🔶

chapter03. process in UNIX

objectives

1. Learn how to create processes
2. Experiment with fork and exec
3. Explore the implications of process inheritance
4. Use wait for process cleanup
5. Understand the UNIX process model

Process

Process

정의
- is the basic active entity in most operating-system models
- is a program in execution (실행 중인 program)

1. communicate with the OS using system calls
2. can be interrupted at any time by a device interrupt or a system call
3. is represented by a data structure called a process control block (PCB) or a process descriptor.

- CPU is shared among the processes ready to run
  - 한 번에 한 process 실행 가능

- **context switch** : CPU가 process들을 빠르게 스위치
  - saves the state of the current running process
  - and then loads the state of the next process to be executed.
  - needs to save enough information about the current running process so that it can be resumed later as if nothing had happened.

    → this determines important components of the PCB.
  - is initiated by an interrupt
    - software interrupt(system call)
    - device interrupt
    - timer interrupt(quantum expired)

  - steps → assume : user process is interrupted.
    1. CPU senses interrupt
    2. CPU starts the interrupt handler in privileged mode
    3. (Interrupt handler may disable all interrupts)
    4. Interrupt handler stores state of interrupted process
    5. Interrupt handler executes (kernel) code for the interrupt
    6. Interrupt handler ends by calling the CPU scheduler
    7. CPU scheduler loads the state of the process it selects
    8. (Interrupt handler would re-enable interrupts)
    9. CPU switches back to user mode and executes selected process

process identifiation → PID

- a process has a process ID and a parent process ID
  - pid_t getpid(void) : get a PID
  - pid_t getppid(void) : get a parent PID
  - pid_t : is an unsigned integer type

- a parent process 끝나면 system process에 의해 child process 실행

- **user and group ID**
  - system administrators assign a unique integral user ID
  - and an integral group ID to each user when creating the user's account.
  - **UNIX process** : has real/effective user(file access 권한에 따라 나뉨) and group IDs that convey pribileges to the process.
    - user에 따라서 얻어오는 command가 다름.

    - [ex] user 종류가 달라야 하는 경우

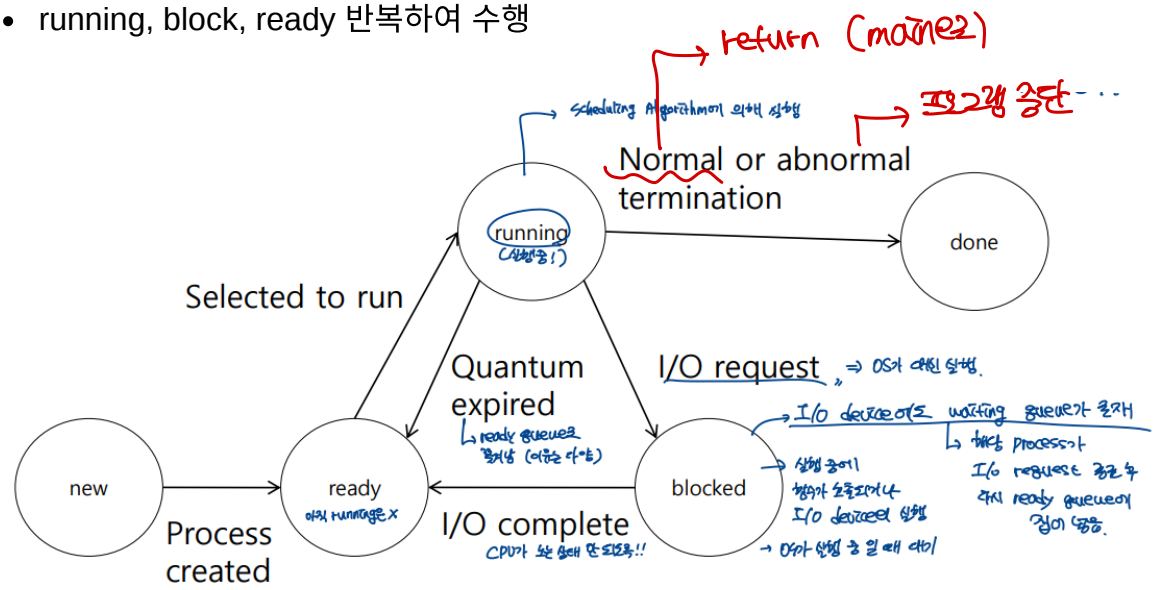
      → UNIX pw 명령어 : shadow file update → 권한에 따라 가능해야 함

      → root만 변경 가능(즉, effective user만 변경이 가능)

    - **method**
      - gid_t getegid(void), uid_t geteuid(void) ⇒ effective
      - gid_t getgid(void), uid_t getuid(void) ⇒ user

process state

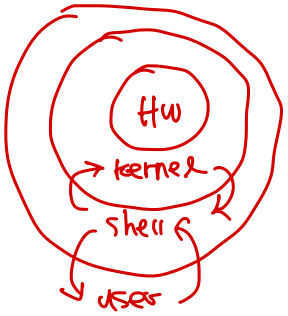
- running, block, ready 반복하여 수행



- **context switch**
- **process context** : the information that
  1. the OS needs about the process
  2. its environment to restart it after a context switch

