CS 344: Operating Systems I

02.13: Part III – Signals and Pipes

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

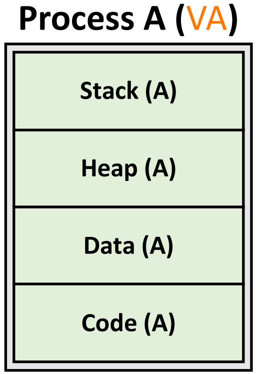
SAIL
Secure AI Systems Lab

Notice

- Announcements
  - Sanghyun is back
  - Sanghyun’s office hours will be on the 16th at 11:00 am to 12:30 pm
    - No office hours on the 17th

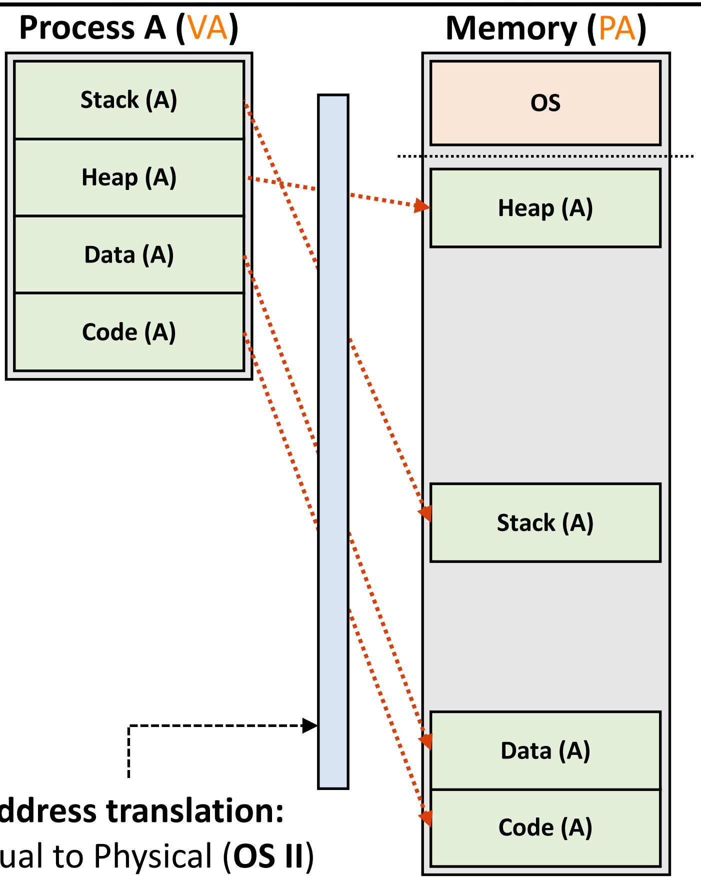
Recap: process isolation

- Process segments
  - Code segment
  - Data segment
  - Heap segment
  - Stack segment



Recap: process isolation

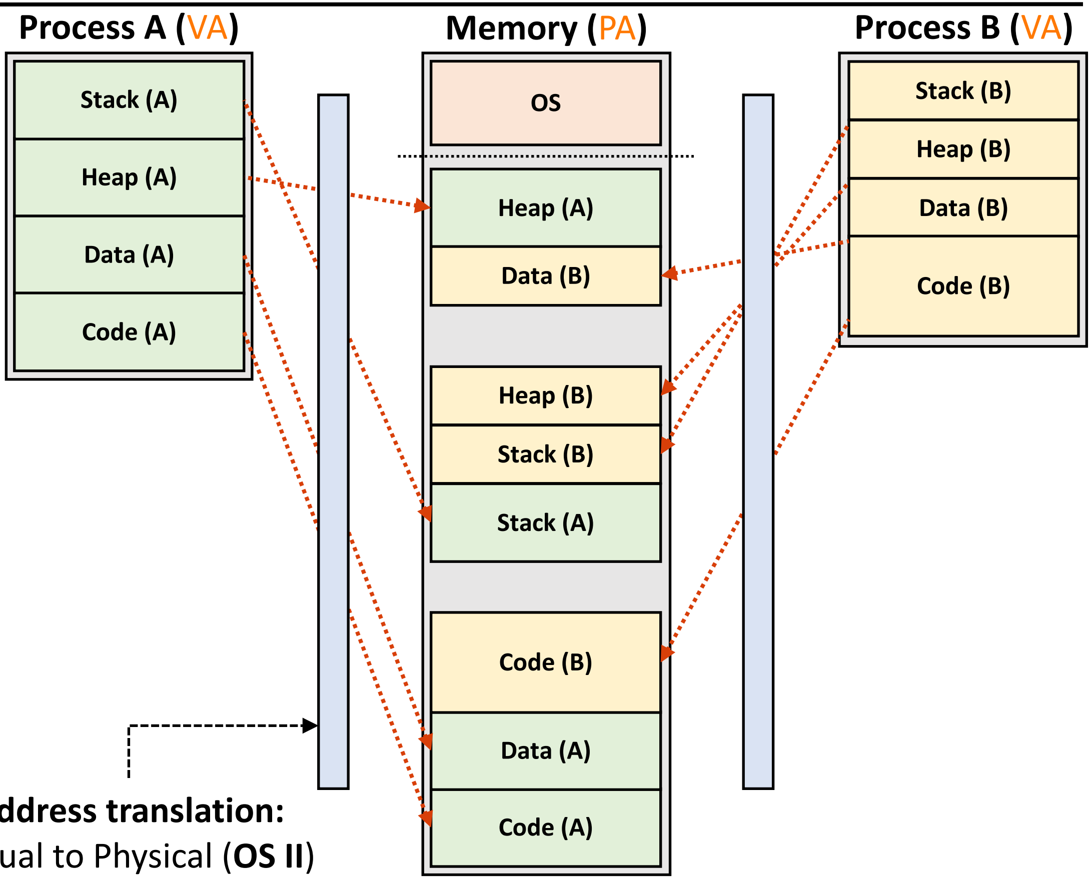
- Process segments
  - Code segment
  - Data segment
  - Heap segment
  - Stack segment



