so far

building programs

hardware and OSes

- exceptions, context switching
- virtual memory

OS abstractions

- system calls and permissions

networking layers

cryptography and ensuring authenticity/secrecy

caches

last time

review of set-associative caches

write policies

- write-allocate / write-no-allocate (if not present, ...)
- write-back / write-through (if present, ...)

compulsory AKA cold / conflict / capacity / coherence

- identify what changes needed to eliminate cache misses

anonymous feedback (1)

"Would you mind explaining question 6 step by step on quiz 6, please? Thanks!"

feedback from last Tuesday evening, I think I'm late
question asked about access to index 4, tag 001101
look at index 4 in cache; that tag not present in valid block
so, we need to add a block with that tag to that index
must replace an existing block
was direct mapped cache, only one candidate
question asked for contents (data) of block replaced

anonymous feedback (2)

"i'm doing the part two of the lab right now and i am on the verge of tears - it's pretty straightforward and it's nothing abt the actual computations that's bad it's just that... I have submitted this EIGHT TIMES ALREADY with most of them having a score of 39/40 i'm just annoyed more than anything most of the time it's minor errors, but i guess i really just am not understanding which addresses to evict because the ones i think are correct sometimes are, but sometimes aren't and i'm failing to see the pattern"

intention was you'd get help in lab/OH to understand this

(also you could get in-person lab checkoff with less than 40/40)

I think most common misunderstanding was not handling least recently used policy

(thing you replace might not be least recently inserted, since that's not same as least recently used)

anonymous feedback (3)

"Among all the negative feedback, I wanted to give you some positive feedback. You are a great professor, people just always have something to complain about. The fact that you look at anonymous feedback and use it to continuously improve the class is already much more than what other professors do. Additionally, you are very well versed in the content of the class; there is almost no question that you do not have the answer to. People tend to get frustrated with the content of the course because of its difficulty and end up taking it out on you in the anonymous feedback, which you do not deserve. So please continue doing what your doing, and I hope this message raises your spirit :)

anonymous feedback (4)

"In my opinion, the content that was taught in class was not sufficient for our proficiency in this Quiz 7. I feel very stressed T.T"

"I wish the examples during class would be more in-depth in that they should go step by step and be as detailed as possible. I feel that you assume we know more than we know, many things that you point out as intuitive are not so for me."

> not sure which examples referred to; most of the cache examples w/o C code were very specific. (I agree the ones with C code could be more specific)

anonymous feedback (5)

"I feel like the content of the class does not match the difficulties of the quizzes where I understand what is going on in class, but not really what is being asked on the quizzes. I try doing the readings to get a better grasp of the context, but I feel like the reading does not encapsulate the material well and is a little too simple. Is there any resources like texts or websites available?"

added some links to additional resources (which I think mostly duplicate the cache reading) to the bottom of the cache reading

probably need to examine this for other topics

Q4/5

```
unsigned char array1[4096];
unsigned char array2[4096];
...
for (int i = 0; i < 2048; ++i) {
    array1[i] = array2[i + 2048] * array2[i];
}
```

(assuming no compiler tricks) 2048 writes to array1

$2048 \div 64 = 32$ cache blocks written to

write-no-allocate: 2048 writes to next level

write-allocate + write-through: 32 reads + 2048 writes to next level

write-allocate + write-back: 32 reads (to fill in rest of block on first write)

schedule note

want to make sure fork/exec covered for lab tomorrow, so might skip ahead

(and return back later)

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

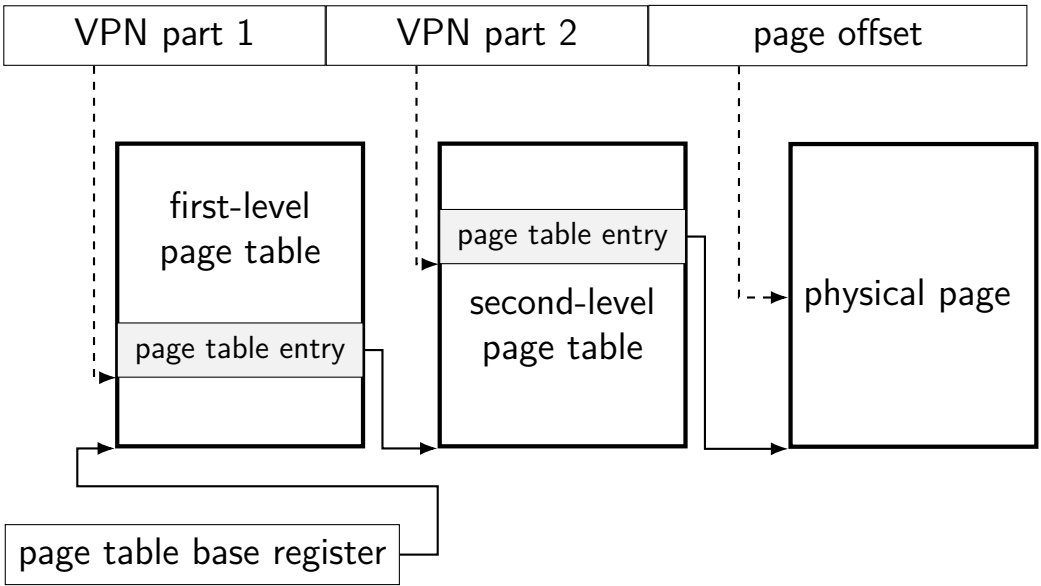
$$\text{average time} = \text{hit time} + \text{miss rate} \times \text{miss penalty}$$

cache optimizations by miss type

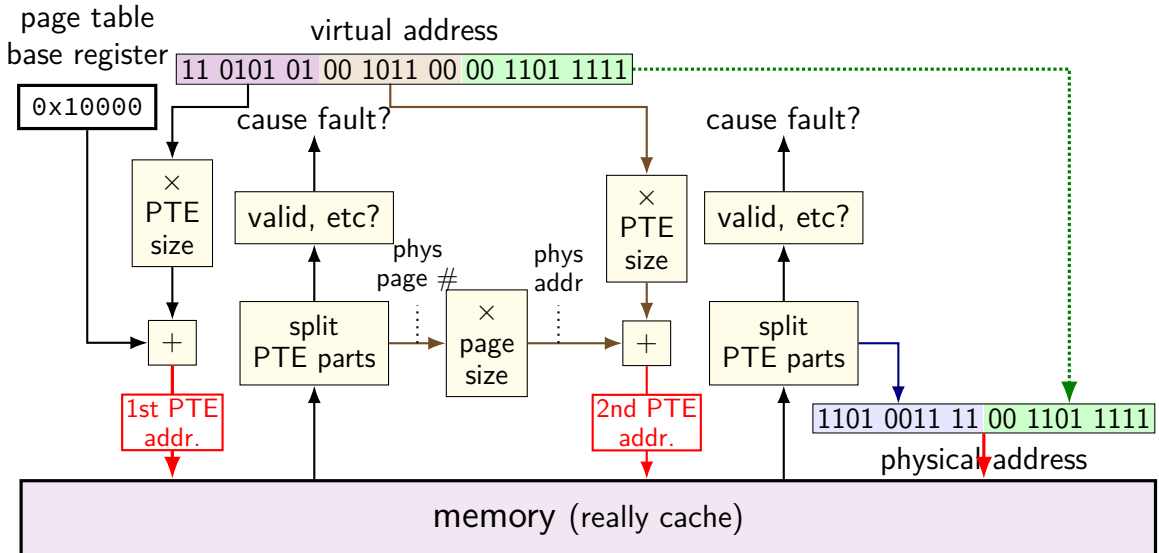
(assuming other listed parameters remain constant)