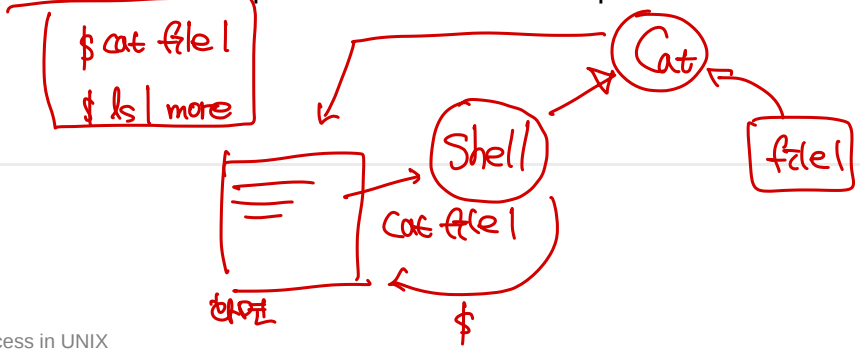
- **ps utility** : displays information about processes

header	option	meaning
F	-l	Flags (octal and additive) associated with the process
S	-l	Process state
UID	-f, -l	User ID of the process owner
PID	(all)	Process ID
PPID	-f, -l	Parent process ID
C	-f, -l	Processor utilization used for scheduling
PRI	-l	Process priority
NI	-l	Nice value
ADDR	-l	Process memory address
SZ	-l	Size in blocks of the process image
WCHAN	-l	Event on which the process is waiting
TTY	(all)	Controlling terminal
TIME	(all)	Cumulative execution time
CMD	(all)	Command name (arguments with –f option)



Process Hierachy

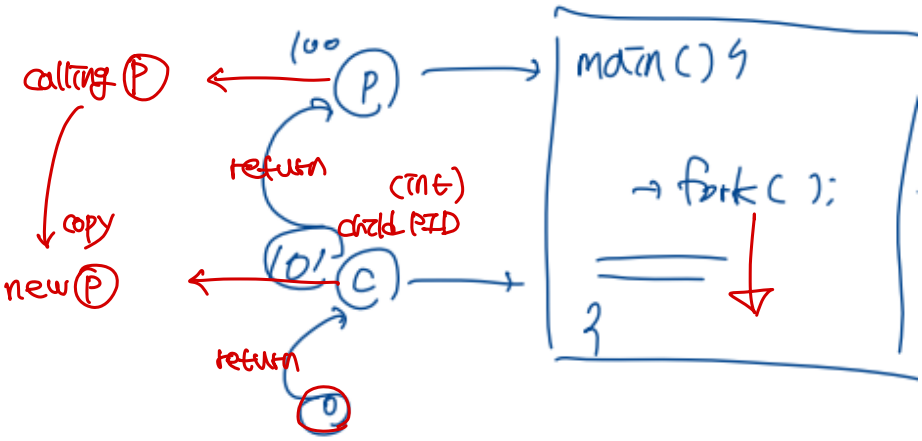
- parent and child : parent creates child
- root process : ancestor of all processes
  - an execution of program init on boot
- **shell example**
  - shell : creates a new process in response
  - shell exampe : shell creates a new process when it receives a command



System Function for process

fork : clone the calling process

- child PID → parent process에게 전달 (process creation)
- Parent ← calling, Child ← new process(a copy of parent)

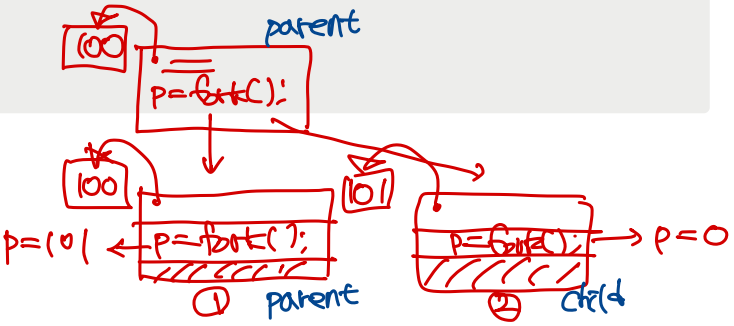


- return 값 받은 순간부터 fork 아래의 라인부터 실행
  - 즉, fork 실행 순간부터

    두 개의 process가 같은 코드를 concurrent하게 실행

    + 실행 후 : fork() return 값에 따라 갈림
- Return
  - 0 to child
  - child process id to parent
  - negative value(-1) to parent on error
- process creation
  - spawn a new process which is a copy(clone) of the calling process

```
#include <sys/types.h> //return -1 if error
#include <unistd.h> //return 0 to child
pid_t fork(void); //return child's pid to parent

//자식 process의 자식 pid 반환은 0
```



