

context switches/POSIX API

Changelog

Changes made in this version not seen in first lecture:

24 Jan 2019: wait: return value for WNOHANG when process not done is 0, not -1

last time

system call flow in xv6

wrapper function $n \rightarrow$ 'gate' in table \rightarrow trap() function \rightarrow ...

other exceptions in xv6

context switching (start)

xv6: two parts — kernel thread to thread, user address space

context stored on kernel stacks

swtch: kernel thread to thread part

homework steps

system call implementation: `sys_writecount`

hint in writeup: imitate `sys_uptime`

need a counter for number of writes

add `writecount` to several tables/lists

(list of handlers, list of library functions to create, etc.)

recommendation: imitate how other system calls are listed

create a userspace program that calls `writecount`

recommendation: copy from given programs

note on locks

some existing code uses acquire/release

you do not have to do this

only for multiprocessor support

...but, copying what's done for ticks would be correct

context switch in xv6

xv6 context switch has two parts

switching threads

switching user address spaces + kernel stack to use for exception

context switch in xv6

xv6 context switch has two parts

switching threads

switching **user address spaces** + kernel stack to use for exception

context switch in xv6

xv6 context switch has two parts

switching threads

switching user address spaces + kernel stack to use for exception

thread switching

```
struct context {  
    uint edi;  
    uint esi;  
    uint ebx;  
    uint ebp;  
    uint eip;  
}
```

```
void swtch(struct context **old, struct context *new);
```

thread switching

structure to save context in
yes, it looks like we're missing
some registers we need...

```
struct context {  
    uint edi;  
    uint esi;  
    uint ebx;  
    uint ebp;  
    uint eip;  
}
```

```
void swtch(struct context **old, struct context *new);
```

thread switching

eip = saved program counter

```
struct context {  
    uint edi;  
    uint esi;  
    uint ebx;  
    uint ebp;  
    uint eip;  
}
```

```
void swtch(struct context **old, struct context *new);
```

thread switching

```
struct context {  
    uint edi;  
    uint esi;  
    uint ebx;  
    uint ebp;  
    uint eip;  
}
```

function to switch contexts
allocate space for context on top of stack
set old to point to it
switch to context new

```
void swtch(struct context **old, struct context *new);
```

thread switching in xv6: C

in thread A:

```
/* switch from A to B */  
  
... // (1)  
switch(&(a->context), b->context); /* returns to (2) */  
... // (4)
```

in thread B:

```
switch(...); // (0) -- called earlier  
... // (2)  
...  
/* later on switch back to A */  
... // (3)  
switch(&(b->context), a->context) /* returns to (4) */  
...
```

thread switching in xv6: C

in thread A:

```
/* switch from A to B */
```

```
... // (1)  
switch(&(a->context), b->context); /* returns to (2) */  
... // (4)
```

in thread B:

```
switch(...); /* (0) -- called earlier  
... // (2)  
...  
/* later on switch back to A */  
... // (3)  
switch(&(b->context), a->context) /* returns to (4) */  
...
```

thread switching in xv6: C

in thread A:

```
/* switch from A to B */
```

```
... // (1)
```

```
switch(&(a->context), b->context); /* returns to (2) */
```

```
... // (4)
```

in thread B:

```
switch(...); // (0) -- called earlier
```

```
... // (2)
```

```
...
```

```
/* later on switch back to A */
```

```
... // (3)
```

```
switch(&(b->context), a->context) /* returns to (4) */
```

```
...
```

thread switching in xv6: C

in thread A:

```
/* switch from A to B */
```

```
... // (1)
```

```
switch(&(a->context), b->context); /* returns to (2) */
```

```
... // (4)
```

in thread B:

```
switch(...); // (0) -- called earlier
```

```
... // (2)
```

```
...
```

```
/* later on switch back to A */
```

```
... // (3)
```

```
switch(&(b->context), a->context) /* returns to (4) */
```

```
...
```


thread switching in xv6: C

in thread A:

```
/* switch from A to B */
```

```
... // (1)
```

```
switch(&(a->context), b->context); /* returns to (2) */
```

```
... // (4)
```

in thread B:

```
switch(...); // (0) -- called earlier
```

```
... // (2)
```

```
...
```

```
/* later on switch back to A */
```

```
... // (3)
```

```
switch(&(b->context), a->context) /* returns to (4) */
```

```
...
```

thread switching in xv6: C

in thread A:

```
/* switch from A to B */
```

```
... // (1)
```

```
switch(&(a->context), b->context); /* returns to (2) */
```

```
... // (4)
```

in thread B:

```
switch(...); // (0) -- called earlier
```

```
... // (2)
```

```
...
```

```
/* later on switch back to A */
```

```
... // (3)
```

```
switch(&(b->context), a->context) /* returns to (4) */
```

```
...
```

thread switching in xv6: assembly

```
.globl swtch
```

```
swtch:
```

```
    movl 4(%esp), %eax
```

```
    movl 8(%esp), %edx
```

```
    # Save old callee-save registers
```

```
    pushl %ebp
```

```
    pushl %ebx
```

```
    pushl %esi
```

```
    pushl %edi
```

```
    # Switch stacks
```

```
    movl %esp, (%eax)
```

```
    movl %edx, %esp
```

```
    # Load new callee-save registers
```

```
    popl %edi
```

```
    popl %esi
```

```
    popl %ebx
```

```
    popl %ebp
```

```
    ret
```

thread switching in xv6: assembly

```
.globl swtch
```

```
swtch:
```

```
movl 4(%esp), %eax
```

```
movl 8(%esp), %edx
```

```
# Save old callee-save registers
```

```
pushl %ebp
```

```
pushl %ebx
```

```
pushl %esi
```

```
pushl %edi
```

```
# Switch stacks
```

```
movl %esp, (%eax)
```

```
movl %edx, %esp
```

```
# Load new callee-save registers
```

```
popl %edi
```

```
popl %esi
```

```
popl %ebx
```

```
popl %ebp
```

```
ret
```

two arguments:

```
struct context **from_context
```

= where to save current context

```
struct context *to_context
```

= where to find new context

context stored on thread's stack

context address = top of stack

thread switching in xv6: assembly

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx
```

saved: ebp, ebx, esi, edi

Save old callee-save registers

```
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi
```

Switch stacks

```
movl %esp, (%eax)
movl %edx, %esp
```

Load new callee-save registers

```
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

thread switching in xv6: assembly

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx

    # Save old callee-save registers
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi

    # Switch stacks
    movl %esp, (%eax)
    movl %edx, %esp

    # Load new callee-save registers
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

what about other parts of context?
eax, ecx, ...: saved by swtch's caller
esp: same as address of context
program counter: set by call of swtch

thread switching in xv6: assembly

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx

    # Save old callee-save registers
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi

    # Switch stacks
    movl %esp, (%eax)
    movl %edx, %esp

    # Load new callee-save registers
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

save stack pointer to first argument
(stack pointer now has all info)
restore stack pointer from second argument

thread switching in xv6: assembly

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx
```

Save old callee-save registers

```
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi
```

Switch stacks

```
    movl %esp, (%eax)
    movl %edx, %esp
```

Load new callee-save registers

```
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

restore program counter
(and other saved registers)
from new context

juggling stacks

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx
```

Save old callee-save registers

```
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi
```

Switch stacks

```
    movl %esp, (%eax)
    movl %edx, %esp
```

Load new callee-save registers

```
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

from stack

caller-saved registers
swtch arguments
swtch return addr.

to stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

juggling stacks

```
.globl swtch  
swtch:
```

```
    movl 4(%esp), %eax  
    movl 8(%esp), %edx  
    %esp →
```

