Operating System

Basic

What Operating Systems Do?

- User View
  - PC users want **convenience**, ease of use and good performance, don't case about resource utilization.
  - Users of shared computer such as mainframe or minicomputer want **maximum resource utilization**.
  - Users of dedicate systems such as workstations want **trade-off** between individual usability and resource utilization because they have dedicated resources at their disposal, but they also share resources such as networking and servers.
  - Mobile computers are resource poor, optimized for usability and battery life.

        - Some computers such as embedded computers in devices and automobiles have little or no user interface.
- System View
    - **Resource allocator**
        - Manages all resources
        - Decides between conflicting requests for efficient and fair resource use
    - **Control program**
        - Controls execution of programs to prevent errors and improper use of the computer

Computer-System Operation

1. **Bootstrap** is loaded at power-up or reboot
    - Stored in firmware, such as ROM or EPROM
    - Initializes all aspects of system
    - Loads OS kernel into memory and starts execution
2. System processes or system daemons
    - Loaded to memory at boot time
    - Run the entire time the kernel is running
    - "init"
3. **Interrupt** can be triggered by hardware and software
    - Hardware sends signal to CPU
    - Software executes a special operation: system call
4. Interrupt procedure
    1. CPU stops what is doing
    2. Execute the service routine for the interrupt
    3. CPU resumes

Storage Structure

- **Cache**
    - Caching copies information into faster storage system
    - Main memory can be viewed as a cache for secondary storage
    - Faster storage (cache) checked first to determine if information is there
        - If it is, information used directly from the cache (fast)
        - If not data copied to cache and used there
    - Cache is smaller than storage being cached
        - Important design problem of cache management: size and replacement policy
    - Cache is an important principle, performed at many levels in a computer (hardware, operating system, software)

Multiprogramming and Multitasking

- **Multiprogramming**:
    - A single program cannot keep CPU and I/O devices busy at all times. Single users frequently have multiple programs running.
    - Increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.
    - Provides an environment in which the various system resources are utilized effectively.
- **Multitasking**(**Time sharing**):
    - A logical extension of multiprogramming.

        - CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Dual-mode Operation

- Allows OS to protect itself and other system components
- **User mode** and **kernel mode**
- **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some of the machine instructions that may cause harm designated as privileged, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user
- Transition from user to kernel mode
    - At system boot time, the hardware starts in kernel mode
    - OS is loaded and starts user application in user mode
    - Interrupt occurs, the hardware switches from user mode to kernel mode
    - Whenever the OS gains control, it is in kernel mode
- Why dual-mode?
    - The dual mode of operation provides **the means for protecting the operating system from errant users—and errant users from one another**
    - The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

OS Services

- For convenience of the user (6)
    - User interface
        - command-line interface (CLI)
        - batch interface
        - graphical user interface (GUI)
    - Program execution
    - I/O operations
    - File-system manipulation
    - Communications (2 models of inter-process communication)
        - shared memory
        - message passing
    - Error detection
- For ensuring efficiency of system (3)
    - Resource allocation
    - Accounting
    - Protection and security

System Calls

- **Programming interface to the services between processes and the kernel** provided by the OS
- Provide the means for a user program to ask the operating system to perform tasks
- Treated by the hardware as a software interrupt
- Are usually

    - Primitive
    - Important
    - Fundamental
- Different from **library calls** (function calls)
    - Library functions are usually compiled and packed inside an object called the library file
    - System calls are primitive and not programmer-friendly
- How to use?

  Mostly accessed by programs via a high-level **API** rather than direct system call use

    - **Application Programming Interface** (API): a set of functions that are available to application programmers
- Implementation
    - **System call interface** invokes system call, provided by the run-time support system, which is a set of functions built into libraries within a compiler
    - Typically, a number associated with each system call
    - System-call interface maintains a table indexed according to the numbers
    - Benefits
        - The caller needs to know nothing about
            - how the system call is implemented
            - what it does during execution
        - Just needs to obey API and understand what OS will do as a result call
        - Most details of OS interface are hidden from programmer by API managed by run-time support library
- Types
    - Process control
    - File manipulation
    - Device manipulation
    - Information maintenance
    - Communications
    - Protection

OS Design and Implementation

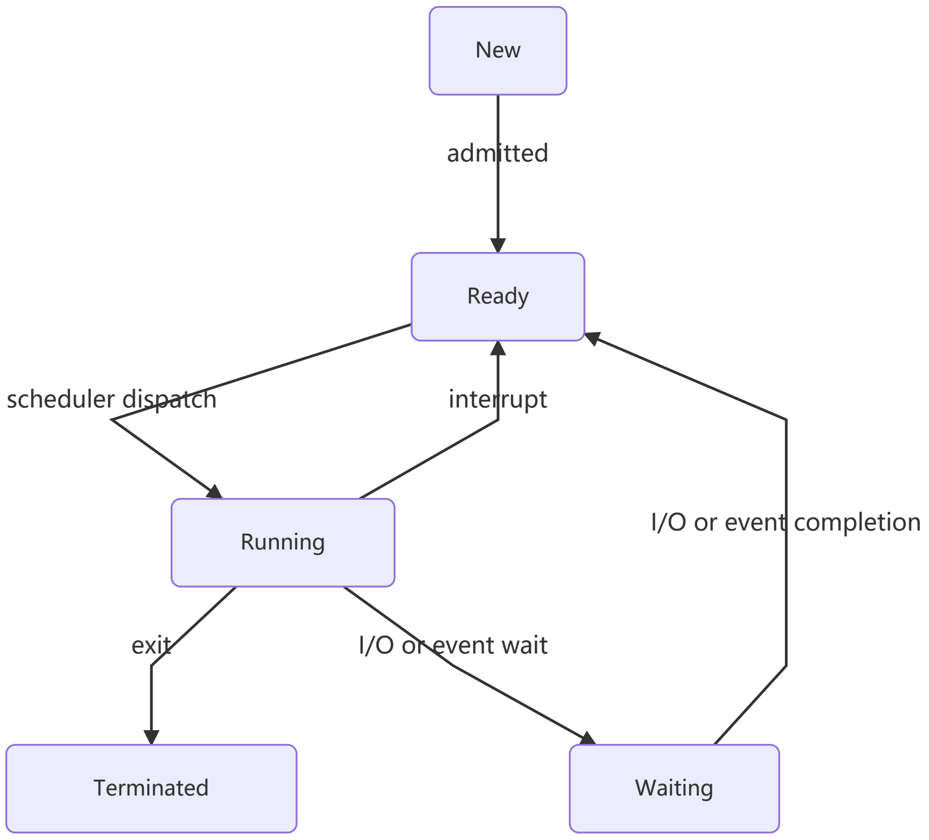
- One important principle is the separation of **policy** from **mechanism**
    - Mechanisms determine *how* to do something
    - Policies determine *what* will be done
    - The separation is important for **maximum flexibility**
        - Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism
        - A general mechanism insensitive to changes in policy would be more desirable
        - A change in policy would then require redefinition of only certain parameters of the system

Process

Concepts

- A **program** is just a piece of code

    - Static
    - May be associated with dynamically-linked files
    - May be compiled from more than one file
- **A process is a program in execution**
    - A program (an executable file) becomes process when it is loaded into memory
    - Active
- Process in memory
    - **Text section**: program code
    - **Data section**: global variables
    - **Stack**: temporary data, such as function parameters, return addresses, local variables
    - **Heap**: dynamically allocated memory during process run time
    - **Program counter**
    - **Contents of registers**
- Process State
    - **New**: the process is being created
    - **Running**: instructions are being executed
    - **Waiting**: the process is waiting for some event to occur
        - I/O completion or reception of a signal
    - **Ready**: the process is waiting to be assigned to be a processor
    - **Terminated**: the process has finished execution
    - Only one process can be running on any processor at any instant
    - Many processes may be ready and waiting



- **Process control block** (**PCB**) or task control block
    - **Process state**
    - **Program counter**: location of next instruction to execute
    - **CPU registers**: contents of all process-centric registers
    - **CPU scheduling information**: priorities, scheduling queue pointers

    - **Memory-management information**: memory allocated to the process
    - **I/O status information**: I/O devices allocated to process, list of open files
    - **Accounting information**: CPU used, clock time elapsed since start, time limits
- Two processes maybe associated with the same program (Two users are running the same program)
    - The text section may be equivalent
    - The data, heap, and stack sections vary
- A process can be an execution environment for other code

Operations

- Process identification
    - Each process is given an unique ID number called the **process ID**, or the **PID**
    - `getpid()` prints the PID of the calling process
- Process creation
    - A process may create several new processes:
        - **Parent process**: the creating process
        - **Children processes**: the new processes
    - The first process "**init**"
        - Created when the kernel is booting up
        - PID=1
        - Running the program code `/sbin/init`
        - First task is to create more processes
    - **Tree hierarchy**
        - A process may become an **orphan** when its parent terminated
        - No one would know the termination of the orphan
        - In Linux, we have the **re-parent operation**: the "init" process will become the step-mother of all orphans. So the processes are always organized as a tree and one tree only
    - `fork()`
        - `fork` n次创建2^{n-1}个（子）进程

Cloned items	**Descriptions**
Program code [File & Memory]	They are sharing the same piece of code
Memory	Including local variables, global variables, and dynamically allocated memory
Open files [Kernel's internal]	If the parent has opened a file, then the child will also have the file opened automatically
Program counter [CPU register]	This's why they both execute from the same line of code after `fork()` returns

Distinct items	**Parent**	**Child**
Return value of `fork()`	PID of the child process	0
PID	Unchanged	Different, not necessarily be "`Parent_PID + 1`"
Parent process	Unchanged	Doesn't have the same parent as that of the parent process

Distinct items	**Parent**	**Child**
Running time	Cumulated	0 (Just created)

- Program execution
    - `exec()` system call family change the code that is executing and never returns to the original code
        - **Throw away**
            - Memory: local variables, global variables, and dynamically allocated memory
            - Register value: the program counter, etc
        - **Preserve**
            - PID
            - Process relationship
            - Running time
    - `fork()` + `exec*()` + `wait()` = `system()`
    - `wait()`
        - Suspends the calling parent process then returns and wakes it up when the one of its child processes changes from running to terminated
        - Does not suspend the calling process if
            - There were no running children
            - There were no children
        - Return the PID of the terminated child
    - Why calling `wait()` is important?
        - It is not about process execution / suspension...
        - It is about *system resource management*

`wait()`	**`waitpid()`**
Wait for any one of the children	Wait for a particular child only
Detect child termination only	Detect child's status changing from running to suspended or from suspended to running

- Process termination
    - `exit()`
        1. Clean up most of the allocated kernel-space memory
        2. Clean up all user-space memory
        3. Notify the parent with SIGCHLD
    - **Zombie process**
        - A process is turned into a zombie when it calls `exit()` and returns from `main()` but terminates abnormally, so that its parent does not get its state.
        - There are no program code in a zombie process which is no longer running.
        - A zombie process will be eliminated after its parent gets relative information and signal which usually implement by the `wait()` system call.

Communication

- Inter-process communication (**IPC**)
    - Used for exchanging data between processes

        - 2 Models
            - **Shared memory**: bounded / unbounded buffer
            - **Message passing**
                - Naming: direct / indirect communication
                - Synchronization: synchronous / asynchronous
                - Buffering: zero / bounded / unbounded capacity
- IPC mechanisms
    - POSIX shared memory: Associate the region of shared memory with a memory-mapped file
    - **Sockets**:
        - defined as endpoints for communication (over a network)
        - A pair of processes employ a pair of sockets
        - A socket is identified by an IP address and a port number
    - **Pipes**:
        - **Ordinary pipes**
            - No name in file system
            - Only used for parent-child related processes
            - Unidirectional (one-way communication)
            - Ceases to exist after communication has finished
        - **Named pipes**
            - pipe with name in file system
            - No parent-child relationship is necessary (but must reside on the same machine)
            - Several processes can use it for communication (may have several writers)
            - Communication is bidirectional (still half-duplex)
            - Continue to exist until it is explicitly deleted
- **Race condition**
    - several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place
    - **Mutual exclusion** is a requirement
    - Defining critical sections is a solution
- **Critical section**
    - Requirements
        - **Mutual exclusion**: No two processes could be simultaneously inside their critical sections.
        - Each process is executing at a nonzero speed, but no assumptions should be made about the relative speed of the processes and the number of CPUs.
        - **Progress**: No process running outside its critical section should block other processes.
        - **Bounded waiting**: No process would have to wait forever in order to enter its critical section.
    - Solutions
        - **Disabling interrupt**
            - Disable context switching when the process is inside the critical section
            - Implementation: A new system call should be provided
            - Not as feasible in a multiprocessor environment and may sacrifice concurrency
        - Hardware solution: Rely on atomic instructions
        - **Mutex Locks**

```
acquire()
{
    while (!available)
        ; // busy wait
    available = false;
}
```

```
release()
{
    available = true;
}
```

```
do
{
    acquire();
    /* critical section */
    release();
    /* remainder section */
} while (true);
```

        - Busy waiting: Waste CPU resource (spinlock)
    - **Strict alternation**
        - Using a new shared object to detect the status of other processes

```
/* Process 0 */
while (TRUE)
{
    while (turn != 0)
        ;
    /* critical section */
    turn = 1;
    /* remainder section */
}
```

```
/* Process 1 */
while (TRUE)
{
    while (turn != 1)
        ;
    /* critical section */
    turn = 0;
    /* remainder section */
}
```

        - Busy waiting
        - Violate the requirement of **progress**
    - **Peterson's solution**
        - Key idea: Sharing data

```
int turn;
boolean flag[2];
```

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

- Busy waiting
- **Priority inversion** problem

- **Semaphore**
  - A data type (additional shared object) accessed only through two standard atomic operations
    - `down()` / `wait()`: Decrementing the count

```c
void down(semaphore *s)
{
    while (*s == 0)
        ; // busy wait
    *s = *s - 1;
}
```

    - `up()` / `signal()`: Incrementing the count

```c
void up(semaphore *s)
{
    *s = *s + 1;
}
```

  - 2 Types
    - Binary: 0 / 1 (similar to mutex lock)
    - Counting: Control finite number of resources
  - Issue
    - Busy waiting: Block the process instead of busy waiting (place the process into a waiting queue)
    - Atomicity: Disabling interrupts
  - **Deadlock**: Two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes
    - **Starvation**: Processes wait indefinitely within the semaphore

```c
void down(semaphore *s)
{
    disable_interrupt();
    while ( *s == 0 )
    {
    enable_interrupt();
    special_sleep();
    disable_interrupt();
    }
    *s = *s - 1;
    enable_interrupt();
}
```

```c
void up(semaphore *s)
{
    disable_interrupt();
    if ( *s == 0 )
        special_wakeup();
    *s = *s + 1;
    enable_interrupt();
}
```

```c
semaphore *s;

do
{
    down(s);
    /* critical section */
    up(s);
    /* remainder section */
} while (true);
```

- **Deadlock**
  - Requirements
    - **Mutual exclusion**: Only one process at a time can use a resource
    - **Hold and wait**: A process must be holding at least one resource and waiting to acquire additional resources held by other processes
    - **No preemption**: A resource can be released only voluntarily by the process holding it after that process has completed its task
    - **Circular wait**
  - Handle
    - Prevent or avoid deadlocks: Ensure the system will never enter a deadlock state
      - **Deadlock Prevention**: Ensuring that at least one of these conditions cannot hold
        - Mutual exclusion: must hold
        - Hold and wait: Guarantee whenever a process requests a resource, it does not hold any other resources
          - Each process to request and be allocated all its resources before it begins execution
          - Allows a process to request resources only when it has none
        - No preemption: If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted
        - Circular wait: Impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
      - **Deadlock avoidance**: Requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the process should wait.
    - Detect deadlock and recover
    - Ignore the problem and pretend that deadlocks never occur (stop functioning and restart manually)
  - A deadlock-free solution does not eliminate starvation

- Classic IPC problems
    - **Producer-consumer problem** (**Bounded-buffer problem**): **Producer** process produces information that is consumed by a **consumer** process
        - Mutual exclusion: A binary semaphore is used as the entry and the exit of the critical sections.
        - Synchronization: Two semaphores are used as counters to monitor the status of the buffer.
    - **Dining philosopher problem**

```c
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

```c
void philosopher(int i)
{
    think();
    take(i);
    eat();
    put(i);
}
```

```c
void take(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

```c
void put(int i)
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
```

```c
void test(int i)
{
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING)
    {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

    - **Reader-writer problem** (Reader priority)

- Mutual exclusion: relate the readers and the writers to one semaphore
- Reader’s concurrency:
    - The first reader coming to the system `down()` the `database` semaphore.
    - The last reader leaving the system `up()` the `database` semaphore.

```
semaphore db = 1;
semaphore mutex = 1;
int read_count = 0;
```

```
void writer(void)
{
    while(TRUE)
    {
        prepare_write();
        down(&db);

        write_database();

        up(&db);
    }
}
```

```
void reader(void)
{
    while(TRUE)
    {
        down(&mutex);
        read_count++;
        if(read_count == 1) // first reader
            down(&db);
        up(&mutex);

        read_database();

        down(&mutex);
        read_count--;
        if(read_count == 0) // last reader
            up(&db);
        up(&mutex);
        process_data();
    }
}
```

- **Second Reader-writer problem** (Writer priority)

```
semaphore w = 1, r = 1;
semaphore mutex1 = 1, mutex2 = 1, mutex3 = 1;
int read_count = 0, write_count = 0;
```

```
void writer(void)
{
    while(TRUE)
    {
        prepare_write();
        down(&mutex2);
        write_count++;
        if (write_count == 1)
            down(&r);
        up(&mutex2);
        down(&w);

        write_database();

        up(&w);
        down(&mutex2);
        write_count--;
        if (write_count == 0)
            up(&r);
        up(&mutex2);
    }
}
```

```
void reader(void)
{
    while(TRUE)
    {
        down(&mutex3);
        down(&r);
        down(&mutex1);
        read_count++;
        if(read_count == 1) // first reader
            down(&w);
        up(&mutex1);
        up(&r);
        up(&mutex3);

        read_database();

        down(&mutex1);
        read_count--;
        if(read_count == 0) // last reader
            up(&w);
        up(&mutex1);

