23k-2001

OS Hometask: Chp04

Date

4.1:

Multithreading Programming Examples

i. An interactive GUI program such as debugger where on thread represent running application, one to monitor user input, and one to monitor performance.

ii. Parallel application such as matrix multiplicator where various steps of the matrix can be worked in parallel.

iii. A web-server that responds to each request in separate threads.

4.2:

Amdahl's law: $\text{Speedup} \leq \frac{1}{S + \frac{1-S}{N}}$

$1 - S = 0.6 \Rightarrow S = 0.4$

a. $N = 2$ (processing cores)

$$\text{speedup} = \frac{1}{0.4 + \frac{0.6}{2}}$$

$\text{speedup} \simeq 1.42$ Ans.

b. $N = 4$ (processing cores)

$$\text{speedup} = \frac{1}{0.4 + \frac{0.6}{4}}$$

$\text{speedup} \simeq 1.82$ Ans.

Date

4.3:

The multithreaded web-server exhibits 'data parallelism', as each thread performs the same task, but on different data. Ans.

4.4:

User-level Threads	Kernel-level Threads
→ Managed by a user-level library, i.e the creation, scheduling & synchronization of the thread.	→ Managed by the OS kernel, i.e the creation, scheduling & synchronization of the thread.
→ Context switching is faster as it does not require a switch to kernel mode.	→ Context switching is slower and more expensive to maintain than user threads
→ Better for providing more control over scheduling.	→ Better for providing better overall system performance.

4.5: Kernel-threads Context Switching

i. Save state: Kernel saves the current state of the thread.
ii. Update TCB: Kernel updates thread control block to reflect new state.
iii. Select next thread: Kernel uses its scheduling algorithm to choose the next thread to run from ready threads.
iv. Load state: kernel loads the saved state of the new thread that has been selected; including program counter & registers.
v. Update CPU: Kernel updates the CPU with the new context & sets it to begin execution.

Date

4.6:

Resource allocation for a thread:

i. Thread Control Block: a structure that stores information about the thread, such as its state, priority and scheduling information.

ii. Stack: each thread has its own stack to store local variables and function call information.

iii. Registers: each thread has its own set of registers.

4.7:

Binding real-time thread to an LWP:

It is necessary to bind a real-time thread to a light-Weight Process, as timing is crucial for real-time applications. Without LWP, the thread may have to wait to be attached to an LWP before running. Consider:

→ Real-time thread is running and is then blocked (either for I/O, or higher priority etc).

→ While this thread is blocked, the LWP it was attached to is assigned to another thread.

→ When the real-time thread has been scheduled to run again, it must wait to be attached to an LWP.

⇒ By binding an LWP to a real time thread we can ensure, that the thread will be able to run with minimal delay once it is rescheduled.

Date

4.8:

Examples where Multi-Threading Is not Better:

i. High contention for shared resources:

If multiple threads need to access a shared resource frequently and there is high contention for that resource, then using multiple threads will not provide any performance benefits. The threads would spend time waiting to acquire the resource, reducing the amount of useful work they perform

ii. Computation bounded tasks:

If a task is computationally bounded and does not involve any I/O operations, then using multiple threads may not provide any performance benefits. This is because if the CPU is already fully utilized by a thread, adding more threads would not increase the amount of work that can be performed concurrently.

4.10:

Components of program state shared across threads:

→ Heap memory & Global variables are shared across threads.

Ans.

Date

4.11:

Multiple User-level threads on a multiprocessor system:

No, a multithreaded solution using multiple user-level threads cannot achieve better performance on a multiprocessor system, compared to a single-processor system.

→ User level threads are not directly managed by OS kernel scheduler.

→ The OS does not recognize these threads individually, the scheduler only sees it as a single process.

→ Since user-level threads are not individually assigned to different processors, the OS schedules the entire process as one unit on a single core.

→ All the user-level threads within a process share the same core, leading to no true parallelism.

