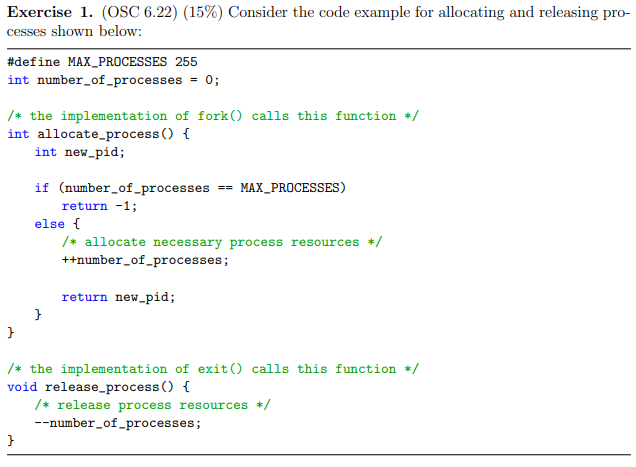
Austin Smothers

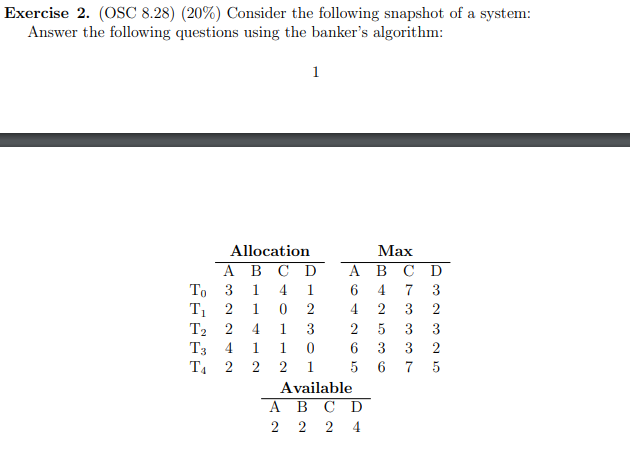
CSC 139

April 26, 2020

Homework 2



1. Identify the race condition(s).
   1. Many forked processes could end up in the else statement at the same time, allowing the number of processes to exceed 255.
   2. More threads cannot be released than existed initially, so this isn't a concern with the release function. Technically you could have a process get denied allocation while another process is mid release though.
2. Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).
   1. aquire() gets placed after the declaration of new\_pid in the allocate\_process() function.
   2. release() gets placed before both return statements in allocate\_process().
   3. For posterity's sake, there SHOULD also be an acquire() in release\_process() before --number\_of\_processes is called, an a release() after --number\_of\_processes is called.



1. Illustrate that the system is in a safe state by demonstrating an order in which the threads may complete.
   1. The need matrix for our algorithm is as follows

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Need | | | | Available | | | |
|  | A | B | C | D | A | B | C | D |
| T0 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 4 |
| T1 | 2 | 1 | 3 | 0 |
| T2 | 0 | 1 | 2 | 0 |
| T3 | 2 | 2 | 2 | 2 |
| T4 | 3 | 4 | 5 | 4 |

* 1. By the Banker's algorithm, T0 cannot be executed, so we try T1, which also cannot be executed, so we try T2, which can safely be executed, and yields the following: **<T2>**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Need | | | | Available | | | |
|  | A | B | C | D | A | B | C | D |
| T0 | 3 | 3 | 3 | 2 | 4 | 6 | 3 | 7 |
| T1 | 2 | 1 | 3 | 0 |
| T3 | 2 | 2 | 2 | 2 |
| T4 | 3 | 4 | 5 | 4 |

* 1. T3 can safely be executed, which yields the following: **<T2, T3>**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Need | | | | Available | | | |
|  | A | B | C | D | A | B | C | D |
| T0 | 3 | 3 | 3 | 2 | 8 | 7 | 4 | 7 |
| T1 | 2 | 1 | 3 | 0 |
| T4 | 3 | 4 | 5 | 4 |

* 1. T4 cannot be safely executed, so we try T0 again, which can be safely executed and yields the following: **<T2, T3, T0>**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Need | | | | Available | | | |
|  | A | B | C | D | A | B | C | D |
| T1 | 2 | 1 | 3 | 0 | 11 | 8 | 8 | 8 |
| T4 | 3 | 4 | 5 | 4 |

* 1. T1 can be safely executed, which yields the following: **<T2, T3, T0, T1>**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Need | | | | Available | | | |
|  | A | B | C | D | A | B | C | D |
| T4 | 3 | 4 | 5 | 4 | 13 | 9 | 8 | 10 |

