IV. DATA BASES

17. Memory Management

The main task of computers is to execute programs (even usually several programs running simultaneously). These programs and their data must be in the main memory of the computer during the execution.

Since the main memory is usually too small to store all these data and programs, modern computer systems have a secondary storage too for the provisional storage of the data and programs.

In this chapter the basic algorithms of memory management will be covered. In Section 17.1 static and dynamic partitioning, while in Section 17.2 the most popular paging methods will be discussed.

In Section 17.3 the most famous anomaly of the history of operating systems—the stunning features of FIFO page changing algorithm, interleaved memory and processing algorithms with lists—will be analysed.

Finally in Section 17.4 the discussion of the optimal and approximation algorithms for the optimisation problem in which there are files with given size to be stored on the least number of disks can be found.

17.1. Partitioning

A simple way of sharing the memory between programs is to divide the whole address space into slices, and assign such a slice to every process. These slices are called ***partitions.*** The solution does not require any special hardware support, the only thing needed is that programs should be ready to be loaded to different memory addresses, i.e., they should be ***relocatable.*** This must be required since it cannot be guaranteed that a program always gets into the same partition, because the total size of the executable programs is usually much more than the size of the whole memory. Furthermore, we cannot determine which programs can run simultaneously and which not, for processes are generally independent of each other, and in many cases their owners are different users. Therefore, it is also possible that the same program is executed by different users at the same time, and different instances work with different data, which can therefore not be stored in the same part of the memory. Relocation can be easily performed if the linker does not work with absolute but with relative memory addresses, which means it does not use exact addresses in the memory but a base address and an offset. This method is called ***base addressing,*** where the initial address is stored in the so called base register. Most processors know this addressing method, therefore, the program will not be slower than in the case using absolute addresses. By using base addressing it can also be avoided that—due to an error or the intentional behaviour of a user—the program reads or modifies the data of other programs stored at lower addresses of the memory. If the solution is extended by another register, the so called limit register which stores the biggest allowed offset, i.e. the size of the partition, then it can be assured that the program cannot access other programs stored at higher memory addresses either.

Partitioning was often used in mainframe computer operating systems before. Most of the modern operating systems, however, use virtual memory management which requires special hardware support.

Partitioning as a memory sharing method is not only applicable in operating systems. When writing a program in a language close to machine code, it can happen that different data structures with variable size—which are created and cancelled dynamically—have to be placed into a continuous memory space. These data structures are similar to processes, with the exception that security problems like addressing outside their own area do not have to be dealt with. Therefore, most of the algorithms listed below with some minor modifications can be useful for application development as well.

Basically, there are two ways of dividing the address space into partitions. One of them divides the initially empty memory area into slices, the number and size of which is predetermined at the beginning, and try to place the processes and other data structures continuously into them, or remove them from the partitions if they are not needed any more. These are called fixed partitions, since both their place and size have been fixed previously, when starting the operating system or the application. The other method is to allocate slices from the free parts of the memory to the newly created processes and data structures continuously, and to deallocate the slices again when those end. This solution is called dynamic partitioning, since partitions are created and destroyed dynamically. Both methods have got advantages as well as disadvantages, and their implementations require totally different algorithms. These will be discussed in the following.

17.1.1. Fixed partitions

Using ***fixed partitions*** the division of the address space is fixed at the beginning, and cannot be changed later while the system is up. In the case of operating systems the operator defines the partition table which is activated at next reboot. Before execution of the first application, the address space is already partitioned. In the case of applications partitioning has to be done before creation of the first data structure in the designated memory space. After that data structures of different sizes can be placed into these partitions.

In the following we examine only the case of operating systems, while we leave to the Reader the rewriting of the problem and the algorithms according to given applications, since these can differ significantly depending on the kind of the applications.

The partitioning of the address space must be done after examination of the sizes and number of possible processes running on the system. Obviously, there is a maximum size, and programs exceeding it cannot be executed. The size of the largest partition corresponds to this maximum size. To reach the optimal partitioning, often statistic surveys have to be carried out, and the sizes of the partitions have to be modified according to these statistics before restarting the system next time. We do not discuss the implementation of this solution now.

Since there are a constant number (m) of partitions, their data can be stored in one or more arrays with constant lengths. We do not deal with the particular place of the partitions on this level of abstraction either; we suppose that they are stored in a constant array as well. When placing a process in a partition, we store the index of that partition in the process header instead of its starting address. However, concrete implementation can differ from this method, of course. The sizes of the partitions are stored in array $size[1\,..\,m]$. Our processes are numbered from 1 to n. The array $part[1\,..\,m]$ keeps track of the processes executed in the individual partitions, while its inverse, array $place[1\,..\,n]$ stores the places where individual processes are executed. A process is either running, or waiting for a partition. This information is stored in Boolean array $waiting[1\,..\,n]$: if process number i is waiting, then $waiting[i]$ = TRUE, else $waiting[i]$ = FALSE. The space requirements of the processes are different. Array $spacereq[1\,..\,n]$ stores the minimum sizes of partitions required to execute the individual processes.

Having partitions of different sizes and processes with different space requirements, we obviously would not like small processes to be placed into large partitions, while smaller partitions are empty, in which larger processes do not fit. Therefore, our goal is to assign each partition to a process fitting into it in a way that there is no larger process that would fit into it as well. This is ensured by the following algorithm:

```
LARGEST-FIT($place, spacereq, size, part, waiting$)

1  for $j \leftarrow 1$ to $m$
2      do if $part[j] = 0$
3            then LOAD-LARGEST($place, spacereq, size, j, part, waiting$)
```

Finding the largest process the whose space requirement is not larger than a particular size is a simple conditional maximum search. If we cannot find any processes meeting the requirements, we must leave the the partition empty.

```
LOAD-LARGEST($place, spacereq, size, p, part, waiting$)

1  $max \leftarrow 0$
2  $ind \leftarrow 0$
```

```
for $i \leftarrow 1$ to $n$
    do if $waiting[i]$ and $spacereq[i] \le size[p]$ and $spacereq[i] > max$
          then $ind \leftarrow i$
               $max \leftarrow spacereq[i]$
if $ind > 0$
   then $part[p] \leftarrow ind$
        $place[ind] \leftarrow p$
        $waiting[ind] \leftarrow$ FALSE
```

The basic criteria of the correctness of all the algorithms loading the processes into the partitions is that they should not load a process into a partition which does not fit. This requirement is fulfilled by the above algorithm, since it can be derived from the conditional maximum search theorem exactly with the mentioned condition.

Another essential criterion is that it should not load more than one processes into the same partition, and also should not load one single process into more partitions simultaneously. The first case can be excluded, because we call the Load-Largest algorithm only for the partitions for which $part[j] = 0$ and if we load a process into partition number p, then we give $part[p]$ the index of the loaded process as a value, which is a positive integer. The second case can be proved similarly: the condition of the conditional maximum search excludes the processes for which $waiting[i] =$ FALSE, and if the process number ind is loaded into one of the partitions, then the value of $waiting[ind]$ is set to FALSE.

However, the fact that the algorithm does not load a process into a partition where it does not fit, does not load more then one processes into the same partition, or one single process into more partitions simultaneously is insufficient. These requirements are fulfilled even by an empty algorithm. Therefore, we have to require something more: namely that it should not leave a partition empty, if there is a process that would fit into it. To ensure this, we need an invariant, which holds during the whole loop, and at the end of the loop it implies our new requirement. Let this invariant be the following: after examination of j partitions, there is no positive $k \le j$, for which $part[k] = 0$, and for which there is a positive $i \le n$, such as $waiting[i] =$ TRUE, and $spacereq[i] \le size[k]$.

Initialisation: At the beginning of the algorithm we have examined $j = 0$ partitions, so there is not any positive $k \le j$.

Maintenance: If the invariant holds for j at the beginning of the loop, first we have to check whether it holds for the same j at the end of the loop as well. It is obvious, since the first j partitions are not modified when examining the $(j + 1)$-th one, and for the processes they contain $waiting[i] =$ FALSE, which does not satisfy the condition of the conditional maximum search in the Load-Largest algorithm. The invariant holds for the $(j + 1)$-th partition at the end of the loop as well, because if there is a process which fulfills the condition, the conditional maximum search certainly finds it, since the condition of our conditional maximum search corresponds to the requirement of our invariant set on each partition.

Termination: Since the loop traverses a fixed interval by one, it will certainly stop. Since the loop body is executed exactly as many times as the number of the partitions, after the end of the loop there is no positive $k \leq m$, for which $part[k] = 0$,, and for which there is a positive $i \leq n$, such that $waiting[i] =$ TRUE and $spacereq[i] \leq size[k]$, which means that we did not fail to fill any partitions that could be assigned to a process fitting into it.

The loop in rows 1–3 of the LARGEST-FIT algorithm is always executed in its entirety, so the loop body is executed $\Theta(m)$ times. The loop body performs a conditional maximum search on the empty partitions – or on partitions for which $part[j] = 0$. Since the condition in row 4 of the LOAD-LARGEST algorithm has to be evaluated for each j, the conditional maximum search runs in $\Theta(n)$. Although the loading algorithm will not be called for partitions for which $part[j] > 0$, as far as running time is concerned, in the worst case even all the partitions might be empty, therefore the time complexity of our algorithm is $\Theta(mn)$.

Unfortunately, the fact that the algorithm fills all the empty partitions with waiting processes fitting into them whenever possible is not always sufficient. A very usual requirement is that the execution of every process should be started within a determined time limit. The above algorithm does not ensure it, even if there is an upper limit for the execution time of the processes. The problem is that whenever the algorithm is executed, there might always be new processes that prevent the ones waiting for long from execution. This is shown in the following example.

Example 17.1 Suppose that we have two partitions with sizes of 5 kB and 10 kB. We also have two processes with space requirements of 8 kB and 9 kB. The execution time of both processes is 2 seconds. But at the end of the first second a new process appears with space requirement of 9 kB and execution time of 2 seconds again, and the same happens in every 2 seconds, i. e., in the third, fifth, etc. second. If we have a look at our algorithm, we can see that it always has to choose between two processes, and the one with space requirement of 9 kB will always be the winner. The other one with 8 kB will never get into the memory, although there is no other partition into which it would fit.

To be able to fulfill this new requirement mentioned above, we have to slightly modify our algorithm: the long waiting processes must be preferred over all the other processes, even if their space requirement is smaller than that of the others. Our new algorithm will process all the partitions, just like the previous one.

```
LARGEST-OR-LONG-WAITING-FIT(place, spacereq, threshold, size, part, waiting)
1  for $j \leftarrow 1$ to $m$
2      do if $part[j] = 0$
3            then LOAD-LARGEST-OR-LONG-WAITING( place, spacereq, threshold,
                                                size, j, part, waiting)
```

However, this time we keep track on the waiting time of each process. Since the algorithm is only executed when one or more partitions become free, we cannot examine the concrete time, but the number of cases where the process would have fit into a partition but we have chosen another process to fill it. To implement this, the conditional maximum search algorithm has to be modified: operations have to be performed also on items that meet the requirement (they are waiting for memory and they would fit), but they are not the largest ones among those. This operation is a simple increment of the value of a counter. We assume that the value of the counter is 0 when the process starts. The condition of the search has to be modified as well: if the value of the counter of a process is too high, (i. e., higher than a certain *threshold*), and it is higher than the value of the counter of the process with the largest space requirement found so far, then we replace it with this new process. The pseudo code of the algorithm is the following:

```
Load-Largest-or-Long-Waiting($place, spacereq, threshold, size, p, part, waiting$)

 1  $max \leftarrow 0$
 2  $ind \leftarrow 0$
 3  for $i \leftarrow 1$ to $n$
 4      do if $waiting[i]$ and $spacereq[i] \leq size[p]$
 5            then if ($points[i] > threshold$ and $points[i] > points[ind]$) or
                      $spacereq[i] > max$
 6                    then $points[ind] \leftarrow points[ind] + 1$
 7                         $ind \leftarrow i$
 8                         $max \leftarrow spacereq[i]$
 9                    else $points[i] \leftarrow points[i] + 1$
10  if $ind > 0$
11     then $part[p] \leftarrow ind$
12          $place[ind] \leftarrow p$
13          $waiting[ind] \leftarrow$ FALSE
```

The fact that the algorithm does not place multiple processes into the same partition can be proved the same way as for the previous algorithm, since the outer loop and the condition of the branch has not been changed. To prove the other two criteria (namely that a process will be placed neither into more then one partitions, nor into a partition into which it does not fit), we have to see that the condition of the conditional maximum search algorithm has been modified in a way that this property stays. It is easy to see that the condition has been split into two parts, so the first part corresponds exactly to our requirement, and if it is not satisfied, the algorithm certainly does not place the process into the partition. The property that there are no partitions left empty also stays, since the condition for choosing a process has not been restricted, but extended. Therefore, if the previous algorithm found all the processes that met the requirements, the new one finds them as well. Only the order of the processes fulfilling the criteria has been altered. The time complexity of the loops has not changed either, just like the condition, according to which the inner loop has to be executed. So the time complexity of the algorithm is the same as in the original case.

We have to examine whether the algorithm satisfies the condition that a process can wait for memory only for a given time, if we suppose that there is some p upper limit for the execution time of the processes (otherwise the problem is insoluble, since all the partitions might be taken by an infinite loop). Furthermore, let us suppose that the system is not overloaded, i. e., we can find a q upper estimation for the number of the waiting processes in every instant of time. Knowing both limits it is easy to see that in the worst case to get assigned to a given partition a process has to wait for the processes with higher counters than its own one (at most q many), and at most *threshold* many processes larger than itself. Therefore, it is indeed possible to give an upper limit for the maximum waiting time for memory in the worst case: it is $(q + \mathit{threshold})p$.

Example 17.2 In our previous example the process with space requirement of 8 kB has to wait for $\mathit{threshold} + 1 = k$ other processes, all of which lasts for 2 seconds, i. e., the process with space requirement of 8 kB has to wait exactly for 2k seconds to get into the partition with size of 10 kB.

In our algorithms so far the absolute space requirement of the processes served as the basis of their priorities. However this method is not fair: if there is a partition, into which two processes would fit, and neither of them fits into a smaller partition, then the difference in their size does not matter, since sooner or later also the smaller one has to be placed into the same, or into another, but not smaller partition. Therefore, instead of the absolute space requirement, the size of the smallest partition into which the given process fits should be taken into consideration when determining the priorities. Furthermore, if the partitions are increasingly ordered according to their sizes, then the index of the smallest partition in this ordered list is the priority of the process. It is called the rank of the process. The following algorithm calculates the ranks of all the processes.

```
Calculate-Rank($spacereq, size, rank$)
 1  $order \leftarrow$ Sort($size$)
 2  for $i \leftarrow 1$ to $n$
 3      do $u \leftarrow 1$
 4         $v \leftarrow m$
 5         $rank[i] \leftarrow \lfloor (u+v)/2 \rfloor$
 6         while $order[rank[i]] < spacereq[i]$ or $order[rank[i]+1] > spacereq[i]$
 7             do if $order[rank[i]] < spacereq[i]$
 8                   then $u \leftarrow rank[i] + 1$
 9                   else $v \leftarrow rank[i] - 1$
10                $rank[i] \leftarrow \lfloor (u+v)/2 \rfloor$
```

It is easy to see that this algorithm first orders the partitions increasingly according to their sizes, and then calculates the rank for each process. However, this has to be done only at the beginning, or when a new process comes. In the latter case the inner loop has to be executed only for the new processes. Ordering of the partitions does not have to be performed again, since the partitions do not change. The only thing that must be calculated is the smallest partition the process fits into. This can be solved by a logarithmic search, an algorithm whose correctness is proved. The time complexity of the rank calculation is easy to determine: the ordering of the partition takes $\Theta(m \log_2 m)$ steps, while the logarithmic search $\Theta(\log_2 m)$, which has to be executed for n processes. Therefore the total number of steps is $\Theta((n+m)\log_2 m)$.

After calculating the ranks we have to do the same as before, but for ranks instead of space requirements.

```
Long-Waiting-or-Not-Fit-Smaller($place, spacereq, threshold, size, part, waiting$)

for $j \leftarrow 1$ to $m$
    do if $part[j] = 0$
          then Load-Long-Waiting-or-Not-Smaller( $place, spacereq,$
                                                 $threshold, size, j,$
                                                 $part, waiting$)
```

In the loading algorithm, the only difference is that the conditional maximum search has to be executed not on array *size*, but on array *rank*:

```
Load-Long-Waiting-or-Not-Smaller($place, spacereq, threshold, size, p, part, waiting$)

$mx \leftarrow 0$
$ind \leftarrow 0$
for $i \leftarrow 1$ to $n$
    do if $waiting[i]$ and $spacereq[i] \le size[p]$
          then if ($points[i] > threshold$ and $points[i] > points[ind]$) or
                  $rank[i] > max$
                  then $points[ind] \leftarrow points[ind] + 1$
                       $ind \leftarrow i$
                       $max \leftarrow rank[i]$
                  else $points[i] \leftarrow points[i] + 1$
if $ind > 0$
   then $part[p] \leftarrow ind$
        $place[ind] \leftarrow p$
        $waiting[ind] \leftarrow$ FALSE
```

The correctness of the algorithm follows from the previous version of the algorithm and the algorithm calculating the rank. The time complexity is the same as that of the previous versions.

