LRU implementation

- Replace a page which has not been used for the longest period
- How to identify a page that was not accessed recently?

LRU implementation

- We need some hardware support to update the timestamp on every access

- x86 provide some help in the form of a reference bit (access bit)

- The access bit is set when a page is accessed by an instruction

Access bit

- mov $100, 0x10101 // write 100 to address 0x10101

0x10 — pte

access bit

Additional reference bit algorithm

- keeps n-bit timestamp for each page in memory
- At regular interval right shift the timestamp by one bit
- Shift the access bit to the high order bit of timestamp
- Replace all the pages with the lowest timestamp
  - or replace the next page in FIFO order

Additional reference bit algorithm

n-bit

1 0 1 0 1

100ms

0 1 1 1 1

1 0 0 0 0

Second chance algorithm

- A simplification of the previous algorithm is second chance algorithm when we don’t want to have additional storage to store the timestamp
  - i.e. n is zero

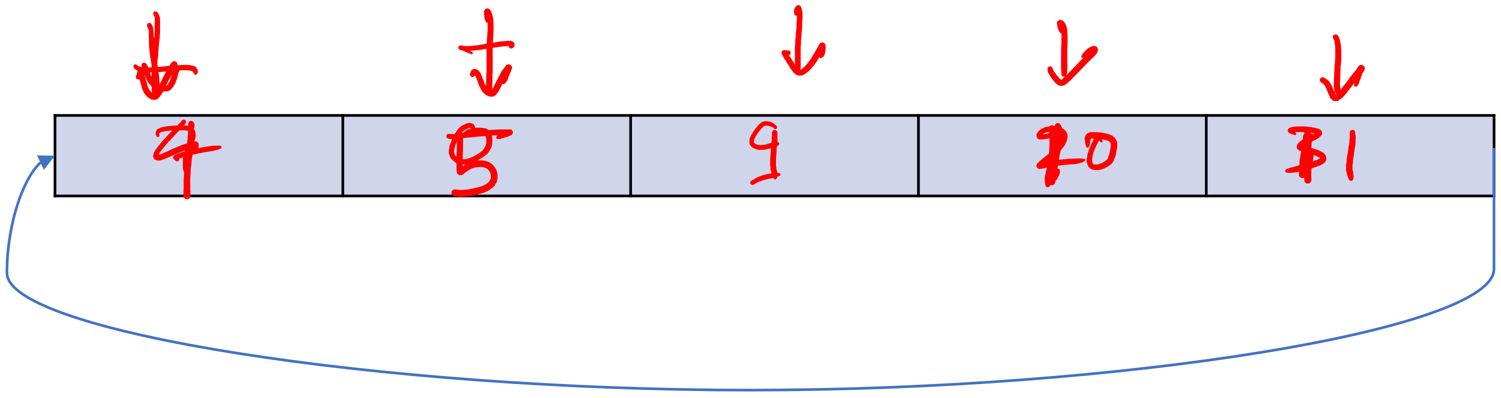
- The algorithm is similar to FIFO but gives more priority to a page which is accessed recently

FIFO

- FIFO algorithm can be implemented using a circular queue
- The size of the circular queue is the number of available physical pages
- When the queue is full, we need to select a victim that is going to be replaced
- A pointer (also called clock tick) points to the next victim in the queue
- After replacing the victim, the clock tick points to the next element in the queue

