Discussion 1

Fundamentals, Processes, Pintos Lists

01/26/24

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	Project 0 Release		Homework 0 Due	Homework 1 Release	Early Drop Deadline	Group Formation Deadline
		Project 0 Due	Project 1 Release			
			Homework 1 Due			

Fundamentals

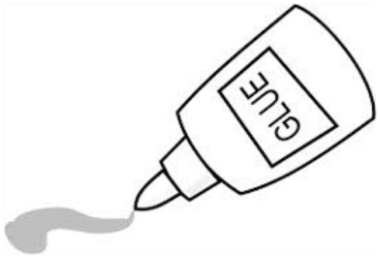
Operating systems

Operating systems (OS) provide hardware abstractions (e.g. file systems, processes) to software applications and manage hardware resources (e.g. memory, CPU).

- Not a well-defined term!
- Special layer of software that provides application software access to hardware resources.

Plays three roles.

- Referee: manage protection, isolation, and sharing of resources.
- Illusionist: provide clean, easy-to-use abstractions of physical resources.
- Glue: provide common services.



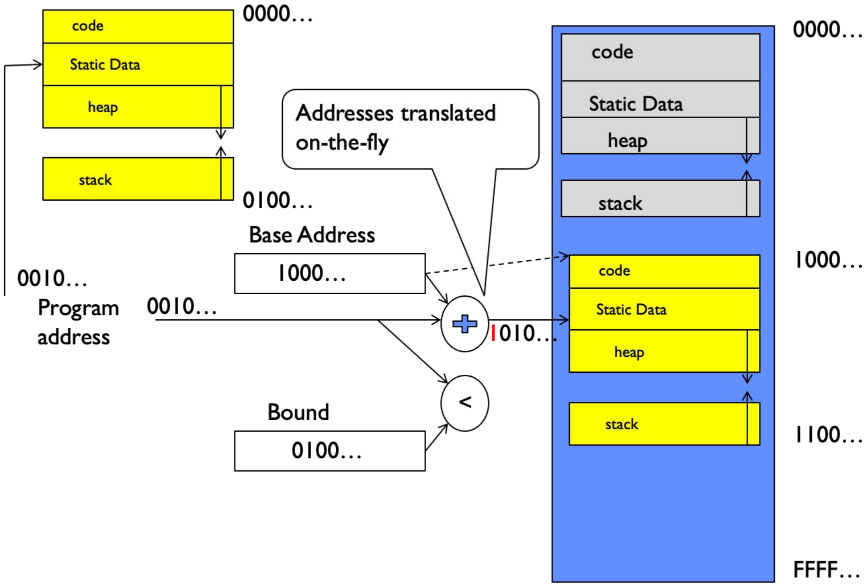
Address Space

Address space is the set of accessible addresses and associated states.

- 32-bit processor has $2^{32} \approx$ 4 billion addresses.
- Entire address space isn't real locations but potential spaces.
- Exception/fault (e.g. segfault) if trying to access restricted memory.

Programs operate with **virtual memory**.

- Instead of accessing physical memory directly, programs request a **virtual address** which is translated into a **physical address**.
- Examples include **base and bound** (shown on right), **segmentation, page tables**



Dual Mode Operation

Hardware provides two modes.

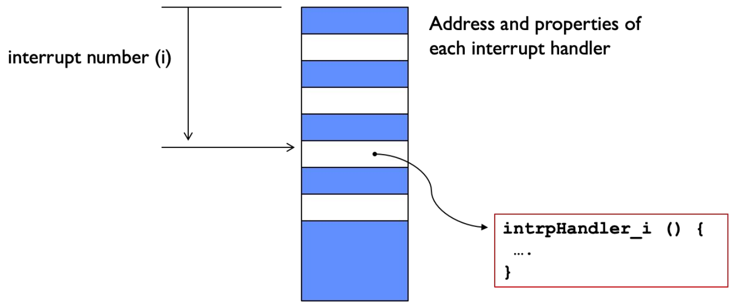
- **Kernel/supervisor/privileged mode** has the most privileges (i.e. kernel and other parts of OS operate in this mode).
- **User mode** prohibits certain operations (i.e. user programs execute)
- Restricted user mode is important to make sure user process cannot maliciously corrupt the system.

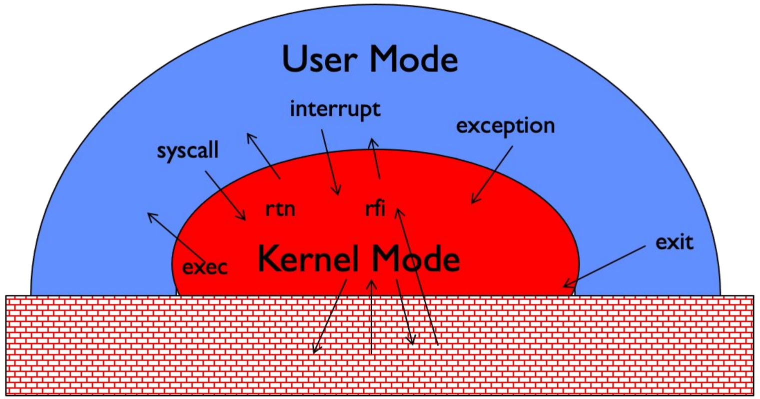
Three main ways of **mode transfers** (switch from user to kernel mode).

- Processes request a system service through a **system call (syscall)** which encompass functionality that require kernel mode privileges.
- **Interrupts** (hardware interrupts) are external asynchronous events (e.g. timer, I/O) that trigger a mode switch.
- **Traps** (software interrupts or **exceptions**) are internal synchronous events (e.g. segfault, divide by zero) that trigger a mode switch.

