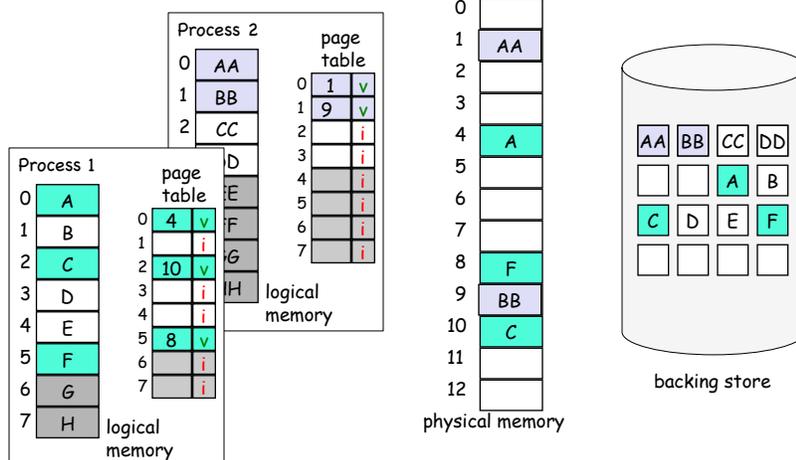


## Virtual Memory

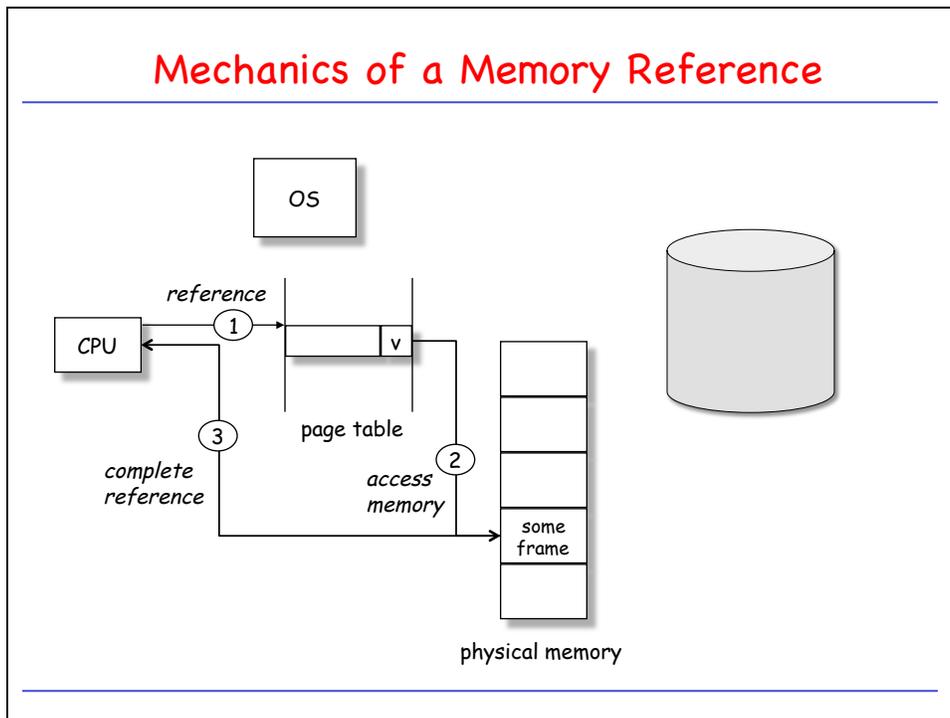
- Overview / Motivation
- Simple Approach: Overlays
- Locality of Reference
- Demand Paging
- Policies
  - Placement
  - Replacement
  - Allocation
- Case Studies: Unix SystemV
- *Reading: Silberschatz, Chapter 9*

