TA. YeouGyu Jeong
(81887821@snu.ac.kr)
System Software &
Architecture Lab.
Seoul National University

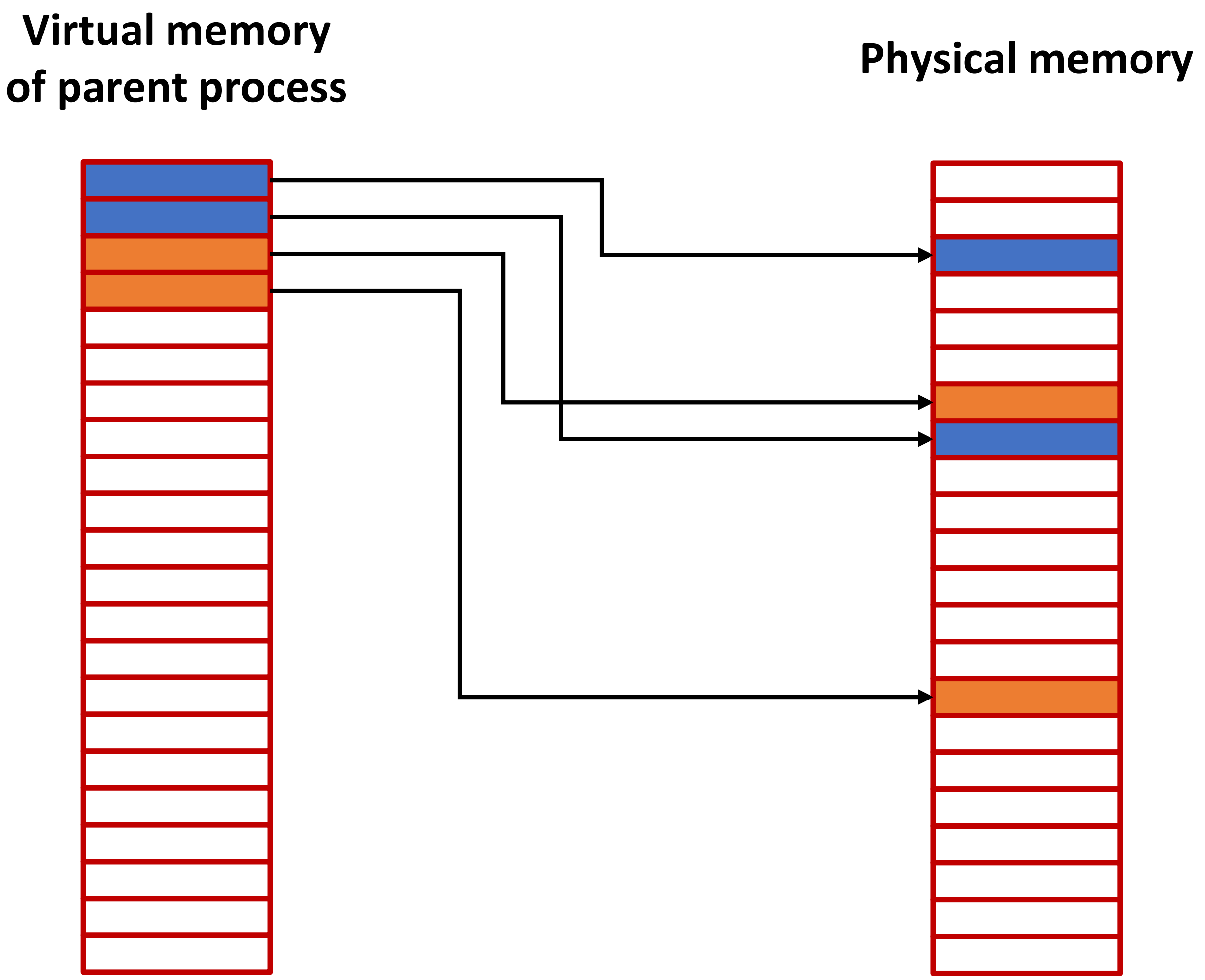
Spring 2020

4190.307: Operating Systems Lab. 4

Project #5 – Memory sharing

- In this project, you have to
  - Share code segment across fork
  - Implement copy-on-write on data, stack, and heap segment
  - Write design report
- Due date is May 24(Sunday)

