[redacted] solutions

Race condition

- Incorrect behaviour of a program due to concurrent execution of critical sections by two or more ~~threads~~ threads.

Ex: If thread A deletes an entry in a linked list while thread 2 is accessing the same entry.

- P1 & P2 are critical code at the same time produce a weird ~~situation~~ behavior.
- Race cond. in is a weird behavior due to the lack control.

We need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their critical section
2. No assumptions may be made about speeds or the number of CPUs
3. No process running outside its critical region may block other processes
4. No process should have to wait forever to enter its critical section

Deadlocks

- Deadlocks occurs when two or more processes stop making progress indefinitely because they are all waiting for some inter-dependent event to occur

Ex: A. If process A waits for process B to release a resource and

B. Process B is waiting for process A to release another resource at the same time

- In this case, neither A nor B can proceeded because both are waiting for the other to proceed

Resource deadlock

~~Solving Deadlocks~~

- Because two processes could be trying to allow each other to write but no one does it

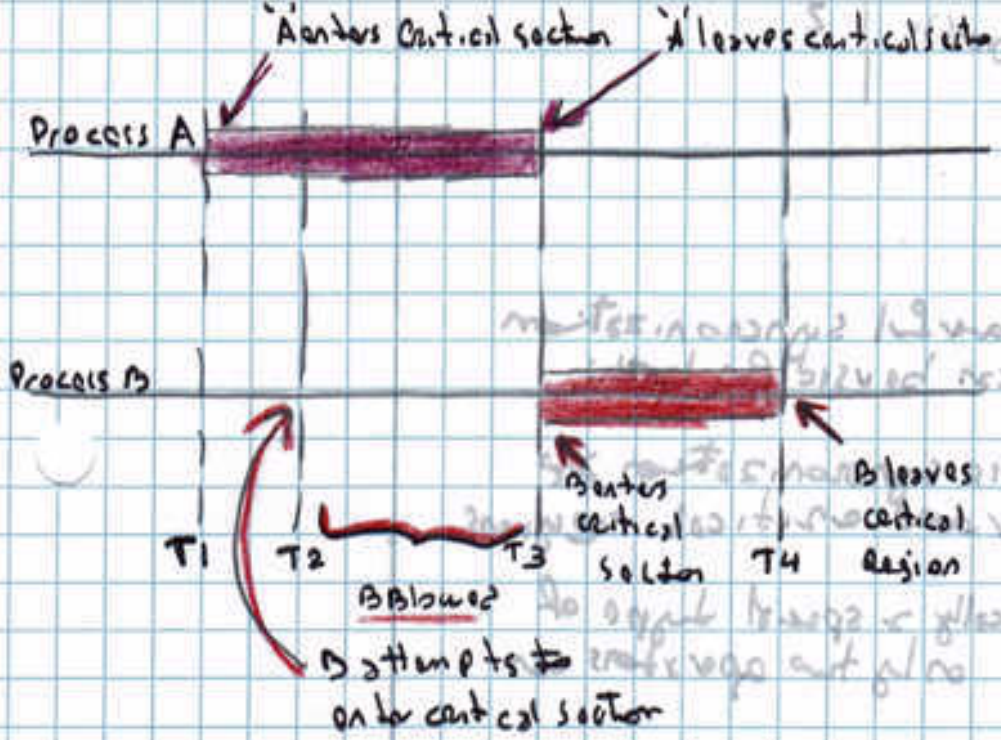
- Property of deadlocks is that there are used for multi-processes and it can be harder to debug

[redacted] solution

Solving Race condition

Mutual Exclusion

- way of making sure that if one process is using a shared variable or file, the other process will be excluded from doing the same thing



Four conditions

1. No 2 processes simultaneously in critical section
2. No assumptions made about speed or number of CPUs (often instructions per second)
3. No process running outside its critical region can block another process
4. No process must wait forever to enter its critical region
   ⇒ waiting forever indefinitely = deadlock or starvation

Mutual Exclusion w/ Busy Waiting

Process 0 Pseudocode

```
while (TRUE) {

    Aloop
    while (turn != 0);
    critical_region();
    turn = 1
    non_critical_region();
    }
}
```

Process 1 Pseudo-code

```
while (TRUE) {
    while (turn != 1)
    critical region();
    turn = 0;
    non critical section();
}
```

Semaphore

- semaphore is a powerful synchronization primitive that can be used for both:
    1. Inter-process synchronization or
    2. Locking around critical regions
- semaphore is basically a special type of integer on which only two operations can be performed:
    1. DOWN (sem)

    - or

    2. UP (sem)

[redacted] self study

The Down (semaphore) Operation

- Let say you down op on th. the process that is calling the semaphore op. will decrement the value of the semaphore
- If (sem > 0) then
  - A. this op. simply decrement the value of semaphore sem by 1 and the calling process continue executing
  - i. this is called a "successful" down operation
- If (sem == 0) then
  - A. this op. puts the calling process to sleep
    - i. Means: the calling process is placed in "blowed" state
  - B. the process continues to sleep until some other process performs an UP operation on the semaphore
  - C. At this time the process comes up and tries to perform DOWN again
  - D. If it succeeds, then it moves up (moves to "ready" state) and continue executing
    - i. Otherwise it goes back to sleep
- When the sem is 0 and one process do down to the semaphore it will not decrement the process that was trying to do the decrementation will be put on sleep.
- the process will be blown until the time some other processes increase the value of the semaphore
- if positive decrease it if already zero, put process on blow state until other process increase from zero.

The UP(sem) Operation

- This operation increments the value of the semaphore sem by 1.
- If the original value of the semaphore was 0, then UP operation wakes up any process that was sleeping on the DOWN(sem) operation.
- All woken up processes compete to perform DOWN(sem) again
  - Only one of them succeeds and the rest go back to sleep, still the next UP(sem) operation

- Let say 5 are blocked, you can't assume which is going to get up!

- If a process is blocked, then another process came then call up to the same semaphore
- Then the semaphore is going to check if a process is blocked for that semaphore
- then this process it will go up to try to put down the semaphore down again.

- UP & Down happens in the atomic (check value, change it and possibly going to sleep, are all done as a single indivisible atomic action)
- If you can guarantee UP & DOWN are atomic you can guarantee that the two processes are.

[redacted] self Study

Mutex

- Mutex is a simply a binary semaphore (0 or 1)
- Mutex is used as a lock around critical sections
- Locking a mutex : Down(mutex)
  1. if mutex = 1 then decrement value to 0
  2. else, sleep until someone performs an UP.
- unlocking a mutex - UP (mutex)
  - Increment mutex value to 1
  - wake up all sleepers on Down(mutex)
  - only one sleeper succeeds in acquiring the mutex. Rest go to sleep

Ex: Down (mutex) ← acquire the lock, sleep, if mutex is 0

crit. col section

UP (mutex) ← release the lock, wake up sleepers

- You have to give an initial value to the semaphore when creating. the value could
- mean a value of process to go inside a section. Any other process would not go inside the section:
  ie: 100 instance is available, but 101 process try to go in, it will not be allowed to go inside the section

Mutex (Continue)

- Any kind of blocking can be used with mutex.
- Mutex is kind like a lock of a door
→ If you lock mutex you blocks outside process to access a section.
→ If you unlock then they are allowed

[redacted] Final Review

- Lecture 11 to 17: Threads, segmentation, filesystems, input/output subsystems, RAID, Virtual Machines

Threads

1. What are threads?

Threaths are light-weight processes.

2. What are the shared/non-shared components of threads?

The shared components are: (All threads share)
a. Global memory: ~~it is share between all processes threads~~
b. Instructions: all threads can call any function
c. Host Data: (threads have their own version of variables
d. Open descriptors such as files, sockets, pipe, etc.
e. Signals and Signals handles: ie: kill signal would make all threads to die.
(f. tgid)
(e. heap is also share with all processes) signal is a function that delivery

The non shared components are: (All threads own their own)
a. Thread id
b. Registers, Program counter, stack pointer: Each thread can be executed in different part of the code or execute the same instruction at the same time. This means every thread have their own control flow
c. Stack: since threads can be calling functions on their own, every thread have their own stack. Arguments and return variables are stored in the stack
d. Errno: Errno is a global variable that provide errors from system call. Since different threads can be calling different system calls, each thread is provided their own error #.

⇒

e. priority: Threads can have different levels of priorities, so we can indicate the CPU which thread have more importance than other.

3. Advantages and disavantages

a. the advantages of using threads:

i. light-weight [which means Lower Context switching Overhead (LSCO) and fewer Os resources]

I. LCSO: when you switch from one thread to another thread, you only switch all the non-shared context. Therefore context switching is lighter, and fewer OS resources are consume

ii. Shared State: ~~there is no need~~ Threads don't need inter process communication mechanism (IPC)-like to communicate between threads of the same process

b. Disadvantages of threads:

i. shared state: Global variables are shared between threads. Accidental changes can be fatal

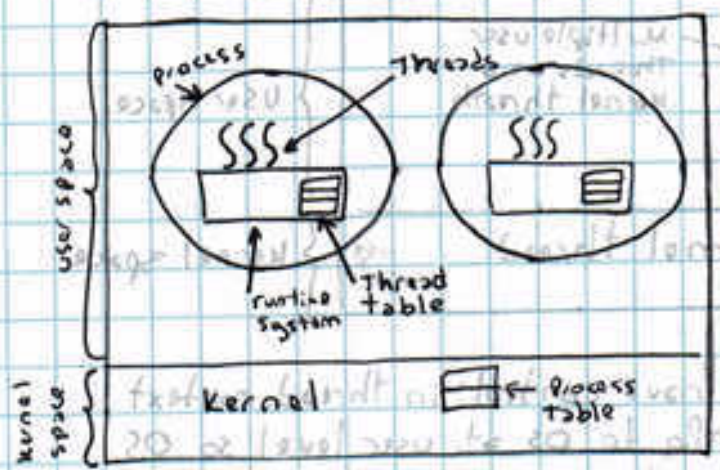
ii. Some libraries functions may ~~not~~ be thread-safe (if a library function store information in an specific place, two threads could try to write in at the same time)

I. gethostbyName() is an example of a library function that return pointes to static internal memory, in which case it could provide different results.

iii. lack of robustness: If one thread crashes then the entire process may crash too (one thread die then all threads die)

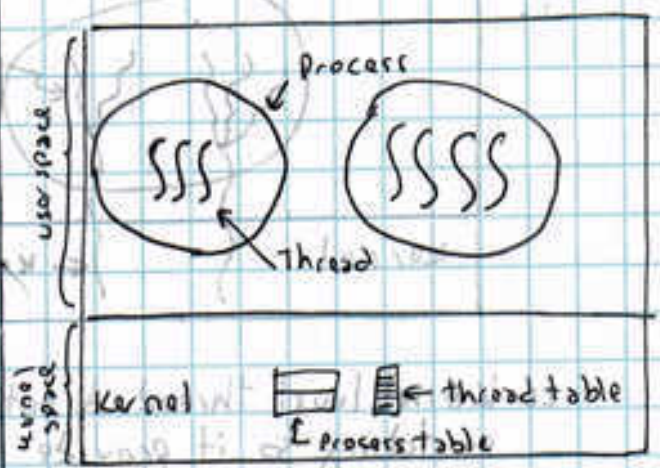
4. User versus kernel threads

user level threads



- user-level threads packages
- The process is giving the timeslide and the local scheduling. It decide how to use the time provided to the process.
- This take care of the context switching between threads
- The threads are dead at user-level while kernel doesn't have idea so the os doesn't get involved in the context switch
- Advantage is that is faster context switch between threads, bad disadvantage is that one thread that blow, can make blow all the other threaths and blow the process.

Kernel level threads



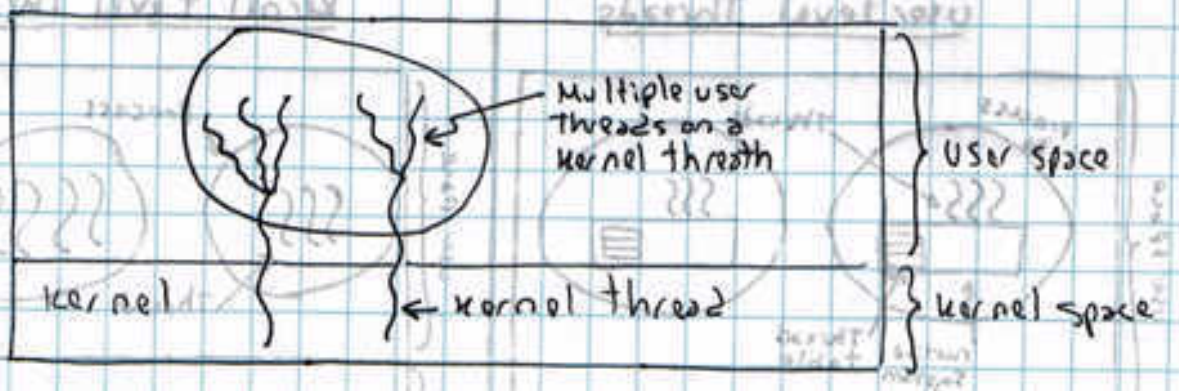
- Threads packages managed by the kernel
- The kernel is choosing which thread should be run by itself, when switch threads.
- every thread is given a timeslide, when a thread have expired its timeslide, the kernel would switch to the next thread
- The kernel can choose to do local user scheduling or local kernel scheduling
- Multiple thread switching is more expensive, ~~maintain~~ because manage privilege switching is done when doing process context switching

(user level have to do the Privilege?)

5. Local versus global versus hybrid thread scheduling

a. Hybrid Implementation: Multiplexing user-level thread within each kernel-level thread

i.



ii Kernel level thread do not have control in thread context switching so it provide info to OS at user level so OS can response to kernel level

b. Local Thread scheduling: When the next thread to be scheduled is picked from ~~among~~ among the threads belonging to the current process, each process gets a timeslice from kernel. Then the timeslice is divided up among the threads within the current process

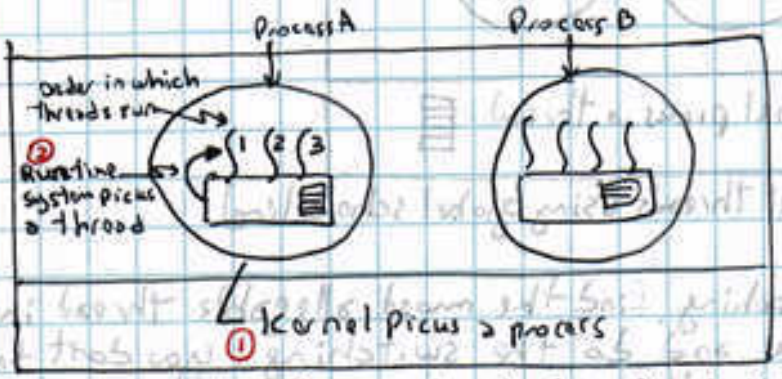
i. Local scheduling can be implemented with either Kerner-level threads or user level threads. (user level threads by default is local thread scheduling

I. If you have user level thread, you can only do local thread scheduling because the kernel does not understand that there are different threads.

ii. scheduling decision requires only local knowledge of threads within the current proces

iii. with local scheduling, when you have to decide which you ~~have to decide which thread you have to decide~~ give to the OS next
P1 → 5 threads
50ms
P2 → 5 threads
the OS schedule decide to take out one thread P1 or P2.
With local scheduling the os can decide, let's say, one thread of P1 take 10ms. when a thread in P1 process is given a time slide, you pick the thread that have the same time slide, after 50 ms is expired you go to P2
- If one of the threads blocks, then this thread is giving up its own process time slide

iv.



Possible: A1, A2, A3, A1, A2, A3
not possible: A1, B1, A2, B2, A3, B3

(user-level threads using local thread scheduling)

I. when it is time to do a context switch, the local thread scheduling can pick any thread in any order

II. with local schedule you stick with threads in the process until the process expired

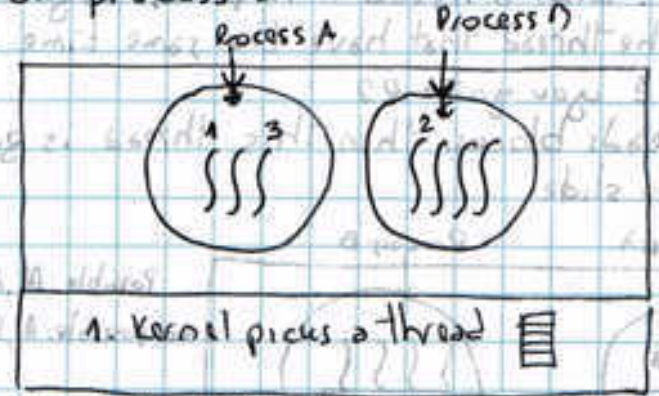
III for example: say process timeslice may be 50 ms and each thread within the process runs for 5 msec / CPU burst

120 c. Global thread scheduling: when the next thread to be scheduled is picked up from any process in the system, not just the current process

i. Timeslice is allocated at the granularity of threads, ~~not~~ (there is no notion of per-process timeslice)

ii. Global scheduling can be implemented only with kernel-level threads Picking the next thread requires global knowledge of threads in all processes.

iii.



Possible: A1, A2, A3 A 1, A2, A3
Also Possible: A1, B1, A2, B2, A3, B3

(kernel-level threads using global scheduling)

I- Global switching, find the most allegable thread in all the processes and do the switching, you don't try to be fair with each process
Ex. Real time
note: definition of allegable means the most important process or could mean the most important thread that didn't receive process time for a long time

II. In some cases, the kernel doesn't want to distinguish between processes and threads.

III. For example each thread runs for 10m sec/CPU

[redacted] final review

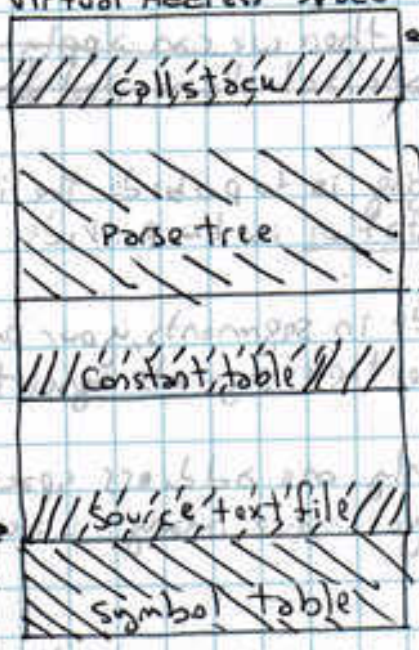
Segmentation

1. What is segmentation?
   - Segmentation is a Memory Management technique in which memory is divided.

~~seg~~

2. Usefulness of segmentation
   a. Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments which:
      i. May have dynamic size
      ii. simplify handling of growing data structure
      iii. Allow programs to be altered and recompiled independently
      iv. Lends itself to sharing data among processes
      v. With segmentation you get isolation between different properties of the program.

I. virtual Address space



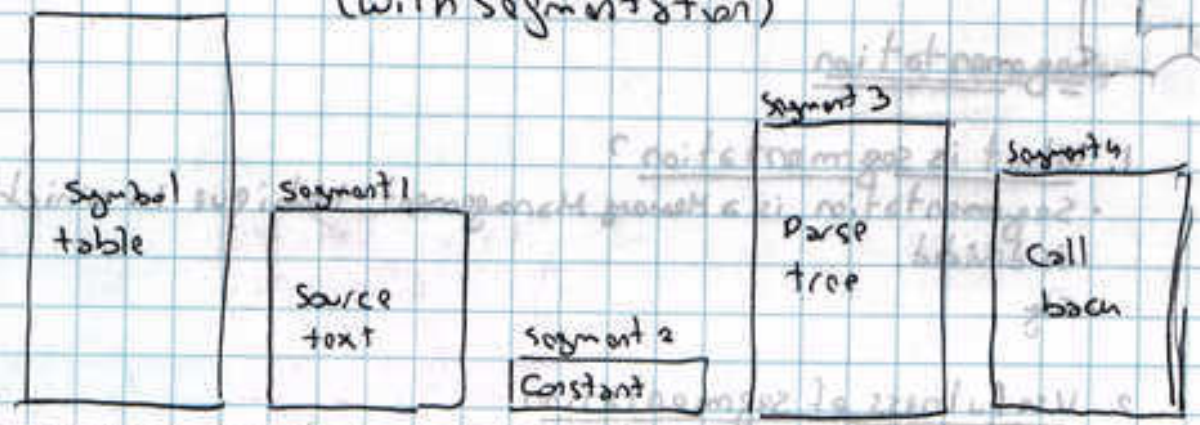
- One-dimensional address space with growing tables
- One table may bump into another

(without segmentation)

⇒

segment 0

(with segmentation)



- Segmentation allow each table to grow or shrink independly without affecting each other

NOTE: you can have segmentation and paging implemented.

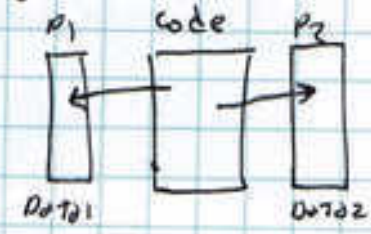
~~- The~~

~~3. How segmentation and paging work together?~~

~~If segments are large then we can apply paging so only those pages that are actually needed to be around~~

3rd - The reason of using paging is to provide the illusion of large memory, while segmentation is to provide isolation

- If you want to divide in segments, your program have to indicate how you want to divide these segments since the system will don't know
- Normally you will have only one address space with paging, but with segmentation you have many. Each address space belong to each program
- Segmentation Sharing: P1 & P2 running the same program. you have one code segment that contains the instructions of the program (which belong to P1 and P2) but each process could have their own data segment because each process would have their own data



45. Comparison between paging and segmentation

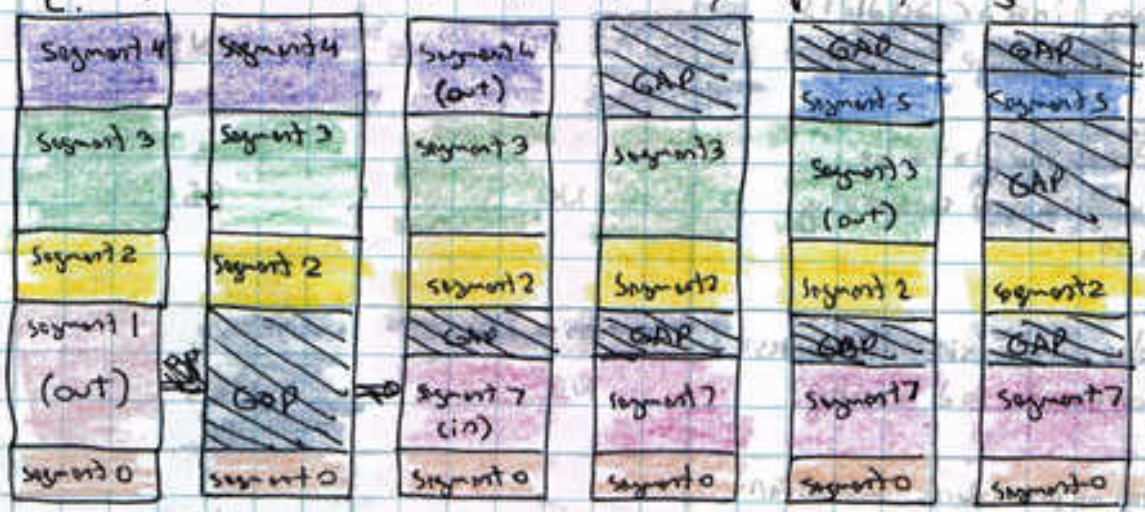
Consideration	Paging	segmentation
Need the programmer be aware that this technique is being used?	NO	YES
How many linear address space are there?	1	MANY
Can procedures and data be distinguished and separately protected?	NO	YES
Can tables whose size fluctuates be accommodated easily?	NO	YES
Is sharing of procedures between users facilitated	NO	YES
Why was this technique invented?	so you can get the illusion to have more memory that the actual memory in the system to get a large linear address space without having to buy more Physical memory	segmentation you can take a program, divide it in different components isolated. Also it can help to share to allow programs and data to be broken up into logically independent address space and to aid sharing and protection
Can the total address space exceed the size of Physical memory?	YES	YES

6. Implementation of Pure segmentation

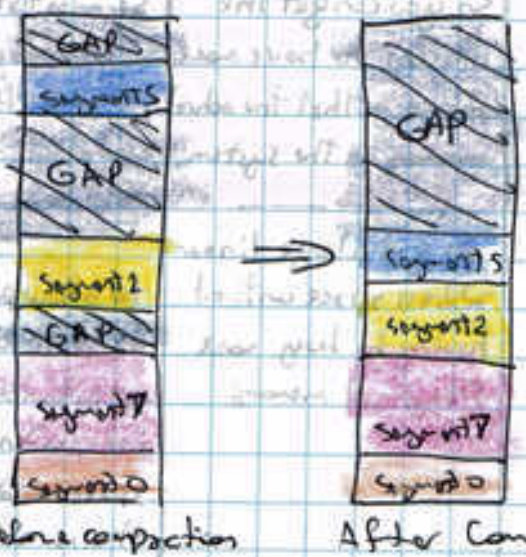
a. The implementation of segmentation differs from paging in that pages are fixed size and segments are not

b. After the system have being running for a while the memory will be divided up into a number of chuncks, some containing segments and some containing holes. This is called checker boarding or external fragmentation (waste of memory)

c.



d. Solution is to remove the external fragmentation by compaction



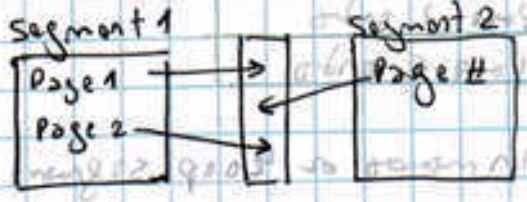
Before compaction After Compaction

e. external fragmentation produce an effect ~~that~~ in which the free space cannot being use by other segment

[redacted] final review

7. How segmentation and paging works together?

- If segments are large then we can apply paging so only those pages that are actually needed be around
- Basic idea: Assign a page table to each of these segments. The main memory is organized in pages, and these pages are organized in segments
- Therefore, we divide a segment into pages. Then we don't need all these pages next to each other in memory except the pages needed.



8. Segmentation with Paging: MULTICS

a. Each program is provided with a virtual memory of up to 2^{18} segments, each which could be up to 36 bits words long (65,536 words long)

b. Each segment as a virtual memory and page it

c. Each MULTICS progra has a segment table, with one descriptor per segment. The segment table is itself a segment and is paged
   i. A segment descriptor contains an indication of whether the segment is in main memory or not.
   I - If any part of the segment is in memory, the segment is to be considered in memory, and its page table will be in memory.
   II - If the segment is in memory, its descriptor contains a 18 bit pointer to its page table

⇒