Operating Systems Notes - Chapter 3

June 9, 2025

Contents

1 Process Concept

1.1 Process Definition and Characteristics

- **Process**: Program in execution.
- Requires resources: CPU time, memory, files, I/O devices.
- Unit of work in most systems.
- Systems comprise OS processes (system code) and user processes (user code), executing concurrently.
- Modern OS supports multiple *threads* of control within processes; threads can run in parallel on multicore systems.
- OS schedules threads onto available processing cores.

1.2 Program vs. Process

- **Program vs. Process**:
  - Program: Passive entity (e.g., executable file on disk).
  - Process: Active entity (program counter, associated resources).
  - Program becomes a process when loaded into memory.
- Multiple processes can be associated with the same program (e.g., multiple instances of a web browser); text sections are identical, but data, heap, and stack sections differ.

1.3 Process Memory Layout

- **Process Memory Layout**:
  - **Text section**: Executable code (fixed size).
  - **Data section**: Global variables (fixed size).
  - **Heap section**: Dynamically allocated memory during runtime (grows/shrinks).
  - **Stack section**: Temporary data for function calls (parameters, return addresses, local variables) (grows/shrinks).
  - Stack and heap grow towards each other; OS prevents overlap.
  - **Activation record**: Pushed onto stack on function call, popped on return.

1.4 Process States

- **Process States**: A process changes state during execution.
  - **New**: Process is being created.
  - **Running**: Instructions are being executed (only one process per processor core at any instant).
  - **Waiting**: Process is waiting for an event (e.g., I/O completion, signal reception).
  - **Ready**: Process is waiting to be assigned to a processor.
  - **Terminated**: Process has finished execution.

1.5 Process Control Block (PCB)

- **Process Control Block (PCB) / Task Control Block**:
  - Data structure in OS representing each process.
  - Contains information:
    - Process state.
    - Program counter.
    - CPU registers.
    - CPU-scheduling information (priority, queue pointers).
    - Memory-management information (base/limit registers, page/segment tables).
    - Accounting information (CPU/real time used, limits).
    - I/O status information (allocated devices, open files).
  - Serves as repository for data needed to start/restart a process.

1.6 Threads

- **Threads**:
  - Single thread of execution: process performs one task at a time.
  - Multithreaded process: multiple threads of execution, performs multiple tasks concurrently.
  - Especially beneficial on multicore systems for parallel execution.
  - PCB expanded to include per-thread information.

Section glossary

Term	Definition
process	A program loaded into memory and executing.
job	A set of commands or processes executed by a batch system.
user programs	User-level programs, as opposed to system programs.
task	A process, a thread activity, or, generally, a unit of computation on a computer.
program counter	A CPU register indicating the main memory location of the next instruction to load and execute.
text section	The executable code of a program or process.
data section	The data part of a program or process; it contains global variables.
heap section	The section of process memory that is dynamically allocated during process run time; it stores temporary variables.
stack section	The section of process memory that contains the stack; it contains activation records and other temporary data.
activation record	A record created when a function or subroutine is called; added to the stack by the call and removed when the call returns. Contains function parameters, local variables, and the return address.
executable file	A file containing a program that is ready to be loaded into memory and executed.
state	The condition of a process, including its current activity as well as its associated memory and disk contents.
process control block	A per-process kernel data structure containing many pieces of information associated with the process.
task control block	A per-process kernel data structure containing many pieces of information associated with the process.
thread	A process control structure that is an execution location. A process with a single thread executes only one task at a time, while a multithreaded process can execute a task per thread.

2 Process Scheduling

2.1 Introduction to Process Scheduling

- **Multiprogramming objective**: Maximize CPU utilization by always having a process running.
- **Time sharing objective**: Switch CPU core among processes frequently for user interaction.
- **Process scheduler**: Selects an available process for execution on a core.
- Single core: one running process; multicore: multiple running processes.
- **Degree of multiprogramming**: Number of processes currently in memory.
- **Process types**:
  - **I/O-bound process**: Spends more time doing I/O than computations.
  - **CPU-bound process**: Spends more time doing computations than I/O.

