CSC 343 Operating Systems, Spring 2020

Dr. Dale E. Parson, Assignment 4, modeling swapping using a FIFO queue and a min queue ordered on process logical memory size.
This assignment is due via **make turnitin** from the swap4csc343spring2020 directory by **11:59 PM on Friday May 1**.

I am handing out the STM code for a simulation that uses swapping; it places processes into a FIFO queue when they must wait to map their logical memory to physical memory via a relocation register. You must write an alternative implementation that uses a min-priority queue, with ordering based on logical memory size, to store the waiting processes. There is very little code to change for this project. The emphasis is more on analysis. You must read and understand the code to understand your assignment, and you must update a **README.txt** file that I have started.

Perform the following steps to get my handout. You will need to test on machine mcgonagall as previously explained (**ssh mcgonagall** from acad). I usually edit in one window on acad and test I another on mcgonagall, so I can run **make graphs** on acad after my program compiles on mcgonagall to generate one or more graphical image files for the project state machine(s). **DO NOT FORGET TO ANSWER THE QUESTIONS IN README.txt – they are worth 30% of this assignment. Working code is worth 70% total.**

```
cd  $HOME              # or start out in your login directory
cd  ./OpSys
cp  ~parson/OpSys/swap4csc343spring2020.problem.zip  swap4csc343spring2020.problem.zip
unzip swap4csc343spring2020.problem.zip
cd ./swap4csc343spring2020
make testfifo          # This tests the handout code. It should work.
make testmin           # This tests your code. It will not work until you make your changes.
```

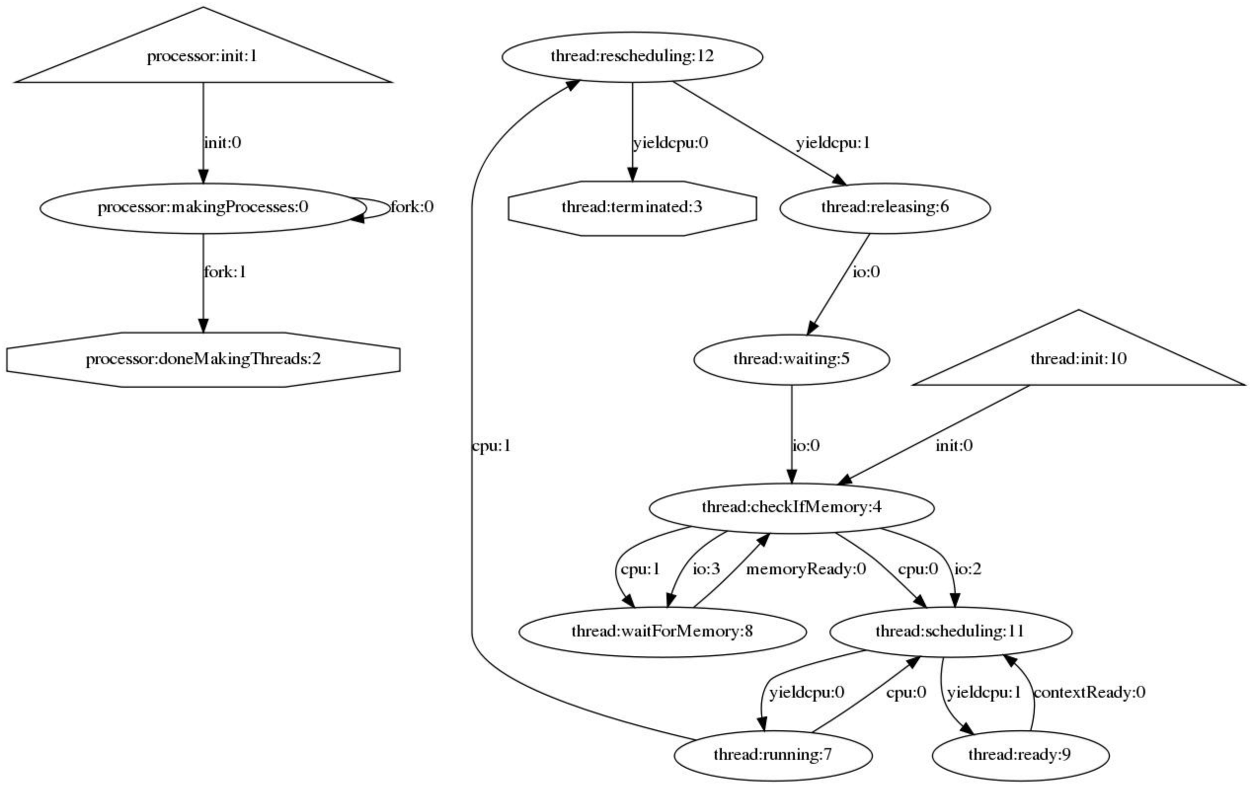
1. Understand the workings of the handout code (See STM.doc.txt for library documentation.)

