.

**Message-Passing Systems**

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions **without sharing the same address space** and is particularly useful in a **distributed environmen**t, where the **communication processes may reside on different computers connected by a network.**

Example: Chat application.

A message-passing facility provides at least two operations:

* send (message)
* receive (message)

Messages sent by a process can be of either fixed or variable size.

**FIXED SIZE:**

* The **system-level implementation is straightforward.** Because it is easy to design a system where data size is fixed .
* But makes the task of programming more difficult. Cannot vary message size -> bigger messages can be divided into multiple chunks -> makes hard to design a algorithm for handling these multiple chunks.

**VARIABLE SIZE:**

* Requires a more complex system-level implementation. Because no hardware can store variable sized data.
* But the programming task becomes simpler.

If processes P and Q want to communicate, they must send messages to and receive messages from each other.

A **communication link must exist** between them.

**This link can be implemented in a variety of ways. There are several methods for logically implementing a link and the send() /receive() operations, like:**

• Direct or indirect communication

•Synchronous or asynchronous communication

• Automatic or explicit buffering

There are several issues related with features like:

➤ Naming

➤ Synchronization

➤ Buffering

**Naming**

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

**Under direct communication-** Each process that wants to communicate must **explicitly name** the recipient or sender of the communication.

* send (P, message) - Send a message to process P.
* receive (Q, message) - Receive a message from process Q.

**A communication link in this scheme has the following properties:**

* A link is established automatically between **every pair of processe**s that want to communicate. The processes need to know only each other's identity to communicate.
* A link is associated with exactly two processes.
* Between each pair of processes, there exists exactly one link.
* This scheme exhibits **symmetry in addressing**; that is, both the sender process and the receiver process must name the other to communicate

**Another variant of Direct Communication-**

Here, only the sender names the recipient; the recipient is not required to name the sender.

* send (P, message) --> Send a message to process P.
* receive (id, message) --> Receive a message from any process. the variable id is set to the name of the process with which communication has taken place.

This scheme employs **asymmetry in addressing.**

* The disadvantage in both schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

**With indirect communication:**

* The messages are sent to and received from mailboxes, or ports.
* A mailbox can be viewed abstractly as an object into which **messages can be placed by processes and from which messages can be removed.**
* **Each mailbox has a unique identification.**
* Two processes can communicate only if the processes have a shared mailbox
* send (A, message) - Send a message to mailbox A.
* receive (A, message) - Receive a message from mailbox A.

**A communication link in this scheme has the following properties:**

* A link is established between a pair of processes only if both members of the pair have a shared mailbox.
* A link may be associated with more than two processes.
* Between each pair of communicating processes, there may be many different links, each corresponding to one mailbox.

**Now if P1, P2, P3 all share same mailbox A:**

Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1?

The answer depends on which of the following methods we choose:

* Allow a link to be associated with two processes at most.
* Allow at most one process at a time to execute a receive () operation.
* Allow the system to *select arbitrarily which process will* receive the message (that is, either P2 or P3, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, round robin where processes take turns receiving messages). The system may identify the receiver to the sender.

**A mailbox may be owned either by a process or by the operating system.**

**Synchronization**

Communication between processes takes place through calls to send() and receive () primitives. There are different design options for implementing each primitive.

Message passing may be either blocking or nonblocking— also known as synchronous and asynchronous.

