Active objects in Symbian OS

Aapo Haapanen

University of Tampere

Department of Computer Sciences

Master's Thesis

April 2008

University of Tampere
Department of Computer Sciences
Aapo Haapanen: Active objects in Symbian OS
Master's Thesis, 51 pages, 17 appendix pages
April 2008

This thesis examines a programming construct of Symbian OS called active objects. Active objects encapsulate a request to an asynchronous service and the completion of that request. They can be used to implement cooperative multitasking inside one thread. Active objects are widely used throughout Symbian OS, and the Symbian documentation encourages their usage instead of multithreading.

In this thesis active objects are compared to threads by implementing a solution to classic producer/consumer problem using both programming methods. The performance of the solutions is then compared. The test results show that the active-object based solution performs the operation more quickly and uses significantly less memory than the thread-based solution.

Key words and phrases: Symbian, active objects, cooperative multitasking, asynchronous processing

Table of Contents

1. Introduction

Almost all computer programs that have a graphical user interface have to be able to perform multiple tasks at the same time. For example, a program has to be able to handle user input while it is performing a time consuming task. Many such programs are also event driven. Most of the time they are doing nothing, except waiting for user input.

Most modern operating systems have the ability to perform multiple tasks at the same time. The user is able to run several programs simultaneously, and within one program several tasks can be performed simultaneously. The most common way to implement multitasking inside one program is to use threads. A thread is one stream of program execution, and the operating system schedules processing time for all threads. This kind of multitasking is called preemptive multitasking.

Symbian operating system (OS) is a C++ based operating system, which is designed for data-enabled mobile devices. It has a fully object oriented design and a compact implementation. Symbian OS has full preemptive multitasking and supports threads, but it also offers an event based, cooperative alternative to threads: active objects. Many programming tasks that are usually implemented with threads can also be implemented with active objects. Active objects are widely used throughout the Symbian OS, thus it is important to understand active objects when developing code to the platform.

The goal of this thesis is to examine the active objects on Symbian OS. It compares the active objects to threads and tries to define some guidelines about when active objects should be used and when using threads would be advantageous. Active objects and threads are analyzed by creating small test programs using both programming techniques, and comparing the implementations in terms of time and memory consumption. Some less easily measurable considerations are also discussed.

This thesis focuses on Symbian OS version 7.0. That means that the test programs are implemented on that operating system version and the tests are run on Nokia 9300i, which uses that operating system. In Symbian OS v8.0b new kernel architecture was introduced, which included some changes to the handling of threads. Most of the results of this thesis should be valid for also other operating system versions, but especially the test results of the thread-based solution should be taken with a grain of salt when dealing with newer OS versions.

Chapter 2 of this thesis gives background information on cooperative and preemptive multitasking and on Symbian OS. Chapters 3 and 4 describe threads and active objects, respectively. Chapter 5 deals with the test programs and in Chapter 6 those programs are analyzed. In Chapter 7 the conclusions from this study are presented.

2. Background

To understand active objects one has to be familiar with issues related to multitasking and Symbian OS. In this chapter those issues are discussed.

2.1 Multitasking

The term *multitasking* is used to describe the ability of a computer system to run several programs seemingly simultaneously. Almost all modern operating systems are able to multitask.

However, computers are basically sequential in nature. A computer processor is able to perform only one instruction at the time. A computer that has no multitasking ability operates strictly sequentially. It can only have one thread of control. Such computer can only execute one program at a time, and if the computer has to perform some time-consuming task, all other tasks have to wait until that operation has completed. [Hansen, 1973]

Thus, it is usually operating system's job to offer multitasking ability to a computer system. Operating system is a set of system programs which controls all the computer's resources and provides the base upon which the application programs can be written. [Tanenbaum, 1987]

On single processor systems the operating system gives central processing unit (CPU) time to different tasks based on the tasks' priorities and on the scheduling algorithm used. Such multitasking is usually called *multiprogramming*. On multiprocessor systems the operating system has several CPUs from which it can give processing time, which adds some complexity to the problem. Sharing the workload between several processors is called *multiprocessing*. [Deitel, 1984] However, this thesis concentrates on single processor systems, because currently no multiprocessor systems exist that use Symbian OS.

Process is a very central term to the multitasking. A process can be defined as a program in execution. It consists of the executable program, its data and stack, program counter, stack pointer, other registers and other data needed to run the program. The difference between program and process is subtle, but important. Process is an activity, which can be started, suspended etc, while a program is more like a static set of instructions. [Tanenbaum, 1987]

The CPU is able to execute only one process at a time. The process that is currently in execution by the CPU is said to be *running*, or in the *running state*. Other processes that are ready to be executed, but don't right now have the CPU, are *ready* processes, or in the *ready state*. In addition to the processes that are ready to be executed there are processes that are waiting for some event to happen. Those processes are *blocked*, or in the *blocked state*. [Tanenbaum, 1987]

The operating system makes sure that all processes in the ready state have a chance to run. To do this, it alternates the process that is in the running state. The changing of the running process is called *context switching*. During the context switch the operating system takes one of the processes that were in the ready state and switches it to the running state. The process that was previously running is changed to the ready state. The operating system must later be able to activate the process at exactly the same state it was before the change. This means that all information about the process must be saved during the context switch. [Tanenbaum, 1987]

The term *thread* is often used in programming languages to denote some kind of a lightweight process. A common technical distinction between threads and processes is that a process runs in its own address space, while a thread runs inside an address space of a process. In theoretical discussion of concurrency the distinction is often unnecessary and the term process is used exclusively. [Ben-Ari, 2006]

The part of the operating system that gives the CPU to different processes is called *dispatcher*. It decides which process will get the CPU when it next becomes available. This is also called low-level scheduling. [Deitel, 1984]

Through multitasking the CPU usage of a computer system can be maximized. Usually on computer system most of the time is spent waiting for some input or output operation to finish – especially if there is a human interacting with the computer and doing the input. By running several processes seemingly simultaneously, other processes can still operate normally when one process is waiting for some external event.

2.1.1 Preemptive and cooperative scheduling

Scheduling of processes is *nonpreemptive* if, once a process has been given the CPU, it cannot be taken away from that process until the process itself relinquishes the CPU. If the CPU can be taken away from the process, the scheduling is *preemptive*. [Deitel, 1984]

In nonpreemptive systems multitasking can be achieved if the processes cooperate. They can voluntarily yield the CPU to each others. Thus such system is called *cooperative multitasking*. Programs designed to run on cooperative multitasking system must be carefully designed, because if a misbehaving process doesn't relinquish the CPU, other processes won't be able to run at all.

Most current operating systems, including Symbian OS, offer preemptive multitasking. When the dispatcher in preemptive multitasking system gives the CPU to a process, it also sets a timer. If the process has not terminated until the timer runs out, the dispatcher switches the CPU to next process and the previously active thread is moved back into the dispatcher's activation queue. Thus all active processes get to run for a set amount of time in turns.

There are several ways to schedule the execution of the processes. One simple and commonly used scheduling method is *round-robin scheduling* where all the processes that are ready to run are executed in turn for a set amount of time. Each process gets a same share of CPU time. The situation gets more complicated if process priorities are added to the system. A higher priority process may always preempt a lower priority process, or the process with a higher priority may get a larger share of the CPU time than the lower priority process.

Preemptive multitasking is a lot more robust solution than cooperative multitasking. In preemptive multitasking one process can't steal all CPU time to itself. It also guarantees a reasonably quick response time for interactive applications.

However, there is also a drawback to preemptive scheduling. A context switch between two processes involves overhead, and in preemptive systems such a context switches can occur very rapidly. A cooperative multitasking can also make some programming tasks simpler, because the programmer can be certain that the CPU can't be switched to another process in the middle of some critical operation. Therefore, in some circumstances cooperative system may be a better solution.

In Symbian OS active objects are a way to implement cooperative multitasking inside a single thread, while the threads are scheduled preemptively. Active objects will be examined in following chapters.

2.1.2 Concurrent processes

Having several processes running simultaneously and perhaps interacting with each other gives rise to some new programming challenges. Two or more processes that exist at the same time are called *concurrent processes*. Although in single-processor systems only one process can be in execution at time, it is useful to think that all the processes that are in the running or ready state are executed concurrently.

On a multitasking system it is important that the data of each process is protected against unintended interference from other processes and that the results of each computation must be independent from the speed at which the computation is performed. [Hansen, 1973]

The simplest situation occurs when two processes are completely independent of each other. Such processes are called *disjoint processes*. Disjoint processes are processes that share no resources between themselves. For instance, they operate on completely different sets of variables. Such processes can't disrupt each other, and the relative order of their execution makes no difference to the results of the operations. [Hansen, 1973]

However, often processes are somehow dependent on each others. If processes operate on common resources, they must not disrupt each other. For instance, when one process is reading data, other processes may not change that data until the reading process is ready. Processes may also be cooperating, and to facilitate the cooperation they must be able to exchange synchronizing information or results between each other.

The simplest form of interacting between processes is *mutual exclusion*. Mutual exclusion means that some operations in processes A and B should never be executed at the same time. For example, there might be some peripheral device, which only one process can use at a time. Another example is a situation where one process is updating some common variable. No other process should use that variable when one process is updating it. [Hansen, 1973, Deitel, 1984]

An elegant solution to the mutual exclusion problem can be achieved through *critical sections*. Critical section is a part of a program that can only be executed by one process at a time. When a process is executing its critical section, other processes wanting to enter critical sections must wait until there are no other processes in a critical section. If there are several processes waiting to enter a critical section, the method of deciding who is allowed to enter a critical section must be fair. That means that no process may be held off indefinitely from entering a critical section. That can be achieved, for example, by first-in-first-out (FIFO) queue. [Dijkstra, 1965a, Dijkstra, 1965b, Hansen, 1973, Deitel, 1984]

Process interaction through critical sections is quite indirect. For processes to be able to cooperate, they must have a more direct means to communicate with each other. A simple synchronizing signal between processes is called *semaphore*. It is a non-negative variable, which can only be accessed through two specific operations. The names of the operations are usually *V* and *P* or *Signal* and *Wait* in the literature. In Symbian OS implementation of semaphores they are called *Signal* and *Wait*, so those names are used in this thesis. *Signal* increases the value of the semaphore, and *Wait* decreases the value, when it is possible to do so without the value becoming negative. If a *Wait* operation would cause the semaphore to become negative, it will wait for a *Signal* before proceeding. Again, if there are several processes waiting for a *Signal*, the method of deciding which *Wait* operation to complete must be fair. [Dijkstra, 1965a, Deitel, 1984]

The semaphore operations are indivisible. Normally increasing or decreasing the value of a variable is done in at least two separate operations: first reading the old value and then saving the new modified value. With concurrent processes there is a risk that another process accesses the variable between the read and write operations. With semaphore operations there is no such risk. [Dijkstra, 1965a]

A simple example of process synchronization through semaphores is a situation where one process (*processone*) must wait for some condition before it can proceed, and another process (*processtwo*) is able to observe the completion of that condition. In that case a semaphore can be initialized to zero. When *processone* gets to a situation where it must wait for the condition it issues a Wait operation for the semaphore. When *processtwo* observes the fulfillment of the condition it issues a Signal operation for the semaphore, and *processone* may proceed. The mechanism works even if *processtwo* signals the semaphore before *processone* issues the Wait operation. [Deitel, 1984] Note that it is also easy to implement mutual exclusion and critical sections with semaphores.

One common relationship between processes is *producer/consumer relationship*. In producer/consumer relationship one process produces information, while another process processes the information as it becomes available. That means there is a one-way information flow between the processes. Usually there is some kind of a buffer where the producer puts the information it produces, and from where the consumer takes the information it processes. Now there is a need for process synchronization. The producer must not create more information if the buffer is full, and the consumer must not try to process information when the buffer is empty. Additionally, only one process may access the buffer at the same time. [Dijkstra, 1965, Deitel, 1984]

2.2 Symbian OS

Symbian OS is an operating system designed for data-enabled mobile phones. The roots of Symbian OS go back to Psion handheld organizers and their operating system, EPOC32. In 1998 Symbian Limited was formed by Psion, Nokia, Motorola and Ericsson, and the operating system was re-branded as

Symbian OS. Later also Siemens, Panasonic and Sony-Ericsson have joined the company, while Psion has sold its share to Nokia. [Symbian]

According to Symbian [Symbian], in 2007 there were 68 phone models using Symbian OS commercially available, and 77.3 million mobile phones using Symbian OS were sold. The market share of phones with Symbian OS was 7% of all mobile phone sales.

Around the time of the formation of Symbian Ltd the mobile phones were getting more and more complicated, and were starting slowly to move from single-use appliances to a platform for many different kinds of applications. Thus, a need for a real operating system emerged. Symbian was formed to develop Psion's EPOC32 operating system into an operating system for data-enabled mobile phones.

According to Mery [2002] the mobile phone market has five key characteristics, which make it unique and require a specifically designed operating system:

- Mobile phones are small and mobile. This limits all resources available for the system. A mobile phone has limited amount of electricity, CPU power, execution memory, and storage size. Yet the operating system should be always available.
- Mobile phones are aimed at the mass market. This means that the operating system should be very reliable. User data should never be lost, and the phone should never lock up or even need to be rebooted.
- Mobile phones are occasionally connected. They can be connected to a mobile phone network or to another device, or they might not be connected to any outside system.
- Manufacturers need to be able to differentiate their products in order to innovate and compete in a fast-evolving market. The operating system needs to adapt to and support different device form-factors and input methods.
- The operating system must allow development of third party applications and services.

