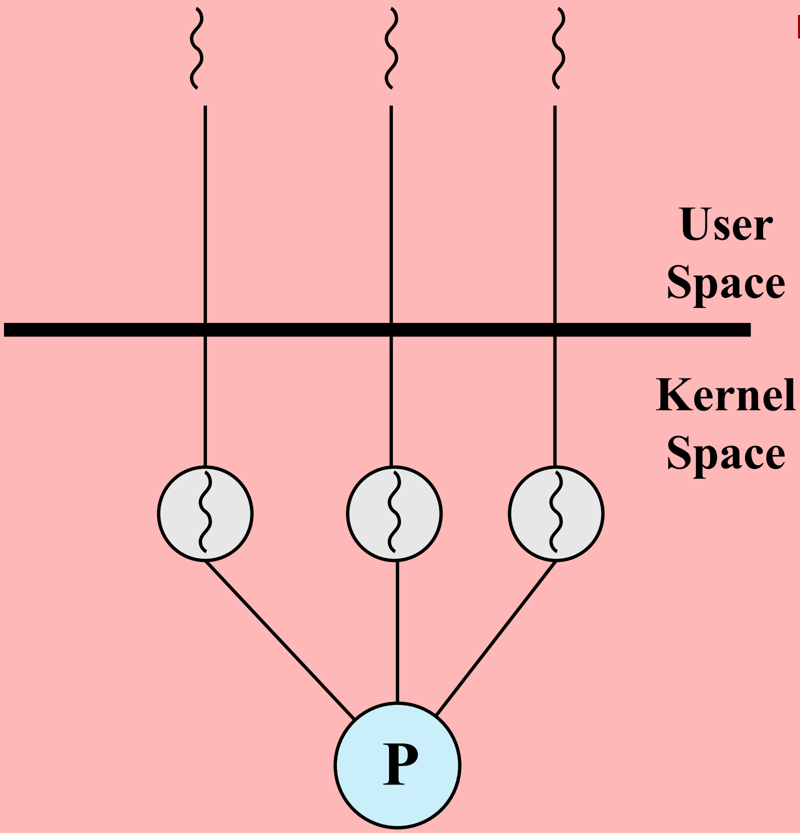
Chapter 4
Threads

Part **B**: Sections 4.**2**–4.**4**

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- **Thread management** is done by the kernel
  - There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
  - Windows is an example of this approach

Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- **Kernel routines themselves can be multithreaded, too,** ***for better OS-Performance***

Disadvantage of KLTs

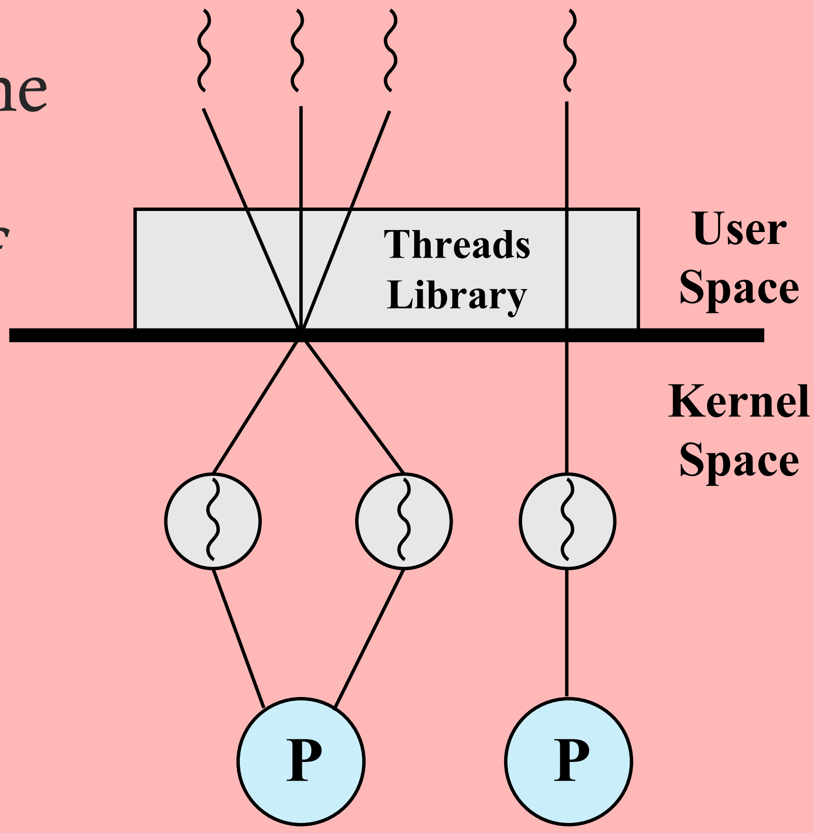
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1
Thread and Process Operation Latencies (μs)

Combined Approaches

- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application
- **Example:** ***Solaris***



(c) Combined

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Table 4.2
Relationship between Threads and Processes

Amdahl's Law !

