Memory Pools

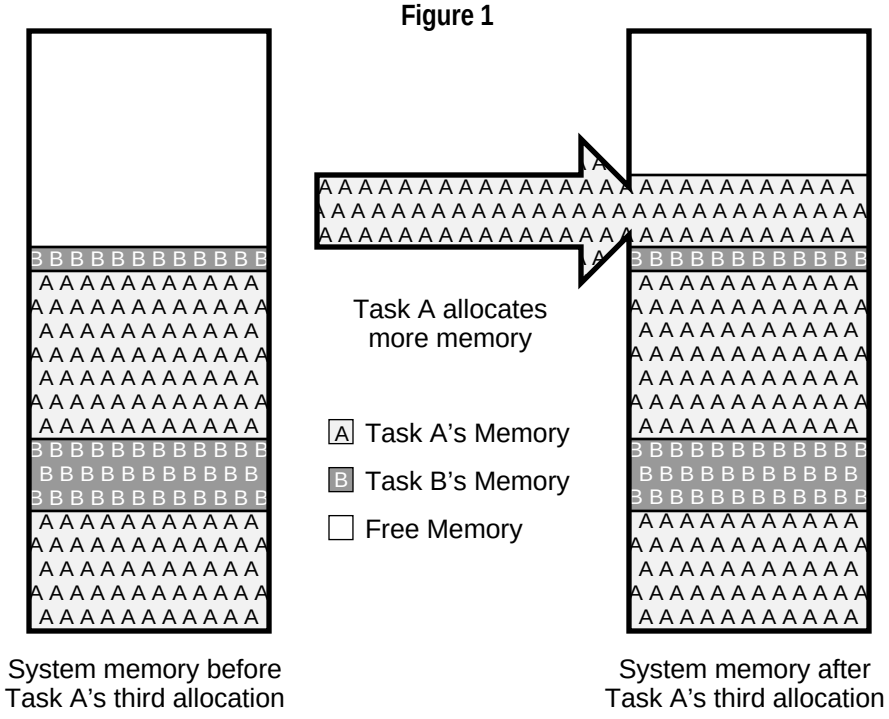
by Mark Ricci

A memory pool is a block of memory that an application allocates for all its memory needs. The application draws from this pool rather than the system's free memory list whenever it requires memory.

Memory pools add to the efficiency of the system and applications in two ways. The first is a potential decrease in memory fragmentation. Memory fragmentation is a condition where system memory is made up of blocks of allocated memory interspersed with blocks of free memory. The second benefit of memory pools is faster memory allocations and deallocations.

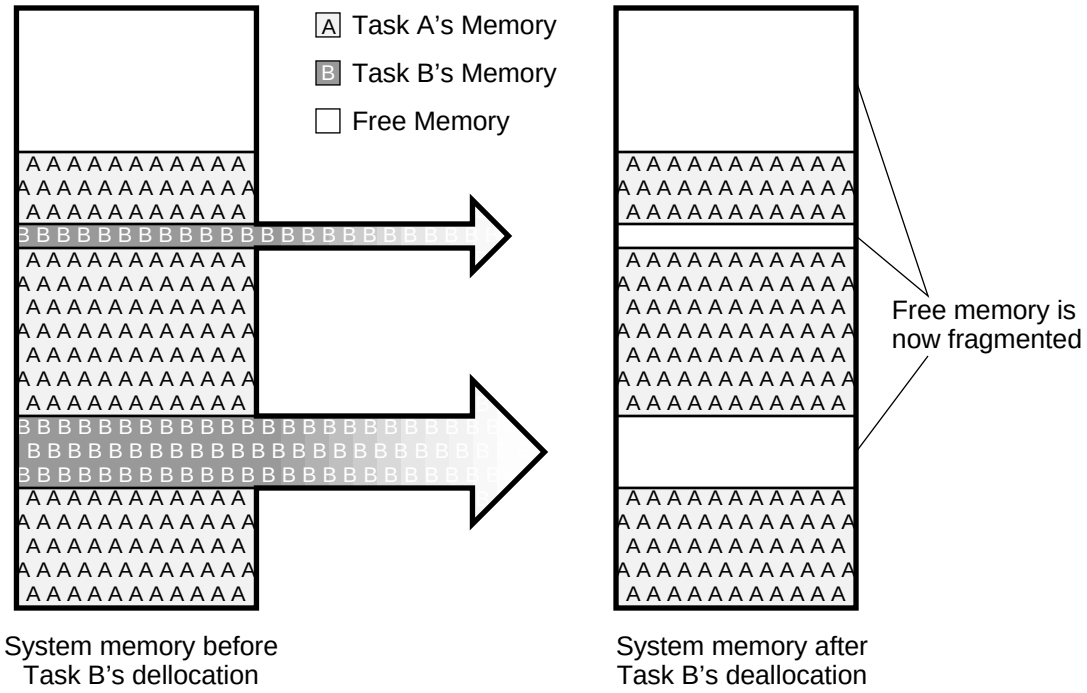
System memory starts out as one large, contiguous block. The first time a task allocates memory, the system memory is broken into two blocks--allocated memory and free memory. As different tasks allocate memory, the allocated memory block increases and the free memory block decreases as shown in Figure 1.

Figure 1



The memory blocks remain contiguous until a task frees some memory. At that point, a gap may appear in the block of allocated memory if the newly freed block was not the last one allocated. While this creates additional free memory, it also means that free memory is now two or more non-contiguous fragments instead of one contiguous block. In a multitasking system like the Amiga, tasks are constantly allocating and freeing memory which can create many gaps in memory. As a result, free memory can consist of many small pieces rather than one large block (Figure 2).

