CS 134:
Operating Systems
Disk Deja-vu

2013-05-17

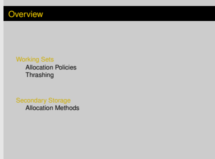
CS34


CS 134:
Operating Systems
Disk Deja-vu

Overview

- Working Sets
  - Allocation Policies
  - Thrashing
- Secondary Storage
  - Allocation Methods

└Overview



Overview

Working Sets
Allocation Policies
Thrashing

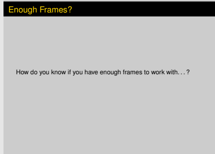
Secondary Storage
Allocation Methods

Enough Frames?

How do you know if you have enough frames to work with. . . ?

2013-05-17

CS34
└Working Sets
  └Enough Frames?



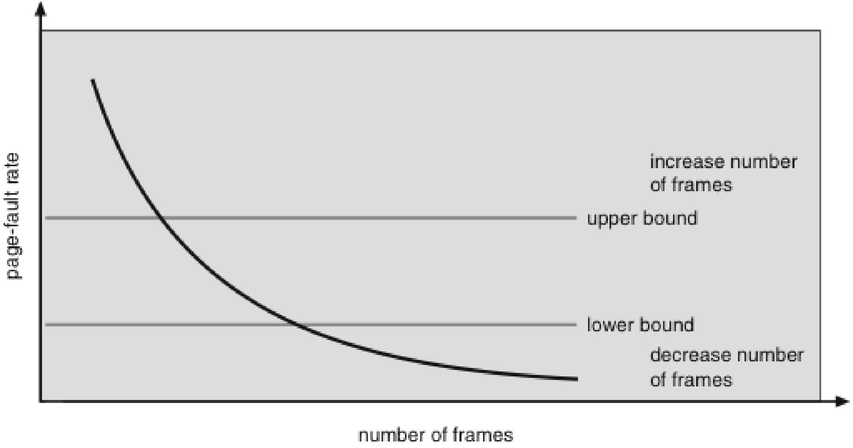
Enough Frames?

How do you know if you have enough frames to work with. . . ?

Working Sets

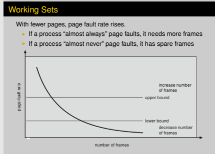
With fewer pages, page fault rate rises.

- If a process “almost always” page faults, it needs more frames
- If a process “almost never” page faults, it has spare frames



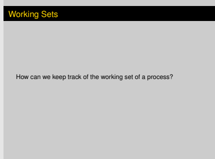
2013-05-17

CS34
└Working Sets
  └Working Sets


Working Sets
With fewer pages, page fault rate rises.
- If a process “almost always” page faults, it needs more frames
- If a process “almost never” page faults, it has spare frames

Working Sets

How can we keep track of the working set of a process?



Working Sets

How can we keep track of the working set of a process?

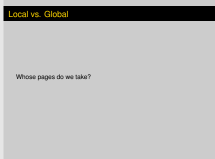
Formal definition is “pages referenced in last *k* accesses.” Close approximation is “pages referenced in last *n* ms.” Note that this is pretty close to what the CLOCK algorithm does, except that other processes can interfere. Which leads to…

Local vs. Global

Whose pages do we take?

2013-05-17

CS34
- Working Sets
  - Allocation Policies
    - Local vs. Global



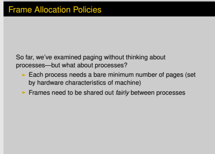
Local vs. Global

Whose pages do we take?

Frame Allocation Policies

So far, we've examined paging without thinking about processes—but what about processes?

- Each process needs a bare minimum number of pages (set by hardware characteristics of machine)
- Frames need to be shared out *fairly* between processes



Frame Allocation Policies

So far, we've examined paging without thinking about processes—but what about processes?

- Each process needs a bare minimum number of pages (set by hardware characteristics of machine)
- Frames need to be shared out *fairly* between processes

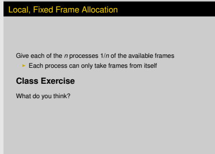
Local, Fixed Frame Allocation

Give each of the n processes $1/n$ of the available frames

- Each process can only take frames from itself

Class Exercise

What do you think?



