Administrative Info

— OS1-Labs —

Administrative Info: Labs

- General info
  - Not mandatory, but you can gain bonus points (cf. Intro Lecture). We strongly recommend to participate in the labs!
  - Programming tasks, C language
  - Overall, there will be 3 labs that offer hands-on experience with the Linux (kernel) programming environment.
  - Labs get published one after another via TUCaN.
    The next lab starts after the testing of the previous one.
    - Lab 1: Monday, Nov 3
    - Lab 2: Monday, Nov 24
    - Lab 3: Monday, Dec 15
  - **Work in groups of 2-3 people!**
    - No individual submissions!
  - Check our website regularly for updates!

Administrative Info: Labs

- Lab submissions
  - Submit your group solutions via email
    - os-lab@deeds.informatik.tu-darmstadt.de
  - List the names, matriculation numbers and email addresses of all group members in your submission email.
  - Package your solution as tarball or zip archive
    - Include all files needed to build and execute your solution
  - **Indicate time collisions** with other courses for testing in your email! (see next slide)

Administrative Info: Labs

- Lab testing
  - For each lab, you have to pass the testing to get the bonus points.
  - Attendance is mandatory for all group members.
  - We build and execute your submitted solutions; you have to explain it and answer some questions.
  - Testing takes place on the Friday after the submission deadline for each lab.
  - We assign testing time slots for each group.
    - **Indicate collisions with other courses in your submission email!** Otherwise, we cannot take collisions into account.
    - The time slots are fixed once they are assigned! We won't accommodate change requests afterwards!

Administrative Info: Labs

- Recommended setup and advice for all labs
  - Use a recent Linux distribution with up-to-date tools and kernel
    - Linux kernel 3.12 or newer should work
      - We use kernel 3.16 (or newer) for testing
      - Do not use unstable or outdated kernel versions
    - GNU make 3.8+ and GCC 4.6+
      - GCC 4.8 or 4.9 recommended
  - Use a virtual machine for your test/development environment
    - Later labs deal with kernel modules and may crash your system!
  - Avoid hardcoded paths in your Makefiles and shell scripts, especially if they include kernel specific components.
    - Paths and kernel versions on your test system may differ from ours.

Administrative Info: Lab 1

- Lab 1: Processes and IPC
  - Starts next week
  - Covers contents from this and last lecture

- Schedule: Lab 1
  - **Monday, Nov 3**
    - Publication of lab description via TUCaN
  - **Tuesday, Nov 18**
    - Deadline for solution submissions
  - **Friday, Nov 21**
    - Testing
      - Takes place in the morning in room E302

Processes (and Threads)

- Process Management
- Thread Models
- Inter-process Communication (IPC)
- Hierarchical Microkernel and IPC

Cooperating Processes

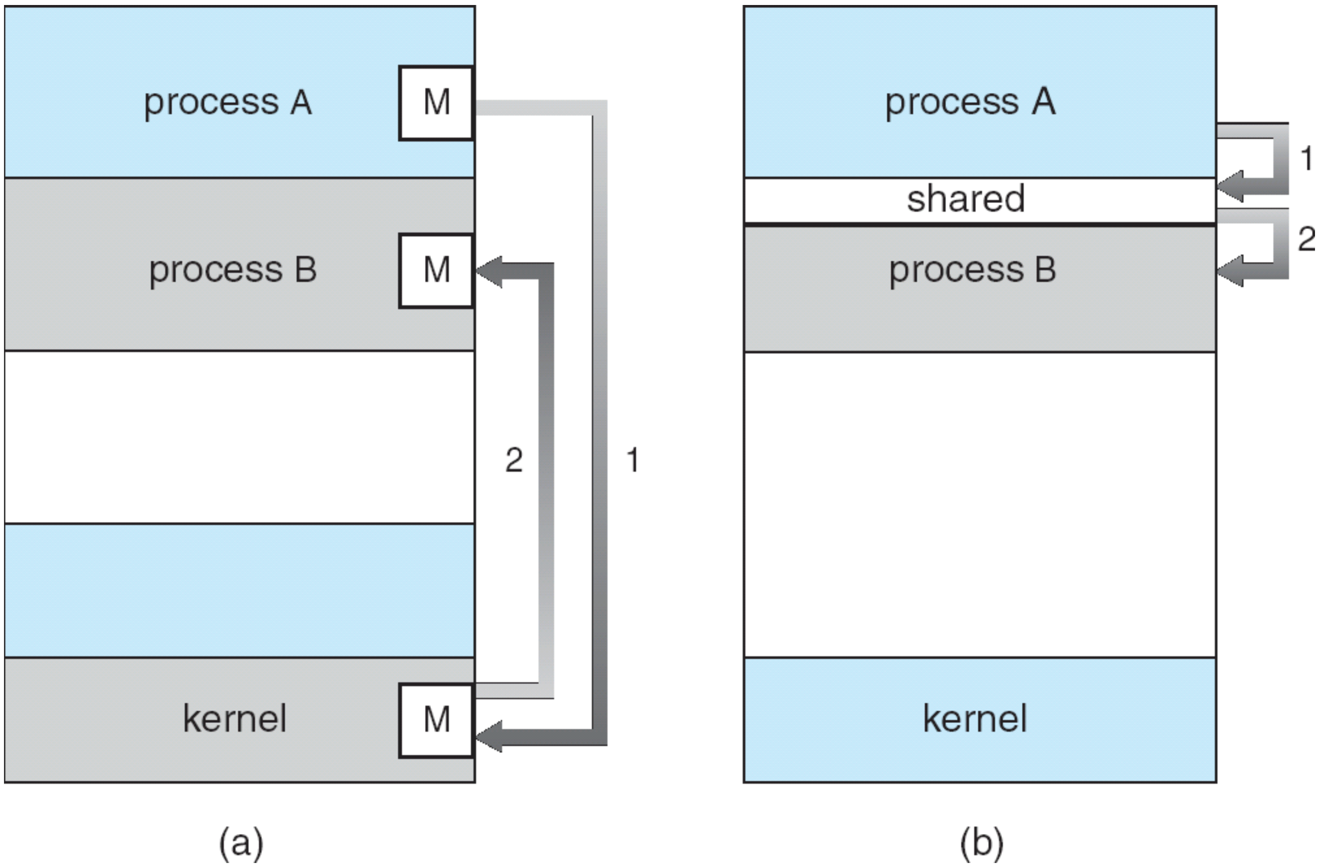
- Processes can be
  - **Independent**
    - A processes is independent if it cannot affect or be affected by other processes executing in the system.
    - Independent processes do not share data with other processes.
  - **Cooperating**
    - A process is cooperating if it can affect or be affected by other processes executing in the system.
    - A process that shares data with other processes is a cooperating process.

