به نام خدا

گزارش تمرین کامپیوتری
شماره پنج
برنامه نویسی موازی

سید علیرضا میرشفیعی - ۸۱۰۱۰۱۵۳۲
محمد صدرا عباسی - ۸۱۰۱۰۱۴۶۹

بخش اول

برای تحلیل کد سریال که به صورت بازگشتی نوشته شده آن رو در حالت دیباگ بیلد می کنیم و سپس از ابزار vtune برای تحلیل hotspot های برنامه استفاده می کنیم که نتیجه گزارش به این شکل خواهد بود:

Elapsed Time: 52.337s

CPU Time: 7.365s
Total Thread Count: 1
Paused Time: 5.348s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
std::operator<<<struct std::char_traits<char> >	Part1.exe	6.083s	82.6%
std::basic_ostream<char,struct std::char_traits<char> >::operator<<	MSVCP140D.dll	1.001s	13.6%
malloc	ucrtbased.dll	0.138s	1.9%
free_dbg	ucrtbased.dll	0.070s	0.9%
displayMaze	Part1.exe	0.037s	0.5%
[Others]	N/A*	0.037s	0.5%

*N/A is applied to non-summable metrics.

این نشان می دهد که گلوگاه اول فراخوانی مداوم تابع displayMaze است که به طور مکرر از cout استفاده می کند در نتیجه این تابع را موقتا کامنت می کنیم تا هات اسپات های دیگر را پیدا کنیم:

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
malloc	ucrtbased.dll	10.993s	47.1%
free_dbg	ucrtbased.dll	9.402s	40.3%
copyVisited	Part1.exe	1.354s	5.8%
findAllPaths	Part1.exe	0.750s	3.2%
isValidMove	Part1.exe	0.422s	1.8%
[Others]	N/A*	0.412s	1.8%

*N/A is applied to non-summable metrics.

هات اسپات بعدی کپی مداوم ماتریس برای هر فراخوانی بازگشتی است که لزومی به دیپ کپی کل ماتریس برای شاخه ها وجود ندارد کافی است یکبار حافظه برای آن بگیریم در هر step مقدار visited مربوطه را یک کنیم و در گام بازگشت visited را دوباره صفر کنیم:

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
isValidMove	Part1.exe	8.748s	57.1%
findAllPaths	Part1.exe	6.574s	42.9%

*N/A is applied to non-summable metrics.

```
void findAllPaths(int x, int y, int endX, int endY,
    int** maze, int** visited)
{
    if (x == endX && y == endY) {
        visited[x][y] = 1;
        //displayMaze(maze, visited);
        totalPaths++;
        visited[x][y] = 0;
        return;
    }

    visited[x][y] = 1;

    for (int dir = 0; dir < 4; dir++) {
        int newX = x + dx[dir];
        int newY = y + dy[dir];

        if (isValidMove(newX, newY, maze, visited)) {
            // delete copy func call
            findAllPaths(newX, newY, endX, endY, maze, visited);
        }
    }
```

for (int dir = 0; dir < 4; dir++) {	11.4%	1.74
int newX = x + dx[dir];	6.5%	1.00
int newY = y + dy[dir];	8.9%	1.36
if (isValidMove(newX, newY, maze, visited)) {	4.8%	0.73
// delete copy func call		
findAllPaths(newX, newY, endX, endY, maze, visited);	42.9%	0.18
}		
}	1.7%	0.25

`bool isValidMove(int x, int y, int** maze, int** visited)`	
`{`	5.6%
`    if (x >= 0 && x < rows && y >= 0 && y < cols) {`	13.4%
`        if (maze[x][y] == 1 && visited[x][y] == 0) {`	23.8%
`            return true;`	4.5%
`        }`	
`    }`	1.9%
`    return false;`	2.6%
`}`	5.3%

زمان زیادی از cpu صرف اجرای تابع findAllPaths شده است پس می تواند یک گزینه برای موازی سازی باشد

موازی سازی findAllPaths با استفاده از openmp:

برای موازی سازی این تابع بازگشتی بهترین روش این است که تسک هر شاخه را به یک ترد جداگانه assign کنیم اما باید برعکس نسخه سریال که از یک ماتریس visited مشترک استفاده می کرد به هر ترد نسخه کپی از این ماتریس بدهیم تا از وقوع شرایط race جلوگیری کنیم پس سربار حافظه خواهیم داشت که باعث ایجاد trade off بین سربار حافظه برای deepcopy ماتریس و کاهش زمان اجرا با موازی سازی می شود. می توان نتیجه گرفت که چالش اصلی پیدا کردن مرزی است که بهترین performance را داشته باشد.

```
void findAllPaths(int x, int y, int endX, int endY,
    int** maze, int** visited, int depth)
{
    if (x == endX && y == endY) {
        //displayMaze(maze, visited);
        #pragma omp atomic
        totalPaths++;
        return;
    }
```

