5 UNIX Memory Management

In this chapter, we will focus on the memory management in UNIX, which is one of the most important services of UNIX kernel. In a computer system, CPU must cooperate with the memory to accomplish any computing. The main memory has scarce space and cannot contain all the programs on the disk. However, a process cannot execute if it is not brought in memory. Thus, the memory management becomes quite important, especially when the sizes of application programs become fairly large. And the memory management has a close relationship with the process management. We will introduce the outline of memory management, process swapping in UNIX, and demand paging in UNIX in this chapter.

5.1 Outline of Memory Management

With the development of computer hardware technology, the size of physical memory has been evolving from very small one (less than one half Mbytes, such as PDP-11) to quite large one (more than 1 Gbyte, such as HP xw4200). At the very beginning, operating system developers had to try very hard to figure out how to use the limited memory space to accomplish as many user applications as possible. To do so, many schemes to manage memory were addressed and among them, the swapping and paging were two of the most important ones and still work well in modern operating systems. The swapping mechanism was adopted in the UNIX System versions of AT&T Bell Lab at first (Ritchie et al 1974) while the paging technique was added to UNIX BSD variants of the University of California at Berkeley at the start (Quarterman et al 1985). With the size of physical memory increasing, the tension of memory space in the computer system, however, has not been eliminated. The reason is that the growing speed of the size of application programs is sometimes faster than the development rate of the physical memory.

Because the physical memory space is always limited and programs become larger and larger, memory must be carefully managed, and the memory management becomes more and more important. Typically, the memory man-

Y. Liu et al., *UNIX Operating System*

agement is responsible to allocate the portion of memory for new processes, keep track of which parts of memory are in use, deallocate parts of memory when they are unused, and manage swapping between main memory and disk and demand paging when main memory is not enough to hold all the processes.

5.1.1 Evolution of Memory Management

As in a single-process operating system only one process at a time can be running, there is just one program sharing the memory, except the operating system. The operating system may be located at the lower-addressed space of the memory and the user program at the rest part. Thus, the memory management is quite simple. That is, there is not too much work to do for the memory management in the single-process operating system. The memory management just handles how to load the program into the user memory space from the disk when a program is typed in by a user and leaves the process management to accomplish the program execution. When a new program name is typed in by the user after the first one finishes, the memory management also loads it into the same space and overwrite the first one.

In multiprocessing operating systems, there are many processes that represent different programs to execute simultaneously, which must be put in different areas of the memory. Multiprogramming increases the CPU utilization, but needs complex schemes to divide and manage the memory space for several processes in order to avoid the processes' interfering with each other when executing and make their execution just like single process executing in the system.

It may be the simplest scheme to divide the physical memory into several fixed areas with different sizes. When a task arrives, the memory management should allocate it the smallest area that is large enough to hold it and mark this area as used. When the task finishes, the management should deallocate the area and mark it as free for the later tasks. A data structure is necessary to hold the information for each size-fixed area, including its size, location and use state. Since a program has a starting address when it is executed and the initial addresses for various areas are different, the memory management should also have an address transformation mechanism to handle this issue. The two biggest disadvantages for this scheme are that the fixed sizes cannot meet the needs of the number increasing of the tasks brought in the system simultaneously and the size growing of application programs. The former problem can be handled by swapping and the latter one via paging.

Mentioned in last chapter, sometimes some processes wait for I/O devices in memory without doing anything. Otherwise, some other processes are ready to run, but there is not enough memory to hold them. Thus, operating system developers consider if the memory management can swap the

waiting-for-I/O processes out of the memory and put them in the disk temporarily to save the memory space for other processes that are ready to run. When the memory is available for the processes swapped out on the disk, the system checks which processes swapped out are ready to run and swap the ready processes in memory again. This memory management strategy is called swapping. The memory is allocated to processes dynamically. As the swapping is fast enough to let the user not to realize the delay and the system can handle more processes, the performance of the whole system becomes better. Since the program is kept in a continuous memory space, the swapping is just done on a whole process.

When the size of an application program becomes too big to load in the memory as a whole at a time to execute, the memory paging is needed. Paging technique can divide the main memory into small portions with the same size — pages, whose size can be 512 or 1024 bytes (Bach 2006; Belay et al 1969; McKusick et al 2005; Quarterman et al 1985; Rashied et al 1988; Stallings 1998). When a long program is executed, the addresses accessed over any short period of time are within an area around a locality. That is, only a number of pages of the process are necessarily loaded in main memory over a short period of time. When some page of the program is needed and not in memory yet, it is acceptable if that page is loaded in main memory fast enough when demanded, which is called demand paging. In this situation, bringing in or out of memory is with pages, rather than a whole process. Demand paging combined with page replacement and swapping implements the virtual memory management in 4.2BSD and UNIX System V (Bach 2006; Bartels et al 1999).

At the very beginning of UNIX development in the early 1970s, UNIX System versions adopted swapping as its memory management strategy. It was designed to transfer entire processes between primary memory and the disk. Swapping was quite suitable for the hardware system at that time, which had a small size memory, such as PDP-11 (whose total physical main memory was about 256 Kbytes). With swapping as the memory management scheme, the size of the physical memory space restricts the size of processes that can be running in the system. However, the swapping is easy to implement and its system overhead is quite small.

Around the second half of 1970s, with the advent of the VAX that had 512-byte pages and a number of gigabytes of virtual address space, the BSD variants of UNIX first implemented demand paging in memory management. Demand paging transfers pages instead of a whole process between the main memory and the disk. To start executing, a process is not necessarily to load in the memory as a whole, but several pages of its initial segment. During its execution, when the process references the pages that are not in memory, the kernel loads them in memory on demand. The demand paging allows a big-sized process to run in a small-sized physical memory space and more processes to execute simultaneously in a system than just swapping. Even though demand paging is more flexible than just swapping, demand paging

implementation needs the swapping technique to replace the pages.

From UNIX System V, UNIX System versions also support demand paging.

Another way used in memory management is segmentation, which divides the user program into logical segments that are corresponding to the natural length of programming and have unequal sizes from one segment to another. As it is used less in UNIX kernel, we won't discuss it deeply in this book.

5.1.2 Memory Allocation Algorithms in Swapping

With swapping, the memory is assigned to processes dynamically when a new process is created or an existent process has to swap in from the disk. The memory management system must handle it. A common method used to keep track of memory usage is linked lists. The system can maintain the allocated and free memory segments with one linked list or two separate lists. The allocated memory segments hold processes that reside currently in memory, and the free memory segments are empty holes that are in between two allocated segments.

With one list, the segment list can be sorted by address and each entry in the list can be specified as allocated or empty with a marker bit.