	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

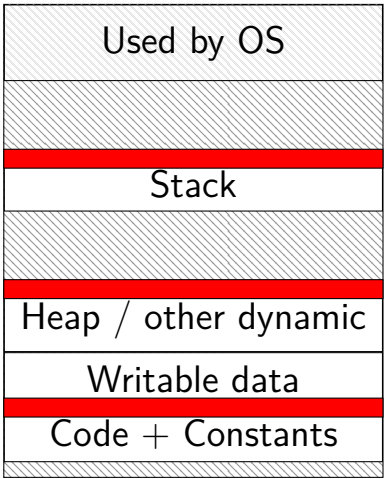
four-level page tables — five cache accesses per program memory access

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

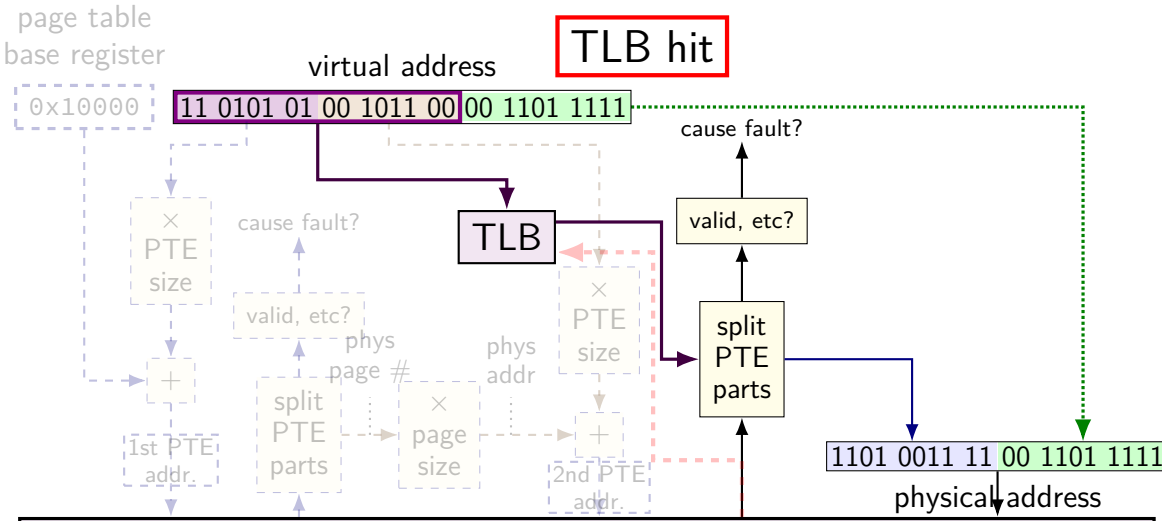
TLB and multi-level page tables

TLB caches valid last-level page table entries

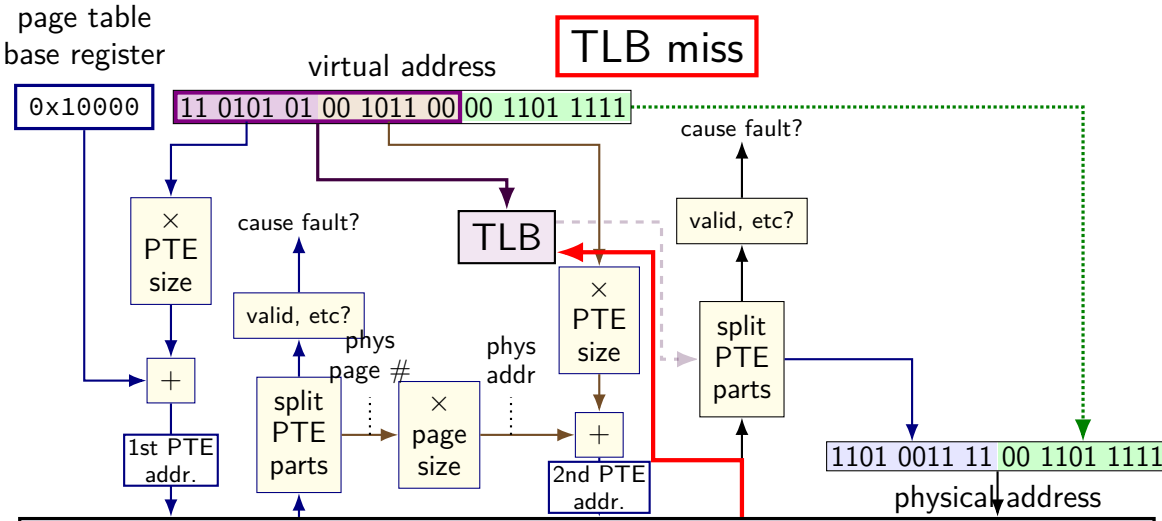
doesn't matter which last-level page table