Cooperating Processes

- Why do we want cooperating processes?
  - Information sharing
  - Speedup
  - Modularity
  - Convenience
  - Privilege separation

- Process cooperation issues
  - data sharing
  - need a communication channel/medium
  - coordination & synchronization (later lectures)
    - race conditions, deadlocks
    - critical sections, locking, semaphores

Inter-process Communication (IPC) Models

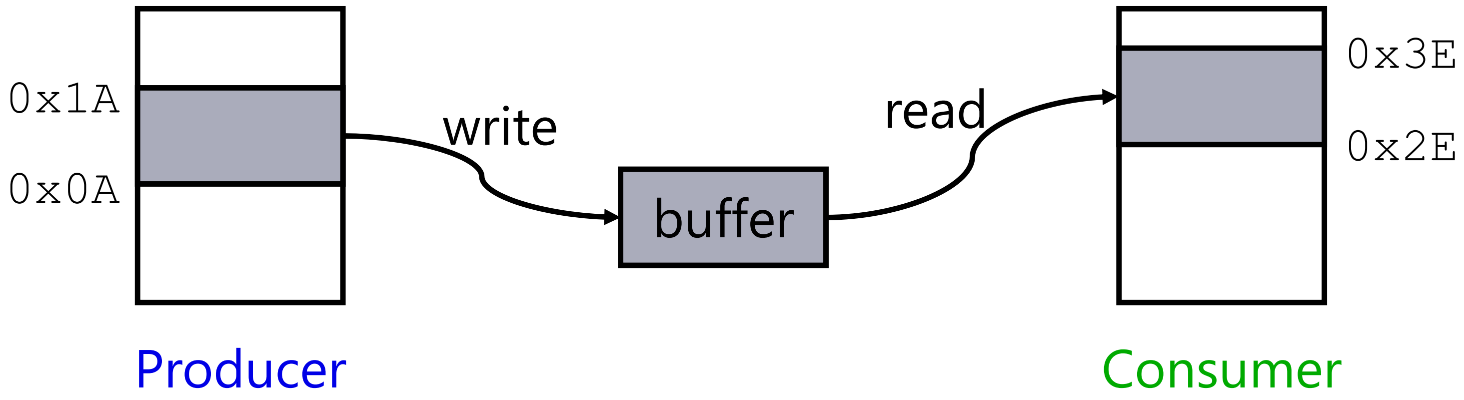


Message Passing

Shared Memory

Shared-Memory: Producer-Consumer Model

- Cooperating processes
  - Producer process produces information/data
  - Consumer process consumes information/data
  - Both share a common memory region (buffer)



- How large should the buffer be?
  - bounded → assumes fixed buffer size
    - full buffer?
  - unbounded → no practical limit on buffer size
    - realistic?

Shared-Memory: Bounded Buffer Producer-Consumer

- Variables shared between producer and consumer processes

```
#define BUFFER_SIZE 10

typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

buffer bound: 10 items

shared circular buffer

next free position, shared

first full position, shared

- How to recognize that buffer is
  - empty? `in == out`
  - full? `(in + 1) % BUFFER_SIZE == out`

Shared-Memory: Bounded Buffer Producer-Consumer

```
while (true) {
  while (((in + 1) % BUFFER_SIZE) == out)
    ; // do nothing, buffer full

  // produce an item
  buffer[in] = new_item;
  in = (in + 1) % BUFFER_SIZE;
}
```

- Producer
  - waits if buffer is full
  - inserts new items

