4.1 - 4.7, 4.8, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20(Ex-9), 4.9 → pdf

OS BONUS EXERCISES

22K-0842, KINZA BCS-4I Date:

4.1 Three Examples where Multithreading provides better performance than a single-threaded solution.

① Image Processing :- When applying filters, edge detection, or transformations to an image, different sections of the image can be processed concurrently using multiple threads, significantly reducing execution time compared to single-threaded approach.

② Game Physics and AI Processing :- Modern games use multithreading to handle physics calculations, AI decision making, & rendering separately. This prevents frame rate drops and ensures smoother gameplay as compared to single-threaded system where these tasks would cause delays.

③ Sorting Large Datasets :- Algorithms like Merge sort and Quick sort can be parallelized using multithreading by sorting subarrays independently. This speeds up sorting operations significantly.

4.2 Speedup Calculation using Amdhals law. 60% parallelizable part.

a) 2 processing cores

a) Speedup = $\frac{1}{(1-0.6)+\frac{0.6}{2}}$

= 1.4285 times

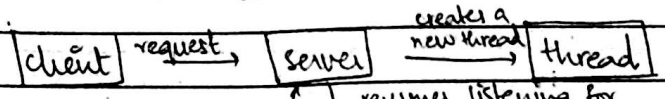
b) 4 processing cores.

b) Speedup = $\frac{1}{(1-0.6)+\frac{0.6}{4}}$

= 1.8181 times

4.3 Does multithreaded web server exhibit task parallelism or data parallelism.

A multithreaded web server exhibits task parallelism. Each thread handles a different client request independently, meaning multiple tasks (serving different web requests) are executed in parallel.



4.4 Differences between User level thread & Kernel level thread?

FEATURE	USER LEVEL	KERNEL LEVEL
① Managed by	User space libraries	Operating system (OS) kernel
② Context switching	Fast (no OS intervention)	Slower (requires system calls)
③ Scheduling	Handled by user-space scheduler	Scheduled by OS.

④ Blocking	If one thread blocks, all threads in the process may block	Other threads can continue running if one blocks
⑤ Portability	More portable across OS	OS-dependent

Under what circumstances is one better than the other?

User-level threads are better when lightweight & fast context switching is required.

Kernel-level threads are better when the application involves I/O operations or requires OS-level scheduling for better resource management.

4.5 Actions taken by kernel to context-switch between kernel level threads.

① Save the state of currently running thread (registers, program counter, stack pointer).
② Update the thread control block (TCB) of the current thread.
③ Select a new thread to run using the OS scheduler.
④ Load the state of the new thread from its TCB (restoring registers, program counter, stack pointer).
⑤ Resume execution of the new thread.

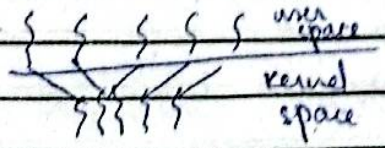
4.6 What resources are used when a thread is created? How do they differ from those used when a process is created?

Resource	Thread Creation	Process Creation
① Memory Allocation	Shares address space / memory with parent process.	Allocates separate memory space.
② Scheduling Data	Requires minimal scheduling information	Requires full process scheduling information.
③ PCB	No new PCB is created, just a TCB (Thread control block)	New PCB is created.
④ Execution Context	Shares execution context with other threads in the process.	Has its own execution context.

Creating a thread requires TCB i.e allocating a small data structure to hold a register set, stack, and priority.

Date:

4.7 Many-to many model → mapping done through LWPs.
Is it necessary to bind a real-time thread to an LWP? Explain
In many-to-many model, multiple user-level threads are mapped to a smaller or equal number of kernel-level threads (LWPs - light weight processes).



For real-time systems, binding a real-time thread to an LWP can be necessary because :-
Direct CPU scheduling :- Real-time threads often require predictable scheduling, which Kernel-level threads (LWPs) can provide.
Avoiding user-space scheduling delays :- If a real-time thread relies on user-level scheduling, it may experience delays.
Ensuring priority handling :- The OS can directly prioritize real-time threads when they are mapped to LWPs.

4.8 Two programming examples in which multithreading doesnot provide better performance than a single-threaded solution.
(1.) Simple Sequential Tasks (CPU-bound with dependencies)
For example :- Calculating fibonacci sequence recursively.
As each step depends on the previous ones, making parallelization difficult. The overhead of thread creation and context switching outweighs any potential speedup.
A single threaded solution avoids synchronization & thread management costs.