* 1. We have more resources than T4 needs now, yielding the following safe assuming the following order of execution: **<T2, T3, T0, T1, T4>**

1. If a request from thread T4 arrives for (2,2,2,4), can the request be granted immediately?
   1. No, as it would yield the following unsafe system state, which is deadlocked:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Need | | | | Available | | | |
|  | A | B | C | D | A | B | C | D |
| T0 | 3 | 3 | 3 | 2 | 0 | 0 | 0 | 0 |
| T1 | 2 | 1 | 3 | 0 |
| T2 | 0 | 1 | 2 | 0 |
| T3 | 2 | 2 | 2 | 2 |
| T4 | 1 | 2 | 3 | 0 |

1. If a request from thread T2 arrives for (0,1,1,0), can the request be granted immediately?
   1. Yes, as it leaves the system in the following safe state:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Need | | | | Available | | | |
|  | A | B | C | D | A | B | C | D |
| T0 | 3 | 3 | 3 | 2 | 2 | 1 | 1 | 4 |
| T1 | 2 | 1 | 3 | 0 |
| T2 | 0 | 0 | 1 | 0 |
| T3 | 2 | 2 | 2 | 2 |
| T4 | 3 | 4 | 5 | 4 |

This system can follow the same order of execution as described in part 1  
**<T2, T3, T0, T1, T4>**

1. If a request from thread T3 arrives for (2,2,1,2), can the request be granted immediately?
   1. Yes, as it leaves the system in the following safe state:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Need | | | | Available | | | |
|  | A | B | C | D | A | B | C | D |
| T0 | 3 | 3 | 3 | 2 | 0 | 0 | 1 | 2 |
| T1 | 2 | 1 | 3 | 0 |
| T2 | 0 | 1 | 2 | 0 |
| T3 | 0 | 0 | 1 | 0 |
| T4 | 3 | 4 | 5 | 4 |

This system can safely execute in the order of: **<T3, T0, T1, T2, T4>**



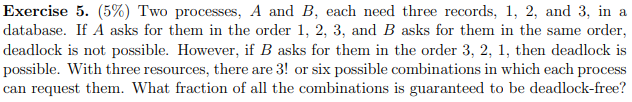
1. Yes, this system is deadlock-free. Each process can only have a maximum of two resources allocated before it successfully executes. Since there are four resources and three processes, at least one process will be able to immediately obtain two resources. That process will execute successfully, leaving four resources to split between two processes which need two resources each. Those processes will both have sufficient resources to execute, leaving the system deadlock-free.



1. An unsafe state implies that deadlock will eventually become a reality. It does not imply that the system is currently deadlocked, as there could be some processes which can still run before the system reaches deadlock. Take the following for example:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Allocated | | | |  | Need | | | | Available | | | |
|  | A | B | C | D |  | A | B | C | D | A | B | C | D |
| T0 | 5 | 5 | 5 | 5 | T0 | 10 | 10 | 10 | 10 | 1 | 1 | 1 | 1 |
| T1 | 1 | 1 | 1 | 1 | T1 | 1 | 1 | 1 | 1 |
| T2 | 5 | 5 | 5 | 5 | T2 | 10 | 10 | 10 | 10 |

In this system, we are definitely unsafe, as T0 and T2 cannot run. However, we are not deadlocked YET, because we can execute T1 successfully before we reach deadlock.



123, 312, 231, 321, 132, 213

Given two processes, we have (3!)2 total combinations. Of those, only the combinations that ask for resources in the exact same order are guaranteed to be deadlock free, the chances of which are (3!).

So in total, we have (3!)/(3!)2 safe combinations, or 6/36 🡪 **1/6 are deadlock free**



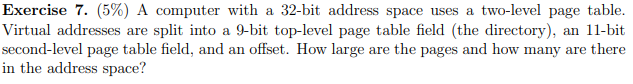
Virtual space = 48-bit

Physical space = 32-bit

Page size = 8KB = 213 Bytes

Offset = 13 bits

# of virt pages = 248 - 13 = **235** = **# of entries in page table per process**



Address space = 32-bit

Bits used for page # = 9 + 11 = 20-bit

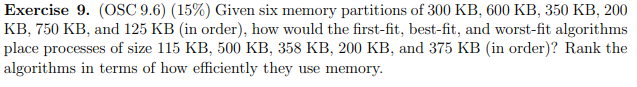
Offset = 32 - 20 = 12 bit

Page Size = 212 Bytes = **4KB**

# of pages = **220 pages**



A page fault occurs when an instruction references a page that is not in memory. The first reference to said page will trap to the operating system. If the operating system determines that the referenced page is invalid, the referencing process aborts. Otherwise, the operating system determines that the reference page simply isn't in memory, and will find a free frame to swap the page into. The OS will set the validation bit for that page, reset the page table, and then restart the instruction that caused the page fault.