```
while (true) {
  while (in == out)
    ; // do nothing, buffer empty

  // remove an item from the buffer
  item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  process(item);
}
```

- Consumer
  - waits if buffer is empty
  - removes items

explicit synchronization needed!
covered in later lectures

Concurrent access to shared variables?

Message Based Communication

- **Shared memory** IPC – processes use shared memory region as communication medium

- **Message passing** IPC – processes use (abstract) communication link
  - two primitive operations, message size may be fixed or variable
    - **send**(message)
    - **receive**(message)
  - If P and Q wish to communicate, they need to:
    - establish a communication link between them
    - exchange messages via send/receive
  - Implementation/design of the communication link
    - physical (shared memory, hardware bus)
    - logical (logical properties)

Link Implementation Issues

- How to establish the links?
- Can a link be associated with more than two processes?
- How many links can exist between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that a link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

→ Addressing, naming (direct vs. indirect communication)

→ Synchronization Aspects (blocking vs. nonblocking)

→ Buffering (link capacity)

Addressing: Direct Communication

- Processes must name each other explicitly:
  - P: **send**(*Q*, *msg*)
    - process P sends a message to process Q
  - Q: **receive**(*P*, *msg*)
    - process Q receives a message from process P

  - Symmetric addressing
    - send(Q, *msg*); receive(P, *msg*)
  - Asymmetric addressing
    - send(*Q*, *msg*); receive(*&pid*, *msg*)

receive msg from any process, receiver also gets sender's process id

- Properties of communication link
  - Links are established automatically
  - Link associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional
  - Various queue models (zero, bounded, unbounded)

