Real OS APIs for Process and Thread Control

Week 11

SDB

Autumn 2025

Agenda: Process and Thread Management in Modern OS

Understanding How OS Interfaces Enable Concurrency

1. **Linux Process Management: ‘fork()‘, ‘exec()‘, ‘wait()‘** *(Classical Unix process model)*
2. **Linux Threading: ‘pthread‘ and ‘clone()‘** *(User-space vs. kernel-level threading)*
3. **Linux Scheduling Policies** *(Controlling CPU allocation)*
4. **CPU Affinity and Priority Control in Linux** *(Fine-tuning performance)*
5. **Windows Process & Thread APIs** *(A different approach to concurrency)*
6. **Windows Scheduling and Priority** *(How Windows manages execution)*

Agenda: Process and Thread Management in Modern OS

Understanding How OS Interfaces Enable Concurrency

1. **Linux Process Management: ‘fork()‘, ‘exec()‘, ‘wait()‘** *(Classical Unix process model)*
2. **Linux Threading: ‘pthread‘ and ‘clone()‘** *(User-space vs. kernel-level threading)*
3. **Linux Scheduling Policies** *(Controlling CPU allocation)*
4. **CPU Affinity and Priority Control in Linux** *(Fine-tuning performance)*
5. **Windows Process & Thread APIs** *(A different approach to concurrency)*
6. **Windows Scheduling and Priority** *(How Windows manages execution)*

Think Ahead: OS as a Resource Manager

We’ve discussed processes and threads conceptually, and synchronization. Now, how do we *create* and *control* them programmatically using operating system services? What tools do OSes provide to manage their execution, priorities, and CPU allocation to achieve desired performance and responsiveness?

Recap: Processes vs. Threads (API Perspective)

Before diving into specific APIs, let's briefly recap the key differences from a practical standpoint:

Feature	Process	Thread
Isolation	**High:** Independent address space, file descriptors (copied), signal handlers. Strong fault isolation.	**Low:** Shares address space, file descriptors, signal handlers with other threads in the same process.
Creation Cost	High (copying/setting up new address space, PCB)	Lower (only thread control block, stack, registers)
Communication	IPC mechanisms (pipes, message queues, shared memory, sockets)	Shared memory, mutexes, condition variables, semaphores
Switching Cost	High (TLB flush, cache invalidation)	Lower (context switch within same address space)
OS Support	Managed by OS kernel, scheduled independently	Managed by OS kernel (kernel-level threads) or user-level library (user-level threads)
Termination	'exit()' ends process, all threads terminated.	'pthread_exit()' (Linux), 'ExitThread()' (Windows) terminates thread; process continues if other threads exist.

OS APIs reflect these differences by offering distinct functions for process

Linux Process Management: ‘fork()‘, ‘exec()‘, ‘wait()‘ I

The Unix/Linux model for process creation is based on the ‘fork-exec‘ paradigm.

1. ‘fork()‘: Create a Child Process (Clone)

```c
#include <unistd.h> // For fork, exec, wait
#include <stdio.h>  // For printf
#include <sys/wait.h> // For wait

int main() {
    printf("Parent process (PID: %d) starting.\n", getpid());
    pid_t pid = fork(); // Creates a child process

    if (pid == -1) {
        // Error handling for fork failure
        perror("fork failed");
        return 1;
    } else if (pid == 0) {
        // This code runs in the CHILD process
        printf("Child process (PID: %d, Parent PID: %d) is running.\n", getpid(), getppid
            ());
        // Child will execute a new program
        // execlp searches PATH for the executable
        printf("Child is replacing itself with 'ls -l'...\n");
        execlp("ls", "ls", "-l", NULL);
        // execlp only returns if an error occurs
        perror("exec failed");
        _exit(127); // Use _exit in child after exec error
    } else {
        // This code runs in the PARENT process
        printf("Parent process (PID: %d) created child with PID: %d.\n", getpid(), pid);
```

Linux Process Management: ‘fork()‘, ‘exec()‘, ‘wait()‘ II

```
        int status;
        // wait() blocks parent until one child terminates
        // waitpid() allows waiting for a specific child or non-blocking wait
        wait(&status); // Wait for any child to terminate
        printf("Child (PID: %d) terminated with status %d.\n", pid, WEXITSTATUS(status));
    }
    printf("Process (PID: %d) exiting.\n", getpid());
    return 0;
}
```

Explanation:

- ‘fork()‘: Creates an almost identical **copy** of the calling process (the parent).
  - Parent process: ‘fork()‘ returns the PID of the child.
  - Child process: ‘fork()‘ returns 0.
  - Return -1 on error.
  - Employs **Copy-on-Write (CoW)**: Memory pages are shared until one process tries to modify them, then a copy is made.

