EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Slides from Christopher Kruegel, UCSB

Shell project

- Project 1 is now available on Piazza
- Deadline: February 15th at 16:30 EDT
- Project 0 autograder will remain available
- Suggest you solve project 0 then project 1

Review

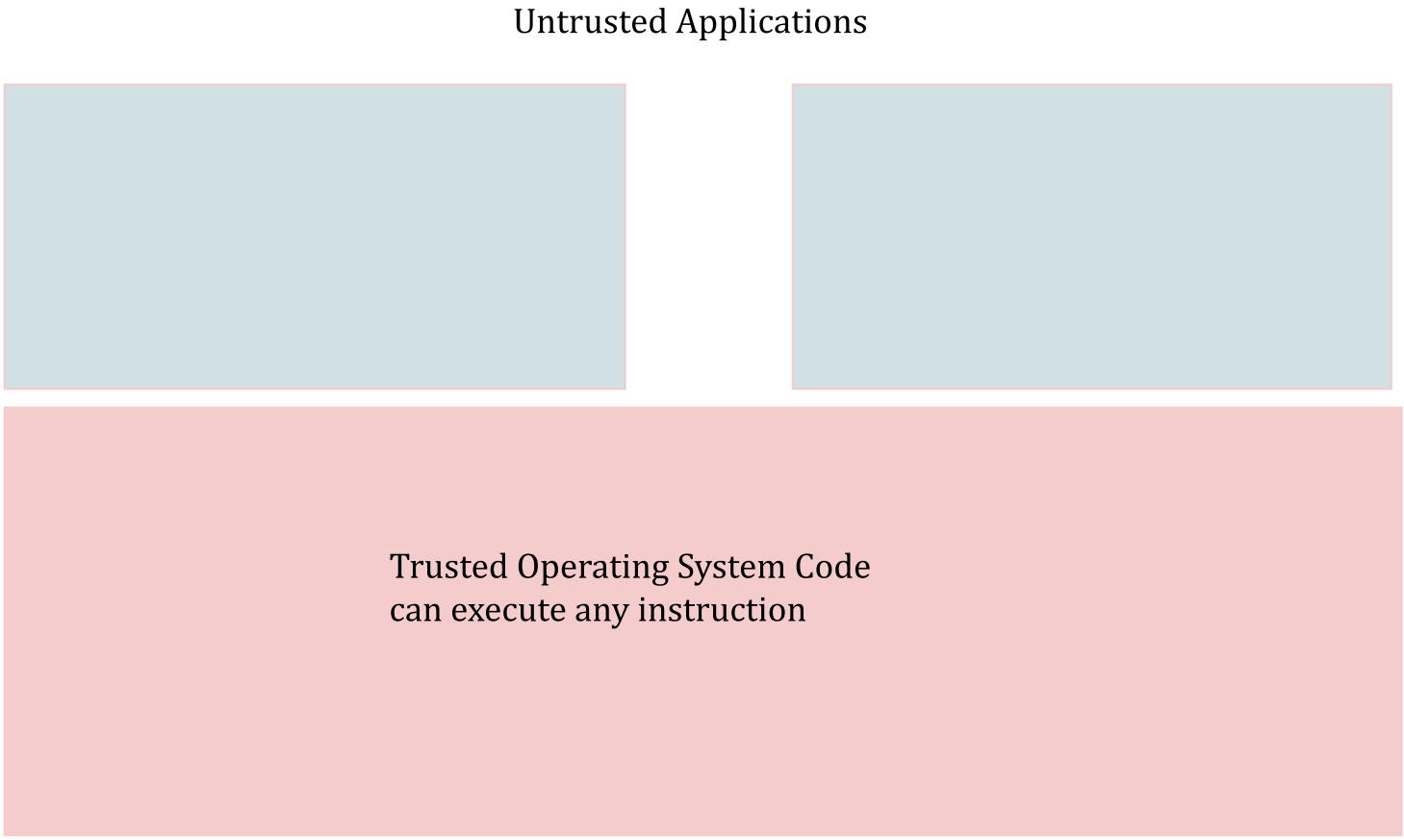
- Purpose of the OS: 1) provide clean abstractions, 2) share the HW
- OS is the fundamental platform in computing...

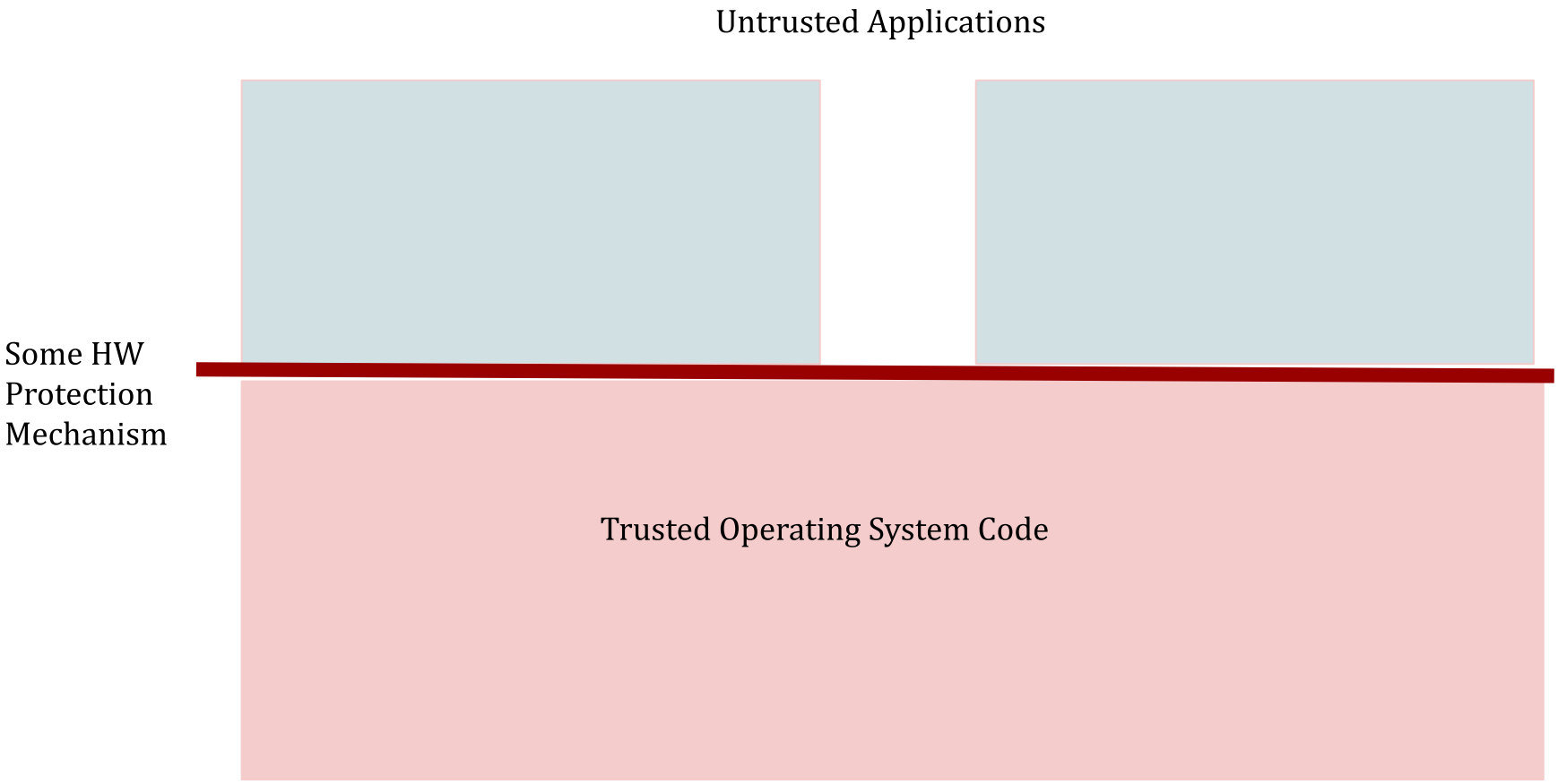
Review continued

- Role of the OS:
  - Provide abstractions over ugly hardware
  - Space/time multiplex hardware
- Key abstractions:
  - Process - virtual computer, container for:
    - Threads - virtual CPU
    - Address - virtual memory, every process has its own range from 0 - 2^{48}
    - File system - abstraction for persisting data

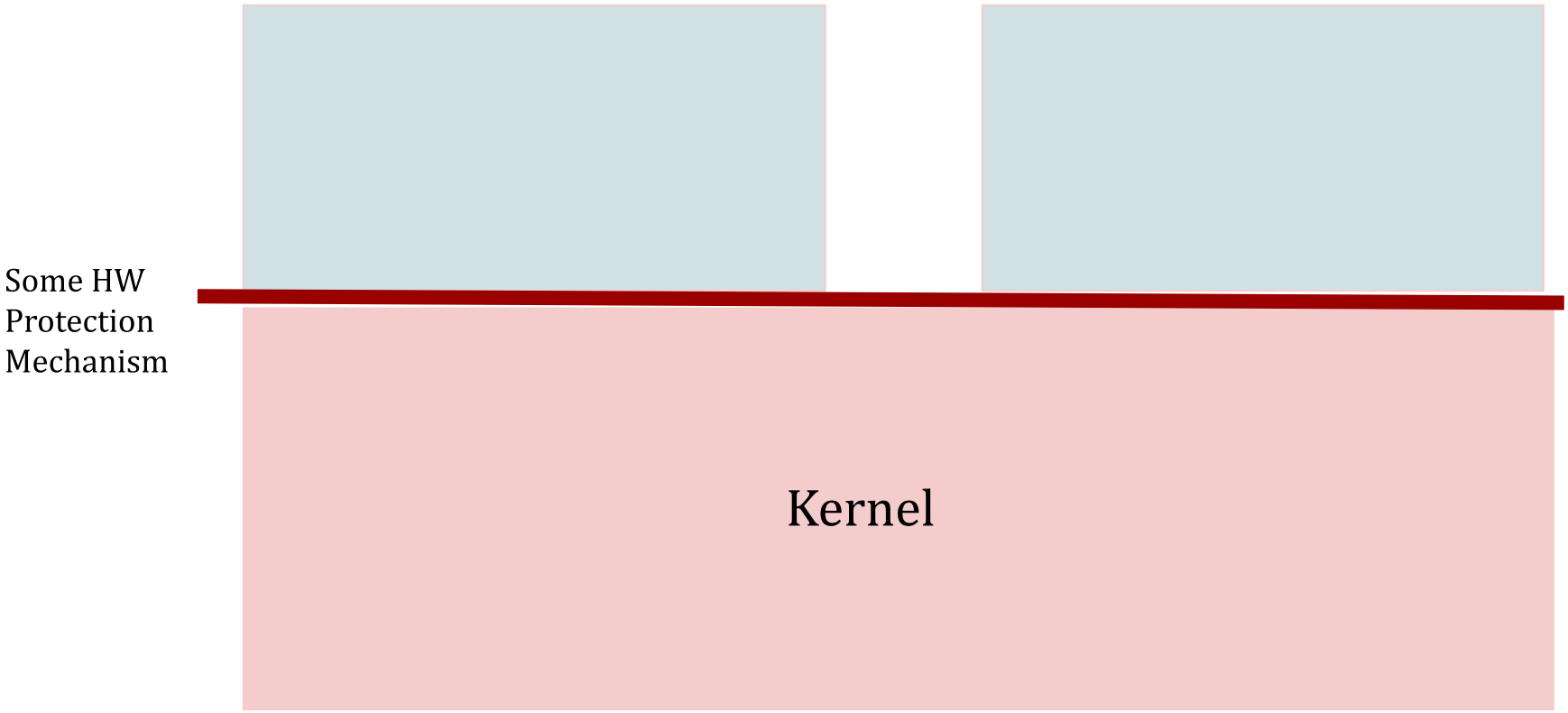
Lecture 3: System calls and everything you need for HW1

