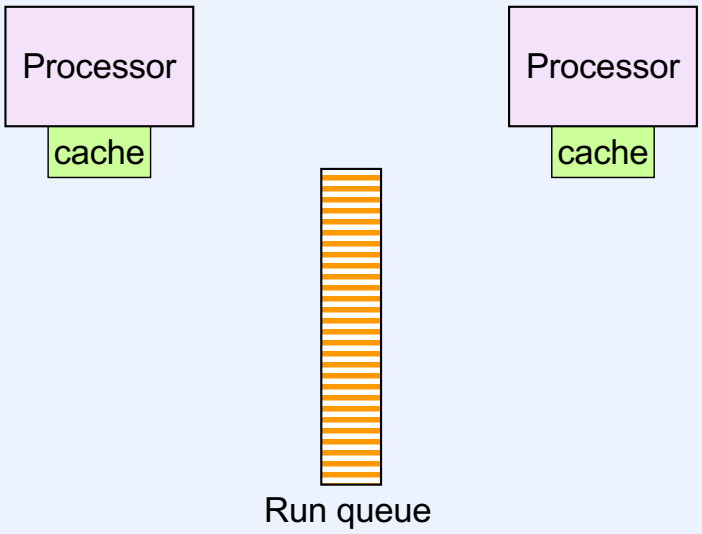
Scheduling Part 3

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Diagram



Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Old Scheduler: Problems

- O(n) execution
- Poor interactive performance with heavy loads
- SMP contention for run-queue lock
- SMP affinity
  - cache "footprint"

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

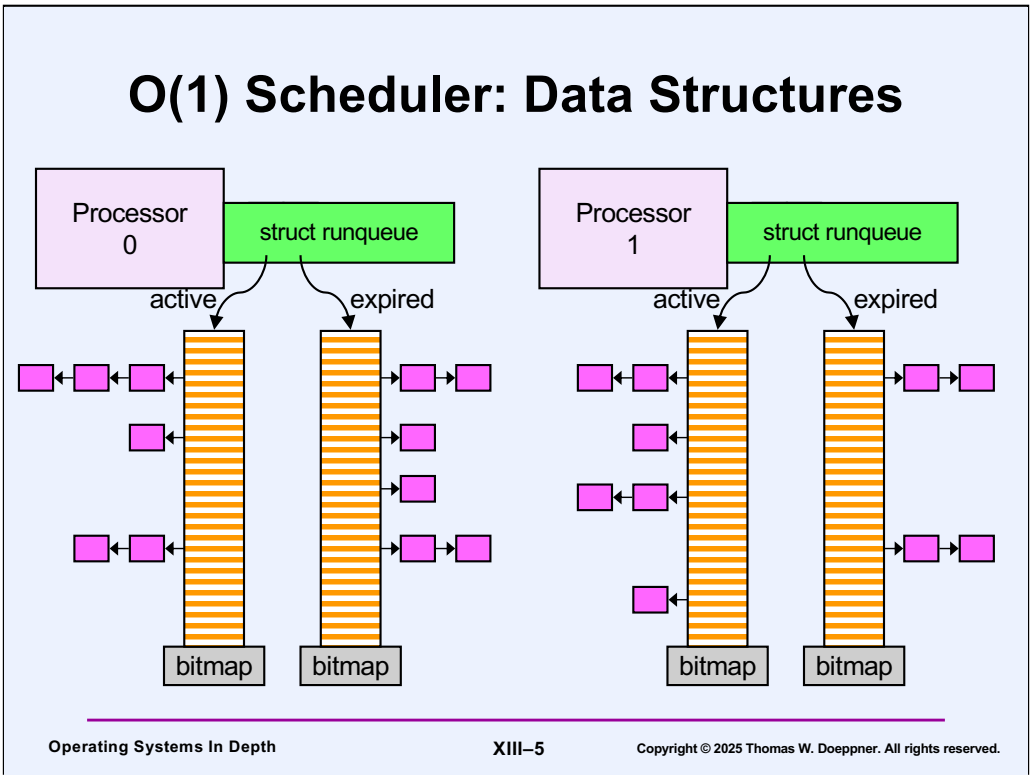
The main problems with the old scheduler are summarized in the slide. As we've seen, the scheduler must periodically examine all processes, as well as examining all runnable processes for each scheduling decision. When a system is running with a heavy load, interactive processes will get a much smaller percentage of processor cycles than they do under a lighter load. Since there's one run queue feeding all processors, all must contend for its lock. Finally, though some attempt is made at dealing with processor-affinity issues, processes still tend to move around among available processors (and thus losing any advantage of their "cache footprint").

O(1) Scheduler

- All concerns of old scheduler plus:
  - efficient, scalable execution
  - identify and favor interactive processes
  - good SMP performance
    - minimal lock overhead
    - processor affinity

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

The O(1) scheduler deals with all the concerns dealt with by the old one along with some additional concerns, as shown in the slide.



