IO-Flexpages in L4/Linux

Bernhard Kauer

`Bernhard.Kauer@inf.tu-dresden.de`

November 12, 2002

Contents

1 Ziel

Das Ziel dieses Praktikums ist die Implementierung von *IO-FlexPages* in *L4/Linux* der Version 2.2.22.

2 Einführung

Der Zugriff auf periphäre Geräte kann in der IA-32 Architektur durch Memory Mapped IO oder durch IO-Ports erfolgen. Diese IO-Ports bilden einen eigenen Adressraum, auf den mit den Befehlen `in/out` zugegriffen werden kann.

Da die unsachgemäße Verwendung von IO-Ports zu Datenverlust und Systemabstürzen führen kann, muss der Zugriff auf diese durch das Betriebssystem geschützt werden. Bei Memory Mapped IO kann dies durch die Schutzmechanismen des Virtual Memory erfolgen. Diese stehen für die IO-Ports leider nicht zur Verfügung.

Für die Abstraktion eines Memory-Adressraumes werden in L4 *Flex-Pages* ("Flexibel große Seiten") verwendet, die als Datentyp des Mapping-Konzeptes dienen. Die Einführung von *IO-FlexPages* erlaubt die Ausdehnung dieses Konzeptes auf den IO-Adressraum. Dabei steht eine *IO-FlexPages* für einen bestimmten IO-Portbereich. Durch Einbinden einer *IO-FlexPages* in den Adressraum der Tasks wird der Zugriff auf diesen Bereich freigegeben. Die Freigabe kann feingranular (byteweise) erfolgen, dennoch sind nur 1:1 Mappings möglich.

3 Ausgangszustand

Einige Linuxtasks müssen auf externe Geräte und damit auf IO-Ports zugreifen. Da zur Zeit keine Unterscheidung zwischen den erlaubten und unerlaubten Zugriffen möglich ist, darf jede *L4/Linux*-Task (also auch jeder Nutzer-Prozess) auf alle 64K Ports zugreifen und die Interrupts sperren.

Da dies ein erhebliches Sicherheitsrisiko darstellt, ist ein Implementieren der *IO-FlexPages* nötig.

3.1 Der IOPL

Für den Schutz des Betriebssystemkernes vor Anwenderprogrammen existiert der *CPL* (Current Privilege Level). Dieser legt in 4 Abstufungen die Privilegien des aktuellen Programmes fest, wobei 0 für den Kernmodus verwendet wird und 3 für die Anwenderprogramme.

Durch den *IOPL* (IO Privilege Level) wird die Maximal-Stufe des *CPL* (Current Privilege Level) festgelegt, die ein Thread haben darf, um auf alle IO-Ports zuzugreifen und `cli/sti` ausführen zu können. *L4/Linux*-Tasks werden ohne *IO-FlexPages* mit einem IOPL von 3 ausgeführt.

Ein direktes Setzen des *IOPL* aus einer Task heraus ist nicht möglich. Erst durch das Anfordern aller IO-Ports kann in L4 der *IOPL* erhöht werden. Dies geschieht lazy, d.h. erst nach einem Pagefault wird der *IOPL* geändert.

3.2 Die IOPBM

Die *IOPBM* (IO Privilege Bitmap) legt per Bitmaske diejenigen Ports fest, auf die trotz *IOPL*-Verletzung zugegriffen werden darf. Eine vollständige *IOPBM* belegt 8k, wobei auch kleinere Bitmap's existieren können. Diese *IOPBM* wird für jede Task in L4 geführt, die *IO-FlexPages* verwendet. Damit kann der Prozessor den Schutz einzelner Ports gewährleisten.

3.3 cli/sti

Die ASM Befehle `cli/sti` sperren bzw. geben die maskierbaren Interrupts frei, indem sie das *IF* (Interrupt Flag) löschen bzw. setzen. Diese Befehle werden z.Z. vor allem an folgenden Stellen in *L4/Linux* eingesetzt:

- bei der Emulation(L4_TAMED) von `cli/sti` in den Funktionen *__global_cli()* und *__global_sti()* zur Synchronisation einer Warteschlange[1]
- in *l4_idle()*[2]
- beim Auslesen des RTC[3]
- beim Programmieren der PIC's[4]

Eine Verwendung dieser Befehle birgt ein erhebliches Sicherheitsrisiko. Da keine Timerinterrupts mehr generiert werden, findet kein Scheduling statt. Wenn dann der laufende Thread blockiert, führt dies zum Stillstand des gesamten Systems.

