



٢٠٢٣

BOSS501

Lubna_175170 C1
لوبنا سعيد

Yaman_199273 C1
يامان سعيد

Haneen_197899 C1
هانين سعيد

Dr. Ibaa Oueichek
دكتور إباه عويشك

إدارة الذاكرة:

الطلب الأول: ماهي التعليمات المستخدمة لحجز الذاكرة على المكبس stack، وعلى الكومة heap؟

لحجز الذاكرة على المكبس (Stack) والكومة (Heap)، يتم استخدام تعليمات مختلفة، أحدها:

- حجز الذاكرة على المكبس (Stack):

الذاكرة على المكبس تُستخدم عادة لتخزين المتغيرات المحلية والمؤشرات إلى العناوين في استدعاءات الدوال. تخصيص الذاكرة على المكبس يتم عادة بطريقة تلقائية وسريعة جداً.

المتغيرات المحلية تُخصص تلقائياً عند دخول نطاق الدالة وتُحرر عند الخروج من النطاق.

```
void function() {  
    int localVariable; // على المكبس  
}
```

- حجز الذاكرة على الكومة (Heap):

الذاكرة على الكومة تُستخدم لتخصيص ذاكرة أكبر وأكثر ديناميكية والتي تكون موجودة حتى يتم تحريرها بشكل صريح.

- تخصيص الذكرة على الكومة用 `malloc, calloc, realloc`.

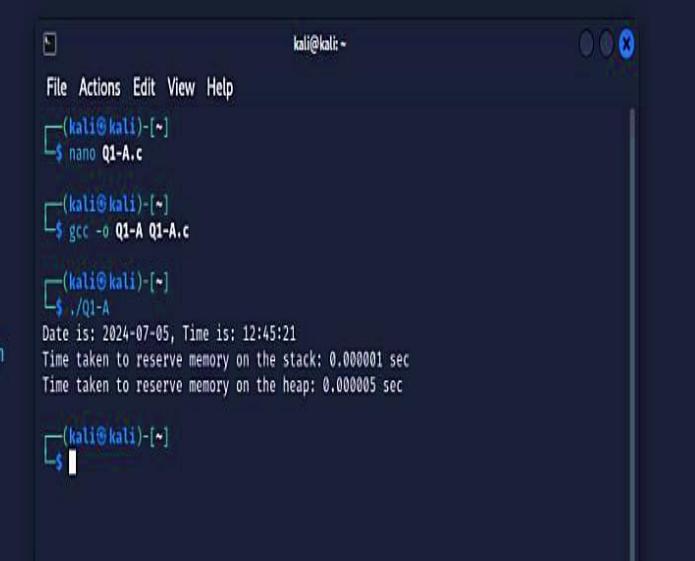
- تحرير الذكرة على الكومة用 `free`.

```
int * ptr = (int *)malloc(sizeof(int) * 10); // تخصيص على الذاكرة الكومة/  
if (ptr != NULL) {  
    // استخدام الذاكرة  
}  
free(ptr); // تحرير الذاكرة
```

يطلب كتابة برنامج بسيط يقوم بحجز كمية كبيرة من الذاكرة بطريقتين مختلفتين: على المكدس stack، وعلى الكومة heap ومقارنة الزمن اللازم لحجز الذاكرة في كلتا الحالتين، ماذا تلاحظ؟



```
File Edit Search View Document Help
File Actions Edit View Help
Date is: 2024-07-05, Time is: 12:45:21
Time taken to reserve memory on the stack: 0.000001 sec
Time taken to reserve memory on the heap: 0.000005 sec
```



```
File Actions Edit View Help
Date is: 2024-07-05, Time is: 12:45:21
Time taken to reserve memory on the stack: 0.000001 sec
Time taken to reserve memory on the heap: 0.000005 sec
```

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<time.h>
4
5 void printDateTime() {
6     time_t current_time;
7     struct tm * time_info;
8     char timeString[12]; // for "HH:MM:SS"
9     char dateString[11]; // for "YYYY-MM-DD"
10
11    time(&current_time);
12    time_info = localtime(&current_time);
13
14    strftime(timeString, sizeof(timeString), "%H:%M:%S", time_info);
15    strftime(dateString, sizeof(dateString), "%Y-%m-%d", time_info);
16
17    printf("Date is: %s, Time is: %s\n", dateString, timeString);
18 }
19
20 double getTime(clock_t start, clock_t end) {
21     return ((double)(end - start)) / CLOCKS_PER_SEC;
22 }
23
24 void stackMemoryTest() {
25     clock_t start, end;
26     start = clock();
27     int Stack[1000000]; // Stack memory allocation
28     end = clock();
29     double stackTime = getTime(start, end);
30     printf("Time taken to reserve memory on the stack: %f sec\n", stackTime);
31 }
32
33 void heapMemoryTest() {
34     clock_t start, end;
35     start = clock();
36     int *Heap = (int *)malloc(1000000 * sizeof(int)); // Heap memory allocation
37     end = clock();
38     double heapTime = getTime(start, end);
39     printf("Time taken to reserve memory on the heap: %f sec\n", heapTime);
40     free(Heap); // Freeing dynamically allocated memory
41 }
42
43 int main() {
44     printDateTime();
45     stackMemoryTest();
46     heapMemoryTest();
47
48     return 0;
49 }
```

الخرج:

```
kali@kali: ~
File Actions Edit View Help
[(kali㉿kali)-~]
$ nano Q1-A.c
[(kali㉿kali)-~]
$ gcc -o Q1-A Q1-A.c
[(kali㉿kali)-~]
$ ./Q1-A
Date is: 2024-07-05, Time is: 12:45:21
Time taken to reserve memory on the stack: 0.000001 sec
Time taken to reserve memory on the heap: 0.000005 sec
[(kali㉿kali)-~]
$
```

شرح الكود: ➔

1. الدالة `:printDateTime()`

- تقوم بالحصول على الوقت الحالي باستخدام `time` و من ثم تحويله إلى هيكل `struct tm` باستخدام `localtime()`.
- تقوم بتنسيق الوقت والتاريخ إلى سلسلة نصية باستخدام `strftime()`.
- تقوم بطباعة التاريخ والوقت المنسقين.

2. الدالة `:getTime()`

- تقوم بحساب الوقت المستغرق بين اثنين من الأوقات المحسوبة بوحدة `clock_t`.
- تستخدم `CLOCKS_PER_SEC` لتحويل الفروق الزمنية إلى ثولي بالأعداد العشرية.

3. الدالة `:stackMemoryTest()`

- تقوم بقياس الوقت الذي يستغرقه تخصيص مساحة ذاكرة على الذاكرة التخزينية "الستاك" `.int Stack[1000000]` باستخدام مصفوفة `(Stack)`.
- تقوم بطباعة الوقت المستغرق.

4. الدالة `:heapMemoryTest()`

- تقوم بقياس الوقت الذي يستغرقه تخصيص مساحة ذاكرة على الذاكرة الحيوية "الهيب" (Heap) باستخدام `malloc()` لتخصيص مساحة ل مليون عنصر من نوع `int`.
- تقوم بطباعة الوقت المستغرق.
- تقوم بحرر الذاكرة المخصصة باستخدام `free()` لتجنب تسريب الذاكرة.

5. دالة `:main()`

- تستدعي الدوال السابقة بالتسلسل.
 - تُطبع توقيت الآن باستخدام `printDateTime()`.
 - تقوم بقياس الوقت اللازم لتخصيص الذاكرة على الستاك باستخدام `stackMemoryTest()`.
 - تقوم بقياس الوقت اللازم لتخصيص الذاكرة على الهيب باستخدام `heapMemoryTest()`.
- الهدف من كتابتنا لهذا الكود هو عرض كيفية استخدام الوقت والتاريخ في C، وكذلك قياس أداء تخصيص الذاكرة على الستاك والهيب. تلك القياسات تساعدي فهم الفروق بين كيفية تخصيص الذاكرة وتأثير ذلك على الأداء العام للبرنامج.

➤ مقارنة الزمن اللازم لحجز الذاكرة في كلتا الحالتين:

عند مقارنة الزمن اللازم لحجز الذاكرة على المكدس وعلى الكومة في هذا الكود، لاحظنا الفروق التالية:

1. زمن التخصيص:

- عملية تخصيص الذاكرة على المكدس (`stackMemoryTest()`) تكون عادة أسرع بكثير مقارنة بتخصيص الذاكرة على الكومة (`heapMemoryTest()`). هذا يعود إلى طبيعة تخصيص الذاكرة على المكدس التي تتم بشكل مباشر وسريع.

2. تأثير على الأداء:

- تخصيص الذاكرة على المكدس يكون عادة أكثر فعالية في الأداء عندما تكون البيانات محلية للدالة أو محدودة النطاق، نظرًا لأن التخصيص والتحرير يتم تلقائيًا مع دورة الحياة للدالة.

- تخصيص الذاكرة على الكومة يمكن أن يكون أكثر مرونة واستخدامًا لأنه يسمح بالتحكم اليدوي بدورة حياة البيانات ويمكن تعين طول الحياة بشكل دقيق.

3. إدارة الذاكرة:

- تخصيص الذاكرة على المكدس يتم بطريقة آلية من قبل النظام، بينما يتطلب تخصيص الذاكرة على الكومة إدارة يدوية أكثر، مثل تخصيصها وتحريرها باستخدام `malloc()` و `free()`.

بالمجمل، اختيار استخدام المكدس أو الكومة يتوقف على طبيعة البيانات واحتياجات التطبيق. إذا كانت البيانات محلية للدالة ومحدودة النطاق، فإن استخدام المكدس يمكن أن يوفر أداءً أفضل، بينما تكون الكومة مفيدة للبيانات ذات الحياة الطويلة أو الحجم الكبير.

.a. البرنامج الأول: حجز كمية بيانات كبيرة على المكبس (وبدون استدعاء عودي).

```
File Edit Search View Document Help
File Actions Edit View Help
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main() {
6     time_t current_time;
7     time(&current_time); // Get the current time
8
9     struct tm *time_info = localtime(&current_time); // Convert time to local time format
10
11    char formatted_time[80];
12    strftime(formatted_time, sizeof(formatted_time), "%Y-%m-%d %H:%M:%S", time_info); // Format the time
13
14    printf("Current time: %s\n", formatted_time);
15
16    size_t large_size = 1000000000;
17    int *large_memory = malloc(large_size); // Dynamically allocate a very large amount of memory
18
19    if (large_memory == NULL) {
20        fprintf(stderr, "Error: Memory allocation failed\n");
21        return 1;
22    }
23
24    printf("First element in allocated memory: %d\n", large_memory[0]);
25
26    free(large_memory); // Free the allocated memory
27
28    return 0;
29 }
```

(kali㉿kali)-[~]\$ date
Sat Jun 29 07:35:34 PM EDT 2024
(kali㉿kali)-[~]\$ nano stack.c
(kali㉿kali)-[~]\$ gcc -o stack stack.c
(kali㉿kali)-[~]\$./stack
Current time: 2024-06-29 19:35:50
Error: Memory allocation failed
(kali㉿kali)-[~]\$

الخرج:

```
File Actions Edit View Help
File Actions Edit View Help
1 (kali㉿kali)-[~]
2 $ date
3 Sat Jun 29 07:35:34 PM EDT 2024
4 (kali㉿kali)-[~]
5 $ nano stack.c
6 (kali㉿kali)-[~]
7 $ gcc -o stack stack.c
8 (kali㉿kali)-[~]
9 $ ./stack
10 Current time: 2024-06-29 19:35:50
11 Error: Memory allocation failed
12 (kali㉿kali)-[~]
```

الكود يوضح كيفية الحصول على الوقت الحالي، تنسيقه، وإدارة الذاكرة الديناميكية. يتكون الكود من عدة أجزاء رئيسية:

1. الحصول على الوقت الحالي وتنسيقه:

- يستخدم الكود دوال من مكتبة time.h للحصول على الوقت والتاريخ الحاليين.
- يتم تحويل الوقت إلى الشكل المحلي باستخدام دالة localtime .
- يتم تنسيق الوقت كسلسلة نصية باستخدام دالة strftime بصيغة "سنة - شهر - يوم ساعة : دقيقة : ثانية".

هذه العملية تسهل عرض الوقت بشكل مفهوم للمستخدم وتحل إمكانية تعديل الصيغة بسهولة.

2. تخصيص ذاكرة ديناميكياً:

- يحاول الكود تخصيص كمية كبيرة من الذاكرة (10 جيجابايت) باستخدام دالة malloc من مكتبة stdlib.h
- هذه العملية توضح كيفية التعامل مع الذاكرة الديناميكية والتحقق من نجاح عملية التخصيص باستخدام الشرط if.
- في حال فشل عملية التخصيص، يتم طباعة رسالة خطأ ويتم إيقاف البرنامج باستخدام return 1.

3. التعامل مع الذاكرة المخصصة:

- في حال نجاح التخصيص، يتم الوصول لأول عنصر في الذاكرة المخصصة وعرض قيمته (والتي قد تكون غير مهيأة).
- هذه الخطوة توضح كيفية الوصول إلى الذاكرة المخصصة، مع ملاحظة أن القيم غير المهيأة قد تحتوي على بيانات غير معروفة (garbage values).

4. تحرير الذاكرة:

- بعد الانتهاء من استخدام الذاكرة، يتم تحريرها باستخدام دالة free لتجنب تسربات الذاكرة.
- تحرير الذاكرة هو ممارسة مهمة في إدارة الذاكرة لتجنب مشاكل الأداء والتسربات التي قد تؤدي إلى استنزاف موارد النظام.

.b. البرنامج الثاني: كتابة إجراء عودي لا منتهي.

The screenshot shows a terminal window with two panes. The left pane displays the source code of a C program named `stackoverflew.c`. The right pane shows the terminal session where the program is compiled and run, resulting in a segmentation fault.

```
File Edit Search View Document Help
File Actions Edit View Help
(kali㉿kali)-[~]
$ date
Sat Jun 29 07:37:51 PM EDT 2024
(kali㉿kali)-[~]
$ nano stackoverflew.c
(kali㉿kali)-[~]
$ gcc -o stackoverflow stackoverflow.c
(kali㉿kali)-[~]
$ ./stackoverflow
Current time: 2024-06-29 19:38:43
zsh: segmentation fault  ./stackoverflow
(kali㉿kali)-[~]
$
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 // Function with infinite recursion
6 int infinite_recursion(int n) {
7     // Recursive call without termination condition
8     return n * infinite_recursion(n - 1);
9 }
10
11 int main() {
12     // Get the current time
13     time_t current_time;
14     time(&current_time);
15
16     // Convert time to local time format
17     struct tm *time_info = localtime(&current_time);
18
19     // Format the time
20     char time_str[80];
21     strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S", time_info);
22
23     // Print the formatted time
24     printf("Current time: %s\n", time_str);
25
26     // Start the infinite recursion
27     int result = infinite_recursion(100);
28
29     return 0;
30 }
31
```

الخرج:

The screenshot shows a terminal window with a single pane displaying the same terminal session as the previous image, starting with the date command and ending with the segmentation fault error message.