Save old callee-save registers

```
    pushl %ebp  
    pushl %ebx  
    pushl %esi  
    pushl %edi
```

Switch stacks

```
    movl %esp, (%eax)  
    movl %edx, %esp
```

Load new callee-save registers

```
    popl %edi  
    popl %esi  
    popl %ebx  
    popl %ebp  
    ret
```

from stack

caller-saved registers
swtch arguments
swtch return addr.

to stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

juggling stacks

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx
```

Save old callee-save reg

```
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi
```

%esp →

from stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

to stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

Switch stacks

```
    movl %esp, (%eax)
    movl %edx, %esp
```

Load new callee-save registers

```
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

juggling stacks

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx
```

Save old callee-save reg

```
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi
```

Switch stacks

```
    movl %esp, (%eax)
    movl %edx, %esp
```

Load new callee-save registers

```
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

from stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

to stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

← %esp

juggling stacks

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx
```

Save old callee-save reg

```
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi
```

Switch stacks

```
    movl %esp, (%eax)
    movl %edx, %esp
```

Load new callee-save registers

```
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

from stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

to stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

← %esp

juggling stacks

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx
```

Save old callee-save reg

```
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi
```

Switch stacks

```
    movl %esp, (%eax)
    movl %edx, %esp
```

Load new callee-save registers

```
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

from stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

to stack

caller-saved registers
swtch arguments
swtch return addr.

← %esp

first call to swtch?

one thread calls swtch and

...return from another thread's call to swtch

what about switching to a **new thread**?

creating a new thread

```
static struct proc*
allocproc(void)
{
    ...
    sp = p->kstack + KSTACKSIZE;

    // Leave room for trap frame.
    sp -= sizeof *p->tf;
    p->tf = (struct trapframe*)sp;

    // Set up new context to start executing at forkret,
    // which returns to trapret.
    sp -= 4;
    *(uint*)sp = (uint)trapret;

    sp -= sizeof *p->context;
    p->context = (struct context*)sp;
    memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;
    ...
}
```

struct proc \approx process
p is new struct proc
p->kstack is its new stack
(for the kernel only)

creating a new thread

new kernel stack

```
static struct proc*
allocproc(void)
{
    ...
    sp = p->kstack + KSTACKSIZE;

    // Leave room for trap frame.
    sp -= sizeof *p->tf;
    p->tf = (struct trapframe*)sp;

    // Set up new context to start executing at forkret,
    // which returns to trapret.
    sp -= 4;
    *(uint*)sp = (uint)trapret;

    sp -= sizeof *p->context;
    p->context = (struct context*)sp;
    memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;
    ...
}
```



creating a new thread

```
static struct proc*
allocproc(void)
{
    ...
    sp = p->kstack + KSTACKSIZE;

    // Leave room for trap frame.
    sp -= sizeof *p->tf;
    p->tf = (struct trapframe*)sp;

    // Set up new context to start executing at forkret,
    // which returns to trapret.
    sp -= 4;
    *(uint*)sp = (uint)trapret;

    sp -= sizeof *p->context;
    p->context = (struct context*)sp;
    memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;
    ...
}
```

new kernel stack

'trapframe'
(saved userspace registers
as if there was an interrupt)



creating a new thread

```
static struct proc*  
allocproc(void)  
{
```

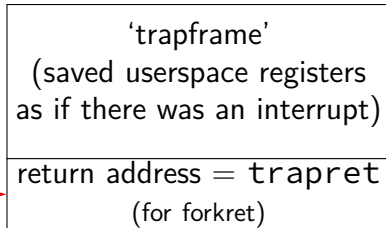
```
    ...  
    sp = p->kstack + KSTACKSIZE;  
assembly code to return to user mode  
same code as for syscall returns  
    p->tf = (struct trapframe*)sp;
```

```
// Set up new context to start executing at forkret,  
// which returns to trapret.
```

```
    sp -= 4;  
    *(uint*)sp = (uint)trapret;
```

```
    sp -= sizeof *p->context;  
    p->context = (struct context*)sp;  
    memset(p->context, 0, sizeof *p->context);  
    p->context->eip = (uint)forkret;  
    ...
```

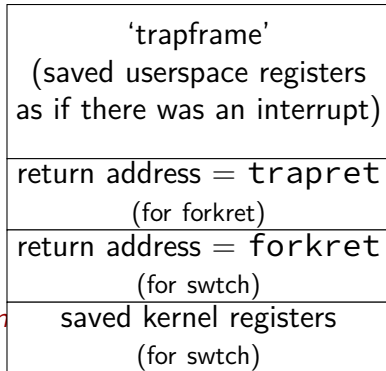
new kernel stack



creating a new thread

```
static struct proc*  
allocproc(void)  
{  
    ...  
    sp = p->kstack + KSTACKSIZE;  
  
    // Leave room for trap frame.  
    sp -= sizeof *p->tf;  
    p->tf = (struct trapframe*)sp;  
  
    // Set up new context to start executing  
    // which returns to trapret.  
    sp -= 4;  
    *(uint*)sp = (uint)trapret;  
  
    sp -= sizeof *p->context;  
    p->context = (struct context*)sp;  
    memset(p->context, 0, sizeof *p->context);  
    p->context->eip = (uint)forkret;  
    ...  
}
```

new kernel stack



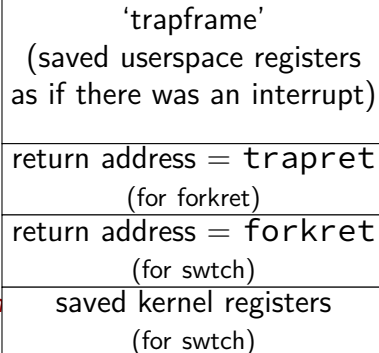
creating a new thread

```
static struct proc*
allocproc(void)
{
    ...
    sp = new stack says: this thread is
    // in middle of calling switch
    // in the middle of a system call
    p->context = (struct trapframe*)sp;

    // Set up new context to start executing
    // which returns to trapret.
    sp -= 4;
    *(uint*)sp = (uint)trapret;

    sp -= sizeof *p->context;
    p->context = (struct context*)sp;
    memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;
    ...
}
```

new kernel stack



juggling stacks

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx
```

Save old callee-save reg

```
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi
```

Switch stacks

```
    movl %esp, (%eax)
    movl %edx, %esp
```

Load new callee-save registers

```
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

from stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

to stack

caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

← %esp



bottom of
new kernel stack

first instruction
executed by new thread

kernel-space context switch summary

swtch function

- saves registers on current kernel stack

- switches to new kernel stack and restores its registers

initial setup — manually construct stack values

juggling stacks

```
.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx
```

Save old callee-save reg

```
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi
```

