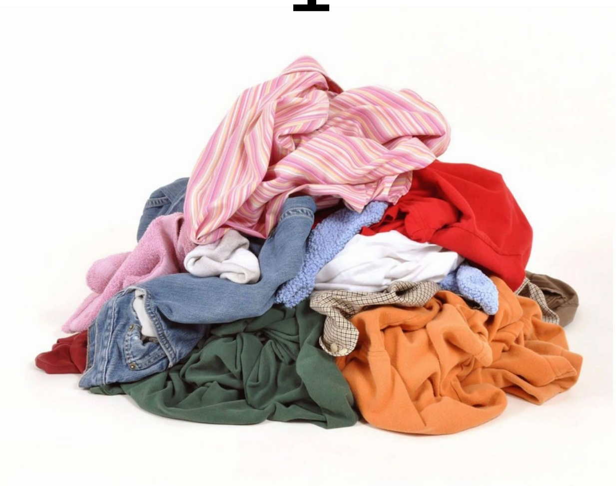
CSE 3100: Systems Programming

Part 2
Lecture 1: Introduction to Processes

The Rock’s Laundry Problem

1



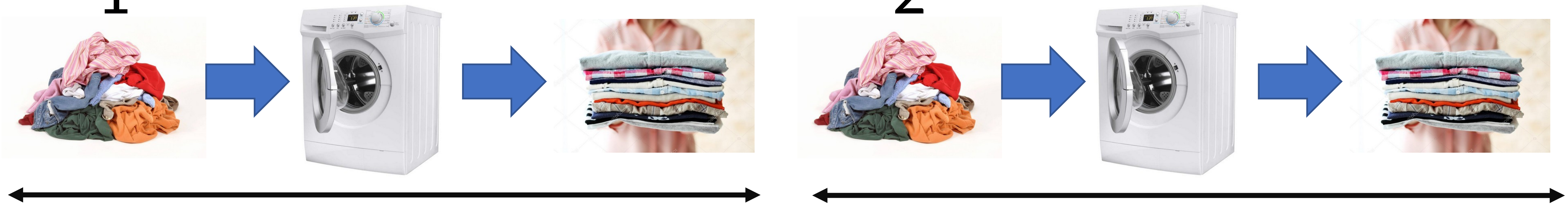
2

- The Rock (pictured above) needs your help doing laundry before the premier of his new movie.
- The Rock has 2 loads of laundry and needs them done in 1 hour.
- Each load of laundry takes 1 hour to do in a washing machine.
- If you finish his laundry on time he will take you to see his new movie.
- If you don’t finish on time he will drop a rock on your foot (that is how he got the name “The Rock”).

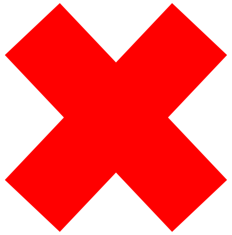
The Rock’s Laundry Problem

- If you use 1 washing machine what will happen?

1 2



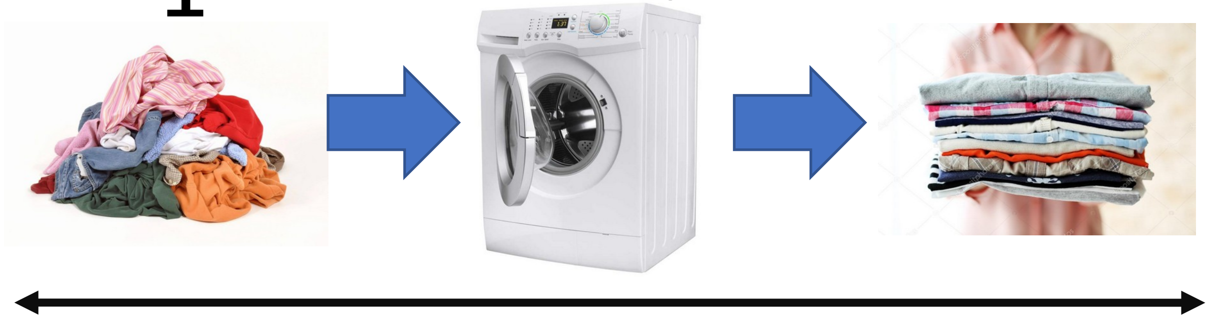
1 Hour 1 Hour

 Total Time = 2 Hours 

If you use 2 washing machine *IN PARALLEL* what will happen?

