

Process Management

# Process Abstraction in Unix

---

Lecture 2b – Unix Case study

# Overview

- Process in Unix
  - Identification
  - Information
  - Creation
  - Termination
  - Parent-Child Synchronization
- Process states in Unix
- Implementation Issues

# Process Abstraction in Unix

## Identification

- PID: Process ID (an integer value)

## Information

- Process State:
  - Running, Sleeping, Stopped, **Zombie**
- Parent PID:
  - PID of the parent process
- Cumulative CPU time:
  - Total amount of CPU time used so far
- etc

## ■ Unix Command for process information:

- **ps** (short for process status)

# Process Creation in Unix: **fork()**

- The main way to create a new process

Header File	<code>#include &lt;unistd.h&gt;</code>
Syntax	<code>int <i>fork</i>( );</code>

- Returns:
  - PID of the newly created process (for parent process) OR
  - 0 (for child process)
- Header files are system dependent
  - "*man fork*" to locate the right files for your system!

# Process Creation in Unix: **fork ( )** (cont)

## ■ Behavior:

- ❑ Creates a new process (known as ***child process***)
- ❑ Child process is a **duplicate** of the current executable image
  - i.e. same code, same address space etc
  - Data in child is a **COPY** of the parent (ie.not shared)
- ❑ Child **differs only in:**
  - Process id (PID)
  - Parent (PPID )
    - ❑ Parent = The process which executed the fork()
  - **fork ( )** return value

## fork () : Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    printf("I am ONE\n");
    fork();
    printf("I am seeing DOUBLE\n");

    return 0;
}
```

### ■ Question:

- What do you think is the output?

## `fork ( )`: Example Explained

- **Both** parent and child processes continue executing after `fork ( )`
- A common usage is to use the parent/child process differently
  - For example:
    - The parent spawn off a child to carry out some work
    - And then the parent is ready to take another order
  - **Use the return value of `fork ( )` to distinguish parent and child**

# fork ( ) : Parent and Child Example

```
... ..  
int result;  
  
result = fork();  
{ if (result != 0) {  
    printf("P:My Id is %i\n", getpid());  
    printf("P:Child Id is %i\n", result);  
} else {  
    printf("C:My Id is %i\n", getpid() );  
    printf("C:Parent Id is %i\n", getppid() );  
}  
... ..
```

Parent Process

Child Process



# `fork()` : Independent Memory Space

```
... ..  
int var = 1234;  
int result;  
  
result = fork();  
if (result != 0) {  
    printf("Parent: Var is %i\n", var);  
    var++;  
    printf("Parent: Var is %i\n", var);  
} else {  
    printf("Child: Var is %i\n", var);  
    var--;  
    printf("Child: Var is %i\n", var);  
}  
... ..
```

- Question:
  - Is there ONE or TWO `var` variable?

# Executing A New Program/Image

- **fork ( )** itself is not useful:
  - ❑ You still need to provide the full code for the child process
  - ❑ What if we want to execute ***another existing program*** instead?
- Make use of the **exec ( )** system calls family
  - ❑ Many variants:
    - **execv, execl, execl, execlv, execlp**, etc
  - ❑ Will touch on:
    - **execl**
  - ❑ Others are similar ("man XXX" to find out more)

# Sidetrack: Command Line Argument in C

- You can pass arguments to a program in C

- e.g. **a.exe 1 2 3 hello**

```
int main( int argc, char* argv[] )  
{  
    //use argc and argv  
}
```

- **argc:**

- Number of command line arguments
  - Including the program name itself

- **argv:**

- A char strings array
  - Each element in **argv[]** is a C character string

# C Command Line Argument: Example

```
int main( int argc, char* argv[] )
{
    int i;

    for (i = 0; i < argc; i++){
        printf("Arg %i: %s\n",i, argv[i] );
    }
    return 0;
}
```

## ■ Example Run:

a.out 123 hello world

## ■ Output:

Arg 0: a.out

Arg 1: 123

Arg 2: hello

Arg 3: world

# exec1 () System Call

- To **replace** current executing process image with a new one
  - ❑ Code replacement
  - ❑ PID and other information still intact

Header File	<code>#include &lt;unistd.h&gt;</code>
Syntax	<pre>int exec1( const char *path,            const char *arg0,            ...,            const char *argN, NULL );</pre>

- ❑ **path**: Location of the executable
- ❑ **arg0, ..., argN**: Command Line Argument(s)
- ❑ **NULL**: To indicate end of argument list

# `exec1 ()` : Simple Example

