Summary Operating Systems

Marie Simon s1023848

October 30, 2019

General

Conditions to operating systems

- Reliability
- Performance
- Security
- Energy Efficiency

It consists of a program called **kernel**. This manages the execution of applications and connect applications with hardware.

- Scheduler (switch between processes)
- Memory management (translation of memory addresses, keeping track of usage...)
- File system (how files and directories are stored on disk)
- drivers (software that knows how to manage hardware components)

It also has elevated permissions

User mode:
Restrictions on access to resources and memory.
Only a subset of hardware instructions are allowed.

Kernel mode: Supervisor/system/control mode
Full access to resources and memory
All hardware instrcutions are allowed

These modes guarantee that if a program crashes not the whole system crashes (was not the case in the past or on micro-controllers)

Single-Processor System

One main CPU that is capable of executing a general-purpose instruction set, including instructions from user processes. Almost all single-processor systems have other special-purpose processors as well.

Multiprocessor System

Increased throughput , more work done in less time. The speed-up ratio with N processors is not N, however; rather, it is less than N because of a certain overhead to keep all parts working correctly

Economy of scale Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies

Increased reliability If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. Also called *graceful degradation.*

Asymmetric multiprocessing

In an asymmetric multiprocessing the scheme is defined by a boss-worker relationship. One boss processor controls the system while the others wait for an instruction or have a predefined task.

Symmetric multiprocessing

In a symmetric multiprocessing system one processor performs all tasks within the operating system, so processors can be described to be peers.

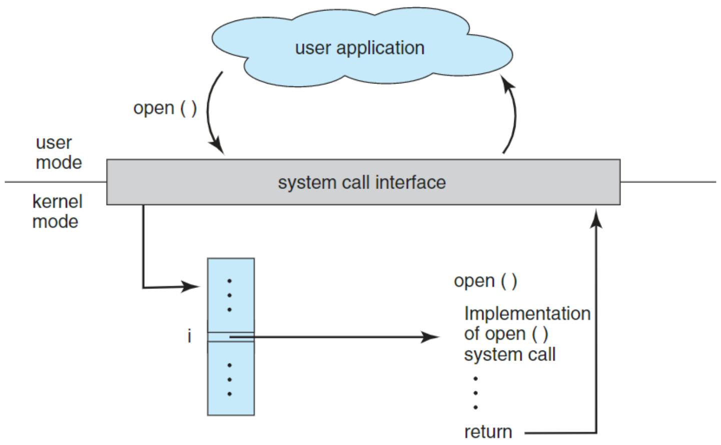
Interrupts and traps

Interrupts are meant to signal an event from hardware or software so the CPU stops whatever it was doing before and starts the process and provide whatever service was required. Once that process is completed the CPU returns to the interrupted computations. A trap is a software-generated interrupt (an exception) caused by an error or a specific request from a user program. A user program can generate a trap intentionally because a trap does not influence other hardware interrupts. This way an error only influences the one program it occurs in and no other processes

System calls

A system call is a way for programs to interact with the operating system. It provides an interface between a process and the operating system to allow user-level processes to request services of the operating system. **System calls are the only points into the kernel system**. Used for

- Process control
- File manipulation
- Device management
- Information maintenance
- Communication



Implementation

- Typically a number is associated with each system call
- System-call interface maintains a table indexed according to these
- System-call interface invokes the intended system call in the OS kernel and returns status and any return-values

- Caller does not need to know anything about how the system call is implemented, just needs to obey API and understand what the OS will do
- Most details of OS interface are hidden from the programmer by the API

Processes

A computer is program is a passive collection of instructions, but a process is the actual execution of those. Each CPU(core) executes a single task at a time. A process is a task that can be assigned and executed on a specific processor. Without processor no process. Following properties:

- A chunk of hardware instructions
- current program state
- Assigned system resources

