XV6

0.1

Generated by Doxygen 1.9.6

Chapter 1

ENVIRONMENT PREPARETION

1.1 LINUX

1. `sudo apt-get install git wget qemu aqemu`
2. `sudo apt-get install libc6-dev:i386 gcc`
3. `sudo apt-get install gdb`
4. `git clone` https://github.com/subhrendu1987/xv6-public/
5. `cd xv6-public && make qemu`
6. If you have 64 bit OS there is a chance Makefile will not be able to find qemu. Check using the command `make qemu-nox` and see the output. In that case you should edit the Makefile at line 54 and add the following code: `QEMU = qemu-system-x86_64`

1. `make qemu-gdb`

1. Open new terminal `cd xv6-public && gdb ./kernel`

1.2 BUILDING AND RUNNING XV6

- To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make".
- On non-x86 or non-ELF machines (like OS X, even on x86), you may need to install a cross-compiler gcc suite capable of producing x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
- Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC simulator and run "make qemu".

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Class Documentation

4.1 backcmd

Public Attributes

- struct cmd * cmd
- int type

4.1.1 Detailed Description

Definition at line 47 of file sh.c.

4.1.2 Member Data Documentation

4.1.2.1 cmd

```
struct cmd* backcmd::cmd
```

Definition at line 49 of file sh.c.

Referenced by backcmd(), nulterminate(), and runcmd().

4.1.2.2 type

```
int backcmd::type
```

Definition at line 48 of file sh.c.

The documentation for this struct was generated from the following file:

- sh.c

4.2 buf

`#include <buf.h>`

Public Attributes

- uint blockno
- uchar data [BSIZE]
- uint dev
- int flags
- struct sleeplock lock
- struct buf * next
- struct buf * prev
- struct buf * qnext
- uint refcnt

4.2.1 Detailed Description

Definition at line 1 of file buf.h.

4.2.2 Member Data Documentation

4.2.2.1 blockno

`uint buf::blockno`

Definition at line 4 of file buf.h.

Referenced by bget(), bread(), iderw(), idestart(), and log_write().

4.2.2.2 data

`uchar buf::data[BSIZE]`

Definition at line 10 of file buf.h.

Referenced by balloc(), bfree(), bmap(), bzero(), ialloc(), ideintr(), iderw(), idestart(), ilock(), install_trans(), itrunc(), iupdate(), read_head(), readi(), readsb(), write_head(), write_log(), and writei().

4.2.2.3 dev

```
uint buf::dev
```

Definition at line 3 of file buf.h.

Referenced by balloc(), bfree(), bget(), bread(), bzero(), iderw(), idestart(), and readsb().

4.2.2.4 flags

```
int buf::flags
```

Definition at line 2 of file buf.h.

Referenced by bget(), bread(), bwrite(), ideintr(), iderw(), idestart(), and log_write().

4.2.2.5 lock

```
struct sleeplock buf::lock
```

Definition at line 5 of file buf.h.

Referenced by bget(), binit(), brelse(), bwrite(), and iderw().

4.2.2.6 next

```
struct buf* buf::next
```

Definition at line 8 of file buf.h.

Referenced by bget(), binit(), and brelse().

4.2.2.7 prev

```
struct buf* buf::prev
```

Definition at line 7 of file buf.h.

Referenced by bget(), binit(), and brelse().

4.2.2.8 qnext

```
struct buf* buf::qnext
```

Definition at line 9 of file buf.h.

Referenced by ideintr(), and iderw().

4.2.2.9 refcnt

```
uint buf::refcnt
```

Definition at line 6 of file buf.h.

Referenced by bget(), and brelse().

The documentation for this struct was generated from the following file:

- buf.h

4.3 cmd

Public Attributes

- int type

4.3.1 Detailed Description

Definition at line 16 of file sh.c.

4.3.2 Member Data Documentation

4.3.2.1 type

```
int cmd::type
```

Definition at line 17 of file sh.c.

Referenced by backcmd(), execcmd(), listcmd(), nulterminate(), pipecmd(), redircmd(), and runcmd().

The documentation for this struct was generated from the following file:

- sh.c

4.4 context

```
#include <proc.h>
```

Public Attributes

- uint ebp
- uint ebx
- uint edi
- uint eip
- uint esi

4.4.1 Detailed Description

Definition at line 27 of file proc.h.

4.4.2 Member Data Documentation

4.4.2.1 ebp

```
uint context::ebp
```

Definition at line 31 of file proc.h.

Referenced by procdump().

4.4.2.2 ebx

```
uint context::ebx
```

Definition at line 30 of file proc.h.

4.4.2.3 edi

```
uint context::edi
```

Definition at line 28 of file proc.h.

4.4.2.4 eip

```
uint context::eip
```

Definition at line 32 of file proc.h.

Referenced by allocproc().

4.4.2.5 esi

```
uint context::esi
```

Definition at line 29 of file proc.h.

The documentation for this struct was generated from the following file:

- proc.h

4.5 cpu

```
#include <proc.h>
```

Public Attributes

- uchar apicid
- struct segdesc gdt [NSEGS]
- int intena
- int ncli
- struct proc * proc
- struct context * scheduler
- volatile uint started
- struct taskstate ts

4.5.1 Detailed Description

Definition at line 2 of file proc.h.

4.5.2 Member Data Documentation

4.5.2.1 apicid

```
uchar cpu::apicid
```

Definition at line 3 of file proc.h.

Referenced by mpinit(), mycpu(), and startothers().

4.5.2.2 gdt

```
struct segdesc cpu::gdt[NSEGS]
```

Definition at line 6 of file proc.h.

Referenced by seginit(), and switchuvm().

4.5.2.3 intena

```
int cpu::intena
```

Definition at line 9 of file proc.h.

Referenced by pushcli(), and sched().

4.5.2.4 ncli

```
int cpu::ncli
```

Definition at line 8 of file proc.h.

Referenced by pushcli().

4.5.2.5 proc

```
struct proc* cpu::proc
```

Definition at line 10 of file proc.h.

Referenced by myproc(), and scheduler().

4.5.2.6 scheduler

```
struct context* cpu::scheduler
```

Definition at line 4 of file proc.h.

Referenced by scheduler().

4.5.2.7 started

```
volatile uint cpu::started
```

Definition at line 7 of file proc.h.

Referenced by startothers().

4.5.2.8 ts

```
struct taskstate cpu::ts
```

Definition at line 5 of file proc.h.

Referenced by switchuvm().

The documentation for this struct was generated from the following file:

- proc.h

4.6 devsw

```
#include <file.h>
```

Public Attributes

- int(* read)(struct inode *, char *, int)
- int(* write)(struct inode *, char *, int)

4.6.1 Detailed Description

Definition at line 30 of file file.h.

4.6.2 Member Data Documentation

4.6.2.1 read

```
int(* devsw::read) (struct inode *, char *, int)
```

Definition at line 31 of file file.h.

Referenced by consoleinit(), and readi().

4.6.2.2 write

```
int(* devsw::write) (struct inode *, char *, int)
```

Definition at line 32 of file file.h.

Referenced by consoleinit(), and writei().

The documentation for this struct was generated from the following file:

- file.h

4.7 dinode

```
#include <fs.h>
```

Public Attributes

- uint addrs [NDIRECT+1]
- short major
- short minor
- short nlink
- uint size
- short type

4.7.1 Detailed Description

Definition at line 29 of file fs.h.

4.7.2 Member Data Documentation

4.7.2.1 addrs

```
uint dinode::addrs[NDIRECT+1]
```

Definition at line 35 of file fs.h.

Referenced by iappend(), ilock(), and iupdate().

4.7.2.2 major

```
short dinode::major
```

Definition at line 31 of file fs.h.

Referenced by ilock(), and iupdate().

4.7.2.3 minor

```
short dinode::minor
```

Definition at line 32 of file fs.h.

Referenced by ilock(), and iupdate().

4.7.2.4 nlink

```
short dinode::nlink
```

Definition at line 33 of file fs.h.

Referenced by ialloc(), ilock(), and iupdate().

4.7.2.5 size

```
uint dinode::size
```

Definition at line 34 of file fs.h.

Referenced by ialloc(), iappend(), ilock(), iupdate(), and main().

4.7.2.6 type

```
short dinode::type
```

Definition at line 30 of file fs.h.

Referenced by ialloc(), ilock(), and iupdate().

The documentation for this struct was generated from the following file:

- fs.h

4.8 dirent

```
#include <fs.h>
```

Public Attributes

- ushort inum
- char name [DIRSIZ]

4.8.1 Detailed Description

Definition at line 53 of file fs.h.

4.8.2 Member Data Documentation

4.8.2.1 inum

```
ushort dirent::inum
```

Definition at line 54 of file fs.h.

Referenced by dirlink(), dirlookup(), ls(), and main().

4.8.2.2 name

```
char dirent::name[DIRSIZ]
```

Definition at line 55 of file fs.h.

Referenced by dirlink(), ls(), and main().

The documentation for this struct was generated from the following file:

- fs.h

4.9 elfhdr

`#include <elf.h>`

Public Attributes

- ushort ehsize
- uchar elf [12]
- uint entry
- uint flags
- ushort machine
- uint magic
- ushort phentsize
- ushort phnum
- uint phoff
- ushort shentsize
- ushort shnum
- uint shoff
- ushort shstrndx
- ushort type
- uint version

4.9.1 Detailed Description

Definition at line 6 of file elf.h.

4.9.2 Member Data Documentation

4.9.2.1 ehsize

`ushort elfhdr::ehsize`

Definition at line 16 of file elf.h.

4.9.2.2 elf

`uchar elfhdr::elf[12]`

Definition at line 8 of file elf.h.

Referenced by bootmain(), and exec().

4.9.2.3 entry

```
uint elfhdr::entry
```

Definition at line 12 of file elf.h.

Referenced by bootmain(), and exec().

4.9.2.4 flags

```
uint elfhdr::flags
```

Definition at line 15 of file elf.h.

4.9.2.5 machine

```
ushort elfhdr::machine
```

Definition at line 10 of file elf.h.

4.9.2.6 magic

```
uint elfhdr::magic
```

Definition at line 7 of file elf.h.

Referenced by exec().

4.9.2.7 phentsize

```
ushort elfhdr::phentsize
```

Definition at line 17 of file elf.h.

4.9.2.8 phnum

```
ushort elfhdr::phnum
```

Definition at line 18 of file elf.h.

Referenced by bootmain(), and exec().

4.9.2.9 phoff

```
uint elfhdr::phoff
```

Definition at line 13 of file elf.h.

Referenced by bootmain(), and exec().

4.9.2.10 shentsize

```
ushort elfhdr::shentsize
```

Definition at line 19 of file elf.h.

4.9.2.11 shnum

```
ushort elfhdr::shnum
```

Definition at line 20 of file elf.h.

4.9.2.12 shoff

```
uint elfhdr::shoff
```

Definition at line 14 of file elf.h.

4.9.2.13 shstrndx

```
ushort elfhdr::shstrndx
```

Definition at line 21 of file elf.h.

4.9.2.14 type

```
ushort elfhdr::type
```

Definition at line 9 of file elf.h.

4.9.2.15 version

```
uint elfhdr::version
```

Definition at line 11 of file elf.h.

The documentation for this struct was generated from the following file:

- elf.h

4.10 execcmd

Public Attributes

- char * argv [MAXARGS]
- char * eargv [MAXARGS]
- int type

4.10.1 Detailed Description

Definition at line 20 of file sh.c.

4.10.2 Member Data Documentation

4.10.2.1 argv

```
char* execcmd::argv[MAXARGS]
```

Definition at line 22 of file sh.c.

Referenced by nulterminate(), and runcmd().

4.10.2.2 eargv

```
char* execcmd::eargv[MAXARGS]
```

Definition at line 23 of file sh.c.

Referenced by nulterminate().

4.10.2.3 type

```
int execcmd::type
```

Definition at line 21 of file sh.c.

The documentation for this struct was generated from the following file:

- sh.c

4.11 file

```
#include <file.h>
```

Public Types

- enum { FD_NONE , FD_PIPE , FD_INODE }

Public Attributes

- struct inode * ip
- uint off
- struct pipe * pipe
- char readable
- int ref
- enum file:: { ... } type
- char writable

4.11.1 Detailed Description

Definition at line 1 of file file.h.

4.11.2 Member Enumeration Documentation

4.11.2.1 anonymous enum

```
anonymous enum
```

Enumerator

FD_NONE	
FD_PIPE	
FD_INODE	

Definition at line 2 of file file.h.
```
00002 { FD_NONE, FD_PIPE, FD_INODE } type;
```

4.11.3 Member Data Documentation

4.11.3.1 ip

```
struct inode* file::ip
```

Definition at line 7 of file file.h.

Referenced by fileclose(), fileread(), filestat(), filewrite(), and sys_open().

4.11.3.2 off

```
uint file::off
```

Definition at line 8 of file file.h.

Referenced by fileread(), filewrite(), and sys_open().

4.11.3.3 pipe

```
struct pipe* file::pipe
```

Definition at line 6 of file file.h.

Referenced by fileclose(), fileread(), and filewrite().

4.11.3.4 readable

```
char file::readable
```

Definition at line 4 of file file.h.

Referenced by fileread(), and sys_open().

4.11.3.5 ref

```
int file::ref
```

Definition at line 3 of file file.h.

Referenced by filealloc(), fileclose(), and filedup().

4.11.3.6

```
enum { ... } file::type
```

Referenced by fileclose(), fileread(), filestat(), filewrite(), and sys_open().

4.11.3.7 writable

```
char file::writable
```

Definition at line 5 of file file.h.

Referenced by fileclose(), filewrite(), and sys_open().

The documentation for this struct was generated from the following file:

- file.h

4.12 gatedesc

```
#include <mmu.h>
```

Public Attributes

- uint args: 5
- uint cs: 16
- uint dpl: 2
- uint off_15_0: 16
- uint off_31_16: 16
- uint p: 1
- uint rsv1: 3
- uint s: 1
- uint type: 4

4.12.1 Detailed Description

Definition at line 148 of file mmu.h.

4.12.2 Member Data Documentation

4.12.2.1 args

```
uint gatedesc::args
```

Definition at line 151 of file mmu.h.

4.12.2.2 cs

```
uint gatedesc::cs
```

Definition at line 150 of file mmu.h.

4.12.2.3 dpl

```
uint gatedesc::dpl
```

Definition at line 155 of file mmu.h.

4.12.2.4 off_15_0

```
uint gatedesc::off_15_0
```

Definition at line 149 of file mmu.h.

4.12.2.5 off_31_16

```
uint gatedesc::off_31_16
```

Definition at line 157 of file mmu.h.

4.12.2.6 p

```
uint gatedesc::p
```

Definition at line 156 of file mmu.h.

Referenced by lidt().

4.12.2.7 rsv1

```
uint gatedesc::rsv1
```

Definition at line 152 of file mmu.h.

4.12.2.8 s

```
uint gatedesc::s
```

Definition at line 154 of file mmu.h.

4.12.2.9 type

```
uint gatedesc::type
```

Definition at line 153 of file mmu.h.

The documentation for this struct was generated from the following file:

- mmu.h

4.13 header

Public Attributes

- struct {
  union header * ptr
  uint size
  } s

- Align x

4.13.1 Detailed Description

Definition at line 11 of file umalloc.c.

4.13.2 Member Data Documentation

4.13.2.1 ptr

```
union header* header::ptr
```

Definition at line 13 of file umalloc.c.

Referenced by free(), and malloc().

4.13.2.2

```
struct { ... } header::s
```

Referenced by free(), malloc(), and morecore().

4.13.2.3 size

```
uint header::size
```

Definition at line 14 of file umalloc.c.

Referenced by free(), malloc(), and morecore().

4.13.2.4 x

```
Align header::x
```

Definition at line 16 of file umalloc.c.

The documentation for this union was generated from the following file:

- umalloc.c

4.14 inode

`#include <file.h>`

Public Attributes

- uint addrs [NDIRECT+1]
- uint dev
- uint inum
- struct sleeplock lock
- short major
- short minor
- short nlink
- int ref
- uint size
- short type
- int valid

4.14.1 Detailed Description

Definition at line 13 of file file.h.

4.14.2 Member Data Documentation

4.14.2.1 addrs

`uint inode::addrs[NDIRECT+1]`

Definition at line 25 of file file.h.

Referenced by bmap(), ilock(), itrunc(), and iupdate().

4.14.2.2 dev

`uint inode::dev`

Definition at line 14 of file file.h.

Referenced by bmap(), create(), dirlookup(), iget(), iinit(), ilock(), itrunc(), iupdate(), readi(), stati(), sys_link(), and writei().

4.14.2.3 inum

```
uint inode::inum
```

Definition at line 15 of file file.h.

Referenced by create(), dirlink(), dirlookup(), ialloc(), iget(), ilock(), iupdate(), stati(), and sys_link().

4.14.2.4 lock

```
struct sleeplock inode::lock
```

Definition at line 17 of file file.h.

Referenced by ilock(), iput(), and iunlock().

4.14.2.5 major

```
short inode::major
```

Definition at line 21 of file file.h.

Referenced by create(), ilock(), iupdate(), readi(), sys_mknod(), and writei().

4.14.2.6 minor

```
short inode::minor
```

Definition at line 22 of file file.h.

Referenced by create(), ilock(), iupdate(), and sys_mknod().

4.14.2.7 nlink

```
short inode::nlink
```

Definition at line 23 of file file.h.

Referenced by create(), ilock(), iput(), iupdate(), stati(), sys_link(), and sys_unlink().

4.14.2.8 ref

```
int inode::ref
```

Definition at line 16 of file file.h.

Referenced by idup(), iget(), ilock(), iput(), and iunlock().

4.14.2.9 size

```
uint inode::size
```

Definition at line 24 of file file.h.

Referenced by dirlink(), dirlookup(), ilock(), isdirempty(), itrunc(), iupdate(), readi(), stati(), and writei().

4.14.2.10 type

```
short inode::type
```

Definition at line 20 of file file.h.

Referenced by create(), dirlookup(), ilock(), iput(), iupdate(), namex(), readi(), stati(), sys_chdir(), sys_link(), sys_open(), sys_unlink(), and writei().

4.14.2.11 valid

```
int inode::valid
```

Definition at line 18 of file file.h.

Referenced by iget(), ilock(), and iput().

The documentation for this struct was generated from the following file:

- file.h

4.15 ioapic

Public Attributes

- uint data
- uint pad [3]
- uint reg

4.15.1 Detailed Description

Definition at line 28 of file ioapic.c.

4.15.2 Member Data Documentation

4.15.2.1 data

```
uint ioapic::data
```

Definition at line 31 of file ioapic.c.

Referenced by ioapicread(), and ioapicwrite().

4.15.2.2 pad

```
uint ioapic::pad[3]
```

Definition at line 30 of file ioapic.c.

4.15.2.3 reg

```
uint ioapic::reg
```

Definition at line 29 of file ioapic.c.

Referenced by ioapicread(), and ioapicwrite().

The documentation for this struct was generated from the following file:

- ioapic.c

4.16 kmap

Public Attributes

- int perm
- uint phys_end
- uint phys_start
- void * virt

4.16.1 Detailed Description

Definition at line 105 of file vm.c.

4.16.2 Member Data Documentation

4.16.2.1 perm

```
int kmap::perm
```

Definition at line 109 of file vm.c.

Referenced by setupkvm().

4.16.2.2 phys_end

```
uint kmap::phys_end
```

Definition at line 108 of file vm.c.

Referenced by setupkvm().

4.16.2.3 phys_start

```
uint kmap::phys_start
```

Definition at line 107 of file vm.c.

Referenced by setupkvm().

4.16.2.4 virt

```
void* kmap::virt
```

Definition at line 106 of file vm.c.

Referenced by setupkvm().

The documentation for this struct was generated from the following file:

- vm.c

4.17 listcmd

Public Attributes

- struct cmd * left
- struct cmd * right
- int type

4.17.1 Detailed Description

Definition at line 41 of file sh.c.

4.17.2 Member Data Documentation

4.17.2.1 left

```
struct cmd* listcmd::left
```

Definition at line 43 of file sh.c.

Referenced by listcmd(), nulterminate(), and runcmd().

4.17.2.2 right

```
struct cmd* listcmd::right
```

Definition at line 44 of file sh.c.

Referenced by listcmd(), nulterminate(), and runcmd().

4.17.2.3 type

```
int listcmd::type
```

Definition at line 42 of file sh.c.

The documentation for this struct was generated from the following file:

- sh.c

4.18 log

Public Attributes

- int committing
- int dev
- struct logheader lh
- struct spinlock lock
- int outstanding
- int size
- int start

4.18.1 Detailed Description

Definition at line 39 of file log.c.

4.18.2 Member Data Documentation

4.18.2.1 committing

```
int log::committing
```

Definition at line 44 of file log.c.

Referenced by begin_op(), and end_op().

4.18.2.2 dev

```
int log::dev
```

Definition at line 45 of file log.c.

Referenced by initlog(), install_trans(), read_head(), write_head(), and write_log().

4.18.2.3 lh

```
struct logheader log::lh
```

Definition at line 46 of file log.c.

Referenced by begin_op(), commit(), install_trans(), log_write(), read_head(), recover_from_log(), write_head(), and write_log().

4.18.2.4 lock

```
struct spinlock log::lock
```

Definition at line 40 of file log.c.

Referenced by begin_op(), end_op(), initlog(), and log_write().

4.18.2.5 outstanding

```
int log::outstanding
```

Definition at line 43 of file log.c.

Referenced by begin_op(), end_op(), and log_write().

4.18.2.6 size

```
int log::size
```

Definition at line 42 of file log.c.

Referenced by initlog(), and log_write().

4.18.2.7 start

```
int log::start
```

Definition at line 41 of file log.c.

Referenced by initlog(), install_trans(), read_head(), write_head(), and write_log().

The documentation for this struct was generated from the following file:

- log.c

4.19 logheader

Public Attributes

- int block [LOGSIZE]
- int n

4.19.1 Detailed Description

Definition at line 34 of file log.c.

4.19.2 Member Data Documentation

4.19.2.1 block

```
int logheader::block[LOGSIZE]
```

Definition at line 36 of file log.c.

Referenced by install_trans(), log_write(), read_head(), write_head(), and write_log().

4.19.2.2 n

```
int logheader::n
```

Definition at line 35 of file log.c.

Referenced by begin_op(), commit(), install_trans(), log_write(), read_head(), recover_from_log(), write_head(), and write_log().

The documentation for this struct was generated from the following file:

- log.c

4.20 mp

```
#include <mp.h>
```

Public Attributes

- uchar checksum
- uchar imcrp
- uchar length
- void * physaddr
- uchar reserved [3]
- uchar signature [4]
- uchar specrev
- uchar type

4.20.1 Detailed Description

Definition at line 3 of file mp.h.

4.20.2 Member Data Documentation

4.20.2.1 checksum

```
uchar mp::checksum
```

Definition at line 8 of file mp.h.

4.20.2.2 imcrp

```
uchar mp::imcrp
```

Definition at line 10 of file mp.h.

Referenced by mpinit().

4.20.2.3 length

```
uchar mp::length
```

Definition at line 6 of file mp.h.

4.20.2.4 physaddr

```
void* mp::physaddr
```

Definition at line 5 of file mp.h.

Referenced by mpconfig().

4.20.2.5 reserved

```
uchar mp::reserved[3]
```

Definition at line 11 of file mp.h.

4.20.2.6 signature

```
uchar mp::signature[4]
```

Definition at line 4 of file mp.h.

4.20.2.7 specrev

```
uchar mp::specrev
```

Definition at line 7 of file mp.h.

4.20.2.8 type

```
uchar mp::type
```

Definition at line 9 of file mp.h.

The documentation for this struct was generated from the following file:

- mp.h

4.21 mpconf

```
#include <mp.h>
```

Public Attributes

- uchar checksum
- ushort entry
- uint * lapicaddr
- ushort length
- ushort oemlength
- uint * oemtable
- uchar product [20]
- uchar reserved
- uchar signature [4]
- uchar version
- uchar xchecksum
- ushort xlength

4.21.1 Detailed Description

Definition at line 14 of file mp.h.

4.21.2 Member Data Documentation

4.21.2.1 checksum

```
uchar mpconf::checksum
```

Definition at line 18 of file mp.h.

4.21.2.2 entry

```
ushort mpconf::entry
```

Definition at line 22 of file mp.h.

4.21.2.3 lapicaddr

```
uint* mpconf::lapicaddr
```

Definition at line 23 of file mp.h.

Referenced by mpinit().

4.21.2.4 length

```
ushort mpconf::length
```

Definition at line 16 of file mp.h.

Referenced by mpconfig(), and mpinit().

4.21.2.5 oemlength

```
ushort mpconf::oemlength
```

Definition at line 21 of file mp.h.

4.21.2.6 oemtable

```
uint* mpconf::oemtable
```

Definition at line 20 of file mp.h.

4.21.2.7 product

```
uchar mpconf::product[20]
```

Definition at line 19 of file mp.h.

4.21.2.8 reserved

```
uchar mpconf::reserved
```

Definition at line 26 of file mp.h.

4.21.2.9 signature

```
uchar mpconf::signature[4]
```

Definition at line 15 of file mp.h.

4.21.2.10 version

```
uchar mpconf::version
```

Definition at line 17 of file mp.h.

Referenced by mpconfig().

4.21.2.11 xchecksum

```
uchar mpconf::xchecksum
```

Definition at line 25 of file mp.h.

4.21.2.12 xlength

```
ushort mpconf::xlength
```

Definition at line 24 of file mp.h.

The documentation for this struct was generated from the following file:

- mp.h

4.22 mpioapic

```
#include <mp.h>
```

Public Attributes

- uint * addr
- uchar apicno
- uchar flags
- uchar type
- uchar version

4.22.1 Detailed Description

Definition at line 40 of file mp.h.

4.22.2 Member Data Documentation

4.22.2.1 addr

```
uint* mpioapic::addr
```

Definition at line 45 of file mp.h.

4.22.2.2 apicno

```
uchar mpioapic::apicno
```

Definition at line 42 of file mp.h.

4.22.2.3 flags

```
uchar mpioapic::flags
```

Definition at line 44 of file mp.h.

4.22.2.4 type

```
uchar mpioapic::type
```

Definition at line 41 of file mp.h.

4.22.2.5 version

```
uchar mpioapic::version
```

Definition at line 43 of file mp.h.

The documentation for this struct was generated from the following file:

- mp.h

4.23 mpproc

```
#include <mp.h>
```

Public Attributes

- uchar apicid
- uint feature
- uchar flags
- uchar reserved [8]
- uchar signature [4]
- uchar type
- uchar version

4.23.1 Detailed Description

Definition at line 29 of file mp.h.

4.23.2 Member Data Documentation

4.23.2.1 apicid

```
uchar mpproc::apicid
```

Definition at line 31 of file mp.h.

4.23.2.2 feature

```
uint mpproc::feature
```

Definition at line 36 of file mp.h.

4.23.2.3 flags

```
uchar mpproc::flags
```

Definition at line 33 of file mp.h.

4.23.2.4 reserved

```
uchar mpproc::reserved[8]
```

Definition at line 37 of file mp.h.

4.23.2.5 signature

```
uchar mpproc::signature[4]
```

Definition at line 35 of file mp.h.

4.23.2.6 type

```
uchar mpproc::type
```

Definition at line 30 of file mp.h.

4.23.2.7 version

```
uchar mpproc::version
```

Definition at line 32 of file mp.h.

The documentation for this struct was generated from the following file:

- mp.h

4.24 pipe

Public Attributes

- char data [PIPESIZE]
- struct spinlock lock
- uint nread
- uint nwrite
- int readopen
- int writeopen

4.24.1 Detailed Description

Definition at line 13 of file pipe.c.

4.24.2 Member Data Documentation

4.24.2.1 data

```
char pipe::data[PIPESIZE]
```

Definition at line 15 of file pipe.c.

Referenced by piperead(), and pipewrite().

4.24.2.2 lock

```
struct spinlock pipe::lock
```

Definition at line 14 of file pipe.c.

Referenced by pipealloc(), pipeclose(), piperead(), and pipewrite().

4.24.2.3 nread

```
uint pipe::nread
```

Definition at line 16 of file pipe.c.

Referenced by pipealloc(), pipeclose(), piperead(), and pipewrite().

4.24.2.4 nwrite

```
uint pipe::nwrite
```

Definition at line 17 of file pipe.c.

Referenced by pipealloc(), pipeclose(), piperead(), and pipewrite().

4.24.2.5 readopen

```
int pipe::readopen
```

Definition at line 18 of file pipe.c.

Referenced by pipealloc(), pipeclose(), and pipewrite().

4.24.2.6 writeopen

```
int pipe::writeopen
```

Definition at line 19 of file pipe.c.

Referenced by pipealloc(), pipeclose(), and piperead().

The documentation for this struct was generated from the following file:

- pipe.c

4.25 pipecmd

Public Attributes

- struct cmd * left
- struct cmd * right
- int type

4.25.1 Detailed Description

Definition at line 35 of file sh.c.

4.25.2 Member Data Documentation

4.25.2.1 left

```
struct cmd* pipecmd::left
```

Definition at line 37 of file sh.c.

Referenced by nulterminate(), pipecmd(), and runcmd().

4.25.2.2 right

```
struct cmd* pipecmd::right
```

Definition at line 38 of file sh.c.

Referenced by nulterminate(), pipecmd(), and runcmd().

4.25.2.3 type

```
int pipecmd::type
```

Definition at line 36 of file sh.c.

The documentation for this struct was generated from the following file:

- sh.c

4.26 proc

`#include <proc.h>`

Public Attributes

- void * chan
- struct context * context
- struct inode * cwd
- int killed
- char * kstack
- char name [16]
- struct file * ofile [NOFILE]
- struct proc * parent
- pde_t * pgdir
- int pid
- int priority
- enum procstate state
- uint sz
- struct trapframe * tf

4.26.1 Detailed Description

Definition at line 38 of file proc.h.

4.26.2 Member Data Documentation

4.26.2.1 chan

`void* proc::chan`

Definition at line 47 of file proc.h.

Referenced by sleep(), wakeup(), and wakeup1().

4.26.2.2 context

`struct context* proc::context`

Definition at line 46 of file proc.h.

Referenced by allocproc(), procdump(), sched(), and scheduler().

4.26.2.3 cwd

```
struct inode* proc::cwd
```

Definition at line 50 of file proc.h.

Referenced by exit(), fork(), sys_chdir(), and userinit().

4.26.2.4 killed

```
int proc::killed
```

Definition at line 48 of file proc.h.

Referenced by kill(), sys_sleep(), trap(), and wait().

4.26.2.5 kstack

```
char* proc::kstack
```

Definition at line 41 of file proc.h.

Referenced by allocproc(), fork(), switchuvm(), and wait().

4.26.2.6 name

```
char proc::name[16]
```

Definition at line 51 of file proc.h.

Referenced by cps(), exec(), fork(), procdump(), sys_getprocs(), syscall(), userinit(), and wait().

4.26.2.7 ofile

```
struct file* proc::ofile[NOFILE]
```

Definition at line 49 of file proc.h.

Referenced by exit(), fdalloc(), fork(), sys_close(), and sys_pipe().

4.26.2.8 parent

```
struct proc* proc::parent
```

Definition at line 44 of file proc.h.

Referenced by exit(), fork(), sys_getprocs(), and wait().

4.26.2.9 pgdir

```
pde_t* proc::pgdir
```

Definition at line 40 of file proc.h.

Referenced by exec(), fork(), growproc(), switchuvm(), userinit(), and wait().

4.26.2.10 pid

```
int proc::pid
```

Definition at line 43 of file proc.h.

Referenced by acquiresleep(), allocproc(), chpr(), cps(), fork(), holdingsleep(), kill(), procdump(), sys_chpr(), sys_getpid(), sys_getprocs(), sys_kill(), syscall(), and wait().

4.26.2.11 priority

```
int proc::priority
```

Definition at line 52 of file proc.h.

Referenced by chpr(), and cps().

4.26.2.12 state

```
enum procstate proc::state
```

Definition at line 42 of file proc.h.

Referenced by allocproc(), cps(), exit(), fork(), kill(), procdump(), sched(), scheduler(), sleep(), sys_getprocs(), userinit(), wait(), wakeup1(), and yield().

4.26.2.13 sz

```
uint proc::sz
```

Definition at line 39 of file proc.h.

Referenced by argptr(), exec(), fetchint(), fetchstr(), fork(), growproc(), sys_sbrk(), and userinit().

4.26.2.14 tf

```
struct trapframe* proc::tf
```

Definition at line 45 of file proc.h.

Referenced by allocproc(), argint(), exec(), fork(), syscall(), trap(), and userinit().

The documentation for this struct was generated from the following file:

- proc.h

4.27 proghdr

```
#include <elf.h>
```

Public Attributes

- uint align
- uint filesz
- uint flags
- uint memsz
- uint off
- uint paddr
- uint type
- uint vaddr

4.27.1 Detailed Description

Definition at line 25 of file elf.h.

4.27.2 Member Data Documentation

4.27.2.1 align

```
uint proghdr::align
```

Definition at line 33 of file elf.h.

4.27.2.2 filesz

```
uint proghdr::filesz
```

Definition at line 30 of file elf.h.

Referenced by bootmain().

4.27.2.3 flags

```
uint proghdr::flags
```

Definition at line 32 of file elf.h.

4.27.2.4 memsz

```
uint proghdr::memsz
```

Definition at line 31 of file elf.h.

Referenced by bootmain().

4.27.2.5 off

```
uint proghdr::off
```

Definition at line 27 of file elf.h.

Referenced by bootmain().

4.27.2.6 paddr

```
uint proghdr::paddr
```

Definition at line 29 of file elf.h.

Referenced by bootmain().

4.27.2.7 type

```
uint proghdr::type
```

Definition at line 26 of file elf.h.

4.27.2.8 vaddr

```
uint proghdr::vaddr
```

Definition at line 28 of file elf.h.

The documentation for this struct was generated from the following file:

- elf.h

4.28 redircmd

Public Attributes

- struct cmd * cmd
- char * efile
- int fd
- char * file
- int mode
- int type

4.28.1 Detailed Description

Definition at line 26 of file sh.c.

4.28.2 Member Data Documentation

4.28.2.1 cmd

```
struct cmd* redircmd::cmd
```

Definition at line 28 of file sh.c.

Referenced by nulterminate(), redircmd(), and runcmd().

4.28.2.2 efile

```
char* redircmd::efile
```

Definition at line 30 of file sh.c.

Referenced by nulterminate(), and redircmd().

4.28.2.3 fd

```
int redircmd::fd
```

Definition at line 32 of file sh.c.

Referenced by redircmd(), and runcmd().

4.28.2.4 file

```
char* redircmd::file
```

Definition at line 29 of file sh.c.

Referenced by redircmd(), and runcmd().

4.28.2.5 mode

```
int redircmd::mode
```

Definition at line 31 of file sh.c.

Referenced by redircmd(), and runcmd().

4.28.2.6 type

```
int redircmd::type
```

Definition at line 27 of file sh.c.

The documentation for this struct was generated from the following file:

- sh.c

4.29 rtcdate

```
#include <date.h>
```

Public Attributes

- uint day
- uint hour
- uint minute
- uint month
- uint second
- uint year

4.29.1 Detailed Description

Definition at line 1 of file date.h.

4.29.2 Member Data Documentation

4.29.2.1 day

```
uint rtcdate::day
```

Definition at line 5 of file date.h.

Referenced by cmostime().

4.29.2.2 hour

```
uint rtcdate::hour
```

Definition at line 4 of file date.h.

Referenced by cmostime().

4.29.2.3 minute

```
uint rtcdate::minute
```

Definition at line 3 of file date.h.

Referenced by cmostime().

4.29.2.4 month

```
uint rtcdate::month
```

Definition at line 6 of file date.h.

Referenced by cmostime().

4.29.2.5 second

```
uint rtcdate::second
```

Definition at line 2 of file date.h.

Referenced by cmostime().

4.29.2.6 year

```
uint rtcdate::year
```

Definition at line 7 of file date.h.

Referenced by cmostime().

The documentation for this struct was generated from the following file:

- date.h

4.30 run

Public Attributes

- struct run * next

4.30.1 Detailed Description

Definition at line 16 of file kalloc.c.

4.30.2 Member Data Documentation

4.30.2.1 next

```
struct run* run::next
```

Definition at line 17 of file kalloc.c.

The documentation for this struct was generated from the following file:

- kalloc.c

4.31 segdesc

```
#include <mmu.h>
```

Public Attributes

- uint avl: 1
- uint base_15_0: 16
- uint base_23_16: 8
- uint base_31_24: 8
- uint db: 1
- uint dpl: 2
- uint g: 1
- uint lim_15_0: 16
- uint lim_19_16: 4
- uint p: 1
- uint rsv1: 1
- uint s: 1
- uint type: 4

4.31.1 Detailed Description

Definition at line 26 of file mmu.h.

4.31.2 Member Data Documentation

4.31.2.1 avl

```
uint segdesc::avl
```

Definition at line 35 of file mmu.h.

4.31.2.2 base_15_0

```
uint segdesc::base_15_0
```

Definition at line 28 of file mmu.h.

4.31.2.3 base_23_16

```
uint segdesc::base_23_16
```

Definition at line 29 of file mmu.h.

4.31.2.4 base_31_24

```
uint segdesc::base_31_24
```

Definition at line 39 of file mmu.h.

4.31.2.5 db

```
uint segdesc::db
```

Definition at line 37 of file mmu.h.

4.31.2.6 dpl

```
uint segdesc::dpl
```

Definition at line 32 of file mmu.h.

4.31.2.7 g

```
uint segdesc::g
```

Definition at line 38 of file mmu.h.

4.31.2.8 lim_15_0

```
uint segdesc::lim_15_0
```

Definition at line 27 of file mmu.h.

4.31.2.9 lim_19_16

```
uint segdesc::lim_19_16
```

Definition at line 34 of file mmu.h.

4.31.2.10 p

```
uint segdesc::p
```

Definition at line 33 of file mmu.h.

Referenced by lgdt().

4.31.2.11 rsv1

```
uint segdesc::rsv1
```

Definition at line 36 of file mmu.h.

4.31.2.12 s

```
uint segdesc::s
```

Definition at line 31 of file mmu.h.

Referenced by switchuvm().

4.31.2.13 type

```
uint segdesc::type
```

Definition at line 30 of file mmu.h.

The documentation for this struct was generated from the following file:

- mmu.h

4.32 sleeplock

```
#include <sleeplock.h>
```

Public Attributes

- struct spinlock lk
- uint locked
- char * name
- int pid

4.32.1 Detailed Description

Definition at line 2 of file sleeplock.h.

4.32.2 Member Data Documentation

4.32.2.1 lk

```
struct spinlock sleeplock::lk
```

Definition at line 4 of file sleeplock.h.

Referenced by acquiresleep(), holdingsleep(), initsleeplock(), and releasesleep().

4.32.2.2 locked

```
uint sleeplock::locked
```

Definition at line 3 of file sleeplock.h.

Referenced by acquiresleep(), holdingsleep(), initsleeplock(), and releasesleep().

4.32.2.3 name

```
char* sleeplock::name
```

Definition at line 7 of file sleeplock.h.

Referenced by initsleeplock().

4.32.2.4 pid

```
int sleeplock::pid
```

Definition at line 8 of file sleeplock.h.

Referenced by acquiresleep(), holdingsleep(), initsleeplock(), and releasesleep().

The documentation for this struct was generated from the following file:

- sleeplock.h

4.33 spinlock

```
#include <spinlock.h>
```

Public Attributes

- struct cpu * cpu
- uint locked
- char * name
- uint pcs [10]

4.33.1 Detailed Description

Definition at line 2 of file spinlock.h.

4.33.2 Member Data Documentation

4.33.2.1 cpu

```
struct cpu* spinlock::cpu
```

Definition at line 7 of file spinlock.h.

Referenced by acquire(), holding(), initlock(), and release().

4.33.2.2 locked

```
uint spinlock::locked
```

Definition at line 3 of file spinlock.h.

Referenced by acquire(), holding(), initlock(), and release().

4.33.2.3 name

```
char* spinlock::name
```

Definition at line 6 of file spinlock.h.

Referenced by initlock().

4.33.2.4 pcs

```
uint spinlock::pcs[10]
```

Definition at line 8 of file spinlock.h.

Referenced by acquire(), panic(), and release().

The documentation for this struct was generated from the following file:

- spinlock.h

4.34 stat

```
#include <stat.h>
```

Public Attributes

- int dev
- uint ino
- short nlink
- uint size
- short type

4.34.1 Detailed Description

Definition at line 5 of file stat.h.

4.34.2 Member Data Documentation

4.34.2.1 dev

```
int stat::dev
```

Definition at line 7 of file stat.h.

Referenced by stati().

4.34.2.2 ino

```
uint stat::ino
```

Definition at line 8 of file stat.h.

Referenced by ls(), and stati().

4.34.2.3 nlink

```
short stat::nlink
```

Definition at line 9 of file stat.h.

Referenced by stati().

4.34.2.4 size

```
uint stat::size
```

Definition at line 10 of file stat.h.

Referenced by ls(), and stati().

4.34.2.5 type

```
short stat::type
```

Definition at line 6 of file stat.h.

Referenced by ls(), and stati().

The documentation for this struct was generated from the following file:

- stat.h

4.35 superblock

```
#include <fs.h>
```

Public Attributes

- uint bmapstart
- uint inodestart
- uint logstart
- uint nblocks
- uint ninodes
- uint nlog
- uint size

4.35.1 Detailed Description

Definition at line 14 of file fs.h.

4.35.2 Member Data Documentation

4.35.2.1 bmapstart

```
uint superblock::bmapstart
```

Definition at line 21 of file fs.h.

Referenced by balloc(), iinit(), and main().

4.35.2.2 inodestart

```
uint superblock::inodestart
```

Definition at line 20 of file fs.h.

Referenced by iinit(), and main().

4.35.2.3 logstart

```
uint superblock::logstart
```

Definition at line 19 of file fs.h.

Referenced by iinit(), initlog(), and main().

4.35.2.4 nblocks

```
uint superblock::nblocks
```

Definition at line 16 of file fs.h.

Referenced by iinit(), and main().

4.35.2.5 ninodes

```
uint superblock::ninodes
```

Definition at line 17 of file fs.h.

Referenced by ialloc(), iinit(), and main().

4.35.2.6 nlog

```
uint superblock::nlog
```

Definition at line 18 of file fs.h.

Referenced by iinit(), initlog(), and main().

4.35.2.7 size

```
uint superblock::size
```

Definition at line 15 of file fs.h.

Referenced by balloc(), iinit(), and main().

The documentation for this struct was generated from the following file:

- fs.h

4.36 taskstate

```
#include <mmu.h>
```

Public Attributes

- void * cr3
- ushort cs
- ushort ds
- uint eax
- uint * ebp
- uint ebx
- uint ecx
- uint edi
- uint edx
- uint eflags
- uint * eip
- ushort es
- uint esi
- uint * esp
- uint esp0
- uint * esp1
- uint * esp2
- ushort fs
- ushort gs
- ushort iomb
- ushort ldt
- uint link

- ushort padding1
- ushort padding10
- ushort padding2
- ushort padding3
- ushort padding4
- ushort padding5
- ushort padding6
- ushort padding7
- ushort padding8
- ushort padding9
- ushort ss
- ushort ss0
- ushort ss1
- ushort ss2
- ushort t

4.36.1 Detailed Description

Definition at line 107 of file mmu.h.

4.36.2 Member Data Documentation

4.36.2.1 cr3

```
void* taskstate::cr3
```

Definition at line 118 of file mmu.h.

4.36.2.2 cs

```
ushort taskstate::cs
```

Definition at line 131 of file mmu.h.

4.36.2.3 ds

```
ushort taskstate::ds
```

Definition at line 135 of file mmu.h.

4.36.2.4 eax

```
uint taskstate::eax
```

Definition at line 121 of file mmu.h.

4.36.2.5 ebp

```
uint* taskstate::ebp
```

Definition at line 126 of file mmu.h.

4.36.2.6 ebx

```
uint taskstate::ebx
```

Definition at line 124 of file mmu.h.

4.36.2.7 ecx

```
uint taskstate::ecx
```

Definition at line 122 of file mmu.h.

4.36.2.8 edi

```
uint taskstate::edi
```

Definition at line 128 of file mmu.h.

4.36.2.9 edx

```
uint taskstate::edx
```

Definition at line 123 of file mmu.h.

4.36.2.10 eflags

```
uint taskstate::eflags
```

Definition at line 120 of file mmu.h.

4.36.2.11 eip

```
uint* taskstate::eip
```

Definition at line 119 of file mmu.h.

4.36.2.12 es

```
ushort taskstate::es
```

Definition at line 129 of file mmu.h.

4.36.2.13 esi

```
uint taskstate::esi
```

Definition at line 127 of file mmu.h.

4.36.2.14 esp

```
uint* taskstate::esp
```

Definition at line 125 of file mmu.h.

4.36.2.15 esp0

```
uint taskstate::esp0
```

Definition at line 109 of file mmu.h.

Referenced by switchuvm().

4.36.2.16 esp1

```
uint* taskstate::esp1
```

Definition at line 112 of file mmu.h.

4.36.2.17 esp2

```
uint* taskstate::esp2
```

Definition at line 115 of file mmu.h.

4.36.2.18 fs

```
ushort taskstate::fs
```

Definition at line 137 of file mmu.h.

4.36.2.19 gs

```
ushort taskstate::gs
```

Definition at line 139 of file mmu.h.

4.36.2.20 iomb

```
ushort taskstate::iomb
```

Definition at line 144 of file mmu.h.

Referenced by switchuvm().

4.36.2.21 ldt

```
ushort taskstate::ldt
```

Definition at line 141 of file mmu.h.

4.36.2.22 link

```
uint taskstate::link
```

Definition at line 108 of file mmu.h.

4.36.2.23 padding1

```
ushort taskstate::padding1
```

Definition at line 111 of file mmu.h.

4.36.2.24 padding10

```
ushort taskstate::padding10
```

Definition at line 142 of file mmu.h.

4.36.2.25 padding2

```
ushort taskstate::padding2
```

Definition at line 114 of file mmu.h.

4.36.2.26 padding3

```
ushort taskstate::padding3
```

Definition at line 117 of file mmu.h.

4.36.2.27 padding4

```
ushort taskstate::padding4
```

Definition at line 130 of file mmu.h.

4.36.2.28 padding5

```
ushort taskstate::padding5
```

Definition at line 132 of file mmu.h.

4.36.2.29 padding6

```
ushort taskstate::padding6
```

Definition at line 134 of file mmu.h.

4.36.2.30 padding7

```
ushort taskstate::padding7
```

Definition at line 136 of file mmu.h.

4.36.2.31 padding8

```
ushort taskstate::padding8
```

Definition at line 138 of file mmu.h.

4.36.2.32 padding9

```
ushort taskstate::padding9
```

Definition at line 140 of file mmu.h.

4.36.2.33 ss

```
ushort taskstate::ss
```

Definition at line 133 of file mmu.h.

4.36.2.34 ss0

```
ushort taskstate::ss0
```

Definition at line 110 of file mmu.h.

Referenced by switchuvm().

4.36.2.35 ss1

```
ushort taskstate::ss1
```

Definition at line 113 of file mmu.h.

4.36.2.36 ss2

```
ushort taskstate::ss2
```

Definition at line 116 of file mmu.h.

4.36.2.37 t

```
ushort taskstate::t
```

Definition at line 143 of file mmu.h.

The documentation for this struct was generated from the following file:

- mmu.h

4.37 trapframe

```
#include <x86.h>
```

Public Attributes

- ushort cs
- ushort ds
- uint eax
- uint ebp
- uint ebx
- uint ecx
- uint edi
- uint edx
- uint eflags
- uint eip
- uint err
- ushort es
- uint esi
- uint esp
- ushort fs
- ushort gs
- uint oesp
- ushort padding1
- ushort padding2
- ushort padding3
- ushort padding4
- ushort padding5
- ushort padding6
- ushort ss
- uint trapno

4.37.1 Detailed Description

Definition at line 150 of file x86.h.

4.37.2 Member Data Documentation

4.37.2.1 cs

```
ushort trapframe::cs
```

Definition at line 175 of file x86.h.

Referenced by trap(), and userinit().

4.37.2.2 ds

```
ushort trapframe::ds
```

Definition at line 168 of file x86.h.

Referenced by userinit().

4.37.2.3 eax

```
uint trapframe::eax
```

Definition at line 159 of file x86.h.

Referenced by fork(), and syscall().

4.37.2.4 ebp

```
uint trapframe::ebp
```

Definition at line 154 of file x86.h.

4.37.2.5 ebx

```
uint trapframe::ebx
```

Definition at line 156 of file x86.h.

4.37.2.6 ecx

```
uint trapframe::ecx
```

Definition at line 158 of file x86.h.

4.37.2.7 edi

```
uint trapframe::edi
```

Definition at line 152 of file x86.h.

4.37.2.8 edx

```
uint trapframe::edx
```

Definition at line 157 of file x86.h.

4.37.2.9 eflags

```
uint trapframe::eflags
```

Definition at line 177 of file x86.h.

Referenced by userinit().

4.37.2.10 eip

```
uint trapframe::eip
```

Definition at line 174 of file x86.h.

Referenced by exec(), trap(), and userinit().

4.37.2.11 err

```
uint trapframe::err
```

Definition at line 173 of file x86.h.

Referenced by trap().

4.37.2.12 es

```
ushort trapframe::es
```

Definition at line 166 of file x86.h.

Referenced by userinit().

4.37.2.13 esi

```
uint trapframe::esi
```

Definition at line 153 of file x86.h.

4.37.2.14 esp

```
uint trapframe::esp
```

Definition at line 180 of file x86.h.

Referenced by argint(), exec(), and userinit().

4.37.2.15 fs

```
ushort trapframe::fs
```

Definition at line 164 of file x86.h.

4.37.2.16 gs

```
ushort trapframe::gs
```

Definition at line 162 of file x86.h.

4.37.2.17 oesp

```
uint trapframe::oesp
```

Definition at line 155 of file x86.h.

4.37.2.18 padding1

```
ushort trapframe::padding1
```

Definition at line 163 of file x86.h.

4.37.2.19 padding2

```
ushort trapframe::padding2
```

Definition at line 165 of file x86.h.

4.37.2.20 padding3

```
ushort trapframe::padding3
```

Definition at line 167 of file x86.h.

4.37.2.21 padding4

```
ushort trapframe::padding4
```

Definition at line 169 of file x86.h.

4.37.2.22 padding5

```
ushort trapframe::padding5
```

Definition at line 176 of file x86.h.

4.37.2.23 padding6

```
ushort trapframe::padding6
```

Definition at line 182 of file x86.h.

4.37.2.24 ss

```
ushort trapframe::ss
```

Definition at line 181 of file x86.h.

Referenced by userinit().

4.37.2.25 trapno

```
uint trapframe::trapno
```

Definition at line 170 of file x86.h.

Referenced by trap().

The documentation for this struct was generated from the following file:

- x86.h

4.38 uproc

```
#include <uproc.h>
```

Public Attributes

- char name [16]
- int pid
- int ppid

4.38.1 Detailed Description

Definition at line 1 of file uproc.h.

4.38.2 Member Data Documentation

4.38.2.1 name

```
char uproc::name[16]
```

Definition at line 4 of file uproc.h.

4.38.2.2 pid

```
int uproc::pid
```

Definition at line 2 of file uproc.h.

Referenced by main(), and sys_getprocs().

4.38.2.3 ppid

```
int uproc::ppid
```

Definition at line 3 of file uproc.h.

Referenced by main(), and sys_getprocs().

The documentation for this struct was generated from the following file:

- uproc.h

Chapter 5

File Documentation

5.1 asm.h File Reference

Macros

- #define SEG_ASM(type, base, lim)
- #define SEG_NULLASM
- #define STA_R 0x2
- #define STA_W 0x2
- #define STA_X 0x8

5.1.1 Macro Definition Documentation

5.1.1.1 SEG_ASM

```
#define SEG_ASM(
              type,
              base,
              lim )
```

Value:

```
        .word (((lim) » 12) & 0xffff), ((base) & 0xffff);       \
        .byte (((base) » 16) & 0xff), (0x90 | (type)),          \
              (0xC0 | (((lim) » 28) & 0xf)), (((base) » 24) & 0xff)
```

Definition at line 11 of file asm.h.

5.1.1.2 SEG_NULLASM

```
#define SEG_NULLASM
```

Value:

```
        .word 0, 0;                                             \
        .byte 0, 0, 0, 0
```

Definition at line 5 of file asm.h.

5.1.1.3 STA_R

```
#define STA_R 0x2
```

Definition at line 18 of file asm.h.

5.1.1.4 STA_W

```
#define STA_W 0x2
```

Definition at line 17 of file asm.h.

5.1.1.5 STA_X

```
#define STA_X 0x8
```

Definition at line 16 of file asm.h.

5.2 asm.h

Go to the documentation of this file.

```
00001 //
00002 // assembler macros to create x86 segments
00003 //
00004
00005 #define SEG_NULLASM                                             \
00006         .word 0, 0;                                             \
00007         .byte 0, 0, 0, 0
00008
00009 // The 0xC0 means the limit is in 4096-byte units
00010 // and (for executable segments) 32-bit mode.
00011 #define SEG_ASM(type,base,lim)                                  \
00012         .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
00013         .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
00014                 (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
00015
00016 #define STA_X     0x8       // Executable segment
00017 #define STA_W     0x2       // Writeable (non-executable segments)
00018 #define STA_R     0x2       // Readable (executable segments)
```

5.3 bio.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "buf.h"
```

Functions

- static struct buf * bget (uint dev, uint blockno)
- void binit (void)
- struct buf * bread (uint dev, uint blockno)
- void brelse (struct buf *b)
- void bwrite (struct buf *b)

Variables

- struct {
  struct buf buf [NBUF]
  struct buf head
  struct spinlock lock
  } bcache

5.3.1 Function Documentation

5.3.1.1 bget()

```
static struct buf * bget (
            uint dev,
            uint blockno )  [static]
```

Definition at line 62 of file bio.c.

```
00063 {
00064   struct buf *b;
00065
00066   acquire(&bcache.lock);
00067
00068   // Is the block already cached?
00069   for(b = bcache.head.next; b != &bcache.head; b = b->next){
00070     if(b->dev == dev && b->blockno == blockno){
00071       b->refcnt++;
00072       release(&bcache.lock);
00073       acquiresleep(&b->lock);
00074       return b;
00075     }
00076   }
00077
00078   // Not cached; recycle an unused buffer.
00079   // Even if refcnt==0, B_DIRTY indicates a buffer is in use
00080   // because log.c has modified it but not yet committed it.
00081   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
00082     if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
00083       b->dev = dev;
00084       b->blockno = blockno;
00085       b->flags = 0;
00086       b->refcnt = 1;
00087       release(&bcache.lock);
00088       acquiresleep(&b->lock);
00089       return b;
00090     }
00091   }
00092   panic("bget: no buffers");
00093 }
```

Referenced by bread().

5.3.1.2 binit()

```
void binit (
            void  )
```

Definition at line 39 of file bio.c.

```
00040 {
00041   struct buf *b;
00042
00043   initlock(&bcache.lock, "bcache");
00044
00045 //PAGEBREAK!
00046   // Create linked list of buffers
00047   bcache.head.prev = &bcache.head;
00048   bcache.head.next = &bcache.head;
00049   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
00050     b->next = bcache.head.next;
00051     b->prev = &bcache.head;
00052     initsleeplock(&b->lock, "buffer");
00053     bcache.head.next->prev = b;
00054     bcache.head.next = b;
00055   }
00056 }
```

5.3.1.3 bread()

```
struct buf * bread (
            uint dev,
            uint blockno )
```

Definition at line 97 of file bio.c.

```
00098 {
00099   struct buf *b;
00100
00101   b = bget(dev, blockno);
00102   if((b->flags & B_VALID) == 0) {
00103     iderw(b);
00104   }
00105   return b;
00106 }
```

Referenced by balloc(), bfree(), bmap(), bzero(), ialloc(), ilock(), install_trans(), itrunc(), iupdate(), read_head(), readi(), readsb(), write_head(), write_log(), and writei().

5.3.1.4 brelse()

```
void brelse (
            struct buf * b )
```

Definition at line 121 of file bio.c.

```
00122 {
00123   if(!holdingsleep(&b->lock))
00124     panic("brelse");
00125
00126   releasesleep(&b->lock);
00127
00128   acquire(&bcache.lock);
00129   b->refcnt--;
00130   if (b->refcnt == 0) {
00131     // no one is waiting for it.
00132     b->next->prev = b->prev;
00133     b->prev->next = b->next;
00134     b->next = bcache.head.next;
00135     b->prev = &bcache.head;
```

```
00136     bcache.head.next->prev = b;
00137     bcache.head.next = b;
00138   }
00139
00140   release(&bcache.lock);
00141 }
```

Referenced by balloc(), bfree(), bmap(), bzero(), ialloc(), ilock(), install_trans(), itrunc(), iupdate(), read_head(), readi(), readsb(), write_head(), write_log(), and writei().

5.3.1.5 bwrite()

```
void bwrite (
              struct buf * b )
```

Definition at line 110 of file bio.c.

```
00111 {
00112   if(!holdingsleep(&b->lock))
00113     panic("bwrite");
00114   b->flags |= B_DIRTY;
00115   iderw(b);
00116 }
```

Referenced by install_trans(), write_head(), and write_log().

5.3.2 Variable Documentation

5.3.2.1

```
struct { ...  } bcache
```

Referenced by bget(), binit(), and brelse().

5.3.2.2 buf

```
struct buf buf[NBUF]
```

Definition at line 31 of file bio.c.

5.3.2.3 head

```
struct buf head
```

Definition at line 35 of file bio.c.

5.3.2.4 lock

```
struct spinlock lock
```

Definition at line 30 of file bio.c.

Referenced by holding().

5.4 bio.c

Go to the documentation of this file.

```
00001 // Buffer cache.
00002 //
00003 // The buffer cache is a linked list of buf structures holding
00004 // cached copies of disk block contents.  Caching disk blocks
00005 // in memory reduces the number of disk reads and also provides
00006 // a synchronization point for disk blocks used by multiple processes.
00007 //
00008 // Interface:
00009 // * To get a buffer for a particular disk block, call bread.
00010 // * After changing buffer data, call bwrite to write it to disk.
00011 // * When done with the buffer, call brelse.
00012 // * Do not use the buffer after calling brelse.
00013 // * Only one process at a time can use a buffer,
00014 //     so do not keep them longer than necessary.
00015 //
00016 // The implementation uses two state flags internally:
00017 // * B_VALID: the buffer data has been read from the disk.
00018 // * B_DIRTY: the buffer data has been modified
00019 //     and needs to be written to disk.
00020
00021 #include "types.h"
00022 #include "defs.h"
00023 #include "param.h"
00024 #include "spinlock.h"
00025 #include "sleeplock.h"
00026 #include "fs.h"
00027 #include "buf.h"
00028
00029 struct {
00030   struct spinlock lock;
00031   struct buf buf[NBUF];
00032
00033   // Linked list of all buffers, through prev/next.
00034   // head.next is most recently used.
00035   struct buf head;
00036 } bcache;
00037
00038 void
00039 binit(void)
00040 {
00041   struct buf *b;
00042
00043   initlock(&bcache.lock, "bcache");
00044
00045 //PAGEBREAK!
00046   // Create linked list of buffers
00047   bcache.head.prev = &bcache.head;
00048   bcache.head.next = &bcache.head;
00049   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
00050     b->next = bcache.head.next;
00051     b->prev = &bcache.head;
00052     initsleeplock(&b->lock, "buffer");
00053     bcache.head.next->prev = b;
00054     bcache.head.next = b;
00055   }
00056 }
00057
00058 // Look through buffer cache for block on device dev.
00059 // If not found, allocate a buffer.
00060 // In either case, return locked buffer.
00061 static struct buf*
00062 bget(uint dev, uint blockno)
00063 {
00064   struct buf *b;
00065
00066   acquire(&bcache.lock);
```

```
00067
00068   // Is the block already cached?
00069   for(b = bcache.head.next; b != &bcache.head; b = b->next){
00070     if(b->dev == dev && b->blockno == blockno){
00071       b->refcnt++;
00072       release(&bcache.lock);
00073       acquiresleep(&b->lock);
00074       return b;
00075     }
00076   }
00077
00078   // Not cached; recycle an unused buffer.
00079   // Even if refcnt==0, B_DIRTY indicates a buffer is in use
00080   // because log.c has modified it but not yet committed it.
00081   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
00082     if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
00083       b->dev = dev;
00084       b->blockno = blockno;
00085       b->flags = 0;
00086       b->refcnt = 1;
00087       release(&bcache.lock);
00088       acquiresleep(&b->lock);
00089       return b;
00090     }
00091   }
00092   panic("bget: no buffers");
00093 }
00094
00095 // Return a locked buf with the contents of the indicated block.
00096 struct buf*
00097 bread(uint dev, uint blockno)
00098 {
00099   struct buf *b;
00100
00101   b = bget(dev, blockno);
00102   if((b->flags & B_VALID) == 0) {
00103     iderw(b);
00104   }
00105   return b;
00106 }
00107
00108 // Write b's contents to disk.  Must be locked.
00109 void
00110 bwrite(struct buf *b)
00111 {
00112   if(!holdingsleep(&b->lock))
00113     panic("bwrite");
00114   b->flags |= B_DIRTY;
00115   iderw(b);
00116 }
00117
00118 // Release a locked buffer.
00119 // Move to the head of the MRU list.
00120 void
00121 brelse(struct buf *b)
00122 {
00123   if(!holdingsleep(&b->lock))
00124     panic("brelse");
00125
00126   releasesleep(&b->lock);
00127
00128   acquire(&bcache.lock);
00129   b->refcnt--;
00130   if (b->refcnt == 0) {
00131     // no one is waiting for it.
00132     b->next->prev = b->prev;
00133     b->prev->next = b->next;
00134     b->next = bcache.head.next;
00135     b->prev = &bcache.head;
00136     bcache.head.next->prev = b;
00137     bcache.head.next = b;
00138   }
00139
00140   release(&bcache.lock);
00141 }
00142 //PAGEBREAK!
00143 // Blank page.
00144
```

5.5 bio.d File Reference

5.6 bio.d

Go to the documentation of this file.

```
00001 bio.o: bio.c /usr/include/stdc-predef.h types.h defs.h param.h spinlock.h \
00002  sleeplock.h fs.h buf.h
```

5.7 bootasm.d File Reference

5.8 bootasm.d

Go to the documentation of this file.

```
00001 bootasm.o: bootasm.S asm.h memlayout.h mmu.h
```

5.9 bootmain.c File Reference

```
#include "types.h"
#include "elf.h"
#include "x86.h"
#include "memlayout.h"
```

Macros

- #define SECTSIZE 512

Functions

- void bootmain (void)
- void readsect (void *dst, uint offset)
- void readseg (uchar *, uint, uint)
- void waitdisk (void)

5.9.1 Macro Definition Documentation

5.9.1.1 SECTSIZE

```
#define SECTSIZE 512
```

Definition at line 13 of file bootmain.c.

5.9.2 Function Documentation

5.9.2.1 bootmain()

```
void bootmain (
              void  )
```

Definition at line 18 of file bootmain.c.

```
00019 {
00020   struct elfhdr *elf;
00021   struct proghdr *ph, *eph;
00022   void (*entry)(void);
00023   uchar* pa;
00024
00025   elf = (struct elfhdr*)0x10000;  // scratch space
00026
00027   // Read 1st page off disk
00028   readseg((uchar*)elf, 4096, 0);
00029
00030   // Is this an ELF executable?
00031   if(elf->magic != ELF_MAGIC)
00032     return;  // let bootasm.S handle error
00033
00034   // Load each program segment (ignores ph flags).
00035   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
00036   eph = ph + elf->phnum;
00037   for(; ph < eph; ph++){
00038     pa = (uchar*)ph->paddr;
00039     readseg(pa, ph->filesz, ph->off);
00040     if(ph->memsz > ph->filesz)
00041       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
00042   }
00043
00044   // Call the entry point from the ELF header.
00045   // Does not return!
00046   entry = (void(*)(void))(elf->entry);
00047   entry();
00048 }
```

5.9.2.2 readsect()

```
void readsect (
              void * dst,
              uint offset )
```

Definition at line 60 of file bootmain.c.

```
00061 {
00062   // Issue command.
00063   waitdisk();
00064   outb(0x1F2, 1);   // count = 1
00065   outb(0x1F3, offset);
00066   outb(0x1F4, offset » 8);
00067   outb(0x1F5, offset » 16);
00068   outb(0x1F6, (offset » 24) | 0xE0);
00069   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
00070
00071   // Read data.
00072   waitdisk();
00073   insl(0x1F0, dst, SECTSIZE/4);
00074 }
```

Referenced by readseg().

5.9.2.3 readseg()

```
void readseg (
              uchar * pa,
              uint count,
              uint offset )
```

Definition at line 79 of file bootmain.c.

```
00080 {
00081   uchar* epa;
00082
00083   epa = pa + count;
00084
00085   // Round down to sector boundary.
00086   pa -= offset % SECTSIZE;
00087
00088   // Translate from bytes to sectors; kernel starts at sector 1.
00089   offset = (offset / SECTSIZE) + 1;
00090
00091   // If this is too slow, we could read lots of sectors at a time.
00092   // We'd write more to memory than asked, but it doesn't matter --
00093   // we load in increasing order.
00094   for(; pa < epa; pa += SECTSIZE, offset++)
00095     readsect(pa, offset);
00096 }
```

Referenced by bootmain().

5.9.2.4 waitdisk()

```
void waitdisk (
              void  )
```

Definition at line 51 of file bootmain.c.

```
00052 {
00053   // Wait for disk ready.
00054   while((inb(0x1F7) & 0xC0) != 0x40)
00055     ;
00056 }
```

Referenced by readsect().

5.10 bootmain.c

Go to the documentation of this file.

```
00001 // Boot loader.
00002 //
00003 // Part of the boot block, along with bootasm.S, which calls bootmain().
00004 // bootasm.S has put the processor into protected 32-bit mode.
00005 // bootmain() loads an ELF kernel image from the disk starting at
00006 // sector 1 and then jumps to the kernel entry routine.
00007
00008 #include "types.h"
00009 #include "elf.h"
00010 #include "x86.h"
00011 #include "memlayout.h"
00012
00013 #define SECTSIZE  512
00014
00015 void readseg(uchar*, uint, uint);
00016
00017 void
00018 bootmain(void)
00019 {
00020   struct elfhdr *elf;
00021   struct proghdr *ph, *eph;
```

```
00022   void (*entry)(void);
00023   uchar* pa;
00024
00025   elf = (struct elfhdr*)0x10000;  // scratch space
00026
00027   // Read 1st page off disk
00028   readseg((uchar*)elf, 4096, 0);
00029
00030   // Is this an ELF executable?
00031   if(elf->magic != ELF_MAGIC)
00032     return;  // let bootasm.S handle error
00033
00034   // Load each program segment (ignores ph flags).
00035   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
00036   eph = ph + elf->phnum;
00037   for(; ph < eph; ph++){
00038     pa = (uchar*)ph->paddr;
00039     readseg(pa, ph->filesz, ph->off);
00040     if(ph->memsz > ph->filesz)
00041       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
00042   }
00043
00044   // Call the entry point from the ELF header.
00045   // Does not return!
00046   entry = (void(*)(void))(elf->entry);
00047   entry();
00048 }
00049
00050 void
00051 waitdisk(void)
00052 {
00053   // Wait for disk ready.
00054   while((inb(0x1F7) & 0xC0) != 0x40)
00055     ;
00056 }
00057
00058 // Read a single sector at offset into dst.
00059 void
00060 readsect(void *dst, uint offset)
00061 {
00062   // Issue command.
00063   waitdisk();
00064   outb(0x1F2, 1);   // count = 1
00065   outb(0x1F3, offset);
00066   outb(0x1F4, offset >> 8);
00067   outb(0x1F5, offset >> 16);
00068   outb(0x1F6, (offset >> 24) | 0xE0);
00069   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
00070
00071   // Read data.
00072   waitdisk();
00073   insl(0x1F0, dst, SECTSIZE/4);
00074 }
00075
00076 // Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
00077 // Might copy more than asked.
00078 void
00079 readseg(uchar* pa, uint count, uint offset)
00080 {
00081   uchar* epa;
00082
00083   epa = pa + count;
00084
00085   // Round down to sector boundary.
00086   pa -= offset % SECTSIZE;
00087
00088   // Translate from bytes to sectors; kernel starts at sector 1.
00089   offset = (offset / SECTSIZE) + 1;
00090
00091   // If this is too slow, we could read lots of sectors at a time.
00092   // We'd write more to memory than asked, but it doesn't matter --
00093   // we load in increasing order.
00094   for(; pa < epa; pa += SECTSIZE, offset++)
00095     readsect(pa, offset);
00096 }
```

5.11 bootmain.d File Reference

5.12 bootmain.d

Go to the documentation of this file.

```
00001 bootmain.o: bootmain.c types.h elf.h x86.h memlayout.h
```

5.13 buf.h File Reference

Classes

- struct buf

Macros

- #define B_DIRTY 0x4
- #define B_VALID 0x2

5.13.1 Macro Definition Documentation

5.13.1.1 B_DIRTY

```
#define B_DIRTY 0x4
```

Definition at line 13 of file buf.h.

5.13.1.2 B_VALID

```
#define B_VALID 0x2
```

Definition at line 12 of file buf.h.

5.14 buf.h

Go to the documentation of this file.

```
00001 struct buf {
00002   int flags;
00003   uint dev;
00004   uint blockno;
00005   struct sleeplock lock;
00006   uint refcnt;
00007   struct buf *prev; // LRU cache list
00008   struct buf *next;
00009   struct buf *qnext; // disk queue
00010   uchar data[BSIZE];
00011 };
00012 #define B_VALID 0x2  // buffer has been read from disk
00013 #define B_DIRTY 0x4  // buffer needs to be written to disk
00014
```

5.15 cat.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- void cat (int fd)
- int main (int argc, char *argv[])

Variables

- char buf [512]

5.15.1 Function Documentation

5.15.1.1 cat()

```
void cat (
            int fd )
```

Definition at line 8 of file cat.c.

```
00009 {
00010   int n;
00011
00012   while((n = read(fd, buf, sizeof(buf))) > 0) {
00013     if (write(1, buf, n) != n) {
00014       printf(1, "cat: write error\n");
00015       exit();
00016     }
00017   }
00018   if(n < 0){
00019     printf(1, "cat: read error\n");
00020     exit();
00021   }
00022 }
```

Referenced by main().

5.15.1.2 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 25 of file cat.c.

```
00026 {
00027   int fd, i;
00028
00029   if(argc <= 1){
00030     cat(0);
00031     exit();
00032   }
00033
00034   for(i = 1; i < argc; i++){
00035     if((fd = open(argv[i], 0)) < 0){
00036       printf(1, "cat: cannot open %s\n", argv[i]);
00037       exit();
00038     }
00039     cat(fd);
00040     close(fd);
00041   }
00042   exit();
00043 }
```

5.15.2 Variable Documentation

5.15.2.1 buf

```
char buf[512]
```

Definition at line 5 of file cat.c.

5.16 cat.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004
00005 char buf[512];
00006
00007 void
00008 cat(int fd)
00009 {
00010   int n;
00011
00012   while((n = read(fd, buf, sizeof(buf))) > 0) {
00013     if (write(1, buf, n) != n) {
00014       printf(1, "cat: write error\n");
00015       exit();
00016     }
00017   }
00018   if(n < 0){
00019     printf(1, "cat: read error\n");
00020     exit();
00021   }
00022 }
00023
00024 int
00025 main(int argc, char *argv[])
00026 {
00027   int fd, i;
00028
```

```
00029   if(argc <= 1){
00030     cat(0);
00031     exit();
00032   }
00033
00034   for(i = 1; i < argc; i++){
00035     if((fd = open(argv[i], 0)) < 0){
00036       printf(1, "cat: cannot open %s\n", argv[i]);
00037       exit();
00038     }
00039     cat(fd);
00040     close(fd);
00041   }
00042   exit();
00043 }
```

5.17 cat.d File Reference

5.18 cat.d

Go to the documentation of this file.

```
00001 cat.o: cat.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.19 console.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "traps.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "file.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
```

Macros

- #define BACKSPACE 0x100
- #define C(x) ((x)-'@')
- #define CRTPORT 0x3d4
- #define INPUT_BUF 128

Functions

- static void cgaputc (int c)
- void consoleinit (void)
- void consoleintr (int(∗getc)(void))
- int consoleread (struct inode ∗ip, char ∗dst, int n)
- int consolewrite (struct inode ∗ip, char ∗buf, int n)
- static void consputc (int)
- void cprintf (char ∗fmt,...)
- void panic (char ∗s)
- static void printint (int xx, int base, int sign)

Variables

- struct {
  struct spinlock lock
  int locking
  } cons

- static ushort * crt = (ushort*)P2V(0xb8000)
- struct {
  char buf [INPUT_BUF]
  uint e
  uint r
  uint w
  } input

- static int panicked = 0

5.19.1 Macro Definition Documentation

5.19.1.1 BACKSPACE

```
#define BACKSPACE 0x100
```

Definition at line 127 of file console.c.

5.19.1.2 C

```
#define C(
              x ) ((x)-'@')
```

Definition at line 189 of file console.c.

5.19.1.3 CRTPORT

```
#define CRTPORT 0x3d4
```

Definition at line 128 of file console.c.

5.19.1.4 INPUT_BUF

```
#define INPUT_BUF 128
```

Definition at line 181 of file console.c.

5.19.2 Function Documentation

5.19.2.1 cgaputc()

```
static void cgaputc (
            int c )  [static]
```

Definition at line 132 of file console.c.

```
00133 {
00134   int pos;
00135
00136   // Cursor position: col + 80*row.
00137   outb(CRTPORT, 14);
00138   pos = inb(CRTPORT+1) « 8;
00139   outb(CRTPORT, 15);
00140   pos |= inb(CRTPORT+1);
00141
00142   if(c == '\n')
00143     pos += 80 - pos%80;
00144   else if(c == BACKSPACE){
00145     if(pos > 0) --pos;
00146   } else
00147     crt[pos++] = (c&0xff) | 0x0700;  // black on white
00148
00149   if(pos < 0 || pos > 25*80)
00150     panic("pos under/overflow");
00151
00152   if((pos/80) >= 24){  // Scroll up.
00153     memmove(crt, crt+80, sizeof(crt[0])*23*80);
00154     pos -= 80;
00155     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
00156   }
00157
00158   outb(CRTPORT, 14);
00159   outb(CRTPORT+1, pos»8);
00160   outb(CRTPORT, 15);
00161   outb(CRTPORT+1, pos);
00162   crt[pos] = ' ' | 0x0700;
00163 }
```

Referenced by consputc().

5.19.2.2 consoleinit()

```
void consoleinit (
            void  )
```

Definition at line 289 of file console.c.

```
00290 {
00291   initlock(&cons.lock, "console");
00292
00293   devsw[CONSOLE].write = consolewrite;
00294   devsw[CONSOLE].read = consoleread;
00295   cons.locking = 1;
00296
00297   ioapicenable(IRQ_KBD, 0);
00298 }
```

5.19.2.3 consoleintr()

```
void consoleintr (
             int(*)(void) getc )
```

Definition at line 192 of file console.c.

```
00193 {
00194   int c, doprocdump = 0;
00195
00196   acquire(&cons.lock);
00197   while((c = getc()) >= 0){
00198     switch(c){
00199     case C('P'):  // Process listing.
00200       // procdump() locks cons.lock indirectly; invoke later
00201       doprocdump = 1;
00202       break;
00203     case C('U'):  // Kill line.
00204       while(input.e != input.w &&
00205             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
00206         input.e--;
00207         consputc(BACKSPACE);
00208       }
00209       break;
00210     case C('H'): case '\x7f':  // Backspace
00211       if(input.e != input.w){
00212         input.e--;
00213         consputc(BACKSPACE);
00214       }
00215       break;
00216     default:
00217       if(c != 0 && input.e-input.r < INPUT_BUF){
00218         c = (c == '\r') ? '\n' : c;
00219         input.buf[input.e++ % INPUT_BUF] = c;
00220         consputc(c);
00221         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
00222           input.w = input.e;
00223           wakeup(&input.r);
00224         }
00225       }
00226       break;
00227     }
00228   }
00229   release(&cons.lock);
00230   if(doprocdump) {
00231     procdump();  // now call procdump() wo. cons.lock held
00232   }
00233 }
```

Referenced by kbdintr(), and uartintr().

5.19.2.4 consoleread()

```
int consoleread (
             struct inode * ip,
             char * dst,
             int n )
```

Definition at line 236 of file console.c.

```
00237 {
00238   uint target;
00239   int c;
00240
00241   iunlock(ip);
00242   target = n;
00243   acquire(&cons.lock);
00244   while(n > 0){
00245     while(input.r == input.w){
00246       if(myproc()->killed){
00247         release(&cons.lock);
00248         ilock(ip);
00249         return -1;
00250       }
00251       sleep(&input.r, &cons.lock);
```

```
00252     }
00253     c = input.buf[input.r++ % INPUT_BUF];
00254     if(c == C('D')){  // EOF
00255       if(n < target){
00256         // Save ^D for next time, to make sure
00257         // caller gets a 0-byte result.
00258         input.r--;
00259       }
00260       break;
00261     }
00262     *dst++ = c;
00263     --n;
00264     if(c == '\n')
00265       break;
00266   }
00267   release(&cons.lock);
00268   ilock(ip);
00269
00270   return target - n;
00271 }
```

Referenced by consoleinit().

5.19.2.5 consolewrite()

```
int consolewrite (
            struct inode * ip,
            char * buf,
            int n )
```

Definition at line 274 of file console.c.

```
00275 {
00276   int i;
00277
00278   iunlock(ip);
00279   acquire(&cons.lock);
00280   for(i = 0; i < n; i++)
00281     consputc(buf[i] & 0xff);
00282   release(&cons.lock);
00283   ilock(ip);
00284
00285   return n;
00286 }
```

Referenced by consoleinit().

5.19.2.6 consputc()

```
void consputc (
            int c )  [static]
```

Definition at line 166 of file console.c.

```
00167 {
00168   if(panicked){
00169     cli();
00170     for(;;)
00171       ;
00172   }
00173
00174   if(c == BACKSPACE){
00175     uartputc('\b'); uartputc(' '); uartputc('\b');
00176   } else
00177     uartputc(c);
00178   cgaputc(c);
00179 }
```

Referenced by consoleintr(), consolewrite(), cprintf(), and printint().

5.19.2.7 cprintf()

```
void cprintf (
              char * fmt,
               ...  )
```

Definition at line 55 of file console.c.

```
00056 {
00057   int i, c, locking;
00058   uint *argp;
00059   char *s;
00060
00061   locking = cons.locking;
00062   if(locking)
00063     acquire(&cons.lock);
00064
00065   if (fmt == 0)
00066     panic("null fmt");
00067
00068   argp = (uint*)(void*)(&fmt + 1);
00069   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
00070     if(c != '%'){
00071       consputc(c);
00072       continue;
00073     }
00074     c = fmt[++i] & 0xff;
00075     if(c == 0)
00076       break;
00077     switch(c){
00078     case 'd':
00079       printint(*argp++, 10, 1);
00080       break;
00081     case 'x':
00082     case 'p':
00083       printint(*argp++, 16, 0);
00084       break;
00085     case 's':
00086       if((s = (char*)*argp++) == 0)
00087         s = "(null)";
00088       for(; *s; s++)
00089         consputc(*s);
00090       break;
00091     case '%':
00092       consputc('%');
00093       break;
00094     default:
00095       // Print unknown % sequence to draw attention.
00096       consputc('%');
00097       consputc(c);
00098       break;
00099     }
00100   }
00101
00102   if(locking)
00103     release(&cons.lock);
00104 }
```

Referenced by allocuvm(), cps(), exec(), iinit(), ioapicinit(), panic(), procdump(), syscall(), and trap().

5.19.2.8 panic()

```
void panic (
              char * s )
```

Definition at line 107 of file console.c.

```
00108 {
00109   int i;
00110   uint pcs[10];
00111
00112   cli();
00113   cons.locking = 0;
00114   // use lapiccpunum so that we can call panic from mycpu()
00115   cprintf("lapicid %d: panic: ", lapicid());
```

```
00116   cprintf(s);
00117   cprintf("\n");
00118   getcallerpcs(&s, pcs);
00119   for(i=0; i<10; i++)
00120     cprintf(" %p", pcs[i]);
00121   panicked = 1; // freeze other CPU
00122   for(;;)
00123     ;
00124 }
```

Referenced by acquire(), balloc(), bfree(), bget(), bmap(), brelse(), bwrite(), cgaputc(), clearpteu(), copyuvm(), cprintf(), create(), deallocuvm(), dirlink(), dirlookup(), end_op(), exit(), fileclose(), filedup(), fileread(), filewrite(), freevm(), ialloc(), iderw(), idestart(), iget(), ilock(), initlog(), inituvm(), isdirempty(), iunlock(), kfree(), loaduvm(), log_write(), mappages(), mpinit(), mycpu(), popcli(), release(), sched(), setupkvm(), sleep(), switchuvm(), sys_unlink(), trap(), and userinit().

5.19.2.9 printint()

```
static void printint (
              int xx,
              int base,
              int sign )  [static]
```

Definition at line 28 of file console.c.

```
00029 {
00030   static char digits[] = "0123456789abcdef";
00031   char buf[16];
00032   int i;
00033   uint x;
00034
00035   if(sign && (sign = xx < 0))
00036     x = -xx;
00037   else
00038     x = xx;
00039
00040   i = 0;
00041   do{
00042     buf[i++] = digits[x % base];
00043   }while((x /= base) != 0);
00044
00045   if(sign)
00046     buf[i++] = '-';
00047
00048   while(--i >= 0)
00049     consputc(buf[i]);
00050 }
```

Referenced by cprintf().

5.19.3 Variable Documentation

5.19.3.1 buf

```
char buf[INPUT_BUF]
```

Definition at line 183 of file console.c.

5.19.3.2

```
struct { ... } cons [static]
```

Referenced by consoleinit(), consoleintr(), consoleread(), consolewrite(), cprintf(), and panic().

5.19.3.3 crt

```
ushort* crt = (ushort*)P2V(0xb8000) [static]
```

Definition at line 129 of file console.c.

Referenced by cgaputc().

5.19.3.4 e

```
uint e
```

Definition at line 186 of file console.c.

Referenced by mpinit(), and mpsearch1().

5.19.3.5

```
struct { ... } input
```

Referenced by consoleintr(), and consoleread().

5.19.3.6 lock

```
struct spinlock lock
```

Definition at line 23 of file console.c.

5.19.3.7 locking

```
int locking
```

Definition at line 24 of file console.c.

Referenced by cprintf().

5.19.3.8 panicked

```
int panicked = 0  [static]
```

Definition at line 20 of file console.c.

Referenced by consputc(), and panic().

5.19.3.9 r

```
uint r
```

Definition at line 184 of file console.c.

Referenced by cmostime(), fileread(), filewrite(), fill_rtcdate(), holding(), holdingsleep(), idewait(), iput(), kalloc(), kfree(), and stat().

5.19.3.10 w

```
uint w
```

Definition at line 185 of file console.c.

Referenced by wc().

5.20 console.c

Go to the documentation of this file.

```
00001 // Console input and output.
00002 // Input is from the keyboard or serial port.
00003 // Output is written to the screen and serial port.
00004
00005 #include "types.h"
00006 #include "defs.h"
00007 #include "param.h"
00008 #include "traps.h"
00009 #include "spinlock.h"
00010 #include "sleeplock.h"
00011 #include "fs.h"
00012 #include "file.h"
00013 #include "memlayout.h"
00014 #include "mmu.h"
00015 #include "proc.h"
00016 #include "x86.h"
00017
00018 static void consputc(int);
00019
00020 static int panicked = 0;
00021
00022 static struct {
00023   struct spinlock lock;
00024   int locking;
00025 } cons;
00026
00027 static void
00028 printint(int xx, int base, int sign)
00029 {
00030   static char digits[] = "0123456789abcdef";
```

```
00031   char buf[16];
00032   int i;
00033   uint x;
00034
00035   if(sign && (sign = xx < 0))
00036     x = -xx;
00037   else
00038     x = xx;
00039
00040   i = 0;
00041   do{
00042     buf[i++] = digits[x % base];
00043   }while((x /= base) != 0);
00044
00045   if(sign)
00046     buf[i++] = '-';
00047
00048   while(--i >= 0)
00049     consputc(buf[i]);
00050 }
00051 //PAGEBREAK: 50
00052
00053 // Print to the console. only understands %d, %x, %p, %s.
00054 void
00055 cprintf(char *fmt, ...)
00056 {
00057   int i, c, locking;
00058   uint *argp;
00059   char *s;
00060
00061   locking = cons.locking;
00062   if(locking)
00063     acquire(&cons.lock);
00064
00065   if (fmt == 0)
00066     panic("null fmt");
00067
00068   argp = (uint*)(void*)(&fmt + 1);
00069   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
00070     if(c != '%'){
00071       consputc(c);
00072       continue;
00073     }
00074     c = fmt[++i] & 0xff;
00075     if(c == 0)
00076       break;
00077     switch(c){
00078     case 'd':
00079       printint(*argp++, 10, 1);
00080       break;
00081     case 'x':
00082     case 'p':
00083       printint(*argp++, 16, 0);
00084       break;
00085     case 's':
00086       if((s = (char*)*argp++) == 0)
00087         s = "(null)";
00088       for(; *s; s++)
00089         consputc(*s);
00090       break;
00091     case '%':
00092       consputc('%');
00093       break;
00094     default:
00095       // Print unknown % sequence to draw attention.
00096       consputc('%');
00097       consputc(c);
00098       break;
00099     }
00100   }
00101
00102   if(locking)
00103     release(&cons.lock);
00104 }
00105
00106 void
00107 panic(char *s)
00108 {
00109   int i;
00110   uint pcs[10];
00111
00112   cli();
00113   cons.locking = 0;
00114   // use lapiccpunum so that we can call panic from mycpu()
00115   cprintf("lapicid %d: panic: ", lapicid());
00116   cprintf(s);
00117   cprintf("\n");
```

```
00118   getcallerpcs(&s, pcs);
00119   for(i=0; i<10; i++)
00120     cprintf(" %p", pcs[i]);
00121   panicked = 1; // freeze other CPU
00122   for(;;)
00123     ;
00124 }
00125
00126 //PAGEBREAK: 50
00127 #define BACKSPACE 0x100
00128 #define CRTPORT 0x3d4
00129 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
00130
00131 static void
00132 cgaputc(int c)
00133 {
00134   int pos;
00135
00136   // Cursor position: col + 80*row.
00137   outb(CRTPORT, 14);
00138   pos = inb(CRTPORT+1) << 8;
00139   outb(CRTPORT, 15);
00140   pos |= inb(CRTPORT+1);
00141
00142   if(c == '\n')
00143     pos += 80 - pos%80;
00144   else if(c == BACKSPACE){
00145     if(pos > 0) --pos;
00146   } else
00147     crt[pos++] = (c&0xff) | 0x0700;  // black on white
00148
00149   if(pos < 0 || pos > 25*80)
00150     panic("pos under/overflow");
00151
00152   if((pos/80) >= 24){  // Scroll up.
00153     memmove(crt, crt+80, sizeof(crt[0])*23*80);
00154     pos -= 80;
00155     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
00156   }
00157
00158   outb(CRTPORT, 14);
00159   outb(CRTPORT+1, pos>>8);
00160   outb(CRTPORT, 15);
00161   outb(CRTPORT+1, pos);
00162   crt[pos] = ' ' | 0x0700;
00163 }
00164
00165 void
00166 consputc(int c)
00167 {
00168   if(panicked){
00169     cli();
00170     for(;;)
00171       ;
00172   }
00173
00174   if(c == BACKSPACE){
00175     uartputc('\b'); uartputc(' '); uartputc('\b');
00176   } else
00177     uartputc(c);
00178   cgaputc(c);
00179 }
00180
00181 #define INPUT_BUF 128
00182 struct {
00183   char buf[INPUT_BUF];
00184   uint r;  // Read index
00185   uint w;  // Write index
00186   uint e;  // Edit index
00187 } input;
00188
00189 #define C(x)  ((x)-'@')  // Control-x
00190
00191 void
00192 consoleintr(int (*getc)(void))
00193 {
00194   int c, doprocdump = 0;
00195
00196   acquire(&cons.lock);
00197   while((c = getc()) >= 0){
00198     switch(c){
00199     case C('P'):  // Process listing.
00200       // procdump() locks cons.lock indirectly; invoke later
00201       doprocdump = 1;
00202       break;
00203     case C('U'):  // Kill line.
00204       while(input.e != input.w &&
```

```
00205             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
00206         input.e--;
00207         consputc(BACKSPACE);
00208       }
00209       break;
00210     case C('H'): case '\x7f':  // Backspace
00211       if(input.e != input.w){
00212         input.e--;
00213         consputc(BACKSPACE);
00214       }
00215       break;
00216     default:
00217       if(c != 0 && input.e-input.r < INPUT_BUF){
00218         c = (c == '\r') ? '\n' : c;
00219         input.buf[input.e++ % INPUT_BUF] = c;
00220         consputc(c);
00221         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
00222           input.w = input.e;
00223           wakeup(&input.r);
00224         }
00225       }
00226       break;
00227     }
00228   }
00229   release(&cons.lock);
00230   if(doprocdump) {
00231     procdump();  // now call procdump() wo. cons.lock held
00232   }
00233 }
00234
00235 int
00236 consoleread(struct inode *ip, char *dst, int n)
00237 {
00238   uint target;
00239   int c;
00240
00241   iunlock(ip);
00242   target = n;
00243   acquire(&cons.lock);
00244   while(n > 0){
00245     while(input.r == input.w){
00246       if(myproc()->killed){
00247         release(&cons.lock);
00248         ilock(ip);
00249         return -1;
00250       }
00251       sleep(&input.r, &cons.lock);
00252     }
00253     c = input.buf[input.r++ % INPUT_BUF];
00254     if(c == C('D')){  // EOF
00255       if(n < target){
00256         // Save ^D for next time, to make sure
00257         // caller gets a 0-byte result.
00258         input.r--;
00259       }
00260       break;
00261     }
00262     *dst++ = c;
00263     --n;
00264     if(c == '\n')
00265       break;
00266   }
00267   release(&cons.lock);
00268   ilock(ip);
00269
00270   return target - n;
00271 }
00272
00273 int
00274 consolewrite(struct inode *ip, char *buf, int n)
00275 {
00276   int i;
00277
00278   iunlock(ip);
00279   acquire(&cons.lock);
00280   for(i = 0; i < n; i++)
00281     consputc(buf[i] & 0xff);
00282   release(&cons.lock);
00283   ilock(ip);
00284
00285   return n;
00286 }
00287
00288 void
00289 consoleinit(void)
00290 {
00291   initlock(&cons.lock, "console");
```

```
00292
00293   devsw[CONSOLE].write = consolewrite;
00294   devsw[CONSOLE].read = consoleread;
00295   cons.locking = 1;
00296
00297   ioapicenable(IRQ_KBD, 0);
00298 }
00299
```

5.21 console.d File Reference

5.22 console.d

Go to the documentation of this file.

```
00001 console.o: console.c /usr/include/stdc-predef.h types.h defs.h param.h \
00002  traps.h spinlock.h sleeplock.h fs.h file.h memlayout.h mmu.h proc.h \
00003  x86.h
```

5.23 date.h File Reference

Classes

- struct rtcdate

5.24 date.h

Go to the documentation of this file.

```
00001 struct rtcdate {
00002   uint second;
00003   uint minute;
00004   uint hour;
00005   uint day;
00006   uint month;
00007   uint year;
00008 };
```

5.25 defs.h File Reference

Macros

- #define NELEM(x) (sizeof(x)/sizeof((x)[0]))

Functions

- void acquire (struct spinlock *)
- void acquiresleep (struct sleeplock *)
- int allocuvm (pde_t *, uint, uint)
- int argint (int, int *)
- int argptr (int, char **, int)
- int argstr (int, char **)
- void begin_op ()
- void binit (void)
- struct buf * bread (uint, uint)
- void brelse (struct buf *)
- void bwrite (struct buf *)
- int chpr (int pid, int priority)
- void clearpteu (pde_t *pgdir, char *uva)
- void cmostime (struct rtcdate *r)
- void consoleinit (void)
- void consoleintr (int(*)(void))
- int copyout (pde_t *, uint, void *, uint)
- pde_t * copyuvm (pde_t *, uint)
- void cprintf (char *,...)
- int cps (void)
- int cpuid (void)
- int deallocuvm (pde_t *, uint, uint)
- int dirlink (struct inode *, char *, uint)
- struct inode * dirlookup (struct inode *, char *, uint *)
- void end_op ()
- int exec (char *, char **)
- void exit (void)
- int fetchint (uint, int *)
- int fetchstr (uint, char **)
- struct file * filealloc (void)
- void fileclose (struct file *)
- struct file * filedup (struct file *)
- void fileinit (void)
- int fileread (struct file *, char *, int n)
- int filestat (struct file *, struct stat *)
- int filewrite (struct file *, char *, int n)
- int fork (void)
- void freevm (pde_t *)
- void getcallerpcs (void *, uint *)
- int getprocs (int max, struct uproc *)
- int growproc (int)
- int holding (struct spinlock *)
- int holdingsleep (struct sleeplock *)
- struct inode * ialloc (uint, short)
- void ideinit (void)
- void ideintr (void)
- void iderw (struct buf *)
- void idtinit (void)
- struct inode * idup (struct inode *)
- void iinit (int dev)
- void ilock (struct inode *)
- void initlock (struct spinlock *, char *)
- void initlog (int dev)

- void initsleeplock (struct sleeplock *, char *)
- void inituvm (pde_t *, char *, uint)
- void ioapicenable (int irq, int cpu)
- void ioapicinit (void)
- void iput (struct inode *)
- void iunlock (struct inode *)
- void iunlockput (struct inode *)
- void iupdate (struct inode *)
- char * kalloc (void)
- void kbdintr (void)
- void kfree (char *)
- int kill (int)
- void kinit1 (void *, void *)
- void kinit2 (void *, void *)
- void kvmalloc (void)
- void lapiceoi (void)
- int lapicid (void)
- void lapicinit (void)
- void lapicstartap (uchar, uint)
- int loaduvm (pde_t *, char *, struct inode *, uint, uint)
- void log_write (struct buf *)
- int memcmp (const void *, const void *, uint)
- void * memmove (void *, const void *, uint)
- void * memset (void *, int, uint)
- void microdelay (int)
- void mpinit (void)
- struct cpu * mycpu (void)
- struct proc * myproc ()
- int namecmp (const char *, const char *)
- struct inode * namei (char *)
- struct inode * nameiparent (char *, char *)
- void panic (char *) __attribute__((noreturn))
- void picenable (int)
- void picinit (void)
- void pinit (void)
- int pipealloc (struct file **, struct file **)
- void pipeclose (struct pipe *, int)
- int piperead (struct pipe *, char *, int)
- int pipewrite (struct pipe *, char *, int)
- void popcli (void)
- void procdump (void)
- void pushcli (void)
- int readi (struct inode *, char *, uint, uint)
- void readsb (int dev, struct superblock *sb)
- void release (struct spinlock *)
- void releasesleep (struct sleeplock *)
- char * safestrcpy (char *, const char *, int)
- void sched (void)
- void scheduler (void) __attribute__((noreturn))
- void seginit (void)
- void setproc (struct proc *)
- pde_t * setupkvm (void)
- void sleep (void *, struct spinlock *)
- void stati (struct inode *, struct stat *)
- int strlen (const char *)

- int strncmp (const char *, const char *, uint)
- char * strncpy (char *, const char *, int)
- void switchkvm (void)
- void switchuvm (struct proc *)
- void swtch (struct context **, struct context *)
- void syscall (void)
- void timerinit (void)
- void tvinit (void)
- void uartinit (void)
- void uartintr (void)
- void uartputc (int)
- void userinit (void)
- char * uva2ka (pde_t *, char *)
- int wait (void)
- void wakeup (void *)
- int writei (struct inode *, char *, uint, uint)
- void yield (void)

Variables

- uchar ioapicid
- int ismp
- volatile uint * lapic
- uint ticks
- struct spinlock tickslock

5.25.1 Macro Definition Documentation

5.25.1.1 NELEM

```
#define NELEM(
              x ) (sizeof(x)/sizeof((x)[0]))
```

Definition at line 195 of file defs.h.

5.25.2 Function Documentation

5.25.2.1 acquire()

```
void acquire (
              struct spinlock * lk )
```

Definition at line 25 of file spinlock.c.

```
00026 {
00027   pushcli(); // disable interrupts to avoid deadlock.
00028   if(holding(lk))
00029     panic("acquire");
00030
00031   // The xchg is atomic.
00032   while(xchg(&lk->locked, 1) != 0)
00033     ;
00034
00035   // Tell the C compiler and the processor to not move loads or stores
00036   // past this point, to ensure that the critical section's memory
00037   // references happen after the lock is acquired.
00038   __sync_synchronize();
00039
00040   // Record info about lock acquisition for debugging.
00041   lk->cpu = mycpu();
00042   getcallerpcs(&lk, lk->pcs);
00043 }
```

Referenced by acquiresleep(), allocproc(), begin_op(), bget(), brelse(), chpr(), consoleintr(), consoleread(), consolewrite(), cprintf(), cps(), end_op(), exit(), filealloc(), fileclose(), filedup(), fork(), holdingsleep(), ideintr(), iderw(), idup(), iget(), iput(), kalloc(), kfree(), kill(), log_write(), pipeclose(), piperead(), pipewrite(), releasesleep(), scheduler(), sleep(), sys_sleep(), sys_uptime(), trap(), userinit(), wait(), wakeup(), and yield().

5.25.2.2 acquiresleep()

```
void acquiresleep (
              struct sleeplock * lk )
```

Definition at line 23 of file sleeplock.c.

```
00024 {
00025   acquire(&lk->lk);
00026   while (lk->locked) {
00027     sleep(lk, &lk->lk);
00028   }
00029   lk->locked = 1;
00030   lk->pid = myproc()->pid;
00031   release(&lk->lk);
00032 }
```

Referenced by bget(), ilock(), and iput().