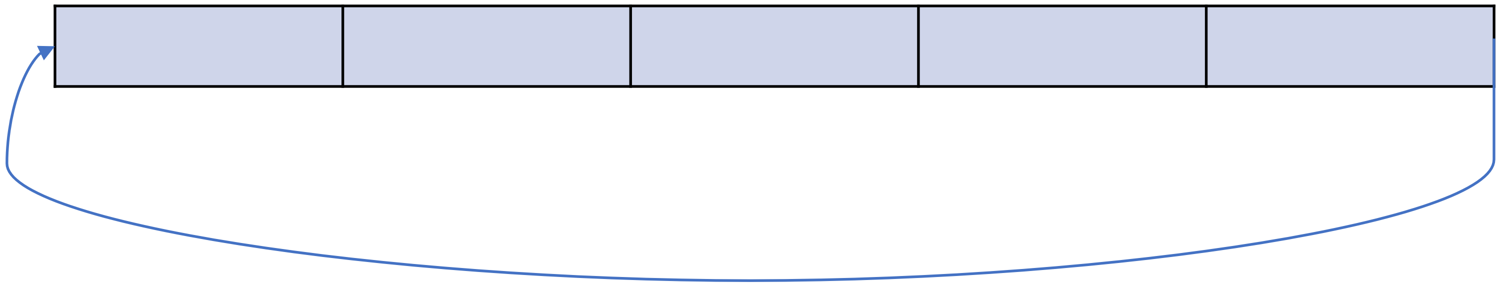
FIFO

- 7 0 1 2 3 4 5 9 10 11 12



FIFO

- 7 0 1 2 3 4 5 9 10

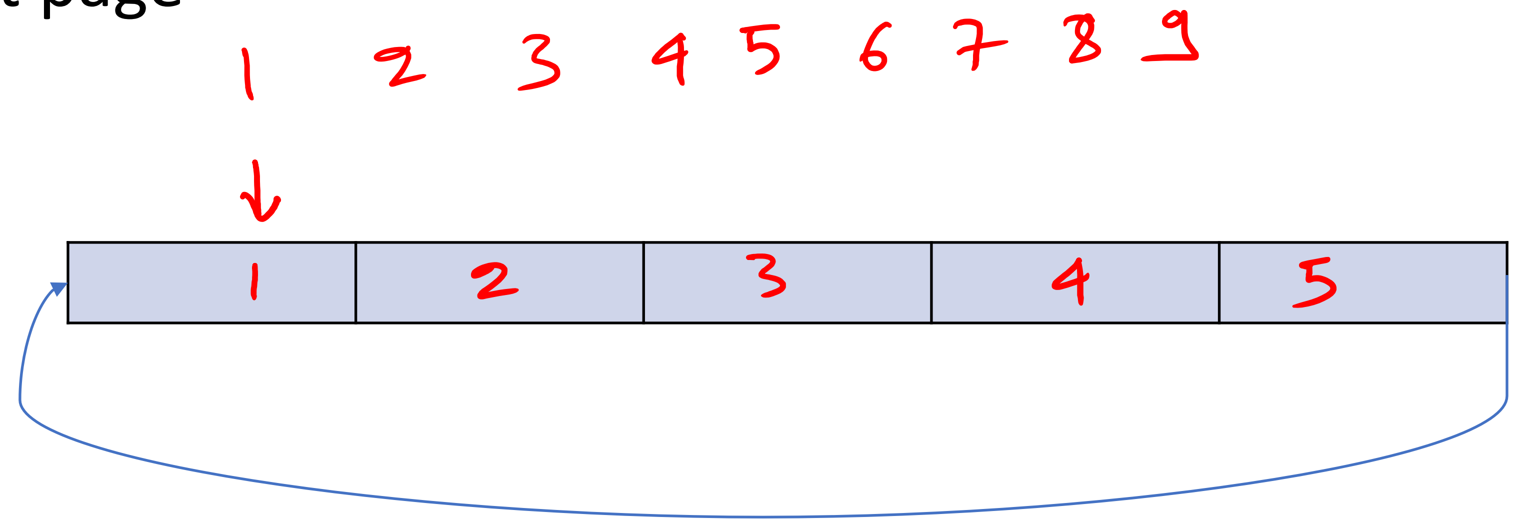


Second change algorithm

```
select_victim (struct queue_elem *clock) {
   while (1) {
         page = clock->page;
         if (!was_accessed (page)) /* check for access bit */
                return page;
         clear_access_bit (page); /* reset access bit */
         clock = clock->next;
    }
}
```

