Chapter 3

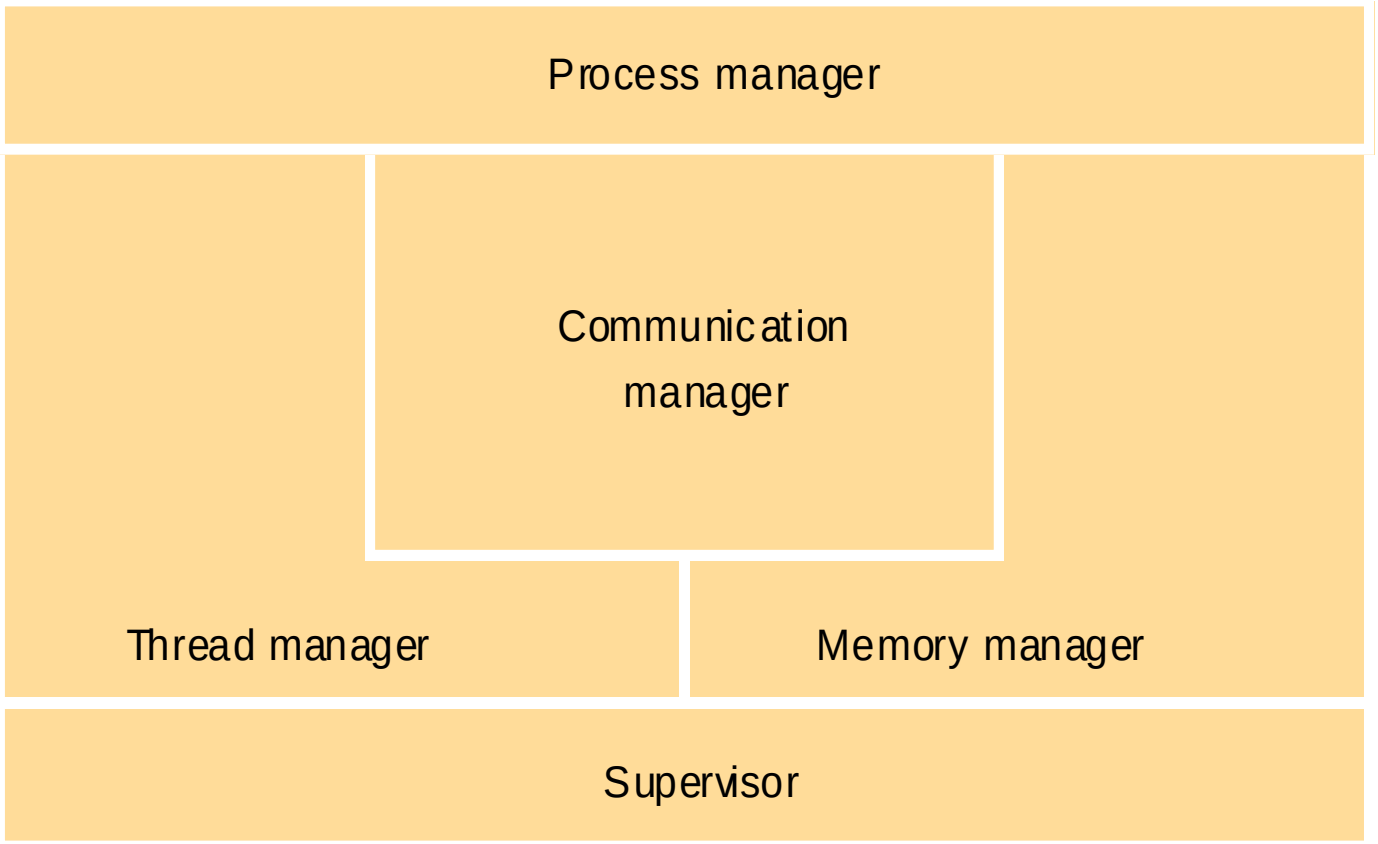
Operating System Support

- **The operating system Layer**
- The OS layer is present below the middleware layer.
- OS provides problem-oriented abstractions of the underlying physical resources.
- The middleware-OS combination of a distributed system should have good performance.
- Middleware is responsible to provide proper utilization of local resources to implement mechanisms for remote invocations between objects or processes at the nodes.
- Kernel and server processes are responsible to manage resources and present clients with an interface to the resources.
- Kernel, as a resource manager, must provide encapsulation, protection and concurrent processing.
- Clients access resources by making invocations to a server object or system calls to a kernel.

