Threads

Key Concepts:

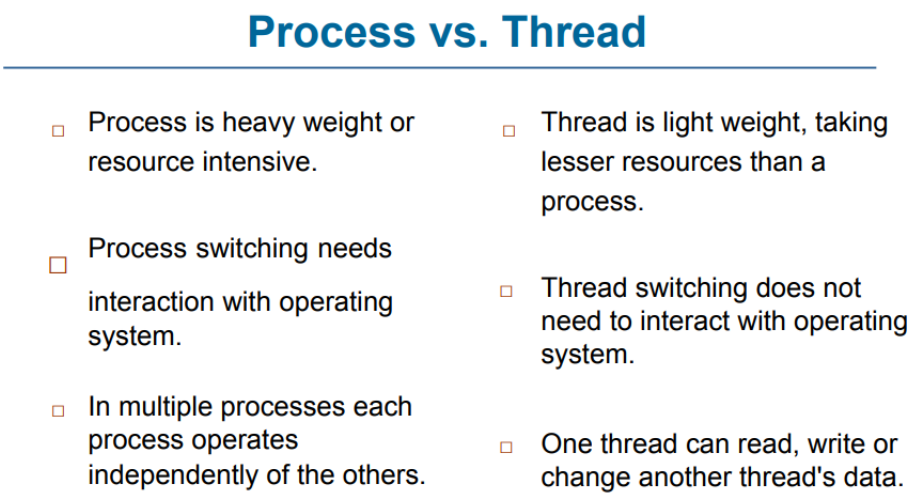
- **Multiprogramming**: Multiple processes running on one CPU, where the CPU switches between processes when they relinquish it.
- **Multiprocessing**: Refers to systems with more than one CPU, allowing for true parallel execution.
- **Multitasking**: Preemptive switching between processes/threads, where the OS assigns each a time quantum.
- **Time Sharing**: A system where CPU time is shared among processes, typical in multitasking and multiprogramming systems.
- **Multithreading**: A model where multiple threads run concurrently within the same process, each representing a code segment.

Motivation for Multithreading

- **Modern Applications**: Most modern software, like MS Word, is multithreaded. For example, one thread handles spell-checking while another processes keystrokes.
- **Lightweight**: Thread creation is lightweight compared to process creation, making threads more efficient.
- **Kernels**: Operating system kernels are typically multithreaded for better efficiency.

Process and Thread

- **Process**:
  - Has its own address space, open files, child processes, accounting information, and signal handlers.
  - Executes a single thread of control.
- **Thread**:
  - A thread is the smallest unit of execution within a process.
  - Shares the process's data, code, and files.
  - Has its own program counter, register, and stack.
- **Processes vs. Threads**:
  - A process groups resources while threads are scheduled for execution on the CPU.
  - Threads are often referred to as lightweight processes (LWP).



Benefits of Threads

- **Responsiveness**: Threads can allow parts of an application to continue execution even if one part is blocked (e.g., a user interface).
- **Resource Sharing**: Threads within the same process share resources, making it easier than using shared memory between processes.
- **Economy**: Threads are cheaper to create and manage compared to processes.
- **Scalability**: Applications with multiple threads can take advantage of multi-core processors.

Multicore Programming

- **Parallelism**: Threads can be run in parallel on multi-core systems, providing real speed-up.
- **Concurrency**: Threads allow tasks to make progress concurrently even on a single-core system.

Types of Parallelism:

- **Data Parallelism**: Distributing subsets of the same data across multiple cores.
- **Task Parallelism**: Each thread performs a unique task, running in parallel.

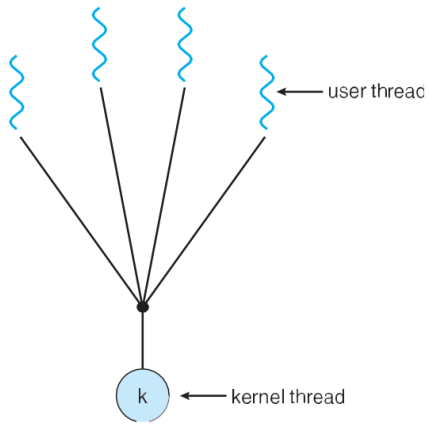
User Threads vs. Kernel Threads

- **User Threads**:
  - Managed by a thread library at the user level.
  - Faster to create and manage.
  - The OS is unaware of user-level threads.
- **Kernel Threads**:
  - Managed directly by the operating system.
  - Slower to create but can take full advantage of multiprocessing.

