Writing Xv6 System Calls

the last scheduling lecture

Dr. Naser Al Madi

Learning objectives

- fork + wait + exit
- Go over Xv6 and qemu
- Write hello world in Xv6
- Write PS system call together
- Go over project 4

Next week we will start with multiprocessing and multithreading in Python and C!

Final exam: Friday, May 13 at 6 pm

Fork, Wait, Exit system calls

How processes get created?

fork() is a system call that allows a process to create a clone (identical copy) of itself.

Actual fork() code:

Parent

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

int main()
{
    fork();
    printf("hello\n");
    return 0;
}
```

Actual fork() code:

Parent

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

int main()
{
    fork();
    printf("hello\n");
    return 0;
}
```

Actual fork() code:

Parent

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

int main()
{
    fork();
    printf("hello\n");
    return 0;
}
```

Actual fork() code:

Parent	Child

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

int main()
{
    fork();
    printf("hello\n");
    return 0;
}
```

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

int main()
{
    fork();
    printf("hello\n");
    return 0;
}
```

Actual fork() code:

Parent	Child

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

int main()
{
    fork();
    printf("hello\n");
    return 0;
}
```

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

int main()
{
    fork();
    printf("hello\n");
    return 0;
}
```

```
>hello
>hello
```

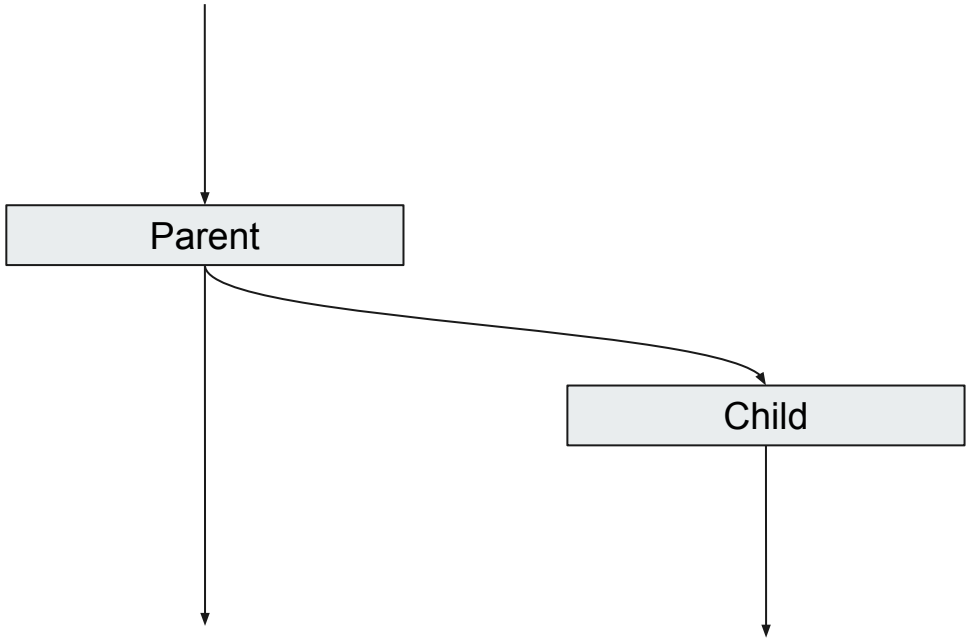
Actual fork() code:

Parent	Child

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

int main()
{
    fork();
    printf("hello\n");
    return 0;
}
```

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

int main()
{
    fork();
    printf("hello\n");
    return 0;
}
```

```
>hello
>hello
```


Parent
Child

Actual fork() code:

Parent

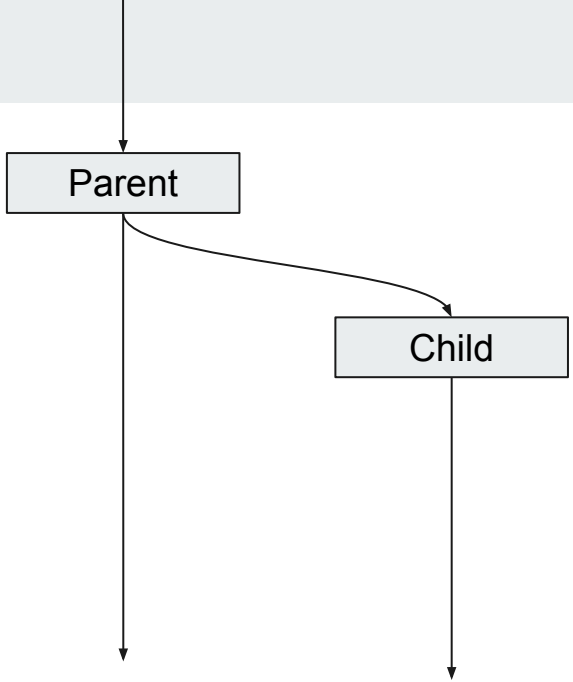
```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

>what will be printed?

Actual fork() code:

Parent

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

int main()
{
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

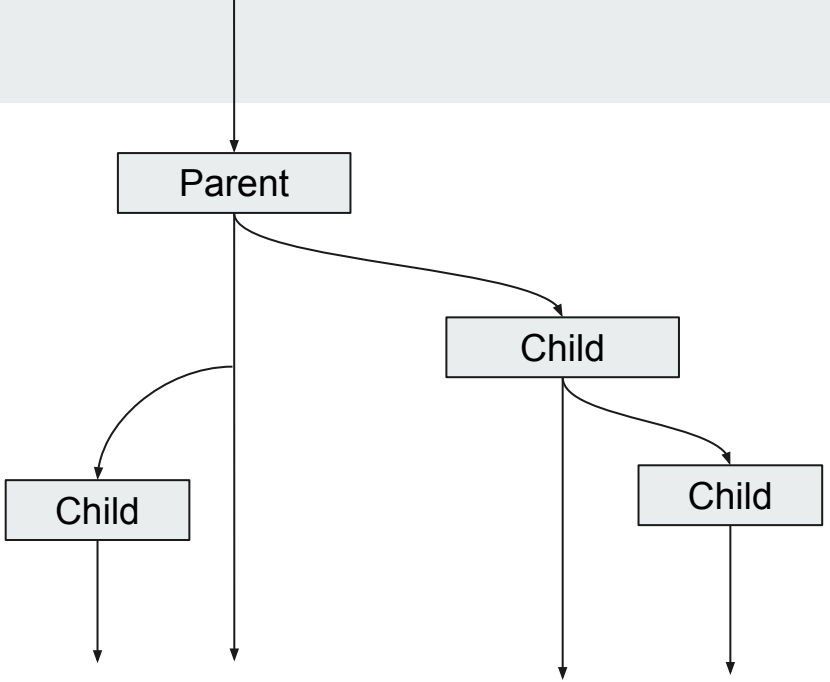


>what will be printed?

Actual fork() code:

Parent

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

int main()
{
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

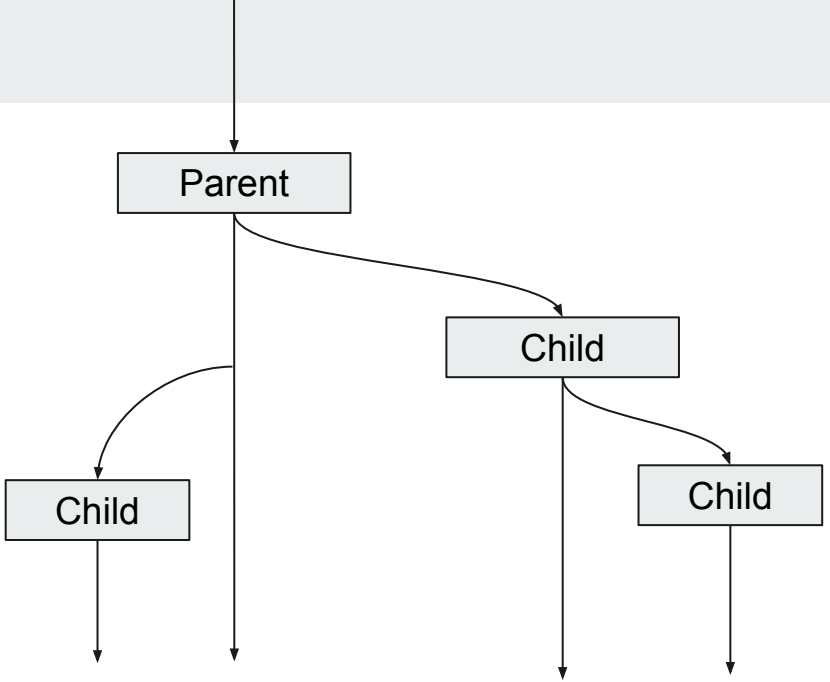


>what will be printed?