        process_data();
    }
}
```

Scheduling

- Process lifecycle:
    - All ready processes are kept on a list called **ready queue**
    - **Short-term scheduling** or **CPU scheduling**: Mainly about how to make all the processes become **running**
- **Context switching**: Actual switching procedure, from one process to another
    - The **context** of a process: User-space memory + Registers' values of the process
    - Backup all registers' values
    - Load the context of the new process into the main memory and into the CPU
    - **Minimizing the number of context switching** may help boosting system performance.
- Scheduling criteria and performance measures
    - 2 Types
        - **Non-preemptive**

            - Good for **the time in finishing tasks**
            - Bad for **user experience** and **multi-tasking**
        - **Preemptive**
            - Particular kinds of interrupts and events are detected
            - If that particular event is the **periodic clock interrupt**, then you can have a **time-sharing system**
            - Good for **inter-activeness**
    - Performance measures
        - **CPU utilization**: Keep CPU as busy as possible
        - **Throughput**: Number of processes that are completed per time unit
        - **Turnaround time**: Interval from the time of submission to the time of completion (Total running time + Waiting time + Doing I/O)
        - **Waiting time**: The time spent waiting in the ready queue
        - **Response time**: Interval from the time of submission to the first response is produced
- Scheduling algorithms
    - First-come, first-served (**FCFS**, or first-in, first-out, **FIFO**)
        - Non-preemptive
    - Shortest-job-first (**SJF**)
        - Can be non-preemptive or preemptive
        - **Minimum average waiting time**
        - **CPU burst prediction**: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
    - Round-robin (**RR**)
        - Preemptive
        - **Time quantum** or **time slice**
        - **A rule of thumb**: 80% CPU burst should be shorter than the time quantum
    - **Priority-scheduling**
    - **Priority with multi-queue**
        - **Different schedulers** may be deployed at each priority class
        - Real implementation
- Applications / Scenarios
    - Real-time scheduling
        - **Rate monotonic** scheduling
            - Assume that processes require CPU at constant periods: processing time t and period p (rate $1/p$)
            - Each process is assigned a priority proportional to its rate, and schedule processes with a **static** priority policy with **preemption**
        - Earliest deadline-first scheduling (**EDF**)
            - **Dynamically** assigns priorities according to deadline (the earlier the deadline, the higher the priority)
            - Requires the announcement of deadlines
    - Linux: Completely Fair Scheduler

Threads

Why Use Threads?

- As creating a process is heavy-weighted and resource intensive, it's more effective to use a multi-threaded process to implement the same tasks.
- Benefits
    - **Responsiveness and multitasking**: application is allowed to do parallel tasks simultaneously
    - **Ease in data sharing**: which can be done using global variables and dynamically allocated memory
    - **Economy**: allocating memory and resources for process creation is costly, dozens of times slower than creating threads as well as context-switch between processes (several times of slower)
    - **Scalability**: threads may be running in parallel on different cores

Structure in Memory

- All threads share the same code
    - A thread starts with one specific function called **thread function**
- All threads share the same global variable zone and same dynamically allocated memory
- Each thread has its own memory range for local variables, so the stack is the private zone for each stack

Concurrency or Parallelism?

- A system is parallel if it can perform more than one task simultaneously.
- A concurrent system supports more than one task by allowing all the tasks to make progress.
- It is possible to have concurrency without
  parallelism.

Models

- Associate the user thread and the kernel thread
- **Many-to-One**
    - All the threads are mapped to one process structure in the kernel
    - Merit: easy for the kernel to implement
    - Drawback: when a blocking system call is called, all the threads will be blocked
- **One-to-One**
    - Each thread is mapped to a process or a thread structure
    - Merit:
        - Calling blocking system calls only block those calling threads
        - A high degree of concurrency
    - Drawback: cannot create too many threads as it is restricted by the size of the kernel memory
- **Many-to-Many**
    - Multiple threads are mapped to multiple structures (group mapping)
    - Merit:
        - Create as many threads as necessary
        - Also have a high degree of concurrency

Programming

- Challenges
    - **Identifying tasks**: divide separate and concurrent tasks
    - **Balance**: tasks should perform equal work of equal value
    - **Data splitting**: data must be divided to run on separate cores
    - **Data dependency**: synchronization needed

    - **Testing and debugging**
- Basic programming: **Pthreads library**

```c
#include <pthread.h>
void *runner(void *param); // threads call this function
int main()
{
    pthread_t tid; // the thread identifier
    pthread_attr_t attr; // set of thread attributes
    pthread attr init(&attr); // get the default attributes
    pthread create(&tid, &attr, runner, argv[1]); // create the thread
    pthread join(tid, NULL); // wait for the thread to exit

    return 0;
}
```

- **Implicit threading**
    - Transfer the creation and management of threading from application developers to compilers and run-time libraries.
    - **Thread Pools**
    - **OpenMP**

Memory Management

User-space Memory Management

- Address space
    - **Segmentation**: Memory of a process is divided into segments
    - **Logical address space**: **Each process has its own address space**, and it can reside in any part of the **physical memory**
- Code & Constants
    - A process is not bounded to one program code
    - Constants are stored in code segment
    - Codes and constants are both **read-only**
- **Data segment** & Block started by symbol (**BSS**)
    - A **static variable** is treated as the same as a **global variable** and Only the compiler cares about the difference

Difference	Data segment	BSS
Property	Containing *initialized* global and static variables	Containing *uninitialized* global and static variables
Location	Front	After
Size	Has the required space *already allocated*	Is just a bunch of symbols. The space is *not yet allocated* to the process which will be allocated once it starts executing

    - Everything in a computer has a limit: **On a 32-bit Linux system, the user-space addressing space is around 3GB**
- Stack: **Local variables**

- Heap: The dynamically **allocated memory**
    - Dynamic: Not defined at compile time
    - Allocation: Only when asked for, the memory would be allocated
    - **External fragmentation**: There is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes.
- **Segmentation fault**: Access a piece of memory that is not allowed to be accessed

Virtual Memory: CPU + MMU

- Dynamic allocation method
    - **First fit**: allocate the first hole that is big enough (fast)
    - **Best fit**: allocate the smallest hole that is big enough
    - **Worst fit**: allocate the largest hole
- Memory-Management Unit (**MMU**): an **on-CPU device**
    - Merit
        - Different processes use the **same virtual addresses**, they may be translated to **different physical addresses**
            - The address translation helps the CPU to retrieve data in a **non-contiguous layout** (the process address space is contiguous)
        - **Memory sharing** can be implemented
        - **Memory growth** can be implemented
    - Implementation: **Paging**
        - **Pages**: Partitions the memory into fixed blocks
        - **Page table**: The lookup table inside the MMU
        - **Page number** + **Page offset**
        - Translation look-aside Buffer (**TLB**)
            - Associative, high-speed memory
            - Each entry consists of two ports, a key (tag) and a value
            - When the associative memory is presented with an item, it is compared with all keys simultaneously. If the item is found, the corresponding value field is returned which makes the search fast
    - **Internal fragmentation**: Space is avoidably wasted when allocation is done in a page-by-page manner.
    - **Demand paging**: A strategy that loads pages only when they are needed
        - **Page fault**: The interrupt generated by the CPU when the MMU finds that **a virtual page involved is invalid**
        - $t_{\text{effective access}} = \left(1 - p_{\text{page fault}}\right) \times t_{\text{memory access}} + p_{\text{page fault}} \times t_{\text{page fault}}$
        - **Swap area**: A space reserved in a permanent storage device
            - Should be at least the same as the size of the physical memory
        - **Copy-on-write** (**COW**)
            - Allows the parent and the child processes to **share pages** after the `fork()` system call is invoked.
            - A new, separated page will be **copied and modified** only when one of the processes wants to write on a shared page.
            - Useful to map the corresponding pages into the virtual address spaces of the 2 programs in a write-protected manner
        - **Page replacement**

- **Belady's anomaly**: for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.
- **Stack algorithms**: never exhibit Belady's anomaly e.g. LRU. The set of pages in memory for n frames is always a **subset** of the set of pages in memory for $n+1$ frames.
- **Thrashing**: The high paging activity that appears when a process does not have the number of frames it needs to support pages in active use which will result in page fault and frequent page replacement
- **Optimal**
  - The first page request will cause a page fault
  - To replace the page that will not be used for the longest period of time
- First-in, first-out (**FIFO**)
  - The first page being swapped into the frames will be the first page being swapped out.
  - Choose the oldest page to be the victim
- Least recently used (**LRU**)
  - When a page is just accessed, no matter that page is originally on a frame or not, set its age to be 0. Other frames' ages are incremented by 1.
- 2_{nd}**-chance algorithm**
  - Give the page a second chance if its reference bit is on
  - If a page is heavily used, its reference bit will be very likely to be on
  - Clock is the efficient implementation of the 2_{nd} chance algorithm (**circular queue**)

Mass Storage

Disk Structure

- **Track**
- **Sector**
- **Cylinder**

Disk Scheduling

- First-come, first-served (**FCFS**)
- Shortest-seek-time-first (**SSTF**): Choose the request closest to the current head position
- **SCAN** (**Elevator algorithm**): Starts at one end, moves toward the other end and reverses
- **LOOK**: Goes only as far as the final request
- **C-SCAN**: Circular scan back and forth
- **C-LOOK**

Solid-State Drives (SSDs)

Redundant Array of Inexpensive (independent) Disks (RAID)