**Message passing may be either blocking or nonblocking— also known as synchronous and asynchronous. Blocking and non-blocking can be both in sending and receiving.**

**Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.

**Nonblocking send:** The sending process sends the message and resumes operation.

**Blocking receive:** The receiver blocks until a message is available.

**Nonblocking receive**: The receiver retrieves either a valid message or a null.

**Buffering**

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

**Zero capacity:**

The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

**Bounded capacity:**

The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

**Unbounded capacity:** The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

**Sockets**

Used for communication in Client-Server Systems

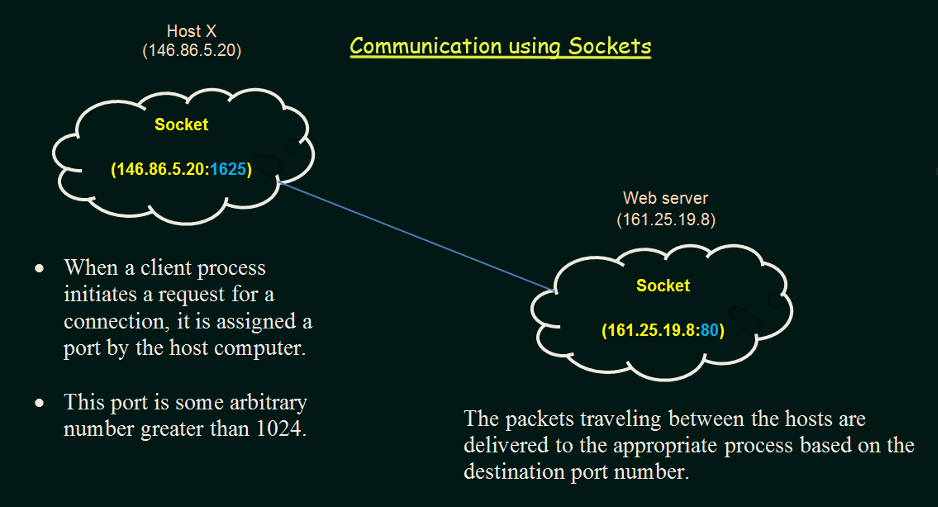
• A socket is defined as an endpoint for communication.

• A pair of processes communicating over a network employ a pair of sockets-one for each process.

• A socket is identified by an IP address concatenated with a port number.

• The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.

• Servers implementing specific services (such as telnet, ftp, and http) listen to well-known ports (a telnet server listens to port 23, an ftp server listens to port 21, and a web, or http, server listens to port 80). All ports below 1024 are considered well known; we can use them to implement standard services



**Memory Management:**

, CPUs are generally considered the most challenging to manufacture due to their extreme complexity and precision requirements. RAM manufacturing follows closely behind, while hard disk manufacturing, while still complex, is often regarded as slightly less difficult due to its more established processes and technologies.

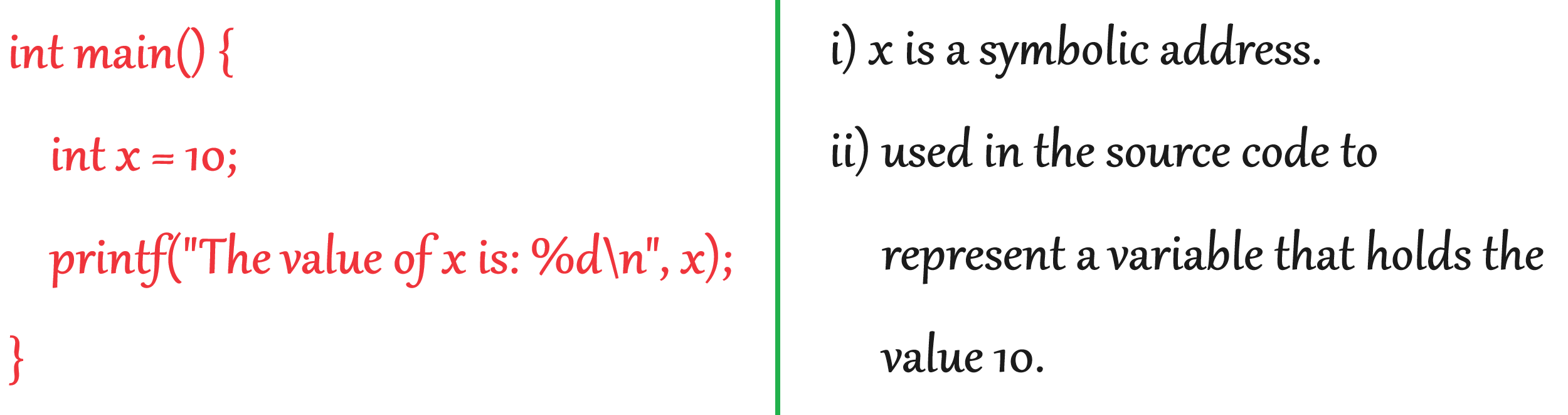
CPU, Ram cost is higher -> We cannot buy more than a threshold -> HDD/SSD contains all program as it is biggest in size.   
CPU, Cache, Register, Ram all are directly connected. SSD not directly connected with CPU. RAM connected to SSD directly

