Operating Systems

1 Processes

1.1 Introduction

Processes provide:

concurrency

isolation Each process has its own address space

simplicity

better ultilization of resources Different processes require different resources at the same time

1.2 Concurrency

time-slicing OS switched application running on CPU for every 50ms

Pseudo concurrency single processor schedules multiple processes

Real concurrency multiple processors run multiple processes

1.3 Context Switches

Context Switch The processor switches from executing process A to process B

When a context switch happens, the information associated with the process is stored in a **process descriptor** or **process control block**, which is kept in the process table.

2 Process Hierarchy

2.1 Creating child processes in Unix

fork creates a fork process of exactly same memory of the **current thread** of the current process.
fork returns 'twice' —process id of the child process from the parent process, and from the child process.
If an error happened, fork returns 0.

3 Unix Inter-Process Communication

3.1 Signals

Signal delivery is similar to delivery of hardware interrupts.
kill() can be used programmatically to deliver a signal.

SIGINT Interrupt from keyboard

SIGABRT Abort

SIGFPE Floating point exception

SIGKILL Kill signal

SIGSEGV Segmentation reference

SIGPIPE Broken pipe

SIGALRM Timer signal from **alarm**

SIGTERM Termination signal

3.2 Handling Signals

Process can use **signal handler** to handle the signals, using signal(SIG, function pointer) system call. SIGKILLL and SIGSTOP cannot be manually handled.

3.2.1 Signals in multithreaded processes

Each native thread has attributes indicating whether this thread is responsible for handling signals. So it's up to the programmer to configure how to handle signals.

3.3 Unix Pipes

A **pipe** is a method of connecting the standard output of one process to the standard input of another. So it is one-way.
int pipe(int fd[2]) creates pipe:

- fd[0] the read end of the pipe
- fd[1] the write end of the pipe

Useful when combined with fork(). Sender should close the read end fd[0], writer should close the write end fd[1]. If no data is available to read, read will block.

3.4 Unix Named Pipes

Also named **FIFO**s. **Persistent pipes** that outlive process which created them. Stored on file system, and can be used just like files. It's completely in-memory so more efficient than real temp files.

```
$ mkfifo /tmp/abc
$ echo ABC > /temp/ac
```

4 Threads

Thread Thread are execution streams that share the same address space.

Threads is like lightweight processes.
Per process items:

- Address space
- Global variables
- Open files
- Child processes
- Signals

Per thread items:

- Program counter
- Registers
- Stack

4.1 Compare Threads with Processes

Threads are lightweight while processes are heavyweight.
Threads have shared memory space, they can write into the others' memory, while processes can't access memory of others.

4.2 Kernel-Level Threads

Kernel-Level threads are managed by OS kernel.

4.2.1 Advantages of Kernel Threads

- Blocking system calls and page faults can be easily accommodated. If these happen, OS can schedule a runnable thread from the same process.

4.2.2 Disadvantages of Kernel Threads

- Thread creation and termination are more expensive. Because it requires system calls. Can use thread pools to mitigate.
- Thread synchronisation is expensive. Requires blocking system calls.
- Thread switching is expensive. Requires system calls.
- No application-specific schedulers.

4.3 User-Level Threads

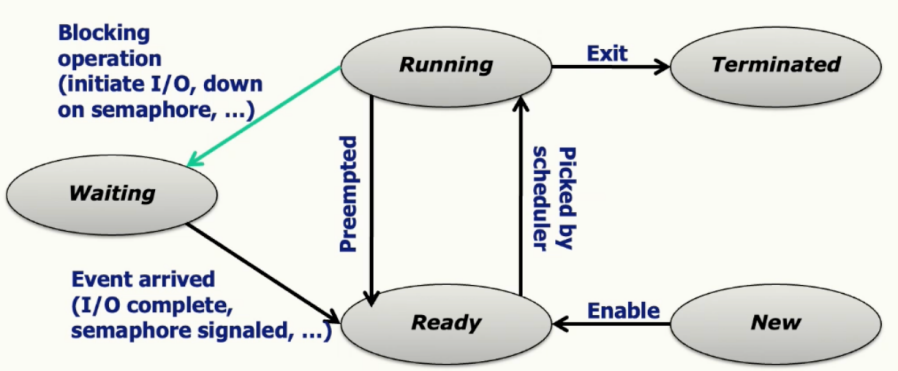
User-Level threads are implemented by software library. Process maintains *thread table* for *thread scheduling*.