Switch stacks

```
    movl %esp, (%eax)
    movl %edx, %esp
```

Load new callee-save registers

```
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

from stack

saved user regs
...
caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

to stack

saved user regs
...
caller-saved registers
swtch arguments
swtch return addr.
saved ebp
saved ebx
saved esi
saved edi

the userspace part?

user registers stored in 'trapframe' struct

- created on kernel stack when interrupt/trap happens
- restored before using `iret` to switch to user mode

initial user registers created manually on stack

- (as if saved by system call)

the userspace part?

user registers stored in 'trapframe' struct

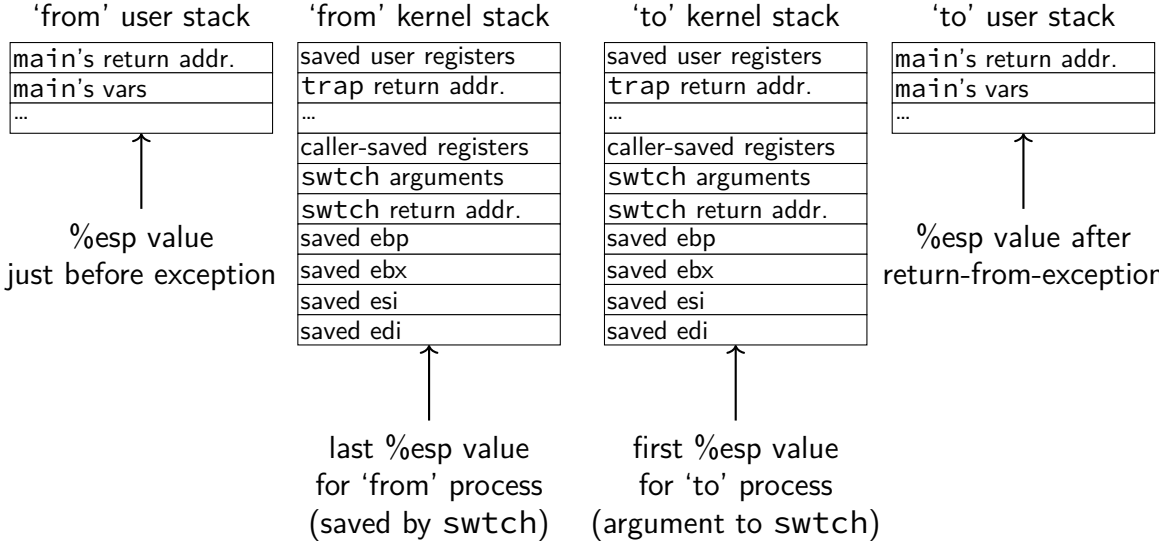
- created on kernel stack when interrupt/trap happens
- restored before using `iret` to switch to user mode

initial user registers created manually on stack

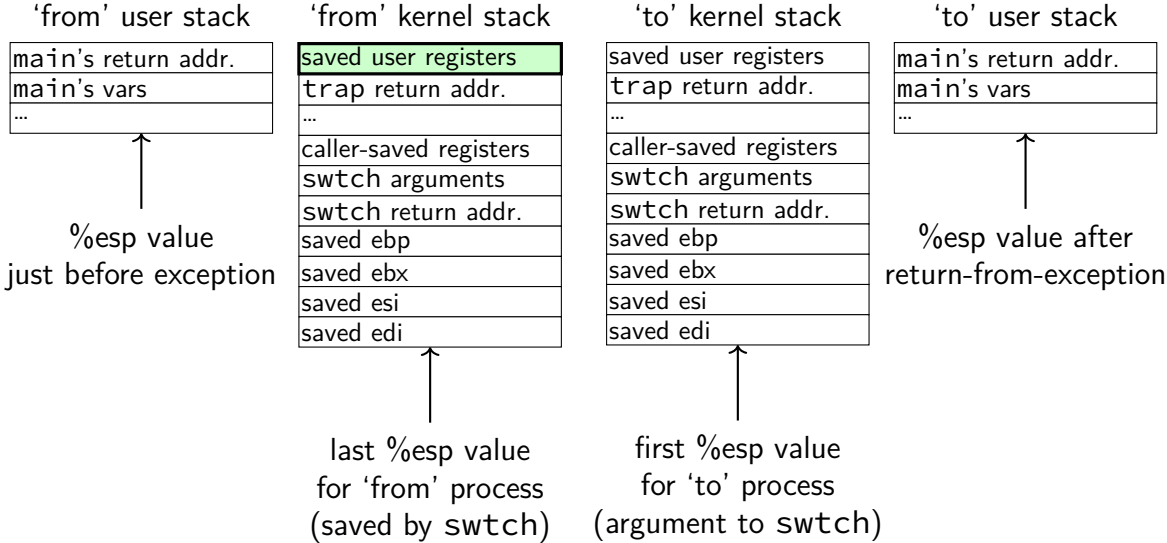
- (as if saved by system call)

other code (not shown) handles setting address space

where things go in context switch



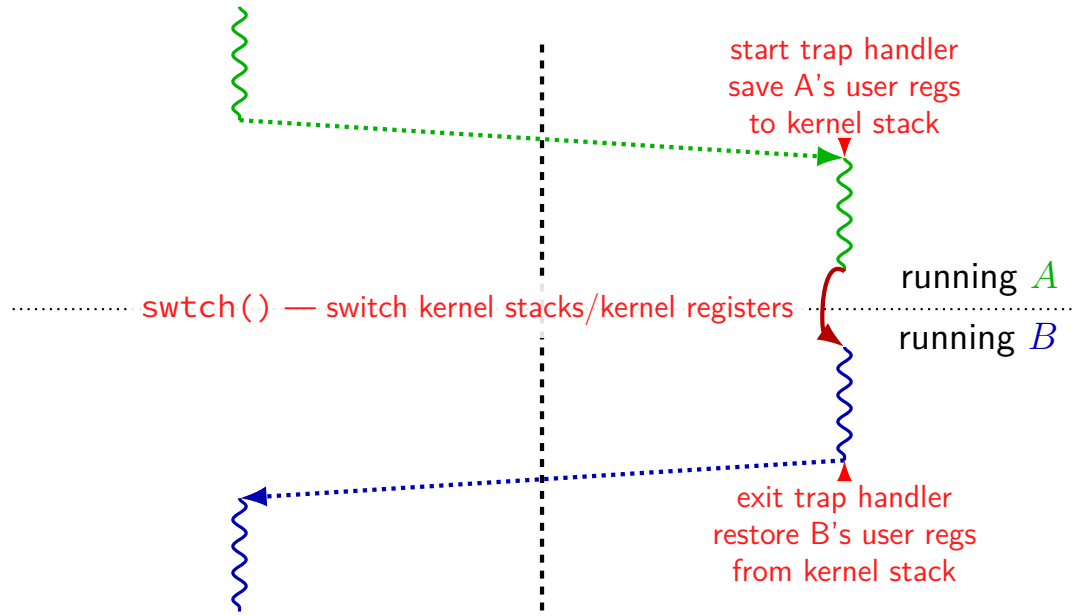
where things go in context switch



xv6 context switch and saving

user mode

kernel mode



exceptions in exceptions

```
alltraps:
    ...
    pushl %esp
    call trap
        /* in trap(): */
        movl ..., %eax

    ...
    ret
```

current kernel stack


eax from trap() 1st time
ecx from trap() 1st time
...

exceptions in exceptions

```
alltraps:
  ...
  pushl %esp
  call trap
  /* in trap(): */
  movl ..., %eax
  ...
  ret
```

current kernel stack

eax from trap() 1st time
ecx from trap() 1st time
...
eax from trap() 2nd time
ecx from trap() 2nd time
...



