SoC drawer: Shared resource management

Focus on memory and I/O

Sam Siewert (Sam.Siewert@Colorado.edu), Adjunct Professor, University of Colorado

Summary: The goal of a system-on-a-chip (SoC) is to provide a single-chip system, and therefore SoC resource analysis and sizing is critical. Failure to properly size processing, memory, or I/O needed by software services can kill an SoC project. But all too often, SoC design analysis focuses on processing at the expense of memory or I/O sizing. And even when memory and I/O are sized properly, efficient use of these resources by software services can still be tricky. Any mis-sizing or mismanagement of memory and I/O on an SoC can at the least cause significant project delay and rework. This article examines sizing estimation and resource sharing pitfalls that the system architect should know well.

View more content in this series

Date: 21 Feb 2006
Level: Introductory
Activity: 716 views
Comments: 0 (Add comments)


Average rating (based on 3 votes)

This article is the fifth in the SoC drawer series. The series aims to provide system architects with a starting point and some tips to make SoC design easier. The first article in this series introduced the concept of a *resource cube*. Once hardware design decisions have been made, based upon layout, pin count, power and thermal factors, and hardware functionality, firmware and software engineers are left with a resource space defined by processing, memory, and I/O.

With reconfigurable SoCs, decisions about whether functionality ought to be implemented by hardware state machines or software functions and services can be made late in the game. However, at some point, almost every SoC design will include software services for upgrade flexibility so that functionality can be upgraded by customers. Most designers will focus on whether the SoC has sufficient processing to host the software functions and services to meet throughput and real-time requirements; but all too often, they'll overlook important memory and I/O resource considerations.

This article examines some simple design rules for estimating memory and I/O requirements that you should keep in mind during sizing and then reviews common memory and I/O resource usage pitfalls that can delay software development and testing. Proper sizing of the software resource space (that is, of the resource cube) and efficient use of those resources will minimize such risk to schedules.

The first cut at resource sizing

An SoC ultimately must host an application and provide a set of features that define its purpose. Applications and their features can be very broad, but often, upon closer examination, the designer will notice that many SoCs will have a set of features that have common I/O, CPU, or memory requirements. The degree to which an application uses I/O, CPU, and memory is most often not balanced. Some SoC applications need high-performance I/O and not so much CPU or memory. Or perhaps an SoC might require significant CPU, but not so much I/O bandwidth or memory. Finally, an SoC application might emphasize storage and therefore working memory and non-volatile memory resources.

Analysis of basic SoC requirements can help with early sizing. At the highest level, an SoC can be characterized by the degree to which it includes each of the following features that emphasize use of a specific resource:

- **Data movement:** Examples of systems with high data movement rates include protocol engines, network processors, switches, audio/video streaming applications, and routers.
- **Computation:** Examples of computationally intensive systems include those dedicated to image processing, cryptography, codecs, simulation, numerical methods, planning, and constraint optimization.
- **Data storage:** Examples of storage-oriented systems include digital assistants, MP3 players, cell phones, and system management information bases.

An SoC could of course provide a combination of these three basic features. Most often, though, one of these features is more significant than the other two and can drive decisions about a system's memory, I/O, and processing requirements significantly. For instance, an SoC that is principally a data mover should include multiported memory for DMAs (direct memory access) into and out of store and forward buffers, low-latency memory to track data unit context, and carefully designed DMA channels and bus interfaces to ensure that throughput and latency goals can be achieved. By comparison, a compute-intensive SoC may have low-bandwidth I/O with significant computation performed in cache. For the data-mover SoC, tightly coupled memory with low latency accessed by the processor that can simultaneously be accessed by DMAs is critical. For the compute-intensive SoC, cache design and the speed of the processor core would be more important. A data store SoC by comparison might not have significant I/O or processing requirements, but rather significant volatile and non-volatile memory requirements.

Most SoCs will be designed with aspects of all of these design philosophies in mind, but you can save time and lower risk by characterizing the problem your SoC is intended to solve so as to provide focus on key resources. All too often, systems are designed with unbalanced resources such that I/O latency, memory bounds, or processing becomes the bottleneck for performance.