* First Fit:

|  |  |
| --- | --- |
| 300KB | 115KB |
|  |
| 600KB | **500KB** |
|  |
| 350KB | **200KB** |
|  |
| 200KB |  |
| 750KB | **358KB** |
| **375KB** |
| 125KB |  |

All processes placed. 185KB + 100KB + 150KB + 17KB =   
**452KB internal fragmentation**

* Best-fit:

|  |  |
| --- | --- |
| 300KB |  |
|  |
| 600KB | **500KB** |
|  |
| 350KB |  |
|  |
| 200KB | **200KB** |
| 750KB | **358KB** |
| **375KB** |
| 125KB | **115KB** |

All processes placed. 100KB + 17KB + 10KB = **127KB internal fragmentation**

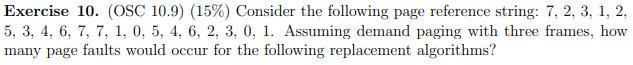
* Worst-fit:

|  |  |
| --- | --- |
| 300KB |  |
|  |
| 600KB | **358KB** |
|  |
|  |
| 350KB | **200KB** |
|  |
| 200KB |  |
| 750KB | **115KB** |
| **500KB** |
| 125KB |  |

Unable to place last process. 242KB + 150KB + 135KB =

**527 KB internal fragmentation**

* Ranking:
  1. Best-fit (lowest fragmentation + all processes placed)
  2. First-fit (intermediate fragmentation + all processes placed)
  3. Worst-fit (highest fragmentation + failed to place one process)



* LRU

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Reference String | | | | | | | | | | | | | | | | | | | |
| **7** | **2** | **3** | **1** | **2** | **5** | **3** | **4** | **6** | **7** | **7** | **1** | **0** | **5** | **4** | **6** | **2** | **3** | **0** | **1** |
| 7 | 7 | 7 | 2 | 3 | 1 | 2 | 5 | 3 | 4 | 4 | 6 | 7 | 1 | 0 | 5 | 4 | 6 | 2 | 3 |
|  | 2 | 2 | 3 | 1 | 2 | 5 | 3 | 4 | 6 | 6 | 7 | 1 | 0 | 5 | 4 | 6 | 2 | 3 | 0 |
|  |  | 3 | 1 | 2 | 5 | 3 | 4 | 6 | 7 | 7 | 1 | 0 | 5 | 4 | 6 | 2 | 3 | 0 | 1 |
| Page Frames | | | | | | | | | | | | | | | | | | | |

18 page faults

* FIFO

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Reference String | | | | | | | | | | | | | | | | | | | |
| **7** | **2** | **3** | **1** | **2** | **5** | **3** | **4** | **6** | **7** | **7** | **1** | **0** | **5** | **4** | **6** | **2** | **3** | **0** | **1** |
| 7 | 7 | 7 | 2 | 2 | 3 | 3 | 1 | 5 | 4 | 4 | 6 | 7 | 1 | 0 | 5 | 4 | 6 | 2 | 3 |
|  | 2 | 2 | 3 | 3 | 1 | 1 | 5 | 4 | 6 | 6 | 7 | 1 | 0 | 5 | 4 | 6 | 2 | 3 | 0 |
|  |  | 3 | 1 | 1 | 5 | 5 | 4 | 6 | 7 | 7 | 1 | 0 | 5 | 4 | 6 | 2 | 3 | 0 | 1 |
| Page Frames | | | | | | | | | | | | | | | | | | | |

17 page faults

* Optimal

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Reference String | | | | | | | | | | | | | | | | | | | |
| **7** | **2** | **3** | **1** | **2** | **5** | **3** | **4** | **6** | **7** | **7** | **1** | **0** | **5** | **4** | **6** | **2** | **3** | **0** | **1** |
| 7 | 7 | 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|  | 2 | 2 | 2 | 2 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 6 | 2 | 3 | 3 | 3 |
|  |  | 3 | 3 | 3 | 3 | 3 | 4 | 6 | 7 | 7 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Page Frames | | | | | | | | | | | | | | | | | | | |

13 page faults

1. I spent about 5 hours on this homework
2. I'd give this homework a 4. The virtual addressing problems in particular were quite difficult
3. Graphing out all of the different problems to show my work was a pain, and I don't feel like the lectures adequately explained problems 6-7 (had to use google to fill those gaps), but otherwise the homework wasn't so bad.