Linux Process Management: ‘fork()‘, ‘exec()‘, ‘wait()‘ III

- ‘exec()‘ (e.g., ‘execlp‘, ‘execvp‘): Replaces the current process’s memory image (code, data, heap, stack) with a **new program**. The PID remains the same.
  - Variants (‘execv‘, ‘execl‘, ‘execve‘, etc.) differ in how arguments are passed and whether environment variables are used.
  - Crucial for running external commands or applications from within a program.
- ‘wait()‘ / ‘waitpid()‘: Parent process waits for a child process to terminate.
  - Prevents ”zombie” processes (terminated children whose parent hasn’t collected their exit status).
  - ‘waitpid()‘ offers more control (wait for specific PID, non-blocking option ‘WNOHANG‘).

Linux User-Level Threading: ‘pthread‘ API I

The POSIX Threads (pthreads) API provides a standardized interface for creating and managing threads in C/C++ on Unix-like systems.

```c
#include <pthread.h> // For pthreads API
#include <stdio.h>   // For printf
#include <stdlib.h>  // For exit

// Function that the new thread will execute
void* thread_work(void* arg) {
    char* message = (char*)arg;
    printf("Thread ID: %lu - Message: %s\n", pthread_self(), message);
    pthread_exit(NULL); // Terminate the thread
}

int main() {
    pthread_t thread_id; // Thread ID handle
    char* msg = "Hello from a new thread!";
    int ret;

    printf("Main thread ID: %lu\n", pthread_self());

    // Create a new thread
    // Arg 1: Pointer to pthread_t variable to store new thread's ID
    // Arg 2: Pointer to pthread_attr_t for thread attributes (NULL for default)
    // Arg 3: Function pointer to the thread's entry point
    // Arg 4: Argument to pass to the thread function (void*)
    ret = pthread_create(&thread_id, NULL, thread_work, (void*)msg);
    if (ret != 0) {
        fprintf(stderr, "Error creating thread: %d\n", ret);
```

Linux User-Level Threading: ‘pthread‘ API II

```
        return 1;
    }

    // Wait for the created thread to terminate
    // Arg 1: The pthread_t ID of the thread to wait for
    // Arg 2: Pointer to store the thread's return value (NULL if not needed)
    ret = pthread_join(thread_id, NULL);
    if (ret != 0) {
        fprintf(stderr, "Error joining thread: %d\n", ret);
        return 1;
    }

    printf("Thread %lu finished. Main thread exiting.\n", thread_id);
    return 0;
}
```

Key ‘pthread‘ Functions and Concepts:

- ‘pthread_create()‘: Creates a new thread. The function ‘thread_work‘ starts executing in the new thread.
- ‘pthread_join()‘: Blocks the calling thread (e.g., ‘main‘) until the specified thread terminates. Used for synchronization and collecting return values.

Linux User-Level Threading: ‘pthread‘ API III

- ‘pthread_detach()‘: Detaches a thread, meaning its resources will be automatically reclaimed upon termination. A detached thread cannot be ‘join()‘ed.
- ‘pthread_exit()‘: Terminates the calling thread. The thread's return value can be passed to ‘pthread_join()‘.
- ‘pthread_self()‘: Returns the ID of the calling thread.
- ‘pthread_attr_t‘: A structure to specify thread attributes (e.g., stack size, scheduling policy, detached state) before creation using functions like ‘pthread_attr_init()‘, ‘pthread_attr_setdetachstate()‘, etc.

Pthreads manage threads largely in user space but map to kernel-level threads for scheduling.

Linux Low-Level Threading: ‘clone()‘ System Call I

While ‘pthread_create()‘ is the standard way to create threads, it’s typically implemented on Linux using the more powerful and flexible ‘clone()‘ system call.

What is ‘clone()‘?

