**Chapter 8: Operation System Support**

Table of Contents

[8.1 Operating System Overview 3](#_Toc48998839)

[User/Computer Interface 3](#_Toc48998840)

[Resource Manager 6](#_Toc48998841)

[Types of Operating Systems 7](#_Toc48998842)

[Early Systems 8](#_Toc48998843)

[Simple Batch Systems 9](#_Toc48998844)

[Multi-Programmed Batch Systems 13](#_Toc48998845)

[Time-Sharing Systems 14](#_Toc48998846)

[8.2 Scheduling 15](#_Toc48998847)

[Long-Term Scheduling 17](#_Toc48998848)

[Medium-Term Scheduling 18](#_Toc48998849)

[Short-Term Scheduling 18](#_Toc48998850)

[Scheduling Techniques 19](#_Toc48998851)

[8.3 Memory Management 22](#_Toc48998852)

[Swapping 22](#_Toc48998853)

[Partitioning 24](#_Toc48998854)

[Paging 26](#_Toc48998855)

[Demand Paging 27](#_Toc48998856)

[Page Table Structure 29](#_Toc48998857)

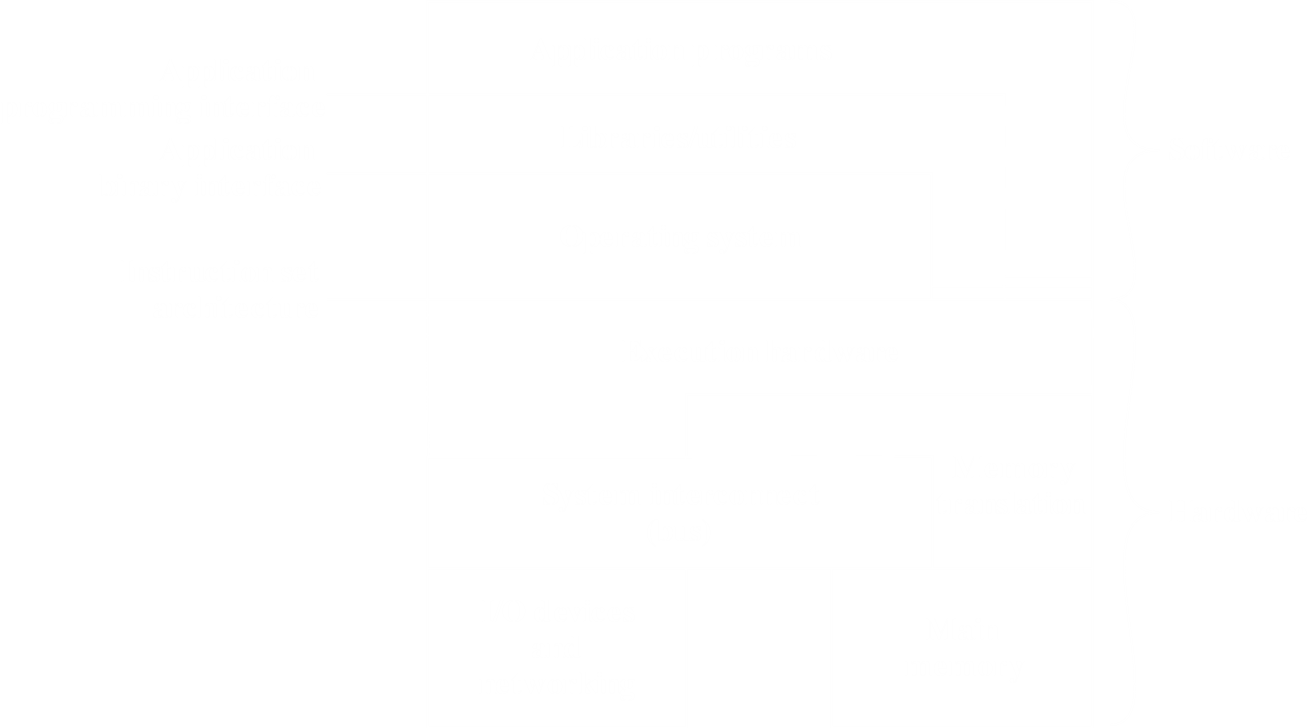
[Translation Lookaside Buffer 31](#_Toc48998858)

[Segmentation 33](#_Toc48998859)

## 8.1 Operating System Overview

An operating system (OS) is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. It has two objectives, making a computer more convenient to use, and allowing the computer system resources to be used efficiently.

### User/Computer Interface



The figure above shows the hardware and software used in providing applications to users. The end user is not generally concerned with the computer’s architecture, since they view the computer system in terms of an application. The application is expressed in a programming language and developed by an application developer. If they had to develop the application as a set of processor instructions that is completely reasonable for controlling the computer hardware, it would be an overwhelmingly complex task. System programs are provided to ease this task. Utilities are system programs that implement frequently used functions that assist in program creation, file management and control of I/O devices. A programmer makes use of these facilities to develop applications, and the applications invoke the utilities to perform certain functions. The most important system program is the OS. The OS masks the details of the hardware from programmers and provides them with a convenient interface for using the system. It acts as a mediator, making it easier for the programmer and the application programs to access and use those facilities and services.

The OS typically provides services in:

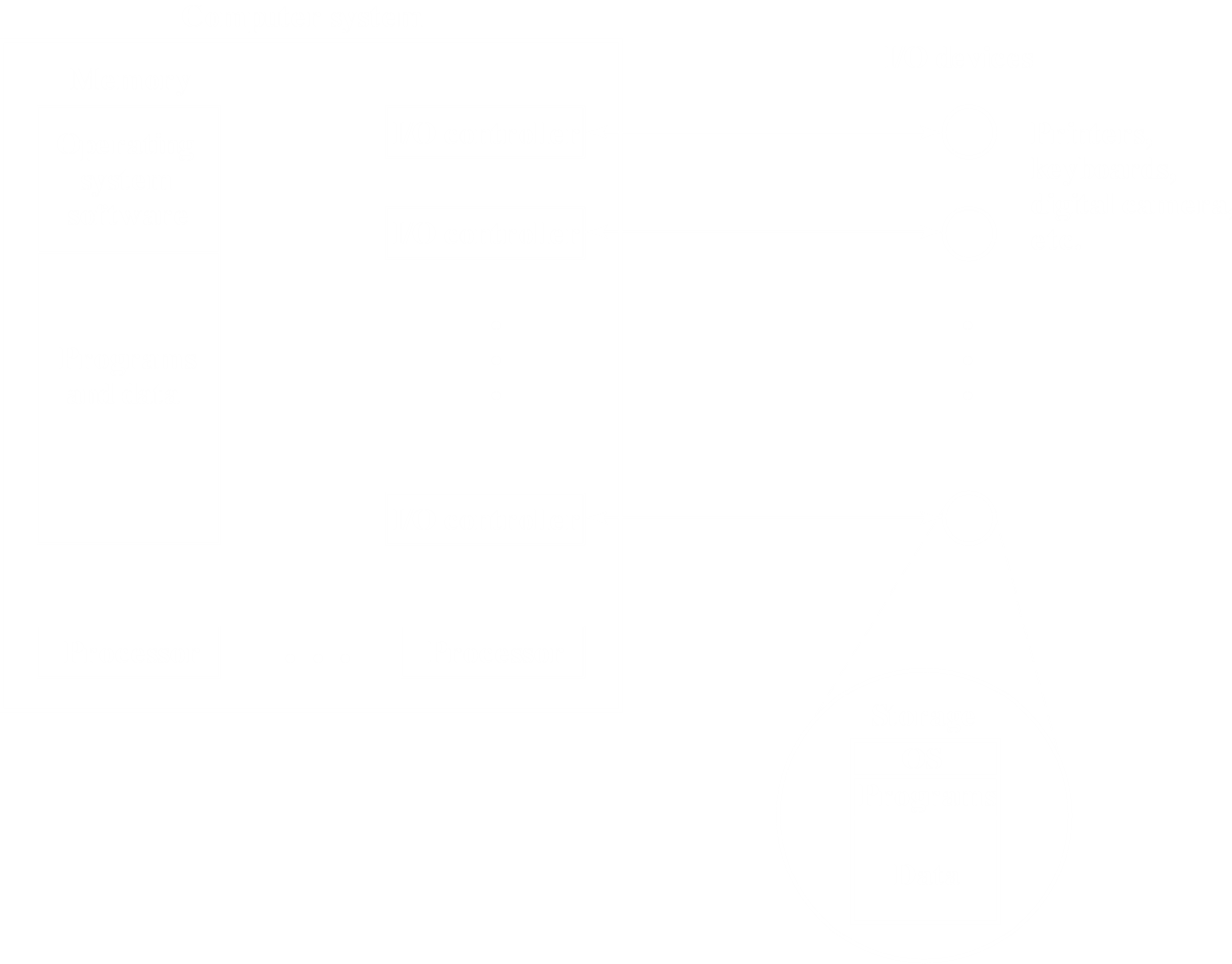
* **Program creation** – The OS provides a variety of facilities and services, such as editors and debuggers. Typically, these services are in the form of utility programs that are not part of the OS but are accessible through the OS.
* **Program execution** – Several steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The OS handles all of this.
* **Access to I/O devices** – Each I/O device requires its own specific set of instructions or control signals. The OS takes care of the details, so programs only have to deal with simple reads and writes.
* **Controlled access to files** – With files, the nature of the I/O device and the file format on the storage medium must both be understood. Again, the OS worries about the details. With multiple users, the OS can also provide protection mechanisms to control access to the files.
* **System access** – In the case of a shared or public system, the OS controls access to the system as a whole and to specific system resources. It must provide protection of resources and data from unauthorized users and resolve conflicts for resource contention.
* **Error detection and response** – A variety of errors can occur while a computer system is running. These include internal and external hardware errors like memory errors, device failure or malfunction and various software errors like arithmetic overflow, attempt to access forbidden memory locations, and inability of the OS to grant the request of an application. In each case, the OS must take the action that resolves the error with the least impact on the running applications. The responses may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.
* **Accounting** – A good OS collects usage statistics for various resources and monitors performance parameters like response time. On any system, this information is needed to anticipate the need for future enhancements and in tuning the system to improve performance. On a multiuser system, this information can be used for billing purposes. The figure shown earlier also indicates the three key interfaces in a typical computer system.
  + **Instruction Set Architecture (ISA)** – The ISA defines the machine language instructions that a computer can follow. This is the boundary between hardware and software. Note that both application programs and utilities can access ISA directly. For these programs, a subset of the instructions is available (user ISA). The OS has access to additional machine language instructions that deal with management of system resources (system ISA).
  + **Application Binary Interface (ABI)** – The ABI defines a standard for binary portability across systems. It defines the system call interface to the operating system and the hardware resources and services available in a system through the user ISA.
  + **Application Programming Interface (API)** – The API gives a program access to the hardware resources and services available in a system through the user ISA supplemented with high-level language (HLL) library calls. Any system calls are usually performed through libraries. Using an API allows application software to be ported easily, through recompilation, to other systems that support the same API.

