日曜研究室

技術的な観点から日常を綴ります

[xv6 #27] Chapter 2 – Traps, interrupts, and drivers – Code: System calls

テキストの36〜37ページ

本文

システムコールの場合、trap関数はsyscall関数を呼ぶ。
syscall関数は、トラップフレームからシステムコール番号を読み込む。
トラップフレームは保存された%eaxを含み、システムコールテーブルの特定のエントリを指す。
最初のシステムコールの場合、%eaxはSYS_execという値を含んでいる。（#define SYS_exec 7）
そしてsyscallは、システムコールテーブルのSYS_exec番目のエントリを呼び出す。
そのエントリは、sys_exec関数の呼び出しに対応している。

syscall.cのsyscall関数

```
01 void
02 syscall(void)
03 {
04   int num;
05
06   num = proc->tf->eax;
07   if(num >= 0 && num < SYS_open && syscalls[num]) {
08     proc->tf->eax = syscalls[num]();
09   } else if (num >= SYS_open && num < NELEM(syscalls) &&
   syscalls[num]) {
10     proc->tf->eax = syscalls[num]();
11   } else {
12     cprintf("%d %s: unknown sys call %d\n",
13             proc->pid, proc->name, num);
14     proc->tf->eax = -1;
15   }
16 }
```

syscall関数は、%eaxが指すシステムコール関数の戻り値を記録する。
トラップからユーザスペースに戻ったとき、それはcp->tfからマシンのレジスタにロードされるだろう。
このように、execから戻ったとき、システムコールハンドラから返った値を返すだろう。（syscall関

数中の最初のproc–>tf–>eax = syscalls[num]();の部分）
システムコールは、慣習的にエラーを通知するために負の値を返し、成功したときは正の値を返す。
もしシステムコール番号が異常なら、syscall関数はエラーを印字し-1を返す。

後の章で、個別のシステムコールの実装について説明する。
この章は、システムコールのメカニズムに関係している。
メカニズムが後一つ残っている。
それはシステムコールの引数を見つける仕組みについてである。
argintとargptrとargstrというヘルパ関数は、n番目のシステムコールの引数を取り出す。
それぞれ順番に、整数、ポインタ、文字列に対応する。
argintは、n番目の引数を特定するために、ユーザ空間の%espレジスタを使う。
%espは、システムコールのスタブのための戻り先アドレスを指している。
引数は、まさにその（戻り先アドレス）のすぐ上位にあり、それは%esp+4で導き出せる。
だからn番目の引数は、%esp+4+4*nで導き出せる。

syscall.cのargint, argptr, argstr関連の部分

```
01 // User code makes a system call with INT T_SYSCALL.
02 // System call number in %eax.
03 // Arguments on the stack, from the user call to the C
04 // library system call function. The saved user %esp points
05 // to a saved program counter, and then the first argument.
06
07 // Fetch the int at addr from process p.
08 int
09 fetchint(struct proc *p, uint addr, int *ip)
10 {
11   if(addr >= p->sz || addr+4 > p->sz)
12     return -1;
13   *ip = *(int*)(addr);
14   return 0;
15 }
16
17 // Fetch the nul-terminated string at addr from process p.
18 // Doesn't actually copy the string - just sets *pp to point at it.
19 // Returns length of string, not including nul.
20 int
21 fetchstr(struct proc *p, uint addr, char **pp)
22 {
23   char *s, *ep;
24
25   if(addr >= p->sz)
26     return -1;
27   *pp = (char*)addr;
28   ep = (char*)p->sz;
29   for(s = *pp; s < ep; s++)
30     if(*s == 0)
31       return s - *pp;
32   return -1;
33 }
34
35 // Fetch the nth 32-bit system call argument.
36 int
37 argint(int n, int *ip)
```

```
38 {
39   return fetchint(proc, proc->tf->esp + 4 + 4*n, ip);
40 }
41
42 // Fetch the nth word-sized system call argument as a pointer
43 // to a block of memory of size n bytes.  Check that the pointer
44 // lies within the process address space.
45 int
46 argptr(int n, char **pp, int size)
47 {
48   int i;
49
50   if(argint(n, &i) < 0)
51     return -1;
52   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
53     return -1;
54   *pp = (char*)i;
55   return 0;
56 }
57
58 // Fetch the nth word-sized system call argument as a string
   pointer.
59 // Check that the pointer is valid and the string is nul-terminated.
60 // (There is no shared writable memory, so the string can't change
61 // between this check and being used by the kernel.)
62 int
63 argstr(int n, char **pp)
64 {
65   int addr;
66   if(argint(n, &addr) < 0)
67     return -1;
68   return fetchstr(proc, addr, pp);
69 }
```

argint関数は、ユーザメモリにあるアドレスから値を読み込み、*ipにそれを書き込むためにfetchint関数を呼ぶ。
fetchintは、そのアドレスを簡単にポインタにキャストすることができる。
なぜなら、ユーザとカーネルは同じページテーブルを共有しているからである。
しかし、カーネルは、ユーザによるそのポインタがアドレス空間のユーザの部分にあるポインタであることを確認しなければならない。
カーネルは、プロセスがそれ自身のローカルでプライベートなメモリの外にアクセス出来ないことを確実にするために、ページテーブルハードウェアをセットアップした。
もし、ユーザプログラムがp->szより上のアドレスを読み書きしようとしたら、プロセッサはセグメンテーショントラップを引き起こし、そしてそのトラップはプロセスを殺す。
我々がいままで見てきたように。
今、カーネルは実行中であり、ユーザが渡したかもしれないどのアドレスも逆参照可能である、とはいえ、そのアドレスがp->sz未満であることをカーネルは確認しなければならない。

argptr関数は、argintの用途と似ている。
argptr関数は、n番目のシステムコールの引数を解釈する。
argptr関数は、整数として引数を取ってくるためにargint関数を使い、そしてそのユーザポインタとしての整数がアドレス空間のユーザの部分を指すかどうかをチェックする。
argptrを呼ぶと2回のチェックが走ることに注意。

最初に、ユーザのスタックポインタは引数を取り出すときにチェックされる。
そして引数、ユーザポインタそれ自身はチェックされる。

argstr関数は、システムコール引数トリオの最後のメンバーである。
それは、ポインタとしてn番目の引数を解釈する。
それは、ポインタがヌル終端文字列を指すことと、その文字列全体がアドレス空間のユーザの部分の終わりより書いに配置されていることを確認する。

システムコールの実装（例えばsysproc.cやsysfile.c）は、典型的なラッパ関数である。
それらは、argint, argptr, argstrを使って引数をデコードしたあと、実際の実装を呼び出す。
第1章で、sys_execはそれらの関数を使って引数を解釈した。

sysproc.cで実装されてる関数の最初の3つ

```
01 int
02 sys_fork(void)
03 {
04   return fork();
05 }
06
07 int
08 sys_exit(void)
09 {
10   exit();
11   return 0;  // not reached
12 }
13
14 int
15 sys_wait(void)
16 {
17   return wait();
18 }
```

感想

たいして難しいところじゃないと思うんですが、今回は凄く本文が読みづらかったです。
見慣れない文法や間違ってるっぽい単語や回りくどい書き方がいくつかありました。
なので、いつもそうではあるのですが、今回は特に訳があやしいです。
書いた人が違うのかな。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/3 土曜日 [http://peta.okechan.net/blog/archives/1394] |