Operating Systems

Steven Hand

12 lectures for CST Ia

Easter Term 2000

Part II: Operating System Functions

(Handout 2 of 2)

Memory Management

- Every process needs memory for:
  - instructions ("code" or "text"),
  - static data (in program), and
  - dynamic data (heap and stack).
- If multiprogramming $\Rightarrow$ many processes in memory simultaneously.
- The memory magagement subsystem handles:
  1. Relocation
  2. Allocation
  3. Protection
  4. Sharing
  5. Logical Organisation
  6. Physical Organisation

Address Binding

Consider a set of application binaries on a queue on disk, ready for execution

- A process may be loaded anywhere in physical memory.
- The application source refers to addresses symbolically e.g. `int count;`
- These are bound to relocatable addresses by the compiler, e.g.
  `module_base = 50, count = module_base + 10`
- Something (linkage editor or loader) will bind the relocatable addresses to absolute addresses

Address binding addresses can occur:

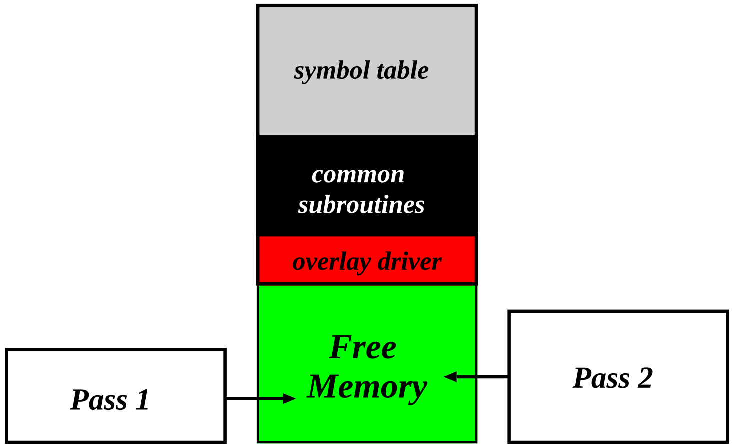
- at compile time (requires knowledge of absolute addresses)
- at load time: when program is first executed, work out position in memory and process code to insert correct addresses
- at runtime: if process is to be moved within memory during execution then memory addresses will need to be modified.

Dynamic Loading & Linking

- Relatively new appearance in OS (early 80's)
- Allows a compiled binary to invoke, at runtime, routines which are *dynamically* linked.
- Typically implemented by set of *stub routines* and a dynamic linker.
- Needs O/S support because libraries can be shared between multiple processes (and O/S is only part of system that knows where all the processes and libraries are ...).
- Benefits:
  1. Reduces size of binaries (on disk & in memory).
  2. Increased flexibility: e.g. a bug in system libraries can be fixed without requiring all third party binaries to be relinked.
  3. Also get modularity / overlay support

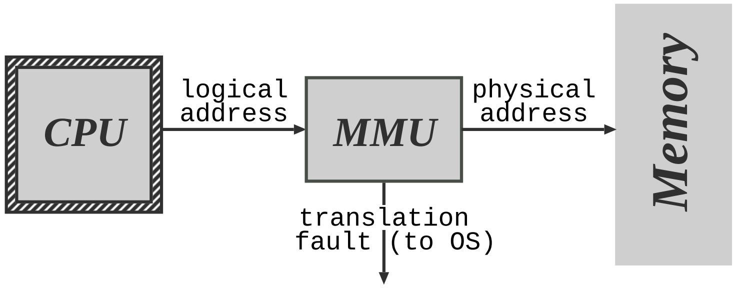
Overlays

Up to now we have assumed that the entire image will be in memory at once:



- Overlays permit a process to be bigger than physical memory size or bigger than space allocated to process.
- Only keep code and data in memory which are required at the moment.
- When other bits of code/data are needed they are loaded on top of existing code/data.

Logical vs Physical Addresses



Can avoid lots of problems by separating concept of *logical* (virtual) and *physical* ("real") addresses.

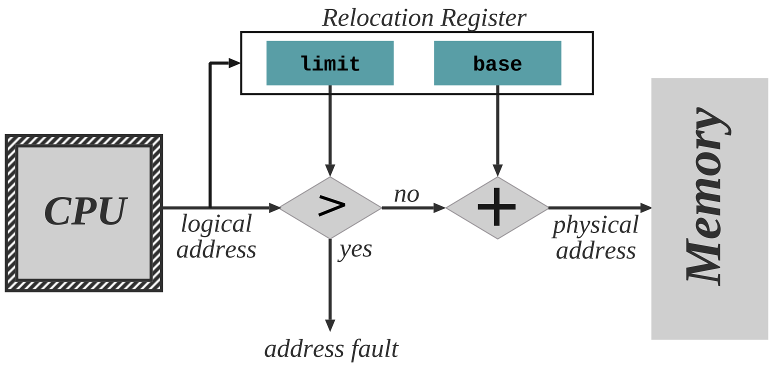
I.e. programs reference logical addresses, which are somehow translated into physical addresses.

This means address binding problem solved:

- bind to logical addresses at compile-time.
- bind to real addresses at load time/run time.

Mapping of logical to physical addresses is done at run-time by the Memory Management Unit (MMU) hardware.

E.g. Segmentation



1. Relocation register holds the value of the base address owned by the process.

2. Relocation register contents are added to each memory address before it is sent to memory.

3. E.g. DOS on 80x86 — 4 relocation registers

4. Logical address is a tuple (s, o).

5. NB: Process never sees physical address — simply manipulates logical addresses

Swapping

- If the number of processes exceeds total physical memory, then process can be swapped out to secondary store (e.g. disk).

