Valós idejű funkcionalitás Windows-ban

Szabó Dávid, Dr Illés Zoltán, Heizlerné B. Viktória

{ sasasoft, illes, hbv }@inf.elte.hu
ELTE IK

Absztrakt: Az elmúlt években tapasztalhatjuk, hogy az időzítések, határidők szerepe az élet minden területén egyre jelentősebbé vált, így van ez a számítógépek világában is, elterjedt a valós idejű alkalmazások iránti igény. Szoftverek, melyeknek nem elég gyorsan reagálniuk, a választ határidőre kell előállítaniuk. Az átlag felhasználók között egyik legelterjedtebb operációs rendszer, a Windows, ennek ellenére teljes funkcionalitásában nem támogatja a valós idejű alkalmazásokat, így azok ütemezését sem. Megvizsgáljuk, hogy milyen szolgáltatásai és programozható felületei érhetők el a rendszernek, melyekkel mégis alkalmassá tehető valós idejű alkalmazások fejlesztéséhez.

Kulcsszavak: valós idő, folyamat, szál, ütemező, Windows, .NET, C#

1. Bevezetés

A számítógépek sebességének növekedésével párhuzamosan a mindennapjaink is egyre gyorsabbak lesznek. Korábban percek, órák vagy akár csak napok alatt elvégezhető tevékenységekre nem várunk néhány másodpercnél többet. Számítógépek, telefonok és egyéb okos eszközök vesznek minket körül a nap 24 órájában és egy olyan világba nyújtanak belépést, melyben minden mindennel össze van kapcsolva. Ezt a változást nem csak a használt hardverek, de a használt szoftverek is követik.

Egy külső szemlélő ebből mindössze annyit láthat, hogy a szoftvereknek még gyorsabbnak kell lenniük, de ez több ennél. A válaszokat és reakciókat határidőre kell a rendszernek előállítania. Nem elég gyorsan vagy még gyorsabban reagálni, valós időben mérhető időbeli követelményeket támasztunk a szoftverrel szemben.

Több időbeli követelmény fajtát is megfigyelhetünk:

1. Bankkártya használata során elvárjuk, hogy a fizetés folyamata gyorsabb legyen, mintha készpénzzel fizetnénk, ezért a terminál és a bank közötti hálózati kommunikációra, tranzakció végrehajtására csupán néhány másodperce van a rendszernek.

2. Online videojátékok esetében meghatározó tényező a ping, avagy a szerver válaszideje. Minél rosszabb a válaszidő annál rosszabb lesz a szinkronizáció a játékosok között, ugyanazt a jelenséget nem ugyanakkor fogják látni a képernyőkön. A manapság feltörő e-sport világában akár néhány milliszekundumos késés is hátrányba juttathat versenyzőket.

3. Az előző problémák előfordulása jórészt kellemetlenséggel és bosszankodással jár, de egyes szoftverkörnyezetekben a határidők elmulasztása katasztrofális következményekkel járhat. Egy gyógyszergyár gyártósorát vezérlő szoftverének milli-, vagy akár mikroszekundumos pontossággal kell a műszereit vezérelnie. Egy apró hiba és a hibásan előállított termék, gyógyszer akár emberi életeket is veszélybe sodorhat!

2. Valós idejű szoftverfejlesztés

Valós idejű szoftverfejlesztésről akkor beszélhetünk, amikor a fejlesztés alatt álló szoftver követelményrendszerében az időtől függő feltételeket és követelményeket adunk meg. Követelmények, melyek megszabják, hogy egy eseményre előállított válasz elkészítésére mennyi ideje van a rendszernek. Megadott határidővel vagy időkorláttal kell a szoftvernek feldolgoznia az eseményt és végrehajtania az esemény által kiváltott feladatokat a válasz előállításához. Egy követelmény akkor jó követelmény, ha minden eshetőségre kitér. Mi a teendő, ha ezt a határidőt mégis elmulasztja a rendszer? Két megközelítést alkalmazunk [1].

