Q1) Consider a system that has 8 tape drives, 9 modems, and 8 printers.

Assume there are 5 processes, P0 - P4, with the following overall requirements:

- P0 requires 0 tape drives, 5 modems, and 2 printers;
- P1 requires 2 tape drives, 4 modems, and 6 printers;
- P2 requires 8 tape drives, 6 modems, and 2 printers;
- P3 requires 6 tape drives, 1 modems, and 1 printers; and
- P4 requires 2 tape drives, 8 modems, and 2 printers.

Further assume the processes are already holding some of their required resources (so, for example, P0 only requires a further 0 tape drives, 2 modems, and 0 printer):

- P0 is currently holding 0 tape drive(s), 3 modem(s), and 2 printer(s);
- P1 is currently holding 2 tape drive(s), 1 modem(s), and 1 printer(s);
- P2 is currently holding 0 tape drive(s), 0 modem(s), and 2 printer(s);
- P3 is currently holding 4 tape drive(s), 0 modem(s), and 0 printer(s); and
- P4 is currently holding 1 tape drive(s), 3 modem(s), and 2 printer(s).

Use the Banker's Algorithm to determine if this state is safe. Show each step of the process.

Answer:

Here, according to the question, we need to check whether the current resource allocation for the system is in a safe state using the Banker’s Algorithm. The following is the process for determining the safe state.

Total Resources:

- ➢ Tape Drives: 8
- ➢ Modems: 9
- ➢ Printers: 8

Step 1: Define the Available Resources

We can start by calculating the available resources by subtracting the currently held resources from the total system resources.

Available = Total Resources - Currently Held

- Tape drives: 8 total - (0+2+0+4+1) held = 1 available
- Modems: 9 total - (3+1+0+0+3) held = 2 available
- Printers: 8 total - (2+1+2+0+2) held = 1 available

Thus, the available vector is:

- Available: [1 tape drive, 2 modems, 1 printer]

Step 2: Define the Need Matrix

We can now calculate the need matrix, which shows how many more resources each process requires to complete its execution:

Need = Maximum Requirement - Currently Held

So,

- P0:

  Max = (0, 5, 2)

  Held = (0, 3, 2)

  Need = (0-0, 5-3, 2-2) = (0 tape drive, 2 modems, 0 printer)

- P1:

  Max = (2, 4, 6)

  Held = (2, 1, 1)

  Need = (2-2, 4-1, 6-1) = (0 tape drive, 3 modems, 5 printer)
- P2:

  Max = (8, 6, 2)

  Held = (0, 0, 2)

  Need = (8-0, 6-0, 2-2) = (8 tape drive, 6 modems, 0 printer)
- P3:

  Max = (6, 1, 1)

  Held = (4, 0, 0)

  Need = (6-4, 1-0, 1-0) = (2 tape drive, 1 modems, 1 printer)
- P4:

  Max = (2, 8, 2)

  Held = (1, 3, 2)

  Need = (2-1, 8-3, 2-2) = (1 tape drive, 5 modems, 0 printer)

Step 3: Banker's Algorithm Procedure

- Look for a process whose needs can be met with the currently available resources (Available (1, 2, 1)).
  - P0: Needs (0, 2, 0). Available (1, 2, 1). P0 can be satisfied.
- Assume P0 finishes. Release its held resources back to the available pool. P0 held (0, 3, 2), so the new available resources are:
  - New Available = (1+0, 2+3, 1+2) = (1, 5, 3)
- Look for the next process that can be satisfied with the new available resources (Available (1, 5, 3)).
  - P1: Needs (0, 3, 5). Available (1, 5, 3). P1 cannot be satisfied (needs more printers than available).
  - P2: Needs (8, 6, 0). Available (1, 5, 3). P2 cannot be satisfied.