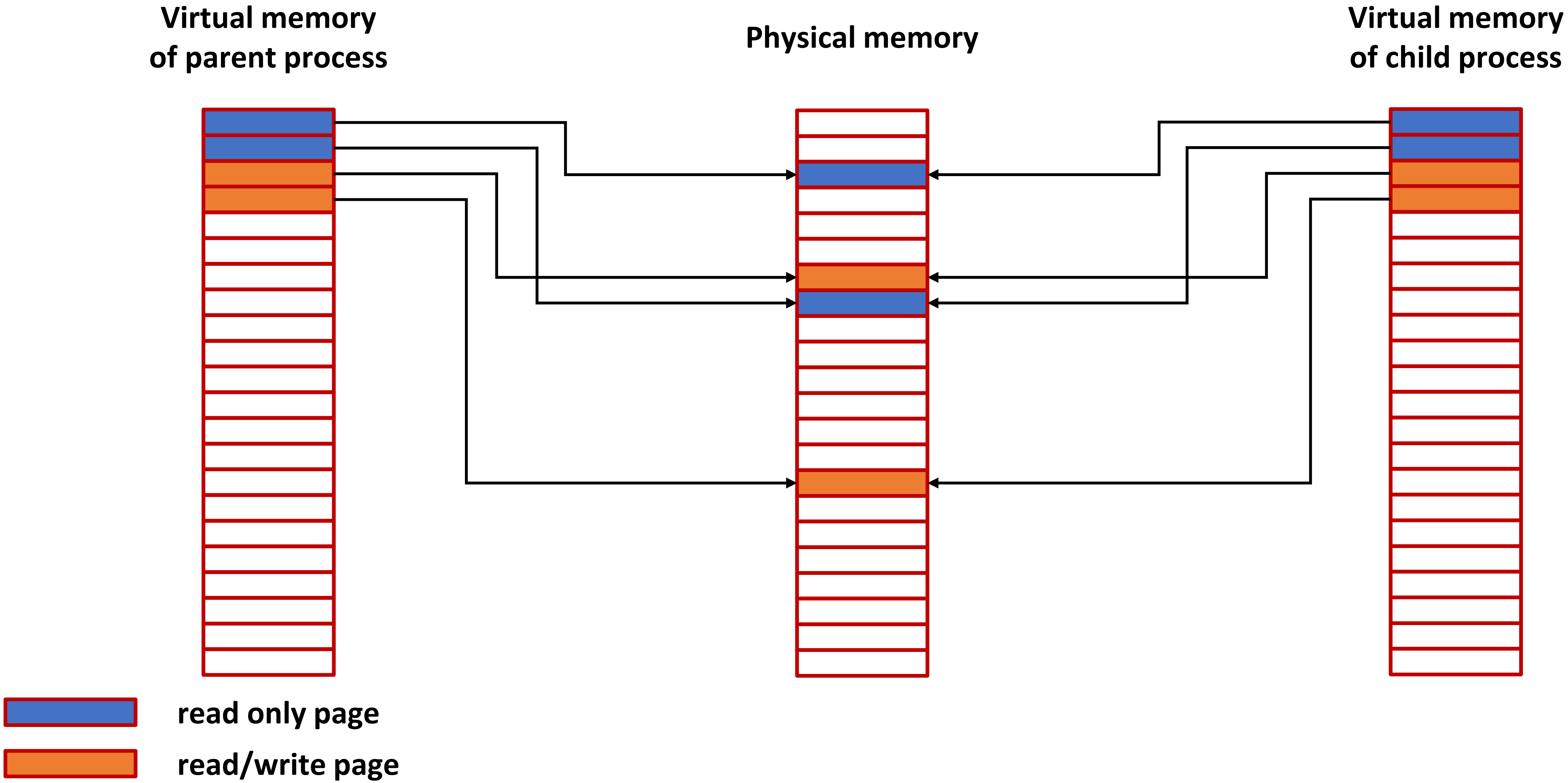
Memory sharing and copy-on-write



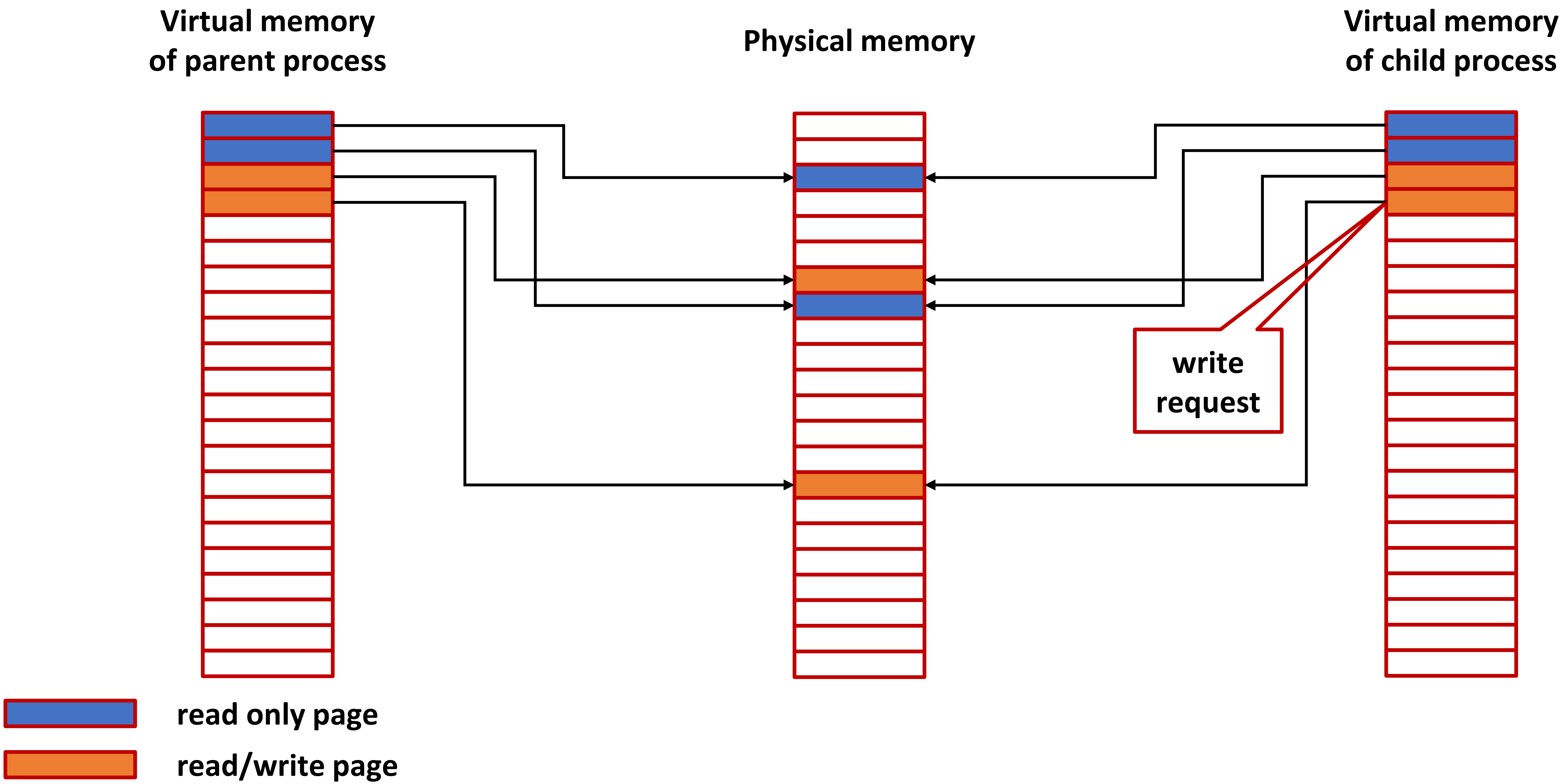
read only page

read/write page

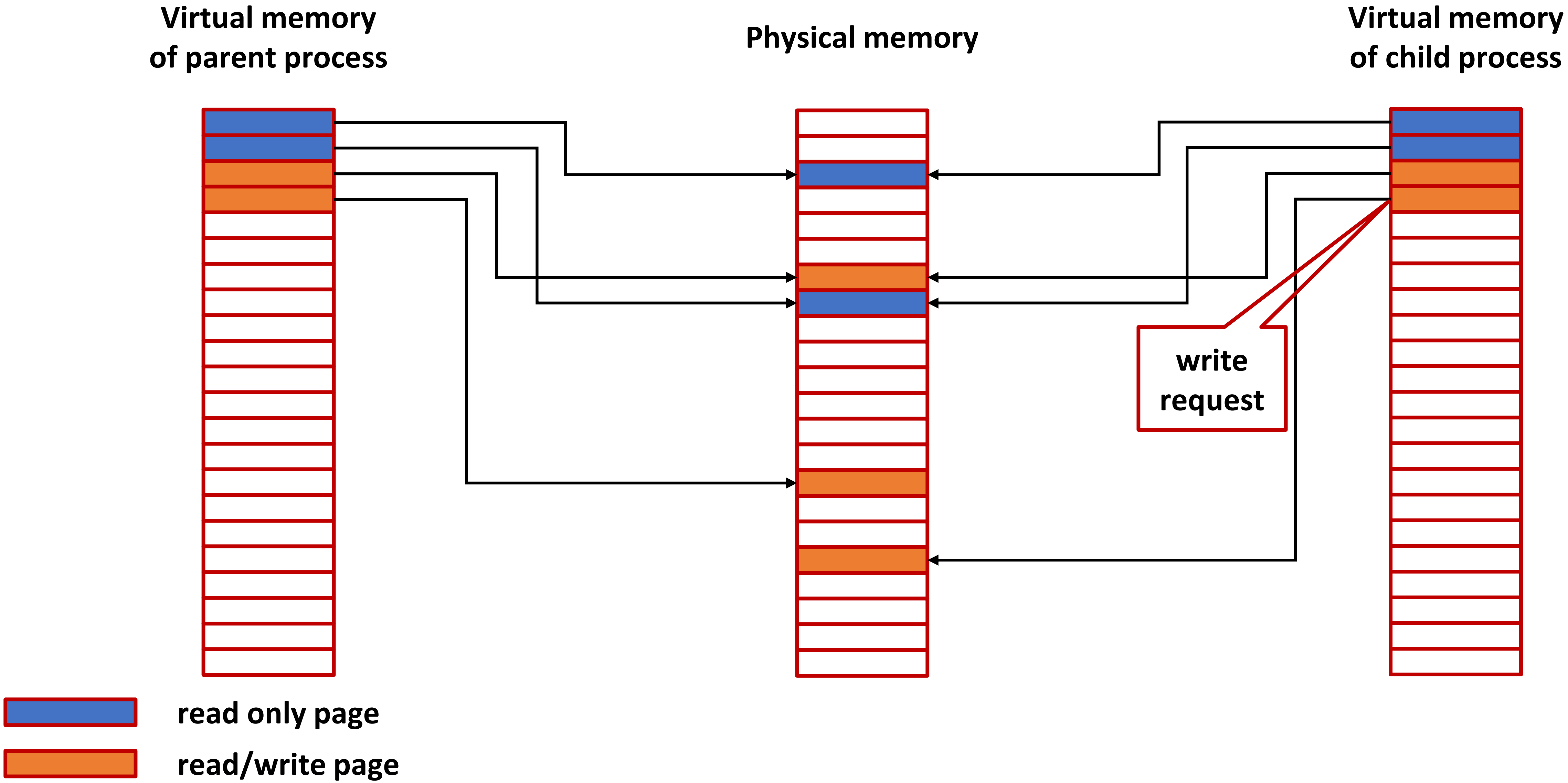
Memory sharing and copy-on-write



Memory sharing and copy-on-write



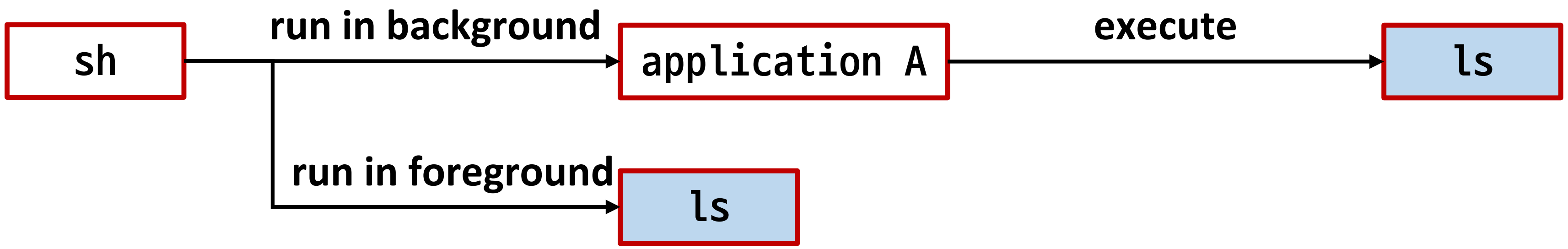
Memory sharing and copy-on-write



How to make code sharing work?

