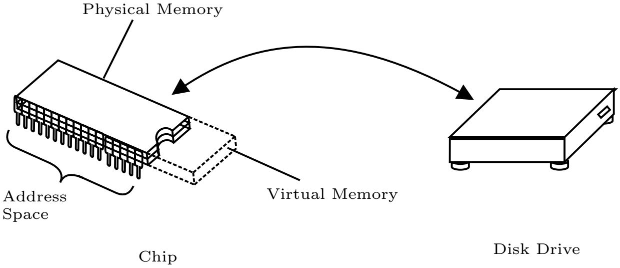
Lab 8: Memory Management

Giancarlo Succi, Leonard Johard

Innopolis University
Course of Operating Systems

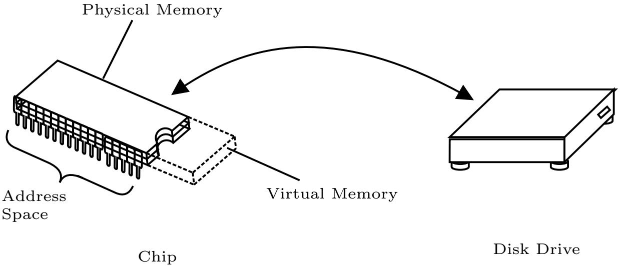
2023

Virtual Memory



- Virtual memory is a memory management technique used by operating systems.
- It allows to temporarily increase the capacity of the phyiscal memory — RAM — by using secondary memory such as a hard drive or solid-state drive (SSD).
- Virtual memory utilizes hardware and software to manage how information is stored and retrieved from the hard drive.

Virtual Memory



- Processor operates with Virtual Memory addresses
- Actual data (source code + data) is stored in Physical Memory
- Page tables: Virtual Memory → Physical Memory

Terminology in Virtual Memory

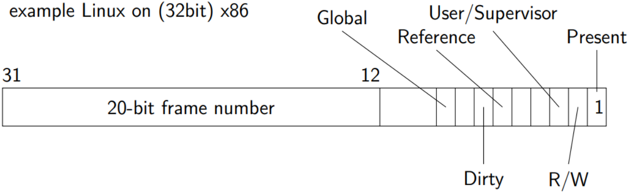
- **A page (virtual page):** is a continuous region of virtual memory and contains n bytes where $n = page\_size$. A page must be *page-aligned*, that is, it starts on a virtual address evenly divisible by the *page_size*. Thus, a 32-bit virtual address can be divided into a 20-bit page number and a 12-bit page offset (or just offset), like this:

```
 31               12 11         0
+-------------------+-----------+
|    Page Number    |   Offset  |
+-------------------+-----------+
         Virtual Address
```

- **A frame (physical frame or a page frame):** is main memory area that can hold one page is a page frame. Like pages, frames must be page-size and page-aligned. Thus, a 32-bit physical address can be divided into a 20-bit frame number and a 12-bit frame offset (or just offset).

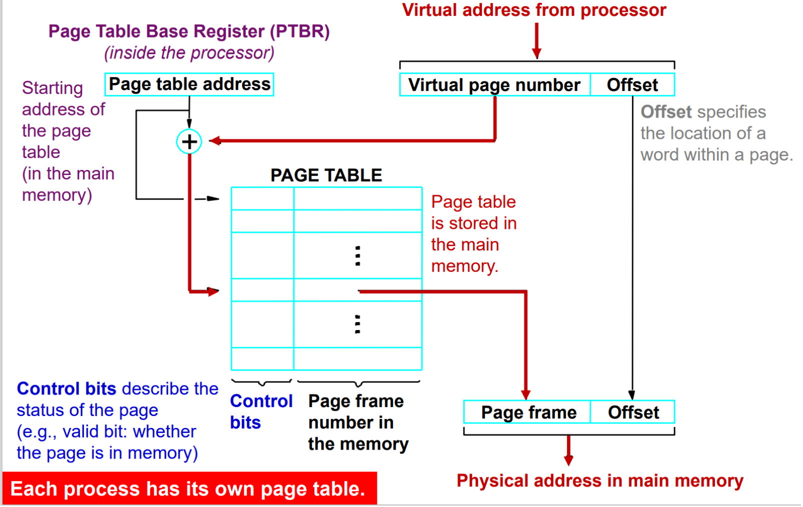
Terminology in Virtual Memory

- **Page table:** is a data structure that the CPU uses to translate a virtual address to a physical address.
  - Each process has its own table (virtual address space). Page table is stored in the main memory.
  - Starting address of the page table is stored in a page table base register (PTBR) inside the processor.
  - Each entry in the table consists of a frame number and control bits.



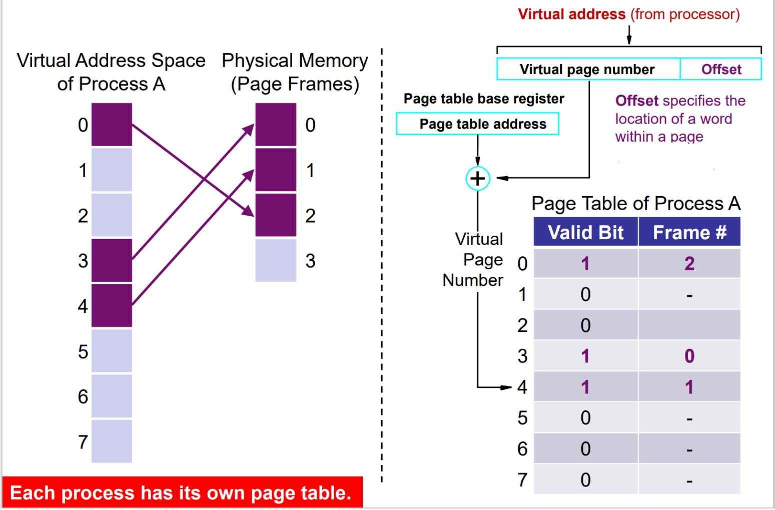
https://www.kernel.org/doc/gorman/html/understand/understand006.html

Address Translation with Paging


Virtual address from processor
Page Table Base Register (PTBR)
(inside the processor)
Page table address
Virtual page number
Offset
Starting address of the page table (in the main memory)
Offset specifies the location of a word within a page.
+
PAGE TABLE
Page table is stored in the main memory.
Control bits describe the status of the page (e.g., valid bit: whether the page is in memory)
Control bits
Page frame number in the memory
Page frame
Offset
Each process has its own page table.
Physical address in main memory

The link to the source.

Page table of a process



Offset specifies the location of a word within a page

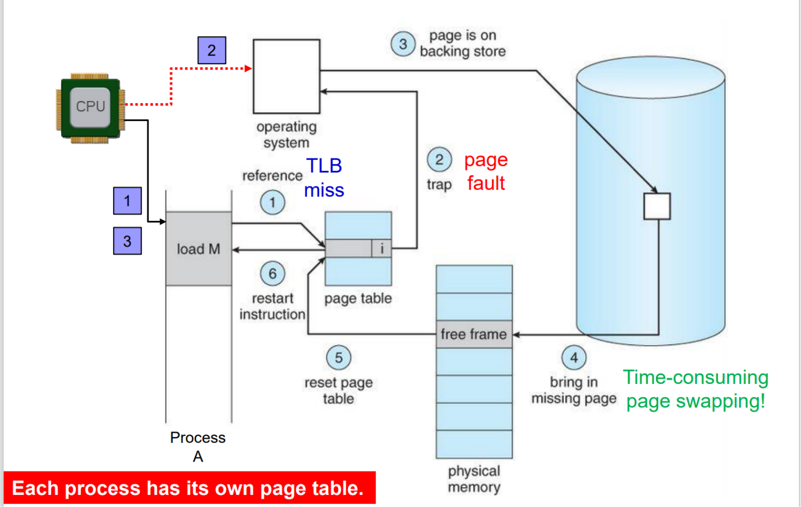
Page Table of Process A

	Valid Bit	Frame #
0	1	2
1	0	-
2	0	
3	1	0
4	1	1
5	0	-
6	0	-
7	0	-

Each process has its own page table.

The link to the source.

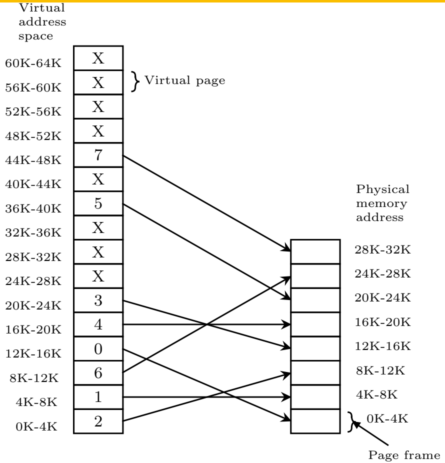
Page fault



Each process has its own page table.

The link to the source.

Purpose of Virtual Memory



Physical memory may not be as large as the “possible space” that can be addressed by a CPU, possible address space is 18.4 TB with 64-bit address, but the space of installed main memory may only be 8GB.

Virtual memory

Example:

- A computer system has a 36-bit virtual address space with a page size of $8KiB$, and 4 bytes per page table entry.
  - How many pages are in the virtual address space?

Virtual memory

Example:

- A computer system has a 36-bit virtual address space with a page size of $8KiB$, and 4 bytes per page table entry.
  - How many pages are in the virtual address space?
    - A 36 bit address can address 2^{36} bytes in a byte addressable machine. Since the size of a page $8KiB$ bytes (2^{13}), the number of addressable pages is $\frac{2^{36}}{2^{13}} = 2^{23}$

Virtual memory

Example:

- A computer system has a 36-bit virtual address space with a page size of $8KiB$, and 4 bytes per page table entry.
  - How many pages are in the virtual address space?
    - A 36 bit address can address 2^{36} bytes in a byte addressable machine. Since the size of a page $8KiB$ bytes (2^{13}), the number of addressable pages is $\frac{2^{36}}{2^{13}} = 2^{23}$
  - What is the maximum size of addressable physical memory in this system?

Virtual memory

Example:

- A computer system has a 36-bit virtual address space with a page size of $8KiB$, and 4 bytes per page table entry.
  - How many pages are in the virtual address space?
    - A 36 bit address can address 2^{36} bytes in a byte addressable machine. Since the size of a page $8KiB$ bytes (2^{13}), the number of addressable pages is $\frac{2^{36}}{2^{13}} = 2^{23}$
  - What is the maximum size of addressable physical memory in this system?
    - With 4 byte entries in the page table we can reference 2^{32} pages. Since each page is 2^{13} Bytes long, the maximum addressable physical memory size is $2^{32} * 2^{13} = 2^{45}$ Bytes.

The link to the source.

Swapping vs. Paging

- Swapping and paging are different memory management techniques.
- **Swapping:** To swap a process means to move that entire process out of main memory and to the swap area on hard disk, whereby all pages of that process are moved at the same time.
  - The disadvantage of a performance penalty. When a swapped out process becomes active, the kernel has to load an entire process (perhaps many pages of memory) back into RAM from the swap space. With large processes this is understandably slow.
- **Paging:** was introduced as a solution to the inefficiency of swapping entire processes in and out of memory at once.
  - With paging, when the kernel requires more main memory for an active process, only the least recently used pages of processes are moved to the swap space (page replacement algorithms will be covered in lab 9).

Swapping vs. Paging

  - When a process that has paged out memory becomes active, it is likely that it will not need access to the pages of memory that have been paged out to the swap space, and if it does then at least only a few pages need to be transferred between disk and RAM.
  - **Demand Paging:** is a technique in which a page is usually brought into the main memory only when it is needed or demanded by the CPU. Initially, only those pages are loaded that are required by the process immediately. Those pages that are never accessed are thus never loaded into the physical memory.
- **Working set:** For efficient paging, the kernel needs to keep regular statistics on the memory activity of processes it keeps track of which pages a process has most recently used. These pages are known as the working set.
  - When the kernel needs memory, it will prefer to keep pages in the working sets of processes in RAM as long as possible and to rather page out the other less recently used pages as they have statistically been proven to be less frequently accessed, and therefore unlikely to be accesses again in the near future.

Virtual memory of a process

Practice (Example):

- Create a simple program which stores only a message in the heap.
- Keep the program running and access the maps file at /proc/{pid}/maps. Print the content of the file to stdout. Locate the *start* and *end* addresses of the heap.
- You can access the virtual memory of the process whose id is *pid* at */proc/{pid}/mem*. Try to access the file using *less* command.
- Access the heap of the program using *xdd* command as follows:

```
sudo xxd -s $start -l $(($end - $start)) /proc/$pid/mem | less
```

- Navigate the heap memory file till finding the message. Locate the page number where the message is stored and its address.
- Access the file /proc/{pid}/smaps and specify what the total size of pages of the heap. What is the file /proc/{pid}/map_files?
- Use the command **pmap** to check the memory mappings of the process.

Exercise 1

- Write a C program “ex1.c” which writes its pid in /tmp/ex1.pid and generates a random password using the file /dev/random or /dev/urandom. The password should start with “pass:”. It should consist of only printable characters. The length is 8 characters.
- Store the password in the memory using **mmap** (Create a shared anonymous mapping) or in the heap. Then, the program waits in an infinite loop.
- Write a script “ex1_hack.sh” which reads the pages of the virtual memory of the program “ex1.c” while it is running. It should find the password that is generated by “ex1.c”. Print the password and its memory address to **stdout** and send *SIGKILL* to the program “ex1.c”.
- Submit **ex1.c** and **ex1_hack.sh**.
- **Hints:** You need to search in /proc/{pid}/mem, using *xxd* or *gdb* commands. You may need to add *sudo* for these commands.

Exercise 2 (1/15)

- In this exercise, you will implement a simulation of virtual memory in C to further understand how virtual memory works. In addition, you will use *mmap* and signals for process communication.
- Virtual memory allows a process of P pages to run in F frames, even if $F < P$. This mapping is achieved by use of a *page table*, which records which pages are in RAM at which frames, and a page fault mechanism by which the memory management unit (MMU) can ask the operating system (here it is the pager) to bring in a page from disk. Each process has a page table. The page table must be accessible by both the MMU and the pager, and an IPC mechanism is needed for communication between the MMU and pager. In this simulation, the job of the pager is to maintain a process' use of RAM and the page table. The page table is held in memory backed by a file */tmp/ex2/pagetable*, and signals are used for IPC.

Exercise 2 (2/15)

- For simplicity, we assume that we run the program only for one process, the RAM is represented as an array `RAM` of strings of size F. The disk is represented by a an array `disk` of strings of size P. In both arrays, each array element (in this simulation, it is considered a page) is represented by a string of size 8.
- On startup, the array `disk` (represented by the disk) should contain different random messages (you can store manually or generate random printable characters) whereas the `RAM` is empty. When the pager needs to bring in a page i from the disk to frame j, it needs to copy the requested message i from `disk` array to `RAM` array in position j, then prints the `RAM` array. When the pager needs to write to disk (move the page of the frame j from the RAM to the disk in position i), it needs to copy the message j from `RAM` array to `disk` array in position i, then prints the `RAM` array. For simpicity, we assume that the write request, does not change the message but it just sets the dirty field.

Exercise 2 (3/15)

- The page table has mainly four fields in each page table entry and defined as follows:

```
struct PTE{
    // The page is in the physical memory (RAM)
    bool valid;
    // The frame number of the page in the RAM
    int frame;
    // The page should be written to disk
    bool dirty;
    // The page is referenced/requested
    int referenced;
}
```

Exercise 2 (4/15)

- The pager process must create the page table as a memory mapped file, and initialize it to indicate that no pages are loaded (all valid and dirty fields are set to false, frame field is set to -1 and referenced is set to 0). You can add more fields to the structure based on your implementation.
- **Note:** After you create a file to be mapped, you need to truncate it to the size of the page table using `ftruncate` before you map it using *mmap*. The size of the page table is the size of the struct PTE multiplied by the number of pages.
- Write a program **mmu.c** which accepts the command line arguments:
  - The number of pages in the process.
  - A reference string of memory accesses, each of the form mode page , e.g., W3 is a write to page 3.
  - The PID of the pager process.