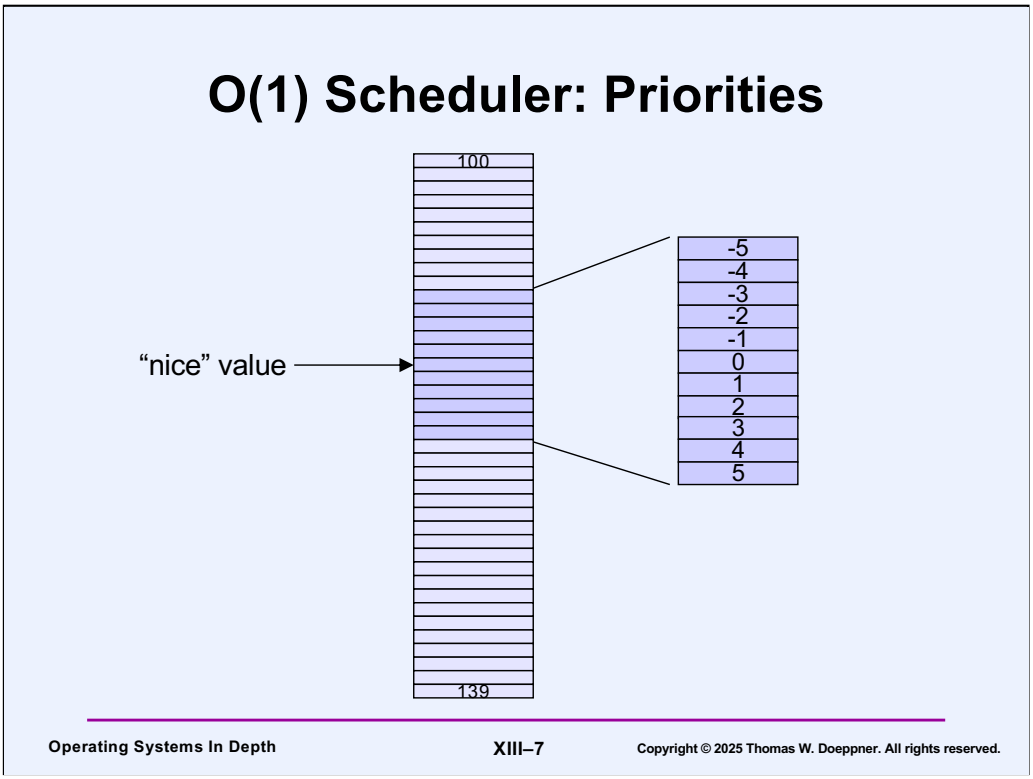
The O(1) scheduler associates a pair of queues with each processor via a per-processor **struct runqueue**. Each queue is actually an array of lists of processes, one list for each possible priority value. There are 140 priorities, running from 0 to 139; "good" priorities have low numbers; "bad" priorities have high numbers. Real-time priorities run from 0 to 99; normal priorities run from 100 to 139. Associated with each queue is a 140-element bit map indicating which priority values have a non-empty list of processes.

O(1) Scheduler: Queues

- Two queues per processor
  - active: processes with remaining time slice
  - expired: processes whose time slices expired
  - each queue is an array of lists of processes of the same priority
    - bitmap indicates which priorities have processes
  - processors scheduled from private queues
    - infrequent lock contention
    - good affinity

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

When processes become runnable they are assigned time slices (based on their priority) and put on some processor's **active** queue. When a processor needs a process to run, it chooses the highest priority process on its active queue (this can be done quickly (and in time bounded by a constant) by scanning the queue's bit vector to find the first non-empty priority level, then selecting the first process from the list at that priority). When a process completes its time slice, it goes back to the expired queue of its processor (as explained shortly). If it blocks for some reason and later wakes up, it will generally go back to the active queue of the processor it last was on. Thus, processes tend to stay on the same processor (providing good use of the cache footprint). Since processors rarely access other processors' queues, there is very little lock contention.



The values over which a process's priority may range are determined by its "nice" value, settable by a system call, and are within a range of +5 and -5. The default is a nice value of 0, in which case the process's priority ranges from 115 through 125. Within the range, the priority is determined by how much time the process has been sleeping in the recent past. In no case will a non-real-time process's priority be less than 100 or greater than 139.

O(1) Scheduler: Actions

- **Process switch**
  - **pick best priority from active queue**
    - **if empty, switch active and expired**
  - **new process's time slice is function of its priority**
- **Wake up**
  - **priority is boosted or dropped depending on sleep time**
  - **interactive processes are defined as those whose priority is above a certain threshold**
- **Time-slice expiration**
  - **normal processes join expired queue**
  - **real-time join active queue**

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

When a process completes its time slice, it is inserted into its processor's expired queue, unless it's a real-time process, in which case it rejoins the active queue. The intent is that real-time processes get another time slice on the processor, while other processes have to wait a bit on the expired queue. When there are no processes remaining in the active queue, the two queues are switched. Of course, if there are interactive processes that often block, then resume execution (and thus their time slices never expire), the active queue might never empty out. So, if the processes in the expired queue have been waiting too long (how long this is depends on the number of runnable processes on the queue), interactive processes go to the expired queue rather than the active queue. Runnable real-time processes never go to the expired queue: they are always in the active queue (and always have priority over non-real-time processes). Thus two queues are employed as a means to guarantee that, in the absence of real-time processes, all processes get some processor time. Lower priority processes will remain on the active queue until all higher-priority processes have moved to the expired queue.

The net effect is similar to the old scheduler: in the absence of real-time processes, processes get the processor in proportion to their priority. However, interactive processes (those that have recently woken up) get extra time slices.

O(1) Scheduler: Load Balancing

- **Processors with empty queues steal from busiest processor**
  - **checked every millisecond**
- **Processors with relatively small queues also steal from busiest processor**
  - **checked every 250 milliseconds**

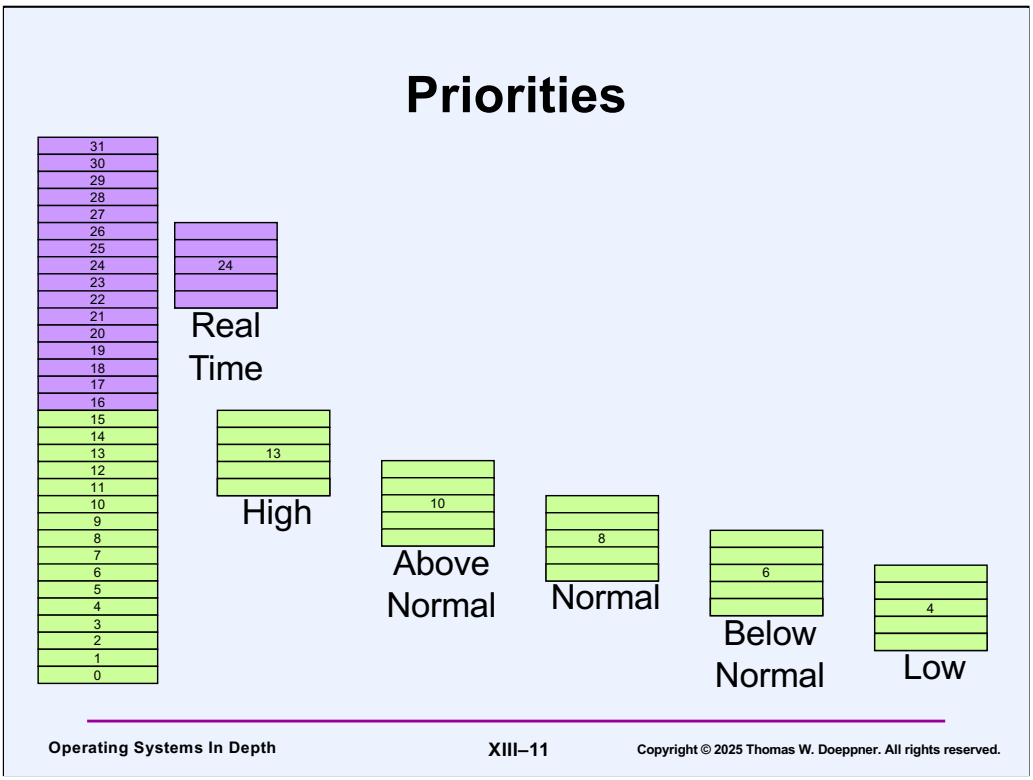
Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Since processors schedule strictly from their own private queues, load balancing is an issue (it wasn't with the old scheduler, since there was only one global queue serving all processors). Each processor checks its queues for emptiness every millisecond. If empty, it calls a load balancing routine to find the processor with the largest queues and then transfers processes from that processor to the idle one until they are no longer imbalanced (they are considered balanced if they are no more the 25% different in size). Similarly, each processor checks the other processors' queues every 250 milliseconds. If an imbalance is found (and it's not just a momentary imbalance but has been that way since the last time the processor's queues were examined), then load balancing is done.