1

Parent washing machine



1 Hour

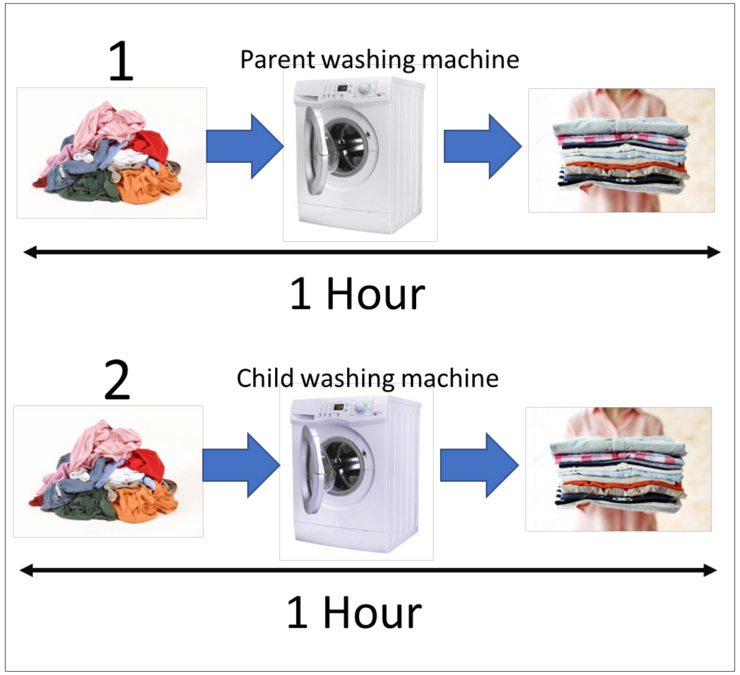
2

Child washing machine

1 Hour

Total Time = 1 Hour

The Rock's Laundry Problem Conclusions



- By doing multiple processes at the *same time* we can speed up computation.
- How would we create multiple processes on a computer to do computations in parallel?
- Answer: The Fork Function.

Fork Function Example

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

Output of the code:

```
kaleel@CentralCompute:~$ gcc test.c -o test
kaleel@CentralCompute:~$ ./test
In main: value =809
In main: value =0
```

What the heck is actually going on?

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

Start in main, the program loads into memory etc.

What the heck is actually going on?

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

We create a variable to identify the process we are currently working in. Think about this as a variable to help us figure out whether we are doing laundry in the child or parent “washing machine”.

What the heck is actually going on?

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

Call the fork function.

Now this creates a NEW child process. Essentially where fork is called, we start a clone of the code, running at the fork line.

Note: we still have the parent process running AT THE SAME TIME.

What the heck is actually going on?

Parent Process

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

Child Process

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

- We now have a child process and parent process BOTH running at the same time.
- Which code will end first? It’s a bit complicated.
- For this example, we will arbitrarily go through the child first but in reality either process may finish first in this case (up to the OS).

What the heck is actually going on?

Parent Process

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

Child Process

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

- In the child process we'll go to the print statement and print the id.
- Note children processes are given ID 0.

What the heck is actually going on?

Parent Process

```c
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value; // process identification used to represent process id
7       value = fork(); //call the fork function to start a separate process
8       printf("In main: value =%d\n", value); //print the id of the process
9   }
```

Child Process

```c
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value; // process identification used to represent process id
7       value = fork(); //call the fork function to start a separate process
8       printf("In main: value =%d\n", value); //print the id of the process
9   }
```

- We reach line 9, the end of the code and we’re done.

```
In main: value =0
```

What the heck is actually going on?

Parent Process

```c
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value; // process identification used to represent process id
7       value = fork(); //call the fork function to start a separate process
8       printf("In main: value =%d\n", value); //print the id of the process
9   }
```

Child Process

```c
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value; // process identification used to represent process id
7       value = fork(); //call the fork function to start a separate process
8       printf("In main: value =%d\n", value); //print the id of the process
9   }
```

- Now let’s look back at the parent and finish the code.
- For the parent we were on line 7 in fork.
- Because we forked pid_t is given a unique value in this version of the code.
- Time to go to line 8.