### Resource Manager

A computer is a set of resources for the movement, storage and processing of data and the control of these functions. The OS is responsible for managing these resources. From one point of view, we can say that the OS is thus responsible for controlling the movement, storage and processing of data, but that is not entirely true. Normally, a controlling mechanism is thought of as something that is independent of that which it is controlling, but that is not the case for an OS. This is for two reasons:

* The OS functions like a normal computer software, executed by the processor
* The OS frequently releases control and depends on the processor to give it control again

Like other programs, the OS also provides instructions for the processor. The difference is that the OS directs the processor in the use of other system resources and in the timing of its execution of other programs. But for the processor to do these things, it must stop executing the OS program. Thus, the OS releases control to allow the processor to do ‘useful’ work, and then takes it back just long enough to prepare it to do the next piece of work.



The figure above shows the main resources managed by the OS. A portion of the OS is in the main memory. This includes the kernel, or nucleus, which contains the most frequently used functions in the OS, and at any given time, other portions of the OS currently being used. The remainder of the main memory contains user programs and data. The allocation of this resource (main memory) is controlled jointly by the OS and memory-management hardware in the processor. The OS decides when an I/O device can be used by a program and controls access to and use of files. The processor itself is a resource, and the OS must determine how much processor time should be devoted to the execution of a particular user program. In the case of a multiple-processor system, this decision must span all of the processors.

### Types of Operating Systems

There are certain key characteristics that differentiate various types of operating systems. The characteristics fall into two categories. The first is whether the system is batch or interactive.

An interactive system allows the user to interact directly with the computer, usually through a keyboard or display terminal, to execute a job or perform a transaction. Further, the user may communicate with the computer during the execution of the job, depending on the program.

A batch system is the opposite, with the user’s program being batched together with programs from other users and submitted by a computer operator. After the program is completed, results are printed out for the user. Pure batch systems are rare today.

The other category of characteristics is whether the system uses multiprogramming or not. With multiprogramming, it is attempted to keep the processor as busy as possible by having it work on more than one program at a time. Several programs are loaded onto memory, and the processor switches rapidly between them. The alternative is uni-programming, which works with only one program at a time.

#### Early Systems

With the earliest systems, the programmer interacted directly with the computer hardware, with no OS. The processors were run from a console, consisting of display lights, toggle switches, some form of input device and a printer. Programs in the processor code were loaded via the input device (e.g. a card reader), and if an error stopped the program, the error condition was indicated with the lights. The programmer could check the registers and main memory to determine the cause of the error. If there were no errors, the output appeared on the printer.

These early systems had two main problems:

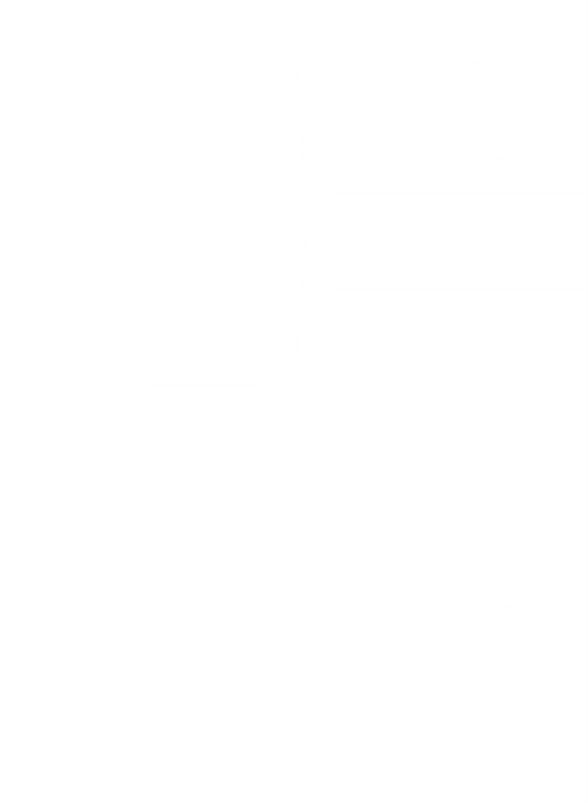
* **Scheduling** – Most systems used sign-up sheets to reserve processor time, with users typically getting blocks of multiples of half an hour. This meant a user finishing their work early left wasted computer idle time, and users not being able to finish their work in time meant them being forced to stop before they could finish.
* **Setup time** – A lot of time was needed to just set up the program to run. A single program, or job, could involve loading the compiler and high-level language program into memory, saving the compiled program, and then loading and linking the object program and common functions. Each of these steps could involve mounting or dismounting tapes, or setting up card decks. If an error occurred, the user typically had to start over.

This type of operation is called serial processing, since the users have access to the computer in a serial. Over time, various system software was developed to make serial processing more efficient. These included libraries of common functions, linkers, loaders, debuggers and I/O driver routines available as common software for all users.

#### Simple Batch Systems

Early processors were very expensive, so it was not acceptable to have any wasted time. Maximizing processor utilization was essential, and so simple batch operating systems were developed. Under a simple batch operating system, or monitor, the user does not have direct access to the processor, Instead, they submit the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device.

Let us look at this process from the monitor’s view and the processor’s view to understand it better. The monitor controls the sequence of events. To do this, much of the monitor must constantly be in main memory and available for execution, as shown below. This part of the monitor is called the resident monitor. The rest of the monitor consists of utilities and common functions that are loaded as subroutines to the user program at the beginning of any job that needs them. The monitor reads in jobs one at a time from the input device (a card reader or magnetic tape), placing the current job in the user program area and passing control to it. When the job is done, control is returned to the monitor, which reads the next job. The result of each job is printed out for delivery to the user.



From the processor’s view, at a certain point the processor is executing instructions from the part of the main memory that has the monitor. The instructions cause the next job to be read into another part of the main memory. Once that is done, the processor finds a branch instruction in the monitor that tells it to continue executing at the start of the user program. The processor then executes instructions from the user program until it finds an error or an ending condition. In either case, the processor will then fetch its next instruction from the monitor. The phrase ‘control is passed to a job’ means the processor is working with user program instructions, and the phrase ‘control is returned to the monitor’ means the processor is working with monitor instructions.

Thus, the monitor handles the scheduling problem. A batch of jobs is queued and jobs are executed as quickly as possible with no idle time.

Now consider the job setup time. This is handled by the monitor as well. Each job has instructions included in a Job Control Language (JCL), which is a special programming language that provides instructions to the monitor. Say the user submits a program in the FORTAN language, along with some data. Each FORTAN instruction and item of data is a separate punched card or record on tape. There are also job control instructions, denoted by a starting $ sign. The overall job looks like this:

$JOB  
$FTN  
// FORTAN instructions  
$LOAD  
$RUN  
// Data  
$END

The monitor reads the $FTN line and loads the appropriate controller from its mass storage (usually tape). The compiler translates the user’s program into object code, which is stored in memory or mass storage. If it is stored in memory, the operation is called ‘compile, load and go’. It if is stored on tape, the $LOAD instruction is needed, which is read by the monitor when it regains control after the compile operation. The monitor invokes the loader, which loads the object program into memory in place of the compiler and transfers control to it. Thus, a large segment of main memory can be shared between different subsystems, though only one subsystem can be in the memory and executing at a time.

The monitor, or batch OS, is just a computer program. It relies on the processor to fetch instructions from different parts of main memory in order to seize and release control alternately. It also has other hardware features such as:

* **Memory protection** – The user program must not alter the memory area containing the monitor. If an attempt is made to do this, the processor should detect it and transfer control to the monitor, which would abort the job, print an error message and load the next job.
* **Timer** – A timer is needed to prevent a single job from monopolizing the system. The time is set at the beginning of each job, and if it expires, an interrupt occurs and control is returned to the monitor.
* **Privileged instructions** – Certain instructions are considered privileged and can only be executed by the monitor. If the processor finds one of these instructions in a user program, an error interrupt occurs and control is transferred back to the monitor. Privileged instructions include I/O instructions, so that I/O devices remain under the control of the monitor. This prevents things like the current user program accidentally reading job control instructions from the next job. If the user program needs to perform I/O, it must request the monitor to perform the operation for it.
* **Interrupts** – Early computer models did not have this capability. Interrupts give the OS more flexibility in releasing and regaining control from user programs.

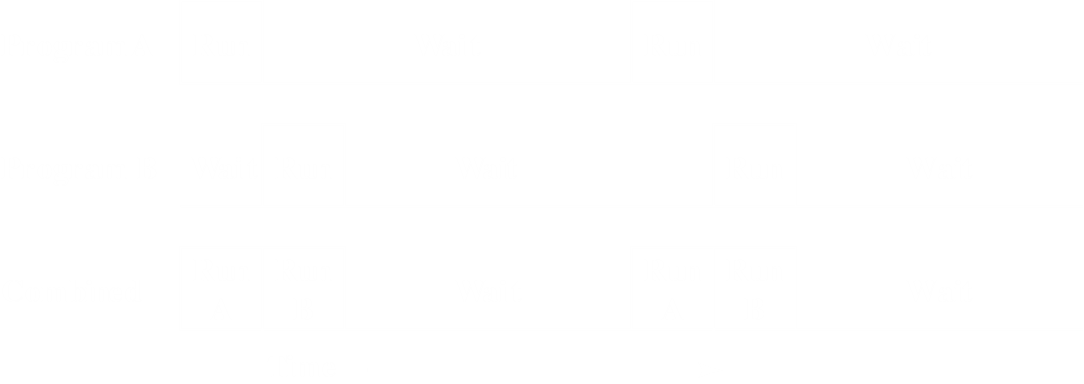
Processor time alternates between executing the user program and the monitor. There are two sacrifices here: some memory has to be given to the monitor, and some processor time is consumed by the monitor. Even with these two overheads, the simple batch system improves utilization of the computer.

#### Multi-Programmed Batch Systems

Even with automatic job sequencing in simple batch OS, the processor is often idle due to the fact that I/O devices are slow compared to the processor. This is called uni-programming, and it looks a little like this:



If we increase our main memory enough so that it can hold the OS and two sets programs, then we can work on another program while the first one is waiting for I/O, since that program most likely is not currently waiting for I/O.



We could expand this idea further to hold even more programs. This technique is called multiprogramming or multitasking.

The most notable additional feature useful to multiprogramming is I/O interrupts and DMA. With interrupt-driven I/O or DMA, the processor can issue an I/O command for one job and proceed executing another job while the I/O is carried out by the device controller. When the I/O operation is complete, the processor is interrupted and control is passed to an interrupt-handling program in the OS. The OS then passes control to another job.

Multiprogramming operating systems are more sophisticated than uni-programming ones. To have several jobs ready to run, the jobs must be kept in main memory, which means some memory management is needed. Additionally, if several jobs are ready to run, the processor must decide which one to run, which means some algorithm for scheduling is needed.

#### Time-Sharing Systems

For many jobs, it is desirable to provide a mode where users interact directly with the computer. For some jobs, like transaction processor, the interactive mode is essential. Today, this requirement is met by the use of a dedicated microcomputer. In earlier times, this was not available, so time sharing was developed.