Example 17.3 Having a look at the previous example it can be seen that both the processes with space requirement of 8 kB and 9 kB can fit only into the partition with size of 10 kB, and cannot fit into the 5 kB one. Therefore their ranks will be the same (it will be two), so they will be loaded into the memory in the order of their arrival, which means that the 8 kB one will be among the first two.

17.1.2. Dynamic partitions

Dynamic partitioning works in a totally different way from the fixed one. Using this method we do not search for the suitable processes for every empty partition, but search for suitable memory space for every waiting process, and there we create partitions dynamically. This section is restricted to the terminology of operating systems as well, but of course, the algorithms can be rewritten to solve problems connected at the application level as well.

If all the processes would finish at the same time, there would not be any problems, since the empty memory space could be filled up from the bottom to the top continuously. Unfortunately, however, the situation is more complicated in the practice, as processes can differ significantly from each other, so their execution time is not the same either. Therefore, the allocated memory area will not always be contiguous, but there might be free partitions between the busy ones. Since copying within the memory is an extremely expensive operation, in practice it is not effective to collect the reserved partitions into the bottom of the memory. Collecting the partitions often cannot even be carried out due to the complicated relative addressing methods often used. Therefore, the free area on which the new processes have to be placed is not contiguous. It is obvious, that every new process must be assigned to the beginning of a free partition, but the question is, which of the many free partitions is the most suitable.

Partitions are the simplest to store in a linked list. Naturally, many other, maybe more efficient data structures could be found, but this is sufficient for the presentation of the algorithms listed below. The address of the first element of linked list P is stored in $head[P]$. The beginning of the partition at address p is stored in $beginning[p]$, its size in $size[p]$, and the process assigned to it is stored in variable $part[p]$. If the identifier of a process is 0, then it is an empty one, otherwise it is a allocated. In the linked list the address of the next partition is $next[p]$.

To create a partition of appropriate size dynamically, first we have to divide a free partition, which is at least as big as needed into two parts. This is done by the next algorithm.

```
Split-Partition($border, beginning, next, size, p, q$)

1  $beginning[q] \leftarrow beginning[p] + border$
2  $size[q] \leftarrow size[p] - border$
3  $size[p] \leftarrow border$
4  $next[q] \leftarrow next[p]$
5  $next[q] \leftarrow q$
```

In contrast to the algorithms connected to the method of fixed partitions, where processes were chosen to partitions, here we use a reverse approach. Here we inspect the list of the processes, and try to find to each waiting process a free partition into which it fits. If we found one, we cut the required part off from the beginning of the partition, and allocate it to the process by storing its beginning address in the process header. If there is no such free partition, then the process remains in the waiting list.

```
PLACE($P$, $head$, $next$, $last$, $beginning$, $size$, $part$, $spacereq$, $place$)
for $i \leftarrow 1$ to $n$
    do if $waiting[i]$ = TRUE
          then $\star$-FIT($P$, $head$, $next$, $last$, $beginning$, $size$, $part$, $spacereq$, $place$,
                    $waiting$, $i$)
```

The $\star$ in the pseudo code is to be replaced by one of the words First, Next, Best, Limited-Best, Worst or Limited-Worst.

There are several possibilities for choosing the suitable free partition. The more simple idea is to go through the list of the partitions from the beginning until we find the first free partition into which it fits. This can easily be solved using linear searching.

```
FIRST-FIT($P$, $head$, $next$, $last$, $beginning$, $size$, $part$, $spacereq$, $place$, $waiting$, $f$)
$p \leftarrow head[P]$
while $waiting[f]$ = TRUE and $p \neq$ NIL
      do if $part[p] = 0$ and $size[p] \geq spacereq[f]$
            then SPLIT-PARTITION($p$, $q$, $spacereq[f]$)
                 $part[p] \leftarrow f$
                 $place[f] \leftarrow p$
                 $waiting[f] \leftarrow$ FALSE
         $p \leftarrow next[p]$
```

To prove the correctness of the algorithm several facts have to be examined. First, we should not load a process into a partition into which it does not fit. This is guaranteed by the linear search theorem, since this criteria is part of the property predicate.

Similarly to the fixed partitioning, the most essential criteria of correctness is that one single process should not be placed into multiple partitions simultaneously, and at most one processes may be placed into one partition. The proof of this criteria is word by word the same as the one stated at fixed partitions. The only difference is that instead of the conditional maximum search the linear search must be used.

Of course, these conditions are not sufficient in this case either, since they are fulfilled by even the empty algorithm. We also need prove that the algorithm finds a place for every process that fits into any of the partitions. For this we need an invariant again: after examining j processes, there is no positive $k \leq j$, for which $waiting[k]$, and for which there is a p partition, such that $part[p] = 0$, and $size[p] \geq spacereq[k]$.

Initialisation: At the beginning of the algorithm we have examined $j = 0$ many partitions, so there is no positive $k \leq j$.

Maintenance: If the invariant holds for j at the beginning of the loop, first we have to check whether it holds for the same j at the end of the loop as well. It is obvious, since the first j processes are not modified when examining the $(j+1)$-th one, and for the partitions containing them $part[p] > 0$, which does not satisfy the predicate of the linear search in the First-Fit algorithm. The invariant statement holds for the $(j+1)$-th process at the end of the loop as well, since if there is a free memory slice which fulfills the condition, the linear search certainly finds it, because the condition of our linear search corresponds to the requirement of our invariant set on each partition.

Termination: Since the loop traverses a fixed interval by one, it certainly stops. Since the loop body is executed exactly as many times as the number of the processes, after the loop has finished, it holds that there is no positive $k \leq j$, for which $waiting[k]$, and for which there is a p partition, such that $part[p] = 0$, and $size[p] \geq spacereq[i]$, which means that we did not keep any processes fitting into any of the partitions waiting.

Again, the time complexity of the algorithm can be calculated easily. We examine all the n processes in any case. If, for instance, all the processes are waiting, and the partitions are all reserved, the algorithm runs in $\Theta(nm)$.

However, when calculating the time complexity, we failed to take some important points of view into consideration. One of them is that m is not constant, but executing the algorithm again and again it probably increases, since the processes are independent of each other, start and end in different instances of time, and their sizes can differ considerably. Therefore, we split a partition into two more often than we merge two neighbouring ones. This phenomenon is called ***fragmentation the memory.*** Hence, the number of steps in the worst case is growing continuously when running the algorithm several times. Furthermore, linear search divides always the first partition with appropriate size into two, so after a while there will be a lot of small partitions at the beginning of the memory area, unusable for most processes. Therefore the average execution time will grow as well. A solution for the latter problem is to not always start searching at the beginning of the list of the partitions, but from the second half of the partition split last time. When reaching the end of the list, we can continue at the beginning until finding the first suitable partition, or reaching the starting partition again. This means we traverse the list of the partitions cyclically.

```
NEXT-FIT($P$, $head$, $next$, $last$, $beginning$, $size$, $part$, $spacereq$, $place$, $waiting$, $f$)
 1  if $last[P] \neq$ NIL
 2     then $p \leftarrow next[last[P]]$
 3     else $p \leftarrow head[P]$
 4  while $waiting[f]$ and $p \neq last[P]$
 5         do if $p =$ NIL
 6               then $p \leftarrow head[P]$
 7            if $part[p] = 0$ and $size[p] \geq spacereq[f]$
 8               then SPLIT-PARTITION($p$, $q$, $spacereq[f]$)
 9                    $part[p] \leftarrow f$
10                    $place[f] \leftarrow p$
11                    $waiting[f] \leftarrow$ FALSE
12                    $last[P] \leftarrow p$
13            $p \leftarrow next[p]$
```

The proof of the correctness of the algorithm is basically the same as that of the First-Fit, as well as its time complexity. Practically, there is a linear search in the inner loop again, only the interval is always rotated in the end. However, this algorithm traverses the list of the free areas evenly, so does not fragment the beginning of the list. As a consequence, the average execution time is expected to be smaller than that of the First-Fit.

If the only thing to be examined about each partition is whether a process fits into it, then it can easily happen that we cut off large partitions for small processes, so that there would not be partitions with appropriate sizes for the later arriving larger processes. Splitting unnecessarily large partitions can be avoided by assigning each process to the smallest possible partition into which it fits.

```
Best-Fit(P,head,next,last,beginning,size,part,spacereq,place,waiting,f)
 1  $min \leftarrow \infty$
 2  $ind \leftarrow$ NIL
 3  $p \leftarrow head[P]$
 4  while $p \neq$ NIL
 5        do if $part[p] = 0$ and $size[p] \geq spacereq[f]$ and $size[p] < min$
 6              then $ind \leftarrow p$
 7                   $min \leftarrow size[p]$
 8           $p \leftarrow next[p]$
 9  if $ind \neq$ NIL
10     then Split-Partition($ind, q, spacereq[f]$)
11          $part[ind] \leftarrow f$
12          $place[f] \leftarrow ind$
13          $waiting[f] \leftarrow$ FALSE
```

All the criteria of the correctness of the algorithm can be proved in the same way as previously. The only difference from the First-Fit is that conditional minimum search is applied instead of linear search. It is also obvious that this algorithm will not split a partition larger than minimally required.

However, it is not always efficient to place each process into the smallest space into which it fits. It is because the remaining part of the partition is often too small, unsuitable for most of the processes. It is disadvantageous for two reasons. On the one hand, these partitions are still on the list of free partitions, so they are examined again and again whenever searching for a place for a process. On the other hand, many small partitions together compose a large area that is useless, since it is not contiguous. Therefore, we have to somehow avoid the creation of too small free partitions. The meaning of too small can be determined by either a constant or a function of the space requirement of the process to be placed. (For example, the free area should be twice as large as the space required for the process.) Since this limit is based on the whole partition and not only its remaining part, we will always consider it as a function depending on the process. Of course, if there is no partition to fulfill this extra condition, then we should place the process into the largest partition. So we get the following algorithm.

```
LIMITED-BEST-FIT(P, head, next, last, beginning, size, part, spacereq, place, waiting, f)
 1  $min \leftarrow \infty$
 2  $ind \leftarrow$ NIL
 3  $p \leftarrow head[P]$
 4  while $p \neq$ NIL
 5       do if $part[p] = 0$ and $size[p] \geq spacereq[f]$ and
                   $((size[p] < min$ and $size[p] \geq$ LIMIT$(f))$
                   or $ind =$ NIL or $(min <$ LIMIT$(f)$ and $size[p] > min))$
 6            then $ind \leftarrow p$
 7                 $min \leftarrow size[p]$
 8          $p \leftarrow next[p]$
 9  if $ind \neq$ NIL
10    then SPLIT-PARTITION$(ind, q, spacereq[f])$
11         $part[ind] \leftarrow f$
12         $place[f] \leftarrow ind$
13         $waiting[f] \leftarrow$ FALSE
```

This algorithm is more complicated than the previous ones. To prove its correctness we have to see that the inner loop is a conditional minimum searching. The first part of the condition, i. e. that $part[p] = 0$, and $size[p] \geq spacereq[f]$ means that we try to find a free partition suitable for the process. The second part is a disjunction: we replace the item found so far with the newly examined one in three cases. The first case is when $size[p] < min$, and $size[p] \geq$ LIMIT$(spacereq[f])$, which means that the size of the examined partition is at least as large as the described minimum, but it is smaller than the the smallest one found so far. If there were no more conditions, this would be a conditional minimum search for the conditions of which we added that the size of the partition should be above a certain limit. But there are two other cases, when we replace the previously found item to the new one. One of the cases is that $ind =$ NIL, i. e., the newly examined partition is the first one which is free, and into which the process fits. This is needed because we stick to the requirement that if there is a free partition suitable for the process, then the algorithm should place the process into such a partition. Finally, according to the third condition, we replace the previously found most suitable item to the current one, if $min <$ LIMIT$(spacereq[f])$ and $size[p] > min$, which means that the minimum found so far did not reach the described limit, and the current item is bigger than this minimum. This condition is important for two reasons. First, if the items examined so far do not fulfill the most recent condition, but the current one does, then we replace it, since in this case $min <$ LIMIT$(spacereq[f]) \leq size[p]$, i. e., the size of the current partition is obviously larger. Second, if neither the size of partition found so far, nor that of the current one reaches the described limit, but the currently examined one approaches it better from below, then $min < size[p] <$ LIMIT$(spacereq[f])$ holds, therefore, also in this case we replace the item found so far by the current one. Hence, if there are partitions at least as large as the described limit, then the algorithm places each process into the smallest one among them, and if there is no such partition, then in the largest suitable one.

There are certain problems, where the only requirement is that the remaining free spaces should be the largest possible. It can be guaranteed if each process is placed into the largest free partition:

```
WORST-FIT($P$,$head$,$next$,$last$,$beginning$,$size$,$part$,$spacereq$,$place$,$waiting$,$f$)
 1  $max \leftarrow 0$
 2  $ind \leftarrow$ NIL
 3  $p \leftarrow head[P]$
 4  while $p \neq$ NIL
 5          do if $part[p] = 0$ and $size[p] \geq spacereq[f]$ and $size[p] > max$
 6                 then $ind \leftarrow p$
 7                      $min \leftarrow size[p]$
 8             $p \leftarrow next[p]$
 9  if $ind \neq$ NIL
10     then SPLIT-PARTITION($ind$,$q$,$spacereq[f]$)
11          $part[ind] \leftarrow f$
12          $place[f] \leftarrow ind$
13          $waiting[f] \leftarrow$ FALSE
```

We can prove the correctness of the algorithm similarly to the Best-Fit algorithm; the only difference is that maximum search has to be used instead of conditional maximum search. As a consequence, it is also obvious that the sizes of the remaining free areas are maximal.

The Worst-Fit algorithm maximises the smallest free partition, i. e. there will be only few partitions which are too small for most of the processes. It follows from the fact that it always splits the largest partitions. However, it also often prevents large processes from getting into the memory, so they have to wait on an auxiliary storage. To avoid this we may extend our conditions with an extra an one, similarly to the Best-Fit algorithm. In this case, however, we give an upper limit instead of a lower one. The algorithm only tries to split partitions smaller than a certain limit. This limit also depends on the space requirement of the process. (For example the double of the space requirement.) If the algorithm can find such partitions, then it chooses the largest one to avoid creating too small partitions. If it finds only partitions exceeding this limit, then it splits the smallest one to save bigger ones for large processes.

```
LIMITED-WORST-FIT($f$,$beginning$,$head$,$place$,$spacereq$,$next$,$size$,$part$,$waiting$,$waiting$)

1  $max \leftarrow 0$
2  $ind \leftarrow$ NIL
3  $p \leftarrow head[P]$
4  while $p \neq$ NIL
5          do if $part[p] = 0$ and $size[p] \geq spacereq[f]$ and
                                ($(size[p] > max$ and $size[p] \leq$ LIMIT($f$)) or $ind =$ NIL or
                                ($max >$ LIMIT($f$) and $size[p] < max$))
```

```
 6            then $ind \leftarrow p$
 7                 $min \leftarrow size[p]$
 8        $p \leftarrow next[p]$
 9  if $ind \neq$ NIL
10    then Split-Partition($ind, q, spacereq[f]$)
11         $part[ind] \leftarrow f$
12         $place[f] \leftarrow ind$
13         $waiting[f] \leftarrow$ FALSE
```

It is easy to see that this algorithm is very similar to the Limited-Best-Fit, only the relation signs are reversed. The difference is not significant indeed. In both algorithms the same two conditions are to be fulfilled: there should not be too small partitions, and large free partitions should not be wasted for small processes. The only difference is which condition is taken account in the first place and which in the second. The actual problem decides which one to use.

Exercises

17.1-1 We have a system containing two fixed partitions with sizes of 100 kB, one of 200 kB and one of 400 kB. All of them are empty at the beginning. One second later five processes arrive almost simultaneously, directly after each other without significant delay. Their sizes are 80 kB, 70 kB, 50 kB, 120 kB and 180 kB respectively. The process with size of 180 kB ends in the fifth second after its arrival, but by that time another process arrives with space requirement of 280 kB. Which processes are in which partitions in the sixth second after the first arrivals, if we suppose that other processes do not end until that time, and the Largest-Fit algorithm is used? What is the case if the Largest-or-Long-Waiting-Fit or the Long-Waiting-or-Not-Fit-Smaller algorithm is used with threshold value of 4?

17.1-2 In a system using dynamic partitions the list of free partition consists of the following items: one with size of 20 kB, followed by one of 100 kB, one of 210 kB, one of 180 kB, one of 50 kB, one of 10 kB, one of 70 kB, one of 130 kB and one of 90 kB respectively. The last process was placed into the partition preceding the one of 180 kB. A new process with space requirement of 40 kB arrives into the system. Into which partition is it to be placed using the First-Fit, Next-Fit, Best-Fit, Limited-Best-Fit, Worst-Fit or the Limited-Worst-Fit algorithms?