- On fork, simply map code page of child process to the same physical page as parent process

- Should "every" process of the same program share the same code page?
  - e.g.) Should 2 `ls` here share the same code page?



  - You don't have to
    This requires implementation of memory-mapped file which is not done in xv6
  - Only make parent and child share the same code page

How to make copy-on-write work?

- On fork,
  - make the page read-only
  - map the physical page to both parent and child
- When the process writes to the page,
  - write fails and exception is raised because the page is read-only
  - The exception handler should copy the page and remap virtual address to copied page

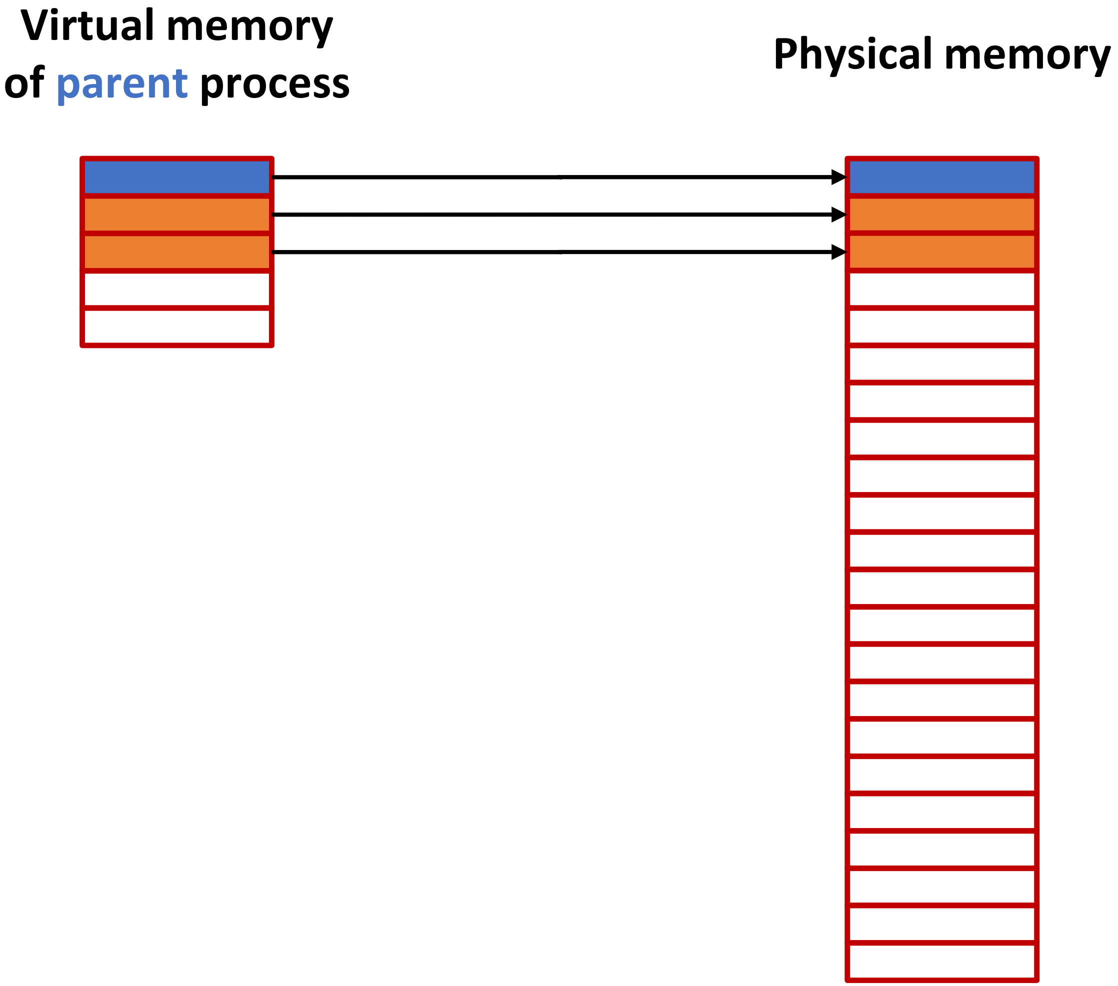
Memory sharing example

```
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {  // child
      wait();
      exec("/bin/ls");
    }
  } else {  // parent
    wait(); exit(0);
  }
}
```

- Suppose that xv6 runs code on left
- Parent forks, waits, and exits
- Child modifies global variable, forks, waits, and exec
  - Grand child modifies global variable and exits

Memory sharing example

```c
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {  // child
      wait();
      exec(“/bin/ls”);
    }
  } else {  // parent
    wait(); exit(0);
  }
}
```

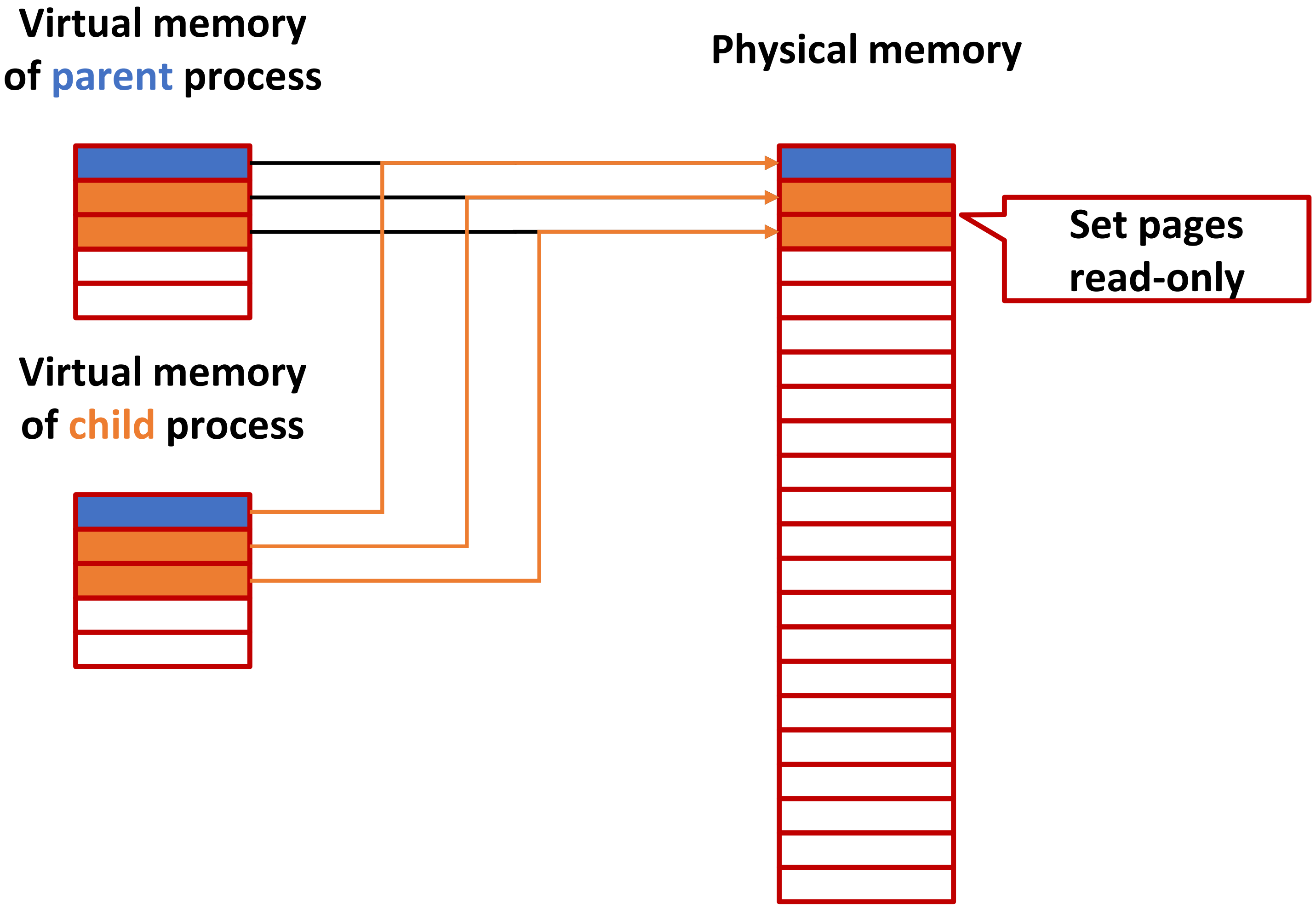


read only page

read/write page

Memory sharing example

```
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {   // child
      wait();
      exec(“/bin/ls”);
    }
  } else {   // parent
    wait(); exit(0);
  }
}
```