```
alltraps:
  ...
  pushl %esp
  call trap
```

exceptions in exceptions

```
alltraps:  
...  
pushl %esp  
call trap  
    /* in trap(): */  
    movl ..., %eax  
  
...  
ret
```

current kernel stack

eax from trap() 1st time
ecx from trap() 1st time
...
eax from trap() 2nd time
ecx from trap() 2nd time
...

x86 rule: switch stacks when switching user to kernel mode (same stack pointer if already in kernel mode)

~~alltraps:
...
pushl %esp
call trap~~

exceptions in exceptions

```
alltraps:  
...  
pushl %esp  
call trap  
    /* in trap(): */  
    movl ..., %eax  
  
...  
ret
```

current kernel stack

eax from trap() 1st time
ecx from trap() 1st time
...
eax from trap() 2nd time
ecx from trap() 2nd time
...

how much stack space??
need to know how deep
nesting might be

~~alltraps:
...
pushl %esp
call trap~~

interrupt disabling

CPU supports **disabling** (most) interrupts

interrupts will **wait** until it is reenabled

CPU has extra state: are interrupts enabled?

xv6 interrupt disabling

xv6 policy: interrupts disabled in non-syscall traps

xv6 interrupt disabling

xv6 policy: interrupts disabled in non-syscall traps

this policy makes xv6 easier to code...

disadvantages?

xv6 interrupt disabling

xv6 policy: interrupts disabled in non-syscall traps

this policy makes xv6 easier to code...

disadvantages?

- slow kernel code makes system hang?

- gaurenteeing minimum reaction time to keypress?

exercise

suppose xv6 is running this `loop.exe`:

```
main:
    mov $0, %eax    // eax ← 0
start_loop:
    add $1, %eax    // eax ← eax + 1
    jmp start_loop  // goto start_loop
```

when xv6 switches away from this program, where is the value of `loop.exe`'s `eax` stored?

- A. `loop.exe`'s user stack
- B. `loop.exe`'s kernel stack
- C. the user stack of the program switched to
- D. the kernel stack for the program switched to
- E. `loop.exe`'s heap
- F. a special register
- G. elsewhere

interrupt descriptor table

x86's interrupt descriptor table has an entry for each kind of exception

- segmentation fault

- timer expired (“your program ran too long”)

- divide-by-zero

- system calls

- ...

xv6 sets all the table entries

...and they always call the `trap()` function

process control block

some data structure needed to represent a process

called **Process Control Block**

process control block

some data structure needed to represent a process

called **Process Control Block**

xv6: `struct proc`

xv6: struct proc

```
struct proc {  
    uint sz; // Size of process memory (bytes)  
    pde_t* pgdir; // Page table  
    char *kstack; // Bottom of kernel stack for this process  
    enum procstate state; // Process state  
    int pid; // Process ID  
    struct proc *parent; // Parent process  
    struct trapframe *tf; // Trap frame for current syscall  
    struct context *context; // swtch() here to run process  
    void *chan; // If non-zero, sleeping on chan  
    int killed; // If non-zero, have been killed  
    struct file *ofile[NOFILE]; // Open files  
    struct inode *cwd; // Current directory  
    char name[16]; // Process name (debugging)  
};
```

xv6: struct proc

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state ≈ thread's state
  int pid; // Process ID
  struct proc *parent; // Parent process
  struct trapframe *tf; // Trap frame for current syscall
  struct context *context; // swtch() here to run process
  void *chan; // If non-zero, sleeping on chan
  int killed; // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd; // Current directory
  char name[16]; // Process name (debugging)
};
```

current registers/PC of process (user and kernel)
stored on (pointer to) its kernel stack
(if not currently running)

≈ thread's state

ss

xv6: struct proc

```
struct proc {
    uint sz;
    pde_t* pgdir;
    char *kstack;
    enum procstate state;
    int pid;
    struct proc *parent;
    struct trapframe *tf;
    struct context *context;
    void *chan;
    int killed;
    struct file *ofile[NOFILE];
    struct inode *cwd;
    char name[16];
};
```

the kernel stack for this process
every process has one kernel stack

```
// Size of process memory (bytes)
// Page table
// Bottom of kernel stack for this process
// Process state
// Process ID
// Parent process
// Trap frame for current syscall
// swtch() here to run process
// If non-zero, sleeping on chan
// If non-zero, have been killed
// Open files
// Current directory
// Process name (debugging)
```

xv6: struct proc

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

```
enum procstate {
  UNUSED, EMBRYO, SLEEPING,
  RUNNABLE, RUNNING, ZOMBIE
};
```

```
// Process state
// Process ID
// Parent process
// Trap frame for current syscall
// swtch() here to run process
// If non-zero, sleeping on chan
// If non-zero, have been killed
// Open files
// Current directory
// Process name (debugging)
```

is process running?
or waiting?
or finished?
if waiting,
waiting for what (chan)?

xv6: struct proc

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

process ID
to identify process in system calls

```
// Size of process memory (bytes)
// Page table
// Bottom of kernel stack for this process
// Process state
// Process ID
// Parent process
// Trap frame for current syscall
// swtch() here to run process
// If non-zero, sleeping on chan
// If non-zero, have been killed
// Open files
// Current directory
// Process name (debugging)
```

xv6: struct proc

```
struct proc {  
    uint sz; // Size of process memory (bytes)  
    pde_t* pgdir; // Page table  
    char *kstack; // Bottom of kernel stack for this process  
    enum procstate state; // Process state  
    int pid; // Proc  
    struct proc *parent; // Pare  
    struct trapframe *tf; // Trap  
    struct context *context; // swtc  
    void *chan; // If n  
    int killed; // If non-zero, have been killed  
    struct file *ofile[NOFILE]; // Open files  
    struct inode *cwd; // Current directory  
    char name[16]; // Process name (debugging)  
};
```

information about address space
pgdir — used by processor
sz — used by OS only

xv6: struct proc

information about open files, etc.

```
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t* pgdir; // Page table
    char *kstack; // Bottom of kernel stack for this process
    enum procstate state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // swtch() here to run process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16]; // Process name (debugging)
};
```


process control blocks generally

contains process's context(s) (registers, PC, ...)

if context is not on a CPU

(in xv6: pointers to these, actual location: process's kernel stack)

process's status — running, waiting, etc.

information for system calls, etc.

open files

memory allocations

process IDs

related processes

xv6 myproc

xv6 function: `myproc()`

retrieves pointer to currently running struct `proc`

myproc: using a global variable

```
struct cpu cpus[NCPU];
```

```
struct proc*
```

```
myproc(void) {  
    struct cpu *c;
```

```
    ...
```

```
    c = mycpu();    /* finds entry of cpus array  
                    using special "ID" register  
                    as array index */
```

```
    p = c->proc;
```

```
    ...
```

```
    return p;
```

```
}
```

this class: focus on Unix

Unix-like OSes will be our focus

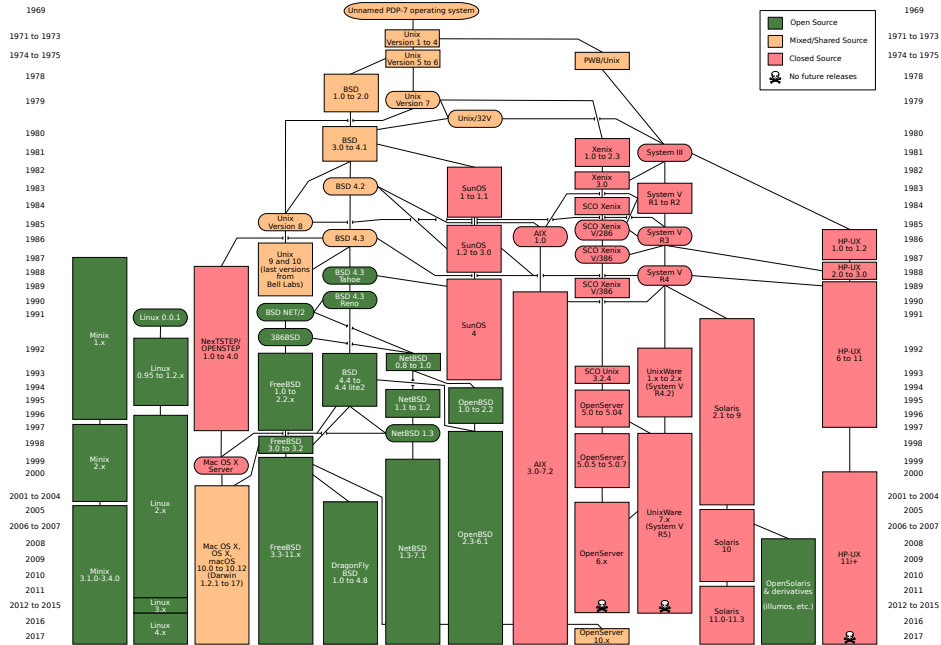
we have source code

used to from 2150, etc.?

have been around for a while

xv6 imitates Unix

Unix history



POSIX: standardized Unix

Portable Operating System Interface (POSIX)

“standard for Unix”

current version online:

<http://pubs.opengroup.org/onlinepubs/9699919799/>

(almost) followed by most current Unix-like OSes

...but OSes add extra features

...but 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, Linux was very immature

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`

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

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

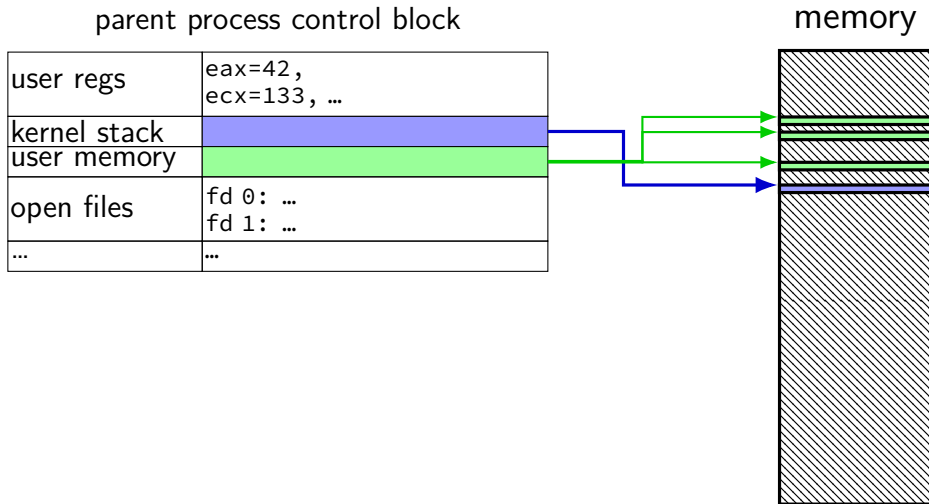
everything (but pid) duplicated in parent, child:

memory

file descriptors (later)

registers

fork and PCBs

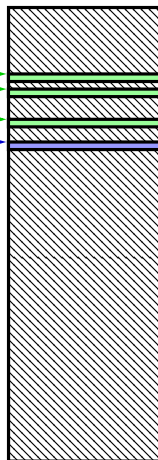


fork and PCBs

parent process control block

user regs	eax=42, ecx=133, ...
kernel stack	
user memory	
open files	fd 0: ... fd 1: ...
...	...

memory

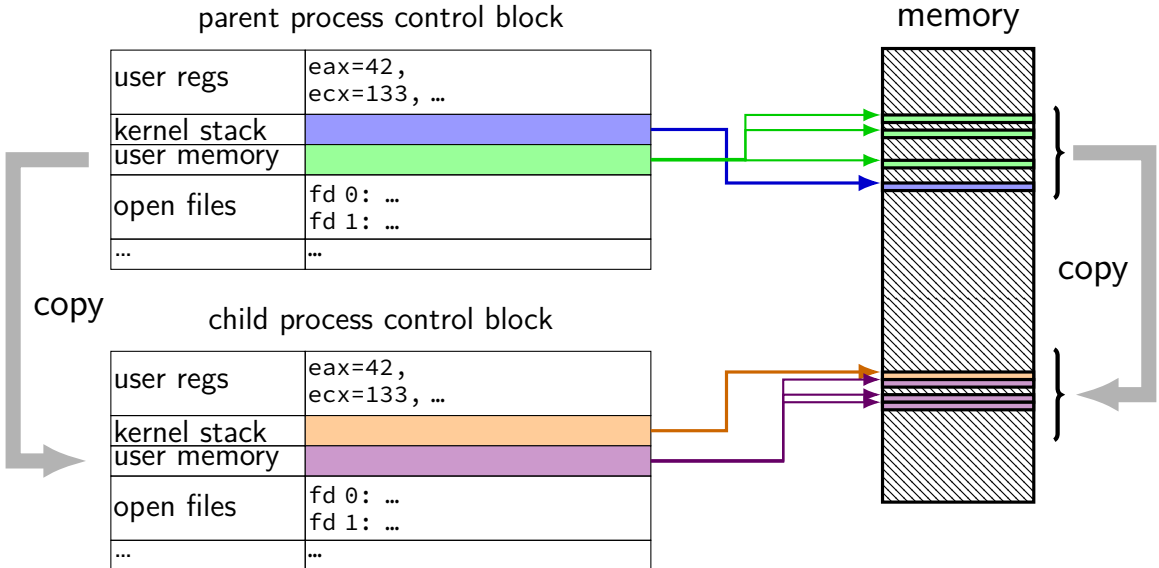


copy

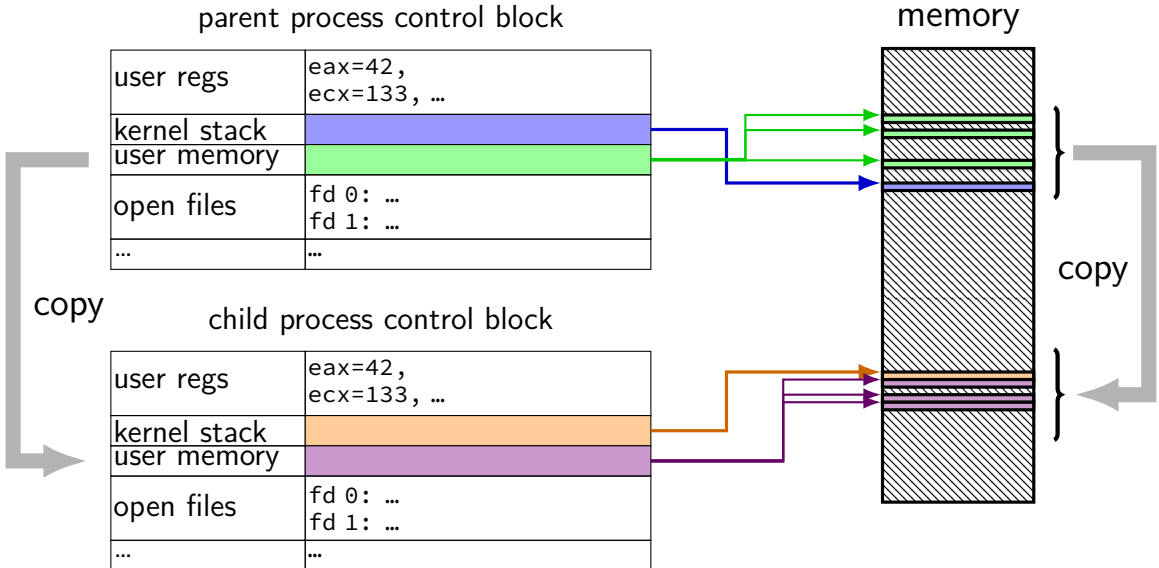
child process control block

user regs	eax=42, ecx=133, ...
kernel stack	
user memory	
open files	fd 0: ... fd 1: ...
...	...

