Journal of Computer Science 10 (8): 1480-1487, 2014
ISSN: 1549-3636

doi:10.3844/jcssp.2014.1480.1487 Published Online 10 (8) 2014 (http://www.thescipub.com/jcs.toc)

ROBUST MEMORY MANAGEMENT USING REAL TIME CONCEPTS

Karthikeyan, V., S. Ravi and M. Anand

Department of ECE, M.G.R. Educational and Research Institute, Chennai, India

Received 2014-02-14; Revised 2014-02-18; Accepted 2014-04-01

ABSTRACT

Memory fragmentation is the development of a large number of separate free areas. Memory management in embedded systems demand effective implementation schemes to avoid fragmentation problem. Existing dynamic memory allocation methods fail to suit real time system requirements. Execution times need to be deterministic and this motivates the need for allocation and deallocation to be done in constant time with the help of API's. In µC/OS-II, memory allocation is semi-dynamic and a buddy allocator dynamic memory allocation algorithm is commonly used. Programmer must statically allocate a memory and partition the region using µC/OS-II Kernel API. Tasks can only request pre-partitioned fixed-size memory space from µC/OS-II. Memory allocation times are influenced by the ratio of memory allocation to the stack size of the task. In this research work memory management in LPC 1768 environment using RTOS µC/OS-II is proposed. Effective sharing of memory blocks among tasks co exists with partition. The captured results shows that the memory allocation and deallocation suits real time. The implication of the work is that, the necessity to reserve a static set of locations ahead of time is eliminated so that memory can be allocated at compile or design time.

Keywords: Memory Allocation, RTOS, µC/OS, Embedded Systems, LPC1768, Fragmentation

1. INTRODUCTION

Optimal methods for memory management involve allocating memory to a process when needed and deal locating memory when no longer in use. Swapping can make the memory used by all the running processes to exceed main memory and hence needs to be used carefully. Medium-term scheduler removes the un used process from memory. At some later time, the process can be reintroduced into memory and its execution can be continued from where it left. Swapping is a technique where processes are moved into and out of memory as and when memory is requested else remain free. Swapping is handled by a medium-term scheduler. Swapping is also used in paging system where a process that is swapped out will release all its page frames for use by other processes. Swapping requires a backing store (commonly a fast disk). The system maintains a ready queue consisting of all processes whose images are in backing store. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher which in turn check whether the process in queue is in memory, if not, there is no free memory region, the dispatcher swap out a process currently in memory and swap in the desired process. It then reloads the register as normal and transfers control to the selected process. Virtual memory makes it possible to run a single program that uses more memory than the main memory (normally RAM) available on the system. Programs refer to parts of memory using addresses. In a virtual memory system, these are virtual addresses. The virtual address is mapped onto physical addresses by a Memory Management Unit (MMU).

2. MOTIVATION BEHIND THE RESEARCH WORK

The need for effective memory management is illustrated in the following examples. In data base application there is a need to manage time varying data and they are referred to as temporal data bases.

Corresponding Author: Karthikeyan, V., Department of ECE, Dr.M.G.R.Educational and Research Institute, Chennai, India

The temporal data model should be designed in a way to reduce the cost of memory storage (Halawani and Al-Romema, 2010)". Anbumozhi and Manoharan (2014) proposed a method of fuzzy based image fusion in that the author highlighted the need for limited memory buffers with low computational complexity in order to reduce the hardware cost. Due to the limited memory capacity of mobile devices the AST node is generated and executed only when required and there is no need of converting all AST nodes to respective code bytes (Patra *et al*., 2013).

In artificial Immune recognition system there is need to allocate resources in such way that only limited memory cells should be used but without reducing the accuracy of AIRS (Golzari *et al*., 2011). The main objective in scanning of data base to mine the spatially co-located moving objects is to minimize the computation cost and memory usage (Manikandan and Srinivasan, 2013)". Revathy and Saravanan (2014) proposed a efficient parity check decoder for low power applications. In this the author specified the importance of effective utilization of hardware considering the rapid growth of technology memory devices becomes larger and powerful. The modern embedded system applications requires large quantity of memory usage but in many cases an embedded system has limited memory capacity only and also size of input data cannot be estimated in advance.

In situations like above there is a need to design a effective memory management system such that memory can be utilized efficiently (Porwal and Mittal, 2013)".