What the heck is actually going on?

Parent Process

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

- We reach the print statement.
- Remember in this version of the code we assigned a value to pid_t using fork.

Child Process

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

What the heck is actually going on?

Parent Process

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

```
In main: value =809
```

Child Process

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    printf("In main: value =%d\n", value); //print the id of the process
}
```

Questions You Should Be Asking Yourself Right Now

Why is the process id different in cloned versions of the code?

Answer: Because we want to write ONE code but have different parts of the code to do different things in parallel.

The process ID allows us to do that.

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Parent Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Parent Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Parent Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Child Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Parent Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Here this is the parent process so it will have a non-zero ID value.

Child Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Here this is the child process so it will have a zero ID value.

Parent Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Here this is the parent process so it will have a non-zero ID value.

```
Let's do the first load of laundry.
```

Child Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Here this is the child process so it will have a zero ID value.

Parent Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Here this is the parent process so it will have a non-zero ID value.

```
Let's do the first load of laundry.
```

Child Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Here this is the child process so it will have a zero ID value.

```
Let's do the second load of laundry.
```

Parent Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Here this is the parent process so it will have a non-zero ID value.

```
Let's do the first load of laundry.
```

Child Process:

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

int main()
{
    pid_t value; // process identification used to represent process id
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        printf("Let's do the first load of laundry.");
    }
    else if(value == 0) //this means we are the child process
    {
        printf("Let's do the second load of laundry.");
    }
}
```

Here this is the child process so it will have a zero ID value.

```
Let's do the second load of laundry.
```

Questions You Should Be Asking Yourself Right Now 2

Do the parent and child (clone) have independent memory?

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

int main()
{
    pid_t value; // process identification used to represent process id
    int x = 100; //some variable
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        x = 5;
        printf("The value of x in the parent process is =%d\n", x);

    }
    else if(value == 0) //this means we are the child process
    {
        sleep(2); //add some intentional delay so this process finishes slower
        printf("The value of x in the child process is =%d\n", x);
    }
}
```

If the parent and child have independent memory what should be printed?

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

int main()
{
    pid_t value; // process identification used to represent process id
    int x = 100; //some variable
    value = fork(); //call the fork function to start a separate process
    if(value != 0)//this means we are in the parent process
    {
        x = 5;
        printf("The value of x in the parent process is =%d\n", x);

    }
    else if(value == 0) //this means we are the child process
    {
        sleep(2); //add some intentional delay so this process finishes slower
        printf("The value of x in the child process is =%d\n", x);
    }
}
```

```
kaleel@CentralCompute:~$ ./test
The value of x in the parent process is =5
kaleel@CentralCompute:~$ The value of x in the child process is =100
```

Variables are new copies in a new process!

By now you should be a process expert…so time for some basic definitions.

Process Basics

- A **process** is an instance of a program being executed
    - Core operating system (OS) concept
- In a **multiprocessing** OS
    - Multiple programs can be executed at the same time
    - Multiple instances of a program can be executed at the same time
- Executing multiple programs
    - Single-core: time-sharing
    - Multi-core: true parallelism + time-sharing

Process Management: OS View

- OS maintains a process table
  - Each process has a table entry, called process control block (PCB)
  - Typical PCB info

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment info Pointer to data segment info Pointer to stack segment info	Root directory Working directory File descriptors User ID Group ID

- OS **scheduler** picks processes to be executed at any given time
  - When a process is suspended, its state is saved in PCB

Process Management: User's View

- Events which cause process creation
  - System initialization
  - User request to create a new process (e.g., **shell command**)
  - Executing a **shell script**, which may create many processes
- Events which cause process termination
  - Normal program exit
  - Error exit
  - Fatal error, e.g., segmentation fault
  - Killed by user command or signal (Ctrl-C)

Useful Unix Commands Related to Processes

- ps
  - List running processes
- pstree
  - Display the **tree** of processes
- top
  - Dynamic view of memory & CPU usage + processes that use most resources (to exit top, press q)
- kill
  - Kill a process given its **process ID**
  - Try -9 option if simple kill does not work

Process Management: Programmer's View

- Process birth
  - Processes are created by other processes!
  - A process always starts as a **clone** of its parent process
  - Then the process may **upgrade itself** to run a different executable
    - Child process **retains access** to the files open in the parent

- Process life
  - Child process can create its own children processes
- Process death
  - Eventually calls **exit** or **abort** to commit "suicide"
  - Or gets killed

Birth via Cloning

Pictured: Clone Troopers from Star Trek created using the Fork function.

