Operating System with Priority Functions and Priority Objects

Rabih Chrabieh

7th February 2005

Additional material and software available at
http://www.portos.org

Contact information
contact@portos.org

Contents

1 Introduction

The question we ask here is "are tasks or threads necessary in an operating system?" The short answer is that if time slicing is needed, they are absolutely necessary. However, time slicing is seldom necessary within one software application. In all other cases, tasks complicate significantly the software architecture without being indispensable.

In this article we introduce new concepts for Operating Systems. The ideas are most relevant to Real Time Operating Systems (RTOS) but are not restricted to Real Time. The general purposes behind these concepts are to simplify the software programming of embedded systems and to enhance the real time performance. The idea is to avoid using tasks. We replace the task by a "priority function". This is a regular function to which we assign a priority level. A scheduler executes "priority functions" at the appropriate priority levels.

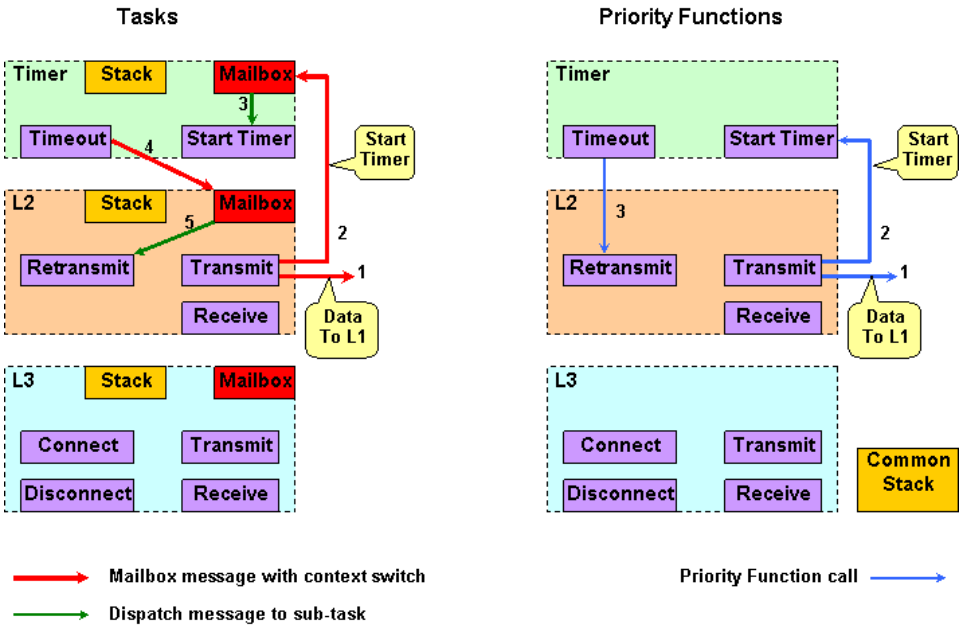

Figure 1: Modem implemented with tasks versus modem implemented with priority functions. Layer 2, layer 3 and the timer tasks are shown. Layer 1 is not shown. Layer 2 transmits data. It also starts a timeout timer. If the data is lost, the timer expires and activates the retransmission code in $L2$. The complexity of the implementation with tasks is obvious. In the priority function implementation, functions call each other directly without going through context switches, semaphores, mailboxes, etc.

The main benefits, that will become clearer throughout the document, are

- Simpler architecture
- Improved modularity

- Code easier to modify and upgrade
- More efficient code
- Less stack space

One more important benefit that is not discussed in this document is that a programming language compiler or interpreter can be augmented with the concepts of priority functions and priority objects. This significantly simplifies the job of writing real time software.

Figure 1 shows a comparison between an implementation with tasks and another one with priority functions.

In the following sections we give a brief description of how traditional RTOS work then we present the new concepts of priority functions and priority objects. Then we give examples of benefits.

2 Traditional RTOS

What is a task? A task is essentially a piece of code that can run, stop, run again, etc. Why does it stop? For instance, in order to wait for new input. Or it can stop if it is preempted by another task with higher priority. Since every modern program uses a stack to execute function calls, a task needs its own stack in order to run, stop and run again. This is to preserve the stack state when the task stops for a moment (a shared stack would get trashed). When a task A stops and another task B resumes, a "context switch" occurs. In the context switch, the registers of task A are saved to stack of A. The registers of task B are restored from stack B. This operation is quite heavy.

