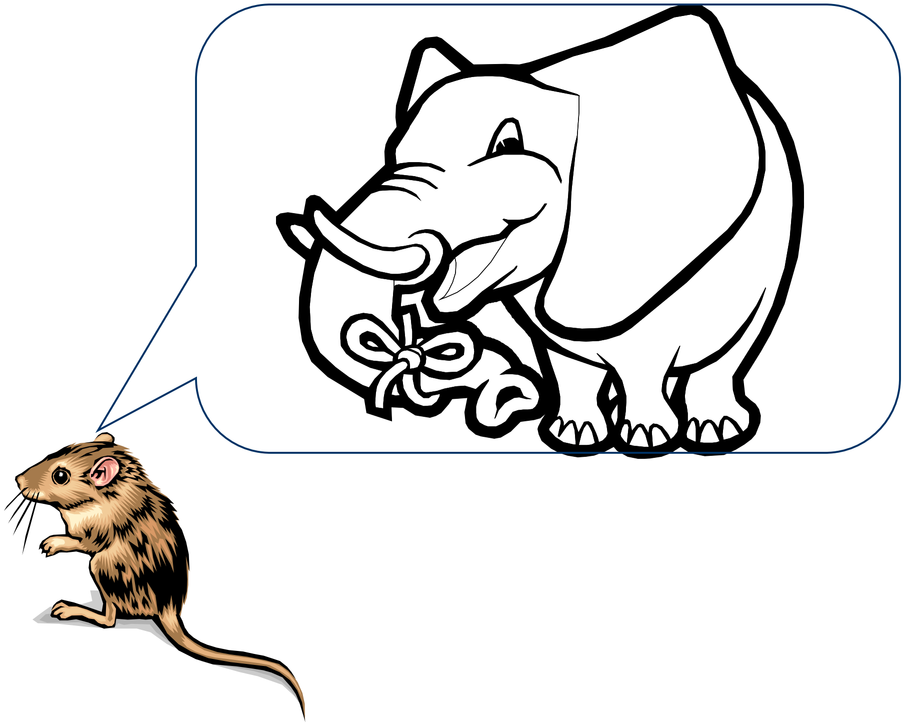
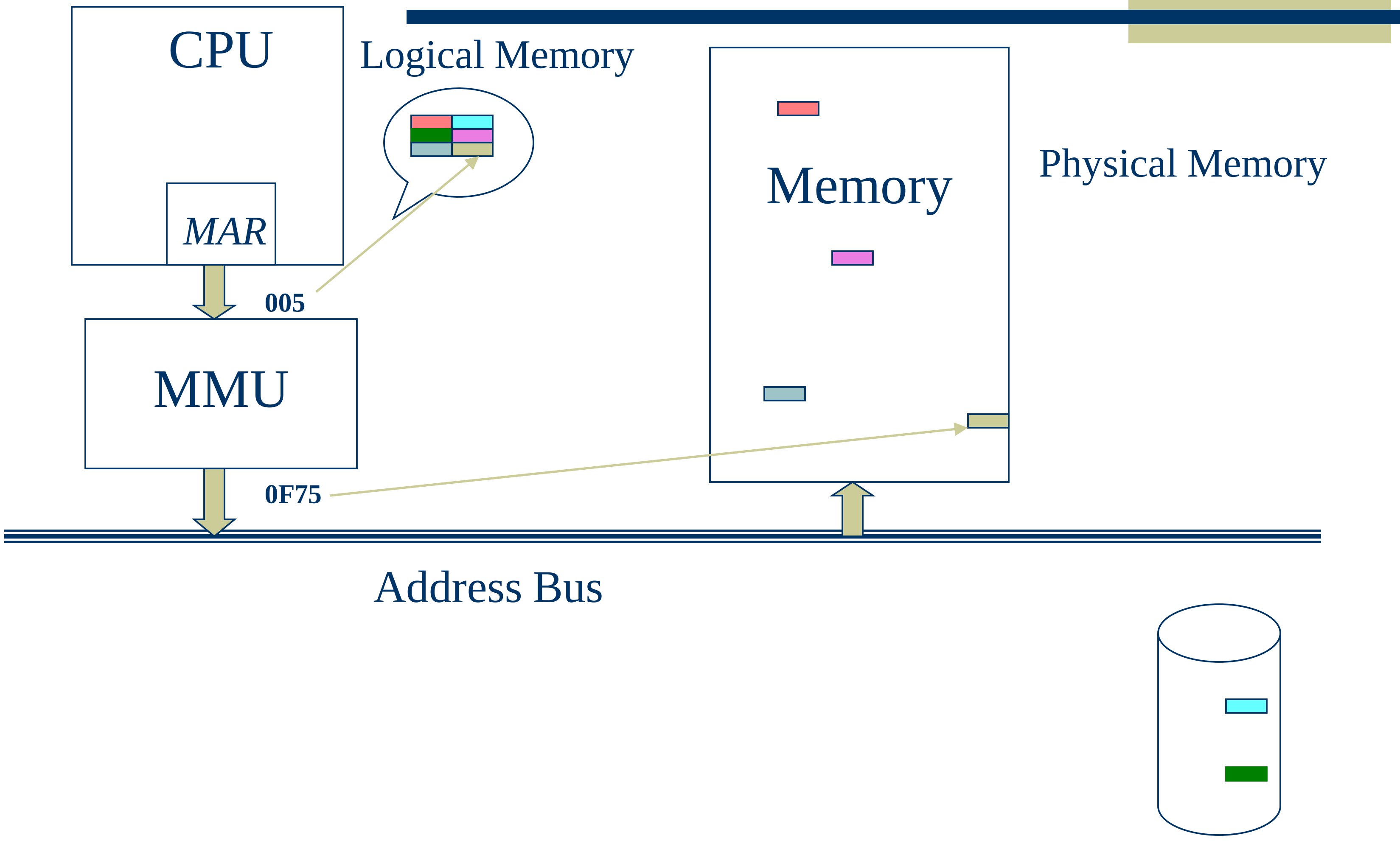
CIS 520, *Operating Systems Concepts*

