

CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Project 3 (document version 1.2)
Memory Management and Virtual Memory

Overview

- This project is due by 11:59:59 PM on Monday, May 9, 2016. Projects are to be submitted electronically.
- This project will count as 8% of your final course grade.
- This project is to be completed **individually** or in a team of at most three students. Do not share your code with anyone else.
- You **must** use one of the following programming languages: C, C++, Java, or Python. If using Java, name your main Java file **Project3.java**.
- Your program **must** successfully compile and run on Ubuntu v14.04.3 LTS.

Project Specifications

In this third and final project, you will simulate a variety of memory management systems, including contiguous, non-contiguous, and virtual memory. In an operating system, each process has specific memory requirements. These memory requirements are met (or not) based on whether free memory is available to fulfill such requirements.

Representing Physical (Main) Memory

For contiguous and non-contiguous memory schemes, to represent physical memory, use a data structure of your choice to represent a memory that contains a configurable number of equally sized “memory units” or frames.

As an example, memory may be represented as shown below (also see examples in the in-class notes). When you display your simulated memory, use the format below, showing 32 frames per line. Also, as a default, simulate a memory consisting of 256 frames (i.e., eight output lines). These two values should be configurable and easily tunable.

```
=====
AAAAAAAABBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBB.....
.....DDDDDDDD.....
.....
.....
.....HHHHHHHHHHHHHHHHHHHHHHHH
HHH.....
.....
=====
```

More specifically, use the following character mappings:

- A '.' character represents a free memory frame.
- An ASCII character in the range 'A'-'Z' (i.e., uppercase letters) represents a user process memory frame.

Note that we will ignore operating system process memory frames.

Input File

For contiguous and non-contiguous memory schemes, details of user processes must be read from an input file, which is to be specified as the first command-line argument of your simulation. The input file is formatted as shown below.

```
N
proc1 p1_mem p1_arr_time_1-p1_exit_time_1 ... p1_arr_time_a-p1_exit_time_a
proc2 p2_mem p2_arr_time_1-p2_exit_time_1 ... p1_arr_time_b-p1_exit_time_b
...
procN pN_mem pN_arr_time_1-pN_exit_time_1 ... p1_arr_time_z-p1_exit_time_z
```

Here, the overall number of processes (N) to simulate is specified first on a line by itself. Each `proc#` value specifies a single character in the range 'A'-'Z' that will uniquely identify the given process. Each `p#_mem` value specifies the required (fixed) number of memory frames.

Each `p#_arr_time_?-p#_exit_time_?` pair specifies a corresponding pair of arrival and exit times that the process will experience. You may assume that each process has an increasing set of these values; in other words, you do not need to validate this in your code. Further, assume there is no overhead in adding/removing processes from memory.

You can assume that the maximum number of processes in the simulation will be 26. Do not assume that processes will be given in alphabetical order.

Below is an example input file (note that values are delimited by one or more space or TAB characters):

```
20
A 45 0-350 400-450 550-750
B 28 0-2650
C 58 0-950 1100-1200
D 86 0-650 1350-1800
E 14 0-1400
F 24 100-480 500-975
G 13 435-1250
etc.
```

Note that for the contiguous memory management scheme, when defragmentation occurs, these numbers must be automatically adjusted by extending arrival/exit times accordingly.

Also note that any “ties” that occur should be handled using lexicographical (i.e., alphabetical) order of process IDs.

Contiguous Memory Management

For this portion of the simulation, each process's memory requirements must be met by the various contiguous memory allocation schemes that we have studied. The algorithms to simulate are the *first-fit*, *next-fit*, and *best-fit* algorithms, each of which is described in more detail below.

Note that we will only use a dynamic partitioning scheme here, meaning that your data structure needs to maintain a list containing (1) where each process is allocated, (2) how much contiguous memory each process uses, and (3) where and how much free memory is available (i.e., where each free partition is).