3. MEMORY ALLOCATION METHODS

In RTOS there are two types of memory allocation; static and dynamic. The classification of memory allocation is shown in **Fig. 1**.

In static memory allocation: Memory is allocated at compile or design time and allocation is prefixed. In dynamic memory allocation memory allocation is done during run time and it requires the status of memory blocks at all instant. In this memory allocation is created and destroyed during run time. Manual memory allocation is simply used for the purpose of understanding the memory management concept. In Automatic memory allocation the allocation task is executed by recycling the blocks and it eliminates memory allocation bugs.

4. DYNAMIC MEMORY ALLOCATION ALGORITHMS

The different types of dynamic memory allocation algorithms are explained below.

4.1. Sequential Fit

The algorithm uses single linear list of all free memory blocks, the blocks are allocated from this list. The algorithm is easy to use but time consuming.

4.2. Indexed Fit

The algorithm uses an indexing data structure like tree to found the free blocks.

4.3. Bit Mapped Fit

The algorithm uses a bitmap to represent the usage of the heap.

4.4. Two Level Segregated Fit (TLSF)

TLSF is a combination of segregated and bitmap fit algorithms. It solves the worst case bound problem and produces high efficient allocation and deallocation in real time applications.

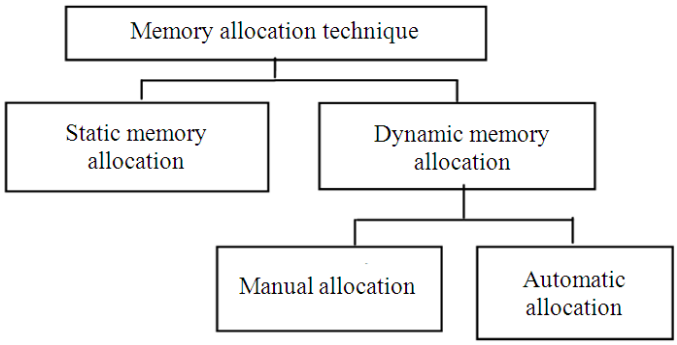

Fig. 1. Memory allocation methods

Table 1. Comparison between memory allocation algorithms

	Parameters	
Algorithms	Response time	Fragmentation
Sequential fit	Slow	Large
Indexed fit	Fast	Large
Bitmapped fit	Fast	Large
TLSF	Faster	Small
Smart memory allocator	The fastest	smallest
Buddy system	Fast	Large

4.5. Buddy Allocator

It uses an array of free lists, one for each allowable block size. The buddy allocator rounds up the requested size to an allowable size and allocates from the corresponding free list. If the allocated free list is empty then another free list is selected. The main advantage of buddy system is that freed blocks can be found quickly by simply address computations. **Table 1** shows the comparison between different dynamic memory allocation algorithms.

5. PROPOSED METHOD FOR MEMORY MANAGEMENT

Malloc() and free() is dangerous in embedded real-time system as inability to obtain a single contiguous memory area due to fragmentation can be fatal (Krishnaveni and Sivakumar, 2013)". Execution time of malloc() and free() are also nondeterministic. When using dynamic memory allocation de-fragmentation of memory will be required, but that can be time consuming. To avoid the above issues, all the memory allocations have to be done initially in the program, so that there is no uncertainty about the availability of memory during runtime. In µC/OS-II, fixed-sized memory blocks are created from a partition made of a contiguous memory area as shown in **Fig. 2**. All memory blocks are the same size and the partition contains an integral number of blocks. Allocation and deallocation of these memory blocks is done in constant time and is deterministic. µC/OS-II controls the partitions with memory control blocks.

The run-time mapping form virtual to physical address is done by a hardware device and forms the memory management unit. Program Status Words (PSW) controls the order of instruction execution and contains various information about the state of a process. There are three types of PSW's namely; current PSW, new PSW and old PSW.

5.1. Virtual Memory and Benefits

Virtual memory is a hardware technique where the computer system appears to have more memory than it actually does (Gopal *et al*., 2010). This is done by time sharing the physical memory and executing a process that may not be completely in main memory. The advantages include:

- A program would no longer be constrained by the amount of physical memory that is available
- Since each program could take less physical memory, more programs could be run at the same time
- Less I/O would be needed to load or swap each user program into memory. So each user program would run faster

5.2. µC/OS-II API and its Functions

OSMemCreate(), OSMemGet(), OSMemPut() and OSMemQuery() are the main RTOS functions used in this research work. The µC/OS-II API's and their purpose is listed in **Table 2**.