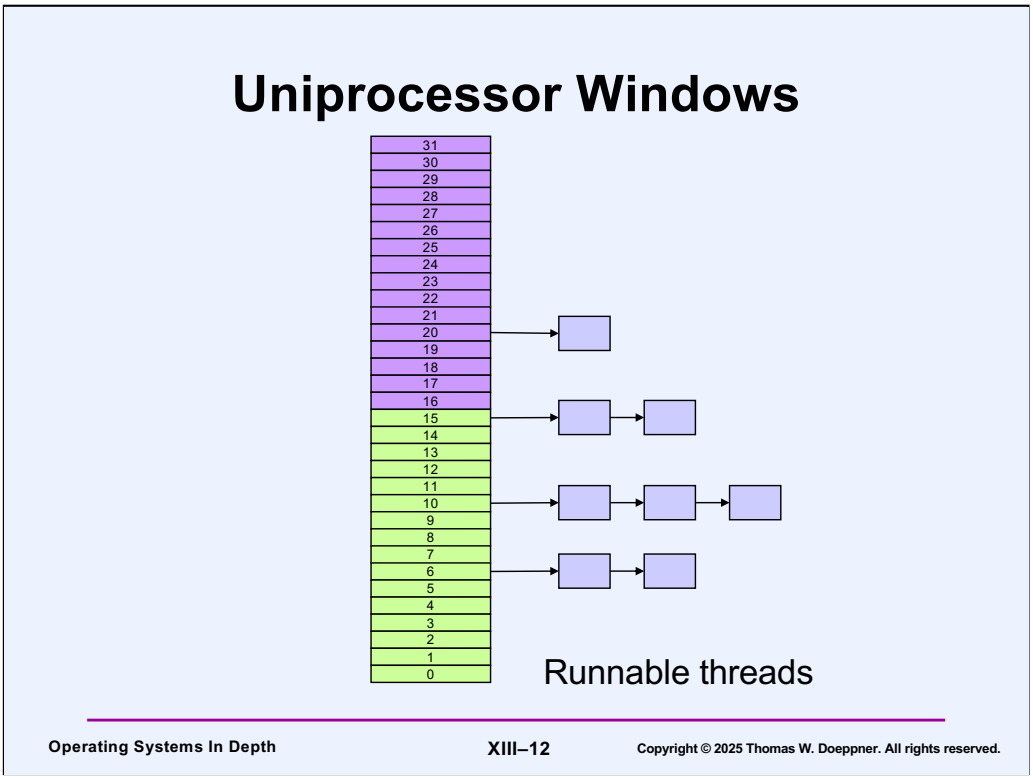
Scheduling in Windows

- Handling “normal” interactive and compute-bound threads
- Real-time threads
- Multiple processors

Copyright © 2025 Thomas W. Doeppner. All rights reserved.



Threads are assigned **base priorities**, chosen from the six ranges shown in the slide (usually the middle of a range). Their **current priority** is some value equal to or greater than their base (always equal to the base for real-time threads). When waking up from a sleep, a thread's current priority is set to its base priority plus some wait-specific value (usually in the range 1 to 6, depending on what sort of event it was waiting for). A thread's current priority is decremented by one, but to no less than the base, each time a quantum expires on it. Threads of the current window get three times the time quantum of other threads, thus giving them a greater portion of available processor time.



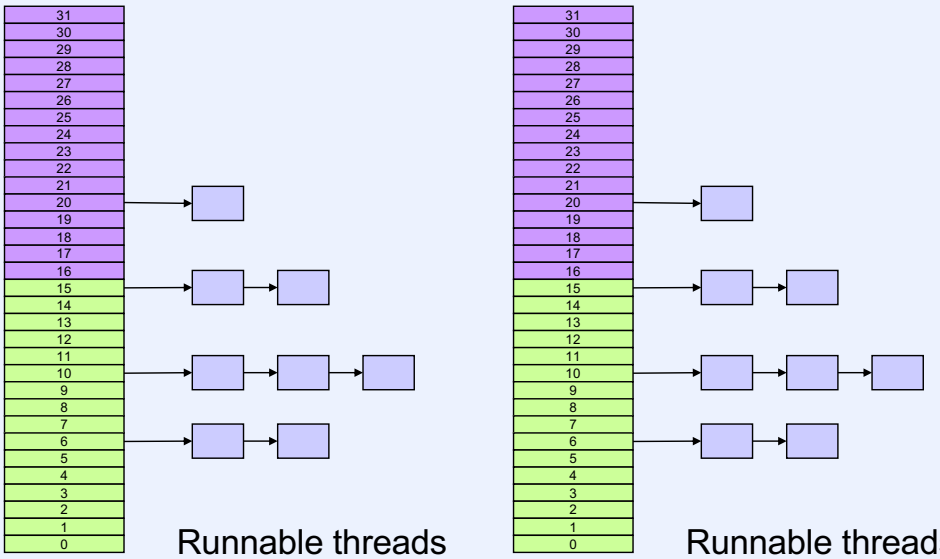
Windows employs an array of dispatch queues, one for each priority level.