- The function to create a new process in your code

```
#include <unistd.h>
pid_t fork(void);
```

- Child is an exact copy of the parent
  - Both return from fork()
- **Only difference is the returned value**
  - In the **parent** process:
    - fork() returns the process identifier of the child (> 0)
    - If a failure occurred, it returns -1 (and sets errno)
  - In the **child** process: fork() returns 0 (zero)

Question Time: When launching a parent process and a child process which one will always return first?

Possible Answers:

1. The child process always returns first.
2. The parent process always returns first.
3. It is impossible to tell with the information given.
4. I am sleeping in class.

Concurrency

- Parent and child processes return from fork() concurrently
  - They may return at the same time (on a multicore machine) or one after the other
  - Cannot assume that they return at the same time or which one “returns first” (even on a uni-core)
    - Order is chosen by OS scheduler

Cloning effects

- On memory
  - The parent and child memory 100% identical
  - But are viewed as distinct by OS (“copy-on-write”)
  - Any memory change (stack/heap) affects only that copy
  - Thus the parent and child can quickly diverge
- On files
  - All files open in the parent are accessible in the child!
  - I/O operations in either one move the file position indicator

In particular

- stdin, stdout, and stderr of the parent are accessible in the child

What can the parent do while the child process is running?

- Depends on application!
  - It could wait until the child is done (dies!)
    - Typical of a shell like bash/ksh/zsh/csh/….
  - It could run concurrently and check back on the child later
  - It could run concurrently and ignore the child
    - If child dies it enters a **zombie** state

Waiting on a child

```
#include <sys/wait.h>

pid_t   wait(int * status);
pid_t   waitpid(pid_t pid, int * status, int options);
```

- Purpose
  - Block the calling process until a child is terminated
    - Or other state changes specified by options
  - Report status in *status (which is ignored if NULL is passed)
    - The cause of death
    - The exit status of the child (what he returned from main)
  - Return value identifies the child process (or -1 on error)
- Run "**man -S2 wait**" for full details

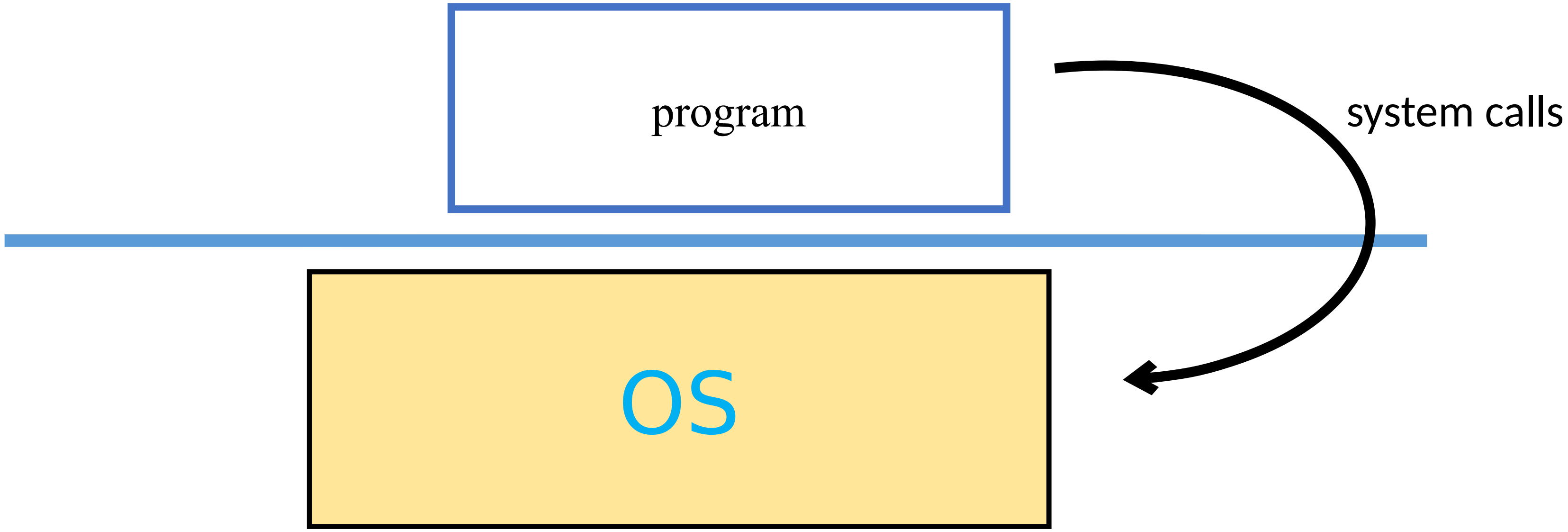
Zombies!

