Exceptional Control Flow: Exceptions & Processes

Control Flow

<startup>
Time
inst₁ → $inst_1$, $inst_2$, ..., $inst_n$
<shutdown>

Processor는 한 가지만.

이전까지: control flow를 바꾸는 2 mechanism.
- Jumps & branches (점프와 분기)
- Call & return (함수의 호출 & 반환)
- ∴ react to changes in program state

우리가 다루고자 하는 것.
- react to changes in system state
- data가 network adapter로부터 도착
- 0으로 나누는 instruction
- keyboard에 ctrl+C
- system timer expire

Exceptional Control Flow

- Low level mechanisms
  1. Exceptions
     - system event에 대한 change in control flow
     - hardware와 software의 결합
     - (제어권) Control을 OS kernel에게 넘기는 것 (event에 따라)
     - ex) divide by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C
- Higher level mechanisms
  2. Process context switch → OS software와 hardware timer로 implemented
  3. Signals → OS software로 implemented
  4. Nonlocal jumps → setjmp() and longjmp() → C의 runtime library

Event

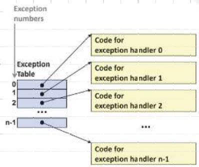
User code / Kernel Code

Event → I_current → Exception → Exception processing by Exception handler → I_next

kernel의 예외처리 3가지 옵션
① Return to I_current (현재 명령으로 복귀)
② Return to I_next ((현재의) 다음 명령으로 복귀)
③ Abort (종료)

Exception Tables

OS 안에 Exception Table
예외에 맞는 처리 (Handler) 목록화



Class of Exceptions

Class	Cause	Async/sync	Return Behavior
① Interrupt	Signal from I/O device	Async	Always returns to (next) instruction → I_next
② Trap	Intentional exception	Sync	
③ Fault	Potentially recoverable error	Sync	Might return to current instruction
④ Abort	Nonrecoverable error	Sync	Never returns

Asynchronous (비동기식): 다른 일을 하면서 기다리기
Synchronous (동기식): 다른 일을 하지 않고 기다리기
: direct result of executing an instruction

Asynchronous Exceptions

① Interrupt
- processor 외부의 이벤트 (I/O 장치) 때문에 발생
- data가 언제 오는지 알 수 X
- interrupt가 걸리면, 현재 수행 명령 끝내고 잠시 suspend, 다시 돌아가 다음 명령 수행
- ※ 항상 "next" instruction으로 return (I_next)
- disk나 network에서 data 읽어오는 상황 ex) fread

examples

1-1) Timer Interrupt
- 매 ms마다 external timer chip이 interrupt trigger
- kernel이 user program으로부터 control 넘겨 받을 때 사용됨.

이거랑 Higher level의 process context switch가 뭐가 다른가?
1. timer chip이 timer interrupt 발생
2. user program 중단, 제어권을 kernel에게 넘김.
3. Kernel은 interrupt handler를 실행하고 이 시점에서 process context switch를 할지 말지 결정. 스케줄링 정책에 따라 다른 프로세스로 전환 or 현재 프로세스 실행

1-2) I/O interrupt from external device
- Ctrl-C
- network에서 packet 도착
- disk에서 data 도착

CPU가 현재 명령을 수행하는 중에 Interrupt pin이 High
인터럽트가 꽂힌 현재 명령을 다 수행하고
I_curr → I_next
인터럽트 핸들러 수행
항상 I_next로 복귀

Process A가 외부장치에 요청보냄 → Process B로 switch → Process B의 명령 실행중 Interrupt → Interrupt handler 동작 → Process A의 I_next로 복귀

Synchronous Exceptions (Traps, Faults, Aborts)

의도적 / 비의도적

② Traps
- Intentional (의도적)
- ex) system calls, breakpoint traps, special instructions → fopen
- return control to "next" instruction (항상 I_next로 돌아간다)

syscall → kernel의 trap handler → I_next

③ Faults
- Unintentional & possibly recoverable → 행동이 2가지
- ex) page faults (recoverable), protection faults (unrecoverable), floating point exceptions
  - DRAM에 없어서 Disk에서 끌어오는 것
- re-execute faulting (current) instruction or aborts

I_curr → Fault handler 수행
① abort 하거나 → abort
② 현재 instruction 다시 수행하거나.

