In This Module...

- Memory management functions
- Types of memory and typical uses
- Simple analysis of multiprogramming performance
- Memory partitioning, relocation, protection
- Techniques for memory allocation, keeping track of unused memory, fragmentation
- Swap space, Knuth’s 50% rule, unused memory rule
Memory management is a major function of an operating system. Some tasks for which it is responsible include

- Keeping track of how each region of memory is being used.
- Allocating and deallocating memory regions in response to explicit and implicit requests.
- Managing the movement of the contents of memory between primary and secondary storage.
Common Types of Memory

- Random Access Memory (RAM) - general purpose memory that can be read or written.
- Read-Only Memory (ROM) - holds static programs and data like device drivers (e.g. the Basic I/O System, or BIOS, in a PC), bootstrap loaders, and firmware. (Some kinds of ROM can be erased and rewritten – slowly.)
- Flash Memory - often used for things previously stored in ROM on earlier systems, since it can be modified more easily than ROM.
- Other memory (usually special purpose): video RAM (dual ported), cache memory, associative memory (for virtual memory), etc.
There are a number of common approaches to managing memory, and each has advantages and disadvantages.

To understand each of these more fully, we will model the efficiency of execution supported by these memory management methods.
Monoprogramming

- In monoprogramming, a single application program (process) uses the entire machine and runs to completion before another process is allowed to execute.
- Memory is used by the operating system, the program being executed, and device drivers.
Monoprogramming Advantages

- The sole advantage of monoprogramming is its simplicity.
- There is very little software required in the operating system for memory management in this case. The process is given all of the available memory which it can then use as it desires.
Monoprogramming Disadvantages

- Monoprogramming is **wasteful** of processor and memory resources.
  - While the program is performing I/O, the processor is idle (on systems with DMA).
  - If the program doesn’t use all of the available memory, the excess memory in the system is wasted.
- Programs cannot be larger than the size of the available **physical memory** (or the part not used by the operating system).
- Programs must be written to occupy specific absolute memory locations.
So Why Use Monoprogramming?

- In early computer systems (through the second generation) the processor could not execute instructions while an input/output device was being used.
- Thus when a program wanted to read or write a device, the processor sat idle - and remember that I/O devices are very slow when compared to the processor.
- In such systems, there was no real advantage to having programs other than the one being executed in memory at the same time, since only one program could be executed at a time!
Absolute Addressing

- Every physical memory location in the primary memory of a computer system has a unique integer address.
- This address is called an **absolute** or **physical** address.
- In monoprogramming, the program can only use these absolute addresses to reference memory.
- The operating system requires that each program be loaded into a specific range of absolute addresses, and that it will begin execution at a specific absolute address.
The CP/M (Control Program/Microcomputer) operating system for the Intel 8080 (and the Zilog Z80) was the first popular operating system for microcomputers, and was the immediate predecessor of MS-DOS.

- CP/M used monoprogramming, and assumed all programs began execution at location $100_{16}$. 

The CP/M Memory Layout

Interrupt vectors
(256\textsubscript{10} = 100\textsubscript{16} bytes)

Address depends on size of RAM, OS, and drivers

Application programs — begin execution at location 100\textsubscript{16}

Operating system

Location 100\textsubscript{16}

Basic I/O System (Device Drivers)

Location 0

Maximum RAM address is FFFF\textsubscript{16}
CP/M Absolute Addressing Problems

- Most 8080-based systems using CP/M had exactly the memory layout just shown.
- Unfortunately, the Tandy/Radio Shack TRS-80 system had more than 100 bytes of read-only memory starting at location 0.
- As a result, applications for CP/M on the TRS-80 had to be available in special versions that used a different starting address. This was difficult for users and software developers.
With multiprogramming, multiple processes are conceptually executed at the same time.

This permits the operating system to switch the processor to a different process when the current process becomes blocked (for example, while it is waiting for an input/output operation to complete).

The difficulty, of course, is that memory is now required to hold the images of each process that is a candidate for execution.
Multiprogramming Questions

- How effective is multiprogramming? That is, how much more quickly can we execute processes than if monoprogramming was used?
- How much higher is CPU utilization?
- How much higher is memory utilization?
- Is the additional memory needed to support multiple processes worth the cost?
One obvious way to answer these questions is to run the same set of processes using monoprogramming, and then again using multiprogramming. (Of course we would like to repeat the tests using multiple different sets of processes, too.)