## Demand Paging

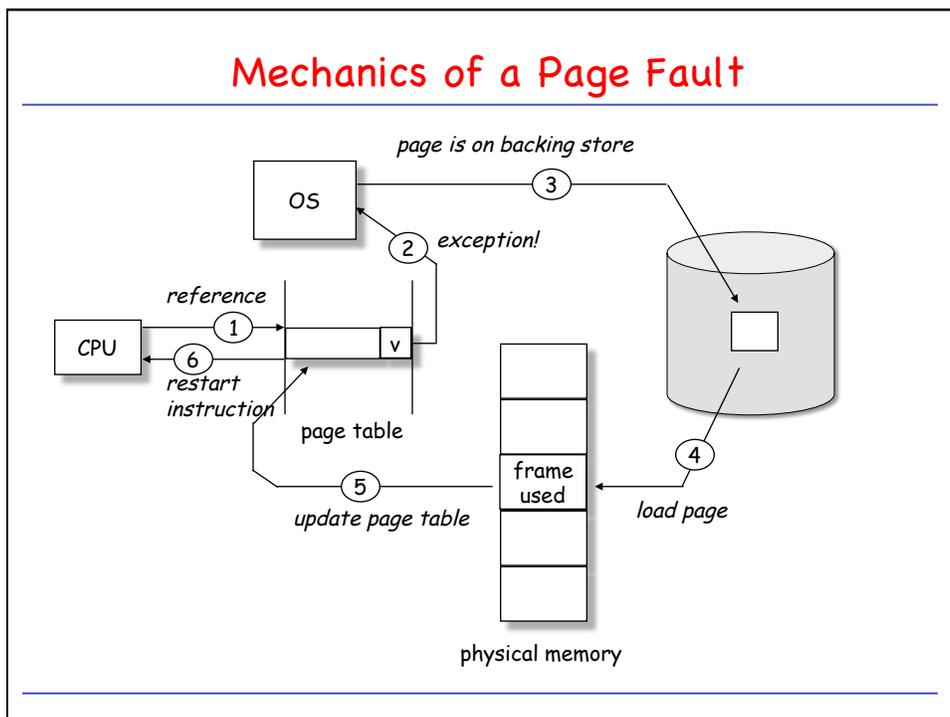


- "Lazy Swapper": only swap in pages that are needed.
- Whenever CPU tries to access a page that is not swapped in, a **page fault** occurs.

## Mechanics of a Memory Reference



## Mechanics of a Page Fault



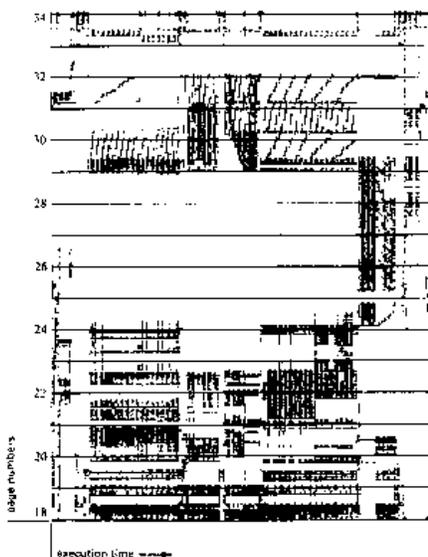
## Locality of Reference

- Page faults are expensive!
- **Thrashing:** Process spends most of the time paging in and out instead of executing code.
- Most programs display a pattern of behavior called the **principle of locality of reference**.

### Locality of Reference

A program that references a location  $n$  at some point in time is likely to reference the same location  $n$  and locations in the immediate vicinity of  $n$  in the near future.

## Memory Access Trace



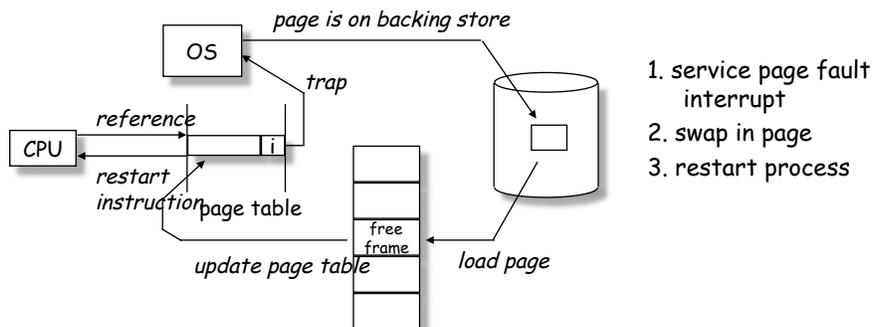
## Architectural Considerations

- Must be able to restart any instruction after a page fault.
- e.g.
 