Single threaded version :-

```
{ if (n <= 1)
    return n;
  return fib(n-1) + fib(n-2);
}
```

Multithreaded version :-

```
{ if (n <= 1)
    return n;
  else {
    pthread_t t1, t2;
    pthread_create (&t1, NULL, fib, &a);
    pthread_create (&t2, NULL, fib, &b);
    pthread_join (t1, NULL);
    pthread_join (t2, NULL);
  }
}
```

Pg No.

Date:

(2) Single Shared I/O Resources

Multiple threads writing to the same file contend for locks making it slower than a single thread.

Single threaded is faster as no lock contention or thread synchronization overhead.

4.9 Under what circumstances does a multithreaded solution using multiple kernel threads perform better than a single threaded solution on a single-processor system?

Multithreaded solution using multiple kernel threads outperforms a single-threaded solution on a single-processor system under following circumstances :-

(1.) I/O Bound operations or Blocking operations : When threads frequently block for I/O (i.e file, database queries), or synchronization (i.e waiting for locks), multithreading allows CPU to switch to other ready threads

Pg No.

Date:

during blocking periods. This improves CPU utilization compared to single thread that would be idle during wait.

(2.) Improved Responsiveness :- In interactive applications, multithreading enhances perceived performance. For e.g :- a background thread can handle computations while a foreground thread manages user input, preventing the application from freezing during long tasks.

(3.) Event-Driven Workflows :- When threads wait for independent external events (e.g. timers, signals), multithreading allows concurrent event handling without serial bottlenecks.

4.10 Which components are shared across threads in a multithreaded process

a. Register values (b.) Heap memory (c.) Global variables d. Stack memory

Dynamically allocated memory (e.g., malloc, new) is shared by all threads

accessible by all threads

Each thread has its own set of register values & own call stack storing local variables & function call frames.

4.11 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain → can be yes or no, depending on situation.

Yes, but only if threading model allows the OS to distribute user level threads across multiple CPUs (e.g.:- in many-to-many or one-to-one models).

No, if the threading model restricts all user level threads to a single CPU. (e.g., many-to-one).

4.12 Google chrome → new tab in separate process vs new tab in separate thread.

Designing chrome to open each new tab in a separate thread instead of separate process wouldnot achieve the same benefits

Separate process ensures isolation & stability as each process has its own memory space. If a tab (process) crashes, it doesnot affect other tabs or the main browser process. whereas, threads share the same memory space. A bug or crash in one thread could crash the entire browser, including all other tabs. True parallelism can be achieved by distributing processes across multiple CPU cores. While threads require synchronization (e.g. locks) which can lead to contention

Pg No. and overhead. Chrome uses sandboxing to restrict each tab's process from accessing sensitive OS resources. Hence chrome's use of process ensures security

Date:

4.13 Is it possible to have concurrency but not parallelism?

Yes. Concurrency refers to managing multiple tasks simultaneously (e.g.:- interleaving execution on a single core), while parallelism requires tasks to execute at the same time (e.g. on multiple cores).

Example:-

Concurrent, Not Parallel

Both task running on Core 1, both making progress one by one.

4.14 Amdahl's law :- $\frac{1}{(1-P)+\frac{P}{N}}$

40% parallel with a) 8 cores, b) 16 cores

a) Speedup = $\frac{1}{(1-0.4)+\frac{0.4}{8}} = 1.538$

b) Speedup = $\frac{1}{(1-0.4)+\frac{0.4}{16}} = 1.6$

67% parallel with a) 2 cores, b) 4 cores

a) Speedup = $\frac{1}{(1-0.67)+\frac{0.67}{2}} = 1.5037$

b) Speedup = $\frac{1}{(1-0.67)+\frac{0.67}{4}} = 2.01$

90% parallel with a) 4 cores, b) 8 cores

a) Speedup = $\frac{1}{(1-0.9)+\frac{0.9}{4}} = 3.0769$

b) Speedup = $\frac{1}{(1-0.9)+\frac{0.9}{8}} = 4.7058$

4.15 Task parallelism or Data parallelism.

- Using separate thread to generate a thumbnail for each photo in a collection. DATA PARALLELISM. Same task ("thumbnail generation") applied to different data (photos).
- Transposing a matrix. DATA PARALLELISM
Same operation (swapping) applied to different parts of matrix.
- Networked app (Read/write threads). TASK PARALLELISM
Different tasks (reading, writing).
- Fork-join array summation. DATA PARALLELISM
Same summation operation applied to subarrays.
- (GCD) Grand Central Dispatch. TASK PARALLELISM
GCD schedules independent tasks/blocks of work.

