last time

multi-level page tables

- tree data structure
- don't have entries for large empty spaces

several layers of page tables

- earlier page tables contain location of next page table
- can be marked invalid in early levels — save space

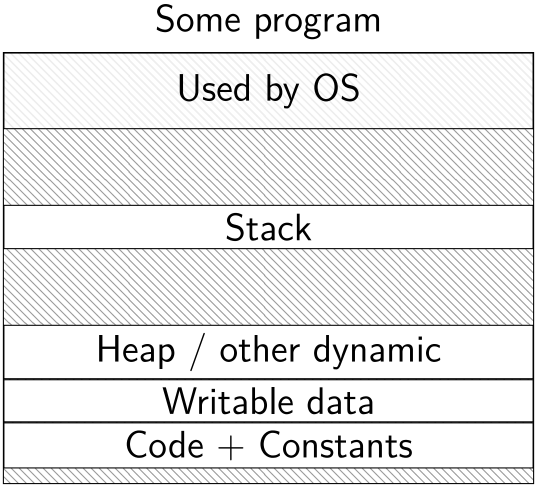
divide virtual page number into parts

anonymous feedback (1)

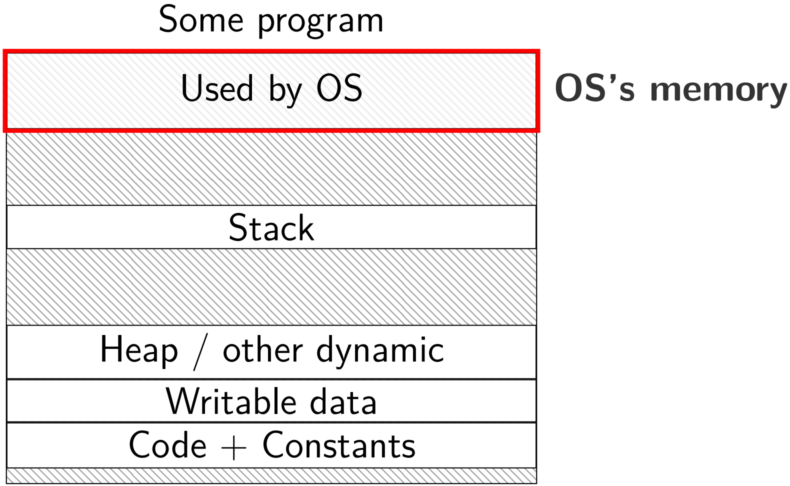
“In the previous class, there was a comment regarding the desire for a longer quiz with questions of lower point values. However, there was also a concern about not making the quiz excessively lengthy. I believe a good way to strike a balance in question weight is to incorporate more questions of an easier difficulty level. This approach would provide us with additional practice without dedicating too much time to each question, while also allowing us to earn extra points. For instance, the first two questions on the last quiz served as excellent practice and enabled us to assess our knowledge without being overly challenging.”

- probably a question complexity (not quite same as difficulty) issue
- for some topics, need to have questions not be bare recall from lecture/reading
- or ‘run this and see what the output is’ limits “minimum” complexity
  - e.g. need to have context re: commands used to build a program for makefile questions
  - don’t want questions where answer is “in the question”

running a program



running a program



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

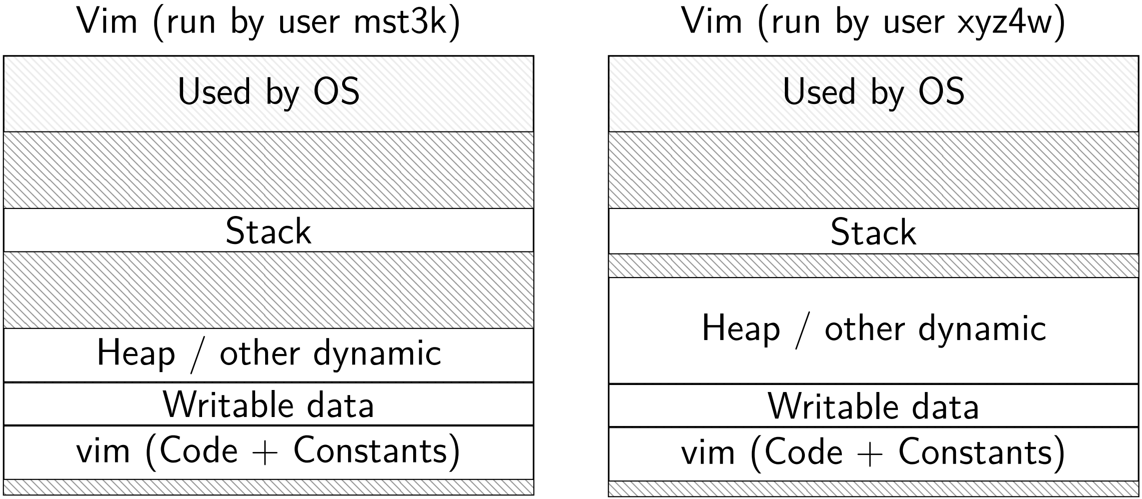
where in memory is the code that does this switching?

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

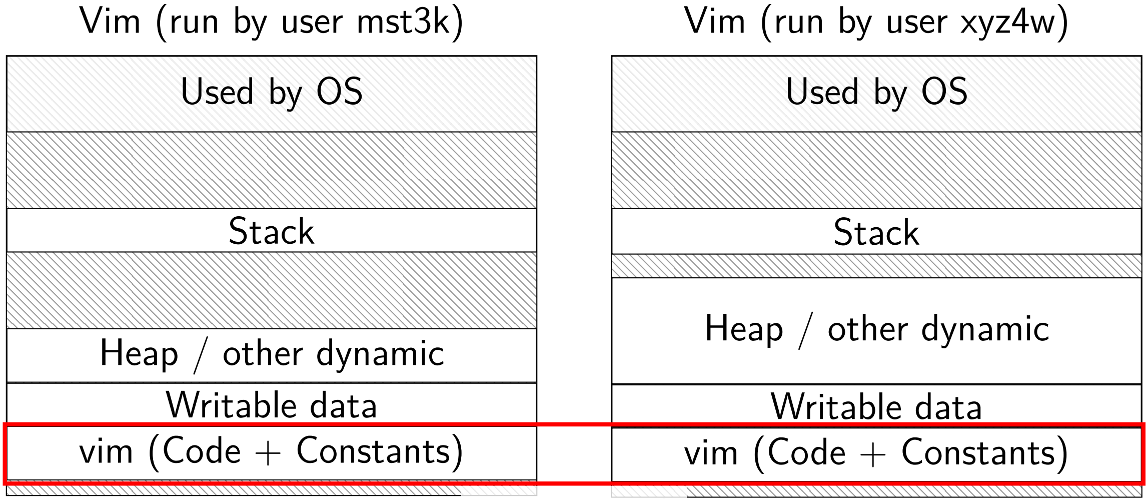
code better not be modified by user program

- otherwise: uncontrolled way to “escape” user mode

vim (two copies)



vim (two copies)



Vim (run by user mst3k)

Used by OS

Stack

Heap / other dynamic

Writable data

vim (Code + Constants)

Vim (run by user xyz4w)

Used by OS

Stack

Heap / other dynamic

Writable data

vim (Code + Constants)

same data?

two copies of program

would like to only have one copy of program

what if `mst3k`'s vim tries to modify its code?

would break process abstraction:

    "illusion of own memory"

permissions bits

page table entry will have more permissions bits

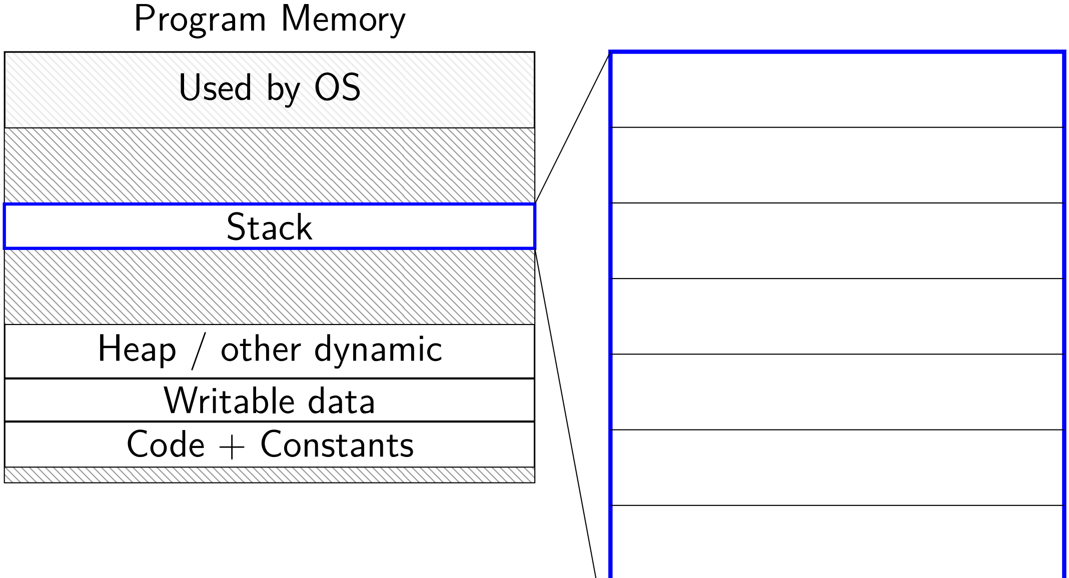
- can access in user mode?
- can read from?
- can write to?
- can execute from?

checked by MMU like valid bit

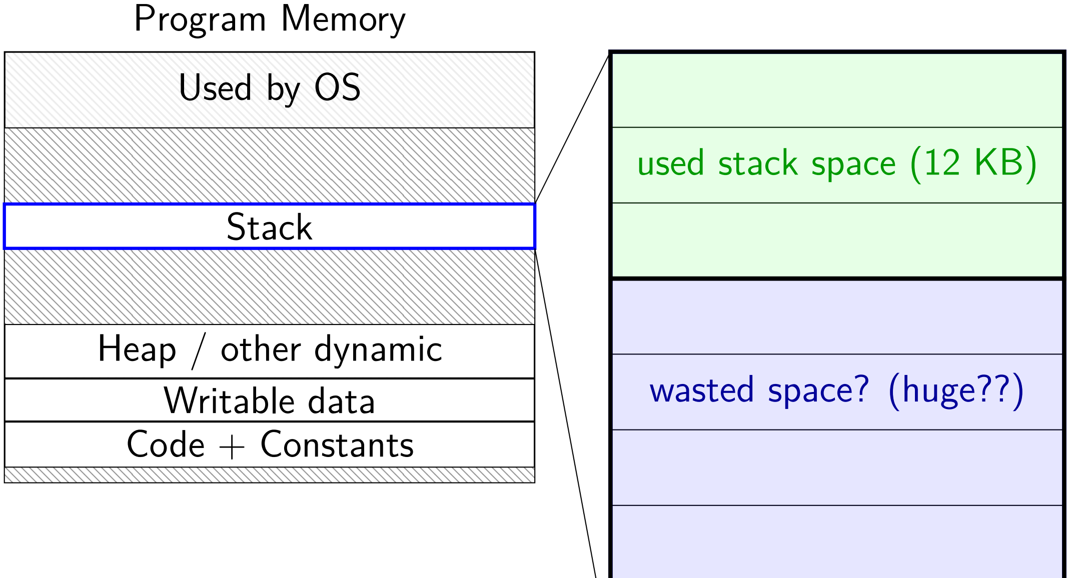
page table (logically)

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

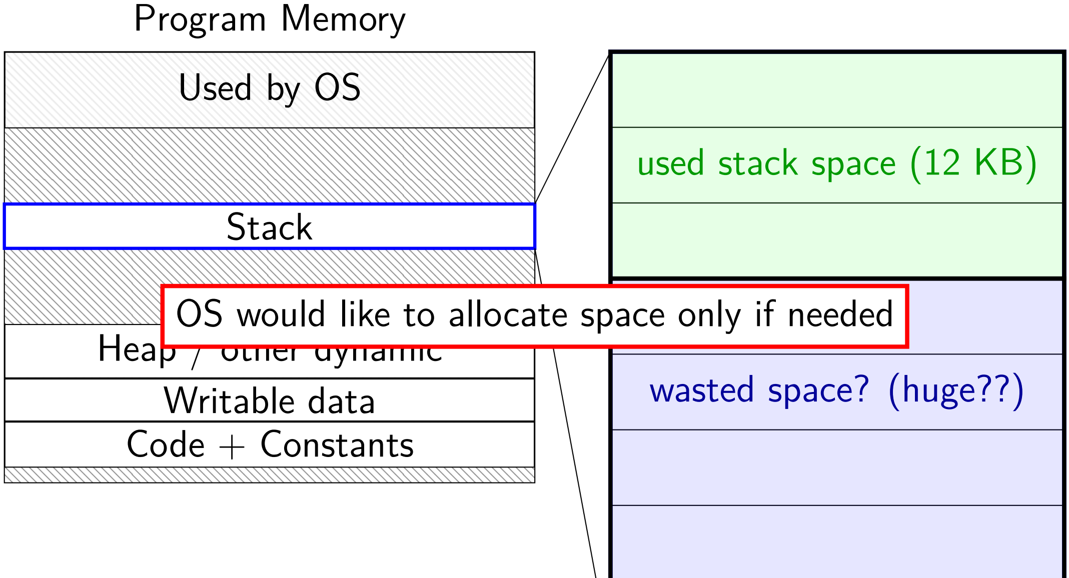
space on demand



space on demand



space on demand



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	valid?	physical page
…	…	…
0x7FFFB	0	---
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
…	…	…

allocating space on demand

%rsp = 0x7FFFC000

```
...
// requires more stack space
A: pushq %rbx

B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

page fault!

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

pushq triggers exception

hardware says "accessing address 0x7FFFBFF8"

OS looks up what's should be there — "stack"

allocating space on demand

```
%rsp = 0x7FFFC000
```

```
...
// requires more stack space
A: pushq %rbx

B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

restarted

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

page tricks generally

deliberately make program trigger page/protection fault

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

have seperate 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

POSIX process management

essential operations

process information: `getpid`

process creation: `fork`

running programs: `exec*`

also `posix_spawn` (not widely supported), …

waiting for processes to finish: `waitpid` (or `wait`)

process destruction, 'signaling': `exit`, `kill`

POSIX process management

essential operations

process information: `getpid`

process creation: `fork`

running programs: `exec*`

also `posix_spawn` (not widely supported), …

waiting for processes to finish: `waitpid` (or `wait`)

process destruction, 'signaling': `exit`, `kill`

fork

`pid_t fork()` — copy the current process

returns twice:

- in *parent* (original process): pid of new *child* process
- in *child* (new process): `0`

everything (but pid) duplicated in parent, child:

- memory
- file descriptors (later)
- registers

fork and process info (w/o copy-on-write)

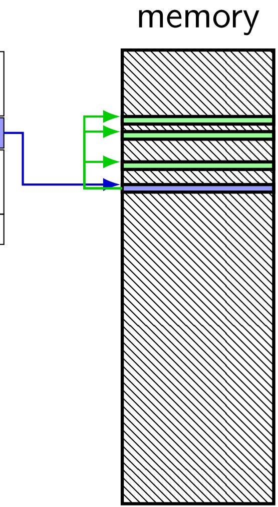
parent process info

user regs	`rax (return val.)=42,`
	`rcx=133, …`

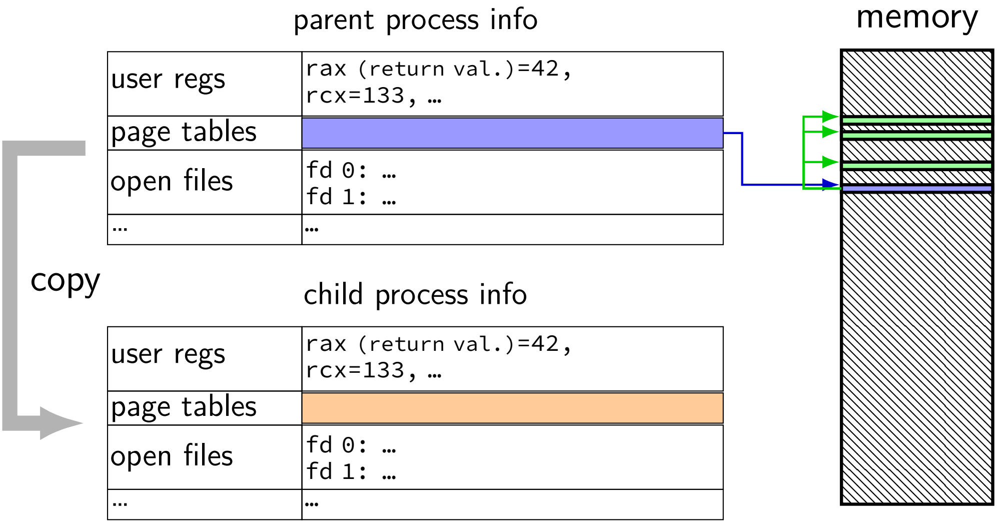
page tables	🟦

open files	`fd 0: …`
	`fd 1: …`

…	…



fork and process info (w/o copy-on-write)



parent process info

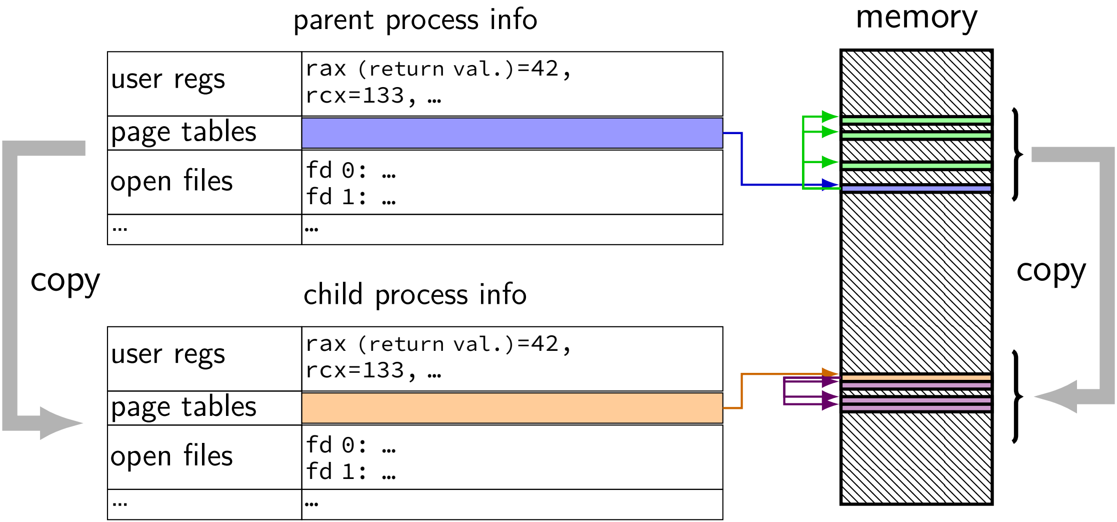
user regs	rax (return val.)=42, rcx=133, …
page tables	🟦
open files	fd 0: … fd 1: …
…	…

copy

child process info

user regs	rax (return val.)=42, rcx=133, …
page tables	🟧
open files	fd 0: … fd 1: …
…	…

fork and process info (w/o copy-on-write)



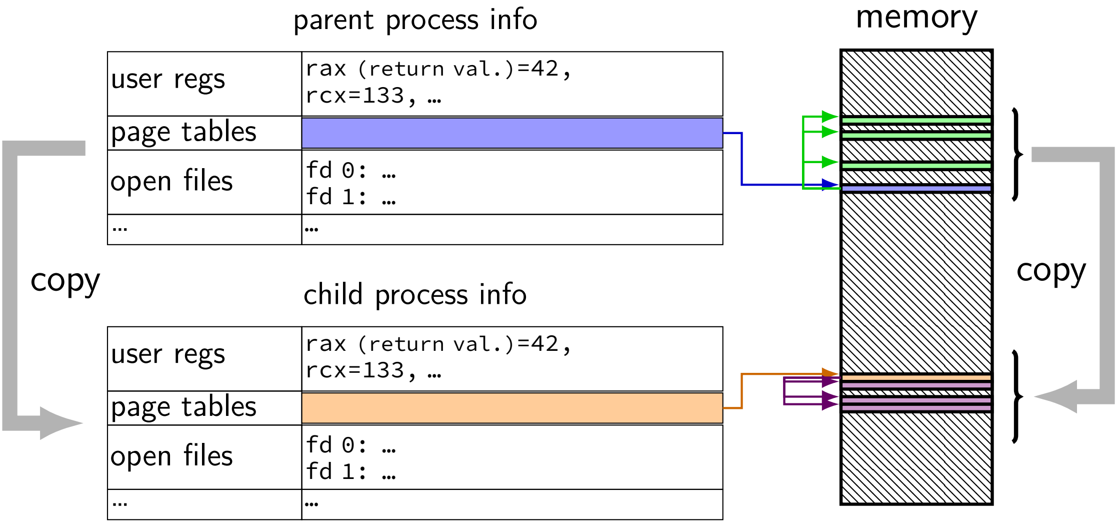
parent process info

user regs	rax (return val.)=42, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

child process info

user regs	rax (return val.)=42, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

fork and process info (w/o copy-on-write)



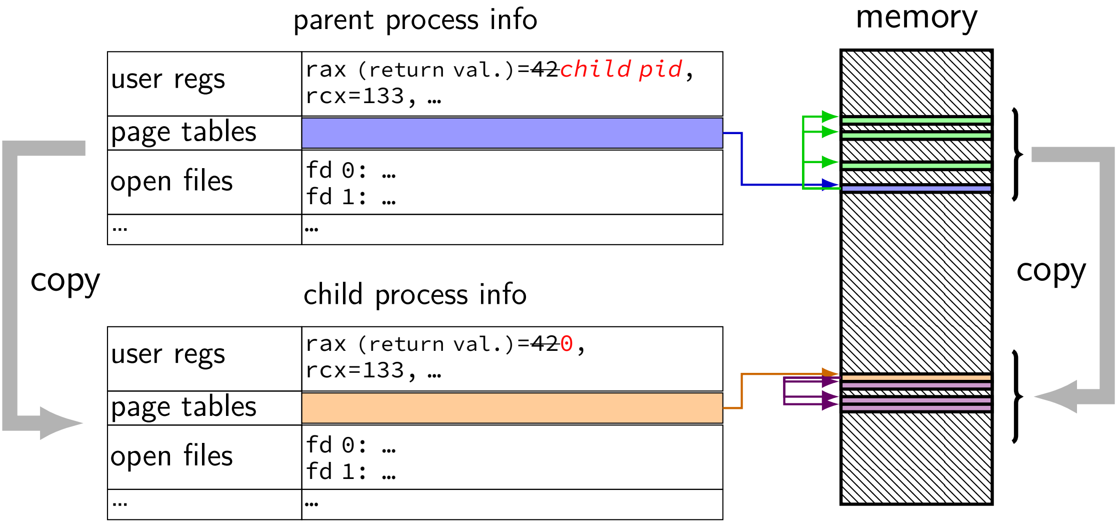
parent process info

user regs	rax (return val.)=42, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

child process info

user regs	rax (return val.)=42, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

fork and process info (w/o copy-on-write)



