

Richard Litt

Scan QR code to add me as a friend.

Week 12 Lecture 2

XMUT-NWEN 241 - 2026 T2

Systems Programming

Richard Littauer

School of Engineering and Computer Science

Victoria University of Wellington

richard.littauer@vuw.ac.nz

Admin

- Exercise 3
  - Due date is 6 December.

- For the exercises, since we use an automated marking process, please make sure to:
  - Submit C files. In particular no Microsoft Word documents and executable files.
  - No special characters. It breaks the script and makes me sad!

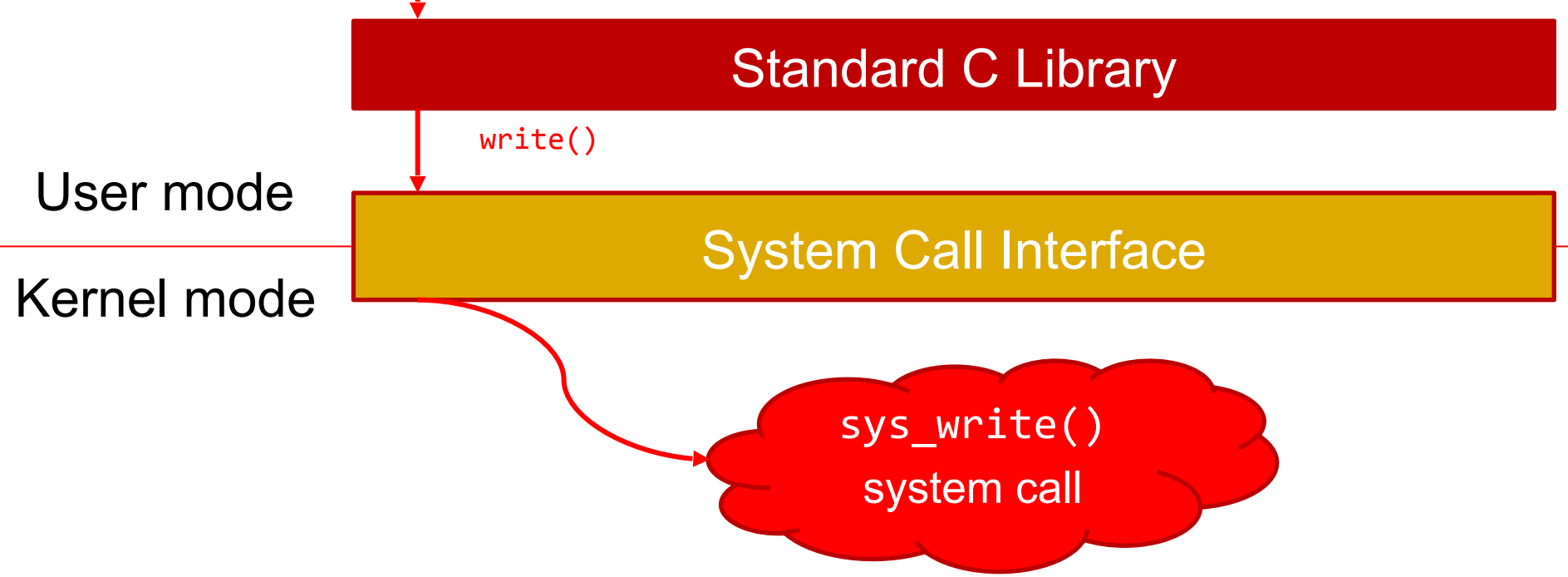
Content

- System calls (in a bit more detail)

- Categories of System Calls

Recap: System call invocation –Example

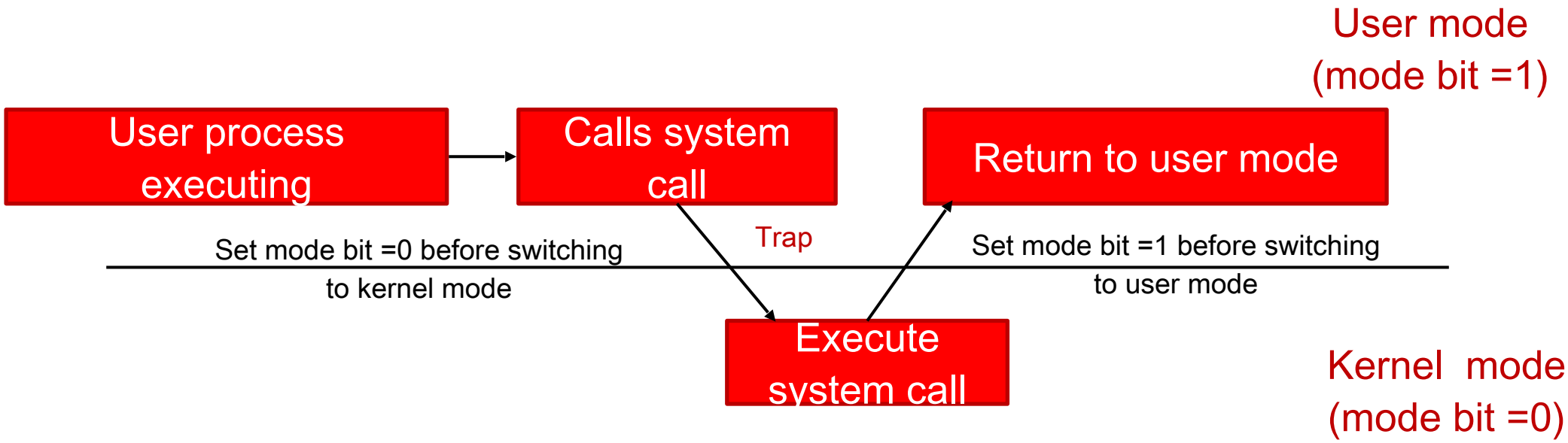
```
#include <stdio.h>
int main(void)
{
    printf("Hello, world\n");
    return 0;
}
```



The Complete picture

- A system call is a call to a function that is a part of the kernel to request service from the operating system.

- When a program needs to access system resources, it makes a system call and a mode switch between the user program and the kernel is performed.