④ Aborts
- unintentional & unrecoverable
- ex) illegal instruction, parity error, machine check, hardware error
- Abort current program

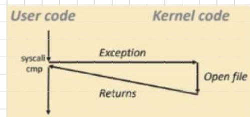
I_curr → abort

System Calls

0	read	read file
1	write	write file
2	open	open file
3	close	close file
4	stat	get info about file
57	fork	create process
59	execve	execute a program
60	_exit	terminate process
62	kill	send signal to process

ex) opening file (trap)

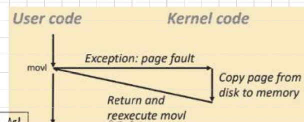
```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%rax     # open is syscall #2
e5d7e:   0f 05               syscall            # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



%rax : syscall number
다른 arguments : %rdi, %rsi, %rdx, %r10, %r8, %r9
return value : %rax

ex) page fault (fault)

```
80483b7:    c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```



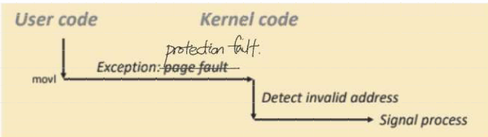
recoverable 하다!

ex) protection fault (fault)

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

protection fault ⟷ segmentation fault?

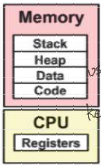
```
80483b7:    c7 05 60 e3 04 08 0d    movl    $0xd,0x804e360
```



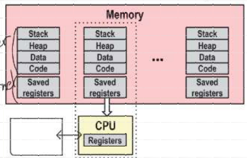
Processes

→ instance of a running program
≠ program or processor

→ 2 key abstractions
① Logical control flow.
↳ context switching
↳ CPU 관점
② Private address space.
↳ virtual memory
↳ 메모리 관점

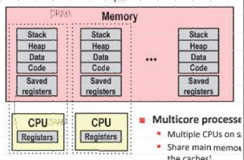


Multiprocessing : The (Traditional) Reality



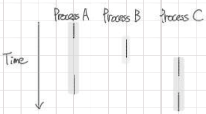
ex) p1부터 처리, 시간 다 지나면
지금까지 처리한 부분까지 OS Kernel 공간의
saved register로 load하고, 주소공간 switch

Multiprocessing : The Modern Reality



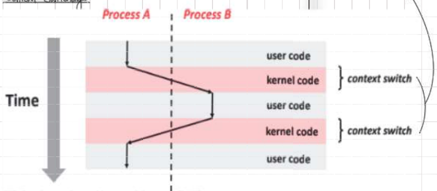
Concurrent Processes

→ 각 process는 logical control flow, concurrent ⟷ sequential



concurrent A & B, A & C
sequential B & C

Context Switching



context switch overhead

System Call Error Handling

→ error 발생시 함수의 반환값 : -1 , 전역변수 "errno"에 에러원인 명시

```
if ((pid = fork()) < 0) {
   fprintf(stderr, "fork error: %s\n", strerror(errno));
   exit(0);
}
```

```
void unix_error(char *msg) /* Unix-style error */
{
   fprintf(stderr, "%s: %s\n", msg, strerror(errno));
   exit(0);
}
```

```
if ((pid = fork()) < 0)
   unix_error("fork error");
```

```
pid_t Fork(void)
{
   pid_t pid;

   if ((pid = fork()) < 0)
      unix_error("Fork error");
   return pid;
}
```

```
pid = Fork();
```

Process ID 얻는법

pid_t getpid(void) 현재 PID

pid_t getppid(void) 부모의 PID

State of Process

① Running : 실행 중이거나, 커널에 의해 스케줄 된 후, 실행되기전에 기다리고 있는 상태

schedule 여부 ↕

② Stopped = Blocked = Waited = Skipped
: 프로세스가 잠시 suspend 되고 언제 실행될 것인지 스케줄되지 않은 상태

③ Terminated : 영구적으로 종료

Terminating Process

종료되는 3가지 이유
- → default action이 termination인 signal을 받았을 때
- → main 함수가 return
- → exit 함수 호출

→ void exit (int status)
- → 'exit status'와 함께 종료
- → 일반적인 경우 0을 넣고? / error 상황이면 non-zero 값
  - ↳ status 인자를 변경하는 것임?

