Computer Science & Engineering Department
I. I. T. Kharagpur

Operating System: CS33007

3rd Year CSE: 5th Semester (*Autumn 2006 - 2007*)
Lecture XIII (**Memory Management**)

Goutam Biswas *Date:* 2006

Memory Management: Virtual Memory

1. In an ordinary paged memory management, the process image is broken into pages and are mapped to different page frames of the main memory. The whole image is loaded for execution.

2. Loading all the pages of a large process results an under utilization of the main memory. Most of the pages are not accessed most of the time due to the small size of working set.

3. OS can load larger number of processes if all the pages of every process are not loaded in the main memory. Larger number of users can get better response from the system.

4. A portion of the disk can be treated as a part of the process space (virtual memory). Part of the memory image of a process can be kept in this portion of the disk, known as the *swap area*.

5. Only the recently used data and text pages are kept in the main memory. Other pages are loaded on *demand*. Some pages may be loaded in anticipation.

6. If no page frame is free, some of the old pages are to be swapped out of the main memory to the *swap area* to accommodate incoming pages.

7. The disk access is several order of magnitude slower than the main memory access. So loading a page from the disk to the main memory is costly process.

8. The system works due to the locality of references. New pages are not required very often. The working set of a process changes slowly.

9. If the page is not present in the main memory -

   (a) A *page* or some of the *page tables* (in a 2 or 3 level paging system) may not be present in the main memory.

   (b) This can be indicated by a *present bit* of the page table or the page directory entry.

   (c) If the virtual address (logical address) generated by an instruction or data refers a page that is either *invalid for access* or $\mathring{3}$ absent from the main memory, an *internal exception* called *page fault* is generated.

10. Page fault -

    (a) A page fault is an *internal* and *non-mask-able exception*.

(b) It may occur to fetch an instruction or to fetch a data.

(c) Possible *points of occurrences* of page faults are fixed in the sequence of micro-operations of an instruction.

(d) The execution of the offending instruction cannot be completed unless the missing page is made available in the main memory (for a valid page).

(e) The control of execution is transfered to the OS (by the hardware) after a page fault and the CPU mode is switched to privileged.

(f) The OS handles the fault. It may have to load the missing page from the disk to the main memory. The process page table (directory) is updated, the mode of the CPU is switched to *non-privileged*, and the offending instruction is *restarted*.

(g) The process is transparent to all this activity. It is blocked if a page is to be read from the disk.

11. Cost of demand paging -

    (a) A machine can execute 100 MIPS if all pages are in the main memory.

    (b) The time taken to load one page (from the disk to the main memory) and restart a process be 10 ms.

    (c) Performance degradation -

One PF per n instructions	MIPS
1000 000	50
100 000	9
10 000	1

12. Virtual memory, swap area and file system -

    (a) The image of a process is often split between the *memory*, the *swap area*, and the *executable file*.

    (b) The read-only part of the image e.g. the text may be linked to the corresponding disk sectors of the executable file under the file system. This is known as file mapping.

    (c) The pages corresponding to the data and stack may be kept in the *swap area* (properly initialized).

13. Memory mapping of a file -

    (a) A file can be mapped to a memory region of a process. in Linux the system calls are `mmap()` and `munmap()`.

    (b) Any access to the mapped memory region is translated by the OS to an access to the file.

```
void  *  mmap(void *start, size_t length, int prot , int flags, int fd,
       off_t offset);
int munmap(void *start, size_t length);
```

```
/*************************************
 * Memory mapped file for read         *
 * cc -Wall mmap1.c                    *
 * The file name is 'mmapFile'         *
 * ***********************************/
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>

int main() {
     int fd, i = 0 ;
     char *cp ;

     fd = open("mmapFile", O_RDONLY, 0) ;
     cp = (char *) mmap(0, 100, PROT_READ, MAP_PRIVATE, fd, 0);
     printf("%p\n", cp) ;
     while(cp[i] != '\n') printf("%c", cp[i++]) ;
      printf("\n");
     munmap(cp,100);

     return 0 ;
}
```

14. Swap area -

    (a) The allocation and management of the swap area is different from the file system. The allocation is contiguous to make the I/O faster. Objects in a swap area are transient in nature.

    (b) The size of the swap area is almost 2 to 3 times larger than the main memory.

    (c) No new process can be created if the swap area is full.

