Notes - Algorithms & Data Structures - System and Memory Structure

- Dr Nick Hayward

A brief intro to system and memory structure for app development.

Contents

Intro

A consideration of system structure, management, and usage applicable to app development assumes the host computer has successfully completed a series of initial tests and initialisations.

For example, we may start with a traditional *power-on self test*, or POST, and then consider the *BOOTSTRAP* loaders for the system.

System components

A computer system includes major components focused in the *kernel*, which allow the system to focus on control, execution, management &c.

For example, an OS may include the following major components

- process manager
- memory manager
- file system manager
- device or I/O manager

Each of these managers will form part of the kernel's role in the system.

System - intro

After running various POST tests, bootstrapping the system, and loading the *kernel*, we may consider a system from a working, program perspective.

For example, how does the system manage a user request to a program, or the various processes running throughout the system. What is the role of the kernel relative to the CPU, and so on.

system - app starts

As a user interacts with a program, e.g. a basic text-editing app, the app, where applicable, will send a request to the *system call* API.

Common system call types include

- communication
- device management
- file management
- information management
- process control

This API abstracts system functions to a uniform interface for apps and programs. These calls will then be converted by the *kernel* to executable functons such as the following examples for files

- create, delete, open, close a file
- rename, copy a file
- read data from and write data to a file
- append data to end of specified file
- delete file contents (truncate content) - file remains with content wiped
- reposition file pointer in defined file

Processes - intro

Processes are a key part of system structure and usage.

For example, as a program runs in a system it is one of many other *processes* currently active and being monitored and controlled by the *kernel*.

process creation and management

So, an app is a process running in the system, which is managed and controlled by the system's *kernel*.

Each process is a child of one parent, for example the `init` process in a Unix system.

This `init` process is created at system boot, and acts as the root process for all child processes in a running system.

It also plays a crucial role in simplifying many API calls for child processes.

The `init` process may also signal a special system call to create a new child process.

kernel and `init`

As the *kernel* loads as part of the system's bootstrap loader, it starts the parent `init` process.

The `init` process may then create and manage any required child processes, which include a few simple and small system calls.

This focus on simplicity and efficiency increases flexibility in the system. It allows the parent process to execute some programs for a child process.

`init` parent and `fork()`

So, all child processes in a system are descendants of the same parent `init` process.

`init` executes the `fork()` function to create a *child* process.

As the new process is created, it is assigned a new address space, e.g. to store the process ID for reference.

Any necessary *text*, *data*, and a *stack* data structure are added to this new address space.

Each child process will also have access to open files, i.e. relative to a process' access privileges in the current system.

child process and `exec()`

The `exec()` function allows a child process to execute any required programs in the assigned address space.

In effect, the process is able to execute the program in its own address, creating the effect of multiple, separate running programs.

Each program appears to be running in its own system.

`init` parent and `wait()`

A system's parent process, `init`, may call the `wait()` function to allow a child process to terminate gracefully.

In effect, the parent process is simply waiting for a child to finish.

inter-process communication

A child process may create and setup inter-process communication with the parent `init` process.

This communication will, commonly, be established prior to a call to the `exec()` function for a new program in the child process.

For example, a child might establish a *pipe* to the parent to call *printing*.

With this communication established, printing from the child process is now through a pipe to the parent `init` process.

kernel and communication

This communication may also support synchronisation between system processes.

This synchronisation is tested and managed by the kernel using one of the following classic sync problems. For example,

- shared buffer
- producers and consumers (bounded buffer)
- readers and writers
- dining philosophers
- cigarette smokers
- barbershop

However, a *locking* mechanism is required to coordinate access to shared memory resources.

Such locking requests may not be initiated by the process itself, they must be implemented and managed by the kernel. The kernel may also implement locking for signals to and from a process.

The kernel may also manage *monitors* (semaphores), which are commonly used for a system's locking mechanism.

Other inter-process facilities include,

- data transfer (e.g. pipes, shared buffers, files &c.)
- shared memory access and usage
- messaging (e.g. signals, `send()` & `receive()` messages &c.)

kernel and PCB

One of the underlying jobs of a system's *kernel* is to manage these processes.