Address translation:
Virtual to Physical (**OS II**)

Recap: process isolation

- Process segments
  - Code segment
  - Data segment
  - Heap segment
  - Stack segment



Address translation:
Virtual to Physical (**OS II**)

Recap: Process Isolation

- Process **isolation**
  - **Definition:** Prevent Process A from reading/writing to Process B
  - Why?
    - Security reasons (e.g., data breach, system crash, …)
    - Management reasons (e.g., easy to control, …)
  - What happens if we access the other process' memory
    - **Segmentation fault**
  - What's the downside?
    - Processes can't talk to each other

Recap: process isolation

- Processes talked to each other a lot:
  - Example scenario A:
    - You’re a YouTuber
    - You’re editing a video with Adobe products
    - You ask the other program (not Adobe) to convert the video format
    - How can OS let the other program know the filename that Adobe uses?
  - Example scenario B:
    - You chat with your friends on Signal app.
    - Your app (process) on your phone needs to share what you type with others
    - How can OS let the remote program know what you type?

Strawman solutions

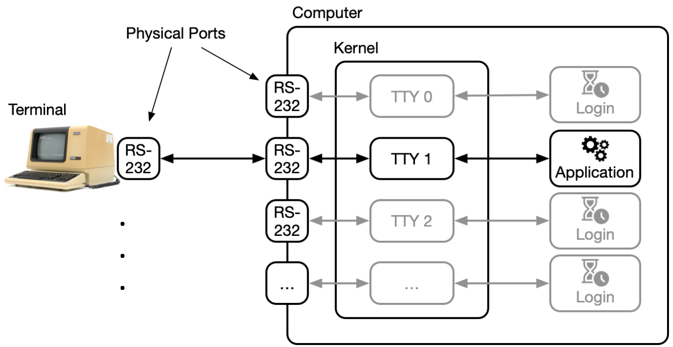
- **Hole punching (Link)!**
  - **Definition:**
    - (from computer networking)
    - A technique that allows two or more parties to communicate directly each other
  - **Downside:**
    - Potentially ignore the security mechanisms (e.g., firewalls)
    - Potentially increase overheads to manage such connections separately
    - …

Topics for today

- Part III: IPC, RPC, and Networking
  - Motivation
    - What is IPC/RPC?
    - Why do we need IPC/RPC?
  - Provide abstractions
    - What is the mechanisms OS support for IPC?
  - Offer standard interface
    - How can we use a signal?
    - How can we use a pipe?
  - Manage resources
    - (Overview) How does OS support these mechanisms?

Provide abstraction: signals

- **Background** (in 1960-70s)
  - Terminals are connected to a (huge) computer
  - You use terminal to control multiple processes
  - You want to kill a process; how would you do?

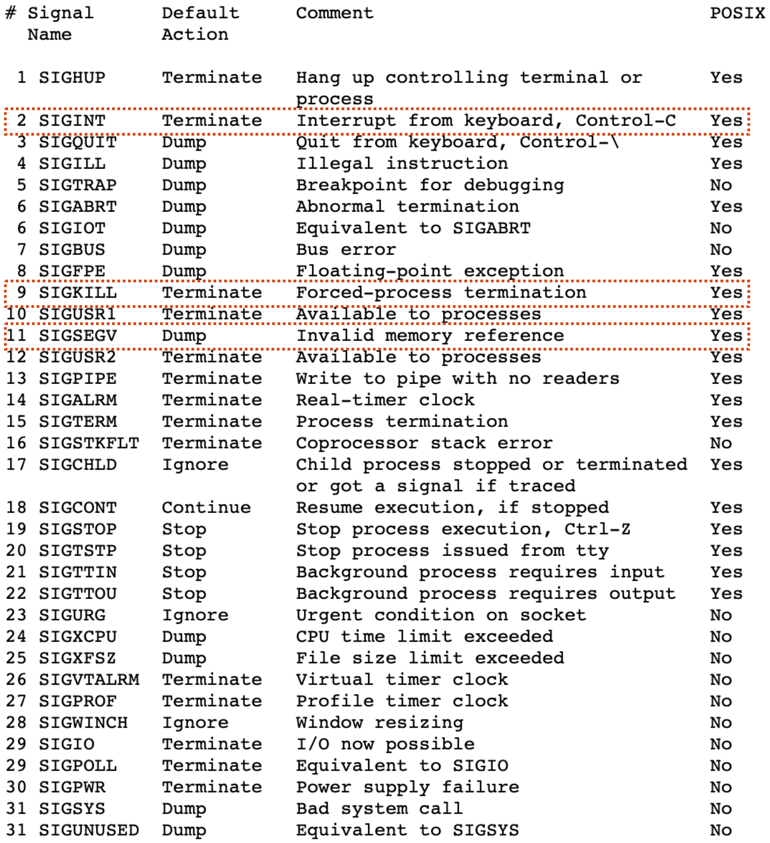


- **OS support "signals"**
  - **Definition:**
    - (Formal) an asynchronous mechanism to notify an event to a process
    - (Informal) **notifications** between processes or a process and a thread

Provide abstraction: signal types

- **Signals in Linux**
  - 32 non-real-time signals (0 to 31)
  - 31 real-time signals (32 to _NSIG [link])