15. Locating page in the swap area -

    (a) Consider a page table of size 4K Bytes, where each Entry is of size 32-bits.

    (b) Let the *least significant bit* (t_0) be the *presence bit*. If the page is *present*, ($t_0 = 1$), bit $t_{31} - t_1$ have their usual meaning (page base address, protection etc.).

    (c) If the page is *not present* i.e. $t_0 = 0$, bits $t_{31} - t_{12}$ can be used to store the *address* (*block number*) of the page in the *swap area*.

    (d) With 20-bits the OS can access 1M pages in the swap area.

16. Creation of Process Image -

    (a) The sizes of some portions of the virtual space e.g. the code and the global data are available from the executable module.

    (b) The OS decides the *initial sizes* of the *stack* and other areas.

    (c) The writable part of the process image is created and initialized in the swap area.

(d) In Linux, the address space of a process is a list of virtual memory regions. Each region starts from a page boundary and its size is an integral multiple of pages.

(e) The virtual space or the regions ares divided into pages. The page tables are created and initialized in the main memory.

(f) In a two or three level paging, only the page directory of the process is always present in the main memory.

(g) All pages are marked *absent* (in the main memory). Their *disk block addresses* and the *protection bits* are written in the page table (directory).

(h) The *base address* of the *page directory*, *memory region list* and other information are saved in *process control block*.

(i) The process is ready to be dispatched.

17. Dispatch a Process -

    (a) The *page directory base register* (PDBR) e.g. **cr3** in case of Pentium is initialized with the *base address* of the page directory.

    (b) The *TLB* entries are invalidated.

    (c) The control is transfered to the process.

18. Process creation in Linux -

    (a) A process in Linux is created by `fork()` system call.

    (b) A 'copy' of the parent process is created as a child process.

    (c) The address space of the parent process is logically duplicated for the child process.

    (d) But the actual copy of the memory frames are not done until one of the processes (either the parent or the child) 'writes' in it.

    Pages are 'shared' between the parent and the child. They are write protected. When one of the processes tries to write in a page, an *exception* is generated and the the page is copied to a new frame (COW). The page is made writable.

    (e) The *execution context of a process* can be replaced by system calls known as *exec calls*.

19. Page fault revisited -

    (a) A *page fault* may be caused by different reasons. The state of the CPU is saved (architectural support) and is made available to the OS.

    (b) A page fault may occur due to (i) absence of the referenced page in the memory, or due to (ii) violation of access rights.

    (c) In Pentium the address of the offending instruction is saved in the control register `cr2`.

    (d) The OS checks whether the fault address is within the address space of the process.

    (e) In case of a page fault generated by a stack instruction e.g push, the address may not be within the valid memory region and the system may expand the stack space.

    (f) If the fault address is outside the address space of the process, an memory violation exception signal is generated.

(g) There may be a page fault on write access to a write protected page. A page with a privileged access right may give page fault even on read.

(h) Two types of page faults -

- *Minor Page Fault*: There is no disk access and the process is not blocked - allocate a page frame for a valid region and initialize it or expand the region for stack and allocate a page frame.
- *Major Page Fault*: There is a disk access and the process is blocked. A page may be copied from the executable file or the swap device.

(i) A *valid page* may not be in the main memory mainly due to two reasons -

- The page has not been addressed so far.
- The page has been swapped out.

In both the cases the process is to be blocked to load the page from the disk.

20. Restarting a process -

(a) Restarting the process after a page fault needs help from the architecture (it is more complicated in a pipelined and out of order execution architecture. The page fault may take place out of order.).

(b) There should be support in the CPU-MMU architecture to roll back to the state of the CPU before the execution of the offending instruction.

(c) Even on an architecture without instruction level parallelism (ILP), roll back is difficult for *instructions with side-effect*.

(d) `mov -(R3), +(R2)`: the semantics is

- Instruction fetch # Page-fault possible (1)
- R2 = R2 + 4
- temp = Memory[R2] # Page-fault possible (2)
- R3 = R3 - 4
- Memory[R3] = temp # Page-fault possible (3)

(e) Not only the PC and the PSW but also R2, R3 are to be in their old values before the restart.

(f) Even for a sequence of simple instructions in a classic 5-stage pipeline architecture,

```
load R1, 100(R2)  # R1 <-- Mem[R2 + 100]
add R2, R3, R4    # R2 <-- R3 + R4
```

there may be an *instruction fetch page-fault* in the 2nd instruction before the *data memory access page-fault* in the 1st instruction. The architecture provide support to handle these situations.