As an example, consider the simulated memory shown below.

```
=====
AAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBB.....
.....DDDDDDDD....
.....
.....
.....HHHHHHHHHHHHHHHHHHHHHHHH
HHH.....
.....
=====
```

In the above example diagram, four processes are allocated in four dynamically allocated partitions. Further, there are three free partitions (i.e., between processes B and D, between processes D and H, and between process H and the “bottom” of memory).

Placement Algorithms

When a process Q wishes to enter the system, memory must first be allocated dynamically. More specifically, a dynamic partition must be created out of an existing free partition.

For the first-fit algorithm, process Q is placed into the first free partition available in scanning from the “top” of memory.

For the next-fit algorithm, process Q is placed into the first free partition available in scanning from the end of the most recently placed process.

For the best-fit algorithm, process Q is placed into the smallest free partition available into which process Q fits. If a “tie” occurs, use the free partition closer to the top of memory.

For all three of these placement algorithms, the memory scan covers the entire memory. If necessary (i.e., if the scan hits the bottom of memory), the scan continues from the top of memory.

If no suitable free partition is available, then an out-of-memory error occurs, at which point defragmentation is required.

Defragmentation

If a process is unable to be placed into memory, defragmentation occurs if the overall total of free memory is sufficient to fit the given process. In such cases, processes are relocated as necessary, starting from the top of memory. If there is not enough memory to admit the given process, your simulation must skip defragmentation and deny the request and move on to the next process.

Note that the given process is only skipped for the given requested interval. It may be admitted to the system at a later interval. For example, given process Q below, if the process is unable to be added in interval 200–580, it might still be successfully added in interval 720–975.

Q 47 200–580 720–975

Once defragmentation is started, it will run through to completion, at which point, the process that triggered defragmentation will be admitted to the system.

While defragmentation is running, no other processes may be executing (i.e., all processes are essentially placed in a suspended state) until all relocations are complete. Thus, arrivals and exits are suspended during defragmentation.

Note that the time to move **one frame of memory** is defined as `t_memmove` and is measured in milliseconds. Assume that the default value of `t_memmove` is 1.

Given the memory shown previously, the results of defragmentation will be as follows (i.e., processes D and H are moved upwards in memory):

```
=====
AAAAAAAABBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBDDDDDDDDHHHHHHHH
HHHHHHHHHHHHHHHHHHHH.....
.....
.....
.....
.....
.....
=====
```

Simulation and Output Requirements

For the given input file, simulate each of the three placement algorithms, displaying output for each “interesting” event that occurs. Interesting events are:

- Simulation start
- Process arrival
- Process placement in physical memory
- Process exit from physical memory
- Start of defragmentation
- End of defragmentation (i.e., process(es) moved)
- Simulation end

As with previous projects, your simulator must keep track of elapsed time t (measured in milliseconds), which is initially set to zero. As your simulation proceeds based on the input file, t advances to each “interesting” event that occurs, displaying a specific line of output describing each event.

Note that your simulator output should be entirely deterministic. To achieve this, your simulator must output each “interesting” event that occurs using the format shown below.

```
time <t>ms: <event-details>
```

Given the example input file at the bottom of page 2 and default configuration parameters, your simulator output would be as follows:

```
time 0ms: Simulator started (Contiguous -- First-Fit)
time 0ms: Process A arrived (requires 45 frames of physical memory)
time 0ms: Placed process A in memory:
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA.....
.....
.....
.....
.....
.....
.....
.....
=====
```

time 0ms: Process B arrived (requires 28 frames of physical memory)
time 0ms: Placed process B in memory:

```
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
BBBBBBBBB.....
.....
.....
.....
.....
.....
=====
```

time 0ms: Process C arrived (requires 58 frames of physical memory)
time 0ms: Placed process C in memory:

```
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
BBBBBBBBBCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCC.....
.....
.....
.....
=====
```

time 0ms: Process D arrived (requires 86 frames of physical memory)
time 0ms: Placed process D in memory:

```
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
BBBBBBBBBCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDD.....
.....
=====
```

