PAWAN LAWALE

Phase II (due March 25)

Before introducing multiprogramming, a memory manager has to be implemented. The multiprogramming system will run under a segmentation and paging memory management scheme. (It is advisable to review the memory management chapters/sections of your OS-I text book or any OS-I level text, especially if you have not passed OS-I or its equivalent recently).

In Phase II, only virtual addressing will be added to the SYSTEM. At this point your operating system will still be running under single stream batch, i.e., the system handles one job per batch (runs one job and stops). The only difference is that a job can be loaded anywhere in memory (each page of the process can be placed in any page frame of the memory) and the pages of a process are loaded dynamically.

The implementation of virtual memory will require changes to your MEMORY and LOADER routines. Each PAGE and each PAGE FRAME will be 8 words long. This means that the main memory will be subdivided into 32 page frames of 8 words each; and that each segment will be subdivided into pages of size 8 words. As expected, the last page of a segment may not be full, thus resulting in internal fragmentation.

PROCESS REPRESENTATION

Each job will consist of a Program Segment, an optional Input-Data Segment, and an Output-Data Segment.

Every job in the SYSTEM will now be represented by a Process Control Block or Process Context Block (PCB). For Phase II, since it is still single stream batch, there will be only a single PCB for each activation of your operating system. The PCB will contain at least the following information:

a) Job Id
b) Program Segment Information ("pointer" to page map table PMT)
c) Input-Data Segment Information ("pointer" to PMT and index into page)
d) Output-Data Segment Information ("pointer" to PMT and index into page)
e) Page Frame 1
f) Page Frame 2
g) Page Frame 3 (the number of frame entries depends on the initial
h) Page Frame 4 allocation size in terms of the number of frames)

Each job will have associated with it an identifier as soon as it "enters" the SYSTEM. The job identifier will be a number in the ascending sequence of integers, starting from one. Each segment will have an identifier composed of the job identifier plus a segment code. A possibility for segment codes would be:

0 --> Program Segment
1 --> Input-Data Segment
2 --> Output-Data Segment

VIRTUAL (Logical) ADDRESS to PHYSICAL (REAL) ADDRESS MAPPING

The virtual address in a user's program (i.e., a given test job) consists of a segment code (e.g., the program segment) and a displacement. The displacement is the address of the desired word in relation to the logical base address of the segment (zero). The translation of such a virtual address to an effective address will involve three levels of indirection as explained below.

The first level of indirection will translate the segment code into the address of the PMT for that segment via the Segment Map Table (SMT) which is included in the PCB (items b, c, and d). If the segment reference is to a new segment, a segment fault will be generated and a PMT has to be created for the new segment. The segment fault handler will be responsible for this.

The second level of indirection will map a page of the segment to the address of the page frame where this page is currently located. If the page referenced is not in the main memory, a page fault will be generated and the page fault handler will be activated.

The third level of indirection involves calculation of the effective address of the word currently being referenced. This calculation requires mapping the displacement in the segment to a displacement in the page frame.

SEGMENT FAULT HANDLER

A Segment Fault occurs when there is no PMT for a segment. Therefore, the Segment Fault Handler is responsible for creating a PMT for that segment and saving the address in the SMT (which is embedded in items b, c, and d of the PCB). In your SYSTEM, the loader will create the PMT for the program segment so that a Segment Fault will occur only with the first RD and with the first WR instructions of a program; that is, when the base address for the corresponding PMTs are nil (meaning that there are no PMTs for the Input-Data and the Output-Data Segments yet). The segment fault handling time is 5 virtual time units.

PAGE FAULT HANDLER

A page fault occurs when a referenced word is not in the main memory. The PFH has to find the page on the secondary store (DISK), determine the page to be replaced, transfer the pages (one from DISK to the memory and perhaps one from memory back to DISK), and update the corresponding tables. The page fault handling time is 10 virtual time units.

PAGE REPLACEMENT POLICY

The Replacement Algorithm to be used is the Second Chance algorithm or the Enhanced Second Chance algorithm (see: A. Silberschatz, P. B. Galvin, and G. Gagne, Operating System Concepts, John Wiley & Sons, Inc., New York, NY or other introductory Operating Systems texts). The replacement policy will be applied locally, that is, each job will choose a page to replace from within its own set of memory resident pages (see LOADING below). In each PMT of each segment of each user job, a Reference Bit (RB) associated with each page will be necessary for the implementation of the replacement algorithm.

The page chosen to be replaced may have been modified since it was transferred into memory, and therefore it may have to be copied back out to DISK. A "Dirty Bit" associated with each page, in the respective PMT, will be used to determine whether or not a transfer is needed.

INPUT SPOOLING

Every job has to be completely spooled onto DISK before any "loading" (i.e., before any transfer of pages from DISK to main memory and hence any execution) can take place. As a simplification point, we will assume that input spooling takes place instantaneously using no system time (i.e., it is done by a dedicated peripheral device or an independent channel).