All three modes known as **unprogrammed control transfer**.

- Process doesn't identify the specific address but rather an index into the **interrupt vector table (IVT)** which contains the address and properties of each interrupt handler.





Concept Check

1. What is the importance of address translation?

2. Similar to what's done in the prologue at calling convention, what needs to happen before a mode transfer occurs?

Concept Check

1. **What is the importance of address translation?**

   Necessary for idea of virtual memory

   a. **Isolation/protection between different processes' address spaces**

   b. **Illusion to processor as sole user of address space**

2. Similar to what's done in the prologue at calling convention, what needs to happen before a mode transfer occurs?

Concept Check

1. What is the importance of address translation?

   Necessary for idea of virtual memory

   a. Isolation/protection between different processes' address spaces
   b. Illusion to processor as sole user of address space

2. **Similar to what's done in the prologue at calling convention, what needs to happen before a mode transfer occurs?**

   Need to save processor state (e.g. registers) in the thread control block (TCB) since kernel may overwrite it.

Concept Check

3. How does the syscall handler protect the kernel from corrupt or malicious user code?

4. Trivia: Contrary to the answer above, in Linux the /dev/kmem file, which contains the entirety of kernel virtual memory, can be read. Why do we let a user program read kernel memory?

Concept Check

3. **How does the syscall handler protect the kernel from corrupt or malicious user code?**

   User program specifies an index instead of direct address of the handler.

   Arguments are validated and copied over to kernel stack to prevent time-of-check to time-of-use (TOCTTOU) attacks.

   After the syscall finishes, the results are copied back in to user memory.

   The user process is not allowed to access the results stored in kernel memory for security reasons.

4. Trivia: Contrary to the answer above, in Linux the /dev/kmem file, which contains the entirety of kernel virtual memory, can be read. Why do we let a user program read kernel memory?

Concept Check

3. How does the syscall handler protect the kernel from corrupt or malicious user code?

   User program specifies an index instead of direct address of the handler.

   Arguments are validated and copied over to kernel stack to prevent time-of-check to time-of-use (TOCTTOU) attacks.

   After the syscall finishes, the results are copied back in to user memory.

   The user process is not allowed to access the results stored in kernel memory for security reasons.

4. **Trivia: Contrary to the answer above, in Linux the /dev/kmem file, which contains the entirety of kernel virtual memory, can be read. Why do we let a user program read kernel memory?**

   This isn't violating any of the OS principles of memory protection. Opening and reading files is a privileged operation, and you need to be running as a user with root privileges in the first place (`sudo` or *superuser*) that can make a syscall to read /dev/kmem.

Processes

Process Control Block

OS needs to run many programs, meaning it needs mechanisms such as

- Switching between user processes and the kernel.
- Switching among user processes through the kernel.
- Protecting the OS from user processes and protecting processes from each other.

Kernel represents each process with a **process control block (PCB)**.

Syscall

OS provides library/API that implements process management syscalls.

- Unix puts it as part of C standard library (libc).
- Use man pages for full documentation.

`void exit(int status)` terminates calling process with exit code `status`.

- Exit code 0 = no errors, nonzero means otherwise.
- Usually not explicitly called by `main` since OS implicitly calls it once `main` returns.

`pid_t fork(void)` creates a new process by copying the current process.

- Process created from `fork` is **child process**, process calling `fork` is **parent process**.
- Parent and child are identical (e.g. same address space) except for PID and a few other things.
- Return type is pid_t (signed integer).
  - > 0 means current process is parent.
  - = 0 means current process is child.
  - -1 means error has occurred

```c
int main() {
 pid_t fork_ret = fork();
 if (fork_ret > 0) {
   /* parent process logic */
 } else if (fork_ret == 0) {
   /* child process logic */
 } else {
   /* error handling */
 }
}
```

Typical `fork` workflow

Syscall

`exec` changes the program being run by the current process.

- *Does not create a new process like fork*.
- `exec` is a family of functions with different signatures.

`pid_t wait(int *wstatus)` waits for a child process to finish.

- Returns PID of terminated child process if successful, -1 on error.
- Store status information in `wstatus` if not `NULL`.

`int kill(pid_t pid, int sig)` sends a signal (interrupt-like notification) to another process.

- `SIGINT` (Ctrl - C), `SIGKILL` (`kill` on command line), `SIGSTOP` (Ctrl - Z).
- Signal handler defines the behavior when a process receives a signal.
- Custom signal handler can be written for most signals except `SIGKILL` and `SIGSTOP` using `sigaction`.

Concept Check

```c
int main(void) {
  int a = 1;
  pid_t fork_ret = fork();
  if (fork_ret) > 0 {
    a++;
    fprintf(stdout, "Parent: int a is %d at %p\n", a, &a);
  } else if (fork_ret == 0) {
    a++;
    fprintf(stdout, "Child: int a is %d at %p\n", a, &a);
  } else {
    printf("Oedipus");
  }
  return 0;
}
```

1. Will the parent and child print the same value for a?
2. Will they print the same address for a?
3. Will they even write to the same STDOUT?

Concept Check

```
int main(void) {
  int a = 1;
  pid_t fork_ret = fork();
  if (fork_ret) > 0 {
    a++;
    fprintf(stdout, "Parent: int a is %d at %p\n", a, &a);
  } else if (fork_ret == 0) {
    a++;
    fprintf(stdout, "Child: int a is %d at %p\n", a, &a);
  } else {
    printf("Oedipus");
  }
  return 0;
}
```

1. **Will the parent and child print the same value for a?**

   Yes. Processes do not share the same memory space, so a is 2 for both.