Ram works as a synchronization between high speed Processor and low speed SSD.

CPU makes Ram to load a program from SSD to RAM. Then CPU directly executes the program.

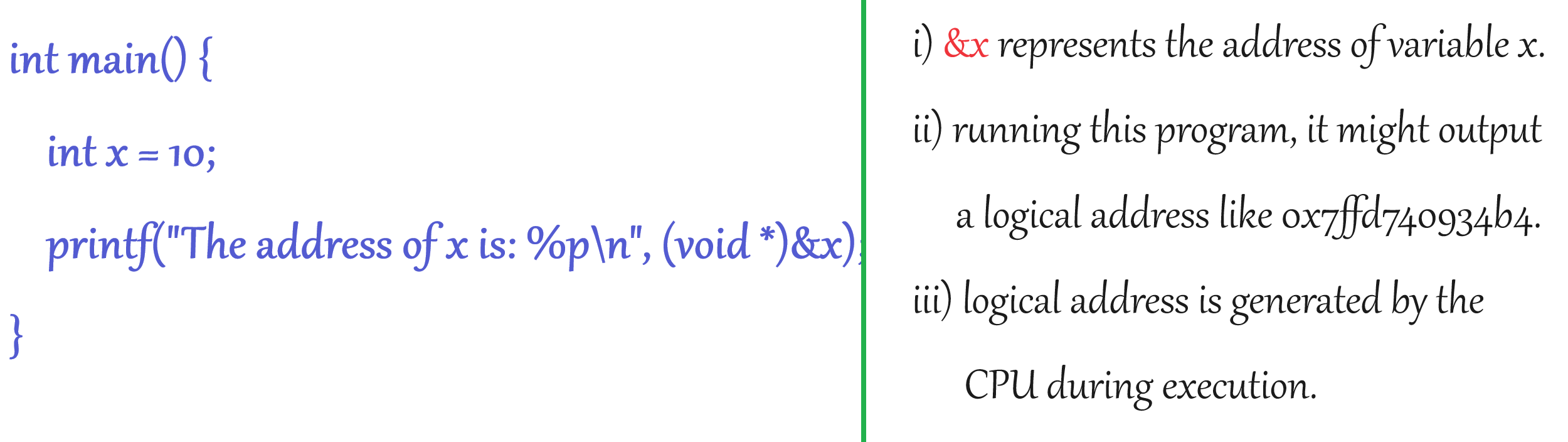
**Symbolic Addresses**:

names given to variables, functions, or other program elements in the source code.



**Logical Addresses**:

* logical addresses also known as virtual addresses
* the addresses generated by the CPU during the execution of a program.
* they represent locations in the virtual address space of a process.



**Physical Memory Addresses**:

refer to actual locations in the computer's physical memory (RAM).

**Absolute Memory Addresses.** Memory locations are uniquely identified by specific numerical values.

**Fixed Memory Addresses**:

Memory addresses are predetermined and remain constant throughout program execution.

They are managed by the operating system's memory management unit (MMU) and translated to physical addresses when accessed.