Local, Fixed Frame Allocation

Give each of the n processes $1/n$ of the available frames

- Each process can only take frames from itself

Class Exercise

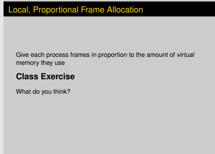
What do you think?

Local, Proportional Frame Allocation

Give each process frames in proportion to the amount of *virtual* memory they use

Class Exercise

What do you think?



Local, Proportional Frame Allocation

Give each process frames in proportion to the amount of *virtual* memory they use

Class Exercise

What do you think?

- Some processes use a lot of VM, but don’t access it often
- Some processes use a little VM, but access it often
- Not fair

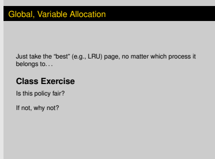
Global, Variable Allocation

Just take the “best” (e.g., LRU) page, no matter which process it belongs to. . .

Class Exercise

Is this policy fair?

If not, why not?



Global, Variable Allocation

Just take the “best” (e.g., LRU) page, no matter which process it belongs to. . .

Class Exercise

Is this policy fair?

If not, why not?

Local, Variable Allocation

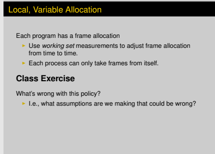
Each program has a frame allocation

- Use *working set* measurements to adjust frame allocation from time to time.
- Each process can only take frames from itself.

Class Exercise

What's wrong with this policy?

- I.e., what assumptions are we making that could be wrong?



Local, Variable Allocation

Each program has a frame allocation
- Use *working set* measurements to adjust frame allocation from time to time.
- Each process can only take frames from itself.

Class Exercise

What's wrong with this policy?
- I.e., what assumptions are we making that could be wrong?

Wrong assumptions: that we can measure working sets properly. That we can fit all working sets in memory.

Local, Variable Allocation

Each program has a frame allocation

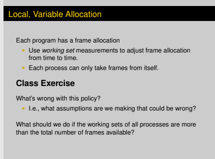
- Use *working set* measurements to adjust frame allocation from time to time.
- Each process can only take frames from itself.

Class Exercise

What’s wrong with this policy?

- I.e., what assumptions are we making that could be wrong?

What should we do if the working sets of all processes are more than the total number of frames available?



Local, Variable Allocation

Each program has a frame allocation
- Use *working set* measurements to adjust frame allocation from time to time.
- Each process can only take frames from itself.

Class Exercise

What’s wrong with this policy?
- I.e., what assumptions are we making that could be wrong?

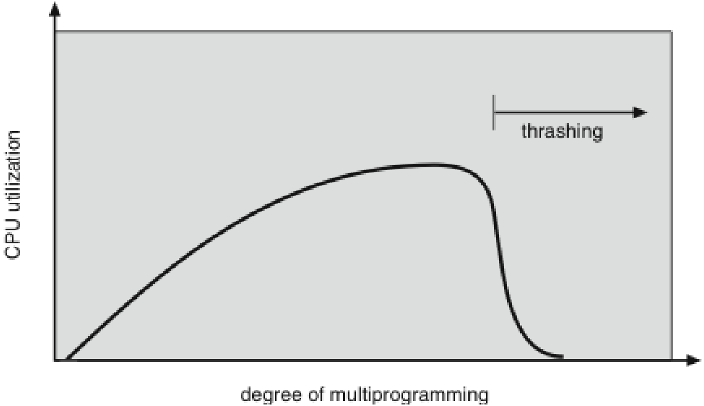
What should we do if the working sets of all processes are more than the total number of frames available?

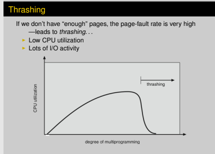
Wrong assumptions: that we can measure working sets properly. That we can fit all working sets in memory.

Thrashing

If we don’t have “enough” pages, the page-fault rate is very high —leads to *thrashing*. . .

