THREADS AND CONCURRENCY

Multithreading in Modern Applications:

- Most modern applications are designed to use multiple threads to handle tasks concurrently, leading to better performance and responsiveness.

Threads Within an Application:

- Threads operate within the scope of the application and share the same memory space.
- They enable the implementation of multiple tasks in parallel, such as:
    - Updating the display
    - Fetching data
    - Spell checking
    - Answering network requests

Lightweight Thread Creation:

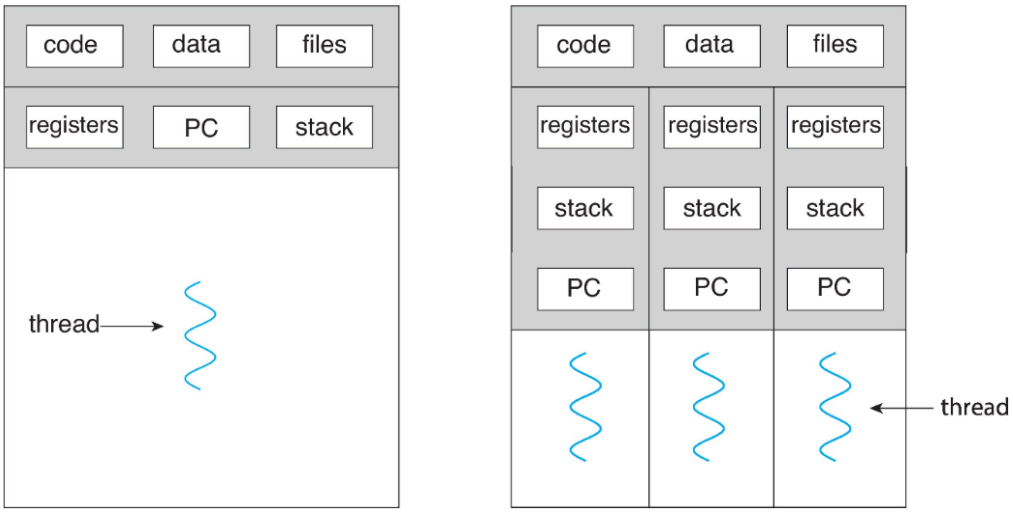
- Compared to process creation, which involves more overhead (separate memory space, process table entries, etc.), thread creation is much lighter and faster.

Code Simplification & Efficiency:

- By dividing tasks into threads, developers can simplify program logic and significantly improve system efficiency.

Kernel Multithreading:

- Modern operating systems themselves use multithreading. The kernel is generally multithreaded to manage system processes and resources efficiently.



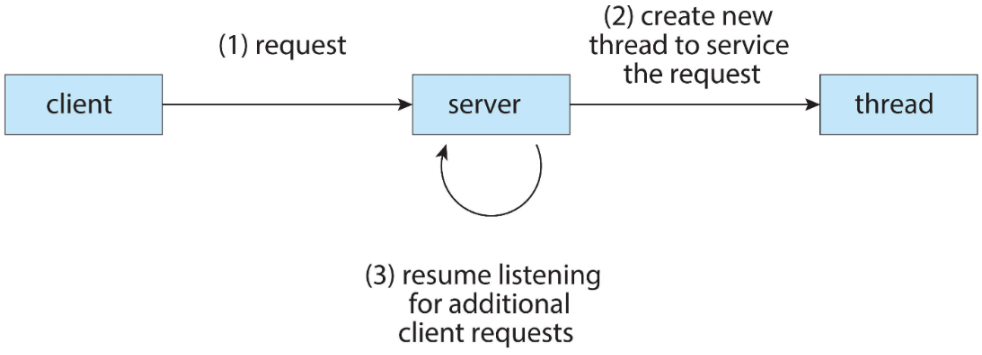
single-threaded process

multithreaded process

Aspect	Multiple Processes	Multiple Threads
Definition	Independent instances of a program	Lightweight units of execution within a process
Memory Usage	High (separate memory space for each process)	Low (shared memory space within a process)
Creation Overhead	High (OS resources required)	Low (less resource-intensive)
Isolation	Strong (processes are isolated)	Weak (threads share memory, risk of conflicts)
Communication	Complex (IPC mechanisms like pipes, and sockets)	Simple (shared memory, but requires synchronization)
Fault Tolerance	High (crash in one process doesn't affect others)	Low (a crash in one thread can crash the entire process)
Scalability	Limited by system resources (CPU, memory)	Limited by shared resources (e.g., GIL in Python)
Parallelism	True parallelism (on multi-core systems)	Limited parallelism (depends on language/runtime)
Synchronization	Not needed (independent memory)	Required (shared memory, risk of race conditions)
Use Case	CPU-bound tasks, independent workloads	I/O-bound tasks, tasks requiring shared data
Examples	Running separate programs, distributed systems	Web servers, GUI applications, async tasks

Multithreaded Server Architecture

1. **Client Sends a Request:** A client initiates communication by sending a request to the server.
2. **Server Creates a New Thread:** Upon receiving the request, the server spawns a new thread specifically to handle that request. This allows the main server to remain free to handle additional incoming requests.
3. **Server Continues Listening:** While the newly created thread processes the client's request, the server resumes listening for more incoming client requests, ensuring it remains responsive.



Benefits

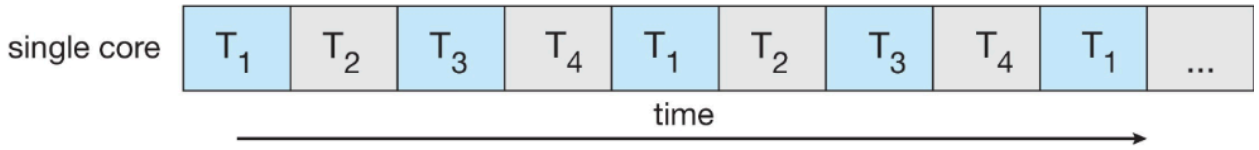
1. **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces.
2. **Resource Sharing –** threads share resources of process, easier than shared memory or message passing.
3. **Economy –** cheaper than process creation, thread switching lower overhead than context switching.
4. **Scalability –** process can take advantage of multicore architectures.

MULTICORE PROGRAMMING

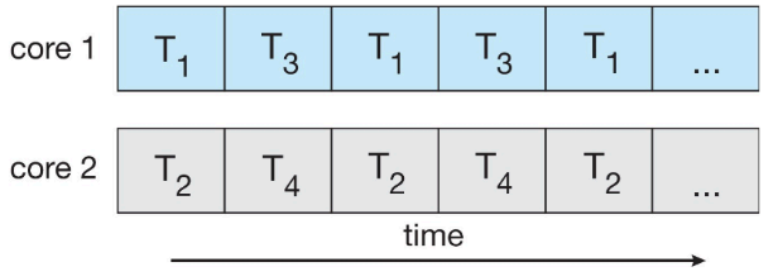
CONCURRENCY VS PARALLELISM:

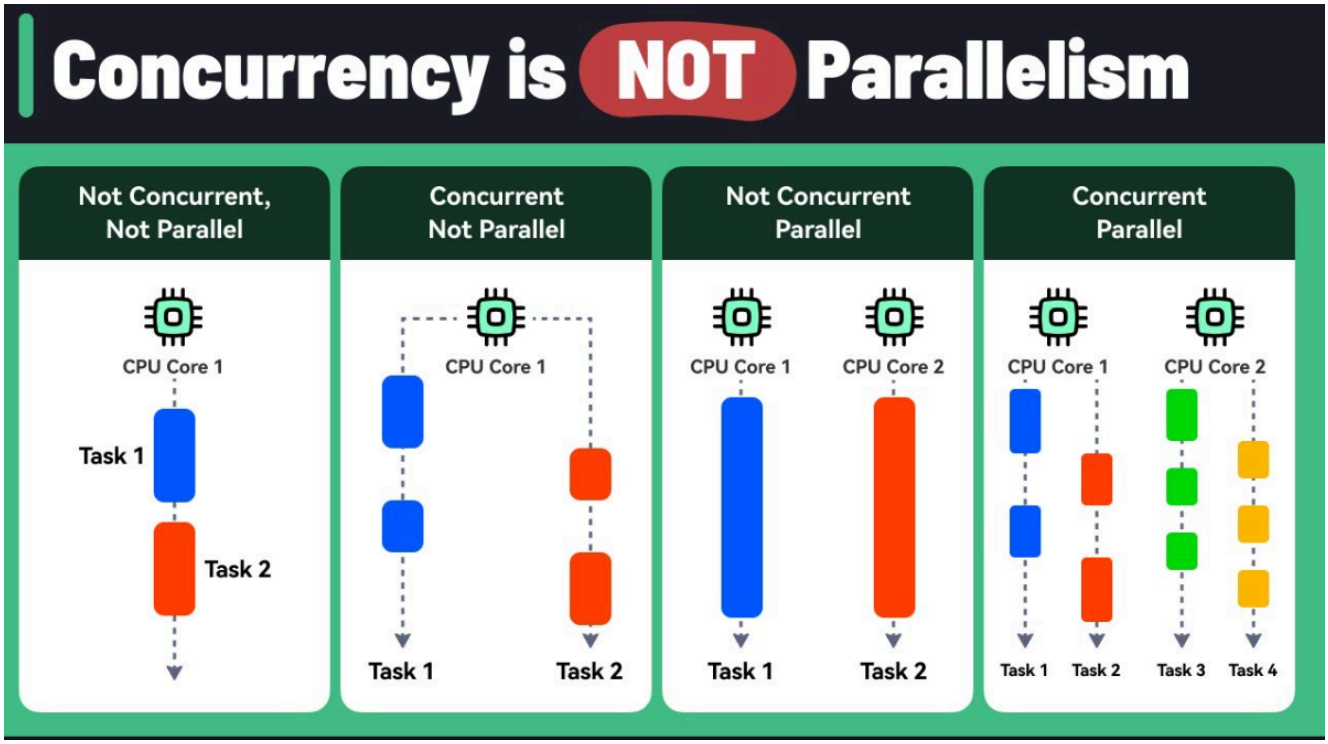
- Parallelism implies a system can perform more than one task simultaneously.
- Concurrency supports more than one task making progress.
  Single processor / core, scheduler providing concurrency.

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**





Not Concurrent, Not Parallel:

- There is a single CPU core (CPU Core 1) that processes Task 1 (blue) fully before moving on to Task 2 (red). This is a purely sequential execution.