What is the purpose of a task? The main reason behind a task is to execute different pieces of code at different priority levels. For instance, a low priority task can run the main algorithms while a higher priority task can execute time sensitive code. The high priority task preempts the low priority task at any point in time (or almost). Note that hardware interrupts can achieve similar behavior but they are usually reserved to very time sensitive instructions. We are not concerned with hardware interrupts here.

Another two reasons behind a task are

- Time-slicing so that every task gets a bit of time to execute. This is seldom needed within one program application.
- Multi-processor architecture where an application is split into several tasks to run on different processors. Again a rare case.

With an architecture built around tasks, a program gets quickly complicated. For example, if a function F runs in task A, it is impossible to call it from task B. This is awkward. The way to call function F is by following the steps:

- send a message to task B (containing information on what function to call with what parameters)

- stop task A
- start task B
- dispatch the message within task B to the appropriate sub-task
- call the function F
- when function F is done, stop task B
- resume task A and proceed

The above is what happens to a simple function call when tasks are involved! Besides the wasted memory resources and the decrease in performance, the program structure is more complex. A programmer has to be aware of all the above, of where the function should run, of what type of messages to send, etc.

Moreover, mailboxes are needed to handle inter-task messages. Semaphores are used to protect data shared between different tasks. Sub-tasks are used to avoid burdening the system with too many tasks (and unfortunately sub-tasks cannot have their own priority). Dispatchers between sub-tasks are necessary.

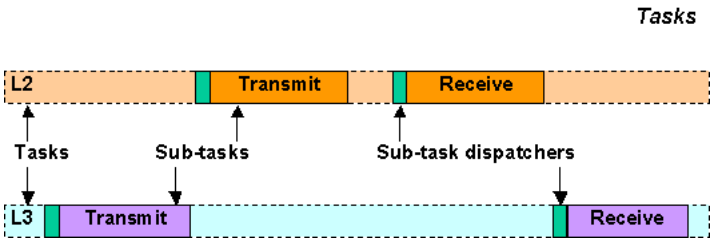

Figure 2: Modem implemented with tasks. Layer 2 and 3 tasks, sub-tasks and dispatchers are shown. Dispatchers manage messages in mailboxes. Mailboxes and dedicated stacks are required but are not shown here.

Note that semaphores result in a large number of context switches. To prevent this, a new technique called "priority ceiling" was developed and consists of increasing the priority level of the task over the critical region, instead of protecting it with a semaphore. A similar technique is used in our proposed RTOS.

3 Priority Functions

Since the main idea of a task is to run different code at different priority levels, we can assign priority levels directly to functions without creating tasks. A priority function is a normal function (e.g., C or C++ function) to which we assign a priority level. The priority function is almost called like a regular function. If its priority is above the current priority, the function is called immediately. If

its priority is below the current priority, the request is stored in a database and the function is called when all higher priority functions have terminated.

Unlike a typical task, a function has to complete its job and return. It cannot suspend or loop forever. A priority function does not possess a dedicated stack (unless the user specifically desires to allocate one, e.g., in slow or fast RAM). There is no context switch when a priority function is called.

Since priority functions do not need a separate stack, and since the context switch is minimal, a large number of priority functions can be defined without degrading performance or wasting system resources. Therefore, every sub-task, in the traditional RTOS sense, can be defined as an independent priority function. It can be assigned any priority level, and it can be called from anywhere in the code.

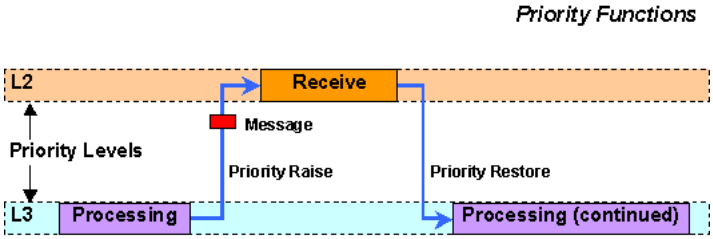

Figure 3: Modem implemented with priority functions. Layer 2 and 3 priority functions are shown. $L3$ Processing is preempted by $L2$ Receive. The priority level is raised, and a message is delivered to $L2$ Receive. When $L2$ Receive is done, the original priority level is restored and $L3$ Processing resumes.

A priority function also takes one (or more) argument. The argument is a pointer to the message to be delivered to the function. This is how inter-function communication works. No message queues or mailboxes are necessary. A priority function does not suspend while waiting for a new message. Rather, it is called when the message is ready.