4.4 Hybrid Approaches

Use kernel threads and multiplex user-level threads onto some kernel threads. (Process maps user-level threads onto kernel threads. e.g. Java's Threading system)

5 Scheduling

5.1 Process States



5.2 Scheduling Algorithms

Turnaround time is time needed for a job to complete after being submitted.

5.2.1 Goals of Scheduling Algorithms

Ensure fairness Comparable process should get comparable services

Avoid indefinite postponement No process should starve

Enfore policy e.g. priorities

Maximize resource utilisation CPU, I/O devices

Minimize overhead from context switches, scheduling decisions

5.2.2 First-Come First-Served Scheduling

- Runnable process added to the end of ready queue
- Non-preemptive

Advantages:

- No indefinite postponement
- Easy to implement

Disadvantages:

- Lower throughput
- Turnaround time can be long

5.2.3 Round-Robin Scheduling

- Process runs until it blocks or time quantum exceeded, and put in the back of ready queue

Advantages:

- Fair
- Short response time
- Short turnaround time when run-times differ

Context switches cause overhead. Large quantum reduces overhead but also cause worse response time. Quantum value should be much larger than context switch cost, but provide decent response time. Typical values: 10 200 ms.

5.2.4 Shortest Job First

Non-preemptive scheduling with run-times known in advance. Pick the shortest job first.
Advantages:

- Short turnaround time
- Short response time

5.2.5 Shortest Remaining Time

Preemptive version of the Shortest Job First. Choose process whose remaining time is shortest.

5.3 General-Purpose Scheduling

- Favour short and I/O-bound jobs to get good response utilisation and short response time.
- Quickly determine the nature of job and adapt to changes.

5.3.1 Multilevel Feedback Queues

Used in modern OS.
Drawbacks:

- Not very flexible. Applications have no control. Priorities make no guarantees.
- Does not react quickly to changes. Often needs warm-up.
- Cheating is a concern.
- Cannot donate priority.

5.3.2 Lottery Scheduling

Processes have lottery tickets for CPU time.

- Number of lottery ticks can change.
- Highly responsive. New job given p% of tickets has the p% change to get resource at next scheduling decision.

Advantages:

- No starvation
- Jobs can exchange tickets —allows priority donation and cooperating jobs to achieve certain goals.

Drawbacks:

- Adding or removing jobs affect remaining jobs' proportionality
- Unpredictable response time

5.4 Summary

Scheduling algorithms often need to balance conflicting goals.
Different scheduling algorithms appropriate in different contexts.

6 Synchronisation

Critical Section Section of code in which processes accesses shared resource

Mutual Exclusion Ensures that if a process is executing its critical section, no other process can be executing i

6.1 Using Locks

Lock Overhead measure of cost associated with using locks

Lock Contention measure of number of processes waiting for lock

6.2 Memory Models

In modern CPUs, other memory models than **Sequential Consistency** can exist due to compiler optimisations .

6.2.1 Sequential Consistency

- The operations of each thread appear in program order
- The operations of all threads are executed in some sequential order atomically

7 Memory Management

7.1 Paging

Frames Frames are fixed-sized blocks of physical memory.

Pages Pages are blocks of same size of logical memory.

Page table is set up when running a program.

7.1.1 Fragmentation

External Fragmentation Total memory exists to satisfy request, but not contiguous.

Internal Fragmentation Allocated memory larger than requested.

