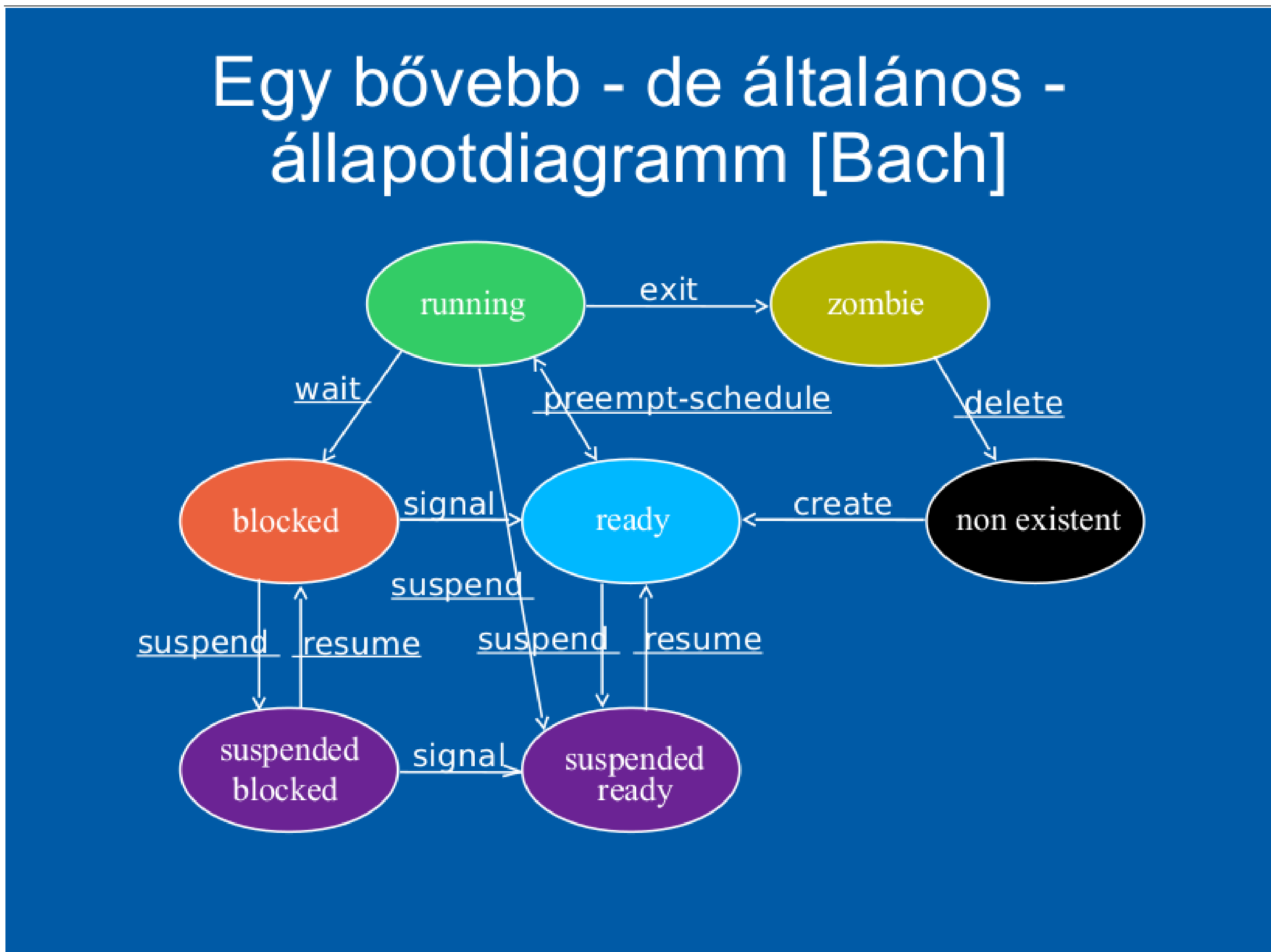
4. Gyakorlat

Processz:

- A processz egy végrehajtási példánya egy párhuzamosságot nem tartalmazó végrehajtható programnak.
- A processz egy futó program példány.
- A klasszikus processz egy szálon fut, a benne futó végrehajtható program nem tartalmaz párhuzamosságot.

Processz állapotok, állapotátmenetek:



Processzek listázása:

- A rendszerben futó processzek kilistázhatók például a **ps** program segítségével. A program paraméter nélküli meghívásával az eredmény csupán néhány processz, amelyek az aktuális terminálhoz (TTY) vannak rendelve (pl.: bash, és maga a ps; ezeknél egyértelműen több kell, hogy fusson), de a paraméterezésétől függően különböző részletességgel és formázással képes megjeleníteni a processz listát.
- A fontosabb kapcsolók a következők:
  - ps – az aktuális terminálhoz rendelt processzek,
  - ps **a** – minden olyan processz, amihez van terminál rendelve,
  - ps **w** – hosszú sorok tördelése,
  - ps **u** – felhasználóorientált megjelenítés,

- ps **x** – minden processz, azok is, amikhez nem tartozik terminál,
- ps **r** – csak a futó állapotban lévő processzek megjelenítése,
- ps **-e** vagy ps **-A** – minden processz (valójában ugyanaz, mint a „ps x”),
- ps **-f** – formázott megjelenítés, kicsit részletesebb, mint a „ps” önmagában,
- ps **-o** mező1,mező2,... – csak a megadott mezők jelennek meg,
- ps **-p** pid – adott pid-del rendelkező processz megjelenítése (p, -q és q kapcsoló is),
- Egyéb kapcsolókért és részletekért, lásd: **man ps**
- Azok a kapcsolók, amelyek előtt nincs kötőjel úgynevezett „BSD stílusú” formázást használnak.

- Mit látunk? Mit jelent az output? (Az alábbiak a **ps awxu** végrehajtásakor láthatók.)
  - USER: a processz tulajdonosának a neve,
  - PID: a processz azonosítója,
  - %CPU: a processz CPU használata az elmúlt időablakban,
  - %MEM: a processz aktuális memóriahasználata,
  - VSZ: a processzhez tartozó virtuális memória mérete,
  - RSS: a processz által ténylegesen használt memória mérete,
  - TTY: melyik terminálhoz tartozik a processz,
  - STAT: a processz állapota,
  - START: mikor indult a processz,
  - TIME: az eddig elhasznált CPU idő,
  - COMMAND: a processz „parancs” neve.
- Processz monitorozó segédprogramok: top, htop, stb.

UNIX API, C függvény-, illetve rendszerhívások:

- \- Mi a különbség egy hagyományos függvényhívás és egy rendszerhívás között?
  \- ???
  \- A rendszerhívások CPU módváltással járnak (trap), segítségükkel külső programból igénybe vehetők az OS különböző, alacsony szintű szolgáltatásai (pl.: I/O kezelés).
- A rendszerhívások különböző osztályokba sorolhatók, például:
  - Processz menedzsment,
  - Fájl, jegyzékek,
  - Informálódó, beállító,
  - Eszköz manipuációk.
- **fork:** A fork() rendszerhívással lehet gyermekprocesszt létrehozni, amely a szülőjével (és más egyéb processzekkel) párhuzamosan fog futni. A fork() visszatérési értéke egy integer, ami alapján megkülönböztethető a szülő és gyermek processz. A gyerek processzben a visszatérési érték 0, a szülőben pedig a gyerek processz PID-je. A negatív visszatérési érték a rendszerhívás sikertelenségét jelzi. (Lásd: **man 2 fork**)
- **getpid:** visszatérési értéke a hívó processz PID-je.
- **getppid:** visszatérési értéke a hívó processz PPID-je, azaz a szülőprocesszének a PID-je.
- **exec:** Az exec rendszerhívás családba tartozó hívásokkal lehet egy programot végrehajtani. A processz PID-je nem változik meg, a hívó processz helyére töltődik be a meghívott program. Ilyen hívások például: execv, execve, execvp, execle, stb. (Lásd: **man exec**)
- **wait:** Várakozás processz állapotváltozására. A gyermekprocessz csak azt követően szűnik meg, miután a szülő meghívta rá a wait()-et. (Lásd: **man 2 wait**)
- **exit:** Processz terminálása. Minden gyerekprocessz PPID-je 1-re állítódik. A processz szülője kiolvashatja az állapotot. (Lásd: **man exit**)

- **WEXITSTATUS:** Ez egy makró, ami a gyerekprocessz állapotának kiolvasására szolgál. Van több különböző lehetőség, lásd: http://www.gnu.org/software/libc/manual/html_node/Process-Completion-Status.html

Feladat 1:

Eddigi ismereteinket felhasználva írjunk programot, amely véletlenszerűen generál 10.000 darab 1 és 10.000 közötti egész számot és 10 szálra bontva meghatározza, hogy összesen hány darab prímszám van a generált számok között.

Feladat 2:

Eddigi ismereteinket felhasználva írjunk két programot, amelyek a következő feladatokat látják el:

- Az egyik bekér egy 0 és 255 közé eső egész számot a felhasználótól, majd visszatér a beolvasott számmal (exit status).
- A másik generál egy 0 és 255 közé eső egész számot, amit a felhasználónak ki kell találnia. A felhasználó tippjeinek beolvasásához az előzőleg írt programunkat kell használni!