[1] linux22/arch/l4/x86/kernel/irq.c
[2] linux22/arch/l4/x86/kernel/dispatch.c
[3] linux22/arch/l4/x86/kernel/time.c
[4] linux22/arch/l4/x86/kernel/irq.c

3.4 popf

Weiterhin existiert der Befehl `popf`, der alle Flags (auch das *IF*) vom Stack neu lädt. Ein Ändern des *IF* ist auch hier nur möglich, wenn der *IOPL* größer oder gleich dem *CPL* ist. Fehlen die Rechte, so ignoriert der Prozessor die IO-Flags beim Setzen, anstatt eine Exception auszulösen.

3.5 iret

Die Flags können auch vom `iret`-Befehl zurückgeladen werden. Dieser wird im L4-Kern benutzt, um eine Task im aktuellen User-Kontext fortzusetzen.

4 L4/Linux

4.1 Pagefaults

Zugriffe auf nicht vorhandenen (eingetragenen) virtuellen Speicher erzeugen Seitenfehler (Pagefaults). Diese Seitenfehler werden von *L4* an den Pager des aktuellen Threads per IPC geschickt. Dieser kann die Seitenfehler auflösen, indem er die entsprechenden Seiten an den Thread als Antwort der IPC zurücksendet.

Die nichterlaubten Zugriffe im IO-Adressraum werden auch durch *l4* in Pagefaults umgewandelt[5], so dass sie entsprechend den Speicherseitenfehlern behandelt werden können.

4.2 Pager

Der Servicethread im Linux-Server[6] dient als Pager für die Linuxanwendungen. Für den *L4/Linux*-Kern existiert ein sogenannter "Root-Pager"[7], der das Paging für alle *L4/Linux*-Kernthreads übernimmt.

Bei der Initialisierung des *L4/Linux*-Kernes bekommt dieser seinen gesamten Speicher vom *RMGR*, der ihn wiederum von *Sigma*0, dem intialen Pager des Systems, hat.

4.3 PageRequest Linux-Kern

Die Anforderung von Seiten für den Kern findet durch die Routinen *request{_virtual,_dev}-_page_from_sigma0* statt[8]. Trotz dieses Namens wird der aktuelle Pager des Threads verwendet[9]. Für den Root-Pager ist dies der *RMGR*.

4.4 Linux Systemcalls

Für das Ändern der Zugriffsrechte auf IO-Ports von Linux-Prozessen aus existieren die Systemcalls *sys_ioperm()* und *sys_iopl()*[10]. Zum Erhöhen der IO-Privilegien ist die Capability `CAP_SYS_RAWIO` nötig, die normalerweise nur Prozesse mit UID 0, also mit Root-Rechten, besitzen.

4.4.1 sys_ioperm

Der Linux-Systemcall *sys_ioperm()* erlaubt das Setzen der Zugriffsrechte für die ersten 1024 Ports. Dabei kann man mit einem Aufruf der Zugriff auf einen beliebigen Port-Bereich (portgranular) erlaubt bzw. verboten werden. Die Speicherung der Zugriffsdaten erfolgt bei Linux/i386 in einer Bitmap (`thread_struct.io_bitmap`), die eine Größe von 1024 Bit hat.

[5] l4/kernel/fiasco/src/kern/thread-io.cpp
[6] linux22/arch/l4/x86/kernel/dispatch.c
[7] linux22/arch/l4/x86/lib/l4_pager.c
[8] linux22/arch/l4/x86/lib/l4_memory.c
[9] *checkpager()*
[10] linux22/arch/i386/kernel/ioport.c

4.4.2 sys_iopl

Soll der Zugriff auf höhere Port-Bereiche erlaubt werden, so ist das Ändern der *IOPL* durch *sys_iopl()* nötig. Da ein hoher *IOPL* neben dem Zugriff auf alle Ports auch das Recht zum Sperren der Interrupts einschließt, ist dies ein unsicheres Verfahren.

Dennoch verwenden z.B. einige X-Server dies, um auf die benötigten Ports zuzugreifen.

4.5 Der Rechteentzug

Die Systemcalls können die Rechte wieder entziehen. Dazu sind die *IO-FlexPages* mit *l4_fpage_unmap()* zu entfernen.

Dies setzt jedoch eine funktionierende[11] Mappingdatenbank voraus und ist durch die (potentiell hohe) Anzahl der beteiligten Tasks aufwendig.

