CS 33

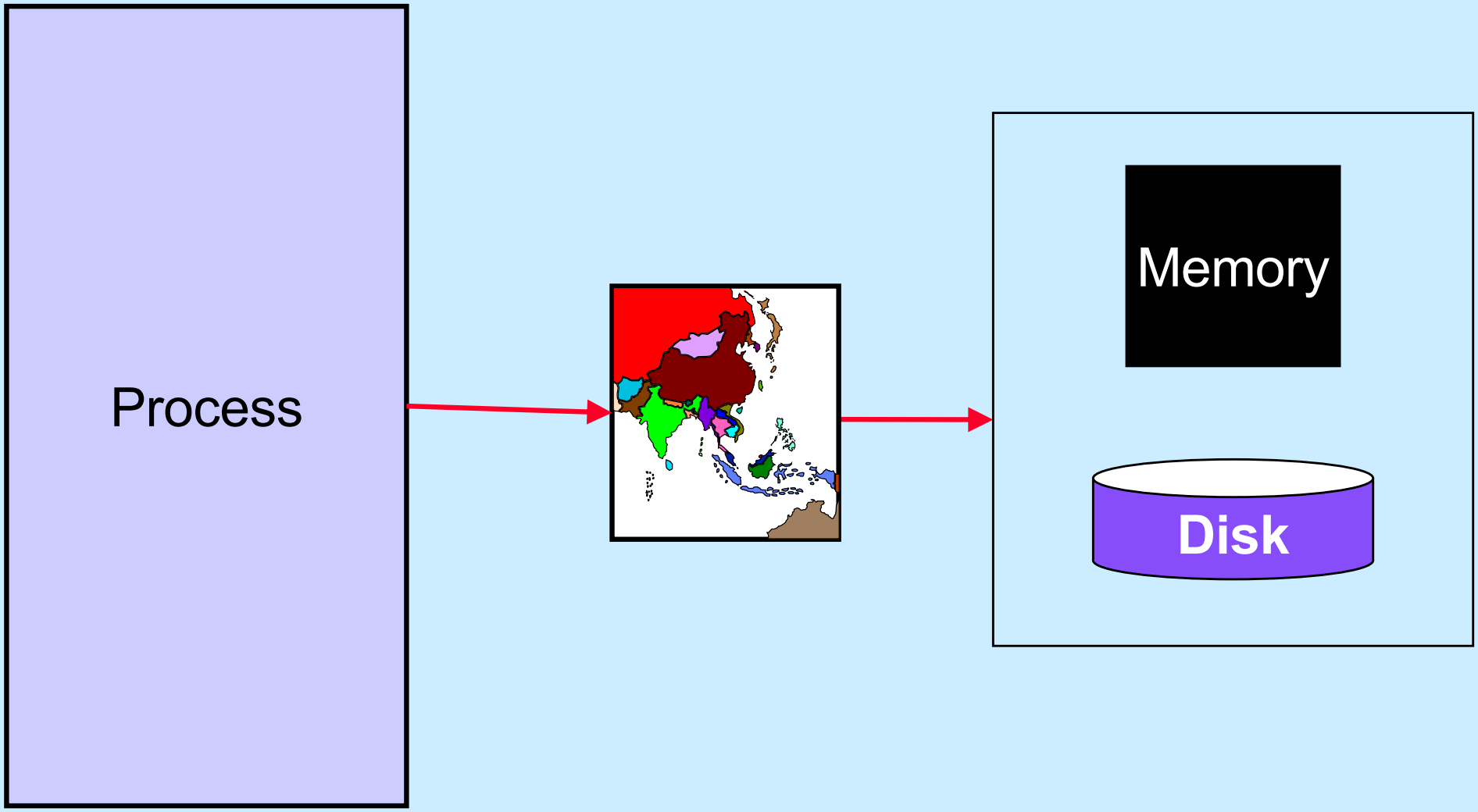
Virtual Memory 2

OS Role in Virtual Memory

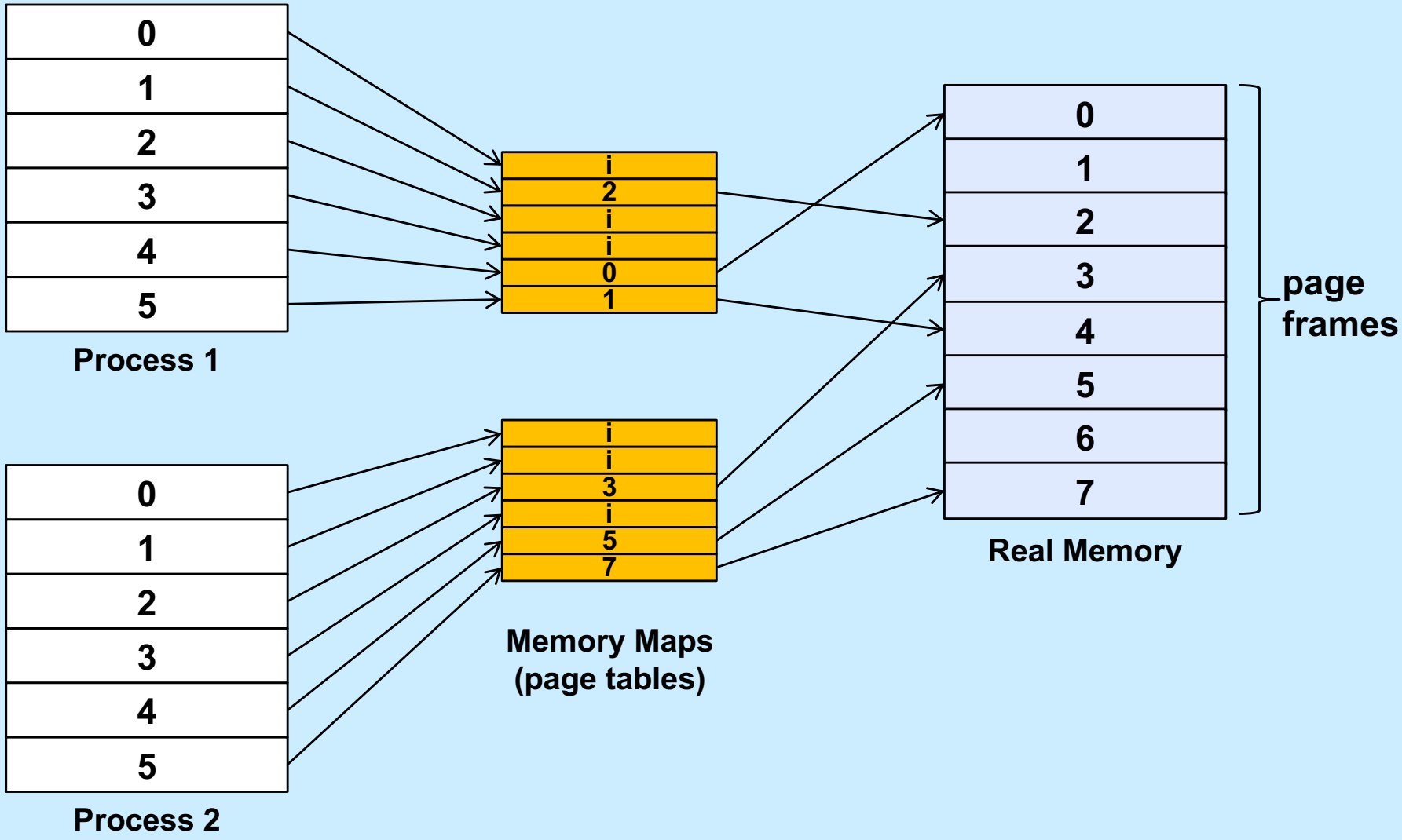
- Memory is like a cache
  - quick access if what's wanted is mapped via page table
  - slow if not — OS assistance required
- OS
  - make sure what's needed is mapped in
  - make sure what's no longer needed is not mapped in