Addressing: Indirect Communication

- Messages are directed to and received from mailboxes or ports
  - Communicating processes do not have to know each others names
  - Each mailbox has a unique id X
    - operations on mailboxes
    - send(*X*, *msg*), receive(*X*, *msg*)
  - Processes can communicate only if they share a mailbox
  - Example Unix pipes: cat xyz | lpr; implicit FIFO mailbox

- Properties of communication link
  - Link is established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
  - Various queue models (zero, bounded, unbounded)

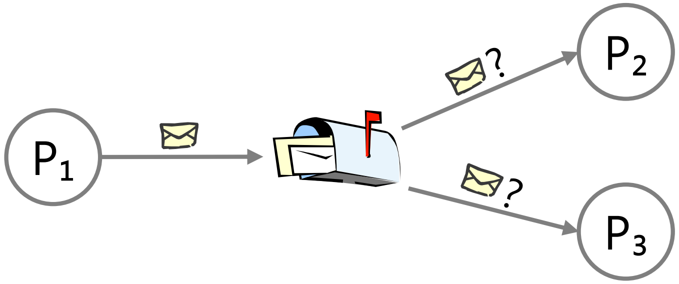
Addressing: Indirect Communication

- Mailbox operations
  - create a new mailbox, once per mailbox
  - send and receive messages through mailbox
  - destroy a mailbox, once per mailbox

- Who owns a mailbox?
  - Process
    - resides in process address space
    - mailbox only exists as long as the process lives
    - owner process receives, others send
  - Operating system
    - mailbox exists on its own, not coupled to some process
    - possibly multiple receivers

Addressing: Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 execute receive
  - Who gets the message?



- Solution choices based on whether we
  - allow a link to be associated with at most two processes
  - allow only one process at a time to execute a receive operation
  - allow the system to select arbitrarily the receiver; sender is notified who the receiver was.

Synchronization Issues

- Message passing may be either blocking or non-blocking
  - different combinations for operations possible

- **Blocking** (B) is considered synchronous
  - **B.Send**: the sender blocks until the message is received by the receiving process or mailbox
  - **B.Receive**: the receiver blocks until a message is available
  - **Rendezvous:** both send and receiver are blocking; no buffers needed

- **Non-blocking** (NB) is considered asynchronous
  - **NB.Send**: the sender sends the message and continues