7.2 Address Translation

Address generated by CPU is divided into **page number**(p) and **page offset**(d).
Page number is used to look into page table.

7.2.1 Memory Protection

Valid-invalid bit attached to each page table entry.

7.2.2 Example: Translating Virtual Addresses to Physical Addresses

An embedded system uses a 16-bit big-endian architecture. It supports virtual memory management with one-level page table. It has a page size of 1 KByte. Each page entry contains the frame's physical address; the least significant bit represents valid bit; the second last significant bit represents dirty bit. Given entries in a page table, translate virtual memory address to physical memory address using the table.
The table is:

(0) 0x2C00

(1) 0x0403

(2) 0xCC01

(3) 0x0000

(4) 0x7C01

The addresses to translate:

(a) 0xB85

(b) 0x1420

(c) 0x1000

(d) 0xC9A

Solution:

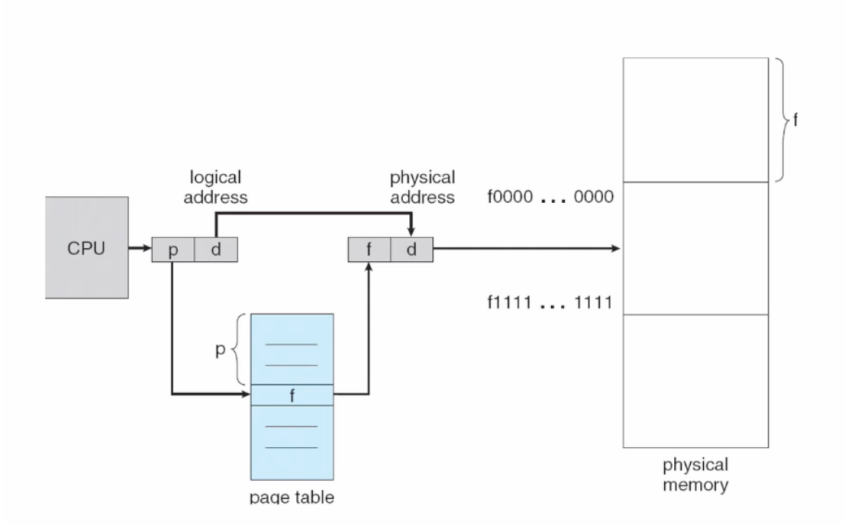


Figure 1: Graphical representation of Translating Virtual Address to Physical Address

1. Convert table entries from Hex to Bin. For example:

$$(0)\ 0x2C00 = (MSB)\ 0010\ 1100\ 0000\ 0000\ (LSB)$$
$$(1)\ 0x0403 = (MSB)\ 0000\ 0100\ 0000\ 0011\ (LSB)$$
$$(2)\ 0xCC01 = (MSB)\ 1100\ 1100\ 0000\ 0001\ (LSB)$$
$$(3)\ 0x0000 = (MSB)\ 0000\ 0000\ 0000\ 0000\ (LSB)$$
$$(4)\ 0x7C01 = (MSB)\ 0111\ 1100\ 0000\ 0001\ (LSB)$$

2. Page size is 1KByte, so we need 10 bits. Count 10 starting from LSB then we can extract remaining bits as page number. For $0x2C00$, page number is 001011.

3. Convert addresses from Hex to Bin.

   $0x0B85$ = (MSB) 0000101100000101 (LSB)

   Count 10 starting from LSB, which is the offset, then we can extract remaining bits as index into the page table. For $0x0B85$, index is 000010 which is 2 in ternary.

4. Lookup the entry with the index in the table, which is $0xCC01$. We get page number of $0xCC01$, which is 110011.

5. If we can't find the entry, fail with page fault.

6. Check if the entry is valid. For $0xCC01$, the LSB is 1, so it is valid, then we proceed. Otherwise, we fail with page fault.

