How To C

- **Struct & Function Pointers**:

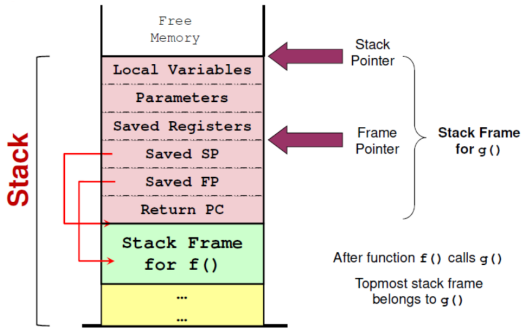
```
typedef struct NODE{
  int data;
  struct NODE* next;
} node;
node* newNode = malloc(sizeof(node));
newNode->data = 5;
//(*newNode).data = 5;

void f(int x);
void (*fptr) (int x); //Means the same.
*ftpr = f;
typedef void (*funcPtrType) (int);
funcPtrType fp1, fp2; //void (*fp1)(int x);
```

Motivation for OS & Definitions

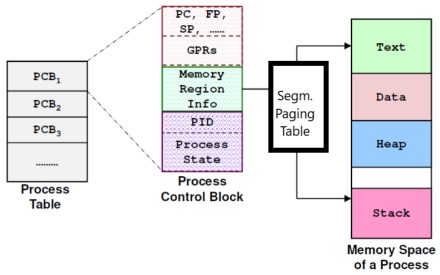
- **Concurrency**: Allows more users / program to run on the same system at once. Illusion of concurrency - each program thinks they are the only one.
- **Abstraction**: Hide low level details e.g. different hard disk capacities / speed / RPM / manufacturer, provide a seamless API and portability of code.
- **Ensure fair resource allocation**: OS manages all resources. (CPU, Mem, I/O)
- **Control**: OS ensures security and protection from rogue programs.
- **Kernel mode**: Have complete access to all resources. OS runs in kernel mode.
- **User mode**: Limited access to resources.
- **Monolithic OS**: Kernel is one big program. Easy to code, easy to pass information around. However, highly coupled components, complicated structure, and if one crash, all crash.
- **Microkernel**: Kernel is very small, only provides basic IPC / address management. Other features run as "services", use IPC to communicate. More robust, but lower performance as multiple calls required.
- **Virtual Machine**: Software emulation of hardware, illusion of complete hardware to lvl abv.
- **Hypervisors**: Type 1: serves like the OS (e.g. Xen). Type 2: Runs on top of host OS (e.g. VMWare).

Memory

- **Stack Frame**:

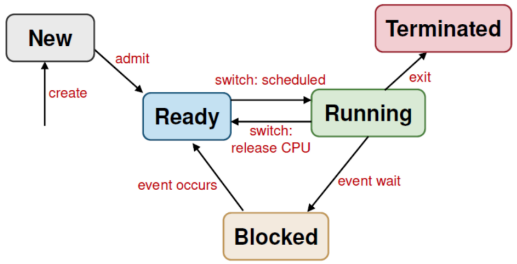


- **Stack Frame Setup**: Execute function call.
  1. Caller: Pass arguments using registers and/or put on stack
  2. Caller: Save return PC on stack.
  [Transfer control to callee]
  3. Callee: (Register spilling) Save old values of registers that I need to use. Save old FP, SP
  4. Callee: Allocate space for local variables / return values. Adjust FP. Adjust SP.
- **Heap**: Dynamically allocated memory. Does not go out of scope when function terminates unlike stack. Allocated with malloc() / new.
- **Execution context of a process**: (Pointer to) Memory: Text (Instructions), Data, Stack, Heap. Hardware: (Value of) Registers, PC, SP, FP. OS: Process ID, Process State.
- **Making system call**: Library (e.g. getpid()) internally sets system call number usually in a register, then calls a TRAP instruction. Kernel will handle with a "dispatcher", carry out the request and return result.
- **Exception**: Like a forced system call. Have to execute an exception handler. Synchronous / because of program fault.
- **Interrupt**: Interrupt execution of program, execute interrupt handler. Async / because of external event eg mouse / timer.



