Ch3. Process (완)

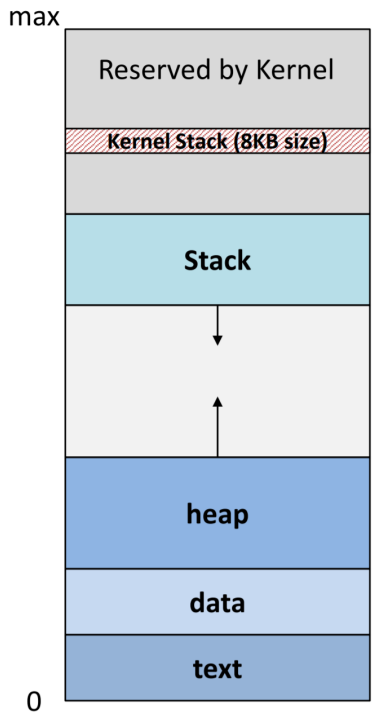
Created	@Jun 3, 2020 5:48 PM
Tags	

Concept of Process

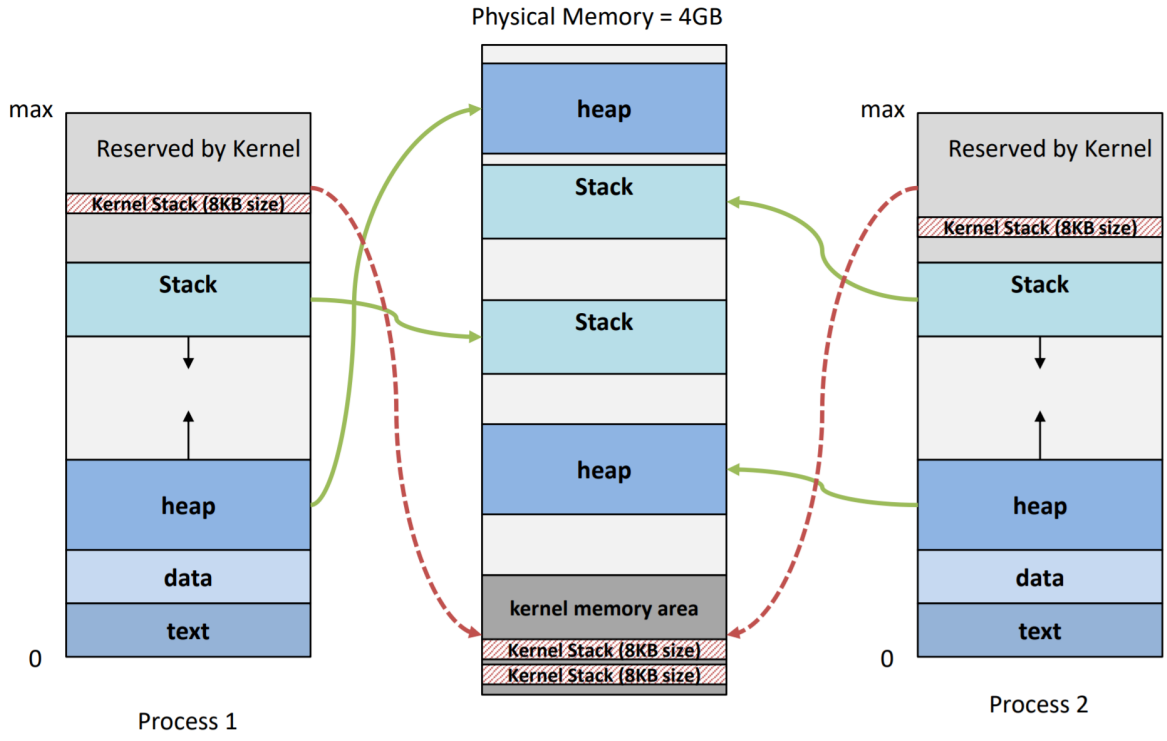
- Process : **Program in execution** = **Instantiated program**
- 일의 단위 = job, task
- 프로그램 : Passive, 프로세스 : Active
- (당연히) 프로그램 하나에 여러 프로세스가 존재할 수 있다.
- 프로세스에는 어떤 정보들이 들어가야 하나?
  - 현재 인스트럭션 주소 : PC
  - 현재 레지스터값
  - 메모리값 (Stack, Heap, 전역변수, ...)
  - 부모 프로세스 정보

Process Memory Layout

Remind: **Virtual Memory**
각 프로세스는 일정한 메모리 공간(logical space, fake space)을 가진다고 내부적으로 간주한다.
32bit의 경우 4GB.
그런데, 이는 실제 물리적 메모리 얘기가 아니다. 각 프로세들이 모두 이러한 메모리 구조를 가진다고 간주하는 것이다.