```
File Actions Edit View Help
(kali㉿kali)-[~]
$ date
Sat Jun 29 07:37:51 PM EDT 2024
(kali㉿kali)-[~]
$ nano stackoverflew.c
(kali㉿kali)-[~]
$ gcc -o stackoverflow stackoverflow.c
(kali㉿kali)-[~]
$ ./stackoverflow
Current time: 2024-06-29 19:38:43
zsh: segmentation fault  ./stackoverflow
(kali㉿kali)-[~]
$
```

الكود يوضح كيفية الحصول على الوقت الحالي، تنسيقه، وتنفيذ استدعاء عودي غير منتهي. يتكون الكود من عدة أجزاء رئيسية:

1. الحصول على الوقت الحالي وتنسيقه:

- يستخدم الكود دوال من مكتبة time.h للحصول على الوقت والتاريخ الحاليين.
- يتم تحويل الوقت إلى الشكل المحلي باستخدام دالة localtime .
- يتم تنسيق الوقت كسلسلة نصية باستخدام دالة strftime بصيغة "سنة - شهر - يوم ساعة : دقيقة : ثانية".

هذه العملية تسهل عرض الوقت بشكل مفهوم للمستخدم وتتيح إمكانية تعديل الصيغة بسهولة.

2. دالة الاستدعاء العودي :infinite_recursion

- هذه الدالة تستدعي نفسها بشكل عودي بدون وجود شرط إنها، مما يؤدي إلى استدعاءات لا نهائية.
- يقوم الكود باستدعاء هذه الدالة بالقيمة 100، مما يتسبب في تجاوز سعة المكدس (stack overflow) لأن الاستدعاء العودي يستمر بلا توقف.

3. بدء الاستدعاء العودي:

يتم استدعاء دالة infinite_recursion بالقيمة 100، مما يؤدي إلى سلسلة استدعاءات لا نهائية وينتج عنه خطأ تجاوز سعة المكدس.

يطلب تحديد الإشارة signal التي يرسلها نظام التشغيل إلى الإجرائية عند وقوع هذا الخطأ، وهل يمكن التقاطها من قبل الإجرائية. هنا هناك فارق ما في حال تم استخدام إجرائية فيها عدة نياسب وقام كل منها بتوليد الخطأ؟

١. الإشارة التي يرسلها نظام التشغيل عند وقوع الخطأ:

عند حدوث استدعاء تكراري غير محدود يؤدي إلى نفاد مساحة المكدس (Stack Overflow)، يقوم نظام التشغيل بإرسال إشارة SIGSEGV (Segmentation Fault) إلى البرنامج.

٢. هل يمكن التقاط هذه الإشارة من قبل الإجرائية؟

في الكود الحالي، لا يمكن التقاط الإشارة SIGSEGV لأنه لم يتم تضمين أي تعليمات لمعالجة الإشارات. بدون إضافة معالج للإشارات، سيقوم البرنامج بإنها الفوري عند تلقى SIGSEGV.

٣. الفارق عند استخدام إجرائية فيها عدة خيوط وقام كل منها بتوليد الخطأ:

إذا كانت الإجرائية تحتوي على عدة خيوط وكل خيط يتسبب في خطأ يؤدي إلى نفاد المكدس، فإن كل خيط سيستقبل إشارة SIGSEGV بشكل مستقل. في هذه الحالة، نظام التشغيل سيقوم بإرسال إشارة SIGSEGV إلى كل خيط يتسبب في الخطأ، مما يؤدي إلى سلوك غير متوقع وربما إنها البرنامج. إذا لم يكن هناك معالج للإشارات، لن تكون هناك طريقة للتحكم في هذا السلوك أو التعامل مع الخطأ بطريقة مخصصة.

النیاں:

The screenshot shows a terminal window on a Kali Linux system. The terminal displays the execution of a C program that demonstrates thread synchronization using mutexes and barriers. The program defines a shared variable `global_counter`, initializes it to 0, and increments it in parallel by 16 threads. Each thread waits at a barrier before incrementing the counter and then exits. The output shows that each of the 16 threads successfully incremented the counter once, resulting in a final value of 16.

```
File Edit Search View Document Help
File Actions Edit View Help
(kali㉿kali)-[~]
$ date
Sat Jun 29 03:11:10 PM EDT 2024
(kali㉿kali)-[~]
$ nano thread.c
(kali㉿kali)-[~]
$ gcc -pthread thread.c -o thread
(kali㉿kali)-[~]
$ ./thread
Number of available CPU cores: 4
Thread 0 incremented global_counter to 1
Thread 2 incremented global_counter to 2
Thread 1 incremented global_counter to 3
Thread 10 incremented global_counter to 4
Thread 3 incremented global_counter to 5
Thread 12 incremented global_counter to 6
Thread 15 incremented global_counter to 7
Thread 13 incremented global_counter to 8
Thread 7 incremented global_counter to 9
Thread 5 incremented global_counter to 10
Thread 9 incremented global_counter to 11
Thread 4 incremented global_counter to 12
Thread 8 incremented global_counter to 13
Thread 11 incremented global_counter to 14
Thread 6 incremented global_counter to 15
Thread 14 incremented global_counter to 16
All threads completed. Final value of global_counter: 16
Time taken: 0.001745 seconds
(kali㉿kali)-[~]
$
```

```
File Edit Search View Document Help
File Actions Edit View Help
(kali㉿kali)-[~]
$ date
Sat Jun 29 03:11:10 PM EDT 2024
(kali㉿kali)-[~]
$ nano thread.c
(kali㉿kali)-[~]
$ gcc -pthread thread.c -o thread
(kali㉿kali)-[~]
$ ./thread
Number of available CPU cores: 4
Thread 0 incremented global_counter to 1
Thread 2 incremented global_counter to 2
Thread 1 incremented global_counter to 3
Thread 10 incremented global_counter to 4
Thread 3 incremented global_counter to 5
Thread 12 incremented global_counter to 6
Thread 15 incremented global_counter to 7
Thread 13 incremented global_counter to 8
Thread 7 incremented global_counter to 9
Thread 5 incremented global_counter to 10
Thread 9 incremented global_counter to 11
Thread 4 incremented global_counter to 12
Thread 8 incremented global_counter to 13
Thread 11 incremented global_counter to 14
Thread 6 incremented global_counter to 15
Thread 14 incremented global_counter to 16
All threads completed. Final value of global_counter: 16
Time taken: 0.001745 seconds
(kali㉿kali)-[~]
```