2.2 Scheduling Queues

- **Scheduling Queues**:
  - **Ready queue**: Processes ready and waiting for CPU execution (linked list of PCBs).
  - **Wait queue**: Processes waiting for a specific event (e.g., I/O completion).
  - **Queueing diagram**: Visual representation of process flow (ready queue, wait queues, resources).
  - **Process flow**: New → Ready → Dispatched (Running) → (I/O wait, Child wait, Interrupt/Time slice expire → Ready) → Terminated.

2.3 CPU Scheduling

- **CPU Scheduling**:
  - **CPU scheduler** role: Selects process from ready queue, allocates CPU core.
  - Executes frequently (e.g., every 100ms or more).
  - May forcibly remove CPU from a process.

2.4 Swapping

- **Swapping**:
  - Intermediate scheduling form.
  - Remove process from memory to disk ("swapped out") to reduce degree of multiprogramming.
  - Reintroduce process to memory ("swapped in") to continue execution.
  - Typically used when memory is overcommitted.

2.5 Context Switch

- **Context Switch**:
  - Interrupts cause OS to change CPU core task to kernel routine.
  - System saves current **context** (CPU registers, process state, memory-management info) of running process in its PCB.
  - **State save**: Copying current CPU state.
  - **State restore**: Copying saved context to resume operations.
  - Context switch: Switching CPU core from one process to another (state save old, state restore new).
  - Pure overhead (no useful work done during switch).
  - Speed varies (memory speed, registers to copy, special instructions). Typical: several microseconds.
  - Hardware support (multiple register sets) can speed up.
  - More complex OS/memory management → more work during context switch.

Section glossary

Term	Definition
process scheduler	A scheduler that selects an available process (possibly from a set of several processes) for execution on a CPU.
degree of multiprogramming	The number of processes in memory.
parent	In a tree data structure, a node that has one or more nodes connected below it.
children	In a tree data structure, nodes connected below another node.
siblings	In a tree data structure, child nodes of the same parent.
I/O-bound process	A process that spends more of its time doing I/O than doing computations
CPU-bound process	A process that spends more time executing on CPU than it does performing I/O.
ready queue	The set of processes ready and waiting to execute.
wait queue	In process scheduling, a queue holding processes waiting for an event to occur before they need to be put on CPU.
dispatched	Selected by the process scheduler to be executed next.
CPU scheduler	Kernel routine that selects a thread from the threads that are ready to execute and allocates a core to that thread.
swapping	Moving a process between main memory and a backing store. A process may be swapped out to free main memory temporarily and then swapped back in to continue execution.
context	When describing a process, the state of its execution, including the contents of registers, its program counter, and its memory context, including its stack and heap.
state save	Copying a process's context to save its state in order to pause its execution in preparation for putting another process on the CPU.
state restore	Copying a process's context from its saved location to the CPU registers in preparation for continuing the process's execution.
context switch	The switching of the CPU from one process or thread to another; requires performing a state save of the current process or thread and a state restore of the other.
foreground	Describes a process or thread that is interactive (has input directed to it), such as a window currently selected as active or a terminal window currently selected to receive input.
background	Describes a process or thread that is not currently interactive (has no interactive input directed to it), such as one not currently being used by a user. In the Grand Central Dispatch Apple OS scheduler, the scheduling class representing tasks that are not time sensitive and are not visible to the user.
split-screen	Running multiple foreground processes (e.g., on an iPad) but splitting the screen among the processes.
service	A software entity running on one or more machines and providing a particular type of function to calling clients. In Android, an application component with no user interface; it runs in the background while executing long-running operations or performing work for remote processes.

3 Operations on Processes

3.1 Introduction to Process Operations

- Processes can execute concurrently; created and deleted dynamically.
- OS provides mechanisms for process creation and termination.

3.2 Process Creation

- Parent process creates child processes, forming a **tree**.
- Each process has a unique **process identifier (pid)**.
- Child resources: from OS or subset of parent's. Parent may partition/share resources.
- Parent may pass initialization data to child.
- **Execution possibilities**:
  1. Parent executes concurrently with children.
  2. Parent waits until children terminate.
- **Address-space possibilities**:
  1. Child is a duplicate of parent (same program/data).
  2. Child has a new program loaded.

