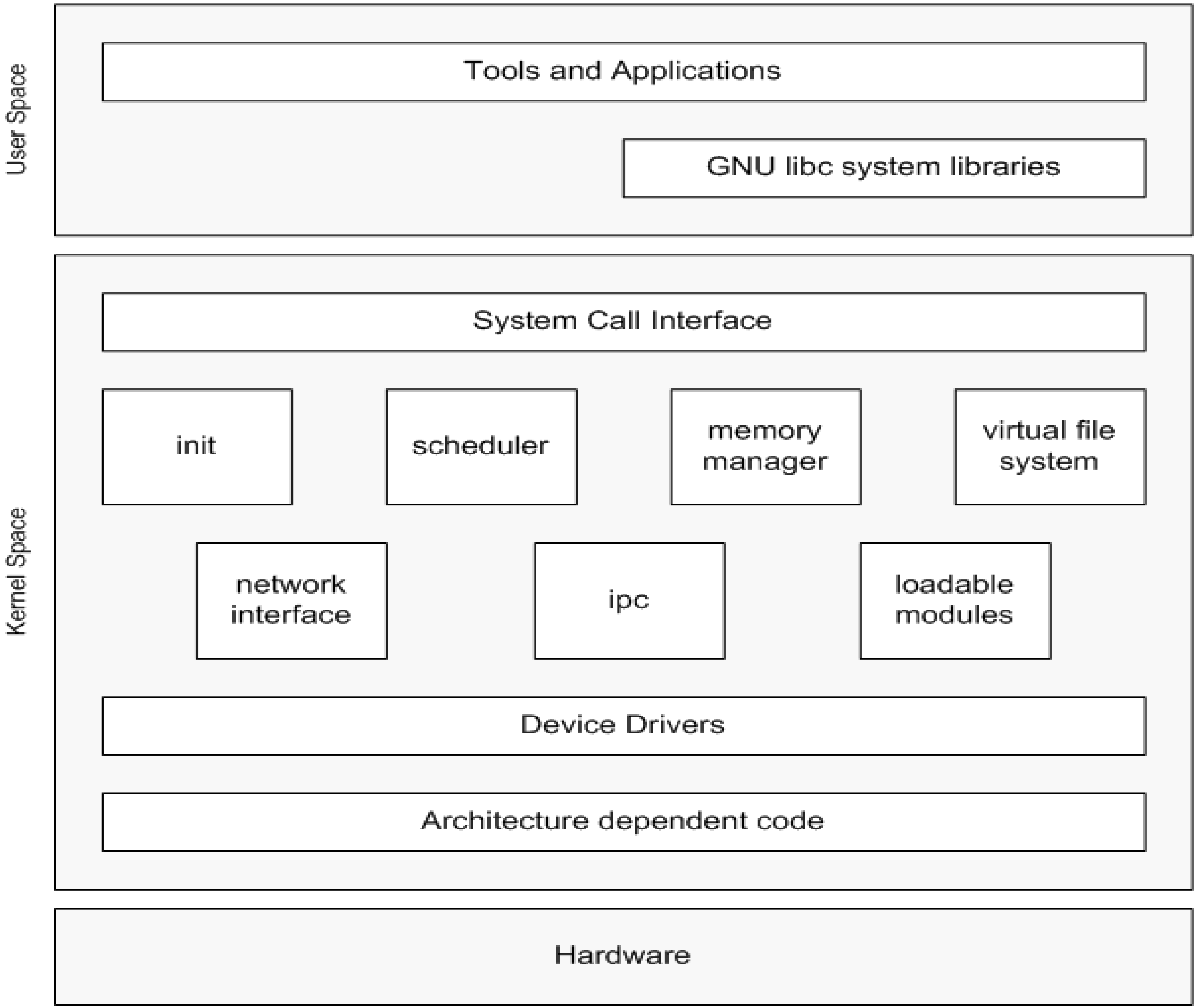
Unix/Linux Primer

- OS knowledge assumed
- experience on GPOS/RTOS assumed
- Introduction to Unix/Linux Architecture will be covered in this session
- More details will be covered when we discuss specific interfaces
- Main focus will be on Linux

Monolithic kernel Architecture

- It is a large kernel composed of several logically different components put together to form a large program image for the kernel
- Supports kernel module loading support for run-time extension of the kernel
- Most Unix variants are of this type
- Mac OS X is one exception


User Space
Tools and Applications
GNU libc system libraries
Kernel Space
System Call Interface
init
scheduler
memory manager
virtual file system
network interface
ipc
loadable modules
Device Drivers
Architecture dependent code
Hardware

System call wrappers

- Machine-dependent wrappers part of glibc
- Used implement services offered by the kernel to user-space applications
- Software generated interrupt
- Mode-switch to kernel-space and back to user-space
- Examples : fork(),read(),write(),pipe(),....

Library function calls

- Not system call wrappers
- May or may not use system call wrappers
- Examples : printf(), fopen(), fread(), strncpy(),...
- More convenient to use but less powerful
- Both system calls wrappers and Library function calls are taken by the Standards discussed in the previous sections

System call wrappers/Interrupts/Exceptions

- All of them use similar mechanisms (subtle differences exist)
- All them end up calling the kernel
- Kernel executes in the context of the process that is currently executing when the event occurs
- Exceptions take care of FATAL program errors (invalid memory access,illegal instruction,.....)

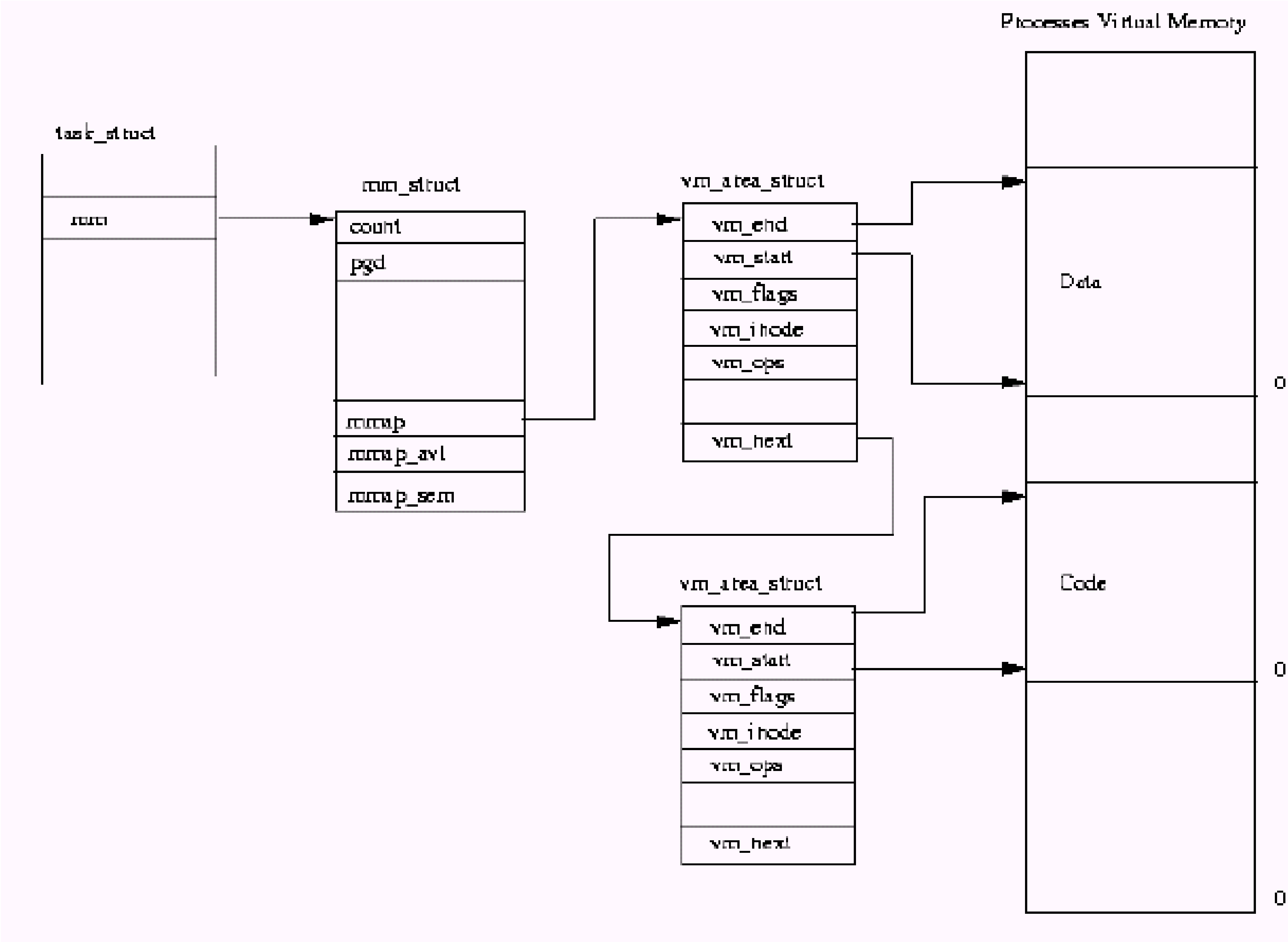
Process model

- Process descriptors(PD) – also known as task structures in Linux represent a process in the system
- PD contains only the most essential information
- Most resources allocated to the process are represented through other add-on data-structures
- Multi-tasking
- Independent and isolated (virtual)address spaces

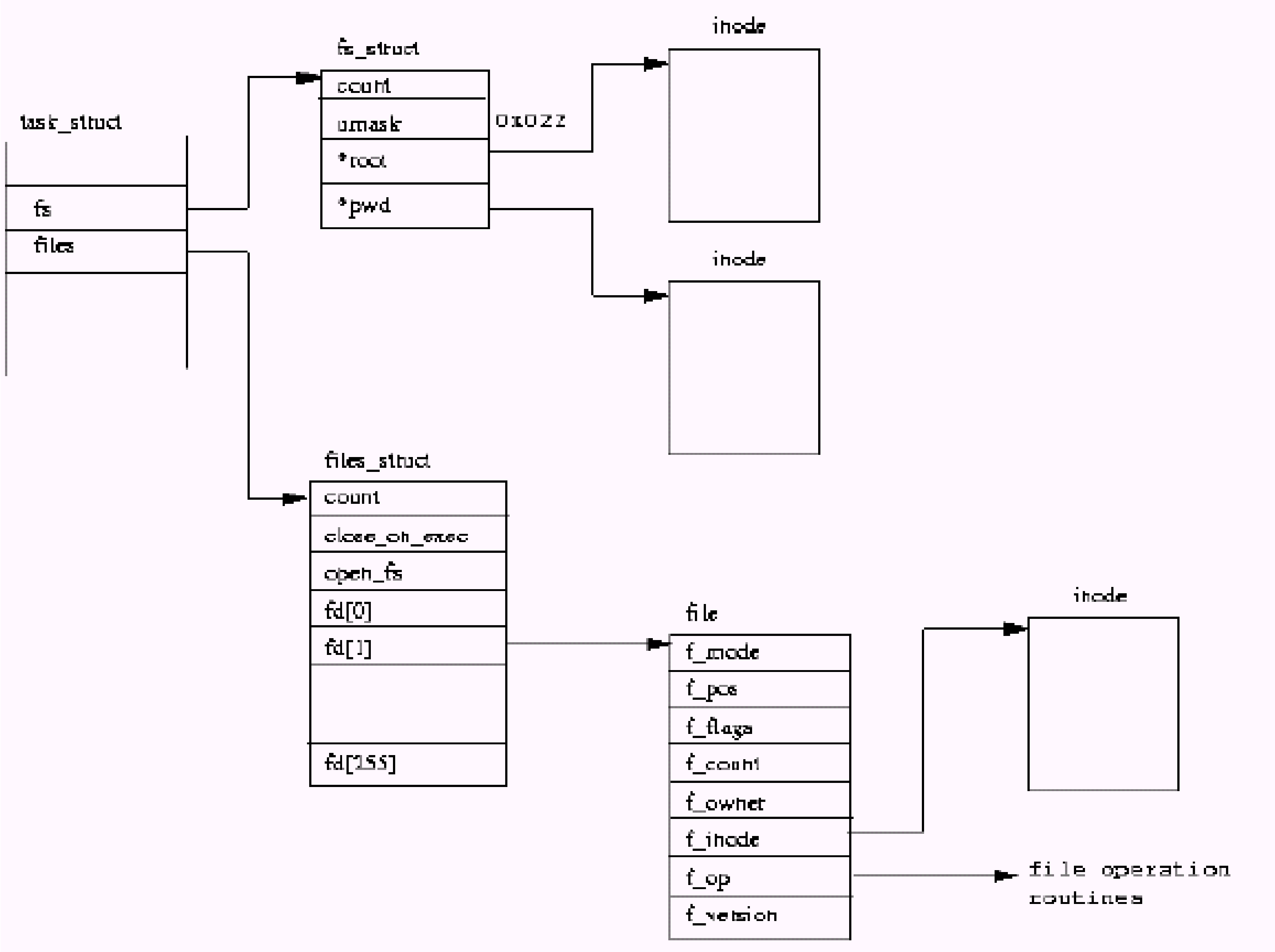
Process model

- Each process is uniquely identified by PID
- Parent – child relationship is maintained
- Parent – child relationship and its importance
- Process states
- Init process with PID = 1
- Zombie processes
- Orphaned processes and Init process

```
struct task_struct{
      volatile long            state;
      long                     counter, priority;
      struct task_struct       *next_task, *prev_task;
      struct task_struct       *next_run, *prev_run;
      int                      exit_code, exit_signal;
      int                      pid;
      struct task_struct       *p_opptr, *p_cptr;
      struct wait_queue        *wait_chldexit;
      struct task_struct       *pidhash_next;
      unsigned long            policy;
      struct tms               times;
      unsigned long            start_time;
      unsigned short           uid, gid;
      struct thread_struct tss;
      struct files_struct  *files;
      struct mm_struct         *mm;
      struct signal_struct *sig;
      sigset_t                 signal, blocked;
};
```



