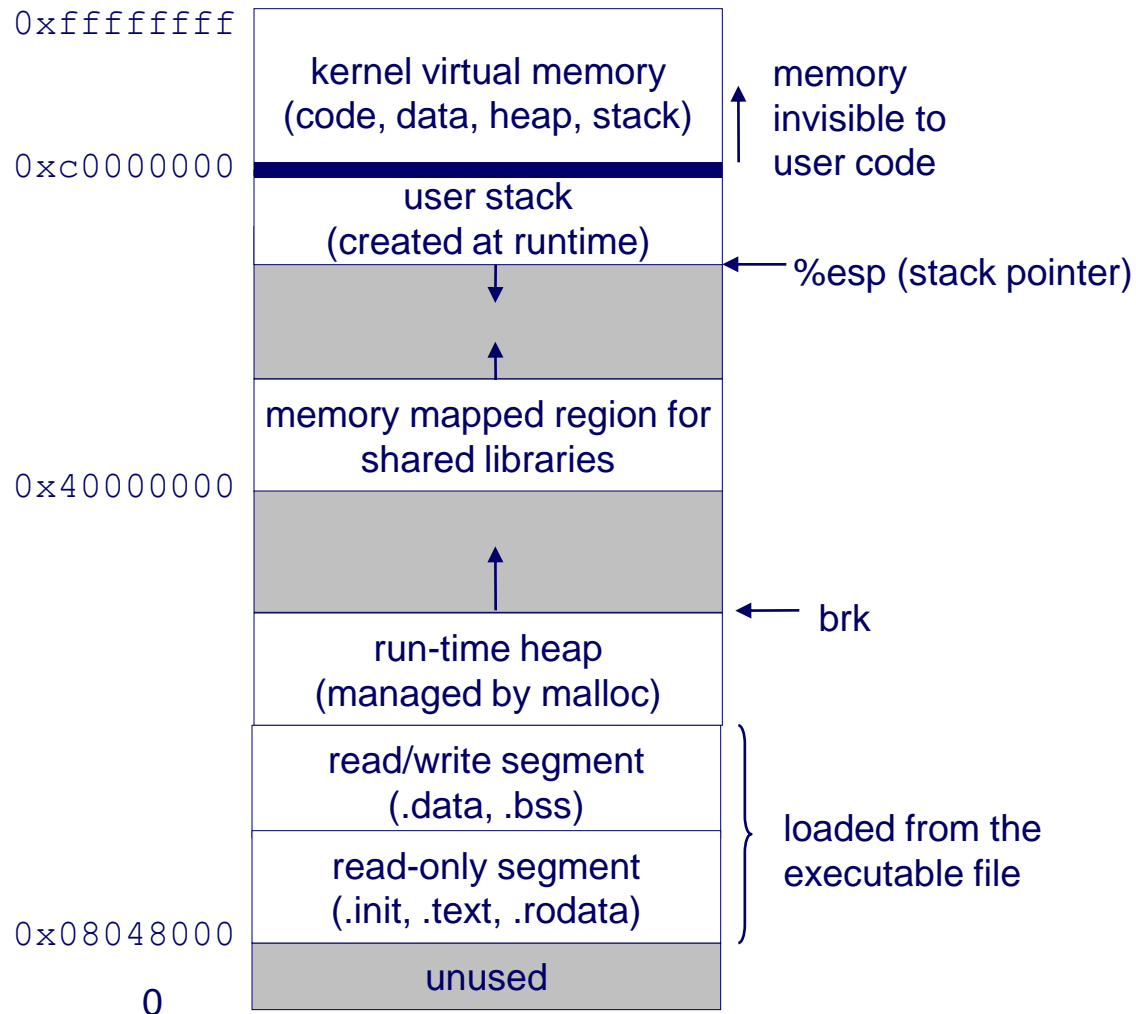


# Private Address Spaces

Each process has its own private address space.



# fork: Creating new processes

`int fork(void)`

- creates a new process (child process) that is identical to the calling process (parent process)
- returns 0 to the child process
- returns child's pid to the parent process

```
if (fork() == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

Fork is interesting  
(and often confusing)  
because it is called  
*once* but returns *twice*

# Fork Example #1

## Key Points

- Parent and child both run same code
  - Distinguish parent from child by return value from `fork`
- Start with same state, but each has private copy
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

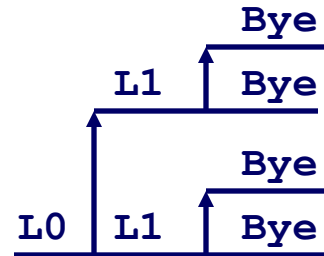
```
void fork1()  
{  
    int x = 1;  
    pid_t pid = fork();  
    if (pid == 0) {  
        printf("Child has x = %d\n", ++x);  
    } else {  
        printf("Parent has x = %d\n", --x);  
    }  
    printf("Bye from process %d with x = %d\n", getpid(), x);  
}
```

# Fork Example #2

## Key Points

- Both parent and child can continue forking

```
void fork2()  
{  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("Bye\n");  
}
```

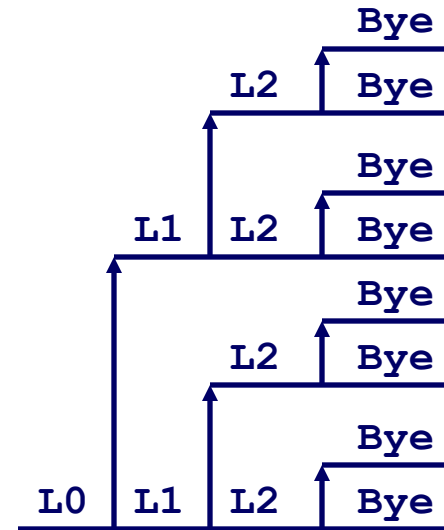


# Fork Example #3

## Key Points

- Both parent and child can continue forking