3.2.1 UNIX/Linux Process Creation

- **UNIX/Linux Process Creation**:
  - `fork()`: creates new process (child) as copy of parent's address space.
  - Both parent/child continue after `fork()`.
  - `fork()` return: 0 for child, child's pid for parent.
  - `exec()`: replaces process's memory space with new program (loads binary). Does not return on success.
  - `wait()`: parent waits for child's termination.
  - Child inherits privileges, scheduling attributes, open files.

3.2.2 Windows Process Creation

- **Windows Process Creation**:
  - `CreateProcess()`: similar to `fork()`, but loads specified program into child's address space at creation.
  - Uses `STARTUPINFO` and `PROCESS_INFORMATION` structures.
  - `WaitForSingleObject()`: parent waits for child's completion.

3.3 Process Termination

- Process finishes, uses `exit()` to ask OS to delete it.
- Returns status value to waiting parent (via `wait()`).
- All process resources deallocated.
- Another process can terminate a process (e.g., `TerminateProcess()` in Windows), usually only by parent.
- **Reasons for parent terminating child**:
  - Child exceeded resource usage.
  - Child's task no longer required.
  - Parent exiting (some OS don't allow child to continue).
- **Cascading termination**: If parent terminates, all children also terminate (OS initiated).

3.3.1 UNIX/Linux Termination

- **UNIX/Linux Termination**:
  - `exit(status)`: terminates process, provides exit status.
  - `wait(&status)`: parent waits for child, gets exit status, returns child's pid.
  - **Zombie process**: Terminated process whose parent hasn't called `wait()`. Entry remains in process table. Brief state.
  - **Orphan process**: Child of a parent that terminated without calling `wait()`.
  - `init` (or `systemd`) becomes new parent for orphans, periodically calls `wait()` to clean up.

3.3.2 Android Process Hierarchy

- **Android Process Hierarchy** (for resource constraints/termination):
  - **Foreground process**: Visible, user interacting (most important).
  - **Visible process**: Not directly visible, but performing activity foreground process refers to.
  - **Service process**: Similar to background, but apparent to user (e.g., streaming music).
  - **Background process**: Performing activity, not apparent to user.
  - **Empty process**: No active components (least important).
  - Android terminates processes from least to most important to reclaim resources.
  - Process state saved before termination, resumed if user navigates back.

Section glossary

Term	Definition
tree	A data structure that can be used to represent data hierarchically; data values in a tree structure are linked through parent-child relationships.
process identifier (pid)	A unique value for each process in the system that can be used as an index to access various attributes of a process within the kernel.
cascading termination	A technique in which, when a process is ended, all of its children are ended as well.
zombie	A process that has terminated but whose parent has not yet called wait() to collect its state and accounting information.
orphan	The child of a parent process that terminates in a system that does not require a terminating parent to cause its children to be terminated.

4 Interprocess Communication (IPC)

4.1 Independent vs. Cooperating Processes

- Processes can be **independent** (no shared data) or **cooperating** (share data, affect each other).

4.2 Reasons for Process Cooperation

- **Reasons for process cooperation**:
  - **Information sharing**: Concurrent access to shared info (e.g., copy/paste).
  - **Computation speedup**: Break task into subtasks for parallel execution (requires multiple cores).
  - **Modularity**: Divide system functions into separate processes/threads.
- Cooperating processes need **IPC mechanism** to exchange data.

4.3 IPC Models

- **Two fundamental IPC models**:
  - **Shared Memory**:
    - Region of memory shared by cooperating processes.
    - Exchange info by reading/writing to shared region.
    - Faster than message passing (after setup), no kernel intervention for access.
  - **Message Passing**:
    - Communication via messages exchanged between processes.
    - Useful for smaller data amounts, no conflicts.
    - Easier in distributed systems.
    - Typically uses system calls (kernel intervention), thus slower than shared memory.

