Preemptive allocation. Nonpreemptive allocation techniques cannot make efficient use of memory in all situations; overflow, i.e., rejection of a memory allocation request due to insufficient space, can be expected to occur with main memory only partially full. Much more efficient use of the available memory space is possible if the occupied space can be reallocated to make room for incoming blocks. This may be done in two ways:

1. The blocks already in main memory can be relocated in main memory to make a gap large enough for the incoming information; this is illustrated in Fig. 5.30.

2. One or more occupied regions can be made available by deallocating or expelling the blocks they contain. This requires a rule for selecting blocks to be replaced. A distinction must also be made between “dirty” blocks, which have been modified since being loaded into main memory, and “clean” blocks, which have not been modified. Blocks of instructions generally remain clean, whereas blocks of data generally become dirty. To replace a clean block, the operating system can simply overwrite it with the new block, and update its entry in the memory map. Before a dirty block can be overwritten, it must be copied onto secondary memory, which involves a time-consuming IO operation.

The relocation of the blocks already occupying main memory can be accomplished by a technique called compaction, which is illustrated in Fig. 5.36. The blocks currently in memory are combined into a single block placed at one end of the memory. This creates a single available region of the maximum possible size. The main disadvantage of this technique is the time required for compacting. If $t_M$ is the cycle time of main memory, then the time required to compact the $S$-word memory is at least $2uSt_M$, where $u$ is the fraction of the memory that is occupied.

A simple allocation technique can be based on compacting alone [11]. After each compaction, incoming blocks are assigned to contiguous regions at the unoccupied end of the memory. The memory is therefore viewed as having a single available region; new available regions due to freed blocks are ignored. When the hole at the end of the memory is filled, compaction is again carried out. The advantage of this scheme is that it eliminates the problem of selecting an available region; it may, however, result in the system’s spending an excessive amount of time compacting memory.

Replacement policies. The second major approach to preemptive allocation involves preemption a region $R$ occupied by block $K$ and allocating it to an incoming block $K'$. The criteria used for selecting $K$ as the block to be replaced constitute the replacement policy. The major objective in choosing a replacement
policy is to maximize the hit ratio or, equivalently, minimize the number of times a referenced block is not in main memory, a condition called a memory fault.

It is generally believed that the hit ratio tends to a maximum if the time intervals between successive memory faults are maximized. An optimal replacement strategy would therefore at time \( t_i \), determine the time \( t_i > t_i \), at which the next reference to block \( K \) is to occur; the \( K \) to be replaced is the one for which \( t_i - t_i \) has the maximum value \( t_K \). This ideal strategy has been called OPT [15]. OPT can be implemented by making two passes through the program. The first is a simulation run to determine the sequence \( S_B \) of distinct logical block addresses generated by the program; the sequence is called the block address stream or block address trace. The values of \( t_K \) at each point in time can be computed from \( S_B \) and used to construct the optimal sequence \( S_B^{OPT} \) of blocks to be replaced. The second run is the execution run, which uses \( S_B^{OPT} \) to specify the blocks to be replaced. OPT is not a practical replacement policy because of the cost of the simulation run and the fact that \( S_B \) may be extremely long, making \( S_B^{OPT} \) very expensive to compute. A practical replacement policy attempts to estimate \( t_K \) using statistics it gathers on the past references to all blocks currently in main memory.

Two of the most commonly implemented replacement policies are first-in first-out (FIFO) and least recently used (LRU). FIFO selects for replacement the block least recently loaded into main memory. FIFO has the advantage that it is easily implemented. A loading sequence number is associated with each block in the occupied space list. Each time a block is transferred to or from main memory, the loading sequence numbers are updated. By inspecting these numbers, the operating system can easily determine the oldest (first-in) block. FIFO has the defect, however, that a frequently used block, e.g. one containing a program loop, may be replaced because it is the oldest block.

The LRU policy selects for replacement the block that was least recently accessed by the processor. It is based on the very reasonable assumption that the least recently used block is the one least likely to be referenced in the future. The LRU policy avoids the replacement of frequently used blocks, which can occur
with FIFO. It is slightly more difficult to implement than FIFO, however, since
the operating system must maintain statistics on the times of references to all
blocks in main memory. LRU can be implemented by associating a hardware or
software counter, called an age register, with every block in main memory (see
also Prob. 5.19). Whenever a block is referenced, its age register is set to a
predetermined positive number. At fixed intervals of time, the age registers of all
the blocks are decremented by a fixed amount. The least recently used block at
any time is the one whose age register contains the smallest number.