How to know which library calls are invoked?

```c
#include<stdio.h>

int main()
{
   printf(”Hello World”) ;
   return 0;
}
```

hello.c

Compile

```
$ gcc -o hello hello.c
```

Run

```
$ ./hello
```

Output

```
Hello World
```

ltrace

```
ltrace ./hello
```

ltrace output

```
printf("Hello World")
Hello World+++ exited (status 0) +++
```

How to know which system calls are invoked?

strace output

```
execve("./hello", ["./hello"], 0x7fffc8e68920 /* 21 vars */) = 0
brk(NULL)                               = 0x7ffff38a6000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=30022, ...}) = 0
mmap(NULL, 30022, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff4e26e1000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2030544, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff4e26d0000
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ff4e2000000
mprotect(0x7ff4e21e7000, 2097152, PROT_NONE) = 0
mmap(0x7ff4e23e7000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7ff4e23e7000
mmap(0x7ff4e23ed000, 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7ff4e23ed000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7ff4e26d14c0) = 0
mprotect(0x7ff4e23e7000, 16384, PROT_READ) = 0
mprotect(0x7ff4e2a00000, 4096, PROT_READ) = 0
mprotect(0x7ff4e2627000, 4096, PROT_READ) = 0
munmap(0x7ff4e26e1000, 30022)           = 0
fstat(1, {st_mode=S_IFCHR|0660, st_rdev=makedev(4, 1), ...}) = 0
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
brk(NULL)                               = 0x7ffff38a6000
brk(0x7ffff38c7000)                     = 0x7ffff38c7000
write(1, "Hello World", 11Hello World)            = 11
exit_group(0)                           = ?
+++ exited with 0 +++
```

Try it: `strace` on Linux

```c
#include<stdio.h>

int main()
{
    printf("Hello World");
    return 0;
}
```

Why it matters

- `strace ./hello` lists every system call.
- `strace -c ./hello` counts them.
- Our own `write` is near the end.

Terminal

```
$ gcc -o hello hello.c
$ strace ./hello
$ strace -c ./hello
```

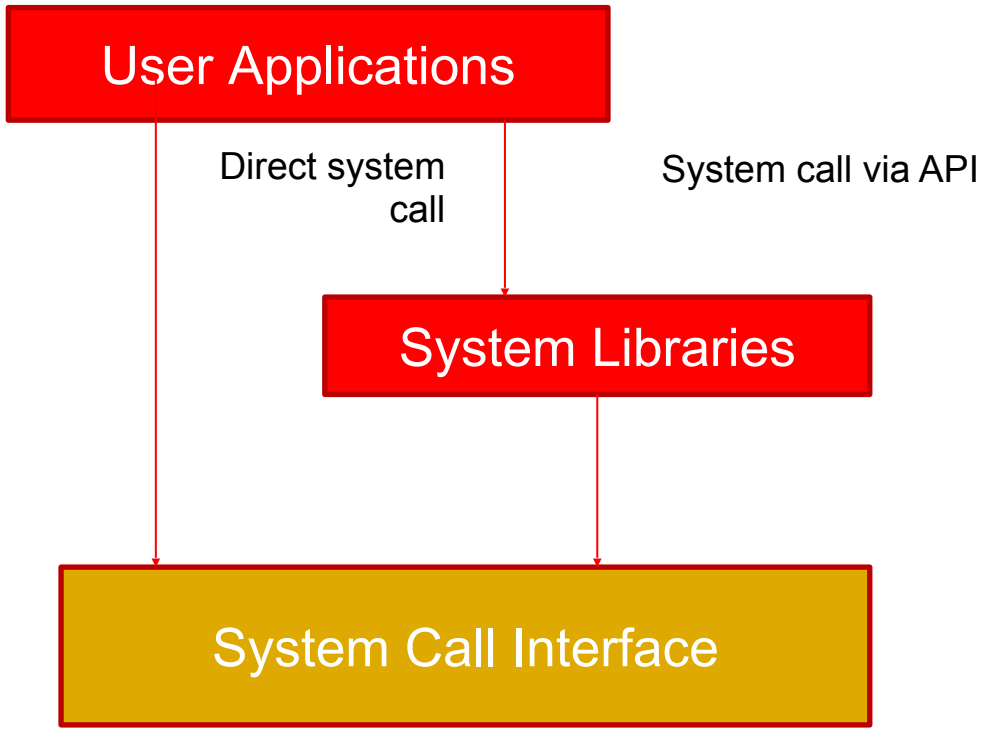
Invoking System calls

There are two different methods by which a program can invoke system calls:

- Directly: by making a system call to a function (i.e., entry point) built directly into the kernel, or
- Indirectly: by calling a higher-level library routine (provided by Linux system library and language library) that invokes the system call.
- The system calls and system libraries together constitute the system call application programming interface (API).

Three most common APIs:

- Win32 API for Windows
- POSIX API for POSIX-based systems (including UNIX, Linux)
- Java API for the Java virtual machine (JVM)



System call implementation

- Typically, a number is associated with each system call
  - System call interface maintains a table indexed according to these numbers

- The system call interface invokes intended an system call in the kernel and returns the status of the system call and any return values

- Caller doesn't need to know about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API

Linux system call table

- First few lines of the table

- For more information: https://github.com/torvalds/linux/blob/v3.13/arch/x86/syscalls/syscall_64.tbl

  Same as https://elixir.bootlin.com/linux/v3.13/source/arch/x86/syscalls/syscall_64.tbl

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
6       common  lstat                   sys_newlstat
7       common  poll                    sys_poll
```

Git: https://mirrors.tuna.tsinghua.edu.cn/git/linux.git/

Directly Invoking System calls

- To make a **direct system call we need low-level programming**, generally in assembly.
- User needs to know the **target architecture;** they cannot create CPU independent code.

Tedious and machine dependent

```
 .global _start

  .text
_start:
  # write(1, message, 13)
  mov     $1, %rax                # system call 1 is to write
  mov     $1, %rdi                # file descriptor 1 is stdout
  mov     $message, %rsi          # address of string to output
  mov     $13, %rdx               # number of bytes
  syscall                         # invoke operating system to do the write
  mov     $60, %rax               # system call 60 is exit
  mov     $0, %rdi                # exit status 0
  syscall                         # invoke operating system to exit
  .data
message:
  .ascii "Hello, world\n"
```

Invoking System calls through library routines



Demo: write

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

int main()
{
    ssize_t r = write(1, "Hello World",
                      strlen("Hello World"));
    printf("\nwrite returned %zd\n", r);
    return 0;
}
```

Demo: write

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

int main()
{
    ssize_t r = write(1, "Hello World",
                      strlen("Hello World"));
    printf("\nwrite returned %zd\n", r);
    return 0;
}
```

Why it matters

- `write` is a library routine that makes the system call.
- It returns how many bytes it wrote.
- `1` is standard output.

Output

```
$ ./prog
Hello World
write returned 11
```

Demo: a system call fails

```
#include <stdio.h>
#include <fcntl.h>

int main(void)
{
    int fd = open("missing.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        printf("open returned %d\n", fd);
        return 1;
    }
    printf("opened, fd is %d\n", fd);
    return 0;
}
```

Demo: a system call fails