Section glossary

Term	Definition
interprocess communication (IPC)	Communication between processes.
shared memory	In interprocess communication, a section of memory shared by multiple processes and used for message passing.
message passing	In interprocess communication, a method of sharing data in which messages are sent and received by processes. Packets of information in predefined formats are moved between processes or between computers.
browser	A process that accepts input in the form of a URL (Uniform Resource Locator), or web address, and displays its contents on a screen.
renderer	A process that contains logic for rendering contents (such as web pages) onto a display.
plug-in	An add-on functionality that expands the primary functionality of a process (e.g., a web browser plug-in that displays a type of content different from what the browser can natively handle).
sandbox	A contained environment (e.g., a virtual machine).

5 IPC in Shared-Memory Systems

5.1 Shared Memory Setup

- **Shared Memory Setup**:
  - Communicating processes establish a shared memory region.
  - Region typically in address space of creator process.
  - Other processes attach it to their address space.
  - OS restriction on memory access removed by agreement.
  - Processes responsible for data form, location, and concurrent access synchronization.

5.2 Producer-Consumer Problem

- **Producer-Consumer Problem**: Common paradigm for cooperating processes.
  - **Producer** process: produces information.
  - **Consumer** process: consumes information.
  - Example: compiler (producer) → assembler (consumer).
  - Metaphor for client-server paradigm (server=producer, client=consumer).
  - Requires a buffer for items.
  - **Buffer types**:
    - **Unbounded buffer**: No practical limit on size. Consumer may wait, producer always produces.
    - **Bounded buffer**: Fixed size. Consumer waits if empty, producer waits if full.

5.3 Bounded Buffer Implementation

- **Bounded Buffer Implementation (Shared Memory Example)**:
  - Shared variables: `buffer` (circular array), `in` (next free position), `out` (first full position).
  - Buffer empty: `in == out`.
  - Buffer full: `((in + 1) % BUFFER_SIZE) == out`.
  - Allows `BUFFER_SIZE - 1` items at most.
  - Producer loop: produce item, wait if buffer full, add to buffer, update `in`.
  - Consumer loop: wait if buffer empty, consume item, update `out`.
  - **Issue**: Concurrent access to shared buffer requires synchronization (covered in later chapters).

Section glossary

Term	Definition
producer	A process role in which the process produces information that is consumed by a consumer process.
consumer	A process role in which the process consumes information produced by a producer process.
unbounded buffer	A buffer with no practical limit on its memory size.
bounded buffer	A buffer with a fixed size.

6 IPC in Message-Passing Systems

6.1 Introduction to Message Passing

- Message passing: allows processes to communicate and synchronize without shared address space.
- Useful in distributed environments (processes on different computers).
- Provides 'send(message)' and 'receive(message)' operations.
- Messages can be fixed or variable size (trade-off: implementation complexity vs. programming ease).
- **Communication Link**: Must exist between communicating processes.
- **Logical Implementation Methods**:
  - Direct or indirect communication.
  - Synchronous or asynchronous communication.
  - Automatic or explicit buffering.

6.2 Naming in Message Passing

- **Naming**: How processes refer to each other.
  - **Direct Communication**: Explicitly name recipient/sender.
    - 'send(P, message)': Send to process P.
    - 'receive(Q, message)': Receive from process Q.
    - Properties: Link established automatically, exactly two processes per link, exactly one link per pair.
    - **Symmetry**: Both sender/receiver name each other.
    - **Asymmetry**: Only sender names recipient ('receive(id, message)' from any process).
    - Disadvantage: Limited modularity, **hard-coding** identifiers is undesirable.
  - **Indirect Communication**: Messages sent/received via **mailboxes** (or **ports**).
    - Mailbox: Object for placing/removing messages, unique ID.
    - 'send(A, message)': Send to mailbox A.
    - 'receive(A, message)': Receive from mailbox A.
    - Properties: Link only if shared mailbox, link may be associated with ¿2 processes, multiple links per pair.
    - Handling multiple receivers for one message:
      - Link associated with $\leq$ two processes.
      - At most one process executes 'receive()' at a time.
      - System arbitrarily selects receiver (e.g., **round robin**).
    - Mailbox ownership:
      - By process: Part of process address space. Owner receives, user sends. Disappears on owner termination.
      - By OS: Independent existence. OS provides create/send/receive/delete mechanisms. Can have multiple receivers.

