



# SA\_RESTART

```
struct sigaction sa; ...  
sa.sa_flags = SA_RESTART;
```

general version:

```
sa.sa_flags = SA_NAME | SA_NAME | SA_NAME; (or 0)
```

if SA\_RESTART included:

after signal handler runs, attempt to restart interrupted operations (e.g. reading from keyboard)

if SA\_RESTART not included:

after signal handler runs, interrupted operations return typically an error (errno == EINTR)

# output of this?

pid 1000

```
void handle_sigusr1(int num) {
    write(1, "X", 1);
    kill(2000, SIGUSR1);
    _exit(0);
}

int main() {
    struct sigaction act;
    act.sa_handler = &handler_usr1;
    sigaction(SIGUSR1, &act, NULL);
    kill(1000, SIGUSR1);
}
```

pid 2000

```
void handle_sigusr1(int num) {
    write(1, "Y", 1);
    _exit(0);
}

int main() {
    struct sigaction act;
    act.sa_handler = &handler_usr1;
    sigaction(SIGUSR1, &act, NULL);
}
```

If these run at same time, expected output?

- A. XY
- B. X
- C. Y
- D. YX
- E. X or XY, depending on timing
- F. crash
- G. (nothing)
- H. something else

# output of this? (v2)

pid 1000

```
void handle_sigusr1(int num) {
    write(1, "X", 1);
    kill(2000, SIGUSR1);
    _exit(0);
}

int main() {
    struct sigaction act;
    act.sa_handler = &handler_usr1;
    sigaction(SIGUSR1, &act);
    kill(1000, SIGUSR1);
    while (1) pause();
}
```

pid 2000

```
void handle_sigusr1(int num) {
    write(1, "Y", 1);
    _exit(0);
}

int main() {
    struct sigaction act;
    act.sa_handler = &handler_usr1;
    sigaction(SIGUSR1, &act);
    while (1) pause();
}
```

If these run at same time, expected output?

- A. XY
- B. X
- C. Y
- D. YX
- E. X or XY, depending on timing
- F. crash
- G. (nothing)
- H. something else

# x86-64 Linux signal delivery (1)

suppose: signal happens while `foo()` is running

OS saves registers **to user stack**

OS modifies user registers, PC to call signal handler

the stack

address of <code>__restore_rt</code>
saved registers
PC when signal happened
local variables for <code>foo</code>
...

→ stack pointer  
when signal handler started

→ stack pointer  
before signal delivered

## x86-64 Linux signal delivery (2)

```
handle_sigint:
```

```
    ...  
    ret
```

```
    ...
```

```
__restore_rt:
```

```
    // 15 = "sigreturn" system call
```

```
    movq $15, %rax
```

```
    syscall
```

\_\_restore\_rt is **return address** for signal handler

sigreturn syscall restores pre-signal state

- if SA\_RESTART set, restarts interrupted operation

- also handles caller-saved registers

- also might change which signals blocked (depending how sigaction was called)

# signal handler unsafety (0)

```
void foo() {  
    /* SIGINT might happen while foo() is running */  
    char *p = malloc(1024);  
    ...  
}  
  
/* signal handler for SIGINT  
(registered elsewhere with sigaction()) */  
void handle_sigint() {  
    printf("You pressed control-C.\n");  
}
```

# signal handler unsafety (1)

```
void *malloc(size_t size) {  
    ...  
    to_return = next_to_return;  
    /* SIGNAL HAPPENS HERE */  
    next_to_return += size;  
    return to_return;  
}  
  
void foo() {  
    /* This malloc() call interrupted */  
    char *p = malloc(1024);  
    p[0] = 'x';  
}  
  
void handle_sigint() {  
    // printf might use malloc()  
    printf("You pressed control-C.\n");  
}
```



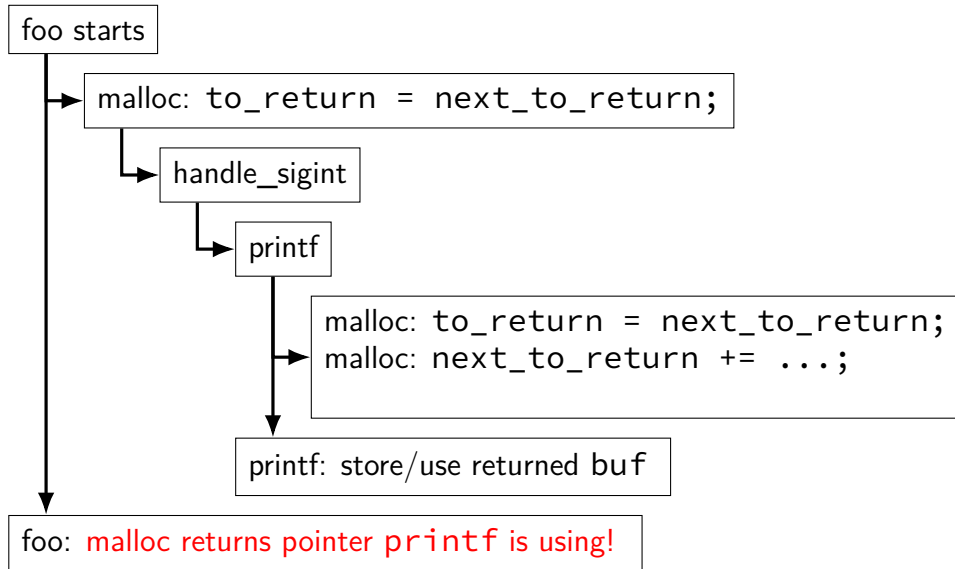
# signal handler unsafety (1)

```
void *malloc(size_t size) {  
    ...  
    to_return = next_to_return;  
    /* SIGNAL HAPPENS HERE */  
    next_to_return += size;  
    return to_return;  
}  
  
void foo() {  
    /* This malloc() call interrupted */  
    char *p = malloc(1024);  
    p[0] = 'x';  
}  
  
void handle_sigint() {  
    // printf might use malloc()  
    printf("You pressed control-C.\n");  
}
```

## signal handler unsafety (2)

```
void handle_sigint() {  
    printf("You pressed control-C.\n");  
}  
  
int printf(...) {  
    static char *buf;  
    ...  
    buf = malloc()  
    ...  
}
```

# signal handler unsafety: timeline



## signal handler unsafety (3)

```
foo() {
    char *p = malloc(1024)... {
        to_return = next_to_return;
        handle_sigint() { /* signal delivered here */
            printf("You pressed control-C.\n") {
                buf = malloc(...) {
                    to_return = next_to_return;
                    next_to_return += size;
                    return to_return;
                }
            }
            ...
        }
    }
    next_to_return += size;
    return to_return;
}
/* now p points to buf used by printf! */
}
```

## signal handler unsafety (3)

```
foo() {  
    char *p = malloc(1024)... {  
        to_return = next_to_return;  
        handle_sigint() { /* signal delivered here */  
            printf("You pressed control-C.\n") {  
                buf = malloc(...) {  
                    to_return = next_to_return;  
                    next_to_return += size;  
                    return to_return;  
                }  
                ...  
            }  
        }  
        next_to_return += size;  
        return to_return;  
    }  
    /* now p points to buf used by printf! */  
}
```

# signal handler safety

POSIX (standard that Linux follows) defines “async-signal-safe” functions

these must work correctly no matter what they interrupt

...and no matter how they are interrupted

includes: `write`, `_exit`

does not include: `printf`, `malloc`, `exit`

# blocking signals

avoid having signal handlers anywhere:

can instead **block signals**

can be done with `sigprocmask` or `pthread_sigmask`

signal will become “pending” instead

OS will not deliver unless unblocked

similar mechanism provided by CPU for interrupts (“disabling interrupts”)

# controlling when signals are handled

first, block a signal

then use system calls to inspect pending signals

example: `sigwait`

and/or unblock signals only at certain times

some special functions to help:

`sigsuspend` (unblock until handler runs),

`pselect` (unblock while checking for I/O), ...



# synchronous signal handling

```
int main(void) {  
    sigset_t set;  
    sigemptyset(&set);  
    sigaddset(&set, SIGINT);  
    sigprocmask(SIG_BLOCK, &set, NULL);  
  
    printf("Waiting for SIGINT (control-C)\n");  
    if (sigwait(&set, NULL) == SIGINT) {  
        printf("Got SIGINT\n");  
    }  
}
```

## opening a file?

```
open("/u/creiss/private.txt", O_RDONLY)
```

say, private file on portal

on Linux: makes *system call*

kernel needs to decide if this should work or not

# how does OS decide this?

argument: needs extra metadata

what would be wrong using...

system call arguments?

where the code calling open came from?

# authorization v authentication

*authentication* — who is who

# authorization v authentication

*authentication* — who is who

*authorization* — who can do what  
probably need authentication first...

# authentication

password

hardware token

...

# user IDs

most common way OSES identify what *domain* process belongs to:

(unspecified for now) procedure sets user IDs

every process has a user ID

user ID used to decide what process is authorized to do

# POSIX user IDs

`uid_t geteuid();` *// get current process's "effective" user ID*

process's user identified with unique number

kernel typically only knows about number

effective user ID is used for all permission checks

also some other user IDs — we'll talk later



# POSIX user IDs

`uid_t geteuid();` *// get current process's "effective" user ID*

process's user identified with unique number

kernel typically only knows about number

effective user ID is used for all permission checks

also some other user IDs — we'll talk later

standard programs/library maintain number to name mapping

`/etc/passwd` on typical single-user systems

network database on department machines

# POSIX groups

```
gid_t getegid(void);  
    // process's "effective" group ID
```

```
int getgroups(int size, gid_t list[]);  
    // process's extra group IDs
```

POSIX also has *group IDs*

like user IDs: kernel only knows numbers  
    standard library+databases for mapping to names

also process has some other group IDs — we'll talk later

# id

```
cr4bd@power4
: /net/zf14/cr4bd ; id
uid=858182(cr4bd) gid=21(csfaculty)
      groups=21(csfaculty),325(instructors),90027(cs4414)
```

id command displays uid, gid, group list

names looked up in database

- kernel doesn't know about this database
- code in the C standard library

# groups that don't correspond to users

example: video group for access to monitor

put process in video group when logged in directly

don't do it when SSH'd in

# groups that don't correspond to users

example: video group for access to monitor

put process in video group when logged in directly

don't do it when SSH'd in

...but: user can keep program running with video group  
in the background after logout?

# POSIX file permissions

POSIX files have a very restricted access control list

one user ID + read/write/execute bits for user

“owner” — also can change permissions

one group ID + read/write/execute bits for group

default setting — read/write/execute

on directories, ‘execute’ means ‘search’ instead

# permissions encoding

permissions encoded as 9-bit number, can write as octal: XYZ

octal divides into three 3-bit parts:

user permissions (X), group permissions (Y), other permission (Z)

each 3-bit part has a bit for 'read' (4), 'write' (2), 'execute' (1)

700 — user read+write+execute; group none; other none

451 — user read; group read+execute; other none

# chmod — exact permissions

```
chmod 700 file
```

```
chmod u=rwx,og= file
```

user read write execute; group/others no access

---

```
chmod 451 file
```

```
chmod u=r,g=rx,o= file
```

user read; group read/execute; others no access



# chmod — adjusting permissions

```
chmod u+rx foo
```

add user read and execute permissions

leave other settings unchanged

---

```
chmod o-rwx,u=rx foo
```

remove other read/write/execute permissions

set user permissions to read/execute

leave group settings unchanged

# POSIX/NTFS ACLs

more flexible access control lists

list of (user or group, read or write or execute or ...)

supported by NTFS (Windows)

a version standardized by POSIX, but usually not supported

# POSIX ACL syntax

```
# group students have read+execute permissions
group:students:r-x
# group faculty has read/write/execute permissions
group:faculty:rwX
# user mst3k has read/write/execute permissions
user:mst3k:rwX
# user tj1a has no permissions
user:tj1a:---

# POSIX acl rule:
    # user take precedence over group entries
```

# POSIX ACLs on command line

`getfacl file`

---

`setfacl -m 'user:tj1a:---' file`

add line to ACL

---

`setfacl -x 'user:tj1a' file`

REMOVE line from acl

---

`setfacl -M acl.txt file`

add to acl, but read what to add from a file

---

`setfacl -X acl.txt file`

remove from acl, but read what to remove from a file

# authorization checking on Unix

checked on system call entry

no relying on libraries, etc. to do checks

files (open, rename, ...) — file/directory permissions

processes (kill, ...) — process UID = user UID

...

# keeping permissions?

which of the following would still be secure?

- A. performing authorization checks in the standard library in addition to system call handlers
- B. performing authorization checks in the standard library instead of system call handlers
- C. making the user ID a system call argument rather than storing it persistently in the OS's memory

# superuser

user ID 0 is special

*superuser* or *root*

(non-Unix) or Administrator or SYSTEM or ...

some system calls: only work for uid 0

shutdown, mount new file systems, etc.

automatically passes all (or almost all) permission checks

# superuser v kernel mode

superuser : OS :: kernel mode : hardware

programs running as superuser still in user mode  
just change in how OS acts on system calls, etc.



# how does login work?

```
somemachine login: jo  
password: ****
```

```
jo@somemachine$ ls  
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

# how does login work?

```
somemachine login: jo  
password: ****
```

```
jo@somemachine$ ls  
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

# Unix password storage

typical single-user system: `/etc/shadow`

only readable by root/superuser

department machines: network service

Kerberos / Active Directory:

server takes (encrypted) passwords

server gives tokens: “yes, really this user”

can cryptographically verify tokens come from server

## aside: beyond passwords

/bin/login entirely user-space code

only thing special about it: when it's run

could use any criteria to decide, not just passwords

- physical tokens

- biometrics

- ...

# how does login work?

```
somemachine login: jo  
password: ****
```

```
jo@somemachine$ ls  
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

# changing user IDs

```
int setuid(uid_t uid);
```

if superuser: sets effective user ID to arbitrary value  
and a “real user ID” and a “saved set-user-ID” (we’ll talk later)

system starts in/login programs run as superuser  
voluntarily restrict own access before running shell, etc.

# sudo

```
tj1a@somemachine$ sudo restart  
Password: ****
```

sudo: run command with superuser permissions  
started by non-superuser

recall: inherits non-superuser UID

can't just call `setuid(0)`

# set-user-ID sudo

extra metadata bit on *executables*: set-user-ID

if set: `exec()` syscall changes effective user ID to owner's ID

sudo program: owned by root, marked set-user-ID

marking setuid: `chmod u+s`



# set-user ID gates

set-user ID program: gate to higher privilege

controlled access to extra functionality

make authorization/authentication decisions *outside the kernel*

way to allow normal users to do *one thing that needs privileges*

- write program that does that one thing — nothing else!

- make it owned by user that can do it (e.g. root)

- mark it set-user-ID

want to allow only some user to do the thing

- make program check which user ran it

# uses for setuid programs

## mount USB stick

- setuid program controls option to kernel mount syscall
- make sure user can't replace sensitive directories
- make sure user can't mess up filesystems on normal hard disks
- make sure user can't mount new setuid root files

## control access to device — printer, monitor, etc.

- setuid program talks to device + decides who can

## write to secure log file

- setuid program ensures that log is append-only for normal users

## bind to a particular port number $< 1024$

- setuid program creates socket, then becomes not root

# set-user-ID program v syscalls

hardware decision: some things only for kernel

system calls: *controlled* access to things kernel can do

decision about how can do it: in the kernel

kernel decision: some things only for root (or other user)

set-user-ID programs: controlled access to things root/... can do

decision about how can do it: made by root/...

# privilege escalation

*privilege escalation* — vulnerabilities that allow more privileges

code execution/corruption in utilities that run with high privilege

e.g. buffer overflow, command injection

login, sudo, system services, ...

bugs in system call implementations

logic errors in checking delegated operations

## a broken setuid program: setup

suppose I have a directory all-grades on shared server

in it I have a folder for each assignment

and within that a text file for each user's grade + other info

say I don't have flexible ACLs and want to give each user access

## a broken setuid program: setup

suppose I have a directory all-grades on shared server

in it I have a folder for each assignment

and within that a text file for each user's grade + other info

say I don't have flexible ACLs and want to give each user access

one (bad?) idea: setuid program to read grade for assignment

```
./print_grade assignment
```

outputs grade from all-grades/assignment/USER.txt

# a very broken setuid program

print\_grade.c:

```
int main(int argc, char **argv) {
    char filename[500];
    sprintf(filename, "all-grades/%s/%s.txt",
            argv[1], getenv("USER"));
    int fd = open(filename, O_RDWR);
    char buffer[1024];
    read(fd, buffer, 1024);
    printf("%s: %s\n", argv[1], buffer);
}
```

HUGE amount of stuff can go wrong

examples?

# set-user ID programs are very hard to write

what if stdin, stdout, stderr start closed?

what if signals setup weirdly?

what if the PATH env. var. set to directory of malicious programs?

what if `argc == 0`?

what if dynamic linker env. vars are set?

what if some bug allows memory corruption?

...



## other privileged escalation issues

sudo problem: trusted code that's supposed to enforce restriction can be fooled into not really enforcing it

also can occur in other contexts:

system call letting program access things it shouldn't?

browser letting web page javascript access things it shouldn't?

web application giving users access to files they shouldn't have?

mobile phone OS allowing location access without location permission?

...

## some security tasks (1)

helping students collaborate in ad-hoc small groups on shared server?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

## some security tasks (2)

letting students assignment files to faculty on shared server?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

## some security tasks (3)

running untrusted game program from Internet?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

**backup slides**

## another very broken setuid program (setup)

allow users to print files, but only if less than 1KB

## another very broken setuid program

print\_short\_file.c:

```
int main(int argc, char **argv) {
    struct stat st;
    if (stat(argv[1], &st) == -1) abort();
    // make sure argv[1] is owned by user running this
    if (st.st_uid != getuid()) abort();
    // and that it's less than 1 KB
    if (st.st_size >= 1024) abort();
    char command[1024];
    sprintf(command, "print %1000s", argv[1]);
    system(command);
    return EXIT_SUCCESS;
}
```

## a delegation problem

consider printing program marked `setuid` to access printer

- decision: no accessing printer directly

- printing program enforces page limits, etc.

command line: file to print

can printing program just call `open()`?



## a broken solution

