Tutorial 1: Operating Systems

Yonghao Lee

March 26, 2025

Motivations

We study operating systems to understand fundamental computing concepts such as:

- The mechanisms behind file reading and writing operations
- How computers maintain responsiveness despite slow I/O operations
- Methods for executing programs that require more memory than physically available
- Techniques for accelerating program execution using multi-core architectures
- The nature and causes of segmentation faults
- Implementation of virtualization to run multiple operating systems on a single machine
- Practical aspects of inter-computer communication

Introduction

Operating System

An operating system is software that manages computer resources and provides service abstractions to application programmers and end-users. These resources include memory, files, accelerators, GPUs, and other hardware components. The operating system has exclusive management authority over these resources and grants privileges to applications to access them.

Operating systems are responsible for:

- **Resource scheduling** — determining which process runs at any given time point
- **Memory management** — allocating and deallocating memory (e.g., when a C program calls `malloc()`)
- **Hardware abstraction** — providing unified interfaces to diverse hardware

The operating system is the only software permitted to execute privileged instructions—operations that can fundamentally alter the computer's state or access special registers.

A key challenge in operating system design is managing resources efficiently without foreknowledge of future demands. Operating systems must make heuristic decisions about scheduling and memory allocation without knowing in advance which applications will request resources. Performance is critical, as OS inefficiencies can delay important processes and degrade overall system responsiveness.

Furthermore, operating systems must:

- Function reliably with complex hardware architectures
- Handle all edge cases and meet real-time deadlines
- Provide robust security against cyber attacks

CPU (Central Processing Unit)

The CPU primarily performs arithmetic operations using registers as its working memory. It can only operate directly on data stored in these registers.

Critical registers include:

1. **PC** — Program Counter, which indicates the next instruction to execute
2. **SP** — Stack Pointer, which manages the program stack
3. **IR** — Instruction Register, which holds the current instruction being processed

Beyond arithmetic operations, the CPU handles data movement (loading and storing) and control flow (branching and jumping).

CPU Instruction Translation

Consider this example of how a C program is translated into machine code:

```c
int abs(int x)
{
    if(x < 0)
    {
        return -x;
    }
    return x;
}
```

Figure 1: Program written in C

```
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        cmpl    $0, -4(%rbp)
        jns     .positive
        movl    -4(%rbp), %eax
        negl    %eax
        jmp     .finish
.positive:
        movl    -4(%rbp), %eax
.finish:
        popq    %rbp
        ret
```

Figure 2: x86 Assembly translation

```
01010101                                // pushq  %rbp
01001000 10001001 11100101              // movq   %rsp, %rbp
10001001 01111101 11111100              // movl   %edi, -4(%rbp)
10000011 01111101 11111100 00000000     // cmpl   $0, -4(%rbp)
01111001 00000111                       // jns    .positive
10001011 01000101 11111100              // movl   -4(%rbp), %eax
11110111 11011000                       // negl   %eax
11101011 00000011                       // jmp    .finish
                                        // .positive:
10001011 01000101 11111100              // movl   -4(%rbp), %eax
                                        // .finish:
01011101                                // popq    %rbp
11000011                                // ret
```

Figure 3: Binary machine code representation

Assembly instructions map directly to machine code in a one-to-one relationship. For

example, the instruction `pushq %rbp` (which pushes the base pointer onto the stack) is represented by a specific binary sequence.

CPU Instruction Lifecycle

The typical CPU instruction lifecycle consists of:

- **Fetch** — Retrieving the instruction from the memory address indicated by the PC
- **Decode** — Transferring the instruction to the IR and determining its operation
- **Execute** — Performing the operation (often using the ALU)
- **Write back** — Storing the result if necessary

CPU Operation Performance

CPU operations vary significantly in execution time and resource requirements:

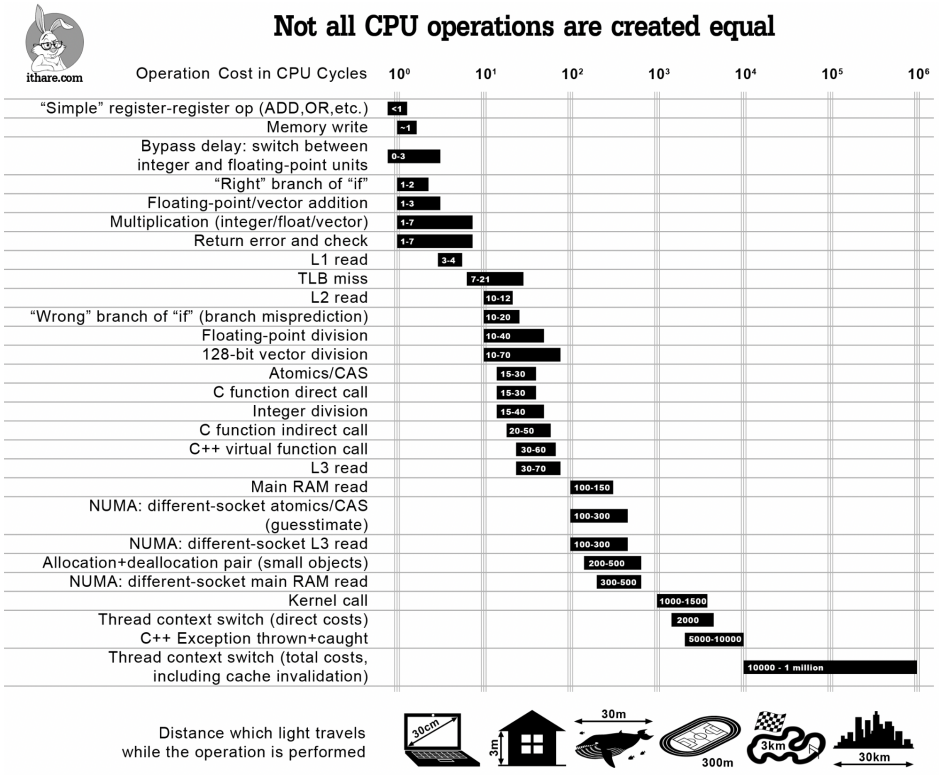

Figure 4: Comparison of CPU operation costs (in cycles)

Simple operations may complete in less than one cycle through vectorization, while kernel calls (OS interactions) are extremely expensive, requiring 1000-1500 cycles.

Memory Hierarchy

The memory hierarchy consists of several layers with different characteristics:

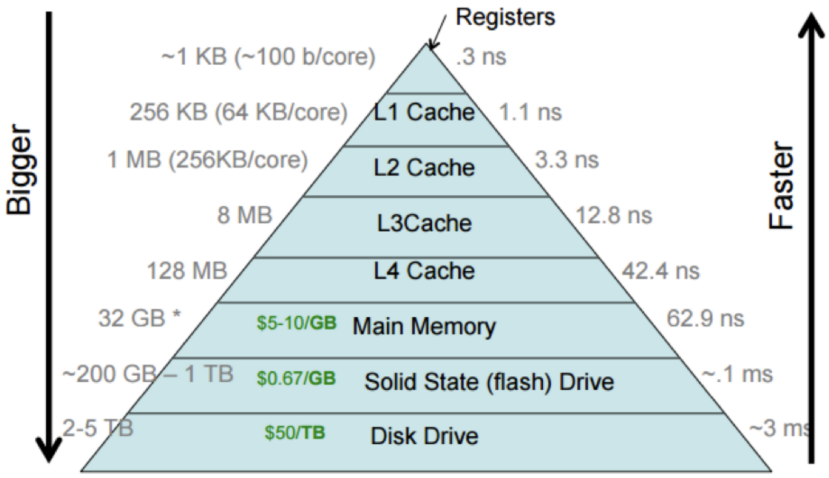

Figure 5: Memory hierarchy showing trade-offs between speed, capacity, and volatility

Main Memory

Main memory (RAM) is volatile storage located on chips inside the computer but outside the CPU. When the computer powers off, the contents of main memory are erased.

Secondary Memory

Secondary memory (e.g., hard drives, SSDs) is non-volatile, retaining data when power is removed. It offers greater capacity than main memory but operates more slowly due to its physical composition and distance from the CPU.

The disk controller uses Direct Memory Access (DMA) to transfer data between secondary storage and main memory without CPU intervention. Programs stored on disk must be loaded into main memory before execution, as the CPU cannot directly access disk data.

Cache

Cache is small, fast memory positioned close to the CPU. It exploits the principles of locality to improve performance:

- **Spatial locality** — If a memory location is accessed, nearby locations are likely to be accessed soon
- **Temporal locality** — Recently accessed locations are likely to be accessed again

Cache is organized in three levels:

1. **L1 cache** — Smallest and fastest (3-4 cycles per read), never shared between cores

2. **L2 cache** — Intermediate size and speed (approximately 10-12 cycles per read)

3. **L3 cache** — Largest and slowest cache level, but still much faster than main memory

Effective use of cache significantly impacts program performance. Variables accessed frequently should ideally remain in cache to minimize memory access latency.

