

Operating Systems

Assignment 2

Daniel Gruss

2021-04-27



Presented Today:





Presented Today:

- **Mandatory:** Virtual Memory (Copy On Write, Swapping)



Presented Today:

- **Mandatory:** Virtual Memory (Copy On Write, Swapping)
- Shared Memory



Presented Today:

- **Mandatory:** Virtual Memory (Copy On Write, Swapping)
- Shared Memory
- Memory Mapped I/O



Presented Today:

- **Mandatory:** Virtual Memory (Copy On Write, Swapping)
- Shared Memory
- Memory Mapped I/O
- Dynamic Memory in the userspace



Presented Today:

- **Mandatory:** Virtual Memory (Copy On Write, Swapping)
- Shared Memory
- Memory Mapped I/O
- Dynamic Memory in the userspace



Presented Today:

- **Mandatory:** Virtual Memory (Copy On Write, Swapping)
- Shared Memory
- Memory Mapped I/O
- Dynamic Memory in the userspace

Other Topics:

- You can do basically anything OS related



Presented Today:

- **Mandatory:** Virtual Memory (Copy On Write, Swapping)
- Shared Memory
- Memory Mapped I/O
- Dynamic Memory in the userspace

Other Topics:

- You can do basically anything OS related
- Just ask your Tutor how many points it brings

Page Replacement







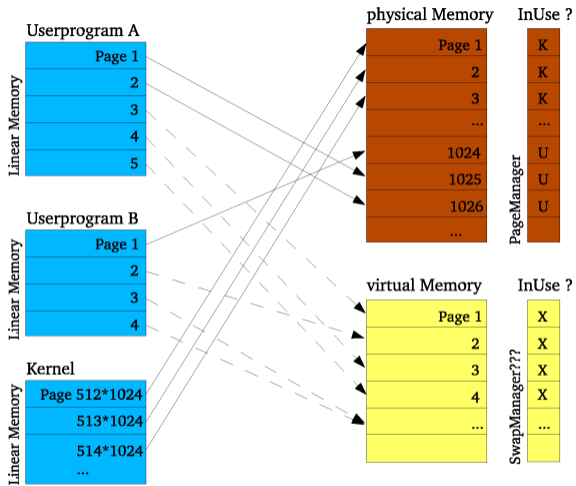
- Swap pages to the swap device (from RAM to HDD)

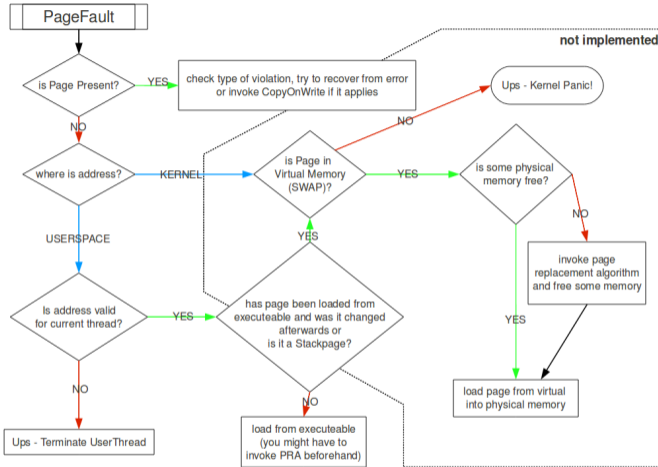


- Swap pages to the swap device (from RAM to HDD)
- Don't forget to **lock** shared resources!

What does the OS need to know?

- Where is the swap device located?
- Where to find free space within the swap device?
- Has a page been swapped out, or is it within RAM?
- Where has a page been swapped to (target address)?





- Virtual Memory is located at the third partition of the first HDD
 - (BD device number 3)
- Responsible Code: `arch_bd_*`

Example (Write to BD (Pseudocode))

```
size_t block = target block number;  
pointer page_data = pointer to source data;  
  
BDVirtualDevice* bd_device = BDManager::getInstance()->getDeviceByNumber(3);  
bd_device->writeData(block*bd_device->getBlockSize(), PAGE_SIZE, page_data);
```

Do not use `BDRequest` directly unless you asked a Tutor!

Using `BDRequest` directly is unsafe!

Which pages are swappable?

- User space pages (where does it make sense?)

- User space pages (where does it make sense?)
- Mark PTs/PDs/PDPTs as non-present and swapped out

- User space pages (where does it make sense?)
- Mark PTs/PDs/PDPTs as non-present and swapped out
- Kernel pages (has not been done before)

present == 0: entry invalid, all bits ignored

by MMU

→ pagefault on access

writable == 0: write protected

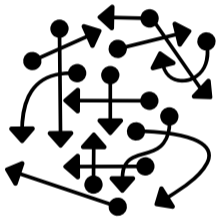
accessed, dirty == 1: has been

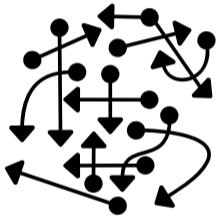
accessed/modified

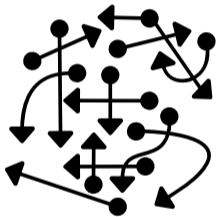
ignored_x : unused bits

page_ppn : physical page number

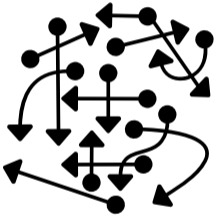
```
typedef struct
{
    uint64 present           :1;
    uint64 writable         :1;
    uint64 user_access      :1;
    uint64 write_through    :1;
    uint64 cache_disabled   :1;
    uint64 accessed         :1;
    uint64 dirty            :1;
    uint64 size             :1;
    uint64 global           :1;
    uint64 ignored_2        :3;
    uint64 page_ppn         :28;
    uint64 reserved_1       :12;
    uint64 ignored_1        :11;
    uint64 execution_disabled :1;
} PageTableEntry;
```

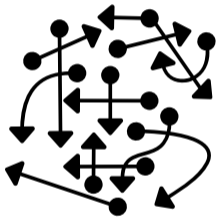




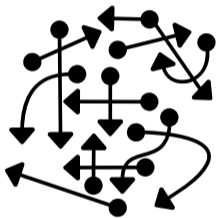
- Use tests which use big arrays
(e.g. `size_t array[BIG_NUMBER];`)



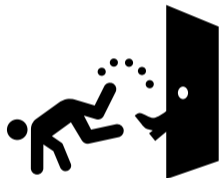
- Use tests which use big arrays
(e.g. `size_t array[BIG_NUMBER];`)
- Test **all** swapping-situations



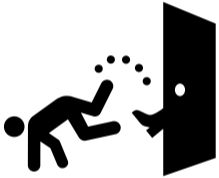
- Use tests which use big arrays
(e.g. `size_t array[BIG_NUMBER];`)
- Test **all** swapping-situations
- ..., without running into the limits of the kernel heap.



- Use tests which use big arrays
(e.g. `size_t array[BIG_NUMBER];`)
- Test **all** swapping-situations
- ..., without running into the limits of the kernel heap.
- Free memory can exhaust soon (even with a good PRA)!

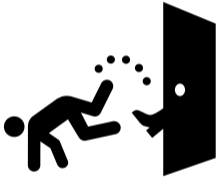


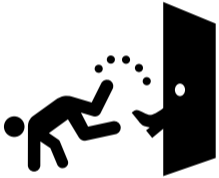
What does a PRA do?



What does a PRA do?

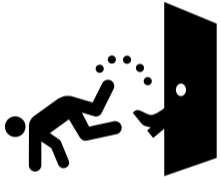
- Searches for pages that have not been used for a while





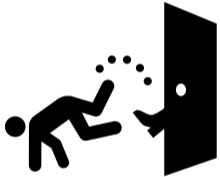
What does a PRA do?

- Searches for pages that have not been used for a while
- Runs if memory is needed or there is nothing to be done



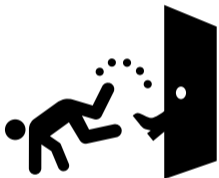
What does a PRA do?

- Searches for pages that have not been used for a while
- Runs if memory is needed or there is nothing to be done
- But not every time...



What does a PRA do?

- Searches for pages that have not been used for a while
- Runs if memory is needed or there is nothing to be done
- But not every time...

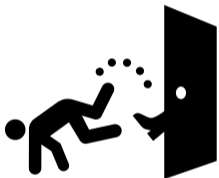


What does a PRA do?

- Searches for pages that have not been used for a while
- Runs if memory is needed or there is nothing to be done
- But not every time...

Which PRA?

- Recommended: Aging or WSRandom

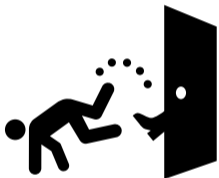


What does a PRA do?

- Searches for pages that have not been used for a while
- Runs if memory is needed or there is nothing to be done
- But not every time...

Which PRA?

- Recommended: Aging or WSRandom
- Create your own PRA (why is it better than other PRAs?)



What does a PRA do?

- Searches for pages that have not been used for a while
- Runs if memory is needed or there is nothing to be done
- But not every time...