Usually, a program resides on a disk as a binary executable file. To run, the program must be brought into memory and placed within the context of a process. As the process executes, it accesses instructions and data from memory. After the process terminates, this memory is freed. Most systems allow a user process to reside in any part of the physical memory. Addresses in the source program are generally symbolic (such as a variable). A compiler typically binds these symbolic addresses to relocatable addresses (such as 14 bytes from the starting address of this module). The linker or loader in turn binds these relocatable addresses to absolute addresses.

Address binding of instructions and data to memory addresses can happen at three different stages:

Address binding in operating systems refers to **the process of associating symbolic addresses used by a program with physical memory addresses where the data and instructions reside during execution.**

There are several types of address binding, including compile-time binding, load-time binding, and run-time binding:

**Compile-time binding**:

* If memory location of a process is known before compilation, then absolute code can be generated. If the start location changes, the program must be recompiled. In this approach, the compiler translates symbolic addresses (such as variable names or function names) directly into absolute memory addresses.
* **Load-time binding**: If the memory location is not known at the compile time, then the compiler must generate relocatable code. In this case, the final binding can be delayed until load time. With load-time binding, the actual memory addresses are determined by the loader when the program is loaded into memory for execution. The loader adjusts the symbolic addresses in the program to match the physical addresses where the program will reside in memory.
* **Run-time binding**: Run-time binding involves delaying the address binding until the program is actually executed. This is commonly seen in dynamically linked libraries (DLLs) and shared objects in systems that support dynamic linking. The addresses are resolved by the operating system's dynamic linker/loader during program execution. If the process can be moved during its execution from one memory segment to another, then the address binding must be delayed until run time. Most OS use this method.

In compile time and load time , logical address will be same. But in Run time binding, logical address is converted into physical address by MMU.

**Source Program**:

* This is the original human-readable code written by the programmer.

**Compile Time**:

* This is the time at which the source code is converted into machine code by a compiler.

compiles the source code ( .c file) into an executable file.

**Object Module**:

* This is the intermediate output produced by the compiler, containing machine code and other necessary information.
* **Compilation**:
* The source code of the program is translated into machine code by a compiler.
* The output of this stage is usually an executable file or intermediate representation.
* **Linking**:
* If the program consists of multiple source files or uses external libraries, the linker combines them into a single executable file.
* **Loading**:
* The operating system loads the executable file into memory before it can be executed. This involves allocating memory space for the program and its data, setting up the program's memory layout, and initializing program resources.
* **Execution**:
* Finally, the CPU executes the loaded program. Instructions are fetched from memory, decoded, and executed sequentially.

**Chapter 9:**

Fixed partitioning is a memory management technique where **the memory is divided into fixed-size partitions,** and **each partition can hold exactly one process.**

**Protection:**

Each partition contains two limit values

When a process attempts to access memory, the operating system checks whether the memory access is valid according to the partition boundaries and access rights.