Virtualization

Virtualization is the process of creating virtual versions of:

- Computer hardware platforms
- Storage devices
- Computer network resources
- Other computing resources

A virtual machine (VM) is an emulation of a computer system. Multiple VMs can run on a single physical computer, with each VM having its own virtual resources and operating system.

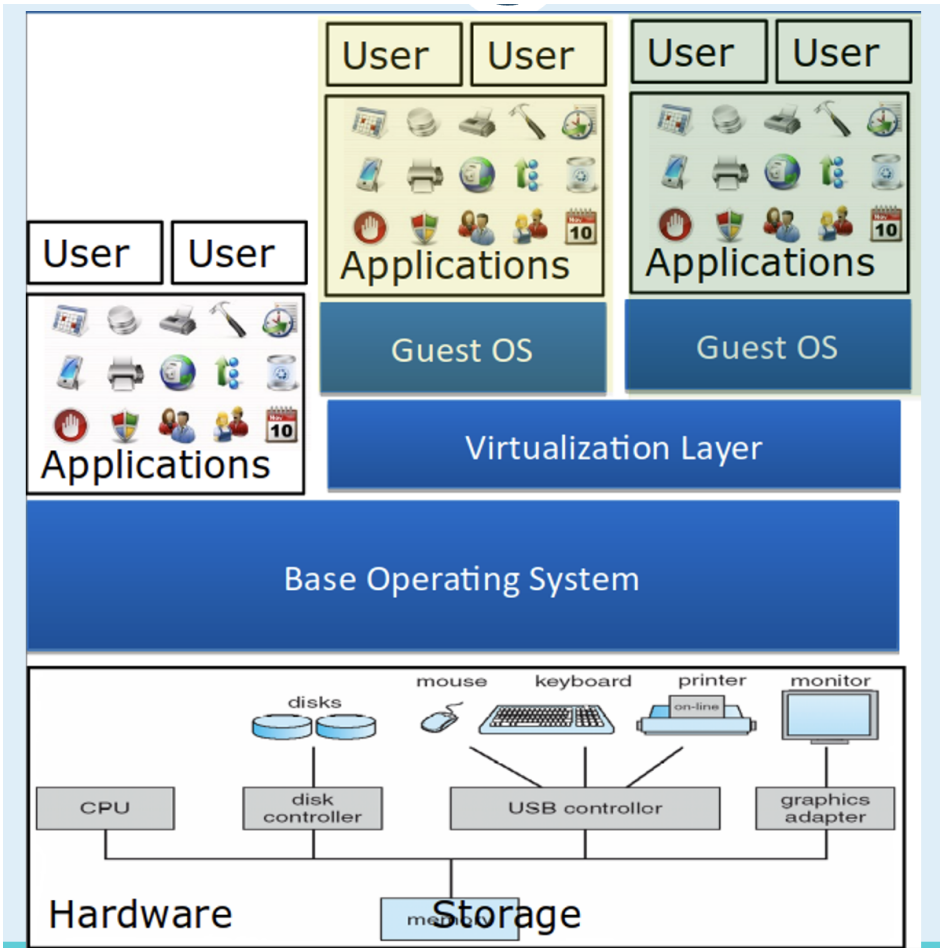

Figure 6: Traditional virtualization architecture showing multiple VMs with separate operating systems

Since running multiple VMs with separate operating systems consumes substantial system resources, container technology was developed as a more efficient alternative.

Containers (e.g., Docker) implement OS-level virtualization. A program running inside a container can only access the content and devices assigned to that container, though the host system can directly access the container files. Multiple containers can run on the same machine and share the host's OS kernel.