As already specified above, a job consists of a Program Segment, an optional Input-Data Segment, and an Output-Data Segment. Whenever a job needs to get spooled onto DISK, space has to be reserved on the DISK for the Program Segment, the Input-Data Segment (if there is one), and the Output-Data Segment for that job.

The DISK must be simulated by an array, subdivided into page frames, analogous to the main memory. The DISK array should be eight times the size of memory. In the implementation of Phase II, you will be responsible for finding the available page frames on DISK and placing user job pages on the DISK array. This can be done basically by using techniques and procedures analogous to those that you will use to manage your memory array. Whenever a job terminates, its respective DISK storage frames will have to be released.

LOADING

The LOADER will be responsible for assigning memory page frames to a job and loading one page from the DISK into a memory page frame; which means that this module has to consult and/or update several tables.

Each job is allocated a fixed number of memory page frames at load time. This allocation remains fixed throughout the life of each user job. For our implementation, the number of page frames used for each job will be min(6, length of job in pages + 2). As soon as a job enters the SYSTEM, (1) these page frames will be allocated to it, (2) the program page containing the initial value of the PC will be loaded, (3) the remaining page frame base addresses will be held in the PCB, and (4) the job will be placed on the ready queue (which will be of size one for Phase II).

To assign memory page frames to a job, it is necessary to find free memory page frames. The data structure used to keep track of the status of each page frame will be called Free Memory Bit Vector (FMBV). The FMBV consists of 32 entries, one for each memory page frame; each entry is a bit to indicate the status of the respective frame. A page frame is in use if its status bit is set. The entry index will be used to calculate the base address of that page frame. For example, if the index into the FMBV is 7, the base address of the associated page frame will be 7*8 = 56.

EXECUTION

Each memory reference will have to be mapped from a Virtual Address to a Physical Address. This mapping takes place in the MEMORY module.

Whenever a segment is referenced (more precisely, when a word in a page in a segment is referenced) and that segment does not have a PMT, a segment fault occurs and the job has to release the CPU. The system will then create a PMT for that segment and load the referenced page.

Whenever a page is referenced and that page is not in the main memory, a page fault occurs and the job has to release the CPU.

Whenever an I/O instruction is executed, the job releases the CPU. Notice that the first input instruction and the first output instruction will cause a segment fault. Once a full page of the Input-Data Segment or the Output-Data Segment has been allocated, the next seven I/O instructions of the same kind will not result in a page fault. However, the job will have to release control of the CPU to the system and allow the system to perform the virtual I/O (in zero time).

OUTPUT SPOOLING

Whenever a job terminates, output spooling (i.e., outputting of the following information into appropriate virtual output devices, releasing the respective memory and DISK pages, and destroying the job's PCB) will take place. The information provided by output spooling is:

a) Job Id
b) Warning/Error messages (if any)
c) Input-Data Segment, labeled by the respective Job Id
d) Output-Data Segment, labeled by the respective Job Id (if no output was generated, your SYSTEM will print a message to that effect)
e) A message indicating the nature of termination (normal or abnormal, plus a descriptive error message if abnormal)
f) CLOCK value at termination (in HEX)
g) Run time for the job (in DECIMAL) subdivided into execution time, I/O time, page-fault handling time, and segment-fault handling time
h) memory utilization: number of words used over total number of words and number of frames used over total number of frames
i) DISK utilization: number of words used over total number of words and number of frames used over total number of frames
j) memory fragmentation: among the occupied frames, number of words unused in the last allocated page
k) DISK fragmentation: among the occupied frames, number of words unused in the last allocated page

As a simplification point, we will assume that output spooling takes place instantaneously and requires zero virtual time units (i.e., it is carried out by an independent channel or a dedicated peripheral device).

SUMMARY OF DATA STRUCTURES AND ALGORITHMS

a) Tables

   i) One: Free Memory Bit Vector (FMBV) (AKA the free frame vector)
   ii) One per active job: Segment Map Table (SMT) (embedded in the PCB)
   iii) One per segment: Page Map Table (PMT)

[in a real system, these are typically in memory/cache/TLB; here, we simply assert their existence]

b) Algorithms

i) Input Spooling
ii) Segment Fault Handler
iii) Page Fault Handler
iv) Replacement Algorithm
v) Virtual Address to Real Address Mapping
vi) Output Spooling

BATCH PACKET

All the information necessary to run a program is provided in the batch packet of each job.

```
**JOB xx yy
       (where
       xx - size of the Input-Data Segment in # of words and
       yy - size of the Output-Data Segment in # of words)
<loader format>
**INPUT
<data>
**FIN
```

Errors occurred during the interpretation of the JCL, the loading of the loader format, and the handling of the data will result in warnings or error traps with the possible termination of the respective job and subsequent output spooling.