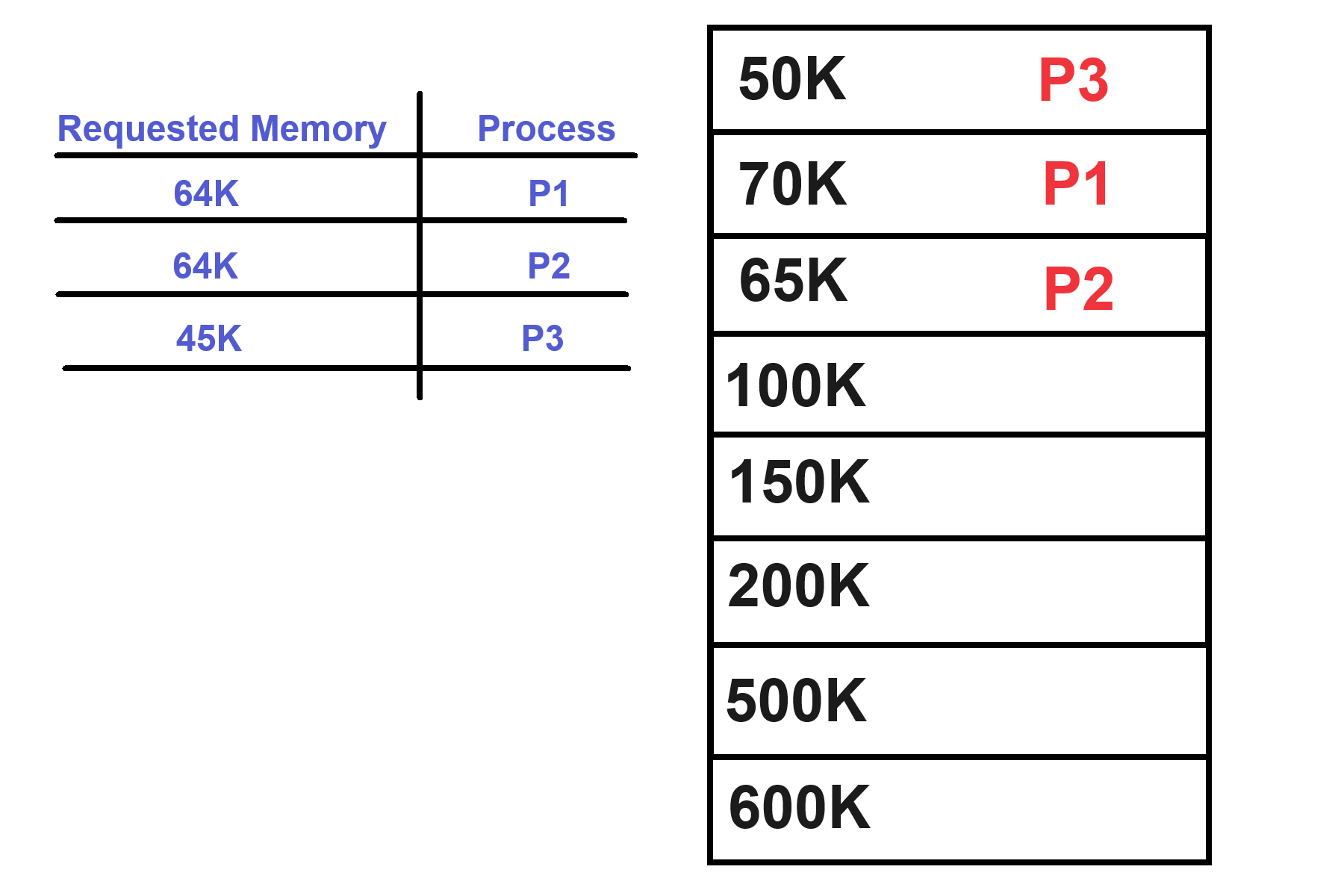
If Process A attempts to access memory outside its allocated partition (e.g., attempting to access memory in Partition 2), the operating system intervenes and terminates the process or generates an error, ensuring that Process A cannot interfere with the memory allocated to Process B.

**Allocation Policy:**

If a process is requested for memory allocation and multiple partitions are available to be allocated, then which partitions should be allocated to the requested process.