With two separate lists for allocated and free memory segments respectively, each entry in one of the lists holds the address and size of the segment, and two pointers. One pointer points to the next segment in the list; the other points to the last segment in the list.

With the lists of memory segments, the following algorithms can be used to allocate memory for a new process or an existent process that has to swap in. Having one mixed list, the algorithms search the only list; with two separate lists, the algorithms scan the list of free memory segments (or the free list), and after allocated, the chosen segment is transferred from the free list to the allocated list.

- First-fit algorithm. It scans the list of memory segments from the beginning until it finds the first free segment that is big enough to hold the process. The chosen free segment is tried to split into two pieces. One piece is just enough for the process. If the rest piece is greater than or equal to the minimum size of free segments, one piece is assigned to the process and the other remains free. If the rest piece is less than the minimum size of free segments, the whole segment is assigned to the process without being divided. The list entries are updated.
- Next-fit algorithm. It works almost the same way as first fit does, except that at the next time it is called to look for a free segment, it starts scanning from the place where it stopped last time. Thus, it has to record the place where it finds the free segment every time. And when the searching reaches the end of the list, it goes back to the beginning of the list and

continues.

- Best-fit algorithm. It searches the whole list, takes the smallest free segment that is enough to hold the process, assigns it to the process, and updates the list entries.
- Quick-fit algorithm. It has different lists of the memory segments and puts the segments with the same level of size into a list. For example, it may have a table with several entries, in which the first entry is a pointer to a list of 8 Kbyte segments, which are the segments whose sizes are less than or equal to 8 Kbytes; the second entry is a pointer to a list of 16 Kbyte segments, which are the segments whose sizes are less than or equal to 16 Kbytes and greater than 8 Kbytes; and so on. When called, it just searches the list of the segments with the size that is close to the requested one. After allocated, the list entries should be updated.

First-fit algorithm is simple and fast. When all the processes do not occupy all the physical memory space, Next-fit algorithm is faster than the first-fit algorithm because first-fit the algorithm does searching always from the beginning of the list (Bach 2006; Bays 1977). When the processes fill up the physical memory and some of them are swapped out on the disk, next-fit algorithm does not necessarily surpass first-fit algorithm. Best-fit algorithm is slower than first-fit and next-fit algorithms because it must search the entire list every time. Quick-fit algorithm can find a required free segment faster than other algorithms (Bach 2006; Bays 1977).

When deallocating memory, the memory management system has to do merge free segments to avoid memory split into a large number of small fragments. That is, if the neighbors of the newly deallocated segment are also free, they are merged into one bigger segment by revising the list entries.

Two separate lists for allocated and free segments can speed up all the algorithms, but make the algorithms more complicated and merging free segments when deallocating memory more costly, especially for quick-fit algorithm.

The early UNIX System versions adopted the fist-fit algorithm to carry out allocation of both main memory and swap space.

5.1.3 Page Replacement Algorithms in Demand Paging

For demand paging, only a number of pages of a process reside in memory (Braams 1995; Brawn et al 1970; Christensen et al 1970; Cmelik et al 1989; Denning 1970; Iftode et al 1999; Park et al 1996; Peachey et al 1984; Weizer et al 1969). When some page has to be referenced but not brought in memory yet, it is called a page fault. When a page fault occurs and there is no room available in memory, the memory management system has to choose a page to remove from memory to make room for the page that has to be brought in. In a virtual memory system, pages in main memory may be either clean or

dirty. If it is clean, the page has not been modified and will not be rewritten to the disk when it is removed from the memory because the disk copy is the same as the one in memory. If it is dirty, the page has been changed in memory, and must be rewritten to the disk if it is removed from the memory. Then the new page to be read will be brought in memory and overwrite the old page.

Several important replacement algorithms (Midorikawa et al 2008; Quarterman et al 1985; Ritchie et al 1974; Stallings 1998) are as follows. When considering how good a page replacement algorithm is, we can examine how frequent the thrashing happens when the algorithm used. The thrashing is the phenomenon that the page that has been just removed from memory is referenced and has to be brought in memory again (Denning 1968).

- The optimal page replacement algorithm. It is an ideal algorithm, and of no use in real systems. But traditionally, it can be used as a basis reference for other realistic algorithms. It removes the optimal page that will be referenced the last among processes currently in memory. But it is difficult and costly to look for this page.
- The first-in-first-out (FIFO) page replacement algorithm. It removes the page that is in memory for the longest time among all the processes currently residing in memory. The memory management system can use a list to maintain all pages currently in memory, and put the page that arrives the most recently at the end of the list. When a page fault happens, the first page in the list, which is the first comer, is removed. The new page is the most recent comer, so it is put at the end of the list.
- The least recently used (LRU) page replacement algorithm. It assumes that pages that have not been used for a long time in the past would remain unused for a long time in the future. Thus, when a page fault occurs, the page unused for the longest time in the past will be removed. To implement LRU paging, it is necessary to have a linked list of all pages in memory. When a page is referenced, it is put at the end of the list. Thus, for a while, the head of the list is the least recently used page, which will be chosen as the one to be removed when a page fault happens. The implementation can also be performed with hardware. One way is to equip each page in memory with a shift register, as shown in Figure 5.1.

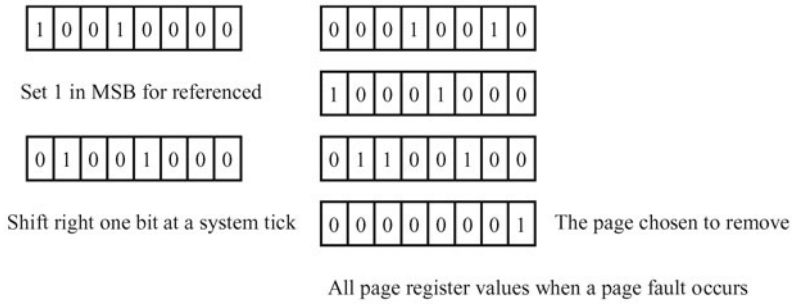

Fig. 5.1 The shift registers for the least recently used page replacement algorithm.

Every time a page is referenced, its register MSB is set. All the registers are shifted right one bit in each slice of system time. When a page fault occurs, the page with the lowest register value is chosen to remove.

- The clock page replacement algorithm. It puts all the pages in memory in a circular list and maintains the list in a clock-hand-moving order, as shown in Figure 5.2. Usually, the virtual memory in a computer system has a status bit associated with each page, for example, R, which is set when the page is referenced. If a page fault occurs, the system begins its page scanning along the clock-like list. If its R bit is zero, the page is chosen to remove, and replaced with the new page. Then the searching pointer, which works like the clock hand, moves to point to next page in the list and stops there until the next page fault. If R is one, the system clears the R bit and moves the searching pointer to the next page in the list. The pointer motion is repeated until a page with zero R bit is found. Then the system does the page replacement.