7. Combine page number of $0xCC01$ with the offset, the final physical address is 1100111100000101.

7.2.3 Virtual Address Space

A pure paging system uses a 3-level paging table. Virtual Addresses are decomposed into 4 fields (a, b, c, d) with d being offset (then a, b, c corresponds to page number in each level).
So there are $2^(a+b+c+d)$ total addresses in the address space, $2^(a+b+c)$ pages,and 2^d on each page.

7.3 Page Table Implementation

Page table can be kept in memory, using

- **Page-table base register** points to page table
- **Page-table length register** indicates size

The problem is low efficiency. Every data access requires two memory accesses, one for page table and one for actual data.

7.3.1 Translation Lock-aside Buffers

To improve page table lookup performance, we use special fast-lookup hardware cache as associative memory.
Associative memory supports parallel search. For address translation (p, d), if it is in associative register, get frame number out directly. Otherwise, get page table from memory.
Some TLBs store **address-space ids**(ASID) in entries. They uniquely identify each process to provide address-space protection for that process.
TLBs usually needs to be flushed after context switch.
If we have too many processes, we will need too many bits for ASID. So ASID is not usually used by modern processors.

7.4 Effective Access Time

For an n-level page stable system, let L be time taken for TLB lookup, M be time for physical memory access, α be TLB hit rate, then

$$\begin{aligned}\text{average memory access time} &= \alpha * (L + 1 * M) \\ &\quad + (1 - \alpha) * (L + (n + 1) * M)\end{aligned}$$

This means if we hit TLB, then we need time for accessing TLB and time for accessing the real data in memory. If we did not hit TLB, then we need time for accessing TLB and n times the time for accessing $n + 1$ level page table, and 1 times the time for the real data in memory.

7.5 Hierarchical Paging

Comparing with One-Level Paring, In Two-Level Paging page number is divided into two parts, one for outer page table and another for inner page table. The offset both takes the remaining bits.

7.6 Inverted Page Table

Inverted Page Tables use some initial bits of physical address as index, instead of using that of virtual addresses. So it enables fast lookup from physical to virtual, but slow from virtual to physical.

7.7 Segmentation

A segment is

- Independent address space from 0 to maximum.
- Can grow/shrink independently.
- Support different kinds of protection (read/write/execute).
- Unlike pages, programmers are aware of segments.

Memory allocation to segments is harder due to variable size.

7.8 Demand Paging

Load pages only if they are needed, when requested.
If page is not loaded, a page fault is generated. Then OS will unload existing not-wanted pages and replace it with the wanted page from backing storage so this takes more tile. Then the instruction is restarted.
Let $0 \leq p \leq 1$ be page fault rate(0 is no page fault), Effective Access Time for Demand Paging is EAT = $(1-p) * \text{memory access} + p * (\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})$

7.8.1 Optimising fork

Forked processes share the same page tabel as its parent as read-only. When either parent or child attempts to write to the page, OS creates and applies a copy of the page.

7.9 Page Replacement Algorithms

Page Replacement Algorithms should be efficient to minimize page faults.

7.9.1 Belady's Anomaly

Intuitively we think the more frames, the less page faults. However, this is not always true.

7.9.2 Optimal Algorithm

The Optimal Algorithm replaces pages that won't be used for the longest period of time.
We can't do this in practice, but this is used for measuring how well other algorithms perform.

7.9.3 FIFO Algorithm

Always replace the oldest page.

7.9.4 LRU Algorithm

When page is referenced, copy clock into counter. When page needs to be replaced, choose the lowest counter.
LRU sounds good but is expensive as we need to count. We can apply approximations.

7.9.5 Algorithms Taking Approximations

- **Reference Bit** approximation assigns each page a reference bit, initially 0. When page is referenced, set the bit to 1. Periodically resting the bit to 0. Replace the page with 0.
- **Second change page replacement(clock policy)** Based on **reference bit** approximation, we use a circular pointer that points to the lsat replaced page index. When replacing, we always start searching from the pointer for pages with reference bit 0.
- **LFU** Count number of references for each page. **LFU** replaces pages with the smallest count. It may replace pages just brought to memory and always forget heavy page usages(pages that were used very frequently but is not used anymore still have a high counter).
- **MFU** Similar to **LFU**, but replaces pages with largets count.