```

73 // Get the number of available CPU cores
74 int num_cores = get_cpu_cores();
75 printf("Number of available CPU cores: %d\n", num_cores);
76
77 // Start measuring time
78 start_time = clock();
79
80 // Initialize mutex and barrier
81 pthread_mutex_init(&counter_lock, NULL);
82 pthread_barrier_init(&barrier, NULL, NUM_THREADS);
83
84 // Create threads
85 create_threads(threads, thread_ids);
86
87 // Wait for all threads to complete
88 join_threads(threads);
89
90 // Destroy mutex and barrier
91 pthread_mutex_destroy(&counter_lock);
92 pthread_barrier_destroy(&barrier);
93
94 // End measuring time
95 end_time = clock();
96 elapsed_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
97
98 // Display final results
99 printf("All threads completed. Final value of global_counter: %d\n", global_counter);
100 printf("Time taken: %f seconds\n", elapsed_time);
101
102 return 0;
103 }
104

```

```

File Actions Edit View Help
(kali㉿kali)-[~]
$ date
Sat Jun 29 03:11:10 PM EDT 2024
(kali㉿kali)-[~]
$ nano thread.c
(kali㉿kali)-[~]
$ gcc -pthread thread.c -o thread
(kali㉿kali)-[~]
$ ./thread
Number of available CPU cores: 4
Thread 0 incremented global_counter to 1
Thread 2 incremented global_counter to 2
Thread 1 incremented global_counter to 3
Thread 10 incremented global_counter to 4
Thread 3 incremented global_counter to 5
Thread 12 incremented global_counter to 6
Thread 15 incremented global_counter to 7
Thread 13 incremented global_counter to 8
Thread 7 incremented global_counter to 9
Thread 5 incremented global_counter to 10
Thread 9 incremented global_counter to 11
Thread 4 incremented global_counter to 12
Thread 8 incremented global_counter to 13
Thread 11 incremented global_counter to 14
Thread 6 incremented global_counter to 15
Thread 14 incremented global_counter to 16
All threads completed. Final value of global_counter: 16
Time taken: 0.001745 seconds

```

خرج الاختبار على نواة واحدة:

```

File Actions Edit View Help
(kali㉿kali)-[~]
$ date
Sat Jun 29 03:16:25 PM EDT 2024
(kali㉿kali)-[~]
$ nano thread.c
(kali㉿kali)-[~]
$ gcc -pthread thread.c -o thread
(kali㉿kali)-[~]
$ taskset -c 0 ./thread

Number of available CPU cores: 4
Thread 13 incremented global_counter to 1
Thread 0 incremented global_counter to 2
Thread 3 incremented global_counter to 3
Thread 4 incremented global_counter to 4
Thread 8 incremented global_counter to 5
Thread 7 incremented global_counter to 6
Thread 2 incremented global_counter to 7
Thread 14 incremented global_counter to 8
Thread 6 incremented global_counter to 9
Thread 5 incremented global_counter to 10
Thread 12 incremented global_counter to 11
Thread 10 incremented global_counter to 12
Thread 11 incremented global_counter to 13
Thread 15 incremented global_counter to 14
Thread 9 incremented global_counter to 15
Thread 1 incremented global_counter to 16
All threads completed. Final value of global_counter: 16
Time taken: 0.000622 seconds


```

The screenshot shows a terminal window with a dark blue background and white text. At the top, there's a menu bar with 'File', 'Actions', 'Edit', 'View', and 'Help'. Below the menu, the terminal prompt is '(kali㉿kali)-[~]'. The user runs several commands:

- \$ date
- Sat Jun 29 03:17:59 PM EDT 2024
- \$ nano thread.c
- \$ gcc -pthread thread.c -o thread
- \$ taskset -c 0,1 ./thread

After executing the last command, the terminal displays the output of the program, which increments a global counter from 1 to 16 across four threads (0, 1, 2, 3). The output is as follows:

```
Number of available CPU cores: 4
Thread 1 incremented global_counter to 1
Thread 2 incremented global_counter to 2
Thread 9 incremented global_counter to 3
Thread 13 incremented global_counter to 4
Thread 7 incremented global_counter to 5
Thread 0 incremented global_counter to 6
Thread 15 incremented global_counter to 7
Thread 14 incremented global_counter to 8
Thread 4 incremented global_counter to 9
Thread 5 incremented global_counter to 10
Thread 3 incremented global_counter to 11
Thread 10 incremented global_counter to 12
Thread 12 incremented global_counter to 13
Thread 8 incremented global_counter to 14
Thread 6 incremented global_counter to 15
Thread 11 incremented global_counter to 16
All threads completed. Final value of global_counter: 16
Time taken: 0.001652 seconds
```

Finally, the user exits the terminal with:

```
$
```

```
File Actions Edit View Help
[(kali㉿kali)-[~]]
$ date
Sat Jun 29 03:11:10 PM EDT 2024
[(kali㉿kali)-[~]]
$ nano thread.c
[(kali㉿kali)-[~]]
$ gcc -pthread thread.c -o thread
[(kali㉿kali)-[~]]
$ ./thread
Number of available CPU cores: 4
Thread 0 incremented global_counter to 1
Thread 2 incremented global_counter to 2
Thread 1 incremented global_counter to 3
Thread 10 incremented global_counter to 4
Thread 3 incremented global_counter to 5
Thread 12 incremented global_counter to 6
Thread 15 incremented global_counter to 7
Thread 13 incremented global_counter to 8
Thread 7 incremented global_counter to 9
Thread 5 incremented global_counter to 10
Thread 9 incremented global_counter to 11
Thread 4 incremented global_counter to 12
Thread 8 incremented global_counter to 13
Thread 11 incremented global_counter to 14
Thread 6 incremented global_counter to 15
Thread 14 incremented global_counter to 16
All threads completed. Final value of global_counter: 16
Time taken: 0.001745 seconds

