making any cache look bad

1. access enough blocks, to fill the cache
2. access an additional block, replacing something
3. access last block replaced
4. access last block replaced
5. access last block replaced

...

but — typical real programs have locality

cache optimizations

(assuming typical locality + keeping cache size constant if possible...)

	miss rate	hit time	miss penalty
increase cache size	better	worse	—
increase associativity	better	worse	worse?
increase block size	depends	worse	worse
add secondary cache	—	—	better
write-allocate	better	—	?
writeback	—	—	?
LRU replacement	better	?	worse?
prefetching	better	—	—

prefetching = guess what program will use, access in advance

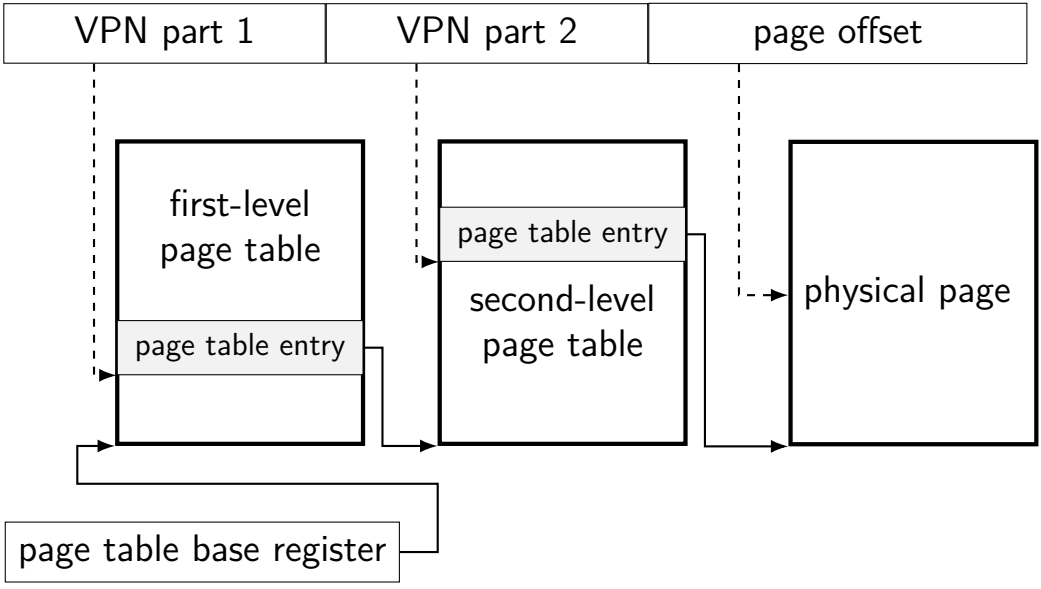
$$\text{average time} = \text{hit time} + \text{miss rate} \times \text{miss penalty}$$

cache optimizations by miss type

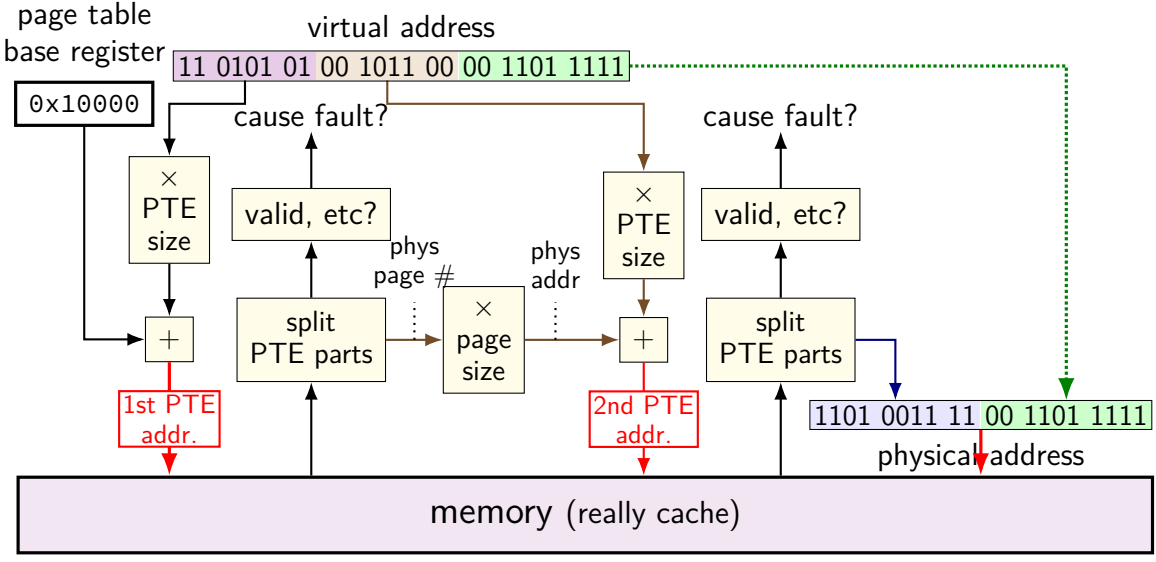
(assuming other listed parameters remain constant)

	capacity	conflict	compulsory
increase cache size	fewer misses	fewer misses	—
increase associativity	—	fewer misses	—
increase block size	more misses?	more misses?	fewer misses
LRU replacement	—	fewer misses	—
prefetching	—	—	fewer misses

another view



two-level page table lookup



cache accesses and multi-level PTs

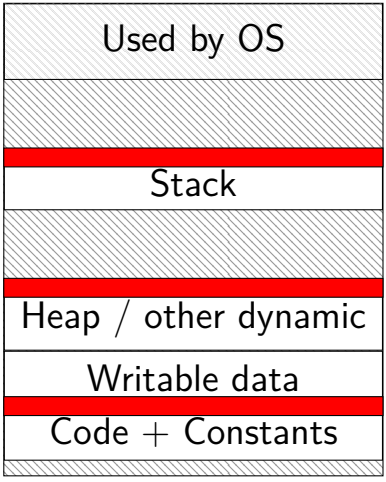
four-level page tables — five cache accesses per program memory access

L1 cache hits — typically a couple cycles each?

so add 8 cycles to each program memory access?

not acceptable

program memory active sets



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

small areas of memory active at a time

one or two pages in each area?

0x0000 0000 0040 0000

page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

each page contains whole functions, arrays, stack frames, etc.

page table entries and locality

page table entries have excellent temporal locality

typically one or two pages of the stack active

typically one or two pages of code active

typically one or two pages of heap/globals active

each page contains whole functions, arrays, stack frames, etc.

needed page table entries are very small

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

only caches the page table lookup itself
(generally) just entries from the last-level page tables

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

not much spatial locality between page table entries
(they're used for kilobytes of data already)
(and if spatial locality, maybe use larger page size?)

page table entry cache

caled a **TLB** (translation lookaside buffer)

very small cache of page table entries

L1 cache	TLB
physical addresses	virtual page numbers
bytes from memory	page table entries
tens of bytes per block	one page table entry per block
usually thousands of blocks	usually tens of entries

few active page table entries at a time
enables highly associative cache designs

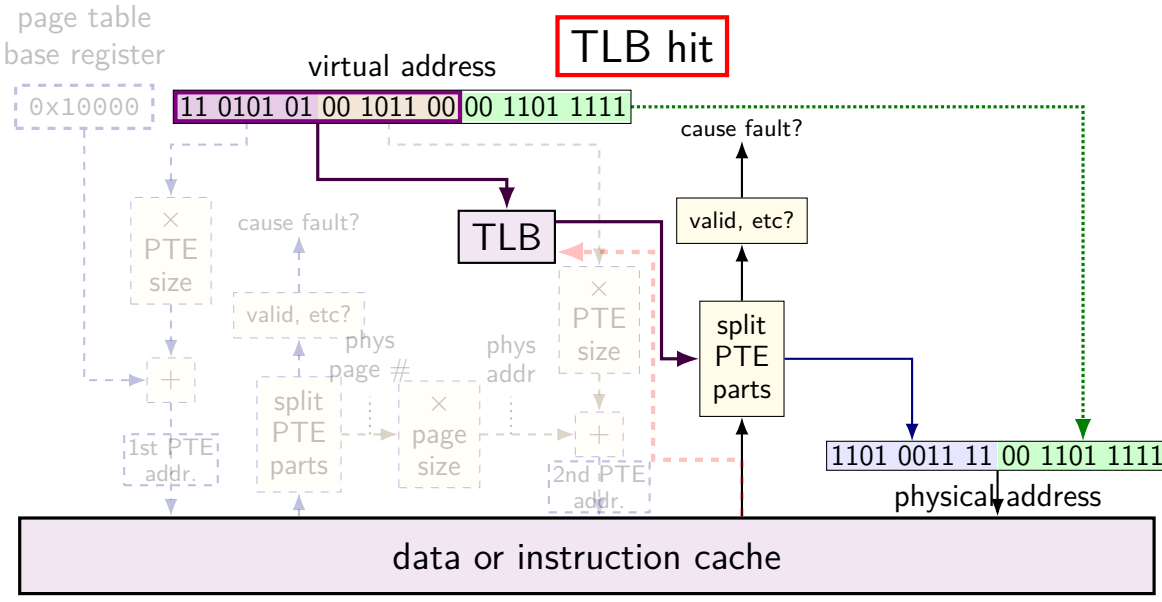
TLB and multi-level page tables

TLB caches valid last-level page table entries

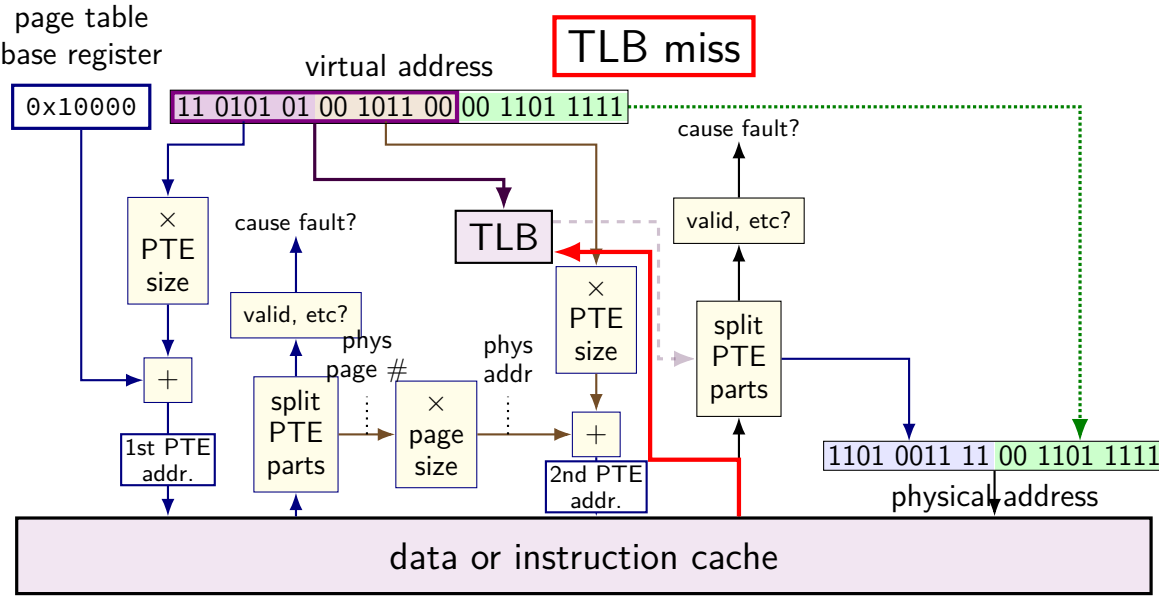
doesn't matter which last-level page table

