Process control

Process

- Process
  - a program in execution
  - program image + environment
    - text(code) /data / stack / heap segment
    - kernel data structure, address space, open files, ...

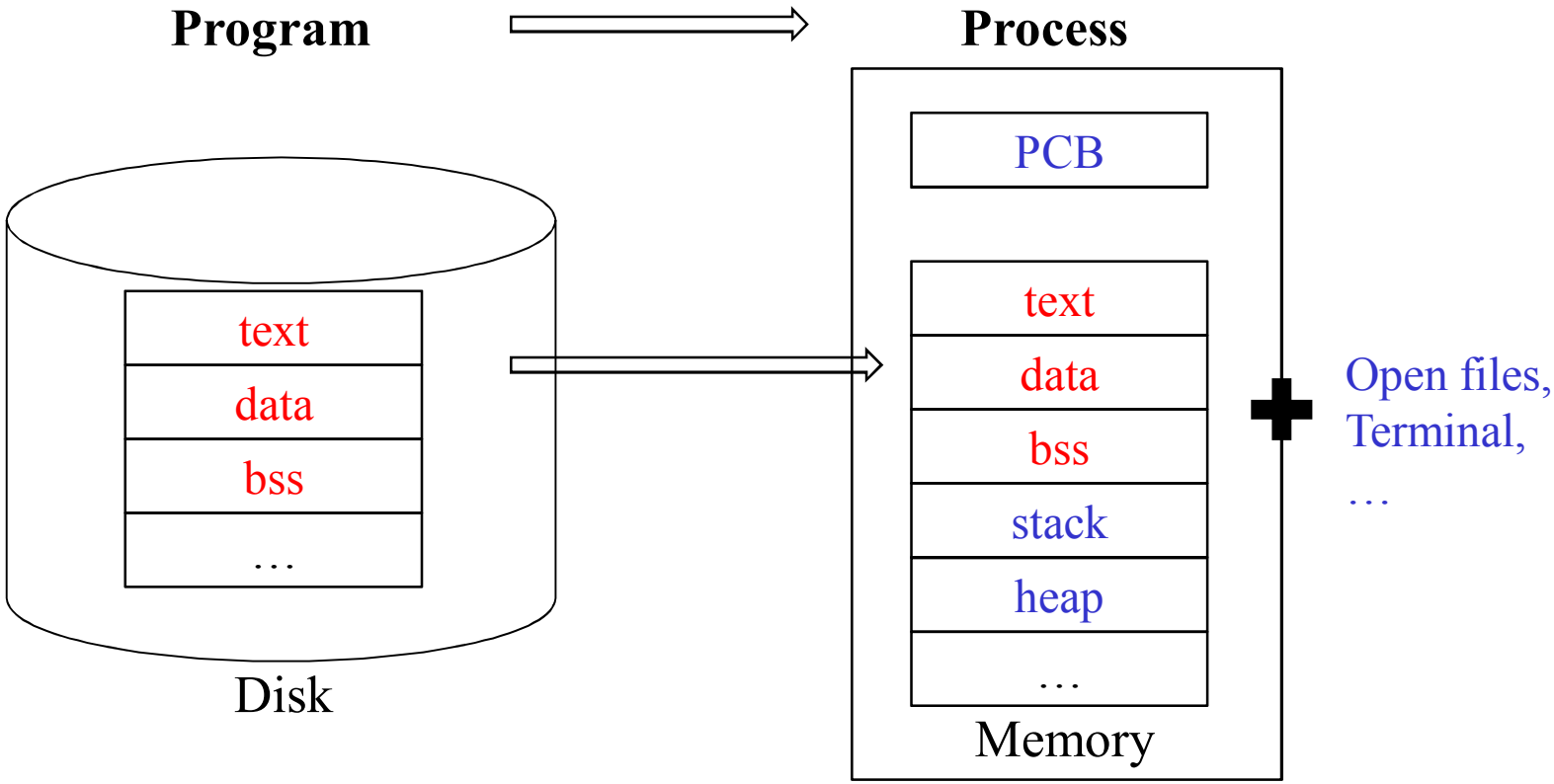
Process

```
/* test_program.c */
#include <stdio.h>
int a,b;
int glob_var = 3;

void test_func(void) {
    int local_var, *buf;

    buf = (int *) malloc(10,000 * sizeof(int));
    …
}

int main(int argc, char *argv[]) {
    int i = 1;
    int local_var;

    a = i + 1;
    printf("value of a = %d\n", a);
    test_func();
}
```

