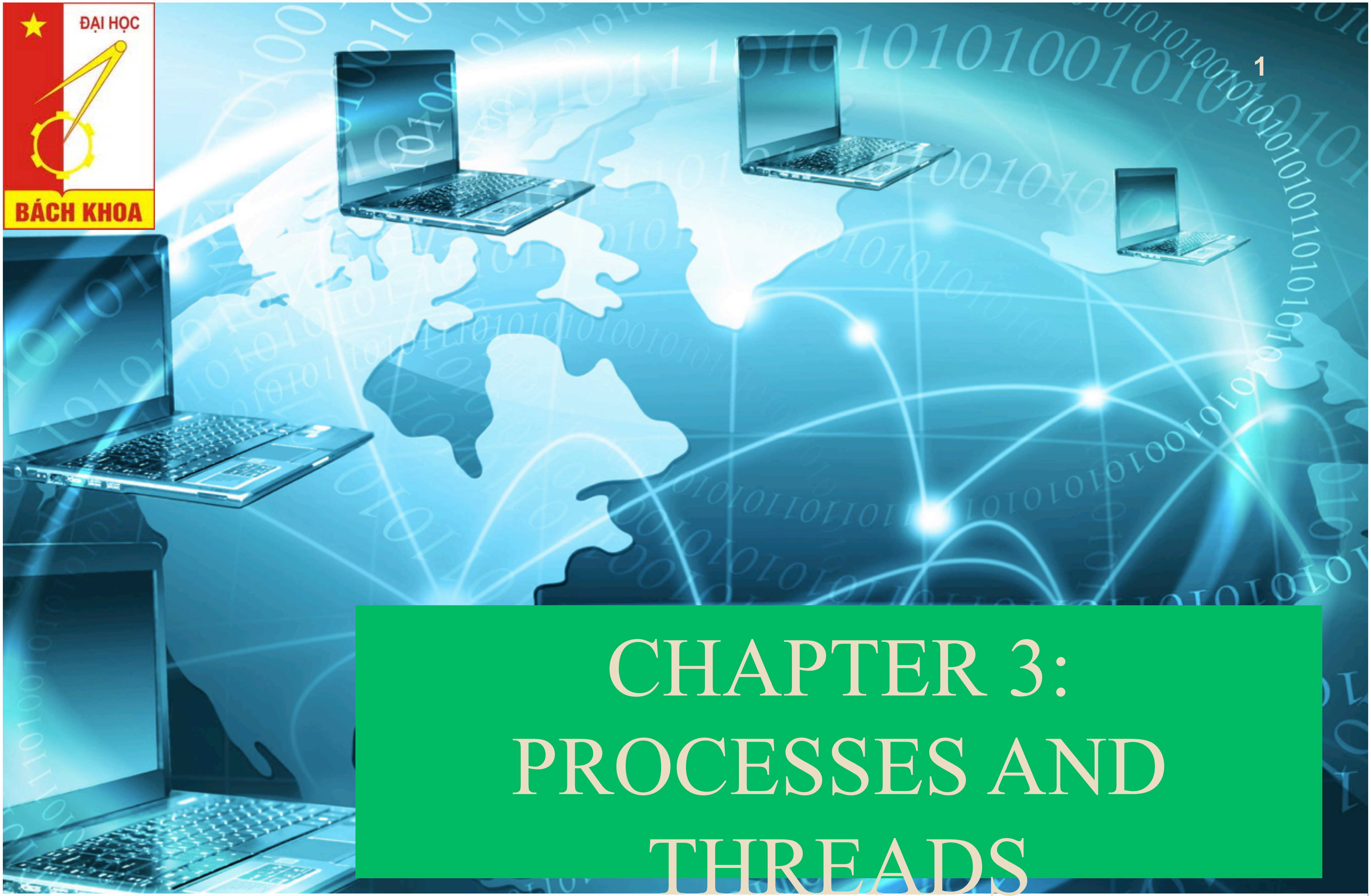
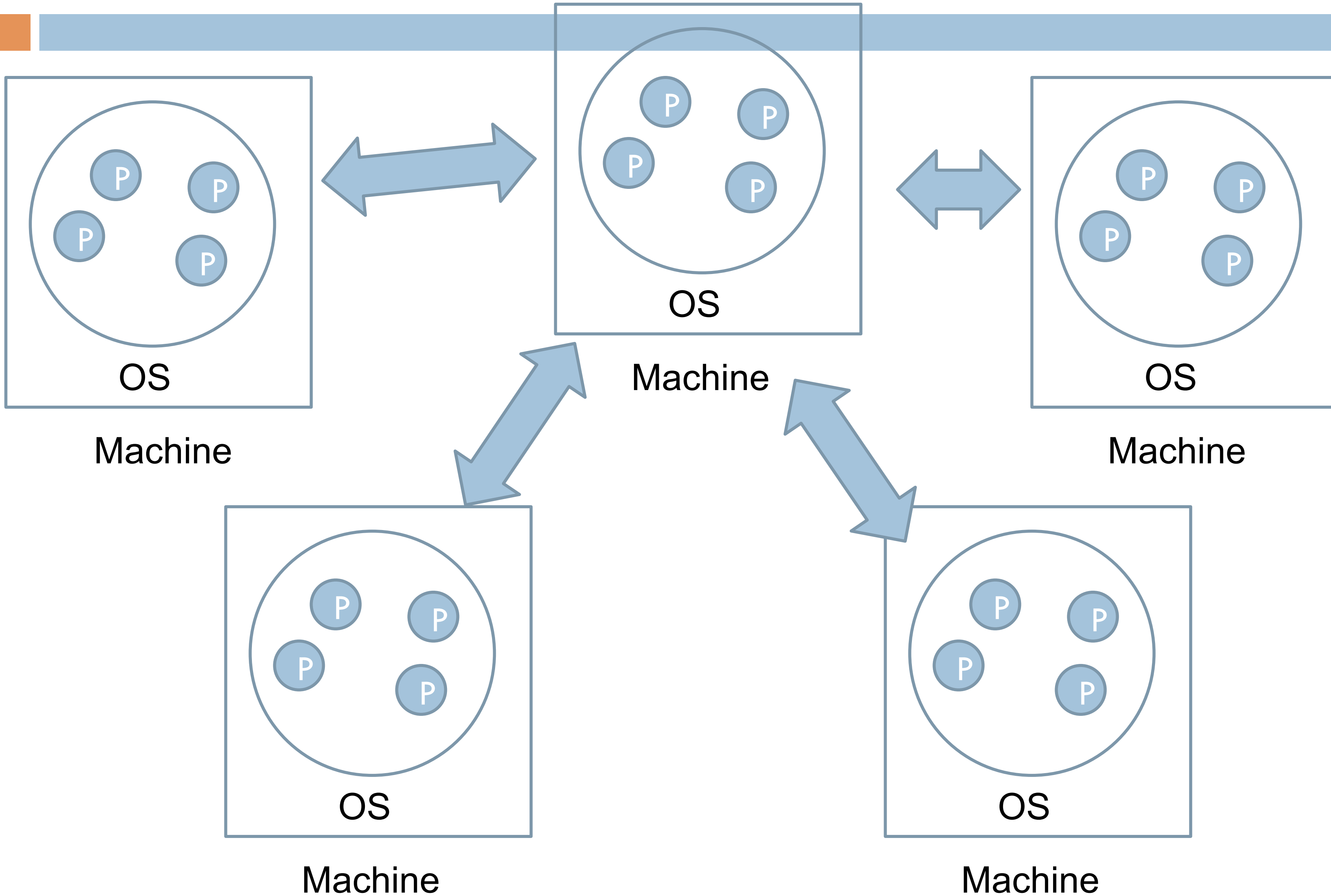


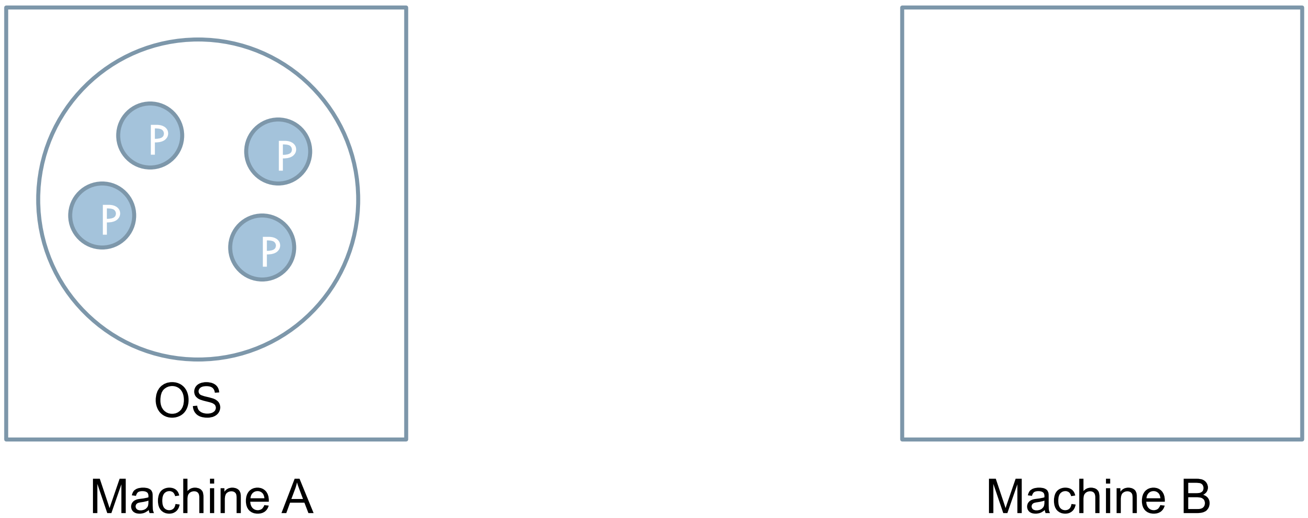
CHAPTER 3: PROCESSES AND THREADS

Dr. Trần Hải Anh

Role of OS in process management



Virtualization



Outline

1. Process and Thread
2. Virtualization
3. Clients
4. Servers
5. Code migration

1. Process and Thread

1.1. Introduction

1.2. Threads in centralized systems

1.3. Threads in distributed systems

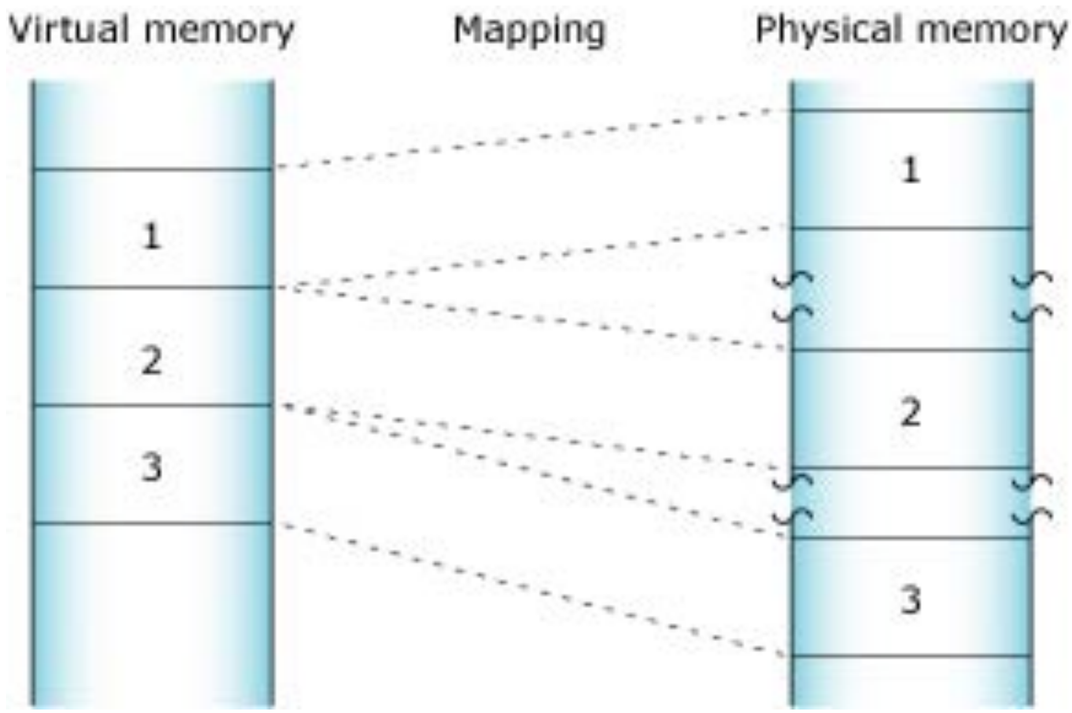
1.1. Introduction

- Process
  - A program in execution
  - Resources
    - Execution environment, memory space, registers, CPU...
    - Virtual processors
    - Virtual memory
  - Concurrency transparency
  - Creating a process:
    - Create a complete independent address space
    - Allocation = initializing memory segments by zeroing a data segment, copying the associated program into a text segment, setup a stack for temporary data
  - Switching the CPU between processes: Saving the CPU context + modify registers of MMU, ...

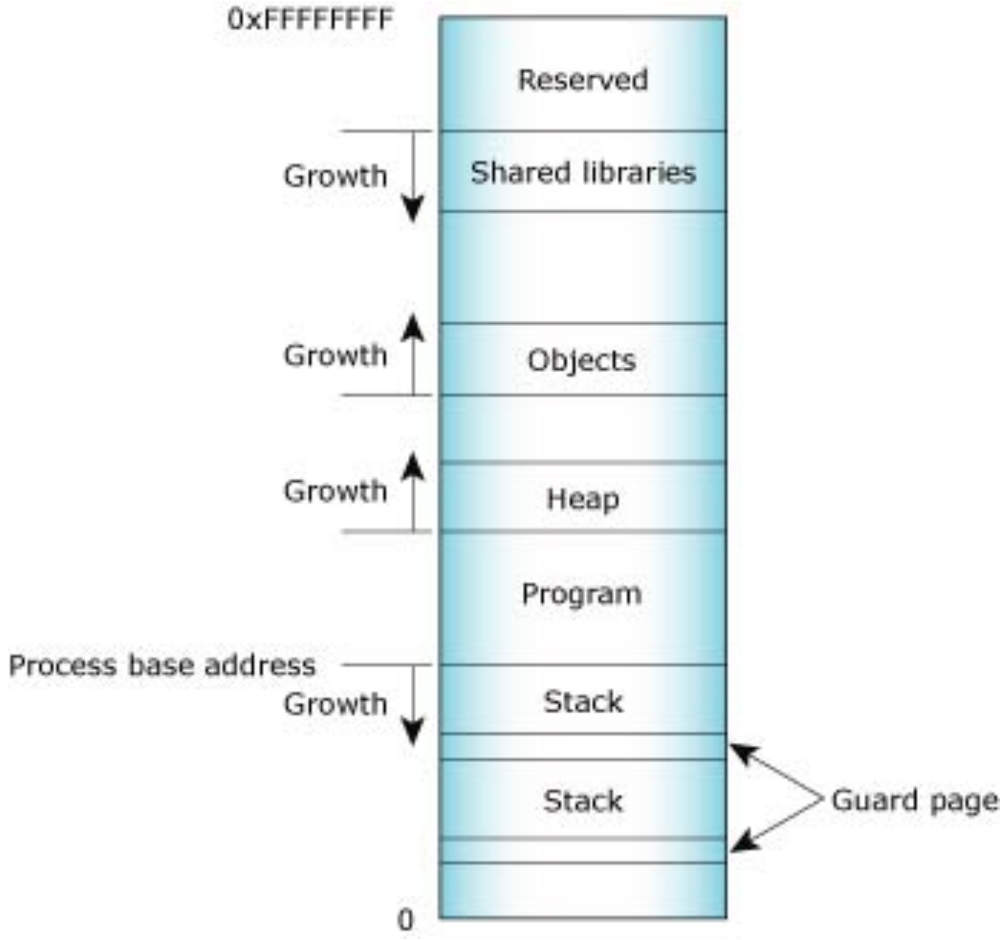
Thread

- A thread executes its own piece of code, independently from other threads.
- Process has several threads → multithreaded process
- Threads of a process use the process'context together
- Thread context: CPU context with some other info for thread management.
- Exchanging info by using shared variable (mutex variable)
- Protecting data against inappropriate access by threads within a single process is left to application developers.

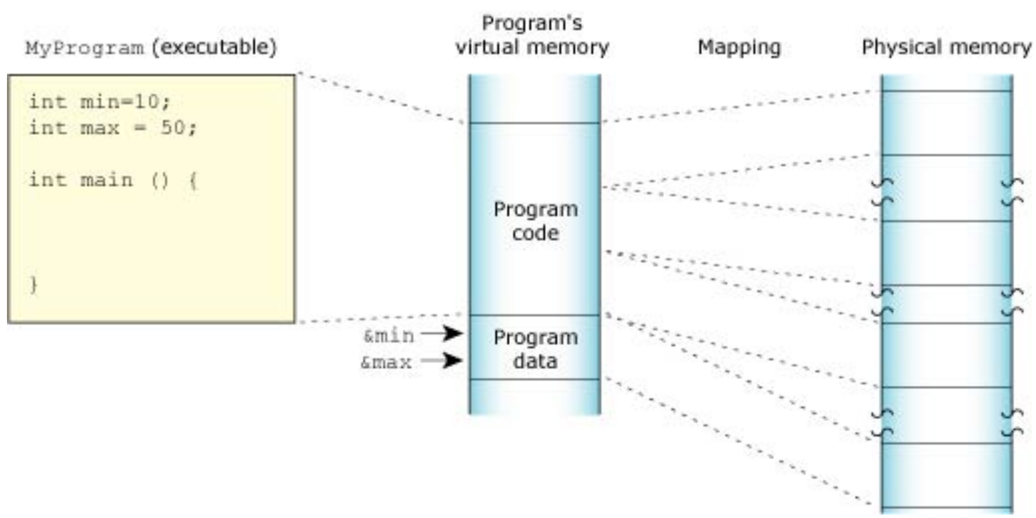
Virtual Memory



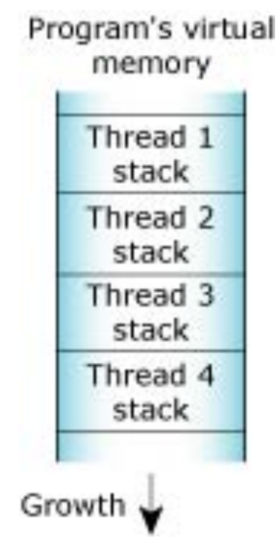
Process Memory layout



Program and Stack memory

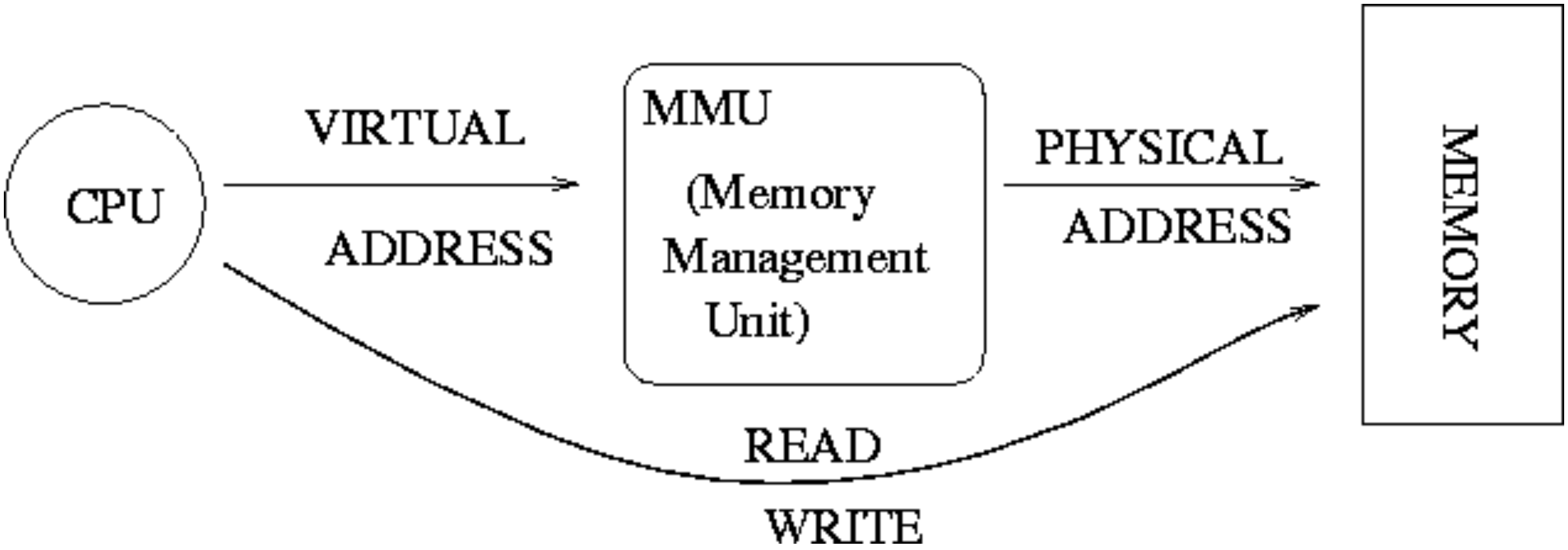


Program memory



Stack memory

Mapping method



1.2. Thread usage in Nondistributed Systems

- Multithreaded program vs multi-processes program
  - Switching context
  - Blocking system calls

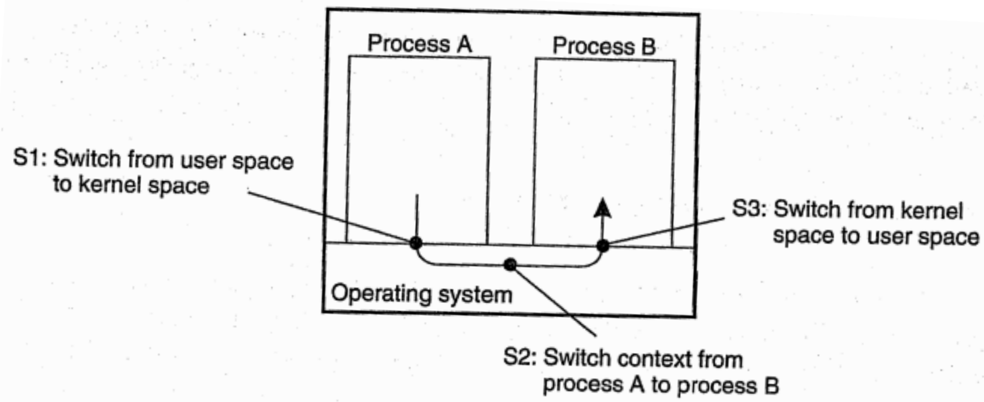


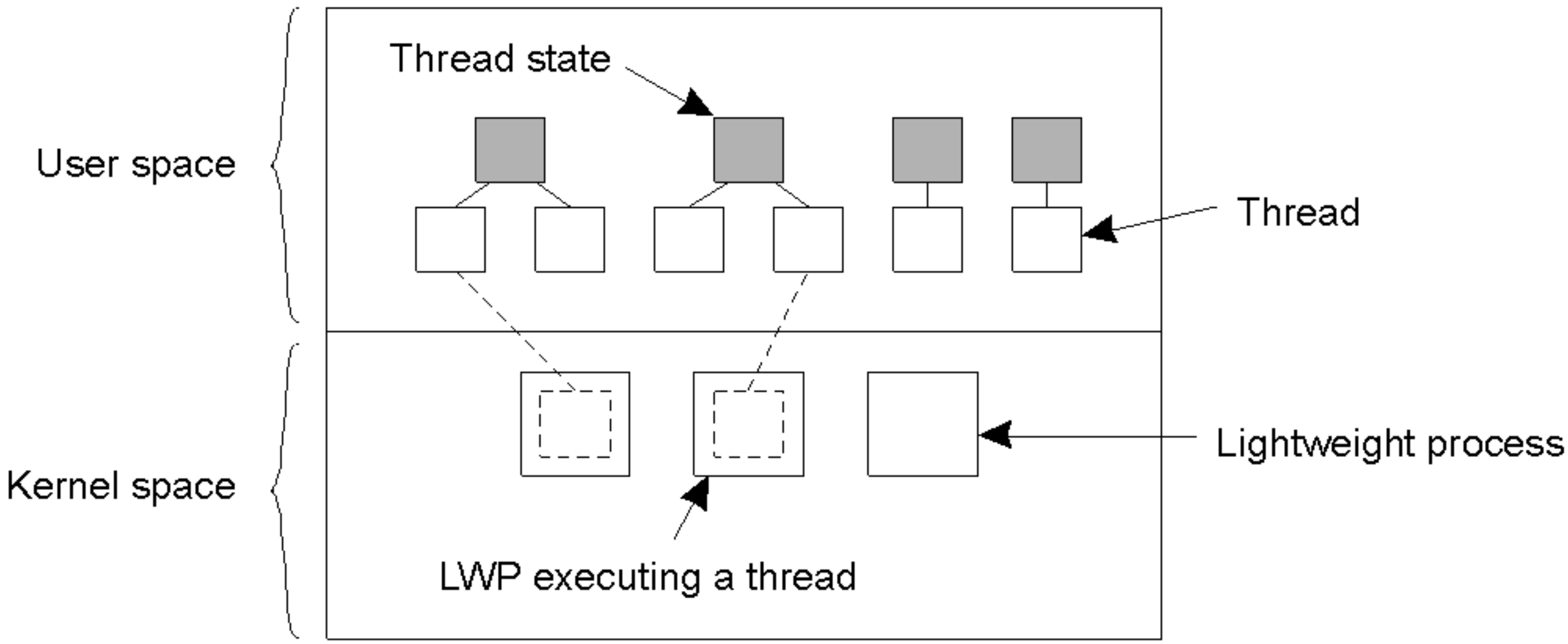
Figure 3-1. Context switching as the result of IPC.

Thread implementation

- Thread package:
  - Creating threads (1)
  - Destroying threads (2)
  - Synchronizing threads (3)
- (1), (2), (3) can be operated in user mode and kernel mode:
  - User mode:
    - Cheap to create and destroy threads
    - Easy to switch thread context
    - Invocation of a blocking system call will block the entire process
  - Kernel mode:

Lightweight processes (LWP)

- Combining kernel-level lightweight processes and user-level threads.



Threads in LINUX

- Threads are constructed with POSIX standard (Portable Operating System Interface for uniX).
- Running in 2 separated spaces:
  - User space: use library *pthread*
  - Kernel: use LWPs
- Mapping 1-1 between 1 thread and 1 LWP
- Linux use clone() to generate a thread, instead of fork().

ID management

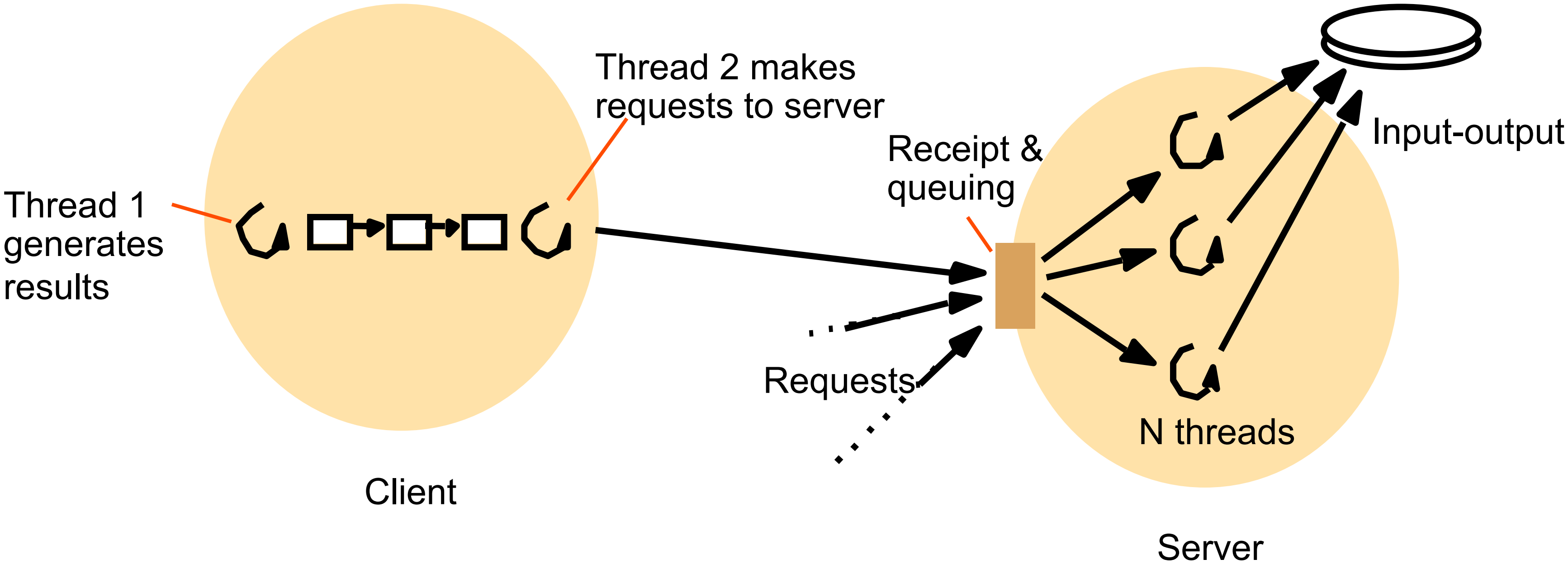
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
  void * function1(void *arg)
{
   pthread_t tid=pthread_self();
   printf("In thread %u and process %u\n",tid,getpid());
}

void * function2(void *arg)
{
   pthread_t tid=pthread_self();
   printf("In thread %u and process %u\n",tid,getpid());
}

int main()
{
   void *status;
   pthread_t tid1,tid2;
   pthread_attr_t attr;

    if(pthread_create(&tid1,NULL,function1,NULL)){
        perror("Failure");
        exit(1);
   }

   if(pthread_create(&tid2,NULL,function2,NULL)){
       perror("Failure");
       exit(2);
   }

   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
   printf("In main thread %u and process %u\n",pthread_self(),getpid());
}
```

```
In thread 3086625680 and process 5480
In thread 3076135824 and process 5480
In main thread 3086628544 and process 5480
```

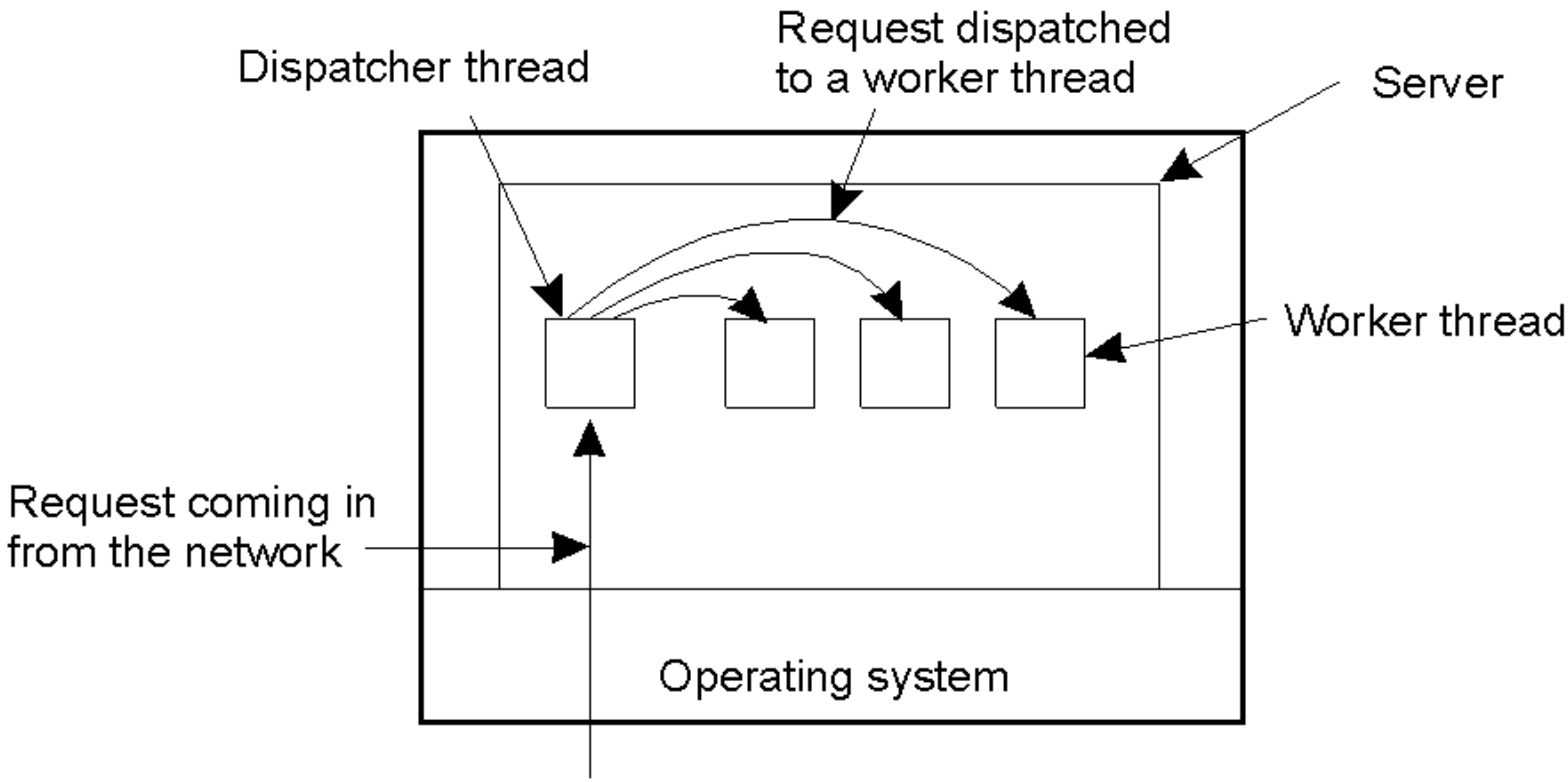
1.3. Threads in Disitrubuted Systems

- Single-threaded server
  - One request at one moment
  - Sequentially
  - Do not guaranty the transparency

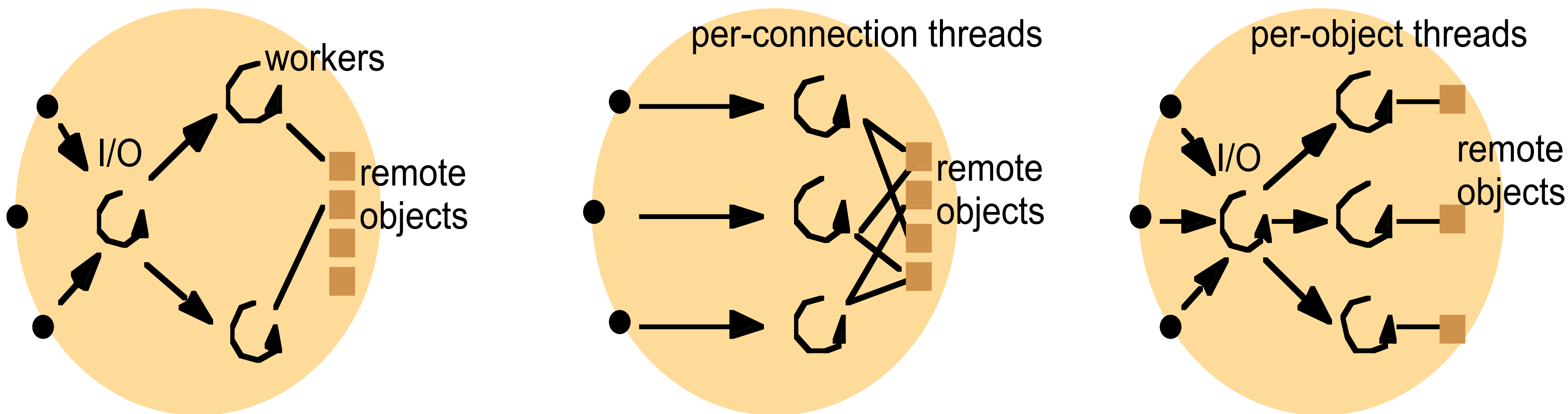
Multithreaded Client and server



Server dispatcher



Multithreaded Server



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

Finite-state machine

- Only one thread
- Non-blocking (asynchronous)
- Record the state of the current request in a table
- Simulating threads and their stacks
- Example: Node.js
  - Asynchronous and Event-driven
  - Single threaded but highly scalable

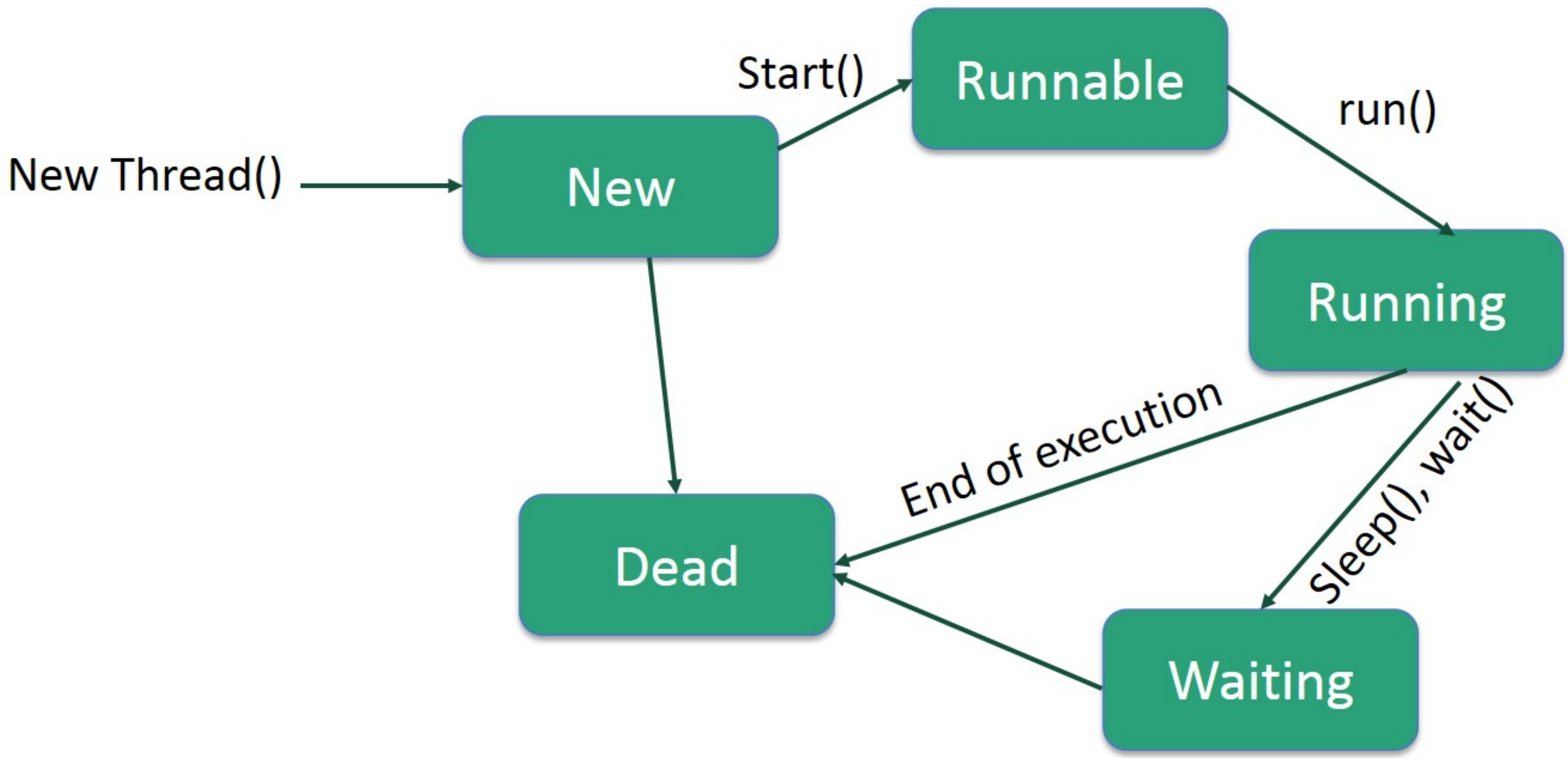
Comparison

Model	Characteristics
Threads	Parallelism, Blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, Non-blocking system calls

Multithreaded Client

- Separate UI and processing task
- Increase the system performance while working with many servers
- E.g. Web browser

Multithreading in Java



Multithreading in Java

- Creating thread in two ways:
  - Inherit the Thread class
  - Implement the interface Runnable
- Methods:
  - getName(): It is used for Obtaining a thread’s name
  - getPriority(): Obtain a thread’s priority
  - isAlive(): Determine if a thread is still running
  - join(): Wait for a thread to terminate
  - run(): Entry point for the thread
  - sleep(): suspend a thread for a period of time
  - start(): start a thread by calling its run() method

Multithreading in Java


```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name){
       threadName = name;
       System.out.println("Creating " +  threadName );
   }
   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
     } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
     }
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}
```

Multithreading in Java


```
public class TestThread {
   public static void main(String args[]) {

      RunnableDemo R1 = new RunnableDemo( "Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo( "Thread-2");
      R2.start();
   }
}
```

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

2. Virtualization

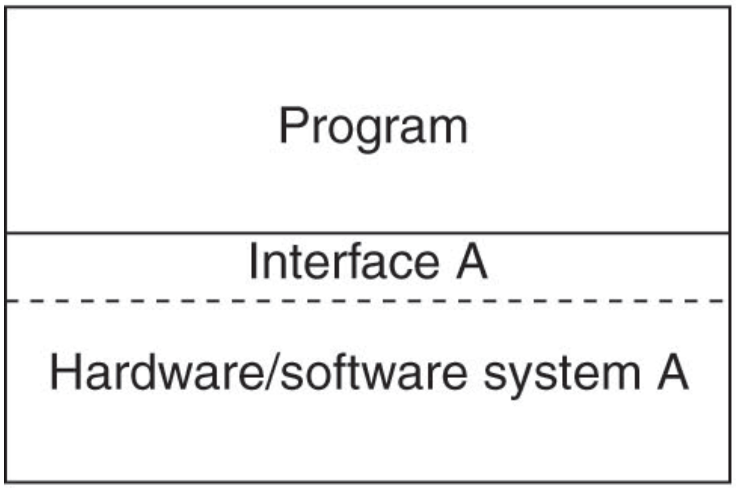
2.1. The Role of Virtualization in Distributed Systems

2.2. Architectures of Virtual Machines

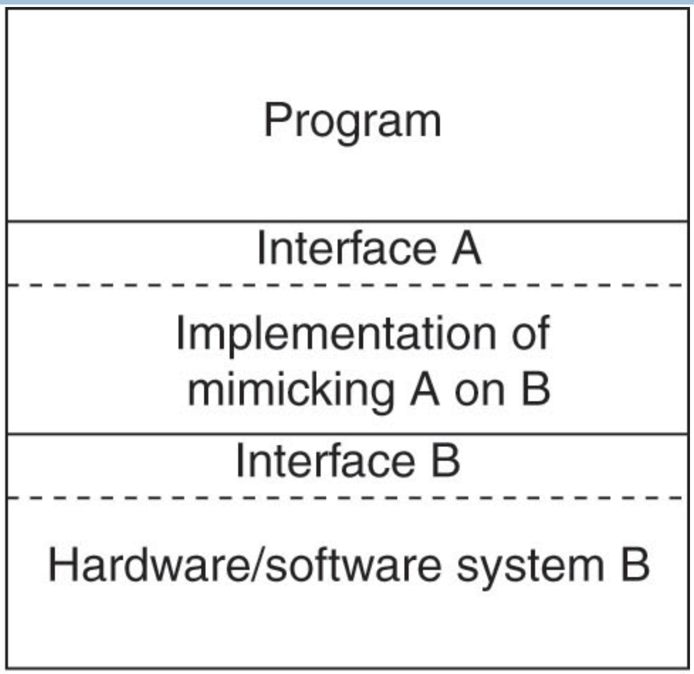
2.1. Role of Virtualization

- In the 1970s, it allows legacy software to run on expensive mainframe hardware
- As hardware became cheaper, virtualization became less of an issue.
- Since the late 1990s, while hardware change reasonably fast, software is much more stable → needs of virtualization
- Diversity of platforms and machines can be reduced by letting each app run on its own virtual machine, which run on a common platform.

How Virtualization works?

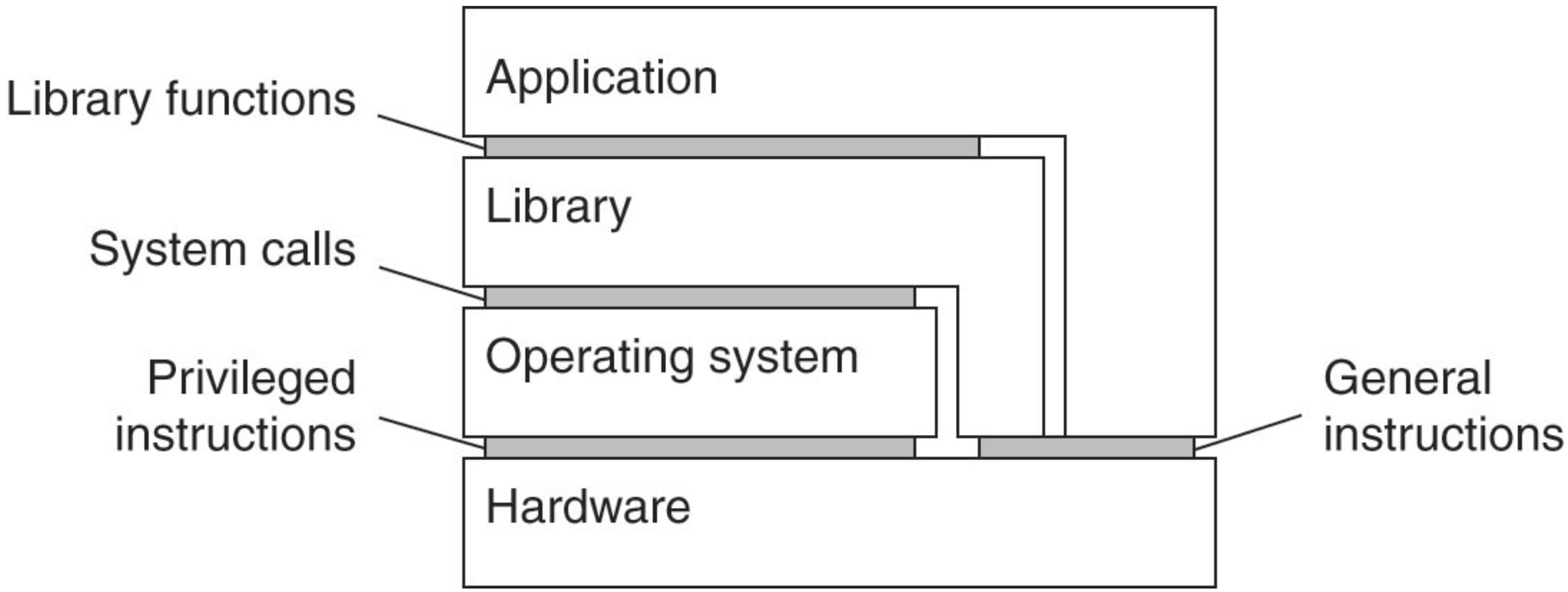


(a)



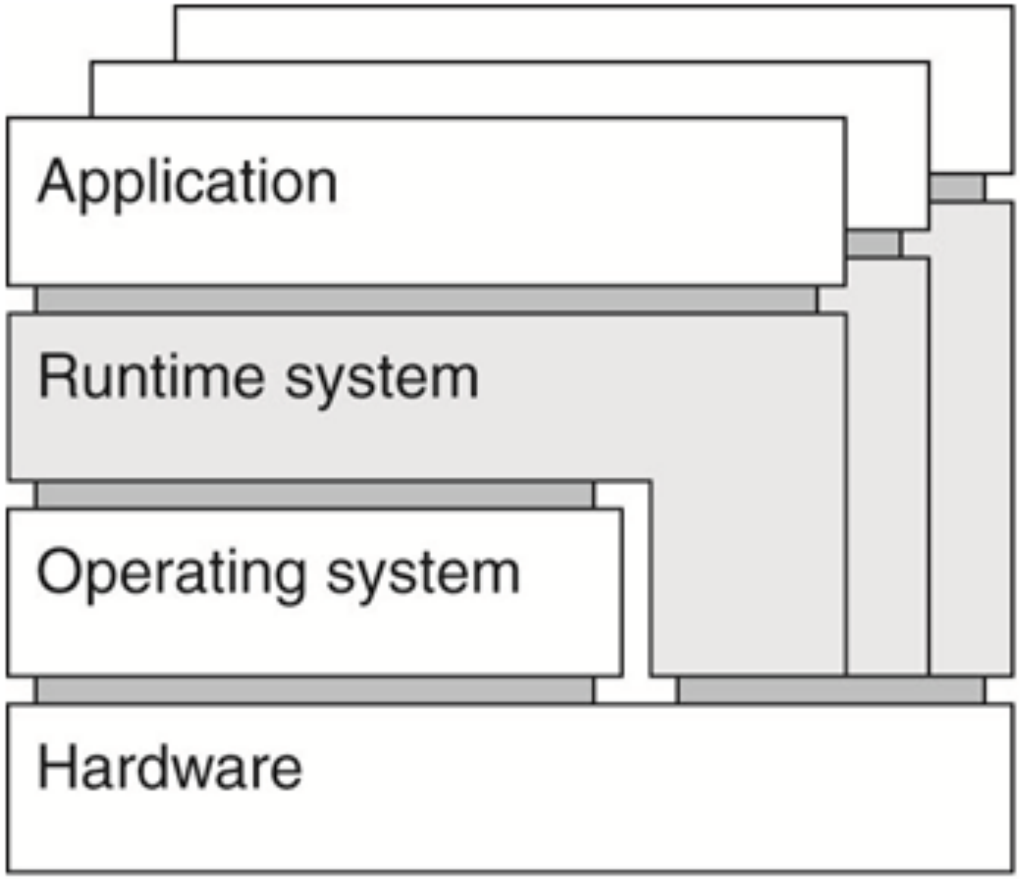
(b)

2.2. Architectures of VMs



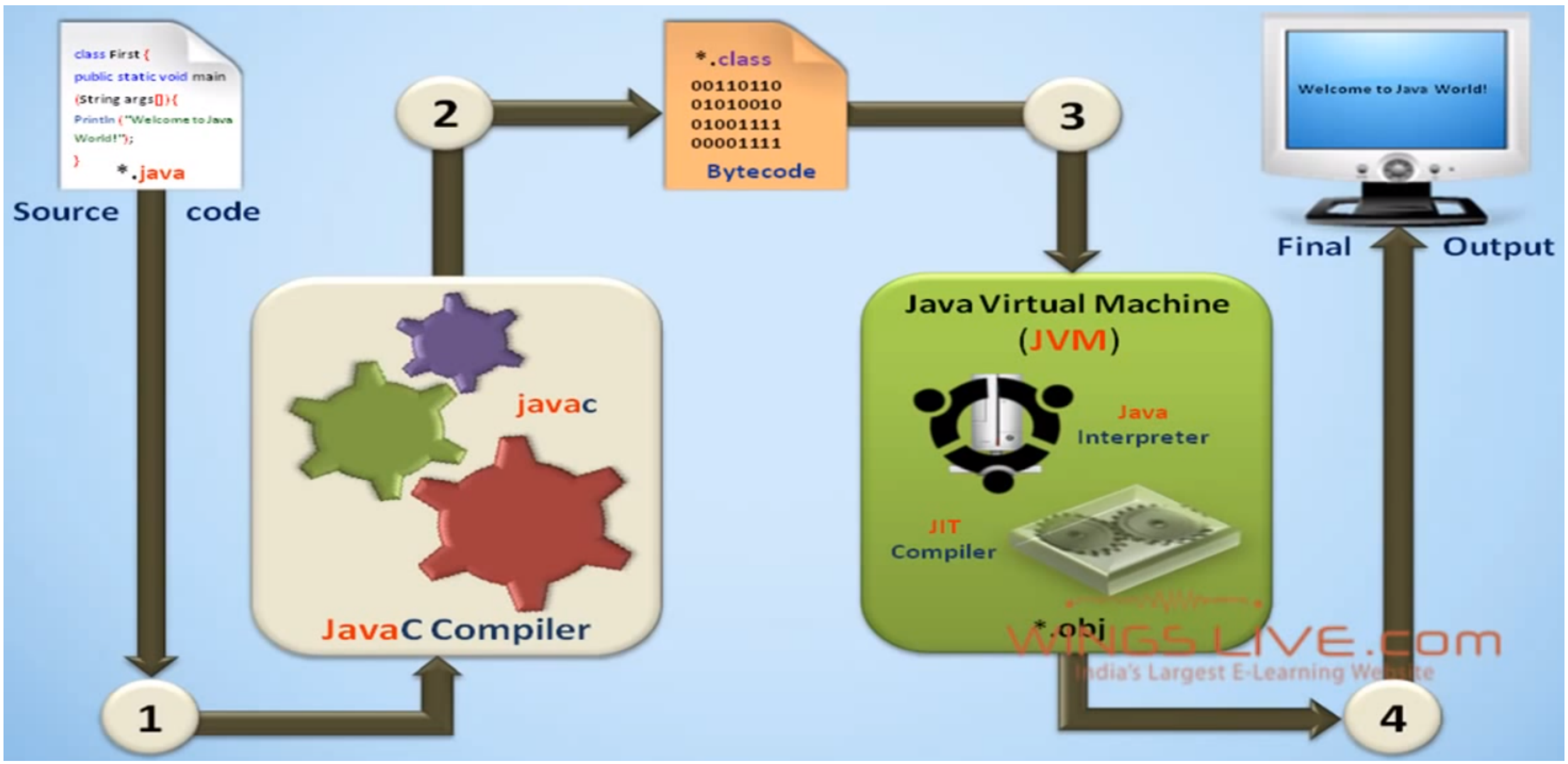
Interfaces offered by computer systems

Process Virtual Machine

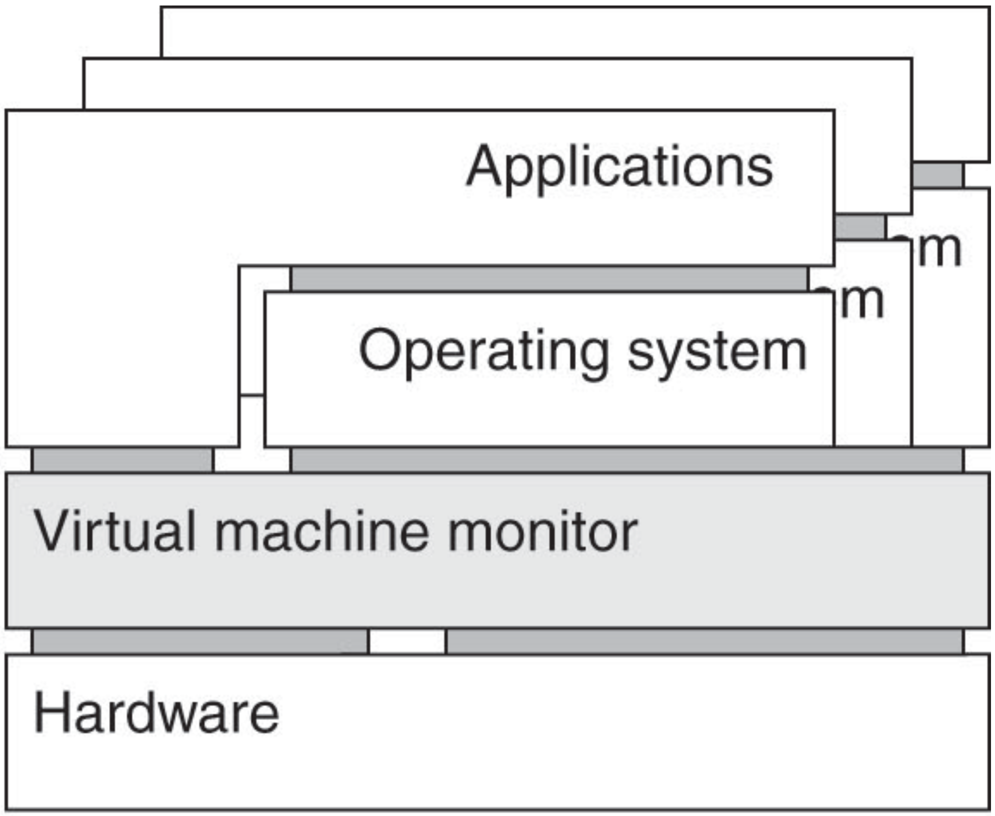


(a)

Java – Platform independent language



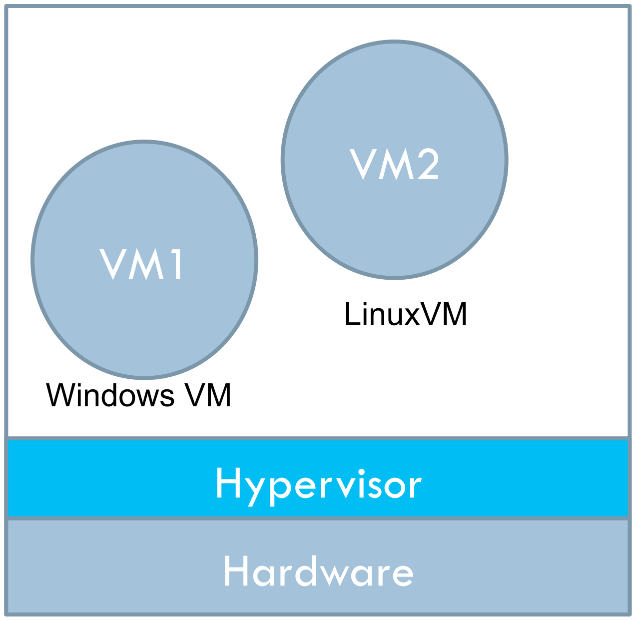
Virtual Machine Monitor



(b)

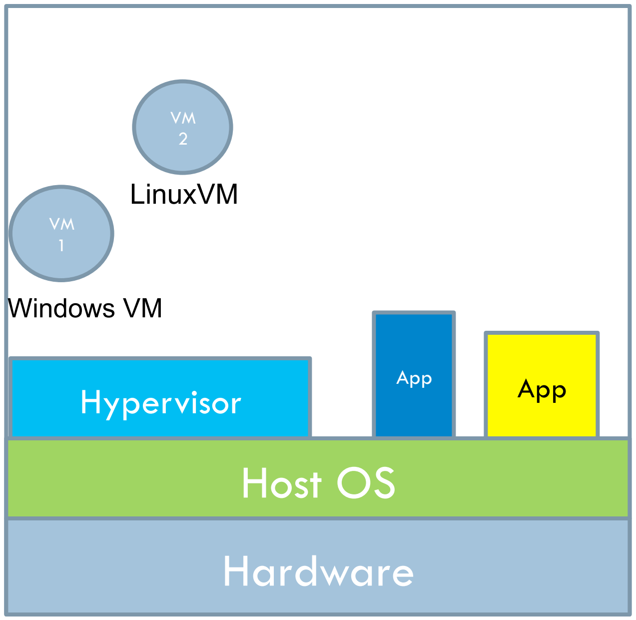
Hypervisor

Type 1: Bare-metal supervisor



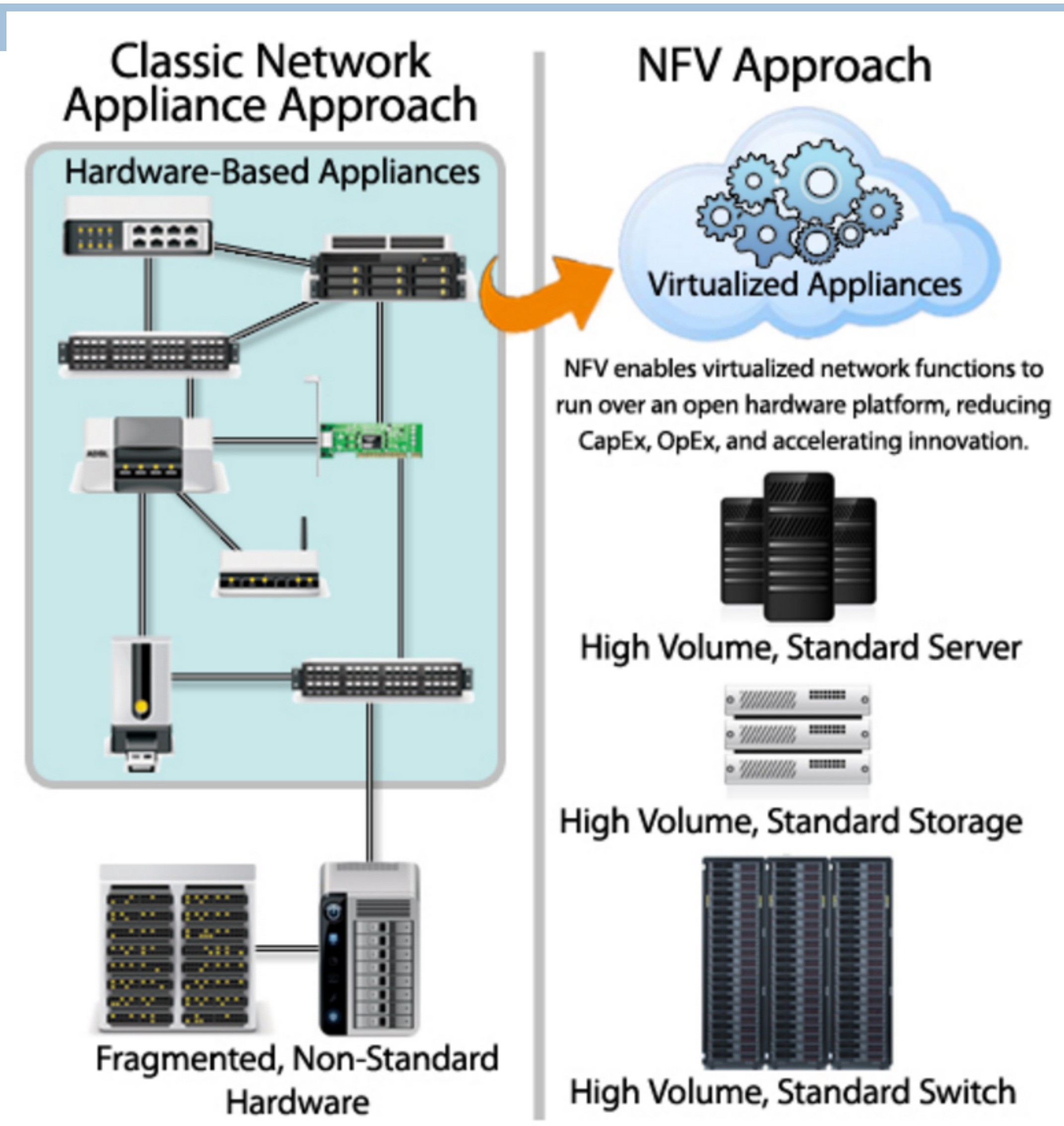
Ex: ESXi (Vmware vSphere)

Type 2

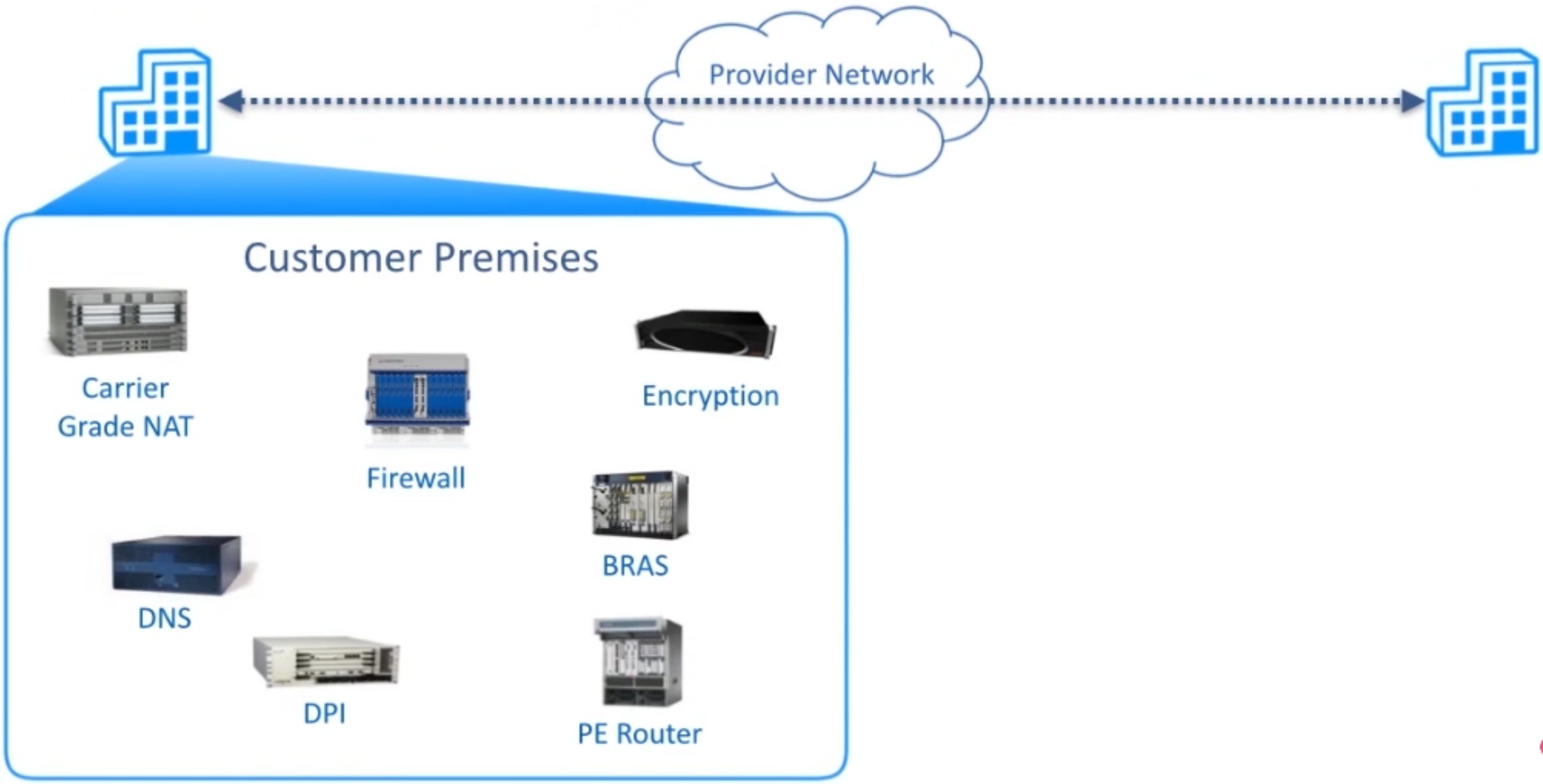


Ex: Vmware, VirtualBox

Network Function Virtualization

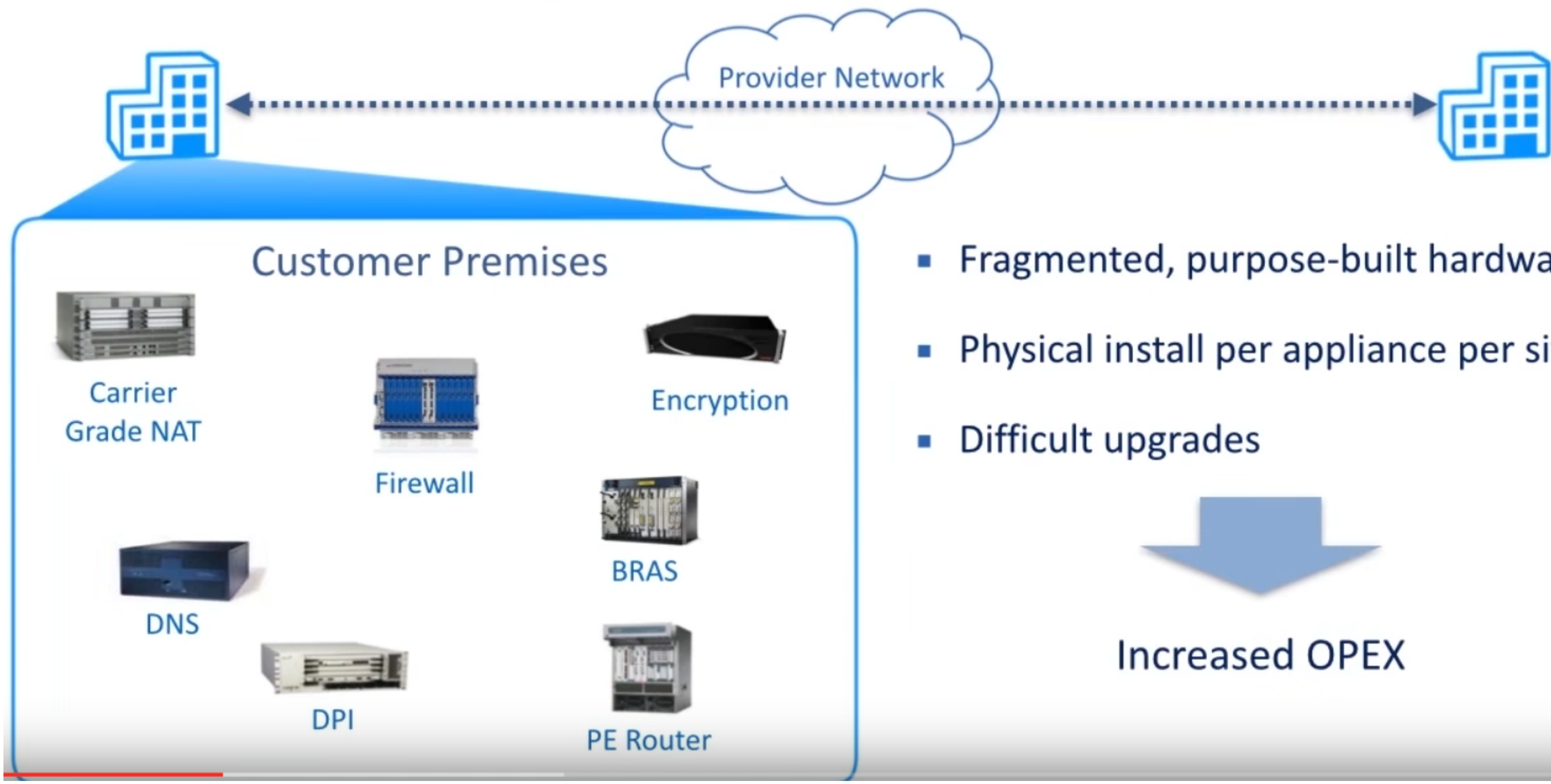


Network Functions



cien

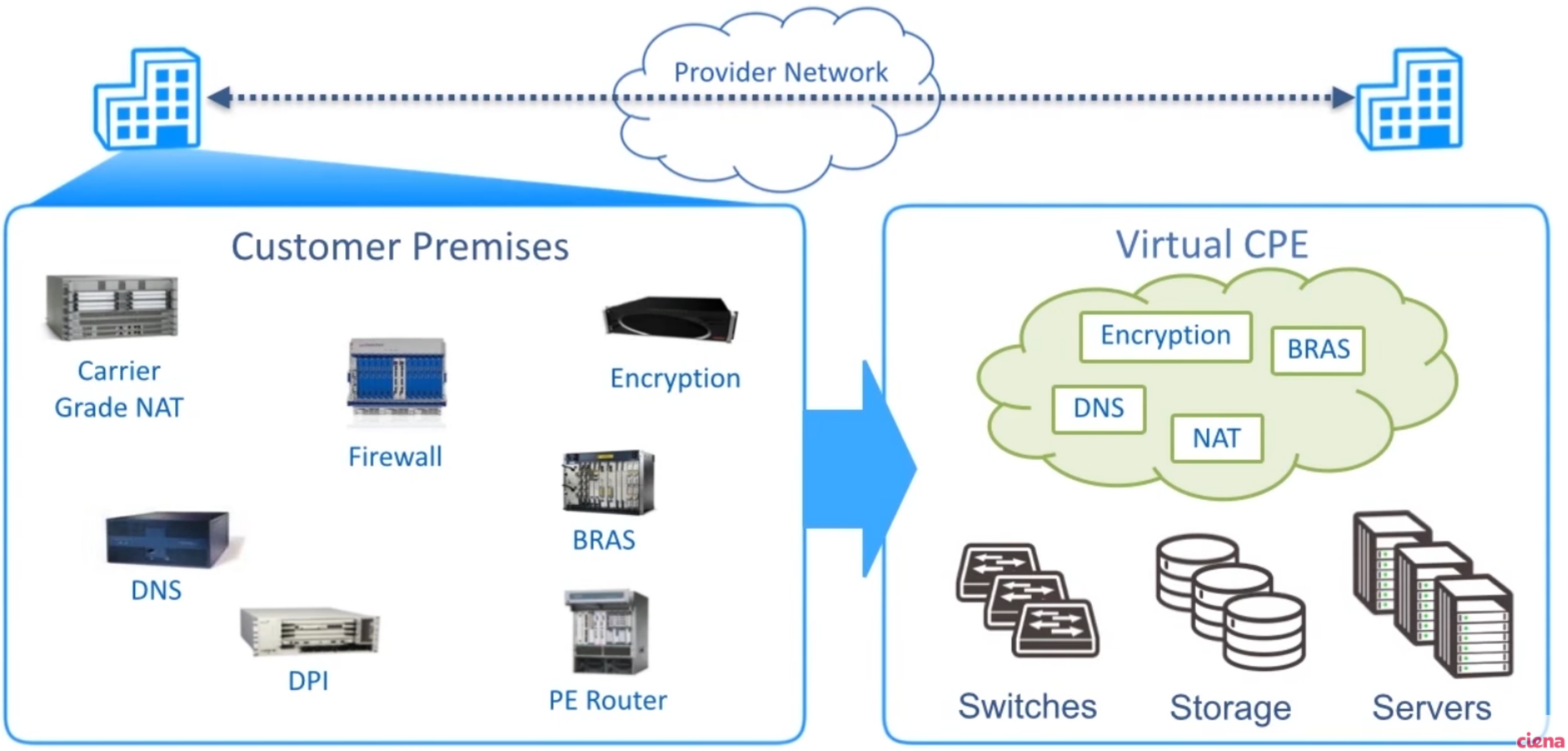
Network Functions



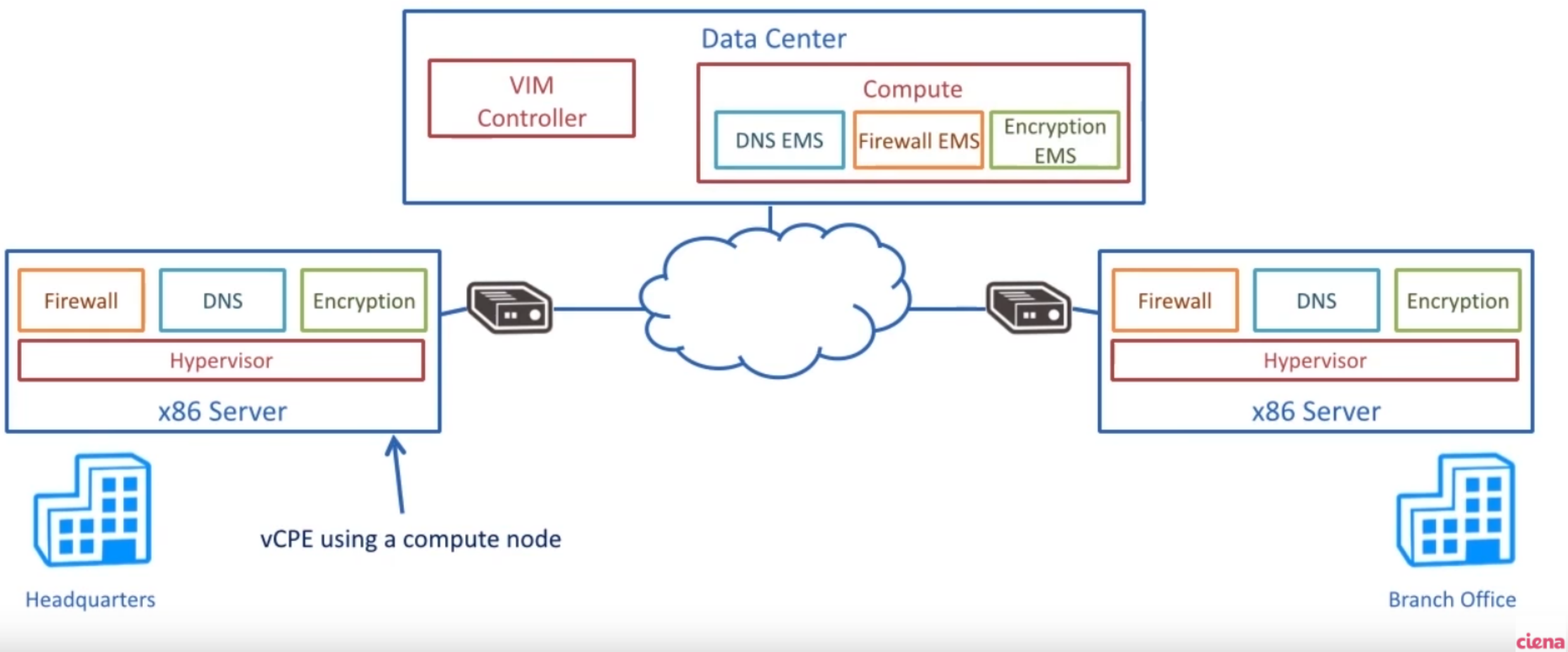
- Fragmented, purpose-built hardware
- Physical install per appliance per site
- Difficult upgrades

Increased OPEX

Network Function Virtualization

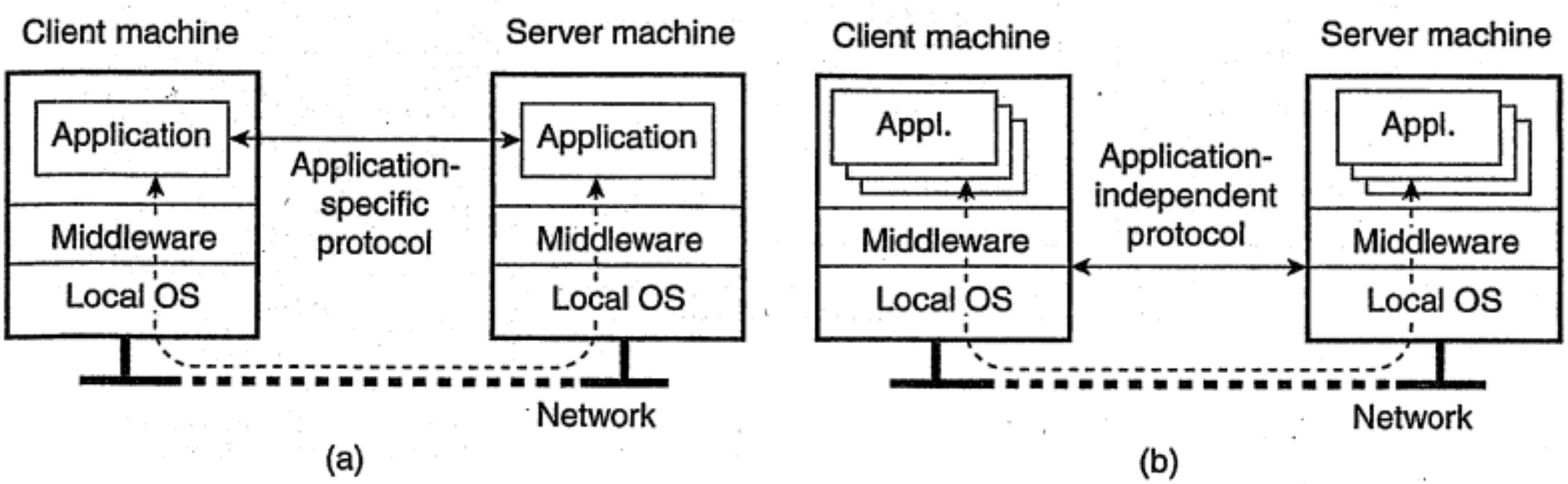


Deploying Network Functions



3. Clients

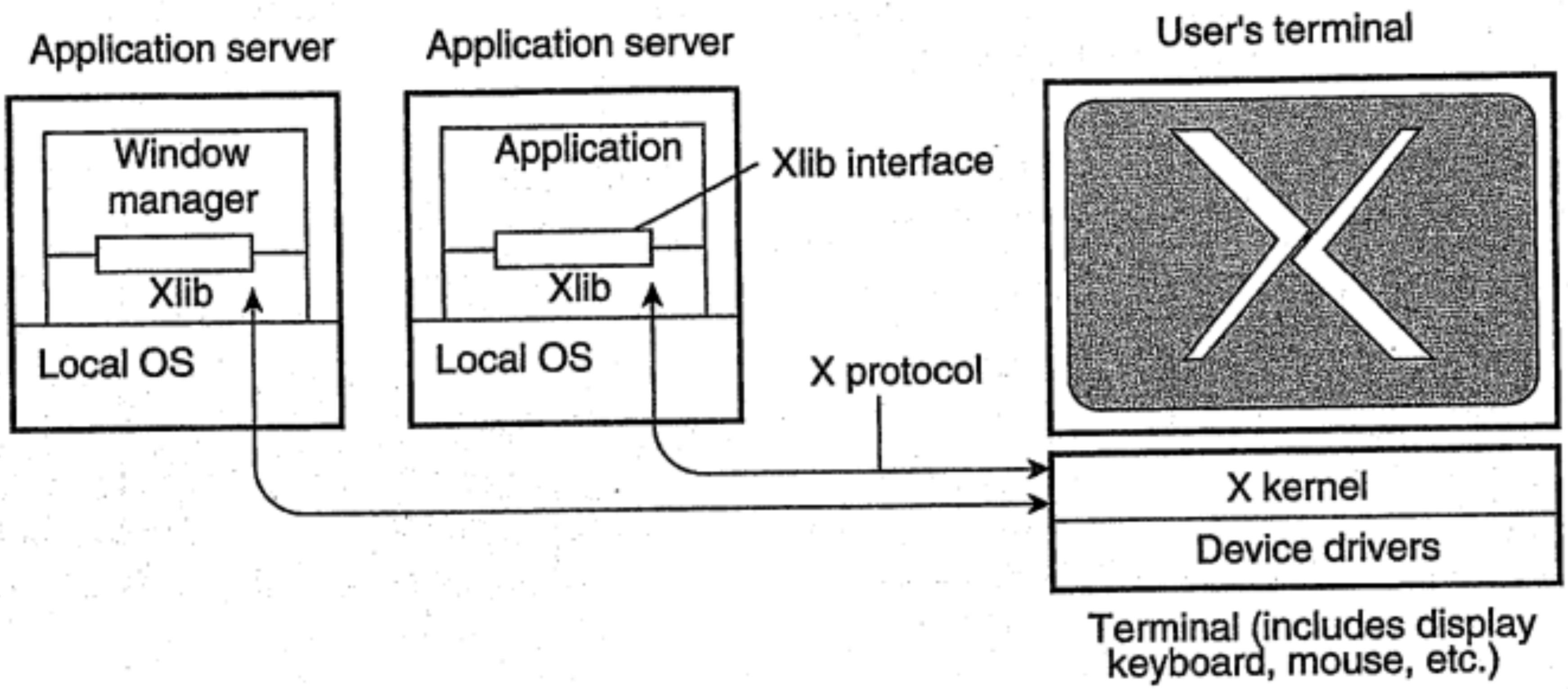
3.1. Networked User Interfaces



A networked app with its own protocol

Thin-client approach

X Window System



Thin-client Network Computing

- X-client versus X-server
- Applications manipulate a display using the specific display commands as offered by X.
- Applications written for X should preferably separate application logic from user-interface commands → not applicable
- Solution: compress X message

Example: a program X-client using Xlib


```
#include <X11/Xlib.h> // Every Xlib program must include this
#include <assert.h>   // I include this to test return values the lazy way
#include <unistd.h>   // So we got the profile for 10 seconds

#define NIL (0)       // A name for the void pointer

main()
{
    // Open the display
    Display *dpy = XOpenDisplay(NIL);
    assert(dpy);

    // Get some colors
    int blackColor = BlackPixel(dpy, DefaultScreen(dpy));
    int whiteColor = WhitePixel(dpy, DefaultScreen(dpy));

    // Create the window
    Window w = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy), 0, 0,
                                   200, 100, 0, blackColor, blackColor);
```

Example: a program X-client using Xlib

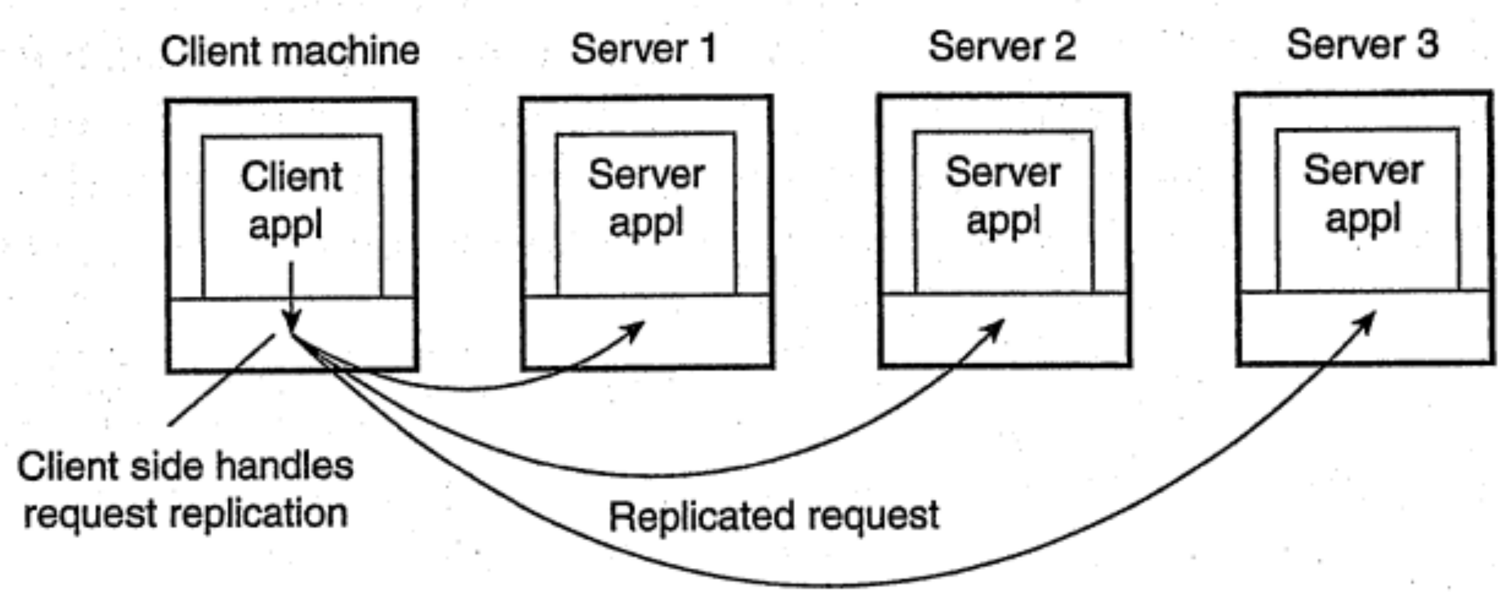

```
// We want to get MapNotify events
    XSelectInput(dpy, w, StructureNotifyMask);

    // "Map" the window (that is, make it appear on the screen)
    XMapWindow(dpy, w);

    // Create a "Graphics Context"
    GC gc = XCreateGC(dpy, w, 0, NIL);

    // Tell the GC we draw using the white color
    XSetForeground(dpy, gc, whiteColor);

    // Wait for the MapNotify event
    for(;;) {
            XEvent e;
            XNextEvent(dpy, &e);
            if (e.type == MapNotify)
                    break;
    }
```

Example: a program X-client using Xlib


```
    // Draw the line
    XDrawLine(dpy, w, gc, 10, 60, 180, 20);

    // Send the "DrawLine" request to the server
    XFlush(dpy);

    // Wait for 10 seconds
    sleep(10);
}
```

3.2. Client-side software for distribution transparency

- Transparent distribution:
  - Transparent access
  - Transparent migration
  - Transparent replication
  - Transparent faults

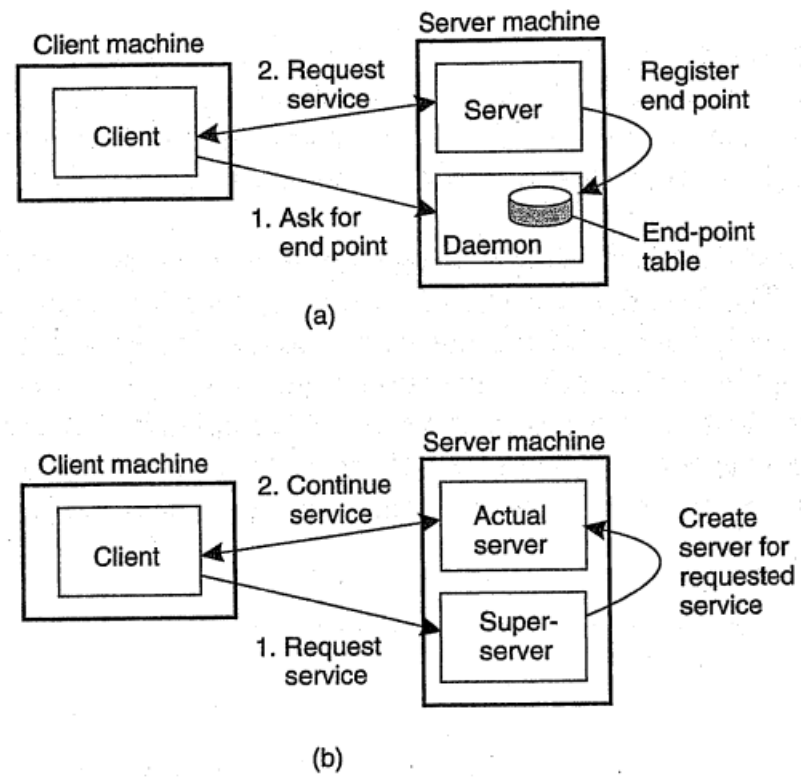


4. Servers

General design issues

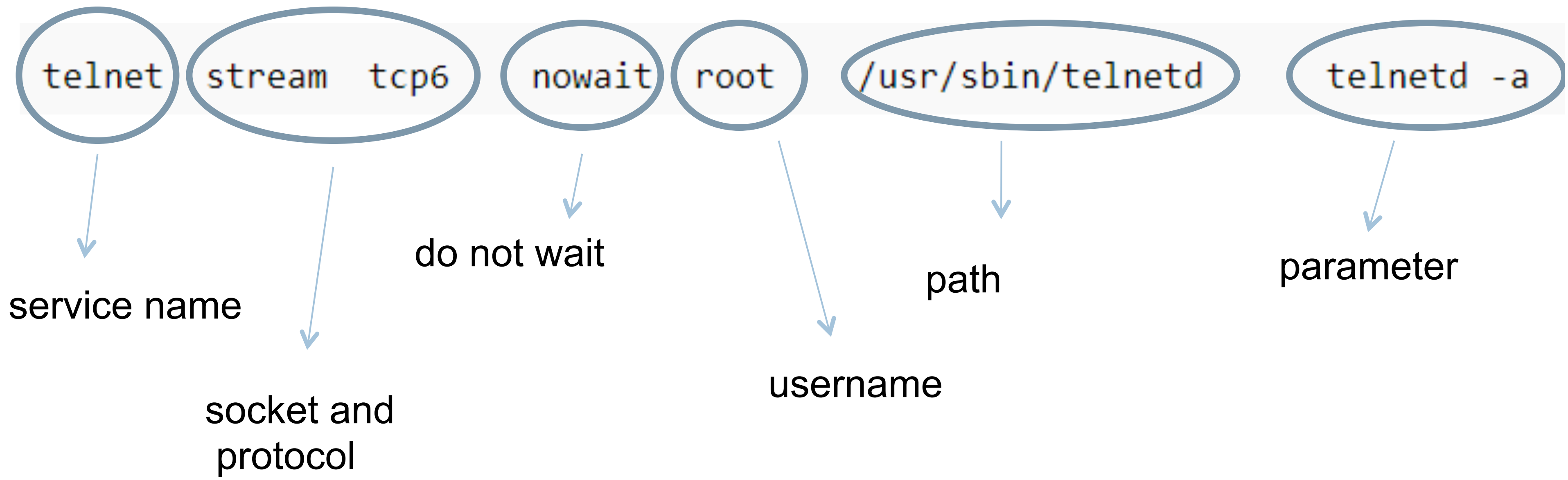
4.1. General design issues

- Organize server
  - Iterative server
  - Concurrent server
- Find server:
  - End-point (port)
  - Deamon
  - Superserver
- Interrupt server
- Stateless & stateful server



Inetd

- Configuration info in the file */etc/inetd.conf*



Example:

- A program *errorLogger.c*

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  const char *fn = argv[1];
  FILE *fp = fopen(fn, "a+");

  if(fp == NULL)
    exit(EXIT_FAILURE);

  char str[4096];
  //inetd passes its information to us in stdin.
  while(fgets(str, sizeof(str), stdin)) {
    fputs(str, fp);
    fflush(fp);
  }
  fclose(fp);
  return 0;
}
```

Configure inetd

- Insert info into */etc/services*

```
errorLogger 9999/udp
```

- Insert info into */etc/inetd.conf*

```
errorLogger dgram udp wait root /usr/local/
bin/errlogd errlogd /tmp/logfile.txt
```

5. Code migration

Why?

- Improve performance
  - Server code to client
  - Client code to server
  - Exploiting parallelism

Code migration models

- Alternatives for code migration.