Deadlock

1. Mutex: Only one process at a time can use a resource
2. Hold resource & wait: A process holding at least one resource is waiting to acquire additional resources held by other processes
3. No pre-emption: A resource can be released only voluntarily by the process holding it, after process has completed its task (hence use of the resource)
4. Circular wait: There exists a set {*P0*, *P1*, …, *Pn*} of waiting processes such that *P0* is waiting for a resource that is held by *P1*, *P1* is waiting for a resource that is held by *P2*, …, *Pn*–1 is waiting for a resource that is held by *Pn*, and *Pn* is waiting for a resource that is held by *P0*

If all 4 happens, deadlock MIGHT happen. Deadlock can be prevented by removing either conditions (necessary conditions but not sufficient)

Pi --> Rj (Pi req instance of Rj); Pi <-- Rj (Pi holding instance of Rj)

if graph no cycle -> no dL; if got cycle -> if only 1 instance per resource type, then dL; if several instances per resource type, possibility of dL

Handling Deadlocks

Real world OS do not handle deadlocks completely all the time

1. Deadlock Avoidance

- Pi needs to declare max resources it ever needs in the beginning
- Requests that lead to unsafe state will have to wait
- Don't give resources that might lead to future deadlocks, even if it is available now | - Always check if requests will lead to circular wait. Only grant requests with safe state.
- Before granting a resource request (even if request is valid and the requested resources are now available), check that the request will not cause the system to enter a deadlock state (not just no deadlock immediately, but not even later)

2. Deadlock Detection

- System allows deadlock to happen and then detects with this algorithm
- After detected, 1) abort all deadlock processes, this allow them to pre-empt resources (get them back) 2) Can also abort all dL processes 1 by 1 until there's no more deadlock; by priority, time exec, resources used 3) Restart all aborted processes

3. Deadlock Prevention

- Avoids potential dL situations by design, which is to disallow either 1 of the 4 necessary conditions for dL to happen
- No resource hold & wait: Must get all resources b4 process exec, only allow request 4 resources if process has none; must guarantee that whenever a process requests a resource, it does not hold any other resources. Con: starvation, low resource utilisation
- Allows pre-emption: Processes must release all resources its already holding if it needs other resources that require wait; restart process, must wait for every resources again Con: pre-emption costs and starvation
- Disallow circular wait: impose a total ordering of all the resource types, & require each P requests resources according to that. Con: Burden on programmer to ensure order by design w/o unnecessarily sacrificing utilisation

Avoidance – Safe State:

- When a process requests an available resource, system must decide if granting the request will leave the system in a safe state.
- System is in **safe state** if there exists a sequence <P1, P2, …, Pn> of all the processes in the system such that for each Pi, the resources that Pi will ever need can be satisfied by the currently available resources plus the resources held by all the preceding Pj, with j < i.
- System in safe state => no dL; system in unsafe state => possibility of dL

Avoidance => ensure system will never enter an unsafe state, so only grant a resource request if after granting, system will still be in a safe state.

Banker's Algorithm

- Multiple instances of each resource
- Each process must declare its maximum needs (will ever need)
- When a process requests a resource, it may have to wait
- After a process got all its resources, it must return them within finite amount of time (when P finishes its task)   Need[i,j] = Max[i,j] - Allocation[i,j]
- n = # processes, m = # resources   - Available[m], Max[n][m], Allocation[n][m], need[n][m]

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
   Work = *Available*   *// NB: Work, Available are both arrays*
   Finish[*i*] = *false* for *i* = 0, 1, …, *n* - 1
2. Find an *i* such that:
   (a) *Finish*[*i*] = *false*
   (b) *Need*[*i*] ≤ *Work*
   If no such *i* exists, go to Step 4
3. *Work* = *Work* + *Allocation*[*i*]   *// Allocation[i] is i-th row of Allocation matrix*
   Finish[*i*] = *true*
   go to Step 2
4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state; otherwise it's unsafe

Algorithm for Granting Request by Process P_i

Request[*i*] = request vector for process P_i. If $Request_i[i,j] = k$, then process P_i wants *k* instances of resource type R_j

1. If *Request*[*i*] ≤ *Need*[*i*] go to Step 2. Otherwise, raise error, since process has exceeded its maximum claim.
2. If *Request*[*i*] ≤ *Available*, go to Step 3. Otherwise P_i must wait, since the resources are not immediately available.
3. Try to allocate the requested resources to P_i by updating the resource allocation state as follows (i.e., assume we grant the new request):
   Available = *Available* - *Request*[*i*];
   $Allocation_i$ = *Allocation*[*i*] + *Request*[*i*];
   Need[*i*] = *Need*[*i*] - *Request*[*i*];

