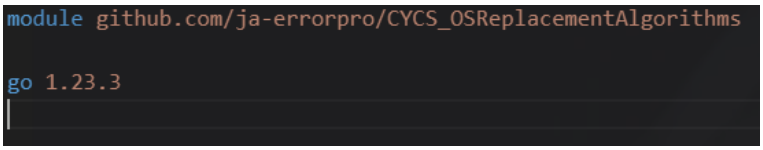
作業系統 HW2 報告

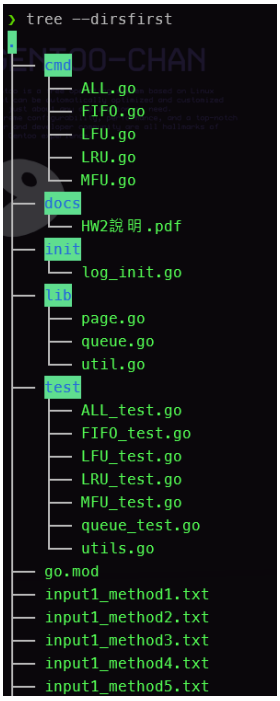
11127137 資工三甲 黃乙家

一、 開發環境



這次作業使用 Go 語言開發，系統為 Windows 11 Pro 23H2，CPU i7-12700H，記憶體 32G，而為了使 Linux 也能方便編譯，也提供了 Makefile，且這次作業沒有使用任何第三方套件，理應不需要 go mod download 或者 go mod tidy 來整理套件。

二、 專案架構

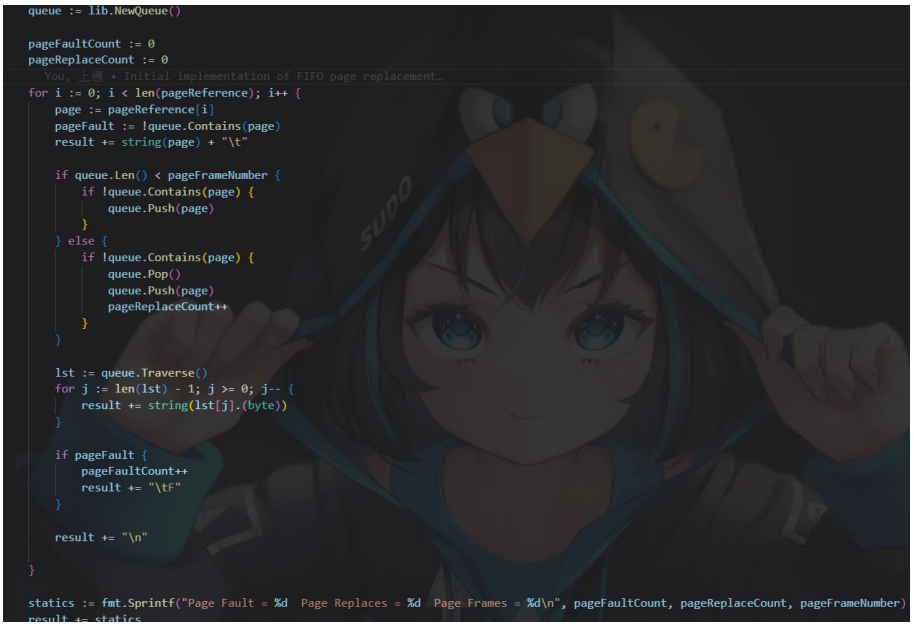


專案根目錄包含 main.go，而 cmd 目錄裡為六種演算法的實作程式碼，init 目錄為主程式執行前的必要初始化(在這次作業僅作為方便 Debug 使用)，lib 目錄包含各種用到的資料結構，test 目錄為資料結構與各種方法的 Unit Test。