- This makes space for a second process to enter memory

- When the original process is resumed it is swapped back into memory

- Can be used to preempt low priority tasks for high priority tasks

- When a process is rolled back in to memory then it must be positioned at same physical address if load/compile time relocation is used.

- If runtime relocation is used then need to change mapping in MMU to reflect new base and limit of process.

Swapping (2)

How it works:

1. OS maintains a ready queue of processes on disk which are ready to be executed.
2. When OS decides to run a process it calls dispatcher to check whether process is in memory.
3. If not the dispatcher may need to swap out a currently resident process and swap in the required process.

Note that:

- can cause very large context switch times $\Rightarrow$ need to make execution time long relative to swap time.
- user program needs to keep OS informed of how much memory it is using.
- What if have pending I/O on a process to be swapped?
- Standard swapping used on few systems in practice (too slow, too difficult to implement).
- Modified swapping used on Unix; starts automatically if memory use reaches a threshold.

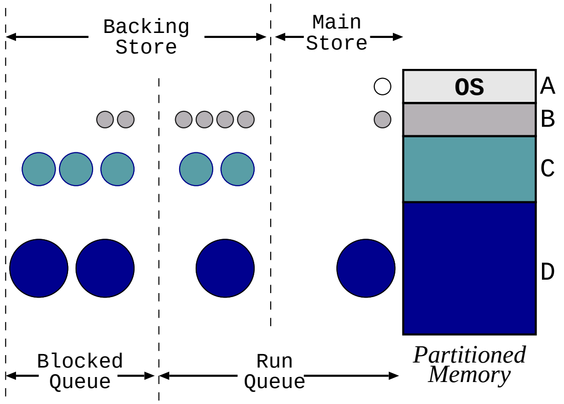
Contiguous Allocation

Given that we want multiple virtual processors, how can we support this in a single address space ?

Where do we put processes in memory ?

- OS typically in low memory due to location of interrupt vectors
- Need to protect OS and user processes from malicious programs
- Use base and limit registers in MMU — keep one context per process
- Easiest way to divide memory is into multiple fixed size partitions (e.g. OS/360 MFT).
- Each contains exactly one process
- When partition is free a new process can be loaded. When process terminates its partition becomes available to new process.

Static Multiprogramming



Static allocation:

- partition memory and allocate pieces to different job queues.
- associate jobs to a job queue and relocate (once)
- swap job to backing store when
  - blocked on I/O (assumes I/O slower than backing store).
  - time sliced: larger the job, larger the time slice
- run job from another queue while swapping jobs

Static Multiprogramming (2)

We can make the system more dynamic

- use a relocating loader
  - cost of relocation paid at each invocation
  - allocate sections of store as required
- or try for position independent code
  - tedious and inefficient on most machines
  - how would you implement position independent data?
- in either case, still suffer from
  - store *fragmentation*,
  - cannot grow a partition.

Such systems are still used; especially when the demands on the system are known in advance, e.g.

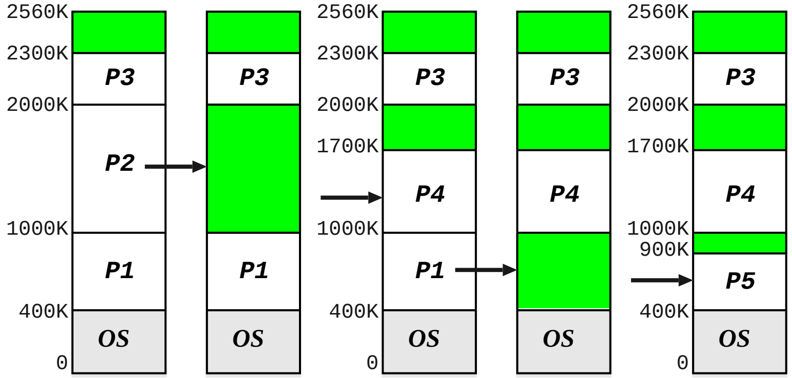
- embedded systems
- real time control systems

Dynamic Partitioning

Get more flexibility if allow partition sizes to be dynamically chosen (e.g. OS/360 MVT) :

- OS keeps table of which areas of mem are available and which occupied.
- When a new process arrives it searches for a hole large enough for the process, and updates its table.
- Free space left becomes a hole.
- At any time OS has list of available holes. When process arrives if a hole is too big it is split in two, and remainder returned to the free list.
- When process terminates it frees its memory onto the free list.

Scheduling Example

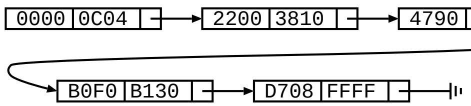


- Consider machine with total of `2560K` memory.
- Operating System requires `400K`.
- The following jobs are in the queue:

Process	Memory	Time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

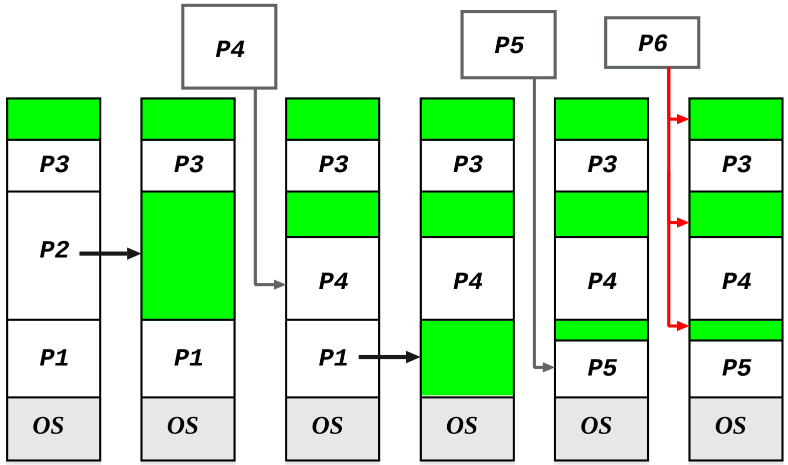
Free Space Management

- Could use a *bitmap*: one bit for each "chunk" of memory.
- Problem: finding contiguous set of n chunks.
- Instead, use one or more *linked lists*, e.g.