Process

- **Five-stage process model**:



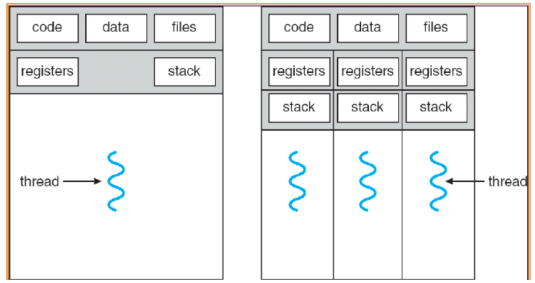
- **Process Control Block**: Contains the entire execution context of a process.
- **Process Table**: An "array" of PCBs for all processes.
- **Fork**: Creates an exact copy of a child at the particular instance. All context in PCB, SP/FP/PC/Registers etc are copied (on write). Even **input buffer**!
  fork() gives the pid of the child, or 0 if you're the parent.
  wait() can be used to wait for any child spawned by the fork. Syntax: int pid = wait(&result);
  actual return status is result >> 8. Can also do waitpid(ID, &status, NULL).
  execl() and variants replace current process with new exec. Returns -1 if error, and continues with program execution, often causing fork bomb!
  Syntax: int result = execl(path, arg0, arg1, ..., NULL);
- **Fork pseudo-implementation**:
  1. Create address space for child allocate new PID. Create entry in PCB Table, init context (e.g. set CPU time = 0), update parent with child PID.
  2. Copy all data (on write) from parent, incl text, data, heap, stack, open file ptrs etc.
  3. Initialise hardware registers etc for child.
  4. Place child on ready scheduler queue.
- **Zombie processes**: When child terminates, data in PCB cannot be deleted, as parent might call wait. So process becomes zombie, freeing most resources (e.g. memory/file ptr), but keeping essential PID/status/CPU time. If parent forgot to wait() on terminated child, child become zombie, cannot remove until parent terminates. If parent has already exited before child return, child becomes "orphan", init() becomes pseudo parent.

Scheduling

- **Context Switch**: To run multiple processes on the same CPU, OS needs to context switch - save running process context, replace registers/etc with context of incoming process, then switch process states. This incurs overhead.
- **Scheduling Problem**: Determining which process gets to run.
- **Criteria for all schedulers**:
  1. Fairness: each should get a fair share of CPU time. No starvation. 2. Balance: All parts of the computing system should be fully utilized (e.g. CPU + IO)
- **Scheduling Policy**: When is scheduling triggered. Non-preemptive: a process stays scheduled/running until it gives up CPU / is blocked. Preemptive: Process is given fixed time quota to run, kicked out after time quota. Possible to give up CPU before time is up.
- **Environment**:
  1. Batch Processing: No user and no interaction. No need to be responsive.
  2. Interactive: User interacting. Should be generally responsive.
  3. Real-time: Have deadline to meet, e.g. FPS games.
- **Batch Processing**: No need pre-emptive.
  Criteria:
  1. Total time: total time to finish all tasks.
  2. Throughput: Number of tasks finished per unit time.
  3. CPU utilisation: percent time when CPU is working.
- **First Come First Serve**: Queue Data Struct, run until task is done or blocked, then next task is run. No starvation as cannot cut queue. But average total waiting time is high if all processes are waiting for one to run finish. Also has convoy effect - all stuck for CPU then all stuck for IO.
- **Shortest Job First**: Not pre-emptive. Select task with shortest CPU time first. Minimises avg waiting time, but need to know CPU time in advance, and might cause starvation for longer jobs. Can predict CPU time using previous CPU-Bound phases:
  $Predict_{n+1} = \alpha Actual_n + (1-\alpha)Predict_n$, with α being the weight placed on past history.
- **Shortest remaining time**: Pre-emptive version of SJF. Kicks out process if a new process appears with a shorter remaining time. Provides good service for short job even if it arrives late. But can cause starvation.
- **Interactive Processing**: Scheduler runs periodically. Scheduler can take over CPU through hardware timer interrupt that invokes the scheduler.
  1. Response time: time between request and system response 2. Predictability: less variation in response time.
- **Time Quantum**: Execution block given to process. No preempt during this time. Shorter TQ = more responsive, but higher overhead.
- **Round Robin**: Similar to FCFS, but pre-emptive after process utilises Time Quantum. Response time guaranteed.
- **Priority Scheduling**: Assign priority to tasks, Pre-emptive version: high priority processes immediately kick out running processes with lower priority. Non pre-emptive: high priority processes wait till TQ is up. However, low priority processes can starve. Also possible to deadlock if low priority process locks resource, but is pre-empted by another process wanting the same resource.
- **Priority Scheduling: Priority Inversion**: Allow lower priority task to pre-empt higher priority task.
- **Multi-level Feedback Queue**: Adaptive. Pre-emptive but does not pre-empt until allocated time quantum is up! New job = highest priority. If use full Time Quantum: drop priority. Otherwise, priority retained. CPU bound algorithms will sink to low priority = more responsive. Lower priority can even get longer Time Quantum to be less interrupted. Tasks with same priority runs in RR.
- **Lottery Scheduling**: Lottery tickets to decide who runs. Higher priority given more tickets. Responsive, good level of control, fairness, and simple implementation.

