process will be 0.

**Exec()**

The exec() system call is also used to create processes. But there is one big difference between fork() and exec() calls. The fork() call creates a new process while preserving the parent process. But, an exec() call replaces the address space, text segment, data segment etc. of the current process with the new process.

It means, after an exec() call, only the new process exists. The process which made the system call, wouldn't exist.

There are many flavors of exec() in UNIX, one being exec1() which is shown below as an example:

//example2.c

#include

void main() {

execl("/bin/ls", "ls", 0); // line A

printf("This text won't be printed unless an error occurs in exec().");

}

As shown above, the first parameter to the execl() function is the address of the program which needs to be executed, in this case, the address of the **ls** utility in UNIX. Then it is followed by the name of the program which is **ls** in this case and followed by optional arguments. Then the list should be terminated by a NULL pointer (0).

When the above example is executed, at line A, the **ls** program is called and executed and the current process is halted. Hence the printf() function is never called since the process has already been halted. The only exception to this is that, if the **execl()** function causes an error, then the printf()function is executed.