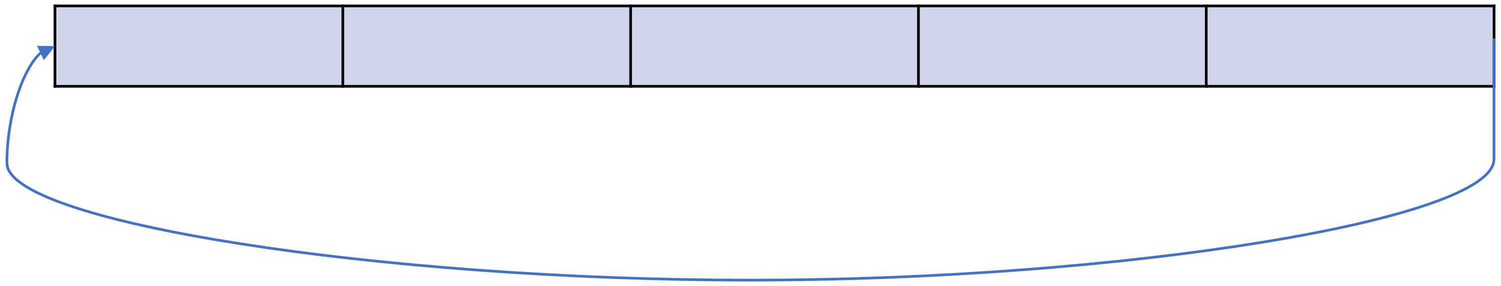
Second chance algorithm

- If a page was referenced recently, clear the access bit and move to the next page



Second chance algorithm

- If a page was referenced recently, clear the access bit and move to the next page



Dirty bit

- mov $100, 0x10101   // write 100 to address 0x10101

0x10

pte = pte (0x10)

pte -> dirty = 1;

Enhanced second chance algorithm

- In the case of memory mapped files, a page need not need to be written to the disk if it is not modified

char *addr = mmap(ls.txt);

- Page table entry also contains a dirty bit
  - dirty bit is set when an instruction writes to a page

*(addr + 100)

munmap (addr);

- The enhanced second chance algorithm gives preference to the dirty page over clean page

Enhanced second chance algorithm

```
select_victim (struct queue_elem *clock) {
  while (1) {
        page = clock->page;
        if (!was_accessed (page)) {   /* check for access bit */
             if (!was_written (page) || page->given_advantage)   /* check for dirty bit */
                 return page;
             page->given_advantage = true;
         } else {
             clear_access_bit (page);
             page->given_advantage = false;
        }
        clock = clock->next;
  }
}
```

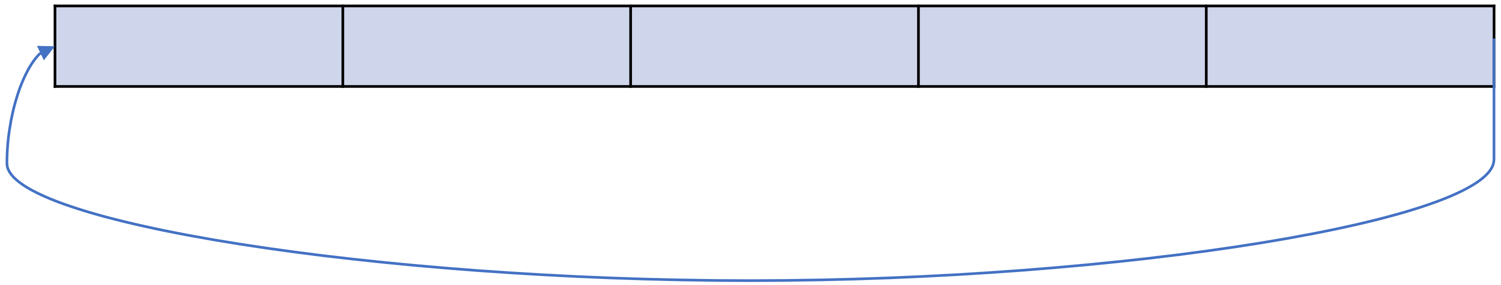
~~access dirty~~

~~0 0~~

dirty	access
0	0
1	0
0	1

Enhanced second chance algorithm

- If a page was referenced recently, clear the access bit and move to the next page
- Otherwise, if a page is dirty, give it another chance