[(kali㉿kali)-[~]]
$
```

- هذا البرنامج يهدف إلى إنشاء 16 خيطا (threads) تعمل معًا لزيادة قيمة متغير مشترك يسمى `global_counter`.
- في البداية، يقوم البرنامج بتعريف عدد الخيوط المطلوبة وتحديد بعض المتغيرات المشتركة مثل العداد العالمي (`global_counter`) والقفل (`counter_lock`) وال حاجز (`barrier`). القفل يستخدم لضمان أن كل خيط يقوم بزيادة العداد العالمي بشكل متسلسل وأمن، بينما الحاجز يستخدم لضمان أن جميع الخيوط تنتظر بعضها البعض قبل البدء في العمل.
- الدالة التي تنفذها الخيوط هي `increment_global_counter`. في هذه الدالة، ينتظر كل خيط عند الحاجز أولاً، ثم بعد أن تصل جميع الخيوط إلى الحاجز، يبدأ كل خيط بزيادة قيمة `global_counter` بشكل متسلسل باستخدام القفل.
- قبل إنشاء الخيوط، يقوم البرنامج بالحصول على عدد النوى المتاحة في النظام لعرضها للمستخدم. ثم يبدأ قياس الزمن باستخدام دالة `clock`. بعد ذلك، يتم تهيئة القفل وال حاجز، ثم يتم إنشاء الخيوط وتشغيلها. بعد انتهاء الخيوط من عملها، يتم تدمير القفل وال حاجز ويتم حساب الزمن المستغرق وعرض النتائج النهائية.
- هل يتأثر زمن التنفيذ عند التحكم بالنوى المستخدمة؟
يعتمد ذلك على عدد أنوبي المعالج المتاحة وكيفية توزيع الخيوط عليها. يمكن استخدام تعليمات `taskset` لتحديد النوى التي يمكن تشغيل البرنامج عليها، مما قد يؤثر على الأداء بناءً على التوافر والتحميل على تلك النوى.
- عند قياس زمن التنفيذ:
 - نواة واحدة: زمن التنفيذ 0.000622
 - نوatiين: زمن التنفيذ 0.001652
 - جميع النوى (4 أنوبي): زمن التنفيذ 0.001745
- نلاحظ أنه عند قياس زمن التنفيذ على نواة واحدة كان الزمن أصغر من زمن التنفيذ على جميع الأنوبية لأن التنافس لم يكن فعلياً على الخيوط ذاتها وإنما على الموارد وبالتالي كل زيادة في الأنوبية تؤدي إلى زيادة في الخطوط ومنه زيادة في التنافس والانتظار وبالتالي سيكون زمن التنفيذ أطول حيث يحتاج إلى وقت أكثر كما حدث عند زيادة عدد الأنوبية من نواة إلى 4 أنوبية.
أما بالنسبة إلى زمن التنفيذ على نوatiين سيكون أكثر مقارنة بنواة واحدة وأصغر مقارنة بجميع الأنوبية.

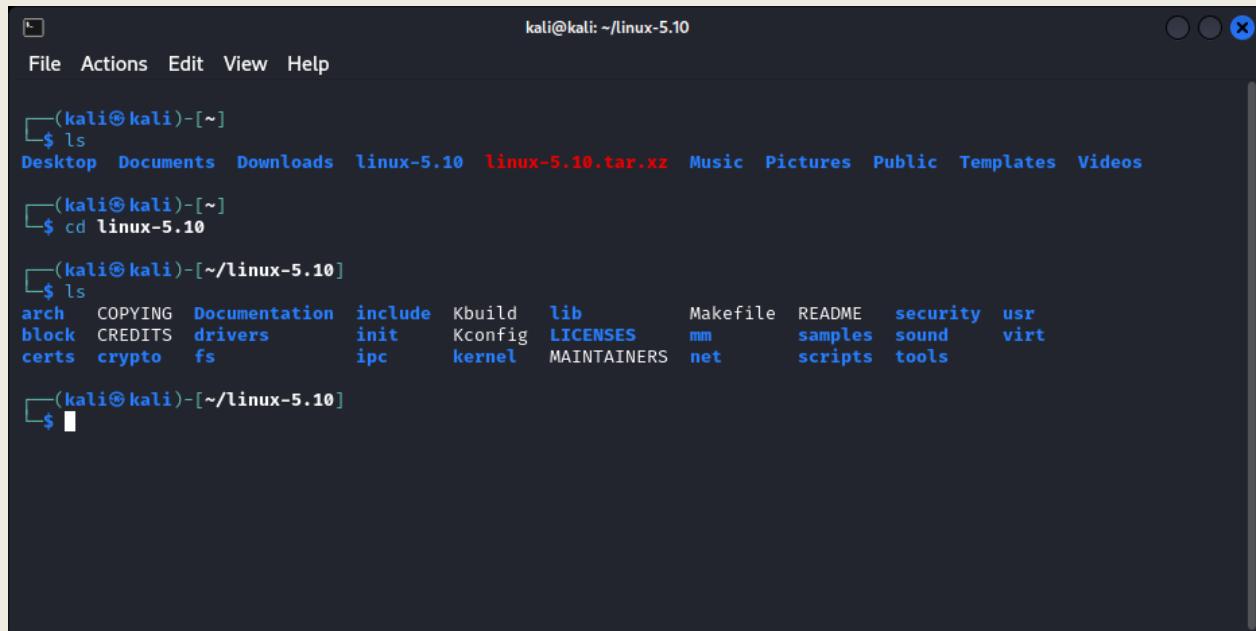
نواة نظام التشغيل:

.a. يطلب البحث ضمن ملفات المصدر الخاص بنواة لينوكس عن محتوى كتلة التحكم بالإجرائية

وتحديد البيانات الخاصة بسياق الإجرائية وتلك الخاصة بسياق النيساب.

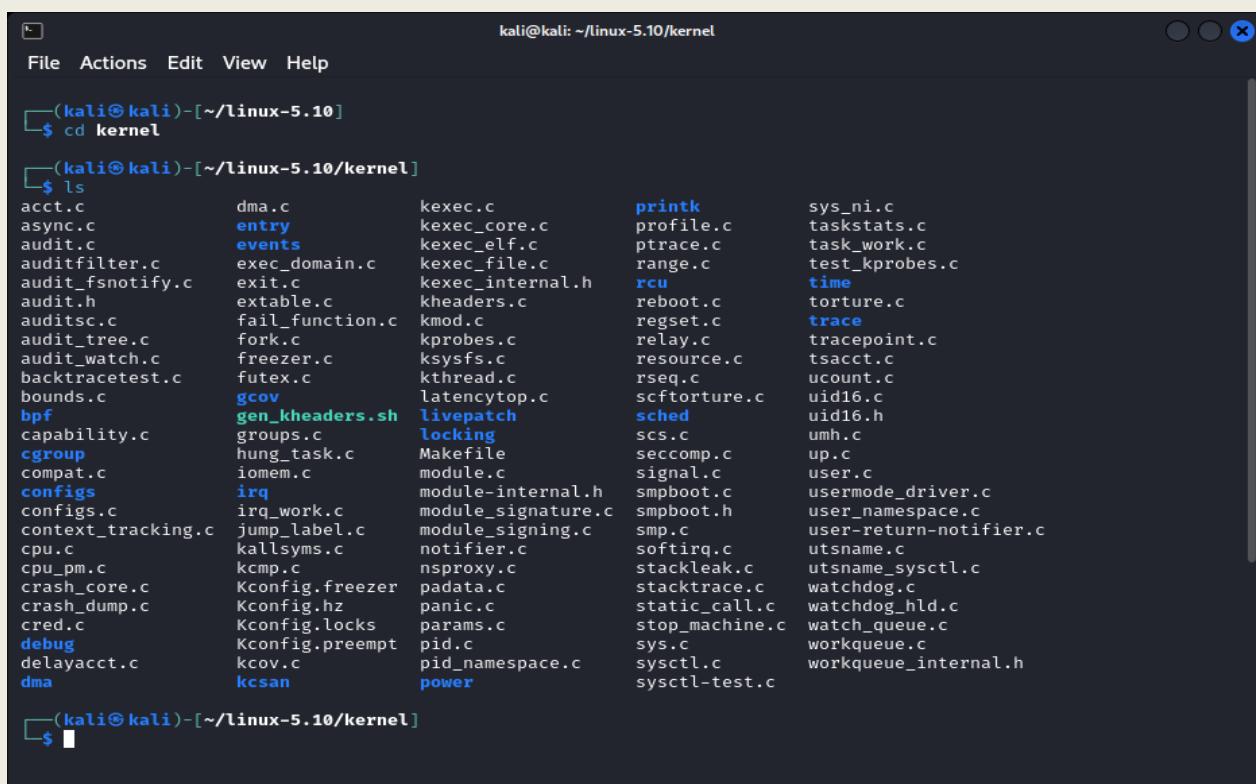
تم بداية تنزيل ملف نواة لينكس مصدرى من موقع kernel.org وفك الضغط عنه .

صورة عما يوجد بداخله :