چون هر ترد تابع را درون خود اجرا می کند باید افزایش مقدار totalPaths که یک متغیر shared است به صورت اتومیک انجام شود(جلوگیری از race)

```
if (isValidMove(newX, newY, maze, visited)) {
    if (depth < maxDepth) {
        int** visitedCopy = new int* [rows];
        for (int i = 0; i < rows; i++) {
            visitedCopy[i] = new int[cols];
        }

        copyVisited(visited, visitedCopy);

        #pragma omp task firstprivate(visitedCopy, newX, newY, depth)
        {
            findAllPaths(newX, newY, endX, endY, maze, visitedCopy, depth + 1);
            // free(visitedCopy);
            for (int i = 0; i < rows; i++) {
                delete[] visitedCopy[i];
            }
            delete[] visitedCopy;
        }
    }
    else {
        findAllPaths(newX, newY, endX, endY, maze, visited, depth + 1);
    }
}
```

برای موازی سازی از ساختار task استفاده شده زیرا بار محاسباتی و سایز مساله برای هر شاخه به صورت نامتقارن و نامشخص است با مکانیزم توزیع بار به صورت dynamic خواهد بود با این روش هر تردی که بیکار باشد یک تسک را به صورت موازی اجرا می کند. از طرف دیگر اگر بخواهیم برای هر زیر مسئله ای آن را به یک ترد assign کنیم زمانی که ابعاد مسئله کوچک می شود (در عمق های پایین تر) شاید بار پردازشی نسبت به سربار حافظه به صرفه نباشد بنابراین عمق را محدود کردیم تا موازی سازی برای بخش های بزرگتر مسئله در لایه های بالاتر انجام شود و بقیه آن به صورت سریال. البته باید نقطه بهینه را با ارزیابی پیدا کنیم.

```
#pragma omp parallel
{
    #pragma omp single
    {
        findAllPaths(0, 0, rows - 1, cols - 1, maze, visited, 0);
    }
}
```

از آنجایی که استراتژی موازی‌سازی در این پروژه بر پایه تخصیص پویا و Assign کردن تسک‌ها به تردهای بیکار است، از الگوی Thread Pool استفاده شده است. بدین معنا که ناحیه موازی تنها یک‌بار در سطح بالا تعریف می‌شود تا تیمی از تردها تشکیل شده و در طول کل زمان اجرا، بدون سربار ساخت و حذف مجدد، مورد استفاده قرار گیرند. در این ساختار، شروع فرآیند و اجرای اولین تسک تنها توسط ترد اصلی در یک ساختار single انجام می‌شود و سایر تردها وظیفه اجرای شاخه‌های بعدی را بر عهده می‌گیرند.

ارزیابی و مقایسه عمق های مختلف: باز آنجایی که موازی سازی این مسئله کاملا وابسته به tradeoff حافظه و بار پردازشی است اگر اندازه مسئله خیلی کوچک باشد به دلیل اینکه نسخه سریال بهینه شده حافظه مجدد از memory نمی گیرد بسیار بهتر از نسخه موازی ای عمل می کند که برای هر ترد یک حافظه جداگانه از memory می گیرد به عبارت دیگر سربار گرفتن حافظه نسبت به بار پردازشی بسیار بیشتر می شود بنابراین سایز مسئله را نسبتا بزرگ مثلا **7*7** در نظر گرفته ایم. از طرف دیگر اگر maxDepth از یک حدی کوچکتر بخش بزرگی از درخت جستجو در لایه‌های پایین‌تر قرار می‌گیرد که طبق شرط توقف، توسط همان ترد سازنده و به صورت **سریال** پردازش می‌شود و موازی سازی ناکارآمد می شود.
* برای ارزیابی دقیق برنامه در حالت release بیلد شده است

maxDepth <= 5:

```
Finding all paths from (0,0) to (6,6)
================================
================================
serial Total paths found: 1295371
serial Time: 0.740176
parallel Total paths found: 1295371
parallel Time: 0.865534
================================
Speedup: 0.855167
```

برای این عمق ها به طور متوسط speedup مثبت نخواهیم داشت

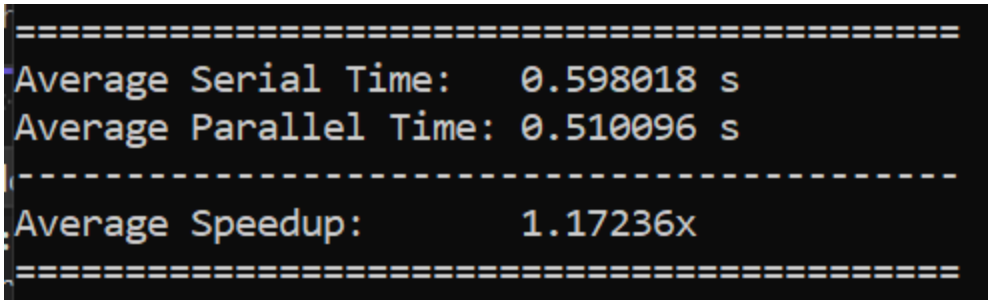
برای عمق های بعدی نتایج به ترتیب به شکل زیر است:

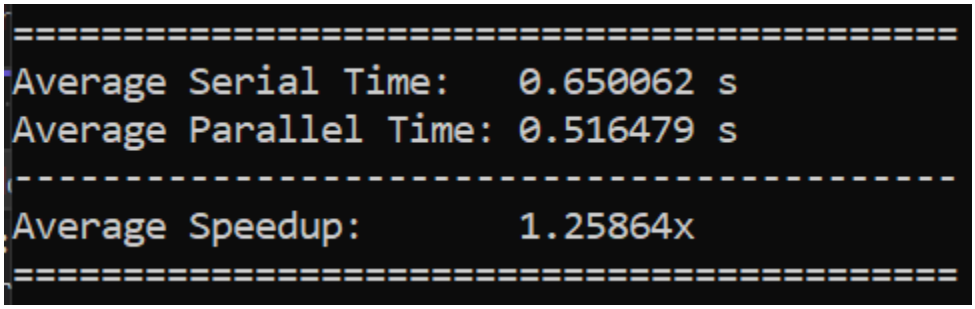
```
==========================================
Average Serial Time:   0.59965 s
Average Parallel Time: 0.623736 s
------------------------------------------
Average Speedup:       0.961384x
==========================================
```

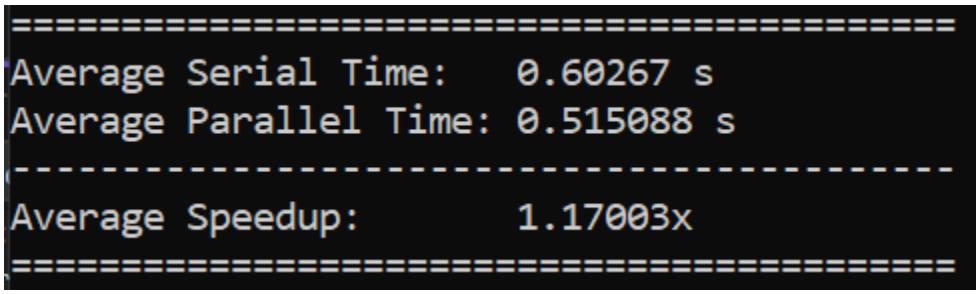
```
==========================================
Average Serial Time:   0.623743 s
Average Parallel Time: 0.601574 s
------------------------------------------
Average Speedup:       1.03685x
==========================================
```

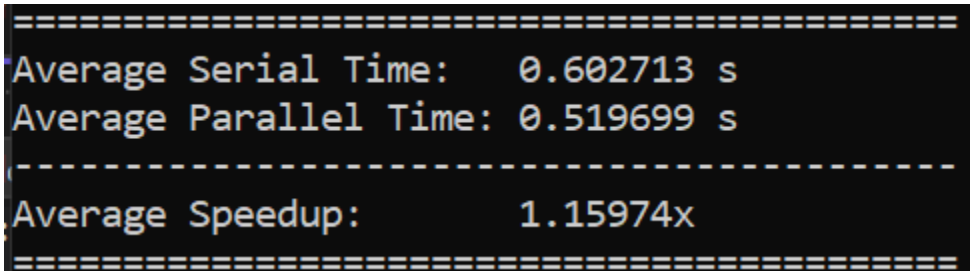
```
==========================================
Average Serial Time:   0.600847 s
Average Parallel Time: 0.566285 s
------------------------------------------
Average Speedup:       1.06103x
==========================================
```

```
==========================================
Average Serial Time:   0.610581 s
Average Parallel Time: 0.52187 s
------------------------------------------
Average Speedup:       1.16999x
==========================================
```





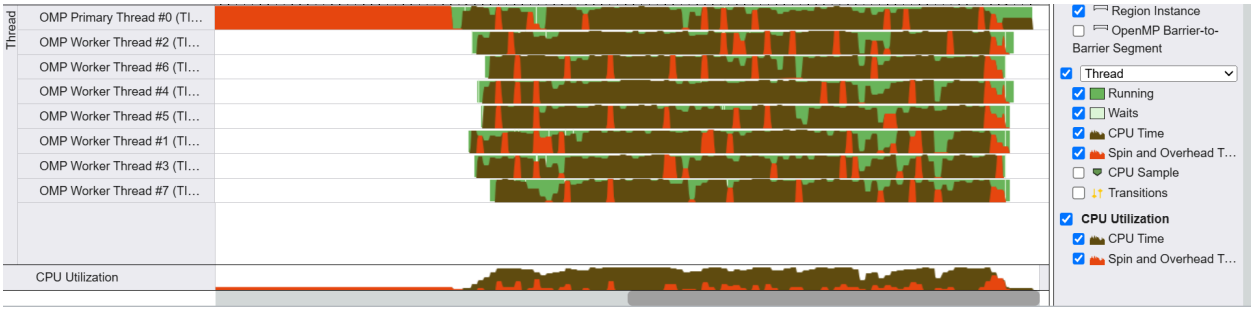




با توجه به نتایج, برای مسئله با سایز **۷*۷** بهترین عمق برابر **۱۱** است که به طور متوسط ۲۶ درصد speedup می دهد