Symbian OS has been developed from these five key points. That has resulted in an operating system that is quite distinct from any desktop/workstation/server operating system, but which is also quite different from embedded operating systems. [Mery, 2002]

2.3 Technical overview of Symbian OS

Symbian OS has a microkernel architecture, where the kernel handles only the lowest level machine resources, CPU cycles and memory. The kernel runs in a supervisor mode, which means that it has a hardware-supported privileged access to hardware resources and only it can execute some CPU operations. Other programs must access hardware though kernel API. Most of the services offered by the operating system are handled by servers that run in user mode. [Tasker, 2000, Morris, 2007]

Symbian OS is fully object-oriented operating system. It is implemented in C++, although some of the design decisions are unique to the Symbian OS. For instance, the exception handling doesn't use the standard C++ model; instead Symbian OS uses its own leave-mechanism.

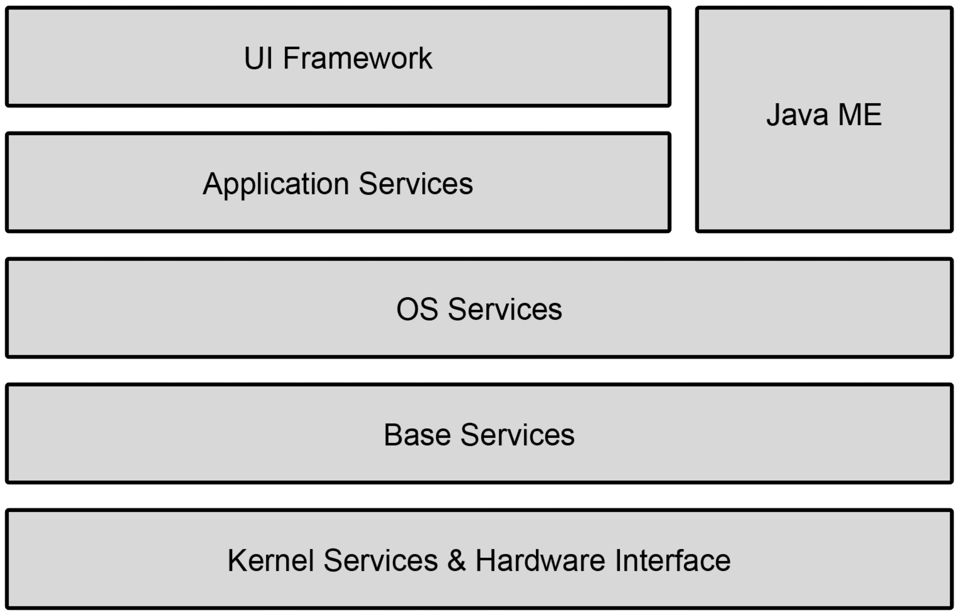

Figure 1: A layered view of the components in Symbian OS [Morris, 2007]

Due to the modular nature of Symbian OS and the various technologies supported, any attempt to draw an architectural image of the system get easily very complicated. Figure 1 gives a one way to look at the different components in the system, where the components are divided into layers starting from UI layer all the way down into the kernel and hardware interface layer.

UI framework layer provides the basic user interface building blocks for the applications. It acts as an interface between Symbian OS and the variant UI layer. Symbian OS doesn't include an actual user interface. The implementation of the UI is left to the licensees. Two most important user interfaces currently available are S60 by Nokia and UIQ by UIQ Technology AB (fully owned by Sony-Ericsson). The purpose of the UI framework layer is to allow the customization of the UI without fragmentation. [Morris, 2007]

Application services layer provides UI-independent services to the applications. It includes system services used by all applications, such as application framework, technology-specific logic that is used by multiple applications, such as support for messaging and multimedia protocols, and it also includes services supporting specific applications such PIM (personal information management) and office applications. [Morris, 2007]

OS services layer provides services that extend the services provided by the kernel and low-level system libraries into a full operating system. It includes servers, frameworks and libraries that implement support for graphics, communications, connectivity and multimedia. It also includes some generic system components such as C standard library. [Morris, 2007]

Base services layer provides the lowest-level user side services. This layer together with the kernel services layer forms the minimal operating system. Since the actual kernel is a microkernel, everything above the basic operating system privileges is kept outside the kernel and runs in the user mode. [Morris, 2007]

The lowest layer, kernel services and hardware interface layer, contains the operating system kernel itself, and also the components that abstract the interface to the underlaying hardware, including device drivers. Everything in this layer runs in the supervisor mode. [Morris, 2007]

As mentioned above, Symbian OS uses client-server model throughout the system to implement services. The kernel itself is a server that offers access to the hardware. File server handles the file system; font and bitmap server offers access to the physical display; window server handles the windowing system and is the heart of the applications' event handling system. [Morris 2007]

Symbian OS uses asynchronous services widely through the system. That means that the services are implemented using the request—callback model instead of the blocking model. Active objects are used to encapsulate the calling of an asynchronous service and handling the completion of the request. Active objects are examined in more detail in Chapter 4.

Symbian OS is optimized for event handling [Harrison, 2003]. With GUI systems the system spends most of the time waiting for user input, after which it has to quickly respond to that input. User input can be thought of as distinct events that the system handles. When the user presses a key on the keyboard, he creates an event, and when the key is released, another event is created. If the user has a pointing device, that also creates events for the system to react to. Both active objects and servers play a central role in the event handling system. When an application wants to handle certain events, it issues an asynchronous request – using active objects – to the server handling such events. And when the event happens, the request is completed and the application can handle the event and possibly renew the request to receive the events.

Symbian OS version 8.0b introduced a new kernel called EKA2 (EPOC Kernel Architecture 2). The most important changes were the introduction of real-time capabilities, which makes it possible to run GSM protocol stack on the operating system, and the introduction of platform security, which is intended to make the operating system more robust against malicious code. EKA2 also introduced some changes to the handling of threads inside the kernel [Sales, 2005]. This thesis focuses on Symbian OS v7.0, which uses the older EKA1 kernel.

3. Threads in Symbian OS

Symbian OS provides preemptively scheduled threads and processes. It also provides standard mechanisms that are needed to synchronize access to resources shared between threads. These methods include semaphores, mutexes and critical sections.

In Symbian OS, a thread is defined as the unit of execution. It is the entity that the kernel schedules, i.e. for which it allocates CPU resources. A process is defined as a collection of threads that share the same address mapping. The process is the fundamental unit of protection in Symbian OS. Each thread within a process can read and write from any others' memory, but they can't directly access the memory of other processes. Each application runs inside its own process. [Sales, 2005, Tasker, 2000]

Threads are preemptively scheduled by the kernel. Each thread has a priority, and the thread that has the highest priority and is not in the blocked state is run by the kernel. If several threads have the same priority, they are time-spliced in a round-robin basis. [Stichbury, 2005] Threads may be blocked because they are waiting for an event to happen, and they resume when the event happens. A context switch between threads inside one process is much cheaper operation than a context switch between threads in different processes. The reason for that is that a context switch between processes requires changes to MMU (memory management unit) settings and various caches need to be flushed. Still, even a context switch between two threads inside the same process is much more expensive operation than, for example, a function call. [Tasker, 2000]

In addition to the user processes, there is also a special kernel process with two threads that run in the supervisor privilege level. One of them is the kernel server thread and it always has the highest priority in the system. The other one is the idle thread, which always has the lowest priority in the system. The idle thread switches the processor to the idle mode, allowing the system to conserve power. [Stichbury, 2005, Tasker, 2000] The focus of this thesis is user level programs, so by default, user threads are meant when discussing threads.

3.1 RThread

The threads in Symbian OS are accessed through *RThread* class. A thread is a kernel object, and *RThread* represents a handle to the object. The full definition of the class is too long to be included here, but Code fragment 1 contains an abbreviated version of the definition, with the most important functions included.

```
class RThread : public RHandleBase
    {
public:
    inline RThread();
    IMPORT_C TInt Create(const TDesC& aName, TThreadFunction
        aFunction,TInt aStackSize,TAny* aPtr,RLibrary* aLibrary,RHeap*
        aHeap, TInt aHeapMinSize,TInt aHeapMaxSize,TOwnerType aType);
    IMPORT_C TInt Create(const TDesC& aName,TThreadFunction
        aFunction,TInt aStackSize,TInt aHeapMinSize,Tint
        aHeapMaxSize,TAny *aPtr,TOwnerType aType=EOwnerProcess);
    IMPORT_C TInt Create(const TDesC& aName,TThreadFunction
        aFunction,TInt aStackSize,RHeap* aHeap,TAny* aPtr,TOwnerType
        aType=EOwnerProcess);
    IMPORT_C TInt Open(const TDesC& aFullName,TOwnerType
        aType=EOwnerProcess);
    IMPORT_C TInt Open(TThreadId aID,TOwnerType aType=EOwnerProcess);
    IMPORT_C void Resume() const;
    IMPORT_C void Suspend() const;
    IMPORT_C void Kill(TInt aReason);
    IMPORT_C void Terminate(TInt aReason);
    IMPORT_C TThreadPriority Priority() const;
    IMPORT_C void SetPriority(TThreadPriority aPriority) const;
    };
```

Code fragment 1: An abbreviated definition of RThread class from e32std.h

The class is inherited from *RHandleBase* class, which provides some standard handle manipulation functionality, such as *Duplicate()* for duplicating the handle and *Close()* for closing the handle.

The default constructor initializes the class to a pseudo handle set to the constant *KCurrentThreadHandle*. That value is treated as a special case by the kernel and can be used to access the current thread. If a proper handle to the current thread is needed, it can be created using *Duplicate()* on the pseudo handle [Stichbury, 2005].

A new thread is created with *Create()* function. There are several overloaded versions of the function to allow setting various options associated with the heap of the thread. A thread may have its own heap, or it may use the heap of the thread that created it. The size of the heap can change during the execution of the thread, and the minimum and maximum sizes of the heap can be defined during the thread creation. The size of the thread's stack is fixed to the value defined at the time of the thread creation. The default stack size is 8 KB. Other parameters given to the *Create()* function include the thread's name, a pointer to the function where thread execution starts, and a pointer to data that is passed as a parameter to the thread function. All new threads are created with priority *EPriorityNormal*, and the priority can be changed after the creation with *SetPriority()* function. [Stichbury, 2005]

A handle to an existing thread can be acquired with *Open()* function. The thread to be opened can be indicated by either the name of the thread or the identification number of the thread. [Symbian, 2002]

When a new thread is created, it is initially put into the suspended state. The thread is started with *Resume()* function. The execution can be again suspended with *Suspend()* function. The thread is permanently ended by using functions *Kill()* and *Terminate()*. Both functions take an integer as a parameter. The parameter represents the exit reason of the thread. The functions act in a similar way, but information about which function was used can be retrieved from the ended thread, along with the exit reason. [Stichbury, 2005]

3.2 Mutexes, semaphores and critical sections

For synchronizing the execution of threads, Symbian OS provides mutexes, semaphores and critical sections. Mutexes and semaphores can be either local, which means they are restricted to the current process, or they can be global, in which case they have a name that can be used to find the object in other processes. Critical sections are always local to a single process. [Symbian, 2002]

Mutexes and semaphores are kernel objects, and they are manipulated through handles *RMutex* and *RSemaphore*. Both classes are used in a similar way. A new mutex or semaphore is created with function *CreateLocal()* or *CreateGlobal()*. The global version of the function takes the name of the object as a parameter. Creating a semaphore also requires the initial value of the semaphore. After the mutex or semaphore is created, it is used with functions *Wait()* and *Signal()*. [Symbian, 2002]

Critical sections are implemented using semaphores, with class *RCriticalSection* inheriting from *RSemaphore*. The critical section handle doesn't expose all of the semaphore's functionality, so only a local critical section can be created. [Symbian, 2002]

4. Active objects

Active objects are a programming concept that is quite unique to Symbian OS. They are a way to implement nonpreemptive multitasking inside a thread. They are also closely related to the event handling system in Symbian OS.

At the core of a thread that uses active objects is an active scheduler. It acts as a kind of a mini-kernel for that thread, while active objects act like nonpreemptively scheduled mini-threads. Each thread may have only one active scheduler, while a thread with an active scheduler has one or more active objects. [Tasker, 1999]

A thread with active objects is basically an event handler. Event handling systems are based around programs requesting some services, which will then complete at a later time. Such completion of a request is called an event. When an outstanding request completes, the requester must handle it. Active objects encapsulate this relation between making a request and handling its completion. Each active object is responsible for making and handling just one outstanding request at a time. [Tasker, 1999]

Often a single thread will issue many outstanding requests. Each request may complete at any time, but each request is guaranteed to complete only once. Active scheduler handles the completion of requests and calls the active object responsible for handling the request. Each active object has a *RunL()* function, which is called when the request of that active object completes. If there are several completed requests to handle, the active scheduler decides the order in which they are handled. [Tasker, 1999]

Since active objects are scheduled nonpreemptively, their *RunL()* functions must return relatively quickly. When an active object is executing its *RunL()* function, no other active object inside the thread can run until the function returns.