The kernel, therefore, provides an *abstract* or *virtual* machine for each process. As noted above, this allows a process to perceive the system as their own, in isolation for execution and running programs.

A *process control block*, or PCB, allows this isolation of process details. The kernel is then responsible for, and controls, switching between available active processes.

So, each process is managed by the kernel with details from the matching PCB.

One PCB per process in the system, and all PCBs stored in an array in the kernel's memory

Each PCB is then indexed by a process ID, which allows it to store the required details for the unique process. Details stored in a PCB commonly include the following

- register values
- logical state
- page map table mapping logical addresses to physical addresses
- type & location of resources it holds
- list of resources it needs
- parent process identification
- security keys
- ...

process and memory usage

As a child process is created, it is assigned an address memory space.

Each process will see their memory space as a contiguous logical unit.

However, such memory address might actually be separate across the system. There will be disparate addressed memory spaces for each process, which may then be organised together, as needed, by the system's kernel.

For example, separate memory stores and addresses organised into a contiguous group per process.

One obvious benefit is the efficient use of memory space. There is no need for pre-assigned large chunks of memory or, perhaps, reserved memory that is never used by a process.

The kernel controls access for a process to memory addresses. In effect, the kernel is controlling the conversion of assigned virtual addresses to a physical address in the system's hardware memory.

process and state

Each child process may have a related *state*, associated during the lifetime of the process itself.

This state may be monitored by the system's *kernel*, and a process will wait until resources are available to allow a change in state.

The kernel may then switch processes relative to an update in a process' state.

Process manager - intro

The *process manager* is responsible for processes in a system.

It is controlled by the system's *kernel*, and manages the following

- process creation and termination
- resource allocation and protection
- cooperation with device manager to implement I/O
- implementation of address space
- process scheduling

process scheduling

As noted above, a key part of managing processes in a system is efficient scheduling.

Whilst such scheduling is part of the *process manager*, it is actually maintained by the system's kernel.

The kernel is responsible for switching between processes, checking and migrating available *ready* state processes to execution in an *active* state.

In effect, the *kernel* is selecting processes to execute in the system on the available CPU. So, the *kernel* is choosing the next process to run on the CPU.

This context switch is informed by the required and available *process properties*.

This selection of process is also determined by the nature of the process itself, i.e. is it I/O bound or CPU bound.

An algorithm helps determine the best process choice to ensure a system runs efficiently and without apparent delays. Example algorithms include,

- first-come, first-served
- shortest job next
- round robin
- multi-level priority queue

So, scheduling is meant to provide a fast and efficient system. The kernel chooses processes to allow the system to run fast. For example, it is common to assign priroty to a *front-facing* process over one running in the *background*.

scheduler metrics

Metrics may also be defined and recorded to help with system diagnostics, in particular relative to performance and resource allocation.

For example, we might record the following

- CPU utilisation
- throughput
- waiting time for process in ready state
- service time by CPU for a given process
- turnaround time for a process
- response time (time from first submission of process to completion...)

System-call API

After receiving a request from a program running in a process, the *system-call* API may signal the *kernel* using a software *interrupt*.

Any parameters, where applicable, may be passed to the kernel using a register. If these parameters are required by the kernel, it may simply load them using the address provided as part of the interrupt signal.

API categories

The system-call API commonly includes five categories,

- communication
- device management
- file management
- information management
- process control

A program may call part of the API to query data, request a file, halt a process, print some text to a connected printer, and so on.

The functionality in the API is abstracted to the system level, and not to a specific program.

In effect, such categories in the system-call API define what an application may signal to the kernel.

API and kernel

A system's *kernel* may receive a system call from one of the available categories. For example, an interrupt signal to read some data.

The kernel converts this system-call request to a function, which it may then pass to the CPU for execution.

Such function code and instructions should be restricted to the kernel, and may be executed using a supervisory (kernel) mode.