- Characteristics of fork() operation
  - a new process by a making a copy of the parent's image in memory.
  - creates an exact copy of the calling process at the same point in the execution
  - **child inherits**
    - parent attributes (environment and privileges).
    - some of the parent's resources (open files and devices).
  - **child not inherits**
    - only PID and PPID are different → child : new PID
    - CPU usage : child는 생성 시기가 다르기 때문에 cpu time reset 필요
    - *Locks and alarm*
      - Lock : 공유 resources에 대한 권한 → app service의 일종
        - 부모 process가 가지고 있던 lock 정보는 물려주지 않음
      - alarm : 함수가 존재 → timer 설정 가능
        - 부모가 가지고 있던 alarm은 child에게 물려주지 않음
    - *pending signals*
      - pending signals : 어떤 event에 대한 sw적인 통제 수단(ch.1)
      - 부모가 pending signals 갖고 있었다고 하더라도 child는 pending signal로 시작하지 않음
    - different return values after fork()
      - returns PID of child to parent
      - returns 0 to child

- **Parent Attributes and Resources**
  - Not every attributes and resources is inherited by child
  - child process competes for processor time with other processes as a seperate entity.
    - 독립적인 객체 잊지 말 것!
  - user running on a crowded time-sharing system can obtain a greater share of the CPU time by creating more processes

- pros and cons
  - pros
    - child inherits all data from parents on fork
    - child does not need to do additional communication to get info from parent.
  - con
    - they have to stick to same code file → 제한적, IPC 도움이 필요함.
      - ↳ signals, FIFOs...

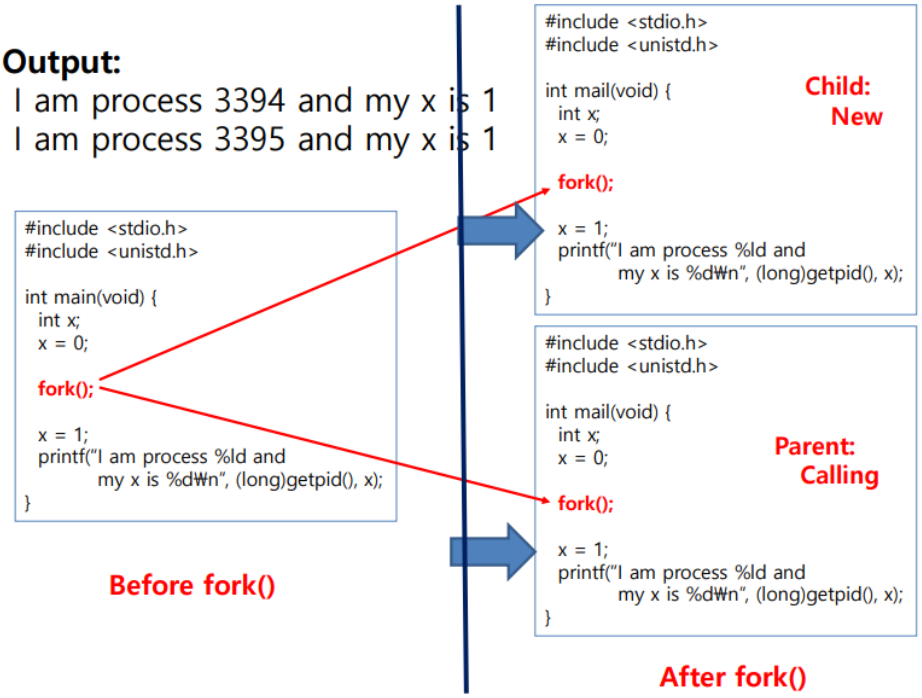
- **example**
  - ex1 - simplefork.c

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

int main(void){
  int x;
  x = 0; //부모 입장에서는 1, child도 1임!!

  fork(); //child process 생성

  x = 1;
  printf("I am process %ld and my x is %d\n", (long)getpid(), x);