Thread

- **Threads**: A lightweight alternative to process, don't need to copy so much data. Don't need IPC to communicate, all threads are part of the same process → can access largely the same memory context.
  Shared: text, data, heap, and OS context (PID/files). Context switch is cheaper because don't need to switch so much data.
  Not shared/unique: threadID, hardware context (registers, PC/SP/FP), stack. Stack cannot be shared due to different function call.
  Context switch involves changing registers and PC/SP/FP only.



- **exit**: In linux, when one thread exists, all exit.
- **fork**: In linux, when one thread forks, the new process only has one thread (the one that calls fork), every other thread is killed.
- **User Thread**: User library, kernel not aware. Can run multithread on any OS, and operations are internal library calls, not system calls → fast. More configurable. However, fatal flaw: OS is not aware → scheduling at a process level. If one thread blocks, all blocks. Can only run on one CPU at a time → useless!
- **Kernel Thread**: Implemented by OS. Thread is system call. Advantage: Multiple threads on multiple CPUs. Disadvantage: Slower system call, less flexible and OS dependent.
- **Hybrid Thread**: OS schedules kernel threads, user can bind user threads to kernel threads. Many-to-many, don't need one-to-one. Offers flexibility: can limit concurrency or increase concurrency as required.
- **Simultaneous multi-threading**: multiple sets of registers allowing multiple threads to run in parallel on the same CPU core due to pipelining. (Intel Hyperthread).
- **Thread safety**: If threads share resources (e.g. global variables), a case could arise where two threads write simultaneously and override each other's changes.
- **pthread_create**: pthread_create(pthread_t* threadID, threadAttributes, void* (*fnPtr)(void*), void* args);
  fnPtr and args needs to be typecasted to void*.
- **pthread_exit**: pthread_exit(void* exitValue); //Always ensure that if you're passing back anything, it exists on the heap!
- **pthread_join**: int success = pthread_join(pthread_t threadID, void **status); Pass in a pointer to the pointer for status. This way the pointer actually gets modified and points to the correct return result. Hence void**.

IPC

- **Shared Memory**: Declare shared memory with shmget, attach with shmat, and pass information around. Can do busy waiting by sleeping until shm[0] = correct value.
- **Message Passing**: Use system call to send message (like Distributed CodeJam). Can be sync or async. Advantages: Easy to implement, portable, easy to keep synchronised.
  Disadvantages: Requires OS intervention + overhead, fixed size/format, hard to use. Can be direct communication (explicitly name the other party) or indirect (send/receive from a mailbox/port, allows multiway communication). Can be blocking (send() waits for message to be delivered, wait() waits for message arrival) or non-blocking (asynchronous).
- **Unix Pipes**: With the pipe() command in C. FIFO queue with buffer. Write wait when pipe is full, Read wait when pipe is empty. int pipeFd[2]; pipe(pipeFd); Read is index 0, write is index 1. dup(oldFd) allocates the lowest avail fd to be an alias of oldFd. dup2(oldFd, newFd) asigns newFd to alias oldFd. By default, 0 = stdin, 1 = stdout, 2 = stderr.
- **Signal**: Use kill(pid, status) to send signal. Need not be a kill signal. Use signal(signalNum, handler) to handle. Handler signature: void handler(int signalNum);

Synchronization

- **Critical Section**: Only one process can enter critical section at any time. Principles:
  1. Mutual Exclusion: If a process is executing, no one else can execute. 2. Progress: If no one in critical section, one of the waiting processes should be granted access. 3. Bounded Wait: After requesting for excess, ∃ a limit on how many times other processes can enter. 4. Independence: Processes not inside the critical section should not block others.
- **Bad scenarios**: Deadlock: all processes blocked. Livelock: all processes keep changing state to avoid deadlock but no progress. Starvation: some processes blocked forever.
- **Test and Set**: A hardware level atomic instruction. Reads value at memory addr, and sets memory addr value to 1 to indicate locked.
- **Peterson's Algorithm**: For 2 processes only, not general. Busy waiting.

```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
       Turn == 1);
Critical Section
Want[0] = 0;
```
Process P0

```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
       Turn == 0);
Critical Section
Want[1] = 0;
```
Process P1