```
if (original user can read file from argument) {  
    open(file from argument);  
    read contents of file;  
    write contents of file to printer  
    close(file from argument);  
}
```

hope: this prevents users from printing files than can't read

problem: race condition!

## a broken solution / why

setuid program

check: can user access? (yes)

open("toprint.txt")

read ...

other user program

create normal file toprint.txt

—

unlink("toprint.txt")

link("/secret", "toprint.txt")

—

—

link: create new directory entry for file

another option: rename, symlink ("symbolic link" — alias for file/directory)

another possibility: run a program that creates secret file (e.g. temporary file used by password-changing program)

time-to-check-to-time-of-use vulnerability

# TOCTTOU solution

temporarily 'become' original user

then open

then turn back into set-uid user

this is why POSIX processes have multiple user IDs

can swap out effective user ID temporarily

# practical TOCTTOU races?

can use symlinks *maze* to make check slower

symlink toprint.txt → a/b/c/d/e/f/g/normal.txt

symlink a/b → ../a

symlink a/c → ../a

...

lots of time spent following symbolic links when program opening toprint.txt

gives more time to sneak in unlink/link or (more likely) rename

## exercise

which (if any) of the following would fix for a TOCTTOU vulnerability in our setuid printing application? (assume the Unix-permissions without ACLs are in use)

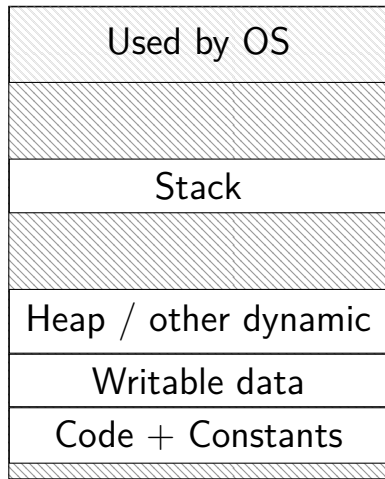
[A] **both before and after** opening the path passed in for reading, check that the path is accessible to the user who ran our application

[B] after opening the path passed in for reading, using `fstat` with the file descriptor opened to check the permissions on the file

[C] before opening the path, verify that the user controls the file referred to by the path **and** the directory containing it



# program memory



0xFFFF FFFF FFFF FFFF

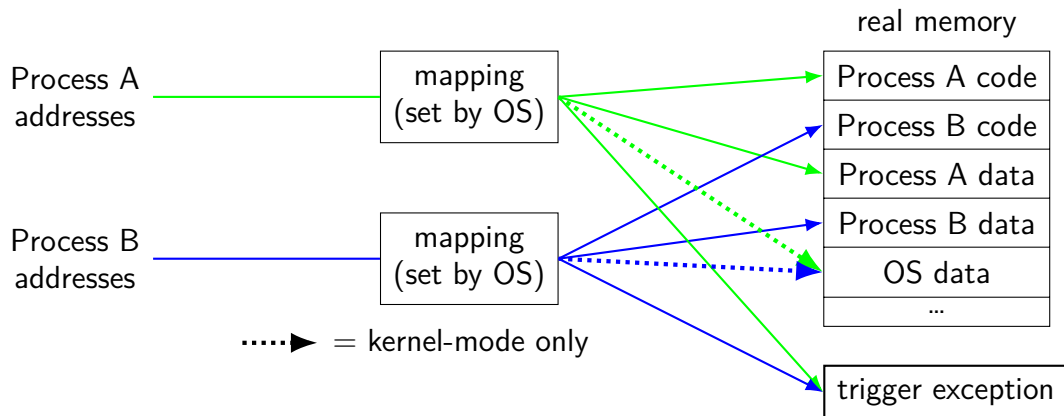
0xFFFF 8000 0000 0000

0x7F...

0x0000 0000 0040 0000

# address spaces

illusion of **dedicated memory**

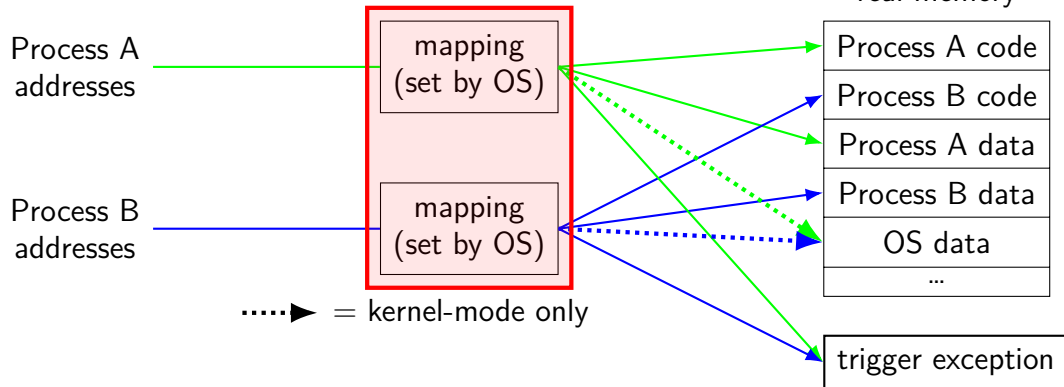




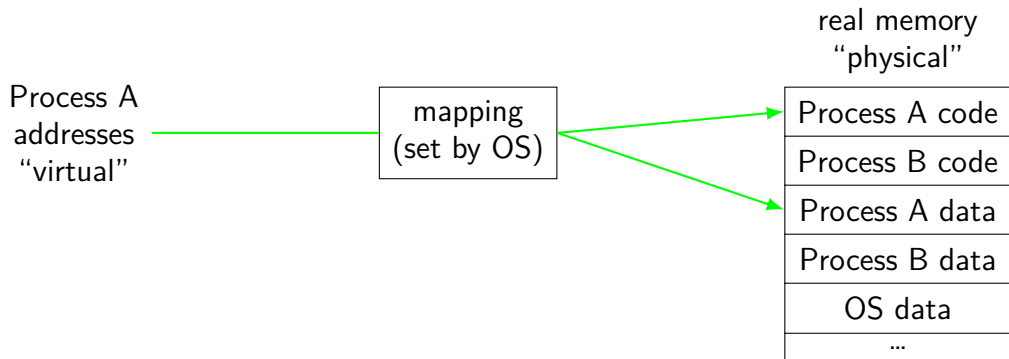
# address spaces

illusion of **dedicated memory**

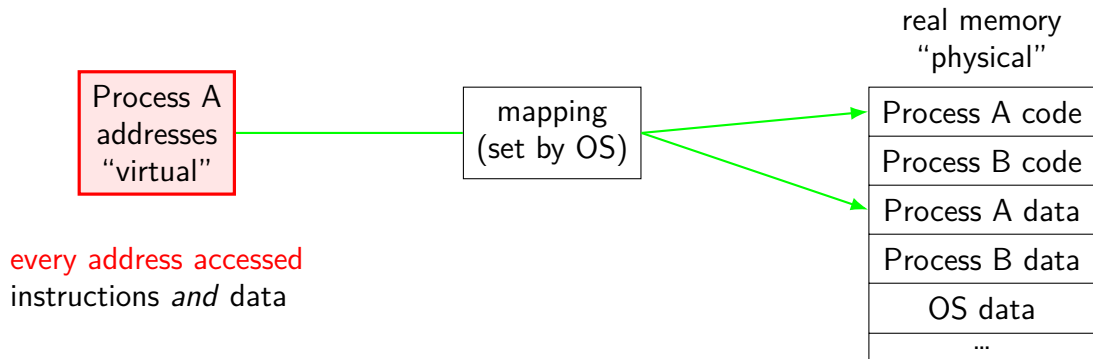
chose one during context switch



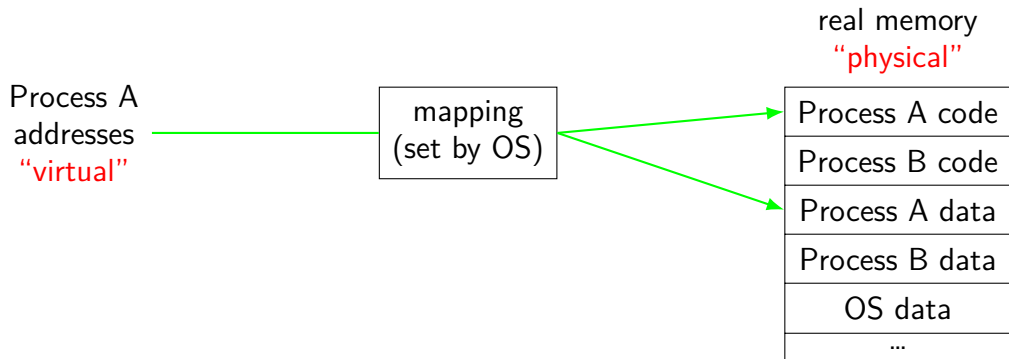
# address translation



# address translation

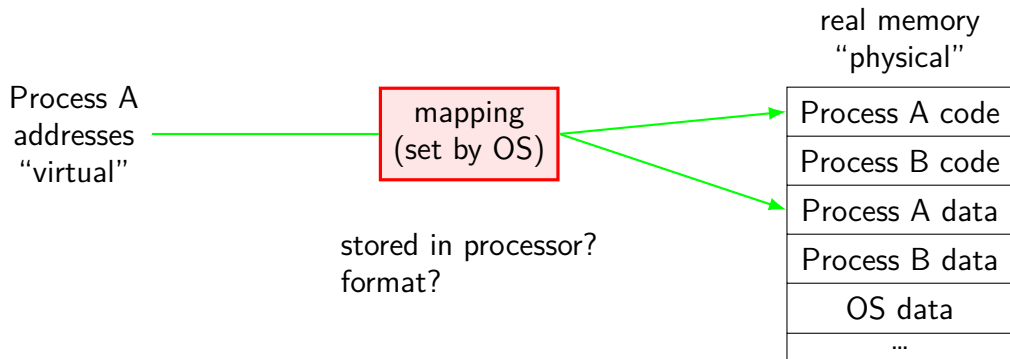


# address translation

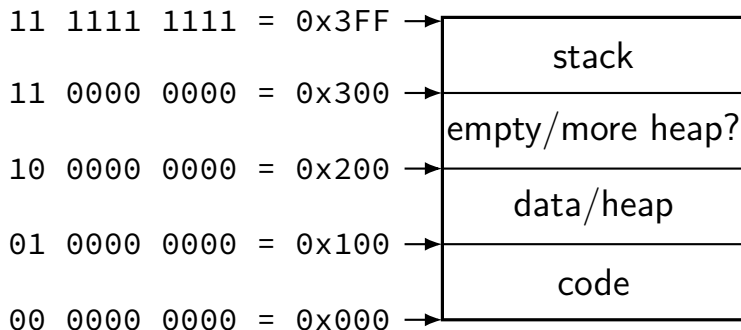


program addresses are 'virtual'  
real addresses are 'physical'  
can be different sizes!

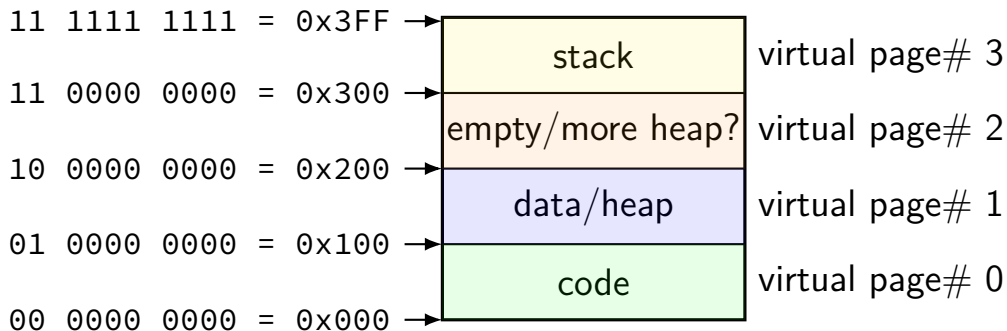
# address translation



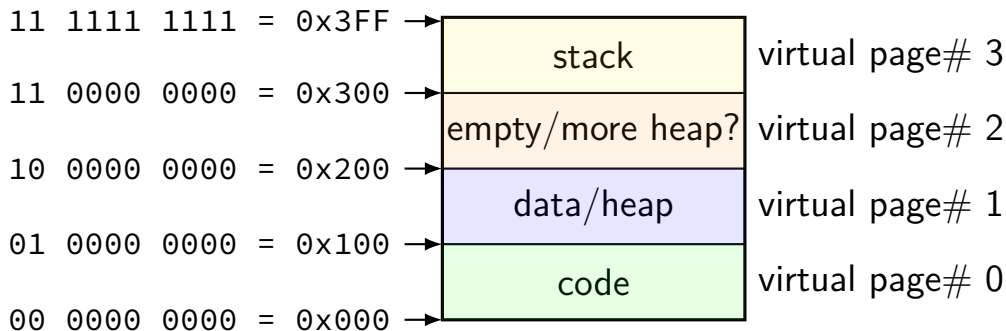
# toy program memory



## toy program memory



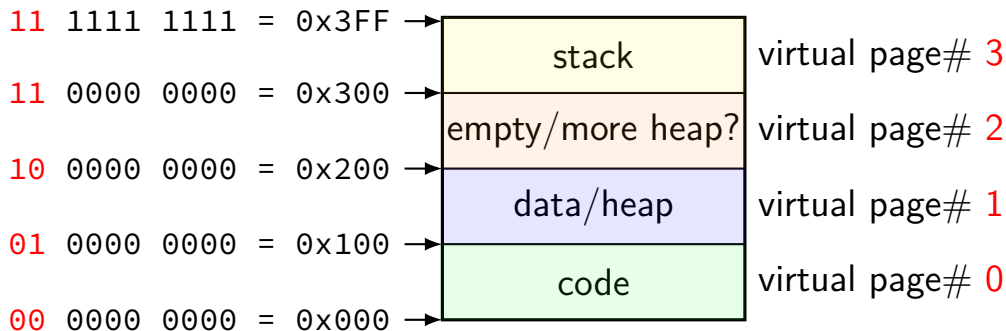
## toy program memory



divide memory into **pages** ( $2^8$  bytes in this case)  
“virtual” = addresses the program sees

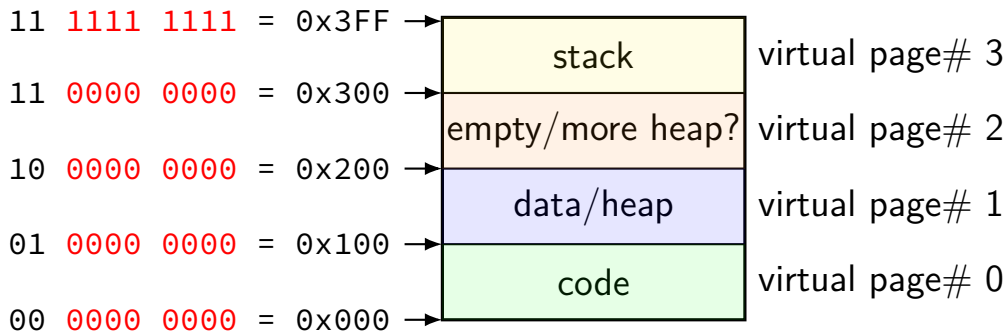


# toy program memory



page number is upper bits of address  
(because page size is power of two)

# toy program memory



rest of address is called **page offset**

# toy physical memory

program memory  
virtual addresses

11 0000 0000 to 11 1111 1111
10 0000 0000 to 10 1111 1111
01 0000 0000 to 01 1111 1111
00 0000 0000 to 00 1111 1111

real memory  
physical addresses

111 0000 0000 to 111 1111 1111
001 0000 0000 to 001 1111 1111
000 0000 0000 to 000 1111 1111

# toy physical memory

program memory  
virtual addresses

11 0000 0000 to
11 1111 1111
10 0000 0000 to
10 1111 1111
01 0000 0000 to
01 1111 1111
00 0000 0000 to
00 1111 1111

real memory  
physical addresses

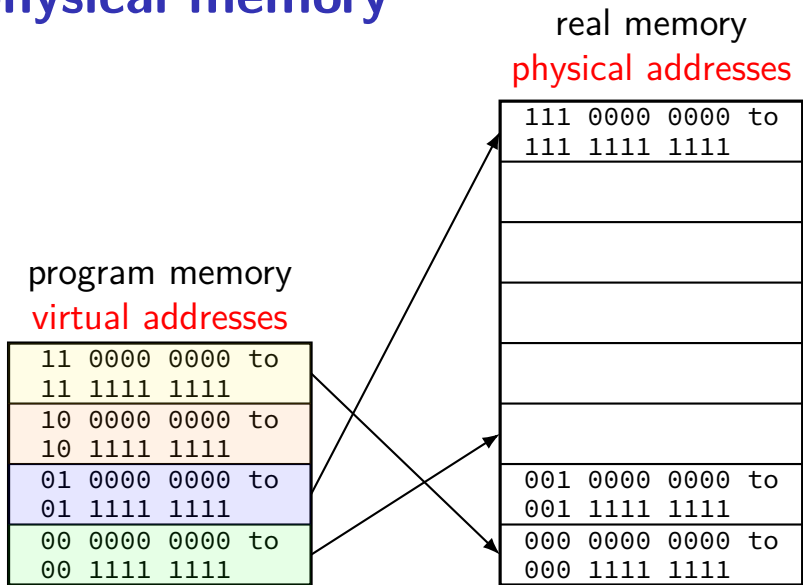
111 0000 0000 to
111 1111 1111
001 0000 0000 to
001 1111 1111
000 0000 0000 to
000 1111 1111

physical page 7

physical page 1

physical page 0

# toy physical memory



# toy physical memory

virtual page #      physical page #

00	010 (2)
01	111 (7)
10	<i>none</i>
11	000 (0)

program memory

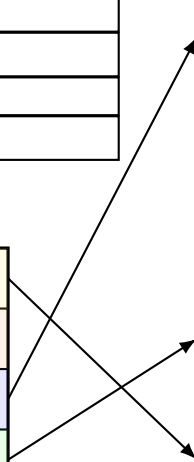
virtual addresses

11 0000 0000 to 11 1111 1111
10 0000 0000 to 10 1111 1111
01 0000 0000 to 01 1111 1111
00 0000 0000 to 00 1111 1111

real memory

physical addresses

111 0000 0000 to 111 1111 1111
001 0000 0000 to 001 1111 1111
000 0000 0000 to 000 1111 1111



# toy physical memory

virtual page #	physical page #
00	010 (2)
01	111 (7)
10	<i>none</i>
11	000 (0)

program memory

virtual addresses

11 0000 0000 to 11 1111 1111
10 0000 0000 to 10 1111 1111
01 0000 0000 to 01 1111 1111
00 0000 0000 to 00 1111 1111

page  
table! real memory  
physical addresses

111 0000 0000 to 111 1111 1111
001 0000 0000 to 001 1111 1111
000 0000 0000 to 000 1111 1111

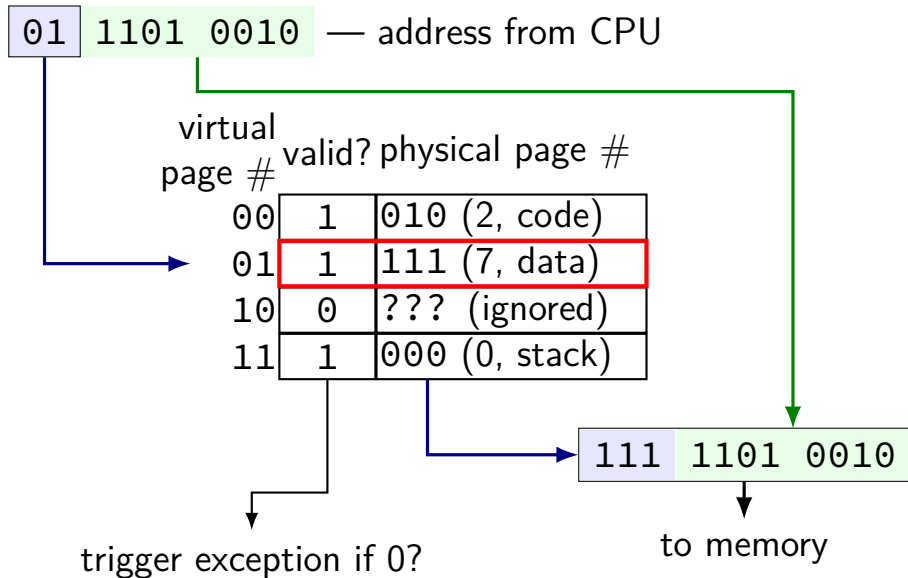
# toy page table lookup

virtual  
page # valid? physical page #

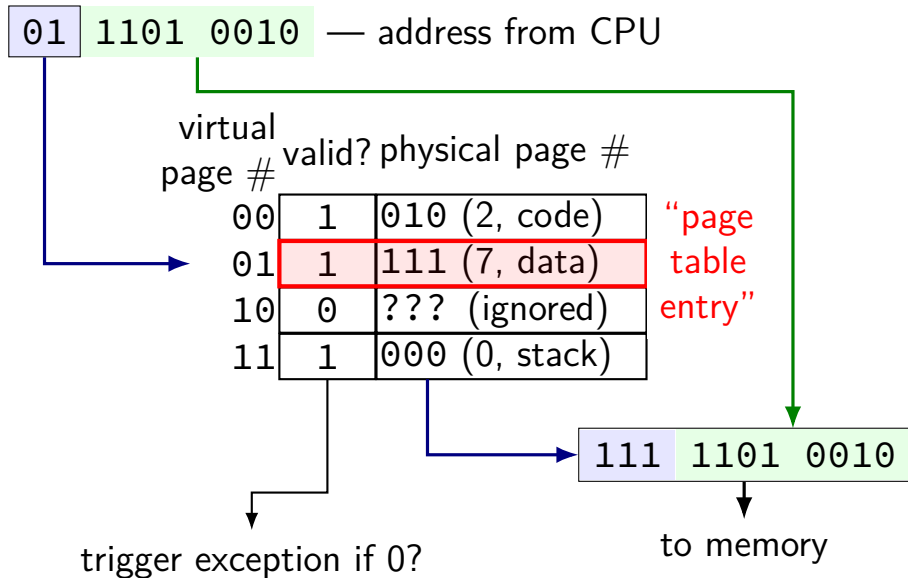
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)



# toy page table lookup



# toy page table lookup



# t “virtual page number” |lookup

01 1101 0010 — address from CPU

virtual  
page # valid? physical page #

00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

trigger exception if 0?

to memory

111 1101 0010

# toy page table lookup

01 1101 0010 — address from CPU

virtual  
page # valid? physical page #