    - P3: Needs (2, 1, 1). Available (1, 5, 3). P3 cannot be satisfied (needs more tape drives).
    - P4: Needs (1, 5, 0). Available (1, 5, 3). P4 can be satisfied.
- Assume P4 finishes. Release its held resources back to the available pool. P4 held (1, 3, 2), so the new available resources are:
    - New Available = (1+1, 5+3, 3+2) = (2, 8, 5)
- Look for the next process that can be satisfied with the new available resources (Available (2, 8, 5)).
    - P1: Needs (0, 3, 5). Available (2, 8, 5). P1 can be satisfied.
- Assume P1 finishes. Release its held resources back to the available pool. P1 held (2, 1, 1), so the new available resources are:
    - New Available = (2+2, 8+1, 5+1) = (4, 9, 6)
- Look for the next process that can be satisfied with the new available resources (Available (4, 9, 6)).
    - P2: Needs (8, 6, 0). Available (4, 9, 6). P2 cannot be satisfied.
    - P3: Needs (2, 1, 1). Available (4, 9, 6). P3 can be satisfied.
- Assume P3 finishes. Release its held resources back to the available pool. P3 held (4, 0, 0), so the new available resources are:
    - New Available = (4+4, 9+0, 6+0) = (8, 9, 6)
- Look for the next process that can be satisfied with the new available resources (Available (8, 9, 6)).
    - P2: Needs (8, 6, 0). Available (8, 9, 6). P2 can be satisfied.
- Assume P2 finishes. Release its held resources back to the available pool. P2 held (0, 0, 2), so the new available resources are:
    - New Available = (8+0, 9+0, 6+2) = (8, 9, 8)

So, from our above calculation, we can see that all processes can finish in the following order: P0 → P4 → P1 → P3 → P2. Hence, since all processes can finish, the system is in a safe state.

Q2) Consider a process that refers to five pages: A, B, C, D, and E. The pages are referenced in the following order: C, A, C, E, B, D, C, C, B, D, E, A, E, B, D. Assuming memory is initially empty, and that no other processes are running, show the contents of each page frame for each step, along with any other necessary data (i.e. status of the referenced bit for second-chance and clock, process queue for second-chance, and the clock pointer for clock), and determine the number of page transfers during this sequence of references for the following page replacement algorithms and numbers of page frames:

- First-in, first-out:
    - ➢ 3 page frames
    - ➢ 4 page frames
    - ➢ 5 page frames
- Second-chance (assuming referenced bit is set when a page is first put in memory):
    - ➢ 3 page frames
    - ➢ 4 page frames
    - ➢ 5 page frames
- Clock (assuming referenced bit is set when a page is first put in memory):
    - ➢ 3 page frames
    - ➢ 4 page frames
    - ➢ 5 page frames

Answer:

- First-In, First-Out (FIFO)

  In FIFO, the oldest page in memory is replaced when a new page is needed.

  - 3 Page Frames

Page	**Frame**	**Page Transfer?**
C	[C, -, -]	Yes
A	[C, A, -]	Yes
C	[C, A, -]	No
E	[C, A, E]	Yes
B	[B, A, E]	Yes
D	[B, D, E]	Yes
C	[B, D, C]	Yes
C	[B, D, C]	No
B	[B, D, C]	No
D	[B, D, C]	No
E	[E, D, C]	Yes
A	[E, A, C]	Yes
E	[E, A, C]	No
B	[E, A, B]	Yes
D	[D, A, B]	Yes

Total Page Transfer = 10

- 4 Page Frames

Page	**Frame**	**Page Transfer?**
C	[C, -, -, -]	Yes
A	[C, A, -, -]	Yes
C	[C, A, -, -]	No
E	[C, A, E, -]	Yes
B	[C, A, E, B]	Yes
D	[D, A, E, B]	Yes
C	[D, C, E, B]	Yes
C	[D, C, E, B]	No
B	[D, C, E, B]	No
D	[D, C, E, B]	No
E	[D, C, E, B]	No
A	[D, C, A, B]	Yes
E	[D, C, A, E]	Yes
B	[B, C, A, E]	Yes
D	[B, D, A, E]	Yes