Actual fork() code:

Parent

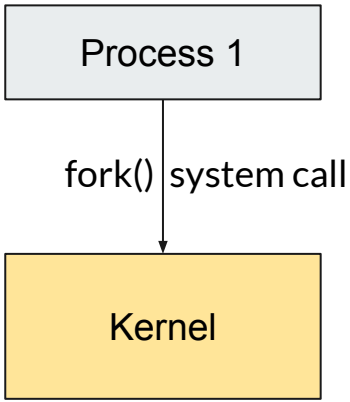
```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```




```
>hello
>hello
>hello
>hello
```

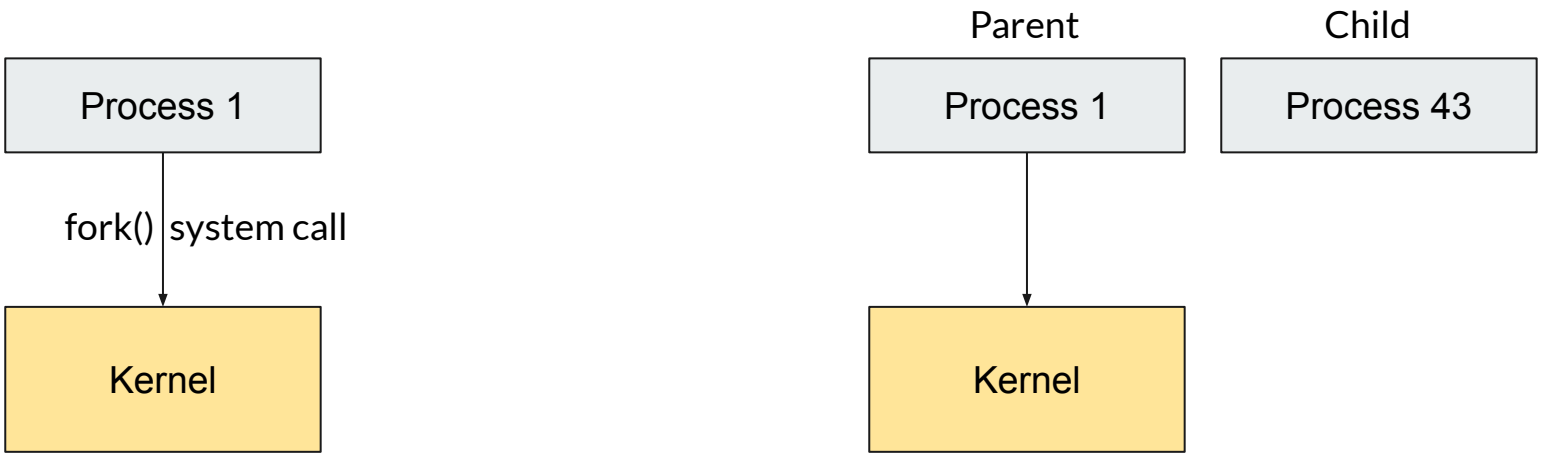
How processes get created?

fork() is a system call that allows a process to create a clone (identical copy) of itself.



How processes get created?

fork() is a system call that allows a process to create a clone (identical copy) of itself.



Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

Child

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = child pid

Child

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = child pid

Child

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = 0

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = child pid

Child

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = 0

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = child pid

Child

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = 0

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = child pid

Child

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = 0

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = child pid

Child

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = 0

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = child pid

Child

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = 0

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = child pid

Child

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = 0

hello, I am child (pid:123)

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = child pid

Child

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = 0

hello, I am child (pid:123)

Actual fork() code:

Parent

```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = child pid

hello, I am parent of 123 (pid:1)

Child

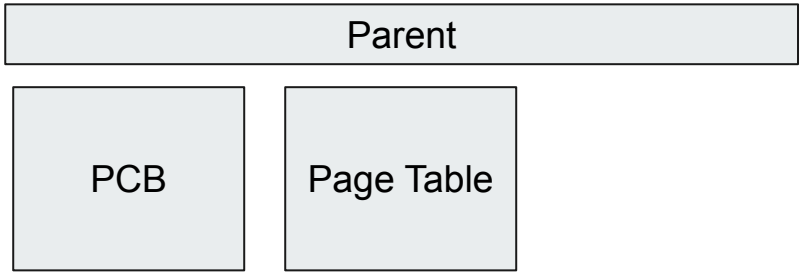
```
printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int)
getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
```

rc = 0

hello, I am child (pid:123)

How does that work from kernel perspective?

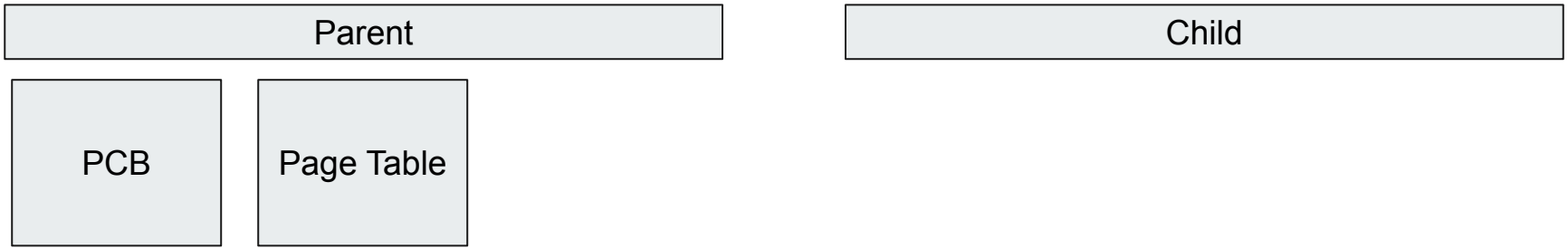
How does that work from kernel perspective?



When fork is executed:

- A PCB is created for the child process.
- An identical page table is created.

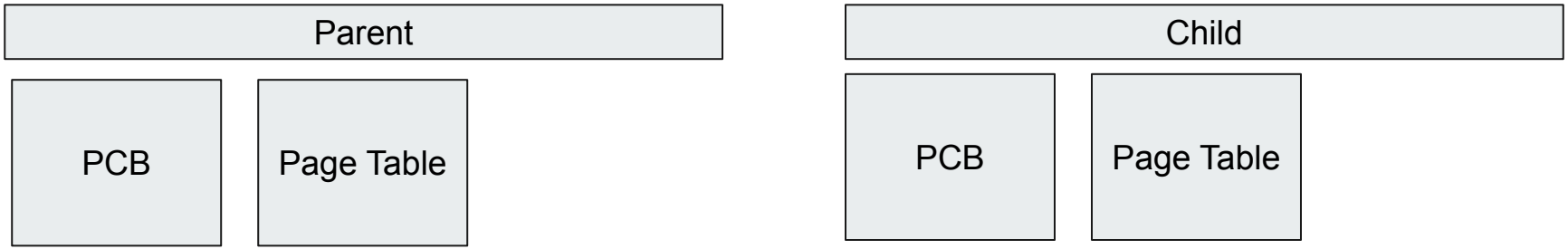
How does that work from kernel perspective?



When fork is executed:

- A PCB is created for the child process.
- An identical page table is created.