```
int main()
{
    exec1( "/bin/ls", "ls", "-l", NULL);
}
```

## ■ Note:

- ❑ `Path = "bin/ls"`

- The "`dir`" command in unix, to list the files in directory

- ❑ `arg0 = "ls"`

- The program name

- ❑ `arg1 = "-l"`

- The above is exactly the same as executing:

`ls -l`

## Hmm... `fork()` + `exec()` ?

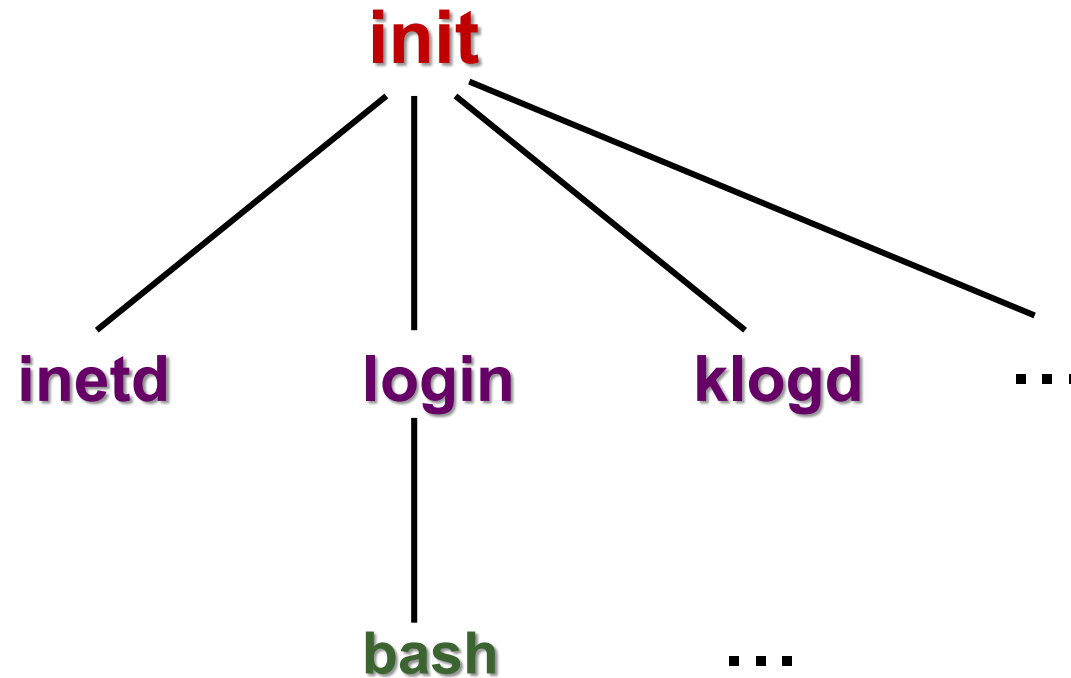
- By combining the two mechanisms, we can:
  - Spawn off a child process
    - Let the child process perform a task through `exec()`
  - Meanwhile, the parent process is still around
    - To accept another request
- Question:
  - Have you used something similar before?
- This combination of mechanisms is the main way in Unix:
  - To get a new process for running a new program

# The Master Process

- Question:
  - ❑ If every process has parent, then which process is the "commonest ancestor"?
- Special initial process:
  - ❑ `init` process
  - ❑ Created in kernel at boot up time
  - ❑ Traditionally has a PID = 1
  - ❑ Watches for other processes and respawns where needed
- `fork()` creates process tree:
  - ❑ `init` is the root process



# Process Tree Example (simplified)



Note: just a simple example, actual process tree varies according to Unix setup

# Process Termination in Unix

## ■ To end execution of process:

Header File	<code>#include &lt;stdlib.h&gt;</code>
Syntax	<code>void exit( int status );</code>

- ❑ Status is returned to the parent process (more later)
- ❑ Unix Convention:
  - 0 = Normal Termination (successful execution)
  - !0 = To indicate problematic execution
- ❑ The function **does not return!**

# Process On Exit

## ■ Process finished execution

- **Most** system resources used by process are released on exit

- E.g. File descriptors

- Each opened file in C has a file descriptor attach to it

- Similar to File object in Java, File Stream Object in C++

- Some basic process resources **not releasable**:

- PID & status needed

- For parent-children synchronization

- Process accounting info, e.g. cpu time

➔ Process table entry **may be** still needed

# Implicit `exit()`

- Most programs have no explicit `exit()` call
- Example:

```
int main()  
{  
    printf("Just to say goodbye!\n");  
}
```

- Return from `main()` implicitly calls `exit()`
  - Open files also get flushed automatically!

# Parent/Child Synchronization in Unix

- Parent process can wait for child process to terminates

Header  
File

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

Syntax

```
int wait( int *status );
```

- ❑ Returns the PID of the terminated child process
- ❑ status (passed by address):
  - Stores the exit status of the terminated child process
  - Use **NULL** if you do not need/want this info

# Parent/Child Synchronization in Unix

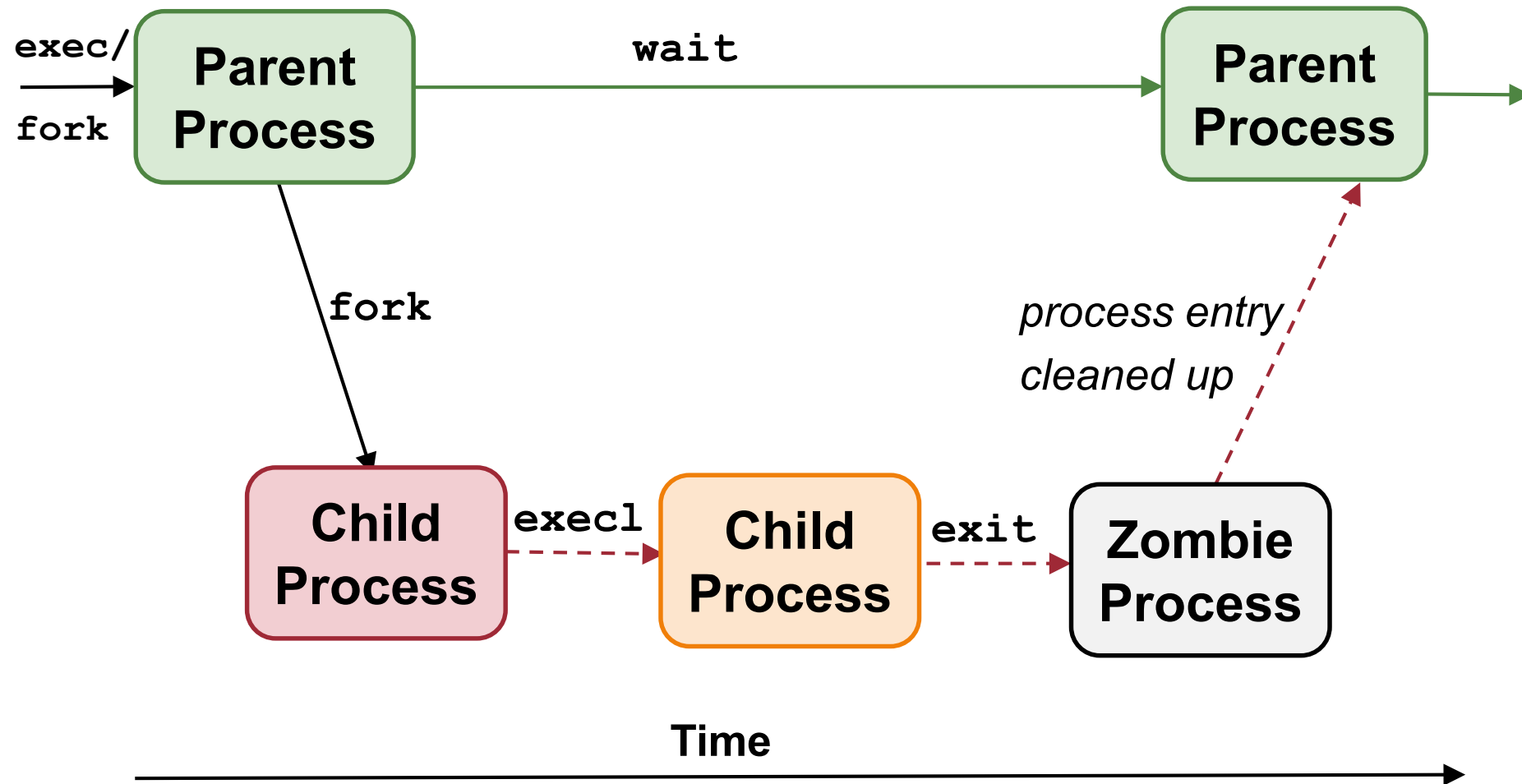
## ■ Behavior:

- ❑ The call is blocking:
  - Parent process blocks until at least one child terminates
- ❑ The call cleans up ***remainder*** of child system resources
  - Those not removed on `exit()`
  - Kill zombie process 😊

## ■ Other variants of `wait()` :

- ❑ `waitpid()`
  - Wait for a specific child process
- ❑ `waitid()`
  - Wait for any child process to **change status**
- ❑ etc...

# Process Interaction in Unix



Note: example uses one ordering of execution, others are possible!

# `wait()` "creates" zombies!!

- On process exit: (see previous slide)
  - becomes **zombie**
- **Cannot delete** all process info
  - What if parent ask for the info in a `wait()` call?
  - Remainder of process data structure can be **cleaned up**
    - only when `wait()` happens
- **Cannot kill** zombie
  - The process is already **dead**!



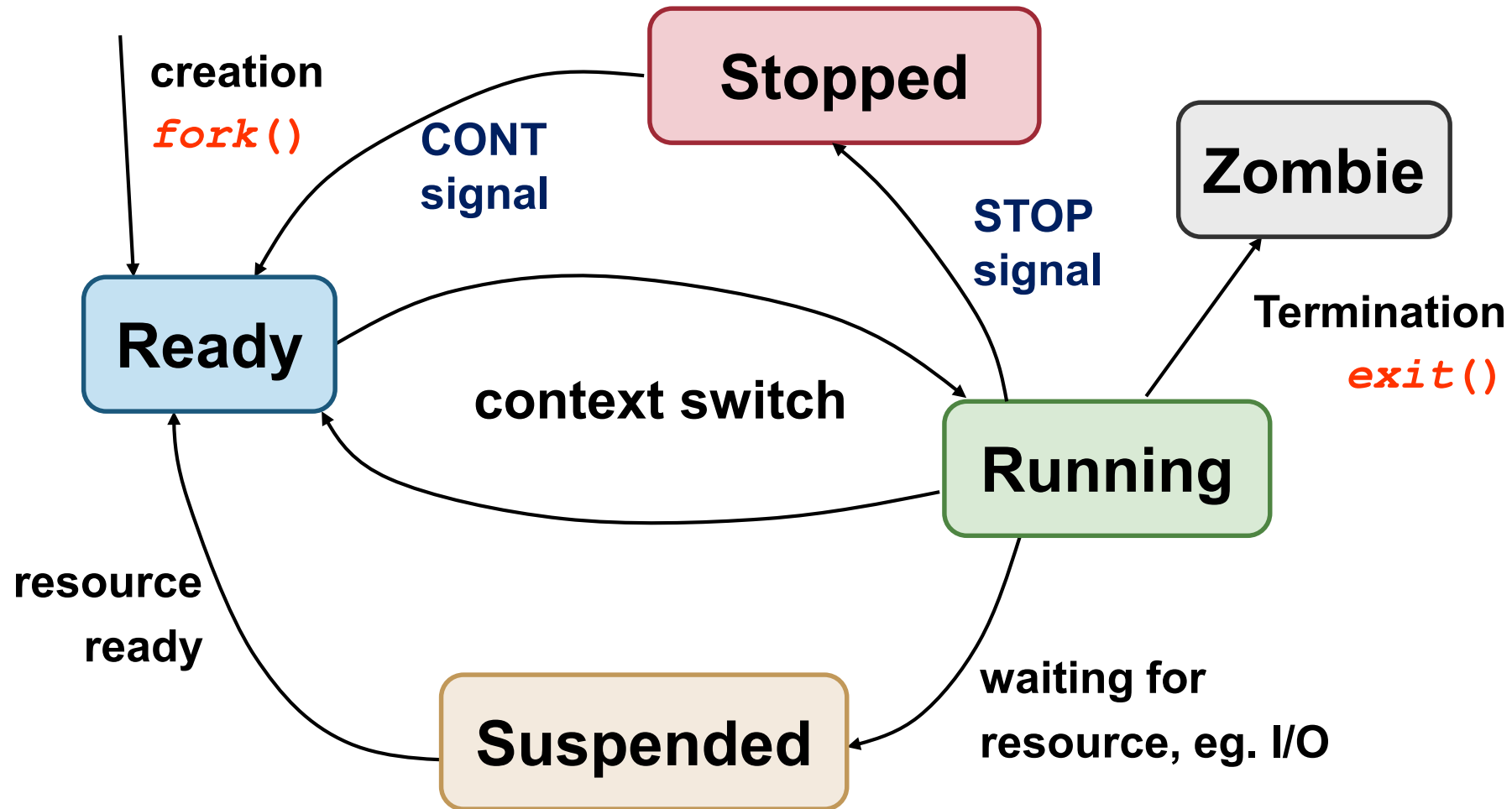
# Zombie Process (2 Cases)

1. Parent process terminates before child process:
  - ❑ `init` process becomes "pseudo" parent of child processes