This function code should not be directly accessible to either the system-call API or the originating program.

So, whilst the function code and instructions are *fetched* by the kernel, they are *executed* by the system's CPU.

kernel scheduling

As noted above, a system's kernel is also responsible for scheduling processes, and their access to resources and CPU time.

After moving a process from a *ready state* to a *running state*, the kernel sends the address of the instruction to execute to the CPU.

This signal from the kernel is then received and managed by the *control unit* (CU) in the CPU.

Kernel management

A system's kernel is responsible for management, including processes, as noted above, hardware, CPU access, and so on.

memory and data stores

The kernel is responsible for managing *memory*, and a process' access to such stores.

These stores may commonly include

- registers (e.g. CPU)
- cache (level 1 and 2)
- main memory - RAM

It is also responsible for controlling memory locations, and process access to such addresses.

Part of this management includes synchronisation of access to ensure a process does not block or interfere with another process.

To helo with this sychronisation of memory, the kernel must ensure that each process may access its defined memory addresses.

CPU and physical data stores

The CPU is connected to the kernel, and controls the system's physical memory stores.

A memory management unit (MMU) is included in most CPUs. The CPU converts a logical address from the kernel to a physical address in the system's available data stores.

To help with this conversion, the MMU has two registers

- memory data register (MDR)
- memory address register (MAR)

Data retrieved from hardware memory is temporarily stored in the MDR register.

The MAR register is used to store logical memory addresses. It uses a *memory map* from the OS to help map logical to physical addresses.

So, the *kernel* is responsible for loading appropriate data into the *MMU* as a process is started.

The kernel may request data from the CPU, and return it to program accessible memory.

kernel and file system

The kernel is also responsible for managing file system usage, its structure, access, process privileges, &c.

This may include data storage on a physical device, which requires *mapping* of block storage to a logical view.

The block of data will be stored on the physical device, and then mapped to *logical views* for use in processes, including programs.

For example, a file is an ordered collection of *blocks*.

The kernel is tasked to allocate and deallocate storage in the overall system.

It may provide, for example, file system directories to the overall system and its programs.

To help with this management, and program access, the system-call API for files commonly includes the following

- create, delete, open, close a file
- rename, copy a file
- read data from and write data to a file
- append data to end of specified file
- delete file contents (truncate content) - file remains with content wiped
- reposition file pointer in defined file

The kernel is also responsible for *disk scheduling* relative to memory access and usage. It uses various algorithms to maintain speed and efficiency.

System requests for disk access are aggregated, and then processed using various algorithms, including

- first-come, first-served
- shortest seek time first
- scan

CPU and kernel

As a process is switched from a *ready state* to a *running state*, the kernel sends the memory address of the instruction to execute to the CPU.

This *interrupt* signal is then received and managed by the *control unit* (CU) in the CPU.

The CU is directly connected to the main memory for most systems.

The CU fetches the required address from the CPU's registers, and then accesses the instructions at the specified memory address.

The current instructions for execution will be held in the CPU's *instruction register* (IR). The next set of instructions will be held in the *instruction pointer* (program counter) ready for execution by the CPU.

control unit (CU)

So, one of the jobs of the CU is to manage data between current and next instruction sets, as defined by the instruction register and instruction pointer.

For example, the CU may send instructions to the CPU's *arithmetic and logic unit* (ALU) or, perhaps, some data back to main memory for access by the kernel.

The CU is also responsible for ensuring that data is sent to the correct place in the system, relative to the instruction set requirements. For example, data might be sent back to main memory or a given system register.

The CU will also send a notification signal back to the kernel, commonly an update on execution handling or errors.

The CU may also monitor signals in and out of the CPU.

CPU, fetch, and execute

The CU and ALU often work as a complementary pair in the system's CPU.

The CU will fetch an instruction set, some data &c., and then pass these instructions and data to the ALU for logic or arithmetic processing.

The instruction set from the kernel to the CPU is controlled by the CU, and then executed by the ALU.

The role of both units is relstive to kernel signals, instructions, and specific data.