Total Page Transfer = 10

- 5 Page Frames

Page	**Frame**	**Page Transfer?**
C	[C, -, -, -, -]	Yes
A	[C, A, -, -, -]	Yes
C	[C, A, -, -, -]	No
E	[C, A, E, -, -]	Yes
B	[C, A, E, B, -]	Yes
D	[C, A, E, B, D]	Yes
C	[C, A, E, B, D]	No
C	[C, A, E, B, D]	No
B	[C, A, E, B, D]	No
D	[C, A, E, B, D]	No
E	[C, A, E, B, D]	No
A	[C, A, E, B, D]	No
E	[C, A, E, B, D]	No
B	[C, A, E, B, D]	No
D	[C, A, E, B, D]	No

Total Page Transfer = 5

- Second-Chance Algorithm

  The Second-Chance algorithm is an enhancement of FIFO, where the referenced bit is used to give pages a "second chance." The referenced bit is set to 1 when a page is accessed. When replacing a page, if its referenced bit is 1, it is reset to 0 and the page is moved to the end of the queue. If its bit is 0, it is replaced.

  - 3 Page Frames

Page	**Frame**	**Bits**	**Page Transfer?**
C	[C, -, -]	1	Yes
A	[C, A, -]	1, 1	Yes
C	[C, A, -]	1, 1	No
E	[C, A, E]	1, 1, 1	Yes
B	[B, A, E]	1, 0, 0	Yes
D	[B, D, E]	1, 1, 0	Yes
C	[B, D, C]	1, 1, 1	Yes
C	[B, D, C]	1, 1, 1	No
B	[B, D, C]	1, 1, 1	No
D	[B, D, C]	1, 1, 1	No
E	[E, D, C]	1, 0, 0	Yes
A	[E, A, C]	1, 1, 0	Yes
E	[E, A, C]	1, 1, 0	No
B	[E, A, B]	1, 1, 1	Yes
D	[D, A, B]	1, 0, 0	Yes

Total Page Transfer = 10

- 4 Page Frames

Page	**Frame**	**Bits**	**Page Transfer?**
C	[C, -, -, -]	1	Yes
A	[C, A, -, -]	1, 1	Yes
C	[C, A, -, -]	1, 1	No
E	[C, A, E, -]	1, 1, 1	Yes
B	[C, A, E, B]	1, 1, 1, 1	Yes
D	[D, A, E, B]	1, 0, 0, 0	Yes
C	[D, C, E, B]	1, 1, 0, 0	Yes
C	[D, C, E, B]	1, 1, 0, 0	No
B	[D, C, E, B]	1, 1, 0, 1	No
D	[D, C, E, B]	1, 1, 0, 1	No
E	[D, C, E, B]	1, 1, 1, 1	No
A	[D, C, A, B]	0, 0, 1, 0	Yes
E	[D, C, A, E]	0, 0, 1, 1	Yes
B	[B, C, A, E]	1, 0, 1, 1	Yes
D	[B, D, A, E]	1, 1, 1, 1	Yes

Total Page Transfer = 10

- 5 Page Frames

Page	**Frame**	**Bits**	**Page Transfer?**
C	[C, -, -, -, -]	1	Yes
A	[C, A, -, -, -]	1, 1	Yes
C	[C, A, -, -, -]	1, 1	No
E	[C, A, E, -, -]	1, 1, 1	Yes
B	[C, A, E, B, -]	1, 1, 1, 1	Yes
D	[C, A, E, B, D]	1, 1, 1, 1, 1	Yes
C	[C, A, E, B, D]	1, 1, 1, 1, 1	No
C	[C, A, E, B, D]	1, 1, 1, 1, 1	No
B	[C, A, E, B, D]	1, 1, 1, 1, 1	No
D	[C, A, E, B, D]	1, 1, 1, 1, 1	No
E	[C, A, E, B, D]	1, 1, 1, 1, 1	No
A	[C, A, E, B, D]	1, 1, 1, 1, 1	No
E	[C, A, E, B, D]	1, 1, 1, 1, 1	No
B	[C, A, E, B, D]	1, 1, 1, 1, 1	No
D	[C, A, E, B, D]	1, 1, 1, 1, 1	No