  - ❑ Child termination sends signal to `init`, which utilizes `wait()` to cleanup
  
2. Child process terminates before parent but parent did not call `wait`:
  - ❑ Child process become a zombie process
  - ❑ Can fill up process table
    - May need a reboot to clear the table on older Unix implementations

# Summary of Unix Process System calls

- **fork()** :
  - Process creation
- **exec()** family:
  - Change executing image/program
  - **execl**, **execv**, **execve**, **execle**, **execvp**
- **exit()** :
  - Process termination
- **wait()** family:
  - Get exit status, synchronize with child
  - **wait**, **waitpid**, **waitid**, etc
- **getpid()** family:
  - Get process information
  - **getpid**, **getppid**, etc

# Process State Diagram in Unix



# IMPLEMENTATION ISSUES

# Implementing **fork()**

- Behavior of **fork()** :
  - ❑ Makes an almost exact copy of parent process
- Simplified implementation:
  1. Create address space of child process
  2. Allocate **p'** = new **PID**
  3. Create kernel process data structures
    - E.g. Entry in Process Table
  4. Copy kernel environment of parent process
    - E.g. Priority (for process scheduling)
  5. Initialize child process context:
    - **PID**= **p'** , **PPID**=**parent** id, zero CPU time

# Implementing **fork()** (cont)

6. Copy memory regions from parent
  - Program, Data, Stack
  - Very expensive operation that can be optimized (more later)
7. Acquires shared resources:
  - Open files, current working directory etc
8. Initialize hardware context for child process:
  - Copy registers, etc. from parent process
9. Child process is now ready to run
  - add to scheduler queue