Which PRA?

- Recommended: Aging or WSRandom
- Create your own PRA (why is it better than other PRAs?)
- Bonus Points: User can switch PRAs





Some PRAs need time information Where to get them from?



Some PRAs need time information Where to get them from?

- Ticks, TSC, RTC



Some PRAs need time information Where to get them from?

- Ticks, TSC, RTC
- Recycle parts of the `sleep`- or `clock`-implementation



Some PRAs need time information Where to get them from?

- Ticks, TSC, RTC
- Recycle parts of the `sleep`- or `clock`-implementation
- Derive the time from the tick sources



Some PRAs need time information Where to get them from?

- Ticks, TSC, RTC
- Recycle parts of the `sleep-` or `clock-`implementation
- Derive the time from the tick sources
- Hint: `InterruptUtils.cpp`

Inverted Page Table (IPT)







- Pages may be used by several processes



- Pages may be used by several processes
- Aka: Page table entries of different user spaces point to the same physical page

Where can pages be shared?

- RAM

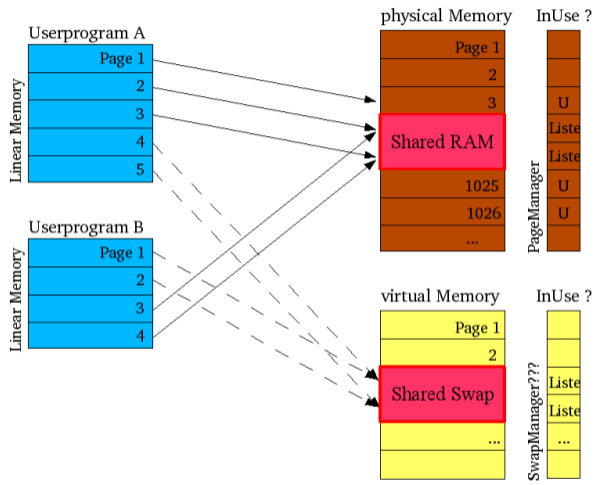
- RAM
- Swap

- RAM
- Swap
- Binary

- RAM
- Swap
- Binary

- RAM
- Swap
- Binary

And what's with copy-on-write?









Inverted Page Table connects a physical page/swapped page to all virtual page usages



Inverted Page Table connects a physical page/swapped page to all virtual page usages



Inverted Page Table connects a physical page/swapped page to all virtual page usages

Only one process uses the page:

- Swap out page + inform process



Inverted Page Table connects a physical page/swapped page to all virtual page usages

Only one process uses the page:

- Swap out page + inform process
- Which process owns the page?



Inverted Page Table connects a physical page/swapped page to all virtual page usages

Only one process uses the page:

- Swap out page + inform process
- Which process owns the page?



Inverted Page Table connects a physical page/swapped page to all virtual page usages

Only one process uses the page:

- Swap out page + inform process
- Which process owns the page?

Shared Pages (several processes use the same page):

- Swap out + inform **all** processes



Inverted Page Table connects a physical page/swapped page to all virtual page usages

Only one process uses the page:

- Swap out page + inform process
- Which process owns the page?

Shared Pages (several processes use the same page):

- Swap out + inform **all** processes
- Which processes own the page?



Inverted Page Table connects a physical page/swapped page to all virtual page usages

Only one process uses the page:

- Swap out page + inform process
- Which process owns the page?

Shared Pages (several processes use the same page):

- Swap out + inform **all** processes
- Which processes own the page?
- But what if a process terminates?

Copy On Write







- Usage of `fork()`:



- Usage of `fork()`:
 1. `fork()` clones a process (copy, copy, copy...)



- Usage of `fork()`:
 1. `fork()` clones a process (copy, copy, copy...)
 2. The child process often uses `exec(...)` after `fork()`



- Usage of `fork()`:
 1. `fork()` clones a process (copy, copy, copy...)
 2. The child process often uses `exec(...)` after `fork()`
 3. There has been much useless deep copying and deleting



- Usage of `fork()`:
 1. `fork()` clones a process (copy, copy, copy...)
 2. The child process often uses `exec(...)` after `fork()`
 3. There has been much useless deep copying and deleting
- Do we really have to copy all the stuff?



- Usage of `fork()`:
 1. `fork()` clones a process (copy, copy, copy...)
 2. The child process often uses `exec(...)` after `fork()`
 3. There has been much useless deep copying and deleting
- Do we really have to copy all the stuff?
 - Both processes use the same physical and swapped pages



- Usage of `fork()`:
 1. `fork()` clones a process (copy, copy, copy...)
 2. The child process often uses `exec(...)` after `fork()`
 3. There has been much useless deep copying and deleting
- Do we really have to copy all the stuff?
 - Both processes use the same physical and swapped pages
 - Two (or more) processes have the same pages in RAM/Swap Device



- Usage of `fork()`:
 1. `fork()` clones a process (copy, copy, copy...)
 2. The child process often uses `exec(...)` after `fork()`
 3. There has been much useless deep copying and deleting
- Do we really have to copy all the stuff?
 - Both processes use the same physical and swapped pages
 - Two (or more) processes have the same pages in RAM/Swap Device
 - Works as long as no one is writing onto them







- How do we realize that someone wants to write onto a page



- How do we realize that someone wants to write onto a page
 - Usually we can't



- How do we realize that someone wants to write onto a page
 - Usually we can't
 - The writeable-flag has to be zero



- How do we realize that someone wants to write onto a page
 - Usually we can't
 - The writeable-flag has to be zero
- Process tries to write onto a read-only page
→ PageFault



- How do we realize that someone wants to write onto a page
 - Usually we can't
 - The writeable-flag has to be zero
- Process tries to write onto a read-only page
→ PageFault
- What now?



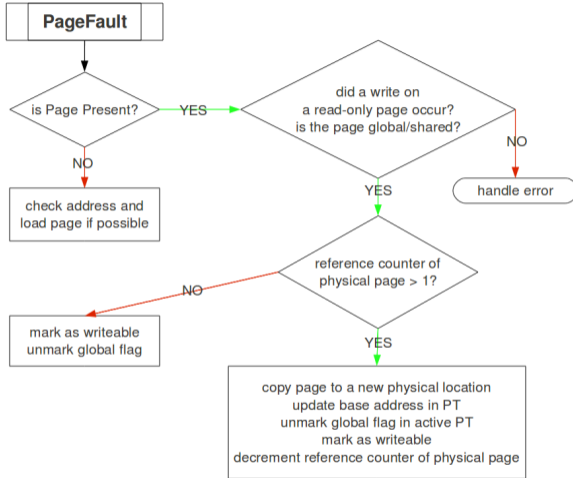
- How do we realize that someone wants to write onto a page
 - Usually we can't
 - The writeable-flag has to be zero
- Process tries to write onto a read-only page
→ PageFault
- What now?
 - → Is it a shared page?

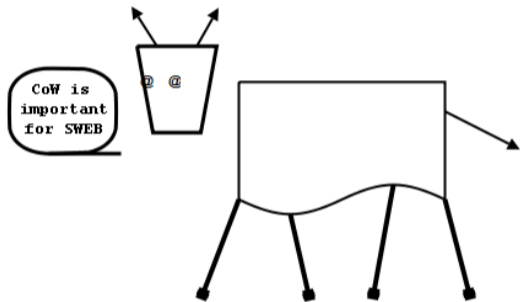


- How do we realize that someone wants to write onto a page
 - Usually we can't
 - The writeable-flag has to be zero
- Process tries to write onto a read-only page
→ PageFault
- What now?
 - → Is it a shared page?
 - → Copy page and link to the new one



- How do we realize that someone wants to write onto a page
 - Usually we can't
 - The writeable-flag has to be zero
- Process tries to write onto a read-only page
→ PageFault
- What now?
 - → Is it a shared page?
 - → Copy page and link to the new one
 - → If only one process is left → no shared pages!





What about the global flag?



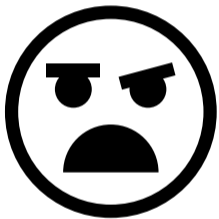




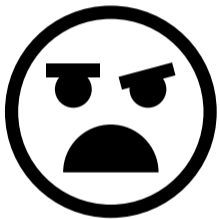
- Don't use it!