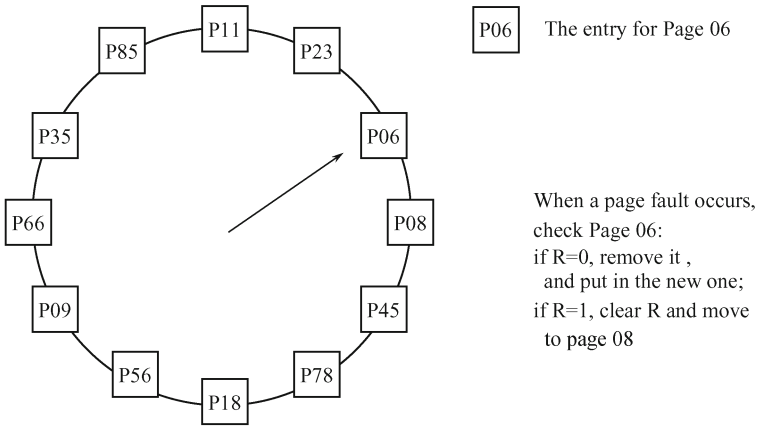

Fig. 5.2 The clock page replacement algorithm.

- The revised clock page replacement algorithm. If the virtual memory management in a computer system has two status bits associated with each page, one for reference (R) and the other for modification (M). That is, R is set when the page is referenced, and M is set when the page is modified (written or dirty). The bits can be contained in each page table entry. If the hardware does not have these bits, they can be simulated with a data structure. Pages in memory can have four cases: R=0 and M=0, in which pages are not referenced recently and not modified; R=0 and M=1, in which pages are not referenced recently but modified; R=1 and M=0, in which pages are referenced recently but not modified; R=1 and M=1, in which pages are referenced recently and modified.

Now there are four cases to choose. Obviously, the case of R=0 and M=0 is the first choice; the case of R=0 and M=1 is the second one. Like the clock

page replacement algorithm, when a page fault happens, the system begins its page scanning along the clock-like list. If both of its R and M bits are zero, the page is chosen to remove, and replaced with the new page. The searching pointer moves to point to next page in the list and stops there until the next page fault.

If the searching takes a whole circle of the clock-like list without finding one page with R=0 and M=0, it starts second searching circle. In the second searching, it does two operations: one is looking for a page with R=0 and M=1; the other is clearing R for pages with R=1. If it encounters a page with R=0 and M=1, it rewrites it to the disk and replaces it with the new page. The searching pointer moves to point to next page in the list and stops there until the next page fault.

If a page with R=0 and M=1 is not found in the second searching circle, the third searching circle has to start. Since in the second circle, R bits have been cleared, a page with R=0 must be found in the third searching circle. Then the system does the page replacement.

Compared to the clock page replacement algorithm, the revised clock page replacement algorithm allows the pages to stay in memory longer and reduce thrashing.

Because the VAX has no page reference bit in hardware memory management, the 4.2BSD that was developed on VAX used an algorithm that is slightly different from the clock page replacement algorithm (Quarterman et al 1985). The missing hardware reference bit is substituted with a field of a data structure. A software clock hand scans all pages in memory repeatedly and labels each page as invalid when it reaches the page for the first time, which is like clearing reference bit. The page can become valid again if it is accessed before the clock hand arrives at it again. But if the page has not been accessed when it is encountered again, it is replaced by a new page.

UNIX System V follows a least recently used page replacement algorithm that was implemented differently from the LRU algorithm introduced above. We will discuss it in Section 5.3.

5.2 Process Swapping in UNIX

As UNIX memory management started from swapping the whole processes out of or in memory, in this section, we will first discuss how to swap a process as a whole out of or in memory.

5.2.1 Swapped Content

We know the swapping moves a whole process between the memory and the

swap space on the disk. What does the swapped content of a process consist of?

In UNIX, since processes can execute in user mode or kernel mode, typically, the major data associated with a process consist of the instruction segment, the user data segment, and the system data segment. Except the private code, the instruction segment may include the shared code, which can be used by several processes. The user data segment includes user data and stack. The system data segment is composed of kernel data and stack. Either data or stack in both user and system segments can grow during the process executing.

The sharable code is not necessary to swap because it is only to read and there is no need to read in a piece of shared code for each process if the kernel has already brought it in memory for some process. On the other hand, multiple processes using the same code can save the memory space. In UNIX, shared code segments are treated with an extra mechanism.

Except the shared code, all the other segments, including the private code of the instruction segment, the user data segment, and the system data segment, can be swapped if necessary.

To swap easily and fast, all the segments have to keep in a contiguous area of memory. Contiguous placement of a process can cause serious external fragmentation of memory. However, in demand paging, it is not necessary to put a process in a contiguous area of memory.

5.2.2 Timing of Swapping

In UNIX kernel, the swapper is responsible for swapping processes between the memory and the swap area on the disk. The swapper is Process 0 (see Section 4.2.2).

The swapper is awaked at least once in a set slice of time (for example 4 seconds) to check whether or not there are processes to be swapped in or out. The swapper does examine the process control table to search a process that has been swapped out and ready to run. If there is free memory space available, the kernel allocates main memory space for the process, copies its segments into memory, and changes its state from ready swapped into ready in memory, and puts it in the proper priority queue to compete CPU with other processes that are also ready in memory.

If the kernel finds the system does not have enough memory to make the process to be swapped in, the swapper will examine the process control table to find a process that sleeps in memory waiting for some event happens and put it in the swap space on the disk. Then the swapper is back to search a process to swap in. The free memory space is allocated to that process.

Except when there is not enough room in memory for all the existing processes, swapping-out can also happen if one of two cases occurs: one is

some segments of a process increase and its old holder cannot accommodate the process; the other is a parent process creating a child process with a fork system call.

As known, both the user and system data segments may grow and exceed the original scope during the process execution. If there is enough memory to allocate a new memory space for the process, the allocation will be done directly by the invocation of the brk system call that can set the highest address of a process's data segment and its old holder will be freed by the kernel.

If not, the kernel does an expansion swap of the process, which includes: to allocate the swap space on the disk for the new size of the process, to modify the address mapping in the process control table according to the new size, to swap the process out on the swap space, to initiate the newly expanded space on the disk, and to modify its state as "ready, swapped". When the swapper is invoked again, the process will be swapped in memory and finally resume its execution in a new larger memory space.

