Project Specification – Phase 2
CS 3243/5243 – Operating Systems

In this phase you will develop an upgraded version of the operating system you implemented in Phase1. The assignment for this version of the operating system is given below.

Step I: You are to execute the same set of processes (30) given to you in the program-file using SJF scheduling algorithm. (Use job size to select the 'shortest' process under SJF.)

Step II: You are to execute the same set of processes using the same scheduling algorithms and introducing the memory management technique – paging, with blocking on I/O and page-faults.

Step I

As done in Phase 1, the processes from the *disk* would be moved into *memory* (or your simulated RAM), ordered by SJF policy, and then into the designated caches for execution. Under the SJF, the short-term scheduler will select the shortest process from the ready queue and then dispatch them to the CPU. Both the –CPU and N-CPU versions of the simulator are to be run for this Step 1.

What to expect/observe – data collection and analysis

Collect and analyze the performance metrics/data. Compare the data for the processes and discuss the effects of the three CPU scheduling algorithms (FIFO, Priority, and SJ).

Step II :

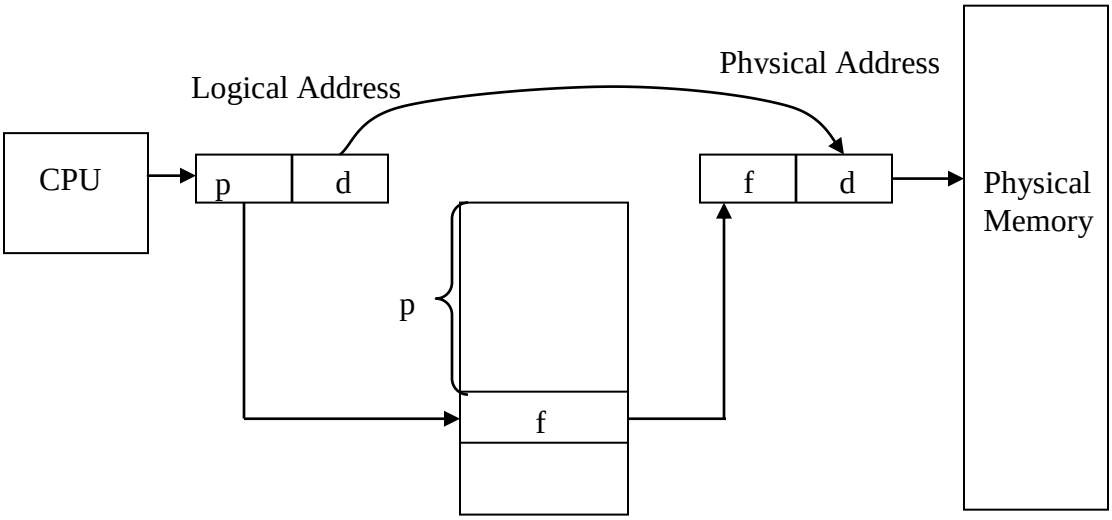
The second step involves the design and implementation of a paging system. This will require the generation of page fault with process-blocking and servicing of faults. This step will also include process-blocking due to I/O and servicing of I/O requests.

The simulated *memory* must now be broken into a pool of fixed-sized blocks called frames and the *disk* must also be broken into blocks of same sizes called pages. Thus, pages and frames are of same sizes: 4 words. Each process's cache is now reorganized as a set of caches: cache-frames for input-data, cache-frames for instructions, and cache-frames for output-data.

Each process will be initially assigned four (4) *memory* frames (4-words per frame), and any additional requests will be honored by a page-manager, depending on availability from the pool. When no more frames are available a process must wait until more frames are returned to the pool (by terminating processes). Remember that *memory* or RAM is still shared. Process instruction and data must still be loaded into the corresponding caches for the CPU to process them. [We are skipping page replacement implementation in Phase 2 but taking care of page fault servicing.]

Page table

You will have to implement a page table. The address generated by the CPU is divided into page number (p) and page offset (d). The page number is used as an index to the page table. The page table contains the base address of each page in physical memory. This base address is combined with the page–offset to define the physical memory address that is sent to the memory unit.