  //(long)getpid() : return 값

  //두 개의 process에 의해서 두 번의 출력
```

↳ return value

```
  //parent executed first
}
```

- output : parent id → child id



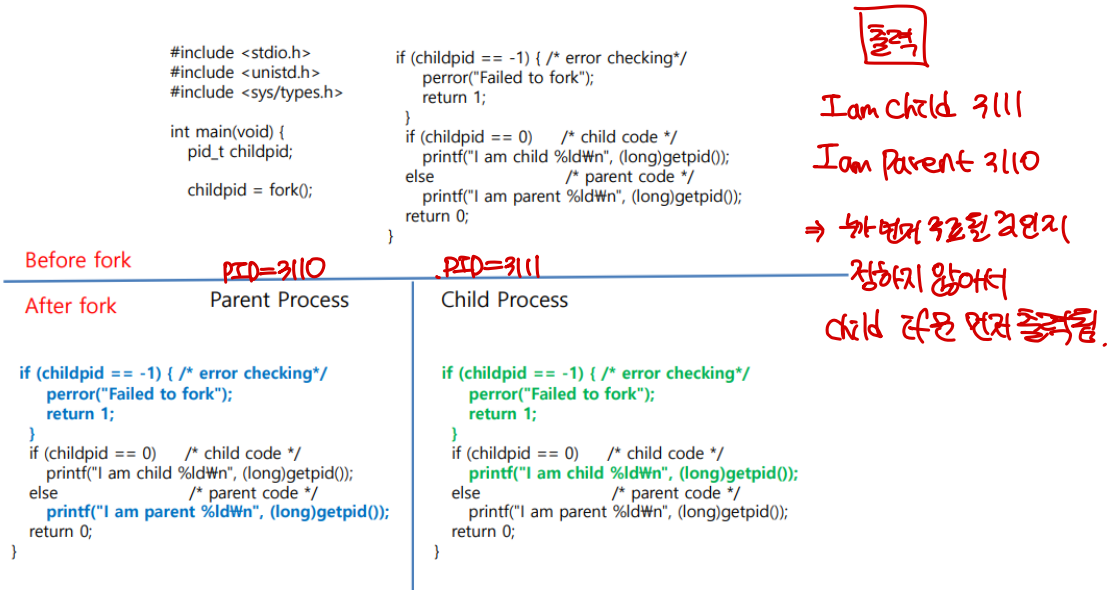
- run different code도 가능
- can execute different instructions by using return value of fork func.
  - if문으로 parent child의 작업 분리 가능 → ex2
- note : return zero to child, but child's PID to parent.

- **ex2 - twoprocs.c**

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

int main(void){
  pid_t childpid;

  childpid = fork();
  if(childpid == -1) {  //error checking
    perror("Failed to fork");
    return 1;
  }
```

```
  if(childpid == 0) //child code
    printf("I am child %ld\n", (long)getpid());
  else     //parent code
    printf("I am parent %ld\n", (long)getpid());

  return 0;
}
```

- output : child id → parent id



ex3 - badprocessID.c

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

int main(void){
  pid_t childpid;
  pid_t mypid;

  mypid = getpid(); //저장해놓은 시점이 parent process
  childpid = fork();
  if(childpid == -1){
    perror("Failed to fork");
    return 1;
  }
  if(childpid == 0)
    //child code -> parent pid 출력
    printf("I am child %ld, ID = %ld\n", (long)getpid(), (long)mypid);
  else
    //parent code -> parent pid 출력
    printf("I am parent %ld, ID = %ld\n", (long)getpid(), (long)mypid):
```

parent parent

child parent

```
  return 0;
}
```

- output ⇒ 중요 2번 정확히 알아둬! child 먼저 출력

  - : i am child (child id) , ID = (parent id)

    i am parent (parent id), ID = (parent id)

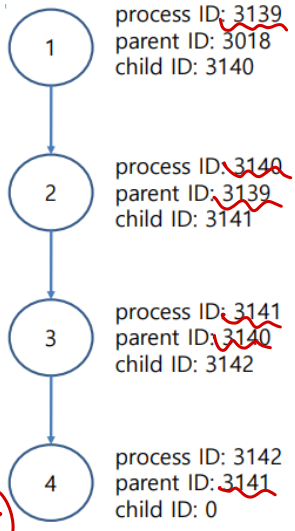
  - getpid → returns the PID of the calling process

- **ex4 - simplechain.c**

  - chain of processes ⇒ 원하는 만큼 process 생성

    - 부모가 자식을 fork한 이후에는 fork x → 반복적인 rule 생성

      → 출력 구문으로 확인할 것



$ simplechain (simplechain , 4)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
                  ↑argument 개수
int main(int argc, char *argv[]) { //argument 필요한 예제
  pid_t childpid = 0;
  int i, n;

  if(argc != 2){
    fprintf(stderr, "Usage: %s processes\n", argv[0]);
    //argv[0] : 첫번째 argument의 str 값
    return 1;
  }
  n = atoi(argv[1]);
```

↳첫번째 argument (simplechain)

```
//converts the initial portion of the str pointed to by nptr to int
for(i=1;i<n;i++) //n-1번 수행하면서 n개의 process 생성
  if(childpid=fork()) //0이 아니면 빠져나가도록 !
    break;
fprintf(stderr,"i:%d process ID:%ld parent ID:%ld child ID:%ld\n"
        , i, (long)getpid(), (long)getppid(), (long)childpid);
```