Page 2 shows the JPEG graph for all of the STMs in this assignment. Your code changes must not change this graph topology. Here is the list of STM files handed out with the project, tested by **make testfifo**. All STMS in this assignment used round robin CPU scheduling.

```
rr_ff_swap.stm  # Locate memory for a process using first fit.
                # Activate (release from memory queue) only one process waiting for memory at a time.
rr_bf_swap.stm # Locate memory for a process using best fit.
allrr_ff_swap.stm # Locate memory for a process using first fit.
                  # Activate all processes waiting for memory at a time.
allrr_bf_swap.stm # Locate memory for a process using best fit.
                # Activate all processes waiting for memory at a time.
```

You will perform the following steps, and then edit the new "min" files with your code changes. **<u>DO NOT MODIFY THE HANDOUT .stm FILES LISTED ABOVE</u>**.

```
cp rr_ff_swap.stm rr_ff_min.stm
cp rr_bf_swap.stm rr_bf_min.stm
cp allrr_ff_swap.stm allrr_ff_min.stm
cp allrr_bf_swap.stm allrr_bf_min.stm
```



State machine graph for all STM files in assignment 4.

Here is a summary of the main transitions in the diagram. We will go over the code in class.

1. The **processor state machine** fork()s 24 single-threaded processes after initializing the following data fields and some unrelated fields.

# Two gig of available memory, initially all free, starts at phyaddr 0.
processor.physicalMemory = 2147483648 ;
# Each entry in this table is a (LIMIT, BASE) pair, with values giving
# (SIZE, START_ADDRESS) of a region in physical memory. As memory gets
# allocated and freed by processes, this initial region gets chopped
# up into multiple smaller regions and the merged where possible
# when freeing. See macros and lambda functions in the thread state
# machine for the code the manages the processor.freeregions and
# pcb.relocationRegister.
processor.freeregions = [(processor.physicalMemory, 0)];
# The waitMemQ is the Queue where processes wait until they get memory.
processor.waitMemQ = Queue(ispriority=False);

2. The **init -> checkIfMemory** transition initializes some memory management variables and performs the first memory allocation request for physical memory.

swapdevice = len(processor.fastio)-1 ;
# The range of process sizes drives contention for memory.

```
# Each process averages (1/16 + 1/2) / 2 = .28125 of physical memory,
# so we'd expect 3 to 4 processes to for into memory at one time,
# and never fewer than 2 will fit at one time. See processesToGo
# in the processor STM above.
pcb.logicalMemorySize = sample(int(processor.physicalMemory/16),
        int(processor.physicalMemory/2), 'uniform');
# pcb.relocationRegister will get a piece of processor.freeregions.
# Its value when memory is allocated is (SIZE, BASE), else None.
pcb.relocationRegister = None ;
```

3. All transitions into state **checkIfMemory** invoke macro **getFirstFit** (or **getBestFit** in best fit STMs) to attempt to allocate memory. When this macro succeeds, it sets variable **findcost** to the cost in ticks for locating the memory or trying to find memory, and it sets **pcb.relocationRegister** to the pair (SIZE, LOC), where SIZE is the number of locations in the process' logical memory, and LOC is the location in physical memory, i.e., the BASE ADDRESS. The **pcb.relocationRegister** is None (Python's null value) when insufficient memory is available; **findcost** is still valid in that case.

4. The transitions between **checkIfMemory** and **waitForMemory** illustrate the fact that, when there is insufficient memory to satisfy the memory request, transitions into **waitForMemory** enqueue the thread into the **processor.waitMemQ**, and transitions back to **checkIfMemory** make another attempt to allocate physical memory after being dequeued and awoken by a **memoryReady** event sent by another thread releasing its physical memory.

5. After acquiring memory, a thread advancing to state **scheduling** performs the round robin CPU scheduling algorithm discussed a few weeks ago.

