Pintos User Programs II

Executing child processes
Controlling memory access

Recap

- So far, we know how to:
  - run and debug pintos tests
  - add our own tests
  - use `lib/kernel/list.h`
  - implement `sleep()` with no busy-wait, thread priority and niceness and system calls to be used by user programs

Next steps

- Today we implement two syscalls: `wait()` and `exec()`
- To implement `exec()`, we need:
  - synchronization
  (with `thread_block()`, or just a semaphore)
  - check the pointers passed by the user
  (requires some understanding of memory paging)

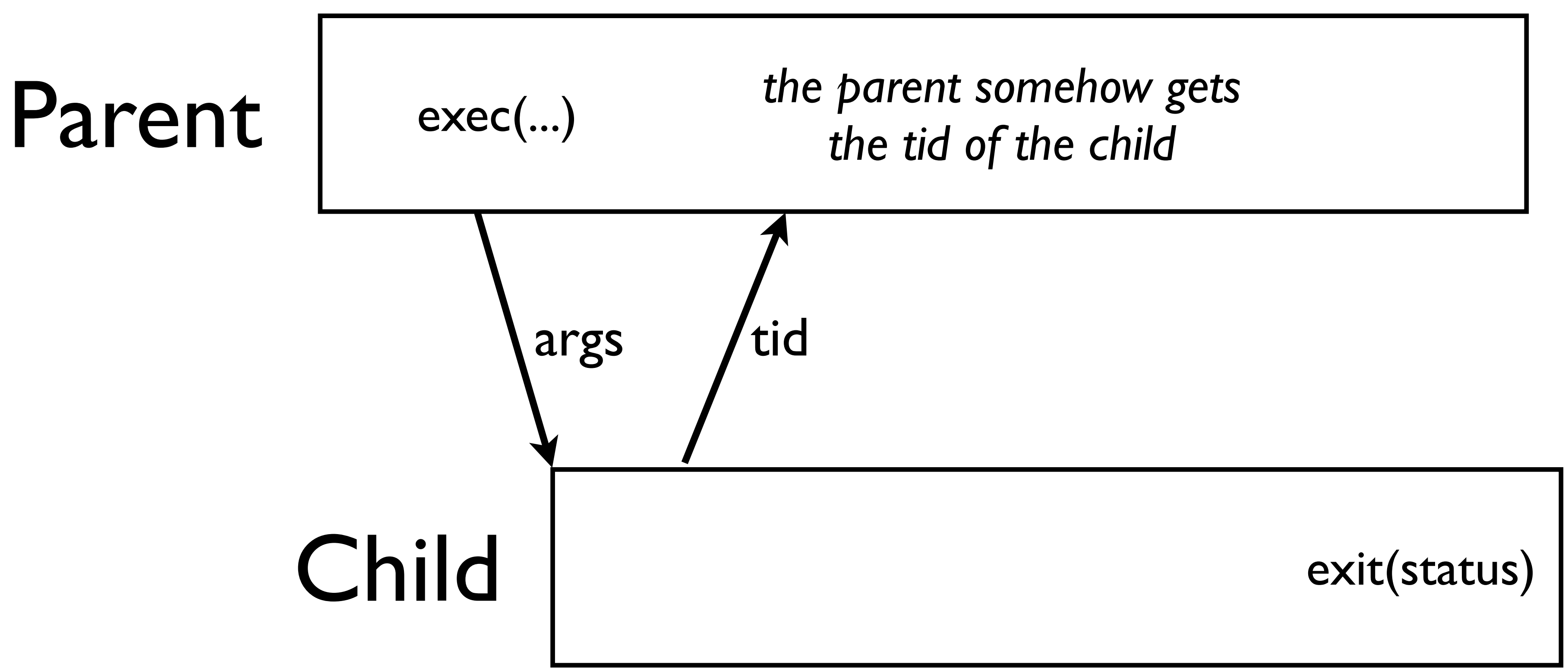
The `wait()` system call

- `int wait (pid_t pid)`
- The calling process blocks until its child p, which has id `pid`, has finished
- The return value must be:
  - the exit status of p, if all was fine
    (even if p finished before `wait` was called!)
  - -1 if p was killed
  - -1 if p is not a direct child of the caller
    (this makes it easier)
  - -1 if the calling process already waited for that pid before