- A Linux-specific system call that creates a new process (or thread) with a high degree of control over what resources are shared with the parent.
- It’s more primitive than ‘fork()‘ because ‘fork()‘ always clones everything, whereas ‘clone()‘ allows selective sharing.
- ‘pthread_create()‘ is a library function that wraps ‘clone()‘ with specific flags to achieve POSIX-compliant thread behavior.

Usage (Simplified Signature):

```
#define _GNU_SOURCE // Required for CLONE_NEWPID, CLONE_NEWNS, etc.
#include <sched.h>   // For clone and clone flags

pid_t clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
// fn:   Function pointer for the new process/thread to execute
// child_stack: Pointer to the stack for the new process/thread
// flags: Control what resources are shared or copied
// arg:   Argument passed to the fn function
```

Linux Low-Level Threading: ‘clone()‘ System Call II

Key ‘clone()‘ Flags and Their Implications:

- ‘CLONE_VM‘: (Clone Virtual Memory) Shares the calling process’s memory space. **Essential for threads.**
- ‘CLONE_FILES‘: Shares the calling process’s open file descriptors.
- ‘CLONE_SIGHAND‘: Shares the calling process’s signal handler table.
- ‘CLONE_FS‘: Shares the file system information (root, current directory).
- ‘CLONE_PARENT‘: Child gets same parent as caller (e.g., for ‘vfork‘ behavior).
- ‘CLONE_THREAD‘: Puts the child into the same thread group, giving it the same PID as the parent (but a different TID). Used by pthreads.
- ‘CLONE_NEWPID‘, ‘CLONE_NEWNET‘, ‘CLONE_NEWNS‘, etc.: Create new **namespaces** (PID, network, mount). These are crucial for implementing container technologies like Docker, where processes need strong isolation.

Linux Low-Level Threading: ‘clone()‘ System Call III

By carefully combining ‘clone()‘ flags, one can create processes that are isolated (like ‘fork‘) or highly integrated (like pthreads), or even create lightweight containers.

Linux Scheduling Policies: Managing CPU Allocation I

Linux provides various scheduling policies to control how processes and threads compete for CPU time. These policies determine fairness, responsiveness, and real-time guarantees.

Main Scheduling Classes/Policies:

1. **'SCHED_OTHER' (Traditional / CFS):**
   - This is the **default time-sharing policy** for regular processes.
   - Implemented by the **Completely Fair Scheduler (CFS)**.
   - **Goal:** Provide a fair share of CPU time to all runnable tasks. Prioritizes interactivity for desktop users.
   - Uses 'nice' values (or 'setpriority') for user-space priority adjustment.

Linux Scheduling Policies: Managing CPU Allocation II

- **‘SCHED_FIFO‘ (First-In, First-Out):** Non-preemptive among tasks of the same priority. A ‘SCHED_FIFO‘ task runs until it explicitly yields the CPU or blocks.
- **‘SCHED_RR‘ (Round-Robin):** Preemptive among tasks of the same priority. Each ‘SCHED_RR‘ task is given a time slice (quantum); if it’s still running when its quantum expires, it’s moved to the end of the ready queue for its priority level.
- Real-time priorities range from **1 (lowest) to 99 (highest)**. (0 is for ‘SCHED_OTHER‘ internally).

Setting Scheduling Policy and Priority:

- **System Call:** ‘sched_setscheduler(pid, policy, param)‘
- **Pthreads API:** ‘pthread_attr_setschedpolicy()‘, ‘pthread_attr_setschedparam()‘ (for thread attributes before creation).
- **Command Line (Linux):**
  - ‘chrt -f 50 ./my_rt_app‘ (sets FIFO policy with priority 50)

Linux Scheduling Policies: Managing CPU Allocation III

- ‘chrt -r 50 ./my_rt_app‘ (sets Round-Robin policy with priority 50)

Linux: Set Process Priority (‘nice‘) and CPU Affinity (‘taskset‘) I

Linux provides tools and APIs to influence where and when a process/thread runs.

1. Process Priority (Nice Level):

- Controls the CPU scheduling priority for ‘SCHED_OTHER‘ (CFS) tasks.
- A higher ‘nice‘ value means a **lower** priority (more ”nice” to other processes).
- Range: **-20 (highest priority) to 19 (lowest priority)**. Default is 0.
- Only root can set negative ‘nice‘ values (increase priority).