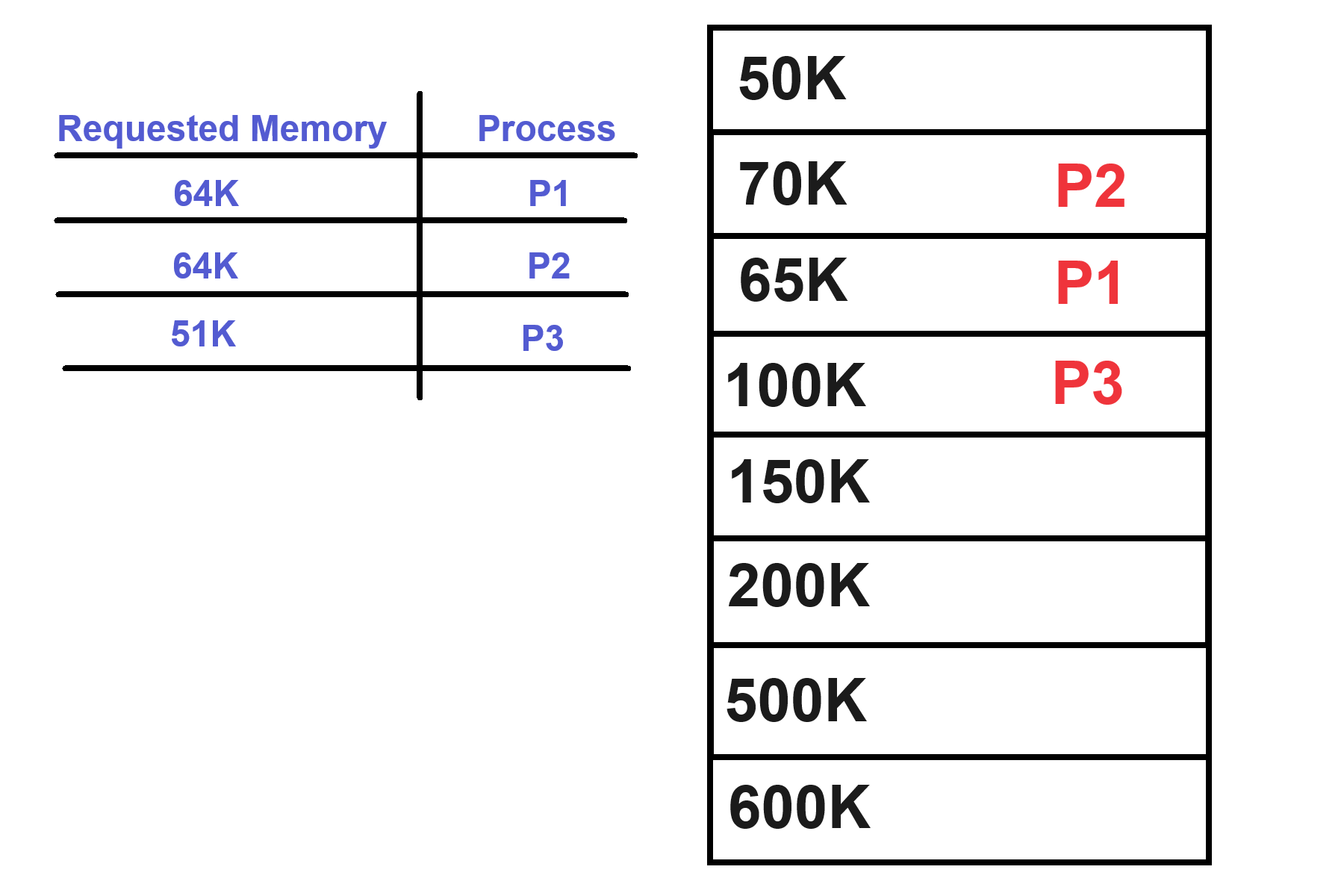
**First Fit**:

In the first fit allocation policy, the operating system allocates a process to the first available partition that is large enough to accommodate it.