- Low CPU utilization
- Lots of I/O activity




Thrashing

If we don’t have “enough” pages, the page-fault rate is very high —leads to *thrashing*. . .

- Low CPU utilization
- Lots of I/O activity

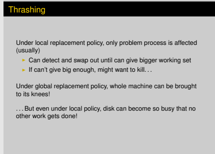
Thrashing

Under local replacement policy, only problem process is affected (usually)

- Can detect and swap out until can give bigger working set
- If can't give big enough, might want to kill. . .

Under global replacement policy, whole machine can be brought to its knees!

. . . But even under local policy, disk can become so busy that no other work gets done!



Thrashing

Under local replacement policy, only problem process is affected (usually)

- Can detect and swap out until can give bigger working set
- If can't give big enough, might want to kill. . .

Under global replacement policy, whole machine can be brought to its knees!

. . . But even under local policy, disk can become so busy that no other work gets done!

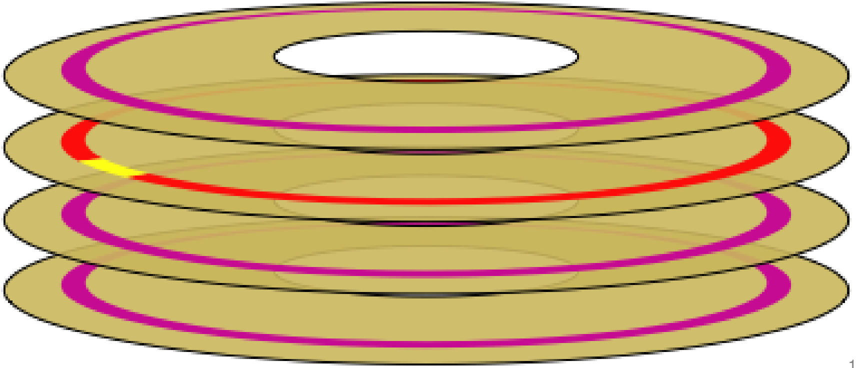
Secondary Storage

I.e., Storing stuff on disk, SSD, or network—why?

Class Exercise

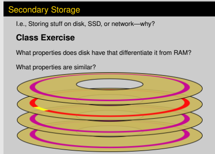
What properties does disk have that differentiate it from RAM?

What properties are similar?



2013-05-17

CS34
└Secondary Storage
　└Secondary Storage



Secondary Storage

I.e., Storing stuff on disk, SSD, or network—why?

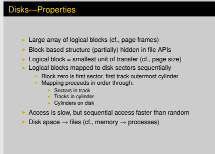
Class Exercise

What properties does disk have that differentiate it from RAM?

What properties are similar?

Disks—Properties

- Large array of logical blocks (cf., page frames)
- Block-based structure (partially) hidden in file APIs
- Logical block = smallest unit of transfer (cf., page size)
- Logical blocks mapped to disk sectors sequentially
  - Block zero is first sector, first track outermost cylinder
  - Mapping proceeds in order through:
    - Sectors in track
    - Tracks in cylinder
    - Cylinders on disk
- Access is slow, but sequential access faster than random
- Disk space → files (cf., memory → processes)



Disks—Properties

- Large array of logical blocks (cf., page frames)
- Block-based structure (partially) hidden in file APIs
- Logical block = smallest unit of transfer (cf., page size)
- Logical blocks mapped to disk sectors sequentially
  - Block zero is first sector, first track outermost cylinder
  - Mapping proceeds in order through:
    - Sectors in track
    - Tracks in cylinder
    - Cylinders on disk
- Access is slow, but sequential access faster than random
- Disk space → files (cf., memory → processes)

Disk as One File

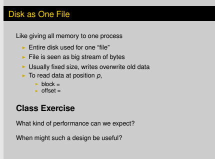
Like giving all memory to one process

- Entire disk used for one “file”
- File is seen as big stream of bytes
- Usually fixed size, writes overwrite old data
- To read data at position *p*,
    - block =
    - offset =

Class Exercise

What kind of performance can we expect?

When might such a design be useful?