6.3 Synchronization in Message Passing

- **Synchronization**: 'send()' and 'receive()' primitives.
  - **Blocking** (synchronous):
    - **Blocking send**: Sender blocked until message received by receiver/mailbox.
    - **Blocking receive**: Receiver blocked until message available.
  - **Nonblocking** (asynchronous):
    - **Nonblocking send**: Sender sends message, resumes operation.
    - **Nonblocking receive**: Receiver retrieves valid message or null.
  - **Rendezvous**: Both 'send()' and 'receive()' are blocking.

6.4 Buffering in Message Passing

- **Buffering**: Temporary queue for messages.
  - **Zero capacity**: Queue length 0. Sender blocks until recipient receives. No buffering.
  - **Bounded capacity**: Finite length n. Sender blocks if queue full. Automatic buffering.
  - **Unbounded capacity**: Potentially infinite length. Sender never blocks. Automatic buffering.

Section glossary

Term	Definition
direct communication	In interprocess communication, a communication mode in which each process that wants to communicate must explicitly name the recipient or sender of the communication.
blocking	In interprocess communication, a mode of communication in which the sending process is blocked until the message is received by the receiving process or by a mailbox and the receiver blocks until a message is available. In I/O, a request that does not return until the I/O completes.
nonblocking	A type of I/O request that allows the initiating thread to continue while the I/O operation executes. In interprocess communication, a communication mode in which the sending process sends the message and resumes operation and the receiver process retrieves either a valid message or a null if no message is available. In I/O, a request that returns whatever data is currently available, even if it is less than requested.
synchronous	In interprocess communication, a mode of communication in which the sending process is blocked until the message is received by the receiving process or by a mailbox and the receiver blocks until a message is available. In I/O, a request that does not return until the I/O completes.
asynchronous	In I/O, a request that executes while the caller continues execution.
rendezvous	In interprocess communication, when blocking mode is used, the meeting point at which a send is picked up by a receive.
communication link	A link between processes that allows them to send messages to and receive messages from each other.
symmetry	In direct communication, a scheme in which both the sender process and the receiver process must name the other to communicate.
asymmetry	In direct communication, a scheme in which only the sender names the recipient; the recipient is not required to name the sender.
hard-coding	Techniques where identifiers must be explicitly stated.
indirect communication	A communication mode in which messages are sent to and received from mailboxes, or ports.
mailboxes	In indirect communication, objects into which messages can be placed by processes and from which messages can be removed.
ports	In indirect communication, objects into which messages can be placed by processes and from which messages can be removed.
round robin	An algorithm for selecting which process will receive a message (e.g., processes take turns receiving messages).

7 Examples of IPC Systems

7.1 Introduction to IPC Examples

- Explores four IPC systems: POSIX shared memory, Mach message passing, Windows IPC, Pipes.

7.2 POSIX Shared Memory

- **POSIX Shared Memory**:
  - Organized using memory-mapped files.
  - `shm_open(name, O_CREAT | O_RDWR, 0666)`: Creates/opens shared-memory object, returns file descriptor.
  - `ftruncate(fd, size)`: Configures size of object in bytes.
  - `mmap()`: Establishes memory-mapped file, returns pointer for access.
  - Producer-Consumer model example: Producer writes, Consumer reads.
  - `MAP_SHARED` flag: Changes visible to all sharing processes.
  - Writing: Use `sprintf()` to pointer, increment pointer by bytes written.
  - Consumer uses `shm_unlink(name)` to remove segment after access.