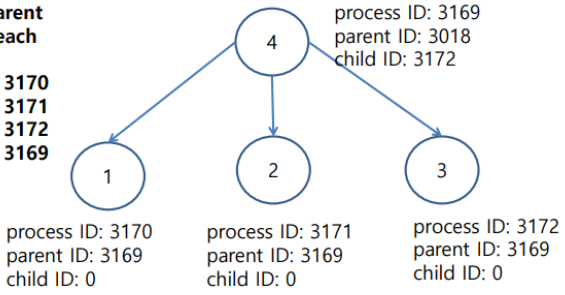
fork()의 return value = 0 → 연달아 생성

- 자식 process가 실행될 때 부모가 종료되면 ppid 출력함
- process 종료 조건에 대해서 지정 없기에 내가 원하는 출력 보장 x

↓ 에러 잡아주기

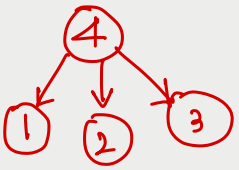
- **ex5 - Fan of processes**

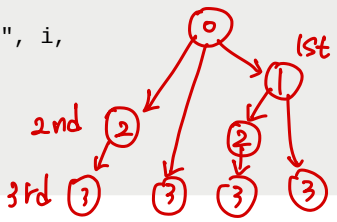
What will be the parent ID and child ID of each process? If
process 1 ID = 3170
process 2 ID = 3171
process 3 ID = 3172
process 4 ID = 3169




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

int main(int argc, char *argv[]){
  pid_t childpid = 0;
  int i, n;

  //error handling
  if(argc != 2){
    fprintf(stderr, "Usage: %s processes\n", argv[0]);
    return 1;
  }

  n = atoi(argv[1]);
  for(i=1;i<n;i++)
    if((childpid = fork()) <= 0)
      break;

  fprintf(stderr, "i:%d process ID:%ld child ID:%ld\n", i,
  (long)getpid(), (long)getppid(), (long)childpid):
  return 0;
}
```

부모가 계속 생성





- if문 조건을 (childpid = fork()) == -1 이면 부모, 자식 모두 for loop

$2^0\ 2^1\ 2^2 \cdots$

- each iteration double n번째 iteration → 2^{n-1}개 child

  - sleep function(일시 정지)
    - debugging 용도로 사용되는 function
    - block each process for any seconds before exit.

ex) sleep(30) ⇒ statement immediately before the return.

---

wait family : wait for child process to terminate.

- wait function
  - wait func causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal.
    - status inform을 parent는 받을 수 없었음. But, wait() 가능!!

    ↳ child 상태변경 → 즉 child가 종료
  - parent가 child 끝나면 시작(기존에는 concurrent)

    → 하위 작업 지정할 때 보통 사용

  ★ sleep과 다른 기능을 수행!!!

  - if wait가 없다면 loop가 필요 ⇒ cpu 낭비 ★

- wait function allows parent to
  1. wait for child process to terminate
  2. receive status info from child
  3. value which the exit function returns

```
#include <sys/wait.h>
//headerfile include 필수!

//#1
//return PID of exited child to caller
pid_t wait(int *stat_loc);
                ↳ output parameter
//can also return exit status of child(in stat_loc)
//including whether child was terminated by a signal, and by which signal
```

process id type

```
//return -1 and set errno if error is occured.

//#2 - more general function of #1
//wait for specified child, or for children from specified process group
pid_t waitpid(pid_t pid, int *stat_loc, int options);
//         child process   output parmeter   부가적인 option

//child process pid : 0보다 큰 값이 general, 0 or -1은 child의 범위보다 큼
//child process pid == 0 : parent가 같은 그룹 내에서 child가 하나라도 종료됨
//option : makes it non-blocking (ex. WNOHANG) -> 바로 return(오류 상관 x)
//         <-> blocking : 함수가 return 반환할 때까지 기다림
// if options == WNOHANG
//  -> returns 0 to report that there are possible unwaited-for children
//      but that their status is not available
//return type : pid_t -> 종료한 child id