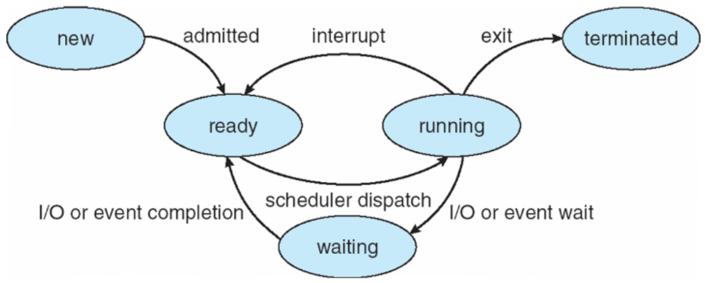
Important condition: Processes are independent of each other, they are isolated. They cannot influence each other directly without the consent if the other process.
Consequence: Every process has their own isolated piece of memory (which is hardware enforced)

Context of processes include

- Process state (running, waiting, etc.)
- Program counter (location of instruction to next execute)
- CPU registers (contents of all process-centric registers)
- CPU-scheduling information (priorities, scheduling queue pointers)
- Memory management information (memory allocated to the process)
- Accounting information (CPU used, clock times elapsed since start, time limits)
- I/O status information (I/O devices allocated to process, list of open files)

This context is stored in a process control block (PCB). This contains all the information to pause and restart a process without the process itself noticing it.

Diagram of process state



Creation of processes

Processes can create new processes (`fork()`). The creating process is then called a parent-process. Each of these new processes can create other processes, forming a tree of processes. This needs to be managed by the operating system because of the isolation property
Most operating systems identify processes according to a unique **process identifier/pid** which is typically an integer number.
A child process needs certain resources (CPU times, memory, files, I/O devices). It can obtain those resources directly from the OS or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children or it may be able to share some resources (such as memory or files)

Context switches

- Switching the running process in a processor/core implies switching the process context (PCB)
- Costs a lot of time (performance)
- Has to store the whole context of the old process and restore the whole context of the new process
- Will probably also be bad for caching your CPU
- When a context-switch occurs the kernel saves the context of the old process in its PCB and loads the saved context of the new process to run.

Multithreaded programming

- Most modern applications are multithreaded
- Threads run within application
- Process creation is heavy weight, thread creation is lightweight (reduced overhead)
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Benefits:

Responsiveness May allow continued execution if part of process is blocked, especially important for user interfaces

Resource sharing Threads share resources of processes, easier than shared memory or message passing

Economy Cheaper than process creation because allocating memory and resources is costly, thread switching has a lower overhead than context switching

Scalability Process can take advantage if multiprocessor architectures

Challenges:

Identifying tasks Examine applications where areas can be divided into seperate concurrent tasks

Balance Tasks may not be of equal value

Data splitting Data must be divided to run on separate cores

Testing and debugging Many possible execution paths make testing and debugging more difficult (verification and testing of concurrent systems is an active research area)

User Threads vs Kernel Threads

User Threads Management done by user-level threads library, above the kernel

Kernel threads Supported by the kernel, managed directly by the operating system

Threading issues

Semantics of `fork()` and `exec()`

`exec()` is a functionality of an operating system that runs an executable file in the context of an already existing process, replacing the previous executable

`fork()` is an operation wherby a process creates a copy of itself. It is usually a system call, implemented in the kernel. Fork is the primary (and historically only) method of process creation on Unix-like operating systems.

Does `fork()` duplicate only the calling thread or all threads? Some Unix have two versions of `fork()`