```c
#include <stdio.h>
#include <fcntl.h>

int main(void)
{
    int fd = open("missing.txt", O_RDONLY);
    if (fd == -1) {
        perror("open");
        printf("open returned %d\n", fd);
        return 1;
    }
    printf("opened, fd is %d\n", fd);
    return 0;
}
```

Why it matters

- A failed system call returns `-1`.
- `errno` says why. `perror` prints it as text.
- Always check for `-1`.

Output

```
$ ./prog
open: No such file or directory
open returned -1
```

Library function or system call?

Library function	System call underneath
printf, puts	write
fgets, scanf	read
fopen, fclose	open, close
malloc, free	brk or mmap (Linux)
exit	exit_group (Linux)
write, read	the system call itself

- `man 2 write`: section 2 is system calls.
- `man 3 printf`: section 3 is library functions.
- A library function can make several system calls, or none: `strlen` makes none.

Terminal

```
$ man 2 write
$ man 3 printf
```

Demo: the cost of a system call

```
// A: one system call per character
for (int i = 0; i < 1000000; i++)
    write(1, "x", 1);

// B: stdio collects them first
for (int i = 0; i < 1000000; i++)
    putchar('x');
```

Demo: the cost of a system call

```
// A: one system call per character
for (int i = 0; i < 1000000; i++)
    write(1, "x", 1);

// B: stdio collects them first
for (int i = 0; i < 1000000; i++)
    putchar('x');
```

Why it matters

- A is a million trips into the kernel.
- B collects the characters in a buffer first.
- That's why `printf` buffers.

A: write

```
$ time ./write_loop > /dev/null
real  0m0.56s
```

B: putchar

```
$ time ./putchar_loop > /dev/null
real  0m0.01s
```

Categories of System calls

Categories and examples of system calls

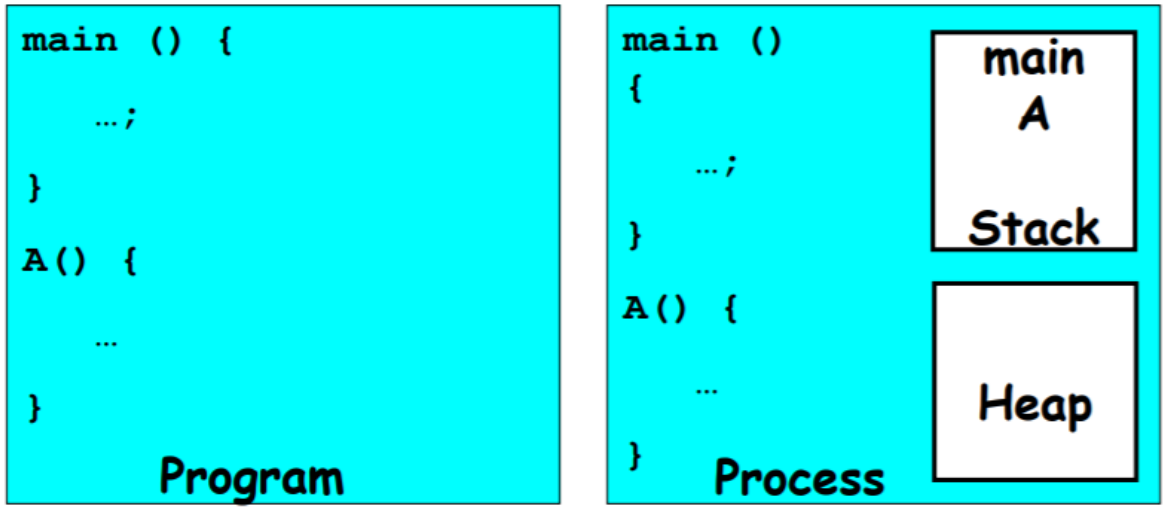
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

- Unix and Linux both conform to POSIX standard (GNU C Library - glibc)
- POSIX: Portable Operating System Interface

Categories of System Calls

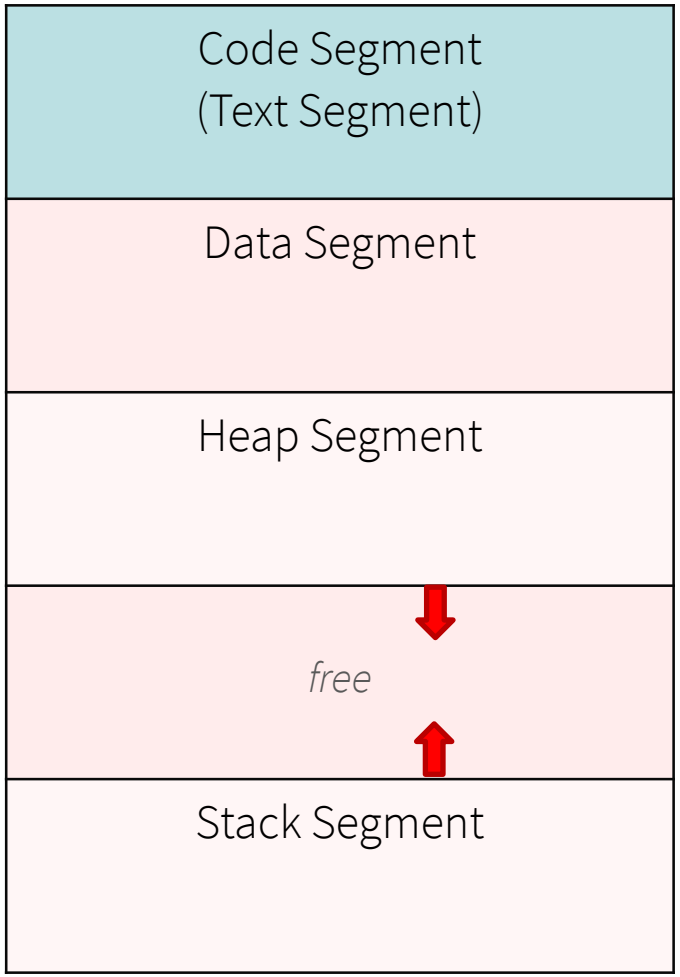
- File manipulation – (create, delete, open, close)
- Process Control – (create, terminate)
- Device Management – (request, release)
- Information Maintenance – (time, date, get / set system date)
- Communications – (create , delete connection, receive, send message)
- Protection – (chmod, chown, umask)

Recap: Process Vs Program



- Program is static, with the potential for execution
- Process is a program in execution and have a state
- One program can be executed several times and thus has several processes

Process in memory



- **Text / Code Segment**
  - Contains the machine code and read-only data, such as string literals
- **Segments for Data**

  spread over:
  - **Data Segment** – Fixed space for global and static variables (zero-filled ones are called bss)
  - **Stack Segment** – For temporary data, *e.g.*, local variables in a function; expands / shrinks as program runs
  - **Heap Segment** – For dynamically allocated memory; expands / shrinks as program runs

Demo: where things live