Command Line ‘nice‘ (for starting a new process):

```
nice -n -5 ./my_cpu_bound_program  # Start with higher priority (lower nice value)
nice -n 10 ./my_background_task    # Start with lower priority (higher nice value)
```

Command Line ‘renice‘ (for a running process):

```
renice +10 -p 1234                 # Change process 1234's nice value to 10
renice -5 -u myuser                # Change all processes owned by 'myuser' to nice -5
```

Linux: Set Process Priority (‘nice‘) and CPU Affinity (‘taskset‘) II

2. CPU Affinity:

- Binds a process or a specific thread to a subset of available CPU cores.
- **Benefits:** Improves cache locality, reduces context switching overhead, can dedicate cores for specific workloads.
- **Drawbacks:** Can reduce overall system flexibility if used improperly.

Command Line ‘taskset‘:

```
taskset -c 0,2 ./my_program        # Run 'my_program' only on cores 0 and 2
taskset -p 0,1 1234                # Bind running process 1234 to cores 0 and 1
```

Programmatic Control (C/C++ using ‘sched_setaffinity‘):

Linux: Set Process Priority (‘nice‘) and CPU Affinity (‘taskset‘) III

```c
#define _GNU_SOURCE // For CPU_SET macros
#include <sched.h>  // For sched_setaffinity, CPU_SET, etc.
#include <unistd.h> // For getpid

// ... inside a function ...
cpu_set_t set;      // CPU set structure
CPU_ZERO(&set);     // Clear all CPUs from the set
CPU_SET(0, &set);   // Add CPU 0 to the set
CPU_SET(2, &set);   // Add CPU 2 to the set

pid_t current_pid = getpid(); // Or a specific thread ID (TID) for threads
// Set affinity for the current process/thread
if (sched_setaffinity(current_pid, sizeof(cpu_set_t), &set) == -1) {
    perror("sched_setaffinity");
}
```

These tools allow administrators and developers to optimize performance for critical applications or specific server roles.

Windows: Process and Thread APIs I

Windows provides a different set of APIs for managing processes and threads, often with more parameters and structures for fine-grained control.

1. Process Creation: ‘CreateProcess()‘

- Unlike Unix’s ‘fork()‘ then ‘exec()‘, Windows uses a single ‘CreateProcess()‘ function.
- It’s a powerful function that directly creates a new process and loads a specified executable into it.
- Returns handles to the new process and its primary thread.
- **Key parameters allow control over:**
  - Application name and command line
  - Security attributes
  - Inheritance of handles
  - Creation flags (e.g., ‘CREATE_NEW_CONSOLE‘, ‘DETACHED_PROCESS‘)
  - Environment variables

Windows: Process and Thread APIs II

- Current directory
- ‘STARTUPINFO‘ structure (window appearance)
- ‘PROCESS_INFORMATION‘ structure (receives handles and IDs of new process/thread)

- Example: ‘CreateProcess(NULL, ”C:
  Windows
  notepad.exe”, ..., &si, &pi);‘

2. Thread Creation: ‘CreateThread()‘ and ‘_beginthreadex()‘

- **‘CreateThread()‘ (Win32 API):**
  - Direct OS API for creating a new thread within the calling process’s address space.
  - Takes arguments for security attributes, stack size, start address (thread function), parameter to thread function, creation flags, and thread ID.
  - ‘HANDLE hThread = CreateThread(NULL, 0, ThreadFunc, (LPVOID)arg, 0, &dwThreadId);‘

Windows: Process and Thread APIs III

- **‘_beginthreadex()‘ (C Runtime Library - CRT):**
  - **Recommended for C/C++ applications** as it correctly initializes the CRT for the new thread.
  - This ensures proper handling of C++ objects, thread-local storage, and resource cleanup.
  - It internally calls ‘CreateThread()‘ but adds necessary CRT setup/teardown.
  - ‘uintptr_t thread_handle = _beginthreadex(NULL, 0, &ThreadFunc, (void*)arg, 0, &thread_id);‘
- **Thread Termination:** ‘ExitThread()‘ (clean exit for the thread), ‘TerminateThread()‘ (forceful termination, use with caution).
- **Waiting for Threads:** ‘WaitForSingleObject()‘ or ‘WaitForMultipleObjects()‘ using the thread handle.

