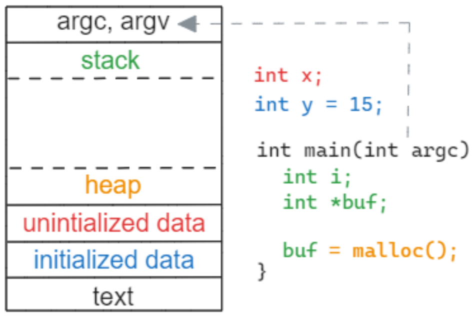
Cheat Sheet 2

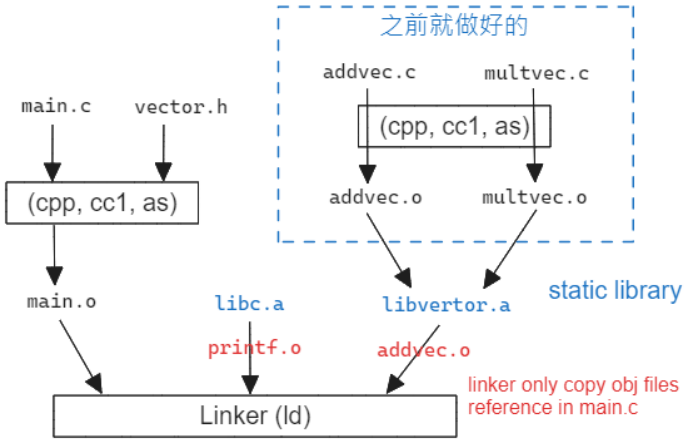
- **RAM**: array of byte-size cells | - each byte has unique PA
- CPU access memory by VA | - **MMU**: translate PA to VA
- **Address Space**: array of byte-size VAs, per-process
- **memory mapping**: OSk connect a VA to PA and manage page table, MMU walk it to address translation
- same VA in different process could map to same or diff PA
- AS mapping are created during `fork()` `exec()`
- child inherit all memory mappings in parent after `fork()`, then OSk perform COW between them



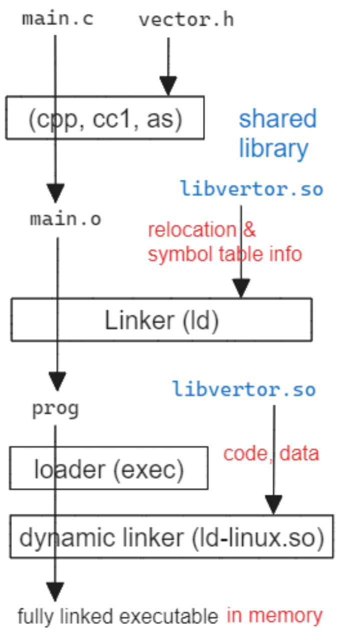
```
int x;
int y = 15;

int main(int argc){
  int i;
  int *buf;

  buf = malloc();
}
```

- `malloc()` : allocate memory from heap, and use `sbrk()` system call to adjust heap size if needed.
- `mmap()` : (system call) create new mapping in AS of caller
addr: mapping 開始位置(如果已有其它 mapping 則選擇其它位置) | ***len***: must >0 | ***prot***: memory protection (`PROT_NONE` can't access, `_READ` can be read) (cannot `_WRITE` if file opened read-only) |
flags: `MAP_SHARED` 更動可以被其它 mapping same region 的進程看到, 其所指檔案也會被更動. `MAP_PRIVATE` 更動不會被看見，其所指檔案不會被更動，而是會建立 COW mapping. `MAP_FIXED` 只能在 *addr* 建立否則失敗. `MAP_ANONYMOUS` 忽略 *fd* 並 zeroed 該 mapping |
fd: fie to be map (can close after `mmap()`) | ***off***: offset of *fd*
- **memory-mapped I/O**: create buffer maps to a memory-mapped file, 對 buffer 讀寫相當於對 file 的對應 bytes 讀寫
- `munmap()`: unmap specific address range
- `mprotect()`: change permission of memory region