- Don't use it!
- "global" means "keep over next context switch"



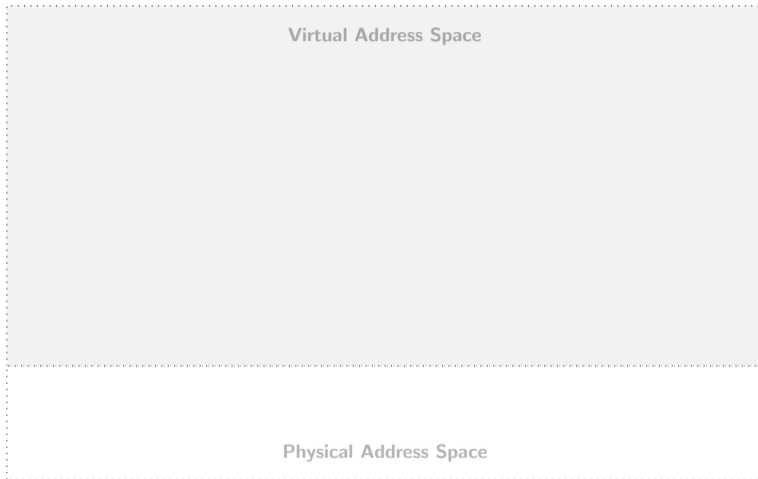
- Don't use it!
- “global” means “keep over next context switch”
- This is not what you want

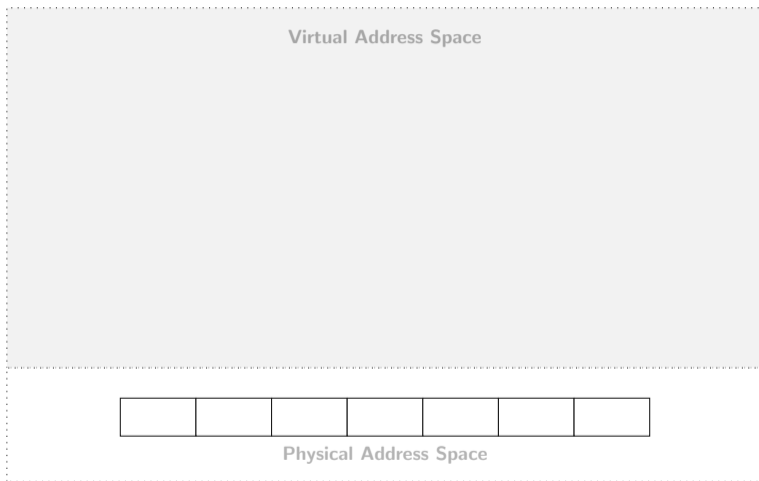


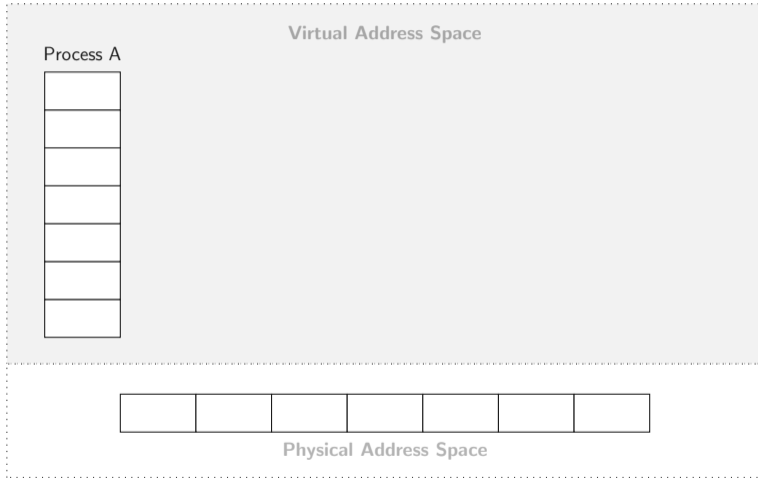
- Don't use it!
- “global” means “keep over next context switch”
- This is not what you want
- Will cause almost untraceable bugs!

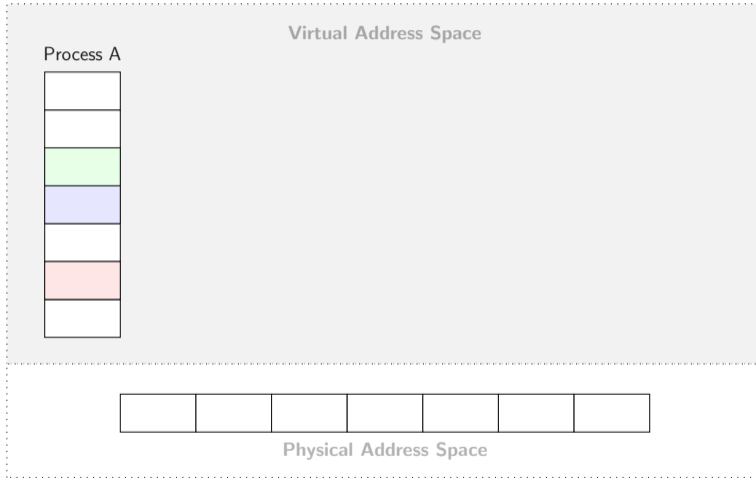


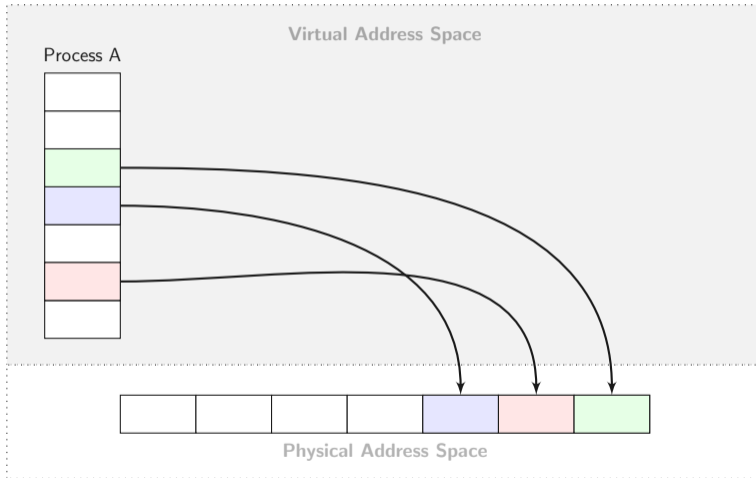
- Don't use it!
- “global” means “keep over next context switch”
- This is not what you want
- Will cause almost untraceable bugs!
- Use and rename an unused bit as “shared” flag instead