```
int global = 1;

int main(void)
{
    int local = 2;
    int *heap = malloc(sizeof(int));

    printf("global: %p\n", (void *)&global);
    printf("heap:   %p\n", (void *)heap);
    printf("local:  %p\n", (void *)&local);
    …
}
```

Demo: where things live

```
int global = 1;

int main(void)
{
    int local = 2;
    int *heap = malloc(sizeof(int));

    printf("global: %p\n", (void *)&global);
    printf("heap:   %p\n", (void *)heap);
    printf("local:  %p\n", (void *)&local);
    …
}
```

Why it matters

- `global` is in the data segment: lowest.
- `malloc` memory is on the heap: in between.
- `local` is on the stack: highest.

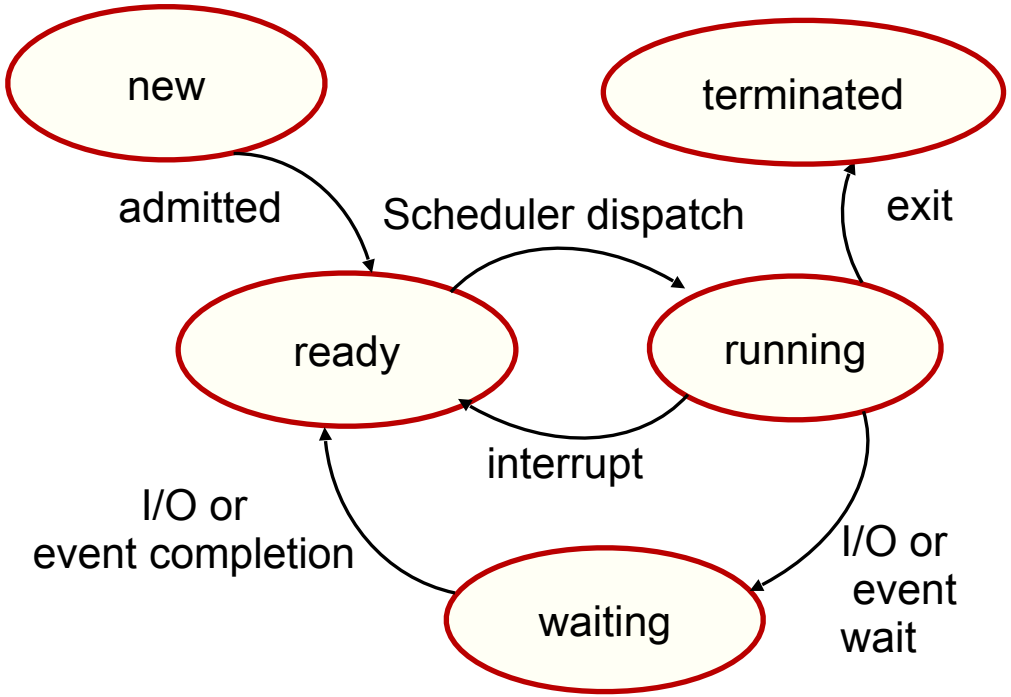
Output

