

In view of the previous slide, it makes sense to design programs so that data is read from and written to disk in relatively large chunks... but there is more.

## Spatial Locality of Reference

In many cases, if a program accesses one part of a file, there is a high probability that the program will access nearby parts of the file in the near future.

**Moral: grab a larger chunk than you immediately need.**

## Temporal Locality of Reference

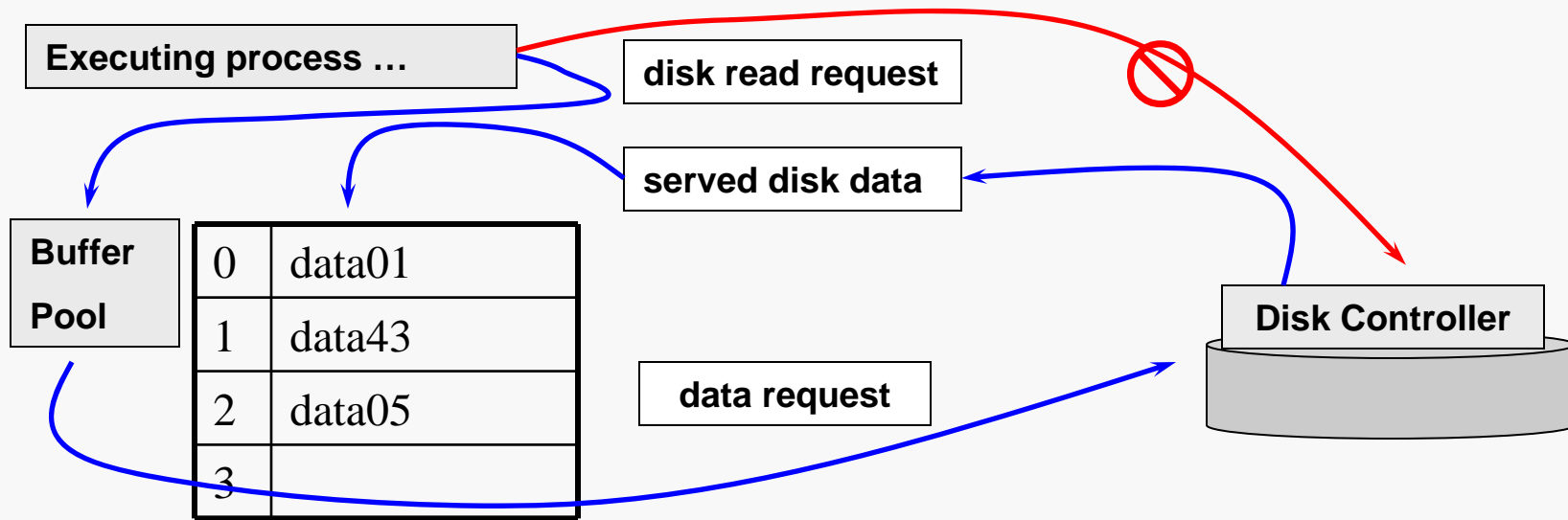
In many cases, if a program accesses one part of a file, there is a high probability that the program will access the same part of the file again in the near future.

**Moral: once you've grabbed a chunk, keep it around.**

buffer pool a series of buffers (memory locations) used by a program to cache disk data

A program that does much disk I/O can often improve its performance by employing a buffer pool to take advantage of locality of reference.

Basically, the buffer pool is just a collection of data chunks. The program reads and writes data in buffer-sized chunks, storing newly-read data chunks into the pool, replacing currently stored chunks as necessary.



The buffer pool must be organized physically and logically.

The physical organization is generally an ordered list of some sort.

The logical organization depends upon how the buffer pool deals with the issue of replacement — if a new data chunk must be added to the pool and all the buffers are currently full, one of the current elements must be replaced.

If the replaced element has been modified, it (usually) must be written back to disk or the changes will be lost. Thus, some replacement strategies may include a consideration of which buffer elements have been modified in choosing one to replace.

Some common buffer replacement strategies:

FIFO (first-in is first-out) organize buffers as a queue

LFU (least frequently used) replace the least-accessed buffer

LRU (least recently used) replace the longest-idle buffer

# FIFO Replacement

Logically the buffer pool is treated as a queue:

655:	655	miss						
289:	655	289	miss					
586:	655	289	586	miss				
289:	655	289	586	hit				
694:	655	289	586	694	miss			
586:	655	289	586	694	hit			
655:	655	289	586	694	hit			
138:	655	289	586	694	138	miss		
289:	655	289	586	694	138	hit		
694:	655	289	586	694	138	hit		
289:	655	289	586	694	138	hit		
694:	655	289	586	694	138	hit		
851:	289	586	694	138	851	miss		
586:	289	586	694	138	851	hit		
330:	586	694	138	851	330	miss		
289:	694	138	851	330	289	miss		
694:	694	138	851	330	289	hit		
331:	138	851	330	289	331	miss		
289:	138	851	330	289	331	hit		
694:	851	330	289	331	694	miss		
Number of accesses:		20						
Number of hits:		10						
Number of misses:		10						
Hit rate:		50.00						

Takes no notice of the access pattern exhibited by the program. Consider what would happen with the sequence:

655  
289  
655  
393  
655  
127  
655  
781  
...

# LFU Replacement

For LFU we must maintain an access count for each element of the buffer pool. It is also useful to keep the elements sorted by that count.

655:	(655, 1)	miss						
289:	(655, 1)	(289, 1)	miss					
586:	(655, 1)	(289, 1)	(586, 1)	miss				
289:	(289, 2)	(655, 1)	(586, 1)	hit				
694:	(289, 2)	(655, 1)	(586, 1)	(694, 1)	miss			
586:	(289, 2)	(586, 2)	(655, 1)	(694, 1)	hit			
655:	(289, 2)	(586, 2)	(655, 2)	(694, 1)	hit			
138:	(289, 2)	(586, 2)	(655, 2)	(694, 1)	(138, 1)			
289:	(289, 3)	(586, 2)	(655, 2)	(694, 1)	(138, 1)			
694:	(289, 3)	(586, 2)	(655, 2)	(694, 2)	(138, 1)			
289:	(289, 4)	(586, 2)	(655, 2)	(694, 2)	(138, 1)			
694:	(289, 4)	(694, 3)	(586, 2)	(655, 2)	(138, 1)			
851:	(289, 4)	(694, 3)	(586, 2)	(655, 2)	(851, 1)			
586:	(289, 4)	(694, 3)	(586, 3)	(655, 2)	(851, 1)			
330:	(289, 4)	(694, 3)	(586, 3)	(655, 2)	(330, 1)			
289:	(289, 5)	(694, 3)	(586, 3)	(655, 2)	(330, 1)			
694:	(289, 5)	(694, 4)	(586, 3)	(655, 2)	(330, 1)			
331:	(289, 5)	(694, 4)	(586, 3)	(655, 2)	(331, 1)			
289:	(289, 6)	(694, 4)	(586, 3)	(655, 2)	(331, 1)			
694:	(289, 6)	(694, 5)	(586, 3)	(655, 2)	(331, 1)			
Number of accesses:	20							
Number of hits:	12							
Number of misses:	8							
Hit rate:	60.00							

Aside from cost of storing and maintaining counter values, and searching for least value, consider the sequence:  
655 (500 times)  
289 (500 times)  
100  
101  
102  
103  
...

# LRU Replacement

With LRU, we may use a simple list structure. On an access, we move the targeted element to the front of the list. That puts the least recently used element at the tail of the list.

655:	655	miss					
289:	289	655	miss				
586:	586	289	655	miss			
289:	289	586	655	hit			
694:	694	289	586	655	miss		
586:	586	694	289	655	hit		
655:	655	586	694	289	hit		
138:	138	655	586	694	289	miss	
289:	289	138	655	586	694	hit	
694:	694	289	138	655	586	hit	
289:	289	694	138	655	586	hit	
694:	694	289	138	655	586	hit	
851:	851	694	289	138	655	miss	
586:	586	851	694	289	138	miss	
330:	330	586	851	694	289	miss	
289:	289	330	586	851	694	hit	
694:	694	289	330	586	851	hit	
331:	331	694	289	330	586	miss	
289:	289	331	694	330	586	hit	
694:	694	289	331	330	586	hit	
Number of accesses:		20					
Number of hits:		11					
Number of misses:		9					
Hit rate:		55.00					

Consider what would happen with the sequence:

- 655
- 289
- 655
- 301
- 302
- 303
- 304
- 289
- ...

The performance of a replacement strategy is commonly measured by its *fault rate*, i.e., the percentage of requests that require a new element to be loaded into the pool.

Some observations:

- faults will occur unless the pool contains the entire collection of data objects that are needed (the *working set*)
- which data objects are needed tends to change over time as the program runs, so the working set varies over time
- if the buffer pool is too small, it may be impossible to keep the current working set resident (in the buffer pool)
- if the buffer pool is too large, the program will waste memory

None of these replacement strategies, or any other feasible one, is best in all cases.

All are used with some frequency.

Intuitively, LRU and LFU make more sense than FIFO.

The performance you get is determined by the access pattern exhibited by the running program, and that is often impossible to predict.

Belady's optimal replacement strategy:

replace the element whose next access lies furthest in the future

Sometimes stated as “replace the element with the maximal forward distance”.

Requires knowing the future, and so is impossible to implement.

Does suggest considering predictive strategies.



There are some general properties a good buffer pool will have:

- the buffer size and number of buffers should be client-configurable
- the buffer pool may deal only in "raw bytes"; i.e., not know anything at all about the internals of the data record format used by the client code

OR

the buffer pool may deal in interpreted data records, parsed from the file and transformed into an object

- if records are fixed-length then each buffer should hold an integer number of records; for variable-length records, things are more complex and it is often necessary for buffers to allow some internal fragmentation
- empirically, a program using a buffer pool is considered to be achieving good performance if less than 10% of the record references require loading a new record into the buffer pool