read only page

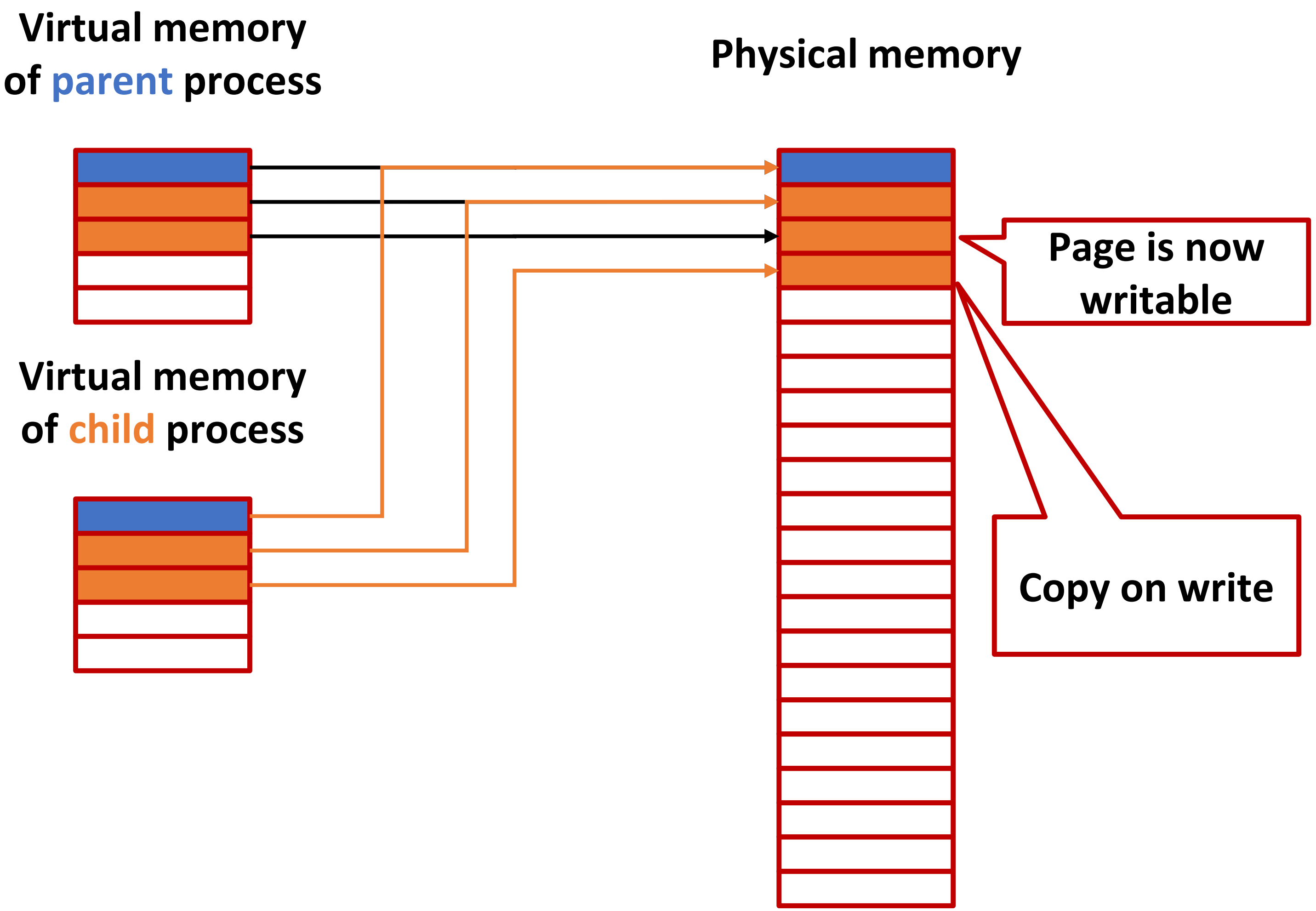
read/write page

Memory sharing example

```
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {  // child
      wait();
      exec("/bin/ls");
    }
  } else {  // parent
    wait(); exit(0);
  }
}
```

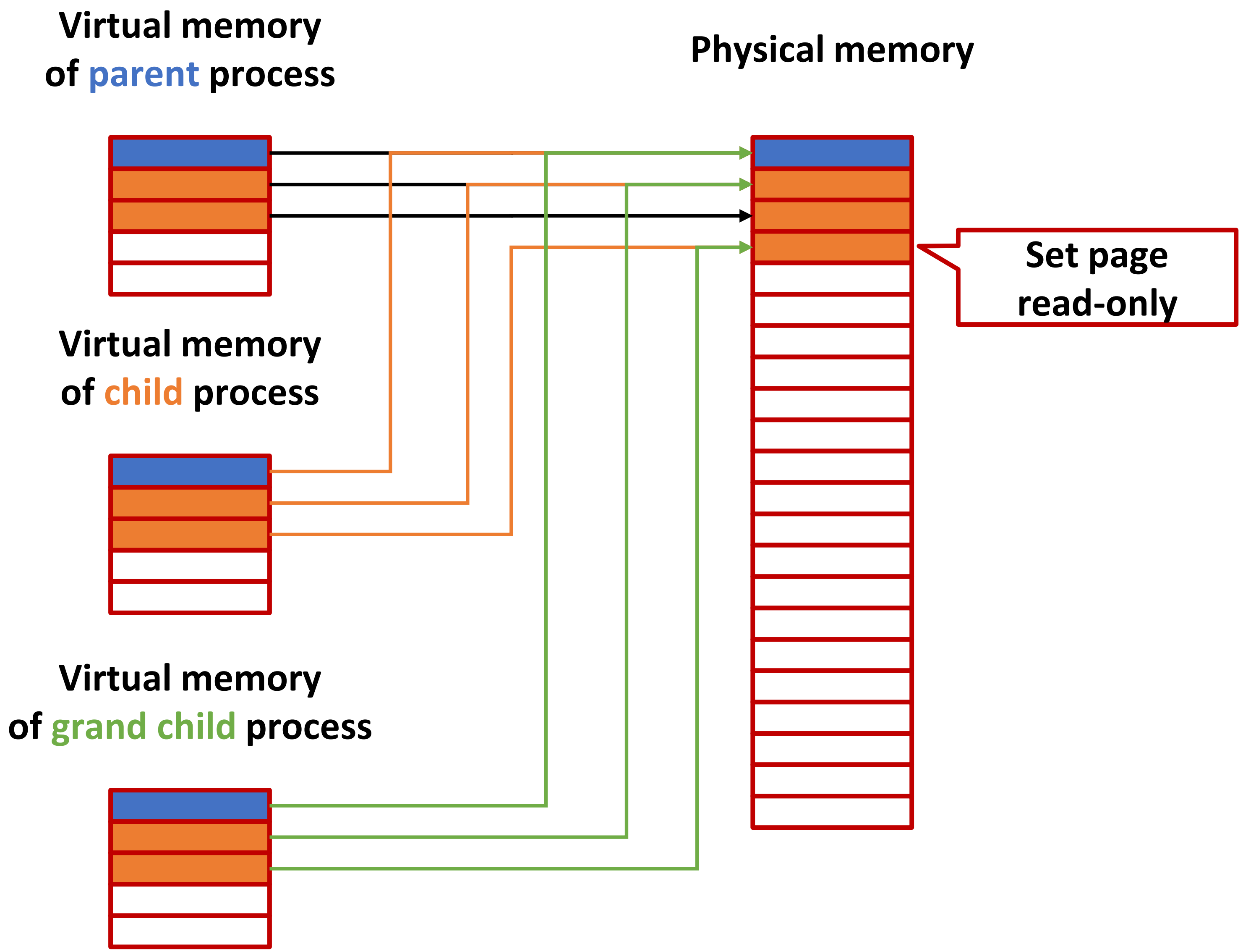
read only page

read/write page



Memory sharing example

```
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {  // child
      wait();
      exec("/bin/ls");
    }
  } else {  // parent
    wait(); exit(0);
  }
}
```