Allocation of frames

- An OS must decide how page frames are allocated for a process
- Two allocation policies
  - Local
  - Global

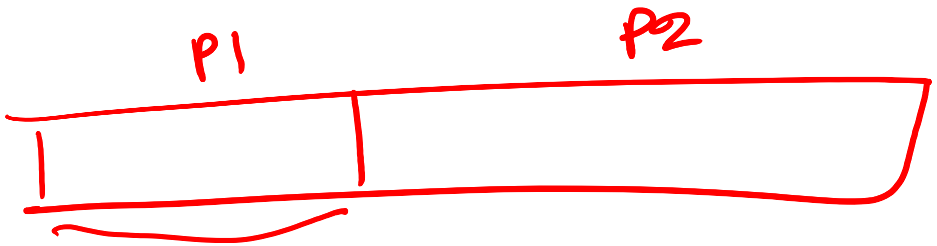
Local allocation

- In the local allocation scheme, each process is allocated a fixed set of physical frames
- The process must allocate physical pages from its assigned quota of physical frames
- One problem of this approach is underutilization of memory
- Number of page faults are deterministic for a given workload

Local allocation

P1 10 physical

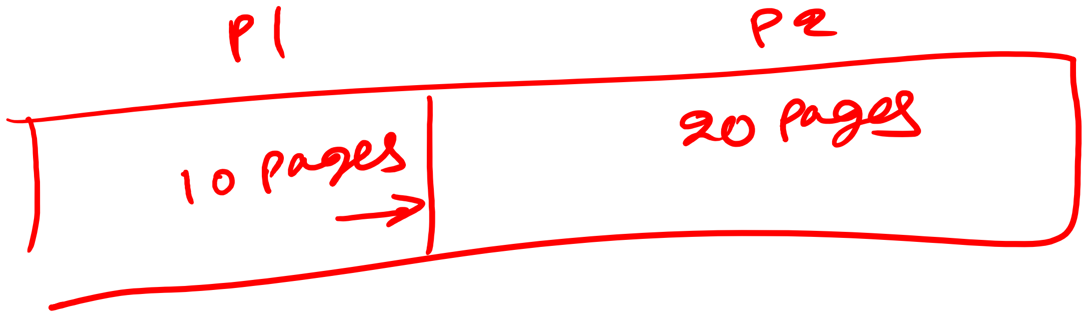
P2 20 physical



Global allocation

- OS allows processes to allocate from the set of all physical frames
- One process can cause replacement for other processes
- Number of page faults are not deterministic
- Good throughput (no underutilization)

Global allocation



Working set

- The working set of a process is the set of pages accessed in last Δ page references

- For example if Δ = 10

2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 ….

Δ t1

{1, 2, 5, 6, 7}

Δ t2

{3, 4}

Thrashing

- If the sum of working sets of active applications is larger than the total number of physical frames, then the applications will spend most of the time in swapping

- This results in very low CPU utilization because most of the time will be spent in disk I/O