- Several mechanisms for determining which hole to use for new process:
  - First fit: stop searching as soon as big enough hole is found
  - Best fit: search entire list to find "best" fitting hole.
  - Worst fit: allocate largest hole (again must search entire list).
- Can use *buddy system* to make allocation faster.

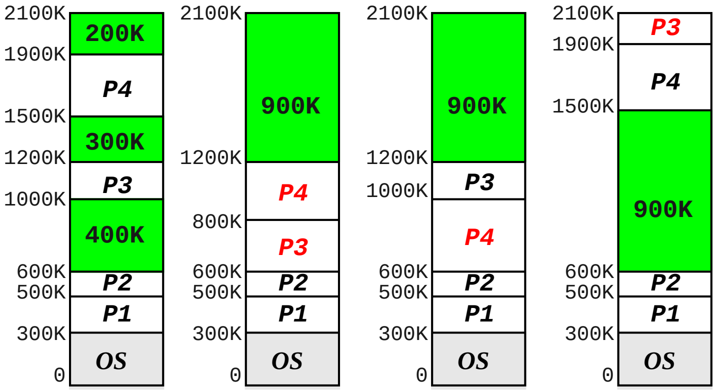
External Fragmentation



- Dynamic partitioning algorithms suffer from external fragmentation: as processes are loaded they leave little fragments which may not be used.
- **External fragmentation** exists when the total available memory is sufficient for a request, but is unusable because it is split into many holes.
- Can also have problems with tiny holes

Solution: compact holes periodically.

Compaction



Choosing optimal strategy quite tricky . . .

Note that:

- Require run-time relocation.
- Can be done more efficiently when process is moved into memory from a swap.
- Some machines used to have hardware support (e.g. CDC Cyber).

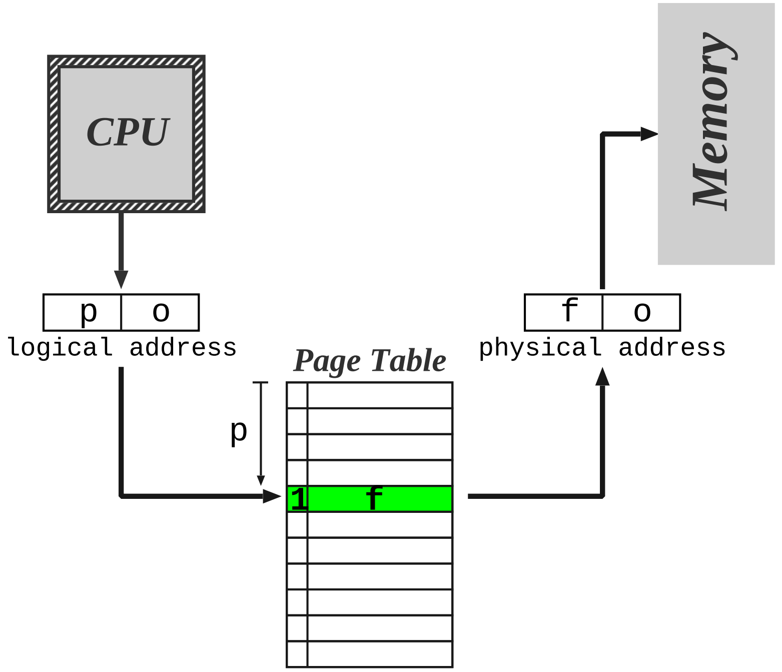
Also get fragmentation in *backing store*, but in this case compaction not really viable . . .

Paging

Another solution is to allow a process to exist in non-contiguous memory

- Divide physical memory into blocks of fixed size, called frames
- Divide logical memory into blocks of same size called pages
- Backing store also composed of blocks of same size as frames
- Now need hardware support for paging: each address generated by CPU is composed of a page number p and page offset o.
- Page number p used as index into *page table*.
- Page table contains address f of each page in physical memory

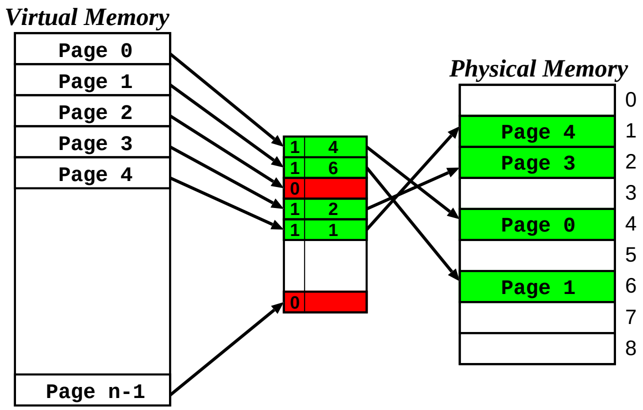
Paging (2)



- Potentially have *virtual* address space >> than physical one (i.e. can have $|\mathtt{p}| > |\mathtt{f}|$).

⇒ need *valid* bit to say if a given page is represented in physical memory.

Paging (3)



Pros and Cons:

✔ Memory allocation easier.

✘ OS must keep page table per process

✔ No external fragmentation (in physical memory).

✘ But get **internal fragmentation**.

✔ Clear separation between user and system view.