```
$ ./prog
global: 0x100f60000
heap:   0x10173d550
local:  0x16eea5a78
```

Recap: Process lifecycle

As a process executes, it changes state

- new: The process is being created
- ready: The process is waiting to be assigned to a processor
- running: Instructions are being executed
- waiting: The process is waiting for some event to occur
- terminated: The process has finished execution

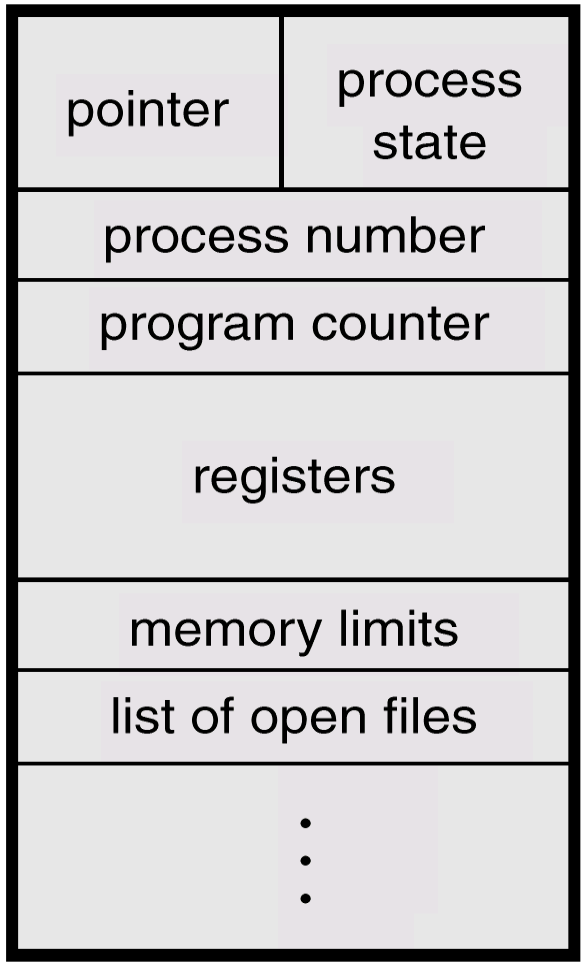


Other process states

- **Stopped**: the process was paused by a signal such as `SIGSTOP` or Ctrl-Z and won't run until it's continued. `ps` shows this as `T`.
- **Zombie (terminated, not reaped)**: the process has exited, but its parent hasn't collected the exit status yet. It holds a process table entry and nothing else. `ps` shows this as `Z`. You can't kill a zombie. The parent has to reap it, or the parent has to exit.
- **Uninterruptible sleep**: the process is waiting on I/O, usually disk or a network filesystem, and ignores signals until the wait finishes. `ps` shows this as `U` on macOS and `D` on Linux. It's why a stuck process sometimes can't be killed.
- **Idle**: the process is sleeping for a long time, which macOS reports as `I`. An idle shell at a prompt is usually `S` or `I`.
- **Swapped or suspended**: the memory pages have been moved to disk. This is a memory-management state, not an execution state, so it can overlap with the others.

Process control block

- Information associated with each process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information
- A process is named using its process ID (PID) or process #
- Data is stored in a process control block (PCB)



Process control block: the details

- **Process state**: whether the process is new, ready to run, running, waiting for an event such as I/O, or terminated.
- **Program counter**: the address of the next instruction the process will execute. It's saved here when the process is descheduled and restored when it resumes.
- **CPU registers**: the saved values of the general-purpose registers, the stack pointer, and status flags such as rax from earlier. They're stored on a context switch so the process resumes exactly where it stopped.
- **CPU scheduling information**: the data the scheduler uses to decide when the process runs, such as priority, scheduling queue pointers, and time slice details.
- **Memory-management information**: the mapping the process uses to access memory, such as page tables or segment tables, base and limit registers, and the process's address-space details.
- **Accounting information**: resource-usage records such as CPU time used, time limits, process and thread IDs, and parent and child relationships.
- **I/O status information**: the I/O devices allocated to the process and the open files it holds, including its file descriptor table, such as fd 4 from before.

Process representation in Linux

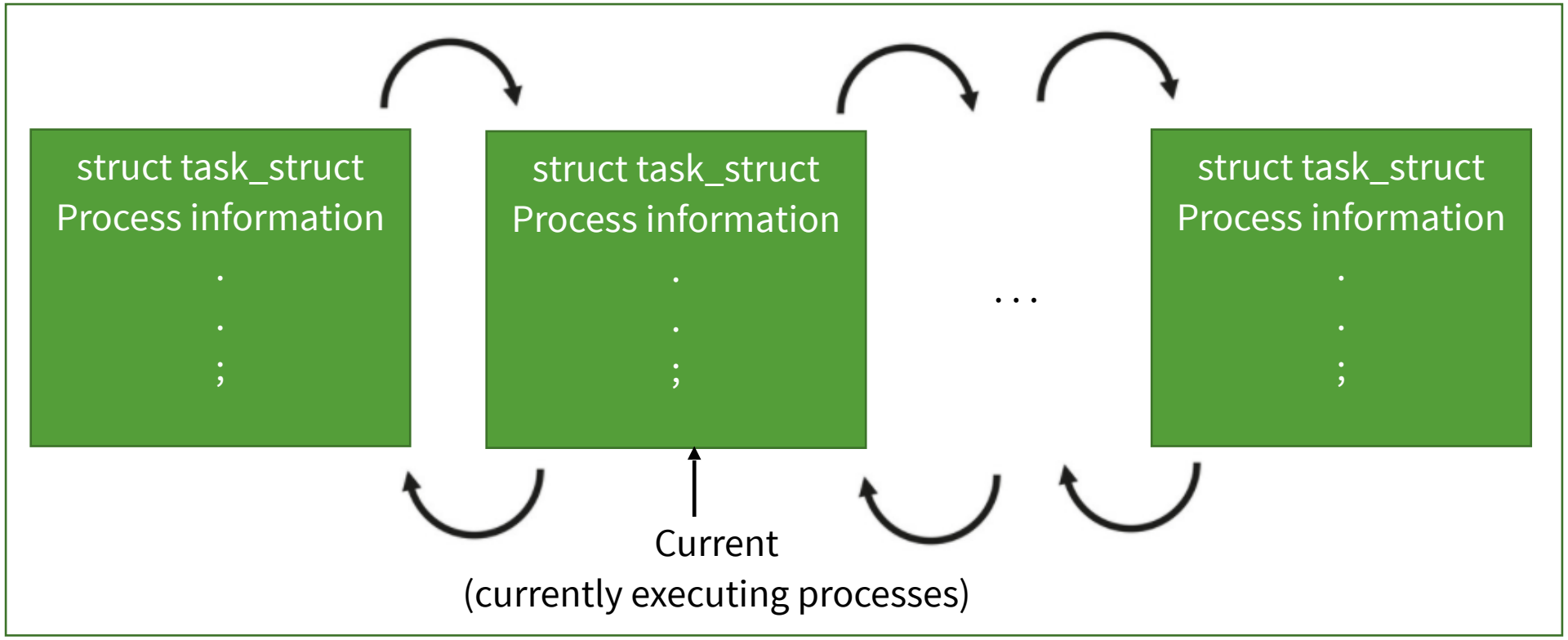
- Represented by structure task_struct
  - See https://github.com/torvalds/linux/blob/master/include/linux/sched.h for more information

- Some of the structure members

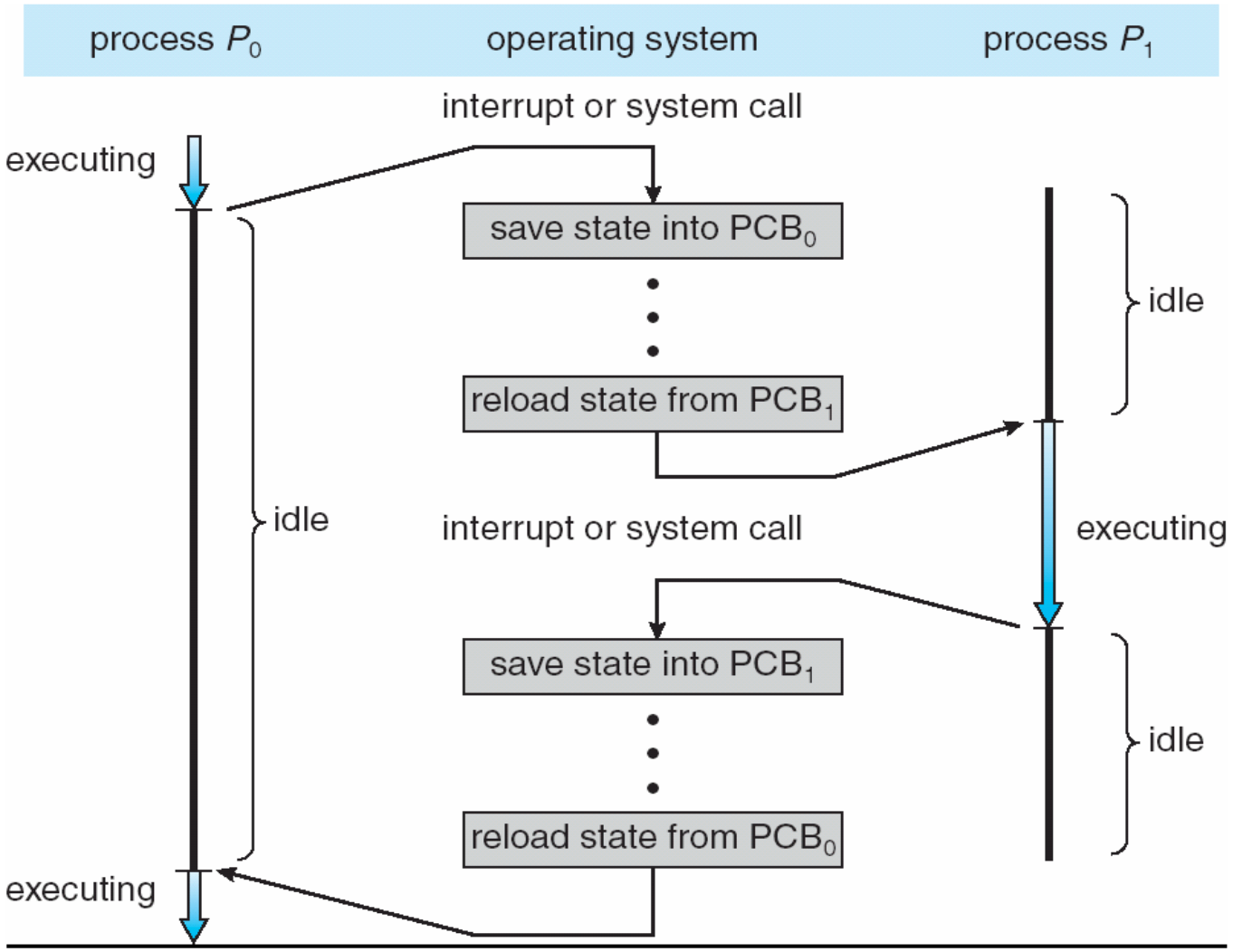
```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