بررسی دقیق نسخه موازی با اندازه 7*7 و maxDepth = 11:

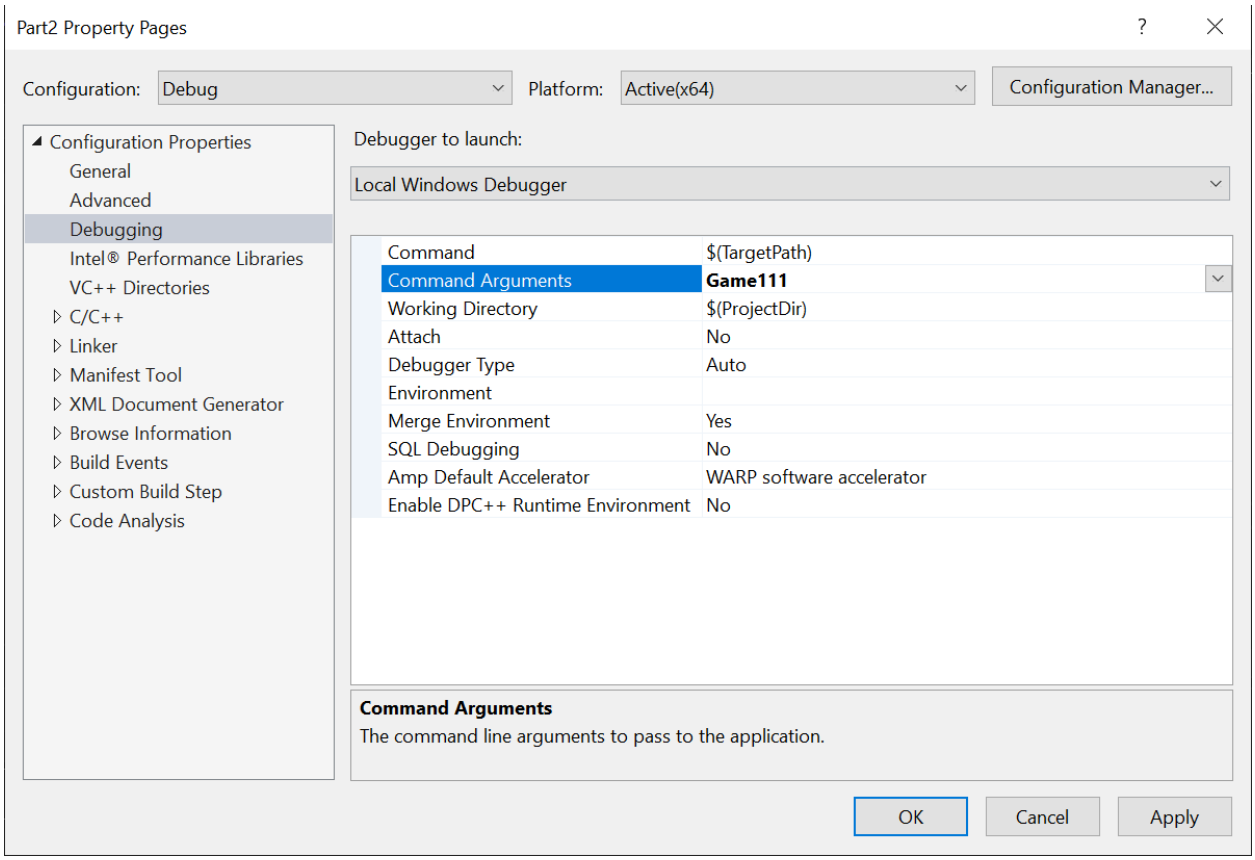
```
================================
serial Total paths found: 1295371
serial Time: 0.81816
parallel Total paths found: 1295371
parallel Time: 0.669579
================================
Speedup: 1.2219
```



بررسی نمودار و مشاهده توزیع یکنواخت نواحی قهوه‌ای رنگ (که نمایانگر زمان فعال پردازنده یا CPU Time است) در تمامی سطرها، نشان می دهد که توزیع بار به شکل مطلوبی میان تردها صورت گرفته و زمان‌بند OpenMP وظایف را به گونه‌ای مدیریت کرده که اکثر تردها به صورت هم‌زمان درگیر پردازش باشند