Signal handling

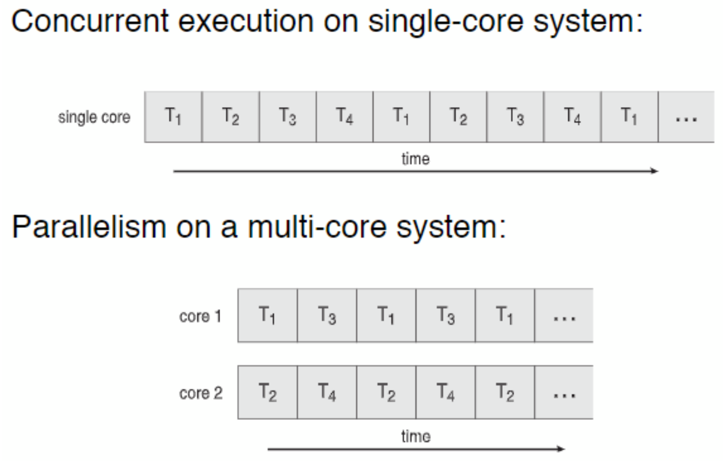
For single-threaded programs a signal simply gets delivered to process. For multi-threaded program there are multiple options:

- Deliver signal to the thread to which is applies (synchronous signals)
- Deliver the signal to every thread in the process (asynchronous signals such as process termination)
- Deliver a singal to certain threads in the process
- Assign a specific thread to receive all signals for the process

Concurrency vs Parallelism

- Parallelism implies a system can perform more than one task simultaneously
- Concurrency supports more than one task making progress

  Single processor/core, scheduler providing concurrency


Concurrent execution on single-core system:
single core
T_1 T_2 T_3 T_4 T_1 T_2 T_3 T_4 T_1 …
time
Parallelism on a multi-core system:
core 1
T_1 T_3 T_1 T_3 T_1 …
core 2
T_2 T_4 T_2 T_4 T_2 …
time

Amdahl's law is a formula that identifies potential performance from adding additional computing cores to an application that both serial (non-parallel) and parallel components. If S is the portion of the application that must be performed serially on a system with N processing cores, the formula appears as follows:

$$speedup \leq \frac{1}{s + \frac{(1-S)}{N}}$$

Types of parallelism

Data parallelism

- Distributing subsets of the same data across multiple computing cores
- performing the same operation on each core

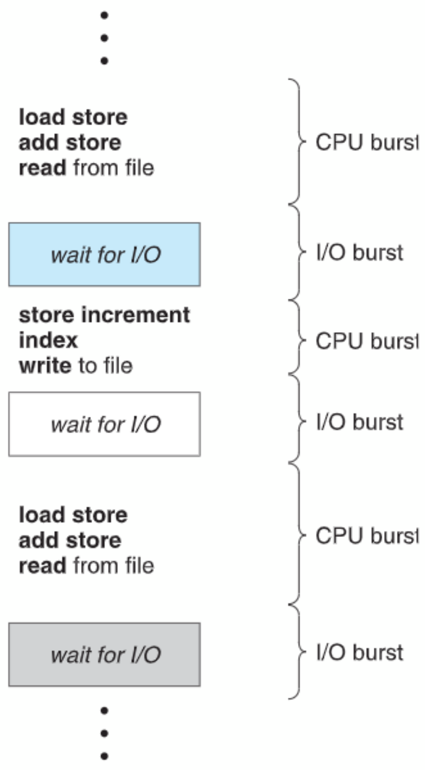
Task parallelism

- Distributing threads across cores, each thread performing a unique operation

Scheduling

CPU-I/O Burst Cycle

Process execution consists of a cycle of CPU execution and I/O wait. CPU burst is followed by I/O burst



CPU scheduler

Also called short-term scheduler. Whenever CPU is idle it selects a process in the ready queue and allocates it to the CPU. Scheduling may happen when

1. When process switches from running to waiting state
2. When a process switches from running state to ready state
3. When a process switches from waiting to ready state
4. When a process terminates

Nonpreemptive scheduling

When scheduling only occurs under circumstance 1 and 4 i.e. process keeps CPU until terminating or waiting

Preemptive scheduling

- Can result in race conditions when data are shared among several processes
- If kernel is busy on behalf of a process and this process gets preempted in the middle of these changes and kernel needs to read or modify the same structure → chaos
- Interrupts occurring during crucial OS activities

Dispatcher

Gives control of CPU to the process that got selected by scheduler

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart the program

Dispatch latency is the time it takes to stop one process and start another running

