VULNERABILITĂȚI ÎN PROGRAMAREA CONCURENTĂ

Programarea concurentă implică execuția, respectiv progresul mai multor task-uri în același interval de timp, dar nu neapărat în același timp. Un proces poate fi divizat în mai multe thread-uri, ce reprezintă cea mai mică unitate de execuție. Fiecare thread sau task constă într-un flux separat de control al programului. Pe un sistem cu resurse reduse (un procesor cu un singur core), aceste task-uri se execută intercalat, într-o manieră preemptivă, întrerupându-se reciproc în funcție de criteriile și algoritmul planificatorului sistemului de operare, fiecare având o anumită cuantă de timp de procesor. Pe un sistem multicore, task-urile se pot executa și simultan.

Programarea concurentă a fost întotdeauna un proces dificil și predispus la erori, chiar și în absența preocupărilor legate de securitate. Multe dintre defectele software pot fi, de asemenea, folosite ca vectori de atac pentru diverse exploatări.

Bug-urile cauzate de concurență au un grad mic de reproductibilitate. Este dificil a le face să se întâmple la fel de două ori. Intercalarea instrucțiunilor este puternic influențată de mediu, de alte programe care rulează, de traficul de rețea, de deciziile de planificare ale sistemului de operare, de variațiile de viteză ale ceasului procesorului. De fiecare dată când este executat un program care conține o condiție de cursă, este posibil un comportament diferit.

Aceste tipuri de bug-uri sunt ”heisenbug-uri”, care sunt nedeterministe și greu de reprodus, spre deosebire de un "bohrbug", care apare în mod repetat ori de câte ori te uiți la el. Aproape toate bug-urile din programarea secvențială sunt bohrbugs.

Un heisenbug poate chiar să dispară atunci când se încearcă detectarea cu println sau un depanator. Motivul este că afișarea și depanarea sunt mult mai lente decât alte operațiuni, adesea de 100-1000x mai lente, încât schimbă dramatic timpul de desfășurare al operațiunilor și intercalarea.

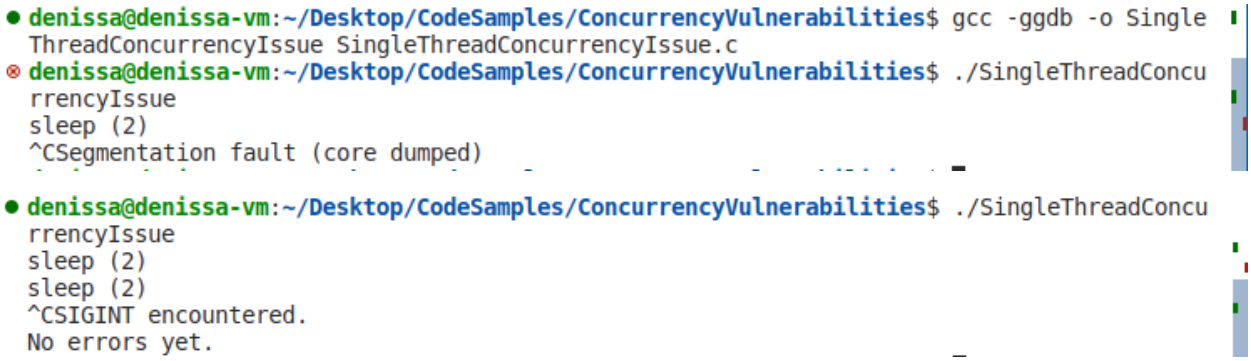
VULNERABILITĂȚI ALE PROGRAMELOR SINGLE-THREAD

UTILIZAREA UNEI RUTINE DE TRATARE A SEMNALULUI CARE NU ESTE ASYNCHRONOUS-SAFE ȘI REENTRANTĂ

Rutinele de tratare a semnalelor funcționează la nivelul sistemului de operare și oferă un mecanism prin care un program să răspundă la evenimente în afara fluxului său normal de control. Un semnal este o întrerupere generată de software care este trimisă unui proces de către sistemul de operare din cauza faptului că utilizatorul apasă ctrl-c sau un alt proces îl declanșează. Când este livrat un semnal, execuția normală a programului este întreruptă, iar handler-ul preia controlul imediat, rulând asincron (fără a aștepta ca executarea funcției curente să se termine). Această întrerupere se poate întâmpla în orice moment în timpul executării programului.