ADD A,B TO C
- What about operations that modify several locations in memory?
  - e.g. block copy operations?
- What about operations with side effects?
  - e.g. PDP-11, 80x86 auto-decrement, auto-increment operations?
  - Add mechanism for OS to "undo" instructions.

## Performance of Demand Paging

- Effective Memory Access time  $ema$ :
 
$$ema = (1-p) * ma + p * \text{"page fault time"}$$
- where
  - $p$  = probability of a page fault
  - $ma$  = memory access time
- Operations during Page Fault:



## OS Policies for Virtual Memory

---

- **Fetch Policy**
    - How/when to get pages into physical memory.
    - demand paging vs. prepaging.
  - **Placement Policy**
    - Where in physical memory to put pages.
    - Only relevant in NUMA machines.
  - **Replacement Policy**
    - Physical memory is full. Which frame to page out?
  - **Resident Set Management Policy**
    - How many frames to allocate to process?
    - Replace someone else's frame?
  - **Cleaning Policy**
    - When to write a modified page to disk.
  - **Load Control**
- 

## Configuring the Win2k Memory Manager

---

- Registry Values that Affect the Memory Manager:

`ClearPageFileAtShutdown`

`DisablePagingExecutive`

`IoPageLockLimit`

`LargePageMinimum`

`LargeSystemCache`

`NonPagedPoolQuota`

`NonPagedPoolSize`

`PagedPoolQuota`

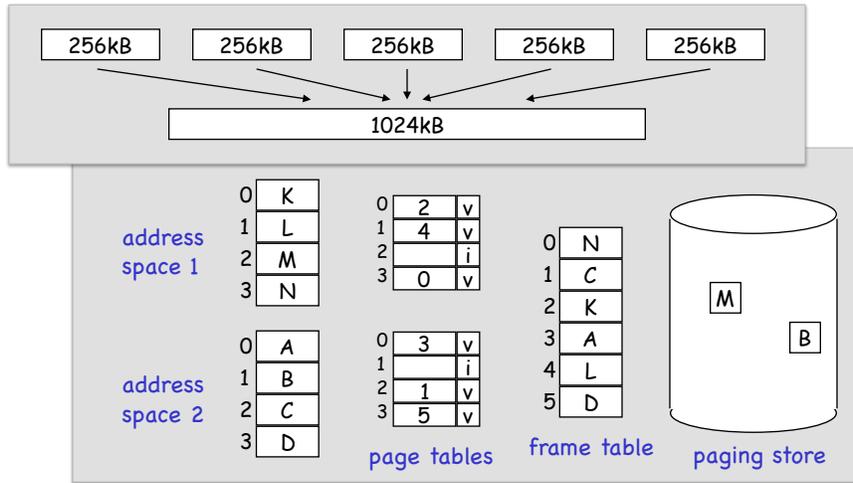
`PagedPoolSize`

`SystemPages`

---

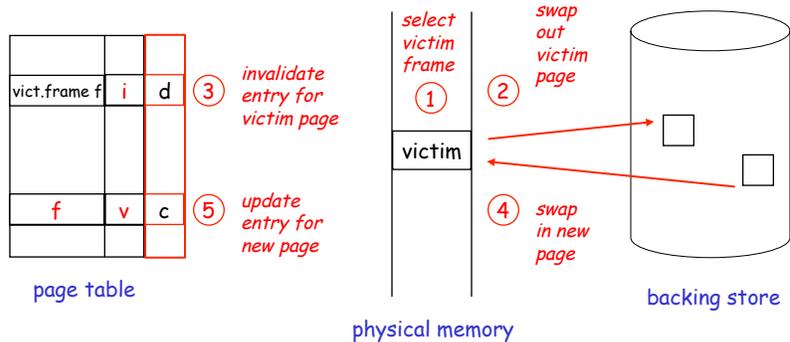
## Page Replacement

- Virtual memory allows higher degrees of multiprogramming by over-allocating memory.



## Mechanics of Page Replacement

- Invoked whenever no free frame can be found.



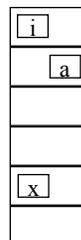
- Problem:** Need two page transfers
- Solution:** **Dirty bit.**

## Page Replacement Algorithms

- Objective: Minimize page fault rate.
- Why bother?