Processes Virtual Memory
task_struct
mm
mm_struct
count
pgd
mmap
mmap_avl
mmap_sem
vm_area_struct
vm_end
vm_start
vm_flags
vm_inode
vm_ops
vm_next
vm_area_struct
vm_end
vm_start
vm_flags
vm_inode
vm_ops
vm_next
Data
Code


task_struct
fs
files
fs_struct
count
umask
*root
*pwd
0x022
inode
inode
files_struct
count
close_on_exec
open_fs
fd[0]
fd[1]
fd[255]
file
f_mode
f_pos
f_flags
f_count
f_owner
f_inode
f_op
f_version
inode
file operation
routines

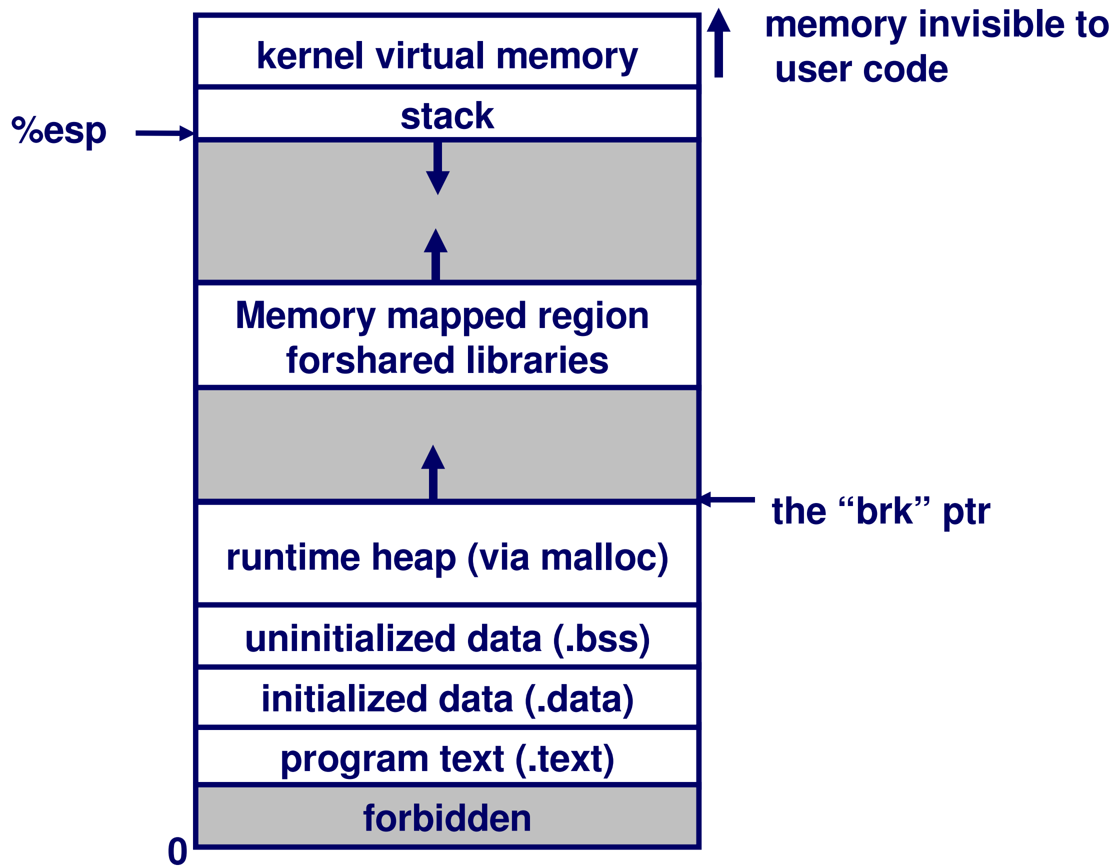
Process Address Space

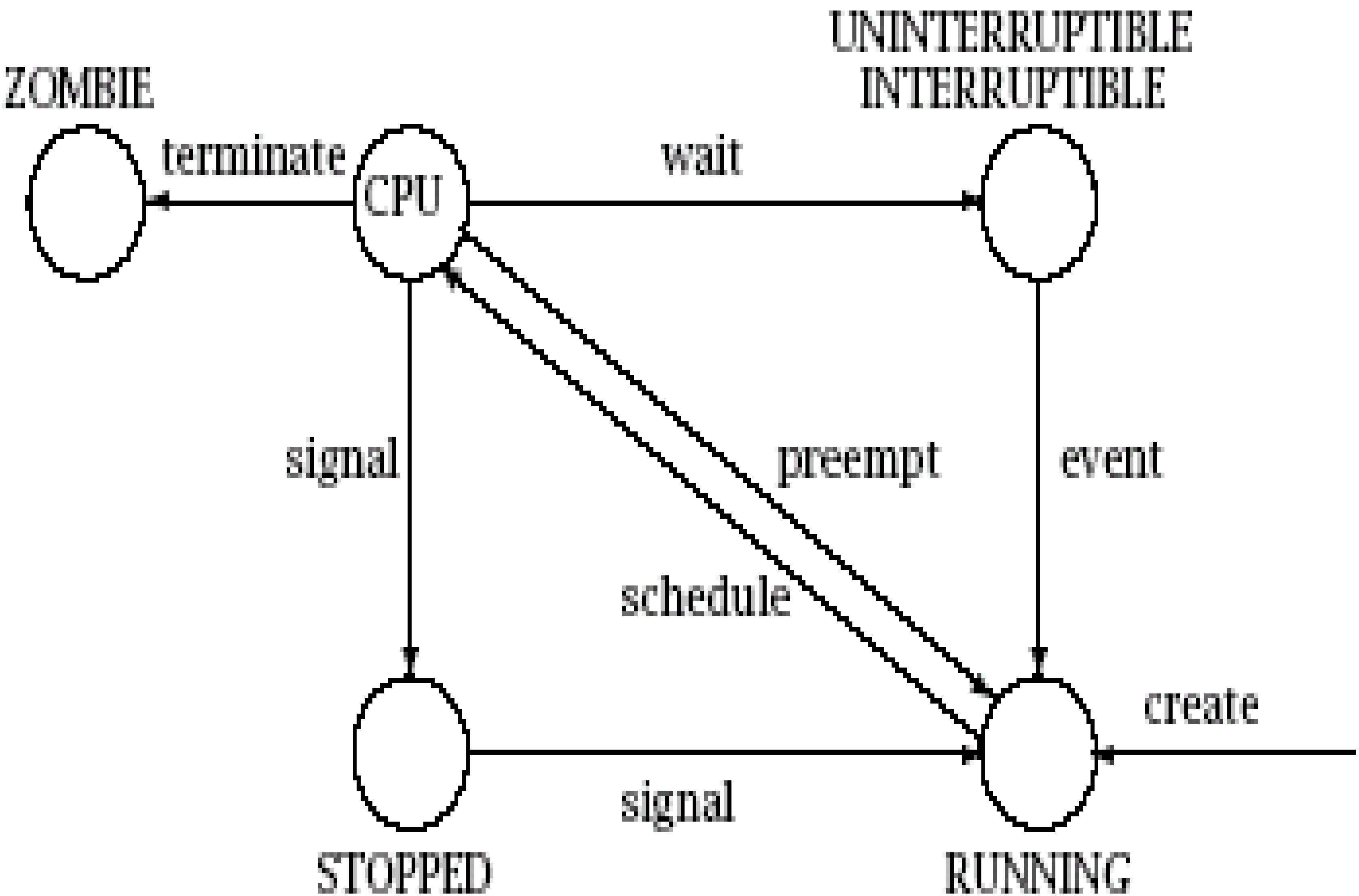
Multiple processes can reside in physical memory.

How do we resolve address conflicts?

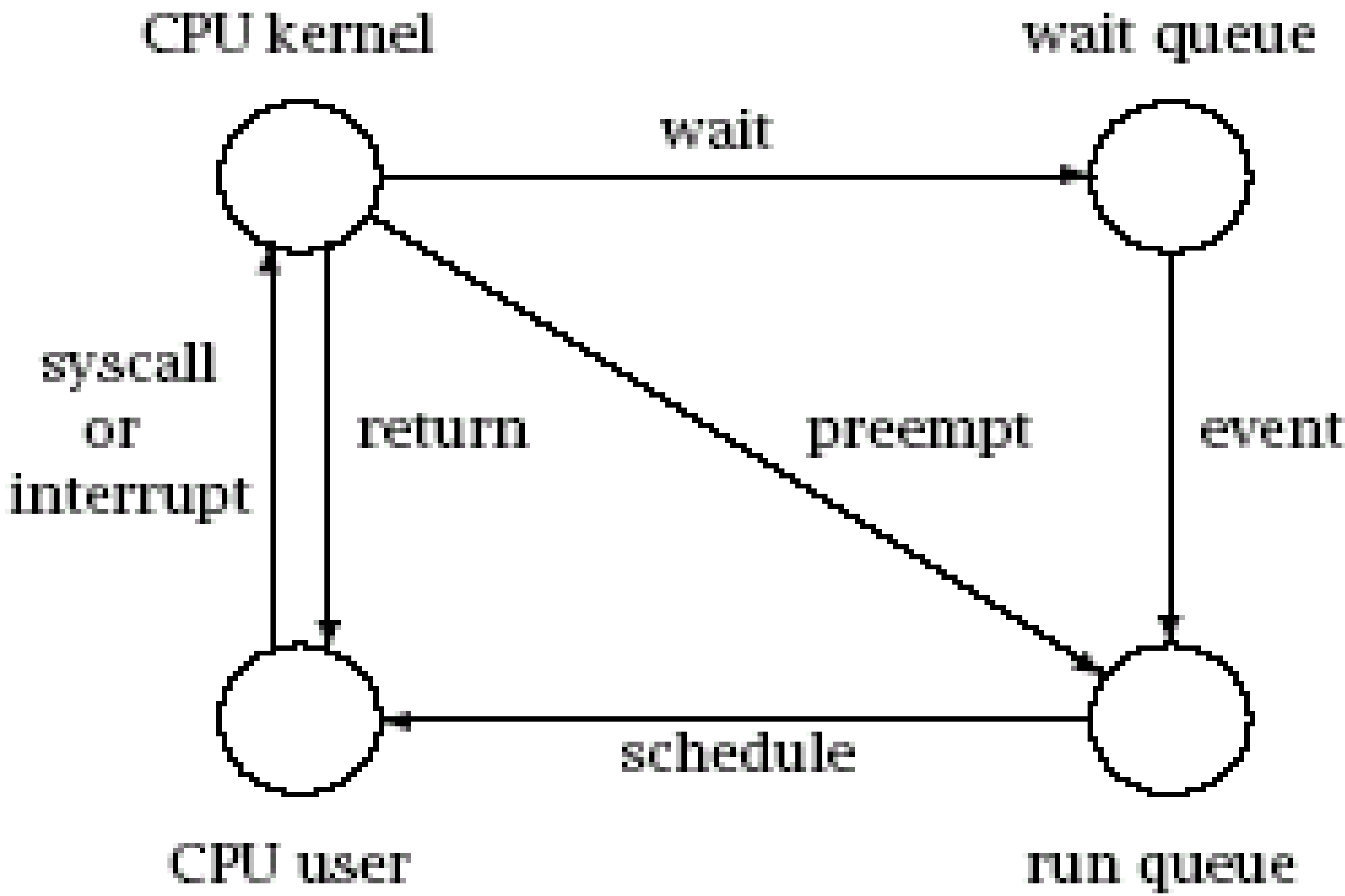
- what if two processes access something at the same address?

Linux/x86 process memory image





ZOMBIE
terminate
CPU
wait
UNINTERRUPTIBLE
INTERRUPTIBLE
signal
preempt
event
schedule
create
signal
STOPPED
RUNNING


CPU kernel
wait queue
wait
syscall
or
interrupt
return
preempt
event
schedule
CPU user
run queue

CPU scheduling policies

- FAIR-SHARE / POSIX SCHED_OTHER
- FIXED-PRIORITY REAL-TIME SCHEDULER/ POSIX SCHED_FIFO
- FIXED-PRIORITY RR REAL-TIME SCHEDULER/ POSIX SCHED_RR
- The schedulers are preemptive
- The kernel is preemptive in both user-space/ kernel-space

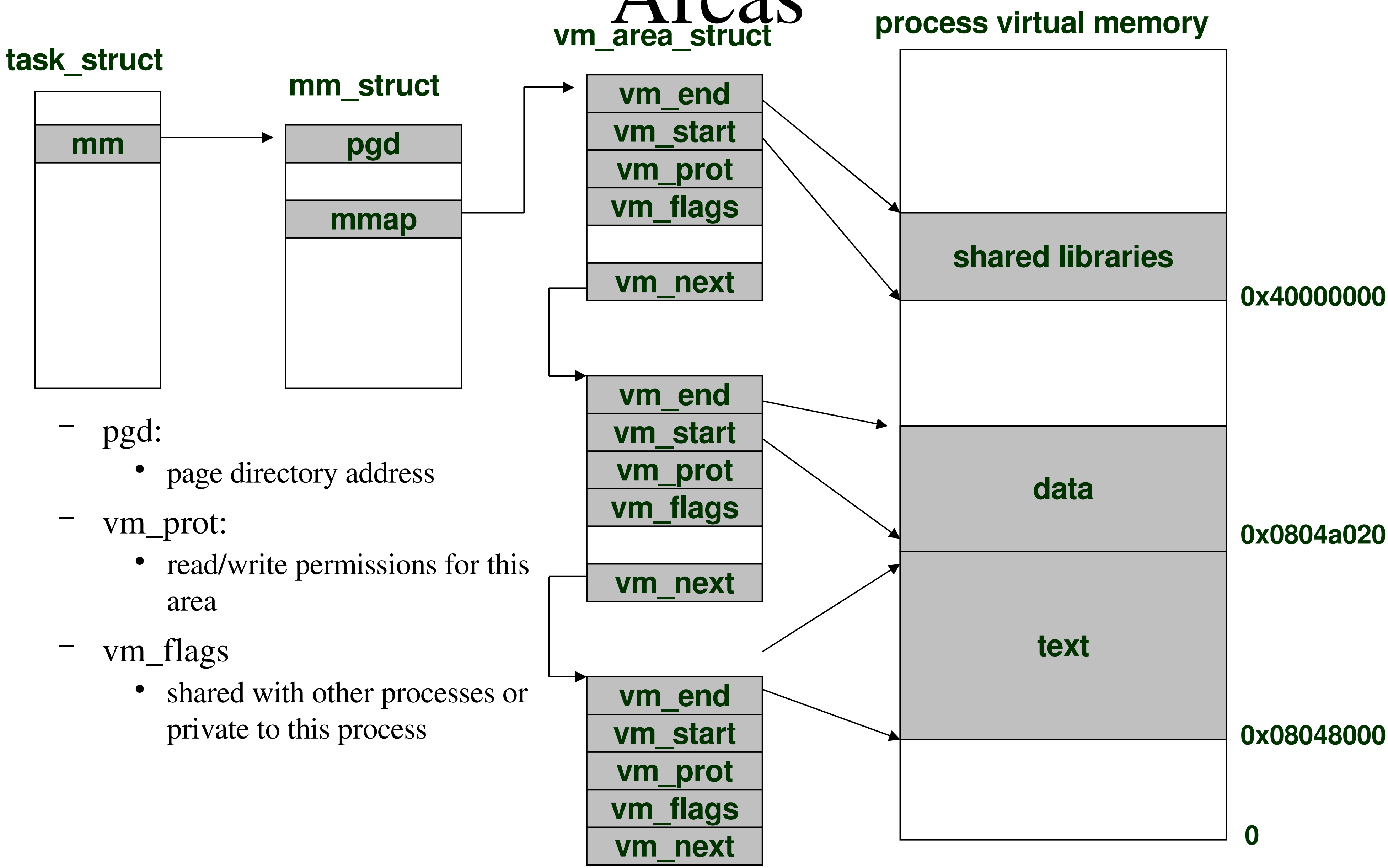
CPU scheduling policies

- Real-time priorities – 1-99
- Non real-time priorities – 100-139
- System calls can be used to adjust the policies/priorities
- A good candidate for soft real-time OS
- Timer handler runs every 1ms (used to be 10ms)

Virtual memory model

- Private address spaces
- May have shared memory areas (shared memory) with other processes
- Demand-paged virtual memory scheme
- Copy-on-write is used during process creation
- Memory area structures and page-tables describe VM for a process
- Swap partitions exist

Linux Organizes VM as Collection of "Areas"



- pgd:
  - page directory address
- vm_prot:
  - read/write permissions for this area
- vm_flags
  - shared with other processes or private to this process

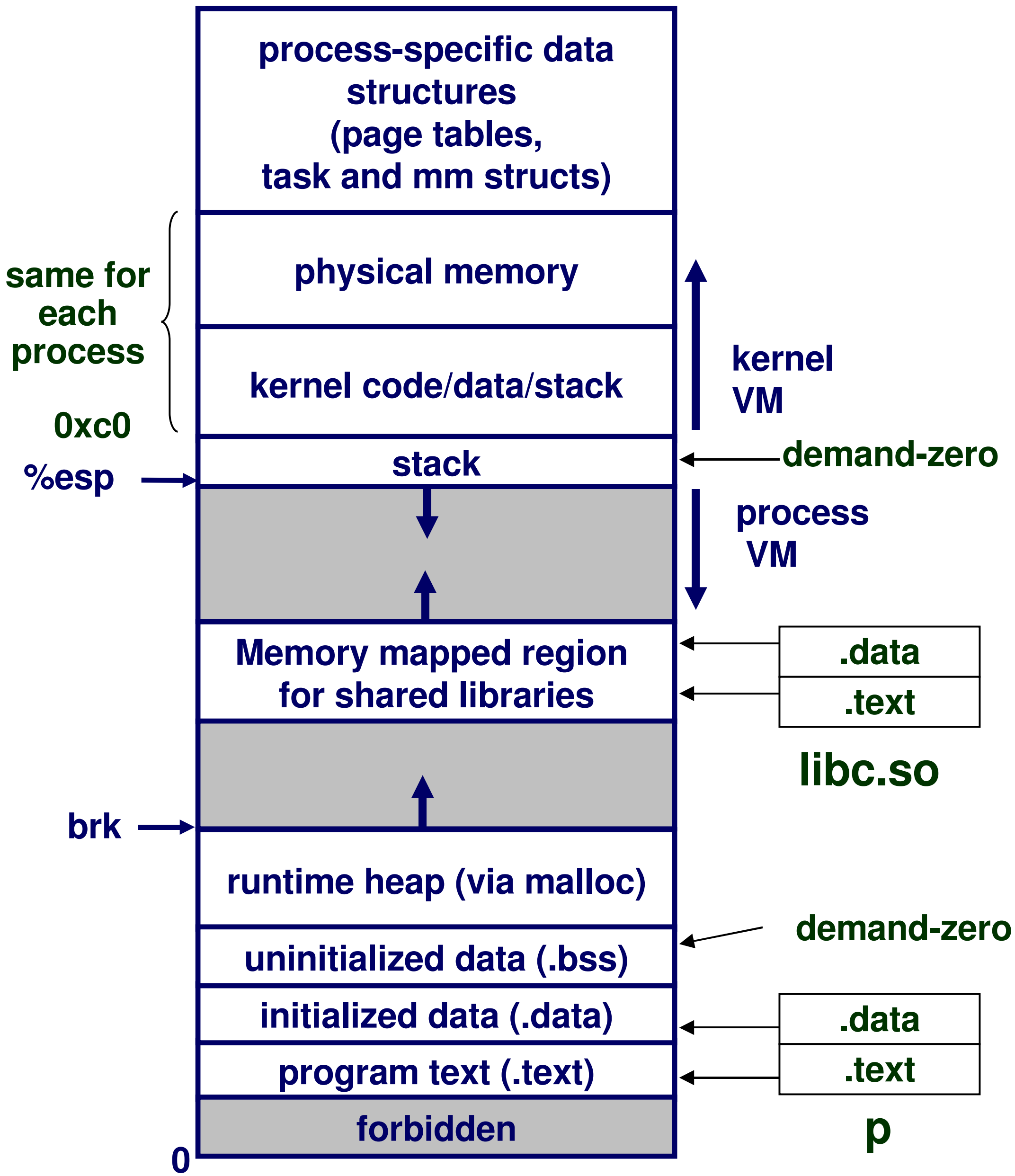
Process model revisited

- Process duplication using fork()
- Process image overlay using exec()
- Linux uses Unix mechanism for creating processes – using fork()
- Changing the currently associated program with the process – using exec()
- We will see the advantages of this during the process system calls

To create a new process using `fork()`:

- make copies of the old process's mm_struct, vm_area_struct's, and page tables.
  - at this point the two processes are sharing all of their pages.
  - How to get separate spaces without copying all the virtual pages from one space to another?
    - "copy on write" technique.
- copy-on-write
  - make pages of writeable areas read-only
  - flag vm_area_struct's for these areas as private "copy-on-write".
  - writes by either process to these pages will cause page faults.
    - fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.
- Net result:
  - copies are deferred until absolutely necessary (i.e., when one

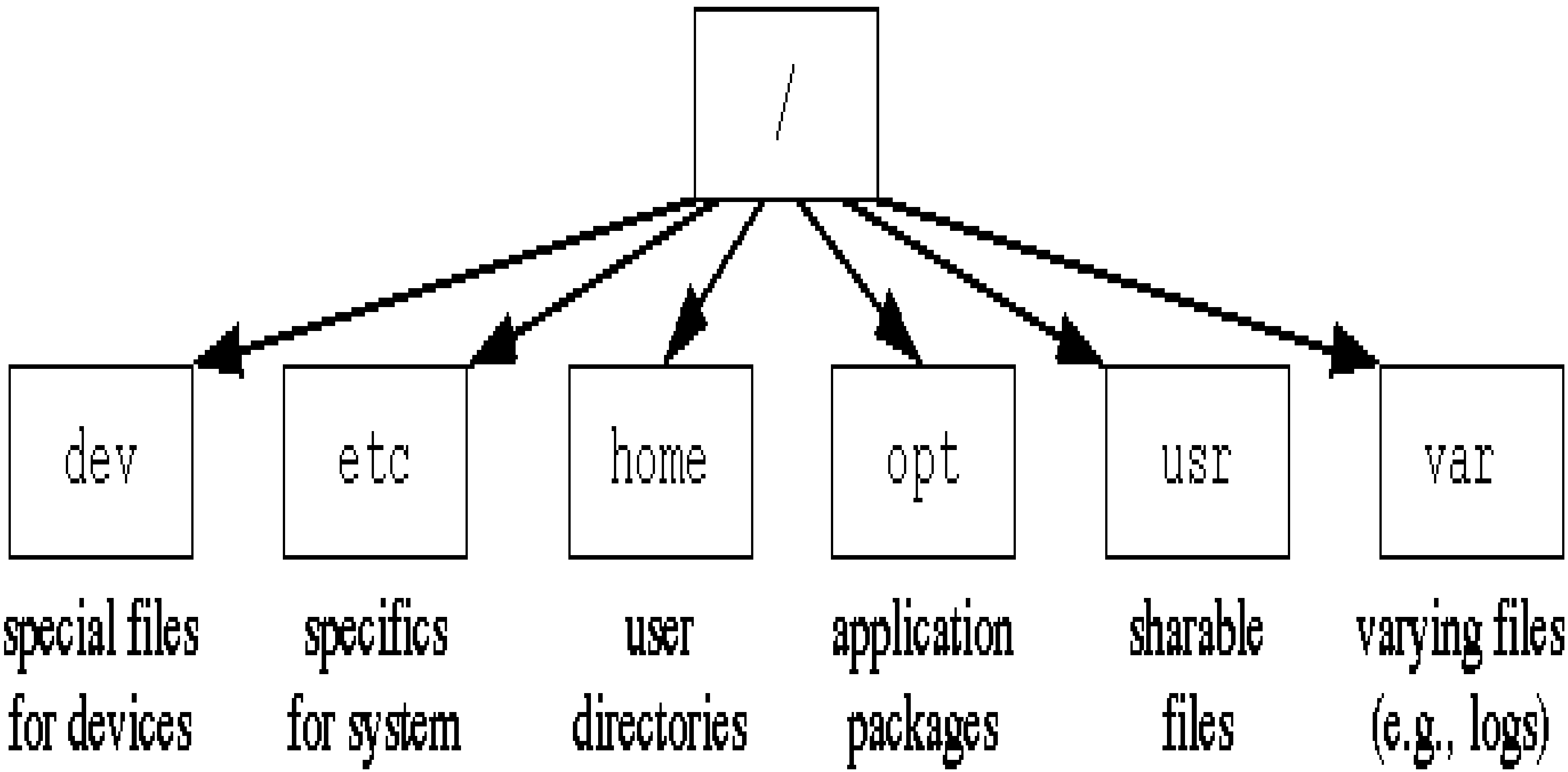
Exec() Revisited



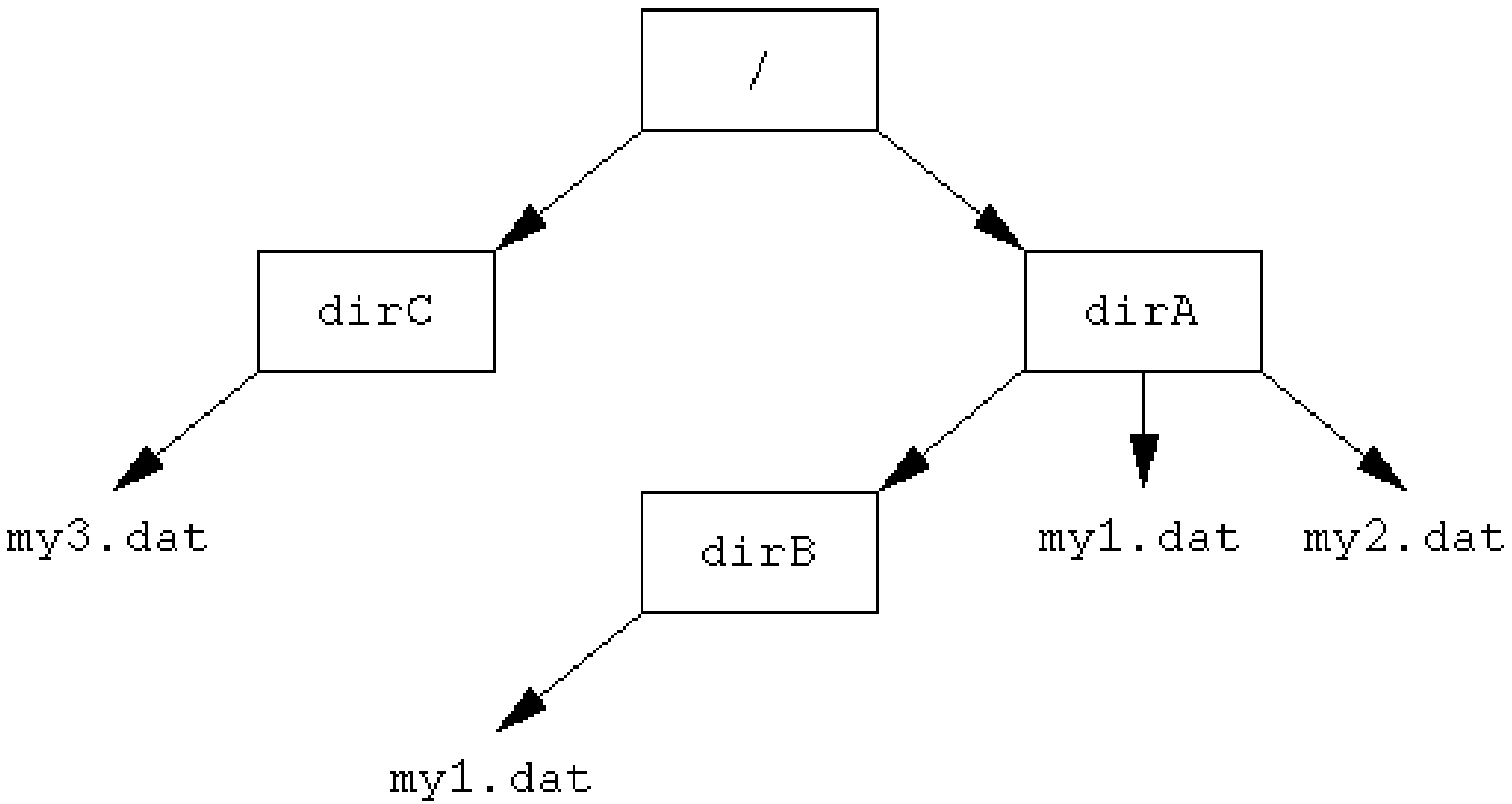
To run a new program p in the current process using `exec()`:

- free vm_area_struct's and page tables for old areas.
- create new vm_area_struct's and page tables for new areas.
  - stack, bss, data, text, shared libs.
  - text and data backed by ELF executable object file.
  - bss and stack initialized to zero.
- set PC to entry point in .text
  - Linux will swap in code and data pages as needed.