2. Will they print the same address for a?

3. Will they even write to the same STDOUT?

Concept Check

```c
int main(void) {
  int a = 1;
  pid_t fork_ret = fork();
  if (fork_ret) > 0 {
    a++;
    fprintf(stdout, "Parent: int a is %d at %p\n", a, &a);
  } else if (fork_ret == 0) {
    a++;
    fprintf(stdout, "Child: int a is %d at %p\n", a, &a);
  } else {
    printf("Oedipus");
  }
  return 0;
}
```

1. Will the parent and child print the same value for a?

   Yes. Processes do not share the same memory space, so a is 2 for both.

2. **Will they print the same address for a?**

   Yes. Fork copies the address space of the parent to the child.

3. Will they even write to the same STDOUT?

Concept Check

```c
int main(void) {
  int a = 1;
  pid_t fork_ret = fork();
  if (fork_ret) > 0 {
    a++;
    fprintf(stdout, "Parent: int a is %d at %p\n", a, &a);
  } else if (fork_ret == 0) {
    a++;
    fprintf(stdout, "Child: int a is %d at %p\n", a, &a);
  } else {
    printf("Oedipus");
  }
  return 0;
}
```

1. Will the parent and child print the same value for a?

   Yes. Processes do not share the same memory space, so a is 2 for both.

2. Will they print the same address for a?

   Yes. Fork copies the address space of the parent to the child.

3. **Will they even write to the same STDOUT?**

   Yes. File descriptors are copied over to the new process, so both STDOUTs will reference the same "file".

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Original Process	
`i`	`0`
`fork_ret`	

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 8	
`i`	`0`
`fork_ret`	`0`

Original Process	
`i`	`0`
`fork_ret`	`8`

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 8	
i	1
fork_ret	0

Original Process	
i	0
fork_ret	8

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 23	
`i`	1
`fork_ret`	0

Process 8	
`i`	1
`fork_ret`	23

Original Process	
`i`	0
`fork_ret`	8

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 23	
i	2
fork_ret	0

Process 8	
i	1
fork_ret	23

Original Process	
i	0
fork_ret	8

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 23	
i	2
fork_ret	25

Process 25	
i	2
fork_ret	0

Process 8	
i	1
fork_ret	23

Original Process	
i	0
fork_ret	8

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Original Process	
i	0
fork_ret	8

Process 8	
i	1
fork_ret	23

Process 23	
i	2
fork_ret	25

Process 25	
i	3
fork_ret	0

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Original Process	
i	0
fork_ret	8

Process 8	
i	1
fork_ret	23

Process 23	
i	2
fork_ret	25

Process 25	
i	3
fork_ret	0

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Original Process	
i	0
fork_ret	8

Process 8	
i	1
fork_ret	23

Process 23	
i	3
fork_ret	25

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 8	
i	1
fork_ret	23

Process 23	
i	3
fork_ret	25

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	0
fork_ret	8

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Original Process	
i	0
fork_ret	8

Process 8	
i	2
fork_ret	23

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 23	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

Process 25	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Process 8	
i	2
fork_ret	5

Process 5	
i	2
fork_ret	0

Original Process	
i	0
fork_ret	8

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 8	
i	2
fork_ret	5

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Process 5	
i	3
fork_ret	0

Original Process	
i	0
fork_ret	8

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 23	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

Process 25	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Process 8	
i	2
fork_ret	5

Process 5	
i	3
fork_ret	0

Original Process	
i	0
fork_ret	8

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 8	
i	3
fork_ret	5

Original Process	
i	0
fork_ret	8

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 23	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

Process 25	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Process 8	
i	3
fork_ret	5

Process 5	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	0
fork_ret	8

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 8	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

Process 23	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

Process 25	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Process 5	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	1
fork_ret	8

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 23	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

Process 25	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Process 8	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

Process 5	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	1
fork_ret	16

Process 16	
i	1
fork_ret	0

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 23	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

Process 25	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Process 8	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

Process 5	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	1
fork_ret	16

Process 16	
i	2
fork_ret	0

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 8~~	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	1
fork_ret	16

Process 16	
i	2
fork_ret	19

Process 19	
i	2
fork_ret	0

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 8~~	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	1
fork_ret	16

Process 16	
i	2
fork_ret	19

Process 19	
i	3
fork_ret	0

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 8~~	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	1
fork_ret	16

Process 16	
i	2
fork_ret	19

Process 19	
i	3
fork_ret	0

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Original Process	
i	1
fork_ret	16

Process 8	
i	3
fork_ret	5

Process 16	
i	3
fork_ret	19

Process 23	
i	3
fork_ret	25

Process 5	
i	3
fork_ret	0

Process 19	
i	3
fork_ret	0

Process 25	
i	3
fork_ret	0

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 23	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

Process 25	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Process 8	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

Process 5	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	1
fork_ret	16

Process 16	
i	3
fork_ret	19

Process 19	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Process 23	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

Process 25	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Process 8	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

Process 5	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	2
fork_ret	16

Process 16	
~~i~~	~~3~~
~~fork_ret~~	~~19~~

Process 19	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Original Process	
i	2
fork_ret	98

~~Process 8~~	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

~~Process 16~~	
~~i~~	~~3~~
~~fork_ret~~	~~19~~

Process 98	
i	2
fork_ret	0

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 19~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 8~~	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Original Process	
i	2
fork_ret	98

~~Process 16~~	
~~i~~	~~3~~
~~fork_ret~~	~~19~~

~~Process 19~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Process 98	
i	3
fork_ret	0

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Original Process	
i	2
fork_ret	98

~~Process 8~~	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

~~Process 16~~	
~~i~~	~~3~~
~~fork_ret~~	~~19~~

Process 98	
i	3
fork_ret	0

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 19~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Original Process	
i	2
fork_ret	98