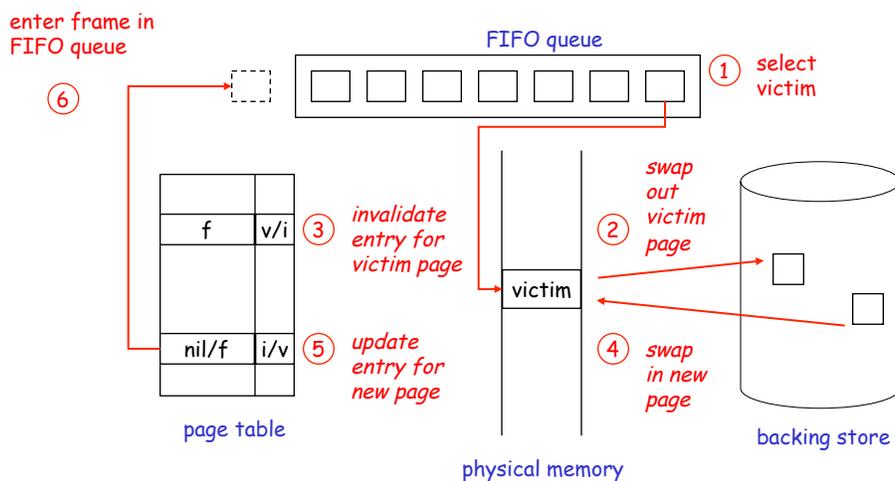
- Example

```
for(int i=0; i<10; i++) {
    a = x * a;
}
```



- Evaluation: Sequence of memory references: **reference string**.

## FIFO Page Replacement



## FIFO Page Replacement (cont.)

- Example:

time	1	2	3	4	5	6	7	8	9	10			
reference string	c	a	d	b	e	b	a	b	c	d			
frames	a	a	a	a	a	e	e	e	e	e	d		
	b	b	b	b	b	b	b	a	a	a	a		
	c	c	c	c	c	c	c	c	b	b	b		
	d	d	d	d	d	d	d	d	d	c	c		
									!	!	!	!	!

- **Advantage:** simplicity
- **Disadvantage:** Assumes that pages residing the longest in memory are the least likely to be referenced in the future (does not exploit *principle of locality*).

## Optimal Replacement Algorithm

- Algorithm with lowest page fault rate of all algorithms:

Replace that page which will not be used for the longest period of time (in the future).

- Example:

time	1	2	3	4	5	6	7	8	9	10		
reference string	c	a	d	b	e	b	a	b	c	d		
frames	a	a	a	a	a	a	a	a	a	a	d	
	b	b	b	b	b	b	b	b	b	b	b	
	c	c	c	c	c	c	c	c	c	c	c	
	d	d	d	d	d	e	e	e	e	e	e	
											!	!

## Approximation to Optimal: LRU

- **Least Recently Used:** replace the page that has not been accessed for longest period of time (in the *past*).
- Example:

time	1	2	3	4	5	6	7	8	9	10	
reference string	c	a	d	b	e	b	a	b	c	d	
frames	a	a	a	a	a	a	a	a	a	a	
	b	b	b	b	b	b	b	b	b	b	
	c	c	c	c	e	e	e	e	e	d	
	d	d	d	d	d	d	d	d	c	c	
									!	!	!

## LRU: Implementation

- Need to keep **chronological history of page references**; need to be reordered upon each reference.
- **Stack:**

stack	?	c	a	d	b	e	b	a	b	c	d
	?	?	c	a	d	b	e	b	a	b	c
	?	?	?	c	a	d	d	e	e	a	b
	?	?	?	?	c	a	a	d	d	e	a

- **Capacitors:** Associate a capacitor with each memory frame. Capacitor is charged with every reference to the frame. The subsequent exponential decay of the charge can be directly converted into a time interval.
- **Aging registers:** Associate aging register of  $n$  bits ( $R_{n-1}, \dots, R_0$ ) with each frame in memory. Set  $R_{n-1}$  to 1 for each reference. Periodically shift registers to the right.