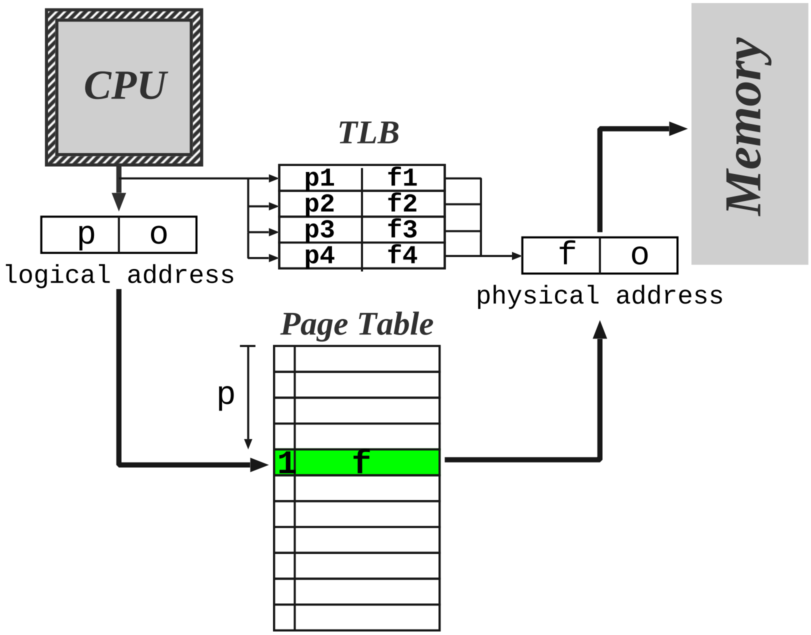
✘ Additional overhead on context switching

Structure of the Page Table

Different kinds of hardware support can be provided:

- Simplest case — page table is a set of dedicated registers
- Each memory reference goes through these so they must be fast
- OS dispatcher/scheduler loads the registers on context switch
- Fine if the page table is small ... but what if have large number of pages ?
- Alternatively keep Page Table Base Register (PTBR) per process
- Standard solution is to use a Translation Lookaside Buffer (TLB)

TLB Operation



- On memory reference present TLB with logical memory address
- If page table entry for the page is present then get an immediate result
- If not then make memory reference to page tables, and update the TLB

TLB Issues

- Updating TLB may be difficult if it is full: need to discard some reference.
- Context switch may requires TLB flush so that next process doesn't use wrong page table entries.
- Today many TLBs support process tags to improve performance.
- Hit ratio defined to be percentage of time a page entry is found in TLB
- e.g. consider TLB search time of $20ns$, memory access time of $100ns$, and a hit ratio of 80%

⇒ assuming one memory reference required for page table lookup, the *effective* memory access time is $0.8 \times 120 + 0.2 \times 220 = 140ns$.

- Increase hit ratio to 98% gives effective access time of 122ns — only a 13% improvement.

Protection Issues

- Associate protection bits with each page – kept in page tables (and TLB).
- E.g. one bit for read, one for write, one for execute.
- May also distinguish whether may only be accessed when executing in *kernel mode*, e.g.

Frame Number	K	R	W	X	V

- At the same time as address is going through page hardware, can check protection bits.
- Attempt to violate protection causes h/w trap to operating system code
- As before, have one bit specifying *valid/invalid*, i.e. determining whether the page is mapped into the process address space.
- In some older systems keep a Page Table Length Register (PTLR) to indicate size of page table

Shared Pages

Another advantage of paging is code/data sharing, e.g.

- binaries: editor, compiler etc.
- libraries: shared objects, dlls.

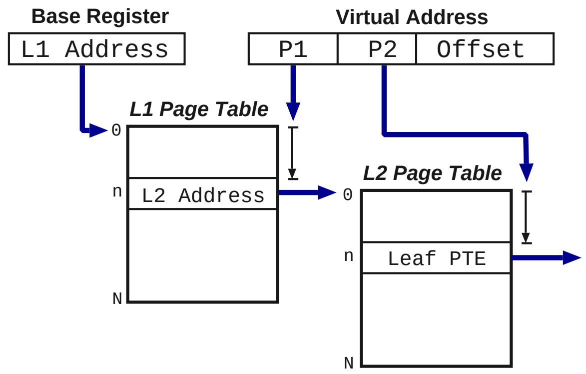
So how does this work?

- Implemented as two logical addresses which map to one physical address.
- If code is *re-entrant* (i.e. stateless, non-self modifying) it can be easily shared between users.
- Otherwise can use *copy-on-write* technique.
- (may use this for lazy data sharing too).

Requires additional book-keeping in OS, but worth it, e.g. over 20Mb of shared code on my linux box.

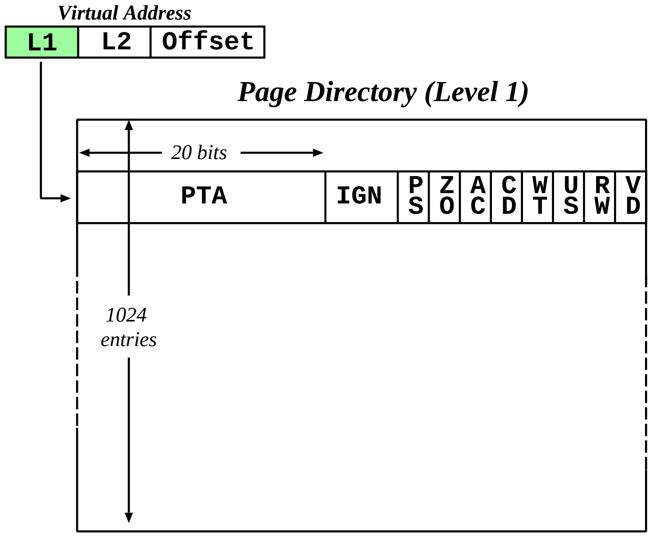
Multilevel Paging

- Most modern systems can support very large ($2^{32}, 2^{64}$) address spaces.
- Solution – split page table into several sub-parts
- Two level paging – page the page table