21. Page replacement -

(a) It is possible that there is a page-fault, a new page is to be brought in, but no page frame is free.

(b) A page frame is to be freed by swapping out a page from the main memory.

(c) The question is, how to select the frame and what is to be done with the content of the frame.

(d) If the page is *read-only*, nothing much is to be done. Only the corresponding page table and TLB entries are to be invalidated.

(e) If the page is *writable* but the *dirty bit* is not set, the actions are similar to that of a *read-only* page.

(f) But if the *dirty bit* is set, then the *swap area* of the disk contains an *old copy* of the page and the page is to be written back.

(g) There are different algorithms to select page frames for replacement.

22. Measure of goodness of a page replacement algorithm - given a sequence of page references, a page replacement algorithm A_1 is said to be better on the given sequence, than another algorithm A_2, if the number of page faults generated by A_1 is less than that of A_2.

23. Optimal page replacement algorithm - A replacement algorithm is said to be *optimal* if it always replaces a page that will not be referenced in future before any other page present in the page frames.

    (a) The sequence of future page references is difficult to predict. In any computing system, different processes are running concurrently, and the mix changes over time.

    (b) The optimal replacement sequence of pages may be used as a bench-mark to measure the goodness of more realistic algorithms. This gives the upper bound of performance i.e. nothing can be better than this.

24. Random page replacement algorithm - a page frame is selected at random, it is swapped out if it is permitted.

    (a) It is simple to implement and does not assume anything about the page reference pattern.

    (b) It may be considered as the *lower bound* of performance in page replacement. If the performance of any other algorithm A is worse than the random replacement, then the assumption about page references behind A is wrong.

25. FIFO page replacement algorithm - OS maintains a queue of page frames.

    (a) The queue is simply a counter that remembers the number of the oldest page frame loaded.

    (b) The counter starts from the first page frame number, say zero (0).

    (c) When a page is to be replaced, the oldest frame number is taken from the counter and the counter is incremented by one (modulo the total number of frames).

    (d) It does not perform well if the older frames contain recently referenced page.

Page Frame	Page References							
	r_0	r_1	r_2	r_3	r_0	r_4	r_0	$\cdots$
0	r_0	r_0	r_0	r_0	$\mathbf{r_0}$	$\mathbf{r_4}$	r_4	$\cdots$
1		r_1	r_1	r_1	r_1	r_1	r_0	$\cdots$
2			r_2	r_2	r_2	r_2	r_2	$\cdots$
3				r_3	r_3	r_3	r_3	$\cdots$
Counter	0	0	0	0	0	1	2	$\cdots$

26. Least recently used (LRU) page replacement algorithm -
    (a) The pages that are not being referenced in recent past are likely not to be referenced soon in future.
    (b) It is a good theory but the algorithm is costly to implement.
    (c) It is necessary to maintain the *list of ages* of the pages present in different page frames.
    (d) It can be maintained as a list of frame numbers with a front and a rear pointers.
    (e) If a page is referenced, the corresponding frame number is inserted at the rear of the list. If the frame number is already present in the list, it is deleted from its earlier position.
    (f) If no page frame available for the requested page, the page frame from the front of the list is freed to accommodate it.
    (g) Every time a page is referenced, the LRU list is to be updated. For every actual memory reference, there should be a memory reference for house keeping. This is not acceptable.

27. Approximation of LRU - Reference bit -
    (a) Each entry in the page table as well as each entry of the TLB has a bit called the reference bit. It is set whenever a page is accessed.
    (b) It can be cleared by programming.
    (c) On a page fault, the OS searches page tables for an entry where the reference bit is 0. If such an entry is found, the content page is replaced if necessary, the entry is made invalid and the frame is used for the incoming page.
    (d) The page table entry is updated for the new page and its reference bit is set. It is also entered in the TLB (It may be necessary to replace a TLB entry).
    (e) A clock algorithm may be used for a round-robin selection.

28. Clock Algorithm -
    (a) Assume that the page frames are arranged in a circle.
    (b) A (modulo)-n counter, where n is the number of page frames, holds the number of a page-frame.
    (c) If the reference bit of this frame is 0, it will be replaced. Otherwise the OS resets the reference bit and increments the counter (modulo-n) to get the next frame number.