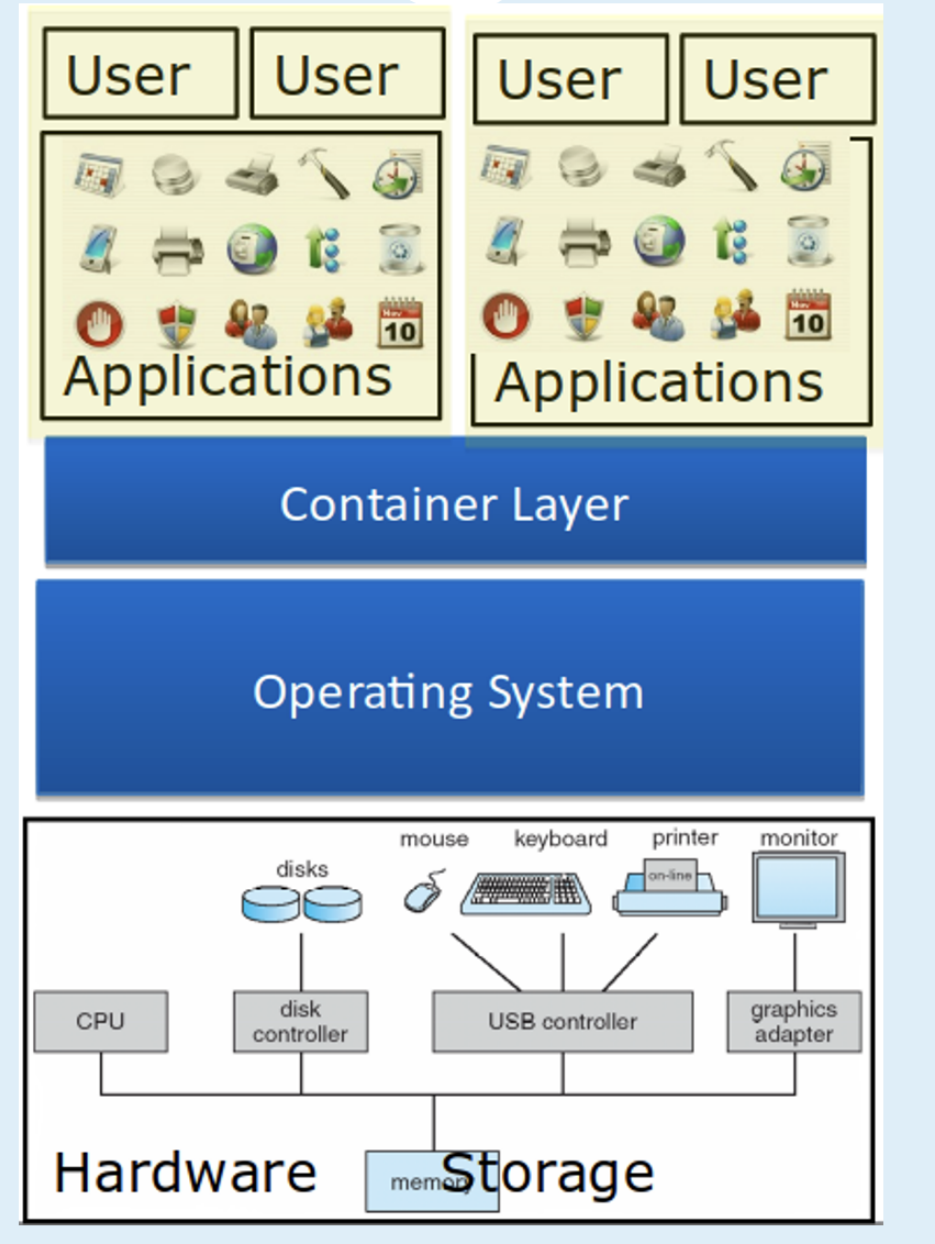

Figure 7: Container architecture showing multiple containers sharing a single OS kernel

Memory Debugging with Valgrind

Programs may contain hidden errors that can pass compilation and even execute without immediately revealing dangerous bugs. Valgrind is a powerful debugging tool for Linux that helps detect:

- Memory leaks
- Improper usage of memory-related functions

- Out-of-bounds access errors
- Potential causes of segmentation faults

The following examples demonstrate Valgrind's capabilities in identifying various memory-related issues.

Example 1: Invalid Memory Access

```c
#include<stdlib.h>
#include<stdio.h>

void foo(int n) {
   int i;
   int *a = (int*) malloc(n*sizeof(int));

   a[0] = 1;
   printf("a[0] = 1\n");
   a[1] = 1;
   printf("a[1] = 1\n");

   for (i=2; i<=n; i++) {
      a[i] = a[i-1] + a[i-2];
      printf("a[%d] = %d\n", i, a[i]);
   }

   free(a);
}

int main(int argc, char *argv[]) {
   foo(10);
   return 0;
}
```

