Usman Ghafoorzai

NTNU

OBL2-OS

1 The process and threads

1.1 Explain the difference between a process and a thread

A process is an independent program execution, with its own allocated memory space (code, data, heap and stack segments), whilst a thread is the smallest unit of execution within a process. Multiple threads within a single process share the same memory space (heap, data, code) but have separate execution stacks. This means that processes are isolated from each other, that is, one process can not directly access the memory of another process, but threads within the same process, however, can easily communicate and share resources since they operate within the same memory space. Lastly, creating a process is more resource-intensive, as it requires allocating a separate memory space and initializing system resources, and since threads share same memory and resources of the parent process it is then less resource-intensive to create.

1.2 Describe a scenario where it is desirable to

a. Write a program that uses threads for parallel processing

Threads are desirable when tasks need to collaborate and share resources, like in a real-time online multiplayer game. For instance, they allow different game elements, like player movements, enemy actions etc. to be handled concurrently. Since threads share the same memory space, they can quickly access and update shared game data, making communication fast and efficient. Threads are lightweight, which makes them easier to create and manage, a crucial aspect for real-time responsiveness in gaming. For instance, one thread could render graphics, another manages user-input, and another control game physics simultaneously, leading to smooth, real-time updates.

b. Write a program that uses processes for parallel processing

Processes are desirable when isolation and stability are critical, like with hosting multiple web applications. When multiple independent websites are running, each website can run in a separate process. This ensures that if one crashes or faces a

security issue, it will not affect the others – improving security and stability. Processes are isolated, with their own memory spaces, preventing data corruption or interference between websites.

1.3 Explain why each thread requires a thread control block (TCB)

The essential information required by the OS to manage and execute the thread is contained in the data structure Thread Control Block (TCB). It contains the state of the computation performed by the thread, including a pointer to the thread's stack and a copy of processor registers, such as the instruction pointer and stack pointer. These elements are crucial for suspending and resuming a thread's execution, as they allow the system to save the thread's current state when it's not running and restore it when the thread is scheduled to run again.

Moreover, the TCB contains the thread's ID, scheduling priority, and resource consumption, which the OS uses to manage thread execution and coordination. This separation of per-thread state (held in the TCB) and shared state (such as program code and global variables) ensures that each thread has its own context for execution while sharing common resources within a multi-threaded process. Without the TCB, the operating system would not be able to track or manage the individual states of threads, making parallel execution and context switching impossible.

1.4 What is the difference between cooperative (voluntary) threading and pre-emptive (involuntary) threading? Briefly describe the steps necessary for a context switch for each case.

Cooperative threading is when a thread must voluntarily yield control of the processor. It follows then that a thread runs until it explicitly calls a function (e.g. `thread_yield()`) to give up control so that another thread can run. This results in simplicity since context switches only occur at well-defined points. However, if a thread forgets or refuses to yield, it can monopolize the processor and cause performance issues.

- Context switch steps (cooperative):
  i. The running thread calls a thread library function (e.g. `thread_yield()`).
  ii. The current thread's registers, including the program counter and stack pointer) are saved by the function to its TCB.
  iii. The thread is then moved to the ready list, and a new thread is chosen to run.
  iv. The new thread's registers are loaded from its TCB into the processor, and the new thread starts or resumes execution.

On the other hand, pre-emptive threading is when the OS can interrupt a running thread without its consent. Typically done using hardware interrupts, such as a timer interrupt, to ensure that no single thread can monopolize the processor. It is the kernel that decides when to switch between threads, resulting in a more efficient way of managing resources.

- Context switch steps (pre-emptive):
  i. A timer interrupt – or other hardware interrupt – occurs, causing the system to pause the current thread.
  ii. The interrupt handler saves the current thread's state (its registers, program counter and stack pointer) to its TCB.
  iii. The scheduler selects a different thread from the ready list.
  iv. The new thread's state is restored from its TCB, and control is then transferred to the new thread by loading its registers into the processor.

Usman Ghafoorzai

NTNU

2 C program with POSIX threads

```
virtualmachine@virtualmachine:~/os-oving$ gcc -o threadHello threadHello.c -lpthread
virtualmachine@virtualmachine:~/os-oving$ ./threadHello
Hello from thread 2
Hello from thread 3
Hello from thread 4
Hello from thread 5
Hello from thread 6
Hello from thread 1
Hello from thread 7
Hello from thread 8
Hello from thread 9
Hello from thread 0
Thread 0 returned with 100
Thread 1 returned with 101
Thread 2 returned with 102
Thread 3 returned with 103
Thread 4 returned with 104
Thread 5 returned with 105
Thread 6 returned with 106
Thread 7 returned with 107
Thread 8 returned with 108
Thread 9 returned with 109
Main thread done.
```

2.1 Which part of the code (e.g., the task) is executed when a thread runs? Identify the function and describe briefly what it does.

The part of the code that is executed when a thread runs, is the function `go()` – as shown below. This function is passed to each thread during its creation via the `pthread_create` call.

```c
void *go (void *n) {
        printf("Hello from thread %1d\n", (long)n);
        pthread_exit(100 + n);
}
```

It receives an argument `n`, which represents the thread's ID, cast to a `void*` pointer. The function prints a message, "Hello from thread X", where X is the thread's ID (the value of n cast back to a long type). Thereafter the function calls `pthread_exit(100+n)`, terminating the thread and returning the value 100 + n (where n is the thread's ID) as the thread's exist status.

2.2 Why does the order of the "Hello from thread X" messages change each time you run the program?

The order changes each time because the threads are executed concurrently, and the OS determines the scheduling of each thread. In a multi-threaded program, the the exact order in which threads are scheduled to run is non-deterministic and depends on several factors, such as:

Thread scheduling (OS's thread scheduler decides which thread is run at any given moment, decided by system load, CPU availability, and other processes running on the system), **race conditions** (all the threads are running independently and potentially in parallel, there is no guarantee on the order in which each thread will print its message) and **pre-emption** (the OS might pause one thread to run another thread based on its internal scheduling algorithm).

2.3 What is the minimum and maximum number of threads that could exist when thread 8 prints "Hello"?

For minimum number of threads: 1 (only thread 8 is running) whilst for maximum number of threads: 10 (all threads are created and running).

2.4 Explain the use of pthread_join function call.

The pthread_join function in POSIX threads is used to wait for a specific thread to complete its execution. When called, it blocks the calling thread (usually the main thread) until the target thread finishes, ensuring synchronization between threads. It takes two arguments: the thread identifier and a pointer to capture the thread's return value. This allows the calling thread to retrieve the exit status or return value of the finished thread, which can be useful if the thread's result is needed. Additionally, pthread_join helps prevent "zombie" threads by ensuring proper cleanup of resources after a thread's termination. Overall, it facilitates orderly execution and resource management in multi-threaded programs.

Usman Ghafoorzai

NTNU

2.5 What would happen if the function go is changed to behave like this:

void *go (void *n) { printf("Hello from thread %ld\n", (long)n); if(n == 5) sleep(2); // Pause thread 5 execution for 2 seconds pthread_exit(100 + n); // REACHED? }

If the function go() is changed to include a sleep(2) call for thread 5 – it would cause thread 5 to pause for 2 seconds, before completing its execution. In the meantime, other threads would continue to run and print their "Hello" messages, potentially finishing before thread 5 resumes. Hence, thread 5's output would be delayed, but the overall behaviour of the program remains the same. The pthread_join function will still ensure that the main thread waits for all threads, including the delayed thread 5, before finishing.

2.6 When pthread_join returns for thread X, in what state is thread X?

Thread X is then in the terminated or finished state, that is, thread X has completed its execution and has exited, either by reaching the end of its function or calling pthread_exit. At this point, the resources associated with thread X have been cleaned up, and the main thread (or the thread that called pthread_join) can safely retrieve the exit status or any other data from thread X.