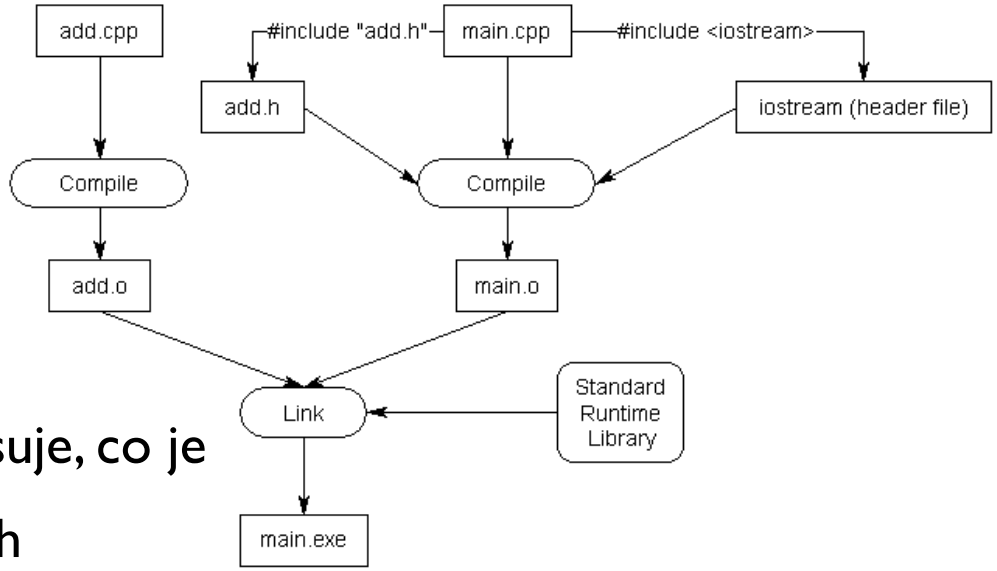
KIV Operační systémy

Procesy a lehká vlákna

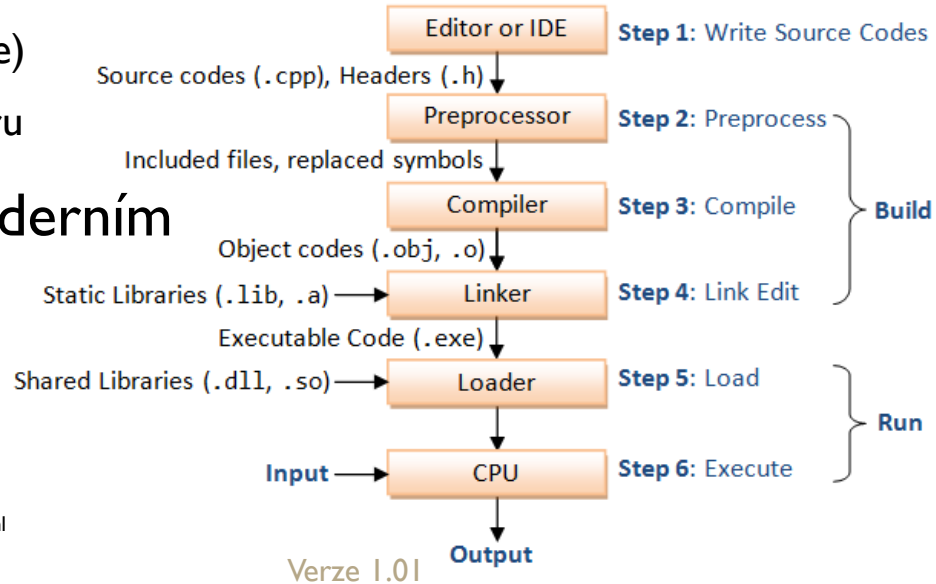
Verze 1.01

Od C++ k exe

- .h* hlavičkový soubor
- .o* objektový soubor
- .lib, .a objektový soubor popisující .dll, .so