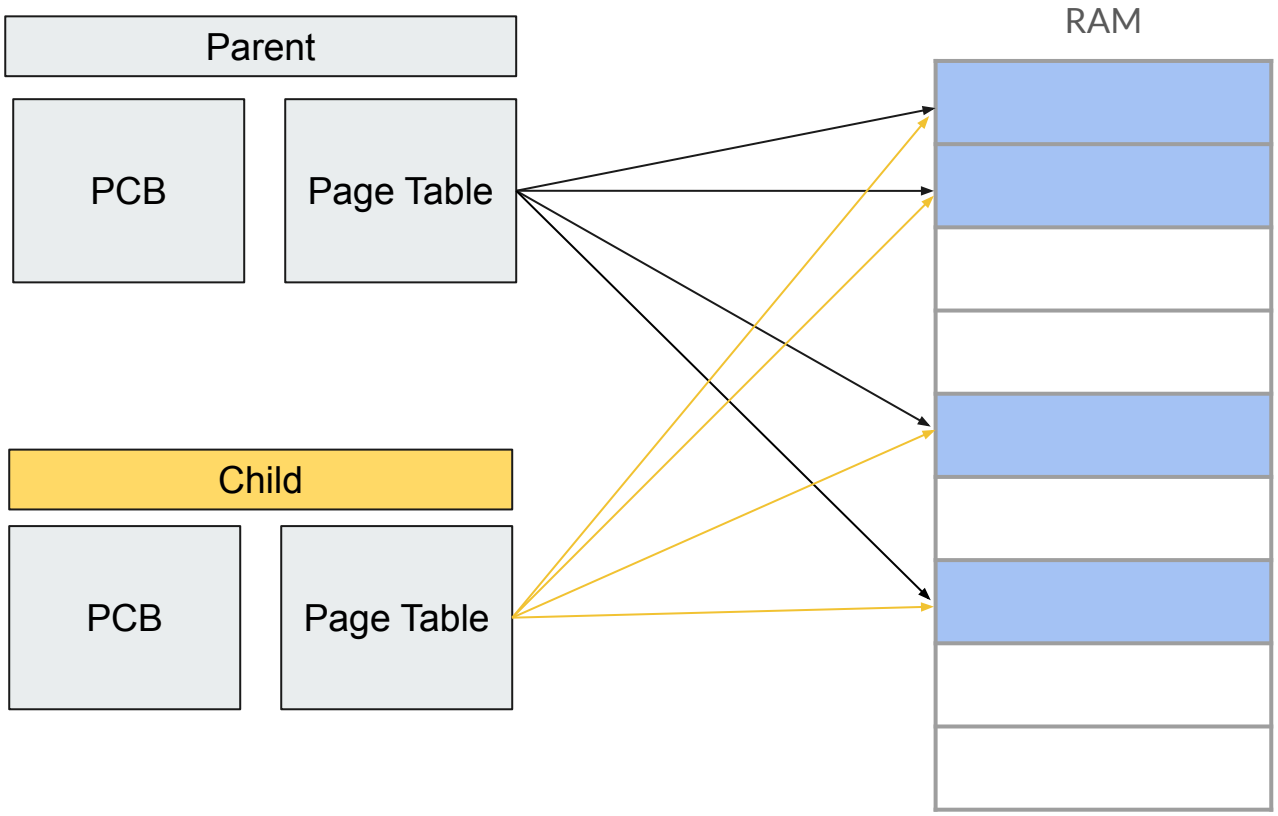
How does that work from kernel perspective?



When fork is executed:

- A PCB is created for the child process.
- An identical page table is created.

How does the identical page table work?



Parent and Child share the same pages in memory

Motivating example:

```
int i = 23, pid;

pid = fork()
if (pid > 0){
   sleep(1);
   printf("parent: %d\n", i);
   wait();      //parent process waits for child to finish
}
else{
   printf("child: %d\n", i);
}
```

Motivating example:

```
int i = 23, pid;

pid = fork()
if (pid > 0){
   sleep(1);
   printf(“parent: %d\n”, i);
   wait();      //parent process waits for child to finish
}
else{
   printf(“child: %d\n”, i);
}
```

```
> child: 23
>Parent: 23
```

Motivating example 2:

```
int i = 23, pid;

pid = fork()
if (pid > 0){
   sleep(1);
   printf(“parent: %d\n”, i);
   wait();      //parent process waits for child to finish
}
else{
   i = i + 1;
   printf(“child: %d\n”, i);
}
```

Motivating example 2:

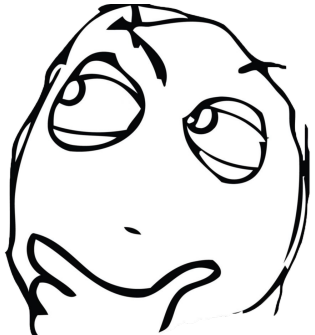
```
int i = 23, pid;

pid = fork()
if (pid > 0){
   sleep(1);
   printf("parent: %d\n", i);
   wait();      //parent process waits for child to finish
}
else{
   i = i + 1;
   printf("child: %d\n", i);
}
```

WHAT IS THE OUTPUT?

Motivating example 2:

```
int i = 23, pid;

pid = fork()
if (pid > 0){
   sleep(1);
   printf(“parent: %d\n”, i);
   wait();      //parent process waits for child to finish
}
else{
   i = i + 1;
   printf(“child: %d\n”, i);
}
```

```
> child: 24
>Parent: 23
```



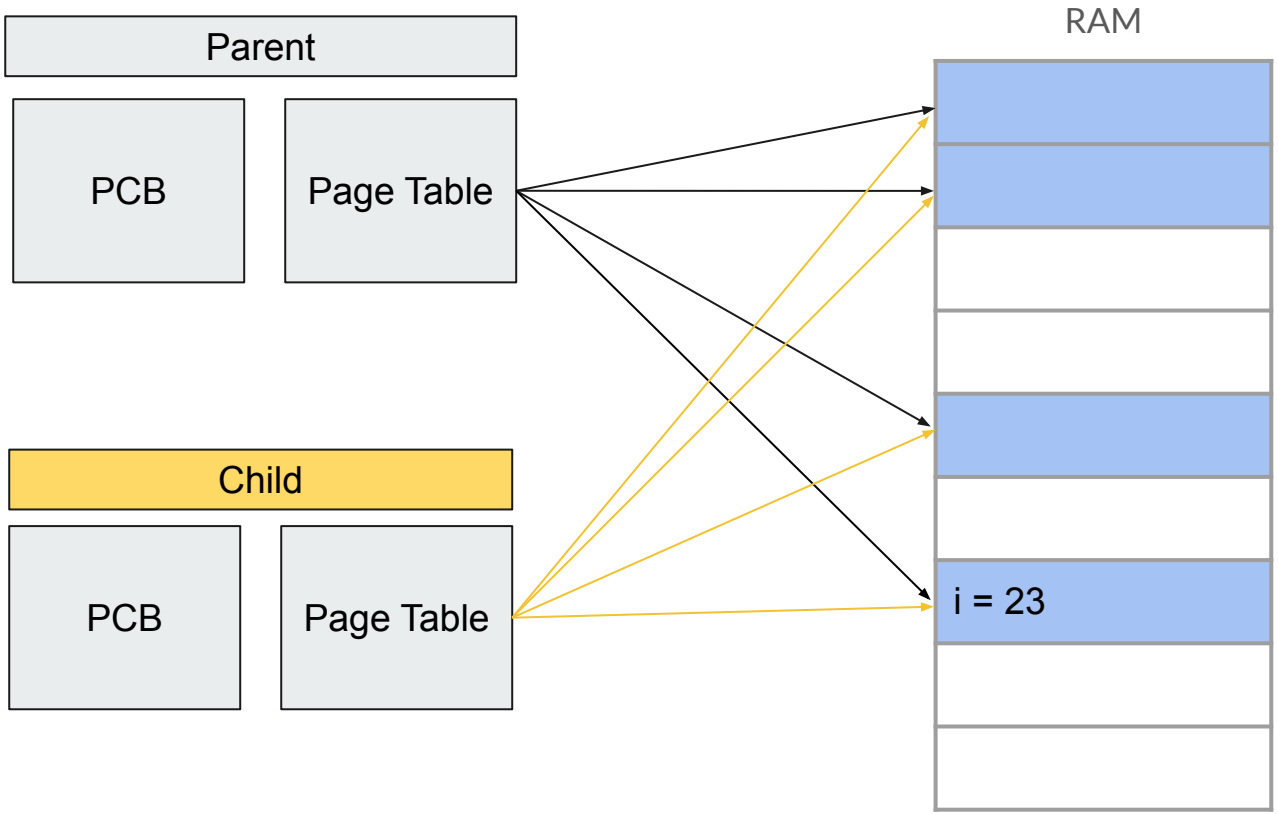
Motivating example 2:

```
int i = 23, pid;

pid = fork()
if (pid > 0){
   sleep(1);
   printf("parent: %d\n", i);
   wait();      //parent process waits for child to finish
}
else{
   i = i + 1;
   printf("child: %d\n", i);
}
```

```
> child: 24
>Parent: 23
```

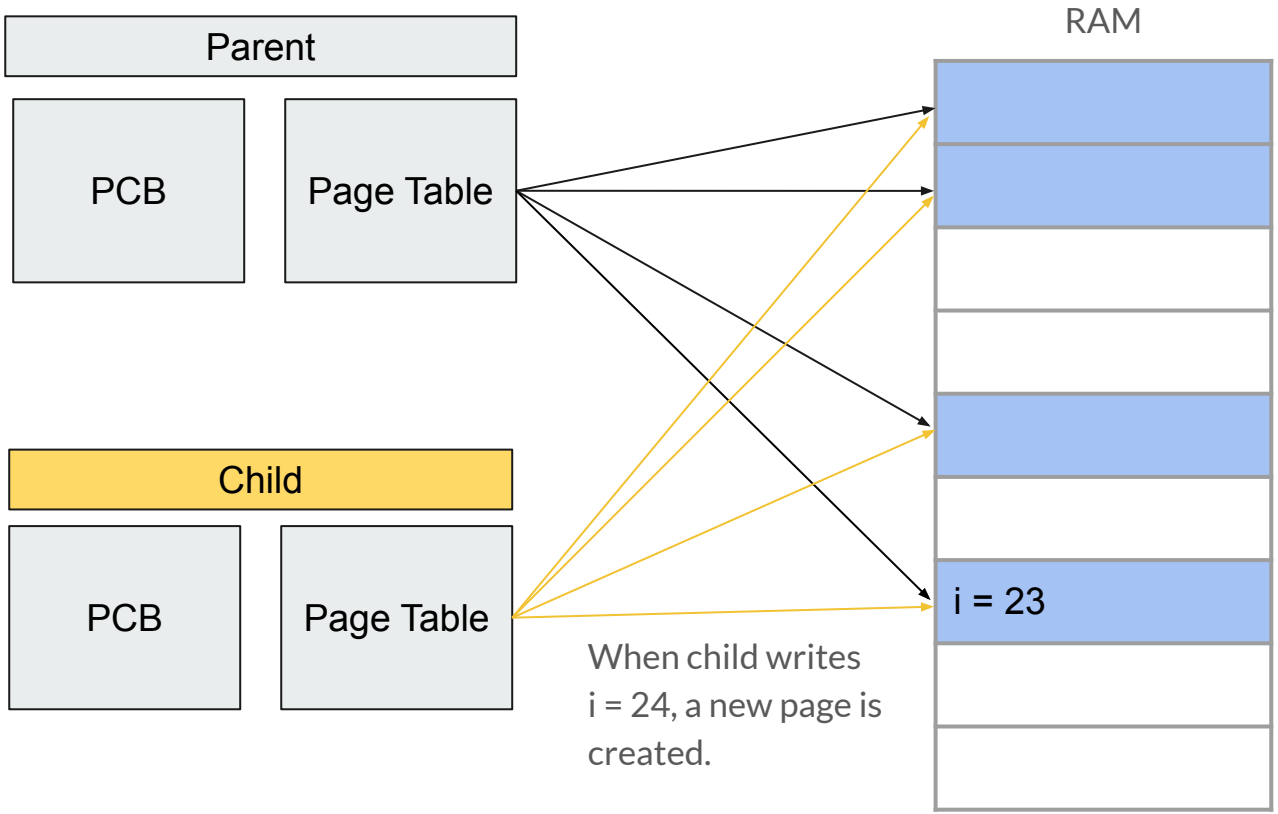
This happens because of Copy on Write (COW)

Copy On Write (COW)



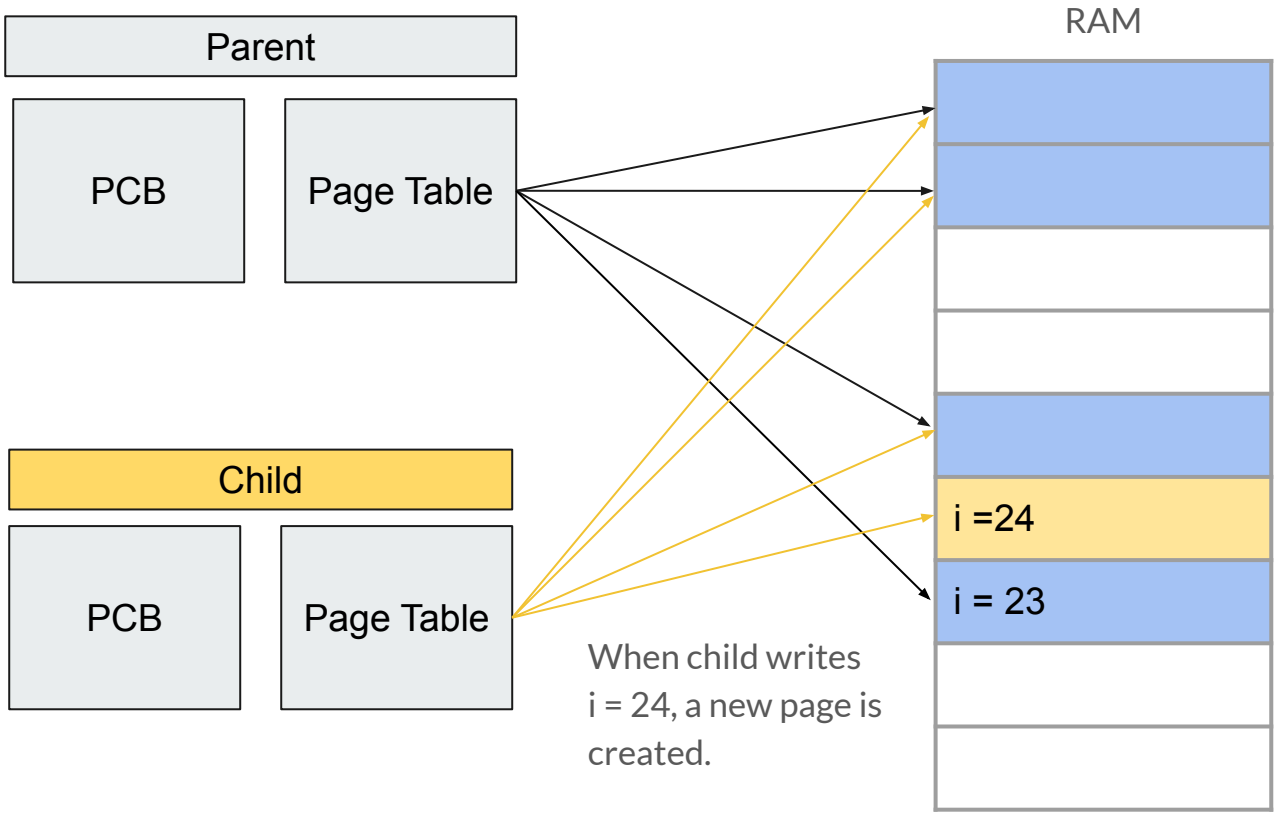
Parent and Child share the same pages in memory

Copy On Write (COW)



Parent and Child share the same pages in memory

Copy On Write (COW)



Parent and Child share the same pages in memory

What about creating a new process?