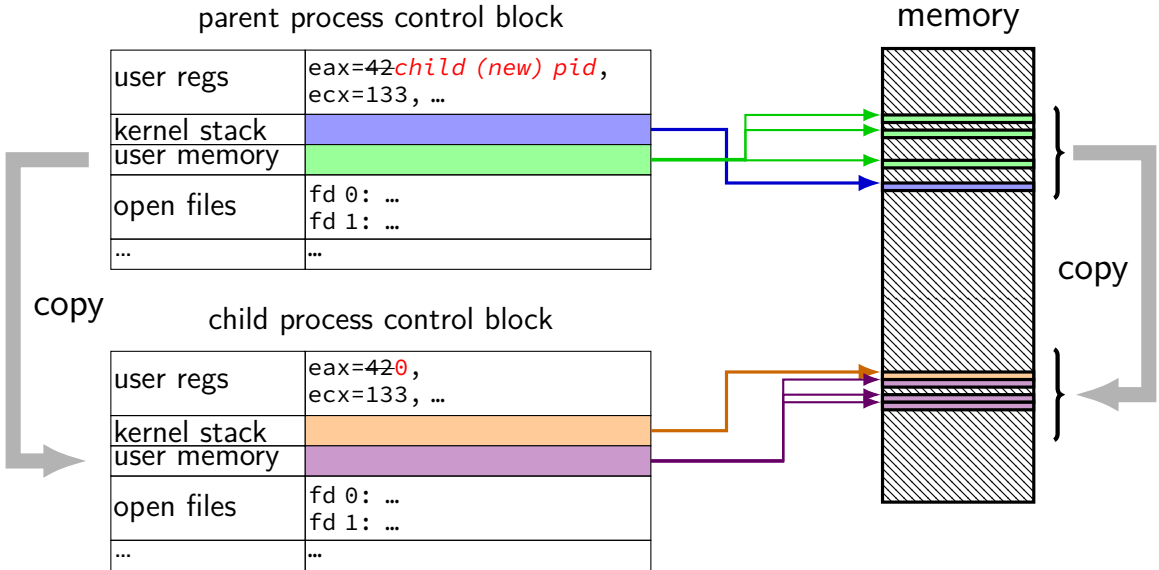
fork and PCBs



fork and PCBs



fork and PCBs



fork example

```
#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

getpid — returns current process pid

```
#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

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char **argv) {
    pid_t pid = fork();
    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;
}
```

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 <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
int main()
{
    pid_t pid;
    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;
}
```

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

fork example

```
#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 execl(const char *path, const char **argv)
```

path: new program to run

argv: array of arguments, terminated 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

used to compute argv, argc

```
...
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 here, it failed. */
    perror("execv");
    exit(1);
} else if (child_pid > 0) {
    /* parent process */
    ...
}
```

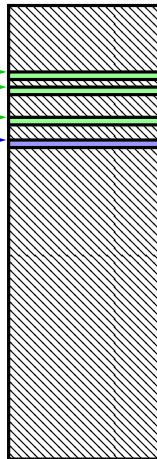
filename of program to run
need not match first argument
(but probably should match it)

exec and PCBs

the process control block

user regs	eax=42 <i>init. val.</i> , ecx=133 <i>init. val.</i> , ...
kernel stack	
user memory	
open files	fd 0: (terminal ...) fd 1: ...
...	...

memory

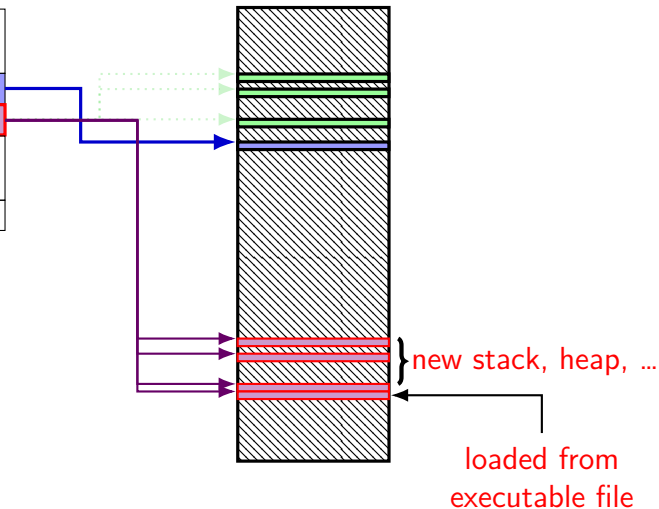


exec and PCBs

the process control block

user regs	eax=42 <i>init. val.</i> , ecx=133 <i>init. val.</i> , ...
kernel stack	
user memory	
open files	fd 0: (terminal ...) fd 1: ...
...	...

memory

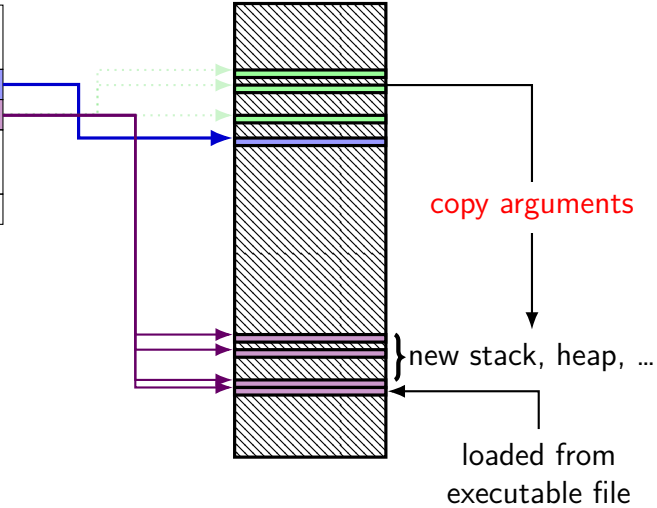


exec and PCBs

the process control block

user regs	eax= <i>42</i> init. val., ecx= <i>133</i> init. val., ...
kernel stack	
user memory	
open files	fd 0: (terminal ...) fd 1: ...
...	...

memory



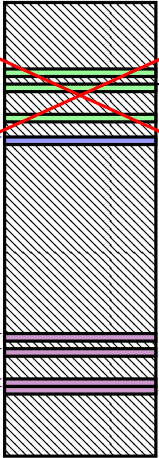
exec and PCBs

the process control block

user regs	eax= <i>42</i> init. val., ecx= <i>133</i> init. val., ...
kernel stack	
user memory	
open files	fd 0: (terminal ...) fd 1: ...
...	...

not changed!
(more on this later)

memory



old memory
discarded

copy arguments

} new stack, heap, ...

loaded from
executable file

execv and const

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

argv is a pointer to constant pointer to char

probably should be a pointer to constant pointer to *constant* char

...this causes some awkwardness:

```
const char *array[] = { /* ... */ };  
execv(path, array); // ERROR
```

solution: cast

```
const char *array[] = { /* ... */ };  
execv(path, (char **) array); // or (char * const *)
```

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:/us
```

```
PWD=/zf14/cr4bd
```

```
LANG=en_US.UTF-8
```

```
MODULEPATH=/sw/centos/Modules/modulefiles:/sw/linux-any/Modules/modulefiles
```

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

...

why fork/exec?

could just have a function to spawn a new program

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

needs to include API to set new program state

open files, current directory, environment variables, ...

with fork: just use 'normal' API

POSIX: `posix_spawn`, but rarely used, often not implemented
(or implemented poorly)

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

WNOHANG — return 0 rather than hanging if process not yet done

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

WNOHANG — return 0 rather than hanging if process not yet done

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 (control-C causes signal %d)\n",
          WTERMSIG(status), SIGINT);
} 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 (control-C causes signal %d)\n",
           WTERMSIG(status), SIGINT);
} 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

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

...

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 */
}
```

