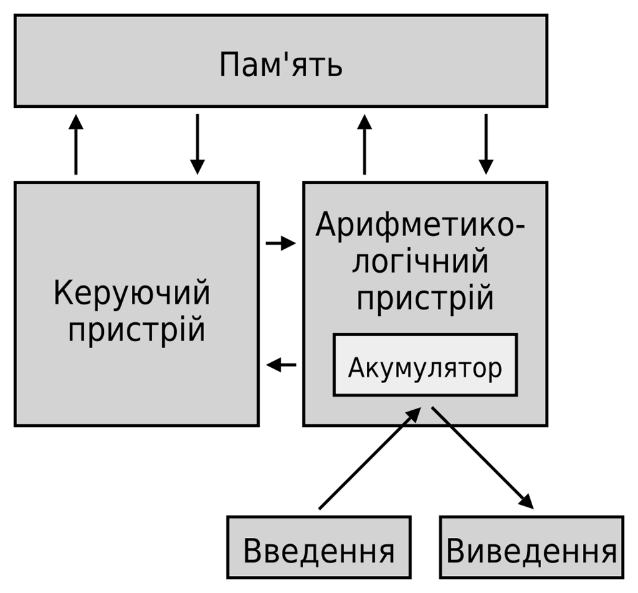
**ЛЕКЦІЯ № 2**

**КЛАСИЧНІ ПРИНЦИПИ ВИКОНАННЯ ПРОГРАМ. ОСОБЛИВОСТІ ПРОГРАМУВАННЯ ПІД УПРАВЛІННЯМ МУЛЬТИЗАДАЧНИХ ОПЕРАЦІЙНИХ СИСТЕМ. ІСТОРІЯ ПЛАТФОРМИ I386.**

**Класичні принципи виконання програм**

**Архітектура фон Неймана** (англ. Von Neumann architecture) — архітектура електронних обчислювальних машин, основною відмінністю якої від інших подібних архітектур є спільне зберігання даних та машинних команд в комірках однієї й тієї ж пам'яті, що унеможливлює їх розрізнення за способом представлення або кодування. Названа так на честь відомого математика та теоретика обчислювальної техніки Джона фон Неймана, та по сьогодні залишається домінуючою схемою організації ЕОМ загального призначення.



**Архітектурні принципи фон Неймана**, припускають, що:

- основу комп'ютера складають центральний процесор (електронна схема, що виконує обчислення) і оперативна пам'ять (електронний пристрій, що забезпечує зберігання інформації та здатне до безпосередньої взаємодії з центральним процесором);

- оперативна пам'ять складається з **комірок пам'яті**; кожна комірка здатна зберігати («пам'ятати») число з певного діапазону (зокрема, переважна більшість сучасних комп'ютерних архітектур використовує комірки розміром 8 двійкових розрядів; така комірка здатна зберігати число від 0 до 255); всі комірки пам'яті мають однаковий пристрій та однаковий розмір (принцип однорідності пам'яті);

- центральний процесор у будь-який момент може записати число з цього діапазону до будь-якої з комірок, а також прочитати вміст будь-якої комірки, тобто. дізнатися, скільки там зберігається (принцип прямого доступу до пам'яті);

- з точки зору центрального процесора комірки пам'яті різняться лише номерами — так званими **адресами** (принцип лінійності пам'яті);

- центральний процесор автоматично виконує одну за іншою операції, передбачені **програмою** (принцип програмного керування);

- програма зберігається в комірках оперативної пам'яті у вигляді **машинних інструкцій** - чисел, що являють собою кодові позначення операцій, які слід виконати (принцип програми, що зберігається);

- комірка пам'яті сама по собі «не знає», належить число що зберігається у ній до коду програми чи це деякі дані (принцип нерозрізненості команд і даних).

Нерозрізненість команд і даних дозволяє трактувати програми як дані і створювати програми, котрим у ролі оброблюваної інформації виступають інші програми. Більше того, ще чверть століття тому на деяких платформах програми могли модифікувати самі себе прямо під час виконання, але сучасні обчислювальні системи таку можливість виключають.

У складі центрального процесора присутні електронні схеми для зберігання інформації, подібні до комірок пам'яті; вони називаються **регістрами.** Зазвичай розрізняють **регістри загального призначення та службові регістри.** Регістри загального призначення призначені для короткострокового розміщення даних, що обробляються; операції з ними виконуються на порядки швидше, ніж з комірками пам'яті, але сукупний обсяг регістрів може бути в мільйони, а в сучасних умовах - у мільярди разів менше обсягу пам'яті, тому в регістрах зазвичай розташовуються вихідні дані та проміжні результати для розрахунків, що виконуються прямо зараз . Після того чи іншого розрахунку дані переносять в оперативну пам'ять, щоб звільнити регістри для інших цілей.

Службові регістри містять інформацію, необхідну самому процесору для організації виконання програми. Найважливіший із службових регістрів - **покажчик інструкції**, іноді званий також **лічильником команд**; в цьому регістрі міститься *адреса тієї комірки пам'яті, звідки процесору потрібно буде витягти код наступної дії*.

Центральний процесор працює, нескінченно повторюючи **цикл виконання команд**, що складається з трьох кроків:

- витягти код чергової машинної команди з пам'яті, починаючи з комірки, адреса якої зараз перебуває в *покажчику інструкції*;

- збільшити значення *покажчика інструкції* на довжину вилученого коду, після чого регістр буде містити адресу інструкції, що настає за поточною;

- дешифрувати витягнутий з пам'яті код команди та виконати відповідну цьому коду дію.

Чомусь багатьох студентів на іспиті ставить у глухий кут питання, звідки центральний процесор знає, яку саме машинну команду (з мільйонів команд, що знаходяться в пам'яті) потрібно виконати прямо зараз; правильна відповідь абсолютно тривіальна - адреса потрібної машинної команди знаходиться в покажчику інструкції. Процесор аж ніяк не намагається вникнути в логіку програми, що виконується, в те, якими повинні виявитися результати програми в цілому; він тільки слідує раз і назавжди встановленому циклу: рахувати код, збільшити покажчик інструкції, виконати інструкцію, почати спочатку. Автоматичне збільшення адреси, в регістрі покажчика інструкції призводить до того, що машинні команди, що становлять програму, виконуються одна за одною у тій послідовності, в якій вони записані в програмі.

Коли покажчик інструкції містить адресу того чи іншого місця, в оперативній пам'яті, кажуть, що в цьому місці пам'яті (або, що те саме, на даній ділянці програми) знаходиться керування. Логіка цього терміну заснована на тому, що дії процесора підпорядковані машинним командам (керуються ними), при цьому чергову команду процесор бере з пам'яті за адресою показчика інструкції.

Для організації добре знайомих нам **розгалужень, циклів та викликів підпрограм** застосовуються машинні команди, які примусово змінюють вміст покажчика інструкції, внаслідок чого послідовність команд порушується, а виконання програми триває з іншого місця — з тієї інструкції, чия адреса занесена до регістру. Це називається **переходом** чи **передачею управління** (На іншу ділянку машинного коду). Зазначимо, що розглянутий вище цикл виконання команди передбачає спочатку збільшити покажчик інструкції, а потім уже виконати команду, так що якщо чергова команда здійснює передачу управління, вона записує в покажчик інструкції нову адресу поверх адреси, що вже перебуває там, обчисленої в ході автоматичного збільшення.

Команди передачі управління бувають **безумовними** та **умовними**: перші просто заносять задану адресу в покажчик інструкції, тоді як другі спочатку перевіряють виконання тієї чи іншої умови, і якщо вона не виконана, не роблять нічого, тобто при цьому ніякого переходу не відбувається, а виконання продовжується, як завжди, з наступної команди. Саме умовні переходи дають змогу організувати розгалуження, а також цикли, тривалість виконання яких залежить від умов; команди безумовних переходів грають скоріше допоміжну роль, хоч і дуже важливу.

Більшість процесорів підтримує також перехід із **запам'ятовуванням адреси повернення**, що використовується для виклику підпрограм. При виконанні такого переходу адреса, що знаходиться в покажчику інструкції, спочатку зберігається десь у пам'яті, і лише після цього замінюється на новий, як правило - адресу початку машинного коду процедури або функції. Оскільки до моменту виконання команди (в даному випадку команди переходу із запам'ятовуванням повернення) в покажчику інструкції знаходиться вже адреса наступної команди, саме вона запам'ятається. Коли підпрограма завершує роботу, вона, здійснює повернення управління, тобто поміщає в покажчик інструкції те значення, яке було запам’ятовано при її виклику, так що в частині програми, що була викликана, виконання триває з команди, що йде за командою переходу із запам'ятовуванням. Зазвичай, таку команду так і називають **командою виклику**.

**Під передачею управління розуміється примусове зміна адреси, що у регістрі покажчика інструкції (лічильника команд)**. Це варто запам'ятати.

**Особливості програмування під управлінням мультизадачних операційних систем**

Практично всі сучасні операційні системи дозволяють запускати та виконувати кілька програм одночасно. Такий режим роботи обчислювальної системи, званий **мультизадачним**, породжує деякі проблеми, що вимагають вирішення з боку апаратури, насамперед — центрального процесора.

По-перше, потрібно захистити виконувані програми один від одного і саму операційну систему від програм користувача. Якщо (нехай навіть не за злим наміром, а помилково) одне з виконуваних завдань змінить щось у пам'яті, що належить іншому завданню, швидше за все це призведе до аварії цього другого завдання, причому знайти причину такої аварії виявиться принципово неможливо. Якщо завдання користувача (знов-таки помилково) внесе зміни в пам'ять операційної системи, це призведе вже до аварії всієї системи, причому, знову-таки, без можливості розібратися в причинах. Тому центральний процесор повинен підтримувати механізм **захисту пам'яті**: кожній задачі, що виконується, виділяється певна область пам'яті, і до комірок за межами цієї області завдання звертатися не може.

По-друге, в мультизадачном режимі завдання користувача, як правило, не допускаються до прямої роботи із зовнішніми пристроями. Якби це правило не виконувалося, завдання постійно б конфліктували за доступ до пристроїв, і такі конфлікти, зрозуміло, призводили б до аварій. Щоб обмежити можливості користувача завдання, творці центрального процесора, оголосили частину наявних машинних інструкцій **привілейованими**. Процесор може працювати або в **привілейованому режимі**, який також називають **режимом суперкористувача**, або в **обмеженому режимі** (він же **режим завдання** або **режим ЦП**). В обмеженому режимі привілейовані команди недоступні; у привілейованому режимі процесор може виконувати всі наявні інструкції, як звичайні, і привілейовані. Операційна система виконується, природно, в привілейованому режимі, а при передачі управління завдання користувача перемикає режим в обмежений. **Процесор може повернутися до привілейованого режиму лише за умови повернення керування операційною системою;** це виключає виконання в привілейованому режимі коду програм користувача. До привілейованих належать інструкції, які здійснюють взаємодію із зовнішніми пристроями; також в цю категорію потрапляють інструкції, що використовуються для налаштування механізмів захисту пам'яті та деякі інші команди, які впливають на роботу системи загалом. Усі такі «глобальні» дії є прерогативою операційної системи.

**Працюючи під управлінням мультизадачной операційної системи користувальницької задачі дозволено лише перетворювати інформацію у відведеної їй області оперативної пам'яті. Вся взаємодія із зовнішнім світом завдання виробляє через звернення до операційної системи**. Навіть просто вивести на екран рядок завдання самостійно не може, йому необхідно попросити операційну систему. Таке звернення користувальницької задачі до операційної системи за тими чи іншими послугами називається **системним викликом**. Цікаво, що завершення завдання теж здатне виконати лише операційна система; це стає очевидним, якщо згадати, що саме завдання - це об'єкт операційної системи, саме операційна система завантажує в пам'ять код програми, виділяє пам'ять для даних, налаштовує захист, запускає завдання, забезпечує виділення процесорного часу; при завершенні завдання необхідно позначити її пам'ять як вільну, припинити виділення цього завдання процесорного часу і зробити це, зрозуміло, може лише операційна система. Таким чином, коректному користувальницької задачі ніяк не обійтися без системних викликів, адже звернутися до операційної системи потрібно навіть для того, щоб просто завершитися.

Ще один важливий момент, який необхідно згадати перед початком вивчення конкретного процесора, — наявність у нашому операційному середовищі механізму **віртуальної пам'яті**. Спробуймо зрозуміти, що це таке. Як мовилося раніше, оперативна пам'ять ділиться на однакові за своєю ємності комірки (кожна комірка, містить 8 біт даних), і кожна така комірка має власний порядковий номер. Саме цей номер використовує центральний процесор для роботи з осередками пам'яті через загальну шину, щоб відрізняти їх одну від іншої. Назвемо цей номер **фізичною адресою** комірки пам'яті. Спочатку ніяких інших адрес, крім фізичних, у коірках пам'яті не було. У машинному коді програм використовувалися саме фізичні адреси, які називали просто «адресами», без уточнюючого слова, «фізичного». З розвитком мультизадачного режиму роботи обчислювальних систем виявилося, що в силу цілого ряду причин використання фізичних адрес незручно. Наприклад, програма в машинному коді, в якій використовуються фізичні адреси, комірок пам'яті, не зможе працювати в іншій області пам'яті — адже в мультизадачній ситуації може виявитися, що потрібна область вже зайнята іншим завданням.

У сучасних процесорах використовують два види адрес. Сам процесор працює з пам'яттю, використовуючи вже знайомі нам фізичні адреси, але в програмах, які на процесорі виконуються, використовуються зовсім інші адреси — віртуальні. **Віртуальна адреса** — це число з абстрактного віртуального адресного простору. На тих процесорах, з якими ми будемо працювати, віртуальні адреси є 32-бітовими цілими числами, тобто віртуальний адресний простір є безліч цілих чисел від 0 до 232 — 1; адреси зазвичай записуються в шістнадцятковій системі, тому адреса може бути числом від 00000000 до ffffffff. Важливо розуміти, що віртуальна адреса зовсім не повинна відповідати якомусь осередку пам'яті. Точніше, деякі віртуальні адреси відповідають фізичним осередкам пам'яті, деякі — не відповідають, а деякі адреси взагалі можуть то відповідати фізичній пам'яті, то не відповідати. Такі відповідності задаються шляхом настроювання центрального процесора, за яку відповідає операційна система. Центральний процесор, отримавши з чергової машинної інструкції віртуальну адресу, перетворює його на фізичну адресу, за якою звертається до оперативної пам'яті. Таким чином, ми в програмах використовуємо як адреси не фізичні номери, осередків пам'яті, а віртуальні (абстрактні) адреси, які потім вже процесор перетворює на справжні номери осередків. Пристрій у складі процесора, що перетворює віртуальні адреси на фізичні, називається memory management unit (ММU); це можна перекласти як «пристрій керування пам'яттю», але зазвичай абревіатуру «ММU» не перекладають.

Наявність у процесорі ММU дозволяє, зокрема, кожній програмі мати свій власний адресний простір: дійсно, ніхто не заважає операційній системі налаштувати перетворення адрес так, щоб одна і та ж віртуальна адреса в одному завдання користувача відображалася на один фізичний осередок, а в іншому завданні — зовсім на іншу.

Питання, пов'язані зі створенням нових операційних систем, ми не розглядатимемо. Натомість ми обмежимося розглядом можливостей процесора i386, доступних користувальницькій задачі, що працює в обмеженому режимі.

**Історія платформи i386**

У 1971 році корпорація Intel випустила у світ сімейство мікросхем, що отримало назву МСS-4. Одна з цих мікросхем, Intel 4004, являла собою перший у світі закінчений центральний процесор на одному кристалі, тобто, інакше кажучи, перший в історії **мікропроцесор** — принаймні з доступних широкій публіці. Машинне слово цього процесора складало чотири біти. Через рік Intel випустила восьмибітний процесор Intel 8008, а в 1974 році - більш досконалий Intel 8080. Цікаво, що 8080 використовував інші коди операцій, але при цьому програми, написані мовою асемблера для 8008, могли бути без змін відтрансльовані і для 8080. Аналогічну «сумісність за вихідним кодом» конструктори Intel підтримали і для 16-бітного процесора Intel 8086, що з'явився в 1978 році. Випущений роком пізніше процесор Intel 8088 являв собою практично такий же пристрій, що відрізняється тільки розрядністю зовнішньої шини (для 8088 вона становила 8 біт 8086 - 16 біт). Саме процесор 8088 був використаний в комп'ютері IВМ РС, що дав початок численному і неймовірно популярному сімейству машин, досі званих **IВМ РС-сумісними**, або просто **IВМ-сумісними**.

Процесори 8086 та 8088 не підтримували захисту пам'яті та не мали поділу команд на звичайні та привілейовані, так що запустити повноцінну мультизадачну операційну систему на комп'ютерах з цими процесорами було неможливо. Так само було з процесором 80186, випущеним в 1982 році. Порівняно зі своїми попередниками цей процесор працював набагато швидше, оскільки в ньому були апаратно реалізовані деякі операції, що виконувались у попередніх процесорах мікрокодом; тактова частота також зросла. Процесор включав деякі підсистеми, які раніше вимагалося підтримувати за допомогою додаткових мікросхем - такі як контролер переривань і контролер прямого доступу до пам'яті. Крім того, система команд процесора була розширена запровадженням додаткових команд; так, стало можливо, за допомогою однієї команди занести в стек всі регістри загального призначення. Адресна шина процесорів 8086, 8088 і 80186 була 20-розрядною, що дозволяло адресувати трохи більше 1 Мb оперативної пам'яті.