- **Semaphore**: A high level wait/queue mechanism. Blocks wait() when S ≤ 0. Invariant: $S_{current} = S_{initial} + \#signal(S) - \#wait(S)$.
- **Producer-Consumer problem**: Multiple producers and multiple consumers.
  Producer: wait(notFull); wait(mutex); produce(); signal(mutex); signal(notEmpty);
  Consumer: wait(notEmpty); wait(mutex); consume(); signal(mutex); signal(notFull);
  Initial values: mutex = S(1). notFull = S(K). notEmpty = S(0).
- **Reader-writer problem**: Writers must have exclusive access, readers can access with other readers. This is the OP version.

```
while (TRUE) {
    wait(wantToWrite);
    wait( roomEmpty );
    Modifies data
    signal( roomEmpty );
    signal(wantToWrite);
}
```
Initial Values:
- roomEmpty = S(1)
- mutex = S(1)
- nReader = 0
- wantToWrite = 1

```
while (TRUE) {
    wait(wantToWrite); signal(wantToWrite);
    wait( mutex );
    nReader++;
    if (nReader == 1)
        wait( roomEmpty );
    signal( mutex );

    Reads data
    wait( mutex );
    nReader--;
    if (nReader == 0)
        signal( roomEmpty );
    signal( mutex );
}
```

- **Dining Philosopher's problem**:

```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)

#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
        Think( );
        takeChpStcks( i );
        Eat( );
        putChpStcks( i );
    }
}
```
[CS2106 L6 - AY181951]

```
void takeChpStcks( i )
{ wait( mutex );
  state[i] = HUNGRY;
  safeToEat( i );
  signal( mutex );
  wait( s[i] );}

void safeToEat( i )
{ if( (state[i] == HUNGRY) &&
      (state[LEFT] != EATING) &&
      (state[RIGHT] != EATING) ) {
    state[ i ] = EATING;
    signal( s[i] );
  } }
void putChpStcks( i )
{ wait( mutex );
  state[i] = THINKING;
  safeToEat( LEFT );
  safeToEat( RIGHT );
  signal( mutex );
```

Alternative solution: Restrict to n-1 people eating → won't deadlock. Or id 0 always take from the right then left.

- **General Semaphore**:

```
int count = <any non-negative integer>;
Semaphore mutex = 1;   //binary semaphore
Semaphore queue = 0;   //binary semaphore, for blocking tasks

GeneralWait() {
  wait( mutex );
  count = count -1;
  if (count < 0) {
      signal(mutex);
      wait(queue)
  } //else removed
  signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    } else {    //else added
        signal(mutex);
    }
}
```

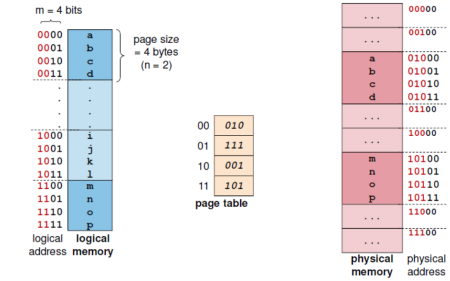
Memory Management

- **Role of OS**: 1. Manage and allocate memory to processes.
  2. Protect memory space of processes from one another.
  3. Provide memory-related system calls.
  4. Manage its own memory space.
- **Memory abstraction**: Provide logical memory. Avoid processes overwriting each other's data. Each program compiles memory references w.r.t. base register. (e.g. lw $1, $[rb + 4096]). Additional Limit Register. Cannot go beyond limit register memory bounds. All memory access is checked.
- **Logical address**: Maps logical address (what process thinks) to physical address (on RAM)

Contiguous memory allocation

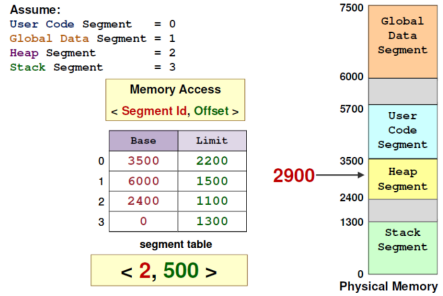
- **Schemes**: 2 schemes.
  1. Fixed-sized partition: Each partition equal size and must fit largest process. Each process gets 1 partition. Pros: Easy to manage, no external fragmentation. Cons: Never use finish ⟹ internal fragmentation.
  2. Dynamic Partitioning: Each process gets (one-time) memory allocated. Causes external fragmentation, as "holes" are useless. Spend time to merge holes and compact memory.
- **Dynamic Partitioning algorithms**:
  1. First-fit. First one that fits.
  2. Best-fit: Smallest one that fits.
  3. Worst-fit: Largest one.
  4. Buddy-system.
  When memory is freed, check whether mergeable with adjacent memory. Also can do defrag/consolidation, but resource intensive. Uses linked list data structure to maintain free/used blocks + size.
- **Buddy System**: Keep an array A[0...K] where K is largest power of 2 allocatable. Every element A[J] in array is linkedlist tracking free blocks of size 2^J.
  Allocation algorithm, request size N:
  1. Find smallest S such that $2S >= N$.
  2. Check if A[S] has a free block.
  3. If yes, return free block. Remove free block from linked list.
  4. If no, search larger block recursively for block, then keep dividing to give a block of size S. Add buddy to A[S]. If no such block exists, alloc fail.
  Deallocation algorithm for block B:
  1. Check whether buddy of B in A[S].
  2. If yes, combine together, remove from A[S] and recursively merge up.
  3. If no, just insert B into A[S].
  Buddy differs from block B by complementing the S^{th} bit.