Următorul program demonstrează concurența intercalată prin faptul că un singur flux de execuție poate avea loc la un moment dat. Programul manifestă un comportament nedefinit care rezultă din utilizarea unui signal handler ce cauzează probleme de concurență fără multi-threading.

```
10  void handler(int signum) {
11      strcpy(err_msg, "SIGINT encountered.");
12      printf("%s\n", err_msg);
13  }
14
15  int main(void) {
16
17      signal(SIGINT, handler);
18
19      printf("sleep (2)\n");
20      sleep(2);
21
22      err_msg = (char *)malloc(MAX_MSG_SIZE);
23      if (err_msg == NULL) {
24          return 1;
25      }
26
27      printf("sleep (2)\n");
28      sleep(2);
29
30      strcpy(err_msg, "No errors yet.");
31      printf("%s\n", err_msg);
32
33      free(err_msg);
34      return 0;
35  }
```



Deși utilizează un singur fir de execuție, acest program are două fluxuri de execuție: unul care utilizează funcția main() și unul care utilizează funcția handler(). Dacă rutina de tratare a semnalului este invocată înainte sau în timpul apelului către malloc(), programul își va termina execuția cu o eroare de tip segmentation fault. Handler-ul poate fi invocat între apelurile malloc() și strcpy(), comportamentul acestuia fiind mascat ulterior de funcția strcpy() din main() care suprascrie aceeași zonă de memorie, rezultatul fiind că err_msg conține "No errors yet". De aceea, este recomandată utilizarea de funcții sigure asincrone în cadrul rutinelor de tratare a semnalelor. Rutinele de tratare a semnalelor sunt adesea asincrone, ceea ce înseamnă că pot întrerupe fluxul normal al unui program pentru a răspunde la semnale. Funcțiile sigure asincrone sunt proiectate pentru a fi sigure pentru utilizarea în cadrul handler-elor. O bună practică este să fie cât mai simple și concise posibil, să evite alocarea dinamică a memoriei și să limiteze utilizarea resurselor partajate sau a variabilelor globale pentru a reduce riscul unui comportament neașteptat într-un context de tratare a semnalelor.

Deoarece rutinele de tratare a semnalelor pot fi întrerupte la rândul lor, chiar de ele înseși, este esențială și evitarea funcțiilor care nu sunt reentrante. O funcție este reentrantă dacă mai multe apeluri ale aceleiași funcții pot rula simultan în același spațiu de adrese (același proces) fără a crea stări inconsecvente, perturbând contextul unui alt apel și provocând un comportament neașteptat. O funcție reentrantă poate fi întreruptă în siguranță și apoi reluată fără a provoca probleme. Aceasta nu modifică starea mediului de execuție prin modificarea unui obiect, în special modificarea structurilor globale de date, modificarea unui fișier sau apelarea unei funcții care face oricare dintre aceste operațiuni.

De exemplu, o rutină de tratare a semnalului sigură asincronă nu modifică o variabilă globală care este modificată în cadrul programului, în fluxul principal, dar poate modifica o altă variabilă globală. Deși este sigură asincronă, nu este reentrantă, din cauză că la o întrerupere ulterioară cauzată de același semnal poate apărea o condiție de cursă. Pentru a fi reentrantă trebuie garantat faptul că nu modifică nicio variabilă globală.

VULNERABILITĂȚI ALE PROGRAMELOR MULTI-THREAD

UTILIZAREA UNEI FUNCȚII CARE ESTE THREAD-SAFE DAR NU ȘI REENTRANTĂ

A fi o funcție thread-safe sau reentrantă reprezintă concepte similare, dar au câteva diferențe importante. Funcțiile reentrante sunt thread-safe, dar funcțiile thread-safe nu sunt întotdeauna reentrante.

O funcție este considerată thread-safe atunci când poate fi apelată în siguranță de mai multe fire de execuție simultan fără a provoca probleme cum ar fi condițiile de cursă sau coruperea datelor partajate. Funcția increment_counter din exemplu este thread-safe deoarece utilizează un mutex (pthread_mutex_t) pentru a sincroniza accesul la variabila partajată count. Atunci când mai multe fire de execuție apelează increment_counter, mutexul se asigură că doar un fir poate incrementa la un moment dat numărul, prevenind astfel problemele de acces concurențial.