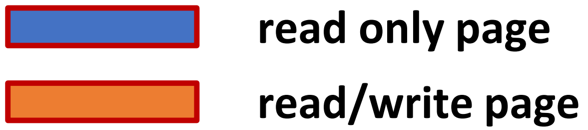


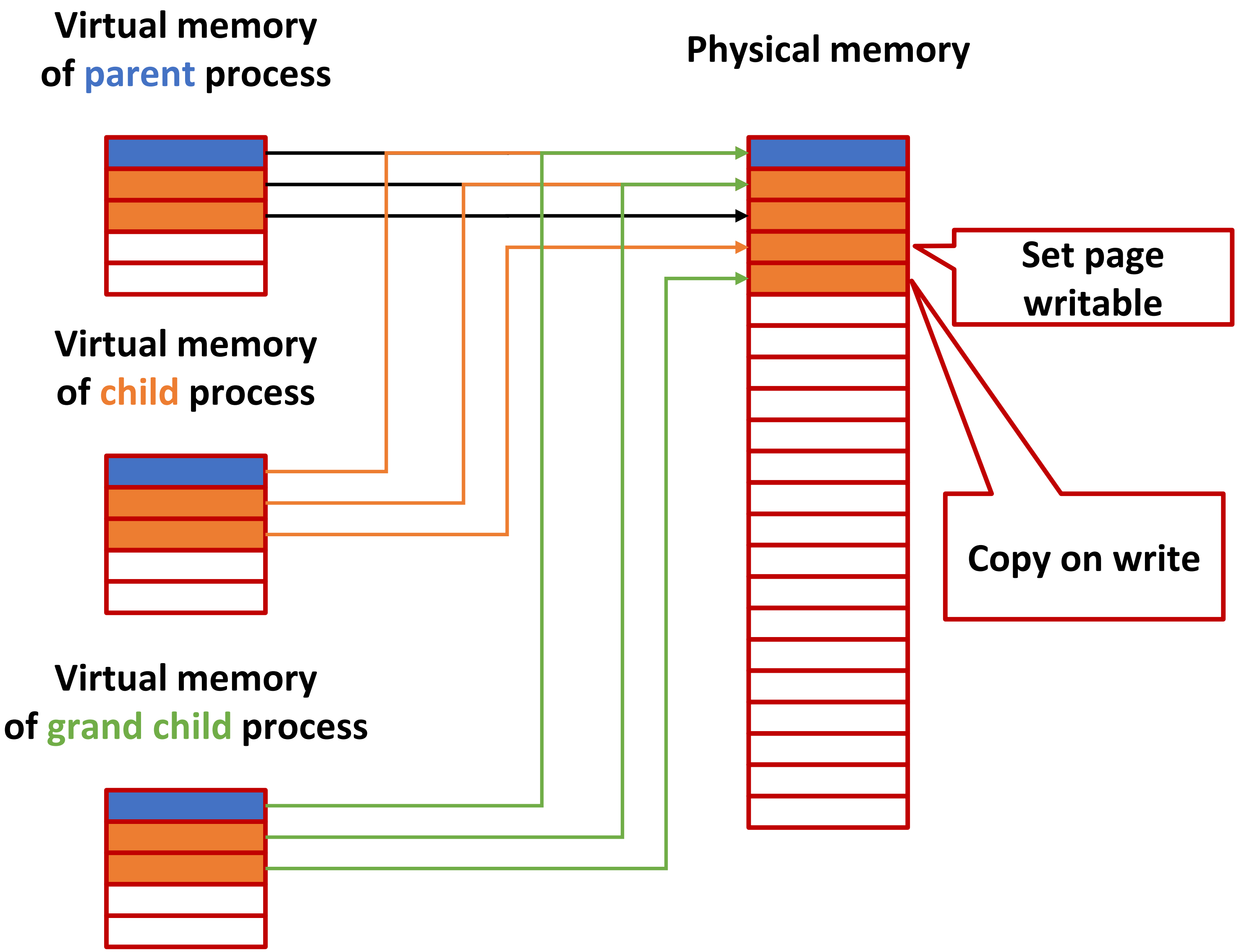
read only page

read/write page

Memory sharing example

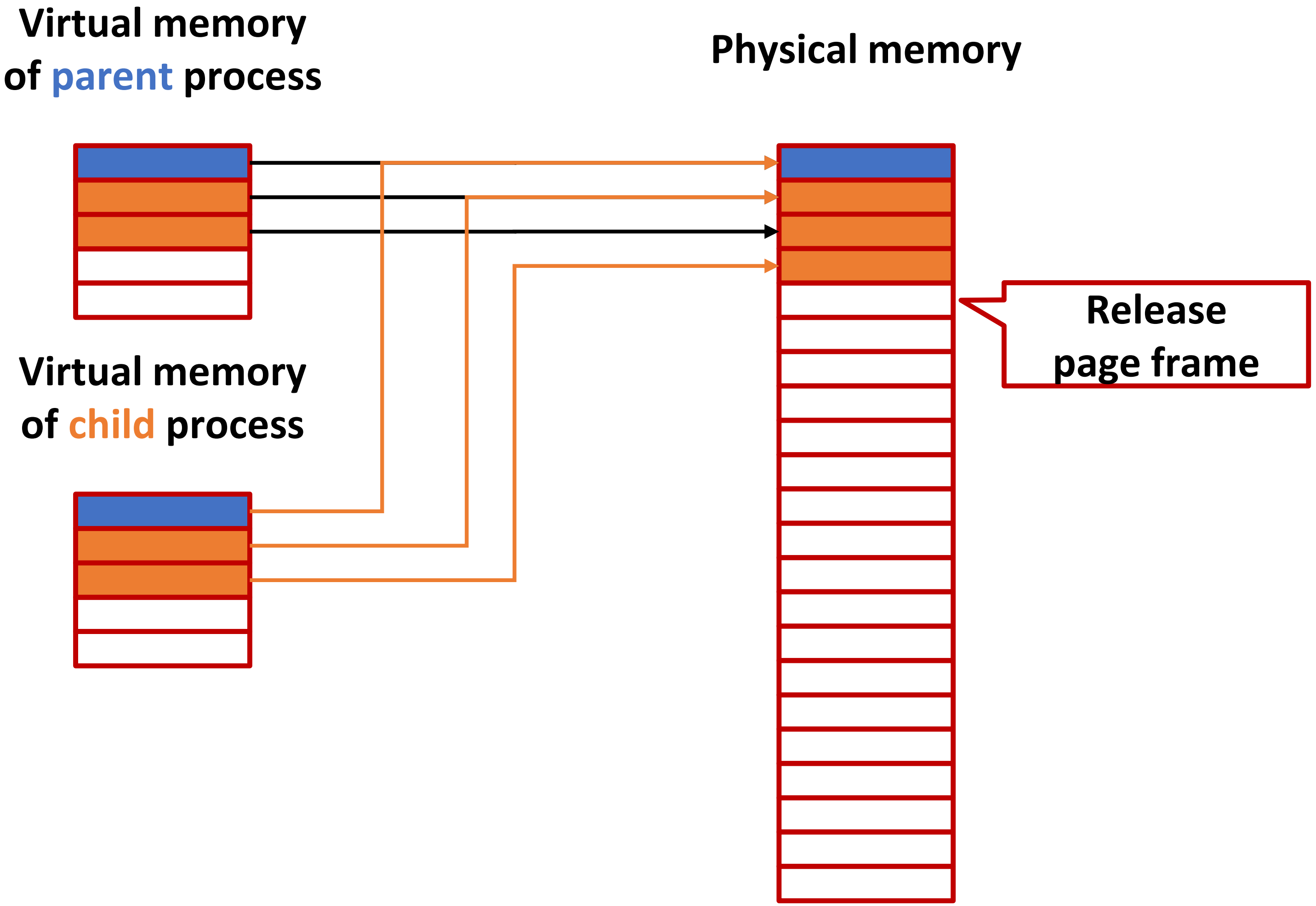
```
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {   // child
      wait();
      exec("/bin/ls");
    }
  } else {  // parent
    wait(); exit(0);
  }
}
```