17.1-3 An effective implementation of the Worst-Fit algorithm is when the partitions are stored in a binary heap instead of a linear linked list. What is the time complexity of the Place algorithm perform in this case?

17.2. Page replacement algorithms

As already mentioned, the memory of the modern computer systems consists of several levels. These levels usually are organised into a seemingly single-level memory, called ***virtual memory.*** Users do not have to know this structure with several levels in detail: operating systems manage these levels.

The most popular methods to control this virtual memory are paging and segmentation. Paging divides both memory levels into fixed-sized units, called ***frames.*** In this case programs are also divided into parts of the same size as frams have: these parts of the programs (and data) are called ***pages.*** Segmentation uses parts of a program with changing size—these parts are called ***segments.***

For the simplicity let us suppose that the memory consists of only two levels: the smaller one with shorter access time is called ***main memory*** (or ***memory*** for short), and the larger one with larger access time is called ***backing memory.***

At the beginning, the main memory is empty, and there is only one program consisting of n parts in the backing memory. Suppose that during the run of the program there are instructions to be executed, and the execution of each instruction there requires an access to a certain page of the program. After processing the reference string, the following problems have to be solved.

1. Where should we place the segment of the program responsible for executing the next instruction in the main memory (if it is not there)?

2. When should we place the segments of the program in the main memory?

3. How should we deallocate space for the segments of the program to be placed into the main memory?

It is the placing algorithms that give the answer to the first question: as far as paging is concerned, the answer is simply anywhere—since the page frames of the main memory are of the same size and access time. During segmentation there are program segments and free memory areas, called holes alternating in the main memory–and it is the segment placing algorithms that gives the answer to the first question.

To the second question the answer is given by the transferring algorithms: in working systems the answer is on demand in most of the cases, which means that a new segment of the program starts to be loaded from the backing memory when it turns out that this certain segment is needed. Another solution would be preloading, but according to the experiences it involves a lot of unnecessary work, so it has not become wide-spread.

It is the replacement algorithms that give the answer to the third question: as far as paging is concerned, these are the page replacement algorithms, which we present in this section. Segment replacement algorithms used by segmentation apply basically the ideas of page replacement algorithms—completed them according to the different sizes of the segments.

Let us suppose that the size of the physical memory is m page frames, while that of the backing memory is n page frames. Naturally the inequality $1 \leq m \leq n$ holds for the parameters. In practice, n is usually many times bigger than m. At the beginning the main memory is empty, and there is only one program in the backing memory. Suppose that during the run of the program there are p instructions to be executed, and to execute the t-th instruction $(1 \leq t \leq p)$ the page r_t is necessary, and the result of the execution of the instruction also can be stored in the same page, i. e., we are modelling the execution of the program by ***reference string*** $R = \langle r_1, r_2, \ldots, r_p \rangle$. In the following we examine only the case of demand paging, to be more precise, only the page replacement algorithms within it.

If it is important to differentiate reading from writing, we will use ***writing array*** $W = \langle w_1, w_2, \ldots, w_p \rangle$ besides array R. Entry w_t of array W is TRUE if we are writing onto page r_t, otherwise $w_1 =$ FALSE.

Demand paging algorithms fall into two groups; there are ***static*** and ***dynamic*** algorithms. At the beginning of the running of the program both types fill the page frames of the physical memory with pages, but after that static algorithms keep exactly m page frames reserved until the end of the running, while dynamic algorithms allocate at most m page frames.

17.2.1. Static page replacement

The input data of static page replacement algorithms are: the size of the main memory measured in number of the page frames (m), the size of the program measured in number of of pages (n), the running time of the program measured in number of instructions (p) and the reference string (R); while their output is the ***number of the page faults.*** (*pagefault*)

Static algorithms are based on managing the ***page table.*** The page table is a matrix with size of $n \times 2$, the i-th ($i \in [0\,..\,n-1]$) row of which refers to the i-th page. The first entry of the row is a logical variable ***(present/absent bit),*** the value of which keeps track of whether the page is in the main memory in that certain instant of time: if the i-th page is in the main memory, then $pagetable[i, 1] =$ TRUE and $pagetable[i, 2] = j$, where $j \in [0\,..\,m-1]$ shows us that the page is in the j-th page frame of the main memory. If the i-th page is not in the main memory, then $pagetable[i, 1] =$ FALSE and $pagetable[i, 2]$ is non-defined. Work variable *busy* contains the number of the busy page frames
indexframe!free of the main memory.

If the size of the pages is z, then the physical address f can be calculated from virtual address v so that $j = \lfloor v/z \rfloor$ gives us the ***index of the virtual page frame***, and $v - z\lfloor v/z \rfloor$ gives us offset s referring to virtual address v. If the j-th page is in the main memory in the given instant of time—which is indicated by $pagetable[i, 1] =$ TRUE—, then $f = s + z \cdot pagetable[i, 2]$. If, however, the i-th page is not in the main memory, then a page fault occurs. In this case we choose one of the page frames of the main memory using the page replacement algorithm, load the j-th page into it, refresh the j-th row of the page table and then calculate f.

The operation of the demand paging algorithms can be described by a Mealy automaton having an initial status. This automaton can be given as $(Q, q_0, X, Y, \delta, \lambda)$, where Q is the set of the control states, $q_o \in Q$ is the initial control state, X is the input alphabet, Y is the output alphabet, $\delta : Q \times X \to Q$ is the state transition function and $\lambda : Q \times X \to Y$ is the output function.

We do not discuss the formalisation of how the automaton stop.

Sequence $R_p = \langle r_1, r_2, \ldots, r_p \rangle$ (or $R_p = \langle r_1, r_2, \ldots, r_\infty \rangle$) is called ***reference string.***

The description of the algorithms can be simplified introducing memory states

S_t $(t = 1, 2, \ldots)$: this state is the set of the pages stored in the main memory of the automat after processing the t-th input sign. In the case of static demand paging algorithms $S_0 = \emptyset$. If the new memory status differs from the old one (which means that a new page had to be swapped in), then a page fault has occurred. Consequently, both a swapping of a page into an empty frame and page replacement are called page fault.

In case of page replacement algorithms—according to Denning's proposition—instead of λ and δ we use the state transition function $g_P : Q \times M \times X \to Q \times Y$. Since for the page replacement algorithms $X = \{0, 1, \ldots, n-1\}$ and $Y = X \cup \emptyset$, holds, these two items can be omitted from the definition, so page replacement algorithm P can be described by the triple (Q, q_0, g_P).

Our first example is one of the simplest page replacement algorithms, the FIFO (**F**irst **I**n **F**irst **O**ut), which replaces the pages in the order of their loading in. Its definition is the following: $q_0 = \langle\rangle$ and

$$g_{\text{FIFO}}(S, q, x) = \begin{cases} (S, q, \epsilon), & \text{if } x \in S \ , \\ (S \cup \{x\}, q', \epsilon), & \text{if } x \notin S, \ |S| = k < m \ , \\ (S \setminus \{y_1\} \cup \{x\}, q", y_1), & \text{if } x \notin S \text{ and } |S| = k = m \ , \end{cases} \qquad (17.1)$$

where $q = \langle y_1, y_2, \ldots, y_k\rangle$, $q' = \langle y_1, y_2, \ldots, y_k, x\rangle$ and $q'' = \langle y_2, y_3, \ldots, y_m, x\rangle$.

Running of the programs is carried out by the following $*$-Run algorithm. In this section the $*$ in the name of the algorithms has to be replaced by the name of the page replacement algorithm to be applied (FIFO, LRU OPT, LFU or NRU). In the pseudocodes it is supposed that the called procedures know the values of the variable used in the calling procedure, and the calling procedure accesses to the new values.

```
*-Run(m, n, p, R, faultnumber, pagetable)

1  faultnumber ← 0
2  busy ← 0
3  for i ← 0 to n − 1                          ▷ Preparing the pagetable.
4      do pagetable[i, 1] ← FALSE
5  *-Prepare(pagetable)
6  for i ← 1 to p                              ▷ Run of the program.
7      do *-Executes(pagetable, i)
8  return faultnumber
```

The following implementation of the algorithm keeps track of the order of loading in the pages by queue Q. The preparing algorithm has to create the empty queue, i. e., to execute the instruction $Q \leftarrow \emptyset$.

In the following pseudocode *swap-out* is the index of the page to be replaced, and *swap-in* is the index of the page of the main memory into which the new page is going to be swapped in.

```
FIFO-EXECUTES($pagetable, t$),
 1  if $pagetable[r_t, 1] = \text{TRUE}$                    ▷ The next page is in.
 2     then NIL
 3  if $pagetable[r_t, 1] = \text{FALSE}$                   ▷ The next page is out.
 4     then $pagefault \leftarrow pagefault + 1$
 5          if $busy < m$                                    ▷ Main memory is not full.
 6             then INQUEUE($Q, r_t$)
 7                  $swap\text{-}in \leftarrow busy$
 8                  $busy \leftarrow busy + 1$
 9          if $busy = m$                                    ▷ Main memory is full.
10             then $replaces \leftarrow$ ENQUEUE($Q$)
11                  $pagetable[swap\text{-}out, 1] \leftarrow \text{FALSE}$
12                  $swap\text{-}in \leftarrow pagetable[swap\text{-}out, 2]$
13          WRITE($swap\text{-}in, swap\text{-}out$)
14          READ($r_t, load$)                                ▷ Reading.
15          $pagetable[r_t, 1] \leftarrow \text{TRUE}$      ▷ Updating of the data.
16          $pagetable[r_t, 2] \leftarrow loads$
```

Procedure writing writes the page chosen to be swapped out into the backing memory: its first parameter answers the question where from (from which page frame of the memory) and its second parameter answers where to (to which page frame of the backing memory). Procedure READING reads the page needed to execute the next instruction from the backing memory into the appropriate page frame of the physical memory: its first parameter is where from (from which page frame of the backing memory) and its second parameter is where to (to which page frame of the memory). When giving the parameters of both the procedures we use the fact that the page frames are of the same size, therefore, the initial address of the j-th page frame is j-times the page size z in both memories. Most of the page replacement algorithms do not need to know the other entries of reference string R to process reference r_t, so when calculating space requirement we do not have to take the space requirement of the series into consideration. An exception for this is algorithm OPT for example. The space requirement of the FIFO-RUN algorithm is determined by the size of the page frame - this space requirement is $O(m)$. The running time of the FIFO-RUN algorithm is de-termined by the loop. Since the procedure called in rows 6 and 7 performs only a constant number of steps (provided that queue-handling operations can be performed in $O(1)$, the run-ning time of the FIFO-RUN algorithm is $O(p)$. Note that some of the pages do not change while being in the memory, so if we assign a modified bit to the pages in the memory, then we can spare the writing in row 12 in some of the cases.

Our next example is one of the most popular page replacement algorithms, the LRU (**L**east **R**ecently **U**sed), which replaces the page used least recently. Its definition is the following: $q_0 = ()$ and

$$g_{\text{LRU}}(S,q,x) = \begin{cases} (S, q''', \epsilon), & \text{if } x \in S \ , \\ (S \cup \{x\}, q', \epsilon), & \text{if } x \notin S, \ |S| = k < m \ , \\ (S \setminus \{y_1\} \cup \{x\}, q", y_1), & \text{if } x \notin S \text{ and } |S| = k = m \ , \end{cases} \tag{17.2}$$

where $q = \langle y_1, y_2, \ldots, y_k \rangle$, $q' = \langle y_1, y_2, \ldots, y_k, x \rangle$, $q'' = \langle y_2, y_3, \ldots, y_m, x \rangle$ and if $x = y_k$, then $q''' = \langle y_1, y_2, \ldots, y_{k-1}, \ldots, y_{k+1} \cdots y_m, y_k \rangle$.

The next implementation of LRU does not need any preparations. We keep a record of the time of the last usage of the certain pages in array *last-call*$[0..n-1]$, and when there is a replacement needed, the least recently used page can be found with linear search.

LRU-Executes(*pagetable*, *t*)

```
 1  if pagetable[$r_t$, 1] = TRUE                          ▷ The next page is in.
 2     then last-ref[$r_t$] ← $t$
 3  if pagetable[$r_t$, 1] = FALSE                         ▷ The next page is not in.
 4     then pagefault ← pagefault + 1
 5          if busy < $m$                                  ▷ The physical memory is not full.
 6             then swap-in ← busy
 7                  busy ← busy + 1
 8          if busy = $m$                                  ▷ The physical memory is full.
 9             then swap-out ← $r_{t-1}$
10                  for $i$ ← 0 to $n - 1$
11                      do if pagetable[$i$, 1] = TRUE and
                              last-ref[$i$] < last-ref[swap-out]
12                            then swap-out ← last-ref[$i$]
13                  pagetable[swap-out, 1] ← FALSE
14                  swap-in ← pagetable[swap-out, 2]
15                  Write(swap-in, swap-out)
16          Read($r_t$, swap-in)                           ▷ Reading.
17          pagetable[$r_t$, 1] ← TRUE                     ▷ Updating.
18          pagetable[$r_t$, 2] ← swap-in
19          last-ref[$r_t$] ← $t$
```

If we consider the values of both n and p as variables, then due to the linear search in rows 10–11, the running time of the LRU-RUN algorithm is $O(np)$.

The following algorithm is optimal in the sense that with the given conditions (fixed m and n) it causes a minimal number of page faults. This algorithm chooses the page from the ones in the memory, which is going to be used at the latest (if there are several page that are not needed any more, then we choose the one at the lowest memory address from them) to be replaced. This algorithm does not need any preparations either.

OPT-Executes(*t*, *pagetable*, *R*)

```
 1  if pagetable[$r_t$, 1] = TRUE                          ▷ The next page is in.
 2     then NIL
 3  if pagetable[$r_t$, 1] = FALSE                         ▷ The next page is not in.
 4     then pagefault ← pagefault + 1
```

```
5      if $busy < m$                              ▷ The main memory is not full.
6         then $swap\text{-}in \leftarrow busy$
7              $busy \leftarrow busy + 1$
8      if $busy = m$                              ▷ The main memory is full.
9         then OPT-Swap-Out$(t, R)$
10             $pagetable[swap\text{-}out, 1] \leftarrow$ FALSE
11             $swap\text{-}in \leftarrow pagetable[swap\text{-}out, 2]$
12             Write$(swap\text{-}in, swap\text{-}out)$
13     Read$(r_t, swap\text{-}in)$                ▷ Reading.
14     $pagetable[r_t, 1] \leftarrow$ TRUE        ▷ Updating.
15     $pagetable[r_t, 2] \leftarrow swap\text{-}in$
```

Procedure OPT-Swap-Out determines the index of the page to be replaced.

OPT-Swap-Out(t, R)

```
1  $guarded \leftarrow 0$                                   ▷ Preparation.
2  for $j \leftarrow 0$ to $m - 1$
3      do $frame[j] \leftarrow$ FALSE
4  $s \leftarrow t + 1$                 ▷ Determining the protection of the page frames.
5  while $s \le p$ and $pagetable[r_s, 1] =$ TRUE and $frame[pagetable[r_s, 2]] =$ FALSE and
           $guarded < m - 1$
6        do $guarded \leftarrow guarded + 1$
7           $frame[r_s] \leftarrow$ TRUE
8           $s \leftarrow s + 1$
9  $swap\text{-}out \leftarrow m - 1$       ▷ Finding the frame containing the page to be replaced.
10 $j \leftarrow 0$
11 while $frame[j] =$ TRUE
12       do $j \leftarrow j + 1$
13 $swap\text{-}out \leftarrow j$
14 return $swap\text{-}out$
```

Information about pages in the main memory is stored in $frame[0 \,..\, m-1]$: $frame[j] =$ TRUE means that the page stored in the j-th frame is protected from being replaced due to its going to be used soon. Variable protected keeps track of how many protected pages we know about. If we either find $m-1$ protected pages or reach the end of R, then we will choose the unprotected page at the lowest memory address for the page to be replaced.