Lecture 7

Virtual Memory

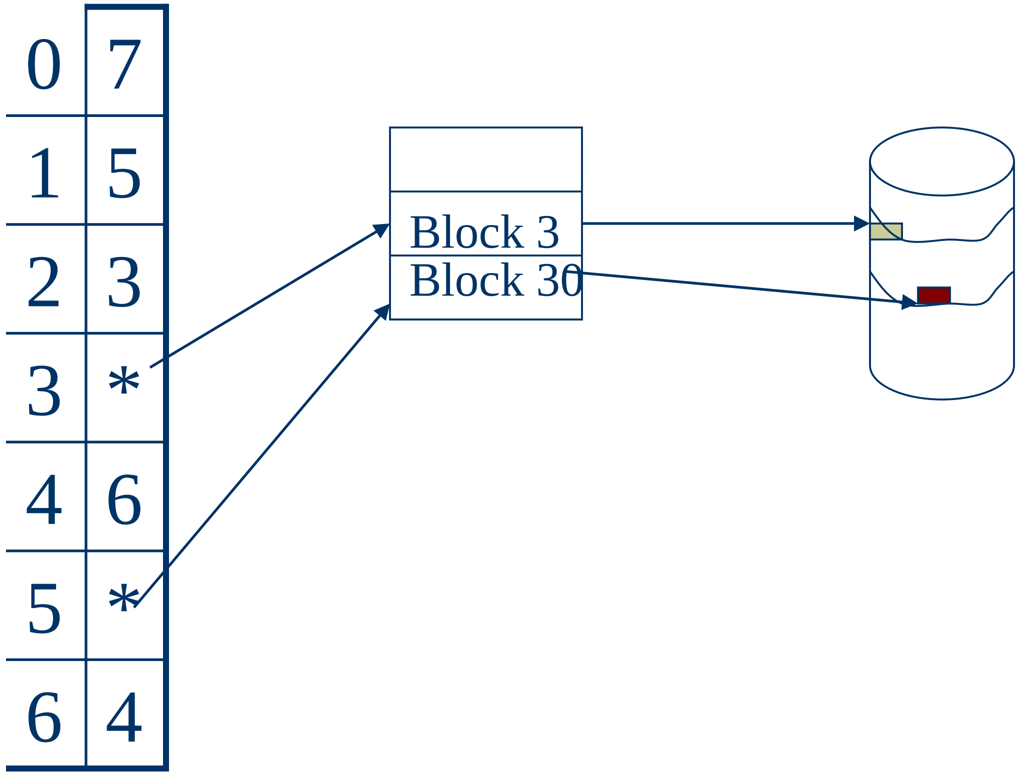


Virtual Memory



The Full Memory Map

From the *Process Table* entry:



A Page Table Entry Revisited

- A *Valid/Invalid* bit is used to indicate whether the page is in the physical memory (or in secondary storage)
- A *Clean/Dirty* bit is used to indicate whether the frame had been written to (and thus needs to be saved to the disk)
- A *Locked/Unlocked* bit is used to indicate whether the frame may not be touched

0	7	r	v	c	u
1	5	rw	v	d	u
2	3	xr	v	c	u
3	9	xr	v	c	u
4	x		i		u
5	1		v	c	l
6	4	x	v	c	u

Memory Reference String

The memory references come from two sources:

1. Code references (by PC)
2. Data references (by the MAR register)

A sequence of page references over the life of a program constitutes a *memory reference string*.

