MİKROBİLGİSAYARLAR İÇİN ÇOK GÖREVLİ GERÇEK ZAMANLI BİR İŞLETİM YAZILIMININ TASARIMI VE GERÇEKLEŞTİRİMİ

Kayhan İmre

Hacettepe Üniversitesi
Fen Bilimleri Enstitüsü Yönetmeliğinin
Bilgisayar Bilimleri Mühendisliği Anabilim Dalı İçin Öngördüğü
YÜKSEK MÜHENDİSLİK TEZİ
Olarak Hazırlanmıştır

Mayıs-1987

TEŞEKKÜR

Tez konumun seçiminden, tasarım ve gerçekleştirimine kadar tüm aşamalarda değerli öneri ve yardımlarını esirgemeyen, beni titiz bir çalışma ortamına yönelten, hocam Sayın Doç. Dr. Ali Saatçi'ye teşekkürü borç bilirim. Çalışmalarım sırasında diğer işlerimde yardımcı olarak tezime daha çok ağırlık vermemi sağlayan tüm çalışma arkadaşlarıma, ve bu tezin yazım işlerini gerçekleştiren Sayın Şerife Çürekçi'ye de ayrıca teşekkür ederim.

ÖZET

IBM PC türü bilgisayarlarda çok görevli gerçek zamanlı bir ortam oluşturmak için bir işletim yazılımı tasarlanmış ve gerçekleştirilmiştir. Bu yazılım hem bir işletim dizgesini gerçekleştirirken en alt katmanı oluşturmada hem de süreç denetimi gibi uygulama alanlarında doğrudan kullanılabilir. Yazılıma erişim, kullanıma sunulan hizmet yordamları aracılığı ile ve C dili olanakları kullanılarak sağlanır. Bu yazılımı kullanan uygulamalarda görev tanımlama, görevleri gerçek zaman saatine uyumlu ve koşut olarak çalıştırma, bu görevler arası zaman uyumlama düzeneklerini kurma, bellek ve kesilmelerin yönetimi gibi işlemlerin kullanımı amaçlanmıştır. Tasarlanan ve gerçekleştirilen yazılım C ve ASM86 dilleri kullanılarak oluşturulmuştur. Bu yazılım ana bellekte 5 KB'lık bir yer kaplamaktadır.

SUMMARY

In order to establish a real-time multitasking environment on IBM PC series or compatible microcomputers, an operating software package was designed and implemented. This software can be used either to constitute the lowest layer of a general purpose operating system or it can be used directly for dedicated applications like process control. The use of this software is provided by the features of the C programming language and some ready to use library routines. For the applications which use this software, it is aimed to define and to run simultaneous tasks in synchronization with real-time clock, to establish synchronization mechanisms among them, to manage some other hardware resources like memory and interrupts. The software is implemented by means of the C language and ASM86 symbolic programming language. The software occupies 5 kBytes of memory.

İÇİNDEKİLER DİZİNİ

ÇİZİMLER DİZİNİ

Çizim **Sayfa**

ÇİZELGELER DİZİNİ

1. GİRİŞ

Bir bilgisayar dizgesinden istenen hizmet, donanım ve yazılım diye adlandırılan kaynakların işe koşulmaları ile sağlanır. Bilgisayar dizgeleri üreten kuruluşlar pazarladıkları donanımın yanı sıra, bu donanımın kolayca kullanılmasını amaçlayan temel yazılımı da sağlamaktadırlar. Donanım üstündeki denetim işlevlerini yürüten, değişik fiziksel birimleri süren bu birimlerin ayrıntı ve karmaşıklığını gizleyerek bilgisayar dizgesini, mantıksal bir model çerçevesinde kullanıma sunan temel yazılım İşletim Dizgesi olarak adlandırılır.

Bilgisayar dizgesini oluşturan, Ana İşlem Birimi, Ana Bellek, ikincil bellekler, giriş/çıkış birimleri gibi donanımsal kaynaklar, dizgeden hizmet bekleyen tek bir iş tarafından tümüyle tüketilemezler. Dizge başarımını yükseltmek bu kaynakların eşzamanlı paylaşımlarıyla olanaklıdır. Çevrim dışı derlenmiş verilerin işlenmesine yönelik uygulamalarda kaynak paylaşımı ekonomik gerekçelere dayanırken, bilgisayarlı süreç denetimi gibi uygulamalarda çevrim içi derlenen verilere ve zamana dayalı denetimlerin gerekli kıldığı bir zorunluluktur. Süreç denetimi gibi, eşzamanlı işlevlerin birlikte yürütülmesi gereken uygulamalarda, bilgisayar dizgesi, koşut çalışan bağımlı bağımsız birçok görev arasında bölüşülür. Görev herhangi bir programın belirli bir veri kümesi ile işletimine verilen addır. Fiziksel bir sürecin denetlenmesi birden çok adımdan oluştuğundan, birlikte yürütülecek her adım için ayrı bir görevin kullanılması gerekir. Bu tür koşut işleme olanak sağlayan temel dizge yazılımları Çok Görevli olarak adlandırılır. Kaynak paylaşımının yapıldığı dizgelerde, dizgenin yanıt süresi, alınan hizmetin niteliği yönünüden önemli bir noktayı oluşturur. Yanıt süresi için bir üst sınırın çizilebildiği ya da gerçek zaman saatine tam

uyumun sağlanabildiği uygulamalara Gerçek Zamanlı uygulamalar denir. Bu tür uygulamalara alt yapı sağlayan İşletim Yazılımları Gerçek Zamanlı olarak tanımlanır. Yukarıda anılan çok görevlilik, aslında gerçek zamanlı niteliğin sağlanmasında kullanılan bir araçtır. Bu durumda koşut işlem ortamı kurarak hızlı yanıt almayı sağlayacak işletim yazılımları Gerçek Zamanlı-Çok Görevli Çekirdek Dizge Yazılımları olarak adlandırılırlar. Bu tür yazılımlar, kimi zaman değişik programlama dilleri tarafından kullanılan dizge işlevler bütünü olarak düşünülürken, kimi zaman da üzerinde çalışacağı donanımın parçası olarak salt okunur belleklerde pazarlanmaktadırlar. Çekirdek dizgenin varoluş biçimi ve özellikleri ne olursa olsun kullanıcıya özde sunduğu hizmetler standart sayılabilecek kapsamdadır. Bu hizmetler;

a. Görev Yönetimi
b. Gerçek Zaman Saatinin Kullanımı
c. Görevler arası zamanuyumlama ve iletişim... gibi alt başlıklarla sınırlandırılırlar.

Çekirdek dizge ile birlikte çalışacak Giriş/Çıkış Güdüm Dizgesi, bu dizge kapsamında düşünülmese de kimi temel giriş/çıkış aygıtlarına ilişkin hizmet yordamlarının sağlanması, uygulamalarda kolaylıklar sağlar. Geliştirilen her gerçek zamanlı çok görevli dizgeye uyumlu bir de giriş/çıkış dizgesi oluşturmak sözkonusudur.

Gerçek zamanlı çok görevli dizge ve giriş/çıkış dizgesi, birlikte kullanılmalarına rağmen iki ayrı yazılım paketi olarak düşünülmek zorundadırlar. Giriş/Çıkış işlevleri geniş çapta donanıma bağımlı olduğundan dolayı bu zorunluluk ortaya çıkmakta, yazılımın diğer kesimi olan çok görevli gerçek zamanlı dizgenin taşınabilirliğini sağlamayı ve tüm dizgenin karmaşıklığını azaltmayı amaçlamaktadır. Büyük bilgisayarlarda işletim dizgesinin alt katmanları çok görevli gerçek zamanlı çekirdek dizge

türünde bir yazılım katmanı kullanılarak gerçekleştirilmektedir. Ayrıca mikrobilgisayarlar için de bu türde çeşitli yazılım paketleri geliştirilmiş ve geliştirilmektedir. Yapılan tez çalışması kapsamında mikrobilgisayarlar için , özellikle 8088/86 serisi işleyiciler içeren IBM-PC uyumlu bilgisayarlarda kullanılabilecek türde bir gerçek zamanlı çok görevli çekirdek dizgeyi oluşturmak amaçlanmıştır. İşletim dizgesi türünde yazılımlara yatkın olması; bellek erişimi, mantıksal/aritmetiksel işlemlerin etkinliği; kısıtlamaların olmayışı ve genel amaçlılık gibi özellikler göz önüne alınarak C programlama dili çekirdek dizgeyi gerçekleştirmede kullanılmak üzere seçilmiştir. Çok görevli gerçek zamanlı dizge, işletim aşamasında sadece PC uyumlu bilgisayarın donanımsal kaynaklarını kullanmakta ve yeniden girilir olmamaları nedeni ile varolan yazılım kaynaklarına (MS-DOS gibi) erişim yasaklanmaktadır.

2. GENEL BİLGİLER VE KİMİ BİLİNEN GERÇEK ZAMANLI ÇOK GÖREVLİ ÇEKİRDEK DİZGELER.

2.1. Genel Bilgiler

Gerçek Zamanlı Çok Görevli Dizgelerin tanıtımını yapmadan önce anlatımda yeralan bazı temel kavram ve tanımlardan sözetmek gerekir. Anahtar niteliğindeki bu kavram ve tanımlar daha üst düzey kavramları oluşturmada kullanılacaktır.

Görev:

Görev, verilerinin saklandığı bir bellek alanıyla, yapacağı işin mantığını içeren komutlardan oluşan ve bir süreci denetlemeye adanmış yazılım bütününün anılan verilerle işletimine verilen addır.

Çok Görevlilik:

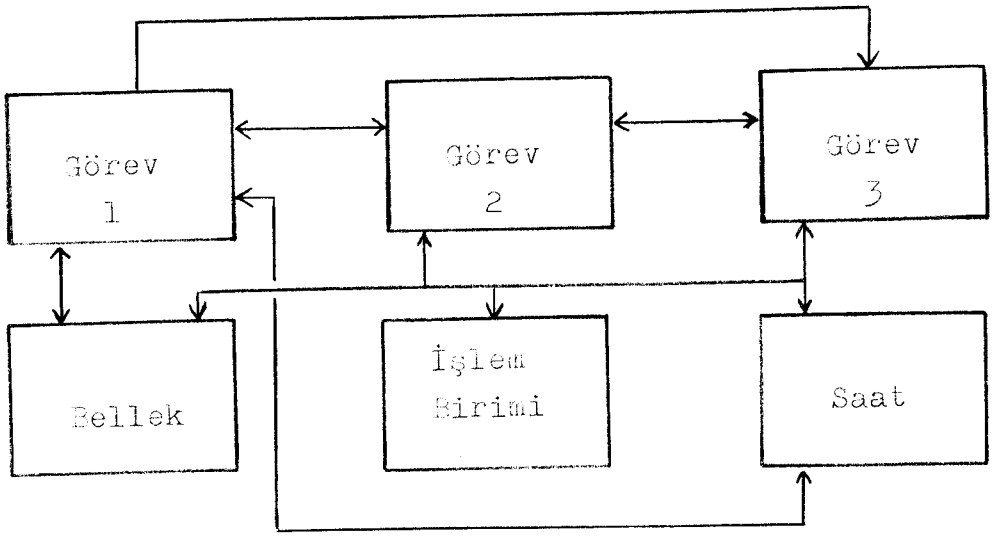
Dizgede birden fazla koşut görev tanımlama olanağı varsa ortam çok görevli olarak nitelenir. Dizgedeki görevlerin koşutluğunu sağlamada iki yöntem kullanılır. İlk yöntemde her göreve bir işlem birimi adanarak görevlerin koşut olarak çalışması sağlanır. Bu yöntemle elde edilen koşutluk gerçek koşutluk olarak adlandırılır. İkinci bir yöntem, görev sayısının işlem birimi sayısından fazla olduğu, özellikle tek işlem birimi içeren dizgeler için geçerlidir. Bu yöntem, işlem birimi ya da birimlerinin görevler tarafından zaman içinde paylaşılmasıdır. İşlem biriminin paylaşımı, hangi görevlerin ne zaman ve ne kadar süreyle işlem birimini kullanacağını saptayan ve bu kullanımı gerçekleştiren görev yönetici tarafından sağlanır. Görev yönetici tarafından sağlanan ve görev sayısının işlem birimi sayısından fazla olduğu durumdaki koşutluk ise görüntü koşutluk olarak adlandırılır.

Zaman Uyumlama:

Görev mantığındaki iş akışın, diğer görevlerden gelen istemler ya da dış uyarılılar doğrultusunda durdurulması ya da sürdürülmesi zaman uyumlama olarak adlandırılır.

Çok görevli ortamda gerçek zaman saatine uyumlu çalışma gerekseyen uygulamalarda, %60'a varan programcı zamanının sadece kimi temel düzenekleri oluşturmak için harcandığı deneysel olarak görülmüştür. Sözkonusu uygulamalar da kurulması gerekli ilk temel düzenek çok görevliliktir. Çok görevlilik, yazılımsal olarak gerçekleştirilmesi ve sınanması oldukça zor olan bir özelliğe sahiptir. Çok görevli ortam oluşturulduktan sonra birbirine koşut çalışan görevlerin zamanuyumlu biçimde çalışmasını sağlayan diğer bir takım düzenekleri de gerçekleştirmek gerekir. Uygulama gerçek zaman saatini de gerektirdiğinde koşut birden

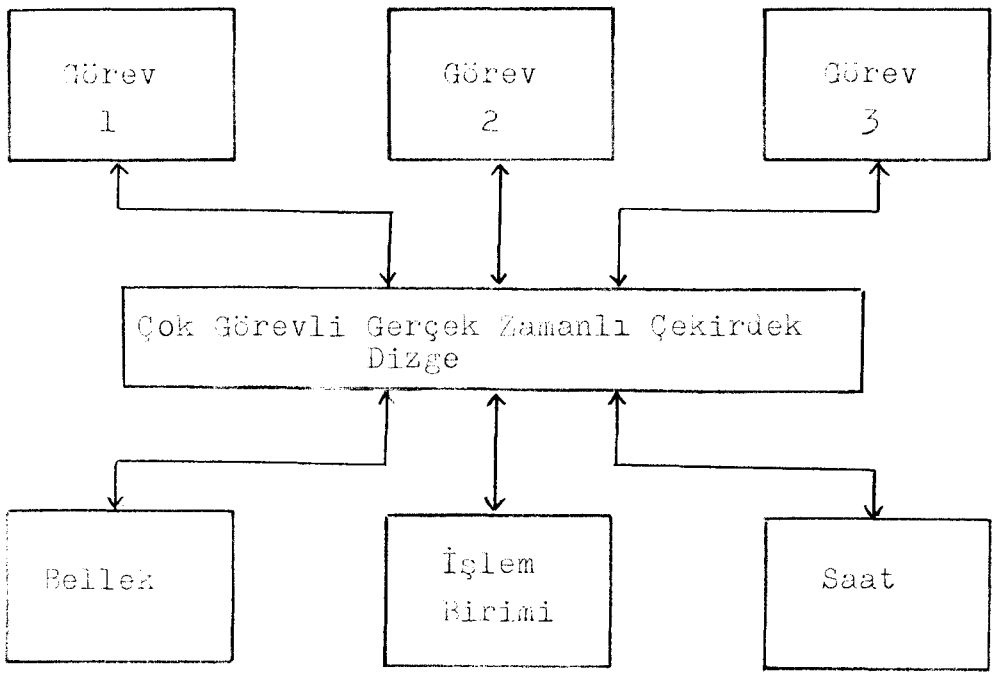
fazla göreve hizmet verebilecek gerçek zaman saatı oluşturmak sorunu iyice karmaşıklaştırmaktadır. Çizim 2.1'de verilen örnekte koşut üç görevin gereksediği iletişim ve kaynak paylaşımı sıkıca bağlı ilişkiler oluşturmaktadır. Sıkıca bağlı ilişkileri oluşturmak, bu ilişkilerin doğruluğunu sınamak oldukça zordur. Ayrıca her uygulama için görevler arasındaki karmaşık ilişkileri kurmak yazılım maliyetini artıracaktır.



Çizim 2.1. Çok Görevli bir ortamın hiçbir altyapı kullanılmadan gerçekleştimi.

Uygulamalarda kullanılan çok görevli altyapıyı uygulama kapsamından çıkararak farklı bir hizmet birimi olarak görmek, görevler arası ilişkilerde birebir bağımlılığı aşmaya doğru

atılan adımdır. Çok görevli altyapının bir kez doğru olarak gerçekleştirilmesi, her uygulamada bu altyapının kulllanılması maliyeti düşürecek ve güvenilirliği artıracaktır. Bu yaklaşımda ilişkiler oldukça yalınlaşarak, çok görevli ortamın yapısını az bilen programcıların bile rahatlıkla uygulamalar yapabilecekleri bir düzeye inecektir. Çizim 2.1'deki ilişkisel gösterim, Çizim 2.2'deki yalın biçimini alacaktır. Uygulamada tüm ilişkiler bir yönetim birimine bağlıdır ve bu birim çekirdek dizge olarak adlandırılmıştır.



Çizim 2.2. Çok Görevli Gerçek Zamanlı Altyapı ile Çok Görevli Uygulamanın Gerçekleştirimi ve İlişkisel Gösterim

Görevler arasındaki ilişkileri yalınlaştıran yönetim birimi, kendi içinde bu ilişkileri doğru yönlendirebilmek için ilişkilerin türlerine göre alt işlevsel birimlere ayrılmıştır. Görevler arasındaki ilişkilerin ortaya çıkardığı temel işlevler genel çizgileri ile şunlardır:

a. AİB'nin Yönetimi,
b. Gerçek Zaman Saatinin Kullanımı,
c. Zaman Uyumlama Düzenekleri,
d. Bellek Yönetimi,
e. Kesilmelerin Yönetimi,

a. Ana İşlem Biriminin Yönetimi:

Dizgede tanımlanmış görevlerin ana işlem biriminin zaman içindeki paylaşımını sağlayan işlevsel birim AİB Yönetimi kesimidir. Bu işlevsel kesim görev yönetici olarak da adlandırılır. Çekirdek dizgeyi oluşturan en önemli kesimdir. Görev yönetici hangi işletim dizgesinde olursa olsun görev kuyruklarını giriş olarak kullanır. Görev yöneticinin çıktısı ise bir görevin işlem birimine anahtarlanmasıdır. Görev yönetimi için benimsenen algoritma, çekirdek dizge ile yapılacak tüm uygulamaları etkileyecektir. Hiçbir görevi uzun süre bekletmeden, gerektiğinde öncelikleri de göz önüne alarak görev anahtarlamayı gerçekleştirmek görev yönetim algoritmalarının karmaşıklaşmasına yol açar. Gerçek zamanlı dizgenin başarımı veri toplama, işleme ve veriyi saklamadaki hızı ile ölçülür. Görev yönetici dış ortamdan gelen yoğun veriyi alan görevleri işlem birimine anahtarlarken bu veriyi işleyen görevleri de veri giriş hızına uygun biçimde işlem biriminden yararlandırmalıdır.

b. Gerçek Zaman Saatinin Kullanımı

Dizge donanımı içinde yer alan gerçek zaman saati, görevlerin dış dünya ile olan ilişkilerinde zamanlama sorununu çözmek için kullanılır. Gerçek zaman saati iki olay arasındaki zaman aralığının bulunması ya da bir olayın oluşmasını beklemedeki zaman sınırını oluşturma gibi sorunların çözümünde yararlı olur. Dış ortamın denetimini hedef alan işletim dizgelerinin üstesinden gelmesi gereken sorun en çok istenen kaynağın en iyi kullanımıdır. Çok iddialı biçimde bu kaynağın zaman olduğunu söylemek hiç de yanlış olmaz. Gerçek zaman saatini yöneten kesim bu kaynağı çok görevli ortama en iyi biçimde sunmakla yükümlüdür. Koşut olarak birden fazla görevin gerçek zaman saatine uyumlu istemlerde bulunması istemlerin çelişmesine yol açabilir. Bunlardan biri gerçek zaman saatinin kurulması ile ilgilidir. Bir görev dizgedeki saati kurduğundan diğer bir görevin saati yeniden kurması işlemlerde çelişkilere yol açar. Bu bakımdan her görevin sanki saati sadece kendisi kullanıyormuş gibi görmesini sağlayacak, başka bir deyişle görüntü saatlere ilişkin altyapıyı oluşturmak gerekecektir. Gerçek zaman saati ile ortaya çıkabilecek bir başka sorun da görev uyandırma hizmetleri ile ilgilidir. Görevler gerçek zaman saatini işleyen kesimden belirli bir süre boyunca uyuma, bu sürenin bitiminde uyandırılarak işlem birimini kullanma isteminde bulunabilirler. Uyuyan görevleri izleyen kesimin dizgeye getireceği yükü en aza indirmek gereklidir. Zaman içinde uyuma isteminde bulunmuş birden fazla görevin raslantısal olarak aynı anda uyanması gerektiğinde hangi görevin işlem birimini ele geçireceğine karar vermek de çözülmesi gerekli bir başka sorundur. Bunun için istemde ilk bulunan ya da en öncelikli olan görev işlem birimini ele geçirir, diğerleri ise işlem birimini ele geçirmek üzere kuyrukta hazır beklerler.

c. Zaman Uyumlama Düzenekleri.

Çok görevli ortamda birbirinden bağımsız görevler koşut olarak çalışırlar. Zaman zaman ortak veri alanları kullanmak ortak bir aygıta erişmek ya da görevler arası iletişim kurmak istenir. Koşut çalışan birbirinden bağımsız görevler tasarlanırken normal bir programlama yaklaşımından farklı davranmak zorunludur. Her iki görevin aynı anda aynı veri alanına erişmesi ya da iletişim sırasında alıcı durumundaki bir görevin veri hazır olmadan veriye ulaşmak istemesi denetlenmelidir. Ortaya çıkan bu tür sorunları çözmek üzere kimi düzenekler öngörülmüştür. Zaman uyumlama düzenekleri olarak anılan bu düzenekler semafor adı verilen özel göstergeleri kullanan işleçler aracılığı ile görevlerin ortak veri alanlarına erişmelerini denetlerler.

d. Bellek Yönetimi

Görevler tarafından gereksenen bellek alanları bellek yönetim kesimi aracılığı ile karşılanır. Görevler zaman içinde kullandıkları bellek alanlarını bırakabildikleri gibi, bellek isteminde de bulunurlar. Çok görevli ortamdaki devingen bellek istemleri bellek yönetiminin gerekliliğini ortaya çıkarır. Görevlerin yığıt ve veri alanları bellek yönetici tarafından sağlanan bellek bölümü içinde tanımlanır. Görev dizgede kaldığı sürece bu bellek alanını kullanır ve sonlandığı zaman belleği dizgeye geri verir.

e. Kesilmelerin Yönetimi

Gerçek zamanlı dizgelerin birincil nitelikleri çok görevlilik ve dış uyarıların ele alınması olarak belirlenmiştir. Çok görevlilik görev yönetici tarafından sağlanmaktadır. Dış uyarılar ise kesilmelerin yönetimi kapsamındadır. Dizge, kesilme

hizmet yordamlarının tanımlanmasına ve kesilmelerin denetlenmesine olanak tanımalıdır. Dış ortamdan gelen verilerin beklenmesinde kullanılabilecek en iyi araç kesilme altyapısıdır. Kesilme altyapısı kullanılmadan yapılan giriş çıkış işlemlerinde işlem biriminin bir kısım zamanı sınama ile harcanmak durumundadır. Ayrıca kesilme altyapısı ile dış ortamdan gelen uyarıya en hızlı biçimde yanıt vermek ve zamanuyumsuz bir biçimde gelişen dış uyarıları sağlıklı olarak ele almak ancak kesilmeler yolu ile olanaklıdır.

2.2. Kimi Bilinen Gerçek Zamanlı Çok Görevli Çekirdek Dizgeler.

Gerçek zamanlı uygulamaları destekleyecek standart işletim dizgelerinin bulunmaması, bilgisayar donanımı ve yazılımı üreten kuruluşları, kendi yazılımlarını gerçekleştirmeye yöneltmiştir. Birçok kuruluş bilgisayar ürünleri yanında destek niteliğinde gerçek zamanlı çok görevli altyapıyı da sağlamıştır.

Kimi donanım ürünleri geliştiren kuruluşlar, donanımları üzerinde salt okunur bellekte çok görevli gerçek zamanlı çekirdek dizgeyi de pazarlamaktadırlar. Bu türde bir yazılım salt okunur bellekte pazarlanmasından dolayı silikon yazılım (silicon software) olarak adlandırılmaktadır. Kimi yazılım üreten kuruluşlar da, ürettikleri yazılım paketleri içinde çok görevli gerçek zamanlı çekirdek dizgeyi de, öngörmektedirler. Doğal olarak yazılım paketleri değişik donanımlara uyarlanabilecek esneklikte tasarlanmaktadır.

Çok görevli dizgeler amaçlarına göre iki kategoriden birine düşmektedirler: Genel amaçlı (multipurpose) ve gömülü (embedded) dizgeler.

Genel amaçlı gerçek zamanlı dizgeler terminali, klavyesi, anabelleği ve giriş/çıkış birimleri ile tam bir mikrobilgisayar dizgesi çerçevesinde kullanılır. Süreç denetiminde ya da veri giriş uygulamalarında kimi özel amaçlı donanım kesimi de dizgeye eklenir.

Gömülü gerçek zamanlı dizgeler genellikle mikroişleyici tabanlı özel dizgeler üzerinde çalışan ve büyük bir dizgenin kimi belirgin kesimlerini denetleyen silikon yazılımlarıdır. Sadece amaçlanan bir donanımı denetlemek için öngörülmüş dizgelerdir. Bir sayısal telefon santralında anahtarlama işlemini denetleyen adanmış bir bilgisayar donanımı üzerindeki gerçek zamanlı denetimi gerçekleştiren yazılımlar bu tür dizgelere örnek gösterilebilir. Daha gelişmiş bir başka örnek de hasta izleme donanımlarına ilişkin denetim dizgeleridir. Bu donanımlar üzerindeki gerçek zamanlı dizge, salt hasta izleme amaçlı olarak donanıma gömülmüş donuk bir dizgedir.

Çizelge 2.1. Bilinen Kimi Gerçek Zamanlı Çok Görevli Çekirdek Dizgeler

Adı	Gerçekleştirildiği İşlem Birimleri
Mini-Exec	Intel 8080, 8085, Zilog Z8
VRTX	Zilog 8002, Intel iAPX-86, Motorola 68000
ZRTS	Zilog Z8001, Z8002
iRMX-86	Intel iAPX-86

Çizelge 2.1.'de verilen çok görevli gerçek zamanlı dizgeler silikon yazılım niteliğini taşırlar.