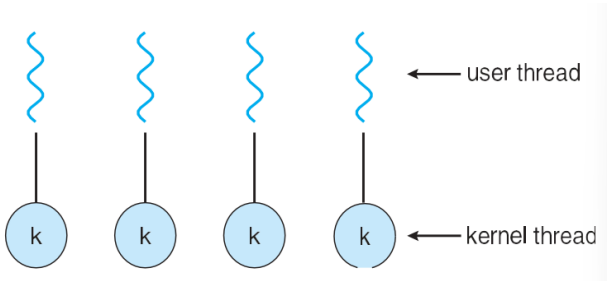
- If a kernel thread is blocked, others can continue execution.

Multithreading Models

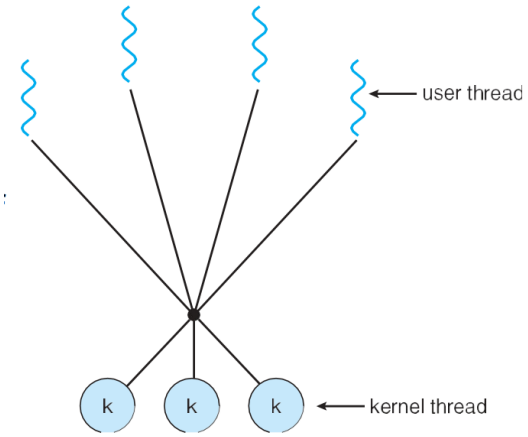
- **Many-to-One**: Many user threads map to one kernel thread. Limits concurrency; if one thread blocks, all others do too. Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time.



- **One-to-One**: Each user thread maps to a kernel thread. Offers greater concurrency but with higher overhead. Even if one thread is blocked, others will still run. Number of threads per process sometimes restricted due to overhead.



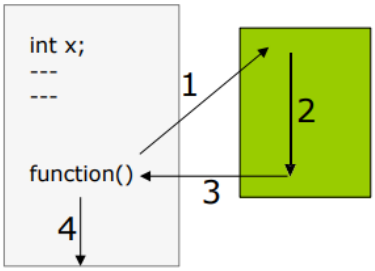
- **Many-to-Many**: Many user threads map to many kernel threads, balancing concurrency and overhead effectively. The number of kernel level threads maybe equal to or less than user level threads.

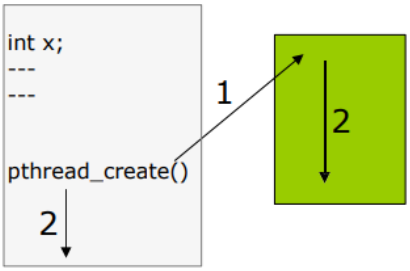


Threads in Action

C Function Call vs. Thread Creation

In C programming, a function call and thread creation can be used to execute code. However, threads allow for concurrent execution, while function calls are synchronous.





Thread Creation Using Pthread:

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

void* myThread(void* arg) {
    printf("This is a thread\\n");
    return NULL;
}

int main() {
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, myThread, NULL); // Creating a new threa
d
    pthread_join(thread_id, NULL);  // Waiting for the thread to finish
```

```
    return 0;
}
```

In the thread example, the `pthread_create` function creates a new thread to execute the function `myThread`. The `pthread_join` function waits for the thread to finish before continuing execution in the `main` thread.

Pthread Operations

POSIX provides a set of functions to manage threads. Here are some common ones:

1. **pthread_create**: Create a thread.
2. **pthread_detach**: Set thread to release resources after termination.
3. **pthread_equal**: Compare two thread IDs.
4. **pthread_exit**: Terminate a thread.
5. **pthread_kill**: Send a signal to a thread.
6. **pthread_join**: Wait for a thread to finish.
7. **pthread_self**: Get the calling thread's ID.

Practical Work Example: pthread_create

The `pthread_create` function is used to create a new thread. Here's how it works:

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

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

int main() {
    pthread_t thread_id;
    int thread_status;

    // Creating a thread
    thread_status = pthread_create(&thread_id, NULL, threadFunction, NULL);

    // Check if thread creation was successful
    if (thread_status != 0) {
        printf("Error: Unable to create thread\\n");
    }

    // Wait for the thread to finish
```

```
    pthread_join(thread_id, NULL);

    return 0;
}
```