三、 實作方法與運作流程

```
func main() {
    var inputFileName string
    fmt.Print("Please enter File Name (eg. input1 、 input1.txt) : ")
    fmt.Scanln(&inputFileName)

    method, pageFrameNumber, pageReferenceSequence := ReadFile(inputFileName)

    var outputFileName string = "out_" + inputFileName

    if outputFileName[len(outputFileName)-4:] != ".txt" {
        outputFileName += ".txt"
    }

    var result string

    switch method {
    case methods.FIFO:
        result = methods.FIFOCache(pageFrameNumber, pageReferenceSequence)
    case methods.LRU:
        result = methods.LRU_Cache(pageFrameNumber, pageReferenceSequence)
    case methods.LFU_FIFO:
        result = methods.LFU_FIFO_Cache(pageFrameNumber, pageReferenceSequence)
    case methods.MFU:
        result = methods.MFU_FIFO_Cache(pageFrameNumber, pageReferenceSequence)
    case methods.LFU_LRU:
        result = methods.LFU_LRU_Cache(pageFrameNumber, pageReferenceSequence)
    case methods.ALL:
        result = methods.AllCache(pageFrameNumber, pageReferenceSequence)
    default:
        log.Fatal("Invalid method")
    }

    WriteFile(outputFileName, result)
    fmt.Print(result)
}
```

一開始，主程式會從 stdin 讀入檔案名稱，讀檔成功後判斷方法數，把必要的參數帶入，呼叫對應的方法。



```
queue := lib.NewQueue()

pageFaultCount := 0
pageReplaceCount := 0
for i := 0; i < len(pageReference); i++ {
    page := pageReference[i]
    pageFault := !queue.Contains(page)
    result += string(page) + "\t"

    if queue.Len() < pageFrameNumber {
        if !queue.Contains(page) {
            queue.Push(page)
        }
    } else {
        if !queue.Contains(page) {
            queue.Pop()
            queue.Push(page)
            pageReplaceCount++
        }
    }

    lst := queue.Traverse()
    for j := len(lst) - 1; j >= 0; j-- {
        result += string(lst[j].(byte))
    }

    if pageFault {
        pageFaultCount++
        result += "\tF"
    }

    result += "\n"

}

statics := fmt.Sprintf("Page Fault = %d  Page Replaces = %d  Page Frames = %d\n", pageFaultCount, pageReplaceCount, pageFrameNumber)
```

上圖為 FIFO 方法的程式，使用佇列資料結構實作，另外實作了 Contains 方法以線性搜尋是否包含元素，如果不包含表示發生 Fault，否則如果佇列滿了則刪除最先進入佇列的 page，然後新的再放進佇列，而若未滿則直接放進佇列即可。

```go
lst := []lib.Page{}
inlist := make(map[byte]bool) // key: page reference, value: is in the page frame
  You, 上週 • Add page replacement algorithm implementations …
pageFaultCount := 0
pageReplaceCount := 0

for i := 0; i < len(pageReference); i++ {
    pageKey := pageReference[i]
    page := lib.NewPage(pageKey)
    pageFault := !inlist[pageKey]
    result += string(pageKey) + "\t"

    if pageFault {
        pageFaultCount++
        if len(lst) == pageFrameNumber {
            pageReplaceCount++
            inlist[lst[0].Reference] = false
            lst = lst[1:]
        }
        lst = append(lst, *page)
        inlist[pageKey] = true
    } else {
        for j, p := range lst {
            if p.Reference == pageKey {
                // move the page to the tail(head) of the list
                lst = append(lst[:j], lst[j+1:]...)
                lst = append(lst, *page)
                break
            }
        }
    }

    for j := len(lst) - 1; j >= 0; j-- {
        result += string(lst[j].Reference)
    }

    if pageFault {
        result += "\tF"
    }

    result += "\n"
}
```

LRU 演算法採用 list 實作，另外還使用 hashMap 進行搜尋加速，每次 pageReference 先查看是否有在 list 中，若有則需將該 page 移到 list 最前面或最後面，表示這是最近使用到的，如果沒有且 list 已滿則將最後面或最前面的 page 替換掉(即淘汰最久沒被使用的)，為了方便實作，程式碼總是會替換最前面的 page。

```go
for i := 0; i < len(pageReference); i++ {
    pageKey := pageReference[i]
    page := lib.NewPage(pageKey)
    pageFault := !inlist[pageKey]
  You, 4 天前 • Implement page replacement algorithms and add i…
    result += string(pageKey) + "\t"

    if pageFault {
        pageFaultCount++
        if len(lst) == pageFrameNumber {
            pageReplaceCount++
            minIndex := FindLFUPage(lst)
            if minIndex != -1 {
                inlist[lst[minIndex].Reference] = false
                // lst[minIndex] = *page
                lst = append(lst[:minIndex], lst[minIndex+1:]...)
                lst = append(lst, *page)
            } else {
                inlist[lst[0].Reference] = false
                lst = lst[1:]
                lst = append(lst, *page)
            }
        } else {
            lst = append(lst, *page)
        }
        inlist[pageKey] = true
    } else {
        for j, p := range lst {
            if p.Reference == pageKey {
                p.Freq++
                lst[j] = p
                break
            }
        }
    }
```

而 LFU/MFU 則是改成每次 reference 需要紀錄次數，如果需要 replace 則將 list 中次數最低或最高的淘汰。

```
if pageFault {
    pageFaultCount++
    if len(lst) == pageFrameNumber {
        pageReplaceCount++
        minIndex := FindLFUPage(lst)
        if minIndex != -1 {
            inlist[lst[minIndex].Reference] = false
            // lst[minIndex] = *page
            lst = append(lst[:minIndex], lst[minIndex+1:]...)
            lst = append(lst, *page)
        } else {
            inlist[lst[0].Reference] = false
            lst = lst[1:]
            lst = append(lst, *page)
        }
    } else {
        lst = append(lst, *page)
    }
    inlist[pageKey] = true
} else {
    for j, p := range lst {
        if p.Reference == pageKey {
            // move the page to the tail(head) of the list
            p.Freq++
            page = &p
            lst = append(lst[:j], lst[j+1:]...)
            lst = append(lst, *page)
            break
        }
    }
```

LFU_LRU 則是將兩種方法結合在一起，如果 reference 需將 page 移到最前面或最後面，同時記錄次數，並且當發生 replace 時若次數相同則依 LRU 的規則淘汰對應的 page。

四、 分析

使用測資二來分析後發現，FIFO 和 MFU 的 Fault 次數為 15 比其他方法多，而 Replacement 次數是 12 也是最多的，推測原因可能是 page reference 重複的次數較多，然而 FIFO 並沒有考慮頻率問題，MFU 則是會將頻率高的淘汰掉，使得錯誤率變高。

```
1  ----------FIFO----------
2  7   7   F
3  0   07  F
4  1   107 F
5  2   210 F
6  0   210
7  3   321 F
8  0   032 F
9  4   403 F
10 2   240 F
11 3   324 F
12 0   032 F
13 3   032
14 2   032
15 1   103 F
16 2   210 F
17 0   210
18 1   210
19 7   721 F
20 0   072 F
21 1   107 F
22 Page Fault = 15  Page Replaces = 12  Page Frames = 3
```

```
1  ----------FIFO----------
2  7   7   F
3  0   07  F
4  1   107 F
5  2   210 F
6  0   210
7  3   321 F
8  0   032 F
9  4   403 F
10 2   240 F
11 3   324 F
12 0   032 F
13 3   032
14 2   032
15 1   103 F
16 2   210 F
17 0   210
18 1   210
19 7   721 F
20 0   072 F
21 1   107 F
22 Page Fault = 15  Page Replaces = 12  Page Frames = 3
```

使用測資一分析時，發現如果 page Frame Number 提高，FIFO 與 MFU 的錯誤率理應下降卻反而上升了，也就是增加頁面幀數卻導致錯誤數增加，這種現象被稱為 Belady’s anomaly

```
1  ----------FIFO----------
2  1   1   F
3  2   21  F
4  3   321 F
5  4   4321    F
6  1   4321
7  2   4321
8  5   5432    F
9  1   1543    F
10 2   2154    F
11 3   3215    F
12 4   4321    F
13 5   5432    F
14 Page Fault = 10  Page Replaces = 6  Page Frames = 4
15
```

```
1  ----------FIFO----------
2  1   1   F
3  2   21  F
4  3   321 F
5  4   432 F
6  1   143 F
7  2   214 F
8  5   521 F
9  1   521
10 2   521
11 3   352 F
12 4   435 F
13 5   435
14 Page Fault = 9  Page Replaces = 6  Page Frames = 3
15
```