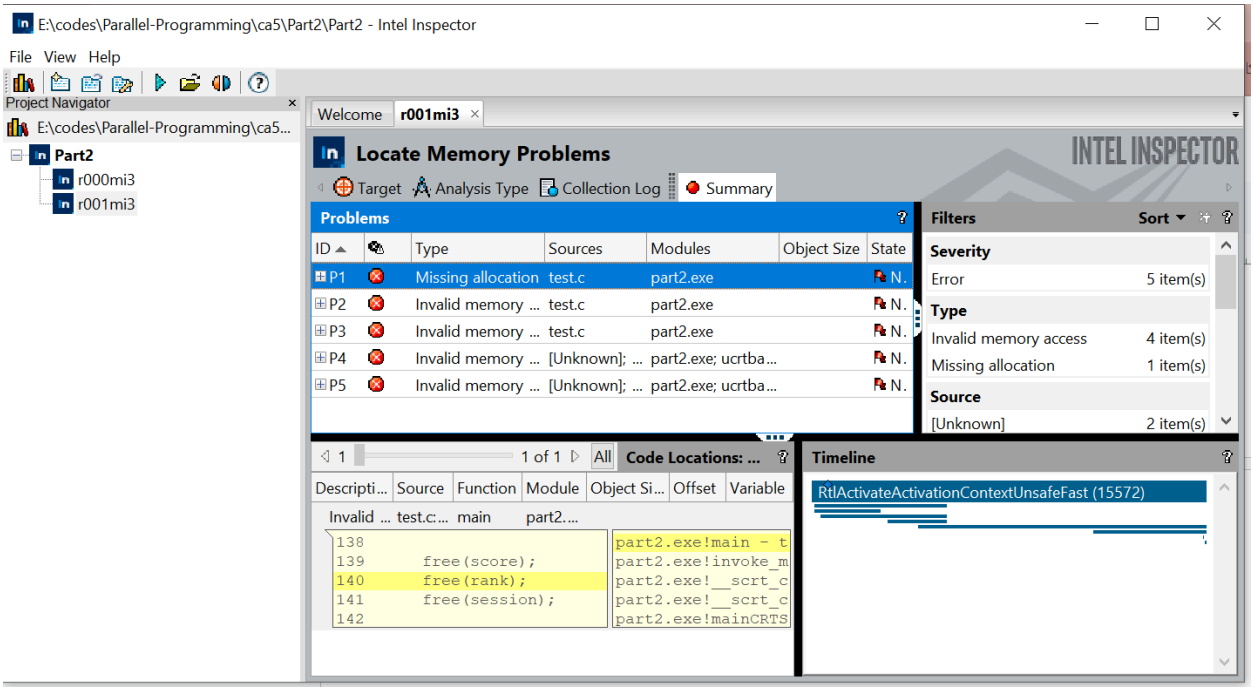
بخش دوم

ابتدا کد بخش دوم را به فایل پروژه اضافه کرده و آن را روی مود دیباگ بیلد می کنیم تا در ادامه با استفاده از ابزار inspector آن را تحلیل کنیم البته چون برنامه از خط فرمان ورودی می گیرد ورودی آن را از مسیر زیر تنظیم می کنیم:



:Memory Error Analysis / Locate Memory Problems

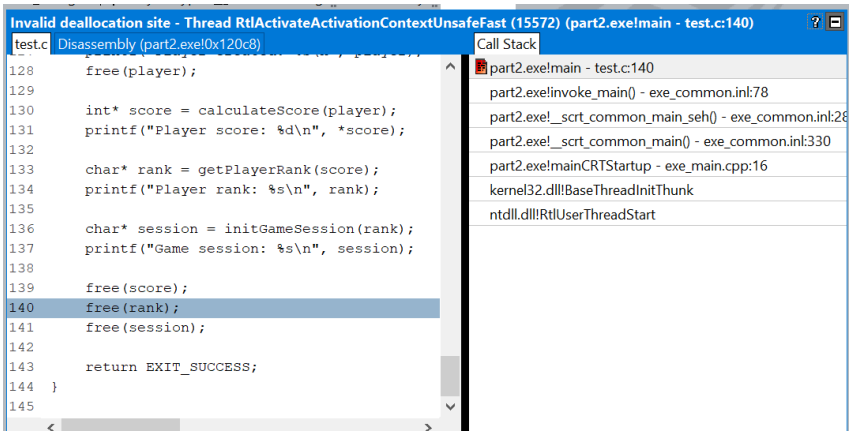
این آنالیز خطاهای سنگین حافظه مثل: Use After Free , Invalid Free / Double Free , Buffer Overflow / Out-of-bounds , Uninitialized memory read/write , Invalid memory access را مشخص می کند که اغلب باعث کرش کردن برنامه می شوند



طبق مشاهدات با دو نوع خطا در این کد مواجه هستیم:

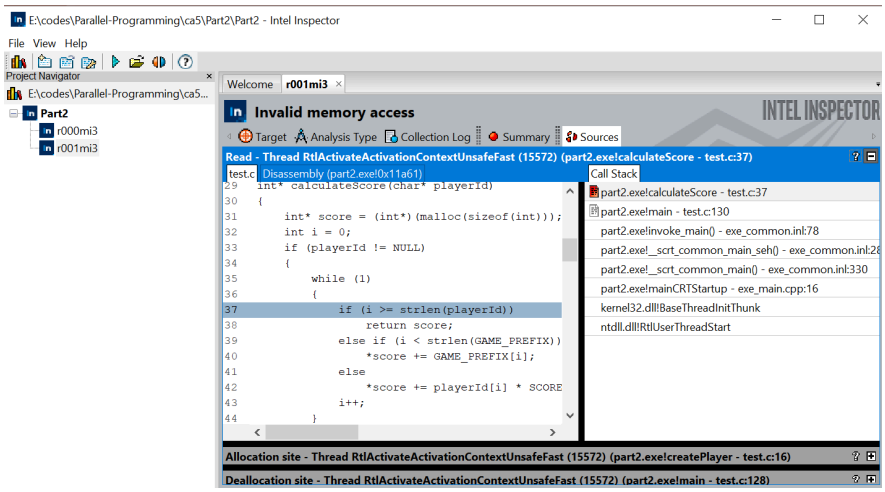
- Missing allocation: تخصیص معتبری برای یک pointer که free شده وجود نداشته
- Invalid memory access: این‌ها معمولاً نتیجه یکی از حالت های use-after-free یا out-of-bounds read/write و یا heap corruption است