Unfortunately, this is a very expensive, time-consuming procedure, and only answers the questions relative to the particular sets of processes used in the analysis.

A more general, mathematical analysis is desired.
One way to model multiprogramming is to use the “birth and death Markov process.” This unfortunately requires a bit more mathematical analysis than we have time for here.

Instead, we’ll use a much simpler model that still gives useful results.
The model we are going to use makes a number of simplifying assumptions.

1. There are $n$ processes in memory.
2. Each process, individually, has a probability $p$ of being blocked.
3. Processes are independent of each other.
4. There is a single processor, and we ignore time spent doing context switches.
Performing the Analysis

- We know the processor (at least in modern systems) will be idle only if all $n$ processes are blocked.
- Using elementary probability, we know that the probability of all $n$ processes being blocked at the same time is $p^n$.
- Similarly, the probability of at least one process not being blocked is $1 - p^n$.
- The chart on the next slide illustrates processor utilization for three different process blocking probabilities: $p = 20\%$, $p = 50\%$, and $p = 80\%$. 

Results for Various I/O Wait Probabilities

CPU Use as Function of Degree of Multiprogramming

- 20%
- 50%
- 80%

Degree of Multiprogramming

CPU Utilization
Scheduling with 80% I/O Wait Time

<table>
<thead>
<tr>
<th>Job</th>
<th>Arrival</th>
<th>CPU Time</th>
<th># of processes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10:00</td>
<td>4</td>
<td>CPU Idle/min</td>
<td>.80</td>
<td>.64</td>
<td>.51</td>
<td>.41</td>
</tr>
<tr>
<td>2</td>
<td>10:10</td>
<td>3</td>
<td>CPU Busy/min</td>
<td>.20</td>
<td>.36</td>
<td>.49</td>
<td>.59</td>
</tr>
<tr>
<td>3</td>
<td>10:15</td>
<td>2</td>
<td>CPU/process/min</td>
<td>.20</td>
<td>.18</td>
<td>.16</td>
<td>.15</td>
</tr>
<tr>
<td>4</td>
<td>10:20</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ CPU \text{ usage per job} = real-time \cdot (1 - 0.8^n) / n \]

<table>
<thead>
<tr>
<th>Job</th>
<th>CPU usage per job</th>
<th>Time (relative to arrival of job 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.9</td>
<td>= 4 min</td>
</tr>
<tr>
<td>2</td>
<td>.8</td>
<td>= 3 min</td>
</tr>
<tr>
<td>3</td>
<td>.3</td>
<td>= 2 min</td>
</tr>
<tr>
<td>4</td>
<td>.3</td>
<td>= 2 min</td>
</tr>
</tbody>
</table>
With monoprogramming, memory management was simple: give all available memory to the job and we’re done.

With multiprogramming we must decide how to arrange the memory images of the jobs in the primary memory of the system. With \( n \) jobs there are \( n! \) orderings of the jobs in memory. We must also consider what happens when one of the jobs finishes - can we fit a new job in the memory space just vacated?
Fixed Partitioning of Memory

- One common early approach to the problem was to statically divide the available memory into a number of regions, or partitions, with the size of each being fixed (at least until the next time the system was rebooted).

- This approach was used in the IBM OS/MFT system (Multiprogramming with a Fixed number of Tasks) operating system (now largely obsolete).
Characteristics of Fixed-Size Partitioned Systems

- Each job (with a maximum memory requirement specified) is ideally run in the smallest unused partition that meets its needs.
- Underutilization of memory can still occur, since job sizes don’t necessarily exactly match partition sizes. Suppose we have two 40K partitions and a 39K and a 41K job. Only the 39K job can be run, wasting 41K.
- Concern must also be given to memory protection, since a program in one partition might access memory in another partition by accident or on purpose. (This problem also exists in monoprogramming, with the OS susceptible to access or modification by the user.)
Scheduling for Fixed-Size Partitioned Systems

There are two obvious scheduling approaches for a system with a fixed number of partitions:

- use a separate queue for each partition, or
- use a single queue for all partitions.

These choices are easily observed in places where people queue. In grocery stores it’s common to pick a queue for a particular server (checker) and stay in it until serviced. Or in other places (e.g. post offices and banks) customers join a single queue and use the first server that becomes available.

But for partitioned memory systems, neither of these approaches is ideal. For example, a small job may be submitted when a partition much larger than its memory requirements is available. Should the small job get processed in the larger partition, or should it wait for a smaller partition?

Some schedulers for these systems may have the ability to “look ahead” in the queue(s) to potentially reorder the submissions and improve performance.
Single and Multiple Input Queues

Multiple Input Queues

Partition 1

OS

100K

200K

400K

700K

Partition 1

Partition 2

Partition 3

Partition 4

OS

Single Input Queue

Partition 1

Partition 2

Partition 3

Partition 4
Fixed Partitioning Scheduling Example

- Assume four jobs are submitted, as follows:
  - Job 1, 35K, runs for 4 min real time.
  - Job 2, 20K, runs for 3 min real time.
  - Job 3, 40K, runs for 10 min real time.
  - Job 4, 60K, runs for 7 min real time.

- Assume two partitions of 40K and 60K.

- If the jobs are run in the order given, the average turnaround time is 10 min.

- If run in the order 4, 2, 1, 3, the average turnaround time is reduced to 8.5 min.
Since each partition begins at a different memory location, jobs must:
- be produced to run at a specific location (corresponding to a particular partition),
- be appropriately adjusted (by the OS) just before execution, or
- be able to dynamically adjust to the location at which they are loaded.

The first two approaches require extra processor time, and the third approach requires extra hardware. The term used for this problem is relocation.
Fixed Partition Sizes: Advantages, Disadvantages

Advantages:
- Easy to implement.
- Provides multiprogramming (faster turnaround, better use of memory and processor).

Disadvantages:
- Scheduling can be complicated.
- Memory resources are still wasted.
- Job size is limited to physical memory size.
- Extra hardware and/or software are needed for relocation and protection.
Generic Application Building
Linking vs. Loading

- Linking is the process of replacing external references in an object module with their definitions.
- Loading is the process of placing a program in memory for execution.
- Relocation may take place in either or both of these actions.
There are two possible ways to prepare a program for execution:

- Specify absolute addresses for the instruction and data objects; or
- Specify "generic" addresses for the instruction and data objects, identify external objects that need to be referenced, and identify all of these in a standard way.

It is the job of relocation to make one of the "generic" executables (or object files) capable of being executed at a specific memory address.
To achieve relocation, OS/MFT modifies the addresses in instructions when the program is loaded to reflect the location at which the program is loaded.

This has some obvious disadvantages:

- The loader must mark those instructions needing modification, and
- The modification of the program is a slow operation.
Another approach to relocation on IBM mainframes causes a base register (one of the processor’s general registers) to be set to the address at which the program is loaded. The instructions in the program then use this base register much like an index register when they need to reference memory.

Disadvantages:

- Additional base registers are required if the offset field in an instruction isn't large enough to reference all locations in the program.
- The approach consumes one of the processor’s general registers.
- A program can’t be easily relocated once it’s started.
Perhaps the most general approach is to have a special processor register used as the base register, and automatically add its content to every memory address generated by a program.

This register is protected from modification by the user and is, in fact, entirely transparent to the user's program.

The user's program now appears to be addressing a region of memory starting at location 0.

The special base register must be included in the registers manipulated by a context switch.
In the IBM System360 (and later models), each 2K byte block of memory has a special 4-bit code called the lock. Only supervisor-mode programs can alter the content of these locks. All locks for each program’s memory are set to a unique value.

A four-bit key is present in the Program Status Word (or PSW, essentially a register containing the next instruction address, interrupt enable information, and so forth). The PSW is included in the context of a process.

During memory references, if key = lock, or key = 0, then the memory access is permitted. Otherwise, a fault is generated.
Another Memory Protection Scheme

- Another approach is to add a special limits register (which can only be set by supervisor mode programs).
- When an instruction references memory, the address of memory operand is compared with the limits register.
- If the memory operand’s address is greater than or equal to the address specified in the limits register, a fault is generated.
Virtually all systems include some hardware support for relocation using a base register approach, either using one (or more) of the system’s general purpose registers as a base register (IBM approach) or using a special purpose base register. Regardless of the approach chosen, the hardware is more expensive and the operating system is more complicated than systems with no relocation support.

Both protection approaches (lock and key, and the limits register) require special hardware, and that implies additional expense in producing the processor.

Of course it also means that managing programs during execution is more complicated, requiring the setting of locks and keys, or setting the limits register appropriately.

As suggested on the next slide, some systems may have multiple pairs of base and limits registers. This hints at the possibility of allowing non-contiguous memory regions for processes, an issue we will continue to consider.
Use of Multiple Relocation and Protection Registers
Both of the memory management approaches we’ve just considered (monoprogramming and fixed-size partitions), and others, can be enhanced by allowing swapping.

Swapping is the process of moving a process between primary memory and secondary storage (recall two-level scheduling).

It is used to increase the probability that a process will be ready to run when the processor would otherwise be idle.

Swapping is not the same technique as virtual memory (as we will see).
Instead of using a fixed number of partitions with fixed sizes, we could allow a variable number of partitions with sizes dynamically adjusted to match the needs of the next process to be run.

IBM used this in its OS/MVT (Multiprogramming with a Variable number of Tasks) system.

After such a system has been operating for some time we will discover a number of holes, or small unused memory regions between processes.

These regions are called holes because they are not large enough to contain a program.
Collectively, the holes in memory might be large enough to allow the execution of another program. That is, the total of the sizes of the holes might be greater than or equal to the size of another program that we wish to run.

But our programs assume they will be executed in a contiguous region of memory, and the holes are not contiguous. (If they were contiguous, then they wouldn’t be individual holes.)

Compaction is a technique that might be used to coalesce all the holes into one larger region of memory. This larger region might be large enough to run another program.

To perform compaction we must move the occupied memory regions so they are adjacent to eliminate the holes.

Compaction can consume a lot of processor time during which processes cannot execute. (Some systems actually used ancillary processors to do compaction.)
Variable-Sized Partitions: Advantages, Disadvantages

- Advantages:
  - Conceptually less wasted memory
  - Scheduling possibly simpler

- Disadvantages:
  - Some memory resources still wasted (holes)
  - Job sizes still limited by the physical memory size
  - Extra hardware and/or software still needed for relocation and protection.
How Much Memory Should Be Allocated?

- Many programs have predictable memory requirements: code, static data, and stack (to hold procedure invocation records and local variables).
- However many others have less predictable memory requirements due to the need for dynamic storage allocation and recursion.
- As a result, processes may request execution in a partition larger than their static memory requirements.
- How much memory should be requested for the execution of a program? That is, for a given program, what size partition should be requested?
To accommodate the potential for growth of some parts of a program’s memory allocation, the code, static data and heap (dynamic data) are usually stored (in order) in the lowest part of a memory region.

The stack is then placed at the high end of the region, growing downward as items are pushed, and contracting upward as items are removed.

This allows expansion of the heap and the stack into the region between them.
Arranging for Heap and Stack Growth

<table>
<thead>
<tr>
<th>Code (Text)</th>
<th>Data (and BSS†)</th>
<th>Heap (Dynamic)</th>
<th>Stack (Dynamic)</th>
</tr>
</thead>
</table>

Lowest Memory Address

Area for growth of heap and stack

Highest Memory Address

† BSS = uninitialized variables
Keeping Track of Memory Usage

- There are three common ways to keep track of memory usage:
  - bit maps
  - linked lists
  - buddy system (variation of linked lists)

- Memory is usually managed in fixed-sized blocks (e.g. 2K bytes). Block sizes may be hardware dependent.

- The basic goals of each system are to:
  - keep track of all used and unused memory regions,
  - support the allocation of a contiguous unused region of n blocks of memory (for applications and the kernel), and
  - handle the deallocation (freeing) of a region of memory.
Bit Maps

- We can use an array of bits is used to keep track of used/unused blocks of memory. We might choose to set a bit to 0 to represent an unused block, with a 1 bit meaning the block is used.
- The position of the bit in the array (its subscript) is directly related to the address of the block.
- The size of the block has several effects.
  - Larger block sizes yield smaller bitmaps, but memory will likely be wasted in the last block for each region.
  - Smaller block sizes result in larger bitmaps, but less waste will occur in a region's last block.
**Bit Maps: Advantages, Disadvantages**

- **Advantages:**
  - easy to implement,
  - easy to deallocate a region, and
  - the size is dependent only on the size of primary memory and of a block.

- **Disadvantage:**
  - it is time-consuming to allocate a contiguous region of size $n$, since the system must search the bit map for $n$ consecutive 0 bits.
Linked Lists

- The basic idea of using linked lists for memory management is to maintain a list of free regions of contiguous memory and another list of occupied regions.
- The free list can be ordered in various ways: unordered, ordered by the region’s memory address, or ordered by the region’s size.
- We explicitly assume region sizes are random.
- Allocation of a region can be done using any of many different algorithms (e.g. first fit, best fit, next fit, worst fit, quick fit).
When allocating a region, it is advantageous to have a list of free regions ordered by size. But when a region is deallocated, it is necessary to consider joining it to any neighboring unused regions (to make a larger hole), and this is most easily done if the list is ordered by memory address.

One set of list nodes can be ordered in several ways by having multiple pointers in each node. Doubly-linked lists are then necessary.

The unused regions of memory can themselves be used as the list nodes.
Scan the free list from its head, looking for the first unused region that satisfies the allocation request (that is, it is larger than or equal to the size of the request).

Then split that region, if necessary, to produce one region of the requested size. The other (unused) region is placed back on the list.

Advantages:
- Easy to implement
- Fast
- Free list does not need to be ordered

Disadvantage:
- Produces lots of very small holes
Next Fit Allocation Technique

- This approach is similar to first fit, but each new search for an acceptable unused memory region begins (in the list) where the last one ended.

- Disadvantage
  - In simulations, it produces slightly worse results than first fit.

- Expected advantage:
  - It should distribute the holes more evenly throughout memory.
**Best Fit Allocation Technique**

- Scan the entire list to locate the smallest unused region that will accommodate the request for memory.

- Disadvantages:
  - it is potentially very slow; or
  - requires the list to be ordered on size; which
  - complicates the deallocation algorithm.

- Expected advantage:
  - It should reduce the number of holes.
Worst Fit Allocation Technique

- Pick the largest available hole to satisfy each allocation request, splitting it as necessary into the allocated region and a new hole.

- Disadvantages:
  - Simulation results for this technique are negative.

- Expected advantage:
  - The basic idea is that this technique produces many fewer unusable holes.
Quick Fit Allocation Technique

- Maintain multiple lists, each list having holes of the same common sizes. To satisfy an allocation request, look first in the list with the appropriate size, then in lists of larger unused regions. (Compare with the buddy system, which we’ll discuss next)

- Disadvantage:
  - Deallocation is costly, since all lists must be scanned to find neighboring unused regions.

- Advantage:
  - Allocation is fast.
In the buddy system, multiple lists recording the addresses of holes of different sizes are maintained.

The size of each hole must be a power of 2 blocks (2, 4, 8, etc.)

An allocation request (say, n blocks) is rounded up to the next larger power of 2, then a search is made for a block of that size on the appropriate list.

If an appropriate block is available, the entire block is allocated for the request.

If an unused region of the required size is not available, then the first available larger region is split into two equal pieces (each still has a size that is a power of two).

If one of the new blocks (from the split) is still too large, it also is split, and so on, until one of the split blocks is the correct size for the allocation. The unused half of each split block is placed on the appropriate list.
Suppose we have the following lists of free memory regions in a system that has 128 blocks of memory (each marked “F”):
- Size 8: one region free starting at block 32
- Size 16: one region free starting at block 16
- Size 64: one region free starting at block 64

```
0  16  32  40  48  64  128
```

```
  F
F   16
  F   8
     F
     64
```
To allocate a block of size 20:

1. Round the request up to next power of 2 ( = 32).
2. Look for a size 32 hole (none found).
3. Look for a size 64 hole (one found at 64).
4. Split the size 64 hole into two size 32 pieces, one at 64 and one at 96.
5. Allocate either size 32 hole for the request, and place the other on the list of size 32 holes. Here we allocate the hole at address 64.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>16</td>
<td>F</td>
<td>8</td>
<td>Allocated</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>96</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128</td>
</tr>
</tbody>
</table>
Finding the “Buddy” of a Block

The most significant feature of the buddy system is the speed with which it can perform allocations and deallocations. We have just seen the simplicity of the allocation algorithm.

The deallocation algorithm can quickly find the address of “buddy” of a block being freed, making it easy to join a block being freed to other free blocks. Here’s how we find the buddy’s address.

Given the address of a block \( A \) and its size \( S \). We will find \( B \), the address of its “buddy”.

1. Compute \( A/S \) (always an integer). This can be done efficiently using by shifting \( A \) right some number of bits, since \( S \) is always a power of two. For example, assume \( S = 2^k \). Then \( A/S = A >> k \).
2. If the result of \( A/S \) is even (that is, the low-order bit of \( A >> k \) is 0), then \( B = A + S \). Otherwise, \( B = A - S \).
Examples of Finding a Buddy’s Address

- A block of size 4 is at 011100100\(_2\). Since 4 = 2\(^2\), we shift 011100100\(_2\) right 2 bits to obtain 0111001\(_2\), an odd number. Thus the buddy is at 011100100\(_2\) − 100\(_2\) = 011100000\(_2\).

- A block of size 8 is at 10010000\(_2\). 8 = 2\(^3\), so we shift 10010000\(_2\) right 3 bits to obtain 10010\(_2\), an even number. The buddy is at 10010000\(_2\) + 1000\(_2\) = 10011000\(_2\).

- Of course, we could have just divided by 4 and 8 in these cases; shifting is faster, though.
To deallocate a block $A$ of size $S$:

1. Find $B$, the address of the buddy for the block at address $A$ of size $S$.
2. If the block at address $B$ is on the free list for size $S$, then coalesce (join) it with block at address $A$ (the block being deallocated, which is taken off the size $S$ list) into a size $2 \times S$ block. Then make that new block the one being deallocated (that is, set $A = \min(A, B)$ and $S = 2 \times S$) and repeat from step 1.
3. If the block at address $B$ (the buddy) is not free, then put the block at address $A$ on the free list for size $S$ blocks.
Buddy System: Advantages, Disadvantages

- Advantages:
  - Relatively easy to implement
  - No unused memory in holes
  - Fast allocation and deallocation

- Disadvantage:
  - Allocations always have a size equal to a power of 2 blocks, which may be larger than needed to satisfy a particular request. Thus memory will likely be wasted.
Buddy System: Application

- The buddy system is used to keep track of the physical memory used by the kernel in Linux systems.
- The smallest allocatable region in Linux is a physical page (discussed in the next module); a page is 4K bytes in the Intel x86 systems.
Memory Fragmentation

- We have seen that every technique for managing memory results in some regions of memory being unused.
- The unused memory **inside** allocated regions (as in the buddy system) is due to **internal fragmentation**.
- Unused holes in memory **outside** any allocated region are due to **external fragmentation**.
In swapping systems (e.g. two-level systems), space must be allocated on secondary storage (disk) for process images.

Swap space must be allocated as an integral number of disk blocks, potentially resulting in internal fragmentation of disk space.

Swap space allocated to a process may also need to grow dynamically as the process itself allocates additional memory (between the heap and the stack).

Swap space has the advantage of not needing to be contiguous, although contiguous swap space is preferred.
Averaged over time, there must be half as many holes in memory as processes.

Discussion. Consider an arbitrary process allocation in the middle of memory when the system reaches equilibrium. During its lifetime, half of the operations on the region above it are allocations, and half are deallocations.

Thus the average number of holes in memory is half the number of processes.
We wish to know what fractional part of memory is wasted in holes.

Define the following variables:

- \( f \) = the fractional part of memory occupied by holes (this is the quantity we’re trying to determine)
- \( s \) = average size of a process
- \( n \) = number of processes
- \( m \) = size of memory
- \( k \times s \) = average hole size, for some \( k > 0 \)
Developing the Unused Memory Rule

- How much total space is occupied by holes?
- There are \( n/2 \) holes (using the 50% rule), each of size \( k \times s \).
- We also know that all memory not occupied by processes is in holes, and they therefore occupy size \( m - n \times s \).
- If we equate these two expressions for the space occupied by holes, we get this:
  \[
  (n/2) \times k \times s = m - n \times s
  \]
- Solving for \( m \), we see
  \[
  m = n \times s \times (1 + k/2)
  \]
The fraction of memory in holes, $f$, is the number of holes, $n/2$, times the average hole size, $k \times s$, divided by the memory size $m$. Solving for this we obtain the following:

$$f = \frac{n \times k \times s/2}{m} = \frac{n \times k \times s/2}{n \times s \times (1 + k/2)} = \frac{k}{k + 2}$$

This result is much simpler than expected, and makes it very easy to determine the fractional part of memory that’s wasted due to holes.
In the next module we will conclude our study of memory management techniques.

We will consider methods to minimize fragmentation, eliminate the need for contiguous memory, and remove the physical memory size limit on programs.