6. The transitions into states **terminated** and **releasing** release their physical memory while terminating or preparing to perform application I/O, waking up one or more other processes waiting for memory. The *all* versions wake up all process threads waiting in **processor.waitMemQ** by sending a **memoryReady** event, while the non-all versions wake up only the thread at the front of **processor.waitMemQ**.

YOUR JOB

Your job after performing the cp copying steps at the bottom of page 1, thereby creating these four files, is as follows.

rr_ff_min.stm
bf_min.stm
allrr_ff_min.stm
allrr_bf_min.stm

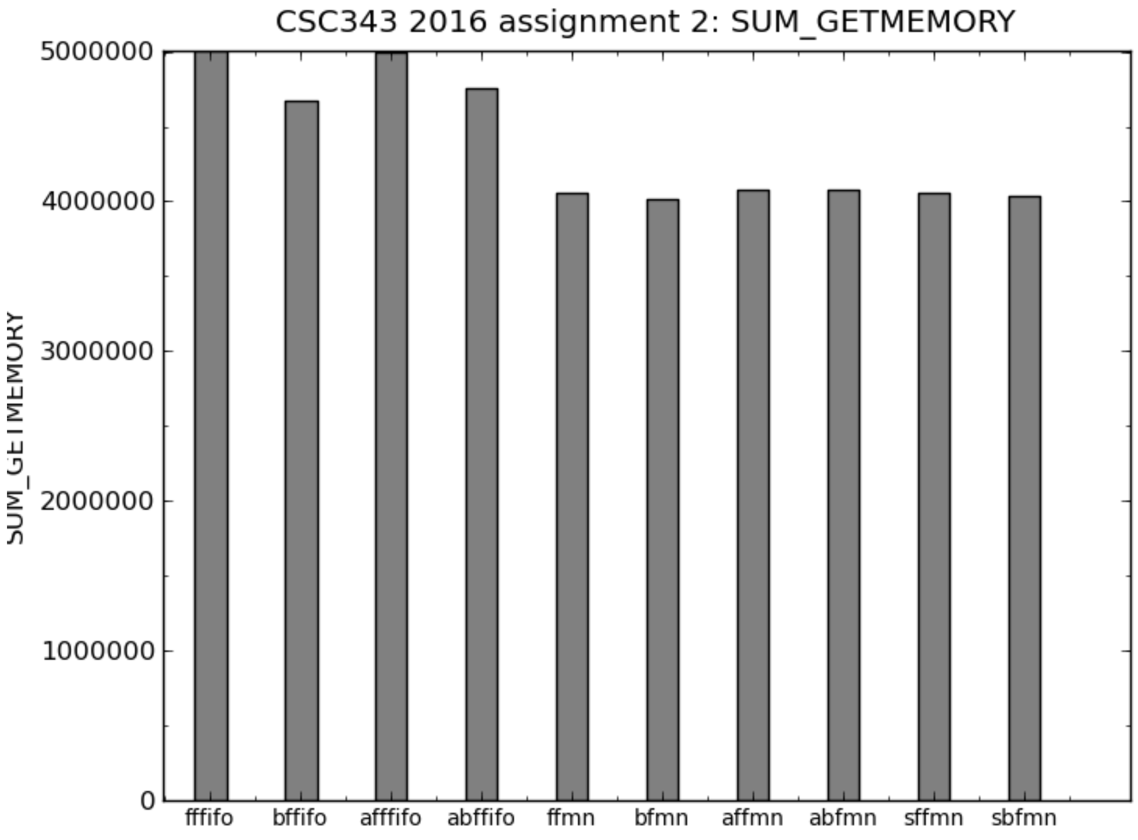
A. Change **processor.waitMemQ** from a FIFO queue to a priority queue. Python priority queues are min-queues, meaning that the minimum priority value goes to the front of the queue upon enqueuing. To see a min-queue, see field **processor.readyq** in ~~**/home/KUTZTOWN/parson/OpSys/state2codeV12/sjf.stm**~~. your assignment 3 solution's sjf.stm. There is a copy in /home/KUTZTOWN/parson/OpSys/stm3CPUschedSP2020/sjf.stm. Note that you are changing **processor.waitMemQ**, not **processor.readyq**, in this assignment.
B. Change all **processor.waitMemQ.enq** function calls to use the process logical memory size as the priority value.
C. At this point, **make testmin** should work**. make clean test** runs **make testmin testfifo**.

