Discussion 2: Signal Handling

Lab 1B

signal handling

✓ --abort

Get it to segmentation fault!

Do it by referencing a pointer set to NULL;

--catch N

Catch any processes that send signal N.

```
#include <signal.h>
#include <stdlib.h>

void segfault_sigaction (int signal, siginfo_t *s,
    void *arg) { exit(signal); }

void segfault_sighandler (int signal)
  { exit (signal);
                    exits needed or else an attempt to
                    be made to reexecute a violating
int main            command will be made forever
  struct_sigaction sa;
  sa.sa_sigaction = segfault_sigaction)
  // sa.sa_handler = segfault_sighandler
  sigaction(SIGSEGV, &sa, NULL)
    int *a = NULL
    int b = *a;
```

--pause We need an external segment fault...
How do we achieve this?

```
#include <unistd.h> // for pause
```

```
void pauseHandler( int signal, siginfo_t * si , void* ar
{
    // do something
}
int main ()
{
    struct sigaction sa;
    sa.sa_sigaction = pauseHandler;
    sigaction (SIGSEGV, &sa, NULL);
    sigaction (SIGINFO, &sa, NULL);
    pause ();
    printf ("continue");
}
```

--ignore(:)

Two ways to try to do it:

- Choose to change the instruction pointer (rip)

rax: 64
eax: 32
ax: 16
ah: 8 → high
al: 8 → low

```
# define _GNU_SOURCE
# include <signal.h>
# include <ucontext.h>
# include <sys/main.h>
```

```
void segfault_sigaction(int signal, siginfo_t *si, void *arg)
{
    //When ignore is enabled
    ucontext_t *context = (ucontext_t *)arg;
    context->uc_mcontext.gregs[REG_RIP]++;
}
```

Lecture 5: 1/20/16

Big OS Goals: A Recap

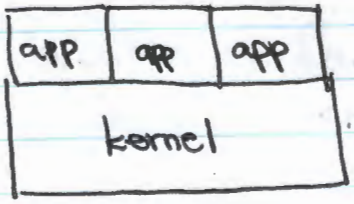
Protection Robustness Flexibility Simplicity

Utilization Performance Abstraction Modularity

x86 supports 4 level's of abstraction

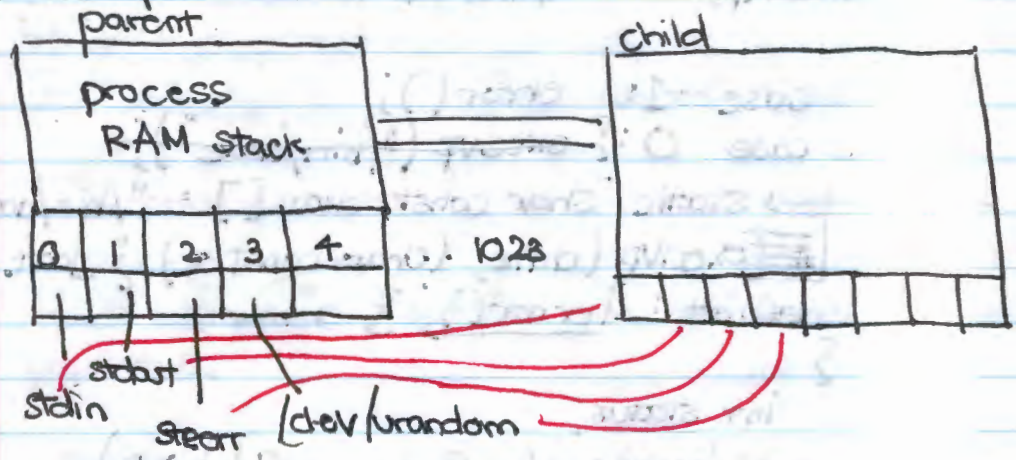
Linux does only 2: apps and kernel
- faster!
- simpler

Here's a different approach:



fork() clone a process, except for ~~process ID~~

1. process ID → pid_t getpid(void);
2. parent ID → pid_t getppid(void); ← return parent's pid
3. file descriptor tables



/dev/urandom is not really random. It uses entropy to seed a RNG.

The child has a file description.

4. CPU time info
5. Pending signals
6. File locks

file argv

execvp(char const *, char * const*)

It destroys a process, and throws everything away, except:

You destroy:
- all variables
- instruction pointer %rip
- registers
- program

∴ We must also reset the signal handlers

v1

```
bool ~~void~~ printdate(void)
{
pid_t p=fork();   // ensure that we don't terminate as soon as
 switch(p)        // the call is over
 {
     case -1: error();
     case 0: { ~~execvp("/bin/date");~~
     static char const date[] = "/bin/date";
     execvp(date, (char const *) {date, NULL});
     default: error(); }
   {
     int status;
     if (waitpid (p, &status, 0) != p)   // true if process exited
       error();
     return (~~WEX~~ WIFEXITED(status)
       && WEXITSTATUS(status) == 0);
   }
}
```

If we were to pass in a parameter char const *outfile for this function, we would open it after the child is successfully created (⟶)

```
int fd = open(outfile, O_WRONLY);
if (fd < 0) error();
if (dup2(fd, 1) < 0) error();
if (fd != 1) close(0)

if (fork() == 0)
{                    ← done before execvp(...)
  // housekeeping
}
```

Another totally different school of thought handles all the housekeeping in one system call.

```
int posix_spawnvp(pid_t * restrict pid;)
```

What does restrict do? It prevents aliasing, allowing for additional optimizations
It's a keyword.

ex: `char * strcpy(char * restrict dest, char * const restrict src)`

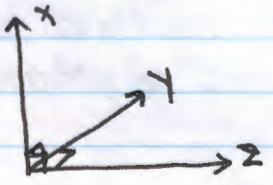
Posix_spawnvp is actually more complicated in terms of parameters.

```
int posix_spawnvp( pid_t * restrict pid,
                   char const * restrict file,
                   posix_spawn_file_actions_t const *
                    restrict file_acts,
                   posix_spawn_attr_t const *
                    restrict attr,
                   char * const * restrict argv,
                   char * const * restrict envp);
```

Look at how annoying this is to use!
Upside: its fast.

Orthogonality

In math:



I can pick any x_0, then freely pick any y_0 and z_0 and form the coordinates (x_0, y_0, z_0).
We want our system calls to be orthogonal to each other.

Orthogonality works with processes

Processes interacting with files, it is slow and unreliable; problems involving robustness and performance.

Network mouse keyboard	vs	flash disk
spontaneous data generation infinite* stream		random access request/response finite

UNIX BIG IDEA

everything is a file.

- open
- lseek
- close
- read
- write
- dup2

The principle is to make a set of operations that works with a lot of different things.

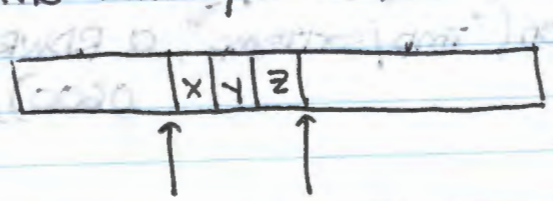
One downside is that we may have ops and files that are incompatible with each other.

There's another situation we must cover: when processes communicate with each other.

Pipes are bounded buffers.

process (A)
write ('xyz')

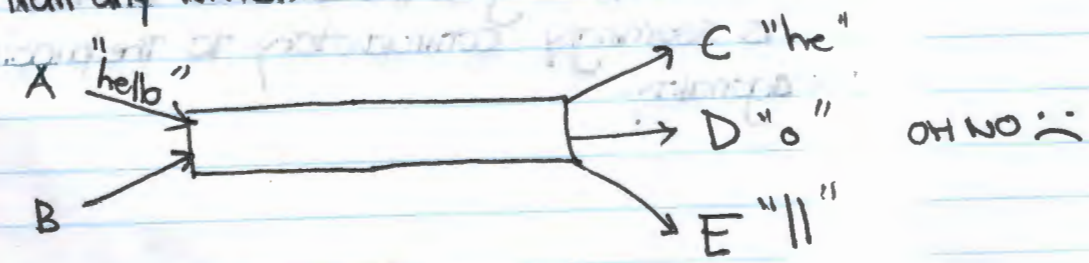
kernel memory buffer



first in first out buffers
If the buffer is written, it is added to the buffer
If the buffer is read, it is removed after reading
If the buffer is full and a write is attempted, the buffer will force the writing to wait.
If the buffer is empty and a read is attempted, the buffer will force the reading to wait.

In this way, the pipe can control the flow of data.

However, in this form any reader will get the messages from any writer.



Race conditions - when behavior of a program changes depending on when things get scheduled.

Say we want to implement a sort.

We might need to create a temporary file.

```
int create_temp_file(void) {
    return open("tmp/sorttemp", O_RDWR|O_TRUNC|O_CREAT,
                0600);
}
```

This breaks because two concurrent sorts will result in an overwritten temporary file!

How do we fix this?

O_EXCL: will cause it to not work if the file exists already!.

Then, we use a RNG to generate a unique filename for us that will hopefully not cause any overlap.

Now we have primitives that work, but cause race conditions depending on the distance between system calls. This is seemingly contradictory to the principle of orthogonality.