- ***Core OS Functionality***

1. Process Manager: As a process manager, OS handles the creation of processes and the operations upon them.
2. Thread Manager: As a thread manager, OS is responsible for thread creation, synchronization and scheduling.
3. Communication Manager: OS is responsible for communication between threads attached to different processes on same computer or in remote processes. In some OS, additional service is necessary for communication.
4. Memory Manager: OS also deals with management of physical and virtual memory to ensure efficient data sharing.
5. Supervisor: OS is responsible to dispatch interrupts, system call traps, memory management control, hardware cache and so on.

Core OS functionality



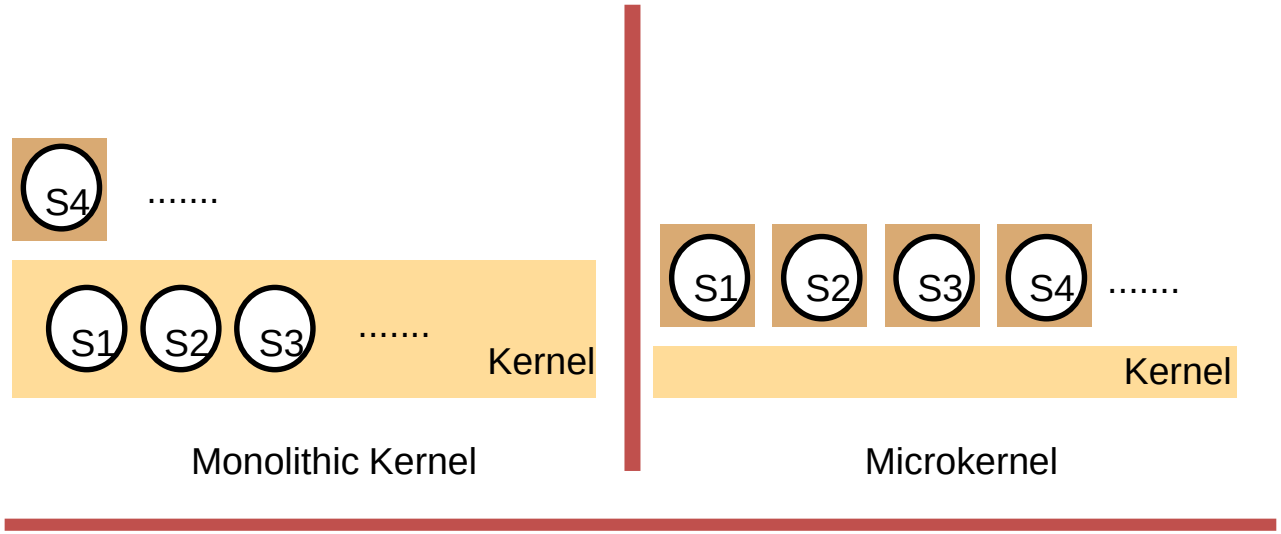
- **Protection**
- The underlying resources in a distributed system should be protected from illegitimate access.
- To ensure protection of resources, operating system must provides a means to provide clients to perform operations on the resources if and only if they have rights to do so.
- For example: Consider a file with only read or write operations. The illegitimate access would be access of file for write by the client who have read access only.
- Resources can be protected by the use of type-safe programming language like JAVA in which no module can access a target module without having a reference to it.
- Hardware support can be employed to protect modules from illegitimate invocations, for which kernel should be implemented.

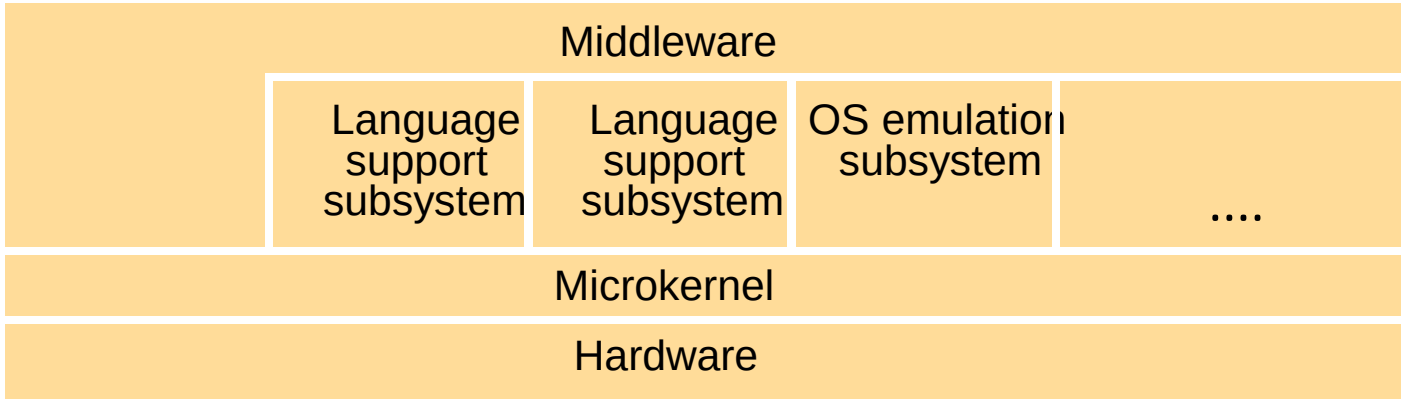
- ***Kernel:***
- A kernel is a program that is executed with complete access privileges for the physical resources on its host computer.
- It controls memory management.
- It ensures access of physical resources by the acceptable codes only.
- A kernel process executes in supervisor mode and restricts other processes to execute in user mode.
- It sets up address space for the process. A process is unable to access memory outside its address space.
- A process can switch the address space via interrupt or system call trap.

