Introduction to Advanced Linux Programming

VIVEN

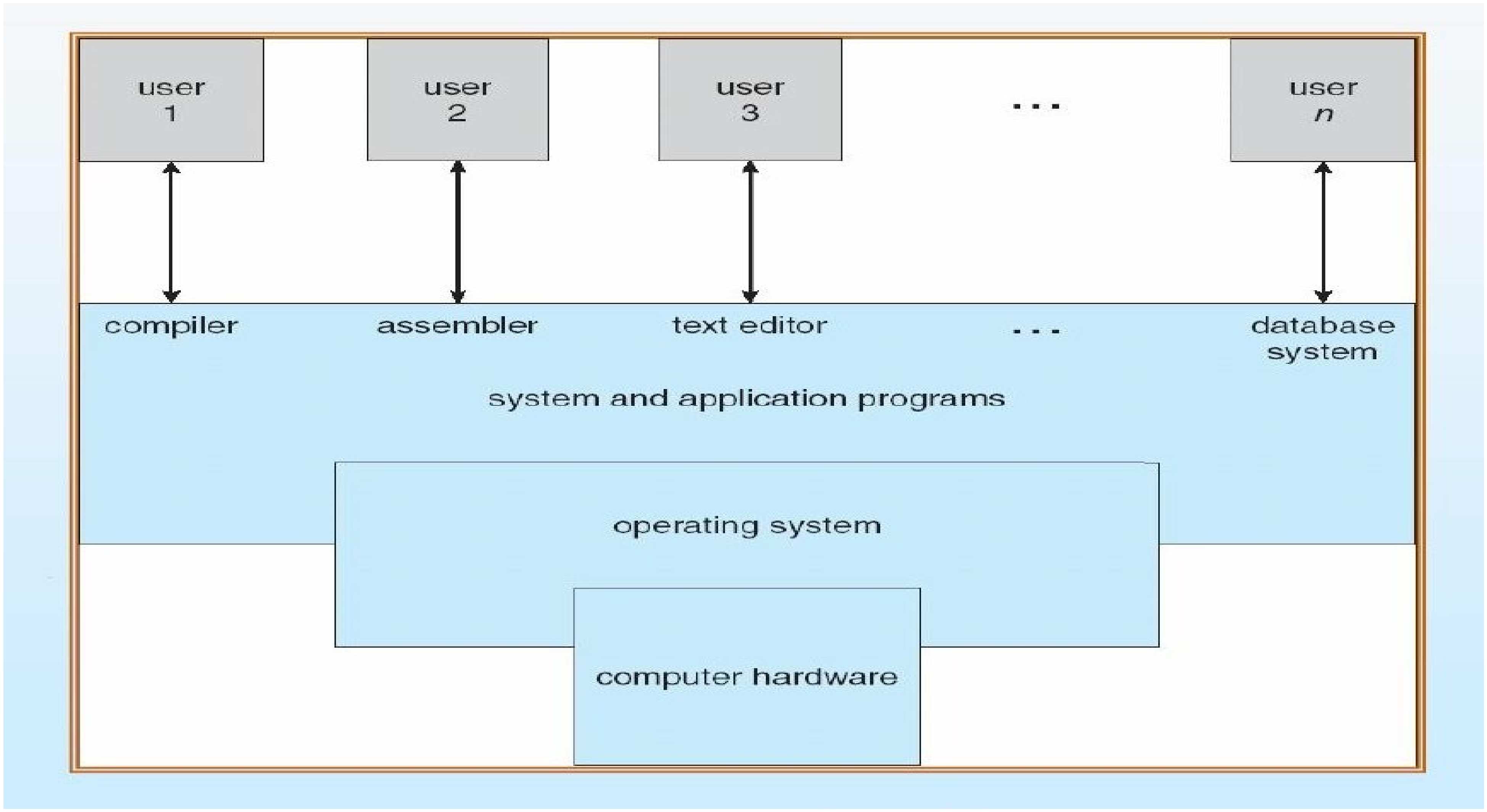
Embedded Academy

Computer System

Computer system can be divided into four components

- Hardware –provides basic computing resources
  - CPU, memory, I/O devices
- Operating system
  - Controls and coordinates use of hardware among various applications and users
- Application programs –define the ways in which the system resources are used to solve the computing problems of the users
  - Word processors, compilers, web browsers, database systems, video games
- Users
  - People, machines, other computers

Computer System



Operating System

Primary Goal:

- provides an environment to execute user programs (appln)

Secondary Goal:

- Use Hardware resouces in “effecient” manner.

Resource Allocator - manages resources and resolve conflicting requests with efficiency and fair resource use.

Resource Controller – controls execution of programs to prevent errors and improper use of computer.

```
User1  User2  User3 ............ User-N
                    |
        Application Programs
                         |  ->system call api
                    O/S
                    | -> device drivers
              Hardware
```

Operating System Architecture

OS Funcationality:

1. Process Management
2. Memory Management
3. File Management
4. I/O Management

Architecture ......

<u>Process Management</u>

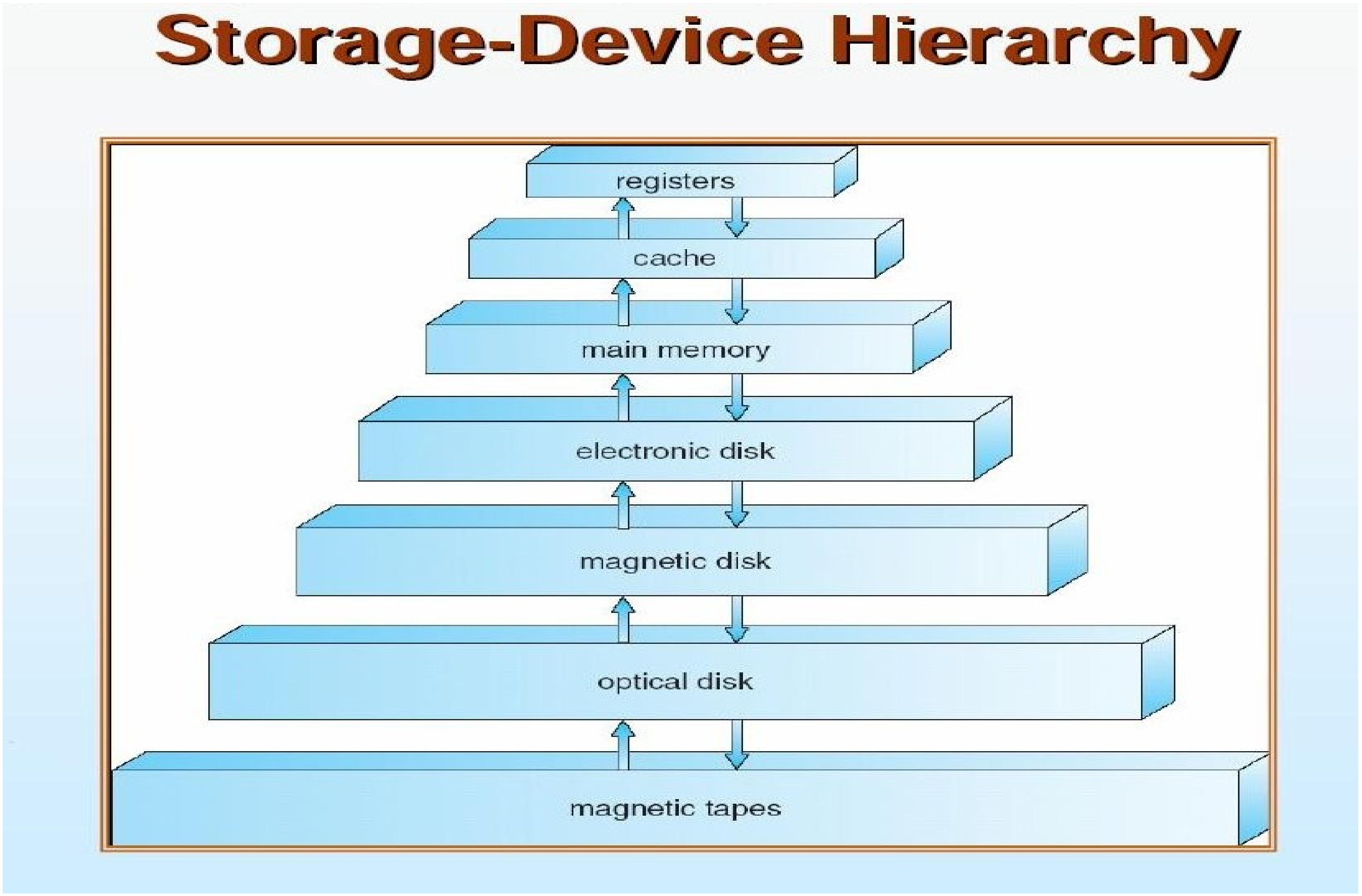
What is Process?

- program Vs process
- A process is protected from other processes but can communicate with them.

- Manages Process creation, deletion of both user and system processes.
- Suspension and resumption of processes.
- provides mechanisms for synchronization b/w processes.
- provides mechanisms for communication b/w processes.
- provides mechanisms for deadlock handling.

Architecture ......

Storage Management



Architecture ......

Storage Management

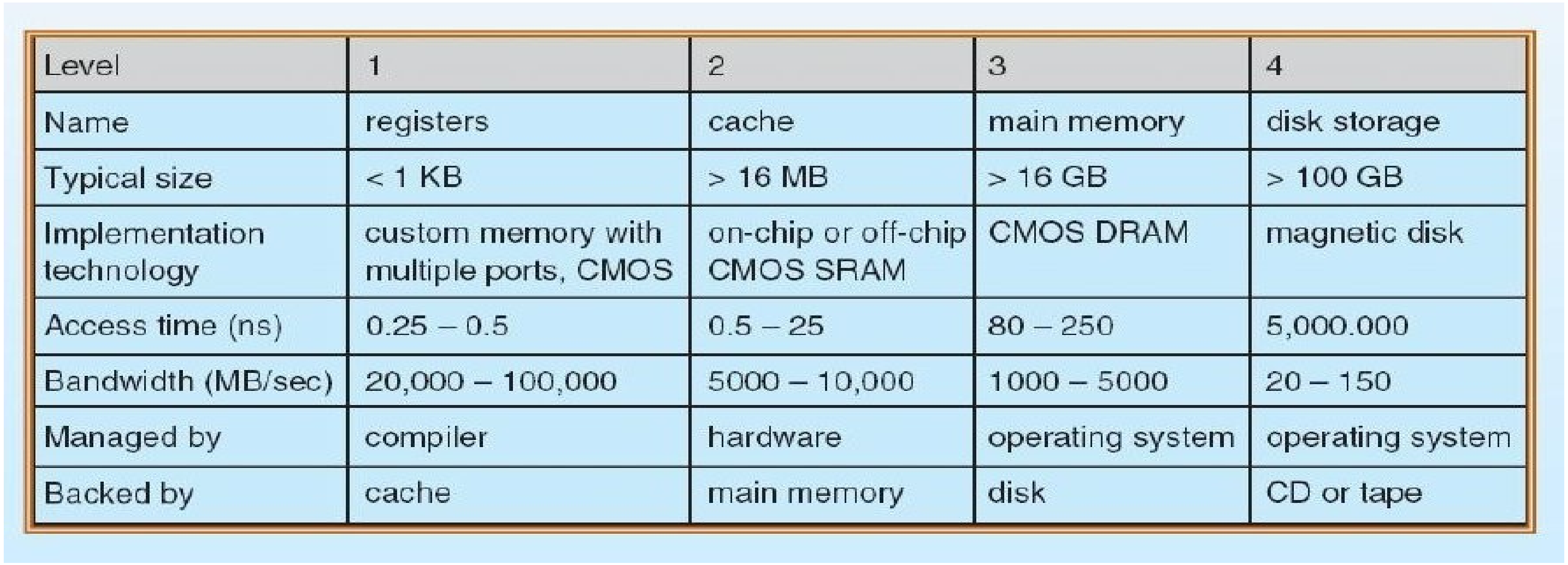
OS provides uniform, logical view of information storage

- Abstract physical properties to logical storage unit-file
- Each medium controlled by device (ie, disk, tape)
  - varying properties including speed, capacity, data transfer rate, access method (sequential or random).
- Entire speed of computer operation hinges on disk subsystem and its algorithms.
- OS activities:
  - Free-Space management
  - Storage Allocation
  - Disk Scheduling

Architecture ......

File-System Management

- Files usally organized into directories.
- Control the creation and removal of files and provide directory maintenance. In UNIX and Linux: everything is a file
- For a multiuser system, every user should have its own right to access files and directories. - access control
- Mapping files to storage.
- back-up files into stable storage media.



Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

Architecture ......

Memory management

- Memory in a computer is divided into main memory (RAM) and secondary storage (usually refer to hard disk).
- Memory is small in capacity but fast in speed, and hard disk is vice versa.
- Data that are not currently used should be saved to hard disk first, while data that are urgently needed should be retrieved and stored in RAM.
- Allocates/ frees the memory space as needed.
- keeps track of which parts of memory are currently being used and by whom
- Decide which processes are to be loaded into memory when memory space becomes available

Architecture ......

I/O Management

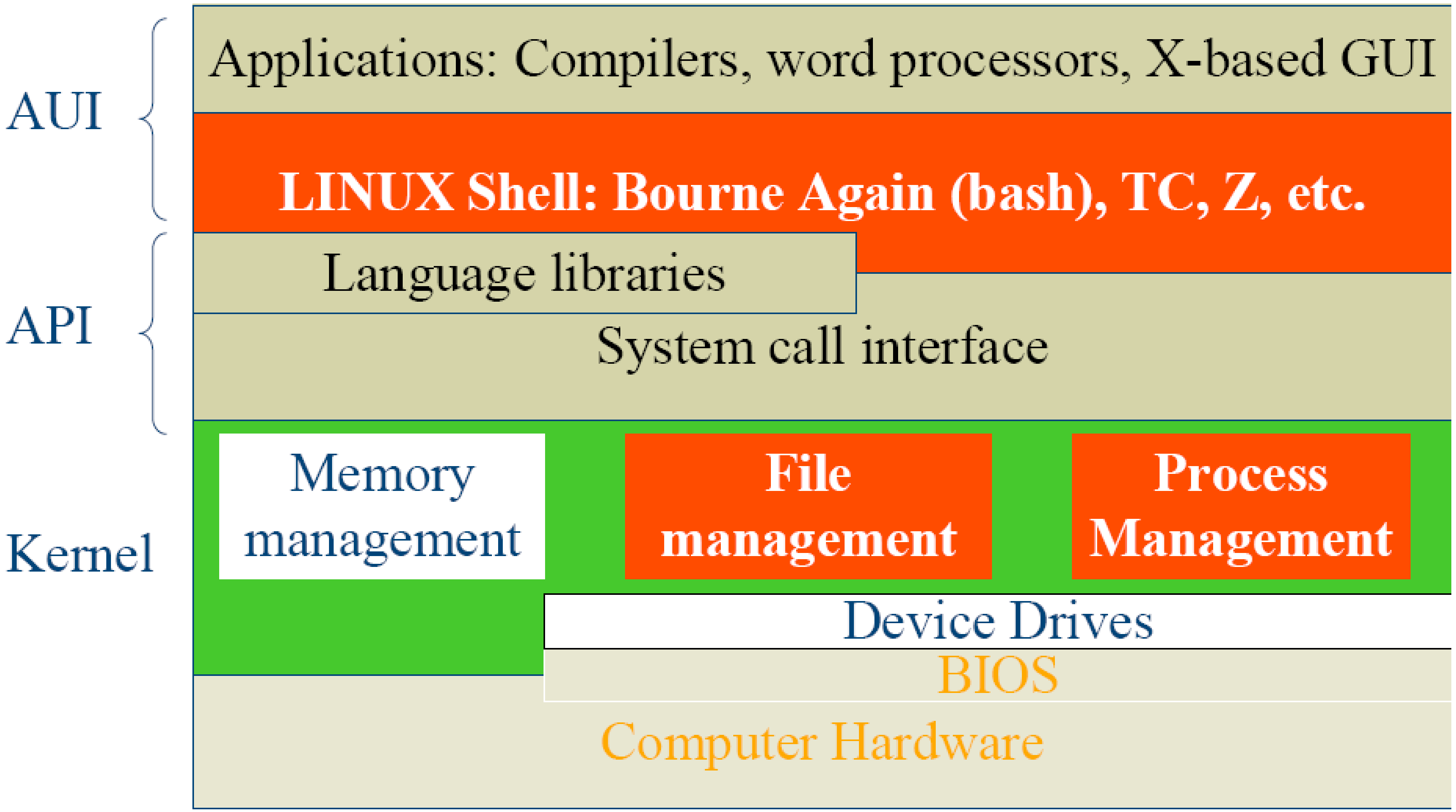
One purpose of OS is to hide peculiarities of h/w devices from user

- provides interface with I/O devices using device drivers.
- provides general device-driver interface.
- provides drivers for specific h/w devices.
- Memory mgmt of I/O includes
  - •buffering – storing data temp. while it is being txed.
  - •caching – storing parts of data in faster storage for performance.
  - •spooling - overlapping of output of one job with input of other jobs

Device drivers

- Interfaces between the kernel and the BIOS.
- Different device has different driver.

Architecture



Architecture ......

Kernel

- The part of an OS where the real work is done

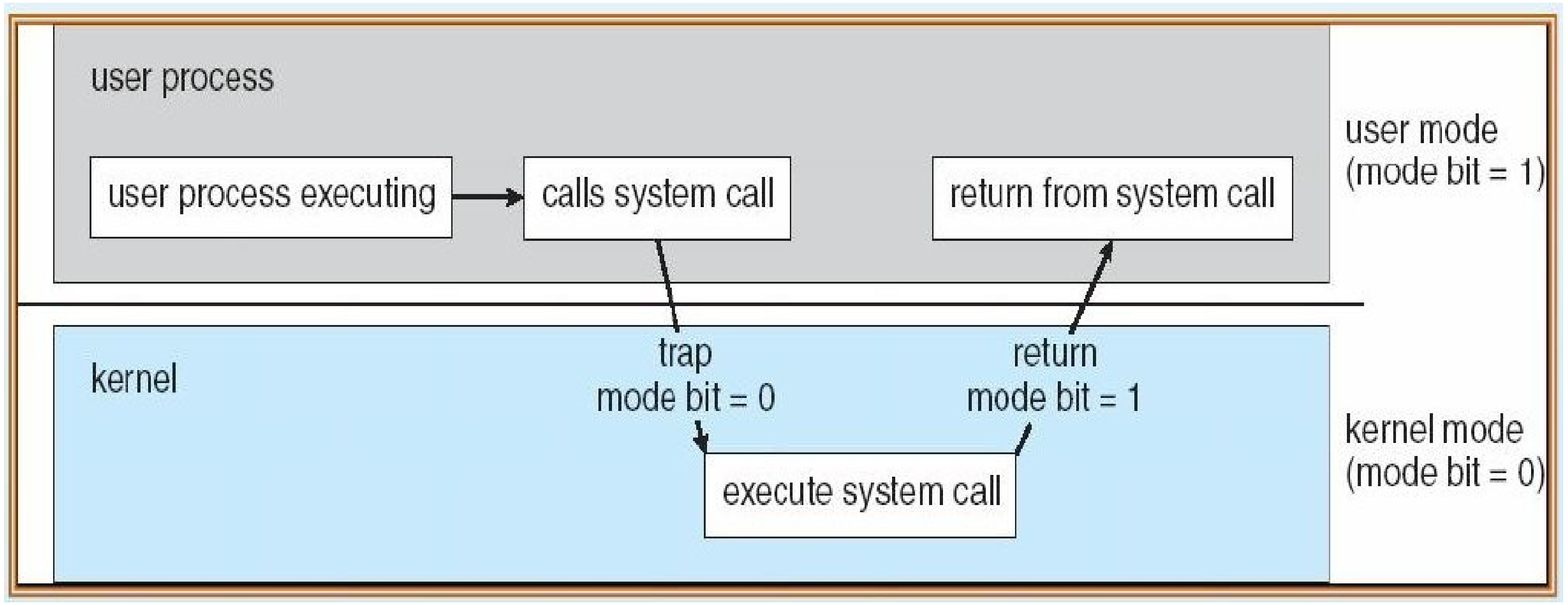
System call interface

- Comprise a set of functions (often known as API) that can be used by the applications and library routines to use the services provided by the kernel

Application User's Interface

- Interface between the kernel and user
- Allow user to make commands to the system
- Divided into text based and graphical based

System Call


user process
user process executing
calls system call
return from system call
user mode
(mode bit = 1)
trap
mode bit = 0
return
mode bit = 1
kernel
execute system call
kernel mode
(mode bit = 0)

System Call

Typically, a number associated with each system call

- System-call interface maintains a table indexed according to these numbers

Three general methods used to pass parameters to the OS

- Simplest: pass the parameters in registers
  - In some cases, may be more parameters than registers
- Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
  - This approach taken by Linux and Solaris
- Parameters placed or pushed, onto the stack by the program and popped off the stack by the operating system

Block and stack methods do not limit the number or length of parameters being passed

System Boot

Operating system must be made available to hardware so hardware can start it

- Small piece of code –bootstrap loader, locates the kernel, loads it into memory, and starts it
- Sometimes two-step process where boot block at fixed location loads bootstrap loader
- When power initialized on system, execution starts at a fixed memory location
    - Firmware used to hold initial boot code

OS Evalution

1. Batch Processing Systems - is execution of a series of programs ("jobs") on a computer without human interaction.

2. Multi Programming Systems – multiple programs residing in memory.

3. Time Sharing (Multi Tasking) Systems – human interaction during execution. time sharing, virtual memory.

4. Multi Processor Systems - more then one control of execution(> 1cpu)

5. Distributed Systems - more then one computer

6. Real Time Systems – time critical applns

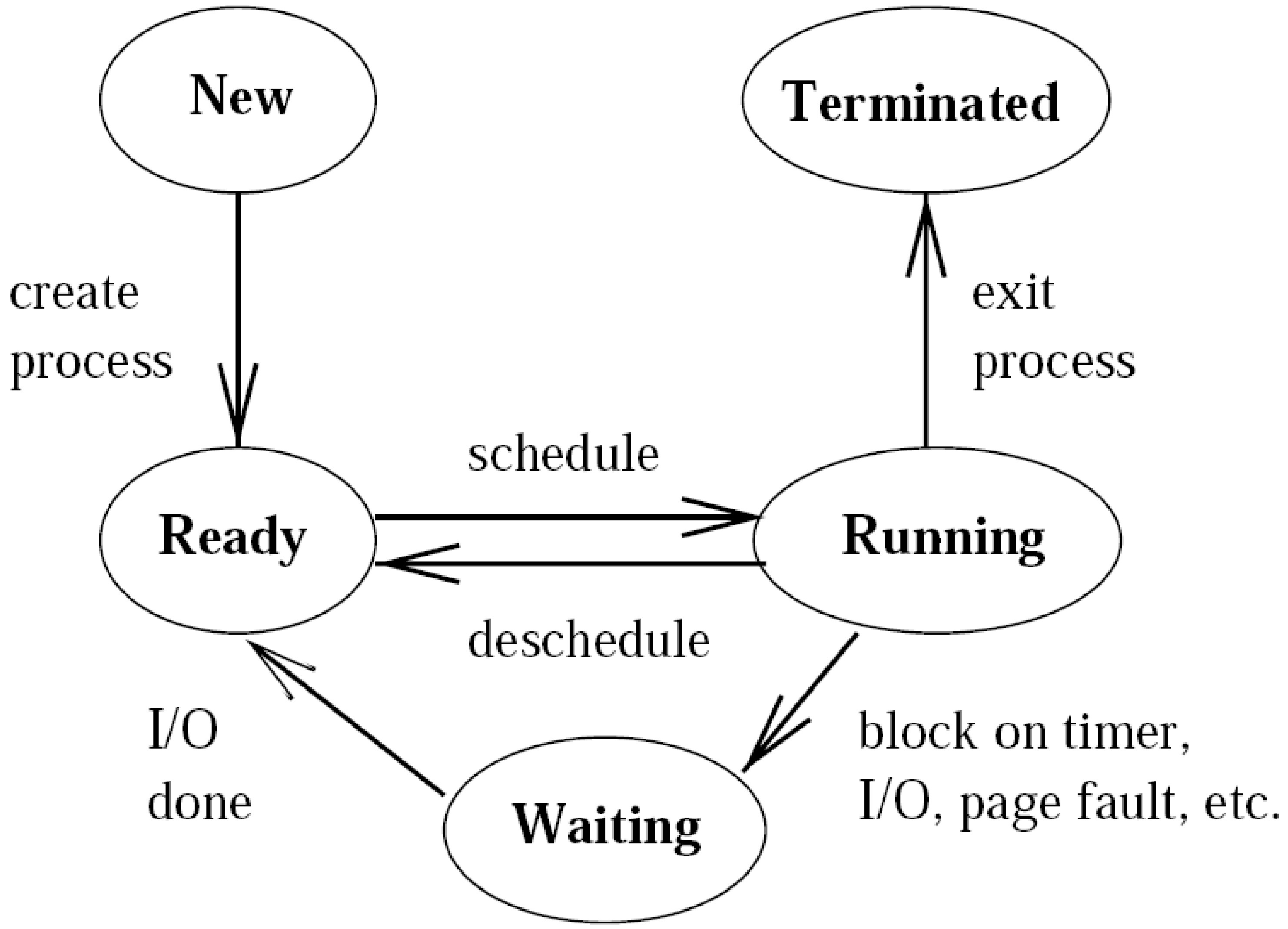
OS Evalution

- **Multiprogramming** needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇨ **process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling**
  - If processes don't fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory

Process Management.....

