**BÁO CÁO BÀI TẬP LỚN**

**HỆ ĐIỀU HÀNH**

**Đề tài: Thiết kế một ứng dụng hiển thị thông tin bao gồm RAM và STORAGE của thiết bị trong hệ điều hành android.**

***Họ và tên: Trần Quang Huy***

***MSSV : 20161863***

***Lớp CNTT2.01 – K61***

***Người hướng dẫn : Ths. Đỗ Tuấn Anh***

1. **Cơ sở lý thuyết**

**I.1 Quản lý bộ nhớ**

**I.1.1 Khái niệm về quản lý bộ nhớ**

* Quản lý bộ nhớ là một trong những nhiệm vụ quan trọng và phức tạp nhất của hệ điều hành.
* Bộ phận quản lý bộ nhớ xem bộ nhớ chính như là một tài nguyên của hệ thống dùng để cấp phát và chia sẻ cho nhiều tiến trình đang ở trong trạng thái active
* Công cụ cơ bản của quản lý bộ nhớ là sự phân trang (paging) và sự phân đoạn (segmentation).
* Hệ điều hành cũng có thể kết hợp giữa phân trang và phân đoạn để có được một chiến lược quản lý bộ nhớ linh hoạt hơn.

**I.1.2 Nhiệm vụ của quản lý bộ nhớ**

*I.1.2.1 Sự tái định vị (Relocation):*

Trong các hệ thống đa chương, không gian bộ nhớ chính thường được chia sẻ cho nhiều tiến trình khác nhau và yêu cầu bộ nhớ của các tiến trình luôn lớn hơn không gian bộ nhớ vật lý mà hệ thống có được.

Một chương trình đang hoạt động trên bộ nhớ cũng có thể bị đưa ra đĩa (swap-out) và nó sẽ được đưa vào lại (swap-in) bộ nhớ tại một thời điểm thích hợp nào đó sau này. Vấn đề đặt ra là khi đưa một chương trình vào lại bộ nhớ thì hệ điều hành phải định vị nó vào đúng vị trí mà nó đã được nạp trước đó. Để thực hiện được điều này hệ điều hành phải có các cơ chế để ghi lại tất cả các thông tin liên quan đến một chương trình bị swap-out, các thông tin này là cơ sở để hệ điều hành swap-in chương trình vào lại bộ nhớ chính và cho nó tiếp tục hoạt động. Hệ điều hành buộc phải swap-out một chương trình vì nó còn không gian bộ nhớ chính để nạp tiến trình khác, do dó sau khi swap-out một chương trình hệ điều hành phải tổ chức lại bộ nhớ để chuẩn bị nạp tiến trình vừa có yêu cầu.

Các nhiệm vụ trên do bộ phần quản lý bộ nhớ của hệ điều hành thực hiện. Ngoài ra trong nhiệm vụ này hệ điều hành phải có khả năng chuyển đổi các địa chỉ bộ nhớ được ghi trong code của chương trình thành các địa chỉ vật lý thực tế trên bộ nhớ chính khi chương trình thực hiện các thao tác truy xuất trên bộ nhớ, bởi vì người lập trình không hề biết trước hiện trạng của bộ nhớ chính và vị trí mà chương trình được nạp khi chương trình của họ hoạt động.

Trong một số trường hợp khác các chương trình bị swap-out có thể được swap-in vào lại bộ nhớ tại vị trí khác với vị trí mà nó được nạp trước đó.

*I.1.2.1 Bảo vệ bộ nhớ (Protection):*

Mỗi tiến trình phải được bảo vệ để chống lại sự truy xuất bất hợp lệ vô tình hay có chủ ý của các tiến trình khác. Vì thế các tiến trình trong các chương trình khác không thể tham chiếu đến các vùng nhớ đã dành cho một tiến trình khác để thực hiện các thao tác đọc/ghi mà không được phép (permission), mà nó chỉ có thể truy xuất đến không gian địa chỉ bộ nhớ mà hệ điều hành đã cấp cho tiến trình đó.