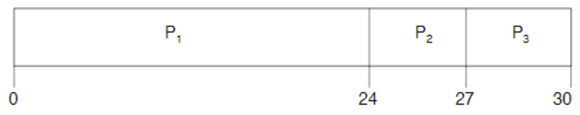
Criteria

- Max. CPU utilization (keep it as busy as possible)
- Max. Throughput (number of processes completed per time unit)
- Min. Turnaround time (amount of time to execute a particular process)
  Waiting to get into memory + waiting in ready queue + CPU execution + I/O
- Min. Waiting time (amount of time in ready queue)
- Min. Response time (from submission of request to first response)

First-Come-First-Served

Process	Burst Time
P_1	24
P_2	3
P_3	3

Gantt Chart, Order P_1, P_2, P_3



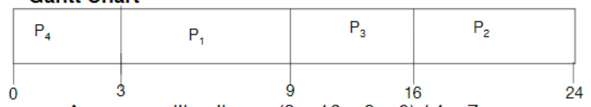
- Waiting times: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

Convoy effect: short process stuck behind long process

Shortest-Job-First

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Gantt Chart

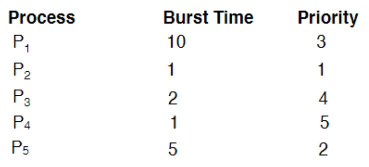


- Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$

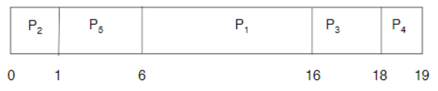
Minimum average waiting time for set of processes. Would be optimal but estimating the length is a problem. Can be done by using the length of the previous CPU bursts and exponential averaging.

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define : $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

Priority scheduling

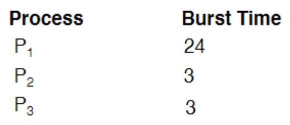


Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

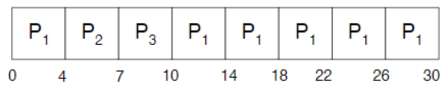


Problem: Starvation, a process with low priority may never get executed.
Solution: Aging, priority of a process increases over time

Round-Robin-Scheduling



Process	Burst Time
P_1	24
P_2	3
P_3	3



- Each process get a **time quantum** q
- Ready queue is a FIFO queue
- Once q elapsed, process is preempted and added to the end of the ready queue
- Timer interrupt every quantum to schedule next process
- Performance:

  q large $\rightarrow$ FIFO queue

  q small $\rightarrow$ q should be large compared to context switch time, otherwise overhead is too high

Multilevel Queue Scheduling

- Partition of processes in groups/separate queues with different response time requirements

  Foreground (Round robin since better response time)

  Background (First come first serve since less context switches)

- Scheduling between the queues

  Fixed priority scheduling (first foreground then background, but starvation)

  Time slice, each queue gets a certain amount of CPU time which it can schedule among its processes

Synchronization

- Processes may execute concurrently

  May be interrupted at any time, only partially completing execution

- Concurrent access to shared data may result in data inconsistency
- Maintaning data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Producer-consumer problem

- Producer produces information that is consumed by consumer
- Using shared memory for information
- (Bounded) buffer of items that can be filled by producer and emptied by consumer
- Access must be synchronized
- Introduce an integer counter that keeps track of the number of items in the buffer

Due to concurrent execution variable gets accessed at the same time and thus we can get to wrong results → **Race Condition**

Critical Section Problem

Critical section segment of code:

- Common variables, updating table, writing file ...
- When one process is in its critical section no other may be in its critical section
- Each process must ask permission to enter critical section in *entry section*, may follow critical section with *exit section* and then *remainder section*