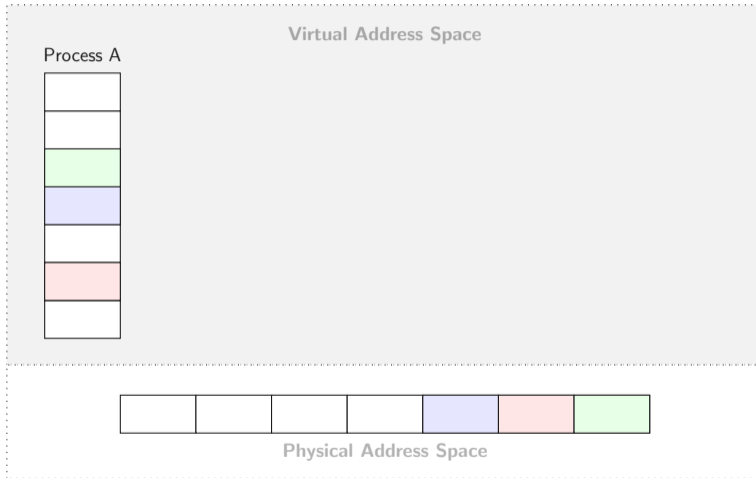


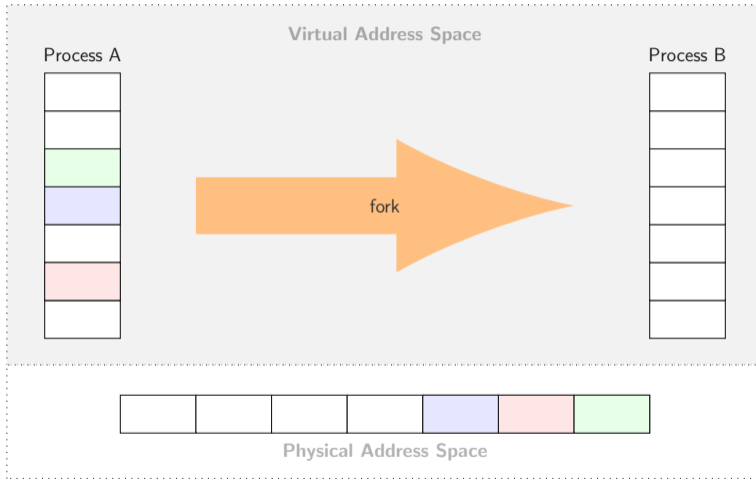


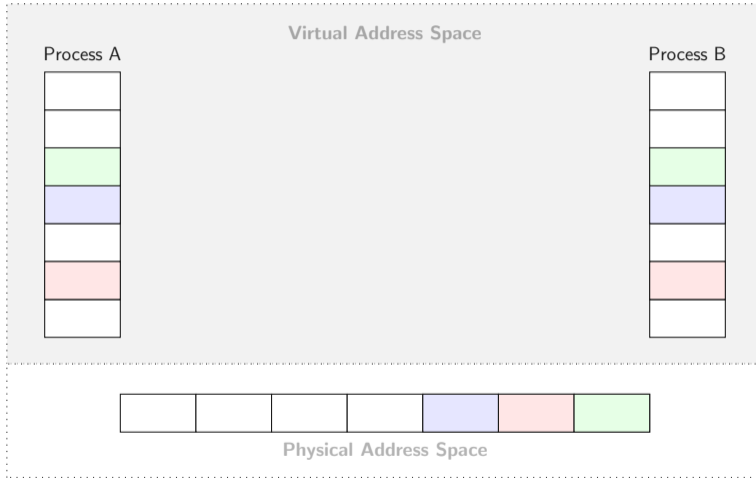


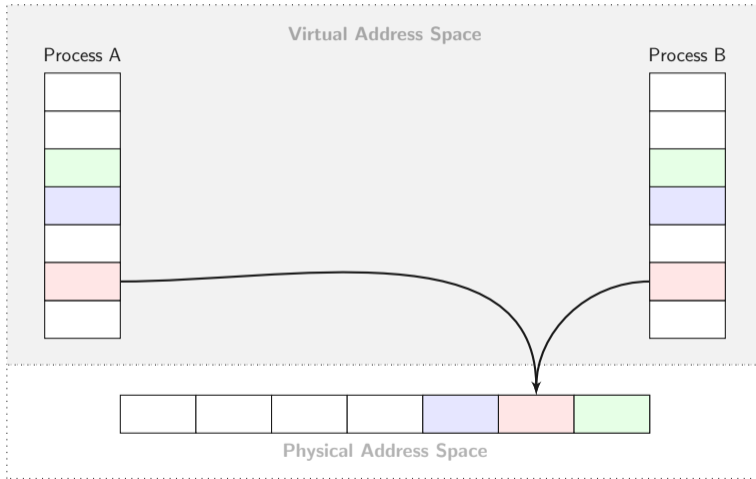


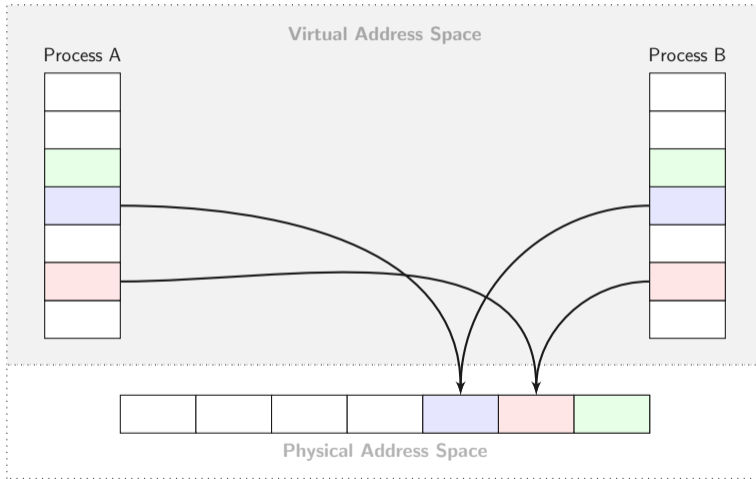


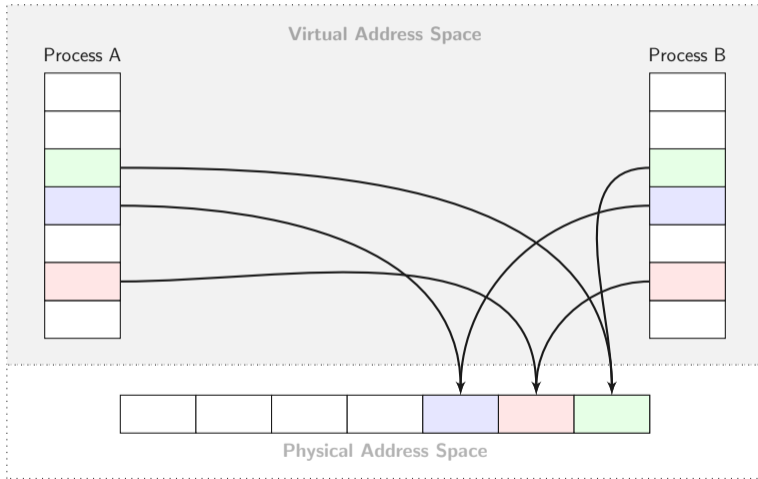


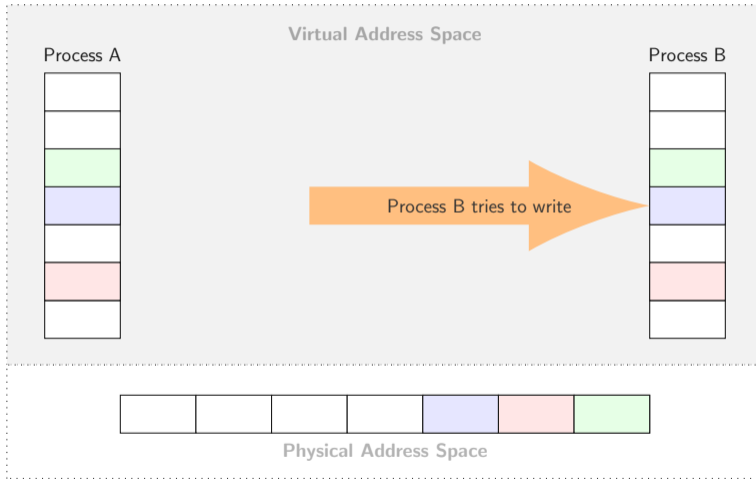


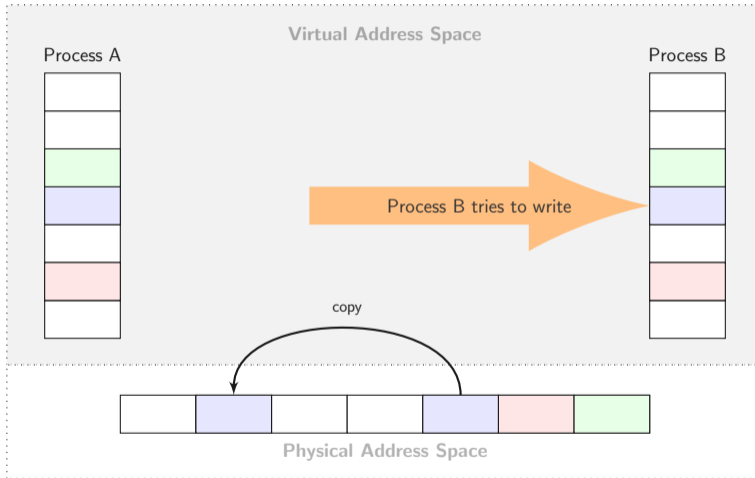


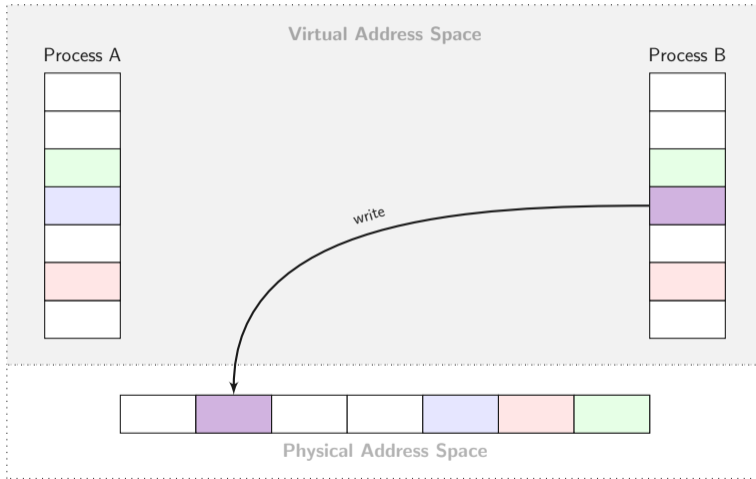


















- Starting `/usr/shell.sweb` twice, without fork



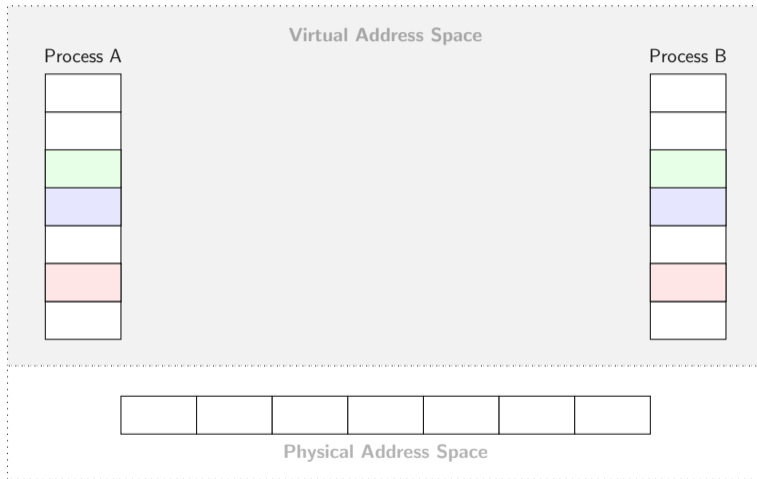
- Starting `/usr/shell.sweb` twice, without fork
- Loading the same image in different programs

