КЕРУВАННЯ ПАМ’ЯТТЮ В ОПЕРАЦІЙНИХ СИСТЕМАХ

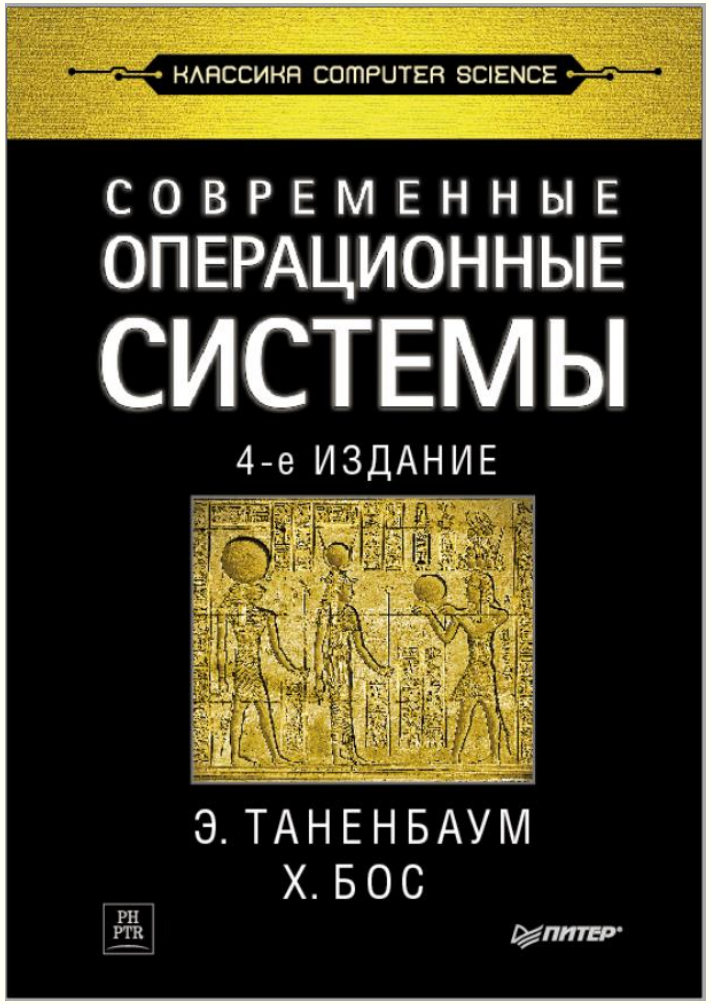
2019-2020 н. р.

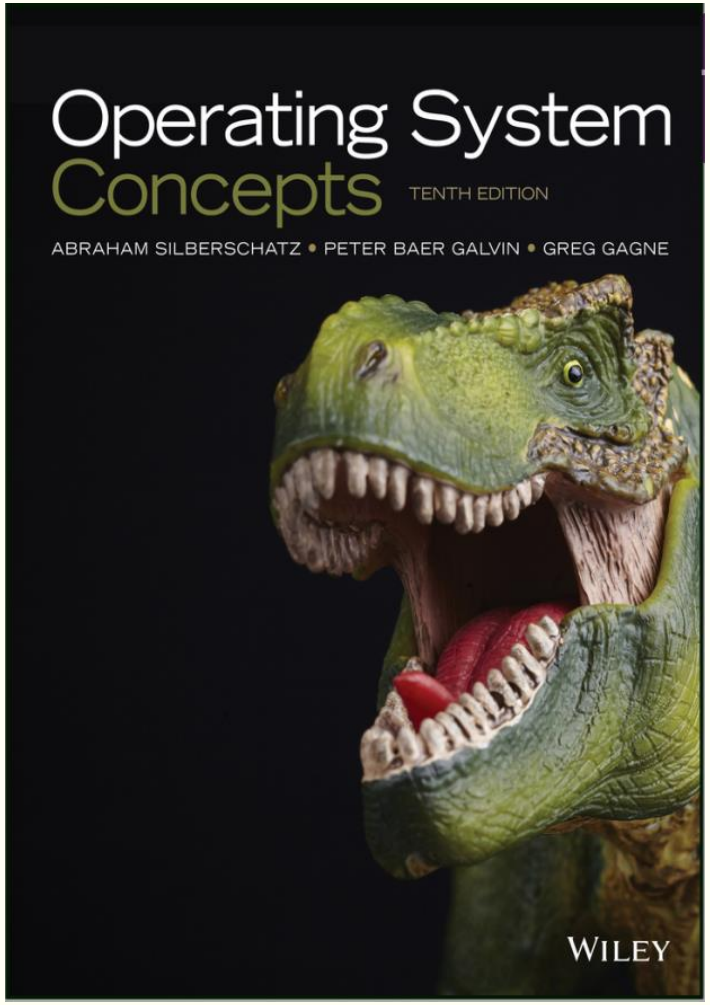
Викладач: Марченко Станіслав Віталійович

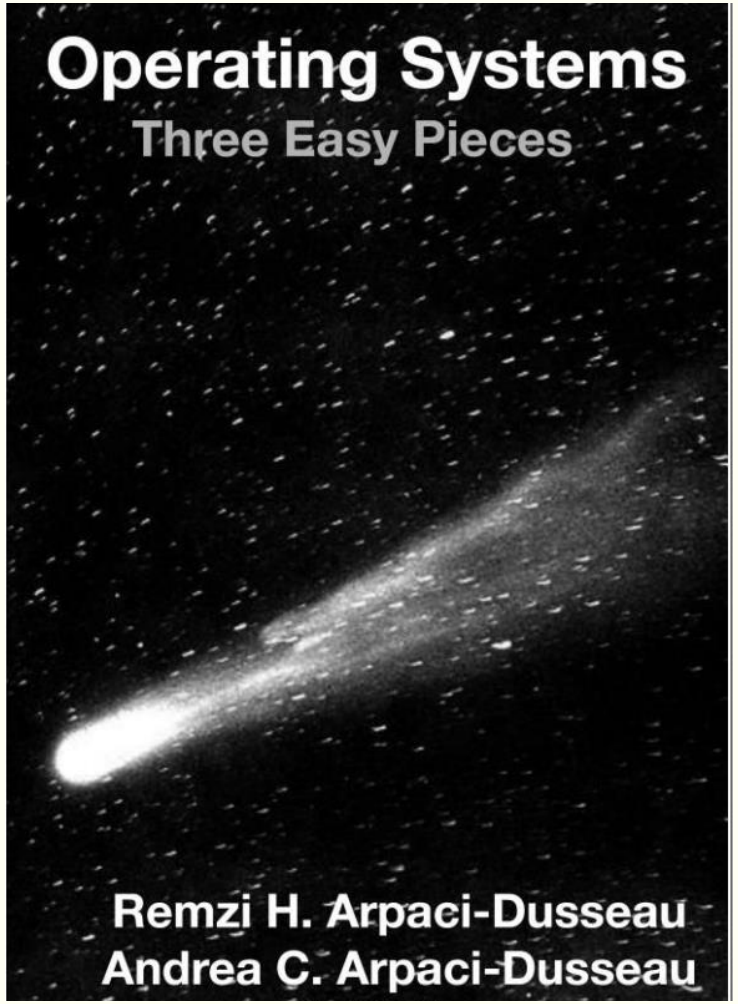
Питання лекції

- Адресні простори.
- Сторінкова організація пам’яті.
- Віртуальна пам’ять.

Рекомендована література







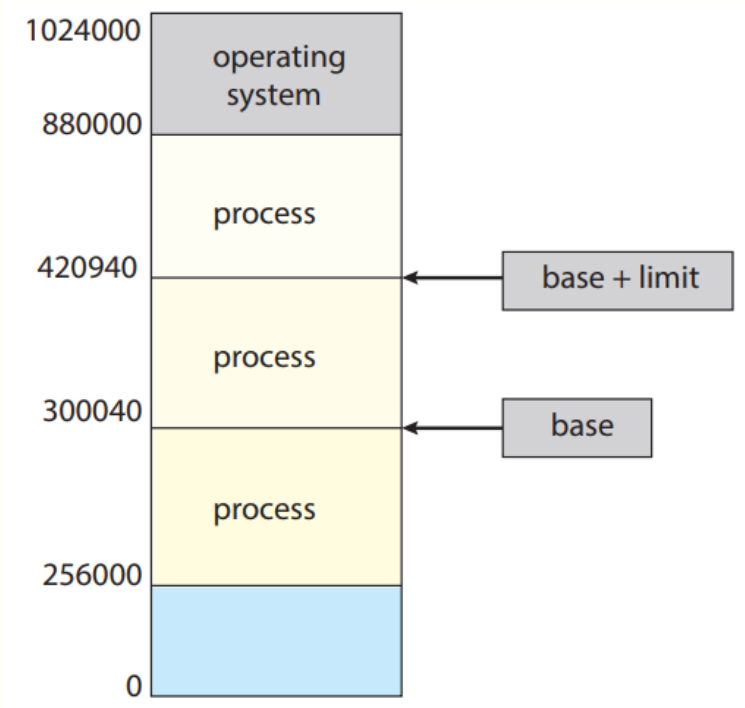
АБСТРАКЦІЇ ПАМ’ЯТІ: АДРЕСНІ ПРОСТОРИ ТА ВІРТУАЛЬНА ПАМ’ЯТЬ.

