Module 4:

Recursion and Storage Management

CO4: (6 hrs), AT2

Subject In-charge

Ms. Pratibha Rane

Refer: Data structure using C by A. Tenenbaum

What is a recursive function?

Refer: Data structure using C by A. Tenenbaum

A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself

Refer 7.7.4: Reema Thareja

A recursive function makes use of the system ___CALL STACK___ to temporarily store the return address and local variables of the calling function

Every recursive solution has two major cases. They are

1. Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.
2. Recursive case, in which **first** the problem at hand is divided into simpler sub-parts. **Second** the function calls itself but with sub-parts of the problem obtained in the first step. **Third**, the result is obtained by combining the solutions of simpler sub-parts.

- Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems.

- In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.

Calculate factorial of n i.e n!

In other words, `n! = n × (n-1)!`

Let us say we need to find the value of `5!`

`5! = 5 × 4 × 3 × 2 × 1`

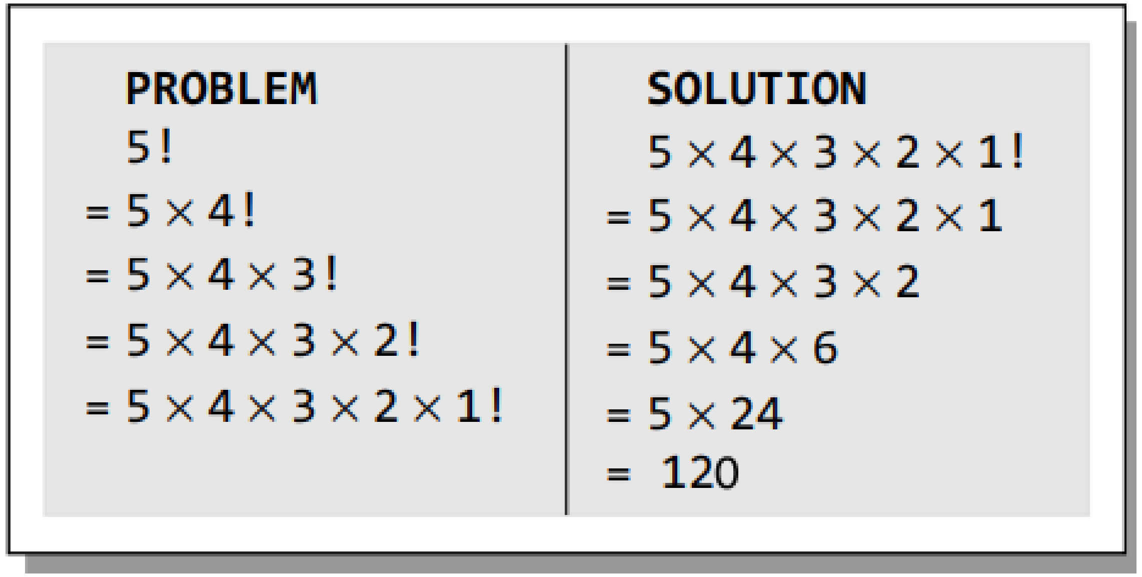
`= 120`

This can be written as

`5! = 5 × 4!, where 4!= 4 × 3!`

Therefore,

`5! = 5 × 4 × 3!`



PROBLEM	SOLUTION
5!	5 × 4 × 3 × 2 × 1!
= 5 × 4!	= 5 × 4 × 3 × 2 × 1
= 5 × 4 × 3!	= 5 × 4 × 3 × 2
= 5 × 4 × 3 × 2!	= 5 × 4 × 6
= 5 × 4 × 3 × 2 × 1!	= 5 × 24
	= 120

Figure 7.27 Recursive factorial function

Calculate factorial of n i.e n!

Base case is when n = 1, because if n = 1, the result will be 1 as 1! = 1.

Recursive case of the factorial function will call itself but with a smaller value of n, this case can be given as factorial(n) = n × factorial (n–1)

Fibonacci Series

The Fibonacci Series The Fibonacci series can be given as 0 1 1 2 3 5 8 13 21 34 55 ……

$$\text{FIB}(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ \text{FIB}(n-1) + \text{FIB}(n-2), & \text{otherwise} \end{cases}$$

Fibonacci Series

The Fibonacci Series The Fibonacci series can be given as 0 1 1 2 3 5 8 13 21 34 55 ……

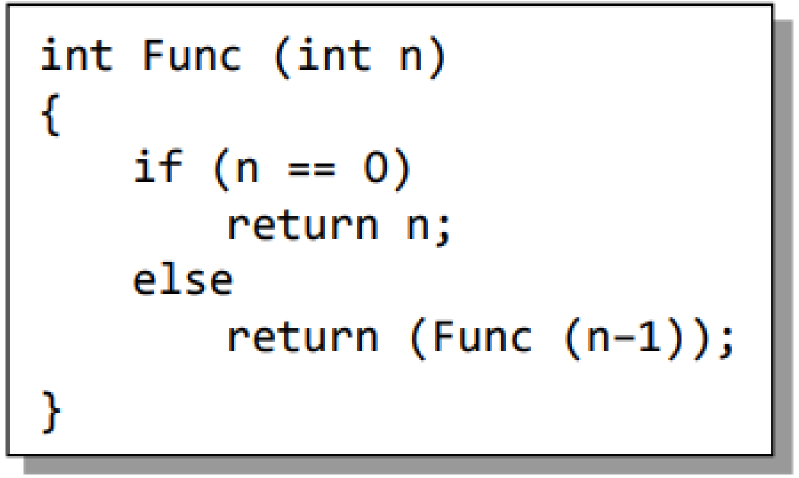
$$\text{FIB}(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ \text{FIB}(n-1) + \text{FIB}(n-2), & \text{otherwise} \end{cases}$$