~~Process 8~~	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

~~Process 16~~	
~~i~~	~~3~~
~~fork_ret~~	~~19~~

~~Process 98~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 19~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

Original Process	
i	3
fork_ret	98

~~Process 8~~	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

~~Process 16~~	
~~i~~	~~3~~
~~fork_ret~~	~~19~~

~~Process 98~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 19~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

~~Original Process~~	
~~i~~	~~3~~
~~fork_ret~~	~~98~~

~~Process 8~~	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

~~Process 16~~	
~~i~~	~~3~~
~~fork_ret~~	~~19~~

~~Process 98~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 19~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```c
int main(void) {
  for (int i = 0; i < 3; i++)
    pid_t fork_ret = fork();
  return 0;
}
```

1. How many new processes (not including the original process) are created when the following program is run? Assume all `fork` calls succeed.

7 process in addition to the original process.

~~Original Process~~	
~~i~~	~~3~~
~~fork_ret~~	~~98~~

~~Process 8~~	
~~i~~	~~3~~
~~fork_ret~~	~~5~~

~~Process 16~~	
~~i~~	~~3~~
~~fork_ret~~	~~19~~

~~Process 98~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 23~~	
~~i~~	~~3~~
~~fork_ret~~	~~25~~

~~Process 5~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 19~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

~~Process 25~~	
~~i~~	~~3~~
~~fork_ret~~	~~0~~

Fork and Friends

```c
int main(void) {
  int* stuff = malloc(sizeof(int));
  *stuff = 5;
  pid_t fork_ret = fork();
  printf("The last digit of pi is %d\n", *stuff);
  if (fork_ret == 0)
    *stuff = 6;
  return 0;
}
```

3. What are the possible outputs when the following program is run?

Fork and Friends

```c
int main(void) {
  int* stuff = malloc(sizeof(int));
  *stuff = 5;
  pid_t fork_ret = fork();
  printf("The last digit of pi is %d\n", *stuff);
  if (fork_ret == 0)
    *stuff = 6;
  return 0;
}
```

3. What are the possible outputs when the following program is run?

Heap is part of the address space like the stack, so output is the same as previous question.

If fork succeeds, then

```
The last digit of pi is 5.
The last digit of pi is 5.
```

Otherwise, only one line is printed.

Fork and Friends

```c
int main(void) {
  pid_t fork_ret = fork();
  int exit;
  if (fork_ret != 0)
    wait(&exit);
  printf("Hello World: %d\n", fork_ret);
  return 0;
}
```

4. What are the possible outputs when the following program is run? Assume the child process has PID 162162.

Fork and Friends

```
int main(void) {
  pid_t fork_ret = fork();
  int exit;
  if (fork_ret != 0)
    wait(&exit);
  printf("Hello World: %d\n", fork_ret);
  return 0;
}
```

4. What are the possible outputs when the following program is run? Assume the child process has PID 162162.

Parent process will wait until child process completes, so it won't print before child prints.

```
Hello World: 0
Hello World: 162162
```

If `fork` fails, then program will print

```
Hello World: -1
```

since `fork` will return -1. Note that wait(&exit) when fork() fails will return immediately.

Fork and Friends

```c
int main(void) {
  char** argv = (char**)malloc(3 * sizeof(char*));
  argv[0] = "/bin/ls";
  argv[1] = ".";
  argv[2] = NULL;
  for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
    if (i == 3) {
      execv("/bin/ls", argv);
    }
  }
  return 0;
}
```

5. Does the following program print all numbers from 0 to 9 as well as the output of running `ls`? If not, what is the minimal code change to accomplish this? Assume all syscalls succeed.

Fork and Friends

```c
int main(void) {
  char** argv = (char**)malloc(3 * sizeof(char*));
  argv[0] = "/bin/ls";
  argv[1] = ".";
  argv[2] = NULL;
  for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
    if (i == 3) {
      execv("/bin/ls", argv);
    }
  }
  return 0;
}
```

5. Does the following program print all numbers from 0 to 9 as well as the output of running `ls`? If not, what is the minimal code change to accomplish this? Assume all syscalls succeed.

Currently, program stops after printing 3, giving an output of

```
0
1
2
3
<output of ls>
```

since execv overwrites the entire process image (i.e. rest of loop will not execute).

Fork and Friends

```c
int main(void) {
  char** argv = (char**)malloc(3 * sizeof(char*));
  argv[0] = "/bin/ls";
  argv[1] = ".";
  argv[2] = NULL;
  for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
    if (i == 3) {
      pid_t fork_ret = fork();
      if (fork_ret == 0)
        execv("/bin/ls", argv);
    }
  }
  return 0;
}
```

5. Does the following program print all numbers from 0 to 9 as well as the output of running `ls`? If not, what is the minimal code change to accomplish this? Assume all syscalls succeed.

Currently, program stops after printing 3, giving an output of

```
0
1
2
3
<output of ls>
```

since execv overwrites the entire process image (i.e. rest of loop will not execute).

Fork and exec in child to make sure parent process continues the loop.

Signal Handling

Signal	Value	Action	Comment
SIGHUP	1	Terminate	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Terminate	Interrupt from keyboard (Ctrl - c)
SIGQUIT	3	Core Dump	Quit from keyboard (Ctrl - \)
SIGILL	4	Core Dump	Illegal Instruction
SIGABRT	6	Core Dump	Abort signal from abort(3)
SIGFPE	8	Core Dump	Floating point exception
SIGKILL	9	Terminate	Kill signal
SIGSEGV	11	Core Dump	Invalid memory reference
SIGPIPE	13	Terminate	Broken pipe: write to pipe with no readers
SIGALRM	14	Terminate	Timer signal from alarm(2)
SIGTERM	15	Terminate	Termination signal
SIGUSR1	30,10,16	Terminate	User-defined signal 1
SIGUSR2	31,12,17	Terminate	User-defined signal 2
SIGCHLD	20,17,18	Ignore	Child stopped or terminated
SIGCONT	19,18,25	Continue	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Signal Handling

1. Overriding SIGSTOP and SIGKILL is disabled. Why?

Signal Handling

1. Overriding SIGSTOP and SIGKILL is disabled. Why?

   If a process were to override their signal handlers to ignore `SIGSTOP` and `SIGKILL`, it can run a malicious process forever.

Signal Handling

```c
void sigquit_handler(int sig) {
  if (sig == SIGINT || sig == SIGQUIT)
    exit(1);
}
void sigint_handler(int sig) {
  if (sig == SIGINT)
    signal(SIGINT, sigquit_handler);
}
int main() {
  signal(SIGQUIT, sigquit_handler);
  signal(SIGINT, sigint_handler);
  while (1) {
    printf("Sleeping for a second ...\n");
    sleep(1);
  }
}
```

2. What are the different ways you can use the keyboard to cause the program to exit? Assume program is run in a bash cell.

Signal Handling

```
void sigquit_handler(int sig) {
  if (sig == SIGINT || sig == SIGQUIT)
    exit(1);
}
void sigint_handler(int sig) {
  if (sig == SIGINT)
    signal(SIGINT, sigquit_handler);
}
int main() {
  signal(SIGQUIT, sigquit_handler);
  signal(SIGINT, sigint_handler);
  while (1) {
    printf("Sleeping for a second ...\n");
    sleep(1);
  }
}
```

2. What are the different ways you can use the keyboard to cause the program to exit? Assume program is run in a bash cell.

`main` initialized `SIGINT` and `SIGQUIT` to custom handlers

Ctrl - \ will be routed to sigquit_handler which exits program upon `SIGQUIT`.

Ctrl - C will be routed to sigint_handler which redefines `SIGINT` handler to sigquit_handler.

- Pressing Ctrl - C or Ctrl - \ after Ctrl-C will exit program.

`SIGSTOP` handler is not redefined → Ctrl-Z will exit program.

Pintos Lists

Regular Linked Lists

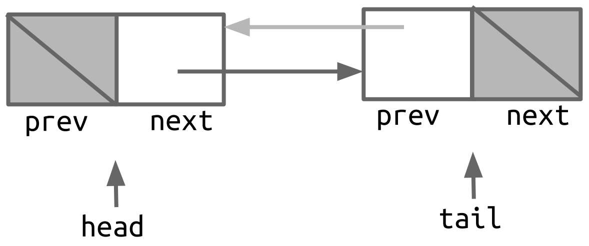
```
struct ll_node {
 int value;
 struct ll_node* next;
};

/* Returns the sum of a linked list. */
int ll_sum(ll_node* start) {
 ll_node* iter;

 int total = 0;
 for (iter = start; iter != NULL; iter = iter->next)
   total += iter->value;

 return total;
}
```

Pintos (Doubly) Linked Lists

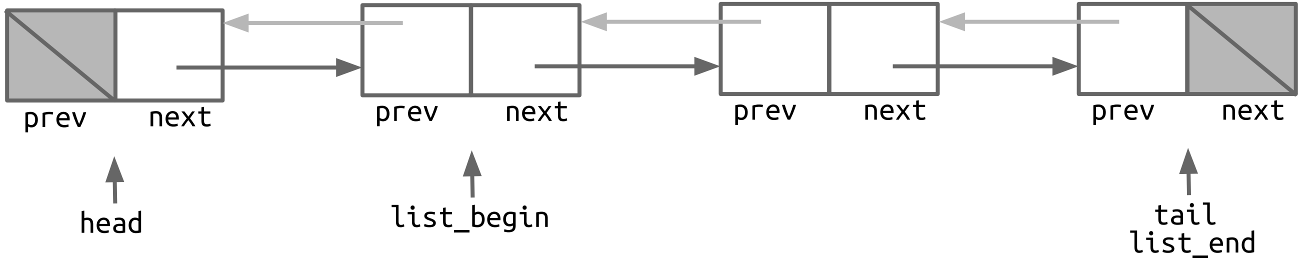
```
/* List element. */
struct list_elem {
  struct list_elem* prev; /* Previous list element. */
  struct list_elem* next; /* Next list element. */
};

/* List. */
struct list {
  struct list_elem head; /* List head. */
  struct list_elem tail; /* List tail. */
};


/* Given a struct list, returns a reference to the
   first list_elem in the list. */
struct list_elem* list_begin(struct list* lst);

/* Given a struct list, returns a reference to the
   last list_elem in the list. */
struct list_elem* list_end(struct list* lst);

/* Given a liter = list_begin(lst)ist_elem, finds the next
   list_elem in the list. */
struct list_elem* list_next(struct list_elem* elem);

/* Converts pointer to list element LIST_ELEM into
   a pointer to the structure that LIST_ELEM is
   embedded inside. You must also provide the name
   of the outer structure STRUCT and the member
   name MEMBER of the list element. */
STRUCT* list_entry(LIST_ELEM, STRUCT, MEMBER);
```

Pintos (Doubly) Linked Lists

Empty Pintos list:

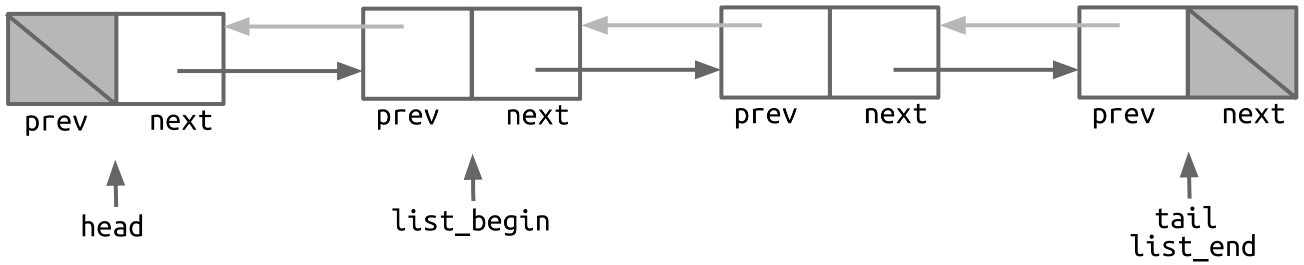


Pintos (Doubly) Linked Lists



Technically, both `list_begin` and `list_end` will give you the tail for the empty Pintos list

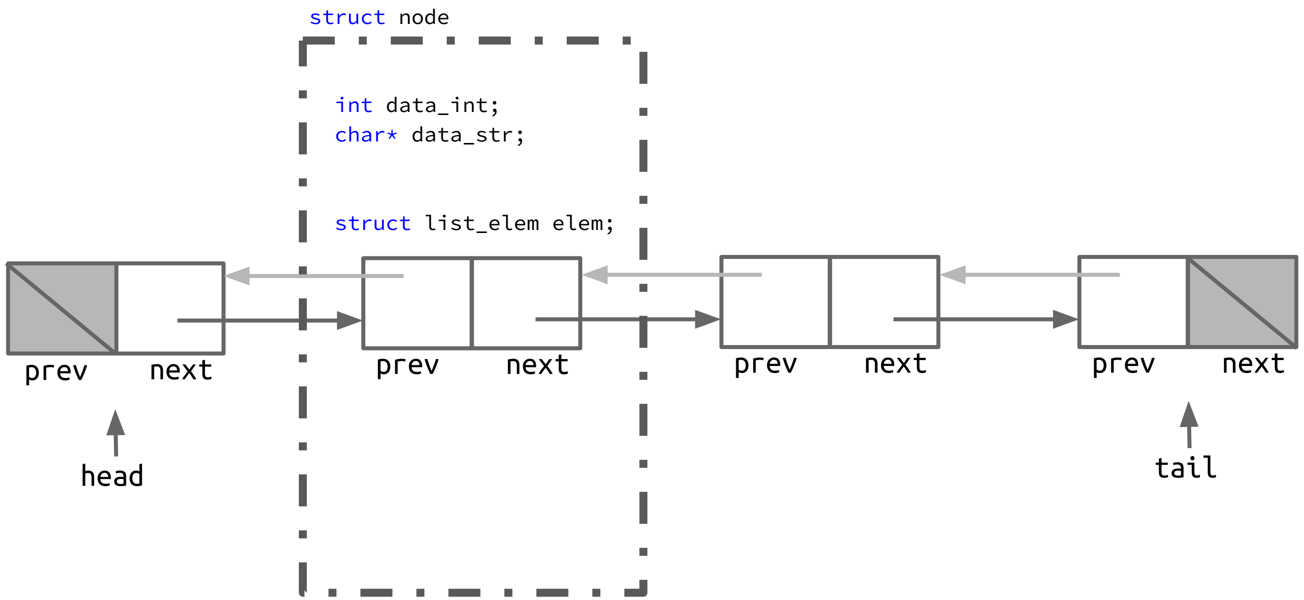
Pintos (Doubly) Linked Lists



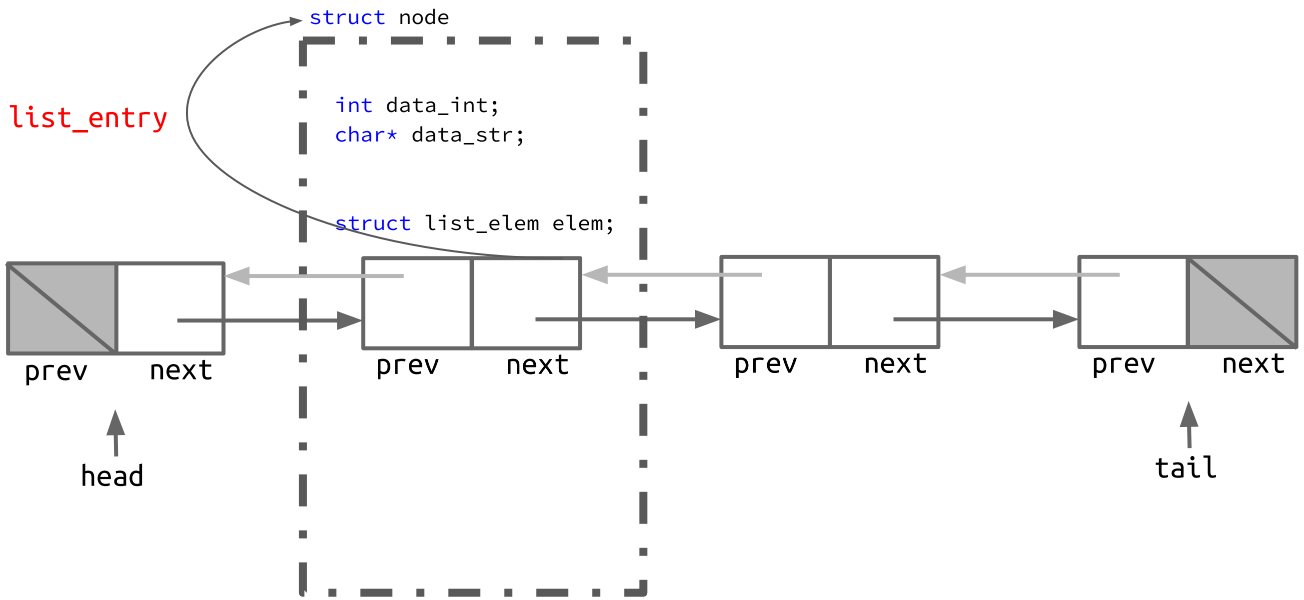
Technically, both `list_begin` and `list_end` will give you the tail for the empty Pintos list

Anyway, how is this useful?

Pintos (Doubly) Linked Lists



Pintos (Doubly) Linked Lists



1. Avoid having to allocate memory for a separate data structure
2. Generic, easy-to-use interface (we'll demonstrate this in the problem)

Pintos Lists