Paging scheme

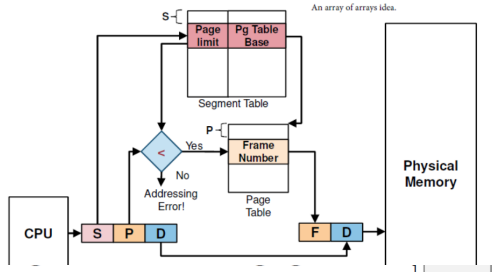


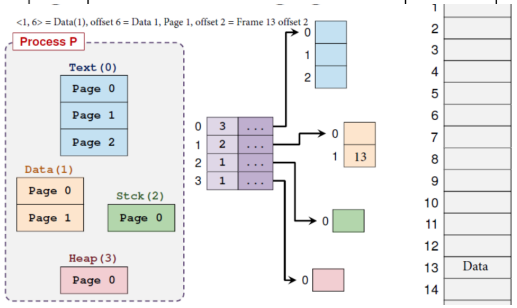
- **Physical frame**: A region of fixed size on RAM.
- **Logical Page**: A region of same size as physical frame, for the process. Size must be power-of-2 to simplify translation.
- **Logical memory**: appears contiguous, just that physically it's now disjoint.
- **Page Table**: Since memory is disjointed, a page table is required to map logical memory pages to physical memory frames. page table == memory context.
- **Logical Address Translation**: Easy to translate. Keep offset, just replace first (m-n) bits with what's in the page table. Can be done with bitshift + AND + OR operations.
- **Paging**: No external fragmentation, but can still have internal fragmentation if page space not fully used.
- **Memory access**: Requires 2 memory access for every memory reference: 1 to access the page table + 1 to actually access data.
- **Translation Look-aside Buffer (TLB)**: Cache of a few page table entries. Resides on the CPU. Part of the hardware context of a process. Significantly improves time taken, searches associatively (in parallel).
- **Protection**:
  1. Use valid bit to indicate validity of page table entry, to prevent program from accessing other process RAM.
  2. Use access right bits to indicate whether memory contents are RWX. E.g. code: R-X. data: RW-.
- **Page sharing**: Multiple processes can share same physical memory by referring to the same physical frame. Can also implement copy on write flag.

Segmentation Scheme



- **Memory segments**: Each type of memory (stack/heap/text) has different requirements. e.g. heap needs to keep growing. Paging scheme cannot allocate new memory if heap and stack are in same block.
- **4 segments**: One each for text/code, global data, heap, stack. Each segment is contiguous in memory.
- **Hardware**: Requires translation table / "segment table".
- **cons**: can cause external fragmentation as variable-size contiguous memory required. similar to dynamic partitioning.
- **Segmentation with paging**: Combine the ideas. Each segment, instead of being contiguous, is now composed of several pages. Segment can grow by adding more pages.





Virtual Memory

- **Page file**: Use secondary storage as extra memory. However, data cannot be accessed directly. Need to swap in to RAM before use. Page table contains extra bit to denote whether entry in RAM or disk.
- **Locality principles**:
  1. Temporal Locality: Memory address that is used, likely to be used again. $\Longrightarrow$ keep page in RAM for a while for future use.
  2. Spatial locality: Memory address close to a used address likely to be used. $\Longrightarrow$ load everything in a page/frame, because subsequent memory addresses likely to be used.

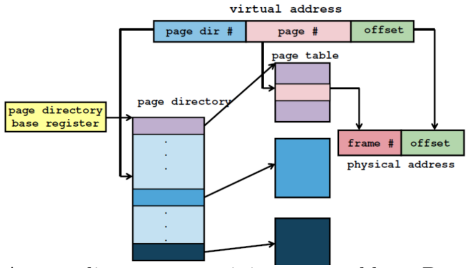
Algorithm for accessing virtual address VA:

1. [HW] VA is decomposed into <Page#, Offset>
2. [HW] Search TLB for <Page#>:
   a. If TLB miss: Trap to OS { *TLB Fault* }
3. [HW] Is <Page#> memory resident?
   a. Non-memory resident: Trap to OS { *Page Fault* }
4. [HW] Use <Frame#><Offset> to access physical memory.