D. Update all of the files you edit correctly, adding your name, and describing the algorithm properly in the comments at the top of the file, including the file name.

You can individually run any of the following tests after you complete each min-STM source file:

make testrr_ff_min
make testrr_bf_min
make testallrr_ff_min
make testallrr_bf_min

Here is a bar graph that shows SUM_GETMEMORY, which is the combined sum of time spent in states checkIfMemory + waitForMemory for all 24 processes, with the "a" prefix signifying the "all" STMs, "fifo" meaning the FIFO xxx and "mn" the min-queue. Ignore the last 2 "s"-prefixed columns below from an experimental run. The "ff" prefix is first fit, and "bf" is best fit.



http://acad.kutztown.edu/~parson/SUM_GETMEMORY.png

See the actual, sorted values in distributed file **README.txt**.

Answer the questions found in the README.txt below each question.

The following instructions for tracing memory allocation & freeing should be helpful in answering the questions.

VIEWING A TRACE OF FIRST FIT AND BEST FIT MEMORY ALLOCATIONS AND DEALLOCATIONS

Invocations of the macro **takeMemory** for transferring memory from **processor.freeregions** to **pcb.relocationRegister**, and **putBackRegion** for transferring memory from **pcb.relocationRegister** to **processor.freeregions**, are already working in the handout code. Each prints a trace of **pcb.relocationRegister** and then **processor.freeregions** at both the start and end of each of these macros. Here are the **msg** invocations within these macros.

```
msg("DEBUG_TKM1|" + str(pcb.relocationRegister) + "|" + str(processor.freeregions) + "|");
# Code for takeMemory appears between the two msg statements.
msg("DEBUG_TKM2|" + str(pcb.relocationRegister) + "|" + str(processor.freeregions) + "|");

msg("DEBUG_PBR1|" + str(pcb.relocationRegister) + "|" + str(processor.freeregions) + "|");
# Code for putBackRegion appears between the two msg statements.
msg("DEBUG_PBR2|" + str(pcb.relocationRegister) + "|" + str(processor.freeregions) + "|");
```

After a successful test run, you can inspect the full trace of memory allocation and deallocation / compaction in the appropriate simulation log file. For example, for handout rr_ff_swap.stm you would inspect rr_ff_swap.log in the following ways.

```
grep 'DEBUG_[TP]' rr_ff_swap.log                # 1424 lines come to the screen
OR
grep 'DEBUG_[TP]' rr_ff_swap.log > junk.txt     # lines to temporary file junk.txt
```

Here are a few of the 1418 lines from the console, or from junk.txt when run with file redirection.

```
000000000001,MSG,thread 0 process 0,DEBUG_TKM1|None|[(2147483648, 0)]|
000000000001,MSG,thread 0 process 0,DEBUG_TKM2|(290113888, 0)|[(1857369760, 290113888)]|
000000000002,MSG,thread 0 process 1,DEBUG_TKM1|None|[(1857369760, 290113888)]|
000000000002,MSG,thread 0 process 1,DEBUG_TKM2|(525642137, 290113888)|[(1331727623, 815756025)]|
000000000003,MSG,thread 0 process 2,DEBUG_TKM1|None|[(1331727623, 815756025)]|
000000000003,MSG,thread 0 process 2,DEBUG_TKM2|(569390073, 815756025)|[(762337550, 1385146098)]|
000000000004,MSG,thread 0 process 3,DEBUG_TKM1|None|[(762337550, 1385146098)]|
000000000004,MSG,thread 0 process 3,DEBUG_TKM2|(466581130, 1385146098)|[(295756420, 1851727228)]|
000000000005,MSG,thread 0 process 4,DEBUG_TKM1|None|[(295756420, 1851727228)]|
000000000005,MSG,thread 0 process 4,DEBUG_TKM2|(180058597, 1851727228)|[(115697823, 2031785825)]|
000000000006,MSG,thread 0 process 5,DEBUG_TKM1|None|[(115697823, 2031785825)]|
000000000006,MSG,thread 0 process 5,DEBUG_TKM2|None|[(115697823, 2031785825)]|
000000000007,MSG,thread 0 process 6,DEBUG_TKM1|None|[(115697823, 2031785825)]|
000000000007,MSG,thread 0 process 6,DEBUG_TKM2|None|[(115697823, 2031785825)]|
000000000008,MSG,thread 0 process 7,DEBUG_TKM1|None|[(115697823, 2031785825)]|
000000000008,MSG,thread 0 process 7,DEBUG_TKM2|None|[(115697823, 2031785825)]|
000000000009,MSG,thread 0 process 8,DEBUG_TKM1|None|[(115697823, 2031785825)]|
000000000009,MSG,thread 0 process 8,DEBUG_TKM2|None|[(115697823, 2031785825)]|
```

