

Sharing

- Overall outline
 - *Preparing a program for execution*
 - *Virtual memory, algorithms*
 - *Shared objects*
- Today
 - *Shared objects*



Sharing procedures and data

- Software development is simplified if individual modules can be developed separately and linked together
- Ideally, the linker should not require modules to be recompiled each time they are included in a user's address space
- It is even better if a module can be linked into more than one address space at the same time



Examples of data sharing

- *Files and directories*: even if files are not shared, their directories usually are
- *Status databases*: the status of resources (free or busy)
- *Critical sections*: used to control data sharing on a sequential basis



Examples of procedure sharing

- *Kernel routines:* for example, all processes share the set of IO programs and device drivers
- *Utilities:* debuggers, linkers/loaders are often shared on a system-wide basis
- *Compilers and text editors*
- *Applications programs*



Linux “top” program

6:27am up 12:41, 7 users, load average: 1.49, 0.52, 0.30
104 processes: 97 sleeping, 6 running, 1 zombie, 0 stopped
CPU states: 99.5% user, 0.3% system, 0.0% nice, 0.0% idle
Mem: 256884K av, 253696K used, 3188K free, 225024K shrd, 24788K buff
Swap: 265032K av, 1096K used, 263936K free 154008K cached

PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	LIB	%CPU	%MEM	TIME	COMMAND
6925	jyh	7	0	284	284	220	R	0	99.3	0.1	0:43	fib
6998	takako	17	0	284	284	220	R	0	33.7	0.1	0:06	fib
6999	takako	14	0	284	284	220	R	0	32.7	0.1	0:06	fib
7000	takako	14	0	284	284	220	R	0	32.5	0.1	0:05	fib
1046	jyh	1	0	572	572	312	R	0	1.1	0.2	6:25	top
980	root	0	0	63968	62M	4740	S	0	0.1	24.9	2:14	X
1	root	0	0	120	120	48	S	0	0.0	0.0	0:06	init
2	root	0	0	0	0	0	SW	0	0.0	0.0	0:06	kflushd
3	root	0	0	0	0	0	SW	0	0.0	0.0	0:02	kupdate
4	root	0	0	0	0	0	SW	0	0.0	0.0	0:00	kpiod
5	root	0	0	0	0	0	SW	0	0.0	0.0	0:01	kswapd
252	root	0	0	128	0	0	SW	0	0.0	0.0	0:00	pump
302	bin	0	0	88	88	0	S	0	0.0	0.0	0:00	portmap
316	root	0	0	0	0	0	SW	0	0.0	0.0	0:00	lockd
317	root	0	0	0	0	0	SW	0	0.0	0.0	0:00	rpciod
320	root	0	0	124	124	0	S	0	0.0	0.0	0:00	rpc.statd
79	root	0	0	280	280	116	S	0	0.0	0.1	0:00	syslogd



Examples of sharing

- Application *executables* (the *text* segment) are shared in Windows and Unix
- Windows DLL (dynamic link library) are shared
- Unix *.so* libraries (like */lib/libc.so.6*) are *shared object*



Requirements for sharing

- Early systems used code modification (for example, to process an array in a loop, the address fields of the load/store operations had to be modified on each iteration)
- Instruction modification is not good practice
 - *The program logic becomes obscure and difficult to follow*
 - *It also makes it impossible to share code*
 - *Instruction caches may be optimized for read-only access (for example, no modification to code in the instruction cache)*



Static linking and sharing

- Sharing is the linking of the *same* physical copy of a module into two or more name spaces
- Three schemes
 - *No segmentation or paging*
 - *Pure paging*
 - *Segmentation (possibly with paging)*



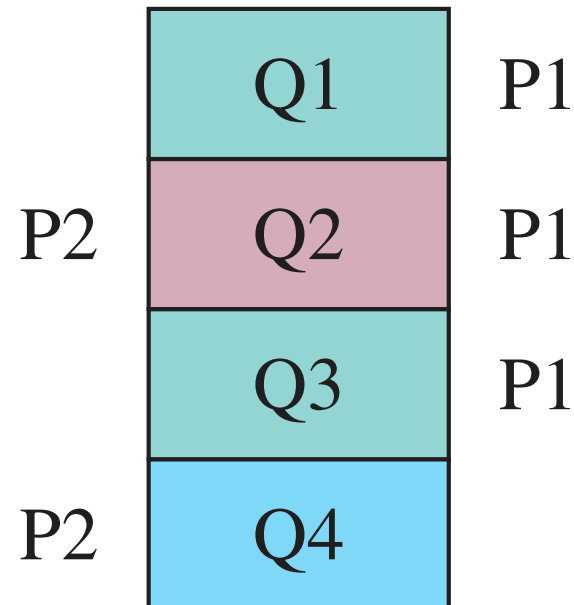
Sharing without segmentation or paging

- The linker/loader searches memory for sharable modules
- A process's address space is usually not contiguous
- Usual problem with fragmentation

