# Processes

# Processes

- What is a process?
- Birth
  - How do I make one?
- Life
  - Wait for one?
- Death
  - Kill one?




# Program or Process?

- Process
  - A process is the *context* (the information/data) maintained for an executing program
    - An executable instance of a program
  - A program can have many processes
  - Each process has a unique identifier
- Unix processes
  - Process #1 is known as the 'init' process (root of the process hierarchy)




# What makes up a process?

- Program code
- Machine registers
- Global data
- Stack
- Open files
- An environment




# Process Context

- Process ID (**`pid`**) — unique integer
- Parent process ID (**`ppid`**) — unique integer
- Current directory
- File descriptor table
- Environment — **`VAR=VALUE`** pairs
- Pointer to program code
- Pointer to data — Mem for global vars
- Pointer to stack — Mem for local vars
- Pointer to heap — Dynamically allocated memory
- Execution priority
- Signal information




# Unix Processes

- Address space
  - The address space is a section of memory that contains the code to execute as well as the process stack
- Set of data structures in the kernel to keep track of that process
  - Address space map
  - Current status of the process
  - Execution priority of the process
  - Resource usage of the process
  - Current signal mask
  - Owner of the process




# Process Lifetime

- Some processes run from system boot to shutdown
  - Servers & Daemons
    (e.g. Apache httpd server)
- Most processes come and go rapidly, as tasks start and complete
  - 'unit of work' on a modern computer
- A process can die a premature, even horrible death (say, due to a crash)




# Know your process

- Each process has a unique identifier

```
int myid = getpid()
```

What is wrong with this?




# Know your process

- better…

```
pid_t myid = getpid()
```

  - `pid_t: int` in linux,
  - `pid_t: long` in other systems

- Know your parent

```
pid_t myparentid = getppid()
```




# Process Creation

- On creation, process needs resources
  - CPU, memory, files, I/O devices
- Get resources from the OS or from the parent process
  - Child process is restricted to a subset of parent resources
  - Prevents many processes from overloading system




# Process Creation

- Execution options
  - Parent continues concurrently with child
  - Parent waits until child has terminated
- Address space options
  - Child process is duplicate of parent process
  - Child process has a new program loaded into it