- Hlavičkový soubor popisuje, co je v objektových souborech
  - Není vyžadováno (byť se dodržuje) mapování 1:1 podle jména souboru
- Hlavičkové soubory v moderním C++ 17 nahrazují moduly
  - #include → import & module



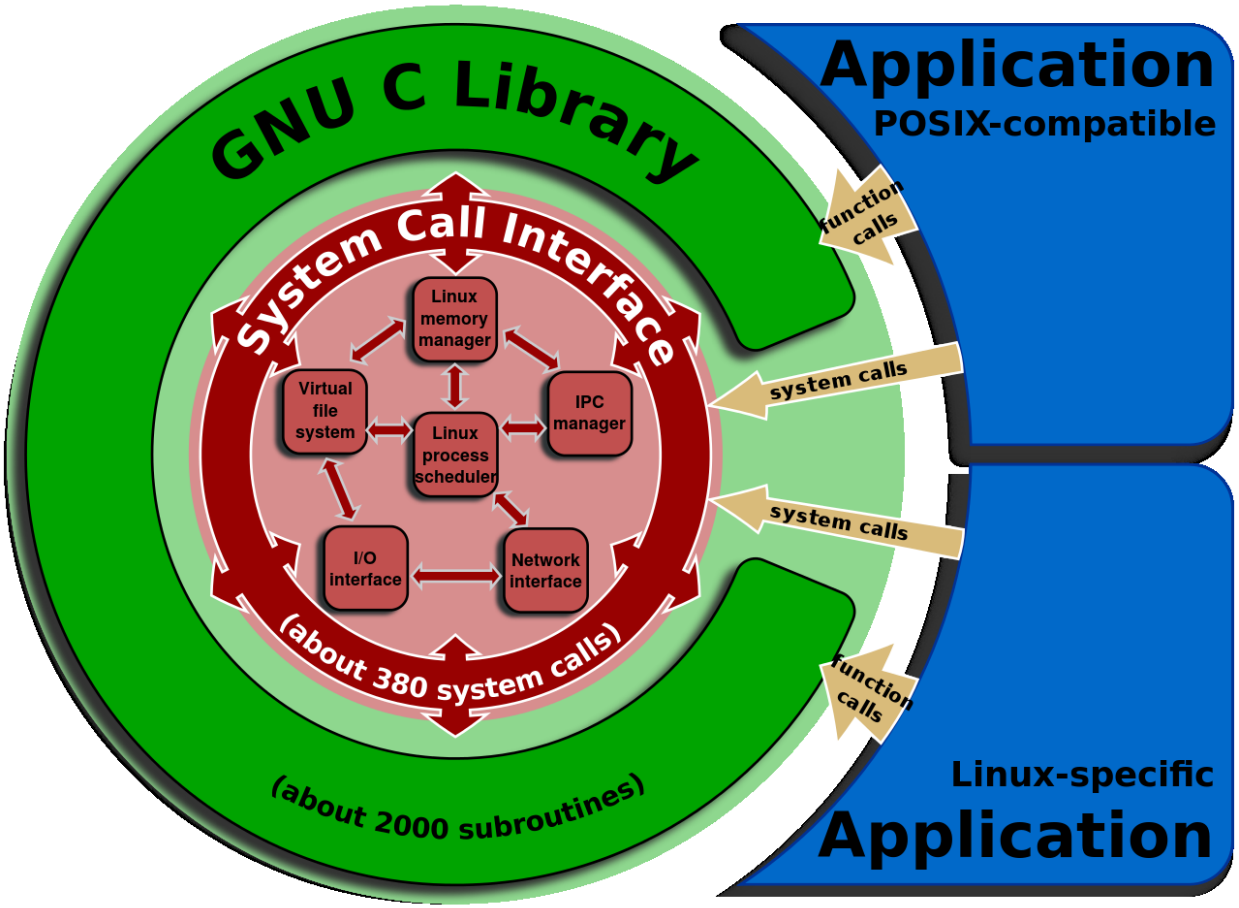
http://www.learncpp.com/cpp-tutorial/19-header-files/

https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp1_Basics.html

Stavební kameny

- Co už víme?
  - Správa paměti
  - Služby OS
  - Vlákna a jejich vzájemná izolace
  - SMP
- Co chybí?
  - Umět načíst a spustit kód uživatelské programu do paměti
  - Uzavřít/zaobalit běžící uživatelská programy do procesu

