Files

- Files are stored on disk
- OS provide read/write interfaces for reading/writing to a file
  - why?
- Every reading/writing to file involves a system call

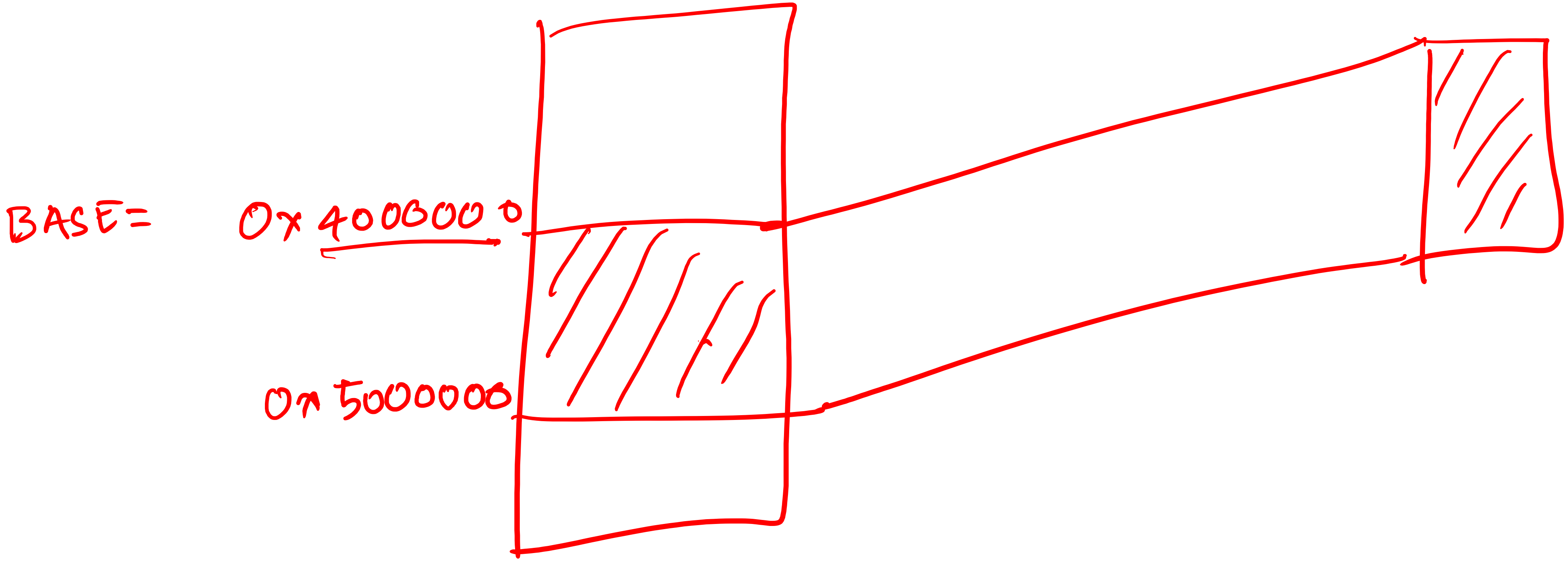
Memory mapped files

- On open system call, the OS checks the permissions and map the entire file in the virtual address space of the application

- The application can then directly read/write to file through reading/writing to memory
  - no system call

- When the application closes the memory mapped file, the OS writes all the memory mapped pages to their original location on disk

Memory mapped files



Page fault

Precise exceptions

- Hardware rollbacks all the changes made by a partially executed instruction which causes an exception

- xadd %eax, (%ecx)
  - exchange the values of source and destination operand and load the sum of these two values in the destination operand

```
t1 = %eax
%eax = *%ecx
*%ecx = t1
*ecx += %eax
```

```
t1 = %eax
%eax = *%ecx
*%ecx = t1
*%ecx += %eax
```

Page fault

- Page fault exception is raised by the hardware when a virtual address is dereferenced without sufficient privilege or no physical address is mapped corresponding to the virtual address

- Page fault is useful for copy on write optimization
  - The OS can make a copy, reinstate the write privilege, and resume the application

- The hardware restarts the execution of the partially executed instruction after returning from the exception handler

Copy on write

→ xadd %eax, (%ecx)

%cr2 = virtual address

kalloc()

copy

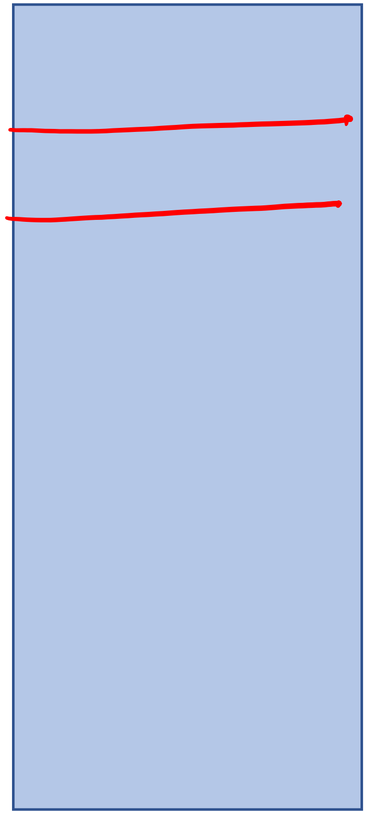
map_page with write permission in child

enable write access in parent

iret

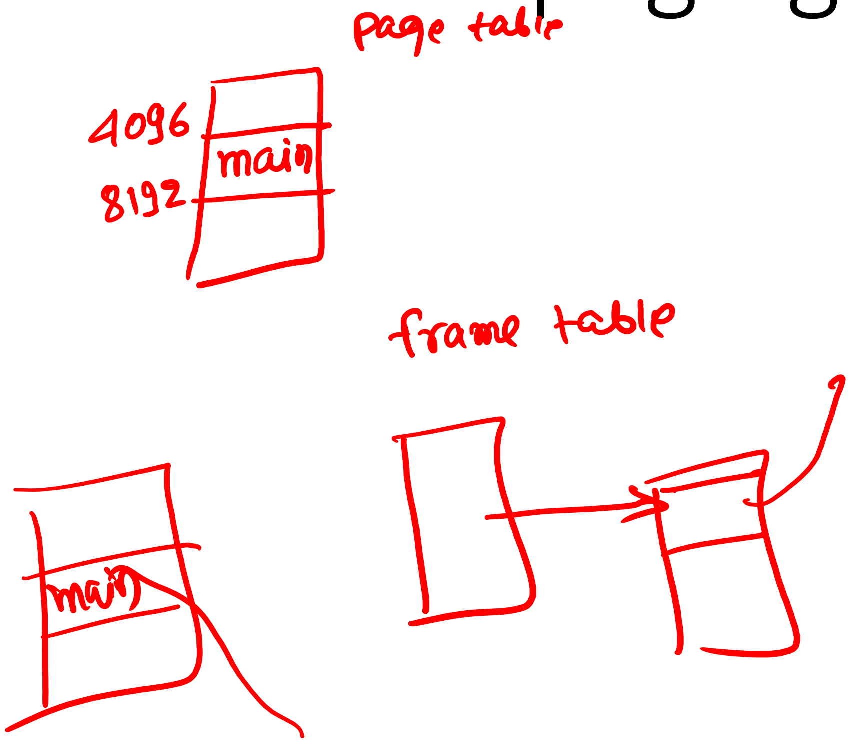
How does precise exception help in copy on write?

Demand paging



- Lazily loads program text and data in the memory
- Memory acts as a cache between the program on disk and processor
- How??

Demand paging

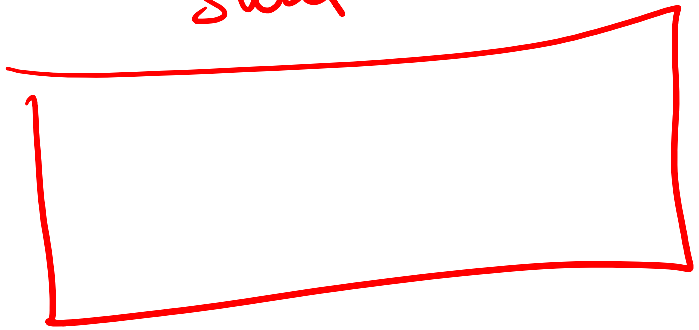


struct ~~frame~~ page_info
{
cat.exe
offset: 10000
}

- Maintain another data structure corresponding to every page table (say frame table)
- Corresponding to every virtual page store the file descriptor and offset in the frame table
- On a page fault, allocate a physical page, copy data from disk to memory, and map the page in the page table

What if no physical page is available to map in the page table?

swap area



Swapping

- Some area of the disk is reserved for saving the contents of a physical page temporarily during the execution of a program