Why is virtual memory used?

More VM than RM

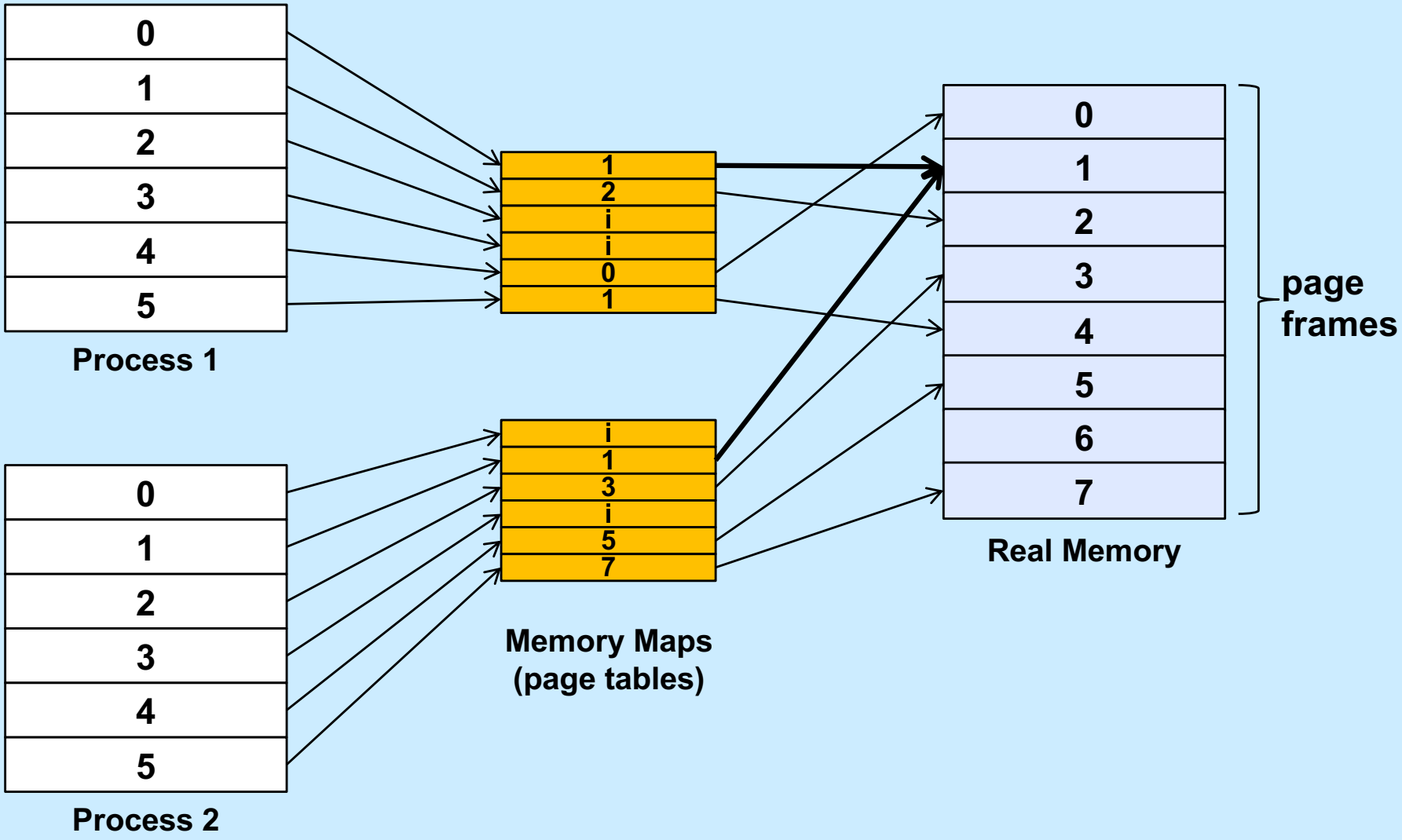


Isolation

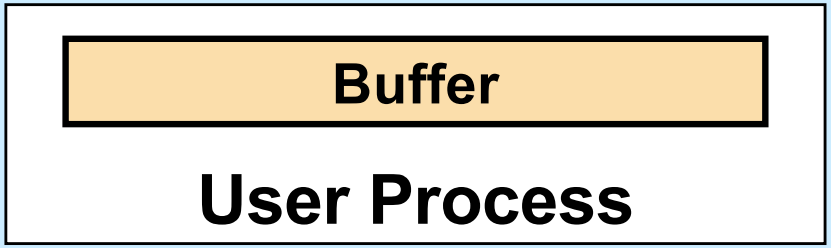


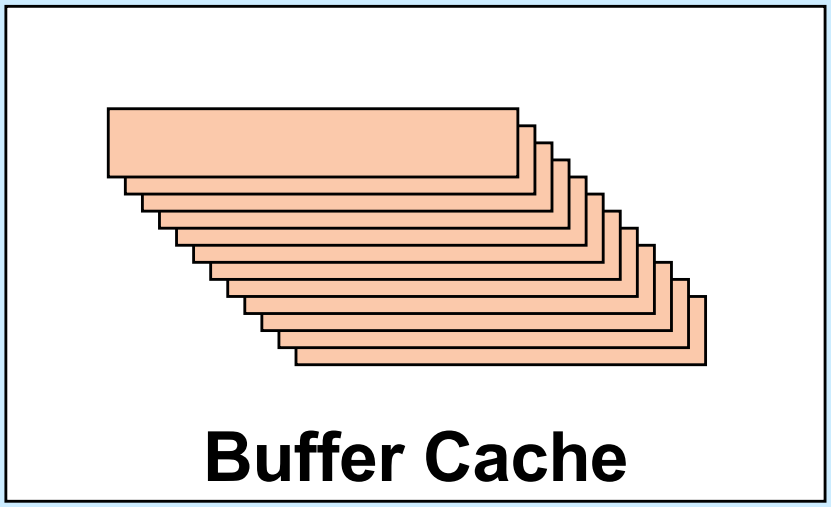
