GEORGE MASON UNIVERSITY
Computer Science Department

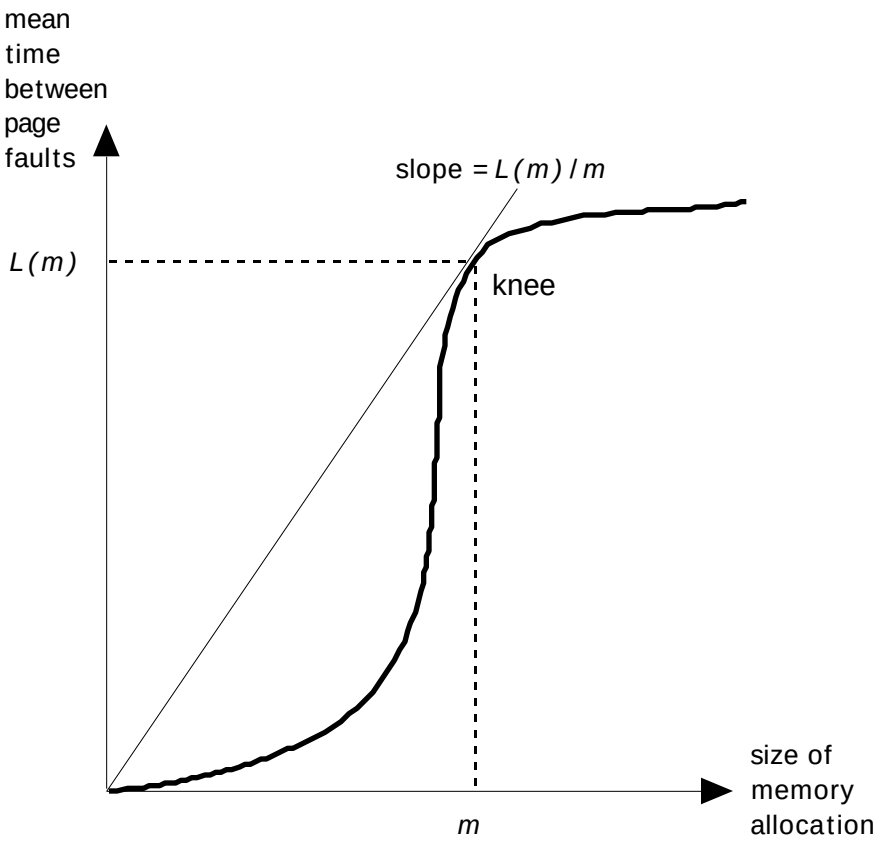
CS 471 -- Operating Systems
Spring 2000

Peter J. Denning

NOTES ON MEMORY MANAGEMENT ANALYSIS ALGORITHMS

Virtual memory is now a central feature of operating systems and microcomputer chip architectures. It is so ordinary that few people remember when it was the center of enormous controversies. It was introduced on the Atlas computer at University of Manchester in 1958 under the name "one-level storage system". The architectural feature, paging, on which it was based had been introduced a decade earlier on a predecessor version of Atlas. In the middle 1960s a team lead by David Sayre and Les Belady at the IBM Watson Research Center conducted an extensive study of paging system performance on an IBM 7044 machine; their principal conclusion was that virtual memory outperformed all manual forms of page management when programs have "good locality". In 1970, Mattson, Gecsei, Slutz, and Traiger of IBM introduced an efficient analysis method called "stack algorithms" that would calculate the fault-count function $F(m)$ in one pass over the program's virtual address. In modeling paging system performance, it is often more convenient to use the lifetime function $L(m)$. The lifetime function gives the mean virtual time between page faults; it is $L(m)=T/F(m)$, where T is the total execution time of the program without interrupts. The basic time unit is the main memory reference time (today T would be on the order of 0.1-0.01 microsecond). A typical lifetime function graph has a rough "S" shape; see the sketch on the next page.

Space Time: The space-time accumulated by a process in a time interval is the integral of its resident set size over that interval. Thus if a process uses 3 pages for 10 seconds, it generates 30 page-seconds of space-time. It has always been a rule of thumb that memory managers should seek to minimize space-time. Suppose that $w_i(t)$ is the resident set size of process i at time t. The mean space-time per process, Y, is the total number of page-seconds of memory use by a job -- the area under the memory-use versus time curve. It is not hard to show that $M=XY$, where M is the total number of pages of memory and X is the throughput. For a given total memory size, minimizing space-time maximizes throughput. We know experimentally that the minimum space-time of a job occurs when its memory allocation is close to the knee of its lifetime curve.