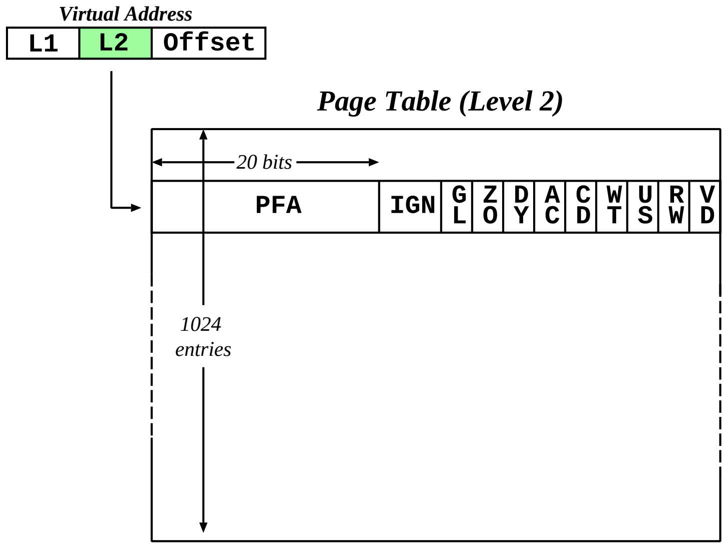
- For 64 bit architectures a two-level paging scheme is not sufficient: need further levels.
- (even some 32 bit machines have > 2 levels).

Example: x86



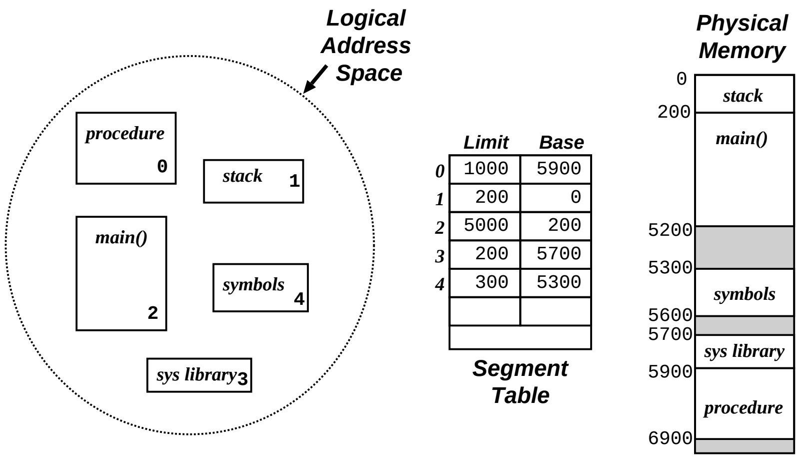
- Page size 4K (or 4Mb).
- First lookup is in the *page directory*: index using most 10 significant bits.
- Address of page directory stored in internal processor register (`cr3`).
- Results (normally) in the address of a *page table*.

Example: x86 (2)



- Use next 10 bits to index into page table.
- Once retrieve page frame address, add in the offset (i.e. the low 12 bits).
- Notice page directory and page tables are exactly one page each themselves.

Segmentation



- User prefers to view memory as a set of segments of no particular size, with no particular ordering
- Segmentation supports this user-view of memory — logical address space is a collection of (typically disjoint) segments.
- Segments have a name (or a number) and a length — addresses specify segment and offset.
- Contrast with paging where user is unaware of memory structure (all managed invisibly).

Implementation

- We need to maintain a segment table for each process:

Segment	Access	Base	Size	Others!

- If program has a very large number of segments then the table is kept in memory, pointed to by ST base register STBR
- Also need a ST length register STLR since number of segs used by different programs will differ widely
- The table is part of the process context and hence is changed on each process switch.

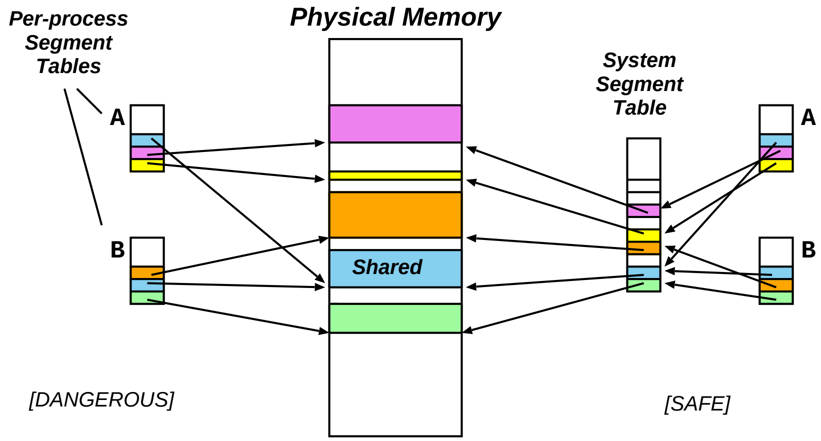
Algorithm:

1. Program presents address (s, d). Check that $s <$ STLR. If not, fault
2. Obtain table entry at reference $s+$ STBR, a tuple of form (b_s, l_s)
3. If $0 \leq d < l_s$ then this is a valid address at location (b_s, d), else fault

Protection and Sharing

- Big advantage of segmentation is that protection is per segment
- Protection bits associated with each ST entry checked in usual way
- E.g. instruction segments (should be non-self modifying!) thus protected against writes etc.
- E.g. place each array in own seg $\Rightarrow$ array limits checked by hardware
- Segmentation also facilitates sharing of code/data
  - Each process has its own STBR/STLR
  - Sharing is enabled when two processes have entries for the same physical locations.
  - For data segments can use copy-on-write as per paged case.
- Several subtle caveats exist with segmentation — e.g. jumps within shared code.