Since the OPT algorithm needs to know the entire array R, its space requirement is $O(p)$. Since in rows 5–8 of the OPT-Swap-Out algorithm at most the remaining part of R has to be looked through, the running time of the OPT-Swap-Out algorithm is $O(p^2)$. The following LFU (Least Frequently Used) algorithm chooses the least frequently used page to be replaced. So that the page replacement would be obvious we suppose that in the case of equal frequencies we replace the page at the lowest address of the physical memory. We keep a record of how many times each page has been referenced since it was loaded into the physical memory with the help of array frequency[1..n - 1]. This algorithm does not need any preparations either.

```
LFU-Executes($pagetable, t$)
 1  if $pagetable[r_t, 1] = $ TRUE                                   ▷ The next page is in.
 2    then $frequency[r_t] \leftarrow frequency[r_t] + 1$
 3  if $pagetable[r_t, 1] = $ FALSE                                  ▷ The next page is not in.
 4    then $pagefault \leftarrow pagefault + 1$
 5         if $busy < m$                                             ▷ The main memory is not full.
 6           then $swap\text{-}in \leftarrow busy$
 7                $busy \leftarrow busy + 1$
 8         if $busy = m$                                             ▷ The physical memory is full.
 9           then $swap\text{-}out \leftarrow r_{t-1}$
10                for $i \leftarrow n - 1$ downto 0
11                    do if $pagetable[i, 1] = $ TRUE and
                            $frequency[i] \leq frequency[swap\text{-}out]$
12                         then $swap\text{-}out \leftarrow last\text{-}ref[i]$
13                $pagetable[swap\text{-}out, 1] \leftarrow$ FALSE
14                $swap\text{-}in \leftarrow pagetable[swap\text{-}out, 2]$
15                Kiír($swap\text{-}in, swap\text{-}out$)
16         Read($r_t, pagetable[swap\text{-}out, 2]$)                ▷ Reading.
17         $pagetable[r_t, 1] \leftarrow$ TRUE                        ▷ Updating.
18         $pagetable[r_t, 2] \leftarrow swap\text{-}in$
19         $frequency[r_t] \leftarrow 1$
```

Since the loop body in rows 11–13 of the LFU-Executes algorithm has to be executed at most n-times, the running time of the algorithm is $O(np)$. There are certain operating systems in which there are two status bits belonging to the pages in the physical memory. The referenced bit is set to TRUE whenever a page is referenced (either for reading or writing), while the dirty bit is set to TRUE whenever modifying (i.e. writing) a page. When starting the program both of the status bits of each page is set to FALSE. At stated intervals (e. g. after every k-th instruction) the operating system sets the referenced bit of the pages which has not been referenced since the last setting to FALSE. Pages fall into four classes according to the values of their two status bits: class 0 contains the pages not referenced and not modified, class 1 the not referenced but modified, class 2 the referenced, but not modified, and finally, class 3 the referenced and modified ones.

The NRU (**N**ot **R**ecently **U**sed) algorithm chooses a page to be replaced from the nonempty class with the smallest index. So that the algorithm would be deterministic, we suppose that the NRU algorithm stores the elements of each class in a row.

The preparation of this algorithm means to fill arrays referenced and dirty containing the indicator bits with FALSE values, to zero the value of variable performed showing the number of the operations performed since the last zeroing and to create four empty queues.

NRU-PREPARES(n)

```
1  for $i \leftarrow 0$ to $n-1$
2      do $referenced[j] \leftarrow$ FALSE
3         $dirty[j] \leftarrow$ FALSE
4  $Q_0 \leftarrow \emptyset$
5  $Q_1 \leftarrow \emptyset$
6  $Q_2 \leftarrow \emptyset$
7  $Q_3 \leftarrow \emptyset$
```

NRU-EXECUTES($referenced, dirty, k, R, W$)

```
 1  if $pagetable[r_t, 1]$ = TRUE                         ▷ The next page is in.
 2     then if $W[r_t]$ = TRUE
 3             then $dirty[r_t] \leftarrow$ TRUE
 4  if $pagetable[r_t, 1]$ = FALSE                        ▷ The next page is not in.
 5     then $pagefault \leftarrow pagefault + 1$
 6          if $busy < m$                                 ▷ The main memory is not full.
 7             then $swap\text{-}in \leftarrow busy$
 8                  $busy \leftarrow busy + 1$
 9                  $referenced[r_t] \leftarrow$ TRUE
10                  if $W[r_t]$ = TRUE
11                     then $dirty[r_t] \leftarrow$ TRUE
12          if $busy = m$                                 ▷ The main memory is full.
13             then NRU-SWAP-OUT($t, swap\text{-}out$)
14                  $pagetable[swap\text{-}out, 1] \leftarrow$ FALSE
15                  $swap\text{-}in \leftarrow pagetable[swap\text{-}out, 2]$
16                  if $dirty[sap\text{-}out]$ = TRUE
17                     then WRITE($swap\text{-}in, swap\text{-}out$)
18          READ($r_t, pagetable[swap\text{-}in, 2]$)                   ▷ Reading.
19          $pagetable[r_t, 1] \leftarrow$ TRUE                            ▷ Updating.
20          $pagetable[r_t, 2] \leftarrow swap\text{-}in$
21  if $t/k = \lfloor t/k \rfloor$
22     then for $i \leftarrow 0$ to $n-1$
23             do if $referenced[i]$ = FALSE
24                   then $dirty[i] \leftarrow$ FALSE
```

Choosing the page to be replaced is based on dividing the pages in the physical memory into four queues (Q_1, Q_2, Q_3, Q_4).

NRU-SWAP-OUT($time$)

```
1  for $i \leftarrow 0$ to $n-1$                        ▷ Classifying the pages.
2     do if $referenced[i]$ = FALSE
3           then if $dirty[i]$ = FALSE
4                   then ENQUEUE($Q_1$,$i$)
5                   else ENQUEUE($Q_2$,$i$)
```

```
 6            elseif dirty[i] = FALSE
 7                then ENQUEUE($Q_3$,$i$)
 8                else ENQUEUE($Q_4$,$i$)
 9  if $Q_1 \neq \emptyset$                         ▷ Choosing the page to be replaced.
10     then swap-out ← DEQUEUE($Q_1$)
11     else if $Q_2$NE$\emptyset$
12           then swap-out ← DEQUEUE($Q_2$)
13           else if $Q_3 \neq \emptyset$
14                 then swap-out ← DEQUEUE($Q_3$)
15                 else swap-out ← DEQUEUE($Q_4$)
16  return swap-out
```

The space requirement of the RUN-NRU algorithm is $O(m)$ and its running time is $O(np)$. The SECOND-CHANCE algorithm is a modification of FIFO. Its main point is that if the referenced bit of the page to be replaced is FALSE according to FIFO, then we swap it out. If, however, its referenced bit is TRUE, then we set it to FALSE and put the page from the beginning of the queue to the end of the queue. This is repeated until a page is found at the be-ginning of the queue, the referenced bit of which is FALSE. A more efficient implementation of this idea is the CLOCK algorithm which stores the in-dices of the m pages in a circular list, and uses a hand to point to the next page to be replaced.

The essence of the LIFO (**L**ast **I**n **F**irst **O**ut) algorithm is that after filling in the physical memory according to the requirements we always replace the last arrived page, i. e., after the initial period there are $m-1$ pages constantly in the memory—and all the replacements are performed in the page frame with the highest address.

17.2.2. Dynamic paging

It is typical of most of the computers that there are multiple programs running simultane-ously on them. If there is paged virtual memory on these computers, it can be managed both locally and globally. In the former case each program's demand is dealt with one by one, while in the latter case a program's demand can be satisfied even at other programs' expenses. Static page replacement algorithms using local management have been discussed in the last section. Now we present two dynamic algorithms. The WS (WORKING-SET) algorithm is based on the experience that when a program is run-ning, in relatively short time there are only few of its pages needed. These pages form the working set belonging to the given time interval. This working set can be defined for example as the set of the pages needed for the last h instructions. The operation of the algorithm can be illustrated as pushing a "window" with length of h along reference array R, and keeping the pages seen through this window in the memory.

```
WS($pagetable, t, h$)

if $pagetable[r_t, 1] = $ FALSE                    ▷ The next page is not in.
   then WS-SWAP-OUT($t$)
        WRITE($pagetable[swap\text{-}out, 2], swap\text{-}out$)
        $pagetable[r_t, 1] \leftarrow$ TRUE
        $pagetable[r_t, 2] \leftarrow swap\text{-}out$
if $t > h$                                         ▷ Does $r_{t-h}$ in the memory?
   then $j \leftarrow h - 1$
        while $r_j \neq r_{t-h}$ and $j < t$
              do $j \leftarrow j + 1$
        if $j > t$
           then $pagetable[r_{t-h}, 1] \leftarrow$ FALSE
```

When discussing the WS algorithm, to make it as simple as possible, we suppose that $h \leq n$,, therefore, storing the pages seen through the window in the memory is possible even if all the h references are different (in practice, h is usually significantly bigger than n due to the many repetitions in the reference string).

The WS-Swap-Oout algorithm can be a static page replacement algorithm, for instance, which chooses the page to be replaced from all the pages in the memory—i. e., globally. If, for example, the FIFO algorithm with running time $\Theta(p)$ is used for this purpose, then the running time of the WS algorithm will be $\Theta(hp)$, since in the worst case it has to examine the pages in the window belonging to every single instruction.

The PFF (**P**age **F**requency **F**ault) algorithm uses a parameter as well. This algorithm keeps record of the number of the instructions executed since the last page fault. If this number is smaller when the next page fault occurs than a previously determined value of parameter d, then the program will get a new page frame to be able to load the page causing page fault. If, however, the number of instructions executed without any page faults reaches value d, then first all the page frames containing pages that have not been used since the last page fault will be taken away from the program, and after that it will be given a page frame for storing the page causing page fault.

```
PFF($pagetable, t, d$)

$counter \leftarrow 0$                              ▷ Preparation.
for $i \leftarrow 1$ to $n$
    do $pagetable[i, 1] \leftarrow$ FALSE
       $referenced[i] \leftarrow$ FALSE
for $j \leftarrow 1$ to $p$                         ▷ Running.
    do if $pagetable[r_t, 1] =$ TRUE
          then $counter \leftarrow counter + 1$
          else PFF-SWAP-IN($t, d, swap\text{-}out$)
               WRITE($pagetable[swap\text{-}out, 2], swap\text{-}out$)
               $pagetable[r_t, 1] \leftarrow$ TRUE
```

```
for $i \leftarrow$ to $n$
    do if $referenced[i] =$ FALSE
          then $pagetable[i, 1] \leftarrow$ FALSE
               $referenced[i] \leftarrow$ FALSE
```

Exercises

17.2-1 Consider the following reference string: $R = \langle 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6 \rangle$. How many page faults will occur when using FIFO, LRU or OPT algorithm on a computer with main memory containing k $(1 \le k \le 8)$ page frames?

17.2-2 Implement the FIFO algorithm using a pointer—instead of queue Q—pointing to the page frame of the main memory, which is the next one to load a page.

17.2-3 What would be the advantages and disadvantages of the page replacement algorithms' using an $m \times 2$ page map—besides the page table—the j-th row of which indicating whether the j-th row of the physical memory is reserved, and also reflecting its content?

17.2-4 Write and analyse the pseudo code pseudocode of Second-Chance, Clock and LIFO algorithms.

17.2-5 Is it possible to decrease the running time of the NFU algorithm (as far as its order of magnitude is concerned) if the pages are not classed only after each page faults, but the queues are maintained continuously?

17.2-6 Another version, NFU', of the NRU algorithm is also known, which uses four sets for classing the pages, and it chooses the page to be replaced from the nonempty set with the smallest index by chance. Write the pseudo code of operations In-Set and From-Set needed for this algorithm, and calculate the space requirement and running time of the NFU' algorithm.

17.2-7⋆ Extend the definition of the page replacement automat so that it would stop after processing the last entry of the finite reference sequence. *Hint.* Complete the set of incoming signs with an 'end of the sequence' sign.

17.3. Anomalies

When the first page replacement algorithms were tested in the IBM Watson Research Institute at the beginning of the 1960's, it caused a great surprise that in certain cases increasing the size of the memory leads to an increase in running time of the programs. In computer systems the phenomenon, when using more recourses leads to worse results is called anomaly. Let us give three concrete examples. The first one is in connection with the FIFO page replacement algorithm, the second one with the List-Scheduling algorithm used for processor scheduling, and the third one with parallel program execution in computers with interleaved memories.

Note that in two examples out of the three ones a very rare phenomenon can be observed, namely that the degree of the anomaly can be any large.

17.3.1. Page replacement

Let m, M, n and p be positive integers ($1 \leq m \leq n < \infty$), k a non-negative integer, $A = \{a_1, \ a_2, \ldots, a_n\}$ a finite alphabet. A^k is the set of the words over A with length k, and A^* the words over A with finite length. Let m be the number of page frames in the main memory of a small, and M a big computer. The FIFO algorithm has already been defined in the previous section. Since in this subsection only the FIFO page replacement algorithm is discussed, the sign of the page replacement algorithm can be omitted from the notations.

Let us denote the number of the page faults by $f_P(R, m)$. The event, when $M > m$ and $f_P(R, M) > f_P(R, m)$ is called ***anomaly.*** In this case the quotient $f_P(R, M)/f_P(R, m)$ is the degree of the anomaly. The efficiency of algorithm P is measured by paging speed $E_P(R, m)$ which is defined as

$$E_P(R, m) = \frac{f_P(R, m)}{p} \ , \tag{17.3}$$

for a finite reference string $R = \langle r_1, r_2, \ldots \rangle$, while for an infinite reference string $R = \langle r_1, r_2, \ldots \rangle$ by

$$E_P(R, m) = \liminf_{k \to \infty} \frac{f_P(R_k, m)}{k} \ . \tag{17.4}$$

Let $1 \leq m < n$ and let $C = (1, 2, \ldots, n)^*$ be an infinite, circular reference sequence. In this case $E_{\text{FIFO}}(C, m) = 1$.

If we process the reference string $R = \langle 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 \rangle$, then we will get 9 page faults in the case of $m = 3$, and 10 ones in the case of $m = 4$, therefore, $f_{\text{FIFO}}(R, m) = 10/9$. Bélády, Nelson and Shedler has given the following necessary and sufficient condition for the existing of the anomaly.

Theorem 17.1 *There exists a reference sequence R for which the* FIFO *page replacement algorithm causes an anomaly if, and only if $m < M < 2m - 1$.*

The following has been proved as far as the degree of the anomaly is concerned.

Theorem 17.2 *If $m < M < 2m - 1$, then for every $\epsilon > 0$ there exists a reference sequence $R = \langle r_1, r_2, \ldots, r_p \rangle$ for which*

$$\frac{f(R, M)}{f(R, m)} > 2 - \epsilon \ . \tag{17.5}$$

Bélády, Nelson and Shedler had the following conjecture.

Conjecture 17.3 *For every reference sequence R and memory sizes $M > m \geq 1$*

$$\frac{f_{\text{FIFO}}(R, M)}{f_{\text{FIFO}}(R, m)} \leq 2 \ . \tag{17.6}$$

This conjecture can be refuted e. g. by the following example. Let $m = 5, \ M = 6, \ n = 7, \ k \geq 1$, and $R = UV^k$, where $V = (1, 2, 3, 4, 5, 6, 7)^3$ and $U = \langle 1, 2,$

$3, 4, 5, 6, 7, 1, 2, 4, 5, 6, 7, 3, 1, 2, 4, 5, 7, 3, 6, 2, 1, 4, 7, 3, 6, 2, 5, 7, 3, 6, 2, 5\rangle$. If execution sequence U is executed using a physical memory with $m = 5$ page frames, then there will be 29 page faults, and the processing results in controlling status (7,3,6,2,5). After that every execution of reference sequence V causes 7 new page faults and results in the same controlling status.

If the reference string U is executed using a main memory with $M = 6$ page frames, then we get control state $\langle 2, 3, 4, 5, 6, 7\rangle$ and 14 page faults. After that every execution of reference sequence V causes 21 new page faults and results in the same control state.

Choosing $k = 7$ the degree of the anomaly will be $(14 + 7 \times 21)/(29 + 7 \times 7) = 161/78 > 2$. As we increment the value of k, the degree of the anomaly will go to three. Even more than that is true: according to the following theorem by Péter Fornai and Antal Iványi the degree of the anomaly can be any arbitrarily large.

Theorem 17.4 *For any large number L it is possible to give parameters m, M and R so that the following holds:*

$$\frac{f(R, M)}{f(R, m)} > L \, . \tag{17.7}$$

17.3.2. Scheduling with lists

Suppose that we would like to execute n ***tasks*** on p processors. By the execution the priority order of the programs has to be taken into consideration. The processors operate according to First Fit, and the execution is carried out according to a given list L. E. G. Coffman jr. wrote in 1976 that decreasing the number of processors, decreasing ***execution time*** t_i of the tasks, reducing the precedence restrictions, and altering the list can each cause an anomaly. Let the ***vector of the execution times*** of the tasks denoted by $\mathbf{t}$, the precedence relation by $<$, the list by L, and execution time of all the tasks with a common list on p equivalent processors by $C(p, L, <, \mathbf{t})$.

The degree of the anomaly is measured by the ratio of the execution time C' at the new parameters and execution time C at the original parameters. First let us show four examples for the different types of the anomaly.

Example 17.4 Consider the following task system τ_1 and its scheduling S_1 received using list $L = (T_1,\ T_2,\ \ldots,\ T_9)$ on $m = 3$ equivalent processors. In this case $C_{\max}(S_1) = 12$ (see Figure 17.1), which can be easily proved to be the optimal value.

Example 17.5 Schedule the previous task system τ_1 for $m = 3$ equivalent processors with list $L' = \langle T_1,\ T_2,\ T_4,\ T_5,\ T_6,\ T_3,\ T_9,\ T_7,\ T_8\rangle$. In this case for the resulting scheduling S_2 we get $C_{max}(S_2) = 14$ (see Figure 17.2).