Types of Recursions:

- whether the function calls itself directly or indirectly (direct or indirect recursion),
- whether any operation is pending at each recursive call (tail recursive or not), and
- the structure of the calling pattern (linear or tree-recursive).

Direct Recursion

A function is said to be directly recursive if it explicitly calls itself. For example, consider the code shown in Fig. 7.28. Here, the function Func() calls itself for all positive values of n, so it is said to be a directly recursive function.



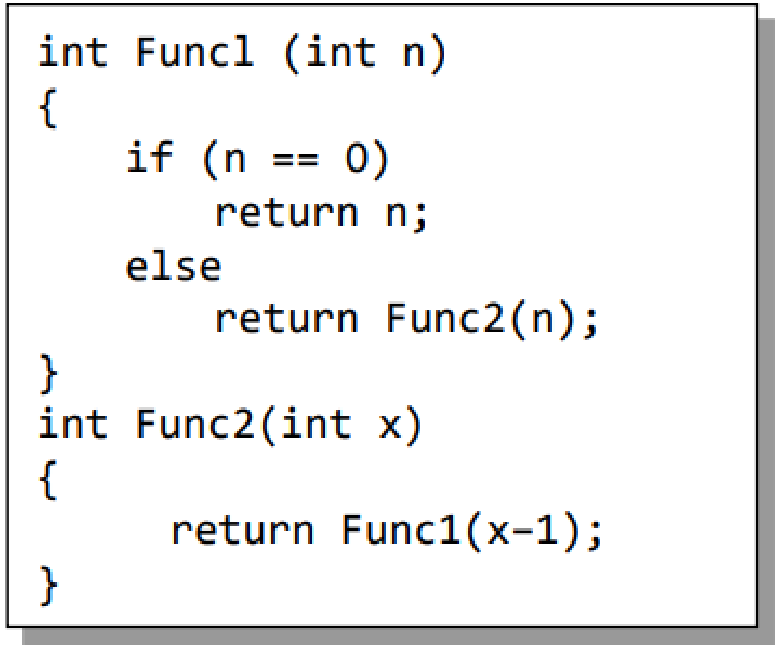
```
int Func (int n)
{
    if (n == 0)
        return n;
    else
        return (Func (n-1));
}
```

Figure 7.28 Direct recursion

Refer 7.7.4: Reema Thareja

Indirect Recursion

A function is said to be indirectly recursive if it contains a call to another function which ultimately calls it. Look at the functions given below. These two functions are indirectly recursive as they both call each other (Fig. 7.29).



```
int Funcl (int n)
{
    if (n == 0)
        return n;
    else
        return Func2(n);
}
int Func2(int x)
{
      return Func1(x-1);
}
```

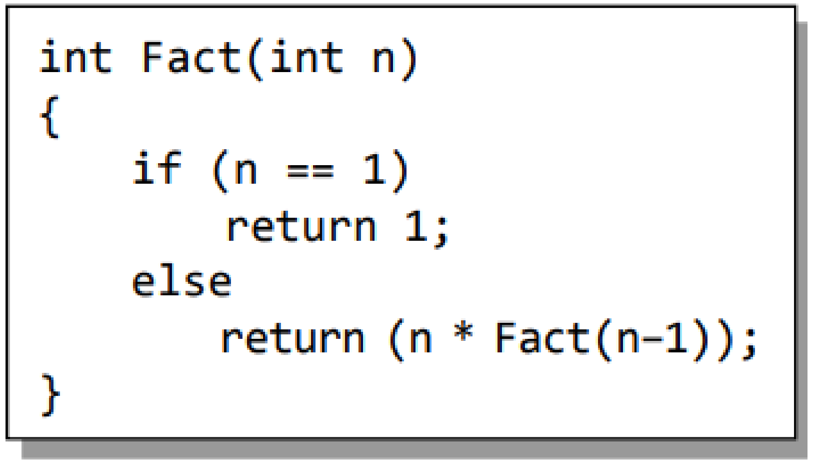
Figure 7.29 Indirect recursion

Refer 7.7.4: Reema Thareja

Non- Tail Recursion

There is a pending operation of multiplication to be performed on return from each recursive call.