In Symbian OS almost all threads have an active scheduler. It follows that almost all code runs inside *RunL()* functions of active objects. However, for most part the programmer doesn't have to think about active objects, because the frameworks of Symbian OS provide them. For instance, a program using GUI (graphical user interface) framework must implement *OfferKeyEventL()* function if it needs to handle keyboard events. That function is called by a *RunL()* function implemented in the framework [Stichbury, 2005]. All the programmer has to know is that the function must return relatively quickly, so that the application remains responsive. Typically applications and servers in Symbian OS contain only one thread, and the asynchronous processing is handled completely by active objects. [Harrison, 2003]

It must be noted that the nonpreemptive multitasking implemented by active objects is a bit different from the traditional cooperative multitasking. In cooperative multitasking tasks must actively tell other tasks when they can be executed, using *Yield()* or similar function. This easily leads to messy programming. In Symbian OS, each event must be handled completely before the other events can be handled. [Harrison, 2003]

Almost everything that can be implemented using threads can also be implemented using active objects. In practice, the use of multithreading in an application is quite rare in Symbian OS. An example of tasks that can’t be implemented using active objects is a long-running task that can’t be reasonably split into short discrete parts. Another is a task that requires a real-time response because no active object can preempt a running active object. [Tasker, 1999]

4.1 Implementation of active objects in Symbian OS

4.1.1 Asynchronous requests

As mentioned above, active objects are used to encapsulate making an asynchronous request and handling its completion. In Symbian OS any function that takes a *TRequestStatus&* parameter is designed to function asynchronously.

[Tasker, 1999] Classes containing such functions are usually called *asynchronous service providers*. [Harrison, 2003]

TRequestStatus is simply a well-encapsulated integer, where the only operations available are assignment and comparison. When an asynchronous request is issued, the asynchronous service provider sets the *TRequestStatus* parameter to *KRequestPending*. When the request completes, the service provider assigns the completion code to *TRequestStatus*. The completion code is usually one of the standard error codes of Symbian OS; anything except *KRequestPending* is permissible. For successful completion of the request the code usually is *KErrNone*. Each request must complete precisely once. [Tasker, 1999, Harrison, 2003]

The issuer of an asynchronous request must call *User::WaitForRequest()* or *User::WaitForAnyRequest()* function to wait for the completion of request, and for each wait, exactly one completion of a request must be handled. When active objects are used to handle the requests, this is done by the active scheduler. [Tasker, 1999] When the asynchronous service provider completes the request, it must call the *User::RequestComplete()* function if the provider is in the same thread as the requestor. If the requestor is in a different thread, it must call *RThread::RequestComplete()* for the requesting thread's handle. [Stichbury, 2005]

Every asynchronous service provider must also provide a cancel function to cancel an outstanding request. When a cancel function is called and the request has not already been completed, the service provider must either immediately complete the request with completion code *KErrCancel*, or if the request can be completed normally in a guaranteed very short time, the request may be completed normally. The cancel function must work synchronously, so that when the function returns, the request has been completed. In any case canceling request may not break the rule that every request must complete exactly once. [Tasker, 1999, Harrison, 2003]

4.1.2 Handling the completion of asynchronous requests

Code fragment 2 shows how a simple call to an asynchronous function could be implemented. [Tasker, 1999]

```
    RFile file;
    TRequestStatus readStatus;
    ...
    file.Read(buffer, readStatus); // issue request
    User::WaitForRequest(readStatus); // wait for completion
```

Code fragment 2: An example of a call to an asynchronous function

However, with this kind of coding the thread cannot do anything while waiting for the request to complete. Normally a single thread will issue several outstanding requests. Each request may complete at any time, but each request is guaranteed to complete exactly once. The thread must be able to handle this correctly. A wait loop for handling several outstanding requests could look like this [Tasker, 1999]:

1. Call *User::WaitForAnyRequest()*. This will either suspend the thread until a request completes, or if one or more requests have already completed, the function will return immediately.
2. Search through the statuses of requests issued but not yet handled for one that is not *KRequestPending* any more.
3. When such a request is found, handle its completion, and mark the request no longer issued.
4. Go back to 1.

It is important that each pass through this function handles exactly one completion of a request, because there must be as many calls to *User::WaitForAnyRequest()* as there are requests issued. If *User::WaitForAnyRequest()* is called when there are no outstanding requests, the function will never return.

The wait loop above with addition of priorities for requests is encapsulated in *CActiveScheduler*.

4.2 CActive

Class *CActive* encapsulates a single request to an asynchronous service provider. It is a pure virtual class, so users of it must inherit from it and implement its pure virtual member functions, such as *RunL*. *CActive* is defined in header e32base.h and the definition is shown in Code fragment 3.

```
class CActive : public CBase
```

```
    {
public:
enum TPriority
    {
    EPriorityIdle=-100,
    EPriorityLow=-20,
    EPriorityStandard=0,
    EPriorityUserInput=10,
    EPriorityHigh=20,
    };
public:
    IMPORT_C ~CActive();
    IMPORT_C void Cancel();
    IMPORT_C void Deque();
    IMPORT_C void SetPriority(TInt aPriority);
    inline TBool IsActive() const;
    inline TBool IsAdded() const;
    inline TInt Priority() const;
protected:
    IMPORT_C CActive(TInt aPriority);
    IMPORT_C void SetActive();
// Pure virtual
    virtual void DoCancel() =0;
    virtual void RunL() =0;
    IMPORT_C virtual TInt RunError(TInt aError);
public:
    TRequestStatus iStatus;
private:
    TBool iActive;
    TPriQueLink iLink;
    friend class CActiveScheduler;
    friend class CServer;
    };
```

Code fragment 3: Definition of CActive class

Code fragment 4 shows what a simple class derived from *CActive* might look like.

```
class CCompleteAfter5Seconds : public CActive
    {
public:
    static CCompleteAfter5Seconds* NewL();
    ~CCompleteAfter5Seconds();
    void StartTimer();

private:
    // constructors
    CCompleteAfter5Seconds();
    void ConstructL();

    // from CActive
    void RunL();
    void DoCancel();

private:
    // data
    RTimer iTimer;
    };
```

Code fragment 4: Example definition of a simple active object [Harrison, 2003]

Consider now the implementation of this simple active object. Code fragment 5 shows the implementation of the functions that handle the construction and destruction of the object.

```
CCompleteAfter5Seconds* CCompleteAfter5Seconds::NewL()
    {
    CCompleteAfter5Seconds* self = new(ELeave) CCompleteAfter5Seconds;
    CleanupStack::Push(self);
    self->ConstructL();
    CleanupStack::Pop(self);
    return self;
    }

CCompleteAfter5Seconds::CCompleteAfter5Seconds()
        : CActive(0)
    {
    CActiveScheduler::Add(this);
    }

void CCompleteAfter5Seconds::ConstructL()
    {
    User::LeaveIfError(iTimer.CreateLocal());
    }

CCompleteAfter5Seconds::~CCompleteAfter5Seconds()
    {
    Cancel();
    iTimer.Close();
    }
```

Code fragment 5: Functions handling the construction and destruction of an example active object [Harrison, 2003]

NewL() is a static function that follows the standard two-phased construction pattern of Symbian OS.

Every active object must implement a C++ constructor, which calls the constructor of *CActive* to set the priority of the object. The priority may be either positive or negative, but if there is no good reason to set it otherwise, it should be 0. If the active scheduler has several completed requests to handle at the same time, it uses the priority to decide which request to handle first. The requests with highest priorities are always handled first. After construction the priority can be accessed with *Priority()* and *SetPriority()* functions. The constructor also adds the object to the active scheduler, so that it can be included in the event handling. [Harrison, 2003]

The second phase constructor *ConstructL()* creates a timer object, which is accessed through *RTimer* handle. The timer will be the asynchronous service provider for this active object. [Harrison, 2003]

The destructor cancels any outstanding request by calling the *Cancel()* function defined in base *CActive* class. *Cancel()* checks whether there is an outstanding request, and if there is, it calls *DoCancel()*, which is a pure virtual member function of *CActive* and must be implemented by the derived class. *DoCancel()* must call the cancel function of the asynchronous service provider. The destructor also closes the handle to the timer, which causes the timer object to be destroyed. The destructor of *CActive* base class removes the active object from active scheduler. [Harrison, 2003]

```
void CCompleteAfter5Seconds::StartTimer()
    {
    __ASSERT_ALWAYS(!IsActive(),
                    User::Panic("CCompleteAfter5Seconds", 1));
    iTimer.After(iStatus, 5000000);
    SetActive();
    }

void CCompleteAfter5Seconds::RunL()
    {
    // 5 seconds has passed
    }

void CCompleteAfter5Seconds::DoCancel()
    {
    iTimer.Cancel();
    }
```

Code fragment 6: Implementation of the functions that handle the request

Code fragment 6 shows the functions that handle the asynchronous request. In *StartTimer()* function the asynchronous request is issued. The *iStatus* member function is defined in *CActive* base class and it contains the *TRequestStatus* for the asynchronous function call. But before issuing the asynchronous request, we must check that there isn't already an outstanding request. A single active object can handle only one outstanding request at a time. [Harrison, 2003] Finally, *SetActive()* is called to set the active flag of the active object. The flag tells the active scheduler that this active object has an outstanding request, and needs to be included in the search for the right object when a request completes. [Tasker, 1999]

When the request issued in *StartTimer()* completes the active scheduler will clear the active flag of the active object and call its *RunL()* function. In this example the active object doesn't actually do anything, but the *RunL()* can be very elaborate. After all, most code running in Symbian OS runs inside a *RunL()*

function. *RunL()* can also issue a new request. If in this example *RunL()* called *StartTimer()* again, this active object would stay active until it is cancelled, and the *RunL()* would be called every five seconds. [Tasker, 1999, Harrison, 2003]

Finally, we have *DoCancel()* function, which cancels an outstanding request. This is called by *Cancel()* function, which is defined in *CActive* base class. It checks if there is an outstanding request, so there is no need to check the status in this function. Also if *Cancel()* calls *DoCancel()* to cause the outstanding request to complete, it also calls *User::WaitForRequest()* and clears the active flag so that the active scheduler doesn't handle the completion of this request. That means that the *RunL()* doesn't get called if the request is explicitly cancelled. [Harrison, 2003]

In *CActive* there is also an error handling function called *RunError()*. Our example object is so simple that there is no need to implement *RunError()*, because the default implementation from *CActive* suffices. Actually in this object the *RunError()* will never be called. The active scheduler calls *RunError()* if *RunL()* leaves. If *RunError()* handles the error situation, it must return *KErrNone*. Otherwise it must return some error code, and the error situation is handled in the active scheduler's *Error()* function. The default implementation of *RunError()* just returns the leave code from *RunL()*. [Harrison, 2003]

The final functions in *CActive* that have not yet been mentioned are *IsAdded()* and *Deque()*. *IsAdded()* just tells if the active object has been added to the active scheduler. *Deque()* removes the active object from active scheduler and it is called by *~CActive()*. There is usually never a reason to call this function elsewhere. [Harrison, 2003]

4.3 CActiveScheduler

The active scheduler implements the active object handling loop described in Subsection 4.1.2. It maintains a doubly linked list of all the active objects added to it, ordered by priority. [Stichbury, 2005] The active scheduler is implemented in class *CActiveScheduler*, and it is defined in e32base.h. The definition is shown in Code fragment 7.

```
class CActiveScheduler : public CBase
    {
```

```
public:
    IMPORT_C CActiveScheduler();
    IMPORT_C ~CActiveScheduler();
    IMPORT_C static void Install(CActivescheduler* aScheduler);
    IMPORT_C static CActiveScheduler* Current();
    IMPORT_C static void Add(CActive* anActive);
    IMPORT_C static void Start();
    IMPORT_C static void Stop();
    IMPORT_C virtual void WaitForAnyRequest();
    IMPORT_C virtual void Error (TInt anError) const;
protected:
    inline TInt Level() const;
private:
    TInt iLevel;
    TPriQue<CActive> iActiveQ;
    };
```

Code fragment 7: The definition of the CActiveScheduler class

For all applications using the CONE framework to implement the GUI an active scheduler is already provided by the framework. However, if the thread has no active scheduler, it must be created and started before active objects can be used. *CActiveScheduler* is a concrete class that can be used as is, but it also has two virtual functions that can be redefined in derived classes. [Harrison, 2003]

Install() sets a pointer to an active scheduler in a privileged thread-local storage (TLS) location that is very fast to access. After installing the active scheduler it can be accessed with *CActiveScheduler::Current(). Add()* adds an active object to the scheduler. [Harrison, 2003]

Start() starts the active scheduler. There should be at least one outstanding request on an active object before calling *Start()*, otherwise the thread will hang. *Start()* includes the central wait loop of the active scheduler, and it will not return until *CActiveScheduler::Stop()* is called. *Stop()* causes *Start()* to return after currently active *RunL()* or *Error()* has completed. *Start()* and *Stop()* functions can be nested, and this is used by the UI framework in modal dialogs. However, that can easily lead to messy code, and should be avoided when possible. [Harrison, 2003]

WaitForAnyRequest() just calls *User::WaitForAnyRequest()*, but it can be overridden in derived classes to perform some special processing before waiting for a request. The overriding function must call *User::WaitForAnyRequest()*. [Harrison, 2003]

Error() handles all errors that have not been handled by active object itself in *CActive::RunError()*. The standard implementation in *CActiveScheduler()* does nothing, but derived classes can implement some error handling there. The active scheduler in CONE displays a natural-language version of the error code. [Harrison, 2003]

5. The test programs

To analyze the differences between threads and active objects a small test program is written using both threads and active objects. The program implements a solution to the producer/consumer problem. There are three versions of the solution. First, a thread-based solution is implemented using semaphores as first described by Dijkstra [1965a]. Then two different solutions using active objects are presented.

In the producer/consumer problem two groups of objects use the same shared resource to communicate with each other. The producers create information, which is used by the consumers. The producers create the data one piece at a time. When producer creates a piece of information, it is put into a shared buffer to wait for a consumer to process it. The consumers take the information from the buffer one piece at a time, and process it in some way.