Creating a process:

```
int main(int argc, char *argv[]){
   int rc = fork();

   if (rc == 0) {
      // child (new process)
      execlp("ls", "", NULL);
      exit(0);
   } else {
      // parent goes down this path (original process)
       wait();
   }
   return 0;
}
```

What about creating a new process?

Creating a process:

- fork() clones the process.

```
int main(int argc, char *argv[]){
  int rc = fork();

  if (rc == 0) {
    // child (new process)
    execlp("ls", "", NULL);
    exit(0);
  } else {
    // parent goes down this path (original process)
     wait();
  }
  return 0;
}
```

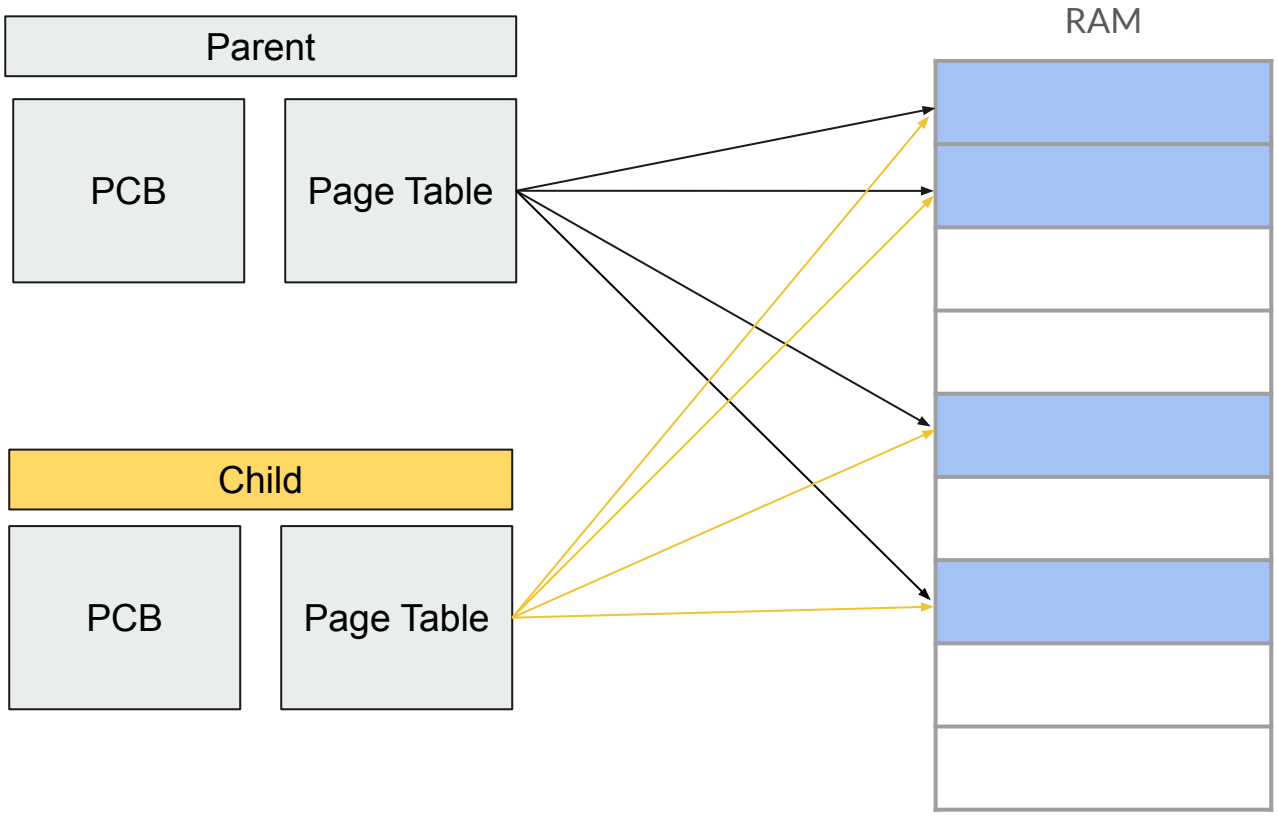
What about creating a new process?

Creating a process:

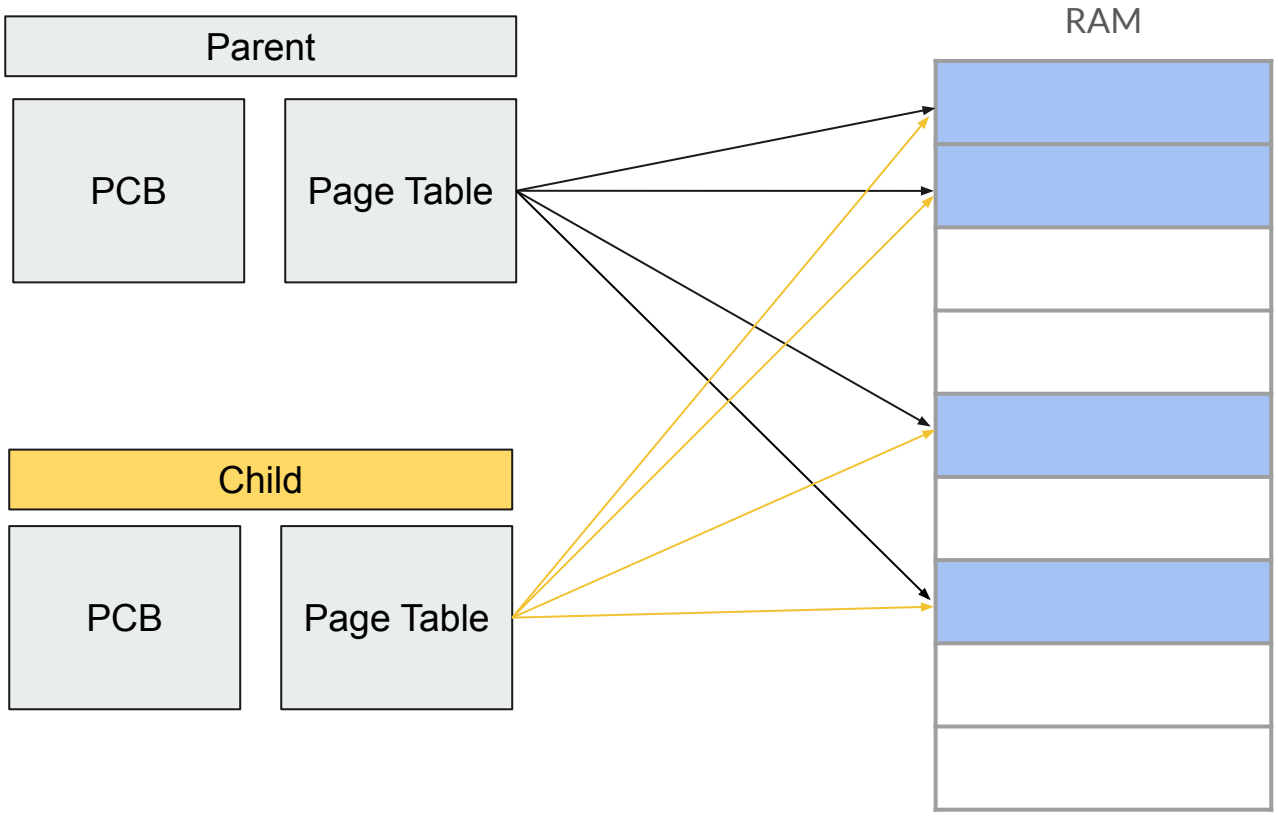
- fork() clones the process.
- exec() loads a new process into the process.

```
int main(int argc, char *argv[]){
  int rc = fork();

  if (rc == 0) {
    // child (new process)
    execlp("ls", "", NULL);
    exit(0);
  } else {
    // parent goes down this path (original process)
     wait();
  }
  return 0;
}
```

exec() from memory perspective:

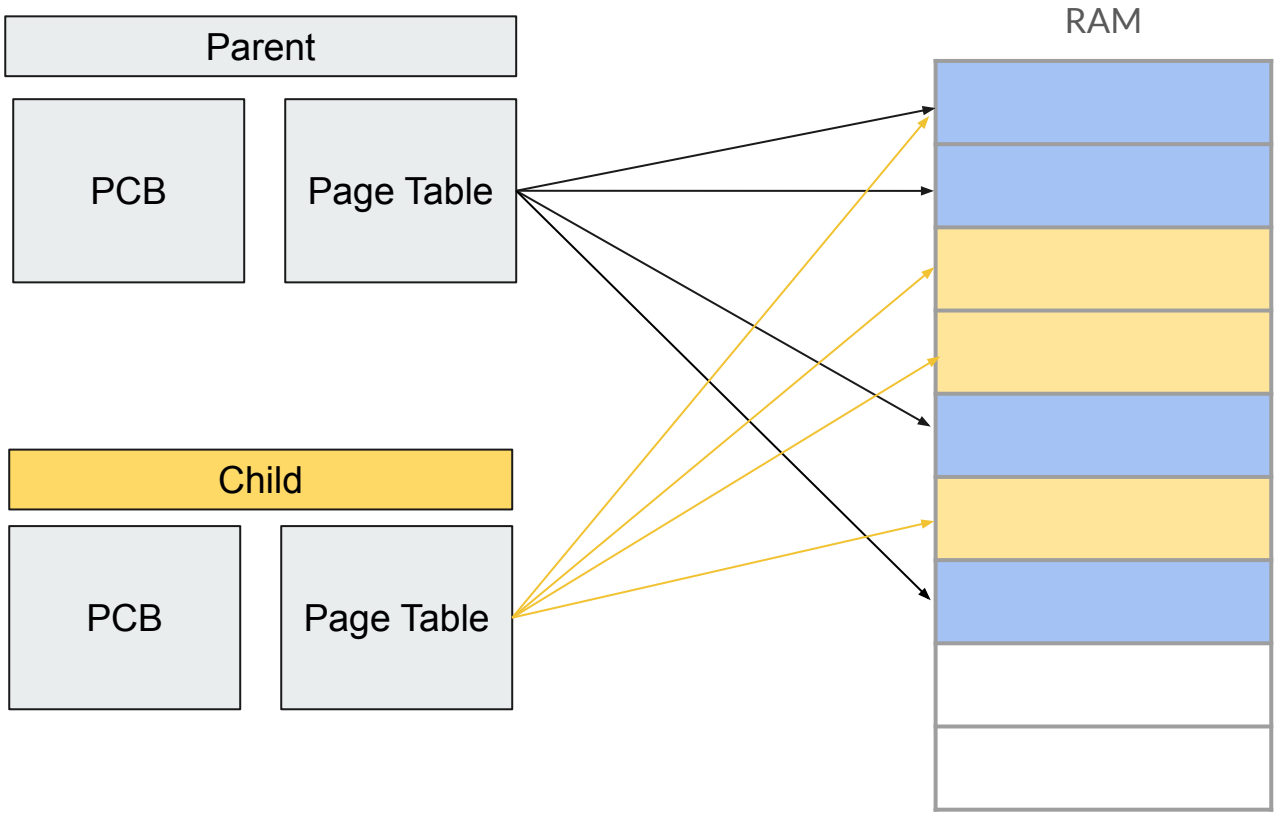


exec() from memory perspective:



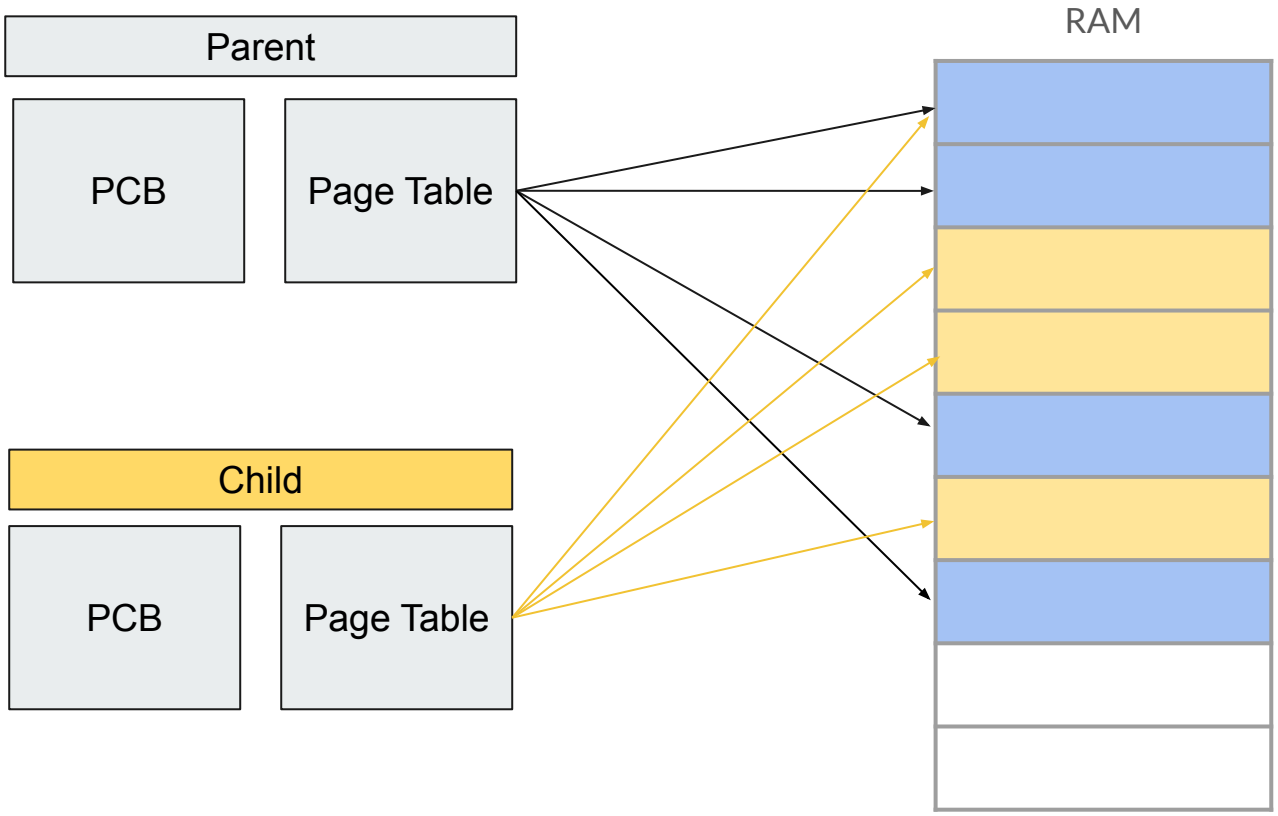
When child calls exec() the pages of the new process are loaded into memory.

exec() from memory perspective:



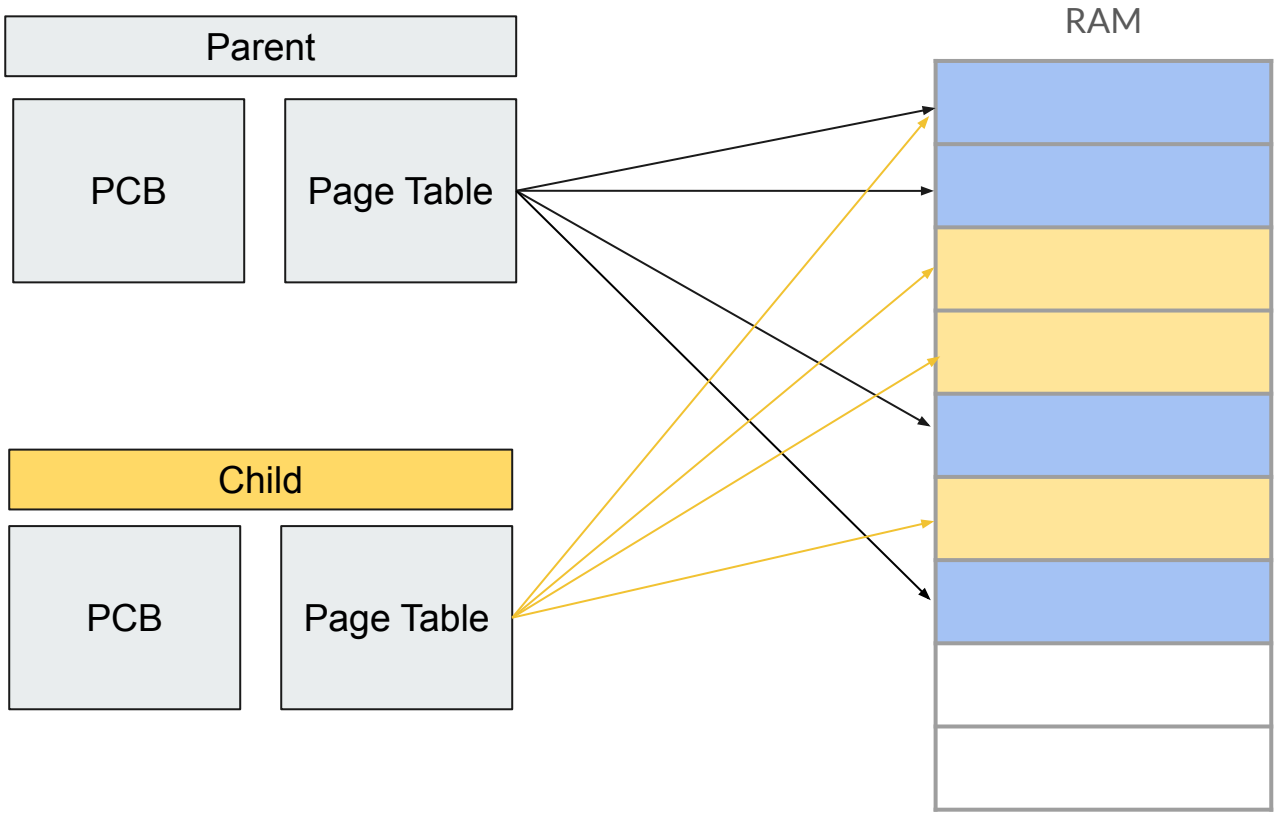
New pages are loaded with instructions and data for the new process.

exec() from memory perspective:



How come some pages are still shared?

exec() from memory perspective:



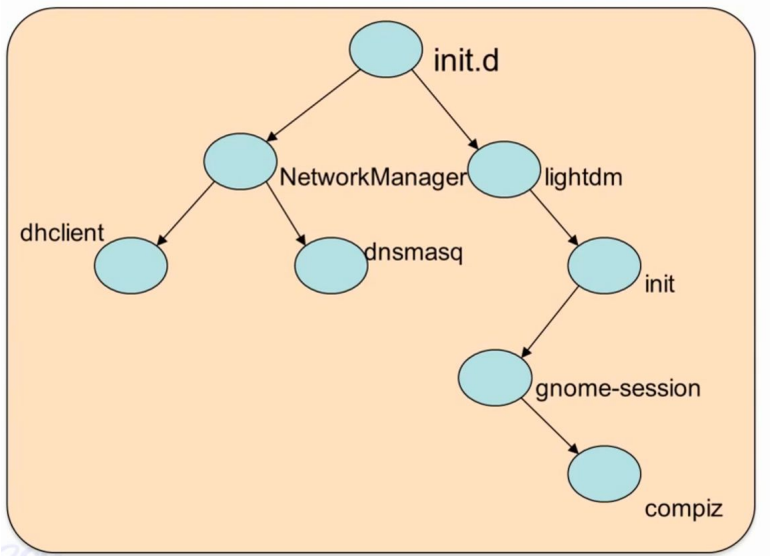
How come some pages are still shared?

Some modules might be in common between the two processes, like cout.

Run code

myshell.c file demonstrating a very simplified shell

pstree



In Xv6: initcode.S is the super-parent

```
[nalmadi@gemini cpu-api]$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager───2*[{NetworkManager}]
        ├─VGAuthService
        ├─2*[abrt-watch-log]
        ├─abrtd
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─at-spi-bus-laun─┬─dbus-daemon
        │                 └─3*[{at-spi-bus-laun}]
        ├─at-spi2-registr───2*[{at-spi2-registr}]
        ├─atd
        ├─atom───atom─┬─atom───atom─┬─2*[apm]
        │             │             ├─atom───{atom}
        │             │             └─14*[{atom}]
        │             ├─atom
        │             └─20*[{atom}]
        ├─atom───atom─┬─atom───atom─┬─2*[apm]
        │             │             ├─atom───{atom}
        │             │             ├─atom───6*[{atom}]
        │             │             └─14*[{atom}]
        │             ├─atom
        │             └─21*[{atom}]
        ├─auditd─┬─audispd─┬─sedispatch
        │        │         └─{audispd}
        │        └─{auditd}
        ├─automount───4*[{automount}]
        ├─avahi-daemon───avahi-daemon
        ├─boltd───2*[{boltd}]
        ├─chronyd
        ├─colord───2*[{colord}]
        ├─crond───2*[crond───sh───sleep]
        ├─cupsd
        ├─4*[dbus-daemon]
        ├─2*[dbus-launch]
        ├─dnsmasq───dnsmasq
        ├─10*[emacs───3*[{emacs}]]
```

wait()

Sometimes, as it turns out, it is quite useful for parent to wait (block) for a child process to finish what it has been doing.

This task is accomplished with the wait() system call.

```
int main(int argc, char *argv[]){
   int rc = fork();

   if (rc == 0) {
      // child (new process)
      execlp("ls", "", NULL);
      exit(0);
   } else {
      // parent goes down this path (original process)
       wait();
   }
   return 0;
}
```

exit()

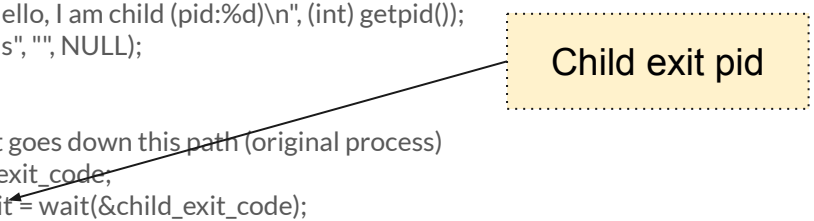
Terminates a process and returns an exit code to parent.

Wakes up parent if parent called wait()

```
int main(int argc, char *argv[]){
  int rc = fork();

  if (rc == 0) {
    // child (new process)
    execlp("ls", "", NULL);
    exit(0);
  } else {
    // parent goes down this path (original process)
     wait();
  }
  return 0;
}
```

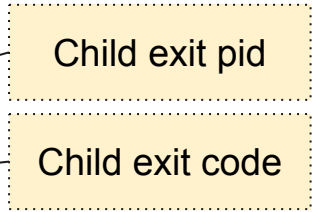
```
int main(int argc, char *argv[])
{
   printf("hello world (pid:%d)\n", (int) getpid());
   int rc = fork();

   if (rc < 0) {
      // fork failed; exit
      fprintf(stderr, "fork failed\n");
      exit(1);
   } else if (rc == 0) {
      // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
      execlp("ls", "", NULL);
      exit(0);
   } else {
      // parent goes down this path (original process)
     int child_exit_code;
     int rc_wait = wait(&child_exit_code);
     printf("hello, I am parent of %d (pid:%d)\nwait pid: %d\nchild exit code:\
%d\n",
          rc, (int) getpid(), rc_wait, child_exit_code);
   }
   return 0;
}
```

```
int main(int argc, char *argv[])
{
   printf("hello world (pid:%d)\n", (int) getpid());
   int rc = fork();

   if (rc < 0) {
      // fork failed; exit
      fprintf(stderr, "fork failed\n");
      exit(1);
   } else if (rc == 0) {
      // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
      execlp("ls", "", NULL);
      exit(0);
   } else {
      // parent goes down this path (original process)
    int child_exit_code;
    int rc_wait = wait(&child_exit_code);
    printf("hello, I am parent of %d (pid:%d)\nwait pid: %d\nchild exit code:\
%d\n",
          rc, (int) getpid(), rc_wait, child_exit_code);
   }
   return 0;
}
```