Laza valós idejű rendszer esetében (soft real-time) a rendszer törekszik a határidők betartására. A késés elkerülendő, de ha mégis bekövetkezik nincs túl nagy probléma, csupán az előállított válasz hasznossága csökken. Ezzel szemben a szigorú valós idejű rendszerek (hard real-time) határidő elmulasztása rendszerhibát vagy egyéb súlyos következményeket okozhat, a késés nem tolerálható.

Adott egy követelmény, melyben szoftverünknek egy eseményre adott határidőn belül kell a választ előállítania, végrehajtania egy kódrészletet. A beérkező eseményhez egy adat tartozik szorosan, a határidő és egy adat lazán, a válasz kódrészlet végrehajtási ideje. Sajnos attól, hogy a szoftverünk elég gyorsan (pl. valamennyi milliszekundum alatt) le tudja futtatni a szükséges kódot, még nem jelenthetjük ki, hogy ez a szoftver minden esetben tartani fogja az előírt határidőt. Ehhez garantálni kellene, hogy a végrehajtáshoz szükséges processzor időt meg is kapja a programunk még a határidő lejárta előtt.

2.1 Ütemezők általánosságban

Egy számítógépen egyszerre sokkal több alkalmazást és folyamatot futtatnak a felhasználók, mint amennyi feldolgozó egység elérhető az adott rendszeren. Az operációs rendszer feladata, hogy ezen folyamatokat úgy ütemezze a processzorokra, hogy a felhasználó számára ne tűnjön föl, hogy igazából ezek az alkalmazások nem egyszerre futnak, hanem felváltva. Az Ütemező az Ütemezési Algoritmus alapján dönti el, hogy melyik folyamat, melyik szála kaphat processzort és mikor kell lemondania arról [2]. Az ütemezésnek pártatlannak kell lennie, minden szál ugyanazon szabályrendszer szerint kell, hogy ütemezésre kerüljön. Az évek során használt rendszerek több Ütemezési Algoritmust is bemutattak, mind más és más környezetben alkalmazható kényelmesen.

Korábbi többfeladatos rendszerek kooperatív ütemezést alkalmaztak, melynek során az ütemező nem dönthet úgy, hogy egy végrehajtás alatt álló folyamattól elveszi a processzort. A folyamatoknak maguktól kell lemondaniuk a processzorról (pl. egy várakozó utasítás, vagy yield parancs használatával). Ennek a Run-to-Completition elvű ütemezőnek előnye, hogy a futás alatt álló folyamat a lehető leggyorsabban végre tudja hajtani feladatát, mert az ütemező nem szakítja meg munkáját váratlan kontextus váltásokkal. Cserébe a rendszer válaszideje és interaktivitása rendkívül gyenge, a futó folyamatok kiéheztetik a többi várakozó folyamatot.

A mai átlag felhasználók által használt rendszerek időosztásos, prioritásos preemptív ütemezőket használnak. Az ütemező nem csak egy processzor szálhoz rendelését teheti meg, a processzort el is veheti a szálaktól (preemption). Egy processzor felszabadulásakor egy várólistában várakozó szál kapja meg a processzort, előre megadott időegységnyi processzor időt kap az ütemezőtől. Az időkorlát lejárásakor az ütemező elveszi a processzort a száltól, a szál visszakerül a várólistába a processzort pedig a következő várakozó szál kapja meg [3]. A felhasználók (és fejlesztők is) prioritásokat rendelhetnek a szálakhoz, ezzel tippet adva az ütemezőnek, hogy mely folyamatainkat szeretnénk nagyobb vagy kisebb sűrűséggel végrehajtás alatt tudni.