means TLB output can be used directly to form address

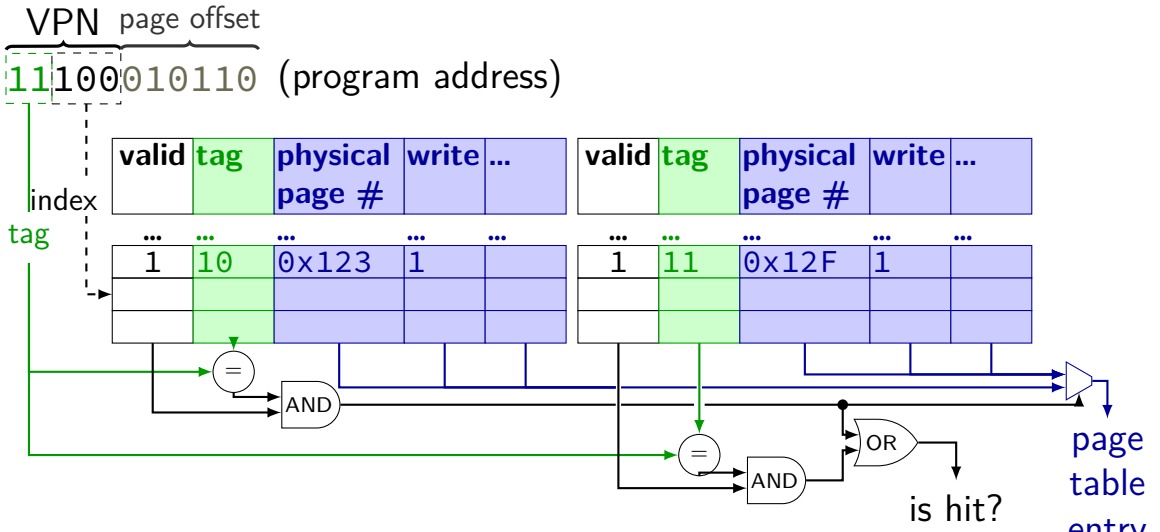
TLB and two-level lookup



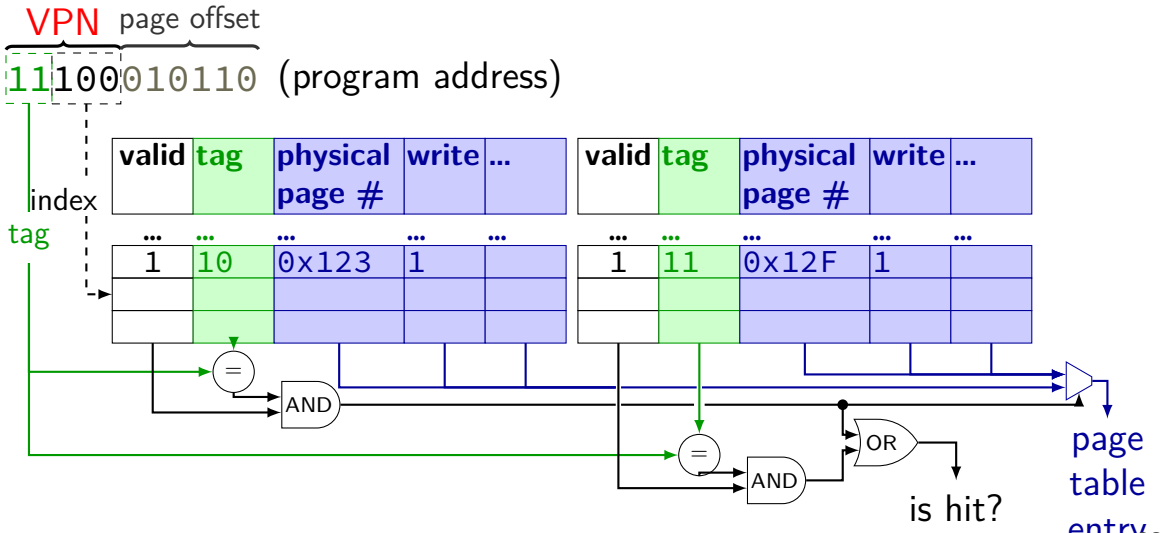
TLB and two-level lookup



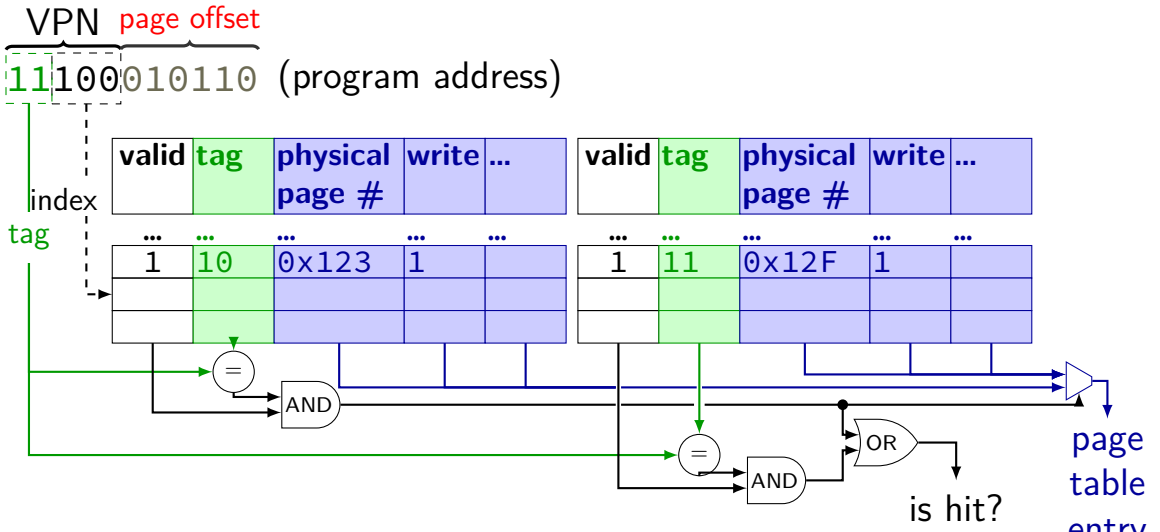
TLB organization (2-way set associative)



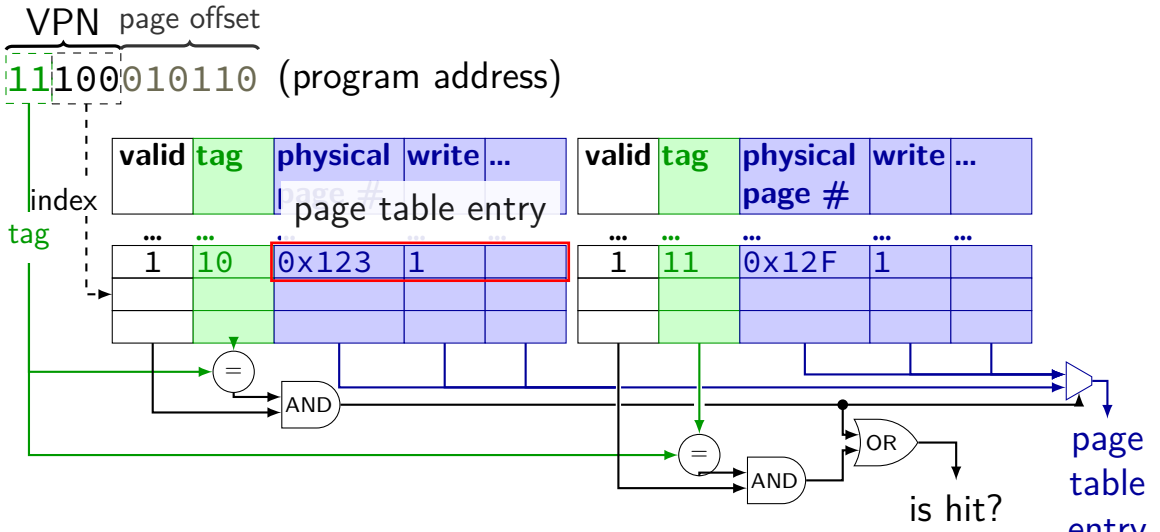
TLB organization (2-way set associative)



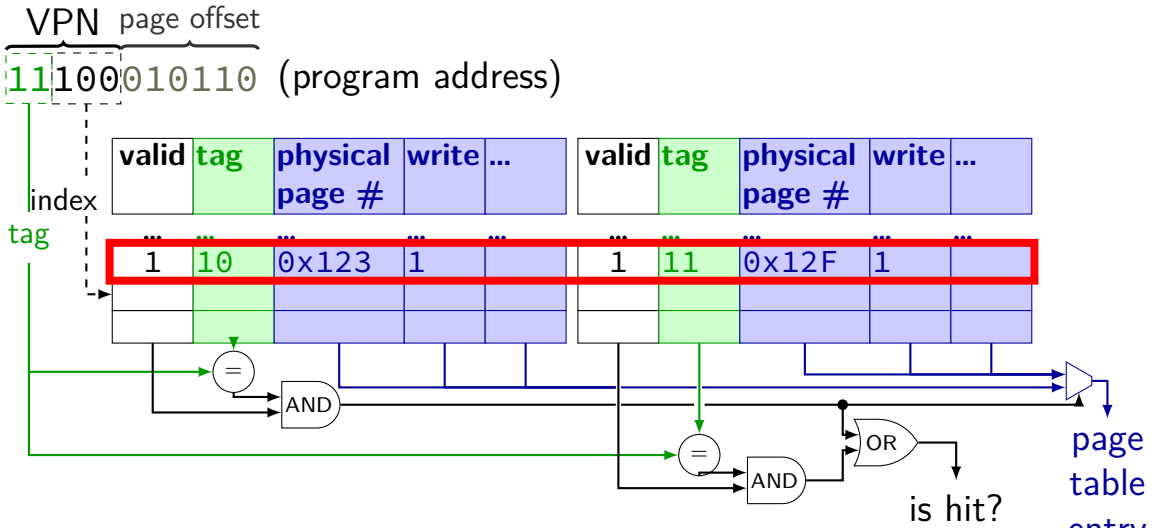
TLB organization (2-way set associative)



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 (1)

my desktop:

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

64-entry, 4-way L1 data TLB

TLB index bits?

$64/4 = 16$ sets — 4 bits

TLB tag bits?

$48 - 12 = 36$ bit virtual page number — $36 - 4 = 32$ bit TLB tag

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?

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?

$1536/12 = 128$ sets — 7 bits

TLB tag bits?

$48 - 12 = 36$ bit virtual page number — $36 - 7 = 29$ bit TLB tag

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	0x310FE8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

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	0x310FE8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

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	0x310FE8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

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	0x310FE8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

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	0x310FE8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

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	0x310FE8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

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	0x310FE8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

TLB access pattern example

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

set idx	V	tag	physical page (page table entry)	write? (page table entry)	user? (page table entry)	···
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	0x310FE8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

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	0x310FE8	
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

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	0x310FE8	hit
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

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	0x310FE8	hit
0x30000	0011 0000	0x8FF000	
0x7CFE0	0111 1100	0x310FE0	

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	0011000	0x8FF	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	0x310FE8	hit
0x30000	0011 0000	0x8FF000	miss
0x7CFE0	0111 1100	0x310FE0	

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	0011000	0x8FF	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	0x310FE8	hit
0x30000	0011 0000	0x8FF000	miss
0x7CFE0	0111 1100	0x310FE0	

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	0x310FE8	hit
0x30000	0011 0000	0x8FF000	miss
0x7CFE0	0111 1100	0x310FE0	miss

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

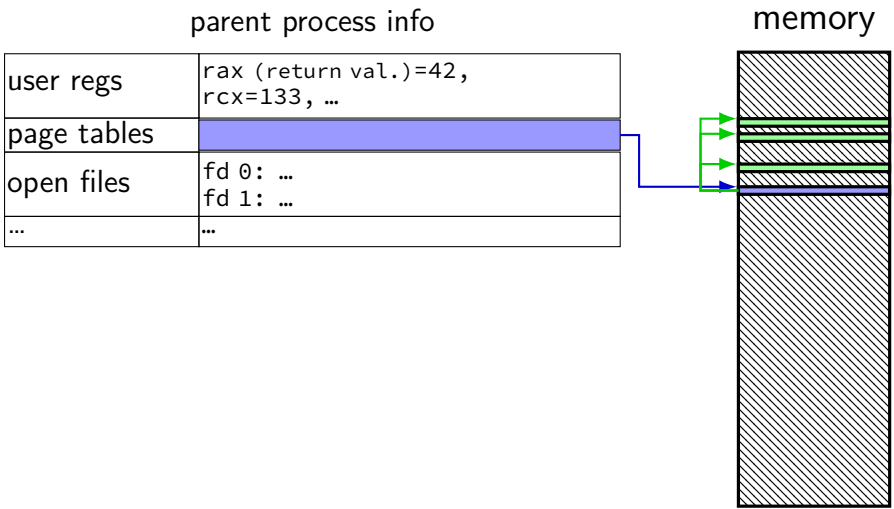
returns twice:

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

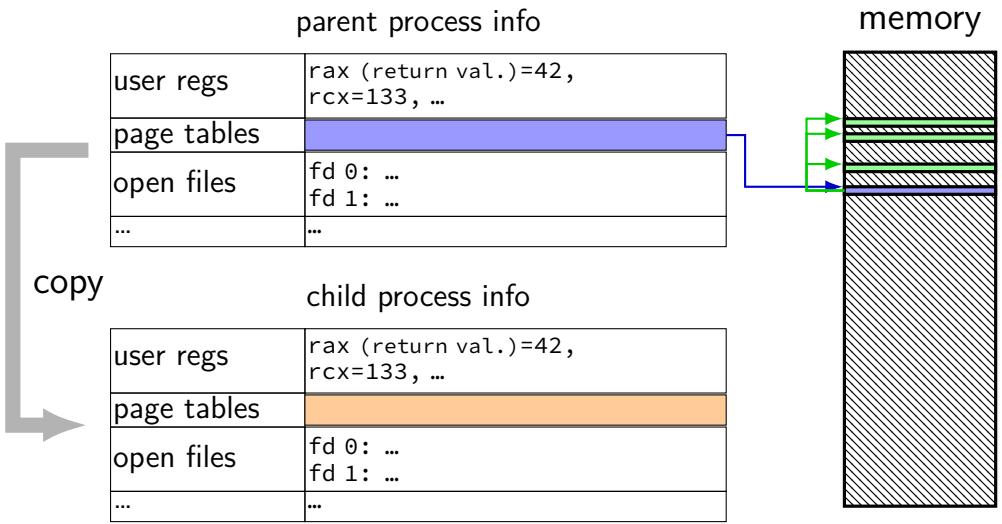
everything (but pid) duplicated in parent, child:

- memory
- file descriptors (later)
- registers

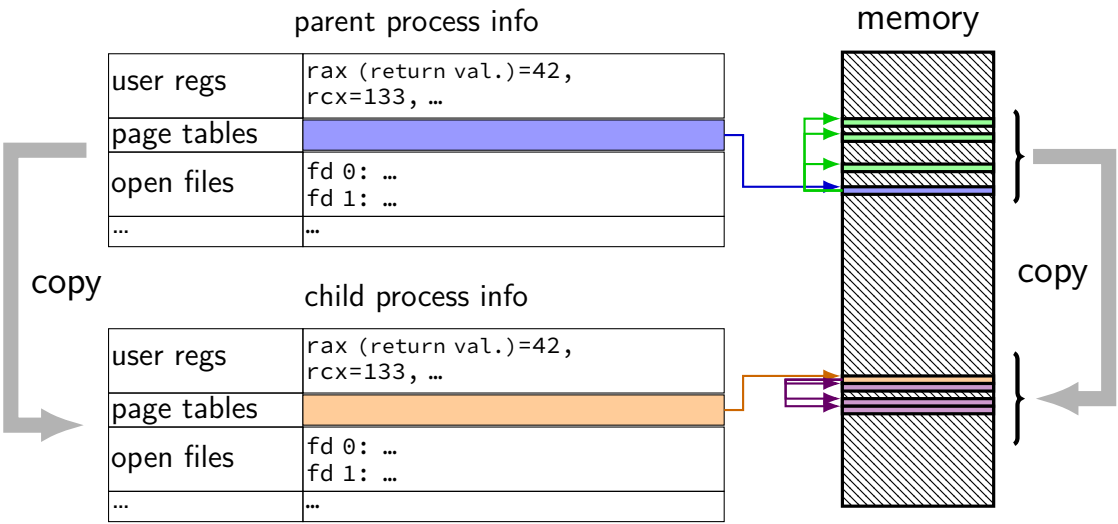
fork and process info (w/o copy-on-write)