  - **NB.Receive**: the receiver receives a valid message or null

Message Buffering (Link Capacity)

- Messages of communicating processes reside in temporary queues; implemented in one of three ways:
  - **Zero capacity** – 0 messages
    - Sender must wait for receiver for each message (rendezvous)
  - **Bounded capacity** – finite length of *n* messages
    - Sender must only wait if link is full
  - **Unbounded capacity** – infinite length
    - Sender never waits (no blocking)

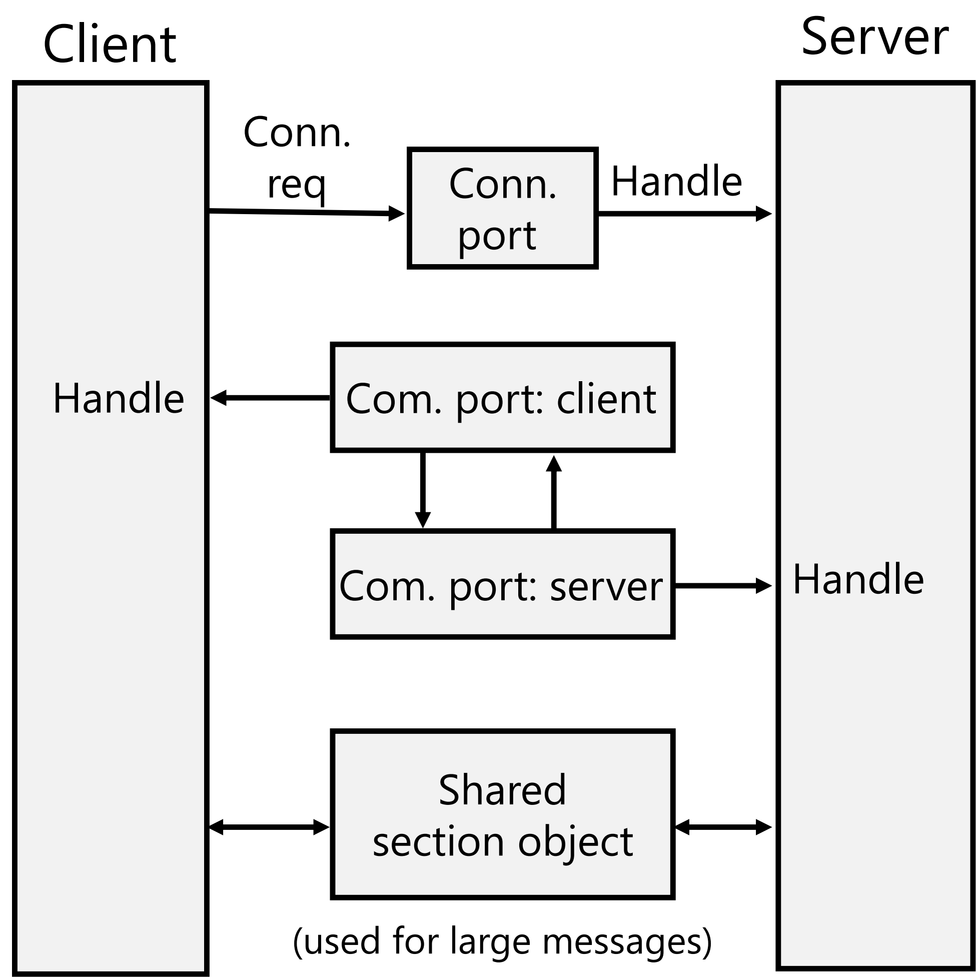
What about receiver? When does it wait?
Are there more possibilities in terms of blocking vs. non-blocking?

IPC Example: (A)LPC in Windows

- (Advanced) Local Procedure Call in Windows
  - Message-passing IPC facility
  - Supports blocking and non-blocking semantics
  - Arbitrary sized messages
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Client-Server style setup
  - Windows internal mechanisms, not directly accessible to 3rd parties
  - further details in
    - Russinovich, Solomon & Ionescu, *Windows Internals*, 5th or 6th edition

IPC Example: (A)LPC in Windows

- Sketch of communication:
  - The client opens a handle to the subsystem's connection port object.
  - The client sends a connection request.
  - The server creates two private communication ports and returns the handle to one of them to the client.
  - The client and server use the corresponding port handle to send messages (or callbacks and to listen for replies).



Client-Server Communication

- Shared Memory Communication
- Message Passing

- Additional strategies suitable for client-server systems
  - Higher level abstractions?

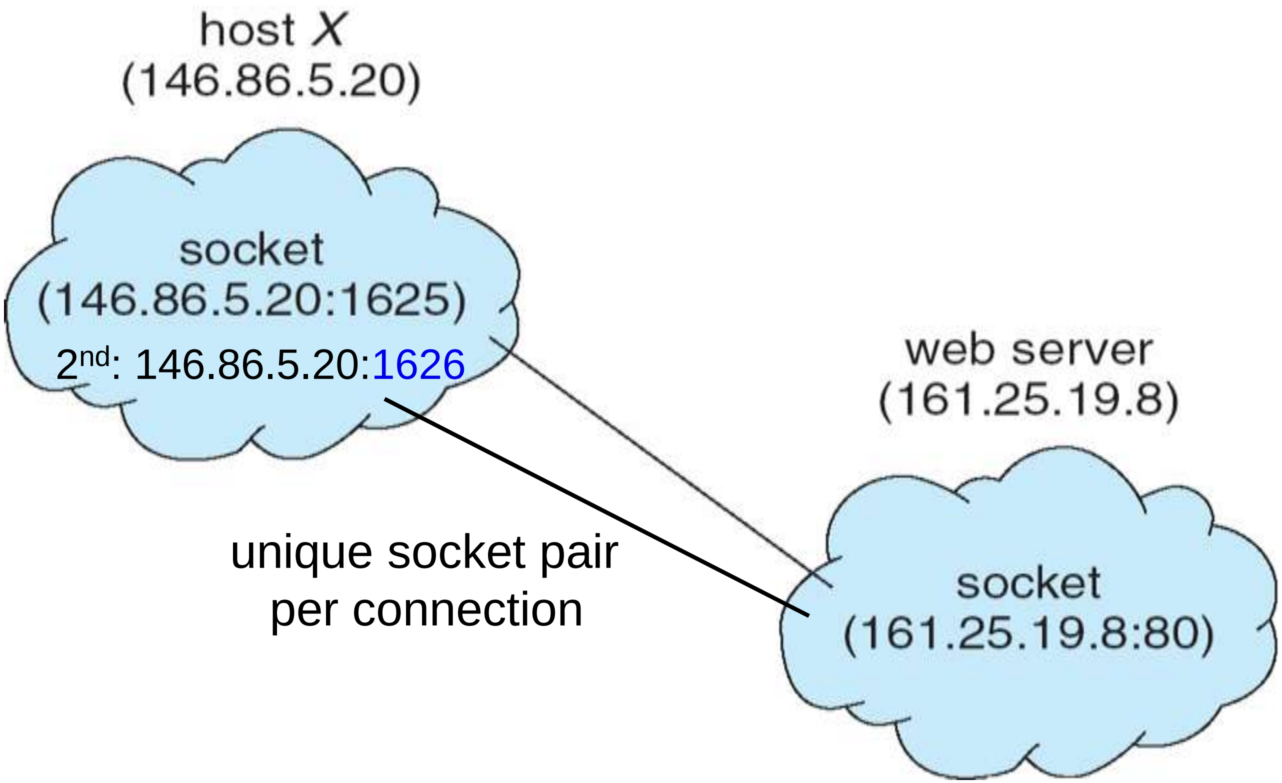
- Sockets
  - Stream/packets of data
- Remote Procedure Calls (RPC)
  - Abstract from procedure calls
- Pipes
  - Stream of data

Sockets

- **Socket:** an endpoint for communication
  - A pair of communicating processes employ sockets at each end (1 per process)