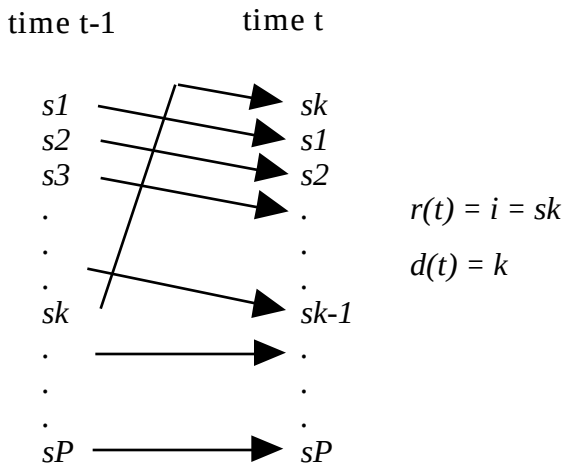
LRU Analysis: The successive memory contents when a program's fixed space is managed by least-recently-used replacement can be displayed in a chart as below. The LRU stacks and distances are defined below. A page fault occurs in a given memory size if and only if the stack distance is larger than that memory size; the distance string thus contains all the information needed to construct the fault function of a given program operating under LRU replacement.

refs:		**1**	**2**	**1**	**...**
memory	4	1	2	1	...
contents	2	4	1	2	...
(LRU order)	3	2	4	4	...
distances:		i	3	2	...

Computing Lifetime Functions for LRU and MIN Page Replacement

Function *F(m)* is called the fault function because it number of page faults inside a process that has been allocated *m* pages of memory (for a given paging strategy). One way to compute *F(m)* for LRU is to run a series of simulations for each value *m=1,2,...,P*. On a typical run, the simulator will calculate the contents of an *m*-page memory after each new reference occurs and count the faults. This appears to require *P* separate simulations, one for each possible allocation up to the size *P* of the program. However, using an approach called "stack algorithms", the entire simulation, for all values of m from 1 to *P*, can be done at once.

Consider the LRU algorithm. Let *r(1),r(2),...,r(t),...,r(T)* denote the reference string, i.e., series of page numbers referenced by the program. Let *S(t)=(s1(t),...sP(t))* denote the "stack", a vector in which each page appears just once and which orders the pages by decreasing "backward distance": *s1(t)* is the most recently referenced page (i.e., *s1(t)=r(t)*), *s2(t)* is the next most recent page, and so on. Whenever there is a page fault because *r(t+1)* is not in memory, the least recently used page is removed from memory; this is none other than *sm(t)*. In fact, the contents of memory at time *t* is none other than the first *m* elements of the stack *S(t)*. It is also easy to see that the elements of *S(t+1)* are related to *S(t)* in a simple fashion: the referenced page moves from position m to the top, and the intervening pages are pushed down as a group one position. None of the pages below position *m* moves at all. The name "stack" refers to the way in which all the memory contents are "stacked up" in one vector of successive differences.

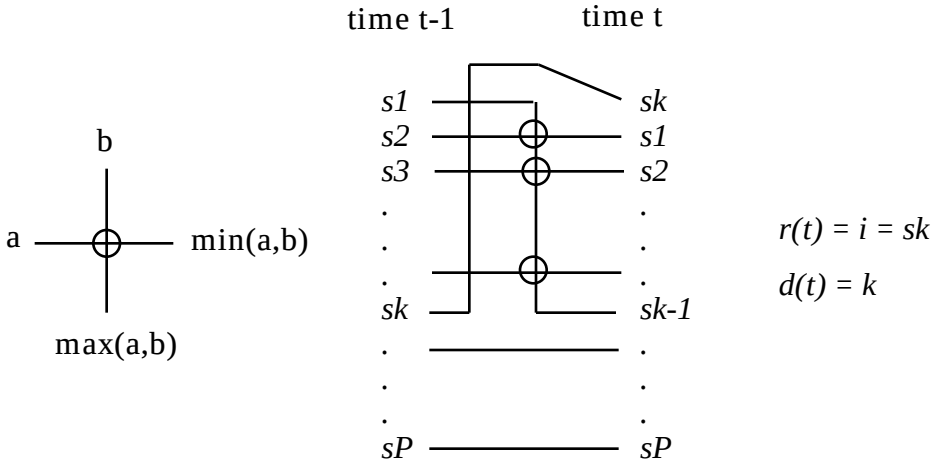


Notice that the contents of an *m*-page memory are the *m*-page prefix of the stack *S(t)*. This guarantees that LRU memory contents satisfy an "inclusion property", so that adding more memory is guaranteed never to increase page faults. It is easy to find examples with paging algorithms that violate this property, notably FIFO, in which adding memory may *increase* faults.

The position of a page reference in the stack is called its "stack distance" or distance for short, and is denoted *d(t)*. For the first reference to a page, we assume *d(t)*=inf (infinity or ∞). The new page is added to the stack, and the other pages are pushed down one position. The position "infinity" is shorthand for "lower than anything else" in the stack; after its reference, the stack contains one more page than before.