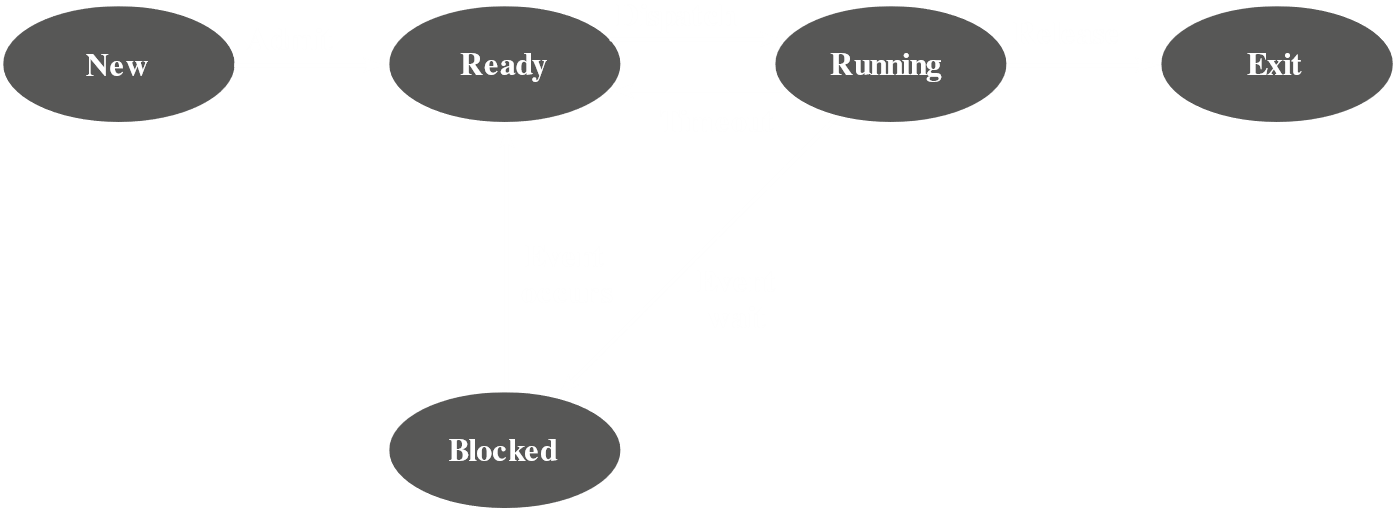
Just like how multiprogramming allows the processor to handle multiple batch jobs at a time, it can also be used to handle multiple interactive jobs. This latter technique is called time sharing, because the processor’s time is shared between multiple users. In a time-sharing system, multiple users simultaneously access the system through terminals, and the OS interleaves the execution of each user program in short bursts of computation. Thus, for users, each user will see roughly the effective computing speed. Due to the relatively slow human reaction time, the response time on a properly designed system is comparable to that on a dedicated computer.

|  |  |  |
| --- | --- | --- |
|  | **Batch Multiprogramming** | **Time Sharing** |
| **Principal objective** | Maximize processor use | Minimize response time |
| **Source of directives to operating system** | Job control language commands provided with the job | Commands entered at the terminal |

## 8.2 Scheduling

The key to multiprogramming is scheduling. Before we get into the details of scheduling, we need to understand the concept of a process. A process is a job that is being executed, i.e. the entity that is currently assigned to the processor.

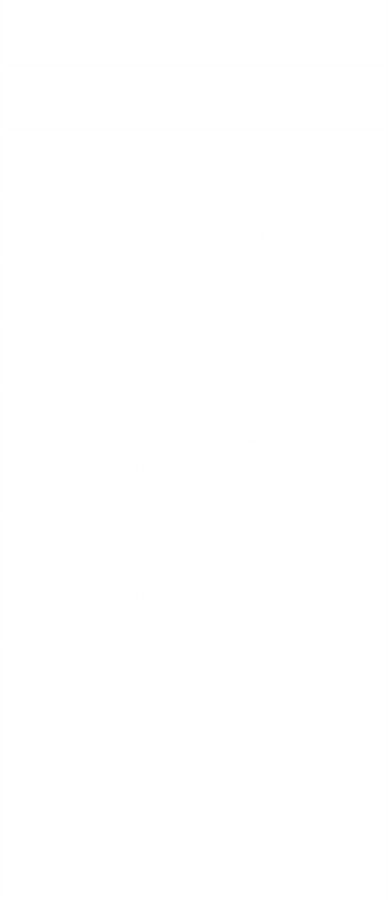
A process can be in one of several states. These states are shown below:



* **New** – A program is admitted by the scheduler, but is not ready to be executed. It needs to be initialized by the OS.
* **Ready** – The process is ready to be executed and is waiting for the processor.
* **Running** – The process is being executed by the processor.
* **Waiting** – The process is suspended from executing and is awaiting some system resource, such as I/O.
* **Halted** – The process has terminated and will be destroyed by the OS.

For each process in the system, the OS must maintain information indicating the state of the process and other necessary information for process execution. To do this, each process is represented in the OS by a process control block (PCB), which contains:

* **Identifier** – Each current process has a unique identifier.
* **State** – The current state of the process.
* **Priority** – Relative priority level.
* **Program Counter** – The address of the next instruction in the program to be executed.
* **Memory Pointers** – The starting and ending locations of the process in memory.
* **Context Data** – This is data that is present in registers in the processor while the process is executing. It will be discussed in detail later. For now, it is enough to know it represents the ‘context’ of the process. The context data plus the program counter are saved when the process leaves the running state and are retrieved by the processor when it resumes execution of the process.
* **I/O Status Information** – Includes outstanding I/O requests, I/O devices assigned to this process, a list of files assigned to the process and so on.
* **Accounting Information** – May include the amount of processor time and clock time used, time limits, account numbers and so on.



When the scheduler accepts a new job or user request for execution, it creates a blank process control block and places the associated process in the new state. After the system has properly filled the process control block, the process is transferred to the ready state.

The key to multiprogramming is scheduling. There are four types of scheduling typically involved:

* Long-term scheduling – The decision to add to the pool of processes to be executed.
* Medium-term scheduling – The decision to add to the number of processes that are partially or fully in main memory.
* Short-term scheduling – The decision as to which available process will be executed by the processor.
* I/O scheduling – The decision as to which process’s pending I/O request shall be handled by the available I/O device.

### Long-Term Scheduling

The long-term scheduler determines which programs are admitted to the system for processing, thus controlling the ‘degree’ of multiprogramming (number of processes in memory). Once admitted, a job or user program becomes a process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition, in which case it is added to the queue for the medium-term scheduler.

In a batch system, or the batch portion of a general-purpose OS, newly submitted jobs are routed to disk and held in a batch queue, and the long-term scheduler creates processes from there when it can. The scheduler must firstly decide that the OS can take on one or more additional processes, and then it must decide which job or jobs to accept and turn into processes. The criteria for the latter may include priority, expected execution time, and I/O requirements.

For interactive programs in a time-sharing system, a process request is generated when a user attempts to connect to the system. Time-sharing users are not queued up and kept waiting until the system can accept them. The OS will accept all authorized users until the system is saturated, using some predefined measure of saturation. At that point, a connection request is met with a message indicating that the system is full.

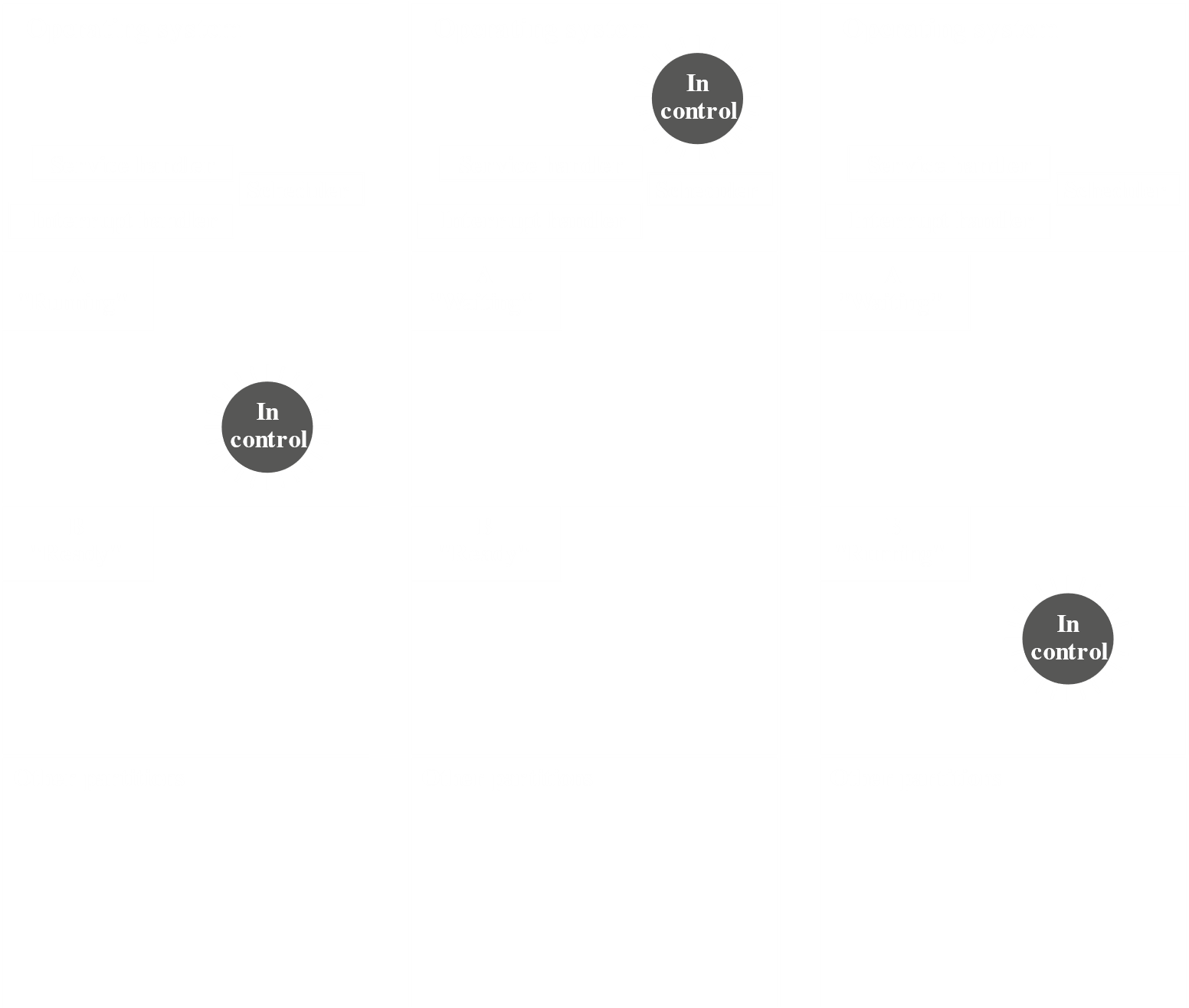
### Medium-Term Scheduling

Medium-term scheduling is part of the swapping function described in the next section. Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming. On a system that does not use virtual memory, memory management is also an issue. Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.

### Short-Term Scheduling

The long-term scheduler executes relatively infrequently and makes the decision of whether or not to take on a new process, and which one to take. The short-term scheduler, or dispatcher, executes frequently and decides which job to execute next.