- A dead process, waiting to be 'reaped' (checked by its parent)
  - You cannot kill it, because it is already dead
  - Most resources released, but still uses an entry in the process table
- Parents should check their kids
  - On some systems, parents can say they do not want to check
- When a parent dies, ‘init’ becomes the new parent
  - Then the zombie child is reaped

System calls

- APIs used to request services from the OS kernel
  - Example: fork()
  - System calls are more expensive than normal function calls
  - Manuals for system calls are in section 2

```
man -S2 intro ;  man -S2 syscalls
```



One more Forking Example

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

int main()
{
    pid_t value;
    value = fork();
    value = fork();
    printf("In main: value =%d\n", value);
    return 0;
}
```

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

int main()
{
    pid_t value;
    value = fork();
    value = fork();
    printf("In main: value =%d\n", value);
    return 0;
}
```

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

int main()
{
    pid_t value;
    value = fork();
    value = fork();
    printf("In main: value =%d\n", value);
    return 0;
}
```

Parent Process:
pid_t = 371

```c
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value;
7       value = fork();
8       value = fork();
9       printf("In main: value =%d\n", value);
10      return 0;
11  }
```

- The parent process clones a child which will start running on line 8.

Child Process:
pid_t = 0

```c
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value;
7       value = fork();
8       value = fork();
9       printf("In main: value =%d\n", value);
10      return 0;
11  }
```

Parent Process:
pid_t = 371

```c
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value;
7       value = fork();
8       value = fork();
9       printf("In main: value =%d\n", value);
10      return 0;
11  }
```

- Line 8 in the child is another call to fork though!

Child Process:
pid_t = 372

```c
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value;
7       value = fork();
8       value = fork();
9       printf("In main: value =%d\n", value);
10      return 0;
11  }
```

Another Child Process:
pid_t = 0

```c
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value;
7       value = fork();
8       value = fork();
9       printf("In main: value =%d\n", value);
10      return 0;
11  }
```

Parent Process:
pid_t = 371

```c
1    #include <stdio.h>
2    #include <unistd.h>
3
4    int main()
5    {
6        pid_t value;
7        value = fork();
8        value = fork();
9        printf("In main: value =%d\n", value);
10       return 0;
11   }
```

- Go back to the parent process. We just finished line 7. Time to call line 8.

Child Process:
pid_t = 372

```c
1    #include <stdio.h>
2    #include <unistd.h>
3
4    int main()
5    {
6        pid_t value;
7        value = fork();
8        value = fork();
9        printf("In main: value =%d\n", value);
10       return 0;
11   }
```

Another Child Process:
pid_t = 0

```c
1    #include <stdio.h>
2    #include <unistd.h>
3
4    int main()
5    {
6        pid_t value;
7        value = fork();
8        value = fork();
9        printf("In main: value =%d\n", value);
10       return 0;
11   }
```

Parent Process:
pid_t = 371

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value;
7       value = fork();
8       value = fork();
9       printf("In main: value =%d\n", value);
10      return 0;
11  }
```

- Uh oh line 8 is ANOTHER call to fork.