5.3. Identified Hardware Setup

In this study the combination of LPC 1768 hardware and µC/OS-II kernel are used to minimize the complexity of the system (Srikanth and Samunuri, 2013)". The LandTiger V2.0 NXP LPC1768 ARM development board is a 32 bit Microprocessor used for embedded system applications. The Board has following features:

- 512KB on chip flash program memory
- 64KB SRAM for high performance CPU
- Standard JTAG test/debug interface
- Two RS 232 serial interfaces
- Two CAN bus communication interfaces
- RS 485 communication interface
- RJ45-10/100M Ethernet network interface
- DAC o/p interface and ADC i/p interface
- USB 2.0 interface
- SD/MMC card (SPI) interface
- Color LCD display interface

LPC 1768 is a high performance and Low power consumption Microprocessor. µC/OS-II is a real time multi tasking operating system kernel version 2. It is used for inter task communication and synchronization and it has the following features:

- Portable, preemptive, multitasking kernel
- It can handle 64 tasks

- It supports processors up to 64 bit
- It has deterministic execution times
- µC/OS-II is simple to use and simple to implement KERNEL

An real time operating system (µC/OS-II) fulfills the requirements of events that happens in real time. RTOS gives an efficient solution for memory management with variable sizes of memory blocks required for different processes (Wang *et al*., 2011).

6. DEVELOPMENT CYCLE

The various steps involved in memory allocation and deallocation is presented here. A new memory partition is created using OSMemCreate() and as an illustration 5 blocks with block size of 5 is created. A request for the memory blocks to remain in a loop until no more blocks are available is placed. At each stage the amount of used and free memory available in the memory block is dynamically tracked. The memory blocks in a loop are released until all the blocks are released and at each stage the amount of used and free memory available in the memory block is displayed. The sequence of steps illustrating the above is given in **Fig. 3**.

6.1. Algorithm

The algorithm for task 1 and interrupt handler is given below:

1. Get memory Status using OSMemQueryAPI
2. Display the used and free memory of the partition
3. Get memory block from partition using OSMemGet API
4. Display the used and free memory of the partition
5. If memory block is not NULL go to step 3
6. Display the used and free memory of the partition
7. Put memory block to the partition using OSMemPut API
8. Display the used and free memory of the partition
9. If used memory in the partition exceeds 0 go to step 7
10. Display the used and free memory of the partition
11. End

The task for the above algorithm is created with function App_Task1 Create() and a stock of sufficient size is created to run task properly with function OSTaskCreate(). Priority of the task is passed to API using the function APP_TASK1_PRIO.OSMemCreate(), OSMemGet(), OSMemPut() and OSMemQuery() are used to create a memory partition ,to get a block from the memory partition, to put the memory block to the partition and to query the status of the memory partition respectively.

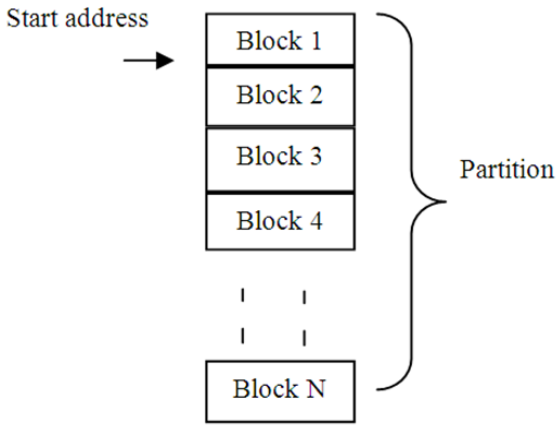

Fig. 2. Memory block partition

Table 2. µC/OS-II API's and its functions

µC/OS-II API	Functions
App_Task 1 create	Used to create Application Task1
Os Task create	Used for Task creation
Os_Task_Name_EN	Used to enable Task name
OS_Mem Create()	To create new Memory Partition
OS_Mem Get()	Get Memory and Allocate the block
OS Mem Put()	Return all the Memory blocks to a memory partition
Mem_Init()	Used for Memory initialization
App_Task_Start_PRIO	To start the Task Priority
OS_Start	Start Multitasking
OS_Mem Query	To get details about Memory

6.2. Code Implementation in C

The following are the codes for Task Creation, Task1, Memory Partition initialization and global declarations.