5.25.2.3 allocuvm()

```
int allocuvm (
              pde_t * pgdir,
              uint oldsz,
              uint newsz )
```

Definition at line 222 of file vm.c.

```
00223 {
00224   char *mem;
00225   uint a;
00226
00227   if(newsz >= KERNBASE)
00228     return 0;
```

```
00229   if(newsz < oldsz)
00230     return oldsz;
00231
00232   a = PGROUNDUP(oldsz);
00233   for(; a < newsz; a += PGSIZE){
00234     mem = kalloc();
00235     if(mem == 0){
00236       cprintf("allocuvm out of memory\n");
00237       deallocuvm(pgdir, newsz, oldsz);
00238       return 0;
00239     }
00240     memset(mem, 0, PGSIZE);
00241     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
00242       cprintf("allocuvm out of memory (2)\n");
00243       deallocuvm(pgdir, newsz, oldsz);
00244       kfree(mem);
00245       return 0;
00246     }
00247   }
00248   return newsz;
00249 }
```

Referenced by exec(), and growproc().

5.25.2.4 argint()

```
int argint (
              int n,
              int * ip )
```

Definition at line 50 of file syscall.c.

```
00051 {
00052   return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
00053 }
```

Referenced by argfd(), argptr(), argstr(), sys_chpr(), sys_exec(), sys_getprocs(), sys_kill(), sys_mknod(), sys_open(), sys_read(), sys_sbrk(), sys_sleep(), and sys_write().

5.25.2.5 argptr()

```
int argptr (
              int n,
              char ** pp,
              int size )
```

Definition at line 59 of file syscall.c.

```
00060 {
00061   int i;
00062   struct proc *curproc = myproc();
00063
00064   if(argint(n, &i) < 0)
00065     return -1;
00066   if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
00067     return -1;
00068   *pp = (char*)i;
00069   return 0;
00070 }
```

Referenced by sys_fstat(), sys_getprocs(), sys_pipe(), sys_read(), and sys_write().

5.25.2.6 argstr()

```
int argstr (
              int n,
              char ** pp )
```

Definition at line 77 of file syscall.c.

```
00078 {
00079   int addr;
00080   if(argint(n, &addr) < 0)
00081     return -1;
00082   return fetchstr(addr, pp);
00083 }
```

Referenced by sys_chdir(), sys_exec(), sys_link(), sys_mkdir(), sys_mknod(), sys_open(), and sys_unlink().

5.25.2.7 begin_op()

```
void begin_op ( )
```

Definition at line 126 of file log.c.

```
00127 {
00128   acquire(&log.lock);
00129   while(1){
00130     if(log.committing){
00131       sleep(&log, &log.lock);
00132     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
00133       // this op might exhaust log space; wait for commit.
00134       sleep(&log, &log.lock);
00135     } else {
00136       log.outstanding += 1;
00137       release(&log.lock);
00138       break;
00139     }
00140   }
00141 }
```

Referenced by exec(), exit(), fileclose(), filewrite(), sys_chdir(), sys_link(), sys_mkdir(), sys_mknod(), sys_open(), and sys_unlink().

5.25.2.8 binit()

```
void binit (
              void  )
```

Definition at line 39 of file bio.c.

```
00040 {
00041   struct buf *b;
00042
00043   initlock(&bcache.lock, "bcache");
00044
00045 //PAGEBREAK!
00046   // Create linked list of buffers
00047   bcache.head.prev = &bcache.head;
00048   bcache.head.next = &bcache.head;
00049   for(b = bcache.buf; b < bcache.buf+NBUF; b++){
00050     b->next = bcache.head.next;
00051     b->prev = &bcache.head;
00052     initsleeplock(&b->lock, "buffer");
00053     bcache.head.next->prev = b;
00054     bcache.head.next = b;
00055   }
00056 }
```

5.25.2.9 bread()

```
struct buf * bread (
            uint dev,
            uint blockno )
```

Definition at line 97 of file bio.c.

```
00098 {
00099   struct buf *b;
00100
00101   b = bget(dev, blockno);
00102   if((b->flags & B_VALID) == 0) {
00103     iderw(b);
00104   }
00105   return b;
00106 }
```

Referenced by balloc(), bfree(), bmap(), bzero(), ialloc(), ilock(), install_trans(), itrunc(), iupdate(), read_head(), readi(), readsb(), write_head(), write_log(), and writei().

5.25.2.10 brelse()

```
void brelse (
            struct buf * b )
```

Definition at line 121 of file bio.c.

```
00122 {
00123   if(!holdingsleep(&b->lock))
00124     panic("brelse");
00125
00126   releasesleep(&b->lock);
00127
00128   acquire(&bcache.lock);
00129   b->refcnt--;
00130   if (b->refcnt == 0) {
00131     // no one is waiting for it.
00132     b->next->prev = b->prev;
00133     b->prev->next = b->next;
00134     b->next = bcache.head.next;
00135     b->prev = &bcache.head;
00136     bcache.head.next->prev = b;
00137     bcache.head.next = b;
00138   }
00139
00140   release(&bcache.lock);
00141 }
```

Referenced by balloc(), bfree(), bmap(), bzero(), ialloc(), ilock(), install_trans(), itrunc(), iupdate(), read_head(), readi(), readsb(), write_head(), write_log(), and writei().

5.25.2.11 bwrite()

```
void bwrite (
            struct buf * b )
```

Definition at line 110 of file bio.c.

```
00111 {
00112   if(!holdingsleep(&b->lock))
00113     panic("bwrite");
00114   b->flags |= B_DIRTY;
00115   iderw(b);
00116 }
```

Referenced by install_trans(), write_head(), and write_log().

5.25.2.12 chpr()

```
int chpr (
              int pid,
              int priority )
```

Definition at line 559 of file proc.c.

```
00560 {
00561   struct proc *p;
00562   acquire(&ptable.lock);
00563   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00564     if(p->pid == pid){
00565       p->priority = priority;
00566       break;
00567     }
00568   }
00569   release(&ptable.lock);
00570   return pid;
00571 }
```

Referenced by main(), and sys_chpr().

5.25.2.13 clearpteu()

```
void clearpteu (
              pde_t * pgdir,
              char * uva )
```

Definition at line 303 of file vm.c.

```
00304 {
00305   pte_t *pte;
00306
00307   pte = walkpgdir(pgdir, uva, 0);
00308   if(pte == 0)
00309     panic("clearpteu");
00310   *pte &= ~PTE_U;
00311 }
```

Referenced by exec().

5.25.2.14 cmostime()

```
void cmostime (
              struct rtcdate * r )
```

Definition at line 196 of file lapic.c.

```
00197 {
00198   struct rtcdate t1, t2;
00199   int sb, bcd;
00200
00201   sb = cmos_read(CMOS_STATB);
00202
00203   bcd = (sb & (1 << 2)) == 0;
00204
00205   // make sure CMOS doesn't modify time while we read it
00206   for(;;) {
00207     fill_rtcdate(&t1);
00208     if(cmos_read(CMOS_STATA) & CMOS_UIP)
00209         continue;
00210     fill_rtcdate(&t2);
00211     if(memcmp(&t1, &t2, sizeof(t1)) == 0)
00212       break;
00213   }
```

```
00214
00215   // convert
00216   if(bcd) {
00217 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
00218     CONV(second);
00219     CONV(minute);
00220     CONV(hour  );
00221     CONV(day   );
00222     CONV(month );
00223     CONV(year  );
00224 #undef     CONV
00225   }
00226
00227   *r = t1;
00228   r->year += 2000;
00229 }
```

5.25.2.15 consoleinit()

```
void consoleinit (
              void  )
```

Definition at line 289 of file console.c.

```
00290 {
00291   initlock(&cons.lock, "console");
00292
00293   devsw[CONSOLE].write = consolewrite;
00294   devsw[CONSOLE].read = consoleread;
00295   cons.locking = 1;
00296
00297   ioapicenable(IRQ_KBD, 0);
00298 }
```

5.25.2.16 consoleintr()

```
void consoleintr (
              int(*)(void) getc )
```

Definition at line 192 of file console.c.

```
00193 {
00194   int c, doprocdump = 0;
00195
00196   acquire(&cons.lock);
00197   while((c = getc()) >= 0){
00198     switch(c){
00199     case C('P'):  // Process listing.
00200       // procdump() locks cons.lock indirectly; invoke later
00201       doprocdump = 1;
00202       break;
00203     case C('U'):  // Kill line.
00204       while(input.e != input.w &&
00205             input.buf[(input.e-1) % INPUT_BUF] != '\n'){
00206         input.e--;
00207         consputc(BACKSPACE);
00208       }
00209       break;
00210     case C('H'): case '\x7f':  // Backspace
00211       if(input.e != input.w){
00212         input.e--;
00213         consputc(BACKSPACE);
00214       }
00215       break;
00216     default:
00217       if(c != 0 && input.e-input.r < INPUT_BUF){
00218         c = (c == '\r') ? '\n' : c;
00219         input.buf[input.e++ % INPUT_BUF] = c;
00220         consputc(c);
00221         if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
00222           input.w = input.e;
```

```
00223           wakeup(&input.r);
00224         }
00225       }
00226       break;
00227     }
00228   }
00229   release(&cons.lock);
00230   if(doprocdump) {
00231     procdump();  // now call procdump() wo. cons.lock held
00232   }
00233 }
```

Referenced by kbdintr(), and uartintr().

5.25.2.17 copyout()

```
int copyout (
              pde_t * pgdir,
              uint va,
              void * p,
              uint len )
```

Definition at line 366 of file vm.c.

```
00367 {
00368   char *buf, *pa0;
00369   uint n, va0;
00370
00371   buf = (char*)p;
00372   while(len > 0){
00373     va0 = (uint)PGROUNDDOWN(va);
00374     pa0 = uva2ka(pgdir, (char*)va0);
00375     if(pa0 == 0)
00376       return -1;
00377     n = PGSIZE - (va - va0);
00378     if(n > len)
00379       n = len;
00380     memmove(pa0 + (va - va0), buf, n);
00381     len -= n;
00382     buf += n;
00383     va = va0 + PGSIZE;
00384   }
00385   return 0;
00386 }
```

Referenced by exec().

5.25.2.18 copyuvm()

```
pde_t * copyuvm (
              pde_t * pgdir,
              uint sz )
```

Definition at line 316 of file vm.c.

```
00317 {
00318   pde_t *d;
00319   pte_t *pte;
00320   uint pa, i, flags;
00321   char *mem;
00322
00323   if((d = setupkvm()) == 0)
00324     return 0;
00325   for(i = 0; i < sz; i += PGSIZE){
00326     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
00327       panic("copyuvm: pte should exist");
00328     if(!(*pte & PTE_P))
```

```
00329       panic("copyuvm: page not present");
00330     pa = PTE_ADDR(*pte);
00331     flags = PTE_FLAGS(*pte);
00332     if((mem = kalloc()) == 0)
00333       goto bad;
00334     memmove(mem, (char*)P2V(pa), PGSIZE);
00335     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
00336       kfree(mem);
00337       goto bad;
00338     }
00339   }
00340   return d;
00341
00342 bad:
00343   freevm(d);
00344   return 0;
00345 }
```

Referenced by fork().

5.25.2.19 cprintf()

```
void cprintf (
              char * fmt,
               ...  )
```

Definition at line 55 of file console.c.

```
00056 {
00057   int i, c, locking;
00058   uint *argp;
00059   char *s;
00060
00061   locking = cons.locking;
00062   if(locking)
00063     acquire(&cons.lock);
00064
00065   if (fmt == 0)
00066     panic("null fmt");
00067
00068   argp = (uint*)(void*)(&fmt + 1);
00069   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
00070     if(c != '%'){
00071       consputc(c);
00072       continue;
00073     }
00074     c = fmt[++i] & 0xff;
00075     if(c == 0)
00076       break;
00077     switch(c){
00078     case 'd':
00079       printint(*argp++, 10, 1);
00080       break;
00081     case 'x':
00082     case 'p':
00083       printint(*argp++, 16, 0);
00084       break;
00085     case 's':
00086       if((s = (char*)*argp++) == 0)
00087         s = "(null)";
00088       for(; *s; s++)
00089         consputc(*s);
00090       break;
00091     case '%':
00092       consputc('%');
00093       break;
00094     default:
00095       // Print unknown % sequence to draw attention.
00096       consputc('%');
00097       consputc(c);
00098       break;
00099     }
00100   }
00101
00102   if(locking)
00103     release(&cons.lock);
00104 }
```

Referenced by allocuvm(), cps(), exec(), iinit(), ioapicinit(), panic(), procdump(), syscall(), and trap().

5.25.2.20 cps()

```
int cps (
              void  )
```

Definition at line 537 of file proc.c.

```
00538 {
00539 struct proc *p;
00540 //Enables interrupts on this processor.
00541 sti();
00542
00543 //Loop over process table looking for process with pid.
00544 acquire(&ptable.lock);
00545 cprintf("name \t pid \t state \t priority \n");
00546 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00547   if(p->state == SLEEPING)
00548     cprintf("%s \t %d \t SLEEPING \t %d \n ", p->name,p->pid,p->priority);
00549   else if(p->state == RUNNING)
00550     cprintf("%s \t %d \t RUNNING \t %d \n ", p->name,p->pid,p->priority);
00551   else if(p->state == RUNNABLE)
00552     cprintf("%s \t %d \t RUNNABLE \t %d \n ", p->name,p->pid,p->priority);
00553 }
00554 release(&ptable.lock);
00555 return 23;
00556 }
```

Referenced by main(), and sys_cps().

5.25.2.21 cpuid()

```
int cpuid (
              void  )
```

Definition at line 31 of file proc.c.

```
00031         {
00032   return mycpu()-cpus;
00033 }
```

Referenced by seginit(), and trap().

5.25.2.22 deallocuvm()

```
int deallocuvm (
              pde_t * pgdir,
              uint oldsz,
              uint newsz )
```

Definition at line 256 of file vm.c.

```
00257 {
00258   pte_t *pte;
00259   uint a, pa;
00260
00261   if(newsz >= oldsz)
00262     return oldsz;
00263
00264   a = PGROUNDUP(newsz);
00265   for(; a  < oldsz; a += PGSIZE){
00266     pte = walkpgdir(pgdir, (char*)a, 0);
00267     if(!pte)
00268       a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
00269     else if((*pte & PTE_P) != 0){
00270       pa = PTE_ADDR(*pte);
00271       if(pa == 0)
```

```
00272         panic("kfree");
00273       char *v = P2V(pa);
00274       kfree(v);
00275       *pte = 0;
00276     }
00277   }
00278   return newsz;
00279 }
```

Referenced by allocuvm(), freevm(), and growproc().

5.25.2.23 dirlink()

```
int dirlink (
              struct inode * dp,
              char * name,
              uint inum )
```

Definition at line 552 of file fs.c.

```
00553 {
00554   int off;
00555   struct dirent de;
00556   struct inode *ip;
00557
00558   // Check that name is not present.
00559   if((ip = dirlookup(dp, name, 0)) != 0){
00560     iput(ip);
00561     return -1;
00562   }
00563
00564   // Look for an empty dirent.
00565   for(off = 0; off < dp->size; off += sizeof(de)){
00566     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00567       panic("dirlink read");
00568     if(de.inum == 0)
00569       break;
00570   }
00571
00572   strncpy(de.name, name, DIRSIZ);
00573   de.inum = inum;
00574   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00575     panic("dirlink");
00576
00577   return 0;
00578 }
```

Referenced by create(), and sys_link().

5.25.2.24 dirlookup()

```
struct inode * dirlookup (
              struct inode * dp,
              char * name,
              uint * poff )
```

Definition at line 525 of file fs.c.

```
00526 {
00527   uint off, inum;
00528   struct dirent de;
00529
00530   if(dp->type != T_DIR)
00531     panic("dirlookup not DIR");
00532
00533   for(off = 0; off < dp->size; off += sizeof(de)){
00534     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
```

```
00535       panic("dirlookup read");
00536     if(de.inum == 0)
00537       continue;
00538     if(namecmp(name, de.name) == 0){
00539       // entry matches path element
00540       if(poff)
00541         *poff = off;
00542       inum = de.inum;
00543       return iget(dp->dev, inum);
00544     }
00545   }
00546
00547   return 0;
00548 }
```

Referenced by create(), dirlink(), namex(), and sys_unlink().

5.25.2.25 end_op()

```
void end_op ( )
```

Definition at line 146 of file log.c.

```
00147 {
00148   int do_commit = 0;
00149
00150   acquire(&log.lock);
00151   log.outstanding -= 1;
00152   if(log.committing)
00153     panic("log.committing");
00154   if(log.outstanding == 0){
00155     do_commit = 1;
00156     log.committing = 1;
00157   } else {
00158     // begin_op() may be waiting for log space,
00159     // and decrementing log.outstanding has decreased
00160     // the amount of reserved space.
00161     wakeup(&log);
00162   }
00163   release(&log.lock);
00164
00165   if(do_commit){
00166     // call commit w/o holding locks, since not allowed
00167     // to sleep with locks.
00168     commit();
00169     acquire(&log.lock);
00170     log.committing = 0;
00171     wakeup(&log);
00172     release(&log.lock);
00173   }
00174 }
```

Referenced by exec(), exit(), fileclose(), filewrite(), sys_chdir(), sys_link(), sys_mkdir(), sys_mknod(), sys_open(), and sys_unlink().

5.25.2.26 exec()

```
int exec (
              char * path,
              char ** argv )
```

Definition at line 11 of file exec.c.

```
00012 {
00013   char *s, *last;
00014   int i, off;
00015   uint argc, sz, sp, ustack[3+MAXARG+1];
00016   struct elfhdr elf;
00017   struct inode *ip;
```

```
00018   struct proghdr ph;
00019   pde_t *pgdir, *oldpgdir;
00020   struct proc *curproc = myproc();
00021
00022   begin_op();
00023
00024   if((ip = namei(path)) == 0){
00025     end_op();
00026     cprintf("exec: fail\n");
00027     return -1;
00028   }
00029   ilock(ip);
00030   pgdir = 0;
00031
00032   // Check ELF header
00033   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
00034     goto bad;
00035   if(elf.magic != ELF_MAGIC)
00036     goto bad;
00037
00038   if((pgdir = setupkvm()) == 0)
00039     goto bad;
00040
00041   // Load program into memory.
00042   sz = 0;
00043   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
00044     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
00045       goto bad;
00046     if(ph.type != ELF_PROG_LOAD)
00047       continue;
00048     if(ph.memsz < ph.filesz)
00049       goto bad;
00050     if(ph.vaddr + ph.memsz < ph.vaddr)
00051       goto bad;
00052     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
00053       goto bad;
00054     if(ph.vaddr % PGSIZE != 0)
00055       goto bad;
00056     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
00057       goto bad;
00058   }
00059   iunlockput(ip);
00060   end_op();
00061   ip = 0;
00062
00063   // Allocate two pages at the next page boundary.
00064   // Make the first inaccessible.  Use the second as the user stack.
00065   sz = PGROUNDUP(sz);
00066   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
00067     goto bad;
00068   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
00069   sp = sz;
00070
00071   // Push argument strings, prepare rest of stack in ustack.
00072   for(argc = 0; argv[argc]; argc++) {
00073     if(argc >= MAXARG)
00074       goto bad;
00075     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
00076     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
00077       goto bad;
00078     ustack[3+argc] = sp;
00079   }
00080   ustack[3+argc] = 0;
00081
00082   ustack[0] = 0xffffffff;  // fake return PC
00083   ustack[1] = argc;
00084   ustack[2] = sp - (argc+1)*4;  // argv pointer
00085
00086   sp -= (3+argc+1) * 4;
00087   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
00088     goto bad;
00089
00090   // Save program name for debugging.
00091   for(last=s=path; *s; s++)
00092     if(*s == '/')
00093       last = s+1;
00094   safestrcpy(curproc->name, last, sizeof(curproc->name));
00095
00096   // Commit to the user image.
00097   oldpgdir = curproc->pgdir;
00098   curproc->pgdir = pgdir;
00099   curproc->sz = sz;
00100   curproc->tf->eip = elf.entry;  // main
00101   curproc->tf->esp = sp;
00102   switchuvm(curproc);
00103   freevm(oldpgdir);
00104   return 0;
```

```
00105
00106  bad:
00107    if(pgdir)
00108      freevm(pgdir);
00109    if(ip){
00110      iunlockput(ip);
00111      end_op();
00112    }
00113    return -1;
00114  }
```

Referenced by bigargtest(), exectest(), main(), runcmd(), and sys_exec().

5.25.2.27 exit()

```
void exit (
              void  )
```

Definition at line 228 of file proc.c.

```
00229 {
00230   struct proc *curproc = myproc();
00231   struct proc *p;
00232   int fd;
00233
00234   if(curproc == initproc)
00235     panic("init exiting");
00236
00237   // Close all open files.
00238   for(fd = 0; fd < NOFILE; fd++){
00239     if(curproc->ofile[fd]){
00240       fileclose(curproc->ofile[fd]);
00241       curproc->ofile[fd] = 0;
00242     }
00243   }
00244
00245   begin_op();
00246   iput(curproc->cwd);
00247   end_op();
00248   curproc->cwd = 0;
00249
00250   acquire(&ptable.lock);
00251
00252   // Parent might be sleeping in wait().
00253   wakeup1(curproc->parent);
00254
00255   // Pass abandoned children to init.
00256   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00257     if(p->parent == curproc){
00258       p->parent = initproc;
00259       if(p->state == ZOMBIE)
00260         wakeup1(initproc);
00261     }
00262   }
00263
00264   // Jump into the scheduler, never to return.
00265   curproc->state = ZOMBIE;
00266   sched();
00267   panic("zombie exit");
00268 }
```

Referenced by argptest(), bigargtest(), bigdir(), bigfile(), bigwrite(), bsstest(), cat(), concreate(), createdelete(), dirfile(), dirtest(), exectest(), exitiputtest(), exitwait(), forktest(), fourfiles(), fourteen(), iputtest(), iref(), linktest(), linkunlink(), main(), mem(), openiputtest(), opentest(), panic(), pipe1(), rmdot(), rsect(), runcmd(), sbrktest(), sharedfd(), subdir(), sys_exit(), trap(), uio(), unlinkread(), validatetest(), wc(), writetest(), writetest1(), and wsect().

5.25.2.28 fetchint()

```
int fetchint (
              uint addr,
              int * ip )
```

Definition at line 18 of file syscall.c.

```
00019 {
00020   struct proc *curproc = myproc();
00021
00022   if(addr >= curproc->sz || addr+4 > curproc->sz)
00023     return -1;
00024   *ip = *(int*)(addr);
00025   return 0;
00026 }
```

Referenced by argint(), and sys_exec().

5.25.2.29 fetchstr()

```
int fetchstr (
              uint addr,
              char ** pp )
```

Definition at line 32 of file syscall.c.

```
00033 {
00034   char *s, *ep;
00035   struct proc *curproc = myproc();
00036
00037   if(addr >= curproc->sz)
00038     return -1;
00039   *pp = (char*)addr;
00040   ep = (char*)curproc->sz;
00041   for(s = *pp; s < ep; s++){
00042     if(*s == 0)
00043       return s - *pp;
00044   }
00045   return -1;
00046 }
```

Referenced by argstr(), and sys_exec().

5.25.2.30 filealloc()

```
struct file * filealloc (
              void  )
```

Definition at line 27 of file file.c.

```
00028 {
00029   struct file *f;
00030
00031   acquire(&ftable.lock);
00032   for(f = ftable.file; f < ftable.file + NFILE; f++){
00033     if(f->ref == 0){
00034       f->ref = 1;
00035       release(&ftable.lock);
00036       return f;
00037     }
00038   }
00039   release(&ftable.lock);
00040   return 0;
00041 }
```

Referenced by pipealloc(), and sys_open().

5.25.2.31 fileclose()

```
void fileclose (
            struct file * f )
```

Definition at line 57 of file file.c.

```
00058 {
00059   struct file ff;
00060
00061   acquire(&ftable.lock);
00062   if(f->ref < 1)
00063     panic("fileclose");
00064   if(--f->ref > 0){
00065     release(&ftable.lock);
00066     return;
00067   }
00068   ff = *f;
00069   f->ref = 0;
00070   f->type = FD_NONE;
00071   release(&ftable.lock);
00072
00073   if(ff.type == FD_PIPE)
00074     pipeclose(ff.pipe, ff.writable);
00075   else if(ff.type == FD_INODE){
00076     begin_op();
00077     iput(ff.ip);
00078     end_op();
00079   }
00080 }
```

Referenced by exit(), pipealloc(), sys_close(), sys_open(), and sys_pipe().

5.25.2.32 filedup()

```
struct file * filedup (
            struct file * f )
```

Definition at line 45 of file file.c.

```
00046 {
00047   acquire(&ftable.lock);
00048   if(f->ref < 1)
00049     panic("filedup");
00050   f->ref++;
00051   release(&ftable.lock);
00052   return f;
00053 }
```

Referenced by fork(), and sys_dup().

5.25.2.33 fileinit()

```
void fileinit (
            void  )
```

Definition at line 20 of file file.c.

```
00021 {
00022   initlock(&ftable.lock, "ftable");
00023 }
```

5.25.2.34 fileread()

```
int fileread (
              struct file * f,
              char * addr,
              int n )
```

Definition at line 97 of file file.c.

```
00098 {
00099   int r;
00100
00101   if(f->readable == 0)
00102     return -1;
00103   if(f->type == FD_PIPE)
00104     return piperead(f->pipe, addr, n);
00105   if(f->type == FD_INODE){
00106     ilock(f->ip);
00107     if((r = readi(f->ip, addr, f->off, n)) > 0)
00108       f->off += r;
00109     iunlock(f->ip);
00110     return r;
00111   }
00112   panic("fileread");
00113 }
```

Referenced by sys_read().

5.25.2.35 filestat()

```
int filestat (
              struct file * f,
              struct stat * st )
```

Definition at line 84 of file file.c.

```
00085 {
00086   if(f->type == FD_INODE){
00087     ilock(f->ip);
00088     stati(f->ip, st);
00089     iunlock(f->ip);
00090     return 0;
00091   }
00092   return -1;
00093 }
```

Referenced by sys_fstat().

5.25.2.36 filewrite()

```
int filewrite (
              struct file * f,
              char * addr,
              int n )
```

Definition at line 118 of file file.c.

```
00119 {
00120   int r;
00121
00122   if(f->writable == 0)
00123     return -1;
00124   if(f->type == FD_PIPE)
00125     return pipewrite(f->pipe, addr, n);
00126   if(f->type == FD_INODE){
```

```
00127     // write a few blocks at a time to avoid exceeding
00128     // the maximum log transaction size, including
00129     // i-node, indirect block, allocation blocks,
00130     // and 2 blocks of slop for non-aligned writes.
00131     // this really belongs lower down, since writei()
00132     // might be writing a device like the console.
00133     int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
00134     int i = 0;
00135     while(i < n){
00136       int n1 = n - i;
00137       if(n1 > max)
00138         n1 = max;
00139
00140       begin_op();
00141       ilock(f->ip);
00142       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
00143         f->off += r;
00144       iunlock(f->ip);
00145       end_op();
00146
00147       if(r < 0)
00148         break;
00149       if(r != n1)
00150         panic("short filewrite");
00151       i += r;
00152     }
00153     return i == n ? n : -1;
00154   }
00155   panic("filewrite");
00156 }
```

Referenced by sys_write().

5.25.2.37 fork()

```
int fork (
              void  )
```

Definition at line 181 of file proc.c.

```
00182 {
00183   int i, pid;
00184   struct proc *np;
00185   struct proc *curproc = myproc();
00186
00187   // Allocate process.
00188   if((np = allocproc()) == 0){
00189     return -1;
00190   }
00191
00192   // Copy process state from proc.
00193   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
00194     kfree(np->kstack);
00195     np->kstack = 0;
00196     np->state = UNUSED;
00197     return -1;
00198   }
00199   np->sz = curproc->sz;
00200   np->parent = curproc;
00201   *np->tf = *curproc->tf;
00202
00203   // Clear %eax so that fork returns 0 in the child.
00204   np->tf->eax = 0;
00205
00206   for(i = 0; i < NOFILE; i++)
00207     if(curproc->ofile[i])
00208       np->ofile[i] = filedup(curproc->ofile[i]);
00209   np->cwd = idup(curproc->cwd);
00210
00211   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
00212
00213   pid = np->pid;
00214
00215   acquire(&ptable.lock);
00216
00217   np->state = RUNNABLE;
00218
00219   release(&ptable.lock);
```

```
00220
00221   return pid;
00222 }
```

Referenced by bigargtest(), concreate(), createdelete(), exitiputtest(), exitwait(), fork1(), forktest(), fourfiles(), linkunlink(), main(), mem(), openiputtest(), pipe1(), preempt(), sbrktest(), sharedfd(), sys_fork(), uio(), and validatetest().

5.25.2.38 freevm()

```
void freevm (
              pde_t * pgdir )
```

Definition at line 284 of file vm.c.

```
00285 {
00286   uint i;
00287
00288   if(pgdir == 0)
00289     panic("freevm: no pgdir");
00290   deallocuvm(pgdir, KERNBASE, 0);
00291   for(i = 0; i < NPDENTRIES; i++){
00292     if(pgdir[i] & PTE_P){
00293       char * v = P2V(PTE_ADDR(pgdir[i]));
00294       kfree(v);
00295     }
00296   }
00297   kfree((char*)pgdir);
00298 }
```

Referenced by copyuvm(), exec(), setupkvm(), and wait().

5.25.2.39 getcallerpcs()

```
void getcallerpcs (
              void * ,
              uint *  )
```

Referenced by panic(), and procdump().

5.25.2.40 getprocs()

```
int getprocs (
              int max,
              struct uproc *  )
```

Referenced by main().

5.25.2.41 growproc()

```
int growproc (
              int n )
```

Definition at line 159 of file proc.c.

```
00160 {
00161   uint sz;
00162   struct proc *curproc = myproc();
00163
00164   sz = curproc->sz;
00165   if(n > 0){
00166     if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
00167       return -1;
00168   } else if(n < 0){
00169     if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
00170       return -1;
00171   }
00172   curproc->sz = sz;
00173   switchuvm(curproc);
00174   return 0;
00175 }
```

Referenced by sys_sbrk().

5.25.2.42 holding()

```
int holding (
              struct spinlock * lock )
```

Definition at line 90 of file spinlock.c.

```
00091 {
00092   int r;
00093   pushcli();
00094   r = lock->locked && lock->cpu == mycpu();
00095   popcli();
00096   return r;
00097 }
```

Referenced by acquire(), release(), and sched().

5.25.2.43 holdingsleep()

```
int holdingsleep (
              struct sleeplock * lk )
```

Definition at line 45 of file sleeplock.c.

```
00046 {
00047   int r;
00048
00049   acquire(&lk->lk);
00050   r = lk->locked && (lk->pid == myproc()->pid);
00051   release(&lk->lk);
00052   return r;
00053 }
```

Referenced by brelse(), bwrite(), iderw(), and iunlock().

5.25.2.44 ialloc()

```
struct inode * ialloc (
            uint dev,
            short type )
```

Definition at line 195 of file fs.c.

```
00196 {
00197   int inum;
00198   struct buf *bp;
00199   struct dinode *dip;
00200
00201   for(inum = 1; inum < sb.ninodes; inum++){
00202     bp = bread(dev, IBLOCK(inum, sb));
00203     dip = (struct dinode*)bp->data + inum%IPB;
00204     if(dip->type == 0){  // a free inode
00205       memset(dip, 0, sizeof(*dip));
00206       dip->type = type;
00207       log_write(bp);   // mark it allocated on the disk
00208       brelse(bp);
00209       return iget(dev, inum);
00210     }
00211     brelse(bp);
00212   }
00213   panic("ialloc: no inodes");
00214 }
```

Referenced by create().

5.25.2.45 ideinit()

```
void ideinit (
            void  )
```

Definition at line 51 of file ide.c.

```
00052 {
00053   int i;
00054
00055   initlock(&idelock, "ide");
00056   ioapicenable(IRQ_IDE, ncpu - 1);
00057   idewait(0);
00058
00059   // Check if disk 1 is present
00060   outb(0x1f6, 0xe0 | (1<<4));
00061   for(i=0; i<1000; i++){
00062     if(inb(0x1f7) != 0){
00063       havedisk1 = 1;
00064       break;
00065     }
00066   }
00067
00068   // Switch back to disk 0.
00069   outb(0x1f6, 0xe0 | (0<<4));
00070 }
```

5.25.2.46 ideintr()

```
void ideintr (
            void  )
```

Definition at line 104 of file ide.c.

```
00105 {
00106   struct buf *b;
00107
```

```
00108   // First queued buffer is the active request.
00109   acquire(&idelock);
00110
00111   if((b = idequeue) == 0){
00112     release(&idelock);
00113     return;
00114   }
00115   idequeue = b->qnext;
00116
00117   // Read data if needed.
00118   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
00119     insl(0x1f0, b->data, BSIZE/4);
00120
00121   // Wake process waiting for this buf.
00122   b->flags |= B_VALID;
00123   b->flags &= ~B_DIRTY;
00124   wakeup(b);
00125
00126   // Start disk on next buf in queue.
00127   if(idequeue != 0)
00128     idestart(idequeue);
00129
00130   release(&idelock);
00131 }
```

Referenced by trap().

5.25.2.47 iderw()

```
void iderw (
            struct buf * b )
```

Definition at line 138 of file ide.c.

```
00139 {
00140   struct buf **pp;
00141
00142   if(!holdingsleep(&b->lock))
00143     panic("iderw: buf not locked");
00144   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
00145     panic("iderw: nothing to do");
00146   if(b->dev != 0 && !havedisk1)
00147     panic("iderw: ide disk 1 not present");
00148
00149   acquire(&idelock);  //DOC:acquire-lock
00150
00151   // Append b to idequeue.
00152   b->qnext = 0;
00153   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC:insert-queue
00154     ;
00155   *pp = b;
00156
00157   // Start disk if necessary.
00158   if(idequeue == b)
00159     idestart(b);
00160
00161   // Wait for request to finish.
00162   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
00163     sleep(b, &idelock);
00164   }
00165
00166
00167   release(&idelock);
00168 }
```

Referenced by bread(), and bwrite().

5.25.2.48 idtinit()

```
void idtinit (
              void  )
```

Definition at line 30 of file trap.c.

```
00031 {
00032   lidt(idt, sizeof(idt));
00033 }
```

5.25.2.49 idup()

```
struct inode * idup (
              struct inode * ip )
```

Definition at line 277 of file fs.c.

```
00278 {
00279   acquire(&icache.lock);
00280   ip->ref++;
00281   release(&icache.lock);
00282   return ip;
00283 }
```

Referenced by fork(), and namex().

5.25.2.50 iinit()

```
void iinit (
              int dev )
```

Definition at line 172 of file fs.c.

```
00173 {
00174   int i = 0;
00175
00176   initlock(&icache.lock, "icache");
00177   for(i = 0; i < NINODE; i++) {
00178     initsleeplock(&icache.inode[i].lock, "inode");
00179   }
00180
00181   readsb(dev, &sb);
00182   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\
00183  inodestart %d bmap start %d\n", sb.size, sb.nblocks,
00184           sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
00185           sb.bmapstart);
00186 }
```

Referenced by forkret().

5.25.2.51 ilock()

```
void ilock (
            struct inode * ip )
```

Definition at line 288 of file fs.c.

```
00289 {
00290   struct buf *bp;
00291   struct dinode *dip;
00292
00293   if(ip == 0 || ip->ref < 1)
00294     panic("ilock");
00295
00296   acquiresleep(&ip->lock);
00297
00298   if(ip->valid == 0){
00299     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
00300     dip = (struct dinode*)bp->data + ip->inum%IPB;
00301     ip->type = dip->type;
00302     ip->major = dip->major;
00303     ip->minor = dip->minor;
00304     ip->nlink = dip->nlink;
00305     ip->size = dip->size;
00306     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
00307     brelse(bp);
00308     ip->valid = 1;
00309     if(ip->type == 0)
00310       panic("ilock: no type");
00311   }
00312 }
```

Referenced by consoleread(), consolewrite(), create(), exec(), fileread(), filestat(), filewrite(), namex(), sys_chdir(), sys_link(), sys_open(), and sys_unlink().

5.25.2.52 initlock()

```
void initlock (
            struct spinlock * lk,
            char * name )
```

Definition at line 13 of file spinlock.c.

```
00014 {
00015   lk->name = name;
00016   lk->locked = 0;
00017   lk->cpu = 0;
00018 }
```

Referenced by binit(), consoleinit(), fileinit(), ideinit(), iinit(), initlog(), initsleeplock(), kinit1(), pinit(), pipealloc(), and tvinit().

5.25.2.53 initlog()

```
void initlog (
            int dev )
```

Definition at line 54 of file log.c.

```
00055 {
00056   if (sizeof(struct logheader) >= BSIZE)
00057     panic("initlog: too big logheader");
00058
00059   struct superblock sb;
00060   initlock(&log.lock, "log");
00061   readsb(dev, &sb);
00062   log.start = sb.logstart;
00063   log.size = sb.nlog;
00064   log.dev = dev;
00065   recover_from_log();
00066 }
```

Referenced by forkret().

5.25.2.54 initsleeplock()

```
void initsleeplock (
              struct sleeplock * lk,
              char * name )
```

Definition at line 14 of file sleeplock.c.

```
00015 {
00016   initlock(&lk->lk, "sleep lock");
00017   lk->name = name;
00018   lk->locked = 0;
00019   lk->pid = 0;
00020 }
```

Referenced by binit(), and iinit().

5.25.2.55 inituvm()

```
void inituvm (
              pde_t * pgdir,
              char * init,
              uint sz )
```

Definition at line 183 of file vm.c.

```
00184 {
00185   char *mem;
00186
00187   if(sz >= PGSIZE)
00188     panic("inituvm: more than a page");
00189   mem = kalloc();
00190   memset(mem, 0, PGSIZE);
00191   mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
00192   memmove(mem, init, sz);
00193 }
```

Referenced by userinit().

5.25.2.56 ioapicenable()

```
void ioapicenable (
              int irq,
              int cpu )
```

Definition at line 68 of file ioapic.c.

```
00069 {
00070   // Mark interrupt edge-triggered, active high,
00071   // enabled, and routed to the given cpunum,
00072   // which happens to be that cpu's APIC ID.
00073   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
00074   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
00075 }
```

Referenced by consoleinit(), ideinit(), and uartinit().

5.25.2.57 ioapicinit()

```
void ioapicinit (
            void  )
```

Definition at line 49 of file ioapic.c.

```
00050 {
00051   int i, id, maxintr;
00052
00053   ioapic = (volatile struct ioapic*)IOAPIC;
00054   maxintr = (ioapicread(REG_VER) » 16) & 0xFF;
00055   id = ioapicread(REG_ID) » 24;
00056   if(id != ioapicid)
00057     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
00058
00059   // Mark all interrupts edge-triggered, active high, disabled,
00060   // and not routed to any CPUs.
00061   for(i = 0; i <= maxintr; i++){
00062     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
00063     ioapicwrite(REG_TABLE+2*i+1, 0);
00064   }
00065 }
```

5.25.2.58 iput()

```
void iput (
            struct inode * ip )
```

Definition at line 332 of file fs.c.

```
00333 {
00334   acquiresleep(&ip->lock);
00335   if(ip->valid && ip->nlink == 0){
00336     acquire(&icache.lock);
00337     int r = ip->ref;
00338     release(&icache.lock);
00339     if(r == 1){
00340       // inode has no links and no other references: truncate and free.
00341       itrunc(ip);
00342       ip->type = 0;
00343       iupdate(ip);
00344       ip->valid = 0;
00345     }
00346   }
00347   releasesleep(&ip->lock);
00348
00349   acquire(&icache.lock);
00350   ip->ref--;
00351   release(&icache.lock);
00352 }
```

Referenced by dirlink(), exit(), fileclose(), iunlockput(), namex(), sys_chdir(), and sys_link().

5.25.2.59 iunlock()

```
void iunlock (
            struct inode * ip )
```

Definition at line 316 of file fs.c.

```
00317 {
00318   if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
00319     panic("iunlock");
00320
00321   releasesleep(&ip->lock);
00322 }
```

Referenced by consoleread(), consolewrite(), fileread(), filestat(), filewrite(), iunlockput(), namex(), sys_chdir(), sys_link(), and sys_open().

5.25.2.60 iunlockput()

```
void iunlockput (
            struct inode * ip )
```

Definition at line 356 of file fs.c.

```
00357 {
00358   iunlock(ip);
00359   iput(ip);
00360 }
```

Referenced by create(), exec(), namex(), sys_chdir(), sys_link(), sys_mkdir(), sys_mknod(), sys_open(), and sys_unlink().

5.25.2.61 iupdate()

```
void iupdate (
            struct inode * ip )
```

Definition at line 221 of file fs.c.

```
00222 {
00223   struct buf *bp;
00224   struct dinode *dip;
00225
00226   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
00227   dip = (struct dinode*)bp->data + ip->inum%IPB;
00228   dip->type = ip->type;
00229   dip->major = ip->major;
00230   dip->minor = ip->minor;
00231   dip->nlink = ip->nlink;
00232   dip->size = ip->size;
00233   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
00234   log_write(bp);
00235   brelse(bp);
00236 }
```

Referenced by create(), iput(), itrunc(), sys_link(), sys_unlink(), and writei().

5.25.2.62 kalloc()

```
char * kalloc (
            void  )
```

Definition at line 83 of file kalloc.c.

```
00084 {
00085   struct run *r;
00086
00087   if(kmem.use_lock)
00088     acquire(&kmem.lock);
00089   r = kmem.freelist;
00090   if(r)
00091     kmem.freelist = r->next;
00092   if(kmem.use_lock)
00093     release(&kmem.lock);
00094   return (char*)r;
00095 }
```

Referenced by allocproc(), allocuvm(), copyuvm(), inituvm(), pipealloc(), setupkvm(), startothers(), and walkpgdir().

5.25.2.63 kbdintr()

```
void kbdintr (
              void  )
```

Definition at line 47 of file kbd.c.

```
00048 {
00049   consoleintr(kbdgetc);
00050 }
```

Referenced by trap().

5.25.2.64 kfree()

```
void kfree (
              char * v )
```

Definition at line 60 of file kalloc.c.

```
00061 {
00062   struct run *r;
00063
00064   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
00065     panic("kfree");
00066
00067   // Fill with junk to catch dangling refs.
00068   memset(v, 1, PGSIZE);
00069
00070   if(kmem.use_lock)
00071     acquire(&kmem.lock);
00072   r = (struct run*)v;
00073   r->next = kmem.freelist;
00074   kmem.freelist = r;
00075   if(kmem.use_lock)
00076     release(&kmem.lock);
00077 }
```

Referenced by allocuvm(), copyuvm(), deallocuvm(), fork(), freerange(), freevm(), pipealloc(), pipeclose(), and wait().

5.25.2.65 kill()

```
int kill (
              int pid )
```

Definition at line 480 of file proc.c.

```
00481 {
00482   struct proc *p;
00483
00484   acquire(&ptable.lock);
00485   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00486     if(p->pid == pid){
00487       p->killed = 1;
00488       // Wake process from sleep if necessary.
00489       if(p->state == SLEEPING)
00490         p->state = RUNNABLE;
00491       release(&ptable.lock);
00492       return 0;
00493     }
00494   }
00495   release(&ptable.lock);
00496   return -1;
00497 }
```

Referenced by main(), mem(), preempt(), sbrktest(), sys_kill(), and validatetest().

5.25.2.66 kinit1()

```
void kinit1 (
             void * vstart,
             void * vend )
```

Definition at line 32 of file kalloc.c.

```
00033 {
00034   initlock(&kmem.lock, "kmem");
00035   kmem.use_lock = 0;
00036   freerange(vstart, vend);
00037 }
```

5.25.2.67 kinit2()

```
void kinit2 (
             void * vstart,
             void * vend )
```

Definition at line 40 of file kalloc.c.

```
00041 {
00042   freerange(vstart, vend);
00043   kmem.use_lock = 1;
00044 }
```

5.25.2.68 kvmalloc()

```
void kvmalloc (
             void  )
```

Definition at line 141 of file vm.c.

```
00142 {
00143   kpgdir = setupkvm();
00144   switchkvm();
00145 }
```

5.25.2.69 lapiceoi()

```
void lapiceoi (
             void  )
```

Definition at line 110 of file lapic.c.

```
00111 {
00112   if(lapic)
00113     lapicw(EOI, 0);
00114 }
```

Referenced by trap().

5.25.2.70 lapicid()

```
int lapicid (
             void  )
```

Definition at line 101 of file lapic.c.

```
00102 {
00103   if (!lapic)
00104     return 0;
00105   return lapic[ID] » 24;
00106 }
```

Referenced by mycpu(), and panic().

5.25.2.71 lapicinit()

```
void lapicinit (
             void  )
```

Definition at line 55 of file lapic.c.

```
00056 {
00057   if(!lapic)
00058     return;
00059
00060   // Enable local APIC; set spurious interrupt vector.
00061   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
00062
00063   // The timer repeatedly counts down at bus frequency
00064   // from lapic[TICR] and then issues an interrupt.
00065   // If xv6 cared more about precise timekeeping,
00066   // TICR would be calibrated using an external time source.
00067   lapicw(TDCR, X1);
00068   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
00069   lapicw(TICR, 10000000);
00070
00071   // Disable logical interrupt lines.
00072   lapicw(LINT0, MASKED);
00073   lapicw(LINT1, MASKED);
00074
00075   // Disable performance counter overflow interrupts
00076   // on machines that provide that interrupt entry.
00077   if(((lapic[VER]»16) & 0xFF) >= 4)
00078     lapicw(PCINT, MASKED);
00079
00080   // Map error interrupt to IRQ_ERROR.
00081   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
00082
00083   // Clear error status register (requires back-to-back writes).
00084   lapicw(ESR, 0);
00085   lapicw(ESR, 0);
00086
00087   // Ack any outstanding interrupts.
00088   lapicw(EOI, 0);
00089
00090   // Send an Init Level De-Assert to synchronise arbitration ID's.
00091   lapicw(ICRHI, 0);
00092   lapicw(ICRLO, BCAST | INIT | LEVEL);
00093   while(lapic[ICRLO] & DELIVS)
00094     ;
00095
00096   // Enable interrupts on the APIC (but not on the processor).
00097   lapicw(TPR, 0);
00098 }
```

Referenced by mpenter().

5.25.2.72 lapicstartap()

```
void lapicstartap (
             uchar apicid,
             uint addr )
```

Definition at line 129 of file lapic.c.

```
00130 {
00131   int i;
00132   ushort *wrv;
00133
00134   // "The BSP must initialize CMOS shutdown code to 0AH
00135   // and the warm reset vector (DWORD based at 40:67) to point at
00136   // the AP startup code prior to the [universal startup algorithm]."
00137   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
00138   outb(CMOS_PORT+1, 0x0A);
00139   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
00140   wrv[0] = 0;
00141   wrv[1] = addr >> 4;
00142
00143   // "Universal startup algorithm."
00144   // Send INIT (level-triggered) interrupt to reset other CPU.
00145   lapicw(ICRHI, apicid<<24);
00146   lapicw(ICRLO, INIT | LEVEL | ASSERT);
00147   microdelay(200);
00148   lapicw(ICRLO, INIT | LEVEL);
00149   microdelay(100);    // should be 10ms, but too slow in Bochs!
00150
00151   // Send startup IPI (twice!) to enter code.
00152   // Regular hardware is supposed to only accept a STARTUP
00153   // when it is in the halted state due to an INIT.  So the second
00154   // should be ignored, but it is part of the official Intel algorithm.
00155   // Bochs complains about the second one.  Too bad for Bochs.
00156   for(i = 0; i < 2; i++){
00157     lapicw(ICRHI, apicid<<24);
00158     lapicw(ICRLO, STARTUP | (addr>>12));
00159     microdelay(200);
00160   }
00161 }
```

Referenced by startothers().

5.25.2.73 loaduvm()

```
int loaduvm (
             pde_t * pgdir,
             char * addr,
             struct inode * ip,
             uint offset,
             uint sz )
```

Definition at line 198 of file vm.c.

```
00199 {
00200   uint i, pa, n;
00201   pte_t *pte;
00202
00203   if((uint) addr % PGSIZE != 0)
00204     panic("loaduvm: addr must be page aligned");
00205   for(i = 0; i < sz; i += PGSIZE){
00206     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
00207       panic("loaduvm: address should exist");
00208     pa = PTE_ADDR(*pte);
00209     if(sz - i < PGSIZE)
00210       n = sz - i;
00211     else
00212       n = PGSIZE;
00213     if(readi(ip, P2V(pa), offset+i, n) != n)
00214       return -1;
00215   }
00216   return 0;
00217 }
```

Referenced by exec().

5.25.2.74 log_write()

```
void log_write (
            struct buf * b )
```

Definition at line 214 of file log.c.

```
00215 {
00216   int i;
00217
00218   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
00219     panic("too big a transaction");
00220   if (log.outstanding < 1)
00221     panic("log_write outside of trans");
00222
00223   acquire(&log.lock);
00224   for (i = 0; i < log.lh.n; i++) {
00225     if (log.lh.block[i] == b->blockno)   // log absorbtion
00226       break;
00227   }
00228   log.lh.block[i] = b->blockno;
00229   if (i == log.lh.n)
00230     log.lh.n++;
00231   b->flags |= B_DIRTY; // prevent eviction
00232   release(&log.lock);
00233 }
```

Referenced by balloc(), bfree(), bmap(), bzero(), ialloc(), iupdate(), and writei().

5.25.2.75 memcmp()

```
int memcmp (
            const void * v1,
            const void * v2,
            uint n )
```

Definition at line 16 of file string.c.

```
00017 {
00018   const uchar *s1, *s2;
00019
00020   s1 = v1;
00021   s2 = v2;
00022   while(n-- > 0){
00023     if(*s1 != *s2)
00024       return *s1 - *s2;
00025     s1++, s2++;
00026   }
00027
00028   return 0;
00029 }
```

Referenced by cmostime(), mpconfig(), and mpsearch1().

5.25.2.76 memmove()

```
void * memmove (
            void * dst,
            const void * src,
            uint n )
```

Definition at line 32 of file string.c.

```
00033 {
00034   const char *s;
```

```
00035   char *d;
00036
00037   s = src;
00038   d = dst;
00039   if(s < d && s + n > d){
00040     s += n;
00041     d += n;
00042     while(n-- > 0)
00043       *--d = *--s;
00044   } else
00045     while(n-- > 0)
00046       *d++ = *s++;
00047
00048   return dst;
00049 }
```

Referenced by cgaputc(), copyout(), copyuvm(), fmtname(), grep(), iderw(), ilock(), inituvm(), install_trans(), iupdate(), ls(), main(), memcpy(), readi(), readsb(), skipelem(), startothers(), write_log(), and writei().

5.25.2.77 memset()

```
void * memset (
            void * dst,
            int c,
            uint n )
```

Definition at line 5 of file string.c.

```
00006 {
00007   if ((int)dst%4 == 0 && n%4 == 0){
00008     c &= 0xFF;
00009     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
00010   } else
00011     stosb(dst, c, n);
00012   return dst;
00013 }
```

Referenced by allocproc(), allocuvm(), backcmd(), bigfile(), bzero(), cgaputc(), concreate(), execcmd(), fmtname(), fourfiles(), getcmd(), ialloc(), inituvm(), kfree(), listcmd(), main(), pipecmd(), redircmd(), setupkvm(), sharedfd(), sys_exec(), sys_unlink(), userinit(), and walkpgdir().

5.25.2.78 microdelay()

```
void microdelay (
            int us )
```

Definition at line 119 of file lapic.c.

```
00120 {
00121 }
```

Referenced by cmos_read(), lapicstartap(), and uartputc().

5.25.2.79 mpinit()

```
void mpinit (
            void  )
```

Definition at line 92 of file mp.c.

```
00093 {
00094   uchar *p, *e;
00095   int ismp;
00096   struct mp *mp;
00097   struct mpconf *conf;
00098   struct mpproc *proc;
00099   struct mpioapic *ioapic;
00100
00101   if((conf = mpconfig(&mp)) == 0)
00102     panic("Expect to run on an SMP");
00103   ismp = 1;
00104   lapic = (uint*)conf->lapicaddr;
00105   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
00106     switch(*p){
00107     case MPPROC:
00108       proc = (struct mpproc*)p;
00109       if(ncpu < NCPU) {
00110         cpus[ncpu].apicid = proc->apicid;  // apicid may differ from ncpu
00111         ncpu++;
00112       }
00113       p += sizeof(struct mpproc);
00114       continue;
00115     case MPIOAPIC:
00116       ioapic = (struct mpioapic*)p;
00117       ioapicid = ioapic->apicno;
00118       p += sizeof(struct mpioapic);
00119       continue;
00120     case MPBUS:
00121     case MPIOINTR:
00122     case MPLINTR:
00123       p += 8;
00124       continue;
00125     default:
00126       ismp = 0;
00127       break;
00128     }
00129   }
00130   if(!ismp)
00131     panic("Didn't find a suitable machine");
00132
00133   if(mp->imcrp){
00134     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
00135     // But it would on real hardware.
00136     outb(0x22, 0x70);   // Select IMCR
00137     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
00138   }
00139 }
```

5.25.2.80 mycpu()

```
struct cpu * mycpu (
            void  )
```

Definition at line 38 of file proc.c.

```
00039 {
00040   int apicid, i;
00041
00042   if(readeflags()&FL_IF)
00043     panic("mycpu called with interrupts enabled\n");
00044
00045   apicid = lapicid();
00046   // APIC IDs are not guaranteed to be contiguous. Maybe we should have
00047   // a reverse map, or reserve a register to store &cpus[i].
00048   for (i = 0; i < ncpu; ++i) {
00049     if (cpus[i].apicid == apicid)
00050       return &cpus[i];
00051   }
00052   panic("unknown apicid\n");
00053 }
```

Referenced by acquire(), cpuid(), holding(), myproc(), popcli(), pushcli(), sched(), scheduler(), startothers(), and switchuvm().

5.25.2.81 myproc()

```
struct proc * myproc ( )
```

Definition at line 58 of file proc.c.

```
00058               {
00059   struct cpu *c;
00060   struct proc *p;
00061   pushcli();
00062   c = mycpu();
00063   p = c->proc;
00064   popcli();
00065   return p;
00066 }
```

Referenced by acquiresleep(), argfd(), argint(), argptr(), consoleread(), exec(), exit(), fdalloc(), fetchint(), fetchstr(), fork(), growproc(), holdingsleep(), namex(), piperead(), pipewrite(), sched(), sleep(), sys_chdir(), sys_close(), sys_getpid(), sys_pipe(), sys_sbrk(), sys_sleep(), syscall(), trap(), wait(), and yield().

5.25.2.82 namecmp()

```
int namecmp (
            const char * s,
            const char * t )
```

Definition at line 517 of file fs.c.

```
00518 {
00519   return strncmp(s, t, DIRSIZ);
00520 }
```

Referenced by dirlookup(), and sys_unlink().

5.25.2.83 namei()

```
struct inode * namei (
            char * path )
```

Definition at line 660 of file fs.c.

```
00661 {
00662   char name[DIRSIZ];
00663   return namex(path, 0, name);
00664 }
```

Referenced by exec(), sys_chdir(), sys_link(), sys_open(), and userinit().

5.25.2.84 nameiparent()

```
struct inode * nameiparent (
            char * path,
            char * name )
```

Definition at line 667 of file fs.c.

```
00668 {
00669   return namex(path, 1, name);
00670 }
```

Referenced by create(), namex(), sys_link(), and sys_unlink().

5.25.2.85 panic()

```
void panic (
            char * s )
```

Definition at line 107 of file console.c.

```
00108 {
00109   int i;
00110   uint pcs[10];
00111
00112   cli();
00113   cons.locking = 0;
00114   // use lapiccpunum so that we can call panic from mycpu()
00115   cprintf("lapicid %d: panic: ", lapicid());
00116   cprintf(s);
00117   cprintf("\n");
00118   getcallerpcs(&s, pcs);
00119   for(i=0; i<10; i++)
00120     cprintf(" %p", pcs[i]);
00121   panicked = 1; // freeze other CPU
00122   for(;;)
00123     ;
00124 }
```

Referenced by acquire(), balloc(), bfree(), bget(), bmap(), brelse(), bwrite(), cgaputc(), clearpteu(), copyuvm(), cprintf(), create(), deallocuvm(), dirlink(), dirlookup(), end_op(), exit(), fileclose(), filedup(), fileread(), filewrite(), fork1(), freevm(), ialloc(), iderw(), idestart(), iget(), ilock(), initlog(), inituvm(), isdirempty(), iunlock(), kfree(), loaduvm(), log_write(), mappages(), mpinit(), mycpu(), parseblock(), parsecmd(), parseexec(), parseredirs(), popcli(), release(), runcmd(), sched(), setupkvm(), sleep(), switchuvm(), sys_unlink(), trap(), and userinit().

5.25.2.86 picenable()

```
void picenable (
            int  )
```

5.25.2.87 picinit()

```
void picinit (
            void  )
```

Definition at line 11 of file picirq.c.

```
00012 {
00013   // mask all interrupts
00014   outb(IO_PIC1+1, 0xFF);
00015   outb(IO_PIC2+1, 0xFF);
00016 }
```

5.25.2.88 pinit()

```
void pinit (
            void  )
```

Definition at line 24 of file proc.c.

```
00025 {
00026   initlock(&ptable.lock, "ptable");
00027 }
```

5.25.2.89 pipealloc()

```
int pipealloc (
              struct file ** f0,
              struct file ** f1 )
```

Definition at line 23 of file pipe.c.

```
00024 {
00025   struct pipe *p;
00026
00027   p = 0;
00028   *f0 = *f1 = 0;
00029   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
00030     goto bad;
00031   if((p = (struct pipe*)kalloc()) == 0)
00032     goto bad;
00033   p->readopen = 1;
00034   p->writeopen = 1;
00035   p->nwrite = 0;
00036   p->nread = 0;
00037   initlock(&p->lock, "pipe");
00038   (*f0)->type = FD_PIPE;
00039   (*f0)->readable = 1;
00040   (*f0)->writable = 0;
00041   (*f0)->pipe = p;
00042   (*f1)->type = FD_PIPE;
00043   (*f1)->readable = 0;
00044   (*f1)->writable = 1;
00045   (*f1)->pipe = p;
00046   return 0;
00047
00048 //PAGEBREAK: 20
00049  bad:
00050   if(p)
00051     kfree((char*)p);
00052   if(*f0)
00053     fileclose(*f0);
00054   if(*f1)
00055     fileclose(*f1);
00056   return -1;
00057 }
```

Referenced by sys_pipe().

5.25.2.90 pipeclose()

```
void pipeclose (
              struct pipe * p,
              int writable )
```

Definition at line 60 of file pipe.c.

```
00061 {
00062   acquire(&p->lock);
00063   if(writable){
00064     p->writeopen = 0;
00065     wakeup(&p->nread);
00066   } else {
00067     p->readopen = 0;
00068     wakeup(&p->nwrite);
00069   }
00070   if(p->readopen == 0 && p->writeopen == 0){
00071     release(&p->lock);
00072     kfree((char*)p);
00073   } else
00074     release(&p->lock);
00075 }
```

Referenced by fileclose().

5.25.2.91 piperead()

```
int piperead (
             struct pipe * p,
             char * addr,
             int n )
```

Definition at line 101 of file pipe.c.

```
00102 {
00103   int i;
00104
00105   acquire(&p->lock);
00106   while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
00107     if(myproc()->killed){
00108       release(&p->lock);
00109       return -1;
00110     }
00111     sleep(&p->nread, &p->lock); //DOC: piperead-sleep
00112   }
00113   for(i = 0; i < n; i++){  //DOC: piperead-copy
00114     if(p->nread == p->nwrite)
00115       break;
00116     addr[i] = p->data[p->nread++ % PIPESIZE];
00117   }
00118   wakeup(&p->nwrite);  //DOC: piperead-wakeup
00119   release(&p->lock);
00120   return i;
00121 }
```

Referenced by fileread().

5.25.2.92 pipewrite()

```
int pipewrite (
             struct pipe * p,
             char * addr,
             int n )
```

Definition at line 79 of file pipe.c.

```
00080 {
00081   int i;
00082
00083   acquire(&p->lock);
00084   for(i = 0; i < n; i++){
00085     while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
00086       if(p->readopen == 0 || myproc()->killed){
00087         release(&p->lock);
00088         return -1;
00089       }
00090       wakeup(&p->nread);
00091       sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
00092     }
00093     p->data[p->nwrite++ % PIPESIZE] = addr[i];
00094   }
00095   wakeup(&p->nread);  //DOC: pipewrite-wakeup1
00096   release(&p->lock);
00097   return n;
00098 }
```

Referenced by filewrite().

5.25.2.93 popcli()

```
void popcli (
             void  )
```

Definition at line 117 of file spinlock.c.

```
00118 {
00119   if(readeflags()&FL_IF)
00120     panic("popcli - interruptible");
00121   if(--mycpu()->ncli < 0)
00122     panic("popcli");
00123   if(mycpu()->ncli == 0 && mycpu()->intena)
00124     sti();
00125 }
```

Referenced by holding(), myproc(), release(), and switchuvm().

5.25.2.94 procdump()

```
void procdump (
             void  )
```

Definition at line 504 of file proc.c.

```
00505 {
00506   static char *states[] = {
00507   [UNUSED]    "unused",
00508   [EMBRYO]    "embryo",
00509   [SLEEPING]  "sleep ",
00510   [RUNNABLE]  "runble",
00511   [RUNNING]   "run   ",
00512   [ZOMBIE]    "zombie"
00513   };
00514   int i;
00515   struct proc *p;
00516   char *state;
00517   uint pc[10];
00518
00519   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00520     if(p->state == UNUSED)
00521       continue;
00522     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
00523       state = states[p->state];
00524     else
00525       state = "???";
00526     cprintf("%d %s %s", p->pid, state, p->name);
00527     if(p->state == SLEEPING){
00528       getcallerpcs((uint*)p->context->ebp+2, pc);
00529       for(i=0; i<10 && pc[i] != 0; i++)
00530         cprintf(" %p", pc[i]);
00531     }
00532     cprintf("\n");
00533   }
00534 }
```

Referenced by consoleintr().

5.25.2.95 pushcli()

```
void pushcli (
             void  )
```

Definition at line 105 of file spinlock.c.

```
00106 {
00107   int eflags;
00108
00109   eflags = readeflags();
00110   cli();
00111   if(mycpu()->ncli == 0)
00112     mycpu()->intena = eflags & FL_IF;
00113   mycpu()->ncli += 1;
00114 }
```

Referenced by acquire(), holding(), myproc(), and switchuvm().

5.25.2.96 readi()

```
int readi (
              struct inode * ip,
              char * dst,
              uint off,
              uint n )
```

Definition at line 453 of file fs.c.

```
00454 {
00455   uint tot, m;
00456   struct buf *bp;
00457
00458   if(ip->type == T_DEV){
00459     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
00460       return -1;
00461     return devsw[ip->major].read(ip, dst, n);
00462   }
00463
00464   if(off > ip->size || off + n < off)
00465     return -1;
00466   if(off + n > ip->size)
00467     n = ip->size - off;
00468
00469   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
00470     bp = bread(ip->dev, bmap(ip, off/BSIZE));
00471     m = min(n - tot, BSIZE - off%BSIZE);
00472     memmove(dst, bp->data + off%BSIZE, m);
00473     brelse(bp);
00474   }
00475   return n;
00476 }
```

Referenced by dirlink(), dirlookup(), exec(), fileread(), isdirempty(), and loaduvm().

5.25.2.97 readsb()

```
void readsb (
              int dev,
              struct superblock * sb )
```

Definition at line 32 of file fs.c.

```
00033 {
00034   struct buf *bp;
00035
00036   bp = bread(dev, 1);
00037   memmove(sb, bp->data, sizeof(*sb));
00038   brelse(bp);
00039 }
```

Referenced by iinit(), and initlog().

5.25.2.98 release()

```
void release (
              struct spinlock * lk )
```

Definition at line 47 of file spinlock.c.

```
00048 {
00049   if(!holding(lk))
00050     panic("release");
00051
```

```
00052   lk->pcs[0] = 0;
00053   lk->cpu = 0;
00054
00055   // Tell the C compiler and the processor to not move loads or stores
00056   // past this point, to ensure that all the stores in the critical
00057   // section are visible to other cores before the lock is released.
00058   // Both the C compiler and the hardware may re-order loads and
00059   // stores; __sync_synchronize() tells them both not to.
00060   __sync_synchronize();
00061
00062   // Release the lock, equivalent to lk->locked = 0.
00063   // This code can't use a C assignment, since it might
00064   // not be atomic. A real OS would use C atomics here.
00065   asm volatile("movl $0, %0" : "+m" (lk->locked) : );
00066
00067   popcli();
00068 }
```

Referenced by acquiresleep(), allocproc(), begin_op(), bget(), brelse(), chpr(), consoleintr(), consoleread(), consolewrite(), cprintf(), cps(), end_op(), filealloc(), fileclose(), filedup(), fork(), forkret(), holdingsleep(), ideintr(), iderw(), idup(), iget(), iput(), kalloc(), kfree(), kill(), log_write(), pipeclose(), piperead(), pipewrite(), releasesleep(), scheduler(), sleep(), sys_sleep(), sys_uptime(), trap(), userinit(), wait(), wakeup(), and yield().

5.25.2.99 releasesleep()

```
void releasesleep (
              struct sleeplock * lk )
```

Definition at line 35 of file sleeplock.c.

```
00036 {
00037   acquire(&lk->lk);
00038   lk->locked = 0;
00039   lk->pid = 0;
00040   wakeup(lk);
00041   release(&lk->lk);
00042 }
```

Referenced by brelse(), iput(), and iunlock().

5.25.2.100 safestrcpy()

```
char * safestrcpy (
              char * s,
              const char * t,
              int n )
```

Definition at line 83 of file string.c.

```
00084 {
00085   char *os;
00086
00087   os = s;
00088   if(n <= 0)
00089     return os;
00090   while(--n > 0 && (*s++ = *t++) != 0)
00091     ;
00092   *s = 0;
00093   return os;
00094 }
```

Referenced by exec(), fork(), and userinit().

5.25.2.101 sched()

```
void sched (
              void  )
```

Definition at line 366 of file proc.c.

```
00367 {
00368   int intena;
00369   struct proc *p = myproc();
00370
00371   if(!holding(&ptable.lock))
00372     panic("sched ptable.lock");
00373   if(mycpu()->ncli != 1)
00374     panic("sched locks");
00375   if(p->state == RUNNING)
00376     panic("sched running");
00377   if(readeflags()&FL_IF)
00378     panic("sched interruptible");
00379   intena = mycpu()->intena;
00380   swtch(&p->context, mycpu()->scheduler);
00381   mycpu()->intena = intena;
00382 }
```

Referenced by exit(), sleep(), and yield().

5.25.2.102 scheduler()

```
void scheduler (
              void  )
```

Definition at line 323 of file proc.c.

```
00324 {
00325   struct proc *p;
00326   struct cpu *c = mycpu();
00327   c->proc = 0;
00328
00329   for(;;){
00330     // Enable interrupts on this processor.
00331     sti();
00332
00333     // Loop over process table looking for process to run.
00334     acquire(&ptable.lock);
00335     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00336       if(p->state != RUNNABLE)
00337         continue;
00338
00339       // Switch to chosen process.  It is the process's job
00340       // to release ptable.lock and then reacquire it
00341       // before jumping back to us.
00342       c->proc = p;
00343       switchuvm(p);
00344       p->state = RUNNING;
00345
00346       swtch(&(c->scheduler), p->context);
00347       switchkvm();
00348
00349       // Process is done running for now.
00350       // It should have changed its p->state before coming back.
00351       c->proc = 0;
00352     }
00353     release(&ptable.lock);
00354
00355   }
00356 }
```

Referenced by sched().

5.25.2.103 seginit()

```
void seginit (
              void  )
```

Definition at line 16 of file vm.c.

```
00017 {
00018   struct cpu *c;
00019
00020   // Map "logical" addresses to virtual addresses using identity map.
00021   // Cannot share a CODE descriptor for both kernel and user
00022   // because it would have to have DPL_USR, but the CPU forbids
00023   // an interrupt from CPL=0 to DPL=3.
00024   c = &cpus[cpuid()];
00025   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
00026   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
00027   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
00028   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
00029   lgdt(c->gdt, sizeof(c->gdt));
00030 }
```

Referenced by mpenter().

5.25.2.104 setproc()

```
void setproc (
              struct proc *  )
```

5.25.2.105 setupkvm()

```
pde_t * setupkvm (
              void  )
```

Definition at line 119 of file vm.c.

```
00120 {
00121   pde_t *pgdir;
00122   struct kmap *k;
00123
00124   if((pgdir = (pde_t*)kalloc()) == 0)
00125     return 0;
00126   memset(pgdir, 0, PGSIZE);
00127   if (P2V(PHYSTOP) > (void*)DEVSPACE)
00128     panic("PHYSTOP too high");
00129   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
00130     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
00131                 (uint)k->phys_start, k->perm) < 0) {
00132       freevm(pgdir);
00133       return 0;
00134     }
00135   return pgdir;
00136 }
```

Referenced by copyuvm(), exec(), kvmalloc(), and userinit().

5.25.2.106 sleep()

```
void sleep (
              void * chan,
              struct spinlock * lk )
```

Definition at line 418 of file proc.c.

```
00419 {
00420   struct proc *p = myproc();
00421
00422   if(p == 0)
00423     panic("sleep");
00424
00425   if(lk == 0)
00426     panic("sleep without lk");
00427
00428   // Must acquire ptable.lock in order to
00429   // change p->state and then call sched.
00430   // Once we hold ptable.lock, we can be
00431   // guaranteed that we won't miss any wakeup
00432   // (wakeup runs with ptable.lock locked),
00433   // so it's okay to release lk.
00434   if(lk != &ptable.lock){  //DOC: sleeplock0
00435     acquire(&ptable.lock);  //DOC: sleeplock1
00436     release(lk);
00437   }
00438   // Go to sleep.
00439   p->chan = chan;
00440   p->state = SLEEPING;
00441
00442   sched();
00443
00444   // Tidy up.
00445   p->chan = 0;
00446
00447   // Reacquire original lock.
00448   if(lk != &ptable.lock){  //DOC: sleeplock2
00449     release(&ptable.lock);
00450     acquire(lk);
00451   }
00452 }
```

Referenced by acquiresleep(), begin_op(), consoleread(), iderw(), main(), openiputtest(), piperead(), pipewrite(), sbrktest(), sys_sleep(), validatetest(), and wait().

5.25.2.107 stati()

```
void stati (
              struct inode * ip,
              struct stat * st )
```

Definition at line 440 of file fs.c.

```
00441 {
00442   st->dev = ip->dev;
00443   st->ino = ip->inum;
00444   st->type = ip->type;
00445   st->nlink = ip->nlink;
00446   st->size = ip->size;
00447 }
```

Referenced by filestat().

5.25.2.108 strlen()

```
int strlen (
            const char * s )
```

Definition at line 97 of file string.c.

```
00098 {
00099   int n;
00100
00101   for(n = 0; s[n]; n++)
00102     ;
00103   return n;
00104 }
```

Referenced by exec(), fmtname(), ls(), main(), parsecmd(), and printf().

5.25.2.109 strncmp()

```
int strncmp (
            const char * p,
            const char * q,
            uint n )
```

Definition at line 59 of file string.c.

```
00060 {
00061   while(n > 0 && *p && *p == *q)
00062     n--, p++, q++;
00063   if(n == 0)
00064     return 0;
00065   return (uchar)*p - (uchar)*q;
00066 }
```

Referenced by namecmp().

5.25.2.110 strncpy()

```
char * strncpy (
            char * s,
            const char * t,
            int n )
```

Definition at line 69 of file string.c.

```
00070 {
00071   char *os;
00072
00073   os = s;
00074   while(n-- > 0 && (*s++ = *t++) != 0)
00075     ;
00076   while(n-- > 0)
00077     *s++ = 0;
00078   return os;
00079 }
```

Referenced by dirlink(), main(), and sys_getprocs().

5.25.2.111 switchkvm()

```
void switchkvm (
            void  )
```

Definition at line 150 of file vm.c.

```
00151 {
00152   lcr3(V2P(kpgdir));   // switch to the kernel page table
00153 }
```

Referenced by kvmalloc(), mpenter(), and scheduler().

5.25.2.112 switchuvm()

```
void switchuvm (
            struct proc * p )
```

Definition at line 157 of file vm.c.

```
00158 {
00159   if(p == 0)
00160     panic("switchuvm: no process");
00161   if(p->kstack == 0)
00162     panic("switchuvm: no kstack");
00163   if(p->pgdir == 0)
00164     panic("switchuvm: no pgdir");
00165
00166   pushcli();
00167   mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
00168                                 sizeof(mycpu()->ts)-1, 0);
00169   mycpu()->gdt[SEG_TSS].s = 0;
00170   mycpu()->ts.ss0 = SEG_KDATA « 3;
00171   mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
00172   // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
00173   // forbids I/O instructions (e.g., inb and outb) from user space
00174   mycpu()->ts.iomb = (ushort) 0xFFFF;
00175   ltr(SEG_TSS « 3);
00176   lcr3(V2P(p->pgdir));  // switch to process's address space
00177   popcli();
00178 }
```

Referenced by exec(), growproc(), and scheduler().

5.25.2.113 swtch()

```
void swtch (
            struct context ** ,
            struct context *  )
```

Referenced by sched(), and scheduler().

5.25.2.114 syscall()

```
void syscall (
              void  )
```

Definition at line 140 of file syscall.c.

```
00141 {
00142   int num;
00143   struct proc *curproc = myproc();
00144
00145   num = curproc->tf->eax;
00146   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
00147     curproc->tf->eax = syscalls[num]();
00148   } else {
00149     cprintf("%d %s: unknown sys call %d\n",
00150             curproc->pid, curproc->name, num);
00151     curproc->tf->eax = -1;
00152   }
00153 }
```

Referenced by trap().

5.25.2.115 timerinit()

```
void timerinit (
              void  )
```

5.25.2.116 tvinit()

```
void tvinit (
              void  )
```

Definition at line 18 of file trap.c.

```
00019 {
00020   int i;
00021
00022   for(i = 0; i < 256; i++)
00023     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
00024   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
00025
00026   initlock(&tickslock, "time");
00027 }
```

5.25.2.117 uartinit()

```
void uartinit (
              void  )
```

Definition at line 20 of file uart.c.

```
00021 {
00022   char *p;
00023
00024   // Turn off the FIFO
00025   outb(COM1+2, 0);
00026
00027   // 9600 baud, 8 data bits, 1 stop bit, parity off.
00028   outb(COM1+3, 0x80);    // Unlock divisor
```

```
00029   outb(COM1+0, 115200/9600);
00030   outb(COM1+1, 0);
00031   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
00032   outb(COM1+4, 0);
00033   outb(COM1+1, 0x01);    // Enable receive interrupts.
00034
00035   // If status is 0xFF, no serial port.
00036   if(inb(COM1+5) == 0xFF)
00037     return;
00038   uart = 1;
00039
00040   // Acknowledge pre-existing interrupt conditions;
00041   // enable interrupts.
00042   inb(COM1+2);
00043   inb(COM1+0);
00044   ioapicenable(IRQ_COM1, 0);
00045
00046   // Announce that we're here.
00047   for(p="xv6...\n"; *p; p++)
00048     uartputc(*p);
00049 }
```

5.25.2.118 uartintr()

```
void uartintr (
              void  )
```

Definition at line 74 of file uart.c.

```
00075 {
00076   consoleintr(uartgetc);
00077 }
```

Referenced by trap().

5.25.2.119 uartputc()

```
void uartputc (
              int c )
```

Definition at line 52 of file uart.c.

```
00053 {
00054   int i;
00055
00056   if(!uart)
00057     return;
00058   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
00059     microdelay(10);
00060   outb(COM1+0, c);
00061 }
```

Referenced by consputc(), and uartinit().

5.25.2.120 userinit()

```
void userinit (
              void  )
```

Definition at line 121 of file proc.c.

```
00122 {
00123   struct proc *p;
00124   extern char _binary_initcode_start[], _binary_initcode_size[];
00125
00126   p = allocproc();
00127
00128   initproc = p;
00129   if((p->pgdir = setupkvm()) == 0)
00130     panic("userinit: out of memory?");
00131   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
00132   p->sz = PGSIZE;
00133   memset(p->tf, 0, sizeof(*p->tf));
00134   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
00135   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
00136   p->tf->es = p->tf->ds;
00137   p->tf->ss = p->tf->ds;
00138   p->tf->eflags = FL_IF;
00139   p->tf->esp = PGSIZE;
00140   p->tf->eip = 0;  // beginning of initcode.S
00141
00142   safestrcpy(p->name, "initcode", sizeof(p->name));
00143   p->cwd = namei("/");
00144
00145   // this assignment to p->state lets other cores
00146   // run this process. the acquire forces the above
00147   // writes to be visible, and the lock is also needed
00148   // because the assignment might not be atomic.
00149   acquire(&ptable.lock);
00150
00151   p->state = RUNNABLE;
00152
00153   release(&ptable.lock);
00154 }
```

5.25.2.121 uva2ka()

```
char * uva2ka (
              pde_t * pgdir,
              char * uva )
```

Definition at line 350 of file vm.c.

```
00351 {
00352   pte_t *pte;
00353
00354   pte = walkpgdir(pgdir, uva, 0);
00355   if((*pte & PTE_P) == 0)
00356     return 0;
00357   if((*pte & PTE_U) == 0)
00358     return 0;
00359   return (char*)P2V(PTE_ADDR(*pte));
00360 }
```

Referenced by copyout().

5.25.2.122 wait()

```
int wait (
              void  )
```

Definition at line 273 of file proc.c.

```
00274 {
00275   struct proc *p;
00276   int havekids, pid;
00277   struct proc *curproc = myproc();
00278
00279   acquire(&ptable.lock);
00280   for(;;){
00281     // Scan through table looking for exited children.
00282     havekids = 0;
00283     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00284       if(p->parent != curproc)
00285         continue;
00286       havekids = 1;
00287       if(p->state == ZOMBIE){
00288         // Found one.
00289         pid = p->pid;
00290         kfree(p->kstack);
00291         p->kstack = 0;
00292         freevm(p->pgdir);
00293         p->pid = 0;
00294         p->parent = 0;
00295         p->name[0] = 0;
00296         p->killed = 0;
00297         p->state = UNUSED;
00298         release(&ptable.lock);
00299         return pid;
00300       }
00301     }
00302
00303     // No point waiting if we don't have any children.
00304     if(!havekids || curproc->killed){
00305       release(&ptable.lock);
00306       return -1;
00307     }
00308
00309     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
00310     sleep(curproc, &ptable.lock);  //DOC: wait-sleep
00311   }
00312 }
```

Referenced by bigargtest(), concreate(), createdelete(), exitiputtest(), exitwait(), forktest(), fourfiles(), linkunlink(), main(), mem(), openiputtest(), pipe1(), preempt(), runcmd(), sbrktest(), sharedfd(), sys_wait(), uio(), and validatetest().

5.25.2.123 wakeup()

```
void wakeup (
              void * chan )
```

Definition at line 469 of file proc.c.

```
00470 {
00471   acquire(&ptable.lock);
00472   wakeup1(chan);
00473   release(&ptable.lock);
00474 }
```

Referenced by consoleintr(), end_op(), ideintr(), pipeclose(), piperead(), pipewrite(), releasesleep(), and trap().

5.25.2.124 writei()

```
int writei (
              struct inode * ip,
              char * src,
              uint off,
              uint n )
```

Definition at line 482 of file fs.c.

```
00483 {
00484   uint tot, m;
00485   struct buf *bp;
00486
00487   if(ip->type == T_DEV){
00488     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
00489       return -1;
00490     return devsw[ip->major].write(ip, src, n);
00491   }
00492
00493   if(off > ip->size || off + n < off)
00494     return -1;
00495   if(off + n > MAXFILE*BSIZE)
00496     return -1;
00497
00498   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
00499     bp = bread(ip->dev, bmap(ip, off/BSIZE));
00500     m = min(n - tot, BSIZE - off%BSIZE);
00501     memmove(bp->data + off%BSIZE, src, m);
00502     log_write(bp);
00503     brelse(bp);
00504   }
00505
00506   if(n > 0 && off > ip->size){
00507     ip->size = off;
00508     iupdate(ip);
00509   }
00510   return n;
00511 }
```

Referenced by dirlink(), filewrite(), and sys_unlink().

5.25.2.125 yield()

```
void yield (
              void  )
```

Definition at line 386 of file proc.c.

```
00387 {
00388   acquire(&ptable.lock);  //DOC: yieldlock
00389   myproc()->state = RUNNABLE;
00390   sched();
00391   release(&ptable.lock);
00392 }
```

Referenced by trap().

5.25.3 Variable Documentation

5.25.3.1 ioapicid

```
uchar ioapicid  [extern]
```

Definition at line 16 of file mp.c.

Referenced by ioapicinit(), and mpinit().

5.25.3.2 ismp

```
int ismp  [extern]
```

Referenced by mpinit().

5.25.3.3 lapic

```
volatile uint* lapic  [extern]
```

Definition at line 44 of file lapic.c.

Referenced by lapiceoi(), lapicid(), lapicinit(), lapicw(), and mpinit().

5.25.3.4 ticks

```
uint ticks  [extern]
```

Definition at line 15 of file trap.c.

Referenced by sys_sleep(), sys_uptime(), and trap().

5.25.3.5 tickslock

```
struct spinlock tickslock  [extern]
```

Definition at line 14 of file trap.c.

Referenced by sys_sleep(), sys_uptime(), trap(), and tvinit().

5.26 defs.h

Go to the documentation of this file.

```
00001 struct buf;
00002 struct context;
00003 struct file;
00004 struct inode;
00005 struct pipe;
00006 struct proc;
00007 struct rtcdate;
00008 struct spinlock;
00009 struct sleeplock;
00010 struct stat;
00011 struct superblock;
00012 struct uproc;
00013
00014 // bio.c
00015 void            binit(void);
00016 struct buf*     bread(uint, uint);
00017 void            brelse(struct buf*);
00018 void            bwrite(struct buf*);
00019
00020 // console.c
00021 void            consoleinit(void);
00022 void            cprintf(char*, ...);
00023 void            consoleintr(int(*)(void));
00024 void            panic(char*) __attribute__((noreturn));
00025
00026 // exec.c
00027 int             exec(char*, char**);
00028
00029 // file.c
00030 struct file*    filealloc(void);
00031 void            fileclose(struct file*);
00032 struct file*    filedup(struct file*);
00033 void            fileinit(void);
00034 int             fileread(struct file*, char*, int n);
00035 int             filestat(struct file*, struct stat*);
00036 int             filewrite(struct file*, char*, int n);
00037
00038 // fs.c
00039 void            readsb(int dev, struct superblock *sb);
00040 int             dirlink(struct inode*, char*, uint);
00041 struct inode*   dirlookup(struct inode*, char*, uint*);
00042 struct inode*   ialloc(uint, short);
00043 struct inode*   idup(struct inode*);
00044 void            iinit(int dev);
00045 void            ilock(struct inode*);
00046 void            iput(struct inode*);
00047 void            iunlock(struct inode*);
00048 void            iunlockput(struct inode*);
00049 void            iupdate(struct inode*);
00050 int             namecmp(const char*, const char*);
00051 struct inode*   namei(char*);
00052 struct inode*   nameiparent(char*, char*);
00053 int             readi(struct inode*, char*, uint, uint);
00054 void            stati(struct inode*, struct stat*);
00055 int             writei(struct inode*, char*, uint, uint);
00056
00057 // ide.c
00058 void            ideinit(void);
00059 void            ideintr(void);
00060 void            iderw(struct buf*);
00061
00062 // ioapic.c
00063 void            ioapicenable(int irq, int cpu);
00064 extern uchar    ioapicid;
00065 void            ioapicinit(void);
00066
00067 // kalloc.c
00068 char*           kalloc(void);
00069 void            kfree(char*);
00070 void            kinit1(void*, void*);
00071 void            kinit2(void*, void*);
00072
00073 // kbd.c
00074 void            kbdintr(void);
00075
00076 // lapic.c
00077 void            cmostime(struct rtcdate *r);
00078 int             lapicid(void);
00079 extern volatile uint*    lapic;
00080 void            lapiceoi(void);
00081 void            lapicinit(void);
00082 void            lapicstartap(uchar, uint);
```

```
00083 void            microdelay(int);
00084
00085 // log.c
00086 void            initlog(int dev);
00087 void            log_write(struct buf*);
00088 void            begin_op();
00089 void            end_op();
00090
00091 // mp.c
00092 extern int      ismp;
00093 void            mpinit(void);
00094
00095 // picirq.c
00096 void            picenable(int);
00097 void            picinit(void);
00098
00099 // pipe.c
00100 int             pipealloc(struct file**, struct file**);
00101 void            pipeclose(struct pipe*, int);
00102 int             piperead(struct pipe*, char*, int);
00103 int             pipewrite(struct pipe*, char*, int);
00104
00105 //PAGEBREAK: 16
00106 // proc.c
00107 int             cpuid(void);
00108 void            exit(void);
00109 int             fork(void);
00110 int             growproc(int);
00111 int             kill(int);
00112 struct cpu*     mycpu(void);
00113 struct proc*    myproc();
00114 void            pinit(void);
00115 void            procdump(void);
00116 void            scheduler(void) __attribute__((noreturn));
00117 void            sched(void);
00118 void            setproc(struct proc*);
00119 void            sleep(void*, struct spinlock*);
00120 void            userinit(void);
00121 int             wait(void);
00122 void            wakeup(void*);
00123 void            yield(void);
00124 int             chpr(int pid, int priority);
00125 int             cps(void);
00126 int             getprocs(int max, struct uproc*);
00127 //int               halt(void);
00128
00129 // swtch.S
00130 void            swtch(struct context**, struct context*);
00131
00132 // spinlock.c
00133 void            acquire(struct spinlock*);
00134 void            getcallerpcs(void*, uint*);
00135 int             holding(struct spinlock*);
00136 void            initlock(struct spinlock*, char*);
00137 void            release(struct spinlock*);
00138 void            pushcli(void);
00139 void            popcli(void);
00140
00141 // sleeplock.c
00142 void            acquiresleep(struct sleeplock*);
00143 void            releasesleep(struct sleeplock*);
00144 int             holdingsleep(struct sleeplock*);
00145 void            initsleeplock(struct sleeplock*, char*);
00146
00147 // string.c
00148 int             memcmp(const void*, const void*, uint);
00149 void*           memmove(void*, const void*, uint);
00150 void*           memset(void*, int, uint);
00151 char*           safestrcpy(char*, const char*, int);
00152 int             strlen(const char*);
00153 int             strncmp(const char*, const char*, uint);
00154 char*           strncpy(char*, const char*, int);
00155
00156 // syscall.c
00157 int             argint(int, int*);
00158 int             argptr(int, char**, int);
00159 int             argstr(int, char**);
00160 int             fetchint(uint, int*);
00161 int             fetchstr(uint, char**);
00162 void            syscall(void);
00163
00164 // timer.c
00165 void            timerinit(void);
00166
00167 // trap.c
00168 void            idtinit(void);
00169 extern uint     ticks;
```

```
00170 void            tvinit(void);
00171 extern struct spinlock tickslock;
00172
00173 // uart.c
00174 void            uartinit(void);
00175 void            uartintr(void);
00176 void            uartputc(int);
00177
00178 // vm.c
00179 void            seginit(void);
00180 void            kvmalloc(void);
00181 pde_t*          setupkvm(void);
00182 char*           uva2ka(pde_t*, char*);
00183 int             allocuvm(pde_t*, uint, uint);
00184 int             deallocuvm(pde_t*, uint, uint);
00185 void            freevm(pde_t*);
00186 void            inituvm(pde_t*, char*, uint);
00187 int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
00188 pde_t*          copyuvm(pde_t*, uint);
00189 void            switchuvm(struct proc*);
00190 void            switchkvm(void);
00191 int             copyout(pde_t*, uint, void*, uint);
00192 void            clearpteu(pde_t *pgdir, char *uva);
00193
00194 // number of elements in fixed-size array
00195 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

5.27 echo.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- int main (int argc, char *argv[ ])

5.27.1 Function Documentation

5.27.1.1 main()

```
int main (
            int argc,
            char * argv[ ] )
```

Definition at line 6 of file echo.c.

```
00007 {
00008   int i;
00009
00010   for(i = 1; i < argc; i++)
00011     printf(1, "%s%s", argv[i], i+1 < argc ? " " : "\n");
00012   exit();
00013 }
```

5.28 echo.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004
00005 int
00006 main(int argc, char *argv[])
00007 {
00008   int i;
00009
00010   for(i = 1; i < argc; i++)
00011     printf(1, "%s%s", argv[i], i+1 < argc ? " " : "\n");
00012   exit();
00013 }
```

5.29 echo.d File Reference

5.30 echo.d

Go to the documentation of this file.

```
00001 echo.o: echo.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.31 elf.h File Reference

Classes

- struct elfhdr
- struct proghdr

Macros

- #define ELF_MAGIC 0x464C457FU
- #define ELF_PROG_FLAG_EXEC 1
- #define ELF_PROG_FLAG_READ 4
- #define ELF_PROG_FLAG_WRITE 2
- #define ELF_PROG_LOAD 1

5.31.1 Macro Definition Documentation

5.31.1.1 ELF_MAGIC

```
#define ELF_MAGIC 0x464C457FU
```

Definition at line 3 of file elf.h.

5.31.1.2 ELF_PROG_FLAG_EXEC

```
#define ELF_PROG_FLAG_EXEC 1
```

Definition at line 40 of file elf.h.

5.31.1.3 ELF_PROG_FLAG_READ

```
#define ELF_PROG_FLAG_READ 4
```

Definition at line 42 of file elf.h.

5.31.1.4 ELF_PROG_FLAG_WRITE

```
#define ELF_PROG_FLAG_WRITE 2
```

Definition at line 41 of file elf.h.

5.31.1.5 ELF_PROG_LOAD

```
#define ELF_PROG_LOAD 1
```

Definition at line 37 of file elf.h.

5.32 elf.h

Go to the documentation of this file.

```
00001 // Format of an ELF executable file
00002
00003 #define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
00004
00005 // File header
00006 struct elfhdr {
00007   uint magic;  // must equal ELF_MAGIC
00008   uchar elf[12];
00009   ushort type;
00010   ushort machine;
00011   uint version;
00012   uint entry;
00013   uint phoff;
00014   uint shoff;
00015   uint flags;
00016   ushort ehsize;
00017   ushort phentsize;
00018   ushort phnum;
00019   ushort shentsize;
00020   ushort shnum;
00021   ushort shstrndx;
00022 };
00023
00024 // Program section header
00025 struct proghdr {
00026   uint type;
```

```
00027   uint off;
00028   uint vaddr;
00029   uint paddr;
00030   uint filesz;
00031   uint memsz;
00032   uint flags;
00033   uint align;
00034 };
00035
00036 // Values for Proghdr type
00037 #define ELF_PROG_LOAD           1
00038
00039 // Flag bits for Proghdr flags
00040 #define ELF_PROG_FLAG_EXEC      1
00041 #define ELF_PROG_FLAG_WRITE     2
00042 #define ELF_PROG_FLAG_READ      4
```

5.33 entryother.d File Reference

5.34 entryother.d

Go to the documentation of this file.

```
00001 entryother.o: entryother.S asm.h memlayout.h mmu.h
```

5.35 exec.c File Reference

```
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "defs.h"
#include "x86.h"
#include "elf.h"
```

Functions

- int exec (char *path, char **argv)

5.35.1 Function Documentation

5.35.1.1 exec()

```
int exec (
            char * path,
            char ** argv )
```

Definition at line 11 of file exec.c.

```
00012 {
00013   char *s, *last;
00014   int i, off;
00015   uint argc, sz, sp, ustack[3+MAXARG+1];
00016   struct elfhdr elf;
00017   struct inode *ip;
00018   struct proghdr ph;
00019   pde_t *pgdir, *oldpgdir;
00020   struct proc *curproc = myproc();
00021
00022   begin_op();
00023
00024   if((ip = namei(path)) == 0){
00025     end_op();
00026     cprintf("exec: fail\n");
00027     return -1;
00028   }
00029   ilock(ip);
00030   pgdir = 0;
00031
00032   // Check ELF header
00033   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
00034     goto bad;
00035   if(elf.magic != ELF_MAGIC)
00036     goto bad;
00037
00038   if((pgdir = setupkvm()) == 0)
00039     goto bad;
00040
00041   // Load program into memory.
00042   sz = 0;
00043   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
00044     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
00045       goto bad;
00046     if(ph.type != ELF_PROG_LOAD)
00047       continue;
00048     if(ph.memsz < ph.filesz)
00049       goto bad;
00050     if(ph.vaddr + ph.memsz < ph.vaddr)
00051       goto bad;
00052     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
00053       goto bad;
00054     if(ph.vaddr % PGSIZE != 0)
00055       goto bad;
00056     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
00057       goto bad;
00058   }
00059   iunlockput(ip);
00060   end_op();
00061   ip = 0;
00062
00063   // Allocate two pages at the next page boundary.
00064   // Make the first inaccessible.  Use the second as the user stack.
00065   sz = PGROUNDUP(sz);
00066   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
00067     goto bad;
00068   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
00069   sp = sz;
00070
00071   // Push argument strings, prepare rest of stack in ustack.
00072   for(argc = 0; argv[argc]; argc++) {
00073     if(argc >= MAXARG)
00074       goto bad;
00075     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
00076     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
00077       goto bad;
00078     ustack[3+argc] = sp;
00079   }
00080   ustack[3+argc] = 0;
00081
00082   ustack[0] = 0xffffffff;  // fake return PC
00083   ustack[1] = argc;
00084   ustack[2] = sp - (argc+1)*4;  // argv pointer
00085
00086   sp -= (3+argc+1) * 4;
00087   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
00088     goto bad;
```

```
00089
00090   // Save program name for debugging.
00091   for(last=s=path; *s; s++)
00092     if(*s == '/')
00093       last = s+1;
00094   safestrcpy(curproc->name, last, sizeof(curproc->name));
00095
00096   // Commit to the user image.
00097   oldpgdir = curproc->pgdir;
00098   curproc->pgdir = pgdir;
00099   curproc->sz = sz;
00100   curproc->tf->eip = elf.entry;  // main
00101   curproc->tf->esp = sp;
00102   switchuvm(curproc);
00103   freevm(oldpgdir);
00104   return 0;
00105
00106  bad:
00107   if(pgdir)
00108     freevm(pgdir);
00109   if(ip){
00110     iunlockput(ip);
00111     end_op();
00112   }
00113   return -1;
00114 }
```

Referenced by bigargtest(), exectest(), main(), runcmd(), and sys_exec().

5.36 exec.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "param.h"
00003 #include "memlayout.h"
00004 #include "mmu.h"
00005 #include "proc.h"
00006 #include "defs.h"
00007 #include "x86.h"
00008 #include "elf.h"
00009
00010 int
00011 exec(char *path, char **argv)
00012 {
00013   char *s, *last;
00014   int i, off;
00015   uint argc, sz, sp, ustack[3+MAXARG+1];
00016   struct elfhdr elf;
00017   struct inode *ip;
00018   struct proghdr ph;
00019   pde_t *pgdir, *oldpgdir;
00020   struct proc *curproc = myproc();
00021
00022   begin_op();
00023
00024   if((ip = namei(path)) == 0){
00025     end_op();
00026     cprintf("exec: fail\n");
00027     return -1;
00028   }
00029   ilock(ip);
00030   pgdir = 0;
00031
00032   // Check ELF header
00033   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
00034     goto bad;
00035   if(elf.magic != ELF_MAGIC)
00036     goto bad;
00037
00038   if((pgdir = setupkvm()) == 0)
00039     goto bad;
00040
00041   // Load program into memory.
00042   sz = 0;
00043   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
00044     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
00045       goto bad;
00046     if(ph.type != ELF_PROG_LOAD)
00047       continue;
00048     if(ph.memsz < ph.filesz)
```

```
00049       goto bad;
00050     if(ph.vaddr + ph.memsz < ph.vaddr)
00051       goto bad;
00052     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
00053       goto bad;
00054     if(ph.vaddr % PGSIZE != 0)
00055       goto bad;
00056     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
00057       goto bad;
00058   }
00059   iunlockput(ip);
00060   end_op();
00061   ip = 0;
00062
00063   // Allocate two pages at the next page boundary.
00064   // Make the first inaccessible.  Use the second as the user stack.
00065   sz = PGROUNDUP(sz);
00066   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
00067     goto bad;
00068   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
00069   sp = sz;
00070
00071   // Push argument strings, prepare rest of stack in ustack.
00072   for(argc = 0; argv[argc]; argc++) {
00073     if(argc >= MAXARG)
00074       goto bad;
00075     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
00076     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
00077       goto bad;
00078     ustack[3+argc] = sp;
00079   }
00080   ustack[3+argc] = 0;
00081
00082   ustack[0] = 0xffffffff;  // fake return PC
00083   ustack[1] = argc;
00084   ustack[2] = sp - (argc+1)*4;  // argv pointer
00085
00086   sp -= (3+argc+1) * 4;
00087   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
00088     goto bad;
00089
00090   // Save program name for debugging.
00091   for(last=s=path; *s; s++)
00092     if(*s == '/')
00093       last = s+1;
00094   safestrcpy(curproc->name, last, sizeof(curproc->name));
00095
00096   // Commit to the user image.
00097   oldpgdir = curproc->pgdir;
00098   curproc->pgdir = pgdir;
00099   curproc->sz = sz;
00100   curproc->tf->eip = elf.entry;  // main
00101   curproc->tf->esp = sp;
00102   switchuvm(curproc);
00103   freevm(oldpgdir);
00104   return 0;
00105
00106  bad:
00107   if(pgdir)
00108     freevm(pgdir);
00109   if(ip){
00110     iunlockput(ip);
00111     end_op();
00112   }
00113   return -1;
00114 }
```

5.37 exec.d File Reference

5.38 exec.d

Go to the documentation of this file.

```
00001 exec.o: exec.c /usr/include/stdc-predef.h types.h param.h memlayout.h \
00002  mmu.h proc.h defs.h x86.h elf.h
```

5.39 exit.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- int main (int argc, char *argv[])

5.39.1 Function Documentation

5.39.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Definition at line 10 of file exit.c.

```
00011 {
00012     halt();
00013     exit();
00014 }
```

5.40 exit.c

Go to the documentation of this file.

```
00001 // Shuts down the system by using the halt() system call
00002 // to send a special signal to QEMU.
00003 // Added by Bill Katsak
00004 // Copied from: http://pdos.csail.mit.edu/6.828/2012/homework/xv6-syscall.html
00005
00006 #include "types.h"
00007 #include "stat.h"
00008 #include "user.h"
00009
00010 int main(int argc, char *argv[])
00011 {
00012     halt();
00013     exit();
00014 }
```

5.41 exit.d File Reference

5.42 exit.d

Go to the documentation of this file.

```
00001 exit.o: exit.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.43 fcntl.h File Reference

Macros

- #define O_CREATE 0x200
- #define O_RDONLY 0x000
- #define O_RDWR 0x002
- #define O_WRONLY 0x001

5.43.1 Macro Definition Documentation

5.43.1.1 O_CREATE

```
#define O_CREATE 0x200
```

Definition at line 4 of file fcntl.h.

5.43.1.2 O_RDONLY

```
#define O_RDONLY 0x000
```

Definition at line 1 of file fcntl.h.

5.43.1.3 O_RDWR

```
#define O_RDWR 0x002
```

Definition at line 3 of file fcntl.h.

5.43.1.4 O_WRONLY

```
#define O_WRONLY 0x001
```

Definition at line 2 of file fcntl.h.

5.44 fcntl.h

Go to the documentation of this file.

