15-410

"Luckily the stack is a simple data structure."

The Process
Jan. 19, 2011

Dave Eckhardt

Synchronization

Project 0 due tonight

- **18 students have already turned something in**
- **Please make sure you can write into your hand-in directory *before 18:00* today**
- **Please see hand-in instructions on P0 web page**

Synchronization

Reminders on collaboration

- **Project 1 will be *individual***
- **Talking about code is ok**
- **Possessing the code of another is *not ok***
- **Different classes have different policies**
- **We expect you to read and follow the policies of *this* class**
    - **(As found in the syllabus, which you are required to read)**
    - **If something is unclear, please mail us**

Synchronization

P2/P3/P4 partners

- 50 students have already registered – thanks!
  - (Still some 1-way registrations)
- You should be partnered soon...
- If you already know who your partner is, please register now
  - It makes it easier for others to partner
  - It will stem the tide of annoying reminder e-mail
  - Both partners in a group should register

Synchronization

Anybody reading comp.risks?

This lecture

- Chapter 3, but not exactly!
  - We are skipping 3.5 and 3.6, including the terrifying "POSIX Shared Memory"

Outline

Process as pseudo-machine

- (that's *all* there is)

Process life cycle

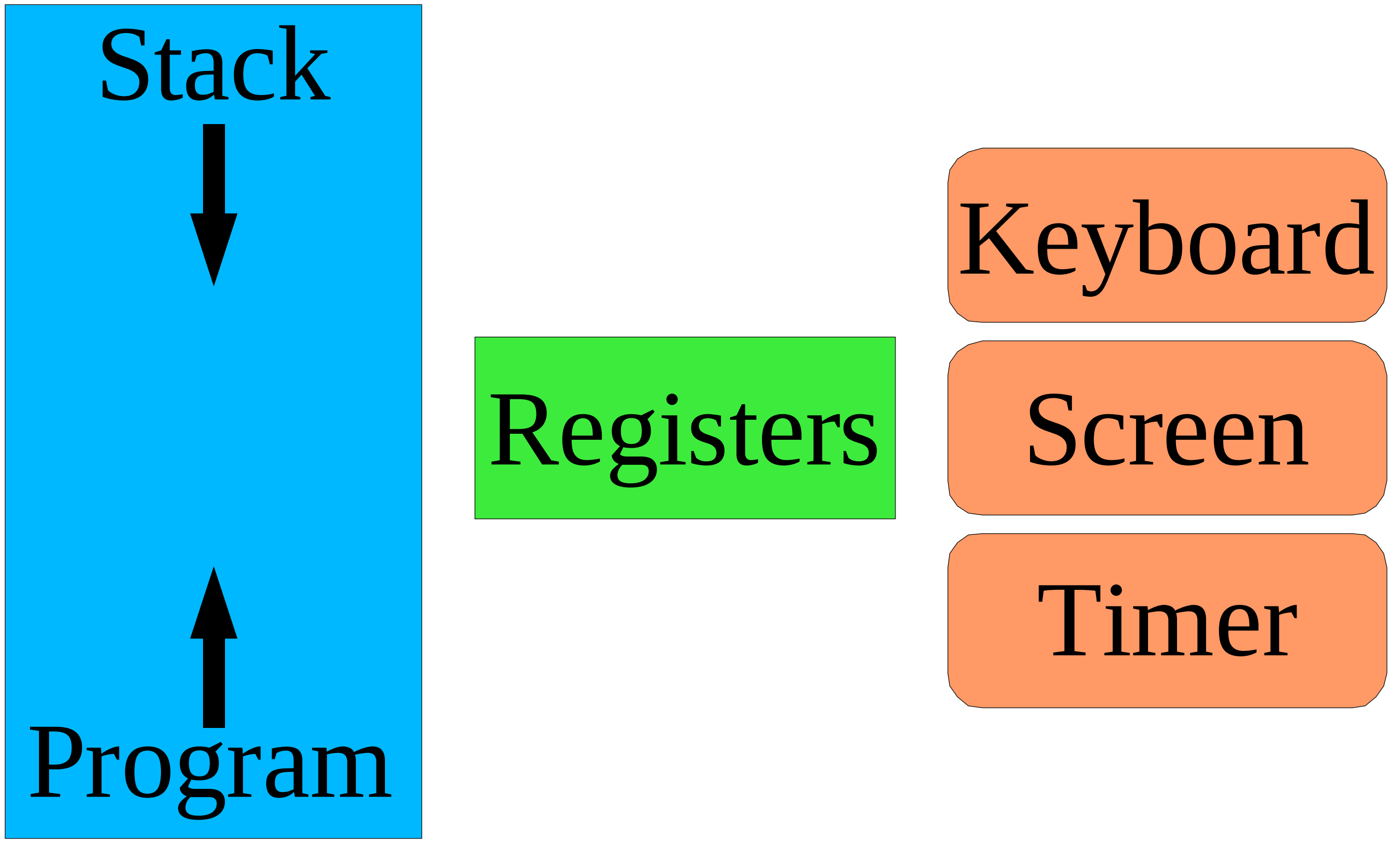
Process kernel states

Process kernel state

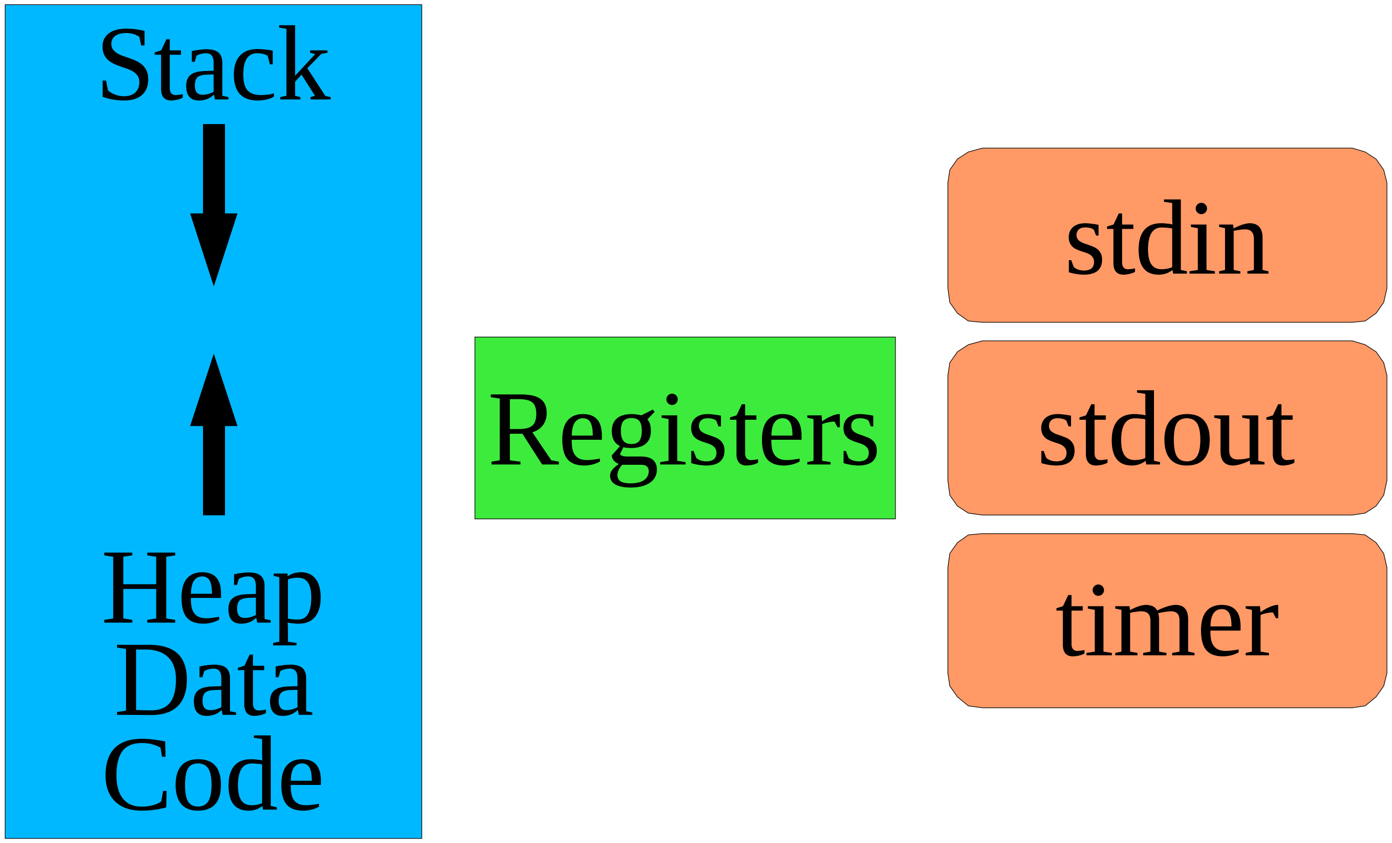
P1/P3 memory layout

- (just a teaser for now)

The Computer



The Process



Process life cycle

(nomenclature courtesy of The Godfathers)

Birth

- (or, well, fission)

School

Work

Death

Birth

Where do new processes come from?

- (Not: under a cabbage leaf, by stork, ...)

What do we need?

- Memory contents
  - Text, data, stack
- CPU register contents (N of them)
- “I/O ports”
  - File descriptors, e.g., stdin/stdout/stderr
- Hidden “stuff”
  - timer state, current directory, umask

Birth

Intimidating?

How to specify all of that stuff?

- What is your {name,quest,favorite_color}?

Gee, we already have *one* process we like...

- Maybe we could use its settings to make a new one...
- Birth via "cloning"

Birth – fork() - 1

“fork” - Original Unix process creation system call

Memory

- Copy all of it
- Later lecture: VM tricks may make copy cheaper

Registers

- Copy all of them
  - All but one: parent learns child's process ID, child gets 0

Birth – fork() - 2

File descriptors

- Copy all of them
- Can't copy the *files!*
- Copy *references* to open-file state

Hidden stuff

- Do whatever is "obvious"

Result

- Original, “parent”, process
- Fully-specified “child” process, despite 0 parameters to fork()