- If *new* state is *safe* ⇒ the resources are allocated to P_i
- If *new* state is *unsafe* ⇒ P_i must wait, and the old resource-allocation state is restored (i.e., new request not granted after all)

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initiali
   Work = *Available*
   For *i* = 1,2, …, *n*, if *Allocation*[i] ≠ 0, then
   Finish[i] = false; else *Finish*[i] = *true*
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b)*Request*[*i*] ≤ *Work*. *// What i requests now is available if all its preceding process*
   // finish; i isn't deadlocked already
   If no such *i* exists, go to Step 4
3. *Work* = *Work* + *Allocation*[*i*]. *// Assume i will finish and return the resources it holds*
   Finish[*i*] = *true*
   go to Step 2
4. If *Finish*[*i*] == *false* for some *i*, then the system is (already) in deadlock
   Moreover, if *Finish*[*i*] == *false*, then P_i is (already) deadlocked.

File: is a named collection of related information that is recorded on secondary storage. **Not the content itself**

- Abstract data type with 2 key attributes:

1) States/attributes: file data & meta data, usage
2) Interface: 'methods' to interact w file

- Regular files: store data – user data / system data

Directories/folders: organise files in a structured namespace

- Formats: NONE (uninterpreted bytes), SIMPLE (lines (.txt), fixed length records, variable length records), COMPLEX (executable files)
- Interpreted by: OS Kernel, System programs, User/application programs

File Attributes (Metadata): a set of data that gives info about other data, usually kept in directory structure, which (like the file's data) is maintained on disk

Name, identifier, type, location, size, protection (who can read,write), (user id, date, time etc)

File Interface (Methods): create, open, read/write, lseek(), delete, memory map (map file into process' address space), delete, truncate (remove data, keep attributes)

File usage information: opening files

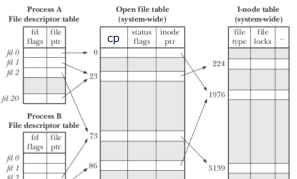
- Not good to pass file name to every file system operation (e.g., **read()**)
  - Name can be long and of variable length
  - Mapping of name to internal data structures takes time
- Program translates name into succinct *file descriptor* ("handle" to the file) at the beginning of a usage session
  - fd is (integer) index into *per-process* **file-descriptor (fd) table** (see Slide 10.11)
  - Index has meaning only in context of its process
- Each fd table entry points to *system-wide* **open file table** about *usage* of the opened file (see Slide 10.11), e.g.,
  - Current file offset (**cp**): pointer to current read/write location (byte offset, starting from 0)
  - Access status: mode granted like read, write/append, execute
  - Open count: how many fd table entries point to it – e.g., can't remove open file table entry if reference count is positive

Unix file system data structures

All these data structures are in kernel space:

NB: Process A did a **dup()** system call with argument 1, and the call returned 20 (i.e., fd 20 is now a *duplicate* of fd 1)

i-node (in right table) is Unix kernel's internal data structure for a file

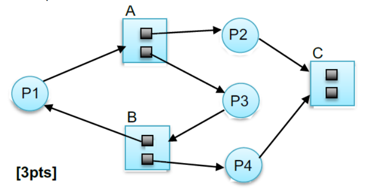


About *usage* of file (e.g., cp – current byte offset for read/write)

Mostly about file itself (e.g., locations of its data blocks)

Mappings between table entries

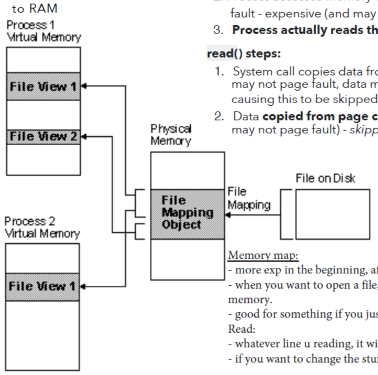
- Multiple file descriptor table entries can point to same open file table entry (*many-to-one* mapping)
  - A process can have two or more file descriptors referencing the same **open file table** (i.e., middle table on Slide 10.11) entry (e.g., after **dup()** as illustrated on previous slide)
  - Different processes can also have their file descriptors point to same open file table entry
    - Child inherits parent's file descriptors after **fork()**
      - Child gets its *own* fd table, but this fd table initially has the same content as the parent's fd table
    - Unrelated processes can pass file descriptors to each other, e.g., using "Unix domain sockets"
  - If two file descriptors **fd1** and **fd2** reference same open file entry, they share *usage* (e.g., **cp**) of the file – read/write through **fd1** will advance **cp** seen by **fd2**
- Multiple open file table entries can point to the same file (right table on Slide 10.11), i.e., also *many-to-one* mapping
  - *Different* usage instances of the same file (the different instances have *independent* **cp**)
  - e.g., a file opened multiple times by separate **open()** system calls (i.e., each open starts a *new* usage, instead of duplicating an existing usage as in **dup()**)
- Hence, in general, concurrent accesses to files are possible by different processes, shared files allow a form of IPC



