Exercise 1 : Process Instruction Execution

↳ * Registers 和它们的作用 ✓

* Process Instruction Set (基本的) ✓

* Instruction Execution Cycle ✓

* Interrupt 和 Context Switch 出现之后的 Instruction cycle ✓

* Function Call ( ) ✓

* 和这些相关的 Assignment 题目

Register: High speed memory

For Addressing:

MAR (Memory Address Register): Holds the memory location of data that needs to be accessed

MDR (Memory Data Register): Holds data that is being transfered or from memory

IP (Instruction pointer a.k.a PC (Program counter)): Holds the memory address of the instruction to be executed next

CIR (Current Instruction Register a.k.a IRC instruction register)): Holds the memory address of the instruction that is being 「executed」 or 「decoded」

SP (Stack pointer): Pointer to the top of the stack (指的是整个进程的 Stack)

↳ 并不是有一个真正意义上的 "stack", 而是当一个东西被放入 stack 中时, stack pointer 先修改自己指的地方, 再把东西存入 (自下而上的 stack)

↳ Stack 是用来存放 stack frame 的, 在 stack frame 中存放的是一个 function call 的 {return address, return value, parameters, local variables}, 一个进程可以 call 很多个 function, 每个 function 都有个 stack frame (栈帧)

BP (Base pointer): Points to the last closed stack frame

↳ EBP (base pointer) 位于当前帧的帧底, 这个 EBP 存放的东西是上一级调用者的帧底 (EBP) 的位置

注: EBP 的前一个单元存放的是当前函数的 return address

↕ BP 中只存一个, 但用位置来 access

Used to reference element within frame of the function

For general purpose:

AC (Accumulator): Holds intermediate arithmetic and logic result

↳ 计算中间产生的结果, 无需把这些结果存入内存

Base register: Holds the smaller legal physical memory address

Limit Register: Specifies the size of the range that the program was allocated

CX (Counting register): Store the loop count in iterative operations

Processor Instructor Set

MOV Ax, 0x1B2J	:	Move the value 0x1b2J to the register Ax
MOV Al, 0x2J	:	Move the value 0x1b2J to the lower part of register Ax
Inc Ax	:	Increments the value in the register Ax by 1
Dec Ax	:	Decrements the value in the register Ax by 1
Push Ax	:	Pushes to content of Ax into the stack, SP-2 (更新栈顶)
Pop Ax	:	Pops the content pointed at [SP] in the stack into Ax
Jmp T	:	Jumps to the instruction labeled with T
Cmp Ax, 1	:	Compare the values stored in the Ax register with 1
Je T	:	Jumps to T if ⇓ were equal
Jle T	:	Jumps to T if Ax ≤ 1
Loop T	:	Decrement Cx and jump to T is Cx ≠ 0
Xor Ax, Bx	:	Xor of Ax and Bx and stores result in Ax
Halt	:	Terminates program

Instruction Execution Cycle:

1 FETCH "Getting program's next instruction from memory"

IP → MAR

1.1 Get instruction address from IP, place it in the MAR and send a read request to RAM to retrieve the content of MAR

get content

[MBR]

1.2 The content retrieved is placed in MBR

→ CIR

1.3 Store the instruction code in CIR

2 DECODE "Instruction decoding and operands fetching"

[CIR]

2.1 The CU (control unit) decodes and transforms the instruction into a sequence of elementary operations

↓ decode

IP updated

2.2 If this instruction require more operands, CPU issue 一个新的 FETCH and get them from MBR

2.3 The operand is stored in one of the general purpose register and the IP is updated

3 EXECUTE "Instruction Execution"

3.1 The ALU (Arithmetic logic unit) executes the instruction

3.2 The state register is updated

* 一些看图时需注意的点...:

* 当指令被放入 decoder 中时, IP 就被更新

* 当 operand 被放入 register 中时, MAR 可以更新为 IP 的值

↳ 先 MAR → MBR → CIR

(MAR: 放地址; MBR: 放具体)

* 在做 Arithmetic 时, MAR 会有变化吗???

如果指令是 Mov Ax, [0x AJ22]

Mov Bx, [0x AJ23]

Add Ax, Bx

* Add Ax, Bx : 把 Ax 和 Bx 的值相加, 结果存入 Ax

Instruction Cycle with Interrupt and context switch

↳ 中断 Vs 系统调用：中断是指，在程序执行过程中，中断会改变 CPU 原本的执行顺序
系统调用通过中断 向 内核提出请求

↳ Interrupt with Context Switch

↳ Assuming that Process P_1 is executing and process P_2 is the next process in the ready queue

If an interrupt occurs:

1. CPU pushes the current PC onto the stack and updates the PC to contain the address of the 1st instruction in the corresponding interrupt handler
2. Interrupt handler start execution:
   - 2.1 It pushes all remaining registers onto the stack(P1) then perform its specific task
   - 2.2 It copies all registers values from the stack (P1) into PCB1
     ⇧ 相当于更新了 PCB1
   - 2.3 It move PCB1 to end of ready / blocked queue, and PCB2 into the head
   - 2.4 It copies register values from PCB 2 into CPU-registers, except the PC values. PC values 会通过新的 SP 找到新的 stack 之后放入 stack中
3. The handler executes RTI (Return from Interrupt) which causes the CPU to pop the PC value from the New stack to CPU's register
4. Then CPU resume/start execution of process P_2

注意 更新 PCB 的任务是 interrupt handler 来完成的
RTI 的时候才把下一个要执行的从 stack 中 load 进 PC

Interrupt without context Switch

Assuming that Process P_1 is executing and a keyboard interrupt occurs:

1. CPU pushes the current PC onto the stack and updates the PC to contain the address of the 1st instruction in the corresponding interrupt handler
2. The interrupt handler starts execution by pushing all other registers onto Stack
3. Performs its specific task that modifies the CPU registers content
4. Pops back the registers value from stack to CPU
5. The handler executes RTI (Return from Interrupt) which causes the CPU to pop the PC value from the New stack to CPU's register
6. Then CPU resume/start execution of process P_2

h d f r q

有 context switch 和没 context switch 的区别在于

↳ 有 context switch 时，handler 负责更新 PCB，并且把 新 PCB 的值放入 register
没 context switch 时，handler 不更新 PCB，且 SP 值不变，恢复的时候从 Stack 中拿 register 的值

Function Call (1)

* Stack : Pg 1 有
* Heap : is different from stack 在 OS Security 中有更详细 heap 解释

↑

Stack 栈内存	VS	Heap 堆内存
操作系统自动分配给进程的，每个 function call 都会在 stack 中加入一个新的 stack frame (栈帧)		由程序员释放，若程序员不释放，程序结束时可能被系统收回，分配方式类似链表 (linked list)

*Return address for a function: tells the function where to resume executing after the function is completed.

总之要记住：{ 1. reverse order
2、BP永远跟在 return address 后

* Function Call:

1 Before executing a function, a program pushes all of the parameters for the function onto the stack in reverse order so that they are documented.

2. Then the program issues a call instruction

2.1 Call instruction pushes the address of the next instruction, which is the return address, onto the stack

2.2 Then it modifies the instruction pointer to point to the start of the instruction

3 Executing the function

4 Done executing

4.1 Stores its return value in EAX register

4.2 frees the stack space it allocated by adding the same amount to the SP

4.3 Pops off base pointer

4.4 resets the stack to what it was when it was called (it gets rid of the current stack frame and puts the stack frame of the caller back into effect)

4.5 return control back to whereever it was called from; pops whatever value is at the top of the stack, and sets the instruction pointer to that value (C这就是为什么 buffer overflow 会发生)

*在 function 刚结束时，the return address 并不在 top of the stack

∴ We need to move the stack pointer to the current stack frame base pointer and restore the caller's frame pointer first (所以现在 SP 指向 BP 指向上一级调用者)

Exercise 2: Understand process creation using fork() and difference with Thread

Difference with Thread:

1. 存储
2. 创建
3. 逻辑控制 (被创建时)

不管是线程还是进程，都可以用以下两个角度去理解：

1. 逻辑控制流，表示一种计算过程
2. 独立的虚拟内存地址空间

什么是 process？

↳ Process: An instance of a computer program that is being executed.

↳ 和 Program 不一样，program 是一个 passive entity (只是一个包含了代码的文件)

但 Process 是一个 active entity！它包含：

↳ 包含 (存储角度的解释)

1. An address space { static, stack, heap }
2. CPU State (values of CPU registers including PC and SP)
3. A set of OS resources: open files, network connections, ..
4. A lifetime (生命线，即时间对这个进程是有影响的)

↳ created - executed - [interrupted - resumed] - terminated

* Process 被 PID (Process Identifier) 唯一代表
* 每个 Process 被 PCB (process control block) 所表示，以 PID: | PCB: 的形式存在 process table 中

创建：fork()

↳ 系统调用 fork()，来创建子进程，这个进程：

逻辑控制流：fork() 创建的子进程从父进程执行 fork() 的下一个位置开始执行

独立的虚拟内存地址空间：fork() 创建的子进程会复制父进程的虚拟地址空间，也就是说 OS 真的会开一个新的位置并把父进程储存的内容完全复制给子进程

1. `fork()`: By executing the primitive `fork()` in a C-program, the involved process makes a system call to the kernel through the POSIX API asking the it to create a new process (child process). The new process starts its execution from the instruction statement that is just after the `fork()` system call that created it. At the same time, the primitive `fork()` returns a value n of type `pid_t`. The parent process will have the value $n > 0$ (equal to the PID of the created child process), whereas the created child process will have $n = 0$. If $n = -1$, then `fork()` has failed its execution. (fork会返还一个pid_t的值n，如果是执行了fork()的父进程，他会收到的是子进程的PID(n > 0), 如果是被创建的子进程，会收到的是0)
2. `getpid()`: This primitive returns the value of the PID of the process that executed it. Recall that each process has a unique identification number called PID (Process IDentification number) that is used by the system to identify the process.
3. `getppid()`: This primitive returns the value of the PID of the parent of the process that executed it. Recall that each process has a parent process that created it (Except init which has PID=1).
4. `exit(v)`: Terminates the process which executes the primitive and returns to the parent process a one-byte integer value contained in the value of v.
5. `wait(&status)`: When executed by a parent process, the later waits for its child process termination notification. When a child process terminates, the primitive `wait()` returns the PID of the child process which terminated. Also, it returns from the address status (`&status`), the value sent by the exit primitive (see the value of v in `exit()`) and the cause of the termination of its child process (all in one-byte). Basically, the most significant byte contains the value of v, and the less significant byte carries the cause of the termination. You can use the macro `WIFEXITED(status)` to retrieve the cause (1 normally terminated, otherwise abnormally terminated) and `WEXITSTATUS(status)` to retrieve the value of v. Finally, if there is no child to wait for, `wait()` returns -1.

什么是 Thread?

↳ Thread: An execution flow of a given program. A Thread is a lightweight process.

↳ Thread 是基于 Process 进程下的，一个进程下可以有多个线程

线程和其进程共享的内容(有点...像进程父类继承给线程子类的东西):

Code section, global data, heap, resources such as open files

各个线程自己独立拥有的内容:

stack, registers

* Thread 被 TID(Thread IDdentifier)唯一代表

* 每个 Thread 被 TCB (Thread control block) 统表示，以 TID: | TCB: 的形式存在 Thread table 中

创建: clone()

系统调用 clone() 来创建线程.

逻辑控制流: clone() 创建的线程可以自己定义. 比如 Java 的 Thread 的 run 方法位置

独立的虚拟内存地址空间: clone() 创建的线程共享进程的 code section, global data, heap, resources such as open files, 但有自己的 stack 和 CPU register 值

∴ 总结起来. Thread VS Process

1. 进程用 fork 创建，线程用 clone 创建
2. 子进程的逻辑流(开始执行)在 fork() 的下一条指令，线程的逻辑流位置则在 clone 调用传入的方法，比如 run
3. 独立的虚拟内存地址空间

Exercise 3: Process Synchronization: Writing sychronization code using semaphores and writing CoBegin / CoEnd

1. Writing Synchronization code
2. CoBegin / CoEnd

相关知识:

* 并发 (concurrency) vs 并行 (parallelism)

指能处理多个同时性活动的能力，并发事件不一定要同一时刻发生

指同时发生的两个并发事件，具有并发的含义，但实际上是同时发起

并发不一定并行，并行一定并发

* Process Synchronization的整个概念就起源于 The corruption of shared data by among several processes

* Critical section: Part of a program code in which the program requests to use shared resources on which the access is mutually exclusive.

↳ 如果要解决这个问题，设计出的 protocol 必须满足以下三个条件:

Mutual Exclusion: Only one process can enter its critical section (manipulate one shared variable) at a time, 就是说 不可以有超过一个进程在同一时刻修改 shared variable. (互斥)

Progress: When one process leave its critical section, it must inform other waiting process so they can start executing their critical section. Also, a process that is not in their critical section can't block other process.

Bounded Waiting: A process must not wait indefinately for getting into their critical section.

Protocol for Critical Section problem: Mutex lock, Semaphore, Hardware Solution

使用相同的原子操作，但意义是完全不一样的，具体的看 markdown 笔记

Atomic Operation / Primitives: "原子操作是不需要synchronization的", 这种操作是不会被线程调度机制所打破，这些操作，它们依赖底层 CPU 的原子操作实现，因此 原子操作是不会被打断的

acquire() (or P() or wait())

```
acquire() {
    while (S <= 0);
    S = S-1;
}
```

凡是在这一步把进程困在 while loop (即占用了一个CPU) 的，都是 busy waiting 在硬等待

release() (or V() or signal())

```
release() {
    S = S+1;
}
```

S 指的就是 Semaphore，在对原子操作下，一个进程想进入其 CS，必须先进行这一个检查，如果 Semaphore <= 0，代表这个进程必须等在 while 中，当 S > 0 时，它会把这个 semaphore 抢走 (S = S-1)。在它执行完它的 CS 之后，它不再需要这个 semaphore 了。于是它执行 release() → S = S+1 去释放这个 S

可见互斥锁可以管理多个资源，binary semaphore只能管理一个资源

Mutex和Binary Semaphore的区别：Binary semaphore的标准定义是范围从0-1的之间的信号量，但是Mutex lock的定义是一个互斥锁，他解决的问题是让两个想占用同一个资源的进程互斥，但是在语义上是有差别的，因为互斥锁管理的是资源的使用权：我有这个互斥锁，所以我可以用，你没有，所以你要等我给你了你才能用；而信号量管理的是资源的数量，我在进入critical section之前，ok现在信号量是10，还有十个可用资源，我拿了一个，信号量对你来说变成了9，你接着拿，当他拿的时候，只剩下0个了，那他去queue等一等，并且先宣布对下一个释放的资源的使用权：信号量变成了-1。所以互斥锁和信号量是有本质语义上的区别的。

解释为什么要>=1

两种写 Synchronization code 的 style：1. Mutual Exclusion style → Sinit ≥1 (为了互斥)

2. Process waiting style → Sinit = 0 (为了让进程遵照某个特定的顺序)

↳ 不一定，midterm 2 有把 semaphore initialize为 -1 的出现

Example (Mutual Exclusion style):

↳ 一个加油站，6个 pump，一堆车 vehicle 要来加油

```
vehicle i ;   Semaphore pump = 6;
Begin
    acquire (pump);
        Pump- gas ( );
    release (pump);
End
```

Waiting style:

↳ P_2 在 P_1 前执行

```
Semaphore S = 0
P1 :                P2 :
acquire (S);        code 2;
code 1 ;            releaseCS) ;
```

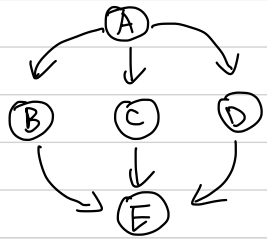
↳ 组，为了解决 busy waiting 这个问题，在 acquire ( ) 这个指令中，把 while (S ≤0); 改成 block P & place P in waiting queue (把当前进程 block 掉，从而这个没干任何事情的 process 在一边呆着，不占用资源)，但是，虽然 S2=0会进入 Queue，但进入 Queue 的进程依然会把 Semaphore 的值减去1，这样是为了表示"我已经在排队等这个信号量了"，所以自然地，在 release，中定义的就是在一个进程执行完 Critical Section 之后会把等在 queue 中的进程给唤醒且释放一个信号量

Peterson's Algorithm

```
do {
    flag[i] = true;
    turn = 1 - i;                  在 1-i 中设置
    while ( flag [1-i] && turn == 1-i );
            CS
    flag [i] = false ;
    ... } while (true);
```

也就是说，当下的进程会把自己的flag设为true，这暗示着自己是ready去执行critical selection的，但是同时他把turn设置为j，也就是说如果此时j也ready了，那么j会先运行，当j运行完了flag[j]会变成false，也就在此时，i会运行自己的critical selection，运行完了之后flag[i]会变成false，接下来就轮到j，一直一直搞，但在这个里面是存在busy waiting的

Process Precedence Graph for Synchronization



=> A → B 代表着 A必须在 B之前完成

在 Begin 与 End 中有序进行的 相邻的 同级别 进程中 若是有线的

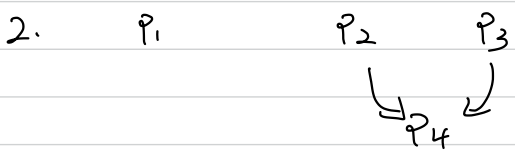
CoBegin / CoEnd Code:

```
Begin
    A
    CoBegin  B , C , D  CoEnd
    E
End
```

同级别 → A对B,C,D都有线

→ 在 Begin 与 End 中间的是有序进行的

→ 在 CoBegin 与 CoEnd 中间的是同时进行的

2. 

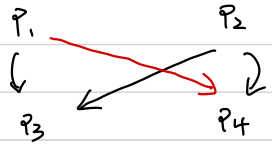
=>

```
CoBegin
    P1
    Begin
        CoBegin  P2 ; P3  CoEnd
        P4
    End
CoEnd
```

对掉一张图，要找到它们的 Precedence Constraint，

这里的 Precedence Constraint 是：P1，P2，P3 必须同时进行，

P4必须在 P2 和 P3 结束后进行；P4 可以在 P1 之前，之后，同时进行

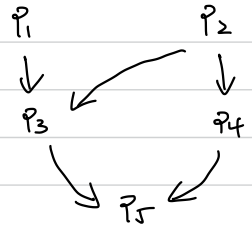
3. 

Constraint： P1 和 P2 同时进行，P3 和 P4 同时进行，P3 必须在 P1 和 P2 结束后进行，P4 必须在 P2 结束后进行

```
Begin
    CoBegin  P1 ; P2 ; CoEnd
    CoBegin  P3 ; P4 ; CoEnd
End
```

这样的话无形之中给 P1 到 P4 加了线，但整个 code 的主要的意义在于不打破 constraint，有时候必须加线

Synchronization Code:



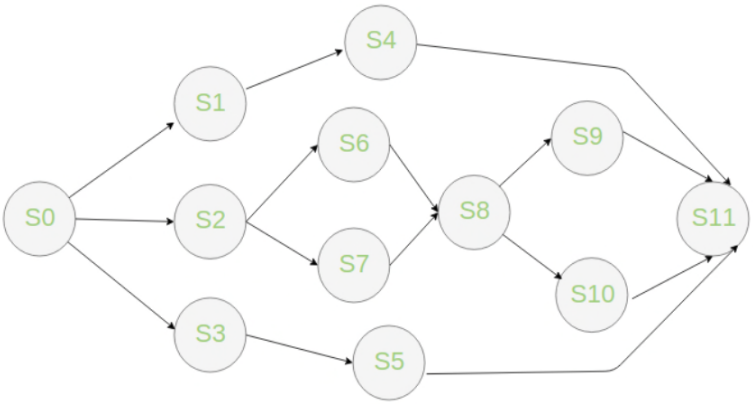
Translate using semaphore:

```
Semaphore S1 , S2 , S3 ;

P1 :                P2 :
    Code 1 ;            Code 2 ;
    V(S1) ;             V(S2) ;
                        V(S2) ;

P3 :                P4 :
    P(S1) ;             P(S2) ;
    P(S2) ;             Code 4 ;
    Code 3 ;            V(S4)
    V(S3) ;

P5 :
    P(S3) ;
    P(S4) ;
    Code 5 ;
```

例題：

A1：

Using (Begin-End) for sequential executions and (ParBegin-ParEnd) for parallel exe-cutions, give the pseudo-code for the following PPG (Process Precedence Graph), whereS0is the first instruction statement:

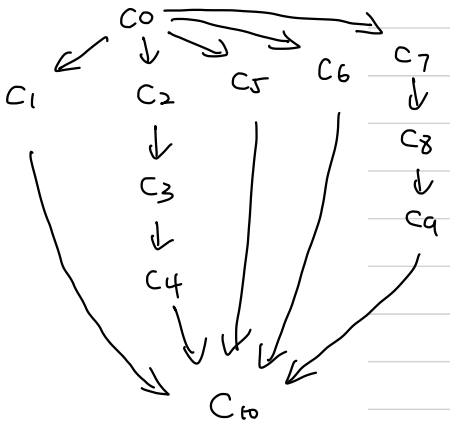



```
Begin
    S0 ;
    CoBegin
        Begin  S1 ; S4 End
        Begin
            S2
            CoBegin S6 ; S7 CoEnd
            S8
            CoBegin  S9 , S10 CoEnd
        End
        Begin  S3 ; S5  End
    Co End
    S11
End
```

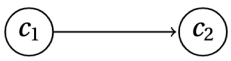
Assignment 2

2. Draw a precedence graph (a.k.a., dependency graph) that reflects the parallelism in the following code. Recall a precedence graph is a *directed graph* in which the nodes (vertices) represent processes and the edges represent the causal relation between two processes. If an edge exists from process node P_i to P_j then the process P_j must start executing its code c_j only when process P_i finishes executing its code c_i. Also we use the notation `ParBegin` and `ParEnd` to indicate a block in which program codes (or instructions) must run in parallel. Another alternative notation that can be found in the literature is `CoBegin` and `CoEnd` for concurrent begin/end.

```
Begin
  c0;
  ParBegin
    c1;
    Begin
      c2;
      c3;
      c4;
    End
    c5;
    c6;
    Begin
      c7;
      c8;
      c9;
    End
  ParEnd
  c10;
End
```



3. Complete the following program code to reflect the process precedence graph shown below (where c_i refers to the code-i). This code basically consists of creating two processes in parallel: the first process executes program P_1 and the second process executes program P_2. You should use one semaphore for that.




```
Begin
   ParBegin P1; P2; ParEnd
End
```

Program P_1:
```
Begin
  code1;
End
```

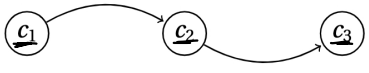
Program P_2:
```
Begin
  code2;
End
```

Semaphore S = 0;

P1 :
code 1;
release(S);

P2 :
acquire(S);
code 2;

4. Complete the following program code to reflect the process precedence graph shown below (where c_i refers to the code-i). This code basically consists of creating two processes in parallel: the first process executes program P_1, and the second process executes program P_2. You should use semaphores for that.




```
Begin
   ParBegin P1; P2; ParEnd
End
```

Program P_1:
```
Begin
  code1;
  code3;
End
```

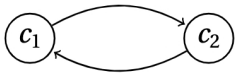
Program P_2:
```
Begin
  code2;
End
```

Semaphore $S_1 = 0$; $S_2 = 0$;

P1 :
code 1 ;
release (S_1)
acquire (S_2);
code 3;

P2 :
acquire (S_1);
code 2 ;
release (S_2);

7. Modify the following code (using two semaphores) so that the two programs P_1 and P_2 execute alternatively (i.e., $\ldots, P_1, P_2, P_1, P_2, P_1, \ldots$ with P_1 being the first to start execution.



Begin
  ParBegin $P_1; P_2;$ **ParEnd**
End

Program P_1:
Begin
  `while (true) {`$code_1$`;}`
End

Program P_2:
Begin
  `while (true) {`$code_2$`;}`
End

Semaphore x = 1; y = 0;

→ 最好是让 Semaphore 的值都初始成 0

P_1:
```
while (true) {
    acquire (x);
    code1;
    release (y);
}
```

P_2:
```
while (true) {
    acquire (y);
    code 2;
    release (x);
}
```

✗

Semaphore x = 0; y = 0;

P_1:
```
while (true) {
    code1;
    release (y);
    acquire (x);
}
```

P_2:
```
while (true) {
    acquire (y);
    code 2;
    release (x);
}
```

7. 和 7 一样的题目，但 modified 成 → $P_1, P_2, P_2, P_1, P_2, P_2$

Semaphore x = 0; y = 0;

P_1:
```
while (true) {
    code 1;
    release (y);
    acquire (x);
    release (y);
    acquire (x);
}
```

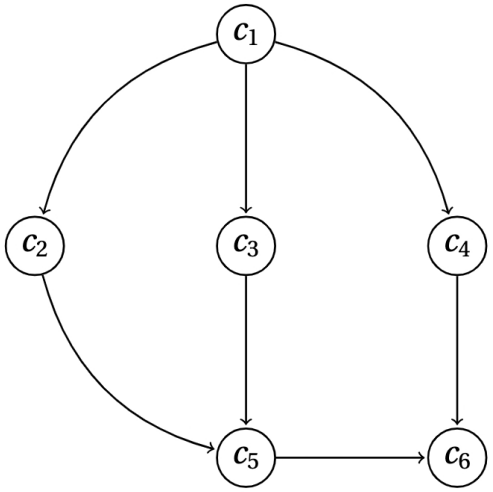
P_2:
```
while (true) {
    acquire (y);
    Code 2;
    release (x);
}
```

Semaphore x = 2; y = 0

P_1:
```
while (true) {
    acquire (y);
    acquire (y);
    code 1;
    release (x);
    release (x)
}
```

P_2
```
while (true) {
    acquire (x);
    code 2;
    release (y);
}
```

10. Write the program code that reflects the following precedence graph using six processes $P_1, \ldots, P_6$ each executing its dedicated code $c_1, \ldots, c_6$. You are required to use only three semaphores (A bonus is offered to students who solve the problem with 2 semaphores).



Semaphore X=0; Y=0; Z=0;

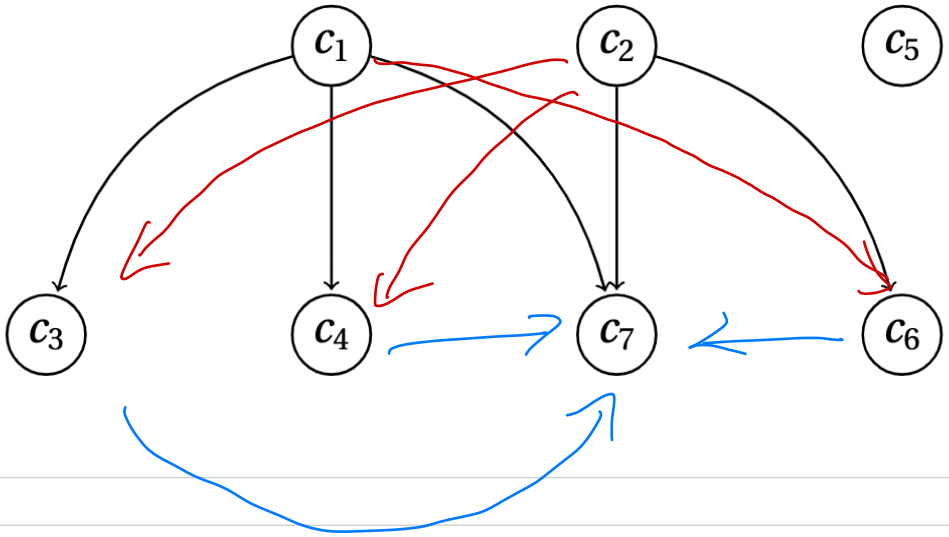
P_1:	P_2:	P_3	P_4	P_5
c_1	P(X)	P(X)	P(X)	P(Y)
V(X);	c_2;	c_3	c_4	P(Y)
V(X);	V(Y)	V(Y)	V(Z)	c_5
V(X);				V(Z)

P_6:
P(Z)
P(Z)
c_6

⇒ Solution 中有一个两个 semaphore 的 bonus 版本

Assignment 3 :

10. Write CoBegin/CoEnd code for the following precedence graph. Make your program express as much parallelism as possible within the limitations of CoBegin/CoEnd, while being sure to enforce all the constraints that are in the precedence graph. (The best solutions introduce two or three extra precedence edges. Try to find one of them.)




```
CoBegin
    C5
    Begin
        CoBegin  C1 , C2  CoEnd
        CoBegin  C3 , C4 , C7 , C6  CoEnd
    End
CoEnd
```

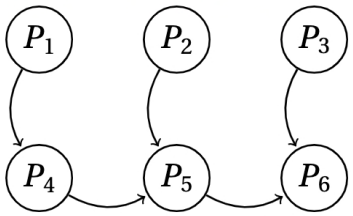
```
CoBegin
    C5
    Begin
      CoBegin
          Begin  C1 , C3 , C4  End
          Begin  C2 , C6  End
      CoEnd
      C7
    End
CoEnd
```

无形中把 $C_1 \to C_4$ 的线去了

```
CoBegin
    C5
    Begin
      CoBegin
          Begin
              C1
              CoBegin C3 ; C4 CoEnd
          End
          Begin  C2 , C6  End
      CoEnd
      C7
    End
CoEnd
```

Midterm 1

2. Use Begin/End and CoBegin/CoEnd programming constructs to write code matching this precedence graph [6 points: 1pt for each block].




```
Begin
  CoBegin
    P3
    Begin
      CoBegin
        Begin  P1 , P4  End
        P2
      CoEnd
      P5
    End
  CoEnd
  P6
End
```

=>

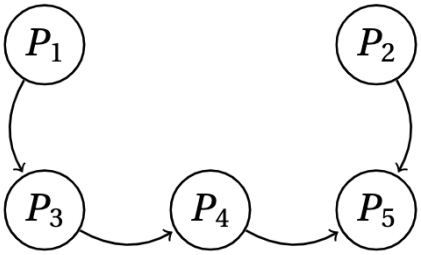
在这之间，只有P3和P5是"同时完成"，所以有P3→P6；P5→P6；

3. Using X, Y, Z (three semaphores) to

```
Semaphore X=0; Y=0; Z=0;

P1:        P2:        P3
  C1         C2         C3
  V(X);      V(Y);      V(Z)

P4:        P5:        P6:
  P(X);      P(Y);      P(Z)
  C4         P(Y);      P(Z)
  V(Y);      C5;        C6
             V(Z);
```

答案就是这么中的，告诉我们：可以 initialize 1个 二值信号量为-1

Midterm 2:



Use semaphore X=0; Y=-1; Z=0;

P( ), V( ) 在一个 process 中只能用一次

```
P1:          P2
  code 1;      code 2;
  V(X);        V(Y)

P3:          P4:
  P(X)         P(Z)
  code 3       code 4
  V(Z)         V(Y)

P5:
  P(Y)
  code 5;
```

Exercise 5：Deadlock Detection and Deadlock Avoidance（Exercise 5和Exercise 3的内容是衔接的）

Deadlock

Deadlock Avoidance （Banker's Algorithm）

Deadlock Detection （Deadlock detection algorithm）

Starvation（饥饿）：指的是当有进程一直拿不到执行 Critical section 的权利时，它处于一直等待的状态（饥饿），比如读者-写者问题中，如果读者优先的话，很有可能读者会源源不断地涌入，写者一直在等 → 写者饥饿。当一个进程一直饥饿直到它被见武予的任务已经失去了意义的时候，这个进程饿死

livelock（活锁）：指 进程各自都没有占用资源，但都在让另一个 进程先使用资源，也就造成了整个程序被卡住的情况，活锁

Deadlock：The implementation of Critical section protocol result in a situation where two or more processes are waiting indefinately for an event that can be caused only by one of the waiting process. 两个或两个以上的进程（或线程）在执行过程中，因争夺资源造成的一种互相等待的情况。它们会一直等待（若无外力干预）

↳ 死锁形成的必要条件：

1. 互斥（Mutual Exclusion）：线程对资源的访问是排他性的
2. 不剥夺条件（Non preemption）：线程已获得的资源，在未使用完之前，不能被其它线程剥夺，只能自己释放
3. 请求和保持（Hold and wait）：

   Karim 解释：Once a resource (critical section) is taken, a process doesn't release it till it's done with it（这样的话和条件二有什么区别？？？）

   更好的解释：一个进程因请求资源而阻塞时，对已获得的资源保持不放
4. 环路等待（circular wait）：在死锁发生时，必然存在一个"进程－资源环形链"，例如 $\{P_0, P_1, \cdots P_n\}$，P_0在等待P_1的资源，P_1在等待P_2的资源 …… P_n在等待P_0的资源

How to deal with deadlock?

1. Deadlock Ignorance （Used by most operating system）
   ↳ 直接不管了，因为死锁一般在很多系统中很难出现一次，与其另一堆代码去解决，还不如直接重启
2. Deadlock Avoidance：Use dynamic rules for requesting resources. OS determines whether a deadlock may arise or not before granting a given resource to a given process. → Banker's Algorithm
3. Deadlock Prevention
4. Deadlock detection and Recovery

Nolock（无锁）：没有对资源进行锁定，即所有的线程都能访问并修改同一个资源，但只有一个资源可以修改成功。如果有多个线程修改同一个值，必定有一个线程可以修改成功，其它没修改成功的线程会不停尝试直到其成功为止。

*和上面三个不一样的是：无锁是一种设计，而上面三个则全是状态

Banker's Algorithm && Deadlock detection (看清楚算法分别是用什么矩阵去判断的)

↘

Banker's Algorithm (Deadlock Avoidance) 和 Deadlock detection 的区别不大，仅有的区别可能在于：Banker Algo 会看失判断 Request 能不能予以批准，它们的区别具体如下：

* 在 Banker Algorithm 中，是压根不会给 Request 这个东西，而是只有 Allocation，Max，Available，Need，从 need 中推出存不存在 Safe sequence。Banker Algo 还可以检测独立的 request，可以根据现有的信息来判断要不要 grant 这个 request
* 在 Deadlock Detection 中，算法的设计都不包含检查 request 这上面，request 已经给你了，用 request 去推 Safe sequence。Deadlock Detection 只在乎当下的 "request" 能不能被满足，但它不在乎这整个进程的总需求 (Need) 能不能最终被满足。

∴ 可以说 Deadlock detection 所判断的是没有 Banker's Algorithm 全面的

1. 算法都用到以下的数据

1. Available : A vector of size m (resource type) to express the currently available number of instances for a given resourse R_j.

2. Max : 一个 n * m 的矩阵，其意义是：对于进程 P_i，它所能申请的资源 R_i 最多能有多少个

	R_0	R_1	R_2	
P_0	2	5	7	→ P_0 最多能申请 7 个 R_2
P_1	6	3	4	
P_2	3	3	6	

3. Allocation: 一个 n * m 的矩阵，其意义是：对于进程 P_i，它目前拥有的资源 R_i 有多少个

	R_0	R_1	R_2	
P_0	2	5	7	→ P_0 当下拥有 7 个 R_2
P_1	6	3	4	
P_2	3	3	6	

4. Need : 一个 n * m 的矩阵，其意义是：对于进程 P_i，它依然需要的资源 R_i 有多少个

	R_0	R_1	R_2	
P_0	2	5	7	→ P_0 依然需要 7 个 R_2
P_1	6	3	4	
P_2	3	3	6	

* Need = Max − Available

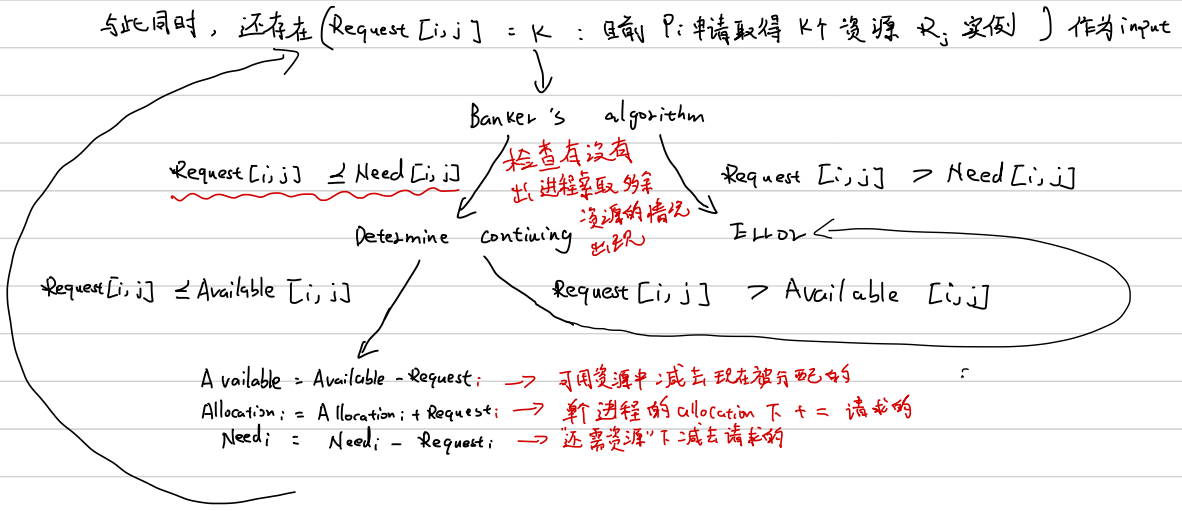
5. Request : 一个 n * m 的矩阵，其意义是：对于进程 P_i，它目前申请的资源 R_i 有多少个

	R_0	R_1	R_2	
P_0	2	5	7	→ P_0 当下申请 7 个 R_2
P_1	6	3	4	
P_2	3	3	6	

只在 deadlock detection 是直接给的，在 Banker 中是还要去判断 grant 不 grant 的

Banker's Algorithm:

1. 判断 Request:



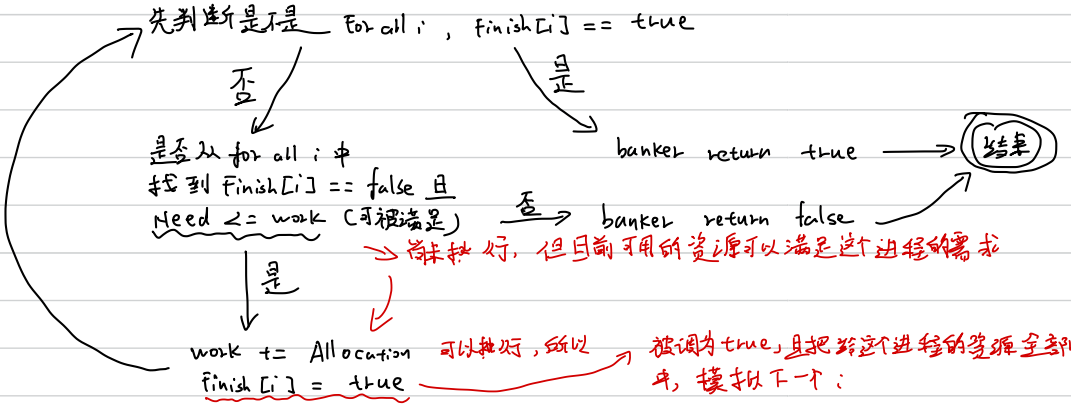
简言之，grant 不 grant 就看这两点... ① Request [i,j] ≤ Need [i,j] ② Request [i,j] ≤ Available [i,j]

↳ 在做题的时候，grant 完之后，还要再跑一遍下面的，找 Safe sequence

2. 根据 Need 判断是否存在 Safe sequence

新 Array: Work = Available

Finish such that Finish [i] = False



Example: Midterm 1

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Need

	A	B	C	D
P_0	0	0	0	0
P_1	0	7	5	0
P_2	1	0	0	2
P_3	0	0	2	0
P_4	0	6	4	2

1. Safe sequence

P_0: work = [1, 5, 3, 2]

↳ P_3: work = [1, 11, 6, 4]

↳ P_1: work = [2, 11, 6, 4]

↳ P_2: work = [3, 14, 11, 8]

↳ P_4: work = [3, 14, 12, 12]

2. P_1 request [0, 4, 2, 0] grant 吗?

grant!: [0,4,2,0] ≤ Need [1]

[0, 4, 2, 0] ≤ Available

Deadlock Detection

Algorithm:

1. Let work and Finish be vectors of length m and n respectively such that work = Available
   ↳ 如果 Allocation i ≠ 0, 那么 Finish[i] = false, else Finish[i] = true

2. Find a i in len(finish): # 遍历每个进程
   if Request i > work i: # 进程申请的超出系统能力
   break;
   Finish[i] = true; # 进程的需求可以被满足，那就满足它
   work = work + Allocation i # 进程被满足了，可以收回其资源
   if false in Finish: # 有进程的需求怎么样都无法被完成，那么就是说，它的 request
   # 会造成 deadlock
   return deadlock
   else:
   return deadlock free

Example, Midterm 2

3. Consider the following snapshot of a system:

	Allocation				Request				Available			
	W	X	Y	Z	W	X	Y	Z	W	X	Y	Z
P_0	0	1	0	0	0	0	0	0	0	0	0	0
P_1	2	0	0	2	2	0	2	2				
P_2	3	0	3	3	0	0	0	0				
P_3	2	1	1	2	1	0	0	1				
P_4	0	0	2	0	0	0	2	0				

Answer the following using deadlock the detection algorithm:

→找 Safe sequence

3.1. Is the system in a deadlocked state? Explain your answer by showing a possible order of execution for the processes [2.5 points].

The system is not in a deadlocked state. To prove that, we need to find a safe sequence: Considering the available resources (0 0 0 0) and the request of each process, we see that we can start executing process P_0 which will return (0 1 0 0), which increases the number of available resources to (0 1 0 0). Then we can execute P_2, so available becomes (3 1 3 3). Then P_3, so available becomes (5 2 4 5). Then $P4$, so available becomes (5 2 6 5). Finally, P_1 can be executed, and available becomes (7 2 6 7). Hence, a safe sequence exists $< P_0, P_2, P_3, P_4, P_1 >$.

3.2. Now, consider that process P_2 requests one additional instance of resource type W and one additional instance of resource type Z. Will the system be in a deadlocked state (Explain your answer)[1.5 points]?

By requesting more resources, the request array for process 2 becomes (1 0 0 1). By running the previous algorithm, we see that we can still run process P_0 and get back one instance of resource type X. So available becomes (0 1 0 0). With this amount of resources, we cannot execute the other remaining processes. Hence, the system is in a deadlocked state.

Exercise 4: Memory Management: Paging, multi-level paging, and segmentation with paging

1. Paging
2. Multi-level Paging
3. Segmentation with paging

* **Word Size (W)** vs **Address size (M)**

指的是一条地址可以存多少字节的东西 → W=8 → 一条地址可以存 1 byte

指的是一条地址由多少个bit来组成，用M可以找出Memory的大小（能存多少条地址）

→ M=10 → 2^{10} = 1024

↓

$2^{10} * 8$ = 1024 Byte = 1KB

基本的 Memory Management 概念：

* Cache
* Logical Address space and Physical address space
  - ↳ and why separate them?
* Address Binding (位址定位)
* Dynamic Loading (动态加载)
* Dynamic Linking and Shared Libraries
* Protection of memory space

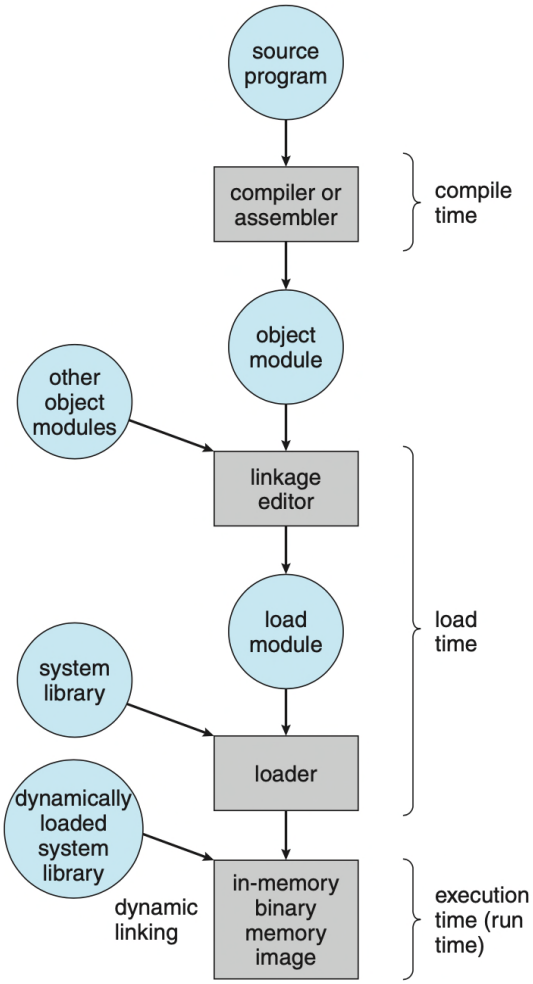
CACHE: CACHE 坐落在 Main memory 和 CPU之间，当CPU要access main memory中取东西时，CPU 先会从 CACHE 中取东西，这是因为 CPU 从 main memory 中取东西特别浪费时间，非常慢 → A memory buffer used to accomadate speed differential

Logical/Physical Address Space

- **Logical Address Space**: Logical Address 是 generated by CPU，而 logcical Address Space 是由一个进程生成的全部逻辑地址的集合（The set of all logical address generated by a program）
- **Physical Address Space**: An address seen by the memory unit, that is the one loaded into memory-address register of memory. 同理，physical address space is the set of all physical address corresponding to those logical address.

↳ **为什么要分成两种不一样的 address 呢?** → User program only deal with logical address, it never knows the "actual" physical address

Address Binding: is a mapping from one address space to another



一个user program在最最最初到被执行之间，经历了多种阶段，address会被不同的表示方法来表示，在source program中，address一般是比较symbolic的(such as count). 然后到了compiler的阶段，compiler会bind这些symbolic的地址to relocatable addresses (such as “14 bytes from the beginning of this module”). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014).

Classically, the binding of instructions and data to memory addresses can be done at any step along the way，位址定位可以在下面几个阶段中的任意一个完成:

Compile time(编译时期): If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

Load time(载入时期): If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

Execution time(执行时期): If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 7.1.3. Most general-purpose operating systems use this method.

↑

* If address binding finish at Compile time or load time, then logical address = physical address

* If address binding finish at execution time, then logical address != physical address

Dynamic Loading 动态加载：

程序会被保存在磁盘上，in a relocatable load format.主程序会首先被加载进内存，并且执行，如果此时需要执行另一个程序，the calling routine要首先检查这个被叫起来的程序有没有已经被加载了，如果没有被加载的话，relocatable linking loader会被执行并且会把那个程序给load进memory，然后更新program的address table。然后控制会给到新进来的程序

Is a software mechanism in which modules referred in a given program are loaded into the main memory only when being used.

Static linking 静态链接 Vs Dynamic Linking 动态链接

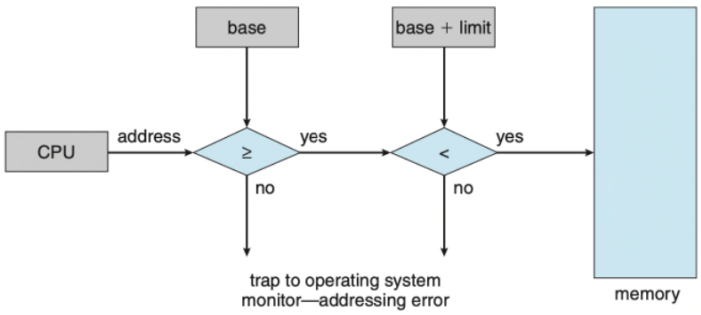
- 静态连接：
  - statically-linked library is a set of routines, external functions and variables which are resolved in a caller at compile-time and copied into a target application by a compiler, linker, or binder, producing an object file and a stand-alone executable
  - 静态连接就是，使用普通的函数库，在程序连接时将库中的代码拷贝到可执行文件中。假设有多个程序同时执行，并且同时调用了同一个库文件，这是内存中就会保留着许多重复的代码副本。造成内存浪费。
- 动态连接：
  - 动态链接就是，只有程序在执行时才将库中的代码装入内存，对于同一个动态链接库，无论有多少个程序在调用，内存中都只有一个动态库的副本。当动态库不再被任何程序使用，系统就会将它调出内存，这样就减少了应用程序对内存的要求。动态链接库是一种程序模块，不仅可以包含可执行的代码，通常还包含各种类型的预定义的数据和资源，扩大了库文件的使用范围。
  - **With dynamic linking, a *stub* is included in the image for each library- routine reference. The stub is a small piece of code（stub是存在哪里的呢？？stub是存在程序里面，当这个程序需要用到system library的时候，他对于system library的reference就会包含这个stub）**that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. **When the stub is executed, it checks to see whether the needed routine is already in memory.** If it is not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine.（所以，第一个需要这个system library的程序会把这个system library routine给加载进内存里面，这样下一个需要这个system library的程序就不需要重复加载了）Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.
  - Under this scheme, all processes that use a language library execute only one copy of the library code.
  - 动态连接同时还关系到了库的更新，如果一个库更新了，使用dynamic linking的系统不会受到太大的影响，因为每个程序是在run time的时候才连接的库，但是如果static linking，在编译的时候就连接的库的话，会造成一定的影响，因为**all such programs would need to be relinked to gain access to the new library**. Other programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries（动态连接库）**
- **链接有两种类型，一种是静态链接，直接将所用到的各个部分拼接起来作为所得到的最终文件。另一种是动态链接，不是直接的进行拼接，最终得到的文件仍然不是完整的程序，而是保留了接口。动态链接所得到的可执行文件在执行的时候按照需要动态的加载用到的部分，与此不同的静态链接没有这个动态加载的过程，因为所有需要的部分已经包含在可执行文件里面了。这里可以看到动态链接生成的文件往往会小一些，因为可重用的部分已经独立出来作为库文件了，这也是构建大型软件所必须的。**（精彩解释）

Protection of memory space: 为了防止操作系统的文件被 User 给 access 到，或者是一个 user 的 memory 被别的 user access 到，必须确保每个进程有自己独立的储存位置

- **Base register**: Holds the smallest legal physical memory address
- **Limit register**: Specifies the size of the range
  - Example: If the **base register** holds 300040 and the **limit register is 120900**, then the program can legally access all addresses from 300040 through 420939 (inclusive). Base和Limit的值对于每个进程都是一样的嘛?
  - 这两个register都只可以被操作系统加载，也就是说只有操作系统自身可以修改这两个值

用 Base register 和 Limit register 表明一个进程被分配到的内存

- 然后cpu会对user mode里面的register生成的每一条地址都进行检查，只要检查到一条是想要access os 或者是别的user的memory的都会令这个程序进入error
- Having the CPU hardware compare *every* address generated in user mode with the registers.



- 但是对于os来说，它拥有很高的权限，它能没有限制的访问os memory和user memory，访问user memory是为了把user program加载进user memory中

Why Paging? → Memory Allocation 2

↳ Contiguous Memory Allocation → 一个进程需要连续的物理地址储存

↓

Single Partition Everytime 只有一个程序被加载

Multiple Partitions

↳ 每次有多个进程被加载

↓

Fix Sized: 给每个进程分配一样大小的空间

Variable Size: 给每个进程分配不一样大小的空间 (按需)

会导致两个问题:

Internal Fragmentation: 如果是用 fix sized scheme 的话，当一个进程出现，当它向内存索要空间时，内存想都不想直接给它一个 fix sized block，但如果这个 block 的空间远大于进程实际所需的空间，那么就浪费掉了许多资源

External Fragmentation: 指的是用 contiguous memory allocation 的话，进程需要连续的物理地址，但此时内存中有足够的不连续的空间，但因为是 Contiguous memory allocation，这些都被浪费了

Non-Contiguous Memory Allocation

- Paging
- Multi-level Paging
- Segmentation with Paging

Paging: A memory management scheme that allows programs to be stored in the main memory in a non-contiguous way by dividing their logical address spaces into fixed and equal sized blocks known as pages.

↳ 就是系统把逻辑地址分成一个个 page，page 对应了物理地址中的不连续的 frame，这样，User program 对于 User 来说是用连续地址存放的，但实际存放在不连续的物理地址空间中.

Total number of pages = logical address space size / Page size

MMU (Memory Management Unit) 怎样找到某个逻辑地址对应的物理地址呢?

↳ 用 page number 从 Page table 中找到对应的 frame 的 base address，再用这个 base address 加上 page offset 即可得到对应的 physical address

	Page number	Frame's base address
Page table: →	~	~

Page table sizes = Total number of page * PTE (Page table entry)

↳* If the size of the logical address space is 2^m and a page size is 2^n, then the high order $m-n$ bits of logical address designate the page number and the n lower-order bits designate the page offset

Example: Address 16 bits, size of page is 8KB, 一个 address 是 0x2213

↳

Logical address space = 2^{16} byte
Page size = 2^{13} byte

Page number = 16 - 13 = 3 bits
Page offset = 13 bits

∴ 0x2213 = 0010 0010 0001 0011
(0010: page number; 0010 0001 0011: page offset)

顺便...补充一下 memory 对照表

Name	bytes)	bits)
byte	1	8
KB	2^{10}	
MB	2^{20}	
GB	2^{30}	
TB	2^{40}	
PB	2^{50}	
EB	2^{60}	
ZB	2^{70}	
YB	2^{80}	

Multi-level Paging

↳ 多级分页，实际上是解决了页表太大的问题，现在的应用程序，要求的逻辑空间非常大，那么可能为存储页表就消耗了很多的内存了

* Side Note：页表存在内存中，一般用 PTBR (page table base register) 来记录页表的 base address，但这样比较慢，所以一般会多搞一个 TLB (Transfer Look-aside buffer) 来储存部分的 page table entry

↳ | key | value |

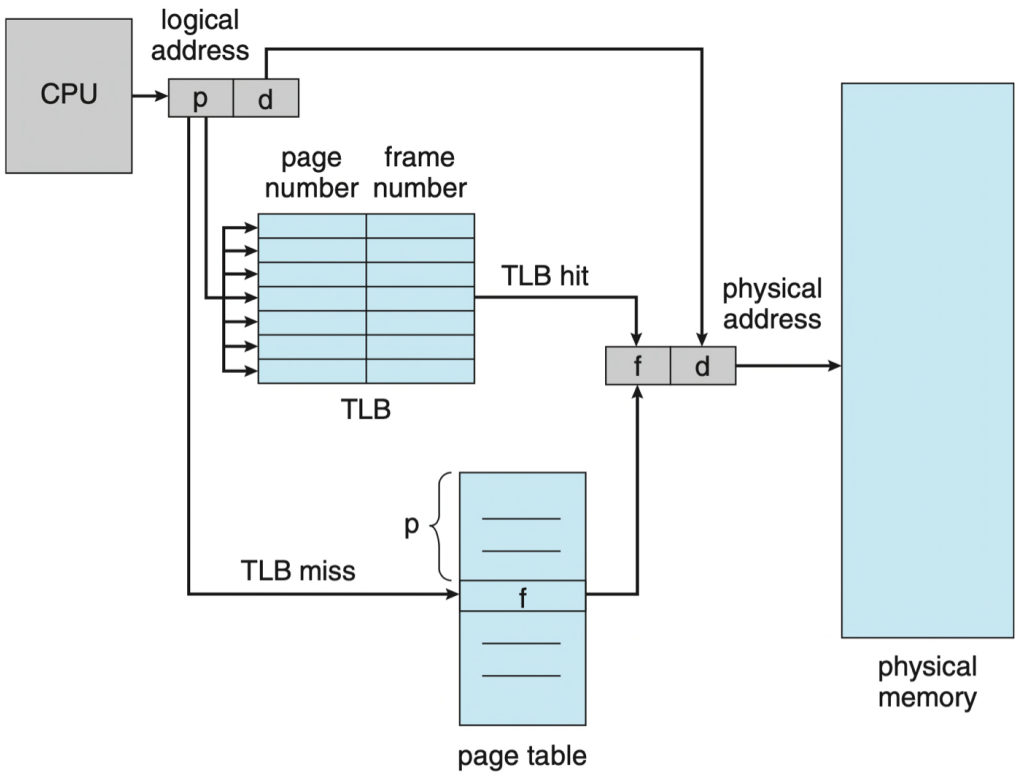
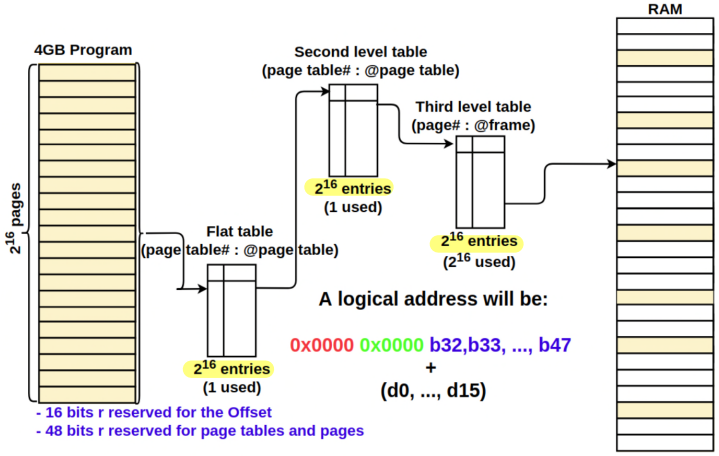


Figure 7.11 Paging hardware with TLB.

Multi-level Paging

↳ 为了节省页表制造的内存浪费，使用多级页表，当我们只需要一点 of the process memory 时，我们不需要一下子把全部页表都 load 进来 (如果没有多级分页机制的话，我们每次都需创建一个 as large as possible 的页表)。The page table is also paged for a certain number of levels.

Example: 64 bits address, page size is 64 KB = 2^{16} byte, PTE = 2^2 bytes



No 多级页：

Page table size for 4GB (as large as possible)

= $2^{64} / 2^{16} * 2^2 = 2^{50}$ = 1PB

三级分页：

∵ 4GB = 2^{32} byte

* 需要 $2^{32}/2^{16} = 2^{16}$ 个 3 level page table
* 需要 $2^{16}/2^{16} = 1$ 个 2 level page table
* 需要 1 个 1 level page table

$2^{16} * 2^{16} * 2^2 + 1 * 2^{16} * 2^2 + 1 * 2^{16} * 2^2$

[3] [2] [1]

= $2^{34} + 2^{18} + 2^{18}$

= 8 GB + 256 KB + 256 KB

设计 Multi-level Paging scheme

↳ 即算level有多少个 entries

↳ 首先，刚刚那个规律依然适用，m-n 那个

∴ Let's say we have 32 bit address, 2^{12} page size, 那么它的地址是怎么分配已

20 | 12

Page number | page offset

在设计的时候 不动 Page offset，动 Page number

Example：2 level paging：

10 | 10 | 12

level 1 | level 2 | page offset

∴ level 1中一个 Page table 含 2^{10} 个 entries，level 2 同理

3 level paging：

7 | 7 | 6 | 12

level 1 | level 2 | level 3 | Page offset

如果是除不尽的话，从头 Clevel 1开始，向里面填尽可能多且维持平均的 bits.

Segmentation with Paging

Segmentation：a memory management scheme in which the user program's logical space is divided into variable sized partitions (按需分配)，called segments and stored into the memory with no pre-defined order. 同样的，存在 Segment table 及 Segment table base register (STBR)

Segmentation with paging：A memory management scheme in which the user program's logical space is divided into variable sized partitions (Segments)，and each segment is again divided into pages.

- 先把逻辑地址的内容分段，然后再把段给分成页
- 所以assignment 4的最后一题显得非常具有代表性
- 存在一个STBR来指向每个程序对应的segment table

segment table 存的是 Segment 对应的 page table 的 base address

- 这样子分段之后，segment table里面本来存着的base物理地址位置，现在变成了对应的page table，然后还有segment limit
- 所以在使用这样子的scheme的时候，逻辑地址对于用户来说是这样子的：
  - Segment#: Segment_offset
  - 然后这个东西被操作系统解释成: Segment#:Page#:Page_Offset
- 也就是说，在追寻一个逻辑地址的时候，先根据segment table base register来追寻到segment table的位置，然后根据这个逻辑地址追寻到对应的segment entry, it would be something like this:

Segment#	Segment size	Page table base address
0	512kb	011
1	512kb	045

- 根据segment table来找到对应的page table base address，然后找到page table，it would be something like this:

Page#	Page base address
0	0742
1	0693

- 然后根据page#找到对应的entry，找到page base address，然后根据page offset锁定最终位置

-Segmentation with paging and TLB:

  - 一个tlb里面存着的是：
    - Segment#:Page#:limit: Frame#

Exercise 7: Page Replacement Algorithm: show understanding of page faults and replacement

Page faults

Segment faults

Reference string

Replacement Algorithm

Page faults: Is an exception that occurs when CPU tries to access a page that is not in the main memory. CPU 请求的内存数据不在物理内存中 → Page fault (缺页)

↳ 处理办法: 使用 handler: (!!! 如果一个page是illegal access的话, 也会是 Page fault → 直接terminate)

1. The current program is interrupted and page fault handler is invoked
2. The handler initiate a disk transfer to load the page into memory
   - 2.1 The memory management module allocates a available frame to host the new page
   - 2.2 If no frame is available, the memory management module swaps out a "victim" page from the memory following a page replacement algorithm

为什么需要 Page replacement

3. Once done, the CPU is informed (e.g. DMA completion interrupt)
4. Interrupt handler updates the page table and TLB
5. The CPU restarts the instruction of the interrupted program

Segment fault: 和 Page fault 很像, 不过这个是 an expection that occurs when the CPU tries to access a segment that is not present in the main memory

↳处理办法一样也是 handler:

1. The current program is interrupted and segment fault handler is invoked
2. The handler initiate a disk transfer to load the page table of that segment (因为 segment 中存的是 page table —— 如果是 Segmentation with Paging 的话)
   - 2.1 if there is available frame, a frame is allocated to host the new page table
3. Once done, the CPU is informed and the segment table is updated
4. Try access the segment. Then if the requested page is not in memory → page fault
5. TLB is updated

Page fault, Segment fault 能帮助我们做什么? → Virtual memory

↳

Virtual memory: Allows programs to be partially loaded into memory and hence

advantage →

1. Program can have larger sizes and still can be executed
   ↳ 那么如果有需要的 page 不在 main memory 怎么把? → Page fault handler 把它给换出来 → Only load pages only as they are needed → demand paging
2. Increase the degree of multi programming
3. Increase the throughput

Pure demand paging: never bring a page into memory until it's required

↳ 它执行一次的话, 需要的每个 page 都会经历 page fault

Valid / Invalid bit: 如果一个 page 对应的 validity bit 是 valid, 说明这个 page 是在内存中的并且是一个 legal access; 如果一个 page 对应的 validity bit 是 invalid 的, 那么就会有 page fault

Effective Access time (用来衡量 performance of demand paging scheme)

↳ p 代表发生 page fault 的概率

Effective Access time = $(1-p) * t_{memory\text{-}access} + p * t_{page\text{-}fault}$

what is $t_{page\text{-}fault}$, page fault takes how long?

↳ Slide 上对 page fault handler 的解释这 3 步，但其实比较关键的是：

1. Service the page fault interrupt
2. Read in the page
3. Restart the process

Page Replacement Algorithm:

↳ 刚刚已经说了，page replacement 实际上就是当 CPU 必需要把内存中的 page 拿出来并且没有 available frame，意即必须要从现在的 frame 中取一个 page 出来，然后把这个 page 换进去 (replacement)

Page Replacement Algorithm: 就是一个决定谁换谁的算法

Reference string: 就是用来标记 page 的办法，它可以是
- A random number generator
- Trace the accessed memory address (only the page number)

同时，我们根据 reference string 会 generate 一个 reference sequence:

1, 4, 1, 1, 6, 1, 6, 1, 6, 6, 1

它代表了 page 会以一种什么样的形式来进入，但值得注意的是忽略掉所有 immediate access to the same page 即 1,1 -> 1 因为 1,1 最终也只会造成一个 page fault

1, 4, 1, 6, 1, 6, 1, 6, 1

当然，要计算 page fault rate 时，分母需要的是在"化简"之前的

FIFO (First in first out) Page replacement: Oldest page is chosen to be replaced

OPT (Optimal / clairvoyant) Page replacement: Replaces the pages that will not be used for the longest period of time (很难实现 因为需要 additional information)

LRU (Least recently used) page replacement: Replaces the pages that had not been referenced for the longest period of time.

MFU (Most frequently used) page replacement: Replaces the pages that had been referenced for the most frequent.

Example:

Assignment 5:

6. The following sequence of virtual memory references is generated when a program is executed. Each memory reference is a 12 bit number written as three hexadecimal digits.

这些代表了每个 Page 的 memory reference

0x019, 0x01A, 0x1E4, 0x170, 0x073, 0x30E, 0x185, 0x24B, 0x24C, 0x430, 0x458, 0x364

(a) What is the reference string, assuming a page size of 256 Bytes? The definition of reference strings is given at the end of textbook section 9.4.1.

2^8 byte → 根据前面提到的，8 bit 在 address 中代表了 Page offset，总共 12 bit 的地址，所以前 4 bit 就是 Page number

If a page has a size of 256 byte, this implies an offset of 8 bits. Considering our logical addresses that are on 12 bits, 4 bits remain from the MSB (Most significant bits) to encode the page number **[1pt]**:

0x019, 0x01A, 0x1E4, 0x170, 0x073, 0x30E, 0x185, 0x24B, 0x24C, 0x430, 0x458, 0x364

which becomes:

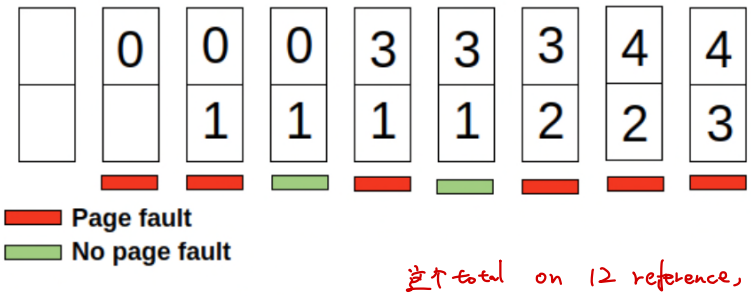
0, 0, 1, 1, 0, 3, 1, 2, 2, 4, 4, 3

把 immediate access 给化简

which becomes:

0, 1, 0, 3, 1, 2, 4, 3 **[1pt]**

(b) Find the page fault rate for the reference string in part (a): assume that 2 frames of main memory are available to the program and the FIFO page replacement algorithm is used. Note that the page fault rate is calculated as "number of page faults" divided by "number of virtual memory references used to form the reference string".



这个 total on 12 reference，指的是没有化简之前的 reference sequence

[2pts] for explanation (e.g., drawing)

We have a total of 6 page faults on 12 references. This give a rate of 6/12=50% **[1pt]**.

A little bit more about Paging and Page fault

↳ Thrashing

Working Set

Exercise 6 : Hard Disk Drive: Unstand the concepts that presented in lecture 32

Hard Disk Drive terminology

Hard Disk Drive Scheduling

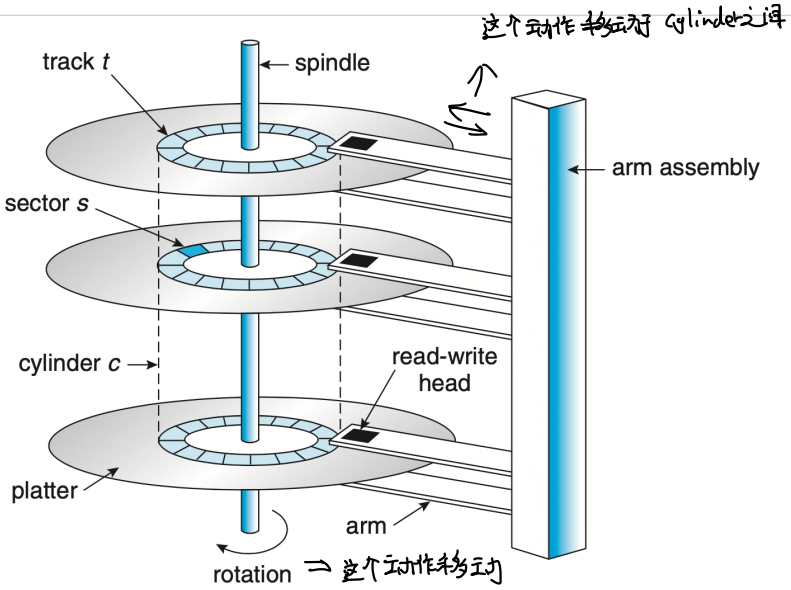
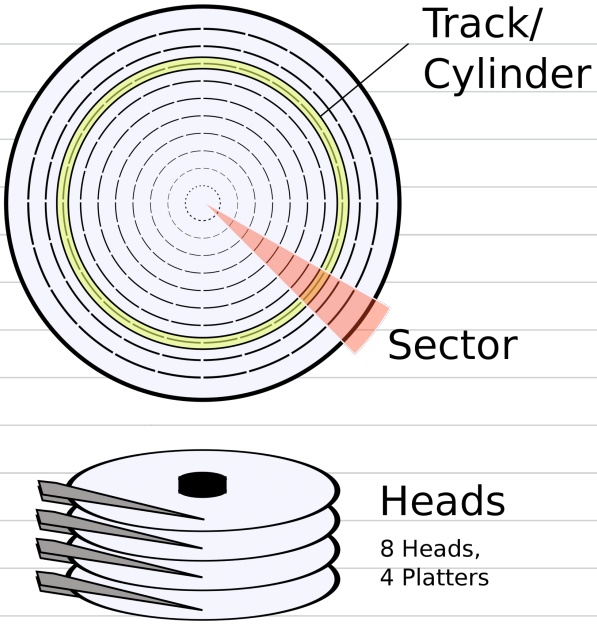


Figure 11.1 Moving-head disk mechanism.



* Platter: 代表的就是一个简单碟片. Platter 的两面都覆盖着 magenetic material

* Read-write head: A read-write head flies above each surface of every sides of every platters. So two heads per one platter.

* Disk Arm: The heads are attached to a disk arm that moves all the heads as a unit. (前后前后)

* Track, sector, cylinder: The surface of a platter is logically divided into circular tracks (所以一圈就是一个 track), which are subdivided into sectors (对于 disk 来说最小的储存单位, 最大可以存 512 byte). The set of tracks that are at one arm position makes up a cylinder.

Rates:

Transfer rate: The rate at which data flow between the drive and the computer.

Positioning time (random-access time): Seek time + rotational latency

↘ Seek time: Time necessary to move the disk arm to the desired cylinder

rotational latency: Time necessary to rotate the desired sector to the disk head

Head Crash: 当 Read-write head 真实地触碰到 platter 上的时候 (make contact) → damaged

Block (Bloc): { Smallest unit of transfer. / number of blocs decided by LBA (logical Block Address) size of the disk controller / one bloc can be one sector or multiple sectors → 一个 sector 最多可存 512 byte

Example: HDD interface generates LBA on 22 bits. One bloc contains one sector (512 byte)
What is the max size of the HDD supported?
↳ 22 bits → 2^{22} 个 blocks * 512 byte (per sector per block)
$= 2^{22} * 2^9 = 2^{31}$ byte = 2GB

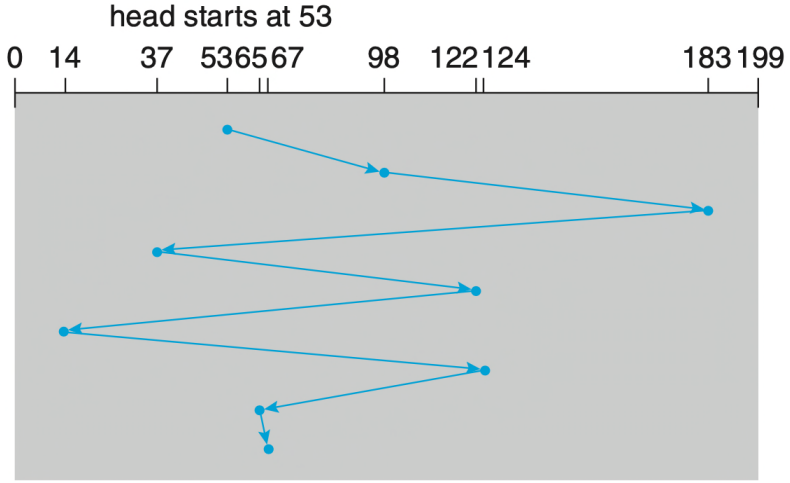
因为 LBA 位数是定值，所以一个 block 储存的 sector 的数量和 max size of HDD 是与相成正比的关系

Internal fragmentation: 因为最小的传输单位是一个 block，所以当一个 block 没有被占满时会发生 internal fragmentation (定义在前面 Paging 那儿)，且对所有比一个 block 大小还小的文件而言，压缩只会带来更大的浪费 (不管再怎么压缩，一个 block 是必然被 allocated)

Hard Disk Scheduling Algorithm
↳ 在 multiprogramming environment，多个对于 HDD 的访问会被放在 queue 中，每个访问被其要访问的 block number 所标识存在一个 queue 中：例：{98, 183, 37, 122, 14, 124, 65, 67}，HDD scheduling Algo 的目地就是以一种最优 (Seek time, rotational latency, number of head movements 都尽可能的小) 的排序来完成 queue

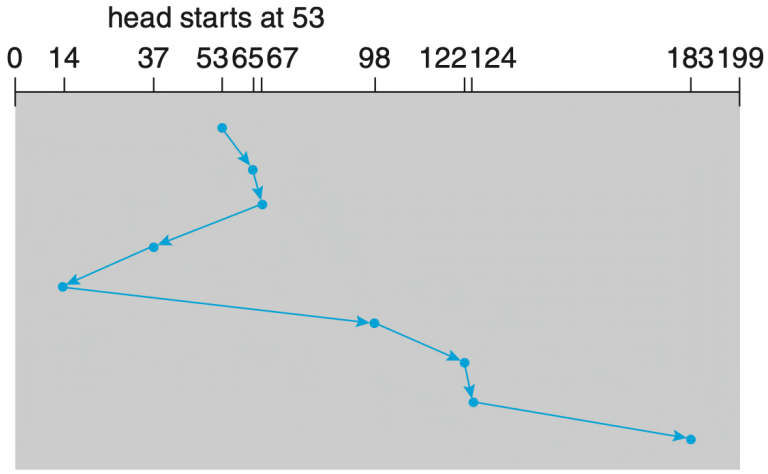
1. FCFS (First come first Serve using FIFO)

{98, 183, 37, 122, 14, 124, 65, 67}: (98-53) + (183-98) + (183-37) + (122-37) + (122-14) + (124-14) + (124-65) + (67-65) = 640 head movements (cylinders)



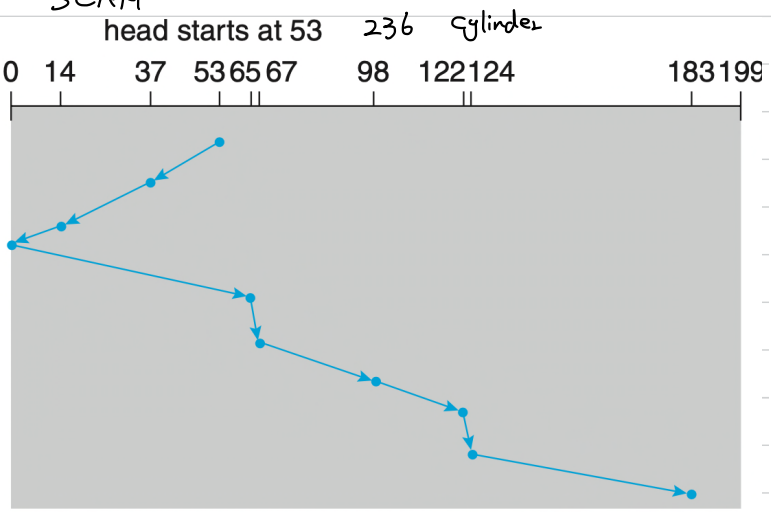
2. SSTF: Shortest-Seek-time-first Algo：从某处开始，下一个要访问的是离它最近的

{98, 183, 37, 122, 14, 124, 65, 67}: Total head movement: 236 (cylinders)

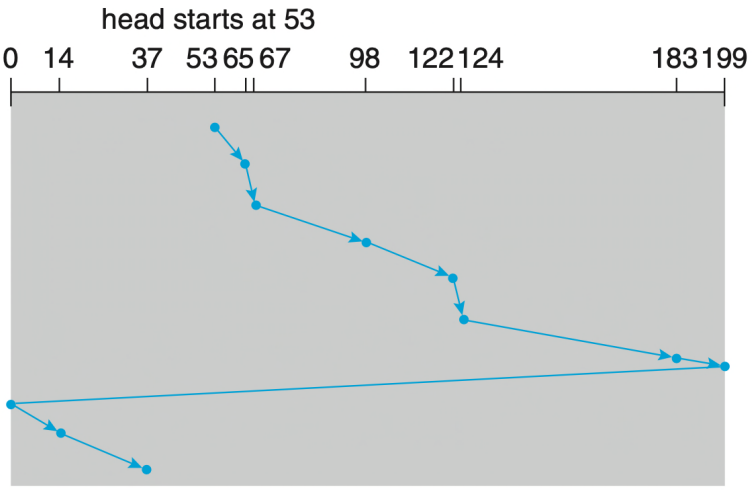


SCAN & C-SCAN : {98, 183, 37, 122, 14, 124, 65, 67}

SCAN



C-SCAN

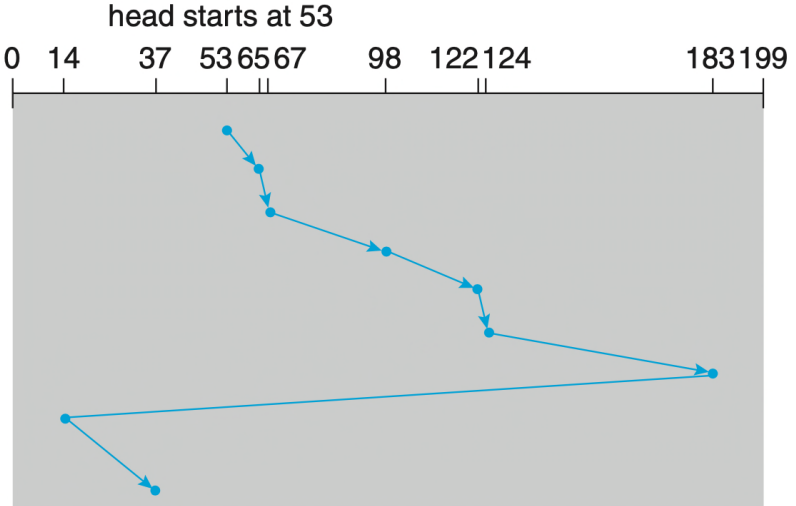


Scan Algorithm (Elevator Algorithm) 和 C-SCAN Algorithm 都像是电梯一样，它们的头都会在磁盘中有序移动，不同的是：

SCAN 从起点开始像负方向（左）移动，途经访问所有在这个方向的 request，到 0 之后，向正方向（右）移动，到 max 之后，又向负方向移动（←，→，←，→ ···）

C-SCAN 从起点开始像正方向（右）移动，途经访问所有在这个方向的 request，到 max 之后，头臂越过没有回到 0，并重新向正方向移动（→，→，→，→ ···）

Look (C-Look Algorithm) : {98, 183, 37, 122, 14, 124, 65, 67}



Arms goes only as far as the final request in each direction, then it reverse direction immediately, 因为是 C-Look 这个 Arm 去到 183 之后，发现正方向已经没有更多的 request 了，那它是从最左边的 request 开始 (14)，而不是从 183 扫过去 14

Exercise 8: Synchronization with Monitors: Show your understanding of using monitors for Process Synchronization

Monitors

Monitor 中的东西 有什么

Monitor 中的 method 和 Semaphore 的 method有什么区别

Monitor (管程): High-level programming language constructs that provides mutual exclusion [and synchronization]. 管程是一种抽象数据类型 (Abstract Data Type), 它为程序提供了一种更便捷、更方便的 Synchronization 的方法. "一个管程定义了一个数据结构和能为并发进程所执行(在该数据结构上)的一组操作, 这组操作能同步进程和改变管程中的数据"

Monitor 中有什么东西:

* A set of local variables (管程中的共享变量)
* A set of condition variables (管程中的条件变量)
* Initialization code (对局部于管程内部的共享数据设置初始值的语句)
* A set of programmer-defined operation

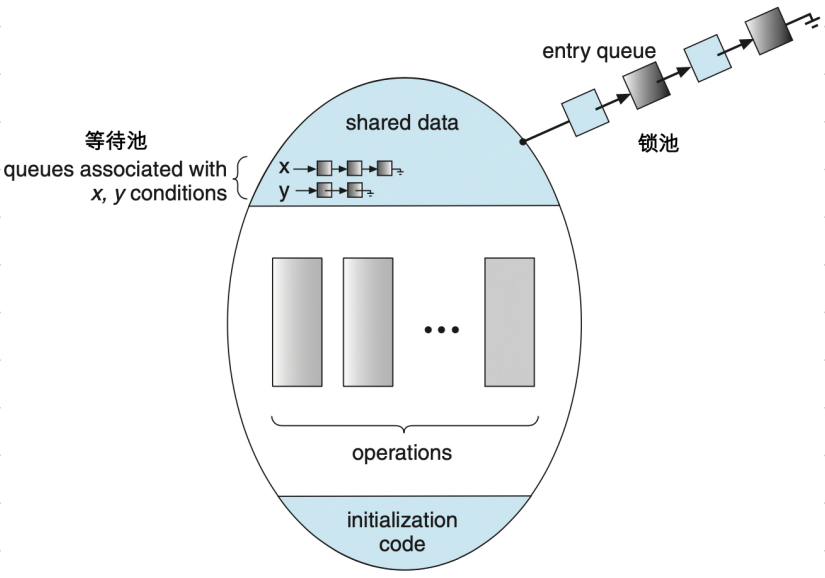
局限在管程中的数据结构, 只能被局限在管程的操作过程访问, 任何在管程之外的操作都不可以访问它

为什么需要 condition variable 条件变量? → 管程保证了同一时刻只有一个进程在管程内活动, 也就说, 管程内定义的 operation 在同一时刻只能被一个进程使用, 但这样不够多

→ Monitor without condition variables → 无法保证程序以设计的顺序执行, 因此需要设置 condition variable, 让已经进入管程但无法执行 (due to some designed condition) 的进程阻塞自己.

"Pure monitor"

一个管程的基本结构:



Entry queue: 队列, 进程在里排队对进入管程. 为了方便理解, 它又被称为锁池, 只有在锁池的进程才可以去竞争条件变量的锁

Condition variable:

* Each condition variable has a queue associated with it (等待池)

* There are only two operation that can be used on condition variable (Scheme 1)

1. wait(): wait() suspend the thread which called and gives back the lock, wait() 把当前执行 wait() 的进程放回等待池, 并把锁还回去.