- This is also called thrashing

P1 $\boxed{2}$ P3 $\boxed{8}$

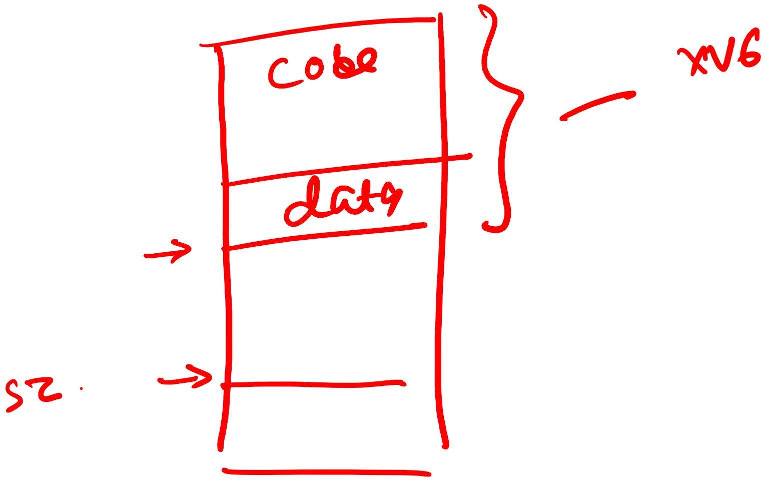
$10 > 8$

Thrashing

- To avoid thrashing the scheduler maintain two lists
  - active list
  - inactive list
- Applications may move between active and inactive list
- The scheduler picks a process from the active list
- Processes are added to the active list in such a way that the sum of working sets of active processes is less than the total number of physical frames

Thrashing

xv6




```
exec( )
{
    va = kalloc()
read(va, executable, size);
```

Other uses of page protection

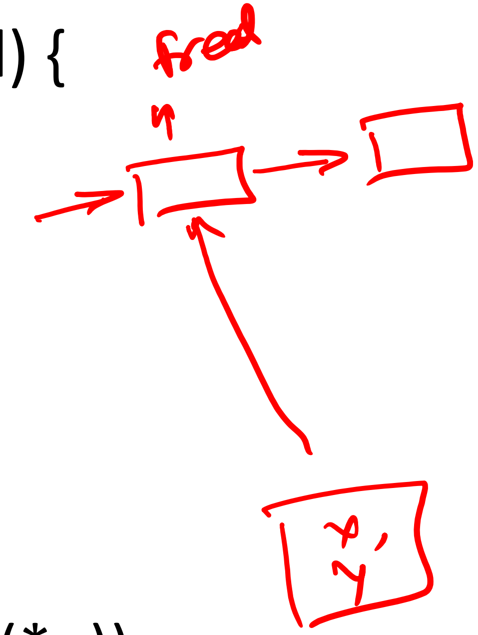
- Temporal safety

Temporal safety

```
struct List {
   int data;
   struct List *next;
};

struct  Point {
    int x;
    int y;
};
```

```
delete_node (struct List *l, int val) {
    while (l) {
        if (l->data == val)
            free (l);
     }
}
add_coordinate (int x, int y) {
    struct Point *p = malloc (sizeof(*p));
    p->x = x;  p->y = y;
    return p;
}
```



Temporal safety

```
struct List {
  int data;
  struct List *next;
};

struct  Point {
   int x;
   int y;
};
```

```
delete_node (struct List *l, int val) {
    while (l) {
        /* may access a dangling reference */
        if (l->data == val)
              free (l);
     }
}
add_coordinate (int x, int y) {
    struct Point *p = malloc (sizeof(*p));
    p->x = x;  p->y = y;
    return p;
}
```

Temporal safety

- Never reuse a virtual address
  - malloc always returns a new virtual address
- Free does nothing
- Why does this scheme work?
- What is this scheme not efficient?

On free take back physical page?

- Does this scheme work?

Allocate virtual pages for every allocation

VP1 = malloc (20);   // VP1 is a virtual page of 4096 bytes

VP2 = malloc (30); // VP2 != VP1 and VP2 is a virtual page of 4096 bytes

On free take back physical page?

- Throw dangling pointer exception if a page fault is encountered

- Does this scheme work?
  - Yes, but requires a lot more physical memory because every allocation needs to be aligned to the page size

Can we do better?

- Share physical pages

Share physical pages

- Allocate different virtual pages for every allocation but share the same physical pages

e.g.,

VP1 = malloc (20);   // VP1 points to PP1 (physical page)

next malloc (30) returns VP2 + 20. // VP2 also points to PP1 and VP2 != VP1

Drawbacks?

- Fragmentation
- TLB pressure