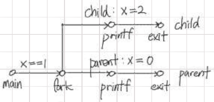
Creating Process

fork

int fork (void)
- → child process 에게는 0을 return
- → parent process 에게는 child pid return
- → call once but return twice
- → concurrent execution : 순서 예측 불가능
- → duplicate but separate address space
- → shared open files : stdout은 same?

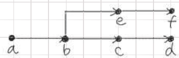
ex01)

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
            exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```



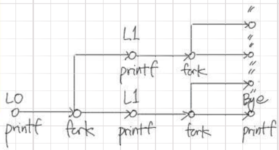
① x=0 x=2
② x=2 x=0
↳ 순서는 바뀔수 o.

a → b
a가 b 전에 일어남.



ex02) two consecutive forks

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



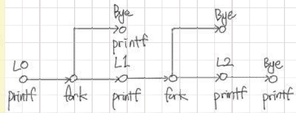
Feasible Output	Infeasible Output
L0	L0
L1	Bye
Bye	L1
Bye	Bye
L1	L1
Bye	Bye
Bye	Bye

03/06

03/11

ex03) Nested forks in parent

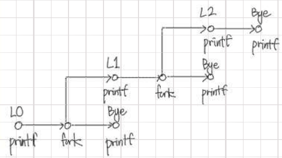
```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```



Feasible Output	Infeasible Output
L0	L0
L1	Bye
Bye	L1
Bye	Bye
L2	Bye
Bye	L2

ex04) Nested forks in children

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```



Feasible Output	Infeasible Output
L0	L0
Bye	Bye
L1	L1
L2	Bye
Bye	Bye
Bye	L2

Reaping Child Processes

- 프로세스가 terminate 되어도, system resource 잡아먹음. (OS 커널 내에 종료된 프로세스에 대한 정보가 남아있음)
  ex) exit status, various OS tables
- 그래서 free 해줘야함. → 안 되면 "zombie"
- Reaping : zombie process를 제거해주는 작업.
  - → parent process가 해야함. (정확히 말하면, parent process가 OS에게 요청하는 것)
    - → wait or waitpid 사용
    - → parent는 exit status information 이 주어지고, kernel이 zombie child process를 delete.
    - → 만약 하지 않는다면.
      - ↳ orphan이 된 child는 Init Process (PID ==1) 가 reap
        - ↳ but programmer가 명시적으로 reaping을 수행해야 Long-Running Process는 유지할수 있음.
          ex) shell과 server는 오랜기간 수행되기에 init에 맡기지 않고, 명시적으로 reaping 해야함.

zombie example

```c
void fork7() {
  if (fork() == 0) {
    /* Child */
    printf("Terminating Child, PID = %d\n", getpid());
    exit(0);                                   // PID 6640
  } else {
    printf("Running Parent, PID = %d\n", getpid());
    while (1)                                  // PID 6639
      ; /* Infinite loop */
  }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks      (부모)
 6640 ttyp9    00:00:00 forks <defunct>   (자식)  zombie를 의미.
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated          parent를 죽임.
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

→ child가 생성되고 나서, 자신이 알아서 종료

→ 그러나 parent는 무한루프 ⇒ 종료×,
parent가 child 지워야하는데 그러지×

parent가 죽어서 →
그 아래 zombie상태인
child 프로세스는 고아가
되어서, init이 대신 kill

non-terminating child example

```c
void fork8()
{
  if (fork() == 0) {
    /* Child */
    printf("Running Child, PID = %d\n",      // PID 6676
      getpid());
    while (1)
      ; /* Infinite loop */
  } else {
    printf("Terminating Parent, PID = %d\n", // PID 6675
      getpid());
    exit(0);
  }
}
```