Hệ thống quản lý bộ nhớ phải biết được không gian địa chỉ của các tiến trình khác trên bộ nhớ và phải kiểm tra tất cả các yêu cầu truy xuất bộ nhớ của mỗi tiến trình khi tiến trình đưa ra địa chỉ truy xuất. Điều này khó thực hiện vì không thể xác định địa chỉ của các chương trình trong bộ nhớ chính trong quá trình biên dịch mà phải thực hiện việc tính toán địa chỉ tại thời điểm chạy chương trình.

Hệ điều hành có nhiều chiến lược khác nhau để thực hiện điều này. Điều quan trọng nhất mà hệ thống quản lý bộ nhớ phải thực hiện là không cho phép các tiến trình của người sử dụng truy cập đến bất kỳ một vị trí nào của chính hệ điều hành, ngoại trừ vùng dữ liệu và các rountine mà hệ điều hành cung cấp cho chương trình người sử dụng.

*I.1.2.2 Chia sẻ bộ nhớ (Sharing):*

Bất kỳ một chiến lược nào được cài đặt đều phải có tính mềm dẻo để cho phép nhiều tiến trình có thể truy cập đến cùng một địa chỉ trên bộ nhớ chính. Ví dụ, khi có nhiều tiến trình cùng thực hiện một chương trình thì việc cho phép mỗi tiến trình cùng truy cập đến một bản copy của chương trình sẽ thuận lợi hơn khi cho phép mỗi tiến trình truy cập đến một bản copy sở hữu riêng.

Các tiến trình đồng thực hiện (co-operating) trên một vài tác vụ có thể cần để chia sẻ truy cập đến cùng một cấu trúc dữ liệu. Hệ thống quản lý bộ nhớ phải điều khiển việc truy cập đến không gian bộ nhớ được chia sẻ mà không vi phạm đến các yêu cầu bảo vệ bộ nhớ.

Trong môi trường hệ điều hành đa nhiệm hệ điều hành phải chia sẻ không gian nhớ cho các tiến trình để hệ điều hành có thể nạp được nhiều tiến trình vào bộ nhớ để các tiến trình này có thể hoạt động đồng thời với nhau.

*I.1.2.3 Tổ chức bộ nhớ logic (Logical organization):*

Bộ nhớ chính của hệ thống máy tính được tổ chức như là một dòng hoặc một mảng, không gian địa chỉ bao gồm một dãy có thứ tự các byte hoặc các word. Bộ nhớ phụ cũng được tổ chức tương tự. Mặc dù việc tổ chức này có sự kết hợp chặt chẽ với phần cứng thực tế của máy nhưng nó không phù hợp với các chương trình.

Đa số các chương trình đều được chia thành các modun, một vài trong số đó là không thể thay đổi (read only, execute only) và một vài trong số đó chứa dữ liệu là có thể thay đổi. Nếu hệ điều hành và phần cứng máy tính có thể giao dịch một cách hiệu quả với các chương trình của người sử dụng và dữ liệu trong các modun thì một số thuận lợi có thể thấy rõ sau đây:

Các modun có thể được viết và biên dịch độc lập, với tất cả các tham chiếu từ một modun đến modun khác được giải quyết bởi hệ thống tại thời điểm chạy.

Các mức độ khác nhau của sự bảo vệ, read-only, execute-only, có thể cho ra các modun khác nhau.

Nó có thể đưa ra các cơ chế để các modun có thể được chia sẻ giữa các tiến trình.

Công cụ đáp ứng cho yêu cầu này là sự phân đoạn (segmentation), đây là một trong những kỹ thuật quản lý bộ nhớ được trình bày trong chương này.