Example

process P1 needs
modules Q1, Q2, Q3

process P2 needs
modules Q2, Q4

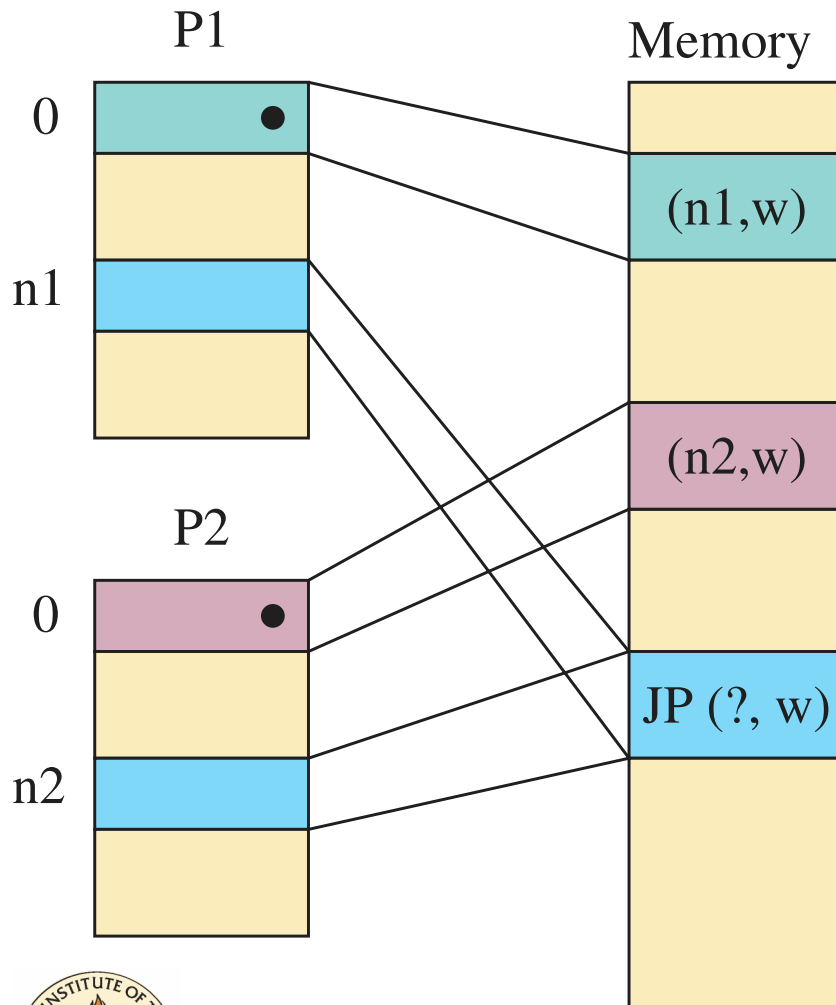


Sharing with paging

- A page can be shared by pointing to the same frame from multiple page tables
- If pages contain only *data* (*no pointers*), then page number assignment can be arbitrary



Sharing with paging



Shared pages n1 and n2 refer to the same page

Page 0 in both processes contains a pointer into the shared page: their values have to be adjusted to point to the right place



Sharing with paging

- If a shared page contains a pointer, like the JP (?, w) instruction, the page must be assigned the *same* page number in all tables
- Leads to page number conflicts
- It is possible to permanently reserve the page numbers for all shared pages, but this is clumsy



Sharing with segmentation

- Logically more elegant, since segments refer to logical components
- A segment is shared by adding that segment in multiple segment tables
- If the segments are paged, the entry points to *shared* page tables