Whenever there is a pending operation to be performed, the function becomes non-tail recursive

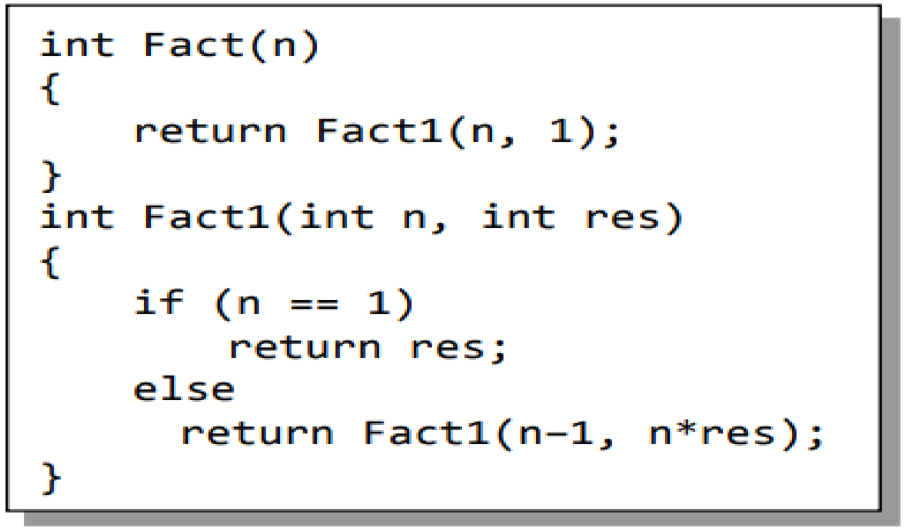


```
int Fact(int n)
{
    if (n == 1)
        return 1;
    else
        return (n * Fact(n-1));
}
```

Figure 7.30 Non-tail recursion

Tail Recursion

- **tail recursive if no operations are pending to be performed when the recursive function returns to its caller**
- when the called function returns, the returned value is immediately returned from the calling function.



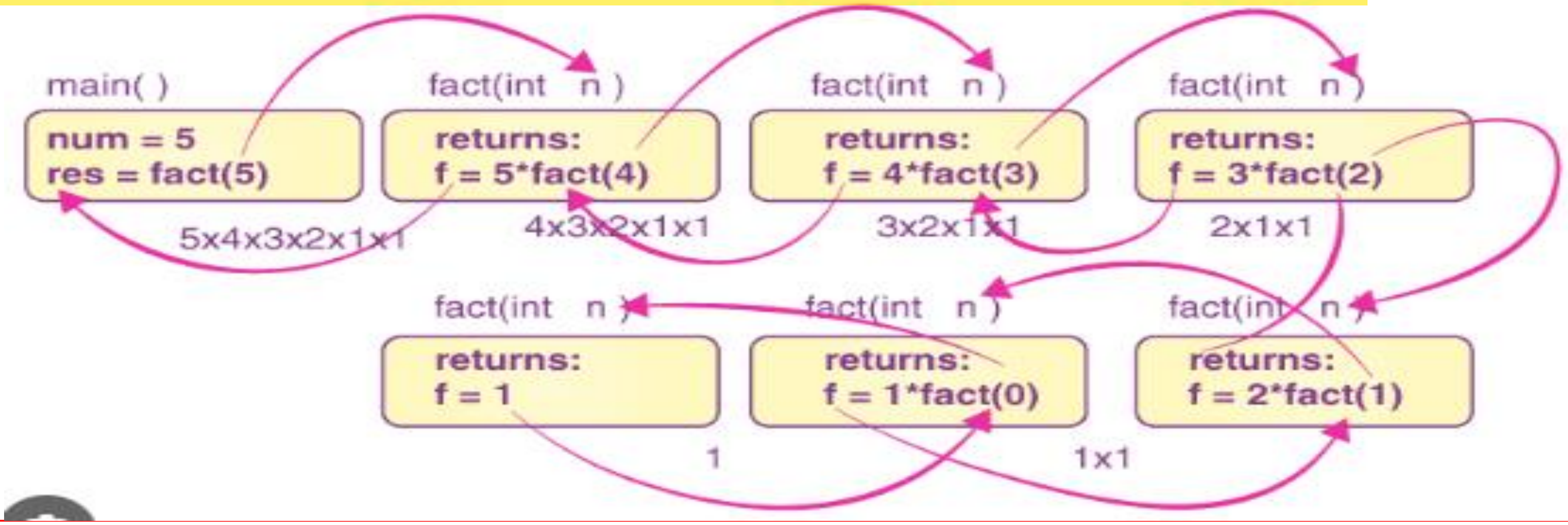
```
int Fact(n)
{
    return Fact1(n, 1);
}
int Fact1(int n, int res)
{
    if (n == 1)
        return res;
    else
      return Fact1(n-1, n*res);
}
```

Figure 7.31 Tail recursion

Winding and un-winding phase in Recursion:

1. In Winding phase, the recursive function keeps calling itself. This phase ends when base condition is reached

1. unwinding phase occurs when base condition is reached. The control goes back to the calling function till it reaches to the original call.