Concurrent, Not Parallel:

- A single CPU core (CPU Core 1) handles both Task 1 (blue) and Task 2 (red) in an interleaved fashion. The tasks are not completed simultaneously but are progressed alternately, giving the impression of multitasking.

Not Concurrent, Parallel:

- Two CPU cores (CPU Core 1 and CPU Core 2) are involved, with each core working on one task independently. Task 1 (blue) is processed on one core while Task 2 (red) runs on the other. However, each task is completed in isolation, without interleaving.

Concurrent, Parallel:

- Two CPU cores (CPU Core 1 and CPU Core 2) are utilized, and they execute four tasks (Task 1 in blue, Task 2 in red, Task 3 in green, and Task 4 in yellow) both simultaneously and in an interleaved manner. This achieves both concurrency and parallelism.

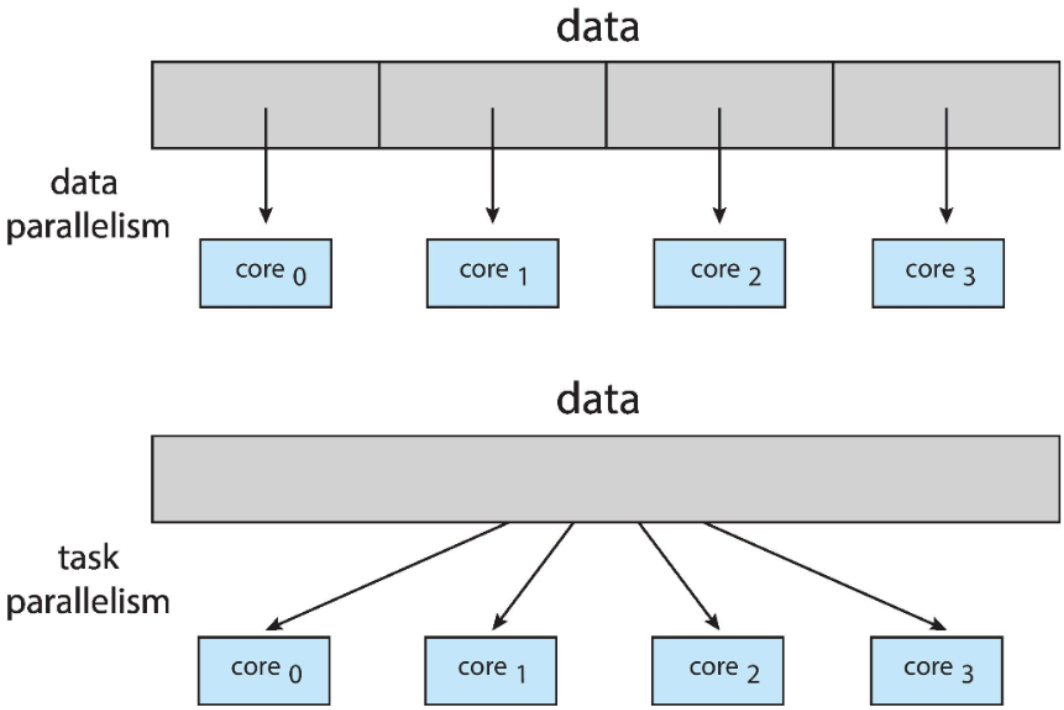
DLP VS TLP

1. Data-Level Parallelism (DLP):

- **Definition**: Involves performing the *same operation* on multiple pieces of data simultaneously.
- **Focus**: The emphasis is on splitting the *data* while keeping the operations uniform.
- **Example**:
    - Matrix operations, like multiplying a matrix with a scalar value. Each element of the matrix is processed in parallel.
    - Graphics processing in GPUs (e.g., rendering each pixel).
- **Use Cases**:
    - Scientific computations, simulations, and machine learning tasks where the same operation is applied across large datasets.
- **Benefits**:
    - High throughput for repetitive tasks.
    - Typically implemented in SIMD (Single Instruction Multiple Data) architectures, like GPUs.

2. Task-Level Parallelism (TLP):

- **Definition**: Involves performing different operations or tasks in parallel, often on the same or different data.
- **Focus**: The emphasis is on distributing *tasks* (functions or threads).
- **Example**:
    - Running a web server where one thread handles HTTP requests, another manages database queries, and another processes logging.
- **Use Cases**:
    - Multithreaded applications, real-time systems, or when various independent tasks need execution.
- **Benefits**:
    - Maximizes resource utilization by running diverse operations.
    - Typically implemented in MIMD (Multiple Instruction Multiple Data) architectures, like CPUs.



Q) Suppose there are 300 exam papers and there are a total of 3 TAs and they have to check all exam papers, each having 3 Q's. Demonstrate DLP and TLP.

Data-Level Parallelism (DLP):

- **How it works in this scenario**:
  - The 3 TAs can divide the 300 exam papers equally, with each TA checking **100 papers**.
  - Within each paper, each TA performs the *same operation* on all the questions—evaluating every answer using the same criteria or rubric.
  - Example:
    - TA 1 checks Papers 1 to 100.
    - TA 2 checks Papers 101 to 200.
    - TA 3 checks Papers 201 to 300.
  - Here, the operation (checking) is uniform across all papers, making it a classic case of DLP.

Task-Level Parallelism (TLP):

- **How it works in this scenario**:
  - Each TA can handle **different tasks** related to the evaluation process for every paper. For instance:
    1. **Task 1** (TA 1): Check Question 1 across all 300 papers.

2. **Task 2** (TA 2): Check Question 2 across all 300 papers.
3. **Task 3** (TA 3): Check Question 3 across all 300 papers.

- Here, each TA specializes in a specific task (e.g., checking a specific question) rather than handling all questions for a set of papers. This separation of tasks represents TLP.

AMDAHL'S LAW:

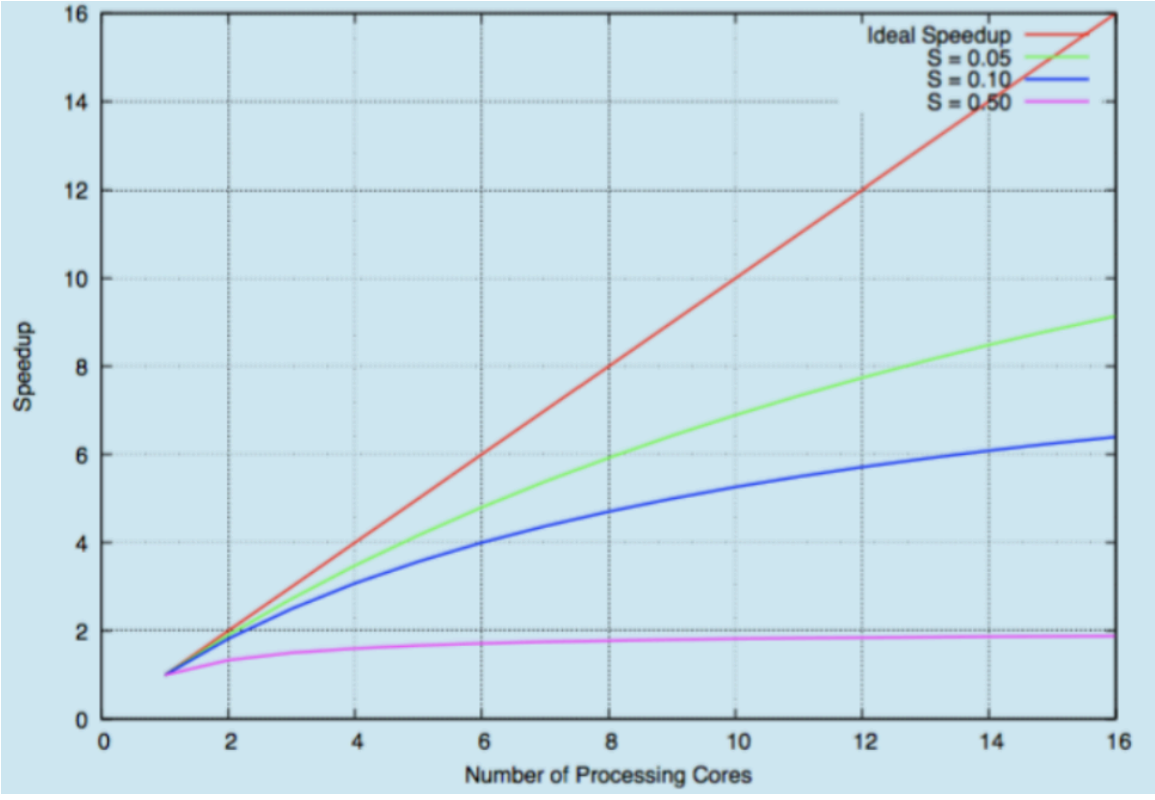
S is serial portion. N processing cores.

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Serial portion of an application has disproportionate effect on performance gained by adding additional cores.

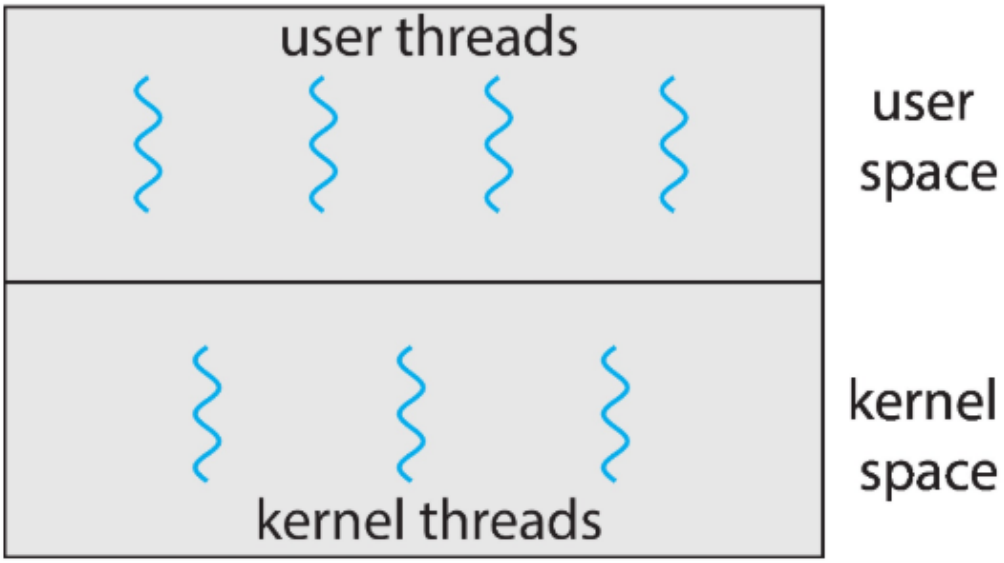
1. **Infinite Processors**:
   - When $N \to \infty$, the speedup becomes: Speedup $= 1/S$

- This highlights that the serial portion determines the upper bound of speedup.



