システムソフトウェア

2021年度

第2回 (10/7)

月曜7-8限・木曜7-8限(Zoom)

講義担当：渡部卓雄 (Takuo Watanabe)

http://titech-os.github.io

e-mail: takuo∅c.titech.ac.jp

本日のメニュー

- システムコール(2)
  - システムコールの実現

CPUの動作モード

- CPUが実行できる命令を規定する
  - ユーザモード
    - 通常のアプリケーションプログラム等を実行するモード
      - 特権命令と呼ばれる一部の命令を除いて実行できる
  - カーネルモード
    - OSカーネルを実行するモード
      - 通常特権命令を含め全命令を実行できる
      - 特権(privileged)モード, スーパーバイザ(supervisor)モードとも呼ばれる
- 例: CPUの動作モード（xv6では下線部を使用）
  - RISC-V: 0 (User) < 1 (Supervisor) < 3 (Machine)
  - x86: 0 > 1 > 2 > 3

システムコールの実行モード

- アプリケーションプロセス
  - ユーザモードで実行される
  - 他プロセスやカーネルが使用しているメモリ領域や，入出力デバイスへのアクセス，およびCPUモードの変更はできない
- システムコール
  - カーネルが使用しているメモリ領域や入出力デバイスへのアクセスが必要である
  - よって，アプリケーションプロセスから呼び出される際に，CPUモードの変更が必要になる

システムコールのインターフェース

user.h

```
...
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
...
```

- xv6や他のUnix系OSでは，システムコールはCの関数として呼び出すことができる
- ヘッダファイルuser.hで左のように定義されているが，対応するCの関数の定義はない

システムコールの実装(RISC-V)(1)

usys.pl

```
...
sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}

entry("fork");
entry("exit");
entry("wait");
entry("pipe");
entry("read");
entry("write");
entry("close");
...
entry("uptime");
```

- システムコールのエントリは左のようなPerlスクリプトで定義されており，実行すると以下のようなアセンブリ言語の記述になる

```
...
.global write
write:
 li a7, SYS_write
 ecall
 ret
...
```

システムコールの実装(RISC-V)(2)

```
.global write
write:
 li a7, SYS_write    # SYS_wirte == 16 (syscall.h)
 ecall
 ret
```

- Cプログラムから関数writeを呼び出すと，上記のアセンブリコードで定義されたラベルwriteに制御が移る.
- a7レジスタに16（writeのシステムコール番号）がセットされ，ecallが実行される.
  - ecallはMachine-Modeへのコール命令

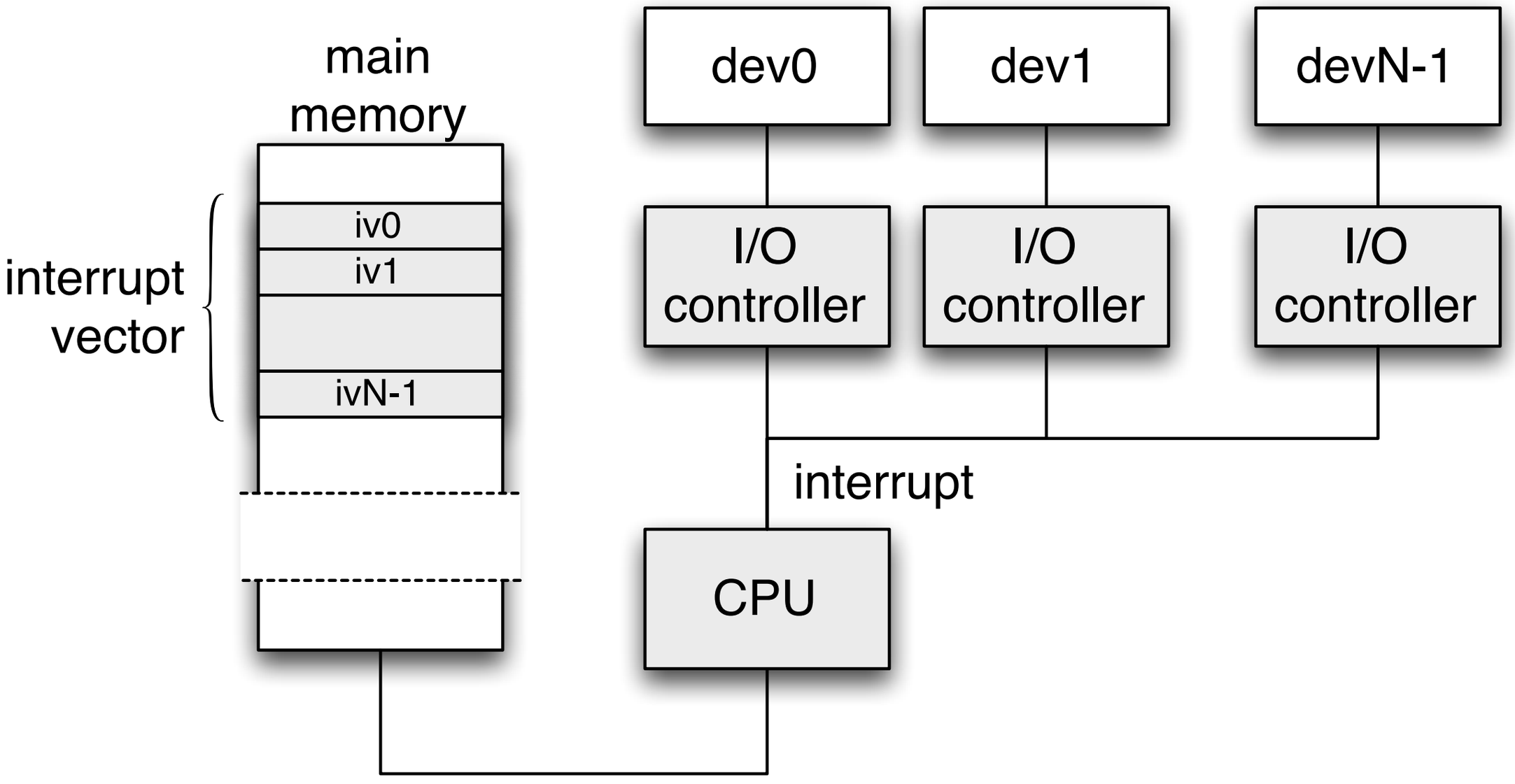
割り込み(interrupt)

- CPUが実行中の処理を中断して，指定された動作を行うこと
- 種類
  - 外部(external)割り込み
    - CPU外部のハードウェアが発生させる
    - 例：周辺機器，タイマー
  - 内部(internal)割り込み
    - CPUの制御回路が発生させる
    - 例：エラー，ページフォルト，保護違反，割り込み命令，ブレークポイント命令

割り込みハンドラ

- 割り込みに応じて起動されるプログラムを割り込みハンドラ(interrupt handler)と呼ぶ.
  - ISR: Interrupt Service Routine とも言う.
- 割り込みが発生すると，CPUは現在実行中のプログラムを一時中断し，所定の割り込みハンドラの実行を開始する.
  - ハンドラはカーネルモードで実行される
- 割り込みハンドラの実行が終わったら，中断していたプログラムの実行を再開する.

割り込み機構


main
memory
interrupt
vector
iv0
iv1
ivN-1
dev0
dev1
devN-1
I/O
controller
I/O
controller
I/O
controller
interrupt
CPU

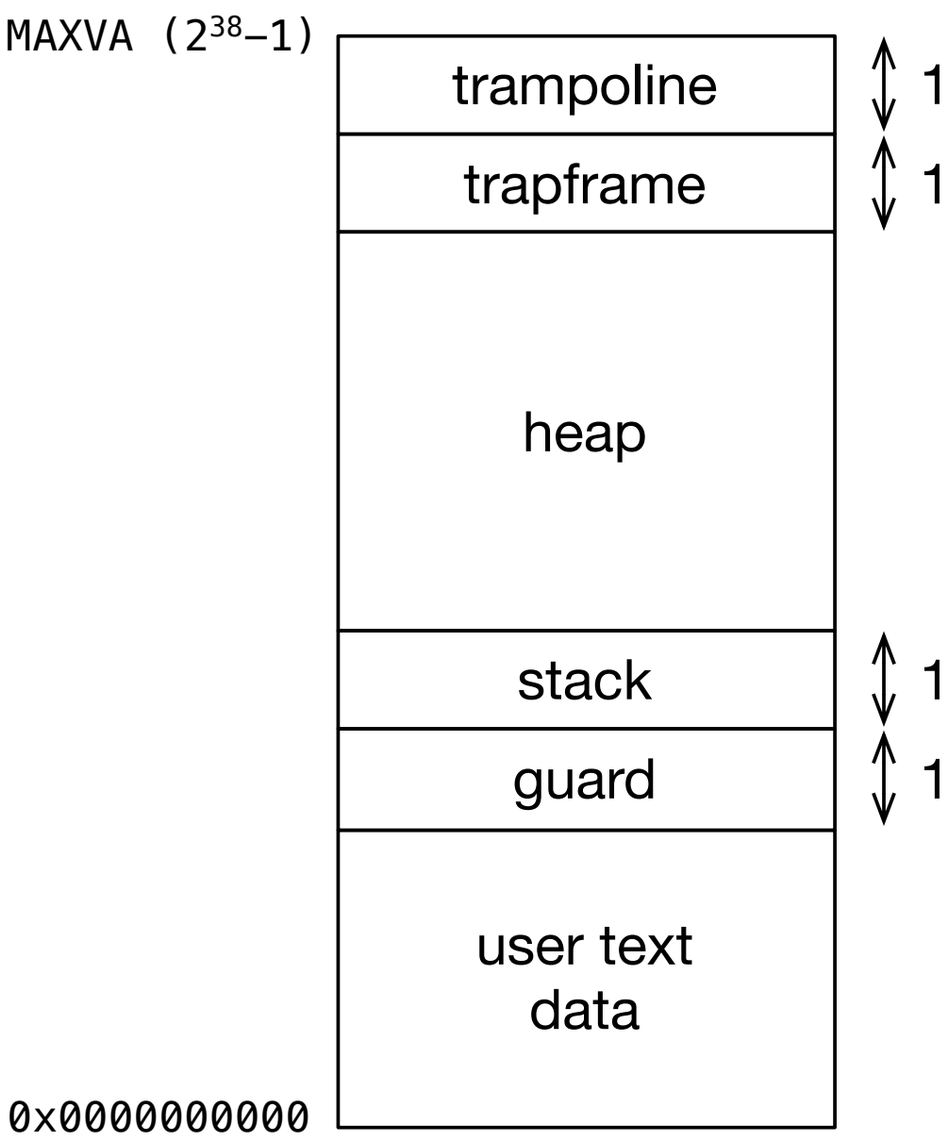
割り込みの動作

- 割り込み要因: 0〜N-1
  - 割り込みを発生するデバイス等に対応する.
  - 対応するハンドラのアドレスは，決められたメモリ領域(割り込みベクタ)に格納されている.
- 割り込み発生に伴う一般的なCPUの動作
  - 現在のPCとPSWをスタックに退避する.
  - 実行モードをカーネルモードにする.
  - 割り込み要因に対応したハンドラを呼び出す.

割り込みベクタとハンドラ (RISC-V)

- ユーザ空間→カーネル(trampoline.S, trap.c)
  - trapframeにレジスタを退避
  - usertrap() にジャンプ
  - 割り込み要因に基づいて処理
    - システムコールの場合 syscall を呼ぶ

xv6のメモリ空間 (RISC-V)



- 64ビット中下位38ビットを使用
- trampoline
  - カーネル空間との切り替えに使うコード
  - カーネル空間と共有
- trapframe
  - システムコールの引数等

syscall(void);　(RISC-V)

- システムコール呼び出しを行う(syscall.c)
  - proc->tf->a7 に，ecall を実行した時点におけるa7レジスタの値が入っている.
    - システムコール番号 : writeの場合は16
  - 配列syscallsのシステムコール番号の場所に，実際にシステムコールを実行する関数へのポインタが入っている.

```
int num;
struct proc *p = myproc();
num = proc->tf->a7;
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
   curproc->tf->a0 = syscalls[num]();
} else { ... }
```

sys_write(void);

- writeシステムコールの実体(sysfile.c)
  - 引数を取り出してfilewrite(file.c)を呼び出す.
- システムコールの引数はトラップフレームに保持されている（呼び出し時点での）.
  - syscall.cのargint, argstr, fetchint, fetchstr等

簡単なシステムコールを作ってみる

- int getppid()
  - 返値：現在のプロセスの親プロセスのプロセスid

- キーワード
  - プロセス，親(子)プロセス，プロセスid
  - fork, wait, exit, getpid

作業手順

- 作成するシステムコールのテスト用プログラム（ユーザプログラム）を作る
- システムコールを実装する
  - xv6のソースコードに手を入れる必要がある
- ビルドしてテスト実行

テスト用プログラム：ppidtest.c

```
#include "kernel/types.h"
#include "user/user.h"

int main(void) {
    int parent = getpid();
    int child = fork();
    if (child == 0) {
        printf("CHILD: parent=%d child=%d getpid()=%d getppid()=%d\n",
               parent, child, getpid(), getppid());
    }
    else {
        wait(0);
        printf("PARENT: parent=%d child=%d getpid()=%d getppid()=%d\n",
               parent, child, getpid(), getppid());
    }
    exit(0);
}
```

getppidの実装方針

- getpidの実装を参考にする

sysproc.c

```
int
sys_getpid(void)
{
  return myproc()->pid;
}
```

- myproc()
  - 現在実行中のプロセスを得るための関数
    - 返値：proc構造体(proc.h)へのポインタ

proc構造体

```
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Bottom of kernel stack for this process
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // Page table
  struct trapframe *tf;        // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

親プロセスへの参照

- proc構造体を見ると，parentというフィールドがあり，その型はproc構造体へのポインタとなっている.
- これが親プロセスのproc構造体へのポインタであろうことは容易に予想がつく.
- つまり，parentフィールドを介して親プロセスの構造体にアクセスし，そこに格納されているpidフィールドの値を取ればよい.

getppidの実装

- 前頁の方針に従い以下のように定義する

```
int
sys_getppid(void)
{
  return myproc()->parent->pid;
}
```

- あとは，この関数をシステムコールとして呼び出せるようにすればよいだけ

システムコールの関連ファイル

- 以下のファイルがシステムコールに関連している
  - user/usys.pl
    - システムコールのエントリポイントを生成するPerlスクリプト
  - user/user.h
    - Cの関数としてのシステムコールの型宣言
  - kernel/syscall.h
    - システムコール番号の定義
  - kernel/syscall.c
    - システムコール関数テーブルの定義
  - kernel/sysfile.c, kernel/sysproc.c
    - ファイル及びプロセス関連のシステムコール関数定義
- 新たなシステムコールを追加するためには，これらを適宜変更すればよい

システムコールの追加作業(1)

- user/usys.pl：以下の行を追加する

```
entry("getppid")
```

- user/user.h：以下の行を追加する

```
int getppid(void);
```

- kernel/syscall.h：以下の行を追加する

```
#define SYS_getppid 22
```

システムコールの追加作業(2)

- syscall.c：以下の行を追加する

```
extern int sys_getppid(void);
```

- syscall.c：配列syscallの初期化要素に以下の行を追加する（カンマを忘れないよう注意）

```
[SYS_getppid] sys_getppid
```

システムコールの追加作業(3)

- sysproc.c：sys_getppid()の定義を追加する

```
int
sys_getppid(void)
{
  return myproc()->parent->pid;
}
```

- Makefile：UPROGSにテスト用プログラムの名前を追加する

```
UPROGS =\
        _cat\
        ...
        _ppidtest
```

実行例

```
$ make qemu
...
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ ppidtest
CHILD: parent=3 child=0 getpid()=4 getppid()=3
PARENT: parent=3 child=4 getpid()=3 getppid()=2
$ ppidtest
CHILD: parent=5 child=0 getpid()=6 getppid()=5
PARENT: parent=5 child=6 getpid()=5 getppid()=2
$
```

まとめ

- システムコール(2)
  - CPUの動作モード
  - 割込み・割込みハンドラ
  - xv6のメモリ空間
  - システムコールの実装