네트워크 프로그래밍

13. 멀티프로세스 기반의 서버구현

프로세스 & 좀비 프로세스

프로세스 제어 블록

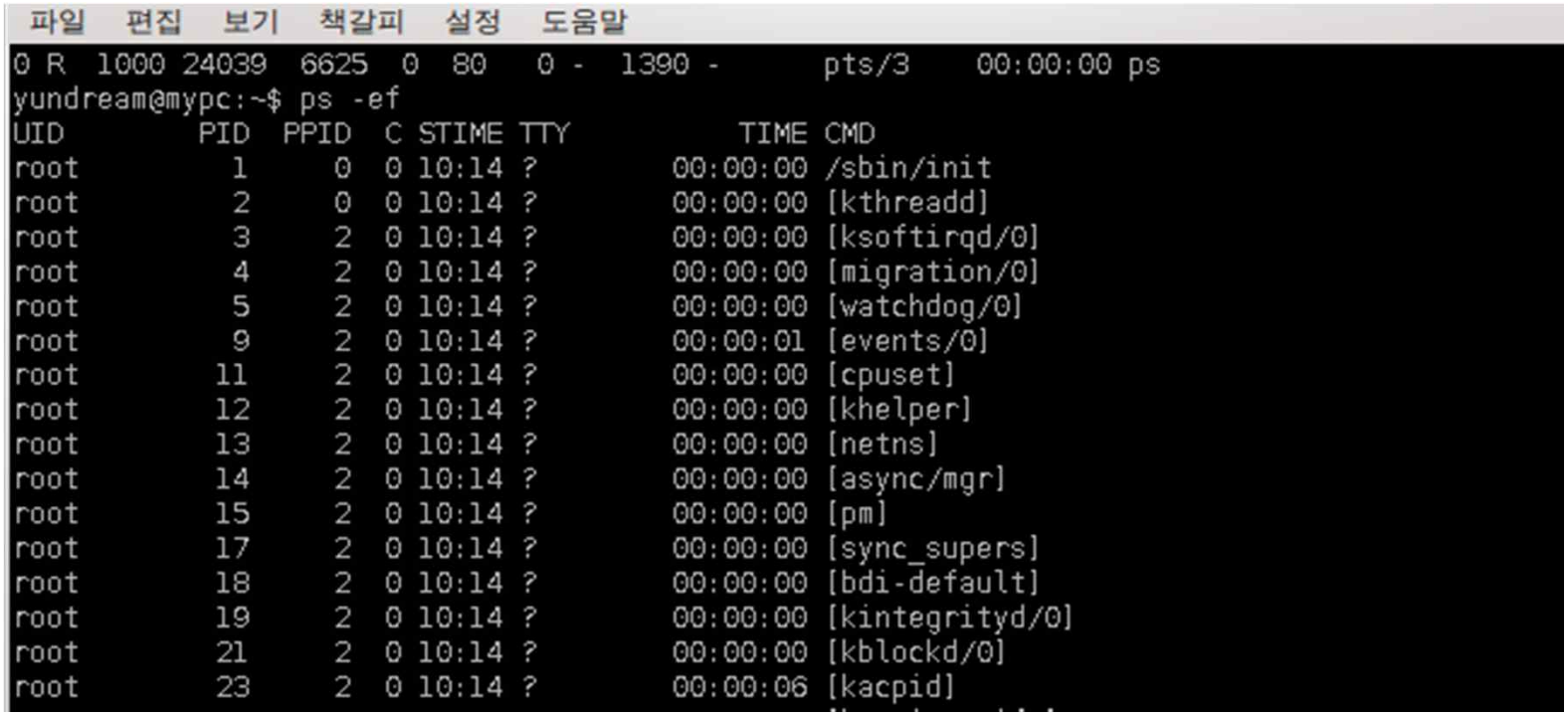
- 프로세스 제어 블록
  - 프로세스는 프로그램과 프로세스 제어 블록으로 구성
  - PCB는 프로세스에 대한 정보를 운영 체제에 제공하는 자료 구조로 주기억장치에 저장
  - 프로세스가 생성될 때 만들어짐
  - 프로세스가 실행을 완료하면 PCB도 삭제

cpu에 레지스터 값 뿐만아니라
프로세스가 만지고 있던 상황들
(어떤 파일을 열고 있었고,
부모는 누구였고,
자식은 누구였고,
pc는 어디까지 있었고…)
등등 그 모든 맥락을 저장해둠
〉〉pcb

Process ID
Pointer to parent process
Pointer area to child processes …
Process state
Program counter
Register save area …
Memory pointers
Priority information
Accounting information
Pointers to shared memory areas, shared processes and libraries, files, and other I/O resources

프로세스 구분

- 프로세스 ID
  - 운영제제는 생성되는 모든 프로세스에 ID를 할당한다.



과제 개발 할때
서버에서 accept할때 fork를 같이 해준다
이후 분기에서
부모의 경우 연결소켓을 close
자식의 경우 리스닝소켓을 close
하면 된다!

ex)
if(pid == 0)
close(리스닝소켓fd)
else
close(연결소켓fd)

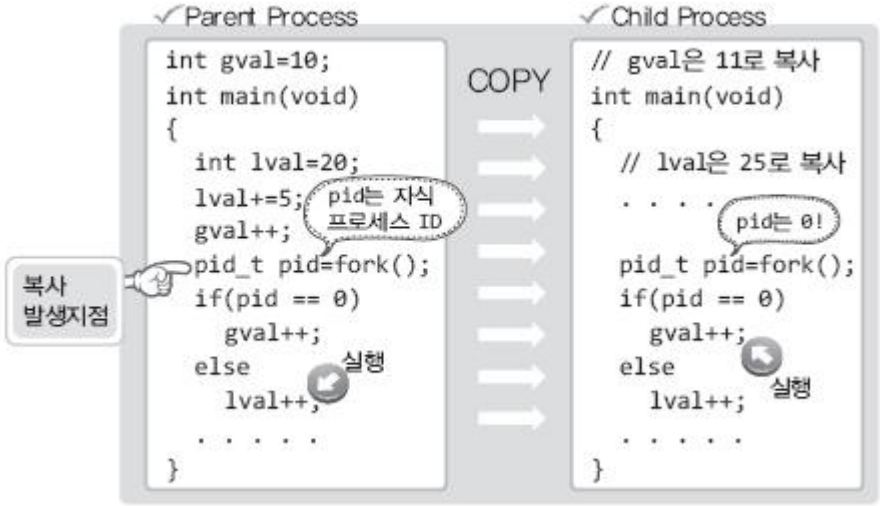
fork 함수의 호출을 통한 프로세스의 생성

```
#include <unistd.h>

pid_t fork(void);
```

➡ 성공 시 프로세스 ID, 실패 시 -1 반환

fork 함수가 호출되면, 호출한 프로세스가 복사되어 fork 함수 호출 이후를 각각의 프로세스가 독립적으로 실행하게 된다.



fork 함수 호출 이후의 반환 값은 다음과 같다. 따라서 반환 값의 차를 통해서 부모 프로세스와 자식 프로세스의 프로그램 흐름을 구분하게 된다.

- 부모 프로세스 fork 함수의 반환 값은 자식 프로세스의 ID
- 자식 프로세스 fork 함수의 반환 값은 0

fork 함수를 호출한 프로세스는 **부모 프로세스**,

fork 함수의 호출을 통해서 생성된 프로세스는 **자식 프로세스**

자식프로세스가 종료되면 어떤 상태이던 바디가 차지하던 메모리는 바로 반환된다
하지만 pcb의 경우 부모가 직접 회수하지 않으면 pcb는 회수되지 않는다.

좀비 프로세스의 이해

좀비 프로세스란?

- 실행이 완료되었음에도 불구하고, 소멸되지 않은 프로세스
- 프로세스도 main 함수가 반환되면 소멸되어야 한다.
- 소멸되지 않았다는 것은 프로세스가 사용한 리소스가 메모리 공간에 여전히 존재한다는 의미이다.

좀비 프로세스의 생성 원인

- 자식 프로세스가 종료되면서 반환하는 상태값이 부모 프로세스에게 전달되지 않으면 해당 프로세스는 소멸되지 않고 좀비가 된다.

- 인자를 전달하면서 exit를 호출하는 경우
- main 함수에서 return문을 실행하면서 값을 반환하는 경우

자식 프로세스의 종료 상태 값이 운영체제에 전달되는 경로

좀비 프로세스의 생성 확인

zombie.c의 일부

```
pid_t pid=fork();

if(pid==0)  // if Child Process
{
    puts("Hi, I am a child process");
}
else
{
    printf("Child Process ID: %d \n", pid);
    sleep(30);  // Sleep 30 sec.
}

if(pid==0)
    puts("End child process");
else
    puts("End parent process");
return 0;
```

```
root@my_linux:/tcpip# gcc zombie.c -o zombie
root@my_linux:/tcpip# ./zombie
Hi, I am a child process
End child process
Child Process ID: 10977
```

Zombie

= = 좀비프로세스의 다른이름

```
root@my_linux:/tcpip# ps au
USER    PID   %CPU %MEM    VSZ   RSS  TTY      STAT START   TIME COMMAND
. . . . .
root    10976 0.0  0.0   1628   368  pts/0    S+   20:26   0:00 ./zombie
root    10977 0.0  0.0      0     0  pts/0    Z+   20:26   0:00 [zom] <defunct>
. . . . .
```

좀비 프로세스의 소멸: wait 함수의 사용

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

pid_t wait(int * statloc);
        → 성공시 종료된 자식 프로세스의 ID, 실패 시 -1 반환
```

- 자식 프로세스가 종료되면서 전달한 값이 매개변수로 전달된 변수에 저장되는데, 이 저장값에는 전달값 이외의 다른 정보도 함께 포함되어 있다.

- 저장값에서 필요한 값을 분리하기 위한 매크로 함수
  - **WIFEXITED** : 자식 프로세스가 정상 종료한 경우 '참(true)=1'을 반환한다.
  - **WEXITSTATUS**: 자식 프로세스의 전달 값을 반환한다.

- wait 함수는 블로킹 상태에 빠질 수 있다.

정상종료
return
exit(1)
등을 통해 죽은경우
exit 의 경우도 내 프로그램
내에서 예상치 못한 상황이지만
내가 쓴 코드(exit)에 의해서 죽은것이니
정상종료이다

비정상 종료
signal등에 의한 종료가 있다.

좀비 프로세스의 소멸: wait 함수의 사용

```
int status;
pid_t pid=fork();

if(pid==0)
{
    return 3;
}
else
{
    printf("Child PID: %d \n", pid);
    pid=fork();
    if(pid==0)
    {
        exit(7);
    }
    else
    {
        printf("Child PID: %d \n", pid);
        wait(&status);
        if(WIFEXITED(status))
            printf("Child send one: %d \n", WEXITSTATUS(status));
        wait(&status);
        if(WIFEXITED(status))
            printf("Child send two: %d \n", WEXITSTATUS(status));
        sleep(30);  // Sleep 30 sec.
    }
}
```

자식 프로세스 생성 종료

자식 프로세스 생성 종료

부모 프로세스 실행 영역

```
root@my_linux:/tcpip# gcc wait.c -o wait
root@my_linux:/tcpip# ./wait
Child PID: 12337
Child PID: 12338
Child send one: 3
Child send two: 7
```

- pid_t wait(int *status);

- WIFEXITED

  자식 프로세스가 정상 종료한 경우 '참(true)'을 반환한다.

- WEXITSTATUS

  자식 프로세스의 전달 값을 반환한다.

wait 함수의 경우 자식 프로세스가 종료되지 않은 상황에서는 반환하지 않고 **블로킹 상태에 놓인다**는 특징이 있다.

좀비 프로세스의 소멸2: waitpid 함수의 사용

wait(statloc) = = waitpid(-1,statloc,0)

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

pid_t waitpid(pid_t pid, int * statloc, int options);
```

➜ 성공 시 종료된 자식 프로세스의 ID(또는 0), 실패 시 −1 반환

- pid — 종료를 확인하고자 하는 자식 프로세스의 ID 전달, 이를 대신해서 −1을 전달하면 wait 함수와 마찬가지로 임의의 자식 프로세스가 종료되기를 기다린다.
- statloc — wait 함수의 매개변수 statloc과 동일한 의미로 사용된다.
- options — 헤더파일 sys/wait.h에 선언된 상수 WNOHANG을 인자로 전달하면, 종료된 자식 프로세스가 존재하지 않아도 블로킹 상태에 있지 않고, 0을 반환하면서 함수를 빠져 나온다.

만약 option에 0넣을경우
〉wait와 같이 블락됨

wait 함수는 블로킹 상태에 빠질 수 있는 반면, waitpid 함수는 블로킹 상태에 놓이지 않게끔 할 수 있다는 장점이 있다.

waitpid 함수의 예

waitpid.c

```
int main(int argc, char *argv[])
{
    int status;
    pid_t pid=fork();

    if(pid==0)
    {
        sleep(15);
        return 24;
    }
    else
    {
        while(!waitpid(-1, &status, WNOHANG))
        {
            sleep(3);
            puts("sleep 3sec.");
        }

        if(WIFEXITED(status))
            printf("Child send %d \n", WEXITSTATUS(status));
    }
    return 0;
}
```

```
root@my_linux:/tcpip# gcc waitpid.c -o waitpid
root@my_linux:/tcpip# ./waitpid
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
sleep 3sec.
Child send 24
```

waitpid 함수 호출 시 첫 번째 인자로 -1, 세 번째 인자로 WNOHANG가 전달되었으니, 임의의 프로세스가 소멸되기를 기다리되, 종료된 자식 프로세스가 없으면 0을 반환하면서 함수를 빠져나온다.

시그널 핸들링

시그널과 시그널 등록의 이해

시그널을 사용하는 이유
〉자식이 좀비가 될때 신호를 받아서 처리하기 위해!

- 시그널이란?
  - 특정 상황이 되었을 때 운영체제가 프로세스에게 해당 상황이 발생했음을 알리는 일종의 메시지를 가리켜 시그널이라 한다.

- 시그널 등록이란?
  - 특정 상황에서 운영체제로부터 프로세스가 시그널을 받기 위해서는 해당 상황에 대해서 등록의 과정을 거쳐야 한다.

- 등록 가능한 시그널의 예

• SIGALRM	alarm 함수호출을 통해서 등록된 시간이 된 상황
• SIGINT	CTRL+C가 입력된 상황
• SIGCHLD	자식 프로세스가 종료된 상황

시그널과 시그널 함수

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

➡ 시그널 발생시 호출되도록 이전에 등록된 함수의 포인터 반환

시그널 등록에 사용되는 함수

- 함수 이름 signal
- 매개변수 선언 int signo, void(*func)(int)
- 반환형 매개변수형이 int이고 반환형이 void인 함수 포인터

시그널 등록의 예

signal(SIGCHLD, mychild);	자식 프로세스가 종료되면 mychild 함수를 호출해 달라!
signal(SIGALRM, timeout);	alarm 함수호출을 통해서 등록된 시간이 지나면 timeout 함수호출!
signal(SIGINT, keycontrol);	CTRL+C가 입력되면 keycontrol 함수를 호출해 달라!

시그널 등록되면, 함께 등록된 함수의 호출을 통해서 운영체제는 시그널의 발생을 알린다.