Linux File System Hierarchy



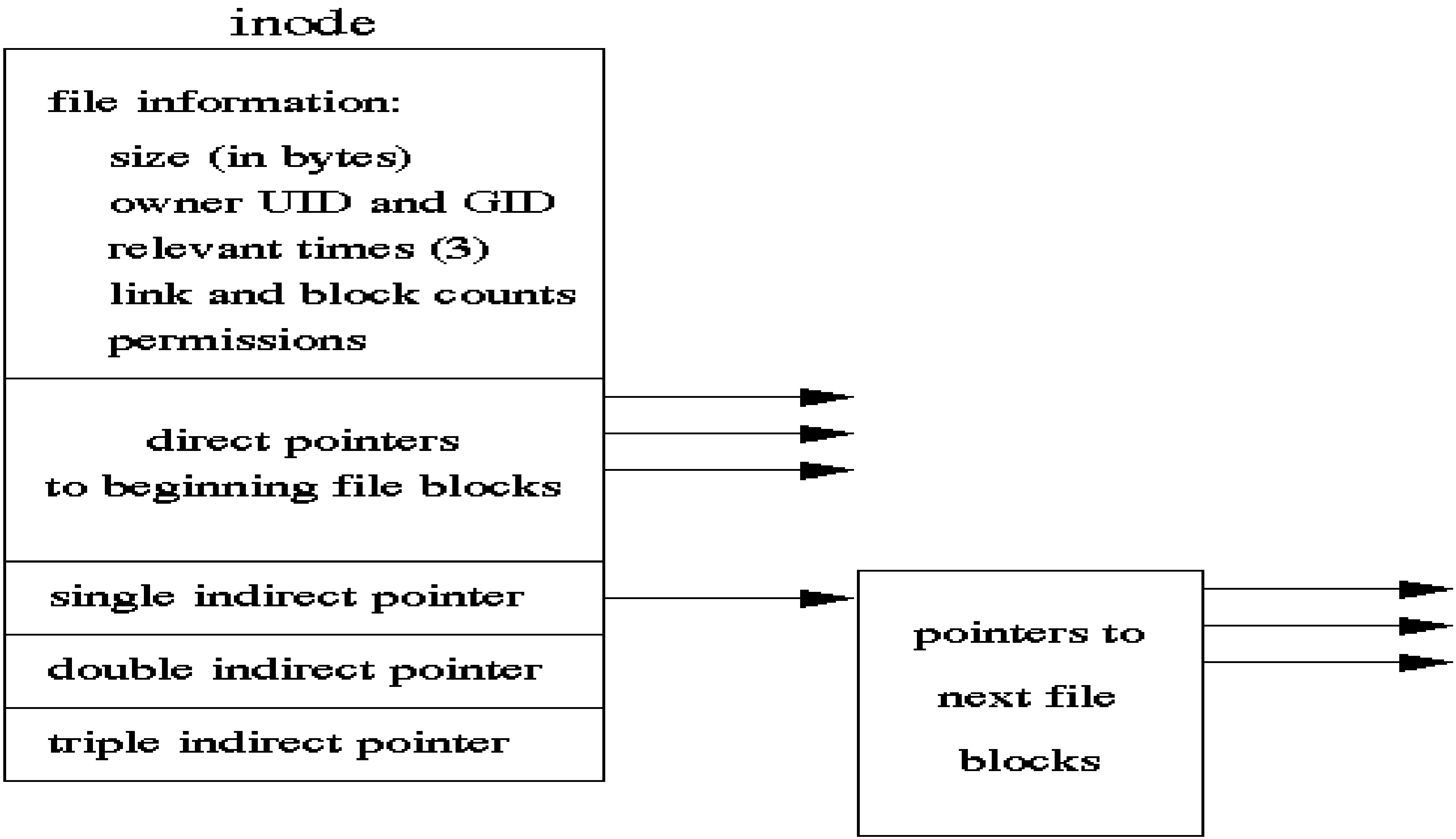
Regular Files and Directories



Linux File system

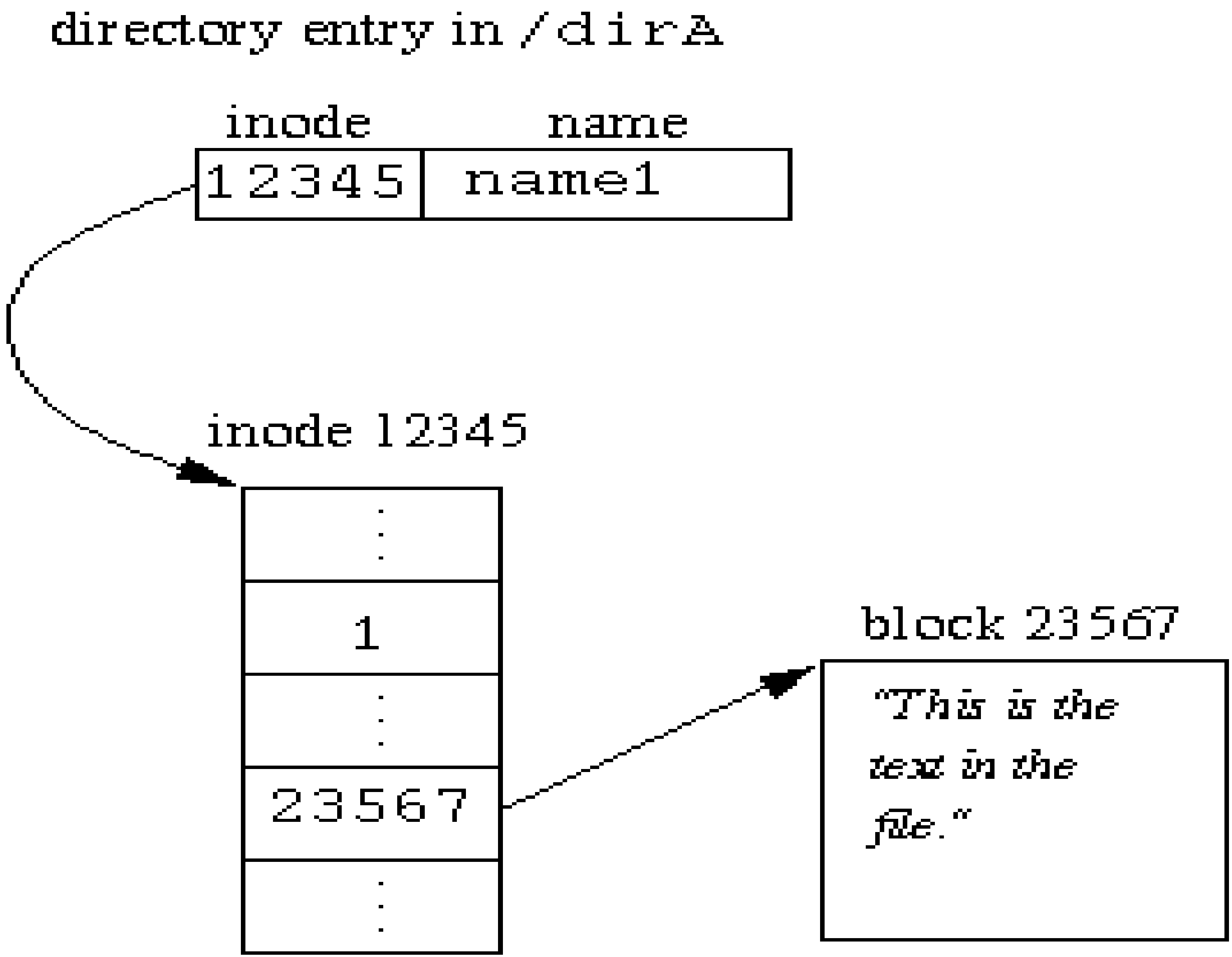
- Each file is uniquely represented by an i-node
- Regular files vs directory files
- Directory entry in the file system
- Hard-links
- Soft-links
- Hard-links vs Soft-links
- Open files and associated data-structures
- VFS – support for multiple file-systems

Linux i-node

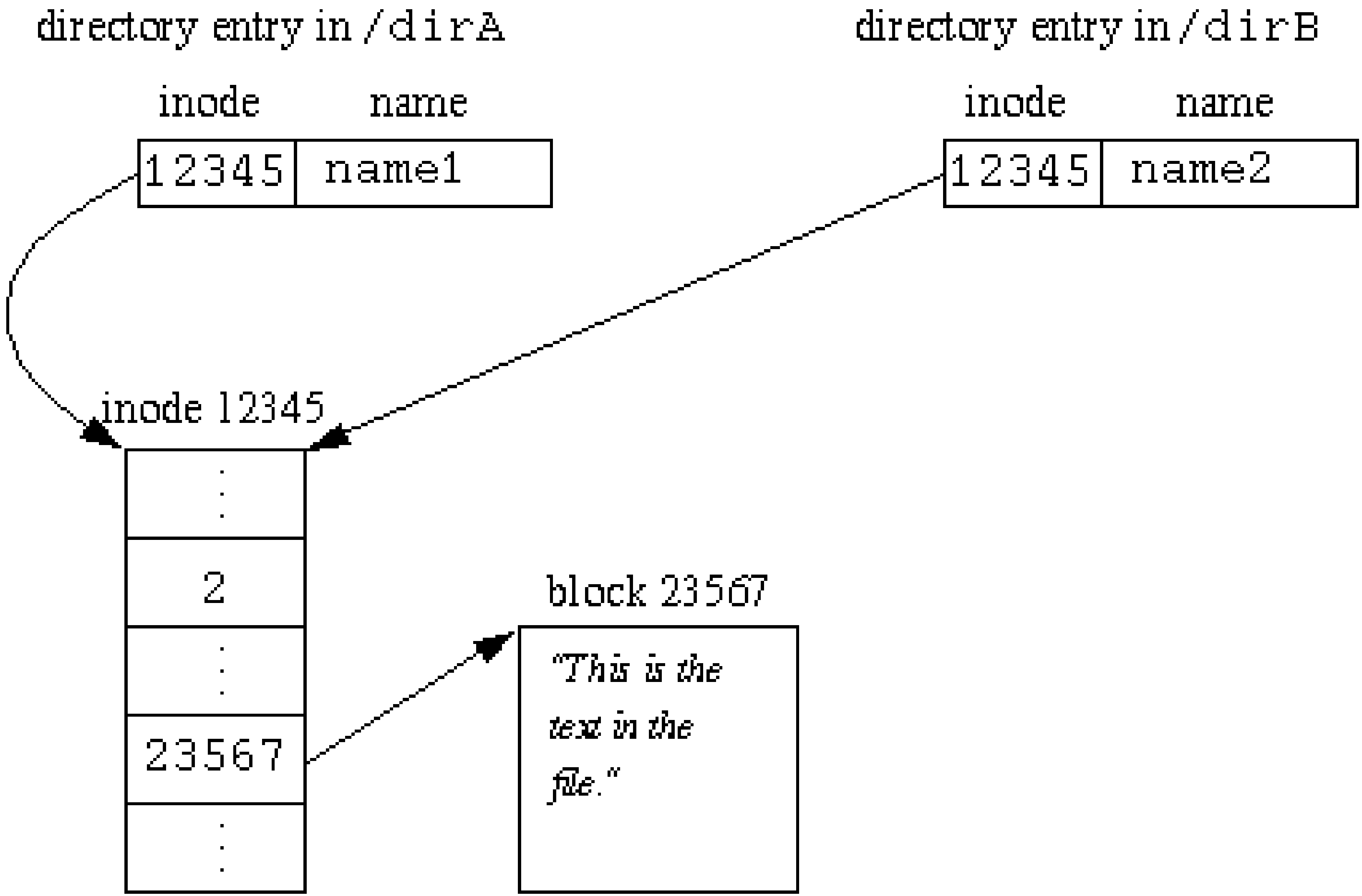

inode
file information:
size (in bytes)
owner UID and GID
relevant times (3)
link and block counts
permissions
direct pointers
to beginning file blocks
single indirect pointer
double indirect pointer
triple indirect pointer
pointers to
next file
blocks

Links

- A *link* is an association between a filename and an inode.
- UNIX has two types of links: hard and symbolic (also called soft).
- Directory entries are called hard links because they directly link filenames to inodes.
- Each inode contains a count of the number of hard links to the inode.
- When a file is created, a new directory entry is created an a new inode is assigned.
- Additional hard links can be created with
  ln newname oldname or with the link system call.
- A new hard link to an existing file creates a new directory entry but assigns no other additional disk space.
- A new hard link increments the link count in the inode.
- A hard link can be removed with the rm command or the unlink system call.
- These decrement the link count.
- The inode and associated disk space are freed when the count is decremented to 0.



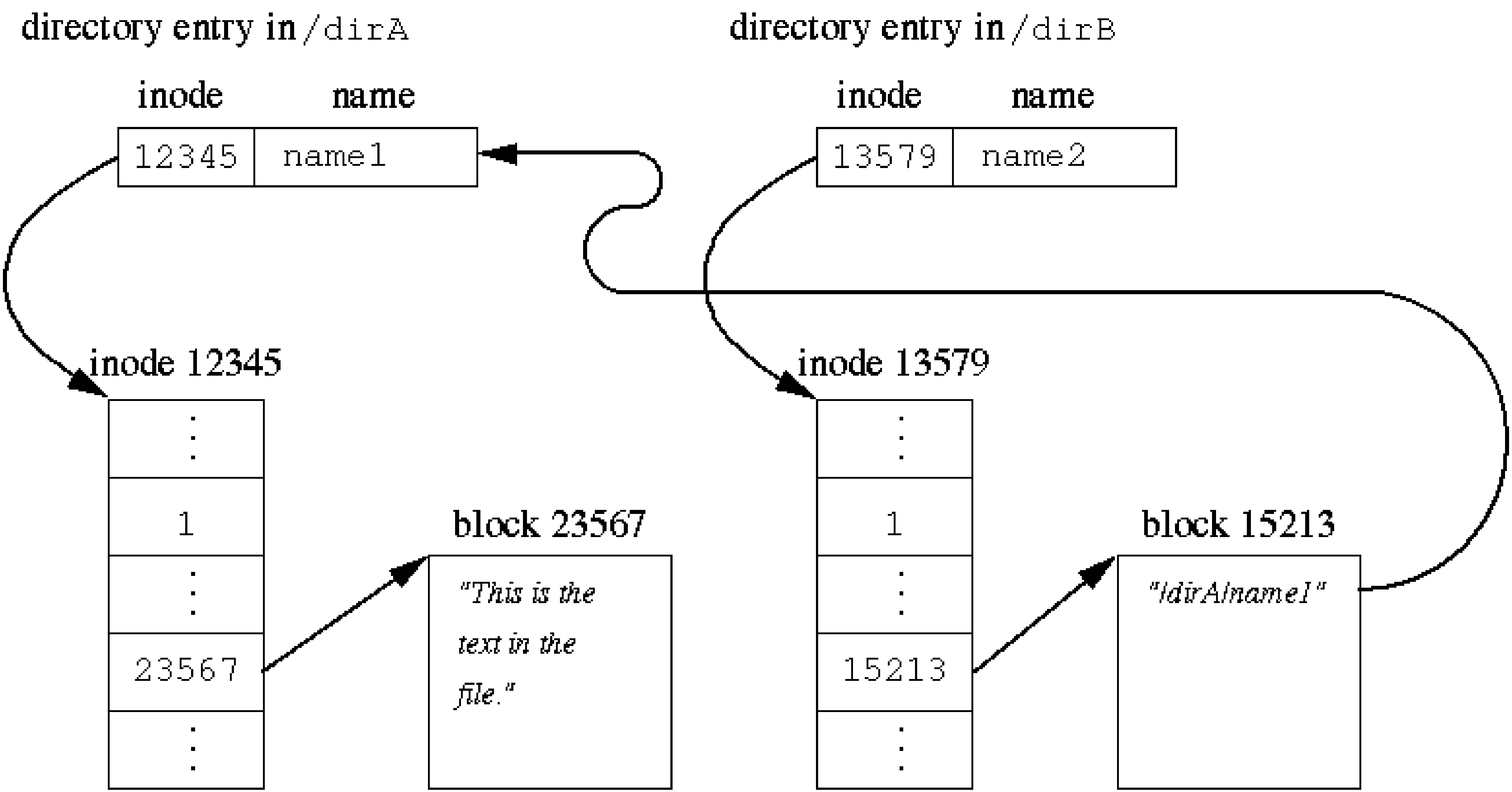
directory entry in /dirA
inode
name
12345
name1
inode 12345
1
23567
block 23567
"This is the
text in the
file."

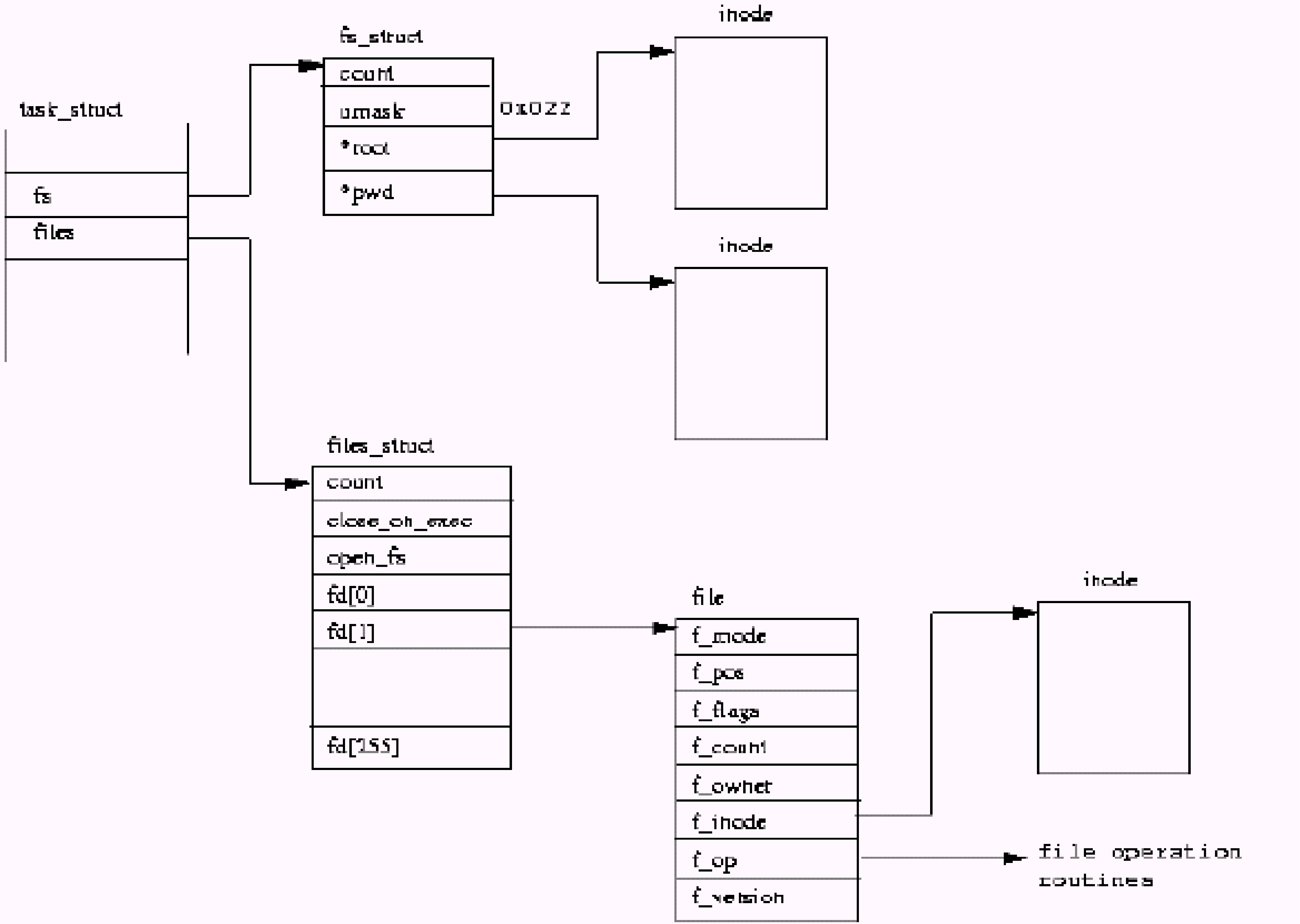

directory entry in /dirA
inode
name
12345
name1
directory entry in /dirB
inode
name
12345
name2
inode 12345
2
23567
block 23567
"This is the text in the file."

Symbolic links or soft links

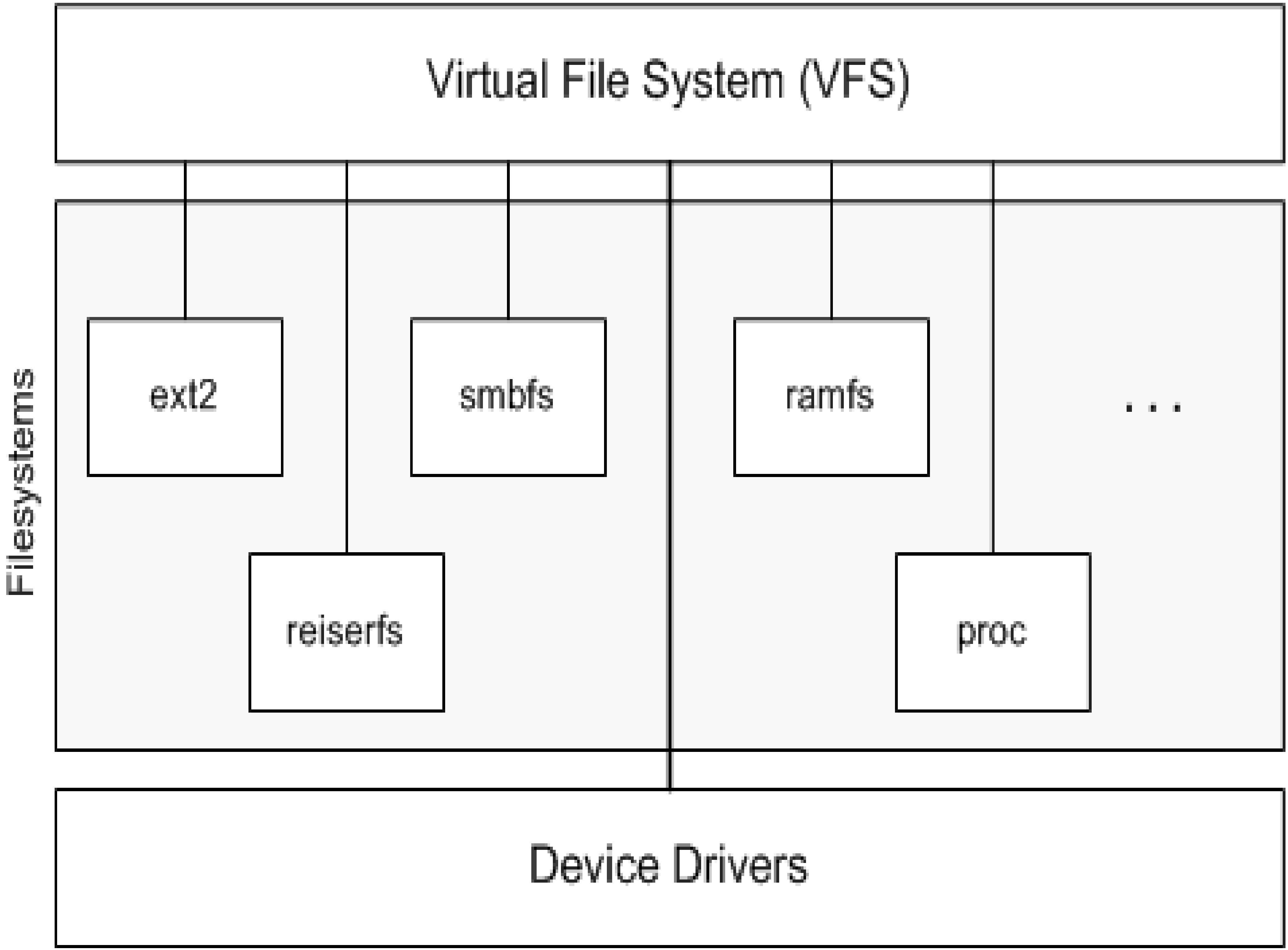
- A symbolic link is a special type of fie that contains the name of another file.
- A reference to the name of a symbolic link causes the operating system to use the name stored in the file, rather than the name itself.
- Symbolic lines are created with the command:
  ln -s newname oldname or symlink() system call
- Symbolic links do not affect the link count in the inode.
- Unlike hard links, symbolic links can span filesystems.

Symbolic links or soft links..





task_struct
fs
files
fs_struct
count
umask
0x022
*root
*pwd
inode
inode
files_struct
count
close_on_exec
open_fs
fd[0]
fd[1]
fd[255]
file
f_mode
f_pos
f_flags
f_count
f_owner
f_inode
f_op
f_version
inode
file operation
routines


Virtual File System (VFS)
Filesystems
ext2
smbfs
ramfs
. . .
reiserfs
proc
Device Drivers

IPCS

- Signals
- Pipes
- FIFOs
- Message queues
- Semaphores
- Shared memory

Signals :

A signal is *generated* when the event that causes the signal occurs.

- **A signal is *delivered* when the process takes action based on the signal.**
- **The *lifetime* of a signal is the interval between its generation and delivery.**
- **A signal that has been generated but not yet delivered is *pending*.**
- **A process *catches* a signal if it executes a *signal handler* when the signal is delivered.**
- **Alternatively, a process can *ignore* as signal when it is delivered, that is to take no action.**
- **The function sigaction is used to specify what is to happen to a signal when it is delivered.**
- **The *signal mask* determines the action to be taken when the signal is generated. It contains a list of signals to be *blocked*.**
- **A *blocked* signal is not delivered to a process until it is unblocked.**
- **The function sigprocmask is used to modify the signal mask.**
- **Each signal has a *default action* which is usually to terminate.**

```
#define SIGINT   2  /* interrupt, generated from terminal */
#define SIGILL   4  /* illegal instruction                */
#define SIGABRT  6  /* abort process                      */
#define SIGFPE   8  /* floating point exception           */
#define SIGKILL  9  /* kill a process                     */
#define SIGUSR1 10  /* user defined signal 1              */
#define SIGSEGV 11  /* segmentation violation             */
#define SIGUSR2 12  /* user defined signal 2              */
#define SIGALRM 14  /* alarm clock timeout                */
#define SIGCHLD 17  /* sent to parent on child exit       */
#define SIGXCPU 24  /* cpu time limit exceeded            */
```

Sending signals from command line

- You can send a signal to a process from the command line using kill
- kill -l will list the signals the system understands
- kill [-signal] pid will send a signal to a process.
- The optional argument may be a name or a number.The default is SIGTERM.
- To unconditionally kill a process, use:
  kill -9 pid which is
  kill -KILL pid.