Speedup decreases as the serial portion increases, due to the bottleneck caused by the non-parallelizable part of the task.

USER LEVEL AND KERNEL LEVEL THREADS:



Aspect	User Threads	Kernel Threads
Management	Managed by user-level libraries (e.g., Pthreads)	Managed directly by the operating system
Creation Overhead	Low (no kernel involvement)	High (system calls and kernel resources)
Scheduling	Scheduled by user-level thread scheduler	Scheduled by OS scheduler
Blocking Behavior	One blocked thread blocks all threads in the process	One blocked thread doesn't affect others
Parallelism	Limited (runs as a single process in the kernel)	True parallelism (can utilize multiple cores)
Portability	High (OS-independent)	Low (OS-dependent)
Complexity	Easier to implement and manage	More complex (requires kernel support)
Performance	Faster context switching (no kernel mode)	Slower context switching (kernel mode)
Fault Tolerance	Low (crash in one thread affects all threads)	High (kernel isolates threads)
Use Case	Lightweight tasks, high-concurrency apps	CPU-bound tasks, real-time systems

MULTITHREADING MODULES:

MANY TO ONE :

Mapping:

- All user-level threads are mapped to one kernel thread, meaning the operating system is only aware of the single kernel thread.

Blocking Behavior:

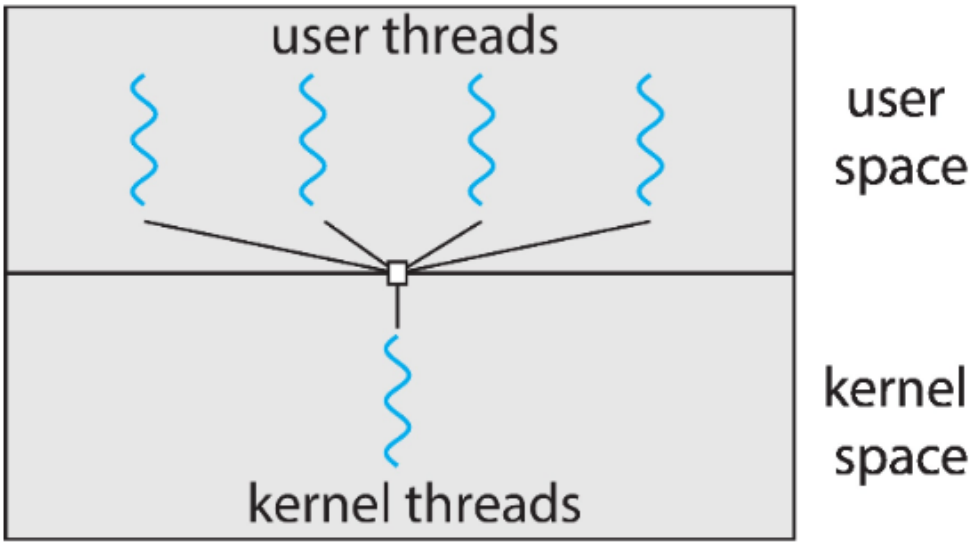
- If one user-level thread performs a blocking operation (e.g., I/O), the entire process is blocked since the kernel cannot distinguish between the user threads.

Concurrency:

- While multiple user-level threads can exist, they cannot run in parallel on multicore systems because the single kernel thread can only execute on one CPU core at a time.

Scheduling:

- User-level threads are scheduled by a user-level thread library, independent of the kernel. However, this scheduling is limited by the single kernel thread.



ONE TO ONE:

Mapping:

- Each user-level thread corresponds to a kernel thread, meaning the operating system is fully aware of all threads.

Concurrency:

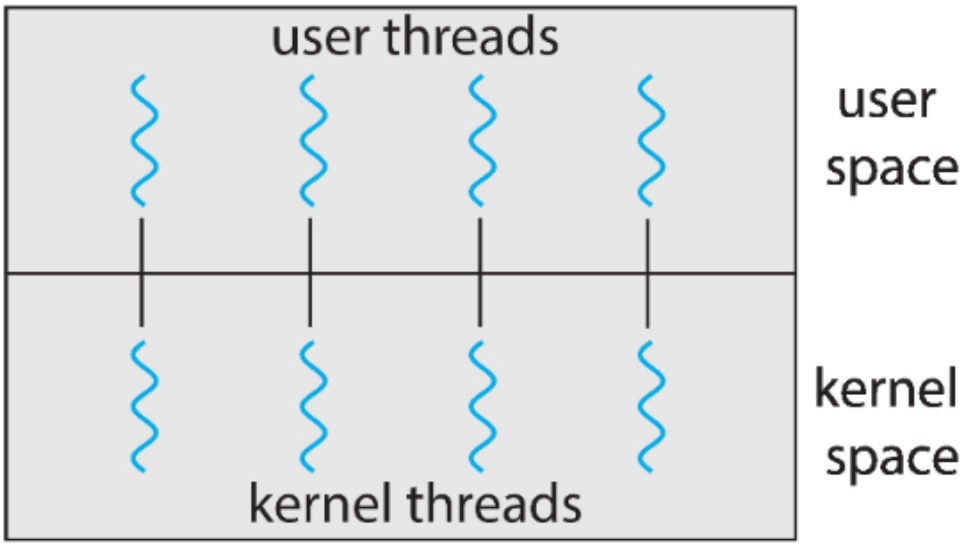
- This model provides **more concurrency** compared to the Many-to-One model because multiple threads can run in parallel on multicore systems.

Thread Creation:

- When a user-level thread is created, a kernel thread is also created. While this provides a direct mapping, it introduces overhead.

Scalability:

- The number of threads per process may be limited by system resources and overhead caused by managing a large number of kernel threads.



MANY TO MANY MODEL:

Mapping:

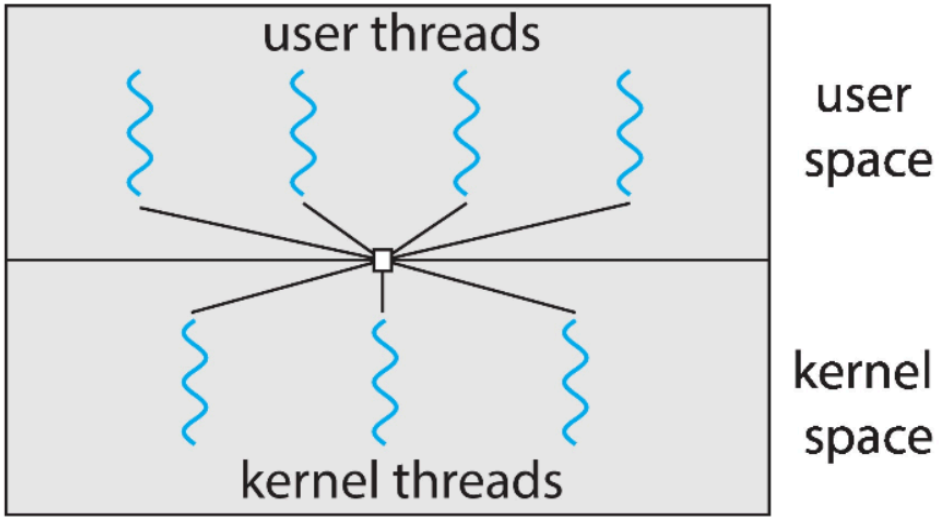
- Multiple user-level threads are dynamically mapped to multiple kernel threads.
- The operating system manages the mapping, allowing the number of kernel threads to be adjusted based on the workload.

Concurrency:

- Provides better concurrency than the Many-to-One model since multiple kernel threads can run in parallel on multicore systems.
- Avoids the bottleneck caused by a single kernel thread, as in the Many-to-One model.

Scalability:

- The operating system creates enough kernel threads to ensure that user-level threads can execute efficiently without overwhelming system resources.



TWO-LEVEL MODEL:

Mapping:

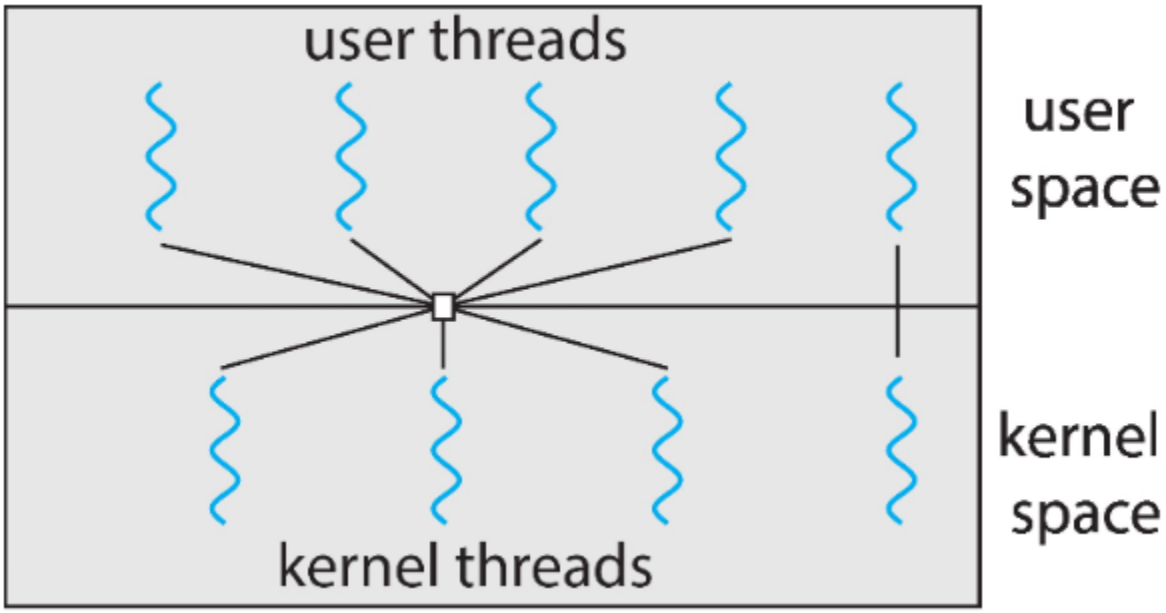
- Like the Many-to-Many model, multiple user-level threads can be mapped to multiple kernel threads.
- However, it also allows for binding a specific user-level thread to a specific kernel thread for cases where such binding is necessary.

