CS202 (003): Operating Systems
Context switches, user-level threading

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

Last Time

Context Switches in WeensyOS

Interrupt occurs

Hardware
saves "trap frame", containing %rip, %rsp, %eflags (processor flags), etc. on the **kernel's stack** at a well-known place in **kernel memory**

Kernel (Assembly interrupt handler)
Moving the stack pointer into %rdi to prepare for the C `exception()` call

Kernel (`exception`)
"brute force structure copy" each field of trap frame from kernel stack to the process's PCB in kernel memory

Kernel (`interrupt handler`)
handling the interrupt

Kernel (`interrupt handler`)
decide to switch to another process

Kernel (`exception_return(&p>p_registers)`)
set %rsp to `&p>p_registers` (in %rdi)
pop the saved registers back into the CPU
add 15 to skip past saved codes
%rsp:"trap frame" when trap happens
issue `iretq`

Hardware (iretq)
restore the original %rip, %rsp, other saved state
return to user space

Done!

User-level Threading

Kernel-level Threading

Kernel directly manage the threads

Multiple threads within the same process share the same %cr3 value
(which points to a page table, meaning threads share the same memory space)

Kernel threads are always preemptive!

User-level Threading

Managed in the user space by a threading library — kernel only sees a single process

Threading package is responsible for:
Maintaining TCBs
Make a new stack for each new thread
Scheduling

User-level threading can be non-preemptive/preemptive

Context Switching for user-level threading

Cooperative multithreading
a context switch takes place only at well-defined points: when the thread calls yield() and when the thread would block on I/O.

Idea: signal as abstracting interrupt (hardware feature)
Hardware has interrupts
OS provides signals
Programs use signals

Remark: OS’s job is to give a user-space process the illusion that it's running on something like a machine, by creating abstractions

Preemptive multithreading
Not commonly used in production systems

How to arrange for package to get interrupted?
Signals!

Deliver a periodic timer interrupt or signal to a thread scheduler [setitimer()].
When it gets its interrupt, swap out the thread, run another one

Benefits: performance (maybe?)

Drawbacks: add complexity (without much gain)

Context Switching

Switching the view of memory (%cr3)

Switching the registers

Context Switching for user-level threading

Switching the registers

`swtch()` does the job

CS 202, Spring 2024
Handout 11 (Class 17)

1. User-level threads and swtch()

We'll study this in the context of user-level threads.

Per-thread state in thread control block:

```
typedef struct tcb {
    unsigned long saved_rsp;    /* Stack pointer of thread */
    char *t_stack;              /* Bottom of thread's stack */
    /* ... */
};
```

higher mem address because stack grows down

Machine-dependent thread initialization function:

```
void thread_init(tcb **t, void (*fn) (void *), void *arg);
```

Machine-dependent thread-switch function:

```
void swtch(tcb *current, tcb *next);
```

Implementation of swtch(current, next):

```
# gcc x86-64 calling convention:
# on entering swtch():
#  register %rdi holds first argument to the function ("current")
#  register %rsi holds second argument to the function ("next")

# Save call-preserved (aka "callee-saved") regs of 'current'
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15

# store old stack pointer, for when we swtch() back to "current" later
movq %rsp, (%rdi)                      # %rdi->saved_rsp = %rsp
movq (%rsi), %rsp                      # %rsp = %rsi->saved_rsp
```

save the current stack pointer (%rsp) to memory (%rdi, first args) where it can be retrieved later

loads the new thread stack pointer (%rsp) to the memory location %rsi points to (second args)

```
# Restore call-preserved (aka "callee-saved") regs of 'next'
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp

# Resume execution, from where "next" was when it last entered swtch()
ret
```

Context Switching for user-level threading

Switching the registers

`swtch()` does the job

`swtch()` is called by `yield()`

2. Example use of swtch(): the yield() call.

A thread is going about its business and decides that it's executed for long enough. So it calls yield(). Conceptually, the overall system needs to now choose another thread, and run it:

```
void yield() {

    tcb* next    = pick_next_thread();  /* get a runnable thread */
    tcb* current = get_current_thread();

    swtch(current, next);

    /* when 'current' is later rescheduled, it starts from here */
}
```

Used when a thread voluntarily gives up CPU

3. How do context switches interact with I/O calls?

This assumes a user-level threading package.