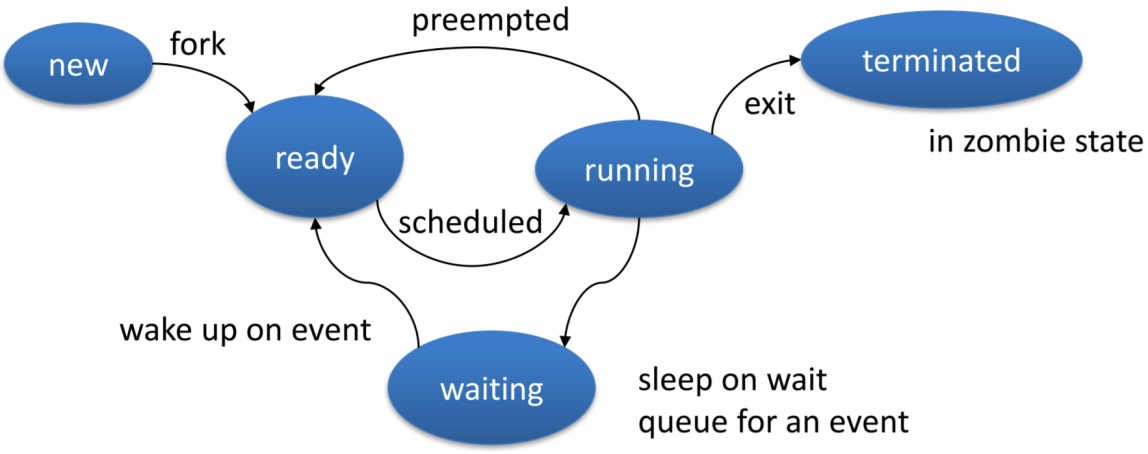


- **Stack area** (= Call stack, Activation area)
  - 위에서 아래로 자란다.
  - 함수 파라미터, 리턴 주소값, 지역변수 등
  - 함수 호출이 종료되면 다시 올라감.
- **Heap area**
  - 아래에서 위로 자란다.
  - 동적할당되는 메모리 (malloc()이나 brk() 시스템콜 등으로 생김)
  - 한번 올라가면, 다시 내려가지 않음.
- **Data segment**
  - 전역변수
- **Text segment**
  - 프로그램 코드 + PC 등 레지스터값



- **Kernel Memory Area**
  - 모든 프로세스마다 1GB씩 차지하는 영역.
  - Kernel binary 코드들이 저장되어 있다.
  - 물리 메모리에서도 한 영역으로, 모든 프로세스가 이를 공유한다.
  - Kernel Stack은 그냥 물리 메모리에서 가져오는 듯. 이는 공유하지 않는 것 같다.
- **Stack, Heap**
  - 각 프로세스들이 보는 메모리는 가상메모리이다. = Logical Space.
  - 각각 물리메모리의 어딘가에 저장되어 있다.
  - Read-Only가 아니다. 프로세스마다 State는 다를 수 있다.
- **Text**
  - Read-Only이다. 프로그램 코드는 공유될 수 있다.

Process State



- **New**
  - 프로세스가 **처음 생성되었을 때**
- **Ready**
  - 프로세스가 **프로세서에게 할당되기를 기다리고 있을 때**
  - Ready Queue에 있다.
- **Running**
  - 프로세스가 **프로세서에 할당되어 실행되고 있을 때**

> 어떠한 상황에도 **각 Core는 한 번에 하나의 프로세스만 실행**한다.
> Time-sharing 같은 걸로 병렬, 동시실행 흉내만 내는 것. 4Core의 경우, 시간 정지하고 보면 프로세스 4개다.