Sharing segments



Sharing segments:

- wasteful (and dangerous) to store common information on shared segment in each process segment table
- assign each segment a unique System Segment Number (SSN)
- process segment table simply maps from a Process Segment Number (PSN) to SSN

Fragmentation

- Long term scheduler must find spots in memory for all segs of a program
- Problem now in that segs are of variable size — leads to fragmentation
- Tradeoff between compaction/delay depends on average segment size
- Extremes: each process 1 seg — reduces to variable sized partitions
- Or each byte one seg separately relocated — quadruples memory use!
- Fixed size small segments $\equiv$ paging!
- In general with small average segment sizes, external fragmentation is small.

Summary

Important considerations when comparing different memory management schemes

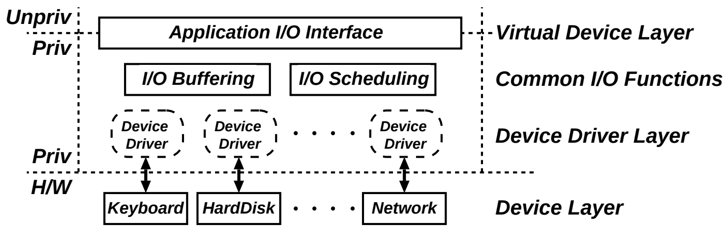
- Hardware support
- Performance
- Fragmentation
- Relocation
- Swapping
- Sharing
- Protection

I/O Hardware

- Wide variety of 'devices' which interact with the computer via I/O, e.g.
  - Human readable: graphical displays, keyboard, mouse, printers
  - Machine readable: disks, tapes, CD, sensors
  - Communications: line drivers, modems, network interfaces
- They differ significantly from one another w.r.t.
  - Data rate
  - Complexity of control
  - Unit of transfer
  - Data representation
  - Error handling

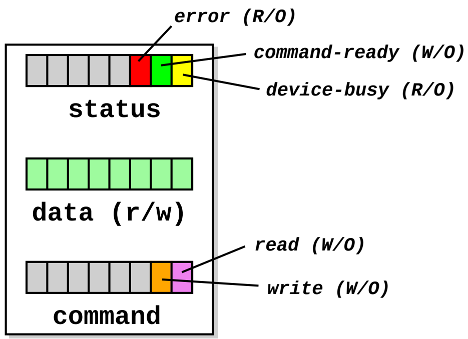
⇒ I/O subsystem is the messiest part of OS.

I/O Subsystem



- Programs access *virtual devices*:
  - terminal streams not terminals
  - windows not frame buffer
  - event stream not raw mouse
  - files not disk blocks
  - printer spooler not parallel port
  - transport protocols not raw Ethernet
- OS deals with processor–device interface:
  - I/O instructions v. memory mapped (where?)
  - I/O hardware type (e.g. 10's of serial chips)
  - polled v. interrupt driven
  - processor interrupt mechanism

Polled Mode I/O



- Consider a simple device with three registers: `status`, `data` and `command`.
- (Host can read and write these via bus)
- Polled mode operation works as follows:
  1. Host repeatedly reads `device_busy` until clear.
  2. Host sets e.g. `write` bit in `command` register, and puts data into `data` register.
  3. Host sets `command_ready` bit in `status` register.
  4. Device sees `command_ready` and sets `device_busy`
  5. Device performs write operation.
  6. Device clears `command_ready` & then `device_busy`.

Interrupts

Polling poor $\Rightarrow$ most OSs use interrupts.

Most modern processors provide at least a basic interrupt mechanism:

- at end of each instruction, check interrupt line(s) for pending interrupt
- save program counter
- save processor status
- change processor mode
- jump to well known address (or its contents)

Some processors provide:

- multiple levels of interrupts
- hardware vectoring of interrupts
- mode dependent registers

More Interrupts

Interrupt handler maps from h/w interrupts to ISR invocations. Handler may need to:

- save more registers
- demultiplex interrupt in software
- establish a language environment (e.g. a C run time stack)

Interrupt Service Routines (ISRs):

1. device, not processor, specific (unless asm!)
2. for programmed I/O device:
    - transfer data
    - clear interrupt (sometimes a side effect of transfer)
3. for DMA device:
    - acknowledge transfer
4. request another transfer if any more I/O requests pending on device
5. signal any waiting processes
6. enter scheduler or return

Question: who is scheduling who?

Device Classes

Completely homogenising device API almost impossible
⇒ OS generally splits devices into four generic classes:

1. Block devices (e.g. disk drives, CD):
    - Commands include `read`, `write`, `seek`
    - Raw I/O or file-system access
2. Character devices (e.g. keyboards, mice, serial):
    - Commands include `get`, `put`
    - Libraries layered on top to allow line editing
3. Network Devices
    - Varying enough from block and character to have own interface
    - Unix and Windows/NT use *socket* interface
4. Miscellaneous (e.g. clocks and timers)
    - Provide current time, elapsed time, timer
    - `ioctl` (on UNIX) covers odd aspects of I/O such as clocks and timers.

I/O Buffering

- Buffering: OS stores (a copy of) data in memory while transferring between devices
  - to cope with device speed mismatch
  - to cope with device transfer size mismatch
  - to maintain "copy semantics"
- OS can use various kinds of buffering:
  1. Single buffering — OS assigns a system buffer to the user request
  2. Double buffering — process consumes from one buffer while system fills the next
  3. Circular buffers — most useful for burst-oriented I/O
- Many aspects of buffering dictated by device type:
  - character devices $\Rightarrow$ line probably sufficient.
  - network devices $\Rightarrow$ bursty (time & space).
  - block devices $\Rightarrow$ lots of fixed size transfers.
  - (last usually major user of buffer memory)

Blocking v. Nonblocking I/O

From programmer's point of view, I/O system calls exhibit one of three kinds of behaviour:

1. Blocking: process suspended until I/O completed
   - Easy to use and understand
   - Insufficient for some needs
2. Nonblocking: I/O call returns as much as available
   - User interface, data copy (buffered I/O)
   - Returns almost immediately with count of bytes read or written.
3. Asynchronous: process runs while I/O executes
   - I/O subsystem explicitly signals process when its I/O request has completed
   - Most flexible (and potentially efficient)
   - ... but also most difficult to use

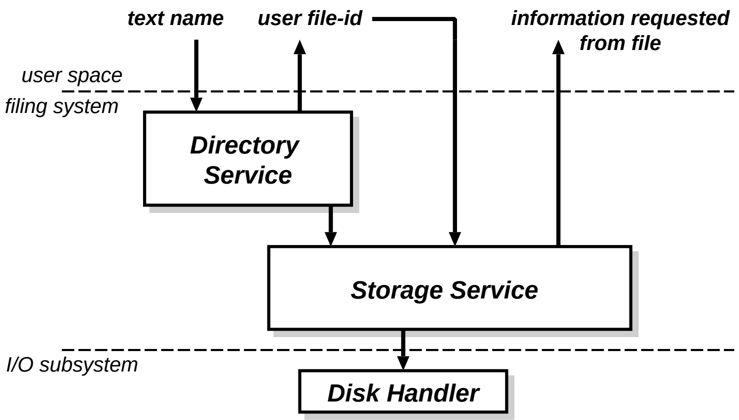
Other I/O Issues

- Caching: fast memory holding copy of data
  - can work with both reads and writes
  - key to I/O performance
- Scheduling:
  - e.g. ordering I/O requests via per-device queue
  - some OSs try fairness ...
- Spooling: queue output for a device
  - useful if device is "single user" (i.e. can serve only one request at a time)
  - e.g. printing
- Device reservation:
  - system calls for acquiring or releasing exclusive access to a device
- Error handling:
  - e.g. recover from disk read, device unavailable, transient write failures, etc.
  - most I/O system calls return an error number or code when an I/O request fails
  - system error logs hold problem reports.

Performance

- I/O a major factor in system performance
  - Demands CPU to execute device driver, kernel I/O code, etc.
  - Context switches due to interrupts
  - Data copying
  - Network traffic especially stressful.
- Improving performance:
  - Reduce number of context switches
  - Reduce data copying
  - Reduce # interrupts by using large transfers, smart controllers, polling
  - Use DMA
  - Balance CPU, memory, bus and I/O performance for highest throughput.

File Management



Filing systems have two main components:

1. Directory Service
   - maps from names to file identifiers.
   - handles access & existence control
2. Storage Service
   - provides mechanism to store data on disk
   - includes means to implement directory service

File Concept

What is a file?

- Basic abstraction for non-volatile storage.
- Typically comprises a single contiguous logical address space.
- Internal structure:
  1. None (e.g. sequence of words, bytes)
  2. Simple record structures
     – lines
     – fixed length
     – variable length
  3. Complex structures
     – formatted document
     – relocatable object file
- Can simulate last two with first method by inserting appropriate control characters.
- All a question of who decides:
  – operating system
  – program(mer).

Naming Files

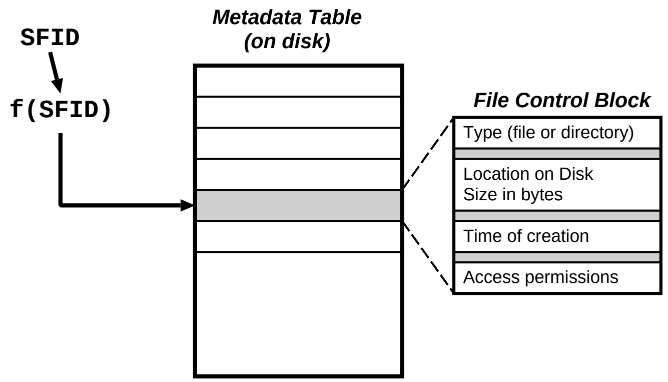
Files usually have at least two kinds of 'name':

1. System file identifier (SFID):
   - (typically) a unique integer value associated with a given file
   - SFIDs are the names used within the filing system itself
2. "Human" name, e.g. `hello.java`
   - What users like to use
   - Mapping from human name to SFID is held in a *directory*, e.g.

Name	SFID
hello.java	12353
Makefile	23812
README	9742

   - Directories also non-volatile $\Rightarrow$ must be stored on disk along with files.
3. Frequently also get user file identifier (UFID).
   - used to identify *open* files (see later)

File Meta-data



In addition to their contents and their name(s), files typically have a number of other attributes, e.g.

- *Location*: pointer to file location on device
- *Size*: current file size
- *Type*: needed if system supports different types
- *Protection*: controls who can read, write, etc.
- *Time*, *date*, and *user identification*: data for protection, security and usage monitoring.

Together this information is called *meta-data*.
It is contained in a *file control block*.

Directory Name Space (I)

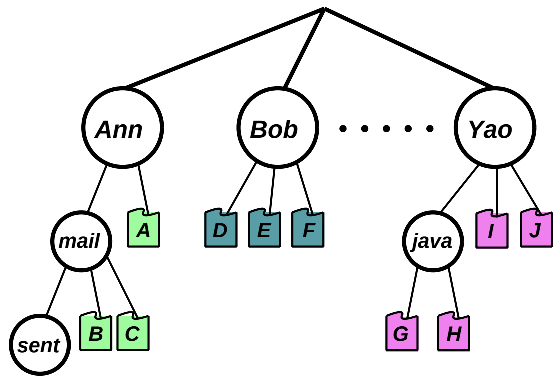
What are the requirements for our name space?

- Efficiency: locating a file quickly.
- Naming: user convenience
  - allow two (or more generally N) users to have the same name for different files
  - allow one file have several different names
- Grouping: logical grouping of files by properties (e.g. all Java programs, all games, ...)

First attempts:

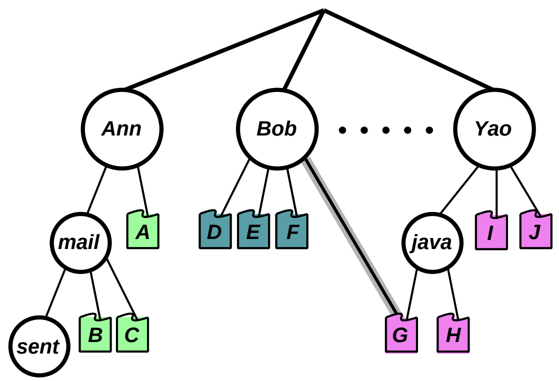
- Single-level: one directory shared between all users
  - ⇒ naming problem
  - ⇒ grouping problem
- Two-level directory: one directory per user
  - access via *pathname* (e.g. `bob:hello.java`)
  - can have same filename for different user
  - but still no grouping capability.

Directory Name Space (II)



- Get more flexibility with a general *hierarchy*.
  - directories hold files or [further] directories
  - create/delete files relative to a given directory
- Human name is full path name, but can get long: e.g. /usr/groups/X11R5/src/mit/server/os/4.2bsd/utils.c
  - offer relative naming
  - login directory
  - current working directory
- What does it mean to delete a [sub]-directory?

Directory Name Space (III)

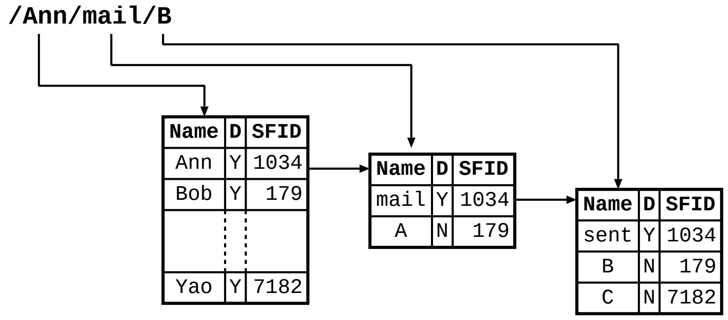


- Hierarchy good, but still only one name per file.

⇒ Extend to directed acyclic graph (DAG) structure:
  - allow shared subdirectories and files.
  - can have multiple *aliases* for the same thing

- Problem: dangling references
- Solutions:
  - Back-references (but variable size records)
  - Reference counts.
- Problem: cycles . . .

Directory Implementation



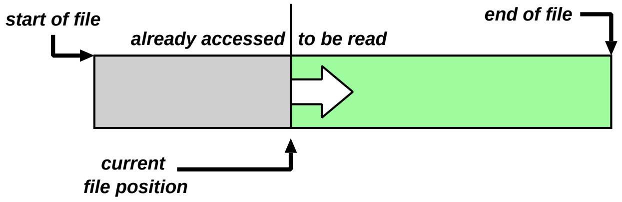
- Directories are non-volatile ⇒ store as "files" on disk, each with own SFID.
- Must be different *types* of file (for traversal)
- Explicit directory operations include:
  - create directory
  - delete directory
  - list contents
  - select current working directory
  - insert an entry for a file (a "link")

File Operations (I)

UFID	SFID	File Control Block (Copy)
1	23421	location on disk, size,...
2	3250	" "
3	10532	" "
4	7122	" "
⋮	⋮	

- Opening a file: `UFID = open(<pathname>)`
  1. directory service recursively searches directories for components of `<pathname>`
  2. if all goes well, eventually get `SFID` of file.
  3. copy file control block into memory.
  4. create new `UFID` and return to caller.
- Creating a new file: `UFID = create(<pathname>)`
- Once have `UFID` can read, write, etc.
  - various modes (see next slide)
- Closing a file: `close(UFID)`
  1. copy [new] file control block back to disk.
  2. invalidate `UFID`

File Operations (II)



- Associate a *cursor* or *file position* with each open file (viz. UFID), initialised to start of file.
- Basic operations: *read next* or *write next*, e.g.
  - `read(UFID, buf, nbytes)`, or
  - `read(UFID, buf, nrecords)`
- Sequential Access: above, plus `rewind(UFID)`.
- Direct Access: *read* N or *write* N
  - allow "random" access to any part of file.
  - can implement with `seek(UFID, pos)`
- Other forms of data access possible ...

Access Control

- File owner/creator should be able to control:
  - what can be done, and
  - by whom
- Access control is normally a function of the directory service ⇒ checks done at file *open* time
- (nothing to do with access modes on prev. slide)
- Various types of access, e.g.
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List
- More advanced schemes possible.

Existence Control

What happens when user deletes a file?

- keep file in existence while there is a valid pathname referencing it
- plus check entire FS periodically for garbage

May also be a factor when a file is renamed/moved.

Concurrency Control

Some form of *locking* to handle simultaneous access

- may be mandatory or advisory
- locks may be shared or exclusive
- granularity may be file or subset

More on locking, etc. in CST IB ...