A kihívás

A kihívás két részből áll. Az első részben a csapatok külön-külön dolgoznak, majd a második részben a csapatok által írt programok egymással „versenyeznek” az alábbiak alapján.

A kihívásban programok próbálják meg elpusztítani egymást egy virtuális számítógép szimulált memóriájában. A programokat speciális programozási nyelven kell megírni. A programozási nyelv és a virtuális számítógép sokkal egyszerűbbek a valóságnál. A virtuális számítógép egy memóriából (későbbiekben aréna) áll, amely lényegében egy utasítástömb. Alapesetben a tömb - a későbbiekben tárgyalt - DAT utasítást tartalmazza a következő formában: DAT #0, #0. A szimuláció indulása során a harcoló programok ebbe a tömbbe másolódnak be, véletlen pozícióba, és a program utasításai ekkor felülírják ezt az alapértéket. Az aréna fix méretű (a konkrét adatokat „A futtatás paraméterei” bekezdésben tárgyaljuk), de körkörös, vagyis az utolsó memóriahely után ismét az első jön. Valójában a programnak nem kell tudnia, hol van a tömb vége, mivel nincsenek abszolút címek. Így a zérus (0) cím nem a memória első utasítását jelenti, hanem magát az aktuális utasítást. A következő utasítás az 1-es címen található, az előző utasítás pedig a -1–es címen.

Az alapegység tehát nem byte, hanem az utasítás. Egy utasítás három részből áll, egy műveletből (opcode), egy forrás címből (A mező) és egy cél címből (B mező). Egy utasítás oszthatatlan egység. Az utasításokból létrejövő harcos program végrehajtása szintén igen egyszerű. A szimulátor egy időben egyetlen utasítást hajt végre, majd a következő memóriacímen található utasítást, kivéve, ha szándékosan máshova ugrottunk. Ha egynél több program van a memóriában (vagyis az arénában), akkor utasításonként váltogat a szimulátor a programok között. Több program esetén először az első program első utasítását hajtja végre a szimulátor, majd a második program első utasítását, majd a harmadikét, stb., amíg van másik program. Ha az összes program első utasítását végrehajtotta, akkor ismét az első programmal folytatja, és végrehajtja annak második utasítását, és így tovább. Ha valamelyik programnak kevesebb utasítása van, mint a másiknak, akkor legközelebb, amikor ismét ő van soron, az ő első utasítása hajtódik végre. Minden utasítás végrehajtása azonos ideig tart! További részletek a végrehajtásról a „Programvégrehajtás” bekezdésben található.

A következő utasítások állnak rendelkezésre:

- DAT – adat, ha ezt végrehajtjuk, az a program halálát jelenti
- MOZ – adat mozgatása az A mező által megadott címről a B mező által megadott címre
- ADD – összeadja az A és B mezőket, az eredményt a B mezőben tárolja el
- SUB – kivonja az A mezőt a B mezőből, az eredményt a B mezőben tárolja el
- MUL – összeszorozza az A és B mezőket, az eredményt a B mezőben tárolja el
- DIV – elosztja a B mezőt az A mezővel, az eredményt a B mezőben tárolja el, ha A nem volt zérus
- MOD – maradékos osztás (modulo), elosztja a B mezőt az A mezővel, a maradékot a B mezőben tárolja el, ha A nem volt zérus
- JMP – ugrás, a program végrehajtása az A mezőben megadott címen folytatódik
- JMZ – a program végrehajtása az A mezőben megadott címen folytatódik, ha a B mező zérus volt

- JMN – a program végrehajtása az A mezőben megadott címen folytatódik, ha a B mező nem volt zérus
- DJN – csökkenti a B mező értékét eggyel, és ha a B mező nem volt zérus, akkor a program végrehajtása az A mezőben megadott címen folytatódik
- FRK – (fork) egy új program elindítása az A mezőben megadott címen
- CMP – a következő utasítás átugrása, ha az A mező értéke egyenlő a B mező értékével
- SLT – a következő utasítás átugrása, ha az A mező értéke kisebb, mint a B mező értéke

A programozási nyelv nem érzékeny a kis- és nagybetűre!