- **object files**: **Executable**: 包含 code and data, 可直接複製並執行 | **Relocatable**: 包含 code and data, 可在編譯時和其他 relocatabl 結合來建立 executable | **Shared**: 可被動態載入運行中程式的 relocatable
- **Global (linker) symbol**: defined by obj M that others can reference | **Global external**: referenced by obj M but defined by others | **Local symbol**: defined and referenced only by obj M (e.g. local static var)
- **ELF rel obj file** contains: **ELF header**: specify info about file (for linker to interpret) | **section header table**: sections' size and location | **sections**: `.text`: machine code | `.data`: inited var | `.rodata`: read-only data | `.bss`: uninited var + zeroed var | `.symtab`: symbol table about funcs and global var defined and referenced in program, no automatic var | `.rel.text`: relocation entries for code: location of code that call external func or global var | `.rel.data`: relocation entries for data: global var referenced or defined by file
- **relocation entry**: generated for references to symbols whose runtime addr is unknown
- **Static Linking with static libray**:
symbol resolution: linker 在輸入的 rel obj files 的 symbol tables 裡尋找每個 symbol 的定義，如果沒找到則報錯
symbol relocation: (1) merge sections from obj files to new aggregated section of same type, assign runtime memory addr to new aggregated sections ; (2) linker check relocation entries and update symbol references in program to reference symbols with correct runtime addr



- **Program Compilation**
`hello.c`: source program
↓ **Preprocessor** (cpp): 插入 .h 檔
`hello.i`: modified source program
↓ **Compiler** (cc1): 轉譯成組合語言
`hello.s`: assembly program
↓ **Assembler** (as): 轉譯成機器語言並打包
`hello.o`: relocatable obj file
↓ **Linker** (ld): 合併其他 rel obj files
`hello`: executable obj file
- **Dynamic Linking wit shared library**



- **Loading**: loader copy code and data in exe from storage to process AS, and run program by jumping to program's entry point (C: addr of `_start()`) ; invoked when `exec()`

- **reentrant func**: 可以被遞迴呼叫 ; 不使用 static/global var, 不要用 `malloc()`, 不呼叫 non-reentrant
- **pending**: 信號已產生但未被處理
- **delivered**: 信號正在被處理
- process **blocking** signal: 該信號 pending 直到被 unblock 或是 action 變成 ignore
- **signal mask**: per process 決定要被 blocking 的 signal set
- **caller**: 呼叫 func 的 func | **callee**: 被呼叫的 func
- **stack frame**: stack 裡面保留給 callee 資訊的空間，callee 返回後釋放
- **nonlocal jmp** 後 static, global var 不會回復

	fork	exec
signal mask	inherit	unchange
pending signal	clear	unchange
signal action	inherit	ignore不變，其他default

- **thread shared**: Text, Global Data, Heap, Open fds, Environment variables, PID, File record locks, Signal, Pending alarms, Signal handlers | **per-thread**: TID, Stack, Signal mask, Errno, Scheduling properties, Thread-specific data
- **Many-to-one**: multi user thread to one kernel thread (context switch by library)
- **One-to-one**: every user thread to one kernel thread (context switch by kernel)
- **pthread 運作**: process 一開始只有一個 main thread, 而 main 會建立其他 peer
- **cleanup handler**: 一個 thread 可以註冊多個 | 被存在 stack, 依照註冊順序相反被呼叫 | 當 `pthread_exit()` 時會自動呼叫(`return()` 不會)
- **mutex**: 一次只有一個 thread 可以讀寫 | **rwlock**: multi-read mode 和 only-write mode | **spin lock**: 和 mutex 相同，但以 busy-waiting 的方式來 block
- **cond variable**: a thread release its mutex and block **waits** on the condition variable | another thread changes the condition and **notifies** the condition variable to unblock the waiting thread
- **spurious wakeup**: thread 被喚醒但發現 condition 沒有被滿足
- **thread and fork**: `fork()` 會繼承 mutex, rwlock, cond | 因此當 parent 內的 threads 持有任何鎖，則 child 也會持有這些鎖。然而 child 內只有一個 thread (呼叫 `fork()` 的那個的複製)，因此 child process 無法得知其他 threads (沒有呼叫 `fork()` 的那些) 持有的鎖，也無法解除這些鎖。| 解決方法是立即呼叫 `exec()` 或是用 fork handler
- **thread and exec**: 某個 thread 呼叫 exec 後，其他的 threads 都會被立即 terminate
- **thread and signal**: 相同 process 的 threads 共享 signal action, 任何人都可以更改 | 信號實際上是被傳遞給 process 內隨機的一個 thread (與硬體故障相關的 signal 除外)
- **thread-safe**: 可以同時被多個 threads 呼叫的 func