```
kali㉿kali: ~/linux-5.10
File Actions Edit View Help
[(kali㉿kali)-[~]
$ ls
Desktop Documents Downloads linux-5.10 linux-5.10.tar.xz Music Pictures Public Templates Videos
[(kali㉿kali)-[~]
$ cd linux-5.10
[(kali㉿kali)-[~/linux-5.10]
$ ls
arch COPYING Documentation include Kbuild lib Makefile README security usr
block CREDITS drivers init Kconfig LICENSES mm samples sound virt
certs crypto fs ipc kernel MAINTAINERS net scripts tools
[(kali㉿kali)-[~/linux-5.10]
$ 
```

قمنا بالذهاب لـ مجلد **kernel** الخاص بكل ما يتعلق بالنواة ووجدنا بداخله ما يلي :



```
kali㉿kali: ~/linux-5.10/kernel
File Actions Edit View Help
[(kali㉿kali)-[~/linux-5.10]
$ cd kernel
[(kali㉿kali)-[~/linux-5.10/kernel]
$ ls
acct.c dma.c kexec.c printk sys_ni.c
async.c entry kexec_core.c profile.c taskstats.c
audit.c events kexec_elf.c ptrace.c task_work.c
auditfilter.c exec_domain.c kexec_file.c range.c test_kprobes.c
audit_fsnotify.c exit.c kexec_internal.h rcu time
audit.h extable.c kheaders.c reboot.c torture.c
auditsc.c fail_function.c kmod.c regset.c trace
audit_tree.c fork.c kprobes.c relay.c tracepoint.c
audit_watch.c freezer.c ksfs.c resource.c tsacct.c
backtracetest.c futex.c kthread.c rseq.c ucount.c
bounds.c gcov latencytop.c scftorture.c uid16.c
bpf gen_kheaders.sh livepatch sched uid16.h
capability.c groups.c locking scks.c umh.c
cgroup hung_task.c Makefile seccomp.c up.c
compat.c iomem.c module.c signal.c user.c
configs irq module-internal.h smpboot.c usermode_driver.c
configs.c irq_work.c module_signature.c smpboot.h user_namespace.c
context_tracking.c jump_label.c module_signing.c smp.c user-return-notifier.c
cpu.c kallsyms.c notifier.c softirq.c utsname.c
cpu_pm.c kcmp.c nsproxy.c stackleak.c utsname_sysctl.c
crash_core.c Kconfig.freezer padata.c stacktrace.c watchdog.c
crash_dump.c Kconfig.hz panic.c static_call.c watchdog_hld.c
cred.c Kconfig.locks params.c stop_machine.c watch_queue.c
debug Kconfig.preempt pid.c sys.c workqueue.c
delayacct.c kcov.c pid_namespace.c sysctl.c workqueue_internal.h
dma kcsan power sysctl-test.c
[(kali㉿kali)-[~/linux-5.10/kernel]
$ 
```

تم الدخول إليه والبحث بداخله عن البيانات الخاصة بسياق الإجرائية وسياق النياسب.

. توصلنا إلى المجلد `sched`