[3pts]

	Allocation A	Allocation B	Allocation C	Request A	Request B	Request C	Available A	Available B	Available C
P1	0	1	0	1	0	0	0	0	2
P2	1	0	0	0	0	1			
P3	1	0	0	0	1	0			
P4	0	1	0	0	0	1			

View: a portion of the file
Mapping: move from disk to RAM



Memory map steps:

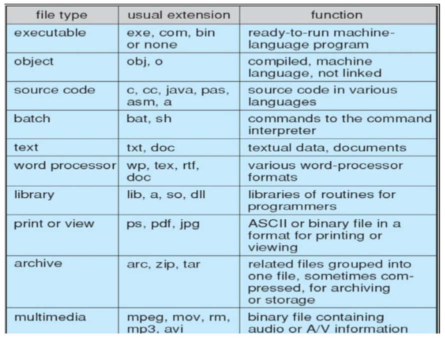
1. System call to create virtual mappings - very expensive
2. Process accesses memory for the first time, causing a page fault - expensive (and may need to be repeated if paged out)
3. **Process actually reads the memory**

read() steps:

1. System call copies data from disk to page cache (may or may not page fault, data may already be in page cache causing this to be skipped)
2. Data **copied from page cache to process memory** (may or may not page fault) - *skipped in memory mapping*

Memory map:
- more exp in the beginning, after that can refer easily
- when you want to open a file, it moves the entire file over to the memory.
- good for something if you just want to read-only

Read:
- whatever line u reading, it will copy from disk to its space.
- if you want to change the stuffs



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Child inherits entire fd table upon fork(), with the same state when fork() is called. After that, child and parent processes progress independently, with independent fd tables.

1. Each resource type in a computing system is given a unique integer id. A process can acquire resources incrementally, but when it requests a new resource, the id of the new resource must be strictly higher than the id of any resources already held by the process.

(a) Processes in the system can't become deadlocked. Why not? **Circular waiting can't happen.**

(b) Is the solution approach deadlock avoidance or deadlock prevention? Explain your answer.

Deadlock prevention. Because a resource request can be granted as long as it is legal and there are sufficient available resources now. There's no need to check the detailed runtime state (e.g., current and future allocation patterns) in the decision.

Is each of the following statements true or false?

(a) Each process in Unix has its own file descriptor table. **True.**

(b) A user process in Unix can change the entries of its file descriptor table without using system calls. **False.**

Is the above system deadlocked? If so, say which processes are involved a deadlock. If not, explain why not. **No. P2, P4, P3, P1 is a possible termination sequence of the processes.**

Storing a File: Directories
- A directory is also a file that has the names of other files as its content. In other words, a meta-data that organises files in a structured namespace.
- create, traverse, rename, search, list, delete

Purposes of Directory Structure:
- Efficiency - Locating a file/group of files quickly
- Naming - convenient & helpful to users
- Organisation - logical grouping of files by various properties
- Circle nodes = files, square nodes = file names
- EG /user1/a each user has own separate name space (no clashes)

Tree-structured Directories:
- Multiple levels of hierarchy allow more elaborate organisation of files (but full path names become long)

File Links:
- New type of file system objects (in addiiton to directories & files). Has path name in namespace like files, but name links to another existing file system object (hence same obj can now have multiple names (aliasing)

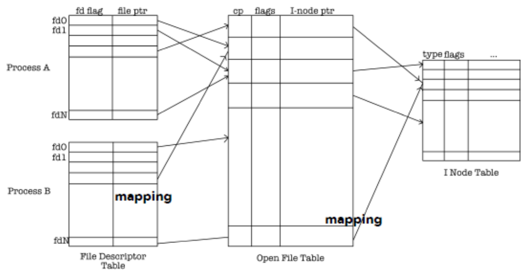
>> Symbolic Links (eg. /spell/count & /dict/count same file has two names)
When you delete the file, the link becomes broken
>> Hard Link (different names for the same file)
- An inode is deleted if all the file and all hard references to it are removed, then space will be cleared ie when reference count=0.