When a child process is created via the fork system call, the kernel should allocate main memory space for it (see Figure 4.4). However, if there is no room in memory available, the kernel will swap out the child process onto the swap space without freeing the memory because the memory has not been allocated yet and set the child in "ready, swapped". Later on, the swapper will swap it in memory.

5.2.3 Allocation Algorithm

As mentioned before, the first-fit algorithm was adopted in UNIX to allocate memory or the swap space on the disk to processes.

In UNIX, the swap space on the disk is allocated to a process with sequential blocks. To do so is for several reasons: first, the use of the swap space on the disk is temporary; second, the speed of transferring processes between the memory and the swap space on the disk is crucial; third, for I/O operations, several contiguous blocks of data transfer are faster than several separate blocks.

The kernel has a mapping array to manage the swap space. Each entry of the mapping array holds the address and number of blocks of a free space. At first, the array has only one entry that consists of the total number of blocks that the swap space in the system can have. After several times of swapping in and out, the mapping array can have many entries.

The malloc system call is used to allocate the swap space to the process to be swapped out. With the first-fit algorithm, the kernel scans the mapping array for the first entry that can make the process fit in. If the size of the process can cover all the blocks of entry, all the blocks are allocated to the process, and this entry is removed from the array. If the process cannot use

all the blocks, the kernel breaks up the blocks into two sequential groups, one that are enough for the process are allocated to the process; the other becomes a new entry with modified address and number of blocks in the mapping array.

To free the swap space, the kernel does some merge just like what it does when deallocating memory. If one or both of the front and back neighbors are free, the newly freed entry is merged with them. The address or the number of blocks of the new entry should be modified, and some entry may be deleted according to the situation. If the newly freed entry is separate, the kernel adds one entry into an appropriate position of the mapping array and fills in its address and number of blocks.

5.2.4 Selection Principle of Swapped Processes

The selection principles for the processes to be swapped out or in are slightly different.

To swap in processes, the swapper has to make a decision on which one to be swapped in earlier than others. Two rules are used.

- It examines processes that have been swapped out and are ready to run.
- It tests how long a process stays on the disk. The longer time a process stays than others, the earlier it will be swapped in.

To swap out processes, the swapper chooses a process according to the rules:

- It examines the processes that are sleeping in memory for some event to occur.
- It checks how long a process stays in memory. The process for the longest time will be first swapped out.

5.2.5 Swapper

The swapper or Process 0 is a kernel process that enters an infinite loop after the system is booted. It tries to swap processes in memory from the swap space on the disk or swap out processes from memory onto the swap space if necessary, or it goes to sleep if there is no process suitable or necessary to swap. The kernel periodically schedules it like other processes in the system. When the swapper is scheduled, it examines all processes whose states are “ready, swapped”, chooses one that has been out on the disk for the longest time. If there is free memory space available and enough for the chosen process, the swapper does the swapping in for the process. If successful, the swapping-in repetition continues to look for other process in “ready, swapped” state and swap them in one by one until no process on the swap space is in “ready, swapped” or there is no room in memory available for

the process to be swapped in. If there is no room in memory for the process to be swapped in, the swapper enters in swapping-out searching, in which it checks the processes that are sleeping in memory, and chooses the one that has been in memory for the longest time to swap out on the swap space of the disk. Then the infinite loop goes back to look for the processes to be swapped in. If there is no chosen process, the swapper goes to sleep. The summary procedure of swapper is shown in Figure 5.3.

```
while (true)  {
   for (all processes in "ready, swapped")  {
    Look for the process swapped out on disk for the longest time;
    if (there is no such process)  {
     sleep (event for there is process to be swapped in);
     continue;
    }
    if (there is free memory space enough for the chosen process)  {
     Allocate free memory space to the process;        /* swap in the process */
    Read the process from the swap space on the disk;
    Mark the process state as "ready, in memory";
    Free the swap space;
     continue;
   }
  }
  for (all processes in "sleeping in memory") {
    Look for the process in memory for the longest time;
    if (there is no such process) {
     sleep (event for there is process to be swapped in);
     continue;
    }
    else {
      Allocate the swap space to the process; /* swap out the chosen process */
      Write the process to the swap space;
      Mark the process state as "sleeping, swap ped";
      Free the memory space;
       continue;
    }
  }
}
```

Fig. 5.3 The C-style summary procedure of swapper.

The procedure of swapper also indicates that the swapping has to handle three parts: the swap device allocation, swapping processes in memory, and swapping out of memory.

5.2.6 Swapping Effect

As known, the swapping is simple and appropriate for the system with the small main memory space. But it has some flaws.

As the swap space is on the disk, the swapping can seriously intensify the file system traffic and increase the usage of disk.

When the kernel wants to swap in a process but there is no free memory space for it, it has to swap out a process. If the size of the process to be swapped out is much less than the process to be swapped in, one time of swapping out cannot make the swapping-in successful. This may cause swapping to delay longer, especially considered that it involves I/O operations.

Further more, there is an extreme case existing for swapping. If some of the swapped-out processes are ready to run, the swapper is invoked to try to swap them in. But if there is not enough room in memory now, the swapper has to swap out some process in memory. However, if there is no room in the swap space on the disk, either, and at the same time some new process is created, a stalemate can happen. If the processes in memory are time-consuming, the deadlock can keep much longer.

5.3 Demand Paging in UNIX

Since the advent of computer systems with the virtual memory, such as VAX, in the early 1970s, operating systems have been developed to manage the memory in a new way related to virtual memory address space that is different from the physical memory space, and even better, expands the memory space into a virtual larger scope and allows more processes to execute in the system simultaneously. Demand paging in virtual memory even enhances the system throughput and makes the concurrently execution of multiple processes in a uni-processor system implement well.

As known, the locality principle was first addressed by Peter J. Denning in 1968 (Denning 1983), which uncovered the fact that the working set of pages that the process references in a short period of time are limited to a few of pages. Because the working set is a little dynamical part of a process, if the memory management handles this dynamicity as soon as possible, the system potentially can increase its throughput and allow more processes concurrently executing. We also know that the swapping needs to transfer the whole processes and may aggravate the disk I/O traffic. Demand paging transfers only some pages of the processes between the memory and the swap space on the disk, and has some mechanisms to reduce the I/O traffic that will be discussed in this section.

Demand paging usually cooperates with page replacement in the virtual memory management. The former tackles how and when to bring one or more pages of a process in memory; the latter handles how to swap out some pages

of a process periodically in order to allow new pages of a process to enter the memory. When a process references a page that is not in memory, it causes a page fault that invokes demand paging. The kernel puts the process in sleeping until the needed page is read in memory and is accessible for the process. When the page is loaded in memory, the process resumes the execution interrupted by the page fault. As there is always some new page to be brought in memory, the page replacement must dynamically swap out some pages onto the swap space. Thus, we will discuss demand paging and page replacement in separate sections.