Disk as One File

Like giving all memory to one process

- Entire disk used for one “file”
- File is seen as big stream of bytes
- Usually fixed size, writes overwrite old data
- To read data at position *p*,
    - block =
    - offset =

Class Exercise

What kind of performance can we expect?

When might such a design be useful?

Random access is fast (as possible). Sequential is fast. Best for special cases, such as databases, paging/swap files.

OS still has work to do, preserving illusion of byte stream of bytes rather than discrete blocks.

Disk as One File

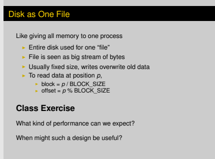
Like giving all memory to one process

- Entire disk used for one “file”
- File is seen as big stream of bytes
- Usually fixed size, writes overwrite old data
- To read data at position *p*,
  - block = *p* / BLOCK_SIZE
  - offset = *p* % BLOCK_SIZE

Class Exercise

What kind of performance can we expect?

When might such a design be useful?



Disk as One File

Like giving all memory to one process

- Entire disk used for one “file”
- File is seen as big stream of bytes
- Usually fixed size, writes overwrite old data
- To read data at position *p*,
  - block = *p* / BLOCK_SIZE
  - offset = *p* % BLOCK_SIZE

Class Exercise

What kind of performance can we expect?

When might such a design be useful?

Random access is fast (as possible). Sequential is fast. Best for special cases, such as databases, paging/swap files.

OS still has work to do, preserving illusion of byte stream of bytes rather than discrete blocks.

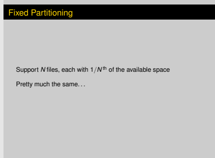
Fixed Partitioning

Support N files, each with $1/N^{\text{th}}$ of the available space

Pretty much the same…

2013-05-17

CS34
└Secondary Storage
  └Allocation Methods
    └Fixed Partitioning



Fixed Partitioning

Support N files, each with $1/N^{\text{th}}$ of the available space

Pretty much the same…

Dynamic Partitioning (aka Contiguous Allocation)

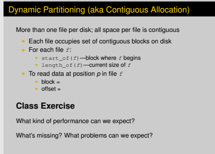
More than one file per disk; all space per file is contiguous

- Each file occupies set of contiguous blocks on disk
- For each file `f`:
    - `start_of(f)`—block where `f` begins
    - `length_of(f)`—current size of `f`
- To read data at position *p* in file `f`
    - block =
    - offset =

Class Exercise

What kind of performance can we expect?

What's missing? What problems can we expect?



Dynamic Partitioning (aka Contiguous Allocation)

More than one file per disk; all space per file is contiguous

- Each file occupies set of contiguous blocks on disk
- For each file `f`:
    - `start_of(f)`—block where `f` begins
    - `length_of(f)`—current size of `f`
- To read data at position *p* in file `f`
    - block =
    - offset =

Class Exercise

What kind of performance can we expect?

What's missing? What problems can we expect?

Random & sequential access fast. External fragmentation when files deleted. What about compaction? Files can't grow unless there's empty space next to them.

Dynamic Partitioning (aka Contiguous Allocation)

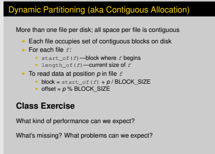
More than one file per disk; all space per file is contiguous

- Each file occupies set of contiguous blocks on disk
- For each file `f`:
  - `start_of(f)`—block where `f` begins
  - `length_of(f)`—current size of `f`
- To read data at position *p* in file `f`
  - block = `start_of(f)` + *p* / BLOCK_SIZE
  - offset = *p* % BLOCK_SIZE

Class Exercise

What kind of performance can we expect?

What's missing? What problems can we expect?