The buffer that is used for communication between the producers and consumers has a limited size. When the buffer is full, the producers must not try to add more pieces of information to it. When the buffer is empty, the consumers must wait for a producer to insert a piece of information to the buffer.

There are many ways to implement such a buffer. One way is to use *cyclic* or *circular buffer*. A cyclic buffer has cursors pointing the location of the first empty space and the location of the next data item to be taken out of the buffer. When data is added to the buffer or removed from it, the corresponding cursor is moved to next location in the buffer, or if the cursor was in the last location, it is moved to the first item. [Bacon and Harris, 2003]

It is important that only one object accesses the shared buffer at a time. A consumer must not try to take a piece of information while a producer is in the middle of adding it to the buffer. Similarly, two producers can’t try to add information at the same time to the buffer, and two consumers must not try to consume the same piece of information.

A simple version of the producer/consumer problem has one producer and one consumer. In the general version of the problem there can be any number of producers and consumers.

5.1 Implementation of thread-based solution using semaphores

The thread-based implementation follows the solution given by Bacon and Harris [2003]. The solution is described in Figure 2.

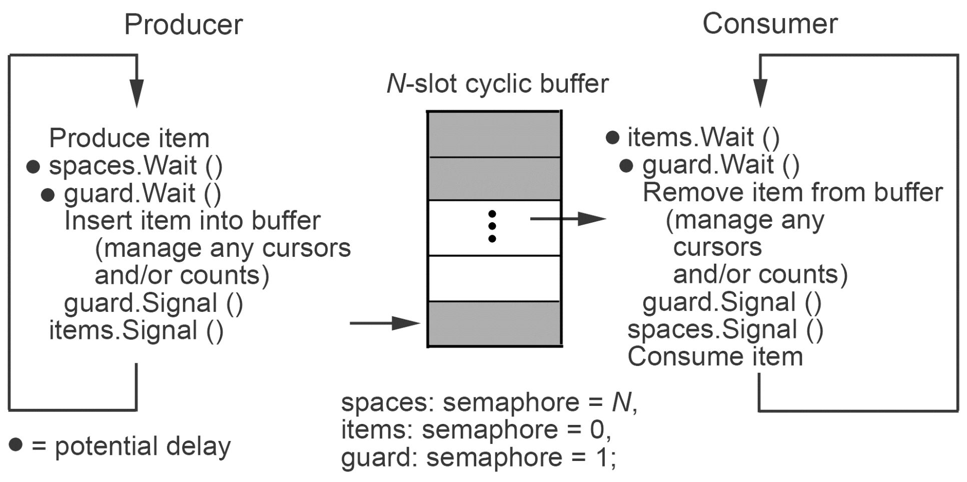

Figure 2: Solution to producer/consumer problem with semaphores [Bacon and Harris, 2003]

This solution uses three semaphores to control the execution of the producers and consumers. First we have a mutex (*iGuard)*, which allows only one object to access the buffer at a time. Another semaphore (*iItems)* controls the consumers, so that the consumers don't try to process information when the buffer is empty. The value of that semaphore is equal to the value of items in the buffer. Finally, a third semaphore (*iSpaces*) controls the producers, so that they don't try to add anything to the buffer when the buffer is full. The value of that semaphore is equal to the empty spaces in the buffer.

In the Symbian implementation of this solution we have a controller class that creates all the threads and owns the circular buffer that is used for communication between threads. Code fragment 8 contains the abbreviated class definition of the controller class *CController*.

```
class CController
    {
    public:
        CController(TInt aNumProducers, TInt aNumConsumers);
        ~CController();
        void ConstructL();
        void Start();
        void Stop();
    public: // data
        CBuffer *iBuffer;
        RSemaphore iSpaces;
        RSemaphore iItems;
    private:
        RArray<RThread> iProducers;
        RArray<RThread> iConsumers;
        TInt iNumProducers;
        TInt iNumConsumers;
    };
```

Code fragment 8: The abbreviated definition of CController class

The semaphore *iGuard* is not defined here. Instead, it is owned by class *CBuffer*. Symbian OS provides a circular buffer class *CCirBuf*. However, that class is not thread safe. In other words, if two threads try to add or remove objects from a *CCirBuf* object at the same time, data corruption may occur. *CBuffer* class encapsulated the *CCirBuf* and protects its access function by a mutex. Code fragment 9 contains the definition of *CBuffer* class.

```
class CBuffer
    {
    public:
        CBuffer();
        ~CBuffer();
        void ConstructL();
        void AddInfo(const TData *aData);
        void GetInfo(TData *aData);

    private:
        RMutex iGuard;
        CCirBuf<TData>* iBuffer;
    };
```

Code fragment 9: The definition of CBuffer class

Code fragment 10 shows the implementation of the access functions. The access to *CCirBuf* is protected by mutex *iGuard*. This means that only one thread can access the buffer at a time. This encapsulation makes the circular buffer thread-safe.

```
void CBuffer::AddInfo(const TData *aData)
    {
    iGuard.Wait();
    iBuffer->Add(aData);
    iGuard.Signal();
    }

void CBuffer::GetInfo(TData *aData)
    {
    iGuard.Wait();
    iBuffer->Remove(aData);
    iGuard.Signal();
    }
```

Code fragment 10: The implementation of access functions of CBuffer class

Code fragment 11 contains the producer thread's main function. The thread gets a pointer to the controller class as its start parameter. The main loop of the thread looks very much like the pseudo code in Figure 2. First we create a data item. If the buffer is full, semaphore *iSpaces* has value 0. In that case the thread stops at the *iSpaces.Wait(),* and continues when a consumer thread signals that semaphore. The semaphore is signaled when a consumer has removed an item from buffer, thus creating an empty space for a new item.

```
TInt ProducerMain(TAny* aPtr)
    {
    CController *controller = (CController*) aPtr;
    FOREVER
        {
        TData data;
        ProduceDataItem(data);
        controller->iSpaces.Wait();
        controller->iBuffer->AddInfo(&data);
        controller->iItems.Signal();
        }
    return KErrNone;
    }
```

Code fragment 11: The implementation of producer thread's main function

Code fragment 12 contains the consumer thread's main function. This also closely resembles the pseudo code in Figure 2. The consumer also gets a pointer to the controller object as a start parameter. If *iItems* has value 0 it means that there are no items in the buffer. In that case the consumer thread waits for a signal from a producer thread. The producer thread signals the *iItems* when it has added an object to the buffer.

```
TInt ConsumerMain(TAny* aPtr)
    {
    CController *controller = (CController*) aPtr;
    FOREVER
        {
        TData data;
        controller->iItems.Wait();
        controller->iBuffer->GetInfo(&data);
        controller->iSpaces.Signal();
        ConsumeDataItem(data);
        }
    return KErrNone;
    }
```

Code fragment 12: The implementation of consumer thread's main function

The controller class *CController*'s constructor functions *CController()* and *ConstructL()* create the semaphores and the producer and consumer threads. The implementation details of those functions and the rest of the program are not interesting. The full source code of this test program is available in Appendix A.

5.2 Solution using active objects, first version

In Chapter 4 we saw that active objects encapsulate an asynchronous request. However, active objects can also be used to implement simple cooperative multitasking without any asynchronous service providers. If we activate an active object and immediately complete the request, *RunL()* will be run as soon as possible by the active scheduler. In this first attempt to solve the producer/consumer problem using active objects we try to create a cooperatively scheduled system of active objects, where an unlimited number of producers and consumers communicate using a limited buffer. In this version of the solution we try to make active objects to behave as much as possible like independently running threads. We will run into some problems with the scheduling of the active objects.

Since all active objects run inside a single thread, we don't need semaphores or mutexes to control the access to a shared object. We know exactly when the control leaves an active object, so we can be sure that all add and remove operation to the buffer are singular. Thus, we don't need to encapsulate the buffer *CBuffer* class. Instead, we can use the *CCirBuf* class directly.

Similarly, we don't need semaphores *iSpaces* and *iItems* to control the access to buffer. Since we know that another object can't access the buffer before our object is ready, the producer can simply ask the buffer if it is empty or not before adding an item to the buffer. Other objects cannot add items to the buffer between the producer asking if there is space and adding its item to it.

Let's start by examining the code for the producer active object, *CProducer*. Code fragment 13 contains the class definition of *CProducer*. Like the producer thread in the thread-based solution, *CProducer* class also gets a pointer to the controller class when it is created.

```
class CProducer : public CActive
    {
    public:
        CProducer(CController* aController);
        ~CProducer();
        void IssueRequest();
        void CreateNewData();
        void Start();
    private: // from CActive
        virtual void DoCancel();
        virtual void RunL();
    private:
        CController* iController;
        TData iData;
};
```

Code fragment 13: The definition of the CProducer class

The constructor just adds the newly created active object to the active scheduler. *Start()* creates the first data item and then calls *IssueRequest()*. The implementation of *IssueRequest()* can be seen in Code fragment 14. This function together with *RunL()* form the heart of program logic of the producer class. First the function checks if there is space for a new item in the buffer. If there is space, the data item is added to the buffer and the request is completed with error code *KErrNone*. If the buffer is full, the request is completed with error code *KErrNotReady*.

```
void CProducer::IssueRequest()
    {
    TInt completion;
    if (iController->iBuffer->Count()<iController->iBuffer->Length())
        {
        iController->iBuffer->Add(&iData);
        completion = KErrNone;
        }
    else
        {
        completion = KErrNotReady;
        }
    SetActive();
    TRequestStatus* status = &iStatus;
    User::RequestComplete(status, completion);
    }
```

Code fragment 14: The implementation of producer active object's IssueRequest function

Code fragment 15 shows the implementation of producer's *RunL()* function. From the request's error code we can see if the data was successfully added to the buffer. If so, we create a new data item; otherwise we try to add the same item again. In both cases we immediately create a new request.

```
void CProducer::RunL()
    {
    if (iStatus == KErrNone)
        {
        CreateNewData();
        }
    IssueRequest();
}
```

Code fragment 15: The implementation of the producer active object's RunL function

The implementation of consumer object is analogous to the producer. Code fragment 16 shows the implementations of *IssueRequest()* and *RunL()* functions. Here we check if the there are any items in the buffer. If so, we remove one from the buffer and consume it. In *RunL()* we just create a new request. Again an error code is used to indicate whether an item was successfully consumed, although in this case the code is not used anywhere. Regardless of whether the previous try to consume an item was successful, we simply try again as soon as the request completes.

```
void CConsumer::IssueRequest()
    {
    TData data;
    TInt completion;
    if (iController->iBuffer->Count() > 0)
        {
        iController->iBuffer->Remove(&data);
        ProcessData();
        completion = KErrNone;
        }
    else
        {
        completion = KErrNotReady;
        }
    SetActive();
    TRequestStatus* status = &iStatus;
    User::RequestComplete(status, completion);
    }

void CConsumer::RunL()
    {
```

```
    IssueRequest();
    }
```

Code fragment 16: The implementation of the consumer active object's IssueRequest and RunL functions

The controller class creates all the active objects and calls *Start()* function in each of them. Also, just like in the thread-based solution, it owns the circular buffer that is used for communication between producers and consumers. Finally, when all the active objects have been started, the active scheduler is started. The call to *CActiveScheduler::Start()* never returns, unless one of the active objects calls *CActiveScheduler::Stop()*.

When we try to execute this program, we quickly notice that the system is not working as intended. Debugging the code reveals that only one active object's *RunL()* function is being called repeatedly. This happens because of the way the active scheduler schedules the completion of the active objects. If there are multiple active objects ready to be called, the active scheduler chooses the one with the highest priority. However, the order of the completion of objects with same priority depends on the implementation details of the active scheduler. The active scheduler maintains a doubly linked list of all its active objects, and the place of the active object is defined when the object is added to the scheduler. [Stichbury, 2005] In our unsophisticated implementation all active objects are always ready to be executed, and all the objects have the same priority. Thus, when the active scheduler starts to look for the next eligible active object to be executed, it searches through the linked list in the priority order, and always finds the same object.

We can circumvent this problem by juggling the priorities of our active objects, thus handling the scheduling of the objects ourselves. If we start each active object with the same priority, and each time an object completes its request we lower its priority by one, we make sure that each object gets a chance to complete its request one after each other. In this way we create a cooperative round robin scheduling for our objects. We also have to make sure that the priority variable doesn't roll over, so we have to add a maintenance function to the *CController*, which raises every object's priority if the priority falls too low.

The full source code of this version of the solution is in Appendix B.

The fact that we have to constantly adjust the priorities of the objects should serve as a strong hint that this is not the proper way to use active objects. This is also an inefficient solution because the active objects are constantly run regardless of whether the producers can add items to the buffer or whether there are any items for the consumer to consume. This is different from the thread-based solution, which uses semaphores to suspend the thread when it can't access the buffer.

However, this kind of solution, where the active object issues a request and immediately completes it by itself, is useful in some situations. One example is a long-running maintenance task that can be completed in small increments. Such a task can be implemented using a low-priority active object. When the maintenance task is started, the active object is activated and the request is immediately completed. The active object's *RunL()* function is called as soon as there are no higher priority active objects to handle. At the *RunL()*, the active object performs one increment of the task and if the task is not completed, issues and completes a new request. This way the task is being run only when the thread would otherwise be idle. [Harrison, 2003]

5.3 Improved solution using active objects

The solution to the producer/consumer problem presented in the previous section shows that active objects can be used to create a system that closely resembles cooperatively scheduled threads. However, that solution was clearly not optimal, and we run into some scheduling problems with it.