```c
struct list_data {
  char* name;
  struct list pl_list;
};

struct pl_node {
  int value;
  struct list_elem elem;
};

/* Returns the sum of a pintos-style list of pl_nodes. */
int pl_sum(struct list_data* data) {
  struct list_elem* iter;
  struct pl_node* temp;
  struct list* lst = ____________;

  int total = 0;

  for (______________________; _____________________; ______________________) {
    temp = list_entry(__________________________);
    ____________________;
  }

  return total;
}
```

Fill in the blanks to sum up the elements of a Pintos list.

```c
/* Given a struct list, returns a reference to the
   first list_elem in the list. */
struct list_elem* list_begin(struct list* lst);

/* Given a struct list, returns a reference to the
   last list_elem in the list. */
struct list_elem* list_end(struct list* lst);

/* Given a list_elem, finds the next list_elem in
   the list. */
struct list_elem* list_next(struct list_elem* elem);

/* Converts pointer to list element LIST_ELEM into
   a pointer to the structure that LIST_ELEM is
   embedded inside. You must also provide the name
   of the outer structure STRUCT and the member
   name MEMBER of the list element. */
STRUCT* list_entry(LIST_ELEM, STRUCT, MEMBER);
```

Pintos Lists

```c
struct list_data {
  char* name;
  struct list pl_list;
};

struct pl_node {
  int value;
  struct list_elem elem;
};

/* Returns the sum of a pintos-style list of pl_nodes. */
int pl_sum(struct list_data* data) {
  struct list_elem* iter;
  struct pl_node* temp;
  struct list* lst = &data->pl_list;

  int total = 0;

  for (______________________; _____________________; ______________________) {
    temp = list_entry(__________________________);
    ____________________;
  }

  return total;
}
```

Fill in the blanks to sum up the elements of a Pintos list.

```c
/* Given a struct list, returns a reference to the
   first list_elem in the list. */
struct list_elem* list_begin(struct list* lst);

/* Given a struct list, returns a reference to the
   last list_elem in the list. */
struct list_elem* list_end(struct list* lst);

/* Given a list_elem, finds the next list_elem in
   the list. */
struct list_elem* list_next(struct list_elem* elem);

/* Converts pointer to list element LIST_ELEM into
   a pointer to the structure that LIST_ELEM is
   embedded inside. You must also provide the name
   of the outer structure STRUCT and the member
   name MEMBER of the list element. */
STRUCT* list_entry(LIST_ELEM, STRUCT, MEMBER);
```

Pintos Lists

```
struct list_data {
  char* name;
  struct list pl_list;
};

struct pl_node {
  int value;
  struct list_elem elem;
};

/* Returns the sum of a pintos-style list of pl_nodes. */
int pl_sum(struct list_data* data) {
  struct list_elem* iter;
  struct pl_node* temp;
  struct list* lst = &data->pl_list;

  int total = 0;

  for (iter = list_begin(lst); _____________________; ______________________) {
    temp = list_entry(__________________________);
    ____________________;
  }

  return total;
}
```

Fill in the blanks to sum up the elements of a Pintos list.

```
/* Given a struct list, returns a reference to the
   first list_elem in the list. */
struct list_elem* list_begin(struct list* lst);

/* Given a struct list, returns a reference to the
   last list_elem in the list. */
struct list_elem* list_end(struct list* lst);

/* Given a liter = list_begin(lst)ist_elem, finds the
next list_elem in
   the list. */
struct list_elem* list_next(struct list_elem* elem);

/* Converts pointer to list element LIST_ELEM into
   a pointer to the structure that LIST_ELEM is
   embedded inside. You must also provide the name
   of the outer structure STRUCT and the member
   name MEMBER of the list element. */
STRUCT* list_entry(LIST_ELEM, STRUCT, MEMBER);
```

Pintos Lists

```
struct list_data {
  char* name;
  struct list pl_list;
};

struct pl_node {
  int value;
  struct list_elem elem;
};

/* Returns the sum of a pintos-style list of pl_nodes. */
int pl_sum(struct list_data* data) {
  struct list_elem* iter;
  struct pl_node* temp;
  struct list* lst = &data->pl_list;

  int total = 0;

  for (iter = list_begin(lst); _____________________; ______________________) {
    temp = list_entry(__________________________);
    ____________________;
  }

  return total;
}
```

Fill in the blanks to sum up the elements of a Pintos list.