Figure 8: C code with array bounds violation

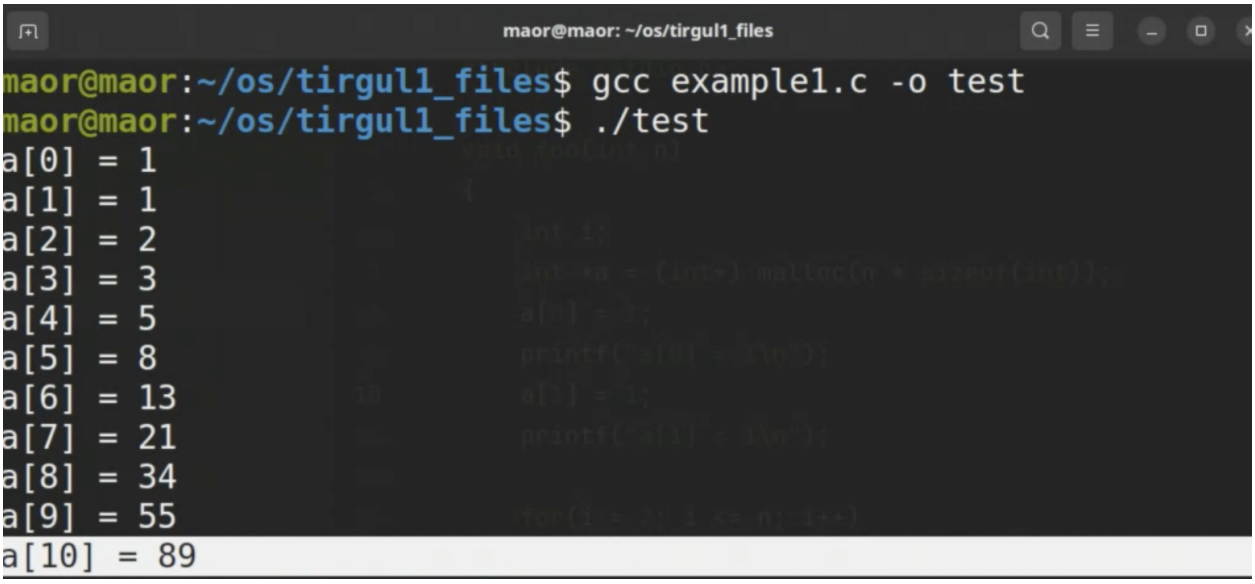

Figure 9: Program compiles and runs without compiler warnings

This code compiles successfully with gcc, and the compiler does not detect any errors. However, the final iteration of the loop accesses memory outside the allocated array bounds. When analyzed with Valgrind:

```
==66567==    by 0x1092B6: main (in /media/maor/PhD/work/os/tirgul1_files/test)
==66567==  Address 0x4a9a068 is 0 bytes after a block of size 40 alloc'd
==66567==    at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==66567==    by 0x1091C8: foo (in /media/maor/PhD/work/os/tirgul1_files/test)
==66567==    by 0x1092B6: main (in /media/maor/PhD/work/os/tirgul1_files/test)
==66567==
==66567== Invalid read of size 4
==66567==    at 0x109264: foo (in /media/maor/PhD/work/os/tirgul1_files/test)
==66567==    by 0x1092B6: main (in /media/maor/PhD/work/os/tirgul1_files/test)
==66567==  Address 0x4a9a068 is 0 bytes after a block of size 40 alloc'd
==66567==    at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==66567==    by 0x1091C8: foo (in /media/maor/PhD/work/os/tirgul1_files/test)
==66567==    by 0x1092B6: main (in /media/maor/PhD/work/os/tirgul1_files/test)
==66567==
a[10] = 89
==66567==
==66567== HEAP SUMMARY:
==66567==     in use at exit: 0 bytes in 0 blocks
==66567==   total heap usage: 2 allocs, 2 frees, 1,064 bytes allocated
==66567==
==66567== All heap blocks were freed -- no leaks are possible
==66567==
==66567== For lists of detected and suppressed errors, rerun with: -s
==66567== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
maor@maor:~/os/tirgul1_files$
```

Figure 10: Valgrind output showing invalid read/write errors

Valgrind identifies both invalid read and write operations of size 4 bytes (the size of an integer), indicating illegal memory access. The output also shows the call stack: `main` called `foo`, which called `malloc`.