Advantages of Recursion

1. Helps to reduce size of program by minimizing the code
2. Easy to maintain function calling related information
3. Evaluation of stack can be implemented through recursion
4. Infix, prefix and postfix notations can be evaluated with recursion

Disadvantages of Recursion

1. Takes more time because of stack
2. Stack overflow may occur
3. Memory requirement is more
4. Lead to infinite call if exit condition does not work
5. Efficiency is less

Difference between Recursive and iterative programming

Parameters	**Recursive programming**	**Iterative programming**
Requirement space	Recursive program has greater space requirement	Iterative programming consumes less space
Speed	It has greater time requirement for operation because of function calls and return overhead	They operate faster than recursive functions
Solving techniques	Recursive functions can be solved by iteratively	Iterative functions can be solved recursively
Examples	recursive program include tower of Hanoi problem	Simple iterative program include finding sum of first n numbers

Storage Management:

- To make the computer suitable for users, the operating system offers a **constant logical view of information**.
- The operating system abstracts from the physical properties of its storage devices to describe a logical storage unit , the file.
- The **OS is responsible** for mapping files which are located on physical media and accessing these files with the help of storage devices

Refer 7.7.4: Reema Thareja

Storage Management:

The heap is the region of main memory from which portions of memory are dynamically allocated upon request of a program

The memory manager of MMU is responsible for:

-the maintenance of free memory blocks

-Assigning specific memory blocks to the user programs

-cleaning memory from unneeded blocks to return them to the memory pool.

scheduling access to shared data,

-Moving code and data between main and secondary memory

-keeping one process away from another

Refer 7.7.4: Reema Thareja

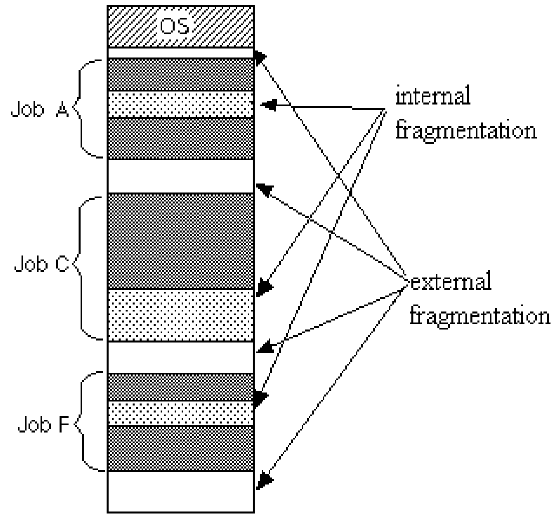
Fragmentation: In **contiguous memory allocation** whenever the processes come into RAM, space is allocated to them. These spaces in RAM are divided either on the basis of **fixed partitioning** (the size of partitions are fixed before the process gets loaded into RAM) or **dynamic partitioning** (the size of the partition is decided at the run time according to the size of the process). As the process gets loaded and removed from the memory these spaces get broken into small pieces of memory that it can't be allocated to the coming processes. This problem is called **fragmentation** .

- **External fragmentation**

although we have total space available that is needed by a process still we are not able to put that process in the memory because that space is not contiguous. This is called external fragmentation.

- **Internal fragmentation** :

The process is allocated a memory block of size more than the size of that process. Due to this some part of the memory is left unused and this cause internal fragmentation.



Sequential Fit methods :

All available memory blocks are linked and the list is searched to find a block whose size is larger than or the same as the requested size

First-fit algorithm : allocates the first block of the memory large enough to meet the request

Best-fit algorithm : allocates a block that is closest in size to the request

Refer 7.7.4: Reema Thareja

Sequential Fit methods :

All available memory blocks are linked and the list is searched to find a block whose size is larger than or the same as the requested size

- **Worst fit algorithm** : finds the largest block on the list so that the remaining part is large enough to be used in later request