Solutions

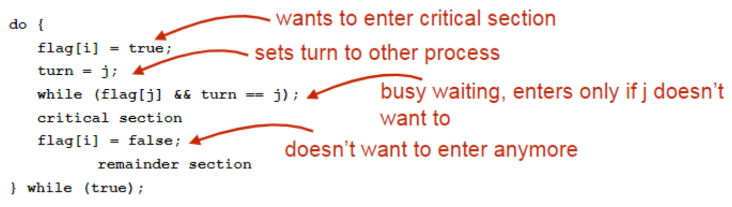
Mutual Exclusion If process P is in its ciritical section not other process can be in its ciritical section

Progress If no process is executing its critical section and some processes wish to enter their critical section, then the selection of the process that is allowed to enter next cannot be postponed indefinitely

Bounded waiting A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Peterson's Solution

Solution for two processes by sharing two variables
`int turn` indicates whose turn it is to enter the critical section
`Boolean flag[2]` arrray is used to indicate if a process is ready to enter the critical section (`flag[i]=true` implies P_i is ready)



```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    critical section
    flag[i] = false;
            remainder section
} while (true);
```

wants to enter critical section
sets turn to other process
busy waiting, enters only if j doesn't want to
doesn't want to enter anymore

Mutex Locks

- Protect critical regions by first `acquire()` a lock and then `release()` it. A boolean variable indicates of lock is available or not
- Calls must be atomic, usually implemented via hardware atomic instructions
- Requires **busy waiting**. While one process is in critical section, other process must loop continuously, therefore also called **spinlock**
- Wastes CPU cycles while spinning, but no context switch required
- One thread can spin on one processor while another thread performs critical section

Semaphores

- Semaphore S is integer variable (number of resources available)
- Two standard operations modify S

```
wait(S){
    while(S <= 0)
       ;//busy wait
    S--;
}
```

```
signal(S){
    S++;
}
```

- **Counting semaphore**, integer value can range over unrestricted domain
- **Binary semaphore** integer value can range between 0 and 1, same as Mutex lock

Deadlocks and Starvation

Deadlock Two or more processes are waiting indefinitely for an event that can be caused by only one of the two waiting processes.

Starvation Indefinite blocking, a process may never be removed from the semaphore quee in which it is suspended

Conditions and preventions

Mutual exclusion: only one process at a time can use a resource

Sharable resources like read-only files do not require mutual exclusion

Hold and wait: A process holding at least one resource is waiting to acquire additional resources held by other processes

Guarantee that whenever a process requests a resource it does not hold any other resources

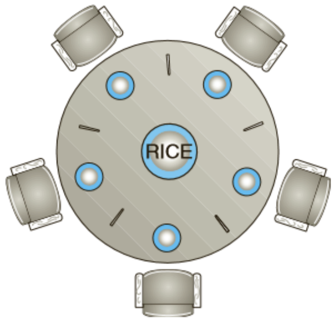
No preemption: A resource can be released only voluntarily by the process holding it, after the process has completed its task

If a process requests a resource that cannot be allocated, all its currently held resources are being released (its preempted)

Circular wait: There exists a set of processes $\{P_0, P_1, \ldots, P_n\}$ such that P_0 is waiting for a resource that is held by P_1. P_1 is waiting for a resource held by P_2 and so on. P_n is waiting for a resource held by P_0

Impose a total ordering of all resource types and require that each process requests resources in an increasing order of enumeration

Dining-philosophers

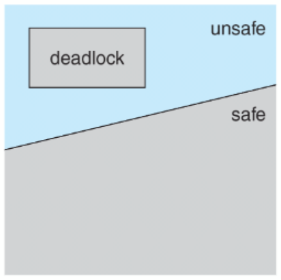


If all philosophers get hungry at the same time and pick up the left chopstick at the same time, all of them will be delayed forever trying to grab the right one. Possible solutions:

- Allow at most four philosophers to be eating simultaneously
- Allow a philosopher only to pick up chopsticks if both are available (thus picking up must be in the critical section)
- Use an asymmetric solution (all even philosophers take left first, all odd take right first)

Safe state

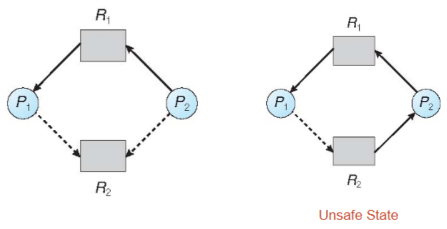
- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence of all processes such that for each process the resources that it can request can be satisfied by currently available resources and resources held by all processes before it.



