Operating Systems

Recitation 4

Plan

- Virtual Memory
- Memory Mapped Files
- Pipes

Memory: OS view

- OS manages physical RAM.

- OS splits physical RAM and handles ***pages***.
  Memory page – a chunk of sequential memory of fixed length. (4 - 32 Kbytes)

- “Handling” pages –
  allocation / deletion / defragmentation

Memory: OS view

For every process

- OS creates a dedicated **Virtual Address Space.**
  Virtual Address Space consists of pages too.

- OS maps virtual pages to physical pages.
  The mapping can change during defragmentation.

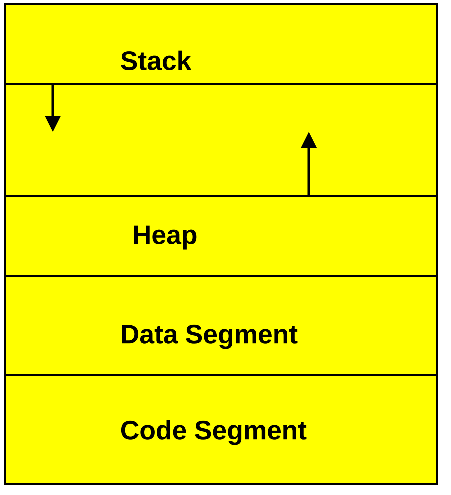
Memory: Processes view

- Process gets its own **Virtual Address Space**.

  4GB (2^{32}bytes) of sequential addressable memory on 32bit.

- Each process has its own

  Heap
  Stack
  Data segment
  Code



Processes & Memory

- Heap

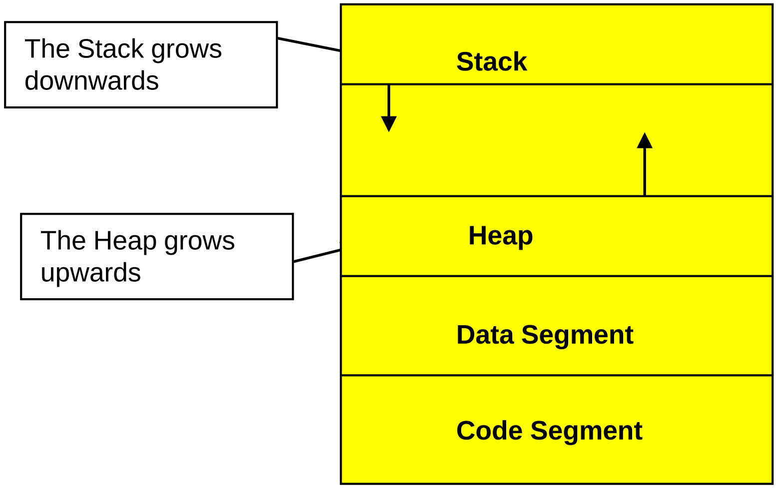
  Dynamic allocations/deletions

```
book* pBook = (book*) malloc(sizeof(book));
```

- Stack

  Automatic variables

```
int function()
{
  int a = 42;
  …
}
```



Processes & Memory

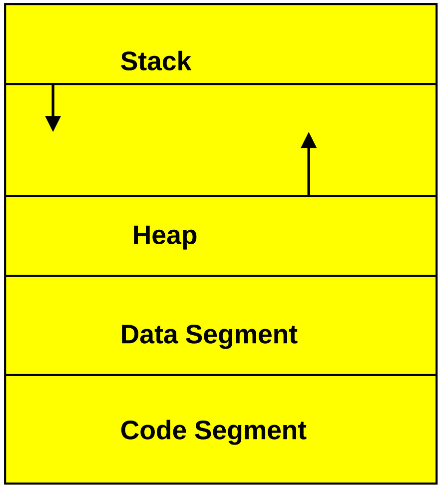
- Data segment

  Global variables, both initialized and not

- Code

  Compiled instructions of your code

  with all the linked libraries



How much virtual memory is used?

There is much more virtual memory pages then physical memory

- Each process has its own address space of 4G for its own code, stack, heap, data

- 10 processes require 40GB, 100 processes – 400GB…

How much virtual memory is used?

Mapping takes memory as well

- **4 bytes** per “virtual -> physical” mapping entry (more on 64-bit systems…)
- Each page is **4KB** (for example)
- **1GB** of virtual memory = **1GB/4KB** entries = **256,000** entries
- **256,000** entries * **4 bytes** per entry

1MB per 1GB of <u>virtual</u> memory just for mapping

- 1 process = **4GB** virtual memory = **4 MB** of memory for mapping
- 50 processes = **4 (MB)** * **50** (processes)…

200MB of physical memory just for mapping!

We don’t have that much. What’s the solution?

Some good news

- Not all the memory is used by a process all the time.
- Some chunks of memory are the same for (almost) every process. (Common libraries)
- Only “used” virtual memory is worth mapping.

Sharing with others

OS recognizes identical data structures and code pages mapped into different virtual addresses of different process.

Single physical copy for the same stuff.