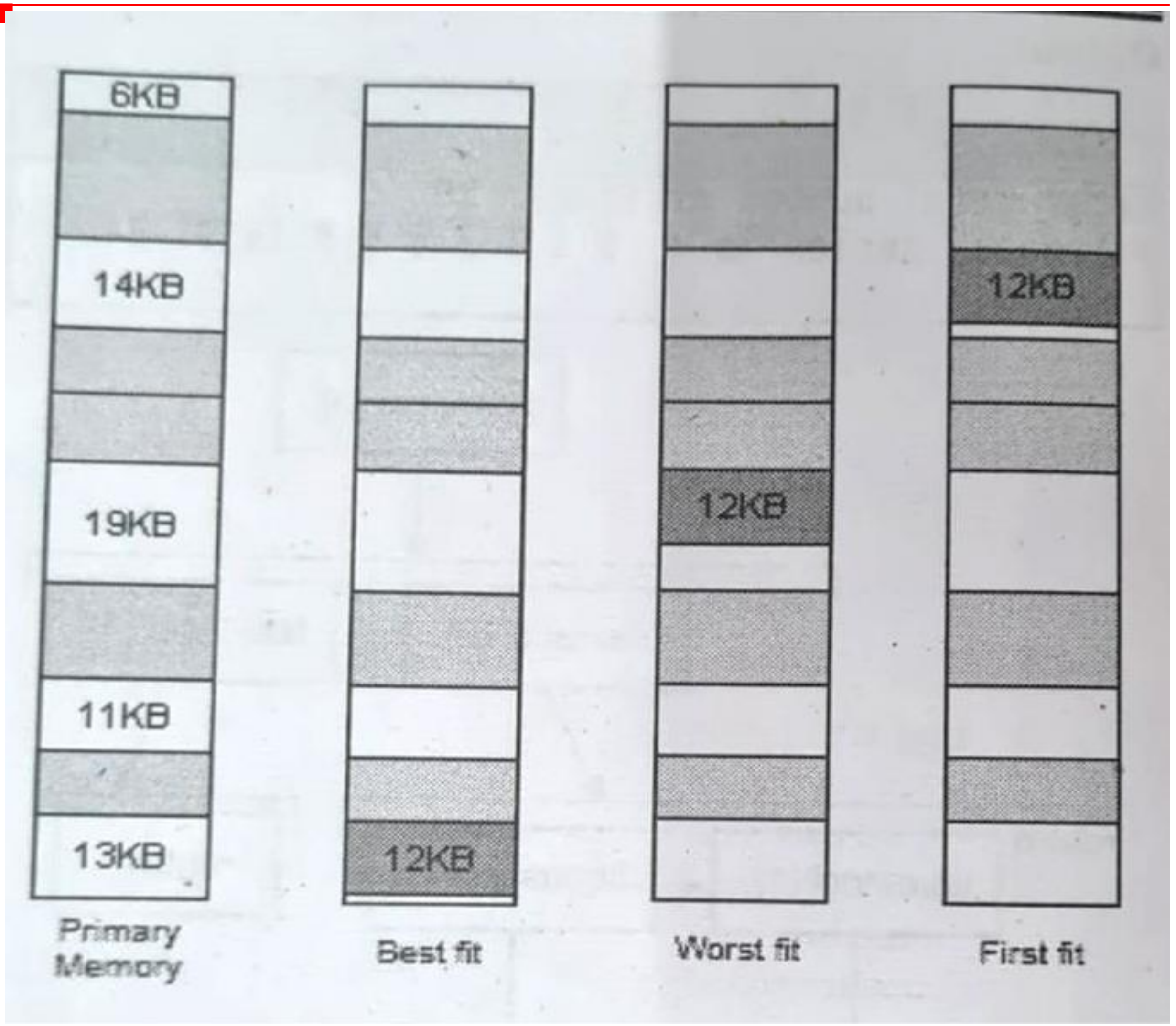
- **Next –fit method** : allocates the next available block that is sufficiently large

Refer 7.7.4: Reema Thareja

Sequential Fit methods :

For best fit and first fit, tiny segment of unallocated memory is left out, if this space is small, it may not be used for other process, this may lead to fragmentation. Thus worst case reduces fragmentation by allowing largest fragments for new processes.

Example: Assume that the main memory has following 5 fixed partitions with following sizes:100KB,500KB,200KB,300KB and 600KB in order. How would each of the first-fit, Best-fit and worst-fit algorithm place processes of 212KB,417KB,112KB and 426KB in order



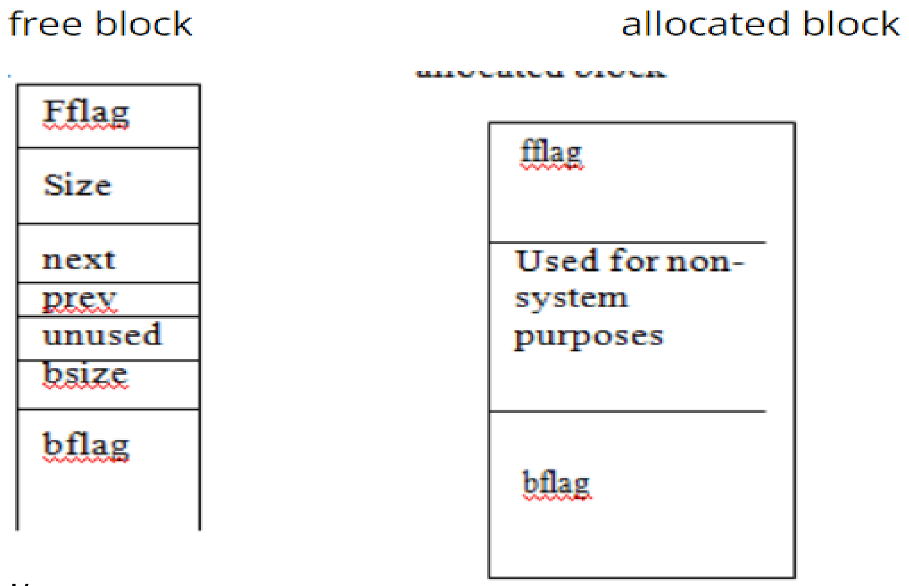
Boundary Tag Method :

To remove the arbitrary block from the free list (to combine it with the newly made free block) without traversing the whole list, the free list should be doubly linked.

Thus each and every free block must contain 2 pointers, **next** and **prev** to the next and previous free block on the free block. (This is required when combining a newly freed block with a free block which immediately proceeds in the memory). Thus the front of the free block should be accessible from its rear.

One way to do this is to introduce a **bsize** field at the given offset from the last location or address of each free block. This field have the same value as the **size** field at the front of the block.

Each of the contract fields fflag, prev, size, next, bsize next and bflag is shown as occupying a complete word, although in practice they can be packed together, several fields to a word. We assume that fflag or bflag are logically flags and that true signifies an allocated block and false indicates a free block. Using the information in the free block and the newly freed block, the system merges the two blocks into a bigger hole.



The figure shows the structure of free and allocated blocks under this method which is called the boundary tag method.

