Memory Management
Part 1

Computer Operating Systems
BLG 312E

2016-2017 Spring

Memory Management Unit

- memory is a critical resource
  - efficient use
  - sharing
- **memory management unit**

Memory Management Unit: Main Aims

- relocation
  - physical memory assigned to process may be different during each run
  - physical and absolute adresses should not be used
- protection
  - process cannot access another process' memory area
- sharing
  - code / data sharing

Memory Management Unit: Main Aims

- logical organization
  - in traditional systems: linear adress space (0→max)
  - programs: written as modules / procedures
- physical organization
  - transfers among main memory and secondary storage

Memory Management Functions

- naming (N)
  - user defined variable → actual location referenced
- memory (M)
  - variables → physical adres
- contents (C)
  - obtain contents of memory locations from address

$$\text{user variable} \xrightarrow[N]{} \text{system variable} \xrightarrow[M]{} \text{memory address} \xrightarrow[C]{} \text{value}$$

Memory Management Functions

- functions performed at different times
  - N: during link phase
  - M: during load phase
  - C: during assignment/retrieval to/from memory

Linkers and Loaders

- aim: binding abstract names to concrete names
- actions performed:
  - symbol resolution
  - relocation
  - program loading

Symbol Resolution

- references from one subprogram to another made through symbols
- linker resolves symbols
  - notes location assigned to called function
  - patches caller's object code
  - e.g. “main” function calls “sqrt” function defined in math library
    - linker finds location assigned to “sqrt”
    - modifies “main” object code so *call* instruction references location

Relocation

- compilers and assemblers generate object code starting at 0
  - all subprograms loaded at non-overlapping locations
- linker creates linked output starting at 0
  - subprograms relocated to locations within complete program
- loader picks actual load adress
  - linked program relocated as a whole

Program Loading

- loader copies program from secondary storage into main memory
  - allocate storage
  - copy data from disk to memory
  - set protection bits
  - arrange virtual memory maps

Final Address Binding

- before OS
  - each program had entire memory
  - assembled and linked for fixed memory addresses
- with OS
  - programs share memory with OS and other programs
  - actual addresses not know until program is loaded
  - final address binding is deferred to load time

Dividing Work

- linker does part of adress binding
  - assigns relative adresses within each program
- loader does final relocation step
  - assigns actual adresses

Multiple Programs

- computers run more than one program
  - frequently copies of same program
  - some parts of program are same among all running instances
  - other parts unique to each instance
- separate same and different parts
  - use single copy of same parts

Linking Multiple Sections

- compilers and linkers generate object code in multiple sectors
  - read-only code section
  - writable data section
- linker combines all
  - all read-only codes together
  - all writable data together