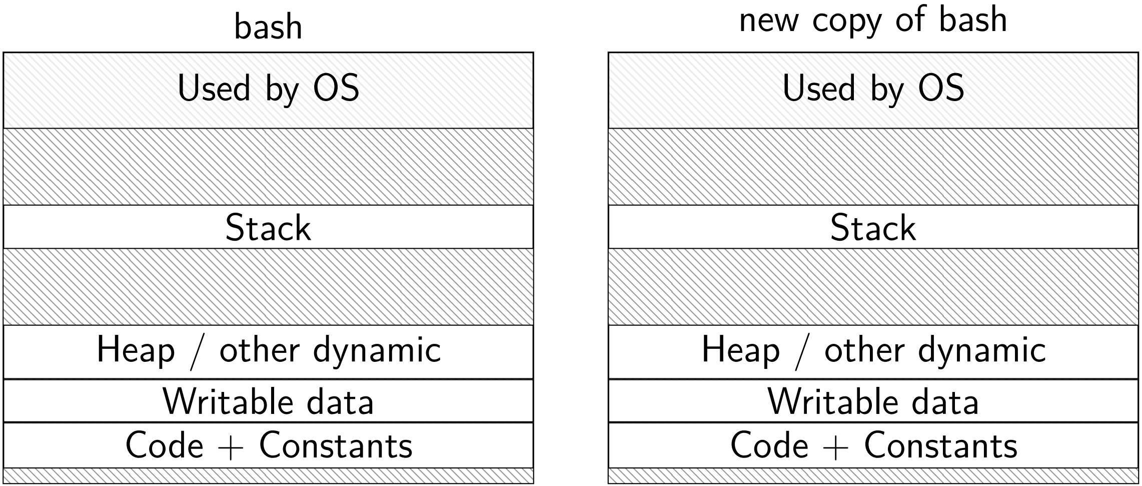
parent process info

user regs	rax (return val.)=~~42~~*child pid*, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

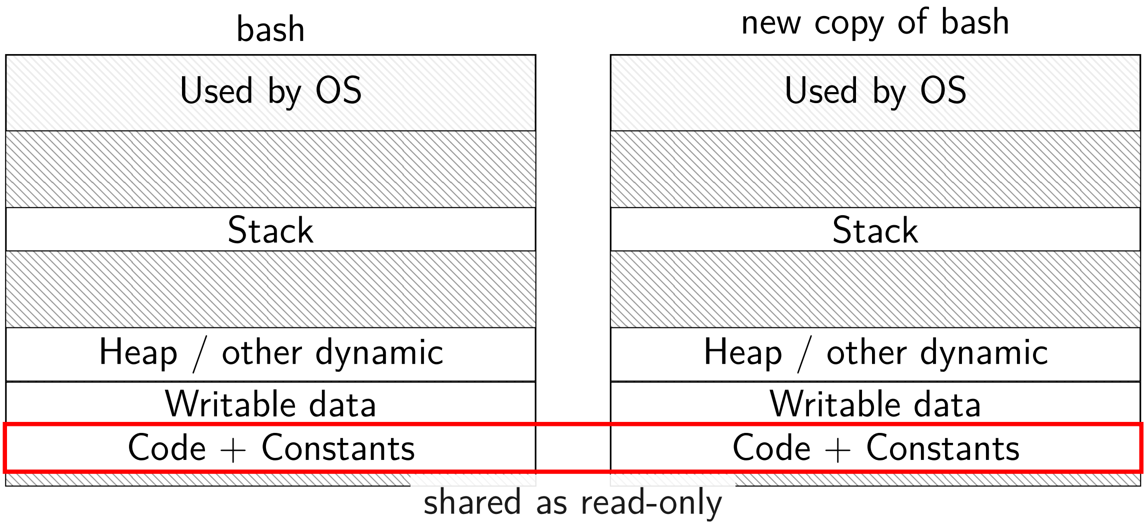
child process info

user regs	rax (return val.)=~~42~~0, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

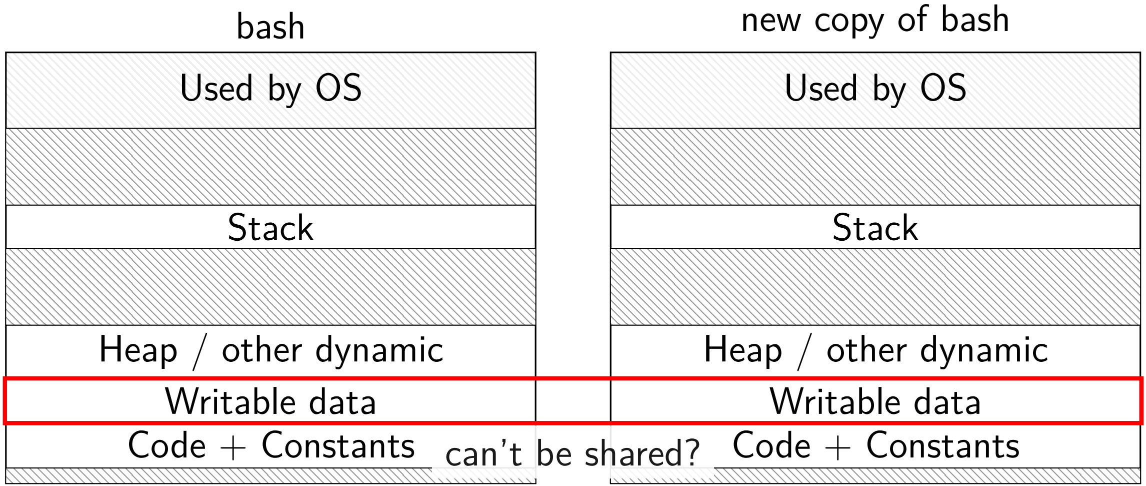
do we really need a complete copy?



do we really need a complete copy?



do we really need a complete copy?



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

fork (w/ copy-on-write, if parent writes first)

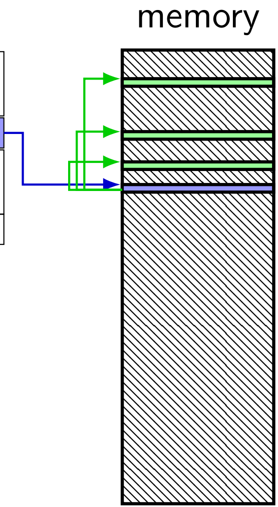
parent process info

user regs	`rax (return val.)=`~~`42`~~*`child pid`*`,`
	`rcx=133, …`

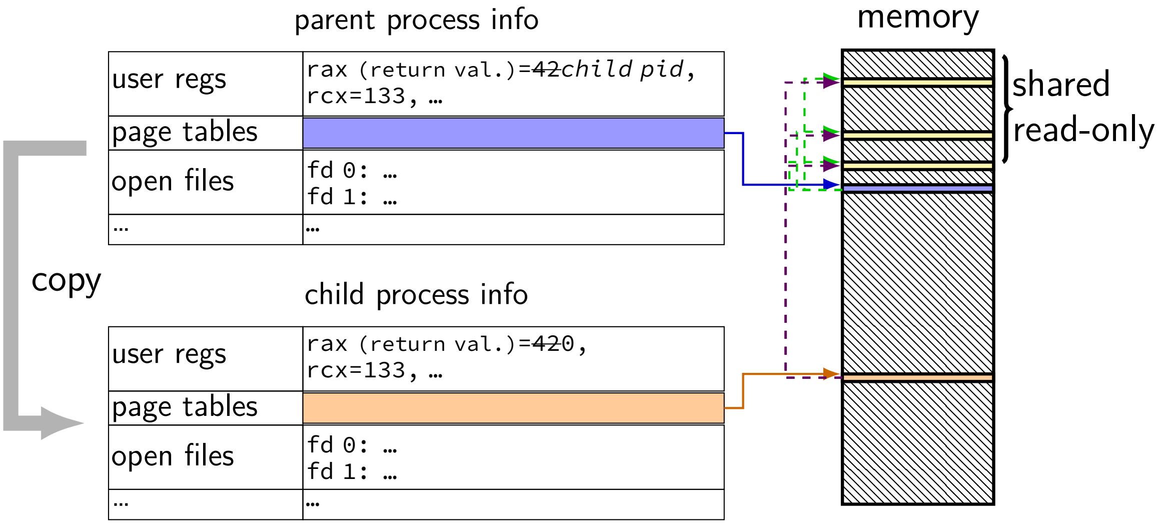
page tables	

open files	`fd 0: …`
	`fd 1: …`

…	…



fork (w/ copy-on-write, if parent writes first)



parent process info

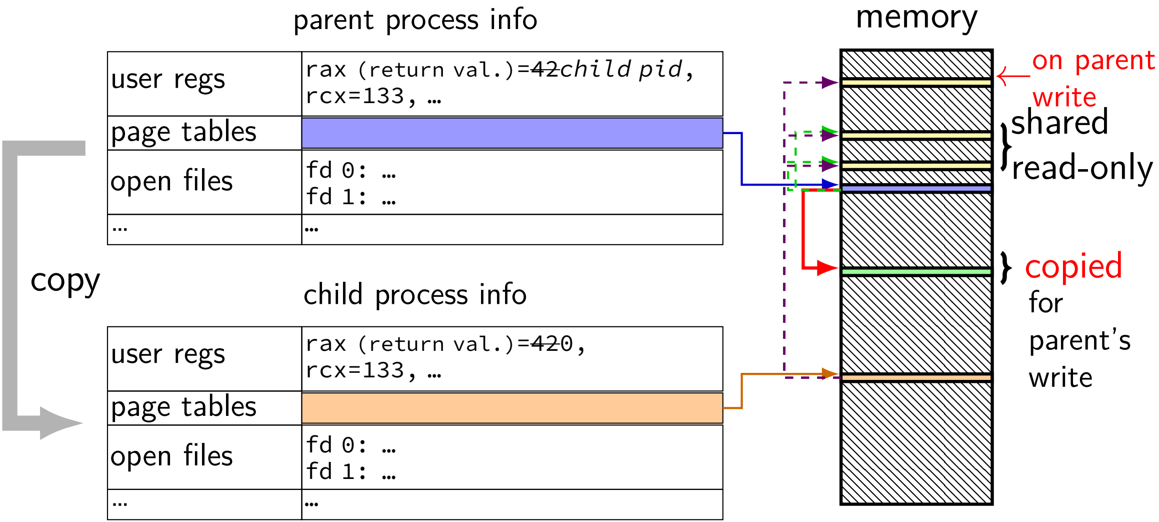
user regs	rax (return val.)=~~42~~*child pid*, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

copy

child process info

user regs	rax (return val.)=~~42~~0, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

fork (w/ copy-on-write, if parent writes first)



parent process info

user regs	rax (return val.)=~~42~~*child pid*, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

copy

child process info

user regs	rax (return val.)=~~42~~0, rcx=133, …
page tables	
open files	fd 0: … fd 1: …
…	…

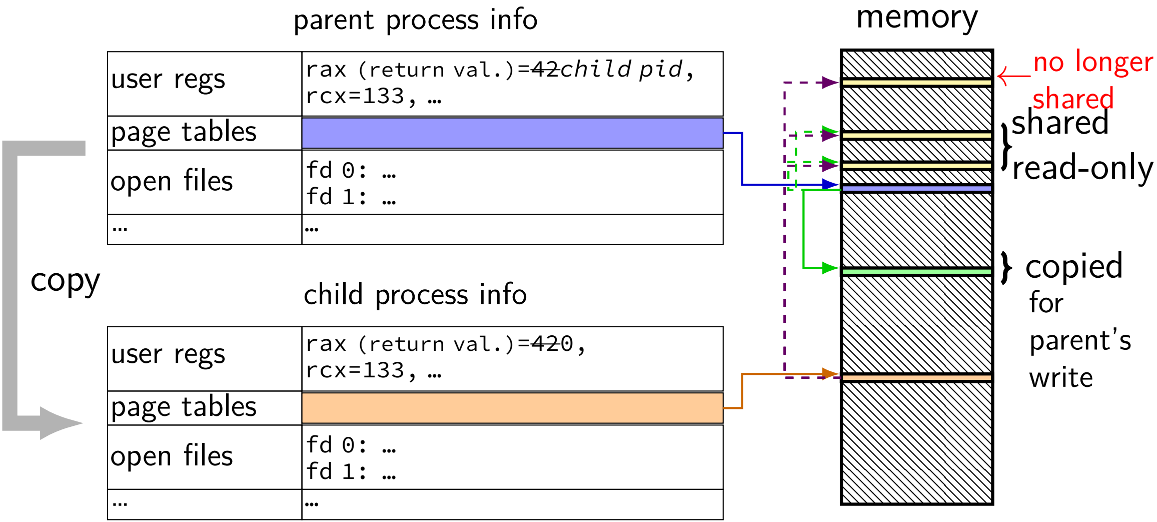
memory

on parent write

shared read-only

copied for parent's write

fork (w/ copy-on-write, if parent writes first)



parent process info

user regs	`rax (return val.)=`~~`42`~~*`child pid`*`,`
	`rcx=133, …`

page tables	🟦

open files	`fd 0: …`
	`fd 1: …`

…	…

child process info

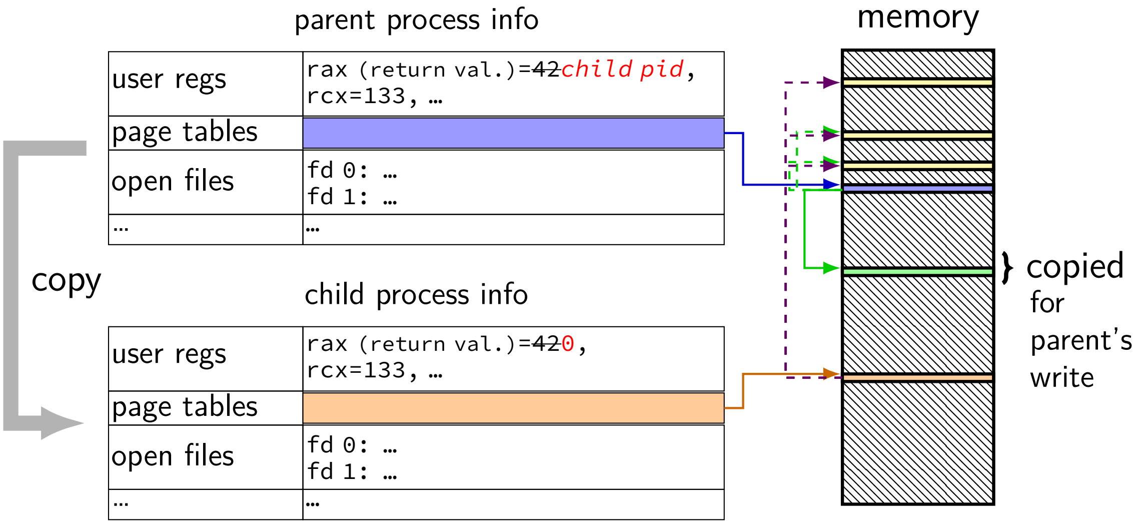
user regs	`rax (return val.)=`~~`42`~~`0,`
	`rcx=133, …`

page tables	🟧

open files	`fd 0: …`
	`fd 1: …`

…	…

fork (w/ copy-on-write, if parent writes first)



parent process info

user regs	`rax (return val.)=`~~`42`~~*`child pid`*`,`
	`rcx=133, …`

page tables	

open files	`fd 0: …`
	`fd 1: …`

…	…

copy

child process info

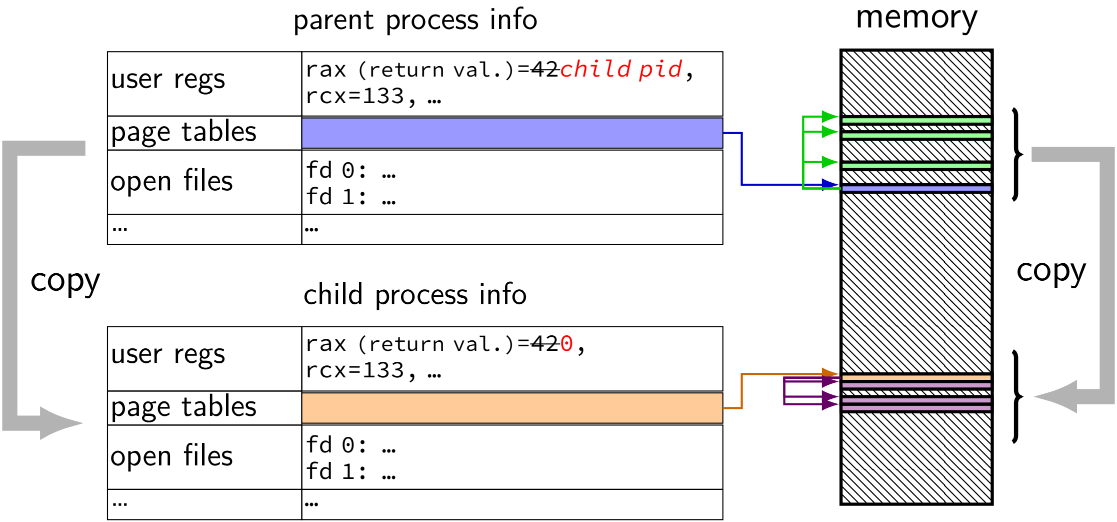
user regs	`rax (return val.)=`~~`42`~~`0,`
	`rcx=133, …`

page tables	

open files	`fd 0: …`
	`fd 1: …`

…	…

fork and process info (w/o copy-on-write)



parent process info

user regs	`rax (return val.)=`~~`42`~~*child pid*`,`
	`rcx=133, …`

page tables	

open files	`fd 0: …`
	`fd 1: …`

…	…

child process info

user regs	`rax (return val.)=`~~`42`~~`0,`
	`rcx=133, …`

page tables	

open files	`fd 0: …`
	`fd 1: …`

…	…

fork example

```
// not shown: #include various headers
int main(int argc, char *argv[]) {
    pid_t pid = getpid();
    printf("Parent pid: %d\n", (int) pid);
    pid_t child_pid = fork();
    if (child_pid > 0) {
        /* Parent Process */
        pid_t my_pid = getpid();
        printf("[%d] parent of [%d]\n",
               (int) my_pid,
               (int) child_pid);
    } else if (child_pid == 0) {
        /* Child Process */
        pid_t my_pid = getpid();
        printf("[%d] child\n",
               (int) my_pid);
    } else {
        perror("Fork failed");
    }
    return 0;
}
```

fork example

```
// not shown: #include various headers
int main(int argc, char *argv[]) {
    pid_t pid = getpid();
    printf("Parent pid: %d\n",
    pid_t child_pid = fork();
    if (child_pid > 0) {
        /* Parent Process */
        pid_t my_pid = getpid();
        printf("[%d] parent of [%d]\n",
               (int) my_pid,
               (int) child_pid);
    } else if (child_pid == 0) {
        /* Child Process */
        pid_t my_pid = getpid();
        printf("[%d] child\n",
               (int) my_pid);
    } else {
        perror("Fork failed");
    }
    return 0;
}
```

`getpid` — returns current process pid

fork example

```
// not shown: #include various headers
int main(int argc, char *argv[]) {
    pid_t pid 
    printf("Pa
    pid_t chil
    if (child_
        /* Par
        pid_t my_pid = getpid();
        printf("[%d] parent of [%d]\n",
               (int) my_pid,
               (int) child_pid);
    } else if (child_pid == 0) {
        /* Child Process */
        pid_t my_pid = getpid();
        printf("[%d] child\n",
               (int) my_pid);
    } else {
        perror("Fork failed");
    }
    return 0;
}
```

cast in case `pid_t` isn't int

POSIX doesn't specify (some systems it is, some not…)

(not necessary if you were using C++'s cout, etc.)

fork example

```
// not shown: #include various headers
int main(int argc, char *argv[]) {
    pid_
    prin
    pid_
    if (
        pid_t my_pid = getpid();
        printf("[%d] parent of [%d]\n",
               (int) my_pid,
               (int) child_pid);
    } else if (child_pid == 0) {
        /* Child Process */
        pid_t my_pid = getpid();
        printf("[%d] child\n",
               (int) my_pid);
    } else {
        perror("Fork failed");
    }
    return 0;
}
```

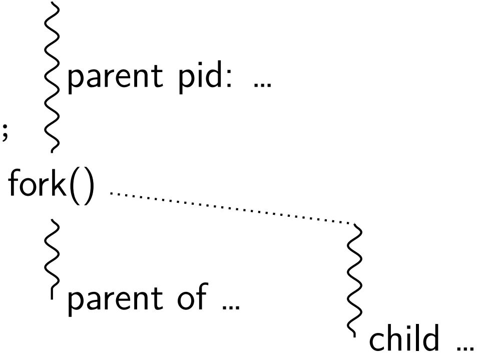
prints out `Fork failed: `*`error message`*

(example *error message*: “Resource temporarily unavailable”)

from error number stored in special global variable `errno`

fork example

```
// not shown: #include various headers
int main(int argc, char *argv[]) {
    pid_t pid = getpid();
    printf("Parent pid: %d\n", (int) pid);
    pid_t child_pid = fork();
    if (child_pid > 0) {
        /* Parent Process */
        pid_t my_pid = getpid();
        printf("[%d] parent of [%d]\n",
               (int) my_pid,
               (int) child_pid);
    } else if (child_pid == 0) {
        /* Child Process */
        pid_t my_pid = getpid();
        printf("[%d] child\n",
               (int) my_pid);
    } else {
        perror("Fork failed");
    }
    return 0;
}
```



parent pid: ...

fork()

parent of ...

child ...

Example output:

```
Parent pid: 100
[100] parent of [432]
[432] child
```

a fork question

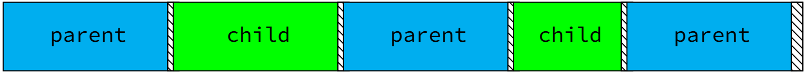
```
int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("In child\n");
    } else {
        printf("Child %d\n", pid);
    }
    printf("Done!\n");
}
```

Exercise: Suppose the pid of the parent process is 99 and child is 100. Give **two** possible outputs. (Assume no crashes, etc.)