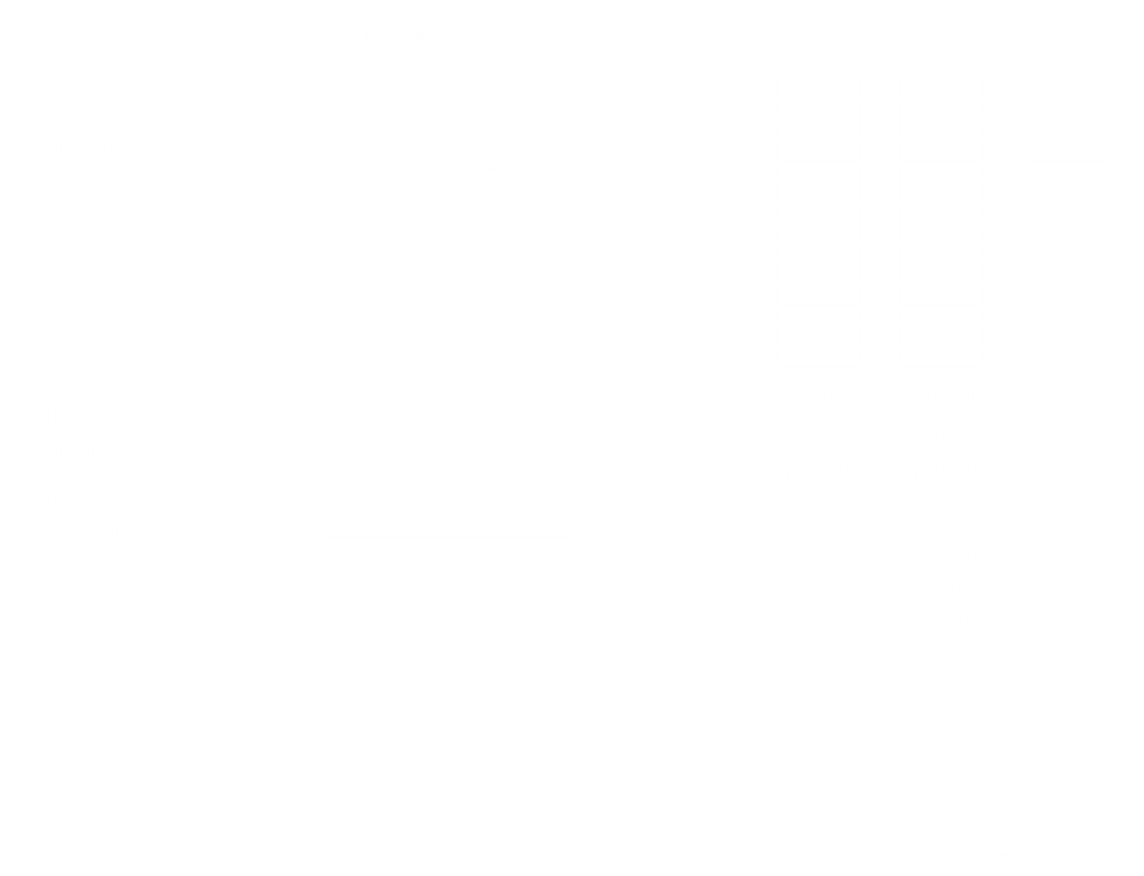
### Scheduling Techniques



The figure above shows how main memory is partitioned at a given point in time. We have the OS, which is always resident, and two active processes A and B. In the beginning, A is running and the processor is executing instructions from the program contained in A’s memory partition. At some point, the processor will cease to do this and begin executing instructions in the OS area. This may happen due to one of three reasons:

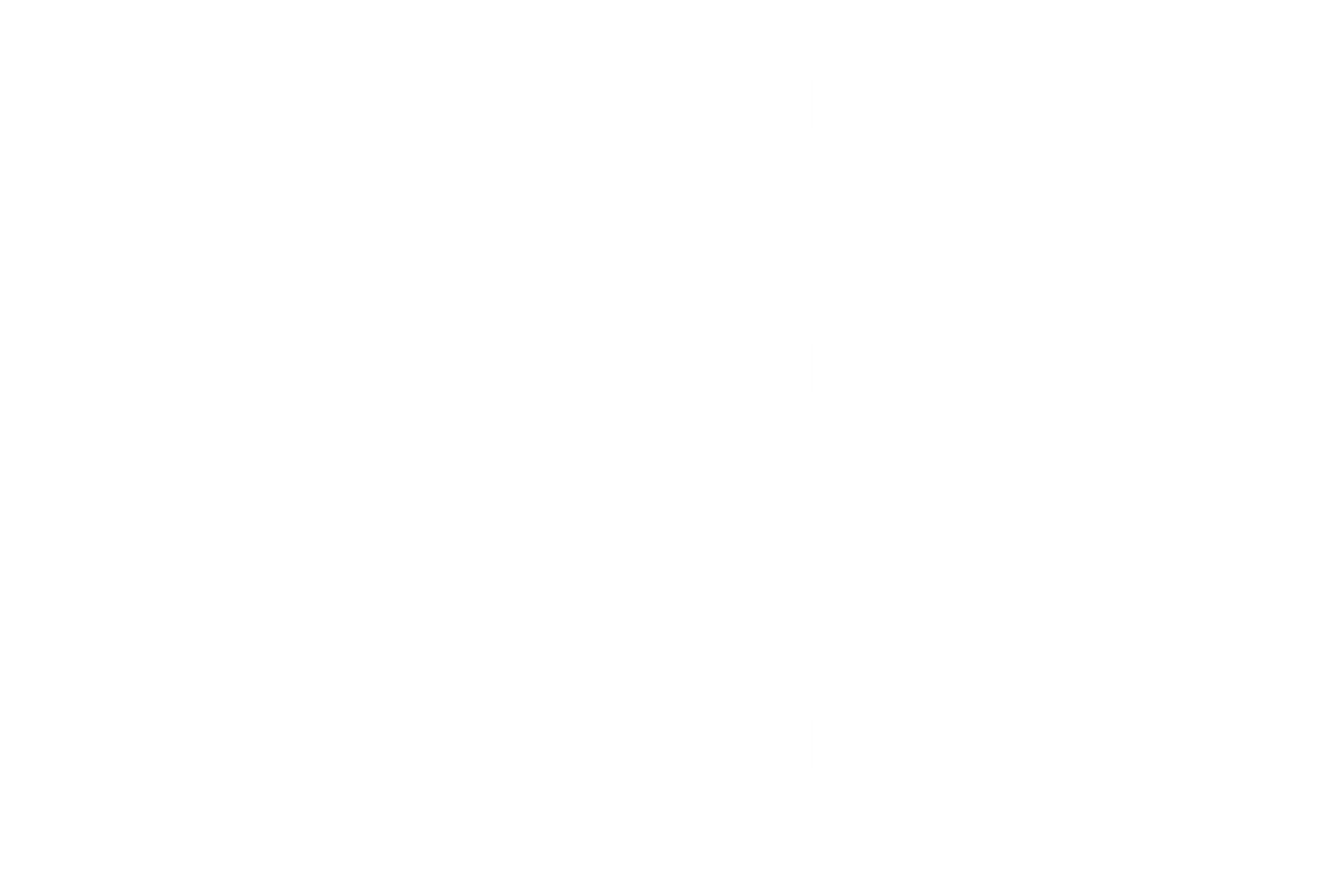
1. Process A issued a service call (like an I/O request) to the OS, which suspends its execution until the call is satisfied.
2. Process A causes an interrupt, so the processor transfers to the interrupt handler in the OS. Many reasons could cause an interrupt, like an error due to trying to execute a privileged instruction, or a timeout to prevent the process from monopolizing the processor.
3. Some event unrelated to process A that requires attention causes an interrupt, like the completion of an I/O operation.

In any case, the processor saves the current context data and program counter for A in A’s process control block and begins executing in the OS. The OS may perform some work, such as initiating an I/O operation, after which the short-term scheduler would decide which process to execute next. In this example, B is chosen, so the OS instructs the processor to restore B’s context data and continue execution of B where it left off.



The figure above shows the major elements of the OS involved in multiprogramming and scheduling of processes. The OS receives control of the processor at the interrupt handler if an interrupt occurs and at the service-call handler if a service call occurs. Once the interrupt or service call is handled, the short-term scheduler is invoked to select a process for execution.

The OS also maintains some queues. Each queue is just a list of processes waiting for some resource. The long-term queue is for jobs waiting to use the system. As conditions permit, the high-level scheduler will allocate memory and create a process for each of them. The short-term queue has processes that are in the ready state. Any of these could use the processor next, depending on which the short-term scheduler picks. Generally, this is done with a round-robin algorithm, giving each process some time in turn. Priority levels may also be used. The I/O queue is for each I/O device. More than one process may ask to use the same I/O device, so they get lined up in that device’s queue.



The figure above shows how processes progress through the computer under the control of the OS. Each process request is put in the long-term queue. When resources become available, a process request becomes a process and is placed in the ready state and put in the short-term queue. The processor alternates between executing OS instructions and user processes. While the OS is in control, it decides which process in the short-term queue should be executed next. When the OS has finished its immediate tasks, it gives the processor to the chosen process.

A process under execution may be suspended for many reasons. If it is due to an I/O request, it is placed in the appropriate I/O queue. If it is due to a timeout or other pressing business requiring the OS, it is put in the ready state and placed in the short-term queue.

When an I/O operation completes, the OS removes the satisfied process from the I/O queue and places it in the short-term queue. It then selects the next waiting process and signals the I/O device to satisfy that process’s request.