Excessive paging activity can cause low processor utility, which is called **thrashing**.

8 Devices

Dedicated devices can only be used by one user at the same time, while shared devices can be used by multiple users simultaneously.

8.1 Spooling

By save some data into intermediate storage like file systems, some dedicated devices can be 'shared'.

8.2 Linux I/O Management

Devices grouped into **device classes**. Members of each device class perform similar functions.
Major and **Minor** identification numbers are used by device drivers to identify their devices. Devices with same **major** number are controlled by same driver, **minor** enables the system to distinguish between devices of same class.
Tips: major numbers are hard coded —you have to ask for the Linux Community or something to request a number.

8.2.1 Device Access in Linux

Device files accessed via **virtual file system**(VFS).
Most drivers implement read, write, seek.
Developer can use open to open a virtual file like /dev/mouse.
Some special operations, like ejecting CD-Rom, can be done using ioctl .

8.3 Buffering

8.3.1 Buffered I/O

Data is first transferred from/to the buffer when reading/writing. Process suspends only when buffer is empty/full.
Used to smooth peaks in I/O traffic.
Caters for differences in data transfer units between devices.

8.3.2 Unbuffered I/O

Perform physical read/write on each I/O request. Can cause high process switching overhead.

8.4 Device Data Communication

Devices can be either **character devices** or **block devices**.

8.4.1 Character Devices

Character devices transmits data as stream of bytes.

8.4.2 Block Devices

Kernel's block I/O system contains number of layers.
Caches are used, while developers can also use O_DIRECT to bypass caching for direct access.

8.5 Ways to Do I/O

- Programmed I/O: Some registers may change, meaning some event happened. Programs keep polling the register values.
- Interrupt-Driven I/O
- Direct Memory Access (DMA)

Most modern devices use Interrupt-Driven with Direct Memory Access.

9 Disk Management

Disk

10 Disk Devices

10.1 HDD Disk Performance

Disk performance is highly influenced by seeking.

10.1.1 Calculating Time for Access

$$t_{\text{access}} = t_{\text{seek}} + t_{\text{seek}} + t_{\text{transfer}} \qquad = t_{\text{seek}} + \frac{1}{2r} + \frac{b}{rN} \tag{1}$$

where:

- r —rotation speed in revolutions per second
- b —number of bytes to be transferred
- N —number of bytes per track

10.2 HDD Accessing Algorithms

10.2.1 First Come First Served

Do requests in queue. Bad performance.

10.2.2 SCAN Scheduling

Change reading direction when reaching outermost/innermost cylinder.
Good overall, and is most common. Long delays for requests at extreme conditions.

10.2.3 Linux Disk Scheduling Algorithms

Linux use variation of SCAN algorithm.

It has deadline scheduler that ensures reads always performed by deadline.

Anticipatory scheduler: delay after read request completes.

10.3 RAID

RAID (Redundant Array of Inexpensive Disks) uses array of physical devices as a single logical device.

Stores data distributed over array of physical devices to allow parallel operations.

Can use redundant disk capacity to respond to disk failures. More disks, lower mean-time-to-failure.

10.3.1 RAID Level 0 —Striping

No redundant disks. Striping just improves speed, but it will be more error prone.

10.3.2 RAID Level 1 —Mirroring

One redundant disk for each device —same data stored on both disks.

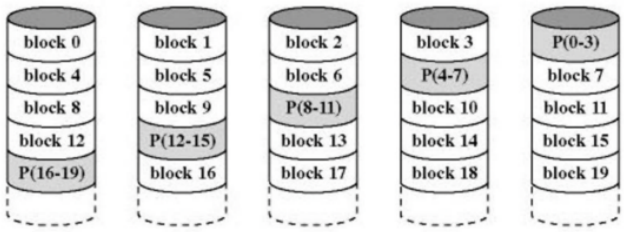
Reads can be services by either disk, so fast. Writes on both disks so slower. Easy to recover from failure.