3.1 Potential Drawbacks and Remedies

There are two important drawbacks to priority functions:

1. Semaphores cannot be used to protect shared data since a priority function cannot suspend execution. A solution is to provide dedicated priority functions that handle the access to the shared data at a fixed priority level (this implicitly acts like a FIFO of data updates). Another simpler solution is to raise the priority level over the critical section the same way the "priority ceiling" in traditional RTOS works.

2. A priority function F cannot suspend and wait for the result of another priority function G. If the result of G is necessary, the priority function F can be attached to a signal that is delivered when G is done. It may

be necessary to split the priority function F into two priority functions, F_1 and F_2. F_1 runs the code that is independent of the result of G. F_2 is attached to a signal delivered by G and runs after G is done.

3.2 Co-existence of priority functions and tasks

It is possible to mix priority functions and tasks within one system. The main difference between a priority function and a task is that the former is more efficient, it does not need a dedicated stack but it cannot suspend.

Also, in order to achieve time-slicing when needed, priority functions can be encapsulated inside a task. The priority functions scheduler can be easily replicated for each task.

4 Priority Objects

Now instead of assigning priority levels to functions, we can push the logic further and assign the priority levels to objects. A priority object is a data structure or a resource that has been assigned a priority level. The functions, in this case, are methods that perform operations on various objects with various priority levels. A priority function inherits the priority level of the object it is processing. Priority objects are essentially a layer on top of the priority functions.

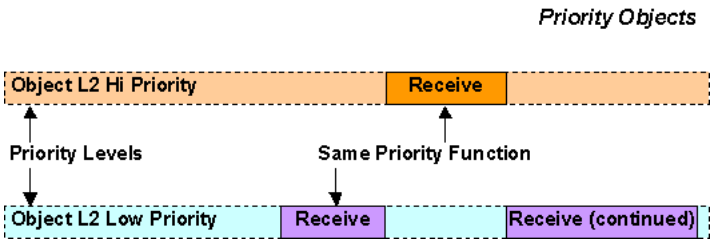

Figure 4: Modem implemented with priority objects. Two, or more, layer 2 objects are defined (e.g., a low priority service and a high priority service). The same *L2* Receive function can handle both objects. It inherits the priority level of the object it is processing. This is too expensive to achieve with traditional tasks.

The priority level of an object is set at instantiation time. The priority level can be inherited from a parent type or from a module's set of priorities.

4.1 Multi Priority Objects

One object can be assigned several priority levels. One priority level, the highest could be reserved to update the object's data. Other priority levels can be assigned to various methods that will operate on the object.

5 Examples of benefits

5.1 Timers

We will start illustrating the benefits of the proposed RTOS with the typical example of the timers.

Designing a good timer application is a difficult problem in traditional RTOS. The main issue is in which task does the timer expiration function run?

Some traditional RTOS handle timers in a flexible way. All timer expiration functions are executed in a special very high priority task. A requirement on the expiration function is to be very short (heavy system calls have to be avoided). The expiration function then sends a message to the task that is meant to execute the bulk of the code.

Although flexible, this solution means that messages have to be defined, dispatchers have to be written and context switches will occur when timers expire.

In our proposed RTOS, the timer application is straightforward. When a timer expires, the attached priority function is called (like a regular function) at the corresponding priority level, and with the corresponding message argument.

Figures 5 and 6 show the example of a modem implemented respectively with tasks and with priority functions.

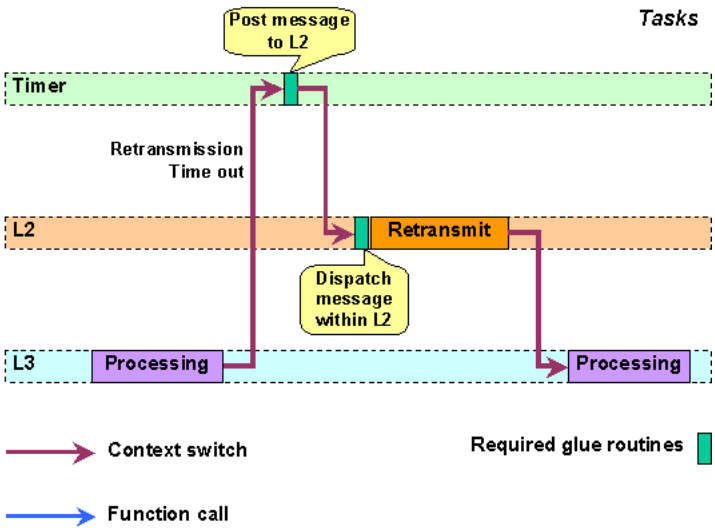

Figure 5: Modem implemented with tasks. An $L2$ retransmission timeout occurs that suspends task $L3$ and awakens task $L2$. Three context switches take place before $L3$ resumes. Glue routines are needed at the timer task level and the $L2$ task level.