Um dennoch die Rechte zu entziehen, existieren zwei Möglichkeiten:

Man kann erstens den Signalthread verwenden, um das Austragen durchzuführen. Dazu schickt man eine bestimmte IPC, die zur Ausführung der Methode führt. Dieses Verfahren setzt ein kooperierenden Thread voraus und ist deswegen sicherheitstechnisch ungünstig.

Zweitens ist es möglich, direkt nach dem Systemcall in der Emulib die Seiten zu entfernen, da jeder Systemcall durch die Emulib gekapselt wird.

Die Sicherheit wird nicht weiter beeinträchtigt. Eine kooperierende Emulib muß schon für die Systemcalls vorausgesetzt werden. Die Verwendung einer modifizierten `glibc` für das direkte Ausführen des Syscalls ohne den Trampolinmechanismus der Emulib ist jedoch hier nicht möglich.

Aus oben genannten Gründen und wegen des geringsten Implementierungsaufwandes ist die zweite Möglichkeit implementiert worden.

---

[11] *IO-FlexPages* sind dort nicht eintragbar.

5 Implementierung

5.1 Root-Pager

Weil der *IOPL* in *Fiasco* lazy erhöht wird, verursacht jeder *L4/Linux-Kernel-Thread* ein Pagefault beim ersten `cli/sti`. Der Pagefault kann ignoriert werden, da der *L4/Linux-Task* zur Zeit mit hoher *IOPL* läuft

Das Eintragen aller IO-Ports in den Adressraum des Kernel-Task findet bereits beim ersten *printk()* während der Initialisierung statt[12], da dies zur Synchronisation ein Spinlock verwendet, welches mit `cli` arbeitet. Ein Request an RMGR ist daher hier momentan nicht nötig.

5.2 Die IODB

Für die Speicherung der erlaubten IO-Rechte in *L4/Linux* wurde *thread_struct* um einen Zeiger *io_db* erweitert[13].

Der Zugriff auf die Rechte erfolgt durch zwei Routinen namens *iodb_read_portrange()* und *iodb_write_portrange*. Dies erlaubt eine einfache Auswechslung der dahinter liegenden Datenstruktur.

Gespeichert werden nicht die einzelnen Ports, sondern nur der Anfang und das Ende der erlaubten Port-Bereiche. Diese Speicherung erlaubt bei wenigen Port-Bereichen eine effizientere Nutzung als die übliche Port-Bitmap.

Die Bereiche sind der Größe nach sortiert und werden minimal gehalten, d.h. Überlappungen oder angrenzende Bereiche werden beim Eintragen verschmolzen.

Außerdem existiert ein Kopf in dieser Struktur, der neben der Anzahl der Einträge und der Größe auch die *IOPL* enthält. Da in Linux die bitweise Freigabe der Ports vom *IOPL* unabhängig ist, ist hier, anders als in *L4*, eine getrennte Speicherung des *IOPL* nötig.

Der Kopf dieser Datenstruktur belegt, genau wie ein Element, 4 Byte. Der Speicher dafür wird in Potenzen von 2 angefordert. Mindestens 16 Byte (Kopf+3 Elemente) werden für Threads, die Einträge in der IODB haben, benötigt.

[12] linux22/init/main.c

[13] linux22/include/asm/l4arch/processor.h

5.3 Servicethread

Der Servicethread in dispatch.c[14] wurde in der Routine *handle_page_fault()* um die Möglichkeit der Behandlung eines IO-Page-Faults erweitert.

Hat ein User-Prozess den *IOPL* von 3 in der IODB durch *sys_iopl()* eingetragen oder den Port durch *sys_ioperm()* freigegeben, so wird ihm die benötige *IO-FlexPage* zugesandt. Andernfalls löst ein *seg_fault_above_task_size()* ein SIGSEGV aus.

5.4 Emulib

Das Austragen der *IO-FlexPages* findet in der Emulib[15] statt. Dazu wurde die Struktur `shared_data` um einen Eintrag `flush_page` für eine FlexPage erweitert. Ist dieser Eintrag ungleich `null` wird ein *unmap()* aufgerufen.

5.5 Die Systemcalls

Die Systemcalls *sys_iopl()* und *sys_ioperm()* wurden analog der i386 Variante implementiert. Nach Abfragen der Rechte (capability CAP_SYS_RAWIO) und dem Testen auf einige Fehler werden mit *iodb_write_portrange()* die IO-Zugriffsrechte gespeichert.