$ gcc –o test_program test_program.c

Process

- $ ./test_program



* bss: Block Started by Symbol
  PCB: Process Control Block

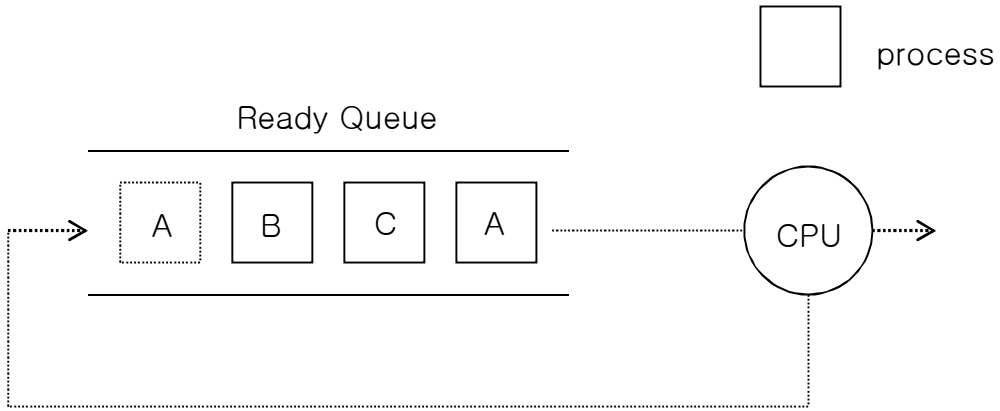
Process

- UNIX is a multitasking system.

```
$ ps –e
PID TTY          TIME CMD
1 ?        00:00:00 init
2 ?        00:00:00 migration/0
3 ?        00:00:00 ksoftirqd/0
4 ?        00:00:00 watchdog/0
5 ?        00:00:00 migration/1
6 ?        00:00:00 ksoftirqd/1
7 ?        00:00:00 watchdog/1
8 ?        00:00:00 migration/2
…
29122 pts/9    00:00:00 bash
29151 ?        00:00:00 sshd
29170 ?        00:00:00 sshd
29171 pts/10   00:00:00 bash
30465 ?        00:00:00 httpd
$
```

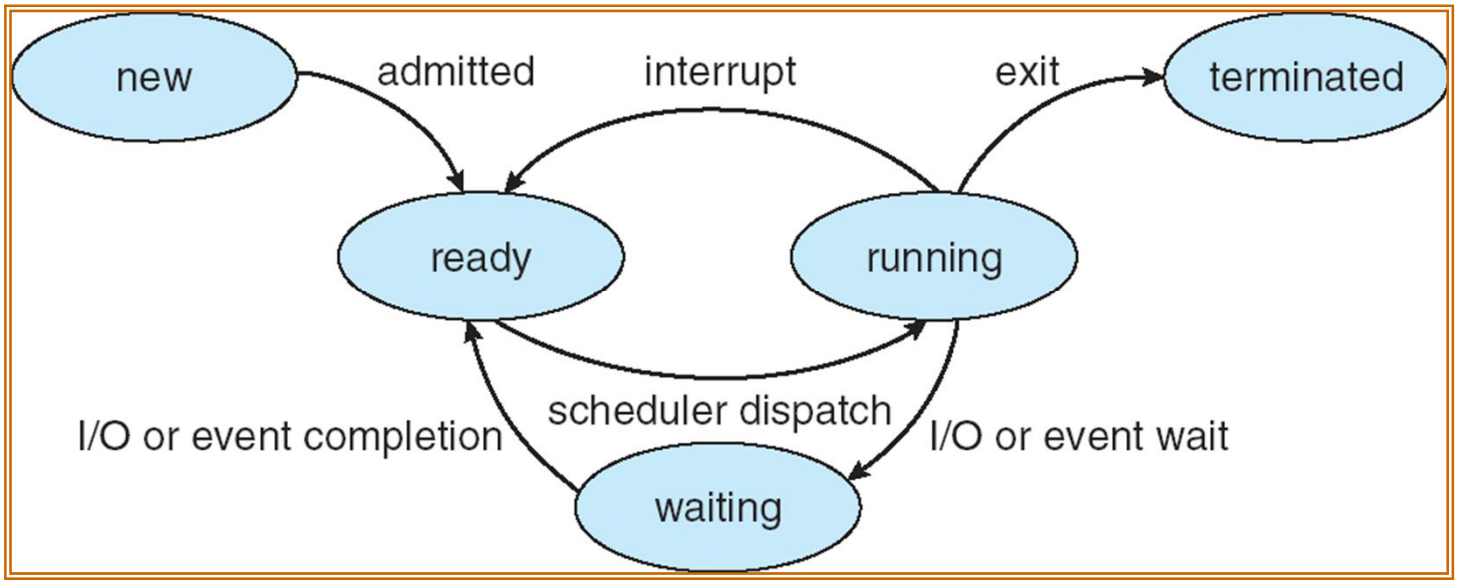
Process

- UNIX is a time sharing system.



Process

- State of a process



Process identifiers

- process id
  - Every process has a unique process ID.
  - As processes terminate, IDs become candidates for reuse.

```
$ ps –e
PID TTY          TIME CMD
1 ?        00:00:00 init
2 ?        00:00:00 migration/0
3 ?        00:00:00 ksoftirqd/0
4 ?        00:00:00 watchdog/0
5 ?        00:00:00 migration/1
…
29122 pts/9    00:00:00 bash
29151 ?        00:00:00 sshd
29170 ?        00:00:00 sshd
29171 pts/10   00:00:00 bash
30465 ?        00:00:00 httpd
$
```

Process identifiers

- Process 0 (scheduler process or swapper)
  - Part of the kernel.
  - System process.

- Process 1 (init process)
  - /sbin/init
  - Invoked at the end of the bootstrap procedure.
  - Initialize the UNIX system with /etc/rc* or /etc/inittab.
  - Never dies.
  - User process with superuser privilege.

Process identifiers

```
#include <unistd.h>

pid_t getpid(void);
                        Returns: process ID of calling process
pid_t getppid(void);
                        Returns: parent process ID of calling process
uid_t getuid(void);
                        Returns: real user ID of calling process
uid_t geteuid(void);
                        Returns: effective user ID of calling process
gid_t getgid(void);
                        Returns: real group ID of calling process
gid_t getegid(void);
                        Returns: effective group ID of calling process
```

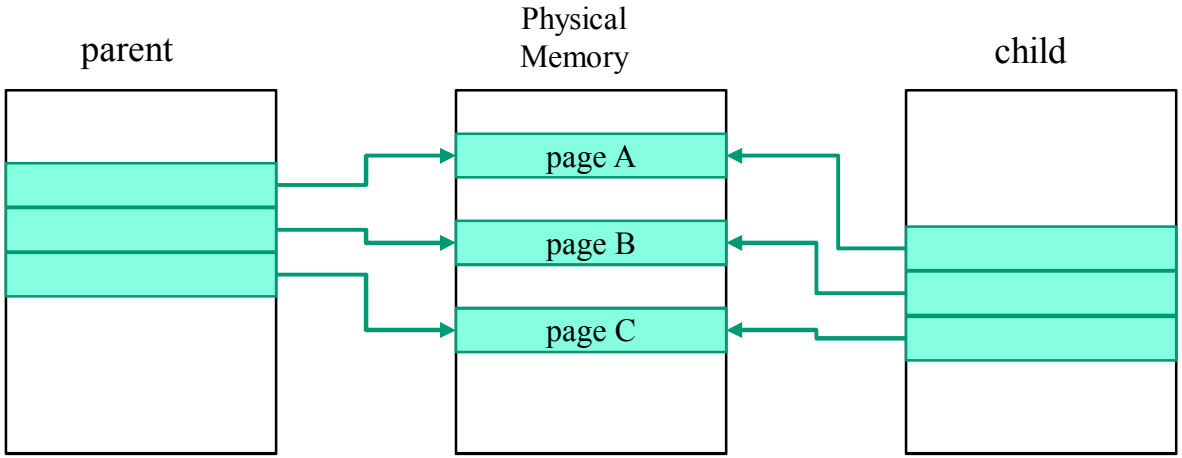
fork()

```
#include <unistd.h>

pid_t fork(void);
                    Returns: 0 in child, process ID of child in parent, -1 on error
```

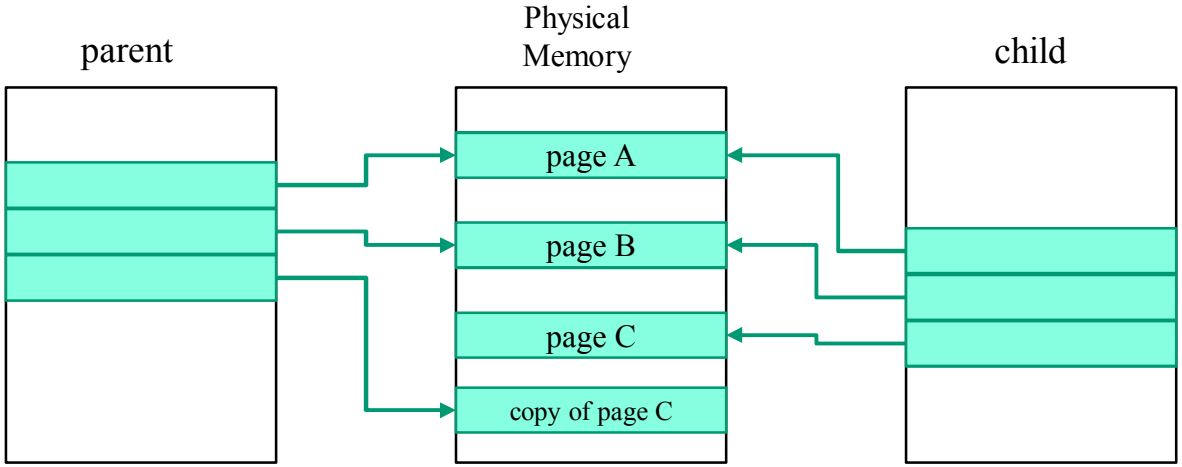
- Create a child process.
  - It is called once but returns twice.
    - Return value of child: 0
    - Return value of parent: PID of child
  - Both child and parent continue executing with the instruction that follows the call to fork().
  - The child is a copy of parent.
    - Child gets a copy of the parent's data, heap, and stack.
    - Parent and child often share the text segment.

fork()

- Copy on write



  - After parent process write a page C.



fork()

- Example

```
#include "apue.h"

int     glob = 6;       /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int       var;      /* automatic variable on the stack */
    pid_t     pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */
```

fork()

- Example(cont.)

```
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {              /* child */
        glob++;                         /* modify variables */
        var++;
    } else {
        sleep(2);                       /* parent */
    }

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

fork()

```
#include "apue.h“
int     glob = 6;
char    buf[] = "a write to stdout\n";

int main(void)
{
    int       var;
    pid_t     pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ((pid = fork()) < 0) {           /* pid is child's pid(non-0). */
        err_sys("fork error");
    } else if (pid == 0) {              /* child */
        glob++;                         /* modify variables */
        var++;
    } else {
        sleep(2);                       /* parent */
    }

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Parent process

```
#include "apue.h“
int     glob = 6;
char    buf[] = "a write to stdout\n";

int main(void)
{
    int       var;
    pid_t     pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ((pid = fork()) < 0) {           /* pid is 0. */
        err_sys("fork error");
    } else if (pid == 0) {              /* child */
        glob++;                         /* modify variables */
        var++;
    } else {
        sleep(2);                       /* parent */
    }

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Child process

fork()

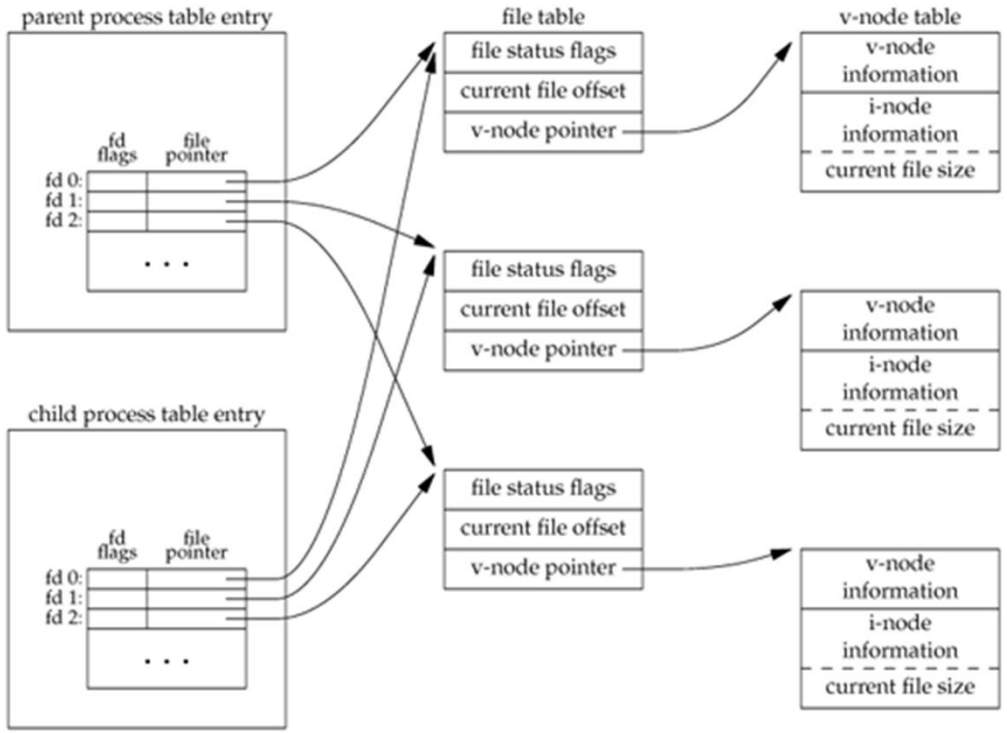
실행

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89
pid = 429, glob = 6, var = 88
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
$
```

We never know if the child starts executing before the parent or vice versa.
- It depends on the process scheduling algorithm.

fork()

- File sharing
  - Sharing of open files between parent and child after fork()



fork()

- Information shared by child and parent.
  - real-uid(gid), effective-uid(gid)
  - controlling terminal
  - current working directory, root directory
  - signal handlers
  - environment
  - resource limits
- differences between child and parent.
  - the return value from fork.
  - PID and PPID
  - child’s resource utilizations are set to 0.
  - pending signals

fork()

- Two main reasons for fork to fail
  - if there are already too many processes in the system.
  - if the total number of processes for this real user ID exceeds the system's limit.
- Two uses for fork
  - A process wants to duplicate itself.
    - Parent and child can each execute different sections of code at the same time.
    - Common for network servers.
  - A process wants to execute a different program.
    - Common for shells.
    - Child does exec() right after it returns from fork().

vfork()

```
#include <unistd.h>

pid_t vfork(void);
                    Returns: 0 in child, process ID of child in parent, -1 on error
```

- same calling sequence and same return values as fork().
- intended to create a new process when the purpose of the new process is to exec() a new program.
  - Does not copy the address space of parent into the child.
  - The child calls exec() or exit() right after the vfork().
  - The child runs in the address space of the parent.
  - Provides an efficiency.
- vfork() guarantees that the child runs first.

vfork()

- Example

```
#include "apue.h“
int     glob = 6;       /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify parent's variables */
        var++;
        exit(0);                /* child terminates */
    }
```

vfork()

Example(cont.)

```
    /*
     * Parent continues here.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

실 행

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
$
```

exit()

```
#include <stdlib.h>
void exit(int status);
void _Exit(int status);

#include <unistd.h>
void _exit(int status);
```

- causes normal program termination and the value of status is returned to the parent.
- _exit and _Exit
  - return to the kernel immediately.
- exit
  - performs cleanup processing and then returns to the kernel.
    - all open streams are flushed and closed.

Process termination

- Two types of process termination
  - normal termination
    - return from main
    - calling exit
    - calling _exit or _Exit
  - abnormal termination
    - calling abort
    - Receipt of a signal
- Regardless of how a process terminate, the same code is eventually executed.
  - Close open descriptors, release the memory, …

Process termination

- Termination status
  - Terminating process notify its parent how it terminated.
  - normal termination
    - pass an exit status as argument to exit() or _exit().
    - return value from main()
    - exit status is converted into a termination status.
  - abnormal termination
    - Kernel generates a termination status to indicate the reason for the abnormal termination.
  - The parent of the terminated process can obtain the termination status from **wait() or waitpid()**.

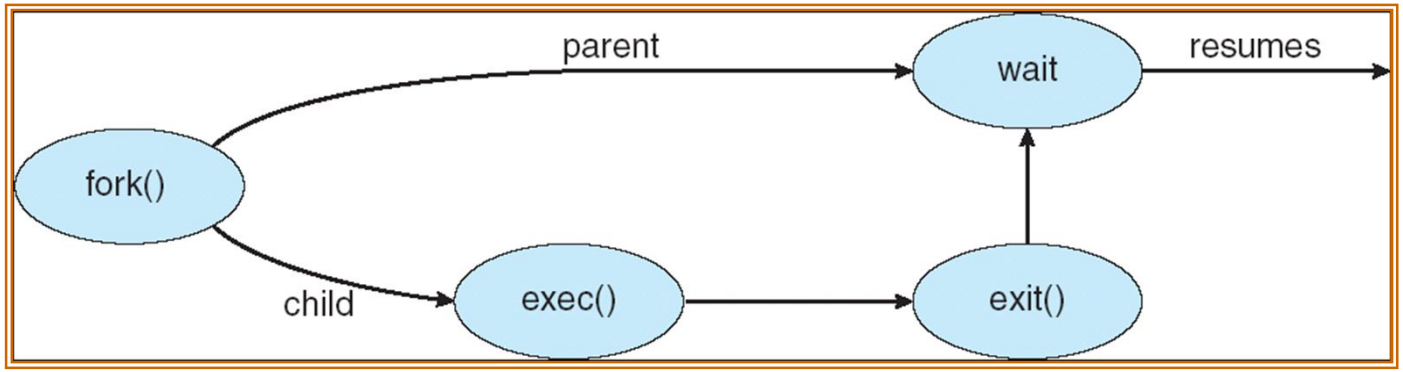
Process termination

- zombie state
  - Kernel has to keep some information
    - PID, termination status, and CPU usage time
    - This information is available when parent calls wait().
  - the process that has terminated, but whose parent has not yet waited for it, is called a zombie.

- If the parent terminates before the child?
  - init process becomes the parent of the orphaned process.
- If an orphaned process is terminated?
  - init calls wait() to fetch the termination status.

wait() and waitpid()

- A process that calls wait() or waitpid()
  - Block, if all of its children are still running.
  - Return immediately if a child has terminated.
  - Return immediately with an error, if it doesn't have any child processes.



wait() and waitpid()

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

pid_t wait(int *statloc);
                        Both return: process ID if OK, 0 (see later), or -1 on error
```

- wait()
  - If a child has already terminated and is a zombie, wait returns immediately with that child's status.
  - Otherwise, it blocks the caller until a child terminates.
  - If the caller blocks and has multiple children, wait returns when one terminates.
- statloc argument
  - Store termination status in the location pointed to by *staloc*.

wait() and waitpid()

- Macro to examine the termination status
  - WIFEXITED(status)
    - 정상적으로 종료한 경우에 참 값을 리턴
  - WEXITEDSTATUS(status)
    - exit 함수의 인자에서 하위 8 비트 값을 리턴
  - WIFSIGNALED(status)
    - 시그널에 의해 종료한 경우에 참 값을 리턴
  - WTERMSIG(status)
    - 시그널 번호를 리턴
  - WIFSTOPPED(status)
    - 실행이 일시 중단된 경우에 참 값을 리턴
  - WSTOPSIG(status)
    - 실행을 일시 중단시킨 시그널 번호를 리턴

wait() and waitpid()

- Example

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

void pr_exit(int status)
{
        if (WIFEXITED(status))
                printf("normal termination, exit status = %d\n",
                        WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
                printf("abnormal termination, signal number = %d\n",
                        WTERMSIG(status));
        else if (WIFSTOPPED(status))
                printf("child stopped, signal number = %d\n",
                        WSTOPSIG(status));
}
```

wait() and waitpid()

- Example(cont.)

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

int main(void)
{
        pid_t   pid;
        int     status;

        if ((pid = fork()) < 0)
                printf("fork error");
        else if (pid == 0)                      /* child */
                exit(7);

        if (wait(&status) != pid)               /* wait for child */
                printf("wait error");
        pr_exit(status);                        /* and print its status */
```

wait() and waitpid()

- Example(cont.)

```
        if ( (pid = fork()) < 0)
                printf("fork error");
        else if (pid == 0)              /* child */
                abort();                /* generates SIGABRT */

        if (wait(&status) != pid)       /* wait for child */
                printf("wait error");
        pr_exit(status);                /* and print its status */

        if ( (pid = fork()) < 0)
                printf("fork error");
        else if (pid == 0)              /* child */
                status /= 0;            /* divide by 0 generates SIGFPE */

        if (wait(&status) != pid)       /* wait for child */
                printf("wait error");
        pr_exit(status);                /* and print its status */
}
```

wait() and waitpid()

- 실행

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6
abnormal termination, signal number = 8
$
```

Each signal number is defined in <signal.h>
SIGABRT: 6
SIGFPE: 8

wait() and waitpid()

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

pid_t waitpid(pid_t pid, int *statloc, int options);
                        Both return: process ID if OK, 0 (see later), or -1 on error
```

- Why waitpid() is required?
  - If we have more than one child, wait() returns on termination of any of the children.
  - What if we want to wait for a specific process to terminate?
- waitpid()
  - Waits for a specific process with *pid* argument.
  - Provides some controls with *options* argument.

wait() and waitpid()

- pid argument
  - pid == -1
    - Waits for any child process.
    - In this case, waitpid() is equivalent to wait().
  - pid > 0
    - Waits for the child whose process ID equals pid.
  - pid == 0
    - Waits for any child whose process group ID equals that of the calling process.
  - pid < -1
    - Waits for any child whose process group ID equals the absolute value of pid.

wait() and waitpid()

- options argument
  - WNOHANG
    - Not block(return immediately) if a child specified by pid is not terminated.
  - WUNTRACED
    - Not block if a child specified by pid has stopped.

wait() and waitpid()

- Example

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

int main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {              /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);                    /* parent from second fork == first child */
        /* We're the second child; our parent becomes init. */
        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }
```

wait() and waitpid()

Example

```
    if (waitpid(pid, NULL, 0) != pid)  /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

실 행

```
$ ./a.out
$ second child, parent pid = 1
```

exec()

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0,  ...  /* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int execle(const char *pathname, const char *arg0, … /* (char *)0,
                char *const envp[] */ );
int execve(const char *pathname, char *const argv[], char *const envp []);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv []);
                                        All six return: -1 on error, no return on success
```

- Execute a program.
  - When a process calls one of the exec functions, the process is completely replaced by the new program.
  - New program starts executing at its main() function.
  - exec() merely replaces the current process(text, data) with a brand new program from disk.

exec()

- pathname(filename) argument
  - pathname(filename) of a file to be executed.
  - execl, execv, execle, execve take a pathname argument.
  - execlp, execvp take a filename argument.
    - If filename contains a '/', it is taken as a pathname.
    - Otherwise, the executable file is searched for in the directories specified by PATH.
      - E.g. PATH=/bin:/usr/bin:/usr/local/bin/:.

exec()

- Argument list
  - arg0, arg1, ..., argn in the execl, execlp, and execle.
  - The list of arguments is terminated by a NULL pointer.
    - execl("/bin/ls", "ls", "-al", 0);

- Argument vector
  - *argv[] in execv, execve and execvp.
  - The array of pointers is terminated by a NULL pointer.
    - char *argv[3] = {"ls", "-al", 0};
    - execv("/bin/ls", argv);

exec()

- Environment variable
  - *envp[] in execve and execle
  - A pointer to an array of pointers to the environment strings.
  - The other functions use the *environ* variable.
    - char *env[2] = {“TERM=vt100”, 0};
    - execle(“/bin/ls”, “ls”, 0, env);

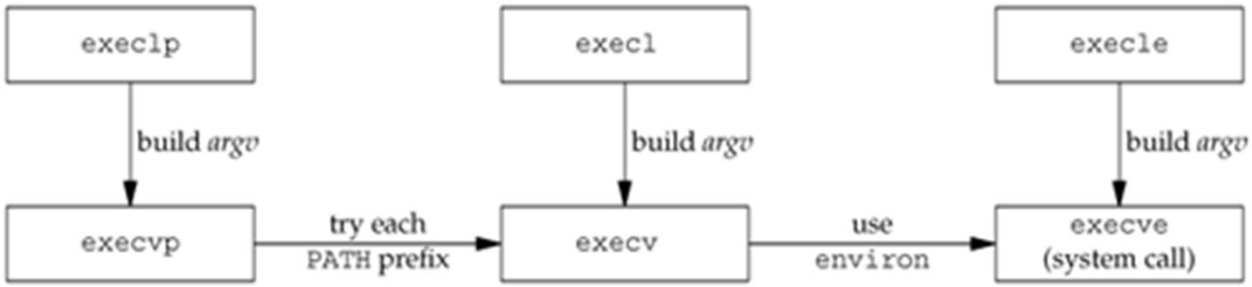
exec()

- Differences among the six exec functions

function	pathname	filename	arg list	argv[]	environ	envp[]
execl	0		0		0	
execlp		0	0		0	
execle	0		0			0
execv	0			0	0	
execvp		0		0	0	
execve	0			0		0
(letter in name)		p	l	v		e

exec()

- execve is a system call.
  - execl, execv, execle, execlp, execvp are library functions.
  - Relationship of the six exec functions.



exec()

- Example

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

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }
```

exec()

- Example(cont.)

```
    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

exec()

- echoall

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int        i;
    char       **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)              /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

exec()

- 실행

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
…
HOME=/home/sar
```