Processor sizing

Many architects put too much emphasis on processor sizing. It is an important concept, and a poorly sized processor can be an SoC bottleneck, especially for compute-intensive SoCs. However, keep in mind that an SoC with a poorly matched memory design or high memory-mapped I/O (MMI/O) latency will stall processing and waste cycles. Ideally, a processor should be sized not only based upon clock rate or benchmark MIPS (millions of instructions per second), but rather on efficiency in execution of services.

A better measure than clock rate or MIPS is CPI (clocks per instruction), a metric sometimes expressed as IPC (instructions per clock). An SoC's IPC should be high (and the CPI should be correspondingly low). For a simple pipelined processor that does not include superscalar or simultaneous multithreading features, 1.0 is a good target CPI, though processor CPIs are often much higher. A processor with a CPI of 2.0 is wasting half of its processing capability. Often, stall cycles result from wait states incurred while application code accesses memory or MMI/O, or while it waits for the completion of DMA transfers. A high CPI indicates unbalanced resource design and/or inefficient use of resources by the software. Rather than trying to incorporate the fastest processor cores, you should aim to design a correctly sized processing complex that is well matched to memory access latency, I/O latency, and efficient execution of algorithms for data processing.

Increasing use of virtual memory in embedded systems

Virtual memory has a history of providing UNIX®, Windows®, and other multiuser desktop systems with memory hierarchy usage flexibility and expandability. It has also helped protect programs from each other. Embedded systems are starting to go beyond simple protection domains and direct physical address mapping to provide VM (Virtual Memory) memory maps. A VM mapping of memory affords run-time (or at least initialization-time) flexibility in how physical memory resources are partitioned, allocated, and accessed. One very simple example is double mapping memory as both cacheable and non-cacheable. The double mapping allows application code to access the same memory two different ways. Accessing memory without possibility of cached data is a coherency assurance for device driver interfaces. However, an application may want to access the same data with cacheability so that processing of data previously transferred by a driver can also be processed in cache.

Memory sizing

Most often, memory sizing is bean counting -- estimating of the size and number of key data structures and buffers needed by software. Data movers providing store and forward functionality will need significant numbers of buffers and I/O-related context. If external memory can be added to the system, it can provide an out for errors in early estimation. However, for many SoCs, a single-chip solution is a system requirement or goal, so external memory parts are unacceptable.

Another trick to increase capacity -- and, perhaps, to improve performance -- is to lock L1 or L2 data/code caches. Memory capacity, access latency, concurrent access through multiporting, and total memory bandwidth are all critical characteristics that must be reviewed during hardware design to ensure that software designers are not left with a memory resource problem. Significant planning by software and firmware engineers to provide flexibility in mapping code, data, heaps, stack, and read-only data should be conducted very early in the design process so that the logical mapping to the hardware physical hierarchy is as mutable as possible. The ability to remap code segments, for example, can alleviate performance problems like cache thrashing. A set associative cache will work more efficiently when code is spread out in memory in multiple segments rather than just concentrated in one.

I/O sizing

Data movers will live or die based upon I/O sizing decisions that determine latency, total bandwidth, and the degree to which interconnections are non-blocking. The use of DMA channels for high-bandwidth data movers is fundamental, and DMA queues should include flexible and easily observed status features, including:

- **Setting for high/low water interrupts:** Software fill/spill logic must keep DMA queues between half full and full to saturate channels.
- **Low latency status:** Quick status checks by software enable rapid servicing with minimal CPU stalls.
- **Notification by exception:** Status and interrupts on errors or for periodic synchronization between hardware and software state can reduce costly software/hardware interaction.

Managing shared resources

Shared resource management becomes complicated in systems with priority preemptive CPU scheduling with multiple services. Priority preemptive scheduling of multiple service threads or tasks is often used by programmers to provide low-latency response to service requests and to handle errors and exceptions. Sharing a CPU with priority preemption is simple when secondary resources like memory or I/O are not required in addition to the CPU. Obtaining the CPU by priority dispatch and also obtaining required memory and I/O resources for service execution complicates multiservice systems considerably and introduces potentially catastrophic failure scenarios.