*I.1.2.4 Tổ chức bộ nhớ vật lý (Physical organization):*

Như chúng ta đã biết bộ nhớ máy tính được tổ chức theo 2 cấp: bộ nhớ chính và bộ nhớ phụ. Bộ nhớ chính cung cấp một tốc độ truy cập dữ liệu cao, nhưng dữ liệu trên nó phải được làm tươi thường xuyên và không thể tồn tại lâu dài trên nó. Bộ nhớ phụ có tốc độ truy xuất chậm và rẻ tiền hơn so với bộ nhớ chính nhưng nó không cần làm tươi thường xuyên. Vì thế bộ nhớ phụ có khả năng lưu trữ lớn và cho phép lưu trữ dữ liệu và chương trình trong một khoảng thời gian dài, trong khi đó bộ nhớ chính chỉ để giữ (hold) một khối lượng nhỏ các chương trình và dữ liệu đang được sử dụng tại thời điểm hiện tại.

Trong giản đồ 2 cấp này, việc tổ chức luồng thông tin giữa bộ nhớ chính và bộ nhớ phụ là một nhiệm vụ quan trọng của hệ thống. Sự chịu trách nhiệm cho luồng này có thể được gán cho từng người lập trình riêng, nhưng điều này là không hợp lý và có thể gây rắc rối, là do hai nguyên nhân:

* Không gian bộ nhớ chính dành cho các chương trình cùng với dữ liệu của nó thường là không đủ, trong trường hợp này, người lập trình phải tiến hành một thao tác được hiểu như là Overlaying, theo đó chương trình và dữ liệu được tổ chức thành các modun khác nhau có thể được gán trong cùng một vùng của bộ nhớ, trong đó có một chương trình chính chịu trách nhiệm chuyển các modun vào và ra khi cần.
* Trong môi trường đa chương trình, người lập trình không thể biết tại một thời điểm xác định có bao nhiêu không gian nhớ còn trống hoặc khi nào thì không gian nhớ sẽ trống.

Như vậy nhiệm vụ di chuyển thông tin giữa 2 cấp bộ nhớ phải do hệ thống thực hiện. Đây là nhiệm vụ cơ bản mà thành phần quản lý bộ nhớ phải thực hiện.

**I.2 Quản lý bộ nhớ trông hệ điều hành android**

**I.2.1 RAM và máy ảo Dalvik**

Random-access memory (RAM) là một nguồn tài nguyên quý giá trong bất kỳ môi trường phát triển phần mềm nào, và nó còn có giá trị hơn trên một hệ điều hành di động, nơi bộ nhớ vật lý thường được hạn chế. Mặc dù máy ảo Dalvik Android thông thường sẽ thực hiện việc thu gom rác thải, nhưng điều này cũng không cho phép bạn bỏ qua khi nào và nơi bạn đã cấp phát hay giải phóng bộ nhớ.

Để cho các bộ thu rác có thể lấy lại bộ nhớ từ ứng dụng, bạn cần tránh vấn đề memory leak và giải phóng bất kỳ một Refernce object nào tại thời điểm thích hợp. Với phần lớn ứng dụng, bộ gom rác Dalvik sẽ sử lý phần còn lại: hệ thống đòi cấp phát bộ nhớ khi object tương ứng ròi khỏi phạm vi của active thread của ứng dụng.

**I.2.2 Cách để quản lý bộ nhớ ứng dụng Android**

Android không cung cấp không gian trao đổi về bộ nhớ, nhưng nó sử dụng paging và memory-mapping (mmapping) để quản lý bộ nhớ.Nghĩa là bất kỳ bộ nhớ nào bạn sửa đổi cho dù bằng cách cấp phát các đối tượng mới hoặc động đến các trang mmapped -vẫn thường trú trong bộ nhớ RAM và không thể được paged ra. Vì vậy, cách duy nhất để hoàn toàn giải phóng bộ nhớ từ ứng dụng của bạn là để phát hành đối tượng quan hệ bạn có thể nắm giữ, làm cho bộ nhớ có thể được giải phóng. Đó là với một ngoại lệ: bất kỳ tập tin mmapped trong mà không sửa đổi, chẳng hạn như mã số, có thể được paged ra RAM nếu hệ thống muốn sử dụng bộ nhớ ở nơi khác.