Ressource-Allocation Graph

1. Claim edge indicates that a process may request a resource (dashed line)

   Must be claimed a priori in the system
2. Claim edge converts to request edge when a process requests a resource
3. Request edge converts to an assignment edge when the resource is allocated to process
4. When a resource is released, assignment edge converts back to claim edge



Requests can only be granted if converting the request edge to an assignment edge does not result in a cycle

Banker's Algorithm

- Applicable for a resource-allocation system with multiple instances of each resource type
- Each process must a-priori claim its maximum use which may not exceed the total number of resources in the system
- When a process requests a resource it may have to wait
- When a process gets all resources it must return them in a finite amount of time
- Data structures where n is the number of processes in the system and m is the number of resource types

  Available A vector of length m indicates the number of available resources of each type. If `available[j]=` k then k instances of resource type R_j are available

  Max An $n \times m$ matrix defines the maximum demand of each process of each resource type

  Allocation An $n \times m$ matrix defines the number of resources of each type currently allocated to that process

  Need An $n \times m$ matrix indicates the remaining resource need of each process.

  Each of the matrices can be converted to a vector on one particular process.

Safety algorithm

1. Let `Work` and `Finish` be vectors of length m and n, respectively. Initialize:
   `Work = Available`
   `Finish[i]=false for i = , 1,...,n-1`
2. Find and index `i` such that both
   a) `Finish[i] = false`
   b) `Need`$_i$ $\leq$ `Work`

   If no such `i` exists, go to step 4
3. `Work = Work + Allocation`$_i$
   `Finish[i] = true`
   Go to step 2
4. If `Finsih[i] == true` for all `i`, then the system is in a safe state

Resource-request Algorithm Added data structure

Request A vector for process P_i. If `Request`$_i$`[j]==k` then P_i wants k instances of resource R_j.

1. If `Request`$_i$ $\leq$`Need`$_i$, go to step 2. Otehrwise raise an error condition since process has exceeded its maximum claim
2. If `Request`$_i$ $\leq$`Available`$_i$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Have the system pretend to have allocated the requested resources to process P_i by modifying state as follows:

```
Available = Available - Request;
Allocation_i = Allocation_i + Request_i;
Need_i = Need_i - Request_i;
```

   If resulting resource-allocation is safe, transaction is completed and process P_i is allocated to its resources. If it is unsafe, the process must wait for `Request`$_i$ and the old resource-allocation state is restored

Memory

- To execute a process, the process must be in memory
- Main memory and registers are only storage CPU can access directly

  Register can be accessed in ¡1 CPU clock

  Main memory can take many cycles, causing a stall
-

Logical vs Physical Address Space

Logical address generated by CPU, also called virtual address

Physical address Address seen be the memory unit

- Logical address space that is bound to physical address space is central to proper memory management
- Same in compile-time and load-time address binding schemes
- Differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware that maps virtual addresses to physical ones at run time
- Base register now called **relocation register**
- Makes **dynamic loading** possible
  - Not entire program is loaded frór execution, routine only gets loaded once its called
  - Unused routine never loaded → better memory space utilization
  - All routines kept on disk in relocatable load format

Swapping

- Problem: We want to run more processes than fit in memory
- Solution: When process is sleeping write to **backing store** and bring back to memory for continued execution
- **Backing store** fast disk large enough to accommodate copies of all memory images for all users, must provide direct access to these
- To use swapping OS must allocate space for processes → Dynamic storage allocation

Dynamic storage allocation problem

- Causes external fragmentation

  Lots of free space, but only available in small sized holes

  First-fit Allocate the first hole that is big enough

  Best-fit Allocate the smallest hole that is big enough

  Worst-fit Allocate the largest hole

Compaction Shuffle memory contents to place all free memory together in one large block (slow and must halt all processes)

