Kötelező félévi házi feladat (Modellellenőrzés)

A feladat címe: **Futtatási szálak kezelése**

Konzulense: **Bartha Tamás**

Leírás

A feladat egy egyszerű operációs rendszer feladatainak és futtatási szálainak kezelését modellezni. A feladatokat az operációs rendszer periodikus módon, kötött idejű intervallumokban futtatja. Minden intervallum elején eldől, mely feladatok „kívánnak" (véletlen választással) az adott periódusban futni. A „futásra jelentkező" feladatok közül azok processzorigénye és prioritása alapján eldől, melyek kapnak az adott periódusban futási jogot. A futtatható feladatok számát a futtatási szálak véges száma is korlátozza (minden feladat külön szálon fut). Az intervallum leteltével a feladatokat leállítják, egy futási periódus lezárul, az operációs rendszer „alaphelyzetbe" kerül. A folyamat innentől elölről kezdődik: a feladatok újra véletlenszerűen döntenek, hogy akarnak-e futni, újra kiválasztják a jelentkezők közül a ténylegesen futási jogot kapókat, új intervallum kezdődik, és így tovább.

A rendszert három fő komponens alkotja:

1. A futtatandó *feladatok*: ezek futtatása operációs rendszer célja. Minden feladat három fő paraméterrel rendelkezik, amelyek minden feladatra külön állíthatók be:
   a. *Affinitás*: ez egy 0 és 10 közötti egész szám. Azt jellemzi, mekkora valószínűséggel „kíván futni" az adott feladat.
   b. *Igény*: ez egy 1 és 10 közötti egész szám. Jelentése, hogy az adott feladat 10*Igény százaléknyi processzorterhelésre tart igényt. Ez a feladatok kiválasztásának egyik fő paramétere, mert csak ≤ 100% lehet a kiválasztott feladatok igényelte összes processzor-terhelés.
   c. *Prioritás*: a feladatok adott számú (konfigurálható paraméter, értéke legyen pl. 3) prioritási szinttel rendelkeznek. Ez a feladatok kiválasztásának másik fő paramétere, ugyanis az igények szabta korláton belül a feladatokat prioritási sorrend szerint kell kiválasztani.
2. Az *ütemező*: ez választja ki a „futásra jelentkező" feladatok közül azokat, amelyek futási jogot kapnak. A kiválasztott feladatokat szálakhoz rendeli. A rendszerben adott számú futtatási szál van (a szálak száma < feladatok száma, pl. 4 szál és 5 feladat). Minden feladat egy adott szálhoz van rendelve, és maximum annyi feladat futhat, ahány szál van.
3. A *CPU*: erőforrás, ami a feladatok futtatásához szükséges. Alapvetően két állapota van: aktív és inaktív. Aktív állapotában futhatnak a feladatok. Az aktív állapot közben bekövetkezhet egy megszakítás, ami preemptív módon beavatkozik a feladatok futásába (erről bővebben később).

A feladatok az alapállapotból kilépve egy 0 és 10 közötti *p* véletlen számot sorsolnak. Ezt hasonlítják össze az Affinitás paraméterükkel, és ha p ≥ Affinitás, akkor futásra jelentkeznek, ha kisebb, akkor inaktív állapotba kerülnek. A futásra jelentkezés illetve az arról való lemondás regisztrálása az ütemezőnél történik. Ehhez a feladatok és az ütemező szinkronizációt és két külön csatornát (egyiket a jelentkezők, másikat a lemondók) használnak.

A regisztrálás után az ütemező feldolgozza a jelentkezéseket. A korlátok betartása (az igényelt összes terhelés ≤ 100% és max. annyi feladat, ahány szál) mellett prioritási szintben a legmagasabbtól lefele haladva, az igényelt terhelés szerint pedig a nagyobbtól a kisebb felé haladva sorrendezi a feladatokat. A futásra kiválasztott feladatokat szálakhoz rendeli és ezt egy globális adatstruktúrában adminisztrálja.

Azokat a feladatokat, akik jelentkeztek ugyan, de nem fértek be a fenti ütemezési stratégia mellett a korlátokba, értesíti a visszautasításról. A visszautasított feladatok a futásról lemondókkal együtt inaktív állapotba kerülnek. A jelentkezett és futásra kiválasztott feladatok szintén értesítést kapnak az elfogadásról. Ezután egy startjelre kezdenek várakozni.

