Ex. 1 — Revisions
    Stack:
    Heap
    Key Difference
Ex. 2 — Personal research
    Power on operation
    BIOS
    Hybrid kernels
    Exokernels
Ex. 3 — Course application
    Kernel instructions
    CPU thread
Ex. 4 — Simple problem
Ex. 5 — Command lines on a Unix system

Ex. 1 — Revisions

Stack:

A stack is a special area of computer's memory which stores temporary variables created by a function. In stack, variables are declared, stored and initialized during runtime.

It is a temporary storage memory. When the computing task is complete, the memory of the variable will be automatically erased. The stack section mostly contains methods, local variable, and reference variables.

Heap

The heap is a memory used by programming languages to store global variables. By default, all global variable are stored in heap memory space. It supports Dynamic memory allocation.

The heap is not managed automatically for you and is not as tightly managed by the CPU. It is more like a free-floating region of memory.

Key Difference

- Stack is a linear data structure whereas Heap is a hierarchical data structure.
- Stack memory will never become fragmented whereas Heap memory can become fragmented as blocks of memory are first allocated and then freed.
- Stack accesses local variables only while Heap allows you to access variables globally.
- Stack variables can't be resized whereas Heap variables can be resized.
- Stack memory is allocated in a contiguous block whereas Heap memory is allocated in any random order.
- Stack doesn't require to de-allocate variables whereas in Heap de-allocation is needed.
- Stack allocation and deallocation are done by compiler instructions whereas Heap allocation and deallocation is done by the programmer.

Ex. 2 — Personal research

Power on operation

The first thing a computer has to do when it's powered on is to start up a special program called an **operation system**.

BIOS

The BIOS chip tells it to look in a fixed place, usually on the lowest-numbered hard disk (the boot disk) for a special program called a boot loader (under Linux the boot loader is called Grub or LILO). The boot loader is pulled into memory and started. The boot loader's job is to start the real operating system.

Hybrid kernels

The traditional kernel categories are monolithic kernels and microkernels (with nanokernels and exokernels seen as more extreme versions of microkernels).

The idea behind a hybrid kernel is to have a kernel structure similar to that of a microkernel, but to implement that structure in the manner of a monolithic kernel. In contrast to a microkernel, all (or nearly all) operating system services in a hybrid kernel are still in kernel space. There are none of the reliability benefits of having services in user space, as with a microkernel. However, just as with an ordinary monolithic kernel, there is none of the performance overhead for message passing and context switching between kernel and user mode that normally comes with a microkernel.

Exokernels

The idea behind exokernels is to force as few abstractions as possible on application developers, enabling them to make as many decisions as possible about hardware abstractions.[2] Exokernels are tiny, since functionality is limited to ensuring protection and multiplexing of resources, which is considerably simpler than conventional microkernels' implementation of message passing and monolithic kernels' implementation of high-level abstractions.

Ex. 3 — Course application

Kernel instructions

a, c, d should only be allowed in kernel mode. Kernel mode, also known as system mode, is one of the central processing unit (CPU) operating modes. **While processes run in kernel mode, they have unrestricted access to the hardware.** If instructions such as disabling interrupts, setting clock, and changing memory map are unrestricted to all the users, computers may continuously crash. Therefore, these three must be run in kernel mode.

CPU thread

- P0, P1 -> CPU1, P2 -> CPU2: 20ms
- P0, P2 -> CPU1, P1 -> CPU2: 25ms
- P1, P2 -> CPU1, P0 -> CPU2: 30ms
- P0, P1, P2 -> CPU1: 35ms

Ex. 4 — Simple problem

- $25 \times 80 = 2000\ Byte \approx 2KB$, the cost is roughly $10

- $1024 \times 768 \times 3 = 2,359,296 Byte \approx 2,304 KB$, the cost is roughly $11,520

Now the price of RAM is roughly $10/GB

Ex. 5 — Command lines on a Unix system

```
# Create a new user
user add -m bill

# List all the currently running processes
ps -alE

# Display the characteristics of the CPU and the available memory
cat /proc/cpuinfo
cat /proc/meminfo

# Redirect some random output into two different files
head -n 1 /dev/random | tee file1.txt > file2.txt

# Concatenate the two previous files
cat file1.txt file2.txt > file3.txt

# Read the content of the resulting file as hexdecimal values (in other words
find a command to read a file as hexadecimal values)
hexdump file1.txt

# Use a single command to find all the files in /usr/src with the word semaphore
in their name and containing the word ddekit_sem_down
grep -Rl "ddekit_sem_down" /usr/src | grep "semaphore"
```