00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

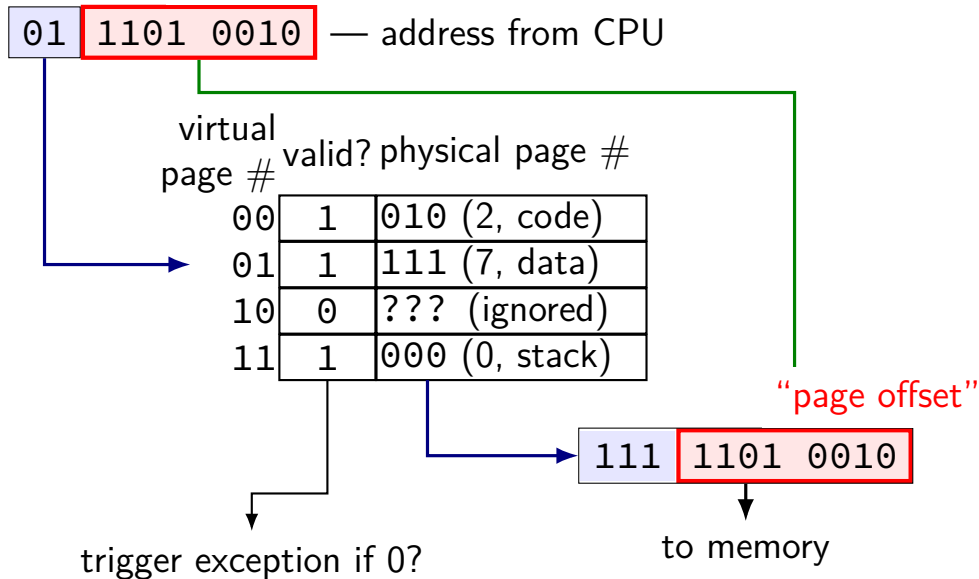
“physical page number”

111 1101 0010

trigger exception if 0?

to memory

# toy pag “page offset” lookup



# switching page tables

part of context switch is changing the page table

extra privileged instructions

# switching page tables

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

# switching page tables

part of context switch is changing the page table

extra **privileged instructions**

where in memory is the code that does this switching?

- probably have a page table entry pointing to it
- hopefully marked kernel-mode-only



# switching page tables

part of context switch is changing the page table

extra **privileged instructions**

where in memory is the code that does this switching?

- probably have a page table entry pointing to it
- hopefully marked kernel-mode-only

code better not be modified by user program

- otherwise: uncontrolled way to “escape” user mode

## on virtual address sizes

virtual address size = size of pointer?

often, but — sometimes part of pointer not used

example: typical x86-64 only use 48 bits

rest of bits have fixed value

virtual address size is amount used for mapping

# address space sizes

amount of stuff that can be addressed = address space size  
based on number of unique addresses

e.g. 32-bit virtual address =  $2^{32}$  byte virtual address space

e.g. 20-bit physical addressss =  $2^{20}$  byte physical address space

# address space sizes

amount of stuff that can be addressed = address space size  
based on number of unique addresses

e.g. 32-bit virtual address =  $2^{32}$  byte virtual address space

e.g. 20-bit physical addressss =  $2^{20}$  byte physical address space

what if my machine has 3GB of memory (not power of two)?

not all addresses in physical address space are useful

most common situation (since CPUs support having a lot of memory)

## exercise: page counting

suppose 32-bit virtual (program) addresses

and each page is 4096 bytes ( $2^{12}$  bytes)

how many virtual pages?

## exercise: page counting

suppose 32-bit virtual (program) addresses

and each page is 4096 bytes ( $2^{12}$  bytes)

how many virtual pages?

## exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

each page is 4096 bytes ( $2^{12}$  bytes)

page table entries have physical page #, valid bit, bit

how big is the page table (if laid out like ones we've seen)?

## exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

each page is 4096 bytes ( $2^{12}$  bytes)

page table entries have physical page #, valid bit, bit

how big is the page table (if laid out like ones we've seen)?

issue: where can we store that?



## exercise: address splitting

and each page is 4096 bytes ( $2^{12}$  bytes)

split the address 0x12345678 into page number and page offset:

## exercise: address splitting

and each page is 4096 bytes ( $2^{12}$  bytes)

split the address 0x12345678 into page number and page offset:

# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)
----------------	-----------------------------	------------------------------------

# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)
----------------	-----------------------------	------------------------------------

page table  
base register

0x00010000

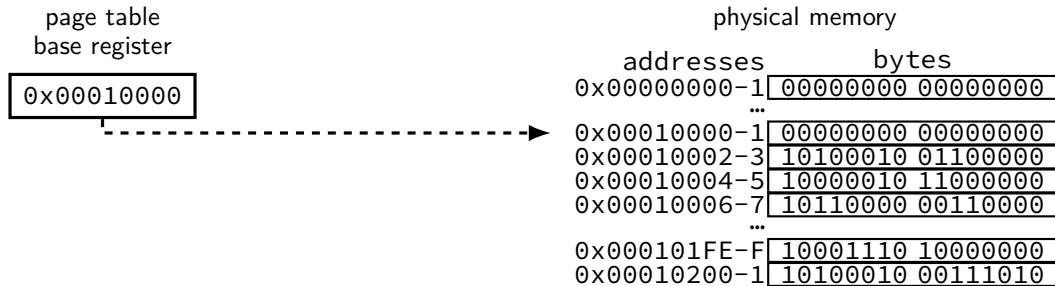


# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)
----------------	-----------------------------	------------------------------------

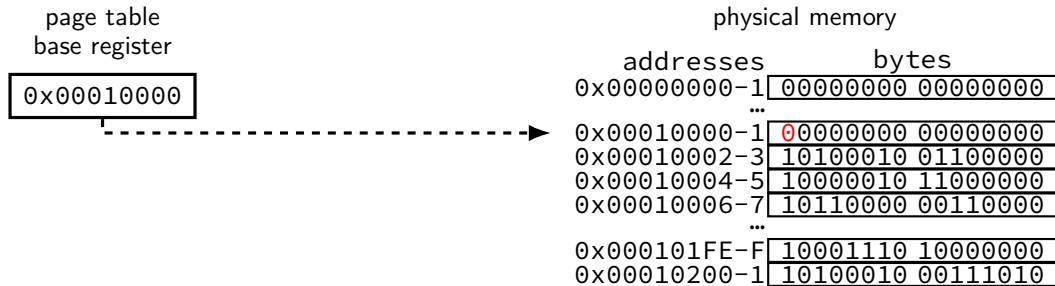


# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)
----------------	-----------------------------	------------------------------------

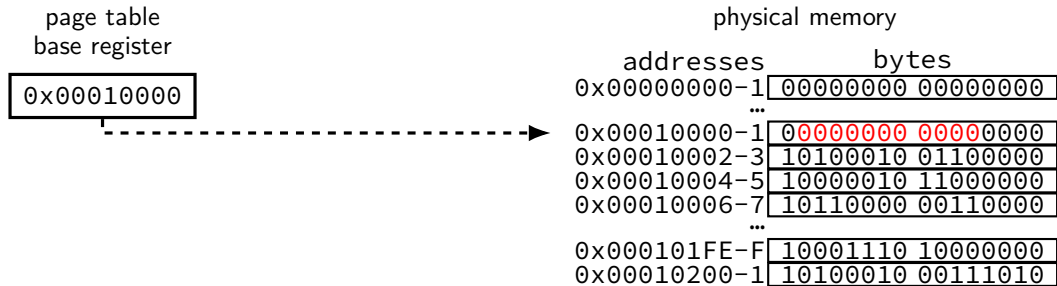


# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)
----------------	-----------------------------	------------------------------------

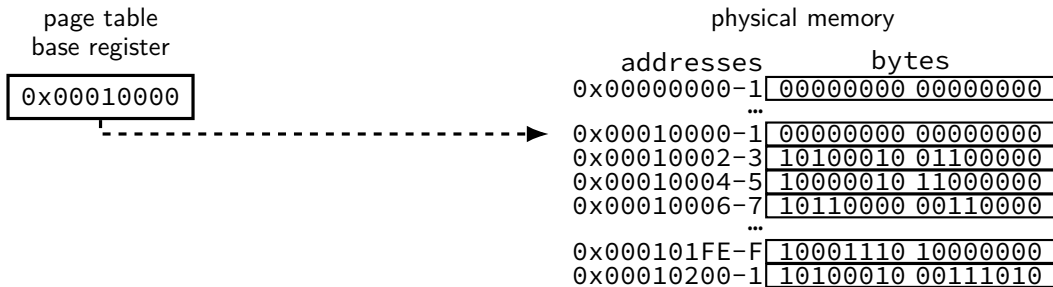


# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)
----------------	-----------------------------	------------------------------------



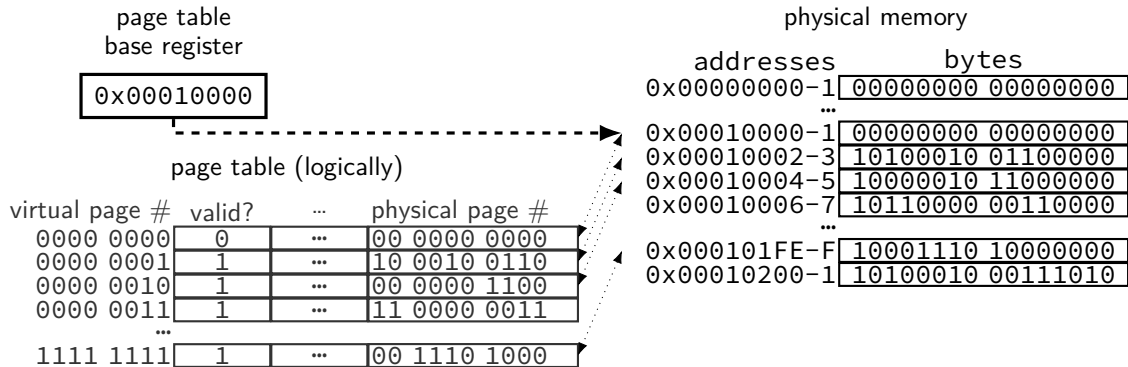


# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)
----------------	-----------------------------	------------------------------------

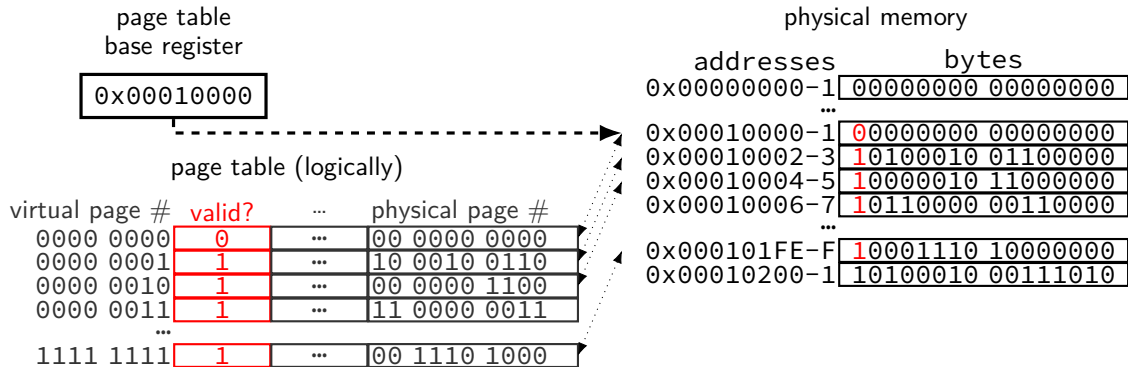


# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)
----------------	-----------------------------	------------------------------------

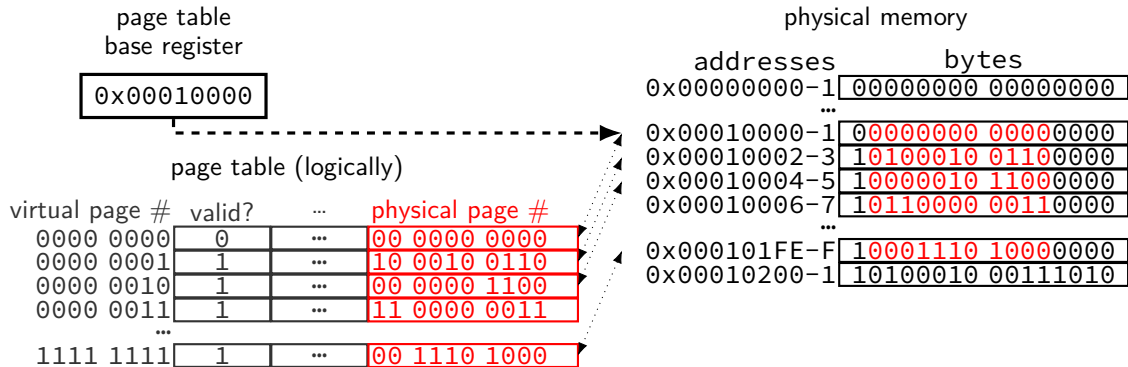


# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)
----------------	-----------------------------	------------------------------------

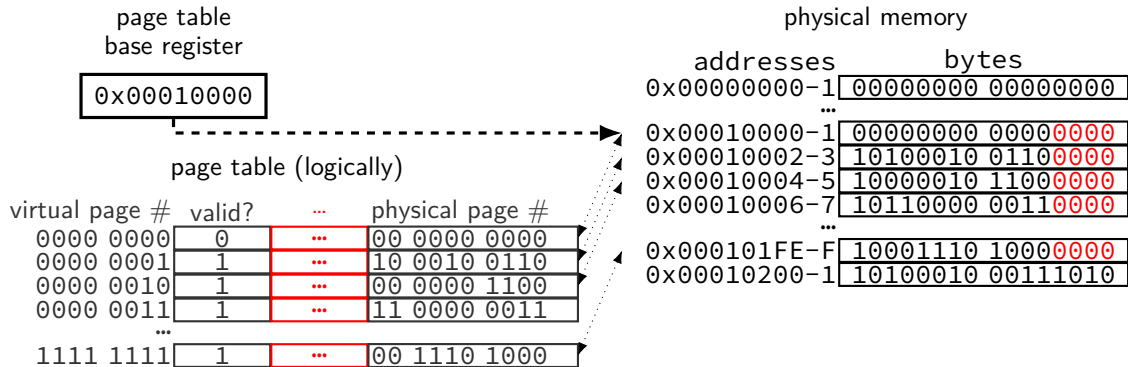


# page tables in memory

where can processor store megabytes of page tables? **in memory**

page table entry layout (chosen by processor)

valid (bit 15)	physical page # (bits 4–14)	other bits and/or unused (bit 0-3)
----------------	-----------------------------	------------------------------------

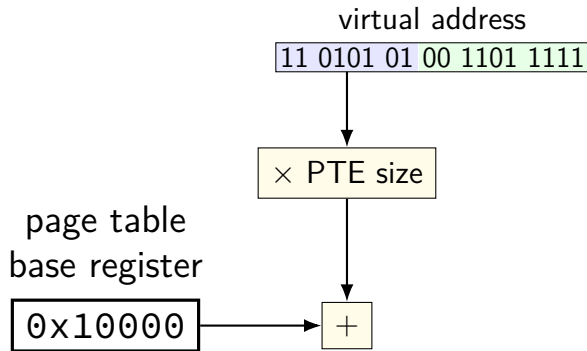


# memory access with page table

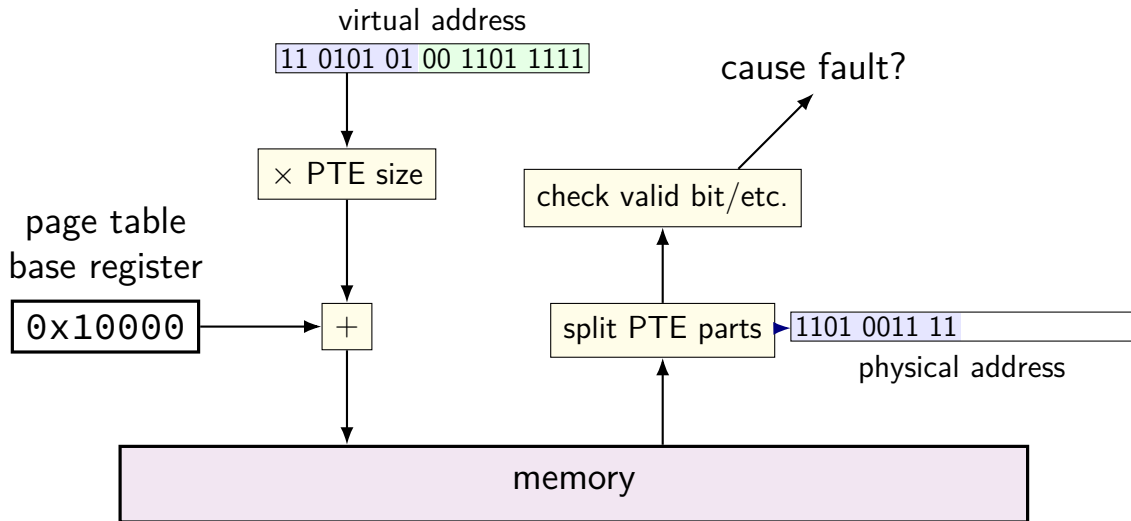
virtual address

11	0101	01	00	1101	1111
----	------	----	----	------	------

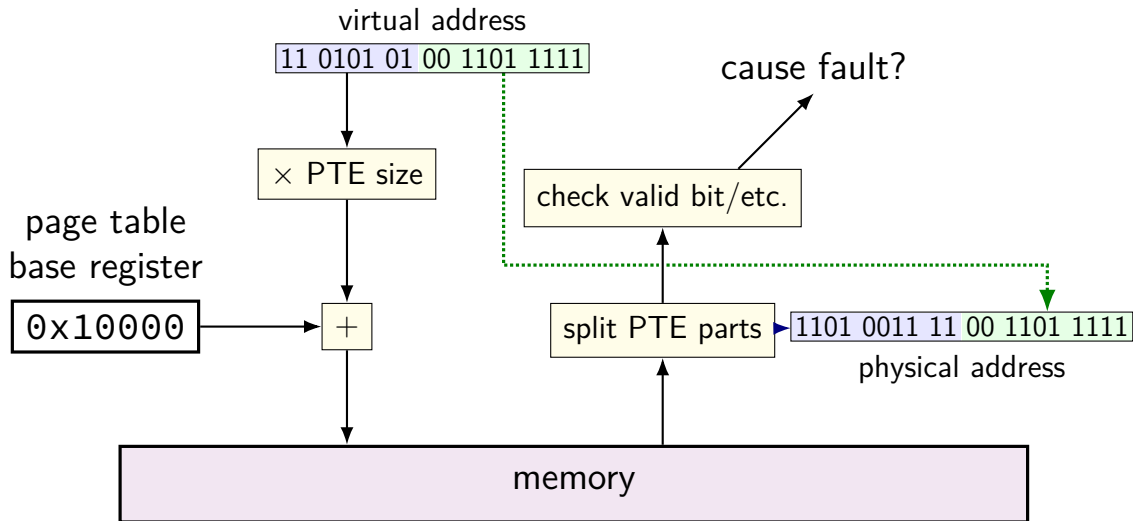