- A process is the dynamic execution context of an executing program.
- Several processes may run the same program, but each is a distinct process with its own state (e.g., vim).

Process Execution State.......



Process Execution State

- Each process has an execution state which indicates what it is currently doing.
  * new: the OS is setting up the process state
  * ready: ready to run, but waiting for the CPU
  * running: executing instructions on the CPU
  * waiting: waiting for an event to complete (e.g., I/O)
  * terminated: the OS is destroying this process
- As the program executes, it moves from state to state, as a result of the program actions (e.g., system calls), OS actions (scheduling), and external actions (interrupts).

Process Execution State.......

Example:

state sequence

```
void main() {
        printf("Hello World");
}
```

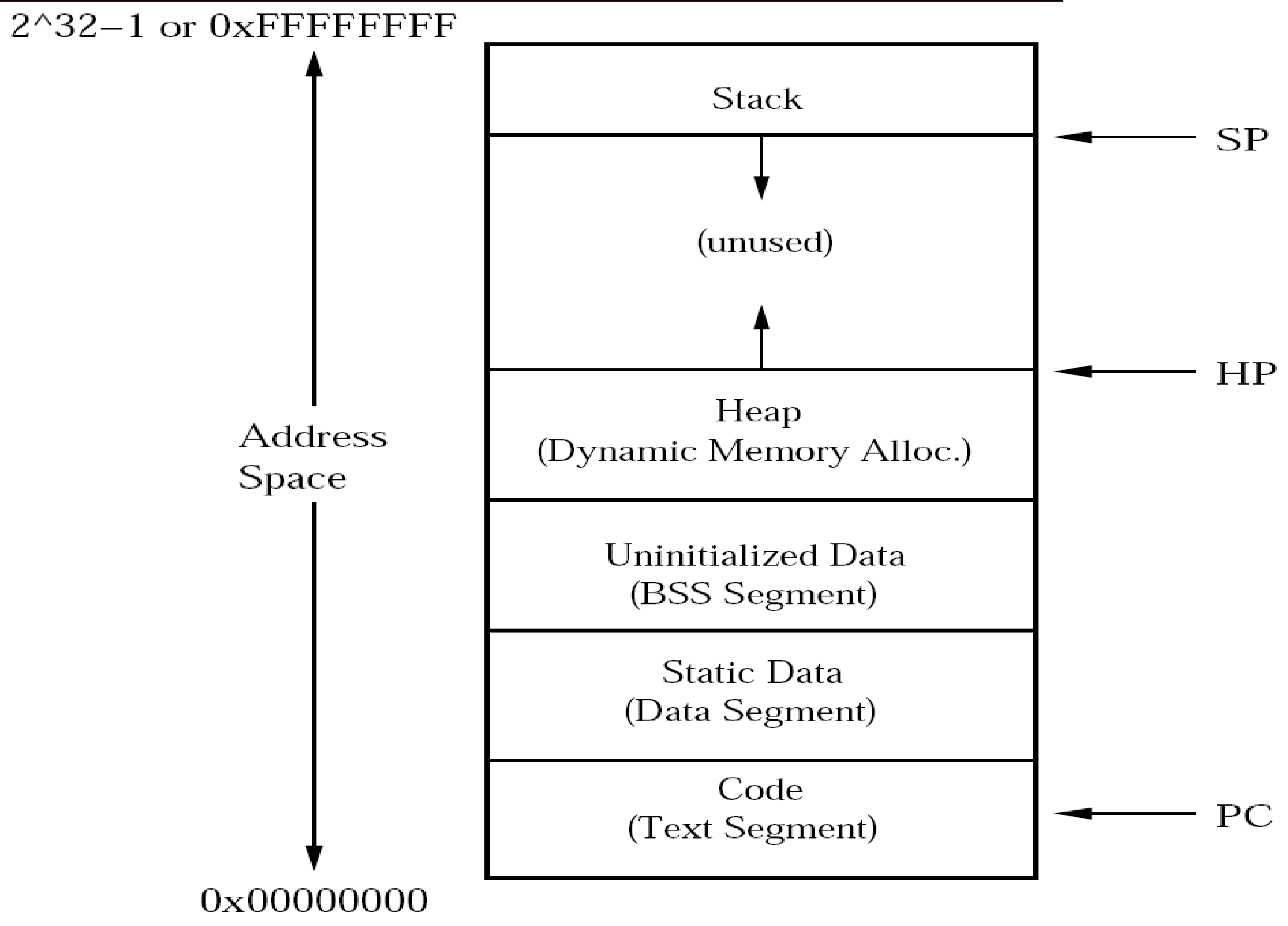
- Since the OS manages many processes at once, it keeps track of processes using queues. For example, ready processes are stored in the ready queue.

Process Management.....

Process state consists of:

- memory state
  * code
  * static data
  * dynamic data: stack and heap
- kernel state
  * process state: ready, running, etc.
  * resources: open files, open sockets, etc.
  * scheduling info: priority, total CPU time, etc.
- processor state
  * program counter (PC)
  * registers

Process Memory State (Virtual Address Space)



2^32–1 or 0xFFFFFFFF
Address Space
0x00000000
Stack
(unused)
Heap (Dynamic Memory Alloc.)
Uninitialized Data (BSS Segment)
Static Data (Data Segment)
Code (Text Segment)
SP
HP
PC

Process Memory Space.....

To learn about segments in a UNIX program, run a command like *size /bin/sh*

On Red Hat Enterprise Linux 4 this prints:

```
  text     data     bss     dec     hex    filename
516891   22836   16784  556511   87ddf    /bin/sh
```

Kernel Data Structures for Processes

- The OS uses a Process Control Block (PCB) | a dynamic data structure | to keep track of each processes (this data structure is in addition to the process state in memory).
- The PCB represents the execution state and location of each process when it is not executing (e.g., when it is waiting).

The PCB contains:

- Process state (running, waiting, etc.)
- PID(Process ID)
- Program Counter
- Stack Pointer
- General Purpose Registers
- Memory Management Information
- Username of owner

Kernel Data Structures for Processes.....

- List of open files
- Queue pointers for state queues (e.g., the waiting queue)
- Scheduling information (e.g., priority)
- I/O status

- When a process is created, the OS allocates and initializes a new PCB, and then places the PCB on the ready queue.
- When a process terminates, the OS deallocates the PCB and cleans up process state (closes open files, recovers memory, etc.).

Kernel Data Structures for Processes.....

- Process State Queues:
  - The OS keeps track of PCBs using queues.
    - All ready processes stay on the ready queue.
    - Each I/O device has its own wait queue.
    - All processes are always on an "all process" queue.
  - Separate queues are used for performance reasons.

- PCBs and Hardware State:
  - The OS uses a context switch to change a process from Ready to Running and vice versa. Content switches are a relatively expensive operation.

Context Switch

- When the OS is ready to start executing a Ready process, it loads registers (PC, SP, PSW, and other registers) from the values stored in that process' PCB.
- While a process is Running, the CPU modifies the Program Counter (PC), Stack Pointer (SP), registers, etc.
- When the OS blocks or pre-empts a process, it saves the current values of the registers, (PC, SP, etc.) into the PCB for that process.
- This process of switching the CPU from one process to another is the context switch.
- Correctness criterion: Context switches must be invisible to the process (unless it looks at a clock).
- Time sharing systems may do 100 to 1000 context switches a sec.

Context Switch

- When the OS is ready to start executing a Ready process, it loads registers (PC, SP, PSW, and other registers) from the values stored in that process' PCB.
- While a process is Running, the CPU modifies the Program Counter (PC), Stack Pointer (SP), registers, etc.
- When the OS blocks or preempts a process, it saves the current values of the registers, (PC, SP, etc.) into the PCB for that process.
- This process of switching the CPU from one process to another is the context switch.
- Correctness criterion: Context switches must be invisible to the process (unless it looks at a clock).
- Time sharing systems may do 100 to 1000 context switches a sec.

Mode, Context and Space

Whole address space is divided into two regions

- Kernel Space
- User Space

- Fixed part of virtual address space is dedicated to kernel
- kernel space is accessible only in kernel mode.
- only one instance of kernel running in the system, hence all processes map a single kernel address space.
- unix kernel is re-entrant. so, unix keeps separate kernel stack for each process. this stack will be in process addr space but can be accessible in kernel mode only.

Unix Process Context

Each Process has well-defined context, comprising all information needed to describe the process.

- User Address Space
- Control Information
    * u-area (User Area)
    * proc structure
- Credentials
- Environment Variables
- H/W context
    * Program Counter (PC), Stack Pointer (SP)
    * Process Control Word (PSW)
    * Memory mgmt resources

Unix Process Control Information

Unix kernel maintains 2 imp per-process objects
- u-area – contains data that need only when proc is running
- proc structure – info that is needed even if proc is not running

U-Area:
- it is part of process space which is mapped and visible only when process is running.
- on many implimentations, u-area is always mapped at same fixed location. context switch will reset this mapping
- kernel may need to access u-area of some other process in c

U-Area

- PCB – stores h/w context when process is not running.
- ptr to proc structure.
- UID & GID
- arguments, return values, error status from current syscall.
- singnal handler info
- info about segments and its sizes and other memory mgmt info.
- open file descriptor
- cpu usage and profiling info, disk quotas & resource limits
- per-process kernel stack.

Proc Structure

- Process ID (PID)
- Location of kernel address map for u-area of this process.
- current process state.
- fwd and bkwd ptrs to process in the queue(might be any Q)
- scheduling priority and related info.
- singnal handling info: masks of signals that are ignored, blocked, posted and handled.
- memory mgmt info
- ptrs to link this structure to list of active, free or zombie processes.
- process relationships.

Executing in Kernel mode

events which causes systems to enter kernel mode

- device interrupts
- Exceptions
- software interrupts (traps)

- kernel saves state(PC and PSW) of interrupted process onto kernel stack.

<u>Interrupts Vs Exceptions:</u>

- Interrupts are asynchronous, they must be serviced in system context. must not block as it is blocking other process.
- Exceptions are synchronous to process. runs in process context. can block.
- traps also synchronous, occur when process executes special instructions.

Executing in Kernel

How system call works

wrapper routine pushes trap number to user stack and calls special instruction called trap.

1. syscall copy args from user stack to kernel stack.
2. saves state or process and store it in kernel stack by accessing u-area.
3. execute in kernel stack
4. copy return values and errors.
5. restore process state and set the mode accordingly

How interrupt handler works

Unix File I/O System Calls

VIVEN

Embedded Academy

Agenda

- File Descriptors
- File Types
- Unix File I/O calls(unbuffered I/O)

    -open, create, close, lseek, read, write, dup, dup2

File Descriptors

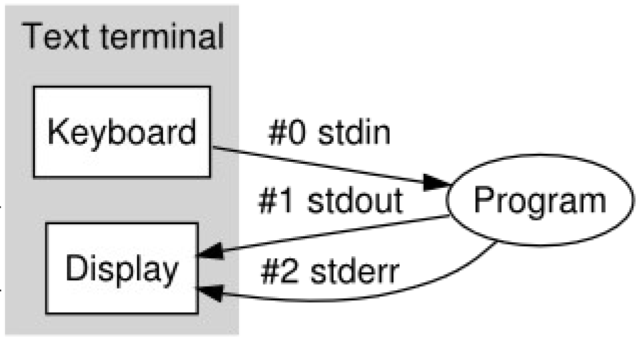
- Every UNIX process has a set of file descriptors
- File descriptors are small integers that map name, opened file objects in the kernel.
- Multiple file descriptors (belonging to the same process or to different processes) can point to the same open file in the kernel.
- Some file descriptors have a special meaning:

  0 is STDIN.

  1 is STDOUT.

  2 is STDERR.

- By default STDIN reads from the keyboard
- STDOUT and STDERR write to the consol

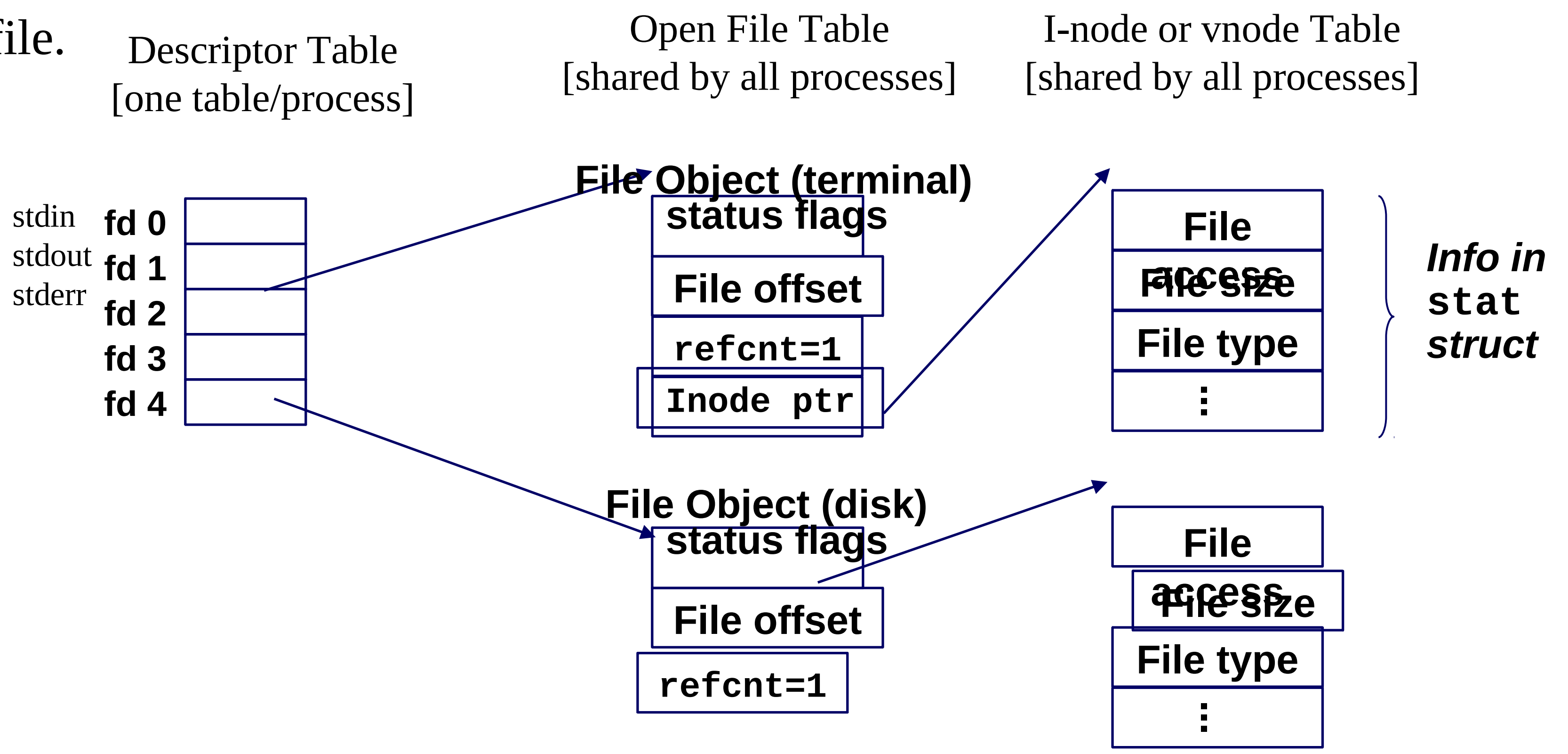


File Descriptors

How Unix kernel represents open files:

Two descriptors referencing two distinct files

Descriptor 1(stdout) points to terminal, and descriptor 4 points to open disk file.



Kernel Data Structure for open files

- When we call an open() system call to open a file, the control will switch to the kernel. The kernel searches the block device(hard disk) for the file's 'i-node'. Every file is associated with a inode.
- Open() call creates i-node structure in memory and copies the i-node contents from block device to i-node structure in memory.
- Next it creates a new structure called 'File Object'. The file object structure contains 'file status flags', 'file offset' and pointer to inode structure created in the above step.
- Opening a file with multiple processes:
    - Multiple processes can open a single file. Then each process will have separate file table entry(file object). But all these file table entries point to the same v-node entry.

File Descriptors …….

- Most UNIX processes have these file descriptors opened when they are created.
- Generally, a file descriptor is an index for an entry in a kernel-resident data structure containing the details of all open files.
- In POSIX this data structure is called a file descriptor table, and each process has its own file descriptor table
- In Unix-like systems, file descriptors can refer to files, directories, block or character devices (also called "special files"), sockets, FIFOs (also called named pipes), or unnamed pipes.

File Types

"On a UNIX system, everything is a file; if something is not a file, it is a process".

- There are four types of files in the Unix file system.
  - **Regular files**: An ordinary file holds data - either ASCII text or binary.
  - **Directory files**: A directory is a unique file that again holds data. The data is restricted to being a list of files.
  - **Character device files**: Character devices such as serial ports and parallel are represented with these files.
  - **Block Device File**: Block devices such as floppies, hard disks and CD-ROMs are represented.
  - **Symbolic Links**: This is a file which contains pointers to another file. The allows single file to have multiple names.

File Types .....

- **FIFOs**: A FIFO is a special file type that permits independent processes to communicate. An IPC mechanism that allows applications to exchange data
- **Sockets**: This file is used for network communication between processses running on different machines or on the same machine.

Unix File I/O calls

- The elegant mapping of files to devices allows kernel to export simple interface called Unix I/O.
- All input and output is handled in a consistent and uniform way.

Basic Unix I/O operations (system calls):

- Opening and closing files: *open()*and *close()*.
- Changing the current file position: *lseek()*
- Reading and writing a file: *read()* and *write()*

Opening files

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int oflag, ... );
```

- Opening a file informs the kernel that you are getting ready to access the file.
- Returns a small identifying integer file descriptor. -1 indicates an error.
- Each process created by Unix shell begins life with three open files.
  - 0: Standard input
  - 1: Standard output
  - 2: Standard error

Closing files

- #include <unistd.h>
  int close(int fildes);
- Closing a file informs kernel that you are finished accessing that file.
- Closing an already closed file is a recipe for disaster in threaded programs (more on this later).
- Some error reports are delayed until close
- Moral: Always check return codes, even for seemingly benign functions such as *close()*.

Creating files

-
  ```
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <fcntl.h>
  int creat(const char *path, mode_t mode);
  ```
- create a new file or rewrite an existing one.
- The function call *creat(path, mode)* is equivalent to:

  open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);

Reading files

- #include <unistd.h>

  *ssize_t read(int fd, void *buf, size_t nbyte);*

- Reading a file copies bytes from the current file position to memory, and then updates file position.
- Returns number of bytes read from *fd* into *buf*. *If* -1 indicates as error has occured, 0 indicates EOF(end of file).
- short counts (nbytes < sizeof(buf)) are possible and are not errors!

Writing files

- #include <unistd.h>

  *ssize_t write(int fd, const void *buf, size_t nbyte);*
- Writing a file copies bytes from memory to the current file position and then updates the cuuernt file position.
- Returns number of bytes written from buf into fd, -1 indicates an error has occured.
- As with *read()*, short count are possible.

Moving files

```
#include <sys/types.h>
#include <unistd.h>
```

off_t lseek(int fd, off_t offset, int whence);

- The lseek() function will set the file offset for the open file description associated with the file descriptor fd.
- The lseek() function will allow the file offset to be set beyond the end of the existing data in the file.
- Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, is returned. Otherwise, (off_t)-1 is returned, errno is set to indicate the error and the file offset will remain unchanged.

Duplicating files

```
#include <unistd.h>
int dup(int fildes);
int dup2(int fildes, int fildes2);
```

- The dup() and dup2() functions provide an alternative interface to the service provided by fcntl() using the F_DUPFD command.
- The dup() system call duplicates the given descriptor by copying its contents to new descriptor entry and returns index of new descriptor.
- With dup2() we specify the value of the new descriptor with fildes2. If fildes2 is already open, it is closed first. The new file descriptor that is returned as a value of the functions shares the same file table entry as the fildes argument. Useful to redirect the standard input or output to some other files.
- Upon successful completion a non-negative integer, namely the file descriptor, is returned.
- Otherwise, -1 is returned and errno is set to indicate the error.

Information about files

When you give 'ls -l' command, it displays lot of information about each file. To get such file information from a program we can use the following system calls

- –int stat(char *pathname,struct stat *buf)
- –int fstat(int filedes, struct stat *buf)
- –int lstat(char *pathname,struct stat *buf)

All the above 3 functions gives the information about a given file in the **stat** structure.

The first function **stat**() takes the name of the file as input, whereas the **fstat** function takes the file descroptor as input. When a symbolic link file is given as input, the **stat**() function gives the information about the file to which this symbol link is pointing to.

But lstat function gives the information abou theis symbol link itself.

Structure declaration for stat structure

```
Struct stat
•{
        mode_t st_mode;         //file type and mode(permissions)
    ino_t st_ino;                   //I-node number
    dev_t st_dev;                   //device number(file system)
    dev_t st_rdev;                  //device number for special files
    nlink_t st_nlink;           //number of links
    uid_t st_uid;                   //user id of owner
    gid_t st_gid;                   //group ID of owner
    off_t st_size;                  //size in bytes, for regular files
    time_t st_atime;            //time of last access
    time_t st_mtime;                //time of last modification
    time_t st_ctime;            //time of last file status change
    Long st_blksize;                //best I/O block size
    Long st_blocks;             //number of 512 byte blocks allocated
    }
```

Changing permissions

File ownership and group are changed with three similar system calls.
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group)
int lchown(const char *path, uid_t owner, gid_t group);

chown works on a pathname argument. **fchown** works on open file, and lchown works on symbolic links instead of on the files pointer to by symbolic links.
Unix systems keeps the info that mapped user names with user ids.
Earlier systems used to store this info in /etc/passwd and etc/gropu.
Present day systems uses API to get the information stored in databases.
Using from Command Line: **chown root:root dumpdemo.txt**

Changing file ownership & Timestamps

Changing permissions is done with one of two system calls.

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

Chmod() works on pathname argument, and fchmod() works on an open file. The mode value is created in the same way for open() and create().

Using Chmod from command line:

utime() system call allows you to change a file's access and modification timestamps.

Standard I/O library functions

VIVEN

Embedded Academy

Agenda

- fopen(), fread(), fwrite(), fclose(), fseek()
- Relationship between file descriptor and file pointer
- Character at a time I/O
- Line at a time I/O
- Formatted I/O

fopen()

*FILE *fopen(const char *path, const char *mode);*

#include <stdio.h>

- The fopen function opens the file whose name is the string pointed to by path and associates a stream with it.
- The argument mode points to a string beginning with one of the following sequences (Additional characters may follow these sequences.)

  r: Open text file for reading. The stream is positioned at the beginning of the file.

  r+: Open for reading and writing. The stream is positioned at the beginning of the file.

fopen() .....

w: Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+: Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

a : Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

a+: Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

Upon successful completion of fopen, it returns a FILE pointer.

Otherwise, NULL is returned and the global variable errno is set to indicate the error.

fread()

*size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);*

- binary stream input
- *fread* reads nmemb elements of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.
- *fread* return the number of items successfully read. If an error occurs, or the end-of-file is reached, the return value is a short item count (or zero).
- *fread* does not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred.

fwrite()

*size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);*

- binary stream output

- The function *fwrite* writes nmemb elements of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr.

- *fwrite* return the number of items successfully written. If an error occurs, or the end-of-file is reached, the return value is a short item count (or zero).

fclose()

*int fclose(FILE *stream);*

- close a stream.

- *fclose* function dissociates the named stream from its underlying file or set of functions. If the stream was being used for output, any buffered data is written first, using fflush(3) .

- Upon successful completion 0 is returned. Otherwise, EOF is returned and the global variable errno is set to indicate the error.

fseek()

*int fseek(FILE *stream, long offset, int whence);*

- reposition a stream
- *fseek* function sets the file position indicator for the stream pointed to by stream.
- the new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence.
- upon successful completion fseek, return 0. Otherwise, -1 is returned and the global variable errno is set to indicate the error.

File pointers and File descriptors

- Files are designated in C programs either by file pointers or file descriptors .
- A file opened by a process is identified by a file descriptor which is private to that process.
- Associated with each file descriptor is a file pointer which records the current location in the file.
- The Standard I/O library functions for ISO C use file pointers.
- The Unix I/O functions use file descriptors.
- both provide logical designations called handles for performing device-independent input and output.

Character-at-a-time I/O

- Read and write one character at a time with std I/O functions.
- Functions to read one character at a time (Input).