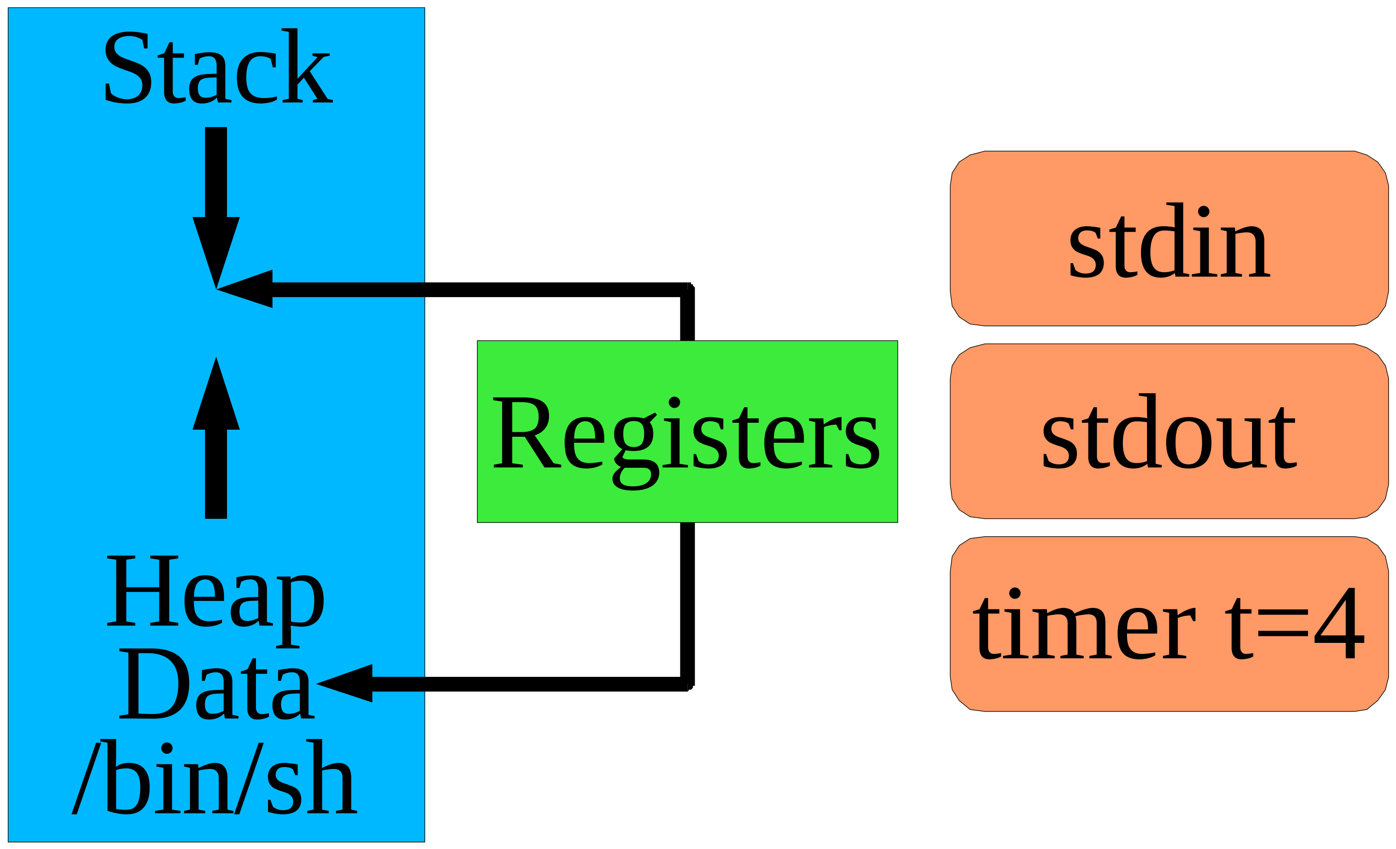
Now what?

Two copies of the same process is *boring*

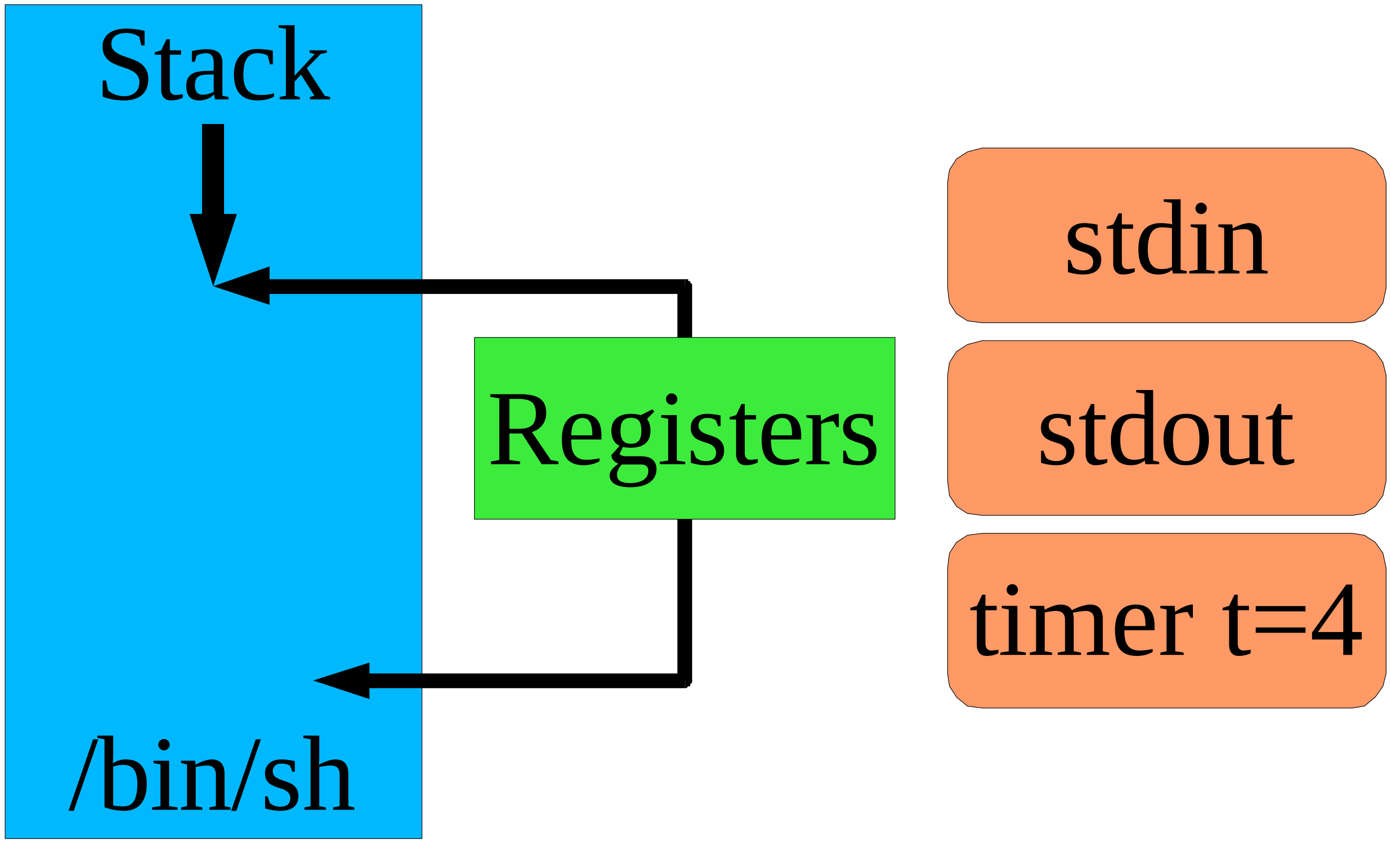
Transplant surgery!

- **Implant new memory!**
  - **New program text**
- **Implant new registers!**
  - **Old ones don't point well into the new memory**
- **Keep (most) file descriptors**
  - **Good for cooperation/delegation**
- **Hidden state?**
  - **Do what's “obvious”**

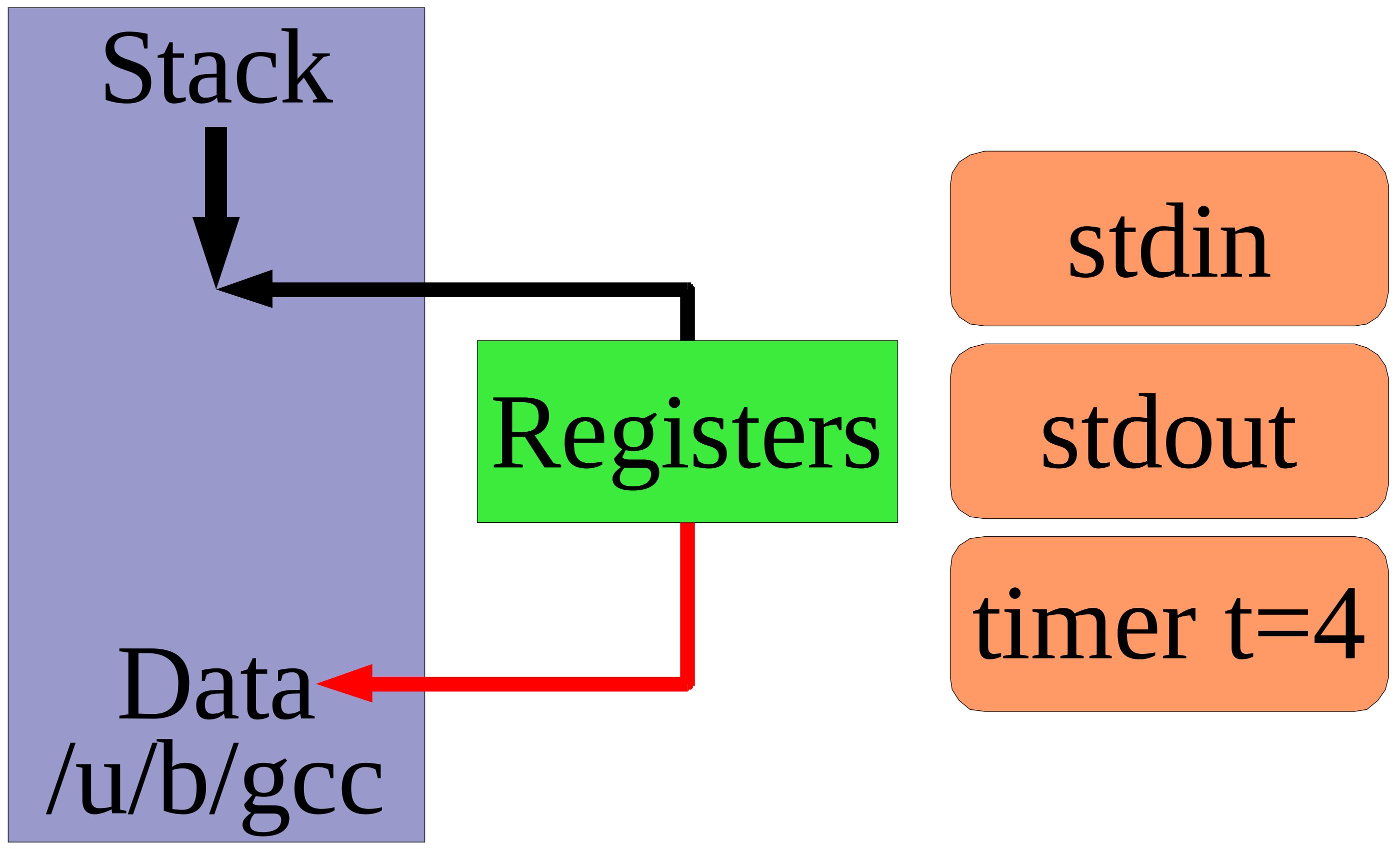
Original Process



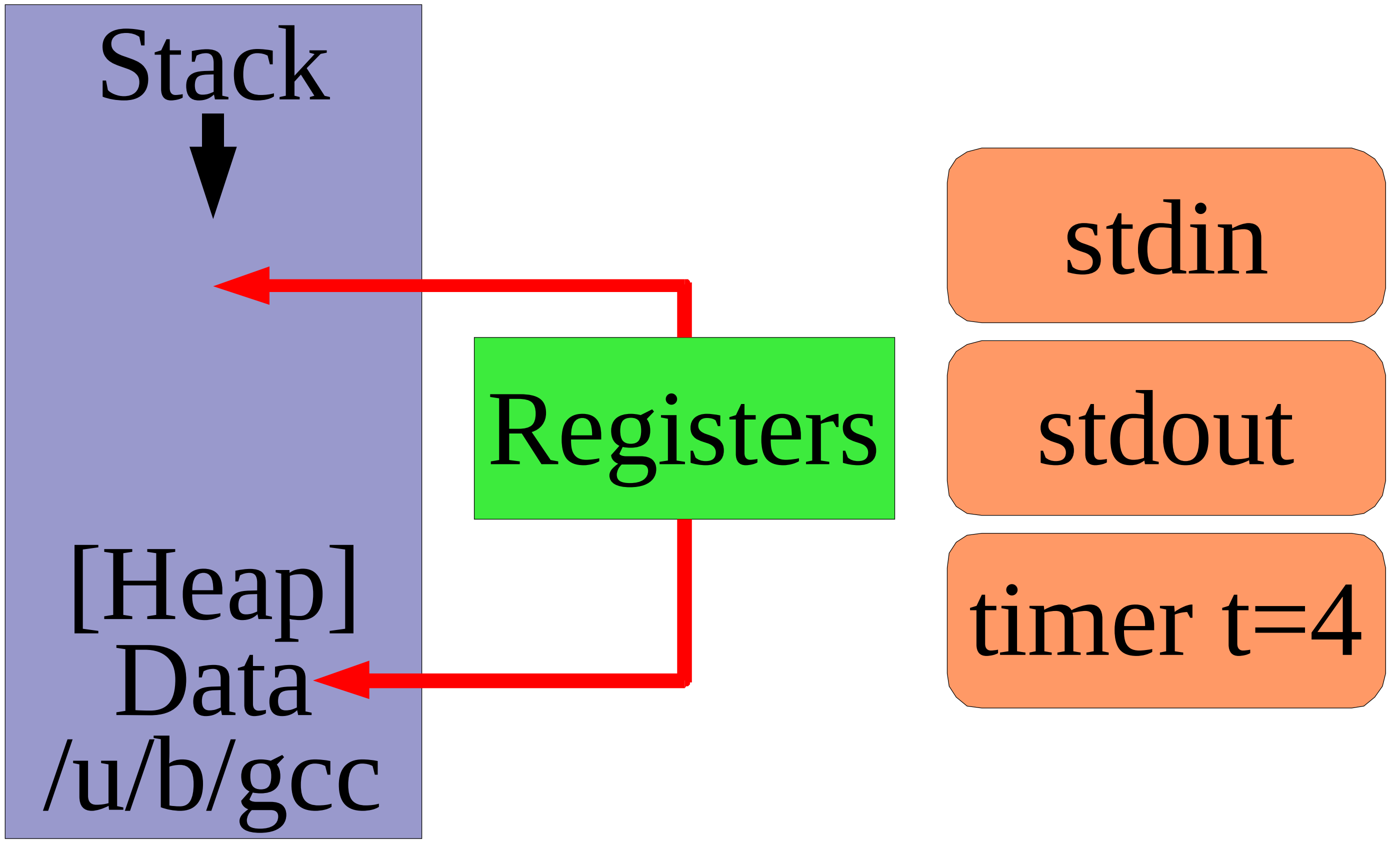
Toss Heap, Data



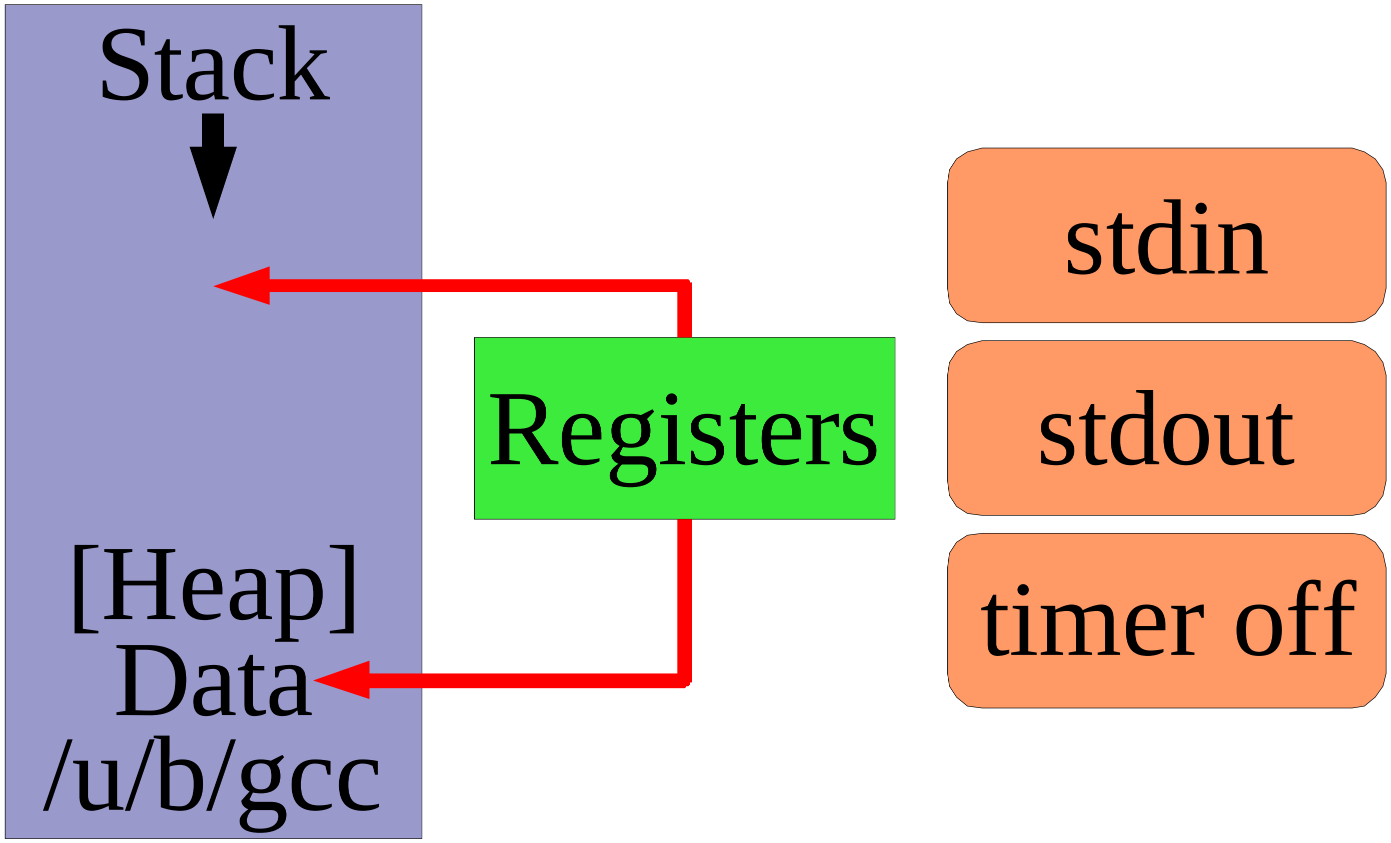
Load New Code, Data From File



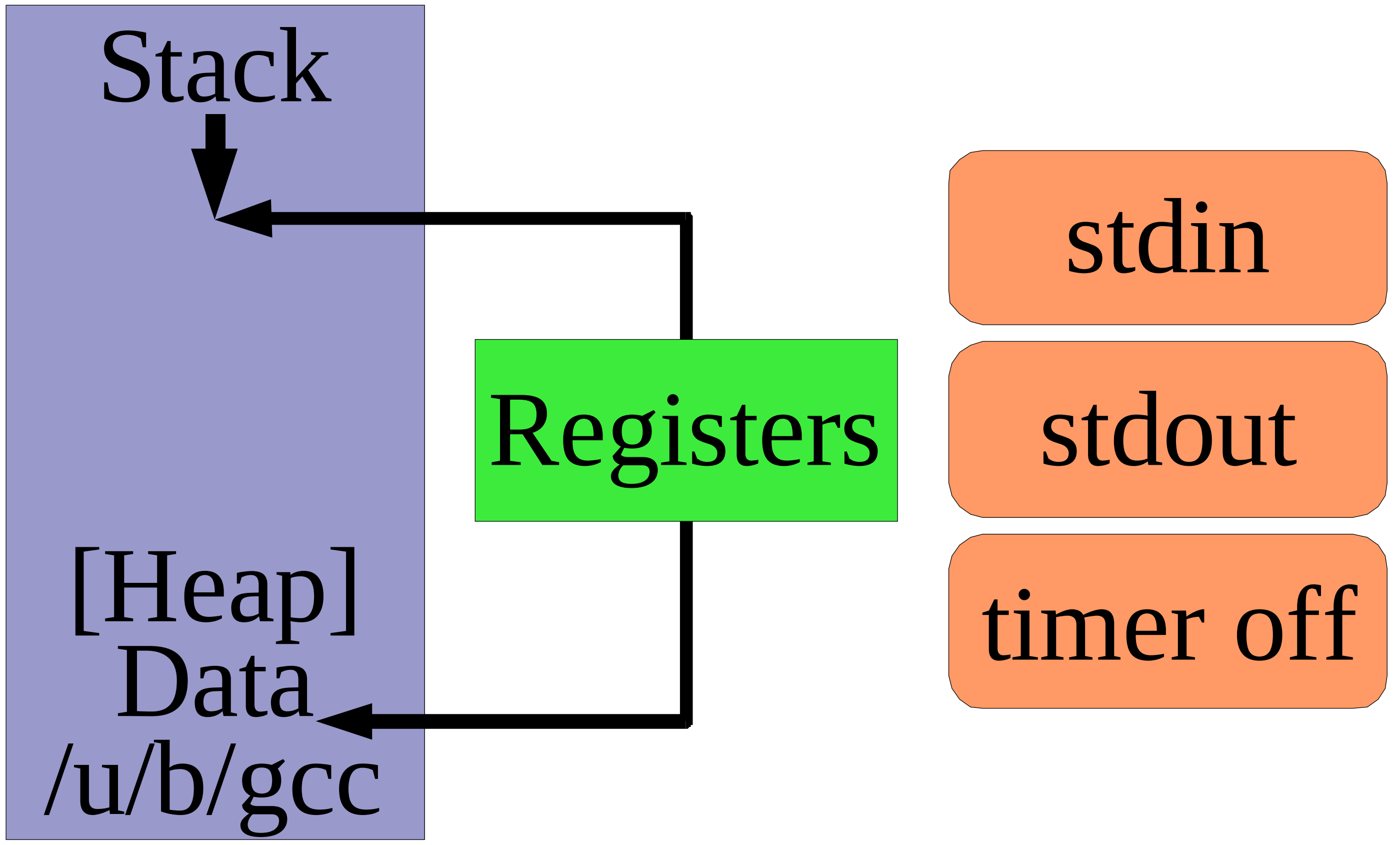
Reset Stack, Heap



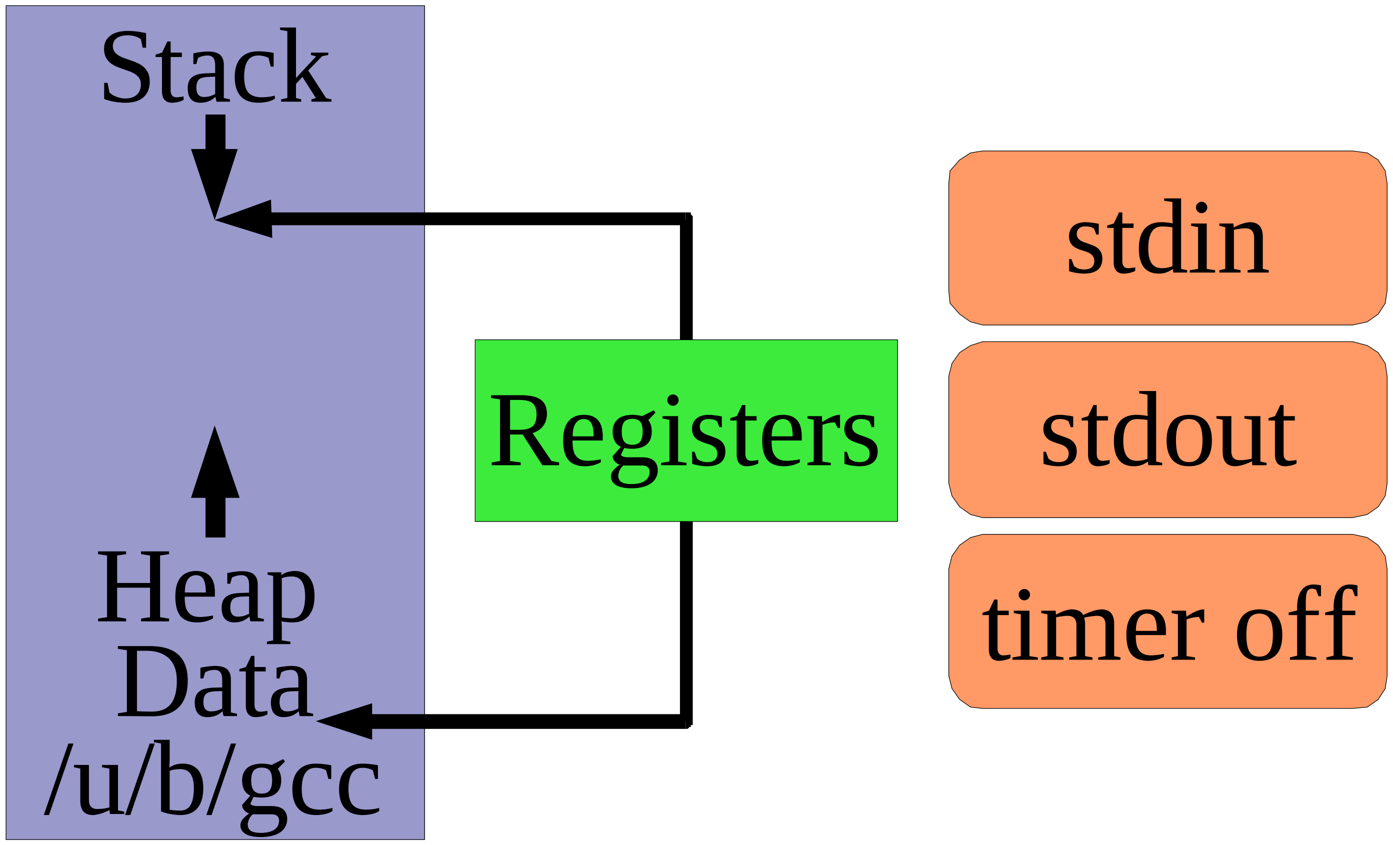
Fix “Stuff”



Initialize Registers



Begin Execution



What's The Implant Procedure Called?

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

Birth - other ways

There is another way

- Well, two

spawn()

- Carefully specify all features of new process
  - Complicated
- Win: don't need to copy stuff you will immediately toss

Plan 9 rfork() / Linux clone()

- Build new process from old one