In this section a different solution is examined. In the previous solution the active objects' asynchronous requests were always completed immediately, regardless of whether the request was successful or not. This time we reintroduce the *CBuffer* class as a wrapper to the circular buffer, but now we make the class into an asynchronous service provider.

The operations to add and remove items from the buffer are made into asynchronous operations. This means that the operation is completed only when the item is successfully added or removed from the buffer, or the active object cancels its request. In the previous solution, when the buffer was full, the producers just kept trying again and again to add a new item to the buffer.

When the add operation is turned into an asynchronous operation, if the producer tries to add an item to the buffer and the buffer is full, the producer active object is suspended until there is a free space in the buffer. Similarly, the consumers that are trying to consume items when the buffer is empty are suspended until there are items to consume.

Let's examine the implementation of the new *CBuffer* class. Code fragment 17 contains the class definition. Here we see the new *Add()* and *Remove()* operations, which take a reference to *TRequestStatus* as a parameter. That indicates that these operations are asynchronous requests. And since the class provides asynchronous services, it also needs to have a cancel function. Since the class handles several requests at the same time, *Cancel()* also takes a reference to *TRequestStatus* as a parameter. This is used to identify the request and the active object that is being cancelled.

The class also needs to keep track of all the outstanding requests. For that reason two new circular buffers are introduced. One of them contains all outstanding producer requests and the other contains all outstanding consumer requests. For each outstanding request a pointer to its request status and a pointer to the data item are saved.

```
class CBuffer
    {
    public:
        CBuffer();
        ~CBuffer();
        void ConstructL(TInt aNumProducers, TInt aNumConsumers);
        void Add(TData* aData, TRequestStatus& aRS);
        void Remove(TData* aData, TRequestStatus& aRS);
        void Cancel(TRequestStatus& aRS);
    private:
        void CompleteAddReq(TData* aData, TRequestStatus* aRS);
        void CompleteRemoveReq(TData* aData, TRequestStatus* aRS);

        struct TRequestData
            {
            TRequestStatus* iRS;
            TData* iData;
            };
        CCirBuf<TData>* iBuffer;
        CCirBuf<TRequestData> *iProducerRequests;
        CCirBuf<TRequestData> *iConsumerRequests;
    };
```

Code fragment 17: The definition of CBuffer class

Next, let's have a look at the implementation of *Add()*. The implementation is shown in Code fragment 18. First, the status of the request is changed to *KRequestPending*. As long as the status is this, the active object that issued the request stays suspended. If the buffer is not full, the request can be completed immediately. If the buffer is full, the active object's request status and the data item are stored into a circular buffer containing all outstanding producer requests. In that case the request's status stays at *KRequestPending*, and the active object is suspended until the request can be completed.

```
void CBuffer::Add(TData* aData, TRequestStatus& aRS)
    {
    aRS = KRequestPending;
    if (iBuffer->Count() < iBuffer->Length())
        {
        CompleteAddReq(aData, &aRS);
        }
    else
        {
        TRequestData rd;
        rd.iData = aData;
        rd.iRS = &aRS;
        iProducerRequests->Add(&rd);
        }
    }
```

Code fragment 18: The implementation of CBuffer::Add function

The remove operation works similarly to the add operation. If there are items in the buffer, the operation is completed immediately. And if the buffer is empty, a pointer to the consumer object data and the request status are stored into another circular buffer, which contains all outstanding consumer requests.

The completion of add operation can be seen in Code fragment 19. First, the data is added to the circular buffer and the active object's request is completed with status *KErrNone*. Since there now is at least one item in the buffer, we'll check if there are any outstanding consume requests. If there are, one of those is completed.

```
void CBuffer::CompleteAddReq(TData* aData, TRequestStatus* aRS)
    {
    iBuffer->Add(aData);
    User::RequestComplete(aRS, KErrNone);

    TRequestData rd;
    if (iConsumerRequests->Remove(&rd))
        {
        CompleteRemoveReq(rd.iData, rd.iRS);
        }
    }
```

Code fragment 19: The implementation of CBuffer::CompleteAddReq function

Completing remove requests again works in a similar way to the add requests. First, an item is removed from the buffer, and if there are any outstanding add requests, one of those is completed.

Since *CBuffer* acts as an asynchronous service provider, it must also provide a cancel function for the requests. Code fragment 20 shows the implementation of the cancel function. Since *CBuffer* handles several asynchronous requests simultaneously, the cancel function needs the request status as a parameter. The status is used to identify the request to be cancelled. The function searches through all the outstanding requests for the one to be cancelled, and when it finds the correct request, it calls *User::RequestComplete()* for that request with completion code *KErrCancel*. When the active object is cancelled using its *Cancel()* function, *RunL()* is not called. That means that it is not necessary to handle *KErrCancel* completion code as a special case in the function.

```
void CBuffer::Cancel(TRequestStatus& aRS)
    {
    TInt reqs = iProducerRequests->Count();
    while (reqs)
        {
        TRequestData rd;
        iProducerRequests->Remove(&rd);
        if (rd.iRS != &aRS)
            iProducerRequests->Add(&rd);
        else
            {
            User::RequestComplete(rd.iRS, KErrCancel);
            return;
            }
        reqs--;
        }

    reqs = iConsumerRequests->Count();
    while (reqs)
        {
        TRequestData rd;
        iConsumerRequests->Remove(&rd);
        if (rd.iRS != &aRS)
            iConsumerRequests->Add(&rd);
        else
            {
            User::RequestComplete(rd.iRS, KErrCancel);
            return;
            }
        reqs--;
        }
    }
```

Code fragment 20: Implementation of CBuffer::Cancel function

In this solution the producer and consumer active objects are simpler than in the previous case. Previously all the program logic was contained in the active object, but this time the *CBuffer* class handles most of the logic. The producer doesn't have to bother if the buffer is full or not. It simply issues the request, and when the request completes, the item has been added to the buffer. The producer object's *IssueRequest()* and *RunL()* functions are shown in Code fragment 21.

The full source code of this solution is available in appendix C.

```
void CProducer::IssueRequest()
    {
    iController->iBuffer->Add(&iData, iStatus);
    SetActive();
    }

void CProducer::RunL()
    {
    if (iStatus == KErrNone)
        {
        CreateNewData();
        IssueRequest();
        }
    }
```

Code fragment 21: The implementation of producer active object's IssueRequest and RunL functions

At least in theory this solution should me more efficient that the previous one. This time the active object are not completed until the request is successfully completed, and the while the request is in pending state, the active object is suspended.

In the next chapter we run some tests for all the three solutions and see how they perform with different amounts of producers and consumers.

6. Tests and analysis of the programs

In this chapter the three test programs are compared, and some tests are run for all of them. All the tests are run on Nokia 9300i mobile phone, which runs uses version 7.0 of the Symbian OS. For the timings, the internal clock of the Symbian OS is used. In this version of the operating system the clock offers 64Hz accuracy, which means that the smallest observable increment is 0.0156 seconds [Harrison, 2003].

6.1 Performance

There are many variables that can affect the performance of a program on a modern mobile phone. An effort is made to eliminate as many of those as possible. The setup for the tests is as follows:

- The phone is formatted, to make sure there are no installed applications running in the background.
- The phone is booted up, but the mobile phone functionality is not switched on.
- After booting the phone, it is left alone for several minutes to let it complete all the background activities related to booting up.
- The phone battery is fully charged up, but while the tests are run, the charger is not connected.
- The test applications are copied to a MMC card from computer, and the card is inserted to the phone.
- The test applications are run directly from the MMC card by opening File Manager application on the phone and navigating to the MMC directory. The test applications are run by opening the application exes from File Manager.

- Since it is impossible to control all the background activity on the test phone, all the tests are run at least five times and the average value of the results is used.

There are also several factors intrinsic to the test programs that might affect the results of the tests. The tests try to analyze the effects of those factors. For instance, how does the amount of the threads or active objects affect the results? Does the increase in the amount of the producers and consumers cause overhead that slows down the application? The first test tries to answer that question.

In the first test the time to produce and consume 50000 items of data is measured. For all three applications the test is run with one producer and one consumer, and then the amount of producers and consumers is increased in steps of ten until we have 101 producers and 101 consumers. The actual data item creation and consuming is very simple in this application, so most of the time should be spent handling the scheduling of threads or active objects and their cooperation. The measured times also include the initialization and destruction of the threads and active objects, and also the creation of the active scheduler for the active-object-based solutions. The size of the buffer is ten items. The results of the test are presented in Figure 3.

From the test results we can clearly see that the first solution using active objects is not performing very well when the amount of active objects increases. It is quite easy to understand why this happens. The thread-based solution uses semaphores to suspend the producers when the buffer is full and the consumers when the buffer is empty. The second active-object-based solution uses the asynchronous nature of active objects for the same purpose. In the first active-object-based solution all the producers and consumers are running all the time, trying again and again to access the buffer until the operation is successful. It is also possible that the constant shuffling of the active objects' priorities causes unnecessary overhead when there are lots of active objects.

Based on these results the first active-object-based solution is discarded from the rest of the tests.

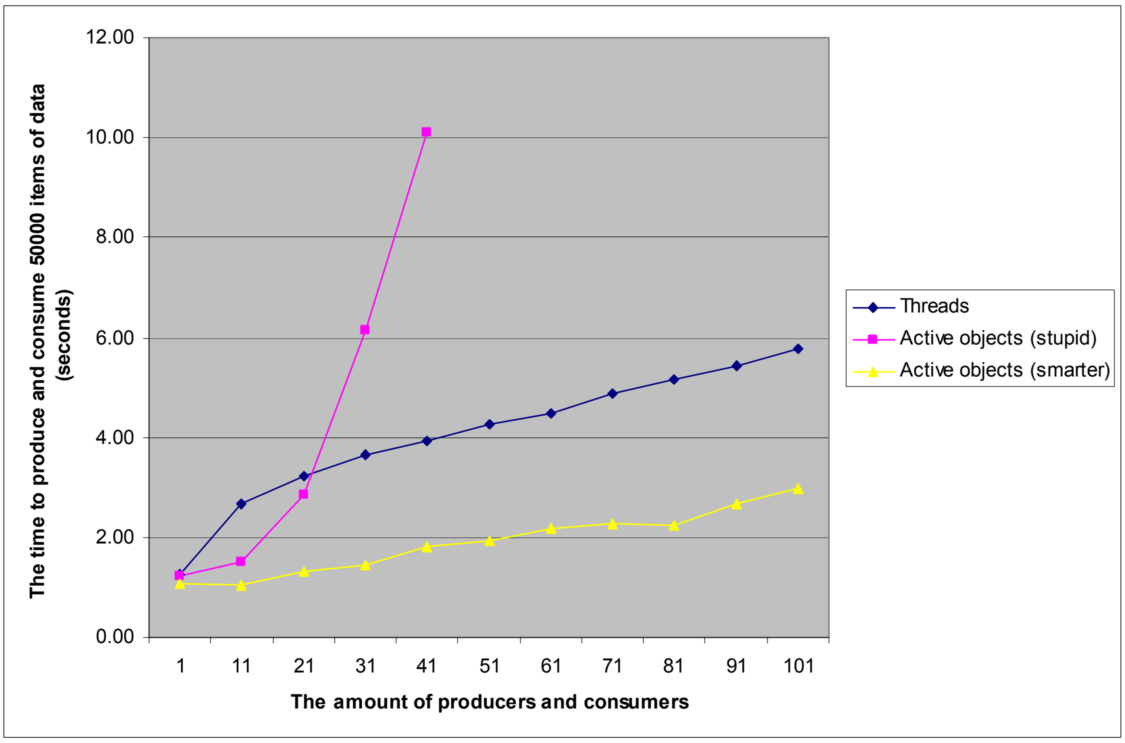

Figure 3: The time to produce 50000 items of data with different amount of producers and consumers.

Comparing the results of the thread-based solution and the second active-object based solution (from now on called simply the active-object-based solution), we can see that when there are only one producer and consumer, the results are about the same, but when the amount of threads increases, the thread-based solution takes at least twice as long time to complete the task than the active-object-based solution. For the thread-based solution there is a big leap from one producer and consumer to eleven producers and consumers, while from that point onward the increase in the amount of threads causes only more or less linear increase in the time required to complete the task.

On the other hand, the active-object-based solution doesn't exhibit this kind of jump from one to eleven producers and consumers. There the time to complete the task increases pretty much linearly as the amount of active objects increases.

Next, let's have a closer look at the thread-based and the active-object-based solutions. The first test included both the time to initialize the threads and active objects and the time to actual time to produce and consume the data items. In the next test the time spent in initialization is separated from the rest of the test. The amount of items to be created is also increased from 50000 to 200000 items.

Figure 4 shows the time taken for the initialization for different amounts of threads and active objects. For the thread-based solution this time includes the creation of the threads, starting the threads, and after the required amount of data has been produced and consumed the killing of all the threads. For the active-object-based solution the time includes the creation of all the active objects, the creation of the active scheduler, activating all the active objects, and when the required amount of data has been consumed, the destroying of the active objects and cancellation of the outstanding asynchronous requests.