# memory access with page table



# memory access with page table

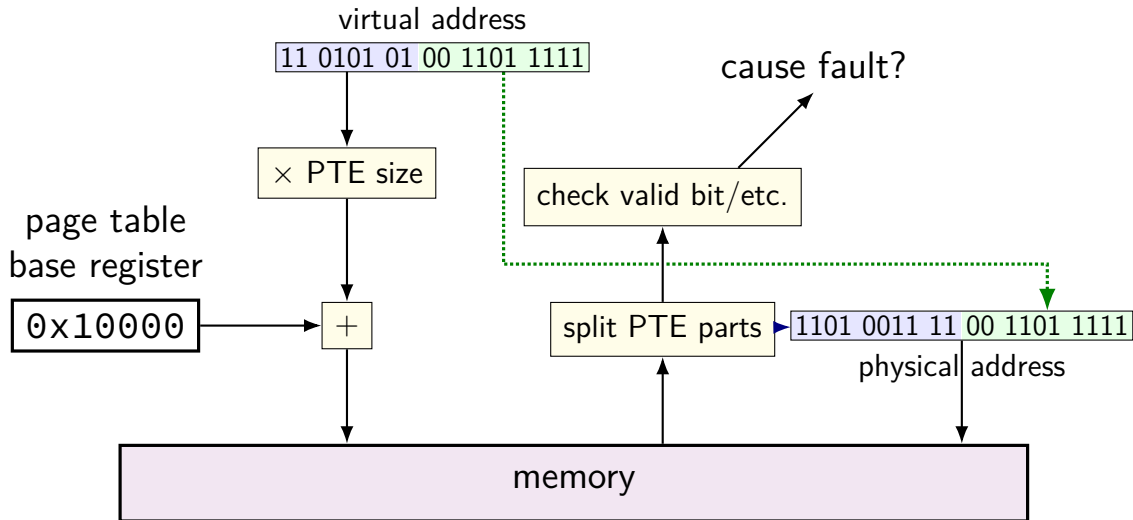


# memory access with page table

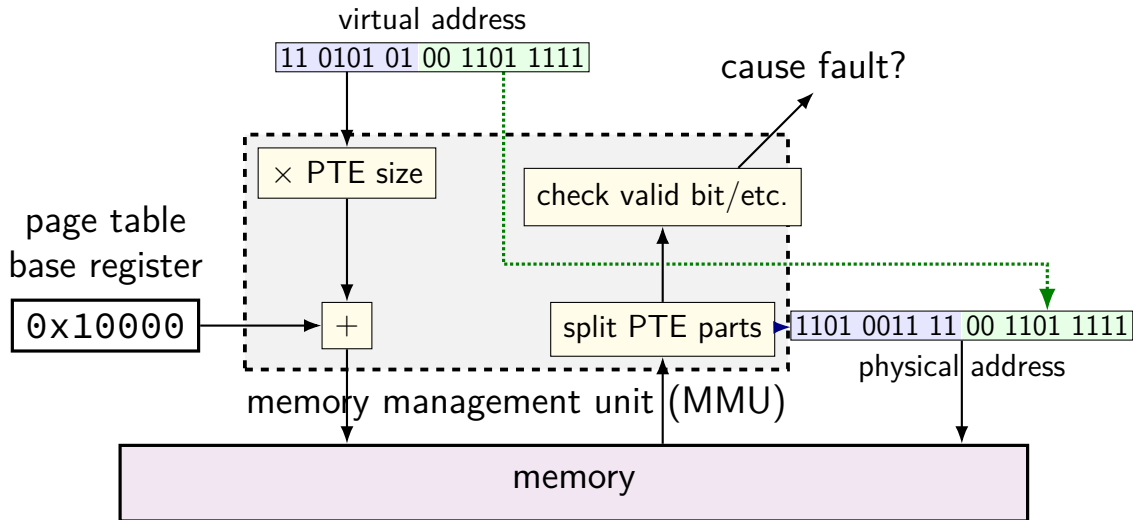




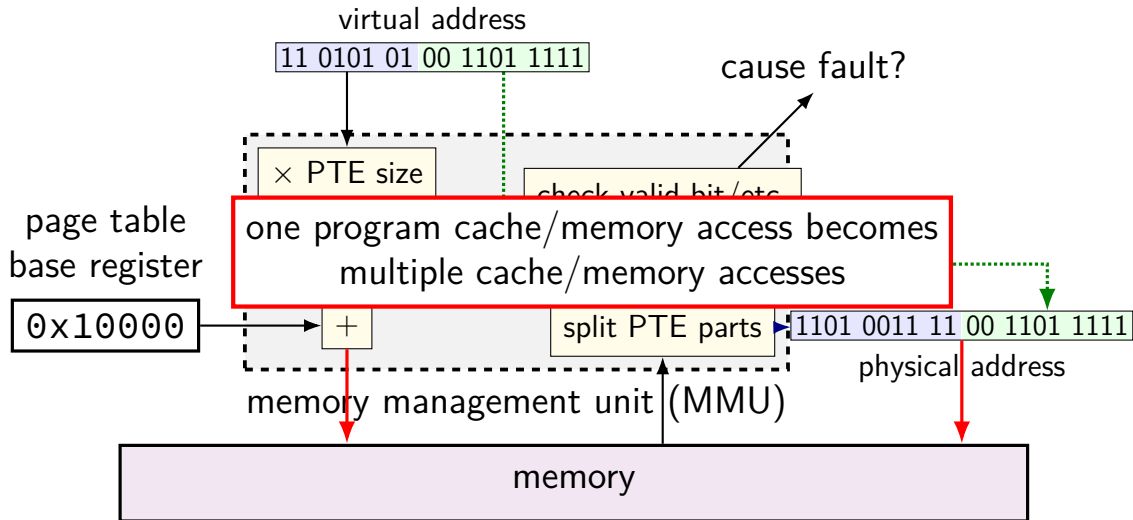
# memory access with page table



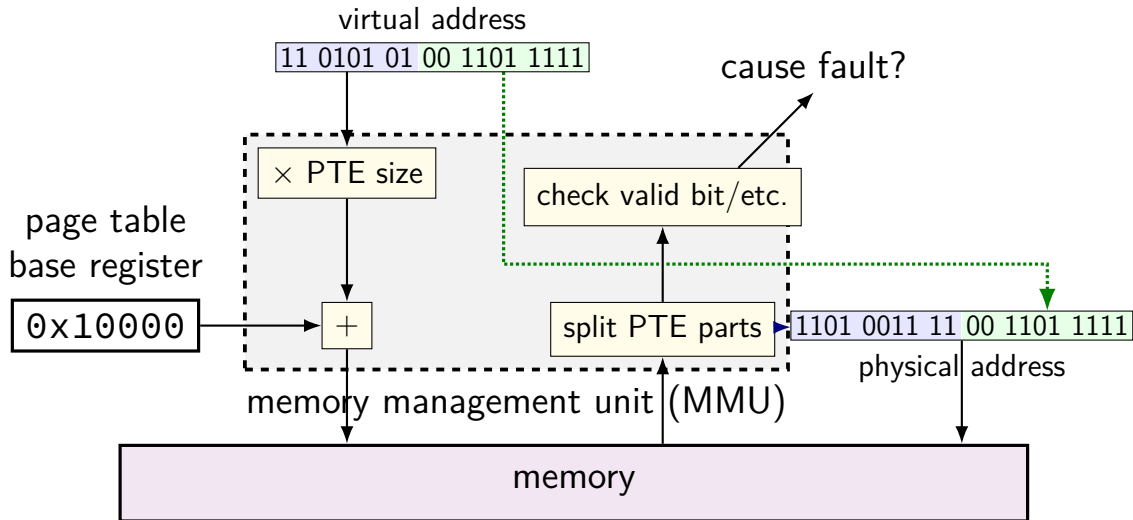
# memory access with page table



# memory access with page table



# memory access with page table



# exercise setup

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

# exercise setup

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	1 D2 D3
0x24-7	5 D6 D7
0x28-B	A AB BC
0x2C-F	E EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

phys. page 0

phys. page 1

# exercise

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses) 0x18 = ???; 0x03 = ???; 0x0A = ???; 0x13 = ???

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

# exercise

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses)  $0x18 = ?$ ;  $0x03 = ???$ ;  $0x0A = ???$ ;  $0x13 = ???$

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
$0x00-3$	00 11 22 33
$0x04-7$	44 55 66 77
$0x08-B$	88 99 AA BB
$0x0C-F$	CC DD EE FF
$0x10-3$	1A 2A 3A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	CB 0B CB 0B
$0x38-B$	DC 0C DC 0C
$0x3C-F$	EC 0C EC 0C



# exercise

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses) 0x18 = ; 0x03 = ; 0x0A = ???; 0x13 = ???

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

# exercise

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses) 0x18 = ; 0x03 = ; 0x0A = ; 0x13 = ???

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

# exercise

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses) 0x18 = ; 0x03 = ; 0x0A = ; 0x13 =

page table

virtual page #	valid?	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

# 1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;  
page table base register 0x20; translate virtual address 0x31

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

# 1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;  
page table base register 0x20; translate virtual address 0x31

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x31 = 11 0001

*PTE addr:*

$0x20 + 6 \times 1 = 0x26$

*PTE value:*

0xF6 = 1111 0110

PPN 111, valid 1

$M[111\ 001] = M[0x39]$

→ 0x0C

# 1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;  
page table base register 0x20; translate virtual address 0x31

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x31 = 11 0001

*PTE addr:*

$0x20 + 6 \times 1 = 0x26$

*PTE value:*

0xF6 = 1111 0110

PPN 111, valid 1

$M[111\ 001] = M[0x39]$

$\rightarrow 0x0C$

# 1-level exercise (1)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;  
page table base register 0x20; translate virtual address 0x31

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x31 = 11 0**001**

*PTE addr:*

$0x20 + 6 \times 1 = 0x26$

*PTE value:*

0xF6 = 1111 0110

PPN 111, valid 1

$M[111 \text{ } 001] = M[0x39]$

$\rightarrow 0x0C$

# 1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other  
page table base register 0x20; translate virtual address 0x12

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C



# 1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other  
page table base register 0x20; translate virtual address 0x12

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x12 = 01 0010

*PTE addr:*

0x20 + 2 × 1 = 0x22

*PTE value:*

0xD2 = 1101 0010

PPN 110, valid 1

M[110 010] = M[0x32]

→ 0xBA

# 1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other  
page table base register 0x20; translate virtual address 0x12

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x12 = 01 0010

*PTE addr:*

$0x20 + 2 \times 1 = 0x22$

*PTE value:*

0xD2 = 1101 0010

PPN 110, valid 1

$M[110\ 010] = M[0x32]$

→ 0xBA

# 1-level exercise (2)

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other  
page table base register 0x20; translate virtual address 0x12

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	F4 F5 F6 F7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x12 = 01 0**010**

*PTE addr:*

$0x20 + 2 \times 1 = 0x22$

*PTE value:*

0xD2 = 1101 0010

PPN 110, valid 1

$M[110 \text{ } 010] = M[0x32]$

→ 0xBA

## exercise: 64-bit system


my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

## exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages



top 16 bits of 64-bit addresses not used for translation

## exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries? (assuming page table like shown before)

exercise: how large are physical page numbers?

## exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries? (assuming page table like shown before)

exercise: how large are physical page numbers?

## exercise: 64-bit system

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries? (assuming page table like shown before)

exercise: how large are physical page numbers?

page table entries are 8 bytes (room for expansion, metadata)

trick: power of two size makes table lookup faster

would take up  $2^{39}$  bytes?? (512GB??)



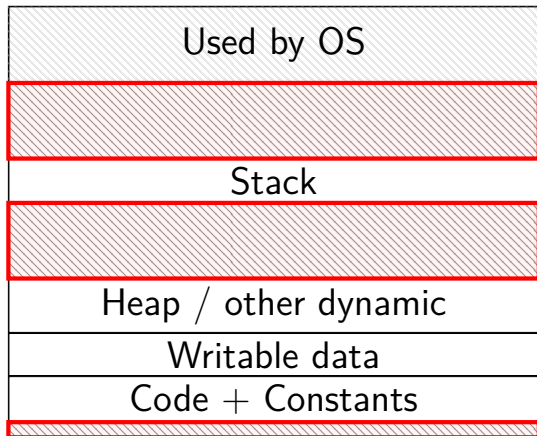
# huge page tables

huge virtual address spaces!

impossible to store PTE for every page

how can we save space?

# holes



most pages are **invalid**

## saving space

basic idea: don't store (most) invalid page table entries

use a data structure other than a flat array

want a map — lookup key (virtual page number), get value (PTE)

options?

## saving space

basic idea: don't store (most) invalid page table entries

use a data structure other than a flat array

want a map — lookup key (virtual page number), get value (PTE)

options?

### hashtable

actually used by some historical processors

but never common

# saving space

basic idea: don't store (most) invalid page table entries

use a data structure other than a flat array

want a map — lookup key (virtual page number), get value (PTE)

options?

hashtable

actually used by some historical processors  
but never common

tree data structure

but not quite a search tree

# search tree tradeoffs

lookup usually implemented in hardware

- lookup should be simple

- solution: lookup splits up address bits (no complex calculations)

lookup should not involve many memory accesses

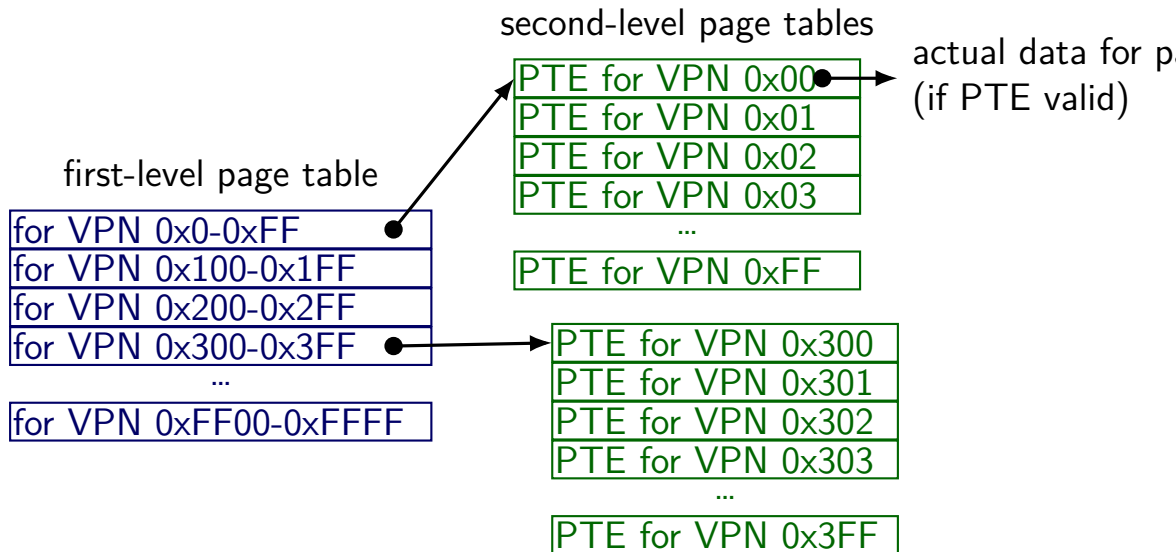
- doing two memory accesses is already very slow

- solution: tree with many children from each node

- (far from binary tree's left/right child)

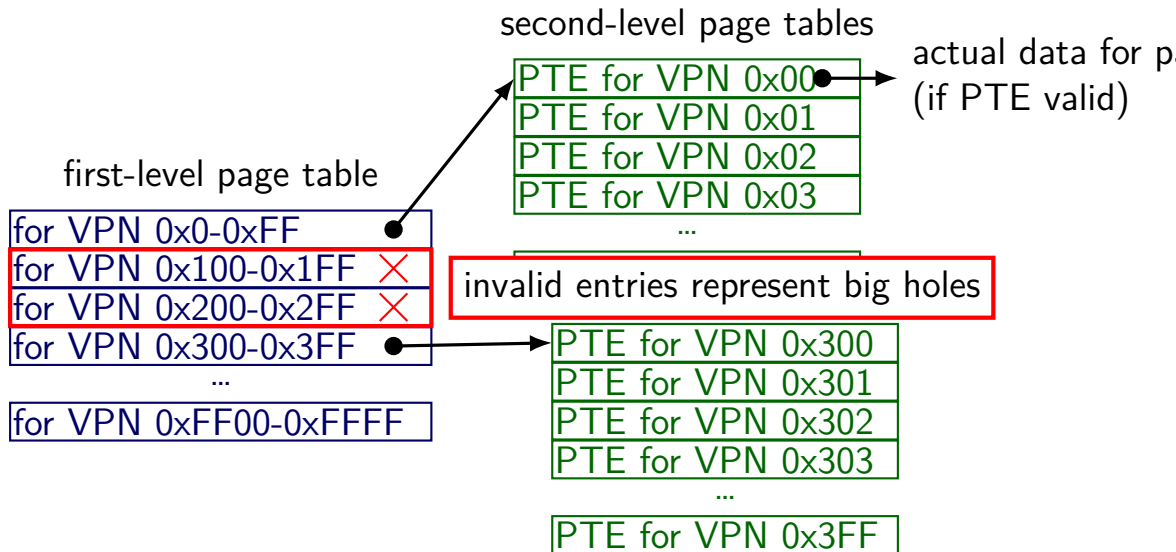
# two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)



# two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)





# two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)

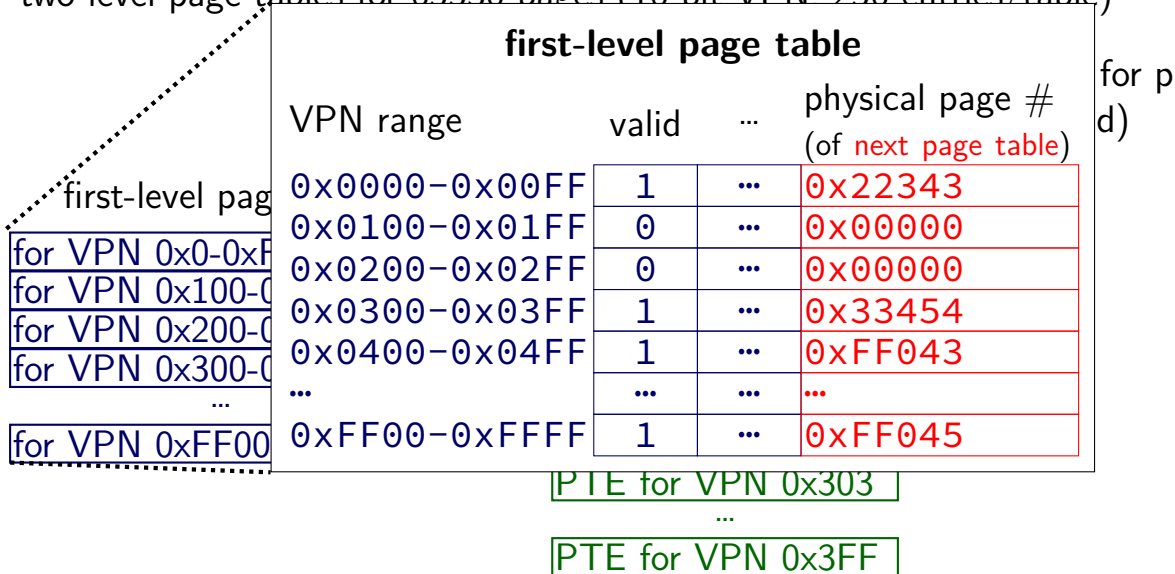
first-level page table				for p d)
VPN range	valid	...	physical page # (of next page table)	
0x0000-0x00FF	1	...	0x22343	
0x0100-0x01FF	0	...	0x00000	
0x0200-0x02FF	0	...	0x00000	
0x0300-0x03FF	1	...	0x33454	
0x0400-0x04FF	1	...	0xFF043	
...	...	...	...	
0xFF00-0xFFFF	1	...	0xFF045	

first-level page table for VPN 0x00-0xFF	PTE for VPN 0x303
for VPN 0x100-0x1FF	...
for VPN 0x200-0x2FF	PTE for VPN 0x3FF
for VPN 0x300-0x3FF	
...	
for VPN 0xFF00-0xFFFF	

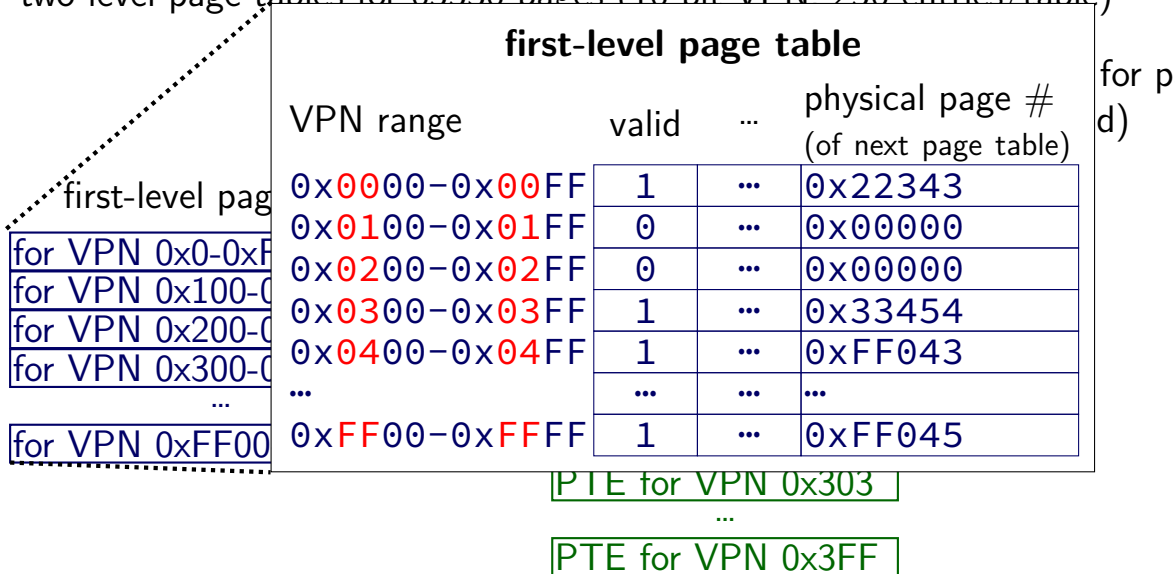
# two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)



# two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)



# two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)

first-level page table

for VPN 0x0-0xFF
for VPN 0x100-0x1FF ✗
for VPN 0x200-0x2FF ✗
for VPN 0x300-0x3FF
...
for VPN 0xFF00-0xFFFF

**a second-level page table**

VPN	valid	...	physical page # (of data)
0x300	0	1	0x42443
0x301	0	1	0x4A9DE
0x302	0	1	0x5C001
0x303	0	1	0x00000
0x304	0	1	0x6C223
...	...	...	...
0x3FF	...	1	0x00000

PTE for VPN 0x303

...

PTE for VPN 0x3FF

or p  
l)

# two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)

first-level page table

for VPN 0x0-0xFF	
for VPN 0x100-0x1FF	×
for VPN 0x200-0x2FF	×
for VPN 0x300-0x3FF	
...	
for VPN 0xFF00-0xFFFF	

a second-level page table

VPN	valid	...	physical page # (of data)
0x300	0	1	0x42443
0x301	0	1	0x4A9DE
0x302	0	1	0x5C001
0x303	0	1	0x00000
0x304	0	1	0x6C223
...	...	...	...
0x3FF	...	1	0x00000

PTE for VPN 0x303

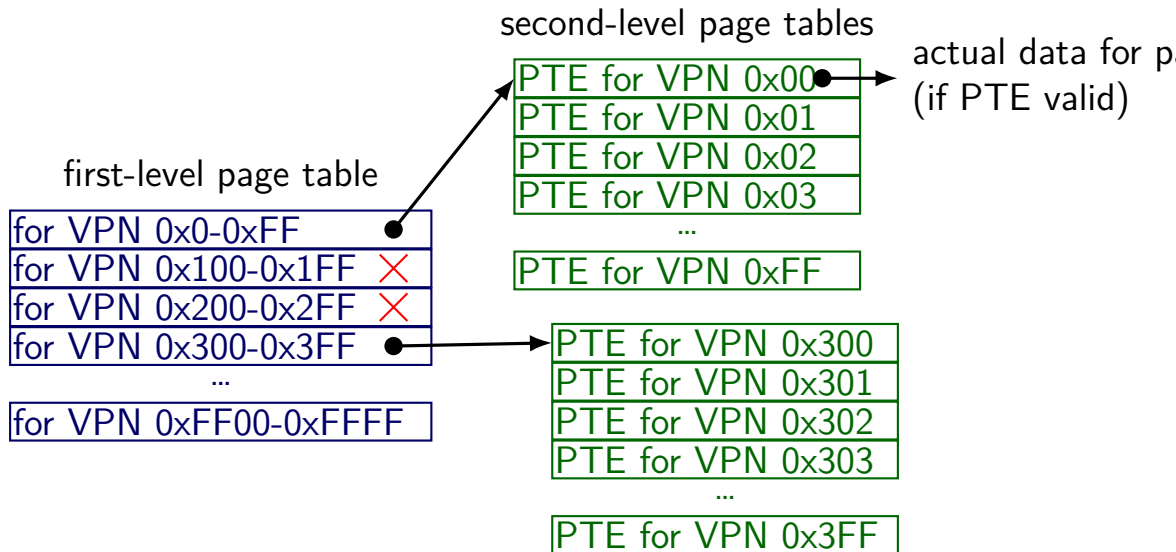
...

PTE for VPN 0x3FF

or p  
l)

# two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)



# two-level page table lookup

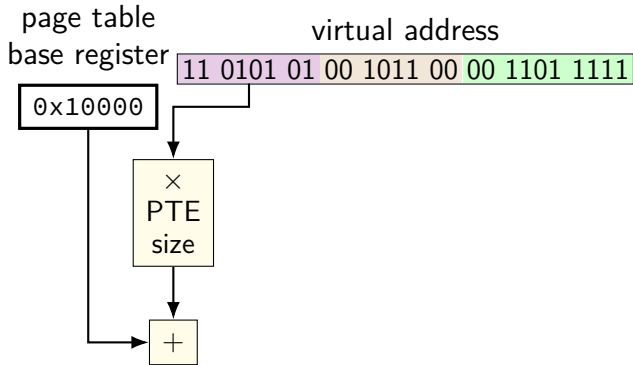
virtual address

11	0101	01	00	1011	00	00	1101	1111
----	------	----	----	------	----	----	------	------

VPN — split into two parts (one per level)

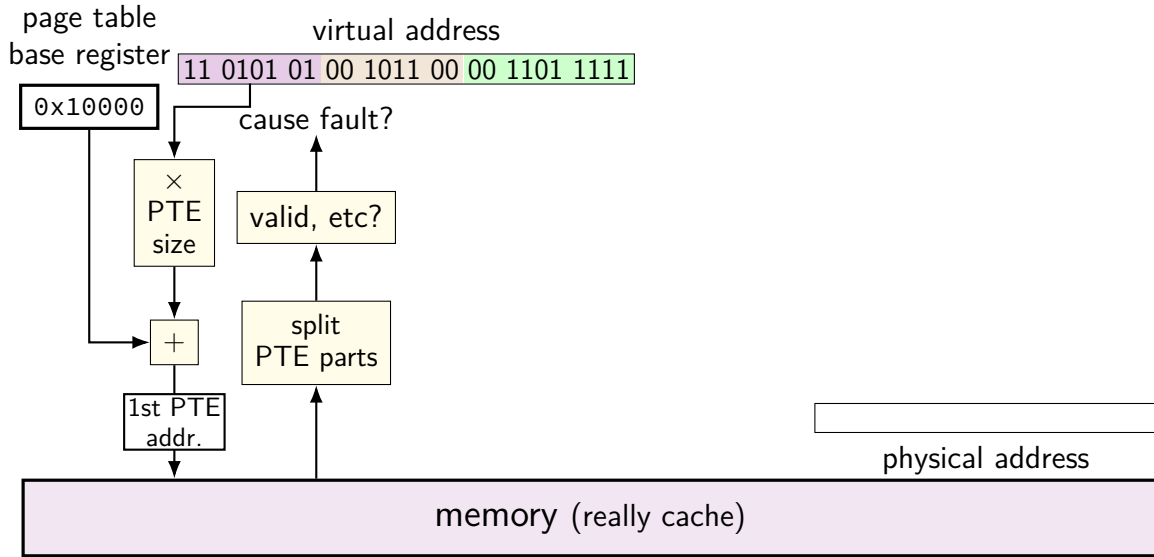
this example: parts equal sized — common, but not required

# two-level page table lookup

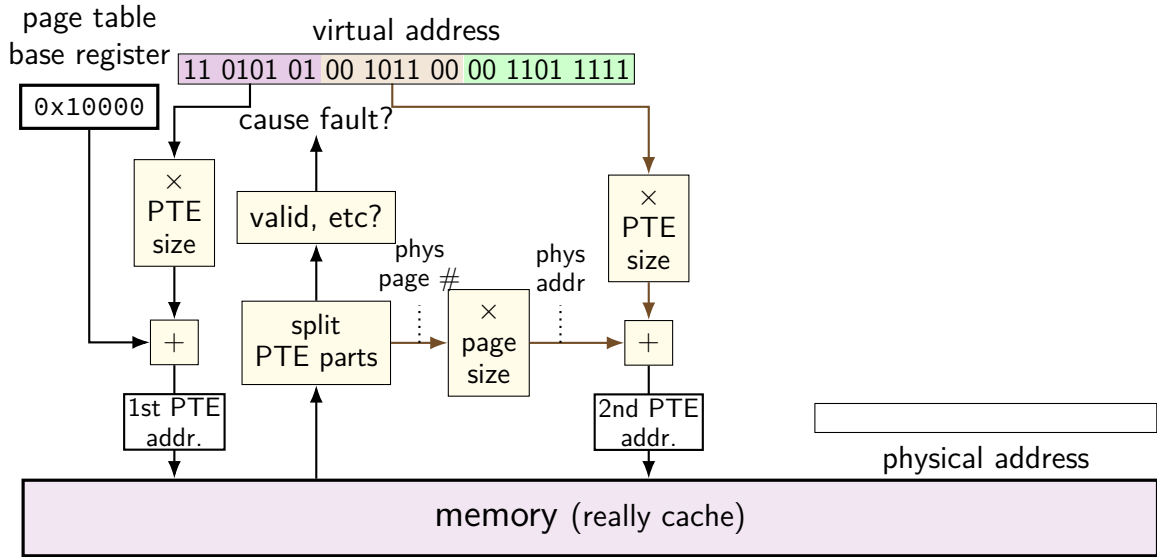




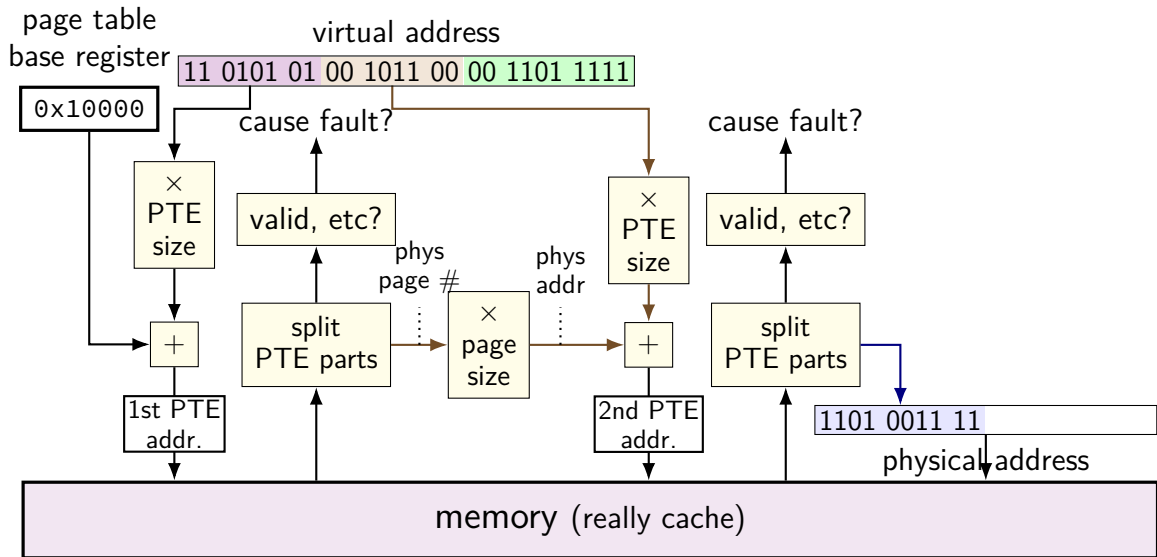
# two-level page table lookup



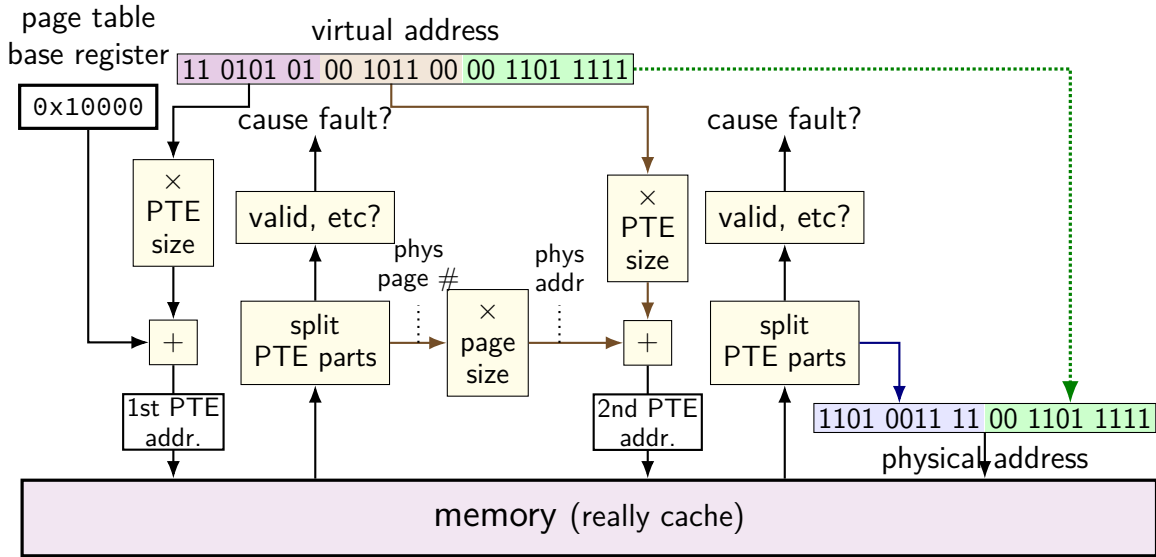
# two-level page table lookup



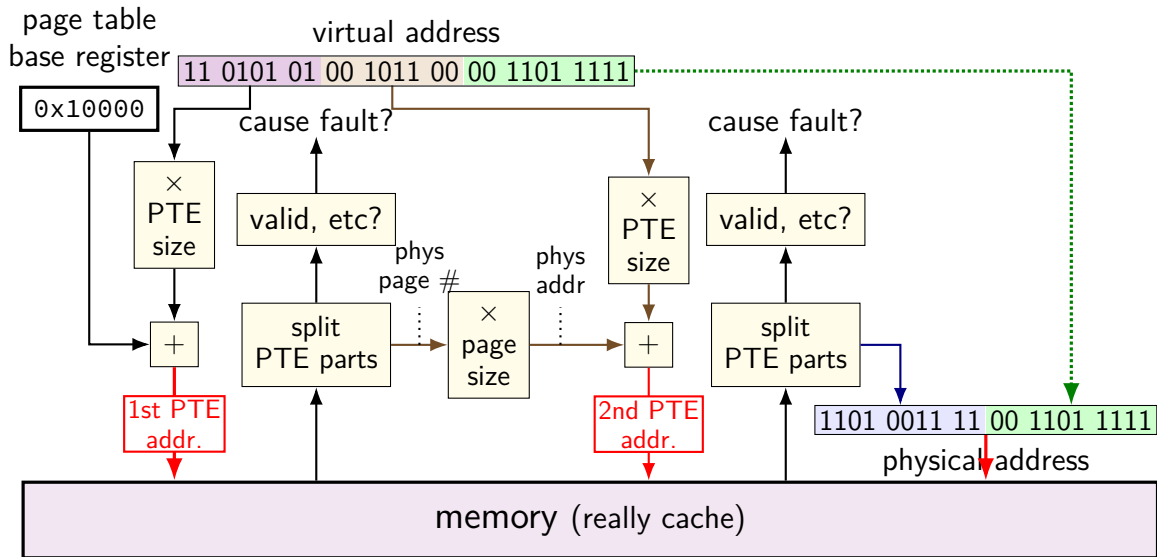
# two-level page table lookup



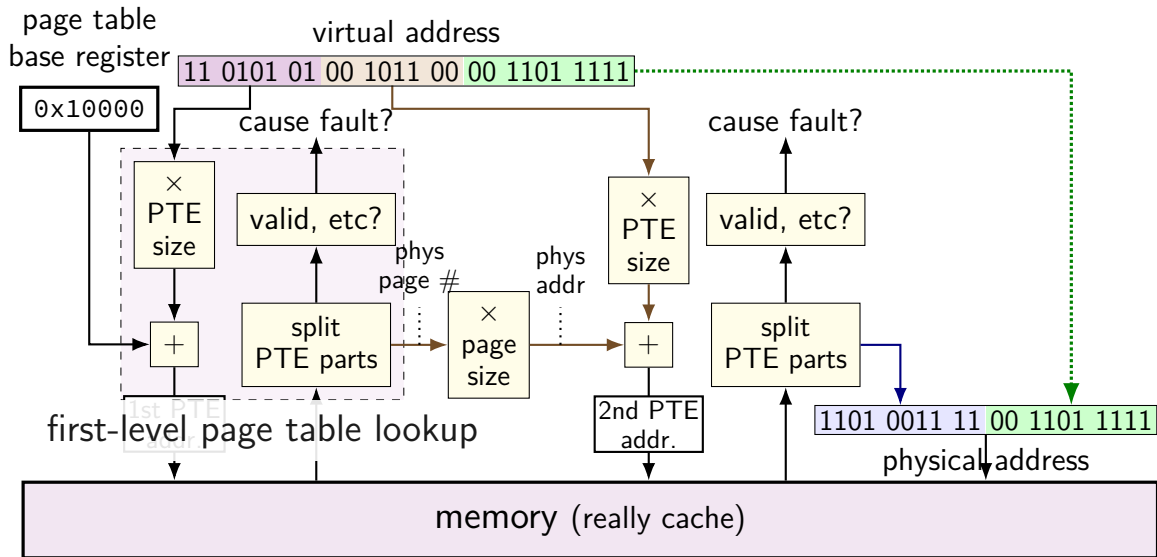
# two-level page table lookup



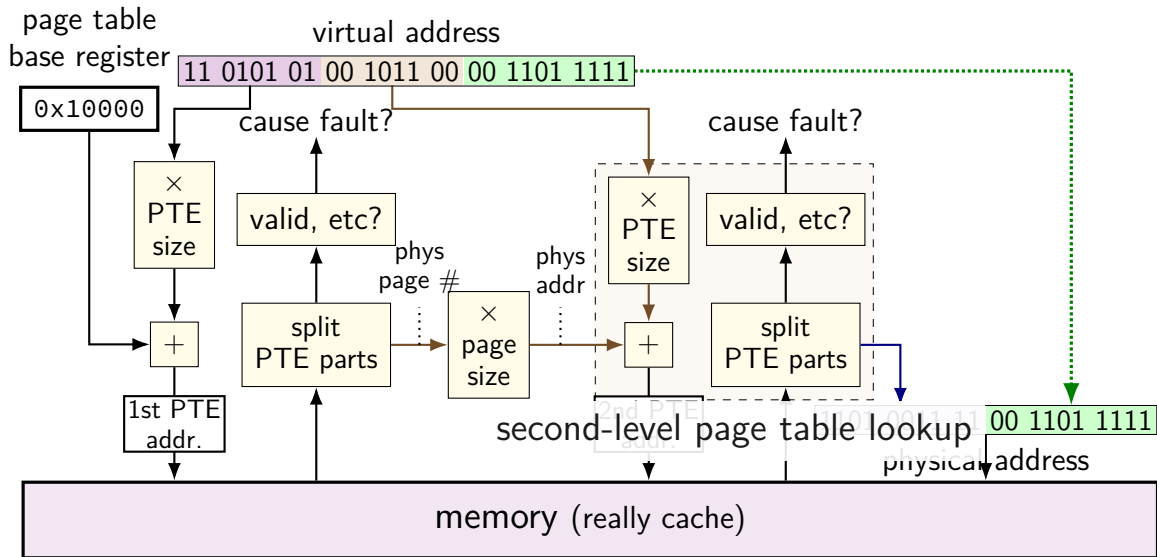
## two-level page table lookup



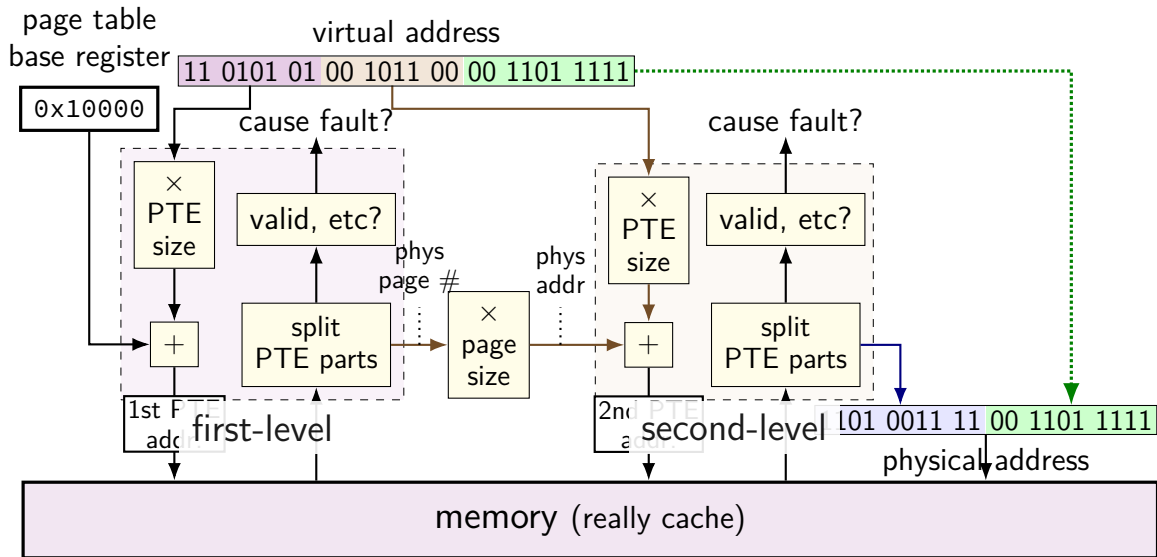
# two-level page table lookup



# two-level page table lookup

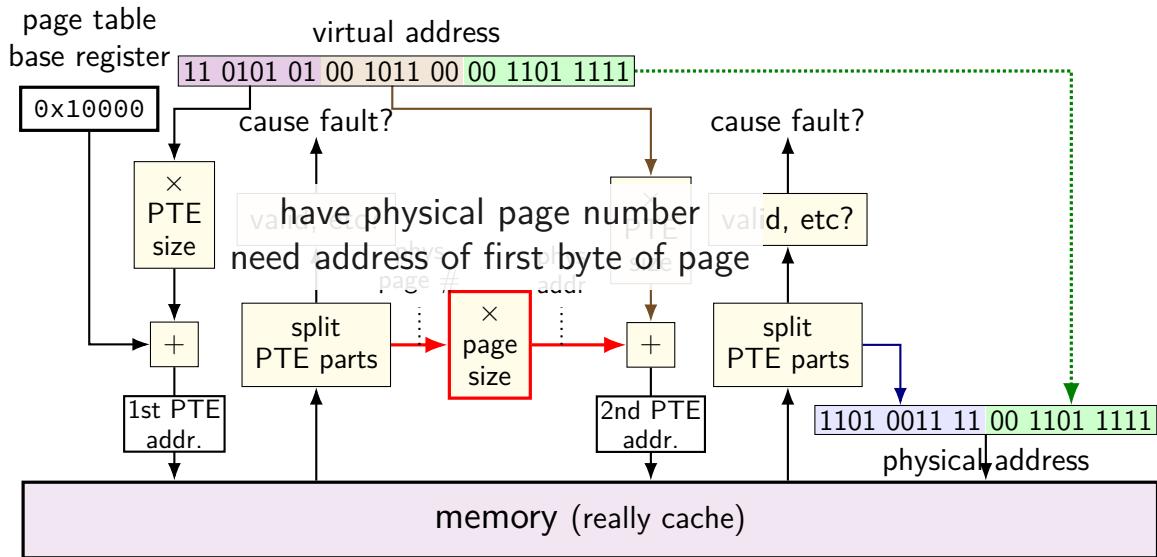


# two-level page table lookup

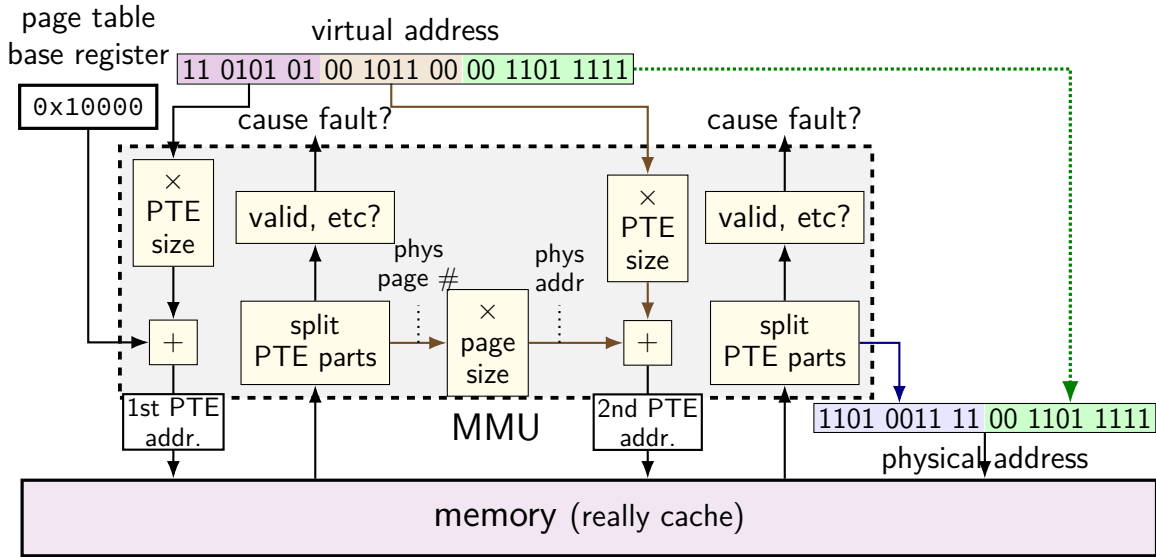




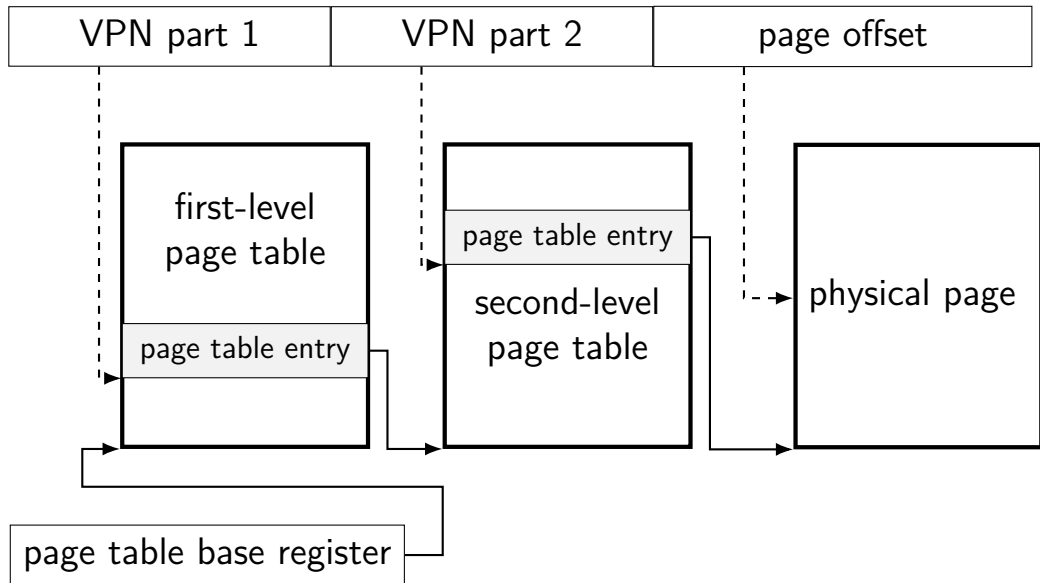
# two-level page table lookup



# two-level page table lookup



## another view



# multi-level page tables

VPN split into pieces for each level of page table

top levels: page table entries point to next page table

usually using physical page number of next page table

bottom level: page table entry points to destination page

validity checks at each level

# x86-64 page table splitting

48-bit virtual address

12-bit page offset (4KB pages)

36-bit virtual page number, split into four 9-bit parts

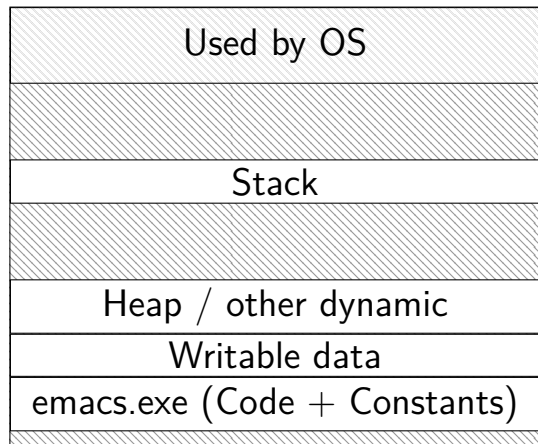
page tables at each level:  $2^9$  entries, 8 bytes/entry  
deliberate choice: each page table is one page

## note on VPN splitting

indexes used for lookup **parts of the virtual page number**  
(there are not multiple VPNs)

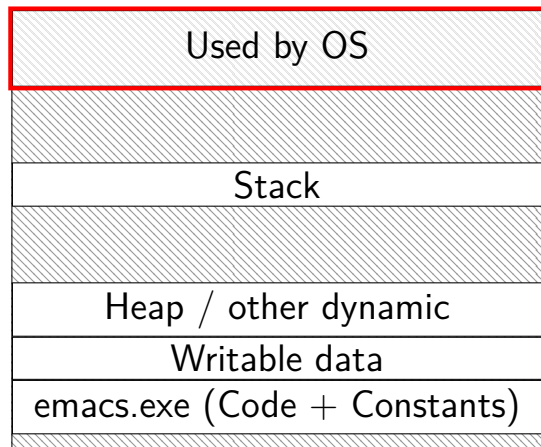
# emacs.exe

Emacs (run by user mst3k)



# emacs.exe

Emacs (run by user mst3k)

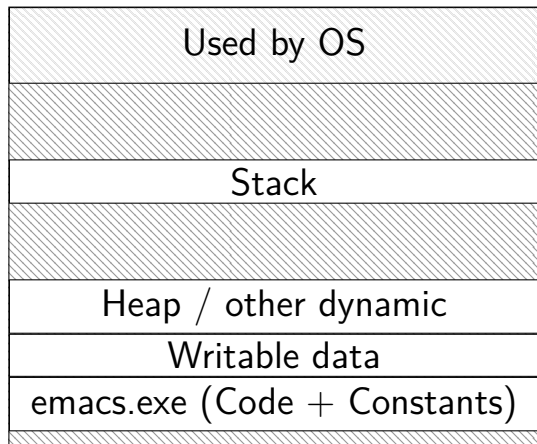


OS's memory

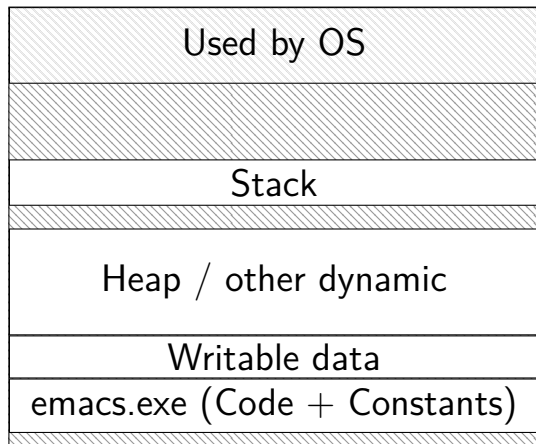


# emacs (two copies)

Emacs (run by user mst3k)

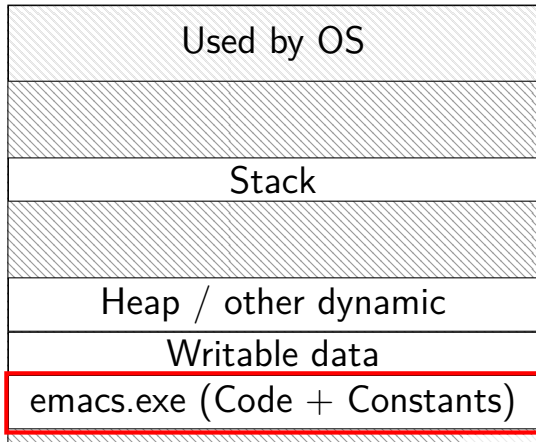


Emacs (run by user xyz4w)

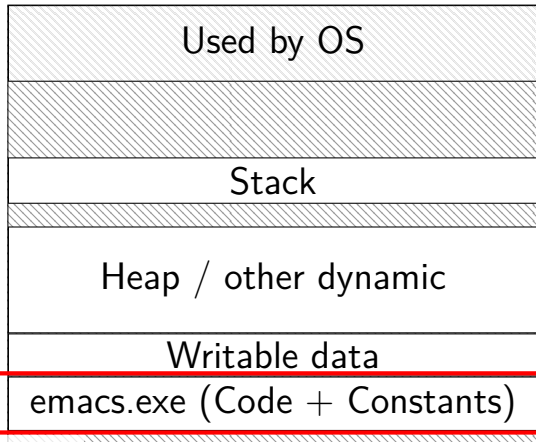


# emacs (two copies)

Emacs (run by user mst3k)



Emacs (run by user xyz4w)



same data?

## two copies of program

would like to only have one copy of program

what if mst3k's emacs tries to modify its code?

would break process abstraction:

“illusion of own memory”

# permissions bits

page table entry will have more **permissions bits**

can access in user mode?

can read from?

can write to?

can execute from?

checked by MMU like valid bit

page table (logically)

virtual page #	valid?	user?	write?	exec?	physical page #
0000 0000	0	0	0	0	00 0000 0000
0000 0001	1	1	1	0	10 0010 0110
0000 0010	1	1	1	0	00 0000 1100
0000 0011	1	1	0	1	11 0000 0011
...					
1111 1111	1	0	1	0	00 1110 1000

# assignment

## 2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE

page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused

page table base register 0x20; translate virtual address 0x131

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	00 91 72 13
0x24-7	D4 F5 36 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x20; translate virtual address 0x131

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	00 91 72 13
0x24-7	D4 F5 36 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x20; translate virtual address 0x131

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	00 91 72 13
0x24-7	D4 F5 36 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C



## 2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x20; translate virtual address 0x131

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	00 91 72 13
0x24-7	D4 F5 36 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x20; translate virtual address 0x131

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	00 91 72 13
0x24-7	D4 F5 36 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level example

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x20; translate virtual address 0x131

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	00 91 72 13
0x24-7	D4 F5 36 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level splitting

9-bit virtual address

6-bit physical address

8-byte pages  $\rightarrow$  3-bit page offset (bottom bits)

9-bit VA: 6 bit VPN + 3 bit PO

6-bit PA: 3 bit PPN + 3 bit PO

8 entry page tables  $\rightarrow$  3-bit VPN parts

9-bit VA: 3 bit VPN part 1; 3 bit VPN part 2

## 2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x08; translate virtual address 0x0FB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x08; translate virtual address 0x0FB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x08; translate virtual address 0x0FB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x08; translate virtual address 0x0FB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C



## 2-level exercise (1)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x08; translate virtual address 0x0FB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (2)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused;  
page table base register 0x10; translate virtual address 0x109

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 5A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x08; translate virtual address 0x00B

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x08; translate virtual address 0x00B

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (3)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x08; translate virtual address 0x00B

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (4)

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE  
page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused  
page table base register 0x08; translate virtual address 0x1CB

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register 0x10; translate virtual address 0x376

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register 0x10; translate virtual address 0x376

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C



## 2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register 0x10; translate virtual address 0x376

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register 0x10; translate virtual address 0x376

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register 0x10; translate virtual address 0x376

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register 0x10; translate virtual address 0x376

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

page table base register 0x10; translate virtual address 0x376

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

## 2-level exercise (5)

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

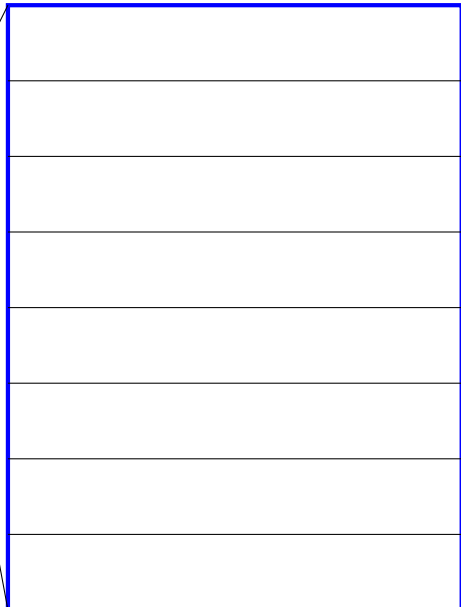
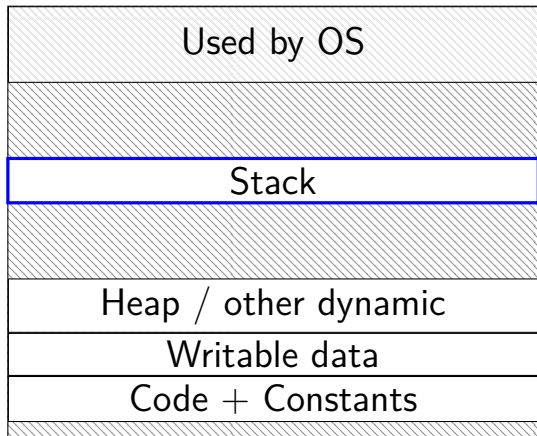
page table base register 0x10; translate virtual address 0x376

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	AC BC DC EC

physical addresses	bytes
0x20-3	D0 E1 D2 D3
0x24-7	D4 E5 D6 E7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	DB 0B DB 0B
0x38-B	EC 0C EC 0C
0x3C-F	FC 0C FC 0C

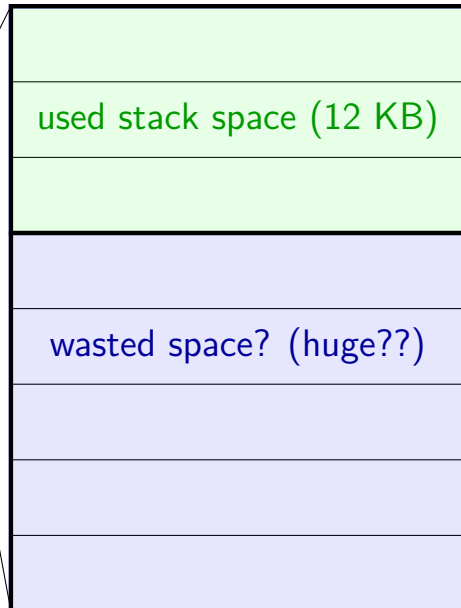
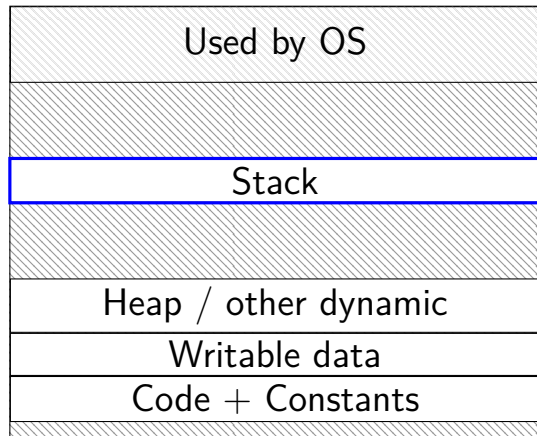
# space on demand

Program Memory



# space on demand

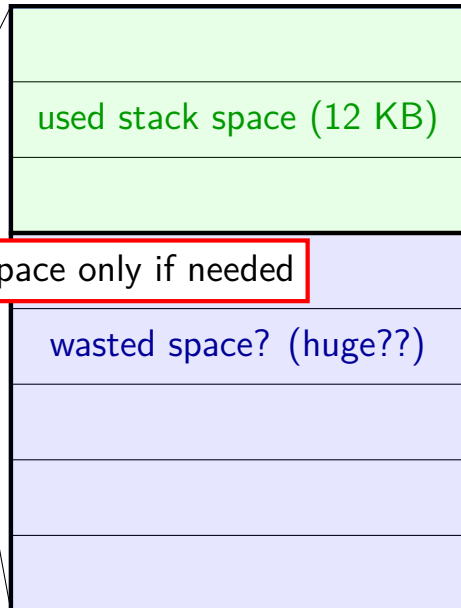
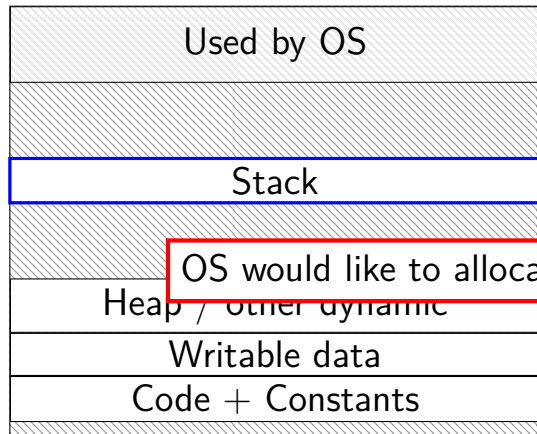
Program Memory





# space on demand

Program Memory



OS would like to allocate space only if needed

# allocating space on demand

%rsp = 0x7FFFC000

```
...  
// requires more stack space  
A: pushq %rbx  
  
B: movq 8(%rcx), %rbx  
C: addq %rbx, %rax  
...
```

VPN

```
...  
0x7FFFB  
0x7FFFC  
0x7FFFD  
0x7FFFE  
0x7FFFF  
...
```

valid? physical  
page

valid?	physical page
...	...
0	---
1	0x200DF
1	0x12340
1	0x12347
1	0x12345
...	...

# allocating space on demand

%rsp = 0x7FFFC000

```
...  
// requires more stack space  
A: pushq %rbx  
   → page fault!  
B: movq 8(%rcx), %rbx  
C: addq %rbx, %rax  
...
```

VPN

```
...  
0x7FFFB  
0x7FFFC  
0x7FFFD  
0x7FFFE  
0x7FFFF  
...
```

valid? physical  
page

valid?	physical page
...	...
0	---
1	0x200DF
1	0x12340
1	0x12347
1	0x12345
...	...

pushq triggers exception  
hardware says “accessing address 0x7FFBFF8”  
OS looks up what’s should be there — “stack”

# allocating space on demand

%rsp = 0x7FFFC000

```
...  
// requires more stack space  
A: pushq %rbx restarted  
B: movq 8(%rcx), %rbx  
C: addq %rbx, %rax  
...
```

VPN	valid?	physical page
...	...	...
0x7FFFB	1	0x200D8
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
...	...	...

in exception handler, OS allocates more stack space  
OS updates the page table  
then returns to retry the instruction

# allocating space on demand

note: the space doesn't have to be initially empty

only change: load from file, etc. instead of allocating empty page

loading program can be merely creating empty page table

everything else can be handled in response to page faults

no time/space spent loading/allocating unneeded space

# mmap

Linux/Unix has a function to “map” a file to memory

```
int file = open("somefile.dat", O_RDWR);
```

```
// data is region of memory that represents file  
char *data = mmap(..., file, 0);
```

```
// read byte 6 from somefile.dat  
char seventh_char = data[6];
```

```
// modifies byte 100 of somefile.dat  
data[100] = 'x';  
// can continue to use 'data' like an array
```

# swapping almost mmap

access mapped file for first time, read from disk  
(like swapping when memory was swapped out)

write “mapped” memory, write to disk eventually  
(like writeback policy in swapping)  
use “dirty” bit

extra detail: other processes should see changes  
all accesses to file use **same physical memory**

# Linux maps: list of maps

```
$ cat /proc/self/maps
```

```
00400000-0040b000 r-xp 00000000 08:01 48328831 /bin/cat
0060a000-0060b000 r-p 0000a000 08:01 48328831 /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 48328831 /bin/cat
01974000-01995000 rw-p 00000000 00:00 0 [heap]
7f60c718b000-7f60c7490000 r-p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
7f60c7490000-7f60c764e000 r-xp 00000000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c764e000-7f60c784e000 -p 001be000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c784e000-7f60c7852000 r-p 001be000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c7852000-7f60c7854000 rw-p 001c2000 08:01 96659129 /lib/x86_64-linux-gnu/libc-2.1
7f60c7854000-7f60c7859000 rw-p 00000000 00:00 0
7f60c7859000-7f60c787c000 r-xp 00000000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19.s
7f60c7a39000-7f60c7a3b000 rw-p 00000000 00:00 0
7f60c7a7a000-7f60c7a7b000 rw-p 00000000 00:00 0
7f60c7a7b000-7f60c7a7c000 r-p 00022000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19.s
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109 /lib/x86_64-linux-gnu/ld-2.19.s
7f60c7a7d000-7f60c7a7e000 rw-p 00000000 00:00 0
7ffc5d2b2000-7ffc5d2d3000 rw-p 00000000 00:00 0 [stack]
7ffc5d3b0000-7ffc5d3b3000 r-p 00000000 00:00 0 [vvar]
7ffc5d3b3000-7ffc5d3b5000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```



# Linux maps: list of maps

```
$ cat /proc/self/maps
```

```
00400000-0040b000 r-xp 00000000 08:01 48328831 /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 48328831 /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 48328831 /bin/cat
01974000-01995000 rw-p 00000000 00:00 0 [heap]
```

```
7f60c718b000-7f60c7490000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7490000-7f60c7490000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c764e000-7f60c764e000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c784e000-7f60c784e000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7852000-7f60c7852000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7854000-7f60c7854000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7859000-7f60c7859000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7a39000-7f60c7a39000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7a7a000-7f60c7a7a000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7a7b000-7f60c7a7b000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7a7c000-7f60c7a7c000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7f60c7a7d000-7f60c7a7d000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7ffc5d2b2000-7ffc5d2b2000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7ffc5d3b0000-7ffc5d3b0000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
7ffc5d3b3000-7ffc5d3b3000 r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

```
ffffffffffff-ffffffffffff r--p 00000000 08:01 77483660 /usr/lib/locale/locale-archive
```

OS tracks list of struct `vm_area_struct` with:  
(shown in this output):

virtual address start, end

permissions

offset in backing file (if any)

pointer to backing file (if any)

(not shown):

info about sharing of non-file data ...

# page tricks generally

deliberately make program trigger page/protection fault

but don't assume page/protection fault is an error

have separate data structures represent logically allocated memory

e.g. “addresses 0x7FFF8000 to 0x7FFFFFFF are the stack”

page table is for the hardware and not the OS

# hardware help for page table tricks

information about the address causing the fault

- e.g. special register with memory address accessed

- harder alternative: OS disassembles instruction, look at registers

(by default) rerun faulting instruction when returning from exception

precise exceptions: no side effects from faulting instruction or after

- e.g. `pushq` that caused did not change `%rsp` before fault

- e.g. can't notice if instructions were executed in parallel

# swapping

early motivation for virtual memory: **swapping**

using disk (or SSD, ...) as the next level of the memory hierarchy  
how our textbook and many other sources presents virtual memory

OS allocates **program space on disk**  
own mapping of virtual addresses to location on disk

DRAM is a cache for disk

# swapping

early motivation for virtual memory: **swapping**

using disk (or SSD, ...) as the next level of the memory hierarchy  
how our textbook and many other sources presents virtual memory

OS allocates **program space on disk**  
own mapping of virtual addresses to location on disk

**DRAM is a cache for disk**

# swapping components

“swap in” a page — exactly like allocating on demand!

- OS gets page fault — invalid in page table
- check where page actually is (from virtual address)
- read from disk
- eventually restart process

“swap out” a page

- OS marks as invalid in the page table(s)
- copy to disk (if modified)

# HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

- minimum size: 512 bytes

- writing tens of kilobytes basically as fast as writing 512 bytes

SSD reads and writes: hundreds of microseconds

- designed for writes/reads of kilobytes (not much smaller)

# HDD/SDDs are slow

HDD reads and writes: **milliseconds to tens of milliseconds**

- minimum size: 512 bytes

- writing tens of kilobytes basically as fast as writing 512 bytes

SSD reads and writes: **hundreds of microseconds**

- designed for reads/writes of kilobytes (not much smaller)



# HDD/SDDs are slow

HDD reads and writes: **milliseconds to tens of milliseconds**

- minimum size: 512 bytes

- writing tens of kilobytes basically as fast as writing 512 bytes

SSD reads and writes: **hundreds of microseconds**

- designed for reads/writes of kilobytes (not much smaller)

# HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

minimum size: 512 bytes

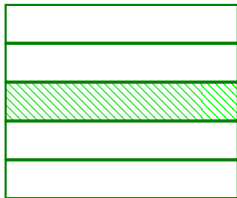
writing tens of **kilobytes** basically as fast as writing 512 bytes

SSD reads and writes: hundreds of microseconds

designed for reads/writes of **kilobytes** (not much smaller)

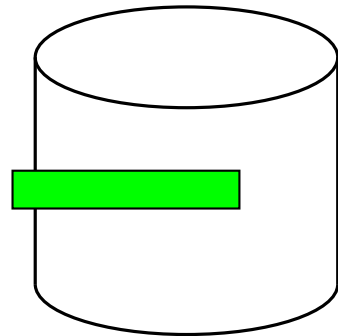
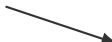
# swapping timeline

program A pages



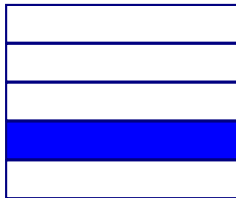
...

page fault



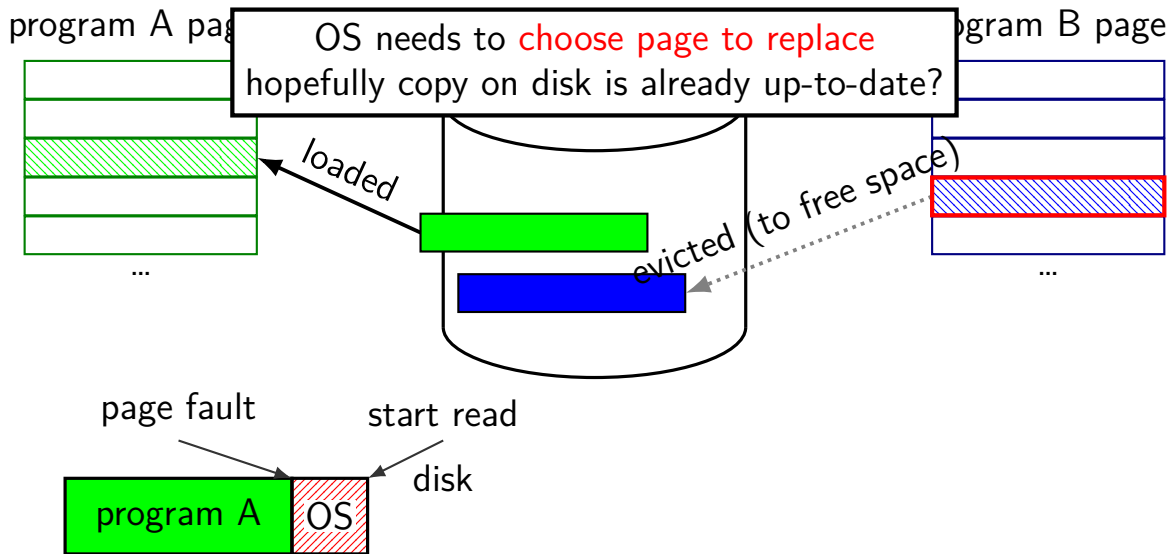
disk

program B page

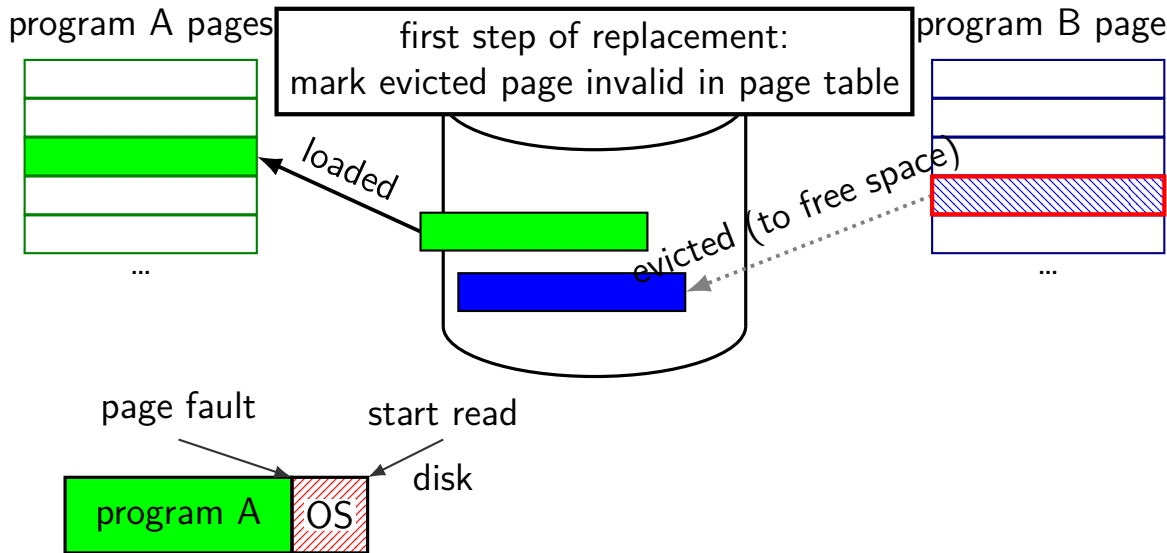


...

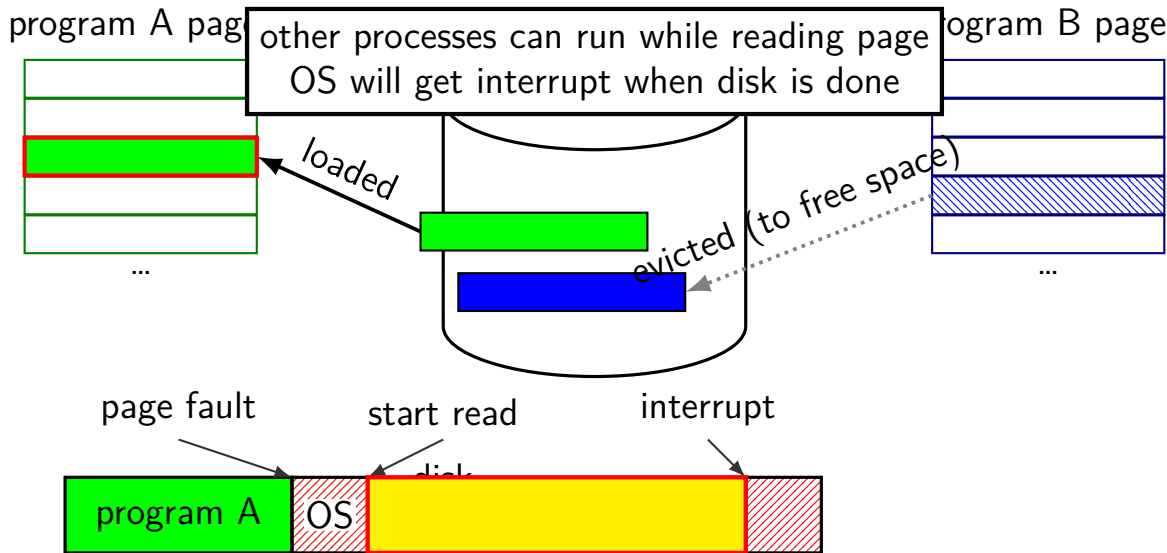
# swapping timeline



# swapping timeline

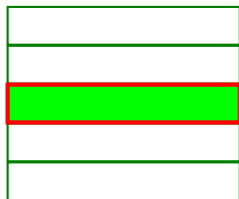


# swapping timeline



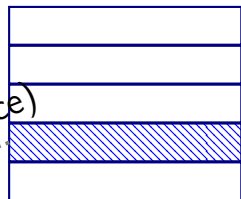
# swapping timeline

program A pages



process A's page table updated  
and restarted from point of fault

program B page



loaded

evicted (to free space)

page fault

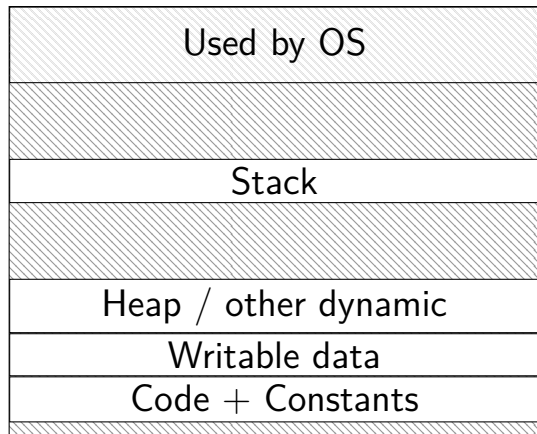
start read

interrupt

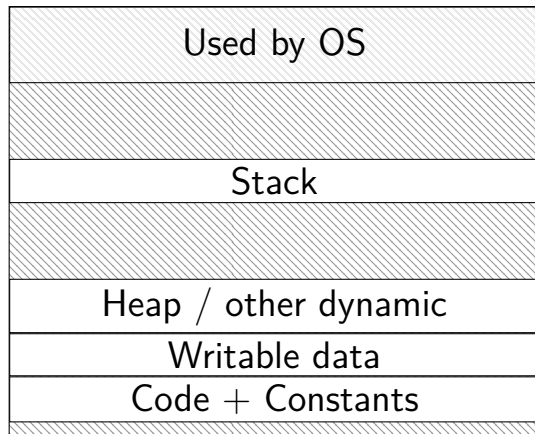


# do we really need a complete copy?

bash



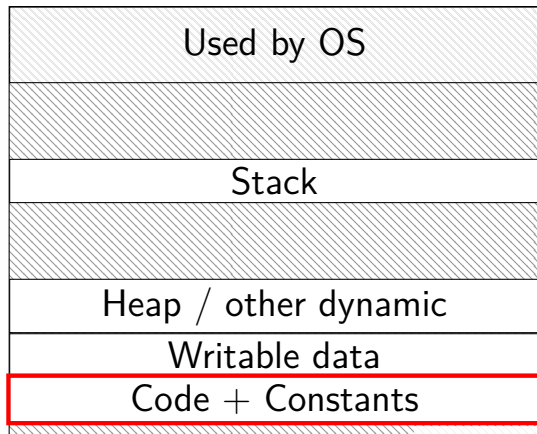
new copy of bash



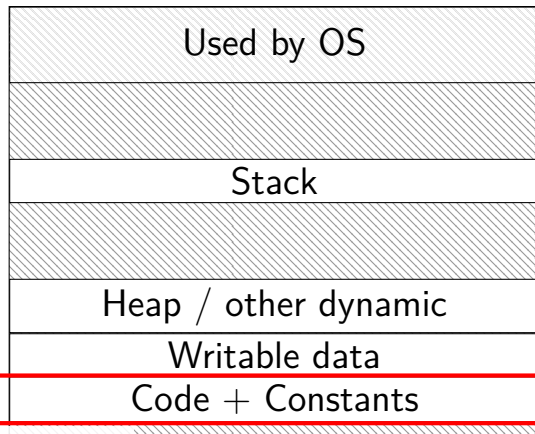


# do we really need a complete copy?

bash



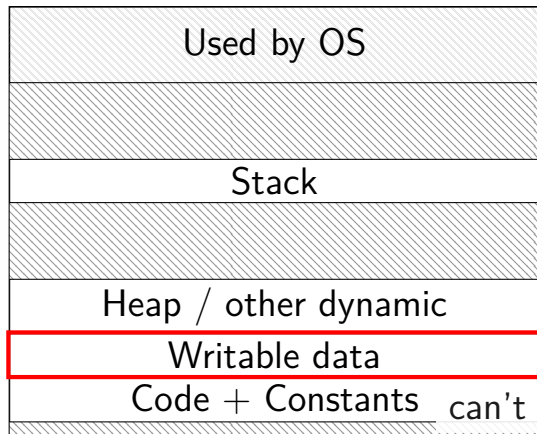
new copy of bash



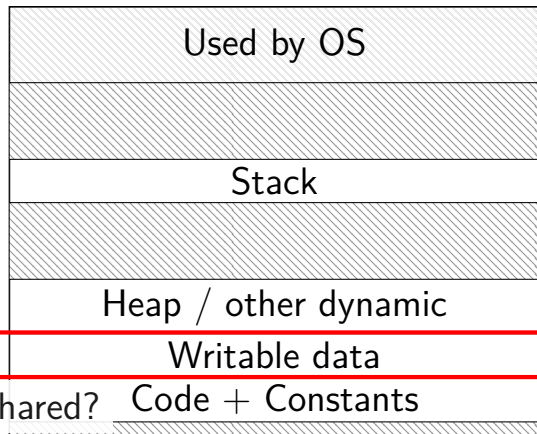
shared as read-only

# do we really need a complete copy?

bash



new copy of bash



can't be shared?

## trick for extra sharing

sharing writeable data is fine — until either process modifies it

- example: default value of global variables

- might typically not change

- (or OS might have preloaded executable's data anyways)

can we detect modifications?

## trick for extra sharing

sharing writeable data is fine — until either process modifies it

- example: default value of global variables

- might typically not change

- (or OS might have preloaded executable's data anyways)

can we detect modifications?

trick: tell CPU (via page table) shared part is read-only

processor will trigger a fault when it's written

# copy-on-write and page tables

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	1	0x12345
0x00602	1	1	0x12347
0x00603	1	1	0x12340
0x00604	1	1	0x200DF
0x00605	1	1	0x200AF
...	...	...	...

# copy-on-write and page tables

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

copy operation actually duplicates page table  
both processes **share all physical pages**  
but marks pages in **both copies as read-only**

# copy-on-write and page tables

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

when either process tries to write read-only page  
triggers a fault — OS actually copies the page

# copy-on-write and page tables

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	1	0x300FD
...	...	...	...

after allocating a copy, OS reruns the write instruction



**backup slides**

# signals

Unix-like **operating system feature**

like exceptions for processes:

can be triggered by external process

- kill command/system call

can be triggered by special events

- pressing control-C

- other events that would normal terminate program

  - 'segmentation fault'

  - illegal instruction

  - divide by zero

can invoke **signal handler** (like exception handler)

# exceptions v signals

(hardware) exceptions

handler runs in kernel mode

hardware decides when

hardware needs to save PC

processor next instruction changes

signals

handler runs in user mode

OS decides when

OS needs to save PC + registers

thread next instruction changes

# exceptions v signals

(hardware) exceptions

handler runs in kernel mode

hardware decides when

hardware needs to save PC

processor next instruction changes

signals

handler runs in user mode

OS decides when

OS needs to save PC + registers

thread next instruction changes

...but OS needs to run to trigger handler  
most likely “forwarding” hardware exception

# exceptions v signals

(hardware) exceptions

handler runs in kernel mode

hardware decides when

hardware needs to save PC

processor next instruction changes

signals

handler runs in user mode

OS decides when

OS needs to save PC + registers

thread next instruction changes

signal handler follows normal calling convention  
not special assembly like typical exception handler

# exceptions v signals

(hardware) exceptions

handler runs in kernel mode

hardware decides when

hardware needs to save PC

**processor** next instruction changes

signals

handler runs in user mode

OS decides when

OS needs to save PC + registers

**thread** next instruction changes

signal handler runs in same thread ('virtual processor')  
as process was using before

not running at 'same time' as the code it interrupts

# base program

```
int main() {  
    char buf[1024];  
    while (fgets(buf, sizeof buf, stdin)) {  
        printf("read %s", buf);  
    }  
}
```

# base program

```
int main() {  
    char buf[1024];  
    while (fgets(buf, sizeof buf, stdin)) {  
        printf("read %s", buf);  
    }  
}
```

---

some input

**read some input**

more input

**read more input**

*(control-C pressed)*

*(program terminates immediately)*



# base program

```
int main() {  
    char buf[1024];  
    while (fgets(buf, sizeof buf, stdin)) {  
        printf("read %s", buf);  
    }  
}
```

---

some input

**read some input**

more input

**read more input**

*(control-C pressed)*

*(program terminates immediately)*

## new program

```
int main() {  
    ... // added stuff shown later  
    char buf[1024];  
    while (fgets(buf, sizeof buf, stdin)) {  
        printf("read %s", buf);  
    }  
}
```

---

some input

**read some input**

more input

**read more input**

*(control-C pressed)*

**Control-C pressed?!**

another input **read another input**

## new program

```
int main() {  
    ... // added stuff shown later  
    char buf[1024];  
    while (fgets(buf, sizeof buf, stdin)) {  
        printf("read %s", buf);  
    }  
}
```

---

some input

**read some input**

more input

**read more input**

*(control-C pressed)*

**Control-C pressed?!**

another input **read another input**

## new program

```
int main() {  
    ... // added stuff shown later  
    char buf[1024];  
    while (fgets(buf, sizeof buf, stdin)) {  
        printf("read %s", buf);  
    }  
}
```

---

some input

**read some input**

more input

**read more input**

*(control-C pressed)*

**Control-C pressed?!**

another input **read another input**

## example signal program

```
void handle_sigint(int signum) {  
    /* signum == SIGINT */  
    write(1, "Control-C pressed?!\n",  
        sizeof("Control-C pressed?!\n"));  
}  
  
int main(void) {  
    struct sigaction act;  
    act.sa_handler = &handle_sigint;  
    sigemptyset(&act.sa_mask);  
    act.sa_flags = SA_RESTART;  
    sigaction(SIGINT, &act, NULL);  
  
    char buf[1024];  
    while (fgets(buf, sizeof buf, stdin)) {  
        printf("read %s", buf);  
    }  
}
```

## example signal program

```
void handle_sigint(int signum) {  
    /* signum == SIGINT */  
    write(1, "Control-C pressed?!\n",  
        sizeof("Control-C pressed?!\n"));  
}  
  
int main(void) {  
    struct sigaction act;  
    act.sa_handler = &handle_sigint;  
    sigemptyset(&act.sa_mask);  
    act.sa_flags = SA_RESTART;  
    sigaction(SIGINT, &act, NULL);  
  
    char buf[1024];  
    while (fgets(buf, sizeof buf, stdin)) {  
        printf("read %s", buf);  
    }  
}
```

## example signal program

```
void handle_sigint(int signum) {  
    /* signum == SIGINT */  
    write(1, "Control-C pressed?!\n",  
        sizeof("Control-C pressed?!\n"));  
}  
  
int main(void) {  
    struct sigaction act;  
    act.sa_handler = &handle_sigint;  
    sigemptyset(&act.sa_mask);  
    act.sa_flags = SA_RESTART;  
    sigaction(SIGINT, &act, NULL);  
  
    char buf[1024];  
    while (fgets(buf, sizeof buf, stdin)) {  
        printf("read %s", buf);  
    }  
}
```

# SIGxxx

signals types identified by number...

constants declared in `<signal.h>`

constant	likely use
SIGBUS	“bus error”; certain types of invalid memory accesses
SIGSEGV	“segmentation fault”; other types of invalid memory accesses
SIGINT	what control-C usually does
SIGFPE	“floating point exception”; includes integer divide-by-zero
SIGHUP, SIGPIPE	reading from/writing to disconnected terminal/socket
SIGUSR1, SIGUSR2	use for whatever you (app developer) wants
SIGKILL	terminates process (cannot be handled by process!)
SIGSTOP	suspends process (cannot be handled by process!)
...	...



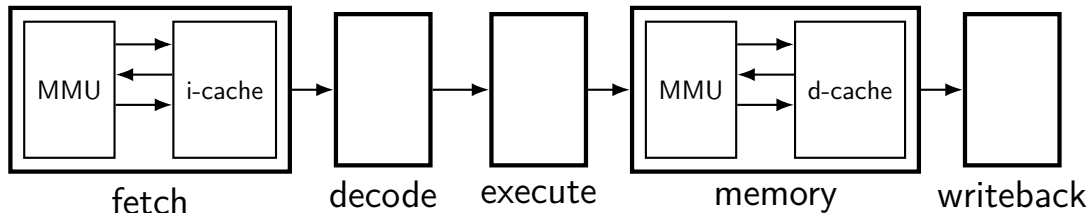
# SIGxxxx

signals types identified by number...

constants declared in `<signal.h>`

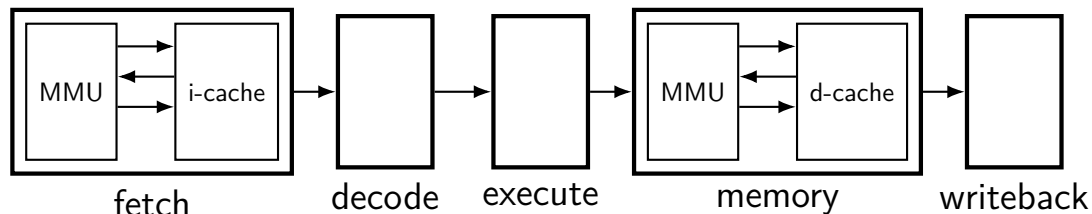
constant	likely use
SIGBUS	“bus error”; certain types of invalid memory accesses
SIGSEGV	“segmentation fault”; other types of invalid memory accesses
SIGINT	what control-C usually does
SIGFPE	“floating point exception”; includes integer divide-by-zero
SIGHUP, SIGPIPE	reading from/writing to disconnected terminal/socket
SIGUSR1, SIGUSR2	use for whatever you (app developer) wants
SIGKILL	terminates process (cannot be handled by process!)
SIGSTOP	suspends process (cannot be handled by process!)
...	...

# MMUs in the pipeline



up to four memory accesses per instruction

# MMUs in the pipeline

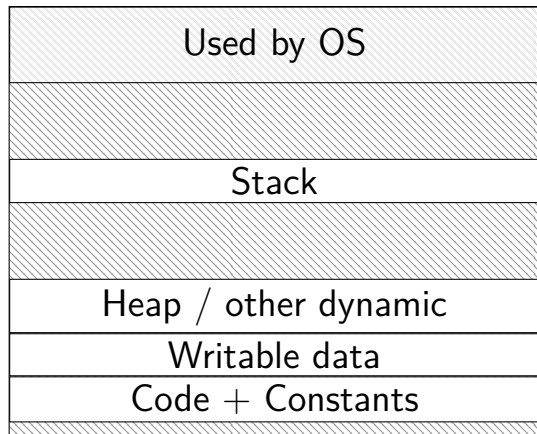


up to four memory accesses per instruction

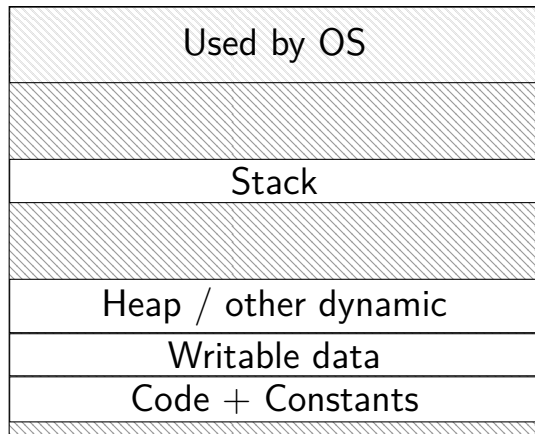
challenging to make this fast (topic for a future date)

# do we really need a complete copy?

bash

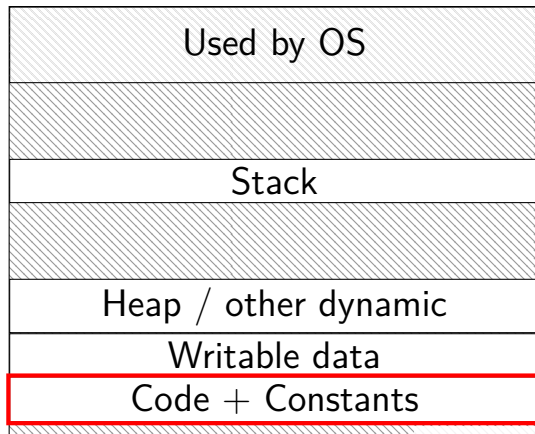


new copy of bash

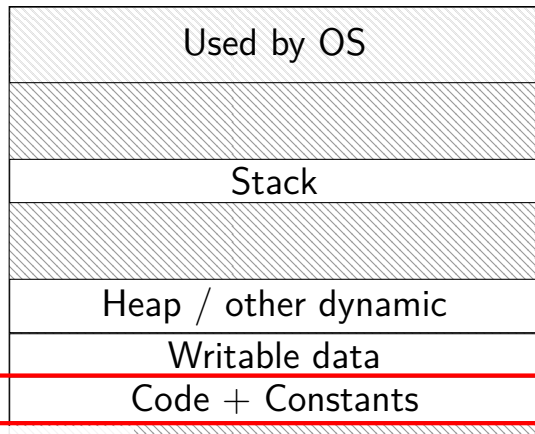


# do we really need a complete copy?

bash



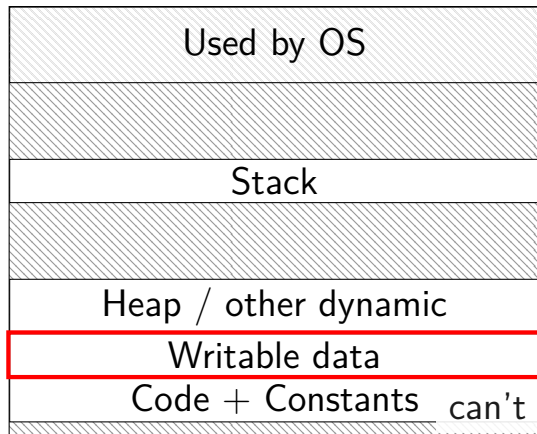
new copy of bash



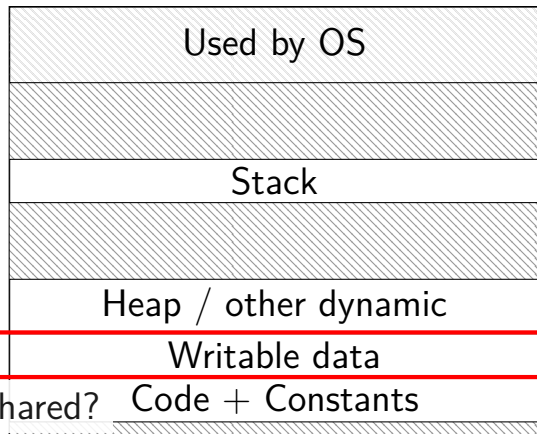
shared as read-only

# do we really need a complete copy?

bash



new copy of bash



can't be shared?

## trick for extra sharing

sharing writeable data is fine — until either process modifies it

- example: default value of global variables

- might typically not change

- (or OS might have preloaded executable's data anyways)

can we detect modifications?

## trick for extra sharing

sharing writeable data is fine — until either process modifies it

- example: default value of global variables

- might typically not change

- (or OS might have preloaded executable's data anyways)

can we detect modifications?

trick: tell CPU (via page table) shared part is read-only

processor will trigger a fault when it's written



# copy-on-write and page tables

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	1	0x12345
0x00602	1	1	0x12347
0x00603	1	1	0x12340
0x00604	1	1	0x200DF
0x00605	1	1	0x200AF
...	...	...	...

# copy-on-write and page tables

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

copy operation actually duplicates page table  
both processes **share all physical pages**  
but marks pages in **both copies as read-only**

# copy-on-write and page tables

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

when either process tries to write read-only page  
triggers a fault — OS actually copies the page

# copy-on-write and page tables

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
...	...	...	...

VPN	valid?	write?	physical page
...	...	...	...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	1	0x300FD
...	...	...	...

after allocating a copy, OS reruns the write instruction