*I.2.2.1 Chia sẻ bộ nhớ*

* Để làm phù hợp tất cả mọi thứ nó cần trong bộ nhớ RAM, Android sẽ cố gắng để chia sẻ các trang bộ nhớ RAM trên quy trình. Nó có thể làm như vậy trong các cách sau:
* Mỗi một tiến trình ứng dụng được chia ra từ một tiến trình có sẵn có tên Zygote. Tiến trình Zygote bắt đầu khi khi hệ thống khởi động và load các mã framework và resource chung (giống như activity theme). Để bắt đầu một ứng dụng tiến trình mới, hệ thống chia tiến trình Zygote ra, sau đó load và chạy code của ứng dụng trong một tiến trình mới. Điều này cho phép hầu hết các trang bộ nhớ RAM cấp phát cho code framework và resource có thể chia sẻ cho tất cả các tiến trình trong app.
* Hầu hết dữ liệu tĩnh được ánh xạ vào một tiến trình. Điều này không chỉ cho phép cùng một dữ liệu được chia sẻ giữa các tiến trình mà còn cho phép nó được paged ra khi cần thiết.
* Ở nhiều nơi, Android chia sẻ RAM động qua quá trình sử dụng được phân bổ một cách rõ ràng các vùng bộ nhớ chia sẻ (hoặc với ashmem hoặc gralloc). Ví dụ, bề mặt cửa sổ sử dụng bộ nhớ giữa các ứng dụng và màn hình compositor chia sẻ, và bộ đệm con trỏ sử dụng bộ nhớ giữa các nhà cung cấp nội dung và khách hàng chia sẻ.

*I.2.2.2 Cấp phát và đòi bộ nhớ:*

* Dalvik heap cho mỗi quá trình được hạn chế đến một phạm vi bộ nhớ ảo duy nhất. Điều này xác định kích thước heap hợp lý, có thể phát triển như nó cần phải (nhưng chỉ đến một giới hạn mà hệ thống định nghĩa cho mỗi ứng dụng).
* Kích thước hợp lý của heap không giống như số lượng bộ nhớ vật lý được sử dụng bởi nó. Khi kiểm tra heap của ứng dụng, Android tính một giá trị gọi là tỉ lệ Set Kích thước (PSS), chiếm cả các trang dirty và clean được chia sẻ với các quá trình khác, nhưng chỉ trong tổng số đó là tỷ lệ thuận với số ứng dụng được chia sẻ RAM. (PSS) tổng số này là những gì hệ thống coi là bộ nhớ vật lý của bạn.
* Các heap Dalvik không thu gọn kích thước vật lý của heap, có nghĩa rằng Android không chống phân mảnh heap để dồn lại không gian. Android chỉ có thể thu nhỏ kích thước heap hợp lý khi có không gian chưa sử dụng vào cuối heap. Nhưng điều này không có nghĩa là bộ nhớ vật lý được sử dụng bởi các heap không thể co lại. Sau khi thu gom rác thải, Dalvik dẫn heap và tìm các trang không sử dụng, sau đó trả lại những trang này về kernel sử dụng madvise. Cải tạo bộ nhớ từ phân bổ nhỏ có thể có hiệu quả vì các trang được sử dụng cho một phân bổ nhỏ vẫn có thể được chia sẻ với những trang khác mà chưa được giải phóng.

*I.2.2.3 Hạn chế bộ nhớ ứng dụng*

* Để duy trì một môi trường chức năng mutil-task, Android đặt một giới hạn cứng về kích thước heap cho mỗi ứng dụng. Giới hạn kích thước heap chính xác thay đổi giữa các thiết bị dựa trên tổng RAM mà thiết bị có sẵn. Nếu ứng dụng của bạn đã đạt đến giới hạn sức chứa của nó và cố gắng để cấp phát bộ nhớ hơn, nó sẽ nhận được một OutOfMemoryError.
* Trong một số trường hợp, bạn có thể muốn truy vấn hệ thống để xác định chính xác bao nhiêu không gian heap bạn đã có sẵn trên thiết bị hiện tại, để xác định có bao nhiêu dữ liệu an toàn để giữ trong bộ nhớ cache. Bạn có thể truy vấn hệ thống cho con số này bằng cách gọi getMemoryClass (). Nó trả về một số nguyên cho biết số lượng của MB có sẵn cho heap của ứng dụng của bạn.