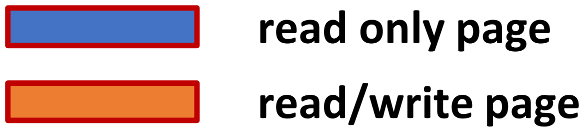




Memory sharing example

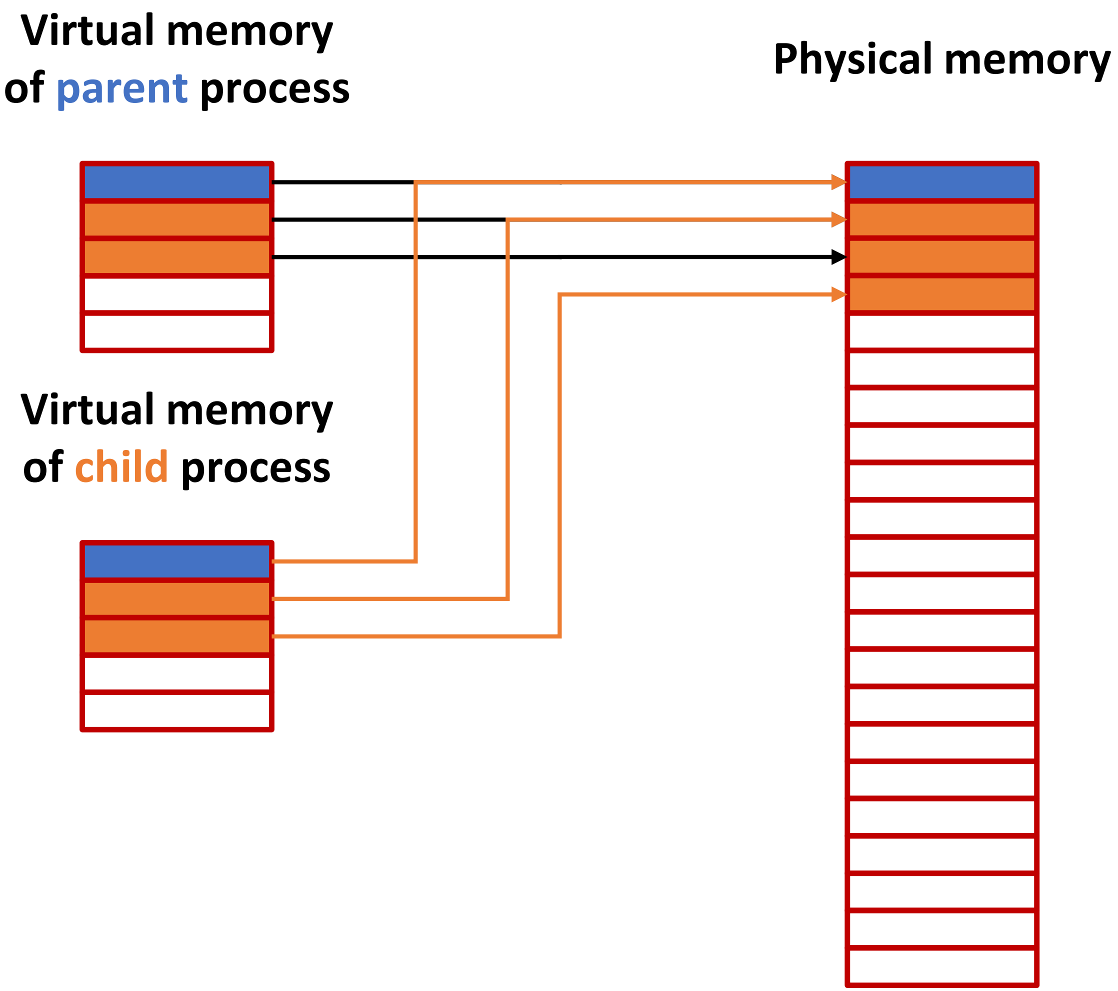
```
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {  // child
      wait();
      exec("/bin/ls");
    }
  } else {  // parent
    wait(); exit(0);
  }
}
```





Memory sharing example

```
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {  // child
      wait();
      exec(“/bin/ls”);
    }
  } else {  // parent
    wait(); exit(0);
  }
}
```

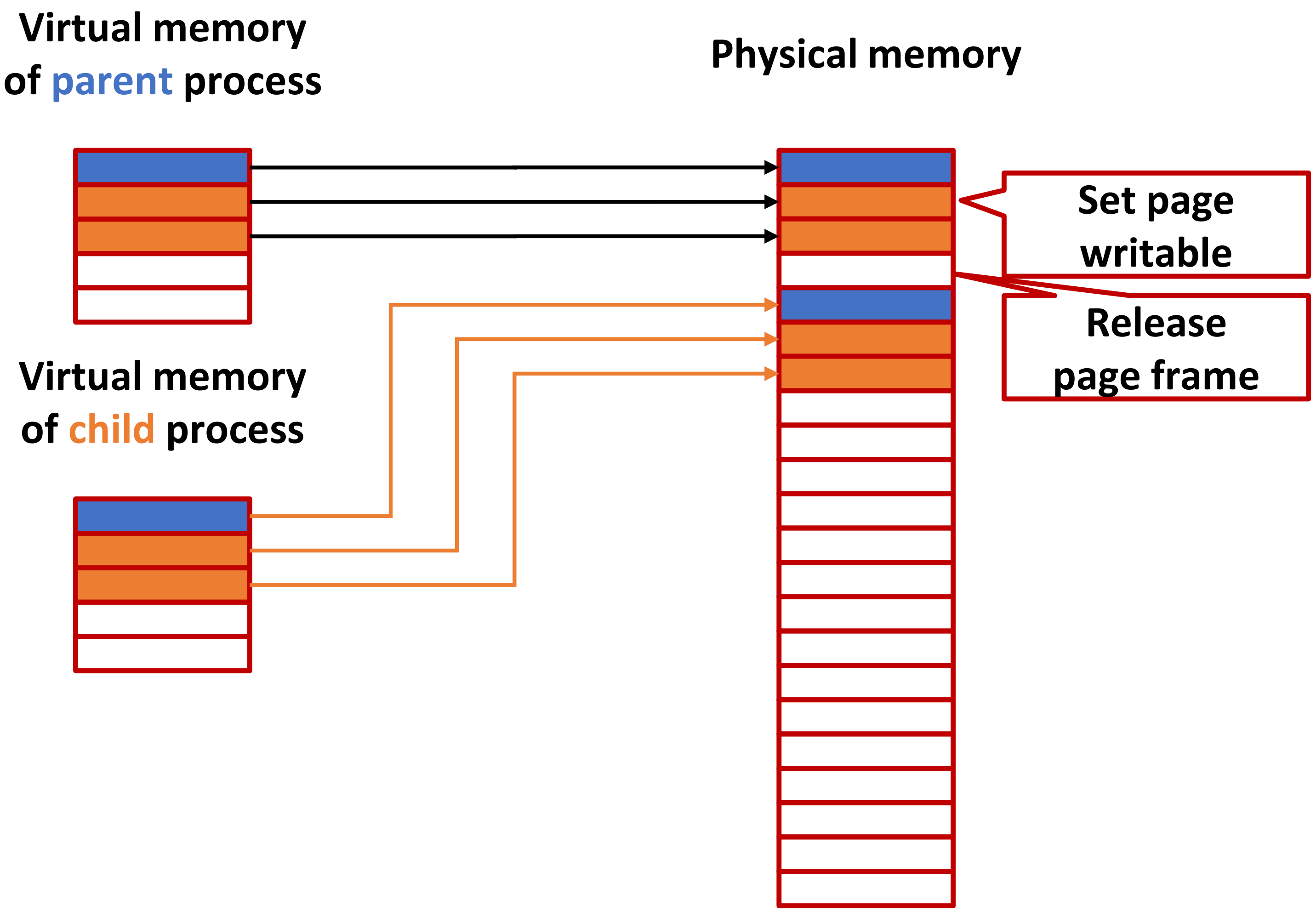


read only page

read/write page

Memory sharing example

```
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {  // child
      wait();
      exec(“/bin/ls”);
    }
  } else {  // parent
    wait(); exit(0);
  }
}
```