- **Signals we might know**
  - SIGINT : To terminate (CTRL+C)
  - SIGKILL : To terminate immediately (kill -9)
  - SIGSEGV: If segmentation fault happens
  - …



#	Signal Name	Default Action	Comment	POSIX
1	SIGHUP	Terminate	Hang up controlling terminal or process	Yes
2	SIGINT	Terminate	Interrupt from keyboard, Control-C	Yes
3	SIGQUIT	Dump	Quit from keyboard, Control-\	Yes
4	SIGILL	Dump	Illegal instruction	Yes
5	SIGTRAP	Dump	Breakpoint for debugging	No
6	SIGABRT	Dump	Abnormal termination	Yes
6	SIGIOT	Dump	Equivalent to SIGABRT	No
7	SIGBUS	Dump	Bus error	No
8	SIGFPE	Dump	Floating-point exception	Yes
9	SIGKILL	Terminate	Forced-process termination	Yes
10	SIGUSR1	Terminate	Available to processes	Yes
11	SIGSEGV	Dump	Invalid memory reference	Yes
12	SIGUSR2	Terminate	Available to processes	Yes
13	SIGPIPE	Terminate	Write to pipe with no readers	Yes
14	SIGALRM	Terminate	Real-timer clock	Yes
15	SIGTERM	Terminate	Process termination	Yes
16	SIGSTKFLT	Terminate	Coprocessor stack error	No
17	SIGCHLD	Ignore	Child process stopped or terminated or got a signal if traced	Yes
18	SIGCONT	Continue	Resume execution, if stopped	Yes
19	SIGSTOP	Stop	Stop process execution, Ctrl-Z	Yes
20	SIGTSTP	Stop	Stop process issued from tty	Yes
21	SIGTTIN	Stop	Background process requires input	Yes
22	SIGTTOU	Stop	Background process requires output	Yes
23	SIGURG	Ignore	Urgent condition on socket	No
24	SIGXCPU	Dump	CPU time limit exceeded	No
25	SIGXFSZ	Dump	File size limit exceeded	No
26	SIGVTALRM	Terminate	Virtual timer clock	No
27	SIGPROF	Terminate	Profile timer clock	No
28	SIGWINCH	Ignore	Window resizing	No
29	SIGIO	Terminate	I/O now possible	No
29	SIGPOLL	Terminate	Equivalent to SIGIO	No
30	SIGPWR	Terminate	Power supply failure	No
31	SIGSYS	Dump	Bad system call	No
31	SIGUNUSED	Dump	Equivalent to SIGSYS	No

Provide abstraction: pipes

- **Are we happy with signals?**
  - Our communication is limited to 31 types
  - We typically want to send more info (*e.g.*, filename to open)

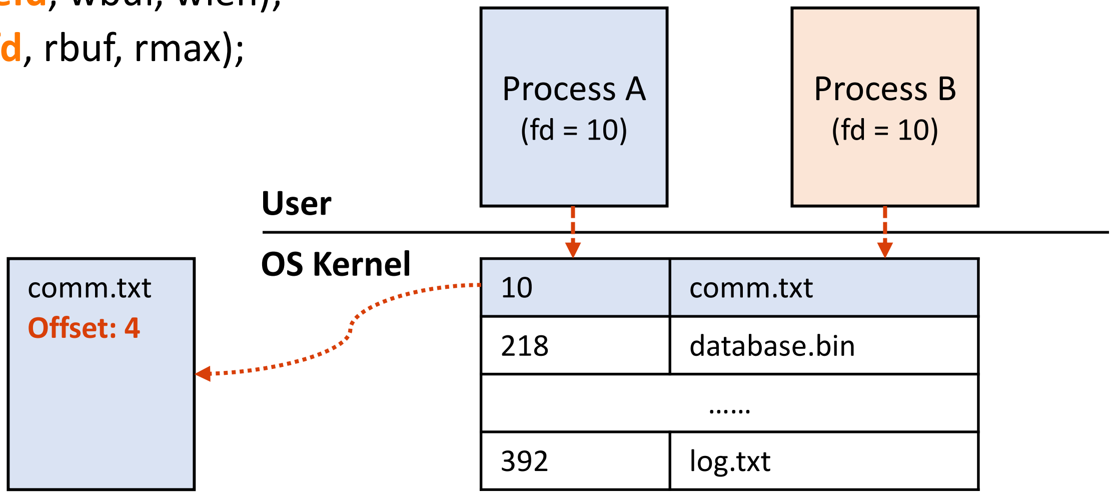
PROVIDE ABSTRACTION: PIPES

- **Are we happy with signals?**
  - Our communication is limited to 31 types
  - We typically want to send more info (*e.g.*, filename to open)

- **PIPE:**
  - **Definition:** a unidirectional data channel, used for inter-process communication
  - **Conceptually:**
    - A file shared between two process (only one can write, and the other can only read)
    - Note: a file descriptor can be shared between two process
      - **To write:** write(**writefd**, wbuf, wlen);
      - **To read :** read(**readfd**, rbuf, rmax);

Provide abstraction: pipes

- **PIPE:**
  - **Definition:** a unidirectional data channel, used for inter-process communication
  - **Conceptually:**
    - A file shared between two process (only one can write, and the other can only read)
    - Note: a file descriptor can be shared (**aliased**) between two process
      - **To write:** write(**writefd**, wbuf, wlen);
      - **To read :** read(**readfd**, rbuf, rmax);
- **Problem?**
  - Too many storage access?