  - Server listens (waits) for incoming client requests
  - different kinds of sockets → here: network (Internet) sockets

- Concatenation of IP address and port
  - IP identifies host
  - port identifies process on host
  - Example: the socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication occurs between socket pairs using some communication protocol
  - Connection-oriented (e.g. TCP)
  - Connection-less (e.g. UDP)

Socket Communication Example



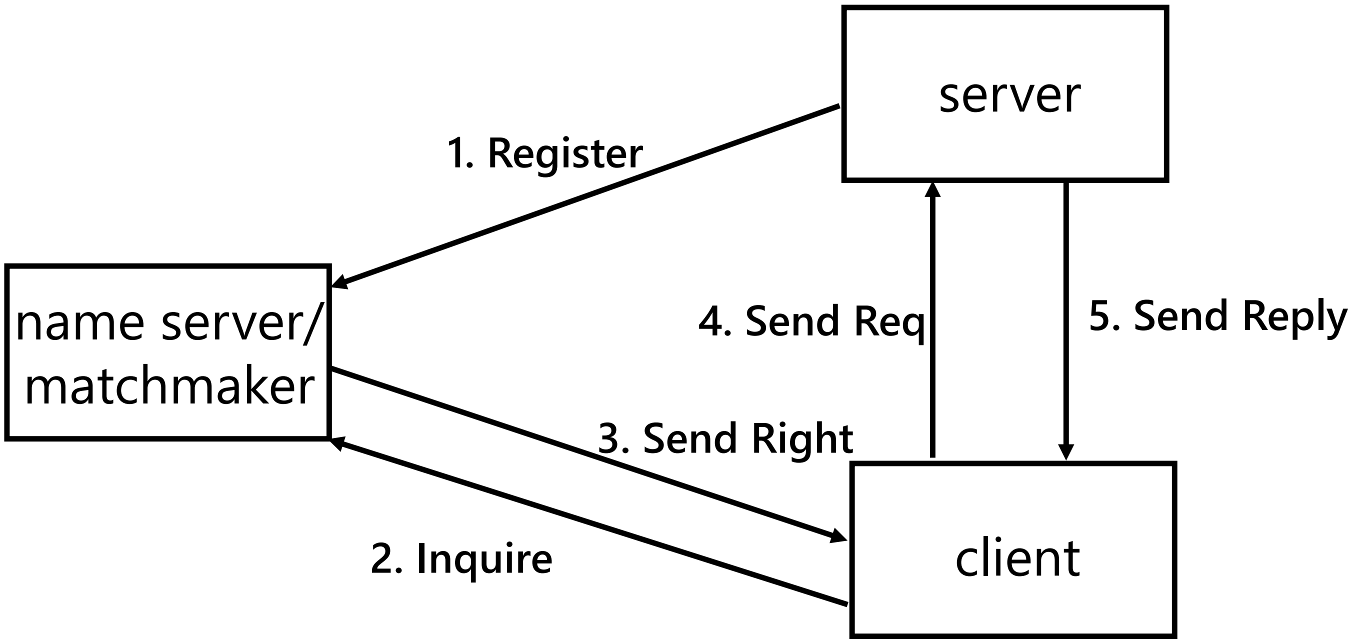
- host X requests webpages
  - 1st request → socket pair: 146.86.2.20:1625, 161.25.19.8:80
  - 2nd request → socket pair: 146.86.2.20:1626, 161.25.19.8:80

Remote Procedure Calls (RPC)

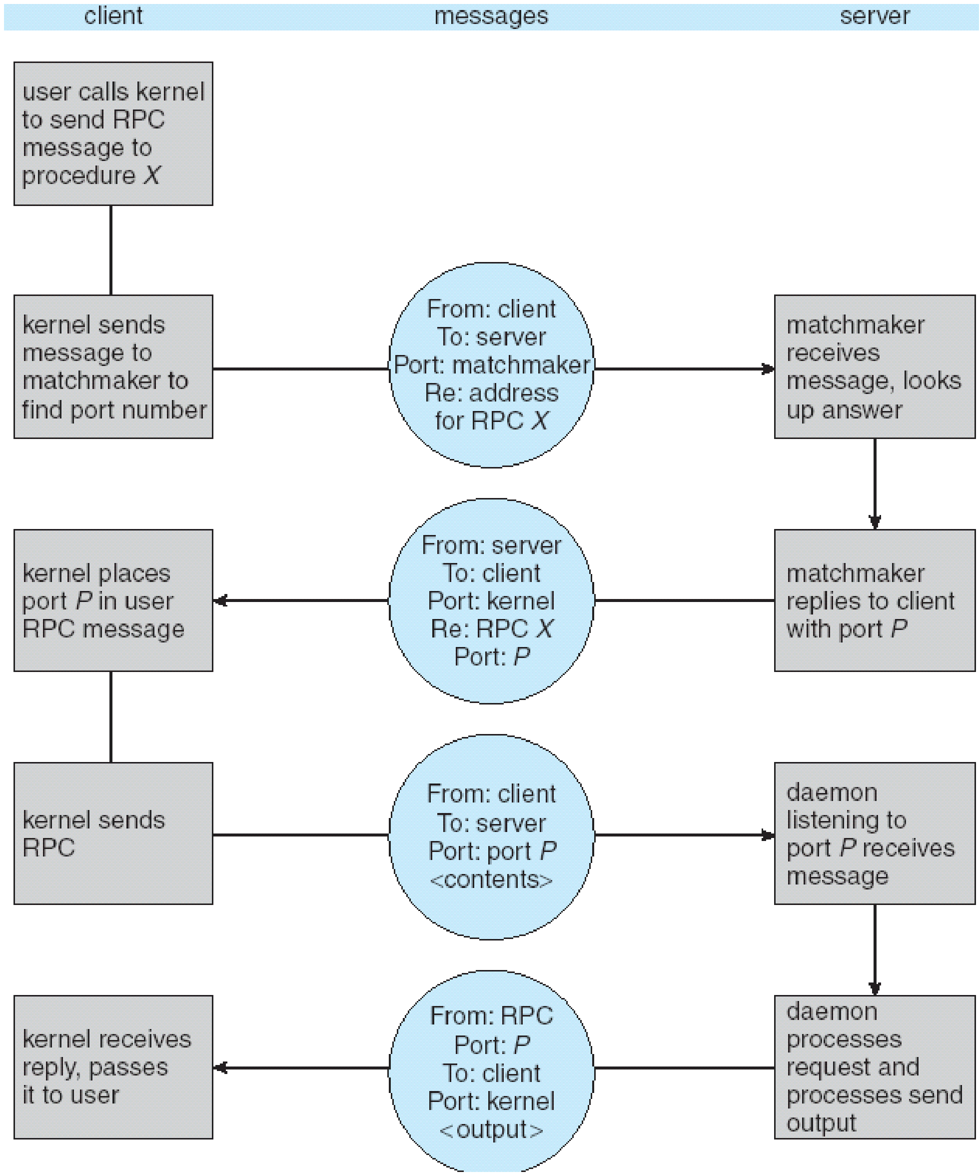
- **Remote procedure call:** abstracts procedure calls between processes on networked systems.
- Stubs – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and marshals the parameters → message.
- The server-side stub receives this message, unpacks the marshaled parameters, and performs the procedure on the server.
- Susceptible to communication faults
  - exactly once and at most once semantics

- How does a client know the server address/socket?

Client-Server Models Behind RPC



Execution of RPC

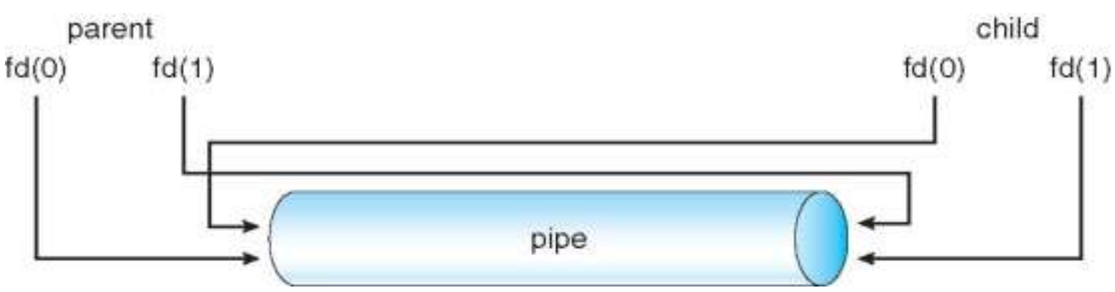


© Silberschatz et al.: Operating System Concepts, 8th ed., Wiley

Pipes

- **Pipe**: conduit, allowing two processes to communicate
- Implementation issues:
  - Unidirectional or bi-directional communication?
  - Half-duplex or full-duplex (for bi-directional mode)?
  - Relationship between communicating processes (e.g. parent/child)?
  - Communication via network or only locally on same machine?

- Ordinary pipe (anonymous)
  - Unidirectional (one-way)
  - Created using pipe() system call
  - Two file descriptors:
    - fd[0]: read-end
    - fd[1]: write-end
  - Not accessible outside the creating process



© Silberschatz et al.: Operating System Concepts, 8th ed., Wiley