Any CPU that hosts multiple demand-driven application services will have to host multiple priority preemptive threads of execution. Multiple services multiplexing a single CPU will also inevitably need to share memory and I/O resources. Again, the best possible approach is to allocate hardware resources such that the memory and I/O sharing can be avoided. Good software and firmware programmers know that, in situations where resources must be shared, one must exercise caution to avoid data corruption, bus contention, and race conditions. However, even the best programmers can be caught by some of the more esoteric multiservice resource pitfalls that the next section discusses.

When bad things happen to good programmers

Perfectly written and even well-tested code can fail dramatically on an SoC in the field with only slight divergences from test workloads, timing, or input data. Three rather infamous problems can crop up due to resources being shared by more than one thread of execution (that is, by more than one software service). They are:

- **Unbounded priority inversion:** A higher priority thread is held off by execution of a lower priority thread for an indeterminate period of time.
- **Resource deadlock:** All threads involved fail to make any further progress due to circular resource wait.
- **Resource livelock:** A deadlock is detected and broken, but leads to subsequent deadlock repeats due to unfortunate timing.

All three problems are the result of shared memory or I/O resources. All three can be avoided. One simple method of avoidance is to provide richer hardware resources so that multiple software services don't need to share memory or I/O resources; however, this is often impossible due to system requirements or prohibitive costs.

Priority inversion and deadlock can kill "five nines"

Both priority inversion and deadlock in software often require recovery through a hardware watchdog timer and a reboot. Rebooting means loss of service time. A "five nines" system (one with 99.999% uptime) can't be out of service for more than five minutes per year, so rebooting to recover directly affects a system's ability to meet this availability measure. Quicker recovery methods can help, but avoiding the necessary conditions for deadlock and priority inversion is a much better design approach.

Figure 1. Priority inversion

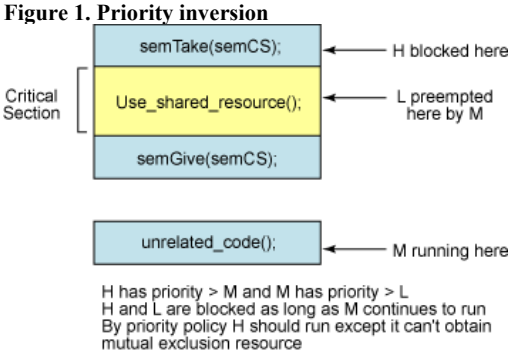

Figure 1 shows a scenario that can cause an unbounded inversion. Here we have a high-priority service (H), a middle-priority service (M), and a low-priority service (L). The necessary condition for priority inversion are:

- **Three or more services in the system sharing a CPU**
- **Two services with priority H and L accessing a shared resource.** The resource must have mutually exclusive access to avoid data corruption or contention.
- **prio(H) > prio(M1) > prio(M2) > ... > prio(Mn) > prio(L).** Thus, all M services with priorities between H and L are not involved in the shared resource, but can preempt L.

If the software service designer can avoid any of the above conditions, then an unbounded priority inversion will not arise. For example, if there are only two tasks, then H will be blocked only for the duration of the critical section -- a well-bounded inversion. Furthermore, if the priorities can be adjusted for the duration of the critical section such that L's priority is amplified temporarily, then the third condition is not met and the issue avoided. Avoidance is the key to three protocols that prevent the third condition from persisting. These are:

- **Priority Inheritance Protocol:** L is loaned H's priority for the duration of the shared resource critical section.
- **Highest Locker or Priority Ceiling Emulation Protocol:** L's priority is amplified to the maximum priority of all of the threads involved in the critical section. This priority is known as the highest locker or priority ceiling for this semaphore.
- **Priority Ceiling Protocol:** A system priority ceiling (known simply as system ceiling) based upon the maximum priority for ALL semaphore highest lockers is maintained by the operating system. Any holder of the system ceiling or higher priority can access its critical section and any other protected resource (since it is safe from deadlock). If a task is blocked, then the task in the critical section will inherit the blocked task's priority.

The VxWorks code in Listing 1 implements the necessary conditions to set up an unbounded priority inversion. The `prio_invert` function can be executed with the wind shell; you can specify the number of M interference seconds, as well as whether or not VxWorks should employ priority inheritance to limit the inversion duration.