Windows Scheduling and Priority Control I

Windows uses a priority-based, preemptive scheduling algorithm with dynamic priority boosting to achieve responsiveness.

1. **Priority Levels:**
   - Windows has **32 priority levels** (0-31), where 0 is the lowest and 31 is the highest.
   - Levels 16-31 are for **real-time threads** (privileged access).
   - Levels 1-15 are for **dynamic priority threads**.
   - Level 0 is for the zero page thread (special system thread).

2. **Priority Classes and Thread Priorities:**
   - Every process belongs to a **priority class** (e.g., 'IDLE_PRIORITY_CLASS', 'NORMAL_PRIORITY_CLASS', 'HIGH_PRIORITY_CLASS', 'REALTIME_PRIORITY_CLASS'). This sets the base priority for threads within that process.
     - 'SetPriorityClass(hProcess, priorityClass);'

Windows Scheduling and Priority Control II

- Within a process's priority class, individual threads can have relative priorities (e.g., 'THREAD_PRIORITY_LOWEST', 'THREAD_PRIORITY_NORMAL', 'THREAD_PRIORITY_HIGHEST').
  - 'SetThreadPriority(hThread, priorityLevel);'
- The effective thread priority is 'Process Base Priority + Thread Relative Priority'.

3. Dynamic Priority Boosting:

- Windows dynamically adjusts the priority of threads within the 'dynamic' range (1-15).
- **Boost on Ready:** When a thread wakes up from a wait state (e.g., I/O completion, semaphore signal), its priority is temporarily boosted to improve responsiveness.
- **Boost on Foreground Window:** Threads belonging to the currently active foreground application typically receive a higher priority boost.

Windows Scheduling and Priority Control III

- This mechanism tries to keep the system responsive for interactive applications even under load.

4. CPU Affinity (Processor Affinity Mask):

- Similar to Linux, threads can be bound to specific CPU cores or a subset of cores.
- ‘SetThreadAffinityMask(hThread, dwProcessorMask);‘
- ‘SetProcessAffinityMask(hProcess, dwProcessAffinityMask);‘

Windows’s scheduler aims for good overall system responsiveness for typical desktop and server workloads, often through adaptive priority adjustments.

Key Takeaways I

- **Linux Process Model (‘fork-exec-wait‘):** ‘fork()‘ creates a copy, ‘exec()‘ replaces code, ‘wait()‘ manages child termination. Provides strong isolation but higher overhead.
- **Linux Threading (‘pthread‘):** User-friendly API built on ‘clone()‘ for creating and managing threads within a process, offering shared memory and lower overhead than processes.
- **Linux ‘clone()‘:** A powerful, low-level system call allowing fine-grained control over resource sharing, used internally by pthreads and for containerization.
- **Linux Scheduling:** ‘SCHED_OTHER‘ (CFS) for general-purpose fair sharing; ‘SCHED_FIFO‘ and ‘SCHED_RR‘ for real-time applications with strict priority levels.
- **Linux Control:** ‘nice‘/‘renice‘ adjust ‘SCHED_OTHER‘ priority; ‘taskset‘ and ‘sched_setaffinity‘ control CPU affinity.

Key Takeaways II

- **Windows Process Model (‘CreateProcess()‘):** A single, comprehensive API to create a new process and load an executable, offering extensive control over its environment.
- **Windows Threading (‘CreateThread()‘, ‘_beginthreadex()‘):** ‘CreateThread()‘ is the native API, ‘_beginthreadex()‘ is recommended for C/C++ to ensure CRT compatibility.
- **Windows Scheduling:** Priority-based, preemptive system with 32 levels, incorporating process priority classes, thread relative priorities, and dynamic priority boosting for responsiveness.
- **Windows Control:** ‘SetPriorityClass()‘, ‘SetThreadPriority()‘, and ‘SetThreadAffinityMask()‘ allow tuning of process/thread execution.

Key Takeaways III

Reflection Prompt: Design Philosophies

Compare and contrast the design philosophies behind Linux's 'fork()'+'exec()' vs. Windows's 'CreateProcess()'. What are the pros and cons of each approach from a programmer's perspective? How do these choices impact the system's overall flexibility and simplicity?