'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:

```
init(1)-->ModemManager(919)-->{ModemManager}(972)
|
|   |--{ModemManager}(1064)
|   |--NetworkManager(1160)-->dhcpcd(1755)
|   |   |--dnsmasq(1985)
|   |   |--{NetworkManager}(1180)
|   |   |--{NetworkManager}(1194)
|   |   |--{NetworkManager}(1195)
|   |--accounts-daemon(1649)-->{accounts-daemon}(1757)
|   |--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(944)
|   |--bluetoothd(924)
|   |--colord(1193)-->{colord}(1329)
|
|   |--nongod(1336)-->{nongod}(1556)
|   |   |--{nongod}(1557)
|   |   |--{nongod}(1983)
|   |   |--{nongod}(2031)
|   |   |--{nongod}(2047)
|   |   |--{nongod}(2048)
|   |   |--{nongod}(2049)
|   |   |--{nongod}(2050)
|   |   |--{nongod}(2051)
|   |   |--{nongod}(2052)
|   |--msh-server(19090)-->bash(19091)-->tmux(5442)
|   |--msh-server(21996)-->bash(21997)
|   |--msh-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.tnmapd(875)
|   |--rpc.statd(954)
|   |--rpcbind(884)
|   |--rsrver(1561)-->{rsrver}(1786)
|   |   |--{rsrver}(1787)
|   |--rsyslogd(1090)-->{rsyslogd}(1092)
|   |   |--{rsyslogd}(1093)
|   |   |--{rsyslogd}(1094)
|   |--rtkit-daemon(2565)-->{rtkit-daemon}(2566)
|   |   |--{rtkit-daemon}(2567)
|   |--sd_clicero(2852)-->sd_clicero(2853)
|   |   |--{sd_clicero}(2854)
|   |   |--{sd_clicero}(2855)
|   |--sd_dunny(2849)-->{sd_dunny}(2850)
|   |   |--{sd_dunny}(2851)
|   |--sd_espeak(2749)-->{sd_espeak}(2845)
|   |   |--{sd_espeak}(2846)
|   |   |--{sd_espeak}(2847)
|   |   |--{sd_espeak}(2848)
|   |--sd_generic(2463)-->{sd_generic}(2464)
```

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()` / `wait()`s 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

typical pattern

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

```
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() {  
    …  
}
```

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, ...);
    ...
}
```

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);  
}
```

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`

shell

allow user (= person at keyboard) to run applications

user's wrapper around process-management functions

upcoming homework — make a simple shell

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 (not in assignment)

```
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 (not in assignment)

```
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
checked in order

one way to implement: [pseudocode]

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

some POSIX command-line features

searching for programs (not in assignment)

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

running in background

```
./someprogram &
```

redirection:

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

pipelines:

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

running in background

```
$ ./long_computation >tmp.txt &  
[1] 4049  
$ ...  
[1]+  Done          ./long_computation > tmp.txt  
$ cat tmp.txt  
the result is ...
```

& — run a program in “background”

initially output PID (above: 4049)

print out after terminated

one way: use `waitpid` with option saying “don’t wait”

some POSIX command-line features

searching for programs (not in assignment)

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

running in background

```
./someprogram &
```

redirection:

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

pipelines:

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


shell redirection

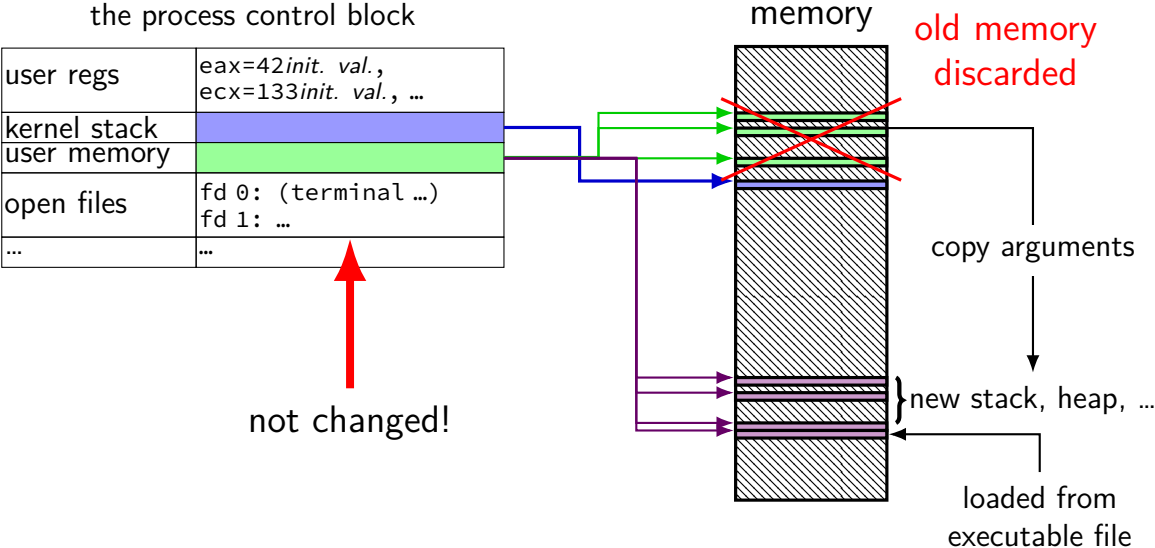
`./my_program ... <input.txt:`

run `./my_program ...` but use `input.txt` as input
like we copied and pasted the file into the terminal

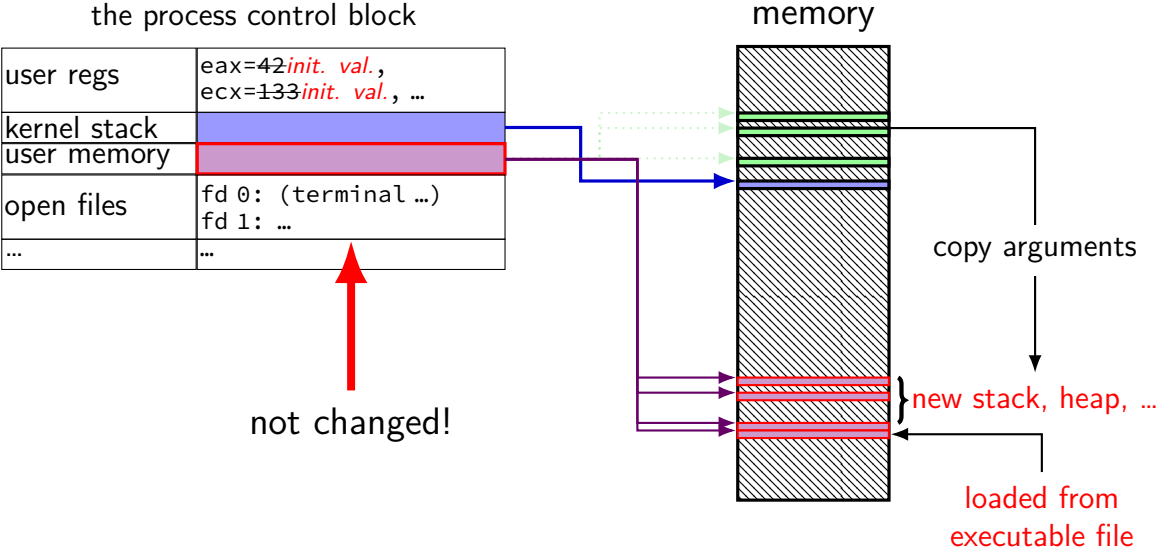
`echo foo >output.txt:`

runs `echo foo`, sends output to `output.txt`
like we copied and pasted the output into that file
(as it was written)

