TÀI LIỆU THI CUỐI KÌ HỆ ĐIỀU HÀNH

Chương 4A: Tiểu trình

1. Khái niệm:

-Là 1 luồng thực thi trong 1 Process. 1 Process chia là nhiều tiểu trình và chúng có thể được thực thi đồng thời.

-Giúp Process thực hiện nhiều tác vụ đồng thời và đặc biệt hiểu quả trên hệ thống đa nhân, đa lõi(core).

-Lợi ích: tính phản hồi nhanh, chia sẻ tài nguyên(các thread chung process chia sẻ chung 1 vùng nhớ), giảm chi phí, khả năng mở rộng(threads có thể thực thi trên nhiều core, processors)

2. Các mô hình thread:

a. User-level Threads

-Được cài đặt bởi thư viện thread(API của ngôn ngữ LT) ở không gian người dùng không có sự hỗ trợ của kernel

-Ưu: hiệu quả(quản lí, chuyển đổi ngữ cảnh, đồng bộ không cần sự can thiệp của OS), không cần sự hỗ trợ từ hệ thống. Nhược: kernel không biết gì về user threads(Process sẽ bị block nếu 1 thread bị block, Không có sự công bằng trong Timesharing system khi cấp 1 lượng thời gian như nhau cho các process)

b. Kernel-level Threads

-Được hỗ trợ trực tiếp bởi OS và được cài ở Kernel space

-Ưu: OS quản lí thread(1 thread blocked, các phần khác của process vẫn được thực thi độc lập). Nhược: chậm và ít hiệu quả hơn vì cần sự can thiệp của OS vào các thao tác với thread.

c. Hybrid thread

-kết hợp 2 mô hình, có nhiều tiến trình ở user và OS tạo tiến trình ở Kernel

-ánh xạ tiểu trình từ User sang Kernel: Many-to-one (Nhiều tiểu trình mức người dùng ánh xạ sang một tiểu trình mức hệ điều hành). One-to-one (Một tiểu trình mức người dùng ánh xạ sang một tiểu trình mức hệ điều hành). Many-to-many (Nhiều tiểu trình mức người dùng ánh xạ sang nhiều tiểu trình mức hệ điều hành). Two-level (Hai cấp độ): Phối hợp giữa one-to-one và many-to-many

3. Quản lý tiểu trình

Thread table chứa thông tin về tiểu trình: ID, register, stack, state

User-level: thread table và các thao tác được quản lý bởi thư viện

Kernel-level: thread table và các thao tác được quản lý bởi OS

-**Lập lịch:** User-level: kernel lụm process và các thread thực thi theo thứ tự trong process. Kernel-level: OS lụm thread ở các process và thứ tự của thread có thể xen kẻ giữa các process.

-**Thư viện Thread:** cung cấp API cho tạo lập và quản lí thread

+POSIX thread(**Pthread**): hỗ trợ cả user và kernel thread

+Windows thread: hỗ trợ kernel threads được quản lí bởi WIndows

+Java thread: hỗ trợ lập trình thread Java

Chương 4B: Đồng bộ hoá

1. Vấn đề tranh chấp(Critical Section)

-**Critical Section(miền găng):** phần code mà ở đó các tiến trình, tiểu trình chia sẻ truy cập đồng thời. -> Giải quyết: Đồng bộ hoá

2. Đồng bộ hoá

a. **Các yêu cầu**: **Độc quyền truy xuất**(1 process/thread in CS at a time), **Có sự tiến triển**(1 p/th bên ngoài k thể block thằng khác vào CS), **Có giới hạn cho sự chờ đợi**(p/th ko nên đợi vô thời hạn để vào CS)

b. Busy waiting solutions

-**Biến Lock**: {while(lock); lock = 1; CS; lock = 0;}. 2 p/th có thể ở trong CS cùng 1 thời điểm

-**Strict Alternation:** {while(turn!=0); CS: turn = 1;}. 1 p/th ở ngoài CS có thể chặn cái khác vào CS(thằng chặn chưa có nhu cầu vào CS và chưa set turn)

-Peterson's solution: phối hợp 2 cái trên {int other = 1 – process; interested[process] = 1; turn = other; while (turn == other && interested[other] == 1); CS; interested[process] = 0;}. Lãng phí CPU wasting và Priority Inversion

-**Hỗ trợ từ phần cứng:**

+vô hiệu hoá Interrupt: cho phép tiến trình vào CS vô hiệu hoá tất cả các ngắt. Hệ thống sẽ bị nguy hiểm khi cho can thiệp, k hiệu quả trên đa nhân

+TSL(Test-And-Set): sử dụng chỉ thị Atomic, ko bị lấy cpu thì thực hiện Test-and-Set-lock để set lại biến Lock.

c. Sleep and Wakeup Solutions

-**Semaphore:** sử dụng biến int để hạn chế truy cập CS thông chỉ thị atomic **down**(gọi trước CS để kiểm tra) và **up**(gọi sau CS và wake up các th sleeping). Có 2 loại: Binary và Counting semaphores

down(semaphore *S) {S->value--; if (S->value < 0) {add calling thread to S->list; block();}}

up(semaphore *S) {S->value++; if (S->value <= 0) {remove a thread A from S->list; wakeup(A);}}

+Độc quyền: semaphore mutex = 1; pA (){ down(mutex); CS; up(mutex); }

+Đếm: semaphore count = 5; pX (){down(count); CS; up(count); }

+Thứ tự: semaphore mutex = 0; pA(){down(mutex);...} pA(){...up(mutex);}

-**Monitor:** cơ chế giống Semaphore, cấu trúc lập trình bậc cao(biến dùng chung, thủ tục, biến điều kiện(đi kèm wait and signal). Được gọi là thread-safe class: chỉ có thủ tục trong monitor mới có thể truy cập biến, chỉ có 1 thread ở trong monitor ở 1 thời điểm.

3. Bài toán đồng bộ hoá

a. **Producer-Consumer Problem:** chia sẻ 1 cái bộ đệm cố định. Producer đẩy item vào bộ đệm (Full -> blocked), Consumer lụm item từ bộ đệm (Empty->blocked) và chỉ có 1 đứa đc truy cập cùng lúc.

b. **Readers-Writers:** share 1 database. Vài reader đc cùng lúc đọc, nếu có 1 writer mấy thằng khác k đc, nếu đã có bất kì reader thi writer k đc vào

c. **Dining Philosophers:** 5 ông ngồi bàn tròn, có 5 nĩa xen kẻ, nhưng mỗi ông cần 2 cái(mỗi thời điểm chỉ đc lụm 1 cái và k thể lấy nếu đã bị lấy)

Chương 5: Deadlock

1. Giới thiệu

a. **Resource-Allocation Graph (RAG):** Đồ thị không có chu trình->**no deadlock**. Đồ thị có 1 chu trình và tài nguyên có 1 instance->**deadlock.** Đồ thị có 1 chu trình và tài nguyên có nhiều instance->**có thể deadlock**

b. **Định nghĩa:** khi mỗi p trong 1 tập process phải **chờ đợi tài nguyên** mà **đang được cấp** cho thằng p **cũng đang đợi**. Do đó không có p nào có đủ tài nguyên để tiếp tục thực thi

c. **Điều kiện để có deadlock:** có đủ: **Độc quyền truy xuất**, **Giữ và chờ**(giữ 1 phần tài nguyên và chờ tài nguyên khác), **Không chiếm đoạt**(tiến trình tự nguyện trả tài nguyên), **Vòng lặp chờ**(giữa ít nhất 2 p)

2. Xử lí Deadlock

a. **Ignorance:** mặc kệ bỏ qua, hiếm khi có deadlock và chi phí xử lí nó tốn kém

b. **Prevention:** phòng ngừa. Ngăn chặn 1 trong 4 điều kiện:

-**Độc quyền truy xuất**(không thực tế)

-**Giữ và chờ**(P phải yêu cầu all tài nguyên khi bắt đầu thực thi hoặc release cái đang giữ trước khi yêu cầu mới): Hiệu suất bị giảm, khó để biết tất cả tài nguyên cần và có thể có starvation.

-**Không chiếm đoạt**(OS có thể lấy lại tài nguyên trong 1 số trường hợp): có thể gây lỗi cho một số tài nguyên (printer, files)

-**vòng lặp chờ:** Gán độ ưu tiên cho tài nguyên và cấp phát theo thứ tự

c. **Avoidance:** tránh deadlock: kiểm tra khả năng xảy ra deadlock khi cấp phát tài nguyên, nếu có thì không cấp phát. Sử dụng thuật toán: **RAG**(tài nguyên 1 đối tượng), **Banker's algorithm**(tn nhiều đối tượng)

-**Hệ thống an toàn:** tìm được 1 chuỗi cấp phát an toàn để cấp phát tài nguyên và thực thi các tiến trình

-**Banker's algorithm: Available:** tài nguyên sẵn có. **Max:** tài nguyên tối đa mà mỗi p yêu cầu. **Allocation:** tài nguyên đã cấp cho mỗi p. Need: tài nguyên mà p sẽ cần: **Need = Max - Allocation**. **Request:** yêu cầu cấp phát của p. Cách tìm chuỗi an toàn: so Available với bảng Need, tìm p có thể cấp, cập nhật lại bảng Available với Allocation của p đó được trả lại, qua lại bước đầu. **Request < Need < Available**

d. **Phát hiện và khôi phục:** để deadlock xảy ra và khôi phục hệ thống

-Phát hiện: **wait-for graph**, **matrix-based** detection algorithm(biến thể banker)

-Khôi phục: Tắt các p bị deadlock, quay lui hệ thống về vị trí an toàn, thu hồi và cấp phát lại tài nguyên.

Chương 6: Memory

1. Khái niệm cơ bản

a. **Không gian địa chỉ:**

-**Địa chỉ ảo(logic)** được tạo bởi CPU. Virtual address space: tất cả địa chỉ logic(không gian địa chỉ của chương trình).

-**Physical address:** địa chỉ thật sự trên bộ nhớ vật lý(or main memory). Physical address space: tất cả địa chỉ vật lí (memory address space)

b. **Address binding:** ánh xạ 1 không gian địa chỉ này sáng 1 các khác. Có thể thực hiện qua thời gian biên dịch, tải(OS xưa) hoặc thực thi(hiện nay).

c. **Memory Management Unit(MMU):** 1 phần mạch tích hợp vào CPU để làm nhiệm vụ chuyển đổi địa chỉ logic đến địa chỉ vật lí

d. **Memory Management:** cấp phát/thu hồi vùng nhớ, Binding(tái định vị địa chỉ), bảo vệ và chia sẻ vùng nhớ.

2. Chiến lược cấp phát vùng nhớ liên tục:

a. **Tính chất:** Các tiến trình không thể bị phân nhỏ và phải được load vào một phần vùng nhớ liên tục trên main memory. Main memory có thể được chia sẵn các phân vùng hoặc được chia khi các tiến trình được nạp vào.

b. **Fixed-partitions:** Các phân vùng sẽ được chia sẵn và 1 tiến trình sẽ được nạp vào 1 phân vùng còn trống. Kích thước có thể ko phù hợp với process

c. **Variable-partitions:** các phân vùng được tạo khi process được load. **Hole**: các block ô nhớ trống. Phân vùng sẽ được cấp fit với kích thước process.
d. **Chiến lược cấp phát bộ nhớ: First-fit:** process sẽ được đặt vào vùng trống đầu tiên có thể chứa nó. **Best-fit:** process sẽ được đặt vào vùng trống nhỏ nhất có thể chứa nó. Cần tốn thời gian cho tìm kiếm. **Worst-fit:** process sẽ được đặt vào vùng trống lớn nhất có thể chứa nó
e. **Bảo vệ địa chỉ:** MMU sử dụng 2 thanh ghi: **Relocation register**(base register): lưu lại địa chỉ physic nhỏ nhất của vùng nhớ của process. **Limit register:** lưu trữ giới hạn địa chỉ của vùng nhớ. Địa chỉ logic kiểm tra < limit -> cộng với relocation = địa chỉ vật lí

3. Swapping:

a. **Nguyên tắc:** Process có thể chuyển vào hoặc ra giữa memory và backing store (fast disk)(1 phần trên ổ đĩa có thể truy xuất nhanh).
b. **Quản lí vùng nhớ với Bitmap:**Bộ nhớ được chia thành các khối (bytes->KB), mỗi chúng sẽ được quản lí bằng 1 bit trên bitmap(0-free, 1-occupied)
c. **Quản lí với LinkedList:** các segment sẽ chứa vị trí bắt đầu, độ lớn và kí hiệu **H**(free segment hay hole) hoặc **P**(chiếm bởi 1 process)
d. **Swap time:** có thể ảnh hưởng tới tốc độ của hệ thống

4. Phân mảnh(Framentation):

a. **External Fragmentation(ngoại vi):** thường có khi cấp phát động, khi các vùng nhớ trống đủ để nạp tiến trình nhưng lại không liên tục. Solution: dồn
b. **Internal Fragmentation(nội vi):** thường xảy ra khi vùng nhớ đc chia sẵn. Khi kích thước của phân vùng không phù hợp với process->ko gian trống nằm trong không sử dụng được. Sln: best-fit

Chương 7: Virtual Memory

1. Khái niệm cơ bản

a. **Định nghĩa cơ bản: Dynamic linking**: linking delay đến giai đoạn thực thi.
Dynamic loading: đoạn mã không được load cho đến khi nó được gọi.
Overlays: 1 số phần của process mà đang được thực thi mới đc load vào
b. **Virtual Memory:**Cho phép swap out 1 số phần tiến trình ra backing store
c. **Non-contiguous Memory Allocation:** kĩ thuật cấp phát không liên tục
-Process đc chia thành nhiều phần khác nhau và đc load vào các phân vùng có thể không liên tục. 2 công nghệ cơ bản là: **Segmentation và Paging**

2. Segmentation: phân đoạn

a. **Nguyên tắc:** 1 tiến trình được chia thành các segments(các đoạn nhiều kích thước khác nhau tuỳ vào ý nghĩa luận lí) và đc load vào các phân vùng không liên tục trong bộ nhớ chúng. 2 thanh ghi đi kèm đoạn: **base(**địa chỉ đầu tiên của vùng nhớ vật lí của đoạn**)** and **limit(**độ dài segment**).**
b. **Ánh xạ địa chỉ và Bảo vệ**: Địa chỉ logic gồm **<s: segment number, d; offset>** (offset: địa chỉ tương đối trong đoạn, khoảng từ 0 tới limit - 1)
-**Segment Table:** dùng để ánh xạ segment-memory. Từng dòng trong bảng chứa base và limit của 1 segment.
-**<s, d> -> d < s.limit ? -(y)-> physical Add = s.base + d**

3. Paging: phân trang

a. **Nguyên tắc:** Process được chia thành các page bằng nhau cố định do phần cứng quy định. Memory được chia thành frames có kích thước cố định = page size. Mỗi trang sẽ được load vào khung trang.
-địa chỉ **logic** <p:page number, d:offset> (offset: địa chỉ tương đối trong page)
-địa chỉ **vật lý**: <f: frame number, d: offset>
-địa chỉ logic: có thể được thể hiện: **<p,d>**, **giá trị tuyệt đối**
-vd: 16bit logic, 12bit vật lí, 4bit cho offset->12b cho page, 8b cho frame, địa chỉ theo byte -> page size = 2^4 byte, logic space = 2^16 byte, physic = 2^12
-**Page Table:** dùng để ánh xạ page-frame. Mỗi dòng gồm **page number** và **frame number** mà page đang được nạp. Ngoài ra còn 1 số bit thông tin: **Valid/Invalid**(page có trong memory ?), **Protection**(1: read-only, 0:read /write) , **Caching**(có quyền caching? 1: disable, 0: enable), **Referenced**(có được truy cập gần đây?), **Modified**(có bị thay đổi mà chưa đc cập nhật)
-**Demand Paging:** chỉ load các trang mà đang đc thực thi bởi CPU vào memory. Swapping zone(back up zone): 1 phần ổ đĩa để chứa all pages of p.
-Ánh xạ: **<p,d> -> pageTable[p].validBit = 1 -> physAdd = <f, d>;**
<p,d>->pageTable[p].validBit = 0 -> lỗi trang -> load page -> reset PageTable
-**EAT=(1-p)* tm + p * tp** (p: tỉ lệ lỗi, tm: memory-acc time, tp: tg xử lí lỗi page)
-**TLB:**bộ đệm tìm trang, hỗ trợ truy xuất nhanh chứa thông tin page quantrọng
-**EAT = h * (tc + tm) + (1-h)*(tc + (1+ nTLBs)*tm)**

b. **Thuật toán thay thế trang:**
-**FIFO:** page nào vào trước thì được thay thế trước, đơn giản nhưng nguy hiểm
-**Optimal**: nhìn về tương lai, thay thế page được truy cập xa nhất, hiệu quả nhưng không thực tế, không thể biết trước tương lai
-**LRU**:nhìn về quá khứ, thay page ko đc truy cập xa nhất.Khả thi, cần hardware
-**Second Change**: Nâng cấp FIFO, cho cơ hội thứ 2 nếu ref bit là 1

c. **Phân trang với không gian dữ liệu lớn**
-vấn đề khi nạp bảng trang có kích thước lớn vào bộ nhớ và tìm kiếm trên nó
-**Phân trang đa cấp:** có pageTable cấp cao trỏ tới các page table thấp hơn
-**Hashed Page Table**: số trang được hash để lưu trong hash table
-**Bảng trang nghịch đảo:** index là frame, mỗi dòng trong page table chứa thông tin của frame tương ứng(chứa pid của process và page number)

Chương 8: File System

1. File system interface

a. **Khái niệm File:** khái niệm trừu tượng, đơn vị lưu trữ logic của 1 tập dữ liệu trên đĩa. **File types**: ASCII or binary files, thư mục, Shortcut , special file (vd: device file). **Thuộc tính:** (metadata) name, size, location, creator, date(tạo, truy cập, chỉnh sửa), type, permission/protection,...
b. **Thao tác file:** fd=creat(name,mode): tạo file mới; fd=open(file,how,..): mở file; s=close(fd): đóng file; n=read(fd,buffer,nbytes): đọc file vào buffer; n=write(fd,buffer,nbytes): ghi buffer vào file; pos=lseek(fd,offset,whence): di chuyển file pointer; s=stat(name, &buf), s=fstat(fd, &buf): lấy thông tin trạng thái; s=pipe(&fd[0]):tạo 1 pipe; s=fcntl(fd,cmd,...):file locking và thaotackhac
c. **Cách truy cập:** tuần tự, truy cập trực tiếp ngẫu nhiên, tuần tự theo Index
d. **Cấu trúc file:** 1 chuỗi unstuctured bytes/words(modern OS), chuỗi records có fixed-length(in mainframes), Cây records(large system cho tài chính)
e. **Thư mục:** tập files/thư mục con, **Path types:** tuyệt đối và tương đối, **Thuộc tính:** name(ko đuôi), path, creator, date, permission/protection.Thao tác thư mục: mkdir,rmdir,link,unlink,chdir,opendir,closedir,readdir,rewinddir
f. **Tổ chức Tmuc:** tiêu chí: hiệu quả, đặt tên, khả năng grouping, chia sẻ
-**Single-level:** Implement, Searching tốt; Naming, grouping dở
-**Two-level:** mỗi user 1 single dir: Naming, Searching ok, Grouping dở
-**Tree-structure**:Naming, Grouping, Searching ok;Truy cập,trùng tên, chia sẻ dở
-**Graph**: Linh hoạt, chia sẻ ok; nhưng tốn chi phí
g. **Protection:** read(R), write(W), execute(X), 3 lớp user: owner, group, universe

2. File System Implementation

-**File System**: 1 phần OS làm việc với file, cung cấp khả năng truy cập file trừu tượng và quản lí, ánh xạ files trên đĩa (cấp/thu, đọc/ghi, bảo mật, chia sẻ)
-**Secondary Storage**: Đĩa cứng được chia thành nhiều phân vùng, cần cài đặt file system trên mỗi phân vùng: NTFS(Windows); ext2,ext3,ext4(Linux); HFS, HFS+, APFS(macOS); FAT32, exFAT(flash disk, removable device)
-**FAT/exFAT**: File Allocation Table đc dùng ngày xưa Win, FAT12/16/32, FAT32 phù hợp với nhiều OS (file < 4GB), exFAT (max file 16EB) cần thêm software
-**NTFS:** NewTechnologyFileSystem, ngày nay win, Có tính năng nâng cao: nhật kí, backup & restore, bảo mật file/dir, nén, mã hoá và giải mã, max; 16EB
-**Unix-based:** ext2/3/4, ext4 đc dùng làm default trên modern Linux, nhật kí (checksum), 16GB-16TB, 1 dir có thể có max 64000 subdir

a. **File và Dir implementation:**
-**Cấp phát liên tục:** file được lưu trên các block liên tục trên đĩa. Dễ cài, hiệu năng cao; bị phân mảnh, ko đủ cho file lớn và phải biết final file size khi tạo
-**Cấp phát LinkedList:** lưu trên các block k liên tục, mỗi block chứa data và con trỏ tới block tiếp theo. Ko phân mảnh, linh động; Ko truy cập random trực tiếp, cần space cho pointer và nguy hiểm khi mất 1 block
-**Cấp phát index**:lưu trên các block k liên tục: data và 1 block để lưu con trỏ đến data. Truy cập random, solve mất block; dư thừa và k hiệu quả với file nhỏ
-**FAT:** nâng cấp Linked list: con trỏ của các phần tử lưu trong 1 cái bảng
-**I-Nodes**: Index đa cấp, Direct block trỏ tới data, Indirect trỏ tới direct block(single) hoặc các Indirect thấp hơn. Linh hoạt cho file lớn và nhỏ
-**Quản lý Free-space**: Bitmap: 1-freeblock, 0-đã cấp, đơn giản, hiệu quả, but phải load bitmap. Linked list; duyệt dở. Grouping(n-free-block), Counting
-**Cài đặt Dir**: Linear Linked-list of filenames. Hashed table để tăng searching

b. **File System Structure**
-**Master Boot Record**: nằm ở sector đầu tiên của đĩa, chứa thông tin cho OS booting và chương trình đọc boot sector record của phân vùng chứa OS

3. Disk Storage

a. **Disk Management**
-**Platter** là lớp đĩa có 2 **Head** là 2 mặt trên dưới, Mỗi vòng tròn là 1 **Track,** tập hợp track cùng bán kính là **Cylinder**. 1 Track chia thành các **Sector**(min unit)
-Head từ 0 từ trên xuống, Track từ 0 từ ngoài vô, Sector từ 1 ngược chiều quay
-đĩa cần format trước khi sử dụng: chia sector, chia phân vùng, cài filesys
-**Truy cập sector:** CHS và LBA. LBA = (C * Nheads + H) * Nsectors + (S – 1).
C=LBA/(Nheads*Nsectors);H=(LBA/Nsectors)%Nheads;S=(LBA%Nsectors)+1

b. **Disk Sheduling Algorithms**
Disk Access Time = Seek Time + Rotational Time + Data Transfer Time
Seek Time tốn nhiều nhất là tgian tìm track
-**Các thuật toán lập lịch trên đĩa**
FCFS(First-Come First-Served). **SSTF**(Shortest Seek Time First) tìm thẳng gần nhất để phục vụ. **SCAN**(thang máy): đi từ đầu tới đuôi, gặp thì phục vụ.
C-SCAN (Circular SCAN): giống SCAN nhưng chỉ phục vụ một chiều
LOOK (cải tiến SCAN): chỉ đi tới thẳng xa nhất. **C-LOOK** giống LOOK nhưng phục vụ 1 chiều