- Specify which things get shared vs. copied
  - “Copy memory, share files, copy environment, share ...”

School

Old process called

```
execve(
char *path,
char *argv[ ],
char *envp[ ]);
```

Result is

```
main(int argc,
     char *argv[ ],
     char *envp[])
{
  ...
}
```

School

How does the magic work?

- ***15-410 motto: No magic***

Kernel process setup: we saw...

- **Toss old data memory**
- **Toss old stack memory**
- **Load executable file**

Also...

The Stack!

Kernel builds stack for new process

- Transfers argv[] and envp[] to top of new process stack
- Hand-crafts stack frame for __main()
- Sets registers
  - Stack pointer (to top frame)
  - Program counter (to start of __main())

Work

Process states

- Running
  - User mode or kernel mode
- Blocked
  - Awaiting some event
    - » I/O completion, exit of another process, message, ...
    - » Maybe sleeping for a fixed period of time
  - Scheduler: “do not run”
  - Q: User mode, kernel mode, both, neither?
- Runnable
  - Q: User mode, kernel mode, both, neither?
    - » Be sure to understand this

Work

Other process states

- Forking
  - Probably obsolete, once used for special treatment
- Zombie
  - Process has called exit(), parent hasn't noticed yet

“Exercise for the reader”

- Draw the state transition diagram

Death

Voluntary

```
void exit(int reason);
```

Hardware exception

- SIGSEGV - no memory there for you!

Software exception

- SIGXCPU – used "too much" CPU time

Death

System call - kill(pid, sig);

- “Deliver `sig` to process `pid`”
  - (negative values of `pid` have “interesting” behaviors)
- Keyboard `^C` ⇒ equivalent of
  - `kill(getpid(), SIGINT);`
- Start/stop logging
  - `kill(daemon_pid, SIGUSR1);`
  - `% kill -USR1 33`
  - `% kill -USR2 33`
  - This is a “non-kill” use of kill()
- Any other key uses of `kill()`?

Death

System call - kill(pid, sig);

- "Deliver `sig` to process `pid`"
  - (negative values of `pid` have "interesting" behaviors)
- Keyboard `^C` ⇒ `kill(getpid(), SIGINT);`
- Start/stop logging - `kill -USR1 33`
- "Lost in Space"!!
  - `kill(Will_Robinson, SIGDANGER);`

Death

System call - kill(pid, sig);

- **"Deliver `sig` to process `pid`"**
  - **(negative values of `pid` have "interesting" behaviors)**
- **Keyboard `^C` ⇒ `kill(getpid(), SIGINT);`**
- **Start/stop logging - `kill -USR1 33`**
- **"Lost in Space"!!**
  - **`kill(Will_Robinson, SIGDANGER);`**
  - **I apologize to IBM for lampooning their serious signal**

Death

System call - kill(pid, sig);

- "Deliver `sig` to process `pid`"
  - (negative values of `pid` have "interesting" behaviors)
- Keyboard `^C` ⇒ `kill(getpid(), SIGINT);`
- Start/stop logging - `kill -USR1 33`
- "Lost in Space"!!
  - `kill(Will_Robinson, SIGDANGER);`
  - I apologize to IBM for lampooning their serious signal
    - » No, I apologize for that apology...

