CS 33

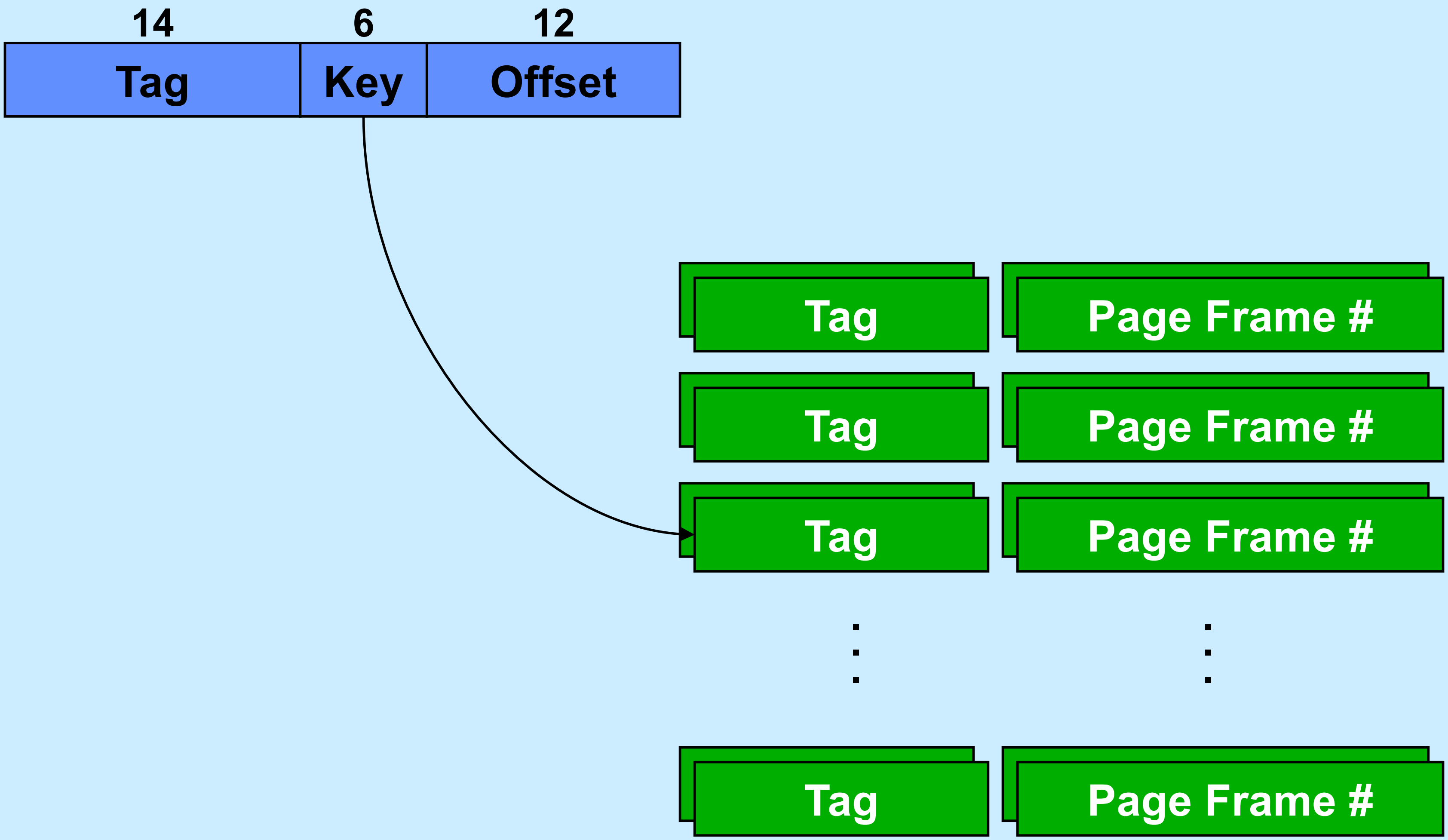
More VM

Libraries

Performance

- **Page table resides in real memory (DRAM)**
- **A 32-bit virtual-to-real translation requires two accesses to page tables, plus the access to the ultimate real address**
  - three real accesses for each virtual access
  - 3X slowdown!
- **A 64-bit virtual-to-real translation requires four accesses to page tables, plus the access to the ultimate real address**
  - 5X slowdown!

Translation Lookaside Buffers



Quiz 1

Recall that there is a 5x slowdown on memory references via virtual memory on the x86-64. If all references are translated via the TLB, the slowdown will be

a) 1x
b) 2x
c) 3x
d) 4x

OS Role in Virtual Memory

- Memory is like a cache
  - quick access if what's wanted is mapped via page table
  - slow if not — OS assistance required
- OS
  - make sure what's needed is mapped in
  - make sure what's no longer needed is not mapped in

Mechanism

- **Program references memory**
  - if reference is mapped, access is quick
    - » even quicker if translation in TLB and referent in on-chip cache
  - if not, page-translation fault occurs and OS is invoked
    - » determines desired page
    - » maps it in, if legal reference

Issues

- **Fetch policy**
  - when are items put in the cache?
- **Placement policy**
  - where do they go in the cache?
- **Replacement policy**
  - what's removed to make room?

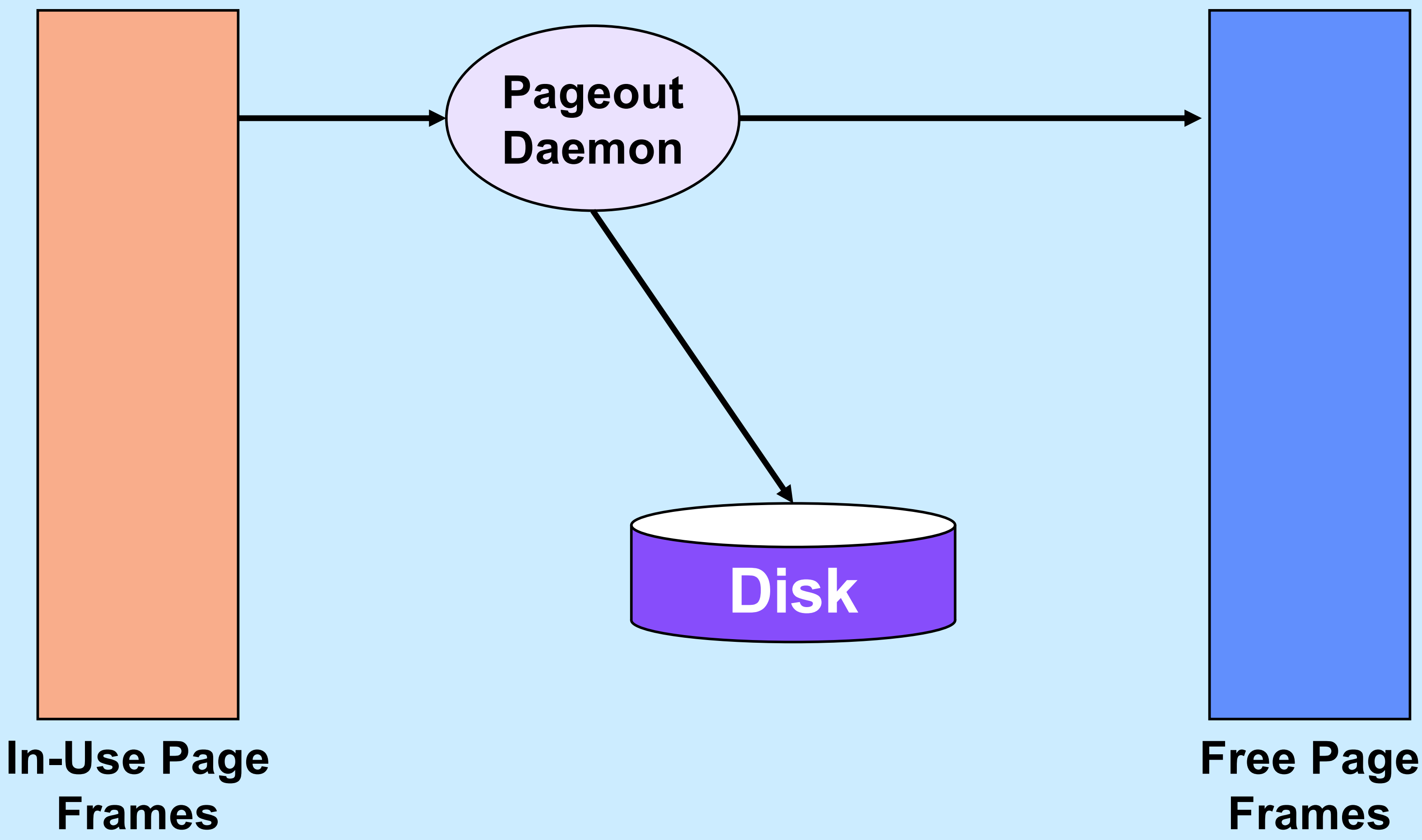
Hardware Caches

- Fetch policy
  - when are items put in the cache?
    - » when they're referenced
    - » prefetch might be possible (e.g., for sequential access)
- Placement policy
  - where do they go in the cache?
    - » usually determined by cache architecture
    - » if there's a choice, it's typically a random choice
- Replacement policy
  - what's removed to make room?
    - » usually determined by cache architecture
    - » if there's a choice, it's typically a random choice

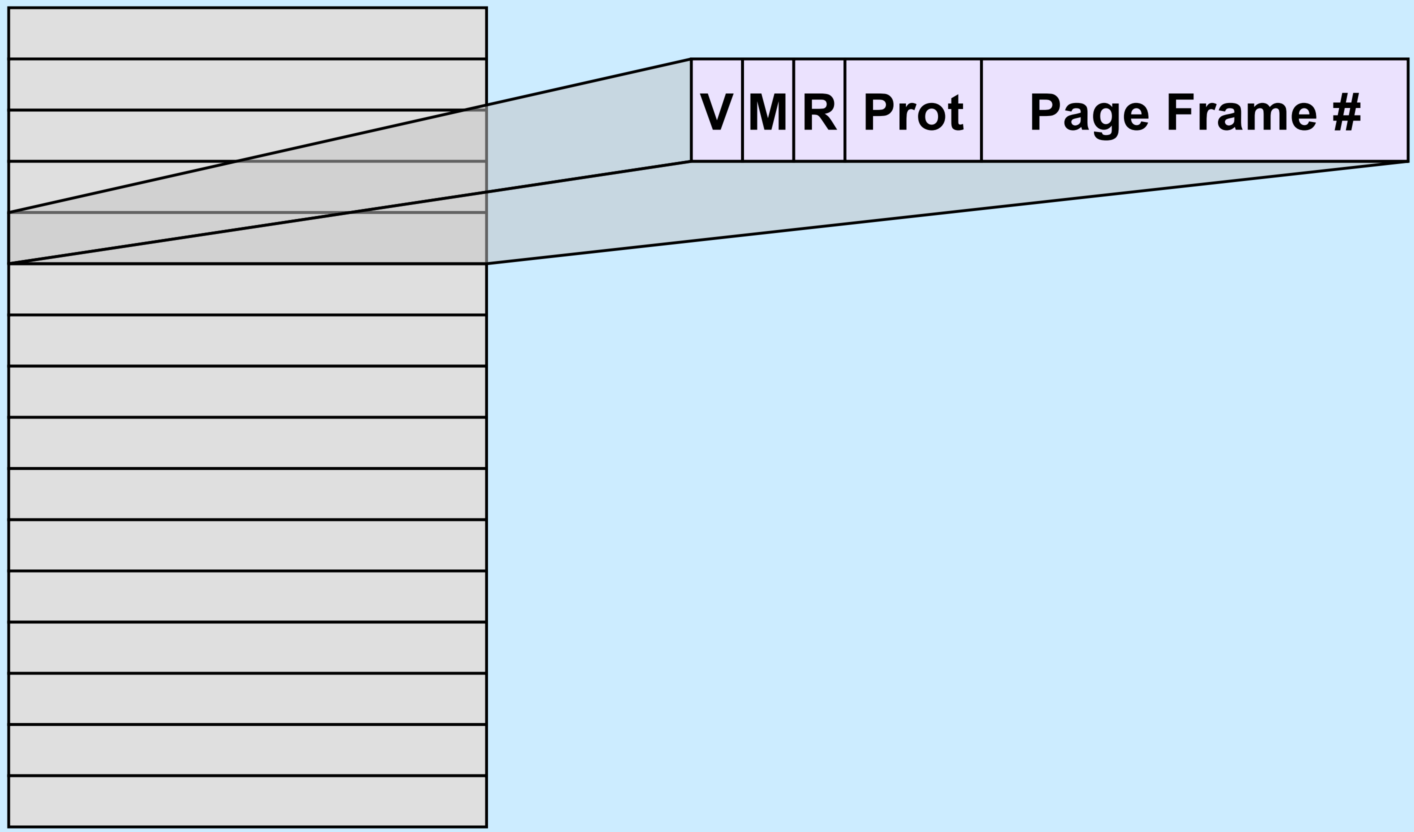
Software Caches

- **Fetch policy**
  - when are items put in the cache?
    - » when they're referenced
    - » prefetch might be easier than for hardware caches
- **Placement policy**
  - where do they go in the cache?
    - » usually doesn't matter (no memory is more equal than others)
- **Replacement policy**
  - what's removed to make room?
    - » would like to remove that whose next use is farthest in future
    - » instead, remove that whose last reference was farthest in the past

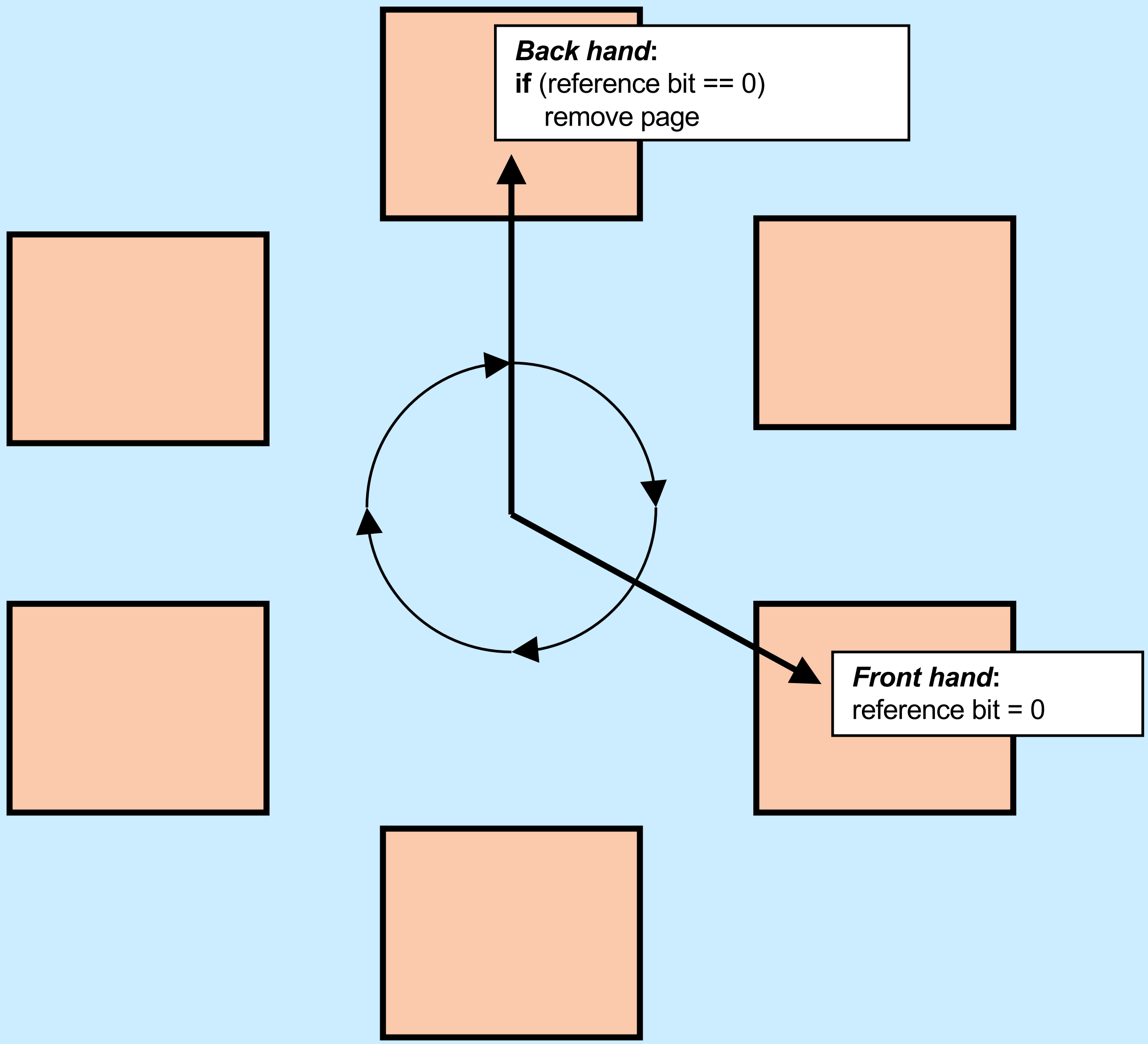
The "Pageout Daemon"



Managing Page Frames

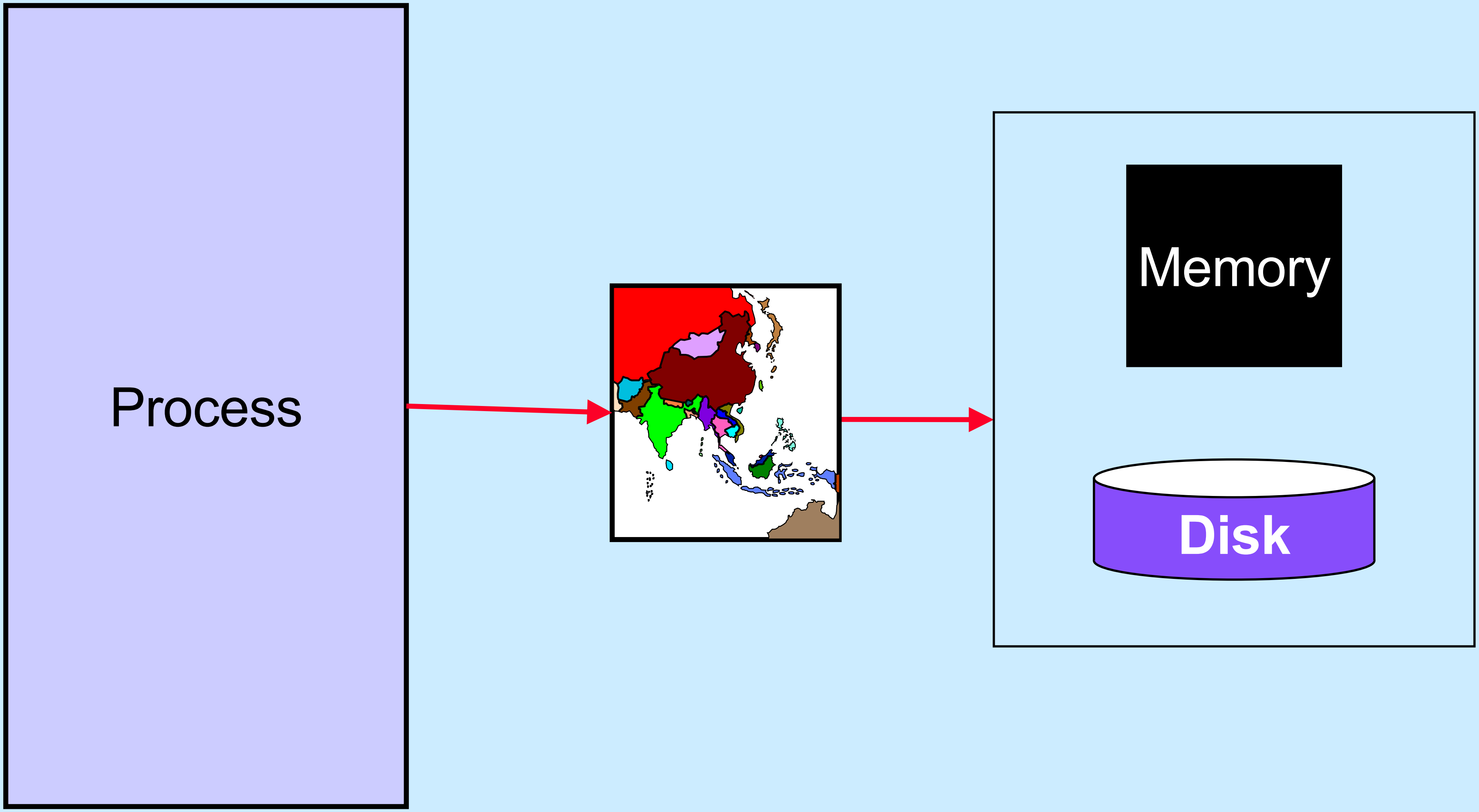


Clock Algorithm

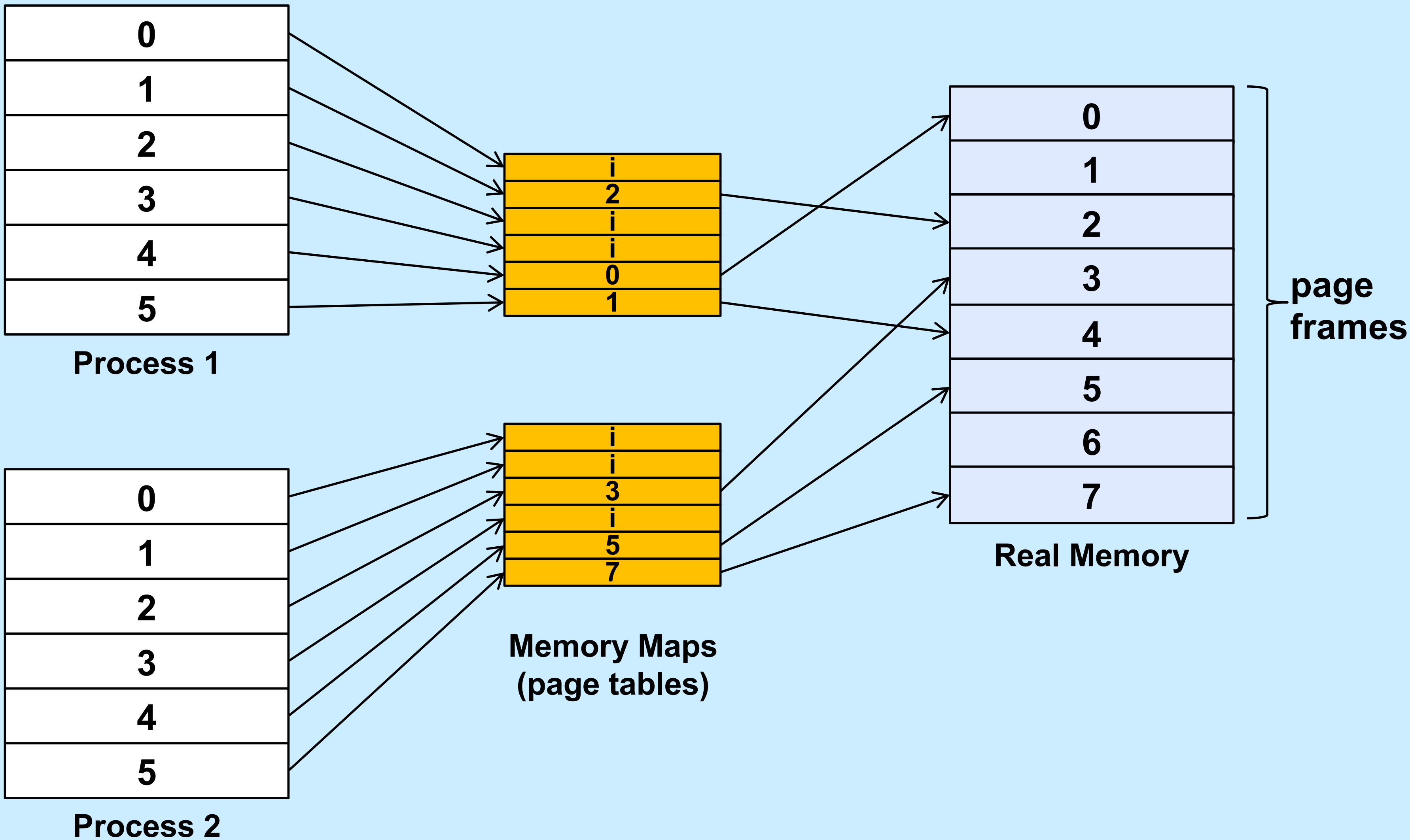


Why is virtual memory used?

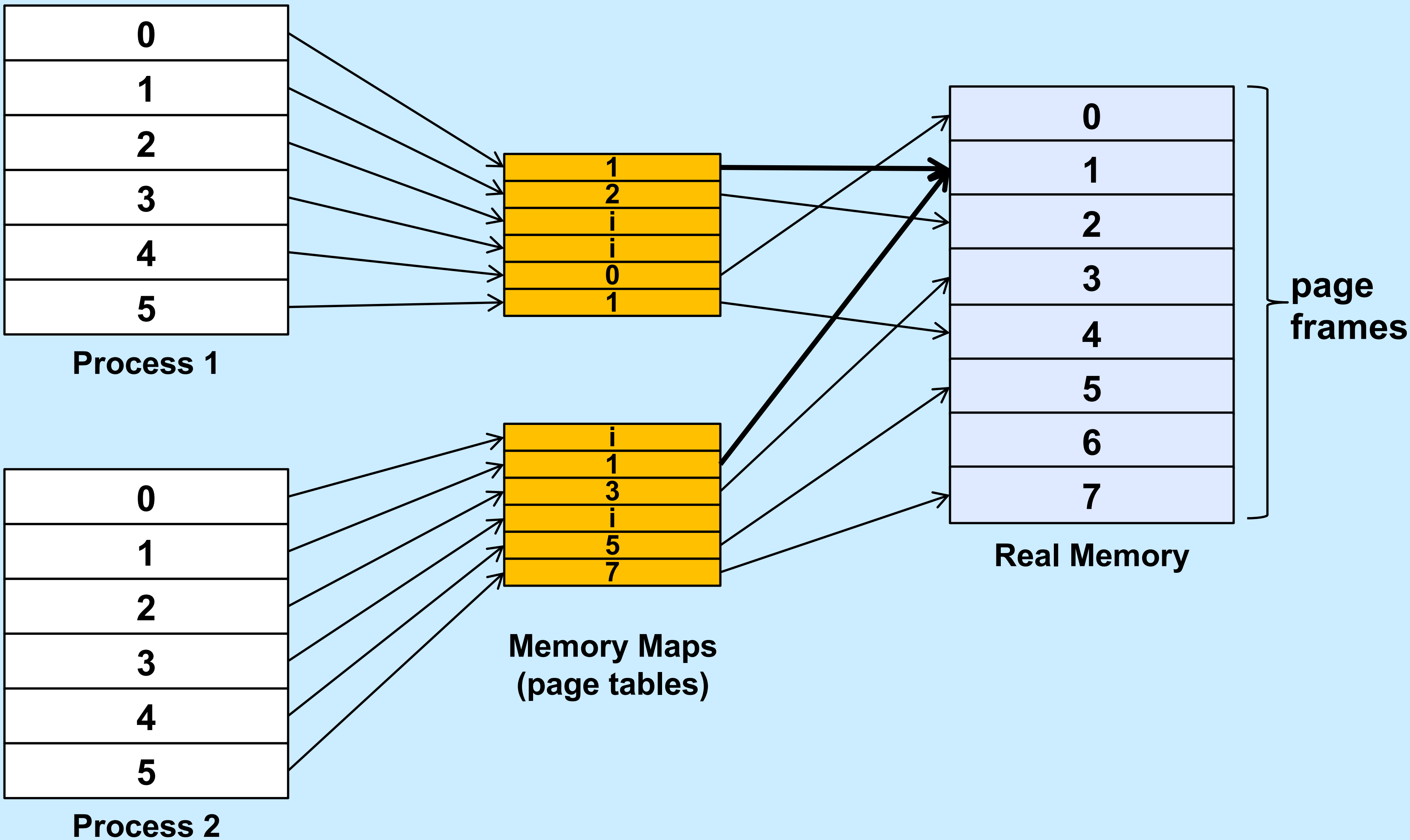
More VM than RM



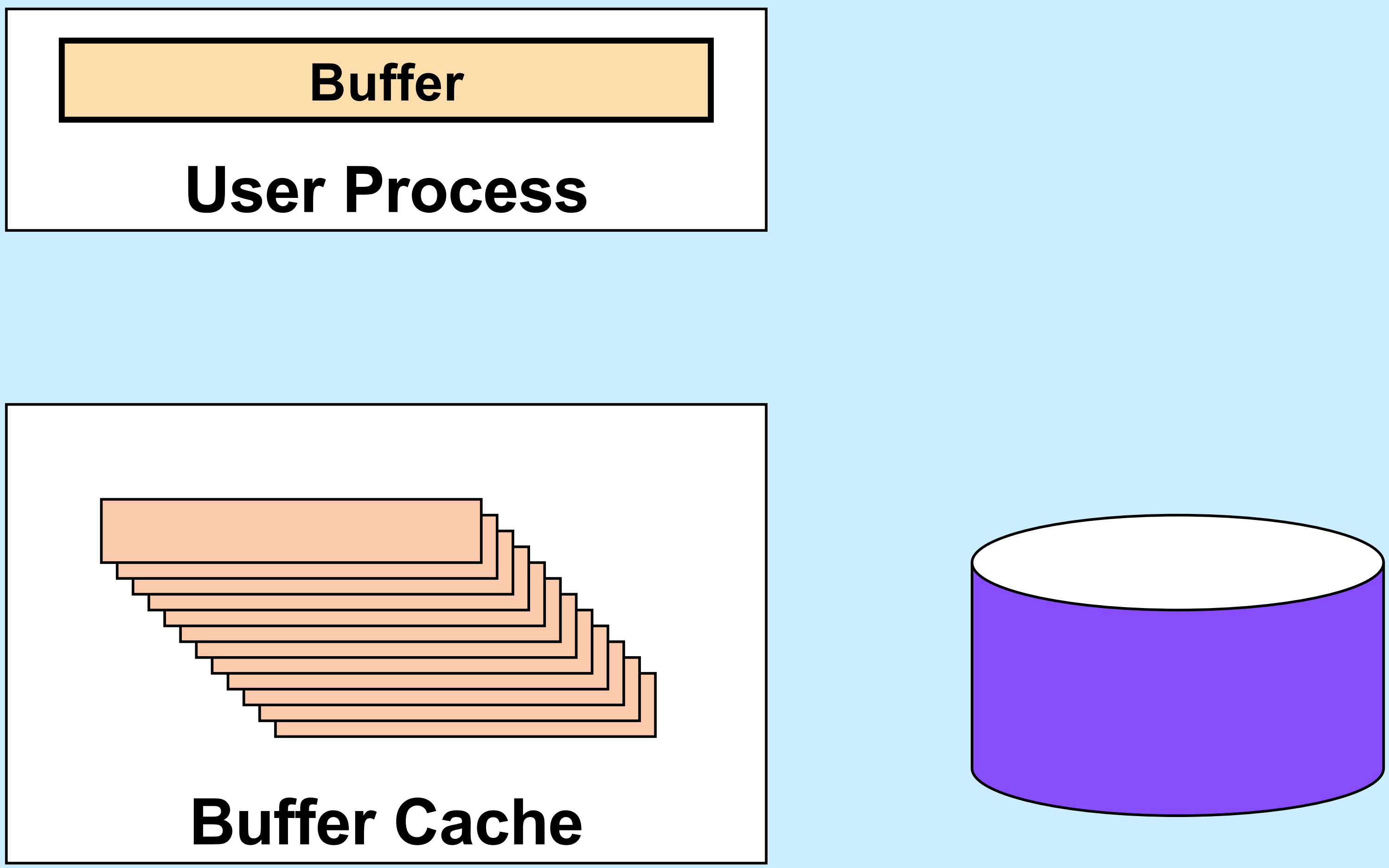
Isolation


0
1
2
3
4
5
Process 1
i
2
i
i
0
1
0
1
2
3
4
5
6
7
page frames
Real Memory
i
i
3
i
5
7
0
1
2
3
4
5
Process 2
Memory Maps
(page tables)

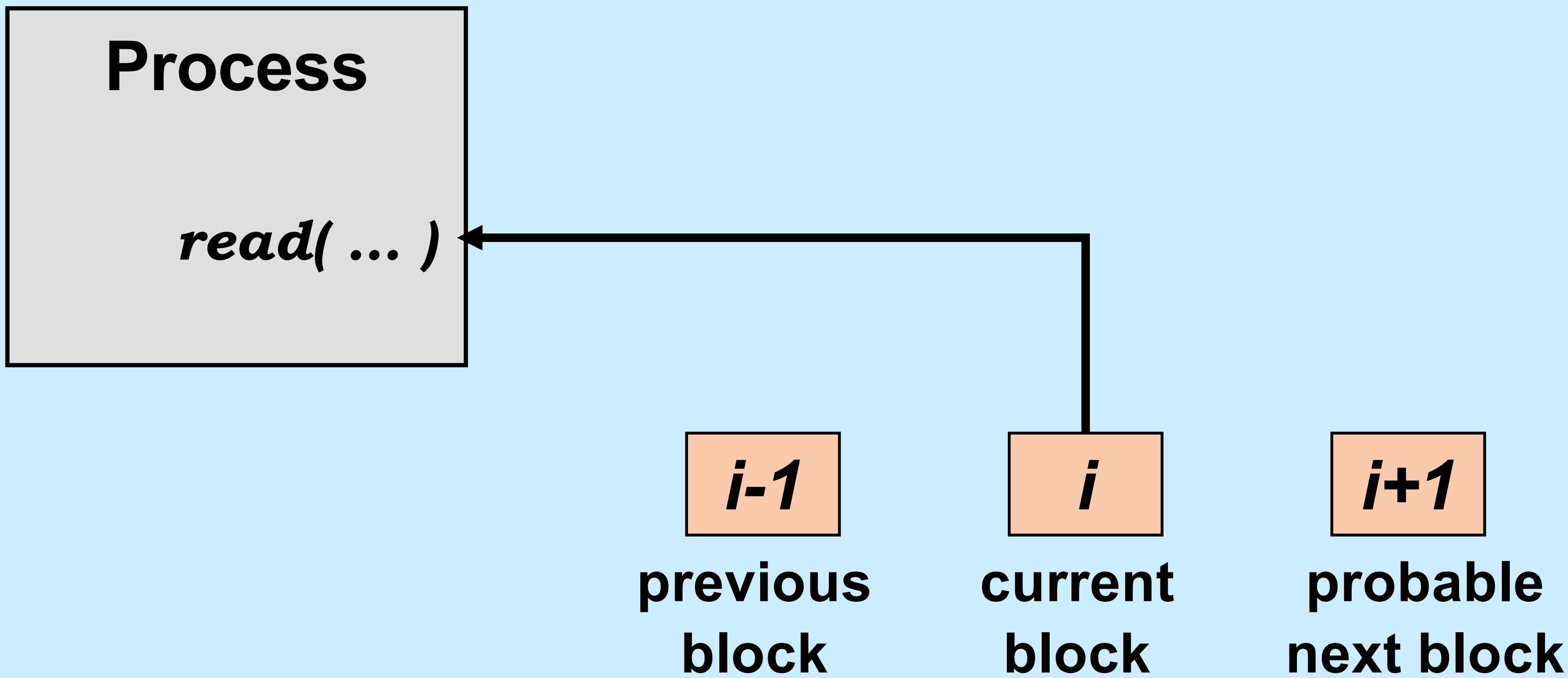
Sharing


0
1
2
3
4
5
Process 1
1
2
i
i
0
1
0
1
2
3
4
5
Process 2
i
1
3
i
5
7
Memory Maps
(page tables)
0
1
2
3
4
5
6
7
Real Memory
page
frames

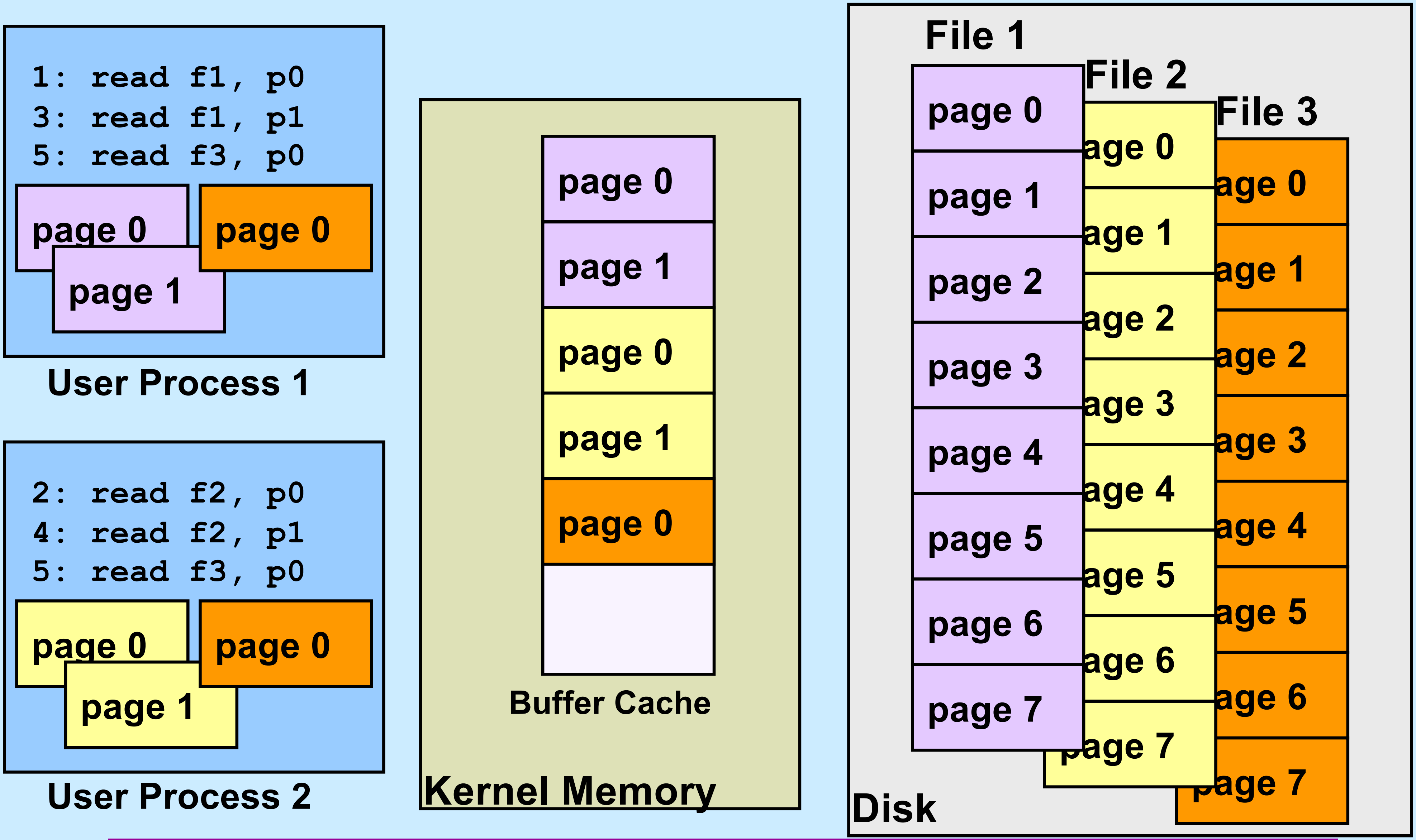
File I/O



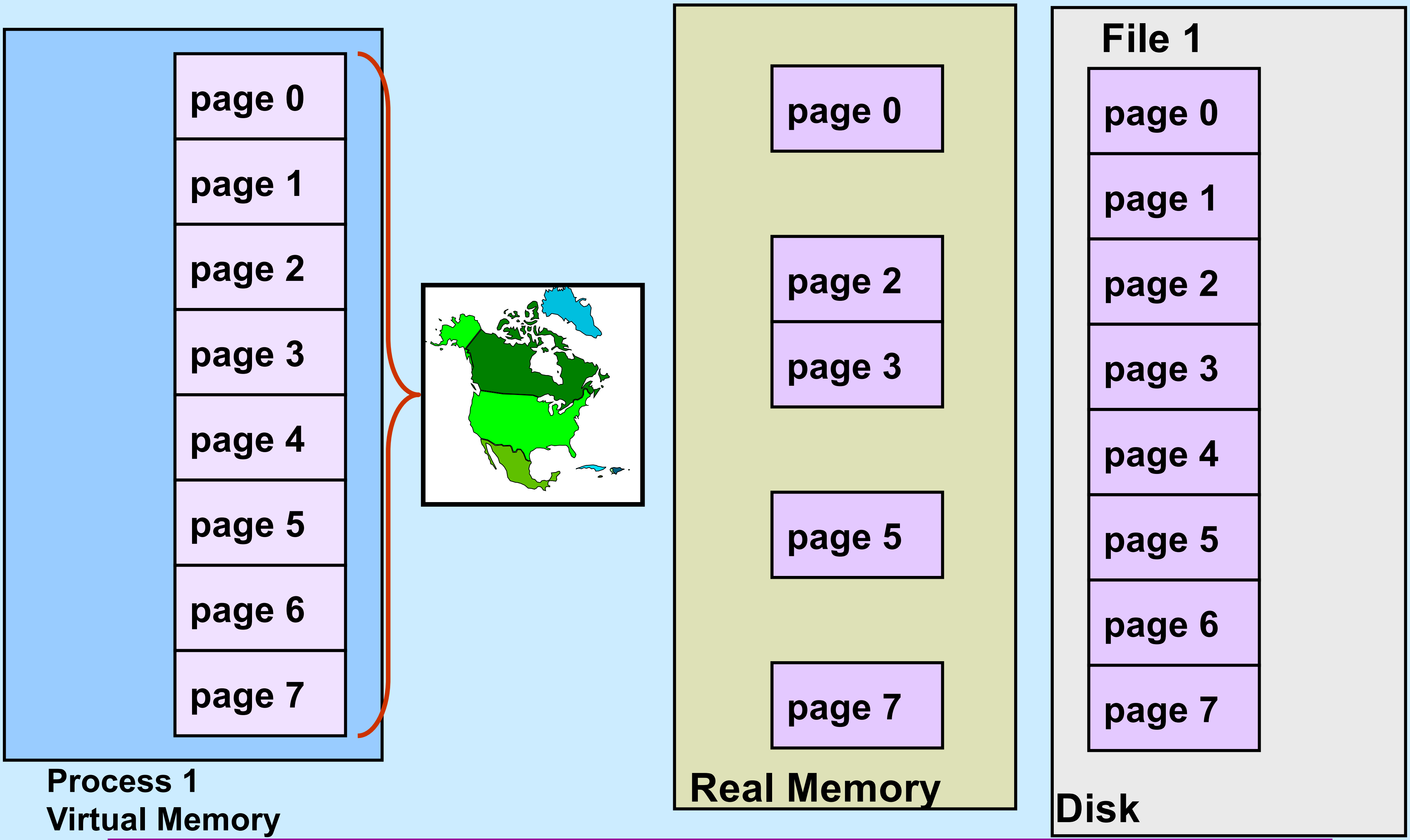
Multi-Buffered I/O



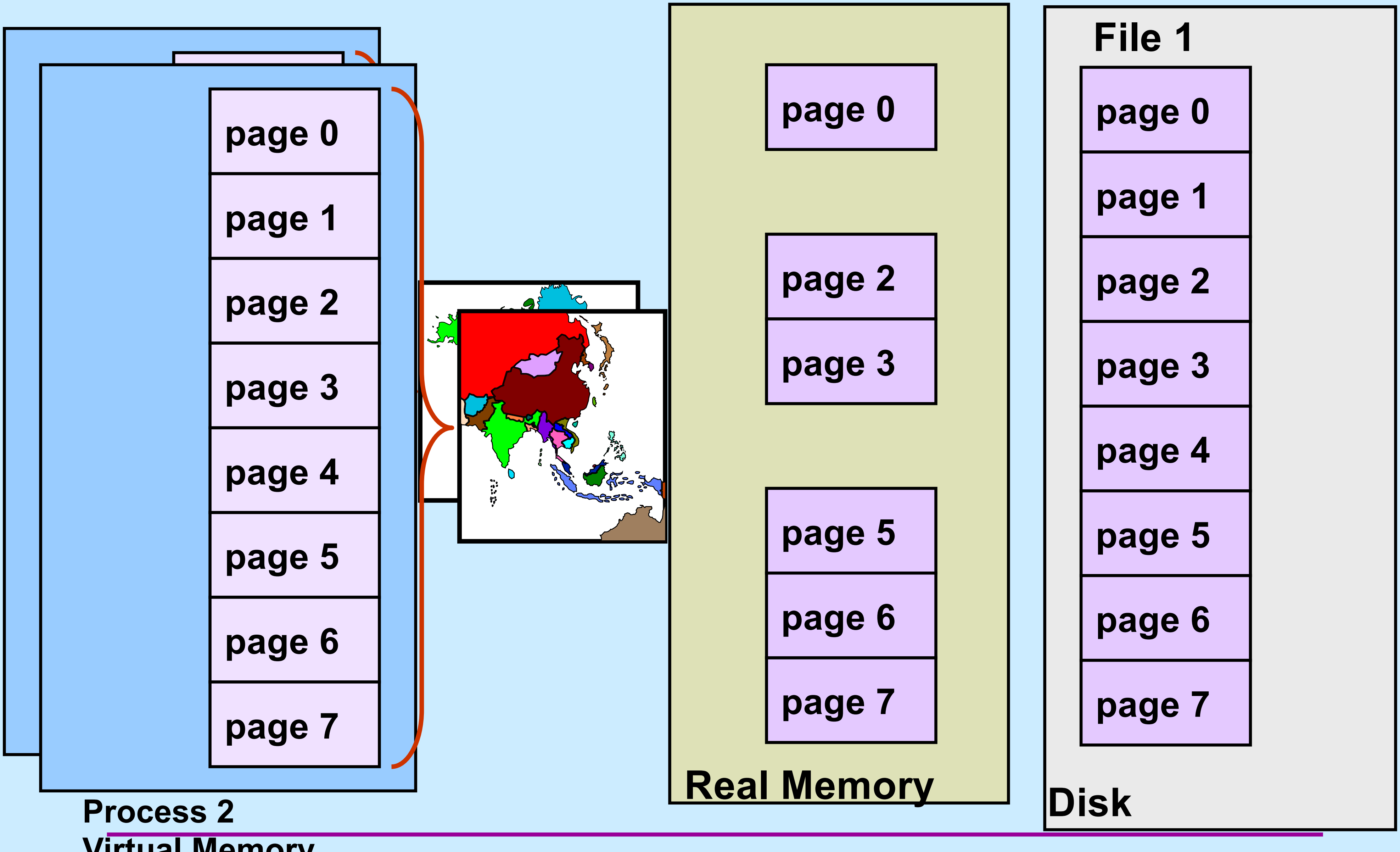
Traditional I/O



Mapped File I/O



Multi-Process Mapped File I/O



Mapped Files

- **Traditional File I/O**

```
char buf[BigEnough];
fd = open(file, O_RDWR);
for (i=0; i<n_recs; i++) {
    read(fd, buf, sizeof(buf));
    use(buf);
}
```