- **Waiting**
  - 프로세스가 **실행 중 작업을 완료하기 위해 특정 이벤트를 기다리고 있을 때** (I/O Request - 입출력 대기 등)
  - Wait Queue에 있다.
- **Terminated**
  - 프로세스가 **실행 중 종료되었을 때**

1. **New → Ready** : fork()로 프로세스가 생성되고, 할당되기를 기다림
2. **Ready → Running** : 프로세스가 프로세서에 의해 Schedule됨.

1. **Running → Waiting** : I/O Request 등으로 입출력 대기. Sleep on wait.
    1. **Waiting → Ready** : 이벤트를 받아서 깨어남. 다시 Ready Queue로 감에 유의.
2. **Running → Ready** : Preemted. OS가 프로세스를 'kick out'해버린 경우.
3. **Running → Terminated** : 프로세스가 종료됨. 프로세스는 Zombie state가 된다.

Process Control Block (PCB)

PCB : 커널에서 쓰이는, 프로세스를 나타내는 자료구조

PCB에 들어있는 정보 - 프로세스를 나타내기 위한 모든 정보들

- **프로세스 상태** : New, Ready, Running, Waiting, Halted, ...
- **Program Counter** : 다음 인스트럭션의 주소
- **CPU 레지스터값**
- **CPU 스케쥴링 정보**
    - 프로세스 우선순위, 스케쥴링 큐를 가리키는 포인터, 스케쥴링 파라미터
- **메모리 관리 정보**
- **Accounting 정보**
    - CPU 사용 시간, 프로세스 번호
    - **스케쥴링 결정을 하기 위함**
- **I/O 상태 정보**
    - 입출력장치 리스트, 오픈 파일 리스트

PCB : 프로세스가 Preempted되고 다시 Restore되어 Running으로 들어올 때, 모든 상태를 다시 불러오기 위해 쓰임.
Context Switch가 일어날 때, PCB에 저장된 정보를 백업하고, 해당 프로세스로 돌아올 때 PCB를 참고한다.

PCB Representation in Linux

PCB는 OS에 따라 다양한 방식으로 구현된다.

리눅스에서는 `struct task_struct` 로 존재한다.

- `long state` : 프로세스 상태
- `struct sched_entity se` : 스케쥴링 정보
- `struct task_struct *parent` : 부모 프로세스 정보
- `struct list_head children` : 자식 프로세스들 정보
- `struct files_struct *files` : 오픈 파일 리스트
- `struct mm_struct *mm` : Address space

커널은 PCB의 목록을 관리한다.