Data General firmasının AOS/VS (Advanced Operating System/Virtual Storage), RT32 (Real Time 32) ve MV/UX (MV/Unix) işletim diz-

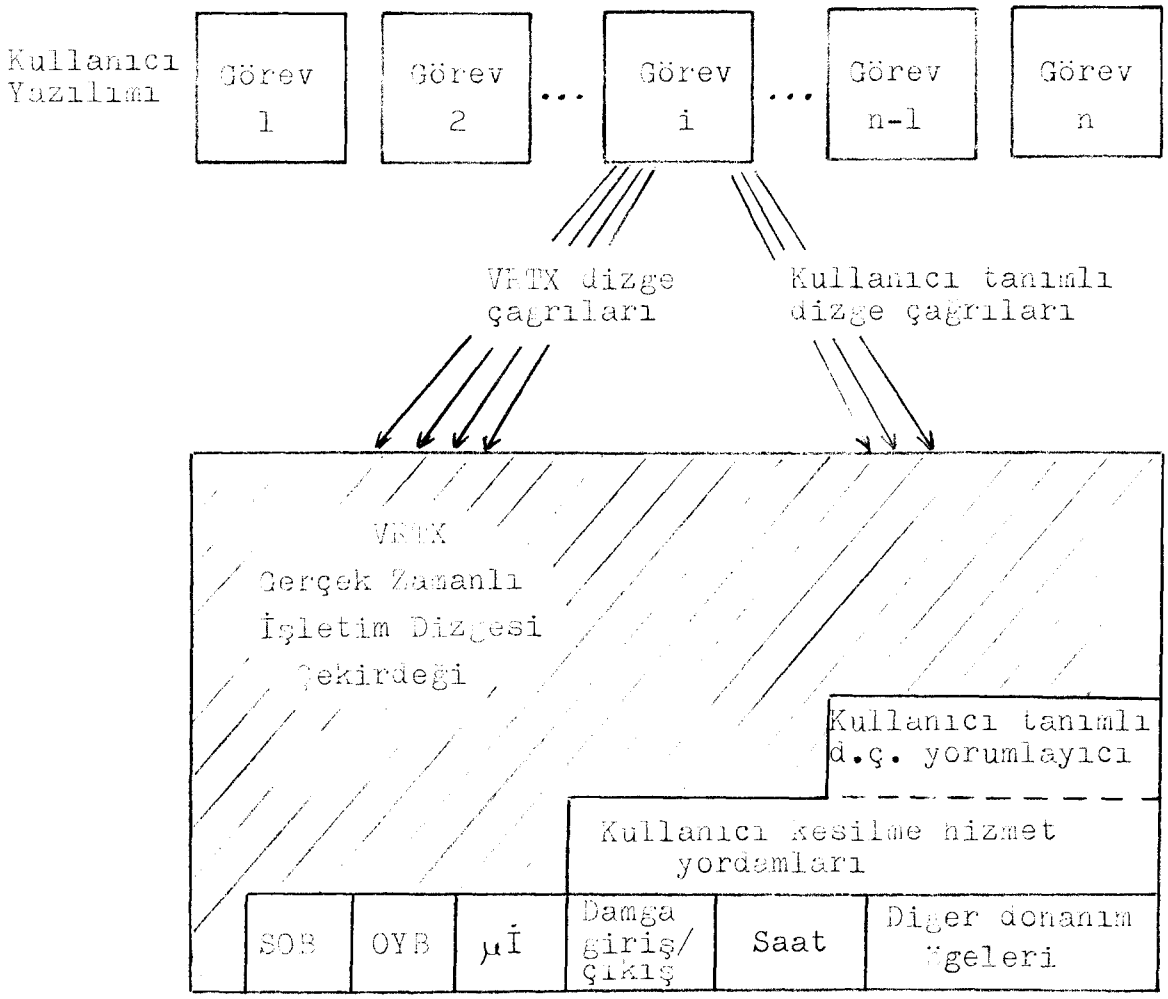
geleri üzerinde çalışan C derleyicisi için çok görevli gerçek zaman nitelikte kitaplık işlevleri öngörülmüştür.

2.2.1. VRTX Çok Görevli Gerçek Zamanlı İşletim Dizgesi

VRTX, üç yaygın mikroişleyici Motorola 68000, İntel 8086-88 ve Zilog Z8002 için gerçekleştirilmiş çok görevli gerçek zamanlı bir yazılımdır. VRTX'in içerdiği kesimler:

a. Kesilme destekli görev yönetimi,
b. Görevler arası iletişim ve zaman uyumlama,
c. Devingen ve durağan bellek yönetimi,
d. Gerçek zaman saati kullanımı,
e. Damga giriş/çıkış işlevleri.

VRTX donuk bir yazılım olmak yerine kullanıcının geliştirmesine açık bırakılmıştır. Bu amaçla kullanıcıya dizge çağrıları tanımlama olanağı verilmiştir. Ayrıca dizgenin çalıştırılacağı donanıma bağımlı özellikler yoktur. VRTX'e ilişkin dizge görünümü Çizim 2.3'de verilmiştir.



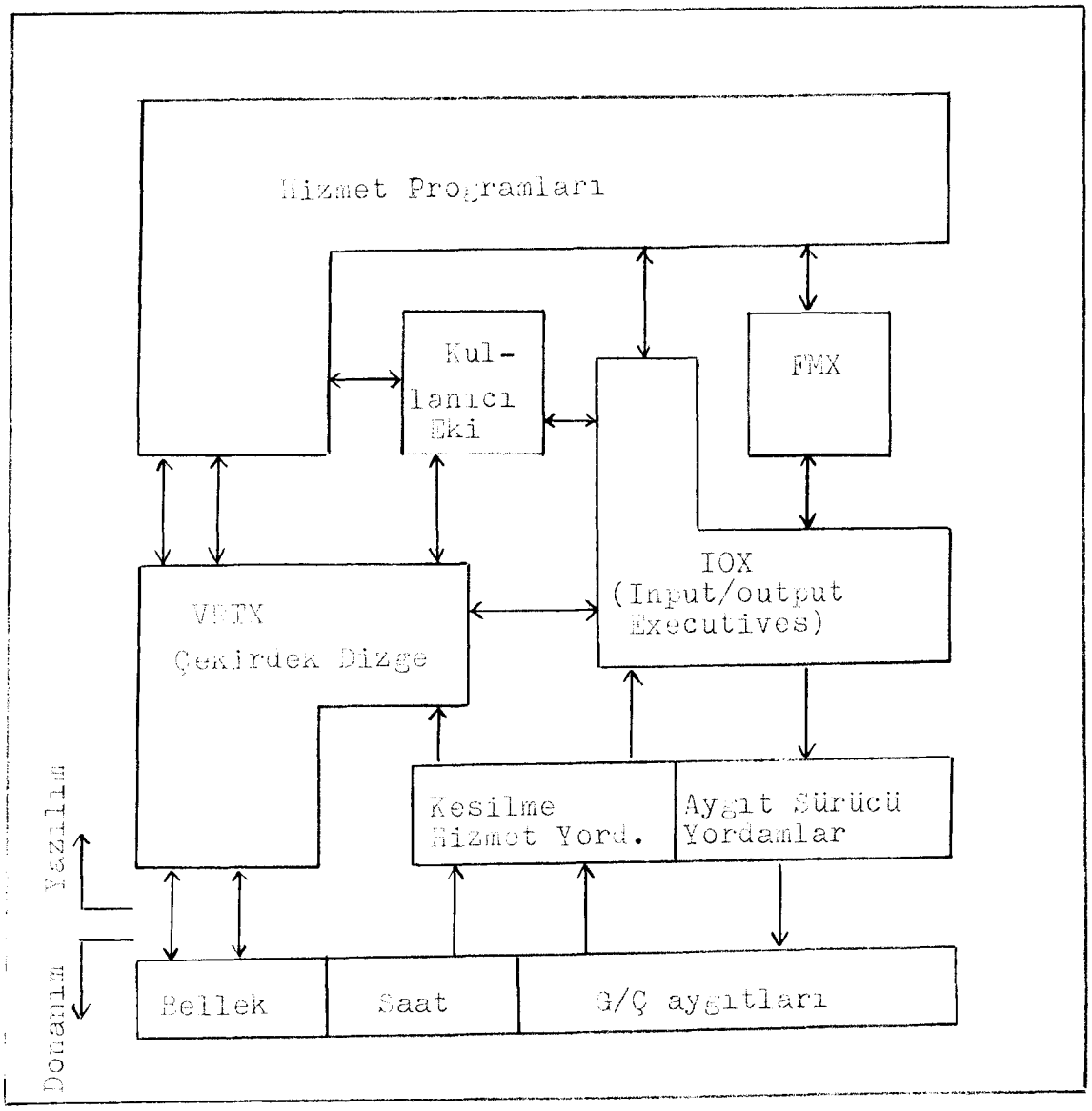
Çizim 2.3. VRTX'in mimarisi

Çok görevli gerçek zamanlı çekirdek dizge, VRTX, sağladığı 22 dizge çağrısı ile 16 ikil mikroişleyiciler üzerindeki gerçek zamanlı uygulamaları kolaylaştırır.

VRTX başka işlevsel öbekler ile birleşip işletim dizgesi oluşturmada kullanılabilir. İşletim dizgesini oluşturmak için giriş/çıkış aygıtlarının sürülmesi ve kütük yönetim dizgesi

gerçekleştirmek gereklidir. VRTX'e uyumlu olarak öngörülmüş giriş/çıkış sürücüleri ve kütük yönetim dizgesi vardır.

IOX (Input Output Executive) gerçek zamanlı kapsamda kullanılabilecek giriş/çıkış işlevlerini içeren yazılım paketidir. Kütük yönetim dizgesi ise, FMX (File Management Executives) olarak adlandırılan yazılım parçasıdır. VRTX'e destek niteliğindeki bu işlevsel birimler kullanılarak, Çizim 2.4.'de ilke çizimi verilen işletim dizgesi gerçekleştirilebilir. Görev yönetimi, giriş/çıkışların yönetimi ve kütük yönetiminin bağımsız birimler olarak düşünülmesi gerçekleştirimi kolaylaştıran, birimsel yaklaşıma iyi bir örnek oluşturulur.



Çizim 2.4. VRTX ile oluşturulan bir işletim dizgesi.

Çizelge 2.2. VRTX'te Kullanıcı görevlerini denetlemek için kullanılan 22 dizge çağrısı.

Görev Denetim	
Task Create	Bir kimlik numarası, öncelik ve başlama adresi ile çalışmaya hazır görev tanımlar.
Task Delete	Tanımlanan görevi dizgeden çıkarır
Task Suspend	Belirlenen görevi beklemeye alır.
Task Resume	Önceden beklemeye alınmış bir görevi çalışmaya hazır konuma alır.
Task Priority	Belirlenen görevin önceliğini değiştirir.
Task İnquiry	Bir görevin kimliğini, önceliğini ve/ya da durumunu sorgulamak için kullanılır.
İletişim ve Zaman Uyumlama	
Post Message	Belirlenen bir posta kutusu aracılığı ile bir görevden diğerine ileti gönderir.
Pend For Message	Ya bir görevden ya da bir kesilme hizmet yordamından bir posta kutusu aracılığı ile gelen iletiyi alır. Eğer ileti gelmemiş ise, ileti gelene kadar görev beklemede bırakılır.

Çizelge 2.2. (Devamı)

Damga Giriş/Çıkış	
Get Character	Giriş/çıkış aygıtından bir damga okur.
Put Character	Giriş/çıkış aygıtına bir damga gönderir.
Wait For Charecter	Giriş/çıkış aygıtından bir damga gelinceye kadar görevi bekleme konumunda tutar.
Saat Yönetimi	
Task delay	Belirlenen saat periyodu boyunca görevi bekletir.
Get time	Dizge saatinini şimdiki değerini alır.
Set time	Dizge saatine değer atamak için kullanılır.
Time_Slice	Görevlere uygulanan "round robin" yönetim biçiminde zaman dilimini saptar.
Bellek Yönetimi	
Get block	Bellek öbeği atama işlemini gerçekleştirir.
Release block	Bellek öbeğini dizgeye geri verir.

Çizelge 2.2 (Devamı)

Kesilme Arabirimi	
Post from interrupt	Kesilme hizmet yordamından bir göreve posta kutusu aracılığı ile ileti gönderir,
Exit from interrupt	Kesilme hizmet yordamının bitiminde kesilen göreve geri dönüş için kullanılır.
Timer İnterrupt	Saat kesilme sıklığını ayarlamak için kullanılır.
Receiver İnterrupt	Giriş/çıkış aygıtından gönderilecek damganın hazır olduğunu bildirmek için kesilme hizmet yordamı tarafından kullanılır.

2.2.2. AOS/VS Altında Çalışan Derleyicilere Sağlanan Gerçek Zamanlı, Çok Görevli Altyapı.

AOS/VS (Advanced Operating System/Virtual Storage) İşletim dizgesi kapsamındaki çok görevlilik, bu işletim dizgesi altında çalışan derleyicilere de kitaplık aracılığı ile yansıtılmıştır. AOS/VS C programlama dili derleyicisi bu programlama dillerinden biridir ve kullanıcılara bağımsız görevler tanımlamaya olanak sağlar.

Çizelge 2.3. AOS/VS C Programlama Dili ile Çok Görevli Ortamın Kullanımına İlişkin Tipik Uygulamalar.

Kullanım	Açıklama
Döküm	Birden fazla döküm işi yapılacak ise koşut döküm görevleri tanımlanarak, kütük açma, kapama, okuma ve yazma işlemleri için beklemelerde işlem birimi diğer görevlere yöneltilir.
Etkileşimli Programlar	Terminal ekranını sürekli günleyen ve klavyeden gelen damgaları okuyan ayrı koşut görevler tanımlanır.
Hizmet Görevleri	İleti kuyrugundan istemleri bekleyen hizmet yordamları, istemleri yitirmeden bekler. İstem gönderilince bekleme konumundan çıkıp hizmete yönelirler.
Veri Giriş Sistemleri	Çok terminalli ortamda, terminallerin sürülmesi, için kullanılır. Çok görevli ortam, her terminal için aynı programın çalıştırılmasından daha az dizge kaynağı harcar.

C Programlama dili ile yazılan yordamlar çok görevli işlevler yardımı ile görev olarak tanımlanarak, bir programın içinde çok görevli bir ortam oluşturulur. Görevler, bu altyapıyı kullanarak görev yönetimini, kaynak paylaşımını, gerçek zaman saati kullanımını sağlar.

Çok görevli ortamda tanımlanan uygulamalar içinde oluşan görev yönetimine ilişkin işletim zamanı hataları evrensel bir değişken içinde saklanır. Bu değişken içindeki kod dizgede yapılan son

hatayı belirler. Ancak çok görevli ortamda koşut çalışan diğer görevlerin yaptığı hatalar ile karışma olasılığı vardır. Belirtilen bu nedenle hata kodu değişkeni çağıran görev düzeyindedir, yani her görevin bir hata kodu vardır.

Çok görevli işlevler Çizelge 2.4.'de belirtilmiştir. Ayrıca gerçek zaman saatine ilişkin görev uyutma işlevi "sleep" de öngörülmüştür. "Sleep" işlevi belirtilen saat periyodu için görevi bekleme konumunda tutar.

Çizelge 2.4. AOS/VS C Programlama Dilindeki Çok Görevli İşlevler

İşlev	Açıklama
mfinit	Belirli bir anda belirtilen kütüğe sadece bir görevin erişmesini sağlar.
midkill	Kimlik numarası belirli görevi dizgeden çıkarır.
midready	Kimlik numarası verilen görevi hazır konuma alır.
midsuspend	Kimlik numarası verilen görevi bekleme konumuna alır.
mkill	Şimdiki görevi sonlandırır.
mlock	Kritik kesime girişi kilitler.
mlongjmp	Başka bir göreve geçişi sağlar.
mnodirect	Görevi kesilmelere karşı korur.
mprikill	Verilen öncelikteki görevleri sonlandırılır.
mpriready	Verilen öncelikteki görevleri bekleme konumuna alır.
mredirect	Görevin kesilmelere karşı korumasını kaldırır.
msuspend	Şimdiki görevi bekleme konumuna alır.
mtask	Yeni bir görev yaratır.
munlock	Kritik kesimden çıkışta, kiritik kesime girişi serbest bırakır.

AOS/VS altında çalışan C dili ile çok görevli bir uygulama örneği listesi Çizim 2.5' de verilmiştir. Bu örnekte ikinci görev başlatılmadan önce ana görev kritik kesime girer ve ikinci görev "mlock" ile kritik kesime erişme girişiminde bulunur. Ana görev kritik kesime girişi "munlock" ile serbest bırakana kadar ikinci görev bekleme konumunda kalır.

```
#include  < multitask.h >
MLCK lock = MLCK_INIT; / * ikili semafor */
void worker() {
     printf("Workers Started.\n");
     sleep (5);
     mlock (&lock, MWAIT|MNOPROTECT);
     printf ("Worker gained lock.\n");
     munlock (&lock);
}
main () {
        printf ("Before tasking worker.\n");
        mlock (&lock, MWAIT|MNOPROTECT);
        if (mtask (worker, 1000)){
           perror ("Mtask");
           exit (1);
         }
       printf("Unlocking lock\n});
       sleep (15);     /* Worker kritik kesime girmeyi
                                  deneyecek */
       munlock(&lock); /* kritik kesim worker'a açıldı */
       sleep (5);      /* worker'ın bitimini bekle     */
}
```

Çizim 2.5. Çok Görevli Ortamda Kritik Kesim Oluşturma Örneği

3. ÇEKİRDEK DİZGENİN GERÇEKLEŞTİRİMİ

Bu tez kapsamında gerçekleştirilen Gerçek Zamanlı Çok Görevli Yazılım Paketi tümü ile IBM-PC,XT,AT türü mikrobilgisayarlar için öngörülmüş ve işlevsel birimlerden oluşturulmuştur. Bu işlevsel birimler birbirinden bağımsız yapıdadır ve aralarındaki eşgüdüm, görev yönetici olarak nitelenebilecek özel bir birim tarafından sağlanmaktadır. Dizgenin kalbini oluşturan görev yönetici, gerçek zaman saati kesilmeleri aracılığı ile işlevini yerine getirir. Dizgedeki diğer birimlerin gereksediği altyapıyı sağlar. Buna göre dizge; görev yönetici, işlevsel birimler ve kullanıcının tanımladığı görevlerden oluşur. Gerçekleştirimin anlatımında bu sınıflama gözönünde tutulmuştur.

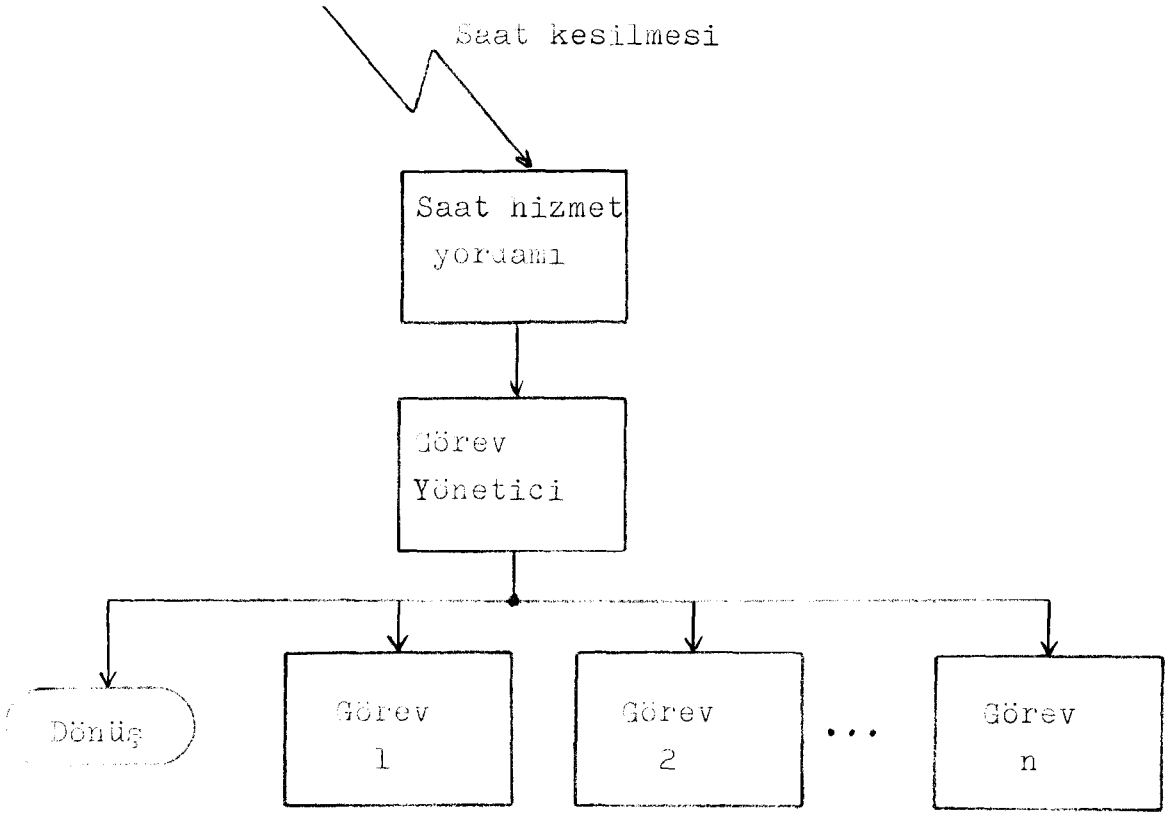
Dizgenin birleşenleri:

1. Görev yönetici,
2. İşlevsel birimler
   a. Görev anahtarlama işlevleri,
   b. Gerçek zaman saati işlevleri,
   c. Zaman uyumlama işlevleri,
   d. Giriş/çıkış işlevleri,
   e. Bellek yönetim işlevleri,
   f. Kesilme yönetim işlevleri,
   g. Diğer işlevlerden oluşmakta ve bu sırada açıklanmaktadır.

Gerçekleştirim açıklamalarının son kesiminde görevlerin yapıları verilmektedir.

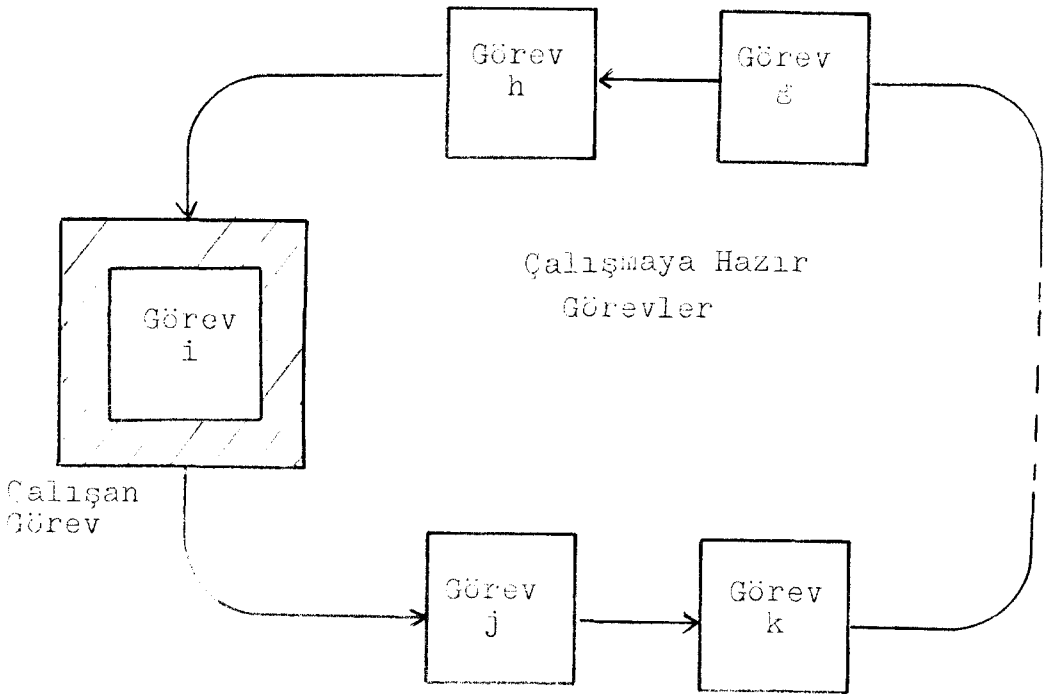
3.1. Görev Yönetici

Dizgenin durumunu sürekli olarak izleyen, gerektiğinde işlem biriminin denetimini başka görevlere veren birim görev yönetici olarak adlandırılır. Görev yöneticinin bu işlevini yerine getirmesi, görevlere koşut çalışmasını gerektirir. Gerçekleştirimde tek işlem birimi bulunan bir bilgisayar dizgesi kullanıldığından görüntü koşutluk söz konusudur. Bu amaçla belirli zaman aralıkları ile işleyiciye kesilme üreten gerçek zaman saati kullanılmıştır. Programlanabilir nitelikte olan bu kesilme altyapısına hizmet veren kesim, gerçek zaman saati ve görev yönetici programlarını içermektedir. (Çizim 3.1.)