Next Week Preview: OS Review & System Design Principles I

Consolidating Knowledge and Looking at the Big Picture

- **Execution Review:** A comprehensive look back at processes, threads, and scheduling across the course.
- **Memory Management Review:** Virtual memory, paging, segmentation, caching.
- **File Systems Review:** Structure, access methods, and common operations.
- **I/O Systems Review:** Devices, drivers, and performance.
- **OS Structures and Lab Consolidation:** How components fit together, and a review of practical lab experiences.
- **Design Discussion:** What are the fundamental principles that make a modern OS efficient, reliable, and secure?
- **Final Quiz/Q&A Session.**

Next Week Preview: OS Review & System Design Principles II

Prep Tip for Next Session

Start reviewing all key concepts covered since the beginning of the course. Think about how different OS components (e.g., CPU scheduling, memory management, file systems) interact and rely on each other to create a functional system.

Outline

Exercise: Process and Thread API Application I

Part 1: Linux ‘fork()‘ and ‘exec()‘ Simulation
Consider a scenario where you want your C program to act as a simple shell that can execute basic commands.
Task:

1. Write a C program that prompts the user for a command (e.g., ”ls -l”, ”date”).
2. Use ‘fork()‘ to create a child process.
3. In the child process, use ‘execlp()‘ to execute the user’s command. Make sure to handle potential ‘exec()‘ errors.
4. In the parent process, use ‘wait()‘ to wait for the child to complete and print its exit status.
5. Implement a loop so the ”shell” continues to prompt for commands until the user types ”exit”.

Exercise: Process and Thread API Application II

Hint: You'll need to parse the input string to separate the command and its arguments for ‘execlp‘ or ‘execvp‘. ‘strtok‘ might be useful, but be careful with its usage.

Part 2: Thread Priority and Affinity Impact
Imagine you are developing a real-time audio processing application on a multi-core Linux system.
Task:

1. You have a critical audio processing thread and a less critical UI update thread. How would you prioritize them using Linux scheduling policies and ‘nice‘ values (or real-time priorities)? Explain your choice.
2. The audio processing thread is very sensitive to cache misses. How could you use CPU affinity to potentially improve its performance? Describe the API/tool you would use and the expected effect.
3. If you were developing this on Windows, how would you set the priorities for these two threads? (Refer to Windows APIs).

Exercise: Process and Thread API Application III

Reminder

For Part 1, focus on the correct ‘fork‘/‘exec‘/‘wait‘ flow and error handling. For Part 2, think about the specific characteristics of real-time vs. normal tasks and how OS features address them.

Appendix: Advanced Topics to Explore I

Going Deeper into Process & Thread Management

I. Inter-Process Communication (IPC)

- **Pipes (Anonymous & Named):** Detailed use cases for inter-process communication, especially parent-child or unrelated processes.
- **Message Queues:** How they allow processes to exchange structured messages asynchronously.
- **Shared Memory:** Fastest IPC mechanism, but requires explicit synchronization (e.g., using semaphores/mutexes).
- **Sockets:** Network-based IPC, enabling communication across different machines or processes on the same machine.

Appendix: Advanced Topics to Explore II

II. Advanced Scheduling & Real-Time Systems

- **Scheduler Implementation Details (Linux CFS):** How red-black trees and run-queues work to achieve fairness and O(log N) scheduling complexity.
- **Real-Time Operating Systems (RTOS):** Characteristics, common RTOS (e.g., FreeRTOS, VxWorks), and their unique scheduling requirements (e.g., strict deadlines, jitter minimization).
- **Preemption and Context Switching Overhead:** Quantitative analysis of the cost of context switches and its impact on performance.

Appendix: Advanced Topics to Explore III

III. Process & Thread Control Beyond Basics

- **Daemonization in Linux:** Best practices for creating background processes that detach from the controlling terminal.
- **Thread Pools:** Managing a pool of worker threads to efficiently handle tasks, reducing creation/destruction overhead.
- **Fibers/Coroutines:** User-mode scheduling primitives for cooperative multitasking, often used in game engines or high-performance networking.
- **Job Objects (Windows):** Grouping processes for resource management (e.g., setting CPU limits, memory limits for a group of processes).