O funcție este reentrantă dacă poate fi întreruptă în mijlocul execuției sale și apelată din nou în siguranță, înainte de finalizarea execuțiilor anterioare. Pentru ca o funcție să fie reentrantă, aceasta nu trebuie să mențină nicio stare statică sau globală și nu trebuie să se bazeze pe blocaje de resurse care ar putea provoca și blocarea acesteia. În consecință, funcția increment_counter nu este și reentrantă.

```
6   static int count = 0;
7   static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8
9   int increment_counter()
10  {
11      pthread_mutex_lock(&mutex);
12      printf("Mutex locked\n");
13      sleep(1);
14      count++;
15      int result = count;
16      pthread_mutex_unlock(&mutex);
17      return result;
18  }
19
20  void *thread_function(void *arg)
21  {
22      for (int i = 0; i < 10; ++i)
23      {
24          int value = increment_counter();
25          printf("Thread %d incremented counter to %d\n", *(int *)arg, value);
26      }
27      return NULL;
28  }
29
30  void signal_handler(int signal)
31  {
32      int value = increment_counter();
33      printf("Signal received. Counter incremented to %d\n", value);
34  }
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities/Reentrancy$ gcc -ggdb -o Thread
SafeButNotReentrant ThreadSafeButNotReentrant.c -lpthread
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities/Reentrancy$ ./ThreadSafeButNotR
eentrant
Mutex locked
Thread 1 incremented counter to 1
Mutex locked
Thread 1 incremented counter to 2
Mutex locked
Thread 1 incremented counter to 3
Mutex locked
Thread 1 incremented counter to 4
Mutex locked
Thread 1 incremented counter to 5
Mutex locked
Thread 1 incremented counter to 6
Mutex locked
^CThread 1 incremented counter to 7
Mutex locked
Thread 1 incremented counter to 8
Mutex locked
^C
```

Scenariul care se întâmplă constă în apelarea funcției dintr-un handler de semnal. Rutina de tratare a semnalului răspunde la semnalul SIGINT (declanșat de obicei prin apăsarea Ctrl+C). Funcția increment_counter nu este reentrantă din cauza blocării mutex-ului și incapacitatea execuției acesteia în contextul curent.

Funcția își începe execuția și acaparează mutex-ul. Înainte de a putea elibera mutex-ul, apare o întrerupere, iar rutina de tratare apelează funcția increment_counter apelează din nou. Acest apel încearcă să acapareze din nou mutex-ul, însă, din cauză că mutex-ul este deja blocat de primul apel, al doilea apel nu își poate continua execuția. Planificatorul nu poate aloca din nou timp de procesor programului deoarece întreruperea are o prioritate mai mare. Acest lucru duce la un deadlock, din cauză că primul apel nu se poate finaliza și elibera mutex-ul până la finalizarea celui de-al doilea apel, dar al doilea apel nu își poate continua execuția din cauză că așteaptă eliberarea mutex-ului.

Astfel, utilizarea incorectă a primitivelor de sincronizare poate rezulta într-un deadlock. Deadlock-ul apare ori de câte ori două sau mai multe fluxuri de control se blochează reciproc în așa fel încât niciunul nu poate continua să se execute.

PREZENȚA UNEI CONDIȚII DE CURSĂ

Secvența de cod următoare reprezintă un exemplu ce conține o funcție care nu este thread-safe din cauza existenței unei condiții de cursă.

```
4   static int UnitsSold = 0;
5
6   void IncrementUnitsSold()
7   {
8       UnitsSold = UnitsSold + 1;
9   }
10
11  int main()
12  {
13      const int NUM_THREADS = 100;
14      std::thread threads[NUM_THREADS];
15
16      for (int i = 0; i < NUM_THREADS; ++i)
17      {
18          threads[i] = std::thread(IncrementUnitsSold);
19      }
20
21      for (int i = 0; i < NUM_THREADS; ++i)
22      {
23          threads[i].join();
24      }
25
26      if (UnitsSold < NUM_THREADS)
27          std::cout << "UnitsSold: " << UnitsSold << " !!!!!!!" << std::
28      else
29          std::cout << "UnitsSold: " << UnitsSold << std::endl;
30
31      return 0;
32  }
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities$ g++ -ggdb -o Unsafe
IncrementBefore UnsafeIncrementBefore.cpp -pthread
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities$ for i in {1..500};
do ./UnsafeIncrementBefore >> output.txt; done
```