시그널 핸들링 예제

signal.c

```
void timeout(int sig)
{
    if(sig==SIGALRM)
        puts("Time out!");
    alarm(2);
}
void keycontrol(int sig)
{
    if(sig==SIGINT)
        puts("CTRL+C pressed");
}

int main(int argc, char *argv[])
{
    int i;
    signal(SIGALRM, timeout);
    signal(SIGINT, keycontrol);
    alarm(2);

    for(i=0; i<3; i++)
    {
        puts("wait...");
        sleep(100);
    }
    return 0;
}
```

이 예제에서 보이는 signal 함수는 운영체제 별로 동작방식의 차이를 보이기 때문에 이어서 설명하는 sigaction 함수를 대신 사용한다. signal 함수는 과거 프로그램과의 호환성을 유지하기 위해서 제공된다.

```
root@my_linux:/tcpip# gcc signal.c -o signal
root@my_linux:/tcpip# ./signal
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

시그널이 발생하면, sleep 함수의 호출을 통해서 블로킹 상태에 있던 프로세스가 깨어난다. 그래서 이 예제의 경우 코드의 내용대로 300초의 sleep 시간을 갖지 않는다.

시그널이 오는 시간은 정해져있지않다
>언제 올지 모르므로 비동기적으로 날아온다고 함

sigaction 함수

구조체와 함수의 이름이 동일

```
#include <signal.h>

int sigaction(int signo, const struct sigaction * act, struct sigaction * oldact);
```

➜ 성공 시 0, 실패 시 -1 반환

- signo — signal 함수와 마찬가지로 시그널의 정보를 인자로 전달.
- act — 첫 번째 인자로 전달된 상수에 해당하는 시그널 발생시 호출될 함수(시그널 핸들러)의 정보 전달.
- oldact — 이전에 등록되었던 시그널 핸들러의 함수 포인터를 얻는데 사용되는 인자, 필요 없다면 0 전달.

```
struct sigaction
{
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
```

sigaction 구조체 변수를 선언해서, 시그널 등록 시 호출될 함수의 정보를 채워서 위의 함수 호출 시 인자로 전달한다. sa_mask의 모든 비트는 0, sa_flags는 0으로 초기화! 이 둘은 시그널관련 정보의 추가 전달에 사용되는데, 좀비의 소멸을 목적으로는 사용되지 않는다.

sigaction 함수의 호출 예

sigaction.c

```
void timeout(int sig)
{
    if(sig==SIGALRM)
        puts("Time out!");
    alarm(2);
}

int main(int argc, char *argv[])
{
    int i;
    struct sigaction act;
    act.sa_handler=timeout;
    sigemptyset(&act.sa_mask);
    act.sa_flags=0;
    sigaction(SIGALRM, &act, 0);

    alarm(2);

    for(i=0; i<3; i++)
    {
        puts("wait...");
        sleep(100);
    }
    return 0;
}
```

```
root@my_linux:/tcpip# gcc sigaction.c -o sigaction
root@my_linux:/tcpip# ./sigaction
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

시그널 핸들링을 통한 좀비 프로세스의 소멸

```
int main(int argc, char *argv[])
{
    pid_t pid;
    struct sigaction act;
    act.sa_handler=read_childproc;
    sigemptyset(&act.sa_mask);
    act.sa_flags=0;
    sigaction(SIGCHLD, &act, 0);
    . . . . .
```

SIGCHLD에 대해서 시그널 핸들링을 등록하였으니, 이 때 등록된 함수 내에서 좀비의 소멸을 막으면 좀비 프로세스는 생성되지 않는다.

```
void read_childproc(int sig)
{
    int status;
    pid_t id=waitpid(-1, &status, WNOHANG);
    if(WIFEXITED(status))
    {
        printf("Removed proc id: %d \n", id);
        printf("Child send: %d \n", WEXITSTATUS(status));
    }
}
```

좀비를 없앤다 == wait or waitpid사용

가장 기본적인 좀비의 소멸 코드로 함수가 정의되어 있다.