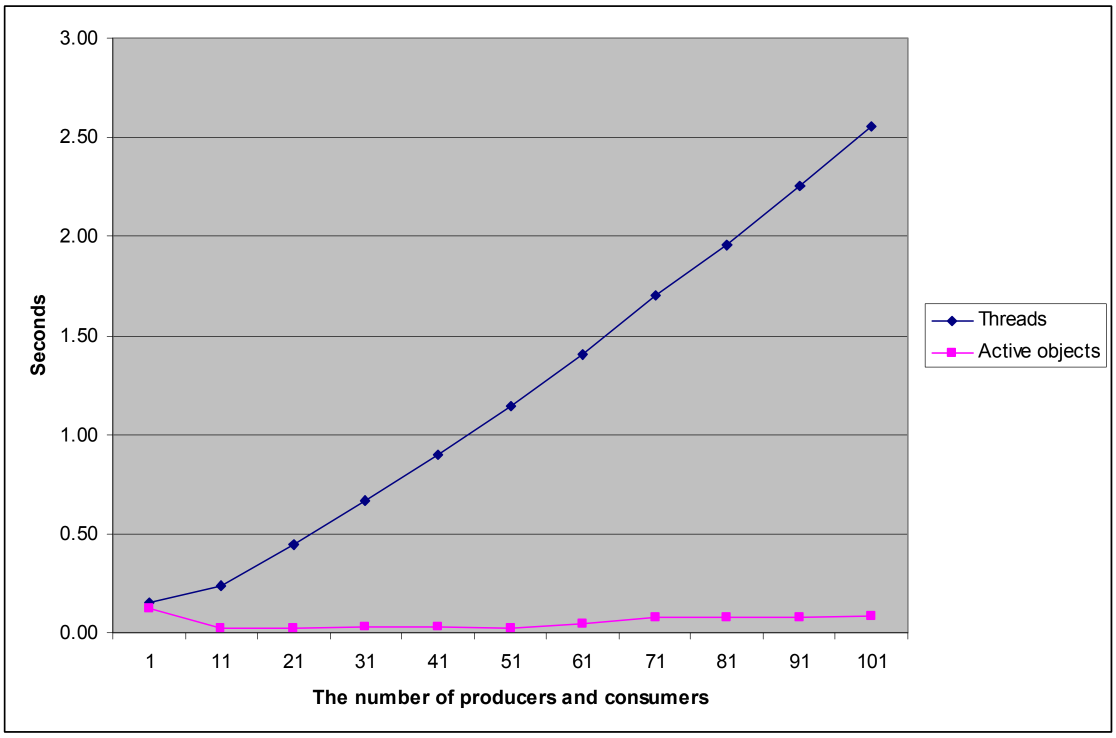

Figure 4: The time taken to initialize and destroy the threads or active objects

The results show that the thread-based solution takes much more time to initialize its threads than the active-object-based solution takes to initialize itself. The time taken to initialize 202 threads is about 2.5 seconds, while the same amount of active objects is created in about 0.1 seconds. The difference is significant. The time taken for initialization increases linearly as the number of threads or active objects increases, which is logical since there are simply more objects to create.

Figure 5 shows the time taken for the test programs to actually produce and consume 200000 items of data. For the thread-based solution the timing starts from the moment all threads have been started and ends when the required amount of data has been consumed. For the active-object-based solution the timing starts when the active scheduler has been started and ends when the required amount of data has been consumed.

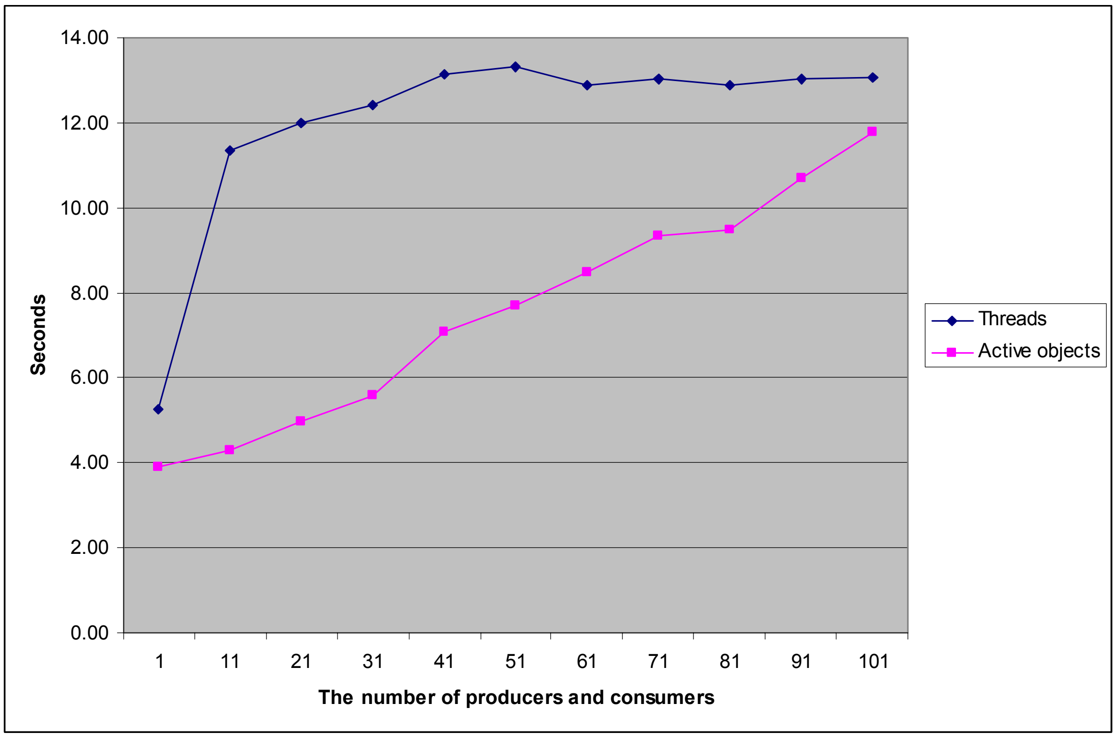

Figure 5: The time taken to produce and consume 200000 items of data, not including the initialization.

From these results we can see that for these amounts of producers and consumers the active-object-based solution is always faster than the thread-based solution. However, the curves are quite different, and it seems likely that if the amount of producers and consumers would still be increased, the thread-based solution would outperform the active-object-based solution. However, those amounts of threads or active objects seem quite unlikely to be reached in a program running inside a mobile phone.

When there are only one producer and one consumer, the performances of both programs are quite similar. When the amount of producers and consumers is increased, the time taken by the thread-based solution increases rapidly. With 11 producers and consumers the thread-based solution takes more than double the time it took with one producer and consumer. After that the performance starts to stabilize, until after about 40 producers and consumers the time taken doesn't increase at all when more threads are added.

The time taken by the active-object-based solution increases more or less linearly as the amount of active objects is increased. The difference in performance between the solutions is at its most striking when there are about 10 producers and consumers each. At that point the thread-based solution takes about 2.5 times as long as the active-object-based solution to consume the required amount of data items.

6.2 Memory consumption

The memory consumption of the test programs has been measured using the *RThread::GetRamSizes()* function. For the thread-based solution the function is called for all threads in the program and the values are added together. For the active-object-based solution the memory usage of the single thread is reported.

Figure 6 shows the memory consumption for different amounts of threads or active objects.

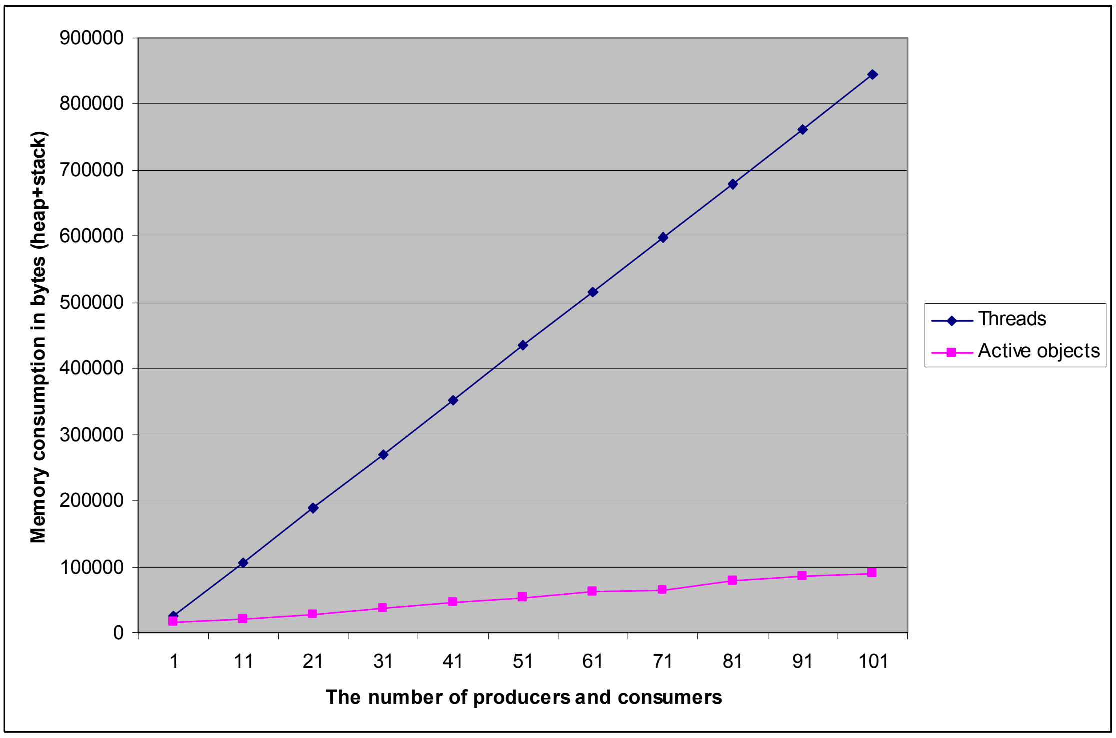

Figure 6: The memory consumption of the test programs for different amounts of threads or active objects

The results show that the thread-based solution consumes about ten times as much memory as the active-object-based solution. This is not surprising, since each thread needs its own stack. In the test program the stack size per thread is 4kB. If the stack size is halved from that, the program won't run any more.

6.3 Other considerations

The time and memory consumptions of a program are very important factors when analyzing these solutions, especially on mobile phones where these resources are much more limited than on many other platforms. However, there are also many less tangible and less easily measurable considerations that need to be taken into account when analyzing the solutions.

For software projects the ease of programming is often very important. The ease of programming is affected by both how long time it takes to implement the solution, and how error-prone the solution is. It is impossible to give any definite measurements of how easy any given solution is to implement, because it very much depends on the individual developers' experience and skill set. However, we can analyze some of the factors that affect the results.

Concurrent programming is difficult. Especially in preemptive multitasking environment all interactions between different threads of processing must be carefully analyzed. Often the access to data must be protected using synchronizing constructs such as mutexes and semaphores. One must also be careful when using those constructs to avoid the possibility of a deadlock. Since the order of execution of the threads is determined at the runtime by the scheduler, and a context switch may occur at any time, the developer must be able to analyze all possible cases.

The nonpreemptive scheduling of active objects simplifies this somewhat. With active objects the developer knows the exact locations where the execution switches from one active object to the next one. Most of the time this means that there is no need to protect the access to common data, because the developer can be sure that the active object can complete the operation before the next one accesses the same data. The developer must still be careful to avoid deadlocks, but certainly synchronizing active objects is easier than synchronizing threads.

Active objects are very widely used in Symbian OS. All applications that use the CONE GUI framework run mostly inside active objects. Since the framework provides the active scheduler, adding own active objects to the program is quite simple. However, since those active objects run in the same thread as the code handling user interaction, extra care must be taken to ensure that all active objects complete their functions relatively quickly.

If code is ported to Symbian OS from another operating system, it probably uses threads in many cases where Symbian code would use active objects. Changing the code into using active objects might not be simple. The first attempt to solve the producer/consumer problem shows that simply thinking of active objects as cooperative threads doesn’t always work. The active objects might need a different approach to the problem. In such cases it is probably easier to continue using threads also in the code ported to Symbian OS.

7. Summary and conclusions

In this thesis I have introduced the concept of active objects in Symbian OS. The practical use of active objects was examined by implementing a solution to the well-known producer/consumer problem using active objects. The solution was examined by comparing it to the more tradition thread-based solution, which uses semaphores for synchronizing the threads.

Active objects are widely used throughout the Symbian OS, and the documentation provided by Symbian strongly encourages their use over multithreading in applications written for Symbian OS. Comparing the thread-based and the active-object-based solutions supported this recommendation. The active-object based solution was more efficient and it consumed significantly less memory than the thread-based solution.

The results also highlighted a potential pitfall in porting thread-based program into using active objects. Using active objects efficiently requires understanding of their nature as an encapsulation of an asynchronous request. Just thinking of active objects as cooperatively scheduled threads may lead to an inefficient code, as was demonstrated by the first attempt to solve the producer/consumer problem with active objects.

There is still lot of room for further study of active objects in Symbian OS. This thesis only examined one specific problem, and there is no proof that these solutions are the most efficient ones to the given problem. Some of the test results may be influenced by implementation details of the test programs. More tests with different types of programming problems should be run to gain a more complete understanding of active objects and their performance in comparison to threads.

All the tests were run on version 7.0 of Symbian OS. That version of the operating system uses the EKA1 kernel architecture. Symbian OS 8.0b introduced a new kernel called EKA2, which had some changes to the way threads are handled inside the kernel. Thus, the results are only valid for older Symbian smartphones. To see how the test programs perform on latest phones using Symbian OS 8.0b or a later version, the tests need to be run also on EKA2 kernel.

References

[Bacon and Harris, 2003] Jean Bacon and Tim Harris, *Operating Systems: Concurrent and Distributed Software Design*, Addison-Wesley, 2003.

[Ben-Ari, 2006] M. Ben-Ari, *Principles of Concurrent and Distributed Programming, Second Edition*, Addison-Wesley, 2006.