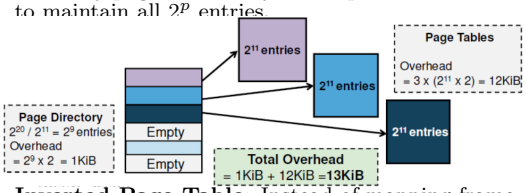
- **TLB Fault**: Access page table for actual page . Replace a TLB entry (replacement algo). Return from TRAP.
- **Page Fault**: Find victim page (page replacement algo). Write out victim page if dirty (copy to HDD, update swap no, update mem resident bit). Bring req page into victim location (update PT, TLB).

Page Table structure

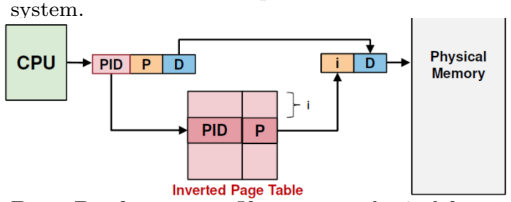
- **Paging**: As page table can be large (e.g. 32-bit system, 4KB page $\Longrightarrow$ 2^{20} page table entries * 2 bytes each = 500 pages), page table is split across multiple pages!
- **2-level Paging / Page Directory**:



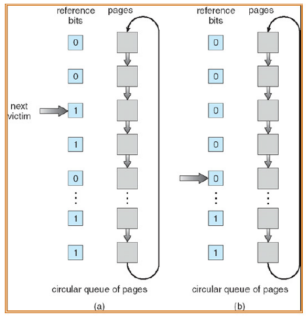
A page directory containing page tablets. Pros: not every page table entry is occupied. Don't need to maintain all 2^p entries



- **Inverted Page Table**: Instead of mapping frame $\Longrightarrow$ data for each process, now one table that maps ¡PID, Page¿ to a global location in RAM. Pros: use at most N bits for page table, where 2^N = max memory size, instead of having many page tables each addressable up to 2^64 on a 64-bit system.



- **Page Replacement**: If no empty physical frame during page fault, need to kick out one frame in memory. If clean, no need write back. If dirty, need to write back.
- **Page Replacement algorithm evaluation**: memory access time: $T_{access} = (1-p) * T_{inmem} + p * T_{fault}$, where p is probability of page fault. T_{fault} significantly greater than T_{mem} so try to reduce p.
- **Optimal Page Replacement**: Replace page that will not be used again for the longest time. Theoretical algorithm because we don't know which page will be used.
- **FIFO**: Evict the oldest page. However, more RAM could actually cause more fault (Belady's Anomaly). Try with 3/4 frames: 1 2 3 4 1 2 5 1 2 3 4 5. Does not exploit temporal locality.
- **Least Recently Used**: Fixes the issue with FIFO/Belady's Anomaly, but not easy as need to keep track of last access and "oldest" access. Can use array/counter or stack, but both don't work well.
- **Second-chance Page Replacement (CLOCK)**: FIFO, but with access bit. The next time a page is on trial to be evicted, if referenced bit == 1, then keep it as it has been used somewhat recently. Reset referenced bit. Otherwise, evict.



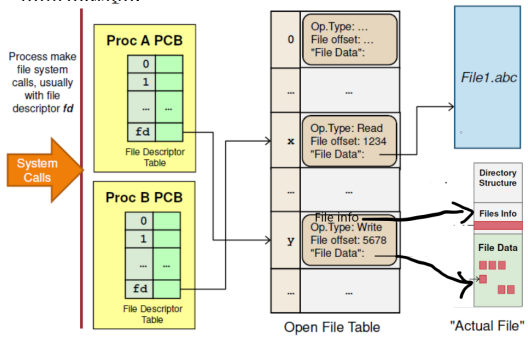
- Use **circular queue** to maintain the pages:
  - With a pointer pointing to the oldest page (the **victim page**)
- To find a page to be replaced:
  - Advance until a page with '0' reference bit
  - Clear the reference bit as pointer passes through

Frame allocation

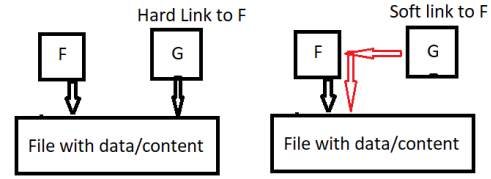
- **Allocate N physical frames to M processes**:
  1. Equal allocation.
  2. Proportional allocation to program size.
- **Local Replacement**: When new frame needed, evict from whatever frame this process is holding. More fair and consistent frame allocation / performance.
- **Global Replacement**: Evict from globally least used when a new frame needed. Allows needy process to use more.
- **Thrashing**: Not enough physical frame results in thrashing. Local replacement: hog I/O. Global replacement: steal frames from other processes, causing them to thrash (cascading thrashing).
- **Working Set Model**: Within a short period, the set of frames required should remain relatively constant. Consider last Δ seconds and see how many frames required. Allocate that amount.