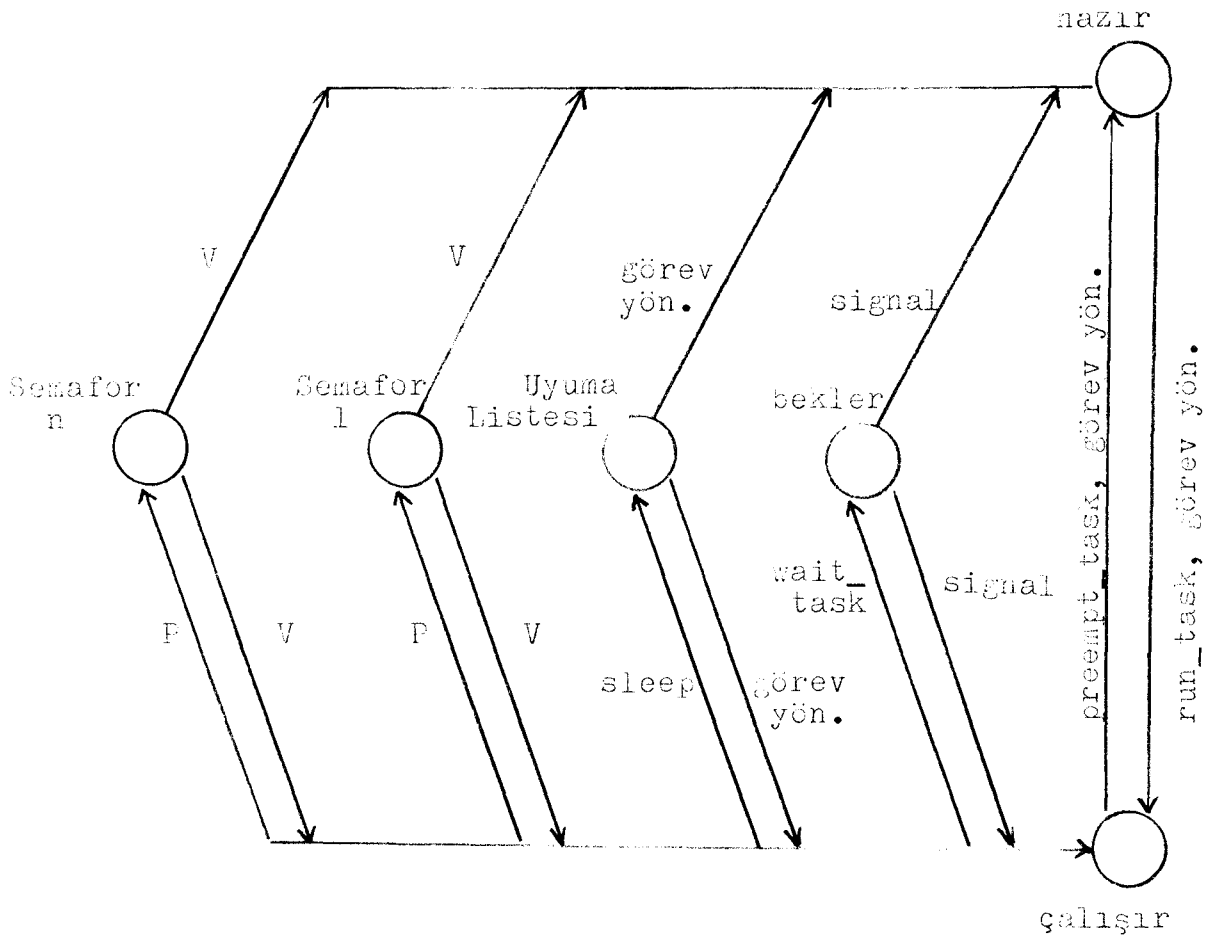
Çizim 3.1 Gerçek zaman saati ve görev yönetici ilke çizimi.

Görev yöneticinin en önemli işlevi çalışma zaman dilimini dolduran görevi kesip işlem birimini hazır görevler kuyruğunun başındaki göreve vermektir. Görev yönetimi için seçilen yöntem "round-robin" diye anılan yöntemdir. Bu yöntemde her göreve eşit zaman dilimleri ayırılmıştır. Görev, ayırılan bu zaman dilimi boyunca işlem birimini kullanır, eğer denetimi bırakmazsa görev yönetici bu görevi hazır görevler listesinin sonuna koyar ve bu listesindeki ilk görevi çalışır konuma alır. Eğer dizgede çalışan tek görev varsa yeni görevler tanımlanmadığı sürece işlem birimini kesintisiz kullanır. Görev yönetiminde kullanılan yöntem Çizim 3.2'de gösterilmiştir.



Çizim 3.2. "Round-robin" görev yönetimi.

Kullanılan yöntemde görevlerin öncelikleri çalışma zamanlarını etkilememektedir. Öncelikli görevlerin işlem birimini kullanım zamanlarını arttırmak için kullanılan yapılardan biri de uyuyan görevler listesidir. Gerçek zaman saati işlevleri kapsamında gerçekleştirilen "sleep" işlevi bu altyapıyı sağlar. Bu listedeki görevler belirtilen saat vuruşu sayısınca bekler durumda kalır. Bu görevlerden uyuma süresi dolanların uyandırılması görev yöneticinin sorumluluğundadır. Uyandırılan görev o anda çalışan görevden yüksek öncelikli ise çalışır konuma, değilse hazır konuma alınır. Görev yönetimi ve diğer işlevler kapsamında görevler kuyruklar aracılığı ile yönetilirler. Gerçekleştirilen dizge kapsamında kullanılan görev kuyrukları Çizim 3.3'deki çizge ile gösterilmiştir. Çizgede düğümler görev durum kuyruklarını göstermektedir. Geçişleri sağlayan işlevler okların üzerinde belirtilmiştir.



Çizim 3.3. Görev Durum Çizgesi.

Çizim 3.3'deki çizgede belirlenmiş durumlardan dört tanesi dizge içinde tanımlanmıştır. Bunlardan üçü görev anahtarlama işlevleri kapsamında ele alınan hazır, çalışır, bekler durumlarıdır. Uyuma listesi ise gerçek zaman saati işlevleri kapsamında ele alınmıştır. Bu durumların dışında kalanlar semaforları içerir. Bu semaforların tanımlanması dizgeyi kullanacak olana

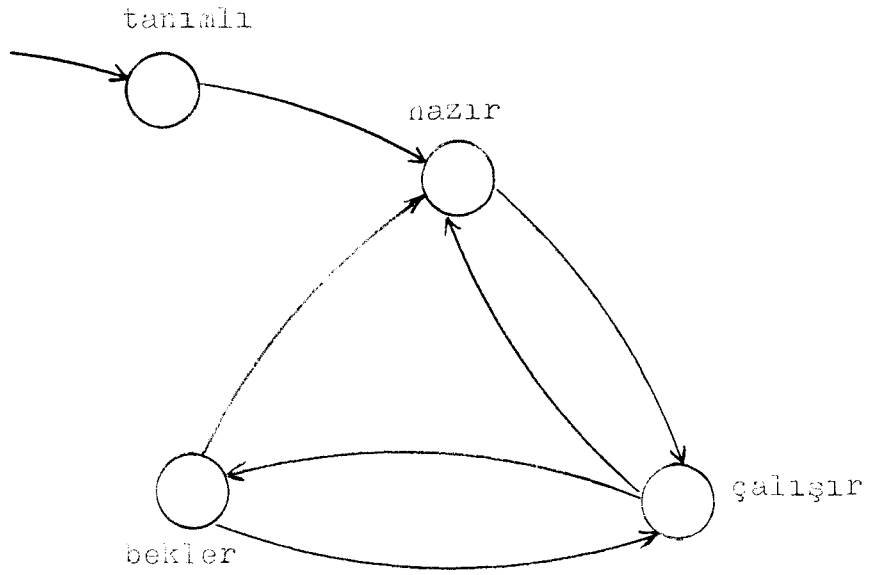
bırakılmıştır. Semaforların tanımlanması ve kullanılması zaman uyumlama işlevleri kapsamında açıklanmıştır.

3.2. İşlevsel Birimler

Bu kesim görevlerde kullanılabilen işlevleri içerdiği için işlevsel birimler olarak anılmıştır. Kullanıma sunulan bu işlevler gerçek zamanlı kapsamda görev yönetimini yönlendirmede kullanılırlar. Bu işlevler sundukları hizmetlerin amacına göre sınıflandırılmışlardır.

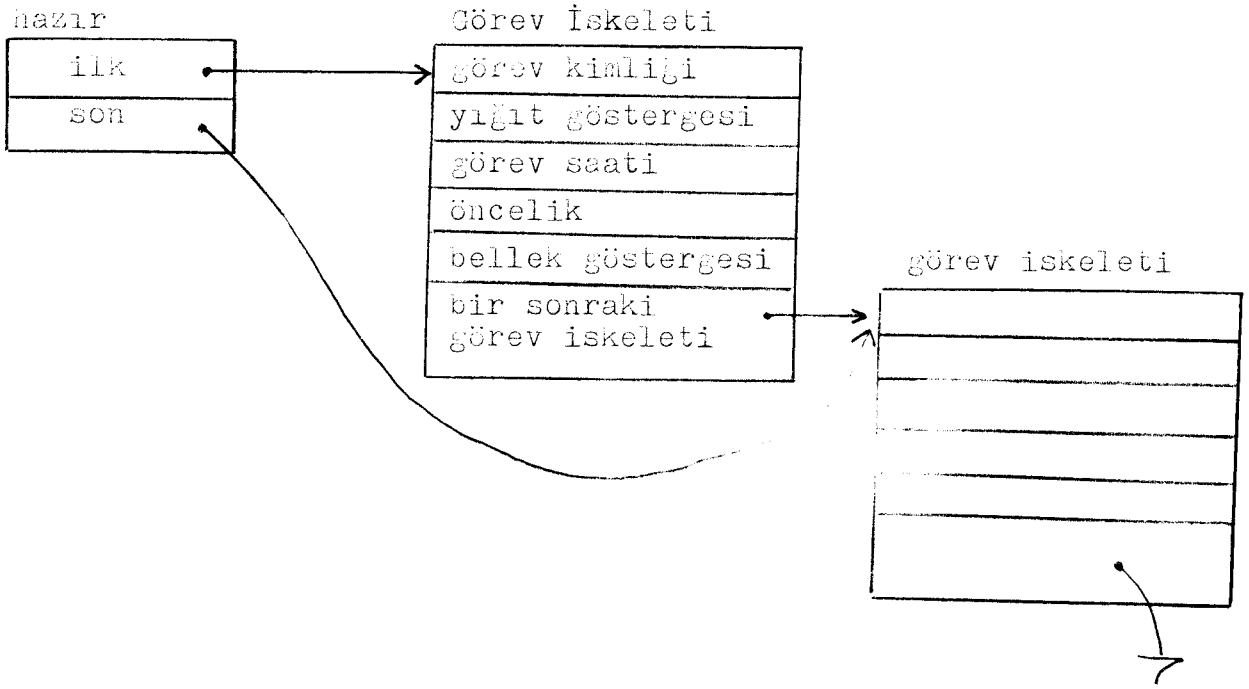
3.2.1. Görev Anahtarlama İşlevleri

Görev anahtarlama sadece dizgedeki üç durumu içerir. Bunlar Çizim 3.4'de de gösterilen hazır, çalışır ve bekler durumlardır. Dizgeye yeni tanımlanan ancak bellek atanması yapılmamış görevlerin bekletildiği 'tanımlı' durumu yapay bir durumdur.



Çizim 3.4. Görev Anahtarlama Kapsamındaki Görev Durumu Çizgesi

Çizim 3.4'de belirtilen görev durumları gerçekleştirimde bağlaçlı liste başlıklarına karşılık gelmektedir. Görevlere ilişkin bilgilerin tutulduğu görev iskeletleri bu listeye bağlanır. Çizim 3.5'de hazır konumundaki görevler liste örneğinde bu iki veri yapısı da gösterilmiştir.



Çizim 3.5. Hazır Durumuna Bağlı Görevlere İlişkin iki Görev İskeleti.

Görev anahtarlamada kullanılan işleçler:

a. "define_task" : görev tanımlama,
b. "init_task" : görevin kaynaklarını atama,
c. "run_task" : görevi çalıştırma,
d. "preempt_task": görevin çalışmasını kesme,
e. "wait_task" : görevi bekletme,
f. "terminate" : görevi sonlandırma,

a. "define_task" işlevi.

Bu işlev aracılığı ile C dilindeki bir altyordama görev kimliği kazandırılır. Görev tanımları C programlama dili aracılığı ile yapıldığı için görevlerin altyapısı bu dilin ürettiği koda uyumlu olacak biçimde hazırlanmıştır. Yordam adresi bir görev kimliği ile bu işleve argüman olur. İşlev uygulandığında, yordam, tanımlı kuyruğuna bağlı bir görev durumuna gelir.

Görev olarak tanımlanan yordamın diğer yordamlardan farklı geri dönüşlerdedir. Normal yordamda "return" komutu ile sağlanan bu işlevin yerini, görev tanımında "terminate" işlevi alır. Eğer bir görevin kullanımında yeniden girilir olmaması sorun yaratmayacaksa değişken tanımlarında kısıtlama yoktur. Eğer yeniden girilirlik sözkonusu ise tüm değişkenler "auto" olarak tanımlanmalıdır. Microsoft C derleyicisinde "auto" tanımlı değişkenler yığıt içinde tutulurlar. Böylece bir yordamın kod kesimi değişik yığıt alanları atanarak birden fazla görevin tanımında kullanılabilir.

b. "init_task" İşlevi.

Bu işlev aracılığı ile kimliği belirlenmiş bir görevin ("define_task") bellek atanması yapılır ve hazır görevler

kuyruğuna bağlanır. Bu aşamadan sonra görev, görev yönetici açısından tanımlanmış olur. Eğer başka hazır görev yoksa bir sonraki saat kesilmesinde bu görev işletime alınır.

"define_task" ve "init_task" işlevleri kullanılarak dizgeyi başlatmak mümkündür. Dizgeyi başlatabilmek için ana program içinde en az bir görev "define_task" ile tanımlanmalıdır. Daha sonra tanımlı herhangi bir görev "init_task" ile hazır konuma alınmalıdır. Bu işlevin sonrasında saat kesilmesini beklemek gerekmektedir. Bu yapıdaki bir dizge başlatma işlemi aşağıdaki örnekte verilmiştir:

Örnek:

```
main ()
{
multitask ();
define_task (1, gorev1);
init_task (1,100,15);
for (;;) { /* sonsuz döngü */};
}
```

Örnekte ilk çağırılan yordam çok görevli ortama ilişkin alt yapının kurulmasını sağlar. "multitask" yordamı çok görevli gerçek zamanlı yapının gereksediği veri yapılarının hazırlanması, saat ve kesilme donanımına ilişkin programlama kesimlerini içerir. Bu yordam dizgeye ilişkin işlevlerden önce bir kez çağırılmalıdır.

Örnekte "gorev1" adlı yordam 1 kimlik numarası ile dizgeye görev olarak tanımlanmıştır. "init_task" işlevi kullanılarak kimlik numarası ile belirlenen "gorev1" hazır konuma alınır. İşlevin ikinci argümanı görevin gereksediği bellek alanını sözcük cinsinden belirler. Üçüncü argüman ise görevin önceliğidir.

Anaprogram sonsuz döngü ile sonlanmıştır. Saat kesilmesi aracılığı ile anahtarlanan görev yönetici, "gorev1" i çalışır konuma alacaktır. Bir kez görev anahtarlama işlemi yapıldıktan sonra anaprograma kesinlikle dönülmeyecektir. Dizgenin güvenilirliğini sağlayabilmek için ana program içinde sadece bir görevin ilk değer ataması "init_task" işlevi ile yapılmalıdır. Diğer görevler tanımlanan bu görev üzerinden dizgeye tanımlanmalıdır.

Üç görevden oluşan bir dizgenin başlatılması aşağıdaki biçimde gerçekleştirilebilir:

```
main ()
{
 multitask ();
 define_task (1, ilk_gorev);
 define_task (2, gorev1);
 define_task (3, gorev2);
 define_task (4, gorev3);
 init_task   (1,50,0);
 for (;;);
}
 gorev1()
  {/*2.Görevin işlemleri */}
 gorev2()
  {/*3.Görevin işlemleri */}
 gorev3()
  {/*4.Görevin işlemleri */}
 ilk_gorev()
  {init_task (2, 100,5);
   init_task (3, 100, 1);
   init_task (4, 50, 10);
   terminate ();  /* Görevi sonlandır. */
  }
```

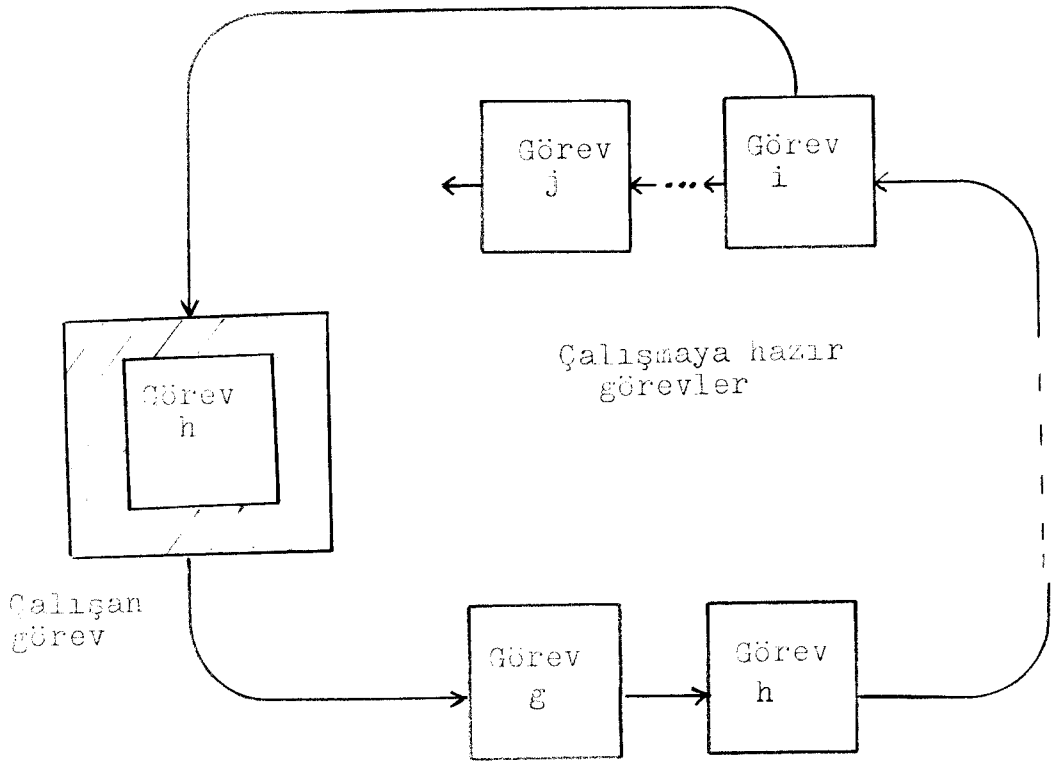
Bu tanımlanan yapıya göre önce "ilk_gorev" dizge içinde çalışır konuma gelecektir ve diğer görevleri görev yöneticiye tanımladıktan sonra kendisini sonlandıracaktır.

c. "run_task" İşlevi.

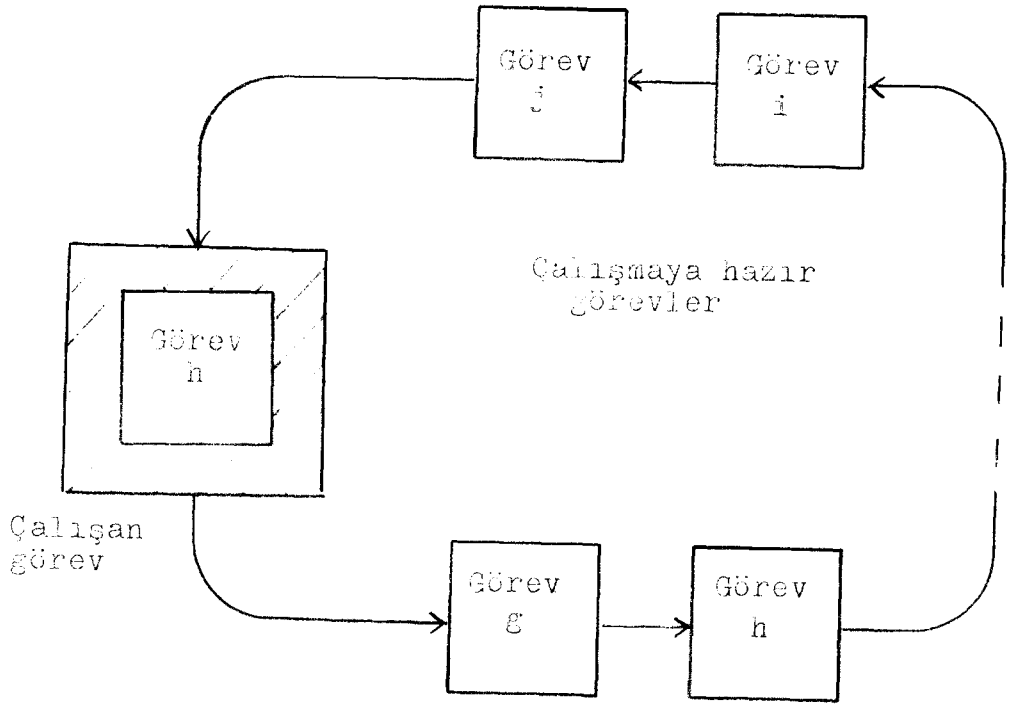
Hazır kuyruğundaki bir görevi çalışır konuma almak için "run_task" işlevi öngörülmüştür. Görev yöneticiyi yönlendirmek amacıyla kullanılan bu işlev bir görev içinden çağırıldıysa içinde bulunduğu görevin hazır kuyruğuna alınmasına neden olur. Dizgenin başlatılması aşamasında ana program içinde kullanıldığında çalışan bir görev söz konusu olmadığı için sadece belirlenen görevi çalıştırır, herhangi bir görevin hazır kuyruğuna alınmasına neden olmaz.

d. "preempt_task" İşlevi.

Bu işlev çağırıldığında içinde bulunduğu görevin hazır kuyruğunun sonuna eklenmesini ve yine hazır kuyruğunun en başındaki görevin çalışır duruma alınmasını sağlamak için öngörülmüştür. "run_task" işlevinden ayrılan yönü çalıştırılacak görevin seçimini görev yöneticiye bırakmasındadır. Böylece görev yöneticinin kullandığı "round-robin" yönetim biçim, devre dışı kalmamış olur. "run_task" ile "preempt_task" arasındaki farklılık Çizim 3.6'da gösterilmiştir.



Çizim 3.6.a. Görev h İçinden Görev i İçin "run_task" İşlevi Uygulanması Durumunda Anahtarlama İşlemi.



Çizim 3.6.b. Görev h İçinden "preempt_task" İşlevi Uygulanması Durumunda Anahtarlama İşlemi.

e. "wait_task" İşlevi.

Bir olayın ortaya çıkmasını bekleyen görevler için bekler ("Waiting") kuyruğu öngörülmüştür. Bu kuyruğa bir görevin katılması için görev içinden "wait_task" işlevinin uygulanması gerekmektedir. Bu işlevin uygulanması sonucunda çalışmakta olan görev bekler kuyruğuna bağlanır ve denetim görev yöneticiye aktarılır.

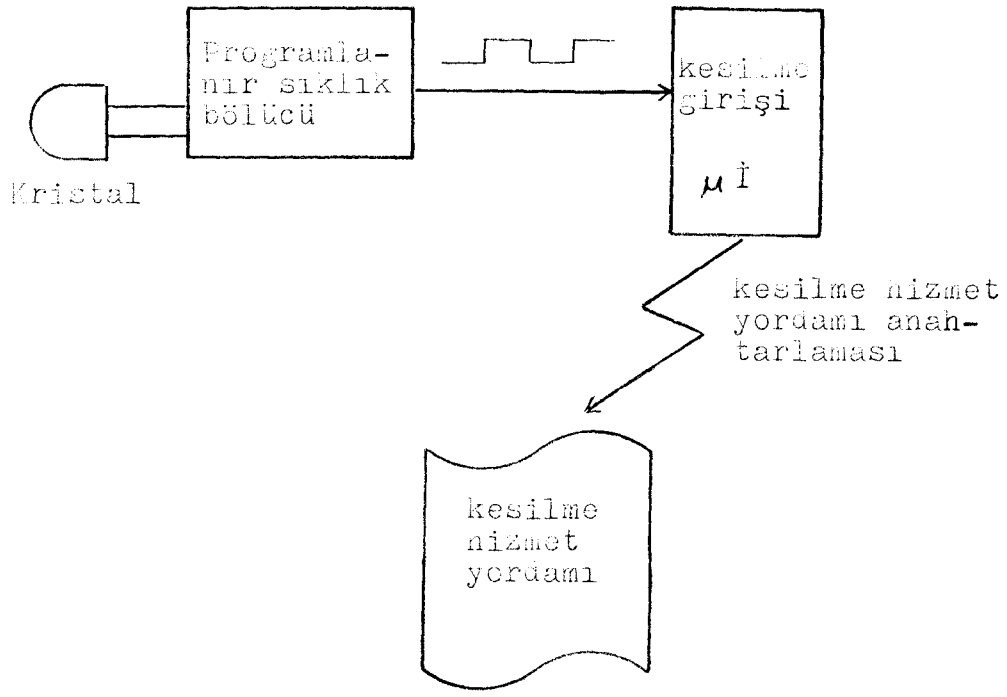
"wait_task" işlevi ile beklemeye alınan görevin hazır ya da çalışır durumuna alınabilmesi için bir başka görev ya da kesilme hizmet yordamı tarafından uyarılması gerekmektedir.

f. "signal" İşlevi.

"wait_task" işlevi aracılığı ile beklemeye alınan görevin hazır ya da çalışır durumuna geçmesini "signal" işlevi sağlar. Bir görevin bekler durumdan çıkarılması sonrasında hangi kuyruğa bağlanacağı görevin önceliği ile ilgilidir. Eğer o anda çalışan görevin önceliği uyarılan görevden büyük ya da eşit ise uyarılan görev hazır kuyruğuna bağlanır. Eğer uyarılan görevin önceliği daha büyükse çalışan görev hazır kuyruğuna bağlanıp, uyarılan görev çalışır durumuna alınır.

3.2.2. Gerçek Zaman Saati İşlevleri

Gerçekleştirilen dizgede kullanılan gerçek zaman saati kesilme üreten donanım ile bu kesilmeye hizmet veren yazılımdan oluşmuştur. Kesilme donanımı dizgedeki bir kristal aracılığı ile yazılımca belirlenen sıklıkta periyodik kesilme üretir. Bu kesilmelere hizmet veren yordam kesilme sıklığını saat oluşturmada kullanır. Çizim 3.7'de saat donanımı ve yazılımının ilke çizimi verilmiştir.



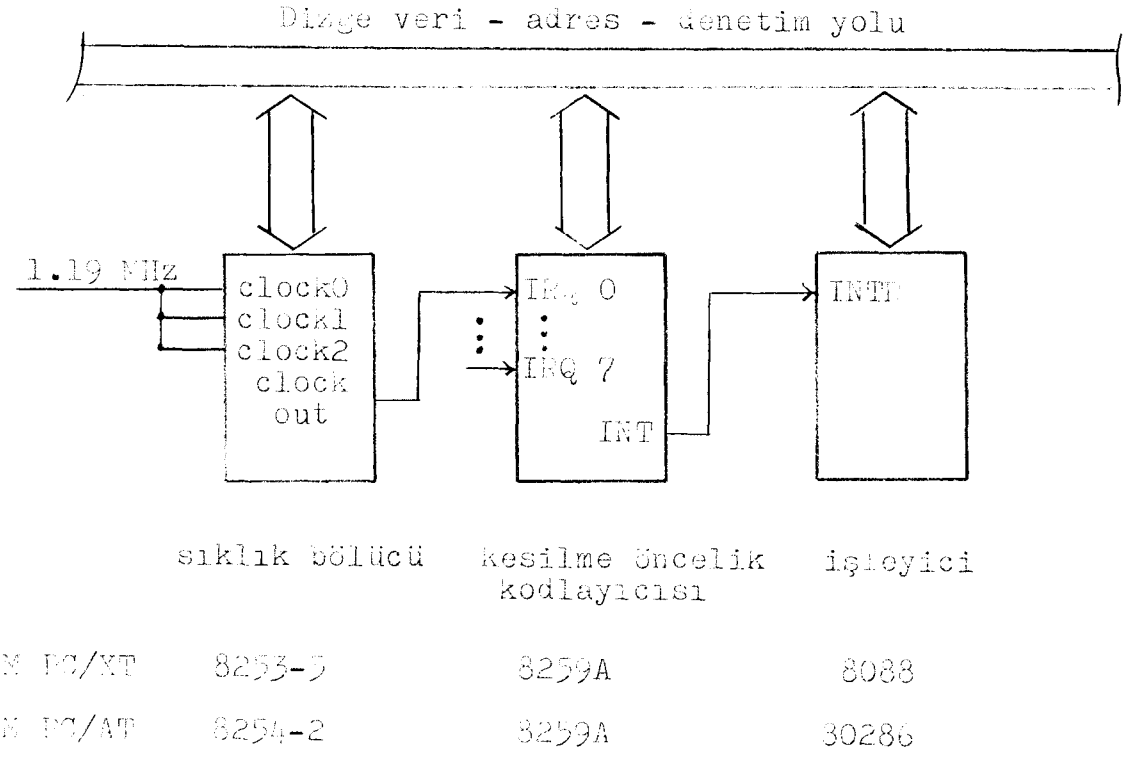
Çizim 3.7. Gerçek Zaman Saati İlke Çizimi.

Programlanır sıklık bölücü bir sayaçtan oluşmuştur. Kare im üretecinden gelen her vuru için sayaç bir eksiltilir. Sayaç içeriği sıfıra vardığında zaman, sıklık bölücü işleyiciye kesilme girişi üstünden bir uyarı iletir. Sayaç ilkdeğeri yeniden yüklenerek döngüsel biçimde işlem sürdürülür. Sıklık bölücüye yüklenen ilk sayaç değeri böylece sıklığı bölen katsayı niteliğini kazanır.

Gerçek zaman saati kesilme sıklığını hesaplamak için kullanılan eşittik

$$F = \frac{1.19}{n+1} \text{ MHz}$$

biçimindedir. Eşitlikte F kesilme sıklığı, n sayaç değeridir. IBM PC/XT-AT türü mikrobilgisayarlarda kullanılan saat kesilme donanımı Çizim 3.8'de verilmiştir.



Çizim 3.8. IBM PC/XT-AT Saat Kesilme Donanımı

IBM-PC türü mikrobilgisayarların XT ve AT modelleri arasındaki donanım farklılıklarına karşı yazılım açısından saat kesilmeleri tümü ile aynıdır. XT ve AT modellerindeki yazılımların taşınılabilirliğini bir ölçüde sağlayabilmek için saat kesilme donanımı adresleri ve kesilme vektörleri iki model içinde standart tutulmuştur. Bu açıdan çok görevli gerçek zamanlı dizge ile gerçekleştirilen uygulamalar zamanlama açısından hiç bir

değişikliğe uğramayacaktır. Ancak AT modelinin daha yüksek sıklıkta çalışması ve 80286'nın gerçek 16 ikil işleyici olmasından dolayı komut işletim hızı artacaktır.

İşleyicinin işlem hızının artması karşısında gerçek zaman saatinin değişikliğe uğramaması için XT ve AT modelleri arasında bazı farklılıklar çıkacaktır.

i. "round-robin" görev yönetim biçimi kapsamında, XT ve AT modelleri üzerinde çalıştırılan aynı uygulama için görevlere ayırılan zaman dilimlerinde AT modelinde daha fazla sayıda görevi ilişkin komut uygulanır.

ii. İlk farklılıktaki soruna bağlı olarak komut işletim hızına bağımlı uygulamalar taşınabilir olmayacaktır. Bu nedenle tüm zamanlama teknikleri çekirdek dizgenin sağladığı olanaklar kullanılarak gerçekleştirilmelidir.

Dizgede oluşturulan saat kesilmeleri iki amaçla kullanılır. Birici kullanım zaman aralığı ölçmeyi amaçlar. Bu amacı karşılamak için saate ilişkin kurma ve değer okuma işlevleri kullanıma sunulmuştur. Gerçek zaman saati kesilmesinin ikinci kullanım amacı görev anahtarlamadır. Görev anahtarlama sıklığında kısıtlama gerektiren görevler için saat destekli anahtarlama işlevi öngörülmüştür. Bu işlevler gerçekleştirimde kullanılan adlarıyla;

a. "set_timer",
b. "get_timer",
c. "sleep" işlevleridir.

a. "set_timer" İşlevi.

İki olayın oraya çıkması arasında geçen zamanı ölçmek için görevler gerçek zaman saatini kullanırlar bu zaman dilimini ölçmek için görevler tarafından "set_timer" işlevi ile zaman sayma işlemi başlatılmış olur.

Bu işlevi gerçekleştirmek için dizgedeki bir saat değişkenini sıfırlamak ve bu değer üzerine zamanı saymak düşünülebilir. Ancak çok görevli bir ortam sözkonusu olduğu için birden fazla görev bu saatin kullanımına gereksinim duyacaktır.

Çok görevli yapıya hizmet vermek için birden fazla saat tutmak soruna çözüm olabilir. Saat kullanımı gerekseyen bir görev saati dizgeye tanımlar. Bu yaklaşımda her saat kesilmesinde bir çok saat değişkeninin artırılması gerekmektedir. Saat değerlerinin artırılması dizgeye ek bir yük getirip etkinliği düşürecektir.

Sonuçta bir tek saat değişkeni üzerinde çok görevli yapıya hizmet verebilmek için döngüsel bir saat yapısı öngörülmüştür. Dizgedeki bir saat değişkeni kesilme hizmet yordamı tarafından artırılmaktadır. Değişken en büyük değerini aldıktan sonra sıfırlanıp döngüsel olarak sayım işlemine devam etmektedir. Saate gereksinim duyan görev "set_timer" işlevi ile saat değişkeni içeriğini görevin yerel saatine kopyalamaktadır. Bu yapı ile dizgede birden çok saat öngörüldüğü gibi, aynı görev içinde de birden fazla saat kullanılmasına olanak tanımlanmıştır.

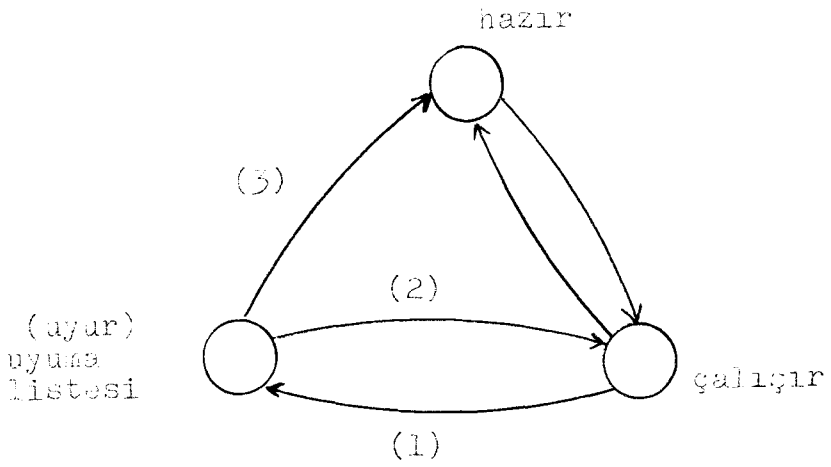
b. "get_timer" işlevi

"set_timer" işlevi ile yerel bir saat değişkenine alınan başlangıç değerlerinden bu yana geçen süreyi hesaplamak için "get_timer" işlevi öngörülmüştür. Şimdiki değerden başlangıç değerini çıkararak zaman dilimi hesaplanır. Gerçekleştirimde 32

ikil uzunluğunda saat değeri kullanılmıştır. İşaretsiz olarak kullanılan saat değişkeni ile en çok 2**32-1 saat birimi sayılabilir. İki saat kesilmesi arasındaki geçen süre yazılımsal olarak, gereksinime göre ayarlanabildiği için bu süre saat birimi diye adlandırılmıştır, saniyenin kesirleri ile ifade edilmemiştir. Ayrıca saat kesilmesinin üretilmesinde dizgenin kristalinden yararlanıldığı için, kristalin değeri de süre hesaplanmasında bir parametredir.

c. "sleep" İşlevi.

"sleep" işlevi, görev yöneticinin desteği ile gerçek zaman saatine uyumlu görev anahtarlamayı gerçekleştirir. Bu işlev aracılığı ile uyutulan görev belirlenen saat vuru sayısı boyunca kuyrukta bekletilir. Belirlenen sayıda saat vurusunun oluşması durumunda görev bu kuyruktan alınır ve önceliğine göre hazır ya da çalışır konuma geçer. Eğer uyandırılan görevin önceliği o anda çalışan görevden büyükse görev çalışır konuma değilse hazır konuma geçer. Uyuyan görevlerin kuyruklandığı konum bekler konuma eşdeğer yapıdaki uyuma listesidir. Çizim 3.9'da uyuma listesinin, görev durumları çizgesindeki ilişkileri göstermiştir.



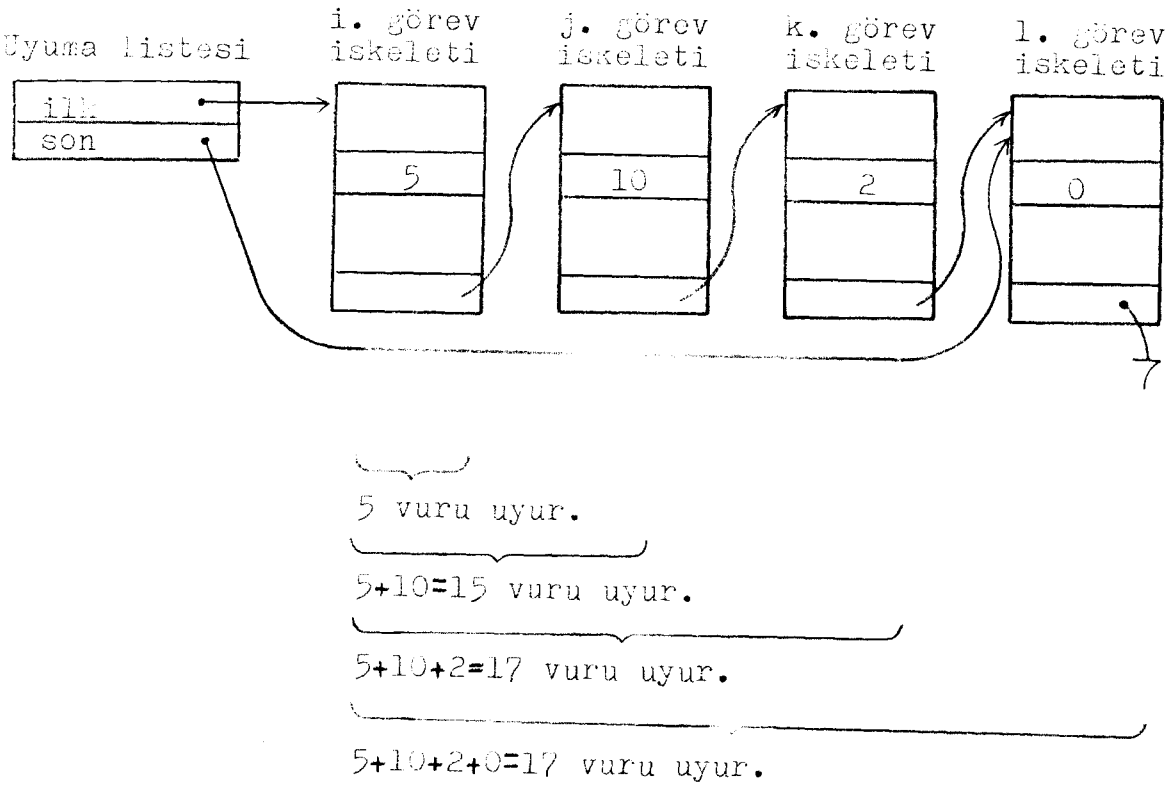
Çizim 3.9. Uyuma Listesinin Görev Durumları Çizgesindeki Yeri

Çalışır durumdaki bir görev "sleep" işlevini vuru sayısını vererek çağırır. Görev çalışır konumdan uyur konuma geçer (1). Her saat kesilmesi geldiğinde görevin vuru sayısı bir eksiltilir. Bu vuru sayısını sıfıra düşmesi durumunda görev hazır konuma (3) ya da çalışır konuma (2) geçer.

Uyuma listesindeki görevlerin iskeletlerindeki görev saati vuru sayılarının tutulması için kullanılır.

Çok görevli ortamda listede birden fazla görevin bulunması kaçınılmazdır. Her saat kesilmesi geldiğinde liste baştan sona taranarak görev saatleri bir eksiltilip sıfır değerine düşen görevler listeden çıkarılabilirler. Bu türde bir yaklaşımda dizgenin yükü sık gelen saat kesilmeleri karşısında artmaya başlar. Saat kesilmelerindeki bu işlem yükünü azaltabilmek amacıyla listedeki görev saatleri liste içinde bir öncekine göreli olarak

tutulmaktadır. Böylece sadece listenin en başındaki görevin saati eksiltilerek listedeki diğer görevler içinde eksiltilme sağlanmaktadır. Çizim 3.10 'da uyuyan dört görevin listedeki örnek durumları gösterilmiştir.



Çizim 3.10. Uyuma Konumundaki Görevlerin bir Örneği

Görevlerin uyuma listesindeki göreli yerleşimleri saat kesilmesi hizmet yordamının yükünü azaltırken ekleme algoritmasının karmaşıklığı artmaktadır. Ekleme algoritmasının kullanım sıklığı kesilme hizmet yordamının kullanım sıklığından çok daha az olması dizgenin toplam etkinliği diğer yaklaşımlara göre yüksektir. Uyuma listesine yeni eklenen bir görev uyuma süresine göre listede uygun yere yerleştirilir.

Uyuma listesindeki görevlerin uyandırılması sorumluluğu kesilme aracılığı ile sürülen görev yöneticinindir. Uyuyan görevlerin uyandırılması sırasında ortaya çıkan bazı sorunlar vardır. Aynı zamanda uyanması gereken görevlerden hangileri hazır konuma, hangileri çalışır konuma geçer? Çalışır konumda ancak ve ancak bir görev bulunabildiği için aynı anda uyanan görevlerden en yüksek öncelikli olanı çalışır konum için aday olur diğerleri hazır konuma geçer. Çalışır konuma aday olan görev o anda çalışan yoksa ya da çalışan görevden daha yüksek öncelikli ise çalışır konuma diğer durumda ise hazır konuma geçer.

3.2.3. Zaman Uyumlama İşlevleri

Görevler arasındaki zaman uyumlamayı ya da zaman uyumlu veri iletişimini sağlamak amacı ile zaman uyumlama işlevleri öngörülmüştür. Zaman uyumlama işlevleri kullanıcılarca tanımlanmış görevlere hizmet verdiği kadar çekirdek dizge tarafından da kullanılmaktadır.

Çok görevli ortamda görevler birbirlerinden bağımsız çalışırlarken ortak kaynaklara erişime gereksinim duyarlar. Ancak iki görev ortak kaynağa eriştiğinde kaynağın paylaşımı sorunlar yaratır. Bu sorunlar kaynağın paylaşılabilir nitelikte olmamasından kaynaklanır. Bu tür kaynaklara kritik kaynak adı verilir. Kritik kaynağın kullanımında görevlerin kaynağa sıra ile erişmelerini sağlamak amacı ile karşılıklı dışlama düzeneğinin kurulması gerekir. Görev içinde ortak kaynağa erişimi sağlayan kesim, kritik kesim olarak tanımlanır. Karışılıklı dışlama ise, sözü edilen kritik kaynağın, belirli bir anda salt bir görev tarafından kullanılmasını sağlamaktır. Kritik kesimine giren bir görev bu kesimden çıktıktan sonra diğer görevler kritik kesimlerine girebilirler.

Gerçekleştirilen çok görevli gerçek zamanlı dizgede değişik durumlara karşı kritik kaynakların korunması için çözümler getirilmiştir. Kritik kaynakların kullanımına getirilen koruma önlemleri; (a) iki kullanıcı tanımlı görev arasında, (b) iki çekirdek dizge işlevi arasında, (c) çekirdek dizge işlevi ile kesilme hizmet yordamı arasındadır.

a) İki kullanıcı tanımlı görev arasında koruma

Kullanıcı tanımlı görevler arasında oluşan kritik kaynakların kullanımında zaman uyumlamayı sağlamak için semafor ile karşılıklı dışlama düzeneği öngörülmüştür. Yalın bir örnek kapsamında değişik görevlerce ortak kullanılan bir değişken kritik kaynak olarak düşünülebilir.

```
Sayaç = sayaç + 1
```

Örnek işlemde "sayac" adı verilen değişkenin, herhangi bir görev içinde değerinin artırılmasını sağlayan program kesimi kritik kesimi oluşturur. Ortak kullanılan bu değişkenin değerini artırmak isteyen bir görev önce değişken değerini bir yerel değişkene (yazmaca) alıp değeri 1 artırdıktan sonra değişken alanına geri yazacaktır. Önlem alınmayan durumlarda sayacın 2 artması beklenirken 1 artması gibi durumlarla karşılaşılabilecektir. Bu kritik kesimi s adı verilen ikili semaforu kullanarak korumak olanaklıdır.

```
lock (s);
sayac=sayac+1;
unlock(s);
```

"lock" işlevi ile ikili semafor sıfırlanır ve diğer görev bu kesime girmeye çalıştığı anda "lock" işlevi içinde bekletilir. "unlock" işlevi ile semaforun değeri 1'e çıkınca "lock" işlevinde bekletilen görev kritik kesime girer.

b) İki çekirdek dizge işlevi arasında koruma

Çekirdek dizge işlevleri arasında zaman uyumlama işlevlerine gereksinim yoktur. Bir çekirdek dizge işlevi bir başkasını kullanmadığı için kritik kesim oluşmaz. Çekirdek dizge içinde kullanılan işlevler kritik kesim oluşturmayacak biçimde geliştirilmişlerdir.

c) Çekirdek dizge işlevi ile kesilme hizmet yordamı arasında koruma

Kesilme hizmet yordamı içinde çekirdek dizgeye ilişkin kesimler kullanılabilir. Kesilmelerin herhangi bir anda ortaya çıkması belirli sorunlara neden olur. Bu sorunlardan en önemlisi çekirdek dizge işlevi içinde bulunulduğu sırada gelen kesilmenin kritik kesim yaratmasıdır. Çekirdek dizge işlevi içinde dizgenin ortak veri alanlarına erişirken kesilme gelip bu veri alanlarına erişmeyi istemesi veri bütünlüğü ya da tutarlılığının bozulmasına neden olur. Bunu engellemek için çekirdek dizge işlevleri içinde kesilmeler kapalı tutulur. Örneğin "preempt_task" işlevi içinde çalışır ve hazır konumlarına ilişkin kuyruk yapılarının bağ alanları düzenlenirken, gelen kesilme ile bu bağları değiştiren başka işlev anahtarlanırsa bağlarda kopmalar olur.

```
preempt_task ()
{ _SKELETON *p;
   disable ();
  _running.first->_stack_ptr =context(0);
  _get_first (&_running, &p);
  _put_last (&_ready,p);
  enable ();
  for (;;);
}
```

"preempt_task" işlevinde kullanılan "disable" ve "enable" işlevleri, kesilmeler ile kritik kesimin bölünmesini engeller.

Kullanıcı görevleri için öngörülmüş çekirdek dizge işlevleri ikili semaforları (binary semaphores), sayan semaforları (counting semaphores) ve posta kutularını (mailboxes) işlemek için öngörülmüştür.

Zaman uyumlama ile ilgili çekirdek dizge işlevleri:

a. "lock" işlevi,
b. "unlock" işlevi,
c. "init_semaph" işlevi,
d. "P" işlevi,
e. "V" işlevi,
f. init_queue" işlevi,
g. "send" işlevi,
h. "receive" işlevi,

a. "lock" işlevi.

İkili semaforlar aracılığı ile kritik kesimlere girişlerde zaman uyumlama, "lock" işlevi tarafından sağlanır. Bu işlevin çalışma ilkesi, bir değişkenin sınanıp, eğer değeri sıfırsa görevi bekletmek, değeri sıfırdan farklı ise değişkeni sıfırlayıp görevi işletimine devam ettirmektedir. Bu düzeneğin gerçekleştiriminde 8086 simgesel dilinde öngörülmüş bazı özel komutlar kullanılmıştır.

```
        .
        .
        xor  ax, ax          ; ax yazmacını sıfırla
lock    xchg ax, [si]        ; ax ile [si] nin gösterdiği
                             ; bellek alanı içeriğini
                             ; değiştir.
        or ax, ax            ; ax sıfır mı ?
        jz lock
```

"xchg" komutu ile "test-and-set" yapısı gerçekleştirilmiştir. Çok işleyicili düzene yönelik olarak "xchg" komutu "lock" öneki ile başka işleyicilerin DMA'yı (Doğrudan bellek erişimi) kullanıp komutu bölmesi engellenmiştir.

İkili semaforlar C programlama dilindeki tamsayı değişken olarak tanımlanmıştır. Yukarıda verilen semafor gerçekleştirme örneğinde olduğu biçimde zaman dilimi boyunca döngüde bekleme gibi etkinliği azaltıcı seçenek çekirdek dizgede engellenmiştir. "lock" işlevinin gerçekleştirimde kullanılan biçimde görevin hemen hazır konuma alınması sağlanmıştır. Eğer "lock" işlevi örnekte belirtildiği gibi gerçekleştirimiş olsaydı, semafor 1 olmadığı sürece görevin döngüde beklenmesi gerekecek ve döngüde bekleme göreve ayırılan zaman dilimi boyunca sürecektir.

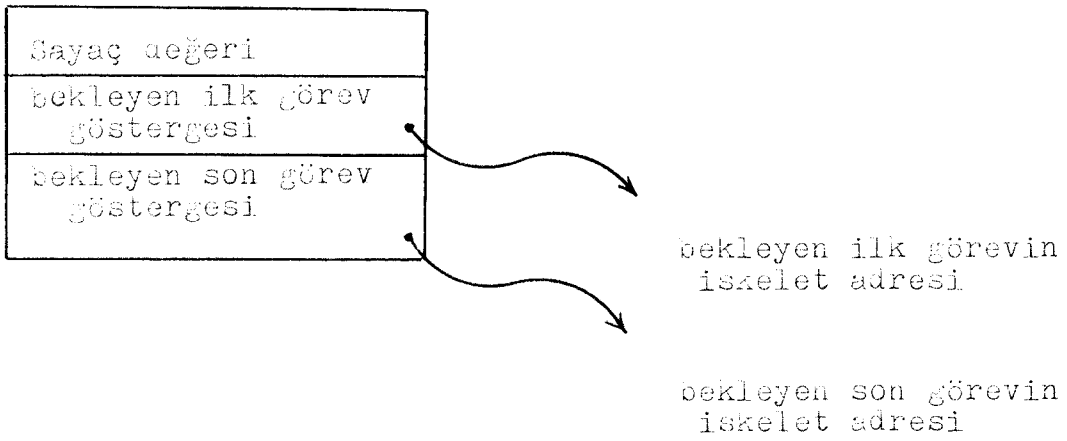
```
loop:  xor  ax, ax
  lock xchg ax, [si]
       or ax, ax
       jnz continue
       call _preempt_task      ; zaman dilimi bitimini
       jmp loop                ; beklemeden görevi hazır
continue:  .                   ; konuma al.
           .
           .
```

b. "unlock" işlevi.

Kritik kesimden çıkarken ikili semaforun bire kurulmasını sağlayıp diğer görevlerin kritik kesimlerine girmesi sağlanır. Semaforu bekleyen görevler hazır konumunda bulundukları için görev yönetici tarafından çalışır konuma atanacaklardır. Bu bakımdan işlev sadece semaforun 1' e kurulmasını içermektedir.

c. "init_semaph" işlevi.

Sayaç semaforlar kullanılmadan önce ilk değer atamaları yapılmalıdır. Bunu sağlamak için "init_semaph" işlevi öngörülmüştür. Bu işlev semaforun sayaç alanına belirlenen değeri ve semaforun bekleme kuyruğu başlığına boş adres aktarır. Çizim 3.11'de semaforun veri yapısı gösterilmiştir.



Çizim 3.11. Semaforun Veri Yapısı

d. "P" İşlevi

"P" işlevi sayan semafor üzerinde işlem yapmak üzere öngörülmüştür. İşlevin tam olarak ne yaptığı algoritmik bir anlatım ile:

```
P İşlevi
  eğer sayaç sıfırdan farklı
      ise sayacı 1 eksilt
      değilse - görevi semaforun bekleme kuyruğuna al.
              - hazır konumdaki görevlerden ilkini çalışır
                konuma al.
son
```

e. "V" işlevi.

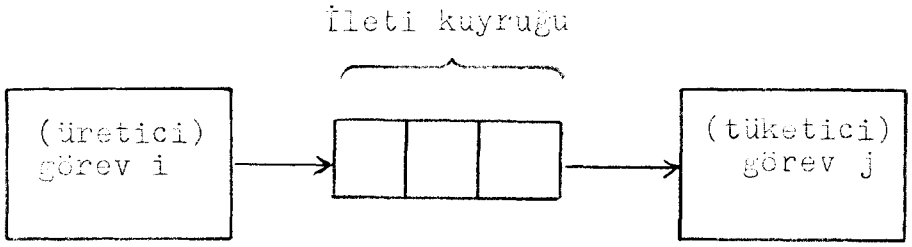
"V" işlevi de "P" gibi sayan semaforlar üzerinde işlem yapmak için öngörülmüştür. İşlevin yaptığı iş algoritmik bir anlatım ile:

```
V işlevi
  eğer semaforun bekleme kuyruğu boş
        ise sayacı 1 artır.
        değilse semafor kuyruğundaki ilk görev o anda
                çalışan görevden yüksek öncelikli
                    ise çalışır konuma al.
                    değilse hazır konuma al.
son
```

Kullanılan iki işlev "P" ve "V"'de, işlevin bölünmezliği girişte kesilmelerin kapatılması çıkışta kesilmelerin açılması ile sağlanmıştır.

f. "init_queue" İşlevi.

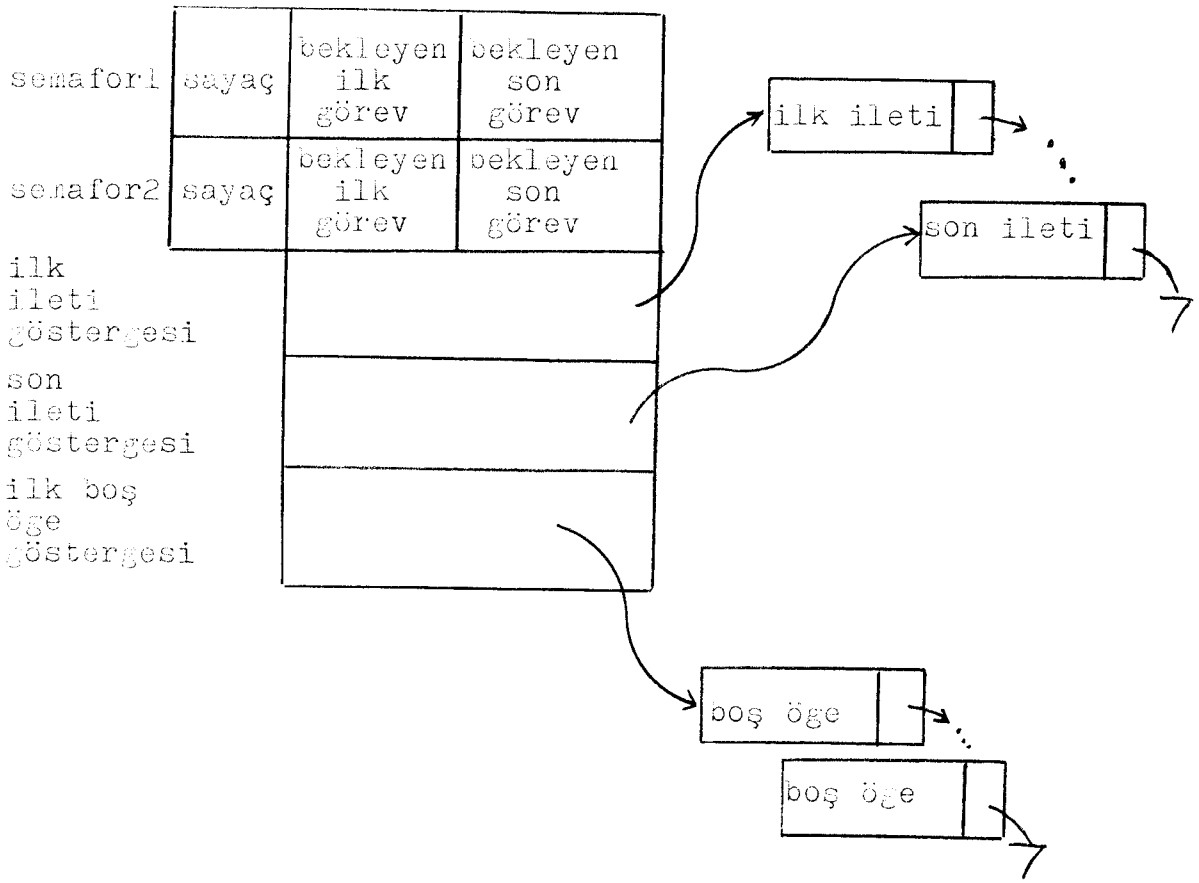
Semaforlar kullanılarak daha üst düzeyli zaman uyumlama düzenekleri öngörülmüştür. İleti kuyrukları ya da posta kutusu (mailbox) olarak adlandırılan üst düzeyli zaman uyumlama düzeneği bunlardan biridir. İleti kuyrukları aslında belleği olan semaforlardır. Belirli sayıda ögeden oluşan bu kuyruk yapısına üretici ve tüketici görevler semaforlar aracılığı ile zaman uyumlu biçimde erişirler. Bir üreticisi bir de tüketicisi bulunan 3 ögeli bir kuyruk Çizim 3.12'de örneklenmiştir. "init_queue" işlevi ile semaforlara ilk değerler atanır ve bir kuyruk yapısı oluşturulur.



Çizim 3.12. Üretici ve Tüketici ile İleti Kuyruğu.

İleti kuyruğu çeşitli veri yapıları ile gerçekleştirilebilir. Bunlar döngülü ya da bağlaçlı kuyruk yapısı olabilir. Gerçekleştirimde bağlaçlı kuyruk yapısı seçilmiştir. Çünkü dizge içinde birçok ileti kuyruğu devingen biçimde tanımlanabildiği için bellek yönünden kaynakların yönetimi sözkonusu olmaktadır. Kullanımına gerek kalmayan ileti kuyruklarının bellek alanları dizgeye "init_queue" ile geri verilmektedir.

"init_queue" işlevinin yeni bir kuyruk mu tanımlayacağı yoksa tanımlı bir kuyruğun kaynaklarını dizgeye geri mi vereceği parametreler ile belirlenir. Eğer işlev çağırıldığında kuyruğun öge sayısını belirleyen parametre sıfırsa bu kaynakların geri verileceği anlamına gelir. İleti kuyruğu veri yapısı Çizim 3.13'de gösterildiği gibi bir başlık ve ona bağlı kuyruk öğelerinden oluşur.



Çizim 3.13. İleti Kuyruğunun İlke Çizimi.

İleti kuyruğunun başlık ögesi beş ana parçadan oluşur. İlk iki parçası kuyruğa erişimde zaman uyumlamada kullanılan (kuyruğun boş ve dolu olması durumlarını sınayan) S1 ve S2 semaforlarıdır. Sonraki iki başlık ögesi gönderilmiş ancak henüz tüketici tarafından alınmamış kuyruk ögelerinin ilkini ve sonuncusunu gösterir. Başlığın son göstergesi kuyruktaki boş ögelerin ilkini

gösterir. Diğer kuyruk ögeleri tek bağlaçlı yapıda ilk ögeye bağlanmışlardır. İletiler değişmez uzunluktaki kuyruk öğeleri üzerinden iletilir. Eğer iletinin boyu kuyruk ögesinin boyundan kısaysa ileti ASCII sıfır damgası ile sonlandırılır. Kuyruk ögesinin boyu uygulamaya özel biçimde saptanabilir niteliktedir.

S1 semaforu ileti kuyruğundaki boş öge sayısının S2 semaforu gönderilmiş ancak alınmamış ileti sayısını belirler. "init_queue" işlevi algoritması ilkesel olarak belitilirse,

- S1 <-- kuyruktaki öge sayısı
- S2 <-- 0
- ilk göst. <-- boş gösterge
- son göst. <-- boş gösterge
- boş öge göstergesine boş ögelerle bağlaçlı kuyruk oluştur.

biçimindedir.

g. "send" İşlevi

"send" işlevinin çalışma ilkesi algoritmik bir anlatımla;

P(S1)
lock (semafor)
- boş kuyruk ögesi al.
- iletiyi bu ögeye aktar.
- son gelen ileti olarak kuyruğun sonuna bağla

unlock (semafor)
V (S2)

Algoritması verilen işlevde S1 semaforu başlangıçta 1 eksiltilir. Bunun anlamı boş kuyruk ögesi sayısının 1 eksiltilmesi biçimindedir. Daha sonra kuyruğa iletinin bağlanması sözkonusudur. Bu işlem kiritik kesim oluşturduğu için "lock" ve

"unlock" işlevleri ile karşılıklı dışlama düzeneği kurulmuştur. İletinin kuyruğa bağlanmasından sonra "V" işlevi ile S2 semaforu 1 artırılarak tüketiciye alınabilecek bir iletinin daha geldiği belirtilmektedir.

h. "receive" işlevi

İleti kuyruğunda bulunan bir iletiyi almak için tüketici görev tarafından "receive" işlevi kullanılır. "receive" işleminin çalışma ilkesi, algoritmasal biçimde aşağıda tanımlanmıştır.

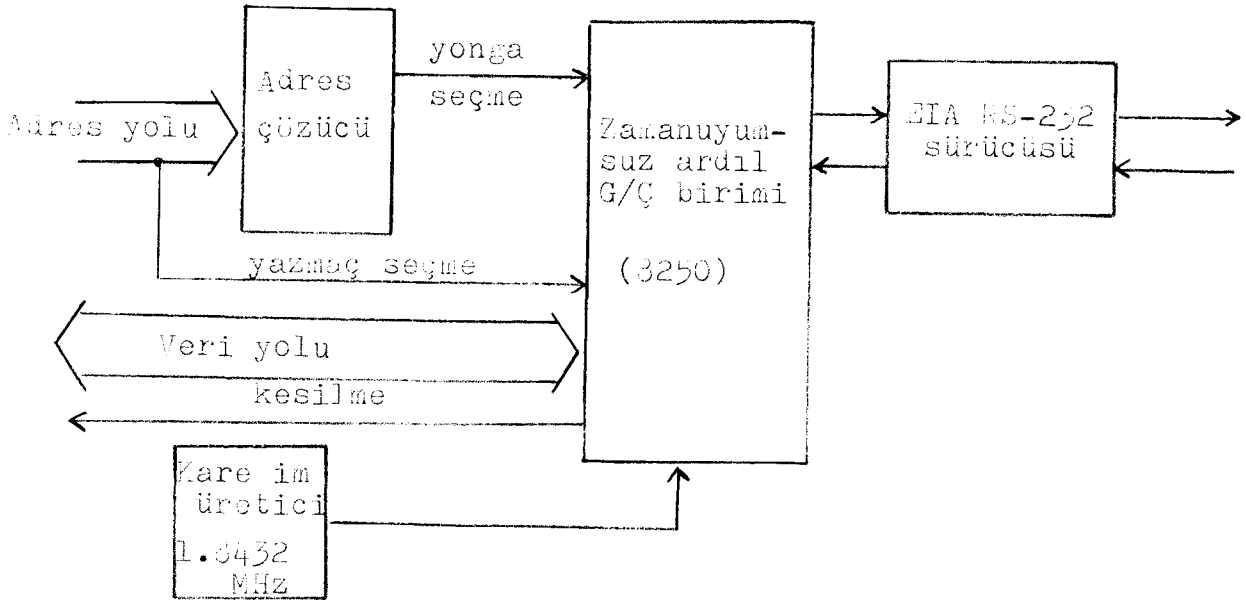
```
P (S2)
lock (semafor)
- kuyruktaki ilk ögeyi al
- kuyruk ögesi içindeki iletiyi aktar
- kuyruk ögesini boş kuyruğuna koy
unlock(semafor)
V(S1)
```

İlk önce S2 semaforu 1 eksiltilerek iletinin alındığı belirtilir. Kuyrukla ilgili işlemler için "send" işlevinde olduğu gibi karşılıklı dışlama düzeneği kullanılmıştır. "V" işlevi ile S1 1 artırılarak kuyrukta bir ögenin daha serbest kaldığı belirlenmiştir.

3.2.4. Giriş/Çıkış İşlevleri

Gerçek zamanlı çok görevli çekirdek dizge kapsamında olmamasına rağmen, ardıl giriş/çıkış birimi gerçekleştirime katılmıştır. Bu biçimde bir tasarım kararının alınmasının ana nedeni, ardıl iletişim biriminin hemen hemen her uygulamada yaygın biçimde kullanılmasıdır.

Kullanıcı görevlere sunulan hizmetler ardıl giriş/çıkış biriminden giriş, çıkış ve durum sorma gibi istemlerdir. Ardıl giriş/çıkış birimine ilişkin donanımın ilke gösterimi Çizim 3.14'de verilmiştir.



Çizim 3.14. Ardıl giriş/çıkış birimi ilke çizimi

Ardıl giriş/çıkış (8250 yongası) birimi tümü ile programlanabilir niteliktedir. Programlama işlemini gerçekleştirmek için bu birimde bir dizi yazmaç öngörülmüştür. Bu yazmaçların kullanım amaçları Çizelge 1'de verilmiştir. IBM PC serisi bilgisayarlarda dizgede en fazla iki ardıl iletişim birimi kullanılabilmektedir. Çizelgede belirtilen adreslerdeki X damgası yerine 2 ya da 3

değerlerini alabilmektedir.

Çizelge 3.1. Ardıl İletişim Birimi Yazmaçları (8250'nin yazmaçları)

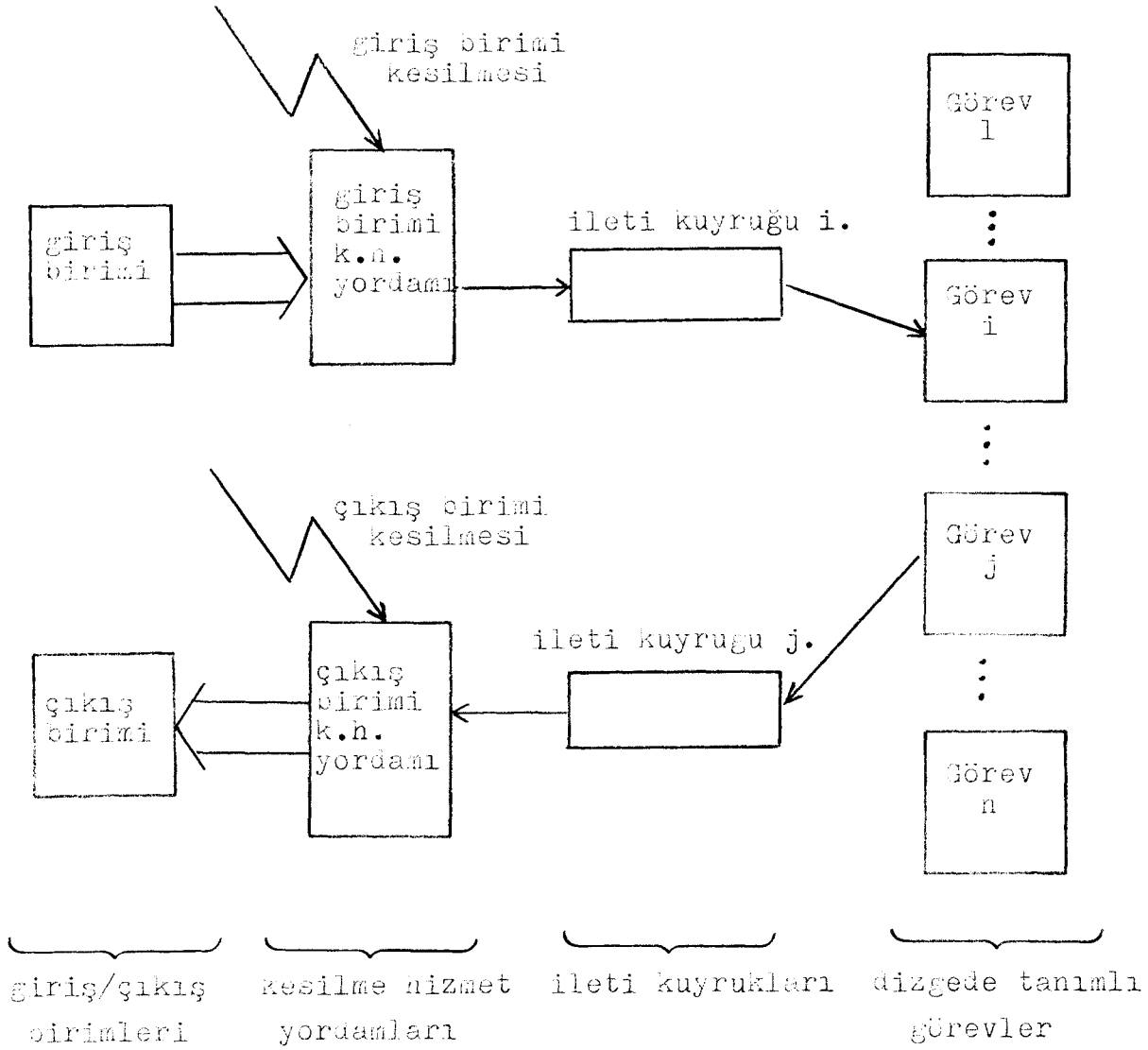
g/ç adresleri	seçilen yazmaç
XF8	TX yastığı (çıkış yastığı)
XF8	RX yastığı (giriş yastığı)
XF8	Bölücü yazmaç (küçük ağırlıklı sekizli)
XF9	Bölücü yazmaç (büyük ağırlıklı sekizli)
XF9	Kesilme maske yazmacı
XFA	Kesilme kimlik yazmacı
XFB	Yol denetim yazmacı
XFC	Modem denetim yazmacı
XFD	Yol durum yazmacı
XFE	Modem durum yazmacı
XFF	Özel yazmaç

Giriş ve çıkışların programlanmasında kesilmeli ve sıralı iletişim yöntemlerinin kullanılabilmesi için gerekli altyapı öngörülmüştür. Kesilmeli giriş/çıkış yönteminde kesilme yönetim işlevlerinden yararlanılmaktadır. Yanıt alma zamanı kritik olan giriş/çıkış işlemleri için kesilmeli yöntem en kullanışlı olanıdır. Sıralı iletişim yöntemi ise yanıt zamanı kritik olmayan giriş/çıkış işlemlerinde kolaylıkla kullanılabilme özelliğine sahiptir.

a. Kesilmeli giriş/çıkış işlemleri.

Kesilmeli giriş/çıkış işlemleri için kesilme hizmet yordamı tanımlama olanağı öngörülmüştür. Kesilme hizmet yordamının içeriği, geliştirilen dizgenin özelliklerine bağımlı olacağından yalnızca gerekli altyapı sağlanmış, donuk bir kesilme giriş/çıkış işlevi öngörülmemiştir. Kesilme hizmet yordamı dizgede bulunan diğer görevlerden bağımsız olduğu için, görevler ile kesilme hizmet yordamı arasındaki veri iletişiminde ileti kuyruklarının kullanımı önerilen bir yaklaşımdır.

Bir giriş birimi tarafından üretilen kesilme uyarısı giriş birimi kesilme hizmet yordamının anahtarlanmasına neden olur. Hizmet yordamı gelen veriyi alır ve bunu ilgili ileti kuyruğuna bağlar. Bu veriye gereksinim duyan görev veriyi ileti kuyruğundan alır. Kesilme hizmet yordamı ile görev arasındaki üretici tüketici ilişkileri, Çizim 3.15' de, kavramsal düzeyde hem giriş hem de çıkış birimi için verilmiştir.

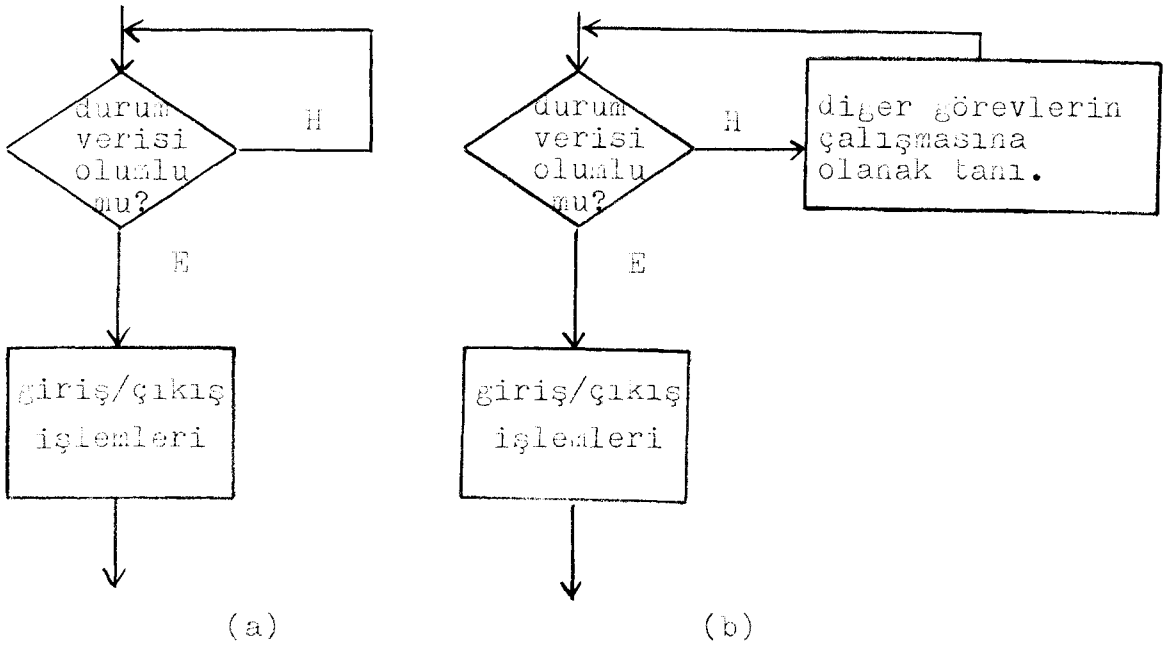


Çizim 3.15. Kesilme Hizmet Yordamı ve Görevler Arasındaki Veri İletişimi.

b. Sıralı giriş/çıkış işlemleri

Sıralı giriş/çıkış, ilke olarak bir durum bilgisinin (genelde bir ikil) sınanması, eğer durum olumsuz ise döngüde beklenmesi,

olumlu olduğu anda ise giriş/çıkış işleminin yapılmasını içerir. Ancak döngüde kalınan süre işlem biriminin verimsiz kullanımına neden olur. Ayrıca gerçek zamanlı çok görevli yapıya ters düşen sıralı işlem diğer görevleri de etkilemiş olur. Sıralı işlemin ilkesini bozmadan çok görevli ortama uyarlamak olanaklıdır. Bu amaçla bekleme yerine belirli bir süre diğer görevlerin çalıştırılması, daha sonra durum bilgisinin sınanması sağlanmıştır. Böylece durum bilgisini sürekli sınama yerine belirli aralıklarla sınama gerçekleştirilmiştir.

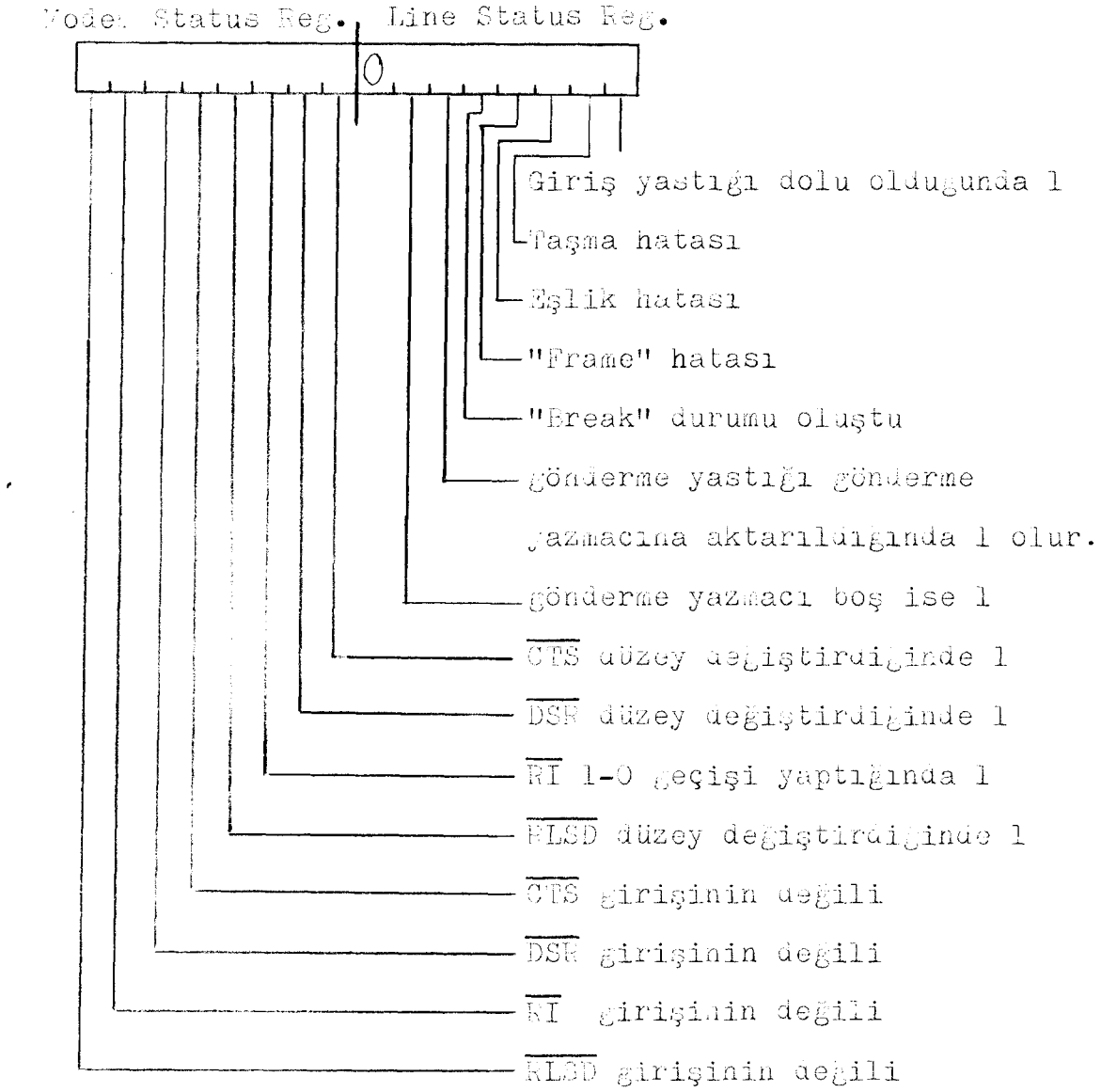


Çizim 3.16. Sıralı Giriş/Çıkış Yapısı (a), Sıralı Giriş/Çıkış Yapısının Çok Görevli Ortama Uyarlanmış Durumu(b)

Ardıl giriş/çıkış birimi için üç adet işlev öngörülmüştür.

a. "modem_status" işlevi

Bu işlev adresi verilen ardıl giriş/çıkış arabiriminin durum sözcüğünü okumada kullanılır. Giriş/çıkış işlemleri sırasında oluşan durumlar Çizim 3.17'de verilmiştir.



Çizim 3.17. Ardıl Giriş/Çıkış Birimi (RS232 C) için Durum Sözcüğü.

b. "serial_in" İşlevi.

Bu işlev adresi verilen ardıl giriş/çıkış biriminden bir sekizlinin okunmasını sağlar. Giriş işlemi sıralı iletişim yöntemi ile yapılmaktadır. Durum sözcüğündeki ("Data Ready") veri hazır ikil sınanıp, eğer bu ikil sıfırsa görev hazır kuyruğuna alınarak diğer görevlerin çalışmasına olanak tanınır. Çizim 3.18'de bu işlevin algoritması C progrmalama dili ile verilmiştir.

```
char serial_in (port)  /* Bir sekizli döndüren yordam */
int port;
{
  while ((modem_status(port) & 0001) == 0)
       { /* Veri hazır ikili sıfır olduğu sürece */
            preempt_task();
            /* görevi hazır kuyruğuna at. */
       };
     /* Veri hazır ikili 1 oldu. */
       return (inp(port));
}
```

Çizim 3.18. "serial_in" İşlevinin C Programlama Dili ile Gerçekleştirimi.

Çizim 3.18'de verilen yaklaşımda veriyi bekleyen görevin hazır konuma alınması yöntemi görev sayısı arttıkça ya da görev çalışma zamanı dilimleri arttıkça etkinliğini yitirebilir. Eğer kullanıcı kesilme yapısını da kullanamıyorsa önerilen yaklaşım diğer işlevleri kullanarak kendi algoritmasını yaptığı işin isterlerine göre oluşturmasıdır. Eğer ardıl birimden verilerin geliş sıklığı biliniyorsa Çizim 3.19'daki algoritma ile bu sıklığı uygun sınama aralıkları belirlenebilir.

```
while ((modem_status(port) & 0x0001) == 0)
  { /* Veri hazır ikili sıfır olduğu sürece */
    sleep (ZAMAN);
   /* göreli ZAMAN değişmezi süresince uyut */
  };
  return (inp(port));
```

Çizim 3.19. Gerçek Zaman Saati Uyumlu Giriş Sınama Yönetimi.

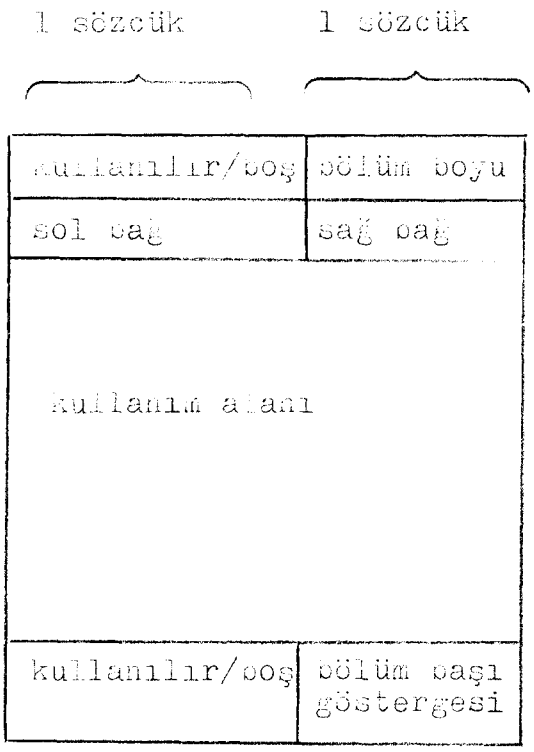
Önerilen yöntemin işlerlik kazanabilmesi için bu kesimi kullanan görevin yüksek öncelikli olması gerekmektedir. Çünkü görev uyandırıldığı anda çalışan görev daha yüksek öncelikli ise uyanan görev hazır konuma alınacaktır. "round_robin" görev yönetim biçiminden dolayı sıra kendisine gelene kadar belirli bir zaman yitimi olacaktır. Eğer uyanan görev çalışandan daha yüksek öncelikli ise işlem birimini doğrudan ele geçirip sınama işlemini yapacaktır.

c. "serial_out" İşlevi.

Ardıl giriş/çıkış biriminden yapılan çıkış işlemleri için "serial_out" işlevi kullanılır. "serial_in" işlevi gibi bu işlev için de aynı bekleme yapısı kurulmuştur. Bir önceki gönderilen verinin alınıp alınmadığı sınanarak işlem yapılmaktadır. "Serial_in" işlevindeki sorunlar bu işlev için de geçerli olduğundan kullanıcı kendi gereksinimlerine göre bekleme süresini kısıtlayan yöntemi kullanabildiği gibi kesilme altyapısını da kullanabilmektedir.

3.2.5. Bellek Yönetim İşlevleri.

Görevlerin değişkenleri ve yığıtının gereksediği bellek alanları bellek yönetim işlevleri ile sağlanır. Ayrıca görevler tarafından bellek istemi yapılarak özel amaçlı bellek kullanımı da olanaklıdır. Çekirdek dizgenin gerçekleştiriminde devingen bölümü bellek yönetimi kullanılmıştır. Çizim 3.20'de bellek yönetiminde kullanılan herhangi bir bölümün veri yapısı verilmiştir.



Çizim 3.20. Bellek Bölümü Veri Yapısı.

Bellek yönetimi için kullanılan alan, başlangıçta tek bir bölümden oluşmaktadır. Bellek istemleri oldukça bu bölüm parçalanarak daha küçük bölümlere ayırılır. Kullanımı bitip geri verilen bölümler komşu bölümler kullanılmıyorsa, onlarla birleştirilerek daha büyük serbest bölümler oluşturulur. Bölümlemede, bellekte kullanılamayan kırpık alanların kalmaması için bölümleri sayfa boyunun katlarına eşitleme yöntemi kullanılmıştır. Böylece bölümleme sonucunda artan bellek alanları sayfa boyundan daha küçük olamaz. Uygun sayfa boyu seçimi ile kalan bellek alanlarının kullanılabilir tutulması sağlanmıştır.

Bu tez kapsamında gerçekleştirilen çekirdek dizgeyi kullanarak uygulama programları yazacak kişinin C programlama dilindeki işlevlere ilişkin veri alanlarının yapıları konusunda bilgisi olmalıdır. Görevlerin gereksediği bellek sığasının hesaplanması için değişkenlerin bellekteki konumlarının uzunlukları bilinmelidir.

Çizelge 3.2. Microsoft C 3.00 Derleyicisi için Değişkenlerin Bellekte Kapladıkları Alan Uzunlukları

Tanım	Bellekte kapladığı alan
char	1 sekizli
int	2 "
short	2 "
long	4 "
unsigned short	2 "
unsigned long	4 "
float	4 "
double	8 "

Microsoft C derleyicisinde "auto" olarak tanımlanan ya da belirtilmediği için "auto" kabul edilen değişkenler yığıt içinde tutulur. Bu nedenle bir C altyordamı çağırıldığında önce geri dönüş bilgileri saklanır daha sonra altyordam içinde tanımlı "auto" değişkenlerin toplam boyu kadar yığıt göstergesi ilerletilir. Yordama girişte yığıt içinde açılan bu alan değişken değerlerinin tutulması için kullanılır. Bu değişkenler yordamın yerel değişkenleridir ve yordamın bir sonraki kullanımında bir önceki değerlerini saklamazlar.

Görevler için bellek isteminde bulunurken görevin değişkenleri ve görevin çağırdığı yordamların değişkenleri göz önüne alınmalıdır. Bu amaçla çağırılan yordamların toplam bellek gereksinimi yerine iç içe çağırılabilecek yordamların toplam gereksinimi hesaba katılmalıdır. Örneğin görevin kendisi 100 sözcük, çağırdığı üç yordam 1000'er sözcük bellek gerekseyen bir uygulamada Çizim 3.21'deki gibi yordamlar ard arda çağırılırsa bellek gereksinimi 1000+100=1100 sözcük olacaktır.

```
gorev()
{ int  dizi [100];
  yordam1();
  yordam2();
  yordam3();
}
yordam1()
  { int   dizi1[1000];
  }
yordam 2()
 { int    dizi2[1000];
    .
    .
    .
 }
yordam3()
 { int    dizi3[1000];
    .
    .
    .
 }
```

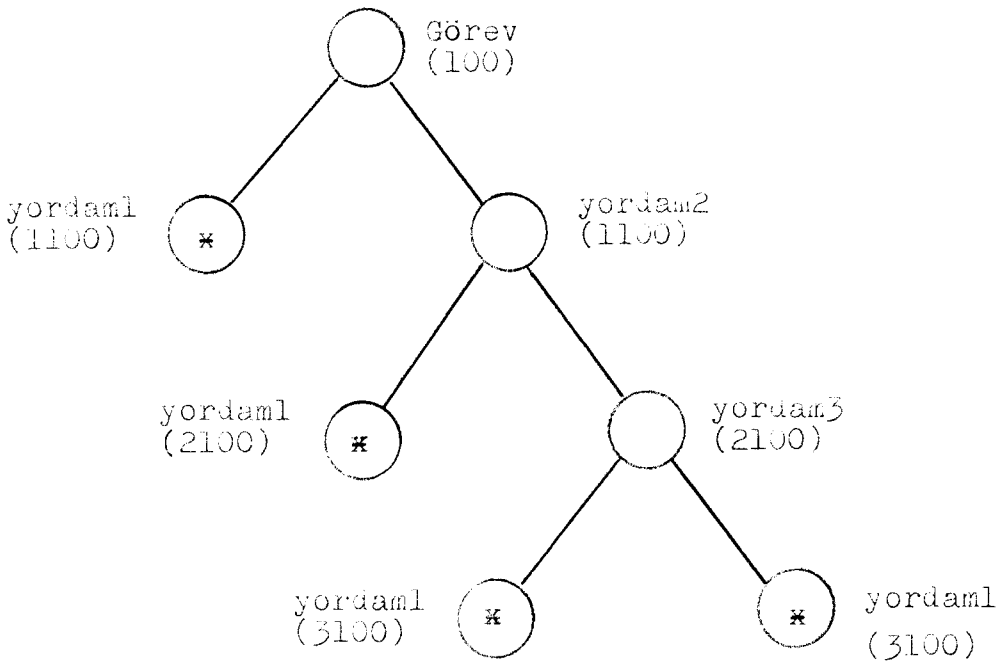
Çizim 3.21. Ardarda Yordam Çağırma Örneği

İç içe yordam çağırmada ise bellek gereksinimi olası tüm iç içe çağırmalardaki bellek gereksinimlerinin en büyüğüdür. Çizim 3.21 için hazırlanan görev yordamlar Çizim 3.22'deki gibi kullanıldığında bellek gereksinimi daha farklı olacaktır.

```
gorev()
 { int   dizi[100];
   yordam1();
   yordam2();
 }
yordam1()
 { int   dizi[1000];
   .
   .
   .
 }
yordam2()
 { int   dizi2[1000]
   yordam1();
   yordam3();
 }
yordam3()
 { int   dizi3[1000]
   yordam1();
   yordam1();
 }
```

Çizim 3.22. İçiçe Yordam Çağırma Örneği.

Çizim 3.22'de verilen bellek gereksinimi, yordam çağırmalarda sıradüzeni gösteren bir ağaç yapısı ile kolayca hesaplanabilir:



Çizim 3.23. Çizim 3.22'de Verilen Örneğin Yordam Çağırmada Sıradüzen Ağacı

Yordamlar arasındaki sıradüzeni görteren ağaçta uç düğümlerdeki en yüksek bellek gereksinimi görevin bellek gereksinimi olarak saptanır. Çizim 3.23'deki örnekte bellek isteminde gereksinim 3100 sözcükten aşağı olmamalıdır.

Gereksenen bellek sığası saptandıktan sonra iki işlev aracılığı ile bellek ataması ve serbest bırakılması gerçekleştirilir. Bu işlevler "get_mem" ve "back_mem" dir.

a. "get_mem" işlevi.

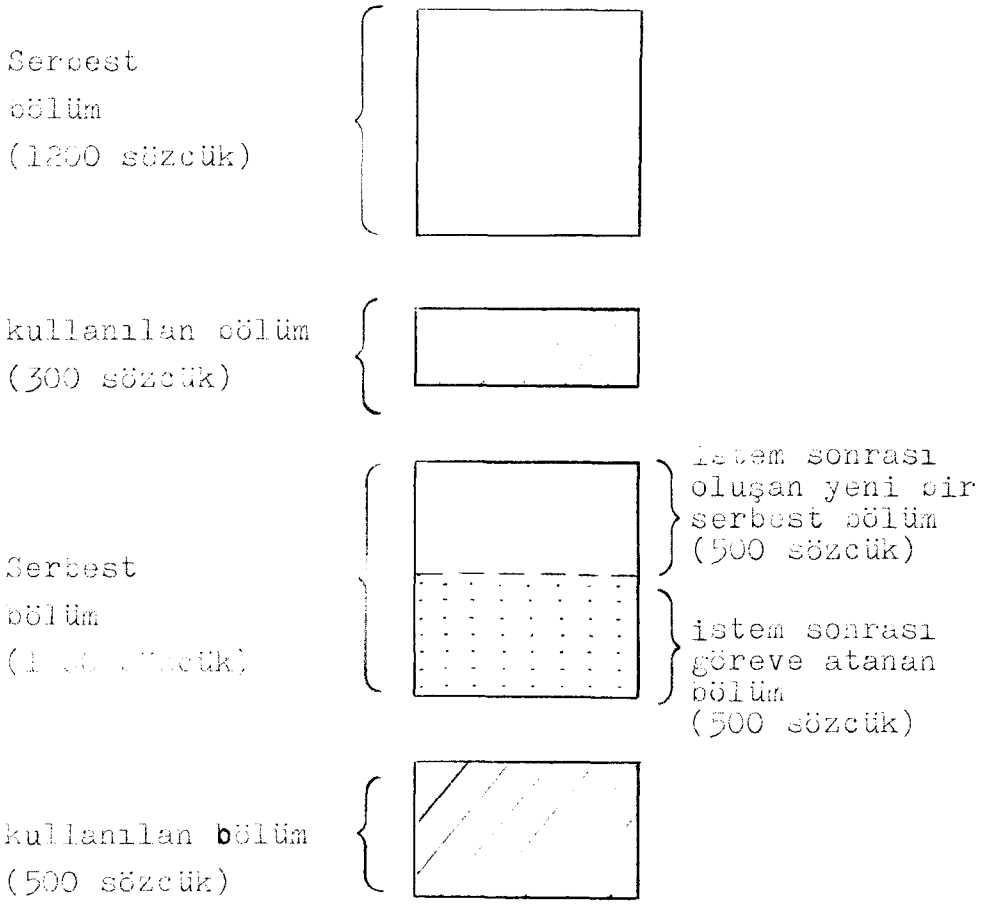
Bu işlev aracılığı ile serbest bellek listesinden bellek alınarak bellek ataması yapılır. Bellek atamada kullanılan yöntem ilk uygun ("firstfit") serbest bellek alanını seçer.

Bu işleve argüman olarak gereksenen belleğin uzunluğu sözcük türünden verilir. İşlev, atama yapıldı ise atanan belleğe gösterge, atama yapılamadı ise olumsuz durum verisi döndürür. Örneğin 1000 sözcük uzunluğunda bellek istemini karşılamak için;

```
if (_get_mem (1000,&p))
     {/*istem karşılandı */)
else
     {/*istem karşılanamadı*/}
```

program kesimine benzer bir yaklaşım kullanılır. Eğer istem karşılandı ise işlevin değeri sıfırdan farklı, karşılanamadı ise sıfır olarak dönecektir. Bu durum verisini kullanarak "p" göstergesi içindeki değerin anlamı olup olmadığına karar verilir.

Bellek yönetiminde devingen bölümlü bellek yönetimi kullanıldığı için kullanılan ve serbest bölümlerin tutulduğu bir liste yapısı sözkonusudur. Bellek atama istemi olduğu anda bu listedeki serbest bölümler arasında gereksinimi sağlayan ilk bölüm seçilir. Eğer seçilen bölümden artan alan sayfa boyundan büyükse, seçilmiş olan bu bölüm iki ayrı bölüm biçimine getirilir. Bunlardan biri serbest diğeri kullanılan bölüm olarak listede yerlerini alırlar. Çizim 3.24'de 500 sözcük uzunluğundaki bellek isteminin karşılanmasına ilişkin bir örnek verilmiştir.

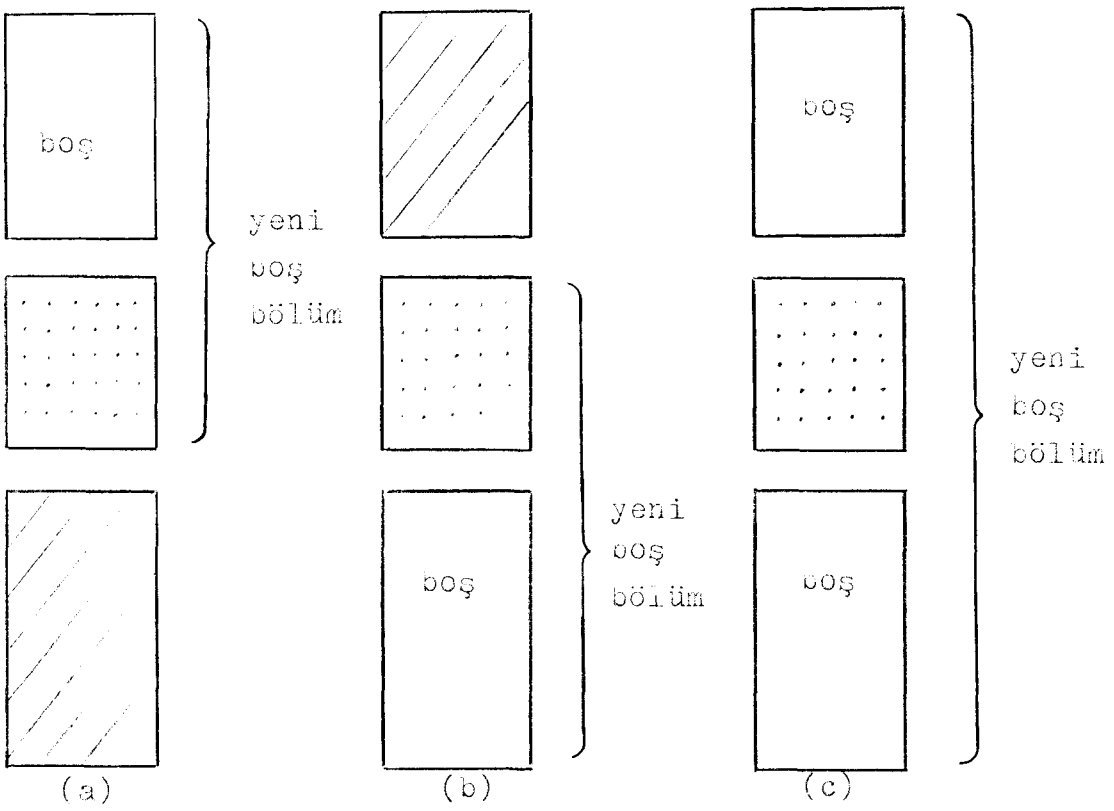


Çizim 3.24. Bellek Atama Örneği. İçi Boş Bölümler Serbest, Taralı Olanlar Kullanılan, Noktalı Olanlar İse Yeni Kullanılacak Bölümü Belirlemektedir.

b. "back_mem" İşlevi.

"get_mem" işlevi aracılığı ile atanmış olan bellek alanının ser-

best bırakılması için "back_mem" işlevi öngörülmüştür. Bellek atama aşamasında bölümlerin daha küçük bölümlere ayırılması büyük bölümleri ortadan kaldıracak ve bellekte yeteri kadar yer olması durumunda bile büyük bellek istemlerini tek bir bölüm biçiminde karşılayamayacaktır. Bu nedenle bellek geri bırakma işlemi sırasında olanaklı ise bölümler birleştirilerek daha büyük bölümler oluşturulur. Bellek bırakma aşamasında üç tür bölüm birleştirme durumu vardır ve bu durumlar Çizim 3.25'de gösterilmiştir.



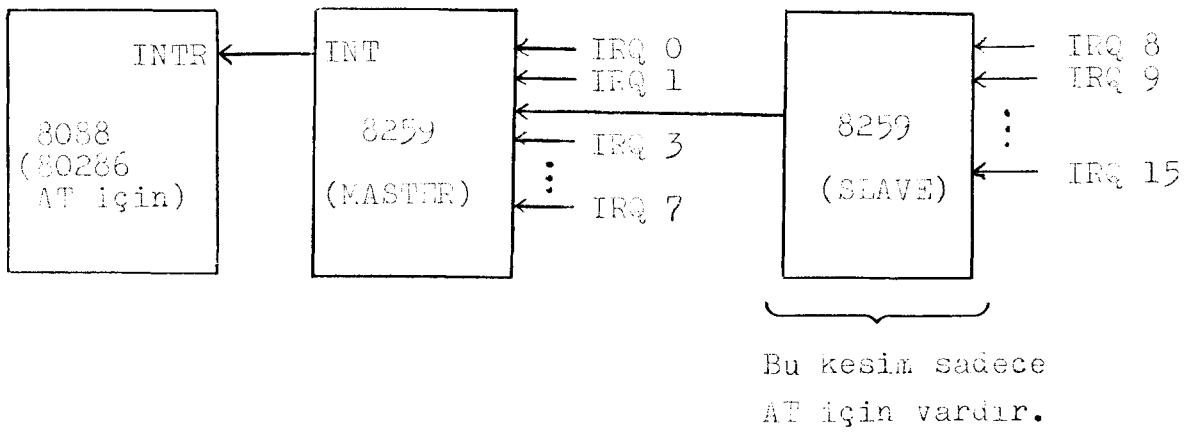
Çizim 3.25. Serbest bırakılan bellek bölümünün (noktalı) komşu bölümlerle ilişkileri: (a) üst bölümle birleştirme, (b) alt bölümle birleştirme, (c) hem alt hemde üst bölümle birleştirme.

Çizim 3.25'de belirlenen üç koşul dışında serbest bırakılan bölümün her iki komşusuda kullanılıyor ise bölüm birleştirme işlemine sokulmadan serbest bırakılır.

3.2.6. Kesilme Yönetim İşlevleri.

Çekirdek dizge ile geliştirilecek bir kısım uygulamalar donanıma yakın düzeyde olacaktır. Donanıma ilişkin aygıtların programlaması C programlama dilindeki giriş/çıkış kapılarına erişimi sağlayan işlevler aracılığı ile gerçekleştirilir. Uygulamalarda kesilme altyapısını da kullanmak için bir gereksinim duyulacaktır. Bu amaçla çekirdek dizgeyi kullanacak olana kesilme hizmet yordamı tanımlayabilme, dizgeyi kesilmelere açıp kapatabilme olanağı sağlanmıştır.

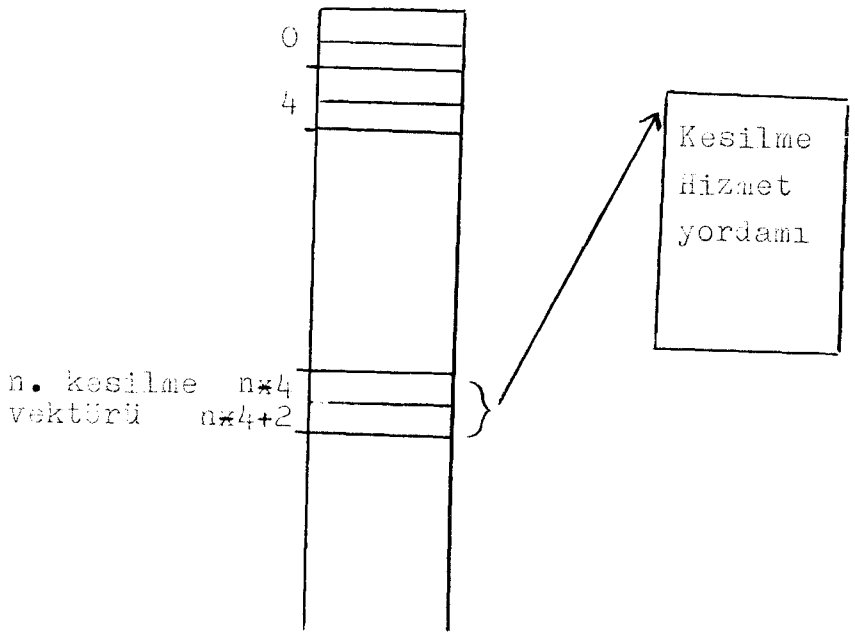
Çekirdek dizgenin IBM-PC serisi mikrobilgisayarlar için öngörülmesi nedeni ile kesilme dizgesine ilişkin yazılım, bu donanımdan kaynaklanan özellikleri de içermektedir. Bu özelliklerin en önemlisi kesilmelerin bir öncelik kodlayıcısından sonra işleyiciye yansımasıdır. IBM-PC XT-AT mikrobilgisayarları için kesilme donanımı Çizim 3.26'da verilmiştir.



Çizim 3.26. IBM PC XT-AT Kesilme Donanımı

Kesilmeler, işleyici düzeyinde açılıp kapatılabildiği gibi kesilme öncelik kodlayıcısı düzeyinde de denetlenebilirler. Bu nedenle her kesilme sonunda kesilme öncelik kodlayıcısını bir sonraki kesilme için açmak gerekir.

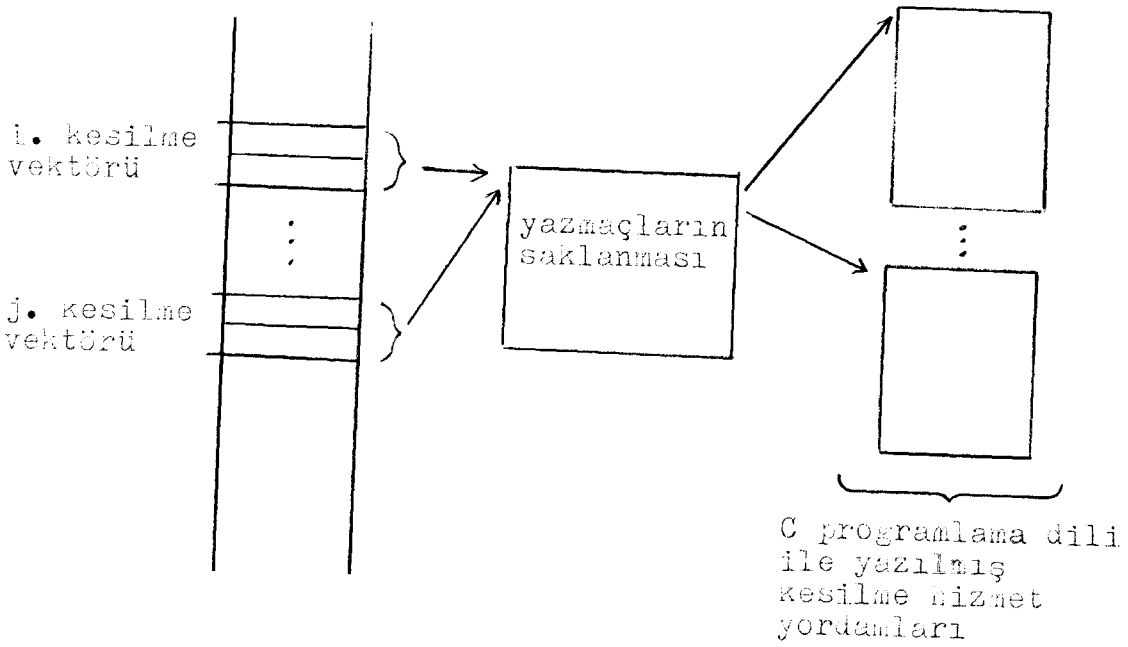
8086 serisi mikroişleyicilerde kesilme hizmet yordamları bir kesilme vektörü aracılığı ile anahtarlanırlar. Belleğin 0 ile 1023 adresleri arasında kalan ve herbiri 4 sekizliden oluşan 256 adet vektör ögesi vardır. Her vektör ögesi bir kesim adresi ve bir de kesim içi adres içerir. Bu iki adres kesilme hizmet yordamını belirler. Çizim 3.27'de kesilme vektörünün kullanımına ilişkin bir örnek verilmiştir.



Çizim 3.27. Kesilme vektörü ve Kesilme Hizmet Yordamı İlişkisi

Bir kesilme herhangi bir anda geldiği için işleyicinin hangi aşamada olduğu bilinemez. Bu nedenle kesilme hizmet yordamı işleyicinin o andaki yazmaç değerlerini bozmamakla yükümlüdür. Bunu gerçekleştirmek için ise kesilme hizmet yordamı girişte yazmaçları saklar dönüşte ise bu yazmaç değerlerini eski değerlerle günleyerek kesilme geldiği andaki işin, kaldığı yerden devam etmesini sağlar. Ancak çekirdek dizgeyi kullanan yazılımlarda C programlama dilini kullandığı için, yazmaçların saklanıp geri günlenmesi karmaşık olacaktır. Bu karmaşıklığı ortadan kaldırmak ve kullanıcıya yazmaçların korunma yükümlülüğünü yansıtmamak için bir yazılım katmanı gerçekleştirilmiştir. Bu yazılım katmanı kesilme hizmet yordamına sapmadan önce yazmaçları saklar ve kesilme öncelik kodlayıcısını kesilmelere açar, dönüşte ise yazmaçları günleyerek kesilen programa kaldığı yerden devam eder. Çizim 3.28'de bu yazılım katmanına ilişkin öbek çizim

verilmiştir.



Çizim 3.28. Çekirdek Dizgede Kesilme Vektörünün Kullanımı.

Geliştirilen kesilme altyapısını kullanan üç adet işlev vardır.

a. "assign_interrupt" işlevi,

C programlama dili ile yazılmış bir altyordamın kesilme hizmet yordamı olarak tanımlanmasını sağlar. Kesilme hizmet yordamının içinde kesilme altyapısına ilişkin özelliklerin kullanılmasına gerek yoktur, sadece kesilmeye hizmet veren kesimi içerir.

Çekirdek dizgede aynı anda en fazla 5 kesilme hizmet yordamının tanımlanabilmesine olanak sağlanmıştır. Örneğin 12 numaralı kesilme vektörü ardıl giriş/çıkış birimine adandığında kesilme giriş yordamı Çizim 3.29'daki gibi olacaktır.

```
      .
      .
   assign_interrupt (1,12,kesilme);
      .
      .
kesilme()
 { int j;
   i= inp(0x3F8); /* gelen damgayı oku */
      .
      .
 }
```

Çizim 3.29. Kesilme Hizmet Yordamı Örneği

"assign_interrupt" işlevinin argümanlarının ilki çekirdek dizge içindeki kesilme numarasını, ikincisi 8086'nın kesilme numarasını belirler. Çekirdek dizge içindeki kesilme numaraları 1'den 5'e kadar, 8086'nın kesilme numaraları ise 0'dan 255'e kadar değişir. Bu değerlerin sınırları işlev tarafından denetlenir ve anlamsız değerler ile karşılaşıldığında hata durumu oluşur. İşlevin son argümanı, kesilme hizmet yordamı olarak tanımlanacak yordamın adıdır.

b. "enable" İşlevi.

İşleyici düzeyinde kesilmeleri açmak için kullanılır.

c. "disable" İşlevi.

İşleyici düzeyinde kesilmeleri kapamak için kullanılır. Kesilmeleri kapatırken dikkat edilmesi gereken nokta saat kesilmesi periyodunu geçmemektir. Eğer dizge saat kesilme periyodunu geçecek biçimde kesilmelere kapalı kalacak olursa gerçek zaman saati işlerliğini yitirir.

Kesilmeleri açıp kapamaya yarayan 'enable' ve 'dısable' işlevleri görevler ile kesilme hizmet yordamı arasında oluşacak kritik kesimleri korumada kullanılabilir. Örneğin 12 numaralı kesilme vektörü için bir dış birim dakikada bir kez kesilme üretirse bu kesilmelere göre saat değerini tutan kesilme hizmet yordamı ile bu saat değerini okuyan görev arasında kritik kesim oluşur. Bu kritik kesim saat ve dakika değerlerine erişilen kesimdir. Bu örneğe ilişkin program kesimi Çizim 3.30'da verilmiştir.

```
static  int saat, dakika;
        .
        .
     assign_interrupt(2,12,saat_say);
        .
        .
     disable();    /*  kritik kesim başı  */
     i=saat;        /*                     */
     j=dakika;      /*                     */
     enable();      /*  kritik kesim sonu  */
      .
      .
   saat_say()
   {
    dakika++;
    if (dakika == 60)
      {dakika = 0;
       saat++;
       if(saat == 24) saat=0;
   }
 }
```

Çizim 3.30. Kesilme Hizmet Yordamı ile Görev Arasında Kritik Kesim Örneği

3.2.7. Diğer İşlevler

Uygulamalarda hatta bulma, çalışan görev hakkında bilgi edinme, çekirdek dizgenin etkinliğini uygulamaya göre değiştirme gibi amaçlarla kullanmak için bir dizi işlev öngörülmüştür.

i. Hata bulmaya yönelik işlevler:

   a. "abort",
   b. "inform",
   c. "status",
   d. "reset_error",
   e. "stop_system",

ii. Etkinliğe yönelik işlevler:

   f. "clock_rate",
   g. "time_slice",
   h. "priority",

iii. Çalışan görev hakkında bilgi edinmeye yönelik işlevler:

   i. "my_id"
   j. "my_priority",
   k. "ready_task",
   l. "remaining_time".

a. "abort" işlevi.

Bu işlev anahtar niteliğindedir. İşlev uygulandıktan sonra hata durumu oluşursa dizge durdurulur ve hata iletisi verilir. Hata

iletisi içinde belirtilen numara hatanın nedenini açıklayan koddur.

b. "inform" İşlevi.

"abort" işlevinin karşıtı işlevdir. İşlev uygulandıktan sonra hata durumu oluştuğunda hata sözcücü hata türünü belirleyen bir kod ile günlenir ve işleme devam edilir. Yazılım aracılığı ile denetlenebilen bu hata sözcüğüne göre dizge yönlendirilir.

c. "status" İşlevi.

Dizgede oluşan hata ile günlenen hata sözcüğü içeriğini okumak için kullanılan işlevdir. Eğer sıfır değeri döndürür ise hata durumu oluşmadığını belirler. Sıfırdan farklı değerler hata durumunu ve bu değer hatanın türünü belirler. Hata kodları ek 1'de verilmiştir.

d. "reset_error" İşlevi.

Hata durumu belirlenip kodu alındıktan sonra ilerideki diğer hataları yakalayabilmek için hata sözcüğü sıfırlanır. Bu işlemi gerçekleştirmek için "reset_error" işlevi kullanılır.

e. "stop_system" İşlevi.

Dizgedeki görev yöneticinin devreden çıkarılması için kullanılır. Çok görevli ortam bu biçimde dondurulur. İşlev uygulandığında içinde bulunan yordam bundan sonra dizgenin durumunu gözlemek için kullanılır.

f. "clock rate" İşlevi.

Çekirdek dizgenin saat kesilme sıklığını programlayabilmek için "clock_rate" işlevi öngörülmüştür. Gerçek zaman saati kesilme sıklığı

$$F = \frac{1.19}{n+1} \text{ MHz}$$

formülü ile bulunur. n değerini belirlemek için "clock_rate" işlevi kullanılır. Örneğin "clock_rate(99) türünde bir ifade ile F =1.19/(99+1) = 11.9 KHz sıklığında saat kesilmesi elde edilir.

g. "time_slice" İşlevi.

Görev yöneticinin kullandığı "round-robin" yönetimdeki zaman dilimlerinin saat vurusu türünde belirlenmesi için "time_slice işlevi kullanılır. Örneğin 11.9 KHz sıklığındaki saat kesilmesi için "time_slice(10)" ifadesi ile 1/(11.9*1000/10)=0.0008 sn=0.8 msn'lik zaman dilimleri elde edilir.

h. "priority" İşlevi.

Çalışmakta olan görevin kendi önceliğini saptaması için öngörülmüş bir işlevdir. Örneğin "priority(100)" ifadesi ile çalışan görevin önceliği 100 olarak belirlenir. İşletim aşamasında bu işlev aracılığı ile öncelik değiştirilmez ise, görev kaynakları atandığı andaki öncelik ile çalışır.

i. "my_id" işlevi.

Çalışan görevin kendi kimlik numarasını alabilmesi için öngörülmüştür. Özellikle birçok görev tarafından kullanılan ortak yordamlar içinde gereksenen bir işlemdir. Yordam o anda hangi görev tarafından çağırıldığını belirleyebilir.

j. "my_priority" İşlevi.

Çalışan görevin kendi önceliğini alabilmesi için öngörülmüştür. İşletim aşamasında öncelikler üzerinde yapılan değişiklikler sırasında gereksenebilir. Örneğin bir görev önceliğini 1 azaltmak için

```
priority(my_priority()-1);
```

deyimi kullanılır.

k. "ready_task" İşlevi

Hazır görevler listesindeki ilk görevin kimliğini okumak amacı ile öngörülmüştür. Özellikle çekirdek dizgenin görev yönetimini denetleyebilmek amacı ile kullanılır. Örneğin kullanıcı saat kesilmesi için bir kesilme hizmet yordamı tanımlayarak görev yöneticisini kendisi yazabilir. Gerçek zaman saatinin kullanlmadığı bir görev yönetici Çizim 3.31'de verilmiştir.

```
        .
        .
   assign_interrupt(1,8,gorev_yonet);
        .
        .
gorev_yonet()
 { int kimlik;
   if (my_id() == 0) /* çalişan görev yoksa */
      {kimlik = ready_task(); /* ilk hazır görevin kimliği */
       if (kimlik !=0)   /* hazır listesi boş değil  */
                run_task(kimlik); /* ise görevi çalıştır. */
      }
 }
```

Çizim 3.31. İşlevler ile Gerçekleştirilmiş Bir Görev Yönetici Örneği

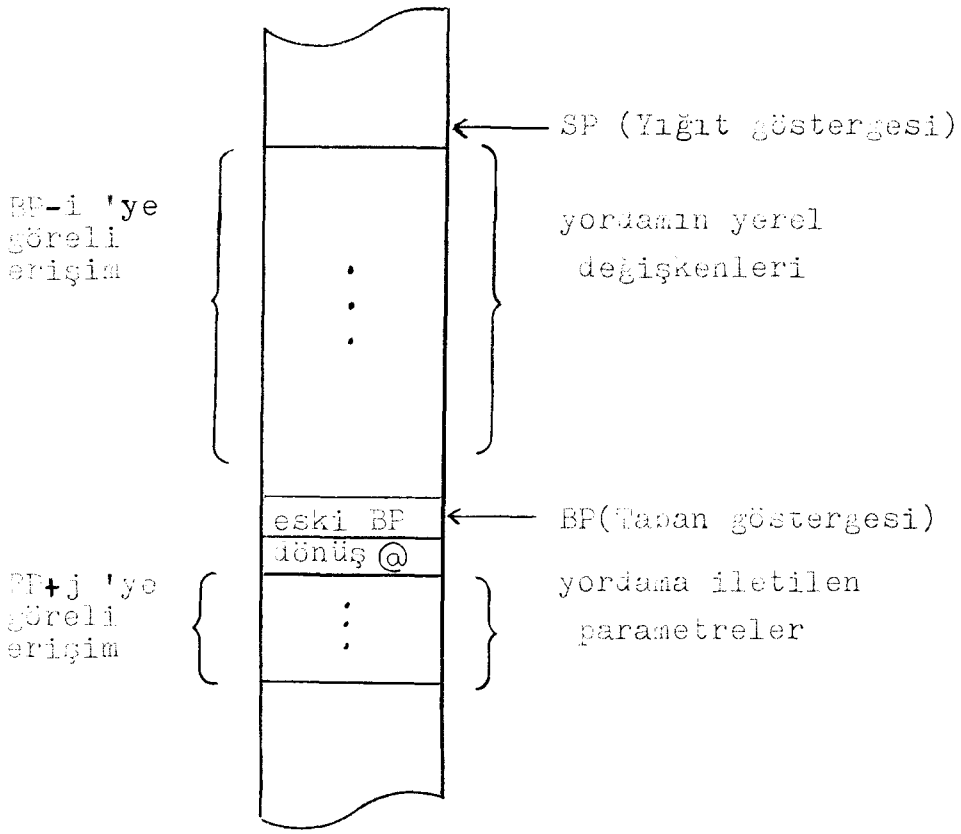
1. "remaining_time" İşlevi

Çalışan görevin kendisi için ayırılmış işletim diliminden geriye kalan zamanı bulmak için öngörülmüş bir işlevdir. Bu işlev çağırıldığında geriye kalan zamanı saat kesilme sayısı cinsinden verir.

3.3. Görevlerin Yapısı

Gerçekleştirilen dizge Microsoft C derleyicisinin bir kitaplığı biçiminde tasarlanması nedeni ile görevlerin tanımında C programlama dilindeki altyordamların yapısı gözönünde tutulmuştur. Özellikle altyordamlarda kullanılan standart yığıt yapısı önem taşımaktadır. Değişkenlerin yığıt içinde açılan bir alanda (activation record) tutulması yeniden girilir nitelikte kod

üretilmesini sağlamaktadır. Bu yığıt yapısı kullanılarak bir yordamın kodu değişik yığıt atamaları ile birden fazla görevin tanımında da kullanılabilir. Çizim 3.32'de Microsoft C derleyicisi tarafından kullanılan yığıt yapısı genel biçimde verilmiştir. Bu yığıt yapısına uyumlu biçimde 8086 simgesel programlama dili ile görev tanımlamak da olanaklıdır.



Çizim 3.32. Microsoft C'de Altyordamın Yığıt Kullanımı

Altyordamın çağırılmasında ve altyordamdan dönüşte kullanılan standart komutlar ile bu yığıt yapısı gerçekleşir.

Derleyici tarafından üretilen yordam çağırma komutları:

```
        .
        .
    push paramn    ; yordamın n. parametresi
        .
        .
    push param2    ;  yordamın 2. parametresi
    push param1    ;  yordamın 1. parametresi
    call yordam    ;  yordamın çağırılması
    add sp, n*2    ;  yığıttan parametrelerin atılması
```

Altyordam için üretilen kodlar:

```
yordam   proc    near
         push    bp      ; eski BP'yi sakla
         mov     bp,sp   ; yığıt göstergesini yeni BP olarak
                         ; belirle.
         mov     ax,m    ; m: yerel değişkenlerin bellekte
                         ; kapladıkları alanın uzunluğu.
         sub     sp,ax   ; yerel değişkenler için yığıtta yer
                         ; açılması (activation record).
         .
         .
; Yordama ilişkin komutlar
; (BP'nin içeriğini bozmayan)
         .
         .
         mov sp, bp  ; yerel değişkenler için açılan alanı
                     ; yığıttan çıkar.
         pop   bp    ; bp'nin eski değirini al.
         ret         ; yordamdan çağırana dön.
yordam endp
```

SONUÇ VE ÖNERİLER

Bu tez kapsamında IBM PC türü mikrobilgisayarlar üzerinde süreç denetimi gibi gerçek zamanlı çok görevli uygulamalara olanak sağlayan bir çekirdek işletim yazılımı tümüyle tasarlanarak gerçekleştirilmiştir. Bu çekirdek dizge kullanıcıya;

a. Çok görevli ortam kurma
b. Gerçek zaman saati kullanma
c. Görevler arası zaman uyumlama
d. Bellek yönetimi
e. Kesilmelerin yönetimi

gibi konularda destek sağlamaktadır.

Gerçek zamanlı, çok görevli çekirdek işletim yazılımları genelde giriş çıkış güdüm kesimlerinden ayrı düşünülen ve üzerinde çalıştığı bilgisayar dizgesinin bu amaçla kullanılan yazılım katmanından yararlanan bir bütün olarak ele alınırlar. Ancak bu tez kapsamında sınamalara taban oluşturacak ardıl bir giriş/çıkış biriminin sürülmesine ilişkin yazılım da ayrıca çok görevli ortam için yeniden oluşturulmuştur.

Kullanıcıya sunulan işletim yazılımı, C programlama dilini kullanmayı gerektirdiğinden, yukarıda sıralanan temel işlevleri kurmada esnek bir yaklaşım sağlanmıştır.

Çekirdek dizgenin süreç denetimi gibi konularda daha etkin kullanımını sağlamak için giriş/çıkış birimlerinin çeşitlendirilmesi ve çok görevli ortama uyacak bir giriş/çıkış dizgesinin yeniden ele alınması bu çalışmanın devamı niteliğinde

önerilebilir önemli bir noktayı oluşturacaktır. IBM PC türü bilgisayarların genişleyebilir olması nedeniyle, uygulamalarda gereksenecek çok çeşitli giriş/çıkış aygıtlarını sınıflayıp uygulamaya özel giriş/çıkış dizgesini geliştirmek olanaklıdır.

DEGINILEN BELGELER

Hansen P.B., 1978, Distrubuted Processes: A Concurrent Programming Concept : Comm. of ACM,Nov.,Vol 21, 934-941.

Ready J., Funck G., 1984, Software components create plug-in OS: Electronic Design April 19, 129-138.

Robert J. Oberg., 1983, Multitasking operating system backs real-time uses on 16-bit processors, Electronics, January 27, 119-121.

Tinnon J., 1982, Real-Time operating system puts its executive on silicon: Electronics, April 21, 137-140.

Crowl D.A., 1985, A Real-Time Fortran Executive: IEEE MICRO, August, 48-46.

Heath W.S., 1984, A System Executive for Real-Time Microcomputer Programs: IEEE MICRO, June 1984, 20-32.

Evanczuk S., 1983, Special Report: OS Real-Time : Electronics, March 24, 105-115.

Pedersen T., 1986, Process Administration in a High Level Language: Software-Practice and Experience Vol 16(4), April, 303-333.

Kerridge J., Simpson D., 1986, Communicating Parallel Processes, Software-Practice and Experience, Vol 16(1), Jan. 63-86.

Gomaa H., 1984, A Software Design method for Real-Time systems, Comm. of ACM, Vol 27(9), 938-949.

E K L E R :

1. Hata iletileri.
2. Tür tanımlarını içeren kütük.
3. Çekirdek dizgenin C programlama dili ile yazılan kesimi.
4. Çekirdek dizgenin ASM86 simgesel dili ile yazılan kesimi.

EK 1. Hata İletileri.

1. Free task skeleton not found (Maximum number of tasks).
2. Memory requirement didn't supplied.
3. Defined task isn't in terminated queue.
4. The task to run isn't in ready queue.
5. Signalled task isn't in waiting queue.
6. Not enough message buffer.
7. Invalid logical interrupt number.
8. Invalid 8086 interrupt number.

E K 2.

Tür tanımlarını içeren kütük.

(MULTASK.H)

```
#define _MAXLENGTH 40

/*****************************************************************
*                                                                *
*               TYPE DEFINITIONS                                 *
*                                                                *
*****************************************************************/

typedef  struct _queue {     /* Skeleton queue header  */
         struct _skt    *first;
         struct _skt    *last;
       } _QHEAD;


typedef  struct _skt   {      /* Skeleton data structures */
         int task_id;
         int *stack_ptr;
         int timer;
         int priority;
         struct _skt *next;
         int *mem_block;
       } _SKELETON;


typedef  struct _semaphore { /* Semaphore data structure */
         unsigned int counter;
         _QHEAD squeue;
       } SEMAPHORE;


typedef  struct _mhead  {
         SEMAPHORE s1;
         SEMAPHORE s2;
         struct _mbuffer *first;
         struct _mbuffer *last;
         struct _mbuffer *mfree;
       } MESSAGE;


typedef  struct _mbuffer {
         char mchars[_MAXLENGTH];
         struct _mbuffer *next;
       } _MBUFF;
```

```
/*************************************************************
*                                                            *
*              TYPE DEFINITIONS                              *
*                                                            *
*************************************************************/
#define _MAXTASK    10
#define _MAXLENGTH 40
#define _MAXMESSAGES 20
#define _MEMORY 20000     /* 20K words of memory */
#define _BLOCK 100
#define NULL 0

typedef struct _queue {     /* Skeleton queue header   */
        struct _skt    *first;
        struct _skt    *last;
      } _QHEAD;


typedef struct _skt  {      /* Skeleton data structures */
        int task_id;
        int *stack_ptr;
        int timer;
        int priority;
        struct _skt *next;
        int *mem_block;
      } _SKELETON;


typedef struct _semaphore { /* Semaphore data structure */
        unsigned int counter;
        _QHEAD squeue;
      } SEMAPHORE;


typedef struct _mhead  {
        SEMAPHORE s1;
        SEMAPHORE s2;
        struct _mbuffer *first;
        struct _mbuffer *last;
        struct _mbuffer *mfree;
     } MESSAGE;


typedef struct _mbuffer {
        char mchars[_MAXLENGTH];
        struct _mbuffer *next;
      } _MBUFF;
```

```
/*************************************************************
*                                                           *
*                    DATA STRUCTURES                        *
*                                                           *
*************************************************************/

/* TASK MANAGEMENT QUEUES */
static _QHEAD _ready,_running,_waiting,_defined;


/* TASK SKELETON FREE LIST */
static _SKELETON _ts[_MAXTASK+1];
static _QHEAD    _ts_free;


/*  MESSAGE BUFFERS  */
static _MBUFF _mb[_MAXMESSAGES+1];
static _MBUFF  *_free_mbuff;
static int _queue_semaph;

/* TASKS STACK AREAS */

static int *_sfree;
static int _tstack[_MEMORY+2];


/* TIMER SLEEP LIST */
static _QHEAD    _sleep_list;

static long int _clock;
static unsigned int _TIME_SLICE;
static unsigned int _system_lock;
static int ms;
static _SKELETON *back;
static int error_status=0;
static int error_switch=1;
```

E K 3.

Çekirdek dizgenin C programlama dili ile yazılan kesimi.

(TMZ.C)

```
/*************************************************************
*                                                           *
*                        FUNCTIONS                          *
*                                                           *
*************************************************************/


multitask()
   {
     int i,j;



     _TIME_SLICE=100;
/* TASK STATUS AND FREE SKELETON LIST */
     _ready.first=_ready.last=NULL;
     _running.first=_running.last=NULL;
     _waiting.first=_waiting.last=NULL;
     _defined.first=_defined.last=NULL;
     _ts_free.first=&_ts[0];
     for(i=0 ; i<_MAXTASK ; i++)
        {
          _ts[i].next=&_ts[i+1];
        };
     _ts[_MAXTASK].next=NULL;
     _ts_free.last=&_ts[_MAXTASK];



/* STACK AREA INITIALIZATION */

_sfree=&_tstack[0];
*_sfree=0;
*(_sfree+1)=_MEMORY;
*(_sfree+2)=_sfree;
*(_sfree+3)=_sfree;
*(_sfree+_MEMORY-2)=0;
*(_sfree+_MEMORY-1)=_sfree;


/* SLEEP LIST INITIALIZATION */

     _sleep_list.first=NULL;
     _sleep_list.last=NULL;


/* MESSAGE QUEUE INITIALIZATION */

     _free_mbuff=&_mb[0];
     for(i=0 ; i<_MAXMESSAGES ; i++)
        {
          _mb[i].next=&_mb[i+1];
        };
     _mb[_MAXMESSAGES].next=NULL;
     _queue_semaph=1;
```

```
        ms=1;

        _hardware_set();

    }


    /* DEFINE TASK EXECUTIVE */


    define_task(task_id,task_addr)
    int task_id;
    int (*task_addr)();

        {
          _SKELETON *ptr;
          disable();
          if(_get_first(&_ts_free,&ptr))
              {
                 ptr->task_id=task_id;
                 ptr->stack_ptr=task_addr;
                 _put_last(&_defined,ptr);
              }
            else error(1);
        enable();
        }

    /*    INITIALIZE TASK EXECUTIVE    */

    init_task(task_id,mem_required,prior)
    int task_id,mem_required,prior;

        {
          _SKELETON *ptr;
          int *mptr;
          disable();
          if(_get_defined(&_defined,task_id,&ptr))
              {
                if(_get_mem(mem_required,&mptr))
                    {
                       ptr->mem_block=mptr;
                       ptr->timer=_TIME_SLICE;
                       ptr->priority=prior;
                       _allocate(&ptr->stack_ptr,mptr+*(mptr+1)-2);   /* Set ini
tial task stack */
                       _put_last(&_ready,ptr);
                    }
                  else {
                          _put_last(&_defined,ptr);
                          error(2);
                       }
              }
            else error(3);
         enable();
        }
```

```
/* GET FIRST LIST ELEMENT SERVICE ROUTINE */


static int _get_first(head,ptr)

_QHEAD *head;
_SKELETON **ptr;

{
    if(head->first != NULL )
        {
         *ptr=head->first;
         head->first=head->first->next;
         if(head->first == NULL)
                head->last=NULL;
         return(1);
        }
     else return(0);
}


/* PUT LAST LIST ELEMENT SERVICE ROUTINE  */


static _put_last(head,ptr)

_QHEAD *head;
_SKELETON *ptr;

{
    if(head->last != NULL )
        {
          head->last->next=ptr;
          head->last=ptr;
        }
     else
        {
          head->first=ptr;
          head->last=ptr;
        };
    head->last->next=NULL;
}
```

```
/* FIND THE DEFINED TASK IN THE QUEUE AND GET IT */

static int _get_defined(head,id,ptr)

_QHEAD *head;
int id;
_SKELETON **ptr;

{
    struct _skt *ptr2,*ptr3;
    *ptr=head->first;
    ptr2=NULL;
    ptr3=*ptr;
    while(ptr3 != NULL && ptr3->task_id != id)
        {
          ptr2=ptr3;
          ptr3=ptr3->next;
        }
    if(ptr3 != NULL)
        {
          if(ptr2 != NULL)
              {
                ptr2->next=ptr3->next;
                if(ptr3 == head->last)
                   head->last=ptr2;
              }
            else _get_first(head,&ptr3);
          *ptr=ptr3;
          return(1); /* Found */
        }
      else return(0);  /* Not found */
}
```

```
/* GET MEMORY BLOCK */

int _get_mem(mem,mptr)

int mem;
int *mptr;

{
    int s,s1;
    int *p,diff;
    s=mem+6;
    s1=s / _BLOCK * _BLOCK;
    if (s1 != s) s=s1+_BLOCK;
    p=_sfree; /* Right pointer */
    do {
          if(*(p+1) >= s && *p == 0)
            {
              diff=*(p+1)-s;
              if(diff < _BLOCK)
                    *mptr=_get_all(p);
                  else *mptr=_get_bottom(p,s,diff);
              return(1);
            }
          p=*(p+3);  /* Advance the pointer */
       }  while(p != _sfree);
    return(0);
}


/* GET THE LOWER PART OF THE BLOCK */

static int _get_bottom(p,n,diff)
int *p,diff;
int n;

{
    int *ptr,*dummy;
    dummy=0;
    ptr=p+diff-2;
    *(ptr++)=0;             /* Upper part - mark2 */
    *(ptr++)=p;             /* Upper part - top pointer */
    *(ptr++)=1;
    *(ptr++)=n;
    *(ptr++)=p;
    *(ptr++)=*(p+3);
    *(p+n+diff-2)=1;
    *(p+n+diff-1)=p+diff;

    *(p+1)=diff;            /* Upper part - Length */
    *(dummy+*(p+3)/2+2)=p+diff;
    *(p+3)=p+diff;          /* Upper part - right pointer */
    return(p+diff);
}
```

```
/* GET THE WHOLE BLOCK */

static int _get_all(p)
int *p;
{
    *p=1;
    *(p+*(p+1)-2)=1;
    return(p);
}



/* GIVE BACK THE BLOCK OF MEMORY */

_back_mem(p)
int *p;

{
    if(*(p-2) == 0 && p >= _sfree)
        if(*(p+*(p+1)) == 0 && p+*(p+1) < _sfree+_MEMORY)
            _back_triple(p);
          else
            _with_top(p);
    else
      if(*(p+*(p+1)) == 0 && p+*(p+1) < _sfree+_MEMORY)
            _with_bottom(p);
        else
            {
              *p=0;
              *(p+*(p+1)-2)=0;
            };
}


/* GET FREE THE BLOCK OF MEMORY WITH NEIGHBOURS  */

static _back_triple(p)
int *p;

{
    int *ptr,l,*dummy;
    dummy=0;
    ptr=*(p-1);
    l=*(p+1);
    *(dummy+*(p+*(p+1)+3)/2+2)=ptr;
    *(ptr+3)=*(p+*(p+1)+3);
    *(ptr+1)+=l+*(p+l+1);
    *(p+l+*(p+l+1)-1)=ptr;
}
```

```
/* GET FREE THE BLOCK OF MEMORY WITH THE LEFT NEIGHBOUR */

static _with_top(p)
int *p;

{   int *dummy;
    dummy=0;
    *(dummy+*(p-1)/2+3)=*(p+3);
    *(dummy+*(p+3)/2+2)=*(p-1);
    *(dummy+*(p-1)/2+1) += *(p+1);
    *(p+*(p+1)-2)=0;
    *(p+*(p+1)-1)=*(p-1);
}


/*  GET FREE THE BLOCK OF MEMORY WITH THE LEFT NEIGHBOUR */

static _with_bottom(p)
int *p;

{   int *dummy;
    dummy=0;
    *p=0;
    *(dummy+*(p+*(p+1)+3)/2+2)=p;
    *(p+3)=*(p+*(p+1)+3);
    *(p+1)+=*(p+*(p+1)+1);
    *(p+*(p+1)+*(p+*(p+1)+1)-1)=p;
}

/* RUN TASK EXECUTIVE */

run_task(task_id)
int task_id;

{   _SKELETON *p;
    disable();
    if(_get_first(&_running,&p))
        {
         p->stack_ptr=context(1);
         _put_last(&_ready,p);
        }
    if(_get_defined(&_ready,task_id,&p))
        {
         _put_last(&_running,p);
         _running.first->timer=_TIME_SLICE;
         _task_switch(_running.first->stack_ptr);
        }
     else { error(4);
            enable();
            for(;;){};
          }
 enable();
}
```

```
/* PREEMPT THE RUNNING TASK EXECUTIVE */
preempt_task()

{
    _SKELETON *p;
    disable();
    _running.first->stack_ptr=context(0);
    _get_first(&_running,&p);
    _put_last(&_ready,p);
    enable();
    for(;;);
}

/* TERMINATE TASK EXECUTIVE */

terminate()
{
    _SKELETON *p;
    disable();
    _get_first(&_running,&p);
    _put_last(&_ts_free,p);
    _back_mem(p->mem_block);
    enable();
    for (;;)();  /* Wait for timer interrupt */
}


/* WAIT TASK EXECUTIVE */
wait_task()

{
    _SKELETON *p;
    disable();
    _running.first->stack_ptr=context(0);
    _get_first(&_running,&p);
    _put_last(&_waiting,p);
    enable();
    for(;;);  /* Wait for timer interrupt */
}
```

```
/* SIGNAL TASK EXECUTIVE */
 signal(task_id)
 int task_id;

{
    _SKELETON *ptr;
    disable();
    if(_get_defined(&_waiting,task_id,&ptr))
        {
          if(ptr->priority > _running.first->priority)
              {
               _running.first->stack_ptr=context(1);
               _put_last(&_running,ptr);
               _get_first(&_running,&ptr);
               _put_last(&_ready,ptr);
               _running.first->timer=_TIME_SLICE;
               _task_switch(_running.first->stack_ptr);
              }
           else
               _put_last(&_ready,ptr);
        }
     else
        error(5);
 enable();

}


init_semaph(s,value)
SEMAPHORE *s;
unsigned int value;
   {
     _SKELETON *p;
     disable();
     s->counter=value;
     s->squeue.first=s->squeue.last=NULL;
     enable();
    }

p(s)
SEMAPHORE *s;
 {
  _SKELETON *p;
  disable();
  if(s->counter == 0)
      {
       _running.first->stack_ptr=context(1);
       _get_first(&_running,&p);
       _put_last(&(s->squeue),p);
       enable();
       for(;;)();
      }
    else s->counter--;
  enable();
 }
```

```
sleep(tick)
int tick;
 { _SKELETON *p;
   disable();
   p=_sleep_list.first;
   if(p == NULL)
       {
         _running.first->stack_ptr=context(1);
         _running.first->timer=tick;
         _get_first(&_running,&p);
         _put_last(&_sleep_list,p);
         enable();
         for(;;)();
       }
   back=NULL;
   while(p != NULL && p->timer < tick)
      { tick -= p->timer;
        back=p;
        p=p->next;
      }
   if(p != NULL) p->timer -= tick;

   if(back == NULL)
      {
       _running.first->stack_ptr=context(1);
       _get_first(&_running,&p);
       p->next=_sleep_list.first;
       _sleep_list.first=p;
      }
   else
      {
       _running.first->stack_ptr=context(1);
       _get_first(&_running,&p);
       p->next=back->next;
       back->next=p;
       if(back == _sleep_list.last) _sleep_list.last=p;
      }
 p->timer=tick;
 enable();
 for(;;)();
}



set_timer(clock_val)
long unsigned int *clock_val;
 {
   disable();
   *clock_val= _clock;
   enable();
  }
```

```
v(s)
SEMAPHORE *s;
 {
  _SKELETON *p;
  disable();
  if(_get_first(&(s->squeue),&p) )
    { if(p->priority > _running.first->priority)
        {
         _running.first->stack_ptr=context(1);
         _put_last(&_running,p);
         _get_first(&_running,&p);
         _put_last(&_ready,p);
         _running.first->timer=_TIME_SLICE;
         _task_switch(_running.first->stack_ptr);
        }
      else _put_last(&_ready,p);
    }
 else s->counter++;
 enable();
}
```

```
get_timer(clock_ptr)
long unsigned int *clock_ptr;
 {
  disable();
  *clock_ptr=_clock- *clock_ptr;
  enable();
 }

time_slice(tick)
unsigned int tick;
 {
  disable();
  _TIME_SLICE=tick;
  enable();
 }



init_queue(queue,buff_count)
MESSAGE *queue;
int buff_count;
 {
  _MBUFF *ptr;
  int i;
  lock(&_queue_semaph);
  if(buff_count == 0)
      {
        while(queue->mfree != NULL)
            { ptr=queue->mfree;
              queue->mfree=queue->mfree->next;
              ptr->next=_free_mbuff;
              _free_mbuff=ptr;
            }
        unlock(&_queue_semaph);
        return;
      }

  init_semaph(&(queue->s1),buff_count);
  init_semaph(&(queue->s2),0);
  queue->first=NULL;
  queue->last=NULL;
  queue->mfree=NULL;
  for(i=1; i <= buff_count ; i++)
      {
        ptr=_free_mbuff;
        if(ptr == NULL) error(6);
        _free_mbuff=_free_mbuff->next;
        ptr->next=queue->mfree;
        queue->mfree=ptr;
       }
  unlock(&_queue_semaph);
 }
```

```
send(string,queue)
char *string;
MESSAGE *queue;

 {
  _MBUFF *ptr;
  char *c1,*c2;
  int i;
  p(&(queue->s1));
  lock(&_queue_semaph);
  ptr=queue->mfree;
  queue->mfree=queue->mfree->next;
  c1=ptr;
  c2=string;

  for(i=1; i<_MAXLENGTH && *c2 != '\0' ;i++)
     {
       *c1=*c2;
       c1++;
       c2++;
     };
  if(i  != _MAXLENGTH)   *c1=*c2;

  if(queue->last != NULL) queue->last->next=ptr;
  queue->last=ptr;
  ptr->next=NULL;
  if(queue->first == NULL) queue->first=ptr;

  unlock(&_queue_semaph);
  v(&(queue->s2));

 }
```

```
receive(queue,string)
char *string;
MESSAGE *queue;

 { _MBUFF *ptr;
   int i;
   char *c1;
   p(&(queue->s2));
   lock(&_queue_semaph);
   ptr=queue->first;
   queue->first=queue->first->next;
   if (queue->first == NULL) queue->last=NULL;
   c1=ptr;

   for(i=1; i<_MAXLENGTH && *c1 != '\0'; i++)
       { *string=*c1;
         c1++;
         string++;
       }
   if(i != _MAXLENGTH)  *string=*c1;
   ptr->next=queue->mfree;
   queue->mfree=ptr;
   unlock(&_queue_semaph);
   v(&(queue->s1));
 }
```

```
/********     SCHEDULER    *********
*                                   *
************************************/

_scheduler()

{_SKELETON *p;
 int greatest,check,id;
 _clock++;
 greatest=check=0;
 if(_sleep_list.first != NULL)
   {_sleep_list.first->timer--;
    while(_sleep_list.first != NULL && _sleep_list.first->timer == 0)
      { check=1;
        _get_first(&_sleep_list,&p);
        if(p->priority > greatest)
          { id=p->task_id;
            greatest=p->priority;
          }
        _put_last(&_ready,p);
      }
    if(_running.first != NULL)
         {if(check && greatest > _running.first->priority)
            {
             _running.first->stack_ptr=_sp_reg();
             _get_first(&_running,&p);
             _put_last(&_ready,p);
             _get_defined(&_ready,id,&p);
             _put_last(&_running,p);
             p->timer=_TIME_SLICE;
             _task_switch(_running.first->stack_ptr);
            }}
           else
            { if(check)
                {
                  _get_defined(&_ready,id,&p);
                  _put_last(&_running,p);
                  p->timer=_TIME_SLICE;
                  _task_switch(_running.first->stack_ptr);
                }
            }

   };
```

```
 if(_running.first == NULL)
   {if(_get_first(&_ready,&p))
          {
            _put_last(&_running,p);
            p->timer=_TIME_SLICE;
            _task_switch(_running.first->stack_ptr);
          }

   }
  else
   {if((_running.first->timer)-- == 0)
       { if(_ready.first != NULL)
            {
                 _running.first->stack_ptr=_sp_reg();
                 if(_ready.first != _ready.last)
                    { p=_running.first;
                      _running.first=_running.last=_ready.first;
                      _ready.first=_ready.first->next;
                      _running.first->next=NULL;
                      _ready.last->next=p;
                      _ready.last=p;
                    }
                  else
                    { _ready.first=_running.last;
                      _running.first=_running.last=_ready.last;
                      _ready.last=_ready.first;
                    }
                 _running.first->timer=_TIME_SLICE;
                 _task_switch(_running.first->stack_ptr);
            }
         else _running.first->timer=_TIME_SLICE;
       };
   };
}
```

```
assign_interrupt(lint,int8086,addr)
int lint,int8086,addr;
 {
   int int1(),int2(),int3(),int4(),int5();
   extern int logical_vector;
   int *p;

   if((lint < 1) || (lint > 5))           error(7);
   if((int8086 < 0) || (int8086 > 255))   error(8);
   p=&logical_vector;
   *(p+lint)=addr;
   disable();
   if(lint == 1)
     { set_vector(int8086,int1);
       enable();
       return;
     };
   disable();
   if(lint == 2)
     { set_vector(int8086,int2);
       enable();
       return;
     };
   disable();
   if(lint == 3)
     { set_vector(int8086,int3);
       enable();
       return;
     };
   disable();
   if(lint == 4)
     { set_vector(int8086,int4);
       enable();
       return;
     };
   disable();
   if(lint == 5)
     { set_vector(int8086,int5);
       enable();
       return;
     };

 }

static error(num)
int num;
{ if(error_switch)
    { stop_system();
      printf("System Aborted. Error no=%d",num);
    }
   else error_status=num;
}
```

```
int modem_status(port)
int port;
 { int temp;
   temp=inp(port+6);
   temp<<=8;
   temp+=(0x00ff & inp(port+5));
   return(temp);
 }

char serial_in(port)
int port;
 {
   while((modem_status(port) & 0x0001) == 0)
       { preempt_task(); };
   return(inp(port));
 }

serial_out(port,c)
int port;
char c;
 {
  while((modem_status(port) & 0x0040) == 0)
      { preempt_task(); }
  outp(port,c);
 }

abort()
 { error_switch=1; }

inform()
 { error_switch=0; }

int status()
 { return(error_status); }

reset_error()
 { error_status=0; }

int my_id()
 { return(_running.first->task_id); }

int my_priority()
 { return(_running.first->priority); }

int remaining_time()
 { return(_running.first->timer); }
```

```
clock_rate(pulse)
int pulse;
 { disable();
   outp(0x43,2);
   outp(0x40,pulse & 0x0ff);
   pulse>>=8;
   outp(0x40,pulse & 0x0ff);
   enable();
 }


priority(prior)
int prior;
 { _running.first->priority=prior; }


int ready_task()
 { return((_ready.first == NULL ? 0 : _ready.first->task_id)); };
```

E K 4.

Çekirdek dizgenin ASM86 simgesel dili ile yazılan kesimi.

(LOW.ASM)

```
        TITLE   low level routines

_TEXT   SEGMENT  BYTE PUBLIC 'CODE'
_TEXT   ENDS
CONST   SEGMENT  WORD PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT  WORD PUBLIC 'BSS'
_BSS    ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
DGROUP  GROUP   CONST,  _BSS,   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
PUBLIC  __allocate,__hardware_set,_context,__task_switch
PUBLIC __sp_reg,_disable,_enable,_lock,_unlock,_memory
PUBLIC __chkstk
extrn __scheduler:near,_preempt_task:near
PUBLIC _set_vector,_int1,_int2,_int3
PUBLIC _int4,_int5,_logical_vector,_stop_system


_TEXT   SEGMENT
enable_flag equ 200h

/*************************************************************************
*                                                                        *
*  name         : allocate stack                                         *
*  description  : performs initial allocation of the stack for           *
*                 the task.                                              *
*  inputs       : [BP+6] stack starting address. Stack grows             *
*                 to lower addresses.                                    *
*  outputs      : [BP+4] address of memory location which is             *
*                 used as task's stack pointer.                          *
*************************************************************************/

__allocate proc near
           push bp
           mov bp,sp
           push di
           push si
           mov di,[bp+6]        ;Task sp value
           pushf
           pop ax
           or ax,enable_flag
           dec di
           dec di
           mov ds:[di],ax          ; Flags
           dec di
           dec di
           mov ds:[di],cs
           mov si,[bp+4]
           mov ax,[si]          ;Task address
           dec di
           dec di
           mov ds:[di],ax
           sub di,2*7           ;AX,BX,CX,DX,SI,DI,BP
```

```
            dec di
            dec di
            mov [di],ds
            dec di
            dec di
            mov [di],ss
            dec di
            dec di
            mov [di],es
            mov word ptr [si],di
            pop si
            pop di
            pop bp
            ret
__allocate endp

vector     equ 8
count_val  equ 2000

/*************************************************************************
*                                                                        *
*  name         : hardware setting.                                      *
*  description  : updates the clock interrupt vector and starts  *
*                 the timer                                              *
*  inputs       : none                                                   *
*  outputs      : none                                                   *
*************************************************************************/

__hardware_set proc near
            cli
            push bp
            push es
            push di
            push ax
            xor ax,ax
            mov es,ax
            mov di,vector*4
            mov ax,es:[di]
            mov iaddress,ax
            mov ax,es:[di+2]
            mov isegment,ax
            mov word ptr es:[di],offset tickcnt
            mov es:[di+2],cs
            mov al,02h              ;counter 0,latch count,mode 2,binary
            out 43h,al
            mov ax,count_val
            out 40h,al
            mov al,ah
            out 40h,al
            pop ax
            pop di
            pop es
            pop bp
            ret
__hardware_set endp
```

```
/*************************************************************************
*                                                                        *
*   name          : timer tick counter                                   *
*   description   : interrupt service routine for timer interrupts.*
*                   Calls the scheduler on every interrupt.              *
*   inputs        : none                                                 *
*   outputs       : none                                                 *
*************************************************************************/
tickcnt proc near
            push ax
            push bx
            push cx
            push dx
            push si
            push di
            push bp
            push ds
            push ss
            push es

            mov al,20h
            out 20h,al          ; End of interrupt to 8259

            mov ax,ss
            mov ds,ax

            call __scheduler
            pop es
            pop ss
            pop ds
            pop bp
            pop di
            pop si
            pop dx
            pop cx
            pop bx
            pop ax
            iret
tickcnt     endp


_length equ 20+6+6
```

```
/*****************************************************************
*                                                                *
*  name         : save the context                               *
*  description  : saves the all registers of interrupted task    *
*  inputs       : number of parameters of calling routine.       *
*  outputs      : stack pointer value returns in AX register.    *
*****************************************************************/
_context proc near
          mov bx,[bp]   ;bp value
          push di
          push si
          push cx
          mov dx,sp     ;top of the stack
          sub dx,_length
          mov si,sp
          mov di,dx
          mov cx,12
          add cx,bp
          sub cx,sp

          mov ax,es
          push ds
          pop es
          cld
          rep  movsb
          mov es,ax

          mov sp,bp
          add sp,4


          mov cx,[bp+2]
          pushf
          pop ax
          or ax,enable_flag
          push ax

          push cs
          push cx
          push ax
          push bx
          push cx
          push dx
          mov di,dx
          mov ax,[di+2]
          push ax
          mov ax,[di+4]
          push ax
          push bx
          push ds
          push ss
          push es
          mov ax,sp
          mov sp,dx
```

```
            sub bp,_length
            pop cx
            pop si
            pop di
            ret
_context    endp


/*************************************************************************
*                                                                        *
*  name          : task switch.                                          *
*  description   : restores the registers of the task and                *
*                  transfers the control to it.                          *
*  inputs        : top of the stack pointer(second element in the        *
*                  current stack)                                        *
*  outputs       : none                                                  *
*************************************************************************/
__task_switch proc near
            pop ax    ;throw away the return address.
            pop ax    ;get task's sp value.
            mov sp,ax
            pop es
            pop ss
            pop ds
            pop bp
            pop di
            pop si
            pop dx
            pop cx
            pop bx
            pop ax
            iret
__task_switch endp


/*************************************************************************
*                                                                        *
*  name          : stack pointer.                                        *
*  description   : fetches the stack pointer value to high level         *
*                  language.                                             *
*  inputs        : none                                                  *
*  outputs       : AX contains the stack pointer value.                  *
*************************************************************************/
__sp_reg proc near
            mov ax,bp
            add ax,4
            ret
__sp_reg endp
```

```
/*************************************************************************
*                                                                        *
*  name         : disable interrupts.                                    *
*  description  : disables all interrupts at processor level.            *
*  inputs       : none                                                   *
*  outputs      : none                                                   *
*************************************************************************/
_disable proc near
         cli
         ret
_disable endp


/*************************************************************************
*                                                                        *
*  name         : enable interrupts                                      *
*  description  : enables all interrupts at processor level.             *
*  inputs       : none                                                   *
*  outputs      : none                                                   *
*************************************************************************/
_enable proc near
         sti
         ret
_enable endp


/*************************************************************************
*                                                                        *
*  name         : check stack.                                           *
*  description  : stack control routine for function entries.            *
*  inputs       : none                                                   *
*  outputs      : none                                                   *
*************************************************************************/
__chkstk proc near
         pop bx
         sub sp,ax
         push bx
         ret
__chkstk endp
```

```
/*************************************************************************
*                                                                        *
*  name         : lock                                                   *
*  description  : performs the lock operation which is used for          *
*                 protection of critical sections.                       *
*  inputs       : [BP+4] address of the binary semaphore.                *
*  outputs      : none                                                   *
*************************************************************************/
_lock     proc near
          push bp
          mov bp,sp
          push si
loop:     xor ax,ax
          mov si,[bp+4]
     lock xchg ax,[si]
          or ax,ax
          jnz continue
          call _preempt_task
          jmp loop
continue:
          pop si
          pop bp
          ret
_lock     endp


/*************************************************************************
*                                                                        *
*  name         : unlock                                                 *
*  description  : contrary to the lock operation. Sets free the          *
*                 semaphore.                                             *
*  inputs       : [BP+4] address of the binary semaphore                 *
*  outputs      : none                                                   *
*************************************************************************/
_unlock   proc near
          push bp
          mov bp,sp
          push si
          mov si,[bp+4]
          mov ax,1
          mov [si],ax
          pop si
          pop bp
          ret
_unlock   endp
```

```
/*************************************************************************
*                                                                        *
*  name         : set vector                                             *
*  description  : assigns the defined values of the predefined           *
*                 interrupt vector.                                      *
*  inputs       : [BP+4] for 8086 interrupt number                       *
*                 [BP+6] for interrupt service routine address.          *
*  outputs      : none                                                   *
*************************************************************************/
_set_vector proc near
          push bp
          mov bp,sp
          push di
          push es
          xor ax,ax
          mov es,ax
          mov di,[bp+4]      ; 8086 interrupt #
          add di,di
          add di,di
          mov ax,[bp+6]      ;interrupt service routine
          mov es:[di],ax
          mov es:[di+2],cs
          pop es
          pop di
          pop bp
          ret
_set_vector endp



/*************************************************************************
*                                                                        *
*  name         : Local interrupt entry points.                          *
*                                                                        *
*************************************************************************/

_int1 proc near
          push ax
          mov ah,2
          call interrupt_n
          pop ax
          iret
_int1 endp

_int2 proc near
          push ax
          mov ah,4
          call interrupt_n
          pop ax
          iret
_int2 endp
```

```
_int3 proc near
          push ax
          mov ah,6
          call interrupt_n
          pop ax
          iret
_int3 endp

_int4 proc near
          push ax
          mov ah,8
          call interrupt_n
          pop ax
          iret
_int4 endp

_int5 proc near
          push ax
          mov ah,10
          call interrupt_n
          pop ax
          iret
_int5 endp



/*************************************************************************
*                                                                        *
*  name         : interrupt handler                                      *
*  description  : switches the interrupt service routine which           *
*                 is written in high level language.                     *
*  inputs       : none                                                   *
*  outputs      : none                                                   *
*************************************************************************/
interrupt_n proc near
          push bx
          push cx
          push dx
          push si
          push di
          push bp
          push ds
          push ss
          push es
          mov al,20h
          out 20h,al
          mov si,offset _logical_vector
          mov al,ah
          xor ah,ah
          mov bx,ax
          call word ptr[si+bx]
          pop es
          pop ss
          pop ds
```

```
          pop bp
          pop di
          pop si
          pop dx
          pop cx
          pop bx
          ret
interrupt_n endp



/*************************************************************************
*                                                                        *
*  name         : stop system                                           *
*  description  : stops all system operations. This subroutine          *
*                 is called when system fails.                           *
*  inputs       : none                                                  *
*  outputs      : none                                                  *
*************************************************************************/
_stop_system proc near
          cli
          push es
          push ax
          push di
          xor ax,ax
          mov es,ax
          mov di,vector*4
          mov ax,iaddress
          mov es:[di],ax
          mov ax,isegment
          mov es:[di+2],ax
          pop di
          pop ax
          pop es
          sti
          ret
_stop_system endp
```

```
/*****************************************************************
*                                                                *
*  name         : memory                                         *
*  description  : it is used for memory access from C language   *
*  inputs       : [BP+4] segment value                           *
*                 [BP+6] offset                                  *
*                 [BP+8] the data which will written to the      *
*                 memory location.                               *
*  outputs      : none                                           *
*****************************************************************/
_memory proc near
          push bp
          mov bp,sp
          push si
          push es
          mov es,[bp+4]
          mov si,[bp+6]
          mov ax,[bp+8]
          mov es:[si],ax
          pop es
          pop si
          mov sp,bp
          pop bp
          ret
_memory   endp



_TEXT    ENDS

_BSS SEGMENT
counter    dw ?
_logical_vector dw 6 dup (?)
iaddress dw 0
isegment dw 0
;_BSS ENDS

                    END
```