Exercise 2 (5/15)

- The MMU opens the mapped file */tmp/ex2/pagetable*, then runs through the reference string. For each memory access, the MMU:
  1. Checks if the page is in RAM.
  2. If not in RAM ($valid = 0$), it sets the referenced field of the page to the PID of the MMU, and simulates a page fault by signaling the pager process with *SIGUSR1*. After that, it sleeps indefinitely until it receives a *SIGCONT* signal from the pager process to indicate that the page has been loaded to RAM.
  3. If the access is a write access, it sets the dirty field of the page.
  4. It prints the updated page table.
- When all memory accesses have been processed, the MMU closes the mapped file and signals the pager one last time (SIGUSR1). That must be detected by the pager process. If the pager process did not find any page referenced then the pager process can unmap the mapped file, delete the file and exit.
- Write another program **pager.c** which must take two arguments:
  - The number of pages in the process.
  - The number of frames allocated to the process.

Exercise 2 (6/15)

- Assume that the pages and frames are numbered 0, 1, 2, ...
- The pager process manages free frames, the disk array and RAM array.
- After creating and initializing the page table in the mapped file, the pager process must wait till accepting a SIGUSR1 signal from the MMU process. When it receives a signal, it must:
    1. Scan through the page table looking for a non-zero value in the referenced field.
    2. If a non-zero value is found, that indicates that the MMU wants the page at that index loaded.
        - If there is a free frame, then allocate it to the page.
        - If there are no free frames, choose a random frame in the table as a victim page (page replacement next lab). If the victim page is dirty, simulate writing the page to disk by copying the change from RAM to disk array, and increment the counter of disk accesses.
        - Update the page table to indicate that the victim page is no longer present in RAM.
        - Update the page table to indicate that the page is valid in the allocated frame, not dirty, and clear the referenced field.

Exercise 2 (7/15)

3. Print the updated page table.
4. Send a SIGCONT signal to the MMU to indicate that the page is now loaded.
5. If no non-zero referenced field was found, the pager process terminates.

- Before terminating the pager process, you must print out the total number of disk accesses, and destroy the mapped file.
- Make sure that the program prints enough informative messages for each step. For example, when there are no free frame, the program should tell the user that a victim page will be selected.
- Make sure that you cannot allocate more than F frames and check the validity of input data.
- Submit **mmu.c** and **pager.c** with a script **ex2.sh** to run the simulation.

Exercise 2 (8/15)

Some test cases:

- Test case 1
  - 4 pages, 2 frames
  - ./pager 4 2
  - ./mmu 4 R0 R1 W1 R0 R2 W2 R0 R3 W2 $*pid_pager*
- Test case 2
  - 5 pages 3 frames
  - ./pager 5 3
  - ./mmu 5 R0 R1 R0 W1 R0 R1 R0 W1 R0 R2 R0 W2 R0 R2 R0 W2 R0 R3 R0 W3 R0 R3 R0 W3 R0 R4 R0 W4 R0 R4 R0 W4 $*pid_pager*
- ...etc

Exercise 2 (9/15)

Sample output format for Test case 1.

pager process

```
-------------------------
Initialized page table
Page 0 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 1 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 2 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 3 ---> valid=0, frame=-1, dirty=0, referenced=0
-------------------------
Initialized RAM
RAM array
Frame 0 --->
Frame 1 --->
-------------------------
Initialized disk
Disk array
Page 0 ----> gEefwaq
Page 1 ----> kjQ2eeq
Page 2 ----> 43R2e2e
Page 3 ----> jji2u32
-------------------------
A disk access request from MMU Process (pid=283032)
Page 0 is referenced
We can allocate it to free frame 0
Copy data from the disk (page=0) to RAM (frame=0)
RAM array
Frame 0 ---> gEefwaq
Frame 1 --->
disk accesses is 1 so far
```

Exercise 2 (10/15)

```
Resume MMU process
-------------------------
A disk access request from MMU Process (pid=283032)
Page 1 is referenced
We can allocate it to free frame 1
Copy data from the disk (page=1) to RAM (frame=1)
RAM array
Frame 0 ---> gEefwaq
Frame 1 ---> kjQ2eeq
disk accesses is 2 so far
Resume MMU process
-------------------------
A disk access request from MMU Process (pid=283032)
Page 2 is referenced
We do not have free frames in RAM
Chose a random victim page 0
Replace/Evict it with page 2 to be allocated to frame 0
Copy data from the disk (page=2) to RAM (frame=0)
RAM array
Frame 0 ---> 43R2e2e
Frame 1 ---> kjQ2eeq
disk accesses is 3 so far
Resume MMU process
-------------------------
```