```
kali@kali: ~/linux-5.10/kernel/sched
File Actions Edit View Help
(kali㉿kali)-[~/linux-5.10/kernel]
$ cd sched
(kali㉿kali)-[~/linux-5.10/kernel/sched]
$ ls
autogroup.c    cpacct.c      cpupri.c     fair.c      Makefile     rt.c       stats.h    wait.c
autogroup.h    cpudeadline.c  cpupri.h     features.h  membarrier.c sched.h    stop_task.c
clock.c        cpudeadline.h  cputime.c   idle.c      pelt.c      sched-pelt.h swait.c
completion.c   cpufreq.c     deadline.c  isolation.c pelt.h      smp.h      topology.c
core.c         cpufreq_schedutil.c debug.c    loadavg.c  psic.c      stats.c    wait_bit.c
(kali㉿kali)-[~/linux-5.10/kernel/sched]
$
```

بالبحث والتقصي داخل `Documentation` على موقع kernel.org وجدنا أن الملف `sched.h` هو

الملف الرئيسي المسؤول عن إدارة الإجرائيات والنياسب الخاصة بالنظام.

يحتوي هذا الملف على كتلة التحكم بالإجرائية - PCB والتي تمثل بنية `Task_struct` وهي تمثل

الهيكل الأساسي الذي يمثل عملية (process) أو خيط (thread) في النظام.

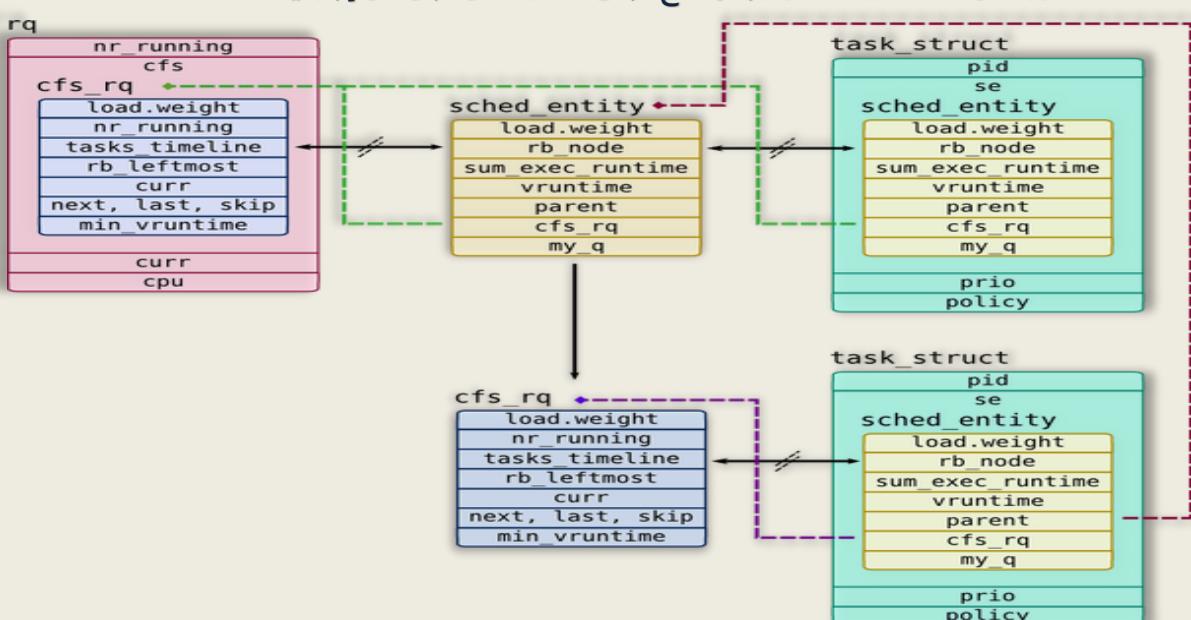
يحتوي أيضاً على بنية `sched_entity` والتي تمثل كيان الجدولة ويستخدم لتجريد عملية الجدولة

بحيث يمكن تطبيقها على الخيوط أو العمليات.

يحتوي أيضاً على رتل `rq` المسمى "runqueue" وهو هيكل بيانات يمثل قائمة الانتظار للعمليات

القابلة للتنفيذ على معالج معين.

صورة عن `Task_Struct` وترابطها مع البنى الأخرى من أجل كل إجرائية:



● من هذه الحقول ما يلي:

1 - **الحالة State**: وهي تمثل حالة الإجرائية نائمة أم قيد التنفيذ أم جاهزة ومن ضمن متغيرات الحالة متغير يمثل أولوية الإجرائية ومتغير يمثل العدد.

2 - **Task State Segment**: وهو هيكل يحتوي على سجلات المعالج والإشارات الأخرى الازمة لاستعادة حالة المعالج عند التبديل بين الإجرائيات.

3 - **هوية الإجرائية PID**: وهو المعرف الخاص بالإجرائية ويحوي رقمين اثنين الأول للإجرائية ذاتها والآخر للأب.

4 - **مؤقتات المستخدم والنظام**: وهو وقت المستخدم الذي استهلكته الإجرائية ووقت النظام المستهلك أيضاً.

5 - **إدارة الذاكرة mm_struct**: وهو هيكل إدارة الذاكرة.

6 - **الإشارات**: وتحوي حقل الإشارات المعلقة ومصفوفة من مؤشرات الدوال لمعالجة الإشارات وإدارتها وأيضاً مؤشر للدالة المستعادة بعد معالجة الإشارة.

7 - **إدارة الجدولة**: وهي تمثل بمؤقت التنبية للإجرائية.

8 - **المكدس Stack**: ويمثل المؤشر إلى مكدس النظام وأيضاً يمثل هيكل يحتوي على معلومات مكدس تنفيذ الإجرائية.

9 - **مجموعة المستخدم والمجموعة**: وتمثل هذه المجموعة معرف المستخدم للإجرائية، معرف المجموعة للإجرائية، معرف المستخدم الفعلي ومعرف المجموعة الفعلية.

هذه الحقول هي الأساسية وتوجد الكثير من الحقول الأخرى التي قد تختلف بناءً على إصدار النواة وتكوين النظام. تستخدم هذه الحقول لإدارة وتحكم الإجرائيات في النظام، وتمكن من التبديل بين الإجرائيات بسلامة، بالإضافة إلى توفير الحماية وإدارة الموارد.

● وجدنا أيضاً في ملف sched.h بيانات أخرى تتعلق بإدارة سياق الإجرائيات والنياسب منها:

1 - **math_state_restore()**:

يحفظ حالة الوحدة الرياضية المعروفة بـ Floating Point Unit للإجرائية السابقة (إذا كانت مستخدمة) ويستعيد حالة الوحدة الرياضية للإجرائية الحالية.

2 - **schedule(void)**:

هوتابع المجدول (scheduler) الرئيسي، الذي يختار الإجرائية التالية التي يجب تشغيلها بناءً على حالتها وأولوية العدد.

3 – sys_pause(void):

يجعل الإجرائية الحالية في حالة TASK_INTERRUPTIBLE، ثم يستدعي المجدول.

4 – sleep_on(struct task_struct ** p):

يجعل الإجرائية الحالية في حالة TASK_UNINTERRUPTIBLE وينتقل إلى الإجرائية التالية في الجدول الزمني. يستخدم للنوم حتى يتم إيقاظه بواسطة . wake_up()

5 – interruptible_sleep_on(struct task_struct ** p):

مشابه للتابع sleep_on ولكن مع الحالة TASK_INTERRUPTIBLE إذا استيقظت الإجرائية بسبب إشارة، فإنها قد تعود للنوم مرة أخرى إذا كانت الشرط لا يزال قائماً.

6 – wake_up(struct task_struct ** p):

يوقظ الإجرائية المحددة إذا كانت نائمة.

7 – do_timer(long cpl):

يتم استدعاؤه بواسطة مقاطعة المؤقت (timer interrupt) لتحديث الوقت المستخدم من قبل الإجرائية الحالية. إما utime أو stime، يستدعي المجدول إذا انتهى عداد الإجرائية الحالية.

8 – sys_alarm(long seconds):

يعين تنبيه (alarm) للإجرائية الحالية، والذي سيتم إطلاقه بعد عدد معين من الثواني.

9 – sys_nice(long increment):

يغير أولوية الإجرائية الحالية بزيادة أو تقليل قيمتها.

10 – sched_init(void):

تهيئة المجدول وتكون الجداول الوصفية TSS (LDT) لجميع الإجرائيات ويقوم بإعداد قنوات المقاطعات لمقاطعة المؤقت واستدعاءات النظام.

• علاقة الملف sched.h بالنياسب والإجرائيات:

ملف sched.h يحتوي على المعلومات الأساسية والتتابع التي تُستخدم لإدارة النياسب (threads) والإجراءات (processes) في نظام لينكس. من خلال البنى والتتابع الموجودة في هذا الملف، يتم التحكم في كيفية جدولة العمليات وإدارتها، مما يضمن تخصيصاً عادلاً وفعالاً لموارد النظام بين جميع العمليات. هذا الملف يعد جزءاً أساسياً من النواة ويحتوي على البنية الأساسية لتحديد الأولويات وتنفيذ الجدولة بين العمليات المختلفة في النظام.

.b يطلب البحث عن إحدى التعليمات الخاصة بمعالج إنتل x86 والتي تعمل في وضعية النواة وتقديم المبرر لكونها لا يمكن أن تستخدم في وضعية المستخدم.

أحد التعليمات الخاصة بمعالج Intel x86 في وضعية النواة و LGDT :

LIDT (Load Interrupt Descriptor Table) و LGDT (Load Global Descriptor Table) هما تعليمات خاصة بمعالج Intel x86 تُستخدم لتحميل عناوين جداول الوصف العام (GDT) وجدول وصف المقاطعات (IDT) في المعالج. هذه التعليمات تُستخدم في إعداد النظام وتغيير سياقات النواة، وهي تعتبر حيوية لأمان واستقرار النظام.

شرح تعليمتي LGDT و LIDT :

: LGDT ○

- الاسم: Load Global Descriptor Table

- الوصف: تُستخدم لتحميل عنوان جدول الوصف العام (GDT) وطوله إلى سجل GDTR في المعالج.

- الصيغة: LGDT m16&32 ، حيث m16&32 هو معامل يشير إلى بنية تحتوي على عنوان 48 بت لـ GDT 16 بت لطول الـ GDT و 32 بت لعنوان البدء.

: LIDT ○

- الاسم: Load Interrupt Descriptor Table

- الوصف: تُستخدم لتحميل عنوان جدول وصف المقاطعات (IDT) وطوله إلى سجل IDTR في المعالج.

- الصيغة: LIDT m16&32 : حيث m16&32 هو معامل يشير إلى بنية تحتوي على عنوان 48 بت لـ IDT 16 بت لطول الـ IDT و 32 بت لعنوان البدء.

سبب عدم إمكانية استخدامها في وضعية المستخدم:

○ أولاً: الأمان والتحكم:

- أمان النظام: تحميل جداول الوصف GDT و IDT يعتبر من العمليات الحساسة جدًا لأنها تحدد وصف القطاعات (segments) ومواصفات المقاطعات (interrupt descriptors) التي تُستخدم للتحكم في الذاكرة والاستجابة للمقاطعات.

إذا تم السماح لهذه التعليمات في وضعية المستخدم ، يمكن للتطبيقات العادلة إعادة تعريف كيفية الوصول إلى الذاكرة وكيفية الاستجابة للمقاطعات، مما يمكن أن يؤدي إلى ثغرات أمنية كبيرة مثل الوصول إلى الذاكرة الممنوعة أو تعطيل نظام المقاطعات.

- التحكم بالنظام: التعليمات LGDT و LIDT تُستخدم عادة في مراحل إعداد النظام أولى تحديات رئيسية للسياق من قبل نظام التشغيل. السماح لها في وضعية المستخدم سيعني أن التطبيقات العادلة يمكن أن تتلاعب بهذه الجداول، مما يؤدي إلى عدم استقرار النظام واحتمال تعطله.
- **ثانياً: حماية وضعية المعالج:**
 - امتيازات التنفيذ: معالجات x86 تفرق بين وضعية النواة (التي تُنفذ فيها التعليمات بامتيازات عالية) ووضعية المستخدم (بامتيازات محدودة). التعليمات التي تغير بيئته النظام بشكل كبير يجب أن تُنفذ فقط في وضعية النواة لحماية نزاهة النظام.
 - استقرار النظام: التعليمات التي تعدل جداول وصف المعالج تؤثر بشكل مباشر على كيفية تعامل النظام مع الذاكرة والمقاطعات. السماح لأي برنامج بتعديل هذه الجداول يمكن أن يؤدي إلى سلوك غير متوقع وتعطل النظام.

الخلاصة:

تعليمات مثل LGDT و LIDT تُستخدم لتحميل جداول وصف هامة في المعالج، وهذه التعليمات يجب أن تكون مقتصرة على وضعية النواة لأسباب تتعلق بالأمان واستقرار النظام. السماح باستخدامها في وضعية المستخدم يمكن أن يؤدي إلى ثغرات أمنية كبيرة وعدم استقرار النظام.

ما هي العلاقة بين نواة نظام التشغيل ومحظى المجلد proc ؟ وأين يمكن العثور على معلومات الإجرائيات والنياسب ضمن هذا المجلد؟ أعطِ مثلاً عن استخدام هذه المعلومات.

- نواة نظام التشغيل لينوكس (Linux kernel) هي الجزء المركزي من نظام التشغيل الذي يتحكم في موارد الجهاز مثل الذاكرة والمعالج والتخزين. نواة النظام تعمل ك وسيط بين التطبيقات والأجهزة المادية، مما يسمح للبرامج بالتفاعل مع الأجهزة دون الحاجة لمعرفة تفاصيلها المادية.
- مجلد proc في لينوكس هو جزء من نظام الملفات الافتراضي procfs، ويستخدم لتوفير واجهة للوصول إلى المعلومات حول النظام والعمليات الجارية. يمكن اعتباره نافذة لنواة النظام، حيث يعرض العديد من الإعدادات والمعلمات والتفاصيل حول النظام والعمليات التي تديرها النواة.
- العلاقة بين نواة نظام التشغيل لينوكس ومحظى المجلد proc:

تكمّن في أن النواة تقوم بإنشاء وتحديث هذه الملفات الافتراضية في الوقت الفعلي لتعكس حالة النظام الحالية. من خلال قراءة الملفات الموجودة في هذا المجلد، يمكن للمستخدمين والبرامج الحصول على معلومات حيوية حول النظام مثل:

- proc cpuinfo: يوفر معلومات عن المعالج مثل النوع، النموذج، السرعة، وعدد النوى. على سبيل المثال، هذه بعض المعلومات التي يمكن العثور عليها في proc cpuinfo