Process representation in Linux

- Represented by structure task_struct
  - See https://github.com/torvalds/linux/blob/master/include/linux/sched.h for more information



Process switching



Mode switch and context switch

	Mode switch	Context switch
What happens	User mode to kernel mode, and back	One process stops and another runs
Which process runs	The same process goes on	A different process
Caused by	A system call or an interrupt	A timer, or a process that has to wait
Cost	Fairly cheap	Costs more: save one PCB, load another

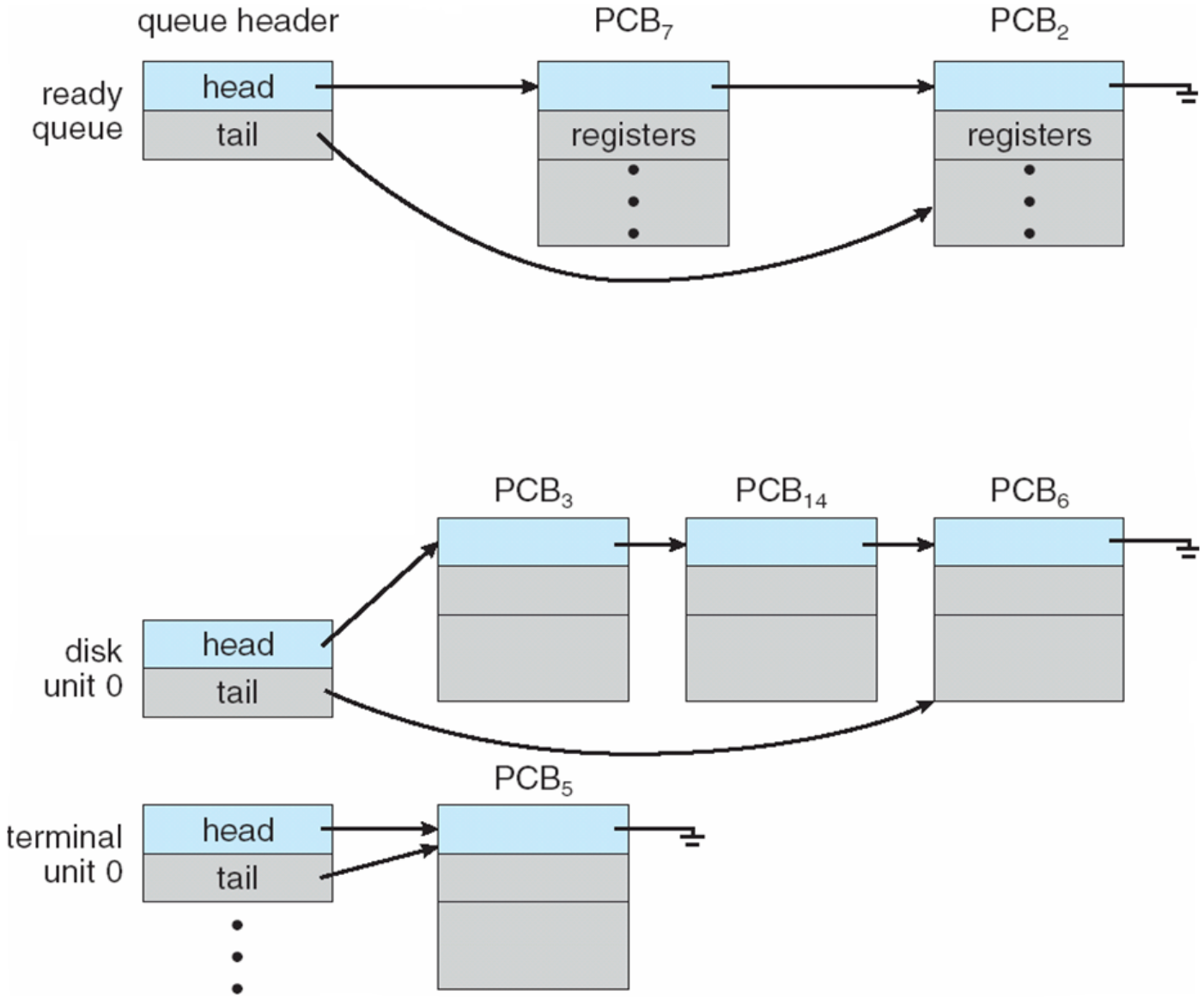
- A system call is a mode switch.
- A context switch changes which process runs.
- A system call can lead to a context switch.

Process scheduling

- Process scheduler selects among ready processes for next execution on CPU

- Maintains scheduling queues of processes
  - Job queue – set of all processes in the system
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute
  - Device queues – set of processes waiting for an I/O device
  - Processes migrate among the various queues