Try calling `prio_invert` with arguments of 10 and 0 so that the H task will be blocked for more than 10 seconds while the M idle task spins and preempts the L task, which will be stuck in the `msgSem` guarded critical section. Next, call `prio_invert` with arguments of 10 and 1. This way, M interference will persist once again for 10 seconds; however, given the priority inheritance protocol, L will now complete the critical section using H's priority and will allow H to unblock, preempt M, and complete much sooner. The priority inheritance mechanism in the VxWorks semaphore breaks one of the necessary conditions by reordering priorities temporarily. The duration that H blocks with VxWorks priority inheritance is bounded by how long it takes L to finish execution of the critical section now rather than by the arbitrarily long run time of M.

The code in Listing 1 can be run on VxWorks PowerPC® systems or on the Wind River VxSim simulation system, which runs on any Tornado host, including Windows, Linux®, or Sun Solaris. You can download the C code from the Downloads section below.

Listing 1. VxWorks priority inversion example

```
#include "stdio.h"
#include "stdlib.h"
#include "semLib.h"
#include "errnoLib.h"

#define HIGH_PRI/O_SERVICE 1
#define MIDDLE_PRI/O_SERVICE 2
#define LOW_PRI/O_SERVICE 3

SEM_ID   msgSem;

int CScount = 0, DoneCount = 0, clkRate, lowStart, highStart;
int runInterference=0;

LOCAL void idle (int arg1, int arg2, int arg3, int arg4, int arg5, int arg6, int arg7, int arg8, int arg9,
                 int arg10)
{
  while(runInterference); exit(0);
}


LOCAL void semPrinter (int id, int msgs, int arg3, int arg4, int arg5, int arg6, int arg7, int arg8, int arg9,
                       int arg10)
{
  int i, Lock;

  if(id == LOW_PRI/O_SERVICE) printf("Low prio task requesting shared resource at %d ticks\n", (lowStart=tickGet()));
  else printf("High prio task requesting shared resource at %d ticks\n", (highStart=tickGet()));

  /* Take semaphore and wait for chance to print */
  if (semTake (msgSem, WAIT_FOREVER) != OK) {printf ("Error taking semaphore in task %d \n", id); exit(-1)};

  CScount++;

  if(id == LOW_PRI/O_SERVICE) {printf("Low prio task in CS\n"); fflush(stdout)};
  else {printf("High prio task in CS\n"); fflush(stdout)};

  /* camp out in CS long enough to get stuck here */
  taskDelay(clkRate);

  for (i=0; (i<msgs); i++)
  {
    printf ("\nPrint msg # %d from task %d", i, id); fflush(stdout);

    if(id == LOW_PRI/O_SERVICE) {printf(" low prio task\n"); fflush(stdout)};
    else {printf(" high prio task\n"); fflush(stdout)};
  }

  if (semGive (msgSem) != OK) {printf ("Error giving semaphore in task %d \n", id); exit(-1)};

  if(id == LOW_PRI/O_SERVICE) printf("Low prio task done in %d ticks\n", (tickGet()-lowStart));
  else printf("High prio task done in %d ticks\n", (tickGet()-highStart));

  DoneCount++;
  exit(0);
}


void prio_invert (int intfCount, int inversionSafe)
{
  int      p1Task, p2Task, p3Task, i;

  CScount=0; DoneCount=0; clkRate=sysClkRateGet();
  printf("%d ticks per second\n", clkRate);

  /* Create print to screen semaphore */
  if(inversionSafe)
  {
    if ((msgSem = semMCreate(SEM_Q_PRI/ORITY | SEM_INVERSI/ON_SAFE)) == NULL)
    {
      printf ("Error creating print to screen semaphore\n"); exit(-1);
    }
    printf ("\nInversion Safe:Print to screen critical section semaphore created\n"); fflush(stdout);
  }

  else
  {
    if ((msgSem = semMCreate(SEM_Q_PRI/ORITY)) == NULL)
    {
      printf ("Error creating print to screen semaphore\n"); exit(-1);
    }
    printf ("\nUnsafe:Print to screen critical section semaphore created\n");
  }
```

```
  /* Spawn task to print message to the screen */
  if ((p3Task = taskSpawn ("lowprio_printer", 100, 0, (1024*8), (FUNCPTR) semPrinter,
  LOW_PRI/O_SERVICE,1,0,0,0,0,0,0,0)) == ERROR)
  {
    printf ("Error creating lowprio_printer task\n"); exit(-1);
  }
  printf ("Low priority print to screen task spawned\n");

  /* spin wait while T3 has not yet entered critical section */
  while(CScount < 1) taskDelay(clkRate/10);

  /* Spawn task to print message to the screen */
  if ((p1Task = taskSpawn ("highprio_printer", 50, 0, (1024*8), (FUNCPTR) semPrinter,
  HIGH_PRI/O_SERVICE,1,0,0,0,0,0,0,0)) == ERROR)
  {
    printf ("Error creating highprio_printer task\n"); exit(-1);
  }
  printf ("High priority print to screen task spawned\n");

  /* Spawn an interfering task */
  runInterference=1;
  if ((p2Task = taskSpawn ("interference", 75, 0, (1024*8), (FUNCPTR) idle, MIDDLE_PRI/O_SERVICE,0,0,0,0,0,0,0,0))
     == ERROR)
  {
    printf ("Error creating idle interference task\n"); exit(-1);
  }
  printf ("Medium priority interference task spawned\n");

  /* interference duration */
  for(i=0;i<intfCount;i++) taskDelay(clkRate);

  /* stop the interference to let low prio finish */
  runInterference=0; printf ("Stopped interference\n");

  /* allow the delayed task to finish up */
  while(DoneCount < 2) taskDelay(clkRate/10);

  printf ("\nAll done\n");

  /* shutdown the system resources */
  semDelete (msgSem);
  exit(0);
}
```