output.txt

```
156   UnitsSold: 100
157   UnitsSold: 100
158   UnitsSold: 100
159   UnitsSold: 100
160   UnitsSold: 99 !!!!!!!
161   UnitsSold: 100
162   UnitsSold: 100
163   UnitsSold: 100
```

Codul furnizat prezintă o condiție de cursă din cauză că implică o operație non-atomică pe variabila partajată, UnitsSold, fără sincronizare adecvată.

Incrementarea implică o operație de citire-modificare-scriere care nu este garantată a fi atomică în absența unei sincronizări adecvate. Aceasta implică citirea valorii curente a UnitsSold din locația din memorie în registrul CPU, incrementarea valorii din registru și apoi scrierea înapoi în locația de memorie.

Dacă două thread-uri execută această operație simultan, ambele pot citi aceeași valoare inițială, ducând la o condiție de cursă în care incrementările sunt pierdute.

Timp	Thread 1	Thread 2
T0	Intră în funcția IncrementUnitsSold()	
T1		Intră în funcția IncrementUnitsSold()
T2		Load (UnitsSold = 0)
T3	Load (UnitsSold = 0)	
T4		Increment (UnitsSold = 1)
T5		Store (UnitsSold = 1)
T6	Increment (UnitsSold = 1)	
T7	Store (UnitsSold = 1)	
T8	Return	
T9		Return

După ce ambele fire de execuție invocă funcția IncrementUnitsSold(), variabila UnitsSold ar trebui să fie 2, dar în schimb este 1 din cauza condiției inerente de cursă care rezultă din nesincronizare.

Pentru a elimina condicția de cursă, programul a fost modificat prin utilizarea operației atomice fetch_add pe variabila atomică UnitsSold. Acest lucru garantează că operația de incrementare este indivizibilă și nu poate fi întreruptă de alte fire de execuție.

```
5   static std::atomic<int> UnitsSold = 0;
6
7   void IncrementUnitsSold()
8   {
9       UnitsSold.fetch_add(1, std::memory_order_relaxed);
10  }
```

```
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities$ g++ -ggdb -o Unsafe
IncrementAfter UnsafeIncrementAfter.cpp -pthread
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities$ for i in {1..500};
do ./UnsafeIncrementAfter >> output.txt; done
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities$ egrep -c '!!!' outp
ut.txt
0
```

Alte operații atomice furnizate de std::atomic includ load, store, exchange și diverse operații de comparare și schimb precum compare_exchange_weak sau compare_exchange_strong. Alegerea operațiunii depinde de cerințele specifice ale codului. Condiția de cursă anterioară poate fi, de asemenea, eliminată și prin utilizarea operației compare_exchange_strong pe variabila atomică UnitsSold.

```
5   static std::atomic<int> UnitsSold = 0;
6
7   void IncrementUnitsSold()
8   {
9       int expected;
10      do
11      {
12          expected = UnitsSold.load(std::memory_order_relaxed);
13      } while (!UnitsSold.compare_exchange_strong(expected, expected + 1,
14  }
```

Bucla do-while încearcă să incrementeze UnitsSold. Metoda compare_exchange_strong compară atomic valoarea variabilei UnitsSold cu valoarea variabilei expected; dacă acestea sunt egale, setează UnitsSold cu expected + 1. Dacă comparația nu reușește (ceea ce înseamnă că un alt fir de execuție a modificat între timp UnitsSold), expected este actualizat automat cu noua valoare a UnitsSold, iar bucla reîncearcă operațiunea.

Argumentul std::memory_order_relaxed specifică ordonarea memoriei pentru această operație, indicând faptul că nu se aplică constrângeri în ceea ce privește operațiile cu memoria. Diferite arhitecturi hardware și compilatoare pot reordona operațiunile cu memoria în scopuri de optimizare.