Refer: https://mucertification.com/topic/4-2-d-storage-management-boundary-tag-method/

Buddy system:

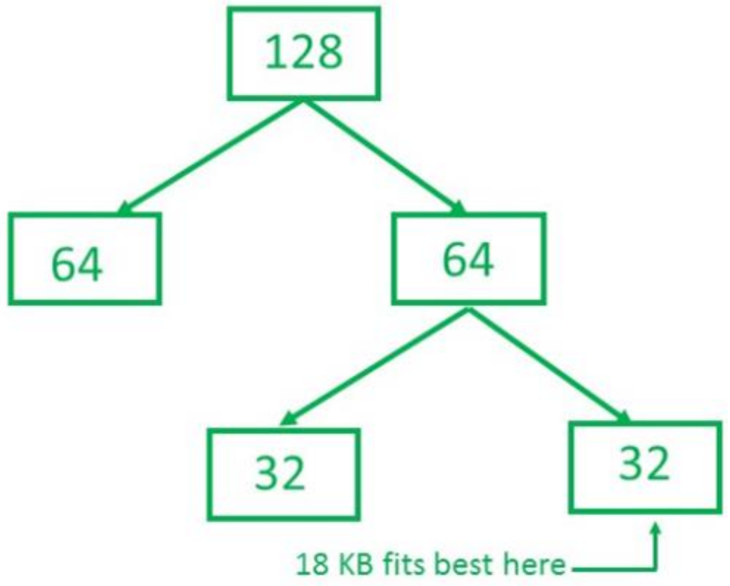
Static partition schemes suffer from the **limitation** of having the fixed number of active processes and the usage of space may also not be optimal.

The **buddy system** is a memory allocation and management algorithm that manages memory in **power of two increments**.

Two types of buddy systems:

Binary buddy system: block of sizes of power of two are allocated
Fibonacci Buddy system : block sizes which are Fibonacci numbers

Binary buddy system: block of sizes of power of two are allocated



The buddy system maintains a list of the free blocks of each size (called a free list) so that it is easy to find a block of the desired size if one is available. If no block of the requested size is available, Allocate searches for the first non-empty list for blocks of at-least the size requested. In either case, a block is removed from the free list.

Binary buddy system: block of sizes of power of two are allocated

The buddy system maintains a list of the free blocks of each size (called a free list) so that it is easy to find a block of the desired size if one is available. If no block of the requested size is available, Allocate searches for the first non-empty list for blocks of at-least the size requested. In either case, a block is removed from the free list.

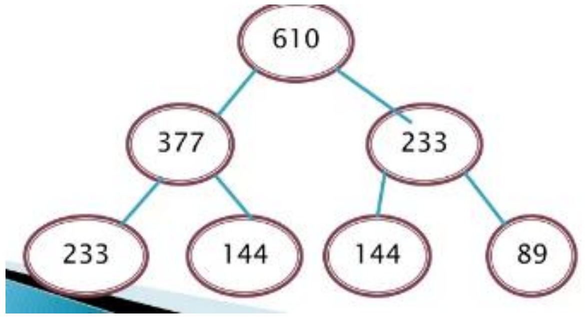
For example, suppose the size of the memory segment is initially 256kb, and the kernel requests 25kb of memory. The segment is initially divided into two buddies. Let's say A1 and A2, each 128kb in size. One of these buddies is further divided into two 64kb buddies, B1 and B2. But the next highest power of 25kb is 32kb so, either B1 or B2 is further divided into two 32kb buddies (C1 and C2), and finally, one of these buddies is used to satisfy the 25kb request. A split block can only be merged with its unique buddy block, which then reforms the larger block they were split from.

Fibonacci Buddy system:

A Fibonacci buddy system is a system in which blocks are divided into sizes which are Fibonacci numbers.

It satisfies the following relation:

$Z_i = Z_{(i-1)}+Z_{(i-2)}$

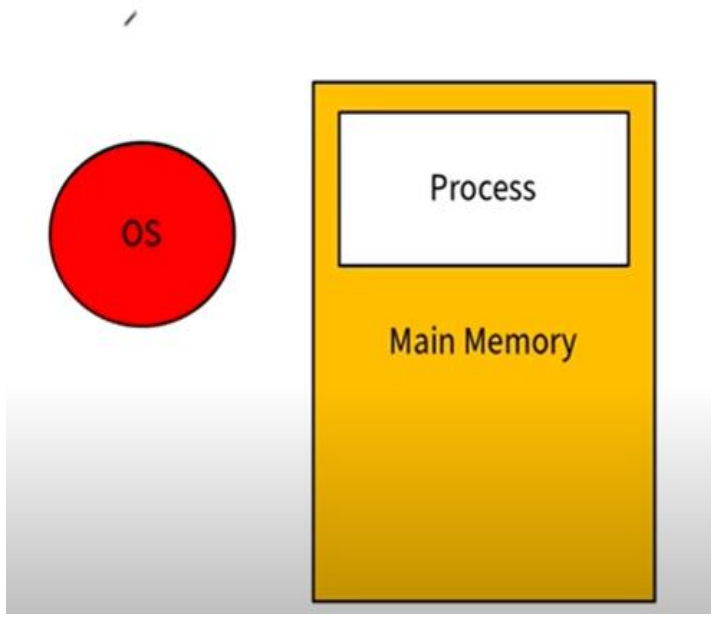


As per the definition, the elements of block size created under this scheme are: 0,1,1,2 3,5,8,13,21,34,55,89,144,233, 377,610, 987…….

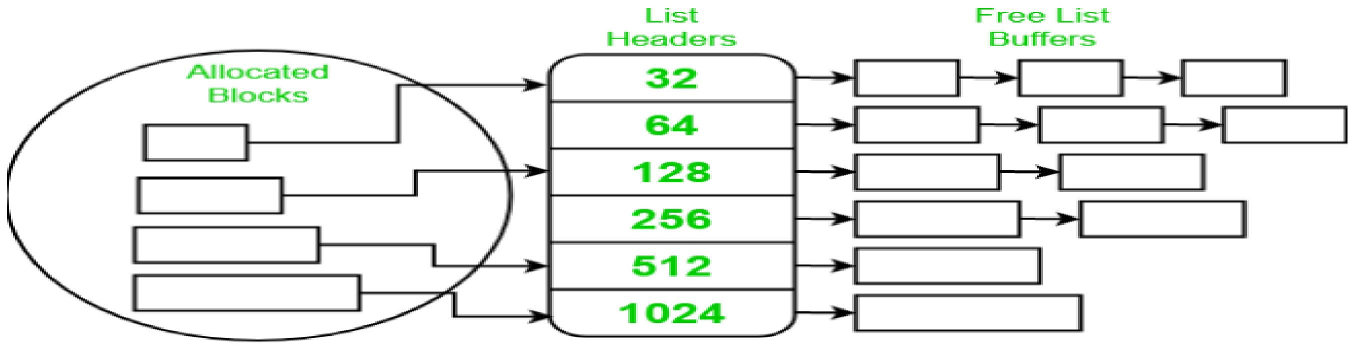
Garbage Collection and comapaction

- To reduce external fragmentation
- Move block back to memory so that fragments can be combined called as heap compaction.

Memory management

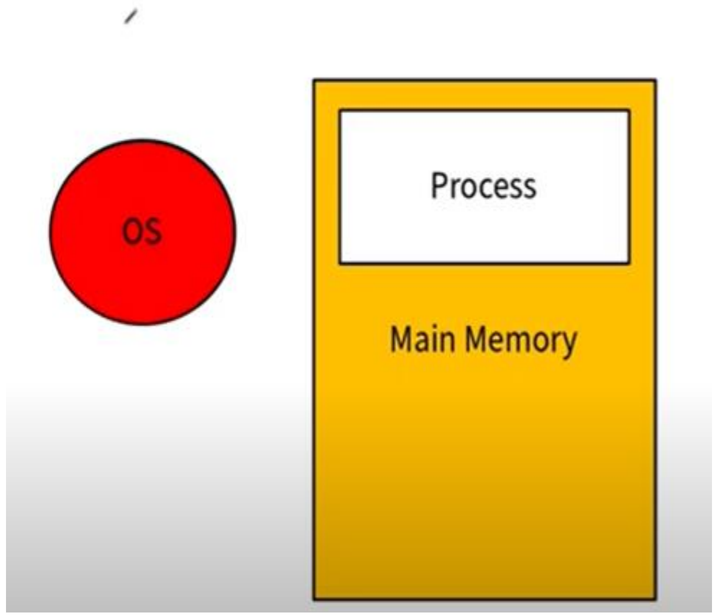


OS is responsible for memory and usage. It will allot memory to process that require them. It also collects it back when process is done with it.



A free list (or freelist) is a data structure used in a scheme for dynamic memory allocation. It operates by connecting unallocated regions of memory together in a linked list, using the first word of each unallocated region as a pointer to the next.

Memory management



But how does a OS know it certain memory is not in use? ----- difficult answer?

Let process decide is it is done with memory or not?

What is process forgets to release the memory it has

https://www.oreilly.com/library/view/introduction-to-data/9788131713921/xhtml/chapter007.xhtml#heading-00009

Memory management

There are two types of programming language methods:

- One type of language provides mutual mechanism to allot and release the memory eg C and C++, such language mainly focus on programmers and places all responsibilities on them for memory management.

Pros : if done right, provides fast mechanism to manage memory.
Cons: main focus is on programmer

Memory management

- automatic mechanism to release memory called Garbage collection eg JAVA and .NET.
- These language will automatically release memory once it is no longer in use

Pros: Programmer is free from managing memory
Cons: Garbage collection systems takes time

Garbage Collection (GC)

- It is automatic memory management
- The garbage collector or just collector attempts to reclaim garbage or memory occupied by objects that are no longer in use by the program
- Garbage collector was invented by American computer scientist john McCarthy in 1959 to simplify manual memory management in LISP

Garbage Collection (GC)

- Garbage collection frees the programmer from manually dealing with memory allocation.
- As a result certain categories of bugs are eliminated or substantially reduced.
- Dangling pointer bugs which occur when a piece of memory is frees while there are still pointers to it and one of those pointers is dereferenced. By then the memory may have been reassigned to another use with unpredictable results.

Garbage Collection (GC)

- Double free bugs which occur when program tries to free a region of memory that has already been freed and perhaps already been allocated again

Certain kinds of memory leaks in which program fails to free memory occupied by objects that have become unreachable which can lead to memory exhaustion

Garbage Collection (GC)

Disadvantages:

- Consuming additional resources, performing impacts, possible stalls in program execution and incompatibility with manual resource management

- Consumes resources in deciding which memory to free, even though the programmer may have already known this information

Garbage Collection (GC)

- The moment when the garbage is actually collected can be unpredictable, resulting in stalls (pause to shift/ free memory) scattered throughout a session

- Unpredictable stalls can be acceptable in some cases but in real time environments like transaction processing or interactive programs the stalls are not acceptable

- Thus real time garbage collectors to address this problem.

Garbage Collection (GC)

Tracing garbage collector : most common type of GC

Determining which objects should be garbage collected by tracing which object is reachable from certain root objects and considering the rest as garbage and collecting them

However there are large number of algorithms used in implementation with widely varying complexity and performance characteristics.

Garbage Collection (GC) :

Mark and Sweep method:

- Each object in memory has a flag (typically single bit) reserved for garbage collection use only
- The flag is always cleared except during the collection cycle

Garbage Collection (GC)

Mark and Sweep method:

The first stage is mark stage which does a traversal of the entire tree for the root set and mark each object that is pointed by the root as being "in-use"

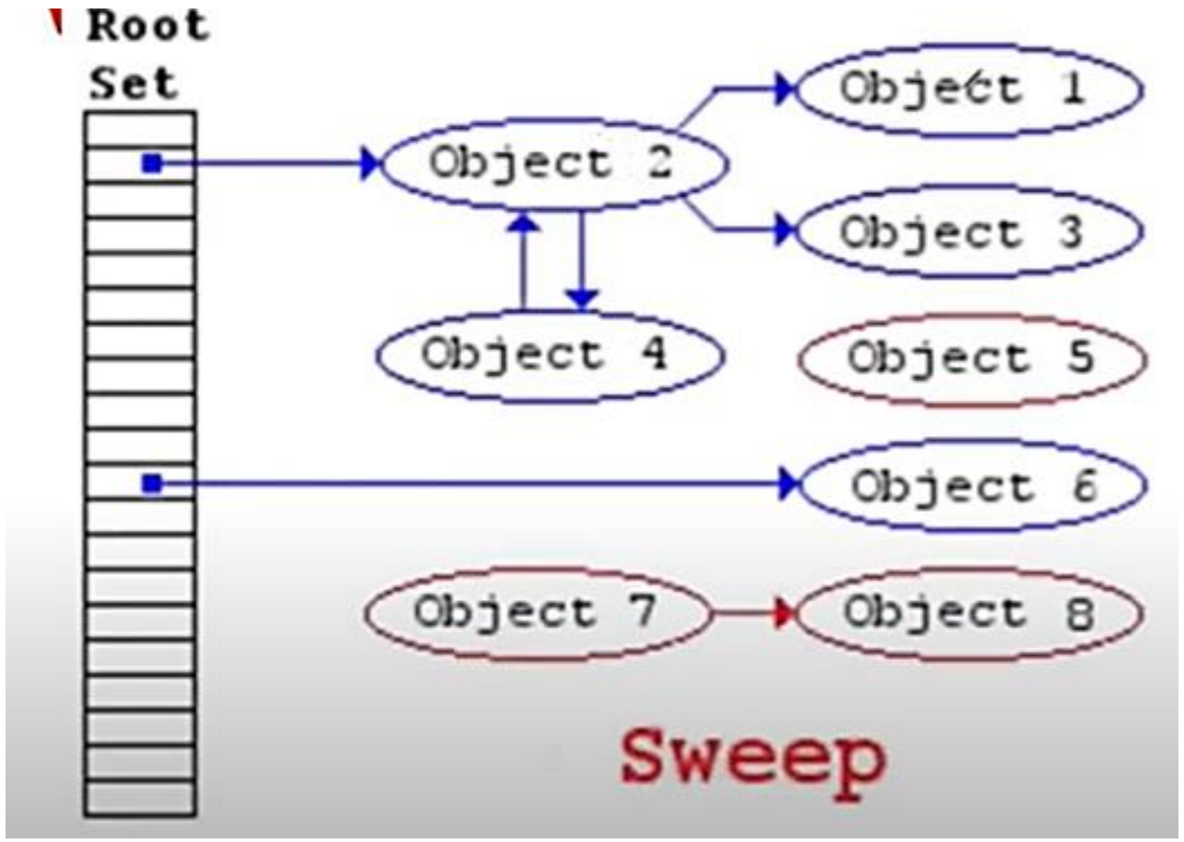
All the objects that those objects point to are also marked as well, so that every object that is reachable via root set is marked.

Garbage Collection (GC)

Mark and Sweep method:

In the second stage, the sweep stage, the all memory is scanned from start to finish, examining all the free or used blocks; those not marked as being “in-use” are not reachable by any roots and their memory is freed. For objects which were marked in-use , the in-use flag is cleared , preparing for next cycle.

Garbage Collection (GC)



Thank You