Figure 2. Deadlock

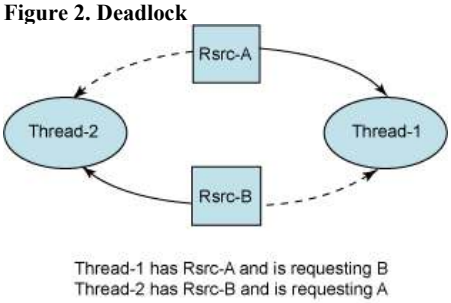

Figure 2 shows a circular wait, which can arise from shared resources used in common by two or more threads. Listing 2 provides a POSIX pthread example that can be run on Linux or Mac OS Darwin (Mach) PowerPC systems. (You can download the C code along with the appropriate makefiles in the downloads section below.) If you run the deadlock executable with the `safe` argument and based upon the `pthread_join` synchronization, the system will not deadlock. But if you run the deadlock executable with the `unsafe` argument, the system will now always deadlock due to lack of thread synchronization and delays between obtaining resources A and B; you'll need to send a Ctrl-C interrupt to the operating system to break the deadlock. Without the synchronization between pthreads and without the delay, the system might or might not deadlock -- a race condition determines whether a circular wait will evolve.

At least three solutions are commonly used to recover systems from deadlock:

- **Avoidance:** Multiresource acquisitions are strictly serialized to preclude circular wait.
- **Detection by monitor with random back-off:** Failure to make progress is detected by a software or hardware monitor, and resources are dropped and acquisition is delayed for a randomly increasing amount of time.
- **Detection by hardware watchdog:** Failure to provide service is detected by a hardware timer and the system is rebooted to start recovery.