Dynamic Partitioning (aka Contiguous Allocation)

More than one file per disk; all space per file is contiguous

- Each file occupies set of contiguous blocks on disk
- For each file `f`:
  - `start_of(f)`—block where `f` begins
  - `length_of(f)`—current size of `f`
- To read data at position *p* in file `f`
  - block = `start_of(f)` + *p* / BLOCK_SIZE
  - offset = *p* % BLOCK_SIZE

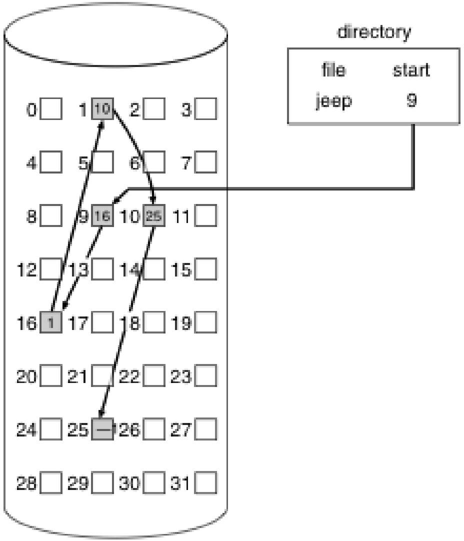
Class Exercise

What kind of performance can we expect?

What's missing? What problems can we expect?

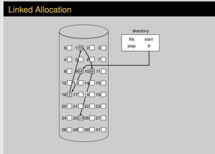
Random & sequential access fast. External fragmentation when files deleted. What about compaction? Files can't grow unless there's empty space next to them.

Linked Allocation



2013-05-17

CS34
└Secondary Storage
  └Allocation Methods
    └Linked Allocation



Linked Allocation

Make each file block point to next one:

- For each file *f*,

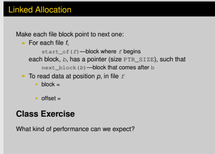
  `start_of(f)`—block where f begins

  each block, b, has a pointer (size `PTR_SIZE`), such that

  `next_block(b)`—block that comes after `b`
- To read data at position *p*, in file f
  - block =
  - offset =

Class Exercise

What kind of performance can we expect?



Find_block is a linear search. No external fragmentation. Random access is slow. Sequential access is okay (but not great.)

Linked Allocation

Make each file block point to next one:

- For each file *f*,

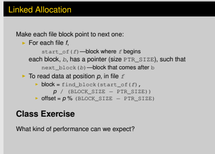
  `start_of(f)`—block where `f` begins

  each block, `b`, has a pointer (size `PTR_SIZE`), such that

  `next_block(b)`—block that comes after `b`
- To read data at position *p*, in file `f`
  - block = `find_block(start_of(f),`
    p `/ (BLOCK_SIZE − PTR_SIZE))`
  - offset = *p* % `(BLOCK_SIZE − PTR_SIZE)`

Class Exercise

What kind of performance can we expect?



Linked Allocation

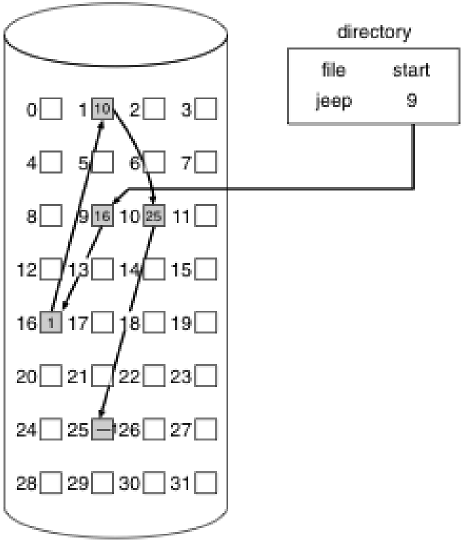
Make each file block point to next one:
For each file f,
start_of(f)—block where f begins
each block, b, has a pointer (size PTR_SIZE), such that
next_block(b)—block that comes after b
To read data at position p, in file f
block = find_block(start_of(f), p / (BLOCK_SIZE − PTR_SIZE))
offset = p % (BLOCK_SIZE − PTR_SIZE)

Class Exercise

What kind of performance can we expect?

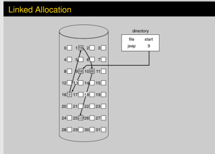
Find_block is a linear search. No external fragmentation. Random access is slow. Sequential access is okay (but not great.)