Virtual Memory

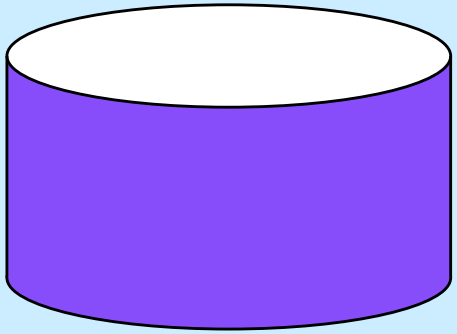
Sharing



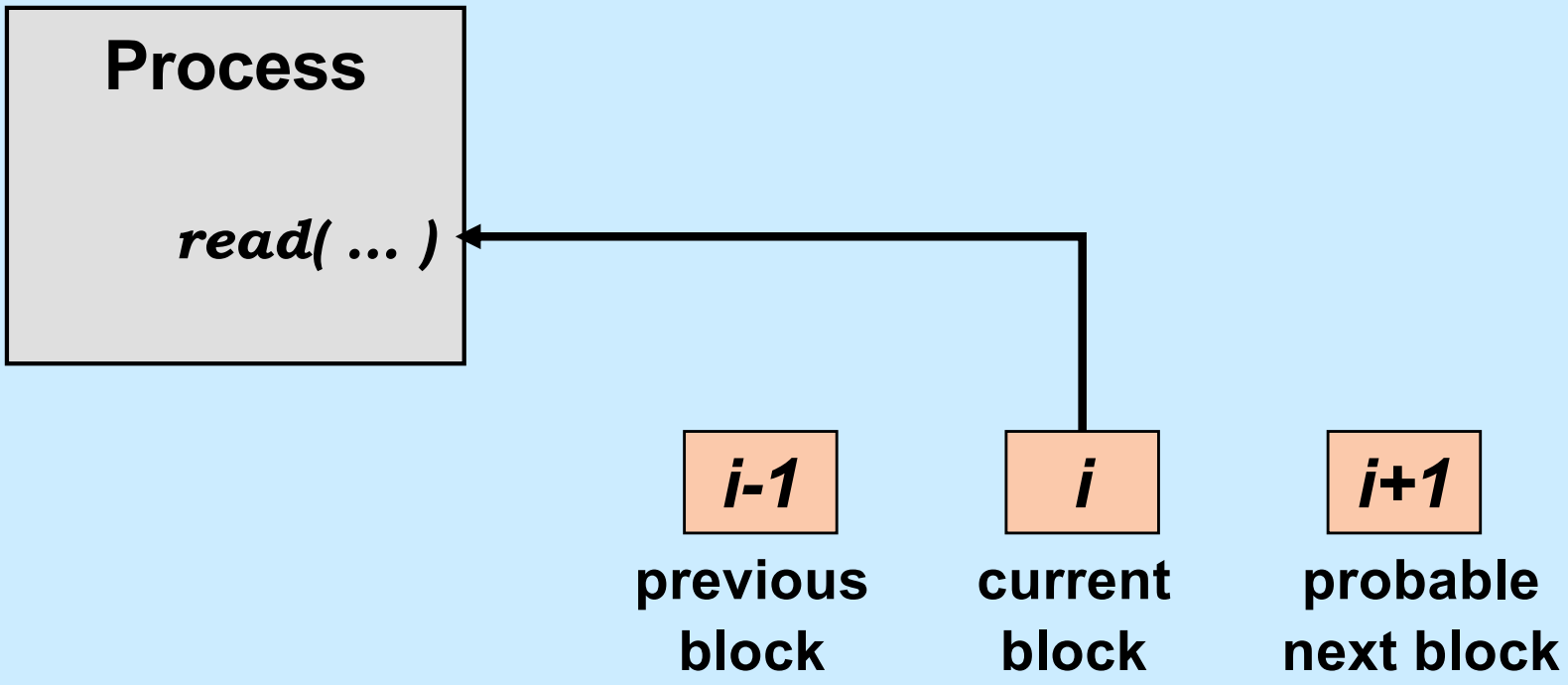
File I/O



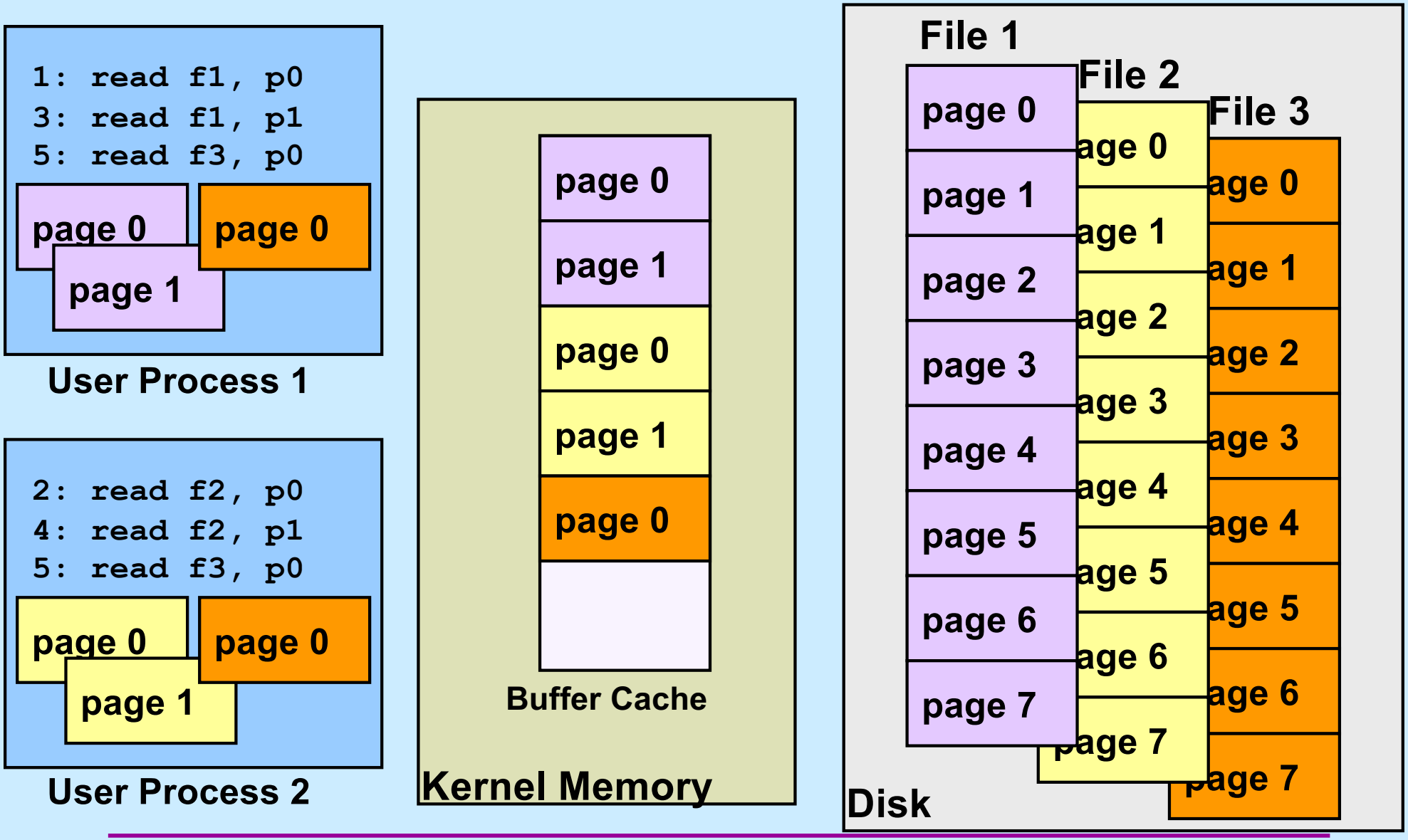