[Deitel, 1984] Harvey M. Deitel, *An Introduction to Operating Systems.* Addison-Wesley, 1984.

[Dijkstra, 1965a] E. W. Dijkstra, *Cooperating Sequential Processes*. Technical Report EWD-123, Technical University, Eindhoven, Netherlands, 1965.

[Dijkstra, 1965b] E. W. Dijkstra, Solution of a problem in concurrent control. *Communications of ACM* **8**, 9 (Sept. 1965), *569*.

[Hansen, 1973] Peter Brinch Hansen, *Operating System Principles.* Prentice-Hall, 1973.

[Harrison, 2003] Richard Harrison, *Symbian OS C++ for Mobile Phones*, John Wiley & Sons Ltd, 2003.

[Mery, 2002] David Mery, Symbian white paper: Why is a different operating system needed?, Symbian Ltd, 2002. Available at http://www.symbian.com/technology/why-diff-os.html.

[Morris, 2007] Ben Morris, *The Symbian OS Architecture Sourcebook – Design and Evolution of a Mobile Phone OS*, John Wiley & Sons Ltd, 2007.

[Sales, 2005] Jane Sales, *Symbian OS Internals – Real-time Kernel Programming*, John Wiley & Sons Ltd, 2005.

[Stichbury, 2005] Jo Stichbury, *Symbian OS Explained – Effective C++ Programming For Smartphones*, John Wiley & Sons Ltd, 2005.

[Symbian] Symbian website, http://www.symbian.com.

[Symbian, 2002] *Symbian OS v7.0s Developer Library*. Available at http://www.symbian.com/Developer/techlib/v70sdocs/doc_source/index.html.

[Tanenbaum, 1987] Andrew S. Tanenbaum, *Operating Systems Design and Implementation*. Prentice-Hall, 1987.

[Tasker, 1999] Martin Tasker, Active Objects, Symbian Ltd, 1999. Available at http://www.symbian.com/developer/techlib/papers/tp_active_objects/active.htm.

[Tasker, 2000] Martin Tasker, *Professional Symbian Programming*. Wrox, 2000.

Appendix A: Full source of the thread-based solution

```
#include <e32base.h>
#include <e32cons.h>
#include <e32svr.h>

LOCAL_C void doTestsL();

_LIT(KThreadProducerName, "producer");
_LIT(KThreadConsumerName, "consumer");

const TInt KBufferSize = 10;
const TInt KEndCondition = 200000;

struct TData
    {
    public:
        TInt iDataNumber;
        TBuf<32> iMessage;
    };

class CBuffer
    {
    public:
        CBuffer();
        ~CBuffer();
        void ConstructL();
        void AddInfo(const TData *aData);
        void GetInfo(TData *aData);

    private:
        RMutex iGuard;
        CCirBuf<TData>* iBuffer;
    };

class CController
    {
    public:
        CController(TInt aNumProducers, TInt aNumConsumers);
        ~CController();
        void ConstructL();
        void Start();
        void Stop();

    public: // data
        CBuffer *iBuffer;
        RSemaphore iSpaces;
        RSemaphore iItems;
        RSemaphore iObjectsConsumed;
        RSemaphore iEndCondition;

    private:
        RArray<RThread> iProducers;
        RArray<RThread> iConsumers;
        TInt iNumProducers;
        TInt iNumConsumers;
    };

// producer thread

void ProduceDataItem(TData& aData)
    {
    aData.iDataNumber = 1;
    aData.iMessage = _L("producer is producing!");
    }

TInt ProducerMain(TAny* aPtr)
    {
    CController *controller = (CController*) aPtr;
    FOREVER
        {
        TData data;
        ProduceDataItem(data);
        controller->iSpaces.Wait();
        controller->iBuffer->AddInfo(&data);
        controller->iItems.Signal();
        }
    return KErrNone;
    }

// consumer thread

void ConsumeDataItem(TData& /*aData*/)
    {
    }

TInt ConsumerMain(TAny* aPtr)
    {
    CController *controller = (CController*) aPtr;
    FOREVER
        {
        TData data;
        controller->iItems.Wait();
        controller->iBuffer->GetInfo(&data);
        controller->iSpaces.Signal();
        ConsumeDataItem(data);
        controller->iObjectsConsumed.Signal();
        if (controller->iObjectsConsumed.Count() >= KEndCondition)
            controller->iEndCondition.Signal();
        }
    return KErrNone;
    }

// CBuffer

CBuffer::CBuffer()
    {
    iGuard.CreateLocal();
    }

CBuffer::~CBuffer()
    {
    delete iBuffer;
    iGuard.Close();
    }

void CBuffer::ConstructL()
    {
    iBuffer = new (ELeave) CCirBuf<TData>;
    iBuffer->SetLengthL(KBufferSize);
    }

void CBuffer::AddInfo(const TData *aData)
    {
    iGuard.Wait();
    iBuffer->Add(aData);
    iGuard.Signal();
    }

void CBuffer::GetInfo(TData *aData)
    {
    iGuard.Wait();
    iBuffer->Remove(aData);
    iGuard.Signal();
    }

// CController

CController::CController(TInt aNumProducers, TInt aNumConsumers)
: iNumProducers(aNumProducers), iNumConsumers(aNumConsumers)
```

```
    {
    }

CController::~CController()
    {
    // close handles to threads
    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        iProducers[i].Close();
        }

    for (i = 0; i < iNumConsumers; i++)
        {
        iConsumers[i].Close();
        }

    // delete objects and close all other handles
    delete iBuffer;
    iItems.Close();
    iSpaces.Close();
    iObjectsConsumed.Close();
    iEndCondition.Close();
    iProducers.Close();
    iConsumers.Close();
    }

void CController::ConstructL()
    {
    // create semaphores and CBuffer
    iSpaces.CreateLocal(KBufferSize);
    iItems.CreateLocal(0);
    iObjectsConsumed.CreateLocal(0);
    iEndCondition.CreateLocal(0);
    iBuffer = new (ELeave) CBuffer;
    iBuffer->ConstructL();

    // create threads
    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        TName threadName(KThreadProducerName);
        threadName.AppendFormat(_L("%d"), i+1);
        RThread t;
        TInt err = t.Create(threadName, ProducerMain, 2048, NULL, this);
        if (err) User::Panic(threadName, err);
        iProducers.Append(t);
        }
    for (i = 0; i < iNumConsumers; i++)
        {
        TName threadName(KThreadConsumerName);
        threadName.AppendFormat(_L("%d"), i+1);
        RThread t;
        TInt err = t.Create(threadName, ConsumerMain, 2048, NULL, this);
        if (err) User::Panic(threadName, err);
        iConsumers.Append(t);
        }
    }

void CController::Start()
    {
    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        iProducers[i].Resume();
        }
    for (i = 0; i < iNumConsumers; i++)
        {
        iConsumers[i].Resume();
        }
    }
```

```

void CController::Stop()
    {
    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        iProducers[i].Kill(KErrNone);
        }
    for (i = 0; i < iNumConsumers; i++)
        {
        iConsumers[i].Kill(KErrNone);
        }
    }

// run the test
LOCAL_C void doTestsL()
    {
    CConsoleBase *console =
        Console::NewL(_L("Tests"),TSize(KConsFullScreen,KConsFullScreen));
    console->Printf(_L("press any key to start.\n"));
    console->Getch(); // get and ignore character

    TInt iterations;
    for (iterations = 1; iterations <= 111; iterations += 10)
        {
        TTime startTimeIni;
        startTimeIni.UniversalTime();
        TInt numProducers = iterations;
        TInt numConsumers = iterations;
        console->Printf(_L("Running with %d prod. and %d cons. threads.\n"),
            numProducers, numConsumers);

        CController* controller =
            new (ELeave) CController(numProducers, numConsumers);
        CleanupStack::PushL(controller);
        controller->ConstructL();
        controller->Start();

        TTime startTime;
        startTime.UniversalTime();
        controller->iEndCondition.Wait();
        TTime endTime;
        endTime.UniversalTime();

        controller->Stop();

        delete controller;
        CleanupStack::Pop(controller);

        TTime endTimeIni;
        endTimeIni.UniversalTime();

        TInt fullTime =
            endTimeIni.MicroSecondsFrom(startTimeIni).Int64().GetTInt();
        TInt runTime = endTime.MicroSecondsFrom(startTime).Int64().GetTInt();
        console->Printf(_L("Completed in %d + %d microseconds.\n\n"),
            fullTime-runTime, runTime);
        }
    console->Printf(_L("THE END. press any key to exit.\n"));
    console->Getch(); // get and ignore character
    delete console;
    }

GLDEF_C TInt E32Main() // main function called by E32
    {
    __UHEAP_MARK;
    CTrapCleanup* cleanup=CTrapCleanup::New();
    TRAPD(error,doTestsL());
    __ASSERT_ALWAYS(!error,User::Panic(_L("doTestsL"),error));
    delete cleanup;
    __UHEAP_MARKEND;
```

```
        return 0;
        }
```

Appendix B: Full source of the active-object-based solution, first version

```
#include <e32base.h>
#include <e32cons.h>
#include <e32svr.h>

LOCAL_C void doTestsL();

_LIT(KThreadProducerName, "producer");
_LIT(KThreadConsumerName, "consumer");

const TInt KBufferSize = 10;
const TInt KEndCondition = 50000;

class CProducer;
class CConsumer;

struct TData
    {
    public:
        TInt iDataNumber;
        TBuf<32> iMessage;
    };

class CController
    {
    public:
        CController(TInt aNumProducers, TInt aNumConsumers);
        ~CController();
        void ConstructL();
        void ControllerMainL();
        void Start();
        void Stop();
        void ResetPriorities();

    public: // data
        CCirBuf<TData>* iBuffer;
        TInt iConsumeCount;

    private:
        RPointerArray<CProducer> iProducers;
        RPointerArray<CConsumer> iConsumers;
        TInt iNumProducers;
        TInt iNumConsumers;
    };

class CProducer : public CActive
    {
    public:
        CProducer(CController* aController, TName aName);
        void ConstructL();
        ~CProducer();
        void IssueRequest();
        void CreateNewData();
        void Start();
    private:
        virtual void DoCancel();
        virtual void RunL();
    private:
        CController* iController;
        TData iData;
        TInt iCount;
        TName iName;
    };

class CConsumer : public CActive
    {
```

```
    public:
        CConsumer(CController* aController, TName aName);
        void ConstructL();
        ~CConsumer();
        void Start();
        void IssueRequest();
        void ProcessData();
    private:
        virtual void DoCancel();
        virtual void RunL();
    private:
        CController* iController;
        TName iName;
    };

// CProducer

CProducer::CProducer(CController* aController, TName aName)
        : CActive(0),
          iController(aController),
          iName(aName)
    {
    CActiveScheduler::Add(this);
    }

void CProducer::ConstructL()
    {
    }

CProducer::~CProducer()
    {
    Cancel();
    }

void CProducer::Start()
    {
    CreateNewData();
    IssueRequest();
    }

void CProducer::CreateNewData()
    {
    iData.iDataNumber = 1;
    iData.iMessage = _L("producer is producing!");
    }

void CProducer::IssueRequest()
    {
    TInt completion;
    if (iController->iBuffer->Count() < iController->iBuffer->Length())
        {
        iController->iBuffer->Add(&iData);
        completion = KErrNone;
        }
    else
        {
        completion = KErrNotReady;
        }
    SetActive();
    TRequestStatus* status = &iStatus;
    User::RequestComplete(status, completion);
    }

void CProducer::RunL()
    {
    if (Priority() > KMinTInt)
        SetPriority(Priority() - 1);
    else
        iController->ResetPriorities();
    if (iStatus == KErrNone)
        {
```

```
        CreateNewData();
        }
    IssueRequest();
    }

void CProducer::DoCancel()
    {
    }

// CConsumer

CConsumer::CConsumer(CController* aController, TName aName)
        : CActive(0),
          iController(aController),
          iName(aName)
    {
    CActiveScheduler::Add(this);
    }

void CConsumer::ConstructL()
    {
    }

CConsumer::~CConsumer()
    {
    Cancel();
    }

void CConsumer::Start()
    {
    IssueRequest();
    }

void CConsumer::ProcessData()
    {
    iController->iConsumeCount++;
    if (iController->iConsumeCount > KEndCondition)
        iController->Stop();
    }

void CConsumer::IssueRequest()
    {
    TData data;
    TInt completion;
    if (iController->iBuffer->Count() > 0)
        {
        iController->iBuffer->Remove(&data);
        ProcessData();
        completion = KErrNone;
        }
    else
        {
        completion = KErrNotReady;
        }
    SetActive();
    TRequestStatus* status = &iStatus;
    User::RequestComplete(status, completion);
    }

void CConsumer::RunL()
    {
    if (Priority() > KMinTInt)
        SetPriority(Priority() - 1);
    else
        iController->ResetPriorities();

    IssueRequest();
    }

void CConsumer::DoCancel()
    {
    }

// CController

CController::CController(TInt aNumProducers, TInt aNumConsumers)
: iConsumeCount(0), iNumProducers(aNumProducers), iNumConsumers(aNumConsumers)
    {
    }

CController::~CController()
    {
    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        delete iProducers[i];
        }

    for (i = 0; i < iNumConsumers; i++)
        {
        delete iConsumers[i];
        }
    iProducers.Close();
    iConsumers.Close();
    delete iBuffer;
    }

void CController::ConstructL()
    {
    iBuffer = new (ELeave) CCirBuf<TData>;
    iBuffer->SetLengthL(KBufferSize);

    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        TName name(KThreadProducerName);
        name.AppendFormat(_L("%d"), i+1);
        CProducer* p = new (ELeave) CProducer(this, name);
        iProducers.Append(p);
        }

    for (i = 0; i < iNumConsumers; i++)
        {
        TName name(KThreadConsumerName);
        name.AppendFormat(_L("%d"), i+1);
        CConsumer* c = new (ELeave) CConsumer(this, name);
        iConsumers.Append(c);
        }
    }

void CController::Start()
    {
    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        iProducers[i]->Start();
        }

    for (i = 0; i < iNumConsumers; i++)
        {
        iConsumers[i]->Start();
        }
    }

void CController::Stop()
    {
    CActiveScheduler::Stop();
    }

void CController::ResetPriorities()
    {
    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        iProducers[i]->SetPriority(0);
        }

    for (i = 0; i < iNumConsumers; i++)
        {
        iConsumers[i]->SetPriority(0);
        }
    }

// run the test
LOCAL_C void doTestsL()
    {
    CConsoleBase *console =
        Console::NewL(_L("Tests"),TSize(KConsFullScreen,KConsFullScreen));
    console->Printf(_L("press any key to start.\n"));
    console->Getch(); // get and ignore character

    TInt iterations;
    for (iterations = 1; iterations <= 111; iterations += 10)
        {
        TTime startTime;
        startTime.UniversalTime();
        TInt numProducers = iterations;
        TInt numConsumers = iterations;
        console->Printf(_L("Running with %d prod. and %d cons. threads.\n"),
            numProducers, numConsumers);

        CActiveScheduler* activeScheduler = new (ELeave) CActiveScheduler;
        CleanupStack::PushL(activeScheduler) ;
        CActiveScheduler::Install(activeScheduler);

        CController* controller =
            new (ELeave) CController(numProducers, numConsumers);
        CleanupStack::PushL(controller);
        controller->ConstructL();
        controller->Start();

        // Start handling requests
        CActiveScheduler::Start();

        delete controller;
        CleanupStack::Pop(controller);
        CleanupStack::PopAndDestroy(activeScheduler);
        TTime endTime;
        endTime.UniversalTime();

        console->Printf(_L("%d objects consumed in %d microseconds.\n\n"),
            KEndCondition, endTime.MicroSecondsFrom(startTime));
        }
    console->Printf(_L("THE END. press any key to exit.\n"));
    console->Getch(); // get and ignore character
    delete console;
    }

GLDEF_C TInt E32Main() // main function called by E32
    {
    __UHEAP_MARK;
    CTrapCleanup* cleanup=CTrapCleanup::New();
    TRAPD(error,doTestsL());
    __ASSERT_ALWAYS(!error,User::Panic(_L("doTestsL"),error));
    delete cleanup;
    __UHEAP_MARKEND;
    return 0;
    }
```

