# CS2106 Live Class: Disjoint memory schemes

$_/|/|U_Ch@NgRu!$

May 10, 2021

In this section, process memory spacvve can now be disjoint physical memory locations. This can be done via paging and segmentation

## 1 Paging

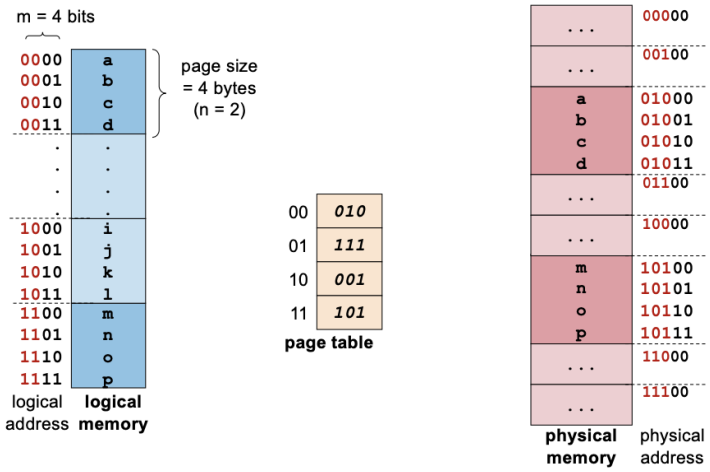
The **physical memory** is split into regions of fixed size, known as physical frame. The logical memory, which has the same size with physical memory, is also split into regions of same size, known as logical page.
During the execution, pages of a process is loaded into any available physical frames. The process will still occupy a continuous logical memory space.
Under paging scheme, we will keep a mapping of a logical page to the physical frame using **page table**. Essentially, the logical memory of a process always start with page 0, so the page table Table can be a 0-based index array.
Suppose the physical frame size, which is also page size, equals S, then the physical address of kth byte of ith page equals Table[i]×S+k
There are also two practical techniques

1. Keep frame size and page size as power of 2
2. Physical frame size is also equals to logical page

Suppose we have $2^n$ bytes in a page and m bits of logical address, then the logical address can be represented by $\underbrace{p}_{m-n}\underbrace{d}_{n}$
where

1. p=Most significant m-n bits of logical address(LA), and is used to find frame number f with page table
2. d=Remaining n bis of LA
3. PA=f*$2^n$ +d

The corresponding physical address is $\underbrace{f}\underbrace{d}_{n}$, where f =Table[p]




In the picture above, the first two bits of the page address is the most significant m-n bits of LA
E.g the b is the second bytes(indicated by the last two bits of the logical address) in the page 00(indicated by the first two bits of the logical address)
With page table, it can be found that the physical frame corresponsign to page 00 is 010, and hence, it can be found that the physical address of b is 010—01=01001

Paging removes external fragmentation, but there exists, still internal fragmentation(because the continuous memory space that one process needed may not be multiple of memory page).But the internal fragmentation is relatively not very serious, because only the last page may have internal fragmentation
The advantage of clear separation of logical and physical address space

1. Allow great flexibility
2. Simple address translation

# 2 Implementation of paging scheme

1. Common pure-software solution:
   (a) OS stores page table information with the process information(e.g.PCB)
   (b) Improved understanding: memory context of a process includes page table(or pinters to it; today's page tables are quite big)
   (c) require two memory access for every memory reference

   The problem of the fostware implementation: require two memory access for every memory refernce: 1st is to read the indexed page table entry to get frame table; second is to access the actual memory items
2. Hard-ware implementation: Translation look-aside buffer

### 2.0.1 Exercise

```
load r1, [2106];
inc r1;
```

```
store r1, [2106];
```

The number of reference to memory be executed: 8 or more

1. 2 access at least to get the data and store the data
2. each time access the memory we need to access the page table first, hence plus 2
3. Both the code and the data is in the memory in VonNewman's machine, so, to execute the code
   (a) get the instruction 1*3
   (b) each time get the instruction, need to visit the page table first 1*3

2+2+3+3=10
In reality, it can be worse if the full instruction or data needs more than one time to get

## 2.1 TLB

Translation look-aside buffer is a cache of a few page table entries
TLB works as below:

1. Page number is used to search TLB associatively(in parallel)
2. If TLB hit, then frame number is retrieved for translation
3. If TLB miss, memory access is invoked to access the full page table, update TLB and do translation(procople of locality)

### 2.1.1 The effectiveness of the TLB

Suppose TLB access takes 1ns, while memory aceesses takes 50 ns. If the TLB hit rate is 90% according to memory,
Memory access time=P(TLB hit)*Latency(hit) + P(TLB miss)*Latency(miss)=90% × (1ns+50ns)+10%(1ns +50ns +50ns)=56 ns When a context switch occurs, TLB entries are flushed. When process is switched back, TLB miss will occur to fill TLB(TLB is a kind of hardware context of process)
In today's system, the TLB hit rate should be at least 99%, and there may be a TLB hierachy(L1,l2 AND l3 TLB)

## 2.2 TLB and context switch

1. The process's data would not be handled during the context swtich
2. Process's locgical memory should not be saved, because the data in the logical memory is the same as that is in the physical memory
3. Process's page table should be saved, otherwise, the physical memory of the data cannot be found
4. Process's TLB content: There is not a private TLB for the process, and the storage of TLB will not be done in practove. it's possible that the TLB can be stored in a relatively high priority cache or memory for later use(but it's very expensive and may not be good for integrity and accuracy, so no one do it in practice); Hence the TLB is usually not considered as hardware context
5. When a context switch occur the TLB is flushed(TLB is considered as one hardware context during the constext swithch)

6. When a process resumes, many TLB misses will be encountered to fill the TLB and it is possible to place some entries initially, this is one reason the context switching itself takes time

The number of TLB: usually every core has two TLB: data TLB(dTLB) and instruction TLB(iTLB) so that the data and the instruction are not missed(relatively the iTLB can usually store more than dTLB)

# 3 Protection

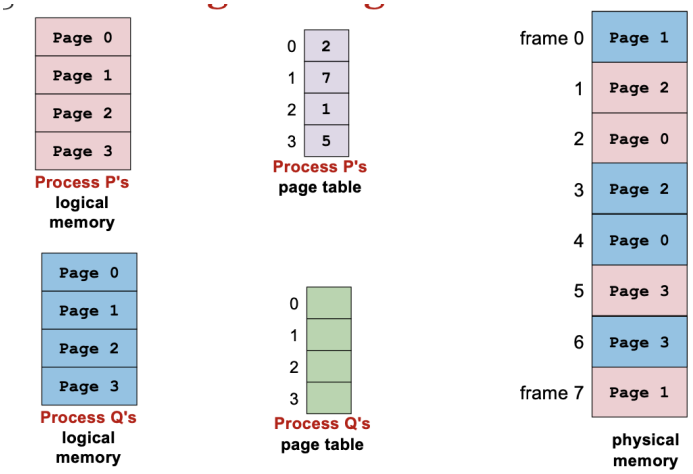
Some extra bits can be added to provide protection to memory

1. Access-right bits: attached to each page table entries to indicate whether it is writable, readable or executable. Memory access is checked against the access right bits
   Each page table entry has writable, readable.executable bits, if the file is just readable, segmentation fault will be sent

2. Valid bit: is attached to each page table entry to indicate whether the page is valid for process to access. This bit is set by OS and out-of-range will be caught
   Observation: the logical memory range is usually the same for all processes,However, not all process utilized the whole range, that's why some pages are out of the process range
   The page table allows several processes to share the same physical fram, by putting the same physical frame number in page table entries. Hence it can be used to share code page(multi-thread) and implement copy-on-write
   For examle, if the page is in the range of the process, the valid bit may be set as 1, and if out-of-range, it will be set as 0; If the process try to visit a page with valid bit as 0, a segmentation error will be invoked

## 3.1 Page sharing

In practice, one page can be shared by multiple processes




To achieve sharing, we just need to write the page table so that the process Q's certain page number will be pointed to the same physical address as some page number of process P

This is very useful especially when the logical memory is somthing needing to be shared widely such as STDl
A very interesting exaple is copy-on-write: when the fork() function is called, the child process basically just copy the page table so that the content would be the same and the child process and parent process will acutually access the same memory

# 4 Segmentation scheme

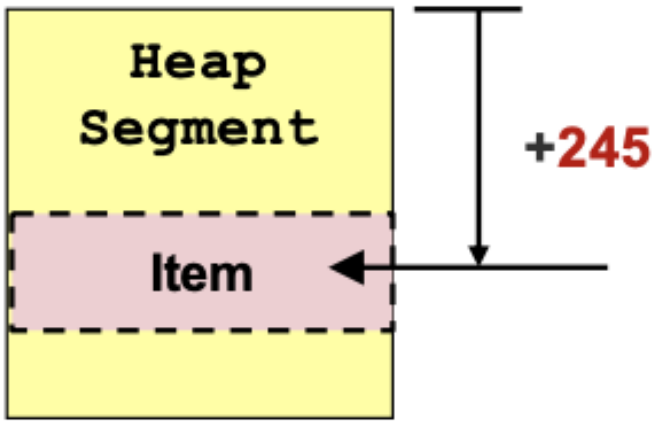
It is hard to place different regions(data,text,stack, heap) in contiguous memory space, physical or logical and allow growing/shrinking freely and also should not be in physical level.
Moreover, different part may have different usage pattern

1. Text: read and executable
2. Data: Read and write, shared for threads
3. Heap: Read and write, shared for threads
4. Stack: Read and write, private for threads

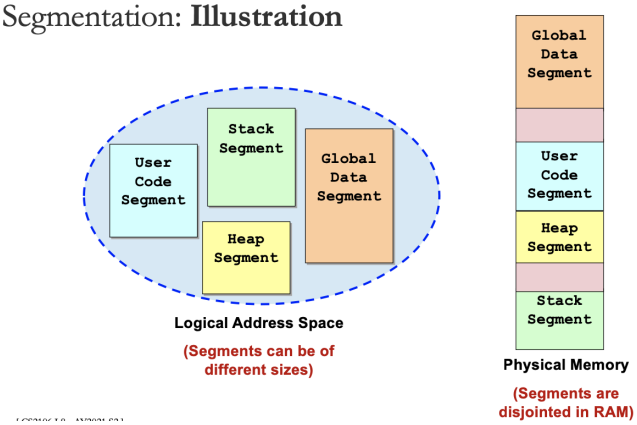
Some regions may grow/shrink at execution time

1. Gaps in-between should be provisioned
2. Gaps are inefficiently handled by page-table

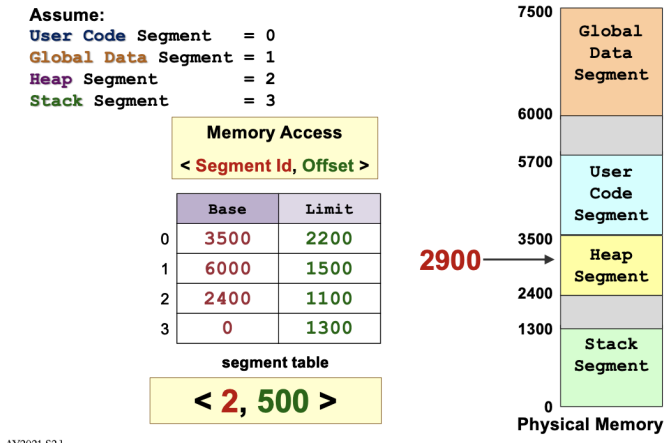
Therefore, we use segmentation scheme, which separate regions into multiple memory segments. Logical memory space is then a collection of segment.
Each memory segment will have a name, and have a limit to indicate the range. The memory reference is now specified as segment name+offset(e.g "Heap"+245)



## Segmentation: **Illustration**

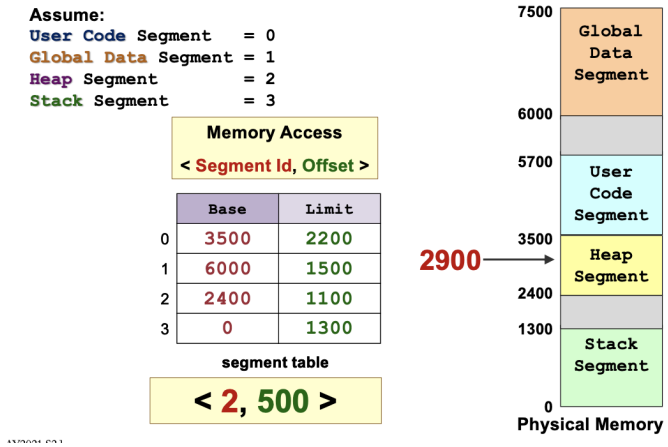



Each process will keep track of a **segment table**, which is a mapping between different segments, identified by segID to a pair: ¡Base,Limit¿. Suppose the logical address is SegID with offset k, then the physical address is Table[SegID]+k, if k is less than limit. Since we need to check limit and also do addition for offset, we do not hardware to support these opeations




Here if one process try access 2,500, $it will first check the physical base with the segment table and found it's 2400 with limit 1100($

## 4.1 Hardware support




There can be a TLB-like registers in the CPU, which can tells the base and limit of each types of segment

## 4.2 Evaluation of segmentation

### 4.2.1 Pro

1. each segment can grow/shrink independently
2. each segment is an independent contiguous memory space
3. can be protected/shared independently
4. More efficient book keeping
5. Naturally matches programmer's view of memory

### 4.2.2 Cons

1. Segmentation requires variable-size contiguous memory regions, which may result in external fragmentation

Today's system is usally using pageing but the segment may be comming back

# 5 Segmentation with paging

The intuitive next step is to combine segment with paging
Basic Idea:

1. Each segment is now composed of several pages instead of a contiguous memory region
2. Essentially, each segment has a page table
3. Segment can grow by allocating new page then add to its page table and similarly shrinking

The problem of segmentation with paging is that multiple memory query needed for one memory access, which may make the memory access very slow