Improving Real Time

- Multimedia applications need 80% of processor time
- Make sure normal applications get at least 20%
- How?
- Windows solution: MMCSS
  - multimedia class scheduler service
  - dynamically manage multimedia threads
    - run at real-time priority 80% of time
    - run at normal priority 20% of time

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Multiprocessor Windows



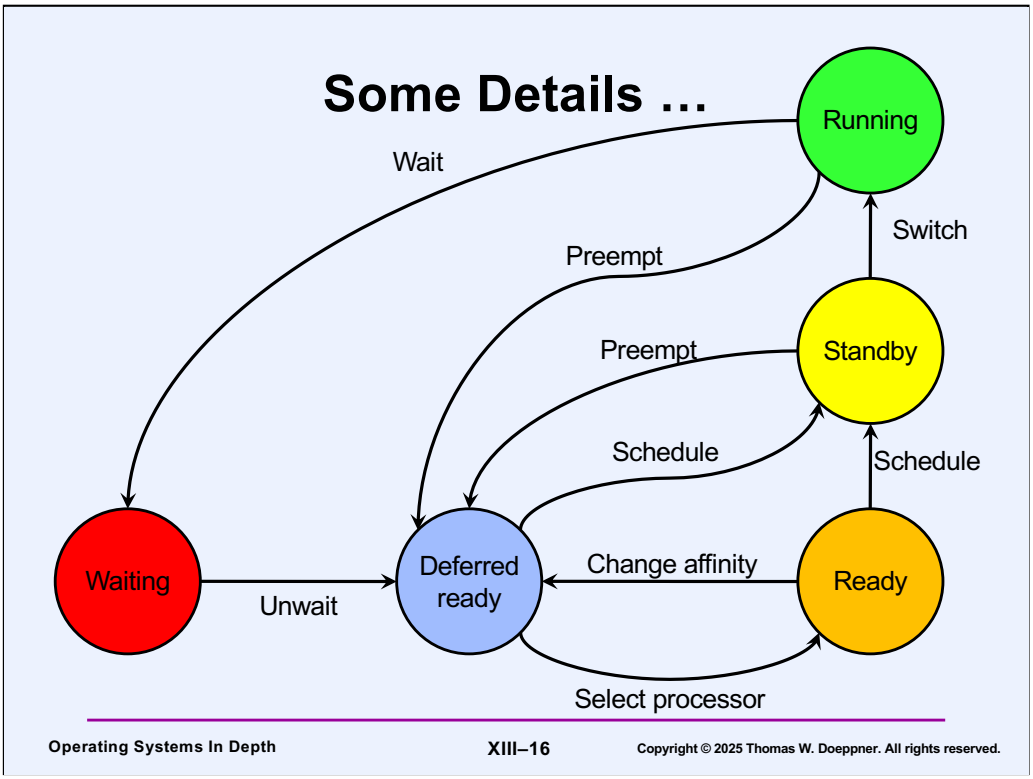
Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Which Processor?

- Newly created thread assigned *ideal processor*
  - randomly chosen
- May also set *affinity mask*
  - may be scheduled only on processors in mask
- Scheduling decision:
  - if idle processors available
    - first preference: ideal processor (if idle)
    - second preference: most recent processor (if idle)
  - otherwise
    - joins run queue of ideal processor

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Recall that each processor has a set of run queues, one for each priority value.



When a thread is created and first made runnable, its creator (running in kernel mode) puts it in the **deferred ready** state and enqueues it in the deferred ready queue associated with the current processor. It's also randomly assigned an **ideal processor**, on which it will be scheduled if available. This helps with load balancing. Its creator (or, later, the thread itself) may also give it an affinity mask indicating the set of processors on which it may run.

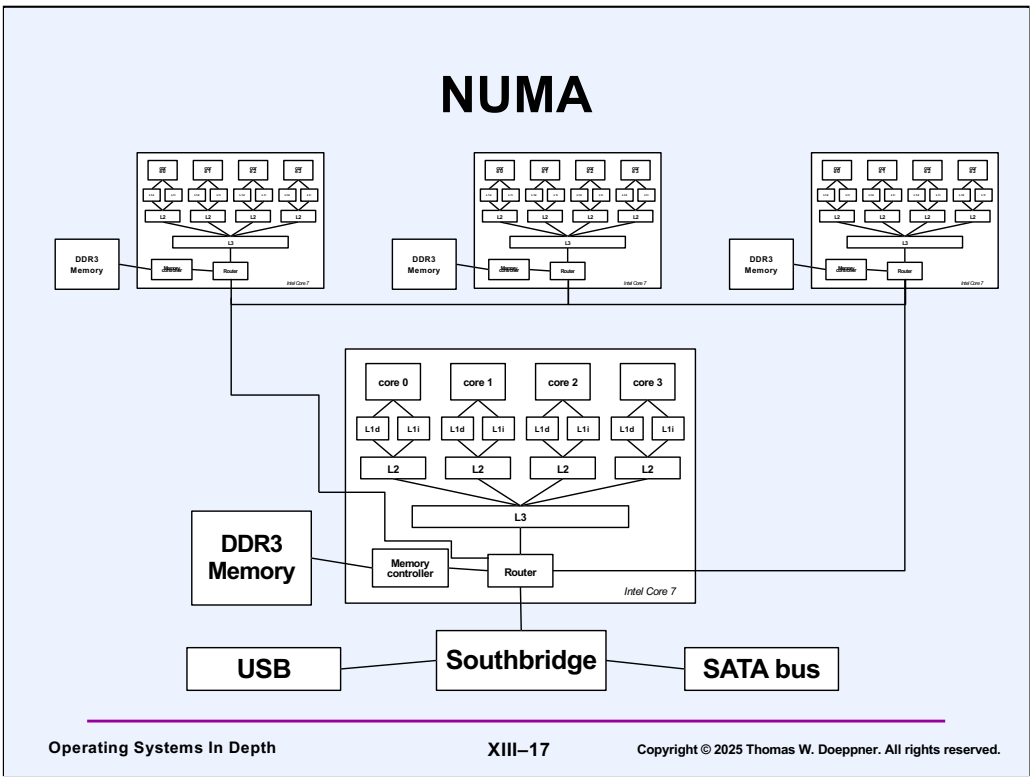
Each processor, each time it completes the handling of the pending DPC requests, checks its deferred ready queue. If there are any threads in it, it processes them, assigning them to processors. This works as follows. The DPC handler first checks to see if there are any idle processors that the thread can run on (if it has an affinity mask, then it can run only on the indicated processors). If there are any acceptable idle processors, preference is given first to the thread's ideal processor, then to the last processor on which it ran (to take advantage of the thread's cache footprint). The thread is then put in the **standby** state and given to the selected processor as its next thread. The processor would be currently running its idle thread, which repeatedly checks for a standby thread. Once found, the processor switches to the standby thread.