*I.2.2.4 Chuyển các ứng dụng*

* Thay vì sử dụng không gian trao đổi khi người dùng chuyển đổi giữa các ứng dụng, Android giữ các quy trình mà không được lưu trữ tại một nền trước ( "người dùng có thể nhìn thấy") thành phần ứng dụng trong một cache LRU (least-recently used). Ví dụ, khi người dùng đầu tiên ra mắt một ứng dụng, một tiến trình được tạo ra cho nó, nhưng khi người dùng rời khỏi ứng dụng, quá trình này không thoát. Hệ thống này giúp quá trình được lưu trữ, vì vậy nếu người dùng sau đó trở lại ứng dụng, tiến trình này được sử dụng lại cho các ứng dụng chuyển đổi nhanh hơn.
* Nếu ứng dụng của bạn có một tiến trình trình lưu trữ và nó vẫn nhớ rằng hiện tại nó không cần, kể cả người dùng không sử dụng nó, làm hạn chế hiệu suất tổng thể của hệ thống

1. **Phần mềm ứng dụng trên hệ điều hành android**

**II.1 Thông tin cơ bản của phần mềm ứng dụng**

Tên ứng dụng: CPU-Z.

Chức năng chính:

* + Hiển thị tên model, brand của thiết bị
  + Hiển thị thông tin liên quan tới bộ nhớ:
  + Tổng dung lượng RAM, memory
  + RAM, Memory đang sử dụng
  + Hiển thị danh sách và thông tin các application đã cài đặt
  + Thông tin cơ bản mỗi ứng dụng bao gồm
    - Tên ứng dụng
    - Tên package (gói) của ứng dụng
    - Đường dẫn nguồn của ứng dụng (file \*.apk)
    - Thời gian cài đặt
    - Icon

**II.2 Công cụ sử dụng**

* + Android studio 3.0.1
  + Thiết bị máy thật android
  + Máy ảo android

**II.3 Cơ chế hoạt động**

*II.3.1 Hiển thị thông tin cơ bản của thiết bị*

Với mỗi một thiết bị được sản xuất, đều có những thông số kèm theo nó, đặc biệt đối với những thiết bị chạy hệ điều hành android cũng vậy, những thông số kèm theo nó được API hỗ trợ và thông tin nằm trong class Build

Tên class: public class Build  extends  Object

Thư viện được import: java.lang.Object.

Chức năng của class: Chứa thông tin thuộc về sản xuất, được lấy từ hệ thống.

Cách sử dụng: Để lấy thông tin, ta sử dụng cú pháp: (Build.\*).  Cụ thể, những thông tin liên quan tới Model, Brand được gọi hàm trực tiếp: Build.MODEL,  Build.BRAND,  ..

*II.3.2 Hiển thị thông tin của RAM, Storage*

Thông tin liên quan tới RAM, Memory, Storage không liên quan tới nhà sản suất nhiều nên không nằm trong class Build, để lấy thông tin về những chủ đề này, ta phải lấy từ class  ActivityManager

Tên class: ActivityManager

Thư viện import  :  java.lang.Object

Chức năng của class : Đưa ra thông tin liên quan tới những sự qua lại lẫn nhau giữa các activities,  services   và những process   liên quan.