در ادامه به بررسی دقیق هر خطا می پردازیم:

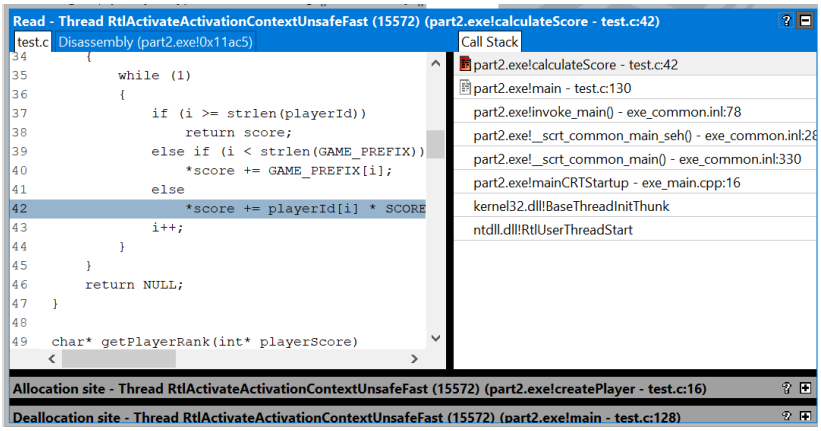


برنامه سعی می‌کند حافظه‌ای را آزاد کند که به‌درستی از Heap تخصیص داده نشده است. مقدار rank از تابع getPlayerRank برمی‌گردد و در برخی حالت‌ها این تابع یک رشته ثابت مثل "Invalid" را برمی‌گرداند. رشته‌های ثابت در

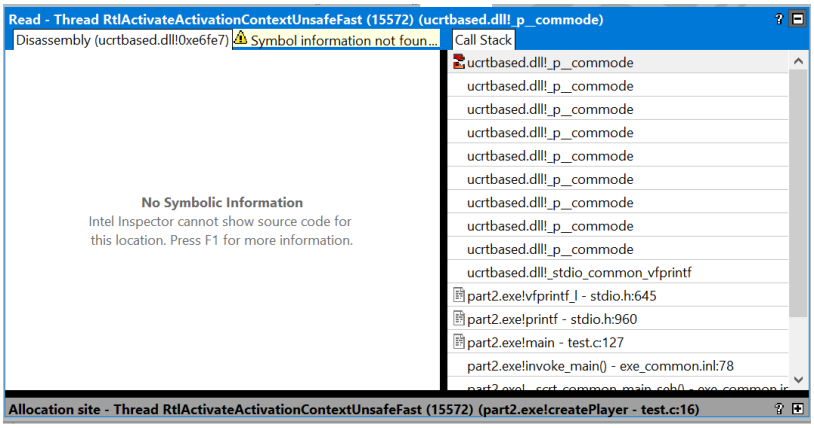
Heap ساخته نمی‌شوند، بنابراین free روی آن‌ها نامعتبر است. برای اصلاح، باید مالکیت حافظه یکدست شود: یا همیشه rank به‌صورت دینامیکی تخصیص داده شود، یا اگر رشته ثابت برگردانده می‌شود، دیگر آن را free نکنیم.



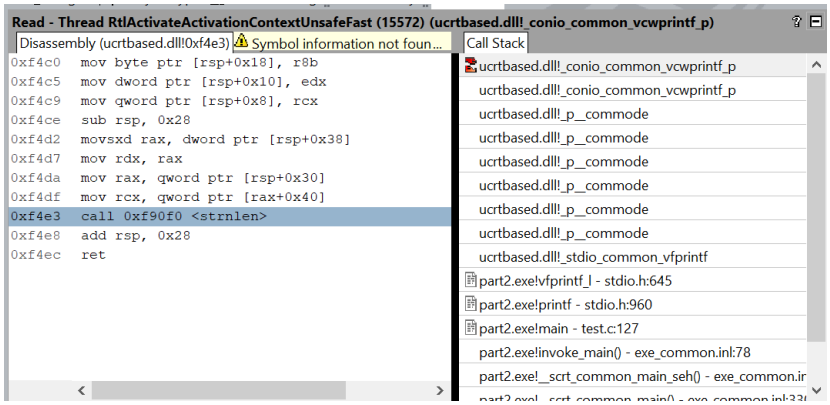
در تابع calculateScore و در خط مربوط به strlen برنامه در حال خواندن داده‌ای از حافظه‌ای است که معتبر نیست. این اتفاق به این دلیل رخ می‌دهد که اشاره‌گر player در تابع main ابتدا آزاد شده و سپس همان اشاره‌گر به calculateScore ارسال شده است. در نتیجه، playerId به حافظه‌ای اشاره می‌کند که دیگر در اختیار برنامه نیست. برای اصلاح، باید ترتیب آزادسازی اصلاح شود؛ یعنی free فقط پس از پایان کامل استفاده از آن انجام شود و تا زمانی که calculateScore به این داده نیاز دارد، اشاره‌گر معتبر باقی بماند.