If there are no acceptable idle processors, then the thread is assigned to its ideal processor. The DPC handler checks to see if the thread has a higher priority than what's running on that processor. If so, it puts the thread in the standby state and sends the processor an interrupt. When the processor returns from its interrupt handler it will notice the higher-priority thread in standby state and switch to it, after first putting its current thread on its deferred ready list. Otherwise, the DPC handler puts the thread in the ready state and puts it in one of the ideal processor's ready queues according to the thread's priority.

When a thread completes its time quantum (time slice) (which is dealt with by a DPC), the processor searches its ready queues for a thread of equal or higher priority and switches to it, if it finds one, and puts the preempted thread on its deferred ready queue, from which it will be assigned a processor as described above. Otherwise, it continues with the current thread.

An executing thread might perform some sort of blocking operation, putting itself in a wait queue. Its processor then searches its ready queues for the next thread to run.

When a thread is made runnable after it has been in the wait state, it's put into the deferred ready queue of the processor on which the thread doing the **unwait** operation was running.



Recent processor designs put a memory controller inside of each processor chip, with separate memory attached to each chip. Thus, access to the attached memory is relatively quick. To reach memory attached to other chips from a particular chip, the request must be routed through one or more other processor chips. Thus, accessing other memory is more time-consuming than accessing local memory. Such systems are referred to as **non-uniform memory access** (NUMA) systems. What's shown here is the Intel Core 7 with Intel's **QuickPath Interconnect**.

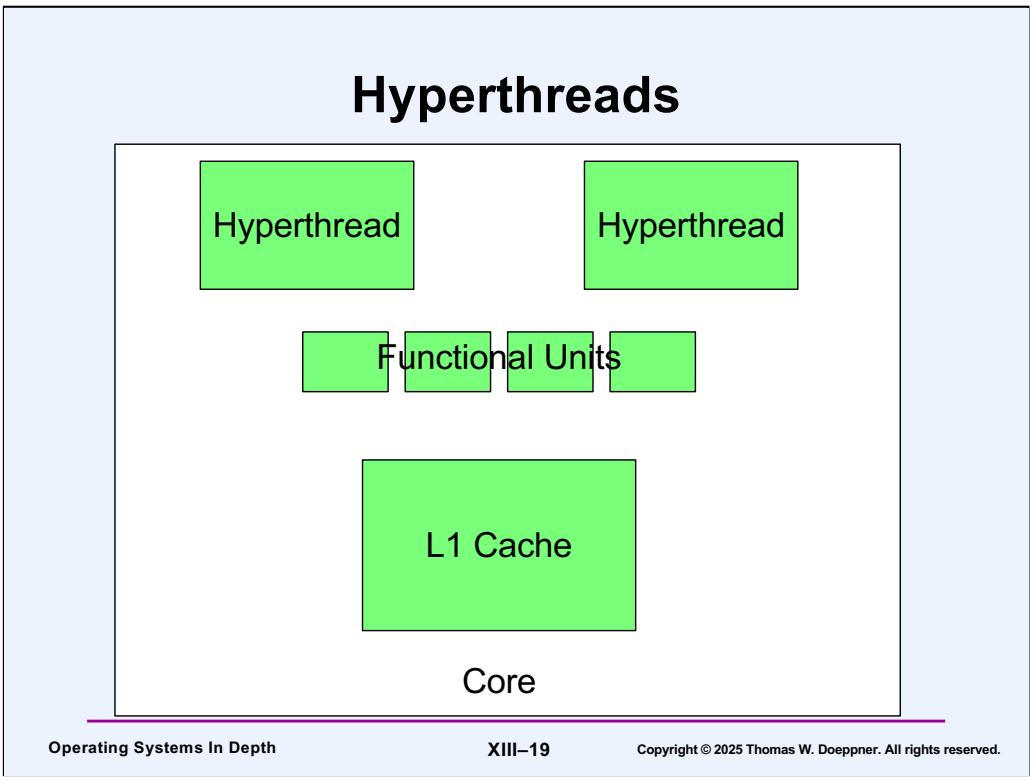
More information can be found at http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.

Scheduling Concerns

- Hyperthreads
  - two instruction streams sharing same functional units and same L1 cache
- How long does cache footprint matter?
  - what cache parameters are important?
- When is it a good idea to put a thread on:
  - a different core?
  - a different NUMA node?

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

How Linux deals with these concerns is discussed at http://lwn.net/Articles/80911/ and https://lwn.net/Articles/752977/.



Since hyper threads share the L1 cache (and also L2 and L3 caches), there's no real cost in switching a thread from one hyperthread to another in the same core. However, note that if two threads are running concurrently on two hyperthreads of the same core, each will most likely run substantially slower than if it were running on the core without a competing hyperthread (this is because the two hyperthreads on the core share the same set of functional units).

