Device Drivers and Memory Management

Linux Device Drivers

User side

Device drivers have a special file in `/dev` directory and have 5 system calls

- `Open` : Make device available
- `read` : Read from device
- `write` : Write to device
- `ioctl` : Input Output Control (Optional)
- `close` : Make device unavailable

Kernel side

Each file has functions associated with it and are called when corresponding system calls made
`linux/fs.h` shows all ops on files
Device drivers implement at least `open` , `read` , `write` , `close`

Categorisation

Kernel tracks:
Physical Dependencies between devices (E.g. devices on USB hub)
Buses Channels between processor and devices. Physical or logical
Classes Sets of same type devices

Handling interrupts

Cycle for interrupt handling:

- Device sends interrupt
- CPU selects interrupt handler
- Handler processes interrupt to do two tasks:
  - Data transferred between device
  - Wake up processes waiting for transfer to finish
- Handler clears interrupt bit of device for next interrupt

Interrupt processing must be as short as possible
Data transfer fast but processing slow
Interrupt processing has 2 halves:
Top half called directly by handler and transfers data between device and kernel buffer and schedules software interrupt to start bottom half
Bottom half runs in interrupt context and does rest of processing (eg working thru protocol stack, wake up processes)

Memory Management

Memory is a limited resource and memory hunger of applications increases with more memory
Requires sophisticated algorithms, with support from HW, compiler and loader

Both programs (logical) and HW (physical) view memory the same, a set of memory cells starting at 0x0 and ending at some value. Problem occurs when we want to store multiple programs simultaneously, all who views memory as starting at 0x0. This requires a *mapping between logical and physical addresses*

This mapping can occur at different times:
Compile time
absolute references generated
Load time
mapping done by *special program (loader)*
Execution time
HW support

Address mapping can take a step further with dynamic linking, allowing a *single copy of a system library to be shared among other programs*. Requires OS support for code to be accessible to multiple processes

Swapping

If memory demand too high, memory for some processes transferred to disk
With *scheduling*, low priority processes are swapped out

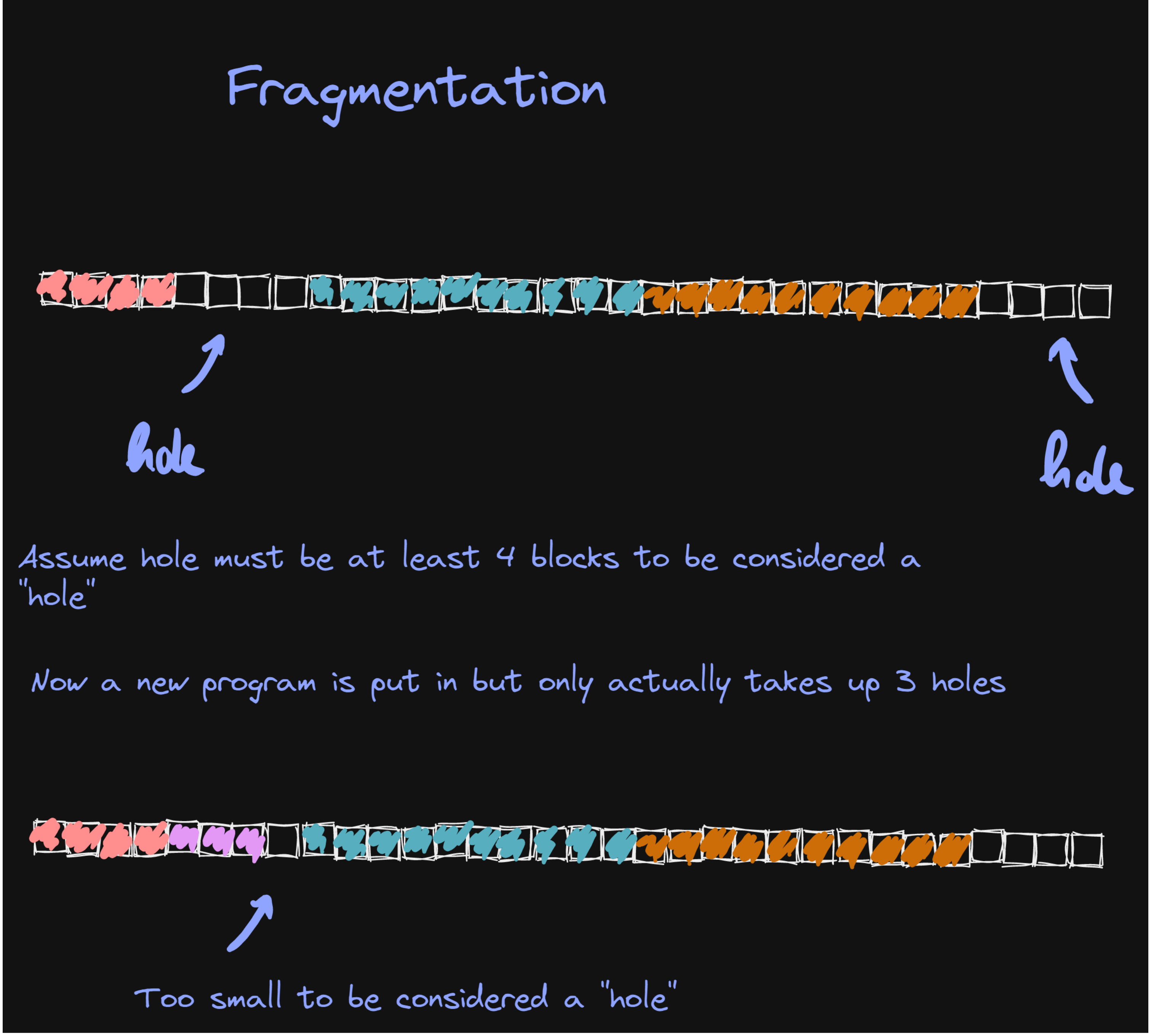
Problems:
Long transfer times
Pending IO?

Therefore, *swapping is not a principle memory management technique*, meaning is not a solution for management

Fragmentation

Swapping creates problems:

- Over time, small holes appear in memory (*external frag*)
- Programs smaller than those holes do not fill them and leftover space too small to be a hole (*internal frag*)



Strategies for choosing holes:
First fit Start from beginning and use first available hole
Rotating first fit start after last assigned part of memory
Best fit find smallest usable hole
Buddy system Free holes admistered according to tree structure; smallest chunk used

Paging

Approach:
Assign memory of fixed size (*page*) which avoids *external frag*
Page table translates logical to physical addresses

HW support mandatory:
If page table small, use *fast registers*. Store large atbles in main mem but *cache entries*. This is the general principle

Memory protection easily added as protection info stored in table

Segmentation

Divide memory according to usage:
Data mutable, different foreach instance
Program Code immutable, same foreach instance
Symbol Table immutable, same foreach instance (*debugging*)

Requires HW support:
Same as *paging* but overflow check needed

Paging for ease of allocation, *segmentation* for memory use so most systems use both

Virtual memory

Completely separates logical and physical memory and programs to use a lot of memory
Works as most programs don't use that much
Efficient implementation tricky as difference between memory and disk access immense

Demand Paging

Virtual memory implemented as *demand paging*: memory divided into *pages of same length*, with *validation bit*

Decisions to make:
-Which processes to *swap out* which moves memory to disk and blocks process (swapper)
-*Which pages to move to disk* when new page required (pager)

Page fault rate must be minimised (page has to be fetched from disk) Page Fault handling

Page replacement algos

FIFO
✔
-*easy to implement*
✘
-*locality not considered*
-*Belady's anomaly*: Increase in number of frames increases number of page faults

NOT EXPLAINED BUT PAGE FRAMES = PHYSICAL MEMORY, PAGE = VIRTUAL MEMORY

Optimal algorithm
Replace pages which won't be used or used last
✔
-*Nice to compare*
✘
-*Unimplementable*

Least recently used
Replace page which has been unused the longest

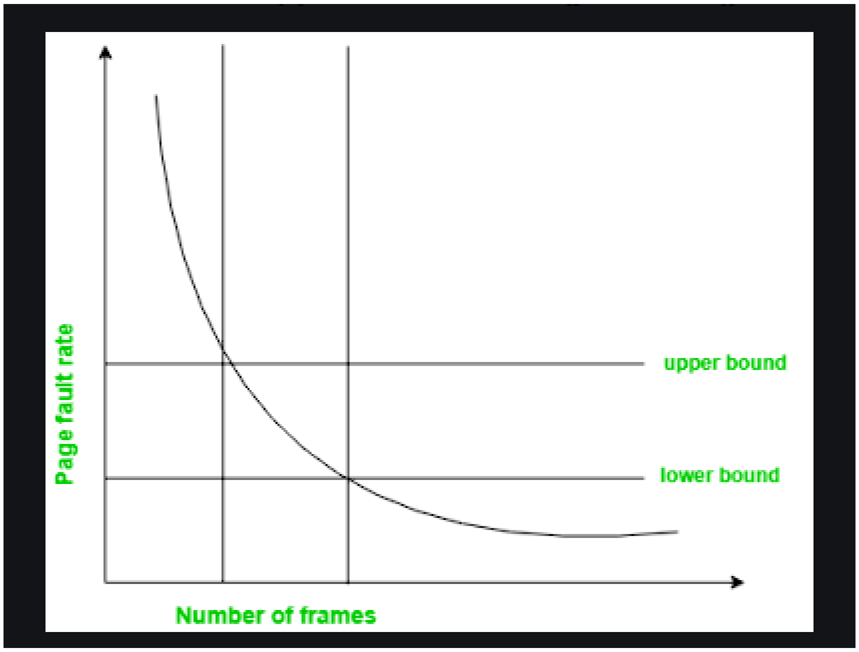
✅
-*Likely to have low page fult rate*
❌
-*Requires a lot of HW support*

Possible HW solutions:
Use a stack in "microcode" (stack within the MMU or HW itself)
Reference bit approximation: HW sets bits to 1 when page referenced, FIFO through pages and use first page with bit=0 and reset pages with bit=1 to 0 as you go along ***second chance algo***

Thrashing

If a process lacks frames it uses constantly, page fault rate high
CPU throughput decreases thus negatively affects performance

Solutions:
Working set model (locality based)
A *Locality* is a set of pages used together, locality model assumes programs execute move from locality to locality.
Define *working set* as pages used in most recent "X" page references and keep these in memory
Achieves high CPU utilisation and prevents thrashing if X ≥ current locality size
Page-Fault Frequency
-Give process more frames if page frequency rate high
-remove frame from process if fault rate low



Memory management in linux kernel

4 Segments:
Kernel code
Kernel data
User code
User data
Paging used with permission system

Kernel and user memory

Separate *logical* addresses for kernel and user memory
For 32bit arch
kernel space is upper 1 GB
user space is lower 3 GB
kernel memory always mapped but protected from user process access
For 64 bit arch
kernel space is upper half
user space is lower half

Page caches (Prolly won't be tested)

Often, there are repeated cycles of alloc and free of similar objects (inodes, dentries)

Pool of pages used as a cache ***slab cache*** maintained by application (eg file system)

`kmalloc` uses slab cache for commonly used sizes

Summary

Device Drivers
implement `open`, `read`, `write`, `close`, with common structure
Mem management
-Serious effort to manage limited resource
-Isolate mem for each process
-If mem damand high, swap out parts of process memory
-*Paging* and *segmentation*
-Requires HW support