Concurrency:

- This model supports high concurrency since multiple user threads can execute in parallel on multiple kernel threads.
- The ability to bind a user thread to a kernel thread provides additional flexibility for certain tasks that require direct control over the mapping.

Flexibility:

- Offers the scalability benefits of the Many-to-Many model, while also allowing for deterministic behavior (binding) where required.



Program to create Threads in Linux

In this post we discuss the use of **pthread_create** function create threads in linux. Various program to create threads in Linux are discussed below that shows how to create threads, how to pass input to thread and how to return value from thread.

Syntax

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

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                        void *(*start_routine) (void *), void arg);
```

The first parameter is the buffer which will contain the ID of the new thread, if pthread_create is successful. The second parameter specifies the attributes of the thread. This parameter is generally NULL until you want to change the default settings. The third parameter is the name the function which the thread will execute. Hence, everything that you want the thread to do should be defined in this function. Lastly, the fourth parameter is the input to the function in the third parameter. If the function in the third parameter does not take any input then the fourth parameter is NULL.

Program 1: Program to create threads in linux. Thread prints 0-4 while the main process prints 20-24.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
void *thread_function(void *arg);
int i,j;
int main() {
pthread_t a_thread;  //thread declaration
pthread_create(&a_thread, NULL, thread_function, NULL);
//thread is created
pthread_join(a_thread, NULL); //process waits for thread to finish.
//Comment this line to see the difference
printf("Inside Main Program\n");
for(j=20;j<25;j++)
{
printf("%d\n",j);
sleep(1);
}
}
void *thread_function(void *arg) {
// the work to be done by the thread is defined in this function
printf("Inside Thread\n");
for(i=0;i<5;i++)
{
printf("%d\n",i);
sleep(1);
}
}
```

Output

0
1
2
3
4
Inside Main Program
20
21
22
23
24

How it works?

pthread_create() creates a new thread which starts to execute thread_function. This function creates a loop which prints 0-4. The sleep function makes the thread go to sleep after each digit is printed. pthread_join() makes the main function wait until the newly created thread finishes its execution. So the control returns to the main function only when the thread finishes. Then the main function prints “Inside Main program” and executes the loop from 20-24.

How a thread returns a value to the main process?

pthread_exit() is used to return a value from the thread back to the main process. The program below shows this. The program also shows how to pass value to a thread from the main process.

EXAMPLE CODE FOR JOINING 10 THREADS:

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

THREAD SCHEDULING:

Contention scope refers to the domain in which threads compete for resources, such as processor time.

1. Process Contention Scope (PCS)

- In PCS, threads within the same process compete for available kernel threads.
- This means user-level threads are scheduled onto kernel threads based on priorities set by the programmer.
- Many-to-one and many-to-many threading models both rely on the kernel threads' availability for PCS to function.
- PTHREAD_SCOPE_PROCESS: This represents PCS and is controlled at the user level by the programmer.

2. System Contention Scope (SCS)

- Here, the kernel handles the scheduling of threads onto physical CPUs.
- This means kernel threads themselves compete for CPU time (rather than user-level threads).
- SCS typically applies to models like one-to-one threading, where each user thread corresponds to a kernel thread.
- PTHREAD_SCOPE_SYSTEM: This uses SCS, and scheduling is determined by the kernel.

THREADING ISSUES:

1. Semantics of fork() and exec() System Calls

- fork() creates a child process by copying the address space of the parent process.
    - Threading Issue: If fork() is called in a multithreaded process, only the calling thread is duplicated in the child process, which may cause inconsistencies if other threads were performing tasks.
- exec() replaces the current process's memory space with a new program. It is often used after fork() to run a different program in the child process.
    - Thread-related problems are mitigated after exec(), as threads no longer exist in the new program.

2. Signal Handling

Signal Handling in Multithreaded Processes

1. **Default and User-Defined Handlers:**
    - Every signal has a default handler provided by the kernel.

    - A **user-defined signal handler** can override this default, giving developers control over how the signal is processed.
2. **Synchronous Signals:**
    - Examples: Division by zero, illegal memory access.
    - These are caused directly by the **thread's actions**, so they must be delivered to the thread that caused them.
3. **Asynchronous Signals:**
    - Examples: Process termination signals (like SIGTERM or SIGKILL), often triggered externally or by the user (Ctrl+C).
    - **Delivery in a Multithreaded Process:**
        - The signal could go to:
            - The thread **most relevant to the signal**.
            - **All threads** in the process.
            - **Certain threads** explicitly designated to handle the signal.
            - A **dedicated signal handler thread** for the entire process, ensuring centralized handling.
    - There's no clear universal standard for asynchronous signals in multithreaded environments. For instance:
        - A termination signal (e.g., SIGTERM) might terminate the entire process, thereby affecting all threads.
        - A pthread_kill() signal, on the other hand, specifically targets a single thread.

Practical Use Cases

1. **For Processes:**
    - The function kill(pid_t pid, int signal) sends signals to an entire process.
2. **For Threads:**
    - The function pthread_kill(pthread_t tid, int signal) delivers a signal to a **specific thread**, making it a more precise tool in multithreaded programs

3. Thread Cancellation

- Threads may need to be terminated before their tasks are complete.
- Two types of cancellation:
    - Asynchronous Cancellation: Immediately terminates the target thread.
        - Risk: It can leave shared resources in an inconsistent state if cleanup isn't handled properly.
    - Deferred Cancellation: The target thread checks periodically if it should terminate, allowing for safer resource deallocation.

Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

 . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

- Cancellation request: Invokes cancellation, but its effect depends on the thread's state.

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- Thread state: If cancellation is disabled, the request remains pending until enabled.
- Default behavior: Deferred cancellation occurs only when the thread reaches a cancellation point (e.g., pthread_testcancel()).
- Cleanup: When cancellation happens, a cleanup handler is invoked to handle resource management.

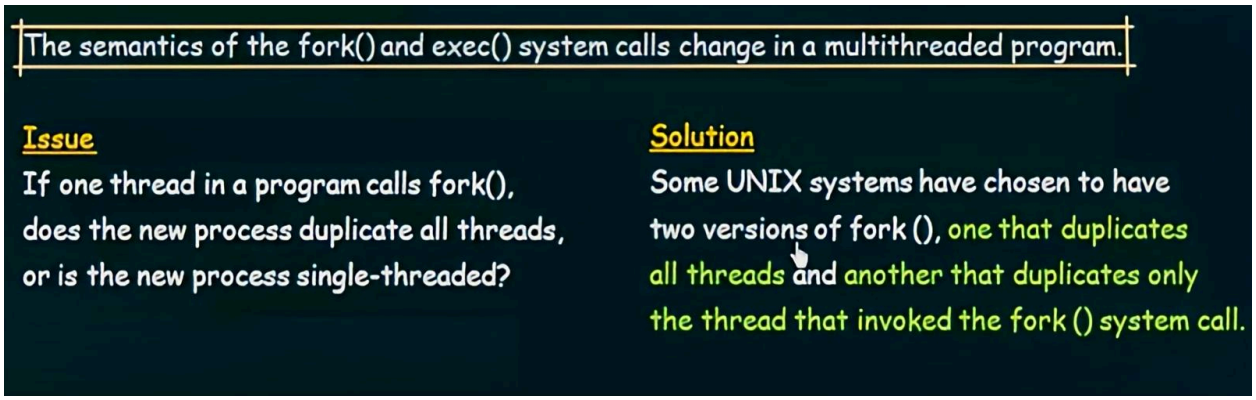
4. Thread-Local Storage (TLS)

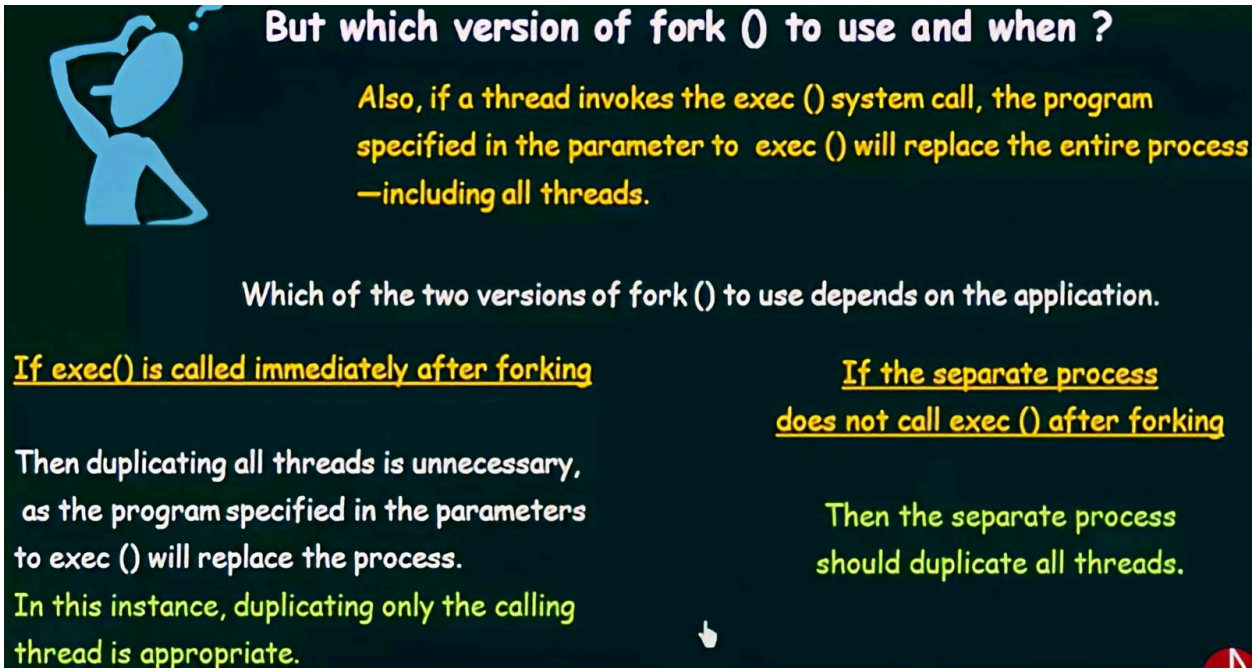
- TLS provides each thread with its own copy of a variable.
    - Why It's Important: Useful when threads need to maintain unique states without interfering with each other.
    - Threading issues arise when improperly used, as managing the lifecycle of thread-local variables can lead to memory leaks.
    - **TLS and Data Isolation**: Each thread has its own unique instance of the data, allowing for isolated operations without interference from other threads. This is especially useful in multi-threaded environments where shared data might lead to race conditions.
    - **When to Use TLS**: TLS is particularly handy when you don't control thread creation, like in thread pools or frameworks managing threads. It ensures that even in such scenarios, threads can safely maintain their own data state.