//can also reprot stopped children
//return -1 and set errno if error is occured.
```

- stat_loc
  - argument of wait or waitpid is a pointer to an integer variable.
  - integer pointer를 통해 child가 넘겨주는 return(상태값) 확인 가능
    - → int형 변수를 parsing하여 가져와야 함 ⇒ macro 사용
  - not NULL = these functions store the return status of the child in this location.
  - child : returns its status by calling exit, _exit, _Exit, or return from main.
  - parent : can only access the 8 least significant bits of the status.

  - POSIX
    - WIFEXITED(int stat_val) : nonzero value when the child terminates normally

      → 종료되었는지 확인
    - WEXITSTATUS(int stat_val) : Returns the low-order 8 bits, if WIFEXITED is nonzero

      → WIFEXITED 종료 확인 후 return 값 확인
    - WIFSIGNALED(int stat_val) : Nonzero value when the child terminates because of an uncaught signal

      → signal로 종료되었는지 확인

- WTERMSIG(int stat_val) : Returns the number of the signal that caused the termination, if WIFSIGNALED is nonzero

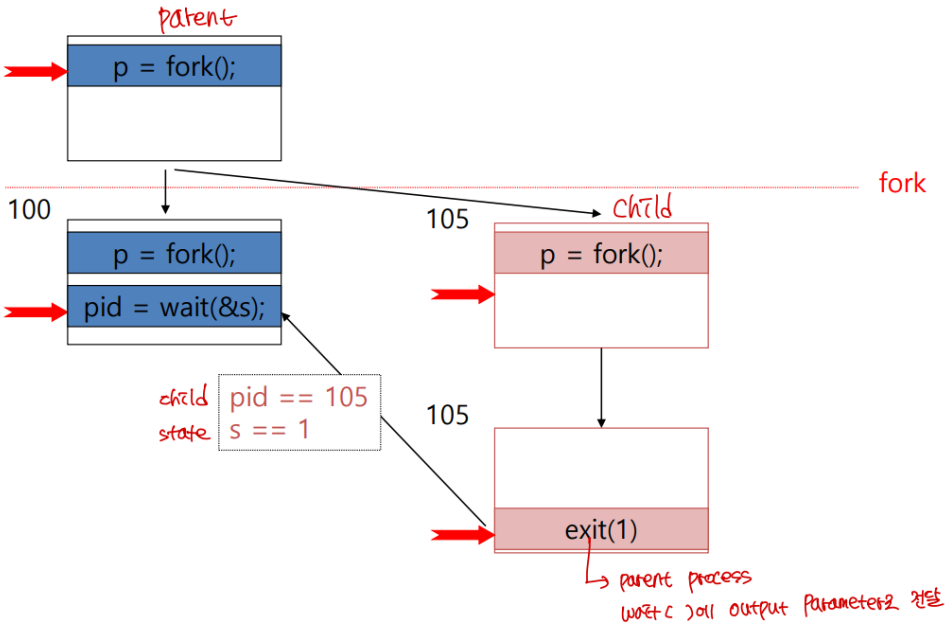
  → WIFSIGNALED로 종료 확인 후 몇 번 signal인지 확인.

- WIFSTOPPED(int stat_val) : Nonzero if a child is currently stopped

  → child가 stop되었는지 확인

- WSTOPSIG(int stat_val) : Returns the number of the signal that caused the child to stop, if WIFSTOPPED is nonzero

  → WIFESTOPPED 확인 후 어떤 signal로 종료되었는지 확인



- waiting for all children
  - while(r_wait(NULL) > 0);
    - wait가 모든 child process가 종료될 때까지 실행
    - 만약 child가 없으면 -1 return ⇒ while문 X
  - r_wait() : interrupt를 고려한 개선된 ver
    - restarts the wait function if it is interrupted by a signal
    - r_ : restart

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