Child Process:
pid_t = 372

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value;
7       value = fork();
8       value = fork();
9       printf("In main: value =%d\n", value);
10      return 0;
11  }
```

Another Child Process:
pid_t = 0

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value;
7       value = fork();
8       value = fork();
9       printf("In main: value =%d\n", value);
10      return 0;
11  }
```

Another other Child Process:
pid_t = 0

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main()
5   {
6       pid_t value;
7       value = fork();
8       value = fork();
9       printf("In main: value =%d\n", value);
10      return 0;
11  }
```

Parent Process:
pid_t = 371

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

int main()
{
    pid_t value;
    value = fork();
    value = fork();
    printf("In main: value =%d\n", value);
    return 0;
}
```

Done forking, print the pid_t value!

- Now advance one line in each part of the process.

Child Process:
pid_t = 372

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

int main()
{
    pid_t value;
    value = fork();
    value = fork();
    printf("In main: value =%d\n", value);
    return 0;
}
```

Done forking, print the pid_t value!

Another Child Process:
pid_t = 0

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

int main()
{
    pid_t value;
    value = fork();
    value = fork();
    printf("In main: value =%d\n", value);
    return 0;
}
```

Done forking, print the pid_t value!

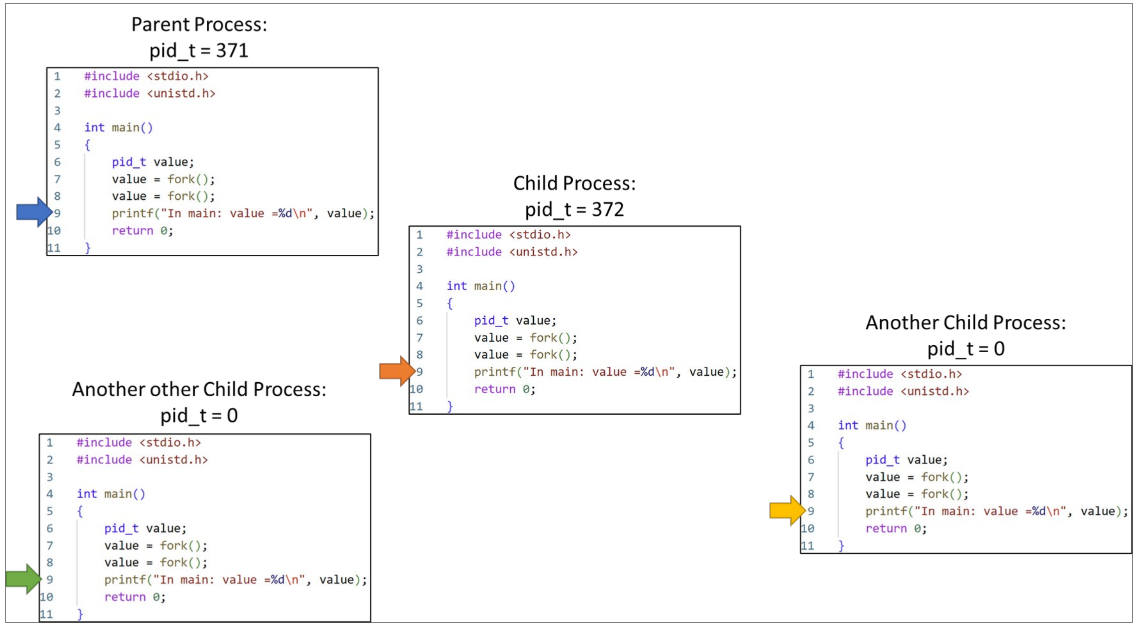
Another other Child Process:
pid_t = 0

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

int main()
{
    pid_t value;
    value = fork();
    value = fork();
    printf("In main: value =%d\n", value);
    return 0;
}
```

Done forking, print the pid_t value!

Quick Question