7.3 Mach Message Passing

- **Mach Message Passing**:
  - Designed for distributed systems (used in MacOS, iOS).
  - Supports **tasks** (like processes, but with multiple threads, fewer resources).
  - Communication via **messages** sent to/received from **ports** (mailboxes).
  - Ports: finite size, unidirectional. Two-way communication needs separate **reply** port.
  - Each port: multiple senders, one receiver.
  - Uses **kernel abstractions** for resources (tasks, threads, memory, processors).
  - **Port rights**: Capabilities for task interaction (e.g., `MACH_PORT_RIGHT_RECEIVE`).
  - Creator of port is owner, only owner can receive. Owner can manipulate capabilities.
  - Ownership at task level: all threads in same task share port rights.
  - Special ports on task creation: **Task Self** (kernel has receive rights, task sends to kernel), **Notify** (kernel sends notifications to task).
  - `mach_port_allocate()`: Creates new port, allocates queue space, identifies rights. Port rights are **names** (integer values, like UNIX file descriptors).
  - **Bootstrap port**: Allows task to register port with system-wide **bootstrap server**. Other tasks can look up and obtain send rights.
  - Port queue: finite, initially empty. Messages copied into queue.
  - Messages delivered reliably, same priority. FIFO for same sender, no absolute ordering across senders.
  - **Mach Message Fields**:
    - Fixed-size header: metadata (size, source/destination ports).
    - Variable-sized body: data.
  - **Message types**:
    - **Simple**: Unstructured user data, not interpreted by kernel.
    - **Complex**: Pointers to memory (out-of-line data) or transferring port rights. Useful for large data (avoids copying).
  - `mach_msg()`: Standard API for send/receive (`MACH_SEND_MSG` or `MACH_RCV_MSG`).
  - Flexible send/receive operations:
    1. Wait indefinitely if queue full.
    2. Wait at most n milliseconds.
    3. Return immediately (do not wait).
    4. Temporarily cache message (for server tasks, allows sending reply even if client port full).
  - Performance: Mach avoids message copying by mapping sender's address space into receiver's (virtual memory techniques). Works for intrasystem messages.

7.4 Windows IPC

- **Windows IPC**:
  - Modern design, modularity. Supports multiple **subsystems**.
  - Applications are clients of subsystem servers, communicate via message passing.
  - **Advanced Local Procedure Call (ALPC)**: Message-passing facility for same-machine communication. Optimized RPC.
  - Uses **port object** for connection. Two types: **connection ports** (published by server, visible) and **communication ports** (private, client-server pair).
  - Server creates channel (pair of communication ports) upon client connection request.
  - Channels support callback mechanism.
  - **ALPC Message-Passing Techniques**:
    1. Small messages ($\leq$ 256 bytes): copied via port's message queue.
    2. Larger messages: passed through **section object** (shared memory region).
    3. Very large data: API allows server to read/write directly into client's address space.
  - ALPC not part of Windows API; applications use standard RPC, which is handled indirectly by ALPC for same-system calls. Kernel services also use ALPC.

7.5 Pipes

- **Pipes**: Conduit for two processes to communicate. Early UNIX IPC.
  - **Considerations**:
    1. Bidirectional or unidirectional?
    2. If two-way, half duplex or full duplex?
    3. Parent-child relationship required?
    4. Network communication or same machine only?

7.5.1 Ordinary Pipes

- **Ordinary Pipes** (UNIX) / **Anonymous Pipes** (Windows):
  - Unidirectional (producer writes to **write end**, consumer reads from **read end**). Two pipes for two-way.
  - UNIX: `pipe(int fd[])` creates pipe (`fd[0]` read, `fd[1]` write). Accessed via `read()`, `write()`.
  - UNIX: Cannot be accessed outside creator process. Parent creates, child inherits via `fork()`. Parent/child close unused ends.
  - Windows: `CreatePipe()` creates. `ReadFile()`, `WriteFile()`.
  - Windows: Requires explicit inheritance (`SECURITY_ATTRIBUTES`, `SetHandleInformation`). Redirect child's standard input/output.
  - Both: Require parent-child relationship. Same machine only.

7.5.2 Named Pipes

- **Named Pipes** (UNIX FIFOs):
  - More powerful than ordinary pipes.
  - Bidirectional communication, no parent-child relationship required.
  - Several processes can use. Continue to exist after processes terminate.
  - UNIX FIFOs: Created with `mkfifo()`, appear as files. Manipulated with `open()`, `read()`, `write()`, `close()`. Exist until explicitly deleted.
  - UNIX FIFOs: Half-duplex only. Same machine only (use sockets for intermachine).
  - Windows Named Pipes: Full-duplex. Communicating processes can be on same or different machines.
  - Windows Named Pipes: Byte- or message-oriented data. Created with `CreateNamedPipe()`, client connects with `ConnectNamedPipe()`. Communication with `ReadFile()`, `WriteFile()`.

Section glossary

Term	Definition
message	In networking, a communication, contained in one or more packets, that includes source and destination information to allow correct delivery. In message-passing communications, a packet of information with metadata about its sender and receiver.
port	A communication address; a system may have one IP address for network connections but many ports, each for a separate communication. In computer I/O, a connection point for devices to attach to computers. In software development, to move code from its current platform to another platform (e.g., between operating systems or hardware systems). In the Mach OS, a mailbox for communication.
bootstrap port	In Mach message passing, a predefined port that allows a task to register a port it has created.
bootstrap server	In Mach message passing, a system-wide service for registering ports.
advanced local procedure call (ALPC)	In Windows OS, a method used for communication between two processes on the same machine.
connection port	In Windows OS, a communications port used to maintain connection between two processes, published by a server process.
communication port	In Windows OS, a port used to send messages between two processes.
section object	The Windows data structure that is used to implement shared memory.
pipe	A logical conduit allowing two processes to communicate.
write end	In ordinary pipes, the end of the pipe to which the producer writes.
read end	In ordinary pipes, the end of the pipe from which the consumer reads.
anonymous pipes	Ordinary pipes on Windows systems.

8 Communication in Client-Server Systems

8.1 Introduction to Client-Server Communication

- IPC techniques (shared memory, message passing) used in client-server systems.
- Focus on **sockets** and **Remote Procedure Calls (RPCs)**.

8.2 Sockets

- **Sockets**:
  - Endpoint for communication.
  - Pair of sockets for network communication (one per process).
  - Identified by IP address + port number.
  - Client-server architecture: Server listens on specified port, accepts client connection.
  - **Well-known ports**: Below 1024 (e.g., SSH 22, FTP 21, HTTP 80).
  - Client assigned arbitrary port ¿ 1024.
  - Connection: unique pair of sockets (client IP:port, server IP:port).
  - Java socket types:
    - **Connection-oriented (TCP)**: 'Socket' class.
    - **Connectionless (UDP)**: 'DatagramSocket' class.
    - 'MulticastSocket': subclass of 'DatagramSocket', sends to multiple recipients.
  - Example: Date server (TCP) listens on port 6013.
    - Server creates 'ServerSocket', 'accept()' blocks until client connects.
    - 'accept()' returns 'Socket' for communication.
    - Server uses 'PrintWriter' to write to socket ('println()').
    - Client creates 'Socket', connects to server IP:port.
    - Client reads from socket using stream I/O.
    - **Loopback** (127.0.0.1): refers to self, allows client/server on same host to communicate via TCP/IP.
  - Sockets are low-level: unstructured stream of bytes. Application imposes data structure.

8.3 Remote Procedure Calls (RPCs)

- **Remote Procedure Calls (RPCs)**:
  - Abstracts procedure-call mechanism for network connections. Built on IPC.
  - Messages are well-structured (function identifier, parameters).
  - Addressed to RPC daemon listening on a **port** on remote system.
  - Client invokes remote procedure as if local.
  - **Stub**: Client-side code that hides communication details.
    - Locates server port, **marshals** parameters (converts to machine-independent format).
    - Transmits message to server.
    - Server-side stub receives, invokes procedure.
    - Return values passed back similarly.
    - Windows: Stub code from **Microsoft Interface Definition Language (MIDL)**.
  - **Parameter marshaling**: Handles data representation differences (e.g., **big-endian** vs. **little-endian**).
    - Uses machine-independent representation (e.g., **External Data Representation (XDR)**).
    - Client: machine-dependent → XDR. Server: XDR → machine-dependent.
  - **Semantics of call**: RPCs can fail/duplicate due to network errors.
    - **Exactly once**: OS ensures message acted on once. Harder to implement. This typically involves the server implementing an "at most once" semantic and acknowledging the client, with the client resending until an acknowledgment is received.
    - **At most once**: Attach timestamp to message. Server keeps history, ignores repeated timestamps.
  - **Binding**: Client needs to know server port numbers.
    - Predetermined: Fixed port addresses at compile time. Less flexible.
    - Dynamic: **Rendezvous** (or **matchmaker**) daemon on fixed RPC port. Client requests port address from daemon. More flexible.
  - Useful for distributed file systems (RPC daemons/clients, messages for disk operations).