Exercise 2 (11/15)

```
A disk access request from MMU Process (pid=283032)
Page 0 is referenced
We do not have free frames in RAM
Chose a random victim page 1
Replace/Evict it with page 0 to be allocated to frame 1
Copy data from the disk (page=0) to RAM (frame=1)
RAM array
Frame 0 ---> 43R2e2e
Frame 1 ---> gEefwaq
disk accesses is 5 so far
Resume MMU process
-------------------------
A disk access request from MMU Process (pid=283032)
Page 3 is referenced
We do not have free frames in RAM
Chose a random victim page 0
Replace/Evict it with page 3 to be allocated to frame 1
Copy data from the disk (page=3) to RAM (frame=1)
RAM array
Frame 0 ---> 43R2e2e
Frame 1 ---> jji2u32
disk accesses is 6 so far
Resume MMU process
-------------------------
6 disk accesses in total
Pager is terminated
```

Exercise 2 (12/15)

Sample output format for Test case 1.

mmu process

```
-------------------------
Initialized page table
Page table
Page 0 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 1 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 2 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 3 ---> valid=0, frame=-1, dirty=0, referenced=0
-------------------------
Read Request for page 0
It is not a valid page --> page fault
Ask pager to load it from disk (SIGUSR1 signal) and wait
MMU resumed by SIGCONT signal from pager
Page table
Page 0 ---> valid=1, frame=0, dirty=0, referenced=0
Page 1 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 2 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 3 ---> valid=0, frame=-1, dirty=0, referenced=0

-------------------------
Read Request for page 1
It is not a valid page --> page fault
Ask pager to load it from disk (SIGUSR1 signal) and wait
MMU resumed by SIGCONT signal from pager
Page table
Page 0 ---> valid=1, frame=0, dirty=0, referenced=0
Page 1 ---> valid=1, frame=1, dirty=0, referenced=0
Page 2 ---> valid=0, frame=-1, dirty=0, referenced=0
```

Exercise 2 (13/15)

```
Page 3 ---> valid=0, frame=-1, dirty=0, referenced=0

-------------------------
Write Request for page 1
It is a valid page
It is a write request then set the dirty field
Page table
Page 0 ---> valid=1, frame=0, dirty=0, referenced=0
Page 1 ---> valid=1, frame=1, dirty=1, referenced=0
Page 2 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 3 ---> valid=0, frame=-1, dirty=0, referenced=0

-------------------------
Read Request for page 0
It is a valid page
Page table
Page 0 ---> valid=1, frame=0, dirty=0, referenced=0
Page 1 ---> valid=1, frame=1, dirty=1, referenced=0
Page 2 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 3 ---> valid=0, frame=-1, dirty=0, referenced=0

-------------------------
Read Request for page 2
It is not a valid page --> page fault
Ask pager to load it from disk (SIGUSR1 signal) and wait
MMU resumed by SIGCONT signal from pager
Page table
Page 0 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 1 ---> valid=1, frame=1, dirty=1, referenced=0
Page 2 ---> valid=1, frame=0, dirty=0, referenced=0
Page 3 ---> valid=0, frame=-1, dirty=0, referenced=0
```