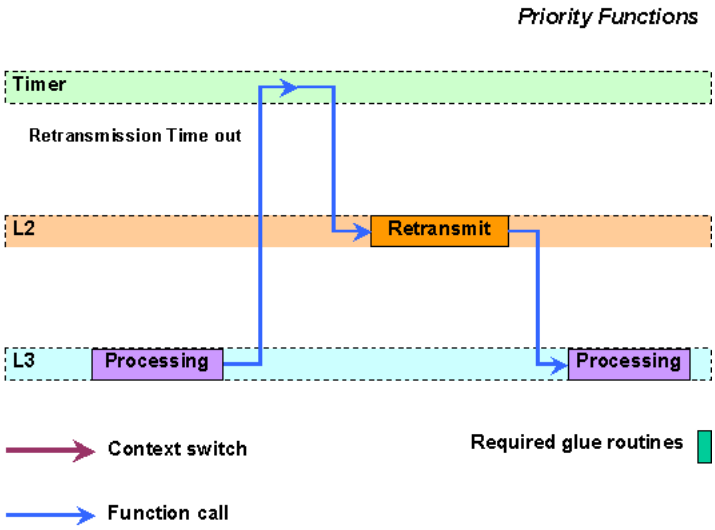

Figure 6: Modem implemented with priority functions. An $L2$ retransmission timeout occurs that preempts the $L3$ processing. No context switches occur. No glue routines are required.

5.2 The user interface and host port

In a traditional RTOS, a user interface is often implemented as a background task. An instruction is received from a keyboard via a serial connection, for instance. The instruction is then dispatched to the appropriate task and sub-task. This normally involves the following steps:

1. Define a message from the user interface task to the recipient task.
2. Write a dispatcher to send the message to the recipient task.
3. In the recipient task, write another dispatcher to call the sub-task that will execute the instruction.

Adding a new user interface instruction requires several steps. It also involves several context switches. If an instruction is moved from one task to another (for some reason), several dispatchers have to be updated. A good knowledge of how the tasks are organized and in which task the instruction should execute are also necessary. The job is pretty involved for a new programmer who does not yet have a global understanding of the overall organization of tasks and sub-tasks in the project.

In our proposed RTOS, a user interface is implemented as a set of priority functions. The instruction received from the user interface is defined as a priority function. The instruction is called in a straightforward manner from the user interface. The steps above are not needed. A global understanding

of the organization of the tasks is not required. This instruction is executed independently outside any task.

The above also applies to the host port case. A host port is a connection from the DSP to some host processor. Messages exchanged over the host port have to be re-routed to the recipient tasks and sub-tasks.

5.3 The writer and reader problem

A writer and reader are often implemented as two tasks in traditional RTOS. When new data is received, the writer writes it to a shared FIFO. When the writer is done, it posts a semaphore or a signal for the reader and it suspends (waiting for new data). The reader is awakened. It reads the FIFO content. When done reading, the reader suspends for the semaphore or signal delivered by the writer. Context switches occur each time the writer or reader is awakened or suspended.

In our proposed RTOS, the solution is essentially the same but without heavy context switches. When new data is received, the writer writes it to the FIFO. It posts a signal for the reader and returns (or it can the reader directly). The reader is called, it reads the data and before returning it re-attaches itself to the signal delivered by the writer.

5.4 Dialog box in a Graphical User Interface

A dialog box in a GUI is sometimes implemented as a thread. This way the software keeps running while the dialog box prompts the user for input. Such a dialog box can instead be implemented with priority functions, saving stack space and complicated mailboxes for inter-thread communication. Embedded RTOS can save resources this way.

In general, it is only necessary to create a new thread when true time slicing is required. This is the case when two different applications are running concurrently. It is seldom the case within one application. For instance, if the dialog box needs to execute a piece of code, usually the remaining of the application can wait, or the dialog box can wait.