- This space is called swap space

- Swap space is needed for recycling of physical pages during low memory scenario

Swapping

EIP   mov %eax, (%ecx)

rep movs

- Example:

VA : 0 – 409600 bytes : 100 virtual pages (VP)

PA : 0 – 4096 bytes : 1 physical page (PP)

User program accesses VP1

Page fault

OS maps VP1 – PP1

Swapping

- Example: 100 VP, 1 PP

User program accesses VP1

Page fault

OS maps VP1 – PP1

User program accesses VP2

page fault

Save PP1 to swap space

OS maps VP2 – PP1

```
arr1 [4096];
arr2 [4096];
arr1 [100] = 4000;
arr2 [100] = 3000;

for (i=0; i<10; i++)
{
    *(VP1);
    *(VP2);
}
```

Swapping

- Example: 100 VP, 1 PP

User program accesses VP1

PF : OS maps VP1 – PP1

User program accesses VP2

PF: Save PP1 to swap space (S1)

OS maps VP2 – PP1

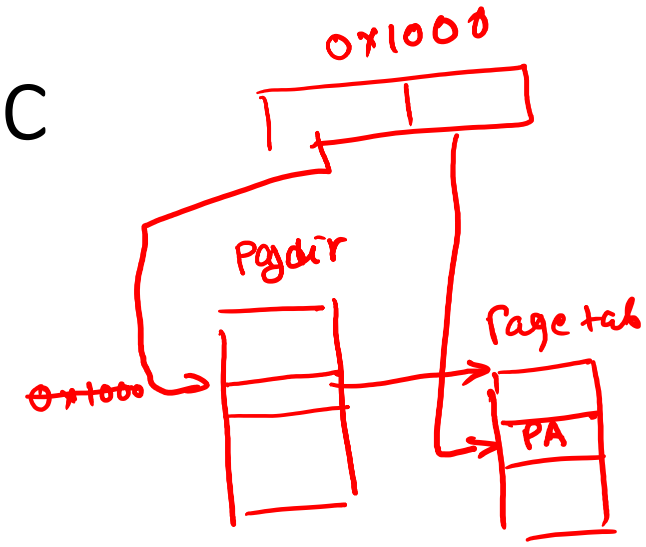
User program accesses VP1

PF: Save PP1 to swap space (S2)

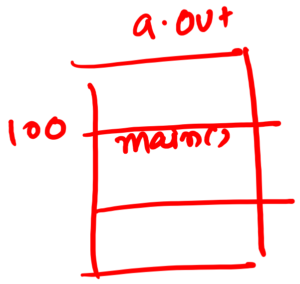
Load PP1 from swap space (S1)

OS maps VP1 – PP1

Exec



ELF
↓
load from a.out
at offset 100
at virtual address
0x1000
size = 4096

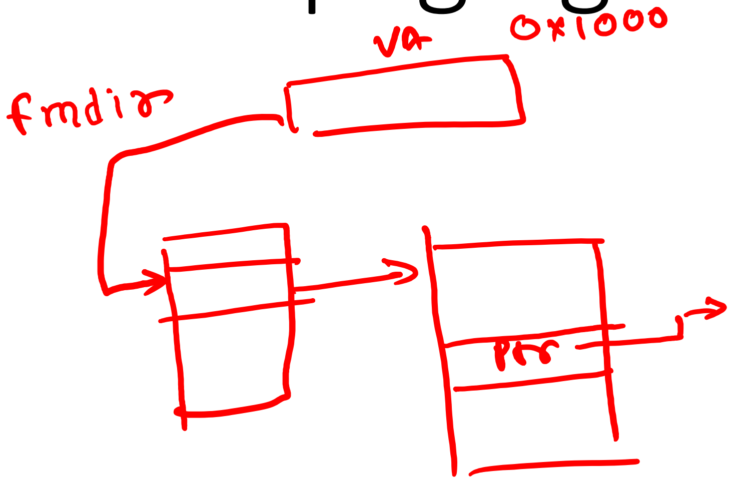


char* ptr = kalloc();

fd = open ("a.out");
seek (fd, 100, 0);
read (fd, ptr, 4096);

map-page (pgdir, 0x1000, V2P(ptr));

Demand paging



exec: from
load offset 100
~~from~~ from a.out
at virtual address
0x1000
size = 4096;

```
struct pageinfo
{
    int fd;
    int offset;   = 100
    int flag;
    unsigned sector_id;
};
```

fd = open(a.out);

