1. [20 points] A byte-addressable segmentation system has 46-bit virtual address, 32-bit physical address, and 16-bit segment number. A segment’s size can be any number of bytes up to its maximum. A segment in physical memory always starts at a 4 KB-aligned address (i.e., the least significant 12 bits are zero). Each segment table entry includes 8 bits for access and usage.

   a. Draw the segment table for a process. Give the number of rows in the table and the label and size of each field.

   **Solution** [11 pt]

   Virtual address (46-bit): 16-bit segment number | 30-bit offset
   hence max segment size = max value of offset = $2^{30}$ bytes

   Physical address (32-bit): 32-bit
   A segment’s physical base address has its 12 least significant bits zero, so $32 - 12 (= 20)$ bits suffice for segment base address.

   Segment table has $2^{16}$ entries (since segment number is 16 bits) [1 pt]

<table>
<thead>
<tr>
<th>valid (1)</th>
<th>base addr (20)</th>
<th>size (30)</th>
<th>access-usage (8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1 pt]</td>
<td>[2 pt]</td>
<td>[2 pt]</td>
<td>[1 pt]</td>
</tr>
<tr>
<td>;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
</tbody>
</table>

   [−1 pt] for having a “segment number” column.

   **End of solution**

   b. The hardware has a TLB of 6 entries managed with LRU replacement. Draw the TLB, showing its fields and their sizes. Indicate which part of the TLB is associatively searched.

   **Solution** [9 pt]

   TLB needs a 3-bit LRU field to maintain the usage order of its 6 entries.

<table>
<thead>
<tr>
<th>valid (1)</th>
<th>seg # (16)</th>
<th>base addr (20)</th>
<th>size (30)</th>
<th>access-usage (8)</th>
<th>LRU (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1 pt]</td>
<td>[2 pt]</td>
<td>[1 pt]</td>
<td>[1 pt]</td>
<td>[1 pt]</td>
<td>[2 pt]</td>
</tr>
<tr>
<td>;</td>
<td>;</td>
<td>;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
</tbody>
</table>

   valid and seg # fields are associatively searched [2 pt]

   **End of solution**
2. [10 points] A process with 3 physical pages initially empty issues the following string of virtual page references. What is the smallest possible number of page faults. Justify your answer.

Solution [11 pt]

The optimal policy yields the smallest number of page faults [2 pt]
The optimal policy is to replace the page that is used farthest in the future. [3 pt]

Applying the optimal policy yields the following, with 9 page faults [5 pt]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual pages in memory</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>pages in memory</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>faults</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

[5 pt] for using LRU correctly.

End of solution

3. [10 points] A demand-paging system uses page-fault frequency to adjust the physical page allocation and swap state of processes. Specifically, each pcb has a variable \( x \) that is zero when the pcb is created or swapped in, and is incremented by 1 at each page fault of its process.

A thread periodically reads and zeros the \( x \) values of all swapped-in processes and then “adjusts” the allocations and swap state. The goal is to keep the \( x \) values it reads close to 20.

Give an appropriate “adjustment” rule (i.e., that selects processes and changes their allocation or swap state)

Solution [9 pt]

If there are (many) free pages,
- allocate a page to any process that needs it (\( x > 0 \)) [3 pt]

If there are no (or hardly any) free pages, do one of the following:

- if all (or most) processes have \( x > 20 \), swap out a process with the highest (or high) \( x \) [1 pt]
- if all (or most) processes have \( x < 20 \), swap in a process (if available) [1 pt]
- otherwise, take pages from low-\( x \) processes and given them to high-\( x \) processes

This requires two scans:
- the first scan to quantify the spread [3 pt]
- the second to adjust the allocation [2 pt]

For example
- sort the process ids by \( x \);
  - then take from the head and give to the tail
- compute average \( m \) and std deviation \( s \) of the \( x \)’s;
  - then move pages from processes with \( x < m - 2s \) to processes with \( x > m + 2s \)

End of solution