A startjel (amit az ütemező ad ki egy broadcast csatornán keresztül) hatására több dolog is történik egyszerre:

- a futásra kiválasztott feladatok futó állapotba kerülnek,
- az ütemező várakozó állapotba kerül, amiből majd csak a végjel hatására fog továbblépni,
- a CPU aktív állapotba kerül, a rajta futó szálak és feladatok listáját az már említett globális adatstruktúra tartalmazza.

A CPU aktív állapotában érkezhet egy nem maszkolható megszakítás, ami preemptív módon félbeszakítja egyes feladatok futását. Hogy mely feladatok kerülnek felfüggesztésre, azt a megszakítás CPU igénye (ami szintén beállítható paraméter, 1 és 10 közötti egész szám) dönti el. Annyi feladatot kell felfüggeszteni (a legkisebb prioritásúaktól felfelé haladva), amennyi már elég CPU kapacitást szabadít fel ahhoz, hogy a megmaradó feladatok és a megszakítás-kezelő rutin CPU igényeinek összege ≤ 100% legyen. A felfüggesztendő feladatok kiválasztását a CPU végzi, majd értesíti is a felfüggesztésre kiválasztott feladatokat, amelyek futó állapotból felfüggesztett állapotba kerülnek. A megszakítás-kezelő rutin lefutása után a CPU értesíti a korábban felfüggesztett feladatokat, amelyek ennek hatására újra futó állapotba kerülnek. Ezután a CPU is újra futó állapotba kerül.

Egy adott idő után véget ér az aktuális futási időintervallum (ezt a CPU aktív állapotára előírt invariáns órafeltétellel modellezzük). Az aktív állapot elhagyásakor a CPU egy broadcast csatornán egy végjelet küld. Ennek hatására ismét több dolog történik egyszerre:

- a CPU inaktív állapotba kerül,
- az ütemező alapállapotba kerül, de ezelőtt a futó szálak és feladatok listáját tartalmazó globális adatstruktúrát alaphelyzetbe (üres) állítja,
- a feladatok is (bizonyos kivétellel) alapállapotba kerülnek.

Tehát a végjel hatására alapesetben a rendszer a kezdőállapotába kerül, és készen áll egy újabb futási periódus végrehajtására.

Ez alól azonban van kivétel. Ugyanis ha a felfüggesztett feladatok túl sok időt töltenek felfüggesztett állapotban (ezt a felfüggesztett állapotra előírt invariáns órafeltétellel modellezzük), akkor annyi „késést” szednek össze, hogy nem tudják befejezni a teendőiket az adott futási periódusban. Ezek a késő feladatok „megkísérlik” a következő futási periódusban folytatni a végrehajtást. Ezért a végjel hatására nem a kezdőállapotba kerülnek, hanem abba az állapotba mennek át, amiben a futásra fognak jelentkezni a következő futási periódusban (tehát átugorják a véletlen döntési fázist).

A feladatra a következő időzítési korlátok vonatkoznak:

- a futásra kiválasztott feladatok, az ütemező és a CPU órái a startjelre egyszerre indulnak,
- a CPU az aktív állapotában max. a 4. időegységig tartózkodhat,
- a megszakításkérés az 1. és 2. időegység között következhet be,
- a megszakítás-kezelő rutin legfeljebb a 3. időegységig futhat, azután visszatér,
- ha a felfüggesztett feladatok a 2. időegység után a felfüggesztett állapotban vannak, akkor késő feladattá válnak.

Az ellenőrzendő követelmények

Temporális logikai kifejezések és modellellenőrzés segítségével igazolja az alábbi követelmények teljesülését (illetve a követelmények nem teljesülése esetén ellenpélda segítségével magyarázza meg a követelmény megsértésének okát és indokát)!