```
void fork3()  
{  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("L2\n");  
    fork();  
    printf("Bye\n");  
}
```

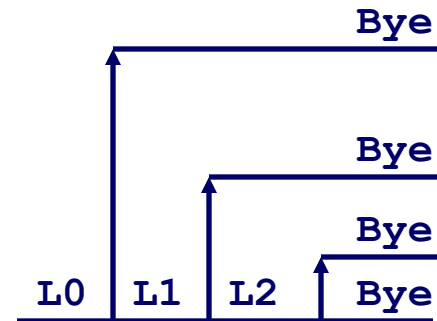


# Fork Example #4

## Key Points

- Both parent and child can continue forking

```
void fork4()  
{  
    printf("L0\n");  
    if (fork() != 0) {  
        printf("L1\n");  
        if (fork() != 0) {  
            printf("L2\n");  
            fork();  
        }  
    }  
    printf("Bye\n");  
}
```

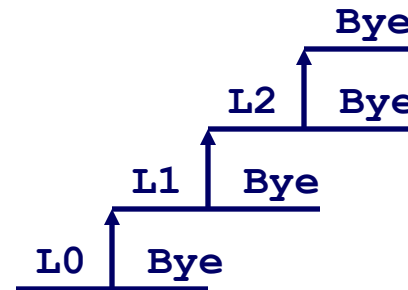


# Fork Example #5

## Key Points

- Both parent and child can continue forking

```
void fork5()  
{  
    printf("L0\n");  
    if (fork() == 0) {  
        printf("L1\n");  
        if (fork() == 0) {  
            printf("L2\n");  
            fork();  
        }  
    }  
    printf("Bye\n");  
}
```



# exit: Destroying Process

`void exit(int status)`

- exits a process
  - Normally return with status 0
- `atexit()` registers functions to be executed upon exit

```
void cleanup(void) {
    printf("cleaning up\n");
}

void fork6() {
    atexit(cleanup);
    fork();
    exit(0);
}
```

# Zombies

## Idea

- When process terminates, still consumes system resources
  - Various tables maintained by OS
- Called a “zombie”
  - Living corpse, half alive and half dead

## Reaping

- Performed by parent on terminated child
- Parent is given exit status information
- Kernel discards process

## What if Parent Doesn't Reap?

- If any parent terminates without reaping a child, then child will be reaped by `init` process
- Only need explicit reaping for long-running processes
  - E.g., shells and servers

# Zombie Example

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6639 ttyp9        00:00:03 forks
 6640 ttyp9        00:00:00 forks <defunct>
 6641 ttyp9        00:00:00 ps
linux> kill 6639
[1]      Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6642 ttyp9        00:00:00 ps
```

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
            getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
            getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

- ps shows child process as “defunct”
- Killing parent allows child to be reaped

# Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
              getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
              getpid());
        exit(0);
    }
}
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
```

```
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6676 ttyp9        00:00:06 forks
 6677 ttyp9        00:00:00 ps
```

```
linux> kill 6676
```

```
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6678 ttyp9        00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

# `wait`: Synchronizing with children

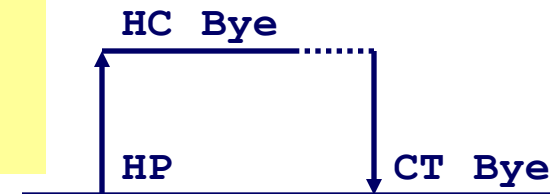
```
int wait(int *child_status)
```

- suspends current process until one of its children terminates
- return value is the `pid` of the child process that terminated
- if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated

# wait: Synchronizing with children

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
    }
    else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    exit();
}
```



# Wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

# Waitpid

- `waitpid(pid, &status, options)`
  - Can wait for specific process
  - Various options

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

# Wait/Waitpid Example Outputs

## Using wait (fork10)

```
Child 3565 terminated with exit status 103  
Child 3564 terminated with exit status 102  
Child 3563 terminated with exit status 101  
Child 3562 terminated with exit status 100  
Child 3566 terminated with exit status 104
```

## Using waitpid (fork11)

```
Child 3568 terminated with exit status 100  
Child 3569 terminated with exit status 101  
Child 3570 terminated with exit status 102  
Child 3571 terminated with exit status 103  
Child 3572 terminated with exit status 104
```

# exec: Running new programs

```
int execl(char *path, char *arg0, char *arg1, ..., 0)
```

- loads and runs executable at `path` with args `arg0`, `arg1`, ...
  - `path` is the complete path of an executable
  - `arg0` becomes the name of the process
    - » typically `arg0` is either identical to `path`, or else it contains only the executable filename from `path`
  - “real” arguments to the executable start with `arg1`, etc.
  - list of args is terminated by a `(char *)0` argument
- returns `-1` if error, otherwise doesn't return!

```
main() {
    if (fork() == 0) {
        execl("/usr/bin/cp", "cp", "foo", "bar", 0);
    }
    wait(NULL);
    printf("copy completed\n");
    exit();
}
```

# Summarizing

## Exceptions

- Events that require nonstandard control flow
- Generated externally (interrupts) or internally (traps and faults)

## Processes

- At any given time, system has multiple active processes
- Only one can execute at a time, though
- Each process appears to have total control of processor + private memory space

# Summarizing (cont.)

## Spawning Processes

- Call to `fork`
  - One call, two returns

## Terminating Processes

- Call `exit`
  - One call, no return

## Reaping Processes

- Call `wait` or `waitpid`

## Replacing Program Executed by Process

- Call `exec1` (or variant)
  - One call, (normally) no return