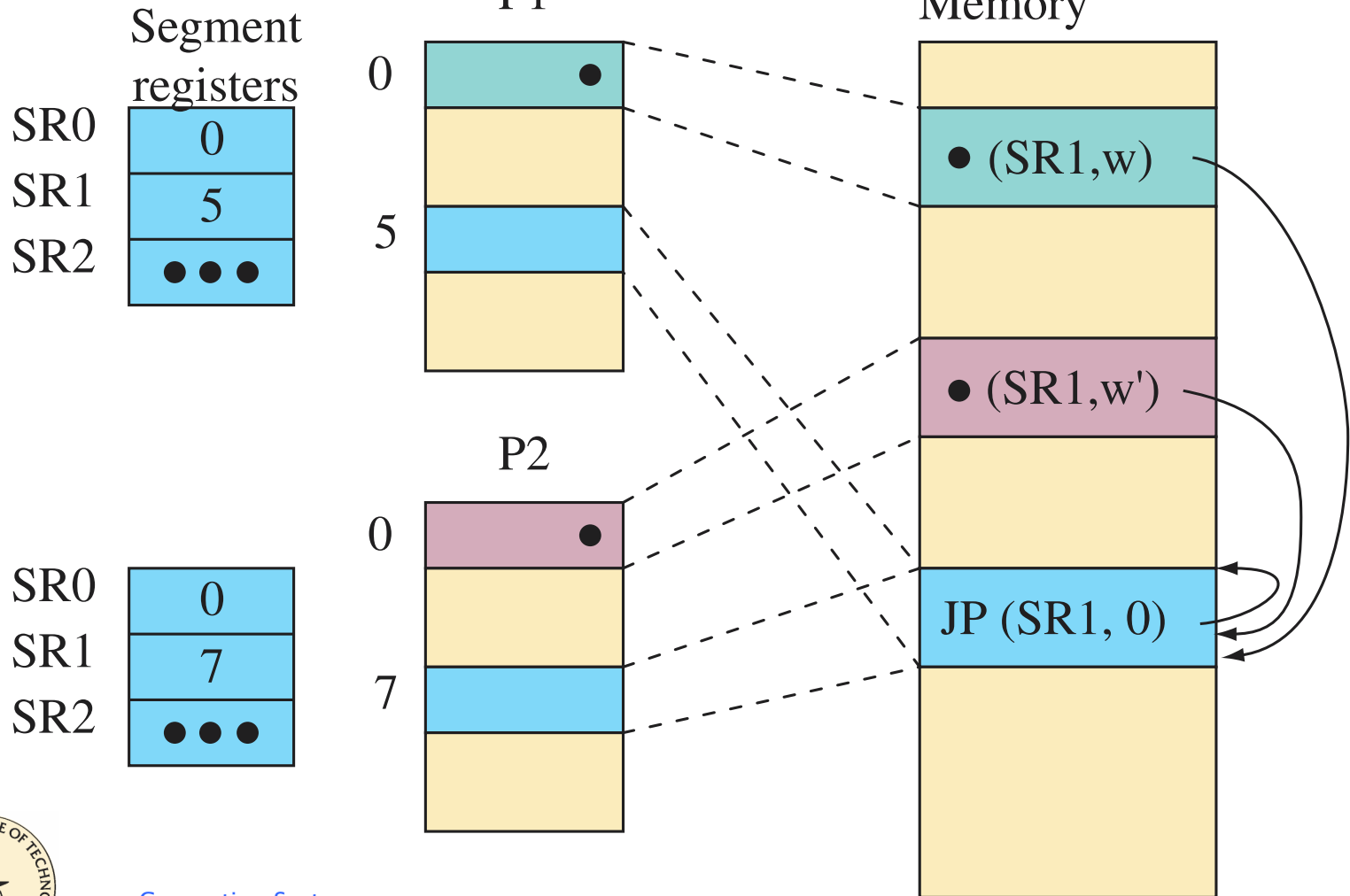


Shared segments that contain pointers

- Method 1: require shared segments to have the *same* segment numbers
- Method 2: add (yet another) level of indirection
 - *Segments are usually accessed through a set of segment registers (for example, x86 has 6 segment registers)*
 - *The segment registers are loaded from a segment table*
 - *All instructions refer to shared segments using the registers*



Shared segments



Dynamic shared segments

- MULTICS (Bash, Benjafield, Gandy 1967)
- No segment numbers appear explicitly in segments; two processes will in general refer to shared segments using different segment numbers
- *Internal references* are resolved using a segment register
- *External references* are represented symbolically, and resolved on first access



Shared segments

- A shared segment contains a *linkage section LS*. Transfer of control, and external pointers are resolved through the linkage section. Each process has a *private* copy of the linkage section.
- Each process has a private *stack* segment
- Three segment registers
 - *lp: linkage pointer*
 - *sp: stack pointer*
 - *pbr: segment number of the current procedure*



A code segment

- Each code segment contains:
 - A symbol table
 - *The pure (read-only) code*
 - A linkage section
- The symbol table contains:
 - *A set of entry points into the segment*
 - *A set of external references that need to be resolved*



The symbol table

- The entry point table contains entries (E_i, e_i, l_i)
 - E_i is the name of the entry point
 - e_i is the offset into the code segment
 - l_i is an index into the linkage section for a linkage procedure
- The external entries refer to
 - data segments (D, X)
 - and code segments (Q, E)

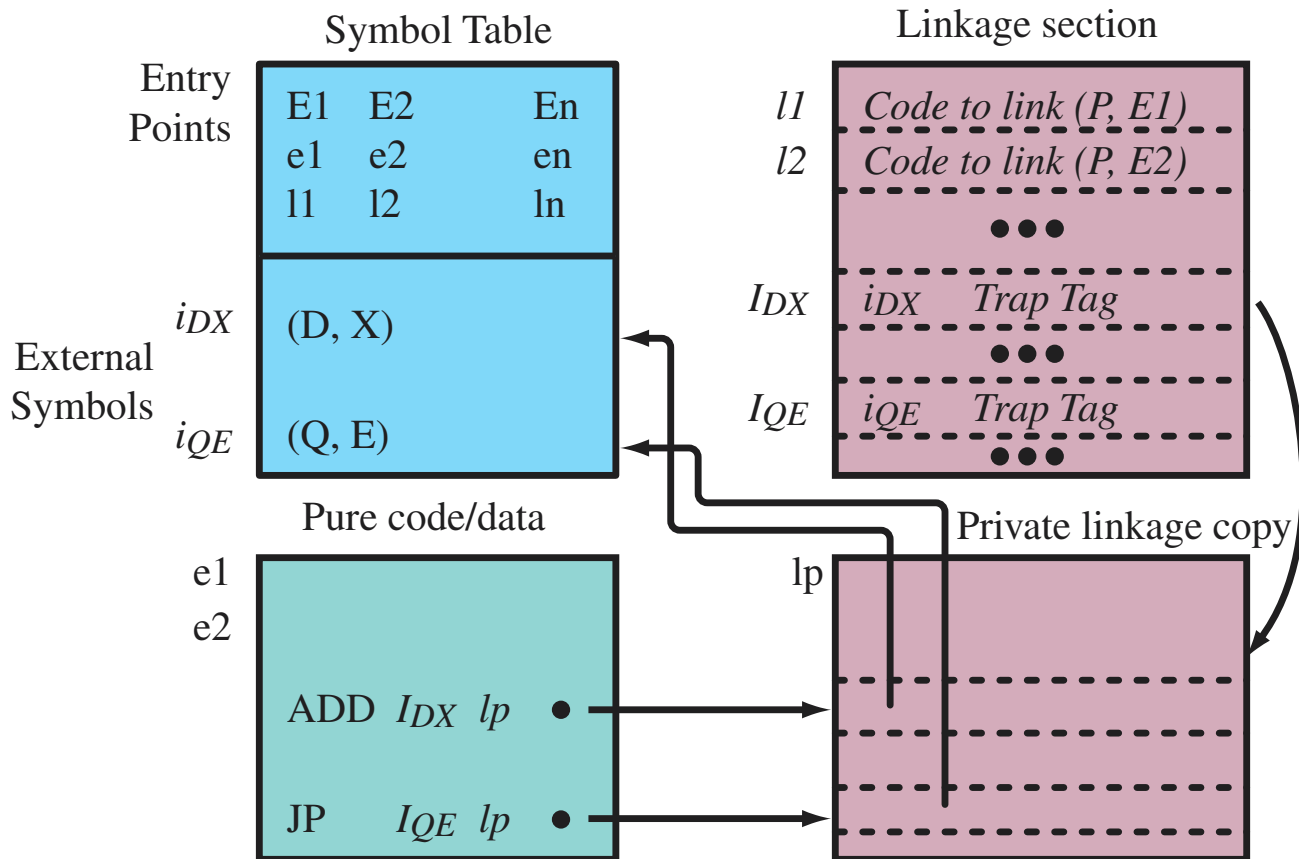


The code segment

- The compiler represents all external references with offsets into the linkage section
- Data references: the instruction $ADD\ I_{DX}\ I_p$ refers to entry I_{DX} in the linkage section, which refers to the i_{DX} entry in the symbol table, which refers to the data segment (D, X)
- Code references: the instruction $JP\ I_{QE}\ I_p$ refers to entry I_{QE} in the linkage section, which is initialized with a *trap tag*
- When a process p links to a code segment P , it is given a *private* copy of the linkage section