- A formula with which we can ***roughly estimate*** **the speed-up** of a program (or ***process***) when it is distributed over **several CPUs** in a concurrent mode of execution

- The formula is *"idealized"* because it **does NOT take into account the time-cost for thread-management** (scheduling, etc.) which the OS must invest to organize concurrency or parallelism. (*These time-costs can be quite large!*)

- For a program or process **P** the formula stipulates:

$$s = \frac{1}{(1-f) + \dfrac{f}{N}}$$

whereby:

  - **1** ≤ **s** ≤ **N** is the (*roughly estimated*) **speed-up** factor,
  - **1** ≤ **N** ≤ TLOC is the **number of available CPUs** over which **P** may be distributed. *Note: more CPUs than Total Lines of Code would be useless, because the superfluous CPUs would stay idle!*
  - **0** ≤ **f** ≤ **1** is the (*estimated*) **fraction** (*percentage %*) of **P**'s total lines of **code which is amenable to parallelization** (concurrency), *under the further assumption that there are no mutual-waiting-dependencies in this fraction. For accurate estimations the TLOC ought to be counted in atomic machine-code-instructions, not in "high level" software language.*

Estimation of *f and s* : a small example

- **Consider this following "snippet"**

  of three lines of code (L1–L3):

```
L1        x := x + 1;
L2        y := y + 5;
L3        z := x + y;
```

- The contents of **L1** and **L2** have nothing to do with each other: they are entirely independent of each other, and could thus be carried out separately and simultaneously by two different CPUs.

- **L3**, however, needs the results from L1 and L2 in order to be able to produce a value for **z**: hence, the computation of **L3** *cannot happen simultaneously* with the computation of lines L1 and L2. **L3** must be computed in a "sequential" mode and is thus **"not amenable" for parallelisation** in this small example.

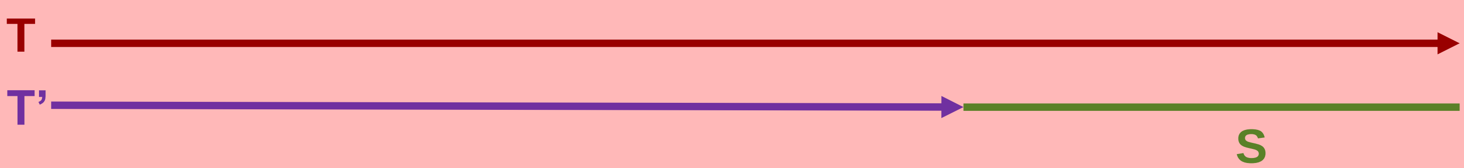
- With only two out of these three lines being "amenable", we get **f = 2/3 ≈ 66%**

- For these only 3 lines of code, **more than 3 CPUs would be useless**. In this example, **even N=3 would be useless, too**, because **L3** is "not amenable": **L3** *could possibly be done by the same CPU that had already computed* L1! Thus, **2 CPUs will suffice** (N=2).

- Hence, **in this tiny example**: **s** ≈ 1 / ((1 – **0.66**) + (**0.66** / 2)) ≈ **1.49**

Speed-up *versus* Time saved

- Let **P** be a program (process) and **T** its **total run-time** which it would need on **1 CPU**.
- Let **s** be the **speed-up** value calculated for **P** by Amdahl's Law for **N** CPUs.
- **P**'s **shortened** run-time **T'** is thus (**T / s**), whereby **T' < T**, if **s > 1**.
- The time **saved** is thus **S** := **T – T'** = **T – (T / s)**



- In our small *example from the previous slide*, with **s = 1.49**, the time saved would thus be T – (T / **1.49**)
- Assume, *for example*, that T=**3** (time-units for the *three lines of code*), then the time **saved** *in this example* would be **S** = **3–(3/1.49)** ≈ **0.98** time-units

Amdahl's Law **must be applied "wisely"**!