Appendix C: Full source of improved active-object-based solution

```
#include <e32base.h>
#include <e32cons.h>
#include <e32svr.h>

LOCAL_C void doTestsL();

_LIT(KThreadProducerName, "producer");
_LIT(KThreadConsumerName, "consumer");

const TInt KBufferSize = 10;
const TInt KEndCondition = 50000;

class CProducer;
class CConsumer;

struct TData
    {
    public:
        TInt iDataNumber;
        TBuf<32> iMessage;
    };

class CBuffer
    {
    public:
        CBuffer();
        ~CBuffer();
        void ConstructL(TInt aNumProducers, TInt aNumConsumers);
        void Add(TData* aData, TRequestStatus& aRS);
        void Remove(TData* aData, TRequestStatus& aRS);
        void Cancel(TRequestStatus& aRS);
    private:
        void CompleteAddReq(TData* aData, TRequestStatus* aRS);
        void CompleteRemoveReq(TData* aData, TRequestStatus* aRS);

        struct TRequestData
            {
            TRequestStatus* iRS;
            TData* iData;
            };
        CCirBuf<TData>* iBuffer;
        CCirBuf<TRequestData> *iProducerRequests;
        CCirBuf<TRequestData> *iConsumerRequests;
    };

class CController
    {
    public:
        CController(TInt aNumProducers, TInt aNumConsumers);
        ~CController();
        void ConstructL();
        void ControllerMainL();
        void Start();
        void Stop();

    public: // data
        CBuffer* iBuffer;
        TInt iConsumeCount;
        TInt iHeap;
        TInt iStack;
        TTime iEndTime;

    private:
        RPointerArray<CProducer> iProducers;
        RPointerArray<CConsumer> iConsumers;
        TInt iNumProducers;
        TInt iNumConsumers;
    };

class CProducer : public CActive
    {
    public:
        CProducer(CController* aController, TName aName);
        void ConstructL();
        ~CProducer();
        void IssueRequest();
        void CreateNewData();
        void Start();
    private:
        virtual void DoCancel();
        virtual void RunL();
    private:
        CController* iController;
        TData iData;
        TInt iCount;
        TName iName;
    };

class CConsumer : public CActive
    {
    public:
        CConsumer(CController* aController, TName aName);
        void ConstructL();
        ~CConsumer();
        void Start();
        void IssueRequest();
        TBool ProcessData();
        TData iData;
    private:
        virtual void DoCancel();
        virtual void RunL();
    private:
        CController* iController;
        TName iName;
    };

// CProducer

CProducer::CProducer(CController* aController, TName aName)
        : CActive(0),
          iController(aController),
          iName(aName)
    {
    CActiveScheduler::Add(this);
    }

void CProducer::ConstructL()
    {
    }

CProducer::~CProducer()
    {
    Cancel();
    }

void CProducer::Start()
    {
    CreateNewData();
    IssueRequest();
    }

void CProducer::CreateNewData()
    {
    iData.iDataNumber = 1;
    iData.iMessage = _L("producer is producing!");
    }

void CProducer::IssueRequest()
```

```
    {
    iController->iBuffer->Add(&iData, iStatus);
    SetActive();
    }

void CProducer::RunL()
    {
    if (iStatus == KErrNone)
        {
        CreateNewData();
        IssueRequest();
        }
    }

void CProducer::DoCancel()
    {
    iController->iBuffer->Cancel(iStatus);
    }

// CConsumer

CConsumer::CConsumer(CController* aController, TName aName)
        : CActive(0),
          iController(aController),
          iName(aName)
    {
    CActiveScheduler::Add(this);
    }

void CConsumer::ConstructL()
    {
    }

CConsumer::~CConsumer()
    {
    Cancel();
    }

void CConsumer::Start()
    {
    IssueRequest();
    }

TBool CConsumer::ProcessData()
    {
    iController->iConsumeCount++;
    if (iController->iConsumeCount > KEndCondition)
        {
        iController->Stop();
        return ETrue;
        }
    return EFalse;
    }

void CConsumer::IssueRequest()
    {
    iController->iBuffer->Remove(&iData, iStatus);
    SetActive();
    }

void CConsumer::RunL()
    {
    if (iStatus == KErrNone)
        {
        if (!ProcessData())
            IssueRequest();
        }
    }

void CConsumer::DoCancel()
    {
```

```
    iController->iBuffer->Cancel(iStatus);
    }

// CCBuffer

CBuffer::CBuffer() {}

CBuffer::~CBuffer()
    {
    delete iBuffer;
    delete iProducerRequests;
    delete iConsumerRequests;
    }

void CBuffer::ConstructL(TInt aNumProducers, TInt aNumConsumers)
    {
    iBuffer = new (ELeave) CCirBuf<TData>;
    iBuffer->SetLengthL(KBufferSize);

    iProducerRequests = new (ELeave) CCirBuf<TRequestData>;
    iProducerRequests->SetLengthL(aNumProducers);

    iConsumerRequests = new (ELeave) CCirBuf<TRequestData>;
    iConsumerRequests->SetLengthL(aNumConsumers);
    }

void CBuffer::Add(TData* aData, TRequestStatus& aRS)
    {
    aRS = KRequestPending;
    if (iBuffer->Count() < iBuffer->Length())
        {
        CompleteAddReq(aData, &aRS);
        }
    else
        {
        TRequestData rd;
        rd.iData = aData;
        rd.iRS = &aRS;
        iProducerRequests->Add(&rd);
        }
    }

void CBuffer::Remove(TData* aData, TRequestStatus& aRS)
    {
    aRS = KRequestPending;
    if (iBuffer->Count() > 0)
        {
        CompleteRemoveReq(aData, &aRS);
        }
    else
        {
        TRequestData rd;
        rd.iData = aData;
        rd.iRS = &aRS;
        iConsumerRequests->Add(&rd);
        }
    }

void CBuffer::CompleteAddReq(TData* aData, TRequestStatus* aRS)
    {
    iBuffer->Add(aData);
    User::RequestComplete(aRS, KErrNone);

    TRequestData rd;
    if (iConsumerRequests->Remove(&rd))
        {
        CompleteRemoveReq(rd.iData, rd.iRS);
        }
    }

void CBuffer::CompleteRemoveReq(TData* aData, TRequestStatus* aRS)
```

```
    {
    iBuffer->Remove(aData);
    User::RequestComplete(aRS, KErrNone);

    TRequestData rd;
    if (iProducerRequests->Remove(&rd))
        {
        CompleteAddReq(rd.iData, rd.iRS);
        }
    }

void CBuffer::Cancel(TRequestStatus& aRS)
    {
    TInt reqs = iProducerRequests->Count();
    while (reqs)
        {
        TRequestData rd;
        iProducerRequests->Remove(&rd);
        if (rd.iRS != &aRS)
            iProducerRequests->Add(&rd);
        else
            {
            User::RequestComplete(rd.iRS, KErrCancel);
            return;
            }
        reqs--;
        }
    reqs = iConsumerRequests->Count();
    while (reqs)
        {
        TRequestData rd;
        iConsumerRequests->Remove(&rd);
        if (rd.iRS != &aRS)
            iConsumerRequests->Add(&rd);
        else
            {
            User::RequestComplete(rd.iRS, KErrCancel);
            return;
            }
        reqs--;
        }
    }

// CController

CController::CController(TInt aNumProducers, TInt aNumConsumers)
: iConsumeCount(0), iNumProducers(aNumProducers), iNumConsumers(aNumConsumers)
    {
    }

CController::~CController()
    {
    iProducers.Close();
    iConsumers.Close();
    delete iBuffer;
    }

void CController::ConstructL()
    {
    iBuffer = new (ELeave) CBuffer;
    iBuffer->ConstructL(iNumProducers, iNumConsumers);

    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        TName name(KThreadProducerName);
        name.AppendFormat(_L("%d"), i+1);
        CProducer* p = new (ELeave) CProducer(this, name);
        iProducers.Append(p);
        }

    for (i = 0; i < iNumConsumers; i++)
        {
        TName name(KThreadConsumerName);
        name.AppendFormat(_L("%d"), i+1);
        CConsumer* c = new (ELeave) CConsumer(this, name);
        iConsumers.Append(c);
        }
    }

void CController::Start()
    {
    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        iProducers[i]->Start();
        }

    for (i = 0; i < iNumConsumers; i++)
        {
        iConsumers[i]->Start();
        }
    }

void CController::Stop()
    {
    iEndTime.UniversalTime();

    TInt i;
    for (i = 0; i < iNumProducers; i++)
        {
        delete iProducers[i];
        }

    for (i = 0; i < iNumConsumers; i++)
        {
        delete iConsumers[i];
        }

    CActiveScheduler::Stop();
    }

// run the test
LOCAL_C void doTestsL()
    {
    CConsoleBase *console =
        Console::NewL(_L("Tests"),TSize(KConsFullScreen,KConsFullScreen));
    console->Printf(_L("press any key to start.\n"));
    console->Getch(); // get and ignore character

    TInt iterations;
    for (iterations = 1; iterations <= 111; iterations += 10)
        {
        TTime startTimeIni;
        startTimeIni.UniversalTime();
        TInt numProducers = iterations;
        TInt numConsumers = iterations;
        console->Printf(_L("Running with %d prod. and %d cons. threads.\n"),
            numProducers, numConsumers);

        CActiveScheduler* activeScheduler = new (ELeave) CActiveScheduler;
        CleanupStack::PushL(activeScheduler) ;
        CActiveScheduler::Install(activeScheduler);

        CController* controller =
            new (ELeave) CController(numProducers, numConsumers);
        CleanupStack::PushL(controller);
        controller->ConstructL();
        controller->Start();

        // Start handling requests
        TTime startTime;
        startTime.UniversalTime();
        CActiveScheduler::Start();
        TTime endTime = controller->iEndTime;

        delete controller;
        CleanupStack::Pop(controller);
        CleanupStack::PopAndDestroy(activeScheduler);

        TTime endTimeIni;
        endTimeIni.UniversalTime();

        TInt fullTime =
            endTimeIni.MicroSecondsFrom(startTimeIni).Int64().GetTInt();
        TInt runTime = endTime.MicroSecondsFrom(startTime).Int64().GetTInt();
        console->Printf(_L("Completed in %d + %d microseconds.\n\n"),
            fullTime-runTime, runTime);
        }
    console->Printf(_L("THE END. press any key to exit.\n"));
    console->Getch(); // get and ignore character
    delete console;
    }

GLDEF_C TInt E32Main() // main function called by E32
    {
    __UHEAP_MARK;
    CTrapCleanup* cleanup=CTrapCleanup::New();
    TRAPD(error,doTestsL());
    __ASSERT_ALWAYS(!error,User::Panic(_L("doTestsL"),error));
    delete cleanup;
    __UHEAP_MARKEND;
    return 0;
    }

```