Segmentation Divide process memory in small logical chunks, different segments can then be stored at different places

- Better utilization of small holes, so external fragmentation reduced but not solved
- Internal fragmentation (allocated memory may be larger than requested memory, since its parted in units. Thus small unused memory that is internal to a partition)

Paging
- Divide physical memory into fixed block sizes called frames
- Logical into same size called pages
- To run a programm of N pages, find N free frames and load it
- Set up page table to translate logical to physical addresses
- Eliminated external fragmentation but still leaves internal fragmentation

File Systems and I/O Systems

Different file types. Each have a bunch of attributes and a bunch of operations to operate on them.

Open file locking

Shared lock Several processes can acquire it concurrently (like reader lock)

Exclusive lock Only one process at a time (like writer lock)

Mandatory file-locking Operating system ensures locking integrity

Advisory file-lockin It is up to software developer to ensure locks are appropriately acquired and released

File Access Methods

Sequential access One record after the other.
`read_next()` reads next portion of file
`write_next()` appends to the end of the file
Such a file can be reset and sometimes skip forward n records, perhaps only $n = 1$

Direct access Fixed length of logical records and reading and writing can happen in no particular order
`read n` reads block n
`write n`, `position to n`, `rewrite n`

Blocking and Non-blocking I/O

Blocking
- Process suspended until I/O completed
- Easy to use, easy to code and easy to understand
- Sometimes inefficient

Nonblocking I/O call returns as much as available
- Used for user interface
- Implemented via multithreading
- Returns quickly the read or written bytes
- `select()` to find if data is ready and then uses `read()` `write()` for transfer

Asynchronous I/O
- Returns immediately without waiting for I/O to complete
- I/O system signals when ready
- Difficult to use

Protection and Security

A computer consists of a collection of hardware and software objects. Protections means we need to ensure that each object is accessed correctly and only by processes that are allowed to do so

Least privilege

- Programs, users, systems should have just enough privileges to perform their tasks
- Limits damage if entity has a bug or gets abused
- Static: during life time or life of process
- Dynamic: Changes as needed (domain switching, privilege escalation)

Protection domain-Domain structure

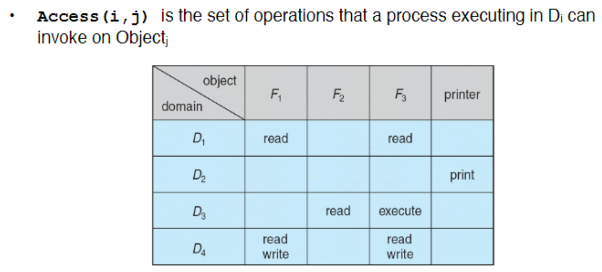
A protection domain specifies the resources a process my access

Access-right `<object-name, rights-set>`
where `rights-set` is a subset of all valid operations that can be performed on object

Domain Set of access rights

Domain switching Dynamically enable a process to switch from one domain to another

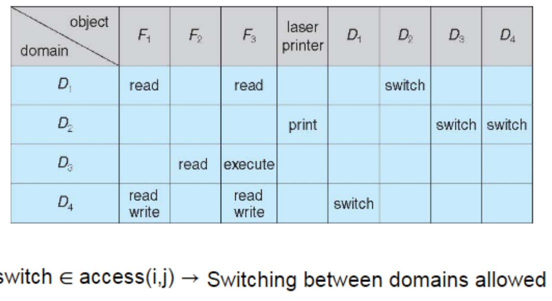
Access Matrix



- `Access(i,j)` is the set of operations that a process executing in D_i can invoke on $Object_j$

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Copy rights are denoted by `*`. It allows to copy right within the same column.



domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

switch $\in$ access(i,j) $\rightarrow$ Switching between domains allowed

Owners are allowed to add and remove rights, denoted by "owner".