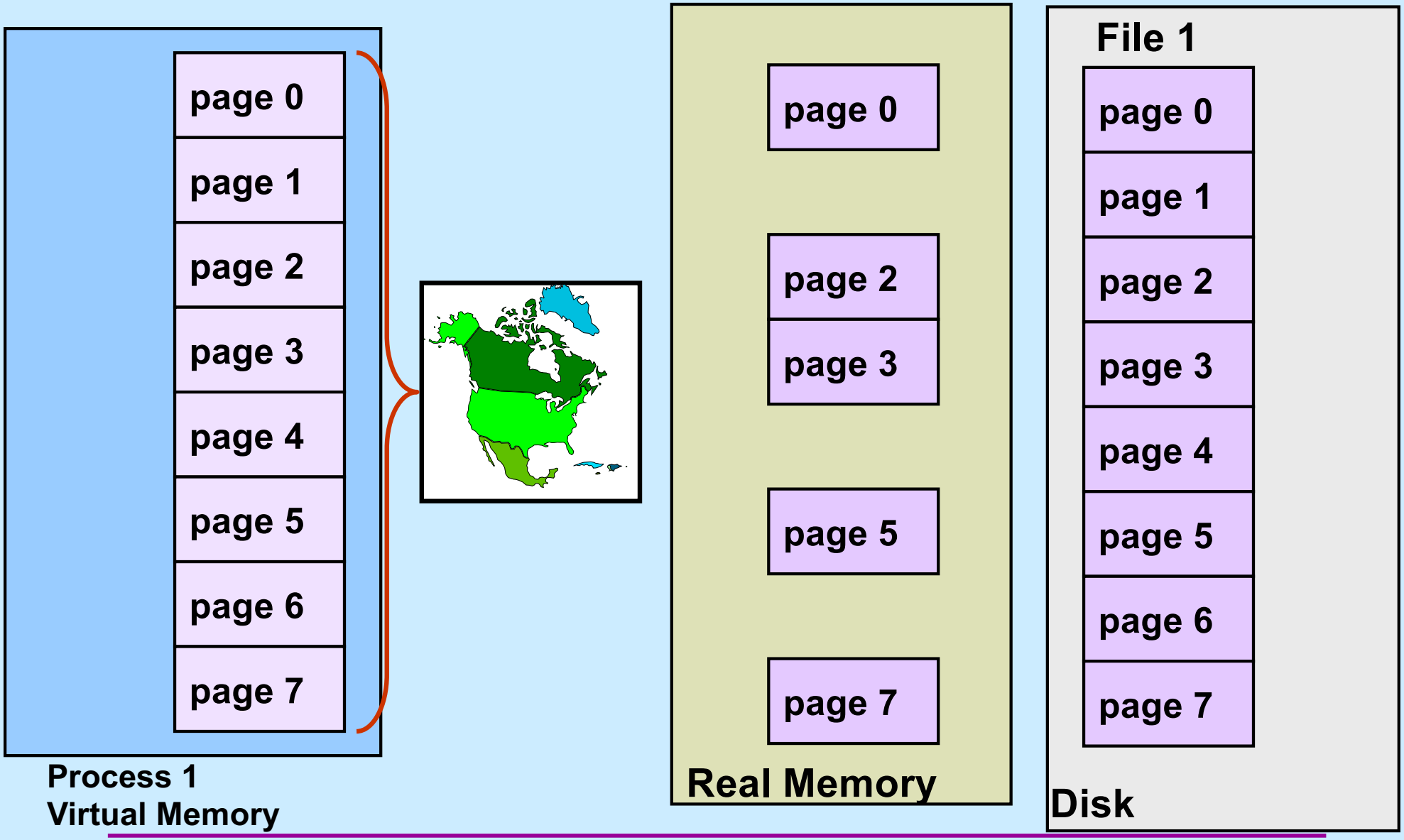
Multi-Buffered I/O



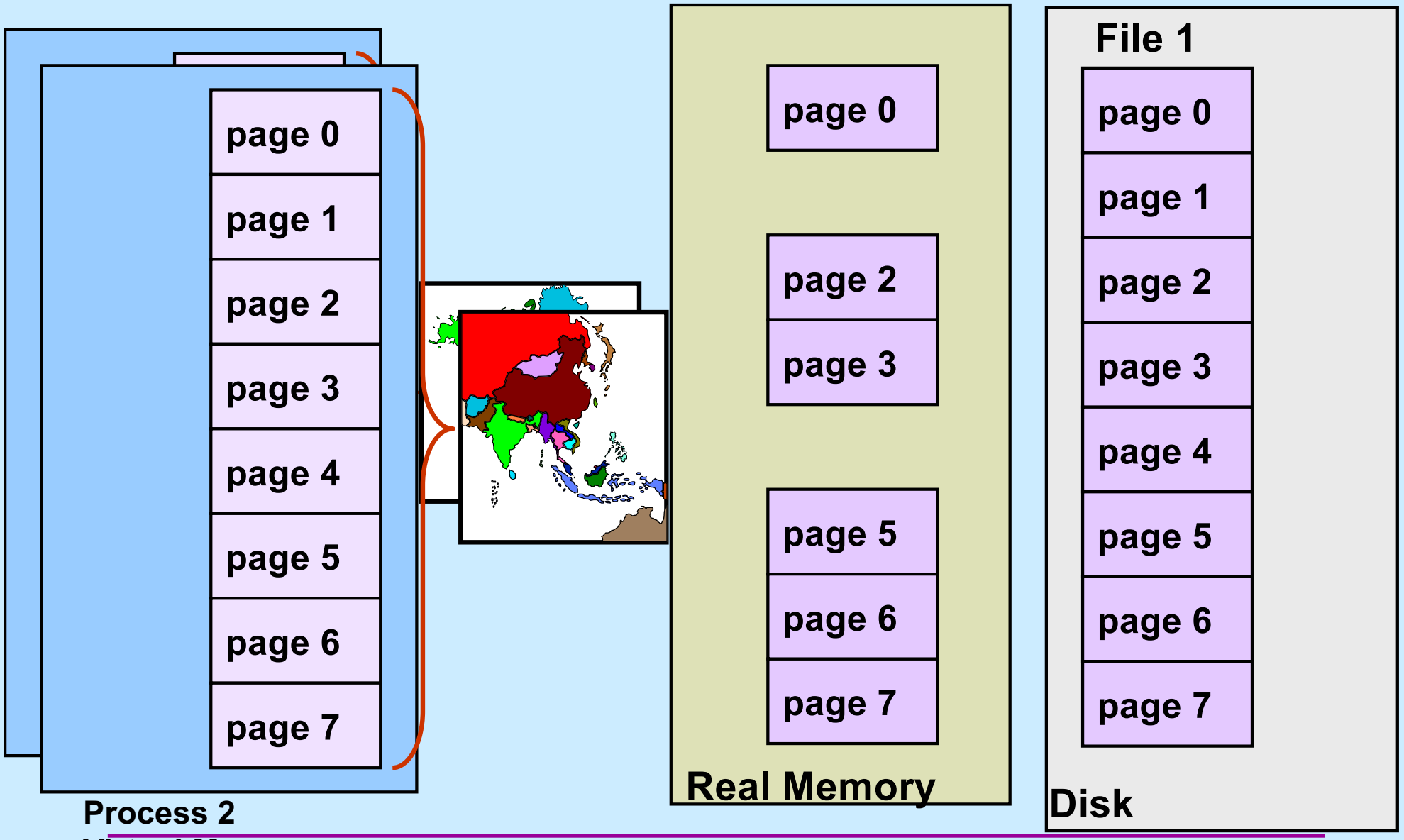
Traditional I/O



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);
}
```