*** Hard link != copy, a copy operation copies entire content on secondary storage**
- Cyclic Directories: possible but dfs directory traversal may not terminate, also it will self-reference so reference count will never reach zero.

iNode table stored and maintained by kernel
Open file table: system wide. Only have 1 per computer, keeps track which files are open. Each entry points to a file
Files in RAM => memory mapping
Files in VM => read/mapping
Each process have their own file descriptor table.
1. Pa and Pb shares same open file table entry => share the same thing (fd0,fd3) Eg. for log files
*Files in RAM so memory mapping

Hello	-> Hi
World	-> World

If Pa rewrites first line, and Pb readline. current pointer is at next line. So Pb reads World
Usage of Pa/b is seen by Pa/b (sharing memory, interprocesses communication)
2. Pa&Pb have different open file entry but same I Node entry (fd3,fdn)
- points to same file but have 2 cp
Q: Pa rewrites first line to Hi
Pb reads Hi, not world *needs condition sync
||Q: A & B share same file entry?
check if they point to same open file entry
* Kernel can read/write open file table & inode table. (only 1 of each per computer)
* System programs cannot write but they can read, thus can show file syster (having root access)



Each table **keeps track of reference count**. Entry is deleted when reference count is zero
Note: Open File Table isn't the only table pointing to I-node table. Files also point to I-node table.

CYCLIC DIRECTORIES

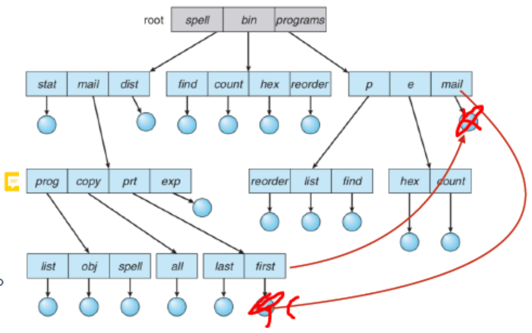
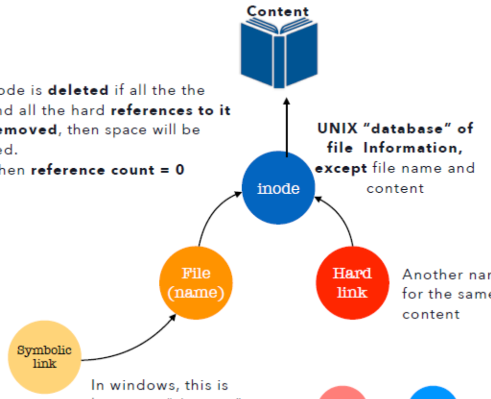


Figure 10.10 Tree-structured directory structure.

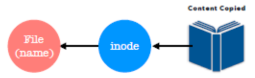


An inode is **deleted** if all the the file and all the hard **references to it are removed**, then space will be cleared.
i.e: when **reference count = 0**

UNIX "database" of file Information, **except** file name and content

Another name for the same content

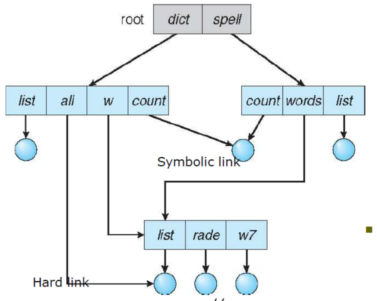
In windows, this is known as "shortcut"



Hard link != copy, a **copy** operation copies the entire content on secondary storage

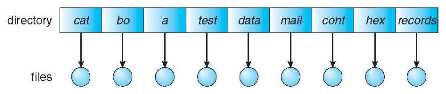
Acyclic graph directories

- Same file, multiple names (through symbolic/hard links)



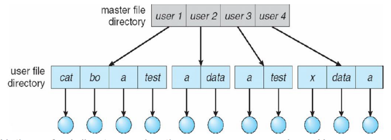
- What if we now delete /spell/count?
  - % rm /spell/count
  - Underlying file *is* removed
  - Symbolic link /dict/count remains, but becomes dangling pointer (name references non-existent file)
- What if we now delete /dict/w/list?
  - % rm /dict/w/list
  - Alternate name /dict/all keeps underlying file alive, i.e., file is *not* removed, exists under /dict/all only
  - Hard link increases reference count of file, file removed only if reference count becomes zero

- A single directory for all the users



- Name clashes between users (users Tom and Amy can't pick same name for their files!)
- No logical grouping or organization

- Separate directory for each user



- Notions of subdirectory and *path name* emerge, e.g., **/user1/a**
- Each user has own separate name space (no clashes and more efficient search)
- Limited logical grouping
- Delete semantics – what happens when you delete non-empty directory?