```
# include <stdio .h>
int fgetc ( FILE * stream );
int getc ( FILE * stream );
int getchar ( void );
int getw ( FILE * stream );
```

- Functions to write one character at a time (Output).

```
int fputc ( int c, FILE * stream );
int putc ( int c, FILE * stream );
int putchar ( int c);
int putw ( int w, FILE * stream );
```

Line-at-a-time I/O

- read or write a line at a time, using *fgets()* and *fputs()*.
- Line-at-a-time input is provided by two functions:

```
# include <stdio .h>
char *fgets ( char *str , int size , FILE * stream );
char *gets ( char * str );
```

- Line-at-a-time output is provided by two functions:

```
# include <stdio .h>
int fputs ( const char *str , FILE * stream );
int puts ( const char * str );
```

Formatted I/O

- Formatted I/O (Output) functions:

```
# include <stdio .h>
int printf ( const char * format , ...);
int fprintf ( FILE * stream , const char * format , ...);
int sprintf ( char *str , const char * format , ...);
```

- The functions *vprintf()*, *vfprintf()*, *vsprintf()* are similar to the previous three, but the variable argument list (. . .).

```
# include <stdarg .h>
int vfprintf(FILE *stream, const char *format, va_list ap);
int vprintf(const char *format, va_list ap);
int vsprintf(char *s, const char *format, va_list ap);
```

Formatted I/O ....

- Formatted I/O (Input) functions:

```
                # include <stdio .h>
int scanf ( const char * format , ...);
int fscanf ( FILE * stream , const char * format , ...);
int sscanf ( const char *str , const char * format , ...);
                # include <stdarg .h>
int vscanf ( const char * format , va_list ap );
int vsscanf ( const char *str , const char * format , va_list ap );
int vfscanf ( FILE * stream , const char * format , va_list ap);
```

Reading and Writing Structures to Files

VIVEN

Embedded Academy

Agenda

- In ASCII format
- In binary format
- Modifying a structure in the file

VIVEN
Embedded Academy

USERS & GROUPS

Users and groups

- Every **user** has a unique login name and an associated numeric **user identifier**(**UID**). **Users** can belong to one or more **groups**. Each **group** also has a unique name and **group identifier**(**GID**).
- The primary purpose of **user** and **group** ID's is to determine ownership of various **system resources** and to control the permissions granted to processes accessing those resources.

  Ex: Each file belongs to a particular user and group, and each process has a number of user and group Ids that determine who owns the process and what permissions it has when accessing a file.
- Different system files are used to define the **users** and **groups** on the system, and you have various library functions used to retrieve information from these files. Ex: crypt() function which is used to encrypt and authenticate login passwords.

The password file: /etc/passwd

- The system ***password file***, ***/etc/passwd*** contains one line for each user account on the system. Each line is composed of seven fields separated by colons(:), as in the following example.

    vamsi:x:500:500:vamsi krishna:/home/vamsi:/bin/bash

- In order, these fields are as follows:
- **Login name**: This is the unique name that the user must enter in order to login. Often this is called as **username**. Its the login name in human readable (symbolic) identifier corresponding to the numeric user identifier. Programs such as **“ls -l”** displays the user name rather than the numeric user id associated with the file.
- **Encrypted Password:** This field contains a 13-character encrypted password.

Embedded Academy

The Environment of a UNIX PROCESSES

Processes and Programs

A process is an instance of an executing program. A program is a file containing the range of information that describes how to construct a process at runtime.

- **Binary format identification:** Each program file includes meta-inforamtion describing the format of the executable file. This enables the kenrnel to interpret the remaining information in the file. Two wildely formats for UNIX executable files were a.out(assembly output) in ELF format and later, more sophisticated COFF(Common object file format). Now a days, most linux systems emply the Executable and Linking Format(ELF), which has lot of advantages over other formats.
- **Machine-language instructions:** These encode the algorithm of the program.
- **Program entry-point address:** This identifies the location of the instruction at which execution of the program should commence.

Processes and Programs (Contd...)

VIVEN Embedded Academy

- **Data:** The program file contains values used to initialize variables and also literal constants used by the program (e.g., strings).
- **Symbol and relocation tables:** These describe the locations and names of functions and variables within the program. These tables are used for a variety of purposes, including debugging and run-time symbol resolution (dynamic linking).
- **Shared-library and dynamic linking information:** The program file includes fields listing the shared libraries that the program needs to use at run time and the pathname of the dynamic linker that should load these libraries.
- **Other information:** The program file contains vairous other information that describes how to construct a process.

Processes and Programs (Contd...)

- One program may be used to construct many processes, or put conversely, many processes may be running the same program.
- A process is a abstract entity, defined by the kernel, to which system resources are allocated in order to execute a program.
- From the kernel point of view, a process consists of user-space **memory containing program code** and variable used by that code, and a range of kernel data structures that maintain information about the state of the process.
- The information recorded in the kernel data structures includes variaous **identifier numbers**(IDs) associated with the process, **virtual memory tables**, the **table of open file descriptors**, information relating to **signal delivery** and **handling**, process **resource usage** and **limits**, the **current working directory** and a host of other information.

Process ID

- Each process has a process ID(PID, a positive integer that uniquely identifies the process on the system. Process Ids are used and returned by a variety of system calls. For example kill() system call allows the caller to send a signal to a process with a specific process ID.
- The process ID is also useful if we need to build an identifier that is unique to a process. A common example of this is the use of the process ID as part of a process-unique filename.
- The getpid() system call returns the process ID of the calling process.

  pid_t getpid(void);
- The linux kernel limits process ID's to being less than or equal to 32,727. When a new process is created, it is assigned the next sequentially available process ID. Each time the limit of 32,727 is reached, the kernel resets the process ID counter to 300 rather than 1. This done because many low-numbered process ID's are in permanent use by system process and daemons.

Parent Process ID

- Each process has a parent – the process that created it. A process can find out the ***process ID*** of its parent using the ***getppid()*** system call.

  pid_t getppid(void);

- In effect, the ***parent process ID*** attribute of each process represents the tree-like relationship of all processes on the system. The parent of each process has its own parent, and so on, going all the way back to process 1, ***init***, the ancestor of all processes.
  - Note: the family tree can be viewed using the ***pstree*** command.
- If a child process becomes ***orphaned*** because its "birth" parent terminates, then the child is adopted by the ***init*** process, and subsequent calls to ***getppid()*** in the child return 1.
- The parent of any process can be found by looking at the ***Ppid*** field provided in the linux specific ***/proc/PID/status*** file.

Memory layout of a process

The memory allocated to each process is composed of a number of parts, usually referred to as segments. These segments are as follows.

- The ***text segment*** contains the machine-language instructions of the program run by the process. The text segment is made ***read-only*** so that a process doesnt accidently modify its own instruction via bad pointer value. Since many processes may be running the same program, the text segment is made sharable so that a single copy of the program code can be mapped into the virtual address space of all of the processes.
- The ***initialized data segment*** contains global and static variables that are explicitly initialized. The values of these varialbles are read from the executable file when the program is loaded into memory.
- The ***uninitialized data segment*** contians global and static variables that are not explicitly intialized. Before starting the program, the system initializes al memory in this segment to 0.

Memory layout of a process

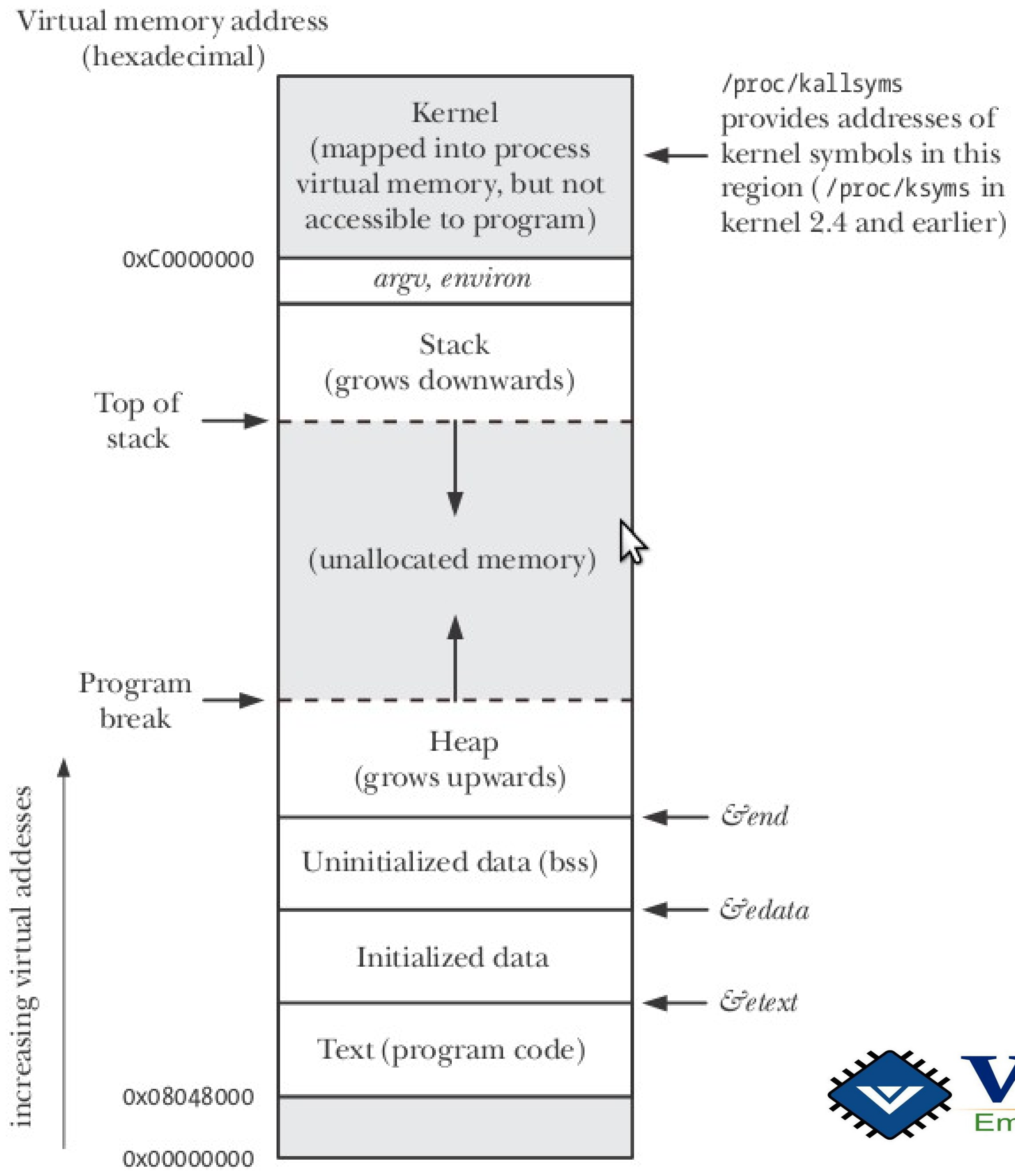
For historical reasons this is called ***BSS segment***, a name derived from an old assembler mnemonics for "block started by symbol". The main reason for placing global and static variables that are initialized into a separate segment from those that are uninitialized is that, when a program is stored on disk, it is not necessary to allocate space for the uninitialized data. Instead, the executable merely needs to record the location and size required for the uninitialized data segment, and this space is allocated by the program loader at run time.

- The ***stack*** is a dynamically growing and shrinking segment containing stack frames. One stack frame allocated for each currently called function. A frame stores the functions local variables(so called automatic variables), arguments, and return value.
- The ***heap*** is an area from which memory(for variables) can be dynamically allocated at run time. The top end of the heap is called the program break. ***Size*** displays size fo text, data and BSS segments.

Note

- An application binary interface(ABI) is a set of rules specifying how a binary executable should exchange information with some service(e.g., the kernel or a library at runtime. Among other things, an ABI specifies which registers and stack locations are used to exchange information, and what meaning is attached to the exchanged values. Once compiled for a particular ABI, a binary executable should be able to run on any system presenting the same ABI.

Typical Memory Layout of a process - Linux/x86-32


Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
0x08048000
0x00000000
increasing virtual addesses

VIVEN Embedded Academy

Note

- The C program enivironment on most UNIX implementations provides three global symbols: **etext, edata,** and **end**. These symbols can be used from within a program to obtain the address of end of the program text, the end of the initialized data segment, and the end of the uninitialized data segment. To make use of these symbols, we must explicitly declare them, as follows
  - Extern char etext, edata, end;

  &etext gives the address of the end of the program text.
- In the previous figure the space labeled **argv**, **environ** at the top of this diagram holds the program command-line arguments(avilable in C via the argv argument of the main functions) and the process environemnt list. The **argv** and **environ** arrays, as well as the strings they initialize point to, reside in a single contiguous are of memory just above the process stack. The grayed-out areas represent invalid ranges in the process virtual address space; that is areas for which page tables have not been created.

Virtual Memory Management

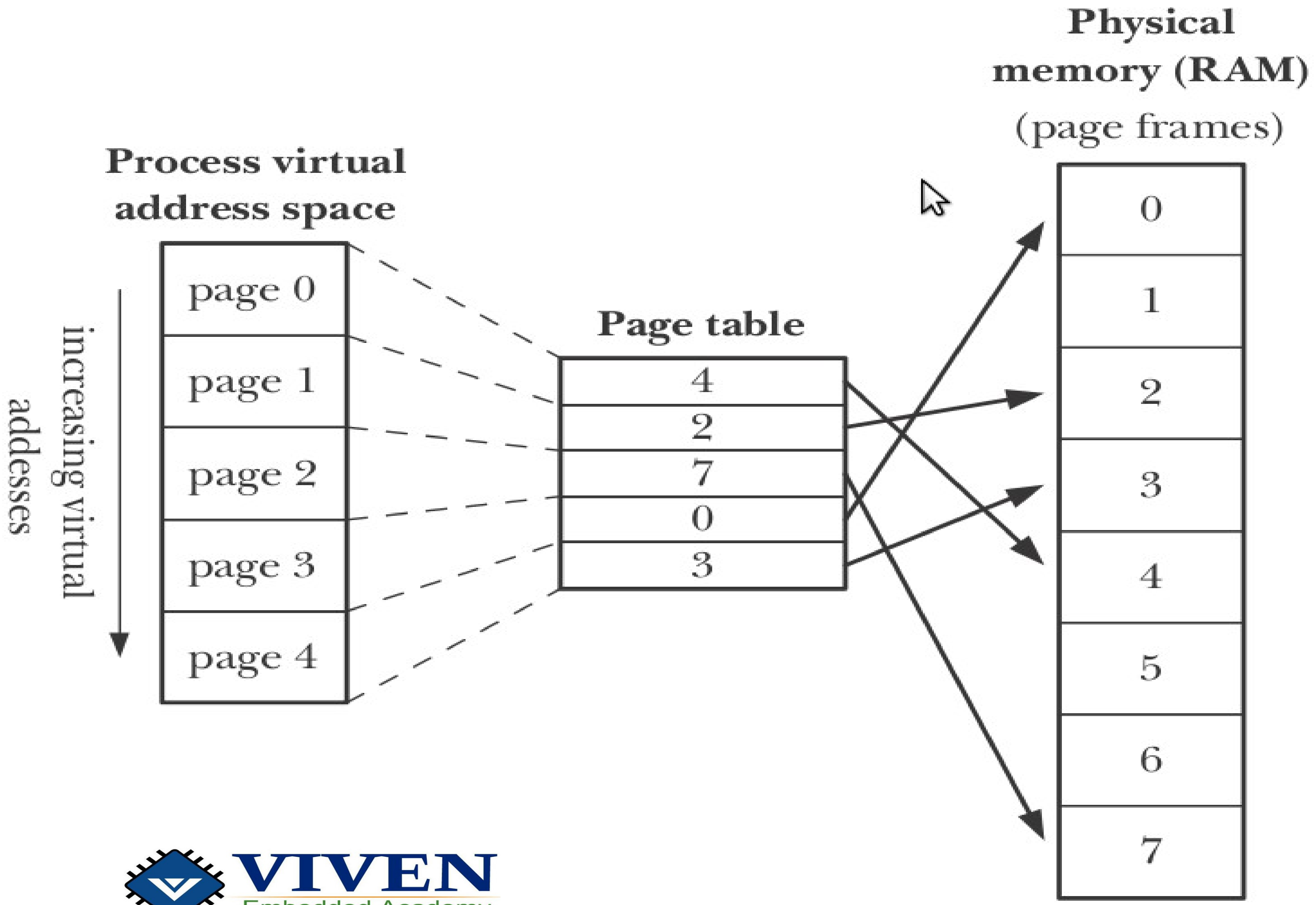
- Most modern kernels including linux employs technique known as virtual memory management. The aim of this technique is to make efficient use of both the CPU and RAM by exploiting a property that is typical of most of programs: **Locality of reference.** Most programs demonstrate two kinds of locality:
  - **Spatial locality** is the tendency of a program to reference memory addresses that are near those that were recently accessed (because of sequential processing of instruction, and sometimes, sequential processing of data structures).
  - **Temporal Locality** is the tendency of a program to access the same memory address in the near future that it accessed in the recent past(because of loops).
- The above two points refer that it is possible to execute a program while maintaining only part of its address space in RAM.

Virtual Memory Management(Contd)

- A **virtual memory scheme** splits the memory used by each program into small fixed-size units called pages. Correspondingly, RAM is divided into a series of **page frames** of the same size. At any one time, only some of the pages of a program need to be resident in physical memory page frames; these pages form the so-called **resident set**. Copies of unused pages of a program are maintained in the **swap area-**a reserved area of disk space used to supplement the computer's RAM- and loaded into physical memory only when required.
- When a process references a page that is not currently resident in physical memory, a **page fault** occurs, at which point the kernel suspends execution of the process while the **page fault** occurs, at which point the kernel suspends execution of the process while the page is loaded infrom disk into memory.

VIVEN Embedded Academy

Virtual Memory Management(Contd)


Process virtual address space
page 0
page 1
page 2
page 3
page 4
increasing virtual addesses
Page table
4
2
7
0
3
Physical memory (RAM) (page frames)
0
1
2
3
4
5
6
7

VIVEN Embedded Academy

Virtual Memory Management(Page Tables)

- In order to support the virtual memory organization, the kernel maintains a ***page table*** for each process. The ***page table*** describes the location of each page in the process's virtual address space(the set of all virtual memory pages avialable to the process). Each entry in the page table either indicates the location of a virtual page in RAM or indicates that it currently resides on disk.
- Not all address ranges in the process's virtual address space require page table entries. Typically, large ranges of the potential virtual address space are unused, so that it isnt necessary to maintain corresponding page table entries. If a process tries to access an address for which there is no corresponding page-table entry, it receives a SIGSEGV signal.

Range of valid Virtual addresses

A process's range of valid virtual addresses can change over its lifetime as the kernel allocates and deallocates pages(and page-table entries) for the process. This can happen in the following circumstances.

- As the **stack** grows **downward** beyond limits previously reached.
- When memory is **allocated** or **deallocated** on the **heap**, by rasing the program using **brk(), sbrk()**, or the **malloc** family of functions.
- When **System V shared memory** regions are **attached** using **shmat**() and **detached** using **shmdt**().
- When **memory mappings** are created using **mmap**() and **unmapped** using **munmap**()

Virtual Memory Management(Advantages)

Virtual memory management separates the virtual address space of a process from the physical address space of RAM. This provides many advantages:

- Processes are isolated from one another and from the kernel, so that one process can't read or modify the memory of another process or the kernel. This is accomplished by having the page-table entries for each process point to distinct sets of physical pages in RAM(or in the swap area).
- Where appropriate, two or more processes can share memory. The kernel makes this possible by having page-table entries in different processes refer to the same pages of RAM. Memory sharing occurs in two common circumstances.
  - Multiple processes executing the same program can share a single (read-only) copy of the program code. This type of sharing happens when multiple processes execute the same program file(or load the same library).
  - Processes can use the shmget() and mmap() system calls to explicitly requests sharing of memory regions with other processes(IPC).

Virtual Memory Management(Advantages)

- Page table entries can be marked to indicate that the contents of the corresponding page are readable, writable, executable or some combination of these protections. Where multiple processes share pages of RAM, it is possible to specify each process has different protections on the memory. For example one process might have read-only access to a page, while another has read-write access.
- Programmers and tools such as the compilers and linker, dont need to be concerned with the physical layout of the program in RAM.
- Because only part of a program needs to reside in memory, the program loads and runs faster. Furthermore, the memory footprint(i.e., virtual size) of a process can exceed the capacity of RAM.

One final advantage of virtual memory management is that since each process use less RAM, more processes can simultaneusly be held in RAM. This leads to better CPU utilization, since it increases the likelihood that at any moment in time, there is atleast one process that the CPU can execute.

The Stack and Stack Frames

- The **stack** grows and shrinks linearly as functions are called and return. For linux on x86-32 architecture(and on most other Linux and UNIX implementations), the stack resides at the **high end** of memory and grows downwards(towards the heap). A special purpose register, the **stack pointer,** tracks the **current top** of the stack. Each time a function is called, an additional frame is allocated on the stack, and this frame is removed when the function returns.
- Sometimes, the term **user stack** is used to distinguish the stack we describe here from the **kernel stack**. The **kernel stack** is a **per-process** memory region maintained in **kernel memory** that is used as the stack for execution of the functions called internally during the execution of a system call.(The kernel can't employ the user stack for this purpose since it resides in unprotected user memory).

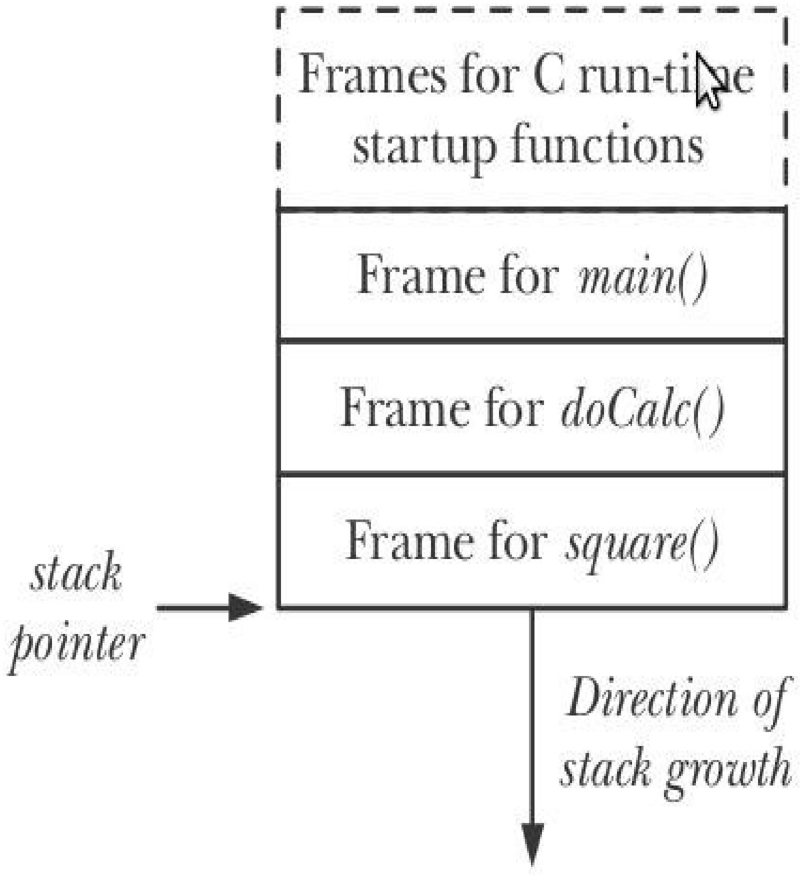
Contents in User Stack

Each (user) Stack frame contains the following information.

- **Function arguments and local variables:** In C these are referred to as **automatic variables,** since they are automatically created when a function is called. These variables also automatically disappear when the function returns since the stack frame disappears, and this forms the primary semantic distiction bewteen automatic and static(and global) variables: the latter have a permanent existence independent of the execution of functions.
- **Call linkage information:** Each function uses certain CPU registers, such as the program counter, which points to the next machine-language instruction to be executed. Each time one function calls another, a copy of these registers is saved in the called function's stack frame so that when the function returns, the appropriate register values can be restored for the calling function.

Example of process Stack

Since functions can call one another, there may be multiple frames on the stack. (If a funciton calls itself recursively, there will be multiple frames on the stack for that function. During the execution of three functions main() doCalc() and square()



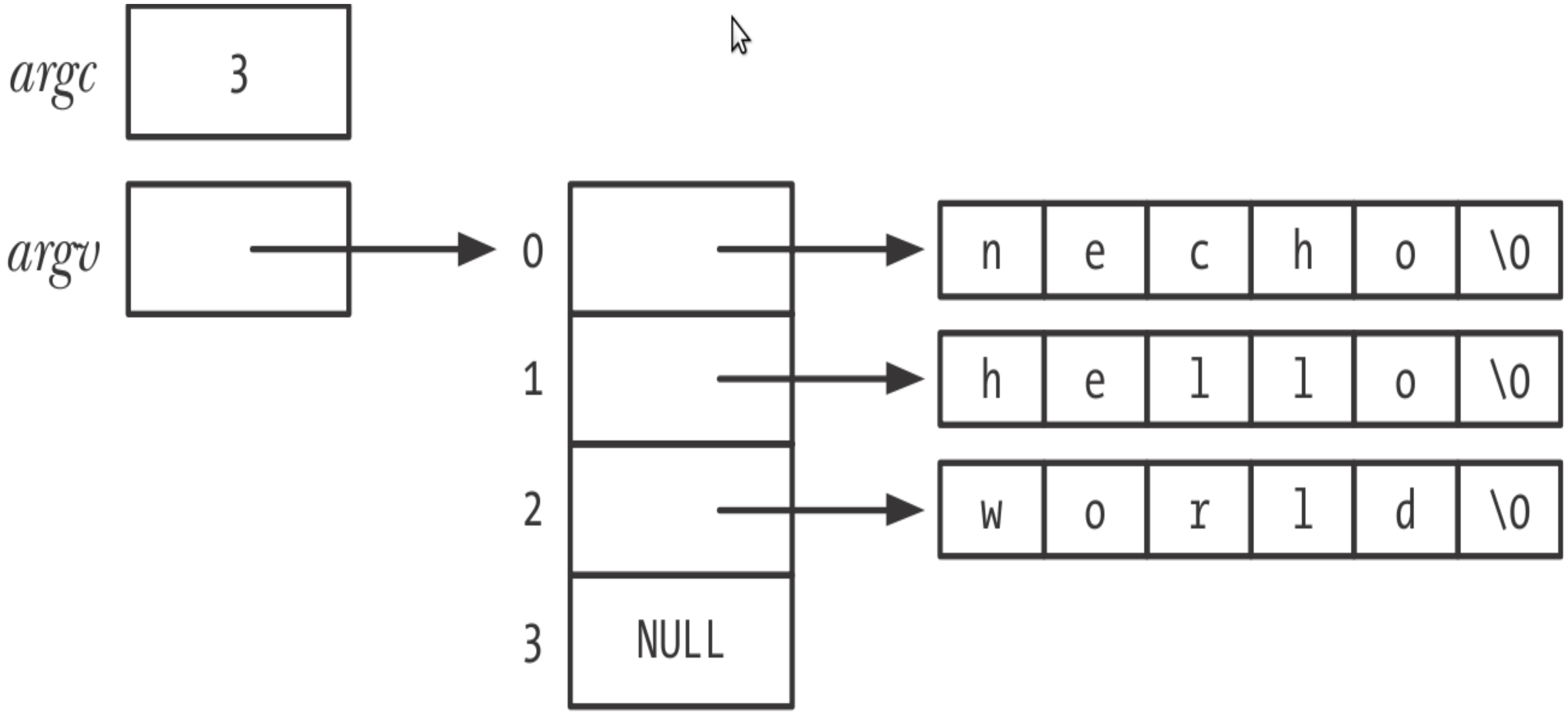
Command-line Arguments(argc, argv)

Every C program must have a function called main(), which is the point where execution of the program starts. When the program is executed, the command line arguments(the separate words parsed by the shell) are made available via two arguments to the function main().

- The **first argument**, int argc, indicates how many command-line arguments there are.
- The **second argument**, char *argv[], is an array of pointers to the command line arguments, each of which is null-terminated character string. The first of these strings, in argv[0], is the name of the program itself. The list of pointers in argv is terminated by a NULL pointer(i.e., argv[argc] is NULL).

One limitation of the *argc/argv* mechanism is that these variables are avialable only as arguments to main(). To portably make the command-line arguments avialable in other functions, we must either pass argv as an argument to those functions or set a global variable pointing to argv.

Values of argc and argv for the command necho hello world



Command-line Arguments(argc, argv)

There are couple of nonportable methods of accessing part of all of this information from anywhere in a program:

- The command-line arguments of any process can be read via the Linux-specific ***/proc/PID/cmdline*****,** with each argument being terminated by a null byte. (A program can access its own command-line arguments via /proc/self/cmdline).
- The GNU C library provides two global variables that may be used anywhere in a program in order to obtain the name used to invoke the program(i.e., the first command-line argument). The first of these, ***program_invocation_name*****,** provides a version of this name with any directory prefix stripped off(i.e., basename component of the pathname). Declarations for these two variables can be obtained from <errno.h> by defining macro _GNU_SOURCE.

Full and simple forms of the above names with which the program is invoked can be found in error.h

- Many programs parse command line options(arguments beginning with a hyphen) using the **getopt()** library function.

Environment List

- Each process has an associated array of strings called the ***environment list,*** or simply the ***environment***. Each of these strings is a definition of the form name=value. Thus the environment represents a set of name=value pairs. The names in the list are referred to as ***environment variables***.
- When a process is created, it **inherits** a copy of its **parent's environment**. The environment provides a way to transfer information from a **parent** process to its **child**. Since the child gets a copy of its parent's environment at the time it is created, this transfer of information is **one-way** and **once-only**. After the child process has been created, either process may change its own environment, and these changes are not seen by the other process.
- **Environment variables** are used for variety of purposes. The Shell defines and uses a range of variables that can be accessed by ***scripts*** and ***programs*** executed from the ***shell***. These include the variable ***HOME***, which specifies the pathname of the user's login directory, and the variable ***PATH***, which specifies a list of directories that the shell should search when looking for programs corresponding to commands entered by the user.

Setting environment variables

- Some library functions allow their behavior to be modified by setting environment variables. In most shells, a value can be added to the envirnment using the **export** command.

  $SHELL=/bin/bash ***Create a shell variable***

  $export SHELL ***Put variable into shell process's environment.***

- In **bash** and **korn** shell, this can be abbreviated to:

  $export SHELL=/bin/bash

- In the C shell, the setenv command is used instead:

  % setenv SHELL /bin/bash

- The above commands permanently add a value to the shell's environment and this environment is then inherited by all child processes that the shell creates. At any point, an environment variable can be removed with the **unset** command(unsetenv in the C shell).
- The **printenv** & **env** command displays the current environment list.

  $ printenv

  $ env

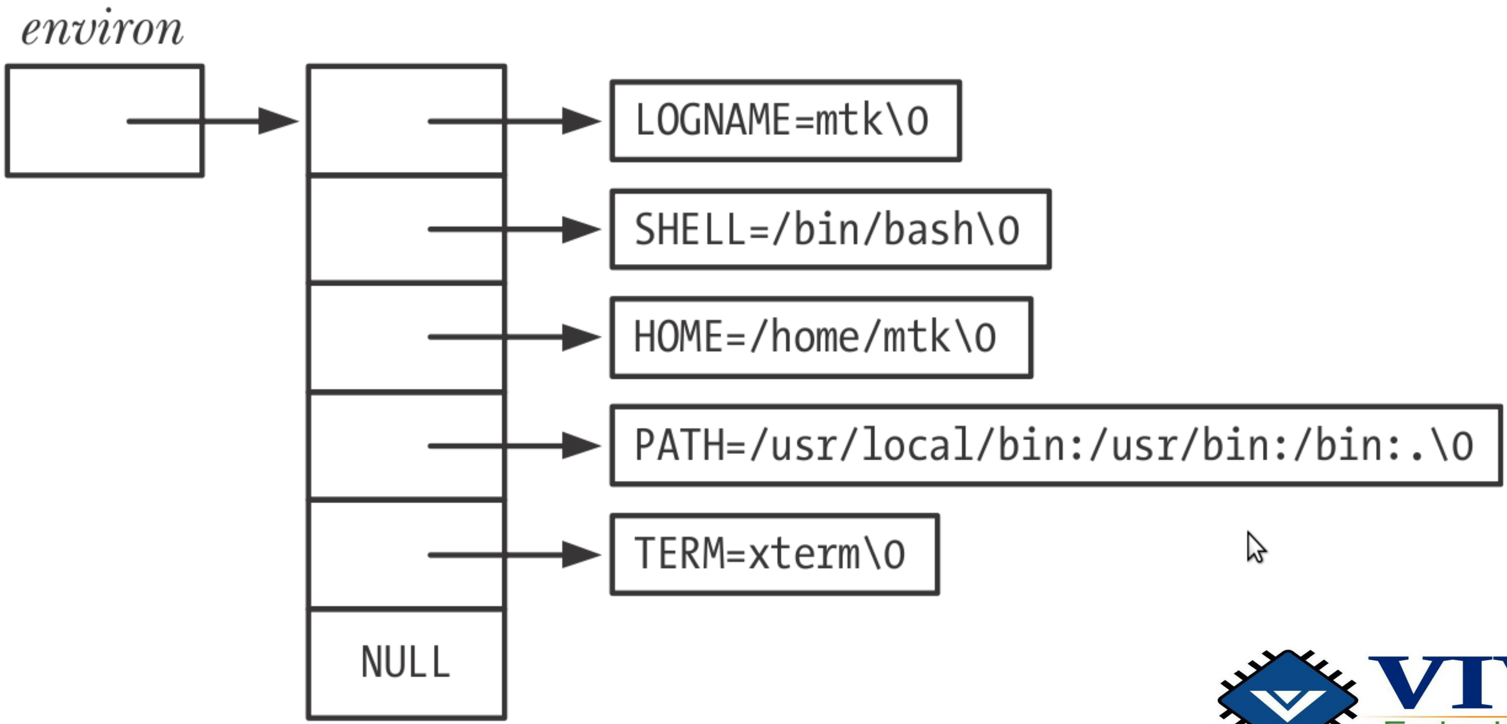
Setting environment variables (Contd...)

- In the **Bourne** shell and its descendants(e.g., bash and the Korn shell), the following syntax can be used to add values to the environment used to execute a single program, without affecting the parent shell(and subsequent commands):
  - ***$ NAME=value program***

This adds a definition to the environment of just child process executing the named program. If desired, multiple assignments (delimited by white space) can precede the program name.

Accessing the environment from a program

- Within a C program, the environment list can be accessed using the global variable **char **environ**.(The C run-time startup code defines this variable and assigns the location of the environment list to it.) Like ***argv***, ***environ*** points to a NULL-terminated list of pointers to null terminated strings. The below figure shows the environment list of data structures as they would appear for the environment displayed by the command above.



VIVEN Embedded Academy

Accessing invidual environment variables

- The getenv() function retrieves individual values from the process environment.

  char *getenv(const char *name);

- Given the name of an environment variable, getenv() returns a pointer to the corresponding value string. If ***SHELL*** was specified as the name argument ***/bin/bash*** would be returned.
- The application should not modify the string returned by ***getenv().*** This is beacause(in most implementations) this string is actually part of the environment (i.e., the ***value*** part of the ***name=value*** string). If we need to change the value of an environment variable, then we can use the ***setenv()*** or ***putenv()*** functions.

VIVEN Embedded Academy

Modifying the environment

- Sometimes, it is useful for a process to modify its environment.
  - One reason is to make a change that is visible to child process subsequently created by that process.
  - Another possiblity is that we want to set a variable that is visible to a new program to be loaded into the memory of this process. In this sense, the environment is not just a form of interprocess communication but also a method of interprogram communication. (Point becomes clearer in ***exec()*** function permit a program to replace itself by a new program within the same process).
- The ***putenv()*** function adds a new variable to the calling process's environment or modifies the value of the existing variable.

  int putenv(char *string);
  - The ***string*** argument is a pointer to a string of the form name=value. After the putenv() call, this string is part of the environment.

Modifying the environment

- The ***setenv()*** function is an alternative to putenv() for adding a variable to the environment.

int setenv(const char *name, const char *value, int overwrite)

- The ***setenv()*** function creates a new environment variable by allocating a memory buffer for a string of the form ***name=value,*** and copying the string pointed to by ***name*** and ***value*** into that buffer. We dont need to supply an equal sign at the end of the ***name*** or the start of a ***value.***
- The ***setenv()*** function doesnt change the environment if the variable identified by ***name*** already exists and ***overwrite*** has the value 0. If ***overwrite*** is nonzero, the environment is always changed.
- The **unsetenv()** function removes the variable identified by ***name*** from the environment.

int unsetenv(const char *name);

- It is useful to erase the entire environment by assigning ***NULL*** to ***environ*** or ***int clearenv(void)***;

The Environment of a Unix Process

Agenda

- How a C program starts and terminates as a process
- Memory layout of a C program
  - main function
  - command-line arguments
  - environment variables
- exit(), _exit and atexit() functions

C program as a process

Processes are the primitive units for allocation of system resources.

- Each process has its own address space and (usually) one thread of control.
- A process executes a program;
- You can have multiple processes executing the same program, but each process has its own copy of the program within itsown address space and executes it independently of the other copies.

Process attributes

–A unique process identifier (the PID).
– A parent process(with an associated identifier, the PPID).
– Permission identifiers (UID,GID).
– An address space, separate from those of all other processes.
–A program running in that address space.
–A current working directory ('.').
–A current root directory (/)
–File descriptor table. A set of open files, directories, or both.
–A set of strings representing the environment.
–A scheduling priority
–Settings for signal disposition.
–A controlling terminal.

Contd....

- When the main function begins execution, all of these things have been put in place for the running program.
- System calls are available to query and change each of the attribute.
- New processes are always created by an existing process. The existing process is called as parent, and the new process is called as child.
  - Upon booting, kernel program(PID – 0) runs **/sbin/init** which has PID 1 and servers several administrative functions. All other processes are descendants of init.
- Each process's address space(memory) is separate from that of every other. Unless two process have made an explicit arrangement to share memory, one process cannot affect the address space of others.

Input & Output

- All programs start out with three files already open:
  - Standard input
  - Standard output
  - Standard error
- These are where input comes from, output goes to, and error messages go to.
- A parent process can open additional files and have them already available for a child process.

Environment

–The environment is a set of strings, each of the form 'name=value'.

– Functions exist for querying and setting environment variables, and child process inherit the environment of their parents.

–Typical environment variables are thinks like PATH and HOME. Many programs look for the existence and value of specific environment variables in order to control their behaviour.

Getenv,
setenv,
putenv,
unsetenv,
clearenv

C program as a process ......

–The system starts a C program by calling the function

main()

–Without main you won't even be able to link your program without errors.

–In ANSI C you can define *main()* either to take no arguments, or to take two arguments that represent the command line arguments to the program, like this:

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

C program as a process....

–In Unix systems you can define main a third way, using three arguments:

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

–When a program is executed, it receives information about the context in which it was invoked in two ways.

- The first mechanism uses the argv and argc arguments to its main function.
- The second mechanism uses environment variables.

C program as a process....

- The usual way for a program to terminate is simply for its main function to return.
- A process terminates normally when the program calls exit. Returning from main is equivalent to calling exit, and the value that main returns is used as the argument to exit.
- The exit status value returned from the main function is used to report information back to the process's parent process or shell.

Memory Layout of a C program

- The basic process memory layout is the result of "linking" (combining the various .o and .a (or .so) files into an executable program format) plus arranging the disk file's data into main memory (allocating data storage and such).
- Process memory is divided into a "text segment" and a "data segment".
  - The "text segment" is the code, the machine language instructions that are your program.
  - It is made read-only via the MMU.

Memory Layout of a C program ......

- The "data segment" is more complicated; it is not read-only, far from it; it changes as your program executes. Some of its components are:
  - command line arguments(argc, argv)
  - the environment (envp)
  - these days usually does not include string literals, which are put into the text segment to make them unwritable.

Memory Layout of a C program ……

- all variables
  - some variables' initial values are determined by values in the a.out file; this is loaded first.
  - the remaining variables are initialized to all bytes zero and thus need not be stored in the a.out file format, just the byte count of all this. This is known as the "BSS" area.
- the "stack", or this may be a separate segment;
- the "heap", which consists of whatever data space is acquired during process execution.

Memory Layout of a C program ......

–Variables with global lifetime are allocated in the data segment.

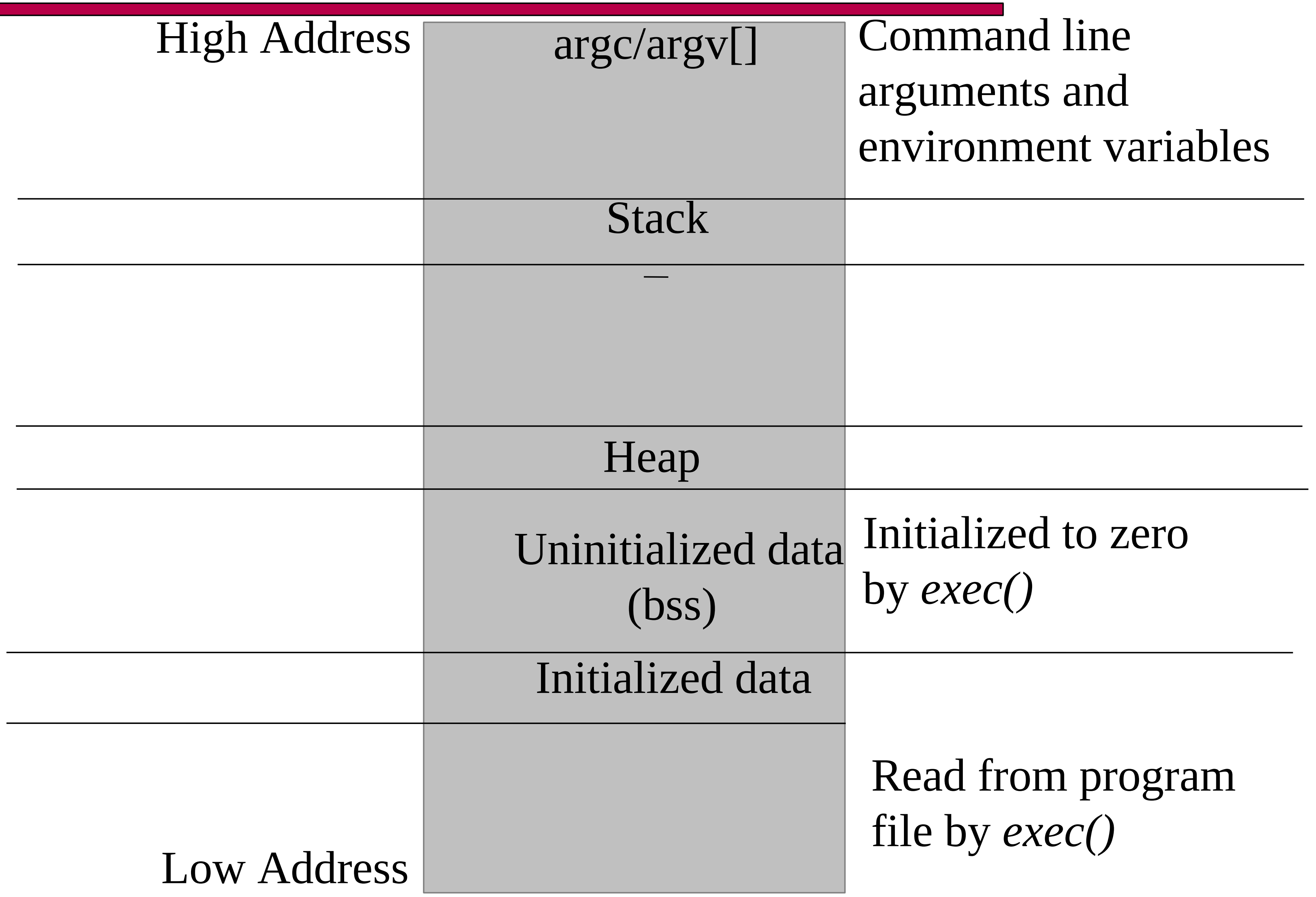
–Scope is an error-checking issue for the C compiler.

  # global versus stack is a lifetime issue.

  - Function-static is in global area.

  - But "auto" variables are allocated on the stack.

Typical logical memory layout of a process


High Address
argc/argv[]
Command line arguments and environment variables
Stack
Heap
Uninitialized data (bss)
Initialized to zero by *exec()*
Initialized data
Read from program file by *exec()*
Low Address

Process Termination

- There are eight ways for a process to terminate.
  - •Return from main
  - •Calling **exit.**
  - •Calling _exit or _Exit.
  - •Return of the last thread from its start routine.
  - •Calling **pthread_exit** from the last thread.

Abnormal termination occurs in three ways:
  - •Calling **abort.**
  - •Receipt of signal.
  - •Response of the last thread to a cancellation request.

exit(), _exit and atexit()

- Terminates a process.
- Standard C Library (libc.a)
- #include <stdlib.h>