## 8.3 Memory Management

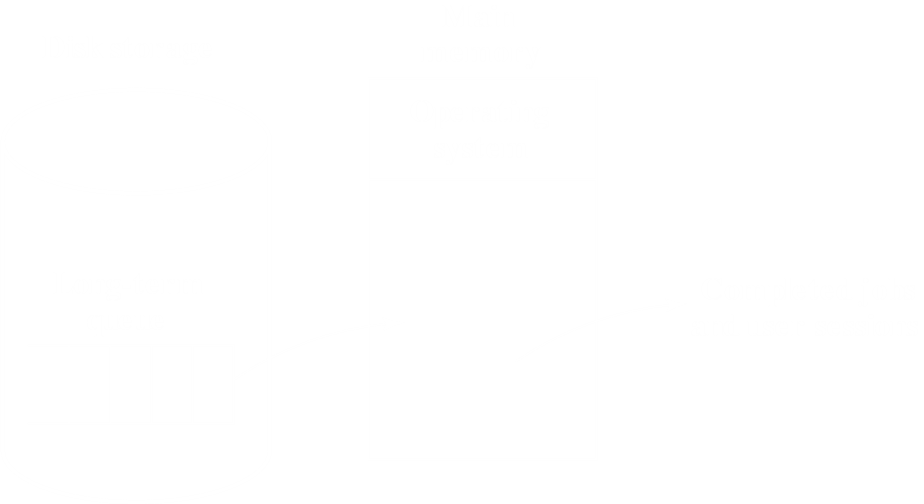
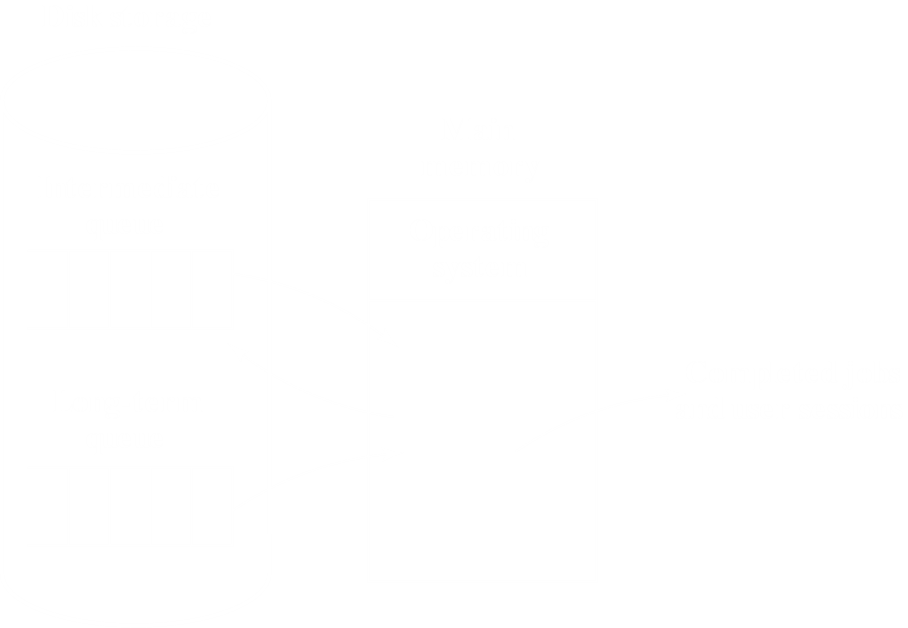
In uni-programming, the main memory is divided into two parts, one for the OS or resident monitor and one for the program currently being executed. In a multi-programming system, the ‘user’ part of memory is subdivided to accommodate multiple processes. The subdivision is carried out dynamically by OS and is called memory management.

Effective memory management is vital in a multi-programming system. If only a few processes are in memory, then for much of the time all the processes will be waiting for I/O and the processor will be idle. Thus, memory needs to be allocated efficiently to pack as many processes into memory as possible.

### Swapping

We have discussed three types of queues so far. The reason for such elaborate usage of queues is that I/O activities are much slower than computation, which causes the processor in a uni-programming system to be idle most of the time. However, the arrangement we have seen does not completely solve the problem. It is true that there are multiple processes in memory and the processor can just move to another process when one process is waiting, but the processor is so much faster than I/O that it is common for all the processes in memory to be waiting for I/O. Thus, even with multi-programming, a processor could be idle for most of the time.

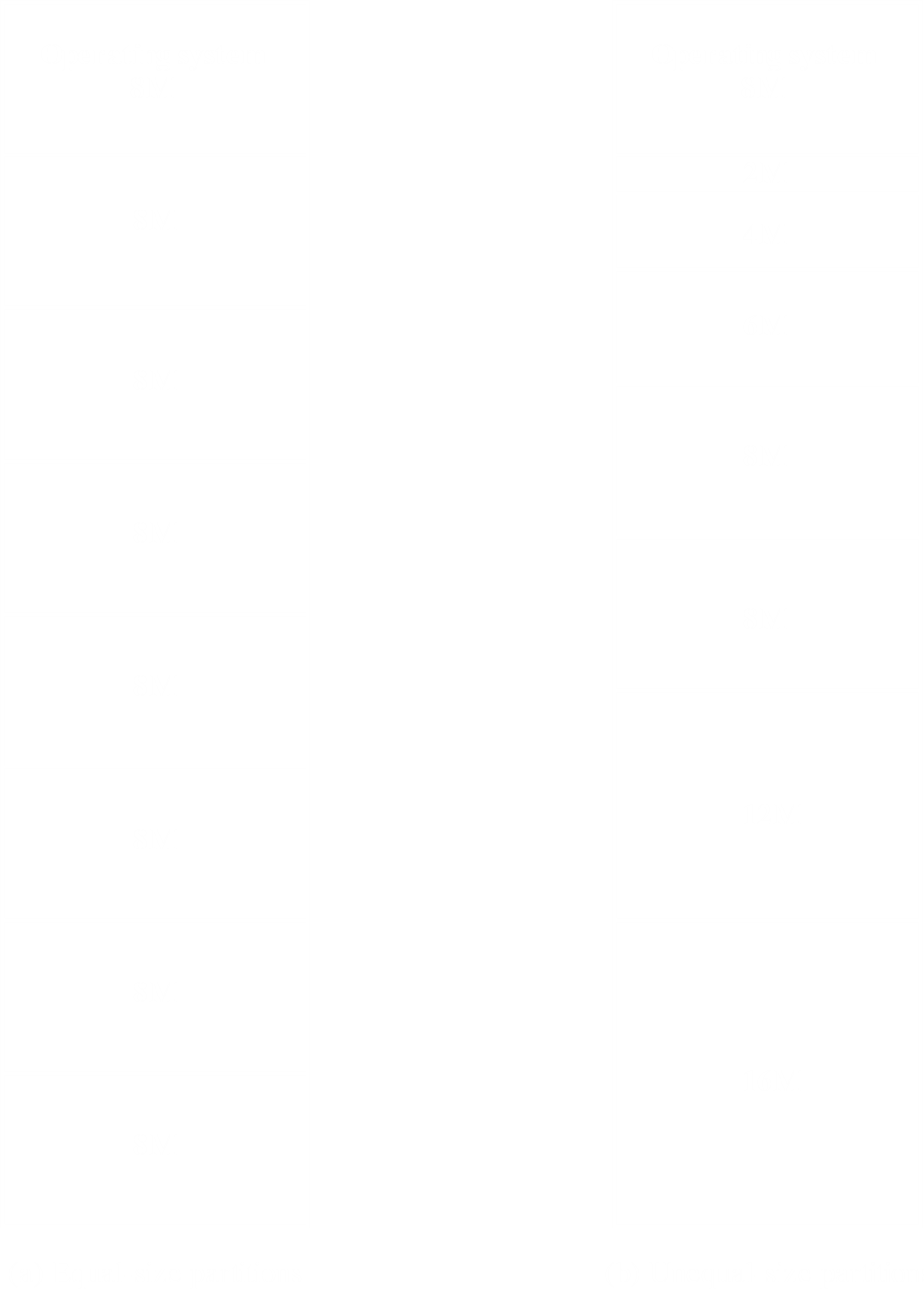
One solution to this problem could be to expand main memory to accommodate even more processes. However, main memory is expensive and the appetite of programs for memory is never ending, so larger memory will result in larger processes not more processes. Another solution is swapping.

The image on the left shows how long-term queues typically work. A process request is brought in one at a time as space becomes available, and as processes are completed, they are moved out of main memory. A situation may arise that none of the processes in memory are in the ready state. Rather than stay idle, the processor swaps one of these processes back out to disk in an intermediate queue, as shown in the image on the right. This queue is for existing processes that have been temporarily removed from memory. The OS then brings in another process from the intermediate queue or a new process request from the long-term queue and continues execution.

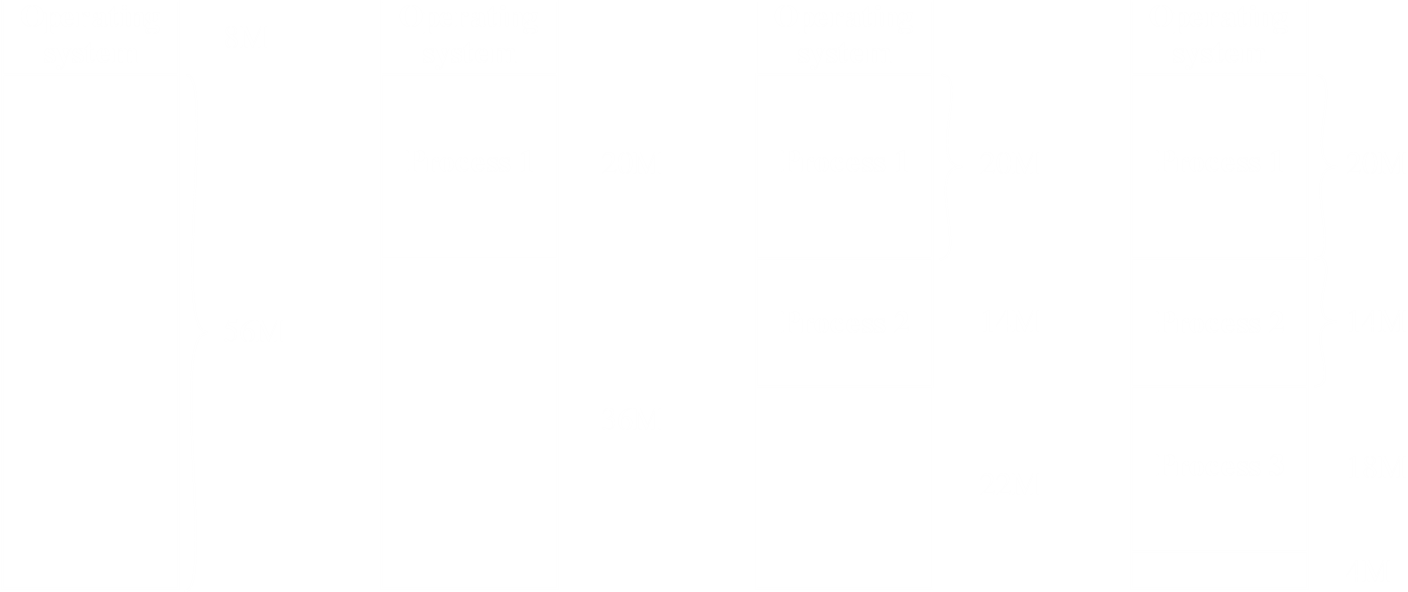
Swapping is an I/O operation though, so there is potential for the problem to get worse instead of better. However, disk I/O is generally the fastest I/O on a system, so swapping usually enhances performance. A more sophisticated scheme, using virtual memory, is even better. Before that can be explained though, we need to look at the concepts of partitioning and paging.