time 0ms: Process E arrived (requires 14 frames of physical memory)
time 0ms: Placed process E in memory:

```
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
BBBBBBBBBCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEE
EEEEEEE.....
=====
```

time 100ms: Process F arrived (requires 24 frames of physical memory)
time 100ms: Placed process F in memory:

```
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
BBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEE
EEEEEEFFFFFFF.
=====
```

time 350ms: Process A removed from physical memory

```
=====
.....
.....BBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEE
EEEEEEFFFFFFF.
=====
```

time 400ms: Process A arrived (requires 45 frames of physical memory)

time 400ms: Placed process A in memory:

```
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
BBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEE
EEEEEEFFFFFFF.
=====
```

time 435ms: Process G arrived (requires 13 frames of physical memory)

time 435ms: Cannot place process G -- skipping process G

```
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
BBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEE
EEEEEEFFFFFFF.
=====
```

```

time 450ms: Process A removed from physical memory
=====
.....
.....BBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCC
CCDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDEEEEEEE
EEEEEEFFFFFFFFFFFFFFFFFFFFFFFFF.
=====
time 480ms: Process F removed from physical memory
=====
.....
.....BBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCC
CCDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDEEEEEEE
EEEEEE.....
=====
time 500ms: Process F arrived (requires 24 frames of physical memory)
time 500ms: Placed process F in memory:
=====
FFFFFFFFFFFFFFFFFFFFFFFF.....
.....BBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCC
CCDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDEEEEEEE
EEEEEE.....
=====
time 550ms: Process A arrived (requires 45 frames of physical memory)
time 550ms: Cannot place process A -- starting defragmentation
time 736ms: Defragmentation complete (moved 186 frames: B, C, D, E)
=====
FFFFFFFFFFFFFFFFFFFFFFFFBBBBBBBB
BBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDEEEEEEEEEEEEEEE.....
.....
=====

```



```

time 736ms: Placed process A in memory:
=====
FFFFFFFFFFFFFFFFFFFFFFFFBBBBBBBB
BBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDEEEEEEEEEEEEEEEEEAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.
=====

time 836ms: Process D removed from physical memory
=====
FFFFFFFFFFFFFFFFFFFFFFFFBBBBBBBB
BBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCC.....
.....
.....
...EEEEEEEEEEEEEEEEEEEEAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.
=====

...
time ###ms: Simulator ended (Contiguous -- First-Fit)

time 0ms: Simulator started (Contiguous -- Next-Fit)
time 0ms: Process A arrived (requires 45 frames of physical memory)
time 0ms: Placed process A in memory:
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA.....
.....
.....
.....
.....
.....
.....
=====

...
time ###ms: Simulator ended (Contiguous -- Next-Fit)

```

```
time 0ms: Simulator started (Contiguous -- Best-Fit)
time 0ms: Process A arrived (requires 45 frames of physical memory)
time 0ms: Placed process A in memory:
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA.....
.....
.....
.....
.....
.....
.....
=====
...
time ###ms: Simulator ended (Contiguous -- Best-Fit)
```

Non-contiguous Memory Management

Extend the above contiguous memory management simulation by next simulating a non-contiguous memory management scheme in which you use a page table to map each logical page of a process to a physical frame of memory.

The key difference here is that defragmentation is no longer necessary. To place pages into frames of physical memory, use a simple first-fit approach, as shown in the example below.

```
time 0ms: Simulator started (Non-contiguous)
time 0ms: Process A arrived (requires 45 frames of physical memory)
time 0ms: Placed process A in memory:
=====
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA.....
.....
.....
.....
.....
.....
.....
=====
...
time 500ms: Process F arrived (requires 24 frames of physical memory)
time 500ms: Placed process F in memory:
=====
FFFFFFFFFFFFFFFFFFFFF.....
.....BBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEE
EEEEEE.....
=====
time 550ms: Process A arrived (requires 45 frames of physical memory)
time 550ms: Placed process A in memory:
=====
FFFFFFFFFFFFFFFFFFFFFAAAAAAAA
AAAAAAAAAAAAABBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEE
EEEEEEAAAAAAAAAAAAAAAAAAAAAAAAA.
=====
...
time ###ms: Simulator ended (Non-contiguous)
```