10.3.3 RAID Level 5 —Block-Level Distributed XOR

Stores parity for a set of blocks.

When one block in each set fails, recover the block data using the parity and the rest blocks of the set.

This is a good storage efficiency and redundancy tradeoff. Most commonly used.



10.4 Memory Caching

Memory caching can be sued to improve performance of disk devices.

10.4.1 Algorithms for Caching

Similarly to paging, LRU, LFU can be adapted.

For disks, we can do a **Frequency-Based Replacement** for **LRU**:

Divide LRU stack into two sections: new and old.
When block from the new section referenced, move to top of stack(new section).
When block from the old section referenced, move to top of stack(new section) and increment reference count.
Replace block with the lowest count in old section.

11 File Systems

11.1 Space Allocation

Dynamic space management methods used for file allocation.

11.1.1 Continuous File Allocation

Place files at continuous addresses on storages.
Advantages:

- Successive logical records typically physically adjacent

Disadvantages:

- External Fragmentation
- Poor performance if files grow beyond size allocated.

11.1.2 Block Linkage

We can instead store files into blocks.
Blocks are of same size, and contains pointer to next block, called chaining (linked list).
Large block sizes can result in significant internal fragmentation. Small block sizes can cause data spread across multiple blocks —poor performance.

11.1.3 Block Allocation Table

Block Allocation Table stores pointers to all file blocks.
Uses a **User Directory** to store mappings of file descriptors and its location in the Block Allocation Table.
Directory entry indicates first block of file. Successive blocks are located using *Block Linkage*.

11.1.4 Index Blocks

To improve performance of random accessing the file, we use Index Blocks for each file.
Index Blocks contains list of pointers that point to file data blocks. So we can access each block in constant time, compared to $O(n)$ access time for the Block Linkage(like a linked list).

11.2 Inodes

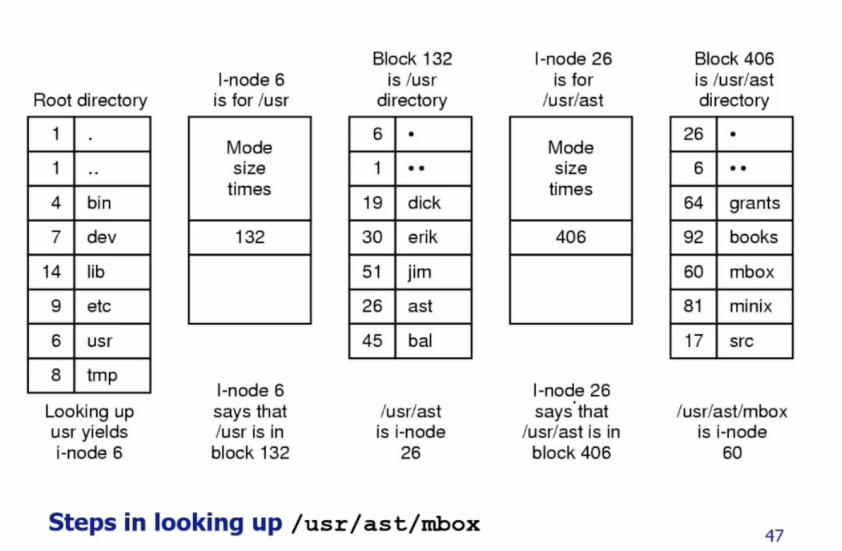
Index Blocks are called **inode**s on Unix. open system call opens the corresponding **inode**.
Inode contains Direct Pointers, Single Indirect Pointers, Double Indirect Pointers, Triple Indirect Pointers, ...