Shared Objects / DLLs

- Most programs (and therefore processes) use a lot of common libraries
  - malloc(), printf()…

- OS “maps” common system libraries to the virtual memory of every process to <u>the same</u> addresses
  - Why?

Swapping memory

- **Memory hit**
  - Process is accessing page which is already in physical memory.
  - **Fast**
- Memory miss/**Page fault**
  - Other page should be unloaded to disk and required page loaded.
  - **Slow**

Swapping memory

“Rarely used” pages are saved **<u>on disk</u>**,
loaded back to physical memory on demand.

- In Linux – special swap partition on /dev/sda
- In Windows - C:\pagefile.sys

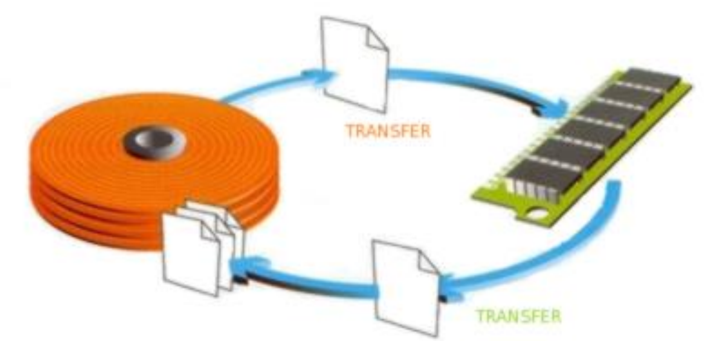
```
moshe@moshe-VirtualBox:~$ sudo fdisk -l /dev/sda
[sudo] password for moshe:

Disk /dev/sda: 8589 MB, 8589934592 bytes
255 heads, 63 sectors/track, 1044 cylinders, total 16777216 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0008b338

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *        2048    15728639     7863296   83  Linux
/dev/sda2        15730686    16775167      522241    5  Extended
/dev/sda5        15730688    16775167      522240   82  Linux swap / Solaris
moshe@moshe-VirtualBox:~$
```

Swapping memory (2)

- We can also “load to memory” and “swap” regular files!
- Called **memory mapping**
- Special system call maps Virtual addresses to offset in file
- Reading/writing to memory address causes read()/write() on file
  - OS in-charge of syncing disk & memory content



Memory mapped files

We can take a regular file and map it into memory.