### Partitioning

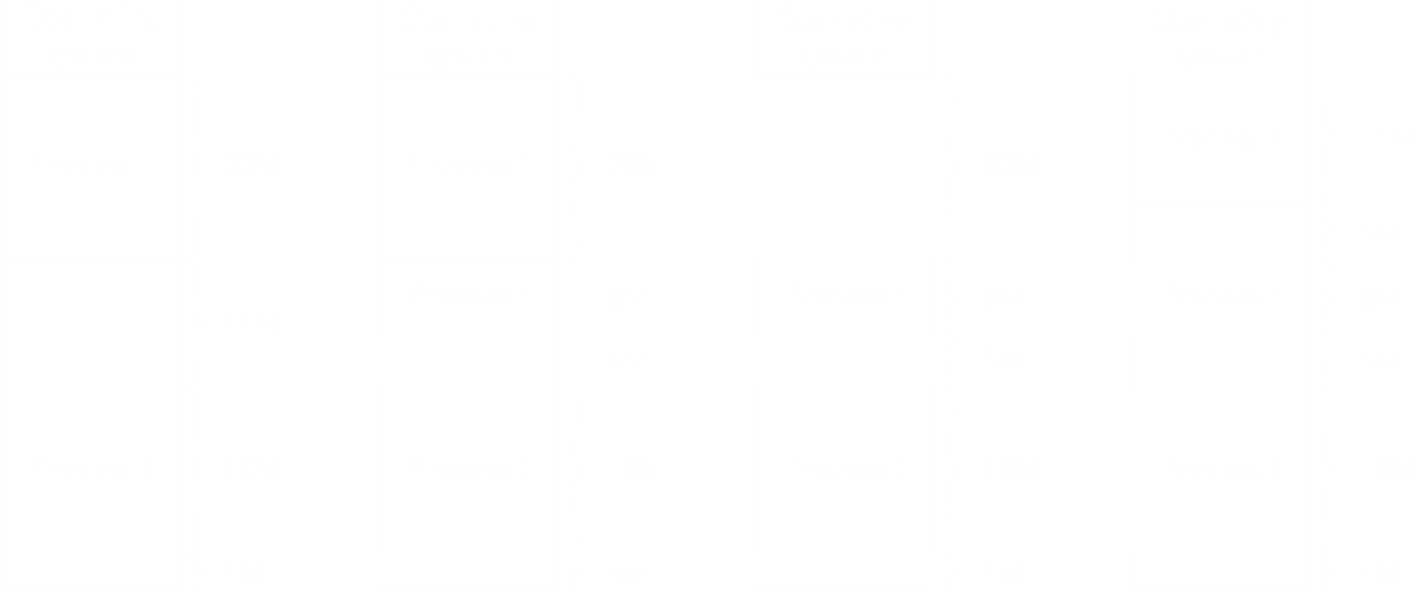


The simplest scheme for partitioning available memory is to use fixed-size partitions. Note that this does not necessarily mean they are equal in size. When a process is brought into memory, it is placed in the smallest available partition that will hold it.

Even with unequal partition, there will be wasted memory. In most cases, a process will not need all the memory provided by the partition. A more efficient system would be to use variable-size partitions.



When a process is brought into memory, it is allocated exactly as much memory as it requires and no more. Though this system starts out well, it leads to a situation where there are a lot of small holes in memory. Over time, memory becomes fragmented and memory utilization declines.



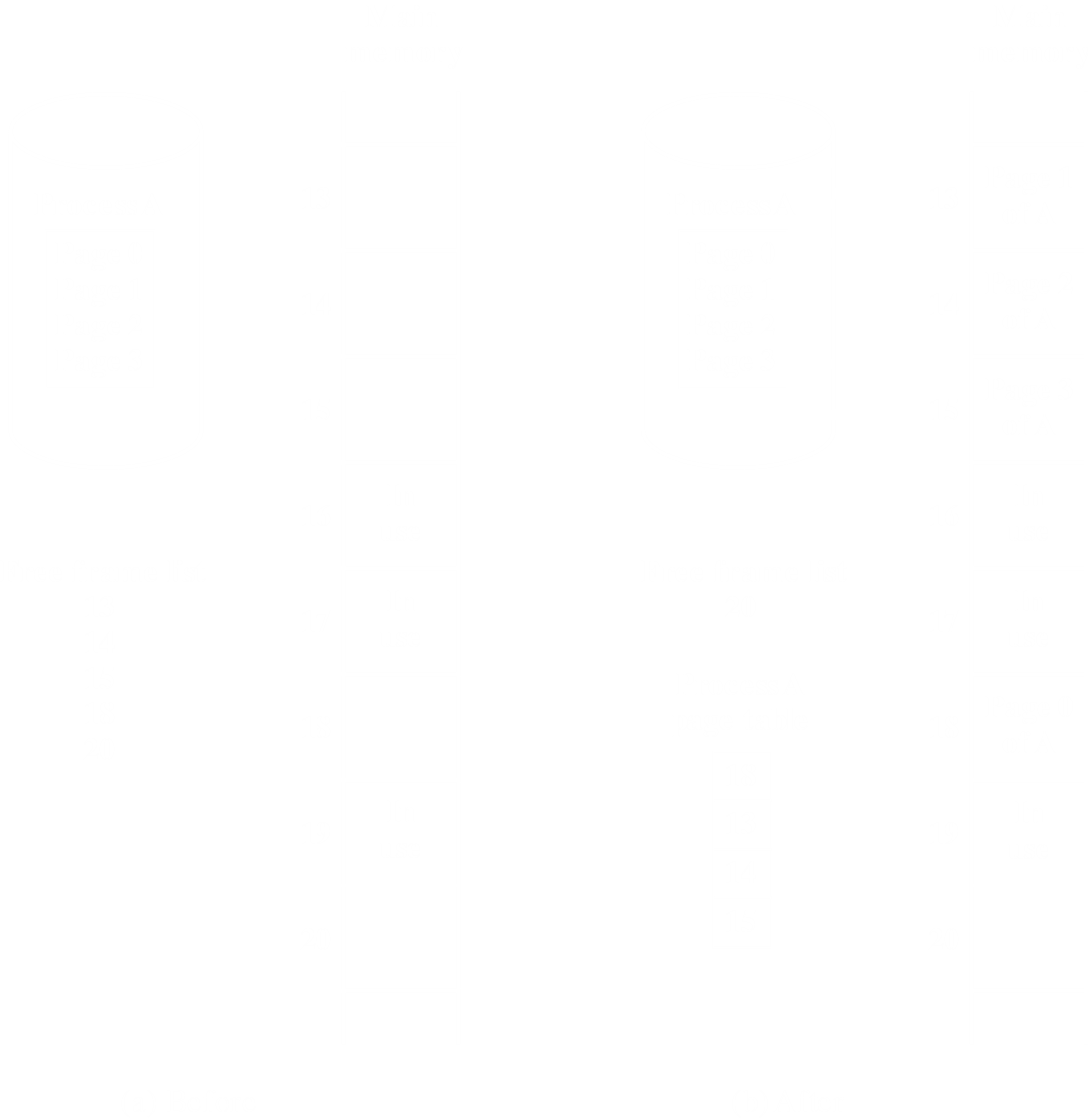
One way to overcome this is compaction. From time to time, the OS shifts the processes in memory to place all the free memory together in one block. This is time-consuming and wasteful of processor time though.

Processes are not likely to be loaded into the same place in main memory each time it is swapped in. Further, with compaction, a process may be shifted while in main memory. A process in memory consists of instructions and data. The instructions contain addresses for the data items and for other instructions, used for branching instructions. These addresses are not fixed and will change each time a process is swapped in. to solve this problem, a distinction is made between logical addresses and physical addresses.

A logical address is expressed as a location relative to the beginning of the program. Instructions in a program contain only logical addresses. A physical address is an actual location in main memory. When the processor executes a process, it automatically converts from logical to physical address by adding the current starting location of the process (its base address) to each logical address. This is another example of a processor hardware feature designed to meet an OS requirement. The exact nature of this hardware depends on the memory management strategy used.

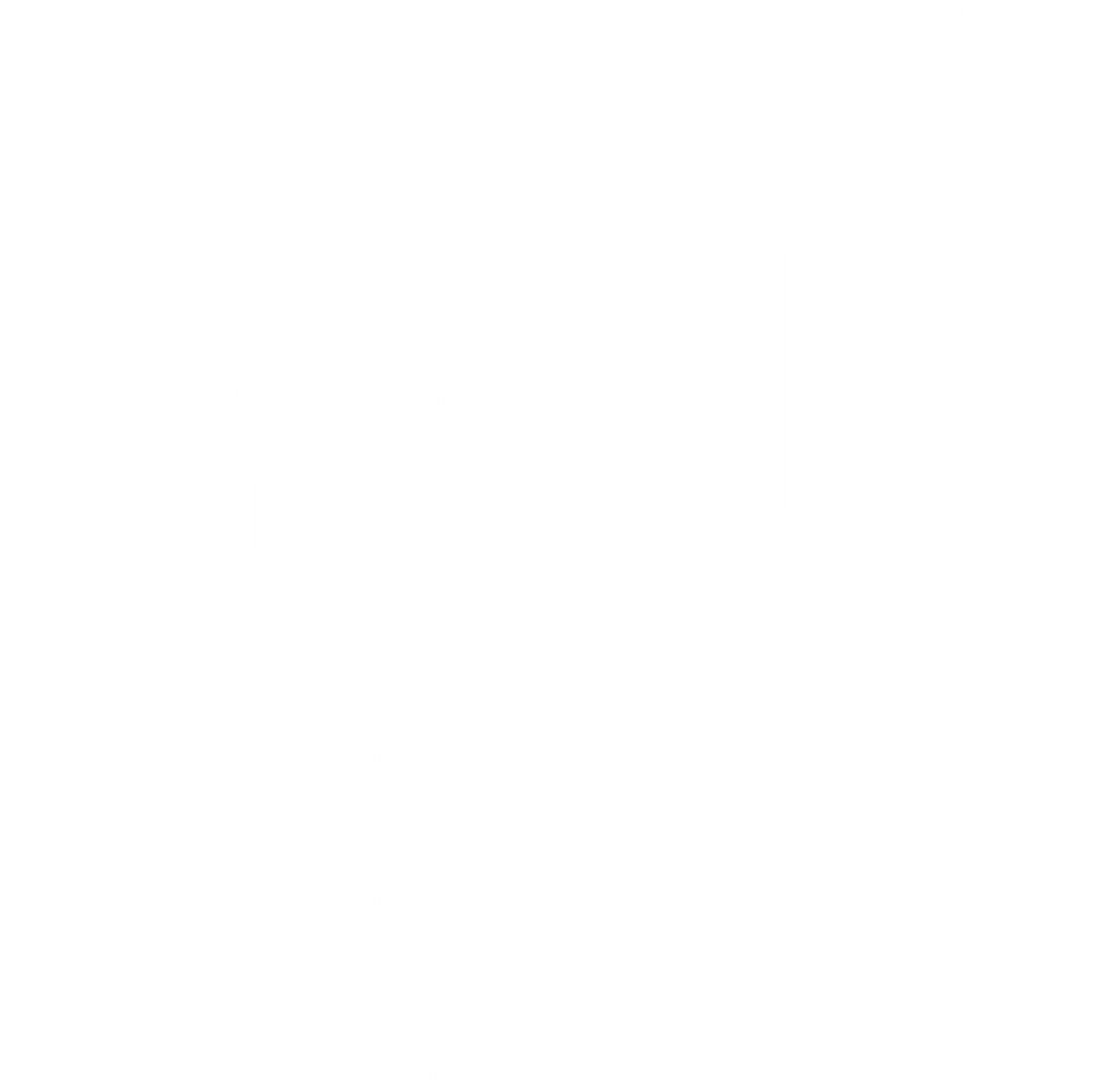
### Paging

Both unequal fixed-size and variable-size partitions are inefficient in the use of memory. Suppose we divide the memory into equal fixed-size chunks that are relatively small and each process into small fixed-size chunks of some size. Then, the chunks of a program, called pages, could be assigned to the chunks of available memory, called frames or page frames. At most, we would be wasting space in memory as a fraction of the last page of a process.