```
00001 #define O_RDONLY  0x000
00002 #define O_WRONLY  0x001
00003 #define O_RDWR    0x002
00004 #define O_CREATE  0x200
```

5.45 file.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "fs.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "file.h"
```

Functions

- struct file * filealloc (void)
- void fileclose (struct file *f)
- struct file * filedup (struct file *f)
- void fileinit (void)
- int fileread (struct file *f, char *addr, int n)
- int filestat (struct file *f, struct stat *st)
- int filewrite (struct file *f, char *addr, int n)

Variables

- struct devsw devsw [NDEV]
- struct {
  struct file file [NFILE]
  struct spinlock lock
  } ftable

5.45.1 Function Documentation

5.45.1.1 filealloc()

```
struct file * filealloc (
            void  )
```

Definition at line 27 of file file.c.

```
00028 {
00029   struct file *f;
00030
00031   acquire(&ftable.lock);
00032   for(f = ftable.file; f < ftable.file + NFILE; f++){
00033     if(f->ref == 0){
00034       f->ref = 1;
00035       release(&ftable.lock);
00036       return f;
00037     }
00038   }
00039   release(&ftable.lock);
00040   return 0;
00041 }
```

Referenced by pipealloc(), and sys_open().

5.45.1.2 fileclose()

```
void fileclose (
            struct file * f )
```

Definition at line 57 of file file.c.

```
00058 {
00059   struct file ff;
00060
00061   acquire(&ftable.lock);
00062   if(f->ref < 1)
00063     panic("fileclose");
00064   if(--f->ref > 0){
00065     release(&ftable.lock);
00066     return;
00067   }
00068   ff = *f;
00069   f->ref = 0;
00070   f->type = FD_NONE;
00071   release(&ftable.lock);
00072
00073   if(ff.type == FD_PIPE)
00074     pipeclose(ff.pipe, ff.writable);
00075   else if(ff.type == FD_INODE){
00076     begin_op();
00077     iput(ff.ip);
00078     end_op();
00079   }
00080 }
```

Referenced by exit(), pipealloc(), sys_close(), sys_open(), and sys_pipe().

5.45.1.3 filedup()

```
struct file * filedup (
            struct file * f )
```

Definition at line 45 of file file.c.

```
00046 {
00047   acquire(&ftable.lock);
00048   if(f->ref < 1)
00049     panic("filedup");
00050   f->ref++;
00051   release(&ftable.lock);
00052   return f;
00053 }
```

Referenced by fork(), and sys_dup().

5.45.1.4 fileinit()

```
void fileinit (
              void  )
```

Definition at line 20 of file file.c.

```
00021 {
00022   initlock(&ftable.lock, "ftable");
00023 }
```

5.45.1.5 fileread()

```
int fileread (
              struct file * f,
              char * addr,
              int n )
```

Definition at line 97 of file file.c.

```
00098 {
00099   int r;
00100
00101   if(f->readable == 0)
00102     return -1;
00103   if(f->type == FD_PIPE)
00104     return piperead(f->pipe, addr, n);
00105   if(f->type == FD_INODE){
00106     ilock(f->ip);
00107     if((r = readi(f->ip, addr, f->off, n)) > 0)
00108       f->off += r;
00109     iunlock(f->ip);
00110     return r;
00111   }
00112   panic("fileread");
00113 }
```

Referenced by sys_read().

5.45.1.6 filestat()

```
int filestat (
              struct file * f,
              struct stat * st )
```

Definition at line 84 of file file.c.

```
00085 {
00086   if(f->type == FD_INODE){
00087     ilock(f->ip);
00088     stati(f->ip, st);
00089     iunlock(f->ip);
00090     return 0;
00091   }
00092   return -1;
00093 }
```

Referenced by sys_fstat().

5.45.1.7 filewrite()

```
int filewrite (
            struct file * f,
            char * addr,
            int n )
```

Definition at line 118 of file file.c.

```
00119 {
00120   int r;
00121
00122   if(f->writable == 0)
00123     return -1;
00124   if(f->type == FD_PIPE)
00125     return pipewrite(f->pipe, addr, n);
00126   if(f->type == FD_INODE){
00127     // write a few blocks at a time to avoid exceeding
00128     // the maximum log transaction size, including
00129     // i-node, indirect block, allocation blocks,
00130     // and 2 blocks of slop for non-aligned writes.
00131     // this really belongs lower down, since writei()
00132     // might be writing a device like the console.
00133     int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
00134     int i = 0;
00135     while(i < n){
00136       int n1 = n - i;
00137       if(n1 > max)
00138         n1 = max;
00139
00140       begin_op();
00141       ilock(f->ip);
00142       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
00143         f->off += r;
00144       iunlock(f->ip);
00145       end_op();
00146
00147       if(r < 0)
00148         break;
00149       if(r != n1)
00150         panic("short filewrite");
00151       i += r;
00152     }
00153     return i == n ? n : -1;
00154   }
00155   panic("filewrite");
00156 }
```

Referenced by sys_write().

5.45.2 Variable Documentation

5.45.2.1 devsw

```
struct devsw devsw[NDEV]
```

Definition at line 13 of file file.c.

5.45.2.2 file

```
struct file file[NFILE]
```

Definition at line 16 of file file.c.

5.45.2.3

```
struct { ...  } ftable
```

Referenced by filealloc(), fileclose(), filedup(), and fileinit().

5.45.2.4 lock

```
struct spinlock lock
```

Definition at line 15 of file file.c.

5.46 file.c

Go to the documentation of this file.

```
00001 //
00002 // File descriptors
00003 //
00004
00005 #include "types.h"
00006 #include "defs.h"
00007 #include "param.h"
00008 #include "fs.h"
00009 #include "spinlock.h"
00010 #include "sleeplock.h"
00011 #include "file.h"
00012
00013 struct devsw devsw[NDEV];
00014 struct {
00015   struct spinlock lock;
00016   struct file file[NFILE];
00017 } ftable;
00018
00019 void
00020 fileinit(void)
00021 {
00022   initlock(&ftable.lock, "ftable");
00023 }
00024
00025 // Allocate a file structure.
00026 struct file*
00027 filealloc(void)
00028 {
00029   struct file *f;
00030
00031   acquire(&ftable.lock);
00032   for(f = ftable.file; f < ftable.file + NFILE; f++){
00033     if(f->ref == 0){
00034       f->ref = 1;
00035       release(&ftable.lock);
00036       return f;
00037     }
00038   }
00039   release(&ftable.lock);
00040   return 0;
00041 }
00042
00043 // Increment ref count for file f.
00044 struct file*
00045 filedup(struct file *f)
00046 {
00047   acquire(&ftable.lock);
00048   if(f->ref < 1)
00049     panic("filedup");
00050   f->ref++;
00051   release(&ftable.lock);
00052   return f;
00053 }
00054
00055 // Close file f.  (Decrement ref count, close when reaches 0.)
```

```
00056 void
00057 fileclose(struct file *f)
00058 {
00059   struct file ff;
00060
00061   acquire(&ftable.lock);
00062   if(f->ref < 1)
00063     panic("fileclose");
00064   if(--f->ref > 0){
00065     release(&ftable.lock);
00066     return;
00067   }
00068   ff = *f;
00069   f->ref = 0;
00070   f->type = FD_NONE;
00071   release(&ftable.lock);
00072
00073   if(ff.type == FD_PIPE)
00074     pipeclose(ff.pipe, ff.writable);
00075   else if(ff.type == FD_INODE){
00076     begin_op();
00077     iput(ff.ip);
00078     end_op();
00079   }
00080 }
00081
00082 // Get metadata about file f.
00083 int
00084 filestat(struct file *f, struct stat *st)
00085 {
00086   if(f->type == FD_INODE){
00087     ilock(f->ip);
00088     stati(f->ip, st);
00089     iunlock(f->ip);
00090     return 0;
00091   }
00092   return -1;
00093 }
00094
00095 // Read from file f.
00096 int
00097 fileread(struct file *f, char *addr, int n)
00098 {
00099   int r;
00100
00101   if(f->readable == 0)
00102     return -1;
00103   if(f->type == FD_PIPE)
00104     return piperead(f->pipe, addr, n);
00105   if(f->type == FD_INODE){
00106     ilock(f->ip);
00107     if((r = readi(f->ip, addr, f->off, n)) > 0)
00108       f->off += r;
00109     iunlock(f->ip);
00110     return r;
00111   }
00112   panic("fileread");
00113 }
00114
00115 //PAGEBREAK!
00116 // Write to file f.
00117 int
00118 filewrite(struct file *f, char *addr, int n)
00119 {
00120   int r;
00121
00122   if(f->writable == 0)
00123     return -1;
00124   if(f->type == FD_PIPE)
00125     return pipewrite(f->pipe, addr, n);
00126   if(f->type == FD_INODE){
00127     // write a few blocks at a time to avoid exceeding
00128     // the maximum log transaction size, including
00129     // i-node, indirect block, allocation blocks,
00130     // and 2 blocks of slop for non-aligned writes.
00131     // this really belongs lower down, since writei()
00132     // might be writing a device like the console.
00133     int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
00134     int i = 0;
00135     while(i < n){
00136       int n1 = n - i;
00137       if(n1 > max)
00138         n1 = max;
00139
00140       begin_op();
00141       ilock(f->ip);
00142       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
```

```
00143         f->off += r;
00144       iunlock(f->ip);
00145       end_op();
00146
00147       if(r < 0)
00148         break;
00149       if(r != n1)
00150         panic("short filewrite");
00151       i += r;
00152     }
00153     return i == n ? n : -1;
00154   }
00155   panic("filewrite");
00156 }
00157
```

5.47 file.d File Reference

5.48 file.d

Go to the documentation of this file.

```
00001 file.o: file.c /usr/include/stdc-predef.h types.h defs.h param.h fs.h \
00002  spinlock.h sleeplock.h file.h
```

5.49 file.h File Reference

Classes

- struct devsw
- struct file
- struct inode

Macros

- #define CONSOLE 1

Variables

- struct devsw devsw [ ]

5.49.1 Macro Definition Documentation

5.49.1.1 CONSOLE

```
#define CONSOLE 1
```

Definition at line 37 of file file.h.

5.49.2 Variable Documentation

5.49.2.1 devsw

```
struct devsw devsw[]  [extern]
```

Definition at line 13 of file file.c.

5.50 file.h

Go to the documentation of this file.

```
00001 struct file {
00002   enum { FD_NONE, FD_PIPE, FD_INODE } type;
00003   int ref; // reference count
00004   char readable;
00005   char writable;
00006   struct pipe *pipe;
00007   struct inode *ip;
00008   uint off;
00009 };
00010
00011
00012 // in-memory copy of an inode
00013 struct inode {
00014   uint dev;           // Device number
00015   uint inum;          // Inode number
00016   int ref;            // Reference count
00017   struct sleeplock lock; // protects everything below here
00018   int valid;          // inode has been read from disk?
00019
00020   short type;         // copy of disk inode
00021   short major;
00022   short minor;
00023   short nlink;
00024   uint size;
00025   uint addrs[NDIRECT+1];
00026 };
00027
00028 // table mapping major device number to
00029 // device functions
00030 struct devsw {
00031   int (*read)(struct inode*, char*, int);
00032   int (*write)(struct inode*, char*, int);
00033 };
00034
00035 extern struct devsw devsw[];
00036
00037 #define CONSOLE 1
```

5.51 foo.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
```

Functions

- int main (int argc, char *argv[])

5.51.1 Function Documentation

5.51.1.1 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 6 of file foo.c.

```
00006                                   {
00007   int pid;
00008   int k, n;
00009   int x, z;
00010
00011   if (argc != 2) {
00012     printf(2, "usage: %s n\n", argv[0]);
00013   }
00014
00015   n = atoi(argv[1]);
00016
00017   for ( k = 0; k < n; k++ ) {
00018     pid = fork ();
00019     if ( pid < 0 ) {
00020       printf(1, "%d failed in fork!\n", getpid());
00021       exit();
00022     } else if (pid == 0) {
00023       // child
00024       printf(1, "Child (with pid=%d) created\n",getpid());
00025       for ( z = 0; z < 100.0; z += 0.1 ){
00026         x =  x + 3.14 * 89.64;   // useless calculations to consume CPU time
00027         if((z== (int)z) && ((z%10) ==0)){
00028           printf(1, "[pid=%d] %d \n",getpid(),z);
00029         }
00030       }
00031
00032       exit();
00033     }
00034   }
00035
00036   for (k = 0; k < n; k++) {
00037     wait();
00038   }
00039
00040   exit();
00041 }
```

5.52 foo.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004 #include "fcntl.h"
00005
00006 int main(int argc, char *argv[]) {
00007   int pid;
00008   int k, n;
00009   int x, z;
00010
00011   if (argc != 2) {
00012     printf(2, "usage: %s n\n", argv[0]);
00013   }
00014
00015   n = atoi(argv[1]);
00016
00017   for ( k = 0; k < n; k++ ) {
00018     pid = fork ();
00019     if ( pid < 0 ) {
00020       printf(1, "%d failed in fork!\n", getpid());
00021       exit();
00022     } else if (pid == 0) {
```

```
00023       // child
00024       printf(1, "Child (with pid=%d) created\n",getpid());
00025       for ( z = 0; z < 100.0; z += 0.1 ){
00026         x =  x + 3.14 * 89.64;   // useless calculations to consume CPU time
00027         if((z== (int)z) && ((z%10) ==0)){
00028           printf(1, "[pid=%d] %d \n",getpid(),z);
00029         }
00030       }
00031
00032       exit();
00033     }
00034   }
00035
00036   for (k = 0; k < n; k++) {
00037     wait();
00038   }
00039
00040   exit();
00041 }
```

5.53 foo.d File Reference

5.54 foo.d

Go to the documentation of this file.

```
00001 foo.o: foo.c /usr/include/stdc-predef.h types.h stat.h user.h fcntl.h
```

5.55 forktest.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Macros

- #define N 1000

Functions

- void forktest (void)
- int main (void)
- void printf (int fd, const char *s,...)

5.55.1 Macro Definition Documentation

5.55.1.1 N

```
#define N 1000
```

Definition at line 8 of file forktest.c.

5.55.2 Function Documentation

5.55.2.1 forktest()

```
void forktest (
              void  )
```

Definition at line 17 of file forktest.c.

```
00018 {
00019   int n, pid;
00020
00021   printf(1, "fork test\n");
00022
00023   for(n=0; n<N; n++){
00024     pid = fork();
00025     if(pid < 0)
00026       break;
00027     if(pid == 0)
00028       exit();
00029   }
00030
00031   if(n == N){
00032     printf(1, "fork claimed to work N times!\n", N);
00033     exit();
00034   }
00035
00036   for(; n > 0; n--){
00037     if(wait() < 0){
00038       printf(1, "wait stopped early\n");
00039       exit();
00040     }
00041   }
00042
00043   if(wait() != -1){
00044     printf(1, "wait got too many\n");
00045     exit();
00046   }
00047
00048   printf(1, "fork test OK\n");
00049 }
```

Referenced by main().

5.55.2.2 main()

```
int main (
              void  )
```

Definition at line 52 of file forktest.c.

```
00053 {
00054   forktest();
00055   exit();
00056 }
```

5.55.2.3 printf()

```
void printf (
             int fd,
             const char * s,
              ...  )
```

Definition at line 11 of file forktest.c.

```
00012 {
00013   write(fd, s, strlen(s));
00014 }
```

Referenced by argptest(), balloc(), bigargtest(), bigdir(), bigfile(), bigwrite(), bsstest(), cat(), concreate(), createdelete(), createtest(), dirfile(), dirtest(), exectest(), exitiputtest(), exitwait(), forktest(), fourfiles(), fourteen(), fsfull(), getcmd(), iputtest(), iref(), linktest(), linkunlink(), ls(), main(), mem(), openiputtest(), opentest(), panic(), parsecmd(), pipe1(), preempt(), rmdot(), runcmd(), sbrktest(), sharedfd(), subdir(), uio(), unlinkread(), validatetest(), wc(), writetest(), and writetest1().

5.56 forktest.c

Go to the documentation of this file.

```
00001 // Test that fork fails gracefully.
00002 // Tiny executable so that the limit can be filling the proc table.
00003
00004 #include "types.h"
00005 #include "stat.h"
00006 #include "user.h"
00007
00008 #define N  1000
00009
00010 void
00011 printf(int fd, const char *s, ...)
00012 {
00013   write(fd, s, strlen(s));
00014 }
00015
00016 void
00017 forktest(void)
00018 {
00019   int n, pid;
00020
00021   printf(1, "fork test\n");
00022
00023   for(n=0; n<N; n++){
00024     pid = fork();
00025     if(pid < 0)
00026       break;
00027     if(pid == 0)
00028       exit();
00029   }
00030
00031   if(n == N){
00032     printf(1, "fork claimed to work N times!\n", N);
00033     exit();
00034   }
00035
00036   for(; n > 0; n--){
00037     if(wait() < 0){
00038       printf(1, "wait stopped early\n");
00039       exit();
00040     }
00041   }
00042
00043   if(wait() != -1){
00044     printf(1, "wait got too many\n");
00045     exit();
00046   }
00047
00048   printf(1, "fork test OK\n");
00049 }
00050
00051 int
00052 main(void)
00053 {
00054   forktest();
00055   exit();
00056 }
```

5.57 forktest.d File Reference

5.58 forktest.d

Go to the documentation of this file.

```
00001 forktest.o: forktest.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.59 fs.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "buf.h"
#include "file.h"
```

Macros

- #define min(a, b) ((a) < (b) ? (a) : (b))

Functions

- static uint balloc (uint dev)
- static void bfree (int dev, uint b)
- static uint bmap (struct inode *ip, uint bn)
- static void bzero (int dev, int bno)
- int dirlink (struct inode *dp, char *name, uint inum)
- struct inode * dirlookup (struct inode *dp, char *name, uint *poff)
- struct inode * ialloc (uint dev, short type)
- struct inode * idup (struct inode *ip)
- static struct inode * iget (uint dev, uint inum)
- void iinit (int dev)
- void ilock (struct inode *ip)
- void iput (struct inode *ip)
- static void itrunc (struct inode *)
- void iunlock (struct inode *ip)
- void iunlockput (struct inode *ip)
- void iupdate (struct inode *ip)
- int namecmp (const char *s, const char *t)
- struct inode * namei (char *path)
- struct inode * nameiparent (char *path, char *name)
- static struct inode * namex (char *path, int nameiparent, char *name)
- int readi (struct inode *ip, char *dst, uint off, uint n)
- void readsb (int dev, struct superblock *sb)
- static char * skipelem (char *path, char *name)
- void stati (struct inode *ip, struct stat *st)
- int writei (struct inode *ip, char *src, uint off, uint n)

Variables

- struct {
  struct inode inode [NINODE]
  struct spinlock lock
  } icache

- struct superblock sb

5.59.1 Macro Definition Documentation

5.59.1.1 min

```
#define min(
              a,
              b ) ((a) < (b) ?  (a) :  (b))
```

Definition at line 24 of file fs.c.

5.59.2 Function Documentation

5.59.2.1 balloc()

```
static uint balloc (
             uint dev )  [static]
```

Definition at line 57 of file fs.c.

```
00058 {
00059   int b, bi, m;
00060   struct buf *bp;
00061
00062   bp = 0;
00063   for(b = 0; b < sb.size; b += BPB){
00064     bp = bread(dev, BBLOCK(b, sb));
00065     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
00066       m = 1 « (bi % 8);
00067       if((bp->data[bi/8] & m) == 0){  // Is block free?
00068         bp->data[bi/8] |= m;  // Mark block in use.
00069         log_write(bp);
00070         brelse(bp);
00071         bzero(dev, b + bi);
00072         return b + bi;
00073       }
00074     }
00075     brelse(bp);
00076   }
00077   panic("balloc: out of blocks");
00078 }
```

Referenced by bmap().

5.59.2.2 bfree()

```
static void bfree (
             int dev,
             uint b )  [static]
```

Definition at line 82 of file fs.c.

```
00083 {
00084   struct buf *bp;
00085   int bi, m;
00086
00087   bp = bread(dev, BBLOCK(b, sb));
00088   bi = b % BPB;
00089   m = 1 « (bi % 8);
00090   if((bp->data[bi/8] & m) == 0)
00091     panic("freeing free block");
00092   bp->data[bi/8] &= ~m;
00093   log_write(bp);
00094   brelse(bp);
00095 }
```

Referenced by itrunc().

5.59.2.3 bmap()

```
static uint bmap (
             struct inode * ip,
             uint bn )  [static]
```

Definition at line 373 of file fs.c.

```
00374 {
00375   uint addr, *a;
00376   struct buf *bp;
00377
00378   if(bn < NDIRECT){
00379     if((addr = ip->addrs[bn]) == 0)
00380       ip->addrs[bn] = addr = balloc(ip->dev);
00381     return addr;
00382   }
00383   bn -= NDIRECT;
00384
00385   if(bn < NINDIRECT){
00386     // Load indirect block, allocating if necessary.
00387     if((addr = ip->addrs[NDIRECT]) == 0)
00388       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
00389     bp = bread(ip->dev, addr);
00390     a = (uint*)bp->data;
00391     if((addr = a[bn]) == 0){
00392       a[bn] = addr = balloc(ip->dev);
00393       log_write(bp);
00394     }
00395     brelse(bp);
00396     return addr;
00397   }
00398
00399   panic("bmap: out of range");
00400 }
```

Referenced by readi(), and writei().

5.59.2.4 bzero()

```
static void bzero (
            int dev,
            int bno )  [static]
```

Definition at line 43 of file fs.c.

```
00044 {
00045   struct buf *bp;
00046
00047   bp = bread(dev, bno);
00048   memset(bp->data, 0, BSIZE);
00049   log_write(bp);
00050   brelse(bp);
00051 }
```

Referenced by balloc(), ialloc(), and main().

5.59.2.5 dirlink()

```
int dirlink (
            struct inode * dp,
            char * name,
            uint inum )
```

Definition at line 552 of file fs.c.

```
00553 {
00554   int off;
00555   struct dirent de;
00556   struct inode *ip;
00557
00558   // Check that name is not present.
00559   if((ip = dirlookup(dp, name, 0)) != 0){
00560     iput(ip);
00561     return -1;
00562   }
00563
00564   // Look for an empty dirent.
00565   for(off = 0; off < dp->size; off += sizeof(de)){
00566     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00567       panic("dirlink read");
00568     if(de.inum == 0)
00569       break;
00570   }
00571
00572   strncpy(de.name, name, DIRSIZ);
00573   de.inum = inum;
00574   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00575     panic("dirlink");
00576
00577   return 0;
00578 }
```

Referenced by create(), and sys_link().

5.59.2.6 dirlookup()

```
struct inode * dirlookup (
              struct inode * dp,
              char * name,
              uint * poff )
```

Definition at line 525 of file fs.c.

```
00526 {
00527   uint off, inum;
00528   struct dirent de;
00529
00530   if(dp->type != T_DIR)
00531     panic("dirlookup not DIR");
00532
00533   for(off = 0; off < dp->size; off += sizeof(de)){
00534     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00535       panic("dirlookup read");
00536     if(de.inum == 0)
00537       continue;
00538     if(namecmp(name, de.name) == 0){
00539       // entry matches path element
00540       if(poff)
00541         *poff = off;
00542       inum = de.inum;
00543       return iget(dp->dev, inum);
00544     }
00545   }
00546
00547   return 0;
00548 }
```

Referenced by create(), dirlink(), namex(), and sys_unlink().

5.59.2.7 ialloc()

```
struct inode * ialloc (
              uint dev,
              short type )
```

Definition at line 195 of file fs.c.

```
00196 {
00197   int inum;
00198   struct buf *bp;
00199   struct dinode *dip;
00200
00201   for(inum = 1; inum < sb.ninodes; inum++){
00202     bp = bread(dev, IBLOCK(inum, sb));
00203     dip = (struct dinode*)bp->data + inum%IPB;
00204     if(dip->type == 0){  // a free inode
00205       memset(dip, 0, sizeof(*dip));
00206       dip->type = type;
00207       log_write(bp);   // mark it allocated on the disk
00208       brelse(bp);
00209       return iget(dev, inum);
00210     }
00211     brelse(bp);
00212   }
00213   panic("ialloc: no inodes");
00214 }
```

Referenced by create().

5.59.2.8 idup()

```
struct inode * idup (
            struct inode * ip )
```

Definition at line 277 of file fs.c.

```
00278 {
00279   acquire(&icache.lock);
00280   ip->ref++;
00281   release(&icache.lock);
00282   return ip;
00283 }
```

Referenced by fork(), and namex().

5.59.2.9 iget()

```
static struct inode * iget (
            uint dev,
            uint inum )  [static]
```

Definition at line 242 of file fs.c.

```
00243 {
00244   struct inode *ip, *empty;
00245
00246   acquire(&icache.lock);
00247
00248   // Is the inode already cached?
00249   empty = 0;
00250   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
00251     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
00252       ip->ref++;
00253       release(&icache.lock);
00254       return ip;
00255     }
00256     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
00257       empty = ip;
00258   }
00259
00260   // Recycle an inode cache entry.
00261   if(empty == 0)
00262     panic("iget: no inodes");
00263
00264   ip = empty;
00265   ip->dev = dev;
00266   ip->inum = inum;
00267   ip->ref = 1;
00268   ip->valid = 0;
00269   release(&icache.lock);
00270
00271   return ip;
00272 }
```

Referenced by dirlookup(), ialloc(), and namex().

5.59.2.10 iinit()

```
void iinit (
             int dev )
```

Definition at line 172 of file fs.c.

```
00173 {
00174   int i = 0;
00175
00176   initlock(&icache.lock, "icache");
00177   for(i = 0; i < NINODE; i++) {
00178     initsleeplock(&icache.inode[i].lock, "inode");
00179   }
00180
00181   readsb(dev, &sb);
00182   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\
00183  inodestart %d bmap start %d\n", sb.size, sb.nblocks,
00184           sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
00185           sb.bmapstart);
00186 }
```

Referenced by forkret().

5.59.2.11 ilock()

```
void ilock (
             struct inode * ip )
```

Definition at line 288 of file fs.c.

```
00289 {
00290   struct buf *bp;
00291   struct dinode *dip;
00292
00293   if(ip == 0 || ip->ref < 1)
00294     panic("ilock");
00295
00296   acquiresleep(&ip->lock);
00297
00298   if(ip->valid == 0){
00299     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
00300     dip = (struct dinode*)bp->data + ip->inum%IPB;
00301     ip->type = dip->type;
00302     ip->major = dip->major;
00303     ip->minor = dip->minor;
00304     ip->nlink = dip->nlink;
00305     ip->size = dip->size;
00306     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
00307     brelse(bp);
00308     ip->valid = 1;
00309     if(ip->type == 0)
00310       panic("ilock: no type");
00311   }
00312 }
```

Referenced by consoleread(), consolewrite(), create(), exec(), fileread(), filestat(), filewrite(), namex(), sys_chdir(), sys_link(), sys_open(), and sys_unlink().

5.59.2.12 iput()

```
void iput (
            struct inode * ip )
```

Definition at line 332 of file fs.c.

```
00333 {
00334   acquiresleep(&ip->lock);
00335   if(ip->valid && ip->nlink == 0){
00336     acquire(&icache.lock);
00337     int r = ip->ref;
00338     release(&icache.lock);
00339     if(r == 1){
00340       // inode has no links and no other references: truncate and free.
00341       itrunc(ip);
00342       ip->type = 0;
00343       iupdate(ip);
00344       ip->valid = 0;
00345     }
00346   }
00347   releasesleep(&ip->lock);
00348
00349   acquire(&icache.lock);
00350   ip->ref--;
00351   release(&icache.lock);
00352 }
```

Referenced by dirlink(), exit(), fileclose(), iunlockput(), namex(), sys_chdir(), and sys_link().

5.59.2.13 itrunc()

```
static void itrunc (
            struct inode * ip )  [static]
```

Definition at line 408 of file fs.c.

```
00409 {
00410   int i, j;
00411   struct buf *bp;
00412   uint *a;
00413
00414   for(i = 0; i < NDIRECT; i++){
00415     if(ip->addrs[i]){
00416       bfree(ip->dev, ip->addrs[i]);
00417       ip->addrs[i] = 0;
00418     }
00419   }
00420
00421   if(ip->addrs[NDIRECT]){
00422     bp = bread(ip->dev, ip->addrs[NDIRECT]);
00423     a = (uint*)bp->data;
00424     for(j = 0; j < NINDIRECT; j++){
00425       if(a[j])
00426         bfree(ip->dev, a[j]);
00427     }
00428     brelse(bp);
00429     bfree(ip->dev, ip->addrs[NDIRECT]);
00430     ip->addrs[NDIRECT] = 0;
00431   }
00432
00433   ip->size = 0;
00434   iupdate(ip);
00435 }
```

Referenced by iput().

5.59.2.14 iunlock()

```
void iunlock (
            struct inode * ip )
```

Definition at line 316 of file fs.c.

```
00317 {
00318   if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
00319     panic("iunlock");
00320
00321   releasesleep(&ip->lock);
00322 }
```

Referenced by consoleread(), consolewrite(), fileread(), filestat(), filewrite(), iunlockput(), namex(), sys_chdir(), sys_link(), and sys_open().

5.59.2.15 iunlockput()

```
void iunlockput (
            struct inode * ip )
```

Definition at line 356 of file fs.c.

```
00357 {
00358   iunlock(ip);
00359   iput(ip);
00360 }
```

Referenced by create(), exec(), namex(), sys_chdir(), sys_link(), sys_mkdir(), sys_mknod(), sys_open(), and sys_unlink().

5.59.2.16 iupdate()

```
void iupdate (
            struct inode * ip )
```

Definition at line 221 of file fs.c.

```
00222 {
00223   struct buf *bp;
00224   struct dinode *dip;
00225
00226   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
00227   dip = (struct dinode*)bp->data + ip->inum%IPB;
00228   dip->type = ip->type;
00229   dip->major = ip->major;
00230   dip->minor = ip->minor;
00231   dip->nlink = ip->nlink;
00232   dip->size = ip->size;
00233   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
00234   log_write(bp);
00235   brelse(bp);
00236 }
```

Referenced by create(), iput(), itrunc(), sys_link(), sys_unlink(), and writei().

5.59.2.17 namecmp()

```
int namecmp (
            const char * s,
            const char * t )
```

Definition at line 517 of file fs.c.

```
00518 {
00519   return strncmp(s, t, DIRSIZ);
00520 }
```

Referenced by dirlookup(), and sys_unlink().

5.59.2.18 namei()

```
struct inode * namei (
            char * path )
```

Definition at line 660 of file fs.c.

```
00661 {
00662   char name[DIRSIZ];
00663   return namex(path, 0, name);
00664 }
```

Referenced by exec(), sys_chdir(), sys_link(), sys_open(), and userinit().

5.59.2.19 nameiparent()

```
struct inode * nameiparent (
            char * path,
            char * name )
```

Definition at line 667 of file fs.c.

```
00668 {
00669   return namex(path, 1, name);
00670 }
```

Referenced by create(), namex(), sys_link(), and sys_unlink().

5.59.2.20 namex()

```
static struct inode * namex (
            char * path,
            int nameiparent,
            char * name )  [static]
```

Definition at line 625 of file fs.c.

```
00626 {
00627   struct inode *ip, *next;
00628
00629   if(*path == '/')
00630     ip = iget(ROOTDEV, ROOTINO);
00631   else
00632     ip = idup(myproc()->cwd);
00633
00634   while((path = skipelem(path, name)) != 0){
00635     ilock(ip);
00636     if(ip->type != T_DIR){
00637       iunlockput(ip);
00638       return 0;
00639     }
00640     if(nameiparent && *path == '\0'){
00641       // Stop one level early.
00642       iunlock(ip);
00643       return ip;
00644     }
00645     if((next = dirlookup(ip, name, 0)) == 0){
00646       iunlockput(ip);
00647       return 0;
00648     }
00649     iunlockput(ip);
00650     ip = next;
00651   }
00652   if(nameiparent){
00653     iput(ip);
00654     return 0;
00655   }
00656   return ip;
00657 }
```

Referenced by namei(), and nameiparent().

5.59.2.21 readi()

```
int readi (
            struct inode * ip,
            char * dst,
            uint off,
            uint n )
```

Definition at line 453 of file fs.c.

```
00454 {
00455   uint tot, m;
00456   struct buf *bp;
00457
00458   if(ip->type == T_DEV){
00459     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
00460       return -1;
00461     return devsw[ip->major].read(ip, dst, n);
00462   }
00463
00464   if(off > ip->size || off + n < off)
00465     return -1;
00466   if(off + n > ip->size)
00467     n = ip->size - off;
00468
00469   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
00470     bp = bread(ip->dev, bmap(ip, off/BSIZE));
00471     m = min(n - tot, BSIZE - off%BSIZE);
00472     memmove(dst, bp->data + off%BSIZE, m);
```

```
00473     brelse(bp);
00474   }
00475   return n;
00476 }
```

Referenced by dirlink(), dirlookup(), exec(), fileread(), isdirempty(), and loaduvm().

5.59.2.22 readsb()

```
void readsb (
              int dev,
              struct superblock * sb )
```

Definition at line 32 of file fs.c.

```
00033 {
00034   struct buf *bp;
00035
00036   bp = bread(dev, 1);
00037   memmove(sb, bp->data, sizeof(*sb));
00038   brelse(bp);
00039 }
```

Referenced by iinit(), and initlog().

5.59.2.23 skipelem()

```
static char * skipelem (
              char * path,
              char * name )  [static]
```

Definition at line 596 of file fs.c.

```
00597 {
00598   char *s;
00599   int len;
00600
00601   while(*path == '/')
00602     path++;
00603   if(*path == 0)
00604     return 0;
00605   s = path;
00606   while(*path != '/' && *path != 0)
00607     path++;
00608   len = path - s;
00609   if(len >= DIRSIZ)
00610     memmove(name, s, DIRSIZ);
00611   else {
00612     memmove(name, s, len);
00613     name[len] = 0;
00614   }
00615   while(*path == '/')
00616     path++;
00617   return path;
00618 }
```

Referenced by namex().

5.59.2.24 stati()

```
void stati (
              struct inode * ip,
              struct stat * st )
```

Definition at line 440 of file fs.c.

```
00441 {
00442   st->dev = ip->dev;
00443   st->ino = ip->inum;
00444   st->type = ip->type;
00445   st->nlink = ip->nlink;
00446   st->size = ip->size;
00447 }
```

Referenced by filestat().

5.59.2.25 writei()

```
int writei (
              struct inode * ip,
              char * src,
              uint off,
              uint n )
```

Definition at line 482 of file fs.c.

```
00483 {
00484   uint tot, m;
00485   struct buf *bp;
00486
00487   if(ip->type == T_DEV){
00488     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
00489       return -1;
00490     return devsw[ip->major].write(ip, src, n);
00491   }
00492
00493   if(off > ip->size || off + n < off)
00494     return -1;
00495   if(off + n > MAXFILE*BSIZE)
00496     return -1;
00497
00498   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
00499     bp = bread(ip->dev, bmap(ip, off/BSIZE));
00500     m = min(n - tot, BSIZE - off%BSIZE);
00501     memmove(bp->data + off%BSIZE, src, m);
00502     log_write(bp);
00503     brelse(bp);
00504   }
00505
00506   if(n > 0 && off > ip->size){
00507     ip->size = off;
00508     iupdate(ip);
00509   }
00510   return n;
00511 }
```

Referenced by dirlink(), filewrite(), and sys_unlink().

5.59.3 Variable Documentation

5.59.3.1

```
struct { ...  } icache
```

Referenced by idup(), iget(), iinit(), and iput().

5.59.3.2 inode

```
struct inode inode[NINODE]
```

Definition at line 168 of file fs.c.

5.59.3.3 lock

```
struct spinlock lock
```

Definition at line 167 of file fs.c.

5.59.3.4 sb

```
struct superblock sb
```

Definition at line 28 of file fs.c.

Referenced by balloc(), bfree(), cmostime(), ialloc(), iinit(), ilock(), initlog(), iupdate(), and readsb().

5.60 fs.c

Go to the documentation of this file.

```
00001 // File system implementation.  Five layers:
00002 //   + Blocks: allocator for raw disk blocks.
00003 //   + Log: crash recovery for multi-step updates.
00004 //   + Files: inode allocator, reading, writing, metadata.
00005 //   + Directories: inode with special contents (list of other inodes!)
00006 //   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
00007 //
00008 // This file contains the low-level file system manipulation
00009 // routines.  The (higher-level) system call implementations
00010 // are in sysfile.c.
00011
00012 #include "types.h"
00013 #include "defs.h"
00014 #include "param.h"
00015 #include "stat.h"
00016 #include "mmu.h"
00017 #include "proc.h"
00018 #include "spinlock.h"
00019 #include "sleeplock.h"
00020 #include "fs.h"
00021 #include "buf.h"
00022 #include "file.h"
00023
```

```
00024 #define min(a, b) ((a) < (b) ? (a) : (b))
00025 static void itrunc(struct inode*);
00026 // there should be one superblock per disk device, but we run with
00027 // only one device
00028 struct superblock sb;
00029
00030 // Read the super block.
00031 void
00032 readsb(int dev, struct superblock *sb)
00033 {
00034   struct buf *bp;
00035
00036   bp = bread(dev, 1);
00037   memmove(sb, bp->data, sizeof(*sb));
00038   brelse(bp);
00039 }
00040
00041 // Zero a block.
00042 static void
00043 bzero(int dev, int bno)
00044 {
00045   struct buf *bp;
00046
00047   bp = bread(dev, bno);
00048   memset(bp->data, 0, BSIZE);
00049   log_write(bp);
00050   brelse(bp);
00051 }
00052
00053 // Blocks.
00054
00055 // Allocate a zeroed disk block.
00056 static uint
00057 balloc(uint dev)
00058 {
00059   int b, bi, m;
00060   struct buf *bp;
00061
00062   bp = 0;
00063   for(b = 0; b < sb.size; b += BPB){
00064     bp = bread(dev, BBLOCK(b, sb));
00065     for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
00066       m = 1 « (bi % 8);
00067       if((bp->data[bi/8] & m) == 0){  // Is block free?
00068         bp->data[bi/8] |= m;  // Mark block in use.
00069         log_write(bp);
00070         brelse(bp);
00071         bzero(dev, b + bi);
00072         return b + bi;
00073       }
00074     }
00075     brelse(bp);
00076   }
00077   panic("balloc: out of blocks");
00078 }
00079
00080 // Free a disk block.
00081 static void
00082 bfree(int dev, uint b)
00083 {
00084   struct buf *bp;
00085   int bi, m;
00086
00087   bp = bread(dev, BBLOCK(b, sb));
00088   bi = b % BPB;
00089   m = 1 « (bi % 8);
00090   if((bp->data[bi/8] & m) == 0)
00091     panic("freeing free block");
00092   bp->data[bi/8] &= ~m;
00093   log_write(bp);
00094   brelse(bp);
00095 }
00096
00097 // Inodes.
00098 //
00099 // An inode describes a single unnamed file.
00100 // The inode disk structure holds metadata: the file's type,
00101 // its size, the number of links referring to it, and the
00102 // list of blocks holding the file's content.
00103 //
00104 // The inodes are laid out sequentially on disk at
00105 // sb.startinode. Each inode has a number, indicating its
00106 // position on the disk.
00107 //
00108 // The kernel keeps a cache of in-use inodes in memory
00109 // to provide a place for synchronizing access
00110 // to inodes used by multiple processes. The cached
```

```
00111 // inodes include book-keeping information that is
00112 // not stored on disk: ip->ref and ip->valid.
00113 //
00114 // An inode and its in-memory representation go through a
00115 // sequence of states before they can be used by the
00116 // rest of the file system code.
00117 //
00118 // * Allocation: an inode is allocated if its type (on disk)
00119 //   is non-zero. ialloc() allocates, and iput() frees if
00120 //   the reference and link counts have fallen to zero.
00121 //
00122 // * Referencing in cache: an entry in the inode cache
00123 //   is free if ip->ref is zero. Otherwise ip->ref tracks
00124 //   the number of in-memory pointers to the entry (open
00125 //   files and current directories). iget() finds or
00126 //   creates a cache entry and increments its ref; iput()
00127 //   decrements ref.
00128 //
00129 // * Valid: the information (type, size, &c) in an inode
00130 //   cache entry is only correct when ip->valid is 1.
00131 //   ilock() reads the inode from
00132 //   the disk and sets ip->valid, while iput() clears
00133 //   ip->valid if ip->ref has fallen to zero.
00134 //
00135 // * Locked: file system code may only examine and modify
00136 //   the information in an inode and its content if it
00137 //   has first locked the inode.
00138 //
00139 // Thus a typical sequence is:
00140 //   ip = iget(dev, inum)
00141 //   ilock(ip)
00142 //   ... examine and modify ip->xxx ...
00143 //   iunlock(ip)
00144 //   iput(ip)
00145 //
00146 // ilock() is separate from iget() so that system calls can
00147 // get a long-term reference to an inode (as for an open file)
00148 // and only lock it for short periods (e.g., in read()).
00149 // The separation also helps avoid deadlock and races during
00150 // pathname lookup. iget() increments ip->ref so that the inode
00151 // stays cached and pointers to it remain valid.
00152 //
00153 // Many internal file system functions expect the caller to
00154 // have locked the inodes involved; this lets callers create
00155 // multi-step atomic operations.
00156 //
00157 // The icache.lock spin-lock protects the allocation of icache
00158 // entries. Since ip->ref indicates whether an entry is free,
00159 // and ip->dev and ip->inum indicate which i-node an entry
00160 // holds, one must hold icache.lock while using any of those fields.
00161 //
00162 // An ip->lock sleep-lock protects all ip-> fields other than ref,
00163 // dev, and inum.  One must hold ip->lock in order to
00164 // read or write that inode's ip->valid, ip->size, ip->type, &c.
00165
00166 struct {
00167   struct spinlock lock;
00168   struct inode inode[NINODE];
00169 } icache;
00170
00171 void
00172 iinit(int dev)
00173 {
00174   int i = 0;
00175
00176   initlock(&icache.lock, "icache");
00177   for(i = 0; i < NINODE; i++) {
00178     initsleeplock(&icache.inode[i].lock, "inode");
00179   }
00180
00181   readsb(dev, &sb);
00182   cprintf("sb: size %d nblocks %d ninodes %d nlog %d logstart %d\
00183  inodestart %d bmap start %d\n", sb.size, sb.nblocks,
00184           sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
00185           sb.bmapstart);
00186 }
00187
00188 static struct inode* iget(uint dev, uint inum);
00189
00190 //PAGEBREAK!
00191 // Allocate an inode on device dev.
00192 // Mark it as allocated by  giving it type type.
00193 // Returns an unlocked but allocated and referenced inode.
00194 struct inode*
00195 ialloc(uint dev, short type)
00196 {
00197   int inum;
```

```
00198   struct buf *bp;
00199   struct dinode *dip;
00200
00201   for(inum = 1; inum < sb.ninodes; inum++){
00202     bp = bread(dev, IBLOCK(inum, sb));
00203     dip = (struct dinode*)bp->data + inum%IPB;
00204     if(dip->type == 0){  // a free inode
00205       memset(dip, 0, sizeof(*dip));
00206       dip->type = type;
00207       log_write(bp);   // mark it allocated on the disk
00208       brelse(bp);
00209       return iget(dev, inum);
00210     }
00211     brelse(bp);
00212   }
00213   panic("ialloc: no inodes");
00214 }
00215
00216 // Copy a modified in-memory inode to disk.
00217 // Must be called after every change to an ip->xxx field
00218 // that lives on disk, since i-node cache is write-through.
00219 // Caller must hold ip->lock.
00220 void
00221 iupdate(struct inode *ip)
00222 {
00223   struct buf *bp;
00224   struct dinode *dip;
00225
00226   bp = bread(ip->dev, IBLOCK(ip->inum, sb));
00227   dip = (struct dinode*)bp->data + ip->inum%IPB;
00228   dip->type = ip->type;
00229   dip->major = ip->major;
00230   dip->minor = ip->minor;
00231   dip->nlink = ip->nlink;
00232   dip->size = ip->size;
00233   memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
00234   log_write(bp);
00235   brelse(bp);
00236 }
00237
00238 // Find the inode with number inum on device dev
00239 // and return the in-memory copy. Does not lock
00240 // the inode and does not read it from disk.
00241 static struct inode*
00242 iget(uint dev, uint inum)
00243 {
00244   struct inode *ip, *empty;
00245
00246   acquire(&icache.lock);
00247
00248   // Is the inode already cached?
00249   empty = 0;
00250   for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
00251     if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
00252       ip->ref++;
00253       release(&icache.lock);
00254       return ip;
00255     }
00256     if(empty == 0 && ip->ref == 0)    // Remember empty slot.
00257       empty = ip;
00258   }
00259
00260   // Recycle an inode cache entry.
00261   if(empty == 0)
00262     panic("iget: no inodes");
00263
00264   ip = empty;
00265   ip->dev = dev;
00266   ip->inum = inum;
00267   ip->ref = 1;
00268   ip->valid = 0;
00269   release(&icache.lock);
00270
00271   return ip;
00272 }
00273
00274 // Increment reference count for ip.
00275 // Returns ip to enable ip = idup(ip1) idiom.
00276 struct inode*
00277 idup(struct inode *ip)
00278 {
00279   acquire(&icache.lock);
00280   ip->ref++;
00281   release(&icache.lock);
00282   return ip;
00283 }
00284
```

```
00285 // Lock the given inode.
00286 // Reads the inode from disk if necessary.
00287 void
00288 ilock(struct inode *ip)
00289 {
00290   struct buf *bp;
00291   struct dinode *dip;
00292
00293   if(ip == 0 || ip->ref < 1)
00294     panic("ilock");
00295
00296   acquiresleep(&ip->lock);
00297
00298   if(ip->valid == 0){
00299     bp = bread(ip->dev, IBLOCK(ip->inum, sb));
00300     dip = (struct dinode*)bp->data + ip->inum%IPB;
00301     ip->type = dip->type;
00302     ip->major = dip->major;
00303     ip->minor = dip->minor;
00304     ip->nlink = dip->nlink;
00305     ip->size = dip->size;
00306     memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
00307     brelse(bp);
00308     ip->valid = 1;
00309     if(ip->type == 0)
00310       panic("ilock: no type");
00311   }
00312 }
00313
00314 // Unlock the given inode.
00315 void
00316 iunlock(struct inode *ip)
00317 {
00318   if(ip == 0 || !holdingsleep(&ip->lock) || ip->ref < 1)
00319     panic("iunlock");
00320
00321   releasesleep(&ip->lock);
00322 }
00323
00324 // Drop a reference to an in-memory inode.
00325 // If that was the last reference, the inode cache entry can
00326 // be recycled.
00327 // If that was the last reference and the inode has no links
00328 // to it, free the inode (and its content) on disk.
00329 // All calls to iput() must be inside a transaction in
00330 // case it has to free the inode.
00331 void
00332 iput(struct inode *ip)
00333 {
00334   acquiresleep(&ip->lock);
00335   if(ip->valid && ip->nlink == 0){
00336     acquire(&icache.lock);
00337     int r = ip->ref;
00338     release(&icache.lock);
00339     if(r == 1){
00340       // inode has no links and no other references: truncate and free.
00341       itrunc(ip);
00342       ip->type = 0;
00343       iupdate(ip);
00344       ip->valid = 0;
00345     }
00346   }
00347   releasesleep(&ip->lock);
00348
00349   acquire(&icache.lock);
00350   ip->ref--;
00351   release(&icache.lock);
00352 }
00353
00354 // Common idiom: unlock, then put.
00355 void
00356 iunlockput(struct inode *ip)
00357 {
00358   iunlock(ip);
00359   iput(ip);
00360 }
00361
00362 //PAGEBREAK!
00363 // Inode content
00364 //
00365 // The content (data) associated with each inode is stored
00366 // in blocks on the disk. The first NDIRECT block numbers
00367 // are listed in ip->addrs[].  The next NINDIRECT blocks are
00368 // listed in block ip->addrs[NDIRECT].
00369
00370 // Return the disk block address of the nth block in inode ip.
00371 // If there is no such block, bmap allocates one.
```

```
00372 static uint
00373 bmap(struct inode *ip, uint bn)
00374 {
00375   uint addr, *a;
00376   struct buf *bp;
00377
00378   if(bn < NDIRECT){
00379     if((addr = ip->addrs[bn]) == 0)
00380       ip->addrs[bn] = addr = balloc(ip->dev);
00381     return addr;
00382   }
00383   bn -= NDIRECT;
00384
00385   if(bn < NINDIRECT){
00386     // Load indirect block, allocating if necessary.
00387     if((addr = ip->addrs[NDIRECT]) == 0)
00388       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
00389     bp = bread(ip->dev, addr);
00390     a = (uint*)bp->data;
00391     if((addr = a[bn]) == 0){
00392       a[bn] = addr = balloc(ip->dev);
00393       log_write(bp);
00394     }
00395     brelse(bp);
00396     return addr;
00397   }
00398
00399   panic("bmap: out of range");
00400 }
00401
00402 // Truncate inode (discard contents).
00403 // Only called when the inode has no links
00404 // to it (no directory entries referring to it)
00405 // and has no in-memory reference to it (is
00406 // not an open file or current directory).
00407 static void
00408 itrunc(struct inode *ip)
00409 {
00410   int i, j;
00411   struct buf *bp;
00412   uint *a;
00413
00414   for(i = 0; i < NDIRECT; i++){
00415     if(ip->addrs[i]){
00416       bfree(ip->dev, ip->addrs[i]);
00417       ip->addrs[i] = 0;
00418     }
00419   }
00420
00421   if(ip->addrs[NDIRECT]){
00422     bp = bread(ip->dev, ip->addrs[NDIRECT]);
00423     a = (uint*)bp->data;
00424     for(j = 0; j < NINDIRECT; j++){
00425       if(a[j])
00426         bfree(ip->dev, a[j]);
00427     }
00428     brelse(bp);
00429     bfree(ip->dev, ip->addrs[NDIRECT]);
00430     ip->addrs[NDIRECT] = 0;
00431   }
00432
00433   ip->size = 0;
00434   iupdate(ip);
00435 }
00436
00437 // Copy stat information from inode.
00438 // Caller must hold ip->lock.
00439 void
00440 stati(struct inode *ip, struct stat *st)
00441 {
00442   st->dev = ip->dev;
00443   st->ino = ip->inum;
00444   st->type = ip->type;
00445   st->nlink = ip->nlink;
00446   st->size = ip->size;
00447 }
00448
00449 //PAGEBREAK!
00450 // Read data from inode.
00451 // Caller must hold ip->lock.
00452 int
00453 readi(struct inode *ip, char *dst, uint off, uint n)
00454 {
00455   uint tot, m;
00456   struct buf *bp;
00457
00458   if(ip->type == T_DEV){
```

```
00459     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
00460       return -1;
00461     return devsw[ip->major].read(ip, dst, n);
00462   }
00463
00464   if(off > ip->size || off + n < off)
00465     return -1;
00466   if(off + n > ip->size)
00467     n = ip->size - off;
00468
00469   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
00470     bp = bread(ip->dev, bmap(ip, off/BSIZE));
00471     m = min(n - tot, BSIZE - off%BSIZE);
00472     memmove(dst, bp->data + off%BSIZE, m);
00473     brelse(bp);
00474   }
00475   return n;
00476 }
00477
00478 // PAGEBREAK!
00479 // Write data to inode.
00480 // Caller must hold ip->lock.
00481 int
00482 writei(struct inode *ip, char *src, uint off, uint n)
00483 {
00484   uint tot, m;
00485   struct buf *bp;
00486
00487   if(ip->type == T_DEV){
00488     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
00489       return -1;
00490     return devsw[ip->major].write(ip, src, n);
00491   }
00492
00493   if(off > ip->size || off + n < off)
00494     return -1;
00495   if(off + n > MAXFILE*BSIZE)
00496     return -1;
00497
00498   for(tot=0; tot<n; tot+=m, off+=m, src+=m){
00499     bp = bread(ip->dev, bmap(ip, off/BSIZE));
00500     m = min(n - tot, BSIZE - off%BSIZE);
00501     memmove(bp->data + off%BSIZE, src, m);
00502     log_write(bp);
00503     brelse(bp);
00504   }
00505
00506   if(n > 0 && off > ip->size){
00507     ip->size = off;
00508     iupdate(ip);
00509   }
00510   return n;
00511 }
00512
00513 //PAGEBREAK!
00514 // Directories
00515
00516 int
00517 namecmp(const char *s, const char *t)
00518 {
00519   return strncmp(s, t, DIRSIZ);
00520 }
00521
00522 // Look for a directory entry in a directory.
00523 // If found, set *poff to byte offset of entry.
00524 struct inode*
00525 dirlookup(struct inode *dp, char *name, uint *poff)
00526 {
00527   uint off, inum;
00528   struct dirent de;
00529
00530   if(dp->type != T_DIR)
00531     panic("dirlookup not DIR");
00532
00533   for(off = 0; off < dp->size; off += sizeof(de)){
00534     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00535       panic("dirlookup read");
00536     if(de.inum == 0)
00537       continue;
00538     if(namecmp(name, de.name) == 0){
00539       // entry matches path element
00540       if(poff)
00541         *poff = off;
00542       inum = de.inum;
00543       return iget(dp->dev, inum);
00544     }
00545   }
```

```
00546
00547   return 0;
00548 }
00549
00550 // Write a new directory entry (name, inum) into the directory dp.
00551 int
00552 dirlink(struct inode *dp, char *name, uint inum)
00553 {
00554   int off;
00555   struct dirent de;
00556   struct inode *ip;
00557
00558   // Check that name is not present.
00559   if((ip = dirlookup(dp, name, 0)) != 0){
00560     iput(ip);
00561     return -1;
00562   }
00563
00564   // Look for an empty dirent.
00565   for(off = 0; off < dp->size; off += sizeof(de)){
00566     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00567       panic("dirlink read");
00568     if(de.inum == 0)
00569       break;
00570   }
00571
00572   strncpy(de.name, name, DIRSIZ);
00573   de.inum = inum;
00574   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00575     panic("dirlink");
00576
00577   return 0;
00578 }
00579
00580 //PAGEBREAK!
00581 // Paths
00582
00583 // Copy the next path element from path into name.
00584 // Return a pointer to the element following the copied one.
00585 // The returned path has no leading slashes,
00586 // so the caller can check *path=='\0' to see if the name is the last one.
00587 // If no name to remove, return 0.
00588 //
00589 // Examples:
00590 //   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
00591 //   skipelem("///a//bb", name) = "bb", setting name = "a"
00592 //   skipelem("a", name) = "", setting name = "a"
00593 //   skipelem("", name) = skipelem("////", name) = 0
00594 //
00595 static char*
00596 skipelem(char *path, char *name)
00597 {
00598   char *s;
00599   int len;
00600
00601   while(*path == '/')
00602     path++;
00603   if(*path == 0)
00604     return 0;
00605   s = path;
00606   while(*path != '/' && *path != 0)
00607     path++;
00608   len = path - s;
00609   if(len >= DIRSIZ)
00610     memmove(name, s, DIRSIZ);
00611   else {
00612     memmove(name, s, len);
00613     name[len] = 0;
00614   }
00615   while(*path == '/')
00616     path++;
00617   return path;
00618 }
00619
00620 // Look up and return the inode for a path name.
00621 // If parent != 0, return the inode for the parent and copy the final
00622 // path element into name, which must have room for DIRSIZ bytes.
00623 // Must be called inside a transaction since it calls iput().
00624 static struct inode*
00625 namex(char *path, int nameiparent, char *name)
00626 {
00627   struct inode *ip, *next;
00628
00629   if(*path == '/')
00630     ip = iget(ROOTDEV, ROOTINO);
00631   else
00632     ip = idup(myproc()->cwd);
```

```
00633
00634   while((path = skipelem(path, name)) != 0){
00635     ilock(ip);
00636     if(ip->type != T_DIR){
00637       iunlockput(ip);
00638       return 0;
00639     }
00640     if(nameiparent && *path == '\0'){
00641       // Stop one level early.
00642       iunlock(ip);
00643       return ip;
00644     }
00645     if((next = dirlookup(ip, name, 0)) == 0){
00646       iunlockput(ip);
00647       return 0;
00648     }
00649     iunlockput(ip);
00650     ip = next;
00651   }
00652   if(nameiparent){
00653     iput(ip);
00654     return 0;
00655   }
00656   return ip;
00657 }
00658
00659 struct inode*
00660 namei(char *path)
00661 {
00662   char name[DIRSIZ];
00663   return namex(path, 0, name);
00664 }
00665
00666 struct inode*
00667 nameiparent(char *path, char *name)
00668 {
00669   return namex(path, 1, name);
00670 }
```

5.61 fs.d File Reference

5.62 fs.d

Go to the documentation of this file.

```
00001 fs.o: fs.c /usr/include/stdc-predef.h types.h defs.h param.h stat.h mmu.h \
00002  proc.h spinlock.h sleeplock.h fs.h buf.h file.h
```

5.63 fs.h File Reference

Classes

- struct dinode
- struct dirent
- struct superblock

Macros

- #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
- #define BPB (BSIZE*8)
- #define BSIZE 512
- #define DIRSIZ 14
- #define IBLOCK(i, sb) ((i) / IPB + sb.inodestart)
- #define IPB (BSIZE / sizeof(struct dinode))
- #define MAXFILE (NDIRECT + NINDIRECT)
- #define NDIRECT 12
- #define NINDIRECT (BSIZE / sizeof(uint))
- #define ROOTINO 1

5.63.1 Macro Definition Documentation

5.63.1.1 BBLOCK

```
#define BBLOCK(
              b,
              sb ) (b/BPB + sb.bmapstart)
```

Definition at line 48 of file fs.h.

5.63.1.2 BPB

```
#define BPB (BSIZE*8)
```

Definition at line 45 of file fs.h.

5.63.1.3 BSIZE

```
#define BSIZE 512
```

Definition at line 6 of file fs.h.

5.63.1.4 DIRSIZ

```
#define DIRSIZ 14
```

Definition at line 51 of file fs.h.

5.63.1.5 IBLOCK

```
#define IBLOCK(
              i,
              sb ) ((i) / IPB + sb.inodestart)
```

Definition at line 42 of file fs.h.

5.63.1.6 IPB

```
#define IPB (BSIZE / sizeof(struct dinode))
```

Definition at line 39 of file fs.h.

5.63.1.7 MAXFILE

```
#define MAXFILE (NDIRECT + NINDIRECT)
```

Definition at line 26 of file fs.h.

5.63.1.8 NDIRECT

```
#define NDIRECT 12
```

Definition at line 24 of file fs.h.

5.63.1.9 NINDIRECT

```
#define NINDIRECT (BSIZE / sizeof(uint))
```

Definition at line 25 of file fs.h.

5.63.1.10 ROOTINO

```
#define ROOTINO 1
```

Definition at line 5 of file fs.h.

5.64 fs.h

Go to the documentation of this file.

```
00001 // On-disk file system format.
00002 // Both the kernel and user programs use this header file.
00003
00004
00005 #define ROOTINO 1  // root i-number
00006 #define BSIZE 512  // block size
00007
00008 // Disk layout:
00009 // [ boot block | super block | log | inode blocks |
00010 //                                          free bit map | data blocks]
00011 //
00012 // mkfs computes the super block and builds an initial file system. The
00013 // super block describes the disk layout:
00014 struct superblock {
00015   uint size;         // Size of file system image (blocks)
00016   uint nblocks;      // Number of data blocks
00017   uint ninodes;      // Number of inodes.
00018   uint nlog;         // Number of log blocks
00019   uint logstart;     // Block number of first log block
00020   uint inodestart;   // Block number of first inode block
00021   uint bmapstart;    // Block number of first free map block
00022 };
00023
00024 #define NDIRECT 12
00025 #define NINDIRECT (BSIZE / sizeof(uint))
00026 #define MAXFILE (NDIRECT + NINDIRECT)
00027
00028 // On-disk inode structure
00029 struct dinode {
00030   short type;           // File type
00031   short major;          // Major device number (T_DEV only)
00032   short minor;          // Minor device number (T_DEV only)
00033   short nlink;          // Number of links to inode in file system
00034   uint size;            // Size of file (bytes)
00035   uint addrs[NDIRECT+1];   // Data block addresses
00036 };
00037
00038 // Inodes per block.
00039 #define IPB           (BSIZE / sizeof(struct dinode))
00040
00041 // Block containing inode i
00042 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
00043
00044 // Bitmap bits per block
00045 #define BPB           (BSIZE*8)
00046
00047 // Block of free map containing bit for block b
00048 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
00049
00050 // Directory is a file containing a sequence of dirent structures.
00051 #define DIRSIZ 14
00052
00053 struct dirent {
00054   ushort inum;
00055   char name[DIRSIZ];
00056 };
00057
```

5.65 grep.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- void grep (char *pattern, int fd)
- int main (int argc, char *argv[])
- int match (char *, char *)
- int matchhere (char *, char *)
- int matchstar (int, char *, char *)

Variables

- char buf [1024]

5.65.1 Function Documentation

5.65.1.1 grep()

```
void grep (
            char * pattern,
            int fd )
```

Definition at line 11 of file grep.c.

```
00012 {
00013   int n, m;
00014   char *p, *q;
00015
00016   m = 0;
00017   while((n = read(fd, buf+m, sizeof(buf)-m-1)) > 0){
00018     m += n;
00019     buf[m] = '\0';
00020     p = buf;
00021     while((q = strchr(p, '\n')) != 0){
00022       *q = 0;
00023       if(match(pattern, p)){
00024         *q = '\n';
00025         write(1, p, q+1 - p);
00026       }
00027       p = q+1;
00028     }
00029     if(p == buf)
00030       m = 0;
00031     if(m > 0){
00032       m -= p - buf;
00033       memmove(buf, p, m);
00034     }
00035   }
00036 }
```

Referenced by main().

5.65.1.2 main()

```
int main (
            int argc,
            char * argv[] )
```

Definition at line 39 of file grep.c.

```
00040 {
00041   int fd, i;
00042   char *pattern;
00043
00044   if(argc <= 1){
00045     printf(2, "usage: grep pattern [file ...]\n");
00046     exit();
00047   }
00048   pattern = argv[1];
00049
00050   if(argc <= 2){
00051     grep(pattern, 0);
00052     exit();
00053   }
```

```
00054
00055   for(i = 2; i < argc; i++){
00056     if((fd = open(argv[i], 0)) < 0){
00057       printf(1, "grep: cannot open %s\n", argv[i]);
00058       exit();
00059     }
00060     grep(pattern, fd);
00061     close(fd);
00062   }
00063   exit();
00064 }
```

5.65.1.3 match()

```
int match (
              char * re,
              char * text )
```

Definition at line 73 of file grep.c.

```
00074 {
00075   if(re[0] == '^')
00076     return matchhere(re+1, text);
00077   do{  // must look at empty string
00078     if(matchhere(re, text))
00079       return 1;
00080   }while(*text++ != '\0');
00081   return 0;
00082 }
```

Referenced by grep().

5.65.1.4 matchhere()

```
int matchhere (
              char * re,
              char * text )
```

Definition at line 85 of file grep.c.

```
00086 {
00087   if(re[0] == '\0')
00088     return 1;
00089   if(re[1] == '*')
00090     return matchstar(re[0], re+2, text);
00091   if(re[0] == '$' && re[1] == '\0')
00092     return *text == '\0';
00093   if(*text!='\0' && (re[0]=='.' || re[0]==*text))
00094     return matchhere(re+1, text+1);
00095   return 0;
00096 }
```

Referenced by match(), matchhere(), and matchstar().

5.65.1.5 matchstar()

```
int matchstar (
              int c,
              char * re,
              char * text )
```

Definition at line 99 of file grep.c.

```
00100 {
00101   do{  // a * matches zero or more instances
00102     if(matchhere(re, text))
00103       return 1;
00104   }while(*text!='\0' && (*text++==c || c=='.'));
00105   return 0;
00106 }
```

Referenced by matchhere().

5.65.2 Variable Documentation

5.65.2.1 buf

```
char buf[1024]
```

Definition at line 7 of file grep.c.

Referenced by grep().

5.66 grep.c

Go to the documentation of this file.

```
00001 // Simple grep.  Only supports ^ . * $ operators.
00002
00003 #include "types.h"
00004 #include "stat.h"
00005 #include "user.h"
00006
00007 char buf[1024];
00008 int match(char*, char*);
00009
00010 void
00011 grep(char *pattern, int fd)
00012 {
00013   int n, m;
00014   char *p, *q;
00015
00016   m = 0;
00017   while((n = read(fd, buf+m, sizeof(buf)-m-1)) > 0){
00018     m += n;
00019     buf[m] = '\0';
00020     p = buf;
00021     while((q = strchr(p, '\n')) != 0){
00022       *q = 0;
00023       if(match(pattern, p)){
00024         *q = '\n';
00025         write(1, p, q+1 - p);
00026       }
00027       p = q+1;
00028     }
00029     if(p == buf)
00030       m = 0;
00031     if(m > 0){
00032       m -= p - buf;
```

```
00033       memmove(buf, p, m);
00034     }
00035   }
00036 }
00037
00038 int
00039 main(int argc, char *argv[])
00040 {
00041   int fd, i;
00042   char *pattern;
00043
00044   if(argc <= 1){
00045     printf(2, "usage: grep pattern [file ...]\n");
00046     exit();
00047   }
00048   pattern = argv[1];
00049
00050   if(argc <= 2){
00051     grep(pattern, 0);
00052     exit();
00053   }
00054
00055   for(i = 2; i < argc; i++){
00056     if((fd = open(argv[i], 0)) < 0){
00057       printf(1, "grep: cannot open %s\n", argv[i]);
00058       exit();
00059     }
00060     grep(pattern, fd);
00061     close(fd);
00062   }
00063   exit();
00064 }
00065
00066 // Regexp matcher from Kernighan & Pike,
00067 // The Practice of Programming, Chapter 9.
00068
00069 int matchhere(char*, char*);
00070 int matchstar(int, char*, char*);
00071
00072 int
00073 match(char *re, char *text)
00074 {
00075   if(re[0] == '^')
00076     return matchhere(re+1, text);
00077   do{  // must look at empty string
00078     if(matchhere(re, text))
00079       return 1;
00080   }while(*text++ != '\0');
00081   return 0;
00082 }
00083
00084 // matchhere: search for re at beginning of text
00085 int matchhere(char *re, char *text)
00086 {
00087   if(re[0] == '\0')
00088     return 1;
00089   if(re[1] == '*')
00090     return matchstar(re[0], re+2, text);
00091   if(re[0] == '$' && re[1] == '\0')
00092     return *text == '\0';
00093   if(*text!='\0' && (re[0]=='.' || re[0]==*text))
00094     return matchhere(re+1, text+1);
00095   return 0;
00096 }
00097
00098 // matchstar: search for c*re at beginning of text
00099 int matchstar(int c, char *re, char *text)
00100 {
00101   do{  // a * matches zero or more instances
00102     if(matchhere(re, text))
00103       return 1;
00104   }while(*text!='\0' && (*text++==c || c=='.'));
00105   return 0;
00106 }
00107
```

5.67 grep.d File Reference

5.68 grep.d

Go to the documentation of this file.

```
00001 grep.o: grep.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.69 ide.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "buf.h"
```

Macros

- #define IDE_BSY 0x80
- #define IDE_CMD_RDMUL 0xc4
- #define IDE_CMD_READ 0x20
- #define IDE_CMD_WRITE 0x30
- #define IDE_CMD_WRMUL 0xc5
- #define IDE_DF 0x20
- #define IDE_DRDY 0x40
- #define IDE_ERR 0x01
- #define SECTOR_SIZE 512

Functions

- void ideinit (void)
- void ideintr (void)
- void iderw (struct buf *b)
- static void idestart (struct buf *)
- static int idewait (int checkerr)

Variables

- static int havedisk1
- static struct spinlock idelock
- static struct buf * idequeue

5.69.1 Macro Definition Documentation

5.69.1.1 IDE_BSY

```
#define IDE_BSY 0x80
```

Definition at line 17 of file ide.c.

5.69.1.2 IDE_CMD_RDMUL

```
#define IDE_CMD_RDMUL 0xc4
```

Definition at line 24 of file ide.c.

5.69.1.3 IDE_CMD_READ

```
#define IDE_CMD_READ 0x20
```

Definition at line 22 of file ide.c.

5.69.1.4 IDE_CMD_WRITE

```
#define IDE_CMD_WRITE 0x30
```

Definition at line 23 of file ide.c.

5.69.1.5 IDE_CMD_WRMUL

```
#define IDE_CMD_WRMUL 0xc5
```

Definition at line 25 of file ide.c.

5.69.1.6 IDE_DF

```
#define IDE_DF 0x20
```

Definition at line 19 of file ide.c.

5.69.1.7 IDE_DRDY

```
#define IDE_DRDY 0x40
```

Definition at line 18 of file ide.c.

5.69.1.8 IDE_ERR

```
#define IDE_ERR 0x01
```

Definition at line 20 of file ide.c.

5.69.1.9 SECTOR_SIZE

```
#define SECTOR_SIZE 512
```

Definition at line 16 of file ide.c.

5.69.2 Function Documentation

5.69.2.1 ideinit()

```
void ideinit (
             void  )
```

Definition at line 51 of file ide.c.

```
00052 {
00053   int i;
00054
00055   initlock(&idelock, "ide");
00056   ioapicenable(IRQ_IDE, ncpu - 1);
00057   idewait(0);
00058
00059   // Check if disk 1 is present
00060   outb(0x1f6, 0xe0 | (1<<4));
00061   for(i=0; i<1000; i++){
00062     if(inb(0x1f7) != 0){
00063       havedisk1 = 1;
00064       break;
00065     }
00066   }
00067
00068   // Switch back to disk 0.
00069   outb(0x1f6, 0xe0 | (0<<4));
00070 }
```

5.69.2.2 ideintr()

```
void ideintr (
              void  )
```

Definition at line 104 of file ide.c.

```
00105 {
00106   struct buf *b;
00107
00108   // First queued buffer is the active request.
00109   acquire(&idelock);
00110
00111   if((b = idequeue) == 0){
00112     release(&idelock);
00113     return;
00114   }
00115   idequeue = b->qnext;
00116
00117   // Read data if needed.
00118   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
00119     insl(0x1f0, b->data, BSIZE/4);
00120
00121   // Wake process waiting for this buf.
00122   b->flags |= B_VALID;
00123   b->flags &= ~B_DIRTY;
00124   wakeup(b);
00125
00126   // Start disk on next buf in queue.
00127   if(idequeue != 0)
00128     idestart(idequeue);
00129
00130   release(&idelock);
00131 }
```

5.69.2.3 iderw()

```
void iderw (
              struct buf * b )
```

Definition at line 138 of file ide.c.

```
00139 {
00140   struct buf **pp;
00141
00142   if(!holdingsleep(&b->lock))
00143     panic("iderw: buf not locked");
00144   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
00145     panic("iderw: nothing to do");
00146   if(b->dev != 0 && !havedisk1)
00147     panic("iderw: ide disk 1 not present");
00148
00149   acquire(&idelock);  //DOC:acquire-lock
00150
00151   // Append b to idequeue.
00152   b->qnext = 0;
00153   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC:insert-queue
00154     ;
00155   *pp = b;
00156
00157   // Start disk if necessary.
00158   if(idequeue == b)
00159     idestart(b);
00160
00161   // Wait for request to finish.
00162   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
00163     sleep(b, &idelock);
00164   }
00165
00166
00167   release(&idelock);
00168 }
```

5.69.2.4 idestart()

```
static void idestart (
            struct buf * b )  [static]
```

Definition at line 74 of file ide.c.

```
00075 {
00076   if(b == 0)
00077     panic("idestart");
00078   if(b->blockno >= FSSIZE)
00079     panic("incorrect blockno");
00080   int sector_per_block =  BSIZE/SECTOR_SIZE;
00081   int sector = b->blockno * sector_per_block;
00082   int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ :  IDE_CMD_RDMUL;
00083   int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;
00084
00085   if (sector_per_block > 7) panic("idestart");
00086
00087   idewait(0);
00088   outb(0x3f6, 0);  // generate interrupt
00089   outb(0x1f2, sector_per_block);  // number of sectors
00090   outb(0x1f3, sector & 0xff);
00091   outb(0x1f4, (sector » 8) & 0xff);
00092   outb(0x1f5, (sector » 16) & 0xff);
00093   outb(0x1f6, 0xe0 | ((b->dev&1)«4) | ((sector»24)&0x0f));
00094   if(b->flags & B_DIRTY){
00095     outb(0x1f7, write_cmd);
00096     outsl(0x1f0, b->data, BSIZE/4);
00097   } else {
00098     outb(0x1f7, read_cmd);
00099   }
00100 }
```

Referenced by ideintr(), and iderw().

5.69.2.5 idewait()

```
static int idewait (
            int checkerr )  [static]
```

Definition at line 39 of file ide.c.

```
00040 {
00041   int r;
00042
00043   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
00044     ;
00045   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
00046     return -1;
00047   return 0;
00048 }
```

Referenced by ideinit(), ideintr(), and idestart().

5.69.3 Variable Documentation

5.69.3.1 havedisk1

```
int havedisk1  [static]
```

Definition at line 34 of file ide.c.

Referenced by ideinit(), and iderw().

5.69.3.2 idelock

```
struct spinlock idelock  [static]
```

Definition at line 31 of file ide.c.

Referenced by ideinit(), ideintr(), and iderw().

5.69.3.3 idequeue

```
struct buf* idequeue  [static]
```

Definition at line 32 of file ide.c.

Referenced by ideintr(), and iderw().

5.70 ide.c

Go to the documentation of this file.

```
00001 // Simple PIO-based (non-DMA) IDE driver code.
00002
00003 #include "types.h"
00004 #include "defs.h"
00005 #include "param.h"
00006 #include "memlayout.h"
00007 #include "mmu.h"
00008 #include "proc.h"
00009 #include "x86.h"
00010 #include "traps.h"
00011 #include "spinlock.h"
00012 #include "sleeplock.h"
00013 #include "fs.h"
00014 #include "buf.h"
00015
00016 #define SECTOR_SIZE   512
00017 #define IDE_BSY       0x80
00018 #define IDE_DRDY      0x40
00019 #define IDE_DF        0x20
00020 #define IDE_ERR       0x01
00021
00022 #define IDE_CMD_READ  0x20
00023 #define IDE_CMD_WRITE 0x30
00024 #define IDE_CMD_RDMUL 0xc4
00025 #define IDE_CMD_WRMUL 0xc5
00026
00027 // idequeue points to the buf now being read/written to the disk.
00028 // idequeue->qnext points to the next buf to be processed.
00029 // You must hold idelock while manipulating queue.
00030
00031 static struct spinlock idelock;
00032 static struct buf *idequeue;
00033
00034 static int havedisk1;
00035 static void idestart(struct buf*);
00036
00037 // Wait for IDE disk to become ready.
00038 static int
00039 idewait(int checkerr)
00040 {
00041   int r;
00042
00043   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
00044     ;
00045   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
00046     return -1;
00047   return 0;
00048 }
00049
```

```
00050 void
00051 ideinit(void)
00052 {
00053   int i;
00054
00055   initlock(&idelock, "ide");
00056   ioapicenable(IRQ_IDE, ncpu - 1);
00057   idewait(0);
00058
00059   // Check if disk 1 is present
00060   outb(0x1f6, 0xe0 | (1<<4));
00061   for(i=0; i<1000; i++){
00062     if(inb(0x1f7) != 0){
00063       havedisk1 = 1;
00064       break;
00065     }
00066   }
00067
00068   // Switch back to disk 0.
00069   outb(0x1f6, 0xe0 | (0<<4));
00070 }
00071
00072 // Start the request for b.  Caller must hold idelock.
00073 static void
00074 idestart(struct buf *b)
00075 {
00076   if(b == 0)
00077     panic("idestart");
00078   if(b->blockno >= FSSIZE)
00079     panic("incorrect blockno");
00080   int sector_per_block =  BSIZE/SECTOR_SIZE;
00081   int sector = b->blockno * sector_per_block;
00082   int read_cmd = (sector_per_block == 1) ? IDE_CMD_READ :  IDE_CMD_RDMUL;
00083   int write_cmd = (sector_per_block == 1) ? IDE_CMD_WRITE : IDE_CMD_WRMUL;
00084
00085   if (sector_per_block > 7) panic("idestart");
00086
00087   idewait(0);
00088   outb(0x3f6, 0);  // generate interrupt
00089   outb(0x1f2, sector_per_block);  // number of sectors
00090   outb(0x1f3, sector & 0xff);
00091   outb(0x1f4, (sector » 8) & 0xff);
00092   outb(0x1f5, (sector » 16) & 0xff);
00093   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector»24)&0x0f));
00094   if(b->flags & B_DIRTY){
00095     outb(0x1f7, write_cmd);
00096     outsl(0x1f0, b->data, BSIZE/4);
00097   } else {
00098     outb(0x1f7, read_cmd);
00099   }
00100 }
00101
00102 // Interrupt handler.
00103 void
00104 ideintr(void)
00105 {
00106   struct buf *b;
00107
00108   // First queued buffer is the active request.
00109   acquire(&idelock);
00110
00111   if((b = idequeue) == 0){
00112     release(&idelock);
00113     return;
00114   }
00115   idequeue = b->qnext;
00116
00117   // Read data if needed.
00118   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
00119     insl(0x1f0, b->data, BSIZE/4);
00120
00121   // Wake process waiting for this buf.
00122   b->flags |= B_VALID;
00123   b->flags &= ~B_DIRTY;
00124   wakeup(b);
00125
00126   // Start disk on next buf in queue.
00127   if(idequeue != 0)
00128     idestart(idequeue);
00129
00130   release(&idelock);
00131 }
00132
00133 //PAGEBREAK!
00134 // Sync buf with disk.
00135 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
00136 // Else if B_VALID is not set, read buf from disk, set B_VALID.
```

```
00137 void
00138 iderw(struct buf *b)
00139 {
00140   struct buf **pp;
00141
00142   if(!holdingsleep(&b->lock))
00143     panic("iderw: buf not locked");
00144   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
00145     panic("iderw: nothing to do");
00146   if(b->dev != 0 && !havedisk1)
00147     panic("iderw: ide disk 1 not present");
00148
00149   acquire(&idelock);  //DOC:acquire-lock
00150
00151   // Append b to idequeue.
00152   b->qnext = 0;
00153   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC:insert-queue
00154     ;
00155   *pp = b;
00156
00157   // Start disk if necessary.
00158   if(idequeue == b)
00159     idestart(b);
00160
00161   // Wait for request to finish.
00162   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
00163     sleep(b, &idelock);
00164   }
00165
00166
00167   release(&idelock);
00168 }
```

5.71 ide.d File Reference

5.72 ide.d

Go to the documentation of this file.

```
00001 ide.o: ide.c /usr/include/stdc-predef.h types.h defs.h param.h \
00002  memlayout.h mmu.h proc.h x86.h traps.h spinlock.h sleeplock.h fs.h buf.h
```

5.73 init.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
```

Functions

- int main (void)

Variables

- char * argv [ ] = { "sh", 0 }

5.73.1 Function Documentation

5.73.1.1 main()

```
int main (
              void  )
```

Definition at line 11 of file init.c.

```
00012 {
00013   int pid, wpid;
00014
00015   if(open("console", O_RDWR) < 0){
00016     mknod("console", 1, 1);
00017     open("console", O_RDWR);
00018   }
00019   dup(0);  // stdout
00020   dup(0);  // stderr
00021
00022   for(;;){
00023     printf(1, "init: starting sh\n");
00024     pid = fork();
00025     if(pid < 0){
00026       printf(1, "init: fork failed\n");
00027       exit();
00028     }
00029     if(pid == 0){
00030       exec("sh", argv);
00031       printf(1, "init: exec sh failed\n");
00032       exit();
00033     }
00034     while((wpid=wait()) >= 0 && wpid != pid)
00035       printf(1, "zombie!\n");
00036   }
00037 }
```

5.73.2 Variable Documentation

5.73.2.1 argv

```
char* argv[] = { "sh", 0 }
```

Definition at line 8 of file init.c.

Referenced by exec(), main(), and sys_exec().

5.74 init.c

Go to the documentation of this file.

```
00001 // init: The initial user-level program
00002
00003 #include "types.h"
00004 #include "stat.h"
00005 #include "user.h"
00006 #include "fcntl.h"
00007
00008 char *argv[] = { "sh", 0 };
00009
00010 int
00011 main(void)
00012 {
00013   int pid, wpid;
00014
00015   if(open("console", O_RDWR) < 0){
00016     mknod("console", 1, 1);
00017     open("console", O_RDWR);
00018   }
```

```
00019   dup(0);  // stdout
00020   dup(0);  // stderr
00021
00022   for(;;){
00023     printf(1, "init: starting sh\n");
00024     pid = fork();
00025     if(pid < 0){
00026       printf(1, "init: fork failed\n");
00027       exit();
00028     }
00029     if(pid == 0){
00030       exec("sh", argv);
00031       printf(1, "init: exec sh failed\n");
00032       exit();
00033     }
00034     while((wpid=wait()) >= 0 && wpid != pid)
00035       printf(1, "zombie!\n");
00036   }
00037 }
```

5.75 init.d File Reference

5.76 init.d

Go to the documentation of this file.

```
00001 init.o: init.c /usr/include/stdc-predef.h types.h stat.h user.h fcntl.h
```

5.77 initcode.d File Reference

5.78 initcode.d

Go to the documentation of this file.

```
00001 initcode.o: initcode.S syscall.h traps.h
```

5.79 ioapic.c File Reference

```
#include "types.h"
#include "defs.h"
#include "traps.h"
```

Classes

- struct ioapic

Macros

- #define INT_ACTIVELOW 0x00002000
- #define INT_DISABLED 0x00010000
- #define INT_LEVEL 0x00008000
- #define INT_LOGICAL 0x00000800
- #define IOAPIC 0xFEC00000
- #define REG_ID 0x00
- #define REG_TABLE 0x10
- #define REG_VER 0x01

Functions

- void ioapicenable (int irq, int cpunum)
- void ioapicinit (void)
- static uint ioapicread (int reg)
- static void ioapicwrite (int reg, uint data)

Variables

- volatile struct ioapic * ioapic

5.79.1 Macro Definition Documentation

5.79.1.1 INT_ACTIVELOW

```
#define INT_ACTIVELOW 0x00002000
```

Definition at line 22 of file ioapic.c.

5.79.1.2 INT_DISABLED

```
#define INT_DISABLED 0x00010000
```

Definition at line 20 of file ioapic.c.

5.79.1.3 INT_LEVEL

```
#define INT_LEVEL 0x00008000
```

Definition at line 21 of file ioapic.c.

5.79.1.4 INT_LOGICAL

```
#define INT_LOGICAL 0x00000800
```

Definition at line 23 of file ioapic.c.

5.79.1.5 IOAPIC

```
#define IOAPIC 0xFEC00000
```

Definition at line 9 of file ioapic.c.

5.79.1.6 REG_ID

```
#define REG_ID 0x00
```

Definition at line 11 of file ioapic.c.

5.79.1.7 REG_TABLE

```
#define REG_TABLE 0x10
```

Definition at line 13 of file ioapic.c.

5.79.1.8 REG_VER

```
#define REG_VER 0x01
```

Definition at line 12 of file ioapic.c.

5.79.2 Function Documentation

5.79.2.1 ioapicenable()

```
void ioapicenable (
            int irq,
            int cpunum )
```

Definition at line 68 of file ioapic.c.

```
00069 {
00070   // Mark interrupt edge-triggered, active high,
00071   // enabled, and routed to the given cpunum,
00072   // which happens to be that cpu's APIC ID.
00073   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
00074   ioapicwrite(REG_TABLE+2*irq+1, cpunum « 24);
00075 }
```

Referenced by consoleinit(), ideinit(), and uartinit().

5.79.2.2 ioapicinit()

```
void ioapicinit (
             void  )
```

Definition at line 49 of file ioapic.c.

```
00050 {
00051   int i, id, maxintr;
00052
00053   ioapic = (volatile struct ioapic*)IOAPIC;
00054   maxintr = (ioapicread(REG_VER) » 16) & 0xFF;
00055   id = ioapicread(REG_ID) » 24;
00056   if(id != ioapicid)
00057     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
00058
00059   // Mark all interrupts edge-triggered, active high, disabled,
00060   // and not routed to any CPUs.
00061   for(i = 0; i <= maxintr; i++){
00062     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
00063     ioapicwrite(REG_TABLE+2*i+1, 0);
00064   }
00065 }
```

5.79.2.3 ioapicread()

```
static uint ioapicread (
             int reg )  [static]
```

Definition at line 35 of file ioapic.c.

```
00036 {
00037   ioapic->reg = reg;
00038   return ioapic->data;
00039 }
```

Referenced by ioapicinit().

5.79.2.4 ioapicwrite()

```
static void ioapicwrite (
             int reg,
             uint data )  [static]
```

Definition at line 42 of file ioapic.c.

```
00043 {
00044   ioapic->reg = reg;
00045   ioapic->data = data;
00046 }
```

Referenced by ioapicenable(), and ioapicinit().

5.79.3 Variable Documentation

5.79.3.1 ioapic

```
volatile struct ioapic* ioapic
```

Definition at line 25 of file ioapic.c.

5.80 ioapic.c

Go to the documentation of this file.

```
00001 // The I/O APIC manages hardware interrupts for an SMP system.
00002 // http://www.intel.com/design/chipsets/datashts/29056601.pdf
00003 // See also picirq.c.
00004
00005 #include "types.h"
00006 #include "defs.h"
00007 #include "traps.h"
00008
00009 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
00010
00011 #define REG_ID     0x00  // Register index: ID
00012 #define REG_VER    0x01  // Register index: version
00013 #define REG_TABLE  0x10  // Redirection table base
00014
00015 // The redirection table starts at REG_TABLE and uses
00016 // two registers to configure each interrupt.
00017 // The first (low) register in a pair contains configuration bits.
00018 // The second (high) register contains a bitmask telling which
00019 // CPUs can serve that interrupt.
00020 #define INT_DISABLED   0x00010000  // Interrupt disabled
00021 #define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
00022 #define INT_ACTIVELOW  0x00002000  // Active low (vs high)
00023 #define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)
00024
00025 volatile struct ioapic *ioapic;
00026
00027 // IO APIC MMIO structure: write reg, then read or write data.
00028 struct ioapic {
00029   uint reg;
00030   uint pad[3];
00031   uint data;
00032 };
00033
00034 static uint
00035 ioapicread(int reg)
00036 {
00037   ioapic->reg = reg;
00038   return ioapic->data;
00039 }
00040
00041 static void
00042 ioapicwrite(int reg, uint data)
00043 {
00044   ioapic->reg = reg;
00045   ioapic->data = data;
00046 }
00047
00048 void
00049 ioapicinit(void)
00050 {
00051   int i, id, maxintr;
00052
00053   ioapic = (volatile struct ioapic*)IOAPIC;
00054   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
00055   id = ioapicread(REG_ID) >> 24;
00056   if(id != ioapicid)
00057     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
00058
00059   // Mark all interrupts edge-triggered, active high, disabled,
00060   // and not routed to any CPUs.
00061   for(i = 0; i <= maxintr; i++){
00062     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
00063     ioapicwrite(REG_TABLE+2*i+1, 0);
00064   }
00065 }
00066
00067 void
00068 ioapicenable(int irq, int cpunum)
00069 {
```

```
00070   // Mark interrupt edge-triggered, active high,
00071   // enabled, and routed to the given cpunum,
00072   // which happens to be that cpu's APIC ID.
00073   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
00074   ioapicwrite(REG_TABLE+2*irq+1, cpunum « 24);
00075 }
```

5.81 ioapic.d File Reference

5.82 ioapic.d

Go to the documentation of this file.

```
00001 ioapic.o: ioapic.c /usr/include/stdc-predef.h types.h defs.h traps.h
```

5.83 kalloc.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "spinlock.h"
```

Classes

- struct run

Functions

- void freerange (void *vstart, void *vend)
- char * kalloc (void)
- void kfree (char *v)
- void kinit1 (void *vstart, void *vend)
- void kinit2 (void *vstart, void *vend)

Variables

- char end [ ]
- struct {
  struct run * freelist
  struct spinlock lock
  int use_lock
  } kmem

5.83.1 Function Documentation

5.83.1.1 freerange()

```
void freerange (
              void * vstart,
              void * vend )
```

Definition at line 47 of file kalloc.c.

```
00048 {
00049   char *p;
00050   p = (char*)PGROUNDUP((uint)vstart);
00051   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
00052     kfree(p);
00053 }
```

Referenced by kinit1(), and kinit2().

5.83.1.2 kalloc()

```
char * kalloc (
              void  )
```

Definition at line 83 of file kalloc.c.

```
00084 {
00085   struct run *r;
00086
00087   if(kmem.use_lock)
00088     acquire(&kmem.lock);
00089   r = kmem.freelist;
00090   if(r)
00091     kmem.freelist = r->next;
00092   if(kmem.use_lock)
00093     release(&kmem.lock);
00094   return (char*)r;
00095 }
```

Referenced by allocproc(), allocuvm(), copyuvm(), inituvm(), pipealloc(), setupkvm(), startothers(), and walkpgdir().

5.83.1.3 kfree()

```
void kfree (
              char * v )
```

Definition at line 60 of file kalloc.c.

```
00061 {
00062   struct run *r;
00063
00064   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
00065     panic("kfree");
00066
00067   // Fill with junk to catch dangling refs.
00068   memset(v, 1, PGSIZE);
00069
00070   if(kmem.use_lock)
00071     acquire(&kmem.lock);
00072   r = (struct run*)v;
00073   r->next = kmem.freelist;
00074   kmem.freelist = r;
00075   if(kmem.use_lock)
00076     release(&kmem.lock);
00077 }
```

Referenced by allocuvm(), copyuvm(), deallocuvm(), fork(), freerange(), freevm(), pipealloc(), pipeclose(), and wait().

5.83.1.4 kinit1()

```
void kinit1 (
              void * vstart,
              void * vend )
```

Definition at line 32 of file kalloc.c.

```
00033 {
00034   initlock(&kmem.lock, "kmem");
00035   kmem.use_lock = 0;
00036   freerange(vstart, vend);
00037 }
```

5.83.1.5 kinit2()

```
void kinit2 (
              void * vstart,
              void * vend )
```

Definition at line 40 of file kalloc.c.

```
00041 {
00042   freerange(vstart, vend);
00043   kmem.use_lock = 1;
00044 }
```

5.83.2 Variable Documentation

5.83.2.1 end

```
char end[]  [extern]
```

Referenced by kfree().

5.83.2.2 freelist

```
struct run* freelist
```

Definition at line 23 of file kalloc.c.

5.83.2.3

```
struct { ...  } kmem
```

Referenced by kalloc(), kfree(), kinit1(), and kinit2().

5.83.2.4 lock

```
struct spinlock lock
```

Definition at line 21 of file kalloc.c.

5.83.2.5 use_lock

```
int use_lock
```

Definition at line 22 of file kalloc.c.

5.84 kalloc.c

Go to the documentation of this file.

```
00001 // Physical memory allocator, intended to allocate
00002 // memory for user processes, kernel stacks, page table pages,
00003 // and pipe buffers. Allocates 4096-byte pages.
00004
00005 #include "types.h"
00006 #include "defs.h"
00007 #include "param.h"
00008 #include "memlayout.h"
00009 #include "mmu.h"
00010 #include "spinlock.h"
00011
00012 void freerange(void *vstart, void *vend);
00013 extern char end[]; // first address after kernel loaded from ELF file
00014                    // defined by the kernel linker script in kernel.ld
00015
00016 struct run {
00017   struct run *next;
00018 };
00019
00020 struct {
00021   struct spinlock lock;
00022   int use_lock;
00023   struct run *freelist;
00024 } kmem;
00025
00026 // Initialization happens in two phases.
00027 // 1. main() calls kinit1() while still using entrypgdir to place just
00028 // the pages mapped by entrypgdir on free list.
00029 // 2. main() calls kinit2() with the rest of the physical pages
00030 // after installing a full page table that maps them on all cores.
00031 void
00032 kinit1(void *vstart, void *vend)
00033 {
00034   initlock(&kmem.lock, "kmem");
00035   kmem.use_lock = 0;
00036   freerange(vstart, vend);
00037 }
00038
00039 void
00040 kinit2(void *vstart, void *vend)
00041 {
00042   freerange(vstart, vend);
00043   kmem.use_lock = 1;
00044 }
00045
00046 void
00047 freerange(void *vstart, void *vend)
00048 {
00049   char *p;
00050   p = (char*)PGROUNDUP((uint)vstart);
00051   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
00052     kfree(p);
00053 }
00054 //PAGEBREAK: 21
00055 // Free the page of physical memory pointed at by v,
```

```
00056 // which normally should have been returned by a
00057 // call to kalloc().  (The exception is when
00058 // initializing the allocator; see kinit above.)
00059 void
00060 kfree(char *v)
00061 {
00062   struct run *r;
00063
00064   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
00065     panic("kfree");
00066
00067   // Fill with junk to catch dangling refs.
00068   memset(v, 1, PGSIZE);
00069
00070   if(kmem.use_lock)
00071     acquire(&kmem.lock);
00072   r = (struct run*)v;
00073   r->next = kmem.freelist;
00074   kmem.freelist = r;
00075   if(kmem.use_lock)
00076     release(&kmem.lock);
00077 }
00078
00079 // Allocate one 4096-byte page of physical memory.
00080 // Returns a pointer that the kernel can use.
00081 // Returns 0 if the memory cannot be allocated.
00082 char*
00083 kalloc(void)
00084 {
00085   struct run *r;
00086
00087   if(kmem.use_lock)
00088     acquire(&kmem.lock);
00089   r = kmem.freelist;
00090   if(r)
00091     kmem.freelist = r->next;
00092   if(kmem.use_lock)
00093     release(&kmem.lock);
00094   return (char*)r;
00095 }
00096
```

5.85 kalloc.d File Reference

5.86 kalloc.d

Go to the documentation of this file.

```
00001 kalloc.o: kalloc.c /usr/include/stdc-predef.h types.h defs.h param.h \
00002  memlayout.h mmu.h spinlock.h
```

5.87 kbd.c File Reference

```
#include "types.h"
#include "x86.h"
#include "defs.h"
#include "kbd.h"
```

Functions

- int kbdgetc (void)
- void kbdintr (void)

5.87.1 Function Documentation

5.87.1.1 kbdgetc()

```
int kbdgetc (
            void  )
```

Definition at line 7 of file kbd.c.

```
00008 {
00009   static uint shift;
00010   static uchar *charcode[4] = {
00011     normalmap, shiftmap, ctlmap, ctlmap
00012   };
00013   uint st, data, c;
00014
00015   st = inb(KBSTATP);
00016   if((st & KBS_DIB) == 0)
00017     return -1;
00018   data = inb(KBDATAP);
00019
00020   if(data == 0xE0){
00021     shift |= E0ESC;
00022     return 0;
00023   } else if(data & 0x80){
00024     // Key released
00025     data = (shift & E0ESC ? data : data & 0x7F);
00026     shift &= ~(shiftcode[data] | E0ESC);
00027     return 0;
00028   } else if(shift & E0ESC){
00029     // Last character was an E0 escape; or with 0x80
00030     data |= 0x80;
00031     shift &= ~E0ESC;
00032   }
00033
00034   shift |= shiftcode[data];
00035   shift ^= togglecode[data];
00036   c = charcode[shift & (CTL | SHIFT)][data];
00037   if(shift & CAPSLOCK){
00038     if('a' <= c && c <= 'z')
00039       c += 'A' - 'a';
00040     else if('A' <= c && c <= 'Z')
00041       c += 'a' - 'A';
00042   }
00043   return c;
00044 }
```

Referenced by kbdintr().

5.87.1.2 kbdintr()

```
void kbdintr (
            void  )
```

Definition at line 47 of file kbd.c.

```
00048 {
00049   consoleintr(kbdgetc);
00050 }
```

Referenced by trap().

5.88 kbd.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "x86.h"
00003 #include "defs.h"
00004 #include "kbd.h"
00005
00006 int
00007 kbdgetc(void)
00008 {
00009   static uint shift;
00010   static uchar *charcode[4] = {
00011     normalmap, shiftmap, ctlmap, ctlmap
00012   };
00013   uint st, data, c;
00014
00015   st = inb(KBSTATP);
00016   if((st & KBS_DIB) == 0)
00017     return -1;
00018   data = inb(KBDATAP);
00019
00020   if(data == 0xE0){
00021     shift |= E0ESC;
00022     return 0;
00023   } else if(data & 0x80){
00024     // Key released
00025     data = (shift & E0ESC ? data : data & 0x7F);
00026     shift &= ~(shiftcode[data] | E0ESC);
00027     return 0;
00028   } else if(shift & E0ESC){
00029     // Last character was an E0 escape; or with 0x80
00030     data |= 0x80;
00031     shift &= ~E0ESC;
00032   }
00033
00034   shift |= shiftcode[data];
00035   shift ^= togglecode[data];
00036   c = charcode[shift & (CTL | SHIFT)][data];
00037   if(shift & CAPSLOCK){
00038     if('a' <= c && c <= 'z')
00039       c += 'A' - 'a';
00040     else if('A' <= c && c <= 'Z')
00041       c += 'a' - 'A';
00042   }
00043   return c;
00044 }
00045
00046 void
00047 kbdintr(void)
00048 {
00049   consoleintr(kbdgetc);
00050 }
```

5.89 kbd.d File Reference

5.90 kbd.d

Go to the documentation of this file.