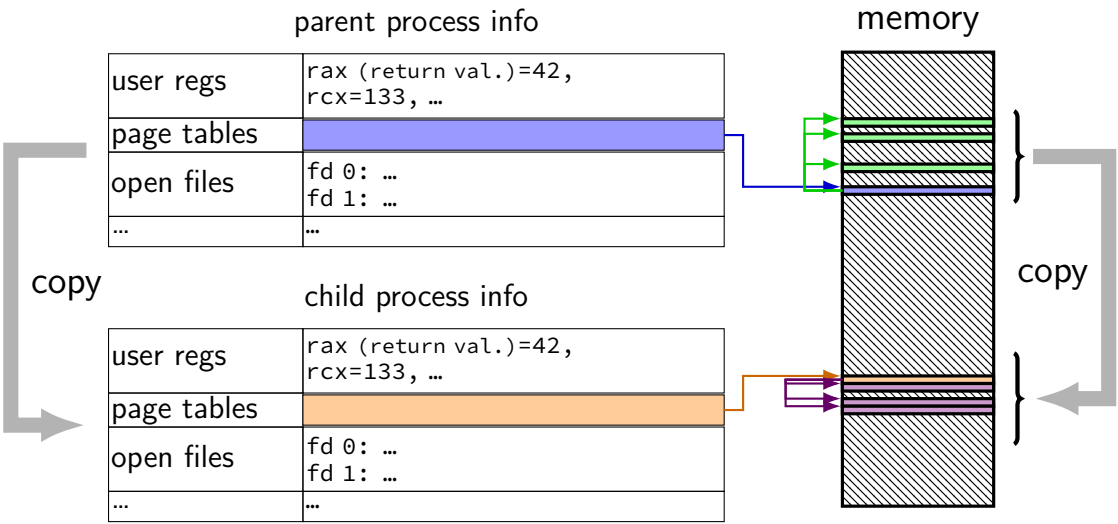
fork and process info (w/o copy-on-write)



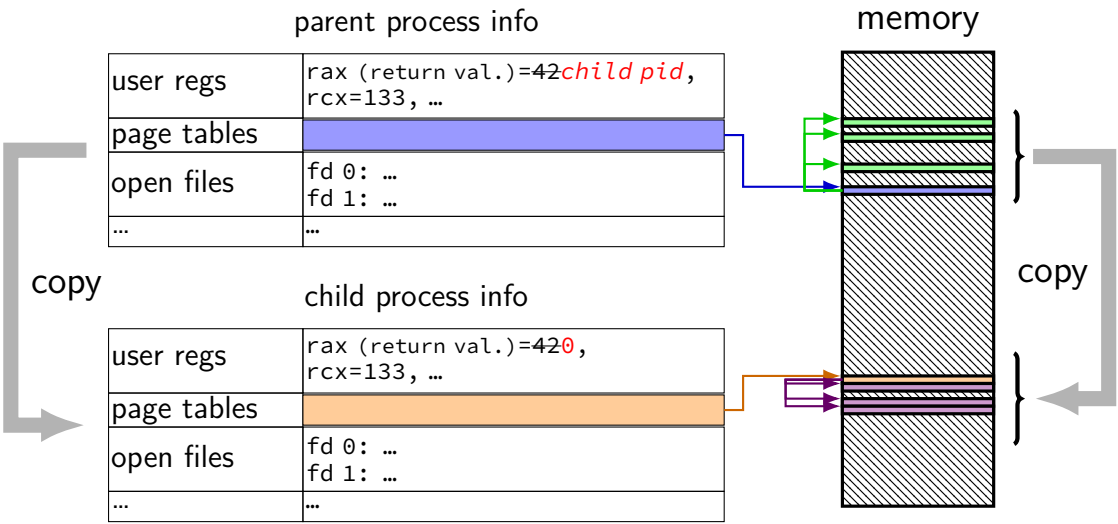
fork and process info (w/o copy-on-write)



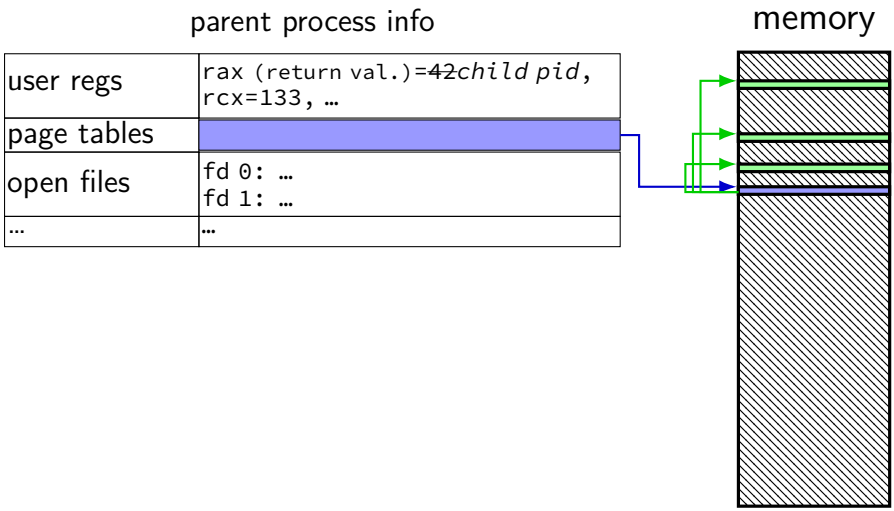
fork and process info (w/o copy-on-write)



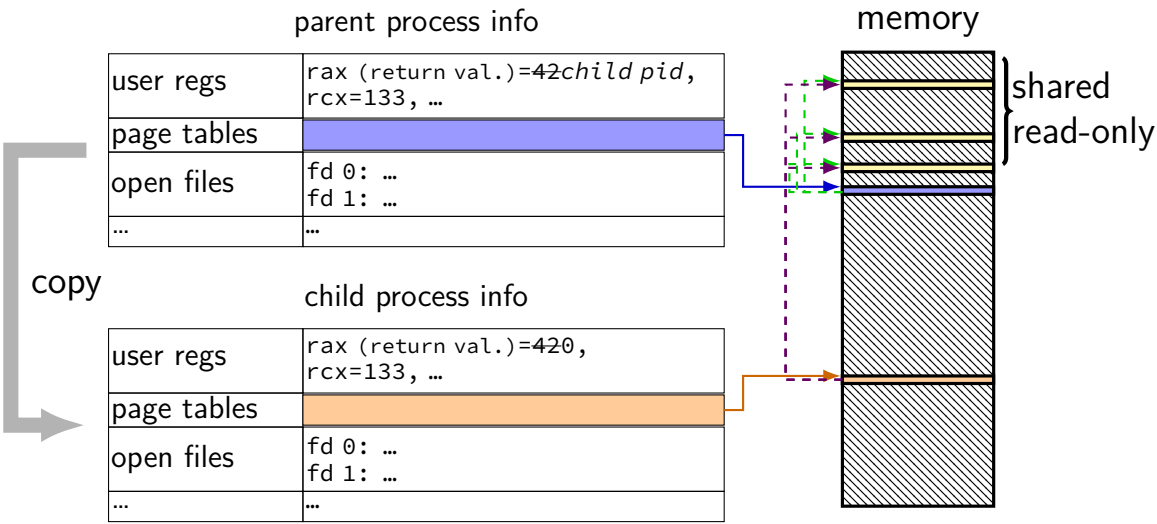
fork and process info (w/o copy-on-write)



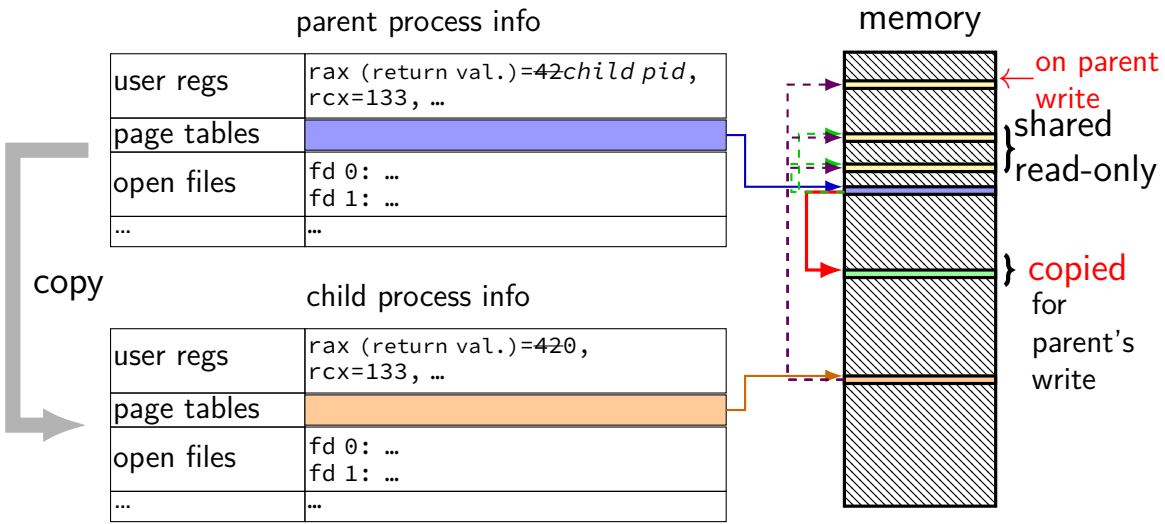
fork (w/ copy-on-write, if parent writes first)



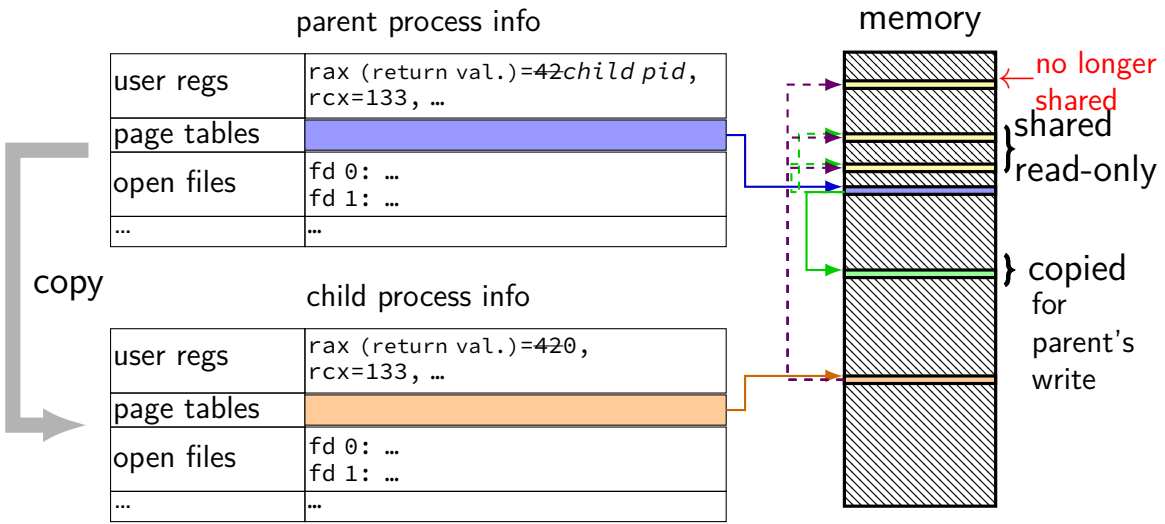
fork (w/ copy-on-write, if parent writes first)



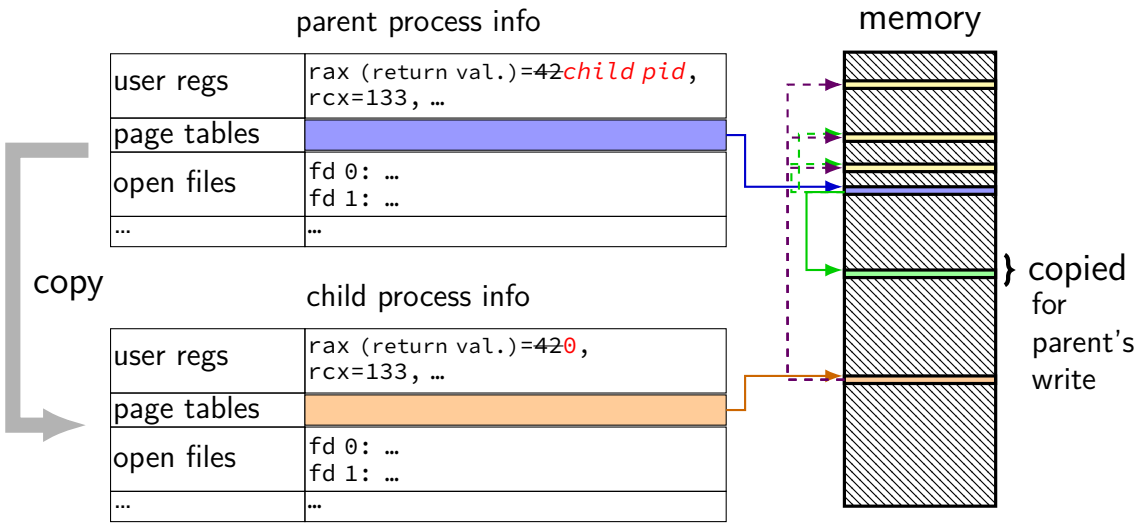
fork (w/ copy-on-write, if parent writes first)



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.)

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

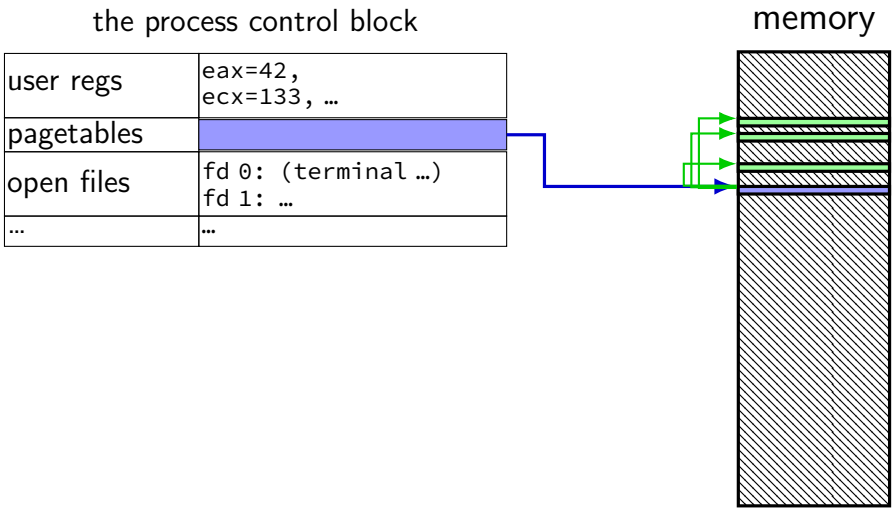
execv example

```
...
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')

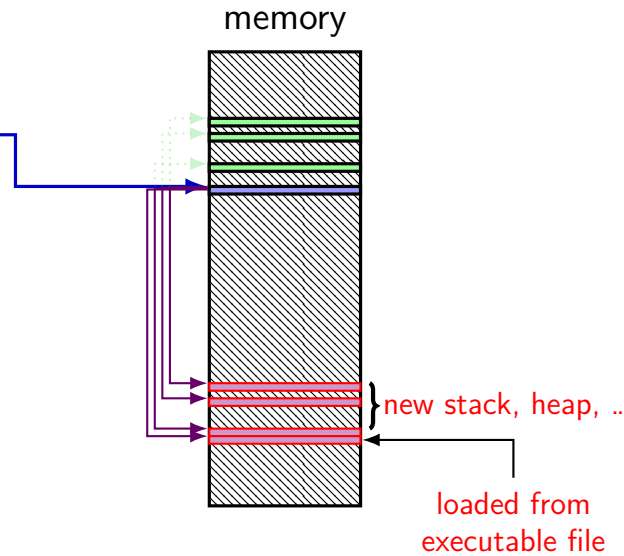
exec in the kernel



exec in the kernel

the process control block

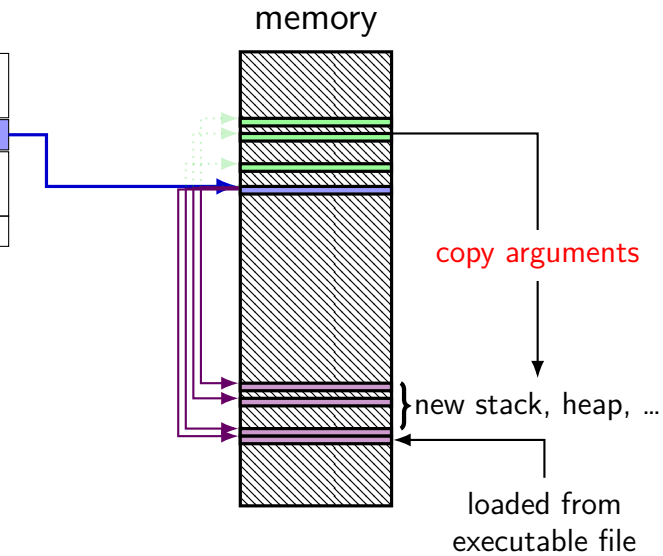
user regs	eax=~~42~~*init. val.*, ecx=~~133~~*init. val.*, …
pagetables	
open files	fd 0: (terminal …) fd 1: …
…	…



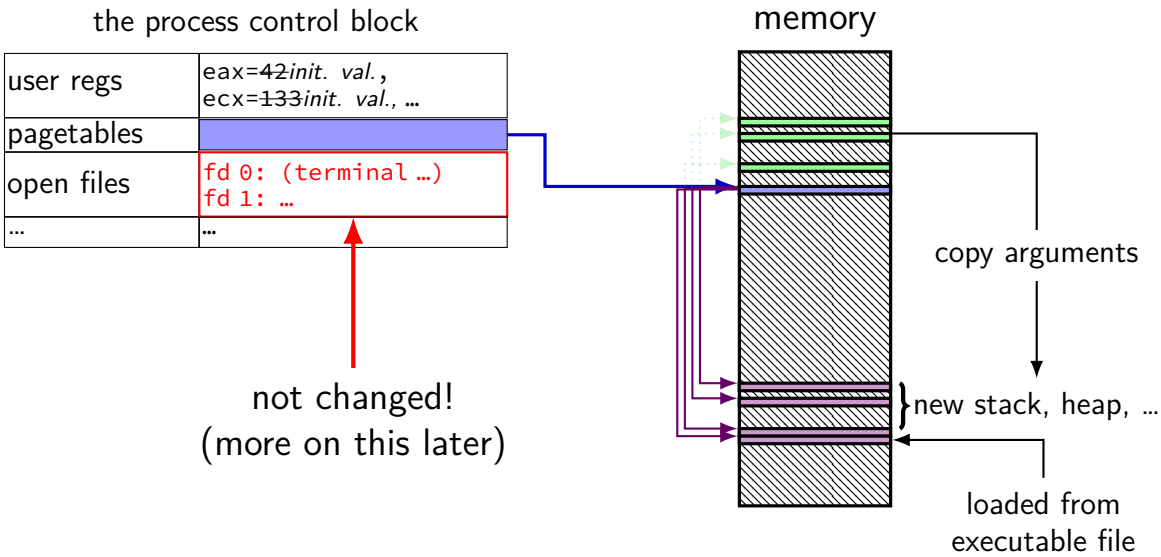
exec in the kernel

the process control block

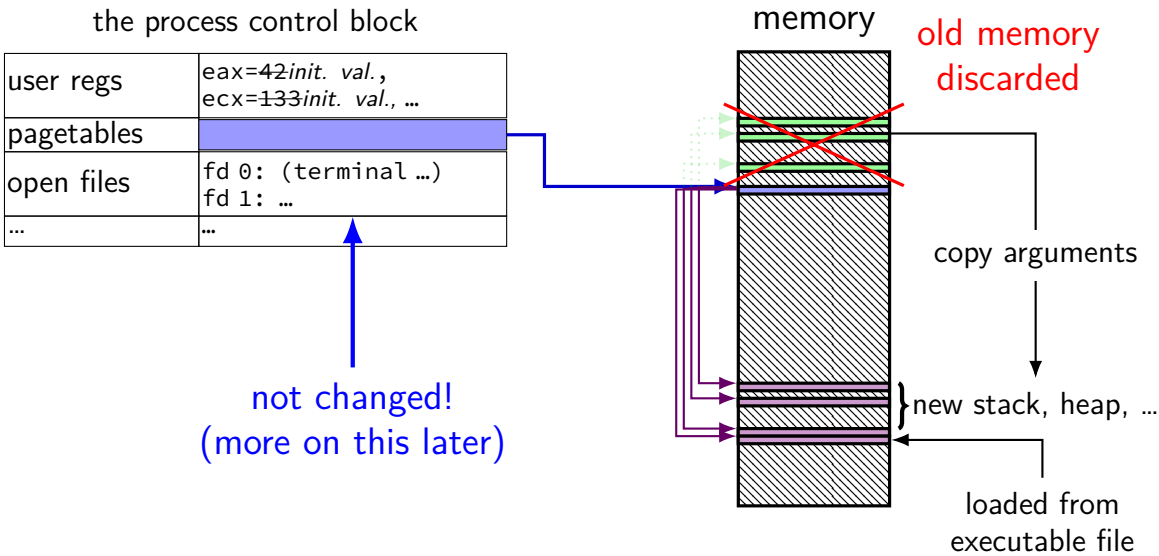
user regs	eax=~~42~~*init. val.*, ecx=~~133~~*init. val.*, …
pagetables	
open files	fd 0: (terminal …) fd 1: …
…	…



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

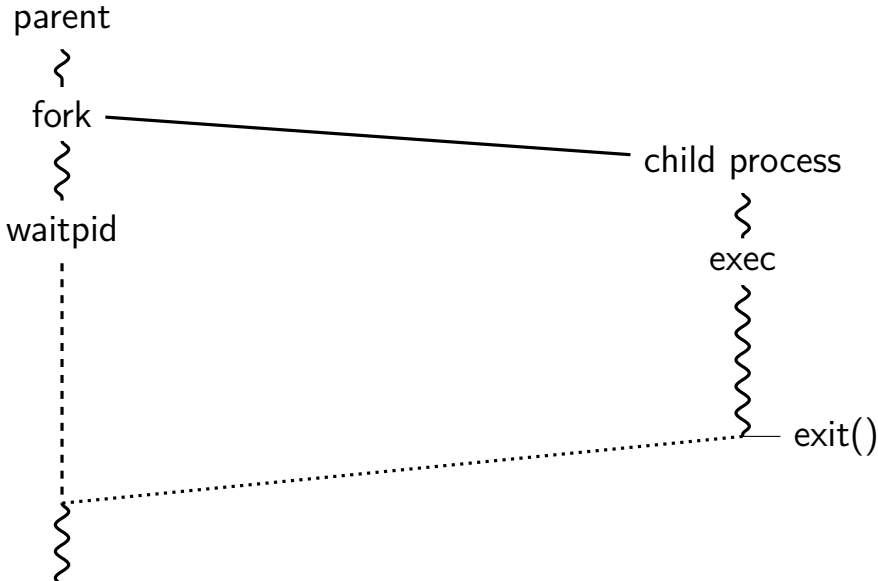
they are also how abnormal termination happens

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

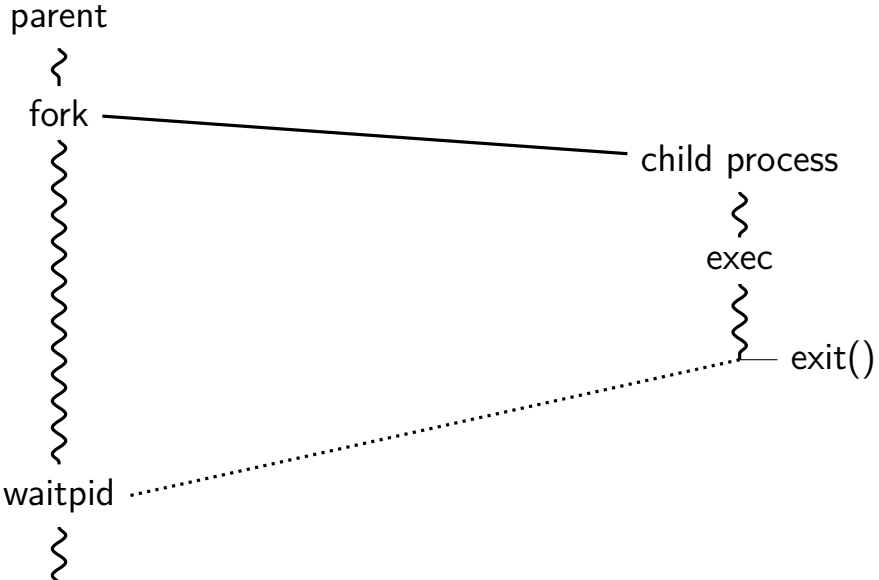
wait's status will tell you when and what signal killed a program

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

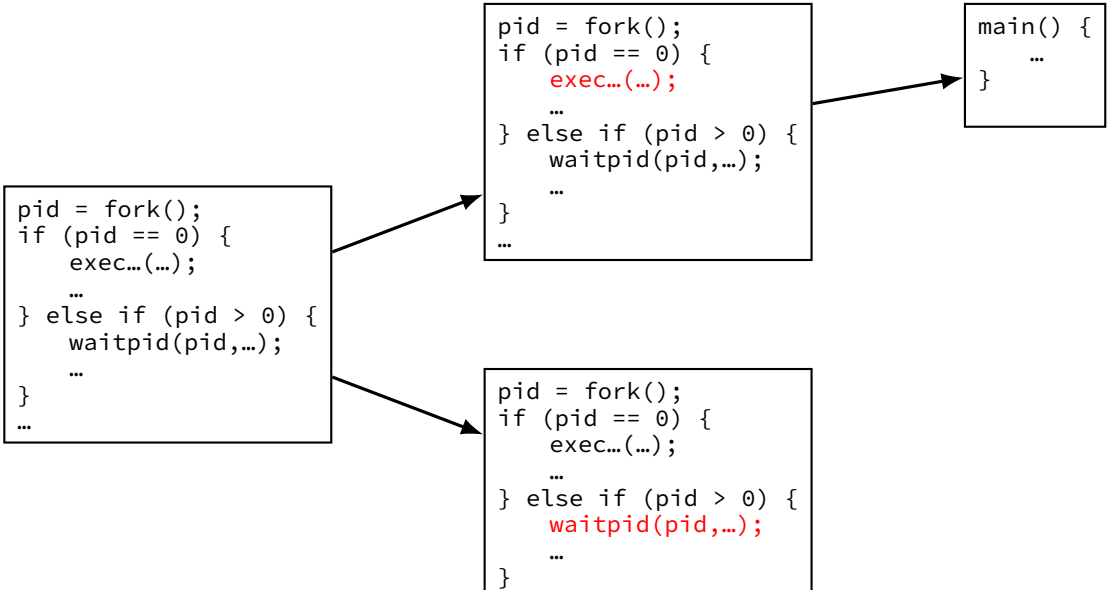
typical pattern



typical pattern (alt)



typical pattern (detail)



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?

exercise: TLB access pattern

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

4096 byte pages

				VPNs of PTEs held in TLB	
type	virtual	physical	result	set 0	set 1
read	0x440030	0x554030	miss	0x440	
write	0x440034	0x554034	hit	0x440	
read	0x7FFFE008	0x556008	miss	0x440	
read	0x7FFFE000	0x556000	hit	0x440, 0x7FFFE	
read	0x7FFFDFF8	0x5F8FF8	miss	0x440, 0x7FFFE	0x7FFFD
read	0x664080	0x5F9080	miss	0x664, 0x7FFFE	0x7FFFD
read	0x440038	0x554038	miss	0x664, 0x440	0x7FFFD
write	0x7FFFDFF0	0x5F8FF0	hit	0x664, 0x440	0x7FFFD

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

exercise: TLB access pattern

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

4096 byte pages

set idx	V	tag	physical page	write?	user?	...	LRU?
0	1	0x00220 (0x440 ≫ 1)	0x554	1	1	...	no
	1	0x00332 (0x00664 ≫ 1)	0x5F9	1	1	...	yes
1	1	0x3FFFF (0x7FFFD ≫ 1)	0x5F8	1	1	...	no
	0	---	---	–	–	...	yes

type					
read	0x440038	0x554038	miss	0x664, 0x440	0x7FFFD
write	0x7FFFDFF0	0x5F8FF0	hit	0x664, 0x440	0x7FFFD

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

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
```

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

...

backup slides