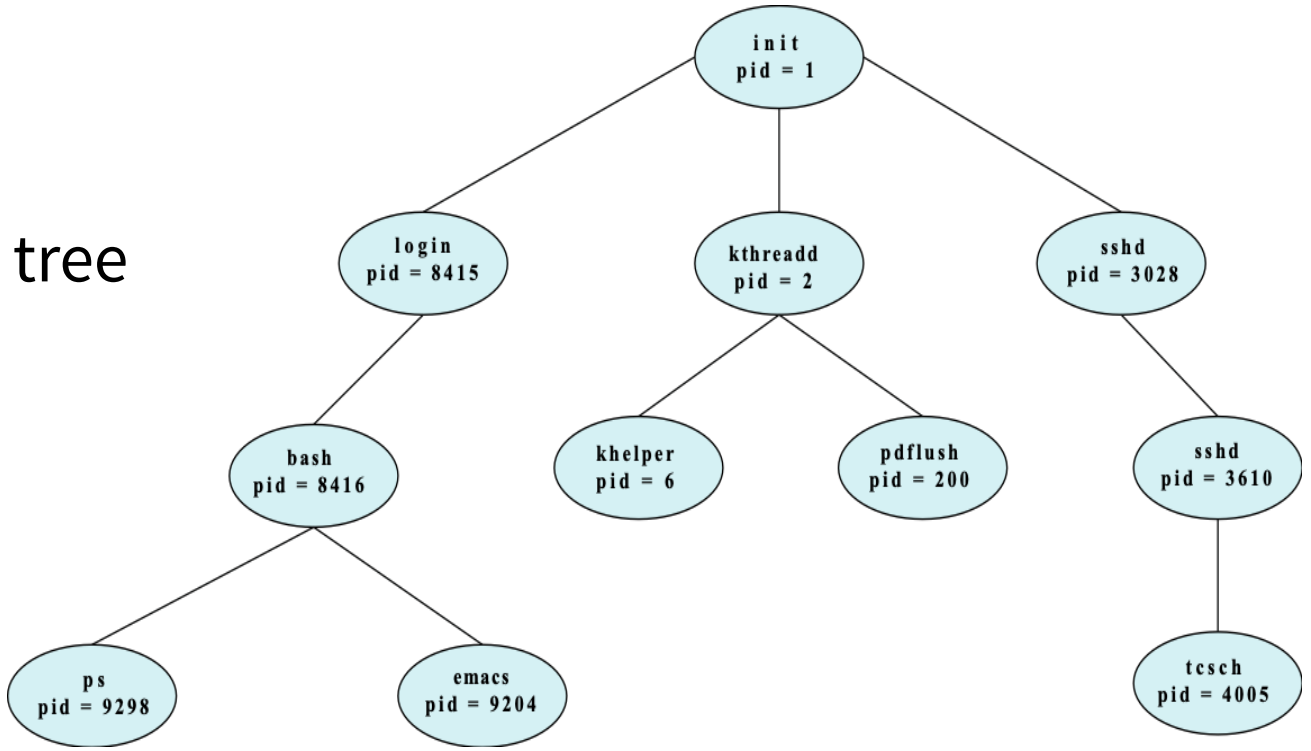
Ready queue and various I/O device queues



Process Initialization on Linux

- The **init** process (Init is the parent of all processes, executed by the kernel during the booting of a system).
- A process is created by another process, which, in turn create other processes → process tree

Linux
process tree

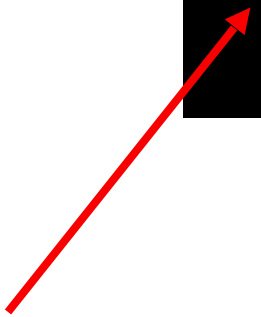


Every process has a process ID (PID)

Linux ps command

- Used to obtain information about processes that are running in the current shell

```
$ ps
  PID  TTY          TIME   CMD
 31843 pts/35    00:00:00  bash
 31850 pts/35    00:00:00  ps
```



Process ID
Every process is assigned a PID by the kernel

Linux ps command

```
$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
sahnijy    31843 31835  0 12:37 pts/35   00:00:00 -bash
sahnijy    32100 31843  0 12:43 pts/35   00:00:00 ps -f
```


Parent Process ID

PID of the process that started the process

Demo: PID and parent PID

```c
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    printf("my pid:     %d\n", getpid());
    printf("parent pid: %d\n", getppid());
    return 0;
}
```

Terminal

```
$ echo $$
$ ./prog
$ ./prog
```

Demo: PID and parent PID

```c
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    printf("my pid:     %d\n", getpid());
    printf("parent pid: %d\n", getppid());
    return 0;
}
```

Why it matters

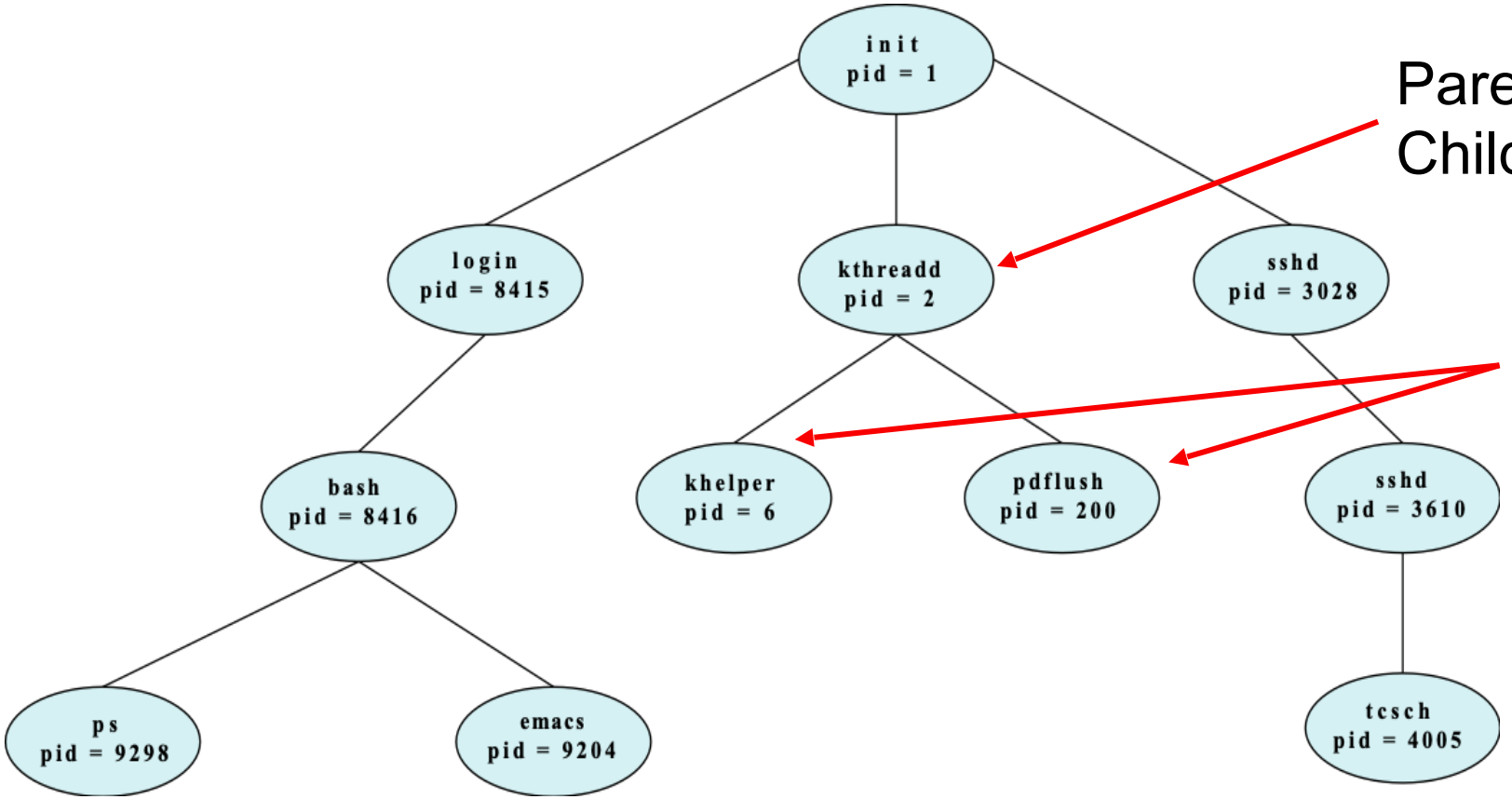
- `getpid` is the process's own number.
- `getppid` is its parent's: the shell.
- `echo $$` prints the shell's PID.