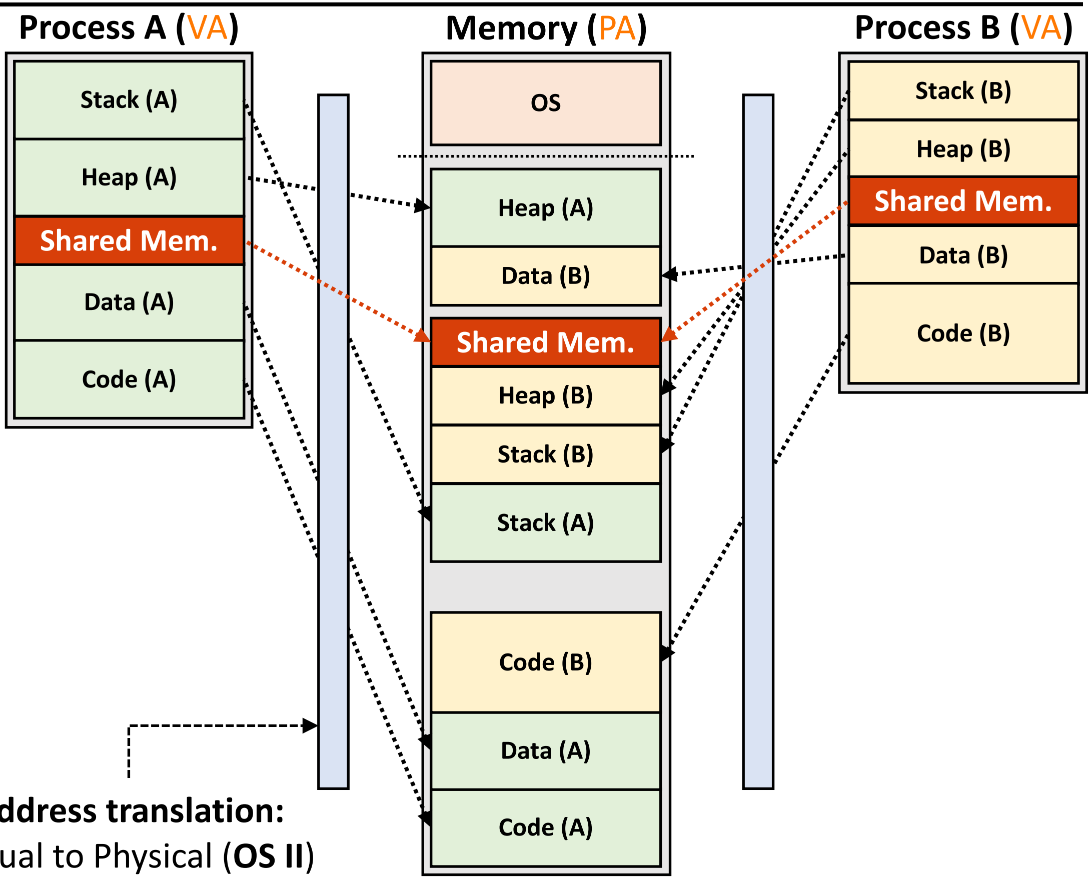
10	comm.txt
218	database.bin
……	
392	log.txt

Provide abstraction: pipes

- **Solution: memory!**
  - Disk access: 10^{-3}s
  - Mem. access: 10^{-9}s
  - Mem is $\mathbf{\sim 10^6 x}$ faster

Provide abstraction: pipes

- **Solution: memory!**
  - Disk access: 10^{-3}s
  - Mem. access: 10^{-9}s
  - Mem is **$\sim 10^6$x** faster