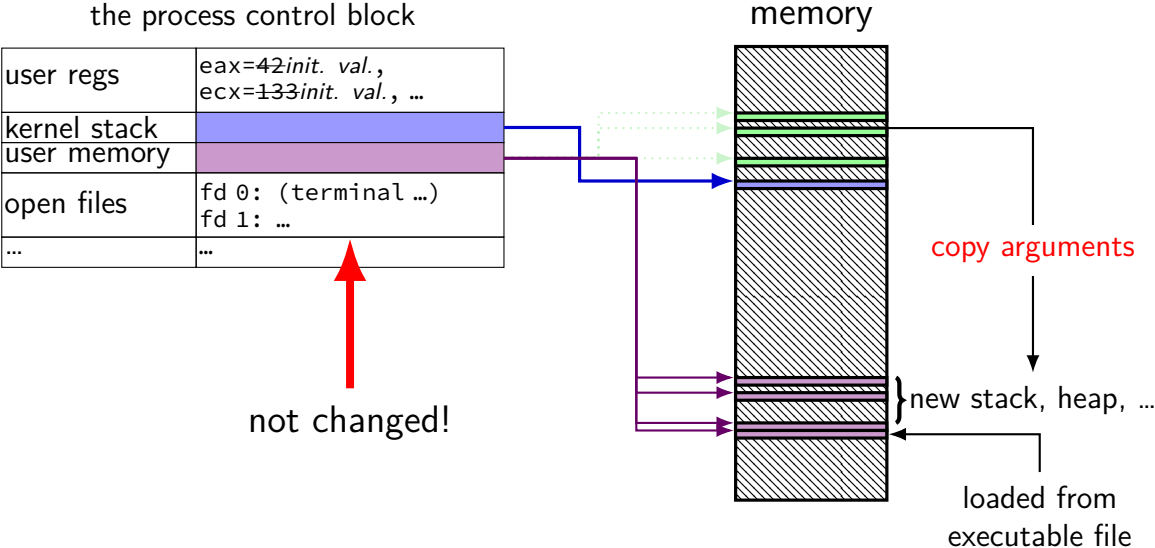
exec preserves open files



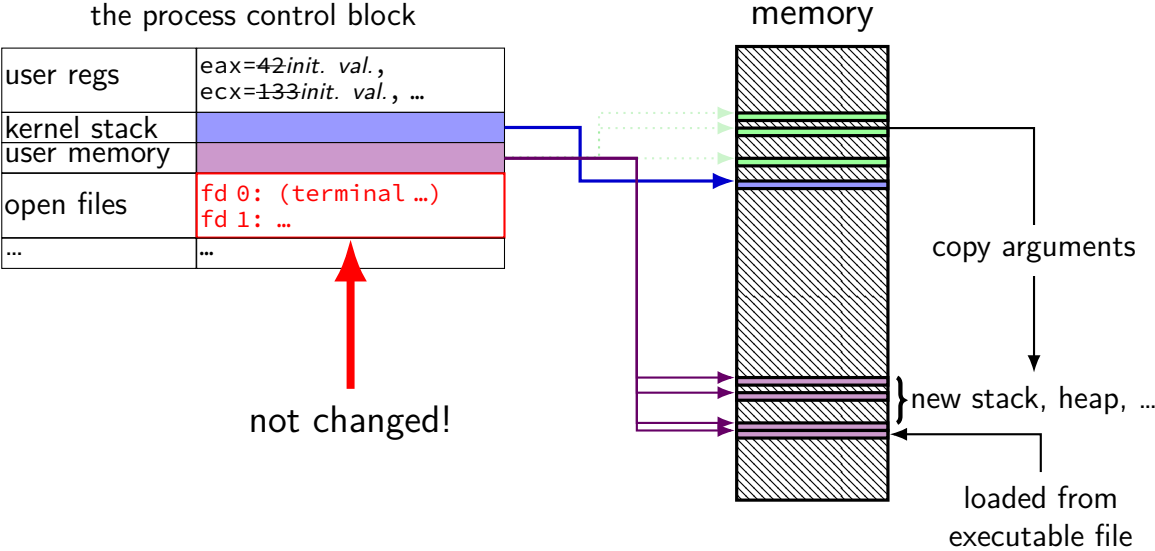
exec preserves open files



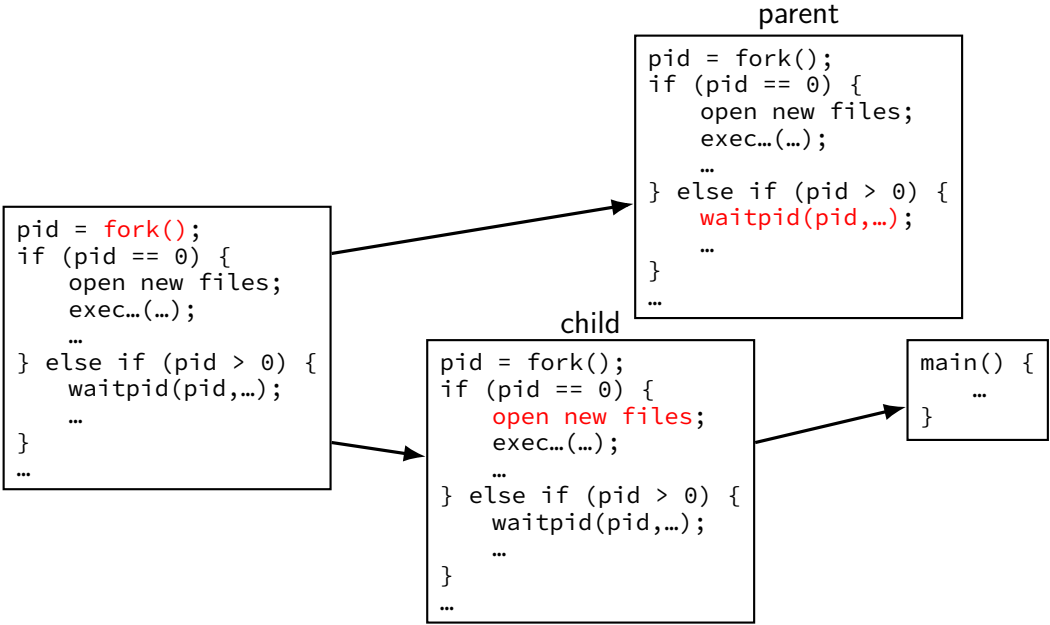
exec preserves open files



exec preserves open files



typical pattern with redirection



redirecting with exec

standard output/error/input are files

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

yes, your terminal is a file

more on this later

after forking, open files to redirect

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

typically using `dup2()`

some POSIX command-line features

searching for programs (not in assignment)

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

running in background

```
./someprogram &
```

redirection:

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

pipelines:

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

shell assignment

implement a simple shell that supports redirection and pipeline

...and prints the exit code of program in the pipeline

simplified parsing: space-separated:

okay: `/bin/ls -l > tmp.txt`

not okay: `/bin/ls -l >tmp.txt`

okay: `/bin/ls -l | /bin/grep foo > tmp.txt`

not okay: `/bin/ls -l | /bin/grep foo >tmp.txt`