Summary

- Processes
  - Definition, PCB
  - States (& changing states)
  - Creation, switching, termination
- Threads
- Inter-process communication
  - Shared memory
  - Message passing
    - Links
    - Addressing, synchronization, buffering
  - Client-Server
    - Sockets
    - RPC
    - Pipes

Hierarchical Microkernel

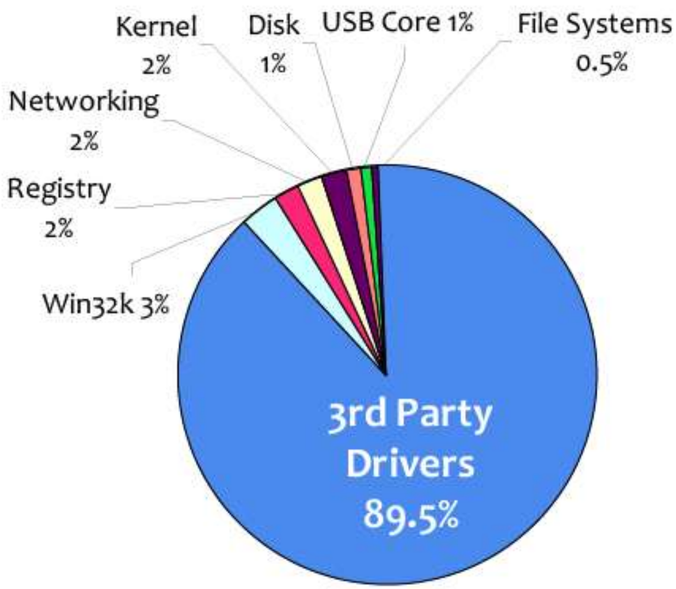
Hierarchical Microkernel: Introduction

- Goals:
  - Make the system flexibly adjustable (for students to play)
  - Make the system robust (for students to play & other reasons)

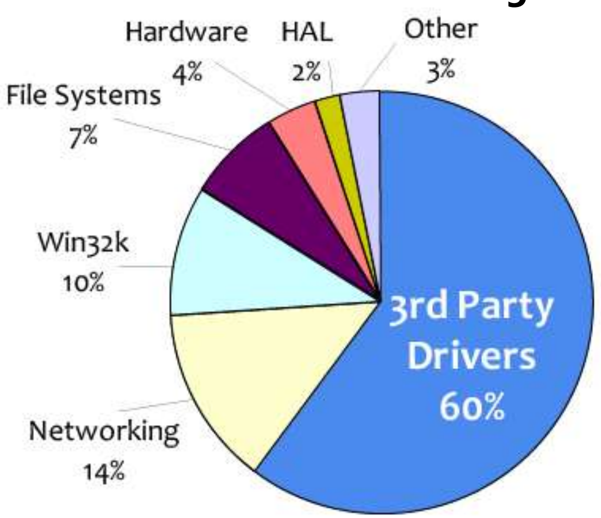
- Flexible adjustability:
  Easily exchange core OS functionality
  - At compile time
  - At runtime
  - Examples: scheduler, memory manager, process loader, …

- Robustness:
  - Easier debugging
  - Prevent system failure caused by single OS component

The Driver Problem: Windows/Linux Example



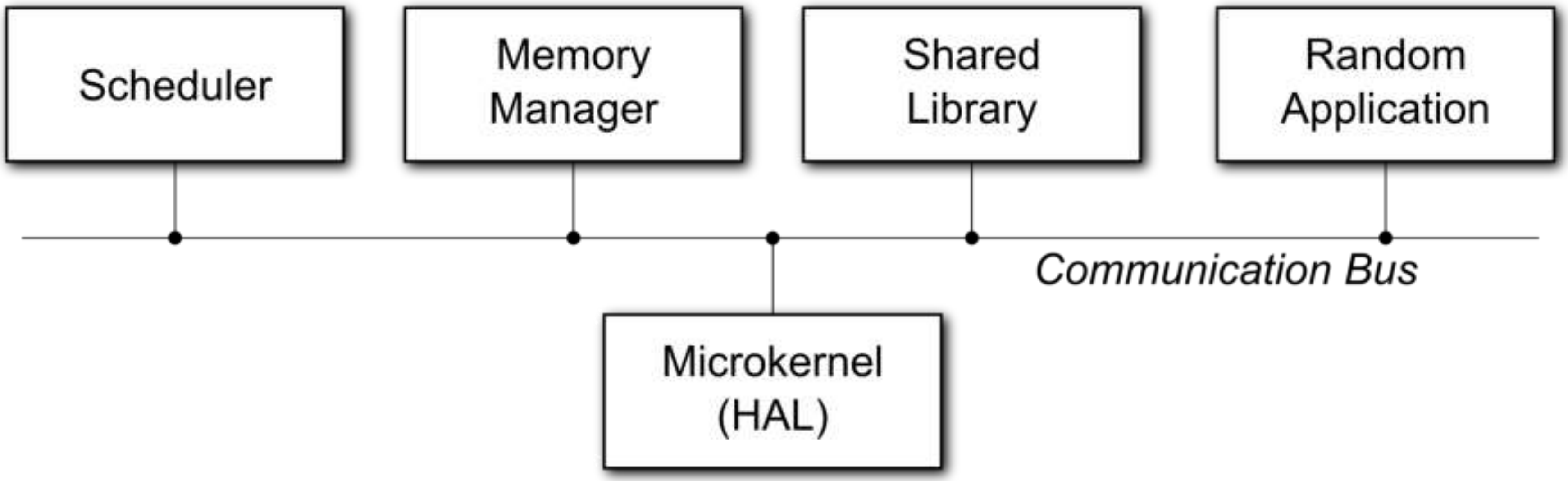
Causes of WinXP outages



Causes of Win2k outages

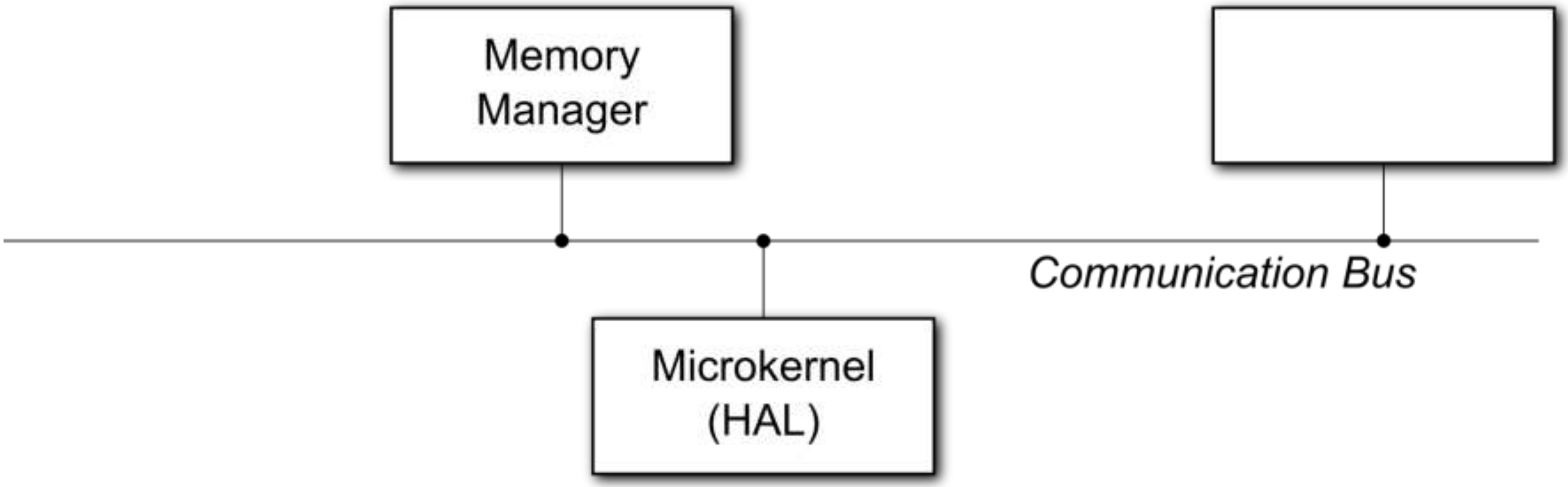
- Device drivers
  - Numerous: 250 installed (100 active) drivers in XP/Vista
  - Large & complex: 70% of Linux code base
  - Immature: every day 25 new / 100 revised versions Vista drivers
  - Access Rights: kernel mode operation in monolithic OSs
- Device drivers are the dominant cause of OS failures <u>despite sustained testing efforts</u>
- Situation similar in Linux and Windows
  - Graph illustrates Windows crash reasons

Hierarchical Microkernel: First Sketch



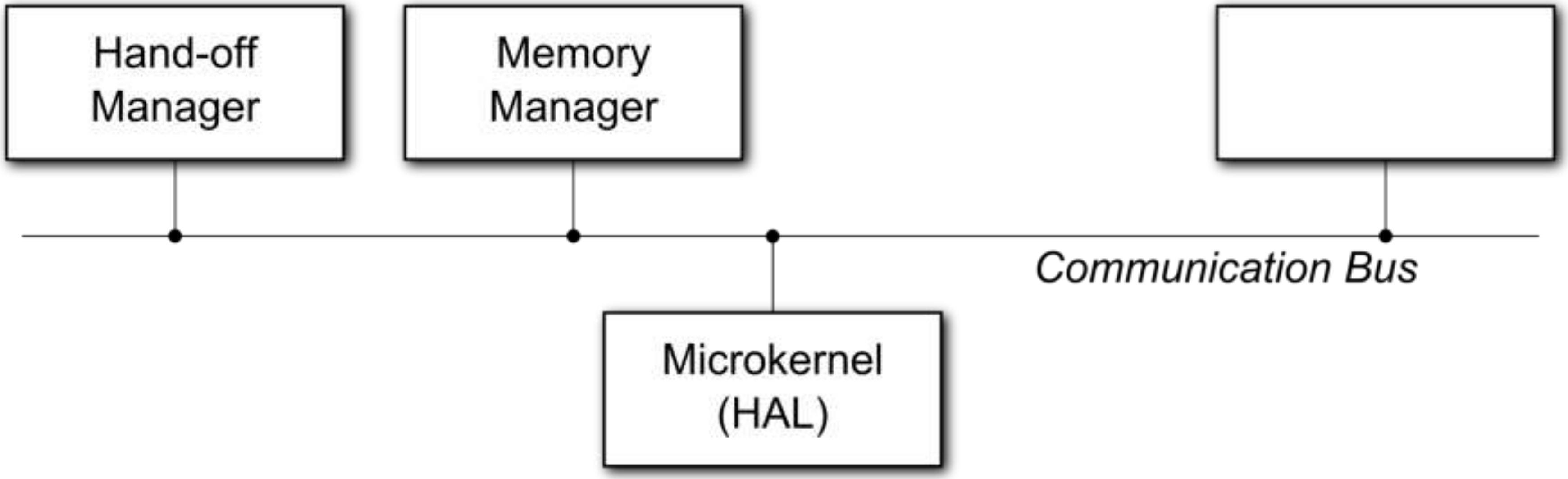
- "Ultra micro" microkernel
  - provides only hardware abstraction layer (HAL)
  - executes in supervisor mode
- Isolated "modules"
  - all user-specific and system functions
- Communication bus
  - simple send/receive interface
  - broadcast communication

Hierarchical microkernel: Flexible Module Exchange



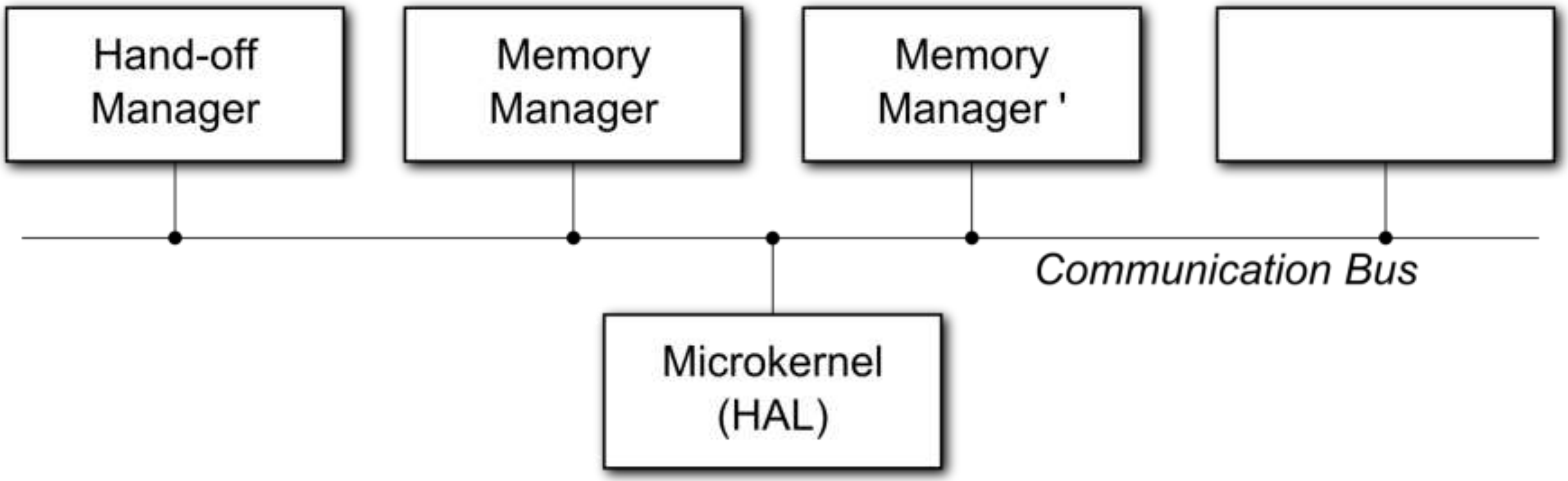
- Example: memory manager exchange
  - microkernel receives memory manager exchange request