Since the code was compiled without debug symbols, Valgrind cannot show which specific lines in the source code generated these errors. Adding the `-g` flag during compilation provides this information:

```
maor@maor:~/os/tirgul1_files$ gcc example1.c -g -o test
```

Figure 11: Compiling with debug symbols using the -g flag

```
==66669== Invalid write of size 4
==66669==    at 0x10924E: foo (example1.c:15)
==66669==    by 0x1092B6: main (example1.c:22)
==66669==  Address 0x4a9a068 is 0 bytes after a block of size 40 alloc'd
==66669==    at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==66669==    by 0x1091C8: foo (example1.c:7)
==66669==    by 0x1092B6: main (example1.c:22)
==66669==
==66669== Invalid read of size 4
==66669==    at 0x109264: foo (example1.c:16)
==66669==    by 0x1092B6: main (example1.c:22)
==66669==  Address 0x4a9a068 is 0 bytes after a block of size 40 alloc'd
==66669==    at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==66669==    by 0x1091C8: foo (example1.c:7)
==66669==    by 0x1092B6: main (example1.c:22)
==66669==
```

Figure 12: Valgrind output with source code line information

With debug symbols, Valgrind can now identify that the error occurs on line 15 of the source code.

Example 2: Memory Leaks

```c
#include<stdlib.h>
#include<stdio.h>

void foo(int n) {
   int *a = (int*) malloc(n*sizeof(int));
   int i;

   for (i=0; i<n; i++) {
      a[i] = i*i;
      printf("a[%d] = %d\n", i, a[i]);
   }
}

int main(int argc, char *argv[]) {
   foo(10);
   return 0;
}
```

Figure 13: C code with memory leak

In this example, memory is allocated but never freed, resulting in a memory leak. Valgrind's heap summary identifies this issue:

```
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
a[5] = 25
a[6] = 36
a[7] = 49
a[8] = 64
a[9] = 81
==66757==
==66757== HEAP SUMMARY:
==66757==     in use at exit: 40 bytes in 1 blocks
==66757==   total heap usage: 2 allocs, 1 frees, 1,064 bytes allocated
==66757==
==66757== LEAK SUMMARY:
==66757==    definitely lost: 40 bytes in 1 blocks
==66757==    indirectly lost: 0 bytes in 0 blocks
==66757==      possibly lost: 0 bytes in 0 blocks
==66757==    still reachable: 0 bytes in 0 blocks
==66757==         suppressed: 0 bytes in 0 blocks
==66757== Rerun with --leak-check=full to see details of leaked memory
==66757==
==66757== For lists of detected and suppressed errors, rerun with: -s
==66757== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
maor@maor:~/os/tirgul1_files$
```

Figure 14: Valgrind output showing memory leak detection

The heap summary indicates that 40 bytes are "definitely lost," meaning they were allocated but never freed.

Example 3: Uninitialized Variables

```c
#include<stdlib.h>
#include<stdio.h>

int main(int argc, char *argv[]) {
    int y;
    y += 1;

    printf("Done\n");
    return 0;
}
```

Figure 15: C code with uninitialized variable

```
maor@maor:~/os/tirgul1_files$ gcc example8.c -g -o test
maor@maor:~/os/tirgul1_files$ ./test
Done, y = 32767
maor@maor:~/os/tirgul1_files$ ./test
Done, y = 32766
maor@maor:~/os/tirgul1_files$ ./test
```

Figure 16: Program output showing unpredictable values for uninitialized variable

Since the variable `y` is uninitialized, its value changes on each program execution. This error is not caught by default Valgrind settings, but adding the `-Wall` flag during compilation reveals the issue:

```
maor@maor:~/os/tirgul1_files$ gcc -Wall example8.c -g -o test
example8.c: In function 'main':
example8.c:7:7: warning: 'y' is used uninitialized [-Wuninitialized]
    7 |     y += 1;
      |     ~~^~~~
example8.c:6:9: note: 'y' was declared here
    6 |     int y;
      |         ^
maor@maor:~/os/tirgul1_files$
```

Figure 17: Compiler warning with -Wall flag showing uninitialized variable

Example 4: Use After Free

```c
#include<stdlib.h>
#include<stdio.h>

int main(int argc, char *argv[]) {
    int arrLen = 10;
    int* arr;
    int i;
    arr = (int*) malloc(arrLen*sizeof(int));

    for(i=0; i<arrLen; i++) {
        arr[i] = i;
    }
    free(arr);

    for(i=0; i<arrLen; i++) {
        arr[i] = i*2;
    }

    printf("Done\n");
    return 0;
}
```