The page table is kept as a part of the PCB and a page-table base register (PTBR), which is also a component of the PCB, points to the page table. A valid-invalid bit should be associated with every entry in the page table. The status of the bit is used to determine if a page fault interrupt should be generated or not. Thus, access to a page marked invalid causes a page fault.

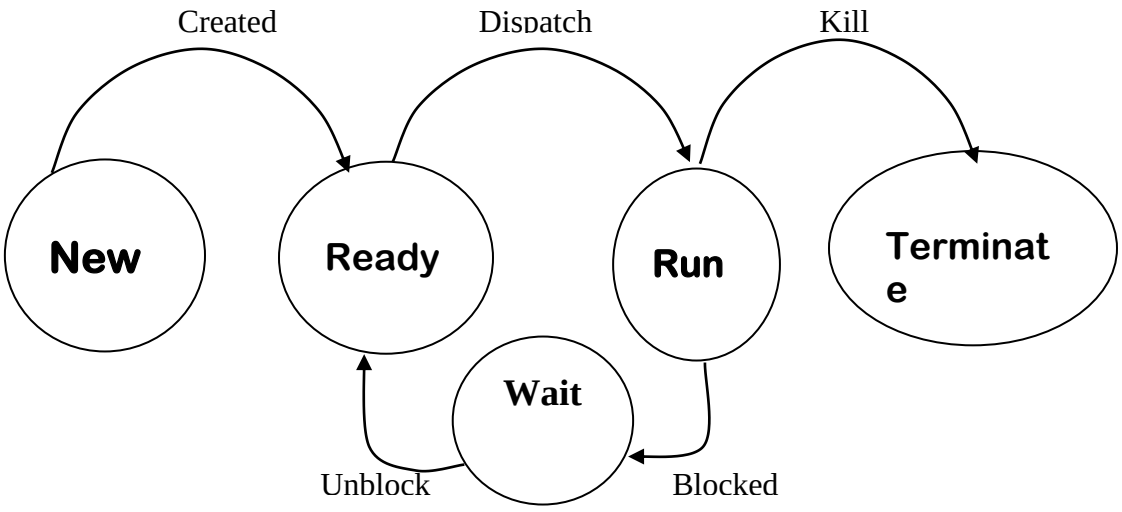
Three major components of the page fault service overhead are:

1. Service the page fault interrupt
2. Read in the page into the physical memory (and write a modified frame back to disk)
3. Restart the instruction that caused the fault

Things to take care of while servicing the page fault

1. Generate an interrupt when a page is not found in physical memory (or invalid bit is set)
2. Save the state and registers of the current process, which caused the interrupt into the PCB
3. Determine the interrupt type, i.e., if it were a page fault or an I/O request
4. Check if the page address is legal then determine the location of the page in *memory*
5. The process moves into the wait state, issues a read for the desired page from *memory* and waits until the page is transferred into a free frame (allocated from the pool)
6. While waiting, the CPU can be allocated to the next process according to the CPU scheduling algorithm. When a process moves into the wait state the appropriate PCB info must be changed or modified. [Note: Generate an interrupt to bring in a

new process and restore its PCB state]. The following state diagram depicts the logical flow of process transitions in the system.



7. Determine the interrupt type (I/O or page fault service completion)
8. Update the page table, showing that the page is now in *memory*
9. Change appropriate PCB info and add this faulting process to the ready queue

Context switching in this step will require

1. Interrupt generation
2. Swapping of pages, if required.

The *short-term loader* module must be written to support the updating and swapping of instructions/data between *memory* and the caches during context-switches or temporary blocking on interrupts.

Each CPU can be assigned any process such that when the process is blocked or context-switched, it may not be reassigned to the same CPU. Therefore, on context switch it is imperative to write back the modified words of the process into its cache – particularly, the output-buffer cache. At all times, the PCB must be updated to reflect the state of the process currently on the CPU.

What to expect/observe – data collection and analysis

You will study how paging or page faults affect the performance metrics/data for each process; and how interrupts (due to page fault + I/O) affect the same.

Collect and analyze data on all the performance metrics, including memory/page utilization, page faults, and fault service times (overhead) for the 30 jobs given the program/data file. Compare the metrics for the processes in Phase 1 with those in Phase 2, and discuss the effects due to the mix of different CPU scheduling algorithms, number of CPUs, and the blocking of jobs due to I/O and paging.