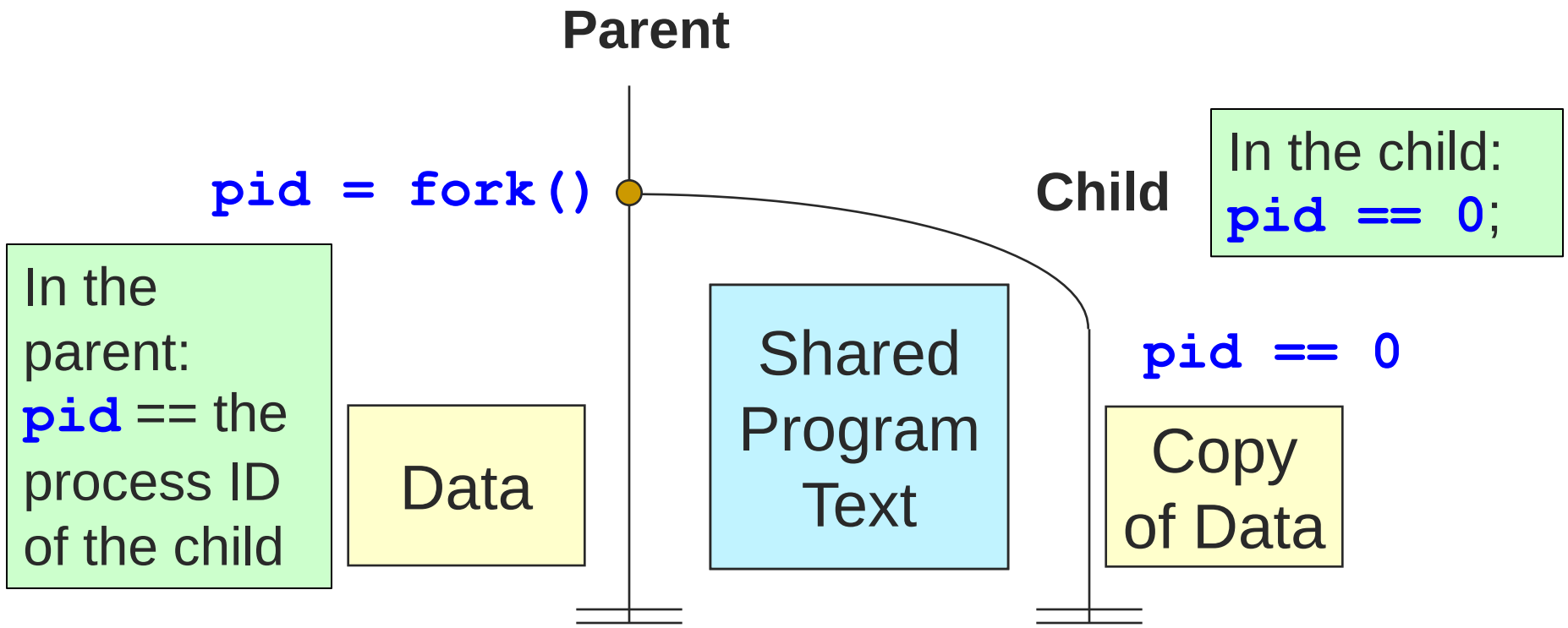

# Creating a Process – `fork()`

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

- Create a child process
  - The child is an (almost) exact copy of the parent
  - The new process and the old process both continue in parallel from the statement that follows the `fork()`
- Returns:
  - To child
    - 0 on success
  - To parent
    - process ID of the child process
    - -1 on error, sets `errno`




# Creating a Process – `fork()`




A program can use this `pid` difference to do different things in the parent and child

# Example – `fork()`

```
int pid;
int status = 0;

if (pid = fork()) {
   /* parent */
   …..
   pid = wait(&status);
} else {
   /* child */
   …..
   exit(status);
}
```

`fork` returns twice:
Parent: `pid` == child process ID (pid)
Child: `pid` == 0

Parent uses `wait` to sleep until the child exits.
`wait` returns child pid and status.




# Creating a Process – `fork()`

- The child process is an exact copy of the parent process except
  - The child process has a unique process ID
  - The child process has a different parent process ID (i.e., the process ID of the calling process)
  - The child process has its own copy of the parent's file descriptors
  - and some other stuff about memory and stuff that we'll learn later …




# Example – `fork()`

Challenge:

write code so that child prints

```
'CHILD: my id is ___ and my parent id is ___'
```

and parent prints

```
'PARENT:my id is ___ and the child's id is ___'
```





# How does `fork` work?

- Parent

```
mypid = 4, myppid = 1

int forked_pid , wait_pid;
int status = 0;

if (forked_pid = fork()) {
    /* parent */
    …..
    wait_pid = wait(&status);
} else {
    /* child */
    …..
    exit(status);
}
```

- Child

```
mypid = 6, myppid = 4

int forked_pid, wait_pid;
int status = 0;

if (forked_pid = fork()) {
    /* parent */
    …..
    wait_pid = wait(&status);
} else {
    /* child */
    …..
    exit(status);
}
```



# How does `fork` really work?

- Parent

  `mypid = 4, myppid = 1`

- Child

  `mypid = 6, myppid = 4`

```
int forked_pid , wait_pid;
int status = 0;

if (forked_pid = fork()) {
    /* parent */
    …..
    wait_pid = wait(&status);
} else {
    /* child */
    …..
    exit(status);
}
```



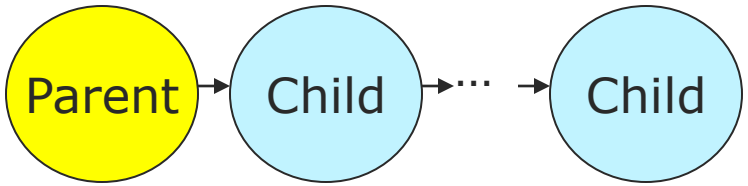
# Chain and Fan

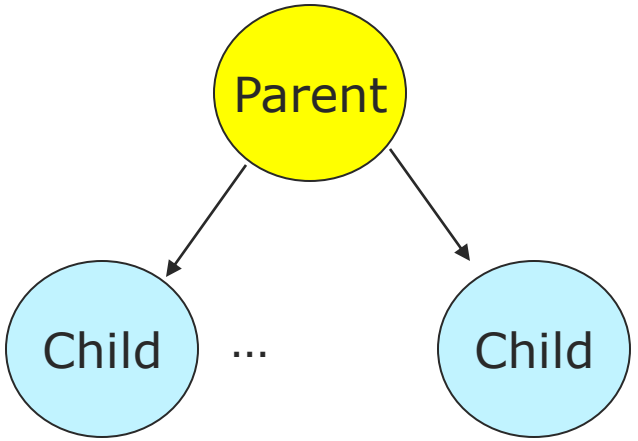
**Chain**

- Write code to make chain

**Fan**

- Code to make N children of one parent process?





