

CS202 (003): Operating Systems

Context switches, user-level threading

Instructor: Jocelyn Chen

mmap()

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

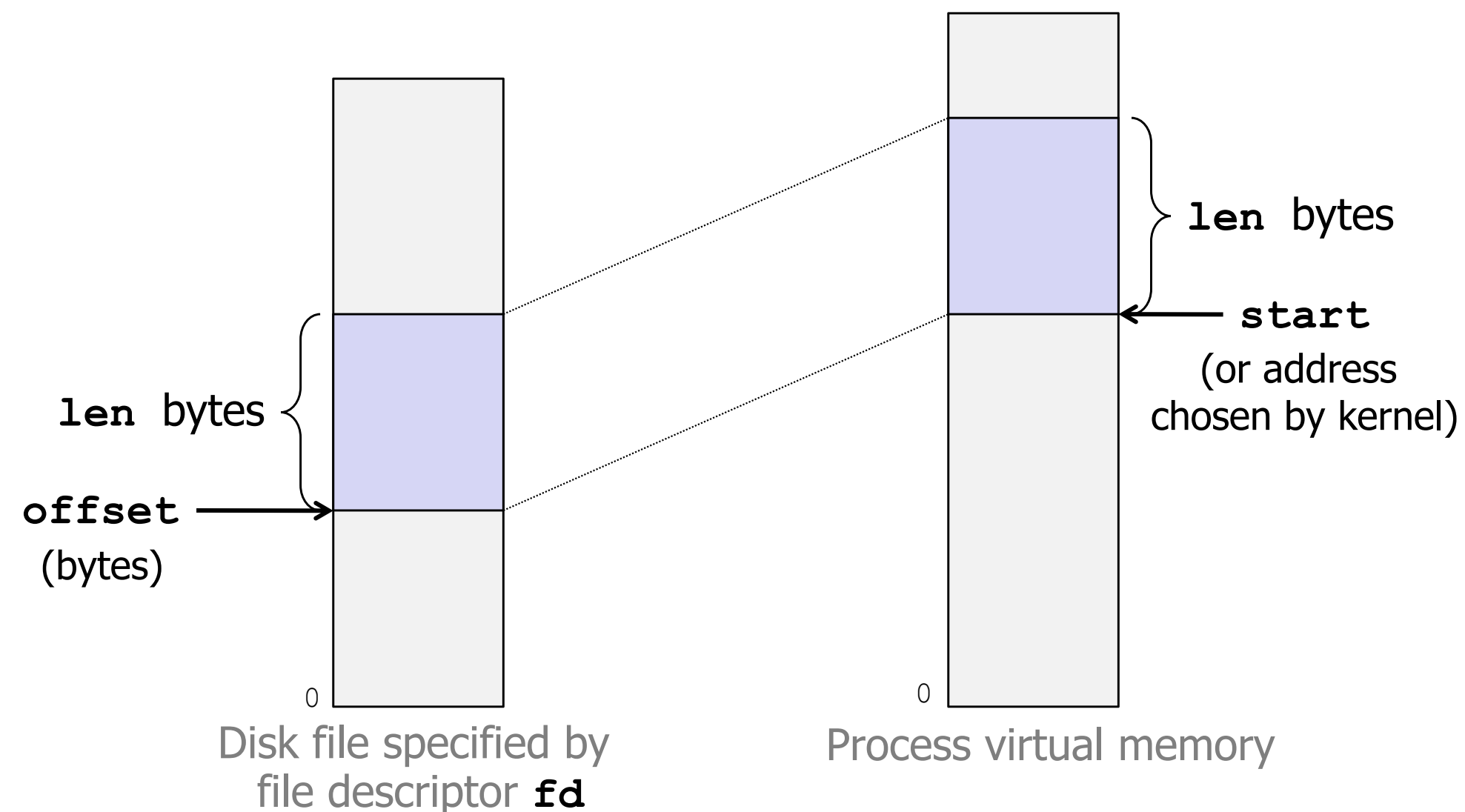
```
void* mmap(void* addr,
           size_t len, int prot,
           int flags, int fd,
           off_t offset)
```

What is "fd"?

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

map the specified open file (fd) into a region of my virtual memory (close to addr, 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

Reading big files

Shared data structures (when flag is MAP_SHARED)

File-based data structures

next slide

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5  #include <sys/stat.h>
6  #include <sys/types.h>
7  #include <unistd.h>
8
9  void mmapcopy(int fd, int size);
10
11 int main(int argc, char **argv) {
12     struct stat stat;
13     int fd;
14
15     /* Check for required cmd line arg */
16     if (argc != 2) {
17         printf("usage: %s <filename>\n", argv[0]);
18         exit(0);
19     }
20
21     /* Copy input file to stdout */
22     if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
23         perror("open");
24
25     fstat(fd, &stat);                get files size using fstat
26     mmapcopy(fd, stat.st_size);
27
28     close(fd);
29
30     return 0;
31 }
32
33 void mmapcopy(int fd, int size) {
34
35     /* Ptr to memory mapped area */
36     char *bufp;
37
38     bufp = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
39
40     write(STDOUT_FILENO, bufp, size); write mapped memory to stdout
41
42     return;
43 }
```

Copying a file to stdout, the naive way:

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

Example of mmap Usage

Copying a file to stdout without transferring data to user space

Reading big files

Shared data structures (when flag is MAP_SHARED)

File-based data structures

Question: how does the OS ensure that it's only writing back modified pages?

previous slide

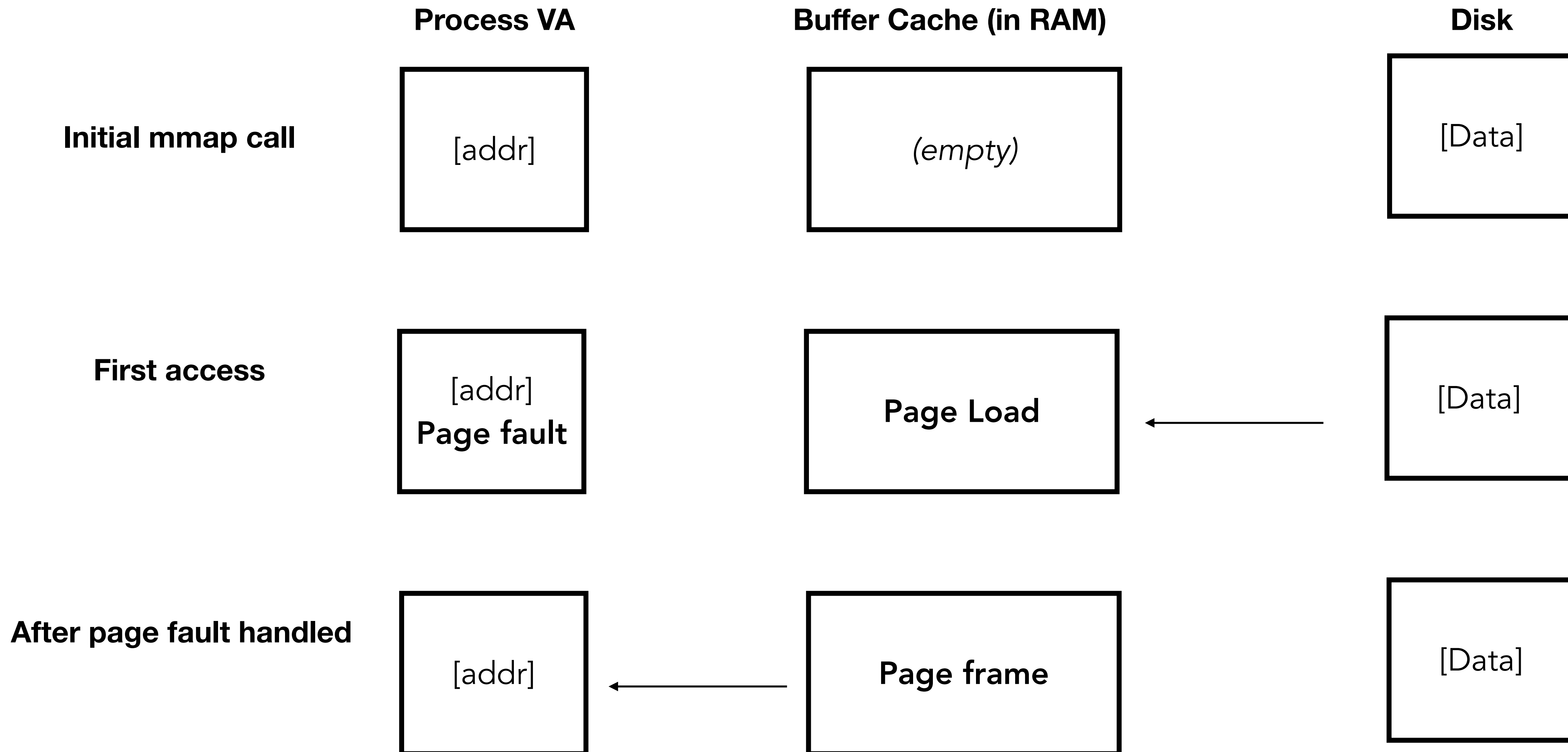
Let OS handle paging naturally, no need to manual chunk the file

Shared memory lives in the same physical memory
Useful for inter-process communication, shared caches, etc.

Database

Dirty bit
(it is set by the hardware when write occurs,
OS only write back pages with dirty bit set)

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

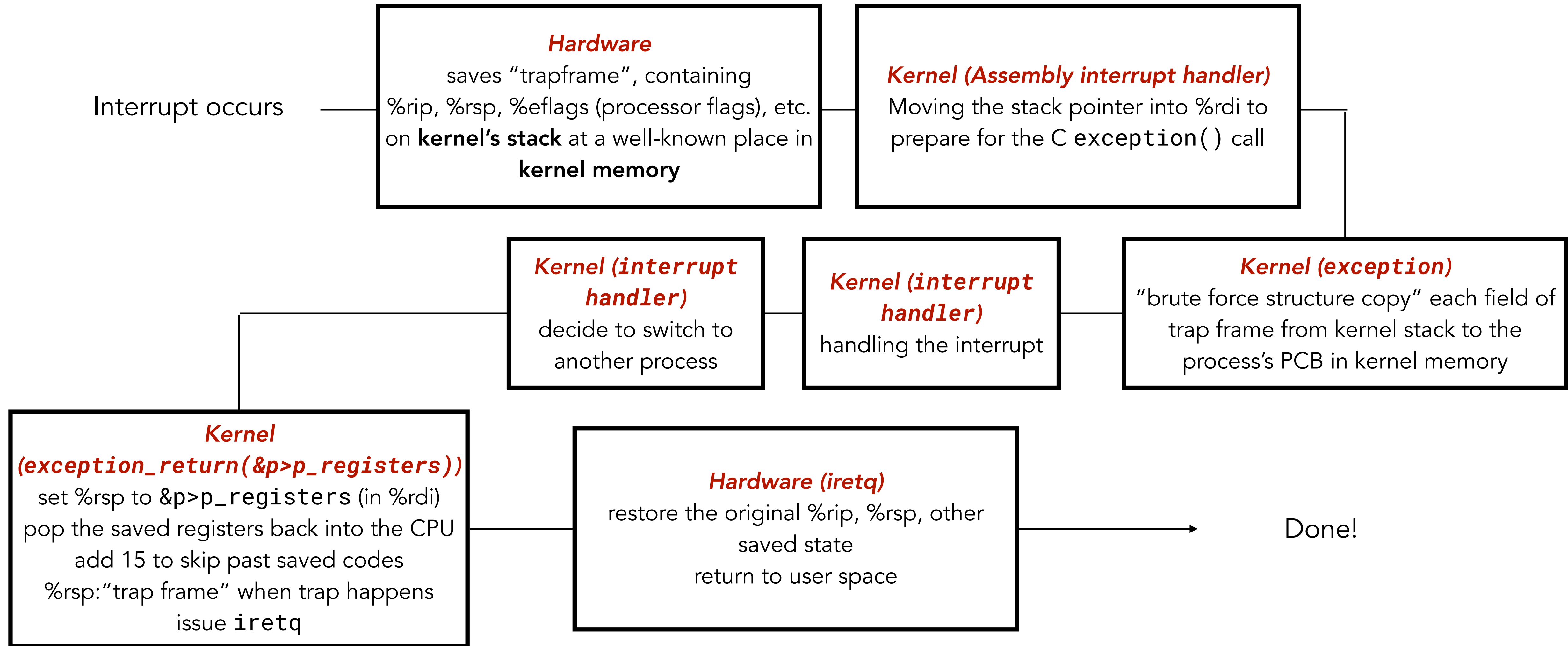
Reverse Mapping Entry:

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 shutdown if needed
3. Write page to disk if dirty
4. Free physical frame

Context Switches in WeensyOS



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

Switching the view of memory (%cr3)

Switching the registers

Context Switching for user-level threading

Switching the registers

swtch() does the job

```

1 CS 202, Spring 2024
2 Handout 11 (Class 17)
3
4 1. User-level threads and swtch()
5
6     We'll study this in the context of user-level threads.
7
8     Per-thread state in thread control block:
9
10        typedef struct tcb {
11            unsigned long saved_rsp;    /* Stack pointer of thread */
12            char *t_stack;              /* Bottom of thread's stack */
13            /* ... */
14        };
15
16     Machine-dependent thread initialization function:
17
18        void thread_init(tcb **t, void (*fn) (void *), void *arg);
19
20     Machine-dependent thread-switch function:
21
22        void swtch(tcb *current, tcb *next);
23
24     Implementation of swtch(current, next):
25
26        # gcc x86-64 calling convention:
27        # on entering swtch():
28        # register %rdi holds first argument to the function ("current")
29        # register %rsi holds second argument to the function ("next")
30
31        # Save call-preserved (aka "callee-saved") regs of 'current'
32        pushq %rbp
33        pushq %rbx
34        pushq %r12
35        pushq %r13
36        pushq %r14
37        pushq %r15
38
39        # store old stack pointer, for when we swtch() back to "current" later
40        movq %rsp, (%rdi)                # %rdi->saved_rsp = %rsp
41        movq (%rsi), %rsp                # %rsp = %rsi->saved_rsp
42
43        # Restore call-preserved (aka "callee-saved") regs of 'next'
44        popq %r15
45        popq %r14
46        popq %r13
47        popq %r12
48        popq %rbx
49        popq %rbp
50
51        # Resume execution, from where "next" was when it last entered swtch()
52        ret
53
54

```

higher mem address because stack grows down

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)

Context Switching for user-level threading

Switching the registers

swtch() does the job

swtch() is called by yield()

55
56 2. Example use of swtch(): the yield() call.
57
58 A thread is going about its business and decides that it's executed for
59 long enough. So it calls yield(). Conceptually, the overall system needs
60 to now choose another thread, and run it:

```
61  
62 void yield() {  
63  
64     tcb* next    = pick_next_thread(); /* get a runnable thread */  
65     tcb* current = get_current_thread();  
66  
67     swtch(current, next);  
68  
69     /* when 'current' is later rescheduled, it starts from here */  
70 }  
71
```

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

73
74 This assumes a user-level threading package.

75
76 The thread calls something like "fake_blocking_read()". This looks
77 to the _thread_ as though the call blocks, but in reality, the call
78 is not blocking:

```
79  
80 int fake_blocking_read(int fd, char* buf, int num) {  
81  
82     int nread = -1;  
83  
84     while (nread == -1) {  
85  
86         /* this is a non-blocking read() syscall */  
87         nread = read(fd, buf, num);  
88  
89         if (nread == -1 && errno == EAGAIN) {  
90             /*  
91              * read would block. so let another thread run  
92              * and try again later (next time through the  
93              * loop).  
94              */  
95             yield();  
96         }  
97     }  
98  
99     return nread;  
100 }  
101
```

Used when a thread voluntarily gives up CPU

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

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.

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)

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