Simulation time 1 shows process 0 with an unmapped **pcb.relocationRegister** at the start of **takeMemory** (DEBUG_TKM1), and with 290,113,888 elements in **pcb.relocationRegister** at the end of **takeMemory** (DEBUG_TKM2). Both **pcb.relocationRegister** and **processor.freeregions** show memory regions as (SIZE, BASEADDRESS) pairs.

Process 5 through 23 are not able to satisfy their needs for contiguous memory from the 115,697,823 locations still available in **processor.freeregions**, so their **DEBUG_TKM2** lines show **None** for a failed allocation in **pcb.relocationRegister**. Those processes must then wait in the **processor.waitMemQ** in state **waitForMemory** until released by a process releasing memory going into state releasing or terminated. Inspecting the full .log files shows the entry into state **waitForMemory** for threads blocking within the **processor.waitMemQ**.

At time 47 in the trace, process 1 releases sufficient memory for process 5 to proceed. Recall that process 5 was the first process to fail to acquire memory at time 6, so it is at the front of the FIFO **processor.waitMemQ**.

```
000000000047,MSG,thread 0 process 1,DEBUG_PBR1|(525642137, 290113888)|[(115697823, 2031785825)]|
000000000047,MSG,thread 0 process 1,DEBUG_PBR2|None|[(525642137, 290113888), (115697823, 2031785825)]|
000000000047,MSG,thread 0 process 5,DEBUG_TKM1|None|[(525642137, 290113888), (115697823, 2031785825)]|
000000000047,MSG,thread 0 process 5,DEBUG_TKM2|(192721691, 290113888)|[(332920446, 482835579), (115697823, 2031785825)]|
```

You can create slightly shorter lines in your DEBUG trace with this Unix command.

grep 'DEBUG_[TP]' rr_ff_swap.log | sed -e 's/MSG.*thread 0 //'
OR
grep 'DEBUG_[TP]' rr_ff_swap.log | sed -e 's/MSG.*thread 0 //' > junk.txt

The **sed** command is a stream editor, and the command in single quotes removes all characters from MSG through thread 0 in the output from **grep**. Make sure to put a space on each side of the 0, and type your single quotes; do not copy & paste quotes from Word documents. This output appears as follows.

```
000000000001,process 0,DEBUG_TKM1|None|[(2147483648, 0)]|
000000000001,process 0,DEBUG_TKM2|(290113888, 0)|[(1857369760, 290113888)]|
000000000002,process 1,DEBUG_TKM1|None|[(1857369760, 290113888)]|
000000000002,process 1,DEBUG_TKM2|(525642137, 290113888)|[(1331727623, 815756025)]|
000000000003,process 2,DEBUG_TKM1|None|[(1331727623, 815756025)]|
000000000003,process 2,DEBUG_TKM2|(569390073, 815756025)|[(762337550, 1385146098)]|
```

Do not try to wade through over 1000 lines of this trace output. When considering answers for the questions in **README.txt**, first think about the effects of queuing (FIFO versus min-queue on process logical memory size) and memory allocation algorithm (first fit versus best fit); you can optionally look at the trace to confirm your suspicions.

Think about the combined effects of FIFO versus min-queuing, and first fit versus best fit, and formulate each answer for README.txt. Use the memory trace as needed, but don't get bogged down in it. If I give hints, it will be in class. Plan to be there.
After **make clean test** passes and you have reread all instructions and written your answers in README.txt, run **make turnitin** on mcgonagall or **make turnitin** on acad before the project deadline.

If you get a time-time error in terms of the __codeTable__, run

/usr/bin/python decode.py STMFILE.py codeTableLine

as before, where STMFILE.py has the same first name as your STM file (e.g., rr_bf_swap.py), and codeTableLine is the compiled code line number in the run-time error message.