Egy interaktív rendszerben ez az algoritmus jól működik, viszont valós idejű környezetben nem garantál semmit! Egy esemény beérkezésekor nem garantált, hogy a folyamatunk ütemezésre kerül, sem az, hogy elegendő processzor időt kap a válasz előállításához szükséges feladatok

végrehajtásához. Egy ténylegesen valós idejű rendszer figyelembe veszi az idő szerepét is a folyamatok végrehajtása során! Az operációs rendszer garantálja, hogy egy esemény bekövetkezésekor a rá várakozó folyamat processzorhoz jut és a szükséges utasítások lefutnak. Ezen funkcionalitást támogató valós idejű rendszerek lehetnek például a beágyazott rendszerek, illetve a Linux rendszernek érhetők el valós idejű bővítményei (pl. Suse Linux Enterprise Real-Time Extension). A Linux kernel 3.14 verziótól kezdve támogatja a SCHED_DEADLINE ütemezést, mely egy EDF (Earliest Deadline First) ütemezés [4]. A folyamatokhoz hozzárendelhető határidőt és szükséges végrehajtási időt figyelembe véve képes a rendszer folyamatokat ütemezni.

A helyzet tovább bonyolódik többmagos/többprocesszoros rendszerek esetében. Futó folyamatok váltása során az ütemezőnek kontextus váltást (context switch) kell végrehajtania. Az eddig futó szállnak az állapotát (utasításszámláló, stack, regiszterek, cache állapotok stb.) el kell tárolni, a következő szál állapotát pedig be kell tölteni. Ez egy drága, időigényes feladat, ha pedig a szál állapotát a különböző processzormagok között is meg kell osztani, akkor még több időt fog igénybe venni. Éppen ezért szeretnénk irányítani, hogy az ütemező, mely processzorokra ütemezheti egyes folyamatainkat. Ezt az opciót processzor affinitásnak nevezik. Valós idejű rendszerek a processzor shield opciót is támogatják, mellyel kijelölhetjük processzoroknak egy halmazát, melyekre csak valós idejű folyamatokat ütemezhetnek, ezzel csökkentve egy beérkező valós idejű folyamat várakozási idejét.

2.2 Windows ütemező

Windows rendszerben megkülönböztetjük a folyamat és szál fogalmát [5]. A folyamat egy kontextus az alkalmazásunknak, menedzseli az erőforrásait, memóriaterületét és nyilvántartja a szálait. A szálak a különböző végrehajtási útjai egy folyamatnak, az ütemező a szálakat ütemezi a processzorra. Két lépésben rendelhetünk prioritást a szálakhoz: először egy prioritási osztályt kell a folyamathoz rendelni (Realtime, High, Above Normal, Normal, Below Normal, Idle), mely minden a folyamatban lévő szálnak a prioritását a megfelelő szintre módosítja, majd az osztályon belül a szálaknak az egymáshoz képesti relatív prioritását is megadhatjuk.

A Windows kliens, illetve szerver változata sem támogatja a valós idejű ütemezéshez szükséges szolgáltatásokat. Habár a prioritási szintek (0-31) között megjelennek a valós idejű prioritások (16-31) ezek nem igazi valós idejű algoritmusokat rejtenek maguk mögött. A rendszer nem garantálja a határidők betartását, csak annyit, hogy a legtöbb szál előtt ütemezésre kerülhet a valós idejű prioritású szálunk. Használata nem veszélymentes: több rendszerszintű szál is a valós idejű szinteken dolgozik, ha egy számításigényes szállal blokkoljuk ezeket a folyamatokat a rendszer instabillá válhat. Egy szál csak akkor lehet valós idejű, ha a szálat tartalmazó folyamat valós idejű prioritási osztályba tartozik, melynek beállításához rendszergazdai jogosultságra van szükség. Valós idejű prioritási osztályban csak valós idejű szálak lehetnek.

Quantum néven hívják az időegységet, mely a folyamatok által felhasználható időszeletet határozza meg. A Quantum hosszát a kernel számítja ki a rendszer indulásakor. Egy szál processzorhoz jutáskor kap valamennyi Quantum-ot, végrehajtási időt. A kapott Quantum-ok száma változó, kliens Windows-on kevesebb Quantum-ot kap egy szál mind szerver Windows-on, illetve az előtérben lévő alkalmazás további Quantum-okat kap.