Volání služeb OS přes RTL



http://en.wikipedia.org/wiki/GIO_%28software%29

Jak to vidí programátor

- Programátor v C si napíše funkci main, a když spustí svůj program, jeho kód se začíná vykonávat na prvním příkazu této funkce
  - int main(int argc, char **argv) { return 0; }
- Jakmile se z funkce main vrátí, tj. udělá return, program skončí
  - Ale kdo a jak mu nastaví hodnoty argc a argv?
  - A kdo a jak zařídí, že program vrátí návratový kód 0?

Jak to vidí programátor RTL

1. OS vytvoří virtuální paměťový prostor pro proces
2. OS rozdělí paměťový prostor na blok pro kód, zásobník, data...
3. OS nahraje soubor programu do paměťové prostoru spolu s načtením knihoven
4. OS předá řízení na první instrukci z načteného programu
   - tzv. crt0 – kód programátora RTL

crt0

- crt znamená „C Runtime“, 0 znamená „úplný začátek“
  - Konkrétně se může jmenovat i jinak – např. u jiného jazyka
- V podstatě se jedná o bootstrap/inicializaci RTL, která např. nastaví hodnoty argc a argv, stdin/out, zavolá konstruktory globálních instancí tříd, atd.
- Jakmile je inicializace RTL hotova, teprve pak se zavolá programátorova funkce main
- Funkce main se pak vrátí do kódu crt0, odkud byla zavolána a RTL provede deinicializaci, včetně nastavení návratové hodnoty main jako exit code procesu

crt0 - zásobník

- Bez zásobníku program nespustíme – nemůžeme dělat call, ani int
  - Ale kdo zásobník vytvoří? OS nebo crt0?
- crt0 je napsaná v assembleru, kde je nějakým způsobem vyhrazena paměť pro zásobník
  - Syntaxe assembleru, která vyhradí příslušně velikou oblast paměti, případně nastaví její atributy a registry ss:esp
  - Registry lze nastavit i manuálně

crt0 - debug

- Pokud je program zkompilován v debug módu, crt0 provádí další dodatečnou inicializaci ladění
  - Může zavádět i dodatečné knihovny, číst specifické konfigurační soubory/registry
- Nebezpečí
  - Pokud je program zapomenut zkompilovaný v debug módu, a např. díky tomu zkouší načítat nějakou specifickou knihovnu, hackerovi stačí podstrčit svou knihovnu, která se bude stejně jmenovat a bude mít stejné jméno a prototyp funkce

crt0 - chyby

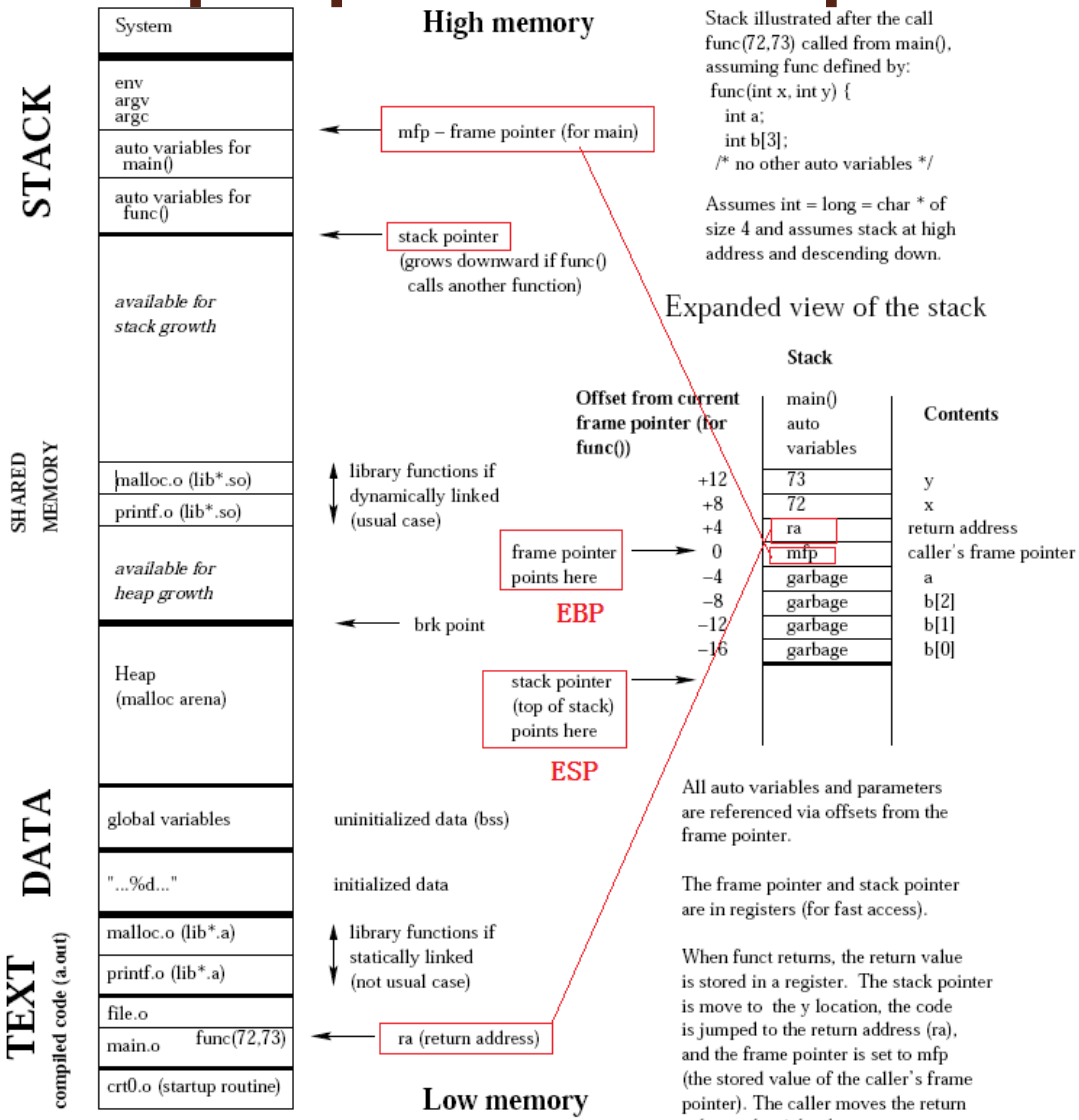
- Žádný program není bez chyby
  - Chyba může být už v crt0
- Např. programy zkompilované s překladačem BP 7.0 včetně, vždy skončily s "Runtime Error 200 (Division by zero") na procesorech Pentium II 266 MHz a novějších.
  - Zajímavé je, že C++ překladače od stejné firmy tu samou chybu neměly
  - A nulou se ve skutečnosti nedělilo

crt0 - Delay

```
CALL    DelayLoop
NOT     AX
NOT     DX
MOV     CX, 55
DIV     CX
MOV     DelayCnt, AX
```

- DelayLoop počítala, kolikrát je možné snížit integer v AX:DX během periody hodin 55ms.
- Jenže rychlý procesor to stihnul příliš mnohokrát a výsledek se nevešel do AX => DivideError exception

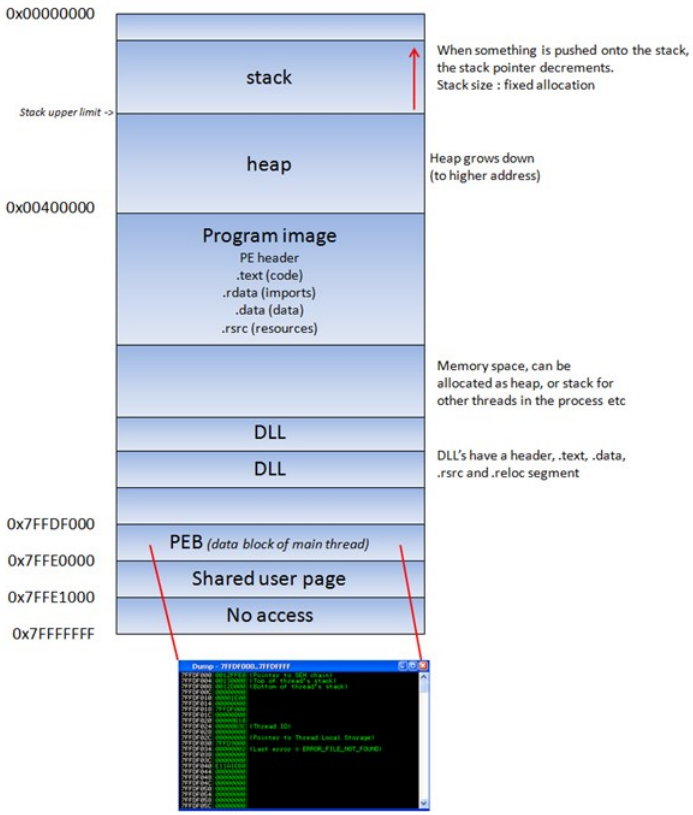
Mapa procesu v paměti

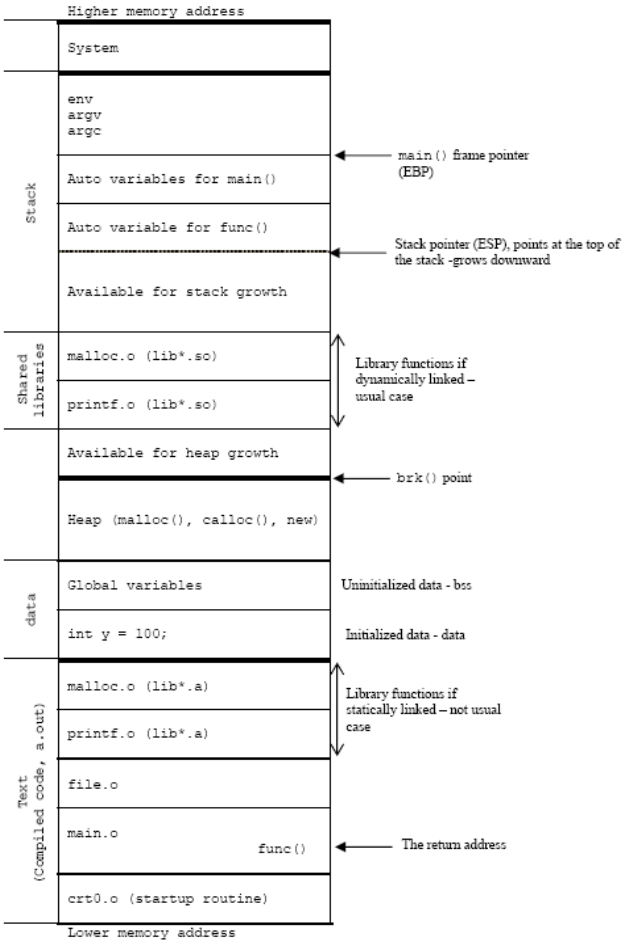


http://www.codingfail.com/cpp-internals-memory-layout/

https://stackoverflow.com/questions/15700694/linux-process-memory-scheme

Windows vs Linux





http://www.cs.fsu.edu/~redwood/OffensiveComputerSecurity/reversing/FSU_Reversing.pptx

Zavedení procesu - OS

1. Služba OS obdrží požadavek na spuštění procesu
2. Existují však alespoň dvě strategie:
   - UNIX-like – program volá fork, čímž se vytvoří nový proces, ale stále s kódem programu, který volal fork. Je proto ještě nutné zavolat exec, který nahraje do paměti kód nového programu.
     - Mezi voláním fork a exec se dá dělat/nastavit cokoliv.
   - Windows – existuje pouze CreateProcess (který ve finále volá i např. ShellExecute).
     - CreateProcess umí jenom to, co lze popsat s jeho parametry.
   - Flexibility vs. security?

fork

```
void DoFork() {

   int rc = fork();

   if (rc<0)  HandleError();

      else if (rc == 0) ChildExecute();

           else ParentExecute();

}
```

fork

- OS vytvoří přesnou kopii stavu rodiče (volajícího fork)
  - Virtuální paměťový prostor, tj. včetně zásobníků a haldy
    - Použije se COW
  - Registrů procesorů
    - Na x86 mimo eax, ve kterém se vrací výsledek operace
    - Souborové deskriptory – tj. soubory se sdílejí
- Potomek bude mít stejný, ale vlastní/oddělený stav
- Plánovač OS je může naplánovat v libovolném pořadí
  - Na rozdíl od UNIXu, na Linuxu by to měl být potomek

fork, vfork, clone

- vfork bývalo optimalizované volání pro první UNIXy s virtuální pamětí – tj. COW
  - fork jinak musel dělat plnou kopii adresového prostoru
    - Ještě před COW, vfork spustil potomka přímo v adresovém prostoru rodiče - nebezpečné
  - Dnes obsolete, discouraged and removed
- clone
  - fork s množstvím parametrů – takže lze použít jako fork i jako pthread_create – říkají, co všechno se zkopíruje
  - fork implementován přes clone

pthread_atfork

- 3 parametry
  - Prepare – spustí se před vykonáním fork
  - Parent – spustí se v rodiči po vykonání fork
  - Child – spustí se v potomkovi po vykonání fork
- Proč?
  - Dejme tomu, že z nějakého důvodu potomek zdědí nějaký zdroj (resource) od rodiče (uzamčený zámek, kus paměti, ..), ale jak ho má správně uvolnit? A kdy?
    - atfork umožní rodiči řádné uvolnění těchto zdrojů, případně jejich opětovné získání

Process Control Block

- Struktura popisují proces operačnímu systému
  - Process ID (PID)
  - Stav – běžící, připravený, blokovaný, ukončený
  - Registry procesorů
  - Info o adresovém prostoru
  - Vlastník a práva
  - Plánovací priorita
  - Pointery na stavové fronty
  - V Linuxy definován v task_struct(linux/sched.h) s více než 95 položkami
- Obsahuje informace o alokovaných resources, které OS uvolní po ukončení procesu

Stavová fronta

- OS udržuje kolekci seznamů (linked list), které drží stav všech procesů a vláken
  - Každý stav má svůj seznam
    - Včetně podstavu – např. proč proces/vlákno zrovna čeká
  - Jak se mění stav procesu/vlákna, PCB/TCB se přesouvá mezi jednotlivými frontami
- Když se vytvoří nový proces, musí se vytvořit i nová PCB a TCB

PCB po fork

- Minimálně má potomek nové PID
  - Tj. na PCB se nepoužije COW, protože by se hned dělala kopie
- Jak proces běží a jeho vlákna jsou plánována, PCB se kontinuálně mění
- Potomek také může skončit s jiným návratovým kódem než rodič
  - Návratový kód je uložen v PCB

Zombie/defunct process

- Proces, který je ve stavu ukončen, ale stále má svou PCB
- PCB drží návratový kód, dokud nebude přečten
  - Např. fcí wait rodičem, který ho forknul
- Zombie není orphan proces – tj. proces, jehož rodič skončil dříve než potomek
- Zombie – fiktivní stvoření, znovuoživený člověk, který zemřel – tj. byl proces byl v jiném než terminated stavu, ve kterém se jako zombie nachází

To reap the zombie

- Reap – viz Grim Reaper (Smrtka)
- Když si někdo nepřečte návratový kód zombie, PCB zůstane v paměti – resource leak
  - Co když bude počet zombie stále narůstat?
  - Sice se tím na dnešních počítačích paměť nevyčerpá, ale může se vyčerpat počet volných položek v tabulce PCB
    - Anebo se vyčerpají volné PID?

Single Unix Specs v3

- Pokud rodič ignoruje SIGCHLD výslovným nastavením signálu na SIG_IGN, nebo má nastaven příznak SA_NOCLDWAIT, z jeho potomků se nestanou zombie
- Nebo lze poslat kill rodiči a pokud ten odmítne přečíst návratový kód zombie (to reap the zombie), killne se rodič a rodičem potomka se stane init
  - init periodicky čte návratové kódy jeho zombies

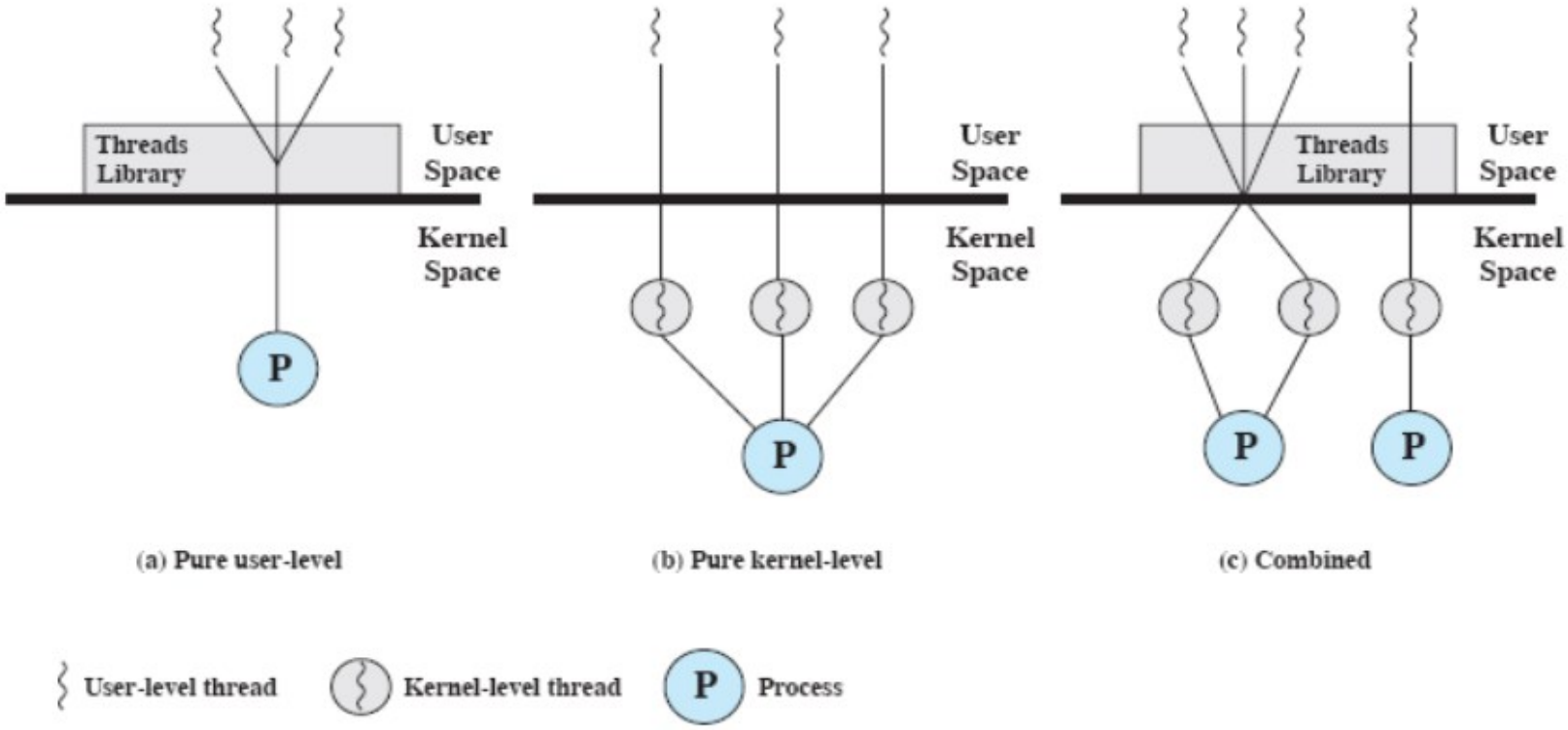
exec

- Nahradí stav aktuálního procesu a jeho kontext
  - Nahraje do paměti nový programový kód
  - Nastaví se nový zásobník
    - Vlákna se zredukují na jedno
  - A předá se řízení crt0
  - Po úspěšném exec se už řízení nevrátí původnímu programovému kódu
- Protože se ale nevytváří nový proces, PID zůstává

Kernel threads

- Spravuje je jádro OS
  - Přepínání
  - Plánování
  - Synchronizace
- Nad kernel threads běží vlákna v processech
  - Buď 1:1 (těžké) – pohodlné, vše už umí a zařídí jádro OS
  - Anebo 1:N (lehké)
    - Výkonnostním cílem je redukce přepínání do režimu jádra

User vs. kernel threads



W. Stallings, Operating Systems, 6th edition

Kernel-backed threads

- Spravuje je jádro OS
  - Přepínání, plánování, synchronizace
  - Kernel thread běží pouze v režimu jádra
- Nad kernel-backed threads běží vlákna v processech
  - Buď 1:1 („těžké“) – pohodlné, vše už umí a zařídí jádro OS
  - Anebo 1:N („lehké“) - výkonnostním cílem je redukce přepínání do režimu jádra
  - Jak to bude, to záleží na RTL

User-space threads

- RTL dokáže „pozastavit“ kernel-backed thread, který vykonává kód procesu, změnit registry tohoto threadu, a znovu ho „spustit“
  - Tj. naplánovat nový thread bez přepnutí do jádra
- Jenomže, bez znalosti plánovače RTL může udělat plánovač jádra rozhodnutí, které nemusí být optimální, může vést i k deadlocku
- =>upcall – jádro informuje RTL o tom, co se chystá udělat – např. blokovat použitý kernel-backed thread

Linux threads

- Na rozdíl od Windows a UNIXu, Linux nemá koncept thread a vše implementuje jako proces
  - Na nich PCB popisuje to, co je relevantní pouze pro proces, a TCB to, co je relevantní pouze pro thread
- fork: clone(SIGCHLD, 0);
- pthread_create: clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
  - Nový proces, který s rodičem sdílí adresový prostor, přístup k souborovému systému vč. souborových deskriptorů, obsluhy signálů

OOP threads

- CreateThread(proc* func, void *data) vytvoří thread, který z pohledu programátora poběží od první instrukce na adrese func – kód píše programátor RTL
  - Func má prototyp int func(void *data)
  - data se interpretují jako pointer this/self

```
int func(void *data) {
        auto thread = static_cast<CThread*>(data);
        thread->Execute();
}
```

CreateThread

- OS ve skutečnosti spustí svoji funkci, která teprve zavolá funkci, jejíž adresa se předala jako argument CreateThread
- Tzn. tato funkce nemusí volat ExitThread, protože se vrátí do funkce poskytnuté OS, která už ExitThread zavolá
  - De-facto jde o obdobu crt0