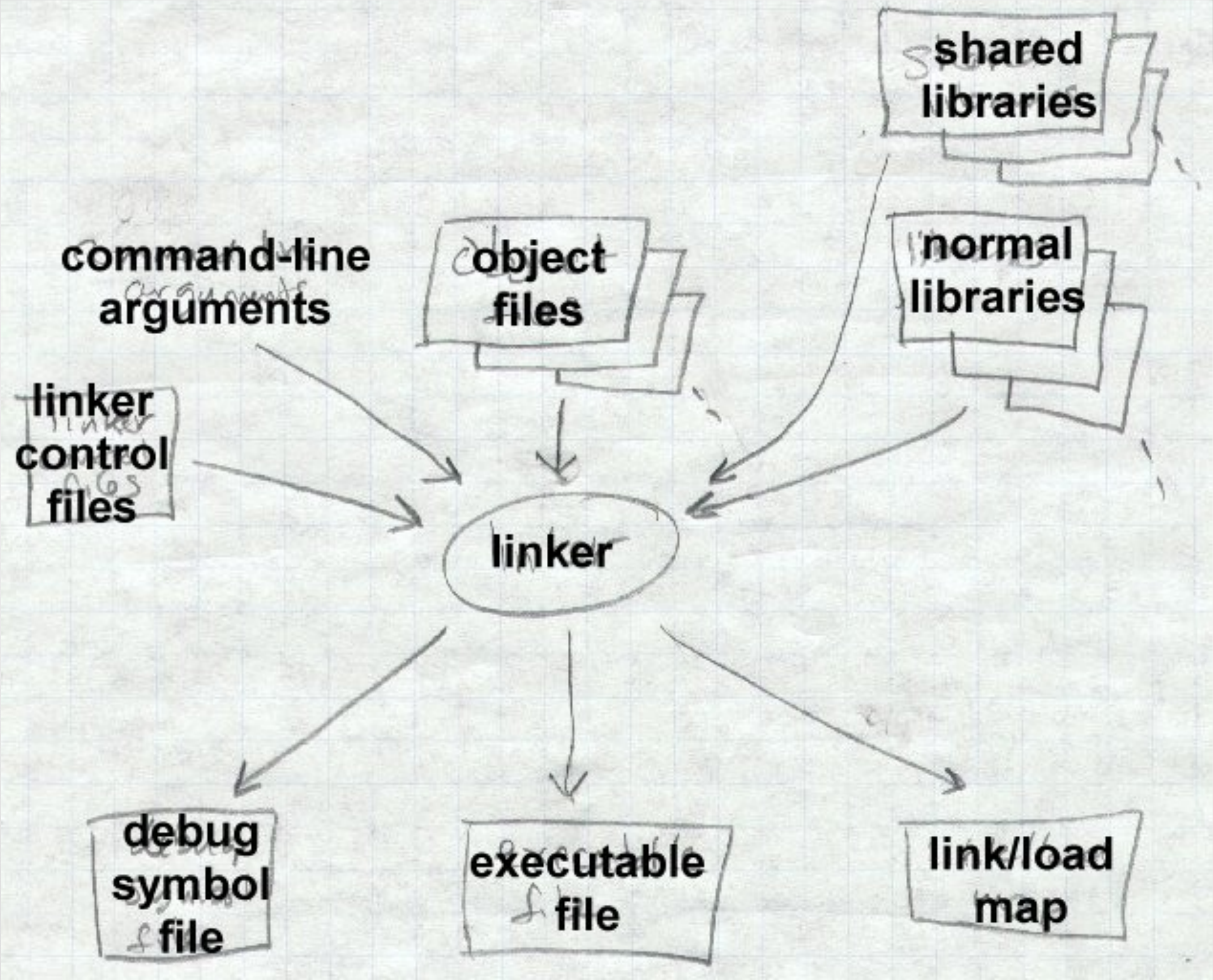
Shared Libraries

- different programs share a lot of common code
  - library routines
    - e.g. printf, fopen in C
- modern systems provide shared libraries
  - all programs share same copy of library
    - improves runtime performance
    - saves disk/memory space

Two-Pass Linking

- input: set of object files
  - each input file contains a set of segments
  - libraries
  - command files
- output: executable object file
  - load map, debugger symbols, ...

Linker Input and Output



Symbol Table

- each input file contains a symbol table
- exported symbols
  - defined within file for use in other files
  - names of routines within file that may be called from elsewhere
- imported symbols
  - used in file but defined elsewhere
  - names of routines called but not present in file