در خط *score += playerId[i] * SCORE_MULTIPLIER; داخل تابع calculateScore، برنامه در حال خواندن از آرایه playerId است. اما این اشاره‌گر به حافظه‌ای اشاره می‌کند که قبلاً در تابع main آزاد شده است. برای اصلاح، باید آزادسازی player به بعد از پایان کامل استفاده از آن منتقل شود؛ یعنی تا زمانی که calculateScore از این داده استفاده می‌کند، نباید حافظه آن آزاد شود.



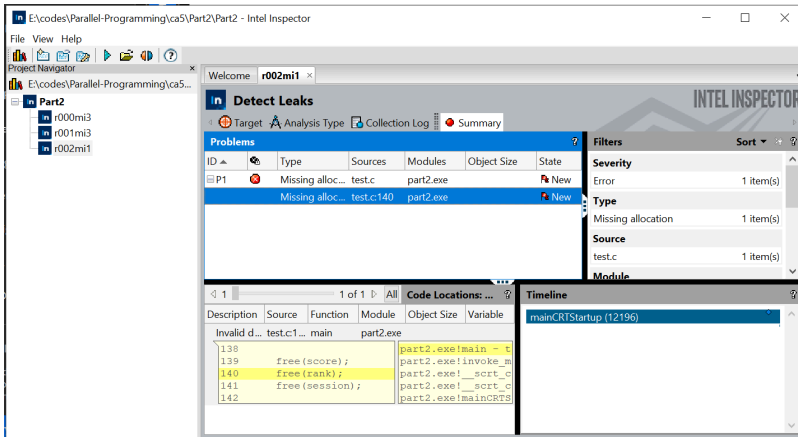
خطا داخل توابع کتابخانه‌ای ویندوز (ucrtbased.dll) دیده می‌شود و Inspector برای آن بخش «symbolic information» ندارد؛ ولی ریشه‌ی واقعی مشکل در در Call Stack مشخص است که مسیر به main در test.c:127 (خط مربوط به printf) می‌رسد. در عمل این یعنی یکی از رشته‌هایی که با s% چاپ می‌شوند یا null-terminate نشده و printf مجبور شده خارج از محدوده دنبال '\0' بگردد، یا اشاره‌گر آن نامعتبر/آزادشده بوده و در زمان چاپ باعث دسترسی غیرمجاز شده است. برای اصلاح، باید مطمئن شویم هر رشته‌ای که چاپ می‌کنیم معتبر و null-terminated است (مثلاً در createPlayer حتماً '\0' در انتها قرار بگیرد و قبل از چاپ هم null بودن اشاره‌گر چک شود). همچنین هر اشاره‌گری که آزاد شده نباید دوباره برای چاپ یا پردازش استفاده شود.

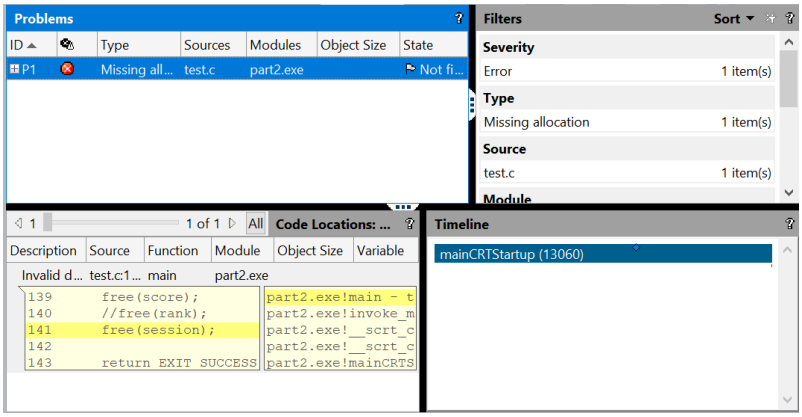


این خطا هم همان سناریوی چاپ رشته ناسالم است، فقط این بار Inspector دقیق‌تر نشان داده که داخل توابع چاپ، فراخوانی strlen انجام شده و همان‌جا به مشکل خورده است.اگر رشته معتبر نباشد یا '\0' نداشته باشد، این فراخوانی به invalid memory access تبدیل می‌شود. در Call Stack هم مشخص است که ریشه به main در test.c:127 می‌رسد.