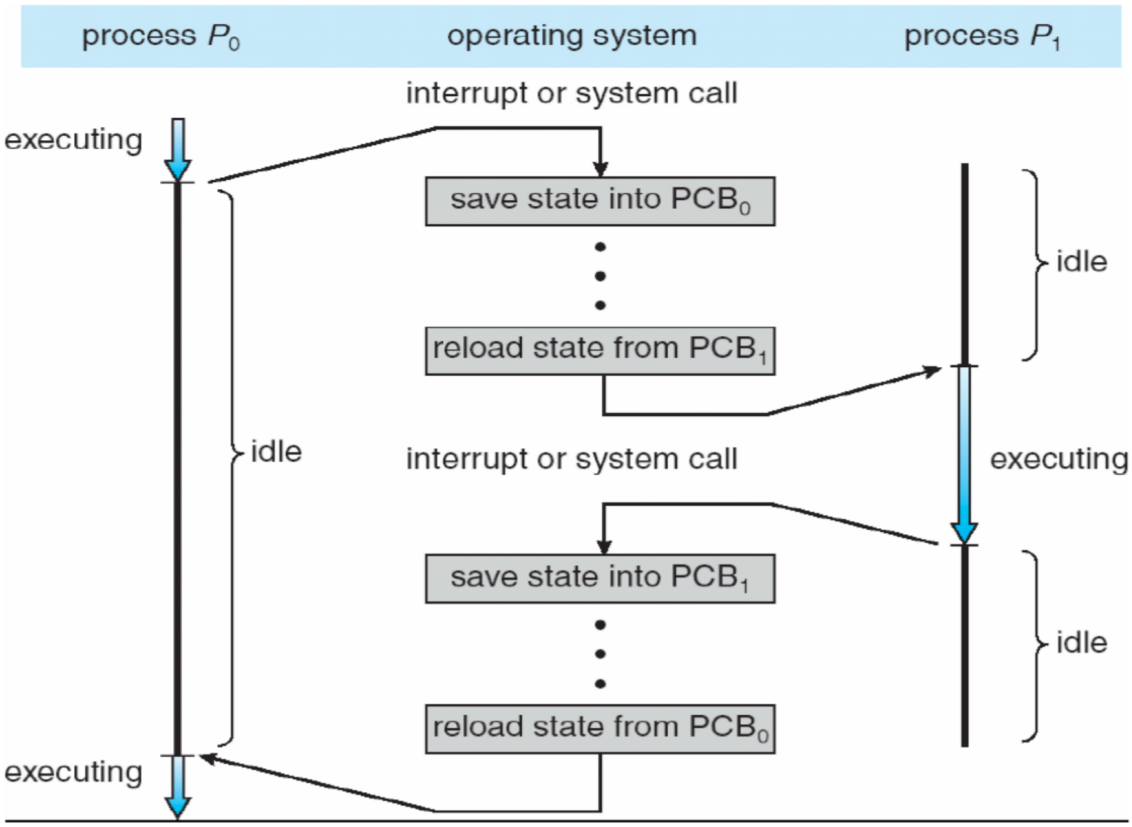
- `struct task_struct *current` : 현재 PCB를 가리키는 포인터 변수
- 현재 프로세스 (state)를 업데이트하려면 : `current->state = new state;`

Threads (→ Ch4)

- 프로세스보다 작은 개념. **프로세스 내부에 있는 하나의 실행 sequence.**
- **분리된 PC, 레지스터값, 스택 영역을 가진다.**
- 리눅스에서, 실제 스케쥴링 단위는 Thread이다.
- (당연히) **하나의 프로세스는 여러 스레드를 가질 수 있다.**
- 서로 다른 프로세스의 스레드끼리는 통신할 수 없다. 한 프로세스 내의 스레드끼리는 자유롭게 통신 가능하다.
    - Process isolation property (Provided by OS)

Process Switch and PCB

Process Switch : 프로세스가 실행되다가 인터럽트가 발생해, OS가 개입하여 프로세서에 할당된 프로세스를 바꾸는 것.



1. 프로세스 P0이 실행 중, P1은 idle.
2. P0에서 인터럽트나 시스템 콜이 발생한다.
3. PCB0에 P0의 현재 상태 저장 후, PCB1에서 P1의 상태를 불러와서 진행한다.
    1. 이 과정에서 P0은 Ready/Waiting, P1은 Running으로 프로세스 상태가 변경된다.
4. P1 실행하다가, 또 P1에서 인터럽트나 시스템 콜이 발생한다.
5. 이후 똑같은 과정으로 P1은 idle, P0이 실행된다.

Process Scheduling

두 개 프로세스 사이의 Switching은 봤다. 그렇다면 많은 프로세스들은 어떻게 관리해야 하는가?

→ **Process scheduling.** 다음으로 실행할 프로세스를 적절히 선택해야 한다.

- Multiprogramming : CPU utilization을 최대화하기 위함.
    - CPU utilization : 단위 시간당 CPU를 얼마나 사용하는지. → 항상 무언가를 해야 한다.