Șablonul std::atomic în C++ este conceput pentru a utiliza cele mai eficiente instrucțiuni disponibile pe arhitectura țintă pentru a îndeplini garanțiile atomice necesare. Codul mașină generat de compilator utilizează instrucțiunile atomice ale procesorului pentru a efectua operații pe variabilele atomice. Majoritatea procesoarelor CISC moderne, cum ar fi arhitecturile x86/x64, oferă suport hardware pentru operațiunile atomice. Procesoarele RISC, cum ar fi cele bazate pe arhitectura ARM, oferă, de asemenea, instrucțiuni pentru operațiile atomice. Cu toate acestea, dacă un procesor RISC nu acceptă anumite operații atomice în mod nativ, deoarece, după cum sugerează și numele, are un set redus de instrucțiuni, implementarea bibliotecii standard C ++ poate utiliza într-adevăr mutex-uri sau alte mecanisme de sincronizare pentru a asigura atomicitatea.

În arhitectura x86, funcția fetch_add pe o variabilă atomică în C++ se traduce de obicei printr-o instrucțiune LOCK XADD la nivel de asamblare. Când este utilizat cu prefixul LOCK, instrucțiunea XADD garantează că operația este atomică pe mai multe procesoare. Aceasta înseamnă că, în timp ce instrucțiunea XADD se execută, niciun alt procesor nu poate accesa locația memoriei modificate. Acest lucru este esențial pentru asigurarea conceptului de thread-safety într-un mediu multi-core sau multi-procesor.

Pe arhitecturile x86, operația compare_exchange_strong pe variabile atomice în C++ este de obicei tradusă în instrucțiunea CMPXCHG, adesea combinată cu prefixul LOCK pentru sincronizarea multi-procesor.

Condiția de cursă poate fi eliminată și prin utilizarea unui mutex în funcția IncrementUnitsSold pentru a sincroniza accesul la variabila partajată UnitsSold. Mutexul garantează că un singur thread poate executa operația de incrementare pe UnitsSold la un moment dat.

```
5   static int UnitsSold = 0;
6   static std::mutex unitsSoldMutex;
7
8   void IncrementUnitsSold()
9   {
10      std::lock_guard<std::mutex> lock(unitsSoldMutex);
11      UnitsSold = UnitsSold + 1;
12  }
```

Un std::lock_guard este utilizat pentru a acapara mutex-ul. Acesta îl eliberează automat atunci când se iese din domeniul de valabilitate (la sfârșitul funcției), asigurându-se că mutex-ul este întotdeauna eliberat corespunzător, chiar dacă este aruncată o excepție.

Deși această abordare de blocare este thread-safe, poate introduce o potențială contenție în alte scenarii de concurență ridicată și, prin urmare, va fi mai puțin eficientă în comparație cu alternativele care utilizează instrucțiuni atomice. Contenția apare atunci când un fir de execuție încearcă să acapareze un mutex deja acaparat. Performanța slabă poate fi rezolvată prin reducerea timpului de acaparare sau prin reducerea cantității de resurse protejate de mutecși. Cu cât este acaparat mai mult un mutex, cu atât este mai mare probabilitatea ca un alt fir să încerce să îl acapareze și să fie forțat să aștepte. Algoritmii fără blocare pot fi mai scalabili în scenarii cu concurență ridicată în comparație cu modelele care utilizează blocări.

OPTIMIZAREA CITIRILOR DIN MEMORIE A VARIABILELOR LA COMPILARE

Următorul program se bazează pe recepționarea unui semnal SIGINT pentru a comuta un flag de terminare a buclei while. Cu toate acestea, citirea variabilei interrupt în main() poate fi optimizată de compilator, în ciuda atribuirii valorii 1 în handler-ul de semnal, iar bucla nu se va termina niciodată.

```
3   int interrupted = 0;
4
5   void handler(int signum) {
6       interrupted = 1;
7   }
8
9   int main() {
10      signal(SIGINT, handler);
11      while (!interrupted) {
12          /* do something */
13      }
14      return 0;
15  }
```

Când este compilat cu GCC și flag-ul de optimizare -O0, programul reușește să se termine prin recepționarea semnalului SIGINT deoarece citește, ulterior întreruperii, valoarea variabilei interrupted.

```
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities/CompilerOptimized
Read$ gcc -ggdb -o CompilerOptimizedReadBefore CompilerOptimizedReadBefore.c -O0
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities/CompilerOptimized
Read$ ./CompilerOptimizedReadBefore
^Cdenissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities/CompilerOptimiz

Dump of assembler code for function main:
   0x0000000000001161 <+0>:     endbr64
   0x0000000000001165 <+4>:     push   %rbp
   0x0000000000001166 <+5>:     mov    %rsp,%rbp
   0x0000000000001169 <+8>:     lea    -0x27(%rip),%rax        # 0x1149 <handler>
   0x0000000000001170 <+15>:    mov    %rax,%rsi
   0x0000000000001173 <+18>:    mov    $0x2,%edi
   0x0000000000001178 <+23>:    call   0x1050 <signal@plt>
   0x000000000000117d <+28>:    nop
   0x000000000000117e <+29>:    mov    0x2e90(%rip),%eax        # 0x4014 <interrupted>
   0x0000000000001184 <+35>:    test   %eax,%eax
   0x0000000000001186 <+37>:    je     0x117e <main+29>
   0x0000000000001188 <+39>:    mov    $0x0,%eax
   0x000000000000118d <+44>:    pop    %rbp
   0x000000000000118e <+45>:    ret
End of assembler dump.
```

Când este compilat cu GCC și flag-ul de optimizare -O1, programul nu reușește să se termine la primirea unui semnal SIGINT.

```
^Cdenissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities/CompilerOptimiz
edReadgcc -ggdb -o CompilerOptimizedReadBefore CompilerOptimizedReadBefore.c -O1
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities/CompilerOptimized
Read$ ./CompilerOptimizedReadBefore
^C^C^C^C^C^C^C^C^C^C^C
```

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000001158 <+0>:     endbr64
   0x000000000000115c <+4>:     sub    $0x8,%rsp
   0x0000000000001160 <+8>:     lea    -0x1e(%rip),%rsi        # 0x1149 <handler>
   0x0000000000001167 <+15>:    mov    $0x2,%edi
   0x000000000000116c <+20>:    call   0x1050 <signal@plt>
   0x0000000000001171 <+25>:    mov    0x2e9d(%rip),%eax        # 0x4014 <interrupted>
   0x0000000000001177 <+31>:    test   %eax,%eax
   0x0000000000001179 <+33>:    je     0x1177 <main+31>
   0x000000000000117b <+35>:    mov    $0x0,%eax
   0x0000000000001180 <+40>:    add    $0x8,%rsp
   0x0000000000001184 <+44>:    ret
End of assembler dump.
```

Cuvântul cheie volatile este folosit pentru a indica compilatorului că valoarea unei variabile se poate schimba neașteptat, un caz des întâlnit atunci când o variabilă este accesată de mai multe thread-uri sau prin gestionarea asincronă a semnalului. Cea mai evidentă utilizare a cuvântului cheie volatile este alături de o variabilă globală sau statică utilizată într-o rutină de tratare a unei întreruperi. Cuvântul cheie volatile garantează faptul că valoarea unei variabile partajate este întotdeauna citită și scrisă din și către locația sa de memorie, împiedicând optimizările.

Atunci când o variabilă nu este declarată volatilă, compilatorul și hardware-ul pot face anumite presupuneri cu privire la comportamentul variabilei. O optimizare comună implică memorarea acesteia în regiștrii CPU sau cache-urile CPU pentru a îmbunătăți performanța. Aceasta este o tehnică comună de optimizare pentru a reduce timpul de acces la memorie. Cu toate acestea, dacă valoarea variabilei poate fi modificată de factori externi asemenea întreruperilor, valoarea memorată în cache poate să nu reflecte cea mai recentă valoare. Procesoarele moderne au diferite niveluri de cache (L1, L2, L3) pentru a stoca datele accesate frecvent.

Prin adăugarea calificatorului volatile la declarația variabilei interrupted, este garantat accesul de la adresa sa originală pentru fiecare iterație a buclei, precum și din interiorul handler-ului de semnal. Acesta forțează compilatorul să recitească valoarea din memorie la fiecare acces, chiar dacă fluxul normal de cod nu modifică valoarea. În acest caz, compilatorul presupune că din moment ce valoarea variabilei interrupted nu a fost modificată în buclă este în regulă să păstreze valoarea într-un registru.

```c
volatile int interrupted = 0;

void handler(int signum) {
    interrupted = 1;
}

int main() {
    signal(SIGINT, handler);
    while (!interrupted) {
        /* do something */
    }
    return 0;
}
```

```
^Cdenissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities/CompilerOptimiz
edReadgcc -ggdb -o CompilerOptimizedReadAfter CompilerOptimizedReadAfter.c -O1
denissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities/CompilerOptimized
Read$ ./CompilerOptimizedReadAfter
^Cdenissa@denissa-vm:~/Desktop/CodeSamples/ConcurrencyVulnerabilities/CompilerOptimiz
```

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000001158 <+0>:     endbr64
   0x000000000000115c <+4>:     sub    $0x8,%rsp
   0x0000000000001160 <+8>:     lea    -0x1e(%rip),%rsi        # 0x1149 <handler>
   0x0000000000001167 <+15>:    mov    $0x2,%edi
   0x000000000000116c <+20>:    call   0x1050 <signal@plt>
   0x0000000000001171 <+25>:    mov    0x2e9d(%rip),%eax        # 0x4014 <interrupted>
   0x0000000000001177 <+31>:    test   %eax,%eax
   0x0000000000001179 <+33>:    je     0x1171 <main+25>
   0x000000000000117b <+35>:    mov    $0x0,%eax
   0x0000000000001180 <+40>:    add    $0x8,%rsp
   0x0000000000001184 <+44>:    ret
End of assembler dump.
```

Cuvântul cheie volatile nu oferă thread-safety atunci când operațiile non-atomice sunt efectuate pe variabile partajate. Operații precum incrementarea și decrementarea sunt operații compuse. Aceste operații implică trei pași: citirea valorii variabilei, actualizarea acesteia și apoi scrierea valorii actualizate înapoi în memorie. Decalajul scurt de timp dintre citirea valorii și scrierea noii valori înapoi în memorie poate crea o condiție de cursă. Firele de execuție care lucrează pe aceeași variabilă pot citi și opera pe valoarea mai veche în acel interval de timp și își pot suprascrie reciproc rezultatele.

Calificativul volatile oferă thread-safety în două scenarii multi-threading. Când un singur thread scrie variabila volatilă și alte thread-uri îi citesc valoarea, thread-urile care citesc văd cea mai recentă valoare a variabilei. Un alt scenariu e reprezentat de când mai multe thread-uri scriu o variabilă partajată, într-un mod pseudo-atomic, precum o asignare, astfel încât noua valoare scrisă nu depinde de valoarea anterioară.

Utilizarea cuvântului cheie volatile poate duce la performanțe reduse, deoarece compilatorul nu poate optimiza codul la fel de mult pe cât ar putea fără cuvântul cheie. Acest lucru poate duce la o execuție mai lentă a codului și la o utilizare crescută a memoriei.

Dacă este cunoscut faptul că rutina de tratare a întreruperii nu va fi întreruptă de o alta care accesează, de asemenea, aceeași variabilă globală se pot utiliza în siguranță variabile locale nevolatile pentru calcule intermediare. Astfel, se vor face mai puține operațiuni cu memoria.

În acest exemplu simplu, beneficiul ar putea să nu fie semnificativ, dar în rutine mai complexe, această practică poate duce la un cod mai eficient.

C CompilerOptimizedReadAfterLocal.c > ...

```c
#include <signal.h>

volatile int interrupted = 0;

void handler(int signum) {
    int localInterrupted = interrupted;
    localInterrupted = 1;
    interrupted = localInterrupted;
}

int main() {
    signal(SIGINT, handler);
    while (!interrupted) {
        /* do something */
    }
    return 0;
}
```

Înainte de ieșirea din rutina de tratare a întreruperii, variabila volatilă globală interrupted este actualizată cu valoarea localInterrupted. Acest lucru asigură că bucla principală va vedea valoarea actualizată.