```
00001 kbd.o: kbd.c /usr/include/stdc-predef.h types.h x86.h defs.h kbd.h
```

5.91 kbd.h File Reference

Macros

- #define ALT (1<<2)
- #define C(x) (x - '@')

- #define CAPSLOCK (1<<3)
- #define CTL (1<<1)
- #define E0ESC (1<<6)
- #define KBDATAP 0x60
- #define KBS_DIB 0x01
- #define KBSTATP 0x64
- #define KEY_DEL 0xE9
- #define KEY_DN 0xE3
- #define KEY_END 0xE1
- #define KEY_HOME 0xE0
- #define KEY_INS 0xE8
- #define KEY_LF 0xE4
- #define KEY_PGDN 0xE7
- #define KEY_PGUP 0xE6
- #define KEY_RT 0xE5
- #define KEY_UP 0xE2
- #define NO 0
- #define NUMLOCK (1<<4)
- #define SCROLLLOCK (1<<5)
- #define SHIFT (1<<0)

Variables

- static uchar ctlmap [256]
- static uchar normalmap [256]
- static uchar shiftcode [256]
- static uchar shiftmap [256]
- static uchar togglecode [256]

5.91.1 Macro Definition Documentation

5.91.1.1 ALT

```
#define ALT (1<<2)
```

Definition at line 11 of file kbd.h.

5.91.1.2 C

```
#define C(
              x ) (x - '@')
```

Definition at line 32 of file kbd.h.

5.91.1.3 CAPSLOCK

```
#define CAPSLOCK (1<<3)
```

Definition at line 13 of file kbd.h.

5.91.1.4 CTL

```
#define CTL (1<<1)
```

Definition at line 10 of file kbd.h.

5.91.1.5 E0ESC

```
#define E0ESC (1<<6)
```

Definition at line 17 of file kbd.h.

5.91.1.6 KBDATAP

```
#define KBDATAP 0x60
```

Definition at line 5 of file kbd.h.

5.91.1.7 KBS_DIB

```
#define KBS_DIB 0x01
```

Definition at line 4 of file kbd.h.

5.91.1.8 KBSTATP

```
#define KBSTATP 0x64
```

Definition at line 3 of file kbd.h.

5.91.1.9 KEY_DEL

```
#define KEY_DEL 0xE9
```

Definition at line 29 of file kbd.h.

5.91.1.10 KEY_DN

```
#define KEY_DN 0xE3
```

Definition at line 23 of file kbd.h.

5.91.1.11 KEY_END

```
#define KEY_END 0xE1
```

Definition at line 21 of file kbd.h.

5.91.1.12 KEY_HOME

```
#define KEY_HOME 0xE0
```

Definition at line 20 of file kbd.h.

5.91.1.13 KEY_INS

```
#define KEY_INS 0xE8
```

Definition at line 28 of file kbd.h.

5.91.1.14 KEY_LF

```
#define KEY_LF 0xE4
```

Definition at line 24 of file kbd.h.

5.91.1.15 KEY_PGDN

```
#define KEY_PGDN 0xE7
```

Definition at line 27 of file kbd.h.

5.91.1.16 KEY_PGUP

```
#define KEY_PGUP 0xE6
```

Definition at line 26 of file kbd.h.

5.91.1.17 KEY_RT

```
#define KEY_RT 0xE5
```

Definition at line 25 of file kbd.h.

5.91.1.18 KEY_UP

```
#define KEY_UP 0xE2
```

Definition at line 22 of file kbd.h.

5.91.1.19 NO

```
#define NO 0
```

Definition at line 7 of file kbd.h.

5.91.1.20 NUMLOCK

```
#define NUMLOCK (1<<4)
```

Definition at line 14 of file kbd.h.

5.91.1.21 SCROLLLOCK

```
#define SCROLLLOCK (1<<5)
```

Definition at line 15 of file kbd.h.

5.91.1.22 SHIFT

```
#define SHIFT (1<<0)
```

Definition at line 9 of file kbd.h.

5.91.2 Variable Documentation

5.91.2.1 ctlmap

```
uchar ctlmap[256]  [static]
```

Initial value:

```
=
{
  NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
  NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
  C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
  C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
  C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
  NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
  C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
  [0x9C] '\r',
  [0xB5] C('/'),
  [0xC8] KEY_UP,     [0xD0] KEY_DN,
  [0xC9] KEY_PGUP,   [0xD1] KEY_PGDN,
  [0xCB] KEY_LF,     [0xCD] KEY_RT,
  [0x97] KEY_HOME,   [0xCF] KEY_END,
  [0xD2] KEY_INS,    [0xD3] KEY_DEL
}
```

Definition at line 95 of file kbd.h.

Referenced by kbdgetc().

5.91.2.2 normalmap

```
uchar normalmap[256]  [static]
```

Initial value:

```
=
{
  NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',
  '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
  'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',
  'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
  'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',
  '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
  'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',
  NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
  NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',
  '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
  '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,
  [0x9C] '\n',
  [0xB5] '/',
  [0xC8] KEY_UP,    [0xD0] KEY_DN,
  [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
  [0xCB] KEY_LF,    [0xCD] KEY_RT,
  [0x97] KEY_HOME,  [0xCF] KEY_END,
  [0xD2] KEY_INS,   [0xD3] KEY_DEL
}
```

Definition at line 51 of file kbd.h.

Referenced by kbdgetc().

5.91.2.3 shiftcode

```
uchar shiftcode[256]  [static]
```

Initial value:

```
=
{
  [0x1D] CTL,
  [0x2A] SHIFT,
  [0x36] SHIFT,
  [0x38] ALT,
  [0x9D] CTL,
  [0xB8] ALT
}
```

Definition at line 34 of file kbd.h.

Referenced by kbdgetc().

5.91.2.4 shiftmap

```
uchar shiftmap[256]  [static]
```

Initial value:

```
=
{
  NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',
  '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
  'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',
  'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
  'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',
  '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
  'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',
```

```
  NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
  NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',
  '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
  '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,
  [0x9C] '\n',
  [0xB5] '/',
  [0xC8] KEY_UP,    [0xD0] KEY_DN,
  [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
  [0xCB] KEY_LF,    [0xCD] KEY_RT,
  [0x97] KEY_HOME,  [0xCF] KEY_END,
  [0xD2] KEY_INS,   [0xD3] KEY_DEL
}
```

Definition at line 73 of file kbd.h.

Referenced by kbdgetc().

5.91.2.5 togglecode

```
uchar togglecode[256]  [static]
```

Initial value:

```
=
{
  [0x3A] CAPSLOCK,
  [0x45] NUMLOCK,
  [0x46] SCROLLLOCK
}
```

Definition at line 44 of file kbd.h.

Referenced by kbdgetc().

5.92 kbd.h

Go to the documentation of this file.

```
00001 // PC keyboard interface constants
00002
00003 #define KBSTATP         0x64    // kbd controller status port(I)
00004 #define KBS_DIB         0x01    // kbd data in buffer
00005 #define KBDATAP         0x60    // kbd data port(I)
00006
00007 #define NO              0
00008
00009 #define SHIFT           (1<<0)
00010 #define CTL             (1<<1)
00011 #define ALT             (1<<2)
00012
00013 #define CAPSLOCK        (1<<3)
00014 #define NUMLOCK         (1<<4)
00015 #define SCROLLLOCK      (1<<5)
00016
00017 #define E0ESC           (1<<6)
00018
00019 // Special keycodes
00020 #define KEY_HOME        0xE0
00021 #define KEY_END         0xE1
00022 #define KEY_UP          0xE2
00023 #define KEY_DN          0xE3
00024 #define KEY_LF          0xE4
00025 #define KEY_RT          0xE5
00026 #define KEY_PGUP        0xE6
00027 #define KEY_PGDN        0xE7
00028 #define KEY_INS         0xE8
00029 #define KEY_DEL         0xE9
00030
00031 // C('A') == Control-A
00032 #define C(x) (x - '@')
00033
```

```
00034 static uchar shiftcode[256] =
00035 {
00036   [0x1D] CTL,
00037   [0x2A] SHIFT,
00038   [0x36] SHIFT,
00039   [0x38] ALT,
00040   [0x9D] CTL,
00041   [0xB8] ALT
00042 };
00043
00044 static uchar togglecode[256] =
00045 {
00046   [0x3A] CAPSLOCK,
00047   [0x45] NUMLOCK,
00048   [0x46] SCROLLLOCK
00049 };
00050
00051 static uchar normalmap[256] =
00052 {
00053   NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
00054   '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
00055   'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
00056   'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
00057   'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
00058   '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
00059   'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
00060   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
00061   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
00062   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
00063   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
00064   [0x9C] '\n',      // KP_Enter
00065   [0xB5] '/',       // KP_Div
00066   [0xC8] KEY_UP,    [0xD0] KEY_DN,
00067   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
00068   [0xCB] KEY_LF,    [0xCD] KEY_RT,
00069   [0x97] KEY_HOME,  [0xCF] KEY_END,
00070   [0xD2] KEY_INS,   [0xD3] KEY_DEL
00071 };
00072
00073 static uchar shiftmap[256] =
00074 {
00075   NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
00076   '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
00077   'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
00078   'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
00079   'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
00080   '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
00081   'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
00082   NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
00083   NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
00084   '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
00085   '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
00086   [0x9C] '\n',      // KP_Enter
00087   [0xB5] '/',       // KP_Div
00088   [0xC8] KEY_UP,    [0xD0] KEY_DN,
00089   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
00090   [0xCB] KEY_LF,    [0xCD] KEY_RT,
00091   [0x97] KEY_HOME,  [0xCF] KEY_END,
00092   [0xD2] KEY_INS,   [0xD3] KEY_DEL
00093 };
00094
00095 static uchar ctlmap[256] =
00096 {
00097   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
00098   NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
00099   C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
00100   C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
00101   C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
00102   NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
00103   C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
00104   [0x9C] '\r',      // KP_Enter
00105   [0xB5] C('/'),    // KP_Div
00106   [0xC8] KEY_UP,    [0xD0] KEY_DN,
00107   [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
00108   [0xCB] KEY_LF,    [0xCD] KEY_RT,
00109   [0x97] KEY_HOME,  [0xCF] KEY_END,
00110   [0xD2] KEY_INS,   [0xD3] KEY_DEL
00111 };
00112
```

5.93 kill.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- int main (int argc, char **argv)

5.93.1 Function Documentation

5.93.1.1 main()

```
int main (
            int argc,
            char ** argv )
```

Definition at line 6 of file kill.c.

```
00007 {
00008   int i;
00009
00010   if(argc < 2){
00011     printf(2, "usage: kill pid...\n");
00012     exit();
00013   }
00014   for(i=1; i<argc; i++)
00015     kill(atoi(argv[i]));
00016   exit();
00017 }
```

5.94 kill.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004
00005 int
00006 main(int argc, char **argv)
00007 {
00008   int i;
00009
00010   if(argc < 2){
00011     printf(2, "usage: kill pid...\n");
00012     exit();
00013   }
00014   for(i=1; i<argc; i++)
00015     kill(atoi(argv[i]));
00016   exit();
00017 }
```

5.95 kill.d File Reference

5.96 kill.d

Go to the documentation of this file.

```
00001 kill.o: kill.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.97 lapic.c File Reference

```
#include "param.h"
#include "types.h"
#include "defs.h"
#include "date.h"
#include "memlayout.h"
#include "traps.h"
#include "mmu.h"
#include "x86.h"
```

Macros

- #define ASSERT 0x00004000
- #define BCAST 0x00080000
- #define BUSY 0x00001000
- #define CMOS_PORT 0x70
- #define CMOS_RETURN 0x71
- #define CMOS_STATA 0x0a
- #define CMOS_STATB 0x0b
- #define CMOS_UIP (1 << 7)
- #define CONV(x) (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
- #define DAY 0x07
- #define DEASSERT 0x00000000
- #define DELIVS 0x00001000
- #define ENABLE 0x00000100
- #define EOI (0x00B0/4)
- #define ERROR (0x0370/4)
- #define ESR (0x0280/4)
- #define FIXED 0x00000000
- #define HOURS 0x04
- #define ICRHI (0x0310/4)
- #define ICRLO (0x0300/4)
- #define ID (0x0020/4)
- #define INIT 0x00000500
- #define LEVEL 0x00008000
- #define LINT0 (0x0350/4)
- #define LINT1 (0x0360/4)
- #define MASKED 0x00010000
- #define MINS 0x02
- #define MONTH 0x08
- #define PCINT (0x0340/4)

- #define PERIODIC 0x00020000
- #define SECS 0x00
- #define STARTUP 0x00000600
- #define SVR (0x00F0/4)
- #define TCCR (0x0390/4)
- #define TDCR (0x03E0/4)
- #define TICR (0x0380/4)
- #define TIMER (0x0320/4)
- #define TPR (0x0080/4)
- #define VER (0x0030/4)
- #define X1 0x0000000B
- #define YEAR 0x09

Functions

- static uint cmos_read (uint reg)
- void cmostime (struct rtcdate *r)
- static void fill_rtcdate (struct rtcdate *r)
- void lapiceoi (void)
- int lapicid (void)
- void lapicinit (void)
- void lapicstartap (uchar apicid, uint addr)
- static void lapicw (int index, int value)
- void microdelay (int us)

Variables

- volatile uint * lapic

5.97.1 Macro Definition Documentation

5.97.1.1 ASSERT

```
#define ASSERT 0x00004000
```

Definition at line 25 of file lapic.c.

5.97.1.2 BCAST

```
#define BCAST 0x00080000
```

Definition at line 28 of file lapic.c.

5.97.1.3 BUSY

```
#define BUSY 0x00001000
```

Definition at line 29 of file lapic.c.

5.97.1.4 CMOS_PORT

```
#define CMOS_PORT 0x70
```

Definition at line 123 of file lapic.c.

5.97.1.5 CMOS_RETURN

```
#define CMOS_RETURN 0x71
```

Definition at line 124 of file lapic.c.

5.97.1.6 CMOS_STATA

```
#define CMOS_STATA 0x0a
```

Definition at line 163 of file lapic.c.

5.97.1.7 CMOS_STATB

```
#define CMOS_STATB 0x0b
```

Definition at line 164 of file lapic.c.

5.97.1.8 CMOS_UIP

```
#define CMOS_UIP (1 << 7)
```

Definition at line 165 of file lapic.c.

5.97.1.9 CONV

```
#define CONV(
              x ) (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
```

5.97.1.10 DAY

```
#define DAY 0x07
```

Definition at line 170 of file lapic.c.

5.97.1.11 DEASSERT

```
#define DEASSERT 0x00000000
```

Definition at line 26 of file lapic.c.

5.97.1.12 DELIVS

```
#define DELIVS 0x00001000
```

Definition at line 24 of file lapic.c.

5.97.1.13 ENABLE

```
#define ENABLE 0x00000100
```

Definition at line 19 of file lapic.c.

5.97.1.14 EOI

```
#define EOI (0x00B0/4)
```

Definition at line 17 of file lapic.c.

5.97.1.15 ERROR

```
#define ERROR (0x0370/4)
```

Definition at line 38 of file lapic.c.

5.97.1.16 ESR

```
#define ESR (0x0280/4)
```

Definition at line 20 of file lapic.c.

5.97.1.17 FIXED

```
#define FIXED 0x00000000
```

Definition at line 30 of file lapic.c.

5.97.1.18 HOURS

```
#define HOURS 0x04
```

Definition at line 169 of file lapic.c.

5.97.1.19 ICRHI

```
#define ICRHI (0x0310/4)
```

Definition at line 31 of file lapic.c.

5.97.1.20 ICRLO

```
#define ICRLO (0x0300/4)
```

Definition at line 21 of file lapic.c.

5.97.1.21 ID

```
#define ID (0x0020/4)
```

Definition at line 14 of file lapic.c.

5.97.1.22 INIT

```
#define INIT 0x00000500
```

Definition at line 22 of file lapic.c.

5.97.1.23 LEVEL

```
#define LEVEL 0x00008000
```

Definition at line 27 of file lapic.c.

5.97.1.24 LINT0

```
#define LINT0 (0x0350/4)
```

Definition at line 36 of file lapic.c.

5.97.1.25 LINT1

```
#define LINT1 (0x0360/4)
```

Definition at line 37 of file lapic.c.

5.97.1.26 MASKED

```
#define MASKED 0x00010000
```

Definition at line 39 of file lapic.c.

5.97.1.27 MINS

```
#define MINS 0x02
```

Definition at line 168 of file lapic.c.

5.97.1.28 MONTH

```
#define MONTH 0x08
```

Definition at line 171 of file lapic.c.

5.97.1.29 PCINT

```
#define PCINT (0x0340/4)
```

Definition at line 35 of file lapic.c.

5.97.1.30 PERIODIC

```
#define PERIODIC 0x00020000
```

Definition at line 34 of file lapic.c.

5.97.1.31 SECS

```
#define SECS 0x00
```

Definition at line 167 of file lapic.c.

5.97.1.32 STARTUP

```
#define STARTUP 0x00000600
```

Definition at line 23 of file lapic.c.

5.97.1.33 SVR

```
#define SVR (0x00F0/4)
```

Definition at line 18 of file lapic.c.

5.97.1.34 TCCR

```
#define TCCR (0x0390/4)
```

Definition at line 41 of file lapic.c.

5.97.1.35 TDCR

```
#define TDCR (0x03E0/4)
```

Definition at line 42 of file lapic.c.

5.97.1.36 TICR

```
#define TICR (0x0380/4)
```

Definition at line 40 of file lapic.c.

5.97.1.37 TIMER

```
#define TIMER (0x0320/4)
```

Definition at line 32 of file lapic.c.

5.97.1.38 TPR

```
#define TPR (0x0080/4)
```

Definition at line 16 of file lapic.c.

5.97.1.39 VER

```
#define VER (0x0030/4)
```

Definition at line 15 of file lapic.c.

5.97.1.40 X1

```
#define X1 0x0000000B
```

Definition at line 33 of file lapic.c.

5.97.1.41 YEAR

```
#define YEAR 0x09
```

Definition at line 172 of file lapic.c.

5.97.2 Function Documentation

5.97.2.1 cmos_read()

```
static uint cmos_read (
             uint reg )  [static]
```

Definition at line 175 of file lapic.c.

```
00176 {
00177   outb(CMOS_PORT,  reg);
00178   microdelay(200);
00179
00180   return inb(CMOS_RETURN);
00181 }
```

Referenced by cmostime(), and fill_rtcdate().

5.97.2.2 cmostime()

```
void cmostime (
             struct rtcdate * r )
```

Definition at line 196 of file lapic.c.

```
00197 {
00198   struct rtcdate t1, t2;
00199   int sb, bcd;
00200
00201   sb = cmos_read(CMOS_STATB);
00202
00203   bcd = (sb & (1 « 2)) == 0;
00204
00205   // make sure CMOS doesn't modify time while we read it
00206   for(;;) {
00207     fill_rtcdate(&t1);
00208     if(cmos_read(CMOS_STATA) & CMOS_UIP)
00209         continue;
00210     fill_rtcdate(&t2);
00211     if(memcmp(&t1, &t2, sizeof(t1)) == 0)
00212       break;
00213   }
00214
00215   // convert
00216   if(bcd) {
00217 #define    CONV(x)     (t1.x = ((t1.x » 4) * 10) + (t1.x & 0xf))
00218     CONV(second);
00219     CONV(minute);
00220     CONV(hour  );
00221     CONV(day   );
00222     CONV(month );
00223     CONV(year  );
00224 #undef     CONV
00225   }
00226
00227   *r = t1;
00228   r->year += 2000;
00229 }
```

5.97.2.3 fill_rtcdate()

```
static void fill_rtcdate (
             struct rtcdate * r )  [static]
```

Definition at line 184 of file lapic.c.

```
00185 {
00186   r->second = cmos_read(SECS);
00187   r->minute = cmos_read(MINS);
00188   r->hour   = cmos_read(HOURS);
00189   r->day    = cmos_read(DAY);
00190   r->month  = cmos_read(MONTH);
00191   r->year   = cmos_read(YEAR);
00192 }
```

Referenced by cmostime().

5.97.2.4 lapiceoi()

```
void lapiceoi (
             void  )
```

Definition at line 110 of file lapic.c.

```
00111 {
00112   if(lapic)
00113     lapicw(EOI, 0);
00114 }
```

Referenced by trap().

5.97.2.5 lapicid()

```
int lapicid (
             void  )
```

Definition at line 101 of file lapic.c.

```
00102 {
00103   if (!lapic)
00104     return 0;
00105   return lapic[ID] » 24;
00106 }
```

Referenced by mycpu(), and panic().

5.97.2.6 lapicinit()

```
void lapicinit (
             void  )
```

Definition at line 55 of file lapic.c.

```
00056 {
00057   if(!lapic)
00058     return;
00059
00060   // Enable local APIC; set spurious interrupt vector.
00061   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
00062
00063   // The timer repeatedly counts down at bus frequency
00064   // from lapic[TICR] and then issues an interrupt.
00065   // If xv6 cared more about precise timekeeping,
00066   // TICR would be calibrated using an external time source.
00067   lapicw(TDCR, X1);
00068   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
00069   lapicw(TICR, 10000000);
00070
00071   // Disable logical interrupt lines.
00072   lapicw(LINT0, MASKED);
00073   lapicw(LINT1, MASKED);
00074
00075   // Disable performance counter overflow interrupts
00076   // on machines that provide that interrupt entry.
00077   if(((lapic[VER]»16) & 0xFF) >= 4)
00078     lapicw(PCINT, MASKED);
00079
00080   // Map error interrupt to IRQ_ERROR.
00081   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
00082
00083   // Clear error status register (requires back-to-back writes).
00084   lapicw(ESR, 0);
00085   lapicw(ESR, 0);
00086
00087   // Ack any outstanding interrupts.
00088   lapicw(EOI, 0);
00089
00090   // Send an Init Level De-Assert to synchronise arbitration ID's.
00091   lapicw(ICRHI, 0);
00092   lapicw(ICRLO, BCAST | INIT | LEVEL);
00093   while(lapic[ICRLO] & DELIVS)
00094     ;
00095
00096   // Enable interrupts on the APIC (but not on the processor).
00097   lapicw(TPR, 0);
00098 }
```

Referenced by mpenter().

5.97.2.7 lapicstartap()

```
void lapicstartap (
             uchar apicid,
             uint addr )
```

Definition at line 129 of file lapic.c.

```
00130 {
00131   int i;
00132   ushort *wrv;
00133
00134   // "The BSP must initialize CMOS shutdown code to 0AH
00135   // and the warm reset vector (DWORD based at 40:67) to point at
00136   // the AP startup code prior to the [universal startup algorithm]."
00137   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
00138   outb(CMOS_PORT+1, 0x0A);
00139   wrv = (ushort*)P2V((0x40<<4 | 0x67));  // Warm reset vector
00140   wrv[0] = 0;
00141   wrv[1] = addr >> 4;
00142
00143   // "Universal startup algorithm."
00144   // Send INIT (level-triggered) interrupt to reset other CPU.
00145   lapicw(ICRHI, apicid<<24);
00146   lapicw(ICRLO, INIT | LEVEL | ASSERT);
00147   microdelay(200);
00148   lapicw(ICRLO, INIT | LEVEL);
00149   microdelay(100);    // should be 10ms, but too slow in Bochs!
00150
00151   // Send startup IPI (twice!) to enter code.
00152   // Regular hardware is supposed to only accept a STARTUP
00153   // when it is in the halted state due to an INIT.  So the second
00154   // should be ignored, but it is part of the official Intel algorithm.
00155   // Bochs complains about the second one.  Too bad for Bochs.
00156   for(i = 0; i < 2; i++){
00157     lapicw(ICRHI, apicid<<24);
00158     lapicw(ICRLO, STARTUP | (addr>>12));
00159     microdelay(200);
00160   }
00161 }
```

Referenced by startothers().

5.97.2.8 lapicw()

```
static void lapicw (
             int index,
             int value )  [static]
```

Definition at line 48 of file lapic.c.

```
00049 {
00050   lapic[index] = value;
00051   lapic[ID];  // wait for write to finish, by reading
00052 }
```

Referenced by lapiceoi(), lapicinit(), and lapicstartap().

5.97.2.9 microdelay()

```
void microdelay (
             int us )
```

Definition at line 119 of file lapic.c.

```
00120 {
00121 }
```

Referenced by cmos_read(), lapicstartap(), and uartputc().

5.97.3 Variable Documentation

5.97.3.1 lapic

```
volatile uint* lapic
```

Definition at line 44 of file lapic.c.

Referenced by lapiceoi(), lapicid(), lapicinit(), lapicw(), and mpinit().

5.98 lapic.c

Go to the documentation of this file.

```
00001 // The local APIC manages internal (non-I/O) interrupts.
00002 // See Chapter 8 & Appendix C of Intel processor manual volume 3.
00003
00004 #include "param.h"
00005 #include "types.h"
00006 #include "defs.h"
00007 #include "date.h"
00008 #include "memlayout.h"
00009 #include "traps.h"
00010 #include "mmu.h"
00011 #include "x86.h"
00012
00013 // Local APIC registers, divided by 4 for use as uint[] indices.
00014 #define ID      (0x0020/4)   // ID
00015 #define VER     (0x0030/4)   // Version
00016 #define TPR     (0x0080/4)   // Task Priority
00017 #define EOI     (0x00B0/4)   // EOI
00018 #define SVR     (0x00F0/4)   // Spurious Interrupt Vector
00019   #define ENABLE     0x00000100   // Unit Enable
00020 #define ESR     (0x0280/4)   // Error Status
00021 #define ICRLO   (0x0300/4)   // Interrupt Command
00022   #define INIT       0x00000500   // INIT/RESET
00023   #define STARTUP    0x00000600   // Startup IPI
00024   #define DELIVS     0x00001000   // Delivery status
00025   #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
00026   #define DEASSERT   0x00000000
00027   #define LEVEL      0x00008000   // Level triggered
00028   #define BCAST      0x00080000   // Send to all APICs, including self.
00029   #define BUSY       0x00001000
00030   #define FIXED      0x00000000
00031 #define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
00032 #define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
00033   #define X1         0x0000000B   // divide counts by 1
00034   #define PERIODIC   0x00020000   // Periodic
00035 #define PCINT   (0x0340/4)   // Performance Counter LVT
00036 #define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
00037 #define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
00038 #define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
00039   #define MASKED     0x00010000   // Interrupt masked
00040 #define TICR    (0x0380/4)   // Timer Initial Count
00041 #define TCCR    (0x0390/4)   // Timer Current Count
00042 #define TDCR    (0x03E0/4)   // Timer Divide Configuration
00043
00044 volatile uint *lapic;  // Initialized in mp.c
00045
00046 //PAGEBREAK!
00047 static void
00048 lapicw(int index, int value)
00049 {
00050   lapic[index] = value;
00051   lapic[ID];  // wait for write to finish, by reading
00052 }
00053
00054 void
00055 lapicinit(void)
00056 {
00057   if(!lapic)
00058     return;
```

```
00059
00060   // Enable local APIC; set spurious interrupt vector.
00061   lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
00062
00063   // The timer repeatedly counts down at bus frequency
00064   // from lapic[TICR] and then issues an interrupt.
00065   // If xv6 cared more about precise timekeeping,
00066   // TICR would be calibrated using an external time source.
00067   lapicw(TDCR, X1);
00068   lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
00069   lapicw(TICR, 10000000);
00070
00071   // Disable logical interrupt lines.
00072   lapicw(LINT0, MASKED);
00073   lapicw(LINT1, MASKED);
00074
00075   // Disable performance counter overflow interrupts
00076   // on machines that provide that interrupt entry.
00077   if(((lapic[VER]»16) & 0xFF) >= 4)
00078     lapicw(PCINT, MASKED);
00079
00080   // Map error interrupt to IRQ_ERROR.
00081   lapicw(ERROR, T_IRQ0 + IRQ_ERROR);
00082
00083   // Clear error status register (requires back-to-back writes).
00084   lapicw(ESR, 0);
00085   lapicw(ESR, 0);
00086
00087   // Ack any outstanding interrupts.
00088   lapicw(EOI, 0);
00089
00090   // Send an Init Level De-Assert to synchronise arbitration ID's.
00091   lapicw(ICRHI, 0);
00092   lapicw(ICRLO, BCAST | INIT | LEVEL);
00093   while(lapic[ICRLO] & DELIVS)
00094     ;
00095
00096   // Enable interrupts on the APIC (but not on the processor).
00097   lapicw(TPR, 0);
00098 }
00099
00100 int
00101 lapicid(void)
00102 {
00103   if (!lapic)
00104     return 0;
00105   return lapic[ID] » 24;
00106 }
00107
00108 // Acknowledge interrupt.
00109 void
00110 lapiceoi(void)
00111 {
00112   if(lapic)
00113     lapicw(EOI, 0);
00114 }
00115
00116 // Spin for a given number of microseconds.
00117 // On real hardware would want to tune this dynamically.
00118 void
00119 microdelay(int us)
00120 {
00121 }
00122
00123 #define CMOS_PORT    0x70
00124 #define CMOS_RETURN  0x71
00125
00126 // Start additional processor running entry code at addr.
00127 // See Appendix B of MultiProcessor Specification.
00128 void
00129 lapicstartap(uchar apicid, uint addr)
00130 {
00131   int i;
00132   ushort *wrv;
00133
00134   // "The BSP must initialize CMOS shutdown code to 0AH
00135   // and the warm reset vector (DWORD based at 40:67) to point at
00136   // the AP startup code prior to the [universal startup algorithm]."
00137   outb(CMOS_PORT, 0xF);  // offset 0xF is shutdown code
00138   outb(CMOS_PORT+1, 0x0A);
00139   wrv = (ushort*)P2V((0x40«4 | 0x67));  // Warm reset vector
00140   wrv[0] = 0;
00141   wrv[1] = addr » 4;
00142
00143   // "Universal startup algorithm."
00144   // Send INIT (level-triggered) interrupt to reset other CPU.
00145   lapicw(ICRHI, apicid«24);
```

```
00146   lapicw(ICRLO, INIT | LEVEL | ASSERT);
00147   microdelay(200);
00148   lapicw(ICRLO, INIT | LEVEL);
00149   microdelay(100);    // should be 10ms, but too slow in Bochs!
00150
00151   // Send startup IPI (twice!) to enter code.
00152   // Regular hardware is supposed to only accept a STARTUP
00153   // when it is in the halted state due to an INIT.  So the second
00154   // should be ignored, but it is part of the official Intel algorithm.
00155   // Bochs complains about the second one.  Too bad for Bochs.
00156   for(i = 0; i < 2; i++){
00157     lapicw(ICRHI, apicid<<24);
00158     lapicw(ICRLO, STARTUP | (addr>>12));
00159     microdelay(200);
00160   }
00161 }
00162
00163 #define CMOS_STATA   0x0a
00164 #define CMOS_STATB   0x0b
00165 #define CMOS_UIP    (1 << 7)        // RTC update in progress
00166
00167 #define SECS    0x00
00168 #define MINS    0x02
00169 #define HOURS   0x04
00170 #define DAY     0x07
00171 #define MONTH   0x08
00172 #define YEAR    0x09
00173
00174 static uint
00175 cmos_read(uint reg)
00176 {
00177   outb(CMOS_PORT,  reg);
00178   microdelay(200);
00179
00180   return inb(CMOS_RETURN);
00181 }
00182
00183 static void
00184 fill_rtcdate(struct rtcdate *r)
00185 {
00186   r->second = cmos_read(SECS);
00187   r->minute = cmos_read(MINS);
00188   r->hour   = cmos_read(HOURS);
00189   r->day    = cmos_read(DAY);
00190   r->month  = cmos_read(MONTH);
00191   r->year   = cmos_read(YEAR);
00192 }
00193
00194 // qemu seems to use 24-hour GWT and the values are BCD encoded
00195 void
00196 cmostime(struct rtcdate *r)
00197 {
00198   struct rtcdate t1, t2;
00199   int sb, bcd;
00200
00201   sb = cmos_read(CMOS_STATB);
00202
00203   bcd = (sb & (1 << 2)) == 0;
00204
00205   // make sure CMOS doesn't modify time while we read it
00206   for(;;) {
00207     fill_rtcdate(&t1);
00208     if(cmos_read(CMOS_STATA) & CMOS_UIP)
00209         continue;
00210     fill_rtcdate(&t2);
00211     if(memcmp(&t1, &t2, sizeof(t1)) == 0)
00212       break;
00213   }
00214
00215   // convert
00216   if(bcd) {
00217 #define    CONV(x)     (t1.x = ((t1.x >> 4) * 10) + (t1.x & 0xf))
00218     CONV(second);
00219     CONV(minute);
00220     CONV(hour  );
00221     CONV(day   );
00222     CONV(month );
00223     CONV(year  );
00224 #undef     CONV
00225   }
00226
00227   *r = t1;
00228   r->year += 2000;
00229 }
```

5.99 lapic.d File Reference

5.100 lapic.d

Go to the documentation of this file.

```
00001 lapic.o: lapic.c /usr/include/stdc-predef.h param.h types.h defs.h date.h \
00002  memlayout.h traps.h mmu.h x86.h
```

5.101 ln.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- int main (int argc, char *argv[])

5.101.1 Function Documentation

5.101.1.1 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 6 of file ln.c.

```
00007 {
00008   if(argc != 3){
00009     printf(2, "Usage: ln old new\n");
00010     exit();
00011   }
00012   if(link(argv[1], argv[2]) < 0)
00013     printf(2, "link %s %s: failed\n", argv[1], argv[2]);
00014   exit();
00015 }
```

5.102 ln.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004
00005 int
00006 main(int argc, char *argv[])
00007 {
00008   if(argc != 3){
00009     printf(2, "Usage: ln old new\n");
00010     exit();
00011   }
00012   if(link(argv[1], argv[2]) < 0)
00013     printf(2, "link %s %s: failed\n", argv[1], argv[2]);
00014   exit();
00015 }
```

5.103 ln.d File Reference

5.104 ln.d

Go to the documentation of this file.
```
00001 ln.o: ln.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.105 log.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "buf.h"
```

Classes

- struct log
- struct logheader

Functions

- void begin_op (void)
- static void commit ()
- void end_op (void)
- void initlog (int dev)
- static void install_trans (void)
- void log_write (struct buf *b)
- static void read_head (void)
- static void recover_from_log (void)
- static void write_head (void)
- static void write_log (void)

Variables

- struct log log

5.105.1 Function Documentation

5.105.1.1 begin_op()

```
void begin_op (
              void  )
```

Definition at line 126 of file log.c.

```
00127 {
00128   acquire(&log.lock);
00129   while(1){
00130     if(log.committing){
00131       sleep(&log, &log.lock);
00132     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
00133       // this op might exhaust log space; wait for commit.
00134       sleep(&log, &log.lock);
00135     } else {
00136       log.outstanding += 1;
00137       release(&log.lock);
00138       break;
00139     }
00140   }
00141 }
```

Referenced by exec(), exit(), fileclose(), filewrite(), sys_chdir(), sys_link(), sys_mkdir(), sys_mknod(), sys_open(), and sys_unlink().

5.105.1.2 commit()

```
static void commit ( )  [static]
```

Definition at line 193 of file log.c.

```
00194 {
00195   if (log.lh.n > 0) {
00196     write_log();     // Write modified blocks from cache to log
00197     write_head();    // Write header to disk -- the real commit
00198     install_trans(); // Now install writes to home locations
00199     log.lh.n = 0;
00200     write_head();    // Erase the transaction from the log
00201   }
00202 }
```

Referenced by end_op().

5.105.1.3 end_op()

```
void end_op (
              void  )
```

Definition at line 146 of file log.c.

```
00147 {
00148   int do_commit = 0;
00149
00150   acquire(&log.lock);
00151   log.outstanding -= 1;
00152   if(log.committing)
00153     panic("log.committing");
00154   if(log.outstanding == 0){
00155     do_commit = 1;
00156     log.committing = 1;
00157   } else {
00158     // begin_op() may be waiting for log space,
00159     // and decrementing log.outstanding has decreased
00160     // the amount of reserved space.
00161     wakeup(&log);
00162   }
```

```
00163   release(&log.lock);
00164
00165   if(do_commit){
00166     // call commit w/o holding locks, since not allowed
00167     // to sleep with locks.
00168     commit();
00169     acquire(&log.lock);
00170     log.committing = 0;
00171     wakeup(&log);
00172     release(&log.lock);
00173   }
00174 }
```

Referenced by exec(), exit(), fileclose(), filewrite(), sys_chdir(), sys_link(), sys_mkdir(), sys_mknod(), sys_open(), and sys_unlink().

5.105.1.4 initlog()

```
void initlog (
             int dev )
```

Definition at line 54 of file log.c.

```
00055 {
00056   if (sizeof(struct logheader) >= BSIZE)
00057     panic("initlog: too big logheader");
00058
00059   struct superblock sb;
00060   initlock(&log.lock, "log");
00061   readsb(dev, &sb);
00062   log.start = sb.logstart;
00063   log.size = sb.nlog;
00064   log.dev = dev;
00065   recover_from_log();
00066 }
```

Referenced by forkret().

5.105.1.5 install_trans()

```
static void install_trans (
             void  )  [static]
```

Definition at line 70 of file log.c.

```
00071 {
00072   int tail;
00073
00074   for (tail = 0; tail < log.lh.n; tail++) {
00075     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
00076     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
00077     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
00078     bwrite(dbuf);  // write dst to disk
00079     brelse(lbuf);
00080     brelse(dbuf);
00081   }
00082 }
```

Referenced by commit(), and recover_from_log().

5.105.1.6 log_write()

```
void log_write (
            struct buf * b )
```

Definition at line 214 of file log.c.

```
00215 {
00216   int i;
00217
00218   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
00219     panic("too big a transaction");
00220   if (log.outstanding < 1)
00221     panic("log_write outside of trans");
00222
00223   acquire(&log.lock);
00224   for (i = 0; i < log.lh.n; i++) {
00225     if (log.lh.block[i] == b->blockno)   // log absorbtion
00226       break;
00227   }
00228   log.lh.block[i] = b->blockno;
00229   if (i == log.lh.n)
00230     log.lh.n++;
00231   b->flags |= B_DIRTY; // prevent eviction
00232   release(&log.lock);
00233 }
```

Referenced by balloc(), bfree(), bmap(), bzero(), ialloc(), iupdate(), and writei().

5.105.1.7 read_head()

```
static void read_head (
            void  )  [static]
```

Definition at line 86 of file log.c.

```
00087 {
00088   struct buf *buf = bread(log.dev, log.start);
00089   struct logheader *lh = (struct logheader *) (buf->data);
00090   int i;
00091   log.lh.n = lh->n;
00092   for (i = 0; i < log.lh.n; i++) {
00093     log.lh.block[i] = lh->block[i];
00094   }
00095   brelse(buf);
00096 }
```

Referenced by recover_from_log().

5.105.1.8 recover_from_log()

```
static void recover_from_log (
            void  )  [static]
```

Definition at line 116 of file log.c.

```
00117 {
00118   read_head();
00119   install_trans(); // if committed, copy from log to disk
00120   log.lh.n = 0;
00121   write_head(); // clear the log
00122 }
```

Referenced by initlog().

5.105.1.9 write_head()

```
static void write_head (
            void  )  [static]
```

Definition at line 102 of file log.c.

```
00103 {
00104   struct buf *buf = bread(log.dev, log.start);
00105   struct logheader *hb = (struct logheader *) (buf->data);
00106   int i;
00107   hb->n = log.lh.n;
00108   for (i = 0; i < log.lh.n; i++) {
00109     hb->block[i] = log.lh.block[i];
00110   }
00111   bwrite(buf);
00112   brelse(buf);
00113 }
```

Referenced by commit(), and recover_from_log().

5.105.1.10 write_log()

```
static void write_log (
            void  )  [static]
```

Definition at line 178 of file log.c.

```
00179 {
00180   int tail;
00181
00182   for (tail = 0; tail < log.lh.n; tail++) {
00183     struct buf *to = bread(log.dev, log.start+tail+1); // log block
00184     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
00185     memmove(to->data, from->data, BSIZE);
00186     bwrite(to);  // write the log
00187     brelse(from);
00188     brelse(to);
00189   }
00190 }
```

Referenced by commit().

5.105.2 Variable Documentation

5.105.2.1 log

```
struct log log
```

Definition at line 48 of file log.c.

5.106 log.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "defs.h"
00003 #include "param.h"
00004 #include "spinlock.h"
00005 #include "sleeplock.h"
00006 #include "fs.h"
00007 #include "buf.h"
00008
00009 // Simple logging that allows concurrent FS system calls.
00010 //
00011 // A log transaction contains the updates of multiple FS system
00012 // calls. The logging system only commits when there are
00013 // no FS system calls active. Thus there is never
00014 // any reasoning required about whether a commit might
00015 // write an uncommitted system call's updates to disk.
00016 //
00017 // A system call should call begin_op()/end_op() to mark
00018 // its start and end. Usually begin_op() just increments
00019 // the count of in-progress FS system calls and returns.
00020 // But if it thinks the log is close to running out, it
00021 // sleeps until the last outstanding end_op() commits.
00022 //
00023 // The log is a physical re-do log containing disk blocks.
00024 // The on-disk log format:
00025 //   header block, containing block #s for block A, B, C, ...
00026 //   block A
00027 //   block B
00028 //   block C
00029 //   ...
00030 // Log appends are synchronous.
00031
00032 // Contents of the header block, used for both the on-disk header block
00033 // and to keep track in memory of logged block# before commit.
00034 struct logheader {
00035   int n;
00036   int block[LOGSIZE];
00037 };
00038
00039 struct log {
00040   struct spinlock lock;
00041   int start;
00042   int size;
00043   int outstanding; // how many FS sys calls are executing.
00044   int committing;  // in commit(), please wait.
00045   int dev;
00046   struct logheader lh;
00047 };
00048 struct log log;
00049
00050 static void recover_from_log(void);
00051 static void commit();
00052
00053 void
00054 initlog(int dev)
00055 {
00056   if (sizeof(struct logheader) >= BSIZE)
00057     panic("initlog: too big logheader");
00058
00059   struct superblock sb;
00060   initlock(&log.lock, "log");
00061   readsb(dev, &sb);
00062   log.start = sb.logstart;
00063   log.size = sb.nlog;
00064   log.dev = dev;
00065   recover_from_log();
00066 }
00067
00068 // Copy committed blocks from log to their home location
00069 static void
00070 install_trans(void)
00071 {
00072   int tail;
00073
00074   for (tail = 0; tail < log.lh.n; tail++) {
00075     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
00076     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
00077     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
00078     bwrite(dbuf);  // write dst to disk
00079     brelse(lbuf);
00080     brelse(dbuf);
00081   }
00082 }
```

```
00083
00084 // Read the log header from disk into the in-memory log header
00085 static void
00086 read_head(void)
00087 {
00088   struct buf *buf = bread(log.dev, log.start);
00089   struct logheader *lh = (struct logheader *) (buf->data);
00090   int i;
00091   log.lh.n = lh->n;
00092   for (i = 0; i < log.lh.n; i++) {
00093     log.lh.block[i] = lh->block[i];
00094   }
00095   brelse(buf);
00096 }
00097
00098 // Write in-memory log header to disk.
00099 // This is the true point at which the
00100 // current transaction commits.
00101 static void
00102 write_head(void)
00103 {
00104   struct buf *buf = bread(log.dev, log.start);
00105   struct logheader *hb = (struct logheader *) (buf->data);
00106   int i;
00107   hb->n = log.lh.n;
00108   for (i = 0; i < log.lh.n; i++) {
00109     hb->block[i] = log.lh.block[i];
00110   }
00111   bwrite(buf);
00112   brelse(buf);
00113 }
00114
00115 static void
00116 recover_from_log(void)
00117 {
00118   read_head();
00119   install_trans(); // if committed, copy from log to disk
00120   log.lh.n = 0;
00121   write_head(); // clear the log
00122 }
00123
00124 // called at the start of each FS system call.
00125 void
00126 begin_op(void)
00127 {
00128   acquire(&log.lock);
00129   while(1){
00130     if(log.committing){
00131       sleep(&log, &log.lock);
00132     } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
00133       // this op might exhaust log space; wait for commit.
00134       sleep(&log, &log.lock);
00135     } else {
00136       log.outstanding += 1;
00137       release(&log.lock);
00138       break;
00139     }
00140   }
00141 }
00142
00143 // called at the end of each FS system call.
00144 // commits if this was the last outstanding operation.
00145 void
00146 end_op(void)
00147 {
00148   int do_commit = 0;
00149
00150   acquire(&log.lock);
00151   log.outstanding -= 1;
00152   if(log.committing)
00153     panic("log.committing");
00154   if(log.outstanding == 0){
00155     do_commit = 1;
00156     log.committing = 1;
00157   } else {
00158     // begin_op() may be waiting for log space,
00159     // and decrementing log.outstanding has decreased
00160     // the amount of reserved space.
00161     wakeup(&log);
00162   }
00163   release(&log.lock);
00164
00165   if(do_commit){
00166     // call commit w/o holding locks, since not allowed
00167     // to sleep with locks.
00168     commit();
00169     acquire(&log.lock);
```

```
00170     log.committing = 0;
00171     wakeup(&log);
00172     release(&log.lock);
00173   }
00174 }
00175
00176 // Copy modified blocks from cache to log.
00177 static void
00178 write_log(void)
00179 {
00180   int tail;
00181
00182   for (tail = 0; tail < log.lh.n; tail++) {
00183     struct buf *to = bread(log.dev, log.start+tail+1); // log block
00184     struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
00185     memmove(to->data, from->data, BSIZE);
00186     bwrite(to);  // write the log
00187     brelse(from);
00188     brelse(to);
00189   }
00190 }
00191
00192 static void
00193 commit()
00194 {
00195   if (log.lh.n > 0) {
00196     write_log();     // Write modified blocks from cache to log
00197     write_head();    // Write header to disk -- the real commit
00198     install_trans(); // Now install writes to home locations
00199     log.lh.n = 0;
00200     write_head();    // Erase the transaction from the log
00201   }
00202 }
00203
00204 // Caller has modified b->data and is done with the buffer.
00205 // Record the block number and pin in the cache with B_DIRTY.
00206 // commit()/write_log() will do the disk write.
00207 //
00208 // log_write() replaces bwrite(); a typical use is:
00209 //   bp = bread(...)
00210 //   modify bp->data[]
00211 //   log_write(bp)
00212 //   brelse(bp)
00213 void
00214 log_write(struct buf *b)
00215 {
00216   int i;
00217
00218   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
00219     panic("too big a transaction");
00220   if (log.outstanding < 1)
00221     panic("log_write outside of trans");
00222
00223   acquire(&log.lock);
00224   for (i = 0; i < log.lh.n; i++) {
00225     if (log.lh.block[i] == b->blockno)   // log absorbtion
00226       break;
00227   }
00228   log.lh.block[i] = b->blockno;
00229   if (i == log.lh.n)
00230     log.lh.n++;
00231   b->flags |= B_DIRTY; // prevent eviction
00232   release(&log.lock);
00233 }
00234
```

5.107 log.d File Reference

5.108 log.d

Go to the documentation of this file.

```
00001 log.o: log.c /usr/include/stdc-predef.h types.h defs.h param.h spinlock.h \
00002  sleeplock.h fs.h buf.h
```

5.109 ls.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fs.h"
```

Functions

- char * fmtname (char *path)
- void ls (char *path)
- int main (int argc, char *argv[])

5.109.1 Function Documentation

5.109.1.1 fmtname()

```
char * fmtname (
            char * path )
```

Definition at line 7 of file ls.c.

```
00008 {
00009   static char buf[DIRSIZ+1];
00010   char *p;
00011
00012   // Find first character after last slash.
00013   for(p=path+strlen(path); p >= path && *p != '/'; p--)
00014     ;
00015   p++;
00016
00017   // Return blank-padded name.
00018   if(strlen(p) >= DIRSIZ)
00019     return p;
00020   memmove(buf, p, strlen(p));
00021   memset(buf+strlen(p), ' ', DIRSIZ-strlen(p));
00022   return buf;
00023 }
```

Referenced by ls().

5.109.1.2 ls()

```
void ls (
            char * path )
```

Definition at line 26 of file ls.c.

```
00027 {
00028   char buf[512], *p;
00029   int fd;
00030   struct dirent de;
00031   struct stat st;
00032
00033   if((fd = open(path, 0)) < 0){
00034     printf(2, "ls: cannot open %s\n", path);
```

```
00035     return;
00036   }
00037
00038   if(fstat(fd, &st) < 0){
00039     printf(2, "ls: cannot stat %s\n", path);
00040     close(fd);
00041     return;
00042   }
00043
00044   switch(st.type){
00045   case T_FILE:
00046     printf(1, "%s %d %d %d\n", fmtname(path), st.type, st.ino, st.size);
00047     break;
00048
00049   case T_DIR:
00050     if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
00051       printf(1, "ls: path too long\n");
00052       break;
00053     }
00054     strcpy(buf, path);
00055     p = buf+strlen(buf);
00056     *p++ = '/';
00057     while(read(fd, &de, sizeof(de)) == sizeof(de)){
00058       if(de.inum == 0)
00059         continue;
00060       memmove(p, de.name, DIRSIZ);
00061       p[DIRSIZ] = 0;
00062       if(stat(buf, &st) < 0){
00063         printf(1, "ls: cannot stat %s\n", buf);
00064         continue;
00065       }
00066       printf(1, "%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
00067     }
00068     break;
00069   }
00070   close(fd);
00071 }
```

Referenced by main().

5.109.1.3 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 74 of file ls.c.

```
00075 {
00076   int i;
00077
00078   if(argc < 2){
00079     ls(".");
00080     exit();
00081   }
00082   for(i=1; i<argc; i++)
00083     ls(argv[i]);
00084   exit();
00085 }
```

5.110 ls.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004 #include "fs.h"
00005
00006 char*
00007 fmtname(char *path)
00008 {
00009   static char buf[DIRSIZ+1];
```

```
00010   char *p;
00011
00012   // Find first character after last slash.
00013   for(p=path+strlen(path); p >= path && *p != '/'; p--)
00014     ;
00015   p++;
00016
00017   // Return blank-padded name.
00018   if(strlen(p) >= DIRSIZ)
00019     return p;
00020   memmove(buf, p, strlen(p));
00021   memset(buf+strlen(p), ' ', DIRSIZ-strlen(p));
00022   return buf;
00023 }
00024
00025 void
00026 ls(char *path)
00027 {
00028   char buf[512], *p;
00029   int fd;
00030   struct dirent de;
00031   struct stat st;
00032
00033   if((fd = open(path, 0)) < 0){
00034     printf(2, "ls: cannot open %s\n", path);
00035     return;
00036   }
00037
00038   if(fstat(fd, &st) < 0){
00039     printf(2, "ls: cannot stat %s\n", path);
00040     close(fd);
00041     return;
00042   }
00043
00044   switch(st.type){
00045   case T_FILE:
00046     printf(1, "%s %d %d %d\n", fmtname(path), st.type, st.ino, st.size);
00047     break;
00048
00049   case T_DIR:
00050     if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
00051       printf(1, "ls: path too long\n");
00052       break;
00053     }
00054     strcpy(buf, path);
00055     p = buf+strlen(buf);
00056     *p++ = '/';
00057     while(read(fd, &de, sizeof(de)) == sizeof(de)){
00058       if(de.inum == 0)
00059         continue;
00060       memmove(p, de.name, DIRSIZ);
00061       p[DIRSIZ] = 0;
00062       if(stat(buf, &st) < 0){
00063         printf(1, "ls: cannot stat %s\n", buf);
00064         continue;
00065       }
00066       printf(1, "%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
00067     }
00068     break;
00069   }
00070   close(fd);
00071 }
00072
00073 int
00074 main(int argc, char *argv[])
00075 {
00076   int i;
00077
00078   if(argc < 2){
00079     ls(".");
00080     exit();
00081   }
00082   for(i=1; i<argc; i++)
00083     ls(argv[i]);
00084   exit();
00085 }
```

5.111 ls.d File Reference

5.112 ls.d

Go to the documentation of this file.
```
00001 ls.o: ls.c /usr/include/stdc-predef.h types.h stat.h user.h fs.h
```

5.113 main.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
```

Functions

- __attribute__ ((__aligned__(PGSIZE)))
- static void mpenter (void)
- static void mpmain (void)
- static void startothers (void)

Variables

- pde_t entrypgdir []

5.113.1 Function Documentation

5.113.1.1 __attribute__()

```
__attribute__ (
            (__aligned__(PGSIZE))   )
```

Definition at line 102 of file main.c.
```
00103                           {
00104   // Map VA's [0, 4MB) to PA's [0, 4MB)
00105   [0] = (0) | PTE_P | PTE_W | PTE_PS,
00106   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
00107   [KERNBASE»PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
00108 };
```

5.113.1.2 mpenter()

```
static void mpenter (
            void  )  [static]
```

Definition at line 42 of file main.c.

```
00043 {
00044   switchkvm();
00045   seginit();
00046   lapicinit();
00047   mpmain();
00048 }
```

Referenced by startothers().

5.113.1.3 mpmain()

```
static void mpmain (
            void  )  [static]
```

Definition at line 10 of file main.c.

```
00019 {
00020   kinit1(end, P2V(4*1024*1024)); // phys page allocator
00021   kvmalloc();      // kernel page table
00022   mpinit();        // detect other processors
00023   lapicinit();     // interrupt controller
00024   seginit();       // segment descriptors
00025   picinit();       // disable pic
00026   ioapicinit();    // another interrupt controller
00027   consoleinit();   // console hardware
00028   uartinit();      // serial port
00029   pinit();         // process table
00030   tvinit();        // trap vectors
00031   binit();         // buffer cache
00032   fileinit();      // file table
00033   ideinit();       // disk
00034   startothers();   // start other processors
00035   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
00036   userinit();      // first user process
00037   mpmain();        // finish this processor's setup
00038 }
```

Referenced by mpenter().

5.113.1.4 startothers()

```
static void startothers (
            void  )  [static]
```

Definition at line 64 of file main.c.

```
00065 {
00066   extern uchar _binary_entryother_start[], _binary_entryother_size[];
00067   uchar *code;
00068   struct cpu *c;
00069   char *stack;
00070
00071   // Write entry code to unused memory at 0x7000.
00072   // The linker has placed the image of entryother.S in
00073   // _binary_entryother_start.
00074   code = P2V(0x7000);
00075   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
00076
00077   for(c = cpus; c < cpus+ncpu; c++){
00078     if(c == mycpu())  // We've started already.
```

```
00079       continue;
00080
00081     // Tell entryother.S what stack to use, where to enter, and what
00082     // pgdir to use. We cannot use kpgdir yet, because the AP processor
00083     // is running in low  memory, so we use entrypgdir for the APs too.
00084     stack = kalloc();
00085     *(void**)(code-4) = stack + KSTACKSIZE;
00086     *(void(**)(void))(code-8) = mpenter;
00087     *(int**)(code-12) = (void *) V2P(entrypgdir);
00088
00089     lapicstartap(c->apicid, V2P(code));
00090
00091     // wait for cpu to finish mpmain()
00092     while(c->started == 0)
00093       ;
00094   }
00095 }
```

5.113.2 Variable Documentation

5.113.2.1 entrypgdir

```
pde_t entrypgdir[]
```

Definition at line 60 of file main.c.

Referenced by startothers().

5.114 main.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "defs.h"
00003 #include "param.h"
00004 #include "memlayout.h"
00005 #include "mmu.h"
00006 #include "proc.h"
00007 #include "x86.h"
00008
00009 static void startothers(void);
00010 static void mpmain(void)  __attribute__((noreturn));
00011 extern pde_t *kpgdir;
00012 extern char end[]; // first address after kernel loaded from ELF file
00013
00014 // Bootstrap processor starts running C code here.
00015 // Allocate a real stack and switch to it, first
00016 // doing some setup required for memory allocator to work.
00017 int
00018 main(void)
00019 {
00020   kinit1(end, P2V(4*1024*1024)); // phys page allocator
00021   kvmalloc();      // kernel page table
00022   mpinit();        // detect other processors
00023   lapicinit();     // interrupt controller
00024   seginit();       // segment descriptors
00025   picinit();       // disable pic
00026   ioapicinit();    // another interrupt controller
00027   consoleinit();   // console hardware
00028   uartinit();      // serial port
00029   pinit();         // process table
00030   tvinit();        // trap vectors
00031   binit();         // buffer cache
00032   fileinit();      // file table
00033   ideinit();       // disk
00034   startothers();   // start other processors
00035   kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
00036   userinit();      // first user process
00037   mpmain();        // finish this processor's setup
```

```
00038 }
00039
00040 // Other CPUs jump here from entryother.S.
00041 static void
00042 mpenter(void)
00043 {
00044   switchkvm();
00045   seginit();
00046   lapicinit();
00047   mpmain();
00048 }
00049
00050 // Common CPU setup code.
00051 static void
00052 mpmain(void)
00053 {
00054   cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
00055   idtinit();       // load idt register
00056   xchg(&(mycpu()->started), 1); // tell startothers() we're up
00057   scheduler();     // start running processes
00058 }
00059
00060 pde_t entrypgdir[];  // For entry.S
00061
00062 // Start the non-boot (AP) processors.
00063 static void
00064 startothers(void)
00065 {
00066   extern uchar _binary_entryother_start[], _binary_entryother_size[];
00067   uchar *code;
00068   struct cpu *c;
00069   char *stack;
00070
00071   // Write entry code to unused memory at 0x7000.
00072   // The linker has placed the image of entryother.S in
00073   // _binary_entryother_start.
00074   code = P2V(0x7000);
00075   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
00076
00077   for(c = cpus; c < cpus+ncpu; c++){
00078     if(c == mycpu())  // We've started already.
00079       continue;
00080
00081     // Tell entryother.S what stack to use, where to enter, and what
00082     // pgdir to use. We cannot use kpgdir yet, because the AP processor
00083     // is running in low  memory, so we use entrypgdir for the APs too.
00084     stack = kalloc();
00085     *(void**)(code-4) = stack + KSTACKSIZE;
00086     *(void(**)(void))(code-8) = mpenter;
00087     *(int**)(code-12) = (void *) V2P(entrypgdir);
00088
00089     lapicstartap(c->apicid, V2P(code));
00090
00091     // wait for cpu to finish mpmain()
00092     while(c->started == 0)
00093       ;
00094   }
00095 }
00096
00097 // The boot page table used in entry.S and entryother.S.
00098 // Page directories (and page tables) must start on page boundaries,
00099 // hence the __aligned__ attribute.
00100 // PTE_PS in a page directory entry enables 4Mbyte pages.
00101
00102 __attribute__((__aligned__(PGSIZE)))
00103 pde_t entrypgdir[NPDENTRIES] = {
00104   // Map VA's [0, 4MB) to PA's [0, 4MB)
00105   [0] = (0) | PTE_P | PTE_W | PTE_PS,
00106   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
00107   [KERNBASE»PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
00108 };
00109
00110 //PAGEBREAK!
00111 // Blank page.
00112 //PAGEBREAK!
00113 // Blank page.
00114 //PAGEBREAK!
00115 // Blank page.
00116
```

5.115 main.d File Reference

5.116 main.d

Go to the documentation of this file.

```
00001 main.o: main.c /usr/include/stdc-predef.h types.h defs.h param.h \
00002  memlayout.h mmu.h proc.h x86.h
```

5.117 memide.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "buf.h"
```

Functions

- void ideinit (void)
- void ideintr (void)
- void iderw (struct buf *b)

Variables

- uchar _binary_fs_img_size []
- uchar _binary_fs_img_start []
- static int disksize
- static uchar * memdisk

5.117.1 Function Documentation

5.117.1.1 ideinit()

```
void ideinit (
              void  )
```

Definition at line 22 of file memide.c.

```
00023 {
00024   memdisk = _binary_fs_img_start;
00025   disksize = (uint)_binary_fs_img_size/BSIZE;
00026 }
```

5.117.1.2 ideintr()

```
void ideintr (
              void  )
```

Definition at line 30 of file memide.c.

```
00031 {
00032   // no-op
00033 }
```

Referenced by trap().

5.117.1.3 iderw()

```
void iderw (
              struct buf * b )
```

Definition at line 39 of file memide.c.

```
00040 {
00041   uchar *p;
00042
00043   if(!holdingsleep(&b->lock))
00044     panic("iderw: buf not locked");
00045   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
00046     panic("iderw: nothing to do");
00047   if(b->dev != 1)
00048     panic("iderw: request not for disk 1");
00049   if(b->blockno >= disksize)
00050     panic("iderw: block out of range");
00051
00052   p = memdisk + b->blockno*BSIZE;
00053
00054   if(b->flags & B_DIRTY){
00055     b->flags &= ~B_DIRTY;
00056     memmove(p, b->data, BSIZE);
00057   } else
00058     memmove(b->data, p, BSIZE);
00059   b->flags |= B_VALID;
00060 }
```

Referenced by bread(), and bwrite().

5.117.2 Variable Documentation

5.117.2.1 _binary_fs_img_size

```
uchar _binary_fs_img_size[]
```

Definition at line 16 of file memide.c.

Referenced by ideinit().

5.117.2.2 _binary_fs_img_start

```
uchar _binary_fs_img_start[]  [extern]
```

Referenced by ideinit().

5.117.2.3 disksize

```
int disksize  [static]
```

Definition at line 18 of file memide.c.

Referenced by ideinit(), and iderw().

5.117.2.4 memdisk

```
uchar* memdisk  [static]
```

Definition at line 19 of file memide.c.

Referenced by ideinit(), and iderw().

5.118 memide.c

Go to the documentation of this file.

```
00001 // Fake IDE disk; stores blocks in memory.
00002 // Useful for running kernel without scratch disk.
00003
00004 #include "types.h"
00005 #include "defs.h"
00006 #include "param.h"
00007 #include "mmu.h"
00008 #include "proc.h"
00009 #include "x86.h"
00010 #include "traps.h"
00011 #include "spinlock.h"
00012 #include "sleeplock.h"
00013 #include "fs.h"
00014 #include "buf.h"
00015
00016 extern uchar _binary_fs_img_start[], _binary_fs_img_size[];
00017
00018 static int disksize;
00019 static uchar *memdisk;
00020
00021 void
00022 ideinit(void)
00023 {
00024   memdisk = _binary_fs_img_start;
00025   disksize = (uint)_binary_fs_img_size/BSIZE;
00026 }
00027
00028 // Interrupt handler.
00029 void
00030 ideintr(void)
00031 {
00032   // no-op
00033 }
00034
00035 // Sync buf with disk.
```

```
00036 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
00037 // Else if B_VALID is not set, read buf from disk, set B_VALID.
00038 void
00039 iderw(struct buf *b)
00040 {
00041   uchar *p;
00042
00043   if(!holdingsleep(&b->lock))
00044     panic("iderw: buf not locked");
00045   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
00046     panic("iderw: nothing to do");
00047   if(b->dev != 1)
00048     panic("iderw: request not for disk 1");
00049   if(b->blockno >= disksize)
00050     panic("iderw: block out of range");
00051
00052   p = memdisk + b->blockno*BSIZE;
00053
00054   if(b->flags & B_DIRTY){
00055     b->flags &= ~B_DIRTY;
00056     memmove(p, b->data, BSIZE);
00057   } else
00058     memmove(b->data, p, BSIZE);
00059   b->flags |= B_VALID;
00060 }
```

5.119 memlayout.h File Reference

Macros

- #define DEVSPACE 0xFE000000
- #define EXTMEM 0x100000
- #define KERNBASE 0x80000000
- #define KERNLINK (KERNBASE+EXTMEM)
- #define P2V(a) ((void *)(((char *) (a)) + KERNBASE))
- #define P2V_WO(x) ((x) + KERNBASE)
- #define PHYSTOP 0xE000000
- #define V2P(a) (((uint) (a)) - KERNBASE)
- #define V2P_WO(x) ((x) - KERNBASE)

5.119.1 Macro Definition Documentation

5.119.1.1 DEVSPACE

```
#define DEVSPACE 0xFE000000
```

Definition at line 5 of file memlayout.h.

5.119.1.2 EXTMEM

```
#define EXTMEM 0x100000
```

Definition at line 3 of file memlayout.h.

5.119.1.3 KERNBASE

```
#define KERNBASE 0x80000000
```

Definition at line 8 of file memlayout.h.

5.119.1.4 KERNLINK

```
#define KERNLINK (KERNBASE+EXTMEM)
```

Definition at line 9 of file memlayout.h.

5.119.1.5 P2V

```
#define P2V(
              a ) ((void *)(((char *) (a)) + KERNBASE))
```

Definition at line 12 of file memlayout.h.

5.119.1.6 P2V_WO

```
#define P2V_WO(
              x ) ((x) + KERNBASE)
```

Definition at line 15 of file memlayout.h.

5.119.1.7 PHYSTOP

```
#define PHYSTOP 0xE000000
```

Definition at line 4 of file memlayout.h.

5.119.1.8 V2P

```
#define V2P(
              a ) (((uint) (a)) - KERNBASE)
```

Definition at line 11 of file memlayout.h.

5.119.1.9 V2P_WO

```
#define V2P_WO(
              x ) ((x) - KERNBASE)
```

Definition at line 14 of file memlayout.h.

5.120 memlayout.h

Go to the documentation of this file.

```
00001 // Memory layout
00002
00003 #define EXTMEM  0x100000            // Start of extended memory
00004 #define PHYSTOP 0xE000000           // Top physical memory
00005 #define DEVSPACE 0xFE000000         // Other devices are at high addresses
00006
00007 // Key addresses for address space layout (see kmap in vm.c for layout)
00008 #define KERNBASE 0x80000000         // First kernel virtual address
00009 #define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
00010
00011 #define V2P(a) (((uint) (a)) - KERNBASE)
00012 #define P2V(a) ((void *)(((char *) (a)) + KERNBASE))
00013
00014 #define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
00015 #define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
```

5.121 mkdir.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- int main (int argc, char *argv[])

5.121.1 Function Documentation

5.121.1.1 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 6 of file mkdir.c.

```
00007 {
00008   int i;
00009
00010   if(argc < 2){
00011     printf(2, "Usage: mkdir files...\n");
00012     exit();
00013   }
00014
00015   for(i = 1; i < argc; i++){
00016     if(mkdir(argv[i]) < 0){
00017       printf(2, "mkdir: %s failed to create\n", argv[i]);
00018       break;
00019     }
00020   }
00021
00022   exit();
00023 }
```

5.122 mkdir.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004
00005 int
00006 main(int argc, char *argv[])
00007 {
00008   int i;
00009
00010   if(argc < 2){
00011     printf(2, "Usage: mkdir files...\n");
00012     exit();
00013   }
00014
00015   for(i = 1; i < argc; i++){
00016     if(mkdir(argv[i]) < 0){
00017       printf(2, "mkdir: %s failed to create\n", argv[i]);
00018       break;
00019     }
00020   }
00021
00022   exit();
00023 }
```

5.123 mkdir.d File Reference

5.124 mkdir.d

Go to the documentation of this file.

```
00001 mkdir.o: mkdir.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.125 mkfs.c File Reference

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <assert.h>
#include "types.h"
#include "fs.h"
#include "stat.h"
#include "param.h"
```

Macros

- #define min(a, b) ((a) < (b) ? (a) : (b))
- #define NINODES 200
- #define stat xv6_stat
- #define static_assert(a, b) do { switch (0) case 0: case (a): ; } while (0)

Functions

- void balloc (int)
- uint ialloc (ushort type)
- void iappend (uint inum, void *p, int n)
- int main (int argc, char *argv[])
- void rinode (uint inum, struct dinode *ip)
- void rsect (uint sec, void *buf)
- void winode (uint, struct dinode *)
- void wsect (uint, void *)
- uint xint (uint x)
- ushort xshort (ushort x)

Variables

- uint freeblock
- uint freeinode = 1
- int fsfd
- int nbitmap = FSSIZE/(BSIZE*8) + 1
- int nblocks
- int ninodeblocks = NINODES / IPB + 1
- int nlog = LOGSIZE
- int nmeta
- struct superblock sb
- char zeroes [BSIZE]

5.125.1 Macro Definition Documentation

5.125.1.1 min

```
#define min(
              a,
              b ) ((a) < (b) ?  (a) :  (b))
```

Definition at line 253 of file mkfs.c.

5.125.1.2 NINODES

```
#define NINODES 200
```

Definition at line 18 of file mkfs.c.

5.125.1.3 stat

```
#define stat xv6_stat
```

Definition at line 8 of file mkfs.c.

5.125.1.4 static_assert

```
#define static_assert(
              a,
              b ) do { switch (0) case 0:  case (a):  ; } while (0)
```

Definition at line 15 of file mkfs.c.

5.125.2 Function Documentation

5.125.2.1 balloc()

```
void balloc (
              int used )
```

Definition at line 238 of file mkfs.c.

```
00239 {
00240   uchar buf[BSIZE];
00241   int i;
00242
00243   printf("balloc: first %d blocks have been allocated\n", used);
00244   assert(used < BSIZE*8);
00245   bzero(buf, BSIZE);
00246   for(i = 0; i < used; i++){
00247     buf[i/8] = buf[i/8] | (0x1 « (i%8));
00248   }
00249   printf("balloc: write bitmap block at sector %d\n", sb.bmapstart);
00250   wsect(sb.bmapstart, buf);
00251 }
```

Referenced by main().

5.125.2.2 ialloc()

```
uint ialloc (
              ushort type )
```

Definition at line 224 of file mkfs.c.

```
00225 {
00226   uint inum = freeinode++;
00227   struct dinode din;
00228
00229   bzero(&din, sizeof(din));
00230   din.type = xshort(type);
00231   din.nlink = xshort(1);
00232   din.size = xint(0);
00233   winode(inum, &din);
00234   return inum;
00235 }
```

Referenced by main().

5.125.2.3 iappend()

```
void iappend (
              uint inum,
              void * p,
              int n )
```

Definition at line 256 of file mkfs.c.

```
00257 {
00258   char *p = (char*)xp;
00259   uint fbn, off, n1;
00260   struct dinode din;
00261   char buf[BSIZE];
00262   uint indirect[NINDIRECT];
00263   uint x;
00264
00265   rinode(inum, &din);
00266   off = xint(din.size);
00267   // printf("append inum %d at off %d sz %d\n", inum, off, n);
00268   while(n > 0){
00269     fbn = off / BSIZE;
00270     assert(fbn < MAXFILE);
00271     if(fbn < NDIRECT){
00272       if(xint(din.addrs[fbn]) == 0){
00273         din.addrs[fbn] = xint(freeblock++);
00274       }
00275       x = xint(din.addrs[fbn]);
00276     } else {
00277       if(xint(din.addrs[NDIRECT]) == 0){
00278         din.addrs[NDIRECT] = xint(freeblock++);
00279       }
00280       rsect(xint(din.addrs[NDIRECT]), (char*)indirect);
00281       if(indirect[fbn - NDIRECT] == 0){
00282         indirect[fbn - NDIRECT] = xint(freeblock++);
00283         wsect(xint(din.addrs[NDIRECT]), (char*)indirect);
00284       }
00285       x = xint(indirect[fbn-NDIRECT]);
00286     }
00287     n1 = min(n, (fbn + 1) * BSIZE - off);
00288     rsect(x, buf);
00289     bcopy(p, buf + off - (fbn * BSIZE), n1);
00290     wsect(x, buf);
00291     n -= n1;
00292     off += n1;
00293     p += n1;
00294   }
00295   din.size = xint(off);
00296   winode(inum, &din);
00297 }
```

Referenced by main().

5.125.2.4 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 68 of file mkfs.c.

```
00069 {
00070   int i, cc, fd;
00071   uint rootino, inum, off;
00072   struct dirent de;
00073   char buf[BSIZE];
00074   struct dinode din;
00075
00076
00077   static_assert(sizeof(int) == 4, "Integers must be 4 bytes!");
00078
00079   if(argc < 2){
00080     fprintf(stderr, "Usage: mkfs fs.img files...\n");
00081     exit(1);
```

```
00082   }
00083
00084   assert((BSIZE % sizeof(struct dinode)) == 0);
00085   assert((BSIZE % sizeof(struct dirent)) == 0);
00086
00087   fsfd = open(argv[1], O_RDWR|O_CREAT|O_TRUNC, 0666);
00088   if(fsfd < 0){
00089     perror(argv[1]);
00090     exit(1);
00091   }
00092
00093   // 1 fs block = 1 disk sector
00094   nmeta = 2 + nlog + ninodeblocks + nbitmap;
00095   nblocks = FSSIZE - nmeta;
00096
00097   sb.size = xint(FSSIZE);
00098   sb.nblocks = xint(nblocks);
00099   sb.ninodes = xint(NINODES);
00100   sb.nlog = xint(nlog);
00101   sb.logstart = xint(2);
00102   sb.inodestart = xint(2+nlog);
00103   sb.bmapstart = xint(2+nlog+ninodeblocks);
00104
00105   printf("nmeta %d (boot, super, log blocks %u inode blocks %u, bitmap blocks %u) blocks %d total
      %d\n",
00106          nmeta, nlog, ninodeblocks, nbitmap, nblocks, FSSIZE);
00107
00108   freeblock = nmeta;     // the first free block that we can allocate
00109
00110   for(i = 0; i < FSSIZE; i++)
00111     wsect(i, zeroes);
00112
00113   memset(buf, 0, sizeof(buf));
00114   memmove(buf, &sb, sizeof(sb));
00115   wsect(1, buf);
00116
00117   rootino = ialloc(T_DIR);
00118   assert(rootino == ROOTINO);
00119
00120   bzero(&de, sizeof(de));
00121   de.inum = xshort(rootino);
00122   strcpy(de.name, ".");
00123   iappend(rootino, &de, sizeof(de));
00124
00125   bzero(&de, sizeof(de));
00126   de.inum = xshort(rootino);
00127   strcpy(de.name, "..");
00128   iappend(rootino, &de, sizeof(de));
00129
00130   for(i = 2; i < argc; i++){
00131     assert(index(argv[i], '/') == 0);
00132
00133     if((fd = open(argv[i], 0)) < 0){
00134       perror(argv[i]);
00135       exit(1);
00136     }
00137
00138     // Skip leading _ in name when writing to file system.
00139     // The binaries are named _rm, _cat, etc. to keep the
00140     // build operating system from trying to execute them
00141     // in place of system binaries like rm and cat.
00142     if(argv[i][0] == '_')
00143       ++argv[i];
00144
00145     inum = ialloc(T_FILE);
00146
00147     bzero(&de, sizeof(de));
00148     de.inum = xshort(inum);
00149     strncpy(de.name, argv[i], DIRSIZ);
00150     iappend(rootino, &de, sizeof(de));
00151
00152     while((cc = read(fd, buf, sizeof(buf))) > 0)
00153       iappend(inum, buf, cc);
00154
00155     close(fd);
00156   }
00157
00158   // fix size of root inode dir
00159   rinode(rootino, &din);
00160   off = xint(din.size);
00161   off = ((off/BSIZE) + 1) * BSIZE;
00162   din.size = xint(off);
00163   winode(rootino, &din);
00164
00165   balloc(freeblock);
00166
00167   exit(0);
```

```
00168 }
```

5.125.2.5 rinode()

```
void rinode (
              uint inum,
              struct dinode * ip )
```

Definition at line 198 of file mkfs.c.

```
00199 {
00200   char buf[BSIZE];
00201   uint bn;
00202   struct dinode *dip;
00203
00204   bn = IBLOCK(inum, sb);
00205   rsect(bn, buf);
00206   dip = ((struct dinode*)buf) + (inum % IPB);
00207   *ip = *dip;
00208 }
```

Referenced by iappend(), and main().

5.125.2.6 rsect()

```
void rsect (
              uint sec,
              void * buf )
```

Definition at line 211 of file mkfs.c.

```
00212 {
00213   if(lseek(fsfd, sec * BSIZE, 0) != sec * BSIZE){
00214     perror("lseek");
00215     exit(1);
00216   }
00217   if(read(fsfd, buf, BSIZE) != BSIZE){
00218     perror("read");
00219     exit(1);
00220   }
00221 }
```

Referenced by iappend(), rinode(), and winode().

5.125.2.7 winode()

```
void winode (
              uint inum,
              struct dinode * ip )
```

Definition at line 184 of file mkfs.c.

```
00185 {
00186   char buf[BSIZE];
00187   uint bn;
00188   struct dinode *dip;
00189
00190   bn = IBLOCK(inum, sb);
00191   rsect(bn, buf);
00192   dip = ((struct dinode*)buf) + (inum % IPB);
00193   *dip = *ip;
00194   wsect(bn, buf);
00195 }
```

Referenced by ialloc(), iappend(), and main().

5.125.2.8 wsect()

```
void wsect (
              uint sec,
              void * buf )
```

Definition at line 171 of file mkfs.c.

```
00172 {
00173   if(lseek(fsfd, sec * BSIZE, 0) != sec * BSIZE){
00174     perror("lseek");
00175     exit(1);
00176   }
00177   if(write(fsfd, buf, BSIZE) != BSIZE){
00178     perror("write");
00179     exit(1);
00180   }
00181 }
```

Referenced by balloc(), iappend(), main(), and winode().

5.125.2.9 xint()

```
uint xint (
              uint x )
```

Definition at line 56 of file mkfs.c.

```
00057 {
00058   uint y;
00059   uchar *a = (uchar*)&y;
00060   a[0] = x;
00061   a[1] = x » 8;
00062   a[2] = x » 16;
00063   a[3] = x » 24;
00064   return y;
00065 }
```

Referenced by ialloc(), iappend(), and main().

5.125.2.10 xshort()

```
ushort xshort (
              ushort x )
```

Definition at line 46 of file mkfs.c.

```
00047 {
00048   ushort y;
00049   uchar *a = (uchar*)&y;
00050   a[0] = x;
00051   a[1] = x » 8;
00052   return y;
00053 }
```

Referenced by ialloc(), and main().

5.125.3 Variable Documentation

5.125.3.1 freeblock

```
uint freeblock
```

Definition at line 33 of file mkfs.c.

Referenced by iappend(), and main().

5.125.3.2 freeinode

```
uint freeinode = 1
```

Definition at line 32 of file mkfs.c.

Referenced by ialloc().

5.125.3.3 fsfd

```
int fsfd
```

Definition at line 29 of file mkfs.c.

Referenced by main(), rsect(), and wsect().

5.125.3.4 nbitmap

```
int nbitmap = FSSIZE/(BSIZE*8) + 1
```

Definition at line 23 of file mkfs.c.

Referenced by main().

5.125.3.5 nblocks

```
int nblocks
```

Definition at line 27 of file mkfs.c.

Referenced by main().

5.125.3.6 ninodeblocks

```
int ninodeblocks = NINODES / IPB + 1
```

Definition at line 24 of file mkfs.c.

Referenced by main().

5.125.3.7 nlog

```
int nlog = LOGSIZE
```

Definition at line 25 of file mkfs.c.

Referenced by main().

5.125.3.8 nmeta

```
int nmeta
```

Definition at line 26 of file mkfs.c.

Referenced by main().

5.125.3.9 sb

```
struct superblock sb
```

Definition at line 30 of file mkfs.c.

Referenced by balloc(), main(), rinode(), and winode().

5.125.3.10 zeroes

```
char zeroes[BSIZE]
```

Definition at line 31 of file mkfs.c.

Referenced by main().

5.126 mkfs.c

Go to the documentation of this file.

```
00001 #include <stdio.h>
00002 #include <unistd.h>
00003 #include <stdlib.h>
00004 #include <string.h>
00005 #include <fcntl.h>
00006 #include <assert.h>
00007
00008 #define stat xv6_stat  // avoid clash with host struct stat
00009 #include "types.h"
00010 #include "fs.h"
00011 #include "stat.h"
00012 #include "param.h"
00013
00014 #ifndef static_assert
00015 #define static_assert(a, b) do { switch (0) case 0: case (a): ; } while (0)
00016 #endif
00017
00018 #define NINODES 200
00019
00020 // Disk layout:
00021 // [ boot block | sb block | log | inode blocks | free bit map | data blocks ]
00022
00023 int nbitmap = FSSIZE/(BSIZE*8) + 1;
00024 int ninodeblocks = NINODES / IPB + 1;
00025 int nlog = LOGSIZE;
00026 int nmeta;    // Number of meta blocks (boot, sb, nlog, inode, bitmap)
00027 int nblocks;  // Number of data blocks
00028
00029 int fsfd;
00030 struct superblock sb;
00031 char zeroes[BSIZE];
00032 uint freeinode = 1;
00033 uint freeblock;
00034
00035
00036 void balloc(int);
00037 void wsect(uint, void*);
00038 void winode(uint, struct dinode*);
00039 void rinode(uint inum, struct dinode *ip);
00040 void rsect(uint sec, void *buf);
00041 uint ialloc(ushort type);
00042 void iappend(uint inum, void *p, int n);
00043
00044 // convert to intel byte order
00045 ushort
00046 xshort(ushort x)
00047 {
00048   ushort y;
00049   uchar *a = (uchar*)&y;
00050   a[0] = x;
00051   a[1] = x >> 8;
00052   return y;
00053 }
00054
00055 uint
00056 xint(uint x)
00057 {
00058   uint y;
00059   uchar *a = (uchar*)&y;
00060   a[0] = x;
00061   a[1] = x >> 8;
00062   a[2] = x >> 16;
00063   a[3] = x >> 24;
00064   return y;
00065 }
00066
00067 int
00068 main(int argc, char *argv[])
00069 {
00070   int i, cc, fd;
00071   uint rootino, inum, off;
00072   struct dirent de;
00073   char buf[BSIZE];
00074   struct dinode din;
00075
00076
00077   static_assert(sizeof(int) == 4, "Integers must be 4 bytes!");
00078
00079   if(argc < 2){
00080     fprintf(stderr, "Usage: mkfs fs.img files...\n");
00081     exit(1);
00082   }
```

```
00083
00084   assert((BSIZE % sizeof(struct dinode)) == 0);
00085   assert((BSIZE % sizeof(struct dirent)) == 0);
00086
00087   fsfd = open(argv[1], O_RDWR|O_CREAT|O_TRUNC, 0666);
00088   if(fsfd < 0){
00089     perror(argv[1]);
00090     exit(1);
00091   }
00092
00093   // 1 fs block = 1 disk sector
00094   nmeta = 2 + nlog + ninodeblocks + nbitmap;
00095   nblocks = FSSIZE - nmeta;
00096
00097   sb.size = xint(FSSIZE);
00098   sb.nblocks = xint(nblocks);
00099   sb.ninodes = xint(NINODES);
00100   sb.nlog = xint(nlog);
00101   sb.logstart = xint(2);
00102   sb.inodestart = xint(2+nlog);
00103   sb.bmapstart = xint(2+nlog+ninodeblocks);
00104
00105   printf("nmeta %d (boot, super, log blocks %u inode blocks %u, bitmap blocks %u) blocks %d total
      %d\n",
00106          nmeta, nlog, ninodeblocks, nbitmap, nblocks, FSSIZE);
00107
00108   freeblock = nmeta;     // the first free block that we can allocate
00109
00110   for(i = 0; i < FSSIZE; i++)
00111     wsect(i, zeroes);
00112
00113   memset(buf, 0, sizeof(buf));
00114   memmove(buf, &sb, sizeof(sb));
00115   wsect(1, buf);
00116
00117   rootino = ialloc(T_DIR);
00118   assert(rootino == ROOTINO);
00119
00120   bzero(&de, sizeof(de));
00121   de.inum = xshort(rootino);
00122   strcpy(de.name, ".");
00123   iappend(rootino, &de, sizeof(de));
00124
00125   bzero(&de, sizeof(de));
00126   de.inum = xshort(rootino);
00127   strcpy(de.name, "..");
00128   iappend(rootino, &de, sizeof(de));
00129
00130   for(i = 2; i < argc; i++){
00131     assert(index(argv[i], '/') == 0);
00132
00133     if((fd = open(argv[i], 0)) < 0){
00134       perror(argv[i]);
00135       exit(1);
00136     }
00137
00138     // Skip leading _ in name when writing to file system.
00139     // The binaries are named _rm, _cat, etc. to keep the
00140     // build operating system from trying to execute them
00141     // in place of system binaries like rm and cat.
00142     if(argv[i][0] == '_')
00143       ++argv[i];
00144
00145     inum = ialloc(T_FILE);
00146
00147     bzero(&de, sizeof(de));
00148     de.inum = xshort(inum);
00149     strncpy(de.name, argv[i], DIRSIZ);
00150     iappend(rootino, &de, sizeof(de));
00151
00152     while((cc = read(fd, buf, sizeof(buf))) > 0)
00153       iappend(inum, buf, cc);
00154
00155     close(fd);
00156   }
00157
00158   // fix size of root inode dir
00159   rinode(rootino, &din);
00160   off = xint(din.size);
00161   off = ((off/BSIZE) + 1) * BSIZE;
00162   din.size = xint(off);
00163   winode(rootino, &din);
00164
00165   balloc(freeblock);
00166
00167   exit(0);
00168 }
```

```
00169
00170 void
00171 wsect(uint sec, void *buf)
00172 {
00173   if(lseek(fsfd, sec * BSIZE, 0) != sec * BSIZE){
00174     perror("lseek");
00175     exit(1);
00176   }
00177   if(write(fsfd, buf, BSIZE) != BSIZE){
00178     perror("write");
00179     exit(1);
00180   }
00181 }
00182
00183 void
00184 winode(uint inum, struct dinode *ip)
00185 {
00186   char buf[BSIZE];
00187   uint bn;
00188   struct dinode *dip;
00189
00190   bn = IBLOCK(inum, sb);
00191   rsect(bn, buf);
00192   dip = ((struct dinode*)buf) + (inum % IPB);
00193   *dip = *ip;
00194   wsect(bn, buf);
00195 }
00196
00197 void
00198 rinode(uint inum, struct dinode *ip)
00199 {
00200   char buf[BSIZE];
00201   uint bn;
00202   struct dinode *dip;
00203
00204   bn = IBLOCK(inum, sb);
00205   rsect(bn, buf);
00206   dip = ((struct dinode*)buf) + (inum % IPB);
00207   *ip = *dip;
00208 }
00209
00210 void
00211 rsect(uint sec, void *buf)
00212 {
00213   if(lseek(fsfd, sec * BSIZE, 0) != sec * BSIZE){
00214     perror("lseek");
00215     exit(1);
00216   }
00217   if(read(fsfd, buf, BSIZE) != BSIZE){
00218     perror("read");
00219     exit(1);
00220   }
00221 }
00222
00223 uint
00224 ialloc(ushort type)
00225 {
00226   uint inum = freeinode++;
00227   struct dinode din;
00228
00229   bzero(&din, sizeof(din));
00230   din.type = xshort(type);
00231   din.nlink = xshort(1);
00232   din.size = xint(0);
00233   winode(inum, &din);
00234   return inum;
00235 }
00236
00237 void
00238 balloc(int used)
00239 {
00240   uchar buf[BSIZE];
00241   int i;
00242
00243   printf("balloc: first %d blocks have been allocated\n", used);
00244   assert(used < BSIZE*8);
00245   bzero(buf, BSIZE);
00246   for(i = 0; i < used; i++){
00247     buf[i/8] = buf[i/8] | (0x1 << (i%8));
00248   }
00249   printf("balloc: write bitmap block at sector %d\n", sb.bmapstart);
00250   wsect(sb.bmapstart, buf);
00251 }
00252
00253 #define min(a, b) ((a) < (b) ? (a) : (b))
00254
00255 void
```

```
00256 iappend(uint inum, void *xp, int n)
00257 {
00258   char *p = (char*)xp;
00259   uint fbn, off, n1;
00260   struct dinode din;
00261   char buf[BSIZE];
00262   uint indirect[NINDIRECT];
00263   uint x;
00264
00265   rinode(inum, &din);
00266   off = xint(din.size);
00267   // printf("append inum %d at off %d sz %d\n", inum, off, n);
00268   while(n > 0){
00269     fbn = off / BSIZE;
00270     assert(fbn < MAXFILE);
00271     if(fbn < NDIRECT){
00272       if(xint(din.addrs[fbn]) == 0){
00273         din.addrs[fbn] = xint(freeblock++);
00274       }
00275       x = xint(din.addrs[fbn]);
00276     } else {
00277       if(xint(din.addrs[NDIRECT]) == 0){
00278         din.addrs[NDIRECT] = xint(freeblock++);
00279       }
00280       rsect(xint(din.addrs[NDIRECT]), (char*)indirect);
00281       if(indirect[fbn - NDIRECT] == 0){
00282         indirect[fbn - NDIRECT] = xint(freeblock++);
00283         wsect(xint(din.addrs[NDIRECT]), (char*)indirect);
00284       }
00285       x = xint(indirect[fbn-NDIRECT]);
00286     }
00287     n1 = min(n, (fbn + 1) * BSIZE - off);
00288     rsect(x, buf);
00289     bcopy(p, buf + off - (fbn * BSIZE), n1);
00290     wsect(x, buf);
00291     n -= n1;
00292     off += n1;
00293     p += n1;
00294   }
00295   din.size = xint(off);
00296   winode(inum, &din);
00297 }
```

5.127 mmu.h File Reference

Classes

- struct gatedesc
- struct segdesc
- struct taskstate

Macros

- #define CR0_PE 0x00000001
- #define CR0_PG 0x80000000
- #define CR0_WP 0x00010000
- #define CR4_PSE 0x00000010
- #define DPL_USER 0x3
- #define FL_IF 0x00000200
- #define NPDENTRIES 1024
- #define NPTENTRIES 1024
- #define NSEGS 6
- #define PDX(va) (((uint)(va) >> PDXSHIFT) & 0x3FF)
- #define PDXSHIFT 22
- #define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
- #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
- #define PGROUNDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))

- #define PGSIZE 4096
- #define PTE_ADDR(pte) ((uint)(pte) & ~0xFFF)
- #define PTE_FLAGS(pte) ((uint)(pte) & 0xFFF)
- #define PTE_P 0x001
- #define PTE_PS 0x080
- #define PTE_U 0x004
- #define PTE_W 0x002
- #define PTX(va) (((uint)(va) >> PTXSHIFT) & 0x3FF)
- #define PTXSHIFT 12
- #define SEG(type, base, lim, dpl)
- #define SEG16(type, base, lim, dpl)
- #define SEG_KCODE 1
- #define SEG_KDATA 2
- #define SEG_TSS 5
- #define SEG_UCODE 3
- #define SEG_UDATA 4
- #define SETGATE(gate, istrap, sel, off, d)
- #define STA_R 0x2
- #define STA_W 0x2
- #define STA_X 0x8
- #define STS_IG32 0xE
- #define STS_T32A 0x9
- #define STS_TG32 0xF

Typedefs

- typedef uint pte_t

5.127.1 Macro Definition Documentation

5.127.1.1 CR0_PE

```
#define CR0_PE 0x00000001
```

Definition at line 8 of file mmu.h.

5.127.1.2 CR0_PG

```
#define CR0_PG 0x80000000
```

Definition at line 10 of file mmu.h.

5.127.1.3 CR0_WP

```
#define CR0_WP 0x00010000
```

Definition at line 9 of file mmu.h.

5.127.1.4 CR4_PSE

```
#define CR4_PSE 0x00000010
```

Definition at line 12 of file mmu.h.

5.127.1.5 DPL_USER

```
#define DPL_USER 0x3
```

Definition at line 53 of file mmu.h.

5.127.1.6 FL_IF

```
#define FL_IF 0x00000200
```

Definition at line 5 of file mmu.h.

5.127.1.7 NPDENTRIES

```
#define NPDENTRIES 1024
```

Definition at line 83 of file mmu.h.

5.127.1.8 NPTENTRIES

```
#define NPTENTRIES 1024
```

Definition at line 84 of file mmu.h.

5.127.1.9 NSEGS

```
#define NSEGS 6
```

Definition at line 22 of file mmu.h.

5.127.1.10 PDX

```
#define PDX(
              va ) (((uint)(va) >> PDXSHIFT) & 0x3FF)
```

Definition at line 74 of file mmu.h.

5.127.1.11 PDXSHIFT

```
#define PDXSHIFT 22
```

Definition at line 88 of file mmu.h.

5.127.1.12 PGADDR

```
#define PGADDR(
              d,
              t,
              o ) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
```

Definition at line 80 of file mmu.h.

5.127.1.13 PGROUNDDOWN

```
#define PGROUNDDOWN(
              a ) (((a)) & ~(PGSIZE-1))
```

Definition at line 91 of file mmu.h.

5.127.1.14 PGROUNDUP

```
#define PGROUNDUP(
              sz ) (((sz)+PGSIZE-1) & ~(PGSIZE-1))
```

Definition at line 90 of file mmu.h.

5.127.1.15 PGSIZE

```
#define PGSIZE 4096
```

Definition at line 85 of file mmu.h.

5.127.1.16 PTE_ADDR

```
#define PTE_ADDR(
              pte ) ((uint)(pte) & ~0xFFF)
```

Definition at line 100 of file mmu.h.

5.127.1.17 PTE_FLAGS

```
#define PTE_FLAGS(
              pte ) ((uint)(pte) & 0xFFF)
```

Definition at line 101 of file mmu.h.

5.127.1.18 PTE_P

```
#define PTE_P 0x001
```

Definition at line 94 of file mmu.h.

5.127.1.19 PTE_PS

```
#define PTE_PS 0x080
```

Definition at line 97 of file mmu.h.

5.127.1.20 PTE_U

```
#define PTE_U 0x004
```

Definition at line 96 of file mmu.h.

5.127.1.21 PTE_W

```
#define PTE_W 0x002
```

Definition at line 95 of file mmu.h.

5.127.1.22 PTX

```
#define PTX(
              va ) (((uint)(va) >> PTXSHIFT) & 0x3FF)
```

Definition at line 77 of file mmu.h.

5.127.1.23 PTXSHIFT

```
#define PTXSHIFT 12
```

Definition at line 87 of file mmu.h.

5.127.1.24 SEG

```
#define SEG(
              type,
              base,
              lim,
              dpl )
```

Value:

```
(struct segdesc)    \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
```

Definition at line 43 of file mmu.h.

5.127.1.25 SEG16

```
#define SEG16(
              type,
              base,
              lim,
              dpl )
```

Value:

```
(struct segdesc)  \
{ (lim) & 0xffff, (uint)(base) & 0xffff,              \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }
```

Definition at line 47 of file mmu.h.

5.127.1.26 SEG_KCODE

```
#define SEG_KCODE 1
```

Definition at line 15 of file mmu.h.

5.127.1.27 SEG_KDATA

```
#define SEG_KDATA 2
```

Definition at line 16 of file mmu.h.

5.127.1.28 SEG_TSS

```
#define SEG_TSS 5
```

Definition at line 19 of file mmu.h.

5.127.1.29 SEG_UCODE

```
#define SEG_UCODE 3
```

Definition at line 17 of file mmu.h.

5.127.1.30 SEG_UDATA

```
#define SEG_UDATA 4
```

Definition at line 18 of file mmu.h.

5.127.1.31 SETGATE

```
#define SETGATE(
                gate,
                istrap,
                sel,
                off,
                d )
```

Value:

```
{                                                       \
  (gate).off_15_0 = (uint)(off) & 0xffff;               \
  (gate).cs = (sel);                                    \
  (gate).args = 0;                                      \
  (gate).rsv1 = 0;                                      \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;         \
  (gate).s = 0;                                         \
  (gate).dpl = (d);                                     \
  (gate).p = 1;                                         \
  (gate).off_31_16 = (uint)(off) » 16;                  \
}
```

Definition at line 168 of file mmu.h.

5.127.1.32 STA_R

```
#define STA_R 0x2
```

Definition at line 58 of file mmu.h.

5.127.1.33 STA_W

```
#define STA_W 0x2
```

Definition at line 57 of file mmu.h.

5.127.1.34 STA_X

```
#define STA_X 0x8
```

Definition at line 56 of file mmu.h.

5.127.1.35 STS_IG32

```
#define STS_IG32 0xE
```

Definition at line 62 of file mmu.h.

5.127.1.36 STS_T32A

```
#define STS_T32A 0x9
```

Definition at line 61 of file mmu.h.

5.127.1.37 STS_TG32

```
#define STS_TG32 0xF
```

Definition at line 63 of file mmu.h.

5.127.2 Typedef Documentation

5.127.2.1 pte_t

```
typedef uint pte_t
```

Definition at line 104 of file mmu.h.

5.128 mmu.h

Go to the documentation of this file.