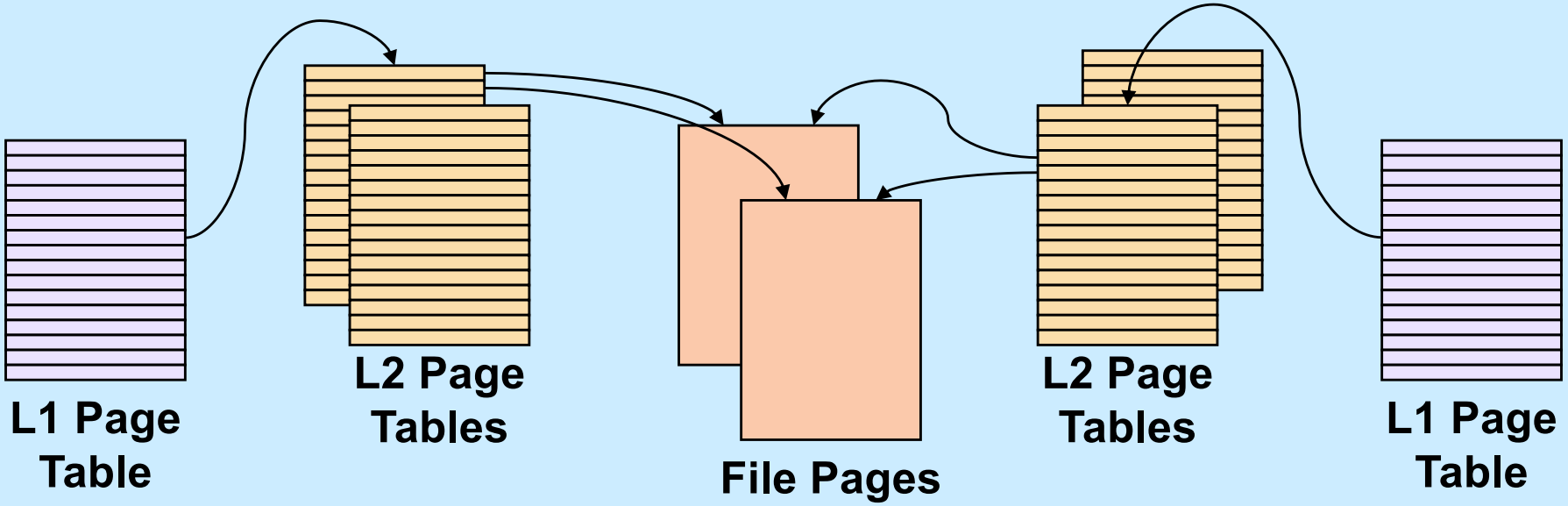
- **Mapped File I/O**

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

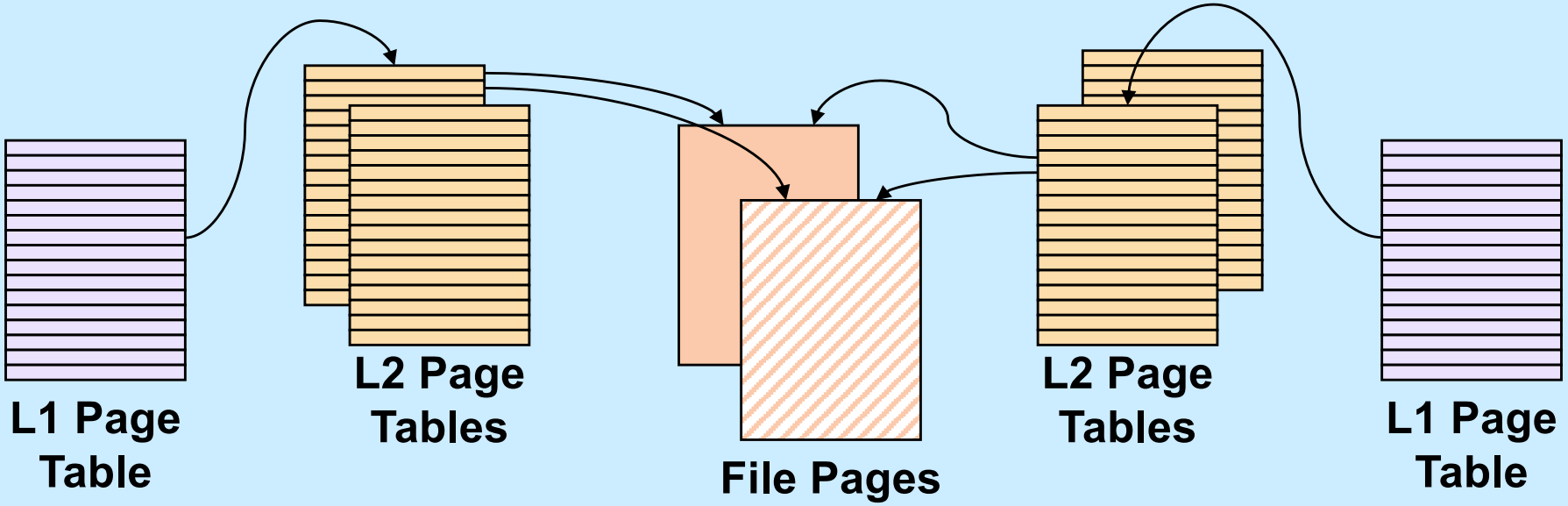
Mmap System Call

```
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

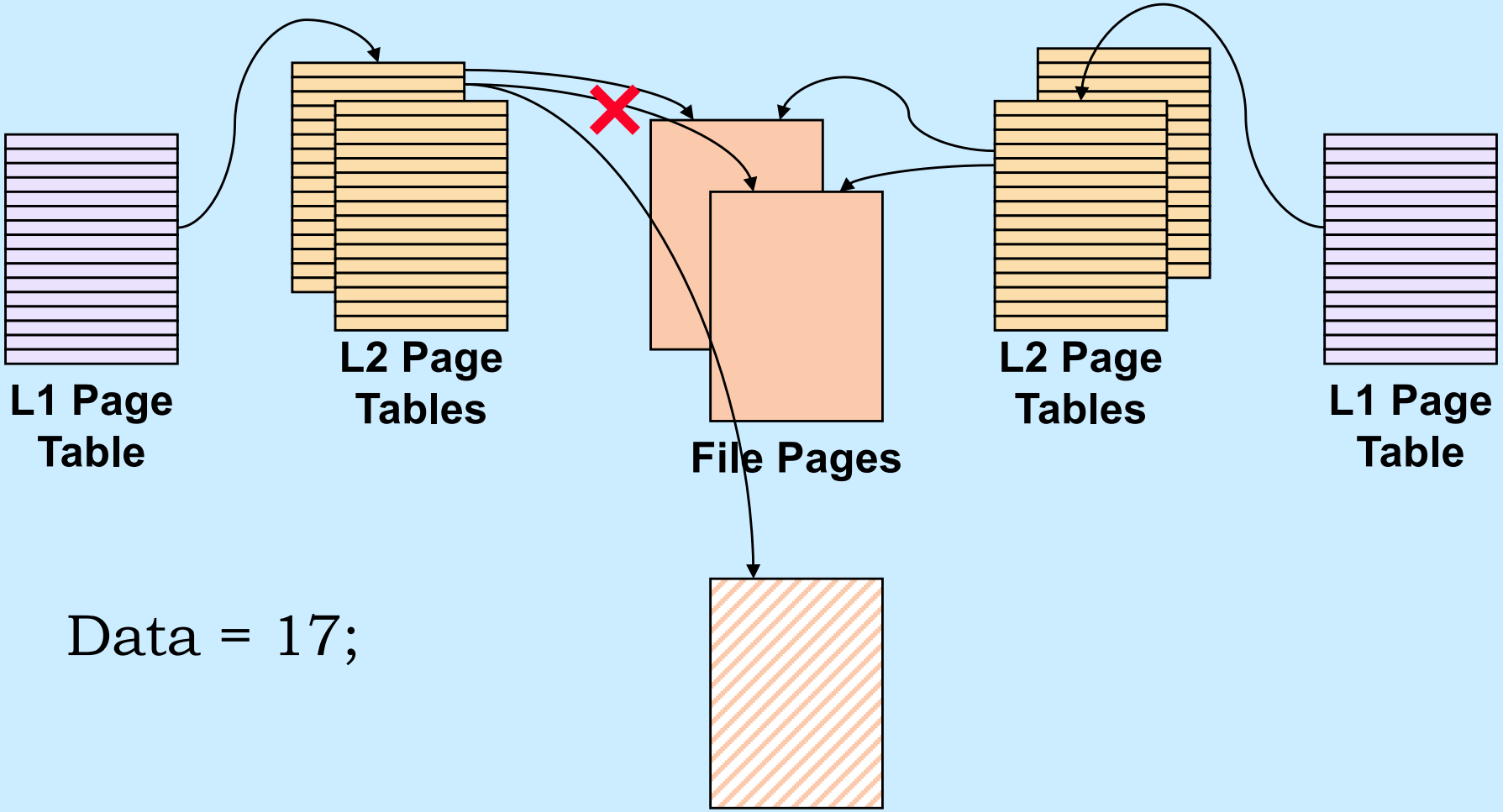


Share-Mapped Files



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;
}
```

Putting Together a Program

gcc Steps

1) Compile
   - to start here, supply .c file
   - to stop here: `gcc -S` (produces .s file)
   - if not stopping here, gcc compiles directly into a .o file, bypassing the assembler
2) Assemble
   - to start here, supply .s file
   - to stop here: `gcc -c` (produces .o file)
3) Link
   - to start here, supply .o file

The Linker

- An executable program is one that is ready to be loaded into memory
- The linker (known as ld: /usr/bin/ld) creates such executables from:
  - object files produced by the compiler/assembler
  - collections of object files (known as libraries or archives)
  - and more we'll get to soon ...

Linker's Job

- Piece together components of program
  - arrange within address space
    - » code (and read-only data) goes into text region
    - » initialized data goes into data region
    - » uninitialized data goes into bss region
- Modify address references, as necessary

A Program

```
int nprimes = 100;
int *prime, *prime2;
int main() {
    int i, j, current = 1;
    prime = (int *)malloc(nprimes*sizeof(*prime));
    prime2 = (int *)malloc(nprimes*sizeof(*prime2));
    prime[0] = 2; prime2[0] = 2*2;
    for (i=1; i<nprimes; i++) {
    NewCandidate:
        current += 2;
        for (j=0; prime2[j] <= current; j++) {
            if (current % prime[j] == 0)
                goto NewCandidate;
        }
        prime[i] = current; prime2[i] = current*current;
    }
    return 0;
}
```

data

bss

dynamic

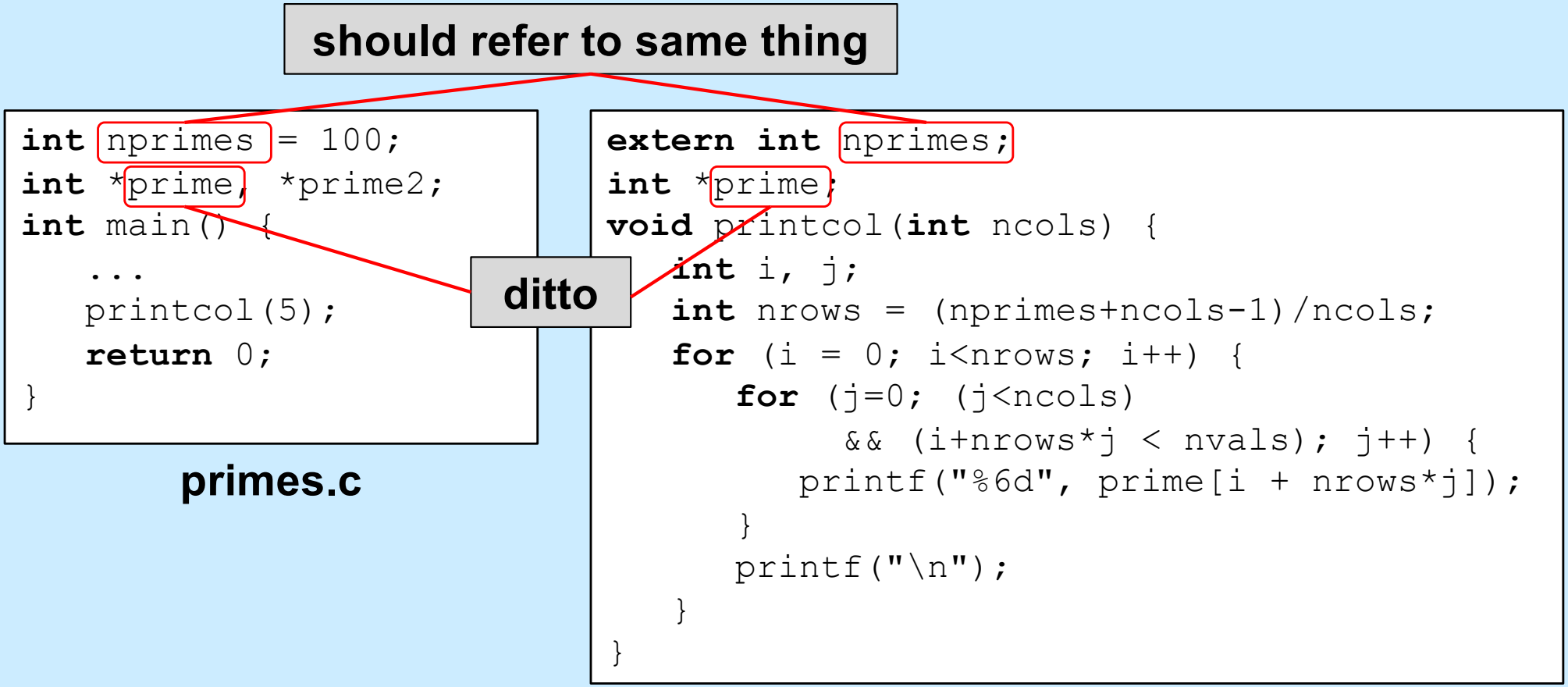
text

... with Output

```
int nprimes = 100;
int *prime, *prime2;
int main() {
   ...
   printcol(5);
   return 0;
}

void printcol(int ncols) {
   int i, j;
   int nrows = (nprimes+ncols-1)/ncols;
   for (i = 0; i<nrows; i++) {
      for (j=0; (j<ncols) && (i+nrows*j < nvals); j++) {
         printf("%6d", prime[i + nrows*j]);
      }
      printf("\n");
   }
}
```


... Compiled Separately



primes.c

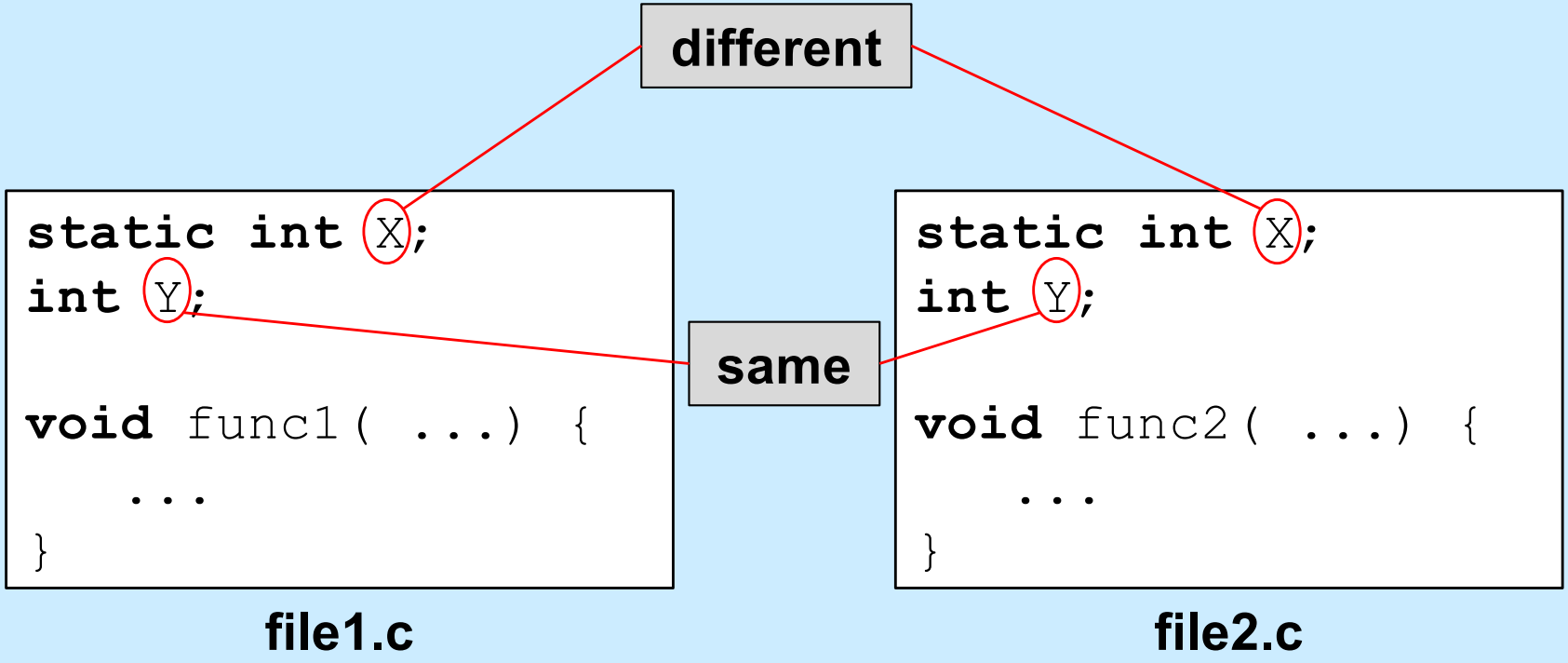
printcol.c

gcc –c primes.c
gcc –c printcol.c
gcc –o primes primes.o printcol.o

Global Variables

- **Initialized vs. uninitialized**
  - **initialized allocated in *data* section**
  - **uninitialized allocated in *bss* section**
    - » **implicitly initialized to zero**
- **File scope vs. program scope**
  - ***static* global variables known only within file that declares them**
    - » **two of same name in different files are different**
    - » **e.g., `static int X;`**
  - **non-static global variables potentially shared across all files**
    - » **two of same name in different files are same**
    - » **e.g., `int X;`**

Scope

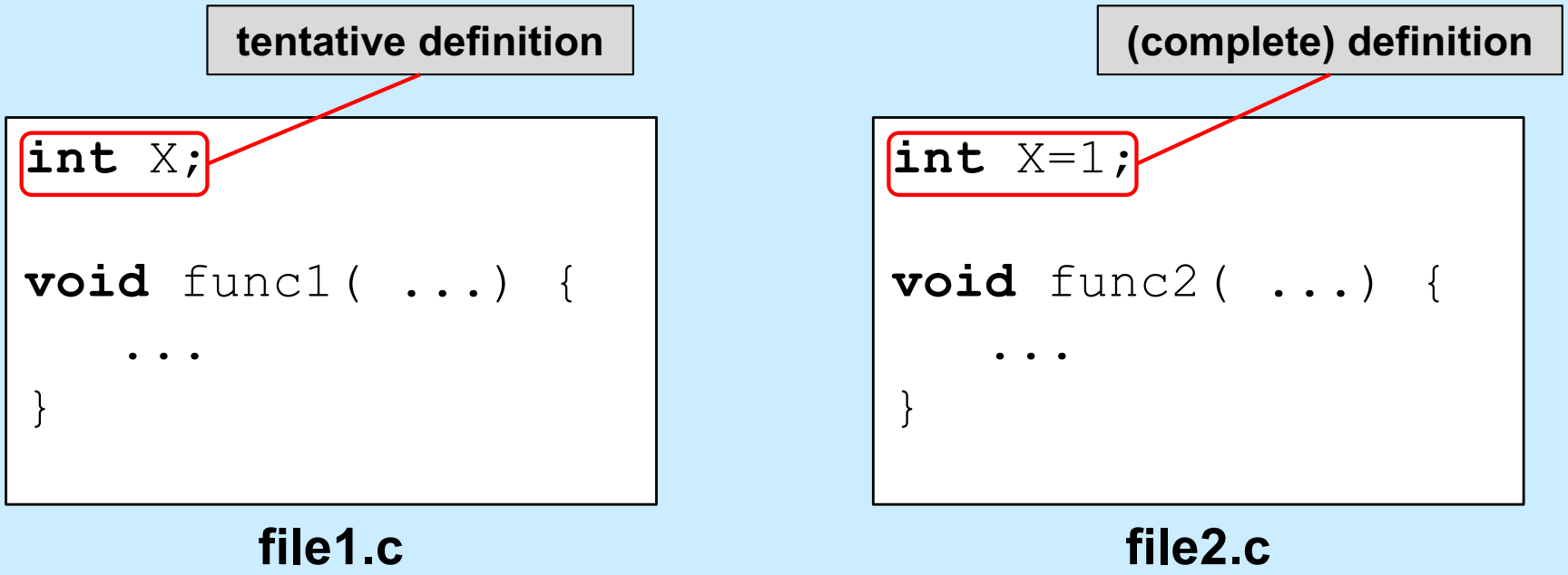


Static Local Variables

```
int *sub1() {
  int var = 1;
  …
  return &var;
  /* amazingly illegal */
}
```

```
int *sub2() {
  static int var = 1;
  …
  return &var;
  /* (amazingly) legal */
}
```

Reconciling Program Scope (1)



Where does X go?
What's its initial value?

- **tentative definitions overridden by compatible (complete) definitions**
- **if not overridden, then initial value is zero**

Reconciling Program Scope (2)

```
int X=2;

void func1( ...) {
    ...
}
```

file1.c

```
int X=1;

void func2( ...) {
    ...
}
```

file2.c

What happens here?

Reconciling Program Scope (3)

```
int X=1;

void func1( ...) {
    ...
}
```

file1.c

```
int X=1;

void func2( ...) {
    ...
}
```

file2.c

Is this ok?

Reconciling Program Scope (4)

```
extern int X;

void func1( ...) {
    ...
}
```

file1.c

```
int X=1;

void func2( ...) {
    ...
}
```

file2.c

What's the purpose of "extern"?

Does Location Matter?

```
int main(int argc, char *[]) {
   return(argc);
}
```

```
main:
  pushq %rbp    ; push frame pointer
  movq  %rsp, %rbp     ; set frame pointer to point to new frame
  movl  %edi, %eax     ; put argc into return register (eax)
  movq  %rbp, %rsp     ; restore stack pointer
  popq  %rbp    ; pop stack into frame pointer
  ret           ; return: pops end of stack into rip
```

Location Matters …

```
int X=6;
int *aX = &X;

int main() {
    void subr(int);
    int y=*aX;
    subr(y);
    return(0);
}

void subr(int i) {
    printf("i = %d\n", i);
}
```

Coping

- **Relocation**
  - **modify internal references according to where module is loaded in memory**
  - **modules needing relocation are said to be *relocatable***
    - » **which means they *require* relocation**
  - **the compiler/assembler provides instructions to the linker on how to do this**