- **TLS vs. Local Variables**: While local variables are confined to the scope and lifetime of a single function, TLS persists across multiple function calls within the same thread, making it accessible throughout the thread's execution.
- **TLS and Static Data**: Like static data, TLS allows variables to maintain state between function invocations. However, the key difference is that static data is shared across threads, whereas TLS provides a distinct copy of data for each thread.

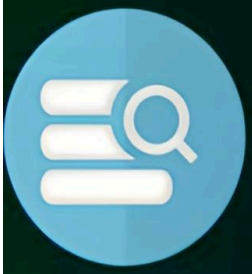
```
static _thread int threadID;
```

Threading issues:



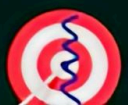


Thread cancellation is the task of terminating a thread before it has completed.



If multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.

When a user presses a button on a web browser that stops a web page from loading any further, all threads loading the page are canceled.

A thread that is to be canceled  is often referred to as the target thread.

Cancellation of a target thread may occur in two different scenarios:

1. Asynchronous cancellation: One thread immediately terminates the target thread.
2. Deferred cancellation: The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Where the difficulty with cancellation lies:

In situations where:

- Resources have been allocated to a canceled thread
- A thread is canceled while in the midst of updating data it is sharing with other threads.

Often, the OS will reclaim system resources from a canceled thread but will not reclaim all resources.

Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

One thread indicates that a target thread is to be canceled.

But cancellation occurs only after the target thread has checked a flag to determine if it should be canceled or not.

This allows a thread to check whether it should be canceled at a point when it can be canceled safely.

LINUX THREAD CREATION SYSTEM CALL:

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)
  - Flags control behavior

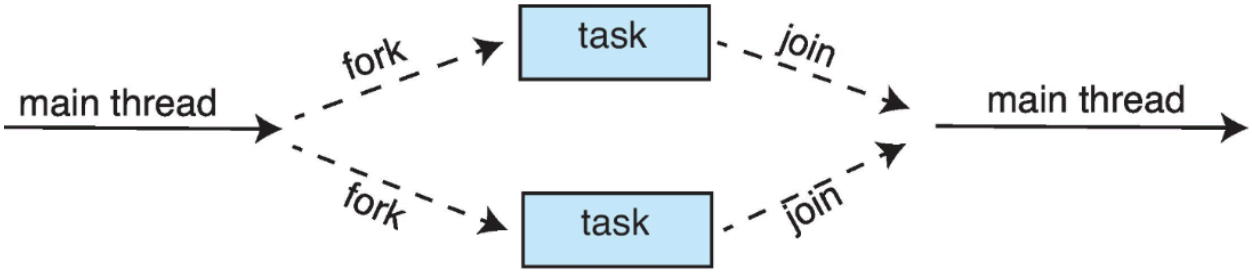
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- The `clone` system call is used by the Linux kernel to create a new process or thread.
- It is a versatile system call that can be used to create threads with varying degrees of sharing of resources such as memory, file descriptors, and signal handlers.
- Threads created using `clone` share the same memory space (unless specified otherwise) and are commonly used by threading libraries like pthreads to implement threads.

Implicit Threading: Delegates thread creation and management to compilers and runtime libraries instead of programmers, making parallel programming easier and less error-prone.

FORK-JOIN PARALLELISM:

Multiple threads (tasks) are forked, and then joined.
Fork-join parallelism is a programming model and parallel computing paradigm that involves splitting ("forking") a task into multiple subtasks, which are then executed concurrently across multiple processing units (e.g., CPU cores or threads).
Once all subtasks have completed their execution, the results are combined ("joined") to produce the final result.



CODES:

```c
#include <signal.h>
#include <unistd.h>

// Signal handler function
void handle_sigint(int sig) {
    printf("\nCaught signal %d (SIGINT). Exiting gracefully.\n", sig);
    // Exit the program when SIGINT is received
    _exit(0);
}

int main() {
    // Register the signal handler for SIGINT
    if (signal(SIGINT, handle_sigint) == SIG_ERR) {
        perror("Error registering signal handler");
        return 1;
    }

    printf("Press Ctrl+C to trigger SIGINT signal.\n");

    // Keep the program running indefinitely
    while (1) {
        printf("Running...\n");
        sleep(2); // Wait for 2 seconds between iterations
    }

    return 0;
}
```

Q) Write a C program using POSIX threads (pthreads) to compute the sum of a large array of integers in parallel. The program should:

1. Use multiple threads to divide the workload equally.
2. Store the array as a global variable and initialize it with random values.
3. Assign each thread a portion of the array to compute the partial sum.
4. Use pthread_join to collect the results from each thread.
5. Sum up the partial sums in the main thread to compute the total sum.

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

#define NUM_THREADS 4
#define ARRAY_SIZE 1000000

int global_array[ARRAY_SIZE]; // Shared array

// Function to initialize the array with random values
void initialize_array() {
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        global_array[i] = rand() % 1000;
    }
}

// Function to find the sum of elements in a portion of the array
void *sum_array(void *arg) {
    int thread_id = *((int *)arg);
    int start = thread_id * (ARRAY_SIZE / NUM_THREADS);
    int end = start + (ARRAY_SIZE / NUM_THREADS);
    int sum = 0;

    // Calculate the sum of elements in the assigned portion of the array
    for (int i = start; i < end; ++i) {
        sum += global_array[i];
    }

    return (void *)(long)sum; // Return the sum as a void pointer
}
```

```c
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    void *thread_results[NUM_THREADS];
    long total_sum = 0;

    // Initialize the array with random values
    initialize_array();

    // Create threads to compute the sum of array elements
    for (int i = 0; i < NUM_THREADS; ++i) {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, sum_array, (void *)&thread_args[i]);
    }

    // Join threads and collect results
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], &thread_results[i]);
        total_sum += (long)thread_results[i]; // Accumulate the partial sums
    }

    printf("Total sum of array elements: %ld\n", total_sum);

    return 0;
}
```

THREAD TRACE:

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

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    br int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

assert(rc == 0); ensures threads are successfully created and joined.

p1 runs mythread("A"). p2 runs mythread("B").

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Table 26.1: **Thread Trace (1)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Table 26.2: **Thread Trace (2)**

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Table 26.3: **Thread Trace (3)**

These are all the possible thread traces of code above.

CHAPTER 4 EXERCISES

4.9 When Does Multithreading Provide Better Performance on a Single-Processor System?

1. **I/O-Bound Operations:** A multithreaded solution can effectively utilize idle CPU time during I/O operations by allowing other threads to execute.
2. **Improved Responsiveness:** In interactive applications (e.g., GUI-based systems), multithreading can ensure that user input is promptly handled while other threads manage background tasks.
3. **Latency-Bound Tasks:** Switching between threads can hide latencies in high-latency tasks (e.g., waiting for user input or network responses), making the system appear more responsive.
4. **Parallelism in Logical Tasks:** Even though threads are not truly parallel on a single processor, they can provide concurrency in logical workflows, such as handling multiple connections in a server application.

4.10 Which of the following components of program state are shared across threads in a multithreaded process?
a. Register values b. Heap memory c. Global variables d. Stack memory

(b) Heap memory ✅ (Shared)

- The heap is shared among all threads in a process, allowing them to allocate and access dynamic memory together.

(c) Global variables ✅ (Shared)

- Global variables exist in a shared memory space, making them accessible to all threads within the process.

4.13 Is it possible to have concurrency but not parallelism? Explain.

Yes, concurrency without parallelism is possible. Concurrency means tasks **overlap in progress**, while parallelism requires tasks to run **at the same time** on multiple processors.

4.15 Determine if the following problems exhibit task or data parallelism:

Using a separate thread to generate a thumbnail for each photo in a collection → Data Parallelism ✅

- Each thread processes a different photo independently.
- The **same operation (thumbnail generation)** is applied to different pieces of data (photos).

Transposing a matrix in parallel → Data Parallelism ✅

- The matrix is divided into smaller parts, and each thread works on a portion of the data.
- The **same operation (swapping elements)** is applied to different elements of the matrix.

A networked application where one thread reads from the network and another writes to the network → Task Parallelism ✅

- Different tasks (reading and writing) are handled by separate threads.
- The operations performed by each thread are **different**, making this task parallelism.

The fork-join array summation application described in Section 4.5.2 → Data Parallelism ✅

- The array is divided into smaller subarrays, and each thread processes a portion of the data independently.
- The **same operation (summation)** is applied to different parts of the array.

The Grand Central Dispatch (GCD) system → Task Parallelism ✅

- GCD is designed to **distribute different tasks** across available cores efficiently.
- It manages execution queues for different tasks, making it an example of task parallelism.

4.17 Consider the following code segment:

```
pid_t pid;

pid = fork();
if (pid == 0) { /* child process */
   fork();
   thread_create( . . .);
}
fork();
```

a. How many unique processes are created?

b. How many unique threads are created?

Final Count of Unique Processes

1. First fork() → 2 processes (Parent + Child)
2. Second fork() (inside first child process) → 3 processes

3. Third fork() (executed by all 3 existing processes) → 6 processes

Total unique processes = 6 ✅

Thread Creation Analysis

- The thread_create() call only executes in the second process (first child process created by the first fork()).
- This creates one new thread in that specific process.

Total unique threads = 1 (excluding the main thread of each process) ✅
Each process has its own main thread, but only one additional thread is created.

Discuss all multithreading modules along with their advantages and disadvantages.

1. Many-to-One Model

  - Multiple user-level threads are mapped to a single kernel thread.
  - Thread management is handled by the user-level thread library.
- Advantages:
  - Efficient: Context switching between user-level threads is fast (since the OS doesn't need to intervene).
  - Lightweight: No system call is needed for thread creation and management.
- Disadvantages:
  - No true parallelism: Since only one kernel thread exists, even if the system has multiple cores, only one user thread can execute at a time.
  - Blocking issue: If one thread performs a blocking system call (e.g., I/O operation), the entire process is blocked, affecting all user threads.

2. One-to-One Model

  - Each user-level thread is mapped to a separate kernel thread.
  - The OS manages threads and schedules them on available cores.
- Advantages:
  - True parallelism: Multiple threads can run simultaneously on multi-core processors.
  - No blocking issue: If one thread blocks (e.g., waiting for I/O), other threads can continue executing.
- Disadvantages:
  - Higher resource usage: Creating a new user thread requires creating a kernel thread, which is expensive.

- Scalability issue: The OS must manage many kernel threads, leading to potential overhead.

3. Many-to-Many Model (Hybrid Model)

- Allows many user-level threads to be mapped to a smaller or equal number of kernel threads.
- The OS dynamically allocates kernel threads based on workload.

- Advantages:
  - Combines the best of both worlds: Efficient resource utilization and support for parallel execution.
  - No single-point blocking issue: If one user thread blocks, other threads mapped to different kernel threads can continue.
  - More scalable: Unlike one-to-one models, this avoids excessive kernel thread creation.
- Disadvantages:
  - More complex implementation: The OS must efficiently manage the mapping between user and kernel threads.

Model	User-Level to Kernel-Level Mapping	Parallelism	Blocking Issue	Overhead
Many-to-One	Many user threads → 1 kernel thread	❌ No	✅ Yes (one blocks all)	✅ Low
One-to-One	1 user thread → 1 kernel thread	✅ Yes	❌ No	❌ High
Many-to-Many	Many user threads → Some kernel threads	✅ Yes	❌ No	🔄 Medium

4.20 Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processing cores.

b. The number of kernel threads allocated to the program is equal to the number of processing cores.

c. The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads.

Scenario a: The number of kernel threads allocated is less than the number of processing cores

- Implication: Underutilization of CPU cores
- Since the number of kernel threads is fewer than the available cores, not all cores will be utilized effectively.
- Some cores may remain idle, reducing overall performance.
- User-level threads must compete for the limited number of kernel threads, leading to increased context switching at the user-level thread scheduler.
- Result: Poor performance due to wasted computational resources.

Scenario b: The number of kernel threads allocated is equal to the number of processing cores

- Implication: Efficient core utilization with minimal overhead
- Each core can execute one kernel thread at a time, ensuring maximum CPU utilization without unnecessary overhead.
- User-level threads will still be multiplexed onto these kernel threads, but the system can balance workload effectively.
- Result: Optimal performance, as all cores are utilized without excessive thread management overhead.

Scenario c: The number of kernel threads allocated is greater than the number of processing cores but less than the number of user-level threads

- Implication: Increased parallelism but also increased context switching
- Since there are more kernel threads than cores, the OS must perform context switching to manage them.
- While this allows more user-level threads to run in parallel, too many kernel threads may increase overhead from frequent thread scheduling.
- Result: Moderate performance—better than Scenario (a), but possibly worse than Scenario (b) if thread-switching overhead becomes too high.

4.19 The program shown in Figure 4.23 uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

Output:

CHILD: value = 5

PARENT: value = 0

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

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

   pid = fork();

   if (pid == 0) { /* child process */
      pthread_attr_init(&attr);
      pthread_create(&tid,&attr,runner,NULL);
      pthread_join(tid,NULL);
      printf("CHILD: value = %d",value); /* LINE C */
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE P */
   }
}

void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```

4.22 Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the mini- mum value. For example, suppose your program is passed the integers 90 81 78 95 79 72 85 The program will report The average value is 82 The minimum value is 72 The maximum value is 95 The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent

thread will output the values once the workers have exited. (We could obviously expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
// Global variables to store results
double average;
int min, max;
// Function prototypes
void *calculate_average(void *param);
void *calculate_min(void *param);
void *calculate_max(void *param);

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s num1 num2 num3 ...\n", argv[0]);
        return 1;
    }

    int count = argc - 1;
    int *numbers = malloc(count * sizeof(int));

    // Convert command-line arguments to integers
    for (int i = 0; i < count; i++) {
        numbers[i] = atoi(argv[i + 1]);
    }

    // Create an array of worker threads
    pthread_t workers[3];
```

```c
    // Create threads
    pthread_create(&workers[0], NULL, calculate_average, numbers);
    pthread_create(&workers[1], NULL, calculate_min, numbers);
    pthread_create(&workers[2], NULL, calculate_max, numbers);

    // Wait for threads to finish
    for (int i = 0; i < 3; i++) {
        pthread_join(workers[i], NULL);
    }

    // Print results
    printf("The average value is %.2f\n", average);
    printf("The minimum value is %d\n", min);
    printf("The maximum value is %d\n", max);

    // Free allocated memory
    free(numbers);

    return 0;
}

// Function to calculate average
void *calculate_average(void *param) {
    int *numbers = (int *)param;
    int count = 0, sum = 0;

    // Find array size dynamically
    while (numbers[count] != '\0') count++;

    for (int i = 0; i < count; i++) {
        sum += numbers[i];
    }
    average = (double)sum / count;
    pthread_exit(0);
}
```

```c
// Function to calculate minimum
void *calculate_min(void *param) {
    int *numbers = (int *)param;
    int count = 0;

    while (numbers[count] != '\0') count++;

    min = numbers[0];
    for (int i = 1; i < count; i++) {
        if (numbers[i] < min) {
            min = numbers[i];
        }
    }
    pthread_exit(0);
}

// Function to calculate maximum
void *calculate_max(void *param) {
    int *numbers = (int *)param;
    int count = 0;

    while (numbers[count] != '\0') count++;

    max = numbers[0];
    for (int i = 1; i < count; i++) {
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    pthread_exit(0);
}
```

4.23 Write a multithreaded program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
// Function prototype
void *find_primes(void *param);
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }
    int limit = atoi(argv[1]);
    if (limit < 2) {
        printf("No prime numbers available.\n");
        return 1;
    }
    pthread_t prime_thread;
    pthread_create(&prime_thread, NULL, find_primes, &limit);
    pthread_join(prime_thread, NULL);
    return 0;
}
```

```c
// Function to find and print prime numbers
void *find_primes(void *param) {
    int limit = *(int *)param;
    printf("Prime numbers up to %d: ", limit);

    for (int num = 2; num <= limit; num++) {
        bool is_prime = true;
        for (int i = 2; i * i <= num; i++) {
            if (num % i == 0) {
                is_prime = false;
                break;
            }
        }
        if (is_prime) {
            printf("%d ", num);
        }
    }
    printf("\n");
    pthread_exit(0);
}
```

```c
#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Function prototype

void *find_primes(void *param);

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

   if (argc != 2) {

      printf("Usage: %s <number>\n", argv[0]);
```

```
        return 1;
    }
    int limit = atoi(argv[1]);
    if (limit < 2) {
        printf("No prime numbers available.\n");
        return 1;
    }
    pthread_t prime_thread;
    pthread_create(&prime_thread, NULL, find_primes, &limit);
    pthread_join(prime_thread, NULL);
    return 0;
}
// Function to find and print prime numbers
void *find_primes(void *param) {
    int limit = *(int *)param;
    printf("Prime numbers up to %d: ", limit);
    for (int num = 2; num <= limit; num++) {
        bool is_prime = true;
        for (int i = 2; i * i <= num; i++) {
            if (num % i == 0) {
                is_prime = false;
                break;
            }
```

```
        }
        if (is_prime) {
            printf("%d ", num);
        }
    }
    printf("\n");
    pthread_exit(0);
}
```

4.27 The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Formally, it can be expressed as:

$$fib_0 = 0$$
$$fib_1 = 1$$
$$fib_n = fib_{n-1} + fib_{n-2}$$

Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish. Use the techniques described in Section 4.4 to meet this requirement.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
// Global array to store Fibonacci sequence
int *fibonacci;
int n;
// Function prototype
void *generate_fibonacci(void *param);
```

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number_of_terms>\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    if (n <= 0) {
        printf("Please enter a positive integer.\n");
        return 1;
    }
    // Allocate memory for Fibonacci sequence
    fibonacci = (int *)malloc(n * sizeof(int));
    if (fibonacci == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    pthread_t fib_thread;
    pthread_create(&fib_thread, NULL, generate_fibonacci, NULL);
    pthread_join(fib_thread, NULL);
    // Print Fibonacci sequence
    printf("Fibonacci sequence: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", fibonacci[i]);
    }
    printf("\n");
    free(fibonacci);
    return 0;
}
// Function to generate Fibonacci sequence
void *generate_fibonacci(void *param) {
    if (n > 0) fibonacci[0] = 0;
    if (n > 1) fibonacci[1] = 1;
    for (int i = 2; i < n; i++) {
        fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
    }
    pthread_exit(0);
}
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
// Global array to store Fibonacci sequence
int *fibonacci;
int n;
// Function prototype
void *generate_fibonacci(void *param);

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number_of_terms>\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    if (n <= 0) {
        printf("Please enter a positive integer.\n");
        return 1;
    }
    // Allocate memory for Fibonacci sequence
    fibonacci = (int *)malloc(n * sizeof(int));
    if (fibonacci == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    pthread_t fib_thread;
    pthread_create(&fib_thread, NULL, generate_fibonacci, NULL);
    pthread_join(fib_thread, NULL);
    // Print Fibonacci sequence
    printf("Fibonacci sequence: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", fibonacci[i]);
    }
    printf("\n");
    free(fibonacci);
    return 0;
}

// Function to generate Fibonacci sequence
void *generate_fibonacci(void *param) {
```

```
    if (n > 0) fibonacci[0] = 0;
    if (n > 1) fibonacci[1] = 1;
    for (int i = 2; i < n; i++) {
        fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
    }
    pthread_exit(0);
}
```

4.14 Using Amdahl's Law, calculate the speedup gain for the following applications:

• 40 percent parallel with (a) eight processing cores and (b) sixteen processing cores.

• 67 percent parallel with (a) two processing cores and (b) four processing cores.

• 90 percent parallel with (a) four processing cores and (b) eight processing cores.

Parallelism	Cores (P)	Speedup (S)
40%	8	**1.54**
40%	16	**1.60**
67%	2	**1.50**
67%	4	**2.01**
90%	4	**3.08**
90%	8	**4.71**

WAYS TO IMPLEMENT THREAD LIBRARY:

A thread library provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library.

The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

1. True or False:
    a. Threads that are part of the same process share the same stack pointer.
    False; each thread has its own stack and stack pointer so it can have own call stack and local.
    b. The OS provides the illusion to each thread that it has its own address space.
    False; each process, not thread, has its own address space.
    c. Threads that are part of the same process share the same general-purpose registers. True
    False; each thread must have own registers (saved and restored when that thread is not executing)
2. For the next questions, assume the following code is compiled and run on a modern Linux machine. Assume any irrelevant details have been omitted and that no routines, such as pthread_create() or pthread_join(), ever fail.