Example 17.6 Schedule the task system τ_1 with list L for $m' = 4$ processors. It results in $C_{max}(S_3) = 15$ (see Figure 17.3).

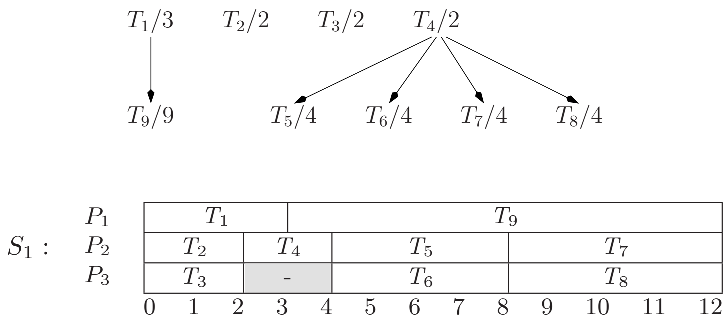


Figure 17.1 Task system τ_1, and its optimal schedule.

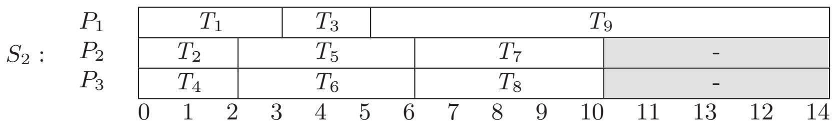


Figure 17.2 Scheduling of the task system τ_1 at list L'.

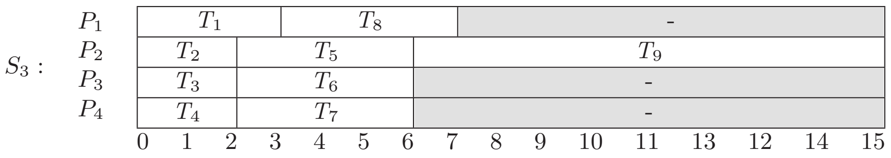


Figure 17.3 Scheduling of the task system τ_1 using list L on $m' = 4$ processors.

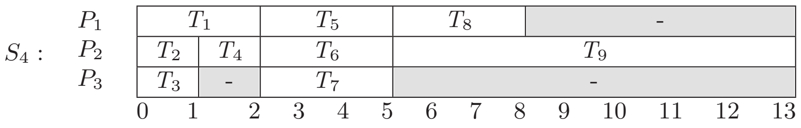


Figure 17.4 Scheduling of τ_2 with list L on $m = 3$ processors.

Example 17.7 Decrement the executing times by one in τ_1. Schedule the resulting task system τ_2 with list L for $m = 3$ processors. The result is: $C_{max}(S_4) = 13$ (see Figure 17.4).

Example 17.8 Reduce the precedence restrictions: omit edges $(T_4,\ T_5)$ and $(T_4,\ T_6)$ from the graph. The result of scheduling S_5 of the resulting task system τ_3 can be seen in Figure 17.5: $C_{\max}(S_5) = 16$.

The following example shows that the increase of maximal finishing time in the worst case can be caused not only by a wrong choice of the list.

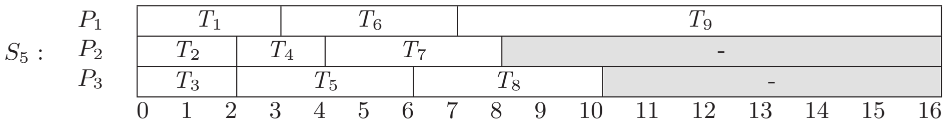
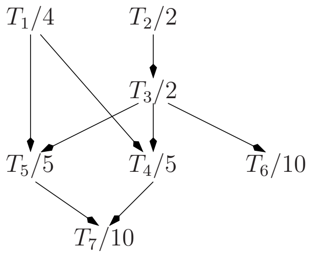


Figure 17.5 Scheduling task system τ_3 on $m = 3$ processors.

Example 17.9 Let task system τ and its optimal scheduling S_{OPT} be as showed by Figure 17.6. In this case $C_{max}(S_{\text{OPT}}) = 19$.



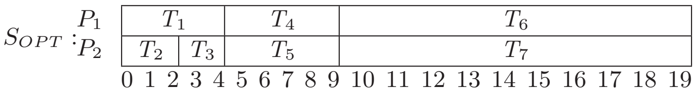


Figure 17.6 Task system τ and its optimal scheduling S_{OPT} on two processors.

We can easily prove that if the executing times are decremented by one, then in the resulting task system τ' we cannot reach a better result than $C_{max}(S_6) = 20$ with any lists (see Figure 17.7).

After these examples we give a relative limit reflecting the effects of the scheduling parameters. Suppose that for given task systems τ and τ' we have $T' = T$, $<' \subseteq <$, $\mathbf{t}' \leq \mathbf{t}$. Task system τ is scheduled with the help of list L, and τ' with L'—the former on m, while the latter on m' equivalent processors. For the resulting schedulings S and S' let $C(S) = C$ and $C(S') = C'$.

Theorem 17.5 (scheduling limit). . *With the above conditions*

$$\frac{C'}{C} \leq 1 + \frac{m-1}{m'} \ . \tag{17.8}$$

Proof Consider scheduling diagram D' for the parameters with apostrophes (for S'). Let the definition of two subsets—A and B—of the interval $[0, C')$ be the following: $A = \{t \in [0, C')|$ all the processors are busy in time $t\}$, $B = [0, C') \setminus A$. Note that both sets are unions of disjoint, half-open (closed from the left and open from the right) intervals.

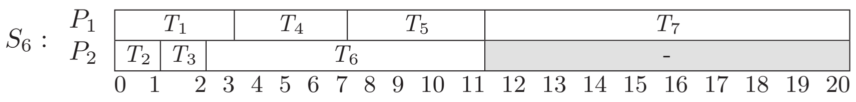


Figure 17.7 Optimal list scheduling of task system τ'.

Let T_{j_1} be a task the execution of which ends in C' instant of time according to D1 (i. e., $f_{j_1} = C'$). In this case there are two possibilities: Starting point s_{j_1} of task T_{j_1} is either an inner point of B, or not.

1. If s_{j_1} is an inner point of B, then according to the definition of B there is a processor for which with $\varepsilon > 0$ it holds that it does not work in the interval $[s_{j_1} - \varepsilon, s_{j_1})$. This can only occur if there is a task T_{j_2} for which $T_{j_2} <' T_{j_1}$ and $f_{j_2} = s_{j_1}$ (case a).

2. If s_{j_1} is not an inner point of B, then either $s_{j_1} = 0$ (case b), or $s_{j_1} > 0$. If B has got a smaller element than s_{j_1} (case c), then let $x_1 = \sup\{x \mid x < s_{j_1} \ \textit{and}\ x \in B\}$, else let $x_1 = 0$ (case d). If $x_1 > 0$, then it follows from the construction of A and B that there is a processor for which a task T_{j_2} can be found the execution of which is still in progress in this time interval, and for which $T_{j_2} <' T_{j_1}$.

Summarising the two cases we can say that either there is a task $T_{j_2} <' T_{j_1}$ for which in the case of $y \in [f_{j_2}, s_{j_1})$ holds $y \in A$ (case a or c), or for every number $x < s_{j_1}$ $x \in A$ or $x < 0$ holds (case a or d).

Repeating this procedure we get a task chain $T_{j_r}, T_{j_{r-1}}, \ldots, T_{j_1}$ for which it holds that in the case of $x < s_{j_r}$ either $x \in A$ or $x < 0$. This proves that there are tasks for which

$$T_{j_r} <' T_{j_{r-1}} <' \cdots <' T_{j_1} \ , \tag{17.9}$$

and in every instant of time t there is a processor which is working, and is executing one of the elements of the chain. It yields

$$\sum_{\phi \in S'} t'(\phi) \le (m' - 1) \sum_{k=1}^{r} t'_{j_k} \ , \tag{17.10}$$

where f denotes the empty periods, so the sum on the left hand side refers to all the empty periods in S'.

Based on (11.9) and $<' \subseteq <$, therefore,

$$C \ge \sum_{k=1}^{r} t_{j_k} \ge \sum_{k=1}^{r} t'_{j_k} \ . \tag{17.11}$$

Since

$$mC \ge \sum_{i=1}^{n} t_i \ge \sum_{i=1}^{n} t'_i \ , \tag{17.12}$$

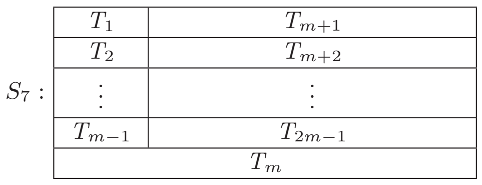


Figure 17.8 Scheduling $S_7(\tau_4)$ belonging to list $L = (T_1, \ldots, T_n)$.

and

$$C' = \frac{1}{m'}\left(\sum_{i=1}^{n} t'_i + \sum_{\phi \in S'} t'(\phi)\right) ,$$

így (17.10), (17.11) and (17.12)

$$C' \le \frac{1}{m'}\Big(mC + (m'-1)C\Big) ,$$

based on (17.10), (17.11) and (17.12) we get

$$C' \le \frac{1}{m'}\Big(mC + (m'-1)C\Big) ,$$

implying $C'/C \le 1 + (m-1)/m'$. ■

The following examples show us not only that the limit in the theory is the best possible, but also that we can get the given limit (at least asymptotically) by altering any of the parameters.

Example 17.10 In this example the list has changed, $<$ is empty, m is arbitrary. Execution times are the following:

$$t_i = \begin{cases} 1, & \text{if } i = 1, \ldots, m-1 , \\ m, & \text{if } i = m , \\ m-1, & \text{if } i = m+1, \ldots, 2m-1 . \end{cases}$$

If this task system τ_4 is scheduled for m processors with list $L = (T_1, \ldots, T_{2m-1}n)$, then we get the optimal scheduling $S_7(\tau_4)$ which can be seen in Figure 17.8.

If we use the list $L' = (T_{m+1}, \ldots, T_{2m-1}, T_1, \ldots, T_{m-1}, T_m)$ instead of list L, then we get scheduling $S_8(\tau_4)$ which can be seen in Figure 17.9.

In this case $C = (S_7) = m$, $C'(S_8) = 2m - 1$, therefore $C'/C = 2 - 1/m$; which means that altering the list results in the theorem holding with equality, i.e., the expression on the right hand side of the $\le$ sign cannot be decreased.

Example 17.11 In this example we decrease the execution times. We use list $L = L' = \langle T_1, \ldots, T_{3m}\rangle$. in both cases. Here as well as in the remaining part of the chapter ε denotes

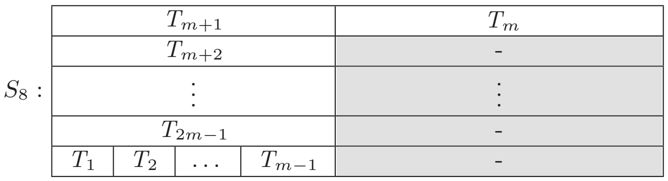


Figure 17.9 Scheduling $S_8(\tau_4)$ belonging to list L'.

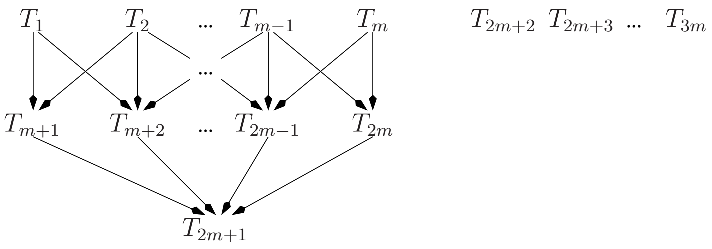


Figure 17.10 Identical graph of task systems τ_5 and τ_5'.

an arbitrarily small positive number. Original execution times are stored in vector $\mathbf{t} = (t_1, \dots, t_n)$, where

$$t_i = \begin{cases} 2\varepsilon, & \text{if } i = 1, \dots, m \ , \\ 1, & \text{if } i = m+1, \dots, 2m \ , \\ m-1, & \text{if } i = 2m+1, \dots, 3m \ . \end{cases}$$

The new execution times are

$$t_i' = \begin{cases} t_i - \varepsilon, & \textit{if } i = 1, \dots, m-1 \ , \\ t_i, & \textit{if } i = m, \dots, 3m \ . \end{cases}$$

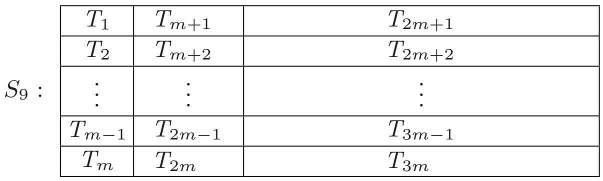
The precedence graph of task system τ_5, and its modification τ_5' are shown in Figure 17.10, while optimal scheduling $S_9(\tau_5)$ and scheduling $S_{10}(\tau_5')$ can be seen in Figure 17.11. Here $C = C_{\max}(S_9(\tau_5)) = m + 2\varepsilon$ and $C' = C_{\max}(S_{10}(\tau_5') = 2m - 1 + \varepsilon \mathrm{C} = \mathrm{Cmax}$, therefore, increasing ε C'/C goes to value $2 - 1/m$ ($\lim_{\varepsilon \to 0} C'/C = 2 - 1/m$). This means that altering the execution times we can approach the limit in the theorem arbitrarily closely.

Example 17.12 In this example we reduce the precedence restrictions. The precedence graph of task system τ_6 is shown in Figure 17.12.

The execution times of the tasks are: $t_1 = \varepsilon$, $t_i = 1$, if $i = 1, \dots, m^2 - m + 1$, and $t_{m^2-m+2} = m$. The optimal scheduling $S_{11}(\tau_6)$ of τ_6 belonging to list $L = (T_1, \dots, T_{m^2-m+2})$ can be seen in Figure 17.13.

Omitting all the precedence restrictions from τ_6 we get the task system τ_6'. Scheduling $S_{12}(\tau_6')$ is shown in Figure 17.14.

Example 17.13 This time the number of the processors will be increased from m to m'.


S_9 :
T_1 T_{m+1} T_{2m+1}
T_2 T_{m+2} T_{2m+2}
T_{m-1} T_{2m-1} T_{3m-1}
T_m T_{2m} T_{3m}

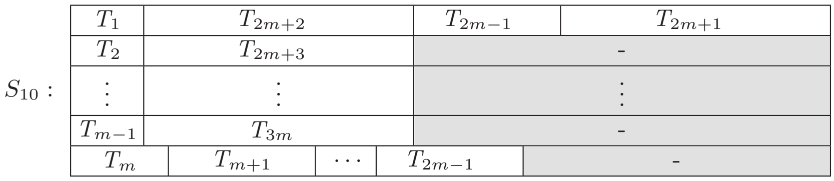

S_{10} :
T_1 T_{2m+2} T_{2m-1} T_{2m+1}
T_2 T_{2m+3} -
T_{m-1} T_{3m} -
T_m T_{m+1} $\cdots$ T_{2m-1} -

Figure 17.11 Schedulings $S_9(\tau_5)$ and $S_{10}(\tau_5')$.

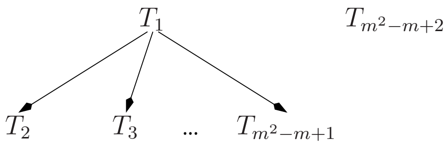


Figure 17.12 Graph of the task system τ_6.

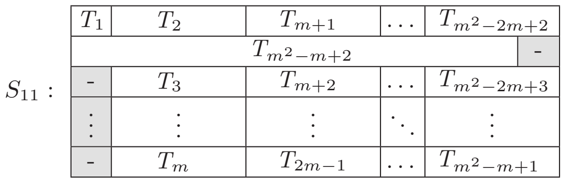

S_{11} :
T_1 T_2 T_{m+1} $\cdots$ T_{m^2-2m+2}
T_{m^2-m+2} -
- T_3 T_{m+2} $\cdots$ T_{m^2-2m+3}
- T_m T_{2m-1} $\cdots$ T_{m^2-m+1}

Figure 17.13 Optimal scheduling $S_{11}(\tau_6)$.

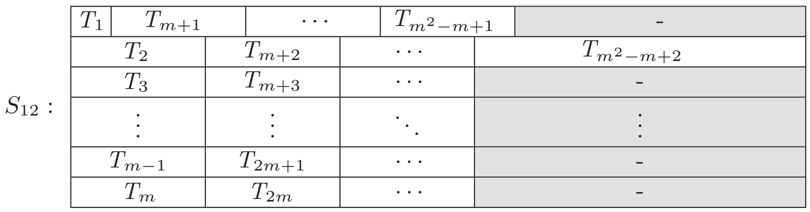

S_{12} :
T_1 T_{m+1} $\cdots$ T_{m^2-m+1} -
T_2 T_{m+2} $\cdots$ T_{m^2-m+2}
T_3 T_{m+3} $\cdots$ -
T_{m-1} T_{2m+1} $\cdots$ -
T_m T_{2m} $\cdots$ -

Figure 17.14 Scheduling $S_{12}(\tau_6')$.