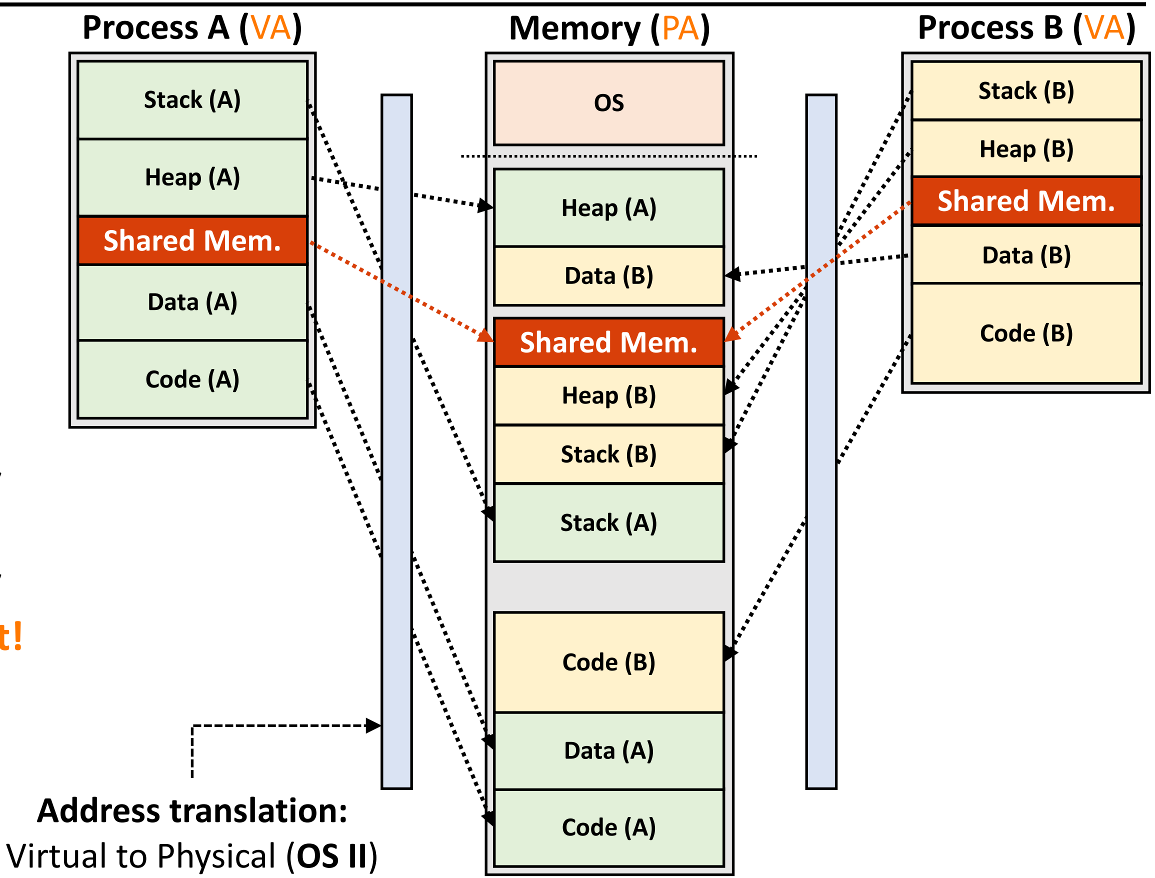


Address translation:
Virtual to Physical (**OS II**)

Provide abstraction: pipes

- **Solution: memory!**
  - Disk access: 10^{-3}s
  - Mem. access: 10^{-9}s
  - Mem is $\sim 10^{6}$x faster

- **Require OS support**
  - We should not allocate shared memory *arbitrarily*
  - We should not control the shared memory *arbitrarily*
  - **Require OS kernel support!**



Topics for today

- Part III: IPC, RPC, and Networking
  - Motivation
    - What is IPC/RPC?
    - Why do we need IPC/RPC?
  - Provide abstractions
    - What is the mechanisms OS support for IPC?
  - Offer standard interface
    - How can we use a signal?
    - How can we use a pipe?
  - Manage resources
    - (Overview) How does OS support these mechanisms?

Offer standard interface: signals

- C APIs

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

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

Member	**Descriptions**
sa_handler	**fn** that will handle a signal(s) (**SIG_DFL**: default action, **SIG_IGN**: ignore this)
sa_sigaction	**fn** that will handle a queued signal(s)
sa_mask	a mask of signals which will be blocked
sa_flags	a set of flags which modify the behavior of signals
sa_restorer	no need to care (not intended for application use)

OFFER STANDARD INTERFACE: SIGNALS

- C APIs

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

Flag	Description
SA_SIGINFO	signal handler takes three arguments, instead of one
… (mostly we don't need it in CS 344)	

- Control signal masks
  - int sigemptyset(sigset_t *set);
  - int sigfillset(sigset_t *set);
  - int sigaddset(sigset_t *set, int signum);
  - int sigdelset(sigset_t *set, int signum);
  - int sigismember(const sigset_t *set, int signum);

Offer standard interface: signals

- **An example code in C**

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

static volatile sig_atomic_t received = 0;

static void hijack_ctrl_c_handler(int sig) {
    received = 1;
}

int main(void) {
    struct sigaction hijack = {0};
    // memset(&hijack, 0, sizeof(struct sigaction));

    hijack.sa_handler = &hijack_ctrl_c_handler;

… (continue to the right)
```

```
… (continue from the left)

    if (sigaction(SIGINT, &hijack, NULL) == -1) {
        perror("Error, failed to change signal action");
        return EXIT_FAILURE;
    }

    while (1) {

        if (received) {
            received = 0;
            printf("Received SIGINT!\n");
        }

        printf("Keep running……\n");
        sleep(2);

    }

    return EXIT_SUCCESS;
}
```

Offer standard interface: signals

- Signalception [Link]
  - A nice example shows how to handle different signal types (Try this out!)

Offer standard interface: pipe

- System call for pipes
  - int pipe(int fds[2]);
    - It returns two file descriptors to “fds”
    - **fds[0]** is the fd for reading from the pipe
    - **fds[1]** is the fd for writing to the pipe
    - Note that the message size limit is 4096 bytes

Offer standard interface: pipe

- System call for pipes
  - int pipe(int fds[2]);
    - It returns two file descriptors to “fds”
    - **fds[0]** is the fd for reading from the pipe
    - **fds[1]** is the fd for writing to the pipe
    - Note that the message size limit is 4096 bytes

- Tips to use “PIPEs” in Terminal
  - If you want to count the total number of files and directories: ls|wc –l
  - If you have many files for a screen: ls -alh | more
  - If you want to catch lines with a specific keywords: cat <filename> | grep <keyword>
  - If you want to remove the files with a prefix: find ./ -name <prefix>* | xargs rm -f {} \;
  - …

Offer standard interface: pipe

- **An example code in C**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define  BUFSIZE    512

int main(void) {
    char *msg = "It’s a message in the pipe.";
    char buf[BUFSIZE];
    int pipe_fd[2];

    if (pipe(pipe_fd) == -1) {
        perror("Error, failed to open a pipe.\n");
        return EXIT_FAILURE;
    }

    ssize_t writelen = write(pipe_fd[1], msg, strlen(msg)+1);
    printf("Send: %s [%ld, %ld]\n", msg, strlen(msg)+1, writelen);

… (continue to the right)
```

```
… (continue from the left)

    ssize_t readlen = read(pipe_fd[0], buf, BUFSIZE);
    printf("Recv: %s [%ld, %ld]\n", buf, strlen(buf)+1, readlen);

    close(pipe_fd[0]);
    close(pipe_fd[1]);

    return 0;
}
```