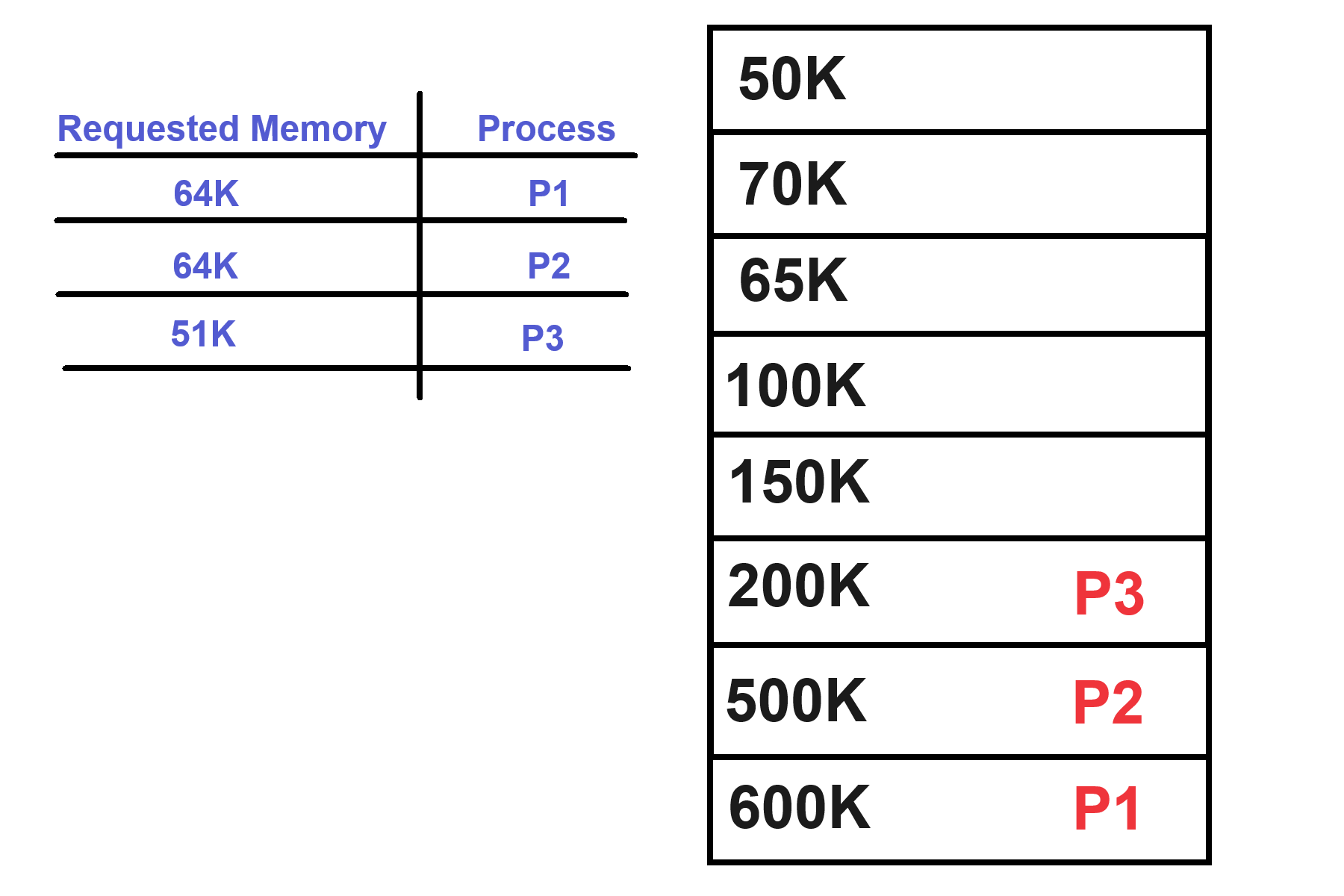
**Best Fit**:

In the best fit allocation policy, the operating system allocates a process to the partition that best fits its size, i.e., **the smallest available partition that can accomm**odate the process.