```
void exit ( Status)
int Status;
void _exit ( Status)
int Status;
```

- *#include <sys/limits.h>*

```
int atexit ( Function)
void (*Function) (void);
```

exit()

- When a program exits, it can return to the parent process a small amount of information about the cause of termination, using the exit status.
- This is a value between 0 and 255 that the exiting process passes as an argument to exit.
- The most common convention is simply 0 for success and 1 for failure
- exit subroutine terminates the calling process after calling the standard I/O library _cleanup function to flush any buffered output.

_exit()

- It is declared in the header file unistd.h
- The _exit function is the primitive used for process termination by exit.
- Calling this function does not execute cleanup functions.

atexit()

- The *atexit()* function registers the function to be called at normal program termination. The function is called with no arguments.
- The return value from *atexit()* is zero on success and nonzero if the function cannot be registered.
- Before the program exits, exit() calls all functions registered with atexit(), flushes and closes all open file streams.

Process System Calls

VIVEN
Embedded Academy

Agenda

- Process Identifiers
- fork()
- vfork()
- exit()
- wait()
- waitpid()
- execv()

Process Identifiers

–Every process has unique process ID, a non-negative number.

–Although unique, process Ids are reused; as process terminate, their IDs become candidates for reuse.

–Process 0 is usually scheduler process and is often known as swapper. It is part of kernel and known as system process.

–Process ID 1 is usually the init program, is invoked by kernel at the end of booting procedure.

Process Identifiers ......

- In addition to process ID, there are other identifiers for every process such as ppid(parent pid), uid(user id) and euid(effective uid).
- The following functions return these identifiers.

```
#include<unistd .h>
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
uid_t geteuid(void);
```

Viewing Active Process ......

-The ps command displays the process that are running on your system. Invoking ps displays the process controlled by the terminal or terminal window in which ps is invoked. For example

```
%ps
    PID TTY          TIME CMD
   1813 pts/0    00:00:00 bash
   1830 pts/0    00:00:00 ps
```

– The invocation of ps shows two process. The first bash, is the shell running on the terminal. The second is the running instance of the ps program itself.

– For a more detailed look at whats running on your system invoke this.

```
%ps -e -o pid,ppid,command
```

-e option instructs to display all process running on the system.

-o pid,ppid,command option tells ps what information to show about each process.

fork()

- fork() function is used to create child processes.

  #include <unistd.h>

  pid_t fork(void);

- The new process (child process) shall be an exact copy of the calling process (parent process).
- The child process shall have a unique process ID.
- When a process calls fork() system call, control goes to the kernel, and fork() system call inside the kernel will execute. The system call creates a new PCB in the kernel and new virtual memory in the user space. Next it copies all the virtual memory of original process (parent process). Into this new process (child process).
- So after fork the virtual memory contents of parent and child process are identical. That is both parent and child processes will have identical Text, Data, BSS, Heap and Stack sections.
- Next kernel copies most of the content of parent process PCB into child process PCB. Only few fields in the child process PCB will be different from the parent process PCB.

fork() .....

–Now parent and child processes are independent processes and both are going to execute same program after fork() statement onwards. While entering into fork system call, only parent process is present, but fork() returns twice. Once in parent process context and next in child process context.

–The return for fork will be different in these two contexts. For the parent the return value of fork would be PID of the child. For child process, the return value of fork is zero.

–The child process ID also shall not match any active process group ID.

–The child process shall have a different parent process ID, which is nothing but the process ID of the calling process.

–The child process shall have its own copy of the parent's file descriptors.

fork() .....

- Each of the child's file descriptors shall refer to the same open file description with the corresponding file descriptor of the parent.
- After fork(), both the parent and the child processes shall be capable of executing independently before either one terminates.

- Both processes shall continue to execute from the fork() function.
- Upon failure, -1 shall be returned to the parent process, no child process shall be created, and errno shall be set to indicate the error.

vfork()

- create a new process; share virtual memory.
- *#include <unistd.h>*
  pid_t vfork(void);
- The *vfork()* function differs from fork() only in that the child process can share code and data with the calling process (parent process).
- Upon successful completion, *vfork()* shall return 0 to the child process and return the process ID of the child process to the parent process.
- Otherwise, -1 shall be returned to the parent, no child process shall be created, and errno shall be set to indicate the error.

wait()

- wait for a child process to stop or terminate.

  #include <sys/wait.h>

  *pid_t wait(int *stat_loc);*

- The wait() function shall suspend execution of the calling thread until status information for one of the terminated child processes of the calling process is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

wait() .....

- The wait() function shall obtain status information pertaining to one of the caller's child processes.
- If *wait()* returns because the status of a child process is available, it return a value equal to the process ID of the child process for which status is reported.

waitpid()

- wait for a child process to stop or terminate
- *#include <sys/wait.h>*

  *pid_t waitpid(pid_t pid, int *stat_loc, int options);*
- *waitpid()* functions shall obtain status information pertaining to one of the caller's child processes.
- The waitpid() function shall be equivalent to wait() if the pid argument is (pid_t)-1 and the options argument is 0.
- If waitpid() returns because the status of a child process is available, it shall return a value equal to the process ID of the child process for which status is reported.

Exec family of calls

- Loading a binary program into address space by replacing the previous contents of the address space, and begins execution of the new program.
- The most frequent use of execve() is in the child produced by a fork(), although
- it is also occasionally used in applications without a preceding fork().
- There is no single exec function, instead these is a famil of exec functions built on a single system call exec().
- Following are the different exec system calls.
  - Int execl(const char *pathname, const char *arg0,...);
  - int execv(const char *pathname, char *const argv[]);
  - Int execlp(const char *filename, const char *arg0,...);
  - Int execvp(const char *filename, char *const argv[]);
- The executable file is loaded into the virtual memory of the current process and starts executing from the beginning. Process id remains the same.
- Once exec() is successful, no instructions after exec() statements will be executed. Because this program is over written with a new program

execl

- A call to execl() replaces the current process image with a new one by loading into memory the program pointed at by the path.
    - int execl(const char *path, const char *arg,...);

    ellipse signifies the variable number of arguments which mean that additional arguments may optionally follow one by one.

    The list of arguments should be NULL terminated.h a new program
- Normally execl() does not return. A successful invocation ends by jumping to the entry point of the new program, and the just executed code no longer exists in the process address space.
- A successful execl() call changes not only the address space and process image, but certain other attributes of the process.
    - Any pending signals are lost.
    - Newly created signal handlers are lost.
    - Memory locks are dropped.
    - atexit() behaviour is dropped.
- PID, PPID, priority, owner, group file desc table remains the same.

execv()

– execv system call is similar to execl(), only difference is in the passing of command line argument strings.
– The execl takes each command argument string as a separate parameter, whereas execv takes command argument string array, as single parameter.
– Last command string in the array should be NULL pointer.

```
#include <unistd.h>
int execv(const char *path, char *const argv[]);
```

execlp() & execvp()

- –execlp is identical to execl, but for execl we must mention full path name of the executable file. Where as for execlp only execuable file name is sufficent.
- – Execvp is identical to execv, and only the executable file name is sufficient.
- –Real usage:
  - So far we have been running execxx function in the same process. But one very common and useful way to run execxx() is in child process. A parent process, whenever it wants to run another program, it creates a child process and in the child process it execs the new executable file. The following program show this usage of execvp() system call.

Exec family calls

- Functions that contain the letter p in their names (execvp and execlp) accept a program name and search for a program by that name in the current execution path; functions that don’t contain the p must be given the full path of the program to be executed.
- Functions that contain the letter v in their names (execv, execvp, and execve) accept the argument list for the new program as a NULL-terminated array of pointers to strings. Functions that contain the letter l (execl, execlp, and execle) accept the argument list using the C language’s varargs mechanism.
- Functions that contain the letter e in their names (execve and execle) accept an additional argument, an array of environment variables.The argument should be a NULL-terminated array of pointers to character strings. Each character string should be of the form “VARIABLE=value”.

VIVEN Embedded Academy

Initial Process Relationships

Agenda

- Terminal Logins

Signals

VIVEN

Embedded Academy

Agenda

- Signal concepts
- signal()
- kill()
- raise()
- alarm()
- pause()

Signal Generation and Delivery

- Signal is an event generated by Unix system in response to some condition, upon receipt of which a process may in turn take some action.
- Signals are software interrupts.
- Kernel sends the signals to processes because of following reasons.
  - When a process performs some illegal operation.
    - Like dividing by zero
    - Accessing a memory location outside process address space.
    - Accessing an integer at unaligned address.
    - Executing an invalid instruction.
    - Executing a privileged instruction.
  - Whenever process performs these illegal operations, CPU will generate an exception or trap. This causes CPU to execute an exception handler function of kernel. In this function kernel identify the process that caused exception and sends a corresponding signal to that process.

Signal Generation and Delivery

- Once a signal is generated and delivered to a process, which than takes some action in response to the signal. Between the time it is generated and the time it is delivered, a signal is said to be **pending**.
- Sometimes, however we need to ensure that a segment of code is not interrupted by the delivery of signal. To do this, we can add a signal to the process's signal mask- a set of signals whose delivery is currently blocked , it remains pending until it is later unblocked(removed from the signal mask). Various ssystem calls allows a process to add and remove signals from its signal mask.

Signal Generation and Delivery

- When user enters some special key at the terminal or when terminal disconnects.
  - Special keys like Control-C or Control-Z, the terminal driver(kernel) detects these special keys and generates signals to the process.
- When another process requests the kernel to send a signal.
  - A process can request the kernel to send a signal to other process(if it got permission) by using kill function. Then kernel will send the specified signal to the specified process.
- When some events which are important to a process occurs.
  - Kernel also sends singal to a process on some software conditions like, child process terminating. Expiration of alarm, writing to a pipe when reader has terminated etc..

How process handles signals

- Every process control block(PCB) maintains information on how to handle each of the signals. This called disposition of signals. So each process can have its own signal disposition table. The signal disposition can specify one of the following three things.
  - **Ignore** the signal: This works for most signals, but there are two signals that can never be ignored. These are SIGKILL and SIGSTOP. The reason these two signals cant be ignored is to provide the super user with a sure way of either killing or stopping a process.
  - Execute a **signal handler**: For this we need to give the address of a signal handler function. When a signal occurs, process will execute this handler function. Invocation of a signal handler may interrupt the main program flow at any time; the kernel calls the handler on the process's behalf, and when the handler returns, execution of the program resumes at the point where the handler interrupted it.
  - Take **default action**: Every signal has default action. Default action for most of the action is to terminate the process. For some signals default action could be ignoring a singal.

How process handles signals

- The process is **terminated**(killed). This is sometimes refered to as abnormal process termination, as opposed to normal process termination that occurs when the process terminates using exit().
- A **core dump file** is generated, and the process is terminated. A core dump file contains an image of the virtual memory of the process which can be loaded into a debugger in order to inspect the state of the process at the time it was terminated.
- The process is **stopped** – execution of the process is suspended.
- Execution of the process is **resumed** after previously stopped.

- It isnt possible to set the disposition of a signal to **terminate** or **dump core (**unless one of these is the default disposition of the signal). The nearest we can get to this is to install a handler for the signal that then calls either exit() or abort(). The abort() function generates SIGABRT signal for the process which causes it to dump core and terminate.

Signal Utilities

- How to get Core Dump.
  - –Use abort() function call.
  - –Use special key “Ctrl \”. SIGQUIT is generated by the terminal driver also called as terminal quit character.
- How to get signals info for each process
  - –Ps -s show pending, blocked, ignored and caught signals.
  - –Kill -l lists the types of signals and their ids.
  - –Also use “man 7 signal”

VIVEN Embedded Academy

Signal Actions

–There are three types of action that can be associated with a signal: SIG_DFL, SIG_IGN, or a pointer to a function.
- •SIG_DFL : Signal-specific default action.
- •SIG_IGN : Ignore signal.

–By default all signal dispositions of a process are set to take default action.

–There are 31 different types of signals. These 31 signals are predefined, each signal is having a name indicating its purpose and each signal is also associated with a number. Every signal name stars with three letters SIG. Following are the names fo some important signals.

Signal Description

–SIGABRT-This signal is generated by calling the abort() funciton. The process terminates abnormally

–SIGALRM: This signal is generated when a timer that we've set with the alarm() function expires.

–SIGCHLD: Whenever a process terminates or stops, the SIGCHLD signal is sent to the parent.

–SIGFPE:This signals an arithmetic expression, such as divide-by-0, floating point overflow, and so on.

–SIGINT: This signal is generated by the terminal driver when we type the interrupt key(Control-C).

–SIGKILL: This signal is one of the two that cant be caught.

Signal Description

- –SIGPIPE: If we write to a pipe but the reader has terminated, this singal is generated.
- –SIGTERM: This is the termination signal sent by the kill(1) command by default.
- –SIGUSR1: This is a user-defined signal, for use in application programs.
- –SIGUSR2: This is a user-defined signal, for use in application programs.

Changing Signal Disposition

•UNIX systems provide two ways of changing the disposition of a signal: signal() and sigaction(). The signal() systems call,was the original API for setting the disposition of a signal, and it provides a simpler interface than sigaction(). Sigaction() provides functionality that is not avialable with signal().

•There are variations in the behaviour of signal() across UNIX implementations which mean that it could never be used for establishing signal handlers in portable programs. Because of these variations sigaction() is strongly preferred API for establishing a signal handler.

signal()

–signal management.

#include <signal.h>

- *void (*signal(int sig, void (*func)(int)))(int);*
- *signal()* function chooses one of three ways in which receipt of the signal number sig is to be subsequently handled.
- If the value of func is SIG_DFL, default handling for that signal shall occur.
- If the value of func is SIG_IGN, if the signal is generated the kernel silently discards it. The process never ever knows that the signal has occured.
- The return value of signal() is previous disposition of the signal.

signal() ......

– Otherwise, the application shall ensure that func points to a function to be called when that signal occurs (Signal Handler).
– If the request can be honoured, *signal()* shall return the value of func for the most recent call to *signal()* for the specified signal sig.
– Otherwise, SIG_ERR shall be returned and a positive value shall be stored in errno.

Signal Categories

–Signals fall into two broad categories.

–The first set constitutes the **traditional** or **standard signals**, which are used by the kernel to notify processes of events. On linux the **standard signals** are numbered from 1 to 31.

–The other set of signals consists of **realtime signals,** numbered from 32 to 63. Realtime signals provides an increased range of signals that can be used for application defined pupose. Only two standard signals are freely available for application-defined purposes: SIGUSR1 and SIGUSR2.

Important Signals

Signal	**Default Action**	**Description**
SIGABRT (6)	A	Process abort signal.
SIGALRM (14)	T	Alarm clock.
SIGCHLD (20)	I	Child process terminated, stopped,
SIGCONT (19)	C	Continue executing, if stopped.
SIGHUP (1)	T	Hangup.
SIGINT (2)	T	Terminal interrupt signal.
SIGKILL (9)	T	Kill (cannot be caught or ignored).
SIGPIPE (13)	T	Write on a pipe with no one to read it.
SIGQUIT (3)	A	Terminal quit signal.
SIGSEGV (11)	A	Invalid memory reference.
SIGSTOP (17)	S	Stop executing (cannot be caught).
SIGTERM (15)	T	Termination signal.

Important Signals ....

The default actions are as follows:

T: Abnormal termination of the process.

A: Abnormal termination of the process.

I: Ignore the signal.

S: Stop the process.

C: Continue the process, if it is stopped; otherwise, ignore the signal.

Signals Sets ....

- Multiple signals are represented using a data structure called a **signal set** provide by the system type sigset_t..
- The sigemptyset() function initializes a signal set to contain no members.
- The sigfillset() function initializes a set to contain all signals(including real-time signals).
  - int sigemptyset(sigset_t *set);
  - int sigfillset(sigset_t *set);
- After initializing, individual signals can be added to a set using sigaddset() and removed using sigdelset().
  - int sigaddset(sigset_t *set, int sig);
  - int sigdelset(sigset_t *set, int sig);
- The sigismember() function is used to test for membership of a set.
  - int sigismember(const sigset_t *set, int sig);

The Signal Mask(Blocking Signal Delivery)

For each process the kernel maintains a signal mask- a set of signals whose delivery to the process is currently blocked. If a signal that is blocked is sent to a process delivery of that signal is delayed until it is unblocked by being removed from the process signal mask.

A signal may be added to the signal mask in the following ways.

- When a signal handler is invoked, the signal that caused its invocation can be automatically added to the signal mask. Whether or not this occurs depends on the flags(SANODEFER- **sigaction**()).
- When a signal handler is established with sigaction(), it is possible to specify an additional set of signals that are to be blocked when the handler is invoked.
- The **sigprocmask** system call can be used at anytime to explicitly add signals and remove signals from, the signal mask.

Sigprocmask()

*int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);*

Sigprocmask() is used to change the process signal mask, to retrieve the existing mask, or both. The how argument determines the changes the sigprocmask() makes to signal mask:
SIG_BLOCK: The signals specied in the signal set pointed to by set are added to the signal mask.
SIG_UNBLOCK: The signals in the signal set pointed to by set are removed from the signal masks.
SIG_SETMASK: The signal set pointed to by set is assigned to the signal mask.
In each case, if the oldset argument is not NULL, it points to sigset_t buffer that is used to return the previous signal mask.

Pending Signals

If a process receives a signal that is currently blocking, that signal is added to the process set of pending signals. When the signals is later unblocked, it is then delivered to the process To determine which signals are pending for a process, we can call sigpending().

– int sigpending(sigset_t *set);

Sigaction()

int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);

Sigaction is used for setting the disposition of a signal. Sigaction is more complex to use than signal(), in return it provides greater flexibility. In particular sigaction() allows us to retrieve the disposition of a signal without changing it, and to set various attributes controlling precisely what happens when a signal handler is invoked.
The structures pointed to by act and oldact are of the follow-
ing type:

```
struct sigaction {
      void (*sa_handler)(int);   /*Address of handler*/
      sigset_t sa_mask;          /*Signals blocked during handler*/
      int void sa_flags;         /*Flags controlling the handler incocation
      (*sa_restorer)(void);         /*Not for application use*/
};
```

kill()

–send a signal to a process or a group of processes.

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

–The *kill()* function shall send a signal to a process or a group of processes specified by pid. The signal to be sent is specified by sig.

–For a process to have permission to send a signal to a process designated by pid, unless the sending process has appropriate privileges.

kill() ....

- If pid is greater than 0, sig shall be sent to the process whose process ID is equal to pid.
- If pid is 0, sig shall be sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.
- If pid is less than -1, the signal is sent to all of the processes in the process group whose ID equals the absolute value of pid.
- Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and errno set to indicate the error.

raise()

–Simple way for a process to send a signal to itself.

```
#include <signal.h>
int raise(int sig);
```

–*Internally the raise() calls kill(getpid(),sig);*

–Upon successful completion, 0 shall be returned.

–Otherwise, a non-zero value shall be returned and errno shall be set to indicate the error.

alarm()

- schedule an alarm signal.
- *#include <unistd.h>*
  unsigned alarm(unsigned seconds);
- The *alarm()* function shall cause the system to generate a SIGALRM signal for the process after the number of real-time seconds specified by seconds have elapsed.
- If there is a previous *alarm()* request with time remaining, alarm() shall return a non-zero value that is the number of seconds until the previous request would have generated a SIGALRM signal.
- Otherwise, a non-zero value shall be returned and errno shall be set to indicate the error.

pause()

- suspend the thread until a signal is received.
- *#include <unistd.h>*
  int pause(void);
- The *pause()* function shall suspend the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.
- There is no successful completion return value.
- A value of -1 shall be returned and errno set to indicate the error.

SYSTEM AND PROCESS INFORMATION

VIVEN
Embedded Academy

The /proc File System

- We will look at various ways accessing a variety of ***system*** and ***process*** information. The primary focus is on ***/proc*** file system and ***uname()*** system call which is used to retrieve various system identifiers.
- In older UNIX implementations, there was typically no easy way to analyze (or change) attributes of the kernel, to answer questions such as the following.
  – How many processes are running on the system and who owns them?
  – What files does a process have open?
  – What files are currently locked, and which processes hold the locks?
  – What sockets are being used on the system?
- In order to provide easier access to kernel information, many modern UNIX implmentations provide **/proc virtual file system**. This file system resides under the /proc directory and contains various files that expose kernel information, allowing processes to conviniently read that information, and change it in some cases, using normal file I/O system calls. The **/proc file sytem** is said to be virtual because the files and subdirectories that it contains **don't reside on disk**. Instead, the kernel creates them “**on the fly**” as processes access them.

Obtaining Information about a process:

- For each process on the system, the kernel provides a corresponding directory named ***/proc/PID*,** where ***PID*** is the ID of the process. Within the directory are various ***files*** and ***subdirectories*** containing information about that process. For example we can obtain information about the ***init*** process, which always has the ***process ID*** 1, by looking at a file under the directory **/proc/1**. Among the files in each **/proc/PID** directory is one named status, which provides a range of information about the process;

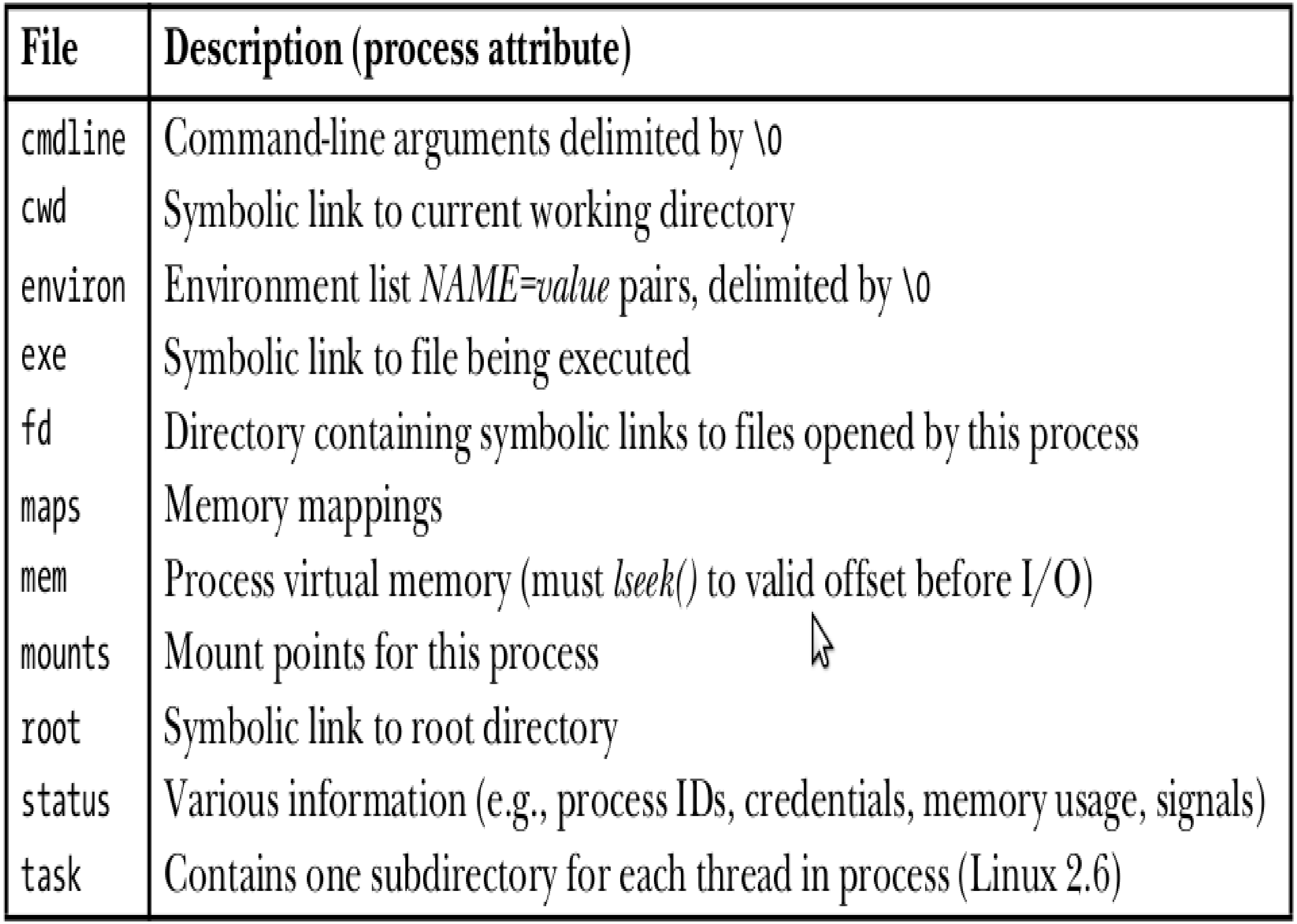
  $cat /proc/1/status

- **The /proc/PID/fd directory:** The ***/proc/PID/fd*** directory contains one symbolic link for each ***file descriptor*** that the process has opened. Each of the ***symbolic link*** has a name that matches the descriptor number; for example ***/proc/1968/1*** is a ***symbolic link*** to the ***standard output*** of process 1968. Any process can acces its own ***/proc/PID*** directory using the symbolic link ***/proc/self.***
- ***Threads: the /proc/PID/task directory:*** For each thread in a process, the kernel provides a subdirectory named **/proc/PID/task/TID**, where TID is the thread id of the thread(returned by gettid()). Under each directory is a set of files and directories exactly like those in **/proc/PID**. Much of the info in the files are same except for thread group, state, pid, Sigpnd, SigBlk, Capink, CapPrm, CapBnd.

```
$ cat /proc/1/status
Name:   init                                Name of command run by this process
State:  S (sleeping)                        State of this process
Tgid:   1                                   Thread group ID (traditional PID, getpid())
Pid:    1                                   Actually, thread ID (gettid())
PPid:   0                                   Parent process ID
TracerPid:      0                           PID of tracing process (0 if not traced)
Uid:    0       0       0       0           Real, effective, saved set, and FS UIDs
Gid:    0       0       0       0           Real, effective, saved set, and FS GIDs
FDSize: 256                                 # of file descriptor slots currently allocated
Groups:                                     Supplementary group IDs
VmPeak:      852 kB                         Peak virtual memory size
VmSize:      724 kB                         Current virtual memory size
VmLck:         0 kB                         Locked memory
VmHWM:       288 kB                         Peak resident set size
VmRSS:       288 kB                         Current resident set size
VmData:      148 kB                         Data segment size
VmStk:        88 kB                         Stack size
VmExe:       484 kB                         Text (executable code) size
VmLib:         0 kB                         Shared library code size
VmPTE:        12 kB                         Size of page table (since 2.6.10)
Threads:        1                           # of threads in this thread's thread group
SigQ:   0/3067                              Current/max. queued signals (since 2.6.12)
SigPnd: 0000000000000000                    Signals pending for thread
ShdPnd: 0000000000000000                    Signals pending for process (since 2.6)
SigBlk: 0000000000000000                    Blocked signals
SigIgn: fffffffe5770d8fc                    Ignored signals
SigCgt: 00000000280b2603                    Caught signals
CapInh: 0000000000000000                    Inheritable capabilities
CapPrm: 00000000ffffffff                    Permitted capabilities

CapEff: 00000000fffffeff                    Effective capabilities
CapBnd: 00000000ffffffff                    Capability bounding set (since 2.6.26)
Cpus_allowed:   1                           CPUs allowed, mask (since 2.6.24)
Cpus_allowed_list:      0                   Same as above, list format (since 2.6.26)
Mems_allowed:   1                           Memory nodes allowed, mask (since 2.6.24)
Mems_allowed_list:      0                   Same as above, list format (since 2.6.26)
voluntary_ctxt_switches:        6998        Voluntary context switches (since 2.6.23)
nonvoluntary_ctxt_switches:     107         Involuntary context switches (since 2.6.23)
Stack usage:    8 kB                        Stack usage high-water mark (since 2.6.32)
```

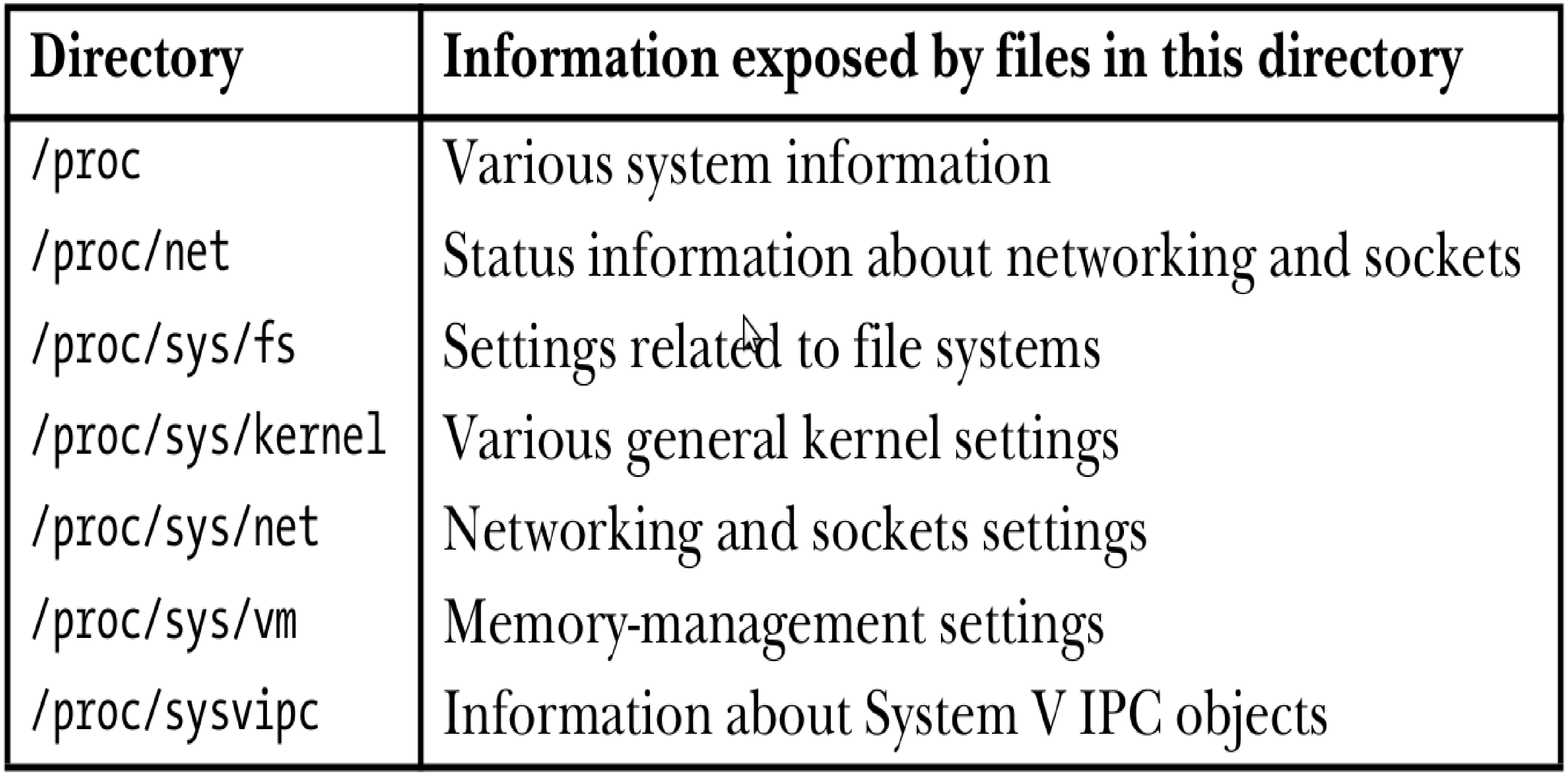
Selected files in each /proc/PID directory



File	Description (process attribute)
`cmdline`	Command-line arguments delimited by `\0`
`cwd`	Symbolic link to current working directory
`environ`	Environment list *NAME=value* pairs, delimited by `\0`
`exe`	Symbolic link to file being executed
`fd`	Directory containing symbolic links to files opened by this process
`maps`	Memory mappings
`mem`	Process virtual memory (must *lseek()* to valid offset before I/O)
`mounts`	Mount points for this process
`root`	Symbolic link to root directory
`status`	Various information (e.g., process IDs, credentials, memory usage, signals)
`task`	Contains one subdirectory for each thread in process (Linux 2.6)

System Information under /proc

- Various files and subdirectories under /proc provide access to system-wide information. A few of these are shown below.



Directory	Information exposed by files in this directory
/proc	Various system information
/proc/net	Status information about networking and sockets
/proc/sys/fs	Settings related to file systems
/proc/sys/kernel	Various general kernel settings
/proc/sys/net	Networking and sockets settings
/proc/sys/vm	Memory-management settings
/proc/sysvipc	Information about System V IPC objects

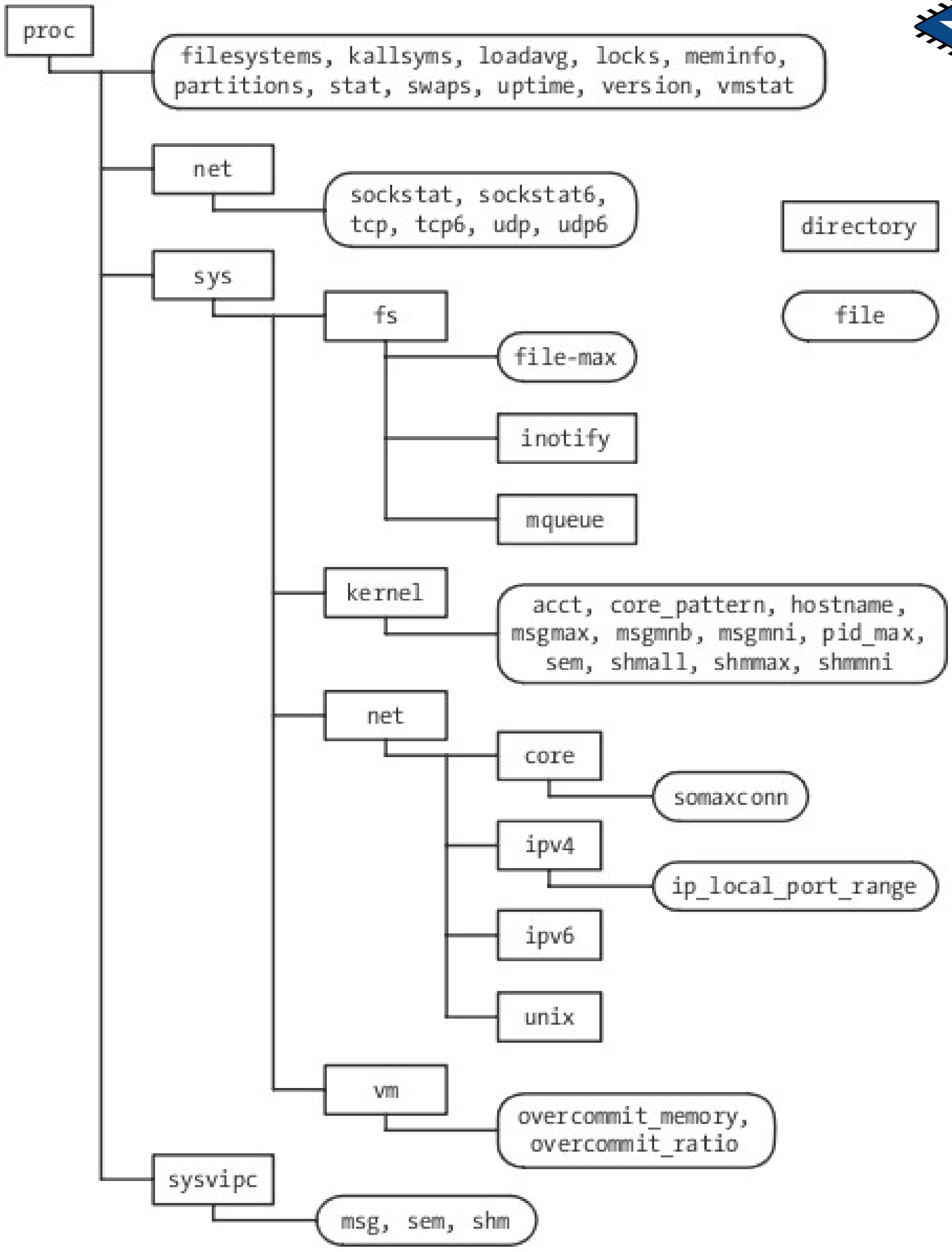
Accessing /proc files

- Files under /proc are often accessed using shell scripts(most /proc files that contain multiple values can be easily parsed with a scripting language such as Phython or Perl).
  – Ex: We can modify and view the contents of a /proc file using shell commands as follows.

```
#echo 100000 > /proc/sys/kernel/pid_max
#cat /proc/sys/kernel/pid_max
100000
```

/proc files can also be accessed from a program using normal file I/O System calls. Some restrictions apply when accessing these files.

- Some /proc files are read-only; that is, they exist only to display kernel information and can't be used to modify that information. This applied to most files under /proc/PID directories
- Some /proc files can be read only by the file owner(or by a privileged process). All files under /proc/PID are owned by the user who owns the corresponding process.Ex: /proc/PID/environ(read permission is granted only to the file owner).
- Other than files in the /proc/PID subdirectories, most files under /proc are owned by root, and the files that are modifiable can be done only by root.



proc
filesystems, kallsyms, loadavg, locks, meminfo, partitions, stat, swaps, uptime, version, vmstat
net
sockstat, sockstat6, tcp, tcp6, udp, udp6
sys
fs
file-max
inotify
mqueue
kernel
acct, core_pattern, hostname, msgmax, msgmnb, msgmni, pid_max, sem, shmall, shmmax, shmmni
net
core
somaxconn
ipv4
ip_local_port_range
ipv6
unix
vm
overcommit_memory, overcommit_ratio
sysvipc
msg, sem, shm
directory
file

System Identification: uname()

- The ***uname()*** system call returns a range of identifying information about the host system on which an application is running, in the structure pointed by ***utsbuf***.

    int uname(struct utsname *utsbuf);

- The ***utsbuf*** argument is a pointer to a ***utsname*** structure, which is defined next. The ***sysname***, ***release***, ***version***, and ***machine*** fields of the **utsname** structure are automatically set by the kernel.
- The ***nodename*** field returns the value that was set using the ***sethostname()*** system call. Often, this name is something like the hostname prefix from the system's DNS domain name.
- The ***domainname*** field returns the value that was set using the ***setdomainname()*** system call. This is the Network Information Services (NIS) domain name of the host (which is not the same thing as the host's DNS domain name).

System Identification: uname()

```
#define _UTSNAME_LENGTH 65

struct utsname {
    char sysname[_UTSNAME_LENGTH];      /* Implementation name */
    char nodename[_UTSNAME_LENGTH];     /* Node name on network */
    char release[_UTSNAME_LENGTH];      /* Implementation release level */
    char version[_UTSNAME_LENGTH];      /* Release version level */
    char machine[_UTSNAME_LENGTH];      /* Hardware on which system
                                           is running */
#ifdef _GNU_SOURCE                      /* Following is Linux-specific */
    char domainname[_UTSNAME_LENGTH];   /* NIS domain name of host */
#endif
};
```

Inter Process Communication

VIVEN

Embedded Academy

Agenda

- Pipes
- FIFO (Named Pipes)
- Message Queues
- Semaphores
- Shared Memory

Inter Process Communication

- Processes communicate with each other and with the kernel to coordinate their activities.
- Linux supports a number of IPC mechanisms.
- Signals and pipes are two of them but Linux also supports other IPC mechanisms such as
    - FIFOs (Named Pipes)
    - Message Queues
    - Semaphores
    - Shared Memory

Pipes

- A pipe is a method of connecting the standard output of one process to the standard input of another.
- They provide a method of one-way communications between processes.(Full-duplex pipes are also there).
- When a process creates a pipe, the kernel sets up two file descriptors for use by the pipe.
- One descriptor is used to allow a path of input into the pipe (write), the other to obtain data from the pipe(read).
- pipes are actually represented internally with a valid inode.

Creating Pipes in C

- To create a simple pipe , we make use of the pipe() system call.
- It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline.
- After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors).
- When one end of the pipe has been closed the following two rules apply.
    - If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read. Normally there will be a single reader and single writer for a pipe.
    - If we write to a pipe whose read end has been close, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns -1.

Creating Pipes in C ....

```
#include <unistd.h>
int pipe( int fd[2] );
```

–Returns 0 on success and -1 on error:

–fd[0] is set up for reading, fd[1] is set up for writing.

–The first integer in the array (fd[0]) is set up and opened for reading, while the second integer (fd[1]) is set up and opened for writing.

–Visually speaking, the output of fd1 becomes the input for fd0.

Creating Pipes in C ....

```
Parent process -----------> PIPE ---------------> child
                        fd[1]                 fd[0]


popen calle  ---------> PIPE --------------> popen caller
                  fd[1]                fd[0]
```

Creating Pipes in C ....

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
        int     fd[2], nbytes;
         pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
                    ........
```

Creating Pipes in C …….

```
         if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
        close(fd[0]);

                /* Send "string" through the output side of pipe */
        write(fd[1], string, (strlen(string)+1));
        exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
        close(fd[1]);

                /* Read in a string from the pipe */
        nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
        printf("Received string: %s", readbuffer);
        }
        return(0);
}
```

FIFOs (Named Pipes)

- A named pipe works much like a regular pipe.
- Named pipes exist as a device special file in the file system.
- Processes of different ancestry can share data through a named pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

FIFOs (Named Pipes) ......

- There are several ways of creating a named pipe.
- mkfifo() - make a FIFO special file (a named pipe).
- mknod() - create a special or ordinary file.
- #include <sys/types.h>
  #include <sys/stat.h>

*int mkfifo(const char *filename, mode_t mode);*

*int mknod(const char *filename, mode_t mode|S_IFIFO, (dev_t) 0);*

FIFOs (Named Pipes) ......

–*Once we have used mkfifo to create FIFO we open it using open. Normal file IO functions work with FIFO's.*

–*When we open a FIFO, the non blocking O_NONBLOCK affects what happened.*

- *If not specified: An open for read only blocks until some other process opens the FIFO for writing.*
- *Similarly an open for write only block until some other process opens the FIFO for reading.*
- *If specified: An open for read only returns immediately. But an open for write-only returns 1 if no process has the FIFO open for reading.*

Creating FIFOs

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
        int main()
        {
                int res = mkfifo(“/tmp/my_fifo”, 0777);
                if (res == 0)
             printf(“FIFO created\n”);
                exit(EXIT_SUCCESS);
        }
```

Message Queues

- In many ways, message queues are like named pipes, but without the complexity associated with opening and closing the pipe.

-Message queues provide a reasonably easy and efficient way of passing data between two unrelated processes.

- Message queues provide a way of sending a block of data from one process to another.

-There’s a maximum size limit imposed on each block of data and also a limit on the maximum total size of all blocks on all queues throughout the system.

msgid_ds structure

```
struct msqid_ds {
        struct ipc_perm msg_perm;
        msgqnum_t msg_qnum: //Num of messages on queue
        msglen_t msg_qbytes; //max num of bytes on queue
        pid_t msg_lspid;          //pid of last msgsnd()
        pid_t msg_lrpid;          //pid of last msgrcv()
        time_t msg_stime;        //last-msgsnd() time
        time_t msg_rtime;        //last-msgrcv() time
        time_t msg_ctime;       //last-change time
        .
        .
        .
        }
```

This structure defines the current status of the queue.

System limits that affects messages queues

- Size in bytes of largest message we can send 8,192 Bytes
- The maximum size in bytes of a particular queue(i.e., the sum of all the messages on the queue – 819200
- The maximum number of messages queues, systemwide – 100.
- ipc_perm(permission structure) is associated with each message queue. It includes owner user id, group id, creator's user id and group id and access mode.
- When a new queue is created, the following members of the msqid_ds structure are initialized.
  - The ipc_perm structure is initialized.
  - msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime are all set to 0.
  - msg_ctime is set to the current time.
  - msg_qbytes is set to the system limit.

Message Queues

-The message queue function definitions are

```
        #include <sys/types.h>
        #include <sys/ipc.h >
        #include <sys/msg.h>
    int msgctl(int msqid, int cmd, struct msqid_ds *buf);
    int msgget(key_t key, int msgflg);
    int msgrcv(int msqid, void *msg_ptr, size_t msg_sz,
        long int msgtype, int msgflg);
    int msgsnd(int msqid, const void *msg_ptr, size_t
            msg_sz, int msgflg);
```

Message Queues

int msgget(ket_t key, int msgflag)

- need to provide a key value with which name is assigned to msg queue.
- msgflag == IPC_PRIVATE -> private queue to that process
- we can pass IPC_CREAT with permission bits
  eg: msgflag = 666 | IPC_CREAT

msgrecv:

The type argument lets us specify which message we want.

Type == 0 :The first message on the queue is returned.

Type > 0: The first message on the queue whose message type equals type is returned.

Type < 0: The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

Message Queues

msgctl: The msgctl function performs various operations on the queue.

#include <sys/msgs.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);

The cmd argument specifies the command to be performed on the queue.

command	Description
IPC_STAT	sets the data in msgqid_ds strcuture to reflect values associated with the message queue
IPC_SET	If the process has permissions to do so, this sets the values associated with the msg queue to those provided in the msqid_ds structure
IPC_RMID	Deletes the msg queue

SHARED MEMORY

VIVEN
Embedded Academy

Shared memory

- Shared memory allows two unrelated processes to access the same logical memory.
- Shared memory is a very efficient way of transferring data between two running processes.
- Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process.
- All processes can access the memory locations just as if the memory had been allocated by *malloc()*.

Shared memory

-The functions for shared memory resemble those for semaphores:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include <sys/shm.h>
```

*void *shmat(int shm_id, const void *shm_addr, int shmflg);*

*int shmctl(int shm_id, int cmd, struct shmid_ds *buf);*

*int shmdt(const void *shm_addr);*

int shmget(key_t key, size_t size, int shmflg);

Debugging

The ipcs command provides information on interprocess communication facilities, including shared segments. Use the -m flag to obtain information about shared memory. For example, this code illustrates that one shared memory segment, numbered 1627649, is in use:

- ***ipcs -m***

If this memory segment is left behind by the program, you can use the ***ipcrm*** command to remove it.

- ***ipcrm shm 1627649***

SYSTEM V SEMAPHORES

VIVEN

Embedded Academy

Semaphores

•System V semaphore isn't a form of IPC similar to the others that we've described(pipes, FIFOs, and message queues). System V semaphors are not used to transfer data between processes. Instead they allow processes to synchronize their actions.

•One common use of semaphore is to synchronize access to a block of shared memory, in order to prevent one process from accessing the shared memory at the same time as another process is updating it.

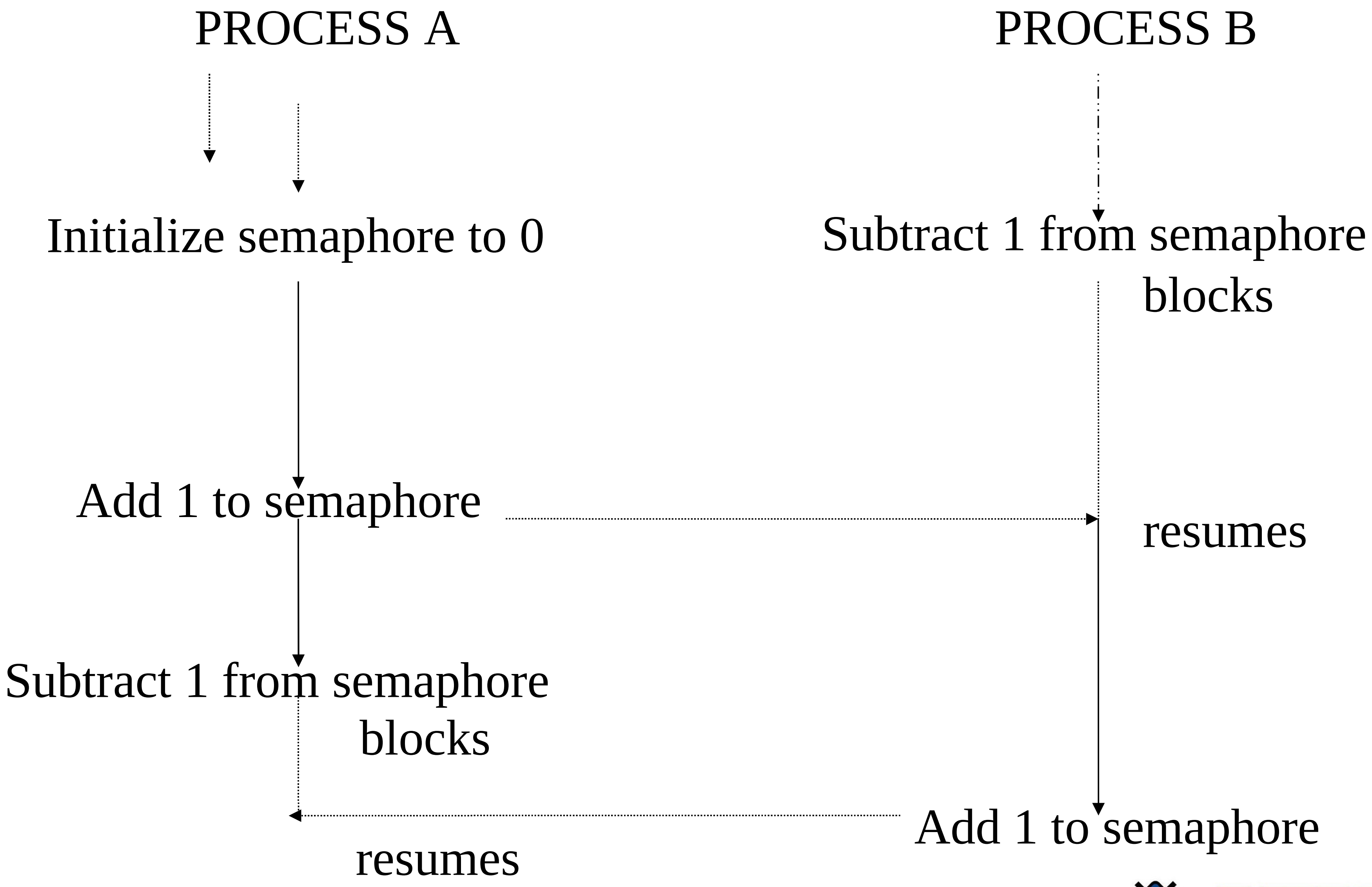
•The simplest semaphore is a variable that can take only the values 0 and 1, a **binary semaphore**.

•Semaphores that can take many positive values are called **general semaphores**.

Semaphores

- A semaphore is a kernel-maintained counter(integer) whose value is restricted to being greater than or equal to 0. Various operations can be performed on a semaphore, including the following.
  - Setting a semaphore to an absolute value
  - Adding a number to the current value of the semaphore
  - Subtracting a number from the current value of the semaphore
  - Waiting for the semaphore value to be equal to 0.
- The last of two operations may cause the calling processes to block.
  - When lowering a semaphore value, the kernel blocks any attempt to decrease value below 0.
  - Similarly waiting for semaphore to equal 0 blocks the calling process if the semaphore value is not currently 0.
- In both cases, the calling process remains blocked until some other process alters the semaphore to a value that allows the operation to proceed, at which point the kernel wakes the blocked process.

Using a Semaphore to synchronize two processes



General Steps for using a System V Semaphore

- Create or open a semaphore set using **semget()**.
- Initialize the semaphores in the set using **semctl()** **SETVAL** or **SETALL** operation.(Only one process should do this).
- Perform operations on semaphore values using **semop()**. The processes using the semaphore typically use these operations to indicate acquisition and release of a shared resource.
- When all processes have finished using the semaphore set, remove the set using the semctl() IPC_RMID operation. (Only one process should do this).