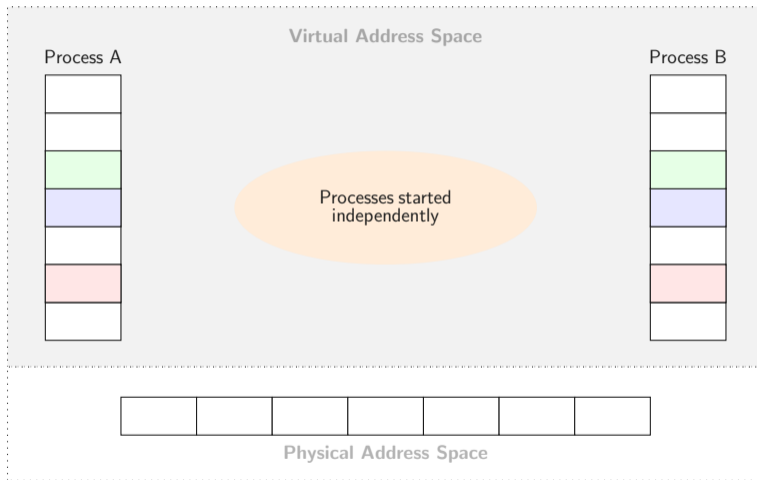


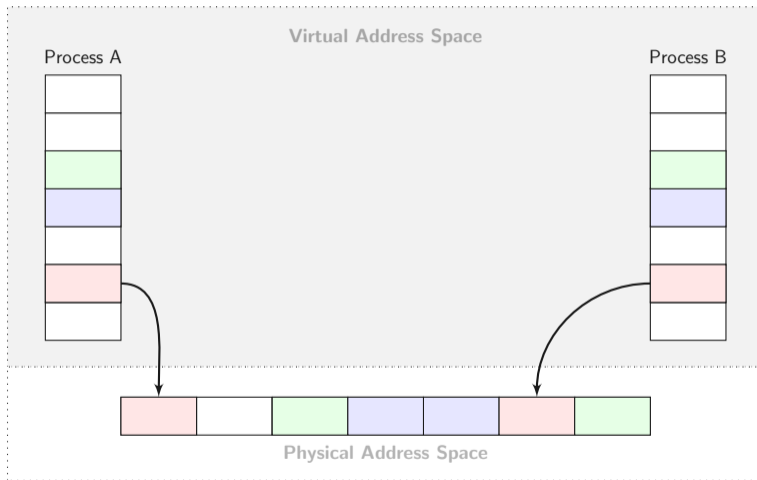
- Starting `/usr/shell.sweb` twice, without fork
- Loading the same image in different programs
- Generating the same data in different programs

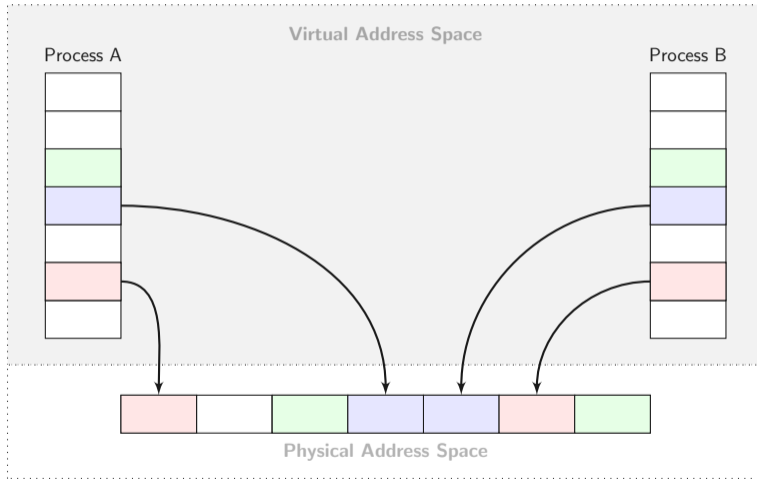


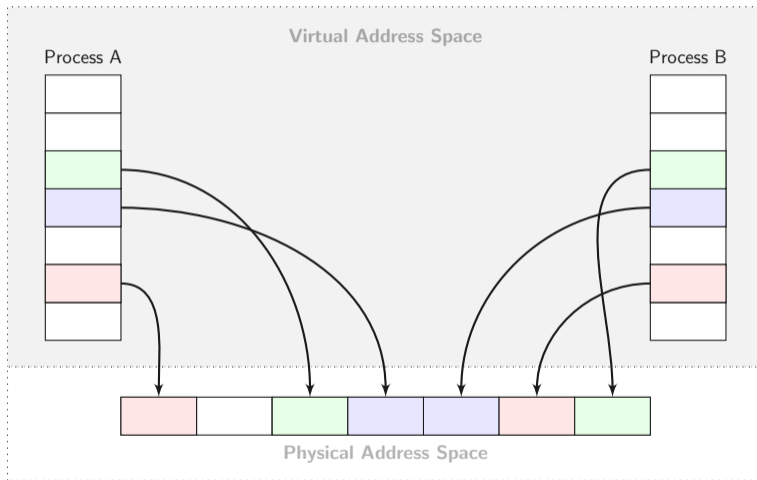
- Starting `/usr/shell.sweb` twice, without fork
- Loading the same image in different programs
- Generating the same data in different programs
- → Page Deduplication