```
/* Given a struct list, returns a reference to the
   first list_elem in the list. */
struct list_elem* list_begin(struct list* lst);

/* Given a struct list, returns a reference to the
   last list_elem in the list. */
struct list_elem* list_end(struct list* lst);

/* Given a liter = list_begin(lst)ist_elem, finds the
next list_elem in
   the list. */
struct list_elem* list_next(struct list_elem* elem);

/* Converts pointer to list element LIST_ELEM into
   a pointer to the structure that LIST_ELEM is
   embedded inside. You must also provide the name
   of the outer structure STRUCT and the member
   name MEMBER of the list element. */
STRUCT* list_entry(LIST_ELEM, STRUCT, MEMBER);
```

Pintos Lists

```c
struct list_data {
  char* name;
  struct list pl_list;
};

struct pl_node {
  int value;
  struct list_elem elem;
};

/* Returns the sum of a pintos-style list of pl_nodes. */
int pl_sum(struct list_data* data) {
  struct list_elem* iter;
  struct pl_node* temp;
  struct list* lst = &data->pl_list;

  int total = 0;

  for (iter = list_begin(lst); iter != list_end(lst); ______________________) {
    temp = list_entry(__________________________);
    ____________________;
  }

  return total;
}
```

Fill in the blanks to sum up the elements of a Pintos list.

```c
/* Given a struct list, returns a reference to the
   first list_elem in the list. */
struct list_elem* list_begin(struct list* lst);

/* Given a struct list, returns a reference to the
   last list_elem in the list. */
struct list_elem* list_end(struct list* lst);

/* Given a liter = list_begin(lst)ist_elem, finds the
next list_elem in
   the list. */
struct list_elem* list_next(struct list_elem* elem);

/* Converts pointer to list element LIST_ELEM into
   a pointer to the structure that LIST_ELEM is
   embedded inside. You must also provide the name
   of the outer structure STRUCT and the member
   name MEMBER of the list element. */
STRUCT* list_entry(LIST_ELEM, STRUCT, MEMBER);
```

Pintos Lists

```c
struct list_data {
  char* name;
  struct list pl_list;
};

struct pl_node {
  int value;
  struct list_elem elem;
};

/* Returns the sum of a pintos-style list of pl_nodes. */
int pl_sum(struct list_data* data) {
  struct list_elem* iter;
  struct pl_node* temp;
  struct list* lst = &data->pl_list;

  int total = 0;

  for (iter = list_begin(lst); iter != list_end(lst); iter = list_next(iter)) {
    temp = list_entry(__________________________);
    ____________________;
  }

  return total;
}
```

Fill in the blanks to sum up the elements of a Pintos list.

```c
/* Given a struct list, returns a reference to the
   first list_elem in the list. */
struct list_elem* list_begin(struct list* lst);

/* Given a struct list, returns a reference to the
   last list_elem in the list. */
struct list_elem* list_end(struct list* lst);

/* Given a liter = list_begin(lst)ist_elem, finds the
next list_elem in
   the list. */
struct list_elem* list_next(struct list_elem* elem);

/* Converts pointer to list element LIST_ELEM into
   a pointer to the structure that LIST_ELEM is
   embedded inside. You must also provide the name
   of the outer structure STRUCT and the member
   name MEMBER of the list element. */
STRUCT* list_entry(LIST_ELEM, STRUCT, MEMBER);
```

Pintos Lists

```c
struct list_data {
  char* name;
  struct list pl_list;
};

struct pl_node {
  int value;
  struct list_elem elem;
};

/* Returns the sum of a pintos-style list of pl_nodes. */
int pl_sum(struct list_data* data) {
  struct list_elem* iter;
  struct pl_node* temp;
  struct list* lst = &data->pl_list;

  int total = 0;

  for (iter = list_begin(lst); iter != list_end(lst); iter = list_next(iter)) {
    temp = list_entry(iter, struct pl_node, elem);
    ____________________;
  }

  return total;
}
```

Fill in the blanks to sum up the elements of a Pintos list.

```c
/* Given a struct list, returns a reference to the
   first list_elem in the list. */
struct list_elem* list_begin(struct list* lst);

/* Given a struct list, returns a reference to the
   last list_elem in the list. */
struct list_elem* list_end(struct list* lst);

/* Given a liter = list_begin(lst)ist_elem, finds the
next list_elem in
   the list. */
struct list_elem* list_next(struct list_elem* elem);

/* Converts pointer to list element LIST_ELEM into
   a pointer to the structure that LIST_ELEM is
   embedded inside. You must also provide the name
   of the outer structure STRUCT and the member
   name MEMBER of the list element. */
STRUCT* list_entry(LIST_ELEM, STRUCT, MEMBER);
```

Pintos Lists

```c
struct list_data {
  char* name;
  struct list pl_list;
};

struct pl_node {
  int value;
  struct list_elem elem;
};

/* Returns the sum of a pintos-style list of pl_nodes. */
int pl_sum(struct list_data* data) {
  struct list_elem* iter;
  struct pl_node* temp;
  struct list* lst = &data->pl_list;

  int total = 0;

  for (iter = list_begin(lst); iter != list_end(lst); iter = list_next(iter)) {
    temp = list_entry(iter, struct pl_node, elem);
    total += temp->value;
  }

  return total;
}
```

Fill in the blanks to sum up the elements of a Pintos list.

```c
/* Given a struct list, returns a reference to the
   first list_elem in the list. */
struct list_elem* list_begin(struct list* lst);

/* Given a struct list, returns a reference to the
   last list_elem in the list. */
struct list_elem* list_end(struct list* lst);

/* Given a liter = list_begin(lst)ist_elem, finds the
next list_elem in
   the list. */
struct list_elem* list_next(struct list_elem* elem);

/* Converts pointer to list element LIST_ELEM into
   a pointer to the structure that LIST_ELEM is
   embedded inside. You must also provide the name
   of the outer structure STRUCT and the member
   name MEMBER of the list element. */
STRUCT* list_entry(LIST_ELEM, STRUCT, MEMBER);
```