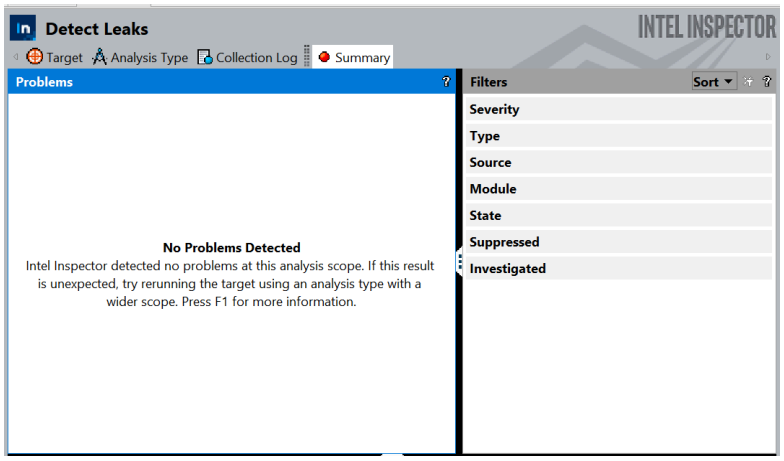
:Memory Error Analysis / Detect Leaks

این آنالیز, نشت حافظه را بررسی می کند در واقع چک می کند که همه حافظه های که بصورت پویا تخصیص داده شده اند, آزاد شده اند یا خیر.



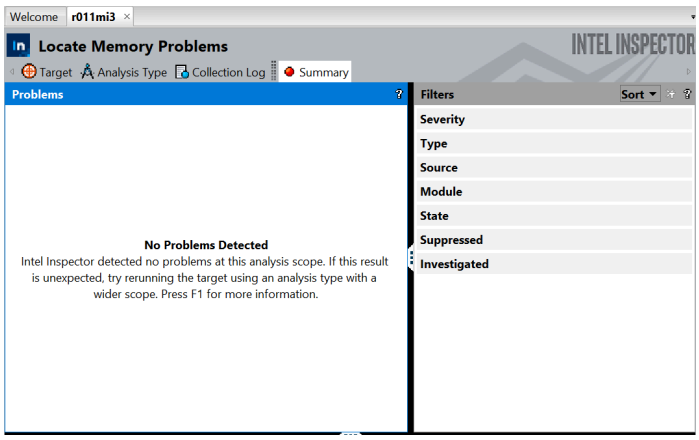


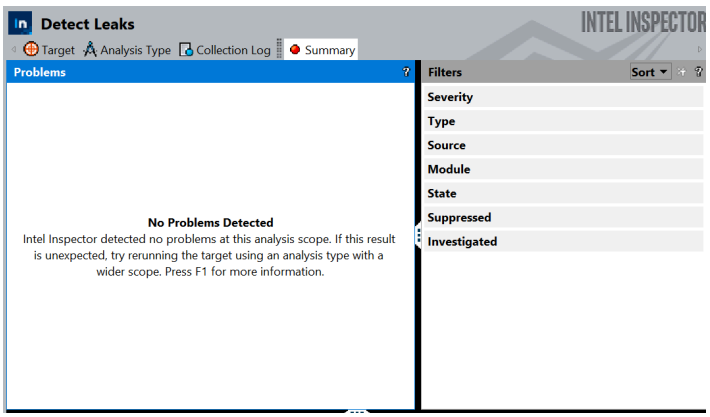
با گرفتن خروجی مشاهده می شود تنها مورد گزارش شده مربوط به آزادسازی پونتری است که قبلا allocate نشده این خطای هیپ باعث شده تا inspector به مرحله بررسی نشت های واقعی نرسد بنابراین این دست خطاها را به صورت موقت کامنت می کنیم تا به نتایج واقعی برسیم.



با کامنت کردن قسمت های مربوط به آزادسازی rank و section که حاوی یک پوینتر نامعتبر بودند مشاهده می شود که آنالیز هیچ مشکل نشت دیگری را تشخیص نمی دهد.

بررسی دوباره پس از اصلاح کد:





بررسی نقاط ضعف و قوت و نمره نهایی کارآموز:

نقاط قوت:

- ساختار مناسب و ماژولار
- نام گذاری مناسب و عدم استفاده از magic numbers

نقاط ضعف:

- همه مشکلات حافظه که در بخش های قبل بررسی شد مانند: use after free , double free, memory leaks, Buffer overflow
- خطاهای منطقی که در کد وجود دارد مانند استفاده از ((srand(time(0 در داخل حلقه که باعث می شود همه حروف تکرار شده یکسان باشد یا عدم مقدار دهی اولیه متغیر score

نمره دهی:

علت کسر نمره	نمره کسب شده	بارم	معیار
وجود خطاهای متعدد حافظه و عدم مدیریت درست حافظه	**۵**	**۴۵**	**مدیریت حافظه**
وجود باگ های منطقی مانند عدم مقدار دهی اولیه score و ...	**۱۰**	**۲۵**	**عملکرد**
برنامه به محض اجرا به دلیل آزادسازی زودرس پوینتر player کرش می‌کند	**۰**	**۱۵**	**پایداری و اجرا**
ماژولار بودن ساختار کد و نام گذاری های مناسب	**۱۵**	**۱۵**	**استایل**
	۳۰	**۱۰۰**	**مجموع**