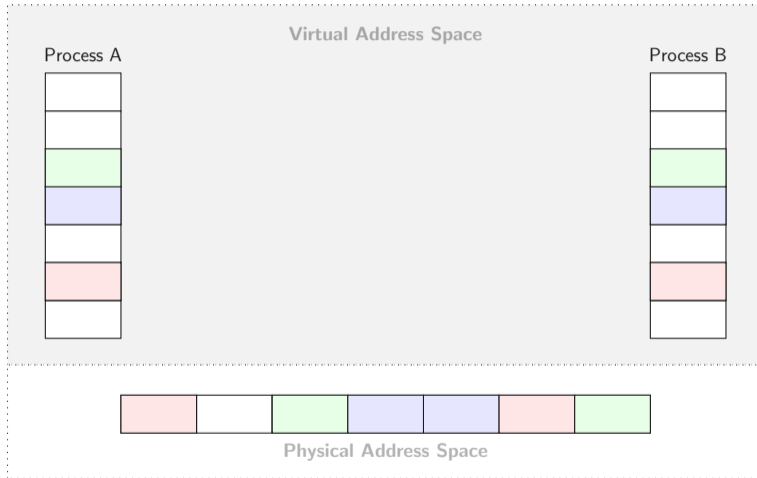


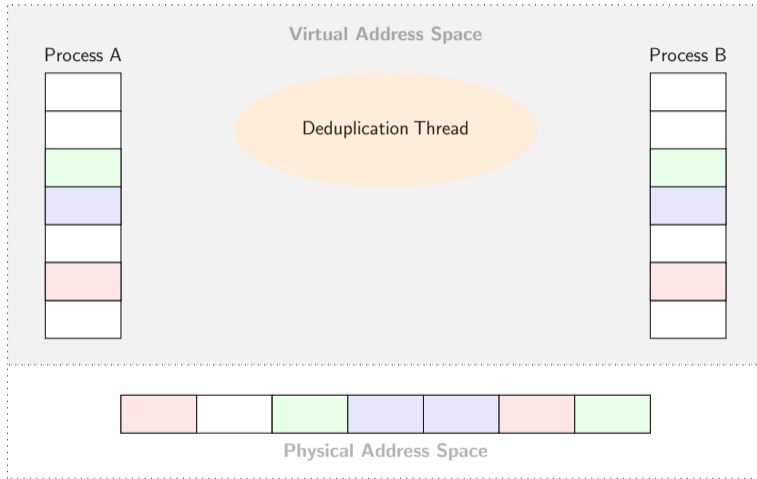


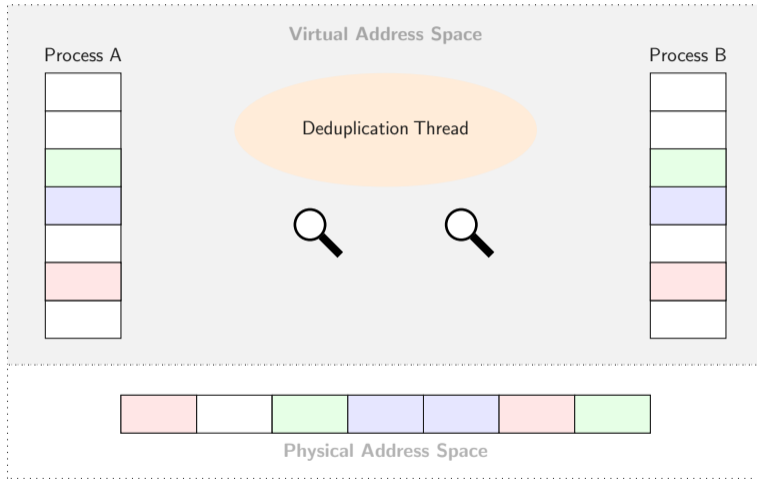


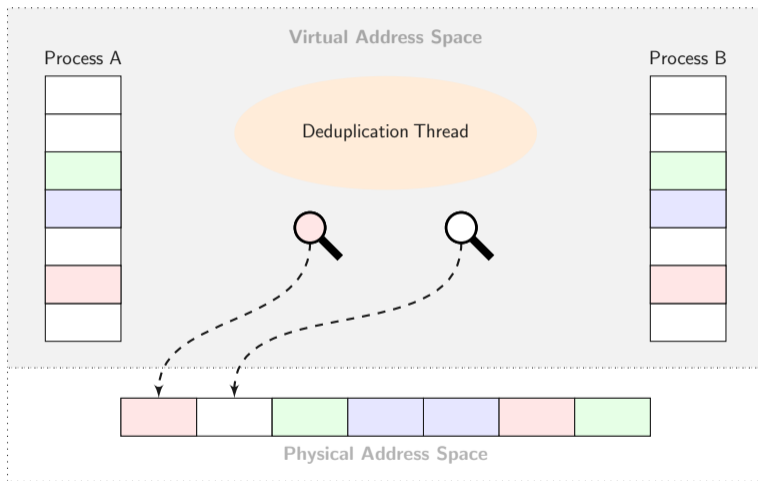


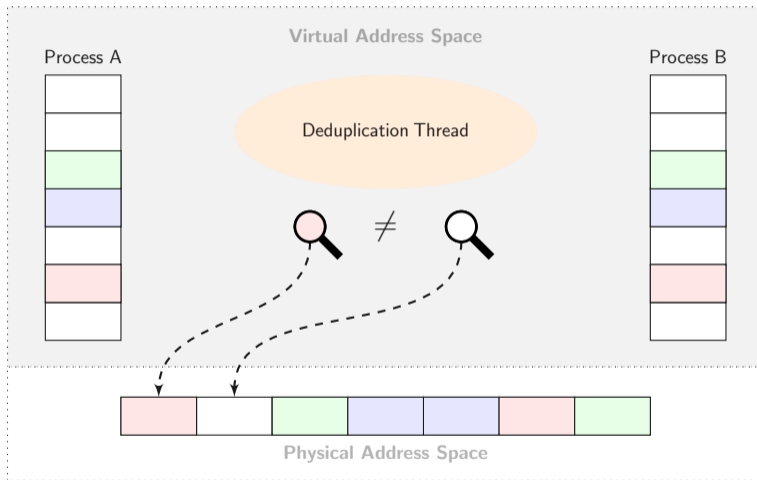


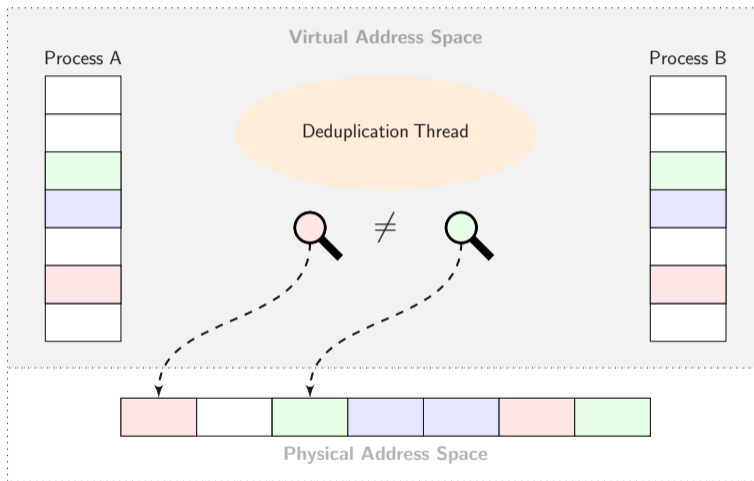


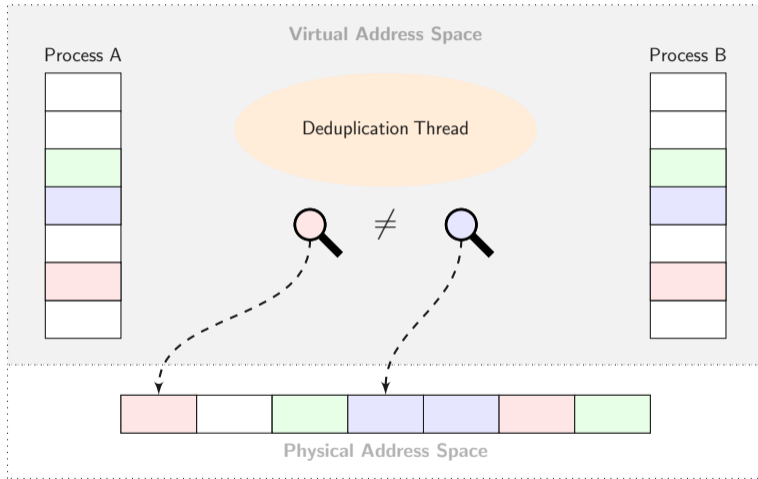


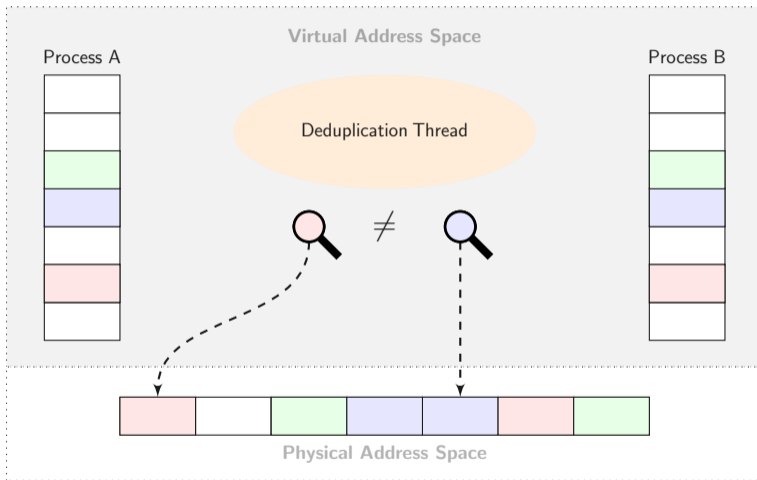


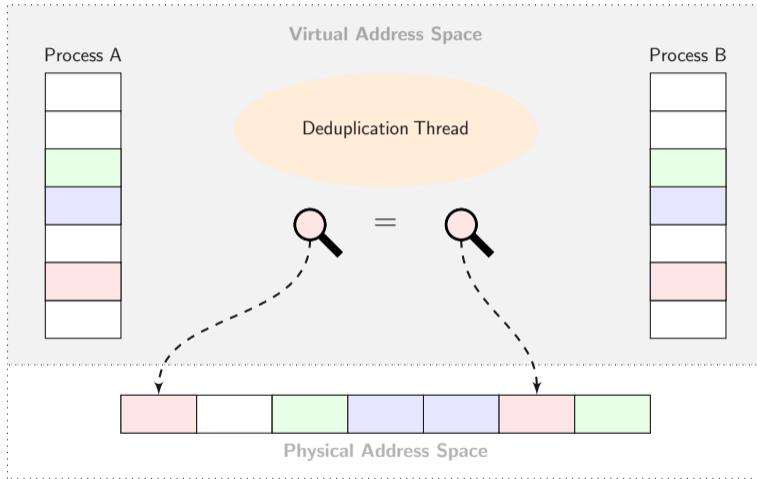


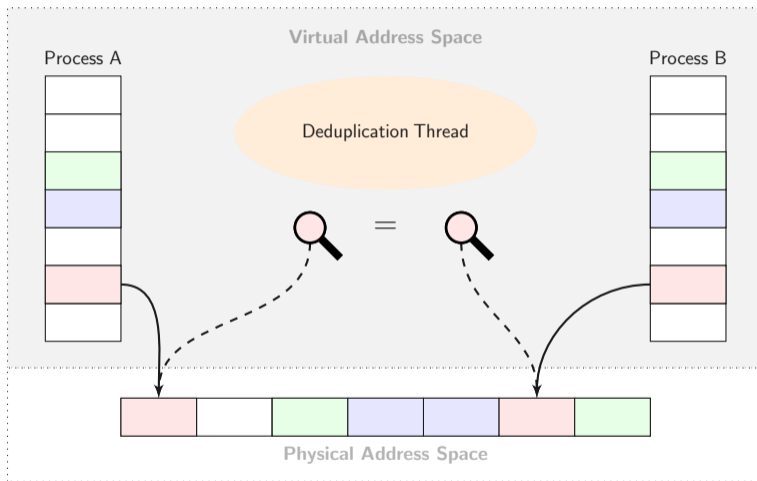


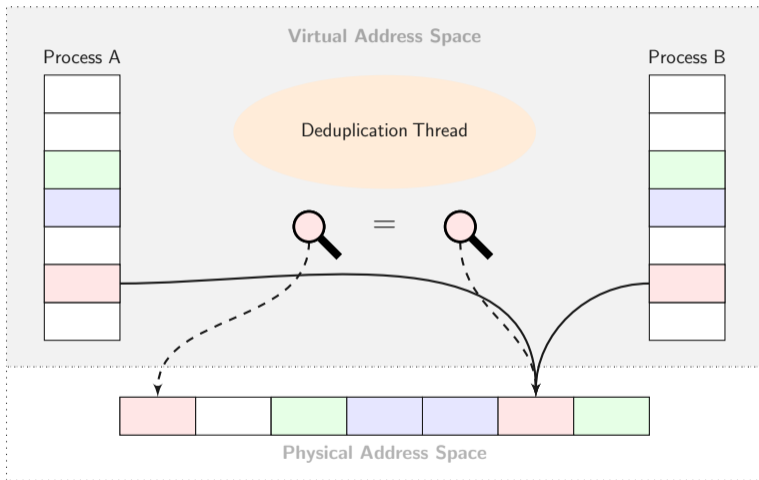


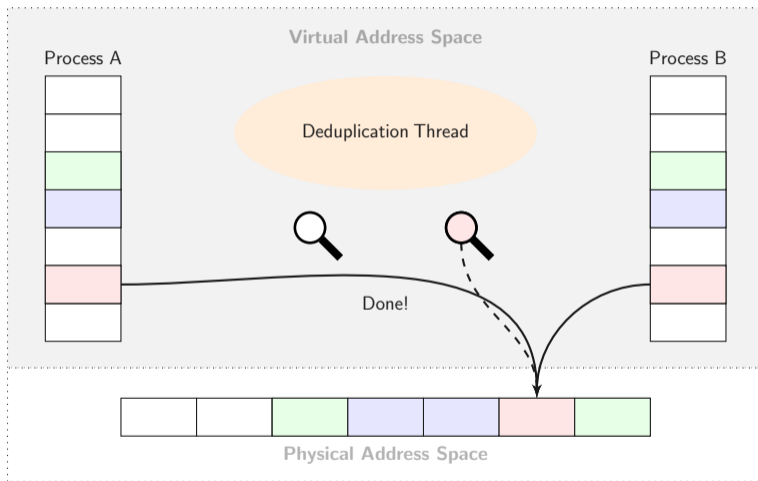


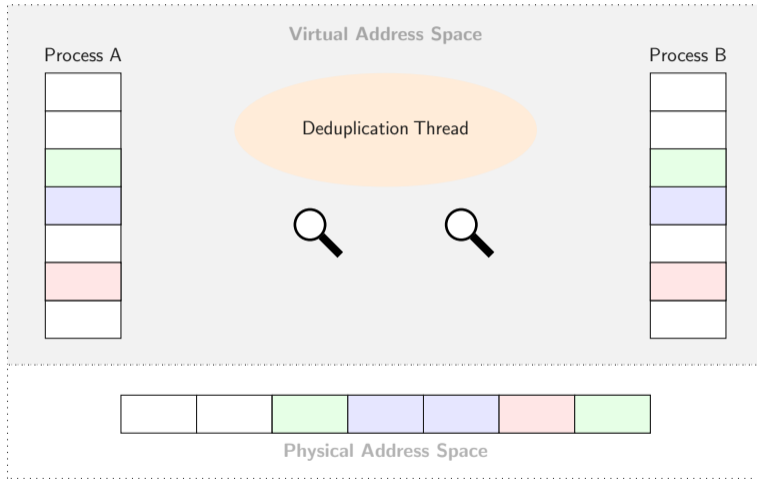












Additional Tasks





1. Process A wants to share 3 pages with process B





1. Process A wants to share 3 pages with process B
2. Process A syscall: get 3 pages of shared memory ID 4



1. Process A wants to share 3 pages with process B
2. Process A syscall: get 3 pages of shared memory ID 4
3. Kernel: maps 3 virtual pages (10-12) of A to physical pages 464, 9078, and 123



1. Process A wants to share 3 pages with process B
2. Process A syscall: get 3 pages of shared memory ID 4
3. Kernel: maps 3 virtual pages (10-12) of A to physical pages 464, 9078, and 123
4. Process B syscall: get 3 pages of shared memory ID 4



1. Process A wants to share 3 pages with process B
2. Process A syscall: get 3 pages of shared memory ID 4
3. Kernel: maps 3 virtual pages (10-12) of A to physical pages 464, 9078, and 123
4. Process B syscall: get 3 pages of shared memory ID 4
5. Kernel: maps 3 virtual pages (22-24) of A to physical pages 464, 9078, and 123



1. Process A wants to share 3 pages with process B
2. Process A syscall: get 3 pages of shared memory ID 4
3. Kernel: maps 3 virtual pages (10-12) of A to physical pages 464, 9078, and 123
4. Process B syscall: get 3 pages of shared memory ID 4
5. Kernel: maps 3 virtual pages (22-24) of A to physical pages 464, 9078, and 123
6. → A and B now share 3 pages







- Syscalls:

```
int shm_open(const char *name, int oflag, mode_t mode);  
int shm_unlink(const char *name);  
void *mmap(void *addr, size_t len, int prot, int flags, int fildes,  
off_t off);  
int munmap(void *addr, size_t len);
```



- Syscalls:

```
int shm_open(const char *name, int oflag, mode_t mode);  
int shm_unlink(const char *name);  
void *mmap(void *addr, size_t len, int prot, int flags, int fildes,  
off_t off);  
int munmap(void *addr, size_t len);
```

- Manages IDs (pseudo file-descriptor) and users of the shared regions



- Syscalls:

```
int shm_open(const char *name, int oflag, mode_t mode);  
int shm_unlink(const char *name);  
void *mmap(void *addr, size_t len, int prot, int flags, int fildes,  
off_t off);  