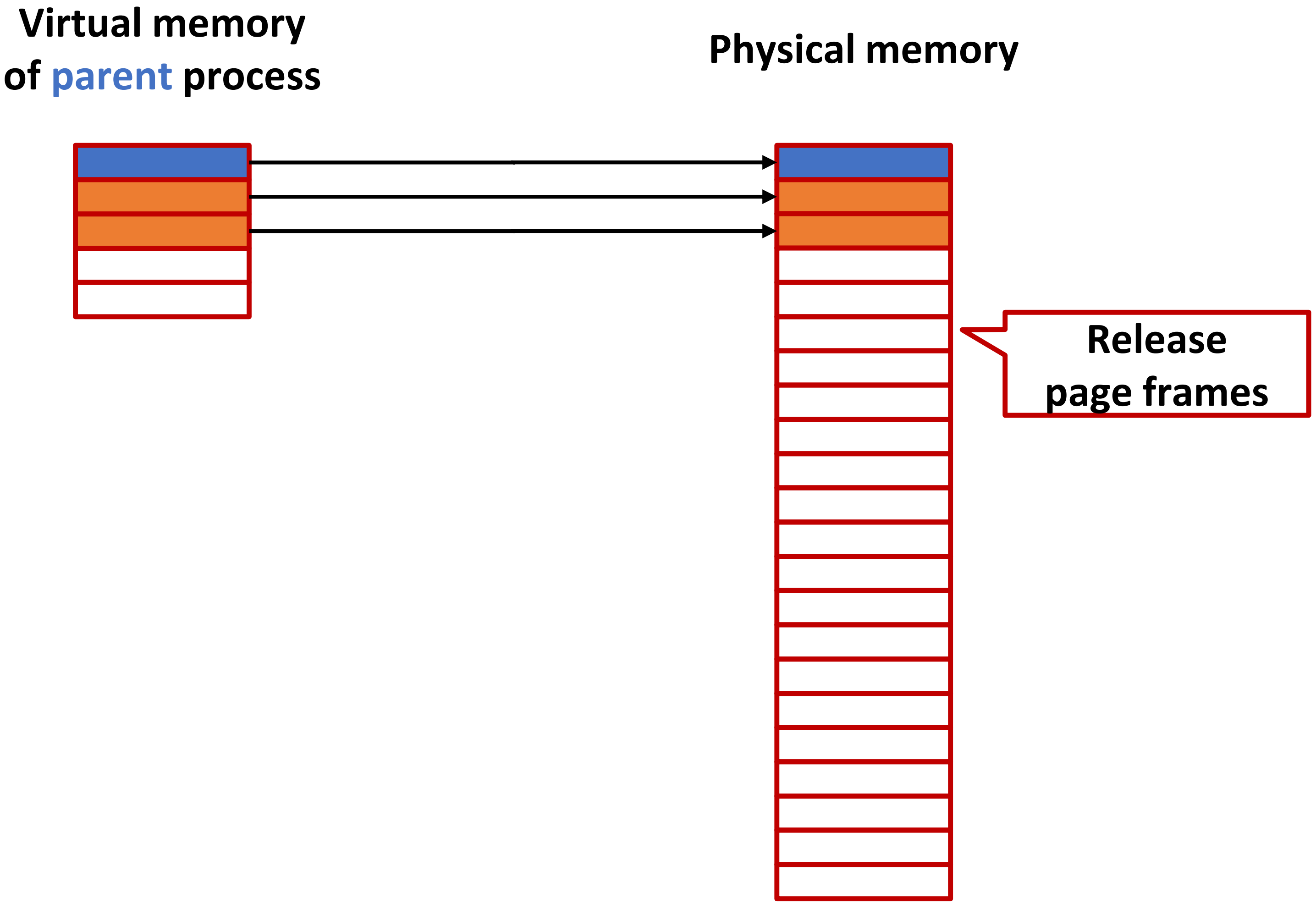


read only page

read/write page

Memory sharing example

```
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {  // child
      wait();
      exec("/bin/ls");
    }
  } else {  // parent
    wait(); exit(0);
  }
}
```



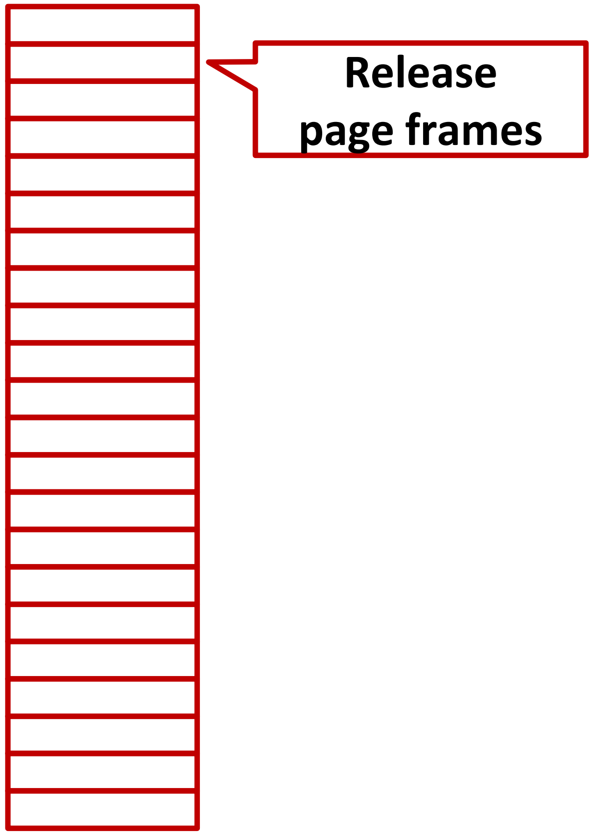
read only page

read/write page

Memory sharing example

```
int global = 10;

int main(void) {
  if (fork() == 0) {
    global = 5;
    if (fork() == 0) {
      // grand child
      global = 3;
      exit(0);
    } else {  // child
      wait();
      exec("/bin/ls");
    }
  } else {  // parent
    wait(); exit(0);
  }
}
```