Pipes

- Pipes are kernel buffers used to communicate between related processes
- They have a pipe data-structure and a kernel buffer of fixed-size
- Unstructured / Stream-oriented
- Typically, uni-directional (exceptions do occur)
- Uses the open file model described earlier
- System call pipe() creates a pipe and provides

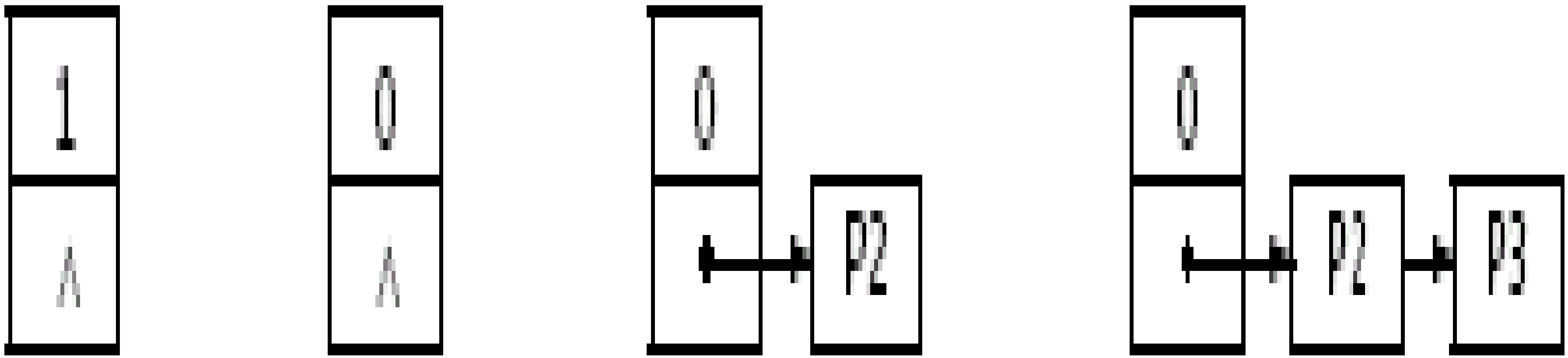
  2 file descriptors for read and write

FIFOs

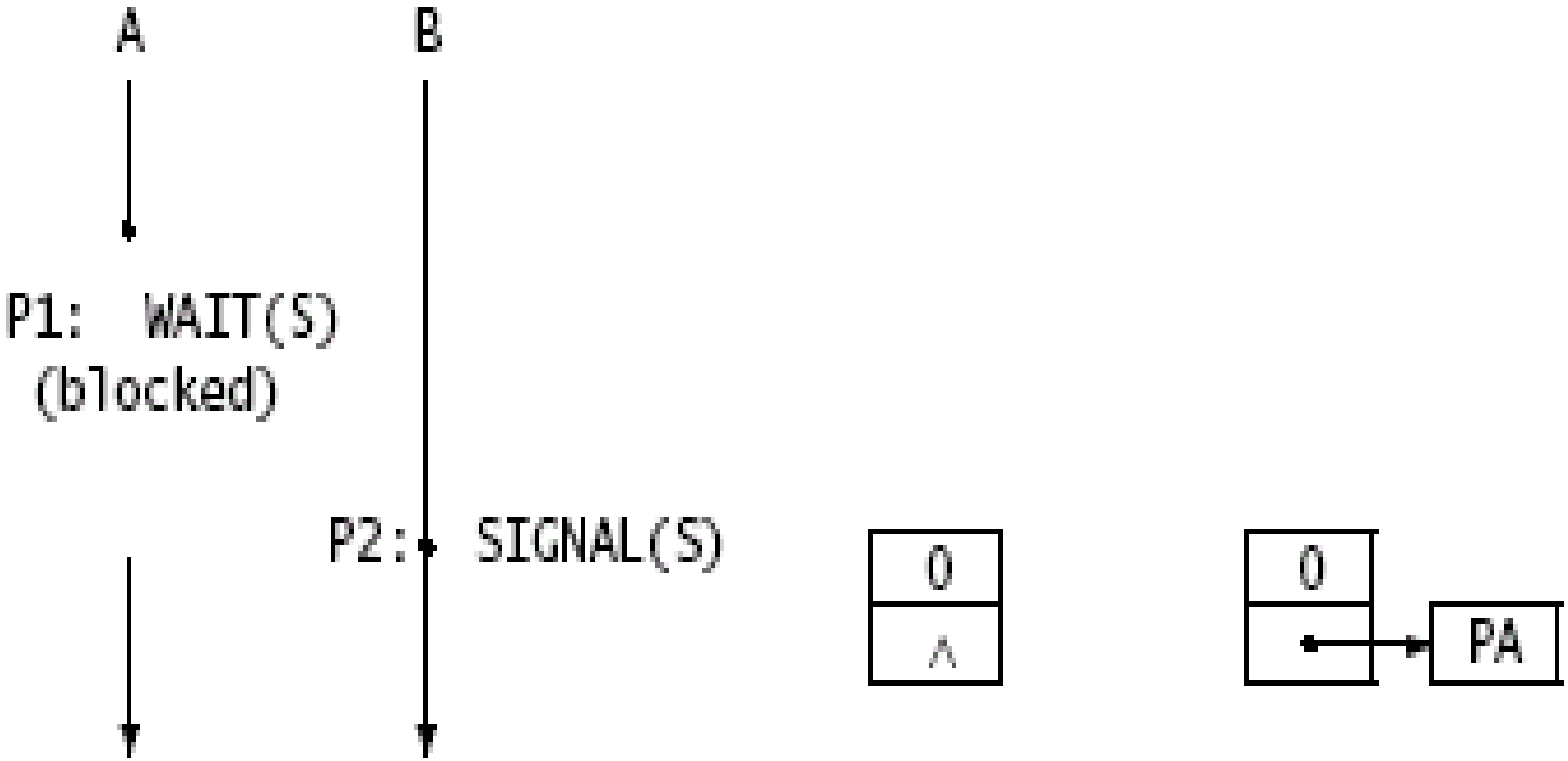
- FIFOs are kernel buffers used to communicate between related and unrelated processes
- The have a pipe data-structure and a kernel buffer of fixed-size
- Unstructured / Stream-oriented
- Typically, uni-directional
- Uses the open file model described earlier
- System call mkfifo() creates a FIFO file on-disk and must be explicitly opened for read and write

SEMAPHORE

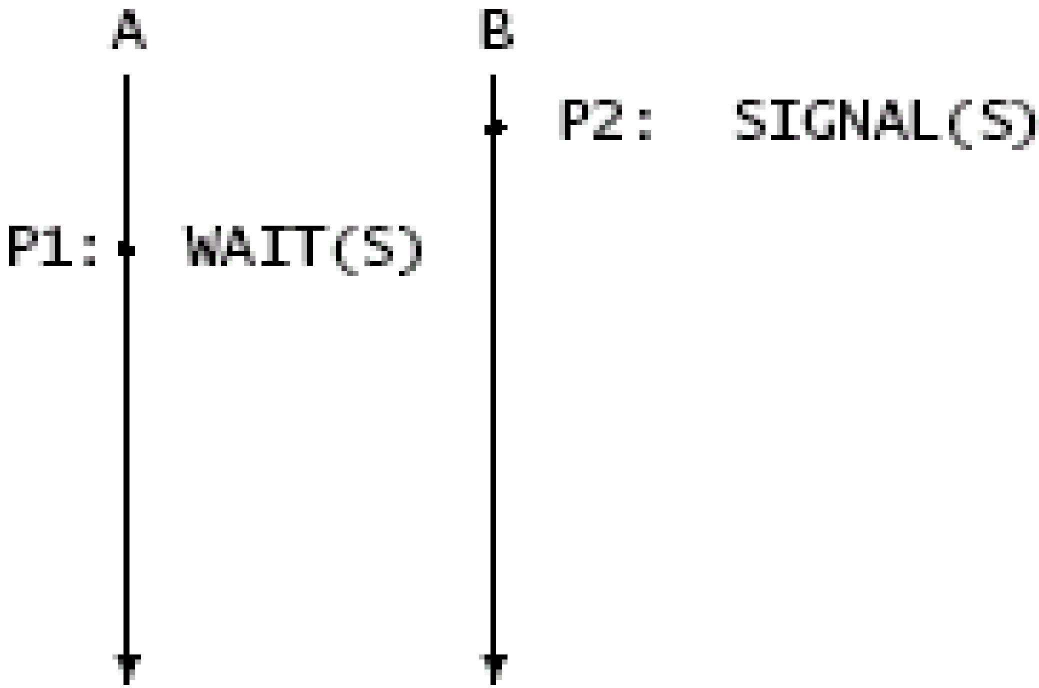
```
beginning section

WAIT(Guard)
critical section
SIGNAL(Guard)

remainder section
```



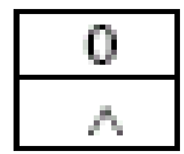
(a) (b) (c) (d)



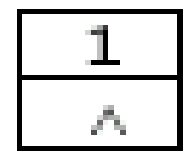
(a) (b) (c)



(a)

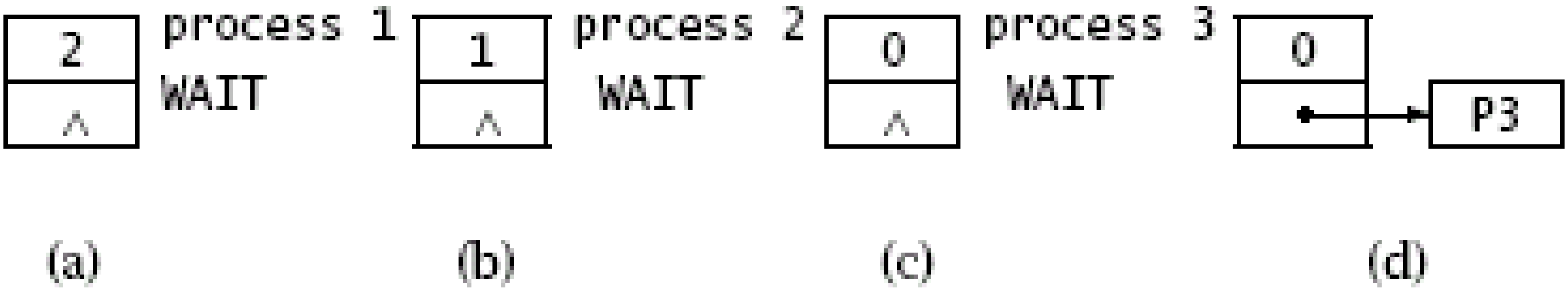


(b)



(c)

COUNTING SEMAPHORE



System V semaphore

```
struct semid_ds{
        struct ipc_perm  sem_perm;
        struct sem       *sem_base;
        struct sem_queue *sem_pending;
};
```

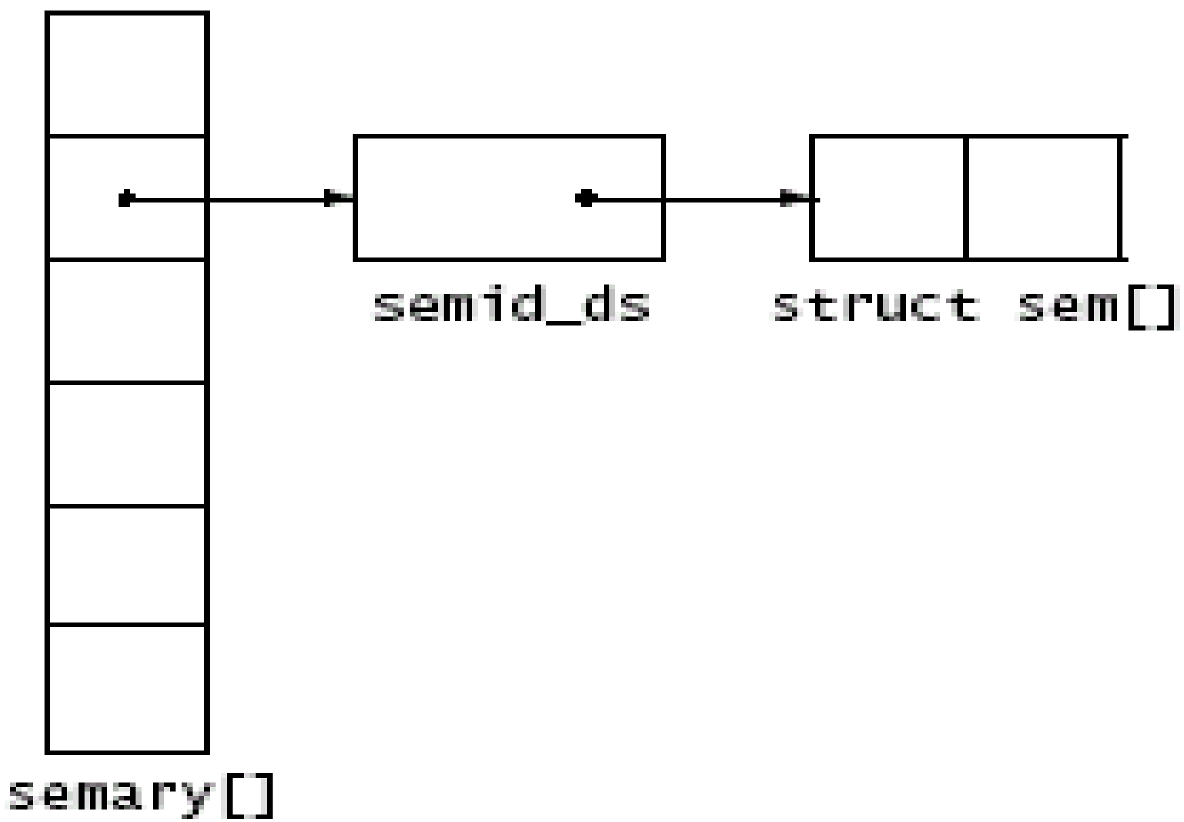
System V semaphores ..

```
struct ipc_perm{
        kernel_key_t  key;  /* user supplied key */
        kernel_uid_t  uid;  /* owner's user id   */
        kernel_gid_t  gid;  /* owner's group id  */
        kernel_mode_t mode; /* access modes      */
};
```

System V semaphore …

```
struct sem{
      int semval; /* current value                              */
      int sempid; /* process which last operated on sem */
};
```

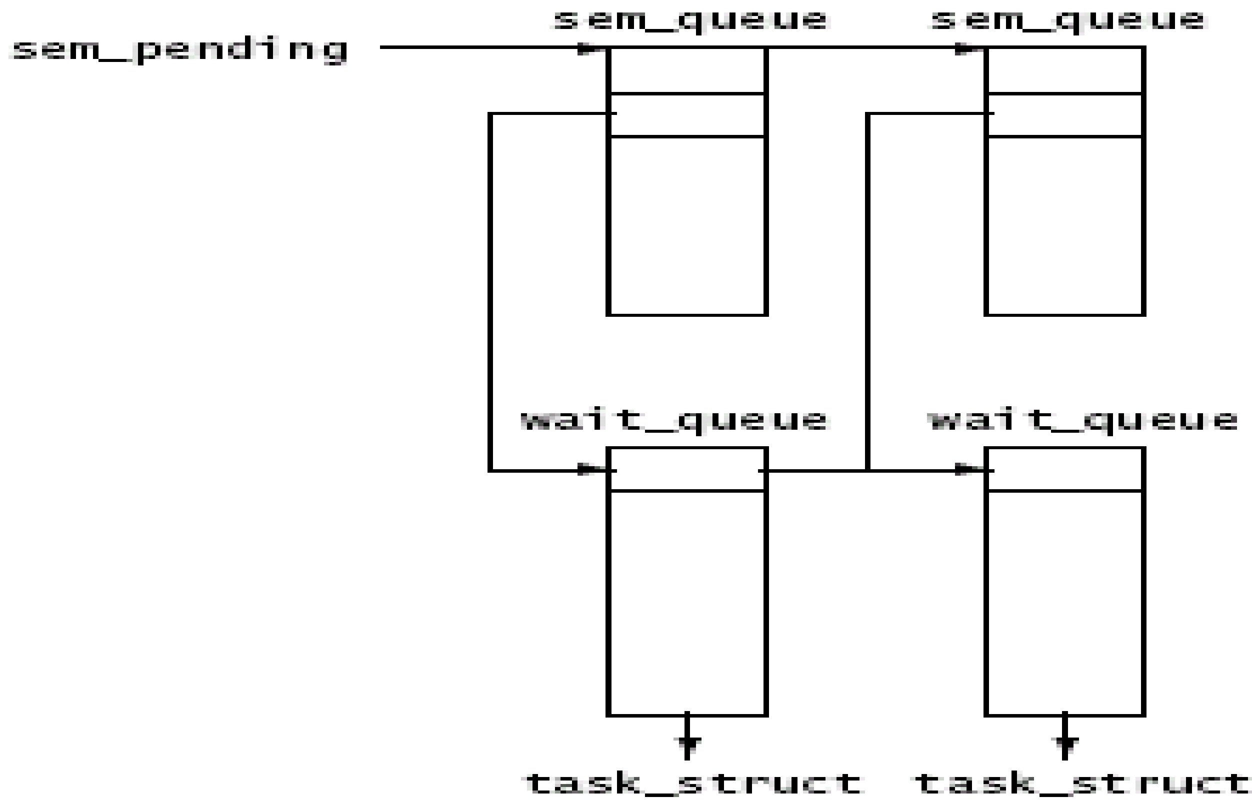
System V semaphore ….

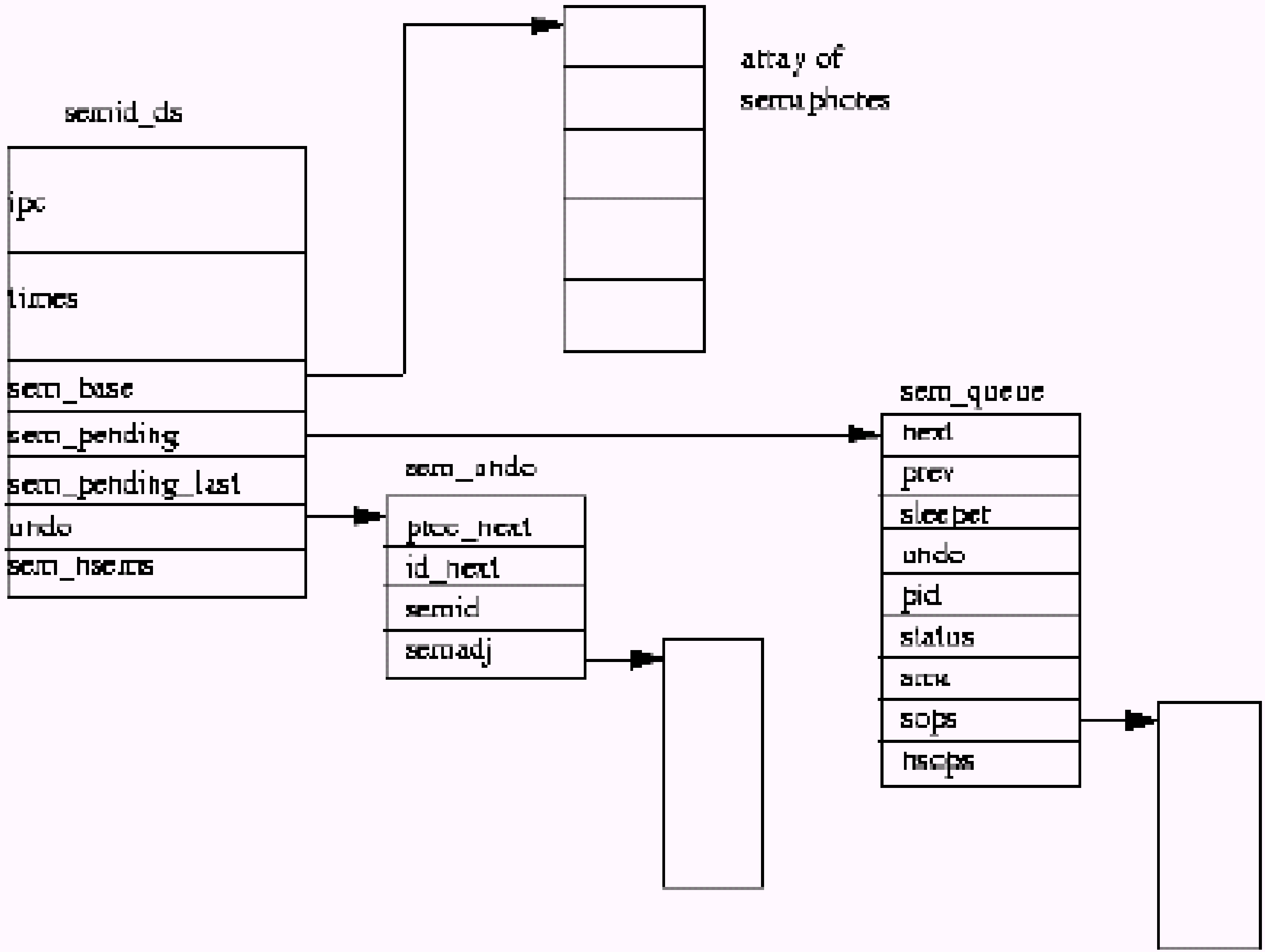


System V semaphore …..

```
struct sem_queue{
        struct sem_queue  *next;
        struct wait_queue *sleeper;
        struct semid_ds   *sma;
        struct sembuf     *sops;
        int               nsops;
};
```

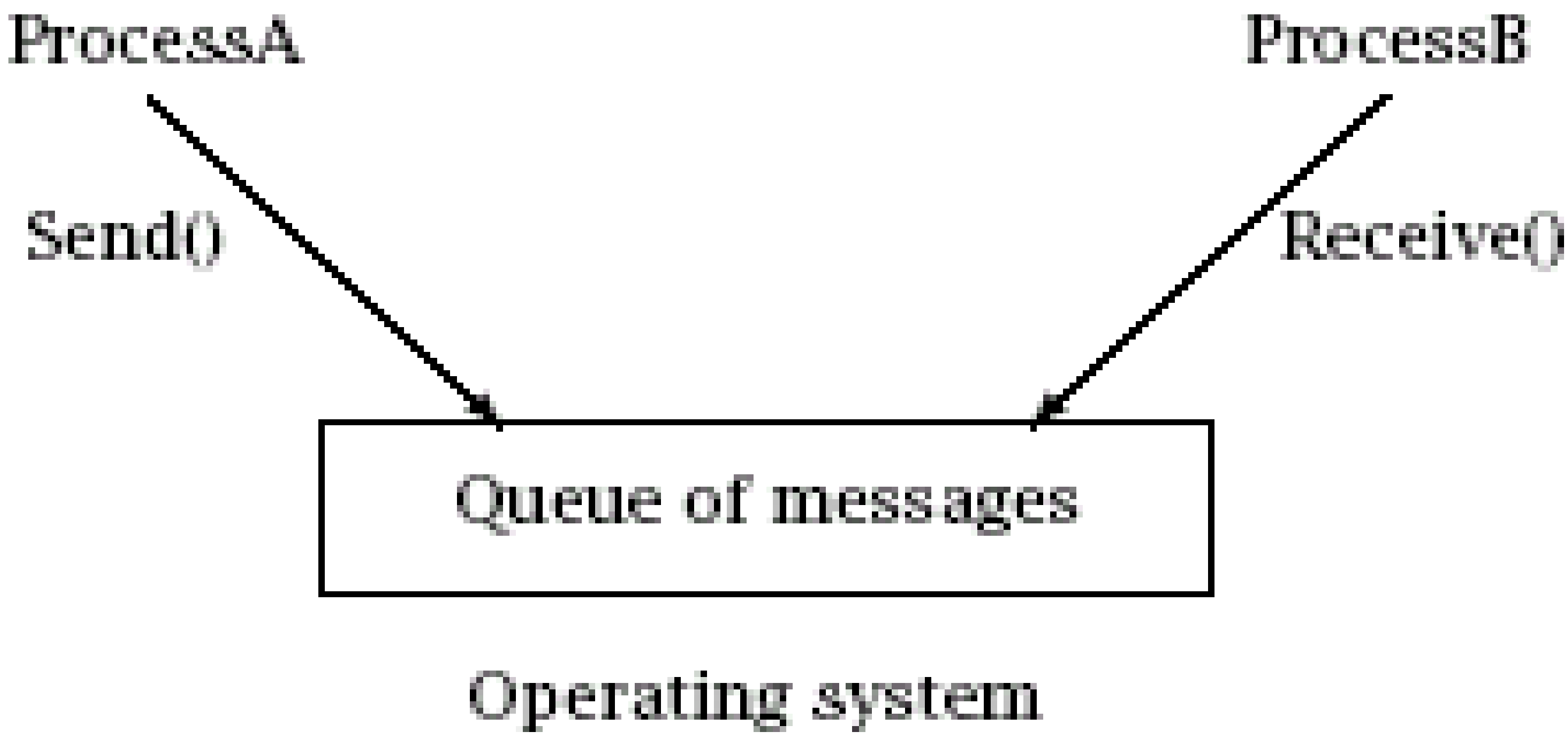
System V semaphore ……





array of
semaphores
semid_ds
ipc
times
sem_base
sem_pending
sem_pending_last
undo
sem_nsems
sem_undo
proc_next
id_next
semid
semadj
sem_queue
next
prev
sleeper
undo
pid
status
sma
sops
nsops

System V message queues



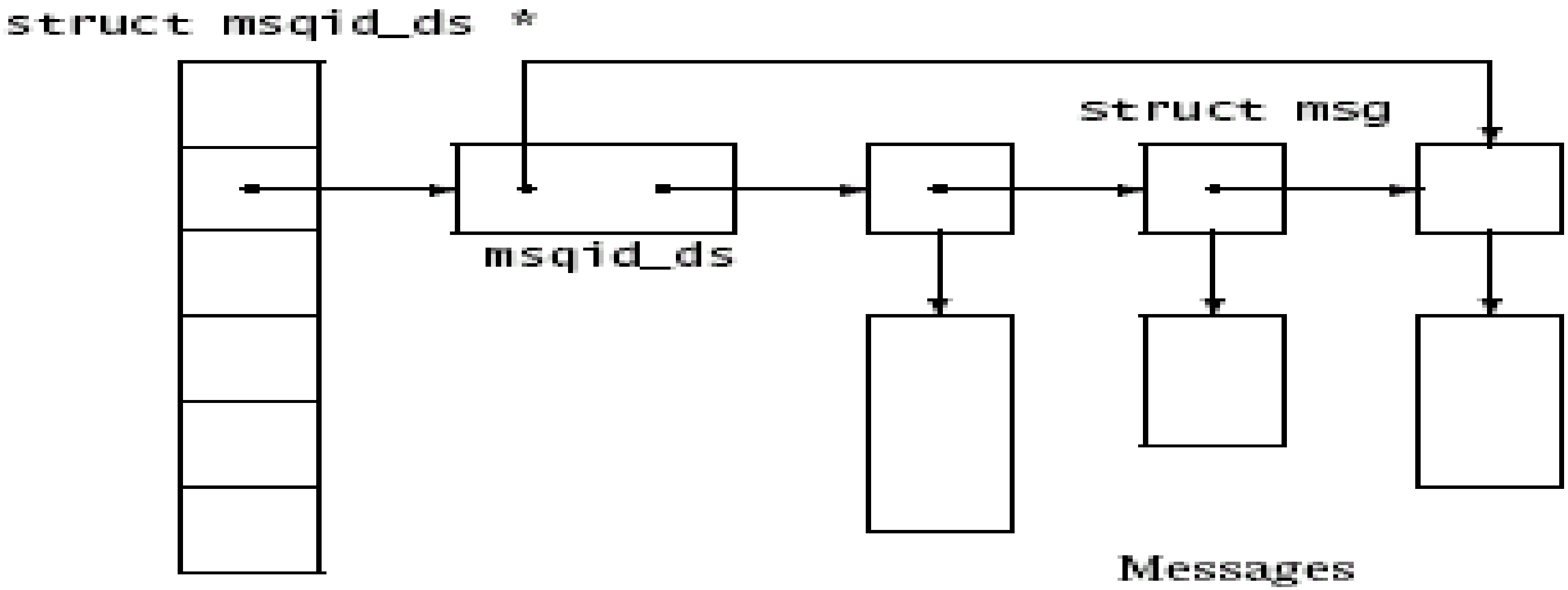
System V message queues..

```
msqid_ds{
    struct ipc_perm   msg_perm;   /* access permissions        */
    struct msg        *msg_first; /* first message on queue    */
    struct msg        *msg_last;  /* last message on queue     */
    struct wait_queue *wwait;     /* blocked writing threads   */
    struct wait_queue *rwait;     /* blocked reading threads   */
    unsigned short    msg_qnum;   /* number of messages on queue */
};
```

System v message queues …

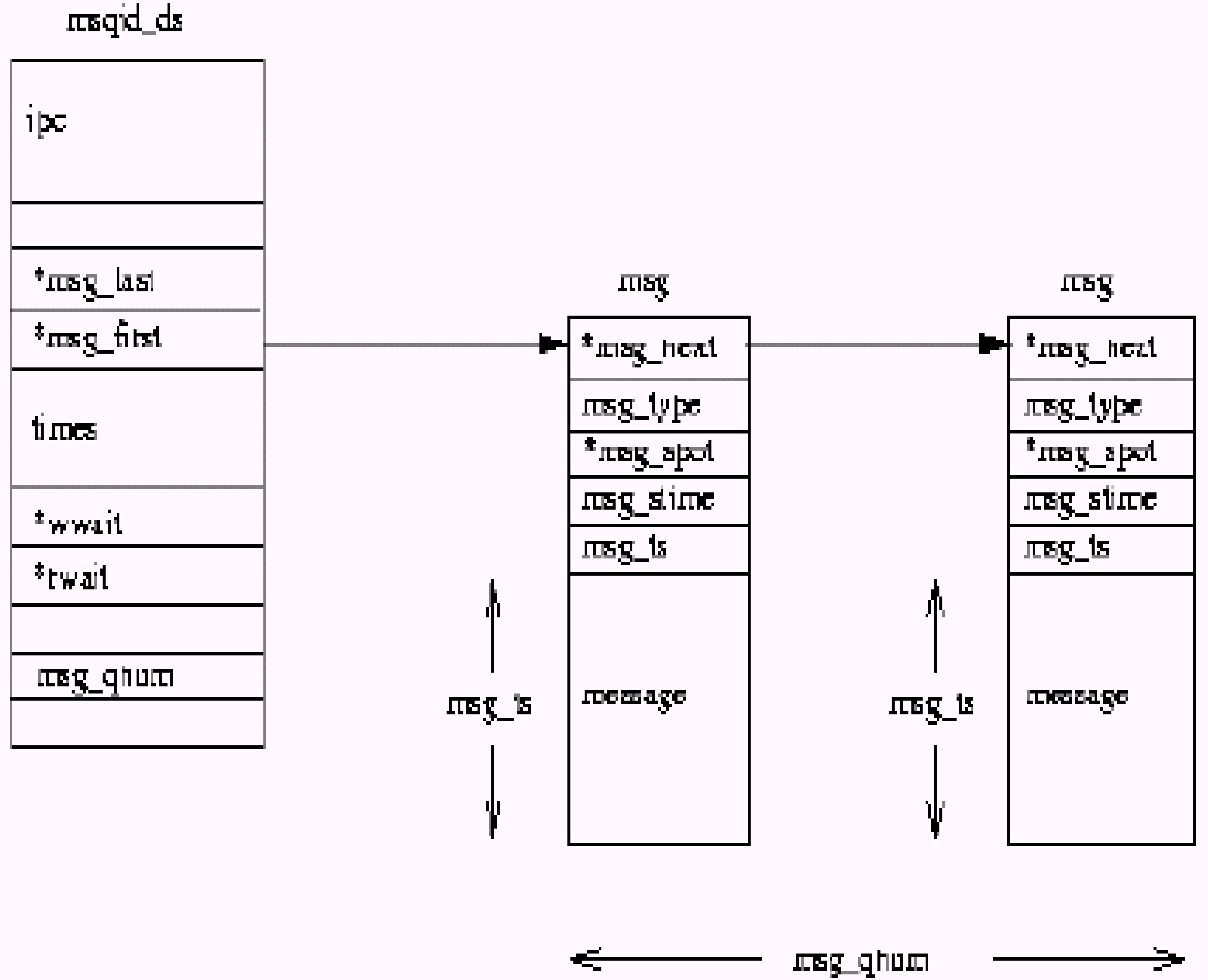
```
struct msg{
      struct msg *msg_next; /* next message on queue  */
      long       msg_type;  /* as specified by sender */
      char       *msg_spot; /* message text address   */
      time_t     msg_stime; /* msgsnd time            */
      short      msg_ts;    /* message text size      */
};
```

System V message queues ….



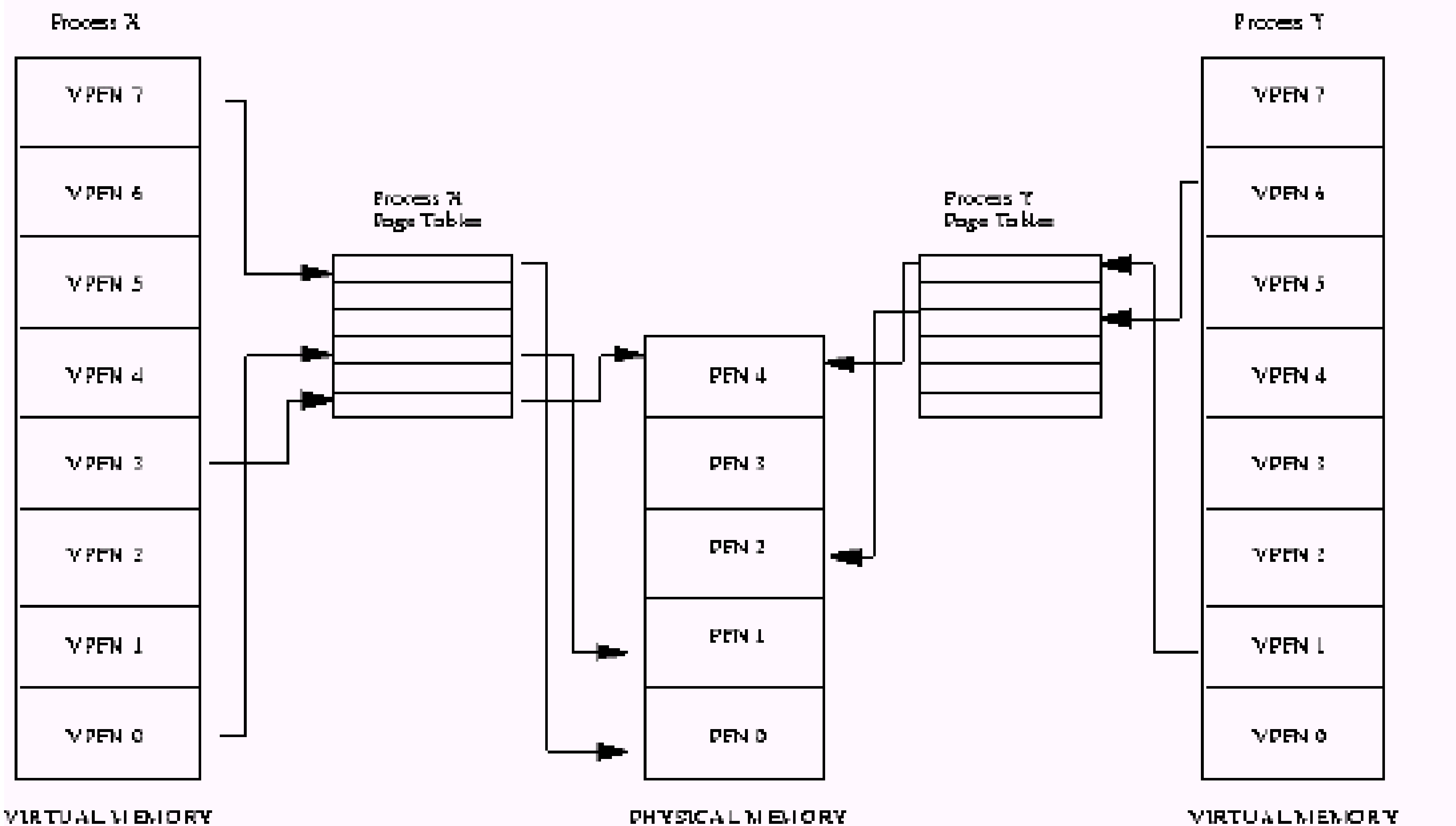
System V message queues …..

```
struct msgbuf{
      long mtype;    /* message type      */
      char mtext[]; /* message contents */
};
```

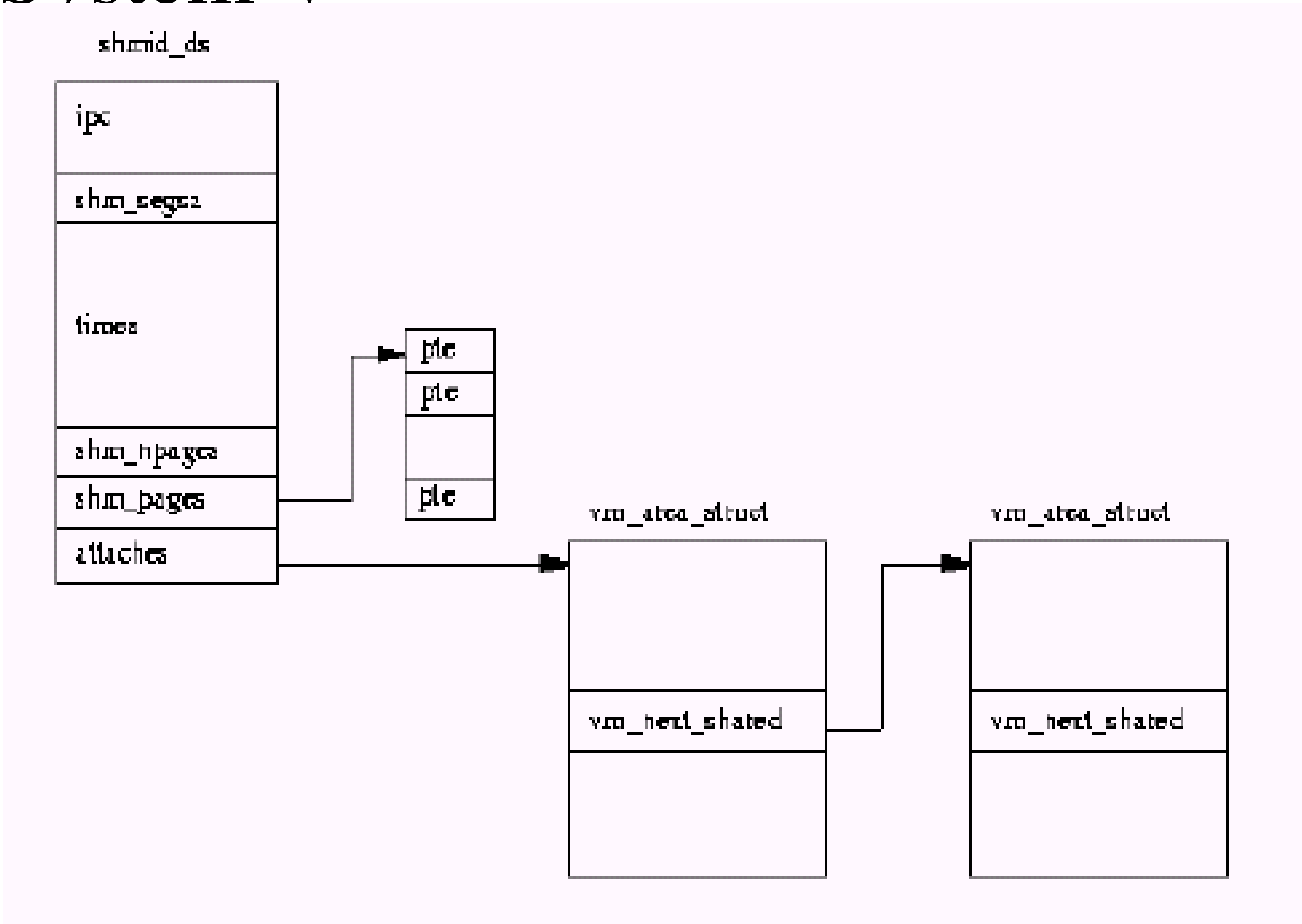


msqid_ds
ipc
*msg_last
*msg_first
times
*wwait
*rwait
msg_qnum
msg
*msg_next
msg_type
*msg_spot
msg_stime
msg_ts
message
msg_ts
msg
*msg_next
msg_type
*msg_spot
msg_stime
msg_ts
message
msg_ts
msg_qnum

An Abstract Model of Virtual Memory



System V


shmid_ds
ipc
shm_segsz
times
shm_npages
shm_pages
attaches
pte
pte
pte
vm_area_struct
vm_next_shared
vm_area_struct
vm_next_shared

Threading Model

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Threading Model

- Kernel-level threading with 1:1 model
- PD also acts as TD/LWP
- NPTL library and kernel support for POSIX Threading Standard
- CPU scheduling discussed earlier applies to multi-threading model