Possible errors before spooling include, but are not limited to:

a) more than one **INPUT
b) missing **JOB
c) missing loader format
d) missing **INPUT
e) conflict between the # of input words specified in the **JOB line and the # of input items given in the INPUT section
f) missing **FIN

Possible run-time errors include, but are not limited to:

a) reading beyond the end of file (Input-Data Segment)
b) writing beyond the end of file (Output-Data Segment)

Spooling errors can be handled by initiating the output spooling of the current (incorrect) job and skipping of the records or lines by the reader until a **FIN or a **JOB is encountered.

TESTING

You will test your Phase II using an input file (very similar to Phase I). In your initialization routine, you would normally zero out the FMBV. However, for Phase II, like Phase I, we will still be initializing the operating system, running a single job, and then terminating the operating system. Hence, for the purpose of testing your Phase II only, change the initialization of your FMBV so that all the memory page frames except 5, 8, 10, 17, 20, and 31 are full.

NOTES:

The "NOTES" part of the specification for Phase I, with some notable differences as explained below, applies to Phase II also. What you are to turn in for Phase II is essentially the same as Phase I, except that there are now more possibilities for errors and warnings. So, in addition to the modifications outlined above in this document, your error-handler will have to be updated and augmented significantly. Your marked and graded Phase I project should be included with your Phase II deliverables.

INPUT: Each time that you run your simulation in this phase, the input to your Phase II (i.e., your SYSTEM) is a file of HEX digits and some JCL, which is a "user job" or "user program" in batch packet format. There will be no terminal input in Phase II. The "input" to the user job, if indeed input is needed, will appear after **INPUT in the batch packet format.

OUTPUT: There will be no screen output in Phase II. The output of a user job (the result of executing the WR instruction) will be placed in the Output-Data segment for that job. The system output file (a virtualized

output device) will contain, among other things (see Phase I output), the information placed there as a result of the output spooling process. Remember to tag all relevant output by adding the term HEX or DECIMAL, as appropriate. In addition to the system output file, there may be another output file called a trace_file generated as a result of the execution of a user job as well (if the trace bit of the user job is set).

YOU ARE TO TURN IN THE FOLLOWING:

- Your phase two simulation listing (modular, readable, and well-documented).

- Your own non-trivial test job in the given assembly language format consisting of English explanation and assembly code (with in-line comments as necessary) followed by its batch packet.
  [non-trivial means with at least as much complexity and number of statements as the test job given in the specification]

- Sample compilation and executions of the test job provided with the trace switch on and with the trace switch off, along with a printout of some of the trace_file.

- Sample compilation and executions of your test job with the trace switch on and with the trace switch off, along with a printout of some of the trace_file.

- Sample executions (plus their assembly and load module) of a number of your own small test jobs (or modified versions of the above-mentioned sample test jobs) containing injected errors including some of the errors specific to Phase II in addition to overflow, suspected infinite loop, invalid loader format character, invalid op code, attempt to divide by zero, address out of range, bad trace flag, etc., in order to exercise your ERROR-HANDLER routine; the trace switch should be off for the test jobs used in these runs.

- A two to three page write-up of the software engineering issues involved in the design and development of your Phase II simulation that must include the following items.

  - Your general approach to the problem, i.e., whether or not you used a design language, whether or not you used pseudo-code, whether or not you used a flow chart, etc. Note that this DOES NOT mean that a flow chart, etc. of your design is required for Phase II; if you need to include a flow chart, etc., it belongs in the external documentation part of your simulation code and not in the software engineering issues. This is a simple question with at most a one-line answer.

  - A list of the utilities used (e.g., makefile, various debuggers, SCCS, RCS, etc.).

  - Bulk complexities of your simulation program including the following items:
    = the total number of lines of code as well as its subdivision into the number of declarations, comment lines, executable statements, blank lines, etc.
    = the number of decisions
    = the number of procedures
    = the number of classes, etc. as appropriate and applicable to your implementation language

  - An approximate break-down of the time spent in the design, coding, and testing of your program.

  - Your comment on the portability of your simulation to other operating systems, architectures, etc.

  - A brief justification of your choice for the implementation language.

Batch packet for the sample encryption job:

```
**JOB 01 02
01 00 0B 44 1
D82DBC04942ADC2E
000AD82EB42BC002
DC2EDC2F00150013
```

```
DC2DD82DB029C424
8830D834DC2CC800
D82F8831942CDC2C
D82D883200038832
942CDC2CD82D8833
00038833942CBC26
C800D82FD82D1414
0018000000010004
0000000000010000
000F00F00F003000
000A000300060008
000B0002000D000F
000100040005000E
000C000700090000
**INPUT
0014
**FIN
```