6.2.1. Task Creation

```
Void App TaskCreate(void)
{
CPU_INT08U os err;
Os_err = os_err;
UART0_SendString("Creating Task 1(Memory Management) with priority 58\r\n");
Os_err = OSTaskCreate((void(*)(void*))uctsk_Task1,
                      (void *)0,
                      (OS_STK*) &
                  App_Task1stk[APP_TASK_STK_SIZ
                  E-1],
                  (INT8U)
                  APP_TASK1_PRIO);
# if OS TASK NAME EN>0
```

6.2.2. Task 1

```
Static void uctsk_Task1 (void*pdata)
{
        INT8U err=0;
        Int cnt = 0;
        OS_MEM_DATA mem_data;
        pdata = pdata;
UART0_SendString("Task1(Memory Management) is Created\r\n");
UART0_SendString("\r\n****MEMORY BLOCKS ALLOCATION****\r\n\n");
```

6.2.3. Memory Partition Init and Global Declarations

```
OS_MEM*CommMem;
INT8U*msg[6];
INT8U CommBuf[5][5];
Void Mem_Init(void)
{
        INT8U err;
        CommMem=OSMemCreate(&CommBuf[0][0],5,5,&err);
}
```

6.3. Experimental Setup

USB cable of the LPC 1768 Board is connected to USB port of the system and Serial port of the LPC 1768 Board is connected to Serial port of the system The memory management program is compiled and downloaded to the LPC 1768 board By resetting the board, the allocation and de allocation of memory blocks in hyper terminal can be seen. The experimental setup is shown in **Fig. 4**.

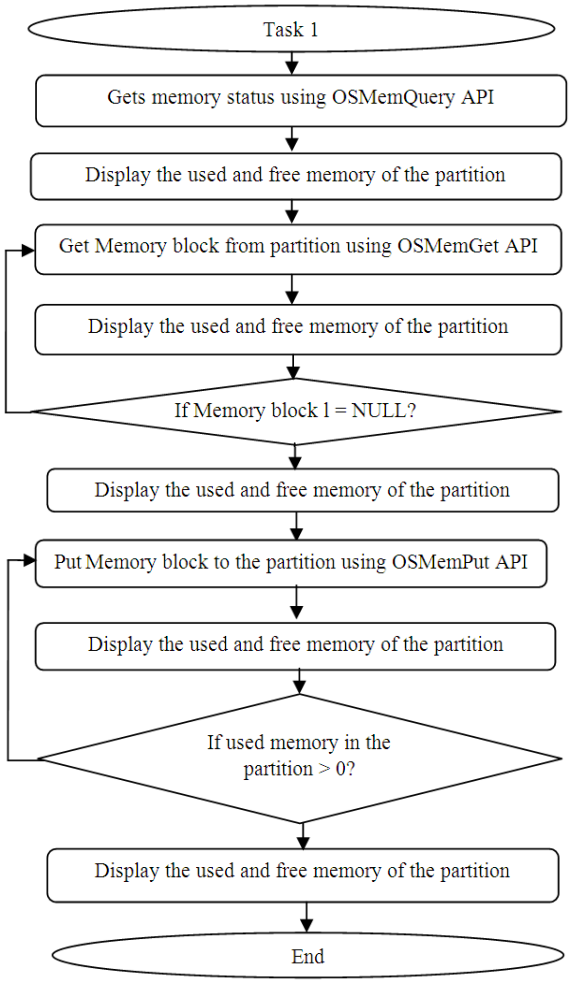

Fig. 3. Steps in memory management

7. RESULTS

Real time implementation of memory allocation is demonstrated in **Fig. 5** which shows a creation of new memory partition with block size of 5 and task1for memory block allocation, release and display of status of memory blocks. The allocation of memory blocks one by one is shown in **Fig. 6**. In which it is clearly seen that free and used memory status is updated dynamically each time a block is allocated. Similarly release of memory blocks after use is shown in **Fig. 7**, here also used and free memory status displayed each time a block is released. The results shows that memory allocation is done in real time.