means TLB output can be used directly to form address

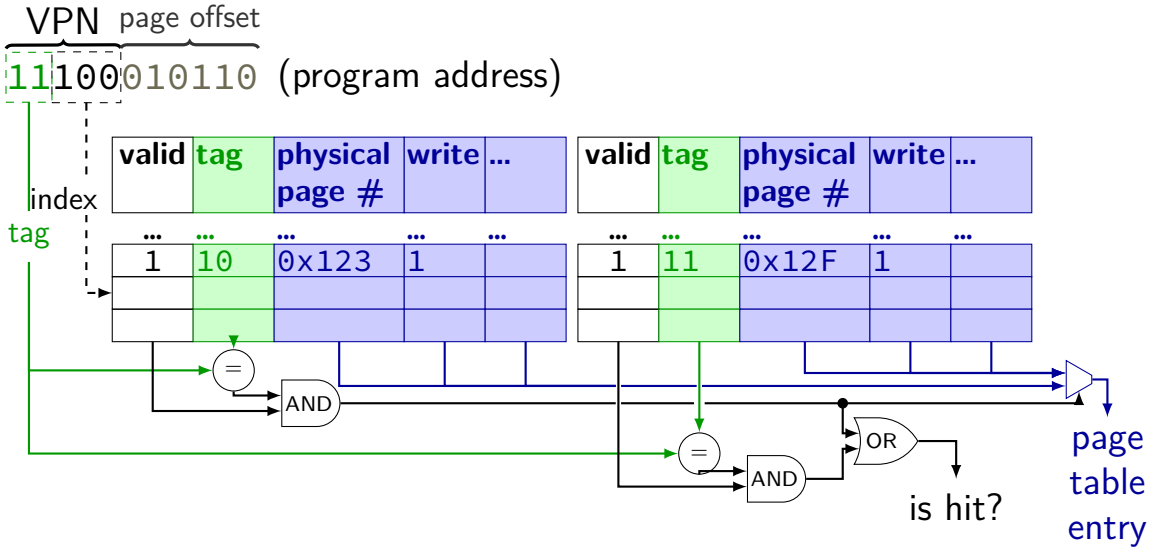
TLB and two-level lookup



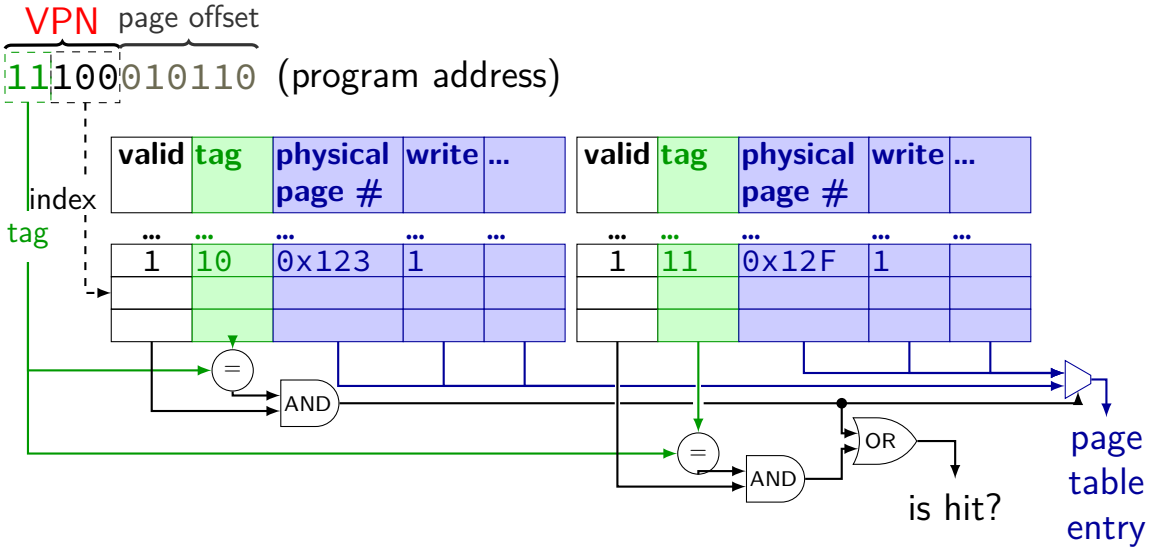
TLB and two-level lookup



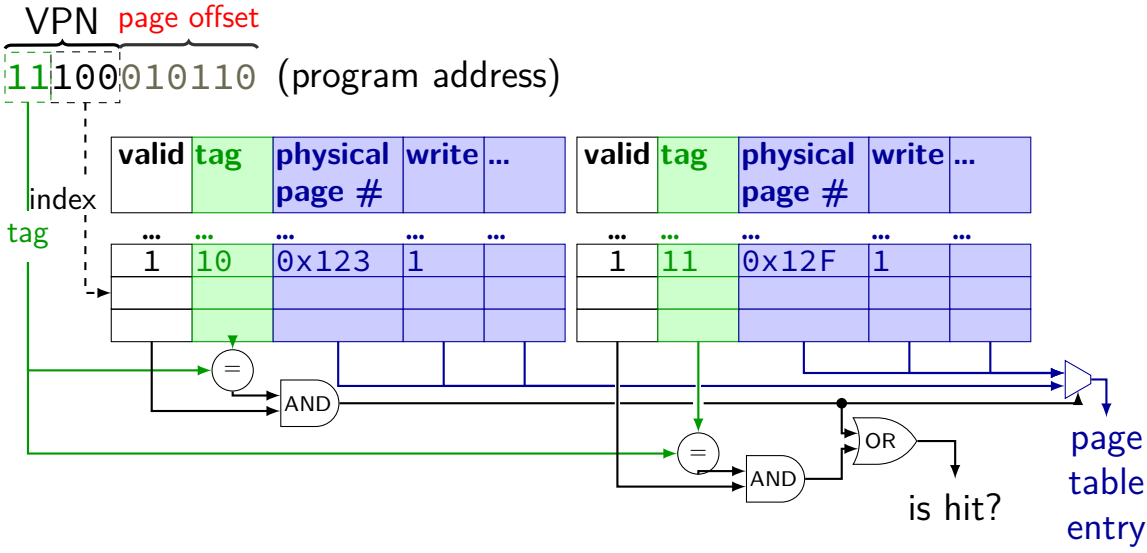
TLB organization (2-way set associative)



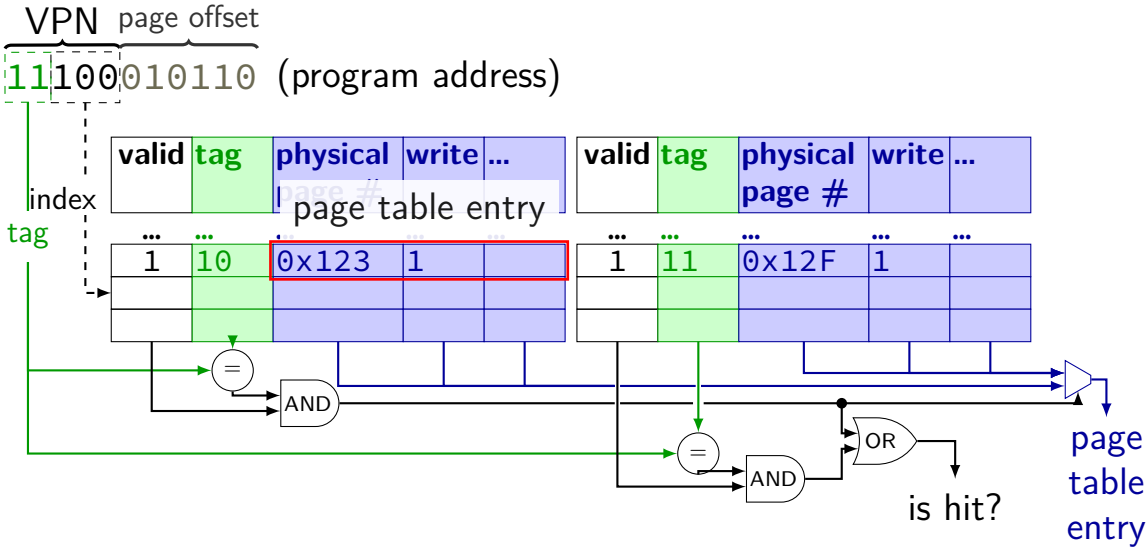
TLB organization (2-way set associative)



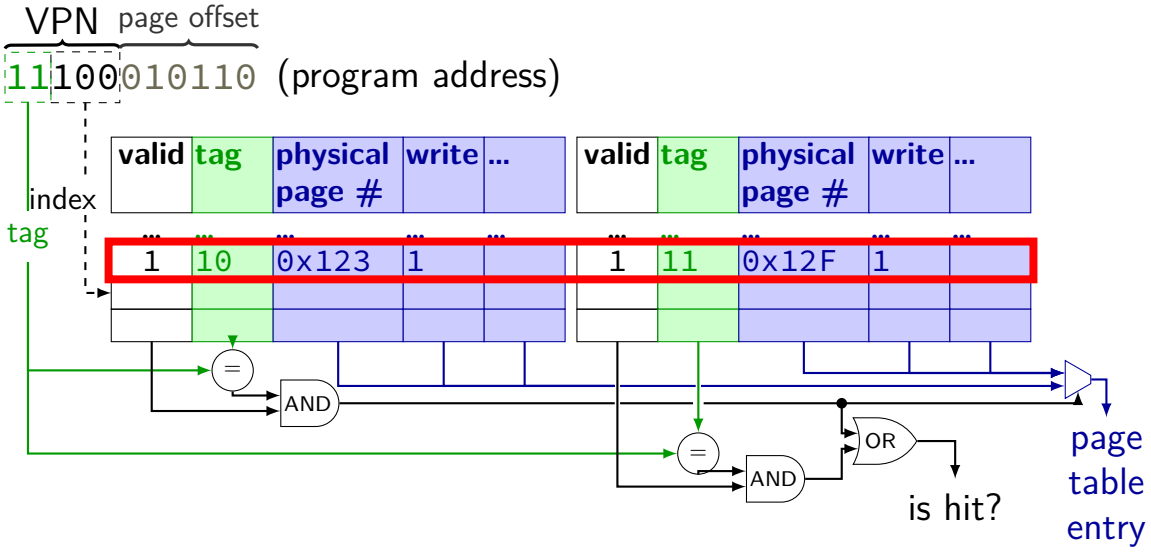
TLB organization (2-way set associative)



TLB organization (2-way set associative)



TLB organization (2-way set associative)



address splitting for TLBs (1)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

64-entry, 4-way L1 data TLB

TLB index bits?

TLB tag bits?

address splitting for TLBs (2)

my desktop:

4KB (2^{12} byte) pages; 48-bit virtual address

1536-entry ($3 \cdot 2^9$), 12-way L2 TLB

TLB index bits?

TLB tag bits?

TLB access pattern example?

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	0					···
1	0					···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	
0x11038	0001 0001	0xFFF038	
0x11040	0001 0001	0xFFF040	
0x7CFF0	0111 1100	0x3100F0	
0x11048	0001 0001	0xFFF048	
0x7CFE8	0111 1100	0x3100E8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	0					···
1	1					···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	
0x11040	0001 0001	0xFFF040	
0x7CFF0	0111 1100	0x3100F0	
0x11048	0001 0001	0xFFF048	
0x7CFE8	0111 1100	0x3100E8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	0					···
1	1	0001000	0xFFF	1	1	···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	
0x11040	0001 0001	0xFFF040	
0x7CFF0	0111 1100	0x3100F0	
0x11048	0001 0001	0xFFF048	
0x7CFE8	0111 1100	0x3100E8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	0					···
1	1	0001000	0xFFF	1	1	···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	hit
0x11040	0001 0001	0xFFF040	
0x7CFF0	0111 1100	0x3100F0	
0x11048	0001 0001	0xFFF048	
0x7CFE8	0111 1100	0x3100E8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	0					···
1	1	0001000	0xFFF	1	1	···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	hit
0x11040	0001 0001	0xFFF040	hit
0x7CFF0	0111 1100	0x3100F0	
0x11048	0001 0001	0xFFF048	
0x7CFE8	0111 1100	0x3100E8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	...
0	0					...
1	1	0001000	0xFFF	1	1	...

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	hit
0x11040	0001 0001	0xFFF040	hit
0x7CFF0	0111 1100	0x3100F0	
0x11048	0001 0001	0xFFF048	
0x7CFE8	0111 1100	0x3100E8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	1	0111110	0x310	1	1	···
1	1	0001000	0xFFF	1	1	···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	hit
0x11040	0001 0001	0xFFF040	hit
0x7CFF0	0111 1100	0x3100F0	miss
0x11048	0001 0001	0xFFF048	
0x7CFE8	0111 1100	0x3100E8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	1	0111110	0x310	1	1	···
1	1	0001000	0xFFF	1	1	···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	hit
0x11040	0001 0001	0xFFF040	hit
0x7CFF0	0111 1100	0x3100F0	miss
0x11048	0001 0001	0xFFF048	hit
0x7CFE8	0111 1100	0x3100E8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	1	0111110	0x310	1	1	···
1	1	0001000	0xFFF	1	1	···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	hit
0x11040	0001 0001	0xFFF040	hit
0x7CFF0	0111 1100	0x3100F0	miss
0x11048	0001 0001	0xFFF048	hit
0x7CFE8	0111 1100	0x3100E8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	1	0111110	0x310	1	1	···
1	1	0001000	0xFFF	1	1	···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	hit
0x11040	0001 0001	0xFFF040	hit
0x7CFF0	0111 1100	0x3100F0	miss
0x11048	0001 0001	0xFFF048	hit
0x7CFE8	0111 1100	0x3100E8	hit
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	1	0111110	0x310	1	1	···
1	1	0001000	0xFFF	1	1	···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	hit
0x11040	0001 0001	0xFFF040	hit
0x7CFF0	0111 1100	0x3100F0	miss
0x11048	0001 0001	0xFFF048	hit
0x7CFE8	0111 1100	0x3100E8	hit
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	1	0111110	0x310	1	1	···
1	1	0001000	0xFFF	1	1	···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	hit
0x11040	0001 0001	0xFFF040	hit
0x7CFF0	0111 1100	0x3100F0	miss
0x11048	0001 0001	0xFFF048	hit
0x7CFE8	0111 1100	0x3100E8	hit
0x30000	0011 0000	0x8FF000	miss
0x7CFE0	0111 0110	0xFFF048	

TLB access pattern example

2-entry, direct-mapped TLB, 4096 byte pages

page table entry

set idx	V	tag	physical page	write?	user?	···
0	1	0111110	0x310	1	1	···
1	1	0001000	0xFFF	1	1	···

virtual	VPN (binary)	physical	hit/miss?
0x11030	0001 0001	0xFFF030	miss
0x11038	0001 0001	0xFFF038	hit
0x11040	0001 0001	0xFFF040	hit
0x7CFF0	0111 1100	0x3100F0	miss
0x11048	0001 0001	0xFFF048	hit
0x7CFE8	0111 1100	0x3100E8	hit
0x30000	0011 0000	0x8FF000	miss
0x7CFE0	0111 0110	0xFFF048	

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

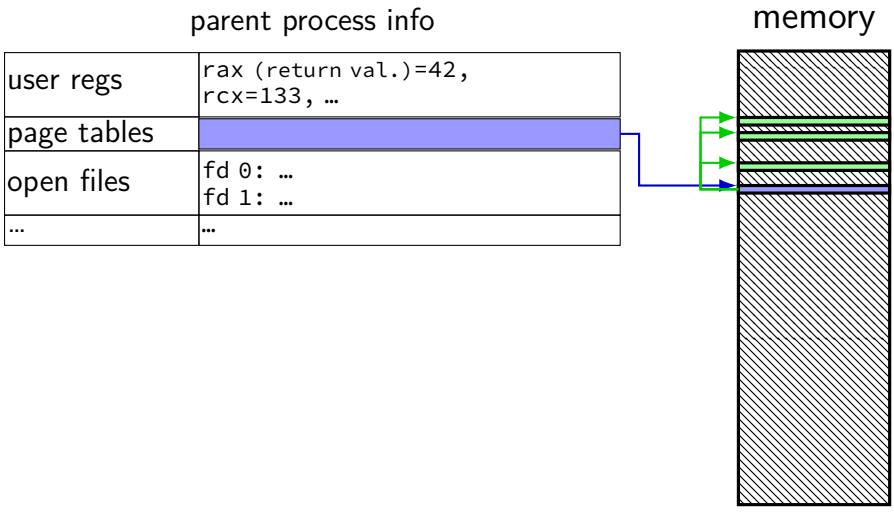
returns twice:

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

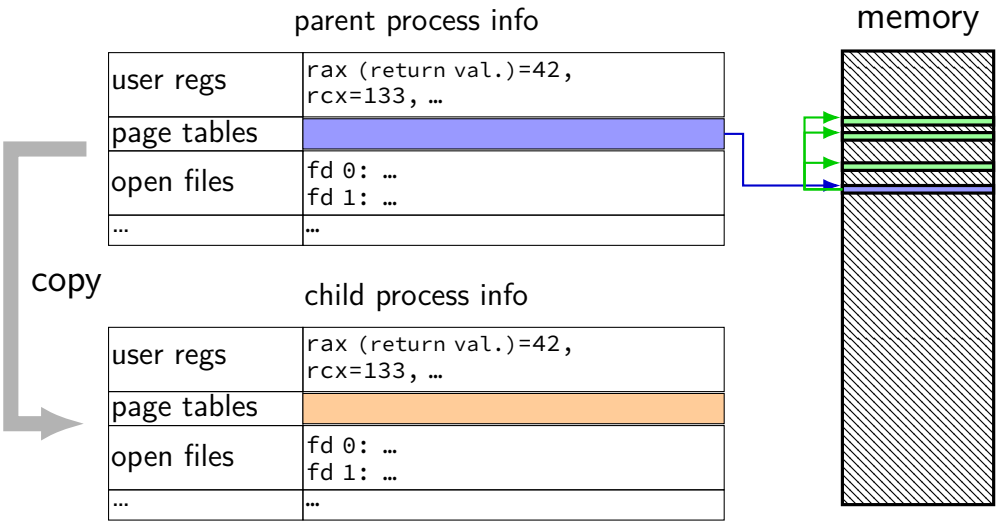
everything (but pid) duplicated in parent, child:

- memory
- file descriptors (later)
- registers

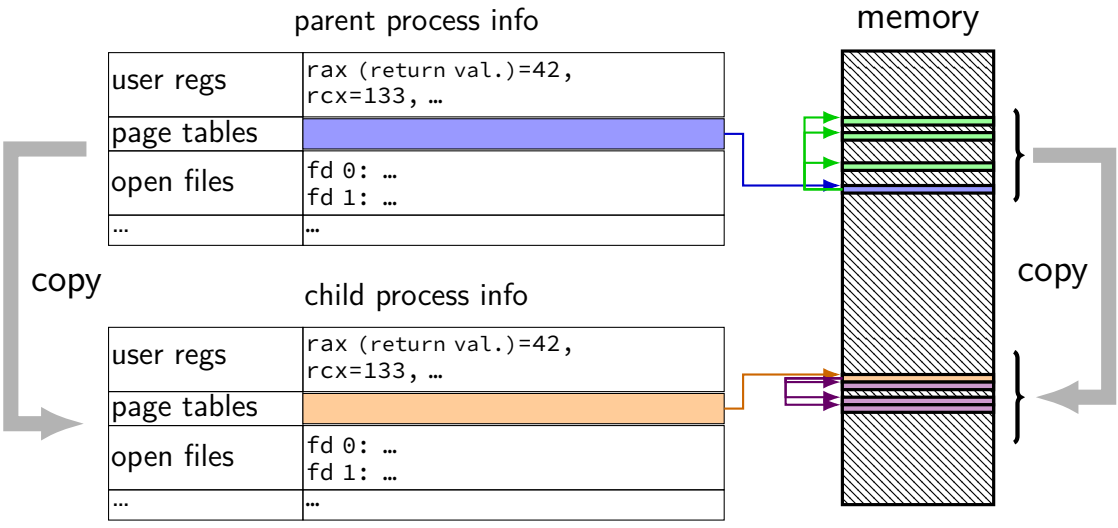
fork and process info (w/o copy-on-write)



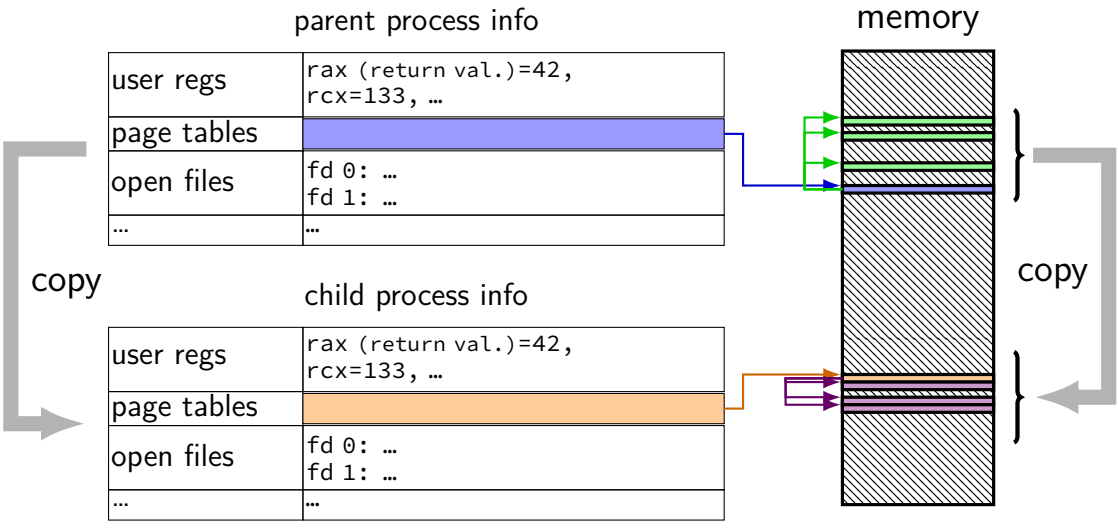
fork and process info (w/o copy-on-write)