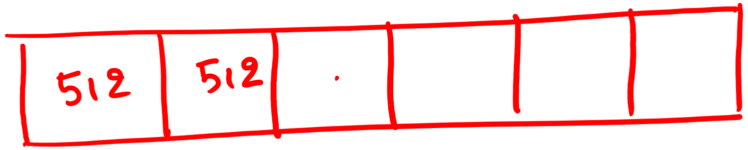
0x1000

fd

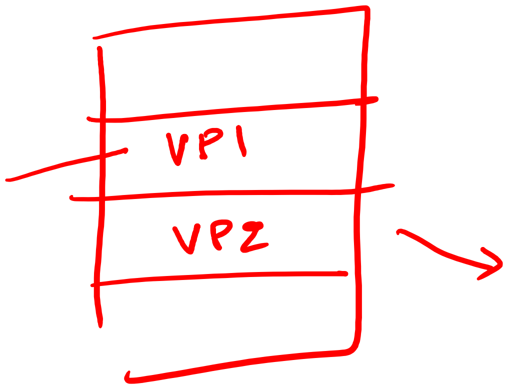
offset

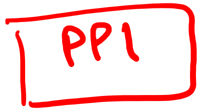
```
ptr = kalloc();
seek(fd, offset);
read(fd, ptr, 4096);
map_page(pgdir, 0x1000, V2P(ptr));
```

```
if (flag & SWAPPED)
{  ptr = kalloc();
   read_swap_page(ptr, sector_id);
   map_pag
   free_swap_space(sector_id);
}
```



Demand paging





swap(VP1);

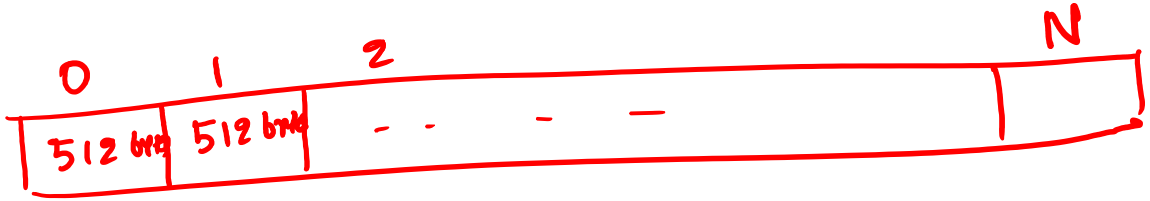
On a page fault

- If a page table entry is not present
  - The frame table contains the offset of the executable file from which the data should be loaded
  - The frame table contains the offset of the memory mapped file from which the data should be loaded
  - The frame table contains the address of the swapped block on disk
  - Otherwise, it must be unused stack and heap pages

Disk latency

- Disk interface allows programmers to read/write a block of data (512 bytes)

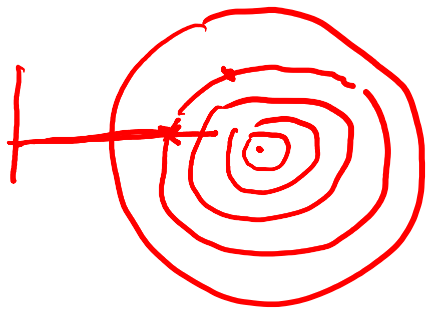
- Disk latency depends on seek time and the rotational latency



Disk latency

1 ms - 100 ms

1 ns - 100 ns



How does page size affect swapping?

How does page size affect swapping?

- Need to copy at least page size of data to the disk
  - Disk device interface allows you to read/write an entire block of data
  - the block size is 512 bytes
  - reading contiguous blocks from the disk is fast
  - the number of contiguous blocks depends on the spatial and temporal locality

Temporal locality

```
int sum = 0;
int i;

for (i = 0; i < 10000; i++) {
    sum += i;
}
```

- Temporal locality states that the data which is currently getting accessed is very likely to be accessed in the future
  - e.g., “sum” and “i” are following temporal locality

Spatial locality

- Spatial locality states that the nearby locations of the recently accessed data locations are very likely to be accessed in the future
  - Here “arr” accesses follow the spatial locality principle

```
int arr[1024];
int i, sum = 0;
…
for (i = 0; i < 1024; i++) {
  sum += arr[i];
}
```

How does page size affect swapping?