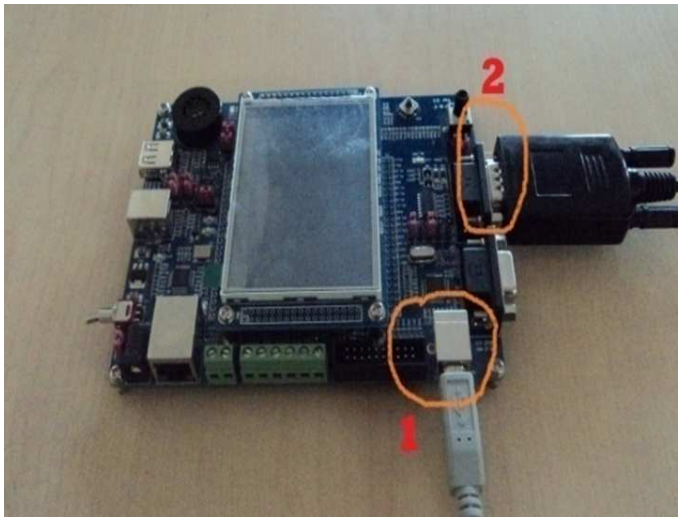

Fig. 4. Experimental setup 1. USB Cable of LPC 1768(Connected to USB Port of PC), 2. Serial port of LPC 1768(Connected to PC Serial port)