Notice that *d(t)* is the position of *r(t)* in the stack*S(t-1)*. Notice further that a fault occurs if and only if *d(t)>m*. We can thus create a histogram *h*(1...*P*,inf), in which *h(i)* denotes the number of times distance=*i* occurred. The total number of faults, *F(m)*, is the sum of *h(i)* for *i>m* (including *h*(inf)). The lifetime function is *L(m)=T/F(m)*, where *T* is the program's execution time.

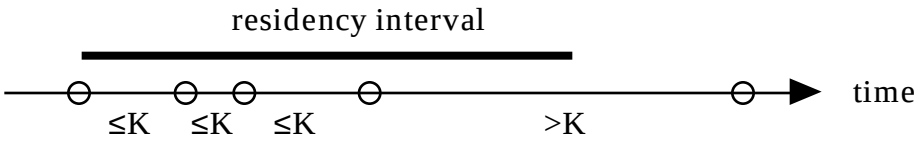
The procedure above can be modified to calculate stack distances for the MIN algorithm (known also as OPT), as suggested in the sketch below. Instead of simply pushing down the stack elements above the one referenced, we make pairwise comparisons. The winner (one with shorter forward distance) goes into the stack slot and the loser (one with the longer forward distance) moves down for the next comparison. On the first reference to a page, the update procedure is performed on the while stack since the referenced page is coming from distance "inf". The fault function is calculated from the distance histogram as it was for LRU. The contents of an *m*-page memory are the *m* elements at the top of the stack.



Computing Fault Functions for Working-Set Memory Management

The working set of a program at time *t* for window size *K*, *w(t,K)*, is the set of pages referenced in the interval *t-K+1,...,t*. Note that page *i=r(t)* causes a page fault at time *t* if and only if the backward interval to the previous reference to page *i* exceeds *K*. This means we can determine the residency of a page in the working set simply by examining the sequence of inter-reference intervals. As shown by the sketch, the page will be resident continuously through a sequence

of inter-reference intervals all ≤*K* and then for *K* time units into a following inter-reference interval >*K*.



Suppose that we collect a histogram of the inter-reference interval lengths: *h(a)* denotes the number of inter-reference intervals of length =*a*, and *h(inf)* is the number of first references. We can see from the diagram that the fault-count for given window size *K* is $F(K)=h(K+1)+h(K+2)+...+F(T)+P$, where $P=h(inf)$ is the number of pages in the reference string.

We can collect this histogram by maintaining an array TIME[*i*] telling the last time at which page *i* was referenced. Initially TIME[*i*]=0 indicates there is no prior reference. On scanning reference $r(t)=i$, we calculate the inter-reference interval length as $a=t$-TIME[*i*], set TIME[*i*]=*t*, and add 1 to *h(a)*.

To calculate the mean working set size for a given window size, we first calculate the space-time consumed by working sets and then divide by the reference string length *T*. This diagram illustrates for window *K*=4 (it's the reference string on which FIFO exhibits the Belady Anomaly):

r:	1	2	3	4	1	2	5	1	2	3	4	5
1:	1	1	1	1	1	1	1	1	1	1	1	
2:		2	2	2	2	2	2	2	2	2	2	2
3:			3	3	3	3				3	3	3
4:				4	4	4	4				4	4
5:							5	5	5	5		5

This reference string is of length 12. The potential space time is 60 (=12x5) page-ticks (where 1 tick is the unit of time). The actual space-time is the number of boxes containing a value, or 40 page-ticks. Dividing 40 by 12 gives the mean working set size $m(3)=3.33$ pages.

To calculate the mean working set size for any *K*, we will use the same principle: determine the total working-set space-time and divide by *T*, the reference string length. From the diagram, we see that all inter-reference intervals ≤*K* contribute their own lengths to the space-time, and all inter-reference intervals >*K* contribute *K* to the space-time. This counting method does not tally up the contribution *after* the last reference. To count these "end corrections", we define as the distance of the last reference to page *i* from the end:

$E(K,i)$ = if ($d(i)>K$) then {*K*} else {$d(i)$}

The total end-correction is. $E(K) = \sum_{i=1}^{K} E(K,i)$ The total space-time is:

$$ST(K) = \sum_{1 \le a \le K} ah(a) + K \sum_{K < a \le T} h(a) + E(K)$$

This total divided by T is $m(K)$, the mean space occupied by the working set. If you are good at manipulating summations, you can show that

$$m(K) = 1 + \frac{F(1) + \ldots + F(K-1) + E(K) - KP}{T}$$

The set of points $(m(K),F(K))$ for $K=1,\ldots,T$ traces a curve of faults versus space that can be compared to the curves of other paging algorithms. The points on the curve are at fractional values of memory size. For fixed-space algorithms, the points on the curve are at integer values of memory size.