- Need to copy at least page size of data to the disk
  - Disk device interface allows you to read/write an entire block of data
  - the block size is 512 bytes
  - reading contiguous blocks from the disk is fast
  - the number of contiguous blocks depends on the spatial and temporal locality

- Hardware designers thought that 4096 bytes (8 disk blocks) is a good number that preserves the spatial locality in common workloads

What if spatial locality is not preserved?

- Hot and cold data

What if spatial locality is not preserved?

- A lot of cold data live in the memory that can be used for hot data
- A good throughput is achievable, if hot data live in memory most of the time

Average latency

- Say the disk latency is 10 ms and memory latency is 10 ns
- 90% of the time the data are present in memory
- 10% of time data live on the disk
- Avg. latency
  - $0.9 * 10\,ns + 0.1 * 10\,ms \approx 1\,ms$ (10^5 times slower than main memory)

Throughput

- A good throughput is only achievable if 99.9% of the time data are present in memory

- In other words, a good throughput is only achievable if all the hot data are present in memory

- Page table size matters for the placement of hot and cold data

Page replacement

- A good demand paging algorithm should minimize the number of page faults

- If the memory is not adequate then we need to swap some physical pages

- A page replacement algorithm decides which page is going to be swapped

Random

- swap a random physical page to the disk
  - pros: very easy to implement
  - cons: may replace a hot page

FIFO

- replace the oldest (in terms of time) page
  - add physical pages to a FIFO queue when they are brought in memory

- reference string

VP

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

15

7	7	7	2	2	2	4	4							
	0	0	0	3	3	3	2							
		1	1	1	0	0	0							

FIFO

- replace the oldest (in terms of time) page
    - add physical pages to a FIFO queue when they are brought in memory

- reference string

1, 2, 3, 4, 1, 2, 5, 1, 1, 2, 3, 4, 5

1	1	1	4	4	4	5	5	5
	2	2	2	1	1	1	3	3
		3	3	3	2	2	2	4

FIFO

- replace the oldest (in terms of time) page
  - add physical pages to a FIFO queue when they are brought in memory

- reference string

1, 2, 3, 4, 1, 2, 5, 1, 1, 2, 3, 4, 5

1	1	1	1	5	5	5	5	4	4
	2	2	2	2	1	1	1	1	5
		3	3	3	3	2	2	2	2
			4	4	4	4	3	3	3

Belady’s anomaly

Optimal page replacement

- Replace a page that will not be used for a longest period of time

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

7	7	7	2	2	2	2	2	7
	0	0	0	0	4	0	0	0
		1	1	3	3	3	1	1

LRU replacement

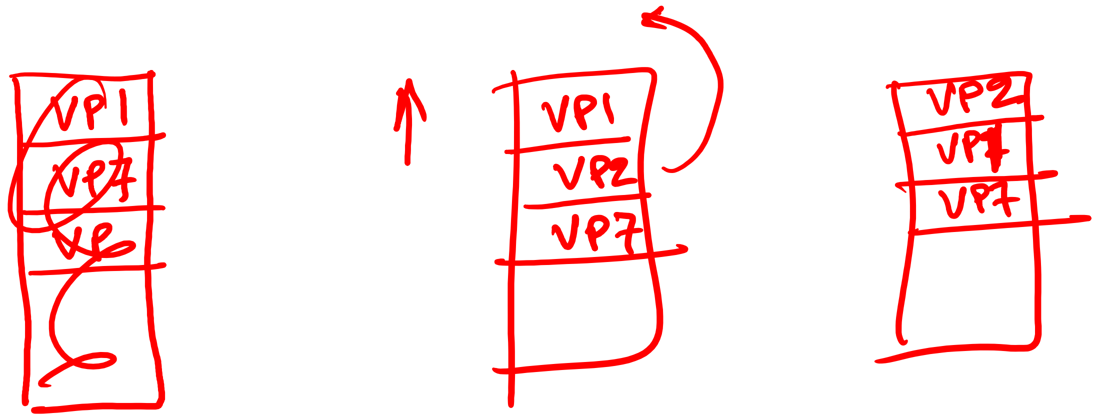
- Replace a page that will not be used for a longest period of time

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

7	7	7	2	2	4	4	4	0	1	1	1
	0	0	0	0	0	0	3	3	3	0	0
		1	1	3	3	2	2	2	2	2	7

LRU replacement

- LRU can be implemented using a timestamp
  - Update the timestamp of a page to the current timestamp on every reference
- Can be implemented using stack
  - Put the newly referenced page to the top of the stack
  - If the page already resides in the stack, remove from the stack and put on the top



LRU replacement