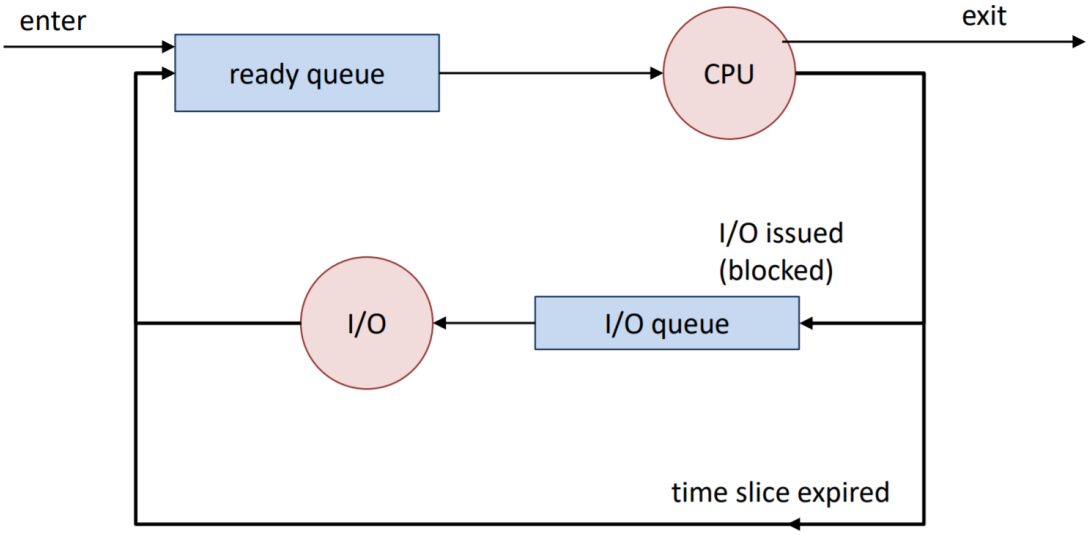
    - Resource utilization : 단위 시간당 리소스를 얼마나 사용하는지.
- Time-sharing : 프로세스를 Interactive하게 만들기 위함.
- **Process scheduling : 프로세서에서 실행할 다음 프로세스를 선택하는 것.**

Scheduling Queues

프로세스는 Queue 사이에서 왔다갔다한다.
Waiting Queue에서 실행 끝난 프로세스는 다시 Ready Queue로 가는 것.

- **Job Queue**
    - 모든 프로세스(PCB)들.
    - 요즘은 없다. 옛날 배치 시스템, 중앙집중형 시스템 정도에만 존재.
        - job 고르고 메모리에 로드하면, 바로 CPU에서 사용할 준비가 된 것. (→ Ready Queue)
          메모리에 로드되면 바로 실행된다.
          메모리는 CPU와 직접 통신 가능한 Storage일 뿐.
- **Ready Queue**
    - 프로세서에서 돌아갈 준비가 된 모든 프로세스들.
    - **PCB의 Linked List**로 구현된다.
- **Device Queue**
    - I/O 장치를 기다리는 프로세스들의 List.
    - **각 장치는 각 장치만의 Device Queue를 가진다.**

Queuing diagram



Ready Queue에서 PCB가 하나씩 나와서 CPU로 들어가고,
CPU에서는 프로세스를 종료시키던지 (exit, Terminated),
I/O가 필요해서 I/O queue (Device Queue)로 PCB를 넣던지, (I/O 끝나면 다시 Ready Queue로 들어가겠죠?),
CPU에서 할당한 시간이 끝나서 Preempted돼서 다시 Ready Queue로 들어가던지 한다.

Scheduler

스케줄러는 어떠한 정책에 의해 큐에서 다음 프로세스를 선택한다.

- **Long-term scheduler**
  - **Job Queue → Ready Queue(Memory)**. 프로세스를 옮김.
    - 프로세스 상태는 **New → Ready**가 되겠다.
  - CPU 외부에서 작업. 가끔 수행됨.
  - **I/O-bound 프로세스와 CPU-bound 프로세스를 적절히 섞어서**, 메모리에 올려주어야 한다.
    - → Underutilize를 막아야 한다.
- **Short-term scheduler**

- **Ready Queue → 프로세서**. 프로세스를 할당.
  - 프로세스 상태는 **Ready → Running**가 되겠다.
- CPU 내부에서 작업. 자주 수행됨.

프로세스의 타입

- **I/O-bound** : I/O 작업을 하는 데 대부분의 시간을 소모하는 프로세스.
- **CPU-bound** : CPU를 쓰는 데 대부분의 시간을 소모하는 프로세스.