Hence, user-level threads do not gain any performance boost from multiple processors, and remain confined to a single core, wasting the potential of other available processors.

Date

4.12:

Each new tab in separate thread rather than separate process?
No, the same benefits would not have been achieved.
→ Each new tab as a separate process provides better isolation between tabs.
→ If one tab crashes, it does not affect the other tabs, as processes have their own memory space and resources.
→ If each tab opened in a separate thread within the same process, then if one tab crashed, it could affect the other threads as threads within a process share memory and other resources.
Hence, for better stability and security handling each tab as a separate process is better rather than a separate thread.

4.13:

Concurrency without parallelism:
Yes, concurrency can be achieved without parallelism by using time-slicing, that gives illusion that tasks are being executed simultaneously. However, there is no parallelism because only one task is being executed at a time due to rapid switching between tasks.

Date

4.14:

Amdahl's Law: $\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$

a. $1 - S = 0.4$

i. $N = 8$

$\Rightarrow S = 0.6$

$\text{speedup} = \frac{1}{0.6 + \frac{0.4}{8}}$

$\text{speedup} \approx 1.53$ Ans.

ii. $N = 16$

$\Rightarrow S = 0.6$

$\text{speedup} = \frac{1}{0.6 + \frac{0.4}{16}}$

$\text{speedup} \approx 1.6$ Ans.

b. $1 - S = 67\%$

i. $N = 2$

$\Rightarrow S = 33\%$

$\text{speedup} = \frac{1}{0.33 + \frac{0.67}{2}}$

$\text{speedup} \approx 1.503$ Ans.

ii. $N = 4$

$\Rightarrow S = 33\%$

$\text{speedup} = \frac{1}{0.33 + \frac{0.67}{4}}$

$\text{speedup} \approx 2.01$ Ans.

c. $1 - S = 90\%$

i. $N = 4$

$\Rightarrow S = 10\%$

$\text{speedup} = \frac{1}{0.1 + \frac{0.9}{4}}$

$\text{speedup} \approx 3.07$ Ans.

ii. $N = 8$

$\Rightarrow S = 10\%$

$\text{speedup} = \frac{1}{0.1 + \frac{0.9}{8}}$

$\text{speedup} \approx 4.705$ Ans.

Date

4.15: Parallelism Type

i. Data parallelism because each thread performs same task on different data.

ii. Data parallelism because same operation is performed on different parts of matrix

iii. Task parallelism because threads complete different tasks, one reads & other writes.

iv. Data parallelism because same operation is performed on different data.

v. Both: data & task parallelism
→ the system can distribute tasks to different cores (UI updates, network request, etc): Task parallelism
→ the system can split the data across threads performing same operation on different parts of the data: Data Parallelism.

4.16: Multithreading Solution

a. For input and output, one thread each because I/O is performed on a single file at startup and termination. Increasing threads for these operations will not increase performance.

b. For CPU-intensive application, four threads → one for each processor. This allows the application to fully utilize the processing power of the system and increases performance.

RC

Date

4.17:

a. 5 unique process created.

b. 2 unique threads created. 1 by the original child, and 1 by the grandchild created inside the if-block of the original child.

4.18:

Linux	Windows
• Uses single task_struct for both processes & threads.	• Uses process control block (PCB) with multiple thread control blocks (TCB).
• Uses clone() system call for thread creation.	• Uses CreateThread() inside a process to create a thread.
• Uses fork() for creating process.	• Uses CreateProcess() for process creation.
• Resources shared are controlled through clone() flags.	• Threads share resources within a process, but processes don't share resources among each.
• No distinction between threads & processes for scheduling	• Threads are scheduled separately from processes.
• Efficient thread creation but with higher context-switch overhead.	• Faster thread context switch but inter-process communication is slower and complex.

23k-2001 Date

4.19:
Program Output

Line C:

CHILD: value = 5

Line P:

PARENT: value = 0