8.3.1 Android RPC

- **Android RPC**:
  - IPC between processes on same system via **binder** framework.
  - **Application component**: Basic building block for Android app.
  - **Service**: Application component with no UI, runs in background (e.g., playing music).
  - 'bindService()': client binds to service.
  - 'onBind()': invoked on service, returns 'Messenger' (for message passing) or interface (for RPC).
  - 'Messenger' service: one-way. Client provides 'replyTo' field for service to reply.
  - RPCs: 'onBind()' returns interface representing remote methods.
  - Uses Android Interface Definition Language (AIDL) to create stub files (client interface).
  - Android binder framework handles parameter marshaling, transfer, invocation.

Section glossary

Term	Definition
socket	An endpoint for communication. An interface for network I/O.
connection-oriented socket (TCP)	In Java, a mode of communication.
connectionless socket (UDP)	In Java, a mode of communication.
loopback	Communication in which a connection is established back to the sender.
port	A communication address; a system may have one IP address for network connections but many ports, each for a separate communication. In computer I/O, a connection point for devices to attach to computers. In software development, to move code from its current platform to another platform (e.g., between operating systems or hardware systems). In the Mach OS, a mailbox for communication.
stub	A small, temporary place-holder function replaced by the full function once its expected behavior is known.
Microsoft Interface Definition Language	The Microsoft text-based interface definition language; used, e.g., to write client stub code and descriptors for RPC.
big-endian	A system architecture in which the most significant byte in a sequence of bytes is stored first.
little-endian	A system architecture that stores the least significant byte first in a sequence of bytes.
external data representation	A system used to resolve differences when data are exchanged between big- and little-endian systems.
matchmaker	A function that matches a caller to a service being called (e.g., a remote procedure call attempting to find a server daemon).
binder	In Android RPC, a framework (system component) for developing object-oriented OS services and allowing them to communicate.
application component	In Android, a basic building block that provides utility to an Android app.
service	A software entity running on one or more machines and providing a particular type of function to calling clients. In Android, an application component with no user interface; it runs in the background while executing long-running operations or performing work for remote processes.

9 Summary

- A process is a program in execution, and the status of the current activity of a process is represented by the program counter, as well as other registers.
- The layout of a process in memory is represented by four different sections: (1) text, (2) data, (3) heap, and (4) stack.
- As a process executes, it changes state. There are four general states of a process: (1) ready, (2) running, (3) waiting, and (4) terminated.
- A process control block (PCB) is the kernel data structure that represents a process in an operating system.
- The role of the process scheduler is to select an available process to run on a CPU.
- An operating system performs a context switch when it switches from running one process to running another.
- The `fork()` and `CreateProcess()` system calls are used to create processes on UNIX and Windows systems, respectively.
- When shared memory is used for communication between processes, two (or more) processes share the same region of memory. POSIX provides an API for shared memory.
- Two processes may communicate by exchanging messages with one another using message passing. The Mach operating system uses message passing as its primary form of interprocess communication. Windows provides a form of message passing as well.
- A pipe provides a conduit for two processes to communicate. There are two forms of pipes, ordinary and named. Ordinary pipes are designed for communication between processes that have a parent-child relationship. Named pipes are more general and allow several processes to communicate.
- UNIX systems provide ordinary pipes through the `pipe()` system call. Ordinary pipes have a read end and a write end. A parent process can, for example, send data to the pipe using its write end, and the child process can read it from its read end. Named pipes in UNIX are termed FIFOs.
- Windows systems also provide two forms of pipes—anonymous and named pipes. Anonymous pipes are similar to UNIX ordinary pipes. They are unidirectional and employ parent-child relationships between the communicating processes. Named pipes offer a richer form of interprocess communication than the UNIX counterpart, FIFOs.
- Two common forms of client-server communication are sockets and remote procedure calls (RPCs). Sockets allow two processes on different machines to communicate over a network. RPCs abstract the concept of function (procedure) calls in such a way that a function can be invoked on another process that may reside on a separate computer.
- The Android operating system uses RPCs as a form of interprocess communication using its binder framework.