스케쥴링 기준

- CPU utilizaiton 최대화
- 평균 대기시간 최소화
- 응답시간 최소화
- 공평하게!
- 등등...

Context switch

> 💡 **Context : Process state stored in PCB**
> **Context Switch : 현재 프로세스 상태 저장, 다른 프로세스 상태 복원**

- **Context Switch는 빨라야 한다.**
  - Context switch time에는 아무런 쓸모있는 작업을 수행할 수 없다. 낭비되는 시간이다.
- 하드웨어가 Context switch time을 줄이는 데 도움을 줄 수 있다.
  - Register set이 여러 개 있고, 단순히 현재 사용 중인 Register set을 가리키는 포인터만 변경시키는 식으로 하면 Context Switch가 더 빠르다.

Operations on Processes

Lifecycle of process!

Process creation

- **모든 프로세스는 부모자식 관계.** 모든 프로세스는 부모를 가진다.
  - → 프로세스들은 트리 구조를 이룬다.
- **PID** : 프로세스를 식별할 수 있는 정수.
- `init` 프로세스 : PID = 1. 제일 먼저 생성됨. 트리의 루트.
- ***d** (sshd, httpd, kthreadd, ...) : Daemon 프로세스를 나타냄.
  프로세스의 특별한 종류로, 무한루프를 가지며 영원히 종료되지 않음

Resource on process creation

PID, connection, port number, ... 프로세스가 실행되려면 많은 리소스들이 필요하다.

- 자식 프로세스가 OS로부터 추가적인 리소스를 얻을 수도 있고,
- 부모 프로세스의 리소스에 한정될 수도 있다.
  - System의 overloading 방지

On creating a new child process

새 자식 프로세스를 만들 때는?

- 실행 타입
  - 부모와 자식이 같이 돌아가거나 (concurrently)
  - 부모가 자식이 끝날 때까지 기다리거나
- 프로그램 타입
  - 자식이 부모 프로세스의 완전한 copy이거나
  - 자식이 아예 새로운 프로그램이거나

Process creation in UNIX/Linux

3개의 시스템 콜을 먼저 알아보자.

- **fork()**
  - **부모 프로세스와 모든 것이 동일한, 새로운 프로세스를 만든다.**
    코드, 메모리 내용, ... 모두 동일하다.

    - fork() 이후 부모와 자식 프로세스는 동시에 - concurrently하게 실행된다.
    - **부모 프로세스와 자식 프로세스의 fork() 리턴 값은 다르다.**
        - **부모 프로세스**에서는 **fork()가 자식 프로세스의 pid를 리턴**
        - **자식 프로세스**에서는 **fork()가 0을 리턴**
- **exec()** (execlp, execvp, ...)
    - **프로세스의 메모리 공간을 새로운 프로그램으로 대체한다.**
    - 디스크에서 새 프로그램을 로드해서, 새 프로그램을 시작한다. 모든 게 지워지고, 새로운 시작.
    - exec() does not return control
        - exec() 즉시 모든 게 지워지고 새로운 프로그램이 시작되므로, exec() 이후의 코드는 쓸모가 없어진다.
- **wait()**
    - 부모는 자식 프로세스가 끝날 때까지 wait()으로 기다릴 수 있다.
    - "blocking system call" : wait() 콜하면 자식 프로세스가 끝날 때까지 다음 라인으로 넘어가지 않는다. Block된다.

Example code using fork() and exec()

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

int main() {
  pid_t pid;
  pid = fork();
  if (pid < 0) {  // Error
    fprintf(stderr, "Fork failed");
    return 1;
  } else if (pid == 0) {  // Child part
      execlp("/bin/ls", "ls", NULL);  // Child process becomes "ls" process.
  } else {  // Parent part
    wait(NULL);  // Blocks until child exits.
    printf("Child complete");
  }

  return 0;
}
```

Process Termination

- 프로세스는 보통 **exit()**을 호출해 프로세스를 끝낸다.
  - 부모 프로세스에게 상태값을 리턴 (보통 0이면 정상종료, 0이 아닌 값이면 문제가 있음을 알림)
  - 메모리, 오픈 파일 등 리소스가 해제됨
  - **Explicitly** exit() : 직접 exit() 시스템 콜 호출
  - **Implicitly** exit() : 프로그램에서 main 종료되면 리턴할 때 자동으로 exit() 호출됨
- **부모 프로세스 또한 자식 프로세스를 끝낼 수 있다.**
  - 자식 프로세스가 리소스를 너무 많이 사용하는 경우
  - 자식 프로세스가 더 이상 필요없는 경우
  - 부모 프로세스가 종료될 것인데, OS에서 고아 프로세스를 허용하지 않는 경우
    - 그 자식 프로세스들이 연속적으로 모두 종료된다. → **cascading termination**
- Remind: wait()은 부모 프로세스에서 자식 프로세스가 끝날 때까지 기다릴 때 사용한다.
  - **wait()으로 자식 프로세스의 상태를 받아올 수 있다.**

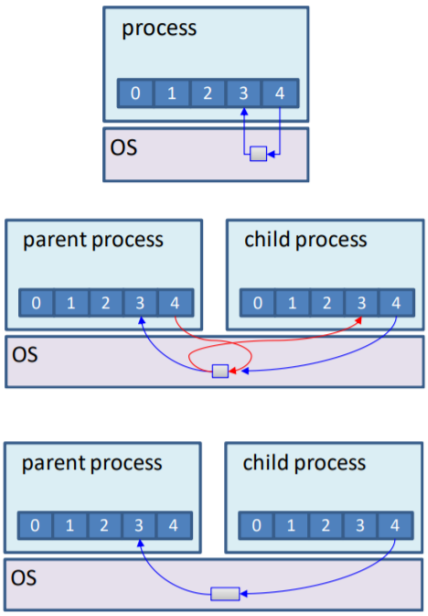
```
pid_t pid;
int status;
pid = wait(&status);
```

- **좀비 프로세스 : 종료되었지만, 부모에서 wait()을 call한 상태가 아닌 경우** (부모 프로세스가 죽었거나, ...)
  - 계속 리턴값을 가지고 있다
  - 부모 프로세스에서 영원히 wait()을 콜하지 않으면?
    - **init 프로세스가 주기적으로 wait()을 call해서 zombie cleanup을 해 줌. (systemd)**

Remote Procedure Calls

Client측, Server측에 **각각 Stub이 존재**해서, **네트워크를 통해 파라미터를 전달**하고, **네트워크를 통해 함수를 call**한다.

Pipes



- **일반적인 파이프 (Ordinary Pipe)**
  - 단방향 통신.
  - `pipe(int fd[])` 시스템 콜 → `fd[0] : read-end, fd[1]: write-end`
  - **파이프를 만든 프로세스에서만 접근할 수 있다. (parent-child 관계에서만 가능)**
    - 부모 프로세스에서 파이프를 만들고, fork()해서 자식 프로세스와 통신하는 식으로 쓰인다.
    - 자식 프로세스는 부모 프로세스의 오픈 파일도 다 받아오기 때문에 가능.
- 다른 프로세스와의 통신은? → **Named Pipes (FIFO)** *(그 First-In-First-Out하고 다른 거)*
  - 양방향 통신. (half-duplex)

- **parent-child 관계 필요 X.**
- **한번 만들면, 여러 프로세스에서 사용할 수 있다.** 하나의 파일처럼 존재한다.
- Lifecycle이 프로세스에 종속되지 않는다.
- `mkfifo()` 시스템 콜
- 프로세스들은 같은 physical machine에 있어야 함.

Pipe initialization code sample

```
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main() {
  char write_msg[BUFFER_SIZE] = “Greetings”;
  char read_msg[BUFFER_SIZE];

  pid_t pid;
  int fd[2];

  /* create the pipe */
  if (pipe(fd) == -1) {
    return 1; /* pipe creation failed */
  }

  pid = fork();
  if (pid < 0) return 1; /* fork failed */
  if (pid > 0) { /* parent */
    close(fd[READ_END]);
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
    close(fd[WRITE_END]);
  } else { /* child */
    close(fd[WRITE_END]);
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf(“read %s”, read_msg);
    close(fd[READ_END]);
  }

  return 0;
}
```

Ch3. Process 끝!