Process cleanup

Resource release

- Open files: close() each
  - TCP: 2 minutes (or more)
  - Solaris disk offline - forever ("*None* shall pass!")
- Memory: release

Accounting

- Record resource usage in a magic file

Gone?

“All You Zombies...”

Zombie process

- Process state reduced to exit code
- Waits around until parent calls wait()
  - Exit code copied to parent's memory
  - PCB deleted from kernel

Kernel process state

The dreaded "PCB"

- **(polychlorinated biphenol?)**

Kernel process state

The dreaded "PCB"

- (polychlorinated biphenol?)

Process Control Block

- “Everything without a user-visible memory address”
  - Kernel management information
  - Scheduler state
  - The “stuff”

Sample PCB contents

Pointer to CPU register save area

Process number, parent process number

Countdown timer value

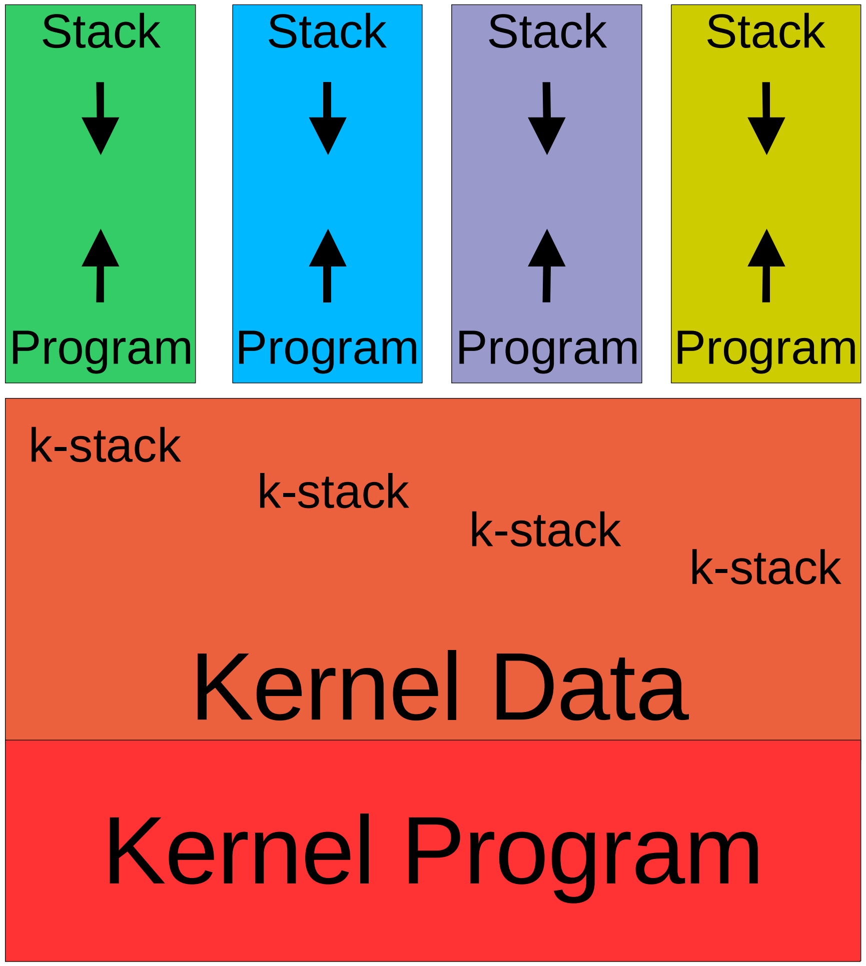
Memory segment info

- User memory segment list
- Kernel stack reference

Scheduler info

- linked list slot, priority, “sleep channel”

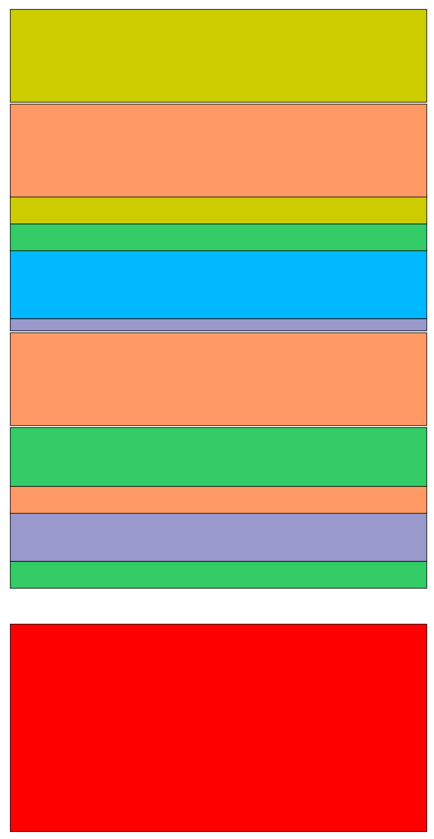
15-410 Virtual Memory Layout



4080 MB

16 MB

15-410 Physical Memory Layout



Ready to Implement All This?

Not so complicated...

- getpid()
- fork()
- exec()
- wait()
- exit()

What could possibly go wrong?

Summary

Parts of a Process

- **Physical – Memory pages, registers, I/O devices**
- **Virtual – Memory regions, registers, I/O “ports”**

Birth, School, Work, Death

“Big Picture” of system memory – both of them

- **(Numbers & arrangement are 15-410–specific)**