int munmap(void *addr, size_t len);
```

- Manages IDs (pseudo file-descriptor) and users of the shared regions
- `munmap` and `shm_unlink` when the process ends or manually



- Syscalls:

```
int shm_open(const char *name, int oflag, mode_t mode);  
int shm_unlink(const char *name);  
void *mmap(void *addr, size_t len, int prot, int flags, int fildes,  
off_t off);  
int munmap(void *addr, size_t len);
```

- Manages IDs (pseudo file-descriptor) and users of the shared regions
- `munmap` and `shm_unlink` when the process ends or manually
- No reference to the shared memory object → destroy it







- Files are not accessed by using (`open/creat/close/read/write`) any longer, they are directly mapped into the address space



- Files are not accessed by using (`open/creat/close/read/write`) any longer, they are directly mapped into the address space
- Parts of the mapped file are copied into RAM on demand!



- Files are not accessed by using (`open/creat/close/read/write`) any longer, they are directly mapped into the address space
- Parts of the mapped file are copied into RAM on demand!
- They are written back when being unmapped (if they have been modified)



- Files are not accessed by using (`open/creat/close/read/write`) any longer, they are directly mapped into the address space
- Parts of the mapped file are copied into RAM on demand!
- They are written back when being unmapped (if they have been modified)
 - Depends on the flags set when being mapped



- Files are not accessed by using (`open/creat/close/read/write`) any longer, they are directly mapped into the address space
- Parts of the mapped file are copied into RAM on demand!
- They are written back when being unmapped (if they have been modified)
 - Depends on the flags set when being mapped
- If several processes have the same file mapped → Shared Memory

- `void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);`
- `int munmap(void *addr, size_t len);`

- fildes**
- `shm_open` or `open`
 - Which processes opened the same file?
- len**
- Only multiples of `PAGE_SIZE`
 - File size usually not `PAGE_SIZE`-aligned



- `void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);`
- `int munmap(void *addr, size_t len);`

protection : Access rights for the mapped areas

- PROT_READ: How to prevent write accesses?
- PROT_WRITE: flags relevant!



- `void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);`
- `int munmap(void *addr, size_t len);`

flags :

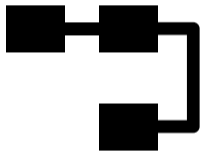
- MAP_PRIVATE:
 - Copy-on-write
 - No write-back
- MAP_SHARED:
 - Write-back to file on `munmap`
 - Changes visible to other processes immediately!

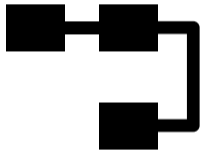


- `void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);`
- `int munmap(void *addr, size_t len);`

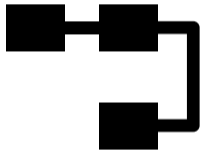
... easy to combine with shared memory syscalls

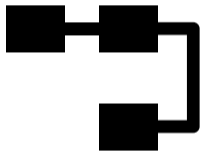




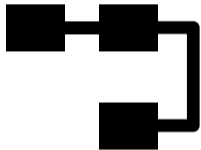


- Memory allocation at runtime





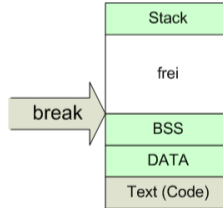
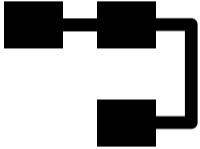
- Memory allocation at runtime
- Implement malloc/free

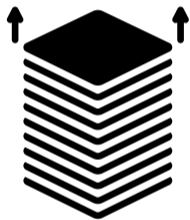


- Memory allocation at runtime
- Implement malloc/free

- Memory allocation at runtime
- Implement malloc/free

Address space of a process:





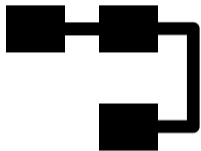
- `int brk(void *end_data_segment);`
- `void *sbrk(int increment);`
- Linker symbol `_end`

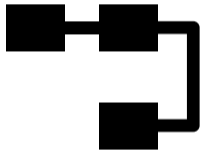


- `int brk(void *end_data_segment);`
- `void *sbrk(int increment);`
- Linker symbol `_end`

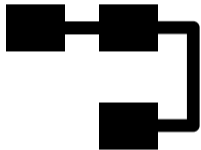
Example (sbrk/break in userspace)

```
extern _end;
//...
size_t heap_start = &_end;
size_t heap_end = heap_start + 4096;
if ( brk(heap_end) == 0)
{
    //do stuff in dynamic memory
}
```

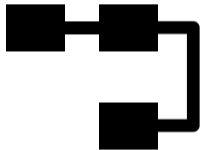


`brk` and `sbrk` are complicated to use - let's implement:



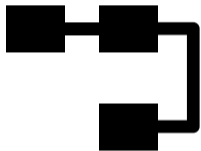
`brk` and `sbrk` are complicated to use - let's implement:

- `malloc(size_t size)/free(void *p)` in libc



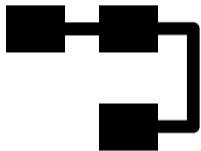
`brk` and `sbrk` are complicated to use - let's implement:

- `malloc(size_t size)/free(void *p)` in libc
- Manages the allocated memory regions



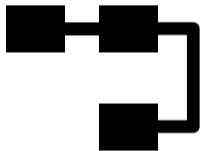
`brk` and `sbrk` are complicated to use - let's implement:

- `malloc(size_t size)/free(void *p)` in libc
- Manages the allocated memory regions
 - Requests pages from the kernel



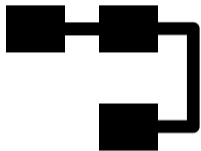
`brk` and `sbrk` are complicated to use - let's implement:

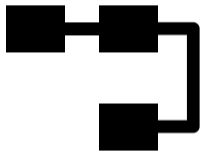
- `malloc(size_t size)/free(void *p)` in libc
- Manages the allocated memory regions
 - Requests pages from the kernel
 - Frees unused pages again

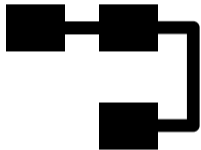


`brk` and `sbrk` are complicated to use - let's implement:

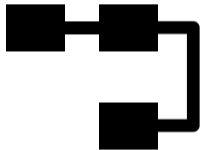
- `malloc(size_t size)/free(void *p)` in libc
- Manages the allocated memory regions
 - Requests pages from the kernel
 - Frees unused pages again
 - therefore uses `brk()/sbrk()`







- simple implementation:
doubly-linked list containing the memory regions



- simple implementation:
doubly-linked list containing the memory regions
- Don't forget about locking!

Design / Submissions

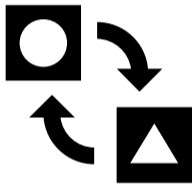


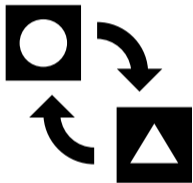
Proof-of-Concept-Implementation as in Assignment 1

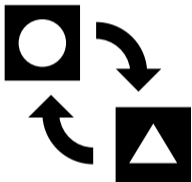


Proof-of-Concept-Implementation as in Assignment 1

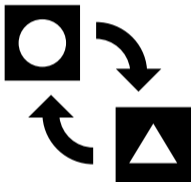
Recommendation: Start with swapping



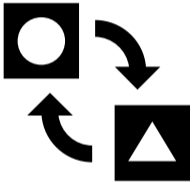




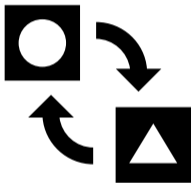
- Normal way: mandatory task virtual memory



- Normal way: mandatory task virtual memory
- You want to go the normal way? Just ignore this slide...



- Normal way: mandatory task virtual memory
- You want to go the normal way? Just ignore this slide...
- Alternative: Discuss with **me** about substituting the mandatory task with either security or driver development as your new mandatory task



- Normal way: mandatory task virtual memory
- You want to go the normal way? Just ignore this slide...
- Alternative: Discuss with **me** about substituting the mandatory task with either security or driver development as your new mandatory task
- This is not possible without discussing it with **me**!







- As in Assignment 1



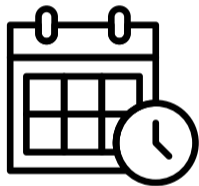
- As in Assignment 1
- Tags:

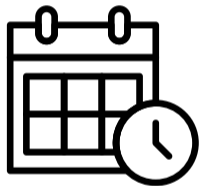


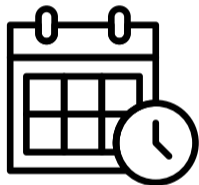
- As in Assignment 1
- Tags:
 - Design/Proof-of-Concept: `SubmissionD2`



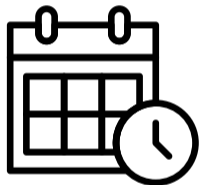
- As in Assignment 1
- Tags:
 - Design/Proof-of-Concept: `SubmissionD2`
 - Implementation: `SubmissionI2`



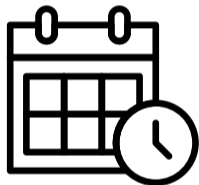




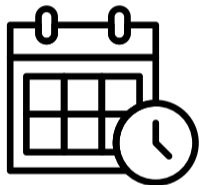
- Design-PoC: 22.05.2021, 18:00



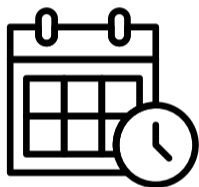
- Design-PoC: 22.05.2021, 18:00
 - Individual feedback meetings ideally between 24.05. and 28.05.



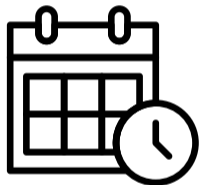
- Design-PoC: 22.05.2021, 18:00
 - Individual feedback meetings ideally between 24.05. and 28.05.
- Implementation: 18.06.2021, 18:00



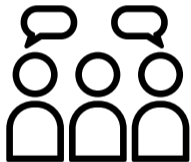
- Design-PoC: 22.05.2021, 18:00
 - Individual feedback meetings ideally between 24.05. and 28.05.
- Implementation: 18.06.2021, 18:00
- Since 2011 we went to a pub after the implementation deadline

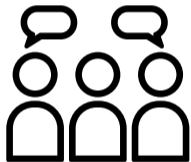


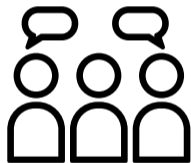
- Design-PoC: 22.05.2021, 18:00
 - Individual feedback meetings ideally between 24.05. and 28.05.
- Implementation: 18.06.2021, 18:00
- Since 2011 we went to a pub after the implementation deadline
- Except for last semester



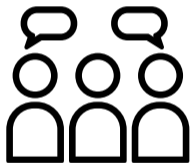
- Design-PoC: 22.05.2021, 18:00
 - Individual feedback meetings ideally between 24.05. and 28.05.
- Implementation: 18.06.2021, 18:00
- Since 2011 we went to a pub after the implementation deadline
- Except for last semester
- Maybe not on the 18th, but maybe later in the summer?



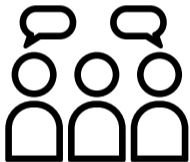




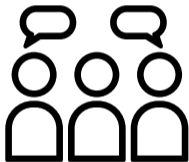
- In two weeks (10.05.-14.05.)



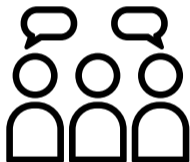
- In two weeks (10.05.-14.05.)
- Like the one from Assignment 1



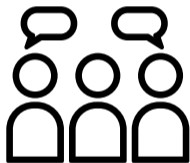
- In two weeks (10.05.-14.05.)
- Like the one from Assignment 1
- Compulsory attendance



- In two weeks (10.05.-14.05.)
- Like the one from Assignment 1
- Compulsory attendance
- Repeating the assignment specification is not enough!



- In two weeks (10.05.-14.05.)
- Like the one from Assignment 1
- Compulsory attendance
- Repeating the assignment specification is not enough!
- Your design should be complete by that time



- In two weeks (10.05.-14.05.)
- Like the one from Assignment 1
- Compulsory attendance
- Repeating the assignment specification is not enough!
- Your design should be complete by that time
- Instant feedback







- Tell us what was good and should remain the same



- Tell us what was good and should remain the same
- Tell us what was bad and should be changed



- Tell us what was good and should remain the same
- Tell us what was bad and should be changed



- Tell us what was good and should remain the same
- Tell us what was bad and should be changed