5.3.1 Demand Paging

As known, in the virtual memory address space, the pages of a process are indexed by the logical page number, which indicates the logical order in the process. We also have a physical memory address space in the system, which is also divided into pages. To avoid making readers confused by two context-related pages, the pages in the virtual memory space are still called pages, but the pages in physical memory are usually called frames. When a page of a process is put in a frame, this page is really allocated in physical memory.

Three data structures in the UNIX kernel are needed to support demand paging: page table, frame table, and swap table. And two handlers are used to accomplish demand paging for different situations.

5.3.1.1 Page Table

Entries of the page table are indexed by the page number of a process, and one entry is for a page. Each entry of the page table has several fields: the physical address of the page, protection bits, valid bit, reference bit, modify bits, copy-on-write bit, age bits, the address on the disk, and disk type (see in Figure 5.4).

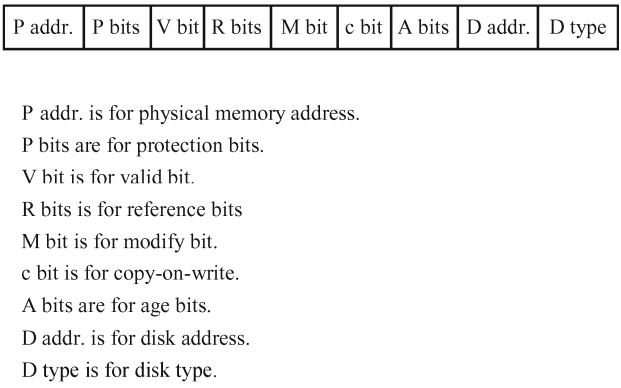

Fig. 5.4 Fields of entry in the page table.

Here gives the explanation of the fields:

- Physical address is the address of the page in the physical memory, which is the frame address that the page occupies.
- Protection bits are the access privileges for processes to read, write or execute the page.
- Valid bit indicates whether or not the content of a page is valid.
- Reference bits indicate how many processes reference the page.
- Modify bit shows whether or not the page is recently modified by processes.
- Copy-on-write bit is used by fork system call when a child process is created.
- Age bits show how long the page is in memory and are used for page replacement.
- Disk address shows the address of the page on the disk, including the logical device number and block number, no matter whether it is in the file system or the swap space on the disk.
- Disk type includes four kinds: file, swap, demand zero, and demand fill. If the page is in an executable file, its disk type is marked as file and its disk address is the logical device number and block number of the page in the file on the file system. If the page is on the swap space, its disk type is marked as swap and its disk address is the logical device number and block number of the page in the swap space on the disk. If the page is marked as "demand zero", which means bss segment (block started by symbol segment) that contains data not initialized at compile time, the kernel will clear the page when it assigns the page to the process. If the page is marked as "demand fill", which contains the data initialized at compile time, the kernel will leave the page to be overwritten with the content of the process when allocating the frame to the process.

5.3.1.2 Frame Table

Frame table is used for physical memory management. One frame of physical memory has an entry in the frame table, and the frame table is indexed with the frame number in physical memory. Entries in frame table can be arranged on one of two lists: a free frame list or a hash frame queue.

The frames on the free frame list are reclaimable. When a frame is put at the end of the free frame list, it will be allocated to a new page of a process if no process does reference it again in a period of time. However, a process may cause a page fault that is found still on the free frame list, and it can save once I/O operation of reading from the swap space on the disk.

The hash frame table is indexed with the key that is the disk address (including the logical device number and block number). One entry of the hash frame table is corresponding to a hash queue with one unique key value. With the key value, the kernel can search for a page on the hash frame table quickly.

When the kernel allocates a free frame to a page of a process, it removes

an entry at the front of the free frame list, modifies its disk address, and inserts the frame into a hash frame queue according to the disk address.

To support the frame allocation and deallocation, each entry in the frame table has several fields:

- Frame state can be several situations, for example, reclaimable, on the swap space, in an executable file, being underway of reading in memory, or accessible.
- The number of referencing processes shows how many processes access the page.
- Disk address where the page is stored in the file system or the swap space on the disk includes the logical device number and block number.
- Pointers to the forward and backward neighbor frames on the free frame list or a hash frame queue.

5.3.1.3 Swap Table

The swap table is used by page replacement and swapping. Each page on the swap space has an entry in the swap table. The entry holds a reference field that indicates how many page table entries point to the page.

5.3.1.4 Page Fault

We have known that when a process references a page that is not in memory, it causes a page fault that invokes demand paging. In fact, demand paging is a handler that is similar to general interrupt handlers, except that the demand paging handler can go to sleep but interrupt handlers cannot. Because the demand paging handler is invoked in a running process and it will be back to the running process, its execution has to be in the context of the running process. Thus, demand paging handler can sleep when I/O operation is done for the page read or swapped in memory.

Since page faults can occur in different situations during a process execution, in the UNIX virtual memory management, there are two kinds of page faults: protection page faults and validity page faults. Thus, there are two demand paging handlers, protection handler and validity handler, which handle protection page faults and validity page faults, respectively.

The protection page faults are often caused by the fork system call. The validity page faults can be resulted from several situations depending on the different stages of the execution of a process, and mostly related to the execve system call. Thus, later we will discuss these two, respectively.

Protection page fault

In the UNIX System V, the kernel manages processes with the per process region table that is usually part of process control block of the process. Each of its entries represents a region in the process and holds a pointer to the starting virtual address of the region. The region contains the page table of this region and the reference field that indicates how many processes reference the region. The per process region table consists of shared regions and private

regions of the process. The former holds one part of the process that can be shared by several processes; the latter contains the other part that is protected from other processes' references.

When the demand paging handler is invoked during the fork system call, the kernel increments the region reference field of shared regions for the child process. For each of private regions of the child process, the kernel allocates a new region table entry and page table. The kernel then examines each entry in page table of the parent process. If a page is valid, the kernel increments the reference process number in its frame table entry, indicating the number of processes that share the page via different regions rather than through the shared region in order to let the parent and child processes go in different ways after the execve system call. Similarly, if the page exists on the swap space, it increments the reference field of the swap table entry for this page. Now the page can be referenced through both regions, which share the page until one of the parent or child processes writes to it.