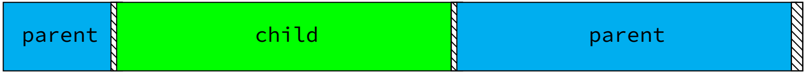
a fork question

```
int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("In child\n");
    } else {
        printf("Child %d\n", pid);
    }
    printf("Done!\n");
}
```

Exercise: Suppose the pid of the parent process is 99 and child is 100. Give **two** possible outputs. (Assume no crashes, etc.)



```
Child 100
In child
Done!
Done!
```



```
In child
Done!
Child 100
Done!
```

POSIX process management

essential operations

process information: `getpid`

process creation: `fork`

running programs: `exec*`

> also `posix_spawn` (not widely supported), …

waiting for processes to finish: `waitpid` (or `wait`)

process destruction, 'signaling': `exit`, `kill`

exec*

exec* — replace current program with new program

- * — multiple variants
- same pid, new process image

```
int execv(const char *path, const char
**argv)
```

- path: new program to run
- argv: array of arguments, termianted by null pointer

also other variants that take argv in different form and/or environment variables*

*environment variables = list of key-value pairs

execv example

```
...
child_pid = fork();
if (child_pid == 0) {
  /* child process */
  char *args[] = {"ls", "-l", NULL};
  execv("/bin/ls", args);
  /* execv doesn't return when it works.
     So, if we got here, it failed. */
  perror("execv");
  exit(1);
} else if (child_pid > 0) {
  /* parent process */
  ...
}
```

execv example

```
...
child_pid = fork();
if (child_pid == 0) {
  /* child process */
  char *args[] = {"ls", "-l", NULL};
  execv("/bin/ls", args);
  /* execv doesn't return when it works.
     So, if we got
  perror("execv");
  exit(1);
} else if (child_p
  /* parent proces
  ...
}
```

used to compute argv, argc
when program's `main` is run

convention: first argument is program name

execv example

```
...
child_pid = fork();
if (child_pid == 0) {
  /* child process */
  char *args[] = {"ls", "-l", NULL};
  execv("/bin/ls", args);
  /* execv doesn't return when it works.
     So, if we got here,
  perror("execv");
  exit(1);
} else if (child_pid > 0
  /* parent process */
  ...
}
```

path of executable to run
need not match first argument
(but probably should match it)

on Unix /bin is a directory
containing many common programs,
including `ls` ('list directory')

exec in the kernel

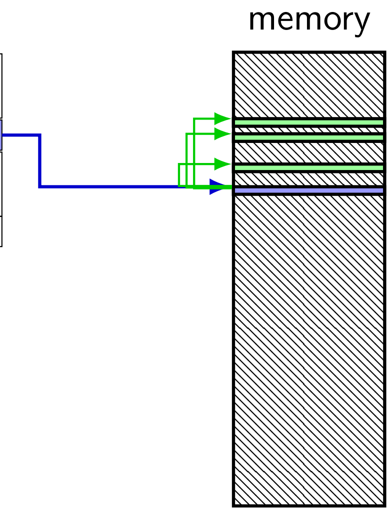
the process control block

user regs	`eax=42,`
	`ecx=133, …`

pagetables	

open files	`fd 0: (terminal …)`
	`fd 1: …`

…	…



exec in the kernel

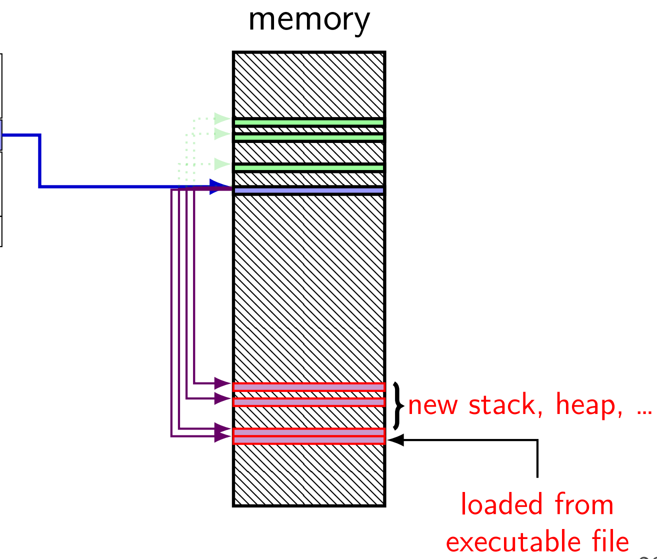
the process control block

user regs	eax=~~42~~*init. val.*,
	ecx=~~133~~*init. val.*, …

pagetables	

open files	`fd 0: (terminal …)`
	`fd 1: …`

…	…



exec in the kernel

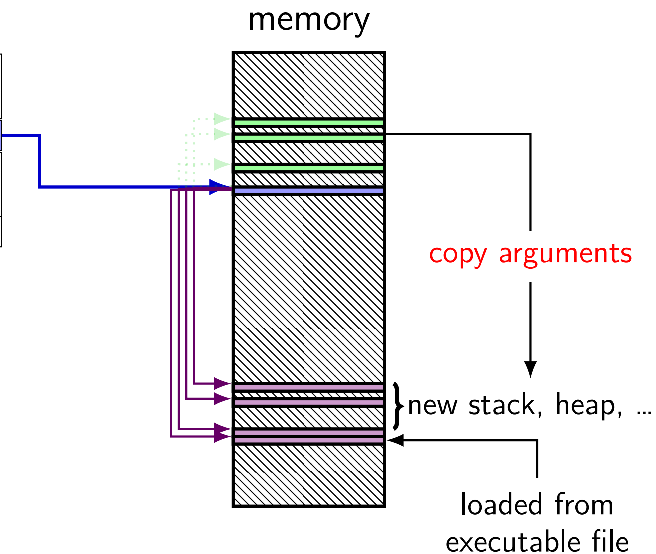
the process control block

user regs	eax=~~42~~*init. val.*,
	ecx=~~133~~*init. val.*, …

pagetables	

open files	`fd 0: (terminal …)`
	`fd 1: …`

…	…



memory

copy arguments

new stack, heap, …

loaded from executable file

exec in the kernel

the process control block

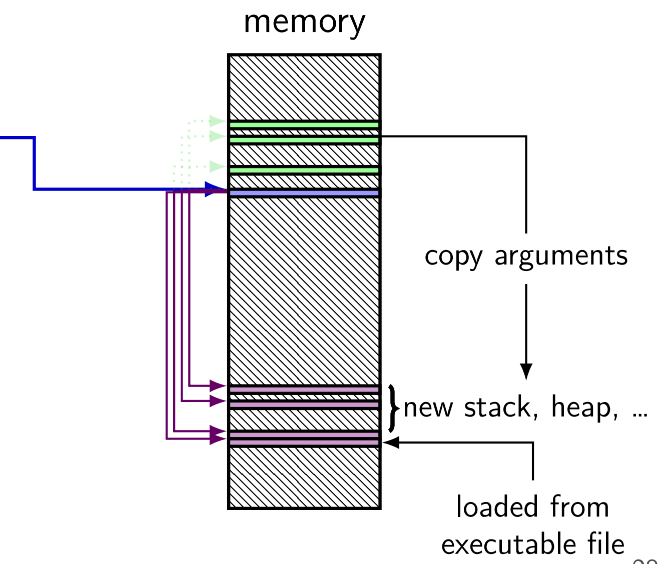
user regs	eax=~~42~~*init. val.*,
	ecx=~~133~~*init. val.*, …

pagetables	

open files	`fd 0: (terminal …)`
	`fd 1: …`

…	…

not changed!
(more on this later)



exec in the kernel

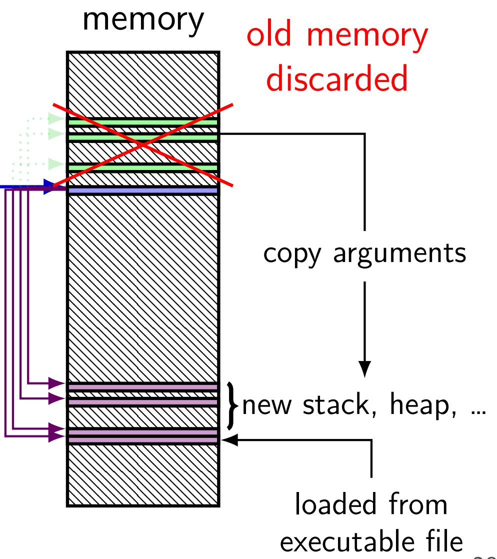
the process control block

user regs	`eax=`~~`42`~~*init. val.*`,` `ecx=`~~`133`~~*init. val.*, …
pagetables	

open files	`fd 0: (terminal …)`
	`fd 1: …`

…	…

not changed!
(more on this later)



why fork/exec?

could just have a function to spawn a new program

- Windows `CreateProcess()`; POSIX's (rarely used) `posix_spawn`

some other OSs do this (e.g. Windows)

needs to include API to set new program's state

- e.g. without fork: either:
- need function to set new program's current directory, *or*
- need to change your directory, then start program, then change back
- e.g. with fork: just change your current directory before exec

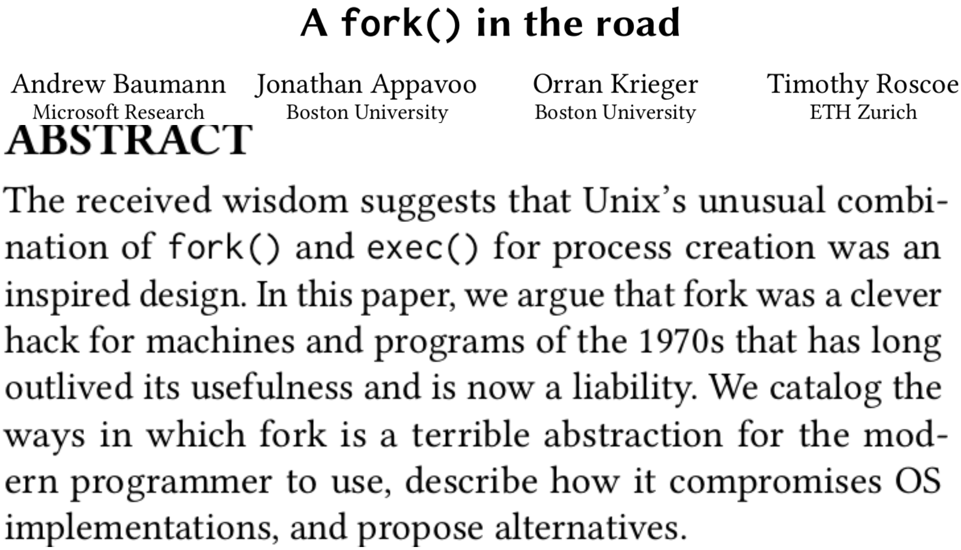
but allows OS to avoid 'copy everything' code

- probably makes OS implementation easier

posix_spawn

```
pid_t new_pid;
const char argv[] = { "ls", "-l", NULL };
int error_code = posix_spawn(
    &new_pid,
    "/bin/ls",
    NULL /* null = copy current process's open files;
            if not null, do something else */,
    NULL /* null = no special settings for new process */,
    argv,
    NULL /* null = copy current process's "environment variab
            if not null, do something else */
);
if (error_code == 0) {
    /* handle error */
}
```

some opinions (via HotOS '19)



A fork() in the road

Andrew Baumann
Microsoft Research

Jonathan Appavoo
Boston University

Orran Krieger
Boston University

Timothy Roscoe
ETH Zurich

ABSTRACT

The received wisdom suggests that Unix's unusual combination of fork() and exec() for process creation was an inspired design. In this paper, we argue that fork was a clever hack for machines and programs of the 1970s that has long outlived its usefulness and is now a liability. We catalog the ways in which fork is a terrible abstraction for the modern programmer to use, describe how it compromises OS implementations, and propose alternatives.

POSIX process management

essential operations

process information: getpid

process creation: fork

running programs: exec*

also posix_spawn (not widely supported), ...

waiting for processes to finish: waitpid (or wait)

process destruction, 'signaling': exit, kill

wait/waitpid

```
pid_t waitpid(pid_t pid, int *status,
                 int options)
```

wait for a child process (with pid=`pid`) to finish

sets `*status` to its “status information”

`pid=-1` → wait for any child process instead

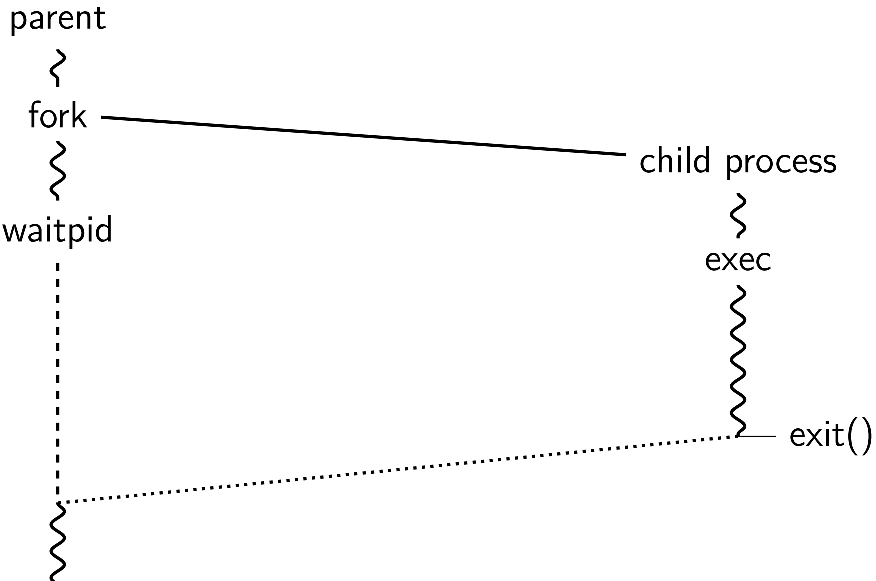
options? see manual page (command `man waitpid`)

- `0` — no options

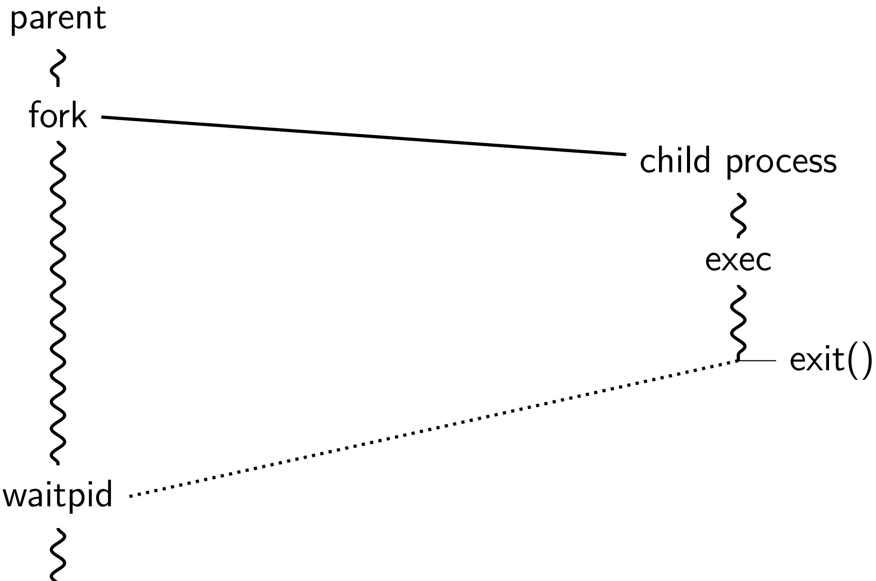
waitpid example

```
#include <sys/wait.h>
...
  child_pid = fork();
  if (child_pid > 0) {
      /* Parent process */
      int status;
      waitpid(child_pid, &status, 0);
  } else if (child_pid == 0) {
      /* Child process */
      ...
```

typical pattern



typical pattern (alt)



parent

fork

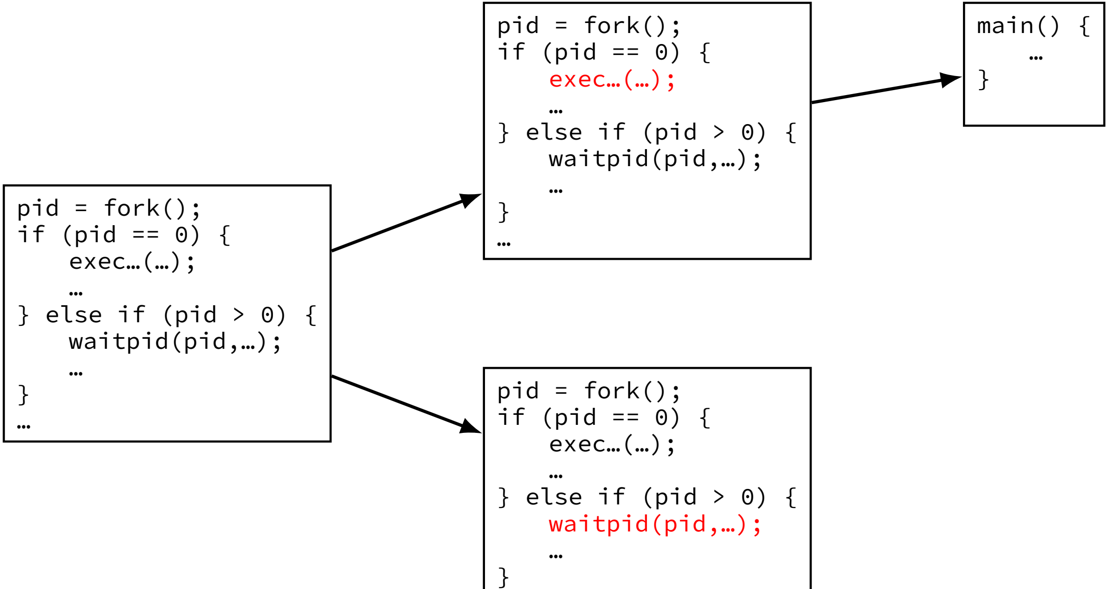
child process

exec

exit()

waitpid

typical pattern (detail)



```
pid = fork();
if (pid == 0) {
    exec…(…);
    …
} else if (pid > 0) {
    waitpid(pid,…);
    …
}
…
```

```
pid = fork();
if (pid == 0) {
    exec…(…);
    …
} else if (pid > 0) {
    waitpid(pid,…);
    …
}
…
```

```
main() {
    …
}
```

```
pid = fork();
if (pid == 0) {
    exec…(…);
    …
} else if (pid > 0) {
    waitpid(pid,…);
    …
}
```

POSIX process management

essential operations

process information: `getpid`

process creation: `fork`

running programs: `exec*`

also `posix_spawn` (not widely supported), …

waiting for processes to finish: `waitpid` (or `wait`)

process destruction, 'signaling': `exit`, `kill`

backup slides

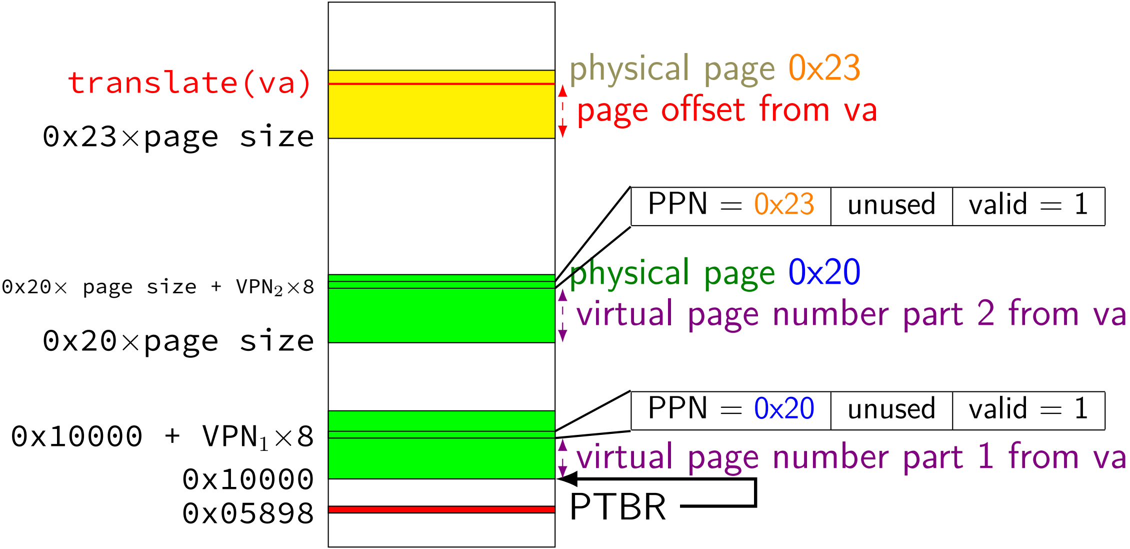
assignment part 2/3

supporting arbitrary numbers of LEVELS, POBITS

code review in lab after reading days

- limited allowed collaboration

pa = translate(va) [LEVELS=2]



translate(va)

0x23×page size

physical page 0x23

page offset from va

PPN = 0x23	unused	valid = 1

0x20× page size + VPN$_2$×8

0x20×page size

physical page 0x20

virtual page number part 2 from va

PPN = 0x20	unused	valid = 1

0x10000 + VPN$_1$×8

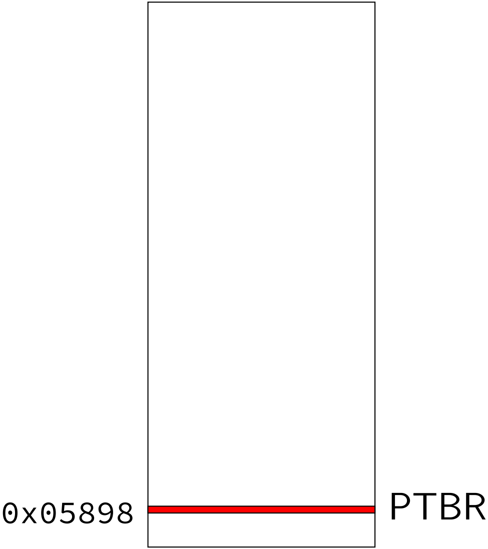
0x10000

0x05898

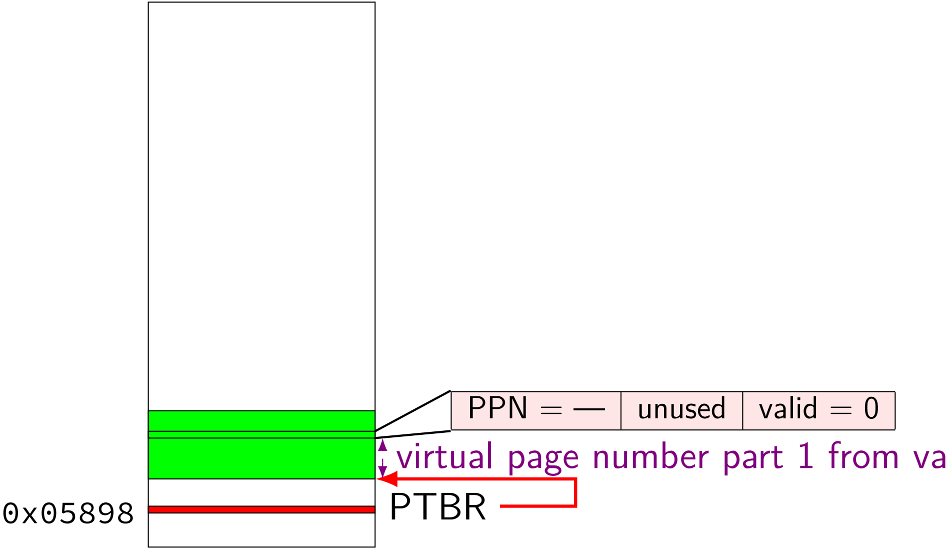
virtual page number part 1 from va

PTBR

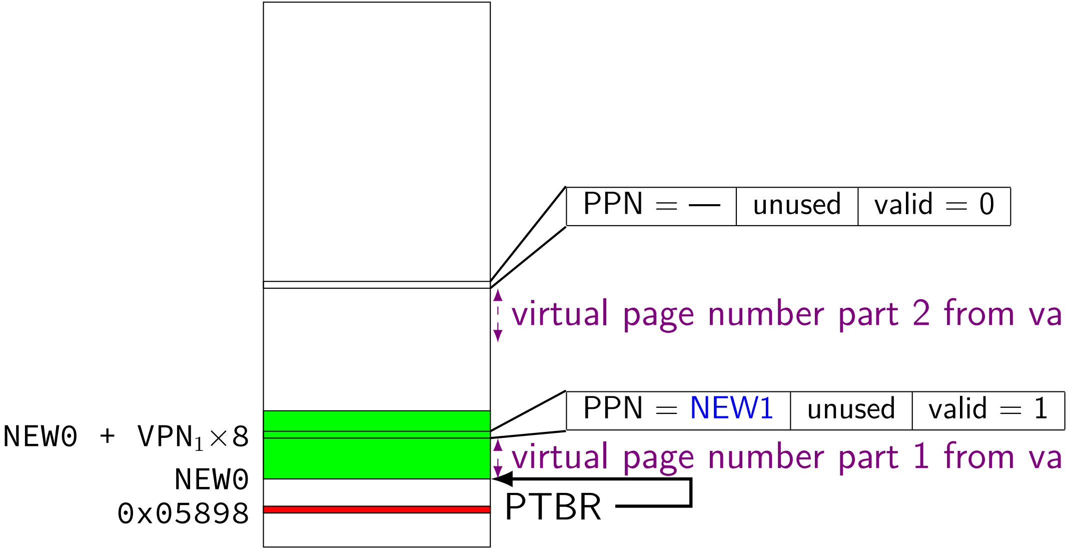
first page_allocate(va) [LEVELS=2]



first page_allocate(va) [LEVELS=2]



first page_allocate(va) [LEVELS=2]



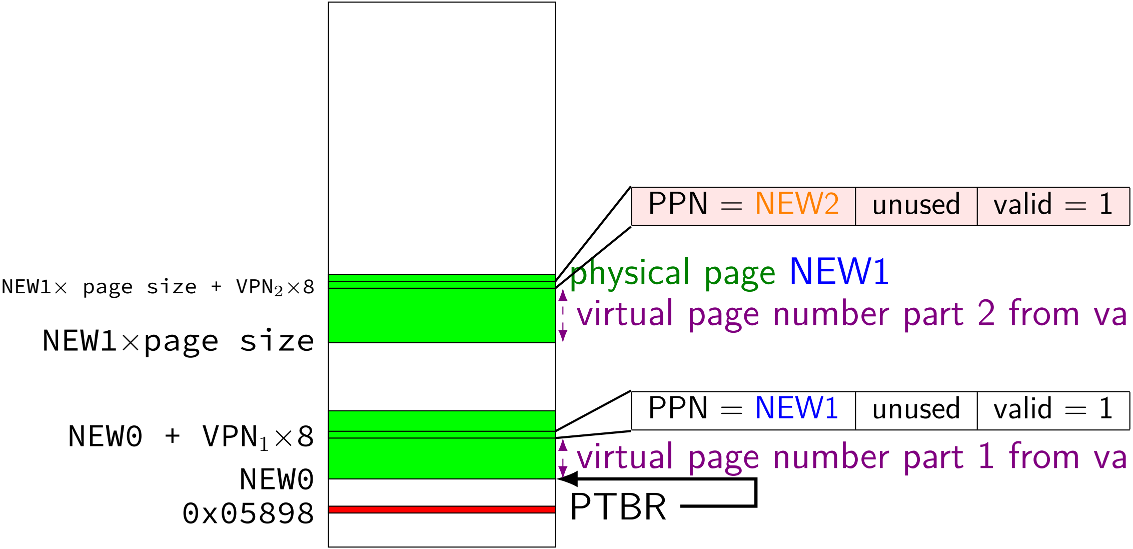
PPN = — | unused | valid = 0

virtual page number part 2 from va

PPN = NEW1 | unused | valid = 1

virtual page number part 1 from va

first page_allocate(va) [LEVELS=2]



PPN = NEW2	unused	valid = 1

physical page NEW1

virtual page number part 2 from va

PPN = NEW1	unused	valid = 1

virtual page number part 1 from va

PTBR

`NEW1× page size + VPN`$_2$`×8`

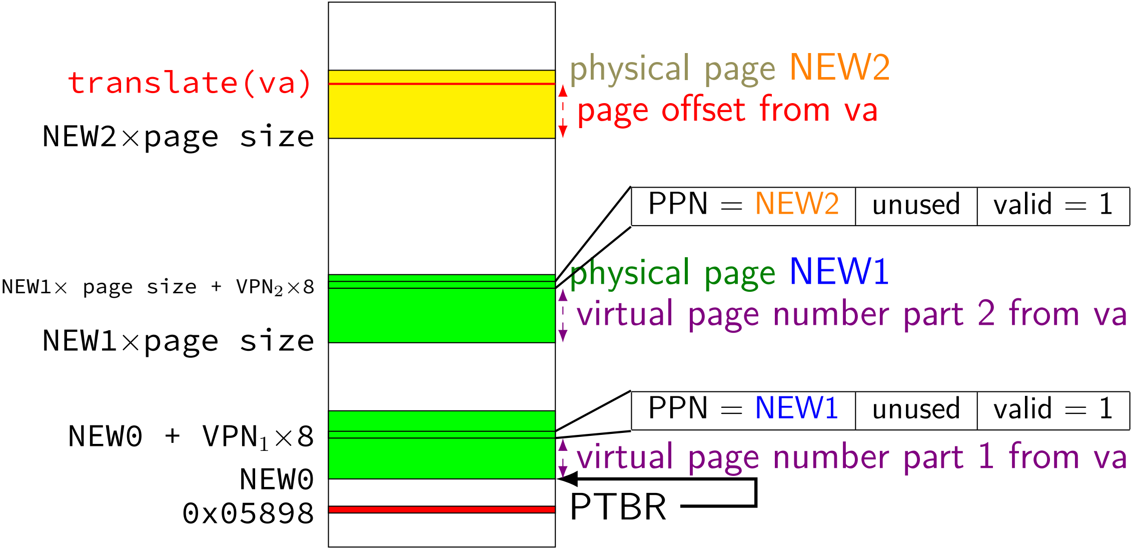
`NEW1×page size`

`NEW0 + VPN`$_1$`×8`

`NEW0`

`0x05898`

first page_allocate(va) [LEVELS=2]



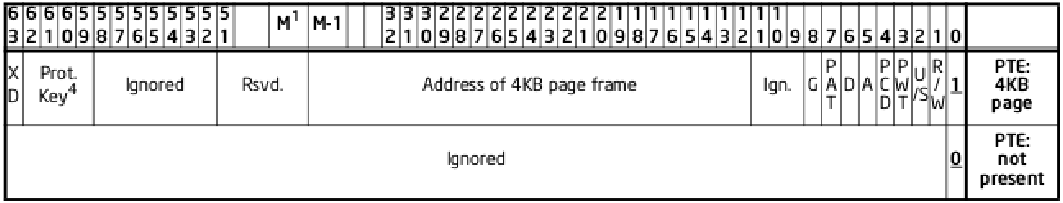
later page allocates?

some of those allocations done earlier

- e.g. ptbr already set

should reuse existing allocation then

x86-64 page table entries (1)



63	62 61 60 59	58 57 56 55 54 53 52	51 … M[1]	M-1 … 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

present = valid

R/W = writes allowed?

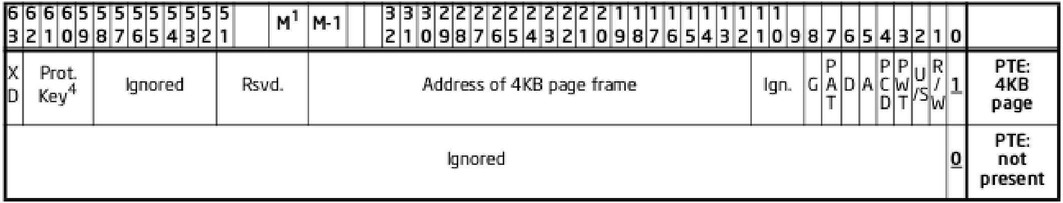
U/S = user-mode allowed? (“user/supervisor”)

XD = execute-disable?

A = accessed? (MMU sets to 1 on page read/write)

D = dirty? (MMU sets to 1 on page write)

x86-64 page table entries (1)



63	62 61 60 59	58 57 56 55 54 53 52	51 … M[1]	M-1 … 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

present = valid

R/W = writes allowed?

U/S = user-mode allowed? (“user/supervisor”)

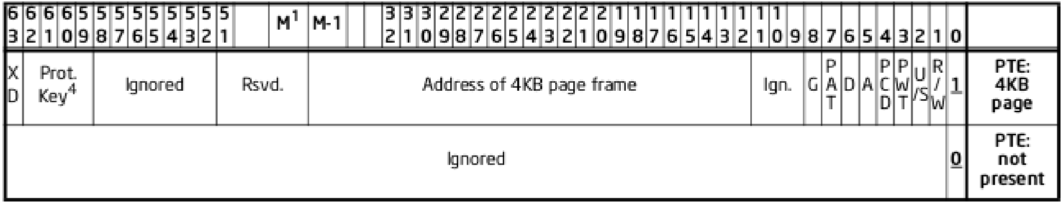
XD = execute-disable?

A = accessed? (MMU sets to 1 on page read/write)

D =

helps support replacement policies for swapping

x86-64 page table entries (1)



63	62 61 60 59	58 57 56 55 54 53 52	51 M^1	M-1 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key^4	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

present = valid

R/W = writes allowed?

U/S = user-mode allowed? (“user/supervisor”)

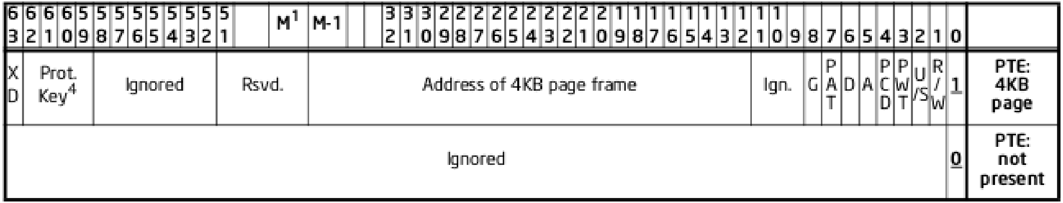
XD = execute-disable?

A = accessed? (MMU sets to 1 on page read/write)

D = dirty? (MMU sets to 1 on page write)

helps support writeback policy for swapping

x86-64 page table entries (2)



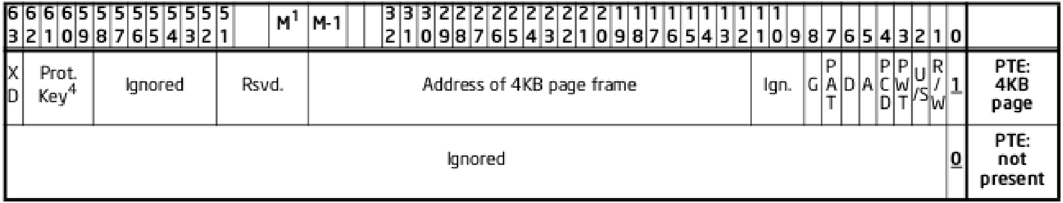
63	62 61 60 59	58 57 56 55 54 53 52	51 … M[1]	M-1 … 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

G = global? (shared between all page tables)

PWT, PCD, PAT = control how caches work when accessing physical page:

- can disable using the cache entirely
- can disable write-back (use write-through instead)
- multicore-related cache settings
- (and some other settings)

x86-64 page table entries (2)



63	62 61 60 59	58 57 56 55 54 53 52	51 … M^1	M-1 … 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

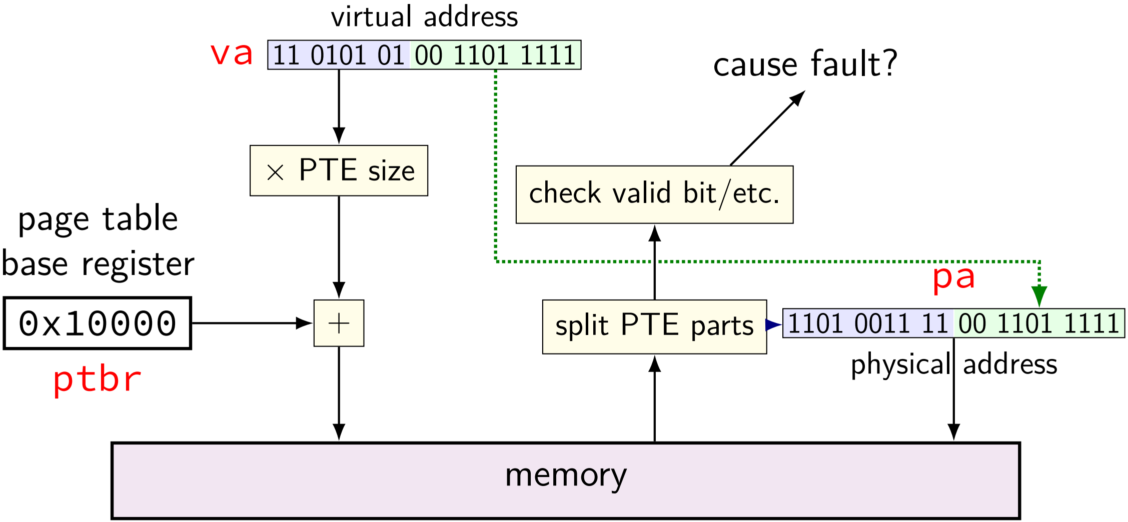
G = global? (shared between all page tables)

CPU won't evict TLB entries on most page table base registers changes

P

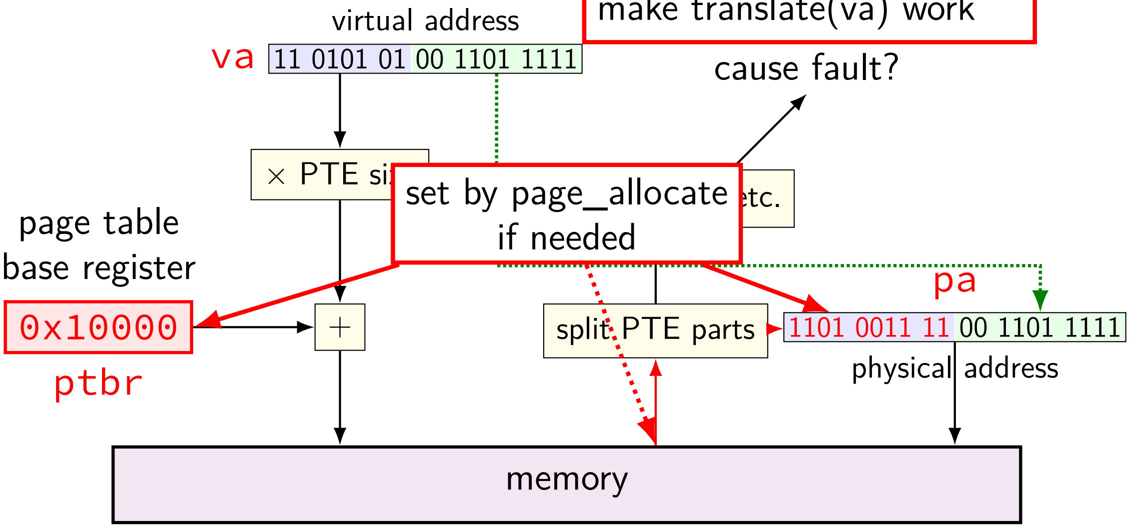
- can disable using the cache entirely
- can disable write-back (use write-through instead)
- multicore-related cache settings
- (and some other settings)

pa=translate(va)

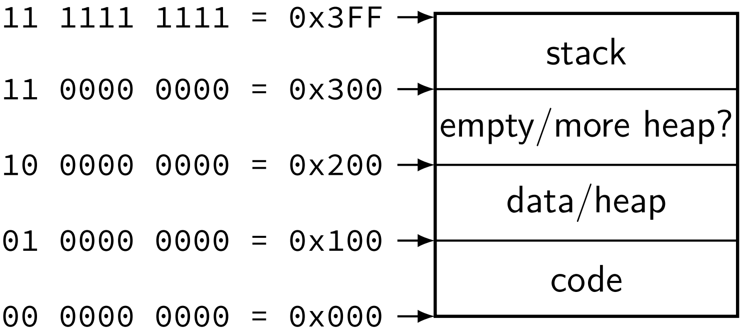


pa=translate(va)

page_allocate(va) needs to make translate(va) work



toy program memory



11 1111 1111 = 0x3FF

stack

11 0000 0000 = 0x300

empty/more heap?

10 0000 0000 = 0x200

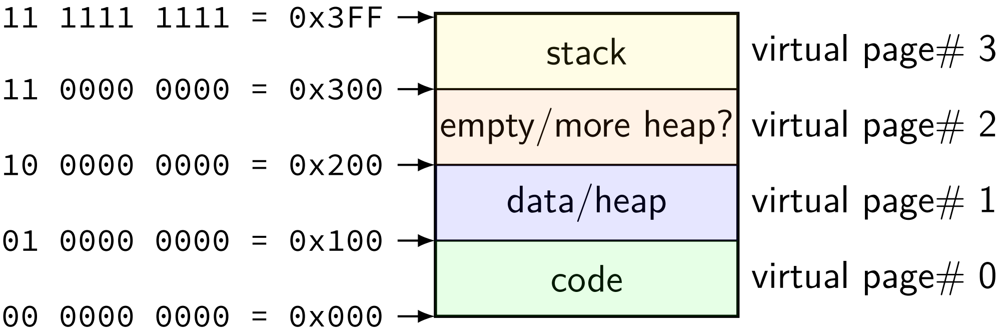
data/heap

01 0000 0000 = 0x100

code

00 0000 0000 = 0x000

toy program memory



11 1111 1111 = 0x3FF

11 0000 0000 = 0x300

10 0000 0000 = 0x200

01 0000 0000 = 0x100

00 0000 0000 = 0x000

stack

empty/more heap?

data/heap

code

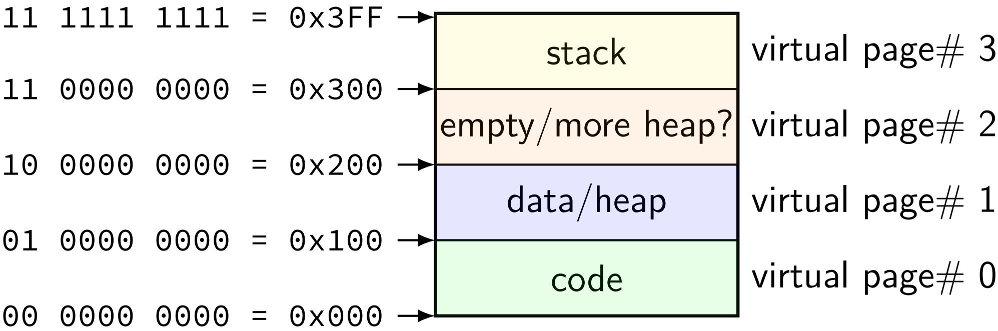
virtual page# 3

virtual page# 2

virtual page# 1

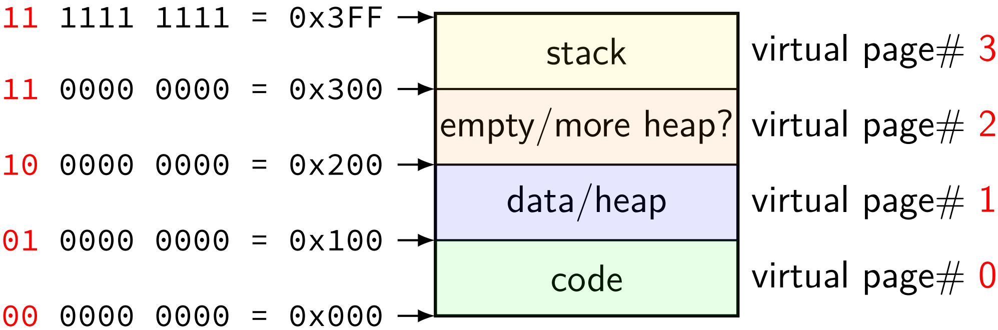
virtual page# 0

toy program memory



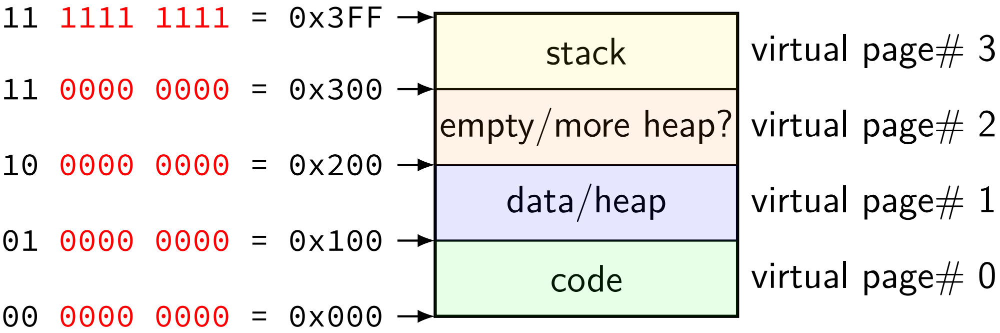
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



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

toy program memory



11 1111 1111 = 0x3FF

11 0000 0000 = 0x300

10 0000 0000 = 0x200

01 0000 0000 = 0x100

00 0000 0000 = 0x000

stack — virtual page# 3

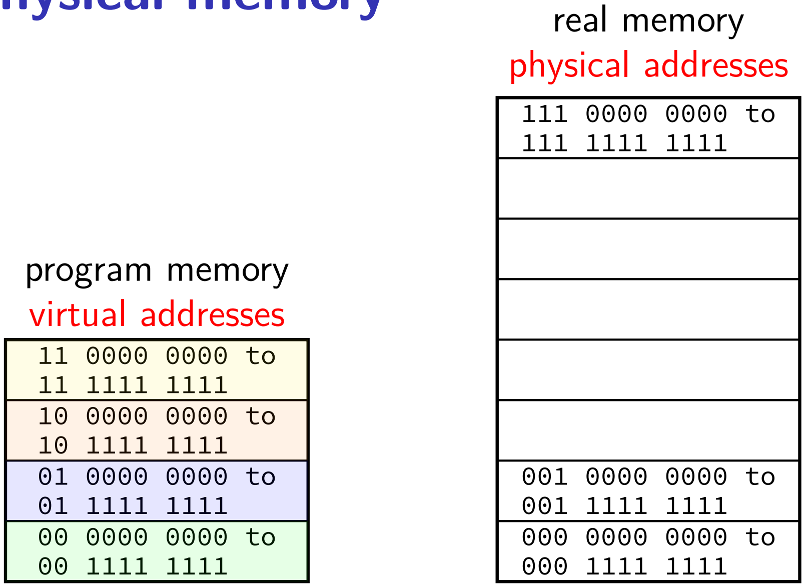
empty/more heap? — virtual page# 2

data/heap — virtual page# 1

code — virtual page# 0

rest of address is called page offset

toy physical memory



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

program memory

virtual addresses

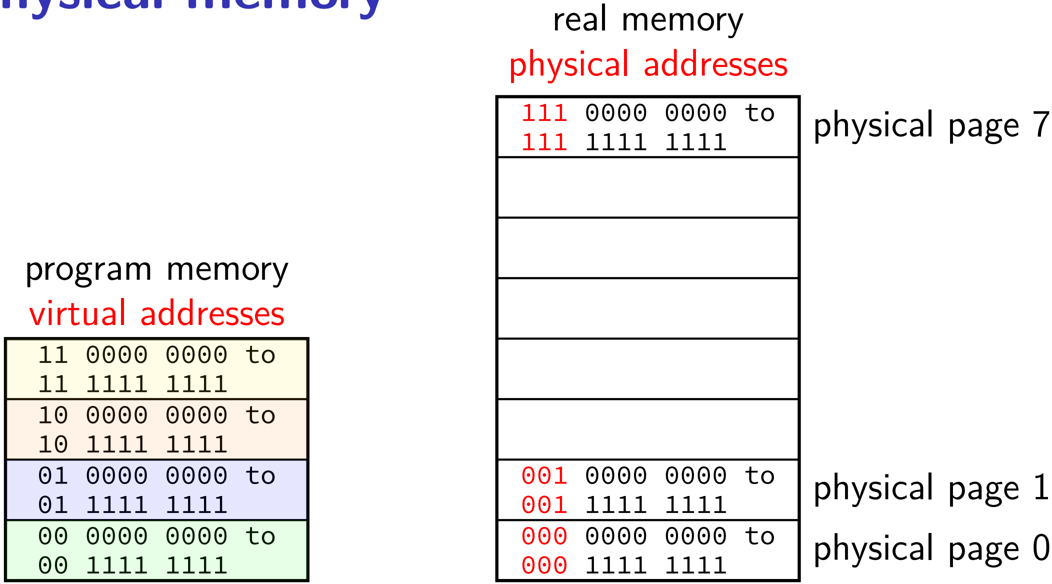
11 0000 0000 to
11 1111 1111

10 0000 0000 to
10 1111 1111

01 0000 0000 to
01 1111 1111

00 0000 0000 to
00 1111 1111

toy physical memory



real memory

physical addresses

```
111 0000 0000 to
111 1111 1111
```

physical page 7

```
001 0000 0000 to
001 1111 1111
```

physical page 1

```
000 0000 0000 to
000 1111 1111
```

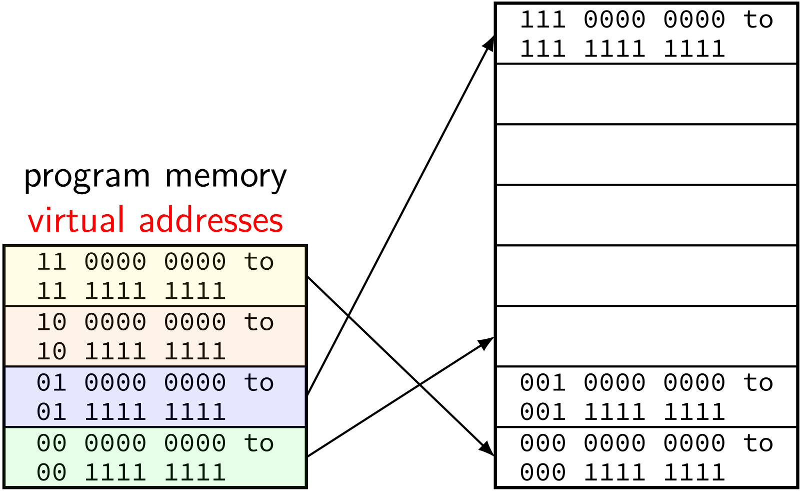
physical page 0

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
```

toy physical memory



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

program memory

virtual addresses

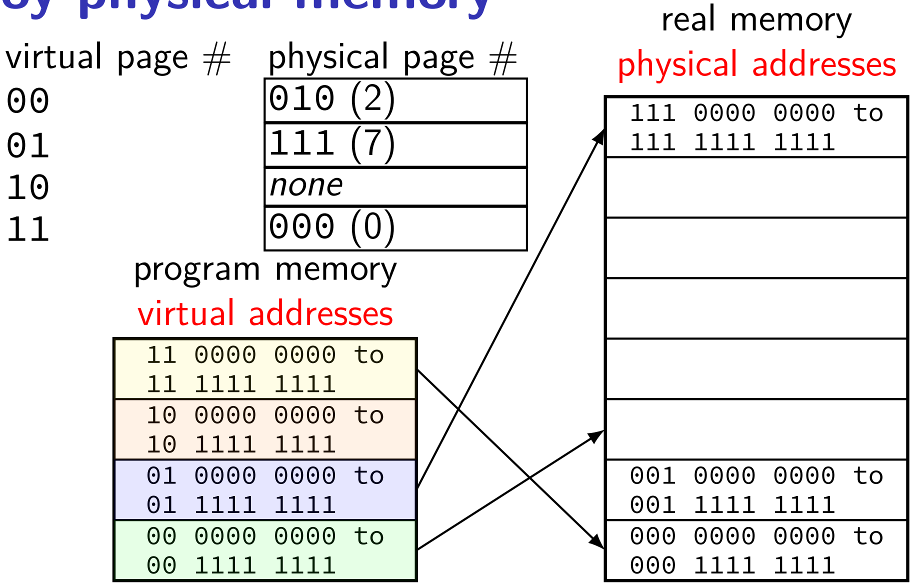
11 0000 0000 to
11 1111 1111

10 0000 0000 to
10 1111 1111

01 0000 0000 to
01 1111 1111

00 0000 0000 to
00 1111 1111

toy physical memory



virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
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

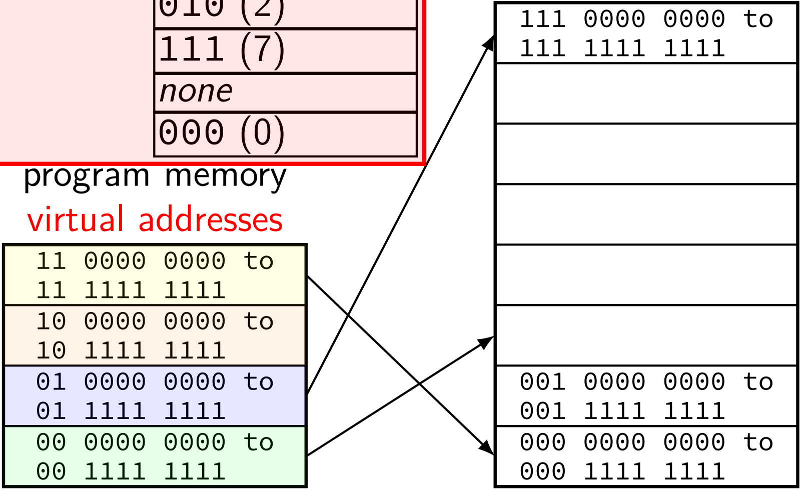
physical addresses

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

toy physical memory

page table!

virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
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

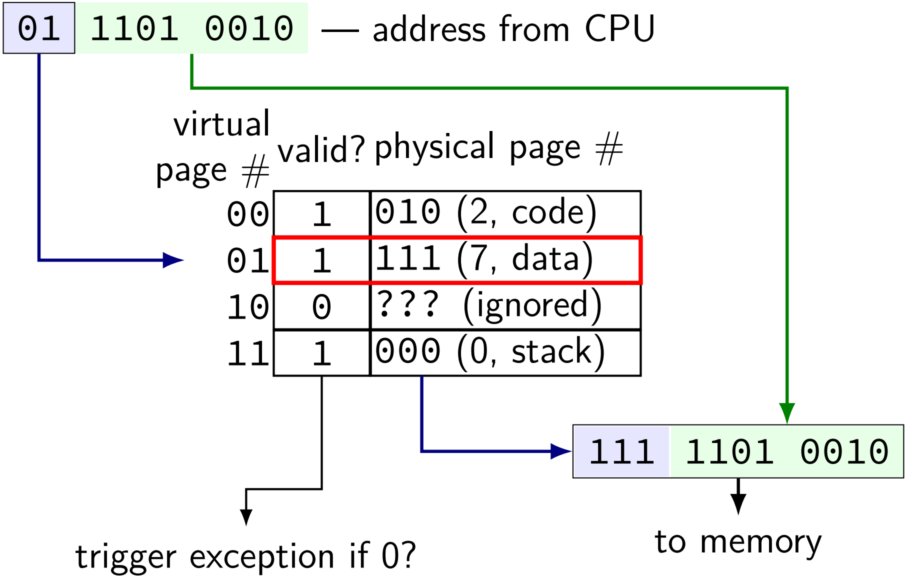
001 0000 0000 to
001 1111 1111

000 0000 0000 to
000 1111 1111

toy 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



01 1101 0010 — address from CPU

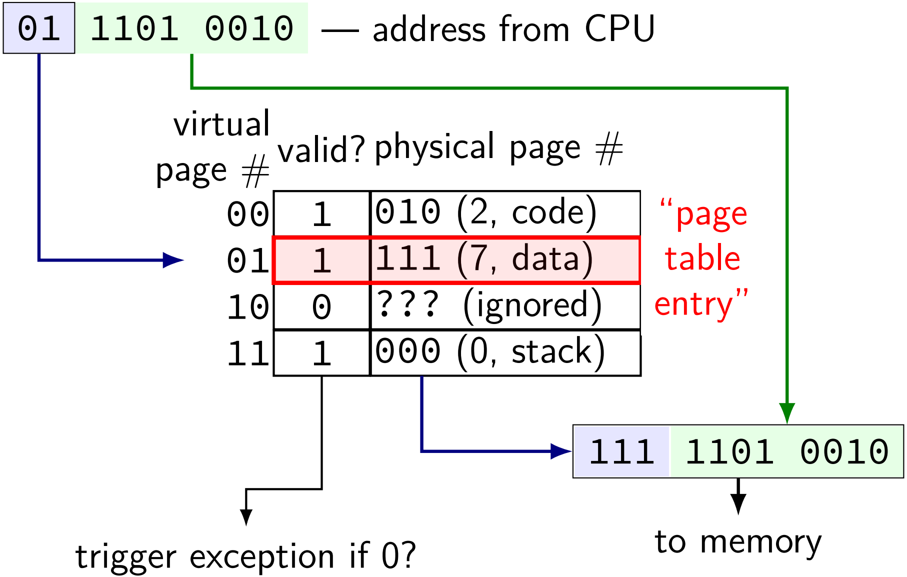
virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

111 1101 0010

trigger exception if 0?

to memory

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)

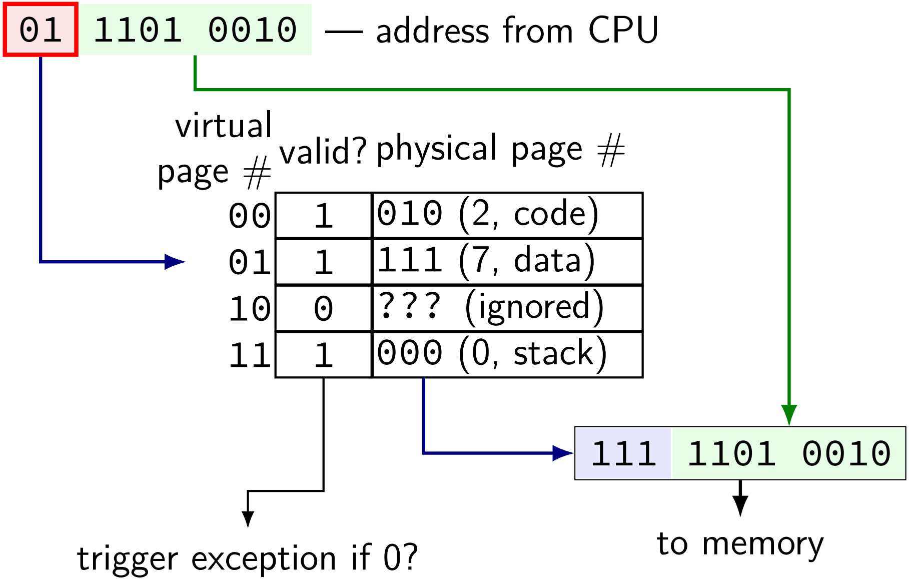
"page table entry"

111 1101 0010

to memory

trigger exception if 0?

t “virtual page number” lookup



01 1101 0010 — address from CPU

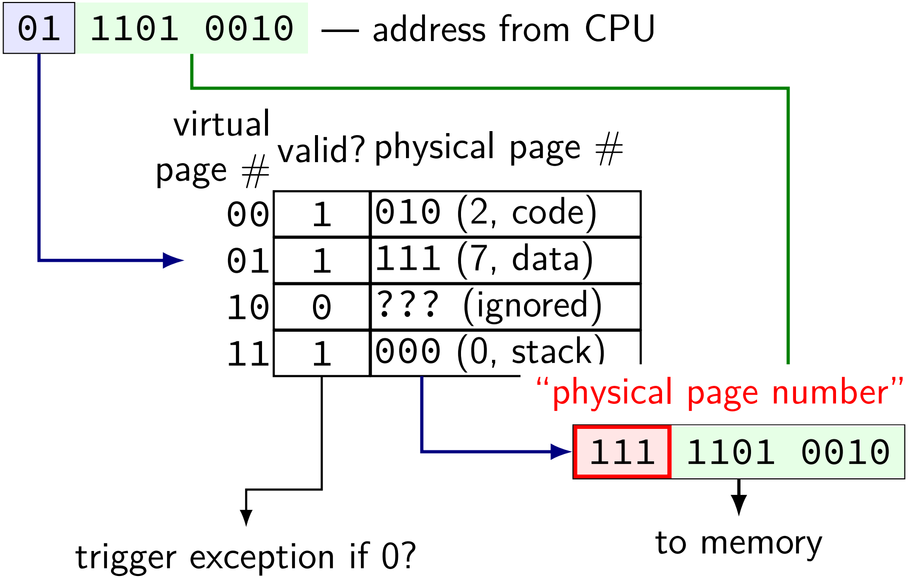
virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

111 1101 0010

trigger exception if 0?

to memory

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)

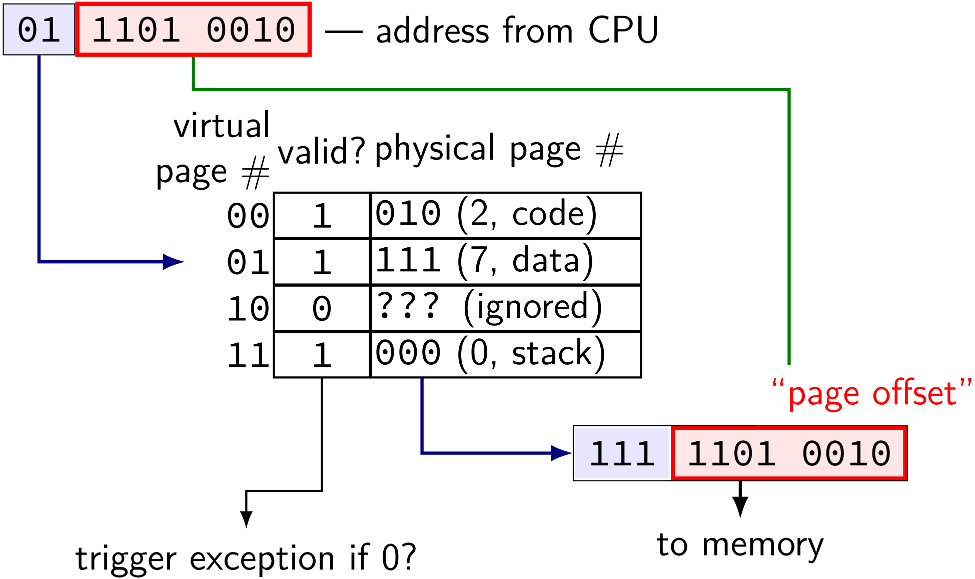
"physical page number"

111 1101 0010

trigger exception if 0?

to memory

toy pag “page offset” ookup



`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)

“page offset”

`111` `1101 0010`

to memory

trigger exception if 0?

exit statuses

```
int main() {
    return 0;  /* or exit(0);  */
}
```

the status

```
#include <sys/wait.h>
...
  waitpid(child_pid, &status, 0);
  if (WIFEXITED(status)) {
    printf("main returned or exit called with %d\n",
           WEXITSTATUS(status));
  } else if (WIFSIGNALED(status)) {
    printf("killed by signal %d\n", WTERMSIG(status));
  } else {
      ...
  }
```

“status code” encodes both return value and if exit was abnormal

W* macros to decode it

the status

```
#include <sys/wait.h>
...
  waitpid(child_pid, &status, 0);
  if (WIFEXITED(status)) {
    printf("main returned or exit called with %d\n",
           WEXITSTATUS(status));
  } else if (WIFSIGNALED(status)) {
    printf("killed by signal %d\n", WTERMSIG(status));
  } else {
      ...
  }
```

“status code” encodes both return value and if exit was abnormal

W* macros to decode it

shell

allow user (= person at keyboard) to run applications

user's wrapper around process-management functions

aside: shell forms

POSIX: command line you have used before

also: graphical shells

- e.g. OS X Finder, Windows explorer

other types of command lines?

completely different interfaces?

searching for programs

POSIX convention: PATH *environment variable*

- example: `/home/cr4bd/bin:/usr/bin:/bin`
- list of directories to check in order

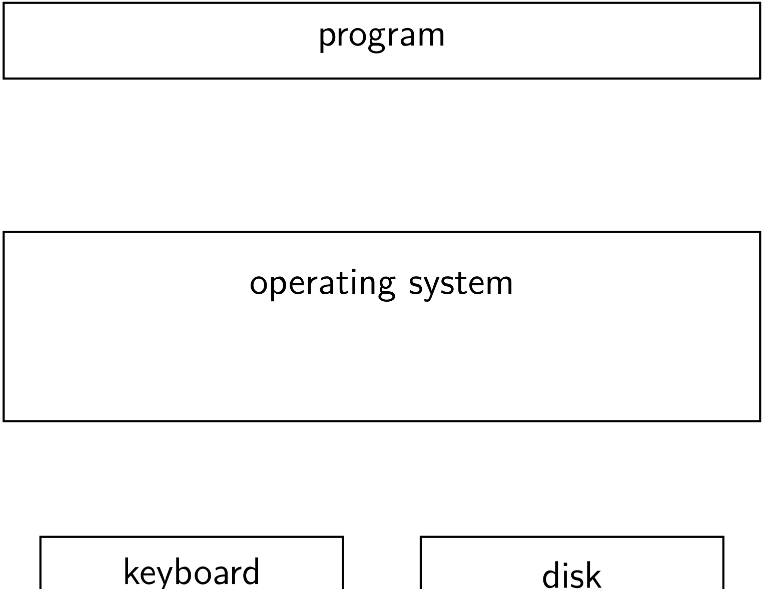
environment variables = key/value pairs stored with process

- by default, left unchanged on execve, fork, etc.

one way to implement: [pseudocode]

```
for (directory in path) {
    execv(directory + "/" + program_name, argv);
}
```

kernel buffering (reads)



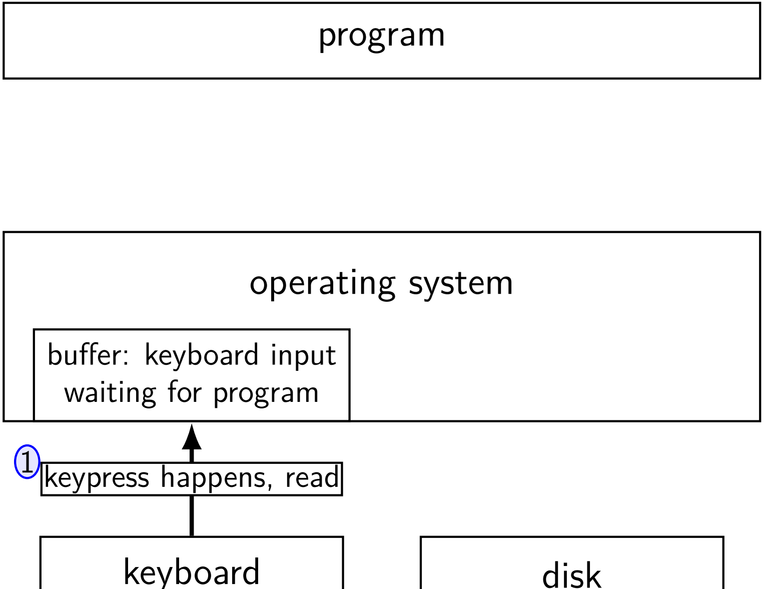
program

operating system

keyboard

disk

kernel buffering (reads)



program

operating system

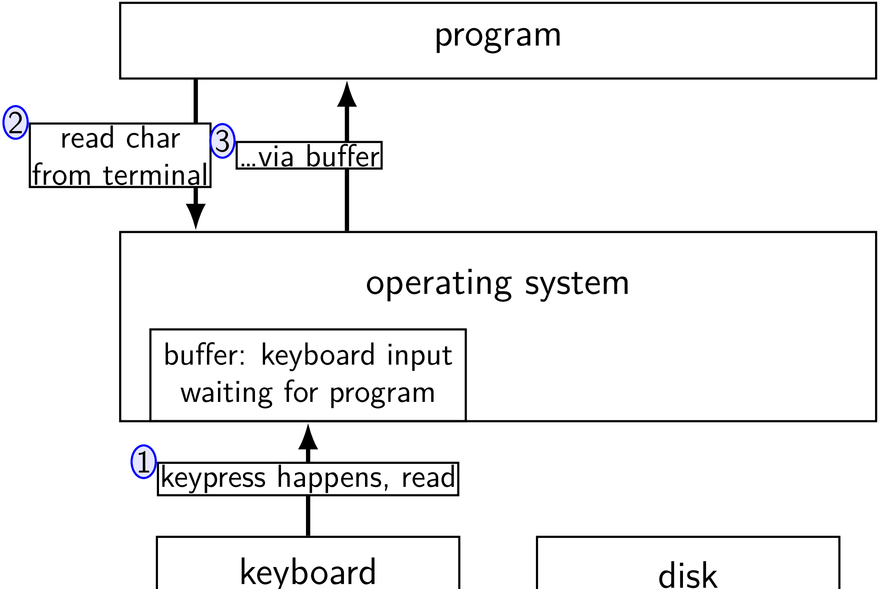
buffer: keyboard input waiting for program

1 keypress happens, read

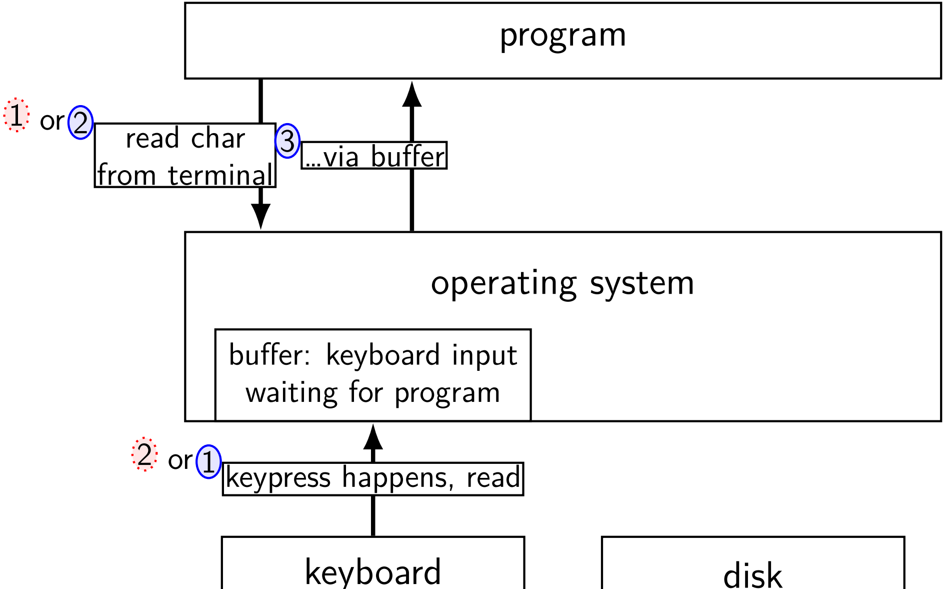
keyboard

disk

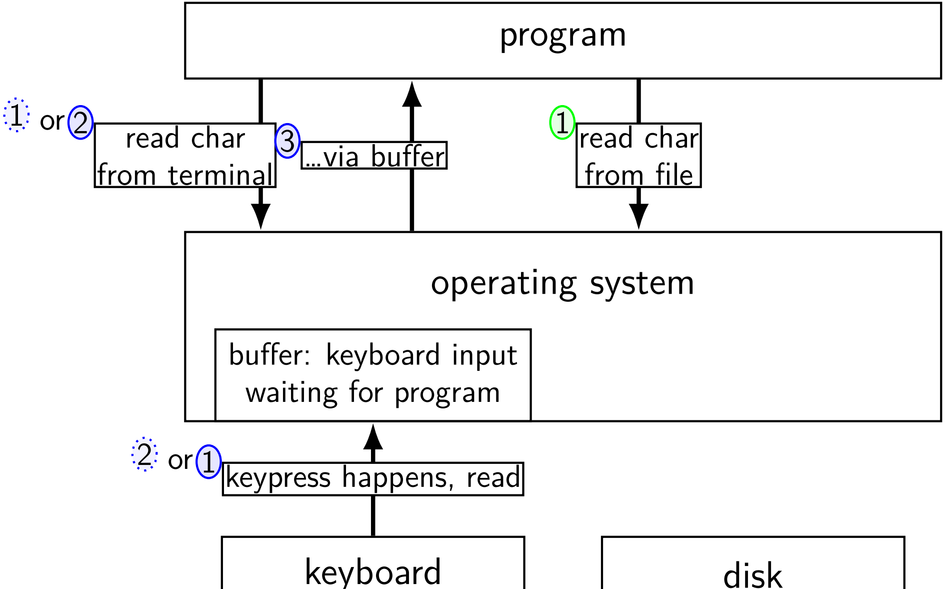
kernel buffering (reads)



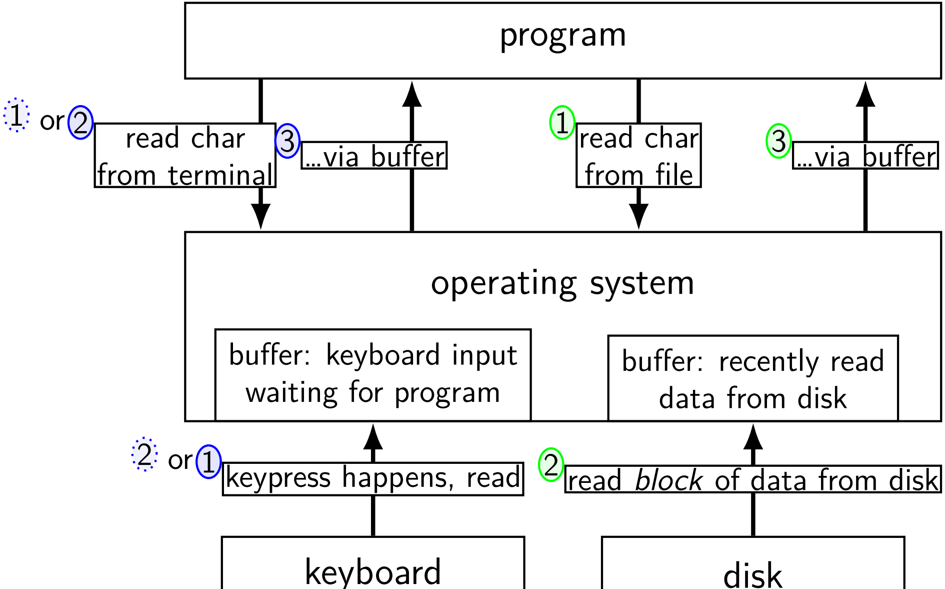
kernel buffering (reads)



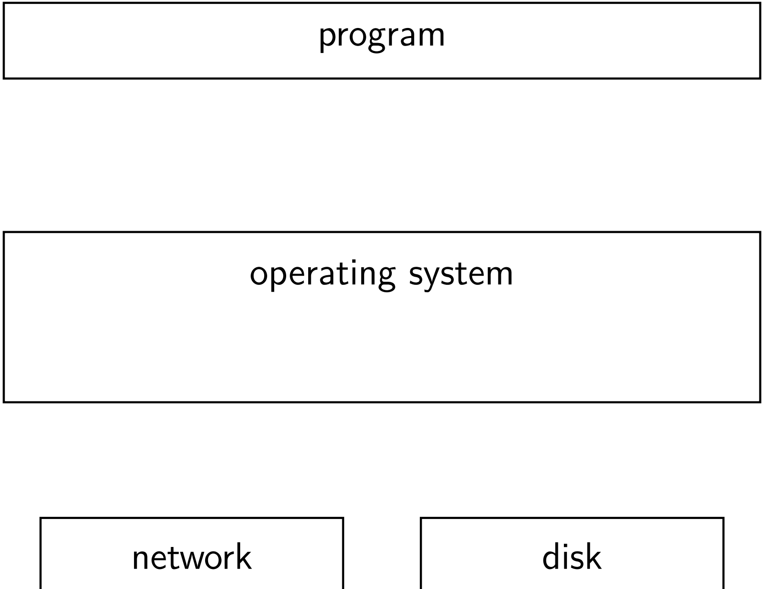
kernel buffering (reads)



kernel buffering (reads)



kernel buffering (writes)



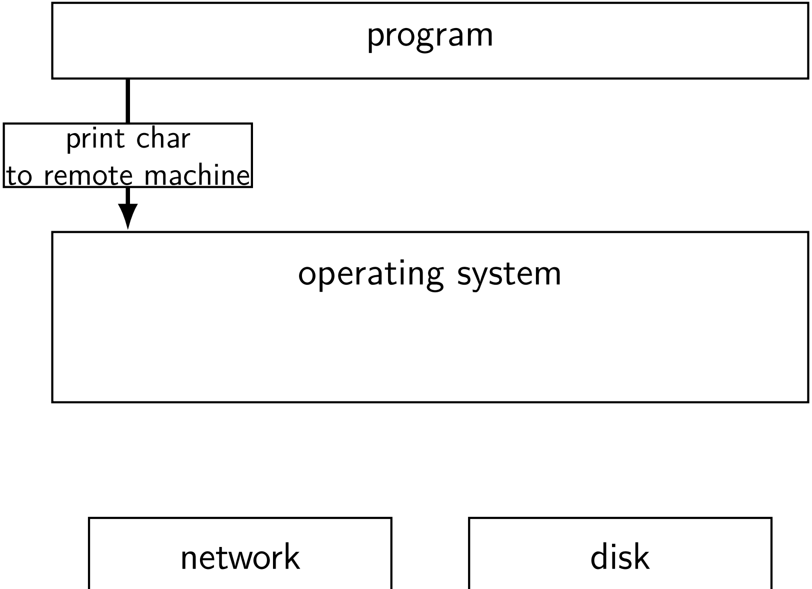
program

operating system

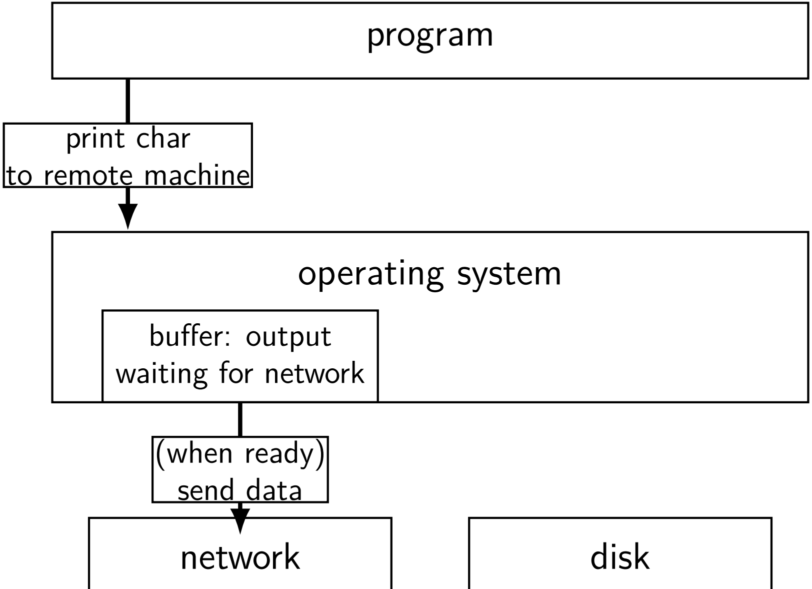
network

disk

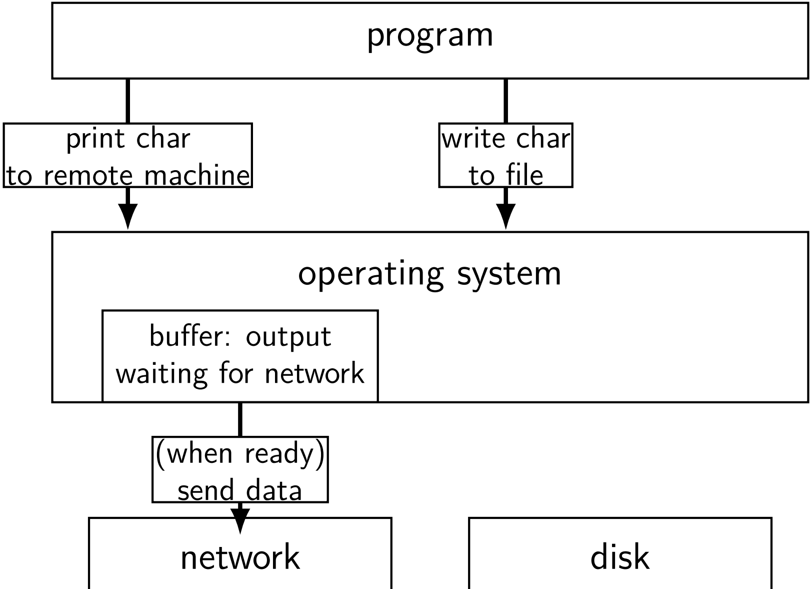
kernel buffering (writes)



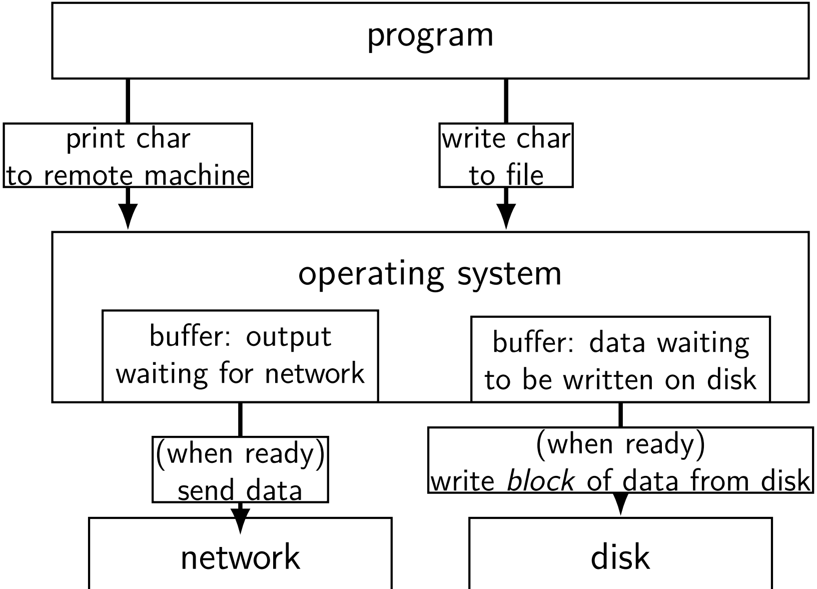
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



program

print char to remote machine

write char to file

operating system

buffer: output waiting for network

buffer: data waiting to be written on disk

(when ready) send data

(when ready) write *block* of data from disk

network

disk

read/write operations

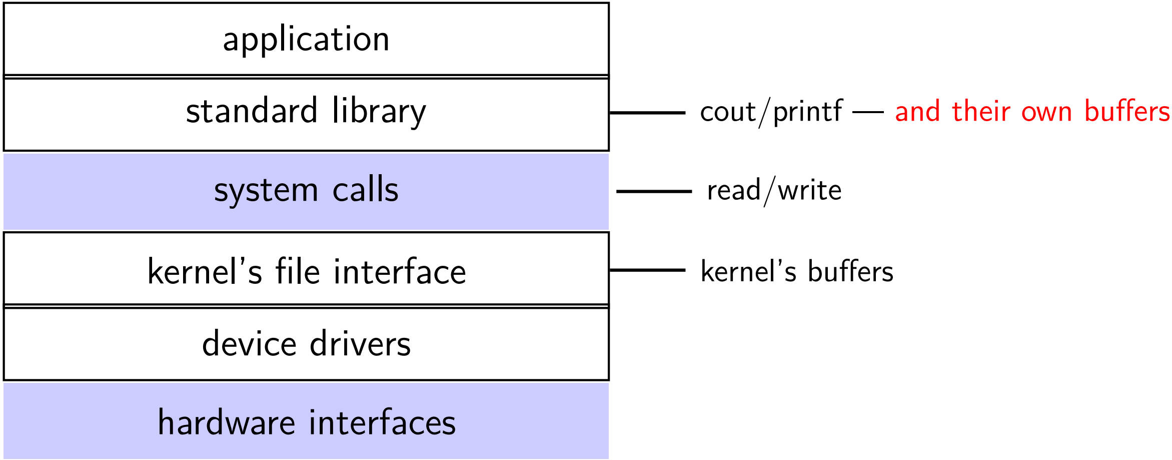
read()/write(): move data into/out of buffer

possibly wait if buffer is empty (read)/full (write)

actual I/O operations — wait for device to be ready

- trigger process to stop waiting if needed

layering



application

standard library — cout/printf — and their own buffers

system calls — read/write

kernel's file interface — kernel's buffers

device drivers

hardware interfaces

why the extra layer

better (but more complex to implement) interface:

- read line
- formatted input (scanf, cin into integer, etc.)
- formatted output

less system calls (bigger reads/writes) sometimes faster

- buffering can combine multiple in/out library calls into one system call

more portable interface

- cin, printf, etc. defined by C and C++ standards

exercise

```c
pid_t p = fork();
int pipe_fds[2];
pipe(pipe_fds);
if (p == 0) { /* child */
  close(pipe_fds[0]);
  char c = 'A';
  write(pipe_fds[1], &c, 1);
  exit(0);
} else { /* parent */
  close(pipe_fds[1]);
  char c;
  int count = read(pipe_fds[0], &c, 1);
  printf("read %d bytes\n", count);
}
```

The child is trying to send the character A to the parent, but the above code outputs `read 0 bytes` instead of `read 1 bytes`. What happened?

exercise solution

pipe() is after fork — two pipes, one in child, one in parent

pipe example (1)

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of file descriptors */
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

pipe example (1)

‘standard’ pattern with fork()

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of file descriptors */
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

pipe example (1)

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of file
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

read() will not indicate end-of-file if write fd is open (any copy of it)

pipe example (1)

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error(); /* e.g. out of fi
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
child_pid = fork();
if (child_pid  == 0) {
    /* in child process, write to pipe */
    close(read_fd);
    write_to_pipe(write_fd); /* function not shown */
    exit(EXIT_SUCCESS);
} else if (child_pid > 0) {
    /* in parent process, read from pipe */
    close(write_fd);
    read_from_pipe(read_fd); /* function not shown */
    waitpid(child_pid, NULL, 0);
    close(read_fd);
} else { /* fork error */ }
```

have habit of closing
to avoid ‘leaking’ file descriptors
you can run out

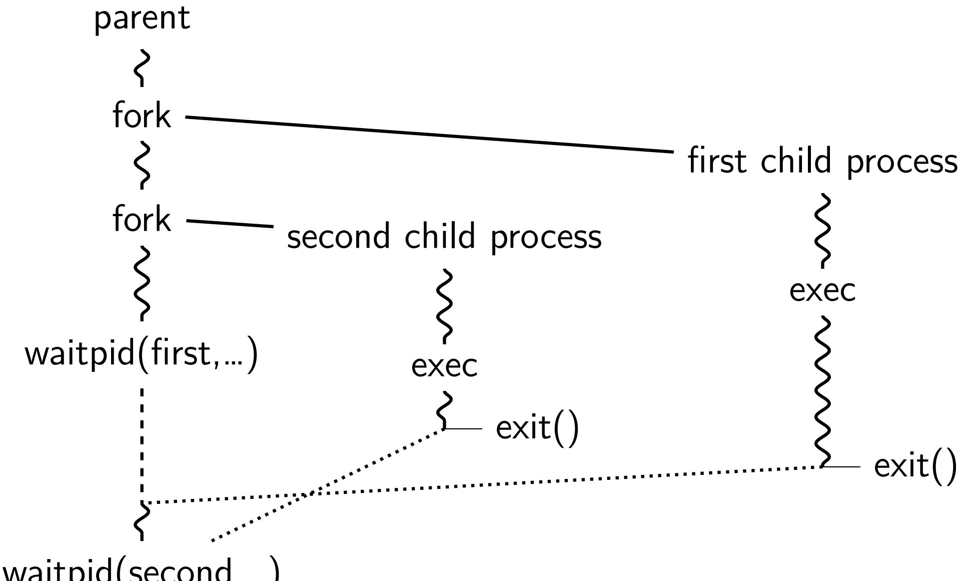
pipe() and blocking

BROKEN example:

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error();
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
write(write_fd, some_buffer, some_big_size);
read(read_fd, some_buffer, some_big_size);
```

This is likely to not terminate. What's the problem?

pattern with multiple?



this class: focus on Unix

Unix-like OSes will be our focus

we have source code

used to from 2150, etc.?

have been around for a while

xv6 imitates Unix

Unix history

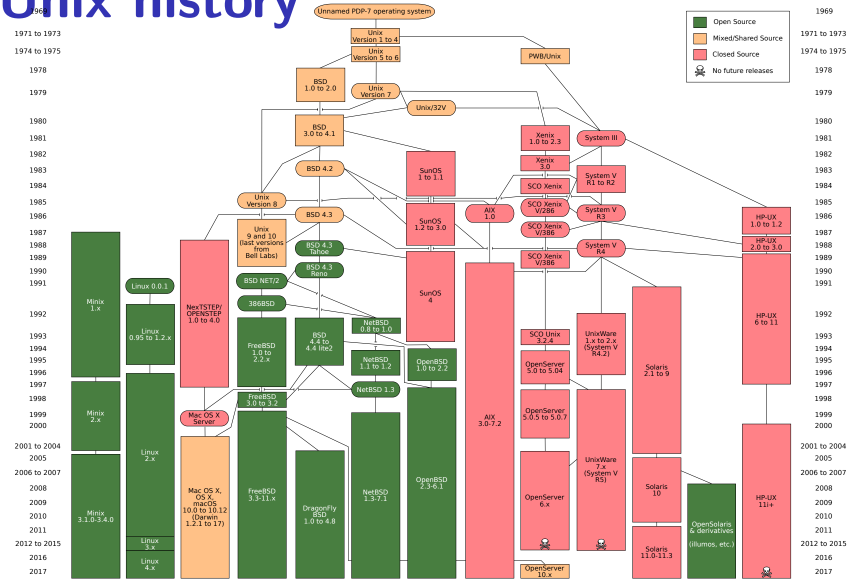


image: Wikpedia/Eraserhead1+Infinity0+Sav_vas

POSIX: standardized Unix

Portable Operating System Interface (POSIX)

“standard for Unix”

current version online:
https://pubs.opengroup.org/onlinepubs/9699919799/

(almost) followed by most current Unix-like OSes

...but OSes add extra features

...and POSIX doesn’t specify everything

what POSIX defines

POSIX specifies the library and shell interface

- source code compatibility

doesn't care what is/is not a system call...

doesn't specify binary formats...

idea: write applications for POSIX, recompile and run on all implementations

- this was a very important goal in the 80s/90s
- at the time, no dominant Unix-like OS (Linux was very immature)

getpid

```
pid_t my_pid = getpid();
printf("my pid is %ld\n", (long) my_pid);
```

process ids in ps

```
cr4bd@machine:~$ ps
  PID TTY          TIME CMD
14777 pts/3    00:00:00 bash
14798 pts/3    00:00:00 ps
```

read/write

```
ssize_t read(int fd, void *buffer, size_t count);
ssize_t write(int fd, void *buffer, size_t count);
```

read/write up to *count* bytes to/from *buffer*

returns number of bytes read/written or -1 on error

- ssize_t is a signed integer type
- error code in `errno`

read returning 0 means end-of-file (*not an error*)

- can read/write less than requested (end of file, broken I/O device, …)

read’ing one byte at a time

```
string s;
ssize_t amount_read;
char c;
/* cast to void * not needed in C */
while ((amount_read = read(STDIN_FILENO, (void*) &c, 1)) > 0) {
    /* amount_read must be exactly 1 */
    s += c;
}
if (amount_read == -1) {
    /* some error happened */
    perror("read"); /* print out a message about it */
} else if (amount_read == 0) {
    /* reached end of file */
}
```

write example

```
/* cast to void * optional in C */
write(STDOUT_FILENO, (void *) "Hello, World!\n", 14);
```

aside: environment variables (1)

key=value pairs associated with every process:

```
$ printenv
MODULE_VERSION_STACK=3.2.10
MANPATH=:/opt/puppetlabs/puppet/share/man
XDG_SESSION_ID=754
HOSTNAME=labsrv01
SELINUX_ROLE_REQUESTED=
TERM=screen
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=128.143.67.91 58432 22
SELINUX_USE_CURRENT_RANGE=
QTDIR=/usr/lib64/qt-3.3
OLDPWD=/zf14/cr4bd
QTINC=/usr/lib64/qt-3.3/include
SSH_TTY=/dev/pts/0
QT_GRAPHICSSYSTEM_CHECKED=1
USER=cr4bd
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or
MODULE_VERSION=3.2.10
MAIL=/var/spool/mail/cr4bd
PATH=/zf14/cr4bd/.cargo/bin:/zf14/cr4bd/bin:/usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/bin:/u
PWD=/zf14/cr4bd
LANG=en_US.UTF-8
```

aside: environment variables (2)

environment variable library functions:

- `getenv("KEY")` → *value*
- `putenv("KEY=value")` (sets KEY to *value*)
- `setenv("KEY", "value")` (sets KEY to *value*)

```
int execve(char *path, char **argv, char **envp)
```

```
    char *envp[] = { "KEY1=value1", "KEY2=value2", NULL };
    char *argv[] = { "somecommand", "some arg", NULL };
    execve("/path/to/somecommand", argv, envp);
```

normal exec versions — keep same environment variables

aside: environment variables (3)

interpretation up to programs, but common ones…

`PATH=/bin:/usr/bin`

> to run a program 'foo', look for an executable in `/bin/foo`, then `/usr/bin/foo`

`HOME=/zf14/cr4bd`

> current user's home directory is '/zf14/cr4bd'

`TERM=screen-256color`

> your output goes to a 'screen-256color'-style terminal

…

multiple processes?

```
while (...) {
    pid = fork();
    if (pid == 0) {
        exec ...
    } else if (pid > 0) {
        pids.push_back(pid);
    }
}

/* retrieve exit statuses in order */
for (pid_t pid : pids) {
    waitpid(pid, ...);
    ...
}
```

waiting for all children

```
#include <sys/wait.h>
...
  while (true) {
    pid_t child_pid = waitpid(-1, &status, 0);
    if (child_pid == (pid_t) -1) {
      if (errno == ECHILD) {
        /* no child process to wait for */
        break;
      } else {
        /* some other error */
      }
    }
    /* handle child_pid exiting */
  }
```

multiple processes?

```
while (...) {
    pid = fork();
    if (pid == 0) {
        exec ...
    } else if (pid > 0) {
        pids.push_back(pid);
    }
}

/* retrieve exit statuses as processes finish */
while ((pid = waitpid(−1, ...)) != −1) {
    handleProcessFinishing(pid);
}
```

‘waiting’ without waiting

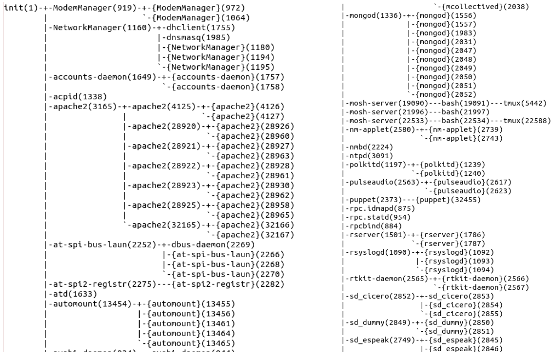
```
#include <sys/wait.h>
...
  pid_t return_value = waitpid(child_pid, &status, WNOHANG);
  if (return_value == (pid_t) 0) {
    /* child process not done yet */
  } else if (child_pid == (pid_t) –1) {
    /* error */
  } else {
    /* handle child_pid exiting */
  }
```

parent and child processes

every process (but process id 1) has a *parent process* (`getppid()`)

this is the process that can wait for it

creates tree of processes (Linux `pstree` command):



```
init(1)-+-ModemManager(919)-+-{ModemManager}(972)
        |                   `-{ModemManager}(1064)
        |-NetworkManager(1160)-+-dhclient(1755)
        |                      |-dnsmasq(1985)
        |                      |-{NetworkManager}(1180)
        |                      |-{NetworkManager}(1194)
        |                      `-{NetworkManager}(1195)
        |-accounts-daemon(1649)-+-{accounts-daemon}(1757)
        |                       `-{accounts-daemon}(1758)
        |-acpid(1338)
        |-apache2(3165)-+-apache2(4125)-+-{apache2}(4126)
        |               |               `-{apache2}(4127)
        |               |-apache2(28920)-+-{apache2}(28926)
        |               |                `-{apache2}(28960)
        |               |-apache2(28921)-+-{apache2}(28927)
        |               |                `-{apache2}(28963)
        |               |-apache2(28922)-+-{apache2}(28928)
        |               |                `-{apache2}(28961)
        |               |-apache2(28923)-+-{apache2}(28930)
        |               |                `-{apache2}(28962)
        |               |-apache2(28925)-+-{apache2}(28958)
        |               |                `-{apache2}(28965)
        |               `-apache2(32165)-+-{apache2}(32166)
        |                                `-{apache2}(32167)
        |-at-spi-bus-laun(2252)-+-dbus-daemon(2269)
        |                       |-{at-spi-bus-laun}(2266)
        |                       |-{at-spi-bus-laun}(2268)
        |                       `-{at-spi-bus-laun}(2270)
        |-at-spi2-registr(2275)---{at-spi2-registr}(2282)
        |-atd(1633)
        |-automount(13454)-+-{automount}(13455)
        |                  |-{automount}(13456)
        |                  |-{automount}(13461)
        |                  |-{automount}(13464)
        |                  `-{automount}(13465)
        |-avahi-daemon(934)---avahi-daemon(941)
```

```
|                    `-{mcollectived}(2038)
|-mongod(1336)-+-{mongod}(1556)
|              |-{mongod}(1557)
|              |-{mongod}(1983)
|              |-{mongod}(2031)
|              |-{mongod}(2047)
|              |-{mongod}(2048)
|              |-{mongod}(2049)
|              |-{mongod}(2050)
|              |-{mongod}(2051)
|              `-{mongod}(2052)
|-mosh-server(19090)---bash(19091)---tmux(5442)
|-mosh-server(21996)---bash(21997)
|-mosh-server(22533)---bash(22534)---tmux(22588)
|-nm-applet(2580)-+-{nm-applet}(2739)
|                 `-{nm-applet}(2743)
|-nmbd(2224)
|-ntpd(3091)
|-polkitd(1197)-+-{polkitd}(1239)
|               `-{polkitd}(1240)
|-pulseaudio(2563)-+-{pulseaudio}(2617)
|                  `-{pulseaudio}(2623)
|-puppet(2373)---{puppet}(32455)
|-rpc.idmapd(875)
|-rpc.statd(954)
|-rpcbind(884)
|-rserver(1501)-+-{rserver}(1786)
|               `-{rserver}(1787)
|-rsyslogd(1090)-+-{rsyslogd}(1092)
|                |-{rsyslogd}(1093)
|                `-{rsyslogd}(1094)
|-rtkit-daemon(2565)-+-{rtkit-daemon}(2566)
|                    `-{rtkit-daemon}(2567)
|-sd_cicero(2852)-+-sd_cicero(2853)
|                 |-{sd_cicero}(2854)
|                 `-{sd_cicero}(2855)
|-sd_dummy(2849)-+-{sd_dummy}(2850)
|                `-{sd_dummy}(2851)
|-sd_espeak(2749)-+-{sd_espeak}(2845)
|                 |-{sd_espeak}(2846)
```

]

parent and child questions...

what if parent process exits before child?

- child's parent process becomes process id 1 (typically called *init*)

what if parent process never `waitpid()`s (or equivalent) for child?

- child process stays around as a "zombie"
- can't reuse pid in case parent wants to use `waitpid()`

what if non-parent tries to `waitpid()` for child?

- waitpid fails

exercise

```
int fd = open("output.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666);
write(fd, "A", 1);
dup2(STDOUT_FILENO, 100);
dup2(fd, STDOUT_FILENO);
write(STDOUT_FILENO, "B", 1);
write(fd, "C", 1);
close(fd);
write(STDOUT_FILENO, "D", 1);
write(100, "E", 1);
```

Assume fd `100` is not what open returns. What is written to `output.txt`?

A. ABCDE

B. ABCD

C. ABC

D. ACD

E. something else

read’ing a fixed amount

```
ssize_t offset = 0;
const ssize_t amount_to_read = 1024;
char result[amount_to_read];
do {
    /* cast to void * optional in C */
    ssize_t amount_read =
        read(STDIN_FILENO,
             (void *) (result + offset),
             amount_to_read − offset);
    if (amount_read < 0) {
        perror("read"); /* print error message */
        ... /* abort??? */
    } else {
        offset += amount_read;
    }
} while (offset != amount_to_read && amount_read != 0);
```

partial reads

on regular file: read reads what you request

but otherwise: usually gives you what's known to be available

- after waiting for something to be available

partial reads

on regular file: read reads what you request

but otherwise: usually gives you what's known to be available

- after waiting for something to be available

reading from network — what's been received

reading from keyboard — what's been typed

write example (with error checking)

```
const char *ptr = "Hello, World!\n";
ssize_t remaining = 14;
while (remaining > 0) {
    /* cast to void * optional in C */
    ssize_t amount_written = write(STDOUT_FILENO,
                                   ptr,
                                   remaining);
    if (amount_written < 0) {
        perror("write"); /* print error message */
        ... /* abort??? */
    } else {
        remaining -= amount_written;
        ptr += amount_written;
    }
}
```

partial writes

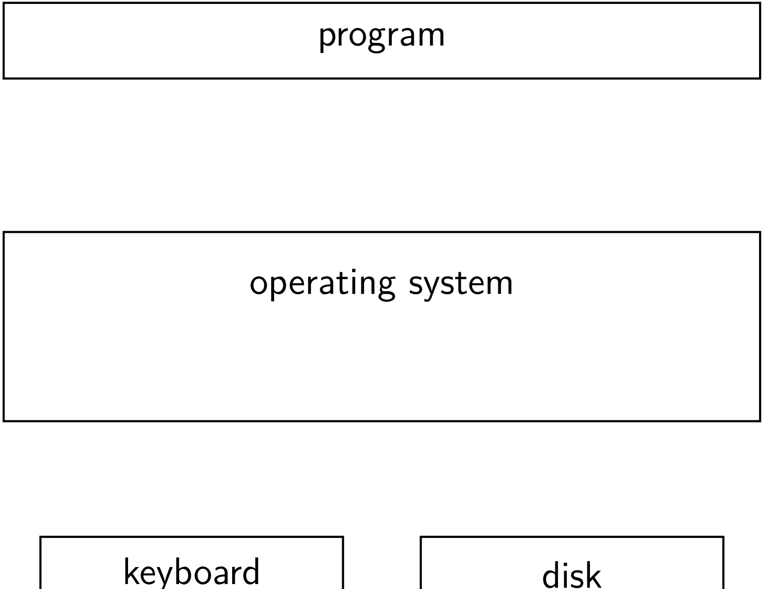
usually only happen on error or interruption

- but can request "non-blocking"
- (interruption: via *signal*)

usually: write waits until it completes

- = until remaining part fits in buffer in kernel
- does not mean data was sent on network, shown to user yet, etc.

kernel buffering (reads)



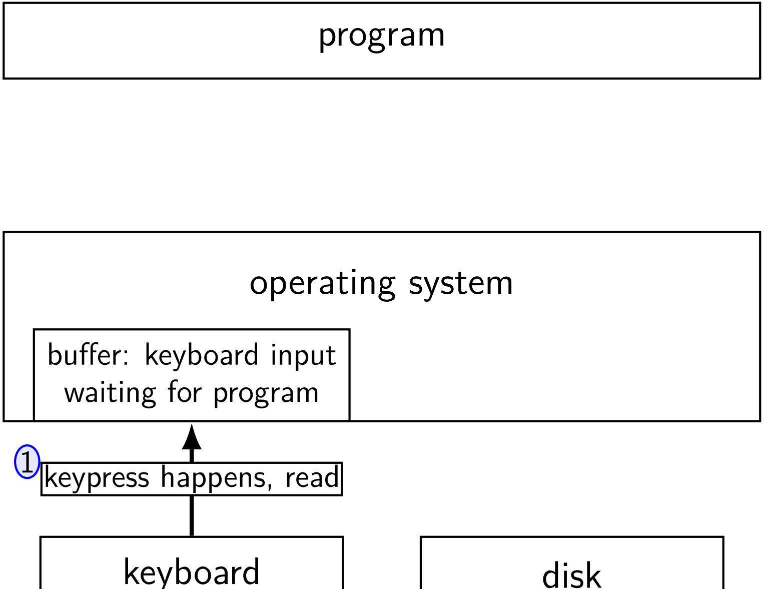
program

operating system

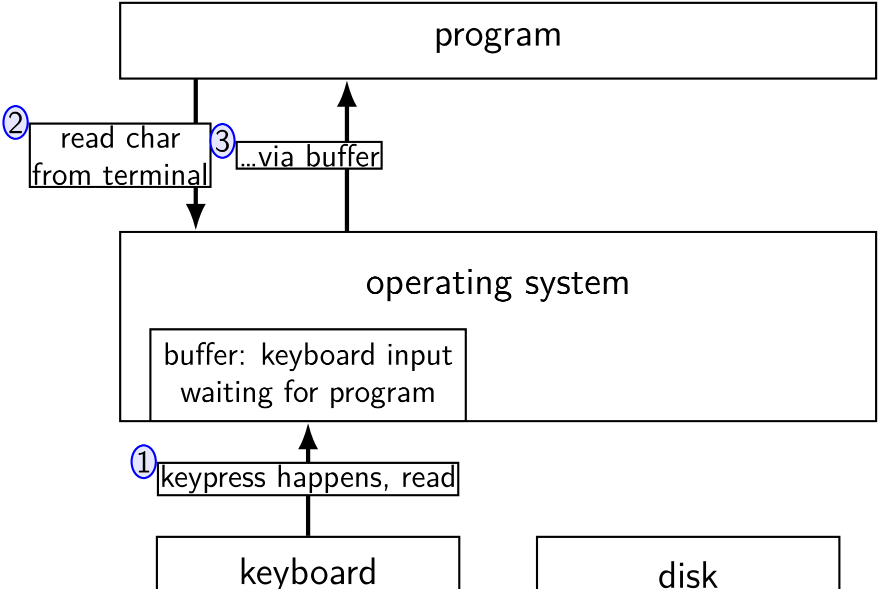
keyboard

disk

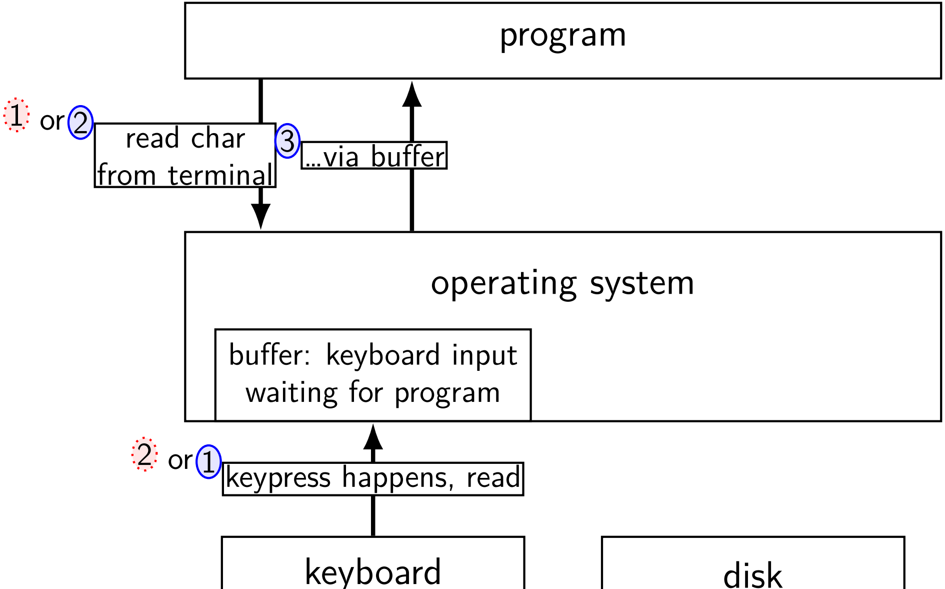
kernel buffering (reads)



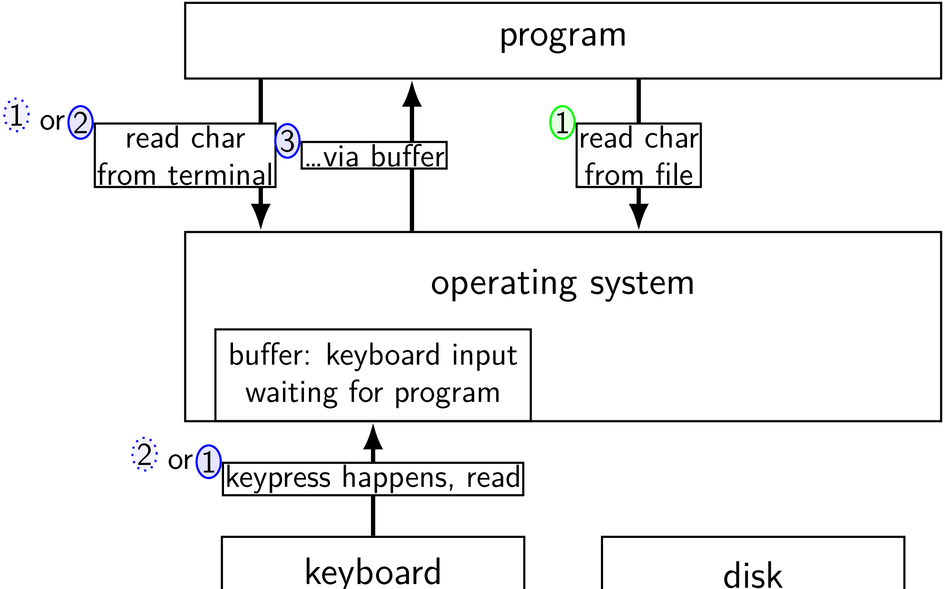
kernel buffering (reads)



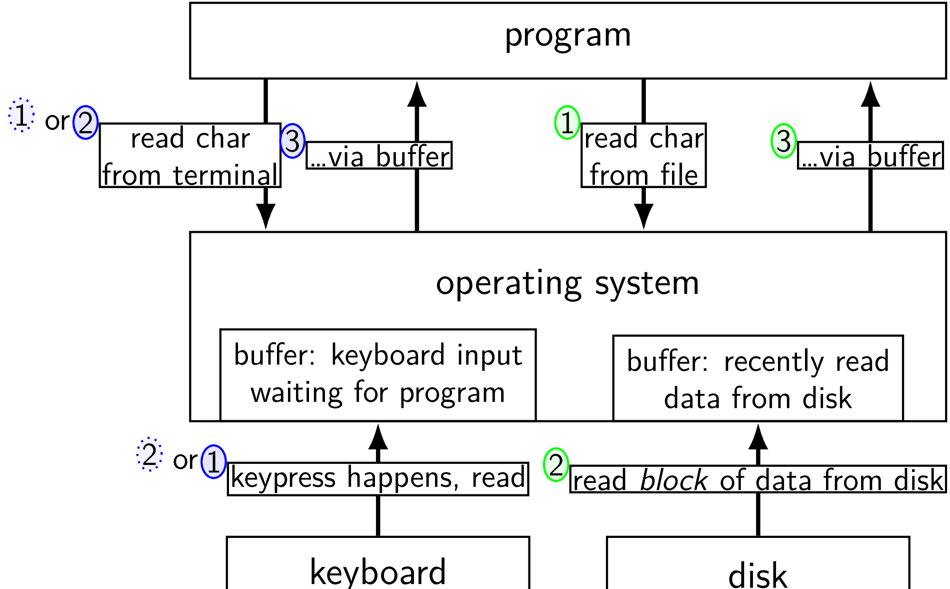
kernel buffering (reads)



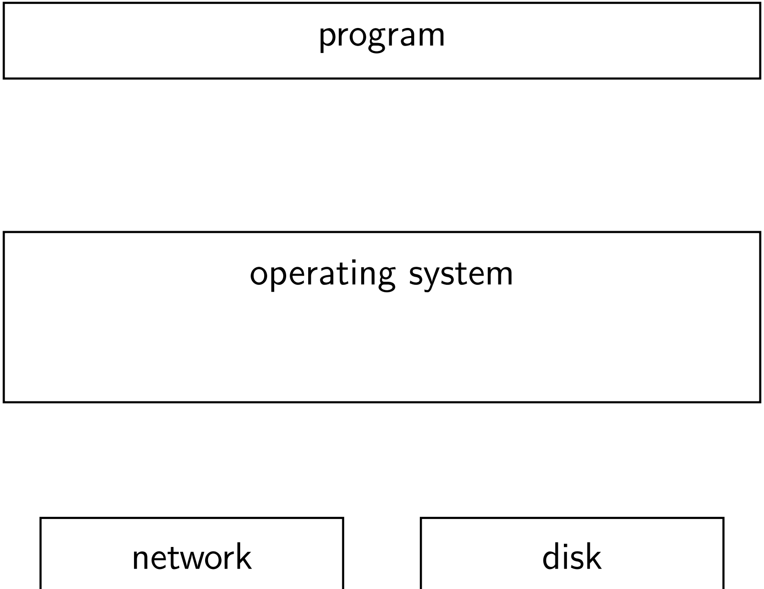
kernel buffering (reads)



kernel buffering (reads)



kernel buffering (writes)



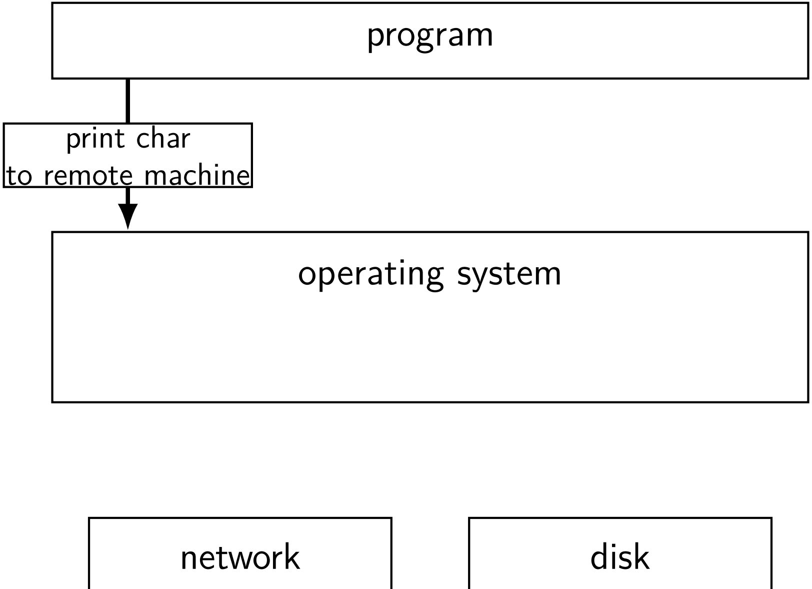
program

operating system

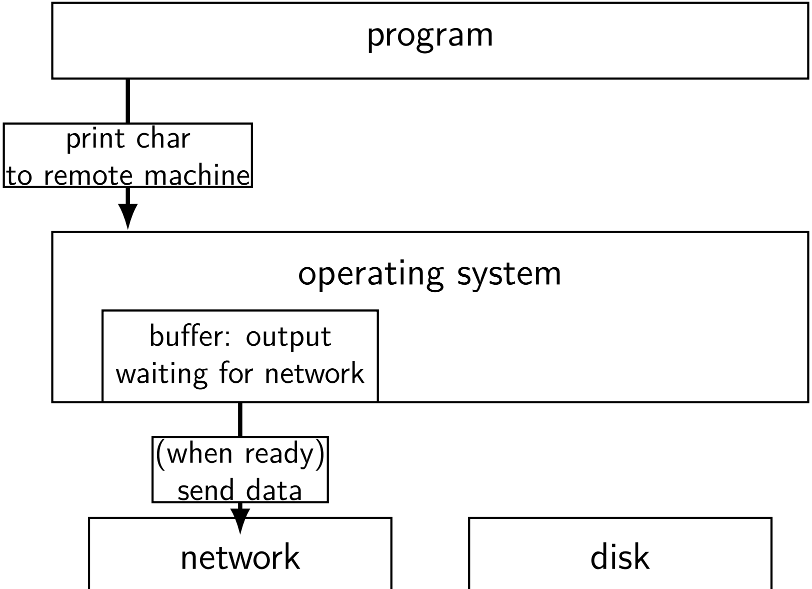
network

disk

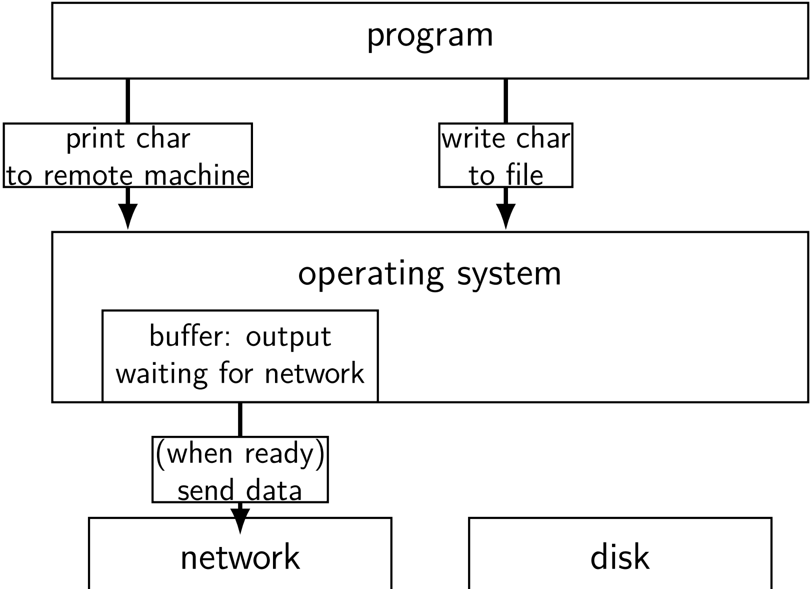
kernel buffering (writes)



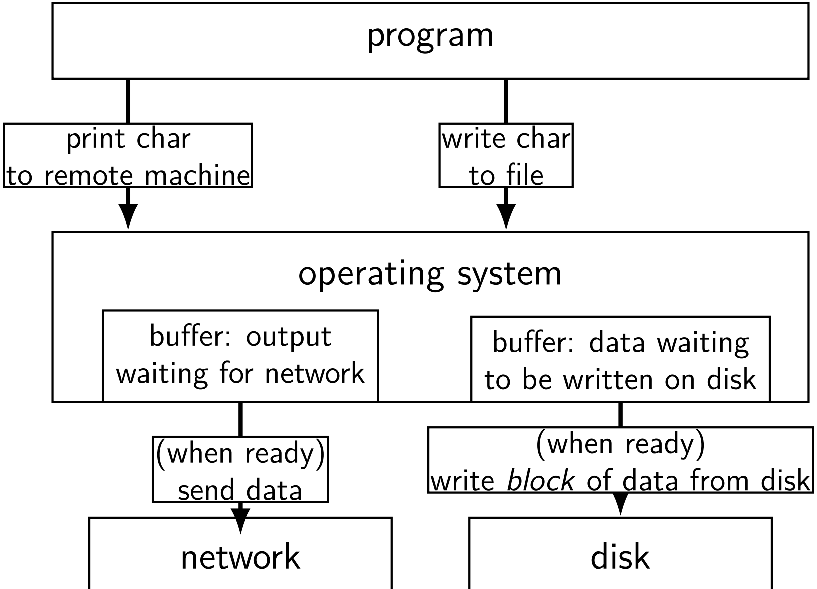
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



program

print char to remote machine

write char to file

operating system

buffer: output waiting for network

buffer: data waiting to be written on disk

(when ready) send data

(when ready) write *block* of data from disk

network

disk

read/write operations

read()/write(): move data into/out of buffer

possibly wait if buffer is empty (read)/full (write)

actual I/O operations — wait for device to be ready

- trigger process to stop waiting if needed

filesystem abstraction

regular files — named collection of bytes

- also: size, modification time, owner, access control info, …

directories — folders containing files and directories

- hierarchical naming: `/net/zf14/cr4bd/fall2018/cs4414`
- *mostly* contains regular files or directories

open

```
int open(const char *path, int flags);
int open(const char *path, int flags, int mode);
...

int read_fd = open("dir/file1", O_RDONLY);
int write_fd = open("/other/file2",
        O_WRONLY | O_CREAT | O_TRUNC, 0666);
int rdwr_fd = open("file3", O_RDWR);
```

open

```
int open(const char *path, int flags);
int open(const char *path, int flags, int mode);
```

path = filename

e.g. `"/foo/bar/file.txt"`
`file.txt` in
directory `bar` in
directory `foo` in
“the root directory”

e.g. `"quux/other.txt`
`other.txt` in
directory `quux` in
“the current working directory” (set with `chdir()`)

open: file descriptors

```
int open(const char *path, int flags);
int open(const char *path, int flags, int mode);
```

return value = file descriptor (or -1 on error)

index into table of *open file descriptions* for each process

used by system calls that deal with open files

POSIX: everything is a file

the file: one interface for

- devices (terminals, printers, ...)
- regular files on disk
- networking (sockets)
- local interprocess communication (pipes, sockets)

basic operations: open(), read(), write(), close()

exercise

```
int pipe_fds[2]; pipe(pipe_fds);
pid_t p = fork();
if (p == 0) {
  close(pipe_fds[0]);
  for (int i = 0; i < 10; ++i) {
    char c = '0' + i;
    write(pipe_fds[1], &c, 1);
  }
  exit(0);
}
close(pipe_fds[1]);
char buffer[10];
ssize_t count = read(pipe_fds[0], buffer, 10);
for (int i = 0; i < count; ++i) {
  printf("%c", buffer[i]);
}
```

Which of these are possible outputs (if pipe, read, write, fork don't fail)?

A. `0123456789` B. `0` C. (nothing)

D. A and B E. A and C F. A, B, and C

exercise

```
int pipe_fds[2]; pipe(pipe_fds);
pid_t p = fork();
if (p == 0) {
  close(pipe_fds[0]);
  for (int i = 0; i < 10; ++i) {
    char c = '0' + i;
    write(pipe_fds[1], &c, 1);
  }
  exit(0);
}
close(pipe_fds[1]);
char buffer[10];
ssize_t count = read(pipe_fds[0], buffer, 10);
for (int i = 0; i < count; ++i) {
  printf("%c", buffer[i]);
}
```

Which of these are possible outputs (if pipe, read, write, fork don't fail)?

A. 0123456789 B. 0 C. (nothing)

D. A and B E. A and C F. A, B, and C

empirical evidence

```
  8 0
374 01
210 012
 30 0123
 12 01234
  3 012345
  1 0123456
  2 01234567
  1 012345678
359 0123456789
```

partial reads

read returning 0 always means end-of-file

- by default, read always waits *if no input available yet*
- but can set read to return *error* instead of waiting

read can return less than requested if not available

- e.g. child hasn't gotten far enough

pipe: closing?

if all write ends of pipe are closed

- can get end-of-file (read() returning 0) on read end
- exit()ing closes them

→ close write end when not using

generally: limited number of file descriptors per process

→ good habit to close file descriptors not being used

(but probably didn't matter for read end of pipes in example)

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

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 writes 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 writes 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 writes and writes: hundreds of microseconds

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

HDD/SDDs are slow

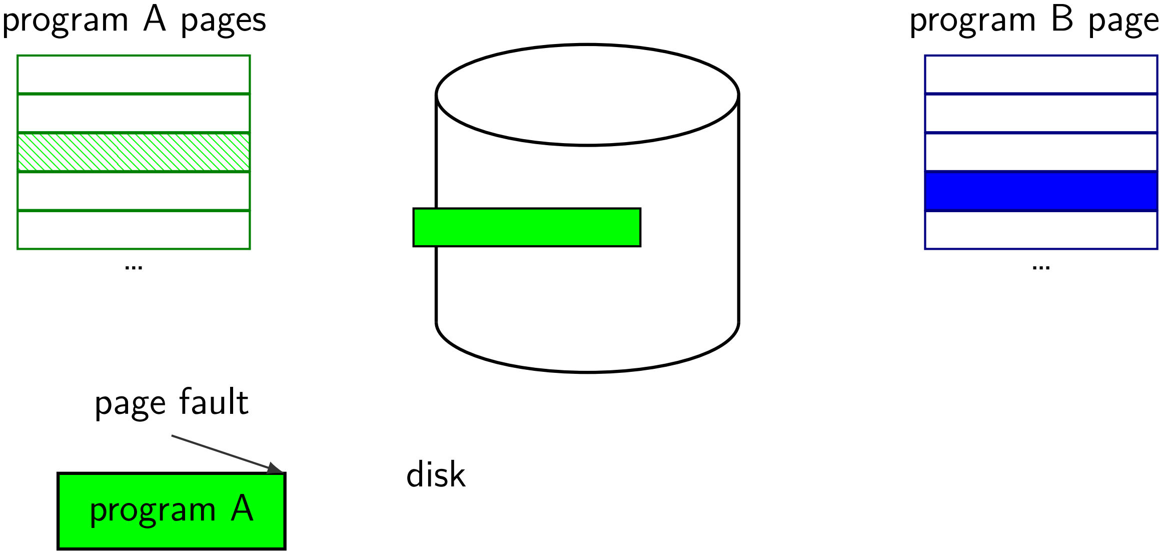
HDD reads and writes: milliseconds to tens of milliseconds

- minimum size: 512 bytes
- writing tens of kilobytes basically as fast as writing 512 bytes

SSD writes and writes: hundreds of microseconds

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

swapping timeline



program A pages

...

program B page

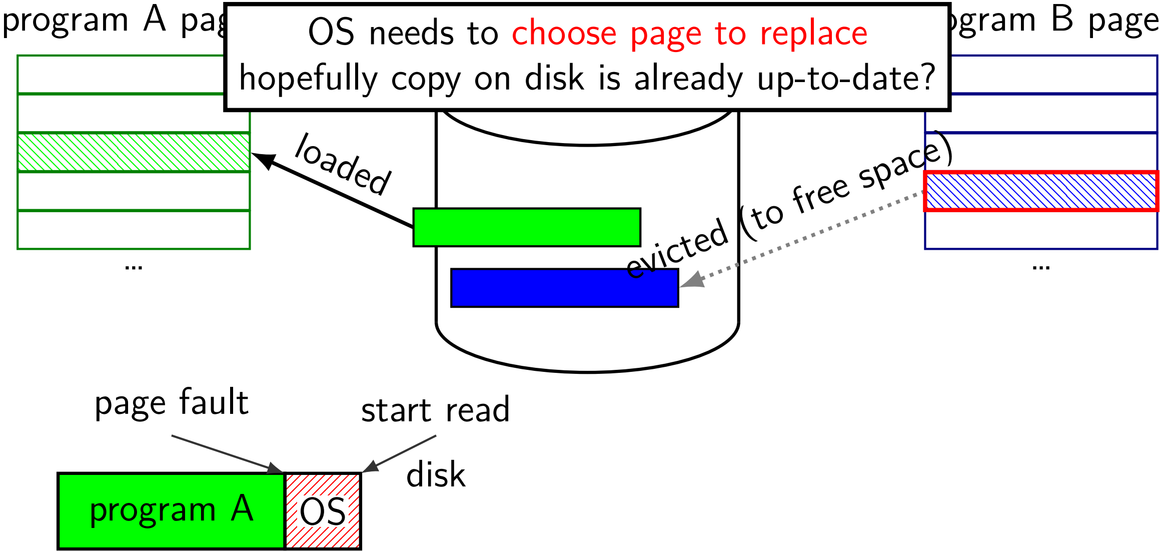
...

page fault

program A

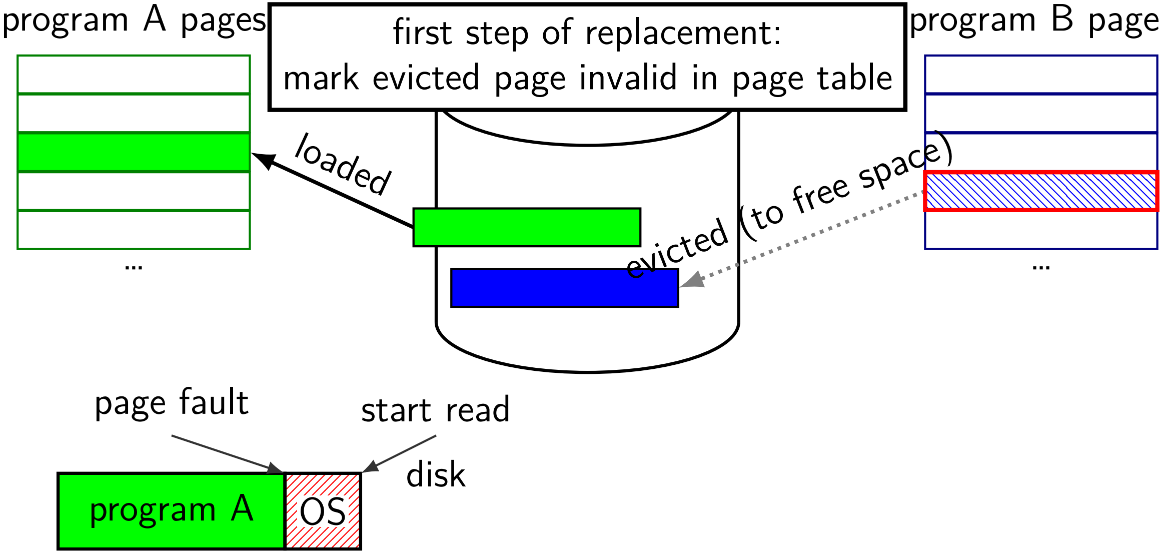
disk

swapping timeline



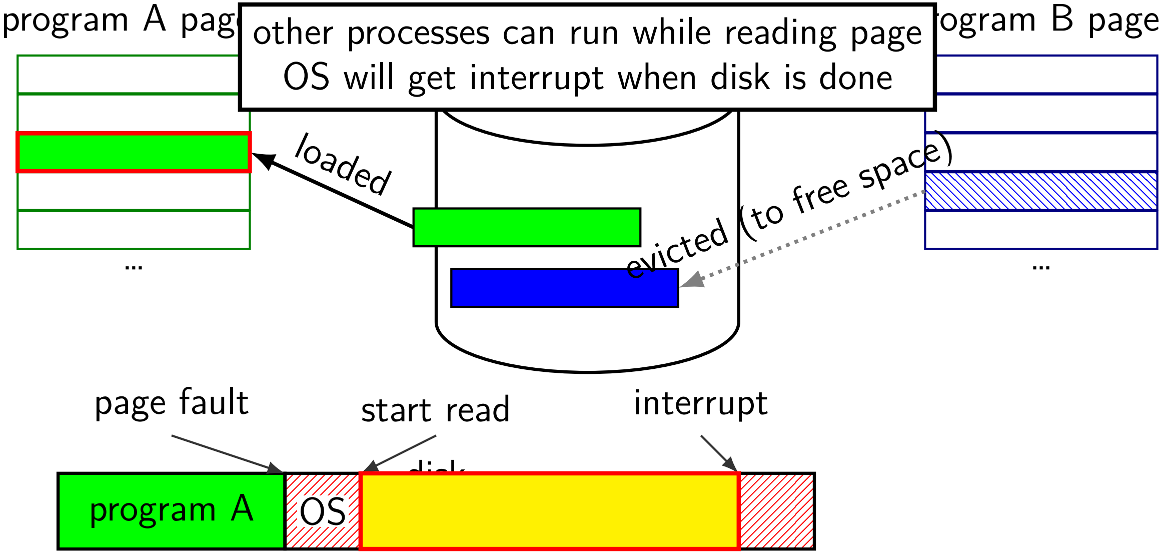
OS needs to choose page to replace
hopefully copy on disk is already up-to-date?

swapping timeline



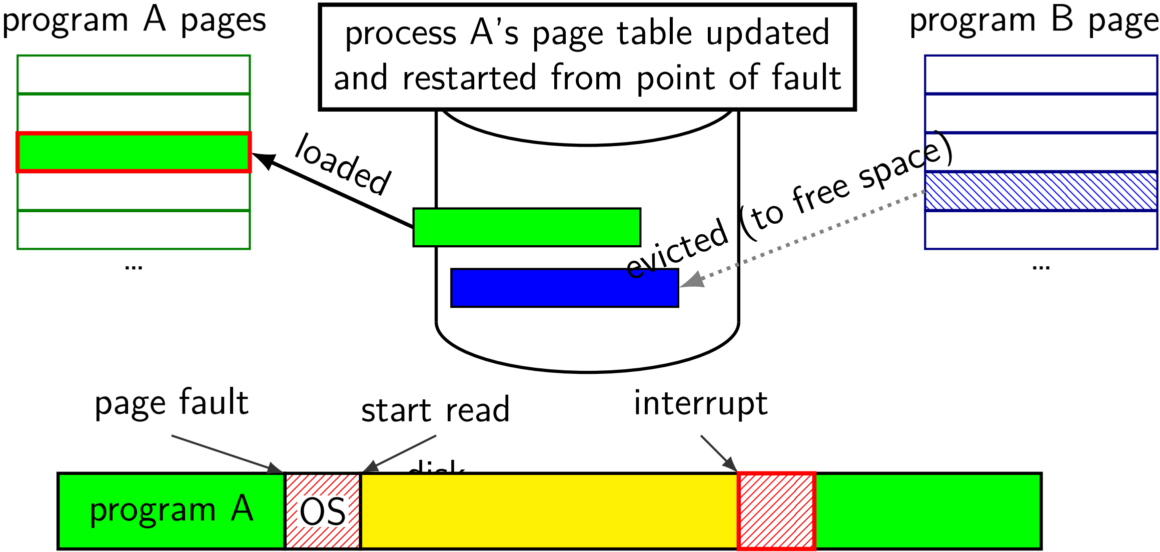
first step of replacement:
mark evicted page invalid in page table

swapping timeline



other processes can run while reading page
OS will get interrupt when disk is done

swapping timeline



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

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]
ffffffffff600000-ffffffffff601000 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
0060b000-0
01974000-0
7f60c718b0                                                                cale-archive
7f60c74900                                                                gnu/libc-2.1
7f60c764e0                                                                gnu/libc-2.1
7f60c784e0                                                                gnu/libc-2.1
7f60c78520                                                                gnu/libc-2.1
7f60c78540
7f60c78590                                                                gnu/ld-2.19.s
7f60c7a390
7f60c7a7a0
7f60c7a7b0                                                                gnu/ld-2.19.s
7f60c7a7c0                                                                gnu/ld-2.19.s
7f60c7a7d0
7ffc5d2b20
7ffc5d3b00
7ffc5d3b30                                               [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

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 ...

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
```