Total Page Transfer = 5

- Clock Algorithm

  The Clock algorithm works similarly to Second Chance but uses a circular queue to track the pages. The "clock hand" advances as pages are checked for replacement.

  - 3 Page Frames

Page	**Frame**	**Bits**	**Clock Hand**	**Page Transfer?**
C	[C, -, -]	1	1	Yes
A	[C, A, -]	1, 1	2	Yes
C	[C, A, -]	1, 1	2	No
E	[C, A, E]	1, 1, 1	0	Yes
B	[B, A, E]	1, 0, 0	1	Yes
D	[B, D, E]	1, 1, 0	2	Yes
C	[B, D, C]	1, 1, 1	0	Yes
C	[B, D, C]	1, 1, 1	0	No
B	[B, D, C]	1, 1, 1	0	No
D	[B, D, C]	1, 1, 1	0	No
E	[E, D, C]	1, 0, 0	1	Yes
A	[E, A, C]	1, 1, 0	2	Yes
E	[E, A, C]	1, 1, 0	2	No
B	[E, A, B]	1, 1, 1	0	Yes
D	[D, A, B]	1, 0, 0	1	Yes

Total Transfer = 10

- 4 Page Frames

Page	**Frame**	**Bits**	**Clock Hand**	**Page Transfer?**
C	C, -, -, -	1	1	Yes
A	C, A, -, -	1, 1	2	Yes
C	C, A, -, -	1, 1	2	No
E	C, A, E, -	1, 1, 1	3	Yes
B	C, A, E, B	1, 1, 1, 1	0	Yes
D	D, A, E, B	1, 0, 0, 0	1	Yes
C	D, C, E, B	1, 1, 0, 0	2	Yes
C	D, C, E, B	1, 1, 0, 0	2	No
B	D, C, E, B	1, 1, 0, 1	2	No
D	D, C, E, B	1, 1, 0, 1	2	No
E	D, C, E, B	1, 1, 1, 1	2	No
A	D, C, A, B	0, 0, 1, 0	3	Yes
E	D, C, A, E	0, 0, 1, 1	0	Yes
B	B, C, A, E	1, 0, 1, 1	1	Yes
D	B, D, A, E	1, 1, 1, 1	2	Yes

Total Transfer = 10

- 5 Page Frames

Page	**Frame**	**Bits**	**Clock Hand**	**Page Transfer?**
C	C, -, -, -, -	1	1	Yes
A	C, A, -, -, -	1, 1	2	Yes
C	C, A, -, -, -	1, 1	2	No
E	C, A, E, -, -	1, 1, 1	3	Yes
B	C, A, E, B, -	1, 1, 1, 1	4	Yes
D	C, A, E, B, D	1, 1, 1, 1, 1	0	Yes
C	C, A, E, B, D	1, 1, 1, 1, 1	0	No
C	C, A, E, B, D	1, 1, 1, 1, 1	0	No
B	C, A, E, B, D	1, 1, 1, 1, 1	0	No
D	C, A, E, B, D	1, 1, 1, 1, 1	0	No
E	C, A, E, B, D	1, 1, 1, 1, 1	0	No
A	C, A, E, B, D	1, 1, 1, 1, 1	0	No
E	C, A, E, B, D	1, 1, 1, 1, 1	0	No
B	C, A, E, B, D	1, 1, 1, 1, 1	0	No
D	C, A, E, B, D	1, 1, 1, 1, 1	0	No

Total Page Transfer = 5

Q3) Suppose an I-node can hold 11 direct block pointers, 5 indirect block pointers, and 3 double-indirect block pointers. Given a block size of 4KB, and 128-bit block numbers:

- What is the size (in bytes) of the smallest file that requires use of an indirect block pointer?