The figure above shows this process. At any time, some frames in memory are in use and some are free. The list of free frames is maintained by the OS. Process A, on disk, consists of 4 pages. When it needs to be loaded, the OS finds 4 free frames and loads the 4 free pages into them.

Notice that the frames do not need to be contiguous, since the concept of logical addresses are used. A simple base address is not sufficient though, so the OS maintains a page table for each process. The page table shows the frame locations for each page of the process. In the program, each logical address consists of a page number and a relative address within the page. The processor uses the page table to find the physical address from the logical one.



This approach solves the problems raised earlier. Main memory is divided into many small equal-size frames, and each process is divided into frame-size pages. Smaller processes need fewer pages and larger ones need more pages. When a process is brought in, its pages are loaded into available frames and a page table is set up.

### Demand Paging

Paging made truly effective multi-programming systems possible. The simple tactic of breaking a process up into pages led to the development of virtual memory. To understand virtual memory, we must first look at demand paging, which simply means a page of a process is only brought in when it is needed, i.e. on demand.

Consider a large process with a long program plus many arrays of data. Over a short period of time, execution is confined to a small section of the program (a subroutine), and perhaps only one or two arrays of data are being used. This is the principle of locality. It would be wasteful to load dozens of pages for that process when only a few are needed before the program is suspended. Thus, to make better use of memory, only the required pages are loaded. If the program branches to an instruction or references data on a page that is not in main memory, a page fault is triggered and the OS is told to bring in the desired page.

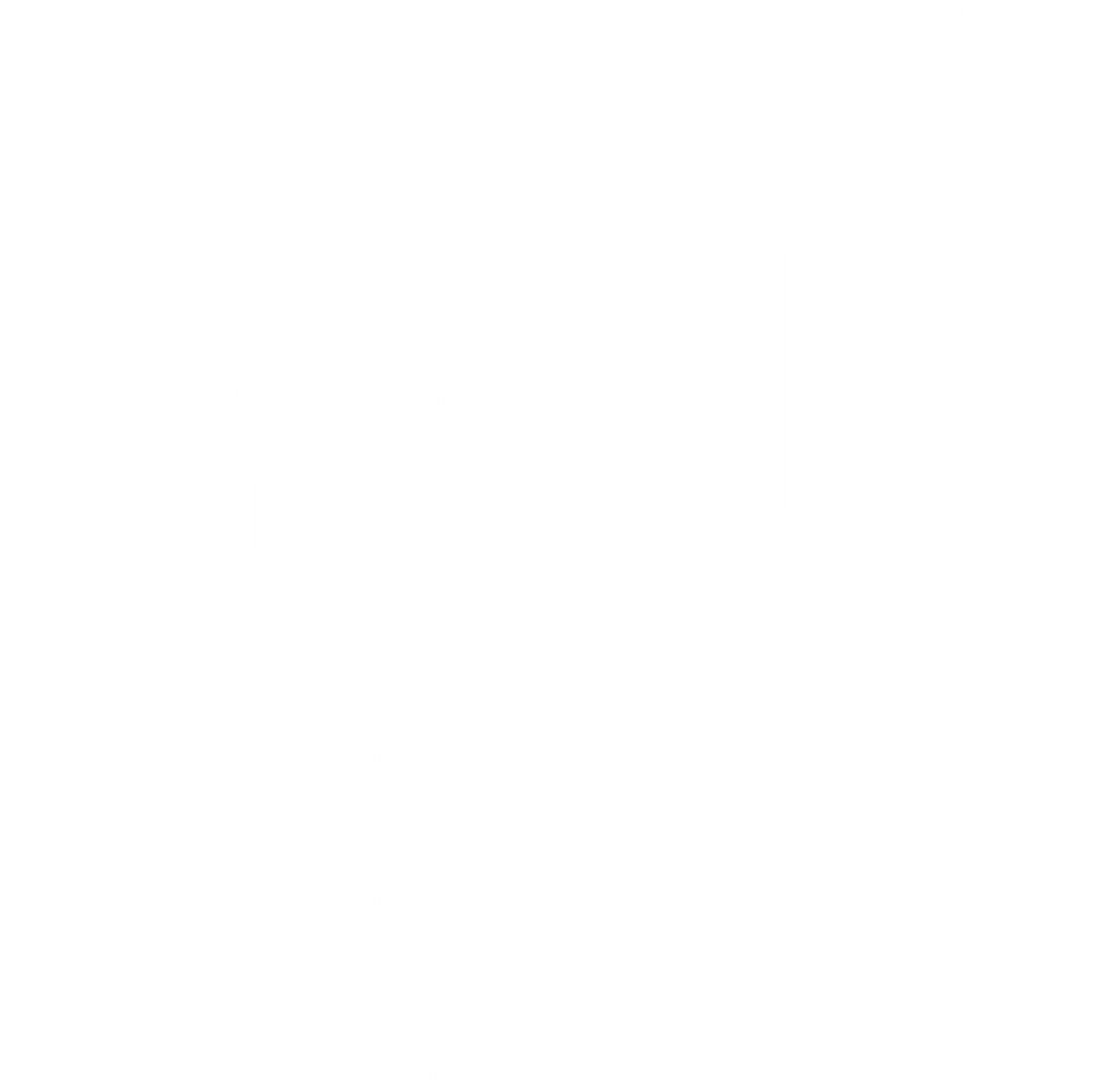
Thus, at any time, only a few pages of a process are in memory, so more processes can be maintained in memory. Furthermore, time is not wasted swapping pages in and out of memory. The OS needs to be clever with how it manages this scheme. When a page is brought in, another must be thrown out. This is called page replacement. If a page is thrown out just before it needs to be used again, it will have to be retrieved again almost immediately. This leads to thrashing, and the processor spends most of its time swapping pages rather than executing instructions. There are a lot of effective algorithms to avoid this, but they will not be discussed here. The OS essentially tries to guess, based on recent history, which pages are least likely to be used in the near future. A potentially effective technique is least recently used (LRU). In practice, LRU is difficult to implement for a virtual memory paging scheme.

With demand paging, since the entire process does not need to be in main memory at once, a process can even be larger than the entire main memory. This lifts one of the fundamental restrictions in programming. Without demand paging, programs that are too large need to be structured so that they can be loaded in pieces. With demand paging, the job is left to the OS and hardware. As far as the programmer is concerned, the only limitation is disk storage.

Because a process executes only in main memory, that memory is referred to as real memory. The programmer or user sees a much larger memory, than which is allocated on the disk. This is called virtual memory. Virtual memory allows for very effective multi-programming and relieves the user of the unnecessarily tight constraints of main memory.

### Page Table Structure

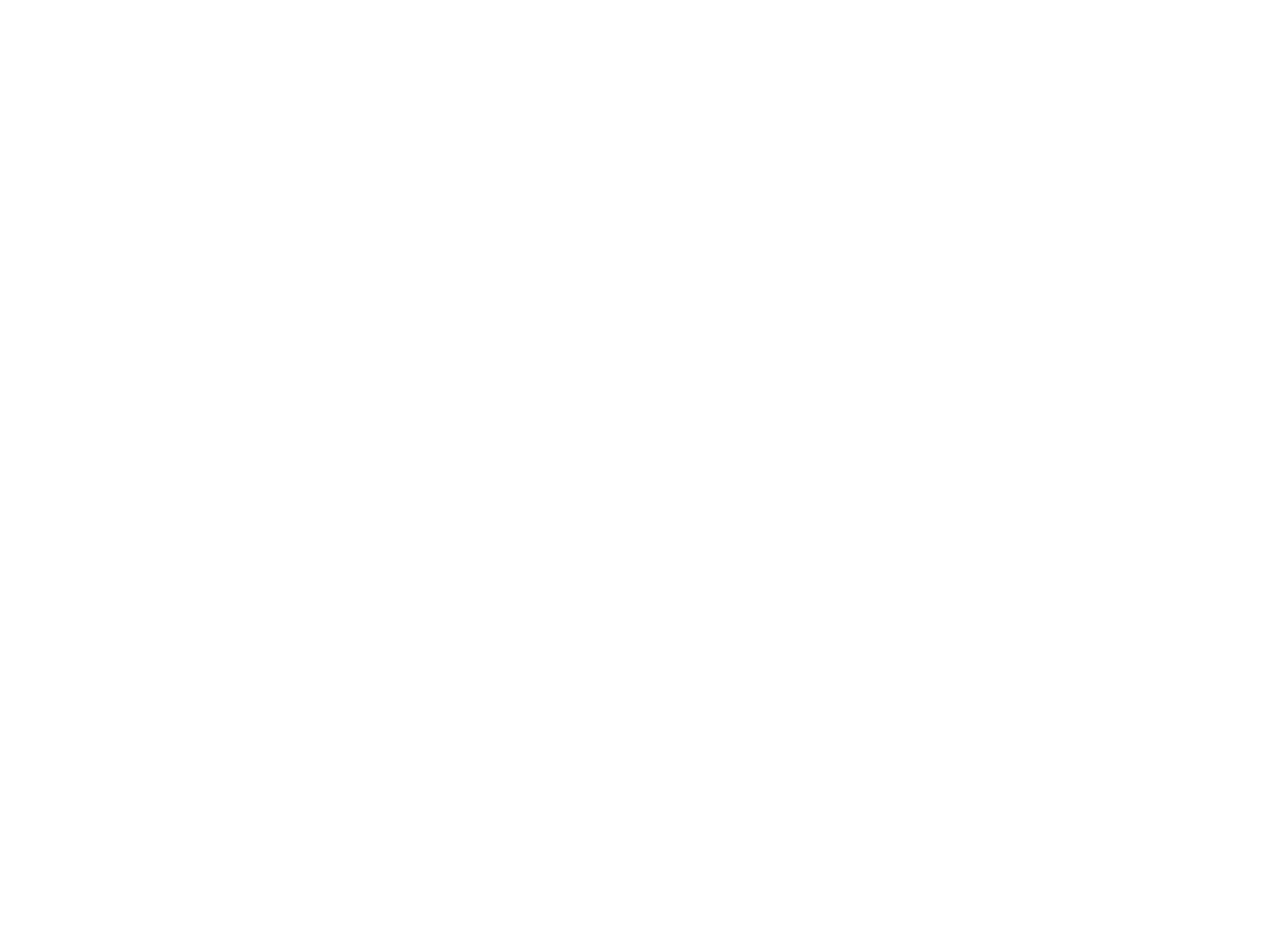
The basic mechanism for reading a word from memory involves the translation of a virtual or logical address consisting of a page number and an offset into a physical address consisting of a frame number and offset, using a page table. Because the page table is of variable length, depending on the size of the process, we cannot hold it in registers. Instead, it is put in main memory to be accessed.



When a process is running, a register holds the starting address of the page table for that process. The page number of a virtual address is used to index that table and look up the corresponding frame number. This is combined with the offset portion of the virtual address to get the desired real address.