### Approximation to LRU: Clock Algorithm

- Associate a *use\_bit* with every frame in memory.
  - Upon each reference, set *use\_bit* to 1.
  - Keep a pointer to first "victim candidate" page.
  - To select victim: If current frame's *use\_bit* is 0, select frame and increment pointer. Otherwise delete *use\_bit* and increment pointer.

time	1	2	3	4	5	6	7	8	9	10
reference string	c	a	d	b	e	b	a	b	c	d
frames	a/1 b/1 c/1 d/1	a/1 b/1 c/1 d/1	a/1 b/1 c/1 d/1	a/1 b/1 c/1 d/1	a/1 b/1 c/1 d/1	e/1 b/0 c/0 d/0	e/1 b/1 c/0 d/0	e/1 b/0 a/1 d/0	e/1 b/1 a/1 c/1	d/1 b/0 a/0 c/0
						!	!	!	!	

### Improvement on Clock Algorithm (Second Chance Algorithm)

- Consider read/write activity of page: *dirty\_bit* (or *modify\_bit*)
- Algorithm same as clock algorithm, except that we scan for frame with both *use\_bit* and *dirty\_bit* equal to 0.
- Each time the pointer advances, the *use\_bit* and *dirty\_bit* are updated as follows:

	ud	ud	ud	ud
before	11	10	01	00
after	01	00	00*	(select)

- Called *Second Chance* because a frame that has been written to is not removed until two full scans of the list later.
- Note: Stallings describes a slightly different algorithm!

## Improved Clock (cont)

- Example:

time	1	2	3	4	5	6	7	8	9	10	
reference string	c	a <sup>w</sup>	d	b <sup>w</sup>	e	b	a <sup>w</sup>	b	c	d	
frames	a/10	a/10	a/11	a/11	a/11	a/00*	a/00*	a/11	a/11	a/11	
	b/10	b/10	b/10	b/10	b/11	b/00*	b/10*	b/10*	b/10*	b/10*	
	c/10	c/10	c/10	c/10	c/10	e/10	e/10	e/10	e/10	e/10	
	d/10	d/10	d/10	d/10	d/10	d/00	d/00	d/00	d/00	c/10	
						!			!	!	

## The Macintosh VM Scheme (see Stallings)

- Uses **use\_bit** and **modify\_bit**.
- **Step 1:** Scan the frame buffer. Select first frame with **use\_bit** and **modify\_bit** cleared.
- **Step 2:** If Step 1 fails, scan frame buffer for frame with **use\_bit** cleared and **modify\_bit** set. During scan, clear **use\_bit** on each bypassed frame.
- Now all **use\_bit**'s are cleared. Repeat Step 1 and, if necessary, Step 2.

## The Macintosh Scheme (cont)

- Example:

time	1	2	3	4	5	6	7	8	9	10	
reference string	c	a <sup>w</sup>	d	b <sup>w</sup>	e	b	a <sup>w</sup>	b	c	d	
frames	a/10	a/10	a/11	a/11	a/11	a/01	a/01	a/11	a/11	a/11	
	b/10	b/10	b/10	b/10	b/11	b/01	b/11	b/11	b/11	b/11	
	c/10	c/10	c/10	c/10	c/10	e/10	e/10	e/10	e/10	e/10	
	d/10	d/10	d/10	d/10	d/10	d/00	d/00	d/00	d/00	c/10	

!
!
!

## Resident Set Management

- **Local vs. Global** replacement policy:
  - The page to be replaced is selected from the resident set of pages of the **faulting** process. (local)
  - The page to be replaced may belong to **any** of the processes in memory.
- Each program requires a certain **minimum set of pages** to be resident in memory to run efficiently.
- The size of this set changes dynamically as a program executes.
- This leads to algorithms that **attempt to maintain an optimal resident set** for each active program. (Page replacement with **variable** number of frames.)

## The Working Set Model

- **Working Set**  $W(t, \Delta)$ : set of pages referenced by process during time interval  $(t-\Delta, t)$

$$\|W(t, 1)\| = 1 \quad 1 \leq \|W(t, \Delta)\| \leq \min(\Delta, N)$$

- The storage management strategy follows two rules:
  - At each reference, the current working set is determined and only those pages belonging to the working set are retained in memory.
  - A program may run only if its entire current working set is in memory.
- Underlying Assumption: cardinality of working set remains constant over small time intervals.

## Working Set Model (cont.)

- Example: ( $\Delta = 4$ )

time	1 2 3 4 5 6 7 8 9 10													
reference string	e	d	a	c	c	d	b	c	e	c	e	a	d	
working set				a	a	a	a						a	a
								b	b	b	b			
								c	c	c	c	c	c	c
								d	d	d	d	d	d	d
		e	e	e	e					e	e	e	e	e

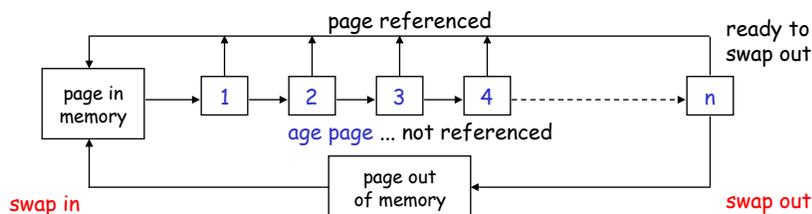
- Problems:
  - Difficulty in keeping track of working set.
  - Estimation of appropriate window size  $\Delta$ .

## Improve Paging Performance: Page Buffering

- Victim frames are not overwritten directly, but are removed from page table of process, and put into:
  - free frame list (clean frames)
  - modified frame list (modified frames)
- Victims are picked from the free frame list in FIFO order.
- If referenced page is in free or modified list, simply reclaim it.
- Periodically (or when running out of free frames) write modified frame list to disk.

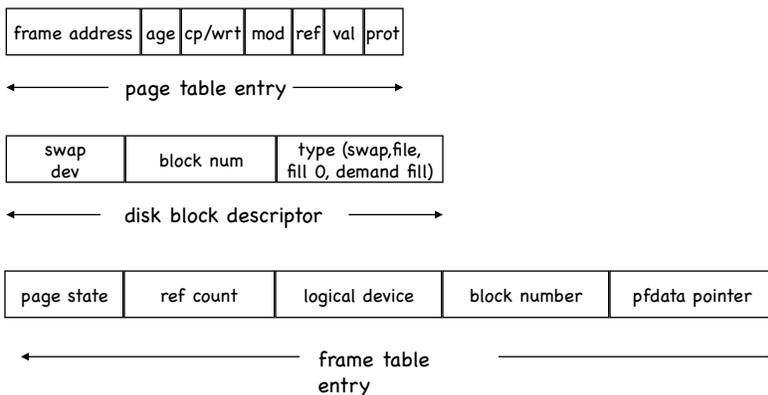
## Page Buffering and Page Stealer

- Kernel process (e.g., **pageout** in Solaris) swaps out memory frames that are no longer part of a working set of a process.
- Periodically increments **age field** in valid pages.



- **Page stealer** wakes up when available free memory is below *low-water mark*. Swaps out frames until available free memory exceeds *high-water mark*.
- Page stealer collects frames to swap and swaps them out in a single run. Until then, frames still available for reference.

## Implementation of Demand Paging in UNIX SVR4



## Linux Frame Table

- Every page is represented by:
 

```

struct page {
    long flags;           // dirty, locked, etc.
    atomic_t count;      // reference counter
    struct list_head list;
    struct AS *mapping;   // address space associated with page
    unsigned long index;
    struct list_head lru;
    (pte)
    (private)
    void * virtual;       // virtual address (could be null)
    /* ... etc. */
}
      
```

## Demand Paging on Less-Sophisticated Hardware

- Demand paging most efficient if hardware sets the *reference* and *dirty* bits and causes a protection fault when a process writes a page whose *copy\_on\_write* bit is set.
- Can duplicate *valid* bit by a *software-valid* bit and have the kernel turn off the *valid* bit. The other bits can then be simulated in software.
- Example: Reference Bit:
  - If process references a page, it incurs a page fault because *valid* bit is off. Page fault handler then checks *software-valid* bit.
  - If set, kernel knows that page is really valid and can set *software-reference* bit.

Hardware Valid	Software Valid	Software Reference	Hardware Valid	Software Valid	Software Reference
Off	On	Off	On	On	On
before referencing page			after referencing page		

## fork () System Call in Paging Systems

- Naive: **fork ()** makes a physical copy of parent address space. However, **fork ()** mostly followed by an **exec ()** call, which overwrites the address space.
- System V: Use *copy\_on\_write* bit:
  - During **fork ()** system call, all *copy\_on\_write* bits of pages of process are turned on. If either process writes to the page, incurs *protection fault*, and, in handling the fault, kernel makes a new copy of the page for the faulting process.
- BSD: Offers **vfork ()** system call, which does not copy address space. Tricky! (May corrupt process memory.)