Referencing string 4,3,2,1,4,3,5,4,3,2,1,5. Assume there are 3 or 4 page frames of physical memory. Show the memory allocation state after each memory reference. number of page faults for mem size of 3 or 4 with these 3 scheduling algorithms

	3	4
LRU	10	8
FIFO	9	10
Min	7	6

Contents of memory (under line is pagefaults for 4, italicize for 3)

LRU

4 3 2 1 4 3 5 4 3 2 1 5

---

4 3 2 1 ***4 3*** 5 4 3 2 1 5

4 3 2 1 4 3 5 4 3 2 1

4 3 2 1 4 3 5 4 3 2

4 3 2 1 4 1 5 4 3

For memory of size 3, erase the bottom row and redo the underlining/circling, All the original page faults are still there

FIFO size 4

4 3 2 1 4 3 5 4 3 2 1 5

---

4 4 4 4 4 4 5 5 5 1 1

3 3 3 3 3 3 4 4 4 4 5

2 2 2 2 2 2 3 3 3 3

1 1 1 1 1 1 2 2 2

FIFO size 3

4 3 2 1 4 3 5 4 3 2 1 5

---

4 4 4 1 1 1 5 5 5 5 5

3 3 3 4 4 4 4 2 2 2

2 2 2 3 3 3 3 1 1

Min size 4

4 3 2 1 4 3 5 4 3 2 1 5

---

4 3 2 1 4 3 5 4 3 2 1 5

4 4 4 1 4 4 5 5 5 5 1

3 3 3 1 3 3 4 3 2 2

2 2 2 2 2 2 4 3 3

at 2, 4 is used before 3, so 4 is in front of 3. Basically, it compares with future history to see who is being used at which time to determine what is at a higher ranking.

The page we throw out is the one that will be used least recently in the future.

Stack algorithm - obeys the inclusion property - the set of pages in memory of size N at time t is always a subset of the set of ages in memory of size N+1 at time t. Obviously cannot have miss ratio increasing with memory size. LRU and min are stack algorithms. Stack is a list of pages in order of size of memory which includes them. (Not actually a stack data-structure).

Implementing LRU: need some form of HW support in order to keep track of which pages have been used recently.

Perfect LRU? Keep a register for each page, and store the system clock into that register on each memory reference. To replace a page, scan through all of them to find the one with the oldest clock. This is expensive if there are lots of memory pages.

Use linked list to maintain LRU stack.

Nobody uses perfect LRU (except CDC-Star). We settle for an approximation that is efficient, just find an old page, not necessarily the oldest.

LRU is just an approximation anyway, so why not approximate a little more.

Clock algorithm: keep a "use" bit for each page frame, hardware sets the bit for the referenced page on every memory reference. Have a pointer pointing to the kth page frame. When a fault occurs, look at the use b it of the page being pointed to. If it is on, turn it off, increment the pointer, and repeat. If it is off, replace the page in the page frame set use(k) = 1. Also called FINUFO - first in, not used, first out. reference bit is also known as use bit.

In effect, the use bit, when used with the clock algorithm breaks the pages into 2 groups: those in use and those not in use. We want to replace the latter.

Some systems also use a dirty bit to give some extra preference to dirty pages. This is because it is more expensive to throw out dirty pages: clean ones need not be written to disk. Tradeoffs are:

- Cost of page fault declines - lower probability of writing out dirty page
- Probability of fault increases - i.e. if clock was a good algorithm and we mess with it, it should make it worse.

How would Least Frequently Used replacement work?
It would be a disaster, since locality changes.

A per process replacement algorithm or local page replacement algorithm or per job replacement algorithm allocates page frames to individual processes: a page fault in one process can only replace one of the process' frames. This relieves interference from other processes.

If all pages from all processes are lumped together by the replacement algorithm, then it is said to be a global replacement algorithm. Under this scheme, each

process competes with all of the other processes for page frames.
- If you are using a local replacement algorithm, you have partitioned memory among the jobs or processes.
- Local algorithm:
  - Protects jobs from others which are badly behaved.
  - Hard to decide how much space to allocate to each process
  - Allocation may be unreasonable
- Global algorithm:
  - Permits memory allocation for process to shift over time
  - Permits memory allocation to adapt to process needs
  - Permits badly behaved process to grab too much memory

Thrashing: a situation when the page fault rate is so high that the system spends most of its time either processing a page fault or waiting for a page to arrive.
- Thrashing means that there is too much page fetch idle time when - when the processor is idle waiting for a page to arrive.
- Suppose there are many users, and that between them their processes are making frequent references to 50 pages, but memory has 40 pages.
- Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out.
  - Compute average memory access time.

The system will spend all of its time reading and writing pages. It will be working very hard but not getting anything done.
  - The progress of the programs will make it look like the access time of memory is as slow as disk rather than disks being as fast as memory.

Plot of CPU utilization vs level of multiprogramming
- Thrashing was a sever problem in early demand paging systems.

Thrashing occurs because the system doesn't know when it has taken on more work than it can handle. LRU mechanisms order pages in terms of last process, but don't give absolute numbers indicating pages that musnt be thrown out.

Solutions to thrashing:
- If a single process is too large for memory, there is nothing the OS can do. The process will simply thrash.
- If the problem arises because the sum of several processes:
  - Figure out how much memory each process needs Change scheduling priorities if needed.