```
00001 // This file contains definitions for the
00002 // x86 memory management unit (MMU).
00003
00004 // Eflags register
00005 #define FL_IF           0x00000200      // Interrupt Enable
00006
00007 // Control Register flags
00008 #define CR0_PE          0x00000001      // Protection Enable
00009 #define CR0_WP          0x00010000      // Write Protect
00010 #define CR0_PG          0x80000000      // Paging
00011
00012 #define CR4_PSE         0x00000010      // Page size extension
00013
00014 // various segment selectors.
00015 #define SEG_KCODE 1  // kernel code
00016 #define SEG_KDATA 2  // kernel data+stack
00017 #define SEG_UCODE 3  // user code
00018 #define SEG_UDATA 4  // user data+stack
00019 #define SEG_TSS   5  // this process's task state
00020
00021 // cpu->gdt[NSEGS] holds the above segments.
00022 #define NSEGS     6
00023
00024 #ifndef __ASSEMBLER__
00025 // Segment Descriptor
00026 struct segdesc {
00027   uint lim_15_0 : 16;  // Low bits of segment limit
00028   uint base_15_0 : 16; // Low bits of segment base address
00029   uint base_23_16 : 8; // Middle bits of segment base address
00030   uint type : 4;       // Segment type (see STS_ constants)
00031   uint s : 1;          // 0 = system, 1 = application
00032   uint dpl : 2;        // Descriptor Privilege Level
00033   uint p : 1;          // Present
00034   uint lim_19_16 : 4;  // High bits of segment limit
```

```
00035   uint avl : 1;        // Unused (available for software use)
00036   uint rsv1 : 1;       // Reserved
00037   uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
00038   uint g : 1;          // Granularity: limit scaled by 4K when set
00039   uint base_31_24 : 8; // High bits of segment base address
00040 };
00041
00042 // Normal segment
00043 #define SEG(type, base, lim, dpl) (struct segdesc)    \
00044 { ((lim) » 12) & 0xffff, (uint)(base) & 0xffff,      \
00045   ((uint)(base) » 16) & 0xff, type, 1, dpl, 1,       \
00046   (uint)(lim) » 28, 0, 0, 1, 1, (uint)(base) » 24 }
00047 #define SEG16(type, base, lim, dpl) (struct segdesc)  \
00048 { (lim) & 0xffff, (uint)(base) & 0xffff,              \
00049   ((uint)(base) » 16) & 0xff, type, 1, dpl, 1,       \
00050   (uint)(lim) » 16, 0, 0, 1, 0, (uint)(base) » 24 }
00051 #endif
00052
00053 #define DPL_USER    0x3     // User DPL
00054
00055 // Application segment type bits
00056 #define STA_X       0x8     // Executable segment
00057 #define STA_W       0x2     // Writeable (non-executable segments)
00058 #define STA_R       0x2     // Readable (executable segments)
00059
00060 // System segment type bits
00061 #define STS_T32A    0x9     // Available 32-bit TSS
00062 #define STS_IG32    0xE     // 32-bit Interrupt Gate
00063 #define STS_TG32    0xF     // 32-bit Trap Gate
00064
00065 // A virtual address 'la' has a three-part structure as follows:
00066 //
00067 // +--------10------+-------10-------+---------12----------+
00068 // | Page Directory |   Page Table   | Offset within Page  |
00069 // |      Index     |      Index     |                     |
00070 // +----------------+----------------+---------------------+
00071 //  \--- PDX(va) --/ \--- PTX(va) --/
00072
00073 // page directory index
00074 #define PDX(va)         (((uint)(va) » PDXSHIFT) & 0x3FF)
00075
00076 // page table index
00077 #define PTX(va)         (((uint)(va) » PTXSHIFT) & 0x3FF)
00078
00079 // construct virtual address from indexes and offset
00080 #define PGADDR(d, t, o) ((uint)((d) « PDXSHIFT | (t) « PTXSHIFT | (o)))
00081
00082 // Page directory and page table constants.
00083 #define NPDENTRIES      1024    // # directory entries per page directory
00084 #define NPTENTRIES      1024    // # PTEs per page table
00085 #define PGSIZE          4096    // bytes mapped by a page
00086
00087 #define PTXSHIFT        12      // offset of PTX in a linear address
00088 #define PDXSHIFT        22      // offset of PDX in a linear address
00089
00090 #define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
00091 #define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
00092
00093 // Page table/directory entry flags.
00094 #define PTE_P           0x001   // Present
00095 #define PTE_W           0x002   // Writeable
00096 #define PTE_U           0x004   // User
00097 #define PTE_PS          0x080   // Page Size
00098
00099 // Address in page table or page directory entry
00100 #define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
00101 #define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
00102
00103 #ifndef __ASSEMBLER__
00104 typedef uint pte_t;
00105
00106 // Task state segment format
00107 struct taskstate {
00108   uint link;         // Old ts selector
00109   uint esp0;         // Stack pointers and segment selectors
00110   ushort ss0;        //   after an increase in privilege level
00111   ushort padding1;
00112   uint *esp1;
00113   ushort ss1;
00114   ushort padding2;
00115   uint *esp2;
00116   ushort ss2;
00117   ushort padding3;
00118   void *cr3;         // Page directory base
00119   uint *eip;         // Saved state from last task switch
00120   uint eflags;
00121   uint eax;          // More saved state (registers)
```

```
00122   uint ecx;
00123   uint edx;
00124   uint ebx;
00125   uint *esp;
00126   uint *ebp;
00127   uint esi;
00128   uint edi;
00129   ushort es;         // Even more saved state (segment selectors)
00130   ushort padding4;
00131   ushort cs;
00132   ushort padding5;
00133   ushort ss;
00134   ushort padding6;
00135   ushort ds;
00136   ushort padding7;
00137   ushort fs;
00138   ushort padding8;
00139   ushort gs;
00140   ushort padding9;
00141   ushort ldt;
00142   ushort padding10;
00143   ushort t;          // Trap on task switch
00144   ushort iomb;       // I/O map base address
00145 };
00146
00147 // Gate descriptors for interrupts and traps
00148 struct gatedesc {
00149   uint off_15_0 : 16;   // low 16 bits of offset in segment
00150   uint cs : 16;         // code segment selector
00151   uint args : 5;        // # args, 0 for interrupt/trap gates
00152   uint rsv1 : 3;        // reserved(should be zero I guess)
00153   uint type : 4;        // type(STS_{IG32,TG32})
00154   uint s : 1;           // must be 0 (system)
00155   uint dpl : 2;         // descriptor(meaning new) privilege level
00156   uint p : 1;           // Present
00157   uint off_31_16 : 16;  // high bits of offset in segment
00158 };
00159
00160 // Set up a normal interrupt/trap gate descriptor.
00161 // - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
00162 //   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
00163 // - sel: Code segment selector for interrupt/trap handler
00164 // - off: Offset in code segment for interrupt/trap handler
00165 // - dpl: Descriptor Privilege Level -
00166 //        the privilege level required for software to invoke
00167 //        this interrupt/trap gate explicitly using an int instruction.
00168 #define SETGATE(gate, istrap, sel, off, d)                \
00169 {                                                         \
00170   (gate).off_15_0 = (uint)(off) & 0xffff;                \
00171   (gate).cs = (sel);                                      \
00172   (gate).args = 0;                                        \
00173   (gate).rsv1 = 0;                                        \
00174   (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
00175   (gate).s = 0;                                           \
00176   (gate).dpl = (d);                                       \
00177   (gate).p = 1;                                           \
00178   (gate).off_31_16 = (uint)(off) >> 16;                  \
00179 }
00180
00181 #endif
```

5.129 mp.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mp.h"
#include "x86.h"
#include "mmu.h"
#include "proc.h"
```

Functions

- static struct mpconf * mpconfig (struct mp **pmp)

- void mpinit (void)
- static struct mp * mpsearch (void)
- static struct mp * mpsearch1 (uint a, int len)
- static uchar sum (uchar *addr, int len)

Variables

- struct cpu cpus [NCPU]
- uchar ioapicid
- int ncpu

5.129.1 Function Documentation

5.129.1.1 mpconfig()

```
static struct mpconf * mpconfig (
            struct mp ** pmp )  [static]
```

Definition at line 73 of file mp.c.

```
00074 {
00075   struct mpconf *conf;
00076   struct mp *mp;
00077
00078   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
00079     return 0;
00080   conf = (struct mpconf*) P2V((uint) mp->physaddr);
00081   if(memcmp(conf, "PCMP", 4) != 0)
00082     return 0;
00083   if(conf->version != 1 && conf->version != 4)
00084     return 0;
00085   if(sum((uchar*)conf, conf->length) != 0)
00086     return 0;
00087   *pmp = mp;
00088   return conf;
00089 }
```

Referenced by mpinit().

5.129.1.2 mpinit()

```
void mpinit (
            void  )
```

Definition at line 92 of file mp.c.

```
00093 {
00094   uchar *p, *e;
00095   int ismp;
00096   struct mp *mp;
00097   struct mpconf *conf;
00098   struct mpproc *proc;
00099   struct mpioapic *ioapic;
00100
00101   if((conf = mpconfig(&mp)) == 0)
00102     panic("Expect to run on an SMP");
00103   ismp = 1;
00104   lapic = (uint*)conf->lapicaddr;
00105   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
```

```
00106     switch(*p){
00107     case MPPROC:
00108       proc = (struct mpproc*)p;
00109       if(ncpu < NCPU) {
00110         cpus[ncpu].apicid = proc->apicid;  // apicid may differ from ncpu
00111         ncpu++;
00112       }
00113       p += sizeof(struct mpproc);
00114       continue;
00115     case MPIOAPIC:
00116       ioapic = (struct mpioapic*)p;
00117       ioapicid = ioapic->apicno;
00118       p += sizeof(struct mpioapic);
00119       continue;
00120     case MPBUS:
00121     case MPIOINTR:
00122     case MPLINTR:
00123       p += 8;
00124       continue;
00125     default:
00126       ismp = 0;
00127       break;
00128     }
00129   }
00130   if(!ismp)
00131     panic("Didn't find a suitable machine");
00132
00133   if(mp->imcrp){
00134     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
00135     // But it would on real hardware.
00136     outb(0x22, 0x70);   // Select IMCR
00137     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
00138   }
00139 }
```

5.129.1.3 mpsearch()

```
static struct mp * mpsearch (
            void  )  [static]
```

Definition at line 49 of file mp.c.

```
00050 {
00051   uchar *bda;
00052   uint p;
00053   struct mp *mp;
00054
00055   bda = (uchar *) P2V(0x400);
00056   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
00057     if((mp = mpsearch1(p, 1024)))
00058       return mp;
00059   } else {
00060     p = ((bda[0x14]<<8)|bda[0x13])*1024;
00061     if((mp = mpsearch1(p-1024, 1024)))
00062       return mp;
00063   }
00064   return mpsearch1(0xF0000, 0x10000);
00065 }
```

Referenced by mpconfig().

5.129.1.4 mpsearch1()

```
static struct mp * mpsearch1 (
            uint a,
            int len )  [static]
```

Definition at line 31 of file mp.c.

```
00032 {
```

```
00033   uchar *e, *p, *addr;
00034
00035   addr = P2V(a);
00036   e = addr+len;
00037   for(p = addr; p < e; p += sizeof(struct mp))
00038     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
00039       return (struct mp*)p;
00040   return 0;
00041 }
```

Referenced by mpsearch().

5.129.1.5 sum()

```
static uchar sum (
             uchar * addr,
             int len )  [static]
```

Definition at line 19 of file mp.c.

```
00020 {
00021   int i, sum;
00022
00023   sum = 0;
00024   for(i=0; i<len; i++)
00025     sum += addr[i];
00026   return sum;
00027 }
```

Referenced by mpconfig(), mpsearch1(), and sum().

5.129.2 Variable Documentation

5.129.2.1 cpus

```
struct cpu cpus[NCPU]
```

Definition at line 14 of file mp.c.

Referenced by cpuid(), mpinit(), mycpu(), seginit(), and startothers().

5.129.2.2 ioapicid

```
uchar ioapicid
```

Definition at line 16 of file mp.c.

Referenced by ioapicinit(), and mpinit().

5.129.2.3 ncpu

```
int ncpu
```

Definition at line 15 of file mp.c.

Referenced by ideinit(), mpinit(), mycpu(), and startothers().

5.130 mp.c

Go to the documentation of this file.

```
00001 // Multiprocessor support
00002 // Search memory for MP description structures.
00003 // http://developer.intel.com/design/pentium/datashts/24201606.pdf
00004
00005 #include "types.h"
00006 #include "defs.h"
00007 #include "param.h"
00008 #include "memlayout.h"
00009 #include "mp.h"
00010 #include "x86.h"
00011 #include "mmu.h"
00012 #include "proc.h"
00013
00014 struct cpu cpus[NCPU];
00015 int ncpu;
00016 uchar ioapicid;
00017
00018 static uchar
00019 sum(uchar *addr, int len)
00020 {
00021   int i, sum;
00022
00023   sum = 0;
00024   for(i=0; i<len; i++)
00025     sum += addr[i];
00026   return sum;
00027 }
00028
00029 // Look for an MP structure in the len bytes at addr.
00030 static struct mp*
00031 mpsearch1(uint a, int len)
00032 {
00033   uchar *e, *p, *addr;
00034
00035   addr = P2V(a);
00036   e = addr+len;
00037   for(p = addr; p < e; p += sizeof(struct mp))
00038     if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
00039       return (struct mp*)p;
00040   return 0;
00041 }
00042
00043 // Search for the MP Floating Pointer Structure, which according to the
00044 // spec is in one of the following three locations:
00045 // 1) in the first KB of the EBDA;
00046 // 2) in the last KB of system base memory;
00047 // 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
00048 static struct mp*
00049 mpsearch(void)
00050 {
00051   uchar *bda;
00052   uint p;
00053   struct mp *mp;
00054
00055   bda = (uchar *) P2V(0x400);
00056   if((p = ((bda[0x0F]<<8)| bda[0x0E]) << 4)){
00057     if((mp = mpsearch1(p, 1024)))
00058       return mp;
00059   } else {
00060     p = ((bda[0x14]<<8)|bda[0x13])*1024;
00061     if((mp = mpsearch1(p-1024, 1024)))
00062       return mp;
00063   }
00064   return mpsearch1(0xF0000, 0x10000);
00065 }
00066
```

```
00067 // Search for an MP configuration table.  For now,
00068 // don't accept the default configurations (physaddr == 0).
00069 // Check for correct signature, calculate the checksum and,
00070 // if correct, check the version.
00071 // To do: check extended table checksum.
00072 static struct mpconf*
00073 mpconfig(struct mp **pmp)
00074 {
00075   struct mpconf *conf;
00076   struct mp *mp;
00077
00078   if((mp = mpsearch()) == 0 || mp->physaddr == 0)
00079     return 0;
00080   conf = (struct mpconf*) P2V((uint) mp->physaddr);
00081   if(memcmp(conf, "PCMP", 4) != 0)
00082     return 0;
00083   if(conf->version != 1 && conf->version != 4)
00084     return 0;
00085   if(sum((uchar*)conf, conf->length) != 0)
00086     return 0;
00087   *pmp = mp;
00088   return conf;
00089 }
00090
00091 void
00092 mpinit(void)
00093 {
00094   uchar *p, *e;
00095   int ismp;
00096   struct mp *mp;
00097   struct mpconf *conf;
00098   struct mpproc *proc;
00099   struct mpioapic *ioapic;
00100
00101   if((conf = mpconfig(&mp)) == 0)
00102     panic("Expect to run on an SMP");
00103   ismp = 1;
00104   lapic = (uint*)conf->lapicaddr;
00105   for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
00106     switch(*p){
00107     case MPPROC:
00108       proc = (struct mpproc*)p;
00109       if(ncpu < NCPU) {
00110         cpus[ncpu].apicid = proc->apicid;  // apicid may differ from ncpu
00111         ncpu++;
00112       }
00113       p += sizeof(struct mpproc);
00114       continue;
00115     case MPIOAPIC:
00116       ioapic = (struct mpioapic*)p;
00117       ioapicid = ioapic->apicno;
00118       p += sizeof(struct mpioapic);
00119       continue;
00120     case MPBUS:
00121     case MPIOINTR:
00122     case MPLINTR:
00123       p += 8;
00124       continue;
00125     default:
00126       ismp = 0;
00127       break;
00128     }
00129   }
00130   if(!ismp)
00131     panic("Didn't find a suitable machine");
00132
00133   if(mp->imcrp){
00134     // Bochs doesn't support IMCR, so this doesn't run on Bochs.
00135     // But it would on real hardware.
00136     outb(0x22, 0x70);   // Select IMCR
00137     outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
00138   }
00139 }
```

5.131 mp.d File Reference

5.132 mp.d

Go to the documentation of this file.

```
00001 mp.o: mp.c /usr/include/stdc-predef.h types.h defs.h param.h memlayout.h \
00002  mp.h x86.h mmu.h proc.h
```

5.133 mp.h File Reference

Classes

- struct mp
- struct mpconf
- struct mpioapic
- struct mpproc

Macros

- #define MPBOOT 0x02
- #define MPBUS 0x01
- #define MPIOAPIC 0x02
- #define MPIOINTR 0x03
- #define MPLINTR 0x04
- #define MPPROC 0x00

5.133.1 Macro Definition Documentation

5.133.1.1 MPBOOT

```
#define MPBOOT 0x02
```

Definition at line 34 of file mp.h.

5.133.1.2 MPBUS

```
#define MPBUS 0x01
```

Definition at line 50 of file mp.h.

5.133.1.3 MPIOAPIC

```
#define MPIOAPIC 0x02
```

Definition at line 51 of file mp.h.

5.133.1.4 MPIOINTR

```
#define MPIOINTR 0x03
```

Definition at line 52 of file mp.h.

5.133.1.5 MPLINTR

```
#define MPLINTR 0x04
```

Definition at line 53 of file mp.h.

5.133.1.6 MPPROC

```
#define MPPROC 0x00
```

Definition at line 49 of file mp.h.

5.134 mp.h

Go to the documentation of this file.

```
00001 // See MultiProcessor Specification Version 1.[14]
00002
00003 struct mp {             // floating pointer
00004   uchar signature[4];           // "_MP_"
00005   void *physaddr;               // phys addr of MP config table
00006   uchar length;                 // 1
00007   uchar specrev;                // [14]
00008   uchar checksum;               // all bytes must add up to 0
00009   uchar type;                   // MP system config type
00010   uchar imcrp;
00011   uchar reserved[3];
00012 };
00013
00014 struct mpconf {         // configuration table header
00015   uchar signature[4];           // "PCMP"
00016   ushort length;                // total table length
00017   uchar version;                // [14]
00018   uchar checksum;               // all bytes must add up to 0
00019   uchar product[20];            // product id
00020   uint *oemtable;               // OEM table pointer
00021   ushort oemlength;             // OEM table length
00022   ushort entry;                 // entry count
00023   uint *lapicaddr;              // address of local APIC
00024   ushort xlength;               // extended table length
00025   uchar xchecksum;              // extended table checksum
00026   uchar reserved;
00027 };
00028
00029 struct mpproc {         // processor table entry
00030   uchar type;                   // entry type (0)
00031   uchar apicid;                 // local APIC id
00032   uchar version;                // local APIC verison
00033   uchar flags;                  // CPU flags
00034     #define MPBOOT 0x02           // This proc is the bootstrap processor.
00035   uchar signature[4];           // CPU signature
00036   uint feature;                 // feature flags from CPUID instruction
00037   uchar reserved[8];
00038 };
00039
00040 struct mpioapic {       // I/O APIC table entry
```

```
00041   uchar type;                  // entry type (2)
00042   uchar apicno;                // I/O APIC id
00043   uchar version;               // I/O APIC version
00044   uchar flags;                 // I/O APIC flags
00045   uint *addr;                 // I/O APIC address
00046 };
00047
00048 // Table entry types
00049 #define MPPROC    0x00  // One per processor
00050 #define MPBUS     0x01  // One per bus
00051 #define MPIOAPIC  0x02  // One per I/O APIC
00052 #define MPIOINTR  0x03  // One per bus interrupt source
00053 #define MPLINTR   0x04  // One per system interrupt source
00054
00055 //PAGEBREAK!
00056 // Blank page.
```

5.135 nice.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
```

Functions

- int main (int argc, char *argv[])

5.135.1 Function Documentation

5.135.1.1 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 7 of file nice.c.

```
00008 {
00009   int priority, pid;
00010   if(argc < 3){
00011     printf(2,"Usage: nice pid priority\n");
00012     exit();
00013   }
00014   pid = atoi(argv[1]);
00015   priority = atoi(argv[2]);
00016   if (priority < 0 || priority > 20){
00017     printf(2,"Invalid priority (0-20)!\n");
00018     exit();
00019   }
00020   chpr(pid, priority);
00021   exit();
00022 }
```

5.136 nice.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004 #include "fcntl.h"
00005
00006 int
00007 main(int argc, char *argv[])
00008 {
00009   int priority, pid;
00010   if(argc < 3){
00011     printf(2,"Usage: nice pid priority\n");
00012     exit();
00013   }
00014   pid = atoi(argv[1]);
00015   priority = atoi(argv[2]);
00016   if (priority < 0 || priority > 20){
00017     printf(2,"Invalid priority (0-20)!\n");
00018     exit();
00019   }
00020   chpr(pid, priority);
00021   exit();
00022 }
```

5.137 nice.d File Reference

5.138 nice.d

Go to the documentation of this file.

```
00001 nice.o: nice.c /usr/include/stdc-predef.h types.h stat.h user.h fcntl.h
```

5.139 param.h File Reference

Macros

- #define FSSIZE 1000
- #define KSTACKSIZE 4096
- #define LOGSIZE (MAXOPBLOCKS∗3)
- #define MAXARG 32
- #define MAXOPBLOCKS 10
- #define NBUF (MAXOPBLOCKS∗3)
- #define NCPU 8
- #define NDEV 10
- #define NFILE 100
- #define NINODE 50
- #define NOFILE 16
- #define NPROC 64
- #define ROOTDEV 1

5.139.1 Macro Definition Documentation

5.139.1.1 FSSIZE

```
#define FSSIZE 1000
```

Definition at line 13 of file param.h.

5.139.1.2 KSTACKSIZE

```
#define KSTACKSIZE 4096
```

Definition at line 2 of file param.h.

5.139.1.3 LOGSIZE

```
#define LOGSIZE (MAXOPBLOCKS*3)
```

Definition at line 11 of file param.h.

5.139.1.4 MAXARG

```
#define MAXARG 32
```

Definition at line 9 of file param.h.

5.139.1.5 MAXOPBLOCKS

```
#define MAXOPBLOCKS 10
```

Definition at line 10 of file param.h.

5.139.1.6 NBUF

```
#define NBUF (MAXOPBLOCKS*3)
```

Definition at line 12 of file param.h.

5.139.1.7 NCPU

```
#define NCPU 8
```

Definition at line 3 of file param.h.

5.139.1.8 NDEV

```
#define NDEV 10
```

Definition at line 7 of file param.h.

5.139.1.9 NFILE

```
#define NFILE 100
```

Definition at line 5 of file param.h.

5.139.1.10 NINODE

```
#define NINODE 50
```

Definition at line 6 of file param.h.

5.139.1.11 NOFILE

```
#define NOFILE 16
```

Definition at line 4 of file param.h.

5.139.1.12 NPROC

```
#define NPROC 64
```

Definition at line 1 of file param.h.

5.139.1.13 ROOTDEV

```
#define ROOTDEV 1
```

Definition at line 8 of file param.h.

5.140 param.h

Go to the documentation of this file.

```
00001 #define NPROC        64  // maximum number of processes
00002 #define KSTACKSIZE 4096  // size of per-process kernel stack
00003 #define NCPU          8  // maximum number of CPUs
00004 #define NOFILE       16  // open files per process
00005 #define NFILE       100  // open files per system
00006 #define NINODE       50  // maximum number of active i-nodes
00007 #define NDEV         10  // maximum major device number
00008 #define ROOTDEV       1  // device number of file system root disk
00009 #define MAXARG       32  // max exec arguments
00010 #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
00011 #define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
00012 #define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
00013 #define FSSIZE       1000  // size of file system in blocks
00014
```

5.141 picirq.c File Reference

```
#include "types.h"
#include "x86.h"
#include "traps.h"
```

Macros

- #define IO_PIC1 0x20
- #define IO_PIC2 0xA0

Functions

- void picinit (void)

5.141.1 Macro Definition Documentation

5.141.1.1 IO_PIC1

```
#define IO_PIC1 0x20
```

Definition at line 6 of file picirq.c.

5.141.1.2 IO_PIC2

```
#define IO_PIC2 0xA0
```

Definition at line 7 of file picirq.c.

5.141.2 Function Documentation

5.141.2.1 picinit()

```
void picinit (
              void  )
```

Definition at line 11 of file picirq.c.

```
00012 {
00013   // mask all interrupts
00014   outb(IO_PIC1+1, 0xFF);
00015   outb(IO_PIC2+1, 0xFF);
00016 }
```

5.142 picirq.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "x86.h"
00003 #include "traps.h"
00004
00005 // I/O Addresses of the two programmable interrupt controllers
00006 #define IO_PIC1         0x20    // Master (IRQs 0-7)
00007 #define IO_PIC2         0xA0    // Slave (IRQs 8-15)
00008
00009 // Don't use the 8259A interrupt controllers.  Xv6 assumes SMP hardware.
00010 void
00011 picinit(void)
00012 {
00013   // mask all interrupts
00014   outb(IO_PIC1+1, 0xFF);
00015   outb(IO_PIC2+1, 0xFF);
00016 }
00017
00018 //PAGEBREAK!
00019 // Blank page.
```

5.143 picirq.d File Reference

5.144 picirq.d

Go to the documentation of this file.

```
00001 picirq.o: picirq.c /usr/include/stdc-predef.h types.h x86.h traps.h
```

5.145 pipe.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "fs.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "file.h"
```

Classes

- struct pipe

Macros

- #define PIPESIZE 512

Functions

- int pipealloc (struct file **f0, struct file **f1)
- void pipeclose (struct pipe *p, int writable)
- int piperead (struct pipe *p, char *addr, int n)
- int pipewrite (struct pipe *p, char *addr, int n)

5.145.1 Macro Definition Documentation

5.145.1.1 PIPESIZE

```
#define PIPESIZE 512
```

Definition at line 11 of file pipe.c.

5.145.2 Function Documentation

5.145.2.1 pipealloc()

```
int pipealloc (
              struct file ** f0,
              struct file ** f1 )
```

Definition at line 23 of file pipe.c.

```
00024 {
00025   struct pipe *p;
00026
00027   p = 0;
00028   *f0 = *f1 = 0;
00029   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
00030     goto bad;
00031   if((p = (struct pipe*)kalloc()) == 0)
00032     goto bad;
00033   p->readopen = 1;
00034   p->writeopen = 1;
00035   p->nwrite = 0;
00036   p->nread = 0;
00037   initlock(&p->lock, "pipe");
00038   (*f0)->type = FD_PIPE;
00039   (*f0)->readable = 1;
00040   (*f0)->writable = 0;
00041   (*f0)->pipe = p;
00042   (*f1)->type = FD_PIPE;
00043   (*f1)->readable = 0;
00044   (*f1)->writable = 1;
00045   (*f1)->pipe = p;
00046   return 0;
00047
00048 //PAGEBREAK: 20
00049  bad:
00050   if(p)
00051     kfree((char*)p);
00052   if(*f0)
00053     fileclose(*f0);
00054   if(*f1)
00055     fileclose(*f1);
00056   return -1;
00057 }
```

Referenced by sys_pipe().

5.145.2.2 pipeclose()

```
void pipeclose (
              struct pipe * p,
              int writable )
```

Definition at line 60 of file pipe.c.

```
00061 {
00062   acquire(&p->lock);
00063   if(writable){
00064     p->writeopen = 0;
00065     wakeup(&p->nread);
00066   } else {
00067     p->readopen = 0;
00068     wakeup(&p->nwrite);
00069   }
00070   if(p->readopen == 0 && p->writeopen == 0){
00071     release(&p->lock);
00072     kfree((char*)p);
00073   } else
00074     release(&p->lock);
00075 }
```

Referenced by fileclose().

5.145.2.3 piperead()

```
int piperead (
              struct pipe * p,
              char * addr,
              int n )
```

Definition at line 101 of file pipe.c.

```
00102 {
00103   int i;
00104
00105   acquire(&p->lock);
00106   while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
00107     if(myproc()->killed){
00108       release(&p->lock);
00109       return -1;
00110     }
00111     sleep(&p->nread, &p->lock); //DOC: piperead-sleep
00112   }
00113   for(i = 0; i < n; i++){  //DOC: piperead-copy
00114     if(p->nread == p->nwrite)
00115       break;
00116     addr[i] = p->data[p->nread++ % PIPESIZE];
00117   }
00118   wakeup(&p->nwrite);  //DOC: piperead-wakeup
00119   release(&p->lock);
00120   return i;
00121 }
```

Referenced by fileread().

5.145.2.4 pipewrite()

```
int pipewrite (
              struct pipe * p,
              char * addr,
              int n )
```

Definition at line 79 of file pipe.c.

```
00080 {
00081   int i;
00082
00083   acquire(&p->lock);
00084   for(i = 0; i < n; i++){
00085     while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
00086       if(p->readopen == 0 || myproc()->killed){
00087         release(&p->lock);
00088         return -1;
00089       }
00090       wakeup(&p->nread);
00091       sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
00092     }
00093     p->data[p->nwrite++ % PIPESIZE] = addr[i];
00094   }
00095   wakeup(&p->nread);  //DOC: pipewrite-wakeup1
00096   release(&p->lock);
00097   return n;
00098 }
```

Referenced by filewrite().

5.146 pipe.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "defs.h"
00003 #include "param.h"
00004 #include "mmu.h"
00005 #include "proc.h"
00006 #include "fs.h"
00007 #include "spinlock.h"
00008 #include "sleeplock.h"
00009 #include "file.h"
00010
00011 #define PIPESIZE 512
00012
00013 struct pipe {
00014   struct spinlock lock;
00015   char data[PIPESIZE];
00016   uint nread;     // number of bytes read
00017   uint nwrite;    // number of bytes written
00018   int readopen;   // read fd is still open
00019   int writeopen;  // write fd is still open
00020 };
00021
00022 int
00023 pipealloc(struct file **f0, struct file **f1)
00024 {
00025   struct pipe *p;
00026
00027   p = 0;
00028   *f0 = *f1 = 0;
00029   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
00030     goto bad;
00031   if((p = (struct pipe*)kalloc()) == 0)
00032     goto bad;
00033   p->readopen = 1;
00034   p->writeopen = 1;
00035   p->nwrite = 0;
00036   p->nread = 0;
00037   initlock(&p->lock, "pipe");
00038   (*f0)->type = FD_PIPE;
00039   (*f0)->readable = 1;
00040   (*f0)->writable = 0;
00041   (*f0)->pipe = p;
00042   (*f1)->type = FD_PIPE;
00043   (*f1)->readable = 0;
00044   (*f1)->writable = 1;
00045   (*f1)->pipe = p;
00046   return 0;
00047
00048 //PAGEBREAK: 20
00049  bad:
00050   if(p)
00051     kfree((char*)p);
00052   if(*f0)
00053     fileclose(*f0);
00054   if(*f1)
00055     fileclose(*f1);
00056   return -1;
00057 }
00058
00059 void
00060 pipeclose(struct pipe *p, int writable)
00061 {
00062   acquire(&p->lock);
00063   if(writable){
00064     p->writeopen = 0;
00065     wakeup(&p->nread);
00066   } else {
00067     p->readopen = 0;
00068     wakeup(&p->nwrite);
00069   }
00070   if(p->readopen == 0 && p->writeopen == 0){
00071     release(&p->lock);
00072     kfree((char*)p);
00073   } else
00074     release(&p->lock);
00075 }
00076
00077 //PAGEBREAK: 40
00078 int
00079 pipewrite(struct pipe *p, char *addr, int n)
00080 {
00081   int i;
00082
```

```
00083   acquire(&p->lock);
00084   for(i = 0; i < n; i++){
00085     while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
00086       if(p->readopen == 0 || myproc()->killed){
00087         release(&p->lock);
00088         return -1;
00089       }
00090       wakeup(&p->nread);
00091       sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
00092     }
00093     p->data[p->nwrite++ % PIPESIZE] = addr[i];
00094   }
00095   wakeup(&p->nread);  //DOC: pipewrite-wakeup1
00096   release(&p->lock);
00097   return n;
00098 }
00099
00100 int
00101 piperead(struct pipe *p, char *addr, int n)
00102 {
00103   int i;
00104
00105   acquire(&p->lock);
00106   while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
00107     if(myproc()->killed){
00108       release(&p->lock);
00109       return -1;
00110     }
00111     sleep(&p->nread, &p->lock); //DOC: piperead-sleep
00112   }
00113   for(i = 0; i < n; i++){  //DOC: piperead-copy
00114     if(p->nread == p->nwrite)
00115       break;
00116     addr[i] = p->data[p->nread++ % PIPESIZE];
00117   }
00118   wakeup(&p->nwrite);  //DOC: piperead-wakeup
00119   release(&p->lock);
00120   return i;
00121 }
```

5.147 pipe.d File Reference

5.148 pipe.d

Go to the documentation of this file.

```
00001 pipe.o: pipe.c /usr/include/stdc-predef.h types.h defs.h param.h mmu.h \
00002  proc.h fs.h spinlock.h sleeplock.h file.h
```

5.149 printf.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- void printf (int fd, const char *fmt,...)
- static void printint (int fd, int xx, int base, int sgn)
- static void putc (int fd, char c)

5.149.1 Function Documentation

5.149.1.1 printf()

```
void printf (
             int fd,
             const char * fmt,
              ...  )
```

Definition at line 40 of file printf.c.

```
00041 {
00042   char *s;
00043   int c, i, state;
00044   uint *ap;
00045
00046   state = 0;
00047   ap = (uint*)(void*)&fmt + 1;
00048   for(i = 0; fmt[i]; i++){
00049     c = fmt[i] & 0xff;
00050     if(state == 0){
00051       if(c == '%'){
00052         state = '%';
00053       } else {
00054         putc(fd, c);
00055       }
00056     } else if(state == '%'){
00057       if(c == 'd'){
00058         printint(fd, *ap, 10, 1);
00059         ap++;
00060       } else if(c == 'x' || c == 'p'){
00061         printint(fd, *ap, 16, 0);
00062         ap++;
00063       } else if(c == 's'){
00064         s = (char*)*ap;
00065         ap++;
00066         if(s == 0)
00067           s = "(null)";
00068         while(*s != 0){
00069           putc(fd, *s);
00070           s++;
00071         }
00072       } else if(c == 'c'){
00073         putc(fd, *ap);
00074         ap++;
00075       } else if(c == '%'){
00076         putc(fd, c);
00077       } else {
00078         // Unknown % sequence.  Print it to draw attention.
00079         putc(fd, '%');
00080         putc(fd, c);
00081       }
00082       state = 0;
00083     }
00084   }
00085 }
```

5.149.1.2 printint()

```
static void printint (
             int fd,
             int xx,
             int base,
             int sgn )  [static]
```

Definition at line 12 of file printf.c.

```
00013 {
```

```
00014   static char digits[] = "0123456789ABCDEF";
00015   char buf[16];
00016   int i, neg;
00017   uint x;
00018
00019   neg = 0;
00020   if(sgn && xx < 0){
00021     neg = 1;
00022     x = -xx;
00023   } else {
00024     x = xx;
00025   }
00026
00027   i = 0;
00028   do{
00029     buf[i++] = digits[x % base];
00030   }while((x /= base) != 0);
00031   if(neg)
00032     buf[i++] = '-';
00033
00034   while(--i >= 0)
00035     putc(fd, buf[i]);
00036 }
```

Referenced by printf().

5.149.1.3 putc()

```
static void putc (
              int fd,
              char c )  [static]
```

Definition at line 6 of file printf.c.

```
00007 {
00008   write(fd, &c, 1);
00009 }
```

Referenced by printf(), and printint().

5.150 printf.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004
00005 static void
00006 putc(int fd, char c)
00007 {
00008   write(fd, &c, 1);
00009 }
00010
00011 static void
00012 printint(int fd, int xx, int base, int sgn)
00013 {
00014   static char digits[] = "0123456789ABCDEF";
00015   char buf[16];
00016   int i, neg;
00017   uint x;
00018
00019   neg = 0;
00020   if(sgn && xx < 0){
00021     neg = 1;
00022     x = -xx;
00023   } else {
00024     x = xx;
00025   }
00026
00027   i = 0;
00028   do{
```

```
00029     buf[i++] = digits[x % base];
00030   }while((x /= base) != 0);
00031   if(neg)
00032     buf[i++] = '-';
00033
00034   while(--i >= 0)
00035     putc(fd, buf[i]);
00036 }
00037
00038 // Print to the given fd. Only understands %d, %x, %p, %s.
00039 void
00040 printf(int fd, const char *fmt, ...)
00041 {
00042   char *s;
00043   int c, i, state;
00044   uint *ap;
00045
00046   state = 0;
00047   ap = (uint*)(void*)&fmt + 1;
00048   for(i = 0; fmt[i]; i++){
00049     c = fmt[i] & 0xff;
00050     if(state == 0){
00051       if(c == '%'){
00052         state = '%';
00053       } else {
00054         putc(fd, c);
00055       }
00056     } else if(state == '%'){
00057       if(c == 'd'){
00058         printint(fd, *ap, 10, 1);
00059         ap++;
00060       } else if(c == 'x' || c == 'p'){
00061         printint(fd, *ap, 16, 0);
00062         ap++;
00063       } else if(c == 's'){
00064         s = (char*)*ap;
00065         ap++;
00066         if(s == 0)
00067           s = "(null)";
00068         while(*s != 0){
00069           putc(fd, *s);
00070           s++;
00071         }
00072       } else if(c == 'c'){
00073         putc(fd, *ap);
00074         ap++;
00075       } else if(c == '%'){
00076         putc(fd, c);
00077       } else {
00078         // Unknown % sequence.  Print it to draw attention.
00079         putc(fd, '%');
00080         putc(fd, c);
00081       }
00082       state = 0;
00083     }
00084   }
00085 }
```

5.151 printf.d File Reference

5.152 printf.d

Go to the documentation of this file.

```
00001 printf.o: printf.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.153 proc.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
```

```
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"
```

Functions

- static struct proc * allocproc (void)
- int chpr (int pid, int priority)
- int cps ()
- int cpuid ()
- void exit (void)
- int fork (void)
- void forkret (void)
- int growproc (int n)
- int kill (int pid)
- struct cpu * mycpu (void)
- struct proc * myproc (void)
- void pinit (void)
- void procdump (void)
- void sched (void)
- void scheduler (void)
- void sleep (void *chan, struct spinlock *lk)
- void trapret (void)
- void userinit (void)
- int wait (void)
- void wakeup (void *chan)
- static void wakeup1 (void *chan)
- void yield (void)

Variables

- static struct proc * initproc
- int nextpid = 1
- struct {
  struct spinlock lock
  struct proc proc [NPROC]
  } ptable

5.153.1 Function Documentation

5.153.1.1 allocproc()

```
static struct proc * allocproc (
            void  )  [static]
```

Definition at line 74 of file proc.c.

```
00075 {
00076   struct proc *p;
00077   char *sp;
00078
00079   acquire(&ptable.lock);
00080
00081   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
00082     if(p->state == UNUSED)
00083       goto found;
00084
00085   release(&ptable.lock);
00086   return 0;
00087
00088 found:
00089   p->state = EMBRYO;
00090   p->pid = nextpid++;
00091
00092   release(&ptable.lock);
00093
00094   // Allocate kernel stack.
00095   if((p->kstack = kalloc()) == 0){
00096     p->state = UNUSED;
00097     return 0;
00098   }
00099   sp = p->kstack + KSTACKSIZE;
00100
00101   // Leave room for trap frame.
00102   sp -= sizeof *p->tf;
00103   p->tf = (struct trapframe*)sp;
00104
00105   // Set up new context to start executing at forkret,
00106   // which returns to trapret.
00107   sp -= 4;
00108   *(uint*)sp = (uint)trapret;
00109
00110   sp -= sizeof *p->context;
00111   p->context = (struct context*)sp;
00112   memset(p->context, 0, sizeof *p->context);
00113   p->context->eip = (uint)forkret;
00114
00115   return p;
00116 }
```

Referenced by fork(), and userinit().

5.153.1.2 chpr()

```
int chpr (
            int pid,
            int priority )
```

Definition at line 559 of file proc.c.

```
00560 {
00561   struct proc *p;
00562   acquire(&ptable.lock);
00563   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00564     if(p->pid == pid){
00565       p->priority = priority;
00566       break;
00567     }
00568   }
00569   release(&ptable.lock);
00570   return pid;
00571 }
```

Referenced by main(), and sys_chpr().

5.153.1.3 cps()

```
int cps (
              void  )
```

Definition at line 537 of file proc.c.

```
00538 {
00539 struct proc *p;
00540 //Enables interrupts on this processor.
00541 sti();
00542
00543 //Loop over process table looking for process with pid.
00544 acquire(&ptable.lock);
00545 cprintf("name \t pid \t state \t priority \n");
00546 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00547   if(p->state == SLEEPING)
00548     cprintf("%s \t %d \t SLEEPING \t %d \n ", p->name,p->pid,p->priority);
00549   else if(p->state == RUNNING)
00550     cprintf("%s \t %d \t RUNNING \t %d \n ", p->name,p->pid,p->priority);
00551   else if(p->state == RUNNABLE)
00552     cprintf("%s \t %d \t RUNNABLE \t %d \n ", p->name,p->pid,p->priority);
00553 }
00554 release(&ptable.lock);
00555 return 23;
00556 }
```

Referenced by main(), and sys_cps().

5.153.1.4 cpuid()

```
int cpuid (
              void  )
```

Definition at line 31 of file proc.c.

```
00031         {
00032   return mycpu()-cpus;
00033 }
```

Referenced by seginit(), and trap().

5.153.1.5 exit()

```
void exit (
              void  )
```

Definition at line 228 of file proc.c.

```
00229 {
00230   struct proc *curproc = myproc();
00231   struct proc *p;
00232   int fd;
00233
00234   if(curproc == initproc)
00235     panic("init exiting");
00236
00237   // Close all open files.
00238   for(fd = 0; fd < NOFILE; fd++){
00239     if(curproc->ofile[fd]){
00240       fileclose(curproc->ofile[fd]);
00241       curproc->ofile[fd] = 0;
00242     }
00243   }
00244
00245   begin_op();
00246   iput(curproc->cwd);
```

```
00247   end_op();
00248   curproc->cwd = 0;
00249
00250   acquire(&ptable.lock);
00251
00252   // Parent might be sleeping in wait().
00253   wakeup1(curproc->parent);
00254
00255   // Pass abandoned children to init.
00256   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00257     if(p->parent == curproc){
00258       p->parent = initproc;
00259       if(p->state == ZOMBIE)
00260         wakeup1(initproc);
00261     }
00262   }
00263
00264   // Jump into the scheduler, never to return.
00265   curproc->state = ZOMBIE;
00266   sched();
00267   panic("zombie exit");
00268 }
```

Referenced by argptest(), bigargtest(), bigdir(), bigfile(), bigwrite(), bsstest(), cat(), concreate(), createdelete(), dirfile(), dirtest(), exectest(), exitiputtest(), exitwait(), forktest(), fourfiles(), fourteen(), iputtest(), iref(), linktest(), linkunlink(), main(), mem(), openiputtest(), opentest(), panic(), pipe1(), rmdot(), rsect(), runcmd(), sbrktest(), sharedfd(), subdir(), sys_exit(), trap(), uio(), unlinkread(), validatetest(), wc(), writetest(), writetest1(), and wsect().

5.153.1.6 fork()

```
int fork (
              void  )
```

Definition at line 181 of file proc.c.

```
00182 {
00183   int i, pid;
00184   struct proc *np;
00185   struct proc *curproc = myproc();
00186
00187   // Allocate process.
00188   if((np = allocproc()) == 0){
00189     return -1;
00190   }
00191
00192   // Copy process state from proc.
00193   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
00194     kfree(np->kstack);
00195     np->kstack = 0;
00196     np->state = UNUSED;
00197     return -1;
00198   }
00199   np->sz = curproc->sz;
00200   np->parent = curproc;
00201   *np->tf = *curproc->tf;
00202
00203   // Clear %eax so that fork returns 0 in the child.
00204   np->tf->eax = 0;
00205
00206   for(i = 0; i < NOFILE; i++)
00207     if(curproc->ofile[i])
00208       np->ofile[i] = filedup(curproc->ofile[i]);
00209   np->cwd = idup(curproc->cwd);
00210
00211   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
00212
00213   pid = np->pid;
00214
00215   acquire(&ptable.lock);
00216
00217   np->state = RUNNABLE;
00218
00219   release(&ptable.lock);
00220
00221   return pid;
00222 }
```

Referenced by bigargtest(), concreate(), createdelete(), exitiputtest(), exitwait(), fork1(), forktest(), fourfiles(), linkunlink(), main(), mem(), openiputtest(), pipe1(), preempt(), sbrktest(), sharedfd(), sys_fork(), uio(), and validatetest().

5.153.1.7 forkret()

```
void forkret (
              void  )
```

Definition at line 397 of file proc.c.

```
00398 {
00399   static int first = 1;
00400   // Still holding ptable.lock from scheduler.
00401   release(&ptable.lock);
00402
00403   if (first) {
00404     // Some initialization functions must be run in the context
00405     // of a regular process (e.g., they call sleep), and thus cannot
00406     // be run from main().
00407     first = 0;
00408     iinit(ROOTDEV);
00409     initlog(ROOTDEV);
00410   }
00411
00412   // Return to "caller", actually trapret (see allocproc).
00413 }
```

Referenced by allocproc().

5.153.1.8 growproc()

```
int growproc (
              int n )
```

Definition at line 159 of file proc.c.

```
00160 {
00161   uint sz;
00162   struct proc *curproc = myproc();
00163
00164   sz = curproc->sz;
00165   if(n > 0){
00166     if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
00167       return -1;
00168   } else if(n < 0){
00169     if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
00170       return -1;
00171   }
00172   curproc->sz = sz;
00173   switchuvm(curproc);
00174   return 0;
00175 }
```

Referenced by sys_sbrk().

5.153.1.9 kill()

```
int kill (
              int pid )
```

Definition at line 480 of file proc.c.

```
00481 {
00482   struct proc *p;
00483
00484   acquire(&ptable.lock);
00485   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00486     if(p->pid == pid){
00487       p->killed = 1;
00488       // Wake process from sleep if necessary.
00489       if(p->state == SLEEPING)
00490         p->state = RUNNABLE;
00491       release(&ptable.lock);
00492       return 0;
00493     }
00494   }
00495   release(&ptable.lock);
00496   return -1;
00497 }
```

Referenced by main(), mem(), preempt(), sbrktest(), sys_kill(), and validatetest().

5.153.1.10 mycpu()

```
struct cpu * mycpu (
              void  )
```

Definition at line 38 of file proc.c.

```
00039 {
00040   int apicid, i;
00041
00042   if(readeflags()&FL_IF)
00043     panic("mycpu called with interrupts enabled\n");
00044
00045   apicid = lapicid();
00046   // APIC IDs are not guaranteed to be contiguous. Maybe we should have
00047   // a reverse map, or reserve a register to store &cpus[i].
00048   for (i = 0; i < ncpu; ++i) {
00049     if (cpus[i].apicid == apicid)
00050       return &cpus[i];
00051   }
00052   panic("unknown apicid\n");
00053 }
```

Referenced by acquire(), cpuid(), holding(), myproc(), popcli(), pushcli(), sched(), scheduler(), startothers(), and switchuvm().

5.153.1.11 myproc()

```
struct proc * myproc (
              void  )
```

Definition at line 58 of file proc.c.

```
00058               {
00059   struct cpu *c;
00060   struct proc *p;
00061   pushcli();
00062   c = mycpu();
00063   p = c->proc;
00064   popcli();
00065   return p;
00066 }
```

Referenced by acquiresleep(), argfd(), argint(), argptr(), consoleread(), exec(), exit(), fdalloc(), fetchint(), fetchstr(), fork(), growproc(), holdingsleep(), namex(), piperead(), pipewrite(), sched(), sleep(), sys_chdir(), sys_close(), sys_getpid(), sys_pipe(), sys_sbrk(), sys_sleep(), syscall(), trap(), wait(), and yield().

5.153.1.12 pinit()

```
void pinit (
              void  )
```

Definition at line 24 of file proc.c.

```
00025 {
00026   initlock(&ptable.lock, "ptable");
00027 }
```

5.153.1.13 procdump()

```
void procdump (
              void  )
```

Definition at line 504 of file proc.c.

```
00505 {
00506   static char *states[] = {
00507   [UNUSED]    "unused",
00508   [EMBRYO]    "embryo",
00509   [SLEEPING]  "sleep ",
00510   [RUNNABLE]  "runble",
00511   [RUNNING]   "run   ",
00512   [ZOMBIE]    "zombie"
00513   };
00514   int i;
00515   struct proc *p;
00516   char *state;
00517   uint pc[10];
00518
00519   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00520     if(p->state == UNUSED)
00521       continue;
00522     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
00523       state = states[p->state];
00524     else
00525       state = "???";
00526     cprintf("%d %s %s", p->pid, state, p->name);
00527     if(p->state == SLEEPING){
00528       getcallerpcs((uint*)p->context->ebp+2, pc);
00529       for(i=0; i<10 && pc[i] != 0; i++)
00530         cprintf(" %p", pc[i]);
00531     }
00532     cprintf("\n");
00533   }
00534 }
```

Referenced by consoleintr().

5.153.1.14 sched()

```
void sched (
              void  )
```

Definition at line 366 of file proc.c.

```
00367 {
00368   int intena;
00369   struct proc *p = myproc();
00370
00371   if(!holding(&ptable.lock))
00372     panic("sched ptable.lock");
00373   if(mycpu()->ncli != 1)
00374     panic("sched locks");
00375   if(p->state == RUNNING)
00376     panic("sched running");
00377   if(readeflags()&FL_IF)
00378     panic("sched interruptible");
00379   intena = mycpu()->intena;
00380   swtch(&p->context, mycpu()->scheduler);
00381   mycpu()->intena = intena;
00382 }
```

Referenced by exit(), sleep(), and yield().

5.153.1.15 scheduler()

```
void scheduler (
            void  )
```

Definition at line 323 of file proc.c.

```
00324 {
00325   struct proc *p;
00326   struct cpu *c = mycpu();
00327   c->proc = 0;
00328
00329   for(;;){
00330     // Enable interrupts on this processor.
00331     sti();
00332
00333     // Loop over process table looking for process to run.
00334     acquire(&ptable.lock);
00335     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00336       if(p->state != RUNNABLE)
00337         continue;
00338
00339       // Switch to chosen process.  It is the process's job
00340       // to release ptable.lock and then reacquire it
00341       // before jumping back to us.
00342       c->proc = p;
00343       switchuvm(p);
00344       p->state = RUNNING;
00345
00346       swtch(&(c->scheduler), p->context);
00347       switchkvm();
00348
00349       // Process is done running for now.
00350       // It should have changed its p->state before coming back.
00351       c->proc = 0;
00352     }
00353     release(&ptable.lock);
00354
00355   }
00356 }
```

Referenced by sched().

5.153.1.16 sleep()

```
void sleep (
            void * chan,
            struct spinlock * lk )
```

Definition at line 418 of file proc.c.

```
00419 {
00420   struct proc *p = myproc();
00421
00422   if(p == 0)
00423     panic("sleep");
00424
00425   if(lk == 0)
00426     panic("sleep without lk");
00427
00428   // Must acquire ptable.lock in order to
00429   // change p->state and then call sched.
00430   // Once we hold ptable.lock, we can be
00431   // guaranteed that we won't miss any wakeup
00432   // (wakeup runs with ptable.lock locked),
00433   // so it's okay to release lk.
00434   if(lk != &ptable.lock){  //DOC: sleeplock0
00435     acquire(&ptable.lock);  //DOC: sleeplock1
00436     release(lk);
00437   }
00438   // Go to sleep.
00439   p->chan = chan;
00440   p->state = SLEEPING;
00441
00442   sched();
```

```
00443
00444   // Tidy up.
00445   p->chan = 0;
00446
00447   // Reacquire original lock.
00448   if(lk != &ptable.lock){  //DOC: sleeplock2
00449     release(&ptable.lock);
00450     acquire(lk);
00451   }
00452 }
```

Referenced by acquiresleep(), begin_op(), consoleread(), iderw(), main(), openiputtest(), piperead(), pipewrite(), sbrktest(), sys_sleep(), validatetest(), and wait().

5.153.1.17 trapret()

```
void trapret (
              void  )
```

Referenced by allocproc().

5.153.1.18 userinit()

```
void userinit (
              void  )
```

Definition at line 121 of file proc.c.

```
00122 {
00123   struct proc *p;
00124   extern char _binary_initcode_start[], _binary_initcode_size[];
00125
00126   p = allocproc();
00127
00128   initproc = p;
00129   if((p->pgdir = setupkvm()) == 0)
00130     panic("userinit: out of memory?");
00131   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
00132   p->sz = PGSIZE;
00133   memset(p->tf, 0, sizeof(*p->tf));
00134   p->tf->cs = (SEG_UCODE « 3) | DPL_USER;
00135   p->tf->ds = (SEG_UDATA « 3) | DPL_USER;
00136   p->tf->es = p->tf->ds;
00137   p->tf->ss = p->tf->ds;
00138   p->tf->eflags = FL_IF;
00139   p->tf->esp = PGSIZE;
00140   p->tf->eip = 0;  // beginning of initcode.S
00141
00142   safestrcpy(p->name, "initcode", sizeof(p->name));
00143   p->cwd = namei("/");
00144
00145   // this assignment to p->state lets other cores
00146   // run this process. the acquire forces the above
00147   // writes to be visible, and the lock is also needed
00148   // because the assignment might not be atomic.
00149   acquire(&ptable.lock);
00150
00151   p->state = RUNNABLE;
00152
00153   release(&ptable.lock);
00154 }
```

5.153.1.19 wait()

```
int wait (
              void  )
```

Definition at line 273 of file proc.c.

```
00274 {
00275   struct proc *p;
00276   int havekids, pid;
00277   struct proc *curproc = myproc();
00278
00279   acquire(&ptable.lock);
00280   for(;;){
00281     // Scan through table looking for exited children.
00282     havekids = 0;
00283     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00284       if(p->parent != curproc)
00285         continue;
00286       havekids = 1;
00287       if(p->state == ZOMBIE){
00288         // Found one.
00289         pid = p->pid;
00290         kfree(p->kstack);
00291         p->kstack = 0;
00292         freevm(p->pgdir);
00293         p->pid = 0;
00294         p->parent = 0;
00295         p->name[0] = 0;
00296         p->killed = 0;
00297         p->state = UNUSED;
00298         release(&ptable.lock);
00299         return pid;
00300       }
00301     }
00302
00303     // No point waiting if we don't have any children.
00304     if(!havekids || curproc->killed){
00305       release(&ptable.lock);
00306       return -1;
00307     }
00308
00309     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
00310     sleep(curproc, &ptable.lock);  //DOC: wait-sleep
00311   }
00312 }
```

Referenced by bigargtest(), concreate(), createdelete(), exitiputtest(), exitwait(), forktest(), fourfiles(), linkunlink(), main(), mem(), openiputtest(), pipe1(), preempt(), runcmd(), sbrktest(), sharedfd(), sys_wait(), uio(), and validatetest().

5.153.1.20 wakeup()

```
void wakeup (
              void * chan )
```

Definition at line 469 of file proc.c.

```
00470 {
00471   acquire(&ptable.lock);
00472   wakeup1(chan);
00473   release(&ptable.lock);
00474 }
```

Referenced by consoleintr(), end_op(), ideintr(), pipeclose(), piperead(), pipewrite(), releasesleep(), and trap().

5.153.1.21 wakeup1()

```
static void wakeup1 (
             void * chan )  [static]
```

Definition at line 458 of file proc.c.

```
00459 {
00460   struct proc *p;
00461
00462   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
00463     if(p->state == SLEEPING && p->chan == chan)
00464       p->state = RUNNABLE;
00465 }
```

Referenced by exit(), and wakeup().

5.153.1.22 yield()

```
void yield (
             void  )
```

Definition at line 386 of file proc.c.

```
00387 {
00388   acquire(&ptable.lock);  //DOC: yieldlock
00389   myproc()->state = RUNNABLE;
00390   sched();
00391   release(&ptable.lock);
00392 }
```

Referenced by trap().

5.153.2 Variable Documentation

5.153.2.1 initproc

```
struct proc* initproc  [static]
```

Definition at line 15 of file proc.c.

Referenced by exit(), and userinit().

5.153.2.2 lock

```
struct spinlock lock
```

Definition at line 11 of file proc.c.

5.153.2.3 nextpid

```
int nextpid = 1
```

Definition at line 17 of file proc.c.

Referenced by allocproc().

5.153.2.4 proc

```
struct proc proc[NPROC]
```

Definition at line 12 of file proc.c.

5.153.2.5

```
struct { ... } ptable
```

Referenced by allocproc(), chpr(), cps(), exit(), fork(), forkret(), kill(), pinit(), procdump(), sched(), scheduler(), sleep(), userinit(), wait(), wakeup(), wakeup1(), and yield().

5.154 proc.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "defs.h"
00003 #include "param.h"
00004 #include "memlayout.h"
00005 #include "mmu.h"
00006 #include "x86.h"
00007 #include "proc.h"
00008 #include "spinlock.h"
00009
00010 struct {
00011   struct spinlock lock;
00012   struct proc proc[NPROC];
00013 } ptable;
00014
00015 static struct proc *initproc;
00016
00017 int nextpid = 1;
00018 extern void forkret(void);
00019 extern void trapret(void);
00020
00021 static void wakeup1(void *chan);
00022
00023 void
00024 pinit(void)
00025 {
00026   initlock(&ptable.lock, "ptable");
00027 }
00028
00029 // Must be called with interrupts disabled
00030 int
00031 cpuid() {
00032   return mycpu()-cpus;
00033 }
00034
00035 // Must be called with interrupts disabled to avoid the caller being
00036 // rescheduled between reading lapicid and running through the loop.
```

```
00037 struct cpu*
00038 mycpu(void)
00039 {
00040   int apicid, i;
00041
00042   if(readeflags()&FL_IF)
00043     panic("mycpu called with interrupts enabled\n");
00044
00045   apicid = lapicid();
00046   // APIC IDs are not guaranteed to be contiguous. Maybe we should have
00047   // a reverse map, or reserve a register to store &cpus[i].
00048   for (i = 0; i < ncpu; ++i) {
00049     if (cpus[i].apicid == apicid)
00050       return &cpus[i];
00051   }
00052   panic("unknown apicid\n");
00053 }
00054
00055 // Disable interrupts so that we are not rescheduled
00056 // while reading proc from the cpu structure
00057 struct proc*
00058 myproc(void) {
00059   struct cpu *c;
00060   struct proc *p;
00061   pushcli();
00062   c = mycpu();
00063   p = c->proc;
00064   popcli();
00065   return p;
00066 }
00067
00068 //PAGEBREAK: 32
00069 // Look in the process table for an UNUSED proc.
00070 // If found, change state to EMBRYO and initialize
00071 // state required to run in the kernel.
00072 // Otherwise return 0.
00073 static struct proc*
00074 allocproc(void)
00075 {
00076   struct proc *p;
00077   char *sp;
00078
00079   acquire(&ptable.lock);
00080
00081   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
00082     if(p->state == UNUSED)
00083       goto found;
00084
00085   release(&ptable.lock);
00086   return 0;
00087
00088 found:
00089   p->state = EMBRYO;
00090   p->pid = nextpid++;
00091
00092   release(&ptable.lock);
00093
00094   // Allocate kernel stack.
00095   if((p->kstack = kalloc()) == 0){
00096     p->state = UNUSED;
00097     return 0;
00098   }
00099   sp = p->kstack + KSTACKSIZE;
00100
00101   // Leave room for trap frame.
00102   sp -= sizeof *p->tf;
00103   p->tf = (struct trapframe*)sp;
00104
00105   // Set up new context to start executing at forkret,
00106   // which returns to trapret.
00107   sp -= 4;
00108   *(uint*)sp = (uint)trapret;
00109
00110   sp -= sizeof *p->context;
00111   p->context = (struct context*)sp;
00112   memset(p->context, 0, sizeof *p->context);
00113   p->context->eip = (uint)forkret;
00114
00115   return p;
00116 }
00117
00118 //PAGEBREAK: 32
00119 // Set up first user process.
00120 void
00121 userinit(void)
00122 {
00123   struct proc *p;
```

```
00124   extern char _binary_initcode_start[], _binary_initcode_size[];
00125
00126   p = allocproc();
00127
00128   initproc = p;
00129   if((p->pgdir = setupkvm()) == 0)
00130     panic("userinit: out of memory?");
00131   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
00132   p->sz = PGSIZE;
00133   memset(p->tf, 0, sizeof(*p->tf));
00134   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
00135   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
00136   p->tf->es = p->tf->ds;
00137   p->tf->ss = p->tf->ds;
00138   p->tf->eflags = FL_IF;
00139   p->tf->esp = PGSIZE;
00140   p->tf->eip = 0;  // beginning of initcode.S
00141
00142   safestrcpy(p->name, "initcode", sizeof(p->name));
00143   p->cwd = namei("/");
00144
00145   // this assignment to p->state lets other cores
00146   // run this process. the acquire forces the above
00147   // writes to be visible, and the lock is also needed
00148   // because the assignment might not be atomic.
00149   acquire(&ptable.lock);
00150
00151   p->state = RUNNABLE;
00152
00153   release(&ptable.lock);
00154 }
00155
00156 // Grow current process's memory by n bytes.
00157 // Return 0 on success, -1 on failure.
00158 int
00159 growproc(int n)
00160 {
00161   uint sz;
00162   struct proc *curproc = myproc();
00163
00164   sz = curproc->sz;
00165   if(n > 0){
00166     if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
00167       return -1;
00168   } else if(n < 0){
00169     if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
00170       return -1;
00171   }
00172   curproc->sz = sz;
00173   switchuvm(curproc);
00174   return 0;
00175 }
00176
00177 // Create a new process copying p as the parent.
00178 // Sets up stack to return as if from system call.
00179 // Caller must set state of returned proc to RUNNABLE.
00180 int
00181 fork(void)
00182 {
00183   int i, pid;
00184   struct proc *np;
00185   struct proc *curproc = myproc();
00186
00187   // Allocate process.
00188   if((np = allocproc()) == 0){
00189     return -1;
00190   }
00191
00192   // Copy process state from proc.
00193   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
00194     kfree(np->kstack);
00195     np->kstack = 0;
00196     np->state = UNUSED;
00197     return -1;
00198   }
00199   np->sz = curproc->sz;
00200   np->parent = curproc;
00201   *np->tf = *curproc->tf;
00202
00203   // Clear %eax so that fork returns 0 in the child.
00204   np->tf->eax = 0;
00205
00206   for(i = 0; i < NOFILE; i++)
00207     if(curproc->ofile[i])
00208       np->ofile[i] = filedup(curproc->ofile[i]);
00209   np->cwd = idup(curproc->cwd);
00210
```

```
00211   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
00212
00213   pid = np->pid;
00214
00215   acquire(&ptable.lock);
00216
00217   np->state = RUNNABLE;
00218
00219   release(&ptable.lock);
00220
00221   return pid;
00222 }
00223
00224 // Exit the current process.  Does not return.
00225 // An exited process remains in the zombie state
00226 // until its parent calls wait() to find out it exited.
00227 void
00228 exit(void)
00229 {
00230   struct proc *curproc = myproc();
00231   struct proc *p;
00232   int fd;
00233
00234   if(curproc == initproc)
00235     panic("init exiting");
00236
00237   // Close all open files.
00238   for(fd = 0; fd < NOFILE; fd++){
00239     if(curproc->ofile[fd]){
00240       fileclose(curproc->ofile[fd]);
00241       curproc->ofile[fd] = 0;
00242     }
00243   }
00244
00245   begin_op();
00246   iput(curproc->cwd);
00247   end_op();
00248   curproc->cwd = 0;
00249
00250   acquire(&ptable.lock);
00251
00252   // Parent might be sleeping in wait().
00253   wakeup1(curproc->parent);
00254
00255   // Pass abandoned children to init.
00256   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00257     if(p->parent == curproc){
00258       p->parent = initproc;
00259       if(p->state == ZOMBIE)
00260         wakeup1(initproc);
00261     }
00262   }
00263
00264   // Jump into the scheduler, never to return.
00265   curproc->state = ZOMBIE;
00266   sched();
00267   panic("zombie exit");
00268 }
00269
00270 // Wait for a child process to exit and return its pid.
00271 // Return -1 if this process has no children.
00272 int
00273 wait(void)
00274 {
00275   struct proc *p;
00276   int havekids, pid;
00277   struct proc *curproc = myproc();
00278
00279   acquire(&ptable.lock);
00280   for(;;){
00281     // Scan through table looking for exited children.
00282     havekids = 0;
00283     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00284       if(p->parent != curproc)
00285         continue;
00286       havekids = 1;
00287       if(p->state == ZOMBIE){
00288         // Found one.
00289         pid = p->pid;
00290         kfree(p->kstack);
00291         p->kstack = 0;
00292         freevm(p->pgdir);
00293         p->pid = 0;
00294         p->parent = 0;
00295         p->name[0] = 0;
00296         p->killed = 0;
00297         p->state = UNUSED;
```

```
00298         release(&ptable.lock);
00299         return pid;
00300       }
00301     }
00302
00303     // No point waiting if we don't have any children.
00304     if(!havekids || curproc->killed){
00305       release(&ptable.lock);
00306       return -1;
00307     }
00308
00309     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
00310     sleep(curproc, &ptable.lock);  //DOC: wait-sleep
00311   }
00312 }
00313
00314 //PAGEBREAK: 42
00315 // Per-CPU process scheduler.
00316 // Each CPU calls scheduler() after setting itself up.
00317 // Scheduler never returns.  It loops, doing:
00318 //  - choose a process to run
00319 //  - swtch to start running that process
00320 //  - eventually that process transfers control
00321 //      via swtch back to the scheduler.
00322 void
00323 scheduler(void)
00324 {
00325   struct proc *p;
00326   struct cpu *c = mycpu();
00327   c->proc = 0;
00328
00329   for(;;){
00330     // Enable interrupts on this processor.
00331     sti();
00332
00333     // Loop over process table looking for process to run.
00334     acquire(&ptable.lock);
00335     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00336       if(p->state != RUNNABLE)
00337         continue;
00338
00339       // Switch to chosen process.  It is the process's job
00340       // to release ptable.lock and then reacquire it
00341       // before jumping back to us.
00342       c->proc = p;
00343       switchuvm(p);
00344       p->state = RUNNING;
00345
00346       swtch(&(c->scheduler), p->context);
00347       switchkvm();
00348
00349       // Process is done running for now.
00350       // It should have changed its p->state before coming back.
00351       c->proc = 0;
00352     }
00353     release(&ptable.lock);
00354
00355   }
00356 }
00357
00358 // Enter scheduler.  Must hold only ptable.lock
00359 // and have changed proc->state. Saves and restores
00360 // intena because intena is a property of this
00361 // kernel thread, not this CPU. It should
00362 // be proc->intena and proc->ncli, but that would
00363 // break in the few places where a lock is held but
00364 // there's no process.
00365 void
00366 sched(void)
00367 {
00368   int intena;
00369   struct proc *p = myproc();
00370
00371   if(!holding(&ptable.lock))
00372     panic("sched ptable.lock");
00373   if(mycpu()->ncli != 1)
00374     panic("sched locks");
00375   if(p->state == RUNNING)
00376     panic("sched running");
00377   if(readeflags()&FL_IF)
00378     panic("sched interruptible");
00379   intena = mycpu()->intena;
00380   swtch(&p->context, mycpu()->scheduler);
00381   mycpu()->intena = intena;
00382 }
00383
00384 // Give up the CPU for one scheduling round.
```

```
00385 void
00386 yield(void)
00387 {
00388   acquire(&ptable.lock);  //DOC: yieldlock
00389   myproc()->state = RUNNABLE;
00390   sched();
00391   release(&ptable.lock);
00392 }
00393
00394 // A fork child's very first scheduling by scheduler()
00395 // will swtch here.  "Return" to user space.
00396 void
00397 forkret(void)
00398 {
00399   static int first = 1;
00400   // Still holding ptable.lock from scheduler.
00401   release(&ptable.lock);
00402
00403   if (first) {
00404     // Some initialization functions must be run in the context
00405     // of a regular process (e.g., they call sleep), and thus cannot
00406     // be run from main().
00407     first = 0;
00408     iinit(ROOTDEV);
00409     initlog(ROOTDEV);
00410   }
00411
00412   // Return to "caller", actually trapret (see allocproc).
00413 }
00414
00415 // Atomically release lock and sleep on chan.
00416 // Reacquires lock when awakened.
00417 void
00418 sleep(void *chan, struct spinlock *lk)
00419 {
00420   struct proc *p = myproc();
00421
00422   if(p == 0)
00423     panic("sleep");
00424
00425   if(lk == 0)
00426     panic("sleep without lk");
00427
00428   // Must acquire ptable.lock in order to
00429   // change p->state and then call sched.
00430   // Once we hold ptable.lock, we can be
00431   // guaranteed that we won't miss any wakeup
00432   // (wakeup runs with ptable.lock locked),
00433   // so it's okay to release lk.
00434   if(lk != &ptable.lock){  //DOC: sleeplock0
00435     acquire(&ptable.lock);  //DOC: sleeplock1
00436     release(lk);
00437   }
00438   // Go to sleep.
00439   p->chan = chan;
00440   p->state = SLEEPING;
00441
00442   sched();
00443
00444   // Tidy up.
00445   p->chan = 0;
00446
00447   // Reacquire original lock.
00448   if(lk != &ptable.lock){  //DOC: sleeplock2
00449     release(&ptable.lock);
00450     acquire(lk);
00451   }
00452 }
00453
00454 //PAGEBREAK!
00455 // Wake up all processes sleeping on chan.
00456 // The ptable lock must be held.
00457 static void
00458 wakeup1(void *chan)
00459 {
00460   struct proc *p;
00461
00462   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
00463     if(p->state == SLEEPING && p->chan == chan)
00464       p->state = RUNNABLE;
00465 }
00466
00467 // Wake up all processes sleeping on chan.
00468 void
00469 wakeup(void *chan)
00470 {
00471   acquire(&ptable.lock);
```

```
00472   wakeup1(chan);
00473   release(&ptable.lock);
00474 }
00475
00476 // Kill the process with the given pid.
00477 // Process won't exit until it returns
00478 // to user space (see trap in trap.c).
00479 int
00480 kill(int pid)
00481 {
00482   struct proc *p;
00483
00484   acquire(&ptable.lock);
00485   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00486     if(p->pid == pid){
00487       p->killed = 1;
00488       // Wake process from sleep if necessary.
00489       if(p->state == SLEEPING)
00490         p->state = RUNNABLE;
00491       release(&ptable.lock);
00492       return 0;
00493     }
00494   }
00495   release(&ptable.lock);
00496   return -1;
00497 }
00498
00499 //PAGEBREAK: 36
00500 // Print a process listing to console.  For debugging.
00501 // Runs when user types ^P on console.
00502 // No lock to avoid wedging a stuck machine further.
00503 void
00504 procdump(void)
00505 {
00506   static char *states[] = {
00507   [UNUSED]    "unused",
00508   [EMBRYO]    "embryo",
00509   [SLEEPING]  "sleep ",
00510   [RUNNABLE]  "runble",
00511   [RUNNING]   "run   ",
00512   [ZOMBIE]    "zombie"
00513   };
00514   int i;
00515   struct proc *p;
00516   char *state;
00517   uint pc[10];
00518
00519   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00520     if(p->state == UNUSED)
00521       continue;
00522     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
00523       state = states[p->state];
00524     else
00525       state = "???";
00526     cprintf("%d %s %s", p->pid, state, p->name);
00527     if(p->state == SLEEPING){
00528       getcallerpcs((uint*)p->context->ebp+2, pc);
00529       for(i=0; i<10 && pc[i] != 0; i++)
00530         cprintf(" %p", pc[i]);
00531     }
00532     cprintf("\n");
00533   }
00534 }
00535
00536 int
00537 cps()
00538 {
00539 struct proc *p;
00540 //Enables interrupts on this processor.
00541 sti();
00542
00543 //Loop over process table looking for process with pid.
00544 acquire(&ptable.lock);
00545 cprintf("name \t pid \t state \t priority \n");
00546 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00547   if(p->state == SLEEPING)
00548     cprintf("%s \t %d \t SLEEPING \t %d \n ", p->name,p->pid,p->priority);
00549   else if(p->state == RUNNING)
00550     cprintf("%s \t %d \t RUNNING \t %d \n ", p->name,p->pid,p->priority);
00551   else if(p->state == RUNNABLE)
00552     cprintf("%s \t %d \t RUNNABLE \t %d \n ", p->name,p->pid,p->priority);
00553 }
00554 release(&ptable.lock);
00555 return 23;
00556 }
00557
00558 int
```

```
00559 chpr(int pid, int priority)
00560 {
00561   struct proc *p;
00562   acquire(&ptable.lock);
00563   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00564     if(p->pid == pid){
00565       p->priority = priority;
00566       break;
00567     }
00568   }
00569   release(&ptable.lock);
00570   return pid;
00571 }
```

5.155 proc.d File Reference

5.156 proc.d

Go to the documentation of this file.

```
00001 proc.o: proc.c /usr/include/stdc-predef.h types.h defs.h param.h \
00002  memlayout.h mmu.h x86.h proc.h spinlock.h
```

5.157 proc.h File Reference

Classes

- struct context
- struct cpu
- struct proc

Enumerations

- enum procstate {
  UNUSED , EMBRYO , SLEEPING , RUNNABLE ,
  RUNNING , ZOMBIE }

Variables

- struct cpu cpus [NCPU]
- int ncpu

5.157.1 Enumeration Type Documentation

5.157.1.1 procstate

```
enum procstate
```

Enumerator

UNUSED	
EMBRYO	
SLEEPING	
RUNNABLE	
RUNNING	
ZOMBIE	

Definition at line 35 of file proc.h.

```
00035 { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

5.157.2 Variable Documentation

5.157.2.1 cpus

```
struct cpu cpus[NCPU]  [extern]
```

Definition at line 14 of file mp.c.

Referenced by cpuid(), mpinit(), mycpu(), seginit(), and startothers().

5.157.2.2 ncpu

```
int ncpu  [extern]
```

Definition at line 15 of file mp.c.

Referenced by ideinit(), mpinit(), mycpu(), and startothers().

5.158 proc.h

Go to the documentation of this file.

```
00001 // Per-CPU state
00002 struct cpu {
00003   uchar apicid;                // Local APIC ID
00004   struct context *scheduler;   // swtch() here to enter scheduler
00005   struct taskstate ts;         // Used by x86 to find stack for interrupt
00006   struct segdesc gdt[NSEGS];   // x86 global descriptor table
00007   volatile uint started;       // Has the CPU started?
00008   int ncli;                    // Depth of pushcli nesting.
00009   int intena;                  // Were interrupts enabled before pushcli?
00010   struct proc *proc;           // The process running on this cpu or null
00011 };
00012
00013 extern struct cpu cpus[NCPU];
00014 extern int ncpu;
00015
00016 //PAGEBREAK: 17
00017 // Saved registers for kernel context switches.
00018 // Don't need to save all the segment registers (%cs, etc),
```

```
00019 // because they are constant across kernel contexts.
00020 // Don't need to save %eax, %ecx, %edx, because the
00021 // x86 convention is that the caller has saved them.
00022 // Contexts are stored at the bottom of the stack they
00023 // describe; the stack pointer is the address of the context.
00024 // The layout of the context matches the layout of the stack in swtch.S
00025 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
00026 // but it is on the stack and allocproc() manipulates it.
00027 struct context {
00028   uint edi;
00029   uint esi;
00030   uint ebx;
00031   uint ebp;
00032   uint eip;
00033 };
00034
00035 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
00036
00037 // Per-process state
00038 struct proc {
00039   uint sz;                     // Size of process memory (bytes)
00040   pde_t* pgdir;                // Page table
00041   char *kstack;                // Bottom of kernel stack for this process
00042   enum procstate state;        // Process state
00043   int pid;                     // Process ID
00044   struct proc *parent;         // Parent process
00045   struct trapframe *tf;        // Trap frame for current syscall
00046   struct context *context;     // swtch() here to run process
00047   void *chan;                  // If non-zero, sleeping on chan
00048   int killed;                  // If non-zero, have been killed
00049   struct file *ofile[NOFILE];  // Open files
00050   struct inode *cwd;           // Current directory
00051   char name[16];               // Process name (debugging)
00052   int priority;                // Process priority
00053 };
00054
00055 // Process memory is laid out contiguously, low addresses first:
00056 //   text
00057 //   original data and bss
00058 //   fixed-size stack
00059 //   expandable heap
```

5.159 ps.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
```

Functions

- int main (void)

5.159.1 Function Documentation

5.159.1.1 main()

```
int main (
            void  )
```

Definition at line 6 of file ps.c.

```
00006             {
00007   cps();
00008   exit();
00009 }
```

5.160 ps.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004 #include "fcntl.h"
00005
00006 int main(void){
00007   cps();
00008   exit();
00009 }
```

5.161 ps.d File Reference

5.162 ps.d

Go to the documentation of this file.

```
00001 ps.o: ps.c /usr/include/stdc-predef.h types.h stat.h user.h fcntl.h
```

5.163 pstree.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "uproc.h"
```

Macros

- #define MAXPROC 64

Functions

- int main (int argc, char *argv[ ])

5.163.1 Macro Definition Documentation

5.163.1.1 MAXPROC

```
#define MAXPROC 64
```

Definition at line 5 of file pstree.c.

5.163.2 Function Documentation

5.163.2.1 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 8 of file pstree.c.

```
00008                           {
00009   // declare a uproc struct with an allocated size of 64
00010   struct uproc *prs = malloc(MAXPROC*sizeof(struct uproc));
00011
00012   // call getprocs in kernel space and get the amount of processes running back
00013   int num = getprocs(MAXPROC, prs);
00014
00015   // getprocs will return -1 if an error is received. Handle errors here
00016   if(num == -1){
00017     printf(1, "Kernel returned an error when getting processes. \n");
00018     exit();
00019   }
00020
00021   // create a counter
00022   int i=0;
00023
00024   // loop through uproc structs that were created in the kernel space for each process
00025   for(; i< num; i++) {
00026     // format output to the user
00027     if(prs[i].pid == 1){
00028       // if the pid is 1 then this is init
00029       printf(1, "%s[%d] \n", prs[i].name, prs[i].pid-1);
00030     }else if(prs[i].ppid == 1){
00031       // if parent pid is 1, then it is a child to parent init
00032       printf(1, "   %s[%d] \n", prs[i].name, prs[i].pid-1);
00033     }else {
00034       // if parent pid is not 1, then it is a grandchild to init and child to another process
00035       printf(1, "      %s[%d] \n", prs[i].name, prs[i].pid-1);
00036     }
00037   }
00038
00039   // exit after pstree command is done
00040   exit();
00041 }
```

5.164 pstree.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004 #include "uproc.h"
00005 #define MAXPROC 64
00006
00007 int
00008 main(int argc, char *argv[]){
00009   // declare a uproc struct with an allocated size of 64
00010   struct uproc *prs = malloc(MAXPROC*sizeof(struct uproc));
00011
00012   // call getprocs in kernel space and get the amount of processes running back
00013   int num = getprocs(MAXPROC, prs);
00014
00015   // getprocs will return -1 if an error is received. Handle errors here
00016   if(num == -1){
00017     printf(1, "Kernel returned an error when getting processes. \n");
00018     exit();
00019   }
00020
00021   // create a counter
00022   int i=0;
00023
00024   // loop through uproc structs that were created in the kernel space for each process
```

```
00025   for(; i< num; i++) {
00026     // format output to the user
00027     if(prs[i].pid == 1){
00028       // if the pid is 1 then this is init
00029       printf(1, "%s[%d] \n", prs[i].name, prs[i].pid-1);
00030     }else if(prs[i].ppid == 1){
00031       // if parent pid is 1, then it is a child to parent init
00032       printf(1, "   %s[%d] \n", prs[i].name, prs[i].pid-1);
00033     }else {
00034       // if parent pid is not 1, then it is a grandchild to init and child to another process
00035       printf(1, "      %s[%d] \n", prs[i].name, prs[i].pid-1);
00036     }
00037   }
00038
00039   // exit after pstree command is done
00040   exit();
00041 }
```

5.165 pstree.d File Reference

5.166 pstree.d

Go to the documentation of this file.

```
00001 pstree.o: pstree.c /usr/include/stdc-predef.h types.h stat.h user.h \
00002  uproc.h
```

5.167 README.md File Reference

5.168 rm.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- int main (int argc, char *argv[ ])

5.168.1 Function Documentation

5.168.1.1 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 6 of file rm.c.

```
00007 {
00008   int i;
00009
00010   if(argc < 2){
00011     printf(2, "Usage: rm files...\n");
00012     exit();
00013   }
00014
00015   for(i = 1; i < argc; i++){
00016     if(unlink(argv[i]) < 0){
00017       printf(2, "rm: %s failed to delete\n", argv[i]);
00018       break;
00019     }
00020   }
00021
00022   exit();
00023 }
```

5.169 rm.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004
00005 int
00006 main(int argc, char *argv[])
00007 {
00008   int i;
00009
00010   if(argc < 2){
00011     printf(2, "Usage: rm files...\n");
00012     exit();
00013   }
00014
00015   for(i = 1; i < argc; i++){
00016     if(unlink(argv[i]) < 0){
00017       printf(2, "rm: %s failed to delete\n", argv[i]);
00018       break;
00019     }
00020   }
00021
00022   exit();
00023 }
```

5.170 rm.d File Reference

5.171 rm.d

Go to the documentation of this file.

```
00001 rm.o: rm.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.172 sh.c File Reference

```
#include "types.h"
#include "user.h"
#include "fcntl.h"
```

Classes

- struct backcmd
- struct cmd
- struct execcmd
- struct listcmd
- struct pipecmd
- struct redircmd

Macros

- #define BACK 5
- #define EXEC 1
- #define LIST 4
- #define MAXARGS 10
- #define PIPE 3
- #define REDIR 2

Functions

- struct cmd * backcmd (struct cmd *subcmd)
- struct cmd * execcmd (void)
- int fork1 (void)
- int getcmd (char *buf, int nbuf)
- int gettoken (char **ps, char *es, char **q, char **eq)
- struct cmd * listcmd (struct cmd *left, struct cmd *right)
- int main (void)
- struct cmd * nulterminate (struct cmd *)
- void panic (char *)
- struct cmd * parseblock (char **ps, char *es)
- struct cmd * parsecmd (char *)
- struct cmd * parseexec (char **, char *)
- struct cmd * parseline (char **, char *)
- struct cmd * parsepipe (char **, char *)
- struct cmd * parseredirs (struct cmd *cmd, char **ps, char *es)
- int peek (char **ps, char *es, char *toks)
- struct cmd * pipecmd (struct cmd *left, struct cmd *right)
- struct cmd * redircmd (struct cmd *subcmd, char *file, char *efile, int mode, int fd)
- void runcmd (struct cmd *cmd)

Variables

- char symbols [ ] = "<|>&;()"
- char whitespace [ ] = " \t\r\n\v"

5.172.1 Macro Definition Documentation

5.172.1.1 BACK

```
#define BACK 5
```

Definition at line 12 of file sh.c.

5.172.1.2 EXEC

```
#define EXEC 1
```

Definition at line 8 of file sh.c.

5.172.1.3 LIST

```
#define LIST 4
```

Definition at line 11 of file sh.c.

5.172.1.4 MAXARGS

```
#define MAXARGS 10
```

Definition at line 14 of file sh.c.

5.172.1.5 PIPE

```
#define PIPE 3
```

Definition at line 10 of file sh.c.

5.172.1.6 REDIR

```
#define REDIR 2
```

Definition at line 9 of file sh.c.

5.172.2 Function Documentation

5.172.2.1 backcmd()

```
struct cmd * backcmd (
            struct cmd * subcmd )
```

Definition at line 249 of file sh.c.

```
00250 {
00251   struct backcmd *cmd;
00252
00253   cmd = malloc(sizeof(*cmd));
00254   memset(cmd, 0, sizeof(*cmd));
00255   cmd->type = BACK;
00256   cmd->cmd = subcmd;
00257   return (struct cmd*)cmd;
00258 }
```

5.172.2.2 execcmd()

```
struct cmd * execcmd (
            void  )
```

Definition at line 196 of file sh.c.

```
00197 {
00198   struct execcmd *cmd;
00199
00200   cmd = malloc(sizeof(*cmd));
00201   memset(cmd, 0, sizeof(*cmd));
00202   cmd->type = EXEC;
00203   return (struct cmd*)cmd;
00204 }
```

5.172.2.3 fork1()

```
int fork1 (
            void  )
```

Definition at line 182 of file sh.c.

```
00183 {
00184   int pid;
00185
00186   pid = fork();
00187   if(pid == -1)
00188     panic("fork");
00189   return pid;
00190 }
```

Referenced by main(), and runcmd().

5.172.2.4 getcmd()

```
int getcmd (
              char * buf,
              int nbuf )
```

Definition at line 134 of file sh.c.

```
00135 {
00136   printf(2, "$ ");
00137   memset(buf, 0, nbuf);
00138   gets(buf, nbuf);
00139   if(buf[0] == 0) // EOF
00140     return -1;
00141   return 0;
00142 }
```

Referenced by main().

5.172.2.5 gettoken()

```
int gettoken (
              char ** ps,
              char * es,
              char ** q,
              char ** eq )
```

Definition at line 266 of file sh.c.

```
00267 {
00268   char *s;
00269   int ret;
00270
00271   s = *ps;
00272   while(s < es && strchr(whitespace, *s))
00273     s++;
00274   if(q)
00275     *q = s;
00276   ret = *s;
00277   switch(*s){
00278   case 0:
00279     break;
00280   case '|':
00281   case '(':
00282   case ')':
00283   case ';':
00284   case '&':
00285   case '<':
00286     s++;
00287     break;
00288   case '>':
00289     s++;
00290     if(*s == '>'){
00291       ret = '+';
00292       s++;
00293     }
00294     break;
00295   default:
00296     ret = 'a';
00297     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
00298       s++;
00299     break;
00300   }
00301   if(eq)
00302     *eq = s;
00303
00304   while(s < es && strchr(whitespace, *s))
00305     s++;
00306   *ps = s;
00307   return ret;
00308 }
```

Referenced by parseblock(), parseexec(), parseline(), parsepipe(), and parseredirs().

5.172.2.6 listcmd()

```
struct cmd * listcmd (
              struct cmd * left,
              struct cmd * right )
```

Definition at line 236 of file sh.c.

```
00237 {
00238   struct listcmd *cmd;
00239
00240   cmd = malloc(sizeof(*cmd));
00241   memset(cmd, 0, sizeof(*cmd));
00242   cmd->type = LIST;
00243   cmd->left = left;
00244   cmd->right = right;
00245   return (struct cmd*)cmd;
00246 }
```

5.172.2.7 main()

```
int main (
              void  )
```

Definition at line 145 of file sh.c.

```
00146 {
00147   static char buf[100];
00148   int fd;
00149
00150   // Ensure that three file descriptors are open.
00151   while((fd = open("console", O_RDWR)) >= 0){
00152     if(fd >= 3){
00153       close(fd);
00154       break;
00155     }
00156   }
00157
00158   // Read and run input commands.
00159   while(getcmd(buf, sizeof(buf)) >= 0){
00160     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
00161       // Chdir must be called by the parent, not the child.
00162       buf[strlen(buf)-1] = 0;  // chop \n
00163       if(chdir(buf+3) < 0)
00164         printf(2, "cannot cd %s\n", buf+3);
00165       continue;
00166     }
00167     if(fork1() == 0)
00168       runcmd(parsecmd(buf));
00169     wait();
00170   }
00171   exit();
00172 }
```

5.172.2.8 nulterminate()

```
struct cmd * nulterminate (
              struct cmd * cmd )
```

Definition at line 450 of file sh.c.

```
00451 {
00452   int i;
00453   struct backcmd *bcmd;
00454   struct execcmd *ecmd;
00455   struct listcmd *lcmd;
00456   struct pipecmd *pcmd;
00457   struct redircmd *rcmd;
```

```
00458
00459   if(cmd == 0)
00460     return 0;
00461
00462   switch(cmd->type){
00463   case EXEC:
00464     ecmd = (struct execcmd*)cmd;
00465     for(i=0; ecmd->argv[i]; i++)
00466       *ecmd->eargv[i] = 0;
00467     break;
00468
00469   case REDIR:
00470     rcmd = (struct redircmd*)cmd;
00471     nulterminate(rcmd->cmd);
00472     *rcmd->efile = 0;
00473     break;
00474
00475   case PIPE:
00476     pcmd = (struct pipecmd*)cmd;
00477     nulterminate(pcmd->left);
00478     nulterminate(pcmd->right);
00479     break;
00480
00481   case LIST:
00482     lcmd = (struct listcmd*)cmd;
00483     nulterminate(lcmd->left);
00484     nulterminate(lcmd->right);
00485     break;
00486
00487   case BACK:
00488     bcmd = (struct backcmd*)cmd;
00489     nulterminate(bcmd->cmd);
00490     break;
00491   }
00492   return cmd;
00493 }
```

Referenced by nulterminate(), and parsecmd().

5.172.2.9 panic()

```
void panic (
              char * s )
```

Definition at line 175 of file sh.c.

```
00176 {
00177   printf(2, "%s\n", s);
00178   exit();
00179 }
```

Referenced by fork1(), parseblock(), parsecmd(), parseexec(), parseredirs(), and runcmd().

5.172.2.10 parseblock()

```
struct cmd * parseblock (
              char ** ps,
              char * es )
```

Definition at line 400 of file sh.c.

```
00401 {
00402   struct cmd *cmd;
00403
00404   if(!peek(ps, es, "("))
00405     panic("parseblock");
00406   gettoken(ps, es, 0, 0);
00407   cmd = parseline(ps, es);
00408   if(!peek(ps, es, ")"))
00409     panic("syntax - missing )");
00410   gettoken(ps, es, 0, 0);
00411   cmd = parseredirs(cmd, ps, es);
00412   return cmd;
00413 }
```

Referenced by parseexec().

5.172.2.11 parsecmd()

```
struct cmd * parsecmd (
              char * s )
```

Definition at line 328 of file sh.c.

```
00329 {
00330   char *es;
00331   struct cmd *cmd;
00332
00333   es = s + strlen(s);
00334   cmd = parseline(&s, es);
00335   peek(&s, es, "");
00336   if(s != es){
00337     printf(2, "leftovers: %s\n", s);
00338     panic("syntax");
00339   }
00340   nulterminate(cmd);
00341   return cmd;
00342 }
```

Referenced by main().

5.172.2.12 parseexec()

```
struct cmd * parseexec (
              char ** ps,
              char * es )
```

Definition at line 416 of file sh.c.

```
00417 {
00418   char *q, *eq;
00419   int tok, argc;
00420   struct execcmd *cmd;
00421   struct cmd *ret;
00422
00423   if(peek(ps, es, "("))
00424     return parseblock(ps, es);
00425
00426   ret = execcmd();
00427   cmd = (struct execcmd*)ret;
00428
00429   argc = 0;
00430   ret = parseredirs(ret, ps, es);
00431   while(!peek(ps, es, "|)&;")){
00432     if((tok=gettoken(ps, es, &q, &eq)) == 0)
00433       break;
00434     if(tok != 'a')
00435       panic("syntax");
00436     cmd->argv[argc] = q;
00437     cmd->eargv[argc] = eq;
00438     argc++;
00439     if(argc >= MAXARGS)
00440       panic("too many args");
00441     ret = parseredirs(ret, ps, es);
00442   }
00443   cmd->argv[argc] = 0;
00444   cmd->eargv[argc] = 0;
00445   return ret;
00446 }
```

Referenced by parsepipe().

5.172.2.13 parseline()

```
struct cmd * parseline (
              char ** ps,
              char * es )
```

Definition at line 345 of file sh.c.

```
00346 {
00347   struct cmd *cmd;
00348
00349   cmd = parsepipe(ps, es);
00350   while(peek(ps, es, "&")){
00351     gettoken(ps, es, 0, 0);
00352     cmd = backcmd(cmd);
00353   }
00354   if(peek(ps, es, ";")){
00355     gettoken(ps, es, 0, 0);
00356     cmd = listcmd(cmd, parseline(ps, es));
00357   }
00358   return cmd;
00359 }
```

Referenced by parseblock(), parsecmd(), and parseline().

5.172.2.14 parsepipe()

```
struct cmd * parsepipe (
              char ** ps,
              char * es )
```

Definition at line 362 of file sh.c.

```
00363 {
00364   struct cmd *cmd;
00365
00366   cmd = parseexec(ps, es);
00367   if(peek(ps, es, "|")){
00368     gettoken(ps, es, 0, 0);
00369     cmd = pipecmd(cmd, parsepipe(ps, es));
00370   }
00371   return cmd;
00372 }
```

Referenced by parseline(), and parsepipe().

5.172.2.15 parseredirs()

```
struct cmd * parseredirs (
              struct cmd * cmd,
              char ** ps,
              char * es )
```

Definition at line 375 of file sh.c.

```
00376 {
00377   int tok;
00378   char *q, *eq;
00379
00380   while(peek(ps, es, "<>")){
00381     tok = gettoken(ps, es, 0, 0);
00382     if(gettoken(ps, es, &q, &eq) != 'a')
00383       panic("missing file for redirection");
00384     switch(tok){
00385     case '<':
```

```
00386       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
00387       break;
00388     case '>':
00389       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
00390       break;
00391     case '+':  // »
00392       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
00393       break;
00394     }
00395   }
00396   return cmd;
00397 }
```

Referenced by parseblock(), and parseexec().

5.172.2.16 peek()

```
int peek (
              char ** ps,
              char * es,
              char * toks )
```

Definition at line 311 of file sh.c.

```
00312 {
00313   char *s;
00314
00315   s = *ps;
00316   while(s < es && strchr(whitespace, *s))
00317     s++;
00318   *ps = s;
00319   return *s && strchr(toks, *s);
00320 }
```

Referenced by parseblock(), parsecmd(), parseexec(), parseline(), parsepipe(), and parseredirs().

5.172.2.17 pipecmd()

```
struct cmd * pipecmd (
              struct cmd * left,
              struct cmd * right )
```

Definition at line 223 of file sh.c.

```
00224 {
00225   struct pipecmd *cmd;
00226
00227   cmd = malloc(sizeof(*cmd));
00228   memset(cmd, 0, sizeof(*cmd));
00229   cmd->type = PIPE;
00230   cmd->left = left;
00231   cmd->right = right;
00232   return (struct cmd*)cmd;
00233 }
```

5.172.2.18 redircmd()

```
struct cmd * redircmd (
              struct cmd * subcmd,
              char * file,
              char * efile,
              int mode,
              int fd )
```

Definition at line 207 of file sh.c.

```
00208 {
00209   struct redircmd *cmd;
00210
00211   cmd = malloc(sizeof(*cmd));
00212   memset(cmd, 0, sizeof(*cmd));
00213   cmd->type = REDIR;
00214   cmd->cmd = subcmd;
00215   cmd->file = file;
00216   cmd->efile = efile;
00217   cmd->mode = mode;
00218   cmd->fd = fd;
00219   return (struct cmd*)cmd;
00220 }
```

5.172.2.19 runcmd()

```
void runcmd (
              struct cmd * cmd )
```

Definition at line 58 of file sh.c.

```
00059 {
00060   int p[2];
00061   struct backcmd *bcmd;
00062   struct execcmd *ecmd;
00063   struct listcmd *lcmd;
00064   struct pipecmd *pcmd;
00065   struct redircmd *rcmd;
00066
00067   if(cmd == 0)
00068     exit();
00069
00070   switch(cmd->type){
00071   default:
00072     panic("runcmd");
00073
00074   case EXEC:
00075     ecmd = (struct execcmd*)cmd;
00076     if(ecmd->argv[0] == 0)
00077       exit();
00078     exec(ecmd->argv[0], ecmd->argv);
00079     printf(2, "exec %s failed\n", ecmd->argv[0]);
00080     break;
00081
00082   case REDIR:
00083     rcmd = (struct redircmd*)cmd;
00084     close(rcmd->fd);
00085     if(open(rcmd->file, rcmd->mode) < 0){
00086       printf(2, "open %s failed\n", rcmd->file);
00087       exit();
00088     }
00089     runcmd(rcmd->cmd);
00090     break;
00091
00092   case LIST:
00093     lcmd = (struct listcmd*)cmd;
00094     if(fork1() == 0)
00095       runcmd(lcmd->left);
00096     wait();
00097     runcmd(lcmd->right);
00098     break;
00099
00100   case PIPE:
00101     pcmd = (struct pipecmd*)cmd;
```

```
00102     if(pipe(p) < 0)
00103       panic("pipe");
00104     if(fork1() == 0){
00105       close(1);
00106       dup(p[1]);
00107       close(p[0]);
00108       close(p[1]);
00109       runcmd(pcmd->left);
00110     }
00111     if(fork1() == 0){
00112       close(0);
00113       dup(p[0]);
00114       close(p[0]);
00115       close(p[1]);
00116       runcmd(pcmd->right);
00117     }
00118     close(p[0]);
00119     close(p[1]);
00120     wait();
00121     wait();
00122     break;
00123
00124   case BACK:
00125     bcmd = (struct backcmd*)cmd;
00126     if(fork1() == 0)
00127       runcmd(bcmd->cmd);
00128     break;
00129   }
00130   exit();
00131 }
```

Referenced by main(), and runcmd().

5.172.3 Variable Documentation

5.172.3.1 symbols

```
char symbols[] = "<|>&;()"
```

Definition at line 263 of file sh.c.

Referenced by gettoken().

5.172.3.2 whitespace

```
char whitespace[] = " \t\r\n\v"
```

Definition at line 262 of file sh.c.

Referenced by gettoken(), and peek().