Parameters:

- **First Parameter**: `pthread_t*` thread ID pointer, unique for each thread.
- **Second Parameter**: `pthread_attr_t` attribute for thread settings (use `NULL` for default attributes).
- **Third Parameter**: The function pointer that the thread will execute.
- **Fourth Parameter**: Argument to the function (only one argument allowed).

Returning Values in pthread_create

- **0**: If `pthread_create` is successful.
- **1**: If unsuccessful, and `errno` is set to an appropriate value.

Practical Work Example: pthread_join

The `pthread_join` function is used to wait for a thread to finish its execution.

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

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

int main() {
    pthread_t thread_id;

    // Creating the thread
    pthread_create(&thread_id, NULL, threadFunction, NULL);

    // Wait for the thread to finish
    pthread_join(thread_id, NULL);

    return 0;
}
```

- **First Parameter**: Thread ID of the thread to wait for.
- **Second Parameter**: Pointer to capture the return value of the thread.

Return Value of pthread_join

- **0**: Successful execution.
- **Error Number**: If there is an issue waiting for the thread.

pthread_exit Function

The `pthread_exit` function terminates the calling thread and provides a return value.

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

void* threadFunction(void* arg) {
    printf("Thread is exiting...\\n");
    pthread_exit(NULL);  // Thread terminates here
}

int main() {
    pthread_t thread_id;

    pthread_create(&thread_id, NULL, threadFunction, NULL);
    pthread_join(thread_id, NULL);  // Wait for thread to exit

    return 0;
}
```

Multithreading Example with Arrays and Command-Line Arguments

Multithreading can also be used to work with arrays or pass command-line arguments.

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

void* processArray(void* arg) {
    int* num = (int*)arg;
    printf("Thread received: %d\\n", *num);
    return NULL;
}

int main(int argc, char* argv[]) {
    pthread_t threads[3];
    int arr[] = {10, 20, 30};

    for (int i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, processArray, &arr[i]);
    }
```

```
    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);  // Wait for each thread
    }

    return 0;
}
```

Command-Line Argument Example

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

void* printMessage(void* arg) {
    char* message = (char*)arg;
    printf("Thread received: %s\\n", message);
    return NULL;
}

int main(int argc, char* argv[]) {
    pthread_t thread;

    pthread_create(&thread, NULL, printMessage, argv[1]);
    pthread_join(thread, NULL);  // Wait for the thread to finish

    return 0;
}
```

Output Example:
If you run this code with the argument `"Hello"`, the thread will print:

```
Thread received: Hello
```

Questions

1. **What happens when a thread calls an `exit()` function?**
    - The thread terminates immediately, releasing resources.
2. **Can a function that is part of a program be called from a thread?**
    - Yes, any function can be called by a thread.
3. **What happens when you call a `fork()` in a thread?**
    - The `fork()` system call creates a new process, and the child process will inherit a single thread from the parent.

4. **What happens when you call `pthread_exit()` in the main thread?**
    - The main thread terminates, but other threads in the process may continue to execute.

Practice Questions:

Practice Problem 1: Summing Numbers in a Thread

Write a program that spawns a thread to sum numbers from 1 to a given number **N**. The main thread waits for the spawned thread to complete and then prints the result.