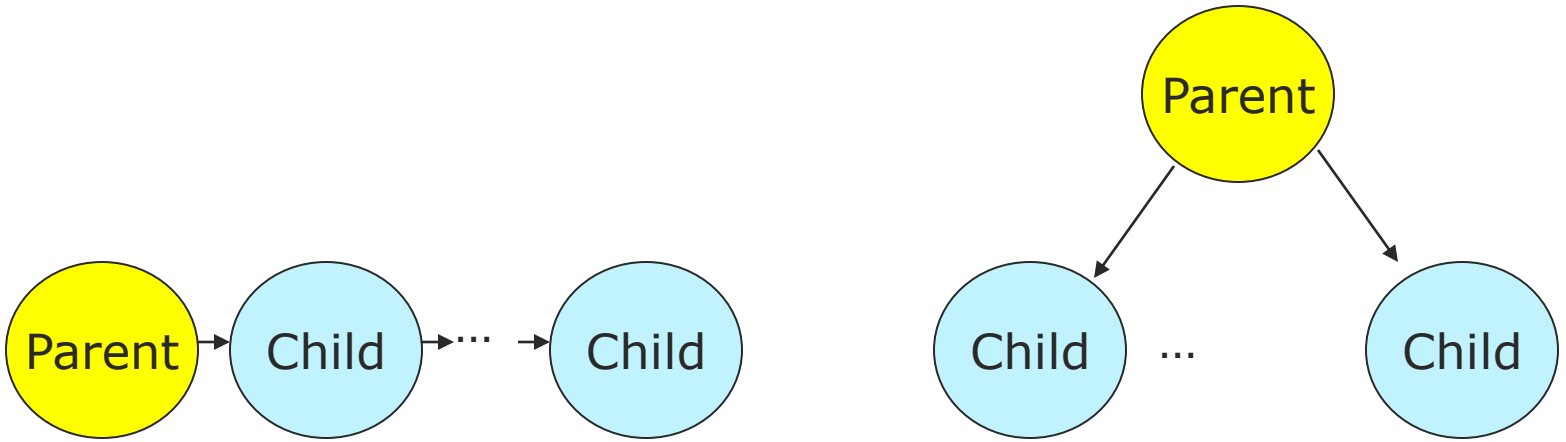



# Chain and Fan

**Chain**

```
pid_t childpid = 0;
for (i=1;i<n;i++)
   if (childpid = fork())
      break;
```

**Fan**







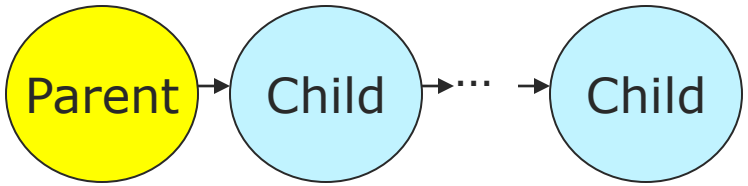

# Chain and Fan

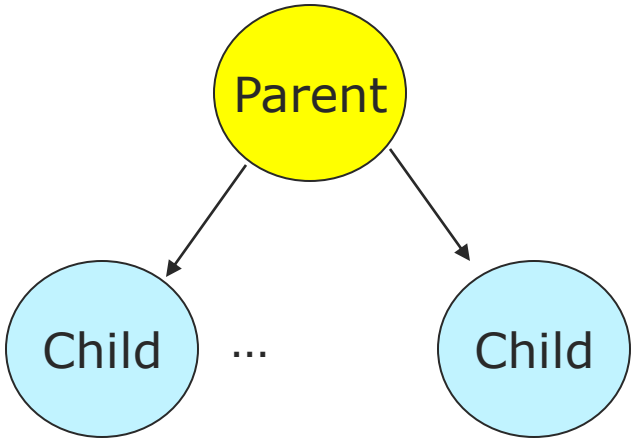
## Chain

```
pid_t childpid = 0;
for (i=1;i<n;i++)
    if (childpid = fork())
        break;
```

## Fan

```
pid_t childpid = 0;
for (i=1;i<n;i++)
    if ((childpid = fork())
        <=0)
        break;
```








# Example – `fork()`

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

int main() {
   pid_t pid;           /* could be int */
   int i;
   pid = fork();
```





# Example – `fork()`