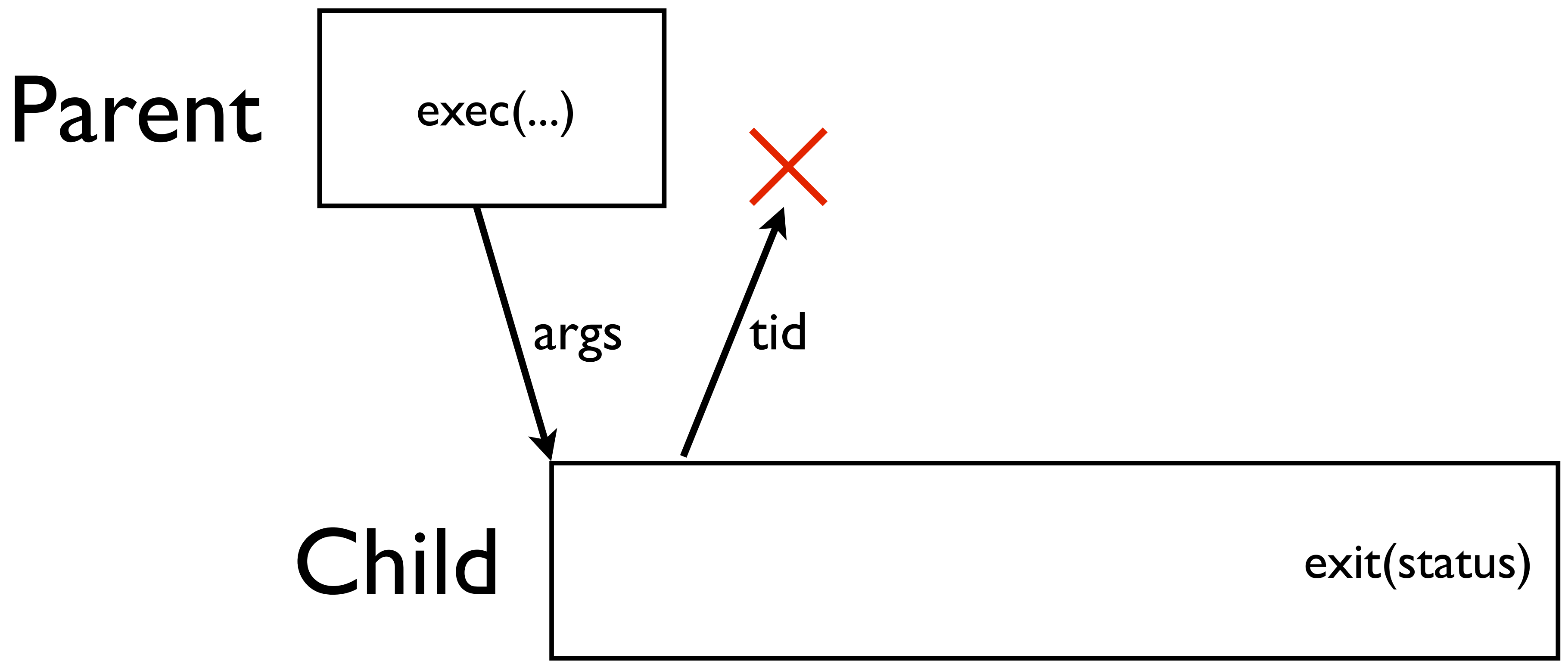
The `exec()` system call

- `pid_t exec (const char *cmd_line)`
- A user program can use it to execute a command, creating a child process
- It tries to run the executable given in `cmd_line`
- The return value is the pid (tid) of the child process (or -1 in case of error)

Executing - `exec(...)`

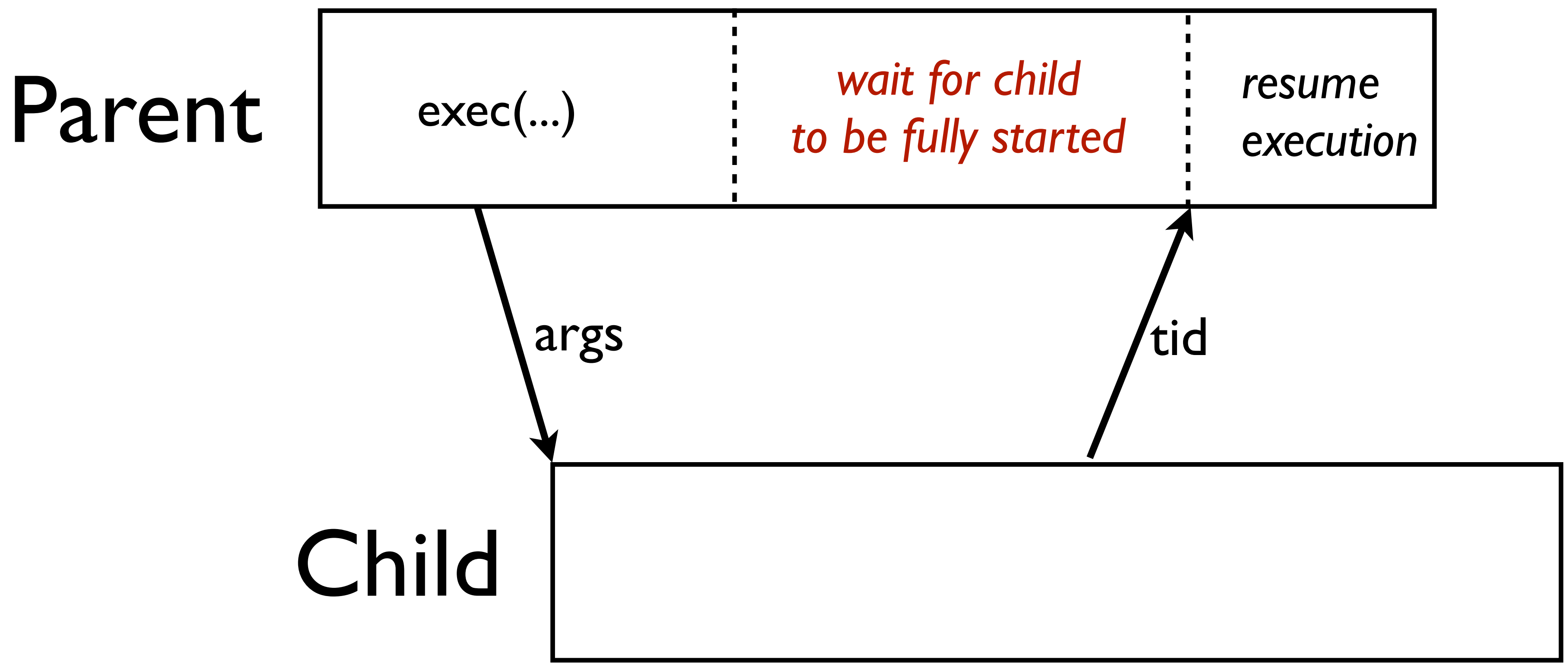


Executing - `exec(...)`



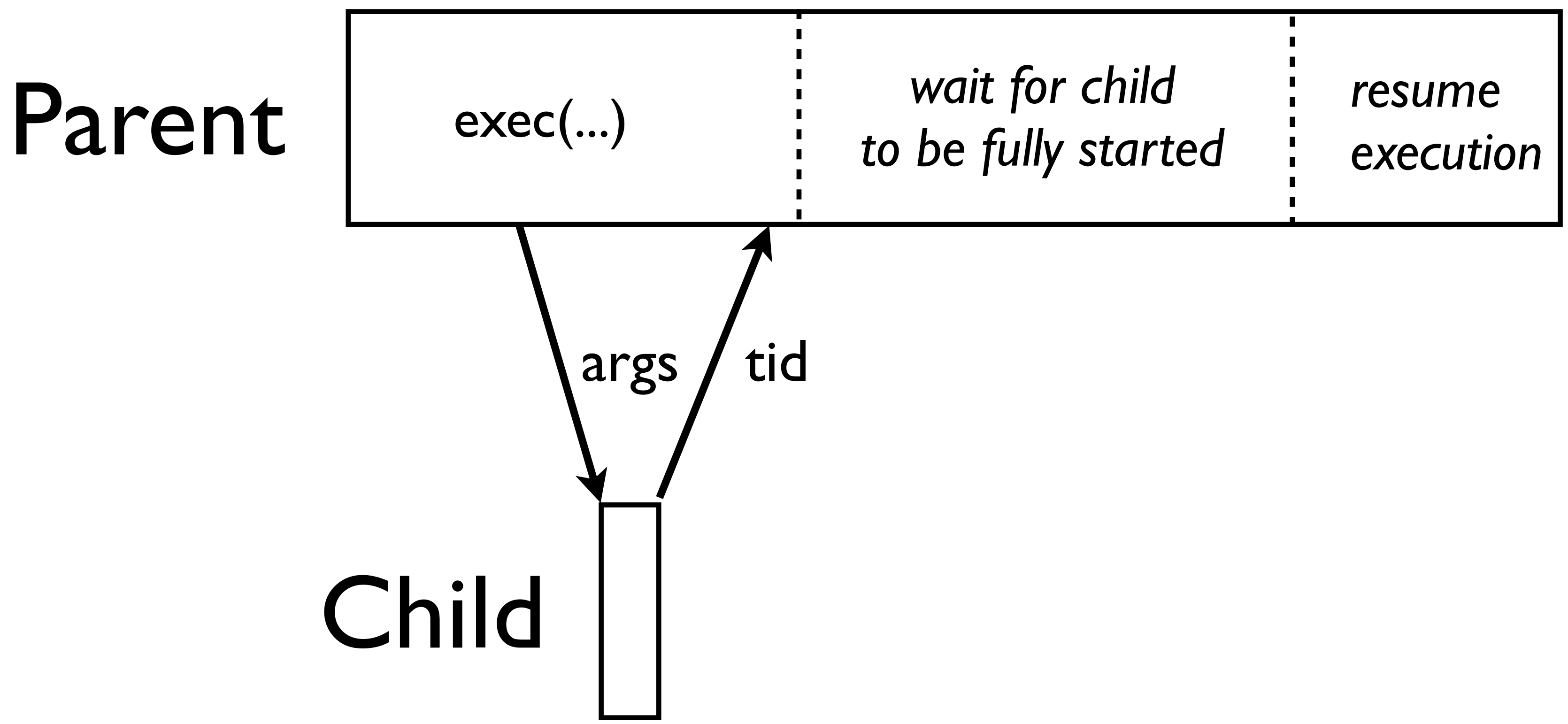
The parent thread must wait for the child creation

Executing - `exec(...)`



The parent thread must wait for the child creation

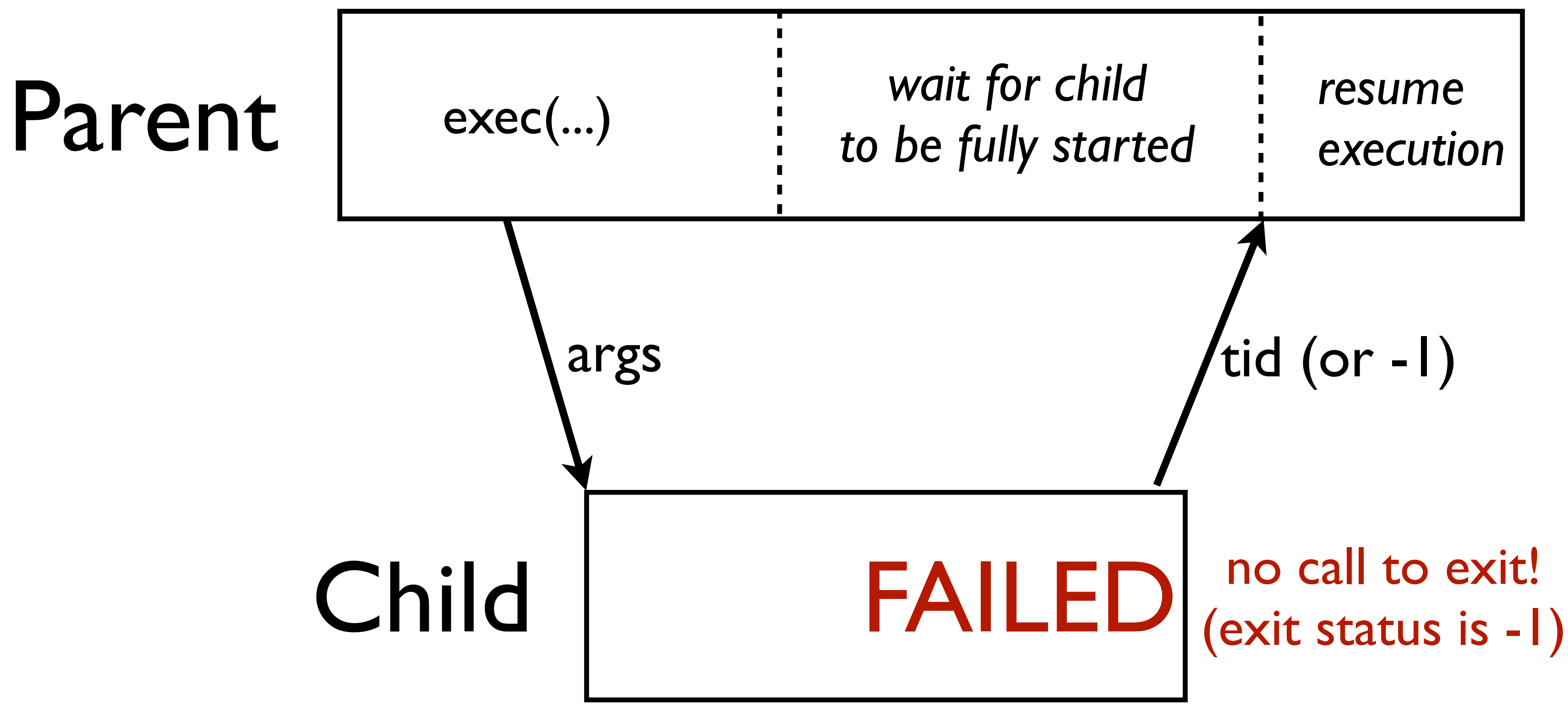
Executing - `exec(...)`



The parent thread must wait for the child creation

The child thread may finish too fast

Executing - `exec(...)`



The parent thread must wait for the child creation
The child thread may finish too fast
The child thread may fail before telling some status

Pintos semaphores

- Used for synchronizing threads
- Can solve the (wait for child) problem of the `exec` syscall
- Available including "thread/synch.h"
- Provided API:

```
struct semaphore s;
sema_init(&s, 0);
sema_up(&s);
sema_down(&s);
```

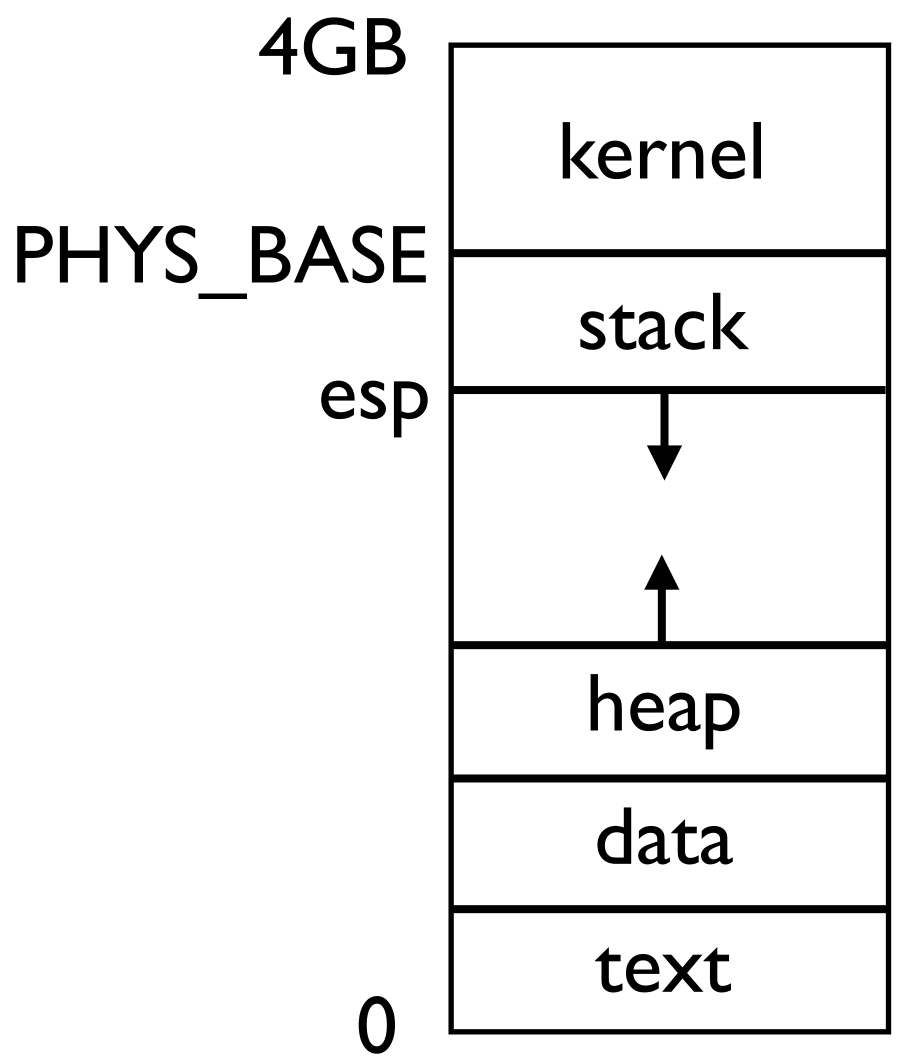
Pintos semaphores API

- `struct semaphore`
  semaphore type; must be initialized with sema_init
- `sema_init (struct semaphore * s, unsigned val)`
  initialize semaphore pointed by `s` with value `val`
- `sema_down (struct semaphore * s)`
  if `s` is 0 (zero), block the calling thread and put it in a list waiting for `s`; otherwise, decrement `s` by 1
- `sema_up (struct semaphore * s)`
  if `s` is 0 (zero) and there is some thread waiting for `s`, unblock one of the threads that are waiting for `s`; otherwise, increment `s` by 1
- neither sema_up or sema_down can be interrupted (they're *atomic*)

Memory access

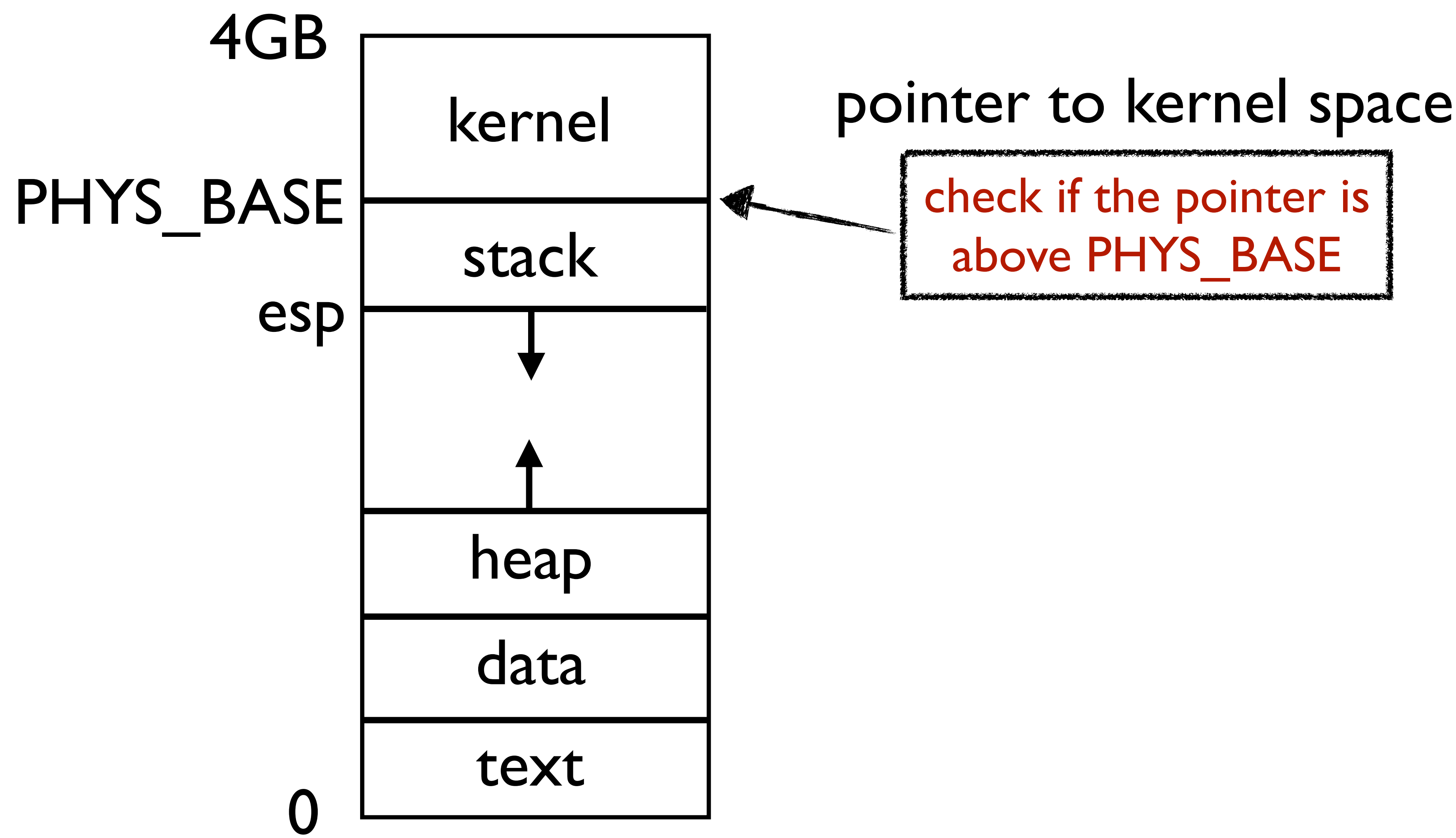
- The user may provide an invalid pointer in a syscall
  - a null pointer
  - a pointer to kernel address space
  - a pointer to unmapped virtual memory
- The kernel (you) should control this

Memory layout



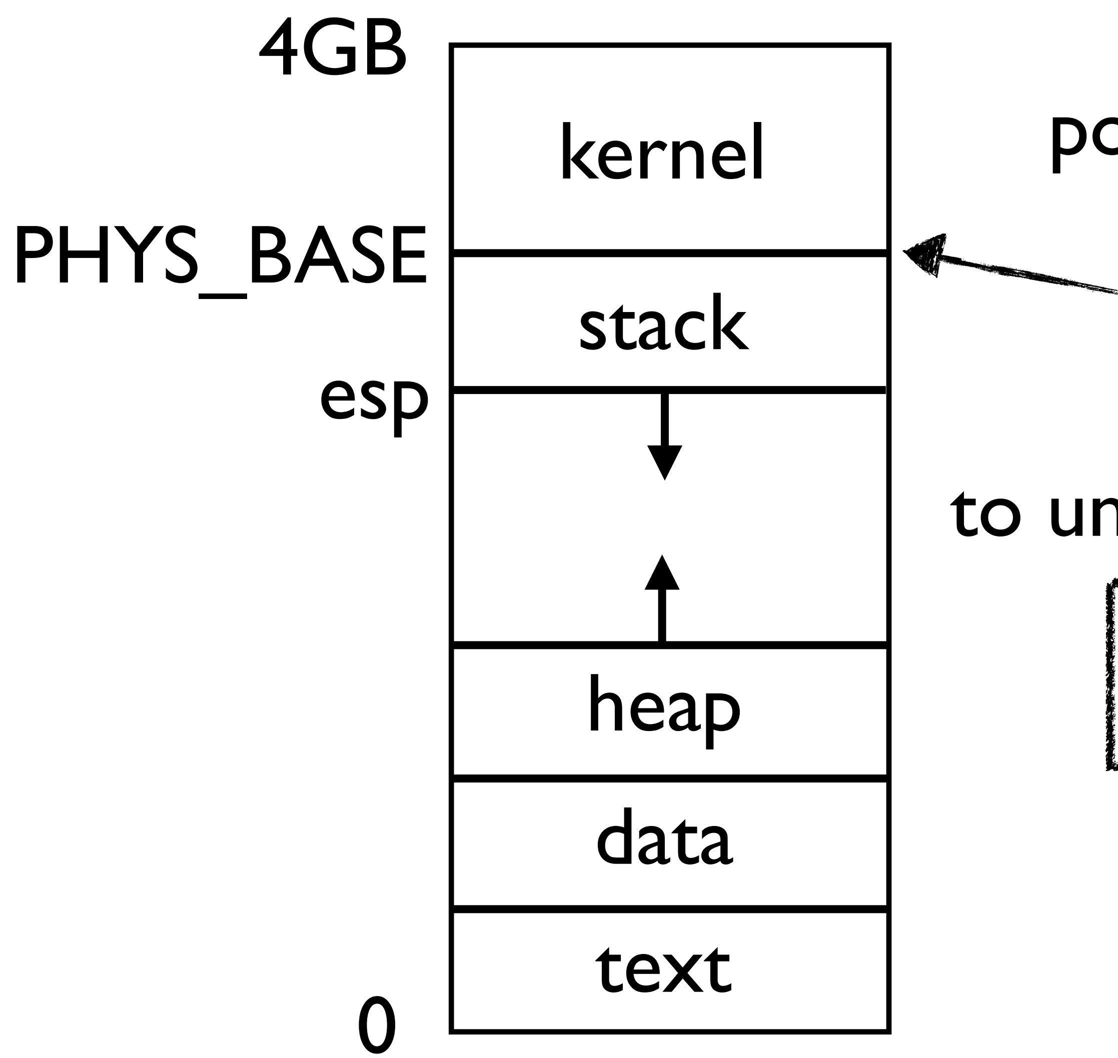
User process

Memory layout



User process

Memory layout



User process

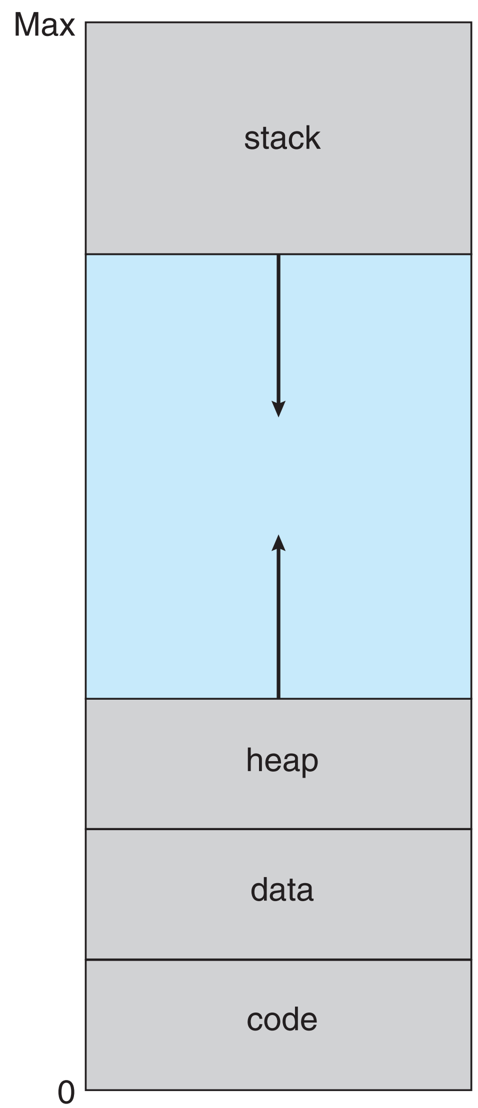
pointer to kernel space

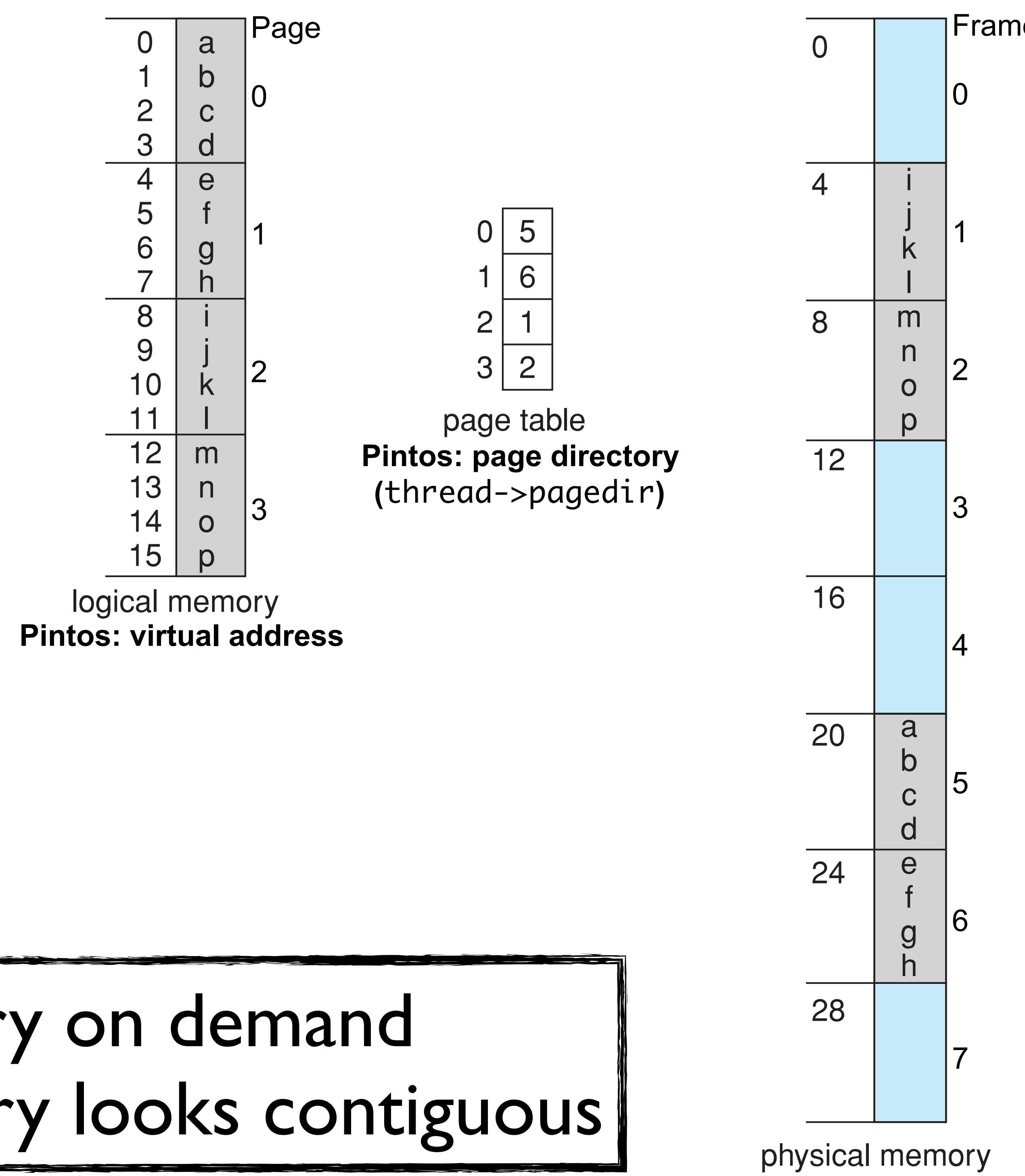
check if the pointer is above PHYS_BASE

to unmapped virtual memory

check if the pointer is in a valid *memory page*

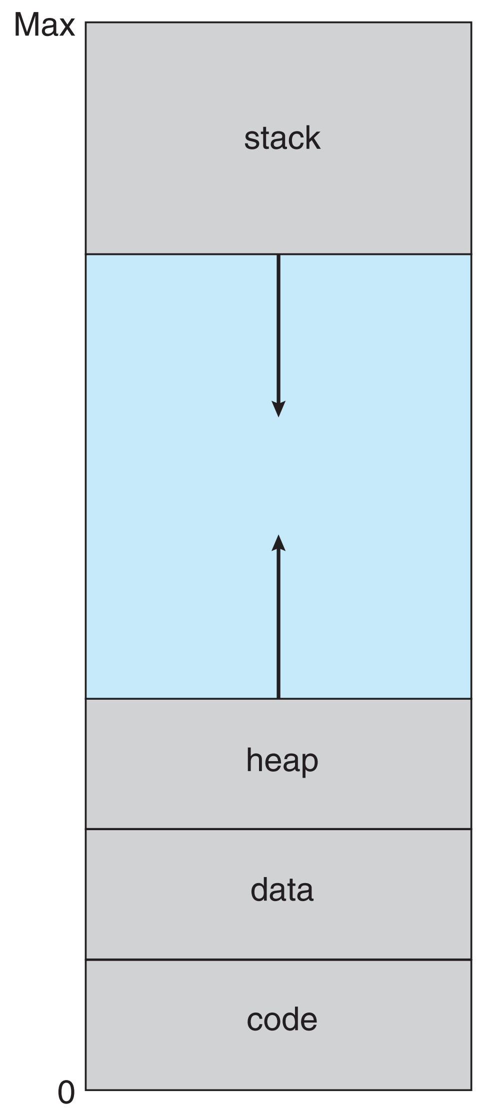
Memory paging

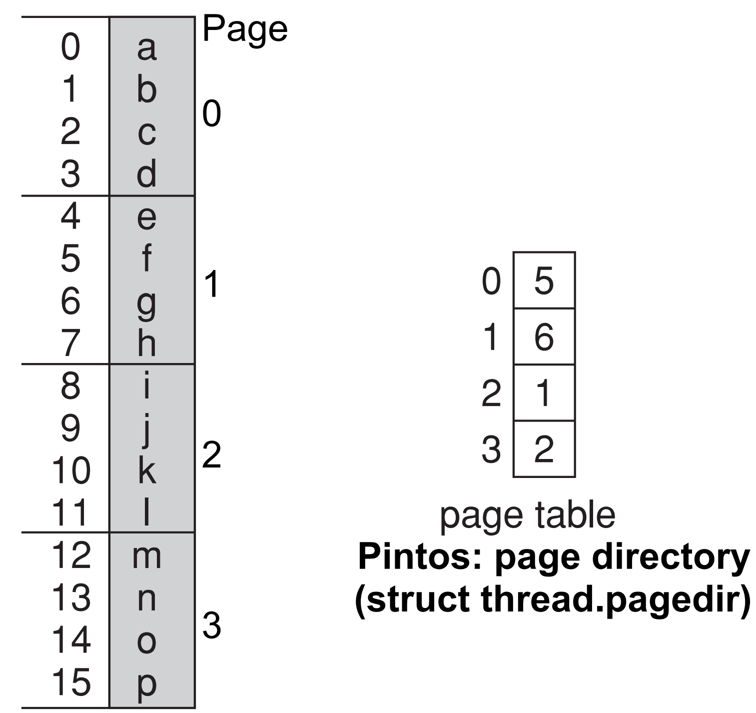




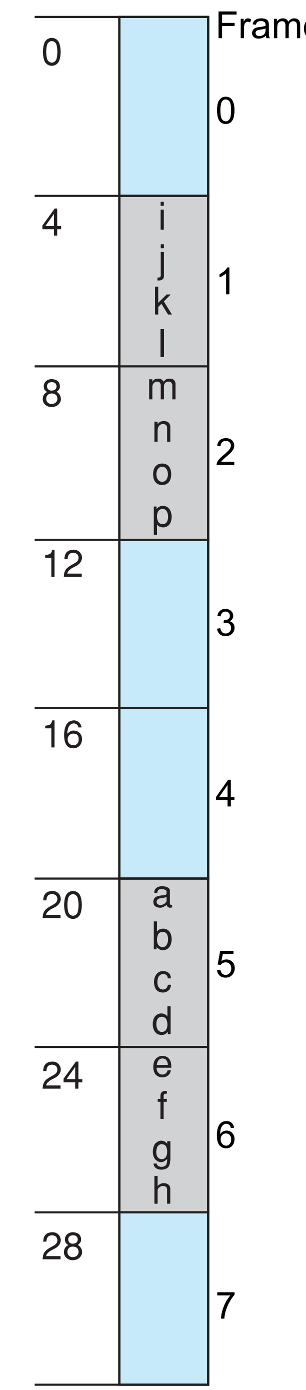
The process allocates memory on demand
Still, though, its virtual memory looks contiguous

Memory paging





logical memory
Pintos: virtual address



physical memory

check if the user pointer is in a valid memory page: hints in vaddr.h and pagedir.c

The process allocates memory on demand
Still, though, its virtual memory looks contiguous

Tests

- After implementing, you should pass:
  - exec-once
  - exec-arg
  - exec-multiple
  - exec-missing
  - exec-bad-ptr
  - wait-simple
  - wait-twice
  - wait-bad-pid

Readings

- Chapter 3:
  specially sections 3.1.4 and 3.3.4