The graph of task system τ_7 is shown by Figure 17.15, and the running times are

$$t_i = \begin{cases} \varepsilon, & \text{if } i = 1, \ldots, m+1 \,, \\ 1, & \text{if } i = m+2, \ldots, mm' - m' + m + 1 \,, \\ m', & \text{if } i = mm' - m' + m + 2 \,. \end{cases}$$

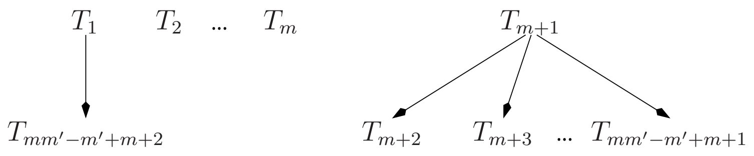


Figure 17.15 Precedence graph of task system τ_7.

S_{13} :

T_1	T_{m+1}	T_{m+2}	$\cdots$	T_a	
T_2	$T_{mm'-m'+2}$				-
T_3	-	T_{m+3}	$\cdots$	T_b	
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
T_m	-	T_{2m}	$\cdots$	T_c	

Figure 17.16 The optimal scheduling $S_{13}(\tau_7)$ ($a = mm'-m'+3$, $b = a+1$, $c = mm'-m'+m+1$).

S_{14} :

T_1	T_{m+2}	$\cdots$	T_a	$T_{mm'-m'+m+2}$
T_2	T_{m+3}	$\cdots$	T_{a+1}	-
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
T_b	T_c	$\cdots$	T_d	-
-	$\vdots$	$\ddots$	$\vdots$	$\vdots$
-	T_e	$\cdots$	T_f	-

Figure 17.17 The optimal scheduling $S_{14}(\tau_7)$ ($a = mm' - 2m' + m + 2$, $b = m + 1$, $c = 2m + 2$, $d = mm' - 2m' + 2m + 2$, $e = m + m' + 1$, $f = mm' - m' + m + 1$).

The optimal scheduling of the task system on m, and m' processors is shown by Figure 17.16 and Figure 17.17.

Comparing the $C = C_{\max}(S_{13}(\tau_7)) = m' + 2\varepsilon$, and $C' = C_{\max}(S_{14}(\tau_7)) = m' + m - 1 + \varepsilon$ maximal finishing times we get the ratio $C'/C = 1 + (m - 1 - \varepsilon)(m' + 2\varepsilon)$ and so again the required asymptotic value: $\lim_{\varepsilon \to 0} C'/C = 1 + (m-1)/m'$

With the help of these examples we proved the following statement.

Theorem 17.6 (sharpness of the scheduling limit). *The limit given for the relative speed (11.8) is asymptotically sharp for the changing of (any of the) parameters m, t, $<$ and L.*

17.3.3. Parallel processing with interleaved memory

We describe the parallel algorithm modelling the operating of computers with interleaved memory in a popular way. The sequence of dumplings is modelling the reference string, the giants the processors and the bites the commands executed simultaneously. Dwarfs $D_0, D_1, \ldots, D_r$ $(r \geq 0)$ cook dumplings of n different types. Every dwarf creates an infinite sequence of dumplings.

These sequences are usually given as random variables with possible values 1, 2, $\ldots, n$. For the following analysis of the extreme cases deterministic sequences are used.

The dumplings eating giants G_b $(b = 1, 2, \ldots)$ eat the dumplings. The units of the eating are the bits.

The appetite of the different giants is characterised by the parameter b. Giant $]_b$ is able to eat up most b dumplings of the same sort at one bite.

Giant G_b eats the following way. He chooses for his first bite from the beginning from the beginning of the dumpling sequence of dwarf T_0 so many dumplings, as possible (at most b of the same sort), and he adds to these dumplings so many ones from the beginning of the sequences of the dwarfs $D1$, D_2, $\ldots$, as possible.

After assembling the first bite the giant eats it, then he assembles and eats the second, third, $\ldots$ bites.

Example 17.14 To illustrate the model let us consider an example. We have two dwarfs (D_0 and D_1) and the giant G_2. The dumpling sequences are

$$\begin{array}{l} 12121233321321321 \\ 24444444, \end{array} \tag{17.13}$$

or in a shorter form

$$\begin{array}{l} (12)^(3)^2(321)^* \\ 2(4)^*, \end{array} \tag{17.14}$$

where the star (*) denotes a subsequence repeated infinitely many times.

For his first bite G_2 chooses from the first sequence the first four dumlings 1212 (because the fifth dumpling is the third one of the sort 1) and no dumpling from the second sequence (because the beginning element is 2, and two dumplings of this sort is chosen already). The second bite contains the subsequence 1233 from the first sequence, and the dumplings 244 from the second one. The other bites are identical: 321321 from the first sequence and 44 from the second one. In a short form the bites are as follows:

$$\begin{array}{llll} \|1212 & \|1233 & \|321321 & \|^* \\ \| - - & \|244 & \|44 & \| \end{array} \tag{17.15}$$

(bites are separated by double lines).

For given dumpling sequences and a given giant G_b let B_t $(t = 1, 2, \ldots)$ denote the number of dumplings in the t-th bite. According to the eating-rules $b \leq B_t \leq bn$ holds for every t.

Considering the elements of the dumpling sequences as random variables with possible values 1, 2, $\ldots$,n and given distribution we define the dumpling-eating speed S_b (concerning the given sequences) of G_b as the average number of dumplings in one bite for a long time, more precisely

$$S_b = \liminf_{t \to \infty} \mathbf{E}\left(\frac{\sum_{i=1}^{t} B_i}{t}\right), \tag{17.16}$$

where $\mathbf{E}(\xi)$ denotes the expected value of the random variable ξ.

One can see that the defined limit always exists.

Maximal and minimal speed-ratio Let us consider the case, when we have at least one dumpling sequence, at least one type of dumplings, and two different giants, that is let $r \geq 0,\ n \geq 1,\ b > c \geq 1$. Let the sequences be deterministic.

Since for every bite-size B_t $(t = 1, 2, \ldots)$ of G_b holds $b \leq B_t \leq bn$, the same bounds are right for every average value $(\sum B_i)_{i=1}^{t})/t$ and for every expected value $E((\sum_{i=1}^{t} B_i)/t)$, too. From this it follows, that the limits S_b and S_c defined in (17.16) also must lie between these bounds, that is

$$b \leq S_b \leq bn, \quad g \leq S_c \leq cn \ . \tag{17.17}$$

Choosing the maximal value of S_b and the minimal value of S_c and vice versa we get the following trivial upper and lower bounds for the speed ratio S_b/S_c:

$$\frac{b}{cn} \leq \frac{S_b}{S_c} \leq \frac{bn}{c} \ . \tag{17.18}$$

Now we show that in many cases these trivial bounds cannot be improved (and so the dumpling eating speed of a small giant can be any times bigger than that of a big giant).

Theorem 17.7 *If* $r \geq 1,\ n \geq 3,\ b > c \geq 1,$ *then there exist dumpling sequences, for which*

$$\frac{S_b}{S_c} = \frac{b}{cn} \ , \tag{17.19}$$

further

$$\frac{S_b}{S_c} = \frac{bn}{c} \ . \tag{17.20}$$

Proof To see the sharpness of the lower limit in the inequality (17.18) giving the natural limits let consider the following sequences:

$$\begin{array}{l} 1^b 2^{2b+1} 1^* \\ 1^{b+1} (23 \ldots n)^* \ . \end{array} \tag{17.21}$$

Giant G_b eats these sequences in the following manner:

$$\begin{array}{lllll} \|1^b 2^b & \|2^b & \|21^b & \|1^b & \|^* \\ \| - - & \|1^b & \| - - & \| - - & \|. \end{array} \tag{17.22}$$

Here $B_1 = 2b,\ B_2 = 2b,\ B_3 = b + 1,\ B_t = b$ (for $t = 4, 5, \ldots$.

For the given sequences we have

$$S_b = \lim_{t \to \infty} \frac{2b + 1 + tb}{t} = b. \tag{17.23}$$

G_c eats these sequences as follows:

$$\begin{array}{llllllll} \|1^c & \|^{\alpha}\ 1^{c_1}2^c & \|2^c & \|^{\beta}\ 2^c & \|2^c & \|^{\gamma}\ 2^{c_4}1^c & \|1^c & \|^* \\ \|\,-\,- & \|1^{c_2} & \|1^c & \|1^{c_3} & \|\,-\,- & \|(23\dots)^{c_3} & \|(23\dots)^c & \|. \end{array} \qquad (17.24)$$

Here

$$\alpha = \left\lceil \frac{b-c}{c} \right\rceil ; \; c_1 = b - \alpha c; \; c_2 = c - c_1 \; ;$$

$$\beta = \left\lceil \frac{b+1-c_2-c}{c} \right\rceil ; \; c_3 = b+1-c_2-\beta c \; ;$$

$$\gamma = \left\lceil \frac{2b+1-c(\beta+2)}{c} \right\rceil ; \; c_4 = 2b+1-c(\beta+\gamma+2) \; ;$$

$$c_5 = c - c_4 .$$

In this case we get (in a similar way, as we have got S_b)

$$S_c = cn \qquad (17.25)$$

and therefore

$$\frac{S_b}{S_c} = \frac{b}{cn} \; . \qquad (17.26)$$

In order to derive the exact upper bound, we consider the following sequence:

$$\begin{array}{l} 12^{2b+1}1^* \\ 1^{b-1}3^{2b}1^b(23\dots n)^* \; . \end{array} \qquad (17.27)$$

G_b eats these sequences as follows:

$$\begin{array}{lllll} \|12^b & \|2^b & \|21^b & \|1^b & \|^* \\ \|1^{b-1}3^b & \|3^b1^b & \|(23\dots)^{b-1} & \|(23\dots n)^b & \| \; . \end{array} \qquad (17.28)$$

From here we get

$$S_b = \lim_{t\to\infty} \frac{3b+3b+n(b-1)+2+(t-3)bn}{t} = bn \; . \qquad (17.29)$$

G_c'c eating is characterised by

$$\begin{array}{lllllllll} \|12^c & \|2^c & \|\alpha 2^c & \|2^c & \|\beta\ 2^{c_3}1^c & \|1^c & \|1^c & \|1^c & \|^* \\ \|1^{c_1} & \|1^c & \|1^{c_2}3^c & \|3^c & \|3^c & \|3^c & \|3^{c_4} & \|\,-\,- & \| \; , \end{array} \qquad (17.30)$$

where

$$c_1 = c-1; \; \alpha = \left\lceil \frac{b-c-c_1}{c} \right\rceil ; \; c_2 = b-1-c_1-\alpha c \; ;$$

$$\beta = \left\lceil \frac{2b+1-c(\alpha+\beta)}{c} \right\rceil ; \gamma = \left\lceil \frac{2b-c(\beta+2)}{c} \right\rceil \quad ;$$

$$c_4 = 2b - c(\beta + \gamma + 2)c_3 = 2b + 1 - c(\alpha + \beta + 2)\ .$$

Since $B_t = c$ for $t = \alpha + \beta + \gamma + 5,\ t = \alpha + \beta + \gamma + 6, \ldots,$ therefore $S_c = c$, and so $S_b/S_c = bn/c$. ■

$$\alpha = \left\lceil \frac{b-c}{c} \right\rceil;\ c_1 = b - \alpha c;\ c_2 = c - c_1\ ;$$

$$\beta = \left\lceil \frac{b+1-c_2-c}{c} \right\rceil;\ c_3 = b + 1 - c_2 - \beta c\ ;$$

$$\gamma = \left\lceil \frac{2b+1-c(\beta+2)}{c} \right\rceil;\ c_4 = 2b + 1 - c(\beta + \gamma + 2)\ ;$$

$$c_5 = c - c_4.$$

17.3.4. Avoiding the anomaly

We usually try to avoid anomalies.

For example at page replacing the sufficient condition of avoiding it is that the replacing algorithm should have the stack property: if the same reference string is run on computers with memory sizes of m and $m+1$, then after every reference it holds that the bigger memory contains all the pages that the smaller does. At the examined scheduling problem it is enough not to require the scheduling algorithm's using a list.

Exercises

17.3-1 Give parameters m, M, n, p and R so that the FIFO algorithm would cause at least three more page faults with a main memory of size M than with that of size m.

17.3-2 Give such parameters that using scheduling with list when increasing the number of processors the maximal stopping time increases at least to half as much again.

17.3-3 Give parameters with which the dumpling eating speed of a small giant is twice as big as that of a big giant.

17.4. Optimal file packing

In this section we will discuss a memory managing problem in which files with given sizes have to be placed onto discs with given sizes. The aim is to minimise the number of the discs used. The problem is the same as the bin-packing problem that can be found among the problems in Section *Approximation algorithms* in the book titled *Introduction to Algorithms*. Also scheduling theory uses this model in connection with minimising the number of processors. There is the number n of the files given, and array vector $\mathbf{t} = (t_1, t_2, \ldots, t_n)$ containing the sizes of the files to be stored, for the elements of which $0 < t_i \leq 1$ holds $(i = 1, 2, \ldots, n)$. The files have to be placed onto the discs taking into consideration that they cannot be divided and the capacity of the discs is a unit.

17.4.1. Approximation algorithms

The given problem is NP-complete. Therefore, different approaching algorithms are used in practice. The input data of these algorithms are: the number n of files, a vector $\mathbf{t} = \langle t_1, t_2, \dots, t_n \rangle$ with the sizes of the files to be placed. And the output data are the number of discs needed (discnumber) and the level array $h = (h_1, h_2, \dots, h_n)$ of discs.

Linear Fit (LF) According to **L**inear **F**it file F_i is placed to disc D_i. The pseudocode of LF is the following.

```
LF($n$, $\mathbf{t}$)
1  for $i \leftarrow 1$ to $n$
2      do $h[i] \leftarrow t[i]$
3  $number\text{-}of\text{-}discs \leftarrow n$
4  return $number\text{-}of\text{-}discs$
```

Both the running time and the place requirement of this algorithm are $O(n)$. If, however, reading the sizes and printing the levels are carried out in the loop in rows 2–3, then the space requirement can be decreased to $O(1)$.

Next Fit (NF) **N**ext **F**it packs the files onto the disc next in line as long as possible. Its pseudocode is the following.

```
NF($n$, $\mathbf{t}$)
1  $h[1] \leftarrow t[1]$
2  $number\text{-}of\text{-}discs \leftarrow 1$
3  for $i \leftarrow 2$ to $n$
4      do if $h[number\text{-}of\text{-}discs] + t[i] \leq 1$
5            then $h[number\text{-}of\text{-}discs] \leftarrow h[number\text{-}of\text{-}discs] + t[i]$
6            else $number\text{-}of\text{-}discs \leftarrow number\text{-}of\text{-}discs + 1$
7                 $h[number\text{-}of\text{-}discs] \leftarrow t[i]$
8  return $number\text{-}of\text{-}discs$
```

Both the running time and the place requirement of this algorithm are $O(n)$. If, however, reading the sizes and taking the levels out are carried out in the loop in rows 3–6, then the space requirement can be decreased to $O(1)$, but the running time is still $O(n)$.

First Fit (FF) **F**irst **F**it packs each files onto the first disc onto which it fits.

FF(n, **t**)

```
number-of-discs ← 1
for $i ← 1$ to $n$
    do $h[i] ← 0$
for $i ← 1$ to $n$
    do $k ← 1$
       while $t[i] + h[k] > 1$
             do $k ← k + 1$
       $h[k] ← h[k] + t[i]$
       if $k >$ number-of-discs
         then number-of-discs ← number-of-discs + 1
return number-of-discs
```

The space requirement of this algorithm is $O(n)$, while its time requirement is $O(n^2)$. If, for example, every file size is 1, then the running time of the algorithm is $\Theta(n^2)$.

Best Fit (BF) **B**est **F**it places each file onto the first disc on which the remaining capacity is the smallest.

BF(n, **t**)

```
number-of-discs ← 1
for $i ← 1$ to $n$
    do $h[i] ← 0$
for $i ← 1$ to $n$
    do free ← 1.0
       ind ← 0
       for $k ← 1$ to number-of-discs
           doif $h[k] + t[i] \le 1$ and $1 - h[k] - t[i] <$ free
                then ind ← $k$
                     szabad ← $1 - h[k] - t[i]$
    if ind $> 0$
       then $h[$ind$] ← h[$ind$] + t[i]$
       else number-of-discs ← number-of-discs + 1
            $h[$number-of-discs$] ← t[i]$
return number-of-discs
```

The space requirement of this algorithm is $O(n)$, while its time requirement is $O(n2)$.

