Virtualization and IPC

OPERATING SYSTEMS COURSE
THE HEBREW UNIVERSITY
SPRING 2023

Overview

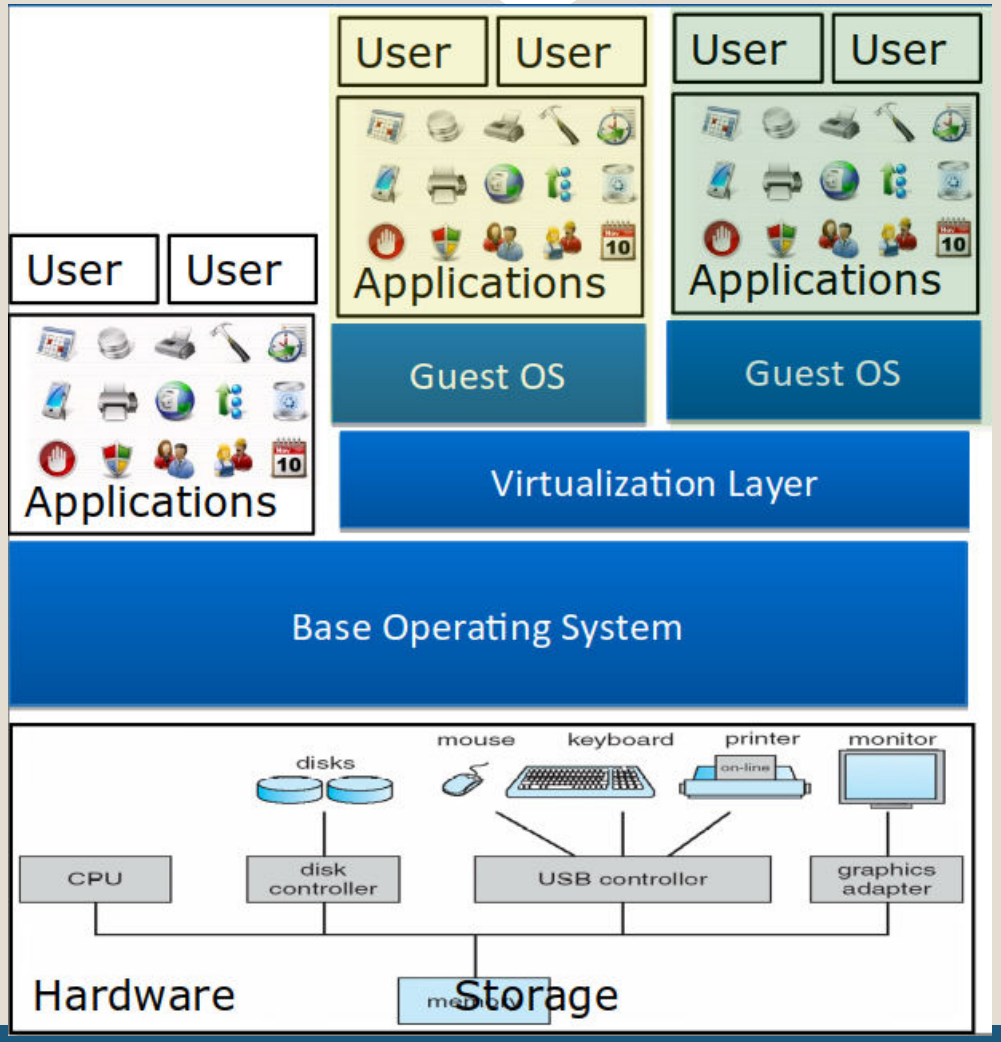
- Virtualization - Recap
- IPC – Inter-processing communication
  - Pipe
  - Files
  - Shared Memory
- Ex5

Virtualization

Recap - Virtualization

- **Virtualization** - creating a virtual version of:
  - Virtual computer hardware platforms
  - Storage devices
  - Computer network resources
  - Etc.
- A **virtual machine** (VM) is an emulation of computer system.
  - There can be many VMs on each physical computer system
  - Each VM may have its own “virtual” resources and operating system.