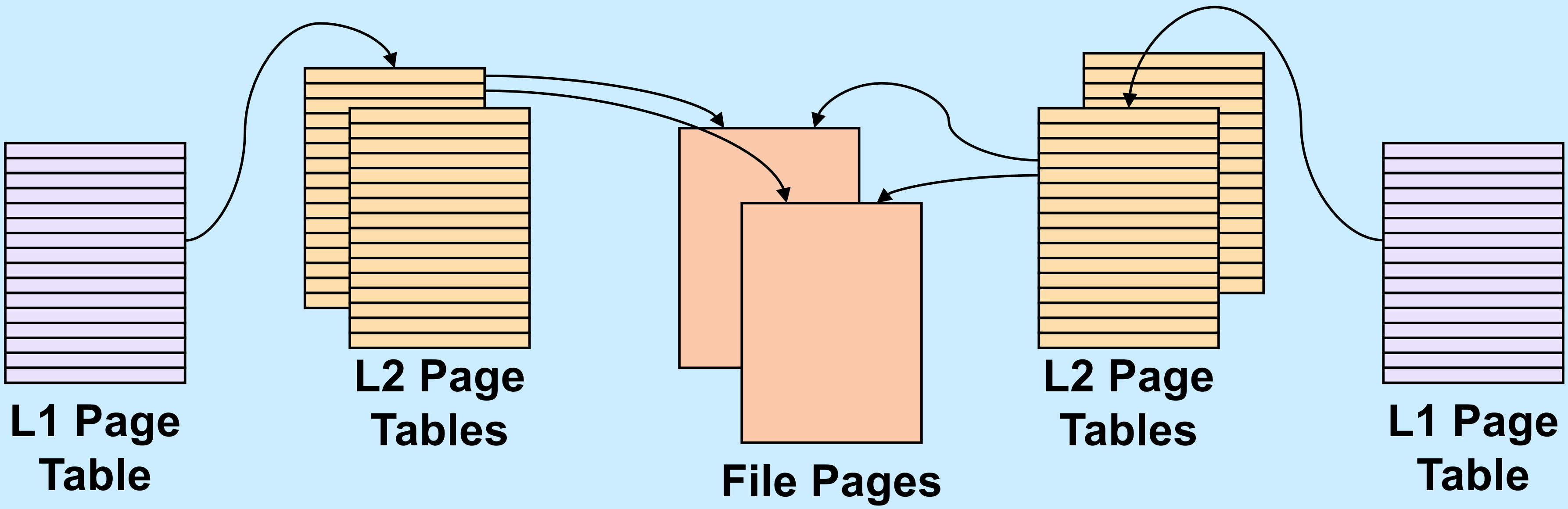
- **Mapped File I/O**

```
record_t *MappedFile;
fd = open(file, O_RDWR);
MappedFile = mmap(... , fd, ...);
for (i=0; i<n_recs; i++)
    use(MappedFile[i]);
```

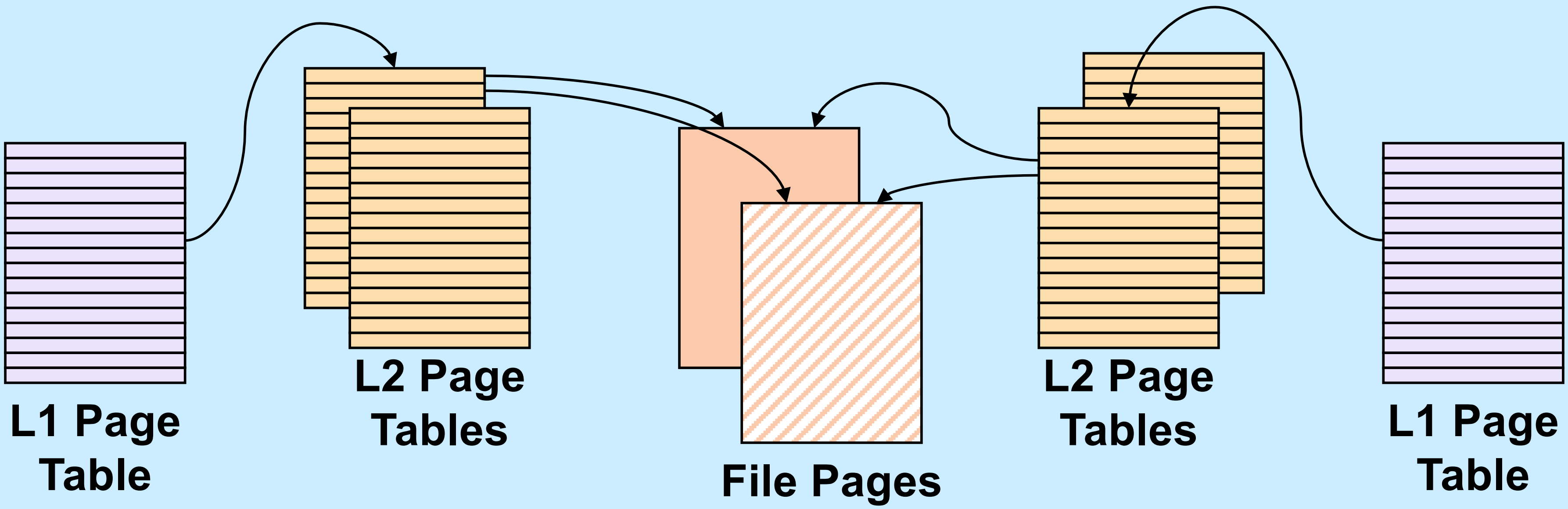

Mmap System Call

```
void *mmap(
  void *addr,
    // where to map file (0 if don't care)
  size_t len,
    // how much to map
  int prot,
    // memory protection (read, write, exec.)
  int flags,
    // shared vs. private, plus more
  int fd,
    // which file
  off_t off
    // starting from where
  );
```


The *mmap* System Call

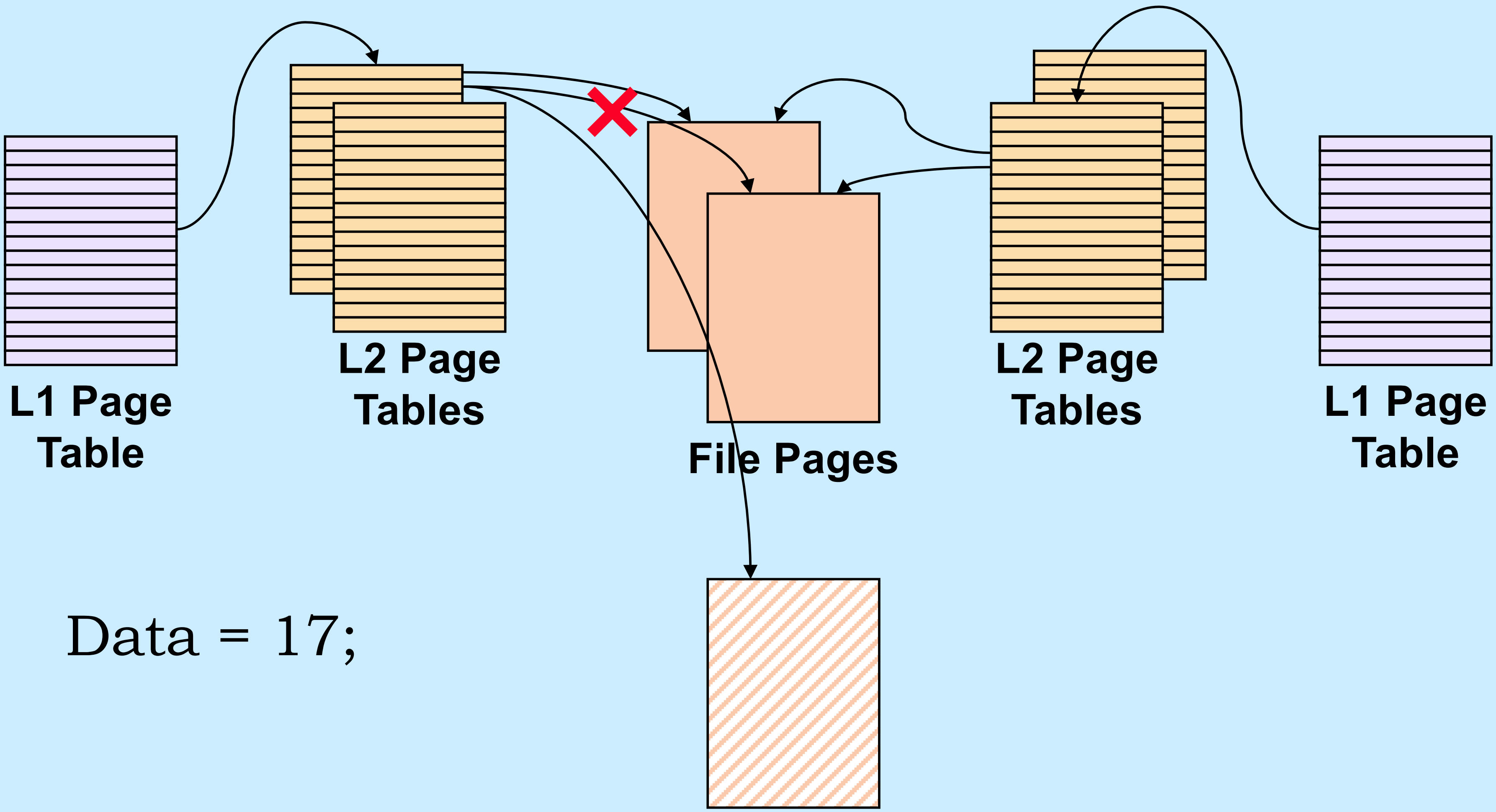


Share-Mapped Files



Data = 17;

Private-Mapped Files



Data = 17;

Example

```
int main( ) {
  int fd;
  dataObject_t *dataObjectp;

  fd = open("file", O_RDWR);
  if ((int)(dataObjectp = (dataObject_t *)mmap(0,
      sizeof(dataObject_t),
      PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == -1) {
    perror("mmap");
    exit(1);
  }

  // dataObjectp points to region of (virtual) memory
  // containing the contents of the file

  ...

}
```

fork and mmap

```
int main() {
  int x=1;

  if (fork() == 0) {
    // in child
    x = 2;
    exit(0);
  }
  // in parent
  while (x==1) {
    // will loop forever
  }
  return 0;
}
```

```
int main() {
  int fd = open( ... );
  int *xp = (int *)mmap(...,
      MAP_SHARED, fd, ...);
  xp[0] = 1;
  if (fork() == 0) {
    // in child
    xp[0] = 2;
    exit(0);
  }
  // in parent
  while (xp[0]==1) {
    // will terminate
  }
  return 0;
}
```

Libraries

- Collections of useful stuff
- Allow you to:
  - incorporate items into your program
  - substitute new stuff for existing items
- Often ugly …

Creating a Library

```
$ gcc -c sub1.c sub2.c sub3.c
$ ls
sub1.c       sub2.c       sub3.c
sub1.o       sub2.o       sub3.o
$ ar cr libpriv1.a sub1.o sub2.o sub3.o
$ ar t libpriv1.a
sub1.o
sub2.o
sub3.o
$
```


Using a Library

```
$ cat prog.c
int main() {
  sub1();
  sub2();
  sub3();
}
$ cat sub1.c
void sub1() {
  puts("sub1");
}
```

```
$ gcc -o prog prog.c -L. -lpriv1
$ ./prog
sub1
sub2
sub3
```

Where does *puts* come from?

```
$ gcc -o prog prog.c -L. \
  -lpriv1 \
  -L/lib/x86_64-linux-gnu -lc
```

Static-Linking: What's in the Executable

- ld puts in the executable:
      - » (assuming all .c files have been compiled into .o files)
  - – all .o files from argument list (including those newly compiled)
  - – .o files from archives as needed to satisfy unresolved references
      - » some may have their own unresolved references that may need to be resolved from additional .o files from archives
      - » each archive processed just once (as ordered in argument list)
          - order matters!