Néhány példa

Nézzünk néhány példát. A példákban a #0 magát a zérus számot jelenti. Ha a szám előtt nincs semmilyen jel, akkor viszont a szám egy relatív címet jelent, így a zérus az aktuális utasításra hivatkozik, az egyes érték a következő utasításra, és így tovább.

1. A legegyszerűbb program:

```
MOZ 0, 1
```

Ennyi. Mit csinál? Nagyon egyszerű, ez az utasítás átmásol egy utasítást. Fentebb már jeleztük, hogy csak relatív címek vannak, így ez a program önmagát másolja le. Nézzük a végrehajtást. Az aktuálisan végrehajtott parancs szürke háttérrel kerül kiemelésre Ez a kiindulási állapot:

```
MOZ 0, 1
DAT #0, #0
DAT #0, #0
DAT #0, #0
…
```

A program átmásolja önmagát:

```
MOZ 0, 1
MOZ 0, 1
DAT #0, #0
DAT #0, #0
…
```

majd a következő utasítás kerül végrehajtásra:

```
MOZ 0, 1
MOZ 0, 1
DAT #0, #0
DAT #0, #0
…
```

ami megint átmásolja önmagát

```
MOZ 0, 1
MOZ 0, 1
MOZ 0, 1
```

```
DAT #0, #0
…
```

és így tovább. Ilyen módon a program betölti az arénát. Mivel az arénának nincs vége, hiszen körkörös, így a program lényegében örökké fut önmagában. Az is látható, hogy a program saját kódját hozza létre. Ez a fajta önmódosítás inkább szabály lesz, mint kivétel. Ha már az önmódosításról van szó, fontos megjegyezni, hogy nincs cache (vagyis nem tudja előszedni a korábban belekerült utasítást), így mindig az aktuális állapotot tudjuk módosítani, illetve az aktuális utasítás a saját végrehajtását nem tudja megváltoztatni.

Ezzel a programmal az a baj, hogy gyakran az ellenfél kódját is felülírja, ami ezután szintén ugyanilyen módon kezd el viselkedni, és így döntetlen lesz a verseny.

2. Egy bombázó

Nézzünk egy bombázó programot, ami DAT utasításokkal írja felül a memória helyeit. A jobb érthetőség kedvéért azt is feltételezzük, hogy most az aréna mérete 20 utasítás lesz. A program maga három utasításból áll, és legyen az aréna állapota a következő. A kezdő utasítás megint szürke háttérrel van kiemelve:

```
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
ADD #4, 3
MOZ 2, @2
JMP -2
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
```

(A további lépéseket oldalanként mutatjuk be, hogy az aréna egyben legyen látható mindig.)

A program egy összeadással kezdődik, az ADD utasítással. Az utasítás a 4-es számot hozzáadja a 3 címmel odébb található utasításhoz (lényegében a B mezőhöz), így az eredmény ez lesz, amit vastagon is szedtünk:

```
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
ADD #4, 3
MOZ 2, @2
JMP -2
DAT #0, #4
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
```

(Más esetben, amikor két utasítást adunk össze, akkor a program összeadja a két utasítás A és B mezőjét is! De erre még visszatérünk.)

A MOZ utasítás egy másik címzési módot mutat, az indirekt címzést. A MOZ utasítás – az A mező alapján – a hozzá képest 2-es címen lévő DAT utasítást másolja, de nem önmagára, hanem a két utasítással odébb levő DAT utasítás B mezője által mutatott címre (ezt adja meg @2 jelölés, amit később tárgyalunk), vagyis a DAT utáni negyedik címre. A végrehajtás után ez lesz az eredmény:

```
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
ADD #4, 3
MOZ 2, @2
JMP -2
DAT #0, #4
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #4
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
```

Lényegében tehát a DAT utasítás 4 címmel odébb lesz másolva.