In most systems, there is one-page table per process. But each process may occupy huge amounts of virtual memory, which means a humongous amount of page table entries. This could make just the amount of memory dedicated to page tables unacceptable. To overcome this, most virtual memory schemes store page tables in virtual memory rather than real memory. This means the pages tables are also subject to paging. When a process is running, at least a part of its page table must be in main memory, including the page table entry of the currently executing page.

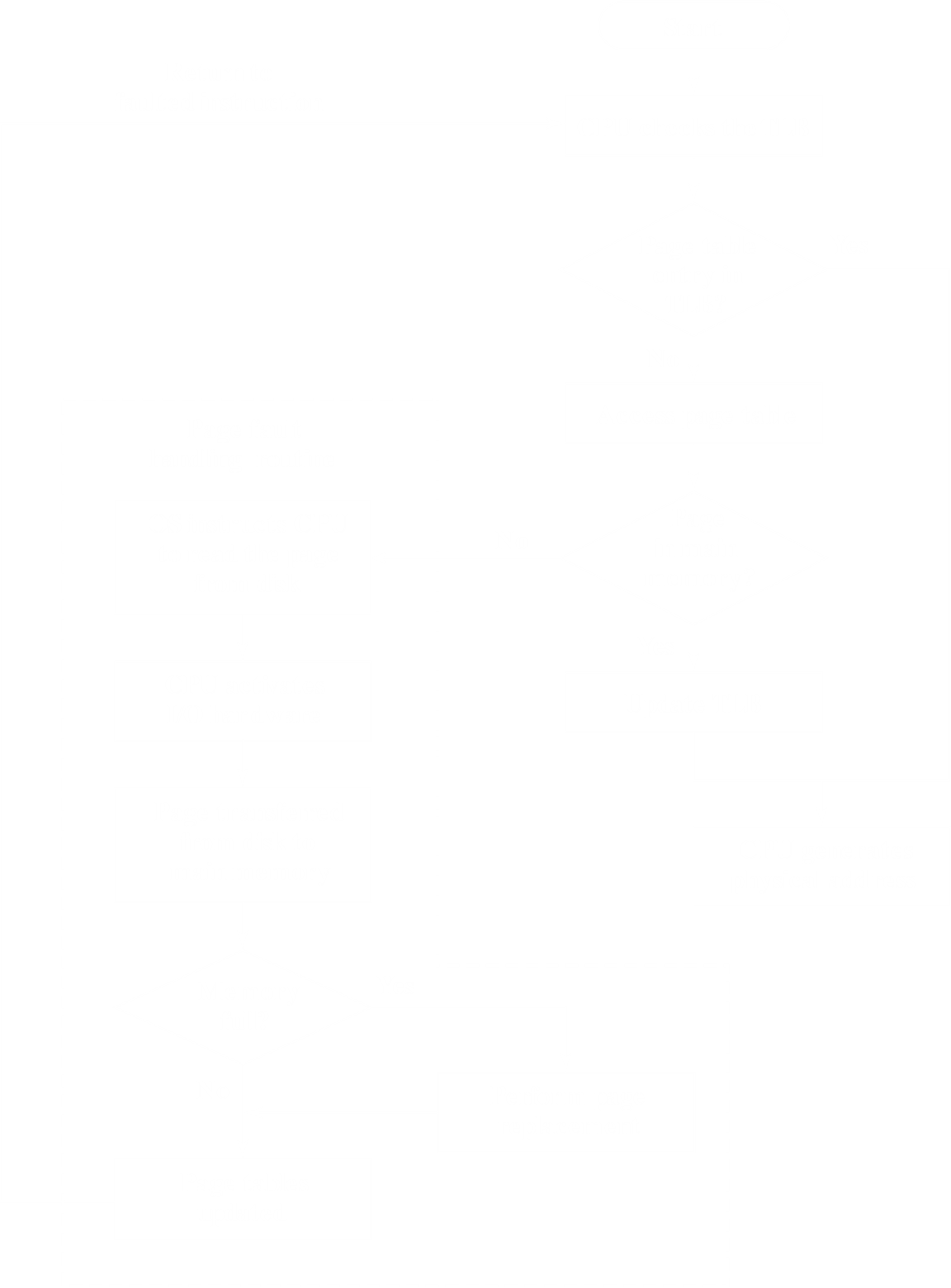
Some processors make use of a two-level scheme to organize large page tables. In this scheme, there is a page directory that points to page tables. If the length of a page directory is and page tables are of length , each process can now have pages. Typically, the maximum length of a page table is restricted to be equal to one page.



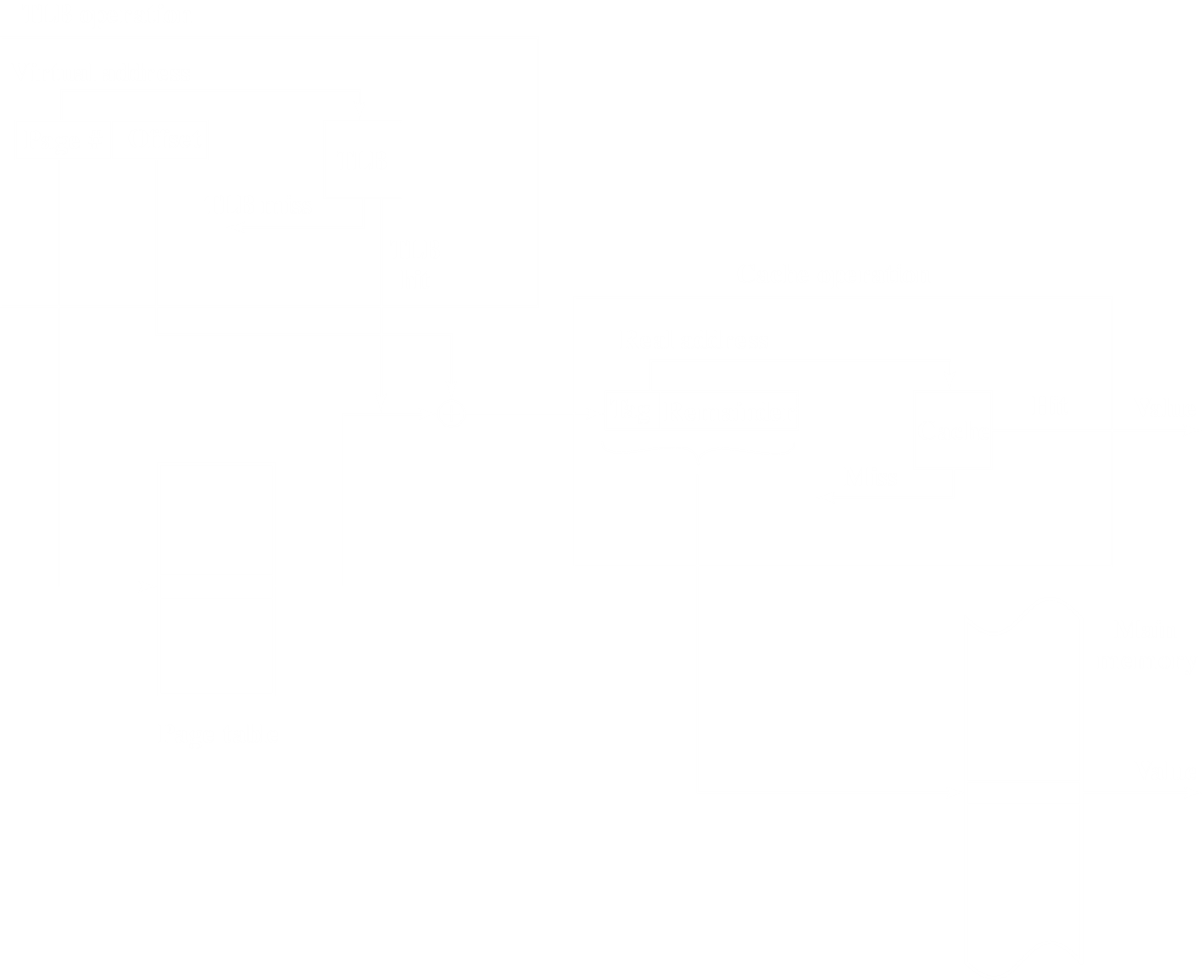
An alternative is to use inverted page tables, shown above. The page number portion of a virtual address is mapped into a hash value using a simple hashing function. The hash value is a pointer to the inverted page table, which contains the page table entries. There is one entry in the inverted page table for each real memory page frame rather than one per virtual page. Thus, a fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported. Since more than one virtual address may map into the same hash table entry, a chaining technique is used for managing the overflow. The chains are typically short, between one and two entries.

### Translation Lookaside Buffer

In principle, a virtual memory reference can cause two physical memory accesses, one to fetch the page table entry and one to fetch the data. Thus, a straight forward virtual memory scheme would double the memory access time. To overcome this, most virtual memory schemes use a special cache for page table entries, called a translation lookaside buffer. This cache functions in the same way as a memory cache and contains the page table entries that have been most recently used. By the principle of locality, most virtual memory references will be to locations in recently used pages, thus involving page table entries in the cache.



Note that the virtual memory mechanism must interact with the cache system (the main memory cache). This is shown below.



A virtual memory address will generally be in the form of a page number, offset. First the memory system consults the TLB to see if the matching page table entry is present. If it is, the physical address is generated. If not, the entry is accessed from a page table. Once the real address is found, which is in the form of a tag and a remainder, the cache is consulted to see if the block containing the word is present. If it is, it is returned to the process. If it is not, it is retrieved from main memory.

The whole process is fairly complicated. The virtual address is translated into a real address, which involved a reference to a page table which is either in the TLB, the main memory or the disk. The referenced word may be in cache, main memory or the disk. In the latter case, the page containing the word must be loaded into main memory and its block loaded into the cache. The page table entry for the page must also be updated.

### Segmentation

Segmentation is another way in which addressable memory can be subdivided. Whereas paging is invisible to the programmer and gives the programmer a larger address space, segmentation is visible to the programmer and makes organizing programs and data convenient and allows the association of privilege and protection attributes with instructions and data.

Segmentation shows memory as segments of variable and dynamic size. Typically, the programmer or OS assigns programs and data to different segments. There may be many program segments for various types of programs, along with various data segments. Each segment may be assigned access and usage rights. Memory references consist of a segment number, offset form of address.

The advantages of this system include:

1. It simplifies handling of growing data structures. If the programmer does not know ahead of time how large a particular data structure will become, rather than guessing, they can assign the data structure to its own segment and the OS will expand or shrink the segment when needed.
2. It allows programs to be altered and recompiled independently without needed an entire set of programs to be relinked and reloaded. Again, this is accomplished using multiple segments.
3. It lends itself to sharing among processes. A programmer can place a utility program or a table of data in a segment and address it from other processes.
4. It lends itself to protection. Because a segment can be constructed to contain a well-defined set of programs or data, the programmer or a system administrator can assign privileges in a convenient fashion.

These advantages are not available with paging. On the other hand, we have seen that paging offers an efficient form of memory management. To combine the advantages of both, some systems are equipped with hardware and OS software to provide both.