02-innlevering.md

OBL2-OS - Kernel, Threads, and Processes

1. Processes and threads

1.1 Difference between process and thread

A process is a program in execution which is isolated with its own memory space. A thread is a smaller unit of execution within a process that shares the same memory space as other threads in the same process. thread is like a child of a process, and can be called inside a process. It is important to note that threads within the same process can share data and resources, while processes are isolated from each other. You can compare this to a house where living alone is a process, and living with family is a thread, sharing things like food and utilities while having your own room. In `c` you can start a thread using the pthread library, while a process can be started using the `fork()` system call.

1.2 Scenarios for parallel processing

Write a program that uses threads for parallel processing (thread)

```c
#include <pthread.h>
#include <stdio.h>

void* thread_func(void* arg) {
    printf("Hello from thread!\n");
    return NULL;
}

int main() {
    pthread_t t;
    // Arguments: thread, attributes, function, arg; return 0 on success
    if(pthread_create(&t, NULL, thread_func, NULL) != 0) {
        perror("Failed to create thread");
        return 1;
    }
    pthread_join(t, NULL); // Wait for thread to finish
    return 0;
}
```

Here is a simple C program with threading, including error checking for thread creation.

Write a program that uses processes for parallel processing (fork)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        // Child process
        printf("Hello from child process!\n");
        exit(0);
    } else if (pid > 0) {
        // Parent process
        printf("Hello from parent process!\n");
        int status;
        wait(&status); // Wait for child to finish
    } else {
        // Fork failed
        perror("fork");
        exit(1);
    }
    return 0;
}
```

Here is a simple C program using processes with fork, including error checking for fork failure.

1.3 Thread Control Block (TCB)

A TCB is a datastructure of a thread, or in other words, it keeps track of all the relevant information about a thread, kind of like a struct with set values; If it was a struct it would look something like this:

```
struct TCB {
    int thread_id;            // ID
    int state;                // Running, Ready, Blocked
    int priority;             // Scheduler priority
    void* stack_pointer;      // SP
    void* program_counter;    // PC
    void* stack_base;         // Start of stack
    size_t stack_size;        // Size of stack
    void* tls_ptr;            // Thread-local storage
    struct process* parent;   // Pointer to PCB
}
```

This is needed to manage and schedule threads effectively, allowing the OS to switch between threads, save their state, and resume them later. It is essential for multitasking and ensuring that each thread gets its fair share of CPU time.

1.4 Cooperative vs Pre-emptive threading

Difference between cooperative and pre-emptive threading

Cooperative threading is ***choosing self*** when to yield control to other threads, which on newer OS is not used that often, but needed in some embedded systems where resources are limited (or older systems). In C in this case you would use the pthread library, and use yield functions like `pthread_yield()` to voluntarily give up control to other threads.

Pre-emptive threading is when the OS decides when to switch between threads, which is the most common way of threading today. In C in this case you would use the pthread library, and not use any yield functions. and the OS would use timer interrupts to switch between threads automatically.

Context switch steps for cooperative threading

1. Thread A is running and decides to yield control.
2. Thread A saves its context (TCB).
3. Thread A updates its state to "Ready" and adds itself to the ready queue.
4. The scheduler selects another thread (for example, Thread B) to run.
5. Thread B restores its context and starts running.

Context switch steps for pre-emptive threading

1. The OS timer interrupt occurs.
2. The currently running thread (for example, Thread A) is pre-empted.
3. The OS saves the context of Thread A (TCB struct).
4. The OS updates the state of Thread A to "Ready" and adds it to the ready queue.
5. The scheduler selects another thread (for example, Thread B) to run.
6. Thread B restores its context and starts running.

2. C program with POSIX threads

2.1 Thread execution

The part that is executed is the `*go` function, which is called inside the main function using `pthread_create`. Each thread will print "Hello from thread X" where X is the thread number, and then exit. When done it will print Main thread done!.

2.2 Order of "Hello from thread X" messages

This is a pre-emptive threading model, so the order of the messages seems random. The threads may print their messages in any order, depending on how the OS schedules them. As seen in the print example in the task description.

2.3 Number of threads when thread 8 prints "Hello"

Minimum number of threads

As it has come to index 8 (including 0), the minimum number is the current thread index + main thread, so **9 threads**.

Maximum number of threads

As the maximum amount of threads is defined in `#NTHREADS 10`, the maximum number of threads is 10 threads + 1 main thread = **11 threads**.

2.4 pthread_join function call

pthread_join is used to wait for a specific thread to finish before continuing execution in the main thread. It works like a return statement for a thread, ensuring that the main thread waits for the specified thread to complete before proceeding. This is important for synchronizing threads and ensuring that resources are properly managed.

2.5 Modified go function with sleep

Adding a `sleep(2)` call inside the function to thread, will make it alot less random. As the threads will be put to sleep for 2 seconds, and the OS will have time to switch between them. This will make the output more predictable, as the threads will print their messages in a more orderly fashion with the 2 second delay.

2.6 Thread state after pthread_join returns

In the treads TCB "struct", the state would now be updated to "TERMINATED", this means the tread has completed, and all resources accociated with the thread have been cleaned up by the OS. This will make the tread ID invalid.