Then the kernel copies the page so that each region has a private version. To do this, the kernel turns on the copy-on-write bit for each page table entry in private regions of the parent and child processes during the fork system call. If either process writes the page, it causes a protection page fault that invokes the protection handler. Now we can see that the copy-on-write bit in a page table entry is designed to separate a child process creation from its physical memory allocation. In this way, via protection page fault, the memory allocation can postpone until it is needed.

The protection page fault can be caused in two situations. One is when a process references a valid page but its permission bits do not allow the process access, and the other is when a process tries to write a page whose copy-on-write bit is set by the fork system call. The kernel has to check first whether or not permission is denied in order to make a decision about what to do next, to signal an error message or to invoke the protection handler. If the latter, the protection handler is invoked.

When the protection handler is invoked, the kernel searches for the appropriate region and page table entry, and locks the region so that the page cannot be swapped out while the protection handler operates on it. If the page is shared with other processes, the kernel allocates a new frame and copies the contents of the old page to it; the other processes still reference the old page. After copying the page and updating the page table entry with the new frame number, the kernel decrements the process reference number of the old frame table entry.

If the copy-on-write bit of the page is set but the page is not shared with other processes, the kernel lets the process retain the old frame. Then the kernel separates the page from its disk copy because the process will write the page in memory but other processes may use the disk copy. Then it decrements the reference field of the swap table entry for the page and if the reference number becomes 0, frees the swap space. It clears the copy-on-write bit and updates the page table entry. Then it recalculates the process

priority because the process has been raised to a kernel-level priority when it invokes the demand paging handler in order to smooth the demand paging process. Finally, before returning to the user mode, it checks signal receipts that reached during handling the demand paging.

Through the processing above, we can see that the page copying of the child process is deferred until the process needs it and causes a protection page fault, rather than when it is created.

BSD systems used demand paging before System V and had their solution to the separate memory allocation for a child process. In BSD, there are two versions of fork system calls: one is the regular one that is just the fork system call; the other is the vfork system call that does not do physical memory allocation for the child process. The fork system call makes a physical copy of the pages of the parent process, which is a wasteful operation if it is closely followed by an execve system call. However the vfork system call, which assumes that a child process will immediately invoke the execve system call after returning from the vfork call, does not copy page tables so it is faster than the fork system call of System V. The potential risk of vfork is that if a programmer uses vfork incorrectly, the system will go into danger. After vfork system call, the child process uses the physical memory address space of the parent process before execve or exit is called, and can ruin the parent's data and stack by accident and make the parent not to be able to go back into its working context.

Validity page fault

When the execve system call is invoked, the kernel reads an executable file whose name is one of the arguments of the execve system call into memory from the file system on the disk. Readers now should realize that the disk in UNIX is divided into two portions: one is used statically for the file system; the other is allocatable dynamically for the swap space. In the memory management with demand paging, the kernel does not pre-allocate memory to the whole executable file, but assigns physical memory and reads in pages when demanded.

The kernel finds all the disk block numbers of the executable file from the inode on the file system during execve system call and puts the list in the in-core inode table entry that is in memory. We will discuss inode in Chapter 6.

To manage the memory in the demand paging way, the kernel has to assign the page tables for the executable file first, and setting the disk type field of each page table entry as demand zero or demand fill. When setting up the page tables for the executable file, the kernel fills in the disk address for each entry with the logical device number and block number containing the page. The starting logical block number of an executable file is usually 0. The validity handler will use the disk address to find and bring the page in memory from the file system on the disk.

When reading in each page of the file, a validity page fault incurs, and

the validity handler is invoked. If it is set as demand zero, the content of the frame is cleared; if it is marked as demand fill, the content of the frame is replaced with one page of the file.

The above is typically what the execve system call does. But how and what does the validity handler do for processes? The validity handler is invoked when a process tries to access a page whose valid bit is not set.

As known, if the page is valid, its content can be used rather than processed halfway. Thus, except in the valid state, in any other state during processing of the page the instruction or data of the page cannot be used to execute.

In fact, the virtual memory management in UNIX does not set the valid bit of a page table entry if the page is outside the virtual address space of a process or if the page is assigned with the virtual address space but does not have an assigned frame. Usually the hardware with virtual memory provides the function to cause a page fault when a page in an invalid state is accessed. The kernel searches for the page in the page table. If it finds it, the kernel locks the region containing the page table entry to prevent the page from being swapped out and allocates a frame of physical memory to bring in the page content from the executable file or the swap device. If not, the reference is invalid and an error signal is sent to the process. It is what the validity handler mainly does.

As analyzed above, the page that can cause the validity page fault has a zero of the valid bit of its page table entry, and typically has five possibilities: file, swap, demand zero, or demand fill in the disk type field of its page table entry, or on the free frame list in memory.

If we put it in the order that the page is processed from far from to close to the valid state of the page, we can have the following sequence of five different invalid states.

1) File

In this situation, the kernel cannot find the page on the free frame list but finds its disk address on the file system from the page table and its disk type is file. It means now the page is in the executable file on the file system and has never been read in memory before. The kernel has to read the page from its original position in the file system, which can be found from its in-code inode.

The validity handler looks into the page table for the disk address of the page, and also finds the in-core inode from its region table. The logical block number is used as an offset to find the disk block number where the page is stored in the file system by adding it to the starting logical address of the array of disk block numbers in the in-core inode. Then the kernel takes a frame table entry from the head of the free frame list, puts the frame table entry on a hash queue, reads the page content from the file system, and modifies the physical memory address field of the page table entry with the frame number. The validity handler sleeps until the disk I/O operation finishes. Then it wakes up the processes sleeping for the content of the page

to be read in.

Finally, the validity handler sets the valid bit and resets the modify bit of the page. It also recalculates the process priority and checks for signals for the same reasons as the protection handler.

2) Demand zero

In this situation, the kernel cannot find the page on the free frame list but finds that it is set as demand zero in its disk type field. It means that the page has to be cleared before being used.

The validity handler allocates a frame table entry, and clears the content of the frame. Finally, the kernel sets the valid bit, and updates the page table and frame table. It also recalculates the process priority and checks for signals.

3) Demand fill

Similar to demand zero, in this case, the kernel cannot find the page on the free frame list but finds that it is set as demand fill in its disk type field. It means that the content of the page has to be overwritten with the content of the executable file.

The validity handler allocates a frame table entry to the page, and copies the page of the file into the frame. And the kernel also does the rest part of the work like the demand zero case.

4) On the free frame list

In this case, even though the page table entry indicates that the page is swapped out, it is still on the free frame list without being reassigned to the other page.

The validity handler finds the page on the free frame list, reassigns the page table entry to the page, and increments its page reference bits. The kernel sets its valid bit, and updates the page table and frame table.