* Thông tin liên quan tới Memory nằm trong class con : ActivityManager.MemoryInfo
* Lấy dung lượng available Memory: new  ActivityManager.MemoryInfo).availMem
* Lấy dung lượng total Memory: (new  ActivityManager.memoryInfo).totalMem

1. private int getAvailableRam(boolean getPercent) {
2. ActivityManager.MemoryInfo mi = new ActivityManager.MemoryInfo();
3. ActivityManager activityManager = (ActivityManager) getSystemService(ACTIVITY\_SERVICE);
4. activityManager.getMemoryInfo(mi);
5. double availableMegs = mi.availMem / 0x100000 L;
7. //Percentage can be calculated for API 16+
8. double percentAvail = mi.availMem / (double) mi.totalMem \* 100.0;
10. return getPercent ? (int) percentAvail : (int) availableMegs;
11. }

Tuy nhiên để dung lượng của bộ nhớ lưu trữ trong thiết bị, ta không dùng class này, mà ta phải thông qua các blocks

1. private long getStorages(int mode) {
2. StatFs internalStatFs = new StatFs(Environment.getRootDirectory().getAbsolutePath());
3. long internalTotal;
4. long internalFree;
6. StatFs externalStatFs = new StatFs(Environment.getExternalStorageDirectory().getAbsolutePath());
7. long externalTotal;
8. long externalFree;
10. if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.JELLY\_BEAN\_MR2) {
11. internalTotal = (internalStatFs.getBlockCountLong() \* internalStatFs.getBlockSizeLong()) / (KILOBYTE \* KILOBYTE);
12. internalFree = (internalStatFs.getAvailableBlocksLong() \* internalStatFs.getBlockSizeLong()) / (KILOBYTE \* KILOBYTE);
13. externalTotal = (externalStatFs.getBlockCountLong() \* externalStatFs.getBlockSizeLong()) / (KILOBYTE \* KILOBYTE);
14. externalFree = (externalStatFs.getAvailableBlocksLong() \* externalStatFs.getBlockSizeLong()) / (KILOBYTE \* KILOBYTE);
15. } else {
16. internalTotal = ((long) internalStatFs.getBlockCount() \* (long) internalStatFs.getBlockSize()) / (KILOBYTE \* KILOBYTE);
17. internalFree = ((long) internalStatFs.getAvailableBlocks() \* (long) internalStatFs.getBlockSize()) / (KILOBYTE \* KILOBYTE);
18. externalTotal = ((long) externalStatFs.getBlockCount() \* (long) externalStatFs.getBlockSize()) / (KILOBYTE \* KILOBYTE);
19. externalFree = ((long) externalStatFs.getAvailableBlocks() \* (long) externalStatFs.getBlockSize()) / (KILOBYTE \* KILOBYTE);
20. }
22. long total = internalTotal + externalTotal;
23. long free = internalFree + externalFree;
24. long used = total - free;
26. int percentAvailable = (int)((100 \* free) / total);
27. int percentUsed = (int)(used / total);
29. return mode == TOTAL ? total : mode == AVAILABLE ? free : mode == USED ? used : mode == PERCENT\_AVAIABLE ? percentAvailable : percentUsed;
31. }

*II.3.3 Hiển thị list ứng dụng đã cài đặt*

Ta lấy thông tin ở class PackageManager rồi chuyển vào adapter để hiện trên ListView

1. private void getListInstalledApp() {
2. final PackageManager pm = getPackageManager();
3. //get a list of installed apps.
4. List < ApplicationInfo > packages = pm.getInstalledApplications(PackageManager.GET\_META\_DATA);
6. for (ApplicationInfo packageInfo: packages) {
7. Log.d(TAG, "Installed package :" + packageInfo.packageName);
8. Log.d(TAG, "Source dir : " + packageInfo.sourceDir);
9. Log.d(TAG, "Launch Activity :" + pm.getLaunchIntentForPackage(packageInfo.packageName));
10. }
11. // the getLaunchIntentForPackage returns an intent that you can use with startActivity()
12. }

**II.4 Link github.com và Một số hình ảnh của ứng dụng:**

Link github: https://github.com/tr4n/BTL-OperatorSystem

|  |  |
| --- | --- |
|  |  |
|  |  |

1. **Tài liệu tham khảo**
2. Nguyễn Gia Định (chủ biên) Nguyễn Kim Tuấn – Nguyên lý hệ điều hành – NXB Khoa học và Kỹ thuật
3. Website: developer.android.com/