5.173 sh.c

Go to the documentation of this file.

```
00001 // Shell.
00002
00003 #include "types.h"
00004 #include "user.h"
00005 #include "fcntl.h"
00006
00007 // Parsed command representation
00008 #define EXEC  1
00009 #define REDIR 2
00010 #define PIPE  3
00011 #define LIST  4
00012 #define BACK  5
00013
00014 #define MAXARGS 10
00015
00016 struct cmd {
00017   int type;
00018 };
00019
00020 struct execcmd {
00021   int type;
00022   char *argv[MAXARGS];
00023   char *eargv[MAXARGS];
00024 };
00025
00026 struct redircmd {
00027   int type;
00028   struct cmd *cmd;
00029   char *file;
00030   char *efile;
00031   int mode;
00032   int fd;
00033 };
00034
00035 struct pipecmd {
00036   int type;
00037   struct cmd *left;
00038   struct cmd *right;
00039 };
00040
00041 struct listcmd {
00042   int type;
00043   struct cmd *left;
00044   struct cmd *right;
00045 };
00046
00047 struct backcmd {
00048   int type;
00049   struct cmd *cmd;
00050 };
00051
00052 int fork1(void);  // Fork but panics on failure.
00053 void panic(char*);
00054 struct cmd *parsecmd(char*);
00055
00056 // Execute cmd.  Never returns.
00057 void
00058 runcmd(struct cmd *cmd)
00059 {
00060   int p[2];
00061   struct backcmd *bcmd;
00062   struct execcmd *ecmd;
00063   struct listcmd *lcmd;
00064   struct pipecmd *pcmd;
00065   struct redircmd *rcmd;
00066
00067   if(cmd == 0)
00068     exit();
00069
00070   switch(cmd->type){
00071   default:
00072     panic("runcmd");
00073
00074   case EXEC:
00075     ecmd = (struct execcmd*)cmd;
00076     if(ecmd->argv[0] == 0)
00077       exit();
00078     exec(ecmd->argv[0], ecmd->argv);
00079     printf(2, "exec %s failed\n", ecmd->argv[0]);
00080     break;
00081
00082   case REDIR:
```

```
00083     rcmd = (struct redircmd*)cmd;
00084     close(rcmd->fd);
00085     if(open(rcmd->file, rcmd->mode) < 0){
00086       printf(2, "open %s failed\n", rcmd->file);
00087       exit();
00088     }
00089     runcmd(rcmd->cmd);
00090     break;
00091
00092   case LIST:
00093     lcmd = (struct listcmd*)cmd;
00094     if(fork1() == 0)
00095       runcmd(lcmd->left);
00096     wait();
00097     runcmd(lcmd->right);
00098     break;
00099
00100   case PIPE:
00101     pcmd = (struct pipecmd*)cmd;
00102     if(pipe(p) < 0)
00103       panic("pipe");
00104     if(fork1() == 0){
00105       close(1);
00106       dup(p[1]);
00107       close(p[0]);
00108       close(p[1]);
00109       runcmd(pcmd->left);
00110     }
00111     if(fork1() == 0){
00112       close(0);
00113       dup(p[0]);
00114       close(p[0]);
00115       close(p[1]);
00116       runcmd(pcmd->right);
00117     }
00118     close(p[0]);
00119     close(p[1]);
00120     wait();
00121     wait();
00122     break;
00123
00124   case BACK:
00125     bcmd = (struct backcmd*)cmd;
00126     if(fork1() == 0)
00127       runcmd(bcmd->cmd);
00128     break;
00129   }
00130   exit();
00131 }
00132
00133 int
00134 getcmd(char *buf, int nbuf)
00135 {
00136   printf(2, "$ ");
00137   memset(buf, 0, nbuf);
00138   gets(buf, nbuf);
00139   if(buf[0] == 0) // EOF
00140     return -1;
00141   return 0;
00142 }
00143
00144 int
00145 main(void)
00146 {
00147   static char buf[100];
00148   int fd;
00149
00150   // Ensure that three file descriptors are open.
00151   while((fd = open("console", O_RDWR)) >= 0){
00152     if(fd >= 3){
00153       close(fd);
00154       break;
00155     }
00156   }
00157
00158   // Read and run input commands.
00159   while(getcmd(buf, sizeof(buf)) >= 0){
00160     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
00161       // Chdir must be called by the parent, not the child.
00162       buf[strlen(buf)-1] = 0;  // chop \n
00163       if(chdir(buf+3) < 0)
00164         printf(2, "cannot cd %s\n", buf+3);
00165       continue;
00166     }
00167     if(fork1() == 0)
00168       runcmd(parsecmd(buf));
00169     wait();
```

```
00170   }
00171   exit();
00172 }
00173
00174 void
00175 panic(char *s)
00176 {
00177   printf(2, "%s\n", s);
00178   exit();
00179 }
00180
00181 int
00182 fork1(void)
00183 {
00184   int pid;
00185
00186   pid = fork();
00187   if(pid == -1)
00188     panic("fork");
00189   return pid;
00190 }
00191
00192 //PAGEBREAK!
00193 // Constructors
00194
00195 struct cmd*
00196 execcmd(void)
00197 {
00198   struct execcmd *cmd;
00199
00200   cmd = malloc(sizeof(*cmd));
00201   memset(cmd, 0, sizeof(*cmd));
00202   cmd->type = EXEC;
00203   return (struct cmd*)cmd;
00204 }
00205
00206 struct cmd*
00207 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
00208 {
00209   struct redircmd *cmd;
00210
00211   cmd = malloc(sizeof(*cmd));
00212   memset(cmd, 0, sizeof(*cmd));
00213   cmd->type = REDIR;
00214   cmd->cmd = subcmd;
00215   cmd->file = file;
00216   cmd->efile = efile;
00217   cmd->mode = mode;
00218   cmd->fd = fd;
00219   return (struct cmd*)cmd;
00220 }
00221
00222 struct cmd*
00223 pipecmd(struct cmd *left, struct cmd *right)
00224 {
00225   struct pipecmd *cmd;
00226
00227   cmd = malloc(sizeof(*cmd));
00228   memset(cmd, 0, sizeof(*cmd));
00229   cmd->type = PIPE;
00230   cmd->left = left;
00231   cmd->right = right;
00232   return (struct cmd*)cmd;
00233 }
00234
00235 struct cmd*
00236 listcmd(struct cmd *left, struct cmd *right)
00237 {
00238   struct listcmd *cmd;
00239
00240   cmd = malloc(sizeof(*cmd));
00241   memset(cmd, 0, sizeof(*cmd));
00242   cmd->type = LIST;
00243   cmd->left = left;
00244   cmd->right = right;
00245   return (struct cmd*)cmd;
00246 }
00247
00248 struct cmd*
00249 backcmd(struct cmd *subcmd)
00250 {
00251   struct backcmd *cmd;
00252
00253   cmd = malloc(sizeof(*cmd));
00254   memset(cmd, 0, sizeof(*cmd));
00255   cmd->type = BACK;
00256   cmd->cmd = subcmd;
```

```
00257   return (struct cmd*)cmd;
00258 }
00259 //PAGEBREAK!
00260 // Parsing
00261
00262 char whitespace[] = " \t\r\n\v";
00263 char symbols[] = "<|>&;()";
00264
00265 int
00266 gettoken(char **ps, char *es, char **q, char **eq)
00267 {
00268   char *s;
00269   int ret;
00270
00271   s = *ps;
00272   while(s < es && strchr(whitespace, *s))
00273     s++;
00274   if(q)
00275     *q = s;
00276   ret = *s;
00277   switch(*s){
00278   case 0:
00279     break;
00280   case '|':
00281   case '(':
00282   case ')':
00283   case ';':
00284   case '&':
00285   case '<':
00286     s++;
00287     break;
00288   case '>':
00289     s++;
00290     if(*s == '>'){
00291       ret = '+';
00292       s++;
00293     }
00294     break;
00295   default:
00296     ret = 'a';
00297     while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
00298       s++;
00299     break;
00300   }
00301   if(eq)
00302     *eq = s;
00303
00304   while(s < es && strchr(whitespace, *s))
00305     s++;
00306   *ps = s;
00307   return ret;
00308 }
00309
00310 int
00311 peek(char **ps, char *es, char *toks)
00312 {
00313   char *s;
00314
00315   s = *ps;
00316   while(s < es && strchr(whitespace, *s))
00317     s++;
00318   *ps = s;
00319   return *s && strchr(toks, *s);
00320 }
00321
00322 struct cmd *parseline(char**, char*);
00323 struct cmd *parsepipe(char**, char*);
00324 struct cmd *parseexec(char**, char*);
00325 struct cmd *nulterminate(struct cmd*);
00326
00327 struct cmd*
00328 parsecmd(char *s)
00329 {
00330   char *es;
00331   struct cmd *cmd;
00332
00333   es = s + strlen(s);
00334   cmd = parseline(&s, es);
00335   peek(&s, es, "");
00336   if(s != es){
00337     printf(2, "leftovers: %s\n", s);
00338     panic("syntax");
00339   }
00340   nulterminate(cmd);
00341   return cmd;
00342 }
00343
```

```
00344 struct cmd*
00345 parseline(char **ps, char *es)
00346 {
00347   struct cmd *cmd;
00348
00349   cmd = parsepipe(ps, es);
00350   while(peek(ps, es, "&")){
00351     gettoken(ps, es, 0, 0);
00352     cmd = backcmd(cmd);
00353   }
00354   if(peek(ps, es, ";")){
00355     gettoken(ps, es, 0, 0);
00356     cmd = listcmd(cmd, parseline(ps, es));
00357   }
00358   return cmd;
00359 }
00360
00361 struct cmd*
00362 parsepipe(char **ps, char *es)
00363 {
00364   struct cmd *cmd;
00365
00366   cmd = parseexec(ps, es);
00367   if(peek(ps, es, "|")){
00368     gettoken(ps, es, 0, 0);
00369     cmd = pipecmd(cmd, parsepipe(ps, es));
00370   }
00371   return cmd;
00372 }
00373
00374 struct cmd*
00375 parseredirs(struct cmd *cmd, char **ps, char *es)
00376 {
00377   int tok;
00378   char *q, *eq;
00379
00380   while(peek(ps, es, "<>")){
00381     tok = gettoken(ps, es, 0, 0);
00382     if(gettoken(ps, es, &q, &eq) != 'a')
00383       panic("missing file for redirection");
00384     switch(tok){
00385     case '<':
00386       cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
00387       break;
00388     case '>':
00389       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
00390       break;
00391     case '+':  // >>
00392       cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
00393       break;
00394     }
00395   }
00396   return cmd;
00397 }
00398
00399 struct cmd*
00400 parseblock(char **ps, char *es)
00401 {
00402   struct cmd *cmd;
00403
00404   if(!peek(ps, es, "("))
00405     panic("parseblock");
00406   gettoken(ps, es, 0, 0);
00407   cmd = parseline(ps, es);
00408   if(!peek(ps, es, ")"))
00409     panic("syntax - missing )");
00410   gettoken(ps, es, 0, 0);
00411   cmd = parseredirs(cmd, ps, es);
00412   return cmd;
00413 }
00414
00415 struct cmd*
00416 parseexec(char **ps, char *es)
00417 {
00418   char *q, *eq;
00419   int tok, argc;
00420   struct execcmd *cmd;
00421   struct cmd *ret;
00422
00423   if(peek(ps, es, "("))
00424     return parseblock(ps, es);
00425
00426   ret = execcmd();
00427   cmd = (struct execcmd*)ret;
00428
00429   argc = 0;
00430   ret = parseredirs(ret, ps, es);
```

```
00431   while(!peek(ps, es, "|)&;")){
00432     if((tok=gettoken(ps, es, &q, &eq)) == 0)
00433       break;
00434     if(tok != 'a')
00435       panic("syntax");
00436     cmd->argv[argc] = q;
00437     cmd->eargv[argc] = eq;
00438     argc++;
00439     if(argc >= MAXARGS)
00440       panic("too many args");
00441     ret = parseredirs(ret, ps, es);
00442   }
00443   cmd->argv[argc] = 0;
00444   cmd->eargv[argc] = 0;
00445   return ret;
00446 }
00447
00448 // NUL-terminate all the counted strings.
00449 struct cmd*
00450 nulterminate(struct cmd *cmd)
00451 {
00452   int i;
00453   struct backcmd *bcmd;
00454   struct execcmd *ecmd;
00455   struct listcmd *lcmd;
00456   struct pipecmd *pcmd;
00457   struct redircmd *rcmd;
00458
00459   if(cmd == 0)
00460     return 0;
00461
00462   switch(cmd->type){
00463   case EXEC:
00464     ecmd = (struct execcmd*)cmd;
00465     for(i=0; ecmd->argv[i]; i++)
00466       *ecmd->eargv[i] = 0;
00467     break;
00468
00469   case REDIR:
00470     rcmd = (struct redircmd*)cmd;
00471     nulterminate(rcmd->cmd);
00472     *rcmd->efile = 0;
00473     break;
00474
00475   case PIPE:
00476     pcmd = (struct pipecmd*)cmd;
00477     nulterminate(pcmd->left);
00478     nulterminate(pcmd->right);
00479     break;
00480
00481   case LIST:
00482     lcmd = (struct listcmd*)cmd;
00483     nulterminate(lcmd->left);
00484     nulterminate(lcmd->right);
00485     break;
00486
00487   case BACK:
00488     bcmd = (struct backcmd*)cmd;
00489     nulterminate(bcmd->cmd);
00490     break;
00491   }
00492   return cmd;
00493 }
```

5.174 sh.d File Reference

5.175 sh.d

Go to the documentation of this file.

```
00001 sh.o: sh.c /usr/include/stdc-predef.h types.h user.h fcntl.h
```

5.176 shutdown.c File Reference

```
#include "types.h"
#include "stat.h"
```

```
#include "user.h"
```

Functions

- int main (int argc, char *argv[])

5.176.1 Function Documentation

5.176.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Definition at line 10 of file shutdown.c.

```
00011 {
00012     halt();
00013     exit();
00014 }
```

5.177 shutdown.c

Go to the documentation of this file.

```
00001 // Shuts down the system by using the halt() system call
00002 // to send a special signal to QEMU.
00003 // Added by Bill Katsak
00004 // Copied from: http://pdos.csail.mit.edu/6.828/2012/homework/xv6-syscall.html
00005
00006 #include "types.h"
00007 #include "stat.h"
00008 #include "user.h"
00009
00010 int main(int argc, char *argv[])
00011 {
00012     halt();
00013     exit();
00014 }
```

5.178 shutdown.d File Reference

5.179 shutdown.d

Go to the documentation of this file.

```
00001 shutdown.o: shutdown.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.180 sleeplock.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"
#include "sleeplock.h"
```

Functions

- void acquiresleep (struct sleeplock *lk)
- int holdingsleep (struct sleeplock *lk)
- void initsleeplock (struct sleeplock *lk, char *name)
- void releasesleep (struct sleeplock *lk)

5.180.1 Function Documentation

5.180.1.1 acquiresleep()

```
void acquiresleep (
            struct sleeplock * lk )
```

Definition at line 23 of file sleeplock.c.

```
00024 {
00025   acquire(&lk->lk);
00026   while (lk->locked) {
00027     sleep(lk, &lk->lk);
00028   }
00029   lk->locked = 1;
00030   lk->pid = myproc()->pid;
00031   release(&lk->lk);
00032 }
```

Referenced by bget(), ilock(), and iput().

5.180.1.2 holdingsleep()

```
int holdingsleep (
            struct sleeplock * lk )
```

Definition at line 45 of file sleeplock.c.

```
00046 {
00047   int r;
00048
00049   acquire(&lk->lk);
00050   r = lk->locked && (lk->pid == myproc()->pid);
00051   release(&lk->lk);
00052   return r;
00053 }
```

Referenced by brelse(), bwrite(), iderw(), and iunlock().

5.180.1.3 initsleeplock()

```
void initsleeplock (
              struct sleeplock * lk,
              char * name )
```

Definition at line 14 of file sleeplock.c.

```
00015 {
00016   initlock(&lk->lk, "sleep lock");
00017   lk->name = name;
00018   lk->locked = 0;
00019   lk->pid = 0;
00020 }
```

Referenced by binit(), and iinit().

5.180.1.4 releasesleep()

```
void releasesleep (
              struct sleeplock * lk )
```

Definition at line 35 of file sleeplock.c.

```
00036 {
00037   acquire(&lk->lk);
00038   lk->locked = 0;
00039   lk->pid = 0;
00040   wakeup(lk);
00041   release(&lk->lk);
00042 }
```

Referenced by brelse(), iput(), and iunlock().

5.181 sleeplock.c

Go to the documentation of this file.

```
00001 // Sleeping locks
00002
00003 #include "types.h"
00004 #include "defs.h"
00005 #include "param.h"
00006 #include "x86.h"
00007 #include "memlayout.h"
00008 #include "mmu.h"
00009 #include "proc.h"
00010 #include "spinlock.h"
00011 #include "sleeplock.h"
00012
00013 void
00014 initsleeplock(struct sleeplock *lk, char *name)
00015 {
00016   initlock(&lk->lk, "sleep lock");
00017   lk->name = name;
00018   lk->locked = 0;
00019   lk->pid = 0;
00020 }
00021
00022 void
00023 acquiresleep(struct sleeplock *lk)
00024 {
00025   acquire(&lk->lk);
00026   while (lk->locked) {
00027     sleep(lk, &lk->lk);
00028   }
00029   lk->locked = 1;
00030   lk->pid = myproc()->pid;
00031   release(&lk->lk);
00032 }
```

```
00033
00034 void
00035 releasesleep(struct sleeplock *lk)
00036 {
00037   acquire(&lk->lk);
00038   lk->locked = 0;
00039   lk->pid = 0;
00040   wakeup(lk);
00041   release(&lk->lk);
00042 }
00043
00044 int
00045 holdingsleep(struct sleeplock *lk)
00046 {
00047   int r;
00048
00049   acquire(&lk->lk);
00050   r = lk->locked && (lk->pid == myproc()->pid);
00051   release(&lk->lk);
00052   return r;
00053 }
00054
00055
00056
```

5.182 sleeplock.d File Reference

5.183 sleeplock.d

Go to the documentation of this file.

```
00001 sleeplock.o: sleeplock.c /usr/include/stdc-predef.h types.h defs.h \
00002  param.h x86.h memlayout.h mmu.h proc.h spinlock.h sleeplock.h
```

5.184 sleeplock.h File Reference

Classes

- struct sleeplock

5.185 sleeplock.h

Go to the documentation of this file.

```
00001 // Long-term locks for processes
00002 struct sleeplock {
00003   uint locked;       // Is the lock held?
00004   struct spinlock lk; // spinlock protecting this sleep lock
00005
00006   // For debugging:
00007   char *name;        // Name of lock.
00008   int pid;           // Process holding lock
00009 };
00010
```

5.186 spinlock.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"
```

Functions

- void acquire (struct spinlock *lk)
- void getcallerpcs (void *v, uint pcs[])
- int holding (struct spinlock *lock)
- void initlock (struct spinlock *lk, char *name)
- void popcli (void)
- void pushcli (void)
- void release (struct spinlock *lk)

5.186.1 Function Documentation

5.186.1.1 acquire()

```
void acquire (
            struct spinlock * lk )
```

Definition at line 25 of file spinlock.c.

```
00026 {
00027   pushcli(); // disable interrupts to avoid deadlock.
00028   if(holding(lk))
00029     panic("acquire");
00030
00031   // The xchg is atomic.
00032   while(xchg(&lk->locked, 1) != 0)
00033     ;
00034
00035   // Tell the C compiler and the processor to not move loads or stores
00036   // past this point, to ensure that the critical section's memory
00037   // references happen after the lock is acquired.
00038   __sync_synchronize();
00039
00040   // Record info about lock acquisition for debugging.
00041   lk->cpu = mycpu();
00042   getcallerpcs(&lk, lk->pcs);
00043 }
```

Referenced by acquiresleep(), allocproc(), begin_op(), bget(), brelse(), chpr(), consoleintr(), consoleread(), consolewrite(), cprintf(), cps(), end_op(), exit(), filealloc(), fileclose(), filedup(), fork(), holdingsleep(), ideintr(), iderw(), idup(), iget(), iput(), kalloc(), kfree(), kill(), log_write(), pipeclose(), piperead(), pipewrite(), releasesleep(), scheduler(), sleep(), sys_sleep(), sys_uptime(), trap(), userinit(), wait(), wakeup(), and yield().

5.186.1.2 getcallerpcs()

```
void getcallerpcs (
            void * v,
            uint pcs[] )
```

Definition at line 72 of file spinlock.c.

```
00073 {
00074   uint *ebp;
00075   int i;
00076
00077   ebp = (uint*)v - 2;
00078   for(i = 0; i < 10; i++){
00079     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
00080       break;
00081     pcs[i] = ebp[1];     // saved %eip
00082     ebp = (uint*)ebp[0]; // saved %ebp
00083   }
00084   for(; i < 10; i++)
00085     pcs[i] = 0;
00086 }
```

Referenced by acquire().

5.186.1.3 holding()

```
int holding (
            struct spinlock * lock )
```

Definition at line 90 of file spinlock.c.

```
00091 {
00092   int r;
00093   pushcli();
00094   r = lock->locked && lock->cpu == mycpu();
00095   popcli();
00096   return r;
00097 }
```

Referenced by acquire(), release(), and sched().

5.186.1.4 initlock()

```
void initlock (
            struct spinlock * lk,
            char * name )
```

Definition at line 13 of file spinlock.c.

```
00014 {
00015   lk->name = name;
00016   lk->locked = 0;
00017   lk->cpu = 0;
00018 }
```

Referenced by binit(), consoleinit(), fileinit(), ideinit(), iinit(), initlog(), initsleeplock(), kinit1(), pinit(), pipealloc(), and tvinit().

5.186.1.5 popcli()

```
void popcli (
            void  )
```

Definition at line 117 of file spinlock.c.

```
00118 {
00119   if(readeflags()&FL_IF)
00120     panic("popcli - interruptible");
00121   if(--mycpu()->ncli < 0)
00122     panic("popcli");
00123   if(mycpu()->ncli == 0 && mycpu()->intena)
00124     sti();
00125 }
```

Referenced by holding(), myproc(), release(), and switchuvm().

5.186.1.6 pushcli()

```
void pushcli (
              void  )
```

Definition at line 105 of file spinlock.c.

```
00106 {
00107   int eflags;
00108
00109   eflags = readeflags();
00110   cli();
00111   if(mycpu()->ncli == 0)
00112     mycpu()->intena = eflags & FL_IF;
00113   mycpu()->ncli += 1;
00114 }
```

Referenced by acquire(), holding(), myproc(), and switchuvm().

5.186.1.7 release()

```
void release (
              struct spinlock * lk )
```

Definition at line 47 of file spinlock.c.

```
00048 {
00049   if(!holding(lk))
00050     panic("release");
00051
00052   lk->pcs[0] = 0;
00053   lk->cpu = 0;
00054
00055   // Tell the C compiler and the processor to not move loads or stores
00056   // past this point, to ensure that all the stores in the critical
00057   // section are visible to other cores before the lock is released.
00058   // Both the C compiler and the hardware may re-order loads and
00059   // stores; __sync_synchronize() tells them both not to.
00060   __sync_synchronize();
00061
00062   // Release the lock, equivalent to lk->locked = 0.
00063   // This code can't use a C assignment, since it might
00064   // not be atomic. A real OS would use C atomics here.
00065   asm volatile("movl $0, %0" : "+m" (lk->locked) : );
00066
00067   popcli();
00068 }
```

Referenced by acquiresleep(), allocproc(), begin_op(), bget(), brelse(), chpr(), consoleintr(), consoleread(), consolewrite(), cprintf(), cps(), end_op(), filealloc(), fileclose(), filedup(), fork(), forkret(), holdingsleep(), ideintr(), iderw(), idup(), iget(), iput(), kalloc(), kfree(), kill(), log_write(), pipeclose(), piperead(), pipewrite(), releasesleep(), scheduler(), sleep(), sys_sleep(), sys_uptime(), trap(), userinit(), wait(), wakeup(), and yield().

5.187 spinlock.c

Go to the documentation of this file.

```
00001 // Mutual exclusion spin locks.
00002
00003 #include "types.h"
00004 #include "defs.h"
00005 #include "param.h"
00006 #include "x86.h"
00007 #include "memlayout.h"
00008 #include "mmu.h"
00009 #include "proc.h"
00010 #include "spinlock.h"
00011
00012 void
```

```
00013 initlock(struct spinlock *lk, char *name)
00014 {
00015   lk->name = name;
00016   lk->locked = 0;
00017   lk->cpu = 0;
00018 }
00019
00020 // Acquire the lock.
00021 // Loops (spins) until the lock is acquired.
00022 // Holding a lock for a long time may cause
00023 // other CPUs to waste time spinning to acquire it.
00024 void
00025 acquire(struct spinlock *lk)
00026 {
00027   pushcli(); // disable interrupts to avoid deadlock.
00028   if(holding(lk))
00029     panic("acquire");
00030
00031   // The xchg is atomic.
00032   while(xchg(&lk->locked, 1) != 0)
00033     ;
00034
00035   // Tell the C compiler and the processor to not move loads or stores
00036   // past this point, to ensure that the critical section's memory
00037   // references happen after the lock is acquired.
00038   __sync_synchronize();
00039
00040   // Record info about lock acquisition for debugging.
00041   lk->cpu = mycpu();
00042   getcallerpcs(&lk, lk->pcs);
00043 }
00044
00045 // Release the lock.
00046 void
00047 release(struct spinlock *lk)
00048 {
00049   if(!holding(lk))
00050     panic("release");
00051
00052   lk->pcs[0] = 0;
00053   lk->cpu = 0;
00054
00055   // Tell the C compiler and the processor to not move loads or stores
00056   // past this point, to ensure that all the stores in the critical
00057   // section are visible to other cores before the lock is released.
00058   // Both the C compiler and the hardware may re-order loads and
00059   // stores; __sync_synchronize() tells them both not to.
00060   __sync_synchronize();
00061
00062   // Release the lock, equivalent to lk->locked = 0.
00063   // This code can't use a C assignment, since it might
00064   // not be atomic. A real OS would use C atomics here.
00065   asm volatile("movl $0, %0" : "+m" (lk->locked) : );
00066
00067   popcli();
00068 }
00069
00070 // Record the current call stack in pcs[] by following the %ebp chain.
00071 void
00072 getcallerpcs(void *v, uint pcs[])
00073 {
00074   uint *ebp;
00075   int i;
00076
00077   ebp = (uint*)v - 2;
00078   for(i = 0; i < 10; i++){
00079     if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
00080       break;
00081     pcs[i] = ebp[1];     // saved %eip
00082     ebp = (uint*)ebp[0]; // saved %ebp
00083   }
00084   for(; i < 10; i++)
00085     pcs[i] = 0;
00086 }
00087
00088 // Check whether this cpu is holding the lock.
00089 int
00090 holding(struct spinlock *lock)
00091 {
00092   int r;
00093   pushcli();
00094   r = lock->locked && lock->cpu == mycpu();
00095   popcli();
00096   return r;
00097 }
00098
00099
```

```
00100 // Pushcli/popcli are like cli/sti except that they are matched:
00101 // it takes two popcli to undo two pushcli.  Also, if interrupts
00102 // are off, then pushcli, popcli leaves them off.
00103
00104 void
00105 pushcli(void)
00106 {
00107   int eflags;
00108
00109   eflags = readeflags();
00110   cli();
00111   if(mycpu()->ncli == 0)
00112     mycpu()->intena = eflags & FL_IF;
00113   mycpu()->ncli += 1;
00114 }
00115
00116 void
00117 popcli(void)
00118 {
00119   if(readeflags()&FL_IF)
00120     panic("popcli - interruptible");
00121   if(--mycpu()->ncli < 0)
00122     panic("popcli");
00123   if(mycpu()->ncli == 0 && mycpu()->intena)
00124     sti();
00125 }
00126
```

5.188 spinlock.d File Reference

5.189 spinlock.d

Go to the documentation of this file.

```
00001 spinlock.o: spinlock.c /usr/include/stdc-predef.h types.h defs.h param.h \
00002  x86.h memlayout.h mmu.h proc.h spinlock.h
```

5.190 spinlock.h File Reference

Classes

- struct spinlock

5.191 spinlock.h

Go to the documentation of this file.

```
00001 // Mutual exclusion lock.
00002 struct spinlock {
00003   uint locked;       // Is the lock held?
00004
00005   // For debugging:
00006   char *name;        // Name of lock.
00007   struct cpu *cpu;   // The cpu holding the lock.
00008   uint pcs[10];      // The call stack (an array of program counters)
00009                      // that locked the lock.
00010 };
00011
```

5.192 stat.h File Reference

Classes

- struct stat

Macros

- #define T_DEV 3
- #define T_DIR 1
- #define T_FILE 2

5.192.1 Macro Definition Documentation

5.192.1.1 T_DEV

```
#define T_DEV 3
```

Definition at line 3 of file stat.h.

5.192.1.2 T_DIR

```
#define T_DIR 1
```

Definition at line 1 of file stat.h.

5.192.1.3 T_FILE

```
#define T_FILE 2
```

Definition at line 2 of file stat.h.

5.193 stat.h

Go to the documentation of this file.

```
00001 #define T_DIR  1   // Directory
00002 #define T_FILE 2   // File
00003 #define T_DEV  3   // Device
00004
00005 struct stat {
00006   short type;  // Type of file
00007   int dev;     // File system's disk device
00008   uint ino;    // Inode number
00009   short nlink; // Number of links to file
00010   uint size;   // Size of file in bytes
00011 };
```

5.194 stressfs.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fs.h"
#include "fcntl.h"
```

Functions

- int main (int argc, char *argv[])

5.194.1 Function Documentation

5.194.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Definition at line 17 of file stressfs.c.

```
00018 {
00019   int fd, i;
00020   char path[] = "stressfs0";
00021   char data[512];
00022
00023   printf(1, "stressfs starting\n");
00024   memset(data, 'a', sizeof(data));
00025
00026   for(i = 0; i < 4; i++)
00027     if(fork() > 0)
00028       break;
00029
00030   printf(1, "write %d\n", i);
00031
00032   path[8] += i;
00033   fd = open(path, O_CREATE | O_RDWR);
00034   for(i = 0; i < 20; i++)
00035 //    printf(fd, "%d\n", i);
00036     write(fd, data, sizeof(data));
00037   close(fd);
00038
00039   printf(1, "read\n");
00040
00041   fd = open(path, O_RDONLY);
00042   for (i = 0; i < 20; i++)
00043     read(fd, data, sizeof(data));
00044   close(fd);
00045
00046   wait();
00047
00048   exit();
00049 }
```

5.195 stressfs.c

Go to the documentation of this file.

```
00001 // Demonstrate that moving the "acquire" in iderw after the loop that
00002 // appends to the idequeue results in a race.
00003
00004 // For this to work, you should also add a spin within iderw's
00005 // idequeue traversal loop.  Adding the following demonstrated a panic
00006 // after about 5 runs of stressfs in QEMU on a 2.1GHz CPU:
00007 //    for (i = 0; i < 40000; i++)
00008 //      asm volatile("");
00009
00010 #include "types.h"
00011 #include "stat.h"
00012 #include "user.h"
00013 #include "fs.h"
00014 #include "fcntl.h"
00015
00016 int
00017 main(int argc, char *argv[])
00018 {
00019   int fd, i;
00020   char path[] = "stressfs0";
00021   char data[512];
00022
00023   printf(1, "stressfs starting\n");
00024   memset(data, 'a', sizeof(data));
00025
00026   for(i = 0; i < 4; i++)
00027     if(fork() > 0)
00028       break;
00029
00030   printf(1, "write %d\n", i);
00031
00032   path[8] += i;
00033   fd = open(path, O_CREATE | O_RDWR);
00034   for(i = 0; i < 20; i++)
00035 //    printf(fd, "%d\n", i);
00036     write(fd, data, sizeof(data));
00037   close(fd);
00038
00039   printf(1, "read\n");
00040
00041   fd = open(path, O_RDONLY);
00042   for (i = 0; i < 20; i++)
00043     read(fd, data, sizeof(data));
00044   close(fd);
00045
00046   wait();
00047
00048   exit();
00049 }
```

5.196 stressfs.d File Reference

5.197 stressfs.d

Go to the documentation of this file.

```
00001 stressfs.o: stressfs.c /usr/include/stdc-predef.h types.h stat.h user.h \
00002  fs.h fcntl.h
```

5.198 string.c File Reference

```
#include "types.h"
#include "x86.h"
```

Functions

- int memcmp (const void *v1, const void *v2, uint n)
- void * memcpy (void *dst, const void *src, uint n)
- void * memmove (void *dst, const void *src, uint n)
- void * memset (void *dst, int c, uint n)
- char * safestrcpy (char *s, const char *t, int n)
- int strlen (const char *s)
- int strncmp (const char *p, const char *q, uint n)
- char * strncpy (char *s, const char *t, int n)

5.198.1 Function Documentation

5.198.1.1 memcmp()

```
int memcmp (
            const void * v1,
            const void * v2,
            uint n )
```

Definition at line 16 of file string.c.

```
00017 {
00018   const uchar *s1, *s2;
00019
00020   s1 = v1;
00021   s2 = v2;
00022   while(n-- > 0){
00023     if(*s1 != *s2)
00024       return *s1 - *s2;
00025     s1++, s2++;
00026   }
00027
00028   return 0;
00029 }
```

Referenced by cmostime(), mpconfig(), and mpsearch1().

5.198.1.2 memcpy()

```
void * memcpy (
            void * dst,
            const void * src,
            uint n )
```

Definition at line 53 of file string.c.

```
00054 {
00055   return memmove(dst, src, n);
00056 }
```

5.198.1.3 memmove()

```
void * memmove (
            void * dst,
            const void * src,
            uint n )
```

Definition at line 32 of file string.c.

```
00033 {
00034   const char *s;
00035   char *d;
00036
00037   s = src;
00038   d = dst;
00039   if(s < d && s + n > d){
00040     s += n;
00041     d += n;
00042     while(n-- > 0)
00043       *--d = *--s;
00044   } else
00045     while(n-- > 0)
00046       *d++ = *s++;
00047
00048   return dst;
00049 }
```

Referenced by cgaputc(), copyout(), copyuvm(), fmtname(), grep(), iderw(), ilock(), inituvm(), install_trans(), iupdate(), ls(), main(), memcpy(), readi(), readsb(), skipelem(), startothers(), write_log(), and writei().

5.198.1.4 memset()

```
void * memset (
            void * dst,
            int c,
            uint n )
```

Definition at line 5 of file string.c.

```
00006 {
00007   if ((int)dst%4 == 0 && n%4 == 0){
00008     c &= 0xFF;
00009     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
00010   } else
00011     stosb(dst, c, n);
00012   return dst;
00013 }
```

5.198.1.5 safestrcpy()

```
char * safestrcpy (
            char * s,
            const char * t,
            int n )
```

Definition at line 83 of file string.c.

```
00084 {
00085   char *os;
00086
00087   os = s;
00088   if(n <= 0)
00089     return os;
00090   while(--n > 0 && (*s++ = *t++) != 0)
00091     ;
00092   *s = 0;
00093   return os;
00094 }
```

Referenced by exec(), fork(), and userinit().

5.198.1.6 strlen()

```
int strlen (
            const char * s )
```

Definition at line 97 of file string.c.

```
00098 {
00099   int n;
00100
00101   for(n = 0; s[n]; n++)
00102     ;
00103   return n;
00104 }
```

5.198.1.7 strncmp()

```
int strncmp (
            const char * p,
            const char * q,
            uint n )
```

Definition at line 59 of file string.c.

```
00060 {
00061   while(n > 0 && *p && *p == *q)
00062     n--, p++, q++;
00063   if(n == 0)
00064     return 0;
00065   return (uchar)*p - (uchar)*q;
00066 }
```

Referenced by namecmp().

5.198.1.8 strncpy()

```
char * strncpy (
            char * s,
            const char * t,
            int n )
```

Definition at line 69 of file string.c.

```
00070 {
00071   char *os;
00072
00073   os = s;
00074   while(n-- > 0 && (*s++ = *t++) != 0)
00075     ;
00076   while(n-- > 0)
00077     *s++ = 0;
00078   return os;
00079 }
```

Referenced by dirlink(), main(), and sys_getprocs().

5.199 string.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "x86.h"
00003
00004 void*
00005 memset(void *dst, int c, uint n)
00006 {
00007   if ((int)dst%4 == 0 && n%4 == 0){
00008     c &= 0xFF;
00009     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
00010   } else
00011     stosb(dst, c, n);
00012   return dst;
00013 }
00014
00015 int
00016 memcmp(const void *v1, const void *v2, uint n)
00017 {
00018   const uchar *s1, *s2;
00019
00020   s1 = v1;
00021   s2 = v2;
00022   while(n-- > 0){
00023     if(*s1 != *s2)
00024       return *s1 - *s2;
00025     s1++, s2++;
00026   }
00027
00028   return 0;
00029 }
00030
00031 void*
00032 memmove(void *dst, const void *src, uint n)
00033 {
00034   const char *s;
00035   char *d;
00036
00037   s = src;
00038   d = dst;
00039   if(s < d && s + n > d){
00040     s += n;
00041     d += n;
00042     while(n-- > 0)
00043       *--d = *--s;
00044   } else
00045     while(n-- > 0)
00046       *d++ = *s++;
00047
00048   return dst;
00049 }
00050
00051 // memcpy exists to placate GCC.  Use memmove.
00052 void*
00053 memcpy(void *dst, const void *src, uint n)
00054 {
00055   return memmove(dst, src, n);
00056 }
00057
00058 int
00059 strncmp(const char *p, const char *q, uint n)
00060 {
00061   while(n > 0 && *p && *p == *q)
00062     n--, p++, q++;
00063   if(n == 0)
00064     return 0;
00065   return (uchar)*p - (uchar)*q;
00066 }
00067
00068 char*
00069 strncpy(char *s, const char *t, int n)
00070 {
00071   char *os;
00072
00073   os = s;
00074   while(n-- > 0 && (*s++ = *t++) != 0)
00075     ;
00076   while(n-- > 0)
00077     *s++ = 0;
00078   return os;
00079 }
00080
00081 // Like strncpy but guaranteed to NUL-terminate.
00082 char*
```

```
00083 safestrcpy(char *s, const char *t, int n)
00084 {
00085   char *os;
00086
00087   os = s;
00088   if(n <= 0)
00089     return os;
00090   while(--n > 0 && (*s++ = *t++) != 0)
00091     ;
00092   *s = 0;
00093   return os;
00094 }
00095
00096 int
00097 strlen(const char *s)
00098 {
00099   int n;
00100
00101   for(n = 0; s[n]; n++)
00102     ;
00103   return n;
00104 }
00105
```

5.200 string.d File Reference

5.201 string.d

Go to the documentation of this file.

```
00001 string.o: string.c /usr/include/stdc-predef.h types.h x86.h
```

5.202 syscall.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "syscall.h"
```

Functions

- int argint (int n, int *ip)
- int argptr (int n, char **pp, int size)
- int argstr (int n, char **pp)
- int fetchint (uint addr, int *ip)
- int fetchstr (uint addr, char **pp)
- int sys_chdir (void)
- int sys_chpr (void)
- int sys_close (void)
- int sys_cps (void)
- int sys_dup (void)
- int sys_exec (void)
- int sys_exit (void)

- int sys_fork (void)
- int sys_fstat (void)
- int sys_getpid (void)
- int sys_getprocs (void)
- int sys_halt (void)
- int sys_kill (void)
- int sys_link (void)
- int sys_mkdir (void)
- int sys_mknod (void)
- int sys_open (void)
- int sys_pipe (void)
- int sys_read (void)
- int sys_sbrk (void)
- int sys_sleep (void)
- int sys_unlink (void)
- int sys_uptime (void)
- int sys_wait (void)
- int sys_write (void)
- void syscall (void)

Variables

- static int(* syscalls [ ])(void)

5.202.1 Function Documentation

5.202.1.1 argint()

```
int argint (
              int n,
              int * ip )
```

Definition at line 50 of file syscall.c.

```
00051 {
00052   return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
00053 }
```

Referenced by argfd(), argptr(), argstr(), sys_chpr(), sys_exec(), sys_getprocs(), sys_kill(), sys_mknod(), sys_open(), sys_read(), sys_sbrk(), sys_sleep(), and sys_write().

5.202.1.2 argptr()

```
int argptr (
              int n,
              char ** pp,
              int size )
```

Definition at line 59 of file syscall.c.

```
00060 {
00061   int i;
00062   struct proc *curproc = myproc();
00063
00064   if(argint(n, &i) < 0)
00065     return -1;
00066   if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
00067     return -1;
00068   *pp = (char*)i;
00069   return 0;
00070 }
```

Referenced by sys_fstat(), sys_getprocs(), sys_pipe(), sys_read(), and sys_write().

5.202.1.3 argstr()

```
int argstr (
              int n,
              char ** pp )
```

Definition at line 77 of file syscall.c.

```
00078 {
00079   int addr;
00080   if(argint(n, &addr) < 0)
00081     return -1;
00082   return fetchstr(addr, pp);
00083 }
```

Referenced by sys_chdir(), sys_exec(), sys_link(), sys_mkdir(), sys_mknod(), sys_open(), and sys_unlink().

5.202.1.4 fetchint()

```
int fetchint (
              uint addr,
              int * ip )
```

Definition at line 18 of file syscall.c.

```
00019 {
00020   struct proc *curproc = myproc();
00021
00022   if(addr >= curproc->sz || addr+4 > curproc->sz)
00023     return -1;
00024   *ip = *(int*)(addr);
00025   return 0;
00026 }
```

Referenced by argint(), and sys_exec().

5.202.1.5 fetchstr()

```
int fetchstr (
              uint addr,
              char ** pp )
```

Definition at line 32 of file syscall.c.

```
00033 {
00034   char *s, *ep;
00035   struct proc *curproc = myproc();
00036
00037   if(addr >= curproc->sz)
00038     return -1;
00039   *pp = (char*)addr;
00040   ep = (char*)curproc->sz;
00041   for(s = *pp; s < ep; s++){
00042     if(*s == 0)
00043       return s - *pp;
00044   }
00045   return -1;
00046 }
```

Referenced by argstr(), and sys_exec().

5.202.1.6 sys_chdir()

```
int sys_chdir (
              void  )
```

Definition at line 372 of file sysfile.c.

```
00373 {
00374   char *path;
00375   struct inode *ip;
00376   struct proc *curproc = myproc();
00377
00378   begin_op();
00379   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
00380     end_op();
00381     return -1;
00382   }
00383   ilock(ip);
00384   if(ip->type != T_DIR){
00385     iunlockput(ip);
00386     end_op();
00387     return -1;
00388   }
00389   iunlock(ip);
00390   iput(curproc->cwd);
00391   end_op();
00392   curproc->cwd = ip;
00393   return 0;
00394 }
```

5.202.1.7 sys_chpr()

```
int sys_chpr (
              void  )
```

Definition at line 125 of file sysproc.c.

```
00126 {
00127   int pid, pr;
00128   if(argint(0, &pid) < 0)
00129     return -1;
00130   if(argint(1, &pr) < 0)
00131     return -1;
00132
00133   return chpr(pid, pr);
00134 }
```

5.202.1.8 sys_close()

```
int sys_close (
              void  )
```

Definition at line 94 of file sysfile.c.

```
00095 {
00096   int fd;
00097   struct file *f;
00098
00099   if(argfd(0, &fd, &f) < 0)
00100     return -1;
00101   myproc()->ofile[fd] = 0;
00102   fileclose(f);
00103   return 0;
00104 }
```

5.202.1.9 sys_cps()

```
int sys_cps (
              void  )
```

Definition at line 119 of file sysproc.c.

```
00120 {
00121   return cps();
00122 }
```

5.202.1.10 sys_dup()

```
int sys_dup (
              void  )
```

Definition at line 56 of file sysfile.c.

```
00057 {
00058   struct file *f;
00059   int fd;
00060
00061   if(argfd(0, 0, &f) < 0)
00062     return -1;
00063   if((fd=fdalloc(f)) < 0)
00064     return -1;
00065   filedup(f);
00066   return fd;
00067 }
```

5.202.1.11 sys_exec()

```
int sys_exec (
              void  )
```

Definition at line 397 of file sysfile.c.

```
00398 {
00399   char *path, *argv[MAXARG];
00400   int i;
00401   uint uargv, uarg;
00402
00403   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
00404     return -1;
00405   }
00406   memset(argv, 0, sizeof(argv));
00407   for(i=0;; i++){
00408     if(i >= NELEM(argv))
00409       return -1;
00410     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
00411       return -1;
00412     if(uarg == 0){
00413       argv[i] = 0;
00414       break;
00415     }
00416     if(fetchstr(uarg, &argv[i]) < 0)
00417       return -1;
00418   }
00419   return exec(path, argv);
00420 }
```

5.202.1.12 sys_exit()

```
int sys_exit (
              void  )
```

Definition at line 25 of file sysproc.c.

```
00026 {
00027   exit();
00028   return 0;  // not reached
00029 }
```

5.202.1.13 sys_fork()

```
int sys_fork (
              void  )
```

Definition at line 19 of file sysproc.c.

```
00020 {
00021   return fork();
00022 }
```

5.202.1.14 sys_fstat()

```
int sys_fstat (
              void  )
```

Definition at line 107 of file sysfile.c.

```
00108 {
00109   struct file *f;
00110   struct stat *st;
00111
00112   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
00113     return -1;
00114   return filestat(f, st);
00115 }
```

5.202.1.15 sys_getpid()

```
int sys_getpid (
              void  )
```

Definition at line 48 of file sysproc.c.

```
00049 {
00050   return myproc()->pid;
00051 }
```

5.202.1.16 sys_getprocs()

```
int sys_getprocs (
              void  )
```

Definition at line 137 of file sysproc.c.

```
00137                   {
00138   // declare local variables for max uproc size, struct uproc, and counter
00139   int max;
00140   struct uproc *p;
00141   int i=0;
00142
00143   // if argint has trouble, return error -1
00144   if(argint(0,&max) < 0)
00145     return -1;
00146
00147   // if argptr for allocating struct size has trouble, return -1
00148   if(argptr(1,(char**)&p, max*sizeof(struct uproc)) < 0)
00149     return -1;
00150
00151   // create pointer to ptable processes
00152   struct proc *ptr = ptable.proc;
00153
00154   // loop through ptable
00155   for(; ptr < &ptable.proc[NPROC]; ptr++)
00156   {
00157     if(!(ptr->state == UNUSED))
00158     {
00159       // if the process in ptable is not UNUSED, assign pid, parent pid, and name to uproc
00160       p[i].pid = ptr->pid;
00161       p[i].ppid = ptr->parent->pid;
00162       strncpy(p[i].name, ptr->name, 16);
00163       // add 1 to the process counter
00164       i++;
00165     }
00166   }
00167
00168   // return the number of processes that are not UNUSED
00169   return i;
00170 }
```

5.202.1.17 sys_halt()

```
int sys_halt (
              void  )
```

Definition at line 111 of file sysproc.c.

```
00112 {
00113   outb(0xf4, 0x00);
00114   return 0;
00115 }
```

5.202.1.18 sys_kill()

```
int sys_kill (
              void  )
```

Definition at line 38 of file sysproc.c.

```
00039 {
00040   int pid;
00041
00042   if(argint(0, &pid) < 0)
00043     return -1;
00044   return kill(pid);
00045 }
```

5.202.1.19 sys_link()

```
int sys_link (
              void  )
```

Definition at line 119 of file sysfile.c.

```
00120 {
00121   char name[DIRSIZ], *new, *old;
00122   struct inode *dp, *ip;
00123
00124   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
00125     return -1;
00126
00127   begin_op();
00128   if((ip = namei(old)) == 0){
00129     end_op();
00130     return -1;
00131   }
00132
00133   ilock(ip);
00134   if(ip->type == T_DIR){
00135     iunlockput(ip);
00136     end_op();
00137     return -1;
00138   }
00139
00140   ip->nlink++;
00141   iupdate(ip);
00142   iunlock(ip);
00143
00144   if((dp = nameiparent(new, name)) == 0)
00145     goto bad;
00146   ilock(dp);
00147   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
00148     iunlockput(dp);
00149     goto bad;
00150   }
00151   iunlockput(dp);
00152   iput(ip);
00153
00154   end_op();
00155
00156   return 0;
00157
00158 bad:
00159   ilock(ip);
00160   ip->nlink--;
00161   iupdate(ip);
00162   iunlockput(ip);
00163   end_op();
00164   return -1;
00165 }
```

5.202.1.20 sys_mkdir()

```
int sys_mkdir (
              void  )
```

Definition at line 336 of file sysfile.c.

```
00337 {
00338   char *path;
00339   struct inode *ip;
00340
00341   begin_op();
00342   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
00343     end_op();
00344     return -1;
00345   }
00346   iunlockput(ip);
00347   end_op();
00348   return 0;
00349 }
```

5.202.1.21 sys_mknod()

```
int sys_mknod (
              void  )
```

Definition at line 352 of file sysfile.c.

```
00353 {
00354   struct inode *ip;
00355   char *path;
00356   int major, minor;
00357
00358   begin_op();
00359   if((argstr(0, &path)) < 0 ||
00360      argint(1, &major) < 0 ||
00361      argint(2, &minor) < 0 ||
00362      (ip = create(path, T_DEV, major, minor)) == 0){
00363     end_op();
00364     return -1;
00365   }
00366   iunlockput(ip);
00367   end_op();
00368   return 0;
00369 }
```

5.202.1.22 sys_open()

```
int sys_open (
              void  )
```

Definition at line 286 of file sysfile.c.

```
00287 {
00288   char *path;
00289   int fd, omode;
00290   struct file *f;
00291   struct inode *ip;
00292
00293   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
00294     return -1;
00295
00296   begin_op();
00297
00298   if(omode & O_CREATE){
00299     ip = create(path, T_FILE, 0, 0);
00300     if(ip == 0){
00301       end_op();
00302       return -1;
```

```
00303     }
00304   } else {
00305     if((ip = namei(path)) == 0){
00306       end_op();
00307       return -1;
00308     }
00309     ilock(ip);
00310     if(ip->type == T_DIR && omode != O_RDONLY){
00311       iunlockput(ip);
00312       end_op();
00313       return -1;
00314     }
00315   }
00316
00317   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
00318     if(f)
00319       fileclose(f);
00320     iunlockput(ip);
00321     end_op();
00322     return -1;
00323   }
00324   iunlock(ip);
00325   end_op();
00326
00327   f->type = FD_INODE;
00328   f->ip = ip;
00329   f->off = 0;
00330   f->readable = !(omode & O_WRONLY);
00331   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
00332   return fd;
00333 }
```

5.202.1.23 sys_pipe()

```
int sys_pipe (
              void  )
```

Definition at line 423 of file sysfile.c.

```
00424 {
00425   int *fd;
00426   struct file *rf, *wf;
00427   int fd0, fd1;
00428
00429   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
00430     return -1;
00431   if(pipealloc(&rf, &wf) < 0)
00432     return -1;
00433   fd0 = -1;
00434   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
00435     if(fd0 >= 0)
00436       myproc()->ofile[fd0] = 0;
00437     fileclose(rf);
00438     fileclose(wf);
00439     return -1;
00440   }
00441   fd[0] = fd0;
00442   fd[1] = fd1;
00443   return 0;
00444 }
```

5.202.1.24 sys_read()

```
int sys_read (
              void  )
```

Definition at line 70 of file sysfile.c.

```
00071 {
00072   struct file *f;
00073   int n;
```

```
00074   char *p;
00075
00076   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
00077     return -1;
00078   return fileread(f, p, n);
00079 }
```

5.202.1.25 sys_sbrk()

```
int sys_sbrk (
              void  )
```

Definition at line 54 of file sysproc.c.

```
00055 {
00056   int addr;
00057   int n;
00058
00059   if(argint(0, &n) < 0)
00060     return -1;
00061   addr = myproc()->sz;
00062   if(growproc(n) < 0)
00063     return -1;
00064   return addr;
00065 }
```

5.202.1.26 sys_sleep()

```
int sys_sleep (
              void  )
```

Definition at line 68 of file sysproc.c.

```
00069 {
00070   int n;
00071   uint ticks0;
00072
00073   if(argint(0, &n) < 0)
00074     return -1;
00075   acquire(&tickslock);
00076   ticks0 = ticks;
00077   while(ticks - ticks0 < n){
00078     if(myproc()->killed){
00079       release(&tickslock);
00080       return -1;
00081     }
00082     sleep(&ticks, &tickslock);
00083   }
00084   release(&tickslock);
00085   return 0;
00086 }
```

5.202.1.27 sys_unlink()

```
int sys_unlink (
            void  )
```

Definition at line 185 of file sysfile.c.

```
00186 {
00187   struct inode *ip, *dp;
00188   struct dirent de;
00189   char name[DIRSIZ], *path;
00190   uint off;
00191
00192   if(argstr(0, &path) < 0)
00193     return -1;
00194
00195   begin_op();
00196   if((dp = nameiparent(path, name)) == 0){
00197     end_op();
00198     return -1;
00199   }
00200
00201   ilock(dp);
00202
00203   // Cannot unlink "." or "..".
00204   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
00205     goto bad;
00206
00207   if((ip = dirlookup(dp, name, &off)) == 0)
00208     goto bad;
00209   ilock(ip);
00210
00211   if(ip->nlink < 1)
00212     panic("unlink: nlink < 1");
00213   if(ip->type == T_DIR && !isdirempty(ip)){
00214     iunlockput(ip);
00215     goto bad;
00216   }
00217
00218   memset(&de, 0, sizeof(de));
00219   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00220     panic("unlink: writei");
00221   if(ip->type == T_DIR){
00222     dp->nlink--;
00223     iupdate(dp);
00224   }
00225   iunlockput(dp);
00226
00227   ip->nlink--;
00228   iupdate(ip);
00229   iunlockput(ip);
00230
00231   end_op();
00232
00233   return 0;
00234
00235 bad:
00236   iunlockput(dp);
00237   end_op();
00238   return -1;
00239 }
```

5.202.1.28 sys_uptime()

```
int sys_uptime (
            void  )
```

Definition at line 91 of file sysproc.c.

```
00092 {
00093   uint xticks;
00094
00095   acquire(&tickslock);
00096   xticks = ticks;
00097   release(&tickslock);
00098   return xticks;
00099 }
```

5.202.1.29 sys_wait()

```
int sys_wait (
              void  )
```

Definition at line 32 of file sysproc.c.

```
00033 {
00034   return wait();
00035 }
```

5.202.1.30 sys_write()

```
int sys_write (
              void  )
```

Definition at line 82 of file sysfile.c.

```
00083 {
00084   struct file *f;
00085   int n;
00086   char *p;
00087
00088   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
00089     return -1;
00090   return filewrite(f, p, n);
00091 }
```

5.202.1.31 syscall()

```
void syscall (
              void  )
```

Definition at line 140 of file syscall.c.

```
00141 {
00142   int num;
00143   struct proc *curproc = myproc();
00144
00145   num = curproc->tf->eax;
00146   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
00147     curproc->tf->eax = syscalls[num]();
00148   } else {
00149     cprintf("%d %s: unknown sys call %d\n",
00150             curproc->pid, curproc->name, num);
00151     curproc->tf->eax = -1;
00152   }
00153 }
```

Referenced by trap().

5.202.2 Variable Documentation

5.202.2.1 syscalls

```
int(* syscalls[])(void) (
              void  )  [static]
```

Initial value:

```
= {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_halt]    sys_halt,
[SYS_cps]     sys_cps,
[SYS_chpr]    sys_chpr,
[SYS_getprocs] sys_getprocs,
}
```

Definition at line 111 of file syscall.c.

Referenced by syscall().

5.203 syscall.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "defs.h"
00003 #include "param.h"
00004 #include "memlayout.h"
00005 #include "mmu.h"
00006 #include "proc.h"
00007 #include "x86.h"
00008 #include "syscall.h"
00009
00010 // User code makes a system call with INT T_SYSCALL.
00011 // System call number in %eax.
00012 // Arguments on the stack, from the user call to the C
00013 // library system call function. The saved user %esp points
00014 // to a saved program counter, and then the first argument.
00015
00016 // Fetch the int at addr from the current process.
00017 int
00018 fetchint(uint addr, int *ip)
00019 {
00020   struct proc *curproc = myproc();
00021
00022   if(addr >= curproc->sz || addr+4 > curproc->sz)
00023     return -1;
00024   *ip = *(int*)(addr);
00025   return 0;
00026 }
00027
00028 // Fetch the nul-terminated string at addr from the current process.
00029 // Doesn't actually copy the string - just sets *pp to point at it.
00030 // Returns length of string, not including nul.
00031 int
00032 fetchstr(uint addr, char **pp)
00033 {
00034   char *s, *ep;
00035   struct proc *curproc = myproc();
```

```
00036
00037   if(addr >= curproc->sz)
00038     return -1;
00039   *pp = (char*)addr;
00040   ep = (char*)curproc->sz;
00041   for(s = *pp; s < ep; s++){
00042     if(*s == 0)
00043       return s - *pp;
00044   }
00045   return -1;
00046 }
00047
00048 // Fetch the nth 32-bit system call argument.
00049 int
00050 argint(int n, int *ip)
00051 {
00052   return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
00053 }
00054
00055 // Fetch the nth word-sized system call argument as a pointer
00056 // to a block of memory of size bytes.  Check that the pointer
00057 // lies within the process address space.
00058 int
00059 argptr(int n, char **pp, int size)
00060 {
00061   int i;
00062   struct proc *curproc = myproc();
00063
00064   if(argint(n, &i) < 0)
00065     return -1;
00066   if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
00067     return -1;
00068   *pp = (char*)i;
00069   return 0;
00070 }
00071
00072 // Fetch the nth word-sized system call argument as a string pointer.
00073 // Check that the pointer is valid and the string is nul-terminated.
00074 // (There is no shared writable memory, so the string can't change
00075 // between this check and being used by the kernel.)
00076 int
00077 argstr(int n, char **pp)
00078 {
00079   int addr;
00080   if(argint(n, &addr) < 0)
00081     return -1;
00082   return fetchstr(addr, pp);
00083 }
00084
00085 extern int sys_chdir(void);
00086 extern int sys_close(void);
00087 extern int sys_dup(void);
00088 extern int sys_exec(void);
00089 extern int sys_exit(void);
00090 extern int sys_fork(void);
00091 extern int sys_fstat(void);
00092 extern int sys_getpid(void);
00093 extern int sys_kill(void);
00094 extern int sys_link(void);
00095 extern int sys_mkdir(void);
00096 extern int sys_mknod(void);
00097 extern int sys_open(void);
00098 extern int sys_pipe(void);
00099 extern int sys_read(void);
00100 extern int sys_sbrk(void);
00101 extern int sys_sleep(void);
00102 extern int sys_unlink(void);
00103 extern int sys_wait(void);
00104 extern int sys_write(void);
00105 extern int sys_uptime(void);
00106 extern int sys_halt(void);
00107 extern int sys_cps(void);
00108 extern int sys_chpr(void);
00109 extern int sys_getprocs(void);
00110
00111 static int (*syscalls[])(void) = {
00112 [SYS_fork]    sys_fork,
00113 [SYS_exit]    sys_exit,
00114 [SYS_wait]    sys_wait,
00115 [SYS_pipe]    sys_pipe,
00116 [SYS_read]    sys_read,
00117 [SYS_kill]    sys_kill,
00118 [SYS_exec]    sys_exec,
00119 [SYS_fstat]   sys_fstat,
00120 [SYS_chdir]   sys_chdir,
00121 [SYS_dup]     sys_dup,
00122 [SYS_getpid]  sys_getpid,
```

```
00123 [SYS_sbrk]    sys_sbrk,
00124 [SYS_sleep]   sys_sleep,
00125 [SYS_uptime]  sys_uptime,
00126 [SYS_open]    sys_open,
00127 [SYS_write]   sys_write,
00128 [SYS_mknod]   sys_mknod,
00129 [SYS_unlink]  sys_unlink,
00130 [SYS_link]    sys_link,
00131 [SYS_mkdir]   sys_mkdir,
00132 [SYS_close]   sys_close,
00133 [SYS_halt]    sys_halt,
00134 [SYS_cps]     sys_cps,
00135 [SYS_chpr]    sys_chpr,
00136 [SYS_getprocs] sys_getprocs,
00137 };
00138
00139 void
00140 syscall(void)
00141 {
00142   int num;
00143   struct proc *curproc = myproc();
00144
00145   num = curproc->tf->eax;
00146   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
00147     curproc->tf->eax = syscalls[num]();
00148   } else {
00149     cprintf("%d %s: unknown sys call %d\n",
00150             curproc->pid, curproc->name, num);
00151     curproc->tf->eax = -1;
00152   }
00153 }
```

5.204 syscall.d File Reference

5.205 syscall.d

Go to the documentation of this file.

```
00001 syscall.o: syscall.c /usr/include/stdc-predef.h types.h defs.h param.h \
00002  memlayout.h mmu.h proc.h x86.h syscall.h
```

5.206 syscall.h File Reference

Macros

- #define SYS_chdir 9
- #define SYS_chpr 24
- #define SYS_close 21
- #define SYS_cps 23
- #define SYS_dup 10
- #define SYS_exec 7
- #define SYS_exit 2
- #define SYS_fork 1
- #define SYS_fstat 8
- #define SYS_getpid 11
- #define SYS_getprocs 25
- #define SYS_halt 22
- #define SYS_kill 6
- #define SYS_link 19
- #define SYS_mkdir 20
- #define SYS_mknod 17
- #define SYS_open 15

- #define SYS_pipe 4
- #define SYS_read 5
- #define SYS_sbrk 12
- #define SYS_sleep 13
- #define SYS_unlink 18
- #define SYS_uptime 14
- #define SYS_wait 3
- #define SYS_write 16

5.206.1 Macro Definition Documentation

5.206.1.1 SYS_chdir

```
#define SYS_chdir 9
```

Definition at line 10 of file syscall.h.

5.206.1.2 SYS_chpr

```
#define SYS_chpr 24
```

Definition at line 25 of file syscall.h.

5.206.1.3 SYS_close

```
#define SYS_close 21
```

Definition at line 22 of file syscall.h.

5.206.1.4 SYS_cps

```
#define SYS_cps 23
```

Definition at line 24 of file syscall.h.

5.206.1.5 SYS_dup

```
#define SYS_dup 10
```

Definition at line 11 of file syscall.h.

5.206.1.6 SYS_exec

```
#define SYS_exec 7
```

Definition at line 8 of file syscall.h.

5.206.1.7 SYS_exit

```
#define SYS_exit 2
```

Definition at line 3 of file syscall.h.

5.206.1.8 SYS_fork

```
#define SYS_fork 1
```

Definition at line 2 of file syscall.h.

5.206.1.9 SYS_fstat

```
#define SYS_fstat 8
```

Definition at line 9 of file syscall.h.

5.206.1.10 SYS_getpid

```
#define SYS_getpid 11
```

Definition at line 12 of file syscall.h.

5.206.1.11 SYS_getprocs

```
#define SYS_getprocs 25
```

Definition at line 26 of file syscall.h.

5.206.1.12 SYS_halt

```
#define SYS_halt 22
```

Definition at line 23 of file syscall.h.

5.206.1.13 SYS_kill

```
#define SYS_kill 6
```

Definition at line 7 of file syscall.h.

5.206.1.14 SYS_link

```
#define SYS_link 19
```

Definition at line 20 of file syscall.h.

5.206.1.15 SYS_mkdir

```
#define SYS_mkdir 20
```

Definition at line 21 of file syscall.h.

5.206.1.16 SYS_mknod

```
#define SYS_mknod 17
```

Definition at line 18 of file syscall.h.

5.206.1.17 SYS_open

```
#define SYS_open 15
```

Definition at line 16 of file syscall.h.

5.206.1.18 SYS_pipe

```
#define SYS_pipe 4
```

Definition at line 5 of file syscall.h.

5.206.1.19 SYS_read

```
#define SYS_read 5
```

Definition at line 6 of file syscall.h.

5.206.1.20 SYS_sbrk

```
#define SYS_sbrk 12
```

Definition at line 13 of file syscall.h.

5.206.1.21 SYS_sleep

```
#define SYS_sleep 13
```

Definition at line 14 of file syscall.h.

5.206.1.22 SYS_unlink

```
#define SYS_unlink 18
```

Definition at line 19 of file syscall.h.

5.206.1.23 SYS_uptime

```
#define SYS_uptime 14
```

Definition at line 15 of file syscall.h.

5.206.1.24 SYS_wait

```
#define SYS_wait 3
```

Definition at line 4 of file syscall.h.

5.206.1.25 SYS_write

```
#define SYS_write 16
```

Definition at line 17 of file syscall.h.

5.207 syscall.h

Go to the documentation of this file.

```
00001 // System call numbers
00002 #define SYS_fork    1
00003 #define SYS_exit    2
00004 #define SYS_wait    3
00005 #define SYS_pipe    4
00006 #define SYS_read    5
00007 #define SYS_kill    6
00008 #define SYS_exec    7
00009 #define SYS_fstat   8
00010 #define SYS_chdir   9
00011 #define SYS_dup    10
00012 #define SYS_getpid 11
00013 #define SYS_sbrk   12
00014 #define SYS_sleep  13
00015 #define SYS_uptime 14
00016 #define SYS_open   15
00017 #define SYS_write  16
00018 #define SYS_mknod  17
00019 #define SYS_unlink 18
00020 #define SYS_link   19
00021 #define SYS_mkdir  20
00022 #define SYS_close  21
00023 #define SYS_halt   22
00024 #define SYS_cps    23
00025 #define SYS_chpr   24
00026 #define SYS_getprocs 25
00027
```

5.208 sysfile.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "fs.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "file.h"
#include "fcntl.h"
```

Functions

- static int argfd (int n, int *pfd, struct file **pf)
- static struct inode * create (char *path, short type, short major, short minor)
- static int fdalloc (struct file *f)
- static int isdirempty (struct inode *dp)
- int sys_chdir (void)
- int sys_close (void)
- int sys_dup (void)
- int sys_exec (void)
- int sys_fstat (void)
- int sys_link (void)
- int sys_mkdir (void)
- int sys_mknod (void)
- int sys_open (void)
- int sys_pipe (void)
- int sys_read (void)
- int sys_unlink (void)
- int sys_write (void)

5.208.1 Function Documentation

5.208.1.1 argfd()

```
static int argfd (
            int n,
            int * pfd,
            struct file ** pf )  [static]
```

Definition at line 22 of file sysfile.c.

```
00023 {
00024   int fd;
00025   struct file *f;
00026
00027   if(argint(n, &fd) < 0)
00028     return -1;
00029   if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
```

```
00030     return -1;
00031   if(pfd)
00032     *pfd = fd;
00033   if(pf)
00034     *pf = f;
00035   return 0;
00036 }
```

Referenced by sys_close(), sys_dup(), sys_fstat(), sys_read(), and sys_write().

5.208.1.2 create()

```
static struct inode * create (
              char * path,
              short type,
              short major,
              short minor )  [static]
```

Definition at line 242 of file sysfile.c.

```
00243 {
00244   struct inode *ip, *dp;
00245   char name[DIRSIZ];
00246
00247   if((dp = nameiparent(path, name)) == 0)
00248     return 0;
00249   ilock(dp);
00250
00251   if((ip = dirlookup(dp, name, 0)) != 0){
00252     iunlockput(dp);
00253     ilock(ip);
00254     if(type == T_FILE && ip->type == T_FILE)
00255       return ip;
00256     iunlockput(ip);
00257     return 0;
00258   }
00259
00260   if((ip = ialloc(dp->dev, type)) == 0)
00261     panic("create: ialloc");
00262
00263   ilock(ip);
00264   ip->major = major;
00265   ip->minor = minor;
00266   ip->nlink = 1;
00267   iupdate(ip);
00268
00269   if(type == T_DIR){  // Create . and .. entries.
00270     dp->nlink++;  // for ".."
00271     iupdate(dp);
00272     // No ip->nlink++ for ".": avoid cyclic ref count.
00273     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
00274       panic("create dots");
00275   }
00276
00277   if(dirlink(dp, name, ip->inum) < 0)
00278     panic("create: dirlink");
00279
00280   iunlockput(dp);
00281
00282   return ip;
00283 }
```

Referenced by sys_mkdir(), sys_mknod(), and sys_open().

5.208.1.3 fdalloc()

```
static int fdalloc (
            struct file * f )  [static]
```

Definition at line 41 of file sysfile.c.

```
00042 {
00043   int fd;
00044   struct proc *curproc = myproc();
00045
00046   for(fd = 0; fd < NOFILE; fd++){
00047     if(curproc->ofile[fd] == 0){
00048       curproc->ofile[fd] = f;
00049       return fd;
00050     }
00051   }
00052   return -1;
00053 }
```

Referenced by sys_dup(), sys_open(), and sys_pipe().

5.208.1.4 isdirempty()

```
static int isdirempty (
            struct inode * dp )  [static]
```

Definition at line 169 of file sysfile.c.

```
00170 {
00171   int off;
00172   struct dirent de;
00173
00174   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
00175     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00176       panic("isdirempty: readi");
00177     if(de.inum != 0)
00178       return 0;
00179   }
00180   return 1;
00181 }
```

Referenced by sys_unlink().

5.208.1.5 sys_chdir()

```
int sys_chdir (
            void  )
```

Definition at line 372 of file sysfile.c.

```
00373 {
00374   char *path;
00375   struct inode *ip;
00376   struct proc *curproc = myproc();
00377
00378   begin_op();
00379   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
00380     end_op();
00381     return -1;
00382   }
00383   ilock(ip);
00384   if(ip->type != T_DIR){
00385     iunlockput(ip);
00386     end_op();
00387     return -1;
00388   }
00389   iunlock(ip);
00390   iput(curproc->cwd);
00391   end_op();
00392   curproc->cwd = ip;
00393   return 0;
00394 }
```

5.208.1.6 sys_close()

```
int sys_close (
              void  )
```

Definition at line 94 of file sysfile.c.

```
00095 {
00096   int fd;
00097   struct file *f;
00098
00099   if(argfd(0, &fd, &f) < 0)
00100     return -1;
00101   myproc()->ofile[fd] = 0;
00102   fileclose(f);
00103   return 0;
00104 }
```

5.208.1.7 sys_dup()

```
int sys_dup (
              void  )
```

Definition at line 56 of file sysfile.c.

```
00057 {
00058   struct file *f;
00059   int fd;
00060
00061   if(argfd(0, 0, &f) < 0)
00062     return -1;
00063   if((fd=fdalloc(f)) < 0)
00064     return -1;
00065   filedup(f);
00066   return fd;
00067 }
```

5.208.1.8 sys_exec()

```
int sys_exec (
              void  )
```

Definition at line 397 of file sysfile.c.

```
00398 {
00399   char *path, *argv[MAXARG];
00400   int i;
00401   uint uargv, uarg;
00402
00403   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
00404     return -1;
00405   }
00406   memset(argv, 0, sizeof(argv));
00407   for(i=0;; i++){
00408     if(i >= NELEM(argv))
00409       return -1;
00410     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
00411       return -1;
00412     if(uarg == 0){
00413       argv[i] = 0;
00414       break;
00415     }
00416     if(fetchstr(uarg, &argv[i]) < 0)
00417       return -1;
00418   }
00419   return exec(path, argv);
00420 }
```

5.208.1.9 sys_fstat()

```
int sys_fstat (
              void  )
```

Definition at line 107 of file sysfile.c.

```
00108 {
00109   struct file *f;
00110   struct stat *st;
00111
00112   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
00113     return -1;
00114   return filestat(f, st);
00115 }
```

5.208.1.10 sys_link()

```
int sys_link (
              void  )
```

Definition at line 119 of file sysfile.c.

```
00120 {
00121   char name[DIRSIZ], *new, *old;
00122   struct inode *dp, *ip;
00123
00124   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
00125     return -1;
00126
00127   begin_op();
00128   if((ip = namei(old)) == 0){
00129     end_op();
00130     return -1;
00131   }
00132
00133   ilock(ip);
00134   if(ip->type == T_DIR){
00135     iunlockput(ip);
00136     end_op();
00137     return -1;
00138   }
00139
00140   ip->nlink++;
00141   iupdate(ip);
00142   iunlock(ip);
00143
00144   if((dp = nameiparent(new, name)) == 0)
00145     goto bad;
00146   ilock(dp);
00147   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
00148     iunlockput(dp);
00149     goto bad;
00150   }
00151   iunlockput(dp);
00152   iput(ip);
00153
00154   end_op();
00155
00156   return 0;
00157
00158 bad:
00159   ilock(ip);
00160   ip->nlink--;
00161   iupdate(ip);
00162   iunlockput(ip);
00163   end_op();
00164   return -1;
00165 }
```

5.208.1.11 sys_mkdir()

```
int sys_mkdir (
              void  )
```

Definition at line 336 of file sysfile.c.

```
00337 {
00338   char *path;
00339   struct inode *ip;
00340
00341   begin_op();
00342   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
00343     end_op();
00344     return -1;
00345   }
00346   iunlockput(ip);
00347   end_op();
00348   return 0;
00349 }
```

5.208.1.12 sys_mknod()

```
int sys_mknod (
              void  )
```

Definition at line 352 of file sysfile.c.

```
00353 {
00354   struct inode *ip;
00355   char *path;
00356   int major, minor;
00357
00358   begin_op();
00359   if((argstr(0, &path)) < 0 ||
00360      argint(1, &major) < 0 ||
00361      argint(2, &minor) < 0 ||
00362      (ip = create(path, T_DEV, major, minor)) == 0){
00363     end_op();
00364     return -1;
00365   }
00366   iunlockput(ip);
00367   end_op();
00368   return 0;
00369 }
```

5.208.1.13 sys_open()

```
int sys_open (
              void  )
```

Definition at line 286 of file sysfile.c.

```
00287 {
00288   char *path;
00289   int fd, omode;
00290   struct file *f;
00291   struct inode *ip;
00292
00293   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
00294     return -1;
00295
00296   begin_op();
00297
00298   if(omode & O_CREATE){
00299     ip = create(path, T_FILE, 0, 0);
00300     if(ip == 0){
00301       end_op();
00302       return -1;
```

```
00303     }
00304   } else {
00305     if((ip = namei(path)) == 0){
00306       end_op();
00307       return -1;
00308     }
00309     ilock(ip);
00310     if(ip->type == T_DIR && omode != O_RDONLY){
00311       iunlockput(ip);
00312       end_op();
00313       return -1;
00314     }
00315   }
00316
00317   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
00318     if(f)
00319       fileclose(f);
00320     iunlockput(ip);
00321     end_op();
00322     return -1;
00323   }
00324   iunlock(ip);
00325   end_op();
00326
00327   f->type = FD_INODE;
00328   f->ip = ip;
00329   f->off = 0;
00330   f->readable = !(omode & O_WRONLY);
00331   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
00332   return fd;
00333 }
```

5.208.1.14 sys_pipe()

```
int sys_pipe (
              void  )
```

Definition at line 423 of file sysfile.c.

```
00424 {
00425   int *fd;
00426   struct file *rf, *wf;
00427   int fd0, fd1;
00428
00429   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
00430     return -1;
00431   if(pipealloc(&rf, &wf) < 0)
00432     return -1;
00433   fd0 = -1;
00434   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
00435     if(fd0 >= 0)
00436       myproc()->ofile[fd0] = 0;
00437     fileclose(rf);
00438     fileclose(wf);
00439     return -1;
00440   }
00441   fd[0] = fd0;
00442   fd[1] = fd1;
00443   return 0;
00444 }
```

5.208.1.15 sys_read()

```
int sys_read (
              void  )
```

Definition at line 70 of file sysfile.c.

```
00071 {
00072   struct file *f;
00073   int n;
```

```
00074   char *p;
00075
00076   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
00077     return -1;
00078   return fileread(f, p, n);
00079 }
```

5.208.1.16 sys_unlink()

```
int sys_unlink (
             void  )
```

Definition at line 185 of file sysfile.c.

```
00186 {
00187   struct inode *ip, *dp;
00188   struct dirent de;
00189   char name[DIRSIZ], *path;
00190   uint off;
00191
00192   if(argstr(0, &path) < 0)
00193     return -1;
00194
00195   begin_op();
00196   if((dp = nameiparent(path, name)) == 0){
00197     end_op();
00198     return -1;
00199   }
00200
00201   ilock(dp);
00202
00203   // Cannot unlink "." or "..".
00204   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
00205     goto bad;
00206
00207   if((ip = dirlookup(dp, name, &off)) == 0)
00208     goto bad;
00209   ilock(ip);
00210
00211   if(ip->nlink < 1)
00212     panic("unlink: nlink < 1");
00213   if(ip->type == T_DIR && !isdirempty(ip)){
00214     iunlockput(ip);
00215     goto bad;
00216   }
00217
00218   memset(&de, 0, sizeof(de));
00219   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00220     panic("unlink: writei");
00221   if(ip->type == T_DIR){
00222     dp->nlink--;
00223     iupdate(dp);
00224   }
00225   iunlockput(dp);
00226
00227   ip->nlink--;
00228   iupdate(ip);
00229   iunlockput(ip);
00230
00231   end_op();
00232
00233   return 0;
00234
00235 bad:
00236   iunlockput(dp);
00237   end_op();
00238   return -1;
00239 }
```

5.208.1.17 sys_write()

```
int sys_write (
             void  )
```

Definition at line 82 of file sysfile.c.

```
00083 {
00084   struct file *f;
00085   int n;
00086   char *p;
00087
00088   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
00089     return -1;
00090   return filewrite(f, p, n);
00091 }
```

5.209 sysfile.c

Go to the documentation of this file.

```
00001 //
00002 // File-system system calls.
00003 // Mostly argument checking, since we don't trust
00004 // user code, and calls into file.c and fs.c.
00005 //
00006
00007 #include "types.h"
00008 #include "defs.h"
00009 #include "param.h"
00010 #include "stat.h"
00011 #include "mmu.h"
00012 #include "proc.h"
00013 #include "fs.h"
00014 #include "spinlock.h"
00015 #include "sleeplock.h"
00016 #include "file.h"
00017 #include "fcntl.h"
00018
00019 // Fetch the nth word-sized system call argument as a file descriptor
00020 // and return both the descriptor and the corresponding struct file.
00021 static int
00022 argfd(int n, int *pfd, struct file **pf)
00023 {
00024   int fd;
00025   struct file *f;
00026
00027   if(argint(n, &fd) < 0)
00028     return -1;
00029   if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
00030     return -1;
00031   if(pfd)
00032     *pfd = fd;
00033   if(pf)
00034     *pf = f;
00035   return 0;
00036 }
00037
00038 // Allocate a file descriptor for the given file.
00039 // Takes over file reference from caller on success.
00040 static int
00041 fdalloc(struct file *f)
00042 {
00043   int fd;
00044   struct proc *curproc = myproc();
00045
00046   for(fd = 0; fd < NOFILE; fd++){
00047     if(curproc->ofile[fd] == 0){
00048       curproc->ofile[fd] = f;
00049       return fd;
00050     }
00051   }
00052   return -1;
00053 }
00054
00055 int
00056 sys_dup(void)
00057 {
00058   struct file *f;
00059   int fd;
```

```
00060
00061   if(argfd(0, 0, &f) < 0)
00062     return -1;
00063   if((fd=fdalloc(f)) < 0)
00064     return -1;
00065   filedup(f);
00066   return fd;
00067 }
00068
00069 int
00070 sys_read(void)
00071 {
00072   struct file *f;
00073   int n;
00074   char *p;
00075
00076   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
00077     return -1;
00078   return fileread(f, p, n);
00079 }
00080
00081 int
00082 sys_write(void)
00083 {
00084   struct file *f;
00085   int n;
00086   char *p;
00087
00088   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
00089     return -1;
00090   return filewrite(f, p, n);
00091 }
00092
00093 int
00094 sys_close(void)
00095 {
00096   int fd;
00097   struct file *f;
00098
00099   if(argfd(0, &fd, &f) < 0)
00100     return -1;
00101   myproc()->ofile[fd] = 0;
00102   fileclose(f);
00103   return 0;
00104 }
00105
00106 int
00107 sys_fstat(void)
00108 {
00109   struct file *f;
00110   struct stat *st;
00111
00112   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
00113     return -1;
00114   return filestat(f, st);
00115 }
00116
00117 // Create the path new as a link to the same inode as old.
00118 int
00119 sys_link(void)
00120 {
00121   char name[DIRSIZ], *new, *old;
00122   struct inode *dp, *ip;
00123
00124   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
00125     return -1;
00126
00127   begin_op();
00128   if((ip = namei(old)) == 0){
00129     end_op();
00130     return -1;
00131   }
00132
00133   ilock(ip);
00134   if(ip->type == T_DIR){
00135     iunlockput(ip);
00136     end_op();
00137     return -1;
00138   }
00139
00140   ip->nlink++;
00141   iupdate(ip);
00142   iunlock(ip);
00143
00144   if((dp = nameiparent(new, name)) == 0)
00145     goto bad;
00146   ilock(dp);
```

```
00147   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
00148     iunlockput(dp);
00149     goto bad;
00150   }
00151   iunlockput(dp);
00152   iput(ip);
00153
00154   end_op();
00155
00156   return 0;
00157
00158 bad:
00159   ilock(ip);
00160   ip->nlink--;
00161   iupdate(ip);
00162   iunlockput(ip);
00163   end_op();
00164   return -1;
00165 }
00166
00167 // Is the directory dp empty except for "." and ".." ?
00168 static int
00169 isdirempty(struct inode *dp)
00170 {
00171   int off;
00172   struct dirent de;
00173
00174   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
00175     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00176       panic("isdirempty: readi");
00177     if(de.inum != 0)
00178       return 0;
00179   }
00180   return 1;
00181 }
00182
00183 //PAGEBREAK!
00184 int
00185 sys_unlink(void)
00186 {
00187   struct inode *ip, *dp;
00188   struct dirent de;
00189   char name[DIRSIZ], *path;
00190   uint off;
00191
00192   if(argstr(0, &path) < 0)
00193     return -1;
00194
00195   begin_op();
00196   if((dp = nameiparent(path, name)) == 0){
00197     end_op();
00198     return -1;
00199   }
00200
00201   ilock(dp);
00202
00203   // Cannot unlink "." or "..".
00204   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
00205     goto bad;
00206
00207   if((ip = dirlookup(dp, name, &off)) == 0)
00208     goto bad;
00209   ilock(ip);
00210
00211   if(ip->nlink < 1)
00212     panic("unlink: nlink < 1");
00213   if(ip->type == T_DIR && !isdirempty(ip)){
00214     iunlockput(ip);
00215     goto bad;
00216   }
00217
00218   memset(&de, 0, sizeof(de));
00219   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
00220     panic("unlink: writei");
00221   if(ip->type == T_DIR){
00222     dp->nlink--;
00223     iupdate(dp);
00224   }
00225   iunlockput(dp);
00226
00227   ip->nlink--;
00228   iupdate(ip);
00229   iunlockput(ip);
00230
00231   end_op();
00232
00233   return 0;
```

```
00234
00235 bad:
00236   iunlockput(dp);
00237   end_op();
00238   return -1;
00239 }
00240
00241 static struct inode*
00242 create(char *path, short type, short major, short minor)
00243 {
00244   struct inode *ip, *dp;
00245   char name[DIRSIZ];
00246
00247   if((dp = nameiparent(path, name)) == 0)
00248     return 0;
00249   ilock(dp);
00250
00251   if((ip = dirlookup(dp, name, 0)) != 0){
00252     iunlockput(dp);
00253     ilock(ip);
00254     if(type == T_FILE && ip->type == T_FILE)
00255       return ip;
00256     iunlockput(ip);
00257     return 0;
00258   }
00259
00260   if((ip = ialloc(dp->dev, type)) == 0)
00261     panic("create: ialloc");
00262
00263   ilock(ip);
00264   ip->major = major;
00265   ip->minor = minor;
00266   ip->nlink = 1;
00267   iupdate(ip);
00268
00269   if(type == T_DIR){  // Create . and .. entries.
00270     dp->nlink++;  // for ".."
00271     iupdate(dp);
00272     // No ip->nlink++ for ".": avoid cyclic ref count.
00273     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
00274       panic("create dots");
00275   }
00276
00277   if(dirlink(dp, name, ip->inum) < 0)
00278     panic("create: dirlink");
00279
00280   iunlockput(dp);
00281
00282   return ip;
00283 }
00284
00285 int
00286 sys_open(void)
00287 {
00288   char *path;
00289   int fd, omode;
00290   struct file *f;
00291   struct inode *ip;
00292
00293   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
00294     return -1;
00295
00296   begin_op();
00297
00298   if(omode & O_CREATE){
00299     ip = create(path, T_FILE, 0, 0);
00300     if(ip == 0){
00301       end_op();
00302       return -1;
00303     }
00304   } else {
00305     if((ip = namei(path)) == 0){
00306       end_op();
00307       return -1;
00308     }
00309     ilock(ip);
00310     if(ip->type == T_DIR && omode != O_RDONLY){
00311       iunlockput(ip);
00312       end_op();
00313       return -1;
00314     }
00315   }
00316
00317   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
00318     if(f)
00319       fileclose(f);
00320     iunlockput(ip);
```

```
00321     end_op();
00322     return -1;
00323   }
00324   iunlock(ip);
00325   end_op();
00326
00327   f->type = FD_INODE;
00328   f->ip = ip;
00329   f->off = 0;
00330   f->readable = !(omode & O_WRONLY);
00331   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
00332   return fd;
00333 }
00334
00335 int
00336 sys_mkdir(void)
00337 {
00338   char *path;
00339   struct inode *ip;
00340
00341   begin_op();
00342   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
00343     end_op();
00344     return -1;
00345   }
00346   iunlockput(ip);
00347   end_op();
00348   return 0;
00349 }
00350
00351 int
00352 sys_mknod(void)
00353 {
00354   struct inode *ip;
00355   char *path;
00356   int major, minor;
00357
00358   begin_op();
00359   if((argstr(0, &path)) < 0 ||
00360      argint(1, &major) < 0 ||
00361      argint(2, &minor) < 0 ||
00362      (ip = create(path, T_DEV, major, minor)) == 0){
00363     end_op();
00364     return -1;
00365   }
00366   iunlockput(ip);
00367   end_op();
00368   return 0;
00369 }
00370
00371 int
00372 sys_chdir(void)
00373 {
00374   char *path;
00375   struct inode *ip;
00376   struct proc *curproc = myproc();
00377
00378   begin_op();
00379   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
00380     end_op();
00381     return -1;
00382   }
00383   ilock(ip);
00384   if(ip->type != T_DIR){
00385     iunlockput(ip);
00386     end_op();
00387     return -1;
00388   }
00389   iunlock(ip);
00390   iput(curproc->cwd);
00391   end_op();
00392   curproc->cwd = ip;
00393   return 0;
00394 }
00395
00396 int
00397 sys_exec(void)
00398 {
00399   char *path, *argv[MAXARG];
00400   int i;
00401   uint uargv, uarg;
00402
00403   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
00404     return -1;
00405   }
00406   memset(argv, 0, sizeof(argv));
00407   for(i=0;; i++){
```

```
00408     if(i >= NELEM(argv))
00409       return -1;
00410     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
00411       return -1;
00412     if(uarg == 0){
00413       argv[i] = 0;
00414       break;
00415     }
00416     if(fetchstr(uarg, &argv[i]) < 0)
00417       return -1;
00418   }
00419   return exec(path, argv);
00420 }
00421
00422 int
00423 sys_pipe(void)
00424 {
00425   int *fd;
00426   struct file *rf, *wf;
00427   int fd0, fd1;
00428
00429   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
00430     return -1;
00431   if(pipealloc(&rf, &wf) < 0)
00432     return -1;
00433   fd0 = -1;
00434   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
00435     if(fd0 >= 0)
00436       myproc()->ofile[fd0] = 0;
00437     fileclose(rf);
00438     fileclose(wf);
00439     return -1;
00440   }
00441   fd[0] = fd0;
00442   fd[1] = fd1;
00443   return 0;
00444 }
```

5.210 sysfile.d File Reference

5.211 sysfile.d

Go to the documentation of this file.

```
00001 sysfile.o: sysfile.c /usr/include/stdc-predef.h types.h defs.h param.h \
00002  stat.h mmu.h proc.h fs.h spinlock.h sleeplock.h file.h fcntl.h
```

5.212 sysproc.c File Reference

```
#include "types.h"
#include "x86.h"
#include "defs.h"
#include "date.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "uproc.h"
#include "spinlock.h"
```

Functions

- int sys_chpr (void)
- int sys_cps (void)
- int sys_exit (void)
- int sys_fork (void)
- int sys_getpid (void)
- int sys_getprocs (void)
- int sys_halt (void)
- int sys_kill (void)
- int sys_sbrk (void)
- int sys_sleep (void)
- int sys_uptime (void)
- int sys_wait (void)

Variables

- struct {
  struct spinlock lock
  struct proc proc [NPROC]
  } ptable

5.212.1 Function Documentation

5.212.1.1 sys_chpr()

```
int sys_chpr (
              void  )
```

Definition at line 125 of file sysproc.c.

```
00126 {
00127   int pid, pr;
00128   if(argint(0, &pid) < 0)
00129     return -1;
00130   if(argint(1, &pr) < 0)
00131     return -1;
00132
00133   return chpr(pid, pr);
00134 }
```

5.212.1.2 sys_cps()

```
int sys_cps (
              void  )
```

Definition at line 119 of file sysproc.c.

```
00120 {
00121   return cps();
00122 }
```

5.212.1.3 sys_exit()

```
int sys_exit (
              void  )
```

Definition at line 25 of file sysproc.c.

```
00026 {
00027   exit();
00028   return 0;  // not reached
00029 }
```

5.212.1.4 sys_fork()

```
int sys_fork (
              void  )
```

Definition at line 19 of file sysproc.c.

```
00020 {
00021   return fork();
00022 }
```

5.212.1.5 sys_getpid()

```
int sys_getpid (
              void  )
```

Definition at line 48 of file sysproc.c.

```
00049 {
00050   return myproc()->pid;
00051 }
```

5.212.1.6 sys_getprocs()

```
int sys_getprocs (
              void  )
```

Definition at line 137 of file sysproc.c.

```
00137                   {
00138   // declare local variables for max uproc size, struct uproc, and counter
00139   int max;
00140   struct uproc *p;
00141   int i=0;
00142
00143   // if argint has trouble, return error -1
00144   if(argint(0,&max) < 0)
00145     return -1;
00146
00147   // if argptr for allocating struct size has trouble, return -1
00148   if(argptr(1,(char**)&p, max*sizeof(struct uproc)) < 0)
00149     return -1;
00150
00151   // create pointer to ptable processes
00152   struct proc *ptr = ptable.proc;
00153
00154   // loop through ptable
00155   for(; ptr < &ptable.proc[NPROC]; ptr++)
00156   {
```

```
00157     if(!(ptr->state == UNUSED))
00158     {
00159       // if the process in ptable is not UNUSED, assign pid, parent pid, and name to uproc
00160       p[i].pid = ptr->pid;
00161       p[i].ppid = ptr->parent->pid;
00162       strncpy(p[i].name, ptr->name, 16);
00163       // add 1 to the process counter
00164       i++;
00165     }
00166   }
00167
00168   // return the number of processes that are not UNUSED
00169   return i;
00170 }
```

5.212.1.7 sys_halt()

```
int sys_halt (
              void  )
```

Definition at line 111 of file sysproc.c.

```
00112 {
00113   outb(0xf4, 0x00);
00114   return 0;
00115 }
```

5.212.1.8 sys_kill()

```
int sys_kill (
              void  )
```

Definition at line 38 of file sysproc.c.

```
00039 {
00040   int pid;
00041
00042   if(argint(0, &pid) < 0)
00043     return -1;
00044   return kill(pid);
00045 }
```

5.212.1.9 sys_sbrk()

```
int sys_sbrk (
              void  )
```

Definition at line 54 of file sysproc.c.

```
00055 {
00056   int addr;
00057   int n;
00058
00059   if(argint(0, &n) < 0)
00060     return -1;
00061   addr = myproc()->sz;
00062   if(growproc(n) < 0)
00063     return -1;
00064   return addr;
00065 }
```

5.212.1.10 sys_sleep()

```
int sys_sleep (
              void  )
```

Definition at line 68 of file sysproc.c.

```
00069 {
00070   int n;
00071   uint ticks0;
00072
00073   if(argint(0, &n) < 0)
00074     return -1;
00075   acquire(&tickslock);
00076   ticks0 = ticks;
00077   while(ticks - ticks0 < n){
00078     if(myproc()->killed){
00079       release(&tickslock);
00080       return -1;
00081     }
00082     sleep(&ticks, &tickslock);
00083   }
00084   release(&tickslock);
00085   return 0;
00086 }
```

5.212.1.11 sys_uptime()

```
int sys_uptime (
              void  )
```

Definition at line 91 of file sysproc.c.

```
00092 {
00093   uint xticks;
00094
00095   acquire(&tickslock);
00096   xticks = ticks;
00097   release(&tickslock);
00098   return xticks;
00099 }
```

5.212.1.12 sys_wait()

```
int sys_wait (
              void  )
```

Definition at line 32 of file sysproc.c.

```
00033 {
00034   return wait();
00035 }
```

5.212.2 Variable Documentation

5.212.2.1 lock

```
struct spinlock lock
```

Definition at line 13 of file sysproc.c.

5.212.2.2 proc

```
struct proc proc[NPROC]
```

Definition at line 14 of file sysproc.c.

5.212.2.3

```
struct { ... } ptable
```

Referenced by sys_getprocs().

5.213 sysproc.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "x86.h"
00003 #include "defs.h"
00004 #include "date.h"
00005 #include "param.h"
00006 #include "memlayout.h"
00007 #include "mmu.h"
00008 #include "proc.h"
00009 #include "uproc.h"
00010 #include "spinlock.h"
00011
00012 struct {
00013   struct spinlock lock;
00014   struct proc proc[NPROC];
00015 } ptable;
00016
00017
00018 int
00019 sys_fork(void)
00020 {
00021   return fork();
00022 }
00023
00024 int
00025 sys_exit(void)
00026 {
00027   exit();
00028   return 0;  // not reached
00029 }
00030
00031 int
00032 sys_wait(void)
00033 {
00034   return wait();
00035 }
00036
00037 int
00038 sys_kill(void)
00039 {
00040   int pid;
```

```
00041
00042   if(argint(0, &pid) < 0)
00043     return -1;
00044   return kill(pid);
00045 }
00046
00047 int
00048 sys_getpid(void)
00049 {
00050   return myproc()->pid;
00051 }
00052
00053 int
00054 sys_sbrk(void)
00055 {
00056   int addr;
00057   int n;
00058
00059   if(argint(0, &n) < 0)
00060     return -1;
00061   addr = myproc()->sz;
00062   if(growproc(n) < 0)
00063     return -1;
00064   return addr;
00065 }
00066
00067 int
00068 sys_sleep(void)
00069 {
00070   int n;
00071   uint ticks0;
00072
00073   if(argint(0, &n) < 0)
00074     return -1;
00075   acquire(&tickslock);
00076   ticks0 = ticks;
00077   while(ticks - ticks0 < n){
00078     if(myproc()->killed){
00079       release(&tickslock);
00080       return -1;
00081     }
00082     sleep(&ticks, &tickslock);
00083   }
00084   release(&tickslock);
00085   return 0;
00086 }
00087
00088 // return how many clock tick interrupts have occurred
00089 // since start.
00090 int
00091 sys_uptime(void)
00092 {
00093   uint xticks;
00094
00095   acquire(&tickslock);
00096   xticks = ticks;
00097   release(&tickslock);
00098   return xticks;
00099 }
00100
00101 /*int
00102 sys_halt(void)
00103 {
00104   char *p = "Shutdown";
00105   for( ; *p; p++)
00106     outw(0xB004, 0x2000);
00107   return 0;
00108 }*/
00109
00110 int
00111 sys_halt(void)
00112 {
00113   outb(0xf4, 0x00);
00114   return 0;
00115 }
00116
00117
00118 int
00119 sys_cps(void)
00120 {
00121   return cps();
00122 }
00123
00124 int
00125 sys_chpr(void)
00126 {
00127   int pid, pr;
```

```
00128   if(argint(0, &pid) < 0)
00129     return -1;
00130   if(argint(1, &pr) < 0)
00131     return -1;
00132
00133   return chpr(pid, pr);
00134 }
00135
00136 int
00137 sys_getprocs(void){
00138   // declare local variables for max uproc size, struct uproc, and counter
00139   int max;
00140   struct uproc *p;
00141   int i=0;
00142
00143   // if argint has trouble, return error -1
00144   if(argint(0,&max) < 0)
00145     return -1;
00146
00147   // if argptr for allocating struct size has trouble, return -1
00148   if(argptr(1,(char**)&p, max*sizeof(struct uproc)) < 0)
00149     return -1;
00150
00151   // create pointer to ptable processes
00152   struct proc *ptr = ptable.proc;
00153
00154   // loop through ptable
00155   for(; ptr < &ptable.proc[NPROC]; ptr++)
00156   {
00157     if(!(ptr->state == UNUSED))
00158     {
00159       // if the process in ptable is not UNUSED, assign pid, parent pid, and name to uproc
00160       p[i].pid = ptr->pid;
00161       p[i].ppid = ptr->parent->pid;
00162       strncpy(p[i].name, ptr->name, 16);
00163       // add 1 to the process counter
00164       i++;
00165     }
00166   }
00167
00168   // return the number of processes that are not UNUSED
00169   return i;
00170 }
00171
00172
00173
```

5.214 sysproc.d File Reference

5.215 sysproc.d

Go to the documentation of this file.

```
00001 sysproc.o: sysproc.c /usr/include/stdc-predef.h types.h x86.h defs.h \
00002  date.h param.h memlayout.h mmu.h proc.h uproc.h spinlock.h
```

5.216 trap.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"
```

Functions

- void idtinit (void)
- void trap (struct trapframe *tf)
- void tvinit (void)

Variables

- struct gatedesc idt [256]
- uint ticks
- struct spinlock tickslock
- uint vectors [ ]

5.216.1 Function Documentation

5.216.1.1 idtinit()

```
void idtinit (
              void  )
```

Definition at line 30 of file trap.c.

```
00031 {
00032   lidt(idt, sizeof(idt));
00033 }
```

5.216.1.2 trap()

```
void trap (
              struct trapframe * tf )
```

Definition at line 37 of file trap.c.

```
00038 {
00039   if(tf->trapno == T_SYSCALL){
00040     if(myproc()->killed)
00041       exit();
00042     myproc()->tf = tf;
00043     syscall();
00044     if(myproc()->killed)
00045       exit();
00046     return;
00047   }
00048
00049   switch(tf->trapno){
00050   case T_IRQ0 + IRQ_TIMER:
00051     if(cpuid() == 0){
00052       acquire(&tickslock);
00053       ticks++;
00054       wakeup(&ticks);
00055       release(&tickslock);
00056     }
00057     lapiceoi();
00058     break;
00059   case T_IRQ0 + IRQ_IDE:
00060     ideintr();
00061     lapiceoi();
00062     break;
00063   case T_IRQ0 + IRQ_IDE+1:
```

```
00064     // Bochs generates spurious IDE1 interrupts.
00065     break;
00066   case T_IRQ0 + IRQ_KBD:
00067     kbdintr();
00068     lapiceoi();
00069     break;
00070   case T_IRQ0 + IRQ_COM1:
00071     uartintr();
00072     lapiceoi();
00073     break;
00074   case T_IRQ0 + 7:
00075   case T_IRQ0 + IRQ_SPURIOUS:
00076     cprintf("cpu%d: spurious interrupt at %x:%x\n",
00077             cpuid(), tf->cs, tf->eip);
00078     lapiceoi();
00079     break;
00080
00081   //PAGEBREAK: 13
00082   default:
00083     if(myproc() == 0 || (tf->cs&3) == 0){
00084       // In kernel, it must be our mistake.
00085       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
00086               tf->trapno, cpuid(), tf->eip, rcr2());
00087       panic("trap");
00088     }
00089     // In user space, assume process misbehaved.
00090     cprintf("pid %d %s: trap %d err %d on cpu %d "
00091             "eip 0x%x addr 0x%x--kill proc\n",
00092             myproc()->pid, myproc()->name, tf->trapno,
00093             tf->err, cpuid(), tf->eip, rcr2());
00094     myproc()->killed = 1;
00095   }
00096
00097   // Force process exit if it has been killed and is in user space.
00098   // (If it is still executing in the kernel, let it keep running
00099   // until it gets to the regular system call return.)
00100   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
00101     exit();
00102
00103   // Force process to give up CPU on clock tick.
00104   // If interrupts were on while locks held, would need to check nlock.
00105   if(myproc() && myproc()->state == RUNNING &&
00106      tf->trapno == T_IRQ0+IRQ_TIMER)
00107     yield();
00108
00109   // Check if the process has been killed since we yielded
00110   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
00111     exit();
00112 }
```

5.216.1.3 tvinit()

```
void tvinit (
              void  )
```

Definition at line 18 of file trap.c.

```
00019 {
00020   int i;
00021
00022   for(i = 0; i < 256; i++)
00023     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
00024   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
00025
00026   initlock(&tickslock, "time");
00027 }
```

5.216.2 Variable Documentation

5.216.2.1 idt

```
struct gatedesc idt[256]
```

Definition at line 12 of file trap.c.

Referenced by idtinit(), and tvinit().

5.216.2.2 ticks

```
uint ticks
```

Definition at line 15 of file trap.c.

Referenced by sys_sleep(), sys_uptime(), and trap().

5.216.2.3 tickslock

```
struct spinlock tickslock
```

Definition at line 14 of file trap.c.

Referenced by sys_sleep(), sys_uptime(), trap(), and tvinit().

5.216.2.4 vectors

```
uint vectors[]  [extern]
```

Referenced by tvinit().