Listing 2. POSIX pthreads deadlock example

```
#include "pthread.h"
#include "stdio.h"
```

```
#include "sched.h"
#include "time.h"

#define NUM_THREADS 2
#define THREAD_1 1
#define THREAD_2 2

pthread_t threads[NUM_THREADS];
pthread_mutex_t rsrcA, rsrcB;
volatile int rsrcACnt=0, rsrcBCnt=0, noWait=0;

void *grabRsrcs(void *threadid)
{
   if((int)threadid == THREAD_1)
   {
     printf("THREAD 1 grabbing resources\n");
     pthread_mutex_lock(&rsrcA);
     rsrcACnt++;
     if(!noWait) usleep(1000000);
     printf("THREAD 1 got A, trying for B\n");
     pthread_mutex_lock(&rsrcB);
     rsrcBCnt++;
     printf("THREAD 1 got A and B\n");
     pthread_mutex_unlock(&rsrcB);
     pthread_mutex_unlock(&rsrcA);
     printf("THREAD 1 done\n");
   }
   else
   {
     printf("THREAD 2 grabbing resources\n");
     pthread_mutex_lock(&rsrcB);
     rsrcBCnt++;
     if(!noWait) usleep(1000000);
     printf("THREAD 1 got B, trying for A\n");
     pthread_mutex_lock(&rsrcA);
     rsrcACnt++;
     printf("THREAD 2 got B and A\n");
     pthread_mutex_unlock(&rsrcA);
     pthread_mutex_unlock(&rsrcB);
     printf("THREAD 2 done\n");
   }
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   int rc, safe=0;

   rsrcACnt=0, rsrcBCnt=0, noWait=0;

   if(argc < 2) printf("Will set up unsafe deadlock scenario\n");
   else if(argc == 2)
   {
     if(strncmp("safe", argv[1], 4) == 0) safe=1;
     else if(strncmp("race", argv[1], 4) == 0) noWait=1;
     else printf("Will set up unsafe deadlock scenario\n");
   }
   else printf("Usage: deadlock [safe|race|unsafe]\n");

   // Set default protocol for mutex
   pthread_mutex_init(&rsrcA, NULL);
   pthread_mutex_init(&rsrcB, NULL);

   printf("Creating thread %d\n", THREAD_1);
   rc = pthread_create(&threads[0], NULL, grabRsrcs, (void *)THREAD_1);
   if (rc) {printf("ERROR; pthread_create() rc is %d\n", rc); perror(NULL); exit(-1);}
   printf("Thread 1 spawned\n");

   if(safe) // Make sure Thread 1 finishes with both resources first
   {
     if(pthread_join(threads[0], NULL) == 0) printf("Thread 1: %d done\n", threads[0]);
     else perror("Thread 1");
   }

   printf("Creating thread %d\n", THREAD_2);
   rc = pthread_create(&threads[1], NULL, grabRsrcs, (void *)THREAD_2);
   if (rc) {printf("ERROR; pthread_create() rc is %d\n", rc); perror(NULL); exit(-1);}
   printf("Thread 2 spawned\n");

   printf("rsrcACnt=%d, rsrcBCnt=%d\n", rsrcACnt, rsrcBCnt);
   printf("will try to join CS threads unless they deadlock\n");

   if(!safe)
   {
     if(pthread_join(threads[0], NULL) == 0) printf("Thread 1: %d done\n", threads[0]);
```

```
    else perror("Thread 1");
  }

  if(pthread_join(threads[1], NULL) == 0) printf("Thread 2: %d done\n", threads[1]);
  else perror("Thread 2");

  if(pthread_mutex_destroy(&rsrcA) != 0) perror("mutex A destroy");
  if(pthread_mutex_destroy(&rsrcB) != 0) perror("mutex B destroy");

  printf("All done\n");
  exit(0);
}
```

Differing resource management philosophies

The idea of embedding Linux for real-time applications has become very fashionable lately. However, it seems that while Linux provides an excellent framework for multiuser resource management, it is still a bit green when it comes to providing mechanisms for real-time resource management by multiple priority-based threads. This problem is no doubt solvable, although it seems that the Linux community has been slow to adopt well-known mechanisms to do this.

Building a single kernel that provides fair, highly available, scalable services for best-effort applications and also provides highly deterministic, low-latency, simple, and preemptable services is a non-trivially complex task. Historically, real-time operating system (RTOS) kernels have been much simpler than UNIX kernels and have placed much more responsibility on the shoulders of the programmer to manage resources. Embedded Linux is attractive because it seems to offer developers the essence of having one's cake and eating it too: it has all the convenience and safety of a desktop OS, yet the hope is that, with some extensions, it can provide sufficient features to also supply all the predictability and application-specific resource management features needed by embedded programmers.