A következő utasítás az ugró utasítás, és ez két címmel ugrik vissza, vagyis az ADD utasításra. Ennek végrehajtás után ez lesz az aréna állapota:

```
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
ADD #4, 3
MOZ 2, @2
JMP -2
DAT #0, #4
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #4
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
```

Az ADD utasítás végrehajtása után ilyen lesz az aréna állapota. A vastagon kiemelt értéket fogja módosítani az ADD utasítás:

```
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
ADD #4, 3
MOZ 2, @2
JMP -2
DAT #0, #8
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #4
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
```

Majd a MOZ utasítás átmásolja a DAT utasítást 8 utasítással odébb. A módosult DAT utasítás vastagon van szedve. Ha ezen a helyen egy másik program utasítása van, és az a másik program megpróbálja ezt végrehajtani, akkor az a másik program leállna/meghalna. Így az a másik program le lett bombázva:

```
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
ADD #4, 3
MOZ 2, @2
JMP -2
DAT #0, #8
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #4
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #8
DAT #0, #0
DAT #0, #0
```

A rövidség miatt, most feltételezzük, hogy a JMP, majd egy újabb ADD és MOZ utasítások lettek végrehajtva. Mivel az aréna körkörös, ezért a végén nem ér véget, hanem az elejétől folytatódik. Az aréna melletti nyíl ezt jelzi. Illetve a sor végén található számok is csak a megértést segítik és nem a kód részei.

DAT #0, #0	11
DAT #0, **#12**	12
DAT #0, #0	
DAT #0, #0	
DAT #0, #0	
DAT #0, #0	
ADD #4, 3	
MOZ 2, @2	
JMP -2	
DAT #0, #12	
DAT #0, #0	1
DAT #0, #0	2
DAT #0, #0	3
DAT #0, #4	4
DAT #0, #0	5
DAT #0, #0	6
DAT #0, #0	7
DAT #0, #8	8
DAT #0, #0	9
DAT #0, #0	10

Egy újabb JMP, ADD és MOZ utasítások után pedig az aréna állapota ez lenne.

```
DAT #0, #0
DAT #0, #12
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #16
ADD #4, 3
MOZ 2, @2
JMP -2
DAT #0, #16
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #4
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #8
DAT #0, #0
DAT #0, #0
```

Itt viszont figyelembe kell venni azt, hogy valójában minden címnek a *0* és *aréna méret*-1 tartományba kell esnie. Ez azt jelenti, hogy ha megadunk egy címet, akkor az értéket elosztjuk az aréna méretével és csak a maradékot tartjuk meg. Jelen esetben mivel az utolsó bombázásnál 16-os címet használtunk, az valójában zérus lesz. Így a program úgy fog kinézni, ahogy a következő oldal mutatja.

A program tényleges állapota, mely figyelembe veszi a körkörös címzést.

```
DAT #0, #0
DAT #0, #12
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
ADD #4, 3
MOZ 2, @2
JMP -2
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #4
DAT #0, #0
DAT #0, #0
DAT #0, #0
DAT #0, #8
DAT #0, #0
DAT #0, #0
```

Ebben az esetben a program önmagára másolja a DAT utasítást, majd az egész folyamat kezdődik elölről. Ilyen módon kerüli el a program önmaga bombázását. Természetesen ez a módszer csak akkor működik, ha az aréna mérete osztható néggyel.

A program mellé két további mintapéldát mellékeltünk.(ld. peldak mappa)

Címzési módok

A két példa után nézzük a címzési módokat:

- # - számot jelöl
- $ - cím jelölés, de a dollár jel elhagyható
- * - indirekt címzés, vagyis a * jel utáni számmal megadott utasítás A mezőjének használata
- @ - indirekt címzés, vagyis az @ jel utáni számmal megadott utasítás B mezőjének használata
- { - indirekt címzés, először az A mező csökkentése, majd az A mező használata
- < - indirekt címzés, először a B mező csökkentése, majd a B mező használata
- } - indirekt címzés, először az A mező használata, majd az A mező növelése
- > - indirekt címzés, először a B mező használata, majd a B mező növelése

Például:

```
DAT #5, #10
MOZ -1, }-1
```

eredménye ez lesz:

```
DAT #6, #10
MOZ -1, }-1
…
…
…
DAT #5, #10
```

Mint látható, a MOZ utasítás az A mezője alapján (-1) a megelőző DAT utasítást fogja átmásolni. A }-1 jelentése itt az, hogy a -1-es (vagyis megint az előző) utasítás A mezőjében található távolságra történik a másolás. Így indirekt módon egy másik utasítás adatát használjuk arra, hogy megkapjuk hova történik a másolás. Ezután az előző utasítás A mezőjét megnöveli.