Virtualization



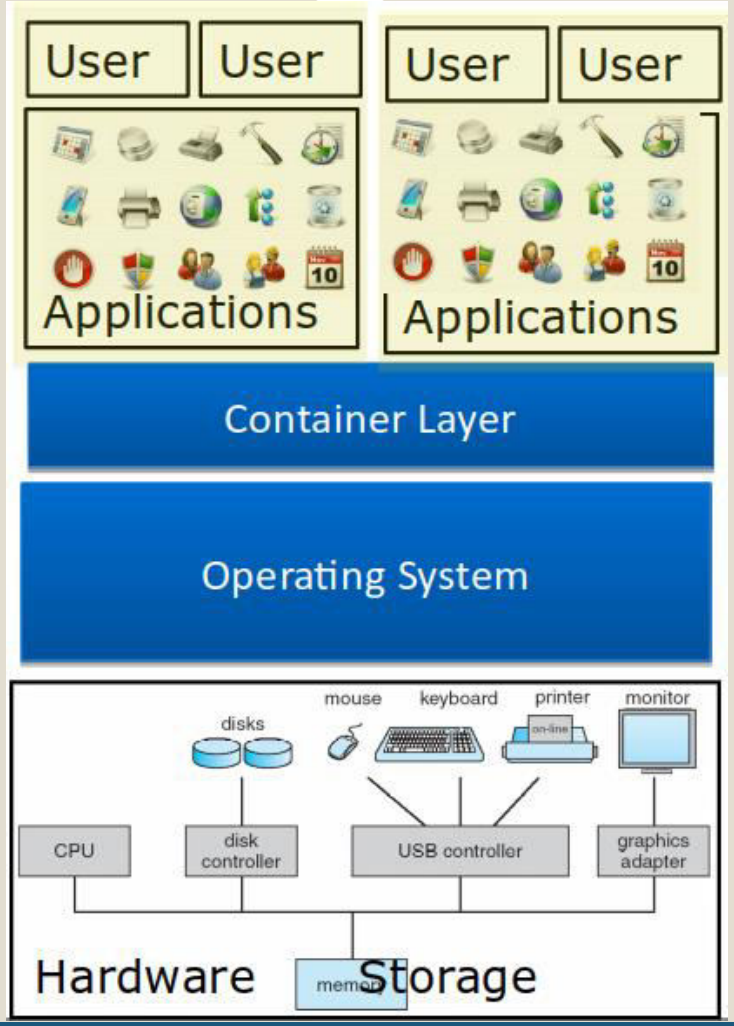
Recap – Virtual Machines

- Running several VMs, each with its own OS, consumes many system resources of the physical machine – disk space, RAM, CPU cycles…
- That's a lot of overhead
- Containers come to the rescue!

Recap - Containers

- **Container** - an OS-level virtualization
- A program running inside of a container can only see the container's contents and devices assigned to the container.
- Multiple containers can run on the same machine and share the OS kernel with other containers.

Virtualization



IPC – Inter-Process Communication

Communication between processes

- Communication between processes is not straightforward
- Each process only sees its own allocated memory
- The OS provides mechanisms to allow processes to communicate with each other

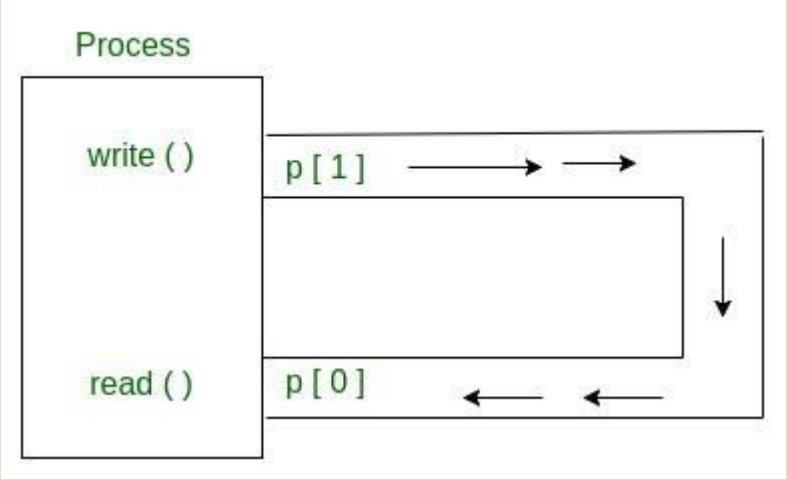
Approaches for IPC

- We will review several different ways for IPC:
  - Signals √
  - Sockets √
  - Pipe
  - File system
  - Shared Memory
- But many others exist

Pipe

Pipes

- A pipe is a one-way communication channel between two processes
- The standard output from one process becomes the standard input of the other process



Fork - Reminder

- fork() – **create a new process**
  - pid = fork();
  - The *fork*() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except some process' system properties
  - Pay attention: two different processes will execute the next line.
  - The return value depend on the process
    - return value == 0 … Child
    - return value > 0 … parent (returned value is the child's *pid*)

Pipe

- pipe – create an interprocess communication channel

```
err = pipe(int fd[2]);
```

  - The *pipe*() function shall create a pipe and place two file descriptors, one each into the arguments `fd[0]` and `fd[1]`, that refer to the open file descriptors for the read and write ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe*() call.
  - A read on the file descriptor `fd[0]` shall access data written to the file descriptor `fd[1]` on a first-in-first-out basis.
  - The details and utilization of this call will be explained later.

dup

- dup, dup2– duplicate an open file descriptor

```
fd_new = dup(int fd);
int dup2(int oldfd, int newfd);
```

  - dup() uses the lowest-numbered unused descriptor for the new descriptor.
  - dup2() makes newfd be the copy of oldfd, closing newfd first if necessary
  - After a successful return from one of these system calls, the old and new file descriptors may be used interchangeably.
  - On success, these system calls return the new descriptor. On error, -1 is returned, and errno is set appropriately.