My experience is that Linux can be embedded nicely (I've done this once) and can work in this space. But if you're taking this route, be aware that you will need an in-depth understanding of the Linux kernel. When I began this article, it was my intention to provide the priority inversion example for both VxWorks and Linux. Example code can be downloaded for both, but the careful reader will notice that the POSIX mechanisms for priority ceiling and priority inheritance are not well supported by Linux without the use of kernel patches. I hope that this does not discourage readers from using embedded Linux, but rather encourages participation in the ongoing projects to provide a better real-time embedded Linux kernel.

Conclusion

Resource management on an SoC should be driven by a clear understanding of requirements, starting with the basic purpose of the system. Sharing hardware resources and multiple threads or services allocated to multiple processor cores increases complexity and risk. Avoidance of well-known resource sharing pitfalls is always the best option for any system. If avoidance can't be assured, then detection and recovery methods are required and will complicate designs. If at all possible, sharing of resources should be avoided, and when multiple resources must be shared, interaction should be well synchronized, brief, and well tested.

Downloads

VxWorks resource example downloads

- prio_invert.c

Linux and Mac OS X (Darwin) resource example downloads

- deadlock.c
- pthread.c
- Makefile
- Makefile.darwin
- Makefile.linux

Resources

Learn

- The resource cube was introduced in the first article in this series.
- Deadlock can be avoided, detected and broken, or can be recovered from by a watch-dog timer. The *IBM Research Journal* article "Avoiding deadlock in multitasking systems," J. W. Havender (1968) discusses the necessary conditions for deadlock and avoidance.
- Network processors and data movers coordinating shared buffer access can be particularly susceptible to deadlocks, as discussed in the *IBM Research Journal* article "Determining deadlock exposure for a class of store and forward communication networks," V. Ahuja (1980).

- Oddly enough, there seems to be confusion in the Linux community regarding how to best provide priority inversion avoidance mechanisms. Some in the open source community see priority inheritance as cumbersome, a view expressed in "Against priority inheritance," Victor Yodaiken (July 2002; document is in PDF format). The position taken in this paper led to a rebuttal: "Priority inheritance: The real story," Doug Locke (LinuxDevices.com, July 2002). I'll have to admit, given a choice between priority inheritance and priority ceiling, I'd agree that ceiling seems to be an easier mechanism to implement, but also agree with Locke's assessment that, in the end, the issue of unbounded inversion must be understood and that avoidance mechanisms shouldn't be considered panaceas to be used carelessly.

- "What happened on Mars," Mike Jones (Microsoft Research, December 1997) discusses what happened when the Mars Pathfinder spacecraft exhibited unbounded priority inversion on final approach. The mission was ultimately saved by patching software so that a semaphore was re-initialized `INVERSION_SAVE`, exactly as is done in the example code in this article. This infamous service priority glitch has helped to stress the importance of understanding esoteric resource issues like unbounded priority inversion, and has prompted RTOS and Linux developers alike to consider offering avoidance mechanisms such as priority inheritance and priority ceiling protocols.

- Find more history and information on the POSIX Standards on Wikipedia, including the POSIX 1003.1b real-time extensions.

Get products and technologies

Get products and technologies

- The Linux community is engaging in several projects to add priority inversion protection, including the RT extensions project and the FUSYN project. The latter provides patches to the 2.6 kernel that add POSIX-1003.1b compliant features, including priority inheritance and priority ceiling protocol mutex semaphores. The FUSYN project is part of the Carrier Grade Linux project.

- Find more information on the VxWorks simulator, Wind River's RT Linux, and PowerPC embedded VxWorks used to prepare and test examples found in this article at Wind River Systems. Numerous real-time Linux distributions are available from sources including: Concurrent RedHawk Linux, LynuxWorks BlueCat Linux, FSMLabs RTLinux, TimeSys Linux, and RT Linux projects too numerous to list here, listed by the Real Time Linux Foundataion.

About the author

Dr. Sam Siewert is an embedded system design and firmware engineer who has worked in the aerospace, telecommunications, and storage industries. He also teaches at the University of Colorado at Boulder part-time in the Embedded Systems Certification Program, which he co-founded. His research interests include autonomic computing, firmware/hardware co-design, microprocessor/SoC architecture, and embedded real-time systems.