In this situation, the kernel does not need any disk I/O operation.

5) Swap, not in memory

In this situation, the kernel cannot find the page on the free frame list but finds its disk address on the swap space from the page table. It means that the page was in memory but now is swapped out on the disk.

The validity handler takes a free frame table entry, and reads the page content in the frame from the swap space. Then the validity handler sleeps until the disk I/O operation finishes. Then it wakes up the processes sleeping for the content of the page to be read in.

Finally, the validity handler sets the valid bit, and updates the page table and frame table. It also recalculates the process priority and checks for signals.

5.3.2 Page Replacement

As known, demand paging should cooperate with page replacement to implement the virtual memory management. When a process executes, its working

pages change dynamically. Some of its pages in memory should be swapped out dynamically and replaced by new pages to let the process keep executing until its work finishes.

Page replacement is similar to the swapping, except that it swaps out pages of a process rather than a whole process.

In UNIX, there are two solutions to page replacement: one is the page stealer of System V; the other is the pagedaemon of the 4.2BSD. We will discuss them in this section, respectively.

5.3.2.1 Page Stealer

Since the system's booted, the page stealer has been created. The kernel sets two threshold values for the number of free frames: one is the low value for the least number of free frames; the other is the high value for the most number. When the number of the free frames is fewer than the low value, kernel awakens the page stealer to swap out the eligible pages on the swap space of the disk. When the number of the free frames increases more than the high value, the page stealer goes to sleep. The two threshold values can be configured to reduce thrashing by system administrators.

The page stealer adopts a least recently used page replacement algorithm, and its implementation is slightly different from the ones introduced in Section 5.1.3.

When the page stealer is invoked, it does several turns of examinations on the all valid pages, for instance, four times, which can be set. At the first turn, the page stealer resets the reference bit of each valid page and begins to count how many turns of examinations have gone since the page was last referenced. Within the four turns, if no new reference to the page, its count number increments once for each turn and so that finally becomes four, which means the page has aged enough to be swapped up. Thus, the page stealer swaps out the page. However, within the four turns, when any new reference appears to the page, its count number is reset. And at the forth turn, its count number is less than four, and the page is unsuitable for swapping. The count number is recorded in the age bits of the page table entry. Thus the maximum number of examination turns can be constrained by the width of the age bit field in the page table entry.

Before the page stealer really swaps out a page, it has to see whether or not the swapping is necessary. There are two situations:

- If there is a copy of the page on the swap space and the modify bit of the page table entry is clear, it does not need to swap out the page really. Thus the page stealer just clears its valid bit of the page table entry, and decrements its reference number of the frame table entry. If the reference number becomes 0, the page stealer puts the frame entry on the free frame list.
- If there is no copy of the page on the swap space or there is a copy on the swap space but the page is modified, the page stealer puts the page on a list of pages to be swapped out, clears its valid bit of the page table entry,

and also decrements its reference number of the frame table entry. If the reference number becomes 0, the page stealer puts its frame entry on the free frame list. When the list of pages to be swapped out is full, the kernel does swap out all the pages in the list on the swap space. The collection of the pages to be swapped out can reduce the disk I/O operations.

The System V retains the swapping in its virtual memory management to reduce thrashing when the kernel cannot allocate pages for a process because the working pages in memory are larger than the physical memory space and the page replacement handlers cannot make the page replacement timely. In this situation, the swapper can swap out entire processes and alleviate the system overload quickly and effectively. When the swapped-out processes become "ready, swapped", the kernel brings them in memory with page faults rather than directly swapping in.

5.3.2.2 Pagedaemon

In the virtual memory management of the 4.2BSD, the page replacement is implemented with the pagedaemon. The pagedaemon is Process 2 of the system, along with Process 0 (called scheduler in BSD systems, swapper in System V) and Process 1 (init). The pagedaemon is used to keep the free frame list contain an enough number of frames during the system running.

There are three threshold values used in the BSD to help control when the pagedaemon or swapper to be invoked. The three threshold values are lotsfree that is a less number of free frames in the free frame list, minfree that is the least number of free frames in the free frame list, and desfree that is the average desirable number of free frames. Usually, the three values have the comparison of minfree $<$ desfree $<$ lotsfree. The pagedaemon spends most of its time sleeping, but the kernel checks periodically the number of free frames. When the number of free frames is below lotsfree, the kernel wakes up the pagedaemon. When the number of free frames increases to lotsfree, the pagedaemon stops and goes to sleep again.

If the system goes into an extreme situation with high overload, the number of free frames drops below minfree, and the average number of free frames over a period of recent time is less than desfree, the swapper is awakened to swap out a process as a whole.

The 4.2BSD also uses two lists to manage frames in physical memory: one is the free frame list that contains the free frames; the other is called the core map (cmap) that holds the used frames recorded with the disk block numbers, which indicates which process pages are mapped into the frames.

As mentioned in Section 5.1.3, the 4.2BSD adopts a slightly different clock page replacement algorithm. An algorithm clock hand scans the frames in cmap cyclically. Each examination turn includes two cycles of scanning all the frames in cmap. If some conditions indicate that a frame is in process or untouched, the frame is skipped. Otherwise, the page table entry corresponding to the frame is located and checked whether or not it is valid. If it is valid, the kernel clears its valid bit and makes the frame reclaimable. If the

page is not referenced again before the next scanning cycle reaches it, it can be allocated. If the page has been modified, it must first be written to disk before the frame is added to the free frame list.

If the page is referenced again before the next scanning cycle reaches it, a page fault occurs and the page is made valid again. It will not be put on the free frame list next time it is scanned.

With VAX's main memory space increasing, one scanning cycle of the algorithm clock hand can take so long time to make the second scanning arrival at a page have less relevance to the operations done in the first cycle. In the 4.3BSD, a second algorithm clock hand is adopted, which starts out later after the first clock hand. The first hand makes a page invalid, later on the second hand lets it reclaimed. The whole process takes less time than a whole cycle of all the frames on the free frame list.

5.4 Summary

In this chapter, we have discussed the outline of memory management, process swapping, and demand paging in UNIX.

In multiprocessing operating systems, there are many processes that must be put in different areas of the memory. Multiprogramming needs complex schemes to divide and manage the memory space for several processes in order to avoid the processes' interfering with each other when executing and make their execution just like single process executing in the system.

Considered that some processes wait for I/O devices in memory without doing anything, swapping is used as a memory management strategy. If the swapping is fast enough to let the user not to realize the delay, the system can handle more processes and the performance of the whole system becomes better.

When the size of an application program becomes too big to load in the memory as a whole at a time, the memory paging is needed.