1. A modellben nincs deadlock.

A további feltételeket bizonyítsa az alábbi paraméterezés esetén:

Feladat azonosító	Affinitás	CPU igény	**Prioritás** (A legkisebb számérték jelenti a legmagasabb prioritást.)
0	0	2	0
1	3	3	1
2	3	4	1
3	3	1	1
4	3	5	2

2. Lehetséges, hogy a futásra jelentkezett feladatok közül legalább egyet vissza kell utasítani.
3. Lehetséges, hogy összes szál foglalt, azaz maximális számú feladat fut.
4. Ha van futó feladat, akkor a globális adatstruktúrában a foglalt szálak száma > 0.
5. Lehetséges, hogy a CPU megszakítás bekövetkezte esetén > 2 szálat felfüggeszt.
6. Lehetséges olyan lefutás, hogy egyetlen feladatot sem függesztenek fel egyetlen futási periódusban sem, de nem lehetséges, hogy minden lefutás ilyen: azaz van legalább egy olyan lefutás, amiben legalább egyszer legalább egy feladatot felfüggesztenek.
7. Nem lehetséges, hogy egy feladat a 3. időegység után felfüggesztett állapotban legyen.
8. Ha egy feladatot felfüggesztenek, akkor lehetséges, hogy valamikor be tudja fejezni a feladatát.

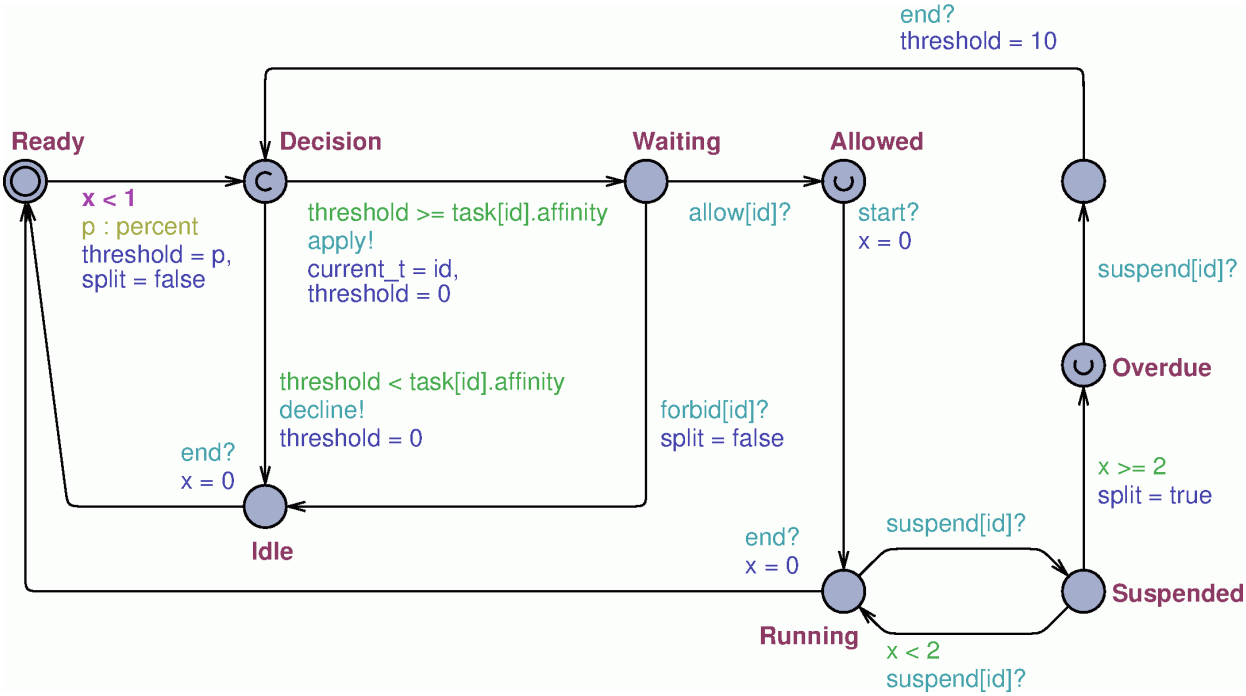
Megoldás

Globális deklarációk:

```
system Task, Scheduler, CPU;

typedef int[0,10] percent;

const int Levels = 3;
typedef int[0,Levels-1] p_level;

const int Tasks = 5;
typedef int[0,Tasks-1] t_id;

t_id current_t; // shared variable

typedef struct {
  percent affinity;
  percent demand;
  p_level pri;
} task_t;

// affinity, demand, priority
const task_t task[Tasks] = {
{0, 2, 0},
{3, 3, 1},
{3, 4, 1},
{3, 1, 1},
{3, 5, 2}
};
```

```
typedef struct {
  int[0,Tasks] length;
  t_id task[Tasks];
} buffer_t;

const int Threads = 4;
typedef struct {
  int[0,Threads] num;
  t_id task[Threads];
} thread_t;

thread_t thread;

chan apply, decline;
urgent chan allow[Tasks], forbid[Tasks];
chan suspend[Tasks];
broadcast chan start, end;
```