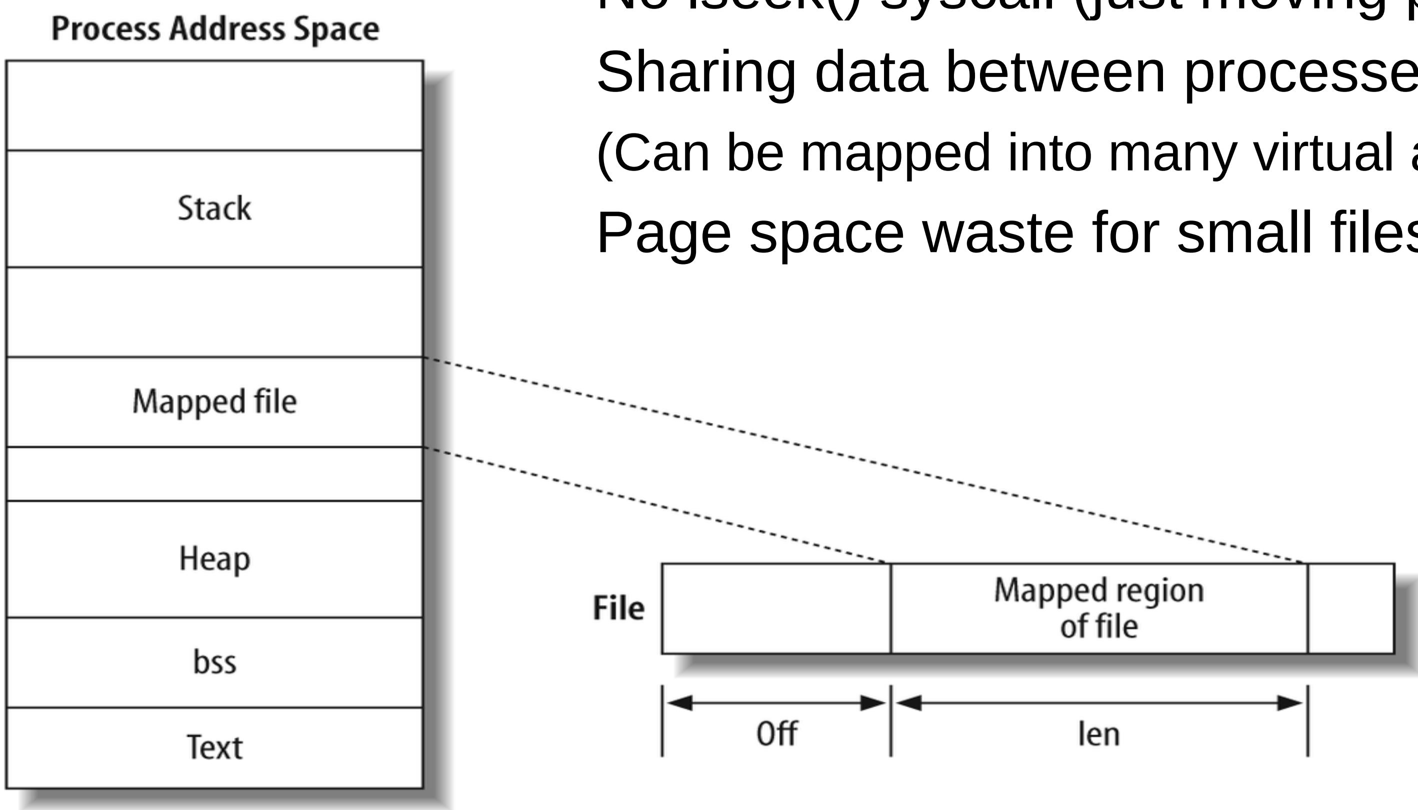
No extra-buffered I/O ☺

No lseek() syscall (just moving pointer) ☺

Sharing data between processes ☺

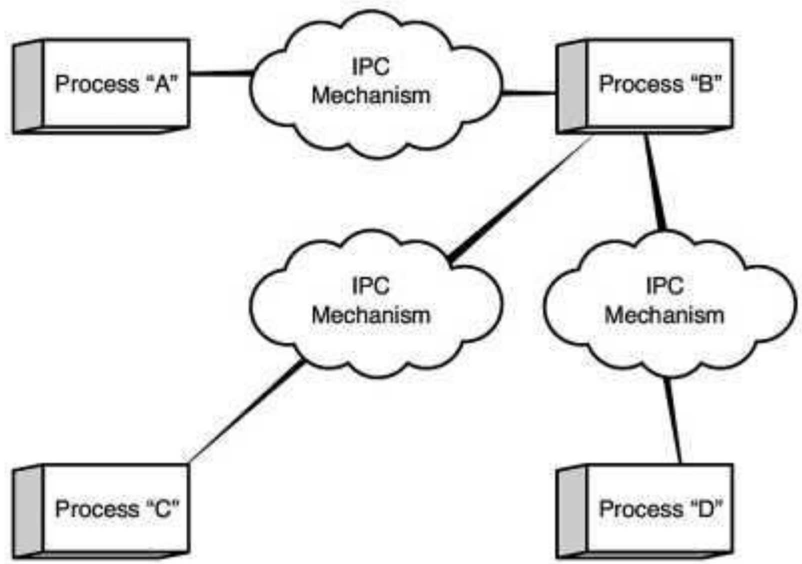
(Can be mapped into many virtual address spaces)

Page space waste for small files ☹



Share data

- Inter-process communication (IPC)
  - Map same file to several processes!
  - Can transfer and share data between processes



mmap()

void* mmap(void* start, size_t length,
int prot, int flags, int fd, off_t offset);

- Memory map virtual addresses to physical
  - <u>start</u> = preferred start addr, if NULL, OS chooses (rec.)
  - <u>length</u> = in bytes
  - <u>fd</u> + <u>offset</u> = file and offset
  - <u>prot</u> = protection (rwx…)
  - <u>flags</u> = visibility, if changes are carried out to file, and therefore visible to other processes

- Returns pointer to the mapped area

munmap(), msync()

int munmap(void *start, size_t length);

- deletes the mapping, flushes to disk any changes to file

int msync(void *start, size_t length, int flags);

- Flushes changes to file
  - flags = MS_SYNC (immediately)
  - MS_ASYNC (schedule sync)

- ***Code example***

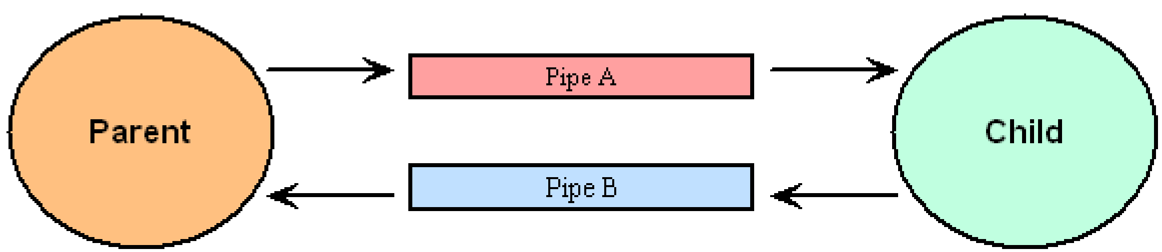
Named Pipes

- A special file type
- Actually special buffer in memory
- `mkfifo` - utility to create “named” pipes
- Can be opened as regular files
- Pipes are unidirectional
  - Can only read/write at any point of time

- So why do we need pipes?

Named Pipes

- One way to transfer data between processes
- Standard use between two processes
  - Two pipes
  - One pipe used to write from 1st process and read from 2nd, and vice-versa



(Nameless) Pipes

- Connect the standard output (`stdout`) of one command to the standard input (`stdin`) of another command.
- Much more common use!
- used with the “`|`” operator in command line
  - `ls –l | grep ‘Nov 19’`