6 Summary of the benefits

In a traditional RTOS, a major effort has to be spent in order to obtain a clean design of tasks, mailboxes, and so on. For each function (sub-task), the designer has to be well aware of where it will run (i.e., in which task), how to route any messages it exchanges with the rest of the system, etc. Moreover, routines from one module can be spread over several tasks (some running at high priority, some running at low priority). A routine running in one task cannot directly call a routine running in another task. The module (and the programmer) has to be aware of the existence of the tasks and their corresponding mailboxes. In order to call a routine running in a different task, a message has to be defined and sent

to that task. If a routine is moved from one task to another, many modifications related to the messages and dispatchers have to take place. Modularity is partly lost. Complexity is high. Finally, context switches decrease performance and stack space is wasted.

When working with priority functions, each priority function is created independently of the rest of the system. No global understanding of the system is necessary. Modularity is better served. The priority function can be called from anywhere. The message it needs is embedded in its argument.

Hence, the software architecture is simpler. A project is faster ported from the algorithmic platform (e.g., Matlab) to the embedded platform. The embedded designer does not spend a great amount of time designing tasks, inter-task communications and assigning routines to tasks. Debugging time should also decrease since functions are kept independent and there is less interaction between various parts of the system. Simpler architecture also results in a more efficient code with no context switches.

7 Advanced Topics

7.1 Implementation Ideas for Priority Functions

A priority function cannot be called directly using a normal function call since a normal function call does not handle priorities (unless the compiler is made aware of priority functions). The priority function can be called via an API routine that decides at which point the priority function will be called (based on the priority level). If the priority function is not called immediately, it is stored in a database that can have a linear structure or a binary tree structure sorted in decreasing order of priorities. A more efficient database may be implemented by limiting the total number of priority levels. In this case, priority functions at each level are stored in linked lists. A bit associated to each level is stored in one or two integers and is set to 1 when the linked list is non-empty. A fast algorithm detects the highest bit that is set to 1 and therefore the highest priority function awaiting execution.

The API routine takes for arguments the priority function, the priority level, and the message to be passed to the function.

It is important to prevent the priority function from being called accidentally as a normal function (which results in a priority level violation). It is therefore preferable to hide the priority function, and to define a priority function handler that contains a pointer to the priority function. Only the handler is made visible to external code. Alternatively, the priority function is made visible only through a cover function that calls the priority function via the API routine.

Priority functions can return values via the message that was sent to them. In this message, fields can be reserved for any return values. In this case, the priority function does not free the message before returning. If the caller expects to receive the result from the priority function immediately after the call, it is mandatory that the priority function is assigned a higher (or equal) priority

than the caller. The message can be allocated on stack (instead of in dynamic memory) for more efficient code. An API can be provided that checks if the priority levels are consistent. If not, the API reports an error.

7.2 Implementation Ideas for Priority Objects

A priority object is defined as a data structure along with a priority handler. The priority handler is a structure containing, among other things, the priority level of the object. The priority handler can sometimes be embedded in the object's data structure so that both are allocated in one shot. The priority level is initialized when the object is created. The priority level should never be modified. This priority level is inherited by every priority function the object is passed to.

For debugging purposes, the priority object can be defined with a special keyword recognized by some pre-processor. The pre-processor tracks the object throughout the code and verifies that the object is only accessed via valid priority functions (its methods). This may be easily achieved in object oriented programming languages. Alternatively, violations can be detected at run-time.

7.3 Priority Management

Priority functions are typically executed at a level above the tasks level and below the hardware interrupts level. In some traditional RTOS, the priority functions can be assimilated to software interrupts with advanced features. The priority functions level is established by issuing a software interrupt (or equivalent). A hardware interrupt can preempt a priority function F_1 running in software interrupt S_1. If the hardware interrupt calls a new priority function F_2 with higher priority level, then a new software interrupt S_2 must be issued. S_2 must have a higher priority than S_1. S_2 handles the call to F_2. When done, S_2 terminates and S_1 resumes.

The priority functions should not be called directly with the object as a parameter (otherwise the priority level is violated). Instead, an API routine receives the call request along with a pointer to the priority handler, and a pointer to the object itself. To prevent a direct call to the function, the function can be hidden in a file and only made visible through a priority function handler containing a pointer to the function.

If the priority level of the priority function is above the current priority level, it is called immediately. If it is below, it is stored in a database to be executed later, at the appropriate priority level. What happens if the priority level is equal to the current priority level? There are two cases:

1. The new priority function F_2 is being called from within another priority function F_1 running at the same priority level. In this case, the new priority function F_2 is executed immediately as if its priority level was above F_1. This is often the case when several priority functions are running

a related job. If there is a real need to run the new priority function afterward, a special API can be provided to handle this case.