Virtual Memory Management

For this part of the simulation, you will focus on page references strings and page replacement algorithms. The second command-line argument to your simulation specifies a file that contains a page reference string for a given process. Open and read the specified file for this part of your simulation. The file contains a page reference string, using spaces to delimit references. A snippet of the sample file given on the course website is shown below. Also note that there is simple Java code on the course website for generating page reference strings.

```
7 3 3 3 3 3 3 3 1 1 8 3 3 3 3 3 3 3 3 6 6 6 6 6 6 6 7 7 7 7 7 3 3 3 3 3 3 3 3 3 8
8 8 8 8 8 8 5 5 5 5 2 2 2 2 2 2 2 2 2 2 7 7 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 5 5 5 1 2 2 5 5 5 5 7 7 7 7 3 3 3 3 3
...
```

In your simulation, the number of frames is initially set to F , which defaults to 3.

For the given page reference string (which we assume is associated with a single process), simulate the OPT, LRU, and LFU algorithms, each of which is described below.

The Optimal (OPT) algorithm is a forward-looking algorithm that selects the “victim” page by identifying the frame that will be accessed the longest time in the future (or not at all). If multiple pages are identified, this “tie” is broken by selecting the lowest-numbered page.

The Least-Recently Used (LRU) algorithm is a backward-looking algorithm that selects the “victim” page by identifying the frame that has the oldest access time.

The Least-Frequently Used (LFU) algorithm is a backward-looking algorithm that selects the “victim” page by identifying the frame with the lowest number of accesses. When a page fault occurs for a given page, its reference count is set (or reset) to 1; each subsequent access increments this reference count. If multiple pages are identified, this “tie” is broken by selecting the lowest-numbered page.

Simulation and Output Requirements

Append to the previous output your virtual memory results. In your output, show the fixed N -frame memory after each page reference, indicating page faults and corresponding victim pages. Sample output for the given page reference string is shown below.

```
Simulating OPT with fixed frame size of 3
referencing page 7 [mem: 7 . .] PAGE FAULT (no victim page)
referencing page 3 [mem: 7 3 .] PAGE FAULT (no victim page)
referencing page 3 [mem: 7 3 .]
referencing page 3 [mem: 7 3 .]
referencing page 3 [mem: 7 3 .]
referencing page 3 [mem: 7 3 .]
referencing page 3 [mem: 7 3 .]
referencing page 3 [mem: 7 3 .]
```

```

referencing page 1 [mem: 7 3 1] PAGE FAULT (no victim page)
referencing page 1 [mem: 7 3 1]
referencing page 8 [mem: 7 3 8] PAGE FAULT (victim page 1)
referencing page 3 [mem: 7 3 8]
referencing page 3 [mem: 7 3 8]
referencing page 3 [mem: 7 3 8]
referencing page 3 [mem: 7 3 8]
referencing page 3 [mem: 7 3 8]
referencing page 3 [mem: 7 3 8]
referencing page 3 [mem: 7 3 8]
referencing page 3 [mem: 7 3 8]
referencing page 6 [mem: 7 3 6] PAGE FAULT (victim page 8)
...
End of OPT simulation (### page faults)

Simulating LRU with fixed frame size of 3
referencing page 7 [mem: 7 . .] PAGE FAULT (no victim page)
...
End of LRU simulation (### page faults)

Simulating LFU with fixed frame size of 3
referencing page 7 [mem: 7 . .] PAGE FAULT (no victim page)
...
End of LFU simulation (### page faults)

```

Submission Instructions

We will use the Homework Submission Server (HSS) for this assignment.

If you are submitting a team project, please have each team member submit the same submission (just to be sure everyone gets a grade). Also be sure to include all names and RCS IDs in comments at the top of each source file.

Finally, please email csci4210@cs.lists.rpi.edu to let us know your team members. Provide a list of names and RCS IDs in your email.