Pairwise Fit (PF) Pairwise Fit creates a pair of the first and the last element of the array of sizes, and places the two files onto either one or two discs—according to the sum of the two sizes. In the pseudocode there are two auxiliary variables: bind is the index of the first element of the current pair, and eind is the index of the second element of the current pair.

```
PF($n, \mathbf{t}$)
 number-of-discs ← 0
 beg-ind ← 1
 end-ind ← n
 while end-ind ≥ beg-ind
       do if end-ind − beg-ind ≥ 1
             then if t[beg-ind] + t[end-ind] > 1
                     then number-of-discs ← number-of-discs + 2
                          h[number-of-discs − 1] ← t[bind]
                          h[number-of-discs] ← t[eind]
                     else number-of-discs ← number-of-discs + 1
                  h[number-of-discs] ← t[beg-ind] + t[eind]
          if end-ind = beg-ind
             then number-of-discs ← number-of-discs + 1
                  h[number-of-discs] ← t[end-ind]
          beg-ind ← beg-ind + 1
          end-ind ← end-ind − 1
 return number-of-discs
```

The space requirement of this algorithm is $O(n)$, while its time requirement is $O(n^2)$. If, however, reading the sizes and taking the levels of the discs out are carried out online, then the space requirement will only be $O(1)$.

Next Fit Decreasing (NFD) The following five algorithms consist of two parts: first they put the tasks into decreasing order according to their executing time, and then they schedule the ordered tasks. **N**ext **F**it **D**ecreasing operates according to NF after ordering. Therefore, both its space and time requirement are made up of that of the applied ordering algorithm and NF.

First Fit Decreasing (FFD) First Fit Decreasing operates according to First Fit (FF) after ordering, therefore its space requirement is $O(n)$ and its time requirement is $O(n^2)$.

Best Fit Decreasing (BFD) **B**est **F**it **D**ecreasing operates according to Best Fit (BF) after ordering, therefore its space requirement is $O(n)$ and its time requirement is $O(n2)$.

Pairwise Fit Decreasing (PFD) **P**airwise **F**it **D**ecreasing creates pairs of the first and the last tasks one after another, and schedules them possibly onto the same processor (if the sum of their executing time is not bigger than one). If it is not possible, then it schedules the given pair onto two processors.

Quick Fit Decreasing (QFD) **Q**uick **F**it **D**ecreasing places the first file after ordering onto the next empty disc, and then adds the biggest possible files (found from the end of the ordered array of sizes) to this file as long as possible. The auxiliary variables used in the pseudocode are: bind is the index of the first file to be examined, and eind is the index of the last file to be examined.

```
QFD($n$, $\mathbf{s}$)
beg-ind ← 1
end-ind ← $n$
number-of-discs ← 0
while end-ind $\geq$ beg-ind
      do number-of-discs ← number-of-discs $+ 1$
         $h$[number-of-discs] ← $s$[bind]
         beg-ind ← beg-ind $+ 1$
         while end-ind $\geq$ beg-ind and $h$[number-of-discs] $+ s$[eind] $\leq 1$
               do ind ← end-ind
                  while ind $>$ beg-ind and $h$[number-of-discs] $+ s$[ind $- 1$] $\leq 1$
                        do ind ← ind $- 1$
h[number-of-discs] ← h[number-of-discs] $+ s$[ind]
if end-ind $>$ ind
   then for $i$ ← ind to end-ind $- 1$
            do $s[i] \leftarrow s[i+1]$
end-ind ← end-ind $- 1$
return number-of-discs
```

The space requirement of this program is $O(n)$, and its running time in worst case is $\Theta(n_2)$, but in practice—in case of executing times of uniform distribution—it is $(n \lg n)$.

17.4.2. Optimal algorithms

Simple Power (SP) This algorithm places each file—independently of each other—on each of the n discs, so it produces n^n placing, from which it chooses an optimal one. Since this algorithm produces all the different packing (supposing that two placing are the same if they allocate the same files to all of the discs), it certainly finds one of the optimal placing.

Factorial Algorithm (FACT) This algorithm produces the permutations of all the files (the number of which is $n!$), and then it places the resulted lists using NF.

The algorithm being optimal can be proved as follows. Consider any file system and its optimal packing is $S_{\text{OPT}}(t)$. Produce a permutation P of the files based on $S_{\text{OPT}}(t)$ so that we list the files placed onto $P_1, P_2, \ldots, P_{\text{OPT}}(t)$ respectively. If permutation P is placed by NF algorithm, then we get either S_{OPT} or another optimal placing (certain tasks might be placed onto processors with smaller indices).

Quick Power (QP) This algorithm tries to decrease the time requirement of SP by placing 'large' files (the size of which is bigger than 0.5) on separate discs, and tries to place only the others (the 'small' ones) onto all the n discs. Therefore, it produces only n^K placing instead of n^n, where K is the number of small files.

Economic Power (EP) This algorithm also takes into consideration that two small files always fit onto a disc—besides the fact that two large ones do not fit. Therefore, denoting the number of large files by N and that of the small ones by K it needs at most $N + (K + 1)/2$ discs. So first we schedule the large discs to separate discs, and then the small ones to each of the discs of the number mentioned above. If, for instance, $N = K = n/2$, then according to this we only have to produce $(0.75n)^{0.5n}$.

17.4.3. Shortening of lists (SL)

With certain conditions it holds that list $\mathbf{t}$ can be split into lists $\mathbf{t}_1$ and $\mathbf{t}_2$ so that $\mathrm{OPT}(\mathbf{t}_1) + \mathrm{OPT}(\mathbf{t}_2) \leq \mathrm{OPT}(\mathbf{t})$ (in these cases the formula holds with equality). Its advantage is that usually shorter lists can be packed optimally in a shorter time than the original list. For example, let us assume that $t_i + t_j = 1$. Let $\mathbf{t}_1 = (t_i, t_j)$ and $\mathbf{t}_2 = \mathbf{t} \setminus \mathbf{t}_1$. In this case $\mathrm{OPT}(\mathbf{t}_1) = 1$ and $\mathrm{OPT}(\mathbf{t}_2) = \mathrm{OPT}(\mathbf{t}) - 1$. To prove this, consider the two discs onto which the elements of list $\mathbf{t}_1$ have been packed by an optimal algorithm. Since next to them there can be files whose sum is at most $1 - t_1$ and $1 - t_2$, their executing time can sum up to at most $2 - (t_1 + t_2)$, i.e., 1. Examining the lists on both ends at the same time we can sort out the pairs of files the sum of whose running time is 1 in $O(n)$. After that we order the list $\mathbf{t}$. Let the ordered list be $\mathbf{s}$. If, for example $s_1 + s_n < 1$, then the first file will be packed onto a different disc by every placing, so $\mathbf{t}_1 = (t_1, t_j)$ and $\mathbf{t}_2 = \mathbf{t} \setminus \mathbf{t}_1$ is a good choice. If for the ordered list $s_1 + s_n < 1$ and $s_1 + s_{n-1} + s_n > 1$ hold, then let s_j be the largest element of the list that can be added to s_1 without exceeding one. In this case with choices $\mathbf{t}_1 = (t_1, t_j)$ and $\mathbf{t}_2 = \mathbf{t} \setminus \mathbf{t}_1$ list $\mathbf{t_2}$ is two elements shorter than list $\mathbf{t}$. With the help of the last two operations lists can often be shortened considerably (in favourable case they can be shortened to such an extent that we can easily get the optimal number of processors for both lists). Naturally, the list remained after shortening has to be processed—for example with one of the previous algorithms.

17.4.4. Upper and lower estimations (ULE)

Algorithms based on upper and lower estimations operate as follows. Using one of the approaching algorithms they produce an upper estimation $\mathrm{A}(\mathbf{t})$ of $\mathrm{OPT}(\mathbf{t})$, and then they give a lower estimation for the value of $\mathrm{OPT}(\mathbf{t}$ as well. For this—among others—the properties of packing are suitable, according to which two large files cannot be placed onto the same disc, and the sum of the size cannot be more than 1 on any of the discs. Therefore, both the number of the large files and the sum of the size of the files, and so also their maximum $\mathrm{MAX}(\mathbf{t})$ is suitable as a lower estimation. If $\mathrm{A}(\mathbf{t} = \mathrm{MAX}(\mathbf{t})$, then algorithm A produced an optimal scheduling. Otherwise it can be continued with one of the time-consuming optimum searching algorithms.

	LF	NF	FF	BF	PF	NFD	FFD	BFD	PFD	QFD	OPT
$\mathbf{t}_1$	4	3	3	3	3	3	2	2	2	2	2
$\mathbf{t}_2$	6	2	2	2	3	3	3	3	3	3	2
$\mathbf{t}_3$	7	3	2	3	4	3	2	3	4	2	2
$\mathbf{t}_4$	8	3	3	2	4	3	3	2	4	3	2
$\mathbf{t}_5$	5	3	3	3	3	2	2	2	3	2	2
$\mathbf{t}_6$	4	3	2	2	2	3	2	2	2	2	2
$\mathbf{t}_7$	4	3	3	3	2	3	2	2	2	2	2

Figure 17.18 Summary of the numbers of discs.

17.4.5. Pairwise comparison of the algorithms

If there are several algorithms known for a scheduling (or other) problem, then a simple way of comparing the algorithms is to examine whether the values of the parameters involved can be given so that the chosen output value is more favourable in the case of one algorithm than in the case of the other one.

In the case of the above discussed placing algorithm the number of processors discs allocated to size array $\mathbf{t}$ by algorithm A and B is denoted by A($\mathbf{t}$ and B($\mathbf{t}$, and we examine whether there are arrays $\mathbf{t}_1$ and $\mathbf{t}_2$ for which $\mathrm{A}(\mathbf{t}_1) < \mathrm{B}(\mathbf{t}_1)$ and $\mathrm{A}(\mathbf{t}_2) > \mathrm{B}(\mathbf{t}_2)$ hold. We answer this question in the case of the above defined ten approaching algorithms and for the optimal one. It follows from the definition of the optimal algorithms that for each $\mathbf{t}$ and each algorithm A holds OPT($\mathbf{t} \leq$ A($\mathbf{t}$). In the following the elements of the arrays in the examples will be twentieth.

Consider the following seven lists:

$$\begin{aligned}
\mathbf{t_1} &= (12/20, 6/20, 8/20, 14/20),\\
\mathbf{t_2} &= (8/20, 6/20, 6/20, 8/20, 6/20, 6/20),\\
\mathbf{t_3} &= (15/20, 8/20, 8/20, 3/20, 2/20, 2/20, 2/20),\\
\mathbf{t_4} &= (14/20, 8/20, 7/20, 3/20, 2/20, 2/20, 2/20, 2/20),\\
\mathbf{t_5} &= (10/20, 8/20, 10/20, 6/20, 6/20),\\
\mathbf{t_6} &= (12/20, 12/20, 8/20, 8/20),\\
\mathbf{t_7} &= (8/20, 8/20, 12/20, 12/20).
\end{aligned}$$

The packing results of these lists are summarised in Figure 17.18.

As shown in Figure 17.18, LF needs four discs for the first list, while the others need fewer than that. In addition, the row of list $\mathbf{t}_1$ shows that FFD, BFD, PFD, QFD and OPT need fewer discs than NF, FF, BF, PF and NFD. Of course, there are no lists for which any of the algorithms would use fewer discs than OPT. It is also obvious that there are no lists for which LF would use fewer discs than any of the other ten algorithms.

These facts are shown in Figure 17.19. In the figure symbols X in the main diagonal indicate that the algorithms are not compared to themselves. Dashes in the first column indicate that for the algorithm belonging to the given row there is no list which would be processed using more disc by this algorithm than by the algorithm belonging to the given column, i.e., LF. Dashes in the last column show that there

	LF	NF	FF	BF	PF	NFD	FFD	BFD	PFD	QFD	OPT
LF	X	1	1	1	1	1	1	1	1	1	1
NF	–	X					1	1	1	1	1
FF	–		X				1	1	1	1	1
BF	–			X			1	1	1	1	1
PF	–				X		1	1	1	1	1
NFD	–					X	1	1	1	1	1
FFD	–						X				
BFD	–							X			
PFD	–								X		
QFD	–									X	
OPT	–	–	–	–	–	–	–	–	–	–	X

Figure 17.19 Pairwise comparison of algorithms.

	LF	NF	FF	BF	PF	NFD	FFD	BFD	PFD	QFD	OPT
LF	X	1	1	1	1	1	1	1	1	1	1
NF	–	X	3	4	7	5	1	1	1	1	1
FF	–	–	X	4	7	5	1	1	1	1	1
BF	–	–	3	X	8	5	1	1	1	1	1
PF	–	2	2	2	X	3	1	1	1	1	1
NFD	–	2	2	2	6	X	1	1	1	1	1
FFD	–	2	2	2		–	X	4		–	2
BFD	–	2	2	2		–	3	X		3	2
PFD	–	2	2	2	3	3	3	3	X	3	2
QFD	–	2	2	2		–	–	4		X	2
OPT	–	–	–	–	–	–	–	–	–	–	X

Figure 17.20 Results of the pairwise comparison of algorithms.

is no list for which the optimal algorithm would use more discs than any of the examined algorithms. Finally, 1's indicate that for list $\mathbf{t}_1$ the algorithm belonging to the row of the given cell in the figure needs more discs than the algorithm belonging to the column of the given cell.

If we keep analysing the numbers of discs in Figure 17.19, we can make up this figure to Figure 17.20.

Since the first row and the first column of the table is filled, we do not deal more with algorithm LF.

For list $\mathbf{t}_2$ NF, FF, BF and OPT use two discs, while the other 6 algorithms use three ones. Therefore we write 2's in the points of intersection of the columns of the 'winners' and the rows of the 'losers' (but we do not rewrite the 1's given in the points of intersection of PF and OPT, and NFD and OPT, so we write 2's in $4 \times 6 - 2 = 22$ cells. Since both the row and the column of OPT have been filled in, it is not dealt with any more in this section. The third list is disadvantageous for PF and PFD, therefore we write 3's in the empty cells in their rows. This list shows an example also for the fact that NF can be worse than FF, BF can be worse than

FF, and BFD than FFD and QFD.

The fourth list can be processed only by BF and BFD optimally, i.e., using two discs. Therefore we can write 4's in the empty cells in the columns of these two algorithms. For the fifth list NFD, FFD, BFD and QFD use only two, while NF, FF, BF, PF and PDF use three discs. So we can fill the suitable cells with 5's. The 'losers' of list $\mathbf{t}_6$ are NF and NFD—therefore, we write 6's in the empty cells in their rows. PF performs better when processing list $\mathbf{t}_7$ than FF. The following theorem helps us filling in the rest of the cells.

Theorem 17.8 *If* $\mathbf{t} \in D$, *then*

$$\mathrm{FF}(\mathbf{t}) \leq \mathrm{NF}(\mathbf{t}) \ .$$

Proof We perform an induction according to the length of the list. Let $\mathbf{t} = \langle t_1, t_2, \ldots, t_n \rangle$ and $\mathbf{t}_i = \langle t_1, t_2, \ldots, t_i \rangle$ $(i = 1, 2, \ldots, n)$. Let $\mathrm{NF}(\mathbf{t}_i) = \mathrm{N}_i$ and $\mathrm{FF}(\mathbf{t}_i) = \mathrm{F}_i$, and let n_i be the level of the last disc according to NF, which means the sum of the lengths of the files placed onto the non empty disc with the higher index, when NF has just processed $\mathbf{t}_i$. Similarly, let f_i be the level of the last disc according to FF. We are going to prove the following invariant property for each i: either $F_i < N_i$, or $F_i = N_i$ and $f_i \leq n_i$. If $i = 1$, then $F_1 = N_1$ and $f_1 = n_1 = t_1$, i.e., the second part of the invariant property holds. Suppose that the property holds for the value $1 \leq i < n$. If the first part of the invariant property holds before packing t_{i+1}, then either inequality $F_i < N_i$ stays true, or the numbers of discs are equal, and $f_i < n_i$ holds. If the numbers of discs were equal before packing of t_{i+1}, then after placing it either the number of discs of FF is smaller, or the numbers of discs are equal and the level of the last disc of FF is at most as big as that of NF. ■

A similar statement can be proved for the pairs of algorithms NF-BF, NFD-FFD and NFD-BFD. Using an induction we could prove that FFD and QFD need the same number of discs for every list. The previous statements are summarised in Figure 11.20.

17.4.6. The error of approximate algorithms

The relative efficiency of two algorithms (A and B) is often described by the ratio of the values of the chosen efficiency measures, this time the relative number of processors $\mathrm{A}(\mathbf{t})/\mathrm{B}(\mathbf{t})$. Several different characteristics can be defined using this ratio. These can be divided into two groups: in the first group there are the quantities describing the worst case, while in the other group there are those describing the usual case. Only the worst case is going to be discussed here (the discussion of the usual case is generally much more difficult). Let D_n denote the real list of n elements and D the set of all the real lists, i.e.,

$$D = \cup_{i=1}^{\infty} D_i \ .$$

Let $\mathcal{A}_{nd}$ be the set of algorithms, determining the number of discs, that is of algorithms, connecting a nonnegative real number to each list $\mathbf{t} \in D$, so implementing the mapping $D \to \mathbb{R}_0^+$).

Let $\mathcal{A}_{opt}$ be the set of the optimal algorithms, that is of algorithms ordering the optimal number of discs to each list, and OPT an element of this set (i.e., an algorithm that gives the number of discs sufficient and necessary to place the files belonging to the list for each list $\mathbf{t} \in D$).