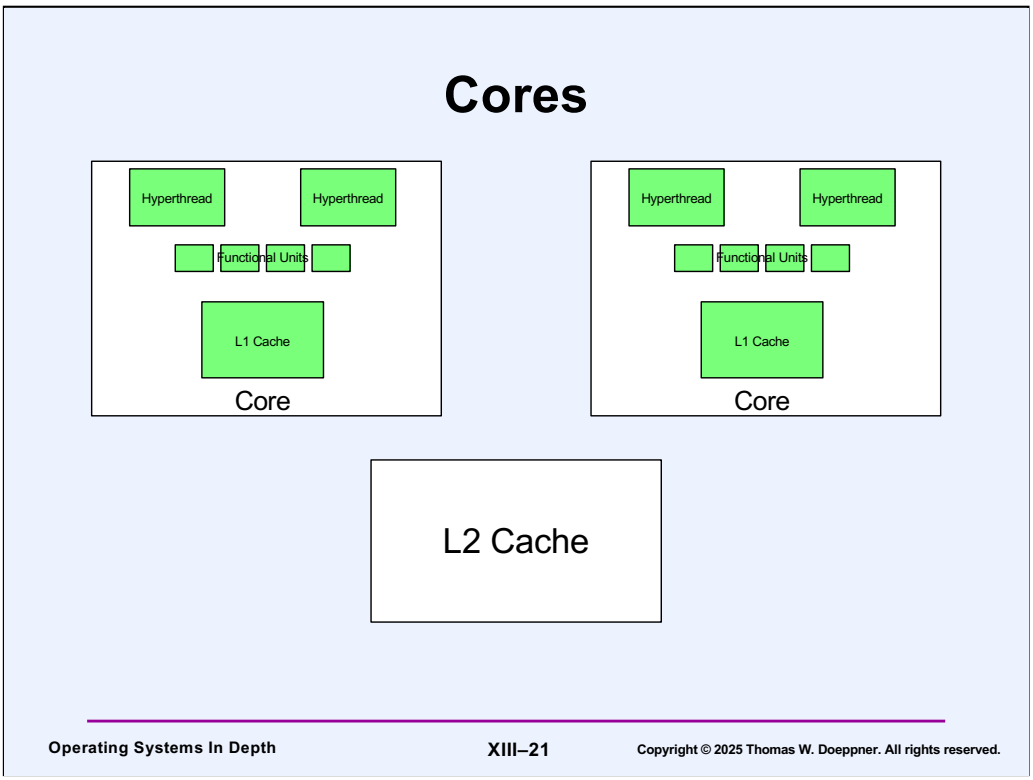
Quiz 1

We have a two-core processor with two hyperthreads per core. We have exactly two runnable threads.

a) for best performance, each thread should run on a separate core

b) it doesn't matter which hyperthreads are used to run the two threads, as long as two hyperthreads are used

Copyright © 2025 Thomas W. Doeppner. All rights reserved.



If a thread has run recently on one core, then it's clearly better to run it on that core again than to move it to another core, due to its footprint in its L1 cache. But suppose both hyperthreads of the other core (sharing the same L2 cache) are idle, and one hyperthread of our thread's original core is busy, but the other hyperthread is idle. Does it now make sense to run our thread on the original core? Or should it be moved to the other core?

Suppose we create a new thread in a multithreaded process. If all cores have at least one busy hyperthread, is it better to run the new thread on the same core as its creator, or on a different core? Similarly, suppose rather than creating a new thread on a multithreaded process, a thread calls fork, thus creating a new (single-threaded) process. Should this new thread run on its creator's core or on a different core?

Quiz 2

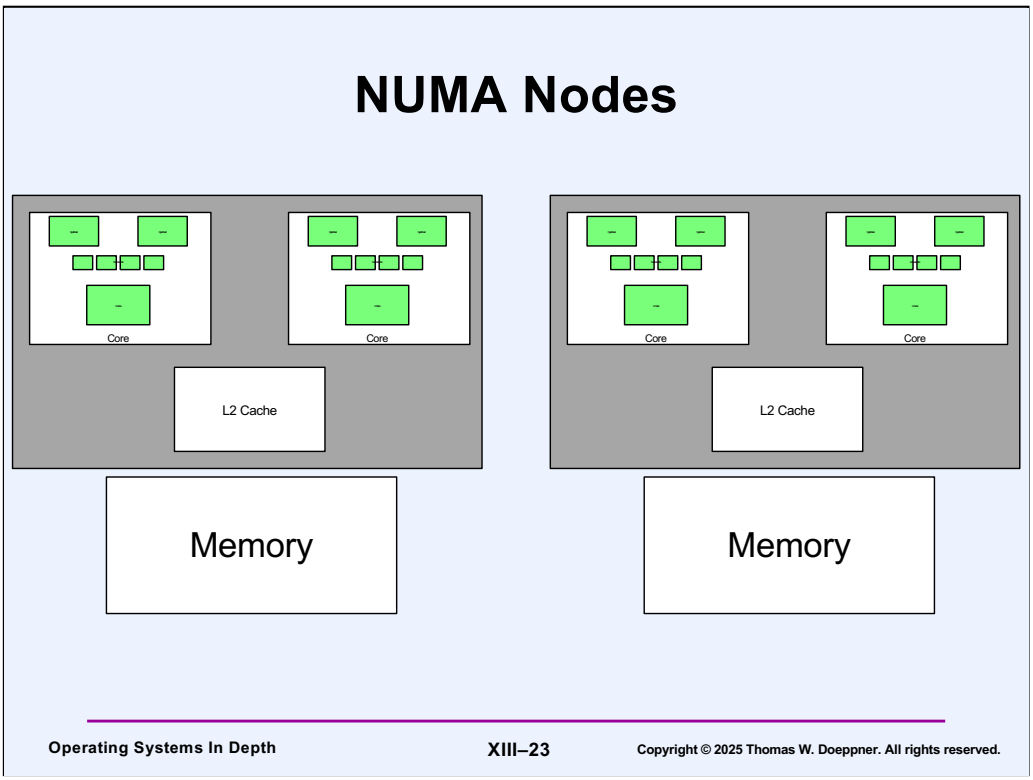
We have a two-core processor with four hyperthreads per core. We have four runnable threads.

a) it doesn't matter which hyperthreads are used to run the four threads, as long as four hyperthreads are used

b) it does matter: we should make sure that two hyperthreads of each core are used

c) it does matter: not only should we make sure that two hyperthreads of each core are used, but we also need to consider whether any of the threads are in the same process

Copyright © 2025 Thomas W. Doeppner. All rights reserved.



Now we have threads running on separate NUMA nodes. It is clearly very expensive to move a thread from one node to another if its address space remains in the memory of the of the original node.