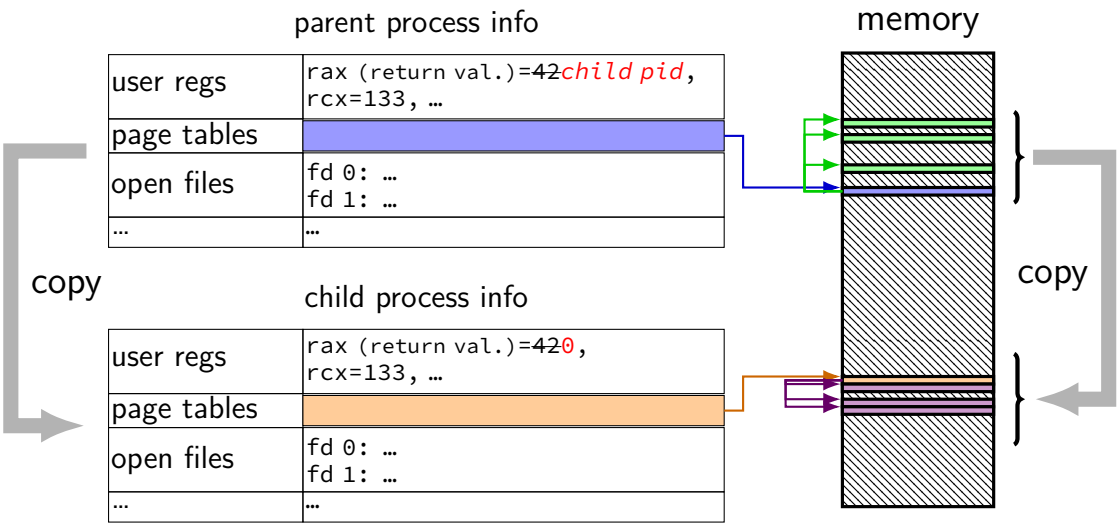
fork and process info (w/o copy-on-write)



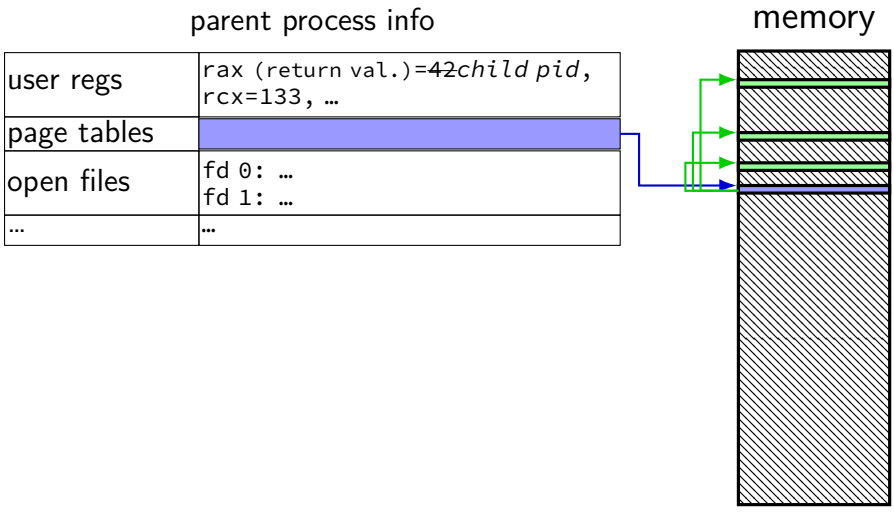
fork and process info (w/o copy-on-write)



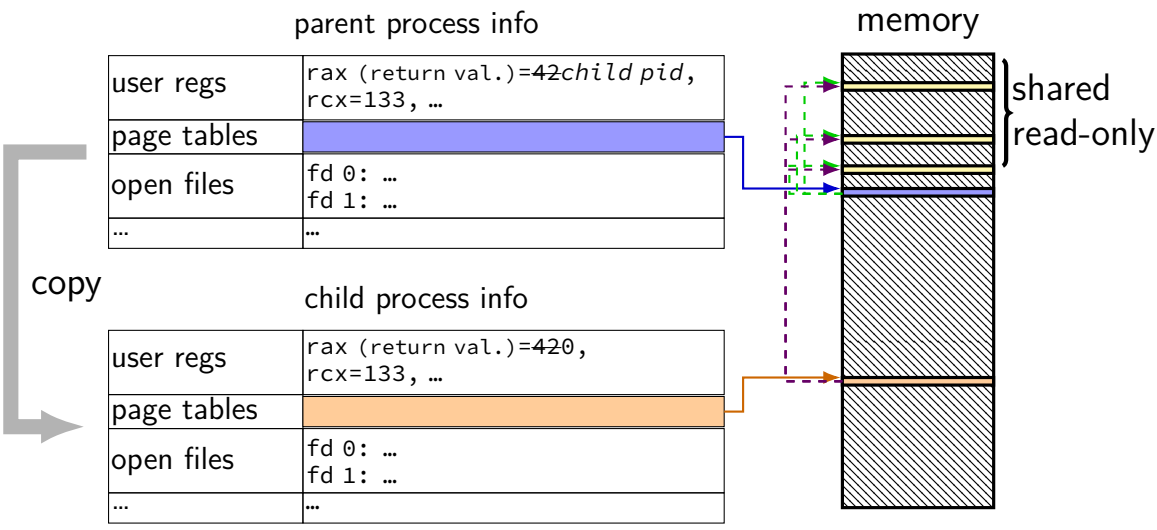
fork and process info (w/o copy-on-write)



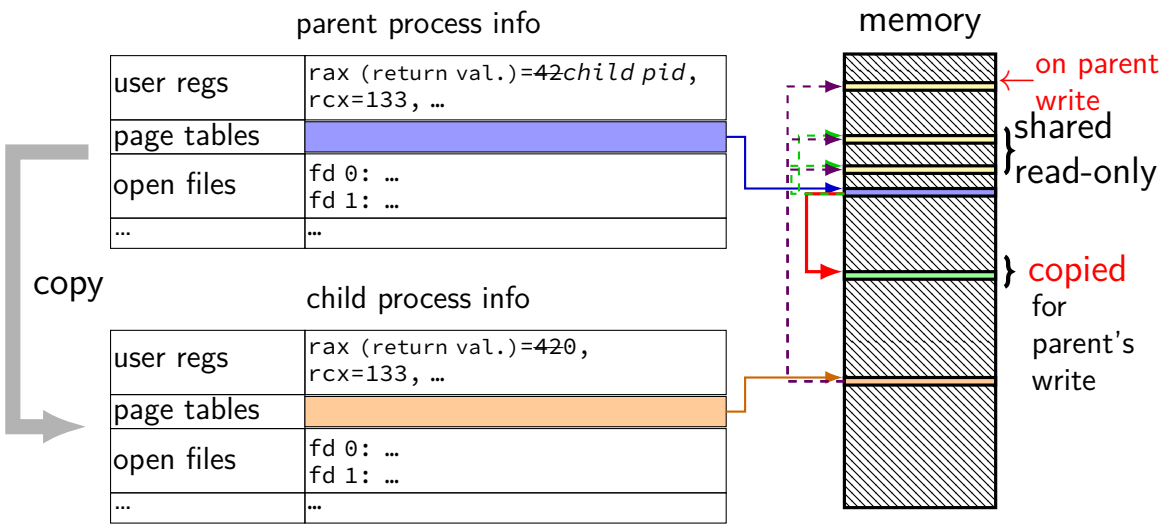
fork (w/ copy-on-write, if parent writes first)



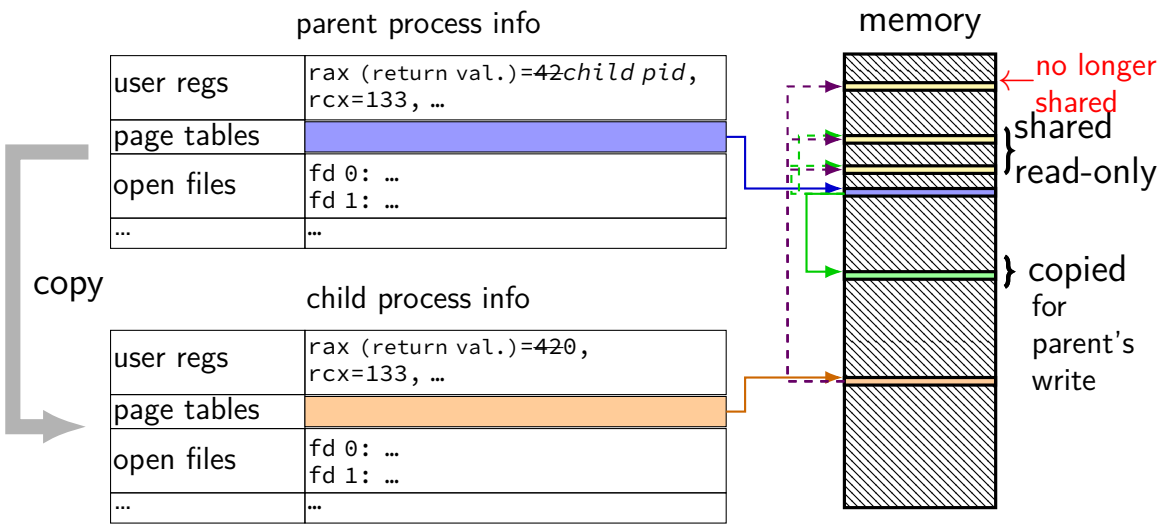
fork (w/ copy-on-write, if parent writes first)



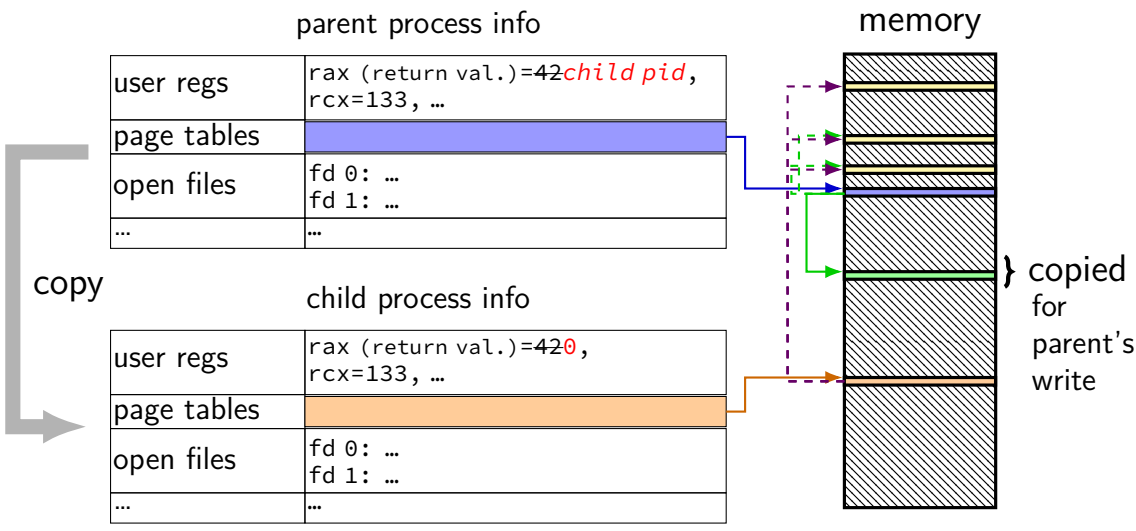
fork (w/ copy-on-write, if parent writes first)



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



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



fork example

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

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
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;
}
```

`getpid` — returns current process pid

fork example

```
#include <stdlib.h>
#include <stdio.h>
#include <unis
#include <sys/
int main(int a
    pid_t pid
    printf("Pa
    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;
}
```

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

```
#include <stdlib.h>
#include <stdio.h>
#include
#include
int main
    pid_
    prin
    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;
}
```

prints out `Fork failed: error message`
(example *error message*: "Resource temporarily unavailable")
from error number stored in special global variable `errno`

fork example

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
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;
}
```

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.)

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

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

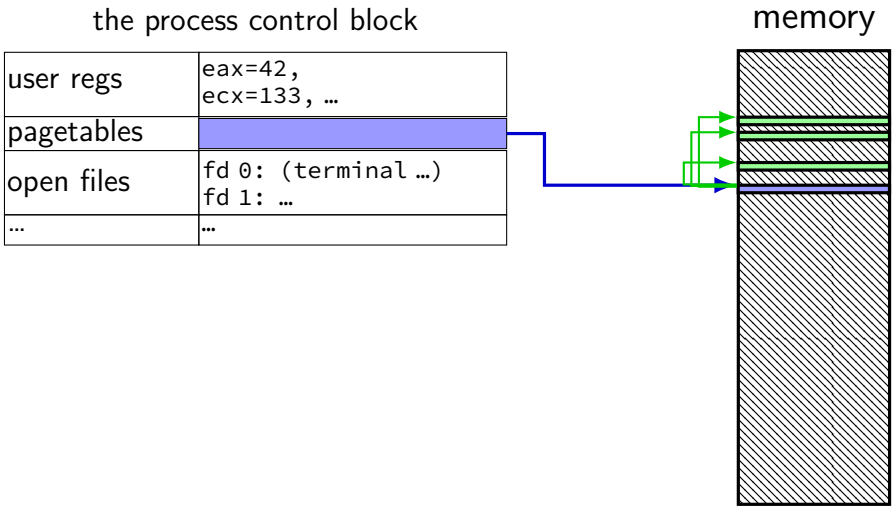
execv example

```
...
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 */
  ...
}
```

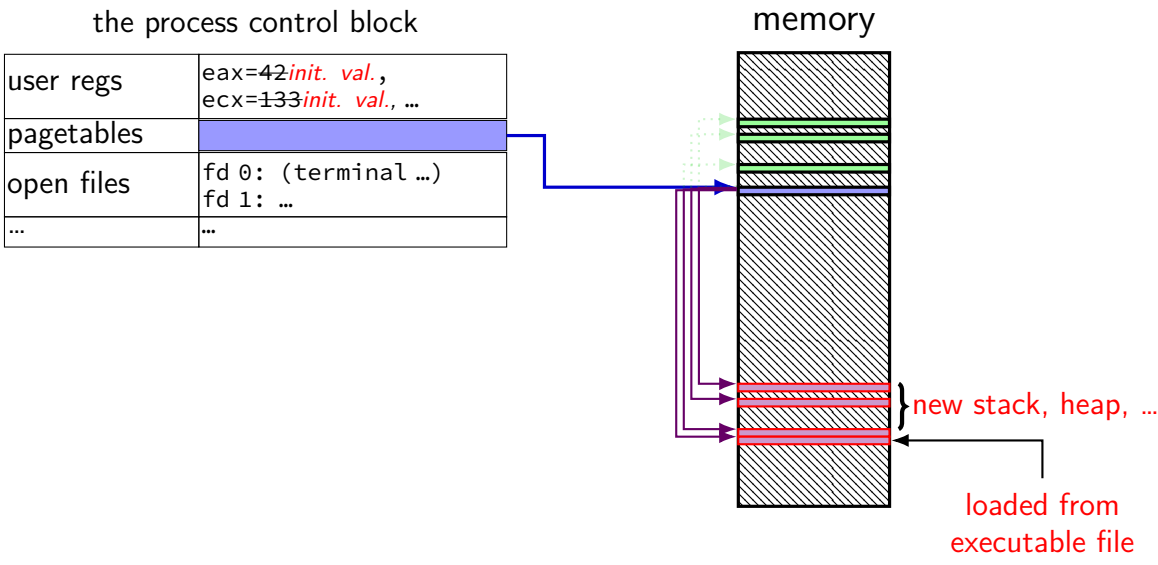
path of executable to run
need not match first argument
(but probably should match it)

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

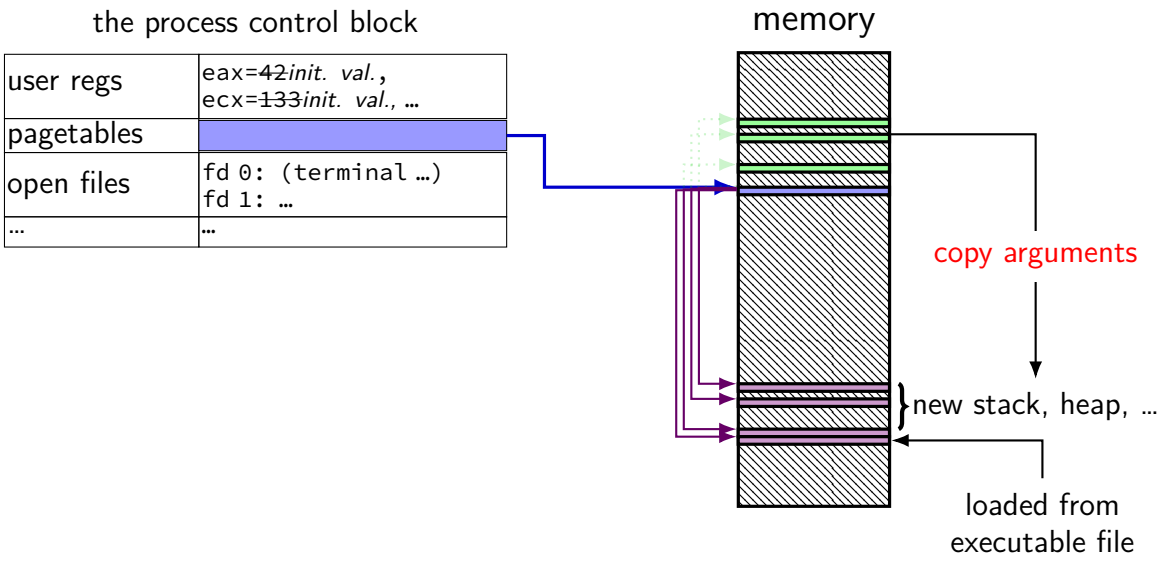
exec in the kernel



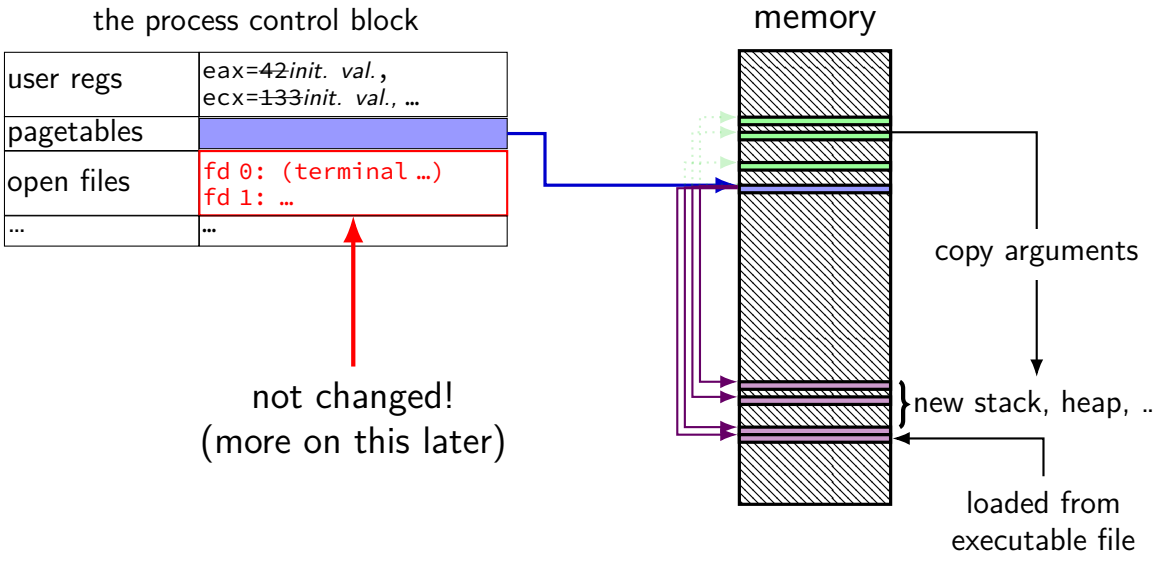
exec in the kernel



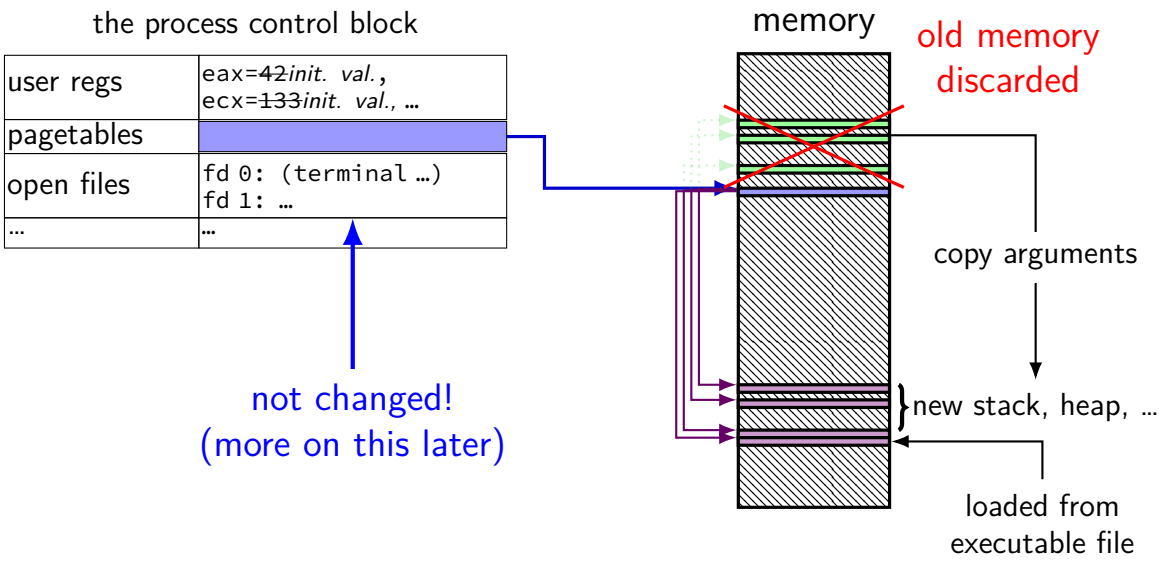
exec in the kernel



exec in the kernel



exec in the kernel



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

exit statuses

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

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 */
      ...
```

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

aside: signals

signals are a way of communicating between processes

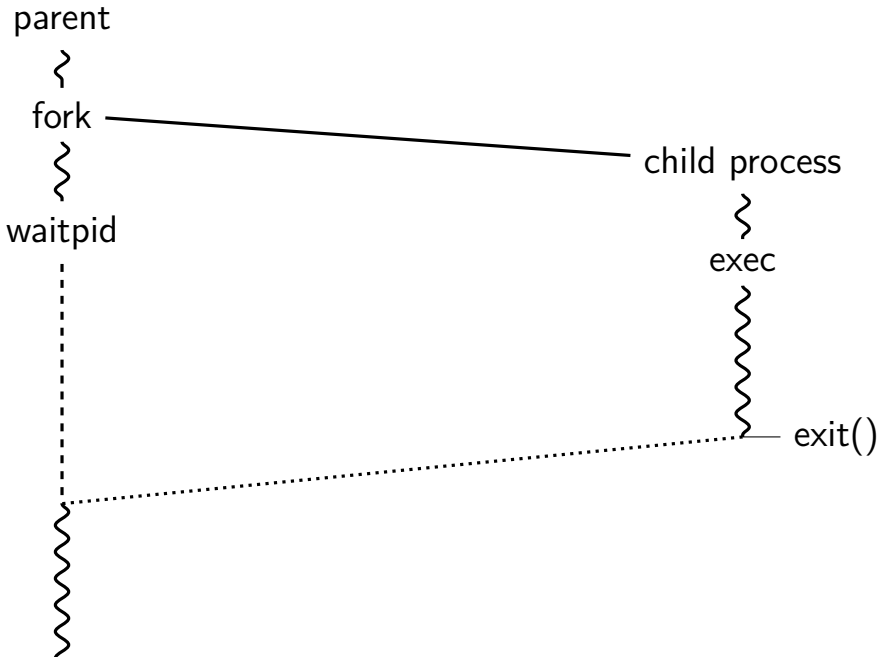
they are also how abnormal termination happens

- kernel communicating "something bad happened" → kills program by default

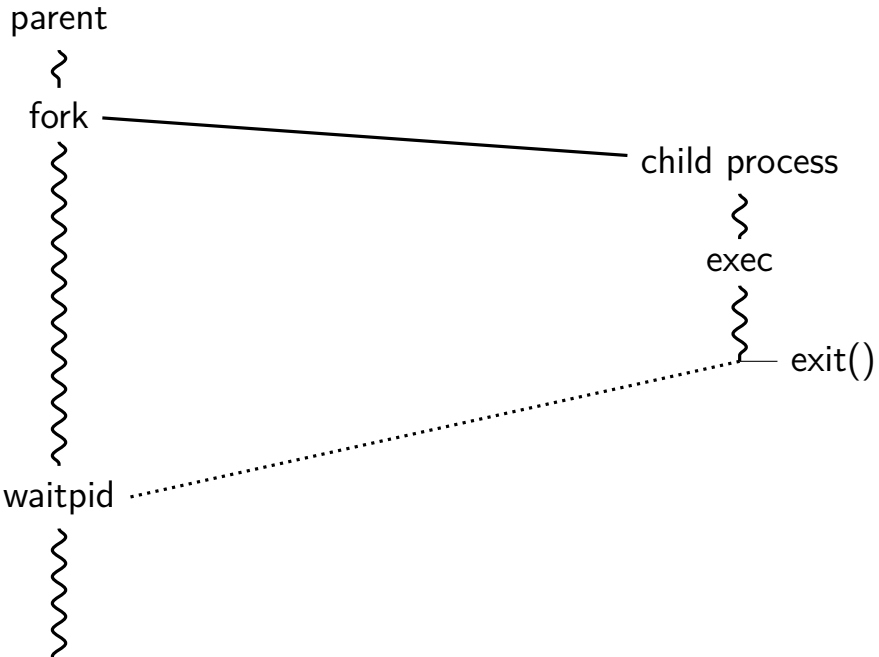
wait's status will tell you when and what signal killed a program

- constants in signal.h
- SIGINT — control-C
- SIGTERM — `kill` command (by default)
- SIGSEGV — segmentation fault
- SIGBUS — bus error
- SIGABRT — `abort()` library function
- ...

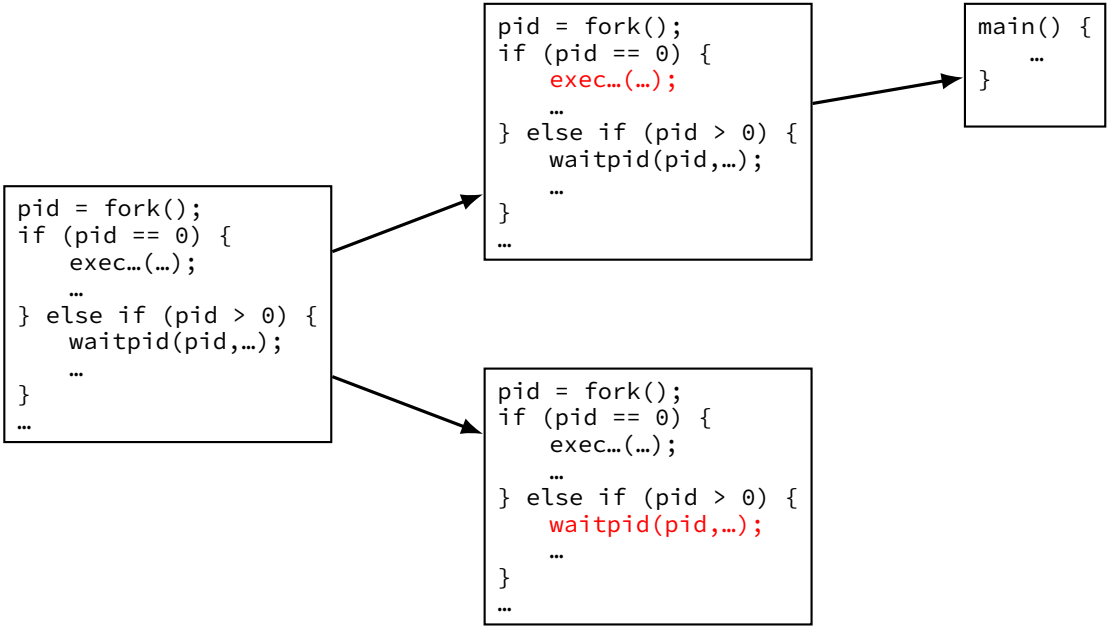
typical pattern



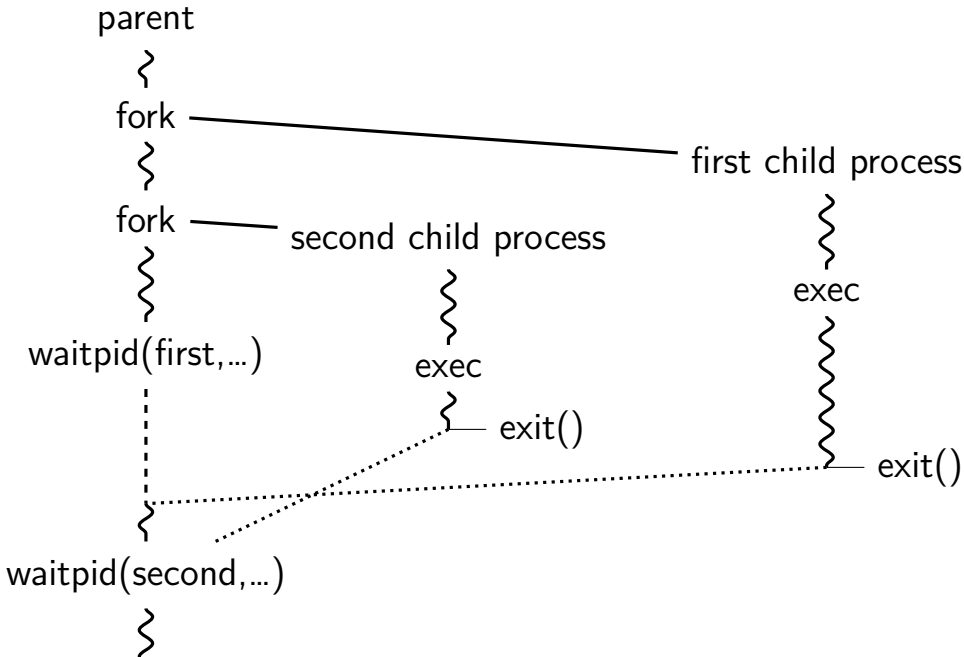
typical pattern (alt)



typical pattern (detail)



pattern with multiple?



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`

exercise (1)

```
int main() {
    pid_t pids[2]; const char *args[] = {"echo", "ARG", NULL};
    const char *extra[] = {"L1", "L2"};
    for (int i = 0; i < 2; ++i) {
        pids[i] = fork();
        if (pids[i] == 0) {
            args[1] = extra[i];
            execv("/bin/echo", args);
        }
    }
    for (int i = 0; i < 2; ++i) {
        waitpid(pids[i], NULL, 0);
    }
}
```

Assuming fork and execv do not fail, which are possible outputs?

A. L1 (newline) L2

B. L1 (newline) L2 (newline) L2

C. L2 (newline) L1

D. A and B

E. A and C

F. all of the above

G. something else

exercise (2)

```
int main() {
    pid_t pids[2]; const char *args[] = {"echo", "0", NULL};
    for (int i = 0; i < 2; ++i) {
        pids[i] = fork();
        if (pids[i] == 0) { execv("/bin/echo", args); }
    }
    printf("1\n"); fflush(stdout);
    for (int i = 0; i < 2; ++i) {
        waitpid(pids[i], NULL, 0);
    }
    printf("2\n"); fflush(stdout);
}
```

Assuming fork and execv do not fail, which are possible outputs?

A. `0` (newline) `0` (newline) `1` (newline) `2`

B. `0` (newline) `1` (newline) `0` (newline) `2`

C. `1` (newline) `0` (newline) `0` (newline) `2`

D. `1` (newline) `0` (newline) `2` (newline) `0`

E. A, B, and C

F. C and D

G. all of the above

H. something else

threads versus processes

so far, was showing each process has one thread

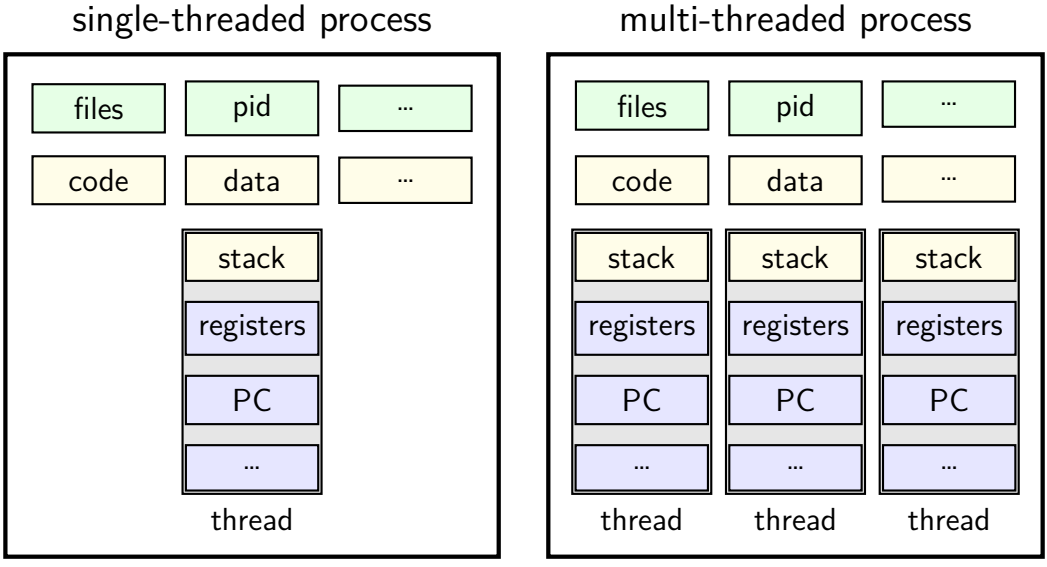
thread = part that gets run on CPU

- saved register values (including own stack pointer)
- save program counter

rest of process

- address space (accessible memory)
- open files
- current working directory
- ...

single and multithread processes



thread versus process state

thread state

- registers (including stack pointer, program counter)
- ...

process state

- address space
- open files
- process id
- list of thread states
- ...

process info with threads

parent process info

thread infos	`thread 0: {PC = 0x123456, rax = 42, rbx = …}`
	`thread 1: {PC = 0x584390, rax = 32, rbx = …}`
	…

page tables	
open files	`fd 0: …` `fd 1: …`
…	…

Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared

- e.g. shared open files: open fd 4 in one task → all sharing can use fd 4

`fork()`-like system call "clone": choose what to share

- `clone(0, ...)` — similar to `fork()`
- `clone(CLONE_FILES, ...)` — like fork(), but **sharing** open files
- `clone(CLONE_VM, new_stack_pointer, ...)` — like fork(), but **sharing** address space

Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared

- e.g. shared open files: open fd 4 in one task → all sharing can use fd 4

`fork()`-like system call "clone": choose what to share

- `clone(0, ...)` — similar to `fork()`
- `clone(CLONE_FILES, ...)` — like fork(), but **sharing** open files
- `clone(CLONE_VM, new_stack_pointer, ...)` — like fork(), but **sharing** address space

advantage: no special logic for threads (mostly)

- two threads in same process = tasks sharing everything possible

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?

some POSIX command-line features

searching for programs

```
ls -l ≈ /bin/ls -l
make ≈ /usr/bin/make
```

running in background

```
./someprogram &
```

redirection:

```
./someprogram >output.txt
./someprogram <input.txt
```

pipelines:

```
./someprogram | ./somefilter
```

some POSIX command-line features

searching for programs

```
ls -l ≈ /bin/ls -l
make ≈ /usr/bin/make
```

running in background

```
./someprogram &
```

redirection:

```
./someprogram >output.txt
./someprogram <input.txt
```

pipelines:

```
./someprogram | ./somefilter
```

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);
}
```

some POSIX command-line features

searching for programs

```
ls -l ≈ /bin/ls -l
make ≈ /usr/bin/make
```

running in background

```
./someprogram &
```

redirection:

```
./someprogram >output.txt
./someprogram <input.txt
```

pipelines:

```
./someprogram | ./somefilter
```

some POSIX command-line features

searching for programs

```
ls -l ≈ /bin/ls -l
make ≈ /usr/bin/make
```

running in background

```
./someprogram &
```

redirection:

```
./someprogram >output.txt
./someprogram <input.txt
```

pipelines:

```
./someprogram | ./somefilter
```

file descriptors

```
struct process_info {
    ...
    struct open_file *files;
};
...
process->files[file_descriptor]
```

Unix: every process has
array (or similar) of *open file descriptions*

"open file": terminal · socket · regular file · pipe

file descriptor = index into array

usually what's used with system calls

stdio.h FILE*s usually have file descriptor + buffer

special file descriptors

file descriptor 0 = standard input

file descriptor 1 = standard output

file descriptor 2 = standard error

constants in `unistd.h`

`STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`

special file descriptors

file descriptor 0 = standard input

file descriptor 1 = standard output

file descriptor 2 = standard error

constants in `unistd.h`

- `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`

but you can't choose which number `open` assigns...?

- more on this later

getting file descriptors

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

used internally by fopen(), etc.

also for files without normal filenames...:

```
int fd = shm_open("/shared_memory", O_RDWR, 0666); // shared memory
int socket_fd = socket(AF_INET, SOCK_STREAM, 0); // TCP socket
int term_fd = posix_openpt(O_RDWR); // pseudo-terminal
int pipe_fds[2]; pipe(pipefds); // "pipes" (later)
...
```

close

```
int close(int fd);
```

close the file descriptor, deallocating that array index

- does not affect other file descriptors
- that refer to same "open file description"
- (e.g. in `fork()`ed child or created via (later) `dup2`)

if last file descriptor for open file description, resources deallocated

returns 0 on success

returns -1 on error

- e.g. ran out of disk space while finishing saving file

shell redirection

`./my_program ... < input.txt`:

run `./my_program ...` but use `input.txt` as input

like we copied and pasted the file into the terminal

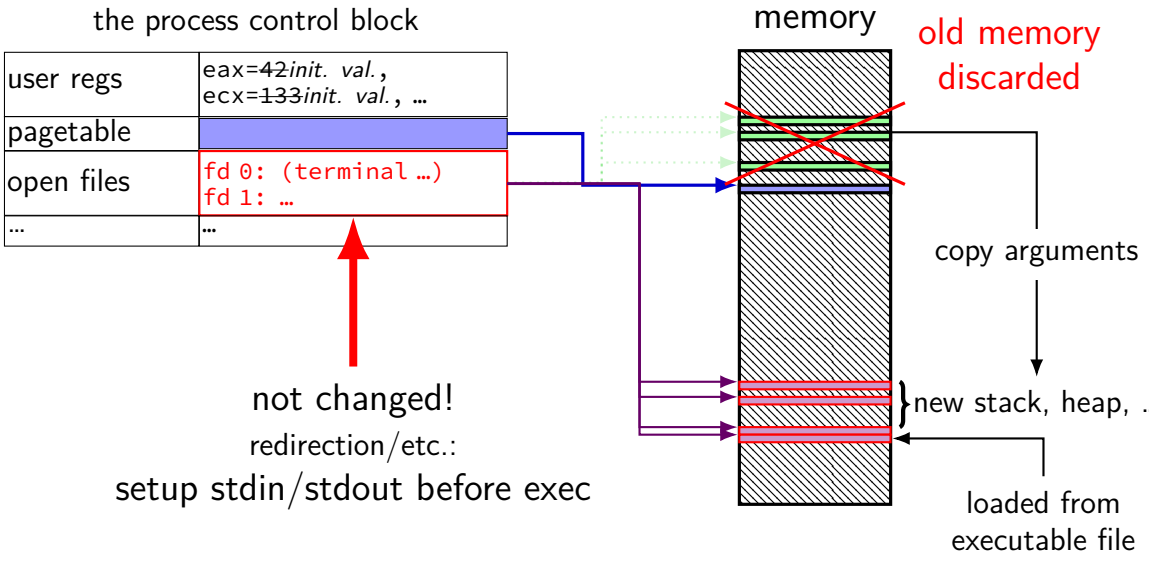
`echo foo > output.txt`:

runs `echo foo`, sends output to `output.txt`

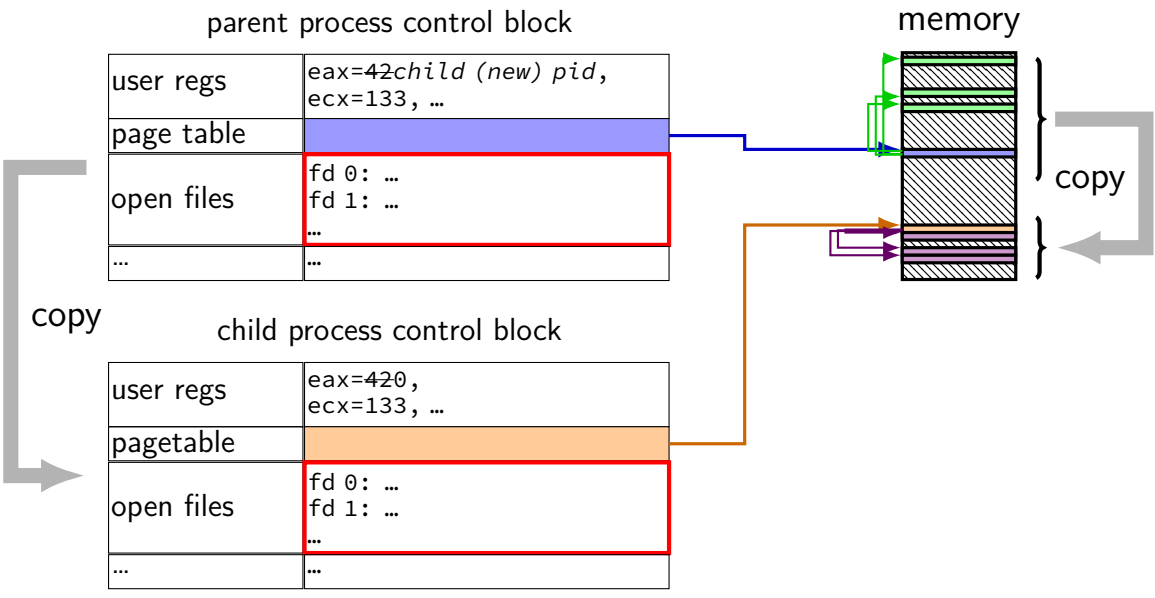
like we copied and pasted the output into that file

(as it was written)

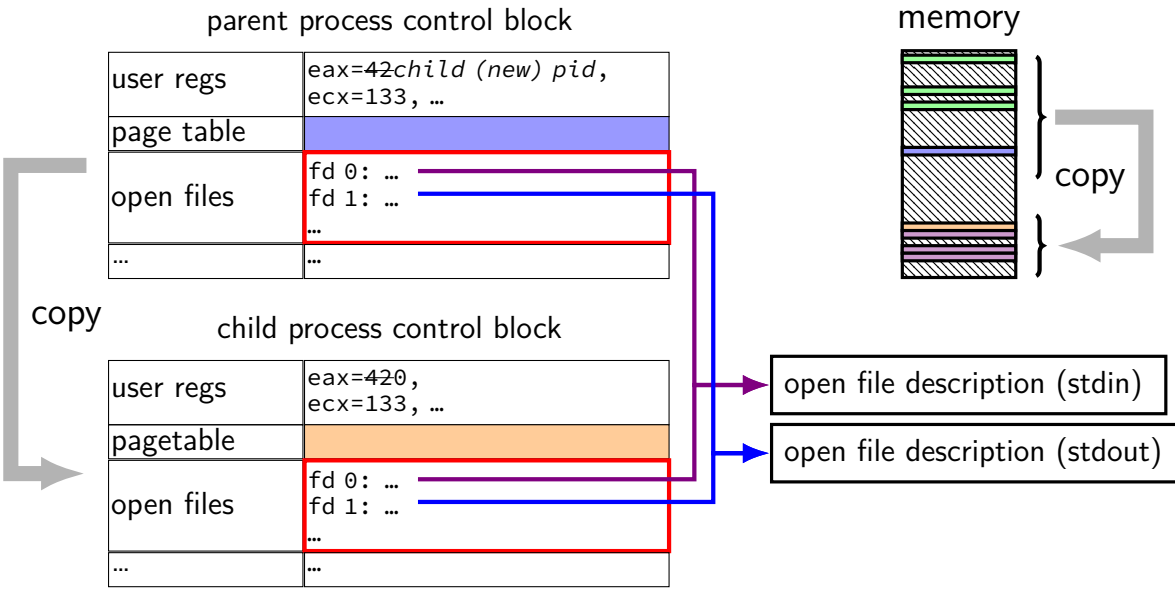
exec preserves open files



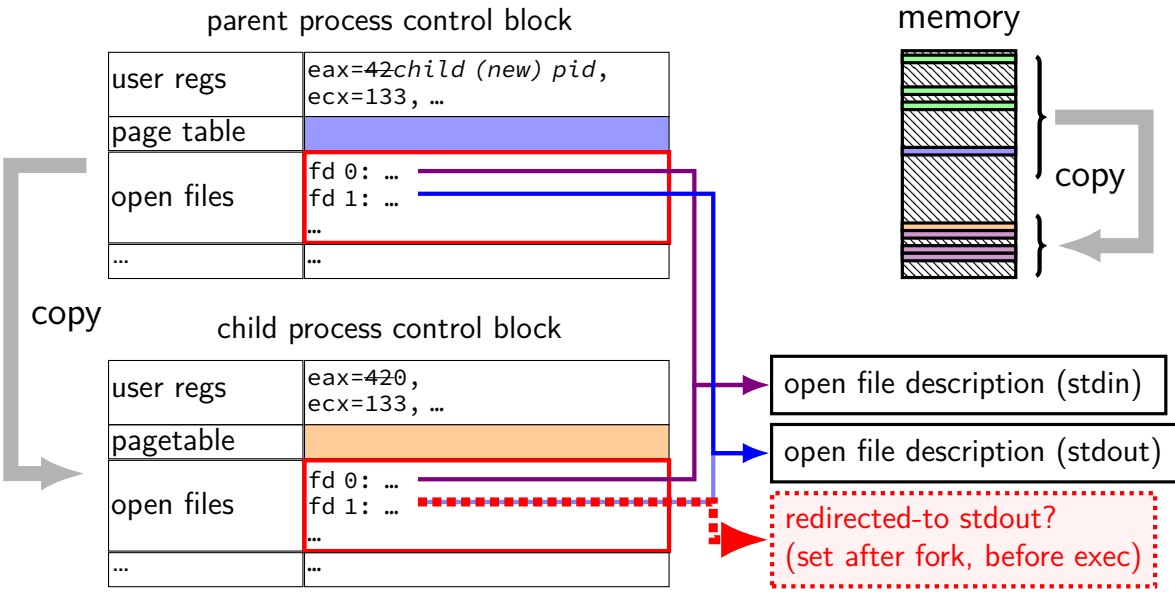
fork copies open file list



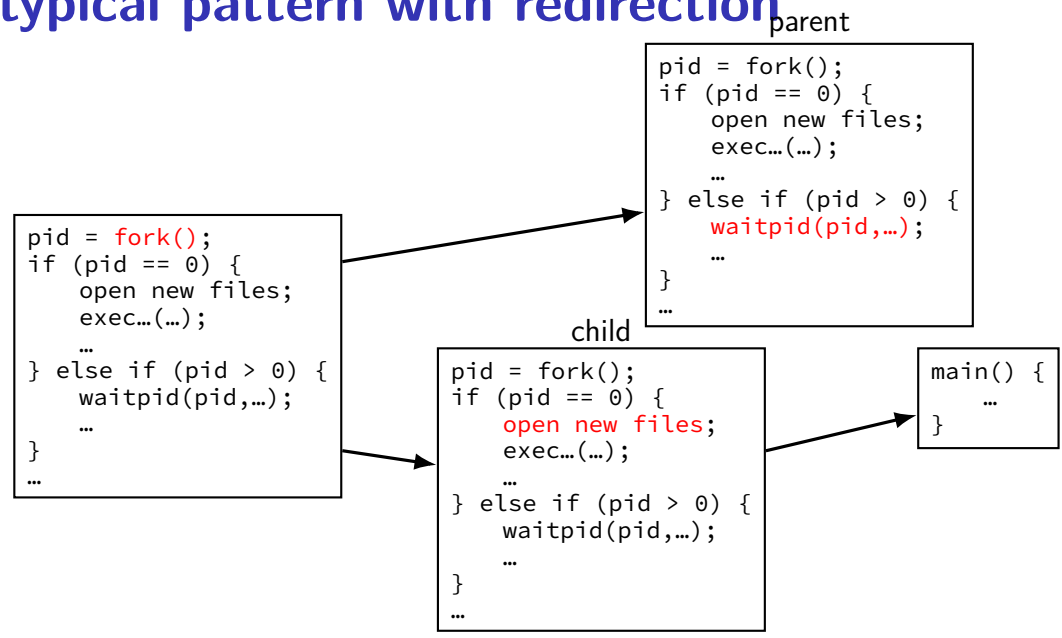
fork copies open file list



fork copies open file list



typical pattern with redirection



redirecting with exec

standard output/error/input are files

(C stdout/stderr/stdin; C++ cout/cerr/cin)

(probably after forking) open files to redirect

...and make them be standard output/error/input

using `dup2()` library call

then exec, preserving new standard output/etc.

reassigning file descriptors

redirection: `./program >output.txt`

step 1: open output.txt for writing, get new file descriptor

step 2: make that new file descriptor stdout (number 1)

reassigning and file table

```
struct process_info {
    ...
    struct open_file *files;
};
...
process->files[STDOUT_FILENO] = process->files[opened-fd];
```

syscall: dup2(*opened-fd*, STDOUT_FILENO);

reassigning file descriptors

redirection: `./program >output.txt`

step 1: open output.txt for writing, get new file descriptor

step 2: make that new file descriptor stdout (number 1)

tool: `int dup2(int oldfd, int newfd)`
make `newfd` refer to same open file as `oldfd`

- same *open file description*
- shares the current location in the file
- (even after more reads/writes)

what if newfd already allocated — closed, then reused

dup2 example

redirects stdout to output to `output.txt`:

```
fflush(stdout);  /* clear printf's buffer */
int fd = open("output.txt",
              O_WRONLY | O_CREAT | O_TRUNC);
if (fd < 0)
    do_something_about_error();

dup2(fd, STDOUT_FILENO);
/* now both write(fd, ...) and write(STDOUT_FILENO, ...)
   write to output.txt
   */

close(fd); /* only close original, copy still works! */

printf("This will be sent to output.txt.\n");
```

open/dup/close/etc. and fd array

```
struct process_info {
  ...
  struct file *files;
};
```

open: `files[new_fd] = ...;`

dup2(from, to): `files[to] = files[from];`

close: `files[fd] = NULL;`

fork:

```
for (int i = ...)
    child->files[i] = parent->files[i];
```

(plus extra work to avoid leaking memory)

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 **C.** ABC **E.** something else
B. ABCD **D.** ACD

pipes

special kind of file: pipes

bytes go in one end, come out the other — once

created with `pipe()` library call

intended use: communicate between processes

- like implementing shell pipelines

pipe()

```
int pipe_fd[2];
if (pipe(pipe_fd) < 0)
    handle_error();
/* normal case: */
int read_fd = pipe_fd[0];
int write_fd = pipe_fd[1];
```

then from one process...

```
write(write_fd, ...);
```

and from another

```
read(read_fd, ...);
```

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?

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 */ }
```

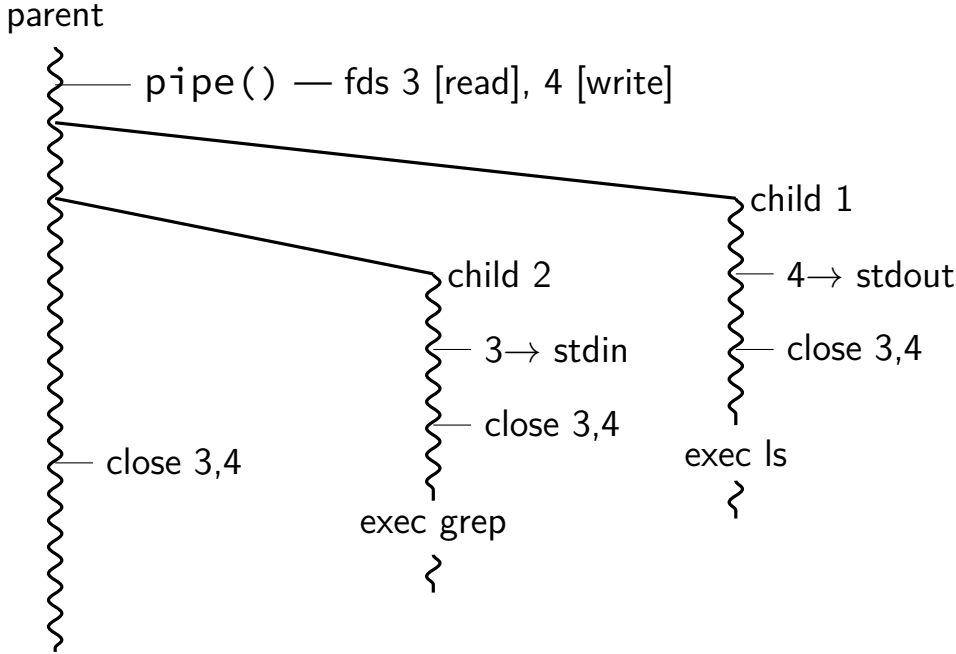
have habit of closing
to avoid 'leaking' file descriptors
you can run out

pipe and pipelines

```
ls -1 | grep foo
```

```
pipe(pipe_fd);
ls_pid = fork();
if (ls_pid == 0) {
    dup2(pipe_fd[1], STDOUT_FILENO);
    close(pipe_fd[0]); close(pipe_fd[1]);
    char *argv[] = {"ls", "-1", NULL};
    execv("/bin/ls", argv);
}
grep_pid = fork();
if (grep_pid == 0) {
    dup2(pipe_fd[0], STDIN_FILENO);
    close(pipe_fd[0]); close(pipe_fd[1]);
    char *argv[] = {"grep", "foo", NULL};
    execv("/bin/grep", argv);
}
close(pipe_fd[0]); close(pipe_fd[1]);
/* wait for processes, etc. */
```

example execution



exercise

```
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

backup slides

exercise: TLB access pattern (setup)

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

how many index bits?

TLB index of virtual address 0x12345?

exercise: TLB access pattern

4-entry, 2-way TLB, LRU replacement policy, initially empty

4096 byte pages

type	virtual	physical
read	0x440030	0x554030
write	0x440034	0x554034
read	0x7FFFE008	0x556008
read	0x7FFFE000	0x556000
read	0x7FFFDFF8	0x5F8FF8
read	0x664080	0x5F9080
read	0x440038	0x554038
write	0x7FFFDFF0	0x5F8FF0

which are TLB hits? which are TLB misses? final contents of TLB?

changing page tables

what happens to TLB when page table base pointer is changed?

e.g. context switch

most entries in TLB refer to things from wrong process

oops — read from the wrong process's stack?

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

option 1: invalidate all TLB entries

- side effect on "change page table base register" instruction

changing page tables

what happens to TLB when page table base pointer is changed?

- e.g. context switch

most entries in TLB refer to things from wrong process

- oops — read from the wrong process's stack?

option 1: invalidate all TLB entries

- side effect on "change page table base register" instruction

option 2: TLB entries contain process ID

- set by OS (special register)
- checked by TLB in addition to TLB tag, valid bit

editing page tables

what happens to TLB when OS changes a page table entry?

most common choice: has to be handled in software

editing page tables

what happens to TLB when OS changes a page table entry?

most common choice: has to be handled in software

invalid to valid — nothing needed

- TLB doesn't contain invalid entries
- MMU will check memory again

valid to invalid — OS needs to tell processor to invalidate it

- special instruction (x86: `invlpg`)

valid to other valid — OS needs to tell processor to invalidate it

backup slides