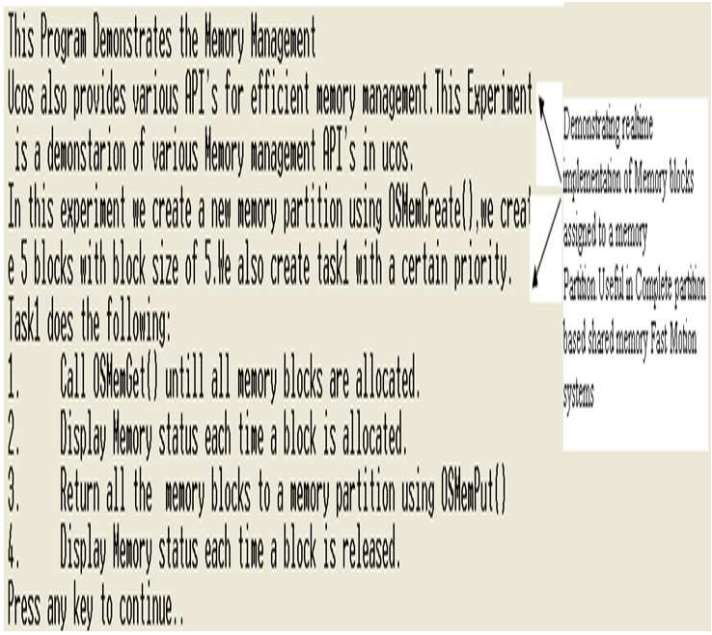

Fig. 5. Real time implementation of memory management

Fig. 6. Memory blocks allocation

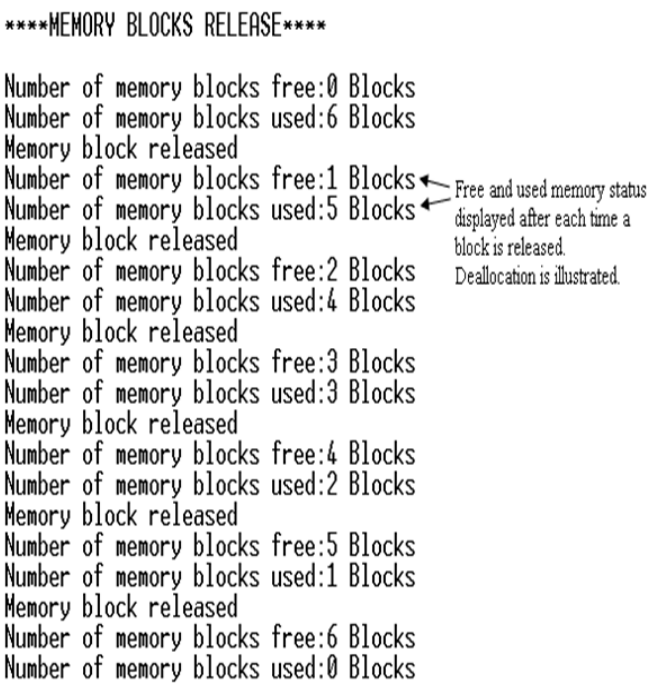

Fig. 7. Memory blocks deallocation

8. DISCUSSION

In reported algorithms, the search time increases particularly when the memory available for allocation is large. In this study, the allocator rounds up the requested size to an allowable value and permits to select a larger block from another vacant list. Masmano *et al*. (2008) proposed a TLSF dynamic memory allocator for a real time application. TLSF is a constant time, good fit allocator. In this suitable list is found by processor bit instructions and word size bitmaps. A comparison is being made on timing performance with other allocators. The results presented by author's shows that percentage of fragmentation is very high in buddy allocator compared with TLSF allocator. The results of proposed buddy allocator coded in Keil C language shows vast improvement in minimization of fragmentation and also it achieves bounded execution time without wasting memory space. In this study memory is considered as a resource in real time application and its efficient management is demonstrated.

9. CONCLUSION

In this study, a buddy allocator dynamic memory allocation algorithm for efficient memory management in real time environment has been implemented. In this method when a memory block is allocated or released then the free and used memory blocks (displayed in the hyper terminal) can be easily found by simple address computation. This method enables easy allocation and deallocation of memory blocks, minimizes the fragmentation, ensures optimal utilization of memory and good locality among memory blocks. Although, this method is optimal, it still has a limitation that it has a residual fragmentation and this could be due to restricted block sizes. Future direction of research shall be on overcoming this limitation and design a memory allocator that solves the problem of allocation based on the task behavior and use of real time concepts in message box and mail queue techniques.

10. REFERENCES

Anbumozhi, S. and P.S. Manoharan, 2014. Performance analysis of high efficient and low power architecture for fuzzy based image fusion. Am. J. Applied Sci., 11: 769-781. DOI: 10.3844/ajassp.2014.769.781

Golzari, S., S. Doraisamy, M.N. Sulaiman and N. Udzir, 2011. An efficient and effective immune based classifier. J. Comput. Sci., 7: 148-153. DOI: 10.3844/jcssp.2011.148.153

Gopal, B., R. Beg and P. Kumar, 2010. Memory management technique for paging on distributed shared memory framework. Int. J. Comput. Sci. Inform. Technol., 2: 141-153.

Halawani, M.S. and N.A. Al-Romema, 2010. Memory storage issues of temporal database applications on relational database management systems. J. Comput. Sci., 6: 296-304. DOI: 10.3844/jcssp.2010.296.304

Krishnaveni, N. and G. Sivakumar, 2013. Survey on dynamic resource allocation strategy in cloud computing enviornment. Int. J. Comput. Applic. Technol. Res., 2: 731-737.

Manikandan, G. and S. Srinivasan, 2013. An efficient algorithm for mining spatially co-located moving objects. Am. J. Applied Sci., 10: 195-208. DOI: 10.3844/ajassp.2013.195.208

Masmano, M., I. Ripoll, P. Balbastre and A. Crespo, 2008. A constant-time dynamic storage allocator for real-time systems. Real-Time Syst., 40: 149-179. DOI: 10.1007/s11241-008-9052-7

Patra, S.K., B.K. Pattanayak and B. Puthal, 2013. Javascript interpreter using non recursive abstract syntax tree based stack. Am. J. Applied Sci., 10: 403-413. DOI: 10.3844/ajassp.2013.403.413

Porwal, S. and H. Mittal, 2013. An efficient memory management technique for smart card operating system. Int. J. Comput. Technol., 5: 124-129.

Revathy, M. and R. Saravanan, 2014. Performance analysis of high efficiency low density parity-check code decoder for low power applications. Am. J. Applied Sci., 11: 558-563. DOI: 10.3844/ajassp.2014.558.563

Srikanth, K. and N. Samunuri, 2013. RTOS Based priority dynamic scheduling for power applications through DMA peripherals. Int. J. Eng. Trends Technol., 4: 3660-3664.

Wang, Y.Y., R.B. Sallie D.A. Chang, W.L. Silver and E.R. Liman, 2011. A TRPA1-dependent mechanism for the pungent sensation of weak acids. J. Cell Biol., 137: 493-505. DOI: 10.1085/jgp.201110615