  Answer:

  Here,

  The file will use direct block pointers first before moving on to indirect block pointers. And each direct block can hold up to 4 KB of data.

  There are 11 direct block pointers, so the maximum size a file can be before requiring an indirect block is:

  11 * 4KB = 11 * 4096 = 45056 bytes

  Now, a file that is one byte larger than 45056 bytes will require an indirect block pointer. Thus, the smallest file that requires an indirect block pointer is 45056 + 1 = 45057 bytes.

- What is the size (in bytes) of the smallest file that requires use of a double-indirect block pointer?

  Answer:

  Here,

  The file will use direct block pointers first before moving on to indirect block pointers. And again, the file will use indirect block pointers first before moving on to double indirect block pointers. Each indirect block can store block pointers. Since each block number is 128 bits = 128/8 bytes = 16 bytes, the number of pointers each indirect block can store is:

  4 KB/16 bytes = 4 * 1024 / 16 = 256

  Again, each indirect block points to 256 direct blocks, and each direct block holds 4 KB. So, the total data an indirect block can point to is:

  256 * 4 KB = 256 * 4 * 1024 = 1048576 bytes

  Now, as there are 5 indirect block pointers in the I-node, so the maximum size of a file before needing a double-indirect block is:

  45,056 bytes (direct) + 5 * 1048576 (indirect) = 5287936 bytes

  Now, a file that is one byte larger than 5287936 bytes will require an indirect block pointer. Thus, the smallest file that requires an indirect block pointer is 5287936 + 1 = 5287937 bytes.

- What is the size (in bytes) of the largest file supported by this system?

  Answer:

  Here,

  As calculated earlier,

  Data from Direct Blocks = 45056 bytes

  Data from Indirect Blocks = 5242880 bytes

  And now,

  A double-indirect block contains pointers to indirect blocks, and each indirect block points to 256 direct blocks. Therefore, each double-indirect block can point to:

  256 indirect blocks * 256 direct blocks = 65536 direct blocks

  As, each direct block holds 4 KB of data, each double-indirect block can address:

  65536 * 4 KB = 65536 * 4096 = 268435456 bytes

  And as there are 3 double-indirect block pointers, so the data from double-indirect blocks is:

  3 * 268435456 bytes = 805306368 bytes

  Now, Total Maximum File Size = 45056 bytes (direct) + 5242880 bytes (indirect) + 805306368 bytes (double-indirect)

  = 810594304 bytes

  Thus, the largest file supported by this system is 810594304 bytes.

- What is the size (in bytes) of the largest file that would be supported by this system if it had 64-bit block numbers instead of 128-bit block numbers?

  Answer:

  Here,

  If the system had 64-bit block numbers instead of 128-bit block numbers then, with 64-bit block numbers, each block number would take up 64/8 bytes = 8 bytes instead of 16 bytes.

  Then,

  Indirect Block Capacity = 4 KB / 8 = 4096/8 = 512 pointers

  So, each indirect block can now point to 512 direct blocks, so the data per indirect block becomes:

  512 * 4 KB = 512 * 4096 = 2097152 bytes

  And as there are 5 indirect blocks,

  Data from Indirect Blocks = 5 * 2097152 = 10485760 bytes

  Now again,

  Each double-indirect block points to 512 indirect blocks, and each indirect block points to 512 direct blocks. So, each double-indirect block can point to:

  512 * 512 = 262144 direct blocks

  Each direct block holds 4 KB, so the data from each double-indirect block is:

  262144 * 4 KB = 262144 * 4096 = 1073741824 bytes

  And again, as there are 3 double-indirect block pointers, so the data from double-indirect blocks is:

  3 * 1073741824 bytes = 3221225472 bytes

  Now, Total Maximum File Size (with 64-bit block numbers)

  = 45056 bytes (direct) + 10485760 bytes (indirect) + 3221225472 bytes (double-indirect)

= 3231756288 bytes

Thus, the largest file supported by this system with 64-bit block numbers is 3231756288 bytes.

Q4) An application program takes time T to execute on a 10-computer cluster

- What is the effective speedup of running this application on the cluster rather than a single computer if 25% of T is spent running on all 10 computers, and the remaining time is spent running on a single computer?

  Answer:

  Here,

  We know,

  Speedup = Time on a Single Computer / Time on the Cluster

  Then,

  Time on the cluster:

  25% of the time the application is running on 10 computers, this takes 0.25T on the cluster.

  75% of the time it runs on a single computer, this takes 0.75T on the cluster.

  Time on a single computer:

  The part that runs on 10 computers takes 10 times longer on a single computer, so 0.25T would take: 10 * 0.25T = 2.5T

  The part that runs on 1 computer remains the same, so it takes 0.75T.

  Now, Total time on a single computer

  = 2.5T + 0.75T = 3.25T

  So, Effective speedup = 3.25T/T = 3.25

- What is the effective speedup of running this application on the cluster rather than a single computer if 50% of T is spent running on all 10 computers, and the remaining time is spent running on a single computer?

  Answer:

  Here,

  We know,

  Speedup = Time on a Single Computer / Time on the Cluster

  And let, Time on the Cluster be T. Then,

  Time on the cluster:

  50% of the time the application is running on 10 computers, this takes 0.5T on the cluster.

  50% of the time it runs on a single computer, this takes 0.5T on the cluster.

  Time on a single computer:

  The part that runs on 10 computers takes 10 times longer on a single computer, so 0.5T would take: 10 * 0.5T = 5T

  The part that runs on 1 computer remains the same, so it takes 0.5T.

  Now, Total time on a single computer

  = 5T + 0.5T = 5.5T

  So, Effective speedup = 5.5T/T = 5.5

- What is the effective speedup of running this application on the cluster rather than a single computer if 7% of T is spent running on all 10 computers, 34% of T is spent running on 7 computers, and the remaining time is spent running on a single computer?

  Answer:

  Here,

  We know,

  Speedup = Time on a Single Computer / Time on the Cluster

  And let, Time on the Cluster be T. Then,

  Time on the cluster:

  7% of the time the application runs on 10 computers, this takes 0.07T.

  34% of the time the application runs on 7 computers, this takes 0.34T.

  59% of the time the application runs on 1 computer, this takes 0.59T.

  Time on a single computer:

  The part that runs on 10 computers would take 10 * 0.07T = 0.7T.

  The part that runs on 7 computers would take 7 * 0.34T = 2.38T.

  The part that runs on 1 computer remains 0.59T.

  Now, Total time on single computer

  = 0.7T + 2.38T + 0.59T = 3.67T

  So, Effective speedup = 3.67T/T = 3.67

- What is the effective speedup of running this application on the cluster rather than a single computer if 94% of T is spent running on all 10 computers,4% of T is spent running on 6 computers, and the remaining time is spent running on a single computer?

  Answer:

  Here,

  We know,

  Speedup = Time on a Single Computer / Time on the Cluster

  And let, Time on the Cluster be T. Then,

  Time on the cluster:

  94% of the time the application runs on 10 computers, this takes 0.94T.

  4% of the time the application runs on 6 computers, this takes 0.04T.

  2% of the time the application runs on 1 computer, this takes 0.02T.

  Time on a single computer:

  The part that runs on 10 computers would take 10 * 0.94T = 9.4T.

  The part that runs on 6 computers would take 6 * 0.04T = 0.24T.

  The part that runs on 1 computer remains 0.02T.