Figure 18: C code attempting to access memory after freeing it

```
maor@maor:~/os/tirgul1_files$ valgrind ./test
==66989== Memcheck, a memory error detector
==66989== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==66989== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==66989== Command: ./test
==66989==
==66989== Invalid write of size 4
==66989==    at 0x109214: main (example9.c:17)
==66989==  Address 0x4a9a040 is 0 bytes inside a block of size 40 free'd
==66989==    at 0x484988F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==66989==    by 0x1091F1: main (example9.c:14)
==66989==  Block was alloc'd at
==66989==    at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==66989==    by 0x1091B3: main (example9.c:9)
==66989==
Done
==66989==
==66989== HEAP SUMMARY:
==66989==     in use at exit: 0 bytes in 0 blocks
==66989==   total heap usage: 2 allocs, 2 frees, 1,064 bytes allocated
==66989==
==66989== All heap blocks were freed -- no leaks are possible
==66989==
==66989== For lists of detected and suppressed errors, rerun with: -s
==66989== ERROR SUMMARY: 10 errors from 1 contexts (suppressed: 0 from 0)
maor@maor:~/os/tirgul1_files$
```

Figure 19: Valgrind output showing invalid memory access after free

This example demonstrates accessing memory that has already been freed, which Valgrind correctly identifies as an error.

Example 5: Double Free

```
maor@maor:~/os/tirgul1_files$ ./test
free(): double free detected in tcache 2
Aborted (core dumped)
maor@maor:~/os/tirgul1_files$
```

Figure 20: C code attempting to free the same memory twice

Attempting to free the same memory location twice is an error that the standard C library can detect:

```
maor@maor:~/os/tirgul1_files$ ./test
free(): double free detected in tcache 2
Aborted (core dumped)
maor@maor:~/os/tirgul1_files$
```

Figure 21: Runtime error from standard library showing double free detection

The tcache (thread cache) mentioned in the error message is part of glibc's malloc implementation, designed to improve memory allocation performance by caching freed memory chunks per thread.

Valgrind provides additional details about this error:

```
maor@maor:~/os/tirgul1_files$ valgrind ./test
==67085== Memcheck, a memory error detector
==67085== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==67085== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==67085== Command: ./test
==67085==
==67085== Invalid free() / delete / delete[] / realloc()
==67085==    at 0x484988F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==67085==    by 0x10925E: main (example10.c:22)
==67085==  Address 0x4a9a040 is 0 bytes inside a block of size 40 free'd
==67085==    at 0x484988F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==67085==    by 0x1091DC: compute (example10.c:10)
==67085==    by 0x10924F: main (example10.c:21)
==67085==  Block was alloc'd at
==67085==    at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==67085==    by 0x10920C: main (example10.c:17)
==67085==
sum = 45==67085==
==67085== HEAP SUMMARY:
==67085==     in use at exit: 0 bytes in 0 blocks
==67085==   total heap usage: 2 allocs, 3 frees, 1,064 bytes allocated
==67085==
==67085== All heap blocks were freed -- no leaks are possible
==67085==
==67085== For lists of detected and suppressed errors, rerun with: -s
==67085== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
maor@maor:~/os/tirgul1_files$
```

Figure 22: Valgrind output showing double free error

Using an Interactive Debugger

While Valgrind excels at detecting memory-related issues, interactive debuggers like CLion's debugger provide additional tools for examining program state during execution. Though we are encouraged to use CLion's debugger, we should avoid relying on its Run button for normal execution.

The following example demonstrates how to use an interactive debugger:

```c
#include<stdlib.h>
#include<stdio.h>

typedef struct Foo {
   int arr[3];
   int bar;
} Foo;

int main(int argc, char *argv[]) {
   setbuf(stdout, 0);
   Foo t;
   int i;
   printf("start\n");
   t.bar = 6;
   printf("t.bar =  %d\n", t.bar);
   for (i=0; i<3; i++) {
      t.arr[i] = i+1;
   }
   t.arr[i] += t.arr[0] + t.arr[1];
   printf("t.arr[2] = %d\n", t.arr[2]);
   printf("t.bar = %d\n", t.bar);
   return 0;
}
```

Figure 23: Example code with struct for debugging demonstration

Breakpoints

A breakpoint is a marker that tells the debugger to pause execution when it reaches that point. This pause allows you to inspect the program's state, including variable values, memory, and the call stack.