Offer standard interface: pipe

- **Another example code in C**

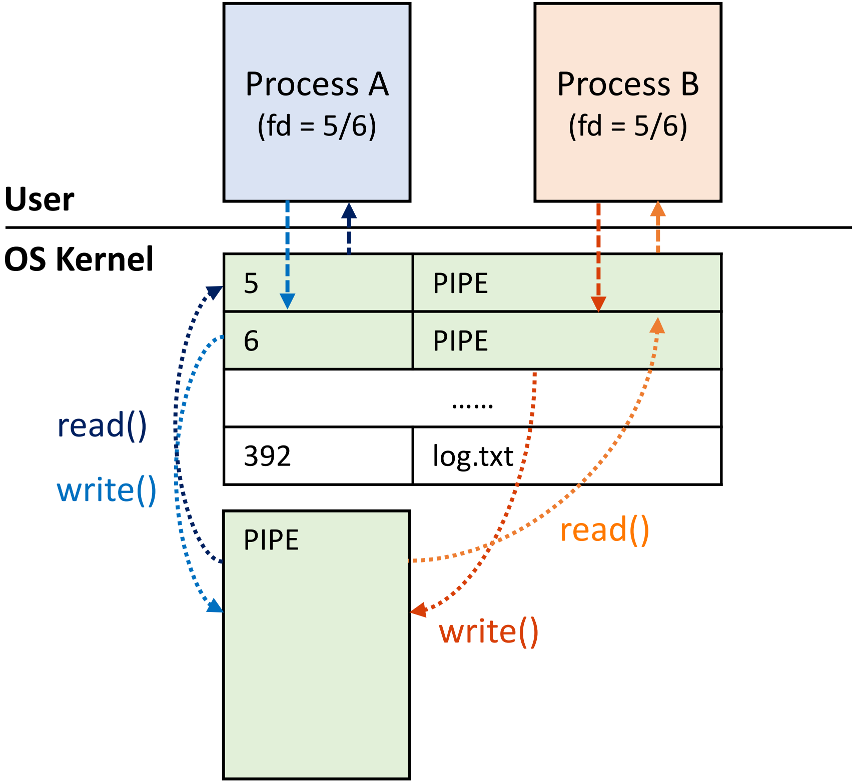
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define  BUFSIZE    512

int main(void) {
    char *msg = "It’s a message in the pipe.\n";
    char buf[BUFSIZE];
    int pipe_fd[2];
    ssize_t readlen, writelen;

    if (pipe(pipe_fd) == -1) {
        perror("Error, failed to open a pipe.\n");
        return EXIT_FAILURE;
    }

    pid_t pid = fork();

… (continue to the right)
```

… (continue from the left)

```
    if (pid < 0) {
        perror("Error, failed to fork().\n");
        return EXIT_FAILURE;
    }

    switch (pid) {
        case 0:
            readlen = read(pipe_fd[0], buf, BUFSIZE);
            printf("Recv: %s [%ld, %ld]\n", buf, strlen(buf)+1, readlen);
            close(pipe_fd[1]);
            break;

        default:
            writelen = write(pipe_fd[1], msg, strlen(msg)+1);
            printf("Send: %s [%ld, %ld]\n", msg, strlen(msg)+1, writelen);
            close(pipe_fd[0]);