  Now, Total time on single computer

  = 9.4T + 0.24T + 0.02T = 9.66T

  So, Effective speedup = 9.66T/T = 9.66

Q5) A computer virus is malicious software that replicates itself by modifying other programs to insert its own code. Antivirus software was initially designed to detect and remove computer viruses, but now also provides protection from other potential computer threats. Write a brief (approx. 500 words) history of the development of antivirus software, describing some of the techniques used by viruses that have required new methods of detection for antivirus software.

Answer:

The development of antivirus software traces back to the early days of computing when computer viruses first emerged as a significant threat. Antivirus software has evolved to tackle increasingly sophisticated malware, adapting its detection and removal techniques to keep pace with advances in malicious software design.

Early Viruses and the First Antivirus Programs

The first recognized computer virus, "Creeper," was created in the early 1970s as an experimental self-replicating program that spread between computers on ARPANET, an early sign to the internet. This virus, while not malicious, prompted the development of "Reaper," which is considered the first antivirus software. Reaper's primary function was to locate and delete Creeper, laying the base for future virus removal tools (Bob Thomas, 2024).

By the mid-1980s, the spread of viruses like "Brain," the first virus for MS-DOS, required more robust antivirus measures. Brain was a boot sector virus, infecting floppy disks and spreading when they were inserted into computers. Its rise underscored the need for early virus detection methods that focused on scanning disks and system files for known malware signatures (Amjad Farooq Alvi, 2024).

Signature-Based Detection

The first antivirus programs primarily used signature-based detection to identify and remove viruses. In this method, the software scans files for unique sequences of code, known as signatures, which are associated with known viruses. When the software identifies a match, it flags the file as infected and takes appropriate action. This approach worked effectively for detecting early viruses that shared common patterns or characteristics (Wikipedia, 2024). However, as malware authors developed techniques like polymorphism—where a virus alters its code each time it replicates—signature-based detection became less effective.

Polymorphic and Metamorphic Viruses

Polymorphic viruses were first seen in the early 1990s, with malware like "Tequila" demonstrating the ability to avoid signature-based detection by altering its appearance with each infection. Antivirus developers responded by creating heuristic analysis techniques. Heuristic analysis allows antivirus software to identify previously unknown viruses by detecting suspicious behaviour patterns, such as code that attempts to self-replicate or modify system files (Wikipedia, 2024).

Metamorphic viruses emerged later as an evolution of polymorphic malware. Metamorphic viruses change their code in even more significant ways, completely rewriting parts of themselves to avoid detection. These viruses posed a challenge to heuristic analysis, leading to the development of more advanced detection methods such as code emulation, where the antivirus software runs a suspicious program in a virtual environment to observe its behaviour before allowing it to execute on the system (Wikipedia, 2024).

Modern Antivirus Techniques

As the internet grew in the late 1990s and early 2000s, viruses began to spread more rapidly through email and web downloads. Malware like the "ILOVEYOU" worm and "Melissa" virus highlighted the need for real-time protection. Antivirus software began incorporating real-time scanning capabilities that monitor system activity and intercept malware before it could cause damage (Onel de Guzman, 2024).

Today, antivirus software uses a combination of signature-based detection, heuristic analysis, machine learning, and behavioural analysis to protect against a wide variety of threats, including viruses, worms, ransomware, and spyware. The ongoing cat-and-mouse game between malware authors and antivirus developers ensures that antivirus software will continue to evolve, adapting to new and more sophisticated threats.

References

- Amjad Farooq Alvi. (2024, July 6). *Brain (computer virus)*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Brain_(computer_virus)
- Bob Thomas. (2024, July 26). *Creeper and Reaper*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Creeper_and_Reaper
- Onel de Guzman. (2024, September 6). *ILOVEYOU*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/ILOVEYOU
- Wikipedia. (2024, September 5). *Antivirus software*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Antivirus_software
- Wikipedia. (2024, January 3). *Metamorphic code*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Metamorphic_code
- Wikipedia. (2024, July 3). *Polymorphic code*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Polymorphic_code