Érdemes egy fontos megjegyzést tenni, hogy az előzetes növelés és csökkentés mindenképpen végrehajtódik. Például:

```
JMP -1, >100
```

az előző utasításra ugrik, és a 100 címmel odébb levő utasítás B mezőjét megnöveli eggyel. Ez a növelés akkor is bekövetkezik, ha magát az értéket az utasítás semmire nem használja Hasonlóan:

```
DAT >50, >60
```

megnöveli a címeken található A mezőket, majd a program leáll, mivel végrehajtotta a DAT utasítást.

Címek

Ismét kiemelnénk, hogy valójában minden cím a *0* és *aréna méret*-1 tartományba konvertálódik. Ez azt jelenti, hogy ha megadunk egy címet, akkor az értéket elosztjuk az aréna méretével, és csak a maradékot tartjuk meg. Negatív számok esetén a címhez hozzáadjuk az aréna méretét, majd ezután végezzük el az osztást, és határozzuk meg a maradékot. Így 8000-es arénaméret esetén a -1 –es cím aktuálisan 7999 lesz.

Ennek nincs jelentősége az egyszerűbb utasítások esetén, például ADD, SUB. Ugyanakkor fontos tudni, hogy a DIV, MOD, SLT utasítások is mindig pozitív számokat kezelnek a fentiek alapján.

Például ha az aréna mérete 8000, akkor:

-2 / 2

nem egyenlő -1 -gyel, hanem:

(8000-2) / 2 = 7998 / 2 = 3999

Hasonlóan, amikor az SLT utasítást használjuk a -2 -es értékkel, akkor valójában a 7998-as értéket hasonlítjuk össze nullával. Vagyis az SLT utasítás esetén nulla a legkisebb szám!

Programvégrehajtás

Figyelmet érdemel az FRK utasítás, mivel egy folyamatot hoz létre. A különbség a JMP és az FRK utasítás között, hogy az FRK utasítás egy új folyamatot kezd az utasítás A mezőjében megadott helyen, és ezen kívül az eredeti folyamat a következő utasításnál is folytatódik. Az így létrehozott folyamatok egyenlő mértékben osztoznak az időn. Ennek hatására azt is tisztázni kell, hogy valójában nem egy mutatót használ a szimulátor, ami a következő végrehajtandó utasításra mutat, hanem egy folyamatsort. A folyamatsor az egymás után végrehajtandó folyamatokat tartalmazza. Az FRK utasítással létrehozott folyamat ennek a sornak a végére kerül. Ha egy folyamat egy DAT utasítást hajt végre, akkor törlődik a sorból.

Mindezek után nézzünk egy példát a szabályok betartásával. Legyen program A-nak 3 folyamata, és program B-nek 1 folyamata. Akkor a végrehajtás a következő lesz:

program A, folyamat 1
program B, folyamat 1
program A, folyamat 2
program B, folyamat 1
program A, folyamat 3
program B, folyamat 1
program A, folyamat 1
program B, folyamat 1

Végül nézzünk egy példát, ami az első mintaprogramot indítja el „végtelen” példányban:

```
FRK 0
MOZ 0, 1
```

Az FRK utasítás önmagára mutat, vagyis egy új folyamat jön létre, majd újra lefuttatja az FRK utasítást, míg az első folyamat maga továbblép a második utasításra, ami elindítja a MOZ sorozatot. A második folyamat az FRK utasítás futtatásával létrehoz egy harmadik folyamatot, ami majd az FRK utasítással kezdődik, és a második folyamat is elindít egy MOZ sorozatot, és így tovább. Ez a program rengeteg folyamatot indít el, amíg az első folyamat körbe nem ér, és felülírja az FRK parancsot.

Ugyanakkor fontos tudni, hogy nem csak az aréna mérete korlátos, hanem az elindítható folyamatok száma is. (Ezeket a korlátokat „A futtatás paraméterei” bekezdésben tárgyaljuk.) Ha elértük a folyamatok számának korlátját, akkor az FRK utasítás csak a következő utasításnál folytatódik, és nem hoz létre új folyamatot a megadott címen.

A futtatás eredménye

A programok futtatása kétféleképpen érhet véget:

- csak egy program marad életben; vagy
- letelt a futtatási ciklusok maximális száma (Lásd A futtatás paraméterei bekezdést).

Minden túlélő program minden forduló után pontot kap a következő formula alapján:

(W*W-1)/S

ahol W a résztvevő programok száma és S a túlélő programok száma. Az osztás (/) egy egész osztásnak felel meg.

A szimulátor az eredményeket a következőképpen jelenti egy programról, amikor n másik programmal csatározott:

win1 win2 win3 … winn vesztés

ahol win1 azt jelenti, hogy hányszor fordult elő, hogy csak egy program maradt életben, hányszor fordult elő, hogy csak két program maradt életben, és így tovább. A lista végén szereplő szám azt adja meg, hogy hányszor halt meg a program. Például 100 futtatás után a következő eredményt kaphatjuk 4 program esetén:

```
Program1 by Csapat    453
     5  13  22  59  1
```

Ebből a futtatásból 5-ször fordult elő, hogy a Program1 maradt csak életben, 13-szor fordult elő, hogy két program maradt életben (köztük a Program1), 22-ször fordult elő, hogy három program maradt életben (köztük a Program1), és így tovább. 1-szer a program meghalt. A Program1 által kapott pont számítása pedig a következőképpen történik:

5*[(4*4-1)/1] + 13*[(4*4-1)/2] + 22*[(4*4-1)/3] + 59*[(4*4-1)/4] =

5*[15] + 13*[7] + 22*[5] + 59*[3] = 453

A futtatás paraméterei

Az aréna mérete: 8000 utasítás

Maximum indítható programok, illetve program folyamatok száma: 8000

Maximum futtatási ciklus egy fordulóban: 80 000 lépés (A programok összesen (nem egyenként) ennyi utasítást hajthat végre.

Egy program maximális mérete: 100 utasítás

A beküldendő program formátuma

A verseny második szakaszában a csapatok egymás ellen versenyeznek. A küzdelemhez az elkészített programot fel kell tölteni egy magadott szerverre. A feltöltendő program kötelező formátuma:

```
;name Program_neve
A programkód ...
...
END
```

Az END után legyen egy üres sor! Ha a szerverre feltöltött programmal bármilyen probléma van (szintaktikai vagy formátum hiba), akkor a szerver az első példában bemutatott 1 soros programot

```
MOZ 0,1
```

tölti be a csapathoz!

Debugger használata

A kihívás megoldásához kétféle szimulátort biztosítunk:

- progwar-server.exe : grafikus interface nélküli szerver változat.
- progwar.exe : egy grafikus interface-szel rendelkező változat debuggoláshoz

Parancssori változat futtatása

A progwar-server.exe program Parancssorból (Command Prompt) futtatható. A futtatás módja:

```
progwar-server.exe prog1.asm prog2.asm
```

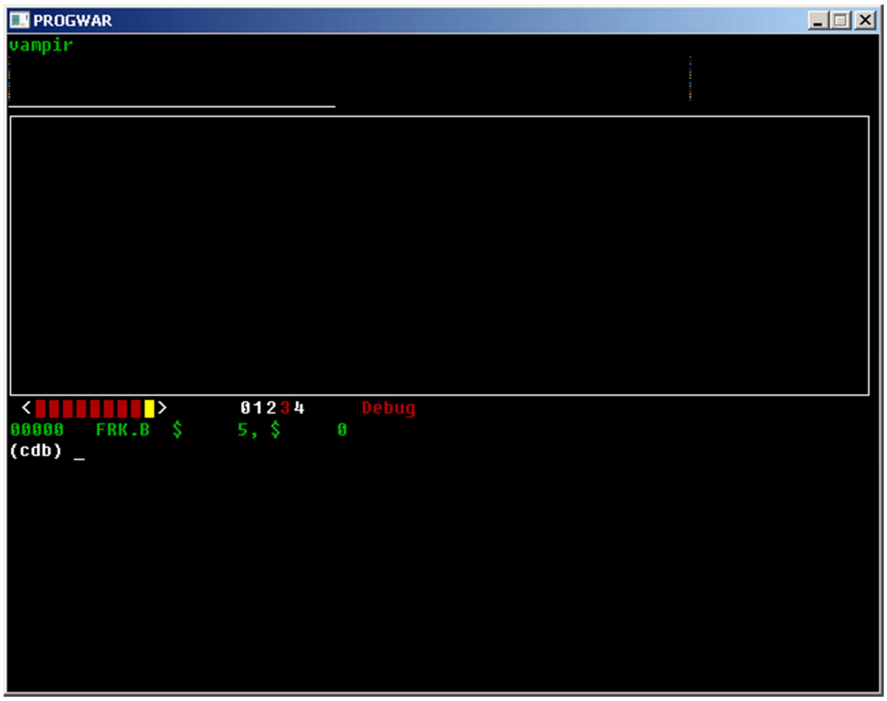
Ekkor csak egy ütközet lesz a prog1 és prog2 programok között.

A végső verseny során a programokat 100-szor küldjük az arénába egymás ellen - egyszerre az összeset. Ennek teszteléséhez a szimulátort így kell futtatni:

```
progwar-server.exe –r 100 prog1.asm prog2.asm
```

Grafikus változat futtatása

A grafikus változat esetén két ablak nyílik meg, egy parancssor és egy grafikus ablak:



A grafikus ablak szerkezete fentről lefelé:

- felső részén a programok neve jelenik meg (de csak két program indítása esetén);
- a programok processzusainak számát jelző színes pixelek vagy vonalak;
    - Egy pixel egy processzust jelent.
- a fehér vonal jelzi, hogy hány iteráció van még hátra;
- a fehér keretezett téglalap az aréna területét mutatja;
- a zöld sor a következő végrehajtandó utasítás címe és maga az utasítás;
- a parancssor: (cdb).

A programból bármikor ki lehet lépni a jobb felső sarokban található X-re kattintva. Másik kilépési lehetőség, ha a 'quit' parancsot gépeljük be a parancssorba. Ebben az esetben a Command Prompt ablakban is egyszer meg kell nyomnunk egy billentyűt. A (Command Prompt ablakra rá kell kattintani.) Ez a megoldás azért kellett, hogy a program ne zárja be a Command prompt-ot azonnal Windows alatt.

A grafikus programot indíthatjuk debugger-rel vagy anélkül. A debugger nélküli indítás ugyanaz, mint a szerver változatnál:

```
progwar.exe prog1.asm prog2.asm
```

Ugyanakkor ebben az esetben is megjelenik egy grafikus ablak, és a programok végigfutnak. A lezáráshoz a **grafikus és a parancssori ablakban** is egy billentyűt kell lenyomni.

Indítás debugger-rel:

```
progwar.exe –e prog1.asm prog2.asm
```

Ebben az esetben a programok betöltődnek, majd a futás megáll, és a debugger parancsokat vár tőlünk. Parancsokat az egérrel vagy a billentyűzettel adhatjuk ki. (A programhoz szükséges a progwar.mac file is.)

Ha az egérrel kattintunk az aréna területére, akkor a program kilistázza a memóriaterületről a kattintás helye körüli részt. Ezután a fel és le nyilakkal mozoghatunk a memóriában. Az arénában színesen jelennek meg azok a mezők, amelyeket módosított valamelyik program.

A szimulátor parancsairól rövid segítség kérhető a 'h' vagy 'help' paranccsal. Hasznos parancsok:

- st vagy step : egy utasítást végrehajt és átvált a következő processzusra
- l vagy list : az aréna listázása
- r vagy registers : szimulátor állapotának kinyomtatása a képernyőre
- p vagy progress : a programok eddigi állapotát nyomtatja ki a képernyőre

Feladat

A verseny végén elsőnek lenni a szerveren nyilvántartott listán!

Néhány lehetséges stratégia

- A többi program lebombázása
- A többi program megkeresése és lebombázása
- A többi program lelassítása és lebombázása
- Saját programunkkal rejtőzködés más programok elől, így minden menetben életben maradunk

Mindezt minél hatékonyabban és gyorsabban.