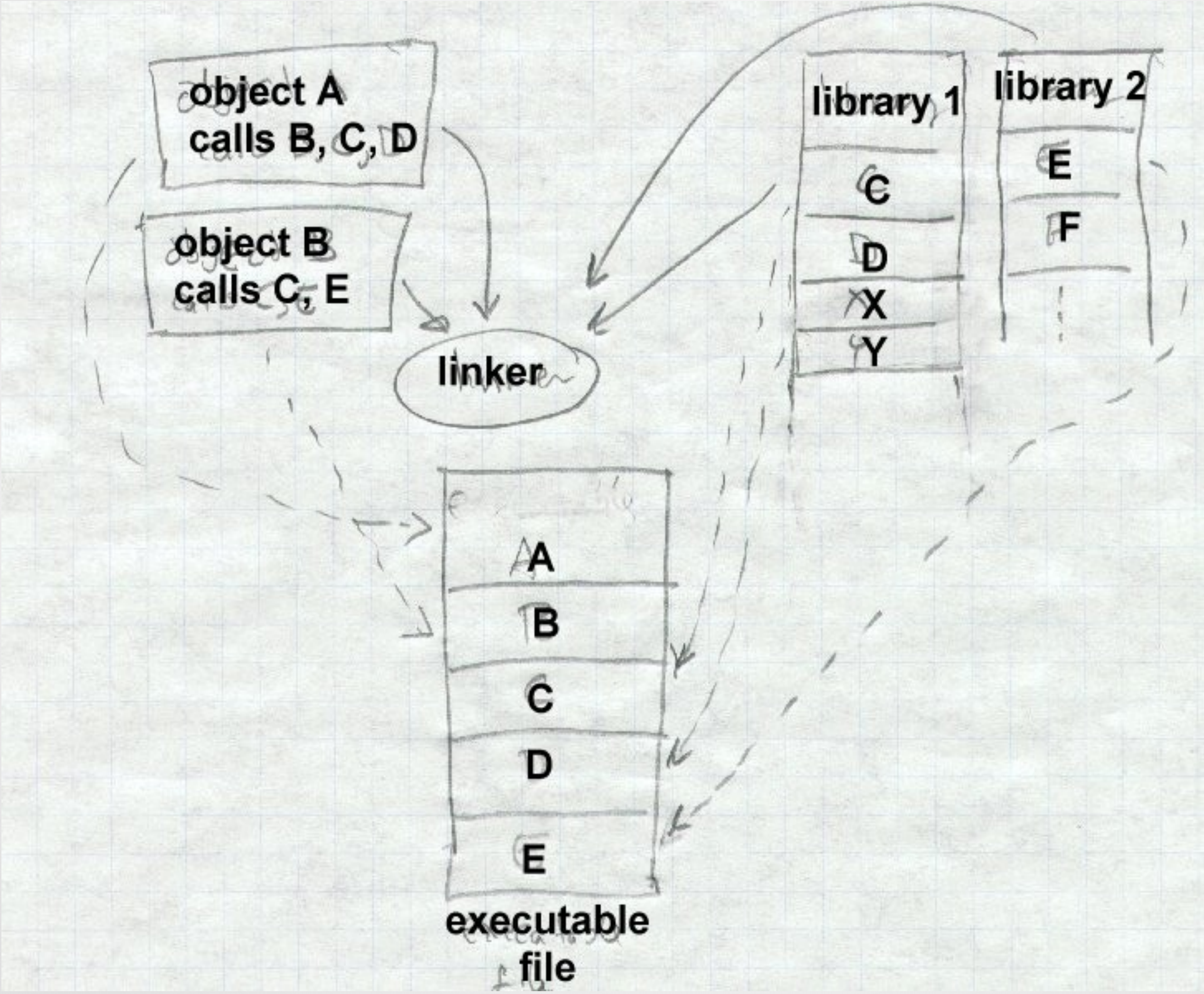
First Pass

- scan input files
  - find size of segments
  - collect definition and references of all symbols
- create:
  - segment table: all segments defined in all input files
  - symbol table: all imported and exported symbols

Second Pass

- use data from first pass:
  - assign numeric locations to symbols
  - determine size and location of segments in the output addres space
  - substitute numeric addreses for symbol references
  - adjust memory addreses in code to reflect relocated segment addreses
  - after all regular input files processed, if any imported names remain undefined
    - run through libraries
    - link required libraries

Linking Libraries



Allocating Memory

- memory allocation: allocate memory to program
- not required to have whole program in memory
  - load as required
  - more efficient memory usage
  - more costly

Static / Dynamic Memory Allocation

- programs with absolute adresses
  - give absolute adresses when writing program (M and N together)
- symbolic programming
    - compiler / linker generates memory adresses from symbolic names

Static / Dynamic Memory Allocation

- memory allocation:

static:
addreses
fixed
when loading
into memory

- generate fixed absolute addreses
  - linking and loading together with compiling → fast)
- use relocatable addreses
  - loader determines absolute addreses
  - addreses remain fixed during execution
  - code remains constant in memory after loading

dynamic

- use relocatable addreses
  - gets absolute addres when referenced

modern operating systems use

“segmentation” + “paging”

Segmentation

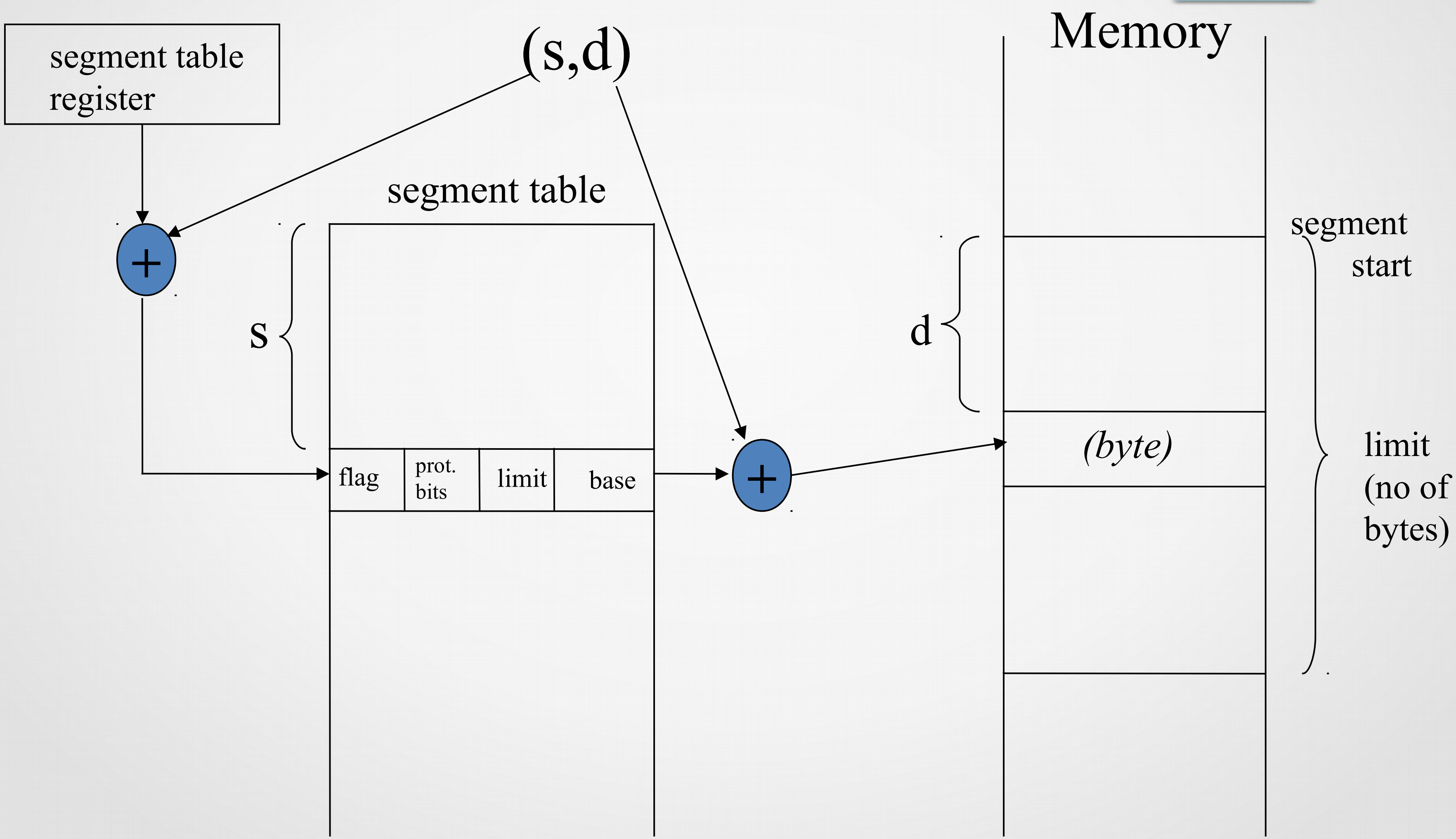
- programs composed of logical parts
- segmentation reflects logical structure of programs
- program divided into segments
  - segment sizes may be different
  - e.g.
    - data area as a segment
    - a procedure / function as a segment

  address = segment start address + offset in segment
- program segments may be in different memory locations
  - may be on disk too (loaded when required)
  - adress calculation requires special hardware

Segmentation

- adress: (s,d)
  - s: segment name
  - d: offset
- each process has a segment table
  - flag: is segment in memory?
  - base adress of segment
  - segment length (limit)
  - protection bits
- starting adress of segment table kept in a register

Segmentation



Segmentation

- check flag before adress calculation
- “segment fault” if not in memory
  - interrupt
- segment loaded into memory
  - if no room in memory, remove another segment from memory
    - segment sizes may be different → fragmentation in memory
  - segment table register points to start of segment table of running process

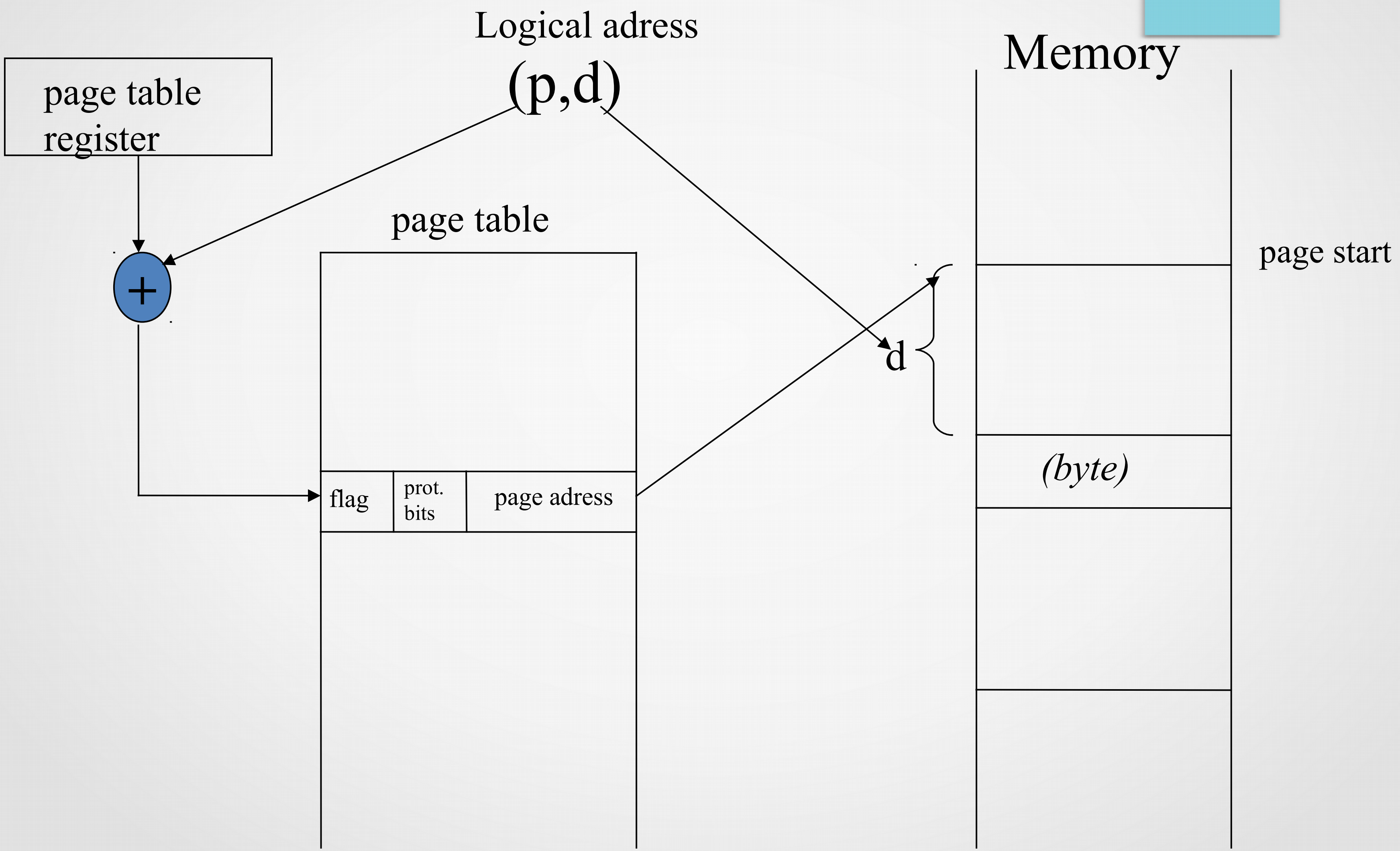
Paging

- memory divided into equal sized blocks
  - page frame
- program and data also divided into same sized logical blocks
  - page
- a page is loaded into a page frame
- adress: (p,d)
  - p: page name
  - d: offset in page

Paging

- page info in page table
- page table entry:
  - flag: is page in memory ?
  - page location (memory/secondary storage adress)
  - protection bits
- page table register
  - points to start of page table of running process

Paging



Logical adress
(p,d)
page table register
+
page table
flag
prot. bits
page adress
Memory
page start
d
(byte)

Paging

- check flag before adress calculation
  - "page fault" if not in memory
  - fetch page from secondary storage
- check protection bits
- operating system keeps list of free page frames
- main memory ⇔ secondary storage page transfers = **page traffic**

Paging

- memory allocation easier than in segmentation
  - fixed page size
- problem: page size may be smaller than a program logical block
  - more than one page
  - fragmentation

Fragmentation

- external fragmentation
  - empty spaces between blocks
- internal fragmentation
  - empty spaces within blocks
- with paging:
  - internal fragmentation may occur
  - no external fragmentation
- with segmentation
  - external fragmentation may occur
  - no internal fragmentation

Page Size Selection

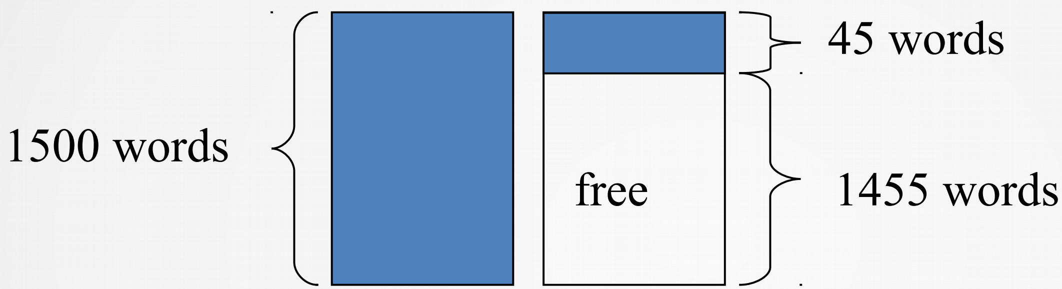
- criteria for page size selection:
  - page traffic
  - internal fragmentation
- large page sizes
  - easier main memory ⇔ secondary storage transfers
  - process has less pages ⇒ less page traffic
  - more internal fragmentation
- small page sizes
  - more page traffic
  - less internal fragmentation

Result: balance internal fragmentation and page traffic costs

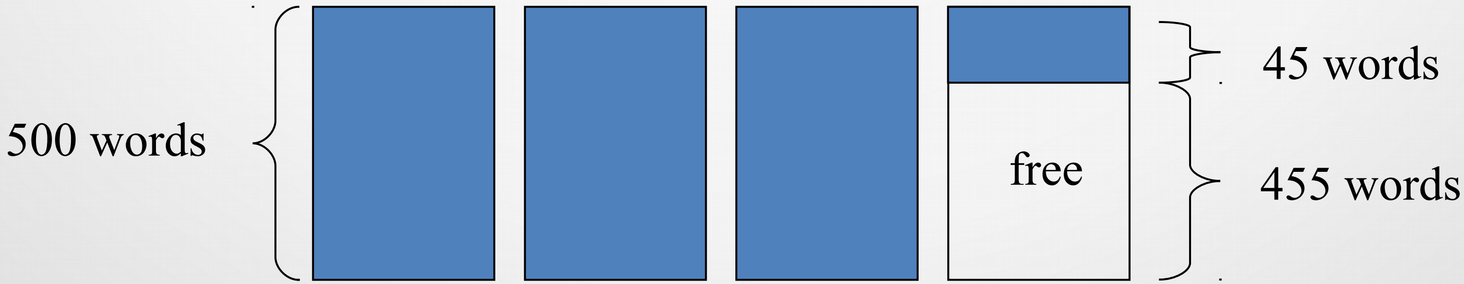
Example

Process size 1545 words

- if page size = 1500 words: process has 2 pages

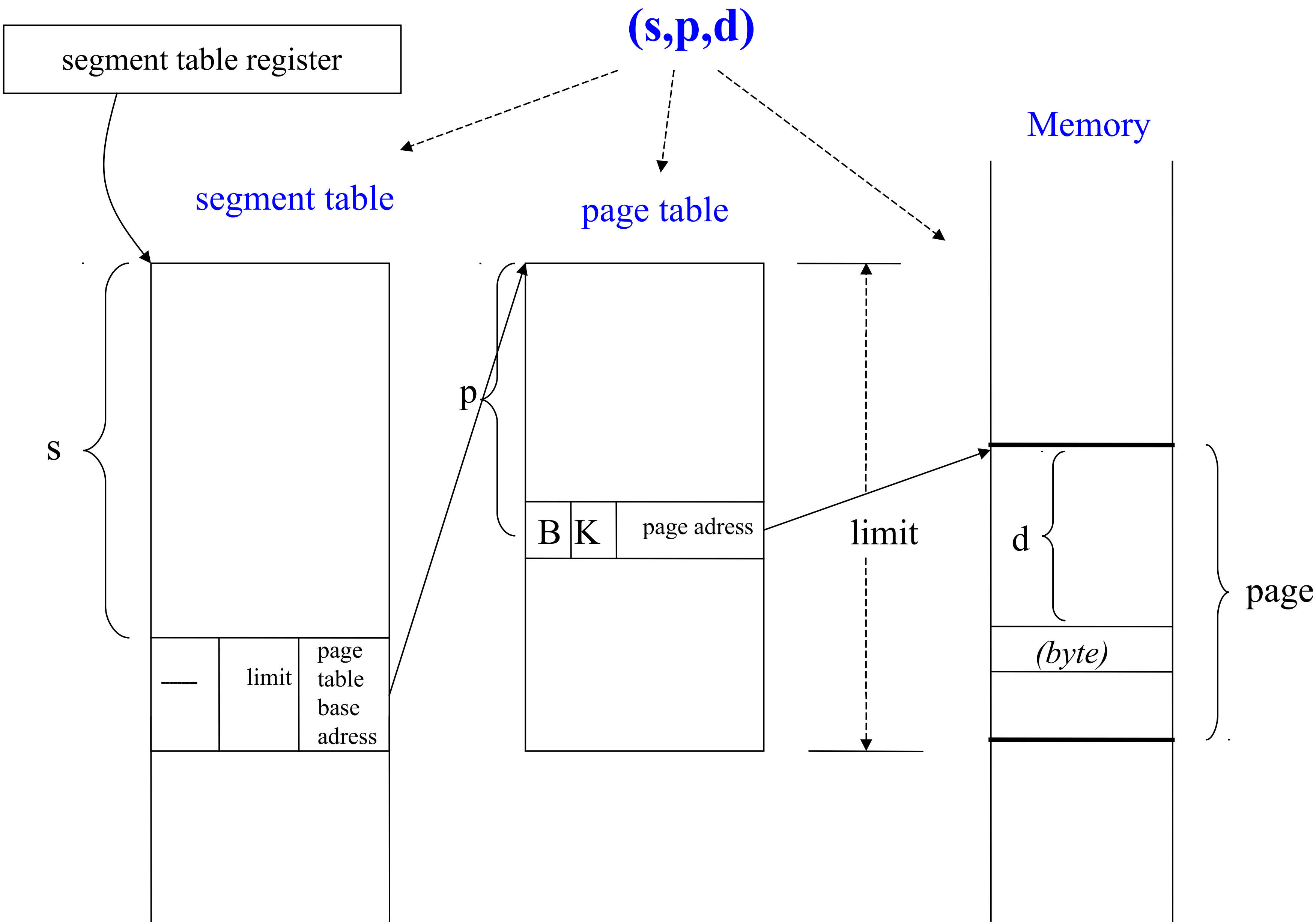


- if page size = 500 words: process has 4 pages



Segmentation with Paging

- segments divided into pages
- each segment has page table
- adress: (s,p,d)
  - s: segment name
  - p: page table access info for segment
  - d: offset in page


(s,p,d)
segment table register
Memory
segment table
page table
s
p
—
limit
page
table
base
adress
B
K
page adress
limit
d
page
(byte)

Segmentation with Paging

- 3 step adress calculation
- time consuming even when hardware used
  - associative registers used ⇒
    TLB (Translation Lookaside Buffer)

Segmentation with Paging

- has advantages of both segmentation and paging
- easy memory allocation due to paging
- no external fragmentation
- through TLB use, adress calculation times become acceptable