Linked Allocation

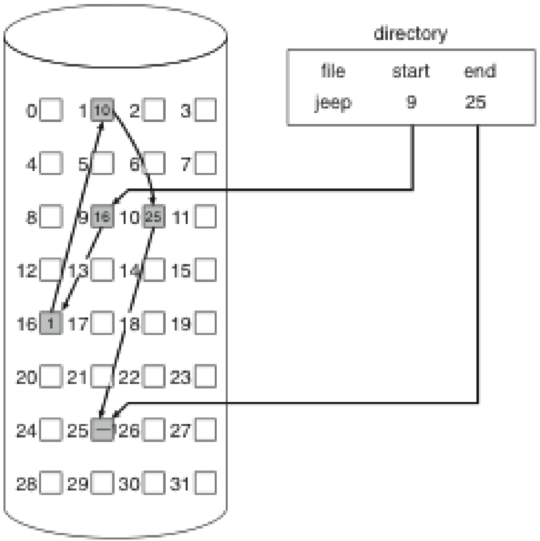


2013-05-17

CS34
└Secondary Storage
  └Allocation Methods
    └Linked Allocation

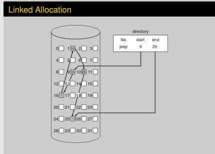


Linked Allocation



2013-05-17

CS34
└Secondary Storage
  └Allocation Methods
    └Linked Allocation



Adding an end pointer makes appends much cheaper.

File Allocation Table

Make each block in an array (`fat`) point to the next one block in file

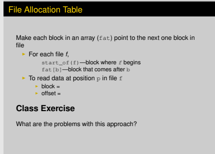
- For each file *f*,

  `start_of(f)`—block where *f* begins

  `fat[b]`—block that comes after `b`
- To read data at position `p` in file `f`
  - block =
  - offset =

Class Exercise

What are the problems with this approach?



File Allocation Table

Make each block in an array (`fat`) point to the next one block in file

- For each file *f*,

  `start_of(f)`—block where *f* begins

  `fat[b]`—block that comes after `b`
- To read data at position `p` in file `f`
  - block =
  - offset =

Class Exercise

What are the problems with this approach?

Here, `find_block` is a list search through the FAT, which is stored in memory: no (or few) extra disk accesses.

Problems: you still have essentially random allocation on the disk.

File Allocation Table

Make each block in an array (`fat`) point to the next one block in file

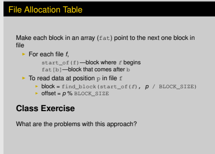
- For each file *f*,

    `start_of(f)`—block where `f` begins
    `fat[b]`—block that comes after `b`

- To read data at position `p` in file `f`
    - block = `find_block(start_of(f), p / BLOCK_SIZE)`
    - offset = *p* % `BLOCK_SIZE`

Class Exercise

What are the problems with this approach?



File Allocation Table

Make each block in an array (`fat`) point to the next one block in file

- For each file *f*,

    `start_of(f)`—block where `f` begins
    `fat[b]`—block that comes after `b`

- To read data at position `p` in file `f`
    - block = `find_block(start_of(f), p / BLOCK_SIZE)`
    - offset = *p* % `BLOCK_SIZE`

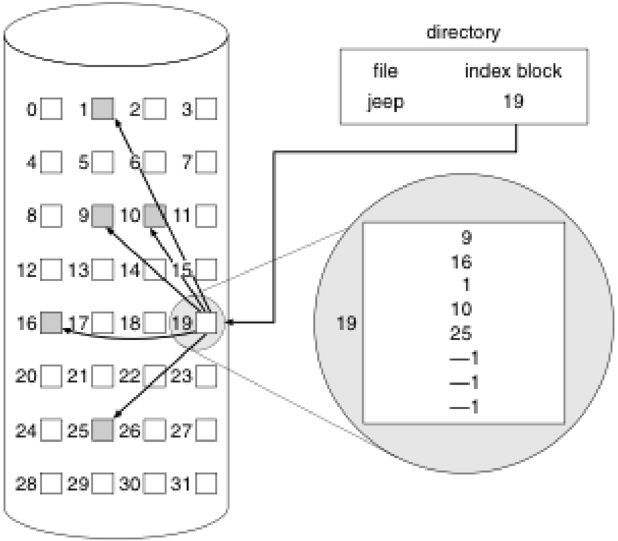
Class Exercise

What are the problems with this approach?