Pipes Example

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main() {
    int pipefd2[2];
    pipe(pipefd2);
    if (fork() == 0) {
        dup2(pipefd2[1], STDOUT_FILENO);
        close(pipefd2[0]);
        close(pipefd2[1]);
        execl("/bin/ls", "ls", NULL);
        exit(EXIT_FAILURE);
    }
    if (fork() == 0) {
        dup2(pipefd2[0], STDIN_FILENO);
        close(pipefd2[0]);
        close(pipefd2[1]);
        execl("/usr/bin/file", "file", "-f-", NULL);
        exit(EXIT_FAILURE);
    }
    close(pipefd2[0]);
    close(pipefd2[1]);
    wait(NULL);
    wait(NULL);
    return 0;
}
```

Files

Files

- Using the file system is one of the simplest ways to allow processes to share data with each other
- A process may open any file it has access to
- Two process may access the same file to communicate with each other

Files – basic commands

- **int open(const char *pathname, int flags);**
  - Opens a file
  - Returns file descriptor (fd), which identifies the file in future operations
  - fd=0 -> standard input, fd=1 -> standard outpout, fd=2 -> standard error
- **ssize_t read(int fd, void *buf, size_t count);**
  - Reads from fd to buf between 1 to count bytes
  - Returns the number of bytes that were read (zero for eof)
- **ssize_t write(int fd, const void *buf, size_t count);**
  - Writes from buf to fd between 1 to count bytes
  - Returns the number of bytes that were written

File Access Calls - lseek

lseek – move the read/write file offset

```
where = lseek(int fd, off_t offset, int whence);
```

- The *lseek*() function shall set the file offset for the open associated with the file descriptor `fd`, as follows:
  - If `whence` is `SEEK_SET`, the file offset shall be set to `offset` bytes.
  - If `whence` is `SEEK_CUR`, the file offset shall be set to its current location plus `offset`.
  - If `whence` is `SEEK_END`, the file offset shall be set to the size of the file plus `offset`.
- The *lseek*() function shall allow the file offset to be set beyond the end of the existing data in the file creating a gap (implements *sparse file*).
- Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, shall be returned.
- An interesting use is: `where = lseek(fd, 0, SEEK_CUR);` will deliver the "current position" in the file.

Shared Memory

Shared memory

- Processes may share a memory segment between them to communicate
- The steps for using a shared memory segment:
  1. Generating a unique key
  2. Allocating the shared memory segment
  3. Attaching to the shared memory segment
  4. Using the shared memory segment
  5. Detaching from the shared memory segment
  6. Destroying the shared memory segment

ftok – Generate a unique key

- key_t ftok(const char **pathname*, int *proj_id*)
  - Used to generate a unique key to be used to identify the shared memory segment
  - *pathname* should refer to an existing, accessible file
  - The same key_t is returned when calling with the same *pathname* and *proj_id*
  - Returns -1 on failure

shmget – Allocate a shared memory segment

- int shmget(key_t *key*, size_t *size*, int *shmflg*)
  - Returns the identifier of the shared memory segment associated with the value of *key*
  - *key* contains the value returned from ftok
  - If no shared memory segment is associated with *key*, shmget creates a new segment
  - *size* should be a multiple of PAGE_SIZE
  - *shmflg* should be specified with IPC_CREAT to create a new shared memory segment
  - Returns -1 on failure

shmat – Attach to a shared memory segment

- void *shmat(int *shmid*, const void **shmaddr*, int *shmflg*)
  - Returns the shared memory segment's start address
  - *shmid* is the shared memory segment identifier returned from shmget
  - *shmaddr* is the requeseted address to attcach the segment. If *shmaddr* is NULL, the OS will automatically choose the address for us
  - Returns -1 on failure

shmdt – Detach a shared memory segment

- int shmdt(const void **shmaddr*)
  - Detaches the shared memory segment located at the address specified by *shmaddr*
  - Returns 0 on success, -1 on failure

shmctl – Destroy a shared memory segment

- int shmctl(int *shmid*, int *cmd*, struct shmid_ds * *buf*)
  - Used to destroy the shared memory segment specified by *shmid*
  - *cmd* should be set to IPC_RMID, and *buf* should be set to NULL
  - The segment will be destroyed only after the last process detaches it
  - You **must** call this function to destroy the segment, as it will remain in memory even after all processes detach from it

Shared memory example - Writer

From: https://www.geeksforgeeks.org/ipc-shared-memory/

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    cout<<"Write Data : ";
    gets(str);

    printf("Data written in memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    return 0;
}
```

Shared memory example - Reader

From: https://www.geeksforgeeks.org/ipc-shared-memory/

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Data read from memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    // destroy the shared memory
    shmctl(shmid,IPC_RMID,NULL);

    return 0;
}
```

Ex5

Ex5

- In Ex5, you will create a process called receiver, which will try to communicate with other processes called senders
- The senders may run:
  - Directly on the machine
  - Inside a VM
  - Inside a container
- The communication between the receiver and senders is handled with:
  - Sockets
  - File system
  - Shared Memory