    }

    return 0;
}
```

Offer standard interface: PIPE

- PIPE between two processes
  - Process A creates a pipe (fd=5/6)
  - A can read/write with the pipe
  - Process A fork()
  - Process B is created (a child)
  - Process B can read/write from (fd=5/6)



Process A
(fd = 5/6)

Process B
(fd = 5/6)

User

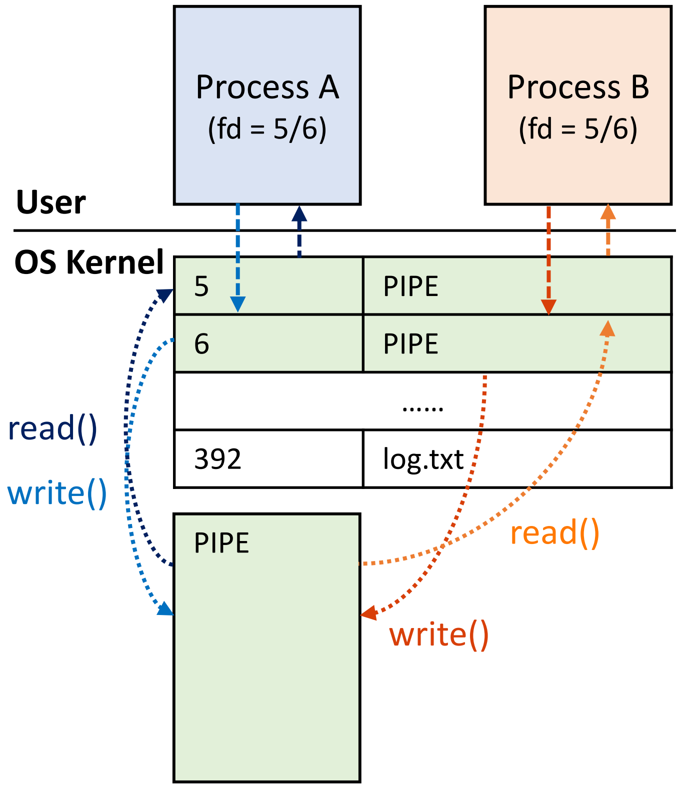
OS Kernel

5	PIPE
6	PIPE
……	
392	log.txt

PIPE

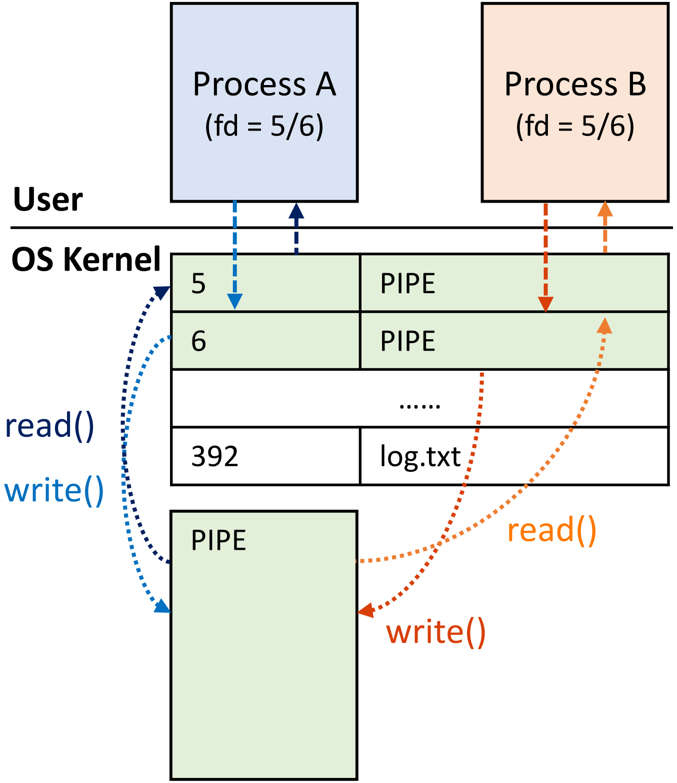
Offer standard interface: PIPE

- PIPE open/close
  - Process A closes “write” file descriptor
    - Process A can still read from the PIPE
    - Process B can still read/write to the PIPE



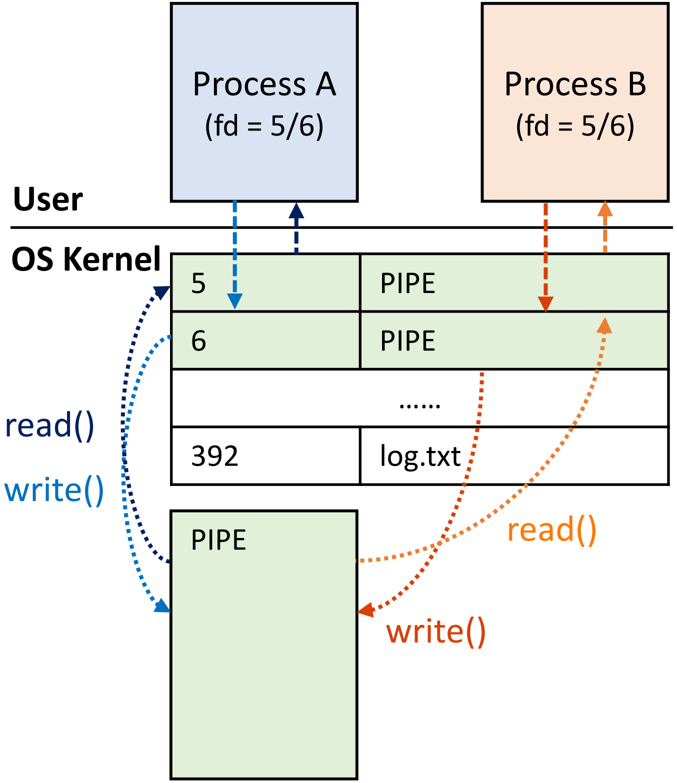
Offer standard interface: PIPE

- PIPE open/close
  - Process A closes “write” file descriptor
    - Process A can still read from the PIPE
    - Process B can still read/write to the PIPE
  - Process A and B close “write” file descriptors
    - Process A and B only read EOF(0) from the PIPE



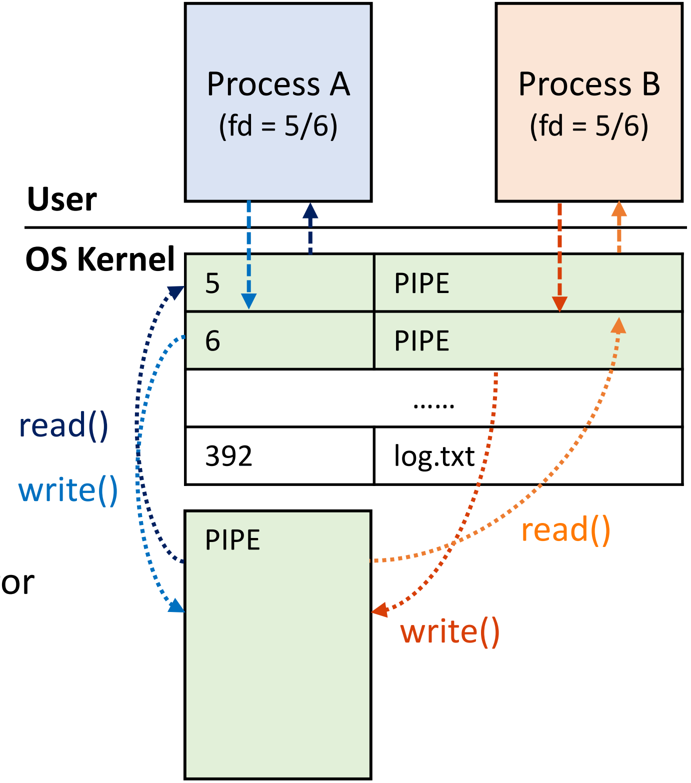
Offer standard interface: PIPE

- PIPE open/close
  - Process A closes "write" file descriptor
    - Process A can still read from the PIPE
    - Process B can still read/write to the PIPE
  - Process A and B close "write" file descriptors
    - Process A and B only read EOF, *i.e.*, 0, from the PIPE
  - Process A closes "read" descriptors
    - Process A and B can write to the PIPE



Offer standard interface: PIPE

- PIPE open/close
  - Process A closes “write” file descriptor
    - Process A can still read from the PIPE
    - Process B can still read/write to the PIPE
  - Process A and B close “write” file descriptors
    - Process A and B only read EOF, *i.e.*, 0, from the PIPE
  - Process A closes “read” descriptors
    - Process A and B can write to the PIPE
  - Process A and B close “read” file descriptors
    - Process A or B’s “write” will fail and return EPIPE error



Topics for today

- Part III: IPC, RPC, and Networking
  - Motivation
    - What is IPC/RPC?
    - Why do we need IPC/RPC?
  - Provide abstractions
    - What is the mechanisms OS support for IPC?
  - Offer standard interface
    - How can we use a signal?
    - How can we use a pipe?
  - Manage resources
    - (Overview) How does OS support these mechanisms?

Manage resources: signal internals

- Signal from Process A -> Process B
  - **OS kernel**
    - Checks if Process B has pending signals
    - Pauses the execution of Process B
    - Invokes do_signal()
    - do_signal() call invokes handle_signal()
  - **Process B**
    - Run code in signal_handler
    - Return back to kernel: sigreturn()
  - **OS Kernel**
    - Resume Process B

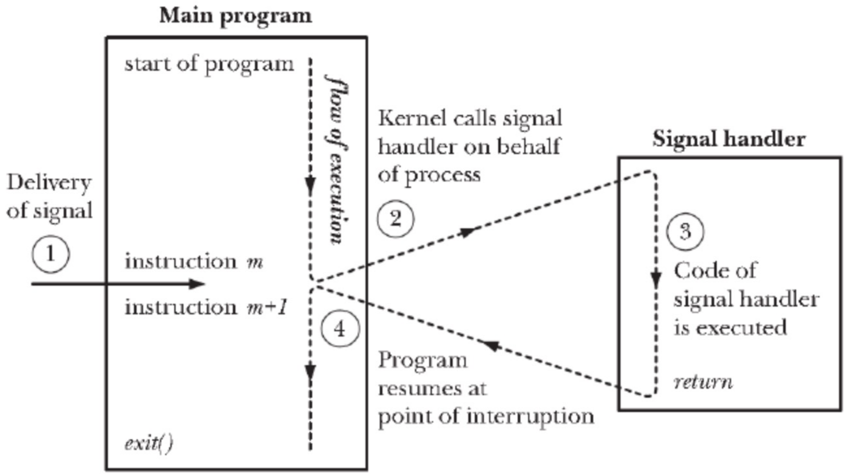
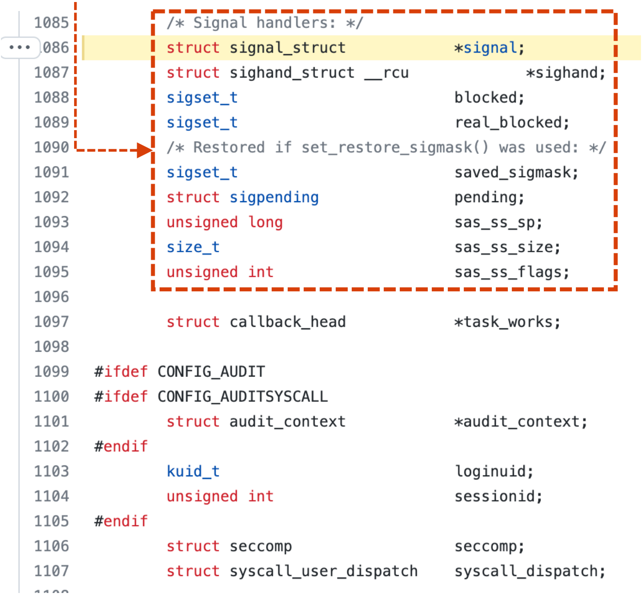


Figure 20-1: Signal delivery and handler execution

Manage resources: signal managed by OS

- Mechanism (OS-level)
  - Process A sends a signal to Process B
  - OS kernel updates B's process context **(Send)**
  - OS kernel asks B to react to the signal **(Receive)**
    - Process B will execute a signal handler
    - Process B declines to receive the signal
  - Multiple processes send signals to B **(Pending)**
    - Up to 1 pending signal per type for each process
    - More signals of the same type will be discarded



