Using processes

fork
exec
wait

using processes

- fork();
  - called once
  - returns twice
  - sets **`errno`** if error occurs
    - no more processes
    - no swap space

parent and child run concurrently

output sequence

printf vs. write

printf buffers output

- flush output
  - write output to physical device
    - close file
    - full buffer
    - `fflush()` system call or end-of-line
    - `sync()` system call

using processes

- exec
  - replace current process image
  - used when creating a child to carry out a specific request
    - lpd
    - syslogd
    - xinetd
    - httpd
  - exec calls do **not** return

using processes

- Overlays
  - Text
  - Data
  - Stack
- Preserves
  - U area

u-area

maintained by operating system

one per process

specific process information

open files

current directory

signal actions are reset to default

accounting information

a system stack segment

system calls

process can access information

via system calls

using processes

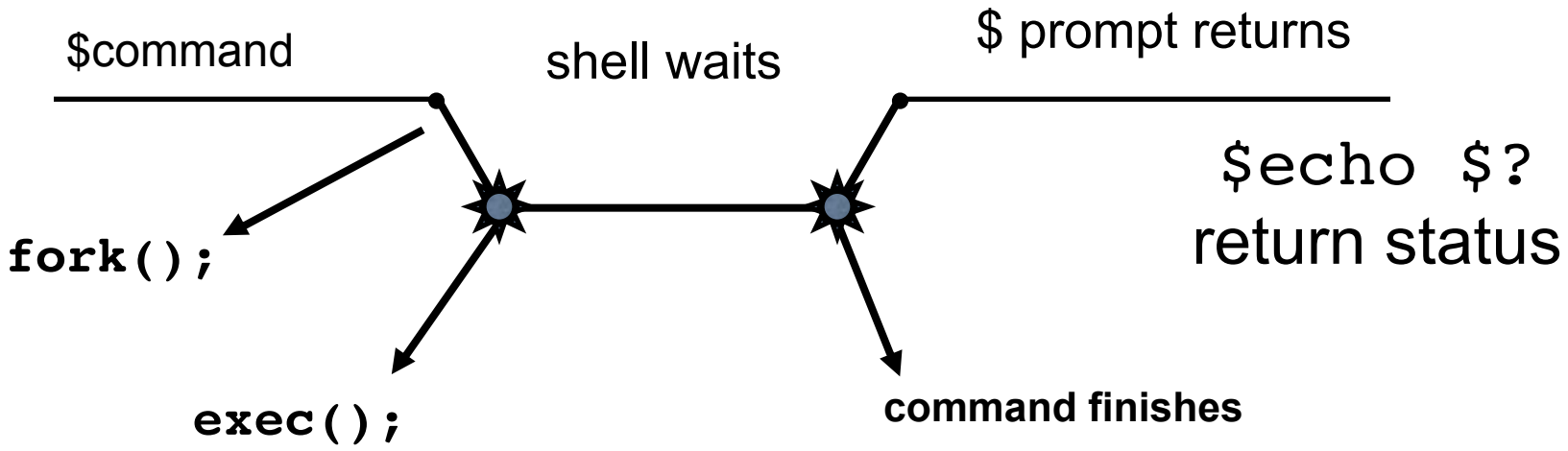
- Signals are reset
  - Default action
  - Address of signal catching routine may not be present
- Profiling is turned off
- If program is SUID
  - EUID and EGID are set accordingly

using processes

```
pid  = fork ();
if ( pid == 0 ){
   // in child process
   exec(…);
   // error
}
// in parent process
// may wait for child process
```

- Example

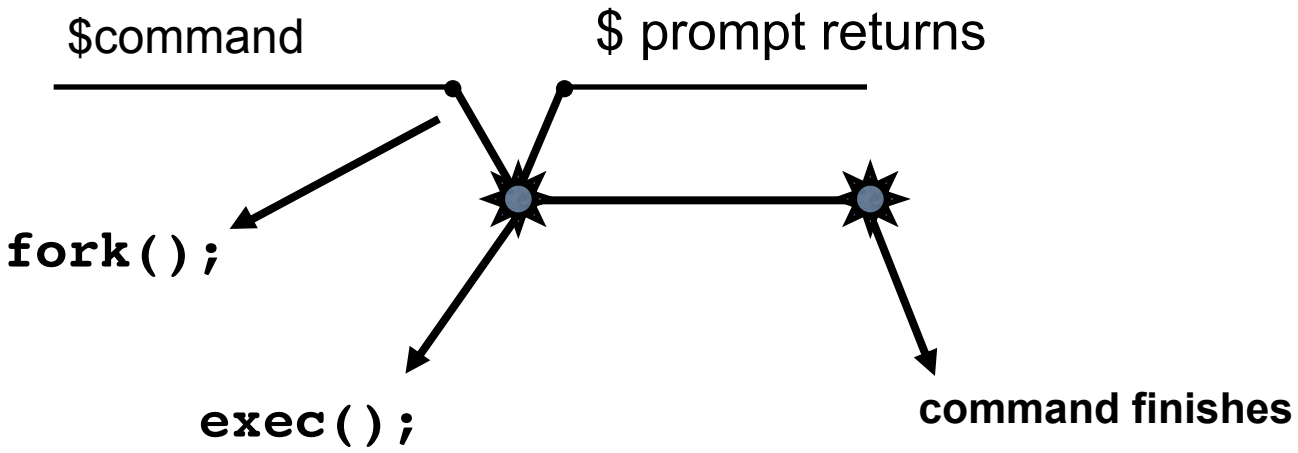
%cat file.txt > file2.txt



What happens with:
`%exec date`

- Example

%cat file.txt > file2.txt &



- exec system call

Table 3.1 The exec Call Prototypes.

```
#include <unistd.h>

extern char **environ;

int execl (const char *path, const char *arg, ...);
int execv (const char *path, char *const argv[]);

int execle(const char *path, const char *arg , ..., char * const envp[]);
int execve(const char *path, char *const argv[],    char * const envp[]);

int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
```

l: list of arguments
v: vector or array and pass own environment variables
e: pass own environment variables as 3rd argument
p: use current environment PATH

execl, execv, execle, execve : fully qualified path is required

- `

Table 3.2 exec Call Functionality.

Library Call Name	Argument Format	Pass Current Set of Environment Variables?	Search of PATH Automatic?
execl	list	yes	no
execv	array	yes	no
execle	list	no	no
execve	array	no	no
execlp	list	yes	yes
execvp	array	yes	yes

- **`execlp()`**
  - ***`file`*** is a pointer to the file that contains the program code
    - relative or absolute path
  - absolute
    - seems superfluous since uses PATH
    - PATH used for other arguments
    - file references within the code
  - use conventions when passing arguments

arg[0] : program name
arg[1] : first parameter
arg[2] : second parameter

```
execlp ("/bin/cat", "cat", argv[1], (char * ) NULL );
perror("execlp failure");
```

using processes

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

int main ( int argc, char *argv[] ) {
 if ( argc > 1 ) {
   execlp ("/bin/cat", "cat", argv[1], (char * ) NULL );
   perror("execlp failure");
   return 1;
 }

 printf("Usage: %s text file\n", argv[0]);
 return 2;
}
```

- execvp
  - ***`file`*** is a pointer to the file that contains the program code
    - relative or absolute path
  - second argument: `char *const argv []`
    - array of pointers to character strings
      - `argv` style
        - null terminated

using processes

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int
main ( int argc, char *argv[] ) {
 if ( argc > 1 ) {
   execvp ( argv[1], &argv[1] );
   perror("execvp failure");
   return 1;
 }
 printf("Usage: %s  exec [arg(s)]\n”, argv[0]);
 return 2;
}
```

```
./a.out cat p3.6.cc
./a.out cat -n p3.6.cc
```

using processes

- create command in program

```
#include <iostream>
#include <cstdio>
#include <unistd.h>
using namespace std;
int
main ( int argc, char *argv[] ) {
  char  *new_arv[ ] = {“cat”, “myfile.txt”, (char *) 0 };
  execvp ( “/bin/cat” , new_arv );
  perror("execvp failure");
  return 1;
}
```

- Ending a process
  - Calls exit
  - Issues a return
  - Falls to the end of function main

```
exit( int status);
```

calls functions registered by `atexit`

in reverse order

By **convention**: a zero for normal termination

- `atexit` system call

```
int atexit(void (*function) (void) );
```

register exit function

returns to exit call;

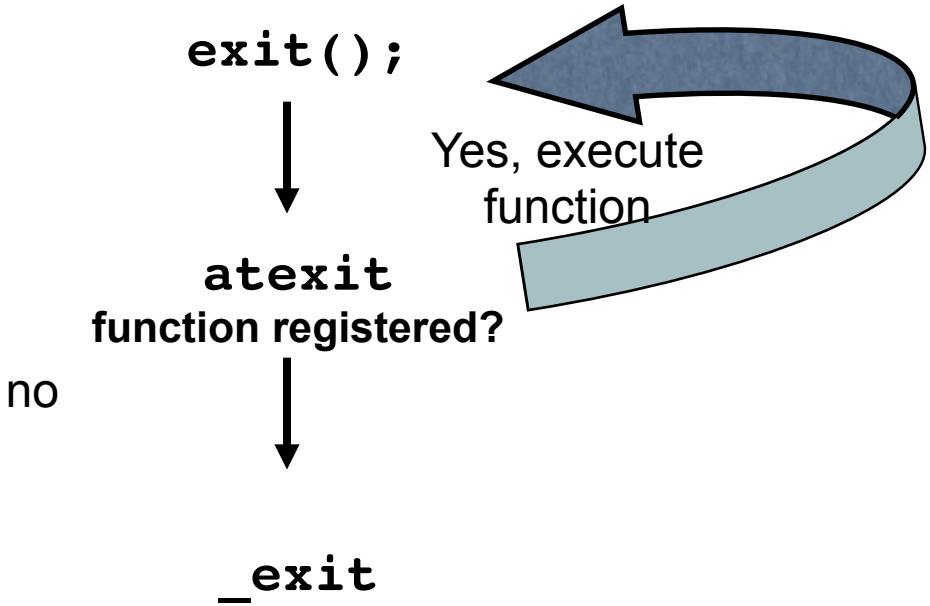
when all `atexit` function were called

no parameters

called in reverse order

- If successful registering function

returns 0



- atexit example

```
main () {
void f1(), f2(), f3;
atexit (f1);
atexit (f2);
atexit (f3);
cout << “Getting ready to exit” << endl;
exit(0);
}
void f1(); {
   cout << “Doing f1” << endl
}
void f2(); {
   cout << “Doing f2” << endl
}
void f3(); {
   cout << “Doing f3” << endl
}
```

```
Getting ready to exit
Doing f3
Doing f2
Doing f1
```

_exit:

file descriptors are closed

parent process notified

SIGCHLD

status information returned

if no parent waiting, status information is stored until wait is issued by parent

children of terminating process inherited by `init`

if group leader

signals SIGHUP/SIGCONT sent

using processes

`zombie` process is a process that completed execution but still in process table.

When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains. The parent can read the child's exit status by executing the wait system call, at which stage the zombie is removed

Problem
What is a UNIX **`zombie`** process and how do you kill these processes?

Solution
Defunct processes, also known as "zombie" processes, are those that for some reason lose the handle to the parent, or the parent loses the handle to the child process, so the last step of cleanup does not occur when the process finishes.

When a process dies, it becomes a zombie process. Normally, the parent performs a wait() and cleans up the PID. But the parent can handle only one signal at a time, and sometimes the parent receives too many SIGCHLD signals at once. It is possible to resend the signal on behalf of the child via kill -18 PPID. Killing the parent or rebooting will also clean up zombies. The better resolution is to fix the faulty parent code that failed to perform the wait() properly.

Usually keeping the OS and the application up to par with respect to patches should take care of the problem; however, occasionally you may encounter zombies. It is not critical that they be cleaned up, as they are inactive and generally harmless.

using processes

- waiting for a process

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

child_pid on success

-1 on failure

no child process

block if active child

zombie: process terminates and no waiting parent

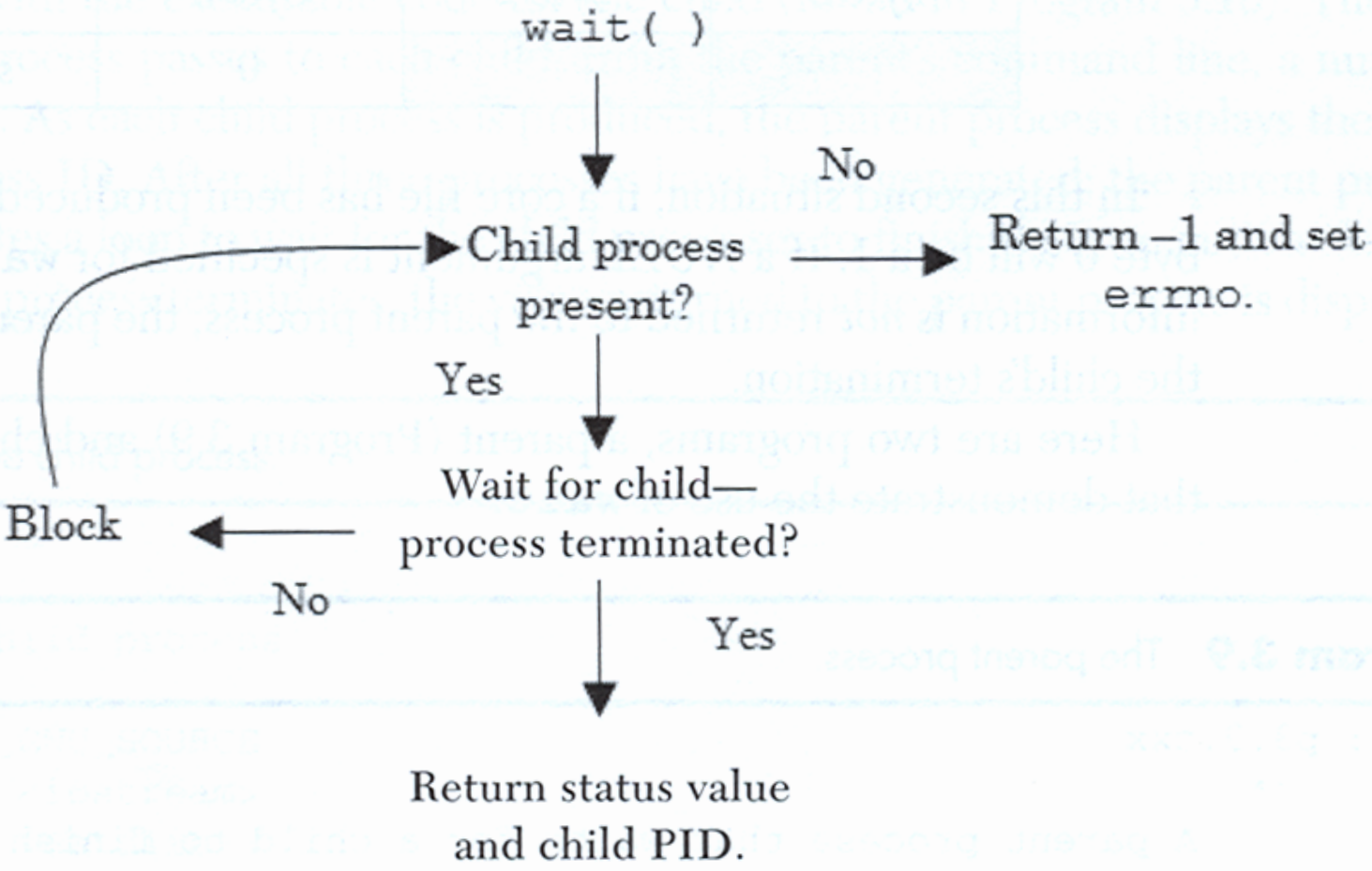


Figure 3.11 Summary of `wait` activities.

Normal termination

Byte3	Byte2	Byte 1	Byte 0
		exit code	0

Terminate due to uncaught signal

Byte3	Byte2	Byte 1	Byte 0
		0	signal #

using processes

- limitations
  - returns pid of first process to terminate
  - always blocks if status information is not available

using processes

- pid_t waitpid(pid_t pid, int *status, int options);
- returns child PID or 0

pid: child to wait for

< -1: any child process whose **process group id** equals the absolute value of **pid**

-1: any child process (similar to wait)

0: any child process whose **process group** id equals the **caller's process group id**

> 0 : child process with this process ID

options:

0: dont'care

WNOHANG: return immediately if no child has exited, returns 0

WUNTRACED: return immediately if child is blocked