2. notify() && notifyAll(): 唤醒量一个/全部位于等待池中的进程, 并把一个/全部被唤醒的本来在等待池的进程放回锁池

两种 implementation:

1. Notify and continue: Notifying thread keeps the monitor lock and continue its execution.
2. Notify and wait: Notifying thread is blocked and the notified one gets the lock

Notify() && NotifyAll():

↓

为什么一般推荐使用 NotifyAll() 而非 Notify()？

想象一下，当一个拿着锁的线程执行x.notify, 这时，有一个处于等待池中的线程(A)被移到了锁池中，那么当目前的线程执行了wait之后，他把锁释放出来，并且把自己放入等待池中，这个时候，位于锁池的线程A就抢到了这个锁，并且开始运行自己的东西，但是如果他在执行wait的时候，并没有执行notify/notifyall，此时，虽然锁被释放出来了，但全部的线程都处于等待池中，他们想要锁，但是他们不处于锁池中，所以他们没有资格抢锁，于是就死锁了。但是只要执行一次notifyall，全部的线程都会进入锁池中抢这个锁。再加上在使用notify的时候，如果有多个线程在等，把哪个线程移入锁池也是完全不由程序员控制的。

*锁池/等待池等是关于Java的管程的，→ 当执行了 wait() 之后，进程是等在管程里面的！

Example: Assignment 6

6. Consider the following producer-consumer problem: A customer enters a fast food to grab $K \geq 1$ hamburgers and leave. If no hamburger is available (N==0), the customer waits in a queue. Customers can enter the shop at anytime and stand in the queue if no hamburger is available. A cooker, prepares $S \geq 1$ hamburgers at a time then randomly chose a customer to grab some hamburgers. The selected customer may not be the right customer I.e., if the number of available hamburgers is less than what the customer needs, the customer needs to go back into the queue. Complete the following code so that customers can get their food and leave the store.

```
Monitor M
{
  Condition Variable:  Q;
  Integer:  N=0;
  Cook_hamburgers(Integer S)
  {
      N = N+S;
      Q.NotifyAll();

  }
}
Customer()
{


  M.Grab_humburgers(Integer K);
  Leave();

}
```

```
Grab_hamburgers(Integer K)
{
   while(K>N)  Q.wait();
    N = N-K;

}

  Cooker()
  {
      while(1)
      {
        M.Cook_hamburgers(Integer S);
        Look_at_customers();
      }
  }
```

当现有的汉堡包数量无法满足顾客需求时

You don't have to worry about the two functions: Leave() and Look_at_customers().

Exercise 9: Operating System Security: Understand the presented attacks during the Lectures

* General terminology regarding Operating System security
* Buffer overflow
  ↳ and Function call
* Integer overflow

General terminology regarding Operating System security (Mostly from lecture slide)

* Basic idea of OS security: Computer resources must be guarded against Unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency

Safe vs Security:

- Safe: A system is said to be safe if the system operates as it was intended to operate under the circumstances for which it was designed. 在OS设计的场景下，OS能按照设计者的意图去执行，那么这个系统是 Safe 的
- Security: A system is said to be secure if the system operates as it was intended to operate under all circumstances. (包括被攻击时的情况，OS都是正常表现 ——> Secure)

⇒ 对更多的 OS Security 的 terminology，都直接在 slide 中查找就好

Buffer Overflow 缓冲区溢出

- The most common way for an attacker to gain unauthorized access to the target system
- 其中，buffer overflow 其中一种形式便是：Programer neglected to code bounds checking on input field. 这种情况下，attacker sends more data than the program was expecting. Then:

1. These "more data" overflow an input field, command-line argument, or input buffer until it writes into the stack. 让输入超过系统所期待着的，这样造成"溢出"
2. Overwrite the return address on the stack with the address of the exploit code loaded in step 3 溢出的数据会一直溢出到 Stack 中，覆盖 Stack 中的 return address，stack 中的 return address 会被 IP 读取 (IP 中存的是程序下一步要执行的地址)
3. Write a simple set of code for the next space in the stack that includes the commands that the attacker wishes to execute 然后系统就会执行那段本不应该执行的代码

以上的形式，是 buffer overflow 中的一种形式，也被称为 Shellcode

Shellcode: is a string that expresses a binary program that starts a shell interpreter

Example:

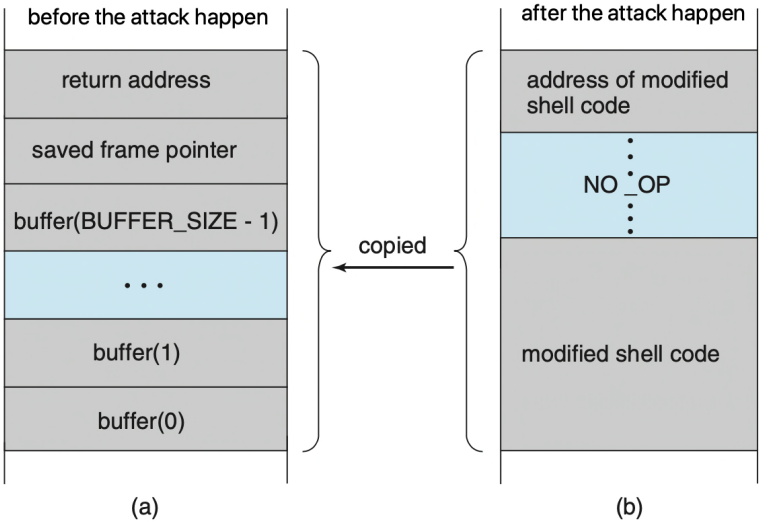
```
#include <stdio.h>
#define BUFFER_SIZE 256

int main(int argc, char *argv[])
{
   char buffer[BUFFER_SIZE];

   if (argc < 2)
      return -1;
   else {
      strcpy(buffer,argv[1]);
      return 0;
   }
}
```

看随的 frame 里，只要程序此时一运行，we return to the modified shell code, which runs with the access rights of the attacked process!

→ 定义了一个 array

→ 把 argv[1] 复制进入 buffer 这个 array 中

→ 如果程序员在写程序的时候没有规定能放入 argv[1] 的上限的话，就有可能会造成 buffer overflow



*这张图展现了 attacker program 是如何占领 stack frame 的 ↓

1. 因为它把 shell code 定义在了图 B 的位置，所以这个程序首先找到 buffer[0] 的位置。buffer[0] 的位置是 attacker 想要运行的

2. 找到这个位置之后，program 会把这个位置放在之后溢出的占领 return address 的位置上.

3. Program 还要加上一堆 NOP (No operation) 来填满这个剩余的 stack frame，从而让 buffer 溢出

至此，整个 shell code program 就设计好了

当然，在 lecture slide 上画的 shell code 是这样的：

Payload

NOP : Shellcode : RET@ : \0

NOP 在 shellcode 后而不是像书里画讲的

Mechanism to detect buffer overflow:

* Stack smashing prevention: Use canaries (金丝雀) to detect PC overwriting attempts
  → 在 stack 中放入一个别的东西，随时检查它有没有被覆盖，有的话，直接 about
* Disallow execution of code in a stack section of memory (shellcode)
* Detect a sequence of NOP
* Detect a sequence such as /bin/bash or /bin/sh
  用来获取系统权限

Heap Overflow: Overwriting a command on the heap to execute an arbitary malicious command.

↓

Heap: is a memory segment allocated by the OS to each program in order to perform dynamic allocation during runtime.

» 指的就是程序员可以自己 动态地在代码中通过 System Call ( ) (malloc ( )) 索要空间，而这些 自己索要的空间不存在 Stack 中，在 heap 中，像之前的 buffer overflow，程序通过 local variable 的形式新建 array，所以"buffer"是存在 stack 中的 (local variable)

The Heap

The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory (and is larger). To allocate memory on the heap, you must use `malloc()` or `calloc()`, which are built-in C functions. Once you have allocated memory on the heap, you are responsible for using `free()` to deallocate that memory once you don't need it any more. If you fail to do this, your program will have what is known as a **memory leak**. That is, memory on the heap will still be set aside (and won't be available to other processes). As we will see in the debugging section, there is a tool called `valgrind` that can help you detect memory leaks.

Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer). Heap memory is slightly slower to be read from and written to, because one has to use **pointers** to access memory on the heap. We will talk about pointers shortly.

Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

Stack vs Heap Pros and Cons

Stack

- very fast access
- don't have to explicitly de-allocate variables
- space is managed efficiently by CPU, memory will not become fragmented
- local variables only
- limit on stack size (OS-dependent)
- variables cannot be resized

Heap

- variables can be accessed globally
- no limit on memory size
- (relatively) slower access
- no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
- you must manage memory (you're in charge of allocating and freeing variables)
- variables can be resized using `realloc()`

```
Void Main ( )
{
    int c = 0;
    int *p;
    p = (int*) malloc(sizeof(int));
    free(p);
}
```

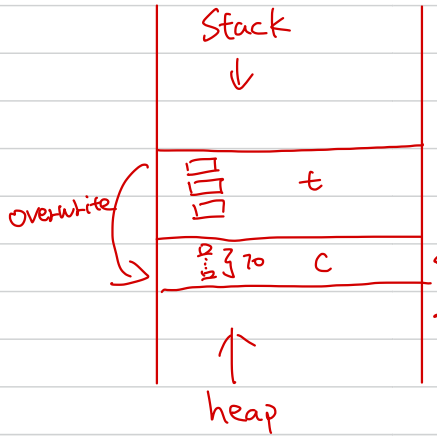
这是一个使用 heap 的例子，用 malloc 来动态地增加 heap 的大小

Example: Heap Overflow

```
Void Main (int argc, char *argv[])
{
    char *t, *c;
    t = (char *) malloc(3);
    c = (char *) malloc(70);
    strcpy(c,"legitimate command")
    strcpy(t, argv[1]);
    system(c);
    return;
}
```

都在 heap 中

=>



这是整个 heap 的架构，目前程序运行的 legal command 在 c 中

但如果，本身只能容纳 3 个单位的 t 被放入了 73 个单位，那么多出的 70 个内容向下伸展，把 未来 legal 的 c 给覆盖了 → heap overflow

Integer Overflow: Occurs when the variable type int is used insted of unsigned_int. It allows an attacker to use values that are not allowed but still getting access to some resources

→ 为什么？因为 Storage of type, int会 wrap around → 124 (分重)

```
1 #include <stdio.h>
2
3 void main ()
4 {
5   int a;
6   int b = 100;
7   printf("Input value that is less than 10 if you want to win:\n");
8   scanf("%d", &a);
9   if(a*b<=1000)
10  {
11        printf("You won: %d dollars.\n",a*10);
12  }
13  else
14  {
15        printf("You lost %d dollars.\n",a);
16  }
17 }
```

→ 如果 a 是一个特别大的数 : wrap around 且在这里依然进入了 if 条件

CISC 124 : Storage of Integers

Storage of Integers

- An "un-signed" 8 digit (one byte) **binary number** can range from 00000000(0 in base 10) to 11111111(255 in base 10)
- Two's Complement
  - Make the most significant bit(最左边的那个) a negative number
  - 例子: 最小的"signed"的数字是10000000, 换算成十进制是(-1)*2^7 >>> -128 in base 10
  - 下表体现了一些基础的Two's Complement的二进制与十进制对应的数字，

- Two's Complement System for 1 byte:

binary	base 10
10000000	-128
10000001	-127
11111111	-1
00000000	0
00000001	1
01111111	127

    - 如果是在Two's Complement里面，那么他的计算是这样的
      - 举个例子：10010101
      - 这时候，只需把最左边的数字，即1，写成-1，然后其他的计算方式都是照旧的
      - 那么这个数字在十进制里面是-107
  - 所以这时候就知道, byte 这个integer type为什么是range from -128-127
  - 如果我们处于Two's Complement的规则里面，想要在最大的byte number，即01111111的基础上再加1
    - 如果是在普通的加法运算的话，这个加上1会变成10000000，即在十进制中是128
    - 但是在Two's Complement里面，这个加上1之后会变成10000000, 这时候会变成负数，即-128 in base 10
    - So integer numbers wrap around, in the case of overflow. No warning from Java!
- An int is stored in 4 bytes using Two's Complement
- An int range from:
  - 10000000 00000000 00000000 00000000 to
  - 01111111 11111111 11111111 11111111
  - 即在十进制里面是 -2147483648 to 2147483647