Quiz 3

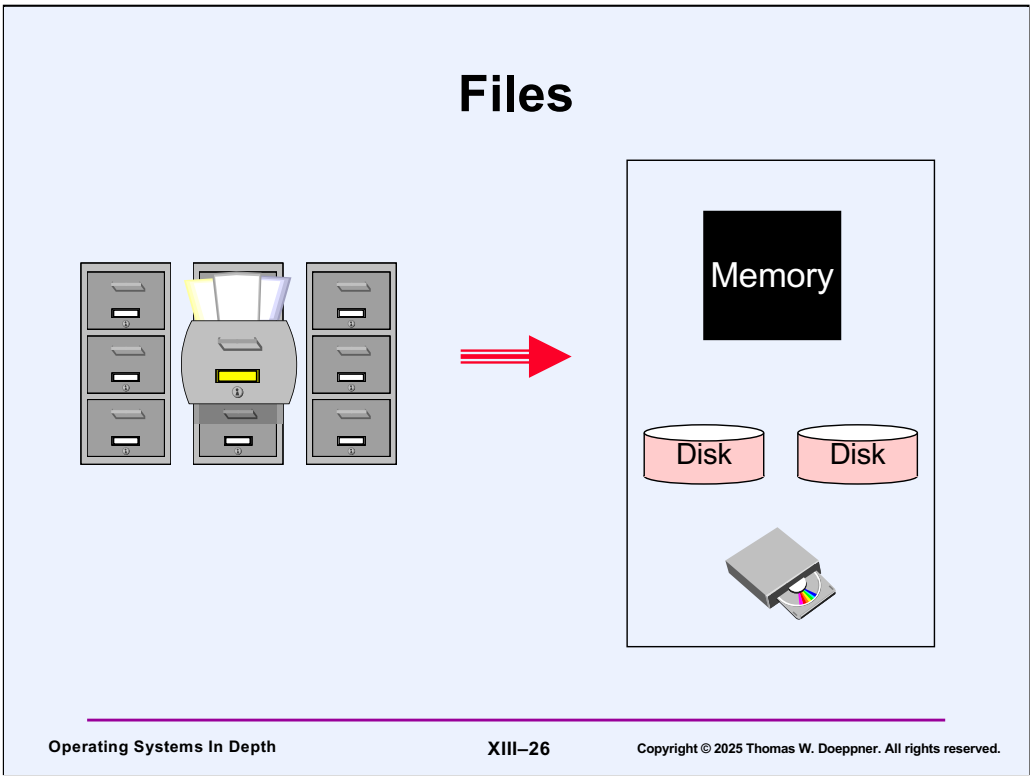
We have a system with two NUMA nodes. Most of the hyperthreads are busy on one of the nodes, the other node is completely idle. What operation performed by a thread would make the thread (or the thread it creates) a good candidate to move to the other node?

a) pthread_create
b) fork
c) execv
d) waitpid

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

File Systems Part 1

Copyright © 2025 Thomas W. Doeppner. All rights reserved.



File systems provide a simple abstraction of permanent storage, i.e., storage that doesn't go away when your program terminates or you shut down the computer. A user accesses information, represented as files, and the operating system is responsible for retrieving and storing this information. We expect files to last a long time and we associate names (as opposed to addresses) with files.

Requirements

- Permanent storage
  - resides on disk (or alternatives)
  - survives software and hardware crashes
    - (including loss of disk?)
- Quick, easy, and efficient
  - satisfies needs of most applications
    - how do applications use permanent storage?

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

File systems are responsible for permanent storage. This usually (but not always) means that disks are used as the storage medium.

Applications

- Software development
  - text editors
  - linkers and loaders
  - source-code control
- Document processing
  - editing
  - browsing
- Web stuff
  - serving
  - browsing
- Program execution
  - paging

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Some typical classes of applications are shown in the slide. What sort of requirements do they place on file systems?

Needs

- Directories
  - convenient naming
  - fast lookup
- File access
  - sequential is very common!
  - "random access" is relatively rare

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

File systems don't provide much functionality (unlike, for example, database systems), but what they provide must be provided well.

S5FS

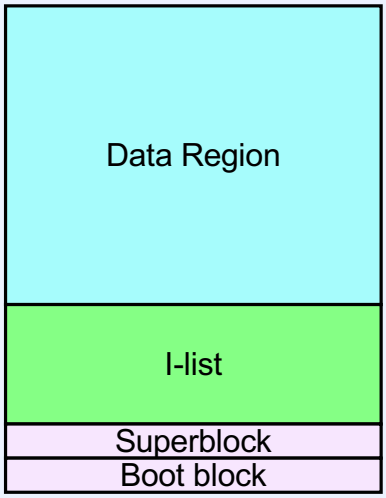
- A simple file system
  - slow
  - not terribly tolerant to crashes
  - reasonably efficient in space
    - no compression
- Concerns
  - on-disk data structures
    - file representation
    - free space
- It's the Weenix file system!

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

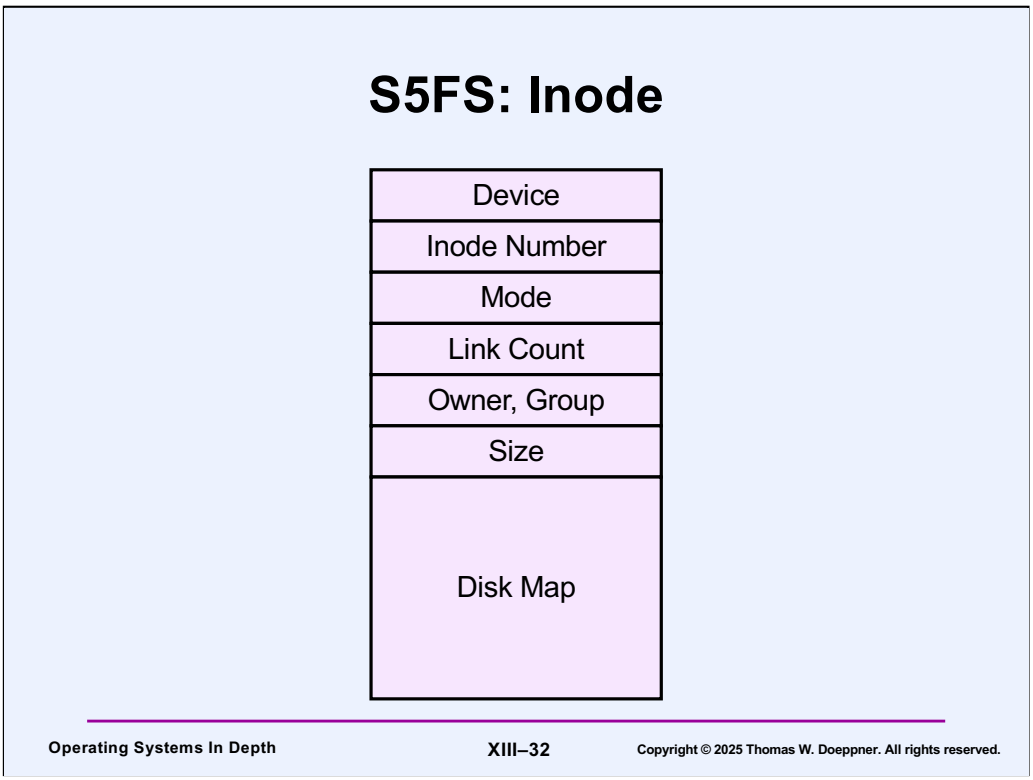
The first file system we discuss is known as the **S5 File System** — it is based on the original Unix file system, developed in the late sixties and early seventies. The name S5 comes from the fact that this was the only file system supported in early versions of what's known as Unix System V.