2. The new priority function F_2 with a priority equal to that of F_1 is being called from a priority function G running at a higher priority level than both F_1 and F_2. The same is valid if F_2 is being called from interrupt level (interrupt level acts like a virtual priority function with very high priority level). In this case, F_2 gets automatically stored in the database and is activated after F_1 is done.

For maximum efficiency, a special API can be provided that inlines the call to a priority function when both the current and new priority levels are known constants. The overhead of calling a priority function is entirely eliminated in the case where the new priority level is equal or above the current one.

In order to manage a large number of priority levels, a separate file can be created to maintain all possible priority levels. The various priority levels are listed in groups of decreasing order of priority. The priority of a group is constant. Each priority level defined in this file represents a certain type of objects, a certain type of operation, or a certain module. Objects defined in other files inherit one of these priorities.

Note: a priority function should not be passed simultaneously objects with different priority levels.

7.4 Signals

Signals are essential for event synchronization and data protection. A priority function can be attached to a signal. When the signal is received, the priority function is sent to the scheduler. An attached priority function can be detached (canceled) before it has been sent to the scheduler.

There are various ways to implement signals. We will discuss two flexible methods.

1. Define each signal as a linked list of priority functions. Each time a priority function is attached to the signal, the handler of the priority function is appended to the linked list. When the signal is posted all priority functions in the linked list are sent to the scheduler.

2. A more general method is to define a signal as an integer value. Different independent groups of signals can be defined, each group consisting of its own set of signals and its own database. The database can consist of a hash table of size dependent on the desired efficiency. Each hash entry points to a tree with one trunk and several branches. In the trunk are listed, in sorted order, the signals with different values. Each signal possess a branch of priority functions. When a signal is posted, the corresponding branch is removed and all priority functions are sent to the scheduler.

The signal handling routines are priority functions. The signal group is a priority object. It should be assigned a priority level above all priority functions

that can be attached to this signal group. The signal handling routines inherit the group priority level. Having a higher priority level than any attached priority function ensures proper priority level behavior. Otherwise, servicing at the right priority level is not guaranteed.

7.5 Timers

Timers are a special case of signals. They can be defined as a layer on top of the second method of signal handling. Each signal value (integer) is the clock value at which the timer expires. When the timer expires, the attached priority function is executed. Different clocks (e.g., frame clock, time slot clock, etc) can be implemented as different groups of signals (with the timer routines running possibly at different priorities for different clocks).

7.6 Message logging

A message logging function can be implemented as a priority function with low priority level. It runs in the background when there are messages for printing. The message logging function can also maintain a linked list of pending messages, i.e., those messages that have not been printed yet. In the event of a system crash, the linked list can be recovered and the pending messages can be printed.

For simplicity and efficiency, the message logging function can take a format string and a fixed number of arguments of type integers or pointers.

7.7 Memory Management

Efficient memory management is a critical aspect of this RTOS. It is important to be able to allocate objects in a very efficient manner and from any level, e.g., from an interrupt level (e.g., to schedule priority functions and pass them messages).

Traditional byte-oriented memory allocation (i.e., allocation of variable size in bytes) is highly inefficient. The allocation and free operations require expensive search and recombining of unused memory spaces.

For this reason, traditional RTOS offer block-oriented memory allocation. In this method, blocks of fixed sizes can be pre-allocated and used by a specific task. The advantage is that memory allocation is extremely efficient. The problem here is the fixed size.

A generalization of the block-oriented memory allocation is a variable-size block-oriented memory allocation. In this method, different block sizes are managed by the memory allocation routines. For example, blocks of sizes 8, 16, 32, 64, etc, can be defined. A block handler maintains, for each block size, a linked list of the free elements. Each linked list is assigned an index. When a block of size say 27 is requested, the size is rounded up to 32. The index of the corresponding linked list is calculated. A free block in the linked list, if any, is returned. If there are no free blocks in this linked list, a new block is

allocated from a large pool of memory. Blocks are allocated from this large pool sequentially and never returned. Once allocated, their size is fixed, and they can only be returned to the appropriate linked list.

With this scheme, there is no need to pre-allocate blocks. The blocks can have a variable size and they are shared by different tasks (they are not pre-allocated by a given task).

For maximum efficiency, and for blocks of constant size, an API can return the index of the linked list corresponding to the block size. Each time such a block is to be allocated, the step of converting from block size to linked list index is bypassed.