Питання 3.1

Базове апаратне забезпечення

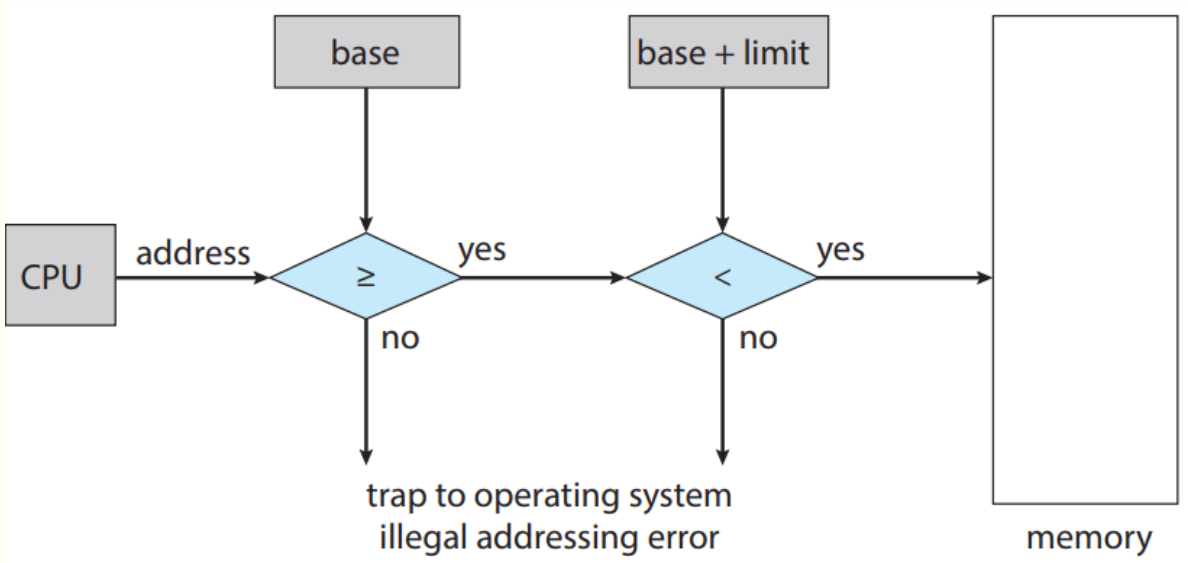
- Основна пам’ять та регістри, вбудовані в кожне процесорне ядро, є єдиними сховищами загального призначення, з якими ЦП може працювати напряму.
  - Існують машинні інструкції, які отримують ***адреси пам’яті (memory addresses)*** в якості аргументів, проте жодна не приймає ***disk addresses***.
  - Якщо дані не в пам’яті, їх туди треба перемістити, щоб ЦП міг ними оперувати.
- Регістри, вбудовані в кожне ядро ЦП, загалом доступні всередині одного машинного циклу.
  - Деякі ядра ЦП можуть декодувати інструкції та виконувати прості операції над вмістом регістрів at the rate of one or more operations per clock tick.
  - Те ж не можна сказати про основну пам’ять (main memory), доступ до якої здійснюється за допомогою транзакцій по шині пам’яті (memory bus).
  - Завершення процесу доступу до пам’яті може зайняти багато тактів ЦП.
  - У таких випадках процесору зазвичай потрібно to **stall**, оскільки він не має даних, щоб завершити інструкцію, яка виконується.
  - Така ситуація недопустима, оскільки різні частоти доступу до пам’яті.
  - Вирішення: додати швидку пам’ять між ЦП та основною пам’яттю – кеш.
  - Так апаратне забезпечення автоматично прискорює доступ до памяті без втручання ОС.

Регістри бази та ліміту визначають логічний адресний простір



- Потрібно забезпечувати коректність операцій доступу разом з відносно високою швидкістю цього доступу до фізичної пам'яті.
  - Для нормальної роботи системи необхідно захищати ОС від доступу користувацьких процесів, як і користувацькі процеси один від одного.
  - Такий захист повинен виконуватись апаратно, оскільки ОС зазвичай не втручається в процес доступу до пам'яті центральним процесором (інакше отримаємо зниження продуктивності).
- Спочатку потрібно переконатись, що кожний процес має окремий простір пам'яті.
  - Власна пам'ять процесу захищає його від інших процесів та лягає в основу завантаження багатьох процесів у пам'ять для конкурентного виконання.
  - Для відокремлення просторів пам'яті потрібно мати можливість визначати діапазон легальних адрес, до яких процес може отримати доступ, та забезпечувати доступ лише до цих адрес.
  - ***Регістр бази (base register)*** містить найменшу легальну фізичну адресу пам'яті; ***регістр ліміту (limit register)*** задає розмір діапазону.
  - Наприклад, якщо регістр бази містить 300040, а регістр ліміту - 120900, програма може легально отримати доступ тільки до адрес з діапазону [300040; 420939].

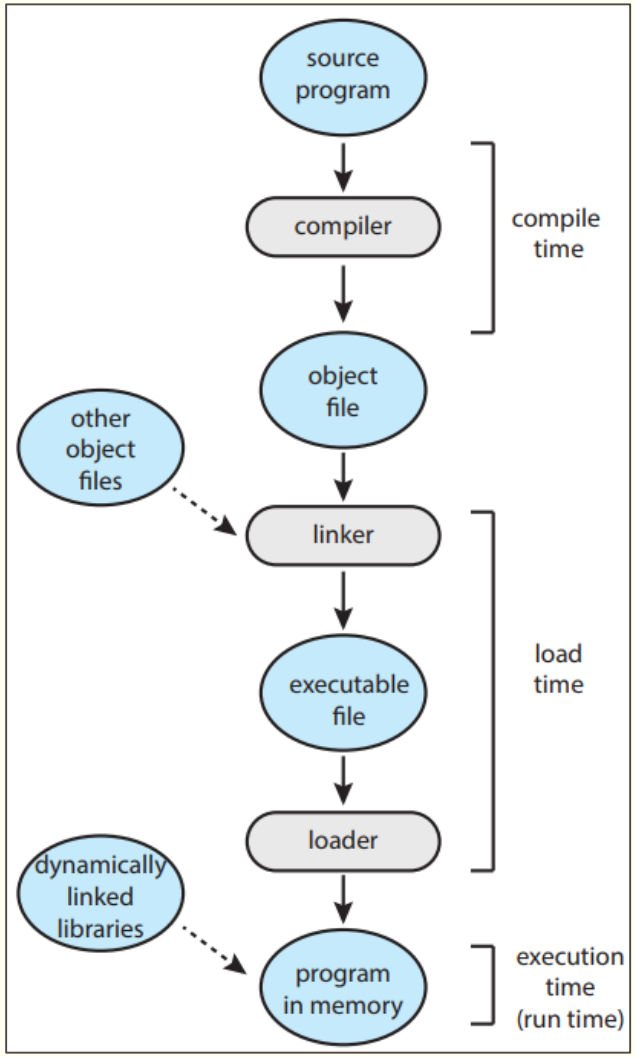
Захист апаратних адрес за допомогою регістрів бази та ліміту



- Захист простору пам'яті забезпечується апаратно процесором, який може порівняти кожну адресу, згенеровану в режимі користувача, з регістрами.
  - Будь-яка спроба програми, що виконується в режимі користувача, отримати доступ до пам'яті ОС або інших програм користувача призводить до винятку (trap) в ОС, який трактується як fatal error.
- Регістри базу та ліміту можуть завантажуватись тільки ОС, яка використовує спеціальну привілейовану інструкцію.
  - Оскільки привілейовані інструкції можуть виконуватись лише в режимі ядра, то тільки ОС може завантажувати регістри бази та ліміту.
  - Дана схема дозволяє ОС змінювати значення в регістрах, проте запобігає зміні вмісту регістрів користувацькими програмами.

ОС, виконуючись у режимі ядра, надається необмежений доступ як до пам'яті ОС, так і до пам'яті користувацьких програм.

Прив'язування (Binding) адрес



У більшості випадків користувацька програма проходить кілька етапів перед своїм виконанням.

- Адреси можуть представлятись різними способами протягом цих етапів.
- Адреси в первинному коді записуються символьно.
- Компілятор зазвичай прив'язує ці символьні адреси до зміщуваних (relocatable) адрес (наприклад, "14 байтів від початку даного модуля").
- Компонувальник та завантажувач, у свою чергу, прив'язує зміщувані адреси до абсолютних адрес. Кожна прив'язка є відображенням одного адресного простору в інший.

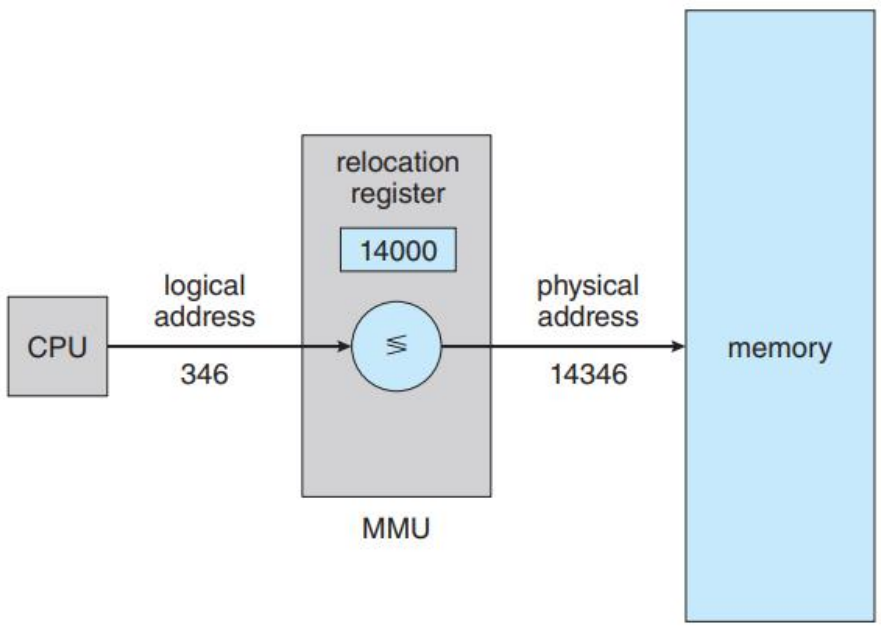
Традиційно прив'язка інструкцій та даних до адрес пам'яті може здійснюватись на будь-якому кроці шляху:

- **Compile time**. Якщо знати на момент компіляції, де процес буде розташований у пам'яті, тоді можна згенерувати ***абсолютний код***.
  - Наприклад, якщо знати, що користувацький процес буде розпочинатись з адреси *R*, тоді згенерований компілятором код розпочинатиметься звідти.
  - Якщо пізніше стартова позиція зміниться, цей код обов'язково перекомпілювати.
- **Load time**. Якщо на момент компіляції невідомо, де процес розміститься в пам'яті, тоді компілятор повинен генерувати ***зміщуваний (relocatable) код***.
  - Тоді остаточне прив'язування відкладається до моменту завантажування.
  - Якщо початкова адреса змінюється, необхідно перезавантажити користувацький код для врахування змін.
- **Execution time**. Якщо процес може переміщуватись протягом виконання з одного сегменту пам'яті в інший, тоді прив'язування повинно відкладатись до моменту run time.
  - Для роботи такої схеми потрібне спеціальне апаратне забезпечення.
  - Більшість ОС використовують цей метод.

Логічний та фізичний адресні простори

- Згенеровану ЦП адресу часто називають ***логічною***, а адресу, яку бачить пристрій керування пам'яттю (завантажену в **memory-address register**)—часто називають ***фізичною***.
- Прив'язування адрес на момент компіляції чи завантаження генерує ідентичні логічні та фізичні адреси.
  - Проте схема прив'язування адрес в момент виконання призводить до різних логічних та фізичних адрес.
  - Тоді логічну адресу зазвичай називають ***віртуальною адресою***.
  - Згенерований програмою набір усіх логічних адрес – ***логічний адресний простір***, а набір відповідних фізичних адрес – ***фізичний адресний простір***. У даній схемі вони відрізняються.
- Run-time-відображення віртуальних адрес у фізичні виконується апаратно за допомогою ***пристрою керування пам'яттю (memory-management unit, MMU)***.
  - Регістр бази тепер називається ***регістром переміщення (relocation register)***. Його значення додається до кожної адреси, згенерованої користувацьким процесом в момент, коли адреса надсилається в пам'ять.
  - Наприклад, якщо base = 14000, то спроба користувача отримати доступ до розташування 346 динамічно переміщується в розташування 14346.

Динамічне переміщення за допомогою регістру переміщення



- Користувацька програма ніколи не отримує доступ до реальних фізичних адрес.
  - Програма може створити вказівник на розташування 346, зберегти його в пам'яті, оперувати ним та порівнювати з іншими адресами.
  - Користувацька програма працює з логічними адресами.
  - Апаратне забезпечення для відображення пам'яті конвертує логічні адреси в фізичні.
  - Фінальне розташування referenced memory address не визначене, поки не буде зроблено посилання.
- Тепер маємо 2 різні типи адрес: логічні адреси [0; *max*] та фізичні адреси [*R*+0; *R*+*max*] для значення бази *R*.
  - Користувацька програма генерує тільки логічні адреси та «думає», що процес працює в діапазоні адрес від 0 до *max.*

Динамічне завантаження (dynamic loading)

- Досі було необхідно всій програмі та даним процесу розміщуватись у фізичній пам’яті для виконання.
  - Розмір процесу таким чином обмежується розміром фізичної пам’яті.
  - Для кращого використання пам’яті можемо використовувати ***динамічне завантаження*** – підпрограма не завантажується, поки її не викликали.
  - Усі підпрограми зберігаються на диску в зміщуваному (relocatable) форматі завантаження.
  - Основна програма завантажується в пам’ять та виконується. Коли потрібно викликати з підпрограми іншу підпрограму, спочатку викликаюча підпрограма перевіряє, чи можливе таке завантаження.
  - Якщо ні, то викликається relocatable linking loader, який завантажить бажану підпрограму в пам’ять та оновить таблицю адрес програми, щоб відобразити зміни. Потім управління передається новозавантаженій підпрограмі.
- Перевага динамічного завантажування – підвантаження підпрограми тільки за потреби.
  - Метод особливо корисний за потреби нечастої обробки великих обсягів коду, таких як error-підпрограми.
- Динамічне завантаження не вимагає спеціальної підтримки від ОС.
  - Це відповідальність користувачів – спроектувати програми з підтримкою переваг такого методу.
  - ОС може допомогти програмісту, постачаючи бібліотеку підпрограм для реалізації динамічного завантаження.

Динамічне компонування та спільні (Shared) бібліотеки

- **Динамічно завантажувані бібліотеки** (**DLL**) – системні бібліотеки, скомпоновані (linked) з користувацькими програмами під час роботи цих програм.
  - Деякі ОС підтримують лише ***статичне компонування (static linking)***, при якому системні бібліотеки розглядаються, як і інші об'єктні модулі, та комбінуються завантажувачем у двійковий образ програми.
  - ***Динамічне компонування (Dynamic linking)*** подібне до динамічного завантаження, проте відкладається до часу виконання. Дана риса зазвичай використовується з системними бібліотеками, на зразок стандартної бібліотеки мови С.
  - Без цього кожна програма в системі повинна включати копію відповідної бібліотеки або, принаймні, потрібних підпрограм у виконуваний образ.
  - Це не лише збільшує розмір образу, а й даремно витрачає основну пам'ять.
  - Інша перевага – можливість ділитись бібліотеками серед багатьох процесів, тримаючи єдину версію DLL в основній пам'яті. Тому DLL також називають ***спільними бібліотеками***, вони широко використовуються в системах Windows та Linux.
- Коли програма посилається на підпрограму з динамічної бібліотеки, завантажувач знаходить DLL, за необхідності завантажуючи її в пам'ять.
  - Потім він adjusts адреси, які спрямовують функції з динамічної бібліотеки до місця в пам'яті, де зберігається DLL.

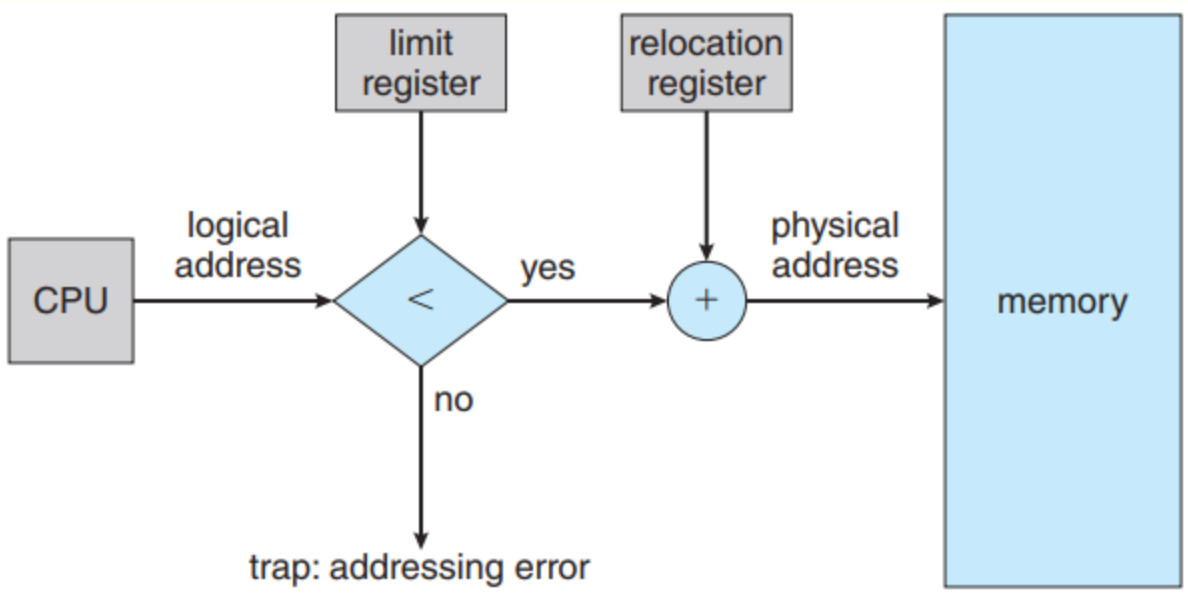
Динамічне компонування та спільні (Shared) бібліотеки

- Динамічно скомпоновані бібліотеки можна розширювати оновленнями.
  - Можна заміняти бібліотеку на новішу версію, і всі програми, що посилаються на бібліотеку, автоматично використовуватимуть її. Без динамічного компонування потрібно перезбирати код.
  - Для того, щоб програми не використовували нові, несумісні версії бібліотек, інформацію про версію включається як у програму, так і в бібліотеку.
  - У пам’ять можна завантажувати багато різних версій бібліотеки, і кожна програма використовуватиме свою версійну інформацію для вибору копії бібліотеки для застосування.
- На відміну від динамічного завантаження, динамічне компонування та спільні бібліотеки вимагають допомоги від ОС.
  - Якщо процеси в пам’яті захищені один від одного, тоді тільки ОС може перевірити, чи потрібна підпрограма знаходиться в просторі пам’яті іншого процесу або чи можливий доступ багатьох процесів до однакових адрес пам’яті.

Безперервне (Contiguous) виділення пам’яті

- Пам’ять зазвичай розділяється на 2 розділи: для ОС та для користувацьких процесів.
  - ОС розміщається в просторі або малих, або великих адрес. Вибір залежить від багатьох чинників, зокрема, розташування вектору переривань.
  - Проте багато ОС (включаючи Linux та Windows) розміщуються у просторі великих адрес (high memory), тому розглядатимемо таку ситуацію.
- Зазвичай бажано, щоб кілька користувацьких процесів залишались у пам’яті в один момент часу.
  - Тому слід розглянути спосіб виділення доступної пам’яті процесам, які очікують на входження в пам’ять.
  - При безперервному виділенні пам’яті кожен процес розміщується в окремій секції пам’яті, суміжній з секцією наступного процесу.

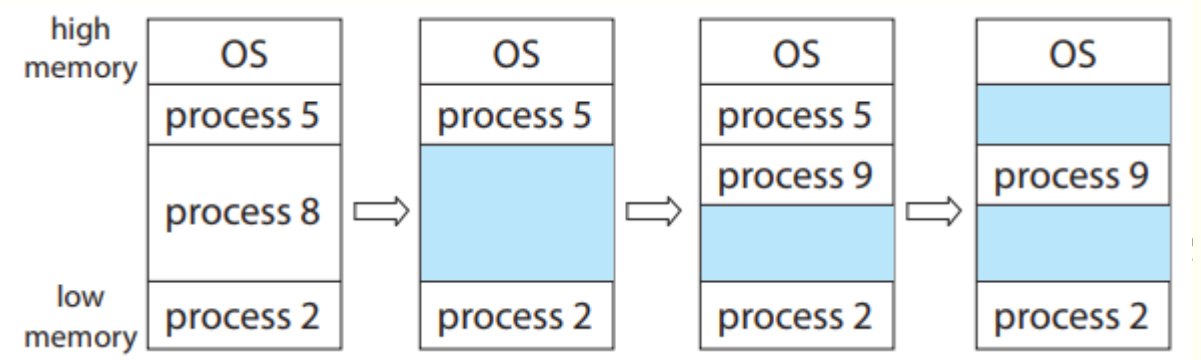
Захист пам’яті



- Можемо запобігти доступ процесу до пам’яті, яка йому не належить, комбінуючи вже згадані ідеї.
  - Якщо маємо систему з регістром переміщення разом з регістром ліміту, ціль досягається.
  - Регістр переміщення містить значення найменшої фізичної адреси; регістр ліміту – діапазон логічних адрес, в який кожна логічна адреса повинна потрапляти.
  - MMU динамічно відображає логічну адресу, додаючи значення регістру переміщення. Відображена адреса надсилається в пам’ять.
- Коли планувальник ЦП обирає процес для виконання, диспетчер завантажує ці регістри з правильними значеннями як частину перемикання контексту.
  - Оскільки кожна згенерована ЦП адреса перевіряється відносно даних регістрів, можна захистити як ОС, так і інші користувацькі програми й дані від змінювання працюючим процесом.

Дана схема надає ефективний спосіб динамічної зміни розміру ОС.

Виділення пам’яті



- Один з найпростіших методів виділення пам’яті – присвоїти процеси розділам пам’яті різного розміру, де кожний розділ може містити рівно 1 процес.
  - За такої схеми ОС утримує таблицю, що вказує, які частини пам’яті доступні, а які – зайняті.
  - Спочатку вся пам’ять доступна користувацьким процесам та вважається одним блоком доступної пам’яті - ***діркою (hole)***. З часом пам’ять міститиме набір дірок різних розмірів.
- На рисунку пам’ять спочатку повністю використовується процесами 5, 8 та 2.
  - Після вивантаження процесу 8 залишається 1 безперервна дірка.
  - Пізніше заходить процес 9, якому виділяється пам’ять.
  - Процес 5 завершується, залишаючи 2 розривні дірки.
- При вході процесів у систему ОС враховує вимоги до пам’яті кожного процесу та обсяг доступної пам’яті при визначенні алокованих процесів.
  - Коли процес в алокованому просторі, він завантажується в пам’ять, де потім буде змагатись за процесорний час.
  - Коли процес завершується, він звільняє пам’ять, яку ОС може потім надати іншому процесу.

Що буде, коли немає достатньо пам'яті для забезпечення потреб процесу, який надійшов?

- Варіант 1: відкинути процес з відповідним повідомленням про помилку.
  - Варіант 2: розташувати процес у черзі очікування.
  - Коли пам'ять пізніше буде звільнена, ОС перевірить цю чергу та можливість задовольнити потреби в пам'яті очікуючого процесу.
- Загалом блоки пам'яті включають набір дірок різного розміру, розпорошений по пам'яті.
  - Коли процес надходить та потребує пам'яті, система шукає в наборі достатньо велику дірку.
  - Якщо дірка надто велика, вона розбивається на 2 частини: одна виділяється по розміру процесу, інша – повертається в набір дірок.
  - Коли процес завершується, звільнений блок пам'яті повертається до набору дірок. Суміжні дірки зливаються.
- Дана процедура є конкретним прикладом загальної ***задачі динамічного виділення пам'яті (dynamic storage-allocation problem)***, яка передбачає задоволення запиту на виділення пам'яті розміром *n* зі списку вільних дірок.
  - Розв'язків такої задачі багато.

Стратегії

- Стратегії **first-fit**, **best-fit** та **worst-fit** є найбільш поширеними при виборі вільної дірки з множини доступних:
  - **First-fit**. Виділяє першу достатньо велику дірку. Пошук стартує або з початку набору дірок, або з місця закінчення попереднього пошуку (next-fit).
  - **Best-fit**. Виділяє найменшу достатньо велику дірку. Необхідно шукати в усьому списку, якщо він не впорядкований за розміром. Дана стратегія продукує найменші залишкові дірки.
  - **Worst fit**. Виділяє найбільшу дірку. Потрібно знову шукати в усьому списку, якщо він не відсортований за розміром дірок. Дана стратегія продукує найбільші залишкові дірки, що може бути кориснішим, ніж менші дірки в підході best-fit.
- Симуляції показують, що як first-fit, так і best-fit кращі за worst-fit у контексті зменшення часу та використання простору.
  - Утилізація простору first-fit та best-fit підходами приблизно рівноцінна, проте first-fit загалом швидший.

Фрагментація

- Як first-fit, так і best-fit стратегії виділення пам’яті страждають від ***зовнішньої фрагментації***.
  - Завантаження та вивантаження процесів створює багато маленьких дірок у просторі пам’яті.
  - У найгіршому випадку отримаємо блок вільної пам’яті між кожним процесом.
  - Якби всі такі малі частини були в єдиному великому блоці, можна було б запустити більше процесів.
- Використання first-fit або best-fit стратегій може вплинути на обсяг фрагментації.
  - Інший фактор – який кінець блоку виділяти: початок чи кінець.
- Залежно від загального обсягу пам’яті та середнього розміру процесу, зовнішня фрагментація може бути як великою, так і малою проблемою.
  - Статистичний аналіз first-fit стратегії, наприклад, показує, що навіть з деякими оптимізаціями при N виділених блоках інші $0.5N$ блоків будуть втрачені завдяки фрагментації.
  - Тому третина пам’яті може бути unusable! Це правило відоме як ***правило 50%***.

Фрагментація

- Фрагментація пам’яті може бути як внутрішньою, так і зовнішньою.
  - Нехай схема виділення пам’яті з багатьма розділами має дірку 18464 байти.
  - Припустимо, що наступний процес звертається за 18,462 байтами. Виділення такого блоку залишить дірку розміром 2 байти.
  - Накладні витрати на відстеження цієї дірки будуть значно вищими, ніж власне дірка.
  - Загальний підхід для уникнення цієї проблеми – розбити фізичну пам’ять на блоки фіксованого розміру та виділяти пам’ять в одиницях, заснованих на розмірі блоку.
  - Тоді виділена процесу пам’ять може бути дещо більшою, ніж запитувана пам’ять. Цю різницю називають ***внутрішньою фрагментацією***.
- Одне з вирішень проблеми зовнішньої фрагментації – ***стиснення (compaction)***.
  - Мета: змішати вміст пам’яті так, щоб розмістити всю вільну пам’ять в одному блоці.
  - Стиснення неможливе, якщо переміщення статичне або виконується в момент збирання чи завантаження програми.
  - Стиснення можливе тільки тоді, коли переміщення динамічне та здійснюється в момент виконання (execution time).
  - Якщо адреси зміщуються динамічно, зміщення потребує тільки пересування програми з даними, а потім зміни регістру бази, щоб відобразити нову базову адресу.
  - Найпростіший алгоритм стиснення – переміщення всіх процесів у один кінець пам’яті, а дірок – в інший.
  - Проте така схема може бути затратною.
- Інше можливе вирішення – дозволити логічному адресному простору процесів бути розривним, тобто дозволити виділяти процесу фізичну пам’ять за її доступності.
  - Дана стратегія називається сторінковою організацією пам’яті ***(paging)***, найбільш поширена техніка управління пам’яттю.