The Weenix file system is a somewhat simplified version of S5FS.

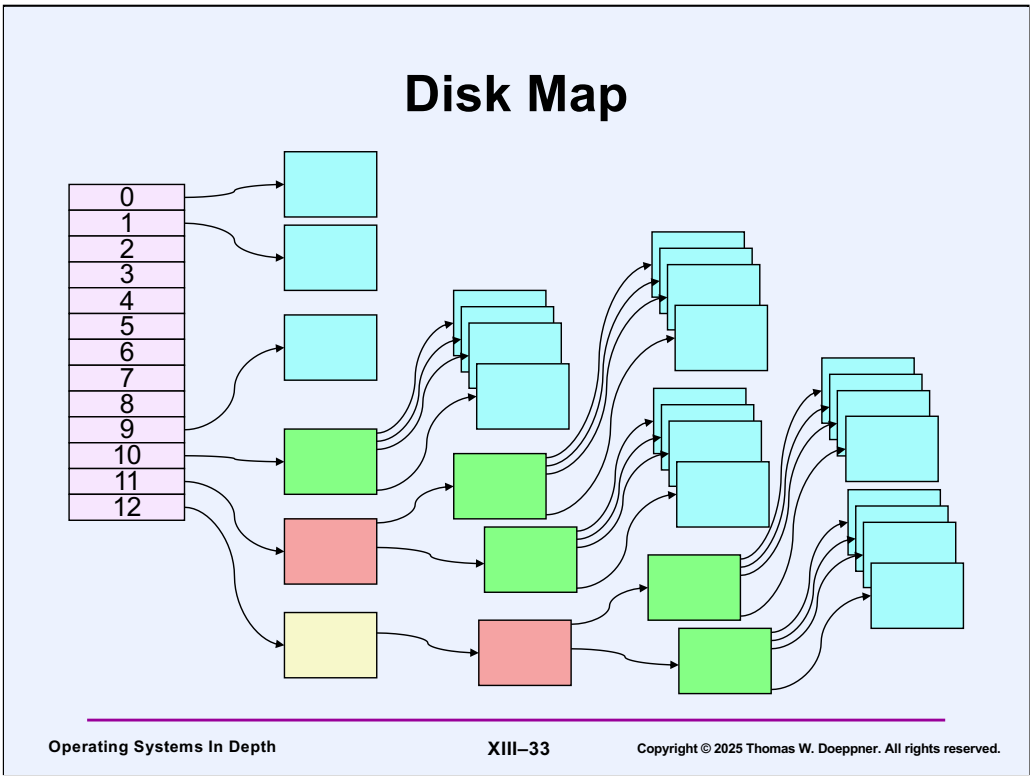
S5FS Layout



Copyright © 2025 Thomas W. Doeppner. All rights reserved.



Inodes (standing for **index nodes**), used in both S5FS and FFS, are essentially what's added to the vnode superclass to form the subclasses for the two file systems (they also include some of what's in the superclass). In addition, they are what's stored on disk to represent the file.



The purpose of the disk-map portion of the inode is to represent where the blocks of a file are on disk. I.e., it maps block numbers relative to the beginning of a file into block numbers relative to the beginning of the file system. Each block is 1024 (1K) bytes long. (It was 512 bytes long in the original Unix file system.) The data structure allows fast access when a file is accessed sequentially, and, with the help of caching, reasonably fast access when the file is used for paging (and other "random" access).

The disk map consists of 13 pointers to disk blocks, the first 10 of which point to the first 10 blocks of the file. (Thus the pointers are actually block numbers on the disk.) The first 10Kb of a file are accessed directly. If the file is larger than 10Kb, then pointer number 10 points to a disk block called an **indirect block**. This block contains up to 256 (4-byte) pointers to **data blocks** (i.e., 256KB of data). If the file is bigger than this (256K +10K = 266K), then pointer number 11 points to a **double indirect block** containing 256 pointers to indirect blocks, each of which contains 256 pointers to data blocks (64MB of data). If the file is bigger than this (64MB + 256KB + 10KB), then pointer number 12 points to a **triple indirect block** containing up to 256 pointers to double indirect blocks, each of which contains up to 256 pointers pointing to single indirect blocks, each of which contains up to 256 pointers pointing to data blocks (potentially 16GB, although the real limit is 2GB, since the file size, a signed number of bytes, must fit in a 32-bit word).

If a disk pointer (block number) is zero, it's treated as if it points to a block containing all zeroes (without the need for this block of zeroes to actually exist on disk).

The Weenix version of S5FS supports the indirect block, but not the double or triple indirect blocks.

Quiz 4

Suppose a new file is created. (At this point it occupies zero blocks.) Then one byte is written to it at byte offset $(266 \times 2^{10}) + 1$. Assume the block size is 2^{10} and block addresses occupy four bytes. How many blocks are required to represent the file, not counting its inode?

a) 1

b) 3

c) 270

d) more than 270

Copyright © 2025 Thomas W. Doeppner. All rights reserved.

Quiz 5

Suppose one now writes to all locations in the file, from its beginning up to the location written to in the previous slide (byte offset (266×2^{10}) + 1). How many blocks are required to represent the file, not counting its inode?

a) 1

b) 3

c) 270

d) more than 270

Copyright © 2025 Thomas W. Doeppner. All rights reserved.