Here, `find_block` is a list search through the FAT, which is stored in memory: no (or few) extra disk accesses.

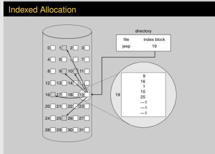
Problems: you still have essentially random allocation on the disk.

Indexed Allocation



2013-05-17

CS34
└Secondary Storage
  └Allocation Methods
    └Indexed Allocation



Indexed Allocation

Bring all pointers together in an index block (cf., page table)

- For each file *f*,

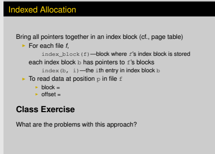
  `index_block(f)`—block where `f`'s index block is stored

  each index block `b` has pointers to `f`'s blocks

  `index(b, i)`—the `i`th entry in index block `b`
- To read data at position `p` in file `f`
  - block =
  - offset =

Class Exercise

What are the problems with this approach?



Random and sequential access are okay; you have dynamic access without external fragmentation. But the index block is overhead, and limits on the size of the index block also limit the sizes of files.

Indexed Allocation

Bring all pointers together in an index block (cf., page table)

- For each file *f*,

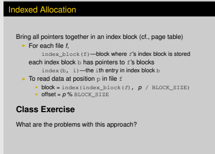
  `index_block(f)`—block where `f`'s index block is stored

  each index block `b` has pointers to `f`'s blocks

  `index(b, i)`—the `i`th entry in index block `b`
- To read data at position `p` in file `f`
  - block = `index(index_block(f), p / BLOCK_SIZE)`
  - offset = *p* % `BLOCK_SIZE`

Class Exercise

What are the problems with this approach?



Indexed Allocation

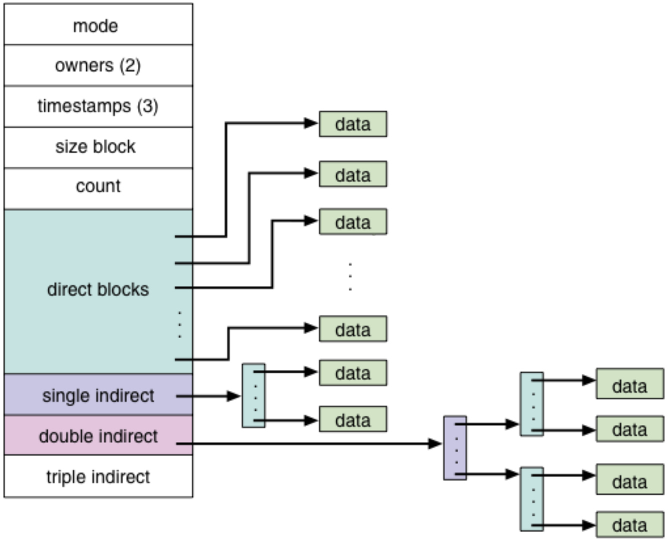
Bring all pointers together in an index block (cf., page table)
For each file f,
index_block(f)—block where f's index block is stored
each index block b has pointers to f's blocks
index(b, i)—the ith entry in index block b
To read data at position p in file f
block = index(index_block(f), p / BLOCK_SIZE)
offset = p % BLOCK_SIZE

Class Exercise

What are the problems with this approach?

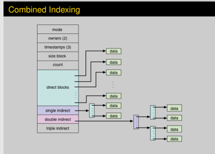
Random and sequential access are okay; you have dynamic access without external fragmentation. But the index block is overhead, and limits on the size of the index block also limit the sizes of files.

Combined Indexing



2013-05-17

CS34
└Secondary Storage
  └Allocation Methods
    └Combined Indexing



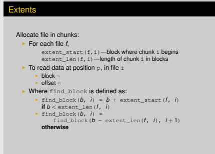
This has long been the standard in Unix-like systems. But it's not the only way!