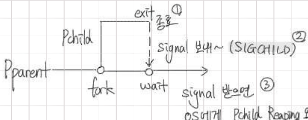
```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks   ← 자식
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

→ child가 안끝나고 있는 상황
→ parent는 먼저 종료
→ defunct도 안 뜨고 살아있음.
→ init이 고아가 된 child를 바로 제거×
왜? parent가 먼저 종료되어 orphan의 존재를 알리지×

wait : Synchronizing with Children.

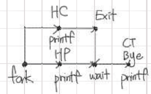
int wait (int *child_status)
- 현재(호출한)프로세스의 Child Process 들이 종료될때까지
현재 process를 suspend
- signal이 오면 현재 process를 wake up 하고,
종료된 Pchild의 pid 전달

wait.h
↳ WIFEXITED, WEXITSTATUS, WIFSIGNALED
WTERMSIG, WIFSTOPPED, WSTOPSIG,
WIFCONTINUED.



wait : Synchronizing with children

```c
void fork9() {
  int child_status;

  if (fork() == 0) {
    printf("HC: hello from child\n");
       exit(0);
  } else {
    printf("HP: hello from parent\n");
    wait(&child_status);
    printf("CT: child has terminated\n");
  }
  printf("Bye\n");
}
```



Feasible Output	Infeasible Output
HC	HP
HP	CT
CT	Bye
Bye	HC

wait 없으면 feasible

Another wait example

```c
void fork10() {
  pid_t pid[N];
  int i, child_status;

  for (i = 0; i < N; i++)                      // random으로
    if ((pid[i] = fork()) == 0) {              // reaping.
      exit(100+i); /* Child */
    }
  for (i = 0; i < N; i++) { /* Parent */
    pid_t wpid = wait(&child_status);
    if (WIFEXITED(child_status))
      printf("Child %d terminated with exit status %d\n",
        wpid, WEXITSTATUS(child_status));
    else
      printf("Child %d terminate abnormally\n", wpid);
  }
}
```

waitpid : waiting for a specific process.

pid_t waitpid (pid_t pid, int &status, int options)

```c
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;                           // N=10

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {                // pid[9]부터
        pid_t wpid = waitpid(pid[i], &child_status, 0);   // reaping
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

forks.c

execve   ex) ls   -al

int execve(char * filename , char *argv[ ] , char* envp[ ])

→ 3개의 인자

→ pid, openfiles, signal context 정보는 유지

→ code, data, stack 은 새로.

→ filename 에는 executable file의 이름.

ex) ls 같은 filename은 /bin 안에 파일 형태로 코드가 저장되어 있는 것

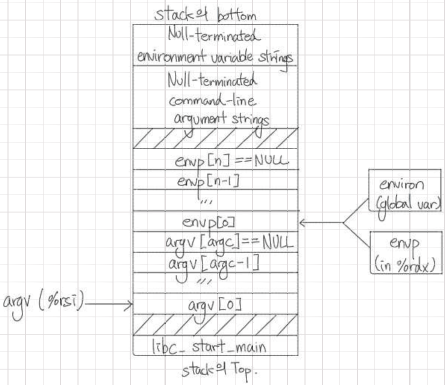
execve는 위치지정해줘야해서 인자로 /bin/ls 문자열 넘김

execvp는 위치 지정 필요없이 그냥 파일명 ls 가능

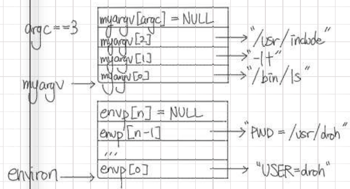
+) ex)

```
char *argv [ ] = {"ls", "-al", NULL};
execvp ("ls", argv);
```

```
char *envp [ ] = {"USER=dave", "PATH=/bin", (char*) 0};
execve ("ls", argv, envp);
```



execve Example




```
if ((pid = Fork()) == 0) {  /* Child runs program */
   if (execve(myargv[0], myargv, environ) < 0) {
      printf("%s: Command not found.\n", myargv[0]);
      exit(1);
   }
}
```