Figure 2



This can lead to problems if an application needs a large block of contiguous memory. For example, consider an application that needs a block of memory 200,000 bytes long. If free memory is severely fragmented, the largest block of free memory may only be 150k. The system may have five megabytes of free memory available, but that five megabytes of memory is broken into fragments no larger than 150k. In this case, the application cannot allocate its 200k.

By having an application take a pool of memory and allocate from it, fragmentation of system memory is decreased. An application may require six allocations ranging from 20 to 678 bytes in size which can be scattered throughout system memory. However, if those same six allocations are taken from a pool, the fragmentation occurs within the pool, but as far as the system is concerned, it's missing one large block instead of six small ones.

An application using a memory pool will have faster memory allocations because the memory list of a pool is smaller, and therefore easier to traverse than the memory list of the system memory. For deallocations, it's even faster because deleting the pool itself deletes all the individual allocations made from it instead of having to deallocate each piece that was allocated.

Memory Pool Organization

A memory pool consists of units called puddles. Other than that, it has no formal organization. There is no pool structure defined anywhere. All you know about a pool is its address. Exec's pool manager handles the rest of the details.

When you create a pool, you specify the size of its puddles and a threshold value, not the size of the pool. This is because memory pools are dynamic, they have no fixed size. The pool manager takes the memory for your application from the puddles. As you require memory, the pool manager expands the pool by adding puddles, and as you free memory, the pool manager shrinks the pool by deleting puddles.

The size of the puddles is important because the pool manager will try to use as much of a puddle as possible before adding a new puddle. Each time you allocate memory from the pool, the pool manager takes the memory from a puddle unless an allocation exceeds the amount of memory remaining in a puddle. If the allocation request exceeds the memory remaining in a puddle, the pool manager adds a new puddle. If the allocation exceeds the pool's puddle size, the pool manager will create a special oversized puddle to accommodate the application.

The key is to use all the drops in a puddle. If you create a pool with puddles of 200 bytes and allocate 150 bytes at a time, you'll waste 50 bytes in every puddle. Set the puddle size in accordance with your memory requirements.

The threshold value is the size of the largest allocation to use within a single puddle. An allocation request larger than the threshold value causes the pool manager to add a new puddle. Ideally, the threshold value should be the size of the largest allocation you will make.

The relationship between puddle size and threshold can be tuned for optimal utilization of a pool. Threshold sizes cannot exceed puddle sizes and are recommended to be one half the puddle size so that you can fit two of your largest allocations in a single puddle. However, if you are going to have a mix of large and small allocations, you might want to set a threshold value that allocates a majority of a puddle for a large allocation, and then uses up the remainder of the puddle with the smaller allocations.

Creating a Memory Pool

You create a memory pool by calling CreatePool() specifying the puddle size, threshold size, and memory type. The memory type is borrowed from the AllocMem() function and is comprised of the MEMF_ flags defined in <*exec/memory.h*>. See the Autodoc for AllocMem() for more information on those flags.

If successful, CreatePool() returns the address of the memory pool.

```
APTR mem_pool;

if (mem_pool = CreatePool(MEMF_FAST,4096,2048))
{
    .  .  .
}

else
    printf("Pool could not be created.\n");
```

The example above attempts to create a pool of Fast memory with a puddle size of 4096 bytes and a threshold size of 2048 bytes.

If your application requires memory of different types (for example, Chip memory and Fast memory), it must create a pool for each type.

Take note, again, that all you know about a pool is its address. Do not poke around it trying to figure out its organization. *It's private for a reason!*

Allocating Memory from a Pool's Puddles

Memory is obtained from a pool's puddles by calling AllocPooled(). AllocPooled() requires a pointer to the memory pool and the size of the memory block needed. If successful, AllocPooled() returns a pointer to the memory.

```
struct timerequest *TimerReq;

if (TimerReq = AllocPooled(mem_pool, sizeof(struct timerequest))
{
    .  .  .
}

else
    printf("Memory could not be allocated.\n");
```

Freeing Memory from a Pool's Puddles

An application can free a block of memory it allocated from a pool by calling FreePooled():

```
FreePooled(mem_pool, mem_drop, mem_size);
```

This function requires a pointer to the memory pool (mem_pool), a pointer to the block of memory being freed (mem_drop), and the size of the memory being freed (mem_size).

Deleting a Memory Pool

A memory pool is deleted by calling DeletePool() with a pointer to the memory pool you wish to delete.

```
DeletePool(mem_pool);
```

Deleting a pool also frees the puddles in it. This is quicker than doing individual FreePooled() calls. For example, a text editor will allocate memory for each line of the file being edited. When the editor is exited, each of the allocations has to be freed. With DeletePool(), it would take only one call instead of many.

Why Use Pools When You Can Allocate Memory Yourself?

You could simulate memory pools yourself by allocating the memory you think you'll need and then using it as you go along. You'll also have to keep track of your usage and if you exceed your allocation, you will have to allocate additional memory. Another potential problem is that unlike pools where you dynamically get memory, the allocation scheme requires you to attempt one large allocation, which may fail, and then requires at least two smaller allocations. At that point, you're no farther ahead than you were before.

With pools, you have the pool manager to handle all the details of allocating memory. You can sit back and create the pool, allocate memory from the puddles and free the pool when you're done. That's a lot easier than doing it yourself.

Here's an additional incentive for using memory pools:

Future memory technologies will be implemented through pools.

The example below illustrates the use of memory pools instead of allocating system memory.