A Memory Reference String *Example:*

Memory measurement unit: 1 byte

Page size: 250 *Instruction size*: 4 bytes (one word)

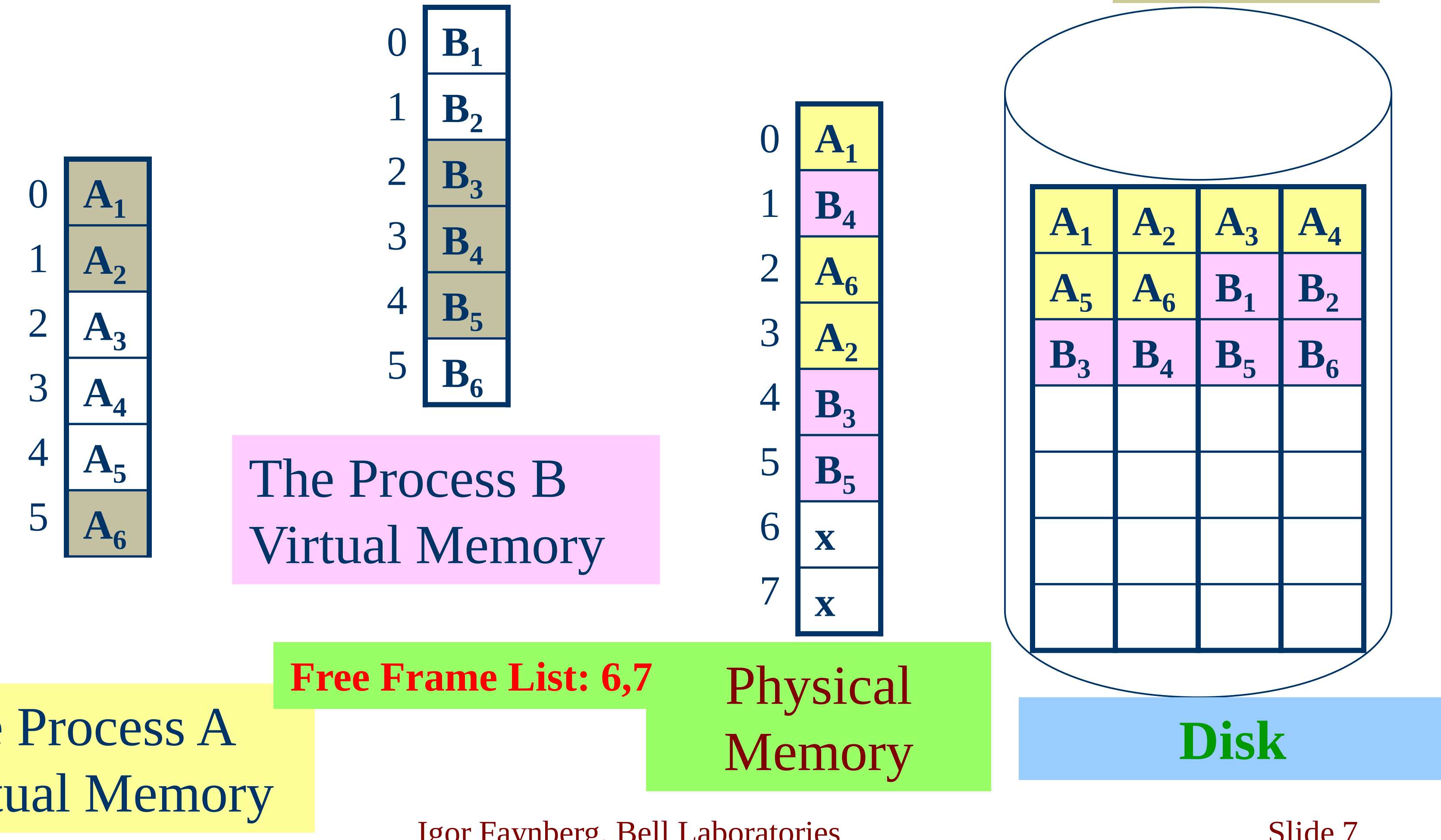
PC at: 480 *SP* at: 25004

$$page = \left\lceil \frac{address}{page\ size} \right\rceil$$

480: *MOVEW @2000, R1*	*1, 8,*
484: *MOVEW R1, @SP*	*1, 100,*
488: *SUBI SP, #4*	*1,*
492: *JSR @5120*	*1, 20, 100, [subroutine string], …*
496: *ADDI SP, #4*	*1,*
500: *CMP R0, #0*	*2,*
504: *JNZ @5152*	*2 or [2, 20]*