```
    if( pid > 0 ) {/* parent */
        for( i=0; i < 1000; i++ )
              printf("\t\t\tPARENT %d\n", i);
    } else { /* child */
        for(i=0; i < 1000; i++)
              printf( "CHILD %d\n", i );
    }
    return 0;
}
```

What will the output be?





# Example – `fork()` Notes

- `i` is copied between parent and child
- Switching between parent and child depends on many factors
  - Machine load, system process scheduling
- I/O buffering effects amount of output shown
- Output interleaving is nondeterministic
  - Cannot determine output by looking at code




# Waiting for a child to finish – `wait()`

```
#include <sys/types.h>
#include <wait.h>
pid_t wait(int *status);
```

- Suspend calling process until child has finished
- Returns:
  - Process ID of terminated child on success
  - -1 on error, sets **`errno`**
- Parameters:
  - **`status`**: status information set by **`wait`** and evaluated using specific macros defined for **`wait`**.




# Waiting for any child to finish

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

pid_t childpid;

childpid = wait(NULL);
if (childpid != -1)
   printf("waited for child with pid %ld\n",
      childpid);
```

(see "**man 2 wait**")





# `wait()` Function

- Allows parent process to wait (block) until child finishes
- Causes the caller to suspend execution until child's status is available

| `errno` | cause |
|---|---|
| `ECHILD` | Caller has no unwaited-for children |
| `EINTR` | Function was interrupted by signal |
| `EINVAL` | Options parameter of waitpid was invalid |




# Waiting for a child to finish – `waitpid()`

```
#include <sys/types.h>
#include <wait.h>
pid_t waitpid(pid_t pid, int *status, int
    options);
```

- Suspend calling process until child specified by **`pid`** has finished
- Returns:
  - Process ID of terminated child on success
  - 0 if **`WNOHANG`** and no child available, sets **`errno`**
  - -1 on error, sets **`errno`**
- Parameters:
  - **`status`**: status information set by **`wait`** and evaluated using specific macros defined for **`wait`**.




# Waiting for a child to finish – `waitpid()`

```
#include <sys/types.h>
#include <wait.h>
pid_t waitpid(pid_t pid, int *status, int
    options);
```

- Suspend calling process until child specified by **`pid`** has finished
- Parameters:
  - **`pid`**:
    - < -1: wait for any child process whose process group ID is equal to the absolute value of **`pid`**.
    - -1 wait for any child process (same as **`wait`**)
    - 0 wait for any child process whose process group ID is equal to that of the calling process.
    - > 0 wait for the child whose process ID is equal to the value of **`pid`**.




# Waiting for a child to finish – `waitpid()`

```
#include <sys/types.h>
#include <wait.h>
pid_t waitpid(pid_t pid, int *status, int
    options);
```

- Suspend calling process until child specified by **`pid`** has finished
- Parameters:
  - **`options`**:
    - **`WNOHANG`**: return immediately if no child has exited.
    - **`WUNTRACED`**: return for children that are stopped, and whose status has not been reported.




# Process Termination

- Upon completion of last statement
  - A process automatically asks the OS to delete it
  - All of the child's resources are de-allocated
  - Child process may return output to parent process
- Other termination possibilities: Aborted by parent process
  - Child has exceeded its usage of some resources
  - Task assigned to child is no longer required
  - Parent is exiting and OS does not allow child to continue without parent




# Process Termination

- Voluntary termination
  - Normal exit
    - End of **`main()`**
    - **`exit(0);`**
  - Error exit
    - **`exit(2)`**
- Involuntary termination
  - Fatal error
    - Divide by 0, core dump / seg fault
  - Killed by another process
    - **`kill`** procID, end task




# How to List all Processes?

- On Windows: run Windows task manager
  - Hit Control+Shift+Esc
  - Click on the “processes” tab
- On UNIX
  - `> ps -e` also, `pstree`
  - Try “`man ps`”




# Example – `fork()`

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

int main() {
     pid_t pid;            /* could be int */
     int i;
     pid = fork();
     if( pid > 0 ) {       /* parent */
          for( i=0; i < 1000; i++ )
          printf("\t\t\tPARENT %d\n", i);
     }
     else { /* child */
          for(i=0; i < 1000; i++)
                    printf( "CHILD %d\n", i );
     }
     return 0;
}
```

How can you use **`ps`** to see the processes that are created?





# Next Week: Memory!