5.217 trap.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "defs.h"
00003 #include "param.h"
00004 #include "memlayout.h"
00005 #include "mmu.h"
00006 #include "proc.h"
00007 #include "x86.h"
00008 #include "traps.h"
00009 #include "spinlock.h"
00010
00011 // Interrupt descriptor table (shared by all CPUs).
00012 struct gatedesc idt[256];
00013 extern uint vectors[];  // in vectors.S: array of 256 entry pointers
00014 struct spinlock tickslock;
00015 uint ticks;
00016
00017 void
00018 tvinit(void)
00019 {
00020   int i;
00021
00022   for(i = 0; i < 256; i++)
00023     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
00024   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
00025
00026   initlock(&tickslock, "time");
00027 }
00028
00029 void
00030 idtinit(void)
00031 {
00032   lidt(idt, sizeof(idt));
00033 }
00034
00035 //PAGEBREAK: 41
00036 void
00037 trap(struct trapframe *tf)
00038 {
00039   if(tf->trapno == T_SYSCALL){
00040     if(myproc()->killed)
00041       exit();
00042     myproc()->tf = tf;
00043     syscall();
00044     if(myproc()->killed)
00045       exit();
00046     return;
00047   }
00048
00049   switch(tf->trapno){
00050   case T_IRQ0 + IRQ_TIMER:
00051     if(cpuid() == 0){
00052       acquire(&tickslock);
00053       ticks++;
00054       wakeup(&ticks);
00055       release(&tickslock);
00056     }
00057     lapiceoi();
00058     break;
00059   case T_IRQ0 + IRQ_IDE:
00060     ideintr();
00061     lapiceoi();
00062     break;
00063   case T_IRQ0 + IRQ_IDE+1:
00064     // Bochs generates spurious IDE1 interrupts.
00065     break;
00066   case T_IRQ0 + IRQ_KBD:
00067     kbdintr();
00068     lapiceoi();
00069     break;
00070   case T_IRQ0 + IRQ_COM1:
00071     uartintr();
00072     lapiceoi();
00073     break;
00074   case T_IRQ0 + 7:
00075   case T_IRQ0 + IRQ_SPURIOUS:
00076     cprintf("cpu%d: spurious interrupt at %x:%x\n",
00077             cpuid(), tf->cs, tf->eip);
00078     lapiceoi();
00079     break;
00080
00081   //PAGEBREAK: 13
00082   default:
```

```
00083     if(myproc() == 0 || (tf->cs&3) == 0){
00084       // In kernel, it must be our mistake.
00085       cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
00086               tf->trapno, cpuid(), tf->eip, rcr2());
00087       panic("trap");
00088     }
00089     // In user space, assume process misbehaved.
00090     cprintf("pid %d %s: trap %d err %d on cpu %d "
00091             "eip 0x%x addr 0x%x--kill proc\n",
00092             myproc()->pid, myproc()->name, tf->trapno,
00093             tf->err, cpuid(), tf->eip, rcr2());
00094     myproc()->killed = 1;
00095   }
00096
00097   // Force process exit if it has been killed and is in user space.
00098   // (If it is still executing in the kernel, let it keep running
00099   // until it gets to the regular system call return.)
00100   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
00101     exit();
00102
00103   // Force process to give up CPU on clock tick.
00104   // If interrupts were on while locks held, would need to check nlock.
00105   if(myproc() && myproc()->state == RUNNING &&
00106      tf->trapno == T_IRQ0+IRQ_TIMER)
00107     yield();
00108
00109   // Check if the process has been killed since we yielded
00110   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
00111     exit();
00112 }
```

5.218 trap.d File Reference

5.219 trap.d

Go to the documentation of this file.

```
00001 trap.o: trap.c /usr/include/stdc-predef.h types.h defs.h param.h \
00002  memlayout.h mmu.h proc.h x86.h traps.h spinlock.h
```

5.220 traps.h File Reference

Macros

- #define IRQ_COM1 4
- #define IRQ_ERROR 19
- #define IRQ_IDE 14
- #define IRQ_KBD 1
- #define IRQ_SPURIOUS 31
- #define IRQ_TIMER 0
- #define T_ALIGN 17
- #define T_BOUND 5
- #define T_BRKPT 3
- #define T_DBLFLT 8
- #define T_DEBUG 1
- #define T_DEFAULT 500
- #define T_DEVICE 7
- #define T_DIVIDE 0
- #define T_FPERR 16
- #define T_GPFLT 13
- #define T_ILLOP 6
- #define T_IRQ0 32

- #define T_MCHK 18
- #define T_NMI 2
- #define T_OFLOW 4
- #define T_PGFLT 14
- #define T_SEGNP 11
- #define T_SIMDERR 19
- #define T_STACK 12
- #define T_SYSCALL 64
- #define T_TSS 10

5.220.1 Macro Definition Documentation

5.220.1.1 IRQ_COM1

```
#define IRQ_COM1 4
```

Definition at line 34 of file traps.h.

5.220.1.2 IRQ_ERROR

```
#define IRQ_ERROR 19
```

Definition at line 36 of file traps.h.

5.220.1.3 IRQ_IDE

```
#define IRQ_IDE 14
```

Definition at line 35 of file traps.h.

5.220.1.4 IRQ_KBD

```
#define IRQ_KBD 1
```

Definition at line 33 of file traps.h.

5.220.1.5 IRQ_SPURIOUS

```
#define IRQ_SPURIOUS 31
```

Definition at line 37 of file traps.h.

5.220.1.6 IRQ_TIMER

```
#define IRQ_TIMER 0
```

Definition at line 32 of file traps.h.

5.220.1.7 T_ALIGN

```
#define T_ALIGN 17
```

Definition at line 21 of file traps.h.

5.220.1.8 T_BOUND

```
#define T_BOUND 5
```

Definition at line 9 of file traps.h.

5.220.1.9 T_BRKPT

```
#define T_BRKPT 3
```

Definition at line 7 of file traps.h.

5.220.1.10 T_DBLFLT

```
#define T_DBLFLT 8
```

Definition at line 12 of file traps.h.

5.220.1.11 T_DEBUG

```
#define T_DEBUG 1
```

Definition at line 5 of file traps.h.

5.220.1.12 T_DEFAULT

```
#define T_DEFAULT 500
```

Definition at line 28 of file traps.h.

5.220.1.13 T_DEVICE

```
#define T_DEVICE 7
```

Definition at line 11 of file traps.h.

5.220.1.14 T_DIVIDE

```
#define T_DIVIDE 0
```

Definition at line 4 of file traps.h.

5.220.1.15 T_FPERR

```
#define T_FPERR 16
```

Definition at line 20 of file traps.h.

5.220.1.16 T_GPFLT

```
#define T_GPFLT 13
```

Definition at line 17 of file traps.h.

5.220.1.17 T_ILLOP

```
#define T_ILLOP 6
```

Definition at line 10 of file traps.h.

5.220.1.18 T_IRQ0

```
#define T_IRQ0 32
```

Definition at line 30 of file traps.h.

5.220.1.19 T_MCHK

```
#define T_MCHK 18
```

Definition at line 22 of file traps.h.

5.220.1.20 T_NMI

```
#define T_NMI 2
```

Definition at line 6 of file traps.h.

5.220.1.21 T_OFLOW

```
#define T_OFLOW 4
```

Definition at line 8 of file traps.h.

5.220.1.22 T_PGFLT

```
#define T_PGFLT 14
```

Definition at line 18 of file traps.h.

5.220.1.23 T_SEGNP

```
#define T_SEGNP 11
```

Definition at line 15 of file traps.h.

5.220.1.24 T_SIMDERR

```
#define T_SIMDERR 19
```

Definition at line 23 of file traps.h.

5.220.1.25 T_STACK

```
#define T_STACK 12
```

Definition at line 16 of file traps.h.

5.220.1.26 T_SYSCALL

```
#define T_SYSCALL 64
```

Definition at line 27 of file traps.h.

5.220.1.27 T_TSS

```
#define T_TSS 10
```

Definition at line 14 of file traps.h.

5.221 traps.h

Go to the documentation of this file.

```
00001 // x86 trap and interrupt constants.
00002
00003 // Processor-defined:
00004 #define T_DIVIDE         0      // divide error
00005 #define T_DEBUG          1      // debug exception
00006 #define T_NMI            2      // non-maskable interrupt
00007 #define T_BRKPT          3      // breakpoint
00008 #define T_OFLOW          4      // overflow
00009 #define T_BOUND          5      // bounds check
00010 #define T_ILLOP          6      // illegal opcode
00011 #define T_DEVICE         7      // device not available
00012 #define T_DBLFLT         8      // double fault
00013 // #define T_COPROC      9      // reserved (not used since 486)
00014 #define T_TSS           10      // invalid task switch segment
00015 #define T_SEGNP         11      // segment not present
00016 #define T_STACK         12      // stack exception
00017 #define T_GPFLT         13      // general protection fault
00018 #define T_PGFLT         14      // page fault
00019 // #define T_RES        15      // reserved
00020 #define T_FPERR         16      // floating point error
00021 #define T_ALIGN         17      // aligment check
00022 #define T_MCHK          18      // machine check
00023 #define T_SIMDERR       19      // SIMD floating point error
00024
00025 // These are arbitrarily chosen, but with care not to overlap
00026 // processor defined exceptions or interrupt vectors.
00027 #define T_SYSCALL       64      // system call
00028 #define T_DEFAULT      500      // catchall
00029
00030 #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
00031
00032 #define IRQ_TIMER        0
00033 #define IRQ_KBD          1
00034 #define IRQ_COM1         4
00035 #define IRQ_IDE         14
00036 #define IRQ_ERROR       19
00037 #define IRQ_SPURIOUS    31
00038
```

5.222 types.h File Reference

Typedefs

- typedef uint pde_t
- typedef unsigned char uchar
- typedef unsigned int uint
- typedef unsigned short ushort

5.222.1 Typedef Documentation

5.222.1.1 pde_t

```
typedef uint pde_t
```

Definition at line 4 of file types.h.

5.222.1.2 uchar

```
typedef unsigned char uchar
```

Definition at line 3 of file types.h.

5.222.1.3 uint

```
typedef unsigned int uint
```

Definition at line 1 of file types.h.

5.222.1.4 ushort

```
typedef unsigned short ushort
```

Definition at line 2 of file types.h.

5.223 types.h

Go to the documentation of this file.

```
00001 typedef unsigned int   uint;
00002 typedef unsigned short ushort;
00003 typedef unsigned char  uchar;
00004 typedef uint pde_t;
```

5.224 uart.c File Reference

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "traps.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "file.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
```

Macros

- #define COM1 0x3f8

Functions

- static int uartgetc (void)
- void uartinit (void)
- void uartintr (void)
- void uartputc (int c)

Variables

- static int uart

5.224.1 Macro Definition Documentation

5.224.1.1 COM1

```
#define COM1 0x3f8
```

Definition at line 15 of file uart.c.

5.224.2 Function Documentation

5.224.2.1 uartgetc()

```
static int uartgetc (
             void  )  [static]
```

Definition at line 64 of file uart.c.

```
00065 {
00066   if(!uart)
00067     return -1;
00068   if(!(inb(COM1+5) & 0x01))
00069     return -1;
00070   return inb(COM1+0);
00071 }
```

Referenced by uartintr().

5.224.2.2 uartinit()

```
void uartinit (
              void  )
```

Definition at line 20 of file uart.c.

```
00021 {
00022   char *p;
00023
00024   // Turn off the FIFO
00025   outb(COM1+2, 0);
00026
00027   // 9600 baud, 8 data bits, 1 stop bit, parity off.
00028   outb(COM1+3, 0x80);    // Unlock divisor
00029   outb(COM1+0, 115200/9600);
00030   outb(COM1+1, 0);
00031   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
00032   outb(COM1+4, 0);
00033   outb(COM1+1, 0x01);    // Enable receive interrupts.
00034
00035   // If status is 0xFF, no serial port.
00036   if(inb(COM1+5) == 0xFF)
00037     return;
00038   uart = 1;
00039
00040   // Acknowledge pre-existing interrupt conditions;
00041   // enable interrupts.
00042   inb(COM1+2);
00043   inb(COM1+0);
00044   ioapicenable(IRQ_COM1, 0);
00045
00046   // Announce that we're here.
00047   for(p="xv6...\n"; *p; p++)
00048     uartputc(*p);
00049 }
```

5.224.2.3 uartintr()

```
void uartintr (
              void  )
```

Definition at line 74 of file uart.c.

```
00075 {
00076   consoleintr(uartgetc);
00077 }
```

Referenced by trap().

5.224.2.4 uartputc()

```
void uartputc (
              int c )
```

Definition at line 52 of file uart.c.

```
00053 {
00054   int i;
00055
00056   if(!uart)
00057     return;
00058   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
00059     microdelay(10);
00060   outb(COM1+0, c);
00061 }
```

Referenced by consputc(), and uartinit().

5.224.3 Variable Documentation

5.224.3.1 uart

```
int uart  [static]
```

Definition at line 17 of file uart.c.

Referenced by uartgetc(), uartinit(), and uartputc().

5.225 uart.c

Go to the documentation of this file.

```
00001 // Intel 8250 serial port (UART).
00002
00003 #include "types.h"
00004 #include "defs.h"
00005 #include "param.h"
00006 #include "traps.h"
00007 #include "spinlock.h"
00008 #include "sleeplock.h"
00009 #include "fs.h"
00010 #include "file.h"
00011 #include "mmu.h"
00012 #include "proc.h"
00013 #include "x86.h"
00014
00015 #define COM1    0x3f8
00016
00017 static int uart;    // is there a uart?
00018
00019 void
00020 uartinit(void)
00021 {
00022   char *p;
00023
00024   // Turn off the FIFO
00025   outb(COM1+2, 0);
00026
00027   // 9600 baud, 8 data bits, 1 stop bit, parity off.
00028   outb(COM1+3, 0x80);    // Unlock divisor
00029   outb(COM1+0, 115200/9600);
00030   outb(COM1+1, 0);
00031   outb(COM1+3, 0x03);    // Lock divisor, 8 data bits.
00032   outb(COM1+4, 0);
00033   outb(COM1+1, 0x01);    // Enable receive interrupts.
00034
00035   // If status is 0xFF, no serial port.
00036   if(inb(COM1+5) == 0xFF)
00037     return;
00038   uart = 1;
00039
00040   // Acknowledge pre-existing interrupt conditions;
00041   // enable interrupts.
00042   inb(COM1+2);
00043   inb(COM1+0);
00044   ioapicenable(IRQ_COM1, 0);
00045
00046   // Announce that we're here.
00047   for(p="xv6...\n"; *p; p++)
00048     uartputc(*p);
00049 }
00050
00051 void
00052 uartputc(int c)
00053 {
00054   int i;
00055
00056   if(!uart)
00057     return;
00058   for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
```

```
00059     microdelay(10);
00060   outb(COM1+0, c);
00061 }
00062
00063 static int
00064 uartgetc(void)
00065 {
00066   if(!uart)
00067     return -1;
00068   if(!(inb(COM1+5) & 0x01))
00069     return -1;
00070   return inb(COM1+0);
00071 }
00072
00073 void
00074 uartintr(void)
00075 {
00076   consoleintr(uartgetc);
00077 }
```

5.226 uart.d File Reference

5.227 uart.d

Go to the documentation of this file.

```
00001 uart.o: uart.c /usr/include/stdc-predef.h types.h defs.h param.h traps.h \
00002  spinlock.h sleeplock.h fs.h file.h mmu.h proc.h x86.h
```

5.228 ulib.c File Reference

```
#include "types.h"
#include "stat.h"
#include "fcntl.h"
#include "user.h"
#include "x86.h"
```

Functions

- int atoi (const char *s)
- char * gets (char *buf, int max)
- void * memmove (void *vdst, const void *vsrc, int n)
- void * memset (void *dst, int c, uint n)
- int stat (const char *n, struct stat *st)
- char * strchr (const char *s, char c)
- int strcmp (const char *p, const char *q)
- char * strcpy (char *s, const char *t)
- uint strlen (const char *s)

5.228.1 Function Documentation

5.228.1.1 atoi()

```
int atoi (
              const char * s )
```

Definition at line 85 of file ulib.c.

```
00086 {
00087   int n;
00088
00089   n = 0;
00090   while('0' <= *s && *s <= '9')
00091     n = n*10 + *s++ - '0';
00092   return n;
00093 }
```

Referenced by main().

5.228.1.2 gets()

```
char * gets (
              char * buf,
              int max )
```

Definition at line 53 of file ulib.c.

```
00054 {
00055   int i, cc;
00056   char c;
00057
00058   for(i=0; i+1 < max; ){
00059     cc = read(0, &c, 1);
00060     if(cc < 1)
00061       break;
00062     buf[i++] = c;
00063     if(c == '\n' || c == '\r')
00064       break;
00065   }
00066   buf[i] = '\0';
00067   return buf;
00068 }
```

Referenced by getcmd().

5.228.1.3 memmove()

```
void * memmove (
              void * vdst,
              const void * vsrc,
              int n )
```

Definition at line 96 of file ulib.c.

```
00097 {
00098   char *dst;
00099   const char *src;
00100
00101   dst = vdst;
00102   src = vsrc;
00103   while(n-- > 0)
00104     *dst++ = *src++;
00105   return vdst;
00106 }
```

5.228.1.4 memset()

```
void * memset (
              void * dst,
              int c,
              uint n )
```

Definition at line 37 of file ulib.c.

```
00038 {
00039   stosb(dst, c, n);
00040   return dst;
00041 }
```

Referenced by allocproc(), allocuvm(), backcmd(), bigfile(), bzero(), cgaputc(), concreate(), execcmd(), fmtname(), fourfiles(), getcmd(), ialloc(), inituvm(), kfree(), listcmd(), main(), pipecmd(), redircmd(), setupkvm(), sharedfd(), sys_exec(), sys_unlink(), userinit(), and walkpgdir().

5.228.1.5 stat()

```
int stat (
              const char * n,
              struct stat * st )
```

Definition at line 71 of file ulib.c.

```
00072 {
00073   int fd;
00074   int r;
00075
00076   fd = open(n, O_RDONLY);
00077   if(fd < 0)
00078     return -1;
00079   r = fstat(fd, st);
00080   close(fd);
00081   return r;
00082 }
```

5.228.1.6 strchr()

```
char * strchr (
              const char * s,
              char c )
```

Definition at line 44 of file ulib.c.

```
00045 {
00046   for(; *s; s++)
00047     if(*s == c)
00048       return (char*)s;
00049   return 0;
00050 }
```

Referenced by gettoken(), grep(), peek(), and wc().

5.228.1.7 strcmp()

```
int strcmp (
             const char * p,
             const char * q )
```

Definition at line 19 of file ulib.c.

```
00020 {
00021   while(*p && *p == *q)
00022     p++, q++;
00023   return (uchar)*p - (uchar)*q;
00024 }
```

5.228.1.8 strcpy()

```
char * strcpy (
             char * s,
             const char * t )
```

Definition at line 8 of file ulib.c.

```
00009 {
00010   char *os;
00011
00012   os = s;
00013   while((*s++ = *t++) != 0)
00014     ;
00015   return os;
00016 }
```

Referenced by ls(), and main().

5.228.1.9 strlen()

```
uint strlen (
             const char * s )
```

Definition at line 27 of file ulib.c.

```
00028 {
00029   int n;
00030
00031   for(n = 0; s[n]; n++)
00032     ;
00033   return n;
00034 }
```

Referenced by exec(), fmtname(), ls(), main(), parsecmd(), and printf().

5.229 ulib.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "fcntl.h"
00004 #include "user.h"
00005 #include "x86.h"
00006
00007 char*
00008 strcpy(char *s, const char *t)
00009 {
00010   char *os;
00011
00012   os = s;
00013   while((*s++ = *t++) != 0)
00014     ;
00015   return os;
00016 }
00017
00018 int
00019 strcmp(const char *p, const char *q)
00020 {
00021   while(*p && *p == *q)
00022     p++, q++;
00023   return (uchar)*p - (uchar)*q;
00024 }
00025
00026 uint
00027 strlen(const char *s)
00028 {
00029   int n;
00030
00031   for(n = 0; s[n]; n++)
00032     ;
00033   return n;
00034 }
00035
00036 void*
00037 memset(void *dst, int c, uint n)
00038 {
00039   stosb(dst, c, n);
00040   return dst;
00041 }
00042
00043 char*
00044 strchr(const char *s, char c)
00045 {
00046   for(; *s; s++)
00047     if(*s == c)
00048       return (char*)s;
00049   return 0;
00050 }
00051
00052 char*
00053 gets(char *buf, int max)
00054 {
00055   int i, cc;
00056   char c;
00057
00058   for(i=0; i+1 < max; ){
00059     cc = read(0, &c, 1);
00060     if(cc < 1)
00061       break;
00062     buf[i++] = c;
00063     if(c == '\n' || c == '\r')
00064       break;
00065   }
00066   buf[i] = '\0';
00067   return buf;
00068 }
00069
00070 int
00071 stat(const char *n, struct stat *st)
00072 {
00073   int fd;
00074   int r;
00075
00076   fd = open(n, O_RDONLY);
00077   if(fd < 0)
00078     return -1;
00079   r = fstat(fd, st);
00080   close(fd);
00081   return r;
00082 }
```

```
00083
00084 int
00085 atoi(const char *s)
00086 {
00087   int n;
00088
00089   n = 0;
00090   while('0' <= *s && *s <= '9')
00091     n = n*10 + *s++ - '0';
00092   return n;
00093 }
00094
00095 void*
00096 memmove(void *vdst, const void *vsrc, int n)
00097 {
00098   char *dst;
00099   const char *src;
00100
00101   dst = vdst;
00102   src = vsrc;
00103   while(n-- > 0)
00104     *dst++ = *src++;
00105   return vdst;
00106 }
```

5.230 ulib.d File Reference

5.231 ulib.d

Go to the documentation of this file.

```
00001 ulib.o: ulib.c /usr/include/stdc-predef.h types.h stat.h fcntl.h user.h \
00002  x86.h
```

5.232 umalloc.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "param.h"
```

Classes

- union header

Typedefs

- typedef long Align
- typedef union header Header

Functions

- void free (void *ap)
- void * malloc (uint nbytes)
- static Header * morecore (uint nu)

Variables

- static Header base
- static Header * freep

5.232.1 Typedef Documentation

5.232.1.1 Align

```
typedef long Align
```

Definition at line 9 of file umalloc.c.

5.232.1.2 Header

```
typedef union header Header
```

Definition at line 19 of file umalloc.c.

5.232.2 Function Documentation

5.232.2.1 free()

```
void free (
            void * ap )
```

Definition at line 25 of file umalloc.c.

```
00026 {
00027   Header *bp, *p;
00028
00029   bp = (Header*)ap - 1;
00030   for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
00031     if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
00032       break;
00033   if(bp + bp->s.size == p->s.ptr){
00034     bp->s.size += p->s.ptr->s.size;
00035     bp->s.ptr = p->s.ptr->s.ptr;
00036   } else
00037     bp->s.ptr = p->s.ptr;
00038   if(p + p->s.size == bp){
00039     p->s.size += bp->s.size;
00040     p->s.ptr = bp->s.ptr;
00041   } else
00042     p->s.ptr = bp;
00043   freep = p;
00044 }
```

Referenced by mem(), and morecore().

5.232.2.2 malloc()

```
void * malloc (
            uint nbytes )
```

Definition at line 64 of file umalloc.c.

```
00065 {
00066   Header *p, *prevp;
00067   uint nunits;
00068
00069   nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
00070   if((prevp = freep) == 0){
00071     base.s.ptr = freep = prevp = &base;
00072     base.s.size = 0;
00073   }
00074   for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
00075     if(p->s.size >= nunits){
00076       if(p->s.size == nunits)
00077         prevp->s.ptr = p->s.ptr;
00078       else {
00079         p->s.size -= nunits;
00080         p += p->s.size;
00081         p->s.size = nunits;
00082       }
00083       freep = prevp;
00084       return (void*)(p + 1);
00085     }
00086     if(p == freep)
00087       if((p = morecore(nunits)) == 0)
00088         return 0;
00089   }
00090 }
```

Referenced by backcmd(), execcmd(), listcmd(), main(), mem(), pipecmd(), and redircmd().

5.232.2.3 morecore()

```
static Header * morecore (
            uint nu )  [static]
```

Definition at line 47 of file umalloc.c.

```
00048 {
00049   char *p;
00050   Header *hp;
00051
00052   if(nu < 4096)
00053     nu = 4096;
00054   p = sbrk(nu * sizeof(Header));
00055   if(p == (char*)-1)
00056     return 0;
00057   hp = (Header*)p;
00058   hp->s.size = nu;
00059   free((void*)(hp + 1));
00060   return freep;
00061 }
```

Referenced by malloc().

5.232.3 Variable Documentation

5.232.3.1 base

```
Header base  [static]
```

Definition at line 21 of file umalloc.c.

Referenced by malloc(), and printint().

5.232.3.2 freep

```
Header* freep  [static]
```

Definition at line 22 of file umalloc.c.

Referenced by free(), malloc(), and morecore().

5.233 umalloc.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004 #include "param.h"
00005
00006 // Memory allocator by Kernighan and Ritchie,
00007 // The C programming Language, 2nd ed.  Section 8.7.
00008
00009 typedef long Align;
00010
00011 union header {
00012   struct {
00013     union header *ptr;
00014     uint size;
00015   } s;
00016   Align x;
00017 };
00018
00019 typedef union header Header;
00020
00021 static Header base;
00022 static Header *freep;
00023
00024 void
00025 free(void *ap)
00026 {
00027   Header *bp, *p;
00028
00029   bp = (Header*)ap - 1;
00030   for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
00031     if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
00032       break;
00033   if(bp + bp->s.size == p->s.ptr){
00034     bp->s.size += p->s.ptr->s.size;
00035     bp->s.ptr = p->s.ptr->s.ptr;
00036   } else
00037     bp->s.ptr = p->s.ptr;
00038   if(p + p->s.size == bp){
00039     p->s.size += bp->s.size;
00040     p->s.ptr = bp->s.ptr;
00041   } else
00042     p->s.ptr = bp;
00043   freep = p;
00044 }
00045
00046 static Header*
00047 morecore(uint nu)
00048 {
00049   char *p;
```

```
00050   Header *hp;
00051
00052   if(nu < 4096)
00053     nu = 4096;
00054   p = sbrk(nu * sizeof(Header));
00055   if(p == (char*)-1)
00056     return 0;
00057   hp = (Header*)p;
00058   hp->s.size = nu;
00059   free((void*)(hp + 1));
00060   return freep;
00061 }
00062
00063 void*
00064 malloc(uint nbytes)
00065 {
00066   Header *p, *prevp;
00067   uint nunits;
00068
00069   nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
00070   if((prevp = freep) == 0){
00071     base.s.ptr = freep = prevp = &base;
00072     base.s.size = 0;
00073   }
00074   for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
00075     if(p->s.size >= nunits){
00076       if(p->s.size == nunits)
00077         prevp->s.ptr = p->s.ptr;
00078       else {
00079         p->s.size -= nunits;
00080         p += p->s.size;
00081         p->s.size = nunits;
00082       }
00083       freep = prevp;
00084       return (void*)(p + 1);
00085     }
00086     if(p == freep)
00087       if((p = morecore(nunits)) == 0)
00088         return 0;
00089   }
00090 }
```

5.234 umalloc.d File Reference

5.235 umalloc.d

Go to the documentation of this file.

```
00001 umalloc.o: umalloc.c /usr/include/stdc-predef.h types.h stat.h user.h \
00002  param.h
```

5.236 uproc.h File Reference

Classes

- struct uproc

5.237 uproc.h

Go to the documentation of this file.

```
00001 struct uproc {
00002     int pid;
00003     int ppid;
00004     char name[16];
00005 };
```

5.238 user.h File Reference

Functions

- int atoi (const char *)
- int chdir (const char *)
- int chpr (int pid, int priority)
- int close (int)
- int cps (void)
- int dup (int)
- int exec (char *, char **)
- int exit (void) __attribute__((noreturn))
- int fork (void)
- void free (void *)
- int fstat (int fd, struct stat *)
- int getpid (void)
- int getprocs (int max, struct uproc *)
- char * gets (char *, int max)
- int halt (void)
- int kill (int)
- int link (const char *, const char *)
- void * malloc (uint)
- void * memmove (void *, const void *, int)
- void * memset (void *, int, uint)
- int mkdir (const char *)
- int mknod (const char *, short, short)
- int open (const char *, int)
- int pipe (int *)
- void printf (int, const char *,...)
- int read (int, void *, int)
- char * sbrk (int)
- int sleep (int)
- int stat (const char *, struct stat *)
- char * strchr (const char *, char c)
- int strcmp (const char *, const char *)
- char * strcpy (char *, const char *)
- uint strlen (const char *)
- int unlink (const char *)
- int uptime (void)
- int wait (void)
- int write (int, const void *, int)

5.238.1 Function Documentation

5.238.1.1 atoi()

```
int atoi (
            const char * s )
```

Definition at line 85 of file ulib.c.

```
00086 {
00087   int n;
00088
00089   n = 0;
00090   while('0' <= *s && *s <= '9')
00091     n = n*10 + *s++ - '0';
00092   return n;
00093 }
```

Referenced by main().

5.238.1.2 chdir()

```
int chdir (
            const char *  )
```

Referenced by dirfile(), dirtest(), exitiputtest(), iputtest(), iref(), main(), rmdot(), and subdir().

5.238.1.3 chpr()

```
int chpr (
            int pid,
            int priority )
```

Definition at line 559 of file proc.c.

```
00560 {
00561   struct proc *p;
00562   acquire(&ptable.lock);
00563   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00564     if(p->pid == pid){
00565       p->priority = priority;
00566       break;
00567     }
00568   }
00569   release(&ptable.lock);
00570   return pid;
00571 }
```

5.238.1.4 close()

```
int close (
            int  )
```

Referenced by argptest(), bigargtest(), bigdir(), bigfile(), bigwrite(), concreate(), createdelete(), createtest(), dirfile(), fourfiles(), fourteen(), fsfull(), iref(), linktest(), linkunlink(), ls(), main(), opentest(), pipe1(), preempt(), runcmd(), sharedfd(), stat(), subdir(), unlinkread(), writetest(), and writetest1().

5.238.1.5 cps()

```
int cps (
              void  )
```

Definition at line 537 of file proc.c.

```
00538 {
00539 struct proc *p;
00540 //Enables interrupts on this processor.
00541 sti();
00542
00543 //Loop over process table looking for process with pid.
00544 acquire(&ptable.lock);
00545 cprintf("name \t pid \t state \t priority \n");
00546 for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00547   if(p->state == SLEEPING)
00548     cprintf("%s \t %d \t SLEEPING \t %d \n ", p->name,p->pid,p->priority);
00549   else if(p->state == RUNNING)
00550     cprintf("%s \t %d \t RUNNING \t %d \n ", p->name,p->pid,p->priority);
00551   else if(p->state == RUNNABLE)
00552     cprintf("%s \t %d \t RUNNABLE \t %d \n ", p->name,p->pid,p->priority);
00553 }
00554 release(&ptable.lock);
00555 return 23;
00556 }
```

5.238.1.6 dup()

```
int dup (
              int  )
```

Referenced by main(), and runcmd().

5.238.1.7 exec()

```
int exec (
              char * path,
              char ** argv )
```

Definition at line 11 of file exec.c.

```
00012 {
00013   char *s, *last;
00014   int i, off;
00015   uint argc, sz, sp, ustack[3+MAXARG+1];
00016   struct elfhdr elf;
00017   struct inode *ip;
00018   struct proghdr ph;
00019   pde_t *pgdir, *oldpgdir;
00020   struct proc *curproc = myproc();
00021
00022   begin_op();
00023
00024   if((ip = namei(path)) == 0){
00025     end_op();
00026     cprintf("exec: fail\n");
00027     return -1;
00028   }
00029   ilock(ip);
00030   pgdir = 0;
00031
00032   // Check ELF header
00033   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
00034     goto bad;
00035   if(elf.magic != ELF_MAGIC)
00036     goto bad;
```

```
00037
00038   if((pgdir = setupkvm()) == 0)
00039     goto bad;
00040
00041   // Load program into memory.
00042   sz = 0;
00043   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
00044     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
00045       goto bad;
00046     if(ph.type != ELF_PROG_LOAD)
00047       continue;
00048     if(ph.memsz < ph.filesz)
00049       goto bad;
00050     if(ph.vaddr + ph.memsz < ph.vaddr)
00051       goto bad;
00052     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
00053       goto bad;
00054     if(ph.vaddr % PGSIZE != 0)
00055       goto bad;
00056     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
00057       goto bad;
00058   }
00059   iunlockput(ip);
00060   end_op();
00061   ip = 0;
00062
00063   // Allocate two pages at the next page boundary.
00064   // Make the first inaccessible.  Use the second as the user stack.
00065   sz = PGROUNDUP(sz);
00066   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
00067     goto bad;
00068   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
00069   sp = sz;
00070
00071   // Push argument strings, prepare rest of stack in ustack.
00072   for(argc = 0; argv[argc]; argc++) {
00073     if(argc >= MAXARG)
00074       goto bad;
00075     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
00076     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
00077       goto bad;
00078     ustack[3+argc] = sp;
00079   }
00080   ustack[3+argc] = 0;
00081
00082   ustack[0] = 0xffffffff;  // fake return PC
00083   ustack[1] = argc;
00084   ustack[2] = sp - (argc+1)*4;  // argv pointer
00085
00086   sp -= (3+argc+1) * 4;
00087   if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
00088     goto bad;
00089
00090   // Save program name for debugging.
00091   for(last=s=path; *s; s++)
00092     if(*s == '/')
00093       last = s+1;
00094   safestrcpy(curproc->name, last, sizeof(curproc->name));
00095
00096   // Commit to the user image.
00097   oldpgdir = curproc->pgdir;
00098   curproc->pgdir = pgdir;
00099   curproc->sz = sz;
00100   curproc->tf->eip = elf.entry;  // main
00101   curproc->tf->esp = sp;
00102   switchuvm(curproc);
00103   freevm(oldpgdir);
00104   return 0;
00105
00106  bad:
00107   if(pgdir)
00108     freevm(pgdir);
00109   if(ip){
00110     iunlockput(ip);
00111     end_op();
00112   }
00113   return -1;
00114 }
```

5.238.1.8 exit()

```
int exit (
```

```
              void  )
```

Definition at line 228 of file proc.c.

```
00229 {
00230   struct proc *curproc = myproc();
00231   struct proc *p;
00232   int fd;
00233
00234   if(curproc == initproc)
00235     panic("init exiting");
00236
00237   // Close all open files.
00238   for(fd = 0; fd < NOFILE; fd++){
00239     if(curproc->ofile[fd]){
00240       fileclose(curproc->ofile[fd]);
00241       curproc->ofile[fd] = 0;
00242     }
00243   }
00244
00245   begin_op();
00246   iput(curproc->cwd);
00247   end_op();
00248   curproc->cwd = 0;
00249
00250   acquire(&ptable.lock);
00251
00252   // Parent might be sleeping in wait().
00253   wakeup1(curproc->parent);
00254
00255   // Pass abandoned children to init.
00256   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00257     if(p->parent == curproc){
00258       p->parent = initproc;
00259       if(p->state == ZOMBIE)
00260         wakeup1(initproc);
00261     }
00262   }
00263
00264   // Jump into the scheduler, never to return.
00265   curproc->state = ZOMBIE;
00266   sched();
00267   panic("zombie exit");
00268 }
```

5.238.1.9 fork()

```
int fork (
              void  )
```

Definition at line 181 of file proc.c.

```
00182 {
00183   int i, pid;
00184   struct proc *np;
00185   struct proc *curproc = myproc();
00186
00187   // Allocate process.
00188   if((np = allocproc()) == 0){
00189     return -1;
00190   }
00191
00192   // Copy process state from proc.
00193   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
00194     kfree(np->kstack);
00195     np->kstack = 0;
00196     np->state = UNUSED;
00197     return -1;
00198   }
00199   np->sz = curproc->sz;
00200   np->parent = curproc;
00201   *np->tf = *curproc->tf;
00202
00203   // Clear %eax so that fork returns 0 in the child.
00204   np->tf->eax = 0;
00205
00206   for(i = 0; i < NOFILE; i++)
00207     if(curproc->ofile[i])
00208       np->ofile[i] = filedup(curproc->ofile[i]);
```

```
00209   np->cwd = idup(curproc->cwd);
00210
00211   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
00212
00213   pid = np->pid;
00214
00215   acquire(&ptable.lock);
00216
00217   np->state = RUNNABLE;
00218
00219   release(&ptable.lock);
00220
00221   return pid;
00222 }
```

5.238.1.10 free()

```
void free (
              void * ap )
```

Definition at line 25 of file umalloc.c.

```
00026 {
00027   Header *bp, *p;
00028
00029   bp = (Header*)ap - 1;
00030   for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
00031     if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
00032       break;
00033   if(bp + bp->s.size == p->s.ptr){
00034     bp->s.size += p->s.ptr->s.size;
00035     bp->s.ptr = p->s.ptr->s.ptr;
00036   } else
00037     bp->s.ptr = p->s.ptr;
00038   if(p + p->s.size == bp){
00039     p->s.size += bp->s.size;
00040     p->s.ptr = bp->s.ptr;
00041   } else
00042     p->s.ptr = bp;
00043   freep = p;
00044 }
```

Referenced by mem(), and morecore().

5.238.1.11 fstat()

```
int fstat (
              int fd,
              struct stat *  )
```

Referenced by ls(), and stat().

5.238.1.12 getpid()

```
int getpid (
              void  )
```

Referenced by main(), mem(), and sbrktest().

5.238.1.13 getprocs()

```
int getprocs (
              int max,
              struct uproc *  )
```

5.238.1.14 gets()

```
char * gets (
              char * buf,
              int max )
```

Definition at line 53 of file ulib.c.

```
00054 {
00055   int i, cc;
00056   char c;
00057
00058   for(i=0; i+1 < max; ){
00059     cc = read(0, &c, 1);
00060     if(cc < 1)
00061       break;
00062     buf[i++] = c;
00063     if(c == '\n' || c == '\r')
00064       break;
00065   }
00066   buf[i] = '\0';
00067   return buf;
00068 }
```

Referenced by getcmd().

5.238.1.15 halt()

```
int halt (
              void  )
```

Referenced by main().

5.238.1.16 kill()

```
int kill (
              int pid )
```

Definition at line 480 of file proc.c.

```
00481 {
00482   struct proc *p;
00483
00484   acquire(&ptable.lock);
00485   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00486     if(p->pid == pid){
00487       p->killed = 1;
00488       // Wake process from sleep if necessary.
00489       if(p->state == SLEEPING)
00490         p->state = RUNNABLE;
00491       release(&ptable.lock);
00492       return 0;
00493     }
00494   }
00495   release(&ptable.lock);
00496   return -1;
00497 }
```

5.238.1.17 link()

```
int link (
              const char * ,
              const char *  )
```

Referenced by bigdir(), concreate(), dirfile(), iref(), linktest(), linkunlink(), main(), subdir(), and validatetest().

5.238.1.18 malloc()

```
void * malloc (
              uint nbytes )
```

Definition at line 64 of file umalloc.c.

```
00065 {
00066   Header *p, *prevp;
00067   uint nunits;
00068
00069   nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
00070   if((prevp = freep) == 0){
00071     base.s.ptr = freep = prevp = &base;
00072     base.s.size = 0;
00073   }
00074   for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
00075     if(p->s.size >= nunits){
00076       if(p->s.size == nunits)
00077         prevp->s.ptr = p->s.ptr;
00078       else {
00079         p->s.size -= nunits;
00080         p += p->s.size;
00081         p->s.size = nunits;
00082       }
00083       freep = prevp;
00084       return (void*)(p + 1);
00085     }
00086     if(p == freep)
00087       if((p = morecore(nunits)) == 0)
00088         return 0;
00089   }
00090 }
```

Referenced by backcmd(), execcmd(), listcmd(), main(), mem(), pipecmd(), and redircmd().

5.238.1.19 memmove()

```
void * memmove (
              void * vdst,
              const void * vsrc,
              int n )
```

Definition at line 96 of file ulib.c.

```
00097 {
00098   char *dst;
00099   const char *src;
00100
00101   dst = vdst;
00102   src = vsrc;
00103   while(n-- > 0)
00104     *dst++ = *src++;
00105   return vdst;
00106 }
```

5.238.1.20 memset()

```
void * memset (
            void * dst,
            int c,
            uint n )
```

Definition at line 5 of file string.c.

```
00006 {
00007   if ((int)dst%4 == 0 && n%4 == 0){
00008     c &= 0xFF;
00009     stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
00010   } else
00011     stosb(dst, c, n);
00012   return dst;
00013 }
```

5.238.1.21 mkdir()

```
int mkdir (
            const char *  )
```

Referenced by dirfile(), dirtest(), exitiputtest(), fourteen(), iputtest(), iref(), main(), openiputtest(), rmdot(), and subdir().

5.238.1.22 mknod()

```
int mknod (
            const char * ,
            short ,
            short  )
```

Referenced by main().

5.238.1.23 open()

```
int open (
            const char * ,
            int  )
```

Referenced by argptest(), bigargtest(), bigdir(), bigfile(), bigwrite(), concreate(), createdelete(), createtest(), dirfile(), fourfiles(), fourteen(), fsfull(), iref(), linktest(), linkunlink(), ls(), main(), openiputtest(), opentest(), runcmd(), sharedfd(), stat(), subdir(), unlinkread(), writetest(), and writetest1().

5.238.1.24 pipe()

```
int pipe (
            int *  )
```

5.238.1.25 printf()

```
void printf (
            int fd,
            const char * s,
             ...  )
```

Definition at line 11 of file forktest.c.

```
00012 {
00013   write(fd, s, strlen(s));
00014 }
```

Referenced by argptest(), balloc(), bigargtest(), bigdir(), bigfile(), bigwrite(), bsstest(), cat(), concreate(), createdelete(), createtest(), dirfile(), dirtest(), exectest(), exitiputtest(), exitwait(), forktest(), fourfiles(), fourteen(), fsfull(), getcmd(), iputtest(), iref(), linktest(), linkunlink(), ls(), main(), mem(), openiputtest(), opentest(), panic(), parsecmd(), pipe1(), preempt(), rmdot(), runcmd(), sbrktest(), sharedfd(), subdir(), uio(), unlinkread(), validatetest(), wc(), writetest(), and writetest1().

5.238.1.26 read()

```
int read (
            int ,
            void * ,
            int  )
```

Referenced by argptest(), bigfile(), cat(), concreate(), fourfiles(), gets(), grep(), linktest(), ls(), main(), pipe1(), preempt(), rsect(), sbrktest(), sharedfd(), subdir(), unlinkread(), wc(), writetest(), and writetest1().

5.238.1.27 sbrk()

```
char * sbrk (
            int  )
```

Referenced by argptest(), morecore(), and sbrktest().

5.238.1.28 sleep()

```
int sleep (
            int  )
```

5.238.1.29 stat()

```
int stat (
              const char * n,
              struct stat * st )
```

Definition at line 71 of file ulib.c.

```
00072 {
00073   int fd;
00074   int r;
00075
00076   fd = open(n, O_RDONLY);
00077   if(fd < 0)
00078     return -1;
00079   r = fstat(fd, st);
00080   close(fd);
00081   return r;
00082 }
```

5.238.1.30 strchr()

```
char * strchr (
              const char * s,
              char c )
```

Definition at line 44 of file ulib.c.

```
00045 {
00046   for(; *s; s++)
00047     if(*s == c)
00048       return (char*)s;
00049   return 0;
00050 }
```

Referenced by gettoken(), grep(), peek(), and wc().

5.238.1.31 strcmp()

```
int strcmp (
              const char * p,
              const char * q )
```

Definition at line 19 of file ulib.c.

```
00020 {
00021   while(*p && *p == *q)
00022     p++, q++;
00023   return (uchar)*p - (uchar)*q;
00024 }
```

5.238.1.32 strcpy()

```
char * strcpy (
              char * s,
              const char * t )
```

Definition at line 8 of file ulib.c.

```
00009 {
00010   char *os;
00011
00012   os = s;
00013   while((*s++ = *t++) != 0)
00014     ;
00015   return os;
00016 }
```

Referenced by ls(), and main().

5.238.1.33 strlen()

```
uint strlen (
              const char * s )
```

Definition at line 97 of file string.c.

```
00098 {
00099   int n;
00100
00101   for(n = 0; s[n]; n++)
00102     ;
00103   return n;
00104 }
```

5.238.1.34 unlink()

```
int unlink (
              const char *  )
```

Referenced by bigargtest(), bigdir(), bigfile(), bigwrite(), concreate(), createdelete(), createtest(), dirfile(), dirtest(), exitiputtest(), fourfiles(), fsfull(), iputtest(), iref(), linktest(), linkunlink(), main(), openiputtest(), rmdot(), sharedfd(), subdir(), unlinkread(), writetest(), and writetest1().

5.238.1.35 uptime()

```
int uptime (
              void  )
```

5.238.1.36 wait()

```
int wait (
              void  )
```

Definition at line 273 of file proc.c.

```
00274 {
00275   struct proc *p;
00276   int havekids, pid;
00277   struct proc *curproc = myproc();
00278
00279   acquire(&ptable.lock);
00280   for(;;){
00281     // Scan through table looking for exited children.
00282     havekids = 0;
00283     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
00284       if(p->parent != curproc)
00285         continue;
00286       havekids = 1;
00287       if(p->state == ZOMBIE){
00288         // Found one.
00289         pid = p->pid;
00290         kfree(p->kstack);
00291         p->kstack = 0;
00292         freevm(p->pgdir);
00293         p->pid = 0;
00294         p->parent = 0;
00295         p->name[0] = 0;
00296         p->killed = 0;
00297         p->state = UNUSED;
00298         release(&ptable.lock);
00299         return pid;
00300       }
00301     }
00302
00303     // No point waiting if we don't have any children.
00304     if(!havekids || curproc->killed){
00305       release(&ptable.lock);
00306       return -1;
00307     }
00308
00309     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
00310     sleep(curproc, &ptable.lock);  //DOC: wait-sleep
00311   }
00312 }
```

5.238.1.37 write()

```
int write (
              int ,
              const void * ,
              int  )
```

Referenced by bigfile(), bigwrite(), cat(), dirfile(), fourfiles(), fsfull(), grep(), linktest(), main(), pipe1(), preempt(), printf(), putc(), sbrktest(), sharedfd(), subdir(), unlinkread(), writetest(), writetest1(), and wsect().

5.239 user.h

Go to the documentation of this file.

```
00001 struct stat;
00002 struct rtcdate;
00003 struct uproc;
00004
00005 // system calls
00006 int fork(void);
00007 int exit(void) __attribute__((noreturn));
00008 int wait(void);
```

```
00009 int pipe(int*);
00010 int write(int, const void*, int);
00011 int read(int, void*, int);
00012 int close(int);
00013 int kill(int);
00014 int exec(char*, char**);
00015 int open(const char*, int);
00016 int mknod(const char*, short, short);
00017 int unlink(const char*);
00018 int fstat(int fd, struct stat*);
00019 int link(const char*, const char*);
00020 int mkdir(const char*);
00021 int chdir(const char*);
00022 int dup(int);
00023 int getpid(void);
00024 char* sbrk(int);
00025 int sleep(int);
00026 int uptime(void);
00027 int halt(void);
00028 int cps(void);
00029 int chpr(int pid, int priority);
00030 int getprocs(int max, struct uproc*);
00031
00032 // ulib.c
00033 int stat(const char*, struct stat*);
00034 char* strcpy(char*, const char*);
00035 void *memmove(void*, const void*, int);
00036 char* strchr(const char*, char c);
00037 int strcmp(const char*, const char*);
00038 void printf(int, const char*, ...);
00039 char* gets(char*, int max);
00040 uint strlen(const char*);
00041 void* memset(void*, int, uint);
00042 void* malloc(uint);
00043 void free(void*);
00044 int atoi(const char*);
```

5.240 usertests.c File Reference

```
#include "param.h"
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fs.h"
#include "fcntl.h"
#include "syscall.h"
#include "traps.h"
#include "memlayout.h"
```

Macros

- #define BIG (100∗1024∗1024)
- #define RTC_ADDR 0x70
- #define RTC_DATA 0x71

Functions

- void argptest ()
- void bigargtest (void)
- void bigdir (void)
- void bigfile (void)
- void bigwrite (void)
- void bsstest (void)

- void concreate (void)
- void createdelete (void)
- void createtest (void)
- void dirfile (void)
- void dirtest (void)
- void exectest (void)
- void exitiputtest (void)
- void exitwait (void)
- void forktest (void)
- void fourfiles (void)
- void fourteen (void)
- void fsfull ()
- void iputtest (void)
- void iref (void)
- void linktest (void)
- void linkunlink ()
- int main (int argc, char *argv[])
- void mem (void)
- void openiputtest (void)
- void opentest (void)
- void pipe1 (void)
- void preempt (void)
- unsigned int rand ()
- void rmdot (void)
- void sbrktest (void)
- void sharedfd (void)
- void subdir (void)
- void uio ()
- void unlinkread (void)
- void validateint (int *p)
- void validatetest (void)
- void writetest (void)
- void writetest1 (void)

Variables

- char buf [8192]
- char * echoargv [] = { "echo", "ALL", "TESTS", "PASSED", 0 }
- char name [3]
- unsigned long randstate = 1
- int stdout = 1
- char uninit [10000]

5.240.1 Macro Definition Documentation

5.240.1.1 BIG

```
#define BIG (100*1024*1024)
```

5.240.1.2 RTC_ADDR

```
#define RTC_ADDR 0x70
```

5.240.1.3 RTC_DATA

```
#define RTC_DATA 0x71
```

5.240.2 Function Documentation

5.240.2.1 argptest()

```
void argptest ( )
```

Definition at line 1727 of file usertests.c.

```
01728 {
01729   int fd;
01730   fd = open("init", O_RDONLY);
01731   if (fd < 0) {
01732     printf(2, "open failed\n");
01733     exit();
01734   }
01735   read(fd, sbrk(0) - 1, -1);
01736   close(fd);
01737   printf(1, "arg test passed\n");
01738 }
```

Referenced by main().

5.240.2.2 bigargtest()

```
void bigargtest (
              void  )
```

Definition at line 1613 of file usertests.c.

```
01614 {
01615   int pid, fd;
01616
01617   unlink("bigarg-ok");
01618   pid = fork();
01619   if(pid == 0){
01620     static char *args[MAXARG];
01621     int i;
01622     for(i = 0; i < MAXARG-1; i++)
01623       args[i] = "bigargs test: failed\n

                                      ";
01624     args[MAXARG-1] = 0;
01625     printf(stdout, "bigarg test\n");
01626     exec("echo", args);
01627     printf(stdout, "bigarg test ok\n");
01628     fd = open("bigarg-ok", O_CREATE);
01629     close(fd);
01630     exit();
01631   } else if(pid < 0){
```

```
01632     printf(stdout, "bigargtest: fork failed\n");
01633     exit();
01634   }
01635   wait();
01636   fd = open("bigarg-ok", 0);
01637   if(fd < 0){
01638     printf(stdout, "bigarg test failed!\n");
01639     exit();
01640   }
01641   close(fd);
01642   unlink("bigarg-ok");
01643 }
```

Referenced by main().

5.240.2.3 bigdir()

```
void bigdir (
              void  )
```

Definition at line 893 of file usertests.c.

```
00894 {
00895   int i, fd;
00896   char name[10];
00897
00898   printf(1, "bigdir test\n");
00899   unlink("bd");
00900
00901   fd = open("bd", O_CREATE);
00902   if(fd < 0){
00903     printf(1, "bigdir create failed\n");
00904     exit();
00905   }
00906   close(fd);
00907
00908   for(i = 0; i < 500; i++){
00909     name[0] = 'x';
00910     name[1] = '0' + (i / 64);
00911     name[2] = '0' + (i % 64);
00912     name[3] = '\0';
00913     if(link("bd", name) != 0){
00914       printf(1, "bigdir link failed\n");
00915       exit();
00916     }
00917   }
00918
00919   unlink("bd");
00920   for(i = 0; i < 500; i++){
00921     name[0] = 'x';
00922     name[1] = '0' + (i / 64);
00923     name[2] = '0' + (i % 64);
00924     name[3] = '\0';
00925     if(unlink(name) != 0){
00926       printf(1, "bigdir unlink failed");
00927       exit();
00928     }
00929   }
00930
00931   printf(1, "bigdir ok\n");
00932 }
```

Referenced by main().

5.240.2.4 bigfile()

```
void bigfile (
              void  )
```

Definition at line 1148 of file usertests.c.

```
01149 {
01150   int fd, i, total, cc;
01151
01152   printf(1, "bigfile test\n");
01153
01154   unlink("bigfile");
01155   fd = open("bigfile", O_CREATE | O_RDWR);
01156   if(fd < 0){
01157     printf(1, "cannot create bigfile");
01158     exit();
01159   }
01160   for(i = 0; i < 20; i++){
01161     memset(buf, i, 600);
01162     if(write(fd, buf, 600) != 600){
01163       printf(1, "write bigfile failed\n");
01164       exit();
01165     }
01166   }
01167   close(fd);
01168
01169   fd = open("bigfile", 0);
01170   if(fd < 0){
01171     printf(1, "cannot open bigfile\n");
01172     exit();
01173   }
01174   total = 0;
01175   for(i = 0; ; i++){
01176     cc = read(fd, buf, 300);
01177     if(cc < 0){
01178       printf(1, "read bigfile failed\n");
01179       exit();
01180     }
01181     if(cc == 0)
01182       break;
01183     if(cc != 300){
01184       printf(1, "short read bigfile\n");
01185       exit();
01186     }
01187     if(buf[0] != i/2 || buf[299] != i/2){
01188       printf(1, "read bigfile wrong data\n");
01189       exit();
01190     }
01191     total += cc;
01192   }
01193   close(fd);
01194   if(total != 20*600){
01195     printf(1, "read bigfile wrong total\n");
01196     exit();
01197   }
01198   unlink("bigfile");
01199
01200   printf(1, "bigfile test ok\n");
01201 }
```

Referenced by main().

5.240.2.5 bigwrite()

```
void bigwrite (
              void  )
```

Definition at line 1119 of file usertests.c.

```
01120 {
01121   int fd, sz;
01122
01123   printf(1, "bigwrite test\n");
01124
01125   unlink("bigwrite");
```

```
01126   for(sz = 499; sz < 12*512; sz += 471){
01127     fd = open("bigwrite", O_CREATE | O_RDWR);
01128     if(fd < 0){
01129       printf(1, "cannot create bigwrite\n");
01130       exit();
01131     }
01132     int i;
01133     for(i = 0; i < 2; i++){
01134       int cc = write(fd, buf, sz);
01135       if(cc != sz){
01136         printf(1, "write(%d) ret %d\n", sz, cc);
01137         exit();
01138       }
01139     }
01140     close(fd);
01141     unlink("bigwrite");
01142   }
01143
01144   printf(1, "bigwrite ok\n");
01145 }
```

Referenced by main().

5.240.2.6 bsstest()

```
void bsstest (
              void  )
```

Definition at line 1595 of file usertests.c.

```
01596 {
01597   int i;
01598
01599   printf(stdout, "bss test\n");
01600   for(i = 0; i < sizeof(uninit); i++){
01601     if(uninit[i] != '\0'){
01602       printf(stdout, "bss test failed\n");
01603       exit();
01604     }
01605   }
01606   printf(stdout, "bss test ok\n");
01607 }
```

Referenced by main().

5.240.2.7 concreate()

```
void concreate (
              void  )
```

Definition at line 765 of file usertests.c.

```
00766 {
00767   char file[3];
00768   int i, pid, n, fd;
00769   char fa[40];
00770   struct {
00771     ushort inum;
00772     char name[14];
00773   } de;
00774
00775   printf(1, "concreate test\n");
00776   file[0] = 'C';
00777   file[2] = '\0';
00778   for(i = 0; i < 40; i++){
00779     file[1] = '0' + i;
00780     unlink(file);
00781     pid = fork();
00782     if(pid && (i % 3) == 1){
```

```
00783       link("C0", file);
00784     } else if(pid == 0 && (i % 5) == 1){
00785       link("C0", file);
00786     } else {
00787       fd = open(file, O_CREATE | O_RDWR);
00788       if(fd < 0){
00789         printf(1, "concreate create %s failed\n", file);
00790         exit();
00791       }
00792       close(fd);
00793     }
00794     if(pid == 0)
00795       exit();
00796     else
00797       wait();
00798   }
00799
00800   memset(fa, 0, sizeof(fa));
00801   fd = open(".", 0);
00802   n = 0;
00803   while(read(fd, &de, sizeof(de)) > 0){
00804     if(de.inum == 0)
00805       continue;
00806     if(de.name[0] == 'C' && de.name[2] == '\0'){
00807       i = de.name[1] - '0';
00808       if(i < 0 || i >= sizeof(fa)){
00809         printf(1, "concreate weird file %s\n", de.name);
00810         exit();
00811       }
00812       if(fa[i]){
00813         printf(1, "concreate duplicate file %s\n", de.name);
00814         exit();
00815       }
00816       fa[i] = 1;
00817       n++;
00818     }
00819   }
00820   close(fd);
00821
00822   if(n != 40){
00823     printf(1, "concreate not enough files in directory listing\n");
00824     exit();
00825   }
00826
00827   for(i = 0; i < 40; i++){
00828     file[1] = '0' + i;
00829     pid = fork();
00830     if(pid < 0){
00831       printf(1, "fork failed\n");
00832       exit();
00833     }
00834     if(((i % 3) == 0 && pid == 0) ||
00835        ((i % 3) == 1 && pid != 0)){
00836       close(open(file, 0));
00837       close(open(file, 0));
00838       close(open(file, 0));
00839       close(open(file, 0));
00840     } else {
00841       unlink(file);
00842       unlink(file);
00843       unlink(file);
00844       unlink(file);
00845     }
00846     if(pid == 0)
00847       exit();
00848     else
00849       wait();
00850   }
00851
00852   printf(1, "concreate ok\n");
00853 }
```

Referenced by main().

5.240.2.8 createdelete()

```
void createdelete (
             void  )
```

Definition at line 584 of file usertests.c.

```
00585 {
00586   enum { N = 20 };
00587   int pid, i, fd, pi;
00588   char name[32];
00589
00590   printf(1, "createdelete test\n");
00591
00592   for(pi = 0; pi < 4; pi++){
00593     pid = fork();
00594     if(pid < 0){
00595       printf(1, "fork failed\n");
00596       exit();
00597     }
00598
00599     if(pid == 0){
00600       name[0] = 'p' + pi;
00601       name[2] = '\0';
00602       for(i = 0; i < N; i++){
00603         name[1] = '0' + i;
00604         fd = open(name, O_CREATE | O_RDWR);
00605         if(fd < 0){
00606           printf(1, "create failed\n");
00607           exit();
00608         }
00609         close(fd);
00610         if(i > 0 && (i % 2 ) == 0){
00611           name[1] = '0' + (i / 2);
00612           if(unlink(name) < 0){
00613             printf(1, "unlink failed\n");
00614             exit();
00615           }
00616         }
00617       }
00618       exit();
00619     }
00620   }
00621
00622   for(pi = 0; pi < 4; pi++){
00623     wait();
00624   }
00625
00626   name[0] = name[1] = name[2] = 0;
00627   for(i = 0; i < N; i++){
00628     for(pi = 0; pi < 4; pi++){
00629       name[0] = 'p' + pi;
00630       name[1] = '0' + i;
00631       fd = open(name, 0);
00632       if((i == 0 || i >= N/2) && fd < 0){
00633         printf(1, "oops createdelete %s didn't exist\n", name);
00634         exit();
00635       } else if((i >= 1 && i < N/2) && fd >= 0){
00636         printf(1, "oops createdelete %s did exist\n", name);
00637         exit();
00638       }
00639       if(fd >= 0)
00640         close(fd);
00641     }
00642   }
00643
00644   for(i = 0; i < N; i++){
00645     for(pi = 0; pi < 4; pi++){
00646       name[0] = 'p' + i;
00647       name[1] = '0' + i;
00648       unlink(name);
00649     }
00650   }
00651
00652   printf(1, "createdelete ok\n");
00653 }
```

Referenced by main().

5.240.2.9 createtest()

```
void createtest (
             void  )
```

Definition at line 245 of file usertests.c.

```
00246 {
00247   int i, fd;
00248
00249   printf(stdout, "many creates, followed by unlink test\n");
00250
00251   name[0] = 'a';
00252   name[2] = '\0';
00253   for(i = 0; i < 52; i++){
00254     name[1] = '0' + i;
00255     fd = open(name, O_CREATE|O_RDWR);
00256     close(fd);
00257   }
00258   name[0] = 'a';
00259   name[2] = '\0';
00260   for(i = 0; i < 52; i++){
00261     name[1] = '0' + i;
00262     unlink(name);
00263   }
00264   printf(stdout, "many creates, followed by unlink; ok\n");
00265 }
```

Referenced by main().

5.240.2.10 dirfile()

```
void dirfile (
              void  )
```

Definition at line 1284 of file usertests.c.

```
01285 {
01286   int fd;
01287
01288   printf(1, "dir vs file\n");
01289
01290   fd = open("dirfile", O_CREATE);
01291   if(fd < 0){
01292     printf(1, "create dirfile failed\n");
01293     exit();
01294   }
01295   close(fd);
01296   if(chdir("dirfile") == 0){
01297     printf(1, "chdir dirfile succeeded!\n");
01298     exit();
01299   }
01300   fd = open("dirfile/xx", 0);
01301   if(fd >= 0){
01302     printf(1, "create dirfile/xx succeeded!\n");
01303     exit();
01304   }
01305   fd = open("dirfile/xx", O_CREATE);
01306   if(fd >= 0){
01307     printf(1, "create dirfile/xx succeeded!\n");
01308     exit();
01309   }
01310   if(mkdir("dirfile/xx") == 0){
01311     printf(1, "mkdir dirfile/xx succeeded!\n");
01312     exit();
01313   }
01314   if(unlink("dirfile/xx") == 0){
01315     printf(1, "unlink dirfile/xx succeeded!\n");
01316     exit();
01317   }
01318   if(link("README", "dirfile/xx") == 0){
01319     printf(1, "link to dirfile/xx succeeded!\n");
01320     exit();
01321   }
01322   if(unlink("dirfile") != 0){
01323     printf(1, "unlink dirfile failed!\n");
01324     exit();
01325   }
01326
01327   fd = open(".", O_RDWR);
01328   if(fd >= 0){
01329     printf(1, "open . for writing succeeded!\n");
01330     exit();
01331   }
```

```
01332   fd = open(".", 0);
01333   if(write(fd, "x", 1) > 0){
01334     printf(1, "write . succeeded!\n");
01335     exit();
01336   }
01337   close(fd);
01338
01339   printf(1, "dir vs file OK\n");
01340 }
```

Referenced by main().

5.240.2.11 dirtest()

```
void dirtest (
              void  )
```

Definition at line 267 of file usertests.c.

```
00268 {
00269   printf(stdout, "mkdir test\n");
00270
00271   if(mkdir("dir0") < 0){
00272     printf(stdout, "mkdir failed\n");
00273     exit();
00274   }
00275
00276   if(chdir("dir0") < 0){
00277     printf(stdout, "chdir dir0 failed\n");
00278     exit();
00279   }
00280
00281   if(chdir("..") < 0){
00282     printf(stdout, "chdir .. failed\n");
00283     exit();
00284   }
00285
00286   if(unlink("dir0") < 0){
00287     printf(stdout, "unlink dir0 failed\n");
00288     exit();
00289   }
00290   printf(stdout, "mkdir test ok\n");
00291 }
```

5.240.2.12 exectest()

```
void exectest (
              void  )
```

Definition at line 294 of file usertests.c.

```
00295 {
00296   printf(stdout, "exec test\n");
00297   if(exec("echo", echoargv) < 0){
00298     printf(stdout, "exec echo failed\n");
00299     exit();
00300   }
00301 }
```

Referenced by main().

5.240.2.13 exitiputtest()

```
void exitiputtest (
             void  )
```

Definition at line 43 of file usertests.c.

```
00044 {
00045   int pid;
00046
00047   printf(stdout, "exitiput test\n");
00048
00049   pid = fork();
00050   if(pid < 0){
00051     printf(stdout, "fork failed\n");
00052     exit();
00053   }
00054   if(pid == 0){
00055     if(mkdir("iputdir") < 0){
00056       printf(stdout, "mkdir failed\n");
00057       exit();
00058     }
00059     if(chdir("iputdir") < 0){
00060       printf(stdout, "child chdir failed\n");
00061       exit();
00062     }
00063     if(unlink("../iputdir") < 0){
00064       printf(stdout, "unlink ../iputdir failed\n");
00065       exit();
00066     }
00067     exit();
00068   }
00069   wait();
00070   printf(stdout, "exitiput test ok\n");
00071 }
```

Referenced by main().

5.240.2.14 exitwait()

```
void exitwait (
             void  )
```

Definition at line 405 of file usertests.c.

```
00406 {
00407   int i, pid;
00408
00409   for(i = 0; i < 100; i++){
00410     pid = fork();
00411     if(pid < 0){
00412       printf(1, "fork failed\n");
00413       return;
00414     }
00415     if(pid){
00416       if(wait() != pid){
00417         printf(1, "wait wrong pid\n");
00418         return;
00419       }
00420     } else {
00421       exit();
00422     }
00423   }
00424   printf(1, "exitwait ok\n");
00425 }
```

Referenced by main().

5.240.2.15 forktest()

```
void forktest (
              void  )
```

Definition at line 1380 of file usertests.c.

```
01381 {
01382   int n, pid;
01383
01384   printf(1, "fork test\n");
01385
01386   for(n=0; n<1000; n++){
01387     pid = fork();
01388     if(pid < 0)
01389       break;
01390     if(pid == 0)
01391       exit();
01392   }
01393
01394   if(n == 1000){
01395     printf(1, "fork claimed to work 1000 times!\n");
01396     exit();
01397   }
01398
01399   for(; n > 0; n--){
01400     if(wait() < 0){
01401       printf(1, "wait stopped early\n");
01402       exit();
01403     }
01404   }
01405
01406   if(wait() != -1){
01407     printf(1, "wait got too many\n");
01408     exit();
01409   }
01410
01411   printf(1, "fork test OK\n");
01412 }
```

Referenced by main().

5.240.2.16 fourfiles()

```
void fourfiles (
              void  )
```

Definition at line 518 of file usertests.c.

```
00519 {
00520   int fd, pid, i, j, n, total, pi;
00521   char *names[] = { "f0", "f1", "f2", "f3" };
00522   char *fname;
00523
00524   printf(1, "fourfiles test\n");
00525
00526   for(pi = 0; pi < 4; pi++){
00527     fname = names[pi];
00528     unlink(fname);
00529
00530     pid = fork();
00531     if(pid < 0){
00532       printf(1, "fork failed\n");
00533       exit();
00534     }
00535
00536     if(pid == 0){
00537       fd = open(fname, O_CREATE | O_RDWR);
00538       if(fd < 0){
00539         printf(1, "create failed\n");
00540         exit();
00541       }
00542
00543       memset(buf, '0'+pi, 512);
00544       for(i = 0; i < 12; i++){
00545         if((n = write(fd, buf, 500)) != 500){
```

```
00546           printf(1, "write failed %d\n", n);
00547           exit();
00548         }
00549       }
00550       exit();
00551     }
00552   }
00553
00554   for(pi = 0; pi < 4; pi++){
00555     wait();
00556   }
00557
00558   for(i = 0; i < 2; i++){
00559     fname = names[i];
00560     fd = open(fname, 0);
00561     total = 0;
00562     while((n = read(fd, buf, sizeof(buf))) > 0){
00563       for(j = 0; j < n; j++){
00564         if(buf[j] != '0'+i){
00565           printf(1, "wrong char\n");
00566           exit();
00567         }
00568       }
00569       total += n;
00570     }
00571     close(fd);
00572     if(total != 12*500){
00573       printf(1, "wrong length %d\n", total);
00574       exit();
00575     }
00576     unlink(fname);
00577   }
00578
00579   printf(1, "fourfiles ok\n");
00580 }
```

Referenced by main().

5.240.2.17 fourteen()

```
void fourteen (
              void  )
```

Definition at line 1204 of file usertests.c.

```
01205 {
01206   int fd;
01207
01208   // DIRSIZ is 14.
01209   printf(1, "fourteen test\n");
01210
01211   if(mkdir("12345678901234") != 0){
01212     printf(1, "mkdir 12345678901234 failed\n");
01213     exit();
01214   }
01215   if(mkdir("12345678901234/123456789012345") != 0){
01216     printf(1, "mkdir 12345678901234/123456789012345 failed\n");
01217     exit();
01218   }
01219   fd = open("123456789012345/123456789012345/123456789012345", O_CREATE);
01220   if(fd < 0){
01221     printf(1, "create 123456789012345/123456789012345/123456789012345 failed\n");
01222     exit();
01223   }
01224   close(fd);
01225   fd = open("12345678901234/12345678901234/12345678901234", 0);
01226   if(fd < 0){
01227     printf(1, "open 12345678901234/12345678901234/12345678901234 failed\n");
01228     exit();
01229   }
01230   close(fd);
01231
01232   if(mkdir("12345678901234/12345678901234") == 0){
01233     printf(1, "mkdir 12345678901234/12345678901234 succeeded!\n");
01234     exit();
01235   }
01236   if(mkdir("123456789012345/12345678901234") == 0){
01237     printf(1, "mkdir 12345678901234/123456789012345 succeeded!\n");
```

```
01238     exit();
01239   }
01240
01241   printf(1, "fourteen ok\n");
01242 }
```

Referenced by main().

5.240.2.18 fsfull()

```
void fsfull ( )
```

Definition at line 1648 of file usertests.c.

```
01649 {
01650   int nfiles;
01651   int fsblocks = 0;
01652
01653   printf(1, "fsfull test\n");
01654
01655   for(nfiles = 0; ; nfiles++){
01656     char name[64];
01657     name[0] = 'f';
01658     name[1] = '0' + nfiles / 1000;
01659     name[2] = '0' + (nfiles % 1000) / 100;
01660     name[3] = '0' + (nfiles % 100) / 10;
01661     name[4] = '0' + (nfiles % 10);
01662     name[5] = '\0';
01663     printf(1, "writing %s\n", name);
01664     int fd = open(name, O_CREATE|O_RDWR);
01665     if(fd < 0){
01666       printf(1, "open %s failed\n", name);
01667       break;
01668     }
01669     int total = 0;
01670     while(1){
01671       int cc = write(fd, buf, 512);
01672       if(cc < 512)
01673         break;
01674       total += cc;
01675       fsblocks++;
01676     }
01677     printf(1, "wrote %d bytes\n", total);
01678     close(fd);
01679     if(total == 0)
01680       break;
01681   }
01682
01683   while(nfiles >= 0){
01684     char name[64];
01685     name[0] = 'f';
01686     name[1] = '0' + nfiles / 1000;
01687     name[2] = '0' + (nfiles % 1000) / 100;
01688     name[3] = '0' + (nfiles % 100) / 10;
01689     name[4] = '0' + (nfiles % 10);
01690     name[5] = '\0';
01691     unlink(name);
01692     nfiles--;
01693   }
01694
01695   printf(1, "fsfull test finished\n");
01696 }
```

5.240.2.19 iputtest()

```
void iputtest (
              void  )
```

Definition at line 18 of file usertests.c.

```
00019 {
```

```
00020   printf(stdout, "iput test\n");
00021
00022   if(mkdir("iputdir") < 0){
00023     printf(stdout, "mkdir failed\n");
00024     exit();
00025   }
00026   if(chdir("iputdir") < 0){
00027     printf(stdout, "chdir iputdir failed\n");
00028     exit();
00029   }
00030   if(unlink("../iputdir") < 0){
00031     printf(stdout, "unlink ../iputdir failed\n");
00032     exit();
00033   }
00034   if(chdir("/") < 0){
00035     printf(stdout, "chdir / failed\n");
00036     exit();
00037   }
00038   printf(stdout, "iput test ok\n");
00039 }
```

Referenced by main().

5.240.2.20 iref()

```
void iref (
              void  )
```

Definition at line 1344 of file usertests.c.

```
01345 {
01346   int i, fd;
01347
01348   printf(1, "empty file name\n");
01349
01350   // the 50 is NINODE
01351   for(i = 0; i < 50 + 1; i++){
01352     if(mkdir("irefd") != 0){
01353       printf(1, "mkdir irefd failed\n");
01354       exit();
01355     }
01356     if(chdir("irefd") != 0){
01357       printf(1, "chdir irefd failed\n");
01358       exit();
01359     }
01360
01361     mkdir("");
01362     link("README", "");
01363     fd = open("", O_CREATE);
01364     if(fd >= 0)
01365       close(fd);
01366     fd = open("xx", O_CREATE);
01367     if(fd >= 0)
01368       close(fd);
01369     unlink("xx");
01370   }
01371
01372   chdir("/");
01373   printf(1, "empty file name OK\n");
01374 }
```

Referenced by main().

5.240.2.21 linktest()

```
void linktest (
              void  )
```

Definition at line 702 of file usertests.c.

```
00703 {
00704   int fd;
00705
00706   printf(1, "linktest\n");
00707
00708   unlink("lf1");
00709   unlink("lf2");
00710
00711   fd = open("lf1", O_CREATE|O_RDWR);
00712   if(fd < 0){
00713     printf(1, "create lf1 failed\n");
00714     exit();
00715   }
00716   if(write(fd, "hello", 5) != 5){
00717     printf(1, "write lf1 failed\n");
00718     exit();
00719   }
00720   close(fd);
00721
00722   if(link("lf1", "lf2") < 0){
00723     printf(1, "link lf1 lf2 failed\n");
00724     exit();
00725   }
00726   unlink("lf1");
00727
00728   if(open("lf1", 0) >= 0){
00729     printf(1, "unlinked lf1 but it is still there!\n");
00730     exit();
00731   }
00732
00733   fd = open("lf2", 0);
00734   if(fd < 0){
00735     printf(1, "open lf2 failed\n");
00736     exit();
00737   }
00738   if(read(fd, buf, sizeof(buf)) != 5){
00739     printf(1, "read lf2 failed\n");
00740     exit();
00741   }
00742   close(fd);
00743
00744   if(link("lf2", "lf2") >= 0){
00745     printf(1, "link lf2 lf2 succeeded! oops\n");
00746     exit();
00747   }
00748
00749   unlink("lf2");
00750   if(link("lf2", "lf1") >= 0){
00751     printf(1, "link non-existant succeeded! oops\n");
00752     exit();
00753   }
00754
00755   if(link(".", "lf1") >= 0){
00756     printf(1, "link . lf1 succeeded! oops\n");
00757     exit();
00758   }
00759
00760   printf(1, "linktest ok\n");
00761 }
```

Referenced by main().

5.240.2.22 linkunlink()

```
void linkunlink ( )
```

Definition at line 858 of file usertests.c.

```
00859 {
```

```
00860   int pid, i;
00861
00862   printf(1, "linkunlink test\n");
00863
00864   unlink("x");
00865   pid = fork();
00866   if(pid < 0){
00867     printf(1, "fork failed\n");
00868     exit();
00869   }
00870
00871   unsigned int x = (pid ? 1 : 97);
00872   for(i = 0; i < 100; i++){
00873     x = x * 1103515245 + 12345;
00874     if((x % 3) == 0){
00875       close(open("x", O_RDWR | O_CREATE));
00876     } else if((x % 3) == 1){
00877       link("cat", "x");
00878     } else {
00879       unlink("x");
00880     }
00881   }
00882
00883   if(pid)
00884     wait();
00885   else
00886     exit();
00887
00888   printf(1, "linkunlink ok\n");
00889 }
```

Referenced by main().

5.240.2.23 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 1749 of file usertests.c.

```
01750 {
01751   printf(1, "usertests starting\n");
01752
01753   if(open("usertests.ran", 0) >= 0){
01754     printf(1, "already ran user tests -- rebuild fs.img\n");
01755     exit();
01756   }
01757   close(open("usertests.ran", O_CREATE));
01758
01759   argptest();
01760   createdelete();
01761   linkunlink();
01762   concreate();
01763   fourfiles();
01764   sharedfd();
01765
01766   bigargtest();
01767   bigwrite();
01768   bigargtest();
01769   bsstest();
01770   sbrktest();
01771   validatetest();
01772
01773   opentest();
01774   writetest();
01775   writetest1();
01776   createtest();
01777
01778   openiputtest();
01779   exitiputtest();
01780   iputtest();
01781
01782   mem();
01783   pipe1();
01784   preempt();
01785   exitwait();
01786
```

```
01787   rmdot();
01788   fourteen();
01789   bigfile();
01790   subdir();
01791   linktest();
01792   unlinkread();
01793   dirfile();
01794   iref();
01795   forktest();
01796   bigdir(); // slow
01797
01798   uio();
01799
01800   exectest();
01801
01802   exit();
01803 }
```

5.240.2.24 mem()

```
void mem (
              void  )
```

Definition at line 428 of file usertests.c.

```
00429 {
00430   void *m1, *m2;
00431   int pid, ppid;
00432
00433   printf(1, "mem test\n");
00434   ppid = getpid();
00435   if((pid = fork()) == 0){
00436     m1 = 0;
00437     while((m2 = malloc(10001)) != 0){
00438       *(char**)m2 = m1;
00439       m1 = m2;
00440     }
00441     while(m1){
00442       m2 = *(char**)m1;
00443       free(m1);
00444       m1 = m2;
00445     }
00446     m1 = malloc(1024*20);
00447     if(m1 == 0){
00448       printf(1, "couldn't allocate mem?!!\n");
00449       kill(ppid);
00450       exit();
00451     }
00452     free(m1);
00453     printf(1, "mem ok\n");
00454     exit();
00455   } else {
00456     wait();
00457   }
00458 }
```

Referenced by allocuvm(), copyuvm(), inituvm(), and main().

5.240.2.25 openiputtest()

```
void openiputtest (
              void  )
```

Definition at line 85 of file usertests.c.

```
00086 {
00087   int pid;
00088
00089   printf(stdout, "openiput test\n");
00090   if(mkdir("oidir") < 0){
```

```
00091     printf(stdout, "mkdir oidir failed\n");
00092     exit();
00093   }
00094   pid = fork();
00095   if(pid < 0){
00096     printf(stdout, "fork failed\n");
00097     exit();
00098   }
00099   if(pid == 0){
00100     int fd = open("oidir", O_RDWR);
00101     if(fd >= 0){
00102       printf(stdout, "open directory for write succeeded\n");
00103       exit();
00104     }
00105     exit();
00106   }
00107   sleep(1);
00108   if(unlink("oidir") != 0){
00109     printf(stdout, "unlink failed\n");
00110     exit();
00111   }
00112   wait();
00113   printf(stdout, "openiput test ok\n");
00114 }
```

Referenced by main().

5.240.2.26 opentest()

```
void opentest (
              void  )
```

Definition at line 119 of file usertests.c.

```
00120 {
00121   int fd;
00122
00123   printf(stdout, "open test\n");
00124   fd = open("echo", 0);
00125   if(fd < 0){
00126     printf(stdout, "open echo failed!\n");
00127     exit();
00128   }
00129   close(fd);
00130   fd = open("doesnotexist", 0);
00131   if(fd >= 0){
00132     printf(stdout, "open doesnotexist succeeded!\n");
00133     exit();
00134   }
00135   printf(stdout, "open test ok\n");
00136 }
```

Referenced by main().

5.240.2.27 pipe1()

```
void pipe1 (
              void  )
```

Definition at line 306 of file usertests.c.

```
00307 {
00308   int fds[2], pid;
00309   int seq, i, n, cc, total;
00310
00311   if(pipe(fds) != 0){
00312     printf(1, "pipe() failed\n");
00313     exit();
00314   }
```

```
00315   pid = fork();
00316   seq = 0;
00317   if(pid == 0){
00318     close(fds[0]);
00319     for(n = 0; n < 5; n++){
00320       for(i = 0; i < 1033; i++)
00321         buf[i] = seq++;
00322       if(write(fds[1], buf, 1033) != 1033){
00323         printf(1, "pipe1 oops 1\n");
00324         exit();
00325       }
00326     }
00327     exit();
00328   } else if(pid > 0){
00329     close(fds[1]);
00330     total = 0;
00331     cc = 1;
00332     while((n = read(fds[0], buf, cc)) > 0){
00333       for(i = 0; i < n; i++){
00334         if((buf[i] & 0xff) != (seq++ & 0xff)){
00335           printf(1, "pipe1 oops 2\n");
00336           return;
00337         }
00338       }
00339       total += n;
00340       cc = cc * 2;
00341       if(cc > sizeof(buf))
00342         cc = sizeof(buf);
00343     }
00344     if(total != 5 * 1033){
00345       printf(1, "pipe1 oops 3 total %d\n", total);
00346       exit();
00347     }
00348     close(fds[0]);
00349     wait();
00350   } else {
00351     printf(1, "fork() failed\n");
00352     exit();
00353   }
00354   printf(1, "pipe1 ok\n");
00355 }
```

Referenced by main().

5.240.2.28 preempt()

```
void preempt (
              void  )
```

Definition at line 359 of file usertests.c.

```
00360 {
00361   int pid1, pid2, pid3;
00362   int pfds[2];
00363
00364   printf(1, "preempt: ");
00365   pid1 = fork();
00366   if(pid1 == 0)
00367     for(;;)
00368       ;
00369
00370   pid2 = fork();
00371   if(pid2 == 0)
00372     for(;;)
00373       ;
00374
00375   pipe(pfds);
00376   pid3 = fork();
00377   if(pid3 == 0){
00378     close(pfds[0]);
00379     if(write(pfds[1], "x", 1) != 1)
00380       printf(1, "preempt write error");
00381     close(pfds[1]);
00382     for(;;)
00383       ;
00384   }
00385
00386   close(pfds[1]);
```

```
00387   if(read(pfds[0], buf, sizeof(buf)) != 1){
00388     printf(1, "preempt read error");
00389     return;
00390   }
00391   close(pfds[0]);
00392   printf(1, "kill... ");
00393   kill(pid1);
00394   kill(pid2);
00395   kill(pid3);
00396   printf(1, "wait... ");
00397   wait();
00398   wait();
00399   wait();
00400   printf(1, "preempt ok\n");
00401 }
```

Referenced by main().

5.240.2.29 rand()

```
unsigned int rand ( )
```

Definition at line 1742 of file usertests.c.

```
01743 {
01744   randstate = randstate * 1664525 + 1013904223;
01745   return randstate;
01746 }
```

5.240.2.30 rmdot()

```
void rmdot (
              void  )
```

Definition at line 1245 of file usertests.c.

```
01246 {
01247   printf(1, "rmdot test\n");
01248   if(mkdir("dots") != 0){
01249     printf(1, "mkdir dots failed\n");
01250     exit();
01251   }
01252   if(chdir("dots") != 0){
01253     printf(1, "chdir dots failed\n");
01254     exit();
01255   }
01256   if(unlink(".") == 0){
01257     printf(1, "rm . worked!\n");
01258     exit();
01259   }
01260   if(unlink("..") == 0){
01261     printf(1, "rm .. worked!\n");
01262     exit();
01263   }
01264   if(chdir("/") != 0){
01265     printf(1, "chdir / failed\n");
01266     exit();
01267   }
01268   if(unlink("dots/.") == 0){
01269     printf(1, "unlink dots/. worked!\n");
01270     exit();
01271   }
01272   if(unlink("dots/..") == 0){
01273     printf(1, "unlink dots/.. worked!\n");
01274     exit();
01275   }
01276   if(unlink("dots") != 0){
01277     printf(1, "unlink dots failed!\n");
01278     exit();
01279   }
01280   printf(1, "rmdot ok\n");
01281 }
```

Referenced by main().

5.240.2.31 sbrktest()

```
void sbrktest (
            void  )
```

Definition at line 1415 of file usertests.c.

```
01416 {
01417   int fds[2], pid, pids[10], ppid;
01418   char *a, *b, *c, *lastaddr, *oldbrk, *p, scratch;
01419   uint amt;
01420
01421   printf(stdout, "sbrk test\n");
01422   oldbrk = sbrk(0);
01423
01424   // can one sbrk() less than a page?
01425   a = sbrk(0);
01426   int i;
01427   for(i = 0; i < 5000; i++){
01428     b = sbrk(1);
01429     if(b != a){
01430       printf(stdout, "sbrk test failed %d %x %x\n", i, a, b);
01431       exit();
01432     }
01433     *b = 1;
01434     a = b + 1;
01435   }
01436   pid = fork();
01437   if(pid < 0){
01438     printf(stdout, "sbrk test fork failed\n");
01439     exit();
01440   }
01441   c = sbrk(1);
01442   c = sbrk(1);
01443   if(c != a + 1){
01444     printf(stdout, "sbrk test failed post-fork\n");
01445     exit();
01446   }
01447   if(pid == 0)
01448     exit();
01449   wait();
01450
01451   // can one grow address space to something big?
01452 #define BIG (100*1024*1024)
01453   a = sbrk(0);
01454   amt = (BIG) - (uint)a;
01455   p = sbrk(amt);
01456   if (p != a) {
01457     printf(stdout, "sbrk test failed to grow big address space; enough phys mem?\n");
01458     exit();
01459   }
01460   lastaddr = (char*) (BIG-1);
01461   *lastaddr = 99;
01462
01463   // can one de-allocate?
01464   a = sbrk(0);
01465   c = sbrk(-4096);
01466   if(c == (char*)0xffffffff){
01467     printf(stdout, "sbrk could not deallocate\n");
01468     exit();
01469   }
01470   c = sbrk(0);
01471   if(c != a - 4096){
01472     printf(stdout, "sbrk deallocation produced wrong address, a %x c %x\n", a, c);
01473     exit();
01474   }
01475
01476   // can one re-allocate that page?
01477   a = sbrk(0);
01478   c = sbrk(4096);
01479   if(c != a || sbrk(0) != a + 4096){
01480     printf(stdout, "sbrk re-allocation failed, a %x c %x\n", a, c);
01481     exit();
01482   }
01483   if(*lastaddr == 99){
01484     // should be zero
01485     printf(stdout, "sbrk de-allocation didn't really deallocate\n");
01486     exit();
01487   }
01488
01489   a = sbrk(0);
01490   c = sbrk(-(sbrk(0) - oldbrk));
01491   if(c != a){
01492     printf(stdout, "sbrk downsize failed, a %x c %x\n", a, c);
01493     exit();
```

```
01494   }
01495
01496   // can we read the kernel's memory?
01497   for(a = (char*)(KERNBASE); a < (char*) (KERNBASE+2000000); a += 50000){
01498     ppid = getpid();
01499     pid = fork();
01500     if(pid < 0){
01501       printf(stdout, "fork failed\n");
01502       exit();
01503     }
01504     if(pid == 0){
01505       printf(stdout, "oops could read %x = %x\n", a, *a);
01506       kill(ppid);
01507       exit();
01508     }
01509     wait();
01510   }
01511
01512   // if we run the system out of memory, does it clean up the last
01513   // failed allocation?
01514   if(pipe(fds) != 0){
01515     printf(1, "pipe() failed\n");
01516     exit();
01517   }
01518   for(i = 0; i < sizeof(pids)/sizeof(pids[0]); i++){
01519     if((pids[i] = fork()) == 0){
01520       // allocate a lot of memory
01521       sbrk(BIG - (uint)sbrk(0));
01522       write(fds[1], "x", 1);
01523       // sit around until killed
01524       for(;;) sleep(1000);
01525     }
01526     if(pids[i] != -1)
01527       read(fds[0], &scratch, 1);
01528   }
01529   // if those failed allocations freed up the pages they did allocate,
01530   // we'll be able to allocate here
01531   c = sbrk(4096);
01532   for(i = 0; i < sizeof(pids)/sizeof(pids[0]); i++){
01533     if(pids[i] == -1)
01534       continue;
01535     kill(pids[i]);
01536     wait();
01537   }
01538   if(c == (char*)0xffffffff){
01539     printf(stdout, "failed sbrk leaked memory\n");
01540     exit();
01541   }
01542
01543   if(sbrk(0) > oldbrk)
01544     sbrk(-(sbrk(0) - oldbrk));
01545
01546   printf(stdout, "sbrk test OK\n");
01547 }
```

Referenced by main().

5.240.2.32 sharedfd()

```
void sharedfd (
              void  )
```

Definition at line 465 of file usertests.c.

```
00466 {
00467   int fd, pid, i, n, nc, np;
00468   char buf[10];
00469
00470   printf(1, "sharedfd test\n");
00471
00472   unlink("sharedfd");
00473   fd = open("sharedfd", O_CREATE|O_RDWR);
00474   if(fd < 0){
00475     printf(1, "fstests: cannot open sharedfd for writing");
00476     return;
00477   }
00478   pid = fork();
00479   memset(buf, pid==0?'c':'p', sizeof(buf));
```

```
00480   for(i = 0; i < 1000; i++){
00481     if(write(fd, buf, sizeof(buf)) != sizeof(buf)){
00482       printf(1, "fstests: write sharedfd failed\n");
00483       break;
00484     }
00485   }
00486   if(pid == 0)
00487     exit();
00488   else
00489     wait();
00490   close(fd);
00491   fd = open("sharedfd", 0);
00492   if(fd < 0){
00493     printf(1, "fstests: cannot open sharedfd for reading\n");
00494     return;
00495   }
00496   nc = np = 0;
00497   while((n = read(fd, buf, sizeof(buf))) > 0){
00498     for(i = 0; i < sizeof(buf); i++){
00499       if(buf[i] == 'c')
00500         nc++;
00501       if(buf[i] == 'p')
00502         np++;
00503     }
00504   }
00505   close(fd);
00506   unlink("sharedfd");
00507   if(nc == 10000 && np == 10000){
00508     printf(1, "sharedfd ok\n");
00509   } else {
00510     printf(1, "sharedfd oops %d %d\n", nc, np);
00511     exit();
00512   }
00513 }
```

Referenced by main().

5.240.2.33 subdir()

```
void subdir (
             void  )
```

Definition at line 935 of file usertests.c.

```
00936 {
00937   int fd, cc;
00938
00939   printf(1, "subdir test\n");
00940
00941   unlink("ff");
00942   if(mkdir("dd") != 0){
00943     printf(1, "subdir mkdir dd failed\n");
00944     exit();
00945   }
00946
00947   fd = open("dd/ff", O_CREATE | O_RDWR);
00948   if(fd < 0){
00949     printf(1, "create dd/ff failed\n");
00950     exit();
00951   }
00952   write(fd, "ff", 2);
00953   close(fd);
00954
00955   if(unlink("dd") >= 0){
00956     printf(1, "unlink dd (non-empty dir) succeeded!\n");
00957     exit();
00958   }
00959
00960   if(mkdir("/dd/dd") != 0){
00961     printf(1, "subdir mkdir dd/dd failed\n");
00962     exit();
00963   }
00964
00965   fd = open("dd/dd/ff", O_CREATE | O_RDWR);
00966   if(fd < 0){
00967     printf(1, "create dd/dd/ff failed\n");
00968     exit();
00969   }
```

```
00970   write(fd, "FF", 2);
00971   close(fd);
00972
00973   fd = open("dd/dd/../ff", 0);
00974   if(fd < 0){
00975     printf(1, "open dd/dd/../ff failed\n");
00976     exit();
00977   }
00978   cc = read(fd, buf, sizeof(buf));
00979   if(cc != 2 || buf[0] != 'f'){
00980     printf(1, "dd/dd/../ff wrong content\n");
00981     exit();
00982   }
00983   close(fd);
00984
00985   if(link("dd/dd/ff", "dd/dd/ffff") != 0){
00986     printf(1, "link dd/dd/ff dd/dd/ffff failed\n");
00987     exit();
00988   }
00989
00990   if(unlink("dd/dd/ff") != 0){
00991     printf(1, "unlink dd/dd/ff failed\n");
00992     exit();
00993   }
00994   if(open("dd/dd/ff", O_RDONLY) >= 0){
00995     printf(1, "open (unlinked) dd/dd/ff succeeded\n");
00996     exit();
00997   }
00998
00999   if(chdir("dd") != 0){
01000     printf(1, "chdir dd failed\n");
01001     exit();
01002   }
01003   if(chdir("dd/../../dd") != 0){
01004     printf(1, "chdir dd/../../dd failed\n");
01005     exit();
01006   }
01007   if(chdir("dd/../../../dd") != 0){
01008     printf(1, "chdir dd/../../dd failed\n");
01009     exit();
01010   }
01011   if(chdir("./..") != 0){
01012     printf(1, "chdir ./.. failed\n");
01013     exit();
01014   }
01015
01016   fd = open("dd/dd/ffff", 0);
01017   if(fd < 0){
01018     printf(1, "open dd/dd/ffff failed\n");
01019     exit();
01020   }
01021   if(read(fd, buf, sizeof(buf)) != 2){
01022     printf(1, "read dd/dd/ffff wrong len\n");
01023     exit();
01024   }
01025   close(fd);
01026
01027   if(open("dd/dd/ff", O_RDONLY) >= 0){
01028     printf(1, "open (unlinked) dd/dd/ff succeeded!\n");
01029     exit();
01030   }
01031
01032   if(open("dd/ff/ff", O_CREATE|O_RDWR) >= 0){
01033     printf(1, "create dd/ff/ff succeeded!\n");
01034     exit();
01035   }
01036   if(open("dd/xx/ff", O_CREATE|O_RDWR) >= 0){
01037     printf(1, "create dd/xx/ff succeeded!\n");
01038     exit();
01039   }
01040   if(open("dd", O_CREATE) >= 0){
01041     printf(1, "create dd succeeded!\n");
01042     exit();
01043   }
01044   if(open("dd", O_RDWR) >= 0){
01045     printf(1, "open dd rdwr succeeded!\n");
01046     exit();
01047   }
01048   if(open("dd", O_WRONLY) >= 0){
01049     printf(1, "open dd wronly succeeded!\n");
01050     exit();
01051   }
01052   if(link("dd/ff/ff", "dd/dd/xx") == 0){
01053     printf(1, "link dd/ff/ff dd/dd/xx succeeded!\n");
01054     exit();
01055   }
01056   if(link("dd/xx/ff", "dd/dd/xx") == 0){
```

```
01057     printf(1, "link dd/xx/ff dd/dd/xx succeeded!\n");
01058     exit();
01059   }
01060   if(link("dd/ff", "dd/dd/ffff") == 0){
01061     printf(1, "link dd/ff dd/dd/ffff succeeded!\n");
01062     exit();
01063   }
01064   if(mkdir("dd/ff/ff") == 0){
01065     printf(1, "mkdir dd/ff/ff succeeded!\n");
01066     exit();
01067   }
01068   if(mkdir("dd/xx/ff") == 0){
01069     printf(1, "mkdir dd/xx/ff succeeded!\n");
01070     exit();
01071   }
01072   if(mkdir("dd/dd/ffff") == 0){
01073     printf(1, "mkdir dd/dd/ffff succeeded!\n");
01074     exit();
01075   }
01076   if(unlink("dd/xx/ff") == 0){
01077     printf(1, "unlink dd/xx/ff succeeded!\n");
01078     exit();
01079   }
01080   if(unlink("dd/ff/ff") == 0){
01081     printf(1, "unlink dd/ff/ff succeeded!\n");
01082     exit();
01083   }
01084   if(chdir("dd/ff") == 0){
01085     printf(1, "chdir dd/ff succeeded!\n");
01086     exit();
01087   }
01088   if(chdir("dd/xx") == 0){
01089     printf(1, "chdir dd/xx succeeded!\n");
01090     exit();
01091   }
01092
01093   if(unlink("dd/dd/ffff") != 0){
01094     printf(1, "unlink dd/dd/ff failed\n");
01095     exit();
01096   }
01097   if(unlink("dd/ff") != 0){
01098     printf(1, "unlink dd/ff failed\n");
01099     exit();
01100   }
01101   if(unlink("dd") == 0){
01102     printf(1, "unlink non-empty dd succeeded!\n");
01103     exit();
01104   }
01105   if(unlink("dd/dd") < 0){
01106     printf(1, "unlink dd/dd failed\n");
01107     exit();
01108   }
01109   if(unlink("dd") < 0){
01110     printf(1, "unlink dd failed\n");
01111     exit();
01112   }
01113
01114   printf(1, "subdir ok\n");
01115 }
```

Referenced by main().

5.240.2.34 uio()

```
void uio ( )
```

Definition at line 1699 of file usertests.c.

```
01700 {
01701   #define RTC_ADDR 0x70
01702   #define RTC_DATA 0x71
01703
01704   ushort port = 0;
01705   uchar val = 0;
01706   int pid;
01707
01708   printf(1, "uio test\n");
01709   pid = fork();
01710   if(pid == 0){
```

```
01711     port = RTC_ADDR;
01712     val = 0x09;  /* year */
01713     /* http://wiki.osdev.org/Inline_Assembly/Examples */
01714     asm volatile("outb %0,%1"::"a"(val), "d" (port));
01715     port = RTC_DATA;
01716     asm volatile("inb %1,%0" : "=a" (val) : "d" (port));
01717     printf(1, "uio: uio succeeded; test FAILED\n");
01718     exit();
01719   } else if(pid < 0){
01720     printf (1, "fork failed\n");
01721     exit();
01722   }
01723   wait();
01724   printf(1, "uio test done\n");
01725 }
```

Referenced by main().

5.240.2.35 unlinkread()

```
void unlinkread (
              void  )
```

Definition at line 657 of file usertests.c.

```
00658 {
00659   int fd, fd1;
00660
00661   printf(1, "unlinkread test\n");
00662   fd = open("unlinkread", O_CREATE | O_RDWR);
00663   if(fd < 0){
00664     printf(1, "create unlinkread failed\n");
00665     exit();
00666   }
00667   write(fd, "hello", 5);
00668   close(fd);
00669
00670   fd = open("unlinkread", O_RDWR);
00671   if(fd < 0){
00672     printf(1, "open unlinkread failed\n");
00673     exit();
00674   }
00675   if(unlink("unlinkread") != 0){
00676     printf(1, "unlink unlinkread failed\n");
00677     exit();
00678   }
00679
00680   fd1 = open("unlinkread", O_CREATE | O_RDWR);
00681   write(fd1, "yyy", 3);
00682   close(fd1);
00683
00684   if(read(fd, buf, sizeof(buf)) != 5){
00685     printf(1, "unlinkread read failed");
00686     exit();
00687   }
00688   if(buf[0] != 'h'){
00689     printf(1, "unlinkread wrong data\n");
00690     exit();
00691   }
00692   if(write(fd, buf, 10) != 10){
00693     printf(1, "unlinkread write failed\n");
00694     exit();
00695   }
00696   close(fd);
00697   unlink("unlinkread");
00698   printf(1, "unlinkread ok\n");
00699 }
```

Referenced by main().

5.240.2.36 validateint()

```
void validateint (
             int * p )
```

Definition at line 1550 of file usertests.c.

```
01551 {
01552   int res;
01553   asm("mov %%esp, %%ebx\n\t"
01554       "mov %3, %%esp\n\t"
01555       "int %2\n\t"
01556       "mov %%ebx, %%esp" :
01557       "=a" (res) :
01558       "a" (SYS_sleep), "n" (T_SYSCALL), "c" (p) :
01559       "ebx");
01560 }
```

Referenced by validatetest().

5.240.2.37 validatetest()

```
void validatetest (
             void  )
```

Definition at line 1563 of file usertests.c.

```
01564 {
01565   int hi, pid;
01566   uint p;
01567
01568   printf(stdout, "validate test\n");
01569   hi = 1100*1024;
01570
01571   for(p = 0; p <= (uint)hi; p += 4096){
01572     if((pid = fork()) == 0){
01573       // try to crash the kernel by passing in a badly placed integer
01574       validateint((int*)p);
01575       exit();
01576     }
01577     sleep(0);
01578     sleep(0);
01579     kill(pid);
01580     wait();
01581
01582     // try to crash the kernel by passing in a bad string pointer
01583     if(link("nosuchfile", (char*)p) != -1){
01584       printf(stdout, "link should not succeed\n");
01585       exit();
01586     }
01587   }
01588
01589   printf(stdout, "validate ok\n");
01590 }
```

Referenced by main().

5.240.2.38 writetest()

```
void writetest (
            void  )
```

Definition at line 139 of file usertests.c.

```
00140 {
00141   int fd;
00142   int i;
00143
00144   printf(stdout, "small file test\n");
00145   fd = open("small", O_CREATE|O_RDWR);
00146   if(fd >= 0){
00147     printf(stdout, "creat small succeeded; ok\n");
00148   } else {
00149     printf(stdout, "error: creat small failed!\n");
00150     exit();
00151   }
00152   for(i = 0; i < 100; i++){
00153     if(write(fd, "aaaaaaaaaa", 10) != 10){
00154       printf(stdout, "error: write aa %d new file failed\n", i);
00155       exit();
00156     }
00157     if(write(fd, "bbbbbbbbbb", 10) != 10){
00158       printf(stdout, "error: write bb %d new file failed\n", i);
00159       exit();
00160     }
00161   }
00162   printf(stdout, "writes ok\n");
00163   close(fd);
00164   fd = open("small", O_RDONLY);
00165   if(fd >= 0){
00166     printf(stdout, "open small succeeded ok\n");
00167   } else {
00168     printf(stdout, "error: open small failed!\n");
00169     exit();
00170   }
00171   i = read(fd, buf, 2000);
00172   if(i == 2000){
00173     printf(stdout, "read succeeded ok\n");
00174   } else {
00175     printf(stdout, "read failed\n");
00176     exit();
00177   }
00178   close(fd);
00179
00180   if(unlink("small") < 0){
00181     printf(stdout, "unlink small failed\n");
00182     exit();
00183   }
00184   printf(stdout, "small file test ok\n");
00185 }
```

Referenced by main().

5.240.2.39 writetest1()

```
void writetest1 (
            void  )
```

Definition at line 188 of file usertests.c.

```
00189 {
00190   int i, fd, n;
00191
00192   printf(stdout, "big files test\n");
00193
00194   fd = open("big", O_CREATE|O_RDWR);
00195   if(fd < 0){
00196     printf(stdout, "error: creat big failed!\n");
00197     exit();
00198   }
00199
00200   for(i = 0; i < MAXFILE; i++){
00201     ((int*)buf)[0] = i;
```

```
00202     if(write(fd, buf, 512) != 512){
00203       printf(stdout, "error: write big file failed\n", i);
00204       exit();
00205     }
00206   }
00207
00208   close(fd);
00209
00210   fd = open("big", O_RDONLY);
00211   if(fd < 0){
00212     printf(stdout, "error: open big failed!\n");
00213     exit();
00214   }
00215
00216   n = 0;
00217   for(;;){
00218     i = read(fd, buf, 512);
00219     if(i == 0){
00220       if(n == MAXFILE - 1){
00221         printf(stdout, "read only %d blocks from big", n);
00222         exit();
00223       }
00224       break;
00225     } else if(i != 512){
00226       printf(stdout, "read failed %d\n", i);
00227       exit();
00228     }
00229     if(((int*)buf)[0] != n){
00230       printf(stdout, "read content of block %d is %d\n",
00231              n, ((int*)buf)[0]);
00232       exit();
00233     }
00234     n++;
00235   }
00236   close(fd);
00237   if(unlink("big") < 0){
00238     printf(stdout, "unlink big failed\n");
00239     exit();
00240   }
00241   printf(stdout, "big files ok\n");
00242 }
```

Referenced by main().

5.240.3 Variable Documentation

5.240.3.1 buf

```
char buf[8192]
```

Definition at line 11 of file usertests.c.

Referenced by pipe1(), and sharedfd().

5.240.3.2 echoargv

```
char* echoargv[] = { "echo", "ALL", "TESTS", "PASSED", 0 }
```

Definition at line 13 of file usertests.c.

Referenced by exectest().

5.240.3.3 name

```
char name[3]
```

Definition at line 12 of file usertests.c.

Referenced by bigdir(), concreate(), create(), createdelete(), createtest(), dirlink(), dirlookup(), fsfull(), initlock(), initsleeplock(), main(), namei(), nameiparent(), namex(), skipelem(), sys_getprocs(), sys_link(), sys_unlink(), trap(), and wc().

5.240.3.4 randstate

```
unsigned long randstate = 1
```

Definition at line 1740 of file usertests.c.

Referenced by rand().

5.240.3.5 stdout

```
int stdout = 1
```

Definition at line 14 of file usertests.c.

Referenced by bigargtest(), bsstest(), createtest(), dirtest(), exectest(), exitiputtest(), iputtest(), openiputtest(), opentest(), sbrktest(), validatetest(), writetest(), and writetest1().

5.240.3.6 uninit

```
char uninit[10000]
```

Definition at line 1593 of file usertests.c.

Referenced by bsstest().

5.241 usertests.c

Go to the documentation of this file.

```
00001 #include "param.h"
00002 #include "types.h"
00003 #include "stat.h"
00004 #include "user.h"
00005 #include "fs.h"
00006 #include "fcntl.h"
00007 #include "syscall.h"
00008 #include "traps.h"
00009 #include "memlayout.h"
00010
00011 char buf[8192];
00012 char name[3];
00013 char *echoargv[] = { "echo", "ALL", "TESTS", "PASSED", 0 };
00014 int stdout = 1;
00015
00016 // does chdir() call iput(p->cwd) in a transaction?
00017 void
00018 iputtest(void)
00019 {
00020   printf(stdout, "iput test\n");
00021
00022   if(mkdir("iputdir") < 0){
00023     printf(stdout, "mkdir failed\n");
00024     exit();
00025   }
00026   if(chdir("iputdir") < 0){
00027     printf(stdout, "chdir iputdir failed\n");
00028     exit();
00029   }
00030   if(unlink("../iputdir") < 0){
00031     printf(stdout, "unlink ../iputdir failed\n");
00032     exit();
00033   }
00034   if(chdir("/") < 0){
00035     printf(stdout, "chdir / failed\n");
00036     exit();
00037   }
00038   printf(stdout, "iput test ok\n");
00039 }
00040
00041 // does exit() call iput(p->cwd) in a transaction?
00042 void
00043 exitiputtest(void)
00044 {
00045   int pid;
00046
00047   printf(stdout, "exitiput test\n");
00048
00049   pid = fork();
00050   if(pid < 0){
00051     printf(stdout, "fork failed\n");
00052     exit();
00053   }
00054   if(pid == 0){
00055     if(mkdir("iputdir") < 0){
00056       printf(stdout, "mkdir failed\n");
00057       exit();
00058     }
00059     if(chdir("iputdir") < 0){
00060       printf(stdout, "child chdir failed\n");
00061       exit();
00062     }
00063     if(unlink("../iputdir") < 0){
00064       printf(stdout, "unlink ../iputdir failed\n");
00065       exit();
00066     }
00067     exit();
00068   }
00069   wait();
00070   printf(stdout, "exitiput test ok\n");
00071 }
00072
00073 // does the error path in open() for attempt to write a
00074 // directory call iput() in a transaction?
00075 // needs a hacked kernel that pauses just after the namei()
00076 // call in sys_open():
00077 //    if((ip = namei(path)) == 0)
00078 //      return -1;
00079 //    {
00080 //      int i;
00081 //      for(i = 0; i < 10000; i++)
00082 //        yield();
```

```
00083 //    }
00084 void
00085 openiputtest(void)
00086 {
00087   int pid;
00088
00089   printf(stdout, "openiput test\n");
00090   if(mkdir("oidir") < 0){
00091     printf(stdout, "mkdir oidir failed\n");
00092     exit();
00093   }
00094   pid = fork();
00095   if(pid < 0){
00096     printf(stdout, "fork failed\n");
00097     exit();
00098   }
00099   if(pid == 0){
00100     int fd = open("oidir", O_RDWR);
00101     if(fd >= 0){
00102       printf(stdout, "open directory for write succeeded\n");
00103       exit();
00104     }
00105     exit();
00106   }
00107   sleep(1);
00108   if(unlink("oidir") != 0){
00109     printf(stdout, "unlink failed\n");
00110     exit();
00111   }
00112   wait();
00113   printf(stdout, "openiput test ok\n");
00114 }
00115
00116 // simple file system tests
00117
00118 void
00119 opentest(void)
00120 {
00121   int fd;
00122
00123   printf(stdout, "open test\n");
00124   fd = open("echo", 0);
00125   if(fd < 0){
00126     printf(stdout, "open echo failed!\n");
00127     exit();
00128   }
00129   close(fd);
00130   fd = open("doesnotexist", 0);
00131   if(fd >= 0){
00132     printf(stdout, "open doesnotexist succeeded!\n");
00133     exit();
00134   }
00135   printf(stdout, "open test ok\n");
00136 }
00137
00138 void
00139 writetest(void)
00140 {
00141   int fd;
00142   int i;
00143
00144   printf(stdout, "small file test\n");
00145   fd = open("small", O_CREATE|O_RDWR);
00146   if(fd >= 0){
00147     printf(stdout, "creat small succeeded; ok\n");
00148   } else {
00149     printf(stdout, "error: creat small failed!\n");
00150     exit();
00151   }
00152   for(i = 0; i < 100; i++){
00153     if(write(fd, "aaaaaaaaaa", 10) != 10){
00154       printf(stdout, "error: write aa %d new file failed\n", i);
00155       exit();
00156     }
00157     if(write(fd, "bbbbbbbbbb", 10) != 10){
00158       printf(stdout, "error: write bb %d new file failed\n", i);
00159       exit();
00160     }
00161   }
00162   printf(stdout, "writes ok\n");
00163   close(fd);
00164   fd = open("small", O_RDONLY);
00165   if(fd >= 0){
00166     printf(stdout, "open small succeeded ok\n");
00167   } else {
00168     printf(stdout, "error: open small failed!\n");
00169     exit();
```

```
00170   }
00171   i = read(fd, buf, 2000);
00172   if(i == 2000){
00173     printf(stdout, "read succeeded ok\n");
00174   } else {
00175     printf(stdout, "read failed\n");
00176     exit();
00177   }
00178   close(fd);
00179
00180   if(unlink("small") < 0){
00181     printf(stdout, "unlink small failed\n");
00182     exit();
00183   }
00184   printf(stdout, "small file test ok\n");
00185 }
00186
00187 void
00188 writetest1(void)
00189 {
00190   int i, fd, n;
00191
00192   printf(stdout, "big files test\n");
00193
00194   fd = open("big", O_CREATE|O_RDWR);
00195   if(fd < 0){
00196     printf(stdout, "error: creat big failed!\n");
00197     exit();
00198   }
00199
00200   for(i = 0; i < MAXFILE; i++){
00201     ((int*)buf)[0] = i;
00202     if(write(fd, buf, 512) != 512){
00203       printf(stdout, "error: write big file failed\n", i);
00204       exit();
00205     }
00206   }
00207
00208   close(fd);
00209
00210   fd = open("big", O_RDONLY);
00211   if(fd < 0){
00212     printf(stdout, "error: open big failed!\n");
00213     exit();
00214   }
00215
00216   n = 0;
00217   for(;;){
00218     i = read(fd, buf, 512);
00219     if(i == 0){
00220       if(n == MAXFILE - 1){
00221         printf(stdout, "read only %d blocks from big", n);
00222         exit();
00223       }
00224       break;
00225     } else if(i != 512){
00226       printf(stdout, "read failed %d\n", i);
00227       exit();
00228     }
00229     if(((int*)buf)[0] != n){
00230       printf(stdout, "read content of block %d is %d\n",
00231              n, ((int*)buf)[0]);
00232       exit();
00233     }
00234     n++;
00235   }
00236   close(fd);
00237   if(unlink("big") < 0){
00238     printf(stdout, "unlink big failed\n");
00239     exit();
00240   }
00241   printf(stdout, "big files ok\n");
00242 }
00243
00244 void
00245 createtest(void)
00246 {
00247   int i, fd;
00248
00249   printf(stdout, "many creates, followed by unlink test\n");
00250
00251   name[0] = 'a';
00252   name[2] = '\0';
00253   for(i = 0; i < 52; i++){
00254     name[1] = '0' + i;
00255     fd = open(name, O_CREATE|O_RDWR);
00256     close(fd);
```

```
00257   }
00258   name[0] = 'a';
00259   name[2] = '\0';
00260   for(i = 0; i < 52; i++){
00261     name[1] = '0' + i;
00262     unlink(name);
00263   }
00264   printf(stdout, "many creates, followed by unlink; ok\n");
00265 }
00266
00267 void dirtest(void)
00268 {
00269   printf(stdout, "mkdir test\n");
00270
00271   if(mkdir("dir0") < 0){
00272     printf(stdout, "mkdir failed\n");
00273     exit();
00274   }
00275
00276   if(chdir("dir0") < 0){
00277     printf(stdout, "chdir dir0 failed\n");
00278     exit();
00279   }
00280
00281   if(chdir("..") < 0){
00282     printf(stdout, "chdir .. failed\n");
00283     exit();
00284   }
00285
00286   if(unlink("dir0") < 0){
00287     printf(stdout, "unlink dir0 failed\n");
00288     exit();
00289   }
00290   printf(stdout, "mkdir test ok\n");
00291 }
00292
00293 void
00294 exectest(void)
00295 {
00296   printf(stdout, "exec test\n");
00297   if(exec("echo", echoargv) < 0){
00298     printf(stdout, "exec echo failed\n");
00299     exit();
00300   }
00301 }
00302
00303 // simple fork and pipe read/write
00304
00305 void
00306 pipe1(void)
00307 {
00308   int fds[2], pid;
00309   int seq, i, n, cc, total;
00310
00311   if(pipe(fds) != 0){
00312     printf(1, "pipe() failed\n");
00313     exit();
00314   }
00315   pid = fork();
00316   seq = 0;
00317   if(pid == 0){
00318     close(fds[0]);
00319     for(n = 0; n < 5; n++){
00320       for(i = 0; i < 1033; i++)
00321         buf[i] = seq++;
00322       if(write(fds[1], buf, 1033) != 1033){
00323         printf(1, "pipe1 oops 1\n");
00324         exit();
00325       }
00326     }
00327     exit();
00328   } else if(pid > 0){
00329     close(fds[1]);
00330     total = 0;
00331     cc = 1;
00332     while((n = read(fds[0], buf, cc)) > 0){
00333       for(i = 0; i < n; i++){
00334         if((buf[i] & 0xff) != (seq++ & 0xff)){
00335           printf(1, "pipe1 oops 2\n");
00336           return;
00337         }
00338       }
00339       total += n;
00340       cc = cc * 2;
00341       if(cc > sizeof(buf))
00342         cc = sizeof(buf);
00343     }
```

```
00344     if(total != 5 * 1033){
00345       printf(1, "pipe1 oops 3 total %d\n", total);
00346       exit();
00347     }
00348     close(fds[0]);
00349     wait();
00350   } else {
00351     printf(1, "fork() failed\n");
00352     exit();
00353   }
00354   printf(1, "pipe1 ok\n");
00355 }
00356
00357 // meant to be run w/ at most two CPUs
00358 void
00359 preempt(void)
00360 {
00361   int pid1, pid2, pid3;
00362   int pfds[2];
00363
00364   printf(1, "preempt: ");
00365   pid1 = fork();
00366   if(pid1 == 0)
00367     for(;;)
00368       ;
00369
00370   pid2 = fork();
00371   if(pid2 == 0)
00372     for(;;)
00373       ;
00374
00375   pipe(pfds);
00376   pid3 = fork();
00377   if(pid3 == 0){
00378     close(pfds[0]);
00379     if(write(pfds[1], "x", 1) != 1)
00380       printf(1, "preempt write error");
00381     close(pfds[1]);
00382     for(;;)
00383       ;
00384   }
00385
00386   close(pfds[1]);
00387   if(read(pfds[0], buf, sizeof(buf)) != 1){
00388     printf(1, "preempt read error");
00389     return;
00390   }
00391   close(pfds[0]);
00392   printf(1, "kill... ");
00393   kill(pid1);
00394   kill(pid2);
00395   kill(pid3);
00396   printf(1, "wait... ");
00397   wait();
00398   wait();
00399   wait();
00400   printf(1, "preempt ok\n");
00401 }
00402
00403 // try to find any races between exit and wait
00404 void
00405 exitwait(void)
00406 {
00407   int i, pid;
00408
00409   for(i = 0; i < 100; i++){
00410     pid = fork();
00411     if(pid < 0){
00412       printf(1, "fork failed\n");
00413       return;
00414     }
00415     if(pid){
00416       if(wait() != pid){
00417         printf(1, "wait wrong pid\n");
00418         return;
00419       }
00420     } else {
00421       exit();
00422     }
00423   }
00424   printf(1, "exitwait ok\n");
00425 }
00426
00427 void
00428 mem(void)
00429 {
00430   void *m1, *m2;
```

```
00431   int pid, ppid;
00432
00433   printf(1, "mem test\n");
00434   ppid = getpid();
00435   if((pid = fork()) == 0){
00436     m1 = 0;
00437     while((m2 = malloc(10001)) != 0){
00438       *(char**)m2 = m1;
00439       m1 = m2;
00440     }
00441     while(m1){
00442       m2 = *(char**)m1;
00443       free(m1);
00444       m1 = m2;
00445     }
00446     m1 = malloc(1024*20);
00447     if(m1 == 0){
00448       printf(1, "couldn't allocate mem?!!\n");
00449       kill(ppid);
00450       exit();
00451     }
00452     free(m1);
00453     printf(1, "mem ok\n");
00454     exit();
00455   } else {
00456     wait();
00457   }
00458 }
00459
00460 // More file system tests
00461
00462 // two processes write to the same file descriptor
00463 // is the offset shared? does inode locking work?
00464 void
00465 sharedfd(void)
00466 {
00467   int fd, pid, i, n, nc, np;
00468   char buf[10];
00469
00470   printf(1, "sharedfd test\n");
00471
00472   unlink("sharedfd");
00473   fd = open("sharedfd", O_CREATE|O_RDWR);
00474   if(fd < 0){
00475     printf(1, "fstests: cannot open sharedfd for writing");
00476     return;
00477   }
00478   pid = fork();
00479   memset(buf, pid==0?'c':'p', sizeof(buf));
00480   for(i = 0; i < 1000; i++){
00481     if(write(fd, buf, sizeof(buf)) != sizeof(buf)){
00482       printf(1, "fstests: write sharedfd failed\n");
00483       break;
00484     }
00485   }
00486   if(pid == 0)
00487     exit();
00488   else
00489     wait();
00490   close(fd);
00491   fd = open("sharedfd", 0);
00492   if(fd < 0){
00493     printf(1, "fstests: cannot open sharedfd for reading\n");
00494     return;
00495   }
00496   nc = np = 0;
00497   while((n = read(fd, buf, sizeof(buf))) > 0){
00498     for(i = 0; i < sizeof(buf); i++){
00499       if(buf[i] == 'c')
00500         nc++;
00501       if(buf[i] == 'p')
00502         np++;
00503     }
00504   }
00505   close(fd);
00506   unlink("sharedfd");
00507   if(nc == 10000 && np == 10000){
00508     printf(1, "sharedfd ok\n");
00509   } else {
00510     printf(1, "sharedfd oops %d %d\n", nc, np);
00511     exit();
00512   }
00513 }
00514
00515 // four processes write different files at the same
00516 // time, to test block allocation.
00517 void
```

```
00518 fourfiles(void)
00519 {
00520   int fd, pid, i, j, n, total, pi;
00521   char *names[] = { "f0", "f1", "f2", "f3" };
00522   char *fname;
00523
00524   printf(1, "fourfiles test\n");
00525
00526   for(pi = 0; pi < 4; pi++){
00527     fname = names[pi];
00528     unlink(fname);
00529
00530     pid = fork();
00531     if(pid < 0){
00532       printf(1, "fork failed\n");
00533       exit();
00534     }
00535
00536     if(pid == 0){
00537       fd = open(fname, O_CREATE | O_RDWR);
00538       if(fd < 0){
00539         printf(1, "create failed\n");
00540         exit();
00541       }
00542
00543       memset(buf, '0'+pi, 512);
00544       for(i = 0; i < 12; i++){
00545         if((n = write(fd, buf, 500)) != 500){
00546           printf(1, "write failed %d\n", n);
00547           exit();
00548         }
00549       }
00550       exit();
00551     }
00552   }
00553
00554   for(pi = 0; pi < 4; pi++){
00555     wait();
00556   }
00557
00558   for(i = 0; i < 2; i++){
00559     fname = names[i];
00560     fd = open(fname, 0);
00561     total = 0;
00562     while((n = read(fd, buf, sizeof(buf))) > 0){
00563       for(j = 0; j < n; j++){
00564         if(buf[j] != '0'+i){
00565           printf(1, "wrong char\n");
00566           exit();
00567         }
00568       }
00569       total += n;
00570     }
00571     close(fd);
00572     if(total != 12*500){
00573       printf(1, "wrong length %d\n", total);
00574       exit();
00575     }
00576     unlink(fname);
00577   }
00578
00579   printf(1, "fourfiles ok\n");
00580 }
00581
00582 // four processes create and delete different files in same directory
00583 void
00584 createdelete(void)
00585 {
00586   enum { N = 20 };
00587   int pid, i, fd, pi;
00588   char name[32];
00589
00590   printf(1, "createdelete test\n");
00591
00592   for(pi = 0; pi < 4; pi++){
00593     pid = fork();
00594     if(pid < 0){
00595       printf(1, "fork failed\n");
00596       exit();
00597     }
00598
00599     if(pid == 0){
00600       name[0] = 'p' + pi;
00601       name[2] = '\0';
00602       for(i = 0; i < N; i++){
00603         name[1] = '0' + i;
00604         fd = open(name, O_CREATE | O_RDWR);
```

```
00605         if(fd < 0){
00606           printf(1, "create failed\n");
00607           exit();
00608         }
00609         close(fd);
00610         if(i > 0 && (i % 2 ) == 0){
00611           name[1] = '0' + (i / 2);
00612           if(unlink(name) < 0){
00613             printf(1, "unlink failed\n");
00614             exit();
00615           }
00616         }
00617       }
00618       exit();
00619     }
00620   }
00621
00622   for(pi = 0; pi < 4; pi++){
00623     wait();
00624   }
00625
00626   name[0] = name[1] = name[2] = 0;
00627   for(i = 0; i < N; i++){
00628     for(pi = 0; pi < 4; pi++){
00629       name[0] = 'p' + pi;
00630       name[1] = '0' + i;
00631       fd = open(name, 0);
00632       if((i == 0 || i >= N/2) && fd < 0){
00633         printf(1, "oops createdelete %s didn't exist\n", name);
00634         exit();
00635       } else if((i >= 1 && i < N/2) && fd >= 0){
00636         printf(1, "oops createdelete %s did exist\n", name);
00637         exit();
00638       }
00639       if(fd >= 0)
00640         close(fd);
00641     }
00642   }
00643
00644   for(i = 0; i < N; i++){
00645     for(pi = 0; pi < 4; pi++){
00646       name[0] = 'p' + i;
00647       name[1] = '0' + i;
00648       unlink(name);
00649     }
00650   }
00651
00652   printf(1, "createdelete ok\n");
00653 }
00654
00655 // can I unlink a file and still read it?
00656 void
00657 unlinkread(void)
00658 {
00659   int fd, fd1;
00660
00661   printf(1, "unlinkread test\n");
00662   fd = open("unlinkread", O_CREATE | O_RDWR);
00663   if(fd < 0){
00664     printf(1, "create unlinkread failed\n");
00665     exit();
00666   }
00667   write(fd, "hello", 5);
00668   close(fd);
00669
00670   fd = open("unlinkread", O_RDWR);
00671   if(fd < 0){
00672     printf(1, "open unlinkread failed\n");
00673     exit();
00674   }
00675   if(unlink("unlinkread") != 0){
00676     printf(1, "unlink unlinkread failed\n");
00677     exit();
00678   }
00679
00680   fd1 = open("unlinkread", O_CREATE | O_RDWR);
00681   write(fd1, "yyy", 3);
00682   close(fd1);
00683
00684   if(read(fd, buf, sizeof(buf)) != 5){
00685     printf(1, "unlinkread read failed");
00686     exit();
00687   }
00688   if(buf[0] != 'h'){
00689     printf(1, "unlinkread wrong data\n");
00690     exit();
00691   }
```

```
00692   if(write(fd, buf, 10) != 10){
00693     printf(1, "unlinkread write failed\n");
00694     exit();
00695   }
00696   close(fd);
00697   unlink("unlinkread");
00698   printf(1, "unlinkread ok\n");
00699 }
00700
00701 void
00702 linktest(void)
00703 {
00704   int fd;
00705
00706   printf(1, "linktest\n");
00707
00708   unlink("lf1");
00709   unlink("lf2");
00710
00711   fd = open("lf1", O_CREATE|O_RDWR);
00712   if(fd < 0){
00713     printf(1, "create lf1 failed\n");
00714     exit();
00715   }
00716   if(write(fd, "hello", 5) != 5){
00717     printf(1, "write lf1 failed\n");
00718     exit();
00719   }
00720   close(fd);
00721
00722   if(link("lf1", "lf2") < 0){
00723     printf(1, "link lf1 lf2 failed\n");
00724     exit();
00725   }
00726   unlink("lf1");
00727
00728   if(open("lf1", 0) >= 0){
00729     printf(1, "unlinked lf1 but it is still there!\n");
00730     exit();
00731   }
00732
00733   fd = open("lf2", 0);
00734   if(fd < 0){
00735     printf(1, "open lf2 failed\n");
00736     exit();
00737   }
00738   if(read(fd, buf, sizeof(buf)) != 5){
00739     printf(1, "read lf2 failed\n");
00740     exit();
00741   }
00742   close(fd);
00743
00744   if(link("lf2", "lf2") >= 0){
00745     printf(1, "link lf2 lf2 succeeded! oops\n");
00746     exit();
00747   }
00748
00749   unlink("lf2");
00750   if(link("lf2", "lf1") >= 0){
00751     printf(1, "link non-existant succeeded! oops\n");
00752     exit();
00753   }
00754
00755   if(link(".", "lf1") >= 0){
00756     printf(1, "link . lf1 succeeded! oops\n");
00757     exit();
00758   }
00759
00760   printf(1, "linktest ok\n");
00761 }
00762
00763 // test concurrent create/link/unlink of the same file
00764 void
00765 concreate(void)
00766 {
00767   char file[3];
00768   int i, pid, n, fd;
00769   char fa[40];
00770   struct {
00771     ushort inum;
00772     char name[14];
00773   } de;
00774
00775   printf(1, "concreate test\n");
00776   file[0] = 'C';
00777   file[2] = '\0';
00778   for(i = 0; i < 40; i++){
```

```
00779     file[1] = '0' + i;
00780     unlink(file);
00781     pid = fork();
00782     if(pid && (i % 3) == 1){
00783       link("C0", file);
00784     } else if(pid == 0 && (i % 5) == 1){
00785       link("C0", file);
00786     } else {
00787       fd = open(file, O_CREATE | O_RDWR);
00788       if(fd < 0){
00789         printf(1, "concreate create %s failed\n", file);
00790         exit();
00791       }
00792       close(fd);
00793     }
00794     if(pid == 0)
00795       exit();
00796     else
00797       wait();
00798   }
00799
00800   memset(fa, 0, sizeof(fa));
00801   fd = open(".", 0);
00802   n = 0;
00803   while(read(fd, &de, sizeof(de)) > 0){
00804     if(de.inum == 0)
00805       continue;
00806     if(de.name[0] == 'C' && de.name[2] == '\0'){
00807       i = de.name[1] - '0';
00808       if(i < 0 || i >= sizeof(fa)){
00809         printf(1, "concreate weird file %s\n", de.name);
00810         exit();
00811       }
00812       if(fa[i]){
00813         printf(1, "concreate duplicate file %s\n", de.name);
00814         exit();
00815       }
00816       fa[i] = 1;
00817       n++;
00818     }
00819   }
00820   close(fd);
00821
00822   if(n != 40){
00823     printf(1, "concreate not enough files in directory listing\n");
00824     exit();
00825   }
00826
00827   for(i = 0; i < 40; i++){
00828     file[1] = '0' + i;
00829     pid = fork();
00830     if(pid < 0){
00831       printf(1, "fork failed\n");
00832       exit();
00833     }
00834     if(((i % 3) == 0 && pid == 0) ||
00835        ((i % 3) == 1 && pid != 0)){
00836       close(open(file, 0));
00837       close(open(file, 0));
00838       close(open(file, 0));
00839       close(open(file, 0));
00840     } else {
00841       unlink(file);
00842       unlink(file);
00843       unlink(file);
00844       unlink(file);
00845     }
00846     if(pid == 0)
00847       exit();
00848     else
00849       wait();
00850   }
00851
00852   printf(1, "concreate ok\n");
00853 }
00854
00855 // another concurrent link/unlink/create test,
00856 // to look for deadlocks.
00857 void
00858 linkunlink()
00859 {
00860   int pid, i;
00861
00862   printf(1, "linkunlink test\n");
00863
00864   unlink("x");
00865   pid = fork();
```

```
00866   if(pid < 0){
00867     printf(1, "fork failed\n");
00868     exit();
00869   }
00870
00871   unsigned int x = (pid ? 1 : 97);
00872   for(i = 0; i < 100; i++){
00873     x = x * 1103515245 + 12345;
00874     if((x % 3) == 0){
00875       close(open("x", O_RDWR | O_CREATE));
00876     } else if((x % 3) == 1){
00877       link("cat", "x");
00878     } else {
00879       unlink("x");
00880     }
00881   }
00882
00883   if(pid)
00884     wait();
00885   else
00886     exit();
00887
00888   printf(1, "linkunlink ok\n");
00889 }
00890
00891 // directory that uses indirect blocks
00892 void
00893 bigdir(void)
00894 {
00895   int i, fd;
00896   char name[10];
00897
00898   printf(1, "bigdir test\n");
00899   unlink("bd");
00900
00901   fd = open("bd", O_CREATE);
00902   if(fd < 0){
00903     printf(1, "bigdir create failed\n");
00904     exit();
00905   }
00906   close(fd);
00907
00908   for(i = 0; i < 500; i++){
00909     name[0] = 'x';
00910     name[1] = '0' + (i / 64);
00911     name[2] = '0' + (i % 64);
00912     name[3] = '\0';
00913     if(link("bd", name) != 0){
00914       printf(1, "bigdir link failed\n");
00915       exit();
00916     }
00917   }
00918
00919   unlink("bd");
00920   for(i = 0; i < 500; i++){
00921     name[0] = 'x';
00922     name[1] = '0' + (i / 64);
00923     name[2] = '0' + (i % 64);
00924     name[3] = '\0';
00925     if(unlink(name) != 0){
00926       printf(1, "bigdir unlink failed");
00927       exit();
00928     }
00929   }
00930
00931   printf(1, "bigdir ok\n");
00932 }
00933
00934 void
00935 subdir(void)
00936 {
00937   int fd, cc;
00938
00939   printf(1, "subdir test\n");
00940
00941   unlink("ff");
00942   if(mkdir("dd") != 0){
00943     printf(1, "subdir mkdir dd failed\n");
00944     exit();
00945   }
00946
00947   fd = open("dd/ff", O_CREATE | O_RDWR);
00948   if(fd < 0){
00949     printf(1, "create dd/ff failed\n");
00950     exit();
00951   }
00952   write(fd, "ff", 2);
```

```
00953   close(fd);
00954
00955   if(unlink("dd") >= 0){
00956     printf(1, "unlink dd (non-empty dir) succeeded!\n");
00957     exit();
00958   }
00959
00960   if(mkdir("/dd/dd") != 0){
00961     printf(1, "subdir mkdir dd/dd failed\n");
00962     exit();
00963   }
00964
00965   fd = open("dd/dd/ff", O_CREATE | O_RDWR);
00966   if(fd < 0){
00967     printf(1, "create dd/dd/ff failed\n");
00968     exit();
00969   }
00970   write(fd, "FF", 2);
00971   close(fd);
00972
00973   fd = open("dd/dd/../ff", 0);
00974   if(fd < 0){
00975     printf(1, "open dd/dd/../ff failed\n");
00976     exit();
00977   }
00978   cc = read(fd, buf, sizeof(buf));
00979   if(cc != 2 || buf[0] != 'f'){
00980     printf(1, "dd/dd/../ff wrong content\n");
00981     exit();
00982   }
00983   close(fd);
00984
00985   if(link("dd/dd/ff", "dd/dd/ffff") != 0){
00986     printf(1, "link dd/dd/ff dd/dd/ffff failed\n");
00987     exit();
00988   }
00989
00990   if(unlink("dd/dd/ff") != 0){
00991     printf(1, "unlink dd/dd/ff failed\n");
00992     exit();
00993   }
00994   if(open("dd/dd/ff", O_RDONLY) >= 0){
00995     printf(1, "open (unlinked) dd/dd/ff succeeded\n");
00996     exit();
00997   }
00998
00999   if(chdir("dd") != 0){
01000     printf(1, "chdir dd failed\n");
01001     exit();
01002   }
01003   if(chdir("dd/../../dd") != 0){
01004     printf(1, "chdir dd/../../dd failed\n");
01005     exit();
01006   }
01007   if(chdir("dd/../../../dd") != 0){
01008     printf(1, "chdir dd/../../dd failed\n");
01009     exit();
01010   }
01011   if(chdir("./..") != 0){
01012     printf(1, "chdir ./.. failed\n");
01013     exit();
01014   }
01015
01016   fd = open("dd/dd/ffff", 0);
01017   if(fd < 0){
01018     printf(1, "open dd/dd/ffff failed\n");
01019     exit();
01020   }
01021   if(read(fd, buf, sizeof(buf)) != 2){
01022     printf(1, "read dd/dd/ffff wrong len\n");
01023     exit();
01024   }
01025   close(fd);
01026
01027   if(open("dd/dd/ff", O_RDONLY) >= 0){
01028     printf(1, "open (unlinked) dd/dd/ff succeeded!\n");
01029     exit();
01030   }
01031
01032   if(open("dd/ff/ff", O_CREATE|O_RDWR) >= 0){
01033     printf(1, "create dd/ff/ff succeeded!\n");
01034     exit();
01035   }
01036   if(open("dd/xx/ff", O_CREATE|O_RDWR) >= 0){
01037     printf(1, "create dd/xx/ff succeeded!\n");
01038     exit();
01039   }
```

```
01040   if(open("dd", O_CREATE) >= 0){
01041     printf(1, "create dd succeeded!\n");
01042     exit();
01043   }
01044   if(open("dd", O_RDWR) >= 0){
01045     printf(1, "open dd rdwr succeeded!\n");
01046     exit();
01047   }
01048   if(open("dd", O_WRONLY) >= 0){
01049     printf(1, "open dd wronly succeeded!\n");
01050     exit();
01051   }
01052   if(link("dd/ff/ff", "dd/dd/xx") == 0){
01053     printf(1, "link dd/ff/ff dd/dd/xx succeeded!\n");
01054     exit();
01055   }
01056   if(link("dd/xx/ff", "dd/dd/xx") == 0){
01057     printf(1, "link dd/xx/ff dd/dd/xx succeeded!\n");
01058     exit();
01059   }
01060   if(link("dd/ff", "dd/dd/ffff") == 0){
01061     printf(1, "link dd/ff dd/dd/ffff succeeded!\n");
01062     exit();
01063   }
01064   if(mkdir("dd/ff/ff") == 0){
01065     printf(1, "mkdir dd/ff/ff succeeded!\n");
01066     exit();
01067   }
01068   if(mkdir("dd/xx/ff") == 0){
01069     printf(1, "mkdir dd/xx/ff succeeded!\n");
01070     exit();
01071   }
01072   if(mkdir("dd/dd/ffff") == 0){
01073     printf(1, "mkdir dd/dd/ffff succeeded!\n");
01074     exit();
01075   }
01076   if(unlink("dd/xx/ff") == 0){
01077     printf(1, "unlink dd/xx/ff succeeded!\n");
01078     exit();
01079   }
01080   if(unlink("dd/ff/ff") == 0){
01081     printf(1, "unlink dd/ff/ff succeeded!\n");
01082     exit();
01083   }
01084   if(chdir("dd/ff") == 0){
01085     printf(1, "chdir dd/ff succeeded!\n");
01086     exit();
01087   }
01088   if(chdir("dd/xx") == 0){
01089     printf(1, "chdir dd/xx succeeded!\n");
01090     exit();
01091   }
01092
01093   if(unlink("dd/dd/ffff") != 0){
01094     printf(1, "unlink dd/dd/ff failed\n");
01095     exit();
01096   }
01097   if(unlink("dd/ff") != 0){
01098     printf(1, "unlink dd/ff failed\n");
01099     exit();
01100   }
01101   if(unlink("dd") == 0){
01102     printf(1, "unlink non-empty dd succeeded!\n");
01103     exit();
01104   }
01105   if(unlink("dd/dd") < 0){
01106     printf(1, "unlink dd/dd failed\n");
01107     exit();
01108   }
01109   if(unlink("dd") < 0){
01110     printf(1, "unlink dd failed\n");
01111     exit();
01112   }
01113
01114   printf(1, "subdir ok\n");
01115 }
01116
01117 // test writes that are larger than the log.
01118 void
01119 bigwrite(void)
01120 {
01121   int fd, sz;
01122
01123   printf(1, "bigwrite test\n");
01124
01125   unlink("bigwrite");
01126   for(sz = 499; sz < 12*512; sz += 471){
```

```
01127     fd = open("bigwrite", O_CREATE | O_RDWR);
01128     if(fd < 0){
01129       printf(1, "cannot create bigwrite\n");
01130       exit();
01131     }
01132     int i;
01133     for(i = 0; i < 2; i++){
01134       int cc = write(fd, buf, sz);
01135       if(cc != sz){
01136         printf(1, "write(%d) ret %d\n", sz, cc);
01137         exit();
01138       }
01139     }
01140     close(fd);
01141     unlink("bigwrite");
01142   }
01143
01144   printf(1, "bigwrite ok\n");
01145 }
01146
01147 void
01148 bigfile(void)
01149 {
01150   int fd, i, total, cc;
01151
01152   printf(1, "bigfile test\n");
01153
01154   unlink("bigfile");
01155   fd = open("bigfile", O_CREATE | O_RDWR);
01156   if(fd < 0){
01157     printf(1, "cannot create bigfile");
01158     exit();
01159   }
01160   for(i = 0; i < 20; i++){
01161     memset(buf, i, 600);
01162     if(write(fd, buf, 600) != 600){
01163       printf(1, "write bigfile failed\n");
01164       exit();
01165     }
01166   }
01167   close(fd);
01168
01169   fd = open("bigfile", 0);
01170   if(fd < 0){
01171     printf(1, "cannot open bigfile\n");
01172     exit();
01173   }
01174   total = 0;
01175   for(i = 0; ; i++){
01176     cc = read(fd, buf, 300);
01177     if(cc < 0){
01178       printf(1, "read bigfile failed\n");
01179       exit();
01180     }
01181     if(cc == 0)
01182       break;
01183     if(cc != 300){
01184       printf(1, "short read bigfile\n");
01185       exit();
01186     }
01187     if(buf[0] != i/2 || buf[299] != i/2){
01188       printf(1, "read bigfile wrong data\n");
01189       exit();
01190     }
01191     total += cc;
01192   }
01193   close(fd);
01194   if(total != 20*600){
01195     printf(1, "read bigfile wrong total\n");
01196     exit();
01197   }
01198   unlink("bigfile");
01199
01200   printf(1, "bigfile test ok\n");
01201 }
01202
01203 void
01204 fourteen(void)
01205 {
01206   int fd;
01207
01208   // DIRSIZ is 14.
01209   printf(1, "fourteen test\n");
01210
01211   if(mkdir("12345678901234") != 0){
01212     printf(1, "mkdir 12345678901234 failed\n");
01213     exit();
```

```
01214   }
01215   if(mkdir("12345678901234/123456789012345") != 0){
01216     printf(1, "mkdir 12345678901234/123456789012345 failed\n");
01217     exit();
01218   }
01219   fd = open("123456789012345/123456789012345/123456789012345", O_CREATE);
01220   if(fd < 0){
01221     printf(1, "create 123456789012345/123456789012345/123456789012345 failed\n");
01222     exit();
01223   }
01224   close(fd);
01225   fd = open("12345678901234/12345678901234/12345678901234", 0);
01226   if(fd < 0){
01227     printf(1, "open 12345678901234/12345678901234/12345678901234 failed\n");
01228     exit();
01229   }
01230   close(fd);
01231
01232   if(mkdir("12345678901234/12345678901234") == 0){
01233     printf(1, "mkdir 12345678901234/12345678901234 succeeded!\n");
01234     exit();
01235   }
01236   if(mkdir("123456789012345/12345678901234") == 0){
01237     printf(1, "mkdir 12345678901234/123456789012345 succeeded!\n");
01238     exit();
01239   }
01240
01241   printf(1, "fourteen ok\n");
01242 }
01243
01244 void
01245 rmdot(void)
01246 {
01247   printf(1, "rmdot test\n");
01248   if(mkdir("dots") != 0){
01249     printf(1, "mkdir dots failed\n");
01250     exit();
01251   }
01252   if(chdir("dots") != 0){
01253     printf(1, "chdir dots failed\n");
01254     exit();
01255   }
01256   if(unlink(".") == 0){
01257     printf(1, "rm . worked!\n");
01258     exit();
01259   }
01260   if(unlink("..") == 0){
01261     printf(1, "rm .. worked!\n");
01262     exit();
01263   }
01264   if(chdir("/") != 0){
01265     printf(1, "chdir / failed\n");
01266     exit();
01267   }
01268   if(unlink("dots/.") == 0){
01269     printf(1, "unlink dots/. worked!\n");
01270     exit();
01271   }
01272   if(unlink("dots/..") == 0){
01273     printf(1, "unlink dots/.. worked!\n");
01274     exit();
01275   }
01276   if(unlink("dots") != 0){
01277     printf(1, "unlink dots failed!\n");
01278     exit();
01279   }
01280   printf(1, "rmdot ok\n");
01281 }
01282
01283 void
01284 dirfile(void)
01285 {
01286   int fd;
01287
01288   printf(1, "dir vs file\n");
01289
01290   fd = open("dirfile", O_CREATE);
01291   if(fd < 0){
01292     printf(1, "create dirfile failed\n");
01293     exit();
01294   }
01295   close(fd);
01296   if(chdir("dirfile") == 0){
01297     printf(1, "chdir dirfile succeeded!\n");
01298     exit();
01299   }
01300   fd = open("dirfile/xx", 0);
```

```
01301   if(fd >= 0){
01302     printf(1, "create dirfile/xx succeeded!\n");
01303     exit();
01304   }
01305   fd = open("dirfile/xx", O_CREATE);
01306   if(fd >= 0){
01307     printf(1, "create dirfile/xx succeeded!\n");
01308     exit();
01309   }
01310   if(mkdir("dirfile/xx") == 0){
01311     printf(1, "mkdir dirfile/xx succeeded!\n");
01312     exit();
01313   }
01314   if(unlink("dirfile/xx") == 0){
01315     printf(1, "unlink dirfile/xx succeeded!\n");
01316     exit();
01317   }
01318   if(link("README", "dirfile/xx") == 0){
01319     printf(1, "link to dirfile/xx succeeded!\n");
01320     exit();
01321   }
01322   if(unlink("dirfile") != 0){
01323     printf(1, "unlink dirfile failed!\n");
01324     exit();
01325   }
01326
01327   fd = open(".", O_RDWR);
01328   if(fd >= 0){
01329     printf(1, "open . for writing succeeded!\n");
01330     exit();
01331   }
01332   fd = open(".", 0);
01333   if(write(fd, "x", 1) > 0){
01334     printf(1, "write . succeeded!\n");
01335     exit();
01336   }
01337   close(fd);
01338
01339   printf(1, "dir vs file OK\n");
01340 }
01341
01342 // test that iput() is called at the end of _namei()
01343 void
01344 iref(void)
01345 {
01346   int i, fd;
01347
01348   printf(1, "empty file name\n");
01349
01350   // the 50 is NINODE
01351   for(i = 0; i < 50 + 1; i++){
01352     if(mkdir("irefd") != 0){
01353       printf(1, "mkdir irefd failed\n");
01354       exit();
01355     }
01356     if(chdir("irefd") != 0){
01357       printf(1, "chdir irefd failed\n");
01358       exit();
01359     }
01360
01361     mkdir("");
01362     link("README", "");
01363     fd = open("", O_CREATE);
01364     if(fd >= 0)
01365       close(fd);
01366     fd = open("xx", O_CREATE);
01367     if(fd >= 0)
01368       close(fd);
01369     unlink("xx");
01370   }
01371
01372   chdir("/");
01373   printf(1, "empty file name OK\n");
01374 }
01375
01376 // test that fork fails gracefully
01377 // the forktest binary also does this, but it runs out of proc entries first.
01378 // inside the bigger usertests binary, we run out of memory first.
01379 void
01380 forktest(void)
01381 {
01382   int n, pid;
01383
01384   printf(1, "fork test\n");
01385
01386   for(n=0; n<1000; n++){
01387     pid = fork();
```

```
01388     if(pid < 0)
01389       break;
01390     if(pid == 0)
01391       exit();
01392   }
01393
01394   if(n == 1000){
01395     printf(1, "fork claimed to work 1000 times!\n");
01396     exit();
01397   }
01398
01399   for(; n > 0; n--){
01400     if(wait() < 0){
01401       printf(1, "wait stopped early\n");
01402       exit();
01403     }
01404   }
01405
01406   if(wait() != -1){
01407     printf(1, "wait got too many\n");
01408     exit();
01409   }
01410
01411   printf(1, "fork test OK\n");
01412 }
01413
01414 void
01415 sbrktest(void)
01416 {
01417   int fds[2], pid, pids[10], ppid;
01418   char *a, *b, *c, *lastaddr, *oldbrk, *p, scratch;
01419   uint amt;
01420
01421   printf(stdout, "sbrk test\n");
01422   oldbrk = sbrk(0);
01423
01424   // can one sbrk() less than a page?
01425   a = sbrk(0);
01426   int i;
01427   for(i = 0; i < 5000; i++){
01428     b = sbrk(1);
01429     if(b != a){
01430       printf(stdout, "sbrk test failed %d %x %x\n", i, a, b);
01431       exit();
01432     }
01433     *b = 1;
01434     a = b + 1;
01435   }
01436   pid = fork();
01437   if(pid < 0){
01438     printf(stdout, "sbrk test fork failed\n");
01439     exit();
01440   }
01441   c = sbrk(1);
01442   c = sbrk(1);
01443   if(c != a + 1){
01444     printf(stdout, "sbrk test failed post-fork\n");
01445     exit();
01446   }
01447   if(pid == 0)
01448     exit();
01449   wait();
01450
01451   // can one grow address space to something big?
01452 #define BIG (100*1024*1024)
01453   a = sbrk(0);
01454   amt = (BIG) - (uint)a;
01455   p = sbrk(amt);
01456   if (p != a) {
01457     printf(stdout, "sbrk test failed to grow big address space; enough phys mem?\n");
01458     exit();
01459   }
01460   lastaddr = (char*) (BIG-1);
01461   *lastaddr = 99;
01462
01463   // can one de-allocate?
01464   a = sbrk(0);
01465   c = sbrk(-4096);
01466   if(c == (char*)0xffffffff){
01467     printf(stdout, "sbrk could not deallocate\n");
01468     exit();
01469   }
01470   c = sbrk(0);
01471   if(c != a - 4096){
01472     printf(stdout, "sbrk deallocation produced wrong address, a %x c %x\n", a, c);
01473     exit();
01474   }
```

```
01475
01476   // can one re-allocate that page?
01477   a = sbrk(0);
01478   c = sbrk(4096);
01479   if(c != a || sbrk(0) != a + 4096){
01480     printf(stdout, "sbrk re-allocation failed, a %x c %x\n", a, c);
01481     exit();
01482   }
01483   if(*lastaddr == 99){
01484     // should be zero
01485     printf(stdout, "sbrk de-allocation didn't really deallocate\n");
01486     exit();
01487   }
01488
01489   a = sbrk(0);
01490   c = sbrk(-(sbrk(0) - oldbrk));
01491   if(c != a){
01492     printf(stdout, "sbrk downsize failed, a %x c %x\n", a, c);
01493     exit();
01494   }
01495
01496   // can we read the kernel's memory?
01497   for(a = (char*)(KERNBASE); a < (char*) (KERNBASE+2000000); a += 50000){
01498     ppid = getpid();
01499     pid = fork();
01500     if(pid < 0){
01501       printf(stdout, "fork failed\n");
01502       exit();
01503     }
01504     if(pid == 0){
01505       printf(stdout, "oops could read %x = %x\n", a, *a);
01506       kill(ppid);
01507       exit();
01508     }
01509     wait();
01510   }
01511
01512   // if we run the system out of memory, does it clean up the last
01513   // failed allocation?
01514   if(pipe(fds) != 0){
01515     printf(1, "pipe() failed\n");
01516     exit();
01517   }
01518   for(i = 0; i < sizeof(pids)/sizeof(pids[0]); i++){
01519     if((pids[i] = fork()) == 0){
01520       // allocate a lot of memory
01521       sbrk(BIG - (uint)sbrk(0));
01522       write(fds[1], "x", 1);
01523       // sit around until killed
01524       for(;;) sleep(1000);
01525     }
01526     if(pids[i] != -1)
01527       read(fds[0], &scratch, 1);
01528   }
01529   // if those failed allocations freed up the pages they did allocate,
01530   // we'll be able to allocate here
01531   c = sbrk(4096);
01532   for(i = 0; i < sizeof(pids)/sizeof(pids[0]); i++){
01533     if(pids[i] == -1)
01534       continue;
01535     kill(pids[i]);
01536     wait();
01537   }
01538   if(c == (char*)0xffffffff){
01539     printf(stdout, "failed sbrk leaked memory\n");
01540     exit();
01541   }
01542
01543   if(sbrk(0) > oldbrk)
01544     sbrk(-(sbrk(0) - oldbrk));
01545
01546   printf(stdout, "sbrk test OK\n");
01547 }
01548
01549 void
01550 validateint(int *p)
01551 {
01552   int res;
01553   asm("mov %%esp, %%ebx\n\t"
01554       "mov %3, %%esp\n\t"
01555       "int %2\n\t"
01556       "mov %%ebx, %%esp" :
01557       "=a" (res) :
01558       "a" (SYS_sleep), "n" (T_SYSCALL), "c" (p) :
01559       "ebx");
01560 }
01561
```

```
01562 void
01563 validatetest(void)
01564 {
01565   int hi, pid;
01566   uint p;
01567
01568   printf(stdout, "validate test\n");
01569   hi = 1100*1024;
01570
01571   for(p = 0; p <= (uint)hi; p += 4096){
01572     if((pid = fork()) == 0){
01573       // try to crash the kernel by passing in a badly placed integer
01574       validateint((int*)p);
01575       exit();
01576     }
01577     sleep(0);
01578     sleep(0);
01579     kill(pid);
01580     wait();
01581
01582     // try to crash the kernel by passing in a bad string pointer
01583     if(link("nosuchfile", (char*)p) != -1){
01584       printf(stdout, "link should not succeed\n");
01585       exit();
01586     }
01587   }
01588
01589   printf(stdout, "validate ok\n");
01590 }
01591
01592 // does unintialized data start out zero?
01593 char uninit[10000];
01594 void
01595 bsstest(void)
01596 {
01597   int i;
01598
01599   printf(stdout, "bss test\n");
01600   for(i = 0; i < sizeof(uninit); i++){
01601     if(uninit[i] != '\0'){
01602       printf(stdout, "bss test failed\n");
01603       exit();
01604     }
01605   }
01606   printf(stdout, "bss test ok\n");
01607 }
01608
01609 // does exec return an error if the arguments
01610 // are larger than a page? or does it write
01611 // below the stack and wreck the instructions/data?
01612 void
01613 bigargtest(void)
01614 {
01615   int pid, fd;
01616
01617   unlink("bigarg-ok");
01618   pid = fork();
01619   if(pid == 0){
01620     static char *args[MAXARG];
01621     int i;
01622     for(i = 0; i < MAXARG-1; i++)
01623       args[i] = "bigargs test: failed\n
                                            ";
01624     args[MAXARG-1] = 0;
01625     printf(stdout, "bigarg test\n");
01626     exec("echo", args);
01627     printf(stdout, "bigarg test ok\n");
01628     fd = open("bigarg-ok", O_CREATE);
01629     close(fd);
01630     exit();
01631   } else if(pid < 0){
01632     printf(stdout, "bigargtest: fork failed\n");
01633     exit();
01634   }
01635   wait();
01636   fd = open("bigarg-ok", 0);
01637   if(fd < 0){
01638     printf(stdout, "bigarg test failed!\n");
01639     exit();
01640   }
01641   close(fd);
01642   unlink("bigarg-ok");
01643 }
01644
01645 // what happens when the file system runs out of blocks?
01646 // answer: balloc panics, so this test is not useful.
```

```
01647 void
01648 fsfull()
01649 {
01650   int nfiles;
01651   int fsblocks = 0;
01652
01653   printf(1, "fsfull test\n");
01654
01655   for(nfiles = 0; ; nfiles++){
01656     char name[64];
01657     name[0] = 'f';
01658     name[1] = '0' + nfiles / 1000;
01659     name[2] = '0' + (nfiles % 1000) / 100;
01660     name[3] = '0' + (nfiles % 100) / 10;
01661     name[4] = '0' + (nfiles % 10);
01662     name[5] = '\0';
01663     printf(1, "writing %s\n", name);
01664     int fd = open(name, O_CREATE|O_RDWR);
01665     if(fd < 0){
01666       printf(1, "open %s failed\n", name);
01667       break;
01668     }
01669     int total = 0;
01670     while(1){
01671       int cc = write(fd, buf, 512);
01672       if(cc < 512)
01673         break;
01674       total += cc;
01675       fsblocks++;
01676     }
01677     printf(1, "wrote %d bytes\n", total);
01678     close(fd);
01679     if(total == 0)
01680       break;
01681   }
01682
01683   while(nfiles >= 0){
01684     char name[64];
01685     name[0] = 'f';
01686     name[1] = '0' + nfiles / 1000;
01687     name[2] = '0' + (nfiles % 1000) / 100;
01688     name[3] = '0' + (nfiles % 100) / 10;
01689     name[4] = '0' + (nfiles % 10);
01690     name[5] = '\0';
01691     unlink(name);
01692     nfiles--;
01693   }
01694
01695   printf(1, "fsfull test finished\n");
01696 }
01697
01698 void
01699 uio()
01700 {
01701   #define RTC_ADDR 0x70
01702   #define RTC_DATA 0x71
01703
01704   ushort port = 0;
01705   uchar val = 0;
01706   int pid;
01707
01708   printf(1, "uio test\n");
01709   pid = fork();
01710   if(pid == 0){
01711     port = RTC_ADDR;
01712     val = 0x09;  /* year */
01713     /* http://wiki.osdev.org/Inline_Assembly/Examples */
01714     asm volatile("outb %0,%1"::"a"(val), "d" (port));
01715     port = RTC_DATA;
01716     asm volatile("inb %1,%0" : "=a" (val) : "d" (port));
01717     printf(1, "uio: uio succeeded; test FAILED\n");
01718     exit();
01719   } else if(pid < 0){
01720     printf (1, "fork failed\n");
01721     exit();
01722   }
01723   wait();
01724   printf(1, "uio test done\n");
01725 }
01726
01727 void argptest()
01728 {
01729   int fd;
01730   fd = open("init", O_RDONLY);
01731   if (fd < 0) {
01732     printf(2, "open failed\n");
01733     exit();
```

```
01734   }
01735   read(fd, sbrk(0) - 1, -1);
01736   close(fd);
01737   printf(1, "arg test passed\n");
01738 }
01739
01740 unsigned long randstate = 1;
01741 unsigned int
01742 rand()
01743 {
01744   randstate = randstate * 1664525 + 1013904223;
01745   return randstate;
01746 }
01747
01748 int
01749 main(int argc, char *argv[])
01750 {
01751   printf(1, "usertests starting\n");
01752
01753   if(open("usertests.ran", 0) >= 0){
01754     printf(1, "already ran user tests -- rebuild fs.img\n");
01755     exit();
01756   }
01757   close(open("usertests.ran", O_CREATE));
01758
01759   argptest();
01760   createdelete();
01761   linkunlink();
01762   concreate();
01763   fourfiles();
01764   sharedfd();
01765
01766   bigargtest();
01767   bigwrite();
01768   bigargtest();
01769   bsstest();
01770   sbrktest();
01771   validatetest();
01772
01773   opentest();
01774   writetest();
01775   writetest1();
01776   createtest();
01777
01778   openiputtest();
01779   exitiputtest();
01780   iputtest();
01781
01782   mem();
01783   pipe1();
01784   preempt();
01785   exitwait();
01786
01787   rmdot();
01788   fourteen();
01789   bigfile();
01790   subdir();
01791   linktest();
01792   unlinkread();
01793   dirfile();
01794   iref();
01795   forktest();
01796   bigdir(); // slow
01797
01798   uio();
01799
01800   exectest();
01801
01802   exit();
01803 }
```

5.242 usertests.d File Reference

5.243 usertests.d

Go to the documentation of this file.

```
00001 usertests.o: usertests.c /usr/include/stdc-predef.h param.h types.h \
00002  stat.h user.h fs.h fcntl.h syscall.h traps.h memlayout.h
```

5.244 vm.c File Reference

```
#include "param.h"
#include "types.h"
#include "defs.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "elf.h"
```

Classes

- struct kmap

Functions

- int allocuvm (pde_t *pgdir, uint oldsz, uint newsz)
- void clearpteu (pde_t *pgdir, char *uva)
- int copyout (pde_t *pgdir, uint va, void *p, uint len)
- pde_t * copyuvm (pde_t *pgdir, uint sz)
- int deallocuvm (pde_t *pgdir, uint oldsz, uint newsz)
- void freevm (pde_t *pgdir)
- void inituvm (pde_t *pgdir, char *init, uint sz)
- void kvmalloc (void)
- int loaduvm (pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
- static int mappages (pde_t *pgdir, void *va, uint size, uint pa, int perm)
- void seginit (void)
- pde_t * setupkvm (void)
- void switchkvm (void)
- void switchuvm (struct proc *p)
- char * uva2ka (pde_t *pgdir, char *uva)
- static pte_t * walkpgdir (pde_t *pgdir, const void *va, int alloc)

Variables

- char data []
- static struct kmap kmap []
- pde_t * kpgdir

5.244.1 Function Documentation

5.244.1.1 allocuvm()

```
int allocuvm (
              pde_t * pgdir,
              uint oldsz,
              uint newsz )
```

Definition at line 222 of file vm.c.

```
00223 {
00224   char *mem;
00225   uint a;
00226
00227   if(newsz >= KERNBASE)
00228     return 0;
00229   if(newsz < oldsz)
00230     return oldsz;
00231
00232   a = PGROUNDUP(oldsz);
00233   for(; a < newsz; a += PGSIZE){
00234     mem = kalloc();
00235     if(mem == 0){
00236       cprintf("allocuvm out of memory\n");
00237       deallocuvm(pgdir, newsz, oldsz);
00238       return 0;
00239     }
00240     memset(mem, 0, PGSIZE);
00241     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
00242       cprintf("allocuvm out of memory (2)\n");
00243       deallocuvm(pgdir, newsz, oldsz);
00244       kfree(mem);
00245       return 0;
00246     }
00247   }
00248   return newsz;
00249 }
```

Referenced by exec(), and growproc().

5.244.1.2 clearpteu()

```
void clearpteu (
              pde_t * pgdir,
              char * uva )
```

Definition at line 303 of file vm.c.

```
00304 {
00305   pte_t *pte;
00306
00307   pte = walkpgdir(pgdir, uva, 0);
00308   if(pte == 0)
00309     panic("clearpteu");
00310   *pte &= ~PTE_U;
00311 }
```

Referenced by exec().

5.244.1.3 copyout()

```
int copyout (
              pde_t * pgdir,
              uint va,
              void * p,
              uint len )
```

Definition at line 366 of file vm.c.

```
00367 {
00368   char *buf, *pa0;
00369   uint n, va0;
00370
00371   buf = (char*)p;
00372   while(len > 0){
00373     va0 = (uint)PGROUNDDOWN(va);
00374     pa0 = uva2ka(pgdir, (char*)va0);
00375     if(pa0 == 0)
00376       return -1;
00377     n = PGSIZE - (va - va0);
00378     if(n > len)
00379       n = len;
00380     memmove(pa0 + (va - va0), buf, n);
00381     len -= n;
00382     buf += n;
00383     va = va0 + PGSIZE;
00384   }
00385   return 0;
00386 }
```

Referenced by exec().

5.244.1.4 copyuvm()

```
pde_t * copyuvm (
              pde_t * pgdir,
              uint sz )
```

Definition at line 316 of file vm.c.

```
00317 {
00318   pde_t *d;
00319   pte_t *pte;
00320   uint pa, i, flags;
00321   char *mem;
00322
00323   if((d = setupkvm()) == 0)
00324     return 0;
00325   for(i = 0; i < sz; i += PGSIZE){
00326     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
00327       panic("copyuvm: pte should exist");
00328     if(!(*pte & PTE_P))
00329       panic("copyuvm: page not present");
00330     pa = PTE_ADDR(*pte);
00331     flags = PTE_FLAGS(*pte);
00332     if((mem = kalloc()) == 0)
00333       goto bad;
00334     memmove(mem, (char*)P2V(pa), PGSIZE);
00335     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
00336       kfree(mem);
00337       goto bad;
00338     }
00339   }
00340   return d;
00341
00342 bad:
00343   freevm(d);
00344   return 0;
00345 }
```

Referenced by fork().

5.244.1.5 deallocuvm()

```
int deallocuvm (
              pde_t * pgdir,
              uint oldsz,
              uint newsz )
```

Definition at line 256 of file vm.c.

```
00257 {
00258   pte_t *pte;
00259   uint a, pa;
00260
00261   if(newsz >= oldsz)
00262     return oldsz;
00263
00264   a = PGROUNDUP(newsz);
00265   for(; a  < oldsz; a += PGSIZE){
00266     pte = walkpgdir(pgdir, (char*)a, 0);
00267     if(!pte)
00268       a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
00269     else if((*pte & PTE_P) != 0){
00270       pa = PTE_ADDR(*pte);
00271       if(pa == 0)
00272         panic("kfree");
00273       char *v = P2V(pa);
00274       kfree(v);
00275       *pte = 0;
00276     }
00277   }
00278   return newsz;
00279 }
```

Referenced by allocuvm(), freevm(), and growproc().

5.244.1.6 freevm()

```
void freevm (
              pde_t * pgdir )
```

Definition at line 284 of file vm.c.

```
00285 {
00286   uint i;
00287
00288   if(pgdir == 0)
00289     panic("freevm: no pgdir");
00290   deallocuvm(pgdir, KERNBASE, 0);
00291   for(i = 0; i < NPDENTRIES; i++){
00292     if(pgdir[i] & PTE_P){
00293       char * v = P2V(PTE_ADDR(pgdir[i]));
00294       kfree(v);
00295     }
00296   }
00297   kfree((char*)pgdir);
00298 }
```

Referenced by copyuvm(), exec(), setupkvm(), and wait().

5.244.1.7 inituvm()

```
void inituvm (
              pde_t * pgdir,
              char * init,
              uint sz )
```

Definition at line 183 of file vm.c.

```
00184 {
00185   char *mem;
00186
00187   if(sz >= PGSIZE)
00188     panic("inituvm: more than a page");
00189   mem = kalloc();
00190   memset(mem, 0, PGSIZE);
00191   mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
00192   memmove(mem, init, sz);
00193 }
```

Referenced by userinit().

5.244.1.8 kvmalloc()

```
void kvmalloc (
              void  )
```

Definition at line 141 of file vm.c.

```
00142 {
00143   kpgdir = setupkvm();
00144   switchkvm();
00145 }
```

5.244.1.9 loaduvm()

```
int loaduvm (
              pde_t * pgdir,
              char * addr,
              struct inode * ip,
              uint offset,
              uint sz )
```

Definition at line 198 of file vm.c.

```
00199 {
00200   uint i, pa, n;
00201   pte_t *pte;
00202
00203   if((uint) addr % PGSIZE != 0)
00204     panic("loaduvm: addr must be page aligned");
00205   for(i = 0; i < sz; i += PGSIZE){
00206     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
00207       panic("loaduvm: address should exist");
00208     pa = PTE_ADDR(*pte);
00209     if(sz - i < PGSIZE)
00210       n = sz - i;
00211     else
00212       n = PGSIZE;
00213     if(readi(ip, P2V(pa), offset+i, n) != n)
00214       return -1;
00215   }
00216   return 0;
00217 }
```

Referenced by exec().

5.244.1.10 mappages()

```
static int mappages (
              pde_t * pgdir,
              void * va,
              uint size,
              uint pa,
              int perm )  [static]
```

Definition at line 61 of file vm.c.

```
00062 {
00063   char *a, *last;
00064   pte_t *pte;
00065
00066   a = (char*)PGROUNDDOWN((uint)va);
00067   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
00068   for(;;){
00069     if((pte = walkpgdir(pgdir, a, 1)) == 0)
00070       return -1;
00071     if(*pte & PTE_P)
00072       panic("remap");
00073     *pte = pa | perm | PTE_P;
00074     if(a == last)
00075       break;
00076     a += PGSIZE;
00077     pa += PGSIZE;
00078   }
00079   return 0;
00080 }
```

Referenced by allocuvm(), copyuvm(), inituvm(), and setupkvm().

5.244.1.11 seginit()

```
void seginit (
              void  )
```

Definition at line 16 of file vm.c.

```
00017 {
00018   struct cpu *c;
00019
00020   // Map "logical" addresses to virtual addresses using identity map.
00021   // Cannot share a CODE descriptor for both kernel and user
00022   // because it would have to have DPL_USR, but the CPU forbids
00023   // an interrupt from CPL=0 to DPL=3.
00024   c = &cpus[cpuid()];
00025   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
00026   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
00027   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
00028   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
00029   lgdt(c->gdt, sizeof(c->gdt));
00030 }
```

Referenced by mpenter().

5.244.1.12 setupkvm()

```
pde_t * setupkvm (
            void  )
```

Definition at line 119 of file vm.c.

```
00120 {
00121   pde_t *pgdir;
00122   struct kmap *k;
00123
00124   if((pgdir = (pde_t*)kalloc()) == 0)
00125     return 0;
00126   memset(pgdir, 0, PGSIZE);
00127   if (P2V(PHYSTOP) > (void*)DEVSPACE)
00128     panic("PHYSTOP too high");
00129   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
00130     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
00131                 (uint)k->phys_start, k->perm) < 0) {
00132       freevm(pgdir);
00133       return 0;
00134     }
00135   return pgdir;
00136 }
```

Referenced by copyuvm(), exec(), kvmalloc(), and userinit().

5.244.1.13 switchkvm()

```
void switchkvm (
            void  )
```

Definition at line 150 of file vm.c.

```
00151 {
00152   lcr3(V2P(kpgdir));   // switch to the kernel page table
00153 }
```

Referenced by kvmalloc(), mpenter(), and scheduler().

5.244.1.14 switchuvm()

```
void switchuvm (
            struct proc * p )
```

Definition at line 157 of file vm.c.

```
00158 {
00159   if(p == 0)
00160     panic("switchuvm: no process");
00161   if(p->kstack == 0)
00162     panic("switchuvm: no kstack");
00163   if(p->pgdir == 0)
00164     panic("switchuvm: no pgdir");
00165
00166   pushcli();
00167   mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
00168                                 sizeof(mycpu()->ts)-1, 0);
00169   mycpu()->gdt[SEG_TSS].s = 0;
00170   mycpu()->ts.ss0 = SEG_KDATA << 3;
00171   mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
00172   // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
00173   // forbids I/O instructions (e.g., inb and outb) from user space
00174   mycpu()->ts.iomb = (ushort) 0xFFFF;
00175   ltr(SEG_TSS << 3);
00176   lcr3(V2P(p->pgdir));  // switch to process's address space
00177   popcli();
00178 }
```

Referenced by exec(), growproc(), and scheduler().

5.244.1.15 uva2ka()

```
char * uva2ka (
              pde_t * pgdir,
              char * uva )
```

Definition at line 350 of file vm.c.

```
00351 {
00352   pte_t *pte;
00353
00354   pte = walkpgdir(pgdir, uva, 0);
00355   if((*pte & PTE_P) == 0)
00356     return 0;
00357   if((*pte & PTE_U) == 0)
00358     return 0;
00359   return (char*)P2V(PTE_ADDR(*pte));
00360 }
```

Referenced by copyout().

5.244.1.16 walkpgdir()

```
static pte_t * walkpgdir (
              pde_t * pgdir,
              const void * va,
              int alloc )  [static]
```

Definition at line 36 of file vm.c.

```
00037 {
00038   pde_t *pde;
00039   pte_t *pgtab;
00040
00041   pde = &pgdir[PDX(va)];
00042   if(*pde & PTE_P){
00043     pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
00044   } else {
00045     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
00046       return 0;
00047     // Make sure all those PTE_P bits are zero.
00048     memset(pgtab, 0, PGSIZE);
00049     // The permissions here are overly generous, but they can
00050     // be further restricted by the permissions in the page table
00051     // entries, if necessary.
00052     *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
00053   }
00054   return &pgtab[PTX(va)];
00055 }
```

Referenced by clearpteu(), copyuvm(), deallocuvm(), loaduvm(), mappages(), and uva2ka().

5.244.2 Variable Documentation

5.244.2.1 data

```
char data[]  [extern]
```

Referenced by inb(), ioapicwrite(), kbdgetc(), main(), outb(), outw(), stosb(), and stosl().

5.244.2.2 kmap

```
struct kmap kmap[]  [static]
```

Initial value:

```
= {
 { (void*)KERNBASE, 0,              EXTMEM,    PTE_W},
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W},
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W},
}
```

5.244.2.3 kpgdir

```
pde_t* kpgdir
```

Definition at line 11 of file vm.c.

Referenced by kvmalloc(), and switchkvm().

5.245 vm.c

Go to the documentation of this file.

```
00001 #include "param.h"
00002 #include "types.h"
00003 #include "defs.h"
00004 #include "x86.h"
00005 #include "memlayout.h"
00006 #include "mmu.h"
00007 #include "proc.h"
00008 #include "elf.h"
00009
00010 extern char data[];  // defined by kernel.ld
00011 pde_t *kpgdir;  // for use in scheduler()
00012
00013 // Set up CPU's kernel segment descriptors.
00014 // Run once on entry on each CPU.
00015 void
00016 seginit(void)
00017 {
00018   struct cpu *c;
00019
00020   // Map "logical" addresses to virtual addresses using identity map.
00021   // Cannot share a CODE descriptor for both kernel and user
00022   // because it would have to have DPL_USR, but the CPU forbids
00023   // an interrupt from CPL=0 to DPL=3.
00024   c = &cpus[cpuid()];
00025   c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
00026   c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
00027   c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
00028   c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
00029   lgdt(c->gdt, sizeof(c->gdt));
00030 }
00031
00032 // Return the address of the PTE in page table pgdir
00033 // that corresponds to virtual address va.  If alloc!=0,
00034 // create any required page table pages.
00035 static pte_t *
00036 walkpgdir(pde_t *pgdir, const void *va, int alloc)
00037 {
00038   pde_t *pde;
00039   pte_t *pgtab;
00040
00041   pde = &pgdir[PDX(va)];
00042   if(*pde & PTE_P){
00043     pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
00044   } else {
00045     if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
00046       return 0;
```

```
00047     // Make sure all those PTE_P bits are zero.
00048     memset(pgtab, 0, PGSIZE);
00049     // The permissions here are overly generous, but they can
00050     // be further restricted by the permissions in the page table
00051     // entries, if necessary.
00052     *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
00053   }
00054   return &pgtab[PTX(va)];
00055 }
00056
00057 // Create PTEs for virtual addresses starting at va that refer to
00058 // physical addresses starting at pa. va and size might not
00059 // be page-aligned.
00060 static int
00061 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
00062 {
00063   char *a, *last;
00064   pte_t *pte;
00065
00066   a = (char*)PGROUNDDOWN((uint)va);
00067   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
00068   for(;;){
00069     if((pte = walkpgdir(pgdir, a, 1)) == 0)
00070       return -1;
00071     if(*pte & PTE_P)
00072       panic("remap");
00073     *pte = pa | perm | PTE_P;
00074     if(a == last)
00075       break;
00076     a += PGSIZE;
00077     pa += PGSIZE;
00078   }
00079   return 0;
00080 }
00081
00082 // There is one page table per process, plus one that's used when
00083 // a CPU is not running any process (kpgdir). The kernel uses the
00084 // current process's page table during system calls and interrupts;
00085 // page protection bits prevent user code from using the kernel's
00086 // mappings.
00087 //
00088 // setupkvm() and exec() set up every page table like this:
00089 //
00090 //   0..KERNBASE: user memory (text+data+stack+heap), mapped to
00091 //                phys memory allocated by the kernel
00092 //   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
00093 //   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
00094 //                for the kernel's instructions and r/o data
00095 //   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
00096 //                                  rw data + free physical memory
00097 //   0xfe000000..0: mapped direct (devices such as ioapic)
00098 //
00099 // The kernel allocates physical memory for its heap and for user memory
00100 // between V2P(end) and the end of physical memory (PHYSTOP)
00101 // (directly addressable from end..P2V(PHYSTOP)).
00102
00103 // This table defines the kernel's mappings, which are present in
00104 // every process's page table.
00105 static struct kmap {
00106   void *virt;
00107   uint phys_start;
00108   uint phys_end;
00109   int perm;
00110 } kmap[] = {
00111  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
00112  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
00113  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
00114  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
00115 };
00116
00117 // Set up kernel part of a page table.
00118 pde_t*
00119 setupkvm(void)
00120 {
00121   pde_t *pgdir;
00122   struct kmap *k;
00123
00124   if((pgdir = (pde_t*)kalloc()) == 0)
00125     return 0;
00126   memset(pgdir, 0, PGSIZE);
00127   if (P2V(PHYSTOP) > (void*)DEVSPACE)
00128     panic("PHYSTOP too high");
00129   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
00130     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
00131                 (uint)k->phys_start, k->perm) < 0) {
00132       freevm(pgdir);
00133       return 0;
```

```
00134     }
00135   return pgdir;
00136 }
00137
00138 // Allocate one page table for the machine for the kernel address
00139 // space for scheduler processes.
00140 void
00141 kvmalloc(void)
00142 {
00143   kpgdir = setupkvm();
00144   switchkvm();
00145 }
00146
00147 // Switch h/w page table register to the kernel-only page table,
00148 // for when no process is running.
00149 void
00150 switchkvm(void)
00151 {
00152   lcr3(V2P(kpgdir));   // switch to the kernel page table
00153 }
00154
00155 // Switch TSS and h/w page table to correspond to process p.
00156 void
00157 switchuvm(struct proc *p)
00158 {
00159   if(p == 0)
00160     panic("switchuvm: no process");
00161   if(p->kstack == 0)
00162     panic("switchuvm: no kstack");
00163   if(p->pgdir == 0)
00164     panic("switchuvm: no pgdir");
00165
00166   pushcli();
00167   mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
00168                                 sizeof(mycpu()->ts)-1, 0);
00169   mycpu()->gdt[SEG_TSS].s = 0;
00170   mycpu()->ts.ss0 = SEG_KDATA << 3;
00171   mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
00172   // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
00173   // forbids I/O instructions (e.g., inb and outb) from user space
00174   mycpu()->ts.iomb = (ushort) 0xFFFF;
00175   ltr(SEG_TSS << 3);
00176   lcr3(V2P(p->pgdir));  // switch to process's address space
00177   popcli();
00178 }
00179
00180 // Load the initcode into address 0 of pgdir.
00181 // sz must be less than a page.
00182 void
00183 inituvm(pde_t *pgdir, char *init, uint sz)
00184 {
00185   char *mem;
00186
00187   if(sz >= PGSIZE)
00188     panic("inituvm: more than a page");
00189   mem = kalloc();
00190   memset(mem, 0, PGSIZE);
00191   mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
00192   memmove(mem, init, sz);
00193 }
00194
00195 // Load a program segment into pgdir.  addr must be page-aligned
00196 // and the pages from addr to addr+sz must already be mapped.
00197 int
00198 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
00199 {
00200   uint i, pa, n;
00201   pte_t *pte;
00202
00203   if((uint) addr % PGSIZE != 0)
00204     panic("loaduvm: addr must be page aligned");
00205   for(i = 0; i < sz; i += PGSIZE){
00206     if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
00207       panic("loaduvm: address should exist");
00208     pa = PTE_ADDR(*pte);
00209     if(sz - i < PGSIZE)
00210       n = sz - i;
00211     else
00212       n = PGSIZE;
00213     if(readi(ip, P2V(pa), offset+i, n) != n)
00214       return -1;
00215   }
00216   return 0;
00217 }
00218
00219 // Allocate page tables and physical memory to grow process from oldsz to
00220 // newsz, which need not be page aligned.  Returns new size or 0 on error.
```

```
00221 int
00222 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
00223 {
00224   char *mem;
00225   uint a;
00226
00227   if(newsz >= KERNBASE)
00228     return 0;
00229   if(newsz < oldsz)
00230     return oldsz;
00231
00232   a = PGROUNDUP(oldsz);
00233   for(; a < newsz; a += PGSIZE){
00234     mem = kalloc();
00235     if(mem == 0){
00236       cprintf("allocuvm out of memory\n");
00237       deallocuvm(pgdir, newsz, oldsz);
00238       return 0;
00239     }
00240     memset(mem, 0, PGSIZE);
00241     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
00242       cprintf("allocuvm out of memory (2)\n");
00243       deallocuvm(pgdir, newsz, oldsz);
00244       kfree(mem);
00245       return 0;
00246     }
00247   }
00248   return newsz;
00249 }
00250
00251 // Deallocate user pages to bring the process size from oldsz to
00252 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
00253 // need to be less than oldsz.  oldsz can be larger than the actual
00254 // process size.  Returns the new process size.
00255 int
00256 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
00257 {
00258   pte_t *pte;
00259   uint a, pa;
00260
00261   if(newsz >= oldsz)
00262     return oldsz;
00263
00264   a = PGROUNDUP(newsz);
00265   for(; a  < oldsz; a += PGSIZE){
00266     pte = walkpgdir(pgdir, (char*)a, 0);
00267     if(!pte)
00268       a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
00269     else if((*pte & PTE_P) != 0){
00270       pa = PTE_ADDR(*pte);
00271       if(pa == 0)
00272         panic("kfree");
00273       char *v = P2V(pa);
00274       kfree(v);
00275       *pte = 0;
00276     }
00277   }
00278   return newsz;
00279 }
00280
00281 // Free a page table and all the physical memory pages
00282 // in the user part.
00283 void
00284 freevm(pde_t *pgdir)
00285 {
00286   uint i;
00287
00288   if(pgdir == 0)
00289     panic("freevm: no pgdir");
00290   deallocuvm(pgdir, KERNBASE, 0);
00291   for(i = 0; i < NPDENTRIES; i++){
00292     if(pgdir[i] & PTE_P){
00293       char * v = P2V(PTE_ADDR(pgdir[i]));
00294       kfree(v);
00295     }
00296   }
00297   kfree((char*)pgdir);
00298 }
00299
00300 // Clear PTE_U on a page. Used to create an inaccessible
00301 // page beneath the user stack.
00302 void
00303 clearpteu(pde_t *pgdir, char *uva)
00304 {
00305   pte_t *pte;
00306
00307   pte = walkpgdir(pgdir, uva, 0);
```

```
00308   if(pte == 0)
00309     panic("clearpteu");
00310   *pte &= ~PTE_U;
00311 }
00312
00313 // Given a parent process's page table, create a copy
00314 // of it for a child.
00315 pde_t*
00316 copyuvm(pde_t *pgdir, uint sz)
00317 {
00318   pde_t *d;
00319   pte_t *pte;
00320   uint pa, i, flags;
00321   char *mem;
00322
00323   if((d = setupkvm()) == 0)
00324     return 0;
00325   for(i = 0; i < sz; i += PGSIZE){
00326     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
00327       panic("copyuvm: pte should exist");
00328     if(!(*pte & PTE_P))
00329       panic("copyuvm: page not present");
00330     pa = PTE_ADDR(*pte);
00331     flags = PTE_FLAGS(*pte);
00332     if((mem = kalloc()) == 0)
00333       goto bad;
00334     memmove(mem, (char*)P2V(pa), PGSIZE);
00335     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
00336       kfree(mem);
00337       goto bad;
00338     }
00339   }
00340   return d;
00341
00342 bad:
00343   freevm(d);
00344   return 0;
00345 }
00346
00347 //PAGEBREAK!
00348 // Map user virtual address to kernel address.
00349 char*
00350 uva2ka(pde_t *pgdir, char *uva)
00351 {
00352   pte_t *pte;
00353
00354   pte = walkpgdir(pgdir, uva, 0);
00355   if((*pte & PTE_P) == 0)
00356     return 0;
00357   if((*pte & PTE_U) == 0)
00358     return 0;
00359   return (char*)P2V(PTE_ADDR(*pte));
00360 }
00361
00362 // Copy len bytes from p to user address va in page table pgdir.
00363 // Most useful when pgdir is not the current page table.
00364 // uva2ka ensures this only works for PTE_U pages.
00365 int
00366 copyout(pde_t *pgdir, uint va, void *p, uint len)
00367 {
00368   char *buf, *pa0;
00369   uint n, va0;
00370
00371   buf = (char*)p;
00372   while(len > 0){
00373     va0 = (uint)PGROUNDDOWN(va);
00374     pa0 = uva2ka(pgdir, (char*)va0);
00375     if(pa0 == 0)
00376       return -1;
00377     n = PGSIZE - (va - va0);
00378     if(n > len)
00379       n = len;
00380     memmove(pa0 + (va - va0), buf, n);
00381     len -= n;
00382     buf += n;
00383     va = va0 + PGSIZE;
00384   }
00385   return 0;
00386 }
00387
00388 //PAGEBREAK!
00389 // Blank page.
00390 //PAGEBREAK!
00391 // Blank page.
00392 //PAGEBREAK!
00393 // Blank page.
00394
```

5.246 vm.d File Reference

5.247 vm.d

Go to the documentation of this file.

```
00001 vm.o: vm.c /usr/include/stdc-predef.h param.h types.h defs.h x86.h \
00002  memlayout.h mmu.h proc.h elf.h
```

5.248 wc.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- int main (int argc, char *argv[])
- void wc (int fd, char *name)

Variables

- char buf [512]

5.248.1 Function Documentation

5.248.1.1 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 36 of file wc.c.

```
00037 {
00038   int fd, i;
00039
00040   if(argc <= 1){
00041     wc(0, "");
00042     exit();
00043   }
00044
00045   for(i = 1; i < argc; i++){
00046     if((fd = open(argv[i], 0)) < 0){
00047       printf(1, "wc: cannot open %s\n", argv[i]);
00048       exit();
00049     }
00050     wc(fd, argv[i]);
00051     close(fd);
00052   }
00053   exit();
00054 }
```

5.248.1.2 wc()

```
void wc (
              int fd,
              char * name )
```

Definition at line 8 of file wc.c.

```
00009 {
00010   int i, n;
00011   int l, w, c, inword;
00012
00013   l = w = c = 0;
00014   inword = 0;
00015   while((n = read(fd, buf, sizeof(buf))) > 0){
00016     for(i=0; i<n; i++){
00017       c++;
00018       if(buf[i] == '\n')
00019         l++;
00020       if(strchr(" \r\t\n\v", buf[i]))
00021         inword = 0;
00022       else if(!inword){
00023         w++;
00024         inword = 1;
00025       }
00026     }
00027   }
00028   if(n < 0){
00029     printf(1, "wc: read error\n");
00030     exit();
00031   }
00032   printf(1, "%d %d %d %s\n", l, w, c, name);
00033 }
```

Referenced by main().

5.248.2 Variable Documentation

5.248.2.1 buf

```
char buf[512]
```

Definition at line 5 of file wc.c.

5.249 wc.c

Go to the documentation of this file.

```
00001 #include "types.h"
00002 #include "stat.h"
00003 #include "user.h"
00004
00005 char buf[512];
00006
00007 void
00008 wc(int fd, char *name)
00009 {
00010   int i, n;
00011   int l, w, c, inword;
00012
00013   l = w = c = 0;
00014   inword = 0;
00015   while((n = read(fd, buf, sizeof(buf))) > 0){
00016     for(i=0; i<n; i++){
00017       c++;
00018       if(buf[i] == '\n')
```

```
00019         l++;
00020       if(strchr(" \r\t\n\v", buf[i]))
00021         inword = 0;
00022       else if(!inword){
00023         w++;
00024         inword = 1;
00025       }
00026     }
00027   }
00028   if(n < 0){
00029     printf(1, "wc: read error\n");
00030     exit();
00031   }
00032   printf(1, "%d %d %d %s\n", l, w, c, name);
00033 }
00034
00035 int
00036 main(int argc, char *argv[])
00037 {
00038   int fd, i;
00039
00040   if(argc <= 1){
00041     wc(0, "");
00042     exit();
00043   }
00044
00045   for(i = 1; i < argc; i++){
00046     if((fd = open(argv[i], 0)) < 0){
00047       printf(1, "wc: cannot open %s\n", argv[i]);
00048       exit();
00049     }
00050     wc(fd, argv[i]);
00051     close(fd);
00052   }
00053   exit();
00054 }
```

5.250 wc.d File Reference

5.251 wc.d

Go to the documentation of this file.

```
00001 wc.o: wc.c /usr/include/stdc-predef.h types.h stat.h user.h
```

5.252 x86.h File Reference

Classes

- struct trapframe

Functions

- static void cli (void)
- static uchar inb (ushort port)
- static void insl (int port, void *addr, int cnt)
- static void lcr3 (uint val)
- static void lgdt (struct segdesc *p, int size)
- static void lidt (struct gatedesc *p, int size)
- static void loadgs (ushort v)
- static void ltr (ushort sel)
- static void outb (ushort port, uchar data)

- static void outsl (int port, const void *addr, int cnt)
- static void outw (ushort port, ushort data)
- static uint rcr2 (void)
- static uint readeflags (void)
- static void sti (void)
- static void stosb (void *addr, int data, int cnt)
- static void stosl (void *addr, int data, int cnt)
- static uint xchg (volatile uint *addr, uint newval)

5.252.1 Function Documentation

5.252.1.1 cli()

```
static void cli (
            void  )  [inline], [static]
```

Definition at line 109 of file x86.h.

```
00110 {
00111   asm volatile("cli");
00112 }
```

Referenced by consputc(), panic(), and pushcli().

5.252.1.2 inb()

```
static uchar inb (
            ushort port )  [inline], [static]
```

Definition at line 4 of file x86.h.

```
00005 {
00006   uchar data;
00007
00008   asm volatile("in %1,%0" : "=a" (data) : "d" (port));
00009   return data;
00010 }
```

Referenced by cgaputc(), cmos_read(), ideinit(), idewait(), kbdgetc(), mpinit(), uartgetc(), uartinit(), uartputc(), and waitdisk().

5.252.1.3 insl()

```
static void insl (
            int port,
            void * addr,
            int cnt )  [inline], [static]
```

Definition at line 13 of file x86.h.

```
00014 {
00015   asm volatile("cld; rep insl" :
00016                "=D" (addr), "=c" (cnt) :
00017                "d" (port), "0" (addr), "1" (cnt) :
00018                "memory", "cc");
00019 }
```

Referenced by ideintr(), and readsect().

5.252.1.4 lcr3()

```
static void lcr3 (
             uint val )  [inline], [static]
```

Definition at line 142 of file x86.h.

```
00143 {
00144   asm volatile("movl %0,%%cr3" : : "r" (val));
00145 }
```

Referenced by switchkvm(), and switchuvm().

5.252.1.5 lgdt()

```
static void lgdt (
             struct segdesc * p,
             int size )  [inline], [static]
```

Definition at line 63 of file x86.h.

```
00064 {
00065   volatile ushort pd[3];
00066
00067   pd[0] = size-1;
00068   pd[1] = (uint)p;
00069   pd[2] = (uint)p >> 16;
00070
00071   asm volatile("lgdt (%0)" : : "r" (pd));
00072 }
```

Referenced by seginit().

5.252.1.6 lidt()

```
static void lidt (
             struct gatedesc * p,
             int size )  [inline], [static]
```

Definition at line 77 of file x86.h.

```
00078 {
00079   volatile ushort pd[3];
00080
00081   pd[0] = size-1;
00082   pd[1] = (uint)p;
00083   pd[2] = (uint)p >> 16;
00084
00085   asm volatile("lidt (%0)" : : "r" (pd));
00086 }
```

Referenced by idtinit().

5.252.1.7 loadgs()

```
static void loadgs (
            ushort v )  [inline], [static]
```

Definition at line 103 of file x86.h.

```
00104 {
00105   asm volatile("movw %0, %%gs" : : "r" (v));
00106 }
```

5.252.1.8 ltr()

```
static void ltr (
            ushort sel )  [inline], [static]
```

Definition at line 89 of file x86.h.

```
00090 {
00091   asm volatile("ltr %0" : : "r" (sel));
00092 }
```

Referenced by switchuvm().

5.252.1.9 outb()

```
static void outb (
            ushort port,
            uchar data )  [inline], [static]
```

Definition at line 22 of file x86.h.

```
00023 {
00024   asm volatile("out %0,%1" : : "a" (data), "d" (port));
00025 }
```

Referenced by cgaputc(), cmos_read(), ideinit(), idestart(), lapicstartap(), mpinit(), picinit(), readsect(), sys_halt(), uartinit(), and uartputc().

5.252.1.10 outsl()

```
static void outsl (
            int port,
            const void * addr,
            int cnt )  [inline], [static]
```

Definition at line 34 of file x86.h.

```
00035 {
00036   asm volatile("cld; rep outsl" :
00037                "=S" (addr), "=c" (cnt) :
00038                "d" (port), "0" (addr), "1" (cnt) :
00039                "cc");
00040 }
```

Referenced by idestart().

5.252.1.11 outw()

```
static void outw (
            ushort port,
            ushort data )  [inline], [static]
```

Definition at line 28 of file x86.h.

```
00029 {
00030   asm volatile("out %0,%1" : : "a" (data), "d" (port));
00031 }
```

5.252.1.12 rcr2()

```
static uint rcr2 (
            void  )  [inline], [static]
```

Definition at line 134 of file x86.h.

```
00135 {
00136   uint val;
00137   asm volatile("movl %%cr2,%0" : "=r" (val));
00138   return val;
00139 }
```

Referenced by trap().

5.252.1.13 readeflags()

```
static uint readeflags (
            void  )  [inline], [static]
```

Definition at line 95 of file x86.h.

```
00096 {
00097   uint eflags;
00098   asm volatile("pushfl; popl %0" : "=r" (eflags));
00099   return eflags;
00100 }
```

Referenced by mycpu(), popcli(), pushcli(), and sched().

5.252.1.14 sti()

```
static void sti (
            void  )  [inline], [static]
```

Definition at line 115 of file x86.h.

```
00116 {
00117   asm volatile("sti");
00118 }
```

Referenced by cps(), popcli(), and scheduler().

5.252.1.15 stosb()

```
static void stosb (
              void * addr,
              int data,
              int cnt )  [inline], [static]
```

Definition at line 43 of file x86.h.

```
00044 {
00045   asm volatile("cld; rep stosb" :
00046                "=D" (addr), "=c" (cnt) :
00047                "0" (addr), "1" (cnt), "a" (data) :
00048                "memory", "cc");
00049 }
```

Referenced by bootmain(), and memset().

5.252.1.16 stosl()

```
static void stosl (
              void * addr,
              int data,
              int cnt )  [inline], [static]
```

Definition at line 52 of file x86.h.

```
00053 {
00054   asm volatile("cld; rep stosl" :
00055                "=D" (addr), "=c" (cnt) :
00056                "0" (addr), "1" (cnt), "a" (data) :
00057                "memory", "cc");
00058 }
```

Referenced by memset().

5.252.1.17 xchg()

```
static uint xchg (
              volatile uint * addr,
              uint newval )  [inline], [static]
```

Definition at line 121 of file x86.h.

```
00122 {
00123   uint result;
00124
00125   // The + in "+m" denotes a read-modify-write operand.
00126   asm volatile("lock; xchgl %0, %1" :
00127                "+m" (*addr), "=a" (result) :
00128                "1" (newval) :
00129                "cc");
00130   return result;
00131 }
```

Referenced by acquire().

5.253 x86.h

Go to the documentation of this file.

```
00001 // Routines to let C code use special x86 instructions.
00002
00003 static inline uchar
00004 inb(ushort port)
00005 {
00006   uchar data;
00007
00008   asm volatile("in %1,%0" : "=a" (data) : "d" (port));
00009   return data;
00010 }
00011
00012 static inline void
00013 insl(int port, void *addr, int cnt)
00014 {
00015   asm volatile("cld; rep insl" :
00016                "=D" (addr), "=c" (cnt) :
00017                "d" (port), "0" (addr), "1" (cnt) :
00018                "memory", "cc");
00019 }
00020
00021 static inline void
00022 outb(ushort port, uchar data)
00023 {
00024   asm volatile("out %0,%1" : : "a" (data), "d" (port));
00025 }
00026
00027 static inline void
00028 outw(ushort port, ushort data)
00029 {
00030   asm volatile("out %0,%1" : : "a" (data), "d" (port));
00031 }
00032
00033 static inline void
00034 outsl(int port, const void *addr, int cnt)
00035 {
00036   asm volatile("cld; rep outsl" :
00037                "=S" (addr), "=c" (cnt) :
00038                "d" (port), "0" (addr), "1" (cnt) :
00039                "cc");
00040 }
00041
00042 static inline void
00043 stosb(void *addr, int data, int cnt)
00044 {
00045   asm volatile("cld; rep stosb" :
00046                "=D" (addr), "=c" (cnt) :
00047                "0" (addr), "1" (cnt), "a" (data) :
00048                "memory", "cc");
00049 }
00050
00051 static inline void
00052 stosl(void *addr, int data, int cnt)
00053 {
00054   asm volatile("cld; rep stosl" :
00055                "=D" (addr), "=c" (cnt) :
00056                "0" (addr), "1" (cnt), "a" (data) :
00057                "memory", "cc");
00058 }
00059
00060 struct segdesc;
00061
00062 static inline void
00063 lgdt(struct segdesc *p, int size)
00064 {
00065   volatile ushort pd[3];
00066
00067   pd[0] = size-1;
00068   pd[1] = (uint)p;
00069   pd[2] = (uint)p >> 16;
00070
00071   asm volatile("lgdt (%0)" : : "r" (pd));
00072 }
00073
00074 struct gatedesc;
00075
00076 static inline void
00077 lidt(struct gatedesc *p, int size)
00078 {
00079   volatile ushort pd[3];
00080
00081   pd[0] = size-1;
00082   pd[1] = (uint)p;
```

```
00083   pd[2] = (uint)p >> 16;
00084
00085   asm volatile("lidt (%0)" : : "r" (pd));
00086 }
00087
00088 static inline void
00089 ltr(ushort sel)
00090 {
00091   asm volatile("ltr %0" : : "r" (sel));
00092 }
00093
00094 static inline uint
00095 readeflags(void)
00096 {
00097   uint eflags;
00098   asm volatile("pushfl; popl %0" : "=r" (eflags));
00099   return eflags;
00100 }
00101
00102 static inline void
00103 loadgs(ushort v)
00104 {
00105   asm volatile("movw %0, %%gs" : : "r" (v));
00106 }
00107
00108 static inline void
00109 cli(void)
00110 {
00111   asm volatile("cli");
00112 }
00113
00114 static inline void
00115 sti(void)
00116 {
00117   asm volatile("sti");
00118 }
00119
00120 static inline uint
00121 xchg(volatile uint *addr, uint newval)
00122 {
00123   uint result;
00124
00125   // The + in "+m" denotes a read-modify-write operand.
00126   asm volatile("lock; xchgl %0, %1" :
00127                "+m" (*addr), "=a" (result) :
00128                "1" (newval) :
00129                "cc");
00130   return result;
00131 }
00132
00133 static inline uint
00134 rcr2(void)
00135 {
00136   uint val;
00137   asm volatile("movl %%cr2,%0" : "=r" (val));
00138   return val;
00139 }
00140
00141 static inline void
00142 lcr3(uint val)
00143 {
00144   asm volatile("movl %0,%%cr3" : : "r" (val));
00145 }
00146
00147 //PAGEBREAK: 36
00148 // Layout of the trap frame built on the stack by the
00149 // hardware and by trapasm.S, and passed to trap().
00150 struct trapframe {
00151   // registers as pushed by pusha
00152   uint edi;
00153   uint esi;
00154   uint ebp;
00155   uint oesp;      // useless & ignored
00156   uint ebx;
00157   uint edx;
00158   uint ecx;
00159   uint eax;
00160
00161   // rest of trap frame
00162   ushort gs;
00163   ushort padding1;
00164   ushort fs;
00165   ushort padding2;
00166   ushort es;
00167   ushort padding3;
00168   ushort ds;
00169   ushort padding4;
```

```
00170   uint trapno;
00171
00172   // below here defined by x86 hardware
00173   uint err;
00174   uint eip;
00175   ushort cs;
00176   ushort padding5;
00177   uint eflags;
00178
00179   // below here only when crossing rings, such as from user to kernel
00180   uint esp;
00181   ushort ss;
00182   ushort padding6;
00183 };
```

5.254 xv6.dox File Reference

5.255 zombie.c File Reference

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

Functions

- int main (void)

5.255.1 Function Documentation

5.255.1.1 main()

```
int main (
              void  )
```

Definition at line 9 of file zombie.c.

```
00010 {
00011   if(fork() > 0)
00012     sleep(5);  // Let child exit before parent.
00013   exit();
00014 }
```

5.256 zombie.c

Go to the documentation of this file.

```
00001 // Create a zombie process that
00002 // must be reparented at exit.
00003
00004 #include "types.h"
00005 #include "stat.h"
00006 #include "user.h"
00007
00008 int
00009 main(void)
00010 {
00011   if(fork() > 0)
00012     sleep(5);  // Let child exit before parent.
00013   exit();
00014 }
```

5.257 zombie.d File Reference

5.258 zombie.d

Go to the documentation of this file.

```
00001 zombie.o: zombie.c /usr/include/stdc-predef.h types.h stat.h user.h
```

Index