У тому ж 1982 році побачив світ процесор 80286, що став останнім 16-бітним процесором у ряді, що розглядається. Цей процесор підтримував так званий «захищений» режим роботи (protected mode) та сегментну модель віртуальної пам'яті, що передбачає серед інших можливостей захист пам'яті; чотири **кільця захисту** дозволили заборонити користувачам завданням виконання дій, що впливають на систему в цілому, що необхідно при роботі мультизадачної операційної системи. Адресна шина отримала чотири додаткові розряди, збільшивши максимальну кількість безпосередньо доступної пам'яті до 16 Мb.

Справжні мультизадачні операційні системи були створені лише для наступного процесора в ряду, 32-розрядного Intel 80386, для стислості, що позначається просто i386. Цей процесор, масовий випуск якого розпочався 1986 року, відрізнявся від своїх попередників збільшенням регістрів до 32 біт, істотним розширенням системи команд, збільшенням адресної шини до 32 розрядів, що дозволяло безпосередньо адресувати до 4 Gb фізичної пам'яті.

Додавання підтримки **сторінкової організації віртуальної пам'яті**, найкраще придатної реалізації мультизадачного режиму роботи, завершило картину. Саме з появою i386 так звані IВМ-сумісні комп'ютери нарешті стали повноцінними обчислювальними системами. Разом з тим i386 повністю зберіг сумісність із попередніми процесорами своєї серії, чим зумовлена досить дивна на перший погляд система регістрів. Наприклад, універсальні регістри процесорів 8086-80286 називалися АХ, ВХ, СХ та DХ та містили 16 біт даних кожен; в процесорі i386 і пізніших процесорах лінійки є регістри, містять по 32 біти і називаються ЕАХ, ЕВХ, ЕСХ і ЕDХ (літера Е означає слово «extended», тобто «розширений»), причому молодші 16 біт кожного з цих регістрів зберігають старі назви (відповідно, АХ, ВХ, СХ та DХ). Більшість інструкцій працює по-різному для операндів довжиною 8 біт, 16 біт та 32 біти, і т.п.

Подальший розвиток сімейства процесорів х86 до 2003 року був суто кількісним: збільшувалася швидкість, додавалися нові команди, але принципових змін архітектури не відбувалося. У 2001 році альянс компаній Hawllet Packard та Intel випустив процесор Itanium (Merced), архітектура якого, що отримала назву IА-64, не мала нічого спільного з х86, але включала емуляцію виконання команд архітектури i386; емуляція виявилася надто повільною для практичного застосування, а створювати програми та операційні системи для нової архітектури ніхто не поспішав. У 2003 році компанія АМD представила новий процесор, Орteron, архітектура, якого стала 64-бітним розширенням архітектури х86 подібно до того, як 32-бітна i386 стала, розширенням вихідної 16-бітної архітектури процесора, 8086. Нова система команд отримала назву «х86\_64». Поява Орteron остаточно домогла архітектуру Itanium, яка так і не змогла набути серйозного поширення, хоча процесори цієї архітектури випускаються досі. Втім, і сам Intel вже в 2004 році випустив процесор Хеоn, що мав архітектуру х86\_64.

«Багатоядерні» архітектури, що послідували за цим, являють собою не більше ніж кількісний розвиток, причому в напрямку, що практично не збільшує реальну швидкодію системи. Справа в тому, що навіть високо завантажені серверні машини в основному «впираються» у своїй продуктивності не в швидкість роботи процесора і тим більше не в конкуренцію програм за єдиний процесор, а швидше в дискових обмінах і роботи шини; ні на те, ні на інше багатоядерність вплинути не в змозі. Як правило, всі ядра, крім одного, в системі більшу частину часу просто простоюють.

Процесори архітектури х86\_64 можуть виконувати 32-бітові програми, що суттєво полегшує міграцію. Зокрема, найімовірніше комп'ютер, на якому ви намагаєтеся програмувати, якраз 64-бітний; при цьому на ньому може бути встановлена 32-бітна або 64-бітна операційна система. Ваші власні програми мовою асемблера будуть 32-бітними, але це жодним чином не завадить їх виконувати.