Let $\mathcal{A}_{app}$ be the set of the approximation algorithms, that is of algorithms $\mathrm{A} \in \mathcal{A}_{nd}$ for which $\mathrm{A}(\mathbf{t}) \geq \mathrm{OPT}(\mathbf{t})$ for each list $\mathbf{t} \in D$, and there is a list $\mathbf{t} \in D$, for which $\mathrm{A}(\mathbf{t}) > \mathrm{OPT}(\mathbf{t})$..

Let A_{est} be the set of estimation algorithms, that is of algorithms $\mathrm{E} \in \mathcal{A}_{lsz}$ for which $\mathrm{E}(\mathbf{t}) \leq \mathrm{OPT}(\mathbf{t})$ for each list $\mathbf{t} \in D$, and there is a list $\mathbf{t} \in D$, for which $\mathrm{E}(\mathbf{t}) < \mathrm{OPT}(\mathbf{t})$.. Let F_n denote the set of real lists for which $\mathrm{OPT}(\mathbf{t}) = n$,, i.e., $F_n = \{\mathbf{t} | \mathbf{t} \in D \text{ and } \mathrm{OPT}(\mathbf{t}) = n\}$ $(n = 1, 2, \ldots)$.. In the following we discuss only algorithms contained in $\mathcal{A}_{nd}$. We define $(\mathrm{A},\ \mathrm{B} \in \mathcal{A})$ $R_{\mathrm{A},\mathrm{B},n}$ error function, $R_{\mathrm{A},\mathrm{B}}$ error (absolute error) and $R_{A,\infty}$ asymptotic error of algorithms A and B $(\mathrm{A},\ \mathrm{B} \in \mathcal{A})$ as follows:

$$R_{\mathrm{A},\mathrm{B},n} = \sup_{t \in F_n} \frac{\mathrm{A}(\mathbf{t})}{\mathrm{B}(\mathbf{t})} \ ,$$

$$R_{\mathrm{A},\mathrm{B}} = \sup_{t \in D} \frac{\mathrm{A}(\mathbf{t})}{\mathrm{B}(\mathbf{t})} \ ,$$

$$R_{\mathrm{A},\mathrm{B},\infty} = \limsup_{n \to \infty} R_{A,B,n} \ .$$

These quantities are interesting especially if $\mathrm{B} \in \mathcal{A}_{opt}$. In this case, to be as simple as possible, we omit B from the denotations, and speak about the error function, error and asymptotic error of algorithms $\mathrm{A} \in \mathcal{A}$, and $\mathrm{E} \in \mathcal{A}$. The characteristic values of NF file placing algorithm are known.

Theorem 17.9 *If* $\mathbf{t} \in F_n$, *then*

$$n = \mathrm{OPT}(\mathbf{t}) \leq \mathrm{NF}(\mathbf{t}) \leq 2\mathrm{OPT}(\mathbf{t}) - 1 = 2n - 1 \ . \tag{17.31}$$

Furthermore, if $k \in \mathbb{Z}$,, *then there are lists* $\boldsymbol{u}_k$ *and* $\boldsymbol{v}_k$ *for which*

$$k = \mathrm{OPT}(\mathbf{u}_k) = \mathrm{NF}(\mathbf{u}_k) \tag{17.32}$$

and

$$k = \mathrm{OPT}(\mathbf{v}_k) \text{ and } \mathrm{NF}(\mathbf{v}_k) = 2k - 1 \ . \tag{17.33}$$

From this statement follows the error function, absolute error and asymptotic error of NF placing algorithm.

Corollary 17.10 *If* $n \in \mathbb{Z}$, *then*

$$R_{\mathrm{NF},n} = 2 - \frac{1}{n} \ , \tag{17.34}$$

and

$$R_{\mathrm{NF}} = R_{\mathrm{NF},\infty} = 2 \ . \tag{17.35}$$

The following statement refers to the worst case of the FF and BF file packing algorithms.

Theorem 17.11 *If* $\mathbf{t} \in F_n$, *then*

$$\mathrm{OPT}(\mathbf{t}) \leq \mathrm{FF}(\mathbf{t}),\ \mathrm{BF}(\mathbf{t}) \leq 1.7\mathrm{OPT}(\mathbf{t}) + 2\ . \tag{17.36}$$

Furthermore, if $k \in \mathbb{Z}$,, *then there are lists* u_k *and* v_k *for which*

$$k = \mathrm{OPT}(\mathbf{u}_k) = \mathrm{FF}(\mathbf{u}_k) = \mathrm{BF}(\mathbf{u}_k) \tag{17.37}$$

and

$$k = \mathrm{OPT}(\mathbf{v}_k)\ \textit{and}\ \mathrm{FF}(\mathbf{v}_k) = \mathrm{BF}(\mathbf{v}_k) = \lfloor 1.7k \rfloor\ . \tag{17.38}$$

For the algorithm FF holds the following stronger upper bound too.

Theorem 17.12 *If* $\mathbf{t} \in F_n$, *then*

$$\mathrm{OPT}(\mathbf{t}) \leq \mathrm{FF}(\mathbf{t}) < 1.7\mathrm{OPT}(\mathbf{t}) + 1\ . \tag{17.39}$$

From this statement follows the asymptotic error of FF and BF, and the good estimation of their error function.

Corollary 17.13 *If* $n \in \mathbb{Z}$, *then*

$$\frac{\lfloor 1.7n \rfloor}{n} \leq R_{\mathrm{FF},n} \leq \frac{\lceil 1.7n \rceil}{n} \tag{17.40}$$

and

$$\frac{\lfloor 1.7n \rfloor}{n} \leq R_{\mathrm{BF},n} \leq \frac{\lfloor 1.7n + 2 \rfloor}{n} \tag{17.41}$$

further

$$R_{\mathrm{FF},\infty} = R_{\mathrm{BF},\infty} = 1.7\ . \tag{17.42}$$

If n is divisible by 10, then the upper and lower limits in inequality (17.40) are equal, thus in this case $1.7 = \mathrm{R}_{\mathrm{FF},n} = R_{\mathrm{BF},n}$.

Exercises

17.4-1 Prove that the absolute error of the FF and BF algorithms is at least 1.7 by an example.

17.4-2 Implement the basic idea of the FF and BF algorithms so that the running time would be $O(n \lg n)$.

17.4-3 Complete Figure 11.20.

Problems

17-1 Smooth process selection for an empty partition

Modify the Long-Waiting-or-Not-Fit-Smaller algorithm in a way that instead of giving priority to processes with *points* above the *threshold*, selects the process with the highest *rank* + *points* among the processes fitting into the partition. Prove the correctness of the algorithm and give an upper bound for the waiting time of a process.

17-2 Partition search algorithms with restricted scope
Modify the Best-Fit, Limited-Best-Fit, Worst-Fit, Limited-Worst-Fit algorithms to only search for their optimal partitions among the next m suitable one following the last split partition, where m is a fixed positive number. Which algorithms do we get in the $m = 1$ and $m = \infty$ cases. Simulate both the original and the new algorithms, and compare their performance regarding execution time, average number of waiting processes and memory fragmentation.

17-3 Avoiding page replacement anomaly
Class the discussed page replacement algorithms based on whether they ensure to avoid the anomaly or not.

17-4 Optimal page replacement algorithm
Prove that for each demanding page replacement algorithm A, memory size m and reference string R holds

$$f_A(m, R) \leq f_{\text{OPT}}(m, R) \ .$$

17-5 Anomaly
Plan (and implement) an algorithm with which it can occur that a given problem takes longer to solve on $q > p$ processors than on $p > 1$ ones.

17-6 Error of file placing algorithms
Give upper and lower limits for the error of the BF, BFD, FF and FFD algorithms.

Chapter Notes

The basic algorithms for dynamic and fixed partitioning and page replacement are discussed according to textbooks by Silberschatz, Galvin and Gagne [15], and Tanenbaum and Woodhull [16].

Defining page replacement algorithms by a Mealy-automat is based on the summarising article by Denning [5], and textbooks by Ferenc Gécseg and István Peák [6], Hopcroft, Motwani and Ullman [7].

Optimizing the MIN algorithm was proved by Mihnovskiy and Shor in 1965 [13], after that by Mattson, Gecsei, Slutz and Traiger in 1970 [12].

The anomaly experienced in practice when using FIFO page replacement algorithm was first described by László Bélády [2] in 1966, after that he proved in a constructive way that the degree of the anomaly can approach two arbitrarily closely in his study he wrote together with Shedler. The conjecture that it cannot actually reach two can be found in the same article (written in 1969).

Péter Formai and Antal Iványi [?] showed that the ratio of the numbers of page replacements needed on a big and on a smaller computer can be arbitrarily large in 2002.

Examples for scheduling anomalies can be found in the books by Coffman [3], Iványi and Smelyanskiy [9] and Roosta [14], and in the article by Lai and Sahni [11].

Analysis of the interleaved memory derives from the article [**?**].

The bound $\mathrm{NF}(\mathbf{t}) \le 2\mathrm{OPT}(\mathbf{t}) + 2$ can be found in D. S. Johnson's PhD dissertation [**?**], the precise Theorem 17.9. comes from [8]. The upper limit for FF and BF is a result by Johnson, Demers, Ullman, Garey and Graham [10], while the proof of the accuracy of the limit is that by [8, **?**]. The source of the upper limit for FFD and BFD is [10], and that of the limit for NFD is [1]. The proof of the NP-completeness of the file packing problem—leading it back to the problem of partial sum—can be found in the chapter on approximation algorithms in *Introduction to Algorithms* [4].

Bibliography

[1] B. Baker. A tight asymptotic bound for next-fit decreasing bin-packing. *SIAM Journal on Algebraic and Discrete Methods*, 2(2):147–152, 1981. 846

[2] L. A. Bélády, R. Nelson, G. S. Shedler. An anomaly in space-time characteristics of certain programs running in paging machine. *Communications of the ACM*, 12(1):349–353, 1969. 845

[3] E. Coffman. *Computer and Job Shop Scheduling*. John Wiley & Sons, 1976. 845

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms* (3rd edition, second corrected printing). The MIT Press/McGraw-Hill, 2010. 846

[5] P. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, 1970. 845

[6] F. Gécseg, I. Peák. *Algebraic Theory of Automata*. Akadémiai Kiadó, 1972. 845

[7] J. E. Hopcroft, R. Motwani, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001 (in German: *Einführung in Automatentheorie, Formale Sprachen und Komplexitätstheorie*, Pearson Studium, 2002). 2nd edition. 845

[8] A. Iványi. Performance bounds for simple bin packing algorithms. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computarorica*, 5:77–82, 1984. 846

[9] A. Iványi, R. Szmeljánszkij. *Elements of Theoretical Programming* (in Russian). Moscow State University, 1985. 846

[10] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, R. L. Graham. Worst-case performance-bounds for simple one-dimensional bin packing algorithms. *SIAM Journal on Computing*, 3:299–325, 1974. 846

[11] T. Lai, S. Sahni. Anomalies in parallel branch and bound algorithms. *Communications of ACM*, 27(6):594–602, 1984. 846

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970. 845

[13] Sz. Mihnovskiy, N. Shor. Estimation of the page fault number in paged memory (in Russian). *Kibernetika (Kiev)*, 1(5):18–20, 1965. 845

[14] S. H. Roosta. *Parallel Processing and Parallel Algorithms*. Springer-Verlag, 1999. 846

[15] A. Silberschatz, P. Galvin, G. Gagne. *Applied Operating System Concepts*. John Wiley & Sons, 2000. 845

[16] A. S. Tanenbaum, A. Woodhull. *Operating Systems. Design and Implementation*. Prentice Hall, 1997. 845

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, "2-3-4-tree" is indexed as if were "two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: *exe* for exercise, *exa* for example, *fig* for figure, *pr* for problem and *fn* for footnote.

The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

A
anomaly, *822*

B
backing memory, *811*
base addressing, *797*
Best Fit, 836
BEST-FIT, *807*, 810*exe*
Best Fit Decreasing, 837
BF, *836*, *see* Best Fit
BFD, *see* Best Fit Decreasing
busy page frame, 812

C
CALCULATE-RANK, *802*
CLOCK, 819, 821*exe*

D
dynamic page replacement algorithms, *812*
dynamic partitions, *803*

E
Economic Power, 839
EP, *see* Economic Power
execution time, *823*

F
FACT, *see* Factorial
Factorial, 838
FF, *836*, *see* First Fit
FFD, 837, *see* First Fit decreasing
FIFO, *813*, *see* First In First Out
FIFO-EXECUTES, *814*
First Fit, 835
FIRST-FIT, *805*, 810*exe*
First In First Out, 813
fixed partitions, *797*
fragmentation the memory, 806
frame, *811*
    busy, 812

I
index of the virtual page frame, *812*

L
LARGEST-FIT, *798*, 810*exe*
LARGEST-OR-LONG-WAITING-FIT, *800*
LARGEST-OR-LONG-WAITING-FIT, 810*exe*
LF, *835*, *see* Linear Fit
LFU-EXECUTES, *817*
LIFO, 821*exe*
LIMITED-BEST-FIT, *808*, 810*exe*
LIMITED-WORST-FIT, *809*, 810*exe*
Linear Fit, 835
LIST-SCHEDULING, 821
LOAD-LARGEST, *798*
LOAD-LARGEST-OR-LONG-WAITING, *801*
LOAD-LONG-WAITING-OR-NOT-SMALLER, *803*
LONG-WAITING-OR-NOT-FIT-SMALLER, *803*,
    810*exe*
LRU, *814*, *see* Least Recently Used
LRU-EXECUTES, *815*

M
main memory, *811*
memory, *811*
memory management, 796–846

N
Next Fit, 835
NEXT-FIT, *806*, 810*exe*
NF, *835*, *see* Next Fit
NFD, 837, *see* Next Fit Decreasing
NFU, 821*exe*
NRU, *817*, 821*exe*

NRU-Executes, *818*
NRU-Prepares, *818*
NRU-Swap-Out, *818*
number of page faults, *812*

O
OPT-Executes, *815*
OPT-Swap-Out, *816*

P
page, *811*
Page Frequency Fault, 820
page replacement algorithm
  dynamic, 812
  static, 812
page replacement algorithms,
page table, *812*
Pairwise Fit, 836
Pairwise Fit Decreasing, 837
partition, *796*
  dynamic, 803
  fixed, 797
PF, *837*, *see* Pairwise Fit
PFD, *see* pairwise Fit Decreasing
PFF, *820*, *see* Page Frequency Fault
Place, *805*, 810*exe*
present/absent bit, *812*

Q
QFD, *838*, *see* Quick Fit Decreasing
QP, *see* Quick Power
Quick Fit Decreasing, 837
Quick Power, 838

R
reference string, *812*
relocatable programs, *796*

S
Second-Chance, 821*exe*
segment, *811*
Shortening of lists, 839
Simple Power, 838
SL, *see* Shortening of lists
SP, *see* Simple Power
Split-Partition, *804*
*-Run, *813*
static page replacement algorithms, *812*
swap-in, 813
swap-out, 813

T
task, *823*
beg-ind, 837
counter, 820
end-ind, 837
free, 836
guarded, 816
ind, 836

U
ULE, *see* upper and lower estimations
upper and lower estimations, 839

V
vector of the execution times, *823*
virtual memory, 811

W
Working-Set, 819
Worst-Fit, *809*, 810*exe*
writing array, *812*
WS, *820*, *see* Working Set

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of her/his birth and death.

B
Baker, B., 847
Balogh, Ádám, 796
Bélády, László A., 822, 845, 847

C
Coffman, Ed G., Jr., 845–847
Cormen, Thomas H., 846, 847

D
Demers, A., 847
Denning, Peter J., 845, 847

F
Fornai, Péter, 823, 845, 846

G
Gagne, Greg, 845, 847
Galvin, Peter Baer, 845, 847
Garey, Michael R., 847
Gécseg, Ferenc, 845, 847
Gecsei, Jan, 845, 847
Graham, Ronald Lewis, 847

H
Hopcroft, John E., 845, 847

I
Iványi, Antal Miklós, 796, 823, 845–847

J
Johnson, David S., 847

L
Lai, Ten Hwang, 846, 847
Leiserson, Charles E., 846, 847

M
Mattson, R. L., 845, 847
Mihnovskiy, S. D., 845, 847
Motwani, Rajeev, 845, 847

N
Nelson, R. A., 822, 847

P
Peák, István (1938–1989), 845, 847

R
Rivest, Ronald Lewis, 846, 847
Roosta, Sayed H., 846, 847

S
Sahni, Sartaj, 846, 847
Shedler, G. S., 822, 845, 847
Shor, N. Z., 845, 847
Silberschatz, Avi, 845, 847
Slutz, D. R., 845, 847
Stein, Clifford, 847

SZ
Szmeljánszkij, Ruszlán, 846, 847

T
Tanenbaum, Andrew S., 845, 847
Traiger, I. L., 845, 847

U
Ullman, Jeffrey David, 845, 847

W
Woodhull, Albert S., 845, 847