File System

- **Criteria for FS**:
  1. Self-contained / has all necessary info. Can "plug-and-play".
  2. Persistent (across reboots, ie not rely on OS).
  3. Efficient, provides good management of used/free space, and has low bookkeeping overhead.
- **File System Abstraction**: File: contains data. Folder: an organisation of files.
- **File ADT**: Contains Data and metadata like RWX, name, size, owner etc.
- **File protection**:
  1. Permission bits for Owner/Group/World.
  2. (Extended) Access Control List with specific permissions per user.
- **File Metadata Operations**: Rename, change attributes, read attributes.
- **File Data operations**: Create, open, read, write, seek, truncate.
- **File access**: Direct Access, akin to an array of data.
- **Role of OS**: Provide an API to access files / file IDs, maintain open files, maintain seek pointer for process. Use an Open File Table to maintain these information.



- **Process file sharing**: Can maintain two separate Open File Entries to a file (i.e. with a seek). Alternatively, like fork, ¿=1 process can point to an entry, but modifying it will affect all other processes.
- **Directory**: Tree structure. Can be referred to with absolute/relative pathname.
- **Hard Link**: Linkable to file only. Creates another pointer to the actual file data.
- **Symbolic Link**: Linkable to file or directory. Creates a special file that points to the original pointer that points to data.

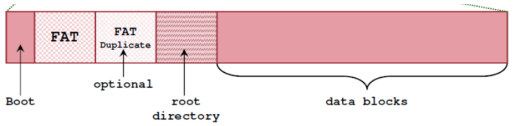


- **Free Space Management**:
  1. Bitmap: 0 = occupied, 1 = free. 1 bit per block. Pros: low overhead.
  2. LinkedList: Each block contains a pointer to next free disk block. Pros: only need first pointer.
- **Directory Implementation**:
  1. Linear List: Each directory consists of a list, containing the files/folders within the directory. Locating a file requires linear search in the directory, which could be inefficient. Mitigation: Cache the entries.
  2. Hash Table: Use Hash Table to maintain directory/file entries. Use separate chaining to

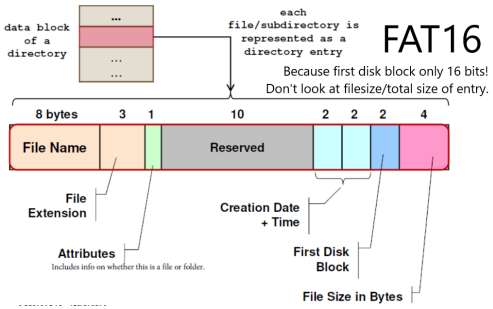
resolve collisions. Fast lookup, but harder to implement, and hashtable has limited size.

- **File information implementation**: Determine where to store file information; together with file or separate.
- **Create file**: At location /.../.../parent/F:
  1. Locate parent directory. Find F and ensure no duplicates. 2. Find a free disk block using free space info.
  3. Add entry to parent directory info.
- **Disk Scheduling**: 3 factors:
  1. Seek time. How long the disk head takes to move to the correct track. (2-10ms)
  2. Rotational Latency. Wait for the correct sector to spin under the head. (2-6ms)
  3. Transfer data (10 μs). Insignificant.
- **Algorithms**: Will focus on reducing seek/rotational latency.
- **First Come First Serve**: Similar to all other FCFS, serve first request first.
- **Shortest Seek First**: Similar to shortest job first.
- **SCAN**: The elevator algorithm. Constantly serving from innermost (track N) to outermost (track 0) and vice versa, bouncing back after serving the last request in the direction. Slightly biased towards center elements.
- **C-SCAN**: Single direction version of SCAN. Outermost $\implies$ Innermost, then reset back to outermost.

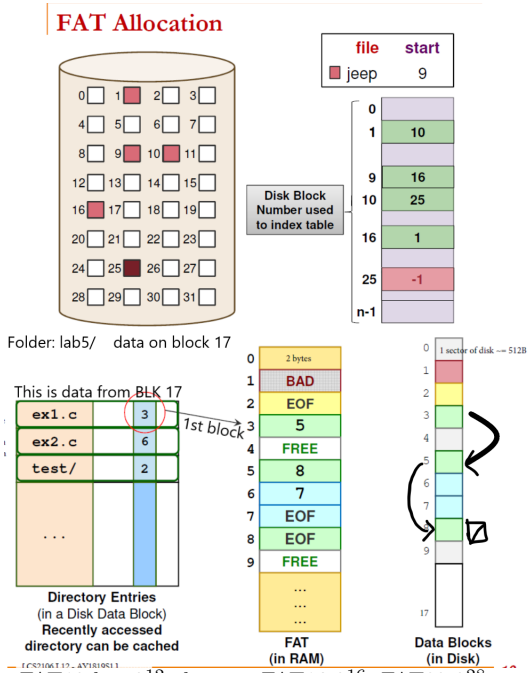
FAT File System



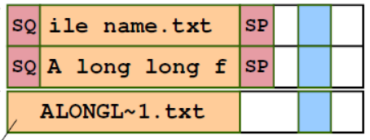
- **Implementation**: Allocation info as linkedlist. OS Cache info in RAM. Free info in the same linked list.