11.2.1 Maximum Data Referenced by Pointers

Direct Pointers directly points to data blocks. n direct pointers can store $n \times$ block size data.
Indirect Pointers point to other Direct Pointers. Single Indirect Pointers contains reference to one Direct Pointer; Double contains two, ...
One Double Indirect Pointer can store $\left(\frac{\text{block size}}{\text{pointer size}}\right)^2$ references to direct pointers, so $\left(\frac{\text{block size}}{\text{pointer size}}\right)^2$ block size data.

12 File Directory Organisation

Operating Systems can maintain hierarchical file system, which allows nested directories and files.
Directories are entries in the file system.



12.1 Links

Link is reference to directory/file in another part of file system.

Hard link references address of file(inode in Unix).

Symbolic link reference full pathname of directory/file, created as directory entry.

When deleting the referenced file, hard link prevents deletion(via reference count) while symbolic link do s not do so. After deletion symbolic link will reference a non-existing file.

12.2 Mounting

Mount combined multiple FSs into one *namespace*, allowing references from single root directory. Soft-links are supported in mounted FSs but not hard-links(because inodes are different in different FSs).
A **mount point** is a directory in native FS assigned to root of mounted FS.
Mounted directories are managed with mount tables.

12.3 Linux ext2fs

User can choose block sizes from 1024 to 8192 bytes.
5% of blocks are reserved for root for safety, to allow root processes to run even if user disk is full.

12.3.1 ext2 inode

A **ext2 inode** has 12 data block pointers, 1 indirect pointer, 1 doubly-indirect pointer, 1 triply-indirect pointer.

12.3.2 Block groups

Contiguous data blocks are clustered as **block group**s. FS attempts to store related data in same block group to improve performance.
A block group consists of:

Superblock Critical data about entire FS including total number of blocks and inodes, size of block groups, time FS was mounted, …There are redundant copies in same groups.

Inode table Inode entries

Inode allocation bitmap Inodes used

Block allocation bitmaps

Group descriptor block numbers for location of the above tables and maps, since the tables and maps are also stored as blocks.

Data blocks Actual data blocks

13 Security

Security Policy specifies what security is provided, i.e. who/what is protected.

Security Mechanism specifies hot to implement the security.

Authentication Verification of identity of principal.

Authorisation

13.1 Unix SUID

In Unix filesystems, permission bit SETUID indicates:

- Effective UID of the file switched to file owner when executed
- Increases privileges when using system programs.

Example: −rwsr−xr−x

13.2 Access Control Matrix

Access Control Matrix specifies authorisation policy —which principal can access which objects.

	Object 1	Object 2	Object 3	Object 4	Object 5
Principal 1	read		read		read
Principal 2		execute		read, print	

13.2.1 Access Control List

Access Control List stores with each object:

- the principals that can access it;
- the operations each principal can perform on it.

It can be displayed on the Access Control Matrix as columns.

13.3 Capabilities

Possession of capability gives right to perform operations specified by it. Capabilities can be displayed on the Access Control Matrix as rows.
Capabilities are managed by OS. Users can get a reference to capabilities and can request which capabilities they need.

13.4 Access Control Mechanisms

Discretionary Access Control Principals determine who may access their objects

Mandatory Access Control Precise system rules determine access to objects

13.4.1 Bell-La Padula Model

Objects and principals have assigned security levels.
Two rules implemented:

The Simple Secuirity Property A process can read only objects at its level or lower.

The * Property A process can write only objects at its level or higher.

Bell-La Padula Model ensures data confidentiality —no information can leak from a higher level to lower level. But it does not provide data integrity —all data at lower security level can be accessed by processes at higher level.

13.4.2 Biba Model

Biba Model inverts two rules in Bell−La Padula Model to achieve data integrity but lose data confidentiality.

13.5 Design Principals for Security

- Give each process the least privilege possible. Default to no access.
- Protection mechanism should be simple and uniform.
- Scheme should be physiologically acceptable.
- System design should be public.