Example

```
$ cat prog2.c
int main() {
  void func1();
  func1();
  return 0;
}
$ cat func1.c
void func1() {
  void func2();
  func2();
}
$ cat func2.c
void func2() {
}
```


Order Matters ...

```
$ ar t libf1.a
func1.o
$ ar t libf2.a
func2.o
$ gcc -o prog2 prog2.c -L. -lf1 -lf2
$
$ gcc -o prog2 prog2.c -L. -lf2 -lf1
./libf1.a(sub1.o): In function `func1':
func1.c:(.text+0xa): undefined reference to `func2'
collect2: error: ld returned 1 exit status
```

Substitution

```
$ cat myputs.c
int puts(char *s) {
  write(1, "My puts: ", 9);
  write(1, s, strlen(s));
  write(1, "\n", 1);
  return 1;
}
$ gcc -c myputs.c
$ ar cr libmyputs.a myputs.o
$ gcc -o prog prog.c -L. -lpriv1 -lmyputs
$ ./prog
My puts: sub1
My puts: sub2
My puts: sub3
```


An Urgent Problem

- printf is found to have a bug
  - perhaps a security problem
- All existing instances must be replaced
  - there are zillions of instances ...
- Do we have to re-link all programs that use printf?

Dynamic Linking

- Executable is not fully linked
  - contains list of needed libraries
- Linkages set up when executable is run

Benefits

- **Without dynamic linking**
  - every executable contains copy of printf (and other stuff)
    - » waste of disk space
    - » waste of primary memory
- **With dynamic linking**
  - just one copy of printf
    - » shared by all

Shared Objects: Unix's Dynamic Linking

1 **Compile program**

2 **Track down references with *ld***

- ***archives*** **(containing *relocatable objects*) in ".a" files are statically linked**
- ***shared objects*** **in ".so" files are dynamically linked**
  - » **names of needed .so files included with executable**

3 **Run program**

- ***ld-linux.so*** **is invoked first to complete the linking and relocation steps, if necessary**

Creating a Shared Library

```
$ gcc -fPIC -c myputs.c
$ ld -shared -o libmyputs.so myputs.o
$ gcc -o prog prog.c –fPIC -L. –lpriv1 –lmyputs -Wl,-rpath \
  /home/twd/libs
$ ldd prog
linux-vdso.so.1 =>  (0x00007fff235ff000)
libmyputs.so => /home/twd/libs/libmyputs.so (0x00007f821370f000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f821314e000)
/lib64/ld-linux-x86-64.so.2 (0x00007f8213912000)
$ ./prog
My puts: sub1
My puts: sub2
My puts: sub3
```


Order Still Matters

- **All shared objects listed in the executable are loaded into the address space**
  - whether needed or not
- **ld-linux.so will find anything that's there**
  - looks in the order in which shared objects are listed

A Problem

- You've put together a library of useful functions
  - libgoodstuff.so
- Lots of people are using it
- It occurs to you that you can make it even better by adding an extra argument to a few of the functions
  - doing so will break all programs that currently use these functions
- You need a means so that old code will continue to use the old version, but new code will use the new version

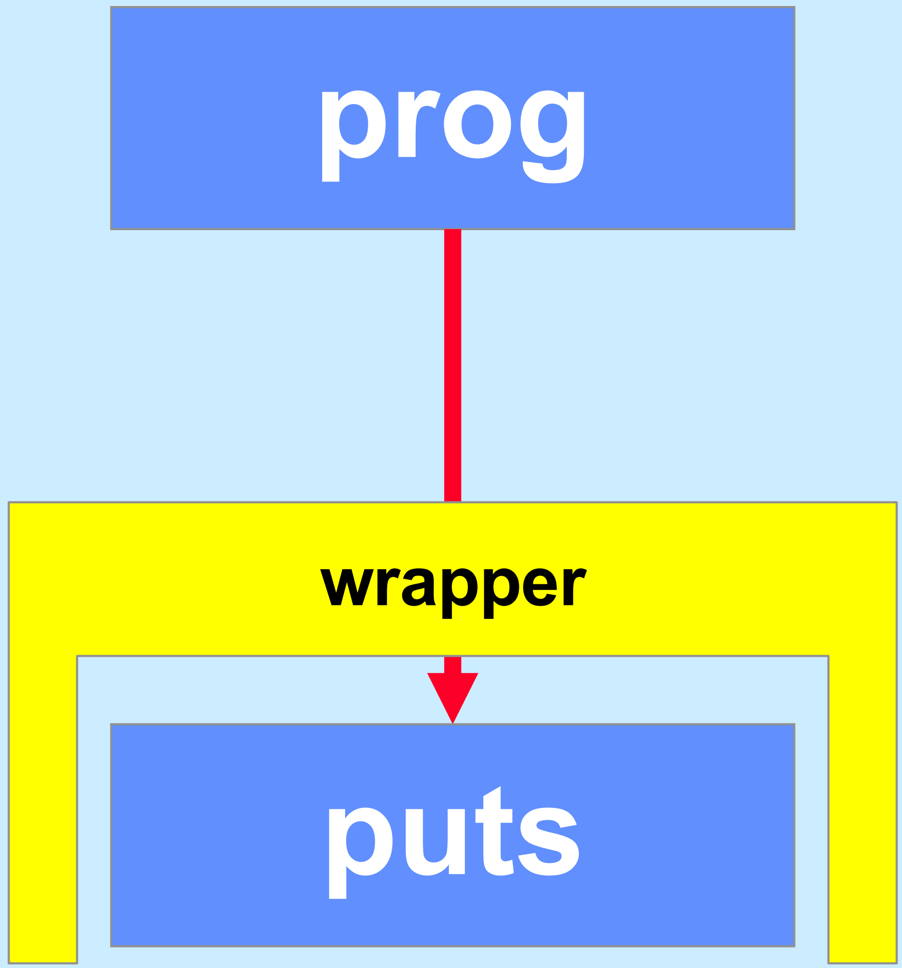
A Solution

- The two versions of your program coexist
  - libgoodstuff.so.1
  - libgoodstuff.so.2
- You arrange so that old code uses the old version, new code uses the new
- Most users of your code don't really want to have to care about version numbers
  - they want always to link with libgoodstuff.so
  - and get the version that was current when they wrote their programs

Versioning

```
$ gcc -fPIC -c goodstuff.c
$ ld -shared -soname libgoodstuff.so.1 \
-o libgoodstuff.so.1 goodstuff.o
$ ln -s libgoodstuff.so.1 libgoodstuff.so
$ gcc -o prog1 prog1.c -L. -lgoodstuff \
-Wl,-rpath .
$ vi goodstuff.c
$ gcc -fPIC -c goodstuff.c
$ ld -shared -soname libgoodstuff.so.2 \
-o libgoodstuff.so.2 goodstuff.o
$ rm -f libgoodstuff.so
$ ln -s libgoodstuff.so.2 libgoodstuff.so
$ gcc -o prog2 prog2.c -L. -lgoodstuff \
-Wl,-rpath .
```

Interpositioning



How To ...

```
int __wrap_puts(const char *s) {
  int __real_puts(const char *);

  write(2, "calling myputs: ", 16);
  return __real_puts(s);
}
```


Compiling/Linking It

```
$ cat tputs.c
int main() {
  puts("This is a boring message.");
  return 0;
}
$ gcc -o tputs -Wl,--wrap=puts tputs.c myputs.c
$ ./tputs
calling myputs: This is a boring message.
$
```

How To (Alternative Approach) …

```
#include <dlfcn.h>

int puts(const char *s) {
  int (*pptr)(const char *);

  pptr = (int(*)())dlsym(RTLD_NEXT, "puts");

  write(2, "calling myputs: ", 16);
  return (*pptr)(s);
}
```


What's Going On …

- **gcc/ld**
  - compiles code
  - does static linking
    - » searches list of libraries
    - » adds references to shared objects
- **runtime**
  - program invokes *ld-linux.so* to finish linking
    - » maps in shared objects
    - » does relocation and procedure linking as required
  - *dlsym* invokes *ld-linux.so* to do more linking
    - » RTLD_NEXT says to use the next (second) occurrence of the symbol

Delayed Wrapping

- **LD_PRELOAD**
  - environment variable checked by *ld-linux.so*
  - specifies additional shared objects to search (first) when program is started

Environment Variables

- **Another form of exec**
  - `int execve(const char *filename,`
    `char *const argv[],`
    `char *const envp[]);`

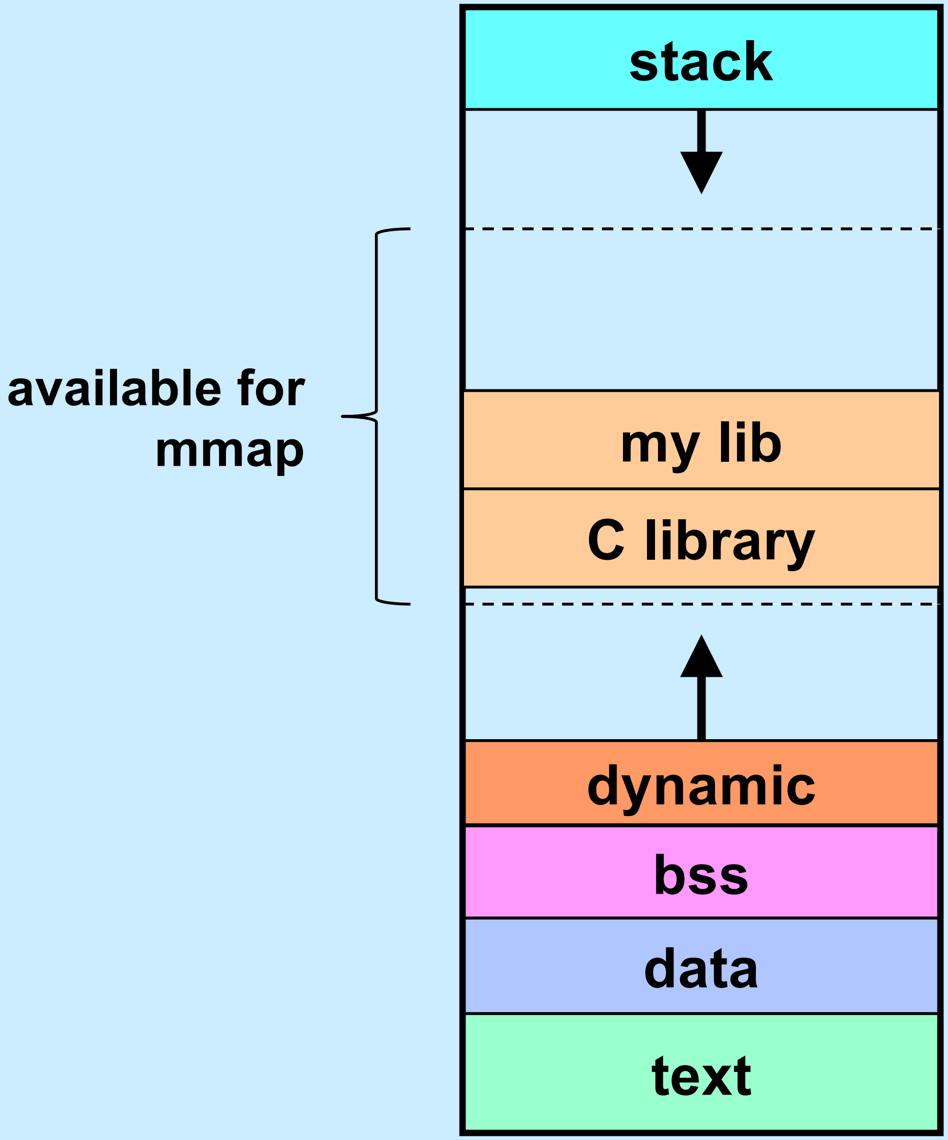
```
int execve(const char *filename,
           char *const argv[],
           char *const envp[]);
```

- **envp is an array of strings, of the form**
  - **key=value**
- **programs can search for values, given a key**
- **example**
  - **PATH=~/bin:/bin:/usr/bin:/course/cs0330/bin**

Example

```
$ gcc -o tputs tputs.c
$ ./tputs
This is a boring message.
$ LD_PRELOAD=./libmyputs.so.1; export LD_PRELOAD
$ ./tputs
calling myputs: This is a boring message.
$
```

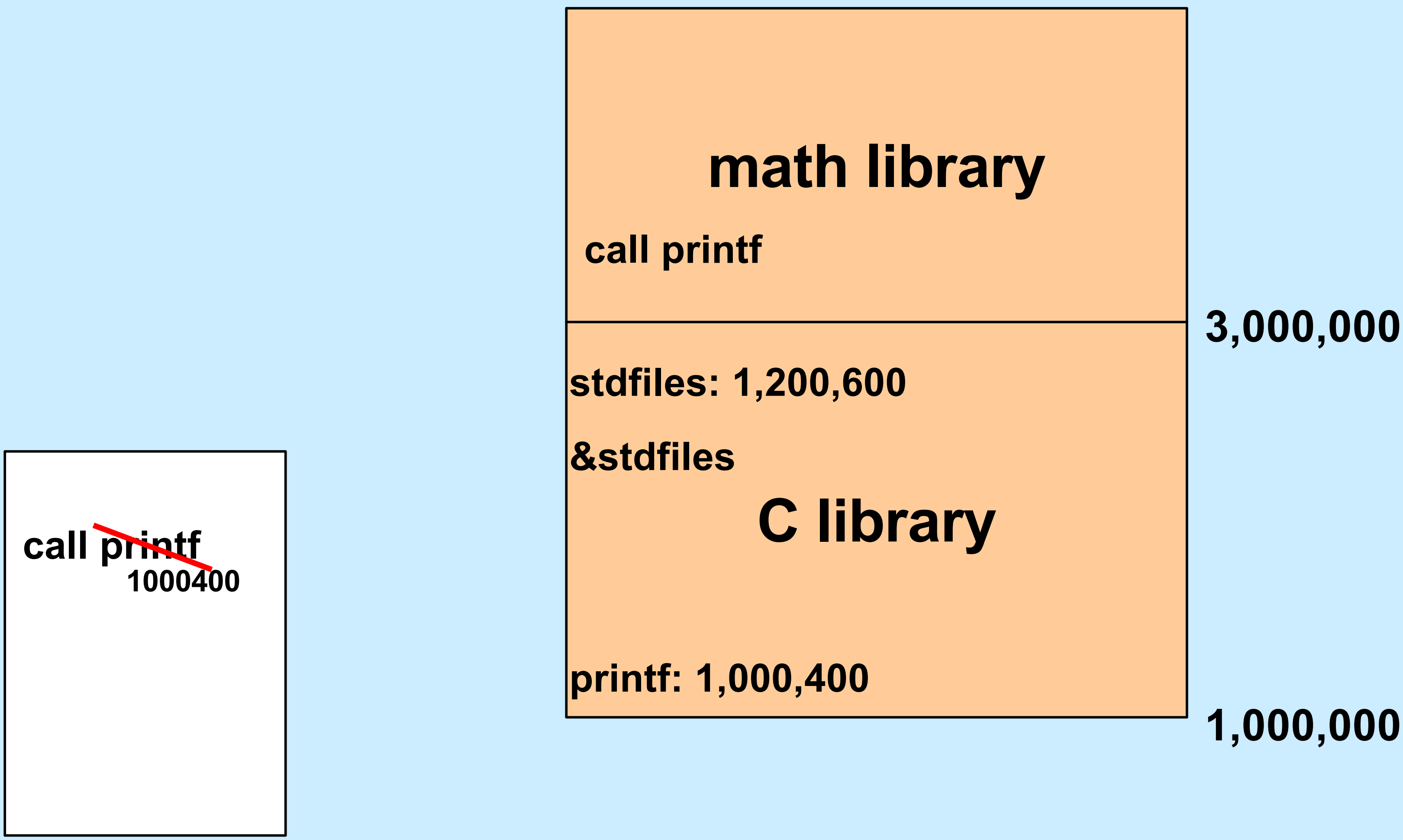

Mmapping Libraries



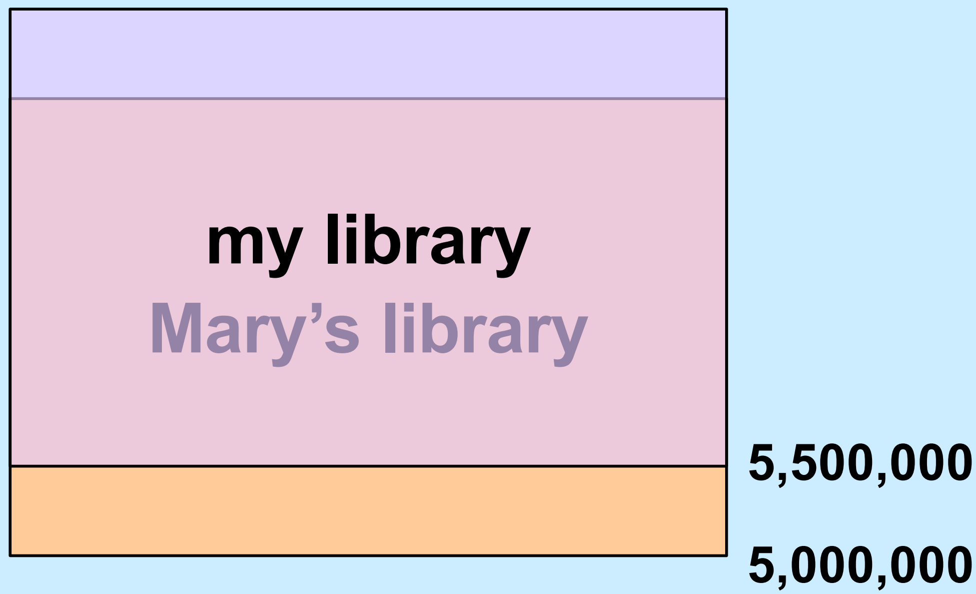
Problem

- **How is relocation handled?**

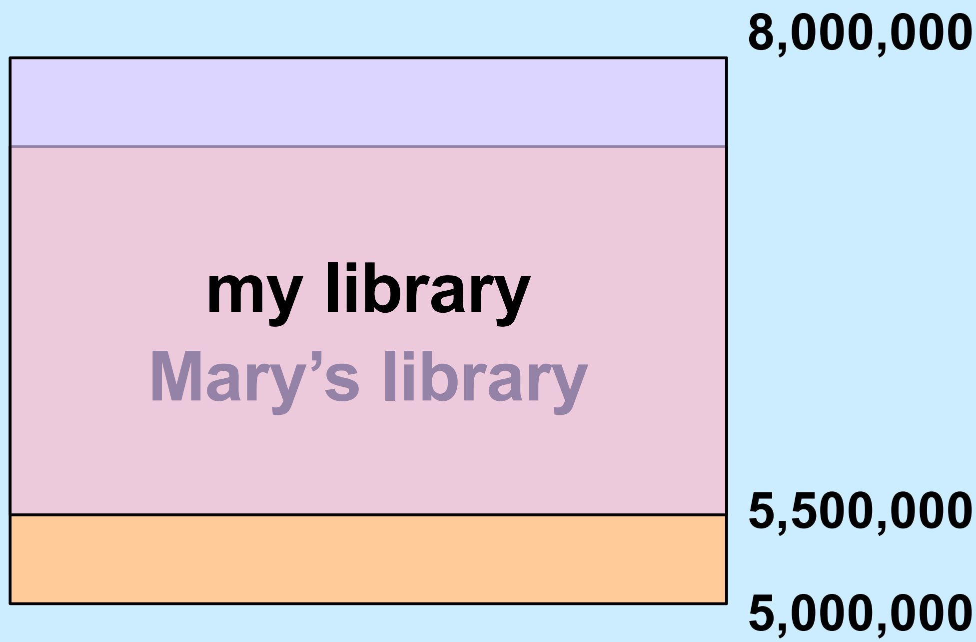
Pre-Relocation



But …



But …



Quiz 2

We need to relocate all references to Mary's library in my library. What option should we give to *mmap* when we map my library into our address space?

a) the MAP_SHARED option

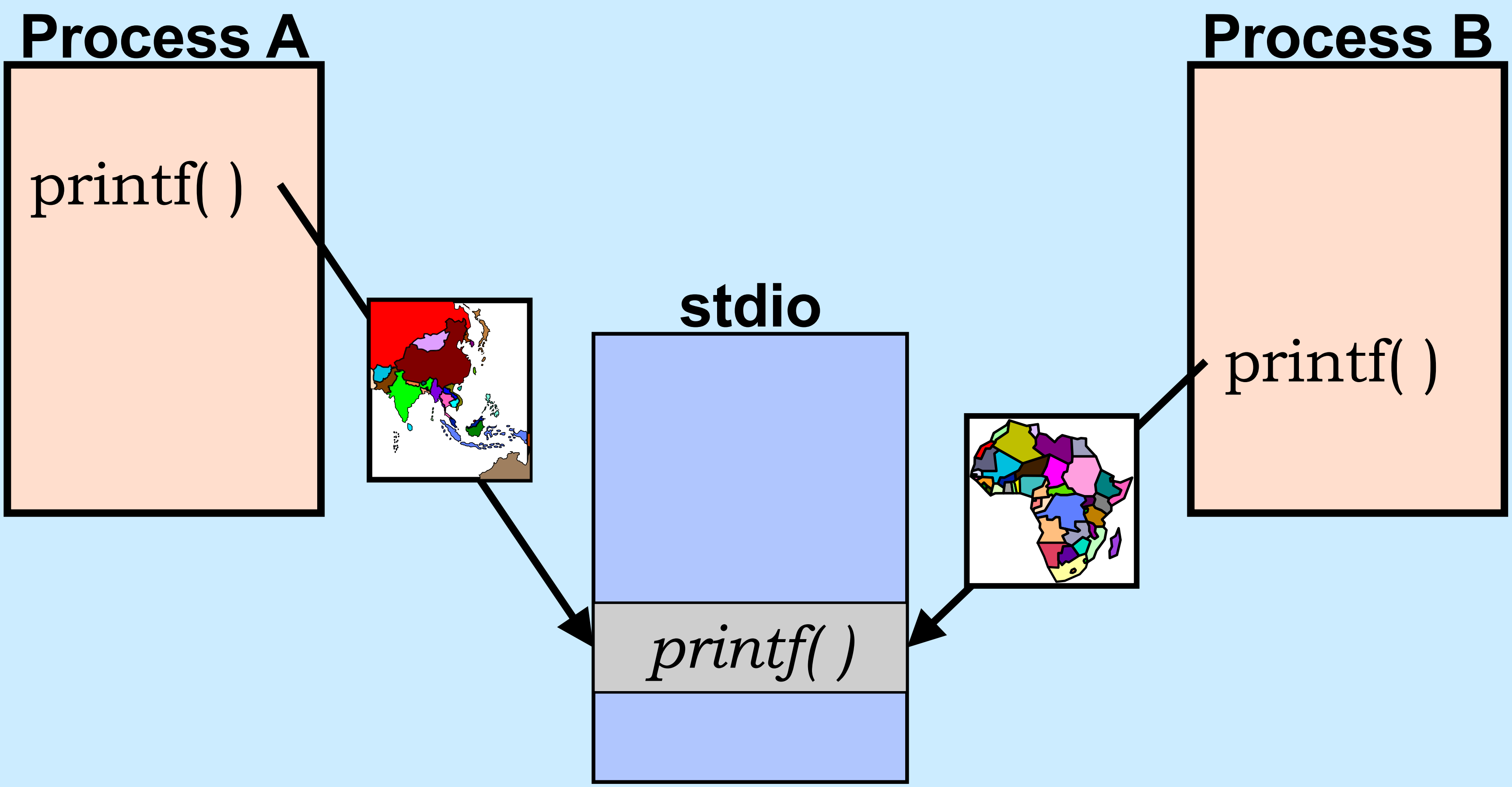
b) the MAP_PRIVATE option

c) mmap can't be used in this situation

Relocation Revisited

- Modify shared code to effect relocation
  - result is no longer shared!
- Separate shared code from (unshared) addresses
  - position-independent code (PIC)
  - code can be placed anywhere
  - addresses in separate private section
    - » pointed to by a register

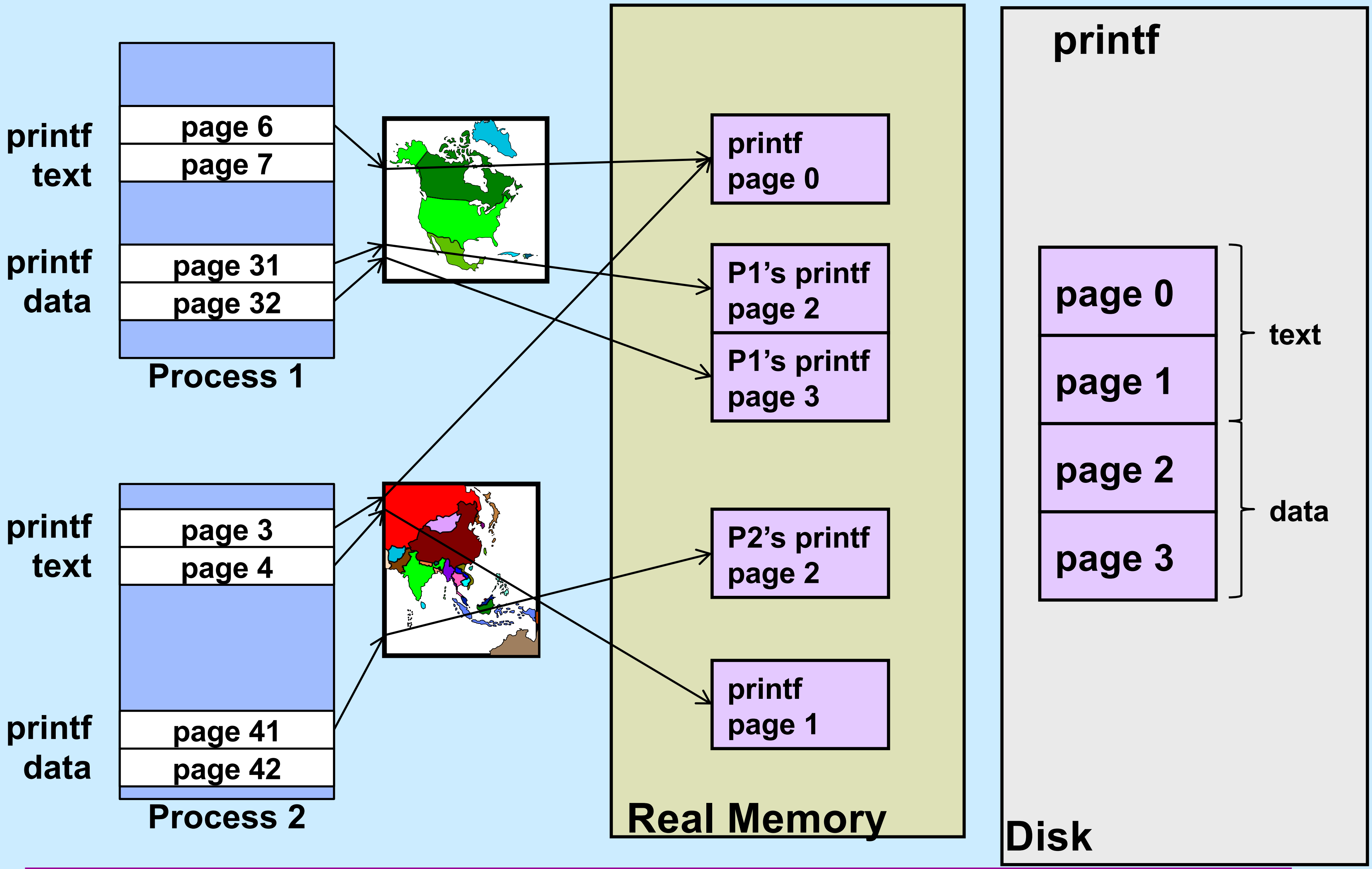
Mapping Shared Objects



Mapping printf into the Address Space

- **Printf's text**
  - read-only
  - can it be shared?
    - » yes: use MAP_SHARED
- **Printf's data**
  - read-write
  - not shared with other processes
  - initial values come from file
  - can mmap be used?
    - » MAP_SHARED wouldn't work
      - changes made to data by one process would be seen by others
    - » MAP_PRIVATE does work!
      - mapped region is initialized from file
      - changes are private

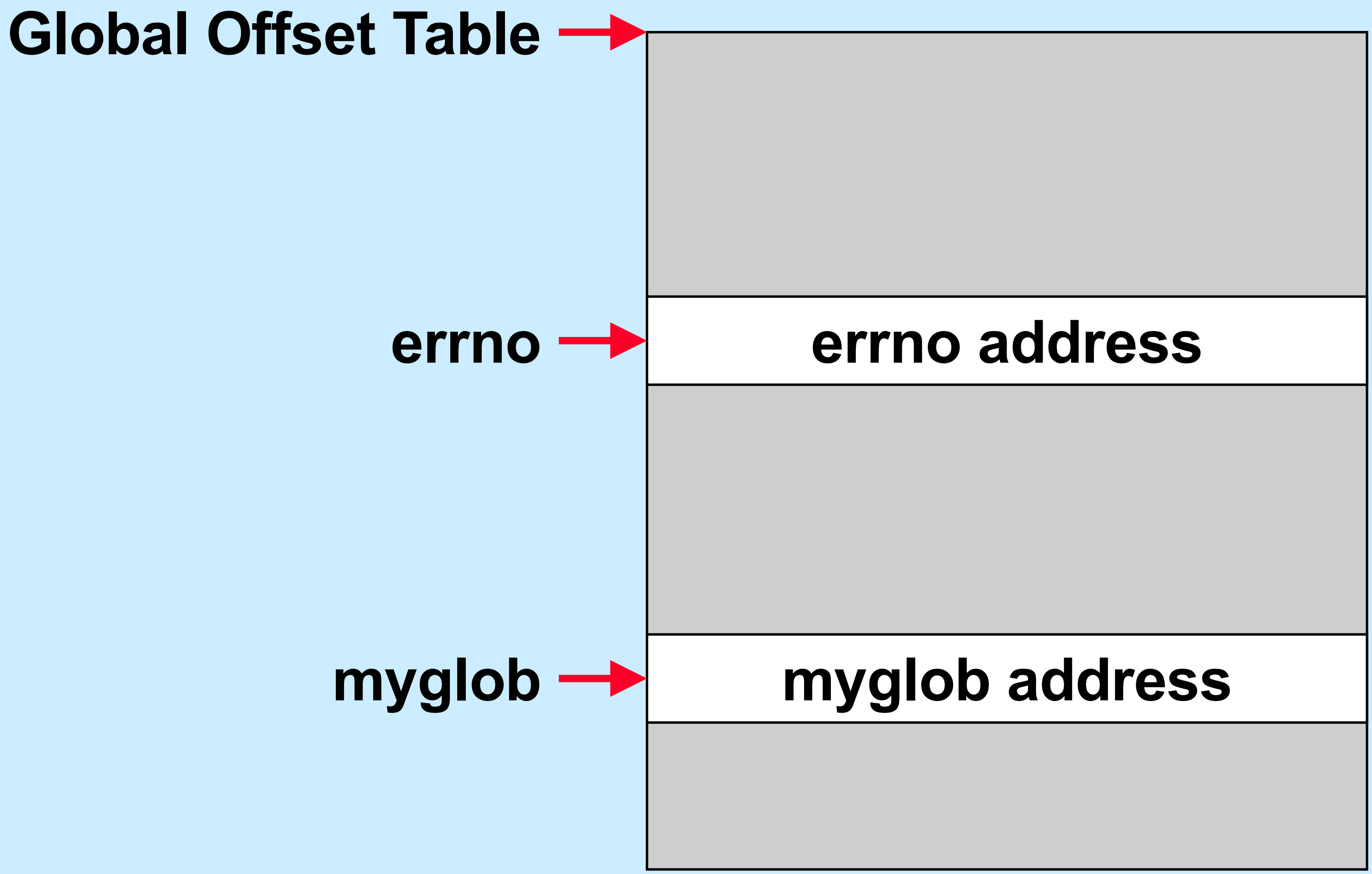
Mapping printf



Position-Independent Code

- Produced by gcc when given the –fPIC flag
- Processor-dependent; x86-64:
  - each dynamic executable and shared object has:
    - » procedure-linkage table
      - shared, read-only executable code
      - essentially stubs for calling functions
    - » global-offset table
      - private, read-write data
      - relocated dynamically for each process
    - » relocation table
      - shared, read-only data
      - contains relocation info and symbol table

Global-Offset Table: Data References



Functions in Shared Objects

- Lots of them
- Many are never used
- Fix up linkages on demand

An Example

```
int main( ) {
    puts("Hello world\n");
    …
    return 0;
}
```

```
00000000000006b0 <main>:
 6b0: 55                    push   %rbp
 6b1: 48 89 e5              mov    %rsp,%rbp
 6b4: 48 8d 3d 99 00 00 00  lea    0x99(%rip),%rdi
 6bb: e8 a0 fe ff ff        callq  560 <puts@plt>
 …
```


Before Calling puts

```
.PLT0:
  pushq GOT+8(%rip)
  jmp   *GOT+16(%rip)
  nop; nop
  nop; nop
.puts:
  jmp   *puts@GOT(%rip)
.putsnext
  pushq $putsRelOffset
  jmp   .PLT0
.PLT2:
  jmp   *name2@GOT(%rip)
.PLT2next
  pushq $name2RelOffset
  jmp   .PLT0
```

Procedure-Linkage Table

```
GOT:
   .quad _DYNAMIC
   .quad identification
   .quad ld-linux.so


puts:
   .quad .putsnext
name2:
   .quad .PLT2next
```

Relocation info:

```
GOT_offset(puts), symx(puts)
```

```
GOT_offset(name2), symx(name2)
```

Relocation Table

After Calling puts

```
.PLT0:
  pushq GOT+8(%rip)
  jmp   *GOT+16(%rip)
  nop; nop
  nop; nop
.puts:
  jmp   *puts@GOT(%rip)
.putsnext
  pushq $putsRelOffset
  jmp   .PLT0
.PLT2:
  jmp   *name2@GOT(%rip)
.PLT2next
  pushq $name2RelOffset
  jmp   .PLT0
```

Procedure-Linkage Table

```
GOT:
   .quad _DYNAMIC
   .quad identification
   .quad ld-linux.so


puts:
   .quad puts
name2:
   .quad .PLT2next
```

Relocation info:

`GOT_offset(puts), symx(puts)`
`GOT_offset(name2), symx(name2)`

Relocation Table

Quiz 2

On the second and subsequent calls to *puts*

a) control goes directly to *puts*
b) control goes to an instruction that jumps to *puts*
c) control still goes to ld-linux.so, but it now transfers control directly to *puts*