```
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 106
model name : unknown
cpu MHz : 2793.439
physical id : 0
siblings : 16
core id : 0
cpu cores : 16
```

- proc/meminfo: يحتوي على معلومات حول الذاكرة المتوفرة والمستخدمة في النظام. على سبيل المثال، يمكن العثور على معلومات مثل:

MemTotal : 1048576 kB

MemFree : 811912 kB

MemAvailable : 811912 kB

Buffers : 0 kB

Cached : 32232 kB

SwapCached : 0 kB

- proc/[PID]: يحتوي على معلومات حول العمليات الجارية، حيث [PID] هو معرف العملية.

على سبيل المثال، يمكن استخدام المعلومات من proc cpuinfo لمعرفة تفاصيل المعالج مثل النموذج والسرعة، ومن proc meminfo لمعرفة مقدار الذاكرة المتوفرة والمستخدمة في النظام. هذه المعلومات يمكن استخدامها لأغراض المراقبة والتحليل لتحديد أداء النظام وتشخيص المشاكل المحتملة. مجلد proc هو نقطة تفاعل بين المستخدم والنواة، حيث يعرض معلومات ديناميكية حول النظام والعمليات النشطة. يتم تحديث محتويات هذا المجلد بشكل مباشر من قبل النواة، وبالتالي تعكس دائمًا الحالة الحالية للنظام.

- معلومات الإجرائيات (Processes) والأنساب (Threads) ضمن مجلد proc :

يمكن العثور على معلومات عن العمليات والأنساب داخل مجلد proc في مجلدات مسممة بأرقام، حيث يمثل كل رقم معرف العملية (PID). داخل كل من هذه المجلدات، توجد ملفات توفر تفاصيل مختلفة عن العملية المعنية.

- بعض الملفات الهامة داخل مجلد [PID] :

proc/[PID]/cmdline: يعرض سطر الأوامر المستخدم لتشغيل العملية.

proc/[PID]/status: يعرض حالة العملية، بما في ذلك معلومات حول الذاكرة واستخدام المعالج.

proc/[PID]/stat: يحتوي على إحصاءات تفصيلية حول العملية.

proc/[PID]/fd: يحتوي على روابط رمزية إلى الملفات المفتوحة من قبل العملية.

proc/[PID]/task/: يحتوي على معلومات حول الأنشطة الفرعية (threads) داخل العملية.

◦ مثال على استخدام هذه المعلومات:

لنفترض أننا نريد مراقبة استخدام الذاكرة والموارد لعملية معينة. إذاً يمكننا استخدام الملفات داخل مجلد proc للحصول على هذه المعلومات.

مثال:

1. العثور على معرف العملية (PID):

يمكننا استخدام أمر ps أو grep للعثور على PID للعملية.

```
ps aux | grep [اسم العملية]
```

2. قراءة معلومات الذاكرة من :proc/[PID]/status

لنفترض أن PID هو 197899.

```
cat /proc/197899/status
```

سيعرض هذا الأمر معلومات شاملة عن حالة العملية، مثل:

Name: [اسم العملية]

State: [حالة العملية]

VmSize: [حجم الذاكرة الافتراضية]

VmRSS: [حجم الذاكرة الفعلية]

3. قراءة إحصاءات المعالج من :proc/[PID]/stat

```
cat /proc/197899/stat
```

سنجد هنا معلومات مفصلة تشمل:

- وقت المعالج المستخدم.

- حالة التنفيذ.

- الأولوية.

الخلاصة:

يوفر مجلد `proc` واجهة ديناميكية يمكن من خلالها الحصول على معلومات مفصلة حول العمليات والنظام في وقت التشغيل. هذه المعلومات يمكن استخدامها لمراقبة أداء النظام، تحليل سلوك العمليات، أو حتى تعديل بعض إعدادات النواة بشكل مباشر في حالة المستخدمين المتقدمين.

الاعتماد على `proc` يساعد في الحصول على معلومات حية ودقيقة حول حالة النظام والعمليات التي تديرها نواة لينوكس، مما يجعله أداة قيمة لإدارة النظام وصيانته.