- **Consider this following "snippet" of three lines of code (L1–L3):**

```
L1        x := x + 1;
L2        y := y + 5;
L3        z := x + y;
```

- From the discussion of the previous slide we can learn that Amdahl's Law **must be applied *"wisely"*** in order to yield **meaningful and near-realistic speed-up estimations**!

- The *formula* of Amdahl's Law can be easily mis-applied because it *does not show the "hidden relation"* between the value **f** and the number of **N** CPUs which is "most appropriate" for this specific **f** !

- For example, we might *naively* guess: "*we have three lines of code here, thus let us set* **N=3** and use 3 CPUs".

- Then we would *naively* calculate: **s** ≈ 1 / ((1–0.66)+(0.66/**3**)) ≈ **1.79** , and *we would wrongly rejoice about this apparently "good" speed-up*, though in reality we cannot hope for more in this example than the **s=1.49** which we had derived on the previous slide. This is because *the 3rd CPU in this example* ***is idle*** *while the first two CPUs are still busy with the processing of lines L1 and L2!*

Moreover:

- *In some cases it is not even possible to properly estimate the fraction-value f* **which is needed for calculating the approximate speed-up value s by means of Amdahl's Law!**
  - In such cases, where theoretical analysis is not feasible, an experimental approach (with many repeated run-time experiments) is the only way of determining the actual speed-up value **s**.
- **The considerations on the following slides will illustrate this dilemma.**

Illustration of the Difficulties of "working with" Amdahl's Law

```
Proc_def Difficult() // sequential version without concurrency
{
 int r := 0;
 int c := 0;

 for (i=0, i<1000, i++) do // loop with thousand repetitions
 {
  if (r==0) then { r := call Random_generator(); } // make any positive number

  if (r > 0) then { c := call Collatz(r); // c counts how often Collatz() was looping on r
                    r := 0; }

  if (c > 0) then { if ((c mod2)==1) then { print("odd") } else { print("even") }
                    c := 0; }
 }
}
```

Attention: This example is NOT in the textbook! Please familiarize yourself with the Collatz algorithm.

Illustration of the Difficulties of "working with" Amdahl's Law

```
Proc_def Difficult() // concurrent versions with threads: each running 1000 loops
{
 int r := 0;
 int c := 0;

Thread_def A() {
 for (i=0, i<1000, i++) do // thousand times
 { await (r==0) then { r := call Random_generator(); }}

Thread_def B() {
 for (i=0, i<1000, i++) do // thousand times
 { await (r > 0) then { c := call Collatz(r);
                         r := 0; } } }
Thread_def C() {
 for (i=0, i<1000, i++) do // thousand times
 { await (c > 0) then { if ((c mod2)==1) then { print("odd") } else { print("even") }
                         c := 0; } } }
}
```

Thread **A** *depends on* Thread **B** via the shared Process variable **r**

Thread **C** *depends on* Thread **B** via the shared Process variable **c**

Attention: This example is NOT in the textbook!

Illustration of the Difficulties of "working with" Amdahl's Law

```
Proc_def Difficult() // concurrent versions with threads: each running 1000 loops
{
 int r := 0;
 int c := 0;
```

"constant" code section, the "size" of which can be easily estimated for Amdahl's formula.

```
 Thread_def A() {
  for (i=0, i<1000, i++) do
  { await (r==0) then { r := call Random_generator(); } }
```

"constant" code section, the "size" of which can be easily estimated for Amdahl's formula.

```
 Thread_def B() {
  for (i=0, i<1000, i++) do
  { await (r > 0) then { c := call Collatz(r);
                         r := 0; } } }
 Thread_def C() {
  for (i=0, i<1000, i++) do
  { await (c > 0) then { if ((c mod2)==1) then { print("odd") } else { print("even") }
                         c := 0; } } }
}
```