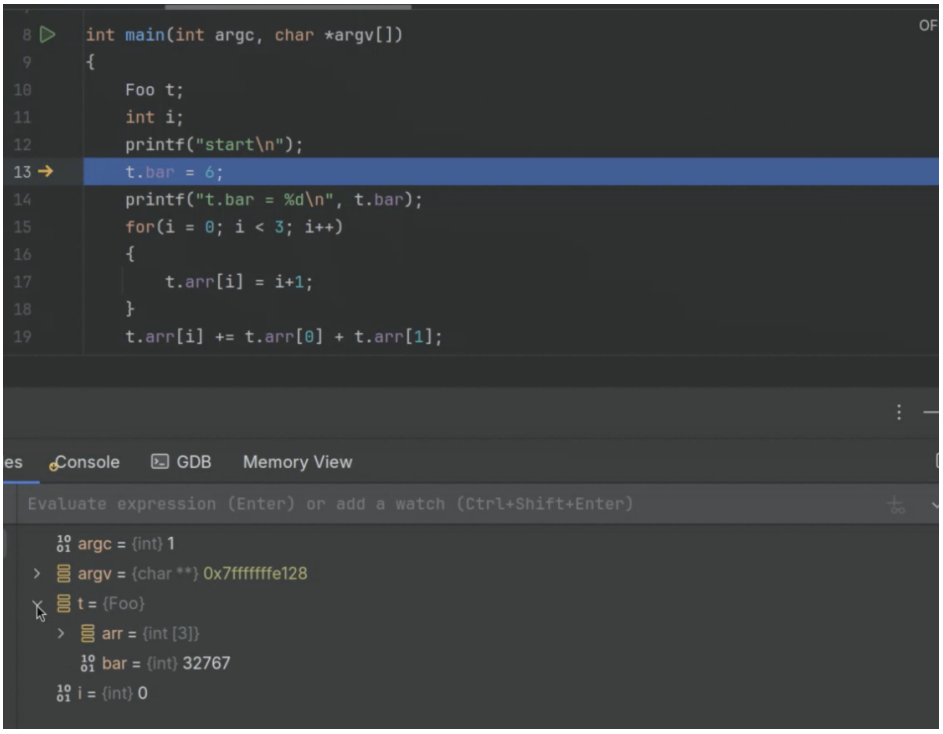

Figure 24: Setting a breakpoint in CLion

Watchpoints

Unlike standard breakpoints that pause execution at specific lines of code, watchpoints monitor memory locations and trigger when data changes.

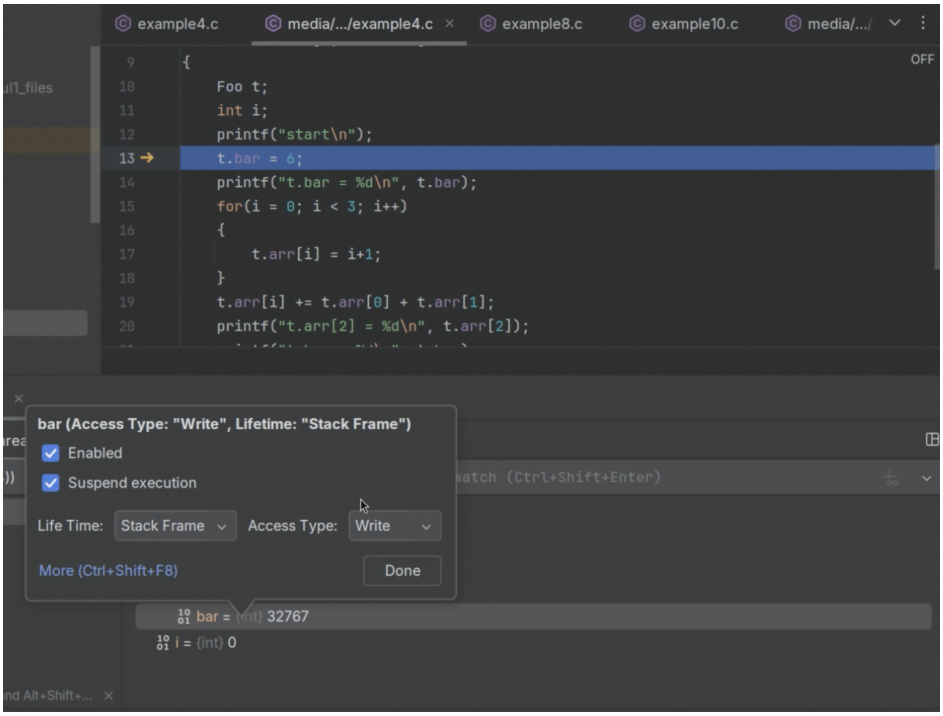

Figure 25: Setting a watchpoint in CLion to monitor variable changes

Watchpoints are particularly useful for tracking when specific variables change. The "Access Type: Write" setting configures the watchpoint to trigger only when the variable's value is modified, helping to identify exactly where in the code changes occur.