- ***Monolithic Kernel:***
- Monolithic kernel is a single large process running in a single address space.
- All kernel services executes in the kernel address space.
- It provides faster execution as there is no necessity of address space switching to execute kernel services.
- The device drivers reside in kernel space making it less secure.
- Fault in a single kernel service collapse the whole kernel.
- Adding new features require recompilation of whole kernel.
- Eg: Kernel of Unix and Linux

- ***Micro Kernel:***
- Micro kernel is a kernel in which it is broken into separate process called servers.
- Servers run in kernel space and user space.
- It provides slower execution.
- Device drivers reside in user space.
- Fault of a single server does not collapse the kernel.
- Eg: Kernel of Mac OS X and Windows NT.
- ***Selection of Kernel:***
- Micro kernel is suitable for distributed operating system.
- The services of distributed system are complex and segregation of micro kernel makes it easy.
- Micro kernel provides faster communication among the processes with low overhead.

Monolithic kernel and microkernel





The microkernel supports middleware via subsystems

Advantages and disadvantages of microkernel

- flexibility and extensibility
  - services can be added, modified and debugged
  - small kernel -> fewer bugs
  - protection of services and resources is still maintained
- service invocation expensive
  - unless LRPC is used
  - extra system calls by services for access to protected resources

- **Process and Thread**
- Process is an instance of a computer program that is being executed.
- In traditional OS, each process has an address space and a single thread of control.
- The distributes systems require internal concurrency for which process is multi threaded.
- Thread is a light weighted process that shares the same address space of the corresponding process but runs in quasi-parallel.
- Thread is the operating system abstraction of an activity.
- A process contains execution environment which is local kernel managed resources to which its threads have access.

- ***Importance of Thread in Distributed System***

1. Distributed system needs to support multiple users. Such concurrency is impossible without multi-threaded support.
2. Thread is able to block system calls without blocking the entire process. This makes distributed system possible to communicate by maintaining multiple logical connections at the same time.
3. Thread can run in a single address space parallel on different CPU.
4. Threads can share a common buffer which make it possible to implement producer-consumer problem easily.

- ***Comparison of Process and Thread***

1. Processes run in separate memory space but threads can run in a same memory space
2. Process is a self-contained entity while thread depends on process for existence.
3. Process depends on resources heavily while threads require minimal amount of resources.
4. Threads within a process can communicate easily but processes must use IPC for communication.
5. Process has considerable overhead while thread has no overhead.

Support for communication and invocation

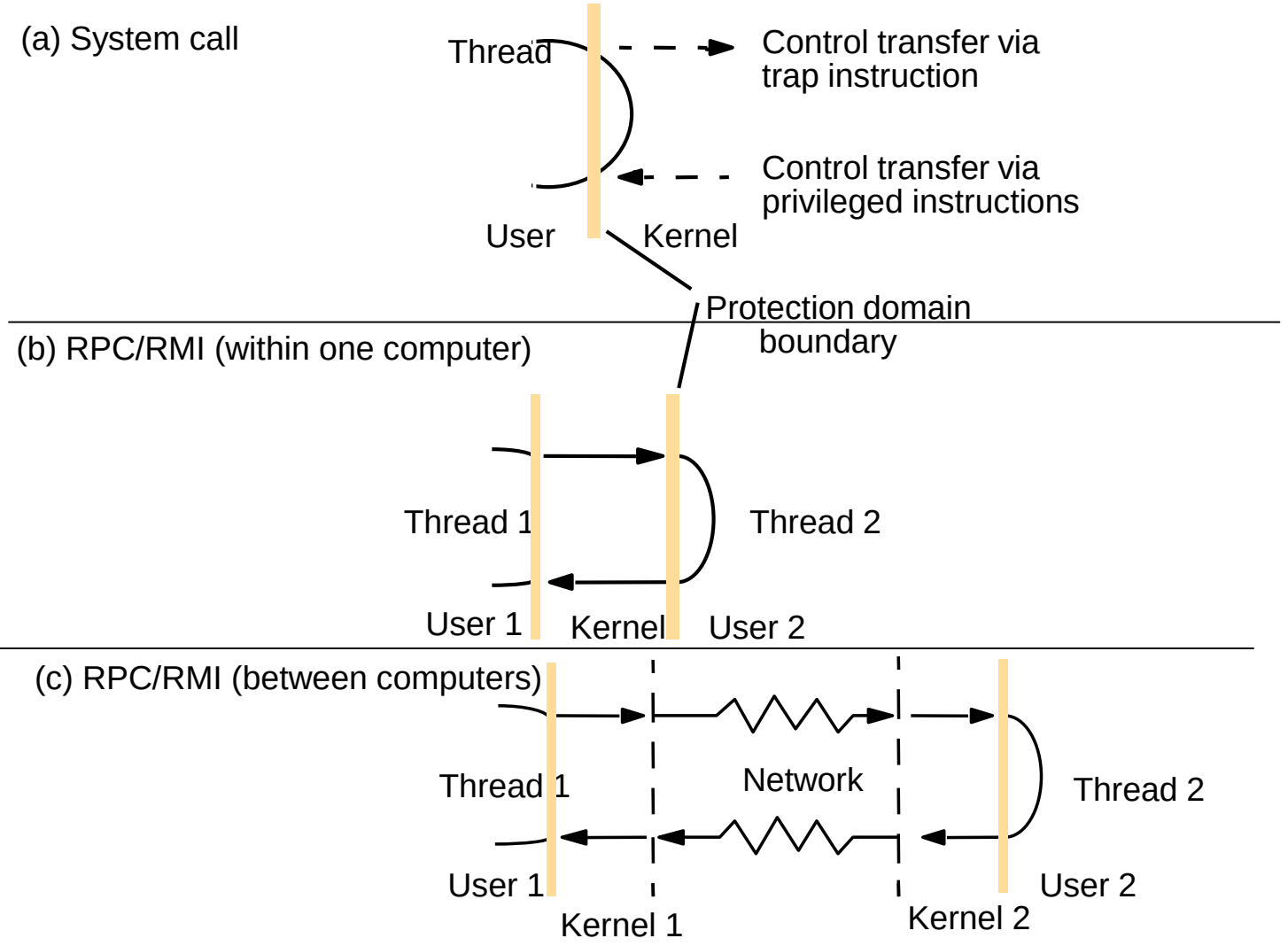
- The performance of RPC and RMI mechanisms is critical for effective distributed systems.
  - Typical times for 'null procedure call':
  - Local procedure call < 1 microseconds
  - Remote procedure call ~ 10 milliseconds

  } 10,000 times slower!
  - 'network time' (involving about 100 bytes transferred, at 100 megabits/sec.) accounts for only .01 millisecond; the remaining delays must be in OS and middleware - latency, not communication time.
- Factors affecting RPC/RMI performance
  - marshalling/unmarshalling + operation despatch at the server
  - data copying:- application -> kernel space -> communication buffers
  - thread scheduling and context switching:- including kernel entry
  - protocol processing:- for each protocol layer
  - network access delays:- connection setup, network latency

Implementation of invocation mechanisms

- Most invocation middleware (Corba, Java RMI, HTTP) is implemented over TCP
  - For universal availability, unlimited message size and reliable transfer
  - Sun RPC (used in NFS) is implemented over both UDP and TCP and generally works faster over UDP
- Research-based systems have implemented much more efficient invocation protocols, E.g.
  - Firefly RPC (see www.cdk3.net/oss)
  - Amoeba's doOperation, *getRequest*, *sendReply* primitives (www.cdk3.net/oss)
- Concurrent and asynchronous invocations
  - middleware or application doesn't block waiting for reply to each invocation

Invocations between address spaces



- **OS Architecture**
- ***Network OS and Distributed OS***
- Network OS has networking capability.
- They can be used to access remote resources.
- Each node has its own system image and a user is capable to log in to another computer and run processes there.
- Distributed OS is an operating system that produces a single system image for all the resources in the distributed system.
- Users are never concerned with where their programs run.
- The OS has control over all the nodes in the system.

- ***Preference of NOS over DOS:***

1. Users invest in applications to meet their current problem solving needs. So, they do not tend to adapt to a new operating system which is unable to run their applications even if they are more efficient.
2. Users always prefer to have a degree of autonomy for their machines, even in a closely knit organization. This is because they do not want to spoil their process performance due to the process run by other users.

THE END