function	effect
`_self`	get tid
`_equal`	check if tid same
`signal`	set signal action | ***handler***: 自定函數指標, `SIG_IGN`, `SIG_DFL`
`sigaction`	change signal action | ***sa_mask***: handler 執行中要 block 的信號 (呼叫handler之前將目前delivered + ***sa_mask*** 內的信號加入mask ; 結束後回復)
`sigprocmask`	change signal mask | ***how***: `SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`
`sigpending`	get pending signals
`kill`	send signal to ***pid***: `>0`:pid ; `=0`:gid=sender's gid ; `<0`:gid=pid ; `=-1`:all | set ***signo***=0, return `-1`:pid 不存在 ; `0`:存在
`raise`	send signal to self
`alarm`	send SIGALRM | ***sec***: `>0`:覆蓋舊 alarm ; `=0`:取消 pending alarm | return seconds left of current alarm
`pause`	pause until any signal catched
`setjmp`	直接呼叫返回0 ; 經由 `longjmp` 則非 0
`longjmp`	跳回設定 ***env*** 的 `setjmp` 處
`sigsuspend`	將 mask 替換成 ***sigmask*** 後暫停直到 (1)收到信號→在handler之後返回並復原 mask (2)收到結束進程信號→不返回
`abort`	unblock SIGABRT and send SIGABRT
`sleep`	pause 直到收到信號→返回 seconds left ; 或時間過完

function	effect
`_join`	block wait joinable thread | copy exit status into location pointed by ***rval_ptr***
`_detach`	detach a thread
`cleanup_push`	註冊 cleanup handler
`cleanup_pop`	移除頂層 handler | ***execute*** != 0 則移除前呼叫該handler
`mutex_lock`	mutex 上鎖, if can't block until unlock
`mutex_trylock`	mutex 上鎖, if can't return EBUSY
`mutex_unlock`	mutex 解鎖 | 解鎖後第一個執行的線程獲得 mutex, 其他則繼續 blocking
`cond_wait`	atomically unlock mutex and block wait until cond signaled, shall acquire mutex when return | ***mutex*** 必須自己持有
`cond_signal`	unlock 至少一個被 cond block 的 thread
`cond_broadcast`	unlock 所有被 cond block 的 threads
`_atfork`	註冊 fork handler | ***prepare***: 建立child之前由parent執行 | ***parent***: 建立child之後，返回之前由parent執行 | ***child***: 返回之前由child執行
`_sigmask`	thread 版的 `sigpromask()`
`sigwait`	atomically unblock ***set*** 裡面的信號, block wait 直到某個delivered, 接著將該信號從進程的 pending set中移除, 並回復signal mask | ***signo***: 儲存接受到的信號 | 如果等待的信號有handler, 則該handler不會被值行 | 呼叫前必須把 ***set*** block 住 | 如果多個thread對同信號sigwait則只有一個會返回
`sigkill`	send signal to thread | 如果信號是結束process, 則所屬process會結束