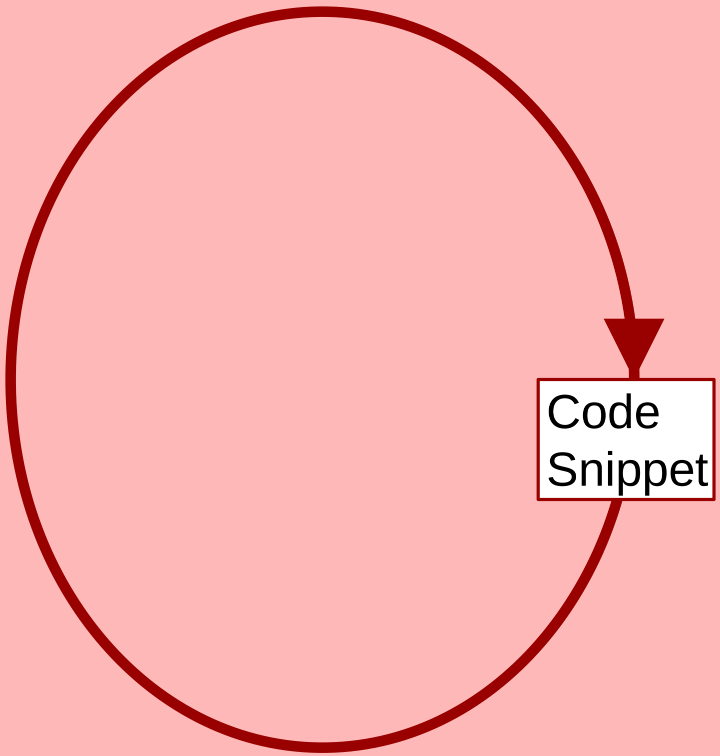
The run-time "size" of this code section *cannot* be estimated, because **for some inputs r the *Collatz* Algorithm returns a result c very quickly**, whereas **for some inputs r the *Collatz* algorithm takes very long time before it yields the output c** for which **Thread C** is dependently waiting (and so the **r:=0** for which **A** waits).

"constant" code section, the "size" of which can be easily estimated for Amdahl's formula.

Attention: This example is NOT in the textbook!

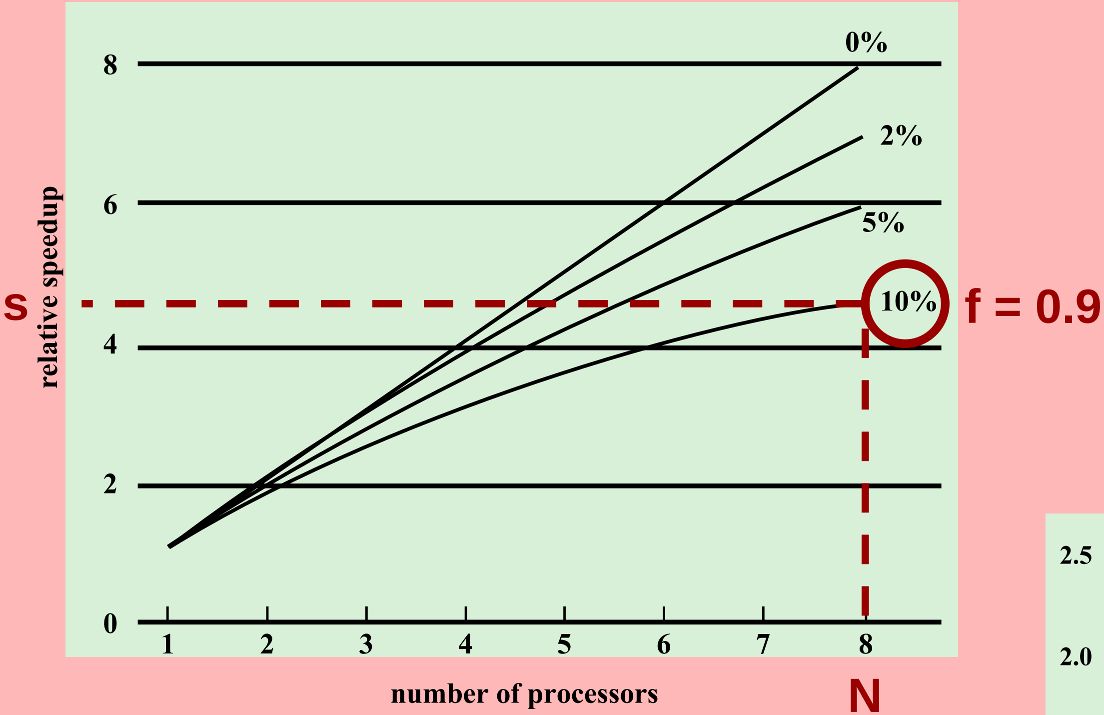
Hence we cannot estimate the Speed-Up for Difficult() with Amdahl's Law *even if we know the number of CPUs*

Thus we have learned:



If some code snippet appears inside a Loop, **its actual "size" depends on how often the Loop is actually repeated!** In many cases, however, this number of repetitions cannot be known beforehand

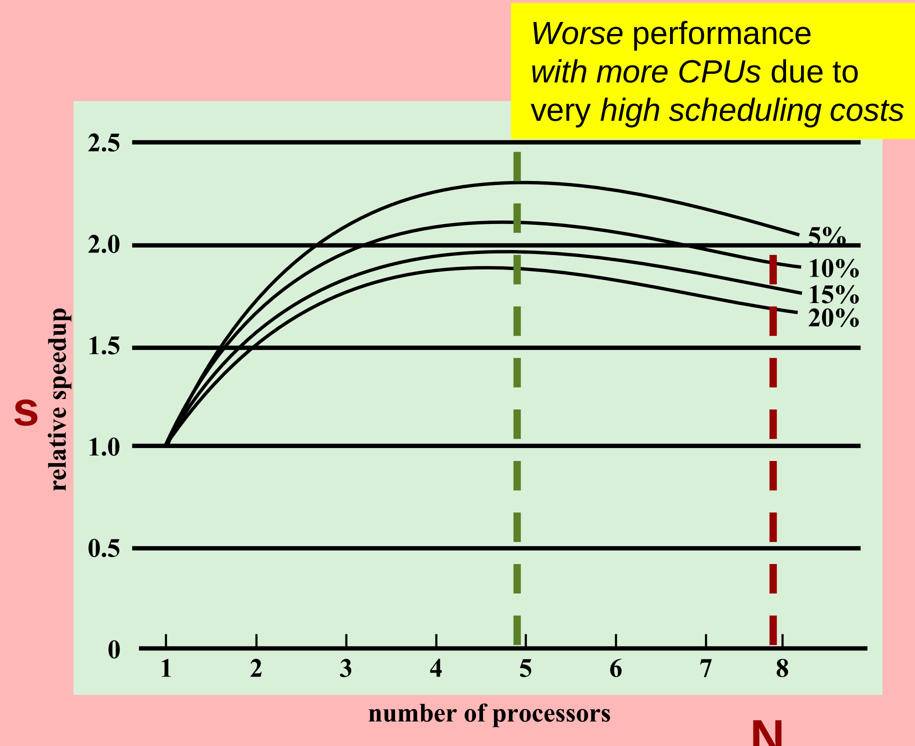
This makes it in some cases very difficult to estimate the "size" of some piece of Loop-Code, which *would* –however– be needed in order to give a reliable value to **f** in Amdahl's Formula



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

My new Laptop has 8 CPUs ☹

Hehehe, my old Laptop has only 4 CPUs ☺



(b) Speedup with overheads

Figure 4.7 Performance Effect of Multiple Cores

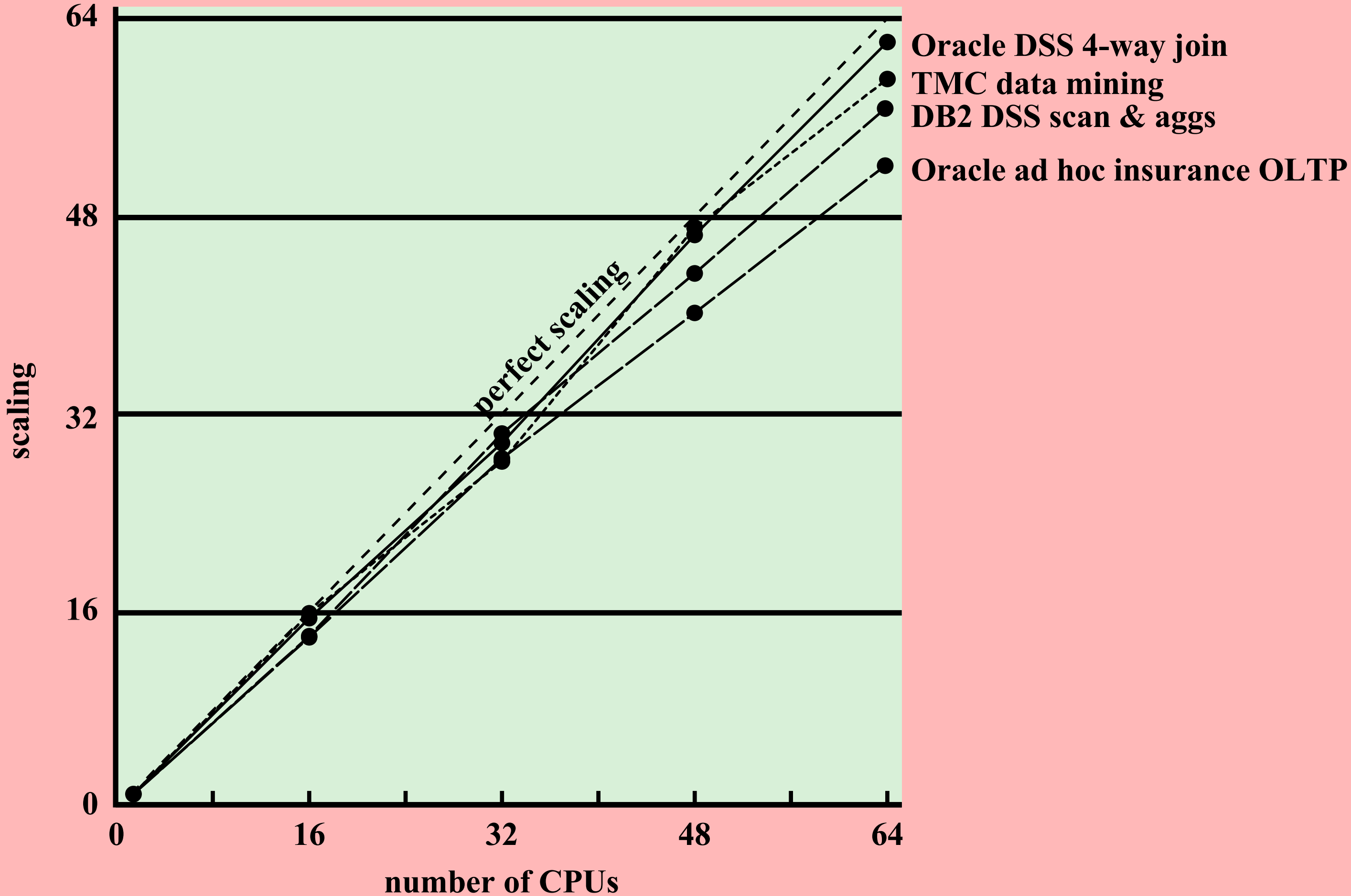


Figure 4.8 Scaling of Database Workloads on Multiple-Processor Hardware

Windows Process and *Thread* Management

- An **application** consists of one or more processes
- Each **process** provides the resources needed to execute a program
- A **thread** is the entity within a process that can be scheduled for execution
- A **job object** allows groups of process to be managed as a unit
- A **thread pool** is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application
- A **fiber** is a unit of execution that must be manually scheduled by the application
- **User-mode scheduling (UMS)** is a lightweight mechanism that applications can use to schedule their own threads

Process and Thread *Objects*

Windows uses two types of process-related **objects**:

Processes	Threads
• An entity corresponding to a user job or application that owns resources	• A dispatchable unit of work that executes sequentially and is interruptible

➔ A deeper understanding of "object-oriented" software will be taught in other courses of the Computer Science curriculum. **For now it suffices to regard an "object" as a "packaged combination" of suitably related Algorithms and Data Structures.**

A *simple* understanding of “**object**”

“Algorithms and their Data Structures in one Package”

- **Status :** *Ready*
- **Priority:** *High*
-
-

Algorithm:
Change_Status

Algorithm:
Change_Priority

Table 4.4 Windows Thread Object's Attributes

Thread ID	A unique value that identifies a thread when it calls a server.
Thread context	The set of register values and other volatile data that defines the execution state of a thread.
Dynamic priority	The thread's execution priority at any given moment.
Base priority	The lower limit of the thread's dynamic priority.
Thread processor affinity	The set of processors on which the thread can run, which is a subset or all of the processor affinity of the thread's process.
Thread execution time	The cumulative amount of time a thread has executed in user mode and in kernel mode.
Alert status	A flag that indicates whether a waiting thread may execute an asynchronous procedure call.
Suspension count	The number of times the thread's execution has been suspended without being resumed.
Impersonation token	A temporary access token allowing a thread to perform operations on behalf of another process (used by subsystems).
Termination port	An interprocess communication channel to which the process manager sends a message when the thread terminates (used by subsystems).
Thread exit status	The reason for a thread's termination.

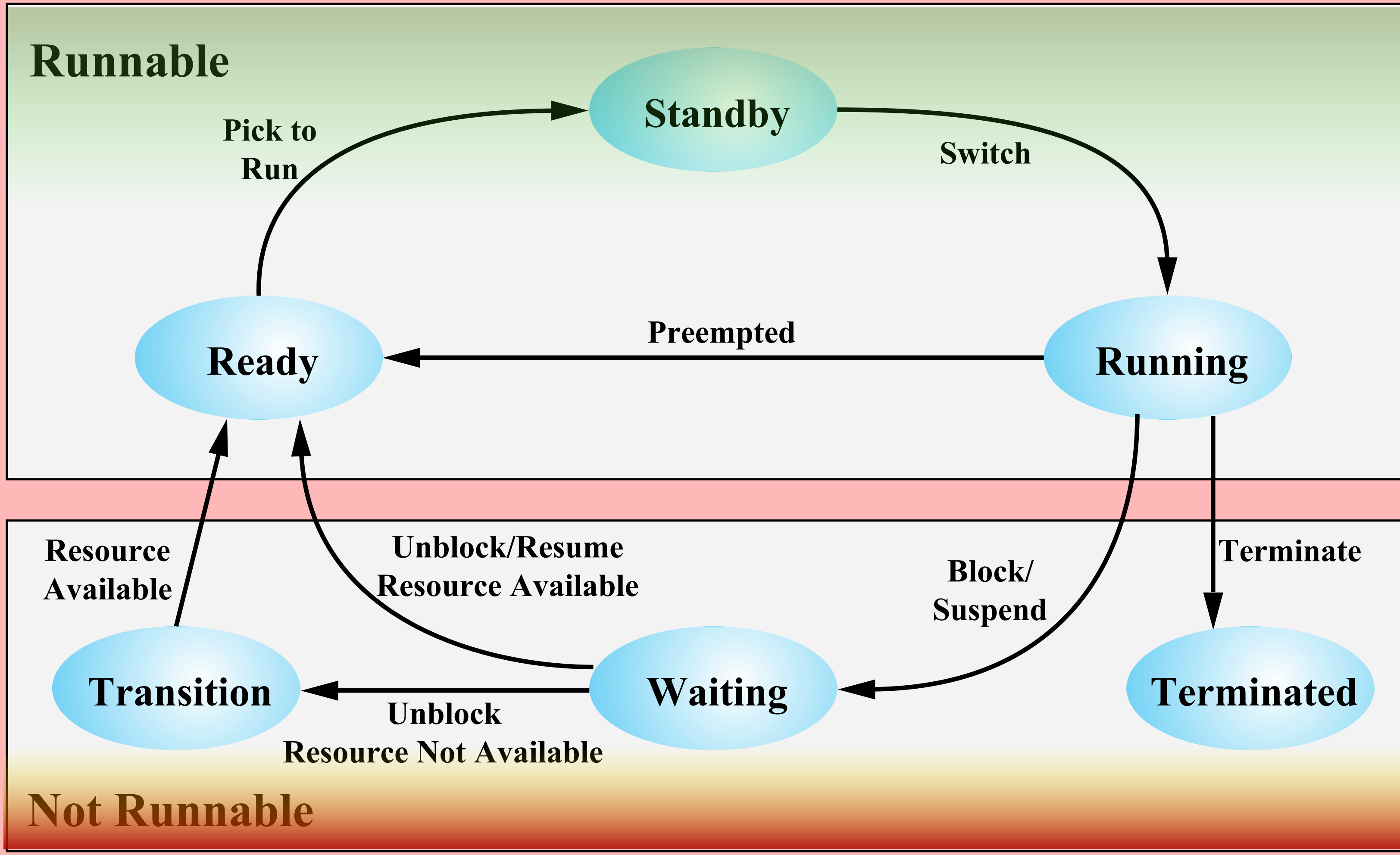


Figure 4.11 Windows Thread States

Multithreading

Achieves concurrency without the overhead of using multiple processes

Threads within the same process can exchange information through their common address space and have access to the shared resources of the process

Threads in different processes can exchange information through shared memory that has been set up between the two processes