Werden die Rechte eingeschränkt, wird in shared_data.flush_page eine Seite über alle IO-Ports eingetragen und später durch die Emulib wie beschrieben entfernt.

5.6 Fiasco

Bisher ist in *Fiasco* das Entfernen von *IO-FlexPages* nicht implementiert [16]. Dies liegt hauptsächlich an einer Einschränkung der Mappingdatenbank, die zur Zeit mit *IO-FlexPages* nicht umgehen kann.

Unter der Annahme, daß Usertasks keine *IO-FlexPages*weitergeben, ist die Kenntnis der Mappinghierarchie unwichtig. Daher reicht hier eine einfache Implementierung des Entfernens nur für den aktuellen Task aus.

Das Problem des Entziehens des *IOPL* in allen Threads einer Task ist bisher noch nicht gelöst. Dazu ist ein Iterieren über alle Threads in *Fiasco* nötig, um die EFLAGS zu ändern, die für jeden Thread, und nicht für jeden Task gespeichert werden.

---

[14] linux22/arch/l4/x86/kernel/dispatch.c
[15] linux22/arch/l4/x86/emulib/int_entry.S
[16] l4/kernel/fiasco/src/kern/map_util.cpp

6 Ausblick

Werden in *L4/Linux* keine Interrupts mehr gesperrt, kann man auch im Linux-Kern auf einen hohen *IOPL* verzichten. Dazu sollte[17]:

- die zur Speicher-Synchronisation eingesetzten `cli/sti` durch geeignete Synchronisationsprimitive (Semaphorethread) ersetzt werden
- die zur Hardware-Synchronisation eingesetzten `cli/sti` in Omega0 verlagert werden
- die Verwendung von `popf` im Linux-Kern zum Verändern des *IF* ausgeschlossen sein

Wird die Verwaltung der *IO-FlexPages* durch die Mappingdatenbank übernommen, so könnte auch ein direkter *l4_fpage_unmap()*-Aufruf in den Syscalls erfolgen, der dann keine Kooperation mehr voraussetzen würde.

[17] Der entsprechende Code existiert, ist jedoch noch nicht im CVS enthalten.

7 Anhang

7.1 Die veränderten Dateien

Im Rahmen der Implementierung wurden folgende Dateien verändert:

Datei	Grund
linux22/arch/l4/x86/lib/l4_pager.c	IO-Pagefaults in Kernelthreads ignorieren
linux22/include/asm/l4arch/processor.h	thread_struct erweitern
linux22/arch/l4/x86/kernel/Makefile	file ioports.c hinzugefügt
linux22/arch/l4/x86/kernel/unimpl.c	*sys_iopl()* und *sys_ioperm()* entfernen
linux22/arch/l4/x86/kernel/dispatch.c	IODB abfragen und reply setzen
linux22/arch/l4/x86/kernel/ioports.c	IODB und Systemcalls implementiert
linux22/include/l4linux/x86/ioports.h	IODB header file
linux22/arch/l4/x86/emulib/int_entry.S	UnMap-Syscall wenn flush-page vorhanden
l4/kernel/fiasco/src/kern/map_util.cpp	UnMapping der *IO-FlexPages* für einzelnen Task

7.2 Bugs

Während der Implementierung sind einige, mehr oder weniger bekannte, Bugs aufgetreten.

Bezeichnung	Grund	Behebung
VMWareBug 6338	Vmware versteht erweiterte PIC-Programmierung nicht	FIASCO mit Option -vmware starten
TimeOut Bug	Vmware liefert nur alle 10ms Timer Interrupts	FIASCO mit SCHED_PIT compilieren

7.3 Tests

Um die Implementation zu testen wurde ein einfaches C-Programm geschrieben, was folgende Tests ausführt:

Routine	Beschreibung
test_clisti	`cli` bei iopl=0
	lazy iopl switching
	`sti` bei iopl=3
	reset iopl
	`sti` bei iopl=0
test_ioport()	`out` ohne Rechte
	Freigabe für einen Port und Zugriff um diesen Port
	Freigabe von 4 Ports und 32-Bit Portzugriff
	Freigabe von 3 Ports und 32-Bit Portzugriff
	IOPL und Portzugriff
	Unabhängigkeit von *iopl* und *ioperm*
test_fork()	Freigabe von Port und iopl
	fork()
	Rücksetzen des IOPL in Parent
	`sti` in child bei iopl=3
	`sti` in child bei iopl=0
	Portzugriff auf vor *fork()* freigegebenen Port