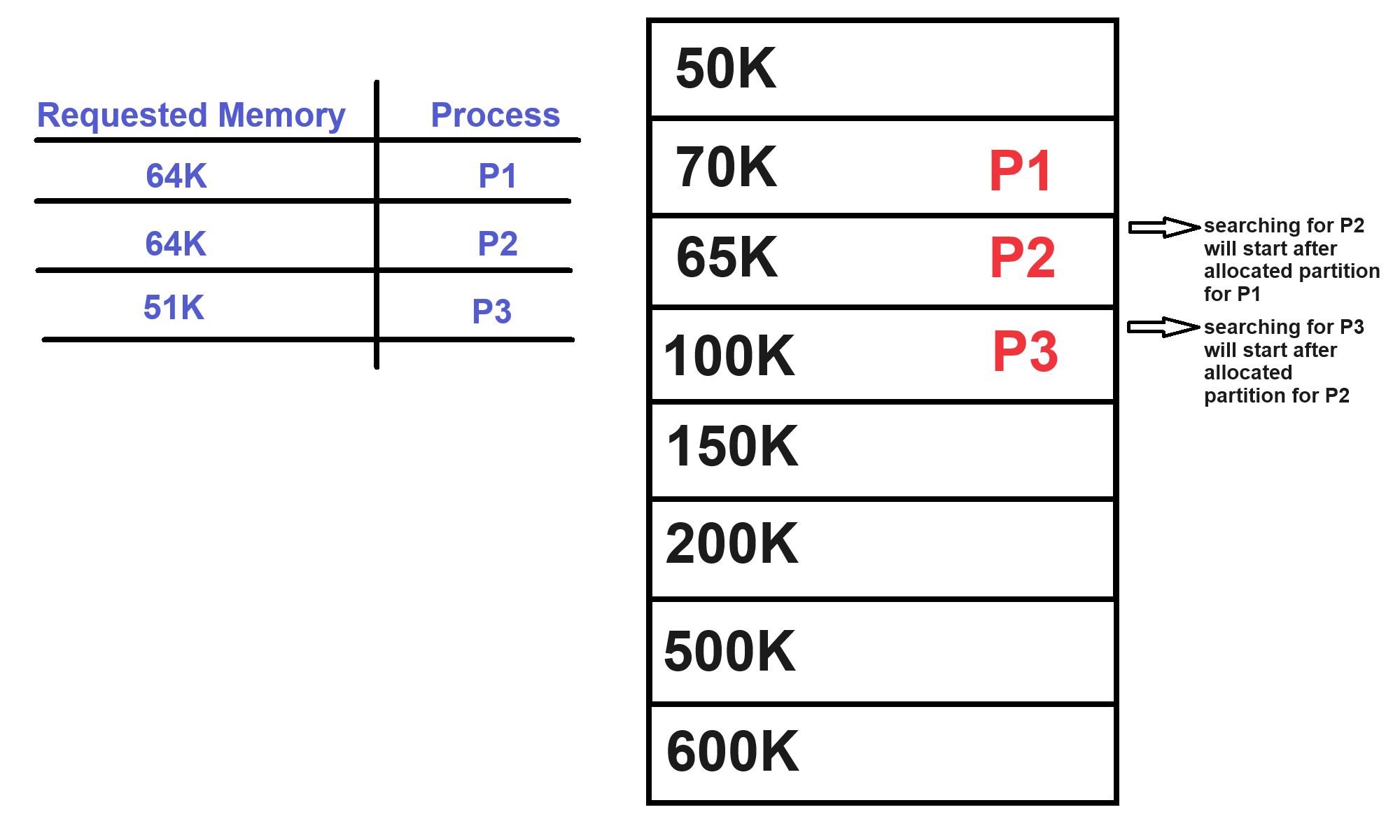
**Worst Fit**:

* In the worst fit allocation policy, the operating system allocates a process to the largest available partition.



**Next Fit:**

Same as First Fit but searching will start after the last allocated partition.



Fragmentation:

**External Fragmentation**: This type of fragmentation happens when there is enough total memory space to satisfy a request or to store a process, but the available space is not contiguous.

**Internal Fragmentation**: Internal fragmentation occurs when memory allocated to a process is larger than what the process actually needs. In fixed partitioning, internal fragmentation happens because each partition is of a fixed size, and if a process doesn't exactly fit the partition size, some memory within the partition remains unused.