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

int sum = 0; // Global sum variable

// Thread function to sum numbers from 1 to N
void* runner(void* param) {
    int n = *((int*) param);
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    pthread_exit(0);
}

int main() {
    pthread_t tid; // Thread identifier
    int N;

    printf("Enter a number N: ");
    scanf("%d", &N);

    pthread_create(&tid, NULL, runner, &N); // Create a new thread
    pthread_join(tid, NULL); // Wait for the thread to complete

    printf("The sum of numbers from 1 to %d is: %d\\n", N, sum);
    return 0;
```

Other method:

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

void* runner(void* param) {
    int n = *((int*) param);
    int* sum = (int*) malloc(sizeof(int)); // Allocate memory for sum
```

```
    *sum = 0;

    for (int i = 1; i <= n; i++) {
        *sum += i;
    }
    pthread_exit(sum); // Return sum using pthread_exit
}

int main() {
    pthread_t tid;
    int N;
    int* result; // Pointer to hold the return value

    printf("Enter a number N: ");
    scanf("%d", &N);

    pthread_create(&tid, NULL, runner, &N); // Create a new thread
    pthread_join(tid, (void**)&result); // Wait for the thread to complete an
d capture the result

    printf("The sum of numbers from 1 to %d is: %d\n", N, *result);

    free(result); // Free the allocated memory
    return 0;
}
```

Practice Problem 2: Multi-threaded Arithmetic

Create two threads to solve the expression **(3 * 10) + (5 * 4)**. The first thread calculates **(3 * 10)** and the second thread calculates **(5 * 4)**. The main thread combines the results and prints the final answer.

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

int result1, result2;

// Thread function for the first half: (3 * 10)
void* thread1_func(void* arg) {
    result1 = 3 * 10;
    pthread_exit(0);
}

// Thread function for the second half: (5 * 4)
void* thread2_func(void* arg) {
    result2 = 5 * 4;
```

```
    pthread_exit(0);
}

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, thread1_func, NULL); // Create first thread
    pthread_create(&tid2, NULL, thread2_func, NULL); // Create second thread

    pthread_join(tid1, NULL); // Wait for first thread
    pthread_join(tid2, NULL); // Wait for second thread

    int result = result1 + result2;
    printf("The result of (3 * 10) + (5 * 4) is: %d\\n", result);
    return 0;
}
```

Practice Problem 3: Multi-threaded Array Search

Write a program to search for a value in an integer array. Split the array into two halves and use two threads to search each half independently.

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

int target;
int found = 0; // Flag to indicate if target is found

// Thread function to search the first half of the array
void* searchFirstHalf(void* arg) {
    int* arr = (int*) arg;
    for (int i = 0; i < 5; i++) {
        if (arr[i] == target) {
            found = 1;
            pthread_exit(0);
        }
    }
    pthread_exit(0);
}

// Thread function to search the second half of the array
void* searchSecondHalf(void* arg) {
    int* arr = (int*) arg;
    for (int i = 5; i < 10; i++) {
        if (arr[i] == target) {
```

```
            found = 1;
            pthread_exit(0);
        }
    }
    pthread_exit(0);
}

int main() {
    int arr[10] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};
    pthread_t tid1, tid2;

    printf("Enter a target value to search: ");
    scanf("%d", &target);

    pthread_create(&tid1, NULL, searchFirstHalf, arr); // Search first half
    pthread_create(&tid2, NULL, searchSecondHalf, arr); // Search second half

    pthread_join(tid1, NULL); // Wait for first thread
    pthread_join(tid2, NULL); // Wait for second thread

    if (found)
        printf("Target %d found in the array.\\n", target);
    else
        printf("Target %d not found in the array.\\n", target);

    return 0;
}
```

Practice Problem 4: Printing Numbers with Delay Using Threads

Write a program that prints the first 15 numbers, each after a delay of 1 second, using threads.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h> // For sleep function

void* printNumber(void* param) {
    int num = *((int*) param);
    sleep(1); // 1-second delay
    printf("%d\\n", num);
    pthread_exit(0);
}

int main() {
    pthread_t tid[15];
```

```
    int numbers[15];

    for (int i = 0; i < 15; i++) {
        numbers[i] = i + 1;
        pthread_create(&tid[i], NULL, printNumber, &numbers[i]); // Create th
reads
    }

    for (int i = 0; i < 15; i++) {
        pthread_join(tid[i], NULL); // Wait for all threads
    }

    return 0;
}
```

Why Do We Fork When We Can Thread?

Forking creates a new process that runs independently of the parent, with its own memory space. Threading, on the other hand, creates a new thread within the same process, sharing memory and resources. Forking is used when isolation between processes is needed, while threading is preferred when tasks need to share data and resources, and run concurrently without the overhead of creating new processes.