The performance of a replacement policy in a given memory organization
can be analyzed using the block address stream generated by a set of representative
computations. Let \( N_1^* \) and \( N_2^* \) denote the number of references to \( M_1 \) and \( M_2 \),
respectively, in the block address stream. The block hit ratio \( H^* \) is defined by

\[
H^* = \frac{N_1^*}{N_1^* + N_2^*}
\]

which is analogous to the (word) hit ratio \( H \) defined by Eq. (5.2). Let \( n^* \) denote
the average number of consecutive word address references within each block. \( H \)
can be estimated from \( H^* \) using the following relation:

\[
H = 1 - \frac{1 - H^*}{n^*}
\]

In a paging system, \( H^* \) is the page hit ratio. \( 1 - H^* \), the page miss ratio, is also
called the page fault probability.

Example Comparison of replacement policies. Consider a paging system in
which main memory has a capacity of three pages. The execution of a program \( Q \)
requires reference to five distinct pages \( P_i \), where \( i = 1, 2, 3, 4, 5 \), and \( i \) is the
page address. The page address stream formed by executing \( Q \) is

\[2 \ 3 \ 2 \ 1 \ 5 \ 2 \ 4 \ 5 \ 3 \ 2 \ 5 \ 2\]

which means that the first page referenced is \( P_2 \), the second is \( P_3 \), etc. Figure 5.37
shows the manner in which the pages are assigned to main memory using FIFO,
LRU, and the ideal OPT replacement policies. The next block to be selected for
replacement is marked by an asterisk in the FIFO and LRU cases. It will be observed
that LRU recognizes that \( P_2 \) and \( P_3 \) are referenced more frequently than other pages,
whereas FIFO does not. Thus FIFO replaces \( P_2 \) twice but LRU does so only once.
The highest page hit ratio is achieved by OPT, the lowest by FIFO. The page hit ratio
of LRU is quite close to that of OPT, a property which seems to be generally true.

Stack replacement policies. As discussed in Sec. 5.2.1, the cost and performance
of a memory hierarchy can be measured by average cost per bit \( c \) and average
access time \( t_A \). Equations (5.1) and (5.4) repeated below are convenient expressions for \( c \) and \( t_A \):

\[
c = \frac{c_1S_1 + c_2S_2}{S_1 + S_2}
\]

\[
t_A = \frac{t_1S_1 + t_2S_2}{S_1 + S_2}
\]
number of analytic models for optimizing memory design have been proposed. Notable among these is a technique called stack processing, which is applicable to paging systems that use a class of replacement algorithms called stack algorithms [15]. Let \( A \) be any page address stream of length \( L \) to be processed using a replacement policy \( R \). Let \( t \) denote the point in time when the first \( t \) pages of \( A \) have been processed. Let \( n \) be a variable denoting the page capacity of \( M \). \( B_t(n) \) denotes the set of pages in \( M \) at time \( t \), and \( L_0 \) denotes the number of distinct pages that have been encountered at time \( t \). \( R \) is called a stack algorithm if it has the following inclusion property:

\[
B_t(n) \subseteq B_t(n+1) \quad \text{if } n < L,
\]

\[
B_t(n) = B_t(n+1) \quad \text{if } n \geq L,
\]

LRU retains in \( M \) the \( n \) most recently used pages. Since these are always included in the \( n+1 \) most recently used pages, it can be immediately concluded that LRU is a stack algorithm. Many other replacement policies are also of this type. FIFO is a notable exception, however. Consider the following page address stream:

\[1 \ 2 \ 3 \ 4 \ 1 \ 2 \ 5 \ 1 \ 2 \ 3 \ 4 \ 5\]

Figure 5.38 shows how this address stream is processed using FIFO and main-memory capacities of three and four pages. It can be seen that at various points of time the conditions for the inclusion property are not satisfied. For example, when \( t = 7 \), \( B_7(3) = \{1, 2, 5\} \) and \( B_7(4) = \{2, 3, 4, 5\} \); therefore \( B_7(3) \nsubseteq B_7(4) \). Hence FIFO is not a stack algorithm.

The usefulness of stack replacement algorithms lies in the fact that the hit
Action of three replacement policies on a common address stream.

\[ t_A = t_{A_1} + (1 - H)t_B \]

The quantities \( c \), \( t_{A_1} \), and \( t_B \) are determined primarily by the memory-device technologies used for \( M_1 \) and \( M_2 \). Once these have been chosen, the hit ratio \( H \) must be computed for various possible system configurations. The major variables on which \( H \) depends are

1. The types of address streams encountered
2. The average block size
3. The capacity of main memory
4. The replacement policy

Simulation is perhaps the most practical technique used for evaluating different memory system designs. \( H \) is determined for a representative sample of address streams, memory technologies, block sizes, memory capacities, and replacement policies. Figure 5.37 shows a sample point in this simulation process. In this example, the block address stream, block size, and main-memory capacity are fixed, and three different replacement strategies are being tested.

Due to the large number of alternatives that exist, the amount of simulation required to optimize the design of a virtual memory system can be very great. A