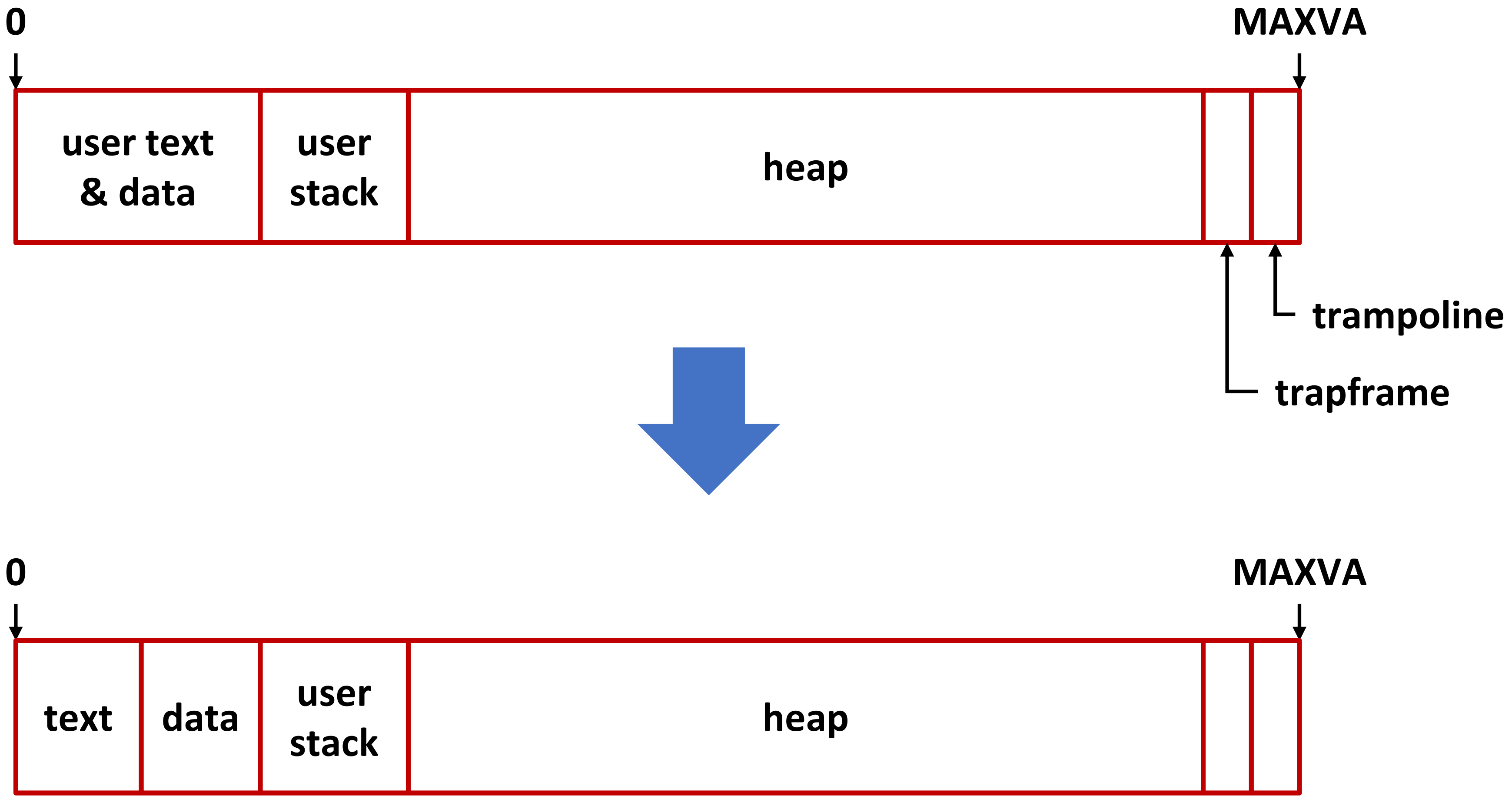
Physical memory



read only page

read/write page

Recap – Virtual address space layout



Initializing process address space

- Address space of a process is initialized by exec function
- To separate .text and .data to different pages, you need to give --no-omagic link option
- And make xv6 to load sections to virtual memory properly

- The above is already done in skeleton code
- What you have to do here is setting .text section read-only
  - You have to use `flags` in `struct proghdr` and permission flags in elf.h

Initializing process address space

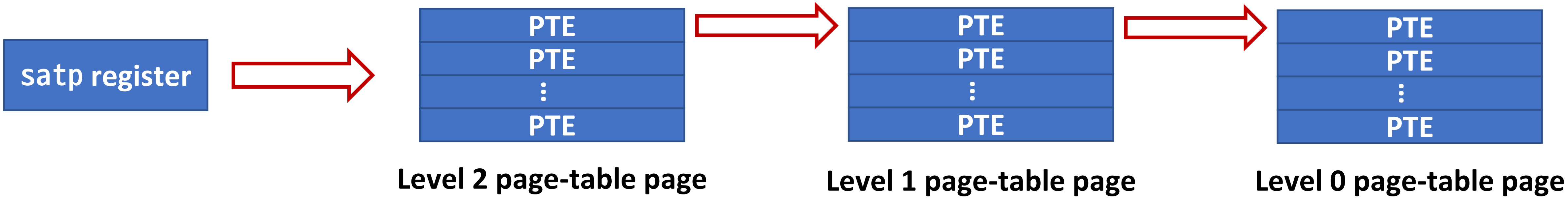
- In exec, xv6 loads all loadable segments from ELF binary

```
$ readelf -l _sh

Elf file type is EXEC (Executable file)
Entry point 0xa60
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000001000 0x0000000000000000 0x0000000000000000
                 0x00000000000013f1 0x00000000000013f1  R E    0x1000
  LOAD           0x00000000000023f8 0x00000000000023f8 0x00000000000023f8
                 0x000000000000000e 0x0000000000000090  RW     0x1000
 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata
   01     .sdata .sbss .bss
```

Virtual address system of RISC-V

- 64-bit RISC-V CPU supports 39 or 48-bit virtual address system
- xv6 uses 39-bit address system called Sv39
- In Sv39,
  - Page size is 4KiB
  - 3-level page-table is used, but any level can be leaf entry(super page)
    - If level 2 entry is a leaf, it points 1GiB sized super page
    - If level 1 entry is a leaf, it points 2MiB sized super page
    - If level 0 entry is a leaf, it points 4KiB sized page
  - Page-table is aligned to page boundary
    - A page-table entry is 8 bytes
    - A page-table page has 512 entries

Paging in Sv39



Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

A page-table entry

Page-table entry bits

- `D:` Dirty bit, set on page write
- `A:` Accessed bit, set on page read/write/instruction fetch
- `G:` Global mapping, set for pages that exist in all address spaces
- `U:` If set, the page can be accessed from user mode
- `X:` If set, the page can be executed
- `W:` If set, the page can be written
- `R:` If set, the page can be read
- `V:` Validity bit, the entry is valid only if `V` bit is set

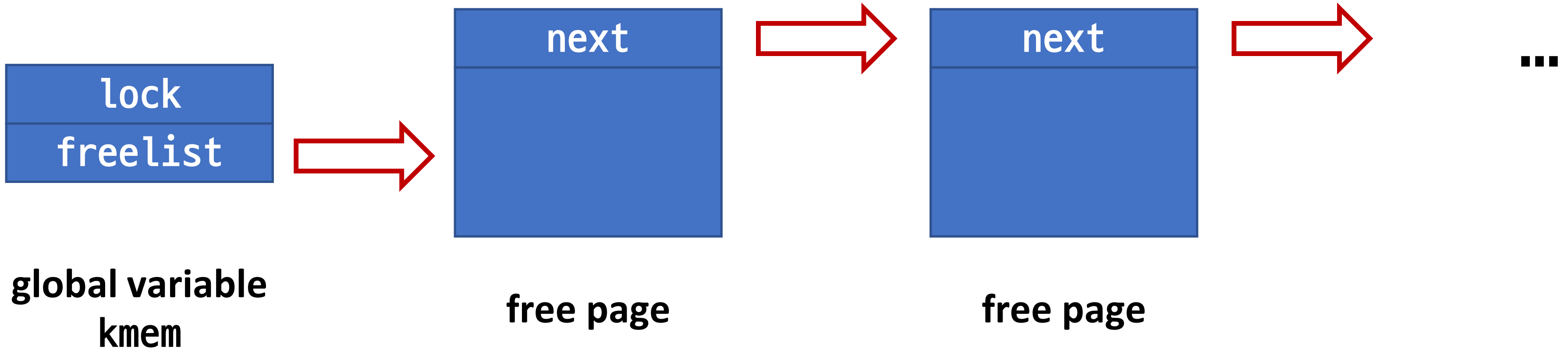
- If `X`, `W`, and `R` are all `0`, the entry is a pointer to next level

Handling page-table in xv6

- `int mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)`
  - Used to map physical pages to virtual address space
- `uint64 walkaddr(pagetable_t pagetable, uint64 va)`
  - Used to get physical address of virtual address

Managing physical memory in xv6

- xv6 uses a simple free list to manage physical pages
- To allocate a physical page, use `kalloc`
- To free the physical page, use `kfree`



Please note that…

- Copy-on-write should be performed only for the page in which the page fault occurs, not the whole memory segment
- You must terminate the program if it accesses invalid memory region, or writes to the code segment
- Make sure there is no memory leak

To verify your kernel

- We added free memory counter to the skeleton code
  - You can see how many pages are available by pressing Ctrl-P
  - Or by `getfreemem` system call

- `v2p` system call and `v2ptest` user space application
  - `v2p` system call gets virtual address and returns physical address
  - You can check if code segment is really shared, data page is copied on write, …

Design document

- Brief summary of modifications you have made
- How do you catch the page fault?
- How do you implement code segment sharing?
- How do you implement copy-on-write on data/stack/heap segment?
- When is a page frame released and how?
- Other things you have considered in your implementation

When you do your project,

- Please read the project description carefully
  - https://github.com/snu-csl/os-pa5
- You have to start the project from pa5 branch
- Please only modify Makefile, and files in kernel directory
  - Changes to other source will be ignored by grading script
- Please remove all the debugging outputs before you submit
- Keep `getfreemem` and `v2p` system call work for grading

You may want to see…

- defs.h
  - For function definitions
- kalloc.c
  - For physical page allocation
- vm.c
  - For virtual address and page-table management
- riscv.h
  - For PTE flags and page-table related macros
- trap.c
  - For exception handling
- exec.c, elf.h
  - For elf binary loading

Thank you!

- Any questions?
- Or feel free to ask us in KakaoTalk