```
volatile int balance = 0;
void *mythread(void *arg) {
 int result = 0;
result = result + 200;
balance = balance + 200;
printf("Result is %d\n", result);
printf("Balance is %d\n", balance);
 return NULL; }
int main(int argc, char *argv[]) {
pthread_t p1, p2;
 pthread_create(&p1, NULL, mythread, "A");
pthread_create(&p2, NULL, mythread, "B");
 pthread_join(p1, NULL);
pthread_join(p2, NULL);
printf("Final Balance is %d\n", balance);
return 0;
}
```

    a. How many total threads are part of this process?
    3: Main thread plus two created threads
    b. When thread p1 prints "Result is %d\n", what value of result will be printed?
    200: 'result' is a local variable allocated on the stack; each thread has its own private copy which only it increments, therefore there are no race conditions.
    c. When "Final Balance is %d\n" is printed, what value of balance will be printed?
    balance is allocated on the heap and shared between the two threads that are each accessing it without locks; there is a race condition.

a) Determine if the following problems exhibit task or data parallelism:

I. Search of a text file for a string **S**.
Data parallelism

II. Find the product of the integers from 5 to 500 and the average of the integers from 5 to 500. Task parallelism

III. Find the dot product of two different matrices.
Task parallelism

IV. Convert the first character of each string in an array to uppercase. Data parallelism

V. Sum of integers from 1 to n.
Data parallelism

a. Determine if the following problems exhibit task/data parallelism or both. Give reasons in one or two lines each:

I. Transpose a matrix in parallel.
II. Calculating the salary of all employees of a company.
III. Taking input from IoT cameras in FAST for single abnormal event detection.
IV. Taking input from IoT cameras in FAST for scene detection (e.g., books, students, blackboard, teacher. detection modules working in parallel for classroom scene detection.
V. National ID card-making procedure at multiple counters of NADRA.

solution:
a) data
b) data
c) data
d) task
e) task

a) Suppose you have written a multithreaded C program using Pthread library that creates ten threads to solve a problem. You also determine that 20% of execution remains serial. Now answer the following: [1.5 + 1.5]

i. **How much speedup** do we get if we run the program on a computing unit having ten cores?

$$S = \frac{1}{0.2 + \frac{0.8}{10}}$$ **= 3.75**

S is speedup, 0.2 is serial portion, 1-0.2 is parallel portions and N =10 (cores) **[0.75]**

ii. **Explain** what happens if the same program runs on a single CPU, single core system.
They run by taking turns as per CPU schedular policy **[1.0]**. Execution time increases as each waits its turn and the added overhead of multiple context switches **[0.5]**.

Recall the threading concepts covered from the textbook. Now, **compare** the following:

i. Concurrent execution vs Parallel execution [1]
Concurrent execution refers to the execution of multiple tasks in different time periods on a single CPU core [0.5]. In parallel execution, multiple tasks run at a time due to the availability of multiple CPU cores [0.5].

ii. Task parallelism vs Data Parallelism [1]
Task parallelism involves dividing a task into smaller independent subtasks that perform different operations on same/different data [0.5]. In data parallelism, many independent subtasks perform the same operations on different subsets of data [0.5].

b) Write a **C program** using the Pthread library that performs a parallel assignment of random numbers to an integer array of three hundred elements using three threads. *Note: There will be no partial award if incorrect syntax or logic.* [1.5 (syntax) + 1.5 (logic) = 3]

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

#define ARRAY_SIZE 300
#define NUM_THREADS 3

int array[ARRAY_SIZE];

void *assignRandomNumbers(void *thread_id) {
    long tid = (long)thread_id;
    int start_index = tid * (ARRAY_SIZE / NUM_THREADS);
    int end_index = start_index + (ARRAY_SIZE / NUM_THREADS);
    for (int i = start_index; i < end_index; i++) { array[i] = rand() % 100; }
    pthread_exit(NULL);
}
int main() {
    pthread_t threads[NUM_THREADS]; int rc; long t;
    srand(time(NULL)); // Seed for random number generator
    for (t = 0; t < NUM_THREADS; t++) {
        rc = pthread_create(&threads[t], NULL, assignRandomNumbers, (void *)t);
        if (rc) {
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    // Wait for all threads to finish
    for (t = 0; t < NUM_THREADS; t++) { pthread_join(threads[t], NULL); }
    exit(0);
}
```

Consider the situation illustrated in Figure 1 where two processes, running in parallel, try to execute the fork () system call at the same time. During execution, both fork () system calls request pid, which involves the identifier **next_available_pid**. Now, explain the synchronization problem shown in this diagram by answering the following questions: *Note: No coding required.* [1.5 + 1.5]

i. How do they get the same value? Explain.
It happens due to race condition on global variable **next_available_pid [0.5]**. While running in parallel on two cores, each reads the previous values 2614, increments it to 2615 and uses and writes it to next_available_pid **[1.0]**

ii. Discuss a solution using which they get a unique value?
They need mutual exclusion during their execution **[0.5]**. This means one needs to busy wait while the other updates the global variable next_availalbe_pid. However, the waiting process should be allowed to update immediately. **[1.0]**.

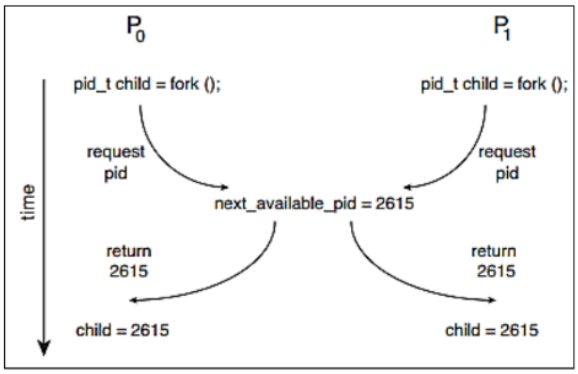
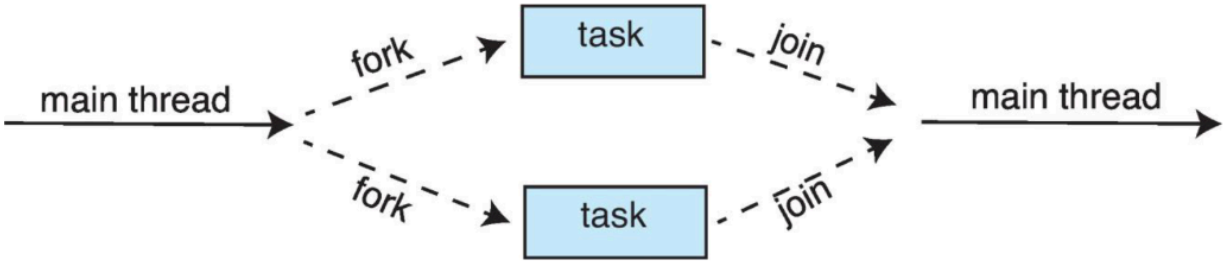


Figure 1: Q2 part (c)

c. Explain Fork-join operation and illustrate by the diagram?
fork-join is issued by one thread when it wants to stop and wait for the termination of another thread. On termination of the target thread, the original thread resumes execution. Grading Scheme: 1 point for this thread waits for the joined-to thread to finish



(A)

1. What is a thread-join operation? **[2]**
   A thread-join operation allows the master thread to wait for the created child thread to finish.

2. Consider the following two threads, to be run **concurrently** in a shared memory (all variables are shared between the two threads): Assume a single-processor system, that load and store are atomic, that x is initialized to 0 before either thread starts, and that x must be loaded into a register before being incremented (and stored back to memory afterwards). The following questions consider the final value of x after both threads have completed. **[2*4=8]**

Thread A	Thread B

for (i=0; i<5; i++)
{
x = x + 1;
}
	for (j=0; j<5; j++)
	{
	x = x + 2;
	}

i) Give a concise proof why x≤15 when both threads have completed.
Given the atomic statements x=x+1 and x=x+2 and alternate execution of threads x takes values based on which tread scheduled to runs first. In case TA runs, it takes values: 1,3,4,6,7,9,10,12,13,15 and for TB running first values are: 2,3,5,6,8,9,11,12,14. So the code shown for TA and TB will always remain <= 15.

ii) Give a concise proof why x≠1 when both threads have completed.
X is initially zero. Complete execution of both threads guarantees that value of x will be either 14 or 15 but not 1. See part (i) for more details.

iii) What needs to be saved and restored on a context switch between two threads in the same process? What if the two threads are in different processes? Be explicit.
All registers (this automatically includes stack pointer) will be stored in TCB and restored. In case threads belongs to two different processes PCBs will saved and restored as well. *Please note that we have not covered thread scheduling beyond this.*

iv) Under what circumstances can a multithreaded program complete more quickly than a non-multithreaded program? Keep in mind that multithreading has context-switch overhead associated with it.
Creating too many threads (e.g. ~100s) each running few instructions will slow things down due to context switching overhead. Therefore, multithreading benefits if a program has sizable parallelizable portions and each of those has ~10s of language statements executing cpu/io bound tasks. This will make total context switch time (in ~10 microseconds per context switch) negligible.

a) Consider Google's Chrome browser as a multi-process architecture. What benefit would this implementation provide to the user? [2]

- If each tab is a separate process, then this provides protection and security as each process memory will be sperate and unaccusable by other processes. can be accessed by other processes.
- Each process can be scheduled on a different core of the processes. Thus, response time will be improved.

b) Show how many processors are needed to gain a 3.5 times speedup if the serial portion is 20%. [2]

speedup = 1 / (serial portion + (parallel portion /N) where N is number of processors. Putting values
3.5 = 1 / (0.2 + (0.8/N))
3.5 (0.2 + (0.8/N)) = 1 ➔ 0.7 + (2.8/N) = 1 ➔ 2.8 / N = 0.3 ➔ N = 2.8 / 0.3 ➔ N = 9.33 ~ 10 processors.