A procedure segment

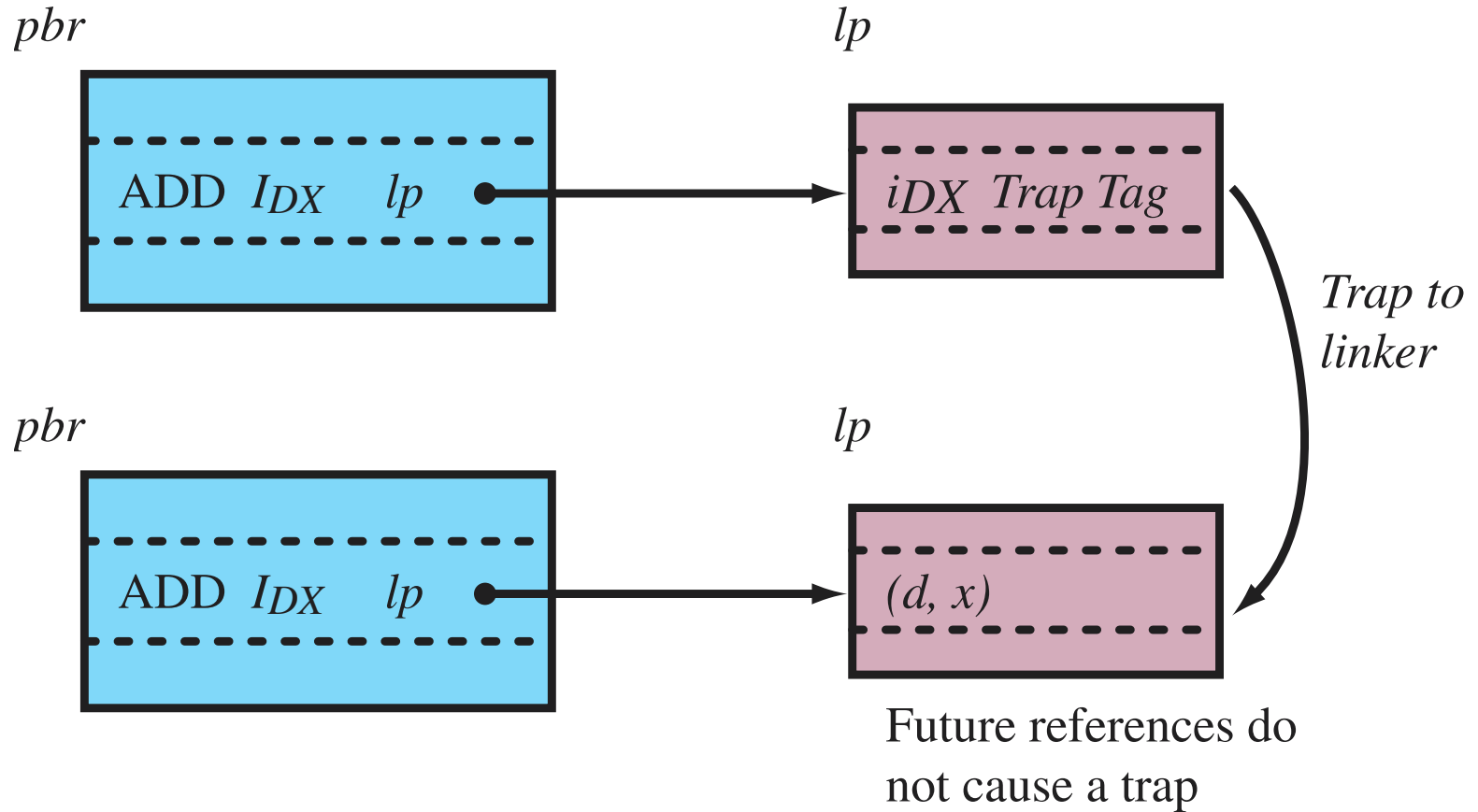


Data linkage

- Example: assume the *ADD* is just about to be executed for the first time
- Registers: *pbr* points to *P*, *lp* points to the copy of the linkage section
- The operand I_{DX} produces the value i_{DX} with the trap tag set
- A trap passes control to the linker, which assigns a segment number d and an offset x and stores the new value in the linkage table (for future reference)



Data linkage



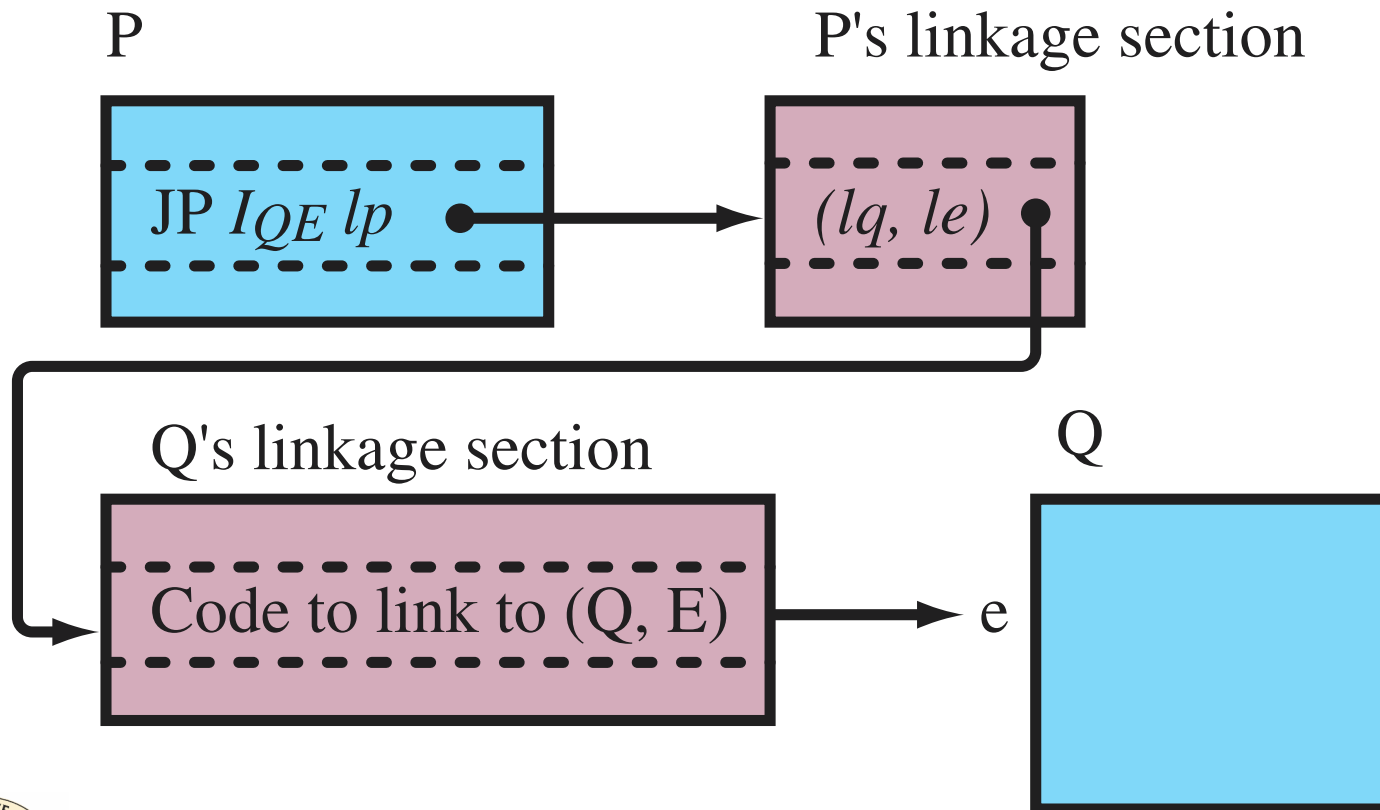
Code linkage

- Example: assume the JP is to be executed for the first time
- The operand I_{QE} produces the value i_{QE} with the trap tag set
- The trap passes control to the linker, which resolves the reference to (q, e)
- Control is transferred to a *linkage entry point stub* in segment Q , which initializes the pbr and lp registers, then transfers control to (q, e)



Code linkage

- After a successful link



Memory management summary

- Relocation is performed at compile, link, load, and execution time
- Virtual memory provides the illusion of a separate machine for each process
 - *Paging*
 - *Segmentation*
 - *Segmentation with paging*