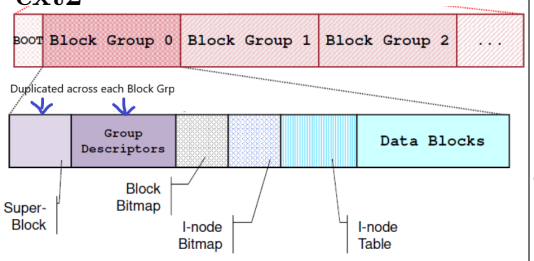
- **Attributes**: Read-Only, Directory/File flag, Hidden etc.
- **Data**: Like a linked list of blocks. Maintain this linked-list in RAM. Somewhat fast random access, as all in memory. However, consumes RAM.



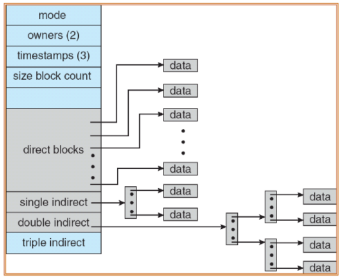
- FAT12 has 2^{12} clusters, FAT16 2^{16}, FAT32 2^{28}
- **Notes**: FAT32 only supports 2^{28} block/clusters, because Microsoft. Actual addressable blocks a bit less than that, because need to reserve special values for FREE/BAD.
- **Cluster**: Determines smallest addressable unit. Larger cluster = can use more disk space, but more internal fragmentation. 4KB on FAT32 = $4 * 2^{28}$ = 1TB.
- **Long filename on FAT**: Supported by using multiple directory entries to continue long file name.



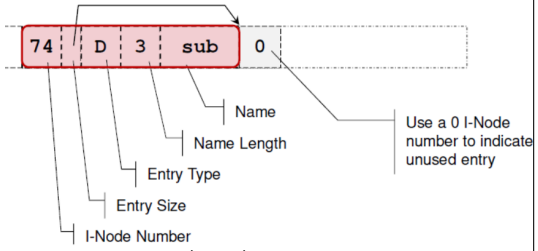
ext2



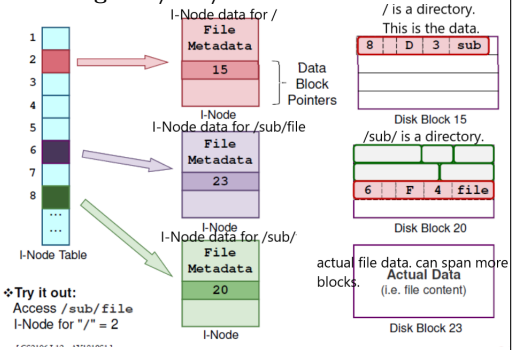
- **Superblock**: Describes entire file system, e.g. total I-Nodes number, total disk blocks etc.
- **Group Descriptor**: Location of bitmaps, number of free disk blocks/I-Nodes.
- **Block bitmap**: Usage status of data blocks.
- **I-Node Bitmap**: Usage status of I-Node blocks.
- **I-Node Table**: An array of I-Nodes, 128 bytes each. If I-Node table takes 1 disk block of 4KB, then max 32 entries only.
- **I-Node Entry**: Metadata about this file/folder. Mode (2) (file, directory, special etc), Owner info (4), file size (4 or 8), Data block pointers (15 * 4), file reference count (2), etc.
- **Indexed Allocation / i-Node**: Maintain index blocks that contains pointers to actual data, or even more pointers (on disk blocks). Each pointer is 4 Bytes. If each disk block is 1KB, then each indirect block can store 1KB/4Bytes = 256 ptrs. Max 16GB file $(12 + 256 + 256^2 + 256^3)$*1KB



- **Directory Entry**: Stored on a data block. A linkedlist. Not fixed size.



- **Accessing file /sub/file**:



- **Unix Dir Permissions**: Read - can list the files in the directory (impact: `ls`). Write - can change the list (impact: add/del/rename file). Execute - can use this dir as working directory (impact: cd).
- Note: Can still modify file in a Read-Only dir. To allow access to file but hide content of dir, just give execute permissions for dir.