Válaszidők csökkentésére és elosztott rendszerekben felmerülő erőforrás és blokkolási problémák feloldására a Windows rendszer a Priority Boost funkciót használja. A felhasználó (vagy fejlesztő) által megadott prioritás a szálaknak az alap prioritása (base priority), a Priority Boost felülbírálhatja ezt a prioritást, melyet jelenlegi prioritásnak (current priority) nevezünk. Egyes események hatására a rendszer ideiglenesen megnövelheti egyes szálak prioritását. Például hosszabb várakozás után, egy erőforrás elérhetővé válásakor a várakozó szál prioritás növelést kaphat, így az ütemező nagyobb eséllyel fogja ütemezni és gyorsabban hozzáláthat a munkához. Az előtérben lévő alkalmazás (mellyel

a felhasználó aktívan foglalkozik) is automatikusan prioritás növelést kap. A Priority Boost Quantum növeléssel is járhat, tehát a szálak nagyobb időszeletet kapnak a processzor használatára.

A Windows is biztosít lehetőséget processzor affinitás beállítására, viszont a rendszer rávilágít egy lehetséges hátrányára az affinitásnak: Egy magasabb prioritású szálat nem fog másik processzorra ütemezni egy alacsonyabb prioritású szál affinitása miatt. Éppen ezért, lehetséges, hogy a szálunknak tovább kell várakoznia, mert egy nagyobb prioritású szál dolgozik a szálunknak megfelelő processzoron. Ennek feloldására vezették be az Ideális és Utolsó Processzor fogalmát. Ideális processzorral megadhatjuk, hogy melyik processzoron szeretnénk futtatni a szálunkat és az alábbi módon befolyásolhatjuk vele az ütemező döntését:

1. Amikor a szálunk processzorhoz jut, először az ideális processzort próbálja az ütemező biztosítani a számára.
2. Ha az ideális processzor nem elérhető akkor az előzőleg a szálat futtató processzor elérhetőségét ellenőrzi.
3. Ha az előző processzor sem elérhető akkor a többi processzor közül fog választani egyet a szál végrehajtására.

Nem garantált, hogy a szálunk mindig az ideális processzoron fog dolgozni. Ennek az algoritmusnak a cache miss minimalizálása a célja, a szál állapot processzorok közötti átvitelének minimalizálása.

Az asztali Windows rendszerek nem támogatják a valós idejű funkcionalitást, de a korábbi Windows CE rendszerek közelebb állnak egy teljes valós idejű rendszerhez. Windows CE rendszert kis teljesítményű, főleg beágyazott rendszerekre fejlesztettek, régebbi Windows telefonok és hordozható eszközök használták főleg a rendszert. A valós idejű feladatok végzését a pontos és determinisztikus megszakítások és megszakításokat kezelő szál segíti. Utolsó verziója 2013-ban készült (illetve a Windows 10 Mobile rendszer tartalmazza egyes részeit), utódja a Windows IoT Core, mely a modern kis teljesítményű Internet-of-Things eszközökre juttatja el a Microsoft ökoszisztémát és valós idejű környezetet együtt.

3. Valós idejű .NET API

Az említett ütemező funkcionalitások elérhetők a Microsoft folyamatos fejlesztése alatt álló .NET C# programozási nyelven keresztül is. Mivel egy virtuális gépben futó nyelvről van szó ezért fontos megemlíteni, hogy vannak veszélyei a C# alapú szigorú valós idejű alkalmazások fejlesztésének. A lefordított programot alkotó IL kód (Intermediate Language) futás közben Just-In-Time fordító segítségével fordul natív gépi utasításokra. Ezek a fordítási fázisok futás közben késleltethetik alkalmazásunkat. Szerencsére a modern .NET Core .NET Native előfordítási technológiájával és CoreRT futtatókörnyezetével eliminálható a JIT fordító használata [6]. A másik veszélyforrás a szemétgyűjtő használata. A memória rohamos csökkenése esetén a futtatókörnyezet szemétgyűjtést végez, melynek során a programunk szálait szüneteltheti. Egy szerencsétlen pillanatban kiváltódó szemétgyűjtés a határidők elmulasztásával járhat.

A futtatókörnyezet menedzseli az alkalmazásunk szálait is, éppen ezért szükséges megkülönböztetni a virtuális gép felügyelt szálait és az operációs rendszer számára ütemezhető natív végrehajtási szálakat. Egy a Thread osztállyal létrehozott felügyelt szálat egy natív szálnak feleltet meg a futtatókörnyezet. Habár a .NET Core dokumentációja felhívja a figyelmet: a felügyelt szálakat a futtatókörnyezet áthelyezheti másik natív szálakra, jelenleg ezt a funkciót még nem támogatja a virtuális gép. ThreadPool és egyéb beépített párhuzamos könyvtárak használata során a virtuális gép különböző felügyelt szálakat ütemez ugyanazon natív szálakra (ezzel lecsökkentve az operációs rendszer ütemezőjének terhelését), tehát ilyen esetekben a natív szálak tulajdonságainak módosítása

nem ajánlott. Natív szálak működésének konfigurálását csak akkor érdemes végezni mikor a szálat mi magunk hoztuk létre a Thread osztály segítségével [7].

```
static void Main(string[] args)
{
      Thread myThread = new Thread(new ThreadStart(ThreadJob));
      myThread.Start();
}

static void ThreadJob()
{
      //Do something...
}
```

A Thread osztálypéldány birtokában az adott szál prioritását állíthatjuk. Ezzel a tulajdonsággal a szálak prioritás osztályon belüli relatív prioritását adhatjuk meg.

```
static void Main(string[] args)
{
      Thread.CurrentThread.Priority = ThreadPriority.Highest;

      Thread myThread = new Thread(new ThreadStart(ThreadJob));
      myThread.Priority = ThreadPriority.BelowNormal;
      myThread.Start();
}
…
```

További konfiguráláshoz hozzá kell férnünk az alkalmazásunk Process osztálypéldányához. Ügyeljünk rá, hogy a Process osztály megvalósítja az IDisposable interfészt ezért használjuk a using blokkot az erőforrások helyes kezelésére! A Process példányon keresztül hozzáférhetünk és módosíthatjuk az alkalmazásunk prioritás osztályát, affinitását és Priority Boost engedélyét [8].

```
static void Main(string[] args)
{
     using (Process process = Process.GetCurrentProcess())
     {
          process.PriorityClass = ProcessPriorityClass.RealTime;
          process.ProcessorAffinity = new IntPtr(0x0002);
          process.PriorityBoostEnabled = true;
     }
}
```

A Process példányon keresztül hozzáférhetünk az alkalmazásunk szálaihoz és bővebb konfigurálásához ProcessThread példányokon keresztül. A Thread és ProcessThread két különböző osztály és különböző funkcionalitást is biztosítanak. Előbbivel a felügyelt szálunk állapotát és élettartamát kezelhetjük, még utóbbival natív szálak tulajdonságaihoz férhetünk hozzá. Több szálat láthatunk, mint amennyit létrehoztunk, mert a futtatókörnyezet is használ szálakat az alkalmazásunk futtatásához (például a szemétgyűjtő vagy a Just-In-Time fordító végrehajtásához). Ahhoz, hogy az aktuális végrehajtás alatt álló szálunk ProcessThread példányához hozzájussunk szükségünk van a szálunk natív azonosítójára. Ehhez a feladathoz a C# API nem biztosít metódusokat, P/Invoke segítségével kell a kernel32.dll-ben elérhető GetCurrentThreadId API vetületet használnunk. A visszatérő érték az aktuális végrehajtási szál natív azonosítója, melyet felhasználva hozzájuthatunk a szál ProcessThread példányához.

```
public static class ThreadExtension
{
     public static ProcessThread GetProcessThread(this Thread thread)
      {
            uint threadID = GetCurrentThreadId();
            using (Process process = Process.GetCurrentProcess())
                  return process.Threads.OfType<ProcessThread>()
                           .FirstOrDefault(pt => pt.Id == threadID);
      }

      [DllImport("kernel32.dll")]
      static extern uint GetCurrentThreadId();
}
…
static void Main(string[] args)
{
     using (ProcessThread pt = Thread.CurrentThread.GetProcessThread())
            pt.IdealProcessor = 7;
}
```

A fenti API-k elérhetők .NET Framework és .NET Core keretrendszerek használatával is. .NET Core-t használva még a .NET Native előfordítási technológiát alkalmazva is elérhetjük a szükséges metódusokat. A Windows 10 rendszerben támogatott Universal Windows Platform biztonsági okokból az API-t csak minimális mértékben támogatja, és legtöbb esetben a támogatott részek használata a Microsoft Store áruházi szabályzatába ütközik.

4. Összegzés

Manapság a valós idejű alkalmazások fejlesztésének igénye már nem csak az ipari, hanem az átlag felhasználói környezetben is jelentős. Az átlag felhasználói operációs rendszerek közül a Linux egyes változatai támogatnak valós idejű funkcionalitást. Azt vizsgáltuk, hogy Windows rendszerben, milyen ütemezési opciók és konfigurálási lehetőségek elérhetők, melyek mégis alkalmassá tehetik a Windows rendszert valós idejű szoftverfejlesztéshez.

A .NET C# programozási nyelv hatalmas fejlődéseken ment át az elmúlt években. Az elért sebesség növekedésnek és technológiai újításoknak köszönhetően sokkal magabiztosabban használhatjuk valós idejű környezetben. Megfelelő könyvtárakat használva a rendszer szintű szálak működésének konfigurálását is elvégezhetjük, ezzel tanácsokat adva az ütemezőnek az alkalmazásunk helyes ütemezéséhez.

Köszönetnyilvánítás

EFOP-3.6.3-VEKOP-16-2017-00001: Tehetséggondozás és kutatói utánpótlás fejlesztése autonóm járműirányítási technológiák területén – A projekt a Magyar Állam és az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Hivatkozások

[1] Dr. Illés Zoltán, Heizlerné Bakonyi Viktória, Illés Zoltán: Valós időben, valós világban: INFODIDACT 2015, Zamárdi (2015), ISBN: 978-963-12-3892-1

[2] Dr. Illés Zoltán: Operációs rendszerek 4. előadás, http://os.inf.elte.hu/index.php?link=ea (utoljára megtekintve: 2018.11.06.)

[3] Dr. Illés Zoltán, Szabó Dávid: Operációs rendszerek 5. előadás http://os.inf.elte.hu/index.php?link=ea (utoljára megtekintve: 2018.11.06.)

[4] Yoshitake Kobayashi: Deadline Miss Detection with SCHED_DEADLINE: ELC 2013. Embedded Linux Conference, San Francisco, California, 2013.02.20-22.

[5] Mark Russinovich, David A. Solomon, Alex Ionescu: Windows Internals, Part 1, 6th Edition, Microsoft Press, 2012, [359-485], ISBN: 978-0-7356-4873-9

[6] Szabó Dávid: C# Multi-Platform Környezetben, MSc Diplomamunka, 2018, https://edit.elte.hu/xmlui/bitstream/handle/10831/38929/Diplomamunka.pdf (utoljára megtekintve: 2018.11.06.)

[7] Bart de Smet: C# 4.0 Unleashed, Pearson Education, 2011, [1431-1452], ISBN: 978-0-672-33079-7

[8] Andrew Troelsen, Philip Japikse: Pro C# 7: With .NET and .NET Core, Apress, 2017, [631-641], ISBN: 978-1-4842-3017-6