Extents

Allocate file in chunks:

- For each file *f*,

  `extent_start(f,i)`—block where chunk `i` begins
  `extent_len(f,i)`—length of chunk `i` in blocks
- To read data at position `p`, in file `f`
  - block =
  - offset =
- Where `find_block` is defined as:
  - `find_block(`b`, `i`) = `b` + extent_start(`f`, `i`)` **if** $b <$ `extent_len(`f`, `i`)`
  - `find_block(`b`, `i`) =`
    `find_block(`b` - extent_len(`f`, `i`), `$i+1$`)`
    otherwise



Extents

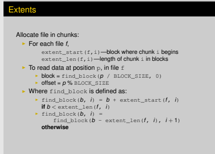
Allocate file in chunks:

- For each file *f*,

  `extent_start(f,i)`—block where chunk `i` begins
  `extent_len(f,i)`—length of chunk `i` in blocks
- To read data at position `p`, in file `f`
  - block = $\texttt{find\_block}(p\ /\ \texttt{BLOCK\_SIZE},\ 0)$
  - offset = p % `BLOCK_SIZE`
- Where `find_block` is defined as:
  - $\texttt{find\_block}(b,\ i) = b + \texttt{extent\_start}(f,\ i)$ **if** $b < \texttt{extent\_len}(f,\ i)$
  - $\texttt{find\_block}(b,\ i) =$ $\texttt{find\_block}(b - \texttt{extent\_len}(f,\ i),\ i+1)$ **otherwise**

2013-05-17

CS34
└ Secondary Storage
  └ Allocation Methods
    └ Extents



Extents

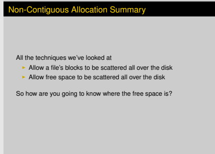
Allocate file in chunks:
- For each file *f*,
  `extent_start(f,i)`—block where chunk `i` begins
  `extent_len(f,i)`—length of chunk `i` in blocks
- To read data at position `p`, in file `f`
  - block = $\texttt{find\_block}(p\ /\ \texttt{BLOCK\_SIZE},\ 0)$
  - offset = p % `BLOCK_SIZE`
- Where `find_block` is defined as:
  - $\texttt{find\_block}(b,\ i) = b + \texttt{extent\_start}(f,\ i)$ **if** $b < \texttt{extent\_len}(f,\ i)$
  - $\texttt{find\_block}(b,\ i) =$ $\texttt{find\_block}(b - \texttt{extent\_len}(f,\ i),\ i+1)$ **otherwise**

Non-Contiguous Allocation Summary

All the techniques we've looked at

- Allow a file's blocks to be scattered all over the disk
- Allow free space to be scattered all over the disk

So how are you going to know where the free space is?



Non-Contiguous Allocation Summary

All the techniques we've looked at

- Allow a file's blocks to be scattered all over the disk
- Allow free space to be scattered all over the disk

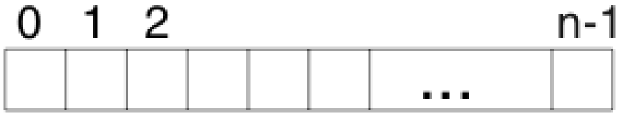
So how are you going to know where the free space is?

Desirable properties:

- Try to locate the file (or large pieces of the file) in the same region of the disk
  - Requires enough free space to be able to pick a region of the disk that has chunks of space free
- Minimize head movement

Free-Space Management—Bit Vector

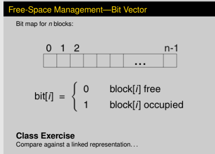
Bit map for *n* blocks:



$$\text{bit}[i] = \begin{cases} 0 & \text{block}[i] \text{ free} \\ 1 & \text{block}[i] \text{ occupied} \end{cases}$$

Class Exercise

Compare against a linked representation…


Free-Space Management—Bit Vector

Bit map for *n* blocks:

$$\text{bit}[i] = \begin{cases} 0 & \text{block}[i] \text{ free} \\ 1 & \text{block}[i] \text{ occupied} \end{cases}$$

Class Exercise

Compare against a linked representation…