Date:

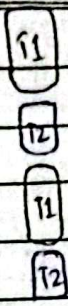
1.13 Is it possible to have concurrency but not parallelism?

Yes. Concurrency refers to managing multiple tasks simultaneously (e.g.:- interleaving execution on a single core), while parallelism requires tasks to execute at the same time (e.g. on multiple cores).

Example:-

Concurrent, Not Parallel

CPU Core 1



Both task running on Core 1, both making progress one by one.

4.14 Amdahl's law :- $\frac{1}{(1-P)+\frac{P}{N}}$

40% parallel with a) 8 cores, b) 16 cores

a) Speedup = $\frac{1}{(1-0.4)+\frac{0.4}{8}}$ = 1.538x

b) Speedup = $\frac{1}{(1-0.4)+\frac{0.4}{16}}$ = 1.6x

67% parallel with a) 2 cores, b) 4 cores

a) Speedup = $\frac{1}{(1-0.67)+\frac{0.67}{2}}$ = 1.5037x

b) Speedup = $\frac{1}{(1-0.67)+\frac{0.67}{4}}$ = 2.01x

90% parallel with a) 4 cores, b) 8 cores

a) Speedup = $\frac{1}{(1-0.9)+\frac{0.9}{4}}$ = 3.0769x

b) Speedup = $\frac{1}{(1-0.9)+\frac{0.9}{8}}$ = 4.7018x

4.15 Task parallelism or Data parallelism.

- Using separate thread to generate a thumbnail for each photo in a collection. DATA PARALLELISM. Same task ("thumbnail generation") applied to different data (photos).
- Transposing a matrix. DATA PARALLELISM
Same operation (swapping) applied to different parts of matrix.
- Networked app (Read/write threads). TASK PARALLELISM
Different tasks (reading, writing).
- Fork-join array summation. DATA PARALLELISM
Same summation operation applied to subarrays.
- (GCD) Grand Central Dispatch. TASK PARALLELISM
GCD schedules independent tasks/blocks of work. Although it can also exhibit data parallelism depending on usage of developer.

Pg No.

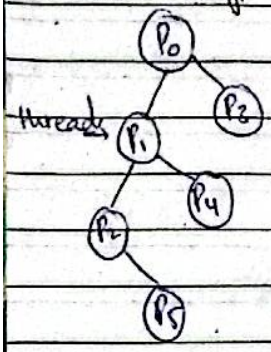
Date:

4.16 Shirt CPU bound terminate, one to one threading

- How many threads will you create to perform the input & output? Use 1 thread for input & 1 thread for output. since it is sequential so multiple threads would introduce contention without performance gains.
- How many threads would you use for CPU intensive portion. Create 4 threads (one per CPU core).
- The system has 4 cores, & the one to one threading model allows parallel execution.
- Maximizes CPU utilization for the CPU-bound workload.

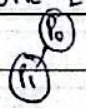
4.17 a. How many unique processes are created?

b. How many unique threads are created?

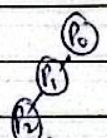


a. Total processes : 6

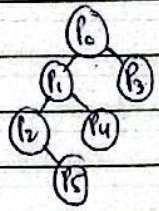
Fork 1 :- creates P_1



Fork 2 : creates P_2



Fork 3 : creates P_3, P_4, P_5



b. Total threads = 2 + 1 + 1 = main thread

thread_create is called once in P1.

4.18 Linux vs. Windows Process / Thread Modeling

Linux treats process, thread same way, no distinguish

Windows treat separately distinguishes processes & threads.

- Linux treats tasks as flexible entities (process-like or thread-like) via clone().
- Windows enforces a strict hierarchy (processes own threads).
- Linux offers more flexibility but less isolation between threads.
- Windows provide stronger isolation but less granularity in resource sharing.

Date:

4.19

CHILD : value = 5 // line C

PARENT : value = 0 // line P

EX-9

4.20

a. If the number of kernel threads allocated to the program is less than the number of processing cores, then not all processing cores will be utilized. This means that the program will not be able to fully take advantage of the available processing power & its performance maybe limited.

b. If the number of kernel threads allocated to the program is equal to the number of processing cores, then all processing cores can be utilized. This allows the program to fully take advantage of the available processing power & can result in optimized performance.

c. If the number of kernel threads allocated to the program is greater than the number of processing cores but less than number of user-level threads, then all processing cores can be utilized & some user-level threads will have to share kernel threads. This can result in improved performance compared to scenario a, but maynot be as efficient as scenario b due to overhead of context switching between user-level threads sharing a kernel thread.