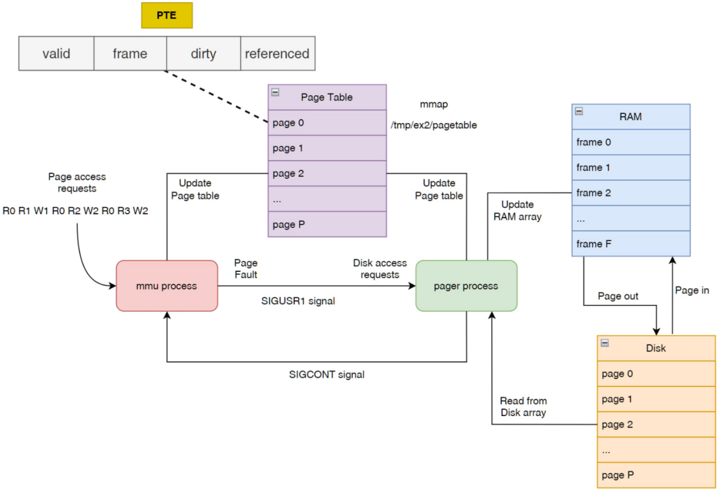
Exercise 2 (14/15)

```
-------------------------
Write Request for page 2
It is a valid page
It is a write request then set the dirty field
Page table
Page 0 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 1 ---> valid=1, frame=1, dirty=1, referenced=0
Page 2 ---> valid=1, frame=0, dirty=1, referenced=0
Page 3 ---> valid=0, frame=-1, dirty=0, referenced=0

-------------------------
Read Request for page 0
It is not a valid page --> page fault
Ask pager to load it from disk (SIGUSR1 signal) and wait
MMU resumed by SIGCONT signal from pager
Page table
Page 0 ---> valid=1, frame=1, dirty=0, referenced=0
Page 1 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 2 ---> valid=1, frame=0, dirty=1, referenced=0
Page 3 ---> valid=0, frame=-1, dirty=0, referenced=0

-------------------------
Read Request for page 3
It is not a valid page --> page fault
Ask pager to load it from disk (SIGUSR1 signal) and wait
MMU resumed by SIGCONT signal from pager
Page table
Page 0 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 1 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 2 ---> valid=1, frame=0, dirty=1, referenced=0
Page 3 ---> valid=1, frame=1, dirty=0, referenced=0
```

Exercise 2 (15/15)

```
-------------------------
Write Request for page 2
It is not a valid page --> page fault
Ask pager to load it from disk (SIGUSR1 signal) and wait
MMU resumed by SIGCONT signal from pager
Page table
Page 0 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 1 ---> valid=0, frame=-1, dirty=0, referenced=0
Page 2 ---> valid=1, frame=0, dirty=1, referenced=0
Page 3 ---> valid=1, frame=1, dirty=0, referenced=0

Done all requests.
MMU sends SIGUSR1 to the pager.
MMU terminates.
```

Exercise 2 (16/16)



A block diagram of the simulation in exercise 2.

free

Example:

- Run 'free -t -h' in the shell or 'vm_stat' on macOS

- **Mem** represents physical memory size
- **Swap** represents size of memory available for swapping
- **Total** represents virtual memory size

vmstat/vm_stat

- Reports information about processes, memory, paging, block IO, traps, and cpu activity
- The first report produced gives averages since the last reboot. Additional reports give information on a sampling period of length delay. The process and memory reports are instantaneous in either case

top

- Provides an ongoing look at processor activity in real time. It displays a listing of the most CPU-intensive tasks on the system, and can provide an interactive interface for manipulating processes

top

Example:

- Run '*top -d 1*' or '*top -i 1*' on macOS
- Run the program in the background and then run 'top'

getrusage()

- C function from `<sys/resource.h>` library to monitor application's memory usage. Refer to 'man 2 getrusage'

```
int getrusage(int who, struct rusage *usage);
```

Check the manual page
https://man7.org/linux/man-pages/man2/getrusage.2.html

Exercise 3

- Write a C program **ex3.c**, that runs for 10 seconds. Every second it should:
  - allocate 10 MB of memory
  - fill it with zeros
  - print memory usage with *getrusage()* function
  - sleep for 1 second
- Compile and run the program in the background (./ex3 &) and run '*vmstat 1*' at the same time. Observe what happens to the memory. Pay attention to **si** and **so** fields.
- Add comments to **ex3.txt** with your findings.
- Hint: use *memset(ptr, value, size)* to fill the allocated memory
- submit **ex3.c** and **ex3.txt**.

Optional exercise

- Download and run Memory Management Simulator
- Installation instructions:
  `http://www.ontko.com/moss/memory/install_unix.html`
- Download:
  `http://www.ontko.com/moss/memory/memory.tgz`
- User guide:
  `http://www.ontko.com/moss/memory/user_guide.html`

End of lab 8