If there are no children, returns -1

```
int main(int argc, char *argv[])
{
   printf("hello world (pid:%d)\n", (int) getpid());
   int rc = fork();

   if (rc < 0) {
      // fork failed; exit
      fprintf(stderr, "fork failed\n");
      exit(1);
   } else if (rc == 0) {
      // child (new process)
      printf("hello, I am child (pid:%d)\n", (int) getpid());
      execlp("ls", "", NULL);
      exit(0);
   } else {
      // parent goes down this path (original process)
     int child_exit_code;
     int rc_wait = wait(&child_exit_code);
     printf("hello, I am parent of %d (pid:%d)\nwait pid: %d\nchild exit code:\
%d\n",
          rc, (int) getpid(), rc_wait, child_exit_code);
   }
   return 0;
}
```



Run: my_exec.c

Process Termination

- Using exit() is called voluntary termination.
- kill(pid, signal) is non-voluntary termination.

Zombie Process

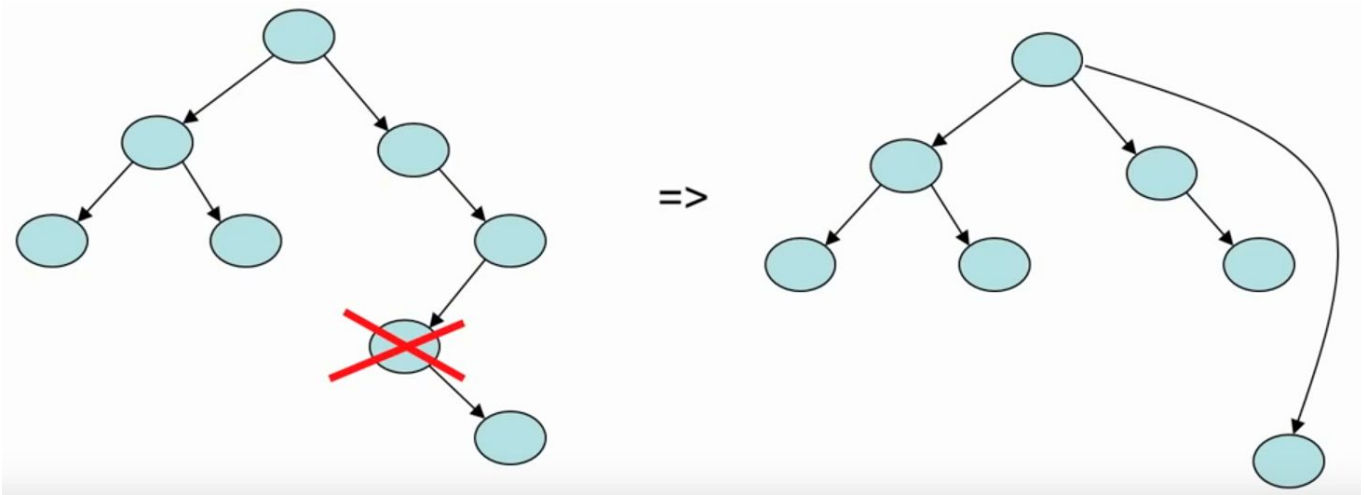
When process terminates it becomes a zombie process:

- Program no longer executes, but it PCB is kept by kernel.
- This is done to allow parent to read the exit code (through wait()) of the zombie process.

When parent reads exit code, the PCB is removed by kernel.
If parent did not read the code, a leak is caused by the PCB remaining in memory.

Orphaned Process

When parent dies before child the child is adopted by init.d



Orphaned Process

Unintentional orphan:

- When parent process crashes.

Intentional orphan:

- Process becomes detached from parent process (daemon).
- Used to run background services.

exit() internals

- Init.d cannot exit.

For all other processes:

- Decrement the usage count for all open files, if count ==0 close file.
- Wake up parent if sleeping and pass exit code.
- Make init.d adopt children of process.
- Set process state to zombie.

Check proc.c in Xv6

Hello World inside Xv6

Do the following:

- Create hello.c inside the Xv6 directory, and write the following code in it:

```
include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"


int
main(void)
{
printf(1, "Hello There!\n");
exit();
}
```

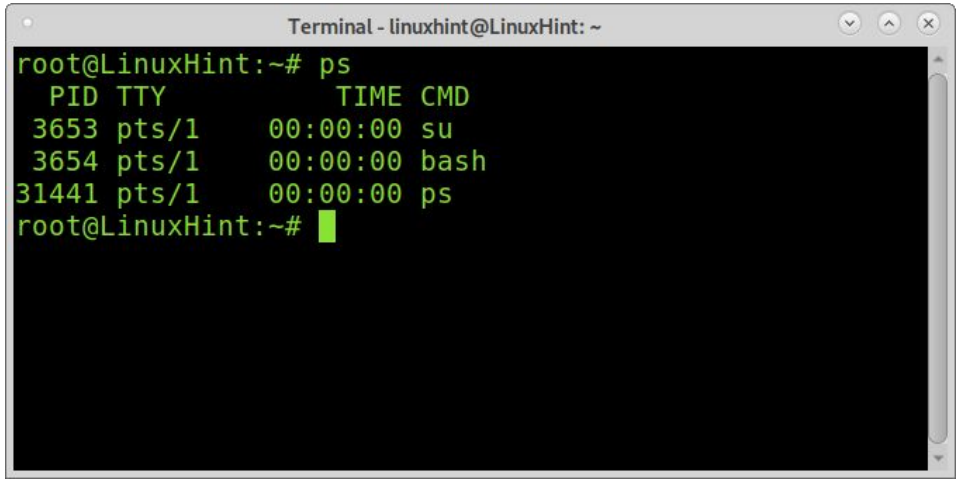
Update Makefile

- Add _hello\ to UPROGS
- Add “hello.c” to EXTRAS
- make
- make qemu-nox
- hello

PS system call

PS

ps command is used to list the currently running processes and their PIDs along with some other information depends on different options.



Do the following:

- Add “#define SYS_ps 22” to syscall.h
- Add “int ps(void);” under proc.c in the file defs.h
- Add “int ps(void);” in the file user.h

Add the following to proc.c

```
int
ps()
{
 struct proc *p;

//enable interrupts on processor
sti();

//loop over process table
acquire(&ptable.lock);

cprintf("name \t pid \t state \t priority\n");

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

 if (p->state == SLEEPING)
  cprintf("%s \t %d \t SLEEPING \t priority \n", p->name, p->pid);
 else if (p->state == RUNNING)
  cprintf("%s \t %d \t RUNNING \t priority \n", p->name, p->pid);
 else if (p->state == RUNNABLE)
  cprintf("%s \t %d \t RUNNABLE \t priority \n", p->name, p->pid);

}
release(&ptable.lock);

return 22;
}
```

then...

- Add the following to sysproc.c:
  - int
  - sys_ps(void)
  - {
  - return ps();
  - }

- Add “SYSCALL(ps)” to usys.s

After that...

- In “sys_call.c” add the lines:
  - extern int sy_sps(void);
  - [SYS_ps] sys_ps,
- Create ps.c and add to it:
  - #include "types.h"
  - #include "stat.h"
  - #include "user.h"
  - #include "fcntl.h"
  -
  - int
  - main(void)
  - {
  - ps();
  - exit();
  - }

Finally ...

- Add “ _ps\” under UPROGS in the Makefile
- Add “ ps.c” under EXTRAS in the Makefile
- make
- make qemu-nox
- ps