Common model today

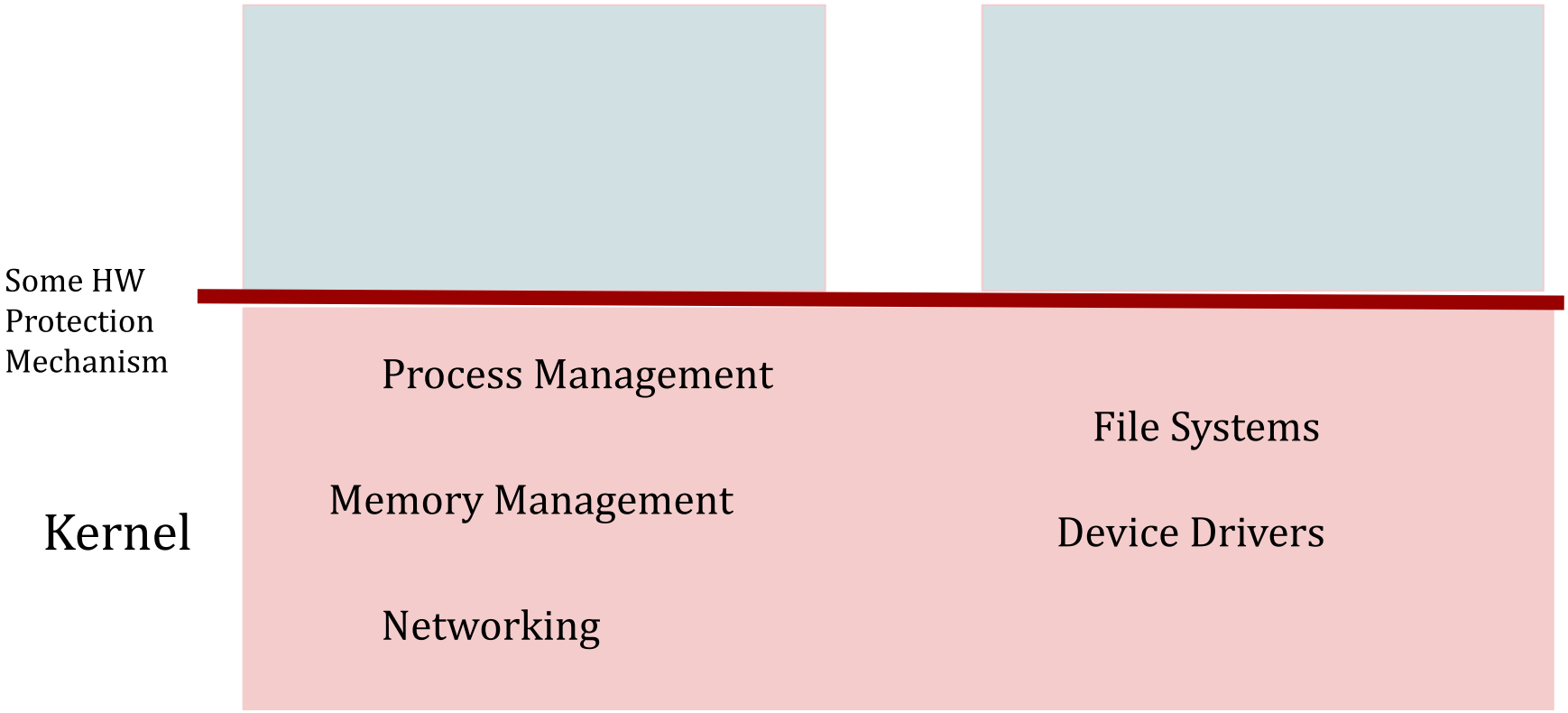



Untrusted Applications
Some HW
Protection
Mechanism
Trusted Operating System Code

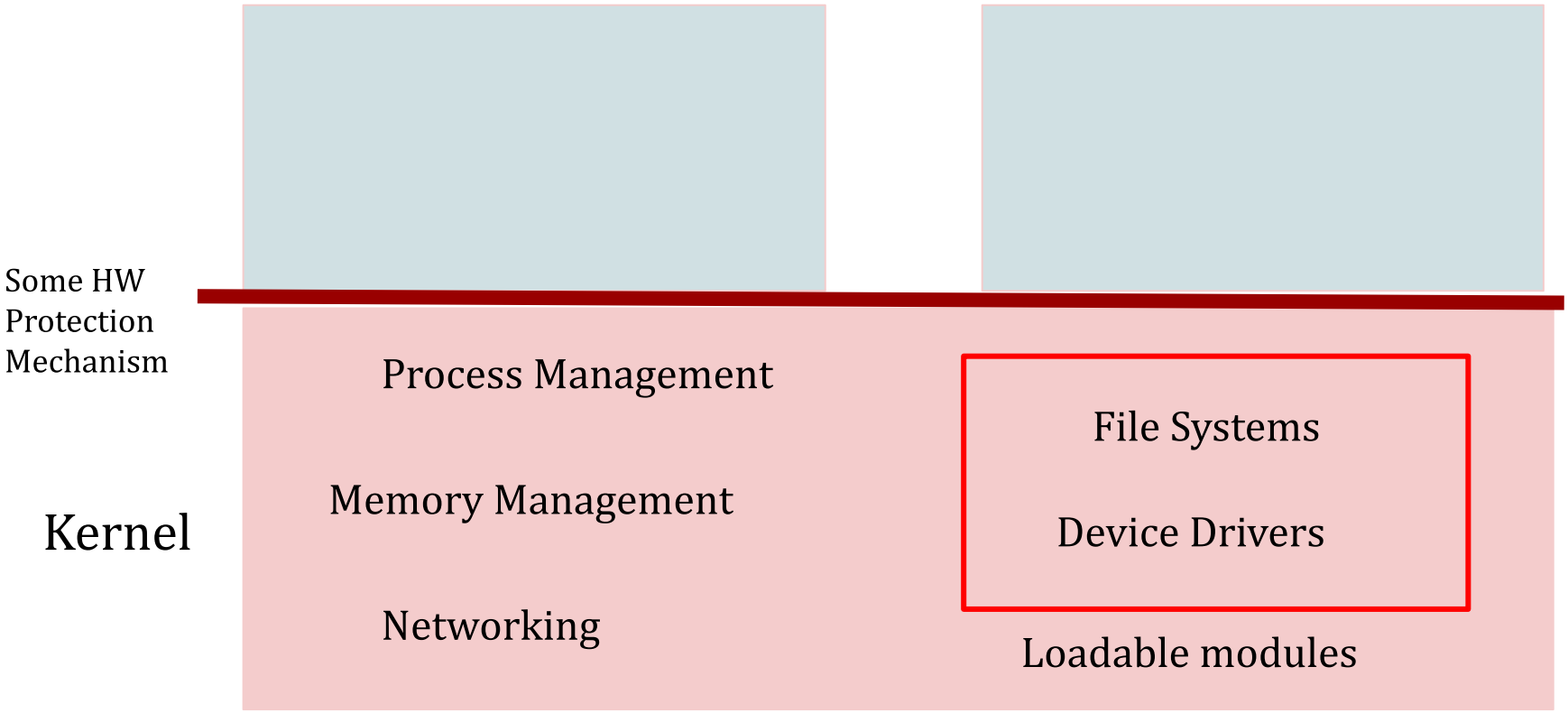
Common model today



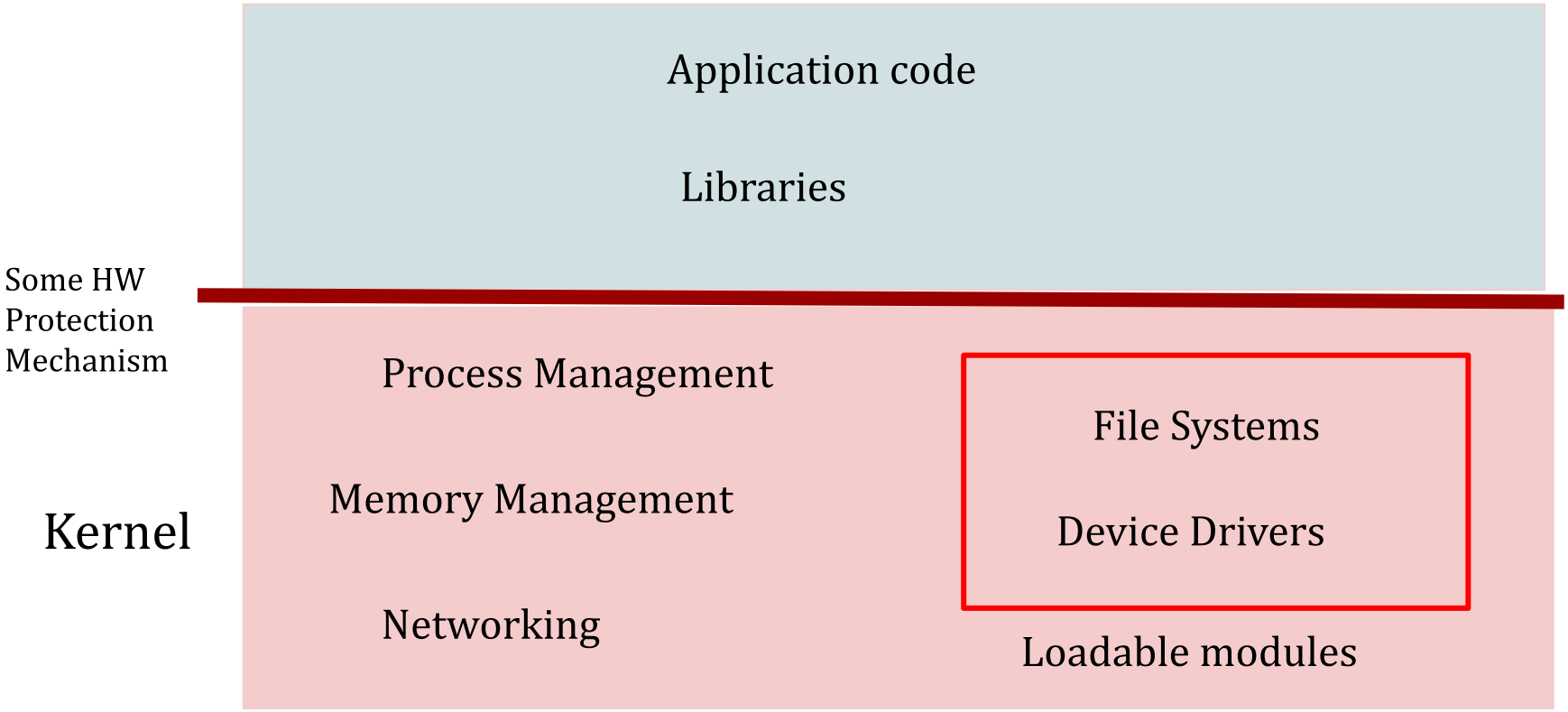
Monolithic OS in kernel



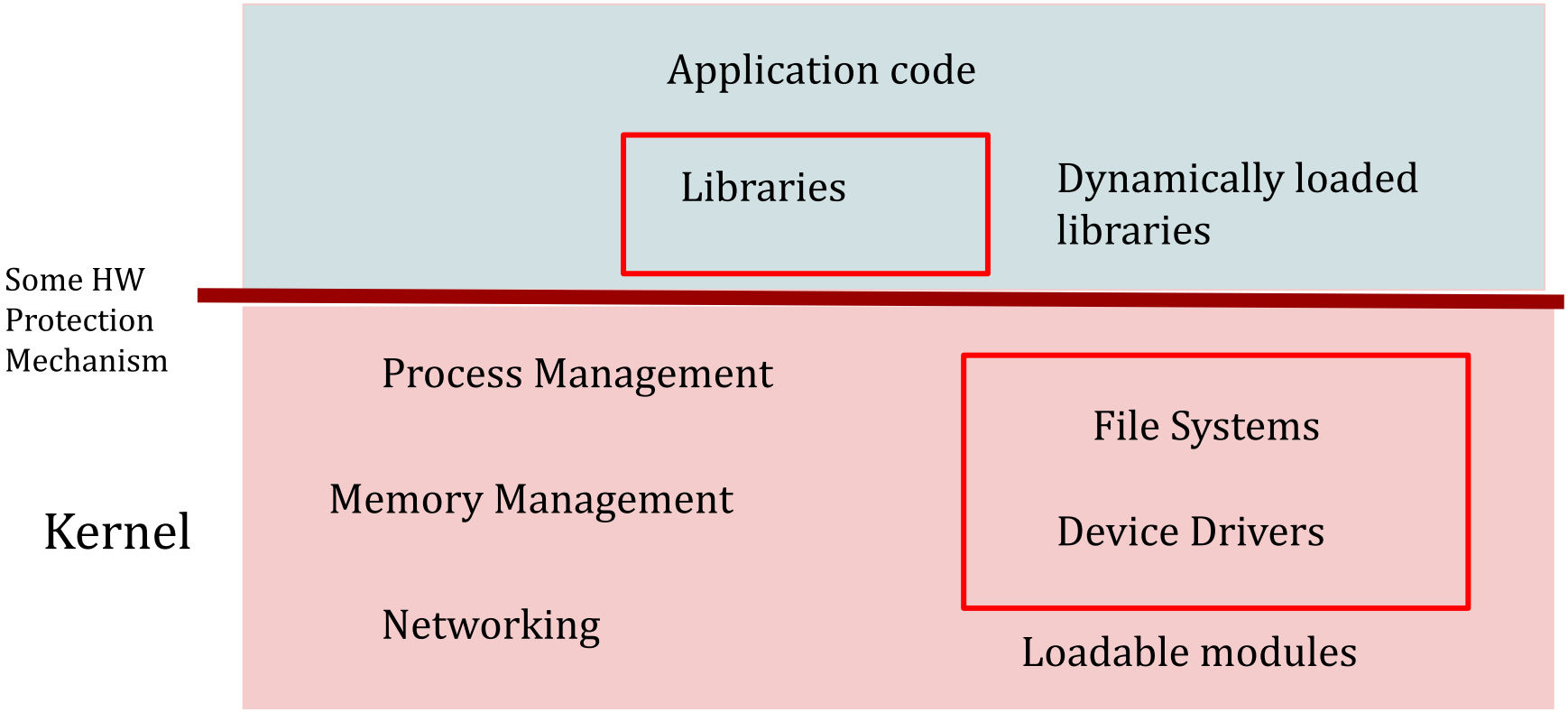
New functionality added dynamically



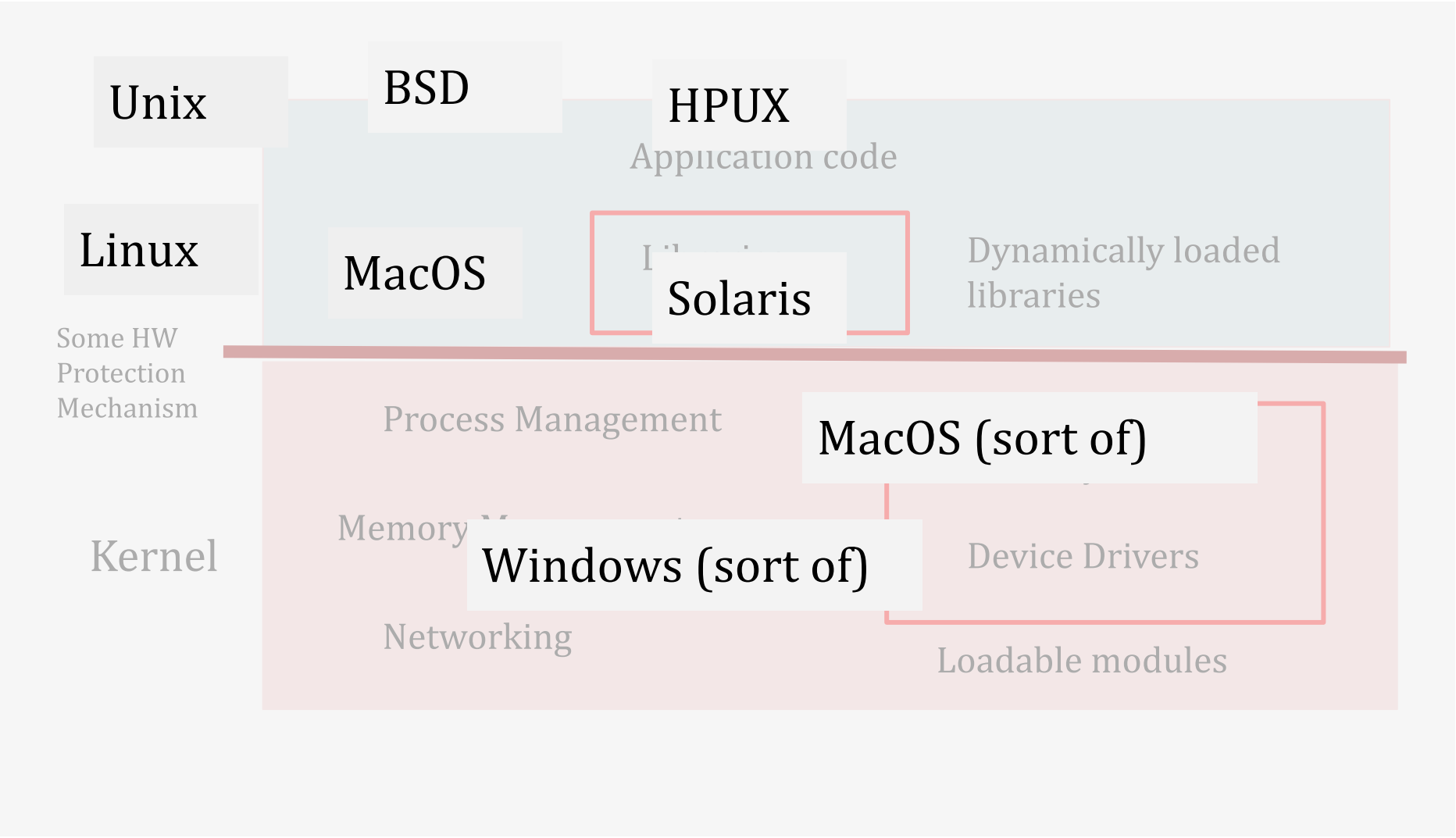
Library in application



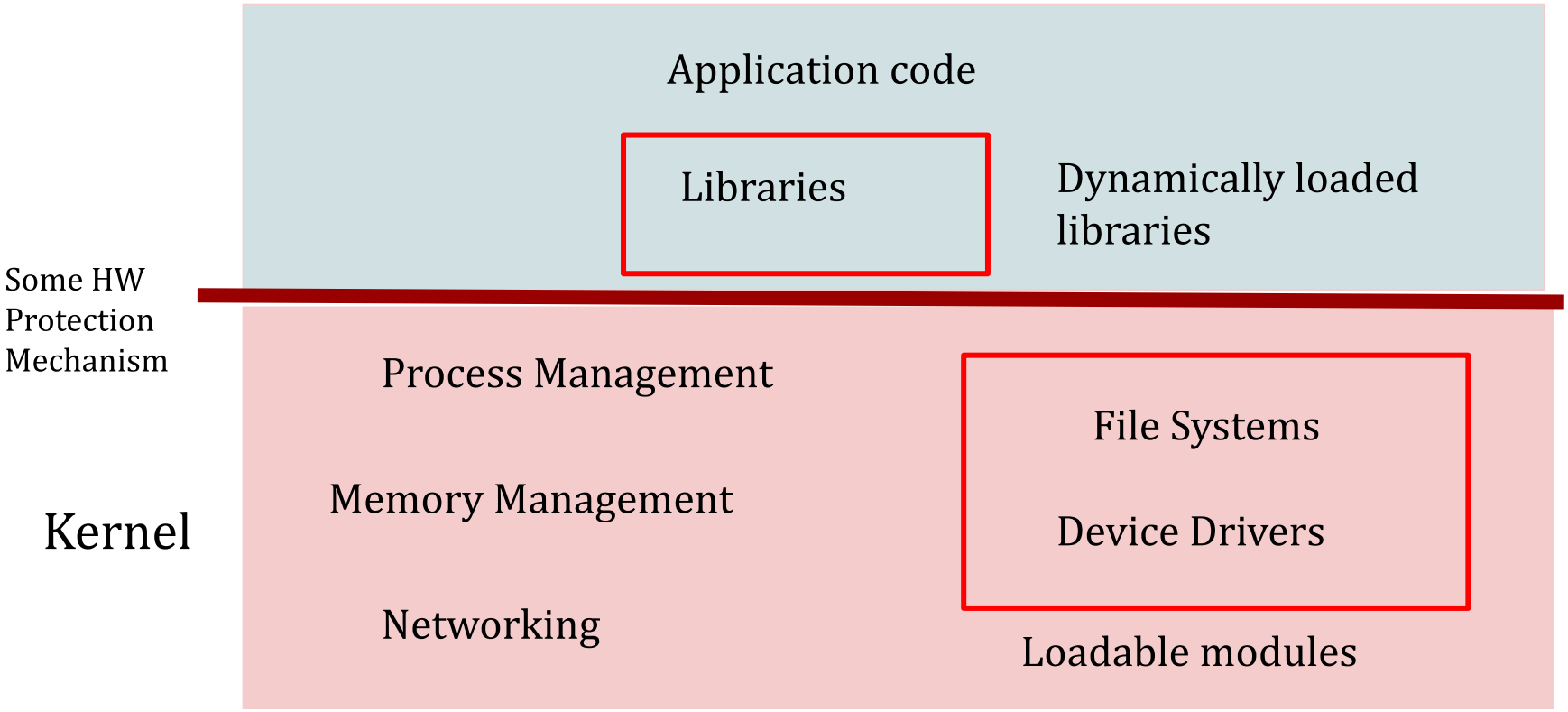
Typically dynamically loaded



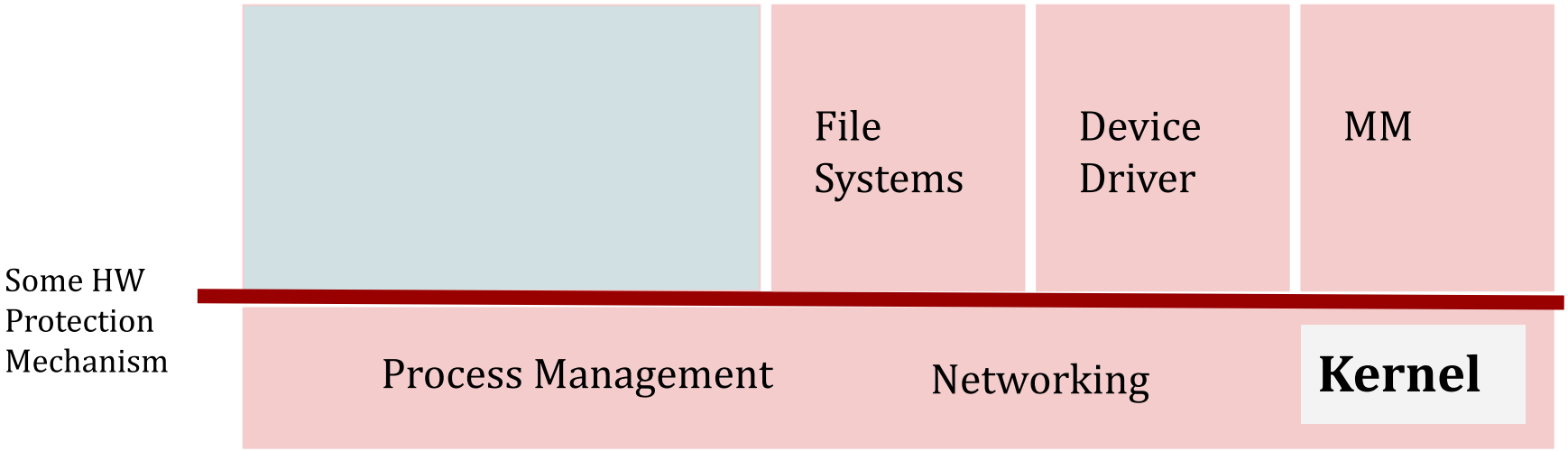
Examples


Unix
BSD
HPUX
Application code
Linux
MacOS
Solaris
Dynamically loaded libraries
Some HW Protection Mechanism
Process Management
MacOS (sort of)
Kernel
Memory
Windows (sort of)
Device Drivers
Networking
Loadable modules

There are other options...

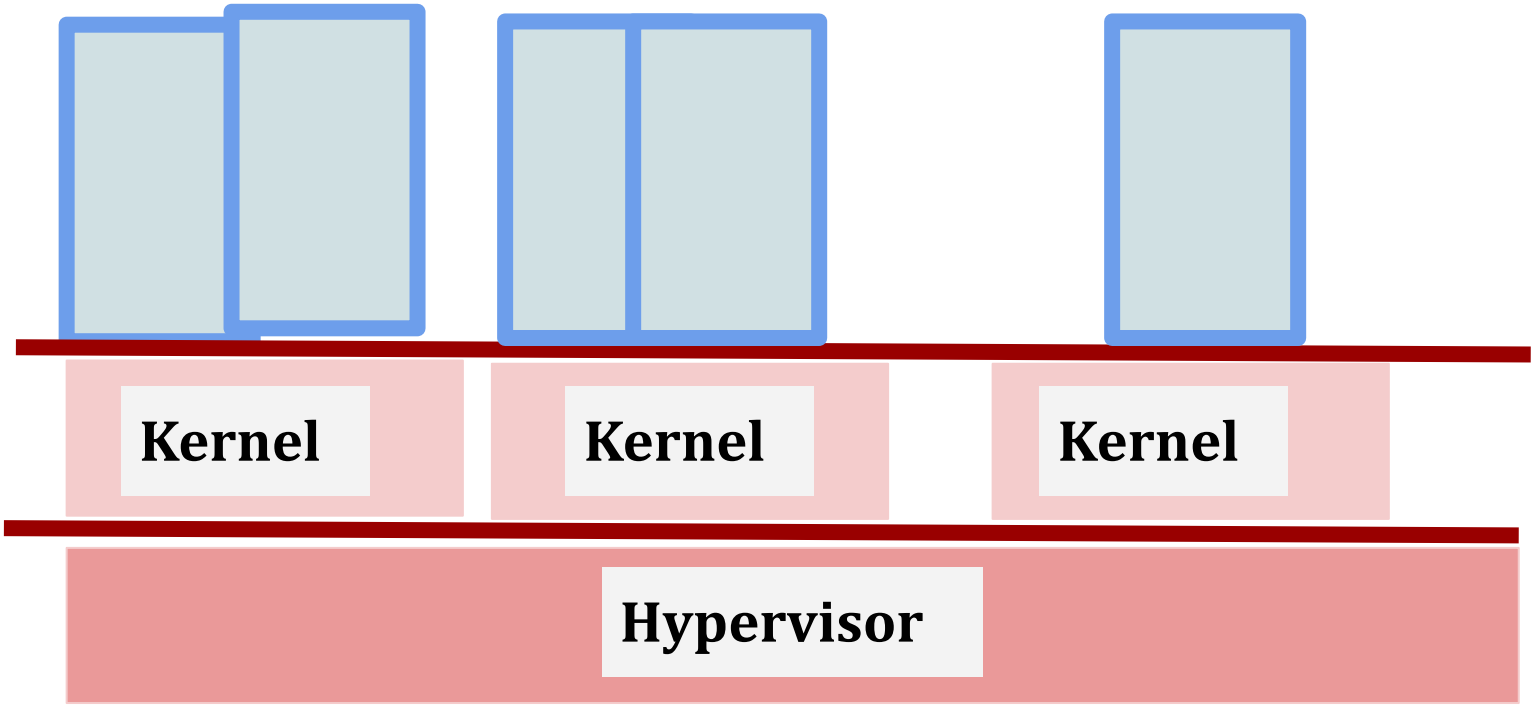


Microkernel

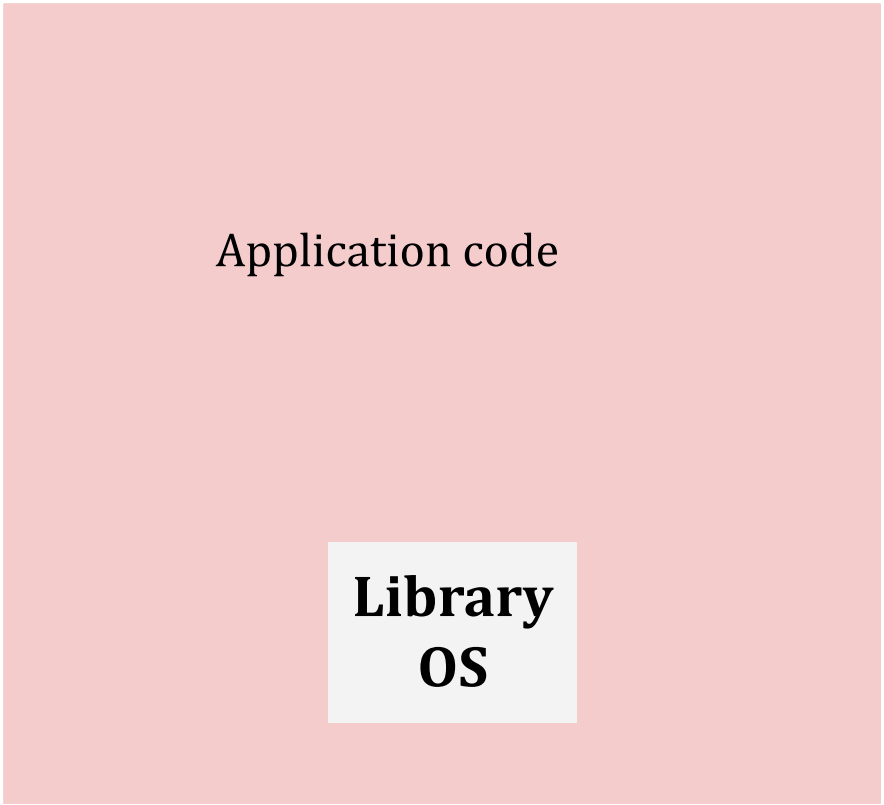


Virtualization

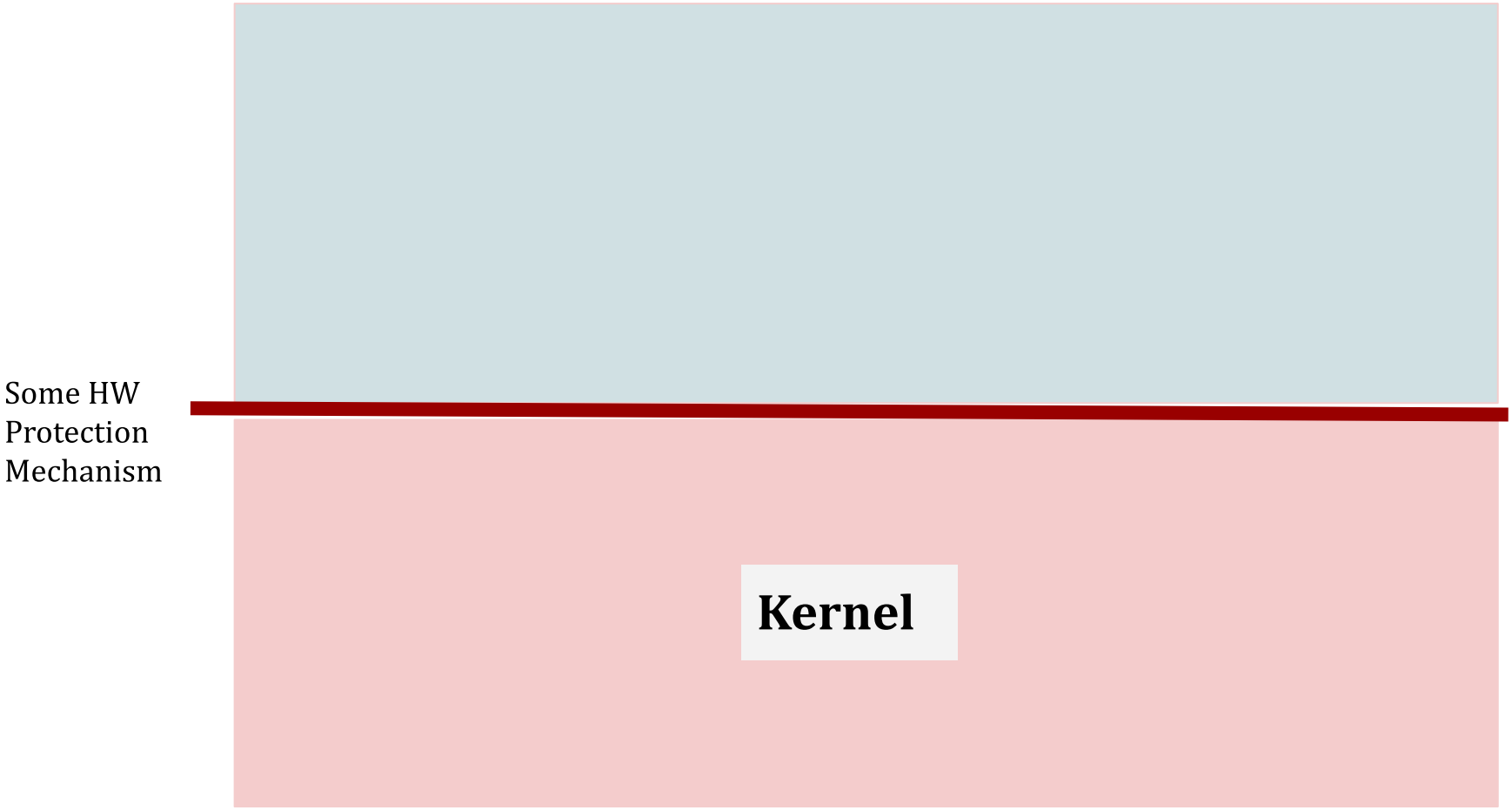
Some HW Protection Mechanism



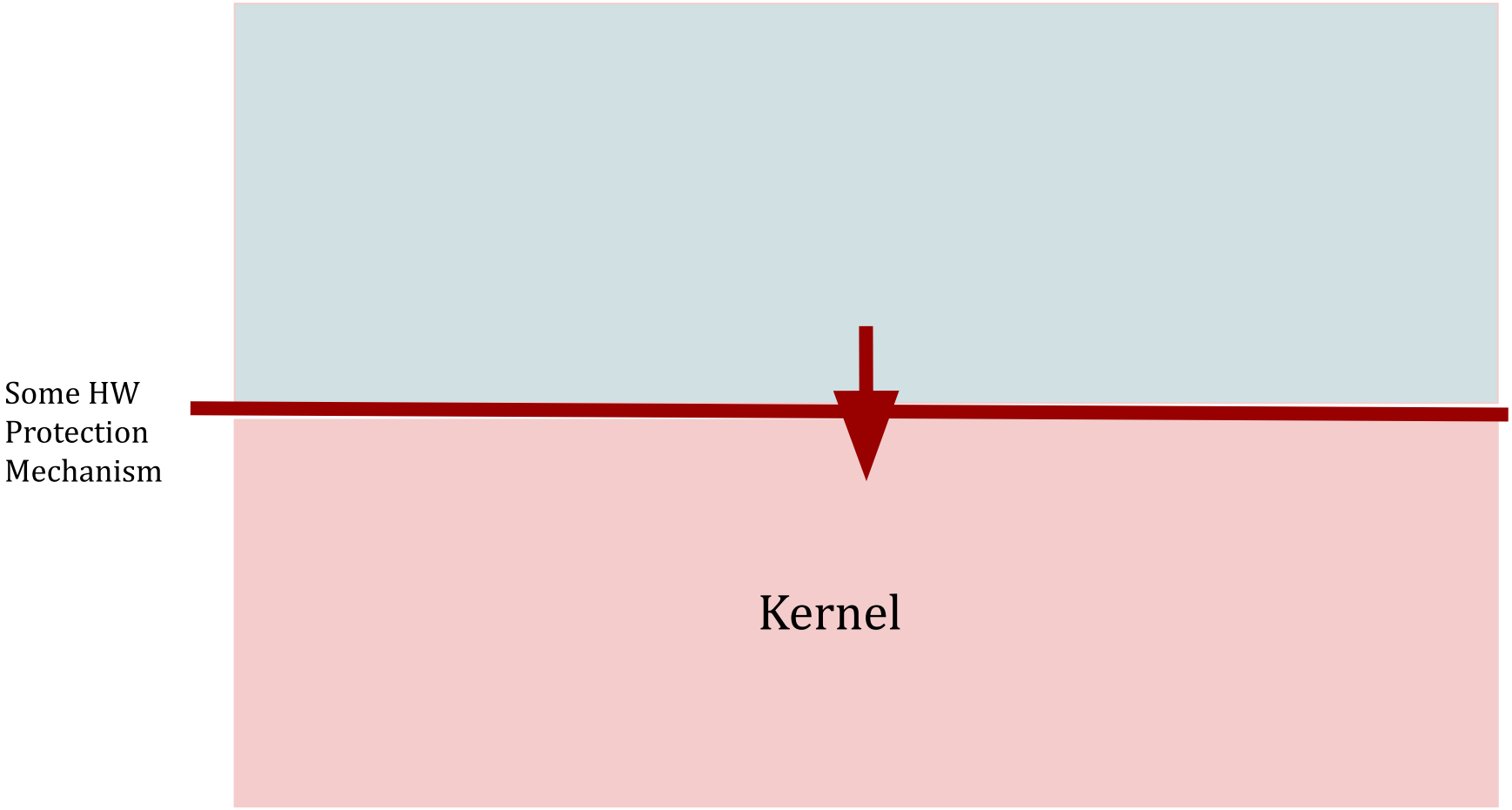
Unikernel/Library OS



For now, focus on this



System calls



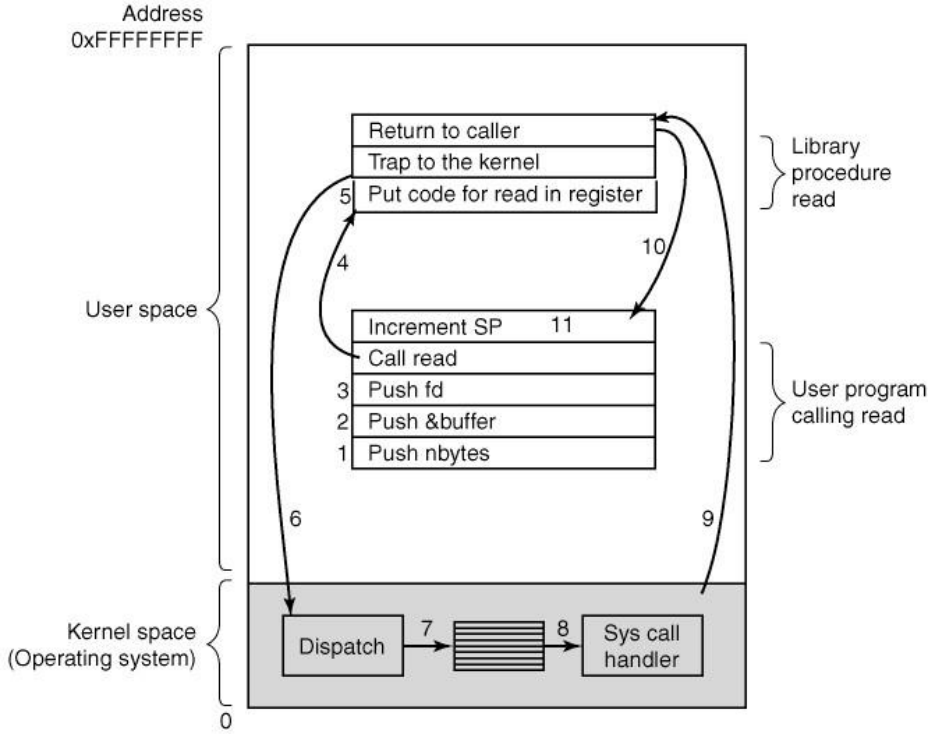
Overview

- System calls (definition and overview) needed for shell project
  - Processes and related system calls
  - Signals and related system calls
  - Files and related system calls

System Calls (aka syscalls)

System calls are the interface to operating system services - they are how we tell the OS to do something on our behalf

- API to the OS
- Hide OS and hardware complexity from us



System Call Interface

- Subroutine call/return that:
  - transfers control between user and kernel spaces
  - Switches between non-privileged and privileged CPU modes
  - switches stacks between user and kernel spaces
- Hardware & OS specific
- Special CPU instructions (hardware)
  - `SYSENTER/SYSCALL` (AMD64)
  - `int` (x86)

    0x80/Linux

    0x2e/Windows
  - `swi` (ARM)

System Call Arguments

Arguments (Hardware & OS specific)

- AMD64/Linux & *BSD
  - Syscall#: %rax
  - Args: %rdi, %rsi, %rdx, %r10, %r8, %r9
- x86/Linux
  - Syscall#: %eax
  - Args: %ebx, %ecx, %edx, %esi, %edi, %ebp
- x86/*BSD
  - Syscall#: %eax
  - Args: on the stack right-to-left

System Calls (Example)

```
#include <unistd.h>

int main(int argc, char* argv[])
{
  int fd, nread;
  char buf[1024];

  fd = open("my_file",0);    /* Open file for reading */
  nread = read(fd,buf,1024);     /* Read some data */

  /* Presumably we do something with data here */

  close(fd);
}
```

System Calls

How the system calls communicate back to us?

- Return value – usually return -1 on error, >= 0 on success
  - library functions set a global variable "errno" based on outcome
    - 0 on success,
    - positive values encode various kinds of errors
  - can use `perror` library function to get a string

- Buffers pointed to by system call arguments
  - e.g., in case of a read system call
  - values need to be copied between user and kernel space

Processes

Concept

- Virtual computer
- program in execution

Each process has its own virtual address space and execution state

Process table entry

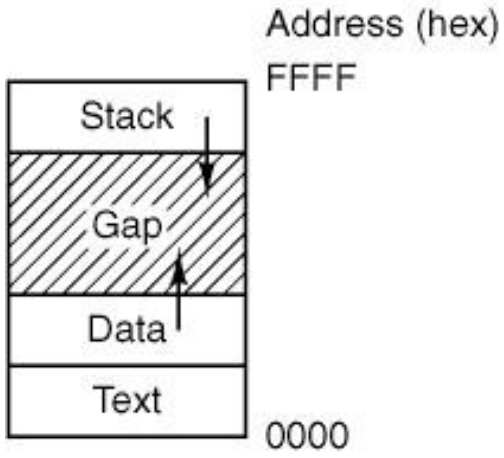
- creates the virtual address space
- stores all information associated with a process (except memory)
- register values, open files, user ID (UID), group ID (GID), ..

Processes are indexed by the process ID (PID)

- integer that indexes into process table

Processes

Memory layout



OS responsible for changing between multiple processes - Context Switching

Process System Calls

- **`fork`** (duplicate current process, create a new process)
- **`exec`** (replace currently running process with executable)
- **`exit`** (end process)
- **`wait`** (wait for a child process)
- **`getpid`** (get process PID)
- **`getpgrp`** (get process GID)

fork()

Syntax: `pid = fork();`

Get almost identical copy (child) of the original (parent)

- File descriptors, arguments, memory, stack … all copied
- Even current program counter
- But not completely identical - why?

Return value from `fork` call is different:

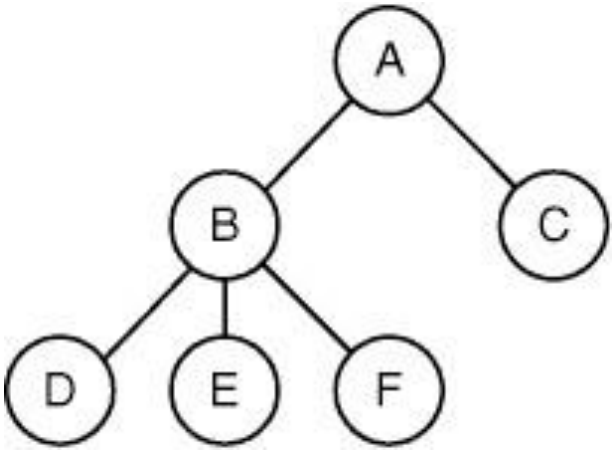
- 0 in child
- PID > 0 of the child when returning in parent

fork() cont.

```
int main(int argc, char* argv[])
{
   pid_t pid;
   if((pid = fork()) > 0)
   {
       /* Parent */
      printf("hello parent\n");
   } else {
      /* Child */
      printf("hello child\n");
   }
}
```

Process Hierarchy

- Notion of a hierarchy (tree) of processes
- Each process has a single parent - parent has special privileges
- In Unix, all user processes have 'init' as their ultimate ancestor



Additional ways to group processes

- Process Groups (job control)
- Sessions (all processes for a login)

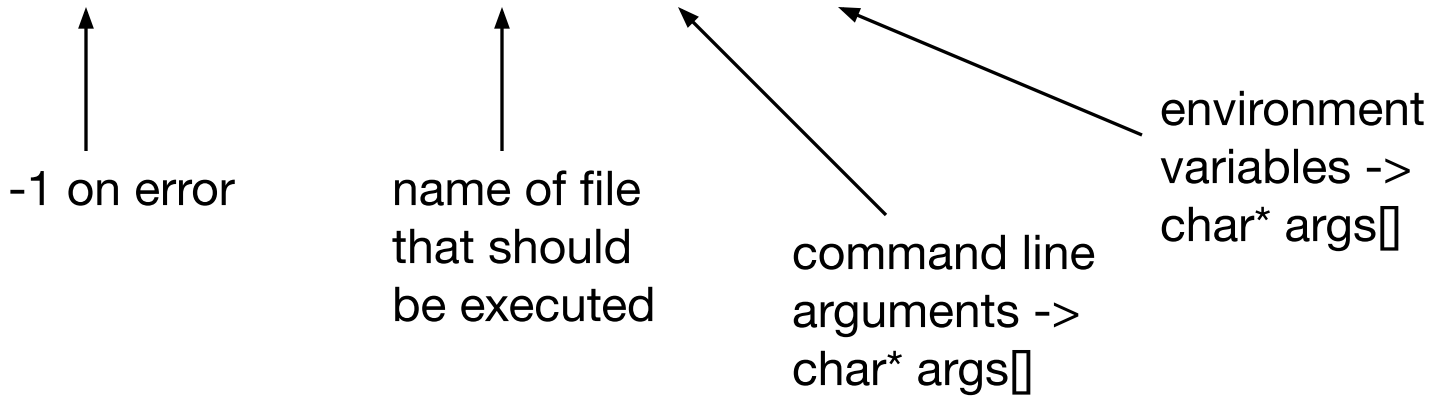
exec()

Change program in process

- i.e., launch a new program that **replaces** the current one

Several different forms with slightly different syntax

```
int status;
status = execve(prog, args, env);
```

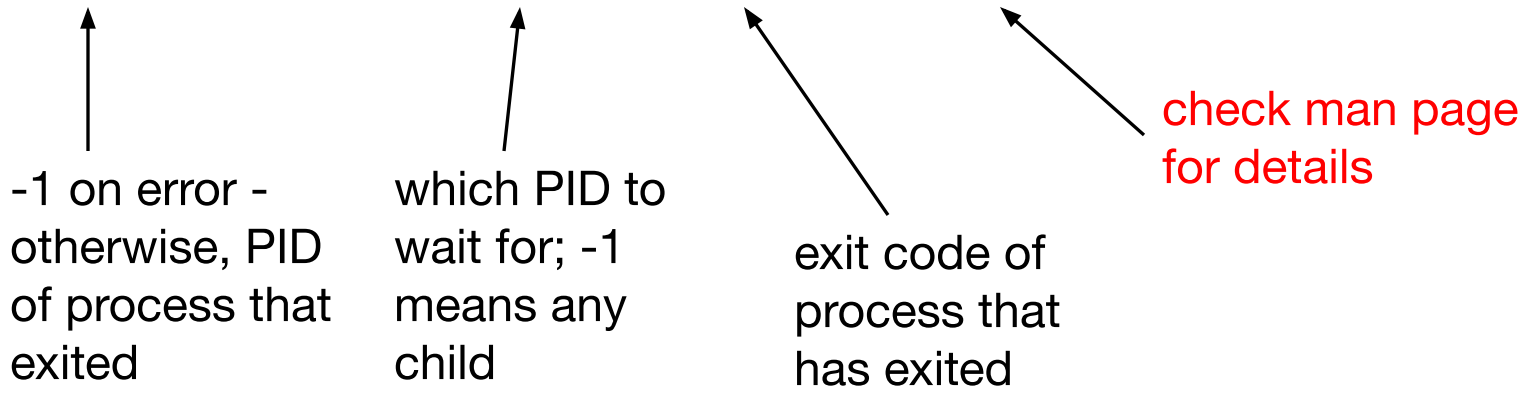


What does execve return on success?

wait()

- When a process is done it can call `exit(status)`
- This is the status that "`echo $?`" can show you in the shell
- Parent can wait for its children (block until they are done)

```
status = waitpid(pid, &statloc, options);
```



Example

```
int main(int argc, char* argv[])
{
    pid_t pid;
    int status;
    char* ls_args[2];
    ls_args[0] = ".";
    ls_args[1] = 0;
    if((pid = fork()) > 0)
    {
        /* Parent */
       waitpid(pid,&status,0);
       exit(status);
    } else {
       /* Child */
       execve("/bin/ls", ls_args,0);
    }
}
```

What will happen & why?

Shell

- Program that makes heavy use of basic process system calls
- Basic cycle:
  1. prompt
  2. read line
  3. parse line
  4. fork (child execs the command, parent waits)

Shell

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
      type_prompt( );                           /* display prompt on the screen */
      read_command(command, parameters);        /* read input from terminal */

      if (fork( ) != 0) {                       /* fork off child process */
           /* Parent code. */
           waitpid(–1, &status, 0);             /* wait for child to exit */
      } else {
           /* Child code. */
           execve(command, parameters, 0);      /* execute command */
      }
}
```

But we still don't know

- pipes (i.e., |)
- re-direction (i.e., <, >)
- background processes (&) and reaping zombies

Signals

Report events to processes in asynchronous fashion

- process stops current execution (saves context)
- invokes signal handler
- resumes previous execution

Examples

- user interrupts process (terminate process with CTRL-C)
- timer expires
- illegal memory access
- child exit

Signal handling

- signals can be ignored
- signals can be mapped to a signal handler (all except SIGKILL)
- signals can lead to forced process termination

Signal System Calls

- **`kill`** (send signal to process)
- **`alarm`**(set a timer)
- **`pause`**(suspend until signal is received)
- **`sigaction`**(map signal handler to signal)*
- **`sigprocmask`**(examine or change signal mask)
- **`sigpending`**(get list of pending signals that are blocked)

sigaction()

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
    struct sigaction *oldact);

struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

Signal number

function pointer to the new action

Signals (man 7 signal)

```
Signal      Value      Action   Comment
SIGHUP        1          Term    Hangup detected on controlling
                                  terminal or death of controlling
                                  process
SIGINT        2          Term    Interrupt from keyboard
SIGQUIT       3          Core    Quit from keyboard
SIGILL        4          Core    Illegal Instruction
...
SIGKILL       9          Term    Kill signal
SIGSEGV      11          Core    Invalid memory reference
SIGPIPE      13          Term    Broken pipe: write to pipe with no
                                  readers; see pipe(7)
SIGALRM      14          Term    Timer signal from alarm(2)
...
SIGCHLD    20,17,18      Ign     Child stopped or terminated
...
```

Very relevant for hw1

You'll see this a lot throughout all homeworks

Files & related system calls

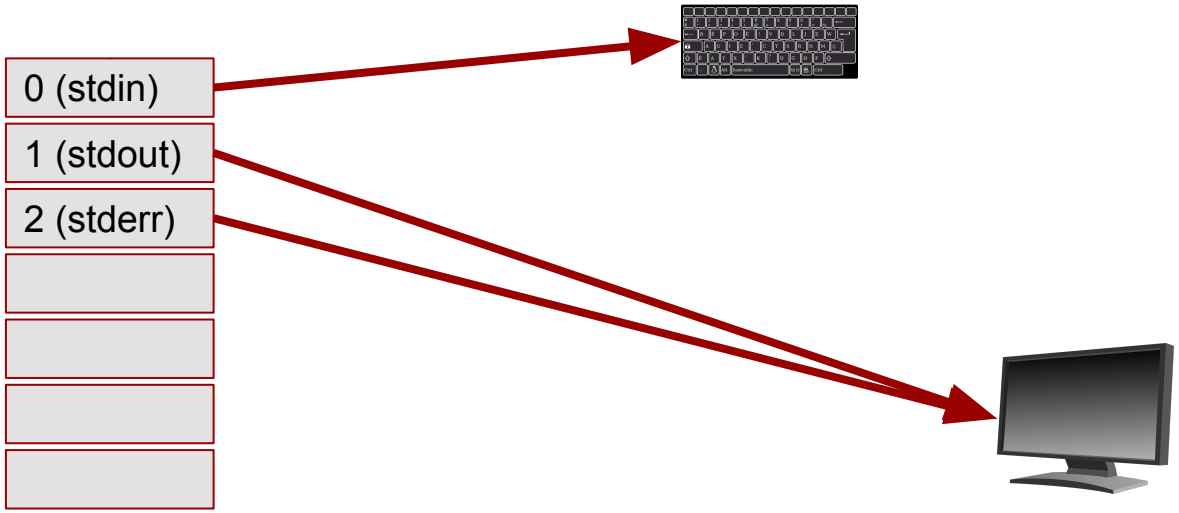
Files

- Conceptually, each file is an array of bytes
- Special files
  - directories
  - block special files (disk)
  - character special files (modem, printer)
- Every process(and therefore running program) has a table of open files (file table)
- File descriptors are integers which index into this table
- Returned by **`open`**, **`creat`**
- Used in **`read`**, **`write`**, etc. to specify which file we mean

Files

- Initially, every process starts out with a few open file descriptors
  - 0 - stdin
  - 1 - stdout
  - 2 - stderr
- We have a file pointer, which marks where we are currently up to in each file (kept in an OS file table)
- File pointer starts at the beginning of the file, but gets moved around as we **`read`** or **`write`** (or can move it ourselves with **`lseek`** system call)

A bit on File descriptors

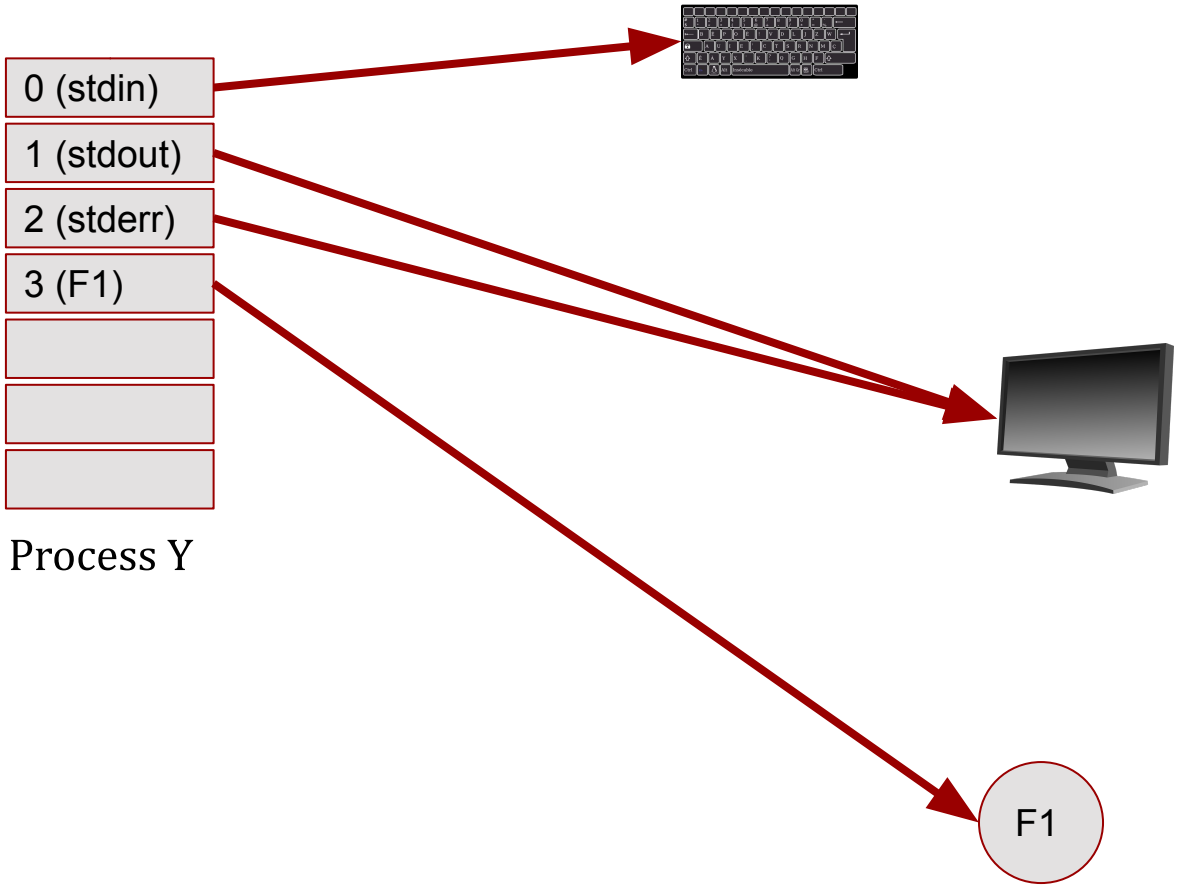


Process Y

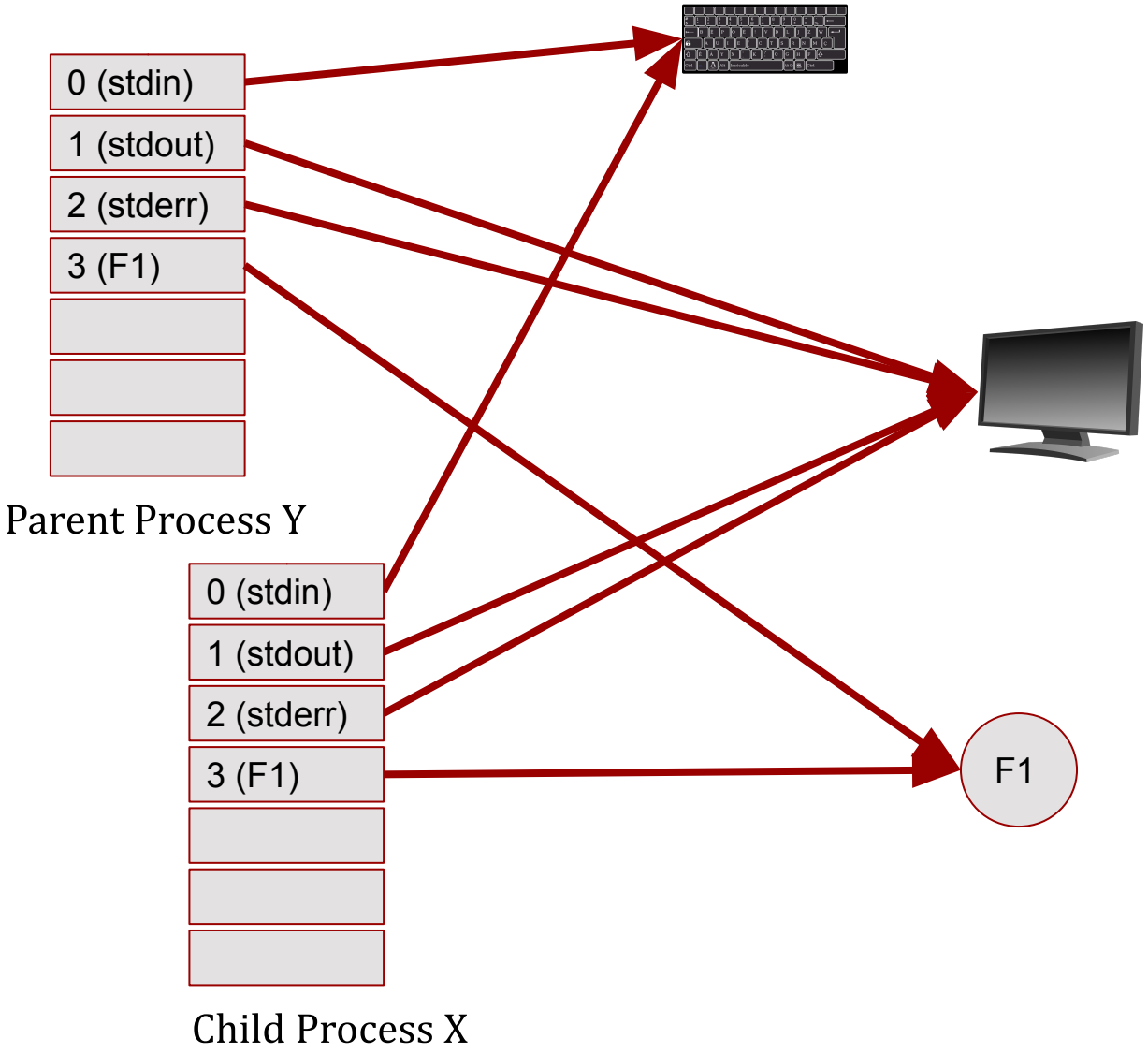
File System System Calls

- **`open`** (open a file)
- **`close`** (close a file)
- **`creat`** (create a file)
- **`read`** (read from file)
- **`write`** (write from file)
- **`chown`** (change owner)
- **`chmod`** (change permission bits)
- **`pipe`** (create FIFO)
- **`dup/dup2`** (duplicate open file descriptor)

File descriptors after open(F1)



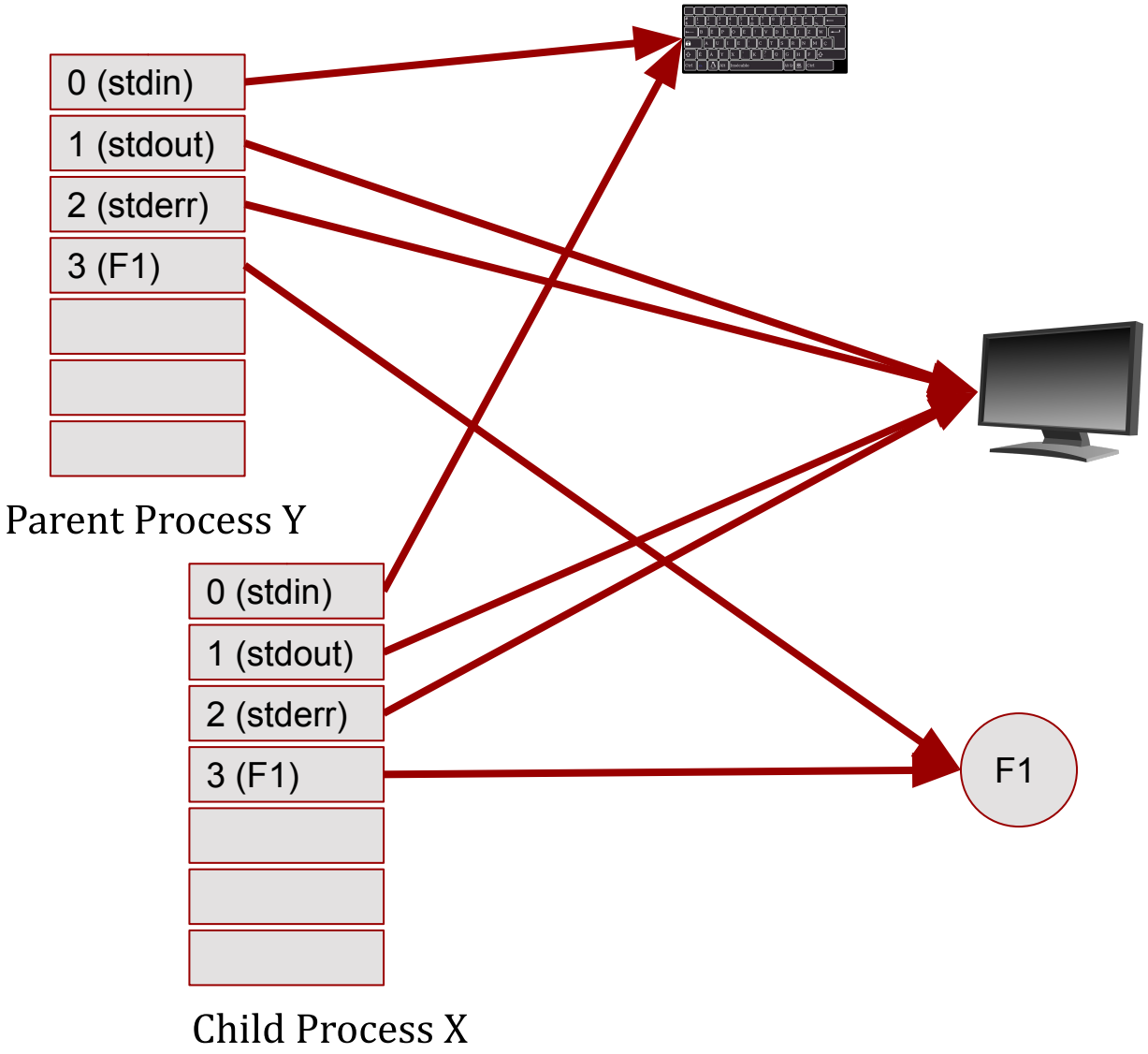
File descriptors after fork()



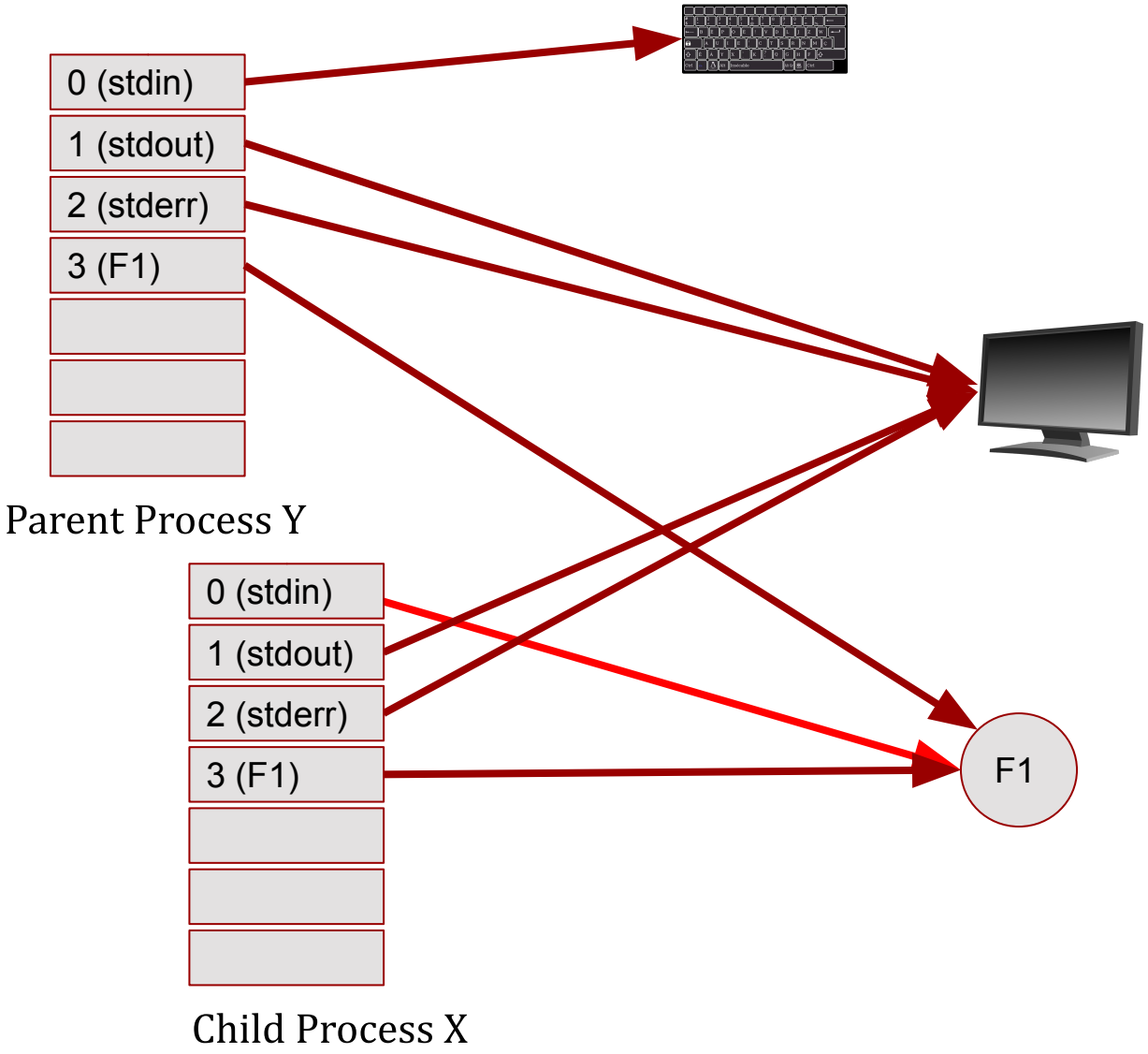
dup/dup2 System Call

- Read dup man page - “man dup”
- Used for shell pipe “|” and shell redirection “<” and “>”
- dup(int oldfd) - create a copy of oldfd in lowest unused file descriptor
  - e.g., close(STDIN) then dup(oldfd) will cause read(STDIN) to be same as read(oldfd)
  - e.g., close(STDOUT) then dup(oldfd) will cause write(STDOUT) to be same as write(oldfd)
- dup2(int oldfd, int newfd) - create a copy of oldfd in the newfd file descriptor
  - e.g., dup2(oldfd, STDIN) will cause read(STDIN) to be same as read(oldfd)
  - e.g., dup2(oldfd, STDOUT) will cause write(STDOUT) to be same as write(oldfd)

Remember this



You could use dup for re-direction



Pipes

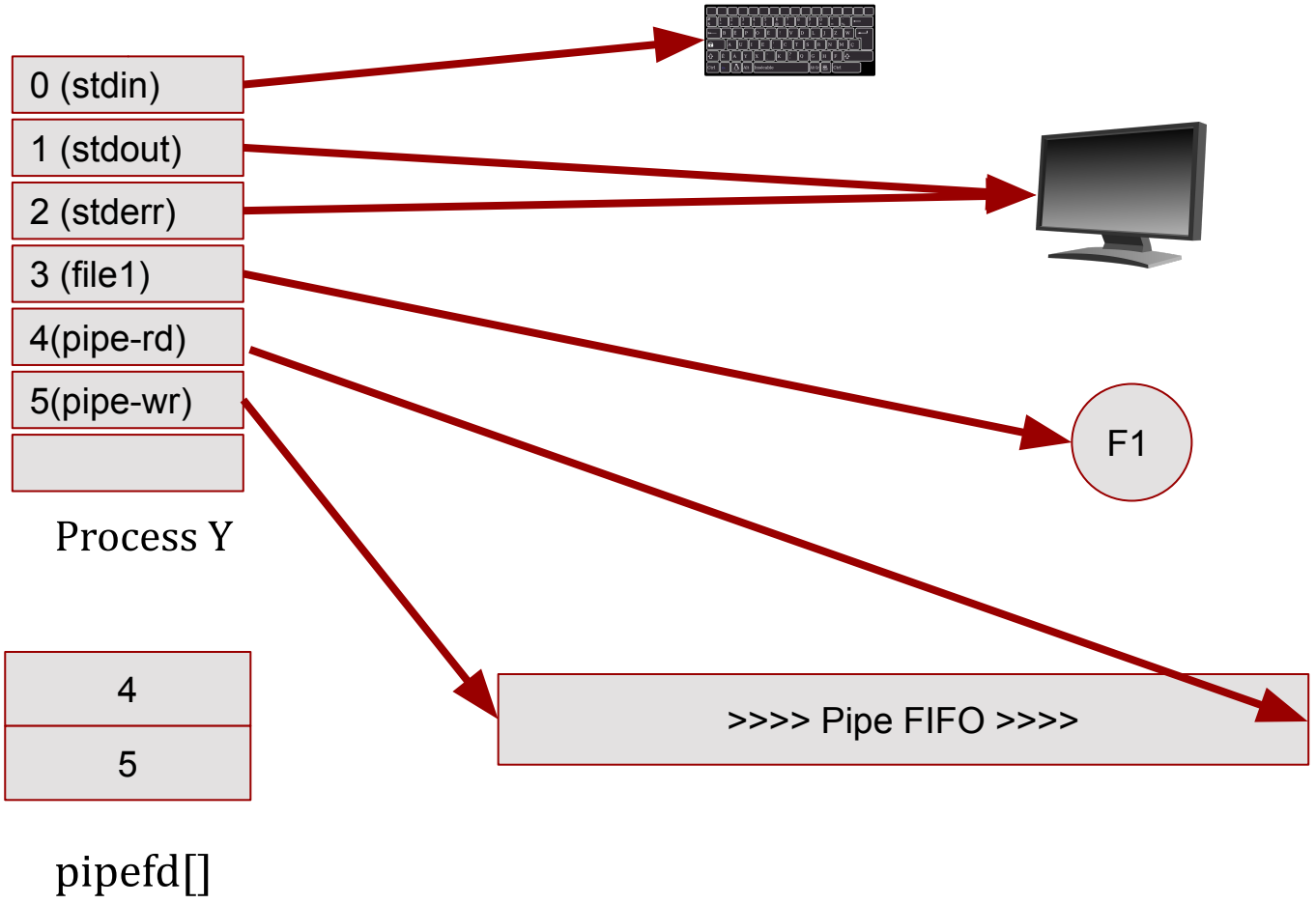
- Common Unix mechanism for processes to communicate with one another (FIFO)
- Pipes are basically special files
- Implemented as circular buffer of fixed size (e.g., 4k)
- Communication through read and write system calls
- Block if reading an empty pipe or writing a full one
- Use at shell level (ls | wc, who | sort | lpr)

Pipe System Call

- Create a pipe:
  - need array of size 2 integers.
  - array[0] is FD for reading, array[1] is FD for writing.

```
int pipefd[2];          /* FD's for pipe. */



pipe(pipefd);           /* create pipe */
...
read(pipefd[0], ...);     /* read from pipe */
...
write(pipefd[1], ...);    /* write to pipe */
```

File descriptors after pipe



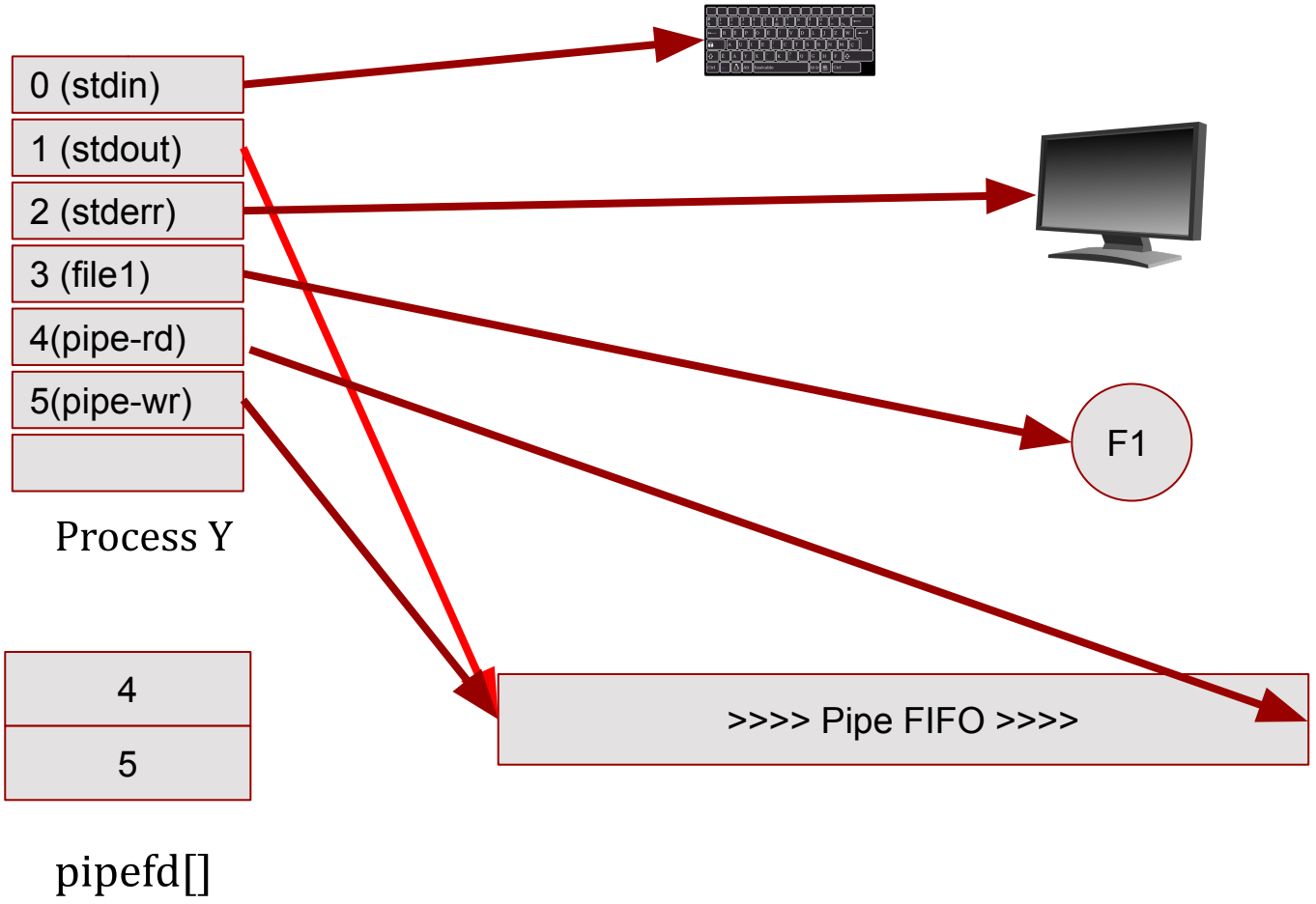
Pipe System Call

- But talking to ourselves is no fun/meaningless.
- Need someone else to talk to – how?
    - Solution - create pipe, then fork
- But, how do we connect the programs?
    - Solution - use dup to tie stdin/stdout to input/output pipe

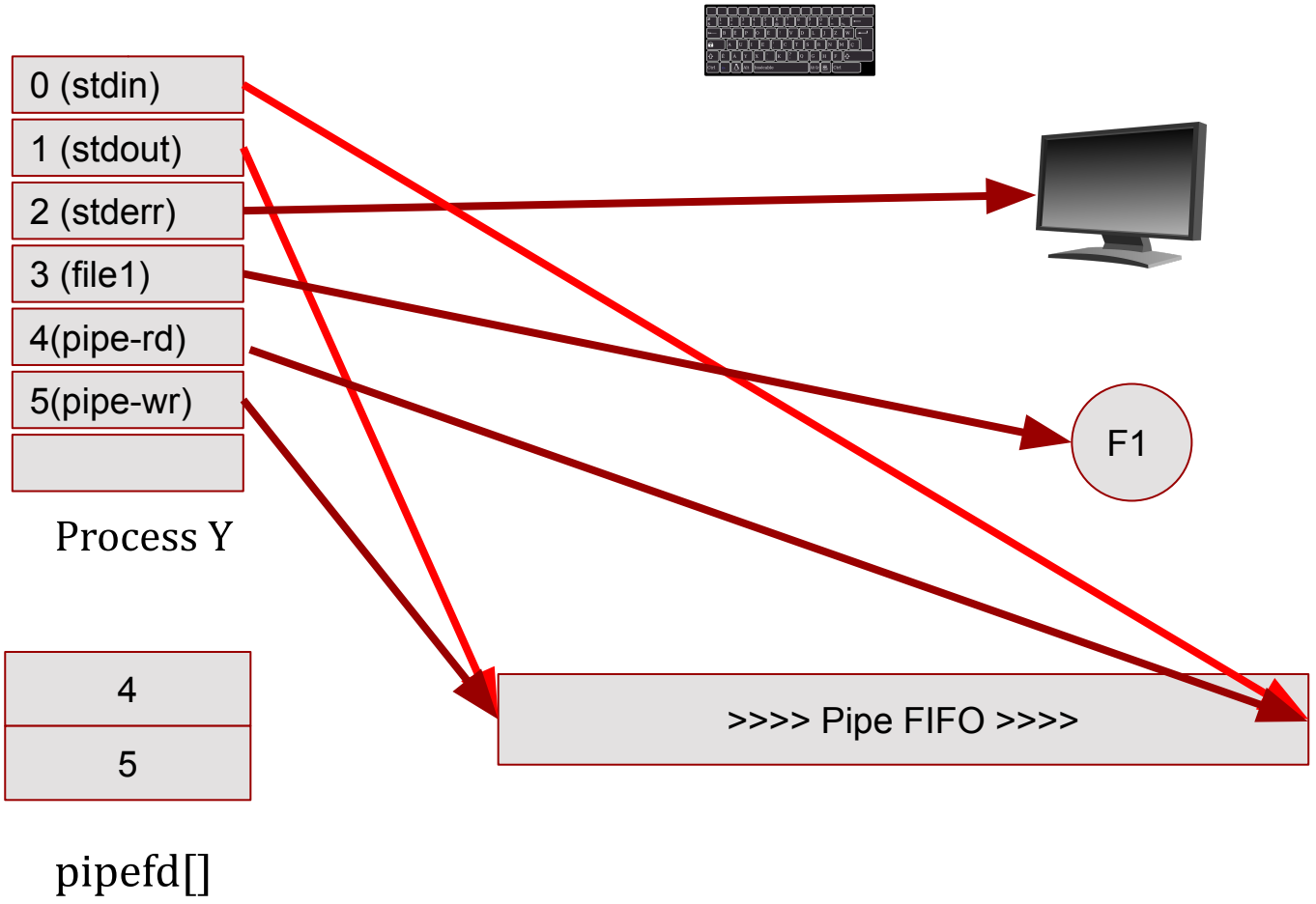
dup/dup2 System Call

- dup(int oldfd) - create a copy of oldfd in lowest unused file descriptor
  - close stdin then dup(oldfd) will cause stdin to read from pipe
  - close stdout then dup(oldfd) will cause stdout to write to pipe
- dup2(int oldfd, int newfd) - create a copy of oldfd in the newfd file descriptor
  - dup2(oldfd, STDIN) will cause stdin to read from pipe
  - dup2(oldfd, STDOUT) will cause stdout to write to pipe

For example, producer could use dup2



File descriptors after dup/dup2



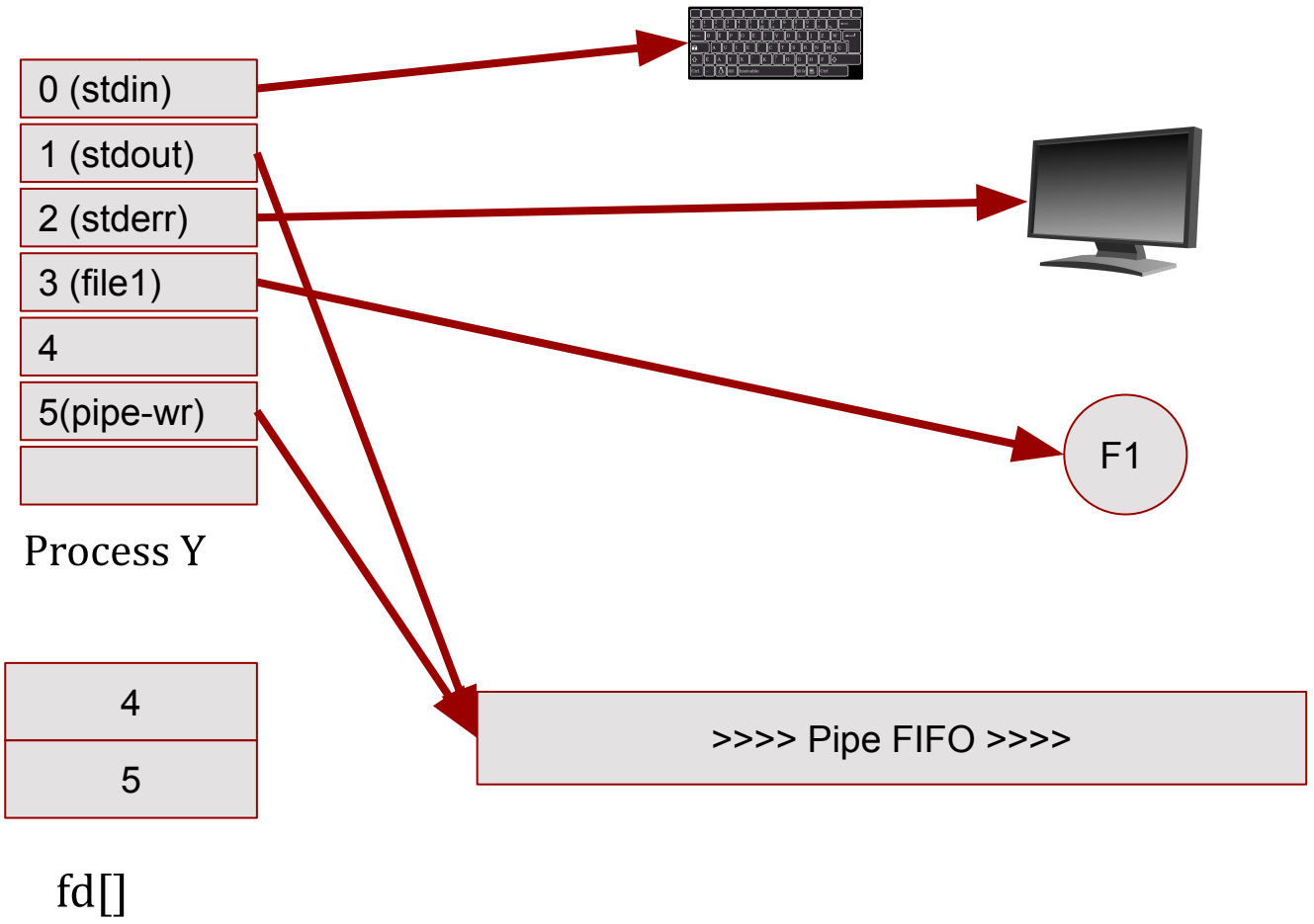
Inter-process Communication(parent)