# Memory Copy Operation

- Memory copy is very expensive:
  - ❑ Potentially need to copy the whole memory space
- Observations:
  - ❑ The child process will not access the whole memory range right away
  - ❑ Additionally:
    - If child just read from a location:
      - ❑ Remain unchanged
      - ❑ Can use a shared version
    - Only when write is perform on a location:
      - ❑ Then two independent copies are needed

# Memory Copy Optimization

- **Copy on Write** is a possible optimization for memory copy operation:
  - ❑ Only duplicate a “memory location” when it is written to
  - ❑ Otherwise parent and child share the same “memory location”
- Note that, actually:
  - ❑ Memory is organized into memory pages
    - A consecutive range of memory locations
  - ❑ Memory is managed on a page level
    - Instead of individual location
  - ❑ **Will be covered in details in Memory Management part of lecture**



# Modern Take on **fork()**

- **fork()** system call is part of the Unix design
  - inherited by most (all?) variants
- However, it is not versatile:
  - A thorough duplication of the parent process
- There are scenarios where a partial duplication may be preferred:
  - e.g. parent and child shares some of the memory regions, or some other resources
- Linux provides **clone()** which supersedes **fork()**

# Summary

- Covered most of the process operations available in Unix:
  - ❑ Creation through `fork()`
  - ❑ Change execution through `exec()`
  - ❑ Termination through `exit()`
  - ❑ Synchronization (Parent  $\leftrightarrow$  Child) through `wait()`
- Process States
  - ❑ Process state diagram
- Implementation issues with `fork()`