System V Semaphore

- System V semaphores are rendered unusually complex as they are allocated in groups called ***semaphore sets***. The number of semaphores in set is specified when the set is created using the **semget()**.
- While it is common to operate on a single semaphore at a time, the **semop()** system call allows us to atomically perform a group of operations on multiple semaphores in the same set.
- Each semaphore set has an associated semid_ds data structure of the following form.

```
    struct semid_ds{
                struct ipc_perm sem_perm;
                unsigned short sem_nsems: /*no of semaphores in set*/
                time_t sem_otime; /*last semop() time*/
                time_t sem_ctime; /*last-change time*/........}
```

Creating or opening a Semaphores Set

•The ***semget()*** system call creates a new semaphore set or obtains the identifier of an existing set.

#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);

•The ***key*** argument is a key generated using one of the methods(usually the value IPC_PRIVATE or a key returned by ftok()).

•If we are using ***semget()*** to create a new semaphore set, then ***nsems*** specifies the number of semaphores in the set, and must be greater than 0. if we are using the ***semget()*** to obtain the identifer of an existing set, then ***nsems*** must be less than or equal to the size of the set(or the error EINVAL results). It is not possible to change the number of semaphores in an existing set.

•The semflag argument is a bit mask specifying the permissions to be placed on a new semaphore set or checked agains an existing set.

- IPC_CREATE: If no semaphore set with the specified key exists, new one is created.
- IPC_EXCL: A semaphore set with the specified key already exists, fail with the error EEXIST.

Semaphore control operations

•The ***semctl()*** system call performs a variety of control operations on a semaphore set or on a individual semaphore within a set.

#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, .../*union semun arg*/);

•The **semid** argument is the identifier of the semaphore set on which the operation is to be performed.

•For those operations performed on a single semaphore, the **semnum** argument identifies a particular semaphore within a set. For other operations, this argument is ignored, and we can specify 0.

•The **cmd** argument specifies the operation to be performed.

•Certain operations require a forth argument to **semctl()**, which we refer to by the name **arg**. This argument is optional.

```
•Union semun{ int val;
              struct semid_ds *buf;
              unsigned short * array;
              #if defined(__linux__)
                  struct seminfo * _buf;
              #endif};
```

Generic control operations

- Below are the list of **cmds** used in **semctl().** In each of the below case, the semnum argument is ignored.
  - IPC_RMID:Immediately remove the semaphore set and its associated ***semid_ds*** data structure. Any process blocked in ***semop()*** calls waitinng on semaphores in this set are immediately awakened, with ***semop()*** reporting the error ***EIDRM***. The ***arg*** argument is not required.
  - IPC_STAT: Place a copy of the ***semid_ds*** data structure associated with this semaphore set in the buffer pointed to by ***arg.buf***.
  - IPC_SET: Update the selected fields of the ***semid_ds*** data structure associated with this semaphore set using values in the buffer pointed to by ***arg.buf***.

Retrieving and initializing semaphore values

- The following operations retrieve or initialize the value(s) of an individual semaphore or of all semaphores in a set. Retrieving a semaphore value requires read permission on the semaphore, while intializing the value requires write permissions.
  - GETVAL: ***semctl()*** returns the vlaue of the ***semnum-th*** semaphore in the semaphore set specified by ***semid***. The arg argument is not required.
  - SETVAL: The value of the ***semnum-th*** semaphore in the set referred to by ***semid*** is initialized to the value specified in ***arg.val***.
  - GETALL: Retrieve the values of all of the semaphore in the set referred to by ***semid***, placing them in the array pointed by the ***arg.array***. The programmer must ensure that this array is of sufficient size. The ***semnum*** argument is ignored.
  - SETALL: Initialize all semaphores in the set referred to by ***semid***, using the values supplied in the array pointed to by ***arg.array***. Semnum is ignored.

Retrieving per-semaphore information

- The following operations return (via the function result value) information about the ***semnum-th*** semaphore of the set referred to by semid. For all these operations, read permission is required on the semaphore set, and the arg argument is not required.
  - GETPID: Return the process ID of the last process to perform a ***semop()*** on this semaphore; this is referred to as the semid value. If no process has yet performed a ***semop()*** on this semaphore, 0 is returned.
  - GETNCNT: Return the number of processes currently waiting for the value of this semaphore to increase; this is referred to as the ***semncnt*** value.
  - Return the number of processes currently waiting for the value of this semaphore to become 0. This is referred to as the ***semzcnt*** value.

VIVEN
Embedded Academy

Semaphore Operations

- The semop() system call performs one or more operations on the semaphores in the semaphore set identified by semid.

```
    Int semop(int semid, struct sembuf *sops, unsigned int nsops);
```

- The sops argument is a pointer to an array that contains the operations to be performed, and nsops gives the size of this array(which must contain atleast one element).The elements of the sops array are structures of the following form.

```
    Struct sembuf{
            unsigned short sem_num;    /*semaphore number*/
            short sem_op;              /*Operation to be performed*/
            short sem_flg;             /*Operation flags
                };         (IPC_NOWAIT and SEM_UNDO)*/
```

- The sem_num field identifies the semaphore within the set upon which the operation is to be performed.

Semaphore Operations

The ***sem_op*** field specifies the operation to be performed.

•If ***sem_op > 0***, the value of **sem_op** is added to the semaphore value. As a result, other process waiting to decrease the semaphore value may be awakened and perform their operations. The calling process must have alter permissions on the semaphore.

•If ***sem_op == 0***, the value of the semaphore is checked to see whether it currently equals 0. If it does, the operation completes immediately; other **semop()** blocks until the semaphore value becomes 0. the calling process must have read permissions.

•If ***sem_op < 0***, decrease the value of the semaphore by the amount specified in ***sem_op***. If the current value of the semaphore is greater than or equal to the absolute value of ***sem_op***, the operation completes immediately. Otherwise, ***semop()*** blocks until the semaphore value has been increased to a level that permits the operation to be performed without resulting in a negative value. The calling process must have alter permissions.

Semaphore Operations

Semantically, increasing the value of a semaphore corresponds to making a resource avialable so that others can use it, while decreasing the value of a semaphore corresponds to reserving a resource for(exclusive) use by the process.

•When decreasing the value of a semaphore, the operating is blocked if the semaphore value is too low-that is, if some other process has already reserved the resource.

•When a ***semop()*** call blocks, the process remains blocks until one of the following occurs:

- Another process modifies the value of the semaphore such that the requested operation can proceed.
- A signal interrupts the semop() call. In this case, the error ***EINTR*** results.
- Another process deletes the semaphore referred to by ***semid***. In this case ***semop()*** fails with the error ***EIDRM***.

Debugging semaphores

•Use the command ***ipcs -s*** to display information about existing semaphore sets.

•Use the ***ipcrm sem*** command to remove a semaphore set from the command line.

•For example, to remove the semaphore set with identifier 5790517 use the below line.

Ipcrm sem 5790517

POSIX SEMAPHORES

VIVEN
Embedded Academy

Named Semaphores

To work with a named semaphore, we employ the following functions:

- The ***sem_open()*** function opens or creates a semaphore, initializes the semaphore if it is created by the call, and returns a handle for use in later calls.
- The ***sem_post(sem)*** and ***sem_wait(sem)*** functions respectively increment and dec-rement a semaphore’s value.
- The ***sem_getvalue()*** function retrieves a semaphore’s current value.
- The ***sem_close()*** function removes the calling process’s association with a semaphore that it previously opened.
- The ***sem_unlink()*** function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.

POSIX SHARED MEMORY

VIVEN

Embedded Academy

Shared Memory

- The shm_open() function creates and opens a new shared memory object or opens an existing object.

  int shm_open(const char *name, int oflag, mode_t mode);

- Shared memory objects are created as files in a special location of a standard file system. Linux uses a dedicated **tmpfs** file system mounted under the directory /dev/shm.
- Use the below command to see the list of shared memory objects created ***$ ls -l /dev/shm***
- When a new shared memory object is created, it initially has zero length. You have to use ***ftruncate()*** to set the size of the object before calling ***mmap()***. The created memory is filled with zero.
- Pass the file descriptor obtained in the previous step in a call to ***mmap()*** that specifies ***MAP_SHARED*** in the ***flags*** argument. This maps the shared memory into the process's virtual address space. Once we have mapped the object, we can close the file descriptor without affecting the mapping.

THREADS

VIVEN
Embedded Academy

Agenda

- Multi Threaded programming
- Synchronization and Mutual exclusion for Threads
- POSIX Semaphores

Multi-threaded programming

VIVEN Embedded Academy

- Like processes threads are a mechanism that permits an application to perform multiple task concurrently. A single process can contain multiple threads.
- All of these threads are independently executing the same program, and they all share the same global memory, including Text, Data, BSS and Heap segments with other threads. But each thread will have a separate stack area within the virtual address space.
- The programs with multi thread execution are called multi-threaded application.
- The threads in a process can execute concurrently. On a multi-processor machine multiple threads can execute parallel. If one thread is blocked on I/O, other threads are still eligible to execute.
- A unix process has single thread of control i.e. each process does only one thing at a time. With multiple threads of control, we can design our program to do more than one thing at a time.

Process Vs Threads

- Threads offer advantages over processes in certain applications.
  - Ex: Network server design where a parent process accepts incoming connections from clients, and then uses fork() to create a separate child to handle communication with each client. This design makes it possible to server multiple clients simultaneously, but has some limitations.
    - It is difficult to share information between processes. Since the child don't share memory(other than read-only text segment), we must use some form of inter-process communication in order to exchange information between processes.
    - Process creation with fork() is relatively expensive as this involves duplications of various process attributes such as page tables and file descriptor tables which is time consuming

Process Vs Threads

- Threads address both these problems.
  - Sharing information between threds is easy and fast using shared (Global -Data,BSS, Heap) variables. However, in order to avoid the problems that can occur when multiple threads try to update the same information, we must employ the synchronization techniques.
  - Thread creation is faster than process creation – typically ten times faster. (In linux threads are implemented using clone() system call, check the diff between clone() & fork()). Thread creation is faster because many of the attributes that must be duplicated in a child created by fork() are instead shared between threads.
- context-switch time may be lower for threads than for processes.

Attributes shared by Threads

- Besides global memory, threads also share a number of other attributes.
  - Process id, parent process ID, process group ID and session ID.
  - Controlling terminal, Open file descriptors, record locks using fcntl(), signal disposition.
  - File system realated information; umask, current working directory and root directory.
  - Interval timers(setitimer()).
  - System V semaphore undo.
  - Resource limits, CPU time consumed, resources consumed and nice value(set by setpriority() and nice())

Attributes that are distinct for each Threads

- –Thread id.
- –Signal mask.
- –Thread specific data
- –Alternate signal stack
- –The errno variable.
- –Floating point environment
- –Real-time sceduling policy and priority.
- –CPU Affinity
- –Capabilities
- –Stack.

•Pthreads stands for POSIX Threads

VIVEN Embedded Academy

Disadvantages of using threads

- When programming with threads we need to ensure that the function we call are thread safe or called in thread safe manner. Multiprocess application dont need to be concerned with this.
- A bug in one thread(modifying memory via a incorrect pointer) can damage all the threads in the process since they share the same address space and other attributes. By contast, processes are isolated from one another.
- Each thread will be competing for use of virtual address space of the host process and each thread's stack and thread-specific data consumes a part of the process virtual address space, which is consequntly unavialable for other threads. Although the virtual address space is large(3GB on x86), this factor may be significant limitation for processes employing large numbers of threads or threads that require large amount of memory. But processes can each employ the full range of avialable virtual memory(subject to limitation of RAM and SWAP Space)

Threads Or Processes?

- Dealing with signals in a multithreaded application requires careful design. It is usually desirable to avoid the use of signals in multithreaded programs.
- In a multithreaded application, all threads must be running the same program(but different functions). In a multiproces application different processes can run different programs.
- Aside from data, threads also also share other information(file descriptors, signal dispositions, current working direcotry, user and group Ids). This may be an advantage or disadvantage depending on the application.

Return values from pthread functions

•The **traditional method** of returning status from systems calls and some library functions is to return 0 on **success** and -1 on error with errno being set to indicate the **error**.

•The functions in the Pthread API do things differently. All **Pthread functions return** 0 on **success** and positive value on **failure**. The failure value is one of the same values that can be placed in errno by traditional unix system calls.

•Programs that use the **Pthread API's** must be **compiled** with the **gcc -lpthread** so that the program is linked with te libpthread library.

VIVEN Embedded Academy

Thread Synchronization

- Threads use two tools to synchronize their actions:
  - **Mutexes** allows threads to synchronize their use of shared resources, so that one thread doesnt try to access a shared variable at a the same as another thread is modifying it
  - **Condition variables** perform a complementary task: they allow threads to inform each shared variable or shared resource has changed state.
- **Critical Section:** Section of code that accesses a shared resources and whose execution should be **atomic**. That is its execution should be interrupted by another thread that simultaneously access the same shared resource.

Mutexes

- To avoid problems that can occur when threads try to updat a **shared variable**, we must use a **mutex**(short for mutual exclusion) to ensure that only one thread at a time can acces the variable. Generally, mutexes can be used to ensure **atomic access** to any shared resources.
- A mutex has two states: **locked** and **unlocked**. At any moment, at most one thread may hold the lock on a mutex. Attempting to lock a mutex that is already locked either blocks or fails with an error, depending on the method used to place the lock.
- When a thread locks a mutex, it becomes the owner of that mutex. Only the mutex owner can unlock the mutex. Acquir and release are also used synonymously for **lock** and **unlock**

Mutexes (continued)

- In general, we employ a different mutex for each shared resources(which may consists of multiple related variables) and each thread employs the following protocol for accessing a resources:
  - Lock the mutex for the shared resource.
  - Access the shared resource, and
  - Unlock the mutex.
- If multiple threads try to execute this block of code(a **critical section**), the fact that only one thread can hold the mutex(the others remain blocked) means that only one thread ata time can enter the block.
- A mutex is a variable of type **pthread_mutex_t**. Before it can be used a mutex must always be initialized. For statically allocated mutex, we can do this by assigning it the value **PTHREAD_MUTEX_INITILIZER** as below.

  pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER.

Locking and Unlocking Mutexes

- To lock and unlock we use the below functions.

*int pthread_mutex_lock(pthread_mutex_t *mutex);*
*int pthread_mutex_unlock(pthread_mutex_t *mutex);*

- To lock a mutex we must specify the mutex in a call to **pthread_mutex_lock()**. If the mutex is currently unlocked, this call locks the mutex and returns immediately. If the mutex is currently locked by another thread, then **pthread_mutex_lock()** blocks until the mutex is unlocked, at which point it locks the mutex and returns.
- If the calling thread itself has already locked the mutex given to **pthread_mutex_lock(),** then for the default type of mutex, one of the two implementations defined possibilities may result:
  - The thread deadblocks, blocked trying to lock a variable it already owns
  - or the call fails, returns the error EDEADLK.

Locking and Unlocking Mutexes(cont...)

- The pthread_mutex_unlock() function unlocks mutex previously locked by the calling thread. It is an error to unlock a mutex that is no currently locked, or to unlock a mutex that is locked by another thread.
- If more than one other thread is waiting to acquire the mutex unlocked by a call to pthread_mutex_unlock(), it is indeterminant which thread will succeed in acquiring it.
- The **pthread API** provides two variants of pthread_mutex_lock(). *pthread_mutex_trylock()* and *pthread_mutex_timedlock()*.
  - The *pthread_mutex_trylock()* function is the same as *pthread_mutex_lock()*, except that if the mutex is currently locked, *pthread_mutex_trylock()* fails, returning the error EBUSY.

VIVEN Embedded Academy

Locking and Unlocking Mutexes(cont...)

- The *pthread_mutex_timedlock()* is the same as *pthread_mutex_lock(* except the caller can specify an additional argument, *abstime*, that places a limit on the time that the thread will sleep while waiting to acquire the mutex. If the interval specified by its *abstime* agument expires without the caller becoming the owner of the mutex, pthread_mutex_timedlock() returns the error ETIMEDOUT.
- The *pthread_mutex_trylock()* and *pthread_mutex_timedlock()* functions are much less frequently used than *pthread_mutex_lock()*. In most well designed applications, thread should hold mutex for only a short time, so that other threads are not prevented from executing in parallel. This guarntees that other threads that are blocked on the mutex will soon be granted a lock on the mutex. A thread that uses the pthread_mutex_trylock() to periodically poll the mutex to see if it can be locked risks being starved of access to the mutex.

Threads

- –Threads are "light weight processes" (LWPs)
- –Heavy-weight processes" (HWPs) have a significant amount of overhead when switching.
- –Threads reduce overhead by sharing fundamental parts (code ("text"), data (VM), file I/O, and signal tables).
- –There are two types of threads: user-level and kernel level.

Kernel-level threads

- –Kernel-level threads are implemented in the kernel using several tables.
- –Kernel schedules each thread within the time slice of each process.
- –I/O blocking is not a problem as in case of user-level threads.
- –Processes automatically can take advantage of SMPs with kernel-level threads.
- –asynchronous I/O implemented using this.

Light Weight Process

- Kernel supported user threads.

drawbacks:

not suitable for appln with large no of threads

too many mode switches

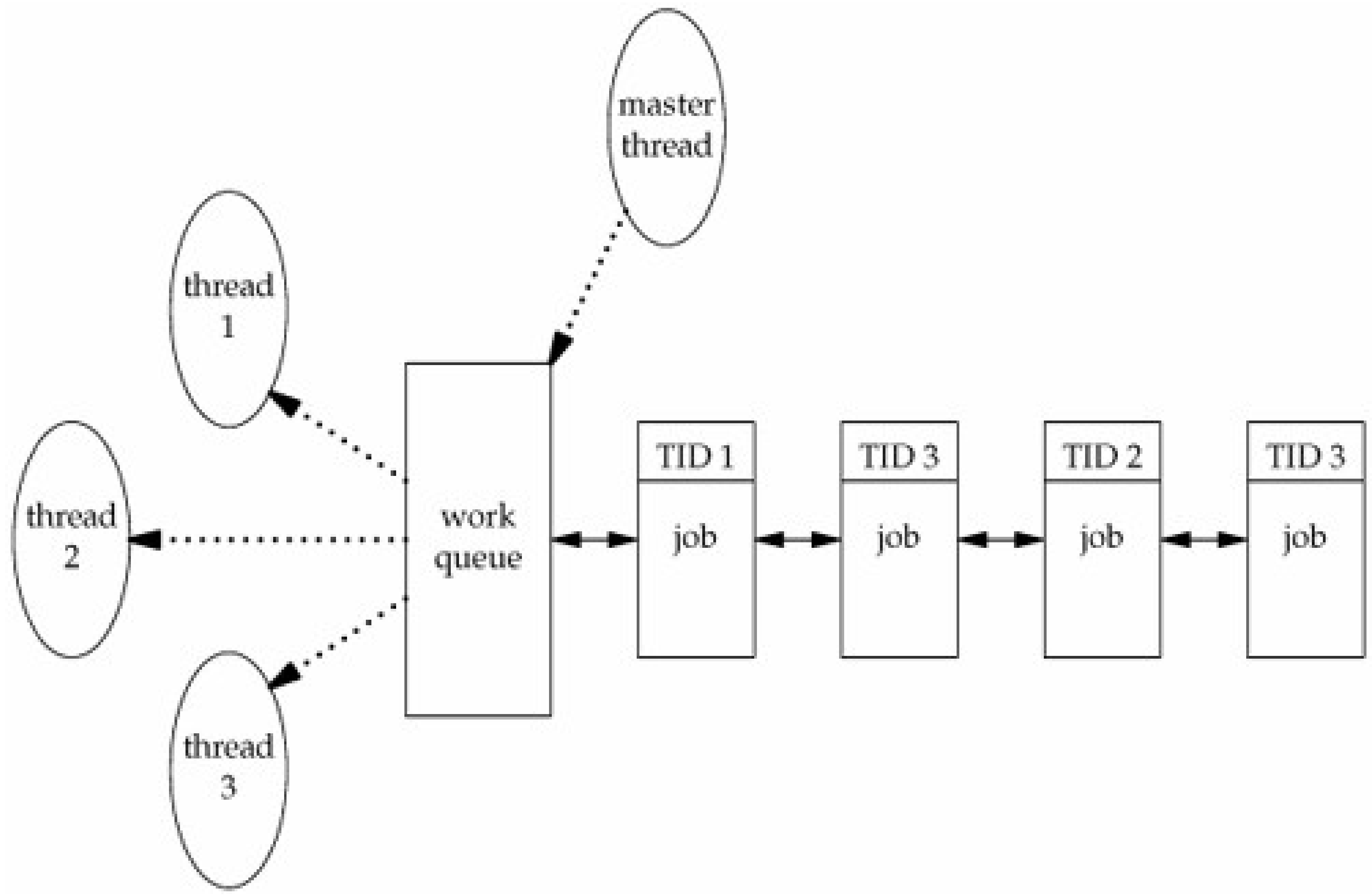
not fair when on process creates more LWPs.

	Creation time	sync time with sem
user thread	52	66
LWP	350	390
process	1700	200

User-level threads

- –User-level threads avoids the kernel and manages the tables itself.
- –User threads typically can switch faster than kernel threads.
- –User-level threads have a problem that a single thread can monopolize the timeslice.
- –Some user-thread libraries have addressed these problems.
- –without kernel support, user threads may improve concurrency but do not increase parallelism. even on multiprocessor's, user threads sharing a single LWP can not execute in parallel.

Work queue example



Thread creation

- When a program is started, the resulting process consits of a single thread called the **initial** or **main thread.**

#include <pthread.h>

*int pthread_create(pthread_t *thread, pthread_attr_t *attr, void*(*start_routine)(void *), void *arg)*

- **pthread_create()** creates a new thread, much as fork creates a new process with attributes specified by attr(stack size, detach stat) within a process.The new thread commences execution by calling the function identified by **start_routine** with arguments in arg.
- The thread that call **pthread_create()** continues execution with next statements that follows the call.
- Upon successful completion, function stores the ID of the created thread in the location referenced by **thread(**first argument).
- If you need to pass more than one argument to the start_rtn function, then you need to store them in a structure and pass the address of the structure in arg.

Thread creation ....

- When a thread is created there is no guarantee which runs first i.e. the newly created thread or the calling thread.
- If the start_routine returns, the effect is as if there was an implicit call to pthread_exit() using the return value of start_routine as the exit status.
- If pthread_create() fails, no new thread is created and the contents of the location referenced by thread are undefined.

Thread termination

- The execution of a thread terminates in one of the following ways without stopping the entire process.
  - The threads **start_routine** function performs a return specify a return value. The return value is the threads exit code.
  - The thread can be cancelled by another thread in same process using **pthread_cancel()**.

  int pthread_cancel(pthread_t tid);
  - Any of the threads call **exit()**, or the main thread performs a **return** in the main function, which causes all threads in the process to terminate.
  - The thread can call **pthread_exit()**

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

Thread termination

VIVEN Embedded Academy

- The pthread_exit() function terminates the calling thread and makes the value **value_ptr** available to other threads in the process by callin the **pthread_join** function.
- Calling ***pthread_exit()*** is equivalent to performing a return in the thread's start function, with the difference that ***pthread_exit()*** can be called from any function that has been called by the thread's start function.
- The ***retval*** argument specifies the return value for the thread. The value pointed to by retval should nor be located on the threads stack, since the contents of that stack become undefined on thread termination. (The region of the process's virtual memory may be reused by the stack for a new thread). The same statement applies to the value given to a return statement in the thread's start function.
- If the main thread calls **pthread_exit()** instead of of calling **exit()** or performing a **return**, then the other threads continue to execute.

Wait for thread termination

#include <pthread.h>

*int pthread_join(pthread_t thread, void **value_ptr);*

–The pthread_join() function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

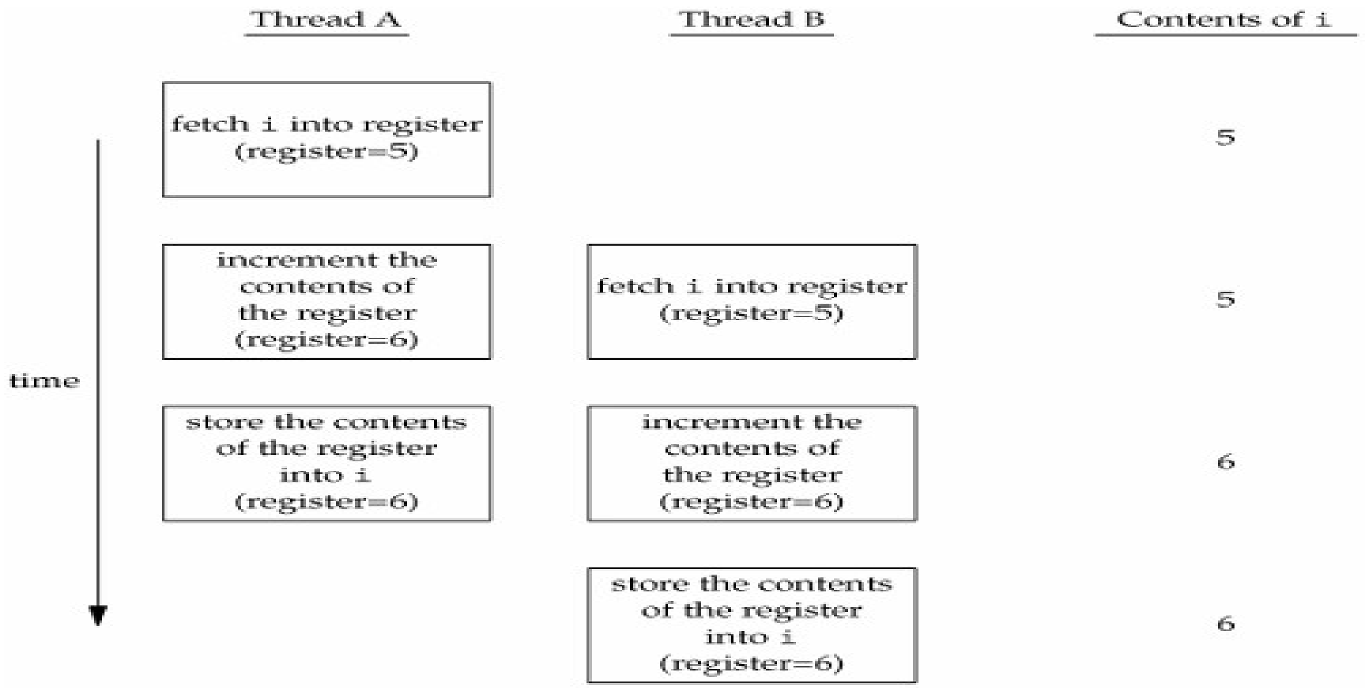
–On return from a successful pthread_join() call with a non-NULL value_ptr argument, the value passed to pthread_exit() by the terminating thread is made available in the location referenced by value_ptr.

–If successful, the pthread_join() function returns zero.

Synchronization between threads

When multiple threads of control trying to increment the same variable at the same time, the increment operations are usually broken down into three steps.

- Read the memory location into a register.
- Increment the value in the register.
- Write the new value back to the memory location.


Thread A
Thread B
Contents of i
time
fetch i into register (register=5)
5
increment the contents of the register (register=6)
fetch i into register (register=5)
5
store the contents of the register into i (register=6)
increment the contents of the register (register=6)
6
store the contents of the register into i (register=6)
6

Synchronization with mutexes

–Mutexes, which act as a mutual exclusion device to protect critical sections of code.

–Mutexes act by allowing the programmer to “lock” an object so that only one thread can access it.

–To control access to a critical section of code you lock a mutex before entering the code section and then unlock it when you have finished.

Synchronization with mutexes

The basic functions required to use mutexes are:

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const
            pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

As usual, 0 is returned for success, and on failure an error code is returned, but errno is not set.

Synchronization with mutexes ...

- A thread that becomes the owner of a mutex is said to have acquired the mutex and the mutex is said to have become locked.
- When a thread gives up ownership of a mutex it is said to have released the mutex and the mutex is said to have become unlocked.

pthread_mutex_init()

initialises the mutex referenced by *mutex* with attributes specified by *attr*.

#include <pthread.h>

*int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);*

- Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

- On success , the pthread_mutex_init() and pthread_mutex_destroy() functions return zero, else an error number is returned to indicate the error.

pthread_mutex_lock()

- The mutex object referenced by mutex is locked by calling pthread_mutex_lock().
- #include <pthread.h>
  *int pthread_mutex_lock(pthread_mutex_t *mutex);*
- If the mutex is already locked, the calling thread blocks until the mutex becomes available.
- Attempting to relock the mutex causes deadlock.
- If successful, the pthread_mutex_lock() return zero.
  - Otherwise, an error number is returned to indicate the error.

pthread_mutex_unlock()

- The pthread_mutex_unlock() function releases the mutex object referenced by mutex.

- #include <pthread.h>

*int pthread_mutex_unlock(pthread_mutex_t *mutex);*

- The manner in which a mutex is released is dependent upon the mutex's type attribute(Scheduling policy).

- If successful, the pthread_mutex_lock() return zero.

- Otherwise, an error number is returned to indicate the error.

pthread_mutex_destroy()

- The pthread_mutex_destroy() function destroys the mutex object referenced by mutex;

- #include <pthread.h>

*int pthread_mutex_destroy(pthread_mutex_t *mutex);*

- A destroyed mutex object can be re-initialised using *pthread_mutex_init();*

- It is safe to destroy an initialised mutex that is unlocked.

- On succes it returns zero, otherwise an error number is returned to indicate the error.

Network Programming Sockets

VIVEN Embedded Academy

Sockets

– Socket represent end-point of network communication link. Communication is typically established between two endpoints. That is between two sockets.Each socket is associated with the following five parameters.
  - Protocol used(datagram(UDP)/stream(TCP).
  - Source IP address.
  - Destination IP address.
  - Source port address.
  - Destination port address.

Sockets

- Both server and client network programs will create socket first. Then the server network program will bind the socket to a well-known port address. The well known port address allows the client network programs to connect to that server socket.
- These server side sockets, which are not yet connected to client side sockets are said to be half bind. This is because these sockets do not have destination IP address and destination port address.
- Once client connects to the server socket, then both sockets(client side socket and server side socket) are fully bind.

Arguments, Options and the environment

- This discussion examines
  - How C Programs access their command-line arguments.
  - Describe standard routines for parsing options.
  - Takes a look at environment.

Options and Argument Conventions

- This word argument has two meanings.

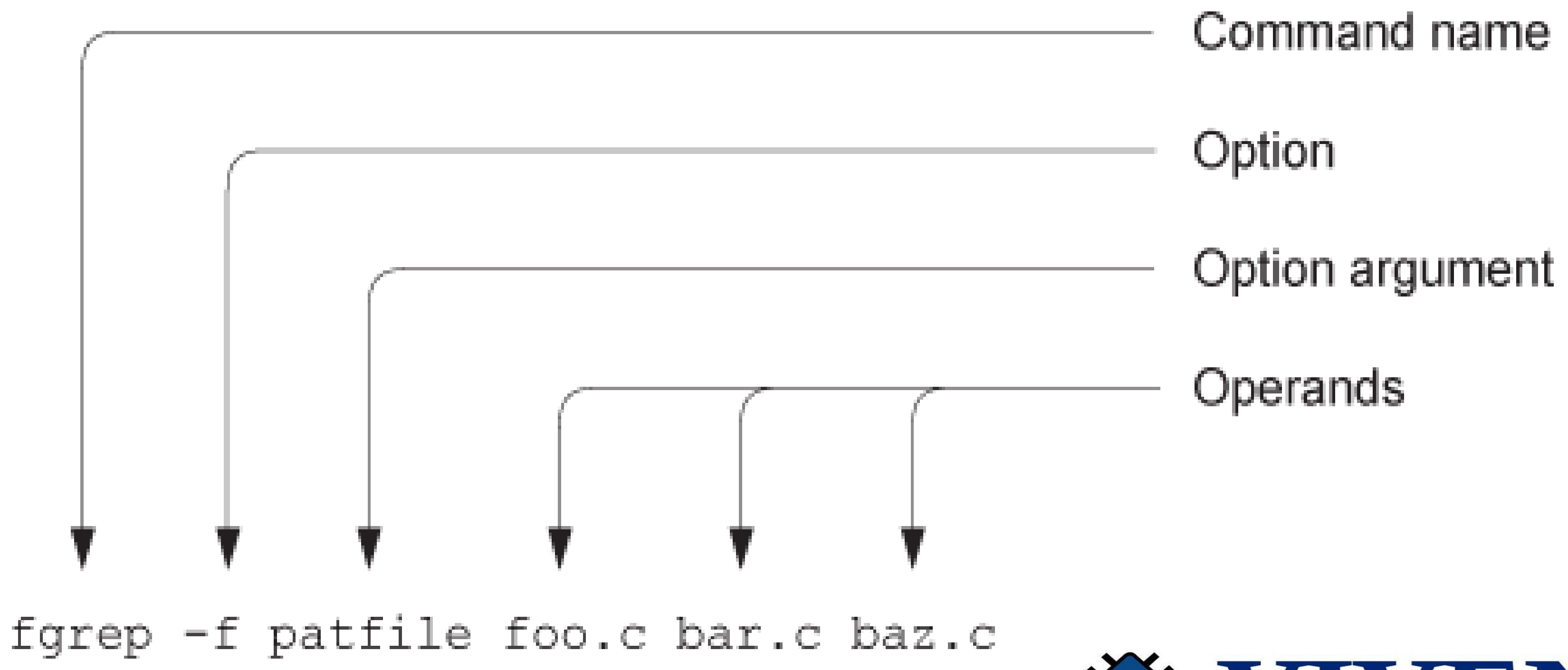
  Ex: $ls main.c opts.c process.c

  - Technical
    - All four words are made available to the program as its arguments.
  - Formal
    - Arguments are all the words on the command line except the command name.
- **Arguments** can be further classified as **options** and **operands**. In the previous example the options were operands(files for ls).
- **Options** are special arguments that each program interprets. They change the program behaviour or they provide information to the program. Options starts with a dash '-' and consists of a single letter.

VIVEN Embedded Academy

Options and Argument Conventions(2)

- **Options arguments** are information needed by an option, as opposed to regular operand arguments.
- Thus, patfile is not a data file to search, its for use by fgrep in defining the list of strings to search for.



CREATING SHARED LIBRARIES

VIVEN Embedded Academy

Shared library

- What is a shared library?
  - A library that is loaded into physical memory only once and reused by multiple processes via virtual memory. Generally shared libraries are .so(or in windows .dll) files.
- Why shared libraries??
  - They reduce memory consumption if used by more than one process and they reduce the size of the executable.
  - They make developing application easier: a small change in the implementation of a function in the library don;t need the user to recompile and re-link his application code every time. You need to only re-link if you make incompatible changes, such as adding arguments to a call or changing the size of a struct.

Creating object files for Shared Library

- All the code that goes into a shared library needs to be **position independent**. We can make gcc emit position-independent code by passing it one of the command-line switches -fpic or -fPIC(the former is preferred, unless the modules have grown so large that the relocatable code table is simply too small in which the compiler will emit an error message, and you have to use -fPIC.
- Firstly create object files for all .c file that goes into shared library.
  - Gcc -c -fPIC calc_mean.c -o cal_mean.o

  (We are compiling calc_mean.c with -fPIC option and generating calc_mean.o object file.

VIVEN Embedded Academy

Creating the Shared Library(1)

•Every **shared library** has a special name called the "**soname**". The **soname** has the prefix "lib", the name of the library, the phrase ".so", followed by a period and a version number that is incremented whenever the interface changes. On a working system a fully qualified **soname** is simply a **symbolic link** to the shared library's "**real name**".

•Every shared library also has a **"real name"**, which is the filename containing the actual library code. The real name adds to the soname a period, a minor number, another period, and the release number. The last number support configuration control by letting you know exactly what version(s) of the library are installed.

All the code that goes into a shared library needs to be position independent. We can make gcc emit position-independent code by passing -fpic or -fPIC.

gcc -c -fPIC calc_mean.c -o calc_mean.o

VIVEN
Embedded Academy

Creating the Shared Library(2)

gcc -shared -Wl,-soname,libmean.so.1 -o libmean.so.1.0.1 calc_mean.o

Above command on successful produces a shared library named "libmean.so.1.0.1".

-shared: Produces a shared object which can then be linked with other objects to form an executable.

-W1: Pass options to linker.

In this example the options to be passed on to the linker are: "-soname libctest.so.1". The name passed with the "-o" option is passed to gcc.

Now we have successfully created a shared library named "libmean.so.1.0.1". Let us see how to include this shared library in our application.

```
    mv libmean.so.1.0.1 /home/nimsme/slib
```

Creating the Shared Library(3)

VIVEN Embedded Academy

Create symbolic links to the created shared library

```
# ln -sf /home/cf/slib/libmean.so.1.0.1 /home/cf/slib/libmean.so
# ln -sf /home/cf/slib/libmean.so.1.0.1 /home/cf/slib/libmean.so.1
```

Use this command to link to your program

```
gcc -o main main.c -lmean -L/home/cf/slib
```

The “-l” option tells the compiler to look for a file named libmean.so The mean is specified by the argument immediately following the “-l”. i.e. -lmean

If compilation is successful an executable named "a.out" is created.

We can check if our library is include successfully into the executable by linker using the following command

```
# $ ldd main
# linux-gate.so.1 =>  (0x00110000)
# libmean.so.1 => not found
# libc.so.6 => /lib/libc.so.6 (0x007bf000)
# /lib/ld-linux.so.2 (0x0079f000)
```

Creating the Shared Library(4)

We have to create or set the environment variable "LD_LIBRARY_PATH" to the directory containing the shared

```
export LD_LIBRARY_PATH=/home/nimsme/lib
```

Now recompile the program

```
gcc -o main main.c -lmean -L/home/cf/slib
```

Now check for ldd command

```
$ ldd main
linux-gate.so.1 =>  (0x00110000)
libmean.so.1 => ./libmean.so.1 (0x00111000)
libc.so.6 => /lib/libc.so.6 (0x007bf000)
/lib/ld-linux.so.2 (0x0079f000)
```

MEMORY MAPPINGS

VIVEN

Embedded Academy

mmap()

- The **mmap()** system call creates a new **memory mapping** in the calling process's virtual address space. A mapping can be of two types.
  - **File mapping:** A file mapping maps a region of a file directly into the calling process's virtual memory. Once a file is mapped, its contents can be accessed by operations on the bytes in the corresponding memory region. The pages of the mapping are(automatically loaded) from the file as required. This type of mapping is also known as a **file-based mapping** or **memory-mapped file**.
  - **Anonymous mapping:** An anonymous mapping doesnt have a corresponding file. Instead the pages of the mapping are initialized to 0.

mmap()

- The memory in one process's mapping may be shared with mapping in other processes.(i.e., the page-table entries of each process point to the same pages of RAM). This can occur in two ways.
  - When two processes map the same region of a file, they share the same pages of physical memory.
  - A child process created by fork() inherits copies of its parent's mapping, and these mappings refer to the same pages of physical memory as the corresponding mappings in the parent.

When two or more process share the same pages, each process can potentially see the changes to the page contents made by other processes, depending on whether the mapping is **private** or **shared**.

Private Mapping & Shared Mapping

- **Private mapping(MAP_PRIVATE)**: Modifications to the contents of the mapping are not visible to other processes and, for a file mapping, are not carried through to the underlying file. Although the pages of a private mapping are initially shared in the circumstances describe above, changes to the contents of the mapping are nevertheless private to each process. The kernel accomplishes this using **copy-on-write** technique. This means that when ever a process attempts to modify the contents of a page, the kernel first creates a new, separate copy of that page for the process(and adjusts the process's page tables).
- **Shared mapping(MAP_SHARED)**: Modifications to the contents of the mapping are visible to other processes that share the same mapping and, for a file mapping, are carried through to the underlying layer.

Mmap() function

VIVEN Embedded Academy

#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

- **Addr** argument indicates the virtual address at which the mapping is to be located. If we specify addr as NULL, the kernel chooses a suitable address for the mapping. This is the preferred way of creating a mapping. We can specify a non-NULL value as addr, which the kernel takes as a hint about the address at which the mapping should be placed. Kernel chooses an address that doesn't conflict with any existing mapping.
- On success mmap() returns the starting addrses of the new mmaping.
- The **length** argument specifies the size of the mapping in bytes.
- The **prot** argument is a bit mask specifying the protection to be placed on the mapping. It can be either PROT_NONE or combination of another three flags PROT_READ, PROT_WRITE, PROT_EXEC.
- The flag argument is a bit mask of options controlling various aspects of the mapping operation. MAP_PRIVATE, MAP_SHARED.
- Offset's are used with file mappings, they are ignored for anonymous mapping.

Linux Boot Process

From Power ON to Loading Kernel:

- **Step1:** When we power on PC, BIOS(Which is stored on Motherboard) loads into RAM. The purpose of BIOS is load to load OS or Kernel into RAM.
- **Step 2:** BIOS search for Bootable Device. When bootable device found goes to next step.
- **Step 3:** When bootable device found it loads 1 stage Boot loader i.e. MBR in RAM. Size of MBR is just 512 bytes. Just first sector of hardisk.
- **Step 4:** First stage boot loader loads Second stage boot loader i.e. GRUB or LILO.
- **Step 5:** When second stage boot loader gets executed in RAM, Splash screen gets displayed. Job of second stage boot loader is to load kernel in RAM.

Linux Boot Process

Step 6: Stage 2 boot loader loads kernel and optional initial Root File system into RAM. It passes control to kernel and kernel get decompressed into RAM and get initialized. At this second stage boot loader checks hardware and mount root device also loads necessary kernel modules. When it completes first Userspace program gets executed i.e. init. Init is father of all processes.

From init to Login prompt:

When the kernel is loaded, it immediately initializes and configures the computers memory and configures various hardware attached to the system, including all processors, i/o subsystems, and storage devices. It then looks for the compressed initrd in a predetermined location in memory. Decompresses it, mounts it and loads all necessary drivers.

After this the kernel locates & Starts the first user-space application /sbin/init.

Linux Boot Process

Init is the father of all processes. Its PID is 1.
Before /sbin/init loads into RAM, it reads /etc/inittab file
/etc/initab is an ASCII text file. Where we can configure multiple parameters for init daemon.

Debugging a core dump(1)

- This process is quick and easy and often leads to immediate identification of the offending code. A core dump file is generated by the kernel when an application program generates a fault, such as accessing a memory location it does not own.
- Many conditions can trigger a core dump, but **SIGSEGV** (segmentation fault) is by far the most common. A **SIGSEGV** is a Linux kernel signal is generated on illegal memory access by a user process. When this signal is generated, the kernel terminates the process. The kernel then dumps a core image if it is so enabled.
- To enable the generation of core dump, your process must have the authority to enable a core dump. This is achieved by setting the process's resource limits using the **setrlimit()** C function call, or from the BASH shell command prompt using **ulimit**.

Debugging a core dump(2)

VIVEN Embedded Academy

- It is common to find the following line in the initialization scripts of an embedded system to enable the generation of core dumps on process errors.

  ulimit -c unlimited (-c:The maximum size of core files created)

- When an application program generates a segmentation fault, Linux terminates the process and generates a core dump. The core dump is a snopshot of the running process at the time the seg fault has occured.
- GDB produces much more useful output with debugging symbols(gcc -g) enabled during build. A core dump file with name core.xxxx gets created in your current directory.

```
[vamsi@localhost ~]$ gcc -g while.c
[vamsi@localhost ~]$ ./a.out
Segmentation fault (core dumped)
[vamsi@localhost ~]$ gdb a.out core.6838
```

l- displays the entire code in gdb

Debugging a core dump(3)

- The last line printed upon GDB start-up is the current location of the program when the fault occurred The line preceded by the #0 string indicates the stack frame. (stack fram zero in a function at some virtual address).
- From here, we can use the **backtrace(bt)** command to see the call chain leading to this error, which might take us back to the source of the error. The **backtrace** displays the call chain all the way back to main(), the start of the user's program. A stack frame number precedes each line on the **backtrace**. You can switch to any given stack frame using the **frame** command which displays the source code in that frame.
- So crashes of this type are very easy to track down using GDB and core dumps.

strace

- **Strace** captures and displays information for every kernel system call executed by a Linux application program. It can be run on programs for which no source code is available.

  strace ./a.out

- Starting the application from command line simply returns control to console. It produces some systems logs and strace quickly identifies the problems. The output can be some times hundreds of lines. Each line represents a discrete kernel system call. We dont need to understand each and every line of the trace. We are needed to look for any anomalies which might help pinpoint why the program won't run.

Strace variations(1)

- **Strace** utility has many command line options. One of the more useful is the ability to select a subset of system calls for tracing. For example if you want to see only the network related activity of a given process, issue the command as follows:

  strace -e trace=network a.out

- This produces a trace of all the network related system calls, such as socket(), connect(), recvfrom(), send(). This is a powerful way to view the network activity of the given program. Several other subsets are available. For example you can view only a program's file related activity by using "**trace=file**" with open(), close(), read(), write() and so on. Additional subset include process related system calls, signal related system calls, and IPC related system calls.
- It is worth nothing that **strace** can deal with tracing programs that spawn additional processes. Invoking **strace** with the -f option instructs **strace** to follow child process that are created using the **fork()** system call.

Strace variations(2)

- One very useful way to employ **strace** is to use the -c option. This option produces a high level profiling for your application. Using -c option accumulates statistics on each system call,
  - how many times it was encountered,
  - how many time errors were returned
  - time spent by each system call

  strace -c ./a.out
- Some errors might be normal part of your applications operations, but others might be consuming time in ways that you did not anticipate.

ltrace

- The **ltrace** and **strace** utilities are closely related. The **ltrace** utility does for **library calls** what **strace** does for **system calls**.

  ltrace ./a.out

- Reproduces the output of **ltrace** on a small program that executes a handful of standard C library calls.
- For each library call, the name of the call is displayed, along with varying portions of the parameters to the call. Similar to strace, the return value of the library call is then displayed. As with strace, this tool can be used on programs for which source code is unavailable.
- Similar to **strace**, a variety of switches affect the behaviour of **ltrace**. You can display the value of the program counter at each library call, which can be helpful in understanding your applications program flow. ***ltrace -c ./a.out***
- The ltrace tool is available only for programs that have been compiled to use dynamically linked shared library objects.

mtrace(1)

- **Mtrace** is a simple utility that analyses and reports on calls to malloc(), realloc(), and free() in your applications. It is easy to use and can potentially help spot trouble in your application.
- **Mtrace** must be configured and compiled in your target architecture. **Mtrace** is a malloc replacement library that is installed on your target. So your embedded Linux distribution should contain the **mtrace** package.
- For you to use mtrace, three conditions must be satisfied.
  - A header file **mcheck.h**, must be included in your source file.
  - The application must call mtrace() to install the handlers.
  - The environment variable MALLOC_TRACE must specify the name of a writeable file to which the trace data is written.
- When these conditions are satisfied, each call to one of the traced functions generates a line in the raw trace file defined by MALLCO_TRACE.

mtrace(2)

- The trace data look like this in the file mentioned in environment.

@ ./a.out:[0x804841e] + 0x9472390 0x10

- The **@** sign signals that the trace line contains an address or function name. The program is executing at the address in square brackets, 0x8048ec. The + sign that this is a call to allocate memory. A call to free would be indicated by minus sign. The next field indicates the virtual address of the memory location being allocated or freed. The last field is the size.
- The data format is not very user friendly. For this reason the **mtrace** utility includes a utility that analyses the raw data and report any inconsistencies. The analysis utility is a perl script supplied with mtrace package. In the simplest the Perl script prints a single line with the message **No Memory Leaks**.

```
[vamsi@localhost day37]$ mtrace ./a.out output.txt
Memory not freed:
Address     Size     Caller
0x088e3378     0x10  at /media/500GB/CLasses/day37/mtrace.c:15
```

mtrace(3)

- This simple tool can help you spot trouble before it happens as well as find trouble when it occurs. Notice that the Perl script displays the filename and line number of each call to **malloc()** that does not have a corresponding call to **free()** for the given memory location.
- This requires debugging information in the executable file generated by passing the -g flag on the compiler. If no debugging information is found the script simply reports the address of the function **malloc().**

dmalloc(3)

- **Dmalloc** picks up where **mtrace** leaves off. The **mtrace** package is simple, relative non intrusive package most useful for simple detection of **malloc/free** unbalance conditions. The **dmalloc** package lets you detect much wider range of dynamic memory-management errors. Compared to **mtrace, dmalloc** is highly intrusive.
- Depending on the configuration, **dmalloc** can slow your application to crawl. It is definitely not the right tool if you suspect memory errors due to race conditions or other timing issues. **dmalloc** will definitely change the timing of your application.
- Dmalloc is a debug malloc library replacement. These conditions must be satisfied.
  – Application code must include the **dmalloc.h** header file.
  – The application must be linked against the **dmalloc** library.
  – The **dmalloc** library and utility must be installed on your embedded target.
  – Certain environment variables that the dmalloc library references must be defined before you run your application on target.

References

- http://www.opengroup.org/onlinepubs/
- http://www.informatik.uni-hamburg.de/RZ/software/gnu/libraries/libc_toc.html
- http://www.linux-tutorial.info/modules.php?name=MContent&pageid=