```c
//wait header file

pid_t r_wait(int *stat_loc){
  int retval;
  //error handling
  while (((retval = wait(stat_loc)) == -1) && (errno == EINTR));
  return retval;
}
```

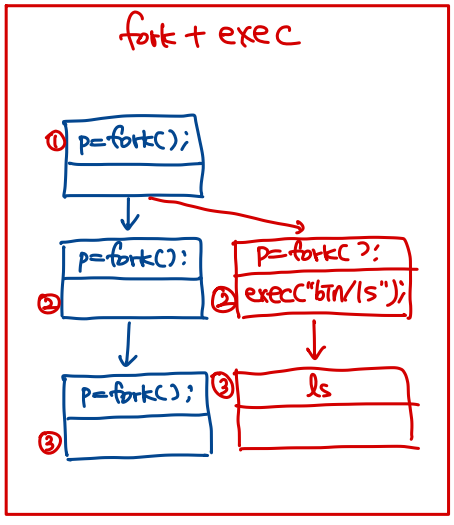
wait func 사용

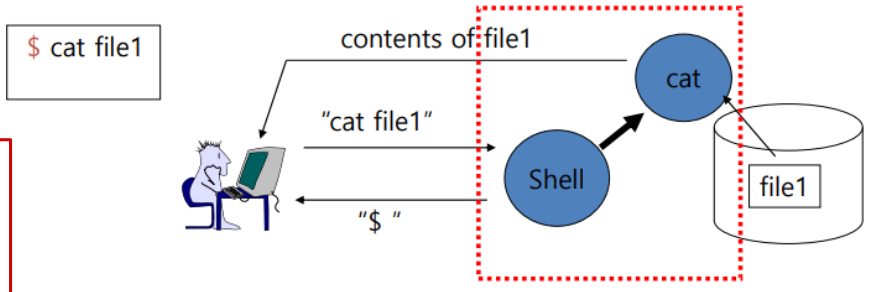
interrupt 의해 error

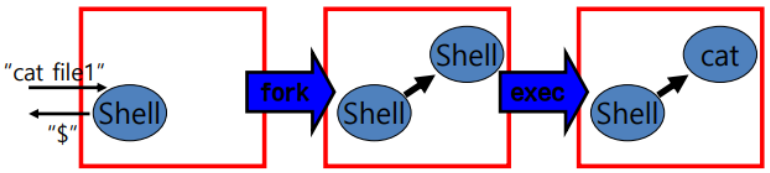
exec
- execl
  - execlp
  - execle
- execv
  - execvp
  - execve ⇒ 결론적으로는 여기

exec family ⇒ fork()와 연동하여 사용 ⇒ Parent와 동일한 코드 수행하는 단점을 보완

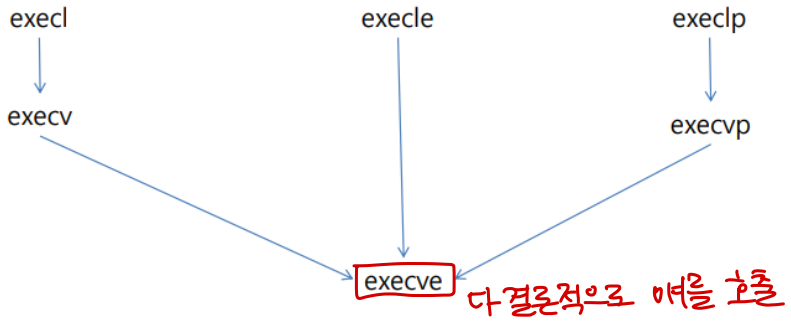
- make calling process run a diffrent program → replace의 개념!! not create
- Run Different code
  - fork() : creates a copy of the calling process
  - exec() : load a new code
    - fork와 다르게 다른 코드를 실행하고자 할 때 사용)
    - replace current code by new process(not create a new process) ★ ★ 생성이 아님!!







①
- **exec family**
  - all **exec** function family will do same work → argument만 다름!!
  - eventually all **exec** functions will cal **exec**ve.



  - will not return to parent process like fork function
  - calling process가 새로운 process로 replaced! (copy가 아닌 new load)

- charateristics of exec
  - overwrittten인 모든 내용들을 낭비 줄일 수 있음
  - environment만 상속받음(process 실행 내용 말고!!)

1)
- **execl family**

l → argument 하나씩씩.
v → 배열

```
#include <unistd.h>
//headerfile

//#1
//실행할 argument들을 일일히 하나씩 지정, 마지막 값은 NULL로 끝을 지정함.
int execl(const char *path, //경로
          const char *arg0, ... , const char *argn, (char*) 0);
int execlp(const char *file, //file명
          const char *arg0, ... , const char *argn, (char*) 0);
int execle(const char *path,
          const char *arg0, ... , const char *argn, (char*) 0,
          char *const envp[]);
```

↳ end of argument

a)
- execl
  - is useful when a known file with known arguments is being called.
  - ① path: points to the name of file holding a command that is to be executed

- fully qualified pathname(절대경로 → root부터 시작)

  or relative to the current directory(상대경로 → 현재 dir부터)

  - arg0 : points to a string that is the same as path

    (or at least its last component)
  - arg1 … argn : pointers to arguments for the command
  - 0(NULL) : the end of the (variable) list of arguments.

b) ◦ execlp

  - file : 실행 파일의 이름
    - if contains a slash, execlp treats file as a pathname and behaves like execl. ⇒ 경로 작성도 가능
    - if file does not have a slash, execlp uses the PATH environment variable to search for the executable. → 환경변수 사용

c) ◦ execle

  - execl parameter와 동일
  - char *const envp[] ex) char *envC] = { "USER=user1", "PATH=/usr/bin: …", (char *)0 };
    - representing the environment of the new process
    - 실행될 때 추가적으로 설정 가능

- example

⇒ 실행결과 : ls -l

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
  pid_t childpid;

  childpid = fork();
  if(childpid == -1){
    perror("Failed to fork");
    return 1;
  }

  if(childpid == 0){
```

fork 실패

Child Code

```
    execl("/bin/ls", "ls", "-l", NULL);
    //ls -l : 실행결과 -> 다른 실행 파일을 실행
    perror("Child failed to exec ls");
    return 1;
  }

  //error handling part
  if(childpid != wait(NULL)) { //child process id 값 return
    perror("Parent failed to wait due to siganl or error");
    return 1;
  }
  return 0;
}
```

- **execv family**

```
//#2
//실행할 argument들을 배열로 선언, 배열의 마지막 index는 NULL
int execv(const char *path, const char *argv[]);
int execvp(const char *file, const char *argv[]);
int execv(const char *path, const char *argv[], char *const envp[]);
```

  - execv
    - use an execv func with an argument array constructed at run time
      - file이나 argument들을 사전에 모를 때 사용하기 편리함
  - execvp, execve 는 execlp, execle와 유사함.
  - example of execv

```
#include <unistd.h>

main(){
  char *const av[] = {"ls", "-l", (char*)0}; //argument arr
  execv("/bin/ls", av);
  perror("execl failed to run ls");
  exit(1);
}
```

- example execcmd.c

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

int main(int argc, char *argv[]) {
  pid_t childpid;

  /* check for valid number of command-line arguments */
  if (argc < 2){
    fprintf (stderr, "Usage: %s command arg1 arg2 ...\n", argv[0]);
    return 1;
  }

  childpid = fork();

  if (childpid == -1) {
    perror("Failed to fork");
    return 1;
  }

  /* child code */
  if (childpid == 0) {
    //argv = { "execcmd" , "ls" , "-l" , NULL }
    execvp(argv[1], &argv[1]); //실행파일명, &array 주소값
    perror("Child failed to execvp the command");
    return 1;
  }

  /* parent code */
  if (childpid != r_wait(NULL)) {
    perror("Parent failed to wait"); //wait으로 interrupt 향상
    return 1;
  }

  return 0;
}

//Question : argv = { "execcmd" , "ls" , "-l" , "*.c", NULL }
//how big is argument array passed as the second argument to execvp?
//Answer:
//.c files의 개수에 따라 다름
//the shell expands *.c before passing the command line to execcmd.
```

exit : terminate calling process

- terminate execution at any point

  ex) 치명적인 오류를 발견했을 때 지속할 필요가 없을 시에 사용

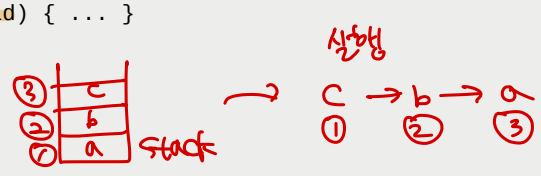
- exit status → wait에게 전달 ⇒ wait
  - return status


```
#include <stdlib.h>

void exit(int status); //return status info : 종료 전에 짧은 유언을 return
```

```
#define OPEN_ERROR 1
int fd;
if((fd = open("test.dat", O_RDONLY)) < 0){
  perror("Open Error"); //system error message
  exit(OPEN_ERROR);
}
```

- **atexit**
  - clean-up on exit → register clean-up functions.
  - register functions : automatically called when process terminates.
  - up to 32 functions can be registered. → 여러 개의 함수를 등록할 수 있음
    - return, parameter 둘 다 void type만!!

```
void a(void) { ... }
void b(void) { ... }
void c(void) { ... }

atexit(a);
atexit(b);
atexit(c);

if(found_error())
  exit(1);
```



```
//stack : |c()|b()|a()|

//output : c() -> b() -> a()
```

```
main() {
  atexit(clean_up); //register function : clean_up
  //...
  fd = open("temp.dat", O_WRONLY);
  //...
  if(found_error())
    exit(1); //parameter 1이 clean_up parameter로 전달
}

void clean_up(void) { //void type
  remove("temp.dat");
}
```

Background Processes and Daemons

- Background Processes : service라는 이름으로 사용자와 I/O communication 없이 알아서 실행 ⇒ keyboard 입력 X
  - ↔ foreground processes : terminal에서 user input interaction으로 실행
- Daemons : 상시적으로 돌고 있는 Process

1) foreground 강제 종료

- ~~Interrupt character~~
  - shell
    - ① prompts for commands
    - ② reads the commands from stdinput,
    - ③ forks children to excute the commands
    - ④ waits for the children to finish.

⇒ standard I/O 가 terminal type의 device로부터 들어오면 user는 interrupt character가 들어옴으로써 실행 중인 command를 terminate 할 수 있음 !!

(EX) Ctrl-C : foreground 강제 종료
background는 아님! ⇒ ls-l 종료 불가

2) Background 명령 종료

- & : 대부분의 shell들이 실행되어야 할 background process들의 ending command로 해석

  +) fg : back → fore로 전환

- shell이 background process를 생성할 때

  ⇒ shell은 process가 끝날 때까지 기다리지 않음.

Daemon

: a background process that normally runs indefinitely.

- UNIX
  - UNIX OS : 많은 daemon을 routine task들을 실행하는 process들로 이용.
- example "runback.c"

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "restart.h"

int makeargv(const char *s, const char *delimiters, char ***argvp);

int main(int argc, char *argv[]) {
   pid_t childpid;
   char delim[] = " \t";
   char **myargv;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s string\n", argv[0]);
      return 1;
   }
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0) {                /* child becomes a background process */
     if (setsid() == -1)
        perror("Child failed to become a session leader");
     else if (makeargv(argv[1], delim, &myargv) == -1)
        fprintf(stderr, "Child failed to construct argument array\n");
     else {
        execvp(myargv[0], &myargv[0]);
        perror("Child failed to exec command");
     }
     return 1;                                /* child should never return */
   }
   return 0;                                                /* parent exits */
}
```

→ background process

→ keyboard 입력 X

⇒ runback "ls -l" (shell에서 "ls -l &" 실행과 유사)