1, 8, 1, 100, 1, 1, 20, 100, ... 1, 2, 2

Pages: Virtual Memory, Physical Memory, Disc



What Happens on a *Page Fault*

1. The MMU causes a bus error
2. The CPU saves the PC and SP (and possibly some other useful information that would help determine the cause) on the stack, disables interrupts, and jumps to the appropriate interrupt service routine
3. The interrupt service routine saves all registers, determines that page fault occurred, adjusts the *interrupt mask*, and calls the appropriate memory management routine
4. The memory management routine checks if the address is within the process space; if not, the process is killed, and the *scheduler* is called
5. If the address is within the space, the memory management routine tries to find a free frame; if it cannot, it selects the victim for *paging out*
6. If the selected frame is *dirty*, the frame is marked as *locked*, and the I/O is scheduled to write it to the disk
7. While the I/O is taking place, the scheduler is called, and another process ends up running; when the page is written, it is marked *clean*

This is why!

What Happens on a *Page Fault (cont.)*

8. As soon as the frame is *clean*, the memory manager computes the disc address of the page to bring in, locks the frame, and schedules an I/O, again releasing the CPU
9. When the I/O is completed; the page table is updated appropriately, and the page is unlocked
10. The memory manager returns to the interrupt routine that called it (most likely, in the context of *another* process!)
11. The interrupt routine restores the process state, reloads the registers, and returns from the interrupt

A Few Notes

- The paging discipline described here is *Paging on Demand* (a page is brought into memory when the page fault occurs)
- When no frame is free, it takes almost twice as long to fix the page fault than in the presence of a free frame. This is mitigated by using a *paging daemon*, which pages out frames when it notices that very few free frames are available
- But so far we have never said how the victim for paging out is selected!

So, *how* can we select a page to evict?

Page Replacement Algorithms

- It is possible to pick a victim randomly, but the idea is to select a page that would not be used soon—too much overhead is involved!
- There are tons of page replacement algorithms
- For each such (implementable) algorithm, it is possible to write a program that would perform the worst if this algorithm is chosen...
- Well, we will start with an unimplementable algorithm!

Page Replacement Algorithms

1. *Optimal* (unimplementable, but essential for benchmarking)
2. *Most-Recently-Used (MRU)*
3. *Not-Recently-Used (NRU)*
4. *Least-Frequently-Used (LFU)*
5. *First-In-First-Out (FIFO)*
   - *Second Chance*
     - *Clock*
6. *Least-Recently-Used (LRU)*
   - *Not-Frequently-Used (NFU)*
     - *Aging*

The *Optimal (OPT)* Page Replacement Algorithm

- Label each page with the number of instructions that will be executed *before* this page is referenced (*is it possible?*)
- Update all labels after each instruction (*is it feasible?*)
- Always page out the frame whose page has the largest label

Although this algorithm cannot be implemented, it can be used in simulations to compare a program’s performance under the OPT with that under another (implementable) page replacement algorithm. (The program can be run to determine its memory reference strings and thus the labels to be used in the simulation.)

Most-Recently-Used (MRU)

- The page that was just accessed is replaced (that simple!)

Least-Frequently-Used (LFU)

- Keep a use count for each page (in expensive hardware)
- Evict a page that has the smallest count

A Page Table Entry Revisited Again

- A *Valid/Invalid* bit indicates whether the page is in the *virtual* address space (or in secondary storage)
- A *Clean/Dirty* bit indicates whether the page had been written to (and thus needs to be saved to the disk)
- A *Locked/Unlocked* bit indicates whether the page may not/may be touched

0	7	r	v	c	u	**r**
1	5	r w	v	d	u	**n**
2	3	xr	v	c	u	**r**
3	9	xr	v	c	u	**r**
4	x		i		u	**r**
5	1		v	c	l	**n**
6	4	x	v	c	u	**r**

- A *referenced/not-referenced* bit indicates whether the page was recently (e.g., since the last clock interrupt) read

The *Not-Recently-Used (NRU)* Page Replacement Algorithm

- The operating system (possibly, with the help of hardware) keeps track of the pages' being read and written to for a period of time
- The pages are then broken into four classes:
  1. Never read, never written to
  2. Never read, written to
  3. Read, never written to
  4. Read and written to

With the NRU, a page is removed at random from the lowest-numbered non-empty class. (The *strategy*: it is better to remove a dirty page than a heavily used clean one.)

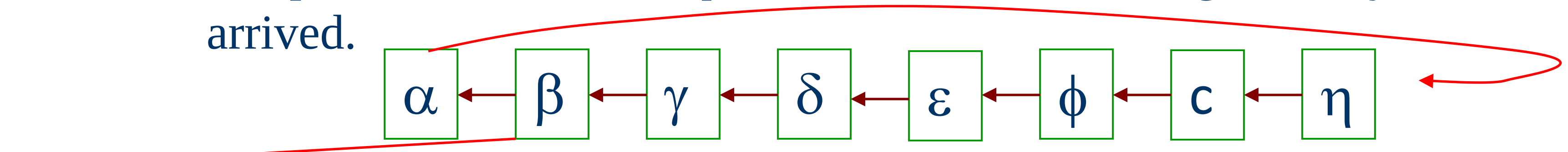
The *First-In-First-Out (FIFO)* Page Replacement Algorithm

- The operating system keeps the list of pages in the order they were brought into memory

With FIFO, a page in front of the list is always removed. (The *strategy*: if it has not been referenced for a long time, it won't be used soon. Well...)

The *Second Chance* Page Replacement Algorithm

- The *reference* bit of the oldest page is inspected. If its value is *n*, it is replaced immediately but if the value is *r*, it is given a second chance: it is put at the end of the queue and marked as though it has just arrived.

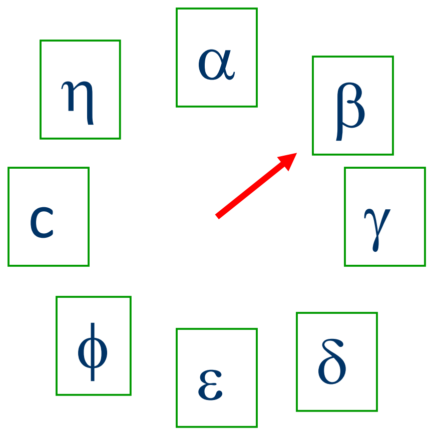


Arrival Time:	10	12	14	18	21	28	34	48
Reference bit:	r	n	r	r	r	n	n	n

The *Second Chance* is a variation on FIFO: it replaces the oldest page (as in FIFO), but only on the condition that it has not been referenced. (The strategy: it is better to remove an old unreferenced page.)

The *Clock* Page Replacement Algorithm

- The *clock* structure, keeps the pages in the circular list. The hand of the clock points at the oldest page. At the time of a page fault, if the page is unreferenced, it is removed; otherwise, the hand moves to the next page in the list.



The *Clock* algorithm is but an *implementation* of the *Second Chance* algorithm

The *Least-Recently-Used (LRU)* *Page Replacement Algorithm*

- With the *LRU* algorithm, a page that has not been used for a long time (huh?! what does it mean "a long time"?) is selected as a victim for replacement.
  1. *LRU* differs from *FIFO* in that *FIFO* keeps track of the time since the page was brought into a frame rather than the time since it was last used.
  2. *LRU* differs from *Optimal* in that *Optimal* is aware of the future, while *LRU* uses what it knows about the past. (In a sense, *Optimal* is a reverse of *LRU*—see the Book for more on this subject.)

The *strategy*: if a page has not been used for a long time, it won't be used soon.

LRU: Problems and Solutions

- It is very expensive to implement LRU. To do so one must
  1. Maintain a list of all pages in memory—a piece of cake unless one needs to
  2. Update it on every memory reference
- What is needed to perform the task is either expensive hardware or clever software approximation

Hardware Solutions for LRU

1. Each page table entry must have a 64-bit counter, which is automatically incremented on each memory reference. Then, on page fault, the OS selects a *(why “a” and not “the”?)* page with the lowest counter value as the victim
2. For n frames, maintain an nxn bit-matrix. If page i is referenced, the i-th row is set to all “1s”, and then the i-th column is set to all “0s”. On page fault, the entry, whose row has the lowest binary value, is chosen

Reformulating the Matrix Algorithm

When page *i* is referenced

1. Make all entries in the *i*-th row "1"

2. Make all entries in the *i*-th column "0"

$$\begin{array}{ccccc} a_{00} & & \ldots & & a_{0,n-1} \\ & & \cdots & & \\ 1 & \ldots & 0 & \ldots & 1 \\ & & \cdots & & \\ a_{n-1,0} & & \ldots & & a_{n-1,n-1} \end{array}$$

On a page fault, select the value of *k* that minimizes

$$b(k) = \sum_{l=0}^{n-1} 2^l a_{kl}$$

An Assignment

Prove that the Matrix implementation of the LRU algorithm works. In other words, prove that

1. If the memory page *k* has just been used, then the binary value of the *k*-th row, $b(k)$ is larger than $b(i)$, for all *i* such that $i \geq 0$; $i < n$, and $i \neq k$; and

2.The inequality $b(i) \geq b(j)$ holds until page *j* is used.

The Matrix LRU Algorithm Example

Consider a page reference string : 0, 1, 3, 2, 0

$$
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0
\end{array}
\qquad
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
2 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0
\end{array}
\qquad
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 \\
2 & 0 & 0 & 0 & 0 \\
3 & 1 & 1 & 1 & 0
\end{array}
\qquad
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 \\
2 & 1 & 1 & 0 & 1 \\
3 & 1 & 1 & 0 & 0
\end{array}
\qquad
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 1 \\
3 & 0 & 1 & 0 & 0
\end{array}
$$

Victims: 1-3 Victims: 2-3 Victim: 2 Victim: 0 Victim: 1

The *Not-Frequently-Used (NFU)* Page Replacement Algorithm

- This is a software implementation (or rather *simulation*) of the LRU:
  - Each page has a software counter, which is initialized to zero
  - On each clock tick, the OR function of the *clean(0)/dirty(1)* and *not referenced(0)/referenced(1)* bits are added to the counter
  - When a page fault occurs, the page whose counter is the lowest is removed.

A problem: NFU does not forget the history of the memory use, so the presence of the pages with high counters that are no longer used might dominate

The *Aging* Page Replacement Algorithm

This is a slight modification of the *NFU*, which fixes the “long memory” problem:

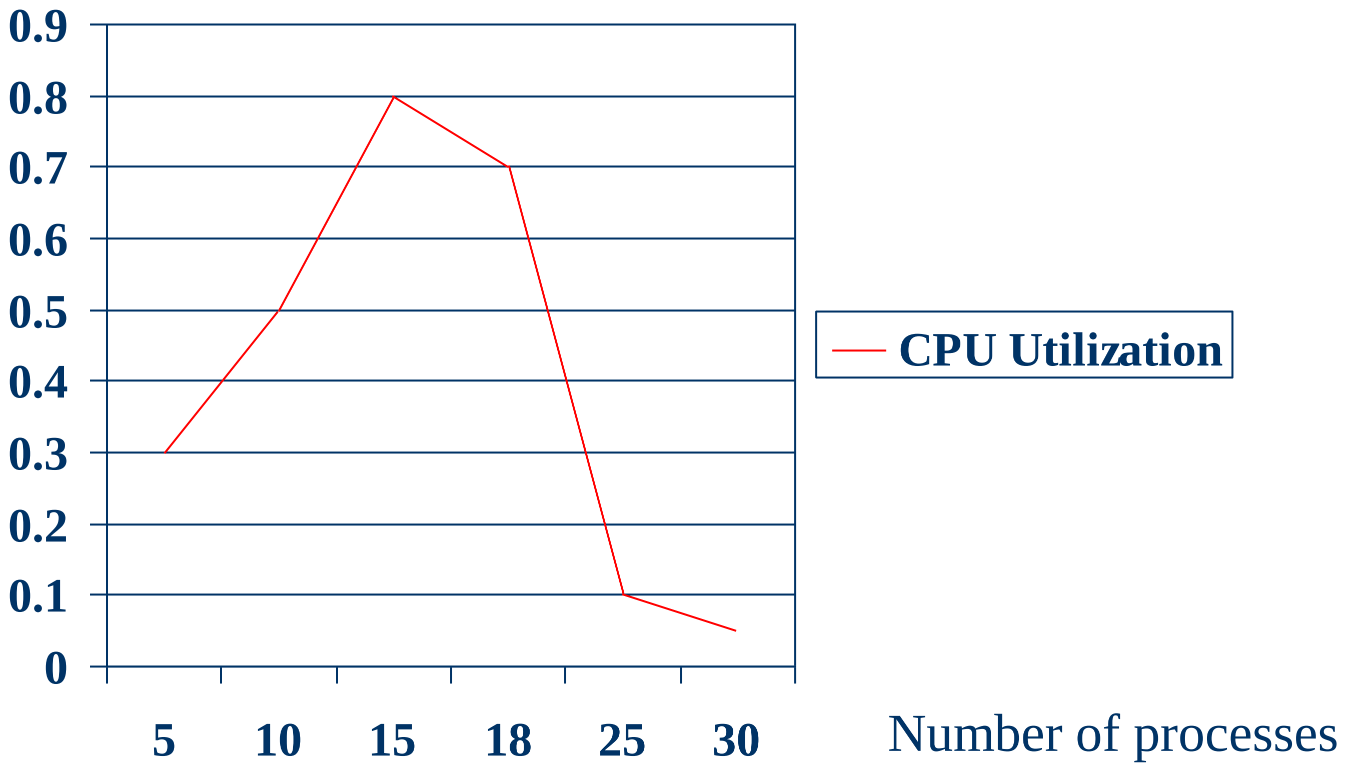
- Each page has a software counter, which is initialized to zero
- On each clock tick, the counters are shifted right (to forget one bit of history) and the OR function of the *clean/dirty* and *referenced/not referenced* bits is added to the left-most bit of the counter
- When a page fault occurs, the page whose counter is the lowest is removed.

Aging fixes the *NFU* problem.

Happy Summary: Page Replacement Algorithms

- *Optimal* (unimplementable, but essential for benchmarking)
- *Not-Recently-Used (NRU)*
- *Most-Recently-Used (MRU)*
- *Least-Frequently-Used (LFU)*
- *First-In-First-Out (FIFO)*
  - *Second Chance*
    - *Clock*
- *Least-Recently-Used (LRU)*
  - *Not-Frequently-Used (NFU)*
    - *Aging*

Thrashing


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15
18
25
30
CPU Utilization
Number of processes

Thrashing

- A situation in which paging becomes the *major* system activity (so that it appears to the users that the system is not progressing—an observable degradation of performance), is called *thrashing*
- Thrashing happens because there are very few pages left in the system, so the frequently used pages need to be replaced to maintain the progress. As the result, CPU spends more time paging than executing processes
- One solution to thrashing is to swap out all low-priority processes and thus decrease the degree of multiprogramming

But the Memory is Getting Cheaper and Cheaper!

So, would increasing the memory, say by a third, help?

Let Us See...

- We are executing a program with *five virtual memory pages*, which is producing the reference string

  0 1 2 3 0 1 4 0 1 2 3 4

- We have the memory with three *frames* (initially empty), and we are using FIFO:

	0	1	2	3	0	1	4	0	1	2	3	4
	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
			0	1	2	3	0	0	0	1	4	4
	F	F	F	F	F	F	F			F	F	

9 page faults

Let us Add One More Frame

- We are executing a program with five virtual memory pages, which is producing the reference string

0 1 2 3 0 1 4 0 1 2 3 4

- We have the memory with ~~three~~ **four** page frames (initially empty), and we are using FIFO:

	0	1	2	3	0	1	4	0	1	2	3	4	
	0	1	2	3	3	3	4	0	1	2	3	4	**10 page faults!**
		0	1	2	2	2	3	4	0	1	2	3	
			0	1	1	1	2	3	4	0	1	2	**Huh?**
				0	0	0	1	2	3	4	0	1	
	F	F	F	F			F	F	F	F	F	F	

Belady's Anomaly and the World After Its Discovery

- The same program that causes nine page faults in the three-frame memory may cause ten page faults in the four-frame memory when FIFO is used!
- After *Belady*'s discovery (in 1969), a whole theory of paging was built. A class of paging algorithms whose behavior does *not* deteriorate on the same reference string after a frame is added was determined
- The algorithms in the class are called *Stack Algorithms*. *LRU* is a stack algorithm; *FIFO* is not (of course).

Stack Algorithms: Formal Definition

- Let
  - $\omega = (s_1, s_2, \ldots s_n)$ be a reference string processed in the memory containing *m* frames; and
  - *M(m, ω)* denote the state of the memory after *ω* has been processed.
- A page replacement algorithm is called a *stack* algorithm if

$$M(m, \omega) \subseteq M(m+1, \omega).$$

Another way to look at it

- Given ω, there exists such a permutation

$\pi(\omega) = \{\pi_1(\omega), \pi_2(\omega), \cdots, \pi_n(\omega)\}$ of virtual pages $\{1, 2, \ldots, n\}$ that for all m = 1, 2, 3,…,

$$M(m,\omega) = \{\pi_1(\omega), \pi_2(\omega), \cdots, \pi_m(\omega)\}$$

- In other words, the contents of real memory are always determined by the first *m* pages of *π(ω)* called the *stack.*

- For LRU, *M(m, ω)* = *{m most recently visited pages}* = *M (m+1, ω)*

Adaptation to *phase transition*

- Consider the string $\omega = (1,2,3)^k (4,5,6)^l$
- With three frames, MRU will result in *3(l+1)* page faults, and LFU will result in *3[min(k,l)+1]* page faults, but FIFO and LRU will result only in six page faults.

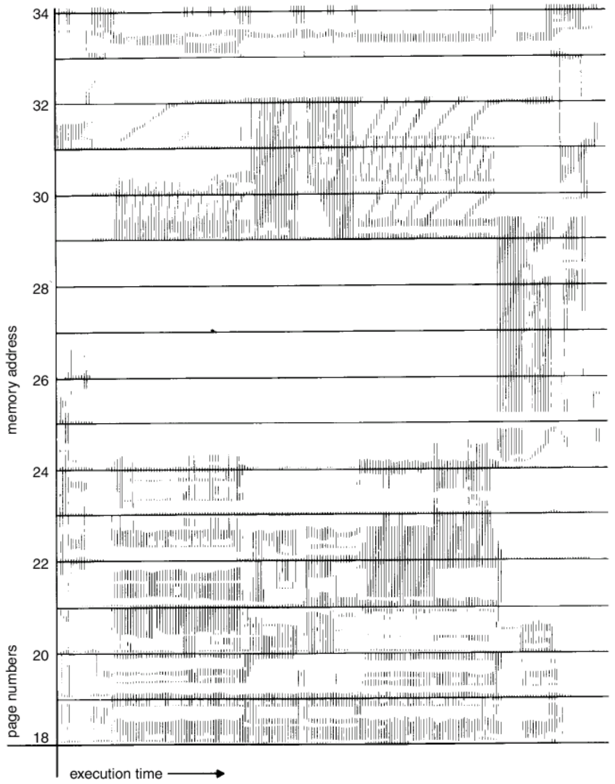
Homework assignment: Prove this!

The *Working Set* Model

- It is easy to write a program that would make any specific paging algorithm to work badly—such a program could generate a page fault on any reference
- But processes that execute real programs exhibit *locality reference*—during the execution, a process uses a relatively small set of pages for a long time. Such stable set is called the *working set* of a process

Locality in Memory Refences

From the Book



References are localized to a few pages, and stay this way for a while

In the beginning, pages 18-24 and 30-34 make a working set, then pages 25-29 make another one

Working with Working Sets

- The operating system uses heuristics to determine a working set of each process and then maintains the frame allocation so as to ensure the working set is always in memory
  1. First, the number of references to determine a stable set is guessed (typically, *10,000* references)
  2. Then the timer value is chosen. The timer causes an interrupt, on which the system clears the *dirty* and *reference* bits, so that it can determine on the next interrupt whether the set has changed
- If it happens that the overall number of free frames falls short of the working sets' requirements, the execution of the low priority processes is postponed

Frame Allocation and Replacement

- Each process can be given either the same number of frames that any other process gets (*equal allocation*) or have the number of frames allocated in proportion to its size
- Typically, a number of frames are given for the stack, text, and heap segments each
- On page fault, either the whole memory is searched for a free (or evicted) page—this is called *global replacement*, or only the memory of the executing process (*local replacement*)
- The local replacement is good in localizing thrashing to just one process—not the whole system

Selecting the Page Size

- Determining the optimum page size requires analysis of several competing factors
  - The smaller the page is, the less the internal fragmentation; for the same reason, smaller pages result in less unused memory
  - But small pages—as we saw in the last lecture—require larger page tables, so large memory address registers dictate large page size
  - Because most of the disk time is spent on *rotational* and *seek* procedures, the transfer of a small page takes about the same time as the transfer of a large page. Then much more memory is transferred in the same amount of time, if large page size is used
  - The space occupied by the page table increases as the page size decreases

Selecting the Page Size (cont.)

- Let
  - *s* be the average process size,
  - *p* be the page size
  - e be the size of a page table entry
- Then the average number of pages per process is *s/p*, and the average page table size is *se/p*
- Given that the average internal fragmentation overhead is *p/2*, the overall overhead is

$$overhead(p) = \frac{se}{p} + \frac{p}{2}$$

Selecting the Page Size (cont.)

$$overhead(p) = \frac{se}{p} + \frac{p}{2}$$

Optimal points, if any, are the roots of the equation:

$$overhead'(p) = -\frac{se}{p^2} + \frac{1}{2} = 0;$$

The only meaningful solution of the above (and thus the optimal point—but please check that it *is* optimal) is

$$p = \sqrt{2se}.$$