Output

```
$ echo $$
59461
$ ./prog
my pid:     59462
parent pid: 59461
$ ./prog
my pid:     59464
parent pid: 59461
```

Parent and child

When Linux starts it runs a single program, **init** with process id **1**



Parent of processes 6 and 200,
Child of process 1

Children of process 2

Next lecture

- System calls for Process Management

Review question 1

```
printf("%d\n", getppid());
```

- You run this from a shell, and it prints 4021. What is the shell's PID?

Review question 2

```
ssize_t r = write(1, "Hello World\n", 12);
printf("%zd\n", r);
```

- What does this print?

Review question 3

```
write(1, "Hello World\n", 5);
```

- What does this print, and where does the prompt appear?

Review question 4

```
int fd = open("nofile.txt", O_RDONLY);
ssize_t r = read(fd, buf, 10);
printf("%zd\n", r);
```

- nofile.txt doesn't exist. What does this print?

More questions: system calls

- What is a trap?
- Why can't a user program touch the disk directly?
- What are the two ways a program can make a system call?

More questions: processes

- What's the difference between a program and a process?
- Name three things stored in a process control block.
- A process has just been created and admitted. Which state does it go to?

More questions: scheduling and the tree

- What does the scheduler do?
- What does PPID mean in the `ps -f` output?
- Which process does every other process come from, and what's its PID?

Review question

```
int fd[2];
char buf[20];
pipe(fd);
if (fork() == 0) {
    read(fd[0], buf, 20);
    printf("got it\n");
    exit(0);
}
sleep(5);
write(fd[1], "hi", 2);
```

- What is the child doing during the 5 seconds? Which design option is this?

More questions: system calls (1)

- Why can't your program just send bytes to the screen itself?
- Is `printf` a system call?
- What's the difference between user mode and kernel mode?

More questions: system calls (2)

- Name a physical resource and a virtual resource.
- `write(1, "Hi\n", 3)`. What does the 1 mean, and what does the 3 mean?

More questions: processes (1)

- Program or process: which one is just a file on disk?
- You run the same program three times at once. How many processes?
- A process is waiting for you to press a key. Which state is it in?

More questions: processes (2)

- What's the difference between ready and running?
- `fork` returned 0. Are you the parent or the child?
- Three forks in a row, then `printf("hi\n")`. How many times does `hi` print?

More questions: processes (3)

- `exec` worked. What happens to the code after it in your program?
- `exec` fails, for example because the program name is misspelled. What happens?
- What does `wait` do?
- Which call ends a process?

More questions: IPC (1)

- Independent or cooperating: a calculator app?
- Independent or cooperating: a chat app?
- Name two reasons for processes to cooperate.

More questions: IPC (2)

- The two models of IPC?
- In a pipe, which end do you write to, `fd[0]` or `fd[1]`?
- A blocking send with a zero-capacity link. When does the sender carry on?

More questions: IPC (3)

- A bounded buffer is full. What happens to a blocking send?
- Which kind of buffer means the sender never waits?
- After `fork`, the child sets a global variable. Does the parent see the change?

More questions: client-server

- Who needs to know whose address first: the client or the server?
- In the two-terminal demo, which terminal was the server?
- In the two-terminal demo, what stood in for the server's address?

Review question

```
int a = socket(AF_INET, SOCK_STREAM, 0);
int b = socket(AF_INET, SOCK_STREAM, 0);
printf("%d %d\n", a, b);
```

- What does this print?

Review question

```
printf("%x\n", htons(0x1234));
```

- What does this print on a little-endian machine?

Review question

```
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = 8080;
bind(fd, (struct sockaddr *)&addr,
     sizeof(addr));
listen(fd, SOMAXCONN);
```

- A client connects to port 8080 and gets Connection refused. Why?

Review question

```
ssize_t r;
while ((r = recv(client_fd, buf, 99, 0)) > 0) {
    buf[r] = '\0';
    printf("%s", buf);
}
printf("done\n");
```

- When does this print done?

Review question

```
int client_fd = accept(fd,
        (struct sockaddr *)&client_addr,
        &addrlen);
char msg[] = "hello";
send(fd, msg, strlen(msg), 0);
```

- The client never gets hello. What's wrong?

Review question

```
char buf[5];
recv(client_fd, buf, 5, 0);
printf("%s\n", buf);
```

- The client sends hello. What's wrong?

More questions: sockets and ports (1)

- What two things make up a socket address?
- What does 127.0.0.1 mean?
- Can two servers on one machine both use port 80?

More questions: sockets and ports (2)

- Which port does a web server usually use?
- A client never calls bind. Where does its port come from?
- Why are well-known port numbers useful?

More questions: TCP and UDP

- Name two things TCP guarantees that UDP doesn't.
- Which socket type is TCP: SOCK_STREAM or SOCK_DGRAM?
- A game sends each player's position 30 times a second. TCP or UDP?

More questions: the server (1)

- Put these in order: accept, bind, listen, socket.
- Which server call waits until a client connects?
- What does accept return?

More questions: the server (2)

- What's the difference between listen and accept?
- What does INADDR_ANY mean?
- Why do we cast &addr to (struct sockaddr *)?

More questions: byte order

- What is network byte order?
- What does htons stand for?
- A port number arrives from the network. Which function makes it print correctly?

More questions: the client

- Which two calls does a TCP client need before it can send?
- connect fails with Connection refused. What's the likely reason?
- What would you change in 52_client.c to reach a server on another machine?

More questions: send, recv and close

- recv returns 0. What happened?
- send(fd, buf, len, 0) does the same as which file I/O call?
- Why do you have to add the '\0' yourself after recv?