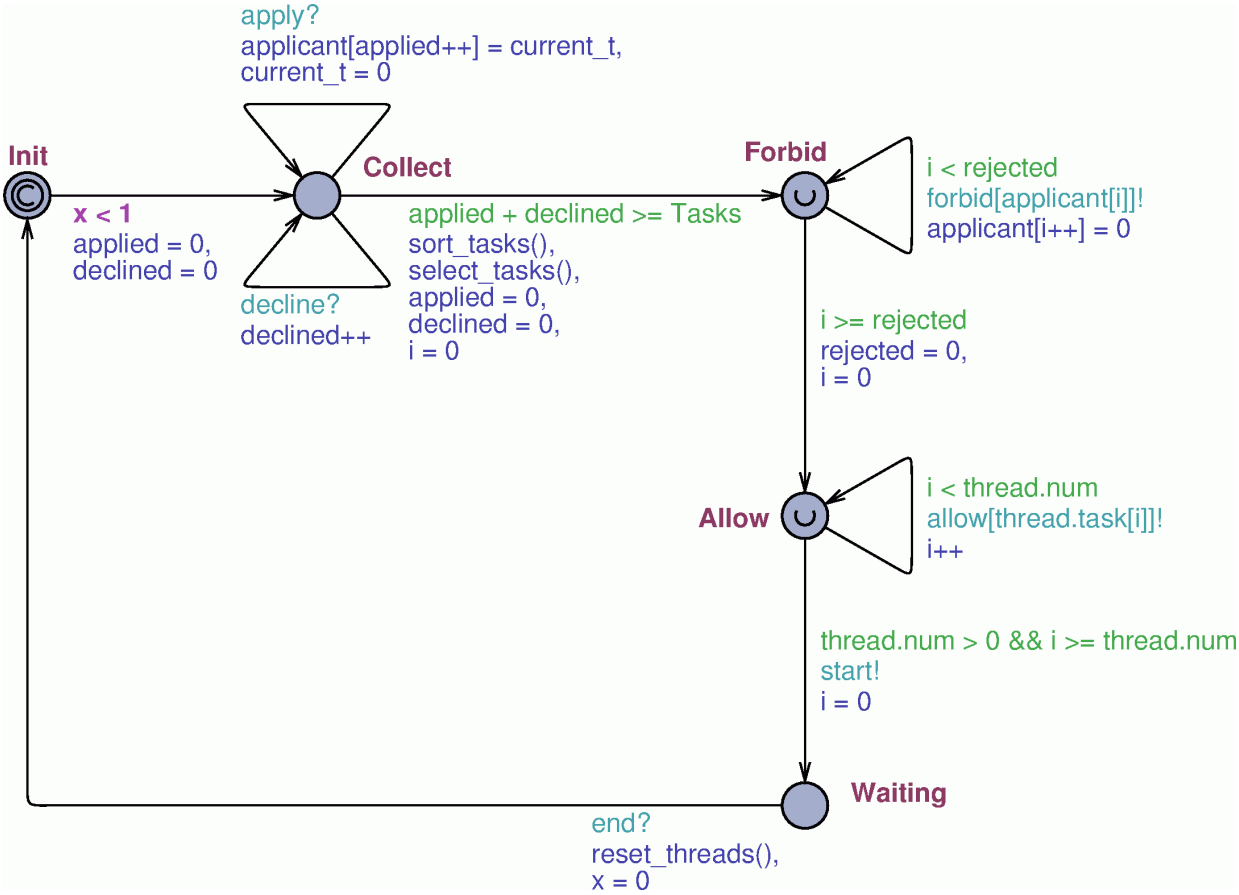
A feladatokat leíró automata:



A feladatok lokális deklarációi:

```
clock x;
meta bool split = false;
percent threshold;
```

Az ütemezőt leíró automata:

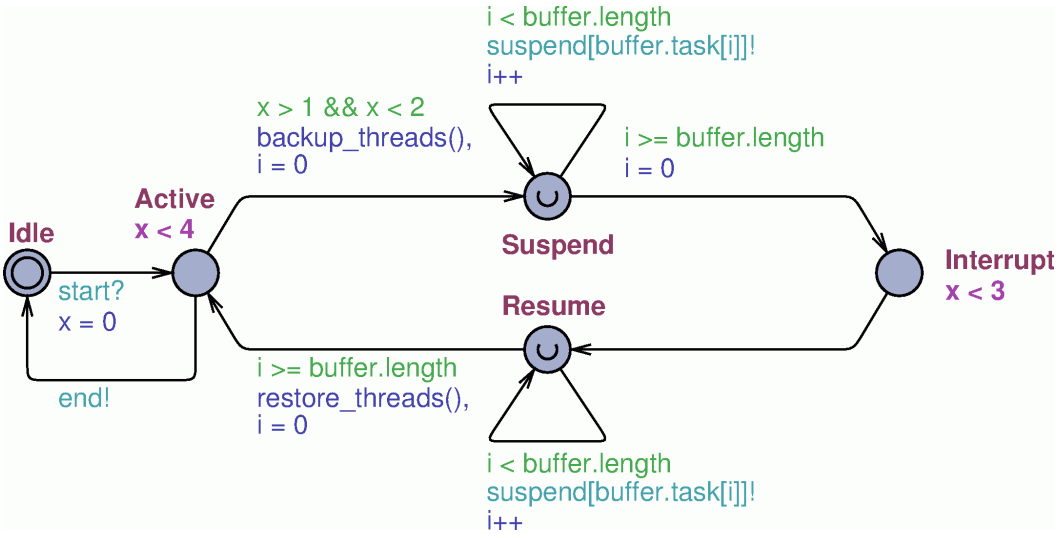


Az ütemező lokális deklarációi és eljárásai:

```
clock x;
int i;
int[0,Tasks] applied, declined, rejected;
t_id applicant[Tasks];
buffer_t buffer;

void insert_at(int[0,Tasks] pos, t_id tid) {
  int i;
  for (i = buffer.length; i > pos; i--) {
    buffer.task[i] = buffer.task[i-1];
  }
  buffer.task[pos] = tid;
  buffer.length++;
}

void sort_tasks() {
  int i, j, pos;
  for (i = 0; i < applied; i++) {
    for (j = 0, pos = -1; j < buffer.length && pos < 0; j++) {
      if (task[applicant[i]].pri <  task[buffer.task[j]].pri ||
         (task[applicant[i]].pri == task[buffer.task[j]].pri &&
          task[applicant[i]].demand > task[buffer.task[j]].demand)) {
        pos = j;
      }
    }
    insert_at(pos < 0 ? buffer.length : pos, applicant[i]);
    applicant[i] = 0;
  }
}
```

```
void select_tasks() {
  int i, pri;
  percent p = 0;
  rejected = 0;
  thread.num = 0;
  for (i = 0; i < buffer.length; i++) {
    if (p + task[buffer.task[i]].demand <= 10 && thread.num < Threads) {
      thread.task[thread.num++] = buffer.task[i];
      p += task[buffer.task[i]].demand;
    }
    else {
      applicant[rejected++] = buffer.task[i];
    }
    buffer.task[i] = 0;
  }
  buffer.length = 0;
}

void reset_threads() {
  while (thread.num > 0) thread.task[thread.num-- - 1] = 0;
}
```

A CPU-t leíró automata:



A CPU lokális deklarációi és eljárásai:

```
clock x;
const percent i_demand = 5;
int i;
buffer_t buffer;

void backup_threads() {
  int i, p;
  t_id tid;
  for (i = 0, p = 0; i < thread.num; i++) p += task[thread.task[i]].demand;
  buffer.length = 0;
  for (i = 0; i < thread.num; i++) {
    if (p + i_demand > 10) {
      tid = thread.task[thread.num - i - 1];
      buffer.task[buffer.length++] = tid;
      thread.task[thread.num - i - 1] = 0;
      p -= task[tid].demand;
    }
  }
  thread.num -= buffer.length;
```

```
}

void restore_threads() {
  int i;
  t_id tid;
  for (i = 0; i < buffer.length; i++) {
    tid = buffer.task[buffer.length - i - 1];
    thread.task[thread.num++] = tid;
    buffer.task[buffer.length - i - 1] = 0;
  }
  buffer.length = 0;
}
```

A verifikációs CTL kifejezések: