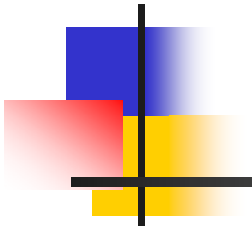
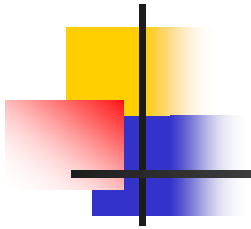
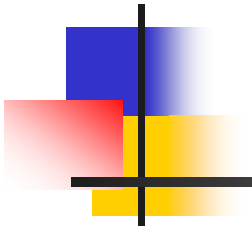


제7장 프로세스 제어

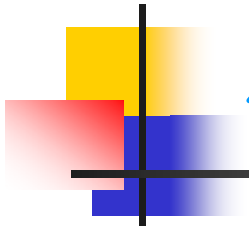


내용

- 프로세스 ID
- 프로세스 생성
- 프로세스 종료
- 레이스 컨디션
- 프로그램 실행
- 기타



프로세스 ID

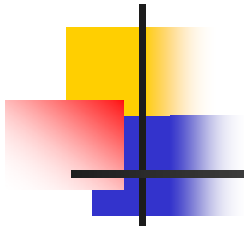


시스템 시작

- The First Process
  - the first process(*sched*) with pid =0 is created during boot time, and fork/exec twice.

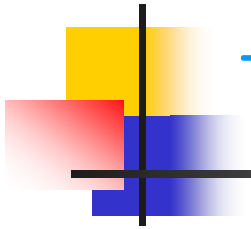
```
0       sched
1       init
2       pagedaemon
```

- Process ID 0 : swapper (scheduler process)
  - system process
  - part of the kernel
  - no program on disk corresponds to this process



시스템 시작

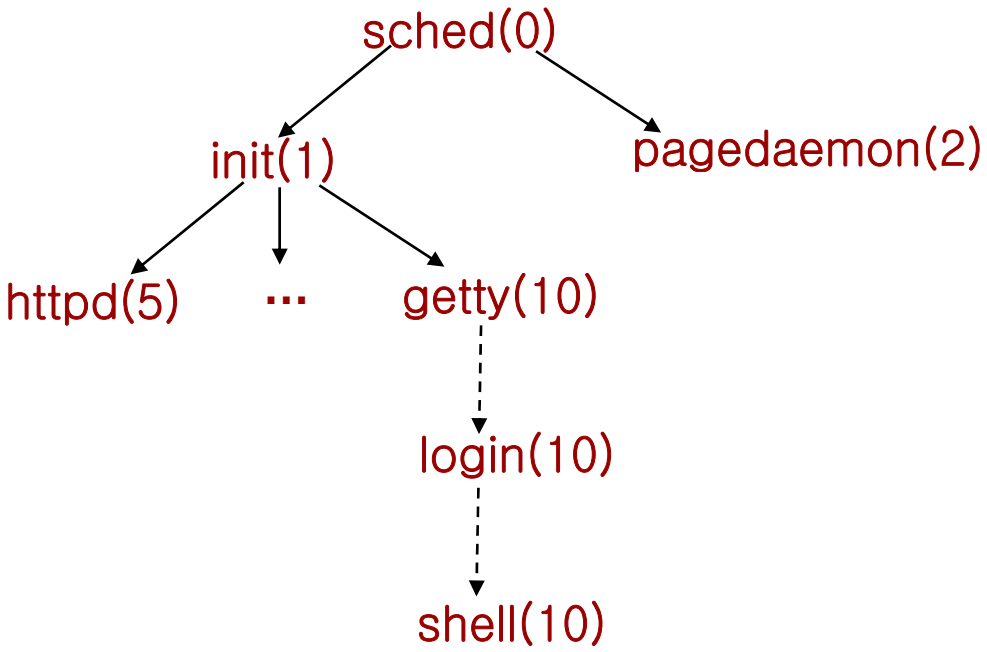
- Process ID 1: init process
  - invoked by the kernel (/etc/init or /sbin/init)
  - reads the system initialization files (/etc/rc*)
    - brings the system to a certain state (multi-user)
    - creates processes based upon script files
    - getty, login process, mounting file systems, start daemon processes
- Process ID 2: pagedaemon
  - supports the paging of the virtual memory system
  - kernel process



The process hierarchy

- getty process(/etc/getty)
  - detects line activity and prompts with login
  - exec /bin/login, after entering a response to the login prompt.
- login process
  - checks a user's login and password against */etc/passwd*
  - exec */bin/sh* or */bin/csh*
  - set the evnironment variables like HOME, LOGNAME, PATH...
- shell process
  - initially read commands form start-up files like */etc/.cshrc* and *$HOME/.cshrc*
  - shell starts its job

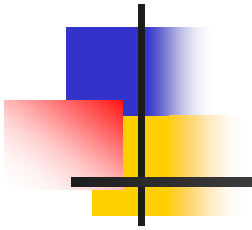
The process hierarchy



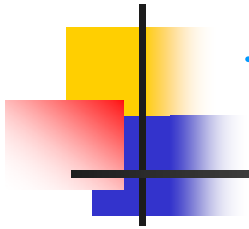
Process Identifiers

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

pid_t getpid(void);          // returns process ID
pid_t getppid(void);         // returns parent process ID
uid_t getuid(void);          // returns real user ID
uid_t geteuid(void);         // returns effective user ID
gid_t getgid(void);          // returns real group ID
gid_t getegid(void);         // returns effective group ID
```

- none of these functions has an error return



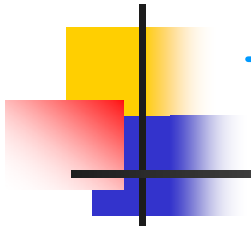
프로세스 생성



fork()

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

- fork() is the ONLY way to create a process
  - 체세포 자기복제
- Child process is the new process created by fork()
  - an almost-exact duplicate of the parent process
  - code, data, stack, open file descriptors, etc.

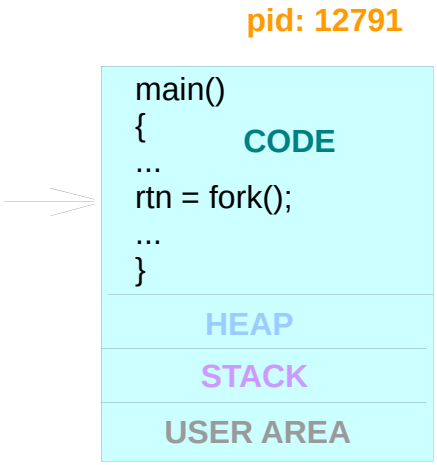


fork()

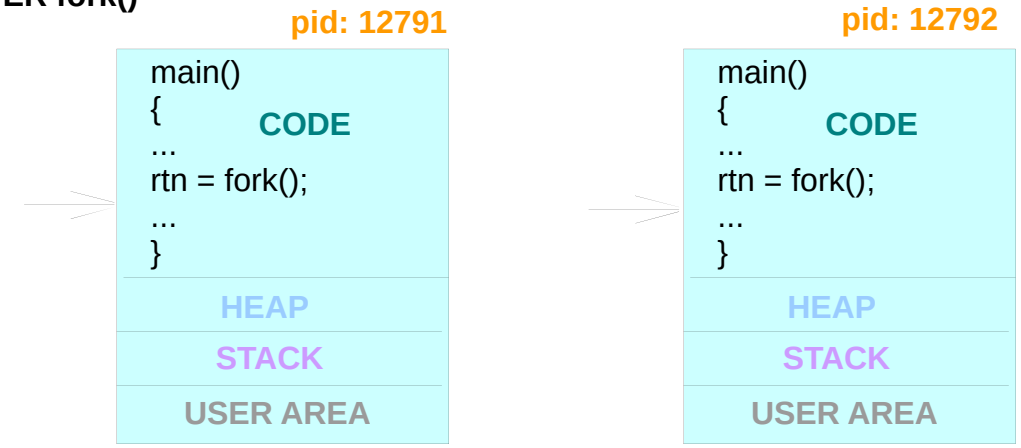
- fork() is called once, but returns twice!
  - returns 0 in child process
  - returns the child process ID in parent process
- Execution
  - Parent and child continue executing instructions following the fork() call
- Often, fork() is followed by exec()

System Calls : Creating a new process

BEFORE fork()

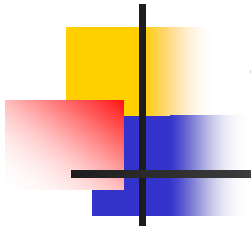


AFTER fork()



예제: fork1.c

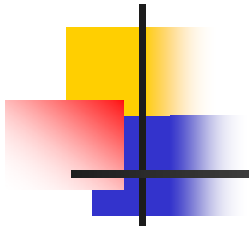
```
#include <sys/types.h>      /* fork1.c */
#include <unistd.h>

int  glob = 6;                      /* global 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)
    perror("write error");

  printf("before fork\n");
  if ( (pid = fork()) < 0)
    perror("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);
}
```



결과: fork1.c

```
$ 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
```

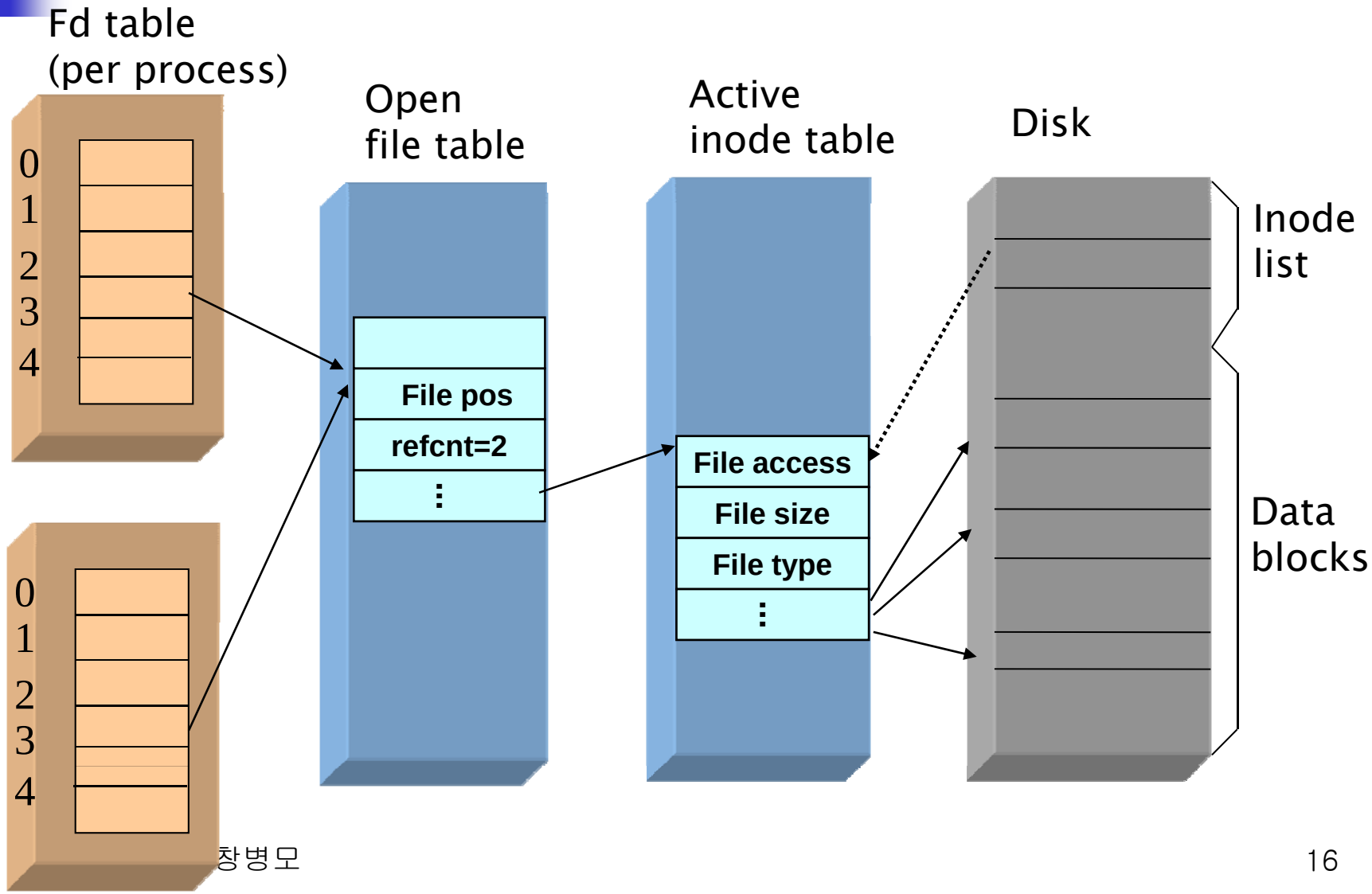
- "before fork" printed twice.
- because it is fully buffered (redirected) in the second execution,
- the standard I/O buffer is copied to the child process and is not yet flushed

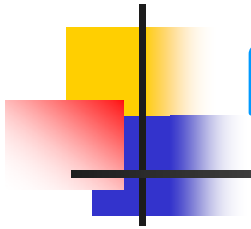


File sharing after fork()

- File sharing mechanism
  - Fd array is in user-area.
  - Child copies the user-area of its parent.
- File sharing effects
  - Parent and child share the same file descriptors
  - Parent and child share the same file offset
  - Intermixed output from parent and child

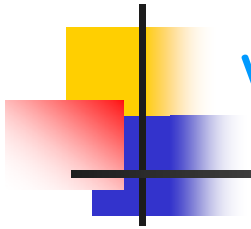
fork()


Fd table
(per process)
0
1
2
3
4
Open
file table
File pos
refcnt=2
Active
inode table
File access
File size
File type
Disk
Inode
list
Data
blocks
0
1
2
3
4



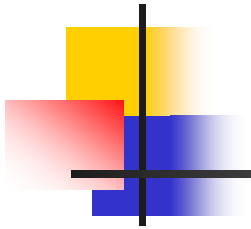
Properties not inherited to the child

- the return value from fork()
- the process IDs are different
- file locks
- pending alarms are cleared for the child
- the set of pending signals for the child is set to the empty set



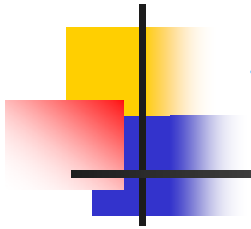
vfork()

- Creates a new process only to exec a new program
  - No copy of parent's address space for child (not needed!)
  - Before exec, child runs in "address space of parent"
  - Efficient in paged virtual memory
- Child runs first
  - Parent waits until child exec or exit
  - Then the parent resume
  - The deadlock possibility if the child wait for something from the parent



예제: vfork1.c

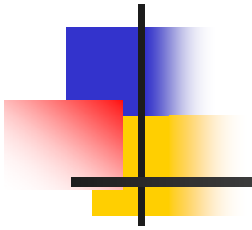
```
#include  <sys/types.h>  /* vfork1.c */

int    glob = 6;              /* global 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)
    perror("vfork error");
  else if (pid == 0) {        /* child */
    glob++;                   /* modify parent's variables */
    var++;
    _exit(0);                 /* child terminates */
  }
  /* parent */
  printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
  exit(0);
}
```



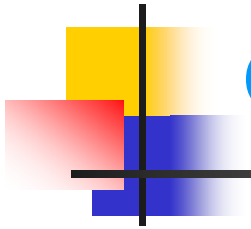
결과:vfork1.c

- increments by child appear in parent address space

- $ a.out
  before vfork
  pid = 607, glob = 7, var = 89



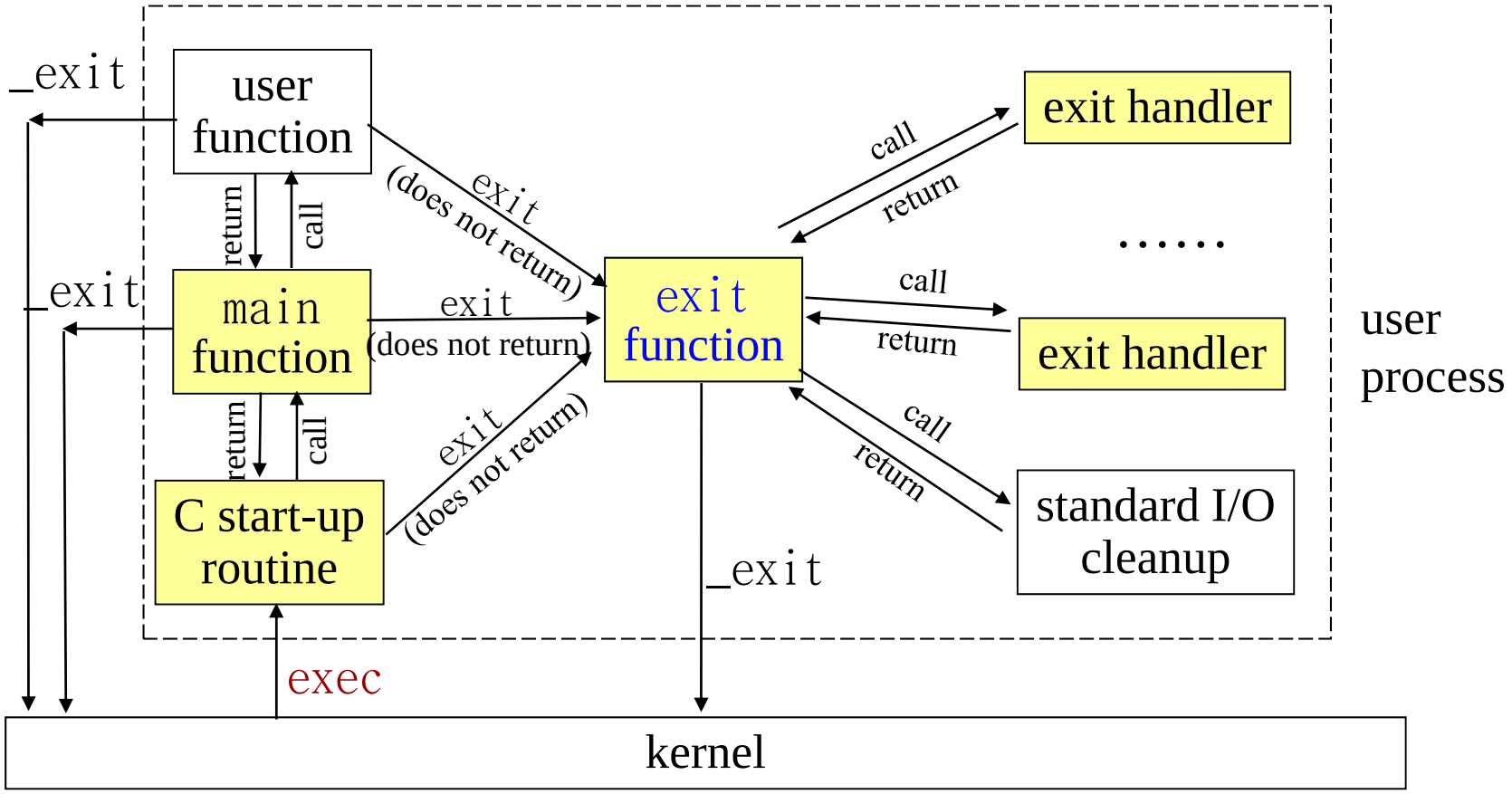
프로세스 종료

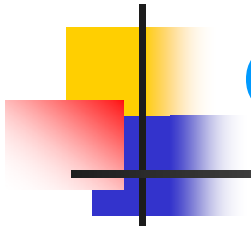


exit() – Process Termination

- A process can terminate in 5 ways:
- Normal Termination
  - return from main()
  - exit()
  - _exit()
- Abnormal Termination
  - calling abort()
    generates SIGABRT signal
  - process receives signals

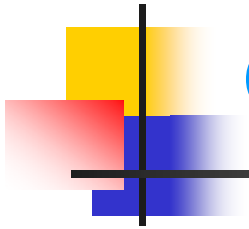
C Program Start/Termination





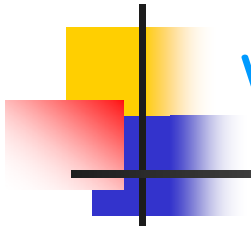
exit() – Process Termination

- Cleanup processing 을 수행하고 정상적으로 종료한다.
  - 출력 버퍼의 내용을 디스크에 쓴다(fflush)
  - 모든 열려진 스트림을 닫고(fclose)
  - 메모리 등 자원을 회수한다.
- Exit status:
  - argument of exit( ), _exit( )
  - the return value from main( ): exit(main( ))
- Termination status:
  - Normal termination:
    - Exit status → Termination status
  - Abnormal termination:
    - kernel indicates reason → Termination status



exit() – Termination status

- Child terminates
  - Kernel sends SIGCHLD signal to parent
- Parent wait() or waitpid()
  - parent can obtain the termination status of the child
  - wait() fetch the termination status of child when receiving SIGCHLD signal
- What if parent terminates before child?
  - init becomes the parent of the child process.

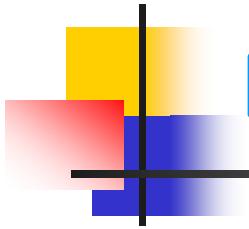


wait() waitpid()

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

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

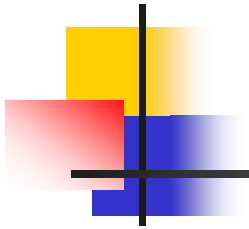
- A process that calls wait or waitpid can
  - block (if all of its children are still running), or
  - return immediately with the termination status of a child, or
  - return immediately with an error
    (if it doesn't have any child processes)

- *statloc*
  - a pointer to an integer to store the termination status



Macros to examine termination status

Macro	Description
WIFEXITED(status)	True if child is terminated normally WEXITSATUS(status) : get exit status (low-order 8 bits)
WIFSIGNALED(status)	True if child is terminated abnormally (by receipt of a signal) WTERMSIG(status) : fetch the signal number that caused the termination. WCOREDUMP(status) : true if a core file was generated
WIFSTOPPED(status)	True if child is currently stopped WSTOPSIG(status) : fetch the signal number that caused the stop

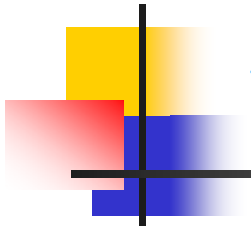


예제: prexit.c

```
#include <sys/types.h>  /* prexit.c */
#include <sys/wait.h>
#include "ourhdr.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%s\n",
        WTERMSIG(status),
#ifdef  WCOREDUMP
        WCOREDUMP(status) ? " (core file generated)" : "");
#else
        "");
#endif
  else if (WIFSTOPPED(status))
    printf("child stopped, signal number = %d\n",
        WSTOPSIG(status));
}
```

예제: wait1.c

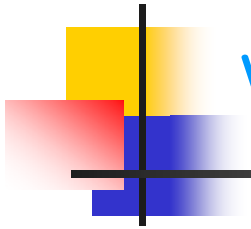
```
#include  <sys/types.h>  /* wait1.c */
#include  <sys/wait.h>
#include  "ourhdr.h"
int main(void) {
  pid_t  pid;  int    status;
  if ( (pid = fork()) < 0) perror("fork error");
  else if (pid == 0)
            exit(7);                              /* child */

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

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

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

  if ( (pid = fork()) < 0) perror("fork error");
  else if (pid == 0)                                /* child */
            status /= 0;                            /* divide by 0 generates SIGFPE */
  if (wait(&status) != pid) perror("wait error");   /* wait for child */
  pr_exit(status);                                  /* and print its status */
  exit(0); 
}
```



결과: wait1.c

- $ a.out
  normal termination, exit status = 7
  abnormal termination, signal number = 6 (core file generated)
  abnormal termination, signal number = 8 (core file generated)

- signal number = 6 ← SIGABRT
- signal number = 8 ← SIGFPE

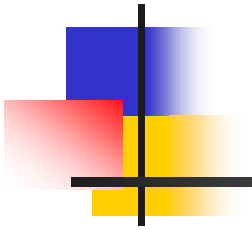


waitpid()

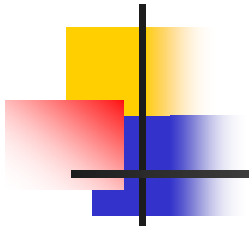
- pid_t waitpid(pid_t *pid*, int **statloc*, int *options*);
  - waits for one particular process
  - provides a nonblocking version of wait

- the *pid* argument
  - -1 : waits for any child process
  - > 0 : waits for the child process whose process ID equals *pid*.
  - 0 : waits for any child whose process GID equals that of the calling process.
  - < -1 : waits for any child whose process GID equals the absolute value of *pid*.

- the *options* constants for `waitpid`
  - `WNOHANG` : does not block if a child specified by *pid* is not immediately available, return 0

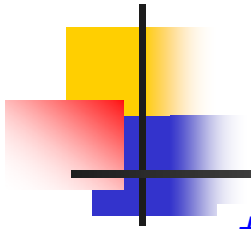


레이스 컨디션



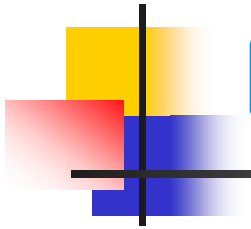
Race Conditions

- Multiple processes share some data
- Outcome depends on the order of their execution (i.e. RACE)
- After fork(), we cannot predict if the parent or the child runs first!
- The order of execution depends on:
  - system load
  - kernel's scheduling algorithm



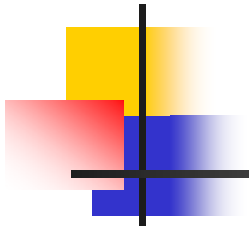
예 제

```
#include  <sys/types.h>  /* tellwait1.c */
#include  <stdio.h>

static void charatatime(char *);

int main(void) {
  pid_t  pid;

  if ( (pid = fork()) < 0) perror("fork error");
  else if (pid == 0)  charatatime("output from child\n");
  else                charatatime("output from parent\n");
  exit(0);
}

static void charatatime(char *str) {
  char  *ptr;
  int    c;

  setbuf(stdout, NULL);      /* set unbuffered */
  for (ptr = str; c = *ptr++; )
    putc(c, stdout);
}
```



Race Conditions

- Race condition problems are hard to detect because they work "most of the time"!
- For parent to wait for child
  - call `wait`, `waitpid`, `wait3`, `wait4`
- For child to wait for parent
  - while (getppid() != 1)
    sleep(1);
  - polling! wastes CPU time
  - use signals or other IPC methods



Race Conditions

- After fork
  - parent and child both need to do something on its own
  - e.g. parent: write a record in a log file
  - e.g. child: creates a log file
- Parent and child need to:
  - TELL each other when its initial set of operations are done, and
  - WAIT for each other to complete

TELL and WAIT

```
#include “ourhdr.h”
TELL_WAIT(); /* setup for TELL_XXX and WAIT_XXX */
if ( (pid = fork()) < 0 )
    err_sys(“fork error”);
else if (pid==0) {    /* child */
    /* child does whatever is necessary */
    TELL_PARENT(getppid()); /* tell parent we're done */
    WAIT_PARENT(); /* & wait for parent */
    /* and child continues on its way */
    exit(0);
}
```

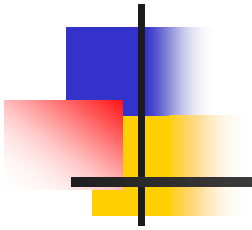
child

```
/* parent does whatever is necessary */
TELL_CHILD(pid); /*tell child we're done */
WAIT_CHILD(); /* wait for child */
/* and parent continues on its way */
exit(0);
```

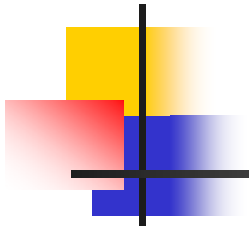
parent

Avoid Race Condition

```
#include  <sys/types.h>  /* tellwait2.c */
#include  "ourhdr.h"

static void charatatime(char *);

int main(void) {
  pid_t  pid;

  TELL_WAIT();

  if ( (pid = fork()) < 0)
    err_sys("fork error");
  else if (pid == 0) {
    WAIT_PARENT();     /* parent goes first */
    charatatime("output from child\n");
  } else {
    charatatime("output from parent\n");
    TELL_CHILD(pid);
  }
  exit(0);
}
```



프로그램 실행

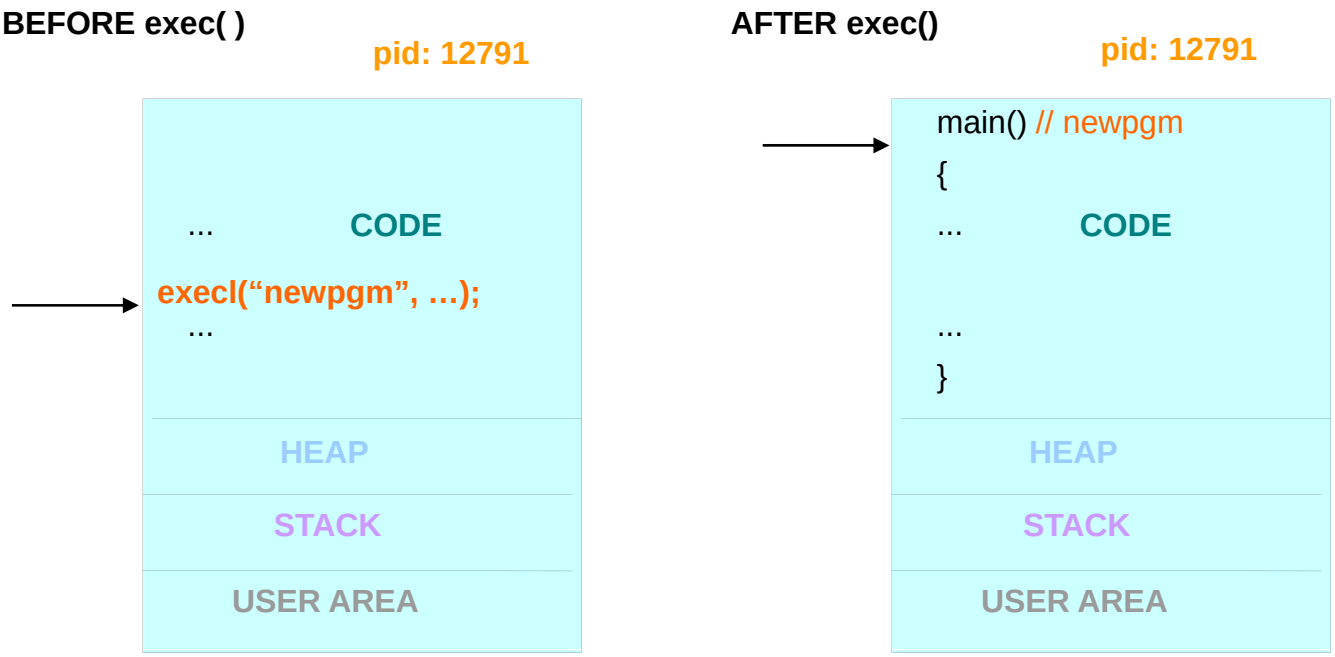


Program Execution: exec()

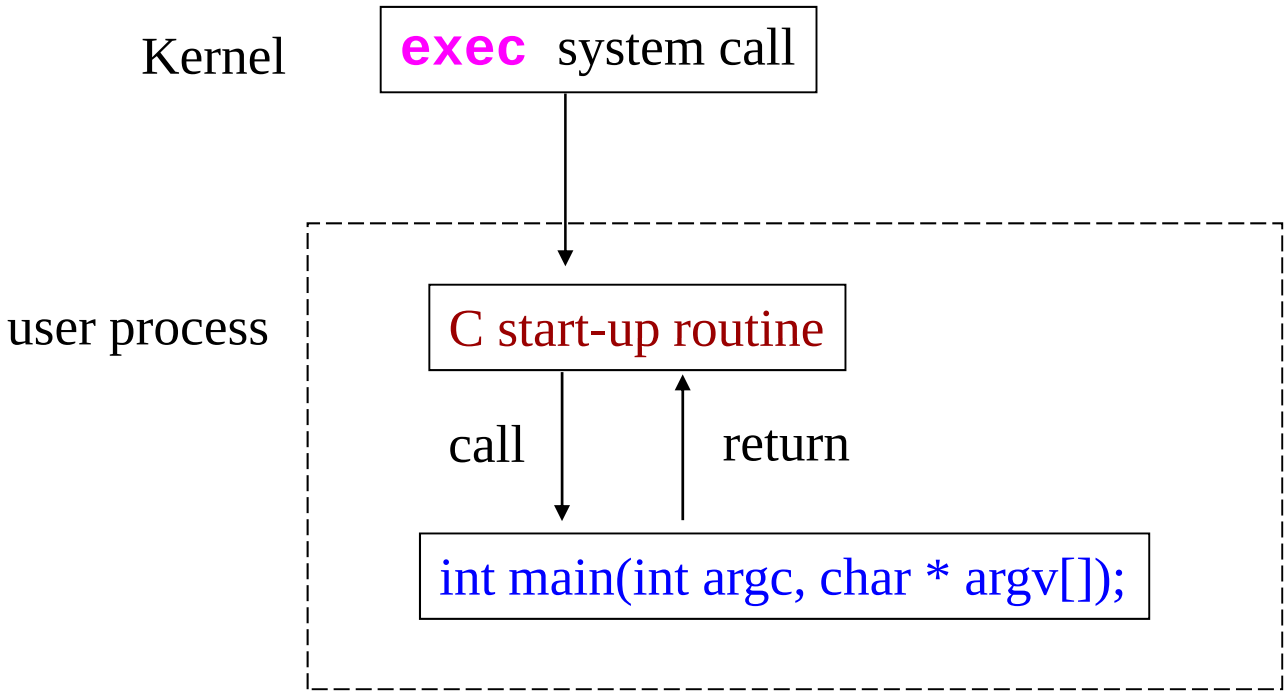
- 프로세스 내의 프로그램을 새 프로그램으로 대치
  - exec() 시스템 호출 사용
  - 보통 fork() 후에 exec( )

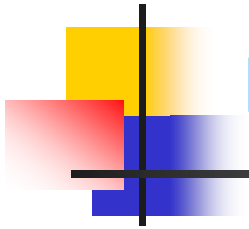
- When a process calls an exec function
  - that process is completely replaced by the new program (text, data, heap, and stack segments)
  - and the new program starts at its main function
  - Successful exec never returns !

exec() 시스템 호출


BEFORE exec( )
pid: 12791
... CODE
execl("newpgm", ...);
...
HEAP
STACK
USER AREA
AFTER exec()
pid: 12791
main() // newpgm
{
... CODE
...
}
HEAP
STACK
USER AREA

Process Start

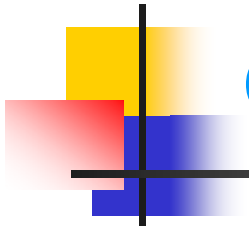




main()

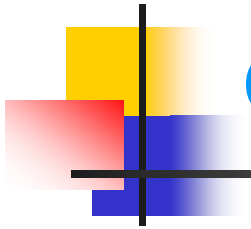
- int main(int argc, char *argv[]);
  - argc : the number of command-line arguments
  - argv[] : an array of pointers to the arguments
- C start-up routine
  - Started by the kernel (by the exec system call)
  - Take the command-line arguments and the environment from the kernel
  - Call main

    exit(main(argc, argv));



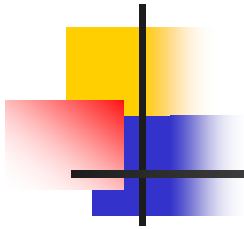
exec functions

```
#include <unistd.h>

int execl(char *pathname, char *arg0, ... /* (char*) 0 */);

int execv(char *pathname, char *argv[]);

int execle(char *pathname, char *arg0, ... /*(char*) 0 */, char *envp[]);

int execve(char *pathname, char *argv[], char *envp[]);

int execlp(char *filename, char *arg0, ... /* (char*) 0 */);

int execvp(char *filename, char *argv[]);

    All six return: -1 on error, no return on success
```



exec functions

- exec? (p, l, v, e)
  - p: filename
  - l : takes a list of arguments
    the last argument should be a null pointer
  - v: takes argv[] vector
  - e: takes envp[] array
    without 'e', the environment variables of the calling process are copied

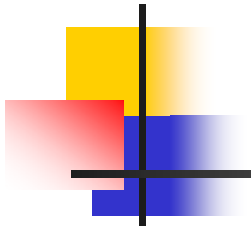


Properties inherited to the new program

- Same ID
  - process ID, parent process ID, real user ID, real group ID, supplementary group IDs, process group ID, session ID
- Controlling terminal
- time left until alarm clock
- current working directory
- root directory
- file mode creation mask
- file locks
- process signal mask
- pending signals
- resource limits
- tms_utime, tms_stime, tms_cutime, tms_ustime values

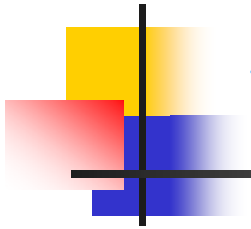
예제: exec1.c

```
#include  <sys/types.h>  /* exec1.c */
#include  <sys/wait.h>
char  *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main(void) {
  pid_t  pid;

  if ( (pid = fork()) < 0)        perror("fork error");
  else if (pid == 0) {            /* specify pathname, specify environment */
    if (execle("./echoall", "echoall", "myarg1", "MY ARG2", (char *) 0, env_init) < 0)
      perror("execle error");
  }

  if (waitpid(pid, NULL, 0) < 0)          perror("wait error");
  if ( (pid = fork()) < 0)
    perror("fork error");
  else if (pid == 0) {            /* specify filename, inherit environment */
    if (execlp("echoall", "echoall", "only 1 arg", (char *) 0) < 0)
      perror("execlp error");
  }
  exit(0);
}
```



예제: echoall.c

```
#include "ourhdr.h"   /* echoall.c */

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);
}
```



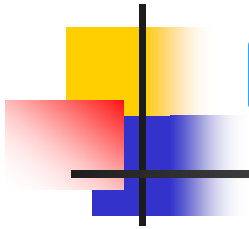
결과: exec1.c

```
$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=chang
HOME=/user/faculty/chang
...
EDITOR=/usr/ucb/vi
```

Example: Foreground Processing

- Execute a program using fork( ) and exec( )

```
#include <stdio.h>
main(argc, argv)
int argc, argv[ ];
{    int child, pid;

     pid = fork( );
     if (pid == 0) {
        execvp(argv[1], &argv[1]);
        fprintf(stderr, "Could not execute %s\n",argv[1]);
     } else {
        child = wait(&status);
        printf("A child with PID %d terminated with exit code %d\n",
                        pid, status>>8);

     }
}
```



Redirection 구현

%redirect file command

1) The parent forks and then waits for the child to terminate

2) The child opens the file "output“, and

3) Duplicates the fd of “output" to the standard output(fd=1), and close the fd of the original file

4) The child execute the “command”.

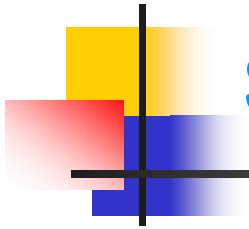
All of the standard output of the command goes to "output“

5) When the child shell terminates, the parent resumes

- %redirect out ls -l

Example: Redirection

```
#include <stdio.h>
main(argc, argv)
int argc, argv[ ];
{
        int fd, status;
        if (fork() == 0) {
                fd=open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0600)
                dup2(fd, 1);
                close(fd);
                execvp(argv[2], &argv[2]);
                perror("main");
        }
        else {
                wait(&status);
                printf("Child is done\n");
        }
}
```



system()

```
#include <stdlib.h>

int system(const char *cmdstring);
```

- Causes the string to be given to sh(1) as input as if the string had been typed as a command at a terminal
  - ex) system("date > file");
- System is implemented by calling fork, exec, and waitpid
- Return values:
  - -1 with errno: fork or waitpid fails
  - 127 : exec fails
  - Termination status of shell: all 3 succeed

예제: system.c

```
#include  <sys/types.h>  /* system.c */
#include  <sys/wait.h>
#include  <errno.h>
#include  <unistd.h>
int system(const char *cmdstring)  /* version without signal handling */
{
  pid_t pid;  int status;

  if (cmdstring == NULL)
     return(1);    /* always a command processor with Unix */

  if ( (pid = fork()) < 0) {
     status = -1;  /* probably out of processes */
  } else if (pid == 0) {                          /* child */
     execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
     _exit(127);    /* execl error */
  } else {                                        /* parent */
     while (waitpid(pid, &status, 0) < 0)
        if (errno != EINTR) {
              status = -1; /* error other than EINTR from waitpid() */
              break;
        }
  }
  return(status);
}
```