Hierarchical microkernel: Flexible Module Exchange



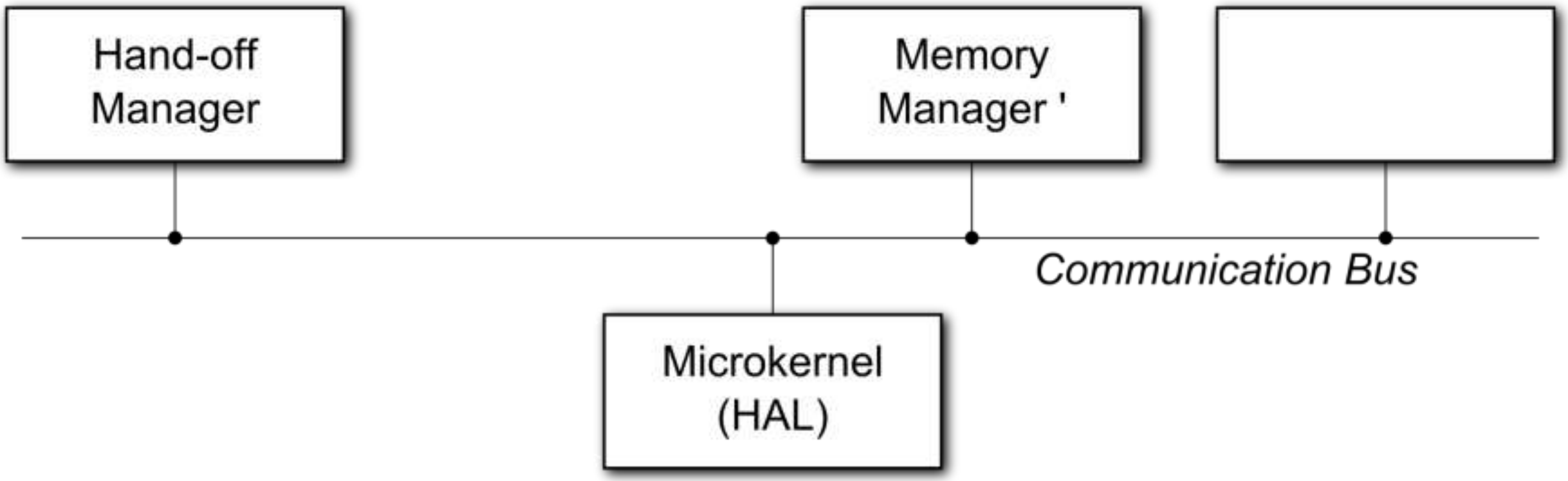
- Example: memory manager exchange
  - microkernel receives memory manager exchange request
  - microkernel loads hand-off manager (via separate loader module)

Hierarchical microkernel: Flexible Module Exchange



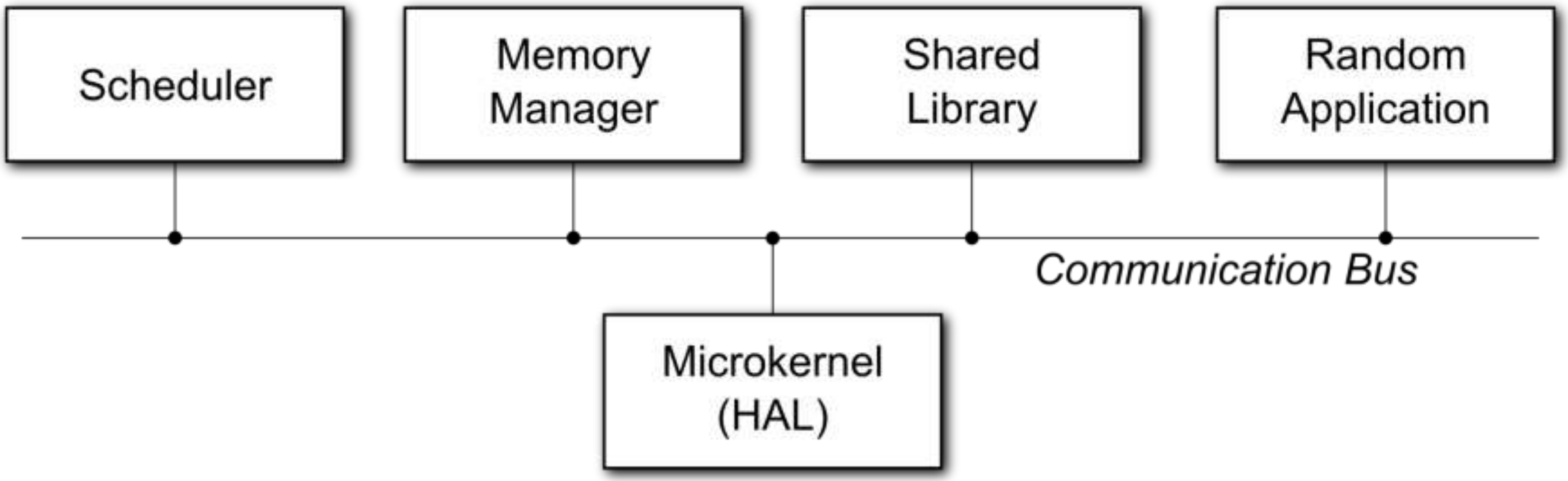
- Example: memory manager exchange
  - microkernel receives memory manager (mm) exchange request
  - microkernel loads hand-off manager (hom) (via separate loader module)
  - hom (indirectly) loads mm’
  - hom (indirectly) stops mm and queues all subsequent requests to mm that traverse the bus
  - hom initiates state transfer from mm to mm’

Hierarchical microkernel: Flexible Module Exchange



- Example: memory manager exchange
  - microkernel receives memory manager (mm) exchange request
  - microkernel loads hand-off manager (hom) (via separate loader module)
  - hom (indirectly) loads mm'
  - hom (indirectly) stops mm and queues all subsequent requests to mm that traverse the bus
  - hom initiates state transfer from mm to mm'
  - hom removes mm, starts mm ', replays all queued requests for mm

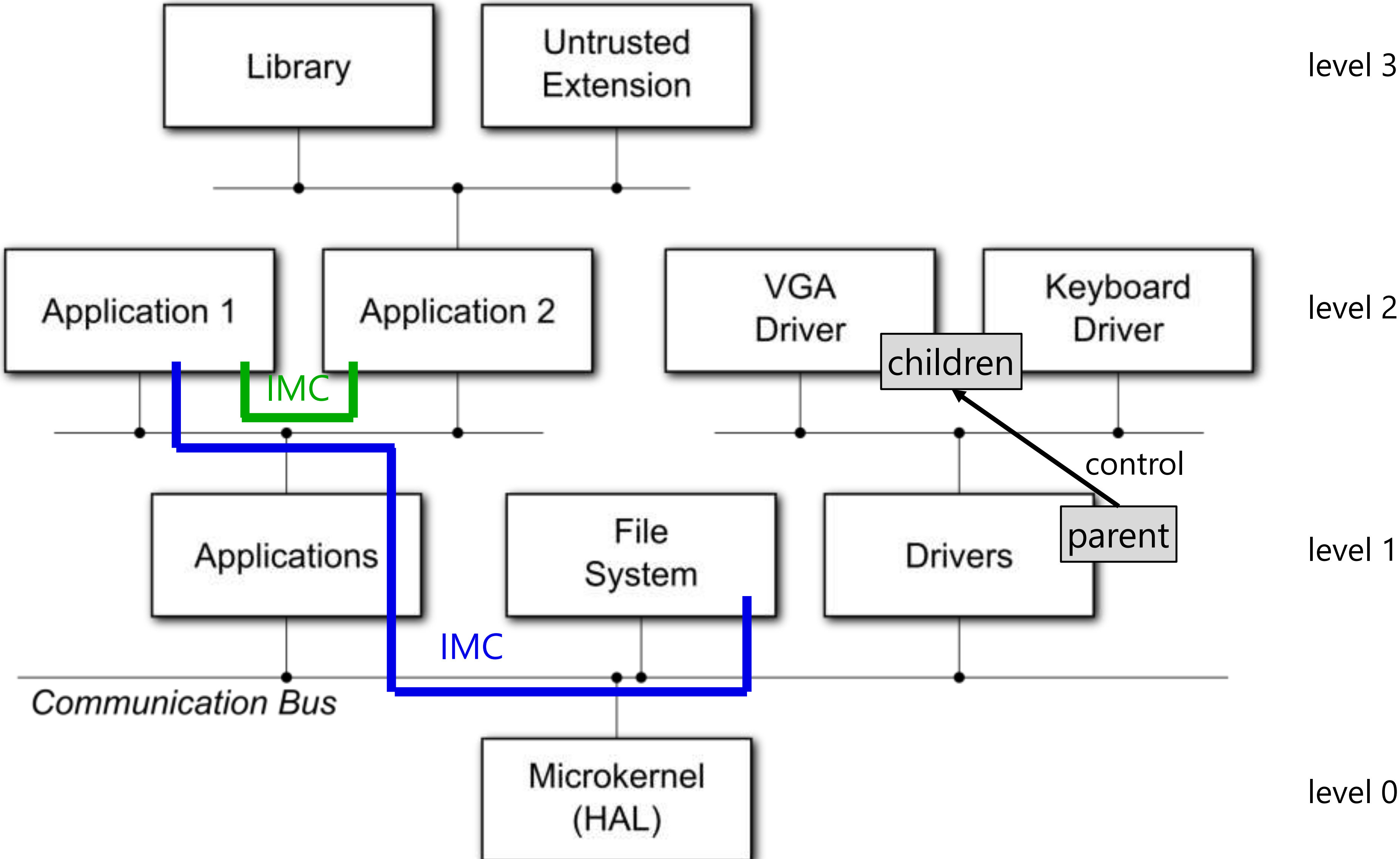
Hierarchical Microkernel: Bus Downsides



- Flexibility (and simple implementation of fault-tolerance mechanisms) through “public” communication

- Congestion?
- Single point of failure (SPOF)?
- Denial of service (DOS) attacks?

Hierarchical microkernel: Hierarchy



Hierarchical Microkernel Wrap-up

- ❑ Modules instead of processes
  - ⇨ inter-module communication (IMC) instead of IPC ☺
  - ⇨ Is there a difference?

- ❑ Strictly message-passing IMC
  - also for "system calls"
  - multicast (bus-local broadcast)
  - bus implementation customizable!
    - synchronous vs asynchronous
    - buffering vs rendezvous
    - message ordering characteristics/guarantees
  - modules can implement custom protocols on top of the busses they are attached to