MORE ON THREADS CODE:

```c
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *helloWorld(void *vargp) {
    sleep(1);
    printf("Hello World \n");
    return NULL;
}
int main() {
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, helloWorld, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

ROLE OF PTHREAD_EXIT(VOID *RETVAL):

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

void* myThread(void* arg) {
    long thread_arg = (long)arg;
    printf("Thread ID: %ld\n", thread_arg);
    printf("Hello from thread %ld\n", (long)arg);
}

int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, myThread, (void*)1);
    pthread_exit(NULL);
    printf("Hello from main thread\n");
    // This line will not be executed because of pthread_exit
    pthread_join(t1, NULL);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out
Thread ID: 1
Hello from thread 1
```

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

void* myThread(void* arg) {
    long thread_arg = (long)arg;
    printf("Thread ID: %ld\n", thread_arg);
    printf("Hello from thread %ld\n", (long)arg);
}

int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, myThread, (void*)1);
    printf("Hello from main thread\n");
    pthread_join(t1, NULL);
    pthread_exit(NULL);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out
Hello from main thread
Thread ID: 1
Hello from thread 1
```

```c
#include <pthread.h>

void* myThread(void* arg) {
    long thread_arg = (long)arg;
    printf("Thread ID: %ld\n", thread_arg);
    printf("Hello from thread %ld\n", (long)arg);
    pthread_exit(NULL);
    // return NULL; // This line is equivalent to pthread_exit(NULL)
    // return 0; // This line is also equivalent to pthread_exit(NULL)
}

int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, myThread, (void*)1);
    printf("Hello from main thread\n");
    pthread_join(t1, NULL);
    return 0;
}
```


```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out
Hello from main thread
Thread ID: 1
Hello from thread 1
```

```c
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5
void *BusyWork(void *threadid) {
    long tid = (long)threadid;
    printf("Thread %ld is working...\n", tid);
    pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc;
    long t;
    void *status;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    for (t = 0; t < NUM_THREADS; t++) {
        printf("Main: creating thread %ld\n", t);
        rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_attr_destroy(&attr);
    for (t = 0; t < NUM_THREADS; t++) {
        pthread_join(thread[t], &status);
    }
    printf("Main: program completed.\n");
    pthread_exit(NULL);
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out
Main: creating thread 0
Main: creating thread 1
Thread 0 is working...
Main: creating thread 2
Thread 1 is working...
Main: creating thread 3
Thread 2 is working...
Main: creating thread 4
Thread 4 is working...
Thread 3 is working...
Main: program completed.
```

- **pthread_detach** ensures you don't have to call **pthread_join**, and the thread's memory/resources are cleaned up automatically after it finishes.
- Useful for background threads that don't return important results.

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void* myThreadFunction(void* arg) {
    printf("Thread is running...\n");
    sleep(2);
    printf("Thread is exiting...\n");
    return NULL;
}
int main() {
    pthread_t thread;
    if (pthread_create(&thread, NULL, myThreadFunction, NULL) != 0) {
        perror("Failed to create thread");
        return 1;
    }
    // Detach the thread
    if (pthread_detach(thread) != 0) {
        perror("Failed to detach thread");
        return 1;
    }
    printf("Main thread is continuing while child thread runs detached...\n");
    sleep(3); // Wait to ensure the detached thread gets time to run
    printf("Main thread exiting.\n");
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out
Main thread is continuing while child thread runs detached...
Thread is running...
Thread is exiting...
Main thread exiting.
```

```
#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
// Signal handler function
void handle_sigint(int sig) {
    printf("\nCaught signal %d (SIGINT). Exiting gracefully...\n", sig);
    exit(0); // Exit the program
}
int main() {
    // Register the signal handler for SIGINT (Ctrl+C)
    if(signal(SIGINT, handle_sigint)==SIG_ERR) {
        perror("Unable to register signal handler");
        exit(EXIT_FAILURE);
    }
    printf("Press Ctrl+C to trigger the signal handler...\n");
    // Set up a signal handler for SIGTERM
    if (signal(SIGTERM, handle_sigint) == SIG_ERR) {
        perror("Unable to register signal handler");
        exit(EXIT_FAILURE);
    }
    printf("Press Ctrl+C or send SIGTERM to trigger the signal handler...\n");
    // Infinite loop to keep the program running
    while (1) {
        printf("Running... Press Ctrl+C to exit.\n");
        sleep(1);
    }
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out
Press Ctrl+C to trigger the signal handler...
Press Ctrl+C or send SIGTERM to trigger the signal handler...
Running... Press Ctrl+C to exit.
Running... Press Ctrl+C to exit.
Running... Press Ctrl+C to exit.
Running... Press Ctrl+C to exit.
^C
Caught signal 2 (SIGINT). Exiting gracefully...
```

2 ways:

```c
#define ARRAY_SIZE 18
#define NUM_THREADS 3

int arr[ARRAY_SIZE];
void* assignRandom()
{
    for (int i=0;i<ARRAY_SIZE;i++)
    {
        arr[i] = rand() % 100;
    }
    return NULL;
}
```

```c
void *minarray()
{
    int min= arr[0];
    for (int i=1;i<ARRAY_SIZE;i++)
    {
        if (arr[i]<min)
        {
            min = arr[i];
        }
    }
    int *min_ptr=malloc(sizeof(int));
    *min_ptr=min;
    pthread_exit((void *)min_ptr);
}
```

```c
void *maxarray()
{
    int max=arr[0];
    for(int i=1;i<ARRAY_SIZE;i++)
    {
        if(arr[i]>max)
        {
            max=arr[i];
        }
    }
    int *max_ptr=malloc(sizeof(int));
    *max_ptr =max;
    pthread_exit((void *)max_ptr);
}
```

```c
void *sumarray(void *arg)
{
int sum=0;
int tid = *((int *) arg);
int start = tid*(ARRAY_SIZE/NUM_THREADS);
int end =start+(ARRAY_SIZE/NUM_THREADS);
for (int i=start;i<end;i++)
{
    sum+=arr[i];

}
int *sum_ptr = malloc(sizeof(int));
*sum_ptr = sum;
pthread_exit((void*) sum_ptr);
}
```

```c
int main()
{
    pthread_t threads[NUM_THREADS];
    pthread_t thread1;
    void *thread_results[NUM_THREADS];
    int total_sum=0;
    int thread_ids[NUM_THREADS];
    srand(time(NULL));
    pthread_create(&thread1,NULL,assignRandom,NULL);
    pthread_join(thread1,NULL);
```

```c
for(int i=0;i<NUM_THREADS;i++)
    {
        thread_ids[i] = i;
        pthread_create(&threads[i],NULL,sumarray,&thread_ids[i]);
    }
    for(int i=0;i<NUM_THREADS;i++)
    {
        pthread_join(threads[i],&thread_results[i]);
        total_sum+= *((int*) thread_results[i]);
        free(thread_results[i]); // Free the allocated memory for each thr
    }
    printf("Sum of array elements: %d\n",total_sum);
    pthread_t t2;
    void *min_result;
    pthread_create(&t2,NULL,minarray,NULL);
    pthread_join(t2,&min_result);
    printf("Minimum element in array: %d\n", *((int *)min_result));
    free(min_result); // Free the allocated memory for the minimum result
    pthread_t t3;
    void *max_result;
    pthread_create(&t3,NULL,maxarray,NULL);
    pthread_join(t3,&max_result);
    printf("Maximum result in array: %d\n", *((int *)max_result));
    free(max_result);
    pthread_exit(NULL);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out
Random numbers assigned to array:
79 46 7 77 61 95 84 24 65 55 46 38 77 98 81 90 12 51
Sum of array elements: 1086
Minimum element in array: 7
Maximum result in array: 98
```

```c
#define ARRAY_SIZE 7
#define NUM_THREADS 3
int arr[ARRAY_SIZE];
int avg;
int min,max;
void *avgarr(void *arg)
{
int sum=0;
for(int i=0;i<ARRAY_SIZE;i++)
{
sum += arr[i];
}
avg =(int)sum/ARRAY_SIZE;
pthread_exit(0);
}
```

```c
void *minarr(void *arg)
{
min=arr[0];
for(int i=1;i<ARRAY_SIZE;i++)
{
if(arr[i] < min)
{
min = arr[i];
}
}
pthread_exit(0);
}
void *maxarr(void *arg)
{
max=arr[0];
for(int i=1;i<ARRAY_SIZE;i++)
{
if(arr[i] > max)
{
max = arr[i];
}
}
pthread_exit(0);
}
```

```c
int main(int argc, char *argv[])
{
if(argc!=8)
{
printf("Usage: %s <arrayelements>", argv[0]);
return 1;
}
for(int i=0;i<7;i++)
{
arr[i]=atoi(argv[i+1]);
}
pthread_t workers[3];
pthread_create(&workers[0], NULL, avgarr, NULL);
pthread_create(&workers[1], NULL, minarr, NULL);
pthread_create(&workers[2], NULL, maxarr, NULL);
    for (int i = 0; i < NUM_THREADS; i++)
    {
        pthread_join(workers[i], NULL);
    }

    printf("Average: %d\n", avg);
    printf("Min: %d\n", min);
    printf("Max: %d\n", max);
return 0;
}
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int *fibonacci;
int n;
void *generate_fibonacci(void *param);
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number_of_terms>\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    if (n <= 0) {
        printf("Please enter a positive integer.\n");
        return 1;
    }
    fibonacci = (int *)malloc(n * sizeof(int));
    if (fibonacci == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    pthread_t fib_thread;
    pthread_create(&fib_thread, NULL, generate_fibonacci, NULL);
    pthread_join(fib_thread, NULL);
    printf("Fibonacci sequence: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", fibonacci[i]);
    }
    printf("\n");
    free(fibonacci);
    return 0;
}
```

```c
// Function to generate Fibonacci sequence
void *generate_fibonacci(void *param) {
    if (n > 0) fibonacci[0] = 0;
    if (n > 1) fibonacci[1] = 1;
    for (int i = 2; i < n; i++) {
        fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
    }
    pthread_exit(0);
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out 4
Fibonacci sequence: 0 1 1 2
kinza@DESKTOP-LKI25JK:~$ ./out 10
Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
void *find_primes(void *param);
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }
    int limit = atoi(argv[1]);
    if (limit < 2) {
        printf("No prime numbers available.\n");
        return 1;
    }
    pthread_t prime_thread;
    pthread_create(&prime_thread, NULL, find_primes,&limit);
    pthread_join(prime_thread, NULL);
    return 0;
}
```

```c
void *find_primes(void *param) {
    int limit = *((int *)param);
    printf("Prime numbers up to %d: ", limit);
    for (int num = 2; num <= limit; num++) {
        bool is_prime = true;
        for (int i = 2; i * i <= num; i++) {
            if (num % i == 0) {
                is_prime = false;
                break;
            }
        }
        if (is_prime) {
            printf("%d ", num);
        }
    }
    printf("\n");
    pthread_exit(0);
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc pthreadcode.c -o out -lpthread
kinza@DESKTOP-LKI25JK:~$ ./out 10
Prime numbers up to 10: 2 3 5 7
```