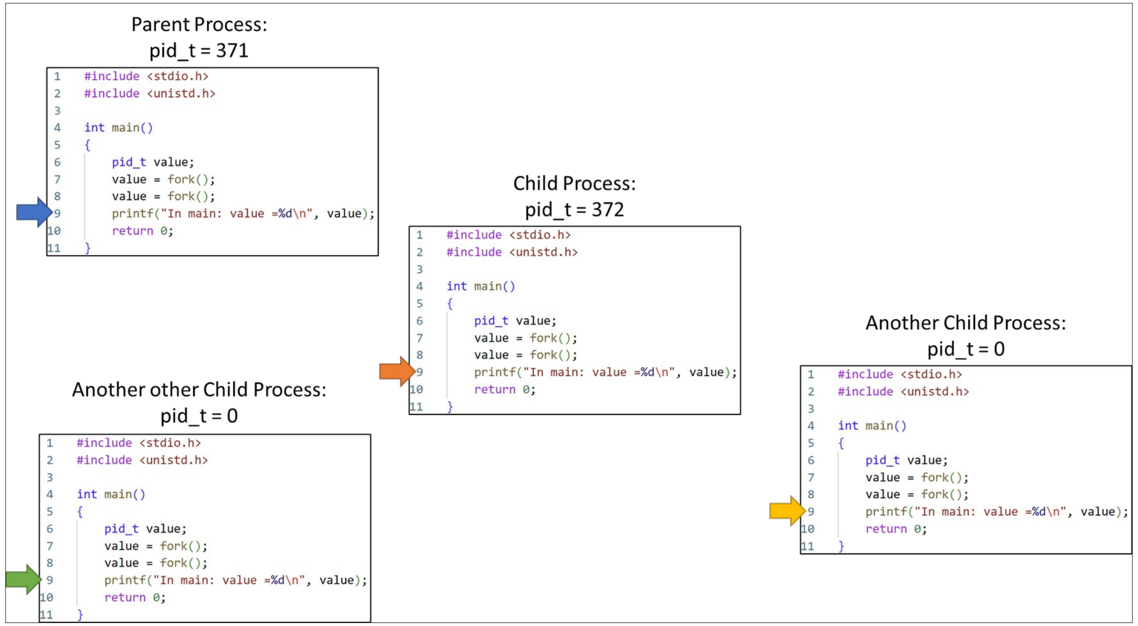


- We know that the pit_d value will be printed in each of the processes. We also know the printed values will be 371, 0, 372 and 0.
- Can we correctly predict the ORDER in which the values will be printed before this code runs?

Can we correctly predict the ORDER in which the values will be printed before this code runs?

(It is up to the OS to schedule things)

The multi-fork code output:




```
kaleel@CentralCompute:~$ ./test
In main: value =371
In main: value =372
In main: value =0
In main: value =0
```

Lecture Conclusions

- To run things in parallel we need to run multiple processes.
- We use the fork function to create processes, i.e., parents and children.
- Cloning processes can be very tricky when it comes to tracking completion time and variables.

Figure Sources

1. https://preview.redd.it/ed9omvbvlim91.png?auto=webp&s=2d4455db0597197b1f5fbfcf02a9046a46110f2b
2. https://www.threegirlsmedia.com/wp-content/uploads/2016/06/LaundryTimeManagement.6.08.2016.jpg
3. https://www.collinsdictionary.com/images/full/washingmachine_71039011_1000.jpg
4. https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fbucketeer-e05bbc84-baa3-437e-9518-adb32be77984.s3.amazonaws.com%2Fpublic%2Fimages%2F3bdb2575-9a92-42f8-8472-bb78c7bd118a_720x405.jpeg
5. https://i.kym-cdn.com/entries/icons/facebook/000/018/124/therock.jpg
6. https://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/Checkmark.svg/2304px-Checkmark.svg.png
7. https://i.ytimg.com/vi/REEBnVkIXQU/maxresdefault.jpg
8. https://media.istockphoto.com/id/1212960962/photo/young-handsome-man-with-beard-wearing-casual-sweater-and-glasses-over-blue-background.jpg?s=612x612&w=0&k=20&c=OROMM-bo6YlzmnsAfQZDyFfYAskJUHcDKE0XDNfKUwM=
9. https://preview.redd.it/xo7f8sugdcn61.jpg?width=640&crop=smart&auto=webp&s=732001dce6fdea3c972f675da3e0f85c81491cf5