The thread calls something like "fake_blocking_read()". This looks to the _thread_ as though the call blocks, but in reality, the call is not blocking:

```
int fake_blocking_read(int fd, char* buf, int num) {

    int nread = -1;

    while (nread == -1) {

        /* this is a non-blocking read() syscall */
        nread = read(fd, buf, num);

        if (nread == -1 && errno == EAGAIN) {
            /*
             * read would block. so let another thread run
             * and try again later (next time through the
             * loop).
             */
            yield();
        }
    }

    return nread;
}
```

Context Switching for user-level threading

Switching the registers

`swtch()` does the job

`swtch()` is called by `yield()`

`yield()` is called by any thread that couldn't make further progress

mmap()

```
fd = open (pathname, mode)
write(fd, buf, sz)
read(fd, buf, sz)
```

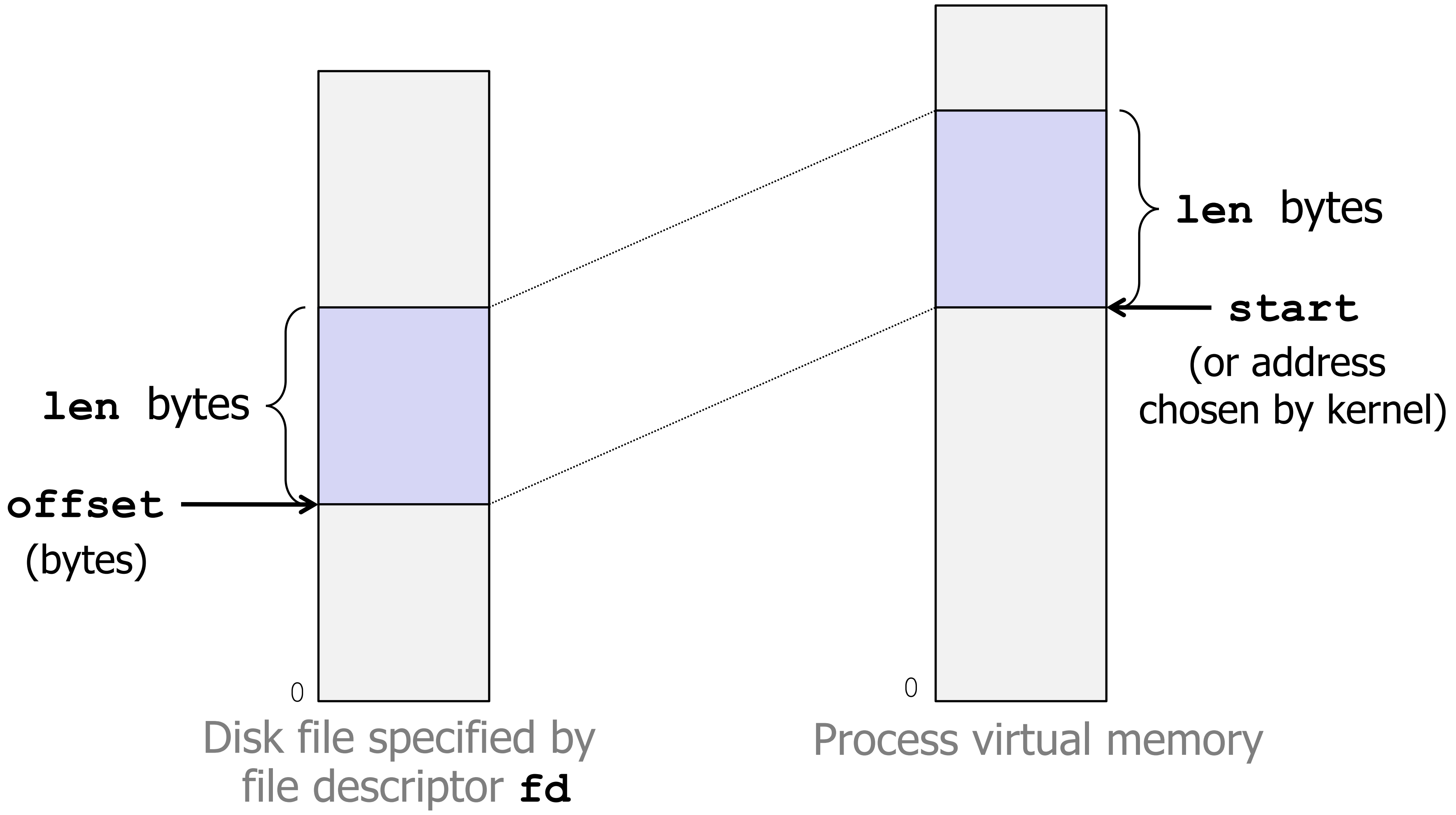
What is “fd”?

indexes into a table maintained by the kernel on behalf of the process

```
void* mmap(void* addr,
           size_t len, int prot,
           int flags, int fd,
           off_t offset)
```

map the specified open file (fd) into a region of my virtual memory (close to add, or at a kernel-selected place if addr is 0), and return pointer to it.

loads and stores to addr[x] are equivalent to reading and writing to the file at offset + x



Normal Memory:
Process Memory ↔ Swap File (default backing store)

With mmap():
Process Memory ↔ Your Specified File (custom backing store)

Example of mmap Usage

Copying a file to stdout without transferring data to user space	**next slide**
Reading big files	Let OS handle paging naturally, no need to manual chunk the file
Shared data structures (when flag is MAP_SHARED)	Shared memory lives in the same physical memory Useful for inter-process communication, shared caches, etc.
File-based data structures	Database
Question: how does the OS ensure that it's only writing back modified pages?	Dirty bit (it is set by the hardware when write occurs, OS only write back pages with dirty bit set)

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

void mmapcopy(int fd, int size);

int main(int argc, char **argv) {
  struct stat stat;
  int fd;

  /* Check for required cmd line arg */
  if (argc != 2) {
    printf("usage: %s <filename>\n", argv[0]);
    exit(0);
  }

  /* Copy input file to stdout */
  if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
    perror("open");

  fstat(fd, &stat);
```

get files size using fstat

```c
  mmapcopy(fd, stat.st_size);

  close(fd);

  return 0;
}

void mmapcopy(int fd, int size) {

    /* Ptr to memory mapped area */
    char *bufp;

    bufp = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);

    write(STDOUT_FILENO, bufp, size);
```

write mapped memory to stdout

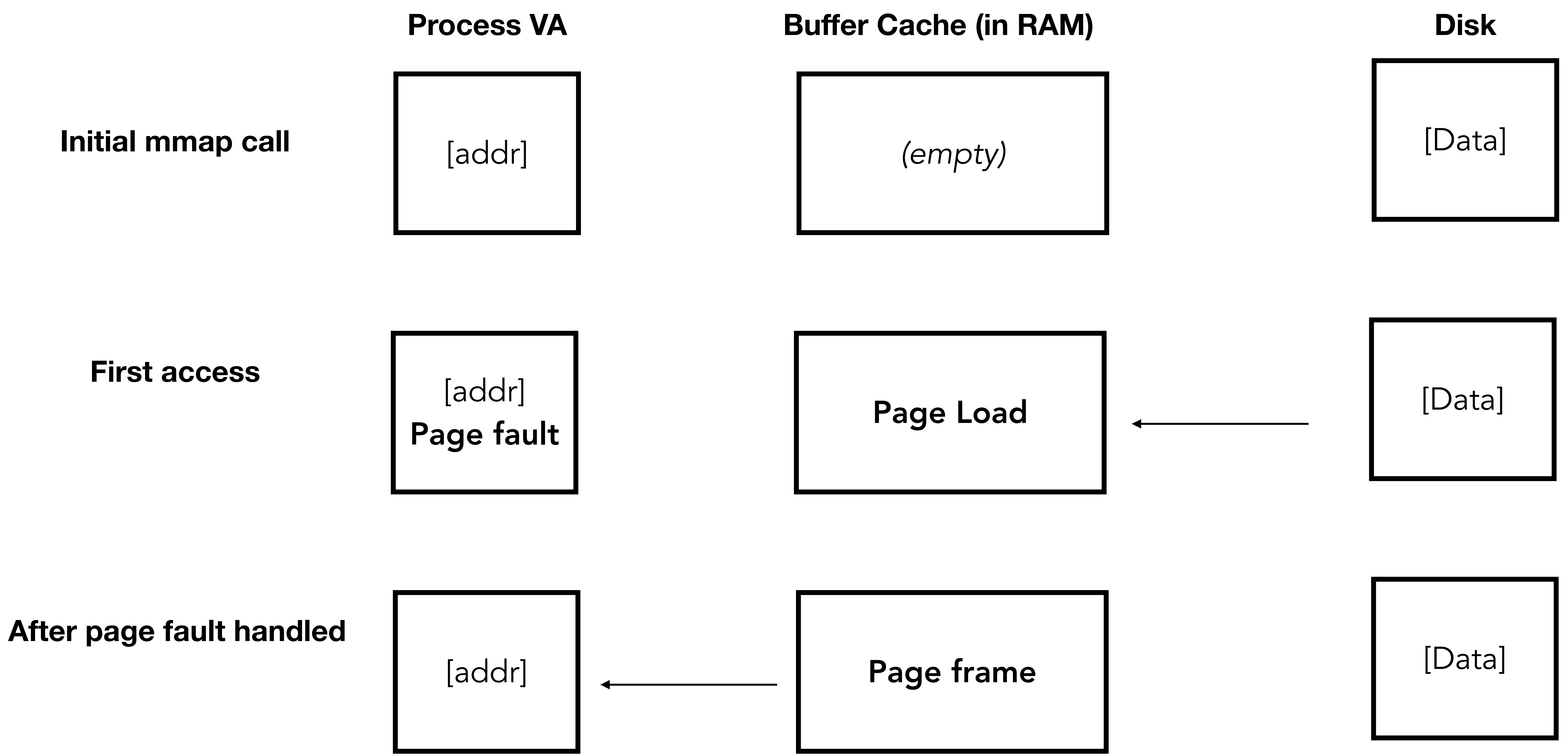
```c

    return;
}
```

Copying a file to stdout, the naive way:

```c
int rc;
char buf[256];
int fd = open(…);
while ((rc = read(fd, buf, sizeof(buf))!=-1) {
  write(1, buf, rc);
}
```

How does mmap work, internally?



What happens when buffer cache eviction?

```
struct reverse_mapping {
    physical_page_t *phys_page;
    struct mapping_entry {
        process_t *process;
        void *virtual_address;
        struct mapping_entry *next;
    } *mappings;
};
```

Physical Page 0x1000 is mapped by:
- Process A at VA 0x40000000
- Process B at VA 0x50000000

Reverse Mapping Entry:
PhysPage[0x1000] → [(A, 0x40000000), (B, 0x50000000)]

When evicting page 0x1000:
1. Look up reverse mapping
2. For each (process, VA) pair:
    1. Find process's page table
    2. Invalidate VA entry
    3. Send TLB shootdown if needed
3. Write page to disk if dirty
4. Free physical frame

Quiz on Thursday!