There are several common memory allocation algorithms. They are first-fit, next-fit, best-fit, and quick-fit algorithms.The first-fit algorithm is simple and fast. next-fit algorithm is often faster than the first-fit algorithm. But when the processes fill up the memory and some of them are swapped out on the disk, the next-fit algorithm does not necessarily surpass first-fit algorithm. The best-fit algorithm is slower than the first-fit and next-fit algorithms. The quick-fit algorithm can find a required free segment faster than other algorithms.

Five common page replacement algorithms have been introduced, which are the optimal, FIFO, LRU, clock, and revised clock page replacement algorithms. The UNIX System V uses a least recently used page replacement algorithm that was implemented differently. The 4.2BSD used a slightly different clock page replacement algorithm.

In swapping in UNIX, swapped context of a process, the timing of swapping, and the selection principle of swapped processes are important. The swapper is a kernel process that tries to swap processes in memory from the swap space on the disk or swap out processes from memory onto the swap space if necessary.

In demand paging in UNIX, three data structures are used to support the demand paging and page replacement, which are page table, frame table, and swap table. When page faults occur, two kinds of handlers can be invoked. The protection handler services for protection page faults that are usually caused by the folk system call. The validity handler tackles the validity page faults that often occur during the execve system call. There are two solutions for page replacement in UNIX: one is the page stealer used in UNIX System V; the other is the pagedaemon existing in the 4.2BSD.

Problems

Problem 5.1 For the memory management scheme that divides the physical memory into different size-fixed areas, what are the biggest problems? How can they be solved?

Problem 5.2 Please explain the swapping mechanism? Why is the swapping mechanism brought into memory management? What is the shortcoming of the swapping?

Problem 5.3 What is the demand paging? For what reason can the demand paging be used in memory management? Compared to the swapping, what advantages does the demand paging have?

Problem 5.4 In swapping, how does the kernel keep track of memory usage? If you are the designer of the memory management system, how can you manage the physical memory of your system in the swapping scheme? Please give your algorithm in detail.

Problem 5.5 In this chapter, several memory allocation algorithms have been discussed. What are they? Please describe them in your way.

Problem 5.6 If you are the developer, please choose one of the three algorithms for your memory management system: the first-fit, next-fit or best-fit algorithm and implement it. Please explain why you choose it and how you will implement it.

Problem 5.7 If you are the developer, how can you implement the quick-fit algorithm? Please describe the practicable implementation in detail.

Problem 5.8 When you use the quick-fit algorithm for the memory allocation, how will you carry out the deallocation of the memory when some process exits the system and needs to free its memory space?

Problem 5.9 We have discussed several page replacement algorithms in this chapter. Try to introduce and compare them in your way.

Problem 5.10 If you are the developer, please use the least recently used

algorithm and implement it. You can use software or hardware to do your implementation. Please explain how you will do.

Problem 5.11 For a system, what reasons can cause the swapper invoked too frequently? If you are the developer of the system, how can you handle it?

Problem 5.12 Please give your version of the malloc system call program to allocate the swap space for the process to be swapped out. You can descript it with a C-like algorithm.

Problem 5.13 If you have to design a virtual memory management with demand paging, please design an algorithm to manipulate the hash frame table for the used frames.

Problem 5.14 Please compare the two handlers: protection handler and validity handler. Give your implementation versions of these handlers, respectively.

Problem 5.15 Please give a solution to deallocate the swap space on the disk in detail.

References

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Bartels G, Karlin A, Anderson D et al (1999) Potentials and limitations of fault-based Markov prefetching for virtual memory pages. SIGMETRICS'99: The 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Atlanta, Georgia, June 1999: 206 – 207

Bays C (1977) A comparison of next-fit, first-fit, and best-fit. Commun ACM 20(3): 191 – 192

Belay LA, Nelson RA, Shedler GS (1969) An anomaly in space-time characteristics of certain programs running in a paging machine. Commun ACM 12(6): 349 – 353

Braams J (1995) Batch class process scheduler for UNIX SVR4. SIGMETERICS'95/PERFORMANCE'95: The 1995 ACM SIGMETERICS Joint International Conference on Measurement and Modeling of Computer Systems, Ottawa, 1995, Canada. ACM, pp 301 – 302

Brawn BS, Gustavson FG, Mankin ES (1970) Sorting in a paging environment. Commun ACM 13(8): 483 – 494

Christensen C, Hause AD (1970) A multiprogramming, virtual memory system for a small computer. AFIPS'70 (spring): Spring Joint Computer Conference, Atlantic City, New Jersey, 5 – 7 May 1970: pp 683 – 690

Cmelik RF, Gehani NH, Roome WD (1989) Experience with multiple processor versions of concurrent C. IEEE T Software Eng 15(3): 335 – 344

Denning PJ (1968) Thrashing: its causes and prevention. AFIPS'68 (Fall, part I): Fall Joint Computer Conference, part I, San Francisco, California, 9 – 11 December 1968: pp 915 – 922

Denning PJ (1970) Virtual memory. ACM Compt Surv 2(3): 153 – 189

Denning PJ (1983) The working set model for program behavior. Commun ACM 26(1): 43 – 48

Iftode L, Blumrich M, Dubnicki C et al (1999) Shared virtual memory with automatic update support. ICS'99: The 13th International Conference on Supercomputing, Rhodes, Greece, May 1999, ACM: 175 – 183

McKusick MK, Neville-Neil GV (2005) The design and implementation of FreeBSD operating system. Addison-Wesley, Boston

Midorikawa ET, Piantola RL, Cassettari HH (2008) On adaptive replacement based on LRU with working area restriction algorithm. ACM SIGOPS Operating Systems Review 42(6): 81 – 92

Park Y, Scott R (1996) Virtual memory versus file interfaces for large, memory-intensive scientific applications. Supercomputing'96: The 1996 ACM/IEEE Conference on Supercomputing (CDROM), Pittsburgh, Pennsylvania, November 1996, IEEE computer society: Article No.: 53

Peachey DR, Bunt RB, Williamson CL et al (1984) An experimental investigation of scheduling strategies for UNIX. SIGMETRICS'84: ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, August 1984, 12(3): pp 158 – 166

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading, Massachusetts

Rashid R, Tevanian A, Michael JR et al (1988) Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. IEEE T Comput 37(8): 896 – 908

Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17 (7): 365 – 375

Stallings W (1998) Operating systems: internals and design principles, 3rd edn. Prentice Hall, Upper Saddle River, New Jersey.

Weizer N, Oppenheimer G (1969) Virtual memory management in a paging environment. Spring Joint Computer Conference, ACM, Boston, Massachusetts, 14 – 16 May 1969: pp 249 – 256