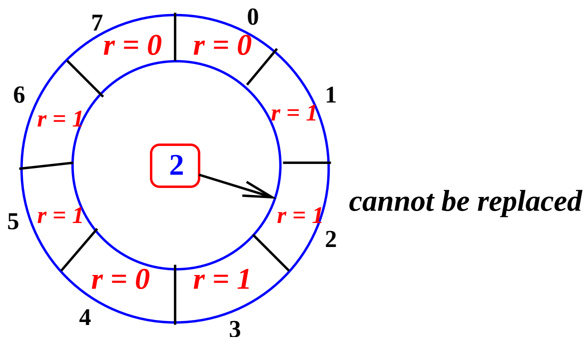

Figure 1: **Clock I**

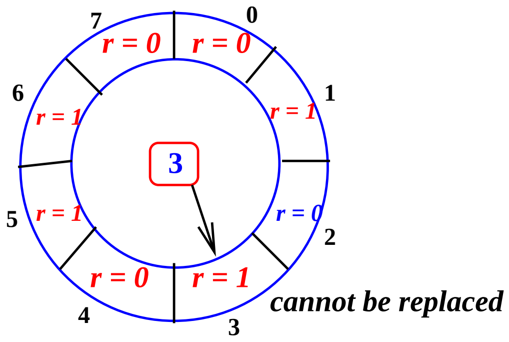

Figure 2: **Clock II**

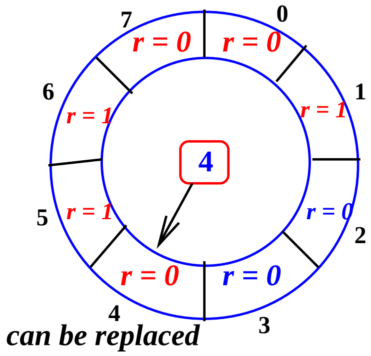

Figure 3: **Clock III**

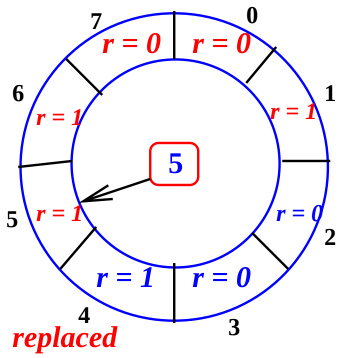

Figure 4: **Clock IV**

29. Age counter -

   (a) A software counter is associated with every page table entry. They are initialized to zero.
   (b) At an interval of say 1 second an *LRU demon* will wake up and test the reference bits of the page table entries.
   (c) If the page has been referenced in the last time slot, its counter is reset and other counters are incremented.
   (d) The page with the highest count (LRU) is replaced.

30. An example -

   (a) Optimal page replacement -

0	3	5	5	7	3	1	4	2	1	6	6	0	7
0^f	0	0	0	0	0	0	0	0	0	0	0	0	0
	3^f	3	3	3	3	3	4^f	2^f	2	6^f	6	6	6
		5^f	5	5	5	1^f	1	1	1	1	1	1	1
				7^f	7	7	7	7	7	7	7	7	7

   There are *8 page faults.*

   (b) LRU page replacement -

0	3	5	5	7	3	1	4	2	1	6	6	0	7
0^f	0	0	0	0	0	1^f	1	1	1	1	1	1	1
	3^f	3	3	3	3	3	3	3	3	6^f	6	6	6
		5^f	5	5	5	5	4^f	4	4	4	4	0^f	0
				7^f	7	7	7	2^f	2	2	2	2	7^f

   There are *10 page faults.*

   (c) Reference bit page replacement -

0	3	5	5	7	3	1	4	2	1	6	6	0	7
0_1^f	0_0	0_0	0_0	0_0	0_0	1_1^f	1_0	2_1^f	2_0	6_1^f	6_1	6_0	6_0
	3_1^f	3_0	3_0	3_0	3_1	3_0	4_1^f	4_0	1_1^f	1_0	1_0	0_1^f	0_1
		5_1^f	5_0	5_0	5_0	5_0	5_0	5_0	5_0	5_0	5_0	5_0	5_0
				7_1^f	7_1	7_0	7_0	7_0	7_0	7_0	7_0	7_0	7_1

   There are *10 page faults.*