```
	/* Signal handlers: */
	struct signal_struct		*signal;
	struct sighand_struct __rcu		*sighand;
	sigset_t			blocked;
	sigset_t			real_blocked;
	/* Restored if set_restore_sigmask() was used: */
	sigset_t			saved_sigmask;
	struct sigpending		pending;
	unsigned long			sas_ss_sp;
	size_t				sas_ss_size;
	unsigned int			sas_ss_flags;

	struct callback_head		*task_works;

#ifdef CONFIG_AUDIT
#ifdef CONFIG_AUDITSYSCALL
	struct audit_context		*audit_context;
#endif
	kuid_t				loginuid;
	unsigned int			sessionid;
#endif
	struct seccomp			seccomp;
	struct syscall_user_dispatch	syscall_dispatch;
```

Manage resources: pipe

- Data structure
  - **Queue** in memory
  - **(Rule)** If Proc A writes data, the data will be in the kernel queue until Proc B reads it

- OS kernel's queue control:
  - Queue can be **full/empty**
    - If the queue is full, OS kernel asks Proc A (write) to wait
    - If the queue is empty, OS kernel asks Proc B (read) to wait

Topics for today

- Part III: IPC, RPC, and Networking
  - Motivation
    - What is IPC/RPC?
    - Why do we need IPC/RPC?
  - Provide abstractions
    - What is the mechanisms OS support for IPC?
  - Offer standard interface
    - How can we use a signal?
    - How can we use a pipe?
  - Manage resources
    - (Overview) How does OS support these mechanisms?

Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab