Operačné systémy / Procesy

Dušan Bernát
bernat@fmph.uniba.sk

Prehľad

- Program a proces.
- Používateľský kód, vzťah k okoliu (súbeh, komunikácia, V/V).
- Stavy procesu a plánovač úloh.
- Preempcia, časovač, prerušenia, prepínanie kontextu.
- Implementácia procesu.
- Prostriedky a ich limity.
- Čas systémový/používateľský, procesorový/reálny.
- Signály (ukončenie, pozastavenie a iné).
- Vznik procesu.
- Vlákna.

Čo je proces?

Čo je proces?

- Program – statický, napríklad vo vykonateľnom súbore na disku.
- Proces – program, ktorý sa práve **vykonáva**. Má stav.
- Z jedného programu môže byť spustených (bežať) viacero procesov.
- Proces je sekvenčný, má jeden tok riadenia.
- OS umožňuje vykonávanie viacerých (mnohých) procesov súčasne.
  - Spravidla ich môže byť viac než počet procesorov.
- Procesy sú nezávislé
  - Vykonávajú sa súbežne, bez ohľadu na ostatné.
  - Neovplyvňujú sa. Sú izolované. Každý má vlastný pamäťový priestor.
- Môžu komunikovať, ale musia o to požiadať operačný systém.

Multitasking

- Operačný systém vytvára pre každý proces dojem, že má procesor (rozšírený stroj) len pre seba. Ide o formu virtualizácie.
- Procesy sa o dostupné fyzické procesory (jadrá) delia.
  - Striedanie pri vykonávaní (*time sharing*) riadi OS.
  - Napr. Keď proces spotrebuje pridelené časové kvantum, odloží sa a začne sa vykonávať iný. Neskôr mu môže byť procesor opäť pridelený.
  - Toto prepínanie je pre proces transparentné.
- Ako sa urobí prepnutie na iný proces (*task switch*)?
- Ako sa odoberie procesor bežiacemu procesu?
- Ako OS rozhodne, ktorý proces má byť ďalší?

Reprezentácia procesu

Reprezentácia procesu

- Aby bolo možné proces odložiť a neskôr pokračovať od prerušeného miesta, je potrebné úplný stav procesu pri prerušení odložiť.
- **PCB** (Process Control Block) každého procesu obsahuje:
  - Stav CPU (**kontext**): *registre*, všeobecné aj špeciálne (napr. Intel AX, BX, CX, DX, ale aj IP, SP, BP, FLAGS, ...).
  - Stav *pamäti*
    - Inštrukcie (text), statické dáta (data), zásobník (stack), halda (heap), ...
    - Každý proces má svoj logický adresový priestor mapovaný do inej časti fyzickej pamäte → uchovať treba (len) tabuľky stránok.
  - Ostatné prostriedky: otvorené súbory, zámky, sieťové spojenia, signály, ...
  - Informácie o alokovaných a spotrebovaných prostriedkoch, ich limity pre proces, ...
  - Plánovacie informácie: **stav** (beží, pripravený, blokovaný, ...), priorita, na čo čaká, ...
  - Identifikácia procesu: PID, rodič, UID (vlastník), CMD (program), ...

Tabuľka procesov

- Všetky potrebné informácie o procese uchováva jadro v jeho PCB.
- PCB všetkých procesov spolu tvoria tabuľku procesov.
- Napríklad, OS Linux:
  - PCB je implementovaný štruktúrou `task_struct`.
  - Tá obsahuje odkazy (pointre) na niektoré ďalšie tabuľky.
    - Tabuľka otvorených súborov, rad čakajúcich signálov, tabuľka namapovaných oblastí v adresovom priestore (`mm_struct`), ...
  - Tabuľka procesov je dvojitý spájaný zoznam jednotlivých `task_struct`.
    - Prístupné aj cez `struct task_struct *pid_hash[pid]`. Prehľadávajú sa len položky s rovnakým hash-om.
  - Pointer `current` ukazuje na `task_struct` práve vykonávaného procesu.
  - Tabuľka procesov je v *user-space* dostupná cez (pseudo) súborový systém `/proc`, alebo príkazy `ps`, `top`.
    - Každý proces má v `/proc` podadresár so svojim PID.

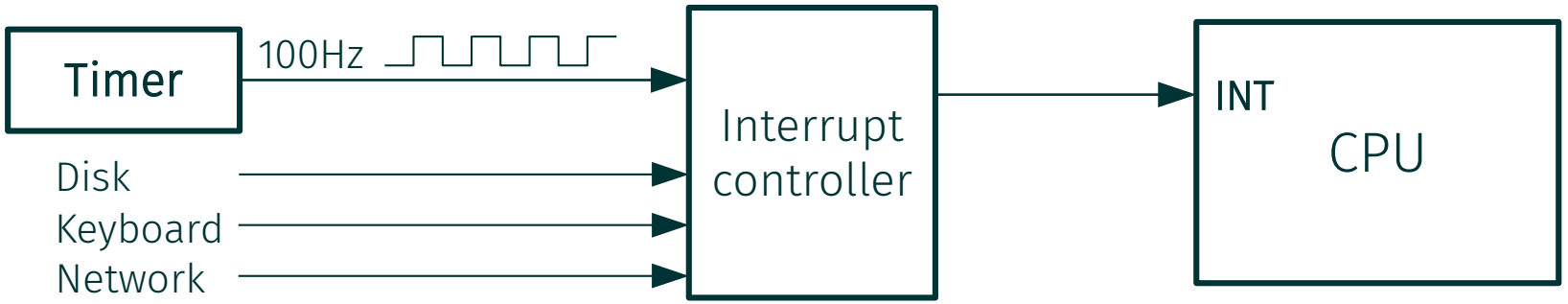
Prepínanie procesov a preempcia

Prepnutie procesu (zjednodušene)

- Zmena aktuálne bežiaceho procesu na iný (= *context switch*).
- Zmena je potrebná ak proces nemôže bežať ďalej
  - napr. z dôvodu čakania na dokončenie V/V operácie pri systémovom volaní, alebo
  - spotrebuje pridelené časové kvantum.
- Môže nastať **pri systémovom volaní** (v podstate dobrovoľne), alebo **pri prerušení**. Súčasťou obsluhy prerušenia je odloženie aktuálenho stavu CPU do príslušnej časti PCB.
- Jadro (konkrétne plánovač procesov) zvolí nový proces na vykonávanie.
- Nastaví do CPU stav z PCB nového procesu.
  - Všetky registre ktoré môže proces používať.
  - Nastaví aj register ukazujúci na začiatok tabuľky stránok (CR3), čím sa “prepne pamäť”.
- Po návrate z prerušenia beží už druhý proces.
  - Napr. Intel, inštrukcia IRET obnoví zo zásobníka EIP, ESP, EFLAGS a zmení CPL.
- S prepnutím procesu je spojená istá réžia. Mali by trvať čo najkratšie a byť zriedkavé.

Prerušenie vykonávajúceho procesu

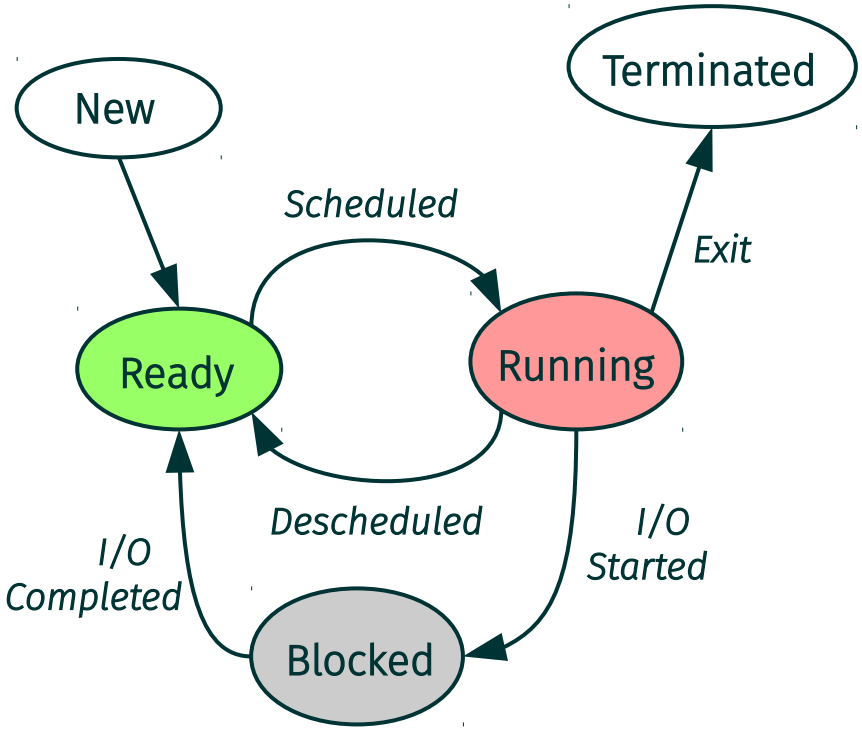
- Aby OS poskytol zdanie súbežného behu procesov (pseudoparalelizmus), musí po uplynutí istej doby zabezpečiť prepnutie procesu.
- Doba počas ktorej môže proces bežať sa nazýva časové kvantum.
- Na zabezpečenie nedobrovoľného odobratia CPU (preempcia) sa využíva hardvérové prerušenie od obvodu časovača.
  - Ten má zvyčajne frekvenciu 100Hz (dá sa nastaviť).
- Obsluha prerušenia od časovača vyvoláva plánovač úloh OS.
  - Ak práve bežiaci proces už spotreboval svoje kvantum, plánovač urobí prepnutie procesu (context switch).



Systémy bez časovača (*tickless*)

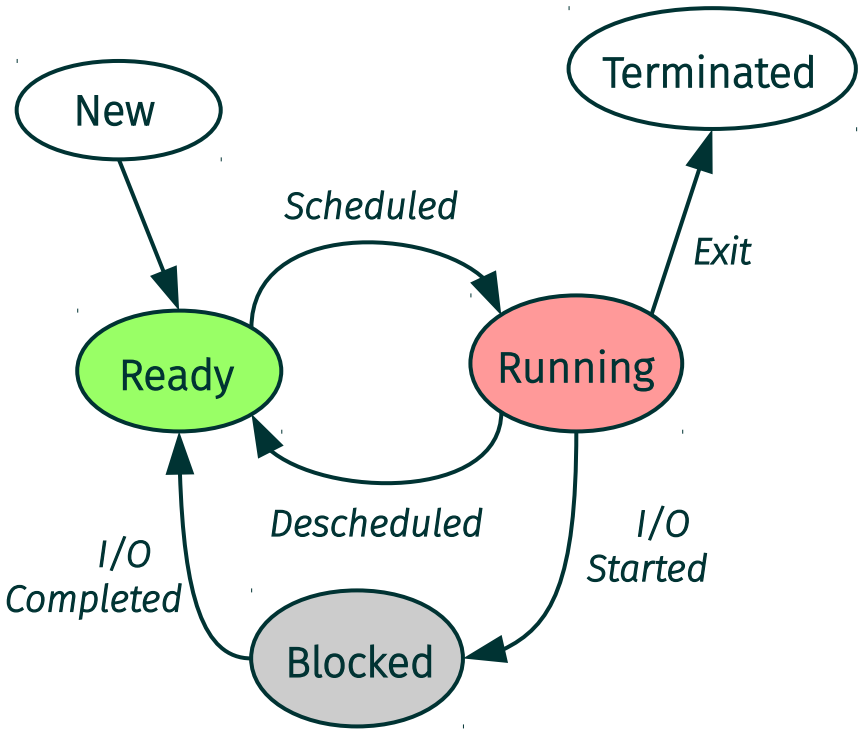
- Periodické prerušenia sa využívajú aj na obsluhu časovačov nastavených procesmi. Napríklad proces urobí volanie `sleep()`.
  - Obsluha prerušenia je však vyvolaná aj ak to nie je potrebné, teda ak doba žiadneho časovača neuplynula.
  - To zvyšuje réžiu a teda aj spotrebu.
- V nových systémoch je možné naprogramovať prerušenie flexibilne podľa toho, kedy najbližšie má nastať takáto udalosť (*on-demand interrupt*).
  - Periodické prerušenie teda nie je potrebné, pokiaľ v systéme nie sú bežiace procesy (respektíve ak je ich počet 0 alebo 1).
  - Ak nie je **žiaden**, procesor nebude zbytočne budený a môže zostať v stave s nižšou spotrebou, čo je výhodné pri mobilných zariadeniach, ale aj virtualizovaných OS (obsluha je v každom).
  - Ak je **jeden**, nie je potrebné sledovať jeho časové kvantum, procesor má stále pridelený, bez réžie obsluhy zbytočných prerušení. Výhodné pre HPC aj RT.
- Kvôli aktualizovaniu štatistík pre plánovač sú prerušenia potrebné aspoň raz za sekundu, čo však aj tak predstavuje značnú úsporu (oproti bežným 100Hz).

Stavy procesu



- New – novovytvorený proces, po inicializácii bude pripravený.
- Terminated – ukončený proces.
- Ready – proces je pripravený sa vykonávať, ale **čaká** kým mu OS pridelí CPU.
- Blocked – proces sa nemôže vykonávať, kým nenastane udalosť na ktorú **čaká** (prídu dáta z disku).
- Running – plánovač pridelil procesu CPU, **vykonáva sa**.

Zmeny stavu



- *Scheduled* – pripravený proces bol plánovačom vybraný na vykonávanie. Bude mu pridelený CPU.
- *Descheduled* – proces spotreboval svoje časové kvantum. Procesor mu bude nedobrovoľne odobraný, hoci by sa mohol vykonávať ďalej.
- *I/O Started* – proces sa nemôže ďalej vykonávať. Potrebuje niečo, čo nemá. Nepotrebuje CPU kým to nenastane.
- *I/O Completed* – nastala udalosť na ktorú proces musel čakať.
- Blokovaný proces čaká na externú udalosť, nezávislú od OS (nemusí to byť vždy I/O).

Stavy procesu – prakticky

- Výstup príkazu `ps`, respektíve `top`
  - **R**, pripravený (Ready/Runnable)
  - **S**, prerušitelné čakanie (Sleep); **I**, ako S, ak trvá dlho (Idle)
  - **D**, neprerušiteľné čakanie, neprerušiteľné signálmi,
    - typicky čakanie v jadre na dokončenie V/V operácie (Disk)
    - môže však zahŕňať aj množstvo iných stavov, napríklad čakanie na zámky.
  - **T**, zastavený (sTopped, suspended)
    - úmyselne, nedobrovoľne, používateľom, cez signál SIGSTOP,
    - po zaslaní signálu SIGCONT môže pokračovať.
  - **Z**, zombie, ukončený
    - nezaberá žiadne prostriedky len položku v tabuľke stránok,
    - rodič ešte neprečítal návratový kód (exit status).

Politika plánovača

Plánovanie procesov

- Úlohou plánovača (*scheduler*) je vybrať z radu pripravených procesov ten, ktorý sa bude vykonávať.
- Môže tiež určiť veľkosť prideleného časového kvanta (*time slice*).
- Plánovanie je transparentné.
- Proces nikdy “nevie” kedy a na ako dlho mu OS pridelí CPU.
- Proces nemôže robiť žiadne predpoklady o svojom časovaní.
  - Môže však využívať explicitnú synchronizáciu
    - s inými procesmi alebo externými udalosťami,
    - prostredníctvom prostriedkov OS.

Ciele plánovania

- Spravodlivosť medzi procesmi.
- Efektívne využitie prostriedkov, minimalizovať čas čakania (*ready*).
- Nízky čas odozvy pre interaktívne úlohy.
- Čo najkratší celkový čas vykonávania úlohy.
- Zvýšiť priepustnosť, teda celkový počet úloh ukončených za nejaký čas.
- Napríklad:
  - Proces by chcel aby všetok čas potrebný na dokončenie dostal naraz.
    - Bolo by to efektívne, zmizla by réžia prepínania, znížila by sa (jeho) doba vykonania.
    - Nebolo by to však spravodlivé, iné procesy by mali vysoký čas odozvy.
    - Ak by to bola dlhá úloha, celková priepustnosť by bola nízka.
- Ciele si vzájomne **odporujú**. Navyše aj samotné plánovanie vnáša réžiu.
- Ciele sa môžu meniť podľa určenia systému: interaktívne, dávkové, RT, ...

Politiky plánovača – bez preempcie

- First Come First Serve (FCFS)
  - Procesy dostanú pridelený procesor v poradí v akom prišli.
  - Procesor majú až kým ho neuvolnia (neskončia, alebo sa nezablokujú).
  - Rad pripravených je FIFO.
  - Môže spôsobovať dlhé čakanie, ak sa objavujú dlhé úlohy.
- Shortest Job First (SJF)
  - Plánovač musí vedieť, na ako dlho proces potrebuje CPU.
  - Dá sa odhadnúť napríklad podľa poslednej doby behu.
  - Dlhé úlohy môžu dlho čakať.

Politiky plánovača

- Round Robin (RR)
  - Každému pridelí rovnaké časové kvantum (10-100 ms) a striedajú sa v tom istom poradí. Využíva preempciu. Po uplynutí kvanta je procesor odobraný.
  - Ak je kvantum malé, dobrá odozva, ale je to neefektívne (veľká réžia).
  - Ak je veľké, zvyšuje sa využitie, ale doba odozvy procesov je tiež veľká.
    - Podobá sa na FCFS.
- Podľa priority
  - Vyberie sa proces s najvyššou prioritou.
  - Procesy s nízkou prioritou by sa však nemuseli dostať na rad → starvácia.
  - Priority sa preto môžu dynamicky meniť. Napríklad nepriamo úmerne spotrebovanému CPU času (*CPU usage* v PCB).

Prostriedky a zaťaženie

Zdroje použité procesom

- Operačný systém pre každý proces zaznamenáva objem prostriedkov ktoré používa.
- Napríklad, Linux `getrusage()`
  - Celkový čas kedy mal pridelený procesor a vykonával svoj program (*user CPU time*).
  - Celkový čas kedy vykonával systémové volania, teda kód jadra (*system CPU time*).
  - Veľkosť obsadenej fyzickej pamäte (RSS – *Resident Set Size*).
  - Veľkosť pamäte zdieľanej s inými procesmi (*shared*).
  - Veľkosť alokovanej dátovej časti pamäte (*data*).
  - Veľkosť alokovanej zásobníkovej časti pamäte (*stack*).
  - Počet zachytených signálov.
  - Počet dobrovoľných a nedobrovoľných prepnutí kontextu.
  - Počet výpadkov stránok.
- Pre niektoré z týchto prostriedkov tiež existujú limity, ktoré proces nemôže prekročiť.

Limity pre proces (Linux)

- Proces si môže nastaviť tzv. *soft limit* a to maximálne po tzv. *hard limit*.
- Hard limit si môže len znižovať.
- Administrátor (*root*) môže procesom nastavovať limity ľubovoľne.
- Limity sa dedia, respektíve zachovávajú, pri volaní `fork()` aj `exec()`.
- Pre limity je možné použiť systémové volania `getrlimit()`, `setrlimit()`, alebo v linuxe `prlimit()`.
- Napríklad:
  - Maximálna veľkosť *data*, *stack*, *RSS*, ale aj celého adresového priestoru.
  - Maximálna veľkosť prideleného CPU času (v sekundách).
  - Limity pre prioritu plánovania (*nice*).
  - Maximálny počet otvorených súborov, maximálna veľkosť súboru.

Zaťaženie procesorov

- Zaťaženie procesorov (*CPU load averages*)
  - Priemerná dĺžka radu pripravených a vykonávaných procesov.
  - Sleduje sa za posledných 1, 5 a 15 minút. Indikuje trendy.
- Hoci počet procesov bežne niekoľkonásobne (aj rádovo) prevyšuje počet CPU, záťaž nemusí byť vysoká.
  - Väčšina procesov často čaká, nepotrebuje CPU.
  - Napríklad interaktívny shell čaká na vstup z terminálu. Systémové procesy (démony, servery) čakajú na podnety alebo vstupy, ktoré majú spracovať.
- Ak je záťaž *L* vyššia než počet CPU *n*, systém je preťažený.
  - V priemere (*L – n)* procesov čaká, hoci sú pripravené sa vykonávať (nie sú blokované).
- Príkazy na sledovanie záťaže
  - `uptime`, `top`, `vmstat`, `/proc/loadavg`

Zaťaženie systému

- Zaťaženie procesorov
  - Priemerná dĺžka radu pripravených a vykonávaných procesov.
- OS Linux využíva na určenie vyťaženia systému aj ďalšie údaje.
- Zaťaženie systému (*system load averages*)
  - Započítavajú sa aj procesy v stave neprerušiteľného čakania (D).
  - Tieto čakajú na dokončenie V/V operácie, na dáta z disku, alebo na uvoľnenie zámku a pod.
  - Takáto metrika odzrkadľuje aj požiadavky procesov na iné prostriedky, nie len CPU.

Spotrebovaný čas

- Operačný systém pre každý proces uchováva tiež informácie o čase.
- Kedy bol process spustený (koľko bolo hodín).
  - Umožňuje určiť koľko času ubehlo od spustenia → *reálny čas*.
  - V tom je však započítaný aj čas kedy sa proces nevykonával (bol blokovaný, alebo nemal pridelený procesor).
- Ako dlho sa vykonával, ako dlho mu bol pridelený procesor pri vykonávaní jeho kódu → *CPU user time*.
  - Na jednoprocesorovom systéme je vždy menší než reálny čas.
  - Pokiaľ má proces viacero vlákien, ich časy sa spočítajú.
  - Na viacprocesorovom systéme teda môže byť CPU čas väčší, než reálny čas ktorý ubehol počas jeho vykonávania.
- Ako dlho tento proces vykonával kód jadra → *CPU system time*.
  - V podstate ide o réžiu a chceme, aby bol čo najmenší.

Vytvorenie nového procesu

Vytvorenie procesu

- Nový proces môže byť vytvorený
  - po zavedení nového programu zo súboru do pamäte,
  - alebo len pre program, ktorý už v pamäti je.
- Prvý prístup využívajú napríklad systémy Windows
  - CreateProcess(...), môže mať až 10 parametrov, medzi nimi sa zadáva aj názov vykonateľného súboru a jeho argumenty.
  - posix_spawn(), implementácia tohto mechanizmu v systémoch podľa štandardu POSIX.
- Druhý prístup tradične využívali unixové systémy
  - systémové volanie fork() vytvorí kópiu volajúceho procesu,
  - Jednoduché na použitie, žiadne parametre. Výhodné ak sa majú prichádzajúce požiadavky spracovávať súbežne (napr. rôzne servery).
  - Jednoduchá implementácia v jadre. Alokovať PCB, skopírovať mapovanie pamäte, nastaviť copy-on-write bit pre stránky nového procesu. Vyžaduje však podporu od hardvéru (MMU).
  - fork() nasledovaný exec() plní funkciu CrreateProcess(). Opačne je to zložitejšie.

Mechanizmus fork() + exec() + wait()

- Systémové volanie `fork()` vytvorí kópiu volajúceho procesu, až na pár vínimiek, najmä PID.
- Volanie má v novovytvorenom procese návratovú hodnotu 0.
- Rodičovskému procesu vráti PID nového procesu, alebo -1 v prípade chyby.
- Býva bežné, že nový proces chce vykonávať iný program.
- Obraz (nového) procesu v pamäti môže byť nahradený obsahom vykonateľného súboru z disku volaním `exec()`.
- Proces (potomok) sa môže ukončiť volaním `exit()`, ktoré má ako argument číselný návratový kód.
- Rodičovský proces typicky čaká na ukončenie potomka volaním `wait()`, respektíve `waitpid()`, ktoré vráti návratový kód ukončeného procesu.

Mechanizmus fork() + exec() + wait()

```
while (1) {
    write_prompt();
    read_command(command, argv);
    child_pid = fork();
    switch (child_pid) {
        case -1: perror("fork"); break;
        case 0: /* child process */
            execve(command, argv);
            perror("execve"); break;
        default: /* parent process */
            waitpid(child_pid, &status, 0);
    }
}
```

Hierarchia procesov (UNIX)

- Prvý proces v systéme (tradične `init`, PID 1) vytvorí jadro pri štarte systému.
- Všetky ďalšie procesy vznikajú volaním `fork()` ktoré rodičovi vráti PID vytvoreného potomka.
- Proces sa ukončí (dobrovoľne) volaním `exit()`, ktorého argument je jeho návratový kód (typicky 0 OK, inak kód chyby).
- Každý proces má svojho rodiča
  - Rodič by mal čakať na ukončenie potomkov a prečítať návratový kód volaním `wait()`.
  - Ak rodič skončí skôr ako potomok, jeho novým rodičom sa stane `init`.
- Proces môže byť ukončený iným procesom zaslaním signálu volaním `kill()`.
- Systémy Windows neuchovávajú pre procesy informácie o rodičoch a potomkoch.

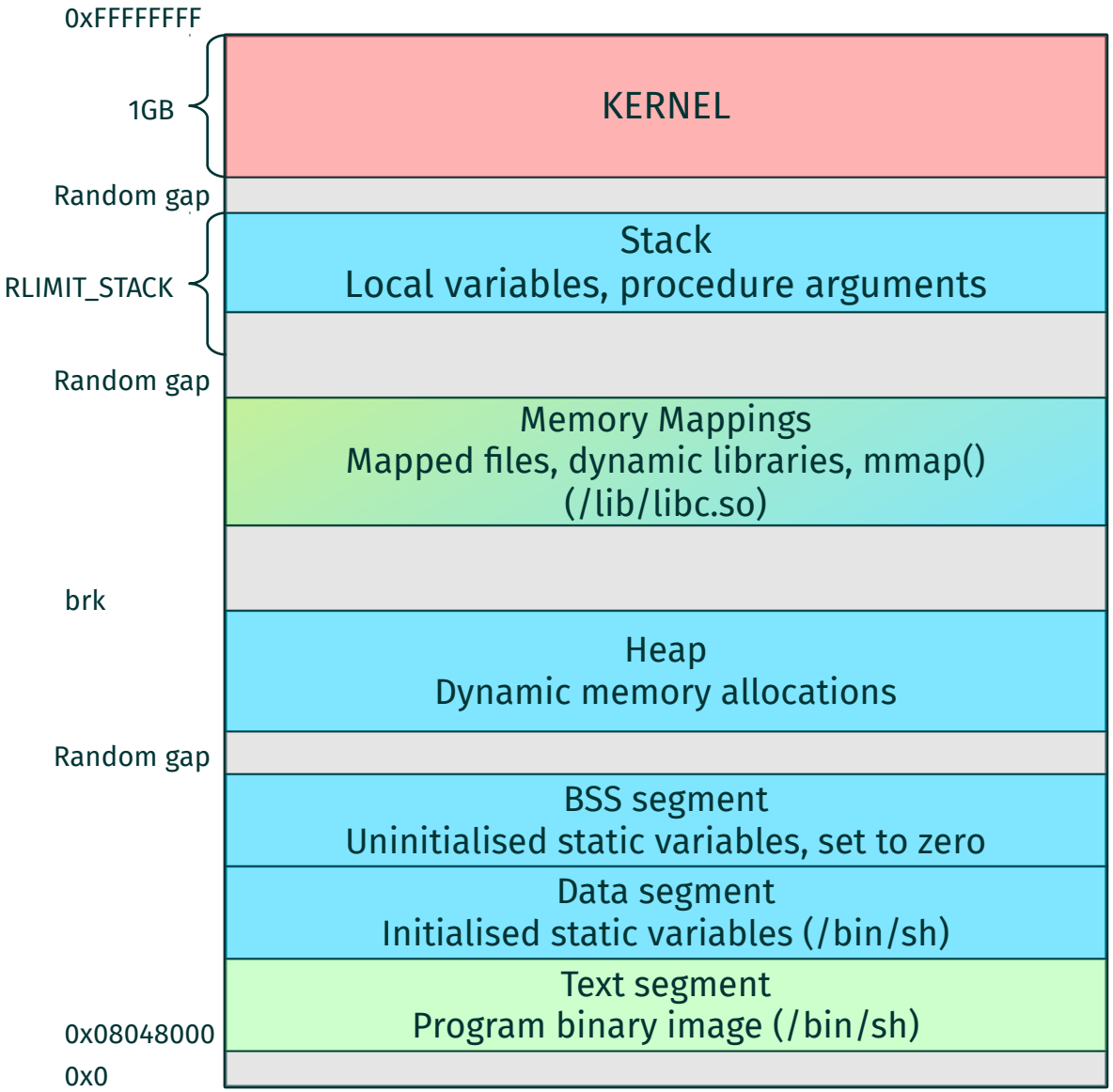
Zavedenie nového programu

```
int execve(const char *filename, char *const argv[],
char *const envp[])
```

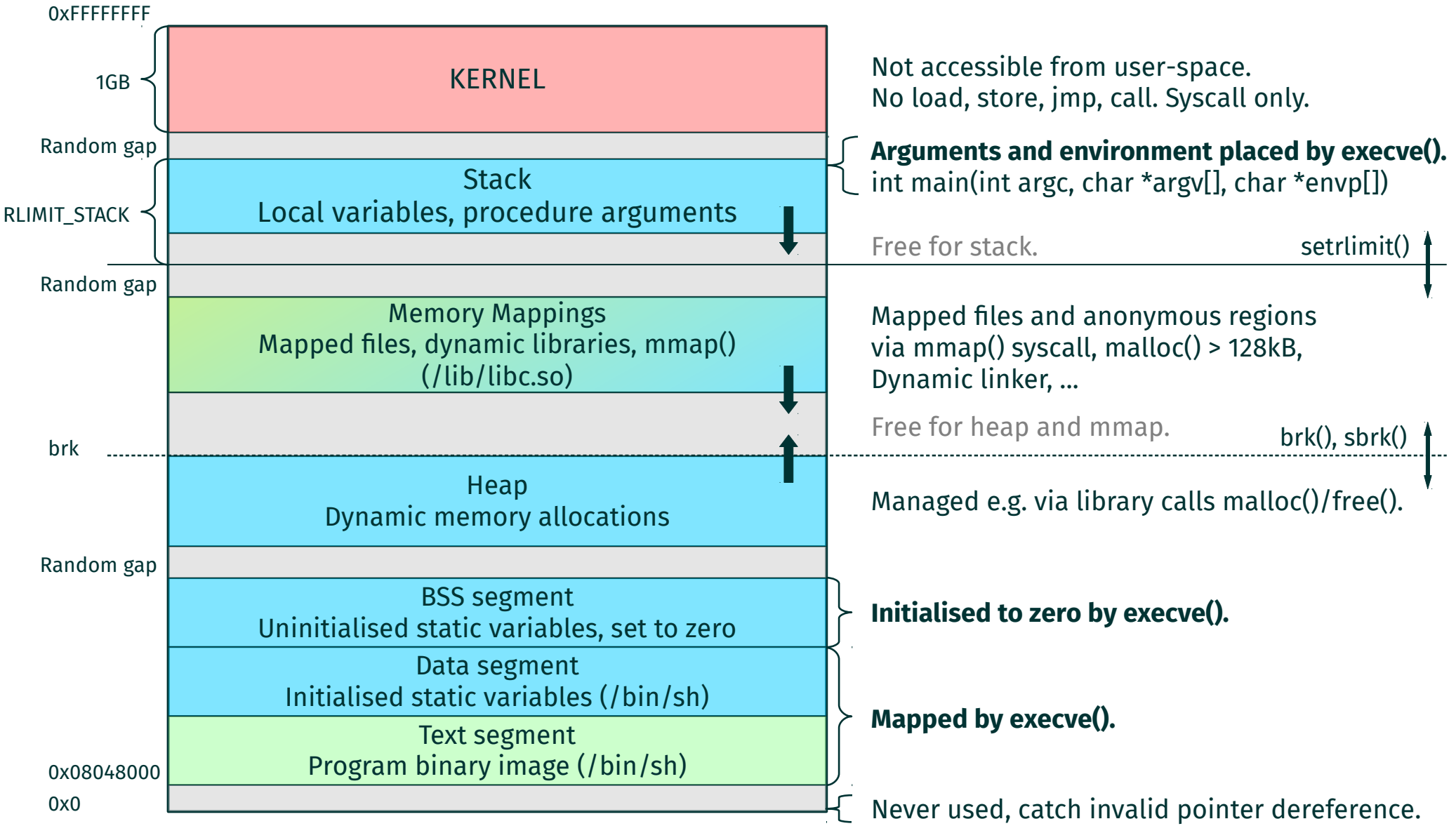
- Systémové volanie:
  - Nahradí obraz volajúceho procesu v pamäti obsahom súboru `filename`.
  - Proces bude mať prístup k reťazcom argumentov `argv` a premenných prostredia `envp`.
- Volanie môže vrátiť chybový kód ak niečo zlyhá.
- V prípade úspechu sa nevráti.
  - Tok riadenia sa nevráti do pôvodného programu, lebo vykonávanie pokračuje v novozavedenom programe, v jeho `main()`.
- Text, statické dáta, zásobník aj ostatné časti budú prepísané.

Mapa pamäti

Mapa pamäti procesu



Mapa pamäti procesu



Mapa pamäti procesu

```
$ cat /proc/self/maps
00110000-00111000 r-xp 00110000 00:00 0          [vdso]
00b62000-00b7d000 r-xp 00000000 fd:01 457870     /lib/ld-2.7.so
00b7d000-00b7e000 r--p 0001a000 fd:01 457870     /lib/ld-2.7.so
00b7e000-00b7f000 rw-p 0001b000 fd:01 457870     /lib/ld-2.7.so
00b81000-00cd4000 r-xp 00000000 fd:01 457871     /lib/libc-2.7.so
00cd4000-00cd6000 r--p 00153000 fd:01 457871     /lib/libc-2.7.so
00cd6000-00cd7000 rw-p 00155000 fd:01 457871     /lib/libc-2.7.so
00cd7000-00cda000 rw-p 00cd7000 00:00 0
08048000-0804d000 r-xp 00000000 fd:01 981130     /bin/cat
0804d000-0804e000 rw-p 00004000 fd:01 981130     /bin/cat
08629000-0864a000 rw-p 08629000 00:00 0          [heap]
b7cc9000-b7cca000 r--p 02a6d000 fd:01 1874928    /usr/lib/locale/locale-archive
b7cca000-b7cd0000 r--p 02a5a000 fd:01 1874928    /usr/lib/locale/locale-archive
b7cd0000-b7d07000 r--p 02a1e000 fd:01 1874928    /usr/lib/locale/locale-archive
b7d07000-b7d08000 r--p 021a8000 fd:01 1874928    /usr/lib/locale/locale-archive
b7d08000-b7d44000 r--p 0214f000 fd:01 1874928    /usr/lib/locale/locale-archive
b7d44000-b7f44000 r--p 00000000 fd:01 1874928    /usr/lib/locale/locale-archive
b7f44000-b7f46000 rw-p b7f44000 00:00 0
bfe4a000-bfe5f000 rw-p bffeb000 00:00 0          [stack]
```

Mapa pamäti procesu

```
$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 103:03 1610303                           /usr/bin/cat
0060b000-0060c000 r--p 0000b000 103:03 1610303                           /usr/bin/cat
0060c000-0060d000 rw-p 0000c000 103:03 1610303                           /usr/bin/cat
014db000-014fc000 rw-p 00000000 00:00 0                                  [heap]
7fadc7432000-7fadcd975000 r--p 00000000 103:03 1641793                   /usr/lib/locale/locale-archive
7fadcd975000-7fadcdb39000 r-xp 00000000 103:03 1578207                   /usr/lib64/libc-2.17.so
7fadcdb39000-7fadcdd38000 ---p 001c4000 103:03 1578207                   /usr/lib64/libc-2.17.so
7fadcdd38000-7fadcdd3c000 r--p 001c3000 103:03 1578207                   /usr/lib64/libc-2.17.so
7fadcdd3c000-7fadcdd3e000 rw-p 001c7000 103:03 1578207                   /usr/lib64/libc-2.17.so
7fadcdd3e000-7fadcdd43000 rw-p 00000000 00:00 0
7fadcdd43000-7fadcdd65000 r-xp 00000000 103:03 1608723                   /usr/lib64/ld-2.17.so
7fadcdf3a000-7fadcdf3d000 rw-p 00000000 00:00 0
7fadcdf63000-7fadcdf64000 rw-p 00000000 00:00 0
7fadcdf64000-7fadcdf65000 r--p 00021000 103:03 1608723                   /usr/lib64/ld-2.17.so
7fadcdf65000-7fadcdf66000 rw-p 00022000 103:03 1608723                   /usr/lib64/ld-2.17.so
7fadcdf66000-7fadcdf67000 rw-p 00000000 00:00 0
7ffdf288d000-7ffdf28ae000 rw-p 00000000 00:00 0                          [stack]
7ffdf2923000-7ffdf2926000 r--p 00000000 00:00 0                          [vvar]
7ffdf2926000-7ffdf2927000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Vlákna

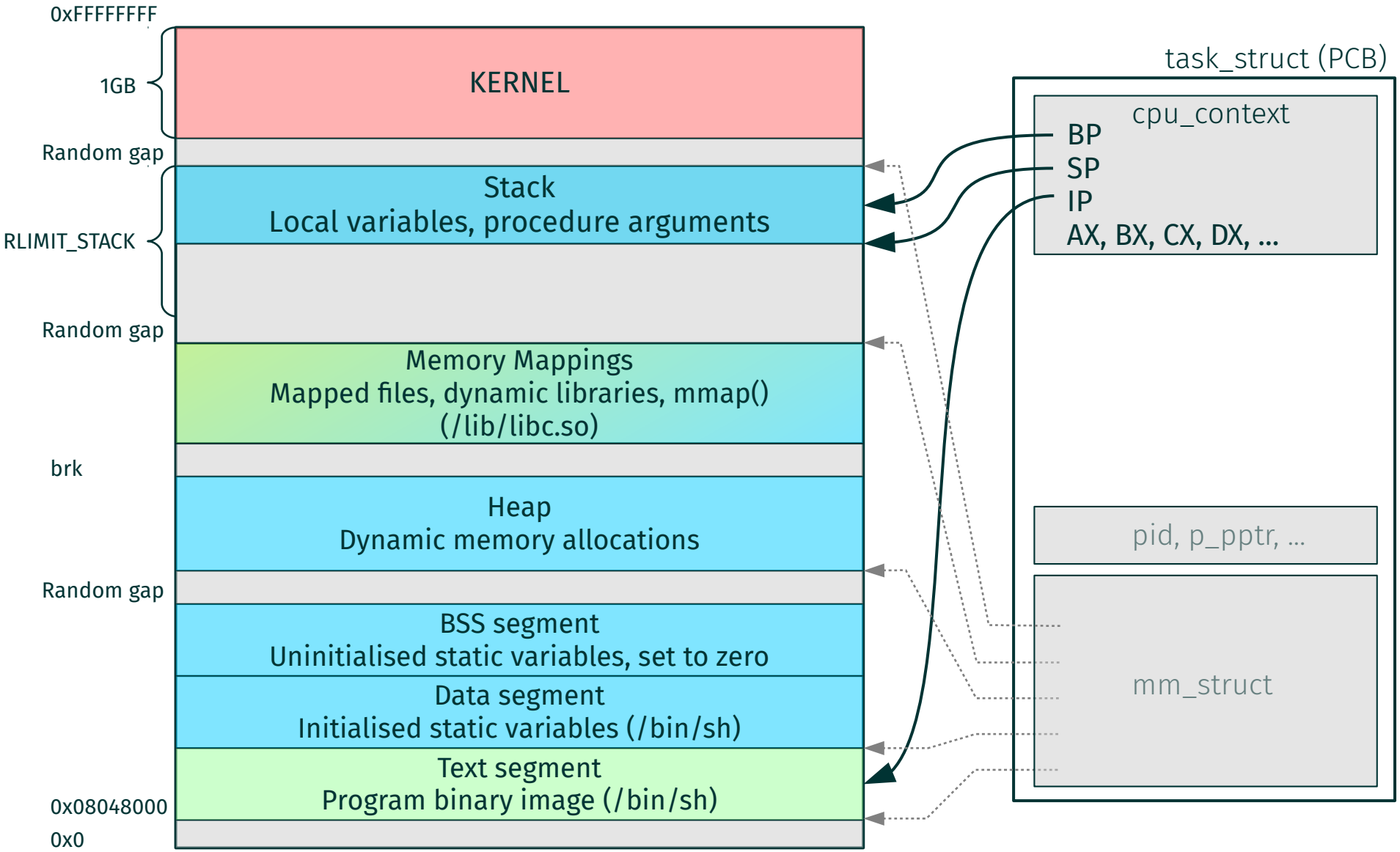
Podpora vlákien

- Vlákno (*thread*) predstavuje samostatný tok riadenia vrámci procesu.
- Vlákna sa môžu vykonávať súbežne v jednom pamäťovom priestore.
- Vlákna môžu byť implementované rôznym spôsobom, napríklad aj bez podpory OS (*user-level threads*).
  - V tomto prípade OS plánuje len procesy a proces prepína medzi viacerými funkciami (podobne ako jadro prepína vykonávanie procesov na jednom CPU).
  - Toto prepínanie je omnoho rýchlejšie (réžia volania procedúry), než keby preplánovanie robilo jadro (réžia systémového volania, vrátane prepnutia kontextu).
  - Zablokovanie jedného vlákna však zablokuje všetky. Nedokáže využiť viac fyzických CPU.
- Vlákna s podporou OS plánuje jadro (*kernel-level threads*).
- Iné modely:
  - JAVA threads, mapujú sa na vlákna OS.
  - PERL (*interpreter-based threads*), každé vlákno má vlastný interpreter, čiže samostatný proces. Nič sa automaticky nezdieľa.

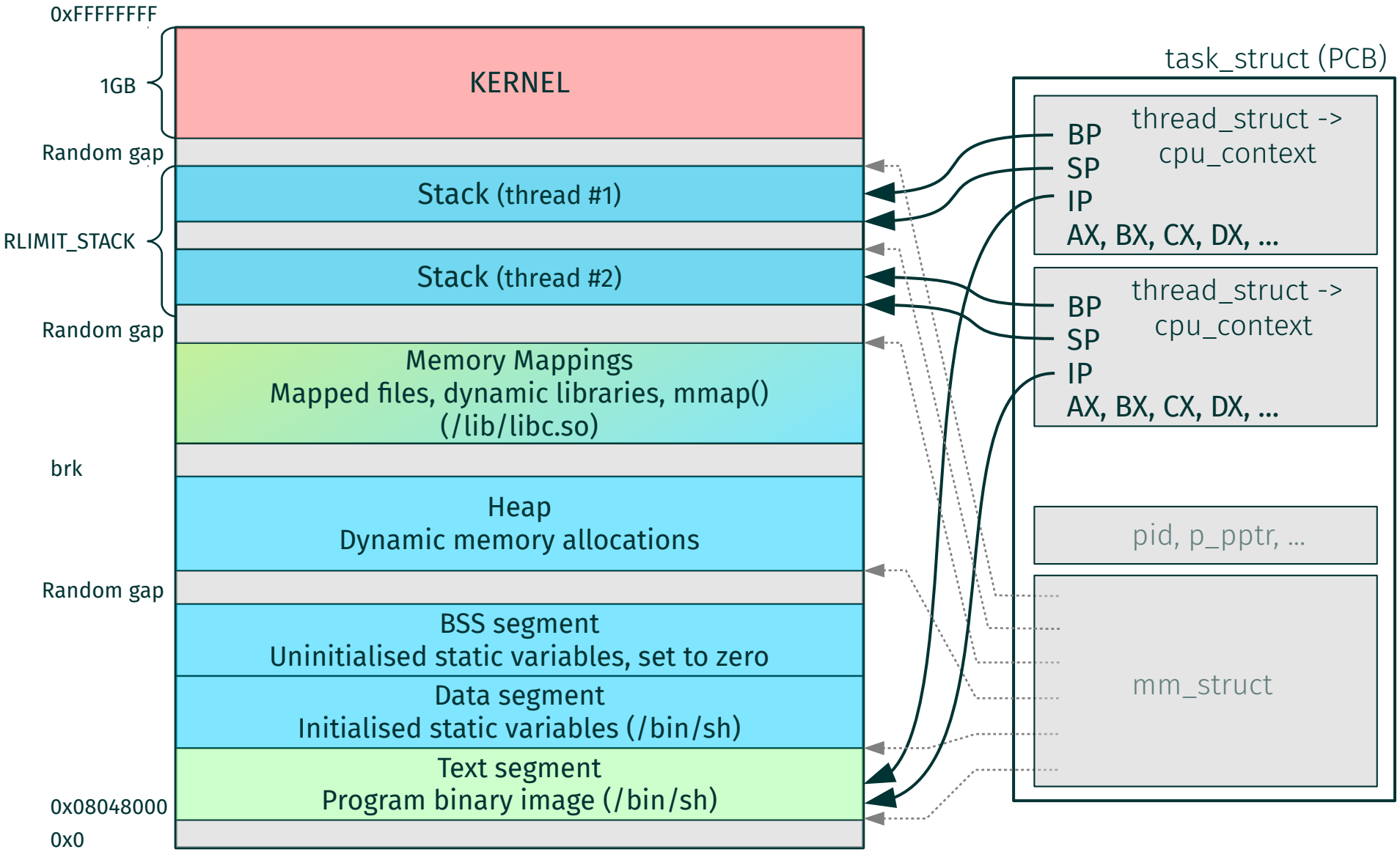
Vlákna a prostriedky procesu

- Všetky vlákna procesu zdieľajú väčšinu jeho prostriedkov, vrátane pamäťového priestoru.
  - Na rozdiel od procesov vlákna (patriace jednému procesu) môžu priamo využívať spoločné premenné (treba však ošetriť synchronizáciu).
  - Vidia tiež napríklad všetky otvorené súbory a podobne.
- Čo potrebuje OS na implementovanie vlákna?
- Čo by bolo treba pridať, aby OS mohol súbežne začať vykonávať ďalšiu funkciu (toho istého procesu)?
  - **Registre** (IP, SP, BP, ...)
  - Nový **zásobník** v adresovom priestore procesu.
  - Plánovacie informácie (stav: *ready*, *running*, *blocked*)

Stav procesu



Stav a prostriedky vlákna



Implementácia vlákien v OS

- Linux implementuje vlákna ako procesy.
  - Každý proces má v PCB zoznam vlákien (`task_struct -> thread_struct`, ...)
  - Každé vlákno má aj vlastnú `task_struct`, pričom štruktúry `task_struct` predstavujúce vlákna jedného procesu viaceré položky zdieľajú (`mm`, `files`, `signal`, ...).
- Pri volaní `clone()` je možné špecifikovať čo všetko sa má kopírovať a čo sa bude zdieľať.
  - Pri vytváraní procesu `fork()` skopíruje všetky prostriedky rodiča (`task_struct`).
  - Pri vytváraní vlákna volaním `clone()` sa prostriedky ktoré bude vlákno zdieľať kopírovať nemusia.
- Vytvorenie vlákna, je teda menej náročné, než vytvorenie procesu.
- Podobne **prepínanie kontextu** (vlákien jedného procesu) je efektívnejšie.
  - Stále je však omnoho pomalšie než *user-level threads*, kde to ide bez prepínania kontextu.
- Štandardná implementácia: POSIX threads, (pthreads.h, pthread_create(), ...).

Problémy pri implementácii vlákien

- Proces združuje prostriedky (pamäť, súbory, ...), ktoré sú pre vlákna spoločné.
- Vlákno predstavuje tok riadenia – plánovač prideľuje procesor vláknam.
- Z toho vyplýva viacero problémov.
  - Ktoré vlákno má obslúžiť signál doručený pre proces?
    - Len jedno? Ktoré? Všetky?
  - Ktoré vlákno urobí alokáciu (spoločnej) pamäte?
  - Môže niektoré vlákno zatvoriť súbor ak ho iné ešte číta/zapisuje?
  - Globálne premenné (napr. chybový kód po systémovom volaní, `errno`).
- Knižničné volania musia byť prispôsobené pre súbežné vykonávanie.
- Programy pre prostredie s vláknami musia byť písané s ohľadom na **súbežné vykonávanie v spoločnej pamäti** – *thread safety*.
- Väčšina úloh sa dá vyriešiť bez použitia vlákien.

POSIX threads

Vytvorenie a ukončenie vlákna

- `man pthread.h`
- `#include <pthread.h>`
- Vytvorenie vlákna:
  - `int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);`
  - vytvorí nové vlákno, jeho identifikátor uloží do `thread`, začne vykonávať funkciu `start_routine` s argumentom `arg`,
  - vlastnosti vlákna pri vytvorení je možné ovplyvňovať cez `attr`, môže byť NULL (použijú sa "default" hodnoty).
- Ukončenie vlákna:
  - `void pthread_exit(void *value_ptr)` - ukončí volajúce vlákno,
  - vlákno čakajúce na jeho ukončenie môže získať ukazovateľ `value_ptr` (návratová hodnota),
  - implicitne sa zavolá po ukončení `start_routine`, riadenie sa nevráti volajúcemu vláknu.
  - `int pthread_cancel(pthread_t thread)` - ukončenie (iného) bežiaceho vlákna,
  - volajúce vlákno nečaká na ukončenie (to je asynchrónne).

Synchronizácia pri ukončení

- Čakanie na ukončenie iného vlákna:
  - `int pthread_join(pthread_t thread, void **value_ptr);`
  - volajúce vlákno sa zablokuje, až kým zadané vlákno `thread` neskončí (nezavolá `pthread_exit()`),
  - ak hodnota ukazovateľa `value_ptr` nie je NULL, nastaví sa na príslušnú hodnotu končiaceho vlákna (návratová hodnota).
- Uvolnenie vlákna:
  - `int pthread_detach(pthread_t thread);`
  - spôsobí, že prostriedky pre uchovanie daného vlákna `thread` sa uvolnia hneď po jeho ukončení a nie až po volaní `pthread_join()`,
  - na každé vlákno by sa malo zavolať `pthread_join` alebo `pthread_detach`,
  - vlákno je možné vytvoriť s atribútom `detachstate`.

Synchronizácia vlákien

- Zámok (binárny semafor):
  - `pthread_mutex_init()`, `pthread_mutex_destroy()`,
  - `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_mutex_trylock()`
- argument: `pthread_mutex_t *mutex`
- Podmienené premenné:
  - `pthread_cond_init()`, `pthread_cond_destroy()`
  - `pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex)`,
- `pthread_cond_signal()`,
- `pthread_cond_broadcast()`

To je zatiaľ všetko