```
#define STD_INPUT 0                     /* file descriptor for standard input */
#define STD_OUTPUT 1                    /* file descriptor for standard output */

pipeline(process1, process2)
char *process1, *process2;              /* pointers to program names */
{
  int fd[2];

  pipe(&fd[0]);                         /* create a pipe */
  if (fork() != 0) {
        /* The parent process executes these statements. */
        close(fd[0]);                   /* process 1 does not need to read from pipe */
        close(STD_OUTPUT);              /* prepare for new standard output */
        dup(fd[1]);                     /* set standard output to fd[1] */
        close(fd[1]);                   /* this file descriptor not needed any more */
        execl(process1, process1, 0);
  } else {
```

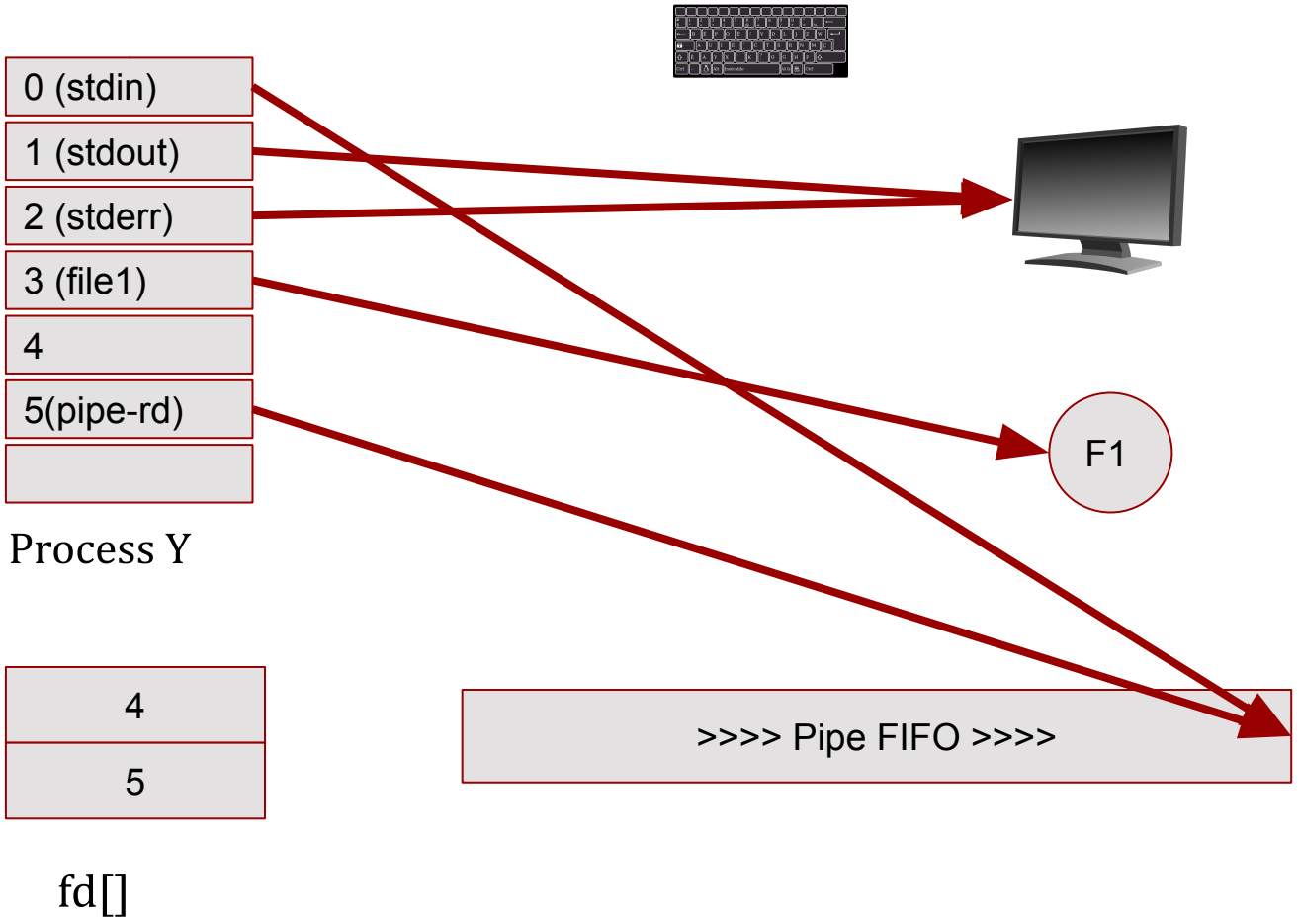
Inter-process Communication(child)

```
        /* The child process executes these statements. */
        close(fd[1]);                   /* process 2 does not need to write to pipe */
        close(STD_INPUT);               /* prepare for new standard input */
        dup(fd[0]);                     /* set standard input to fd[0] */
        close(fd[0]);                   /* this file descriptor not needed any more */
        execl(process2, process2, 0);
    }
}
```

Parent's file descriptors after dup



Childs's file descriptors after dup



Homework 1 Discussion

Remember this... Shell

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
      type_prompt( );                           /* display prompt on the screen */
      read_command(command, parameters);        /* read input from terminal */
                                                (Convenient place to use your parser)
      if (fork( ) != 0) {                       /* fork off child process */
           /* Parent code. */
           waitpid(–1, &status, 0);             /* wait for child to exit */
      } else {
           /* Child code. */
           execve(command, parameters, 0);      /* execute command */
      }
}
```

HW1 Write a simple shell that handles some special characters …

- *command* < *filename*
  - instead of reading from `stdin`, read from *filename*
- *command* > *filename*
  - instead of writing to `stdout`, write to *filename*
- *cmd1* | *cmd2*
  - pipe `stdout` from *cmd1* as `stdin` to *cmd2*
- *command* &
  - run command in background and don't wait until it completes before printing (and processing) the next prompt.

HW1: Full set of required commands

Full set of required commands

What is the full set of required commands for the shell hw? Am I required to add a function for tab to do autocomplete? What about cd? Or is the minimum requirement ls, ps, cat, '<', '>', '|', and '&'? Also, which flags are required for each command?

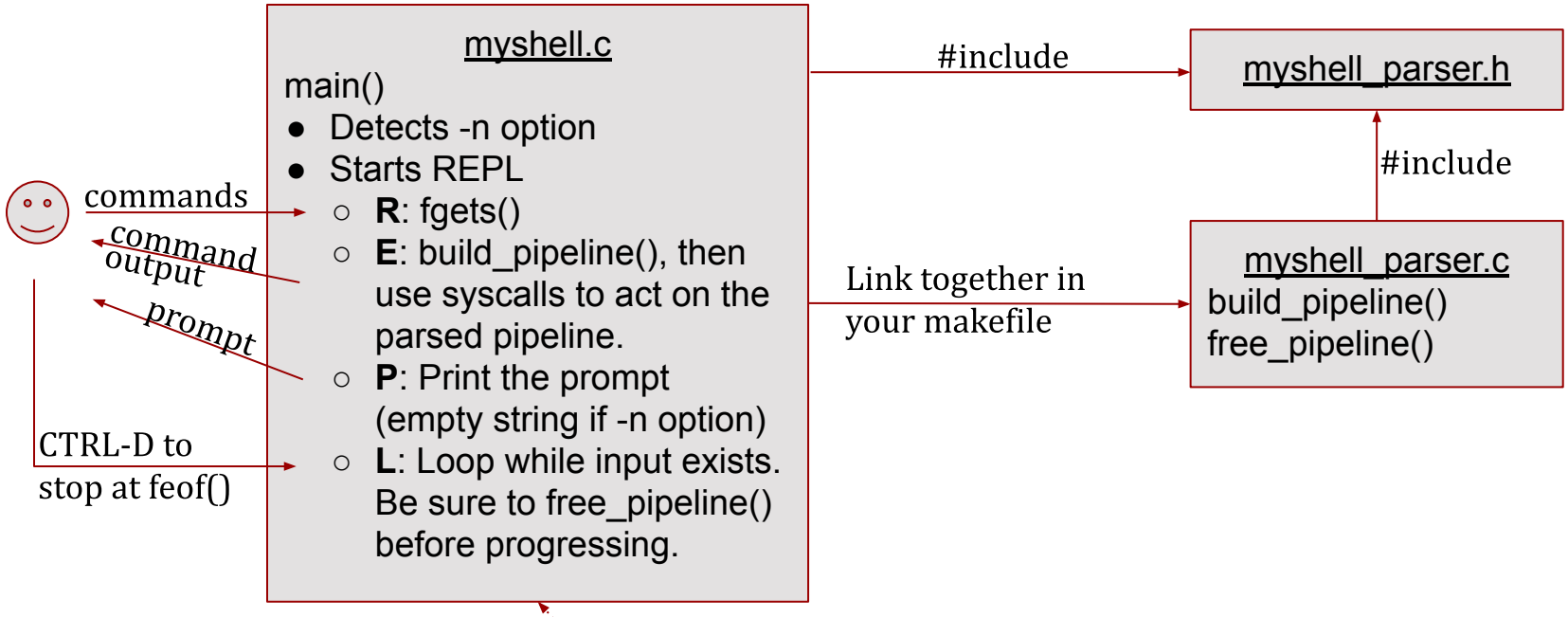
hw1 (Relevant question from a previous semester)

- It's important to keep separate functionality of the shell (e.g., `<>|&`) and regular programs (`/bin/ls`, `/bin/cat`, etc.)
- You have to implement the functionality of the shell, and *your shell should invoke programs* as specified by the user.

Homework 1 Questions

- The assignment outlines expected inputs and how to handle them
- Today's lecture covers
  - syscalls: fork, exec, pipe, dup, etc.
  - processes: hierarchy, reaping
- Questions?
  - Assignment description
  - Autograder
  - Challenges and difficulties
  - Personal test suites

One *Suggested* Design



Note: This is a lot of responsibility for main()
Code will be easier to understand if you divide the work across smaller functions