

# Systems Programming

---

Spring 2023

# 3 주차

---

- Review & Summary

- 01-linking
- 02-relocation
- 03-exceptions
- 04-signals

- NO session on 4/4 화

- Q&A

- Will NOT cover the SAME question that already answered in previous sessions

# 01-linking

---

# Questions

linker puzzle 슬라이드 (첨부 pdf) 에서 네번째 시나리오 -- writes to x in p2 will overwrite y  
가 발생하는 이유는, x를 참조할 때 strong symbol인 int type (4 bytes)로 인식하지만, p2에서 x에 write 할 경우에는  
double type으로 인식해서 8 byte를 할당하게 되고, 그러면 메모리에서 x 옆에 있는 y의 자리까지 차지하게 된다는  
것인가요? 그렇다면 왼쪽 모듈에서 x랑 y는 메모리 할당자리가 연속적으로 붙어있다고 가정하는 것인가요?

---

## from SP-session01.pdf (p.7)

2.

01-linking.pdf 38p의 4번째 예시를 보면 p1()이 있는 파일에서 strong symbol로 int x=7을 정의했으며 p2()가 있는 파일에서는 weak symbol로 double x를 정의했는데, 오른쪽 해설을 보면 x에 어떤 숫자를 넣게 된 경우 y를 overwrite 한다고 되어있습니다.

하지만 strong과 weak가 동시에 있는 경우 strong을 우선적으로 사용하기 때문에 int로 정의된 x의 값이 바뀌게 되며, 만약 int 범위를 넘어가는 값을 x에 넣으려 했다면 오버플로우가 일어나서 다른 값이 저장될 뿐, y에는 영향이 없어야 하는것 아닌가요?

결국 1,2번 질문 모두 int랑 double로 같은 변수를 선언했을 때 컴퓨터가 어떤 방식으로 받아들이는지 잘 이해가 가지 않아 질문드립니다.

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** might overwrite **y**!  
Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** will overwrite **y**!  
Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same initialized variable.

**Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.**

# Questions


첨부 pdf 파일 슬라이드에 관한 질문입니다. link-time interpositioning에서, malloc은 \_\_wrap\_malloc으로 resolve 되고, \_\_real\_malloc은 malloc 으로 resolve 되면, 일종의 순환구조가 형성되는 것 아닌가요? (malloc > \_\_wrap\_malloc > \_\_real\_malloc > malloc > \_\_wrap\_malloc ...) 아니면 \_\_wrap\_malloc 안에서 \_\_real\_malloc이 malloc을 resolve 할 때는 다시 \_\_wrap\_malloc 으로 빠지지 않게 하기 위한 장치가 있을까요?

## Program

```
...  
char *ptr  
    = (char *)malloc(size);  
...
```

## Library

```
...  
void *malloc(size_t size)  
{  
    ...  
}  
...
```



# Questions

첨부 pdf 파일 슬라이드에 관한 질문입니다. link-time interpositioning에서, malloc은 \_\_wrap\_malloc으로 resolve 되고, \_\_real\_malloc은 malloc으로 resolve 되면, 일종의 순환구조가 형성되는 것 아닌가요? (malloc > \_\_wrap\_malloc > \_\_real\_malloc > malloc > \_\_wrap\_malloc ...) 아니면 \_\_wrap\_malloc 안에서 \_\_real\_malloc이 malloc을 resolve 할 때는 다시 \_\_wrap\_malloc으로 빠지지 않게 하기 위한 장치가 있을까요?

## Program

```
...  
char *ptr  
    = (char *)malloc(size);  
...
```

## Library

```
...  
void *malloc(size_t size)  
{  
    ...  
}  
...
```

## Interpositioning

```
void *__wrap_malloc(size_t size)  
{  
    ...  
    void *ptr = __real_malloc(size);  
}
```

# Link-time Interpositioning

```
linux> make hello1
gcc -O2 -Wall -DLINKTIME -c mymalloc.c
gcc -O2 -Wall -Wl,--wrap,malloc -Wl,--wrap,free \
-o hello1 hello.c mymalloc.o
linux> make run1
./hello1
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- The “-Wl” flag passes argument to linker
- Telling linker “--wrap,malloc” tells it to resolve references in a special way:
  - Refs to `malloc` should be resolved as `__wrap_malloc`
  - Refs to `__real_malloc` should be resolved as `malloc`



# 03-exceptions

---

# Questions

2. 03-exceptions 7p를 보면 interrupt vector = exception table이라고 되어있는데 기억상으로는 exception table에는 asynchronous exception인 interrupt 말고도 다른 종류의 exception들도 들어갈 수 있는 것으로 알고있습니다. interrupt vector라고 불리게 된 이유가 있는지, 아니면 interrupt vector와 exception table의 차이가 존재하는지 궁금합니다.

# 04-signals

---

# Questions

1.03-exception에서의 interrupt와 04-signals에서의 SIGINT에 관해 궁금한게 있습니다.  
signal은 kernel에서 send된 이후 destination process에서 이를 receive하기 전까지 시간이 걸릴 수 있으며, 04-signals 28p 및 교수님 강의에서 signal은 다시 flow가 user process로 돌아올때 처리된다고 기억하고 있습니다.  
이러한 signal중에는 ctrl-c등이 눌림으로 인해 발생하는 interrupt신호를 담당하는 SIGINT라는 신호도 있는데, 만약 위에 설명대로 signal이 작동된다면 ctrl-c가 눌린 이후 바로 종료되는 것이 아니라 process context switching이 일어난 후 process에서의 receive가 진행되는 순간이 될때까지 기다린 이후에 종료되는 건가요?  
interrupt신호 말고도 SIGALRM같은 경우에도 signal을 receive하는 것을 기다린 다음 그 이후부터 1초를 측정할 경우 실제 기다리는 시간은 1초를 넘어가게 되는데 이게 옳은지 모르겠습니다.

# Signals

- A **signal** is a small message that notifies a process that an event of some type has occurred in the system
  - akin to exceptions and interrupts
  - sent from the kernel (sometimes at the request of another process) to a process
  - signal type is identified by small integer ID's (1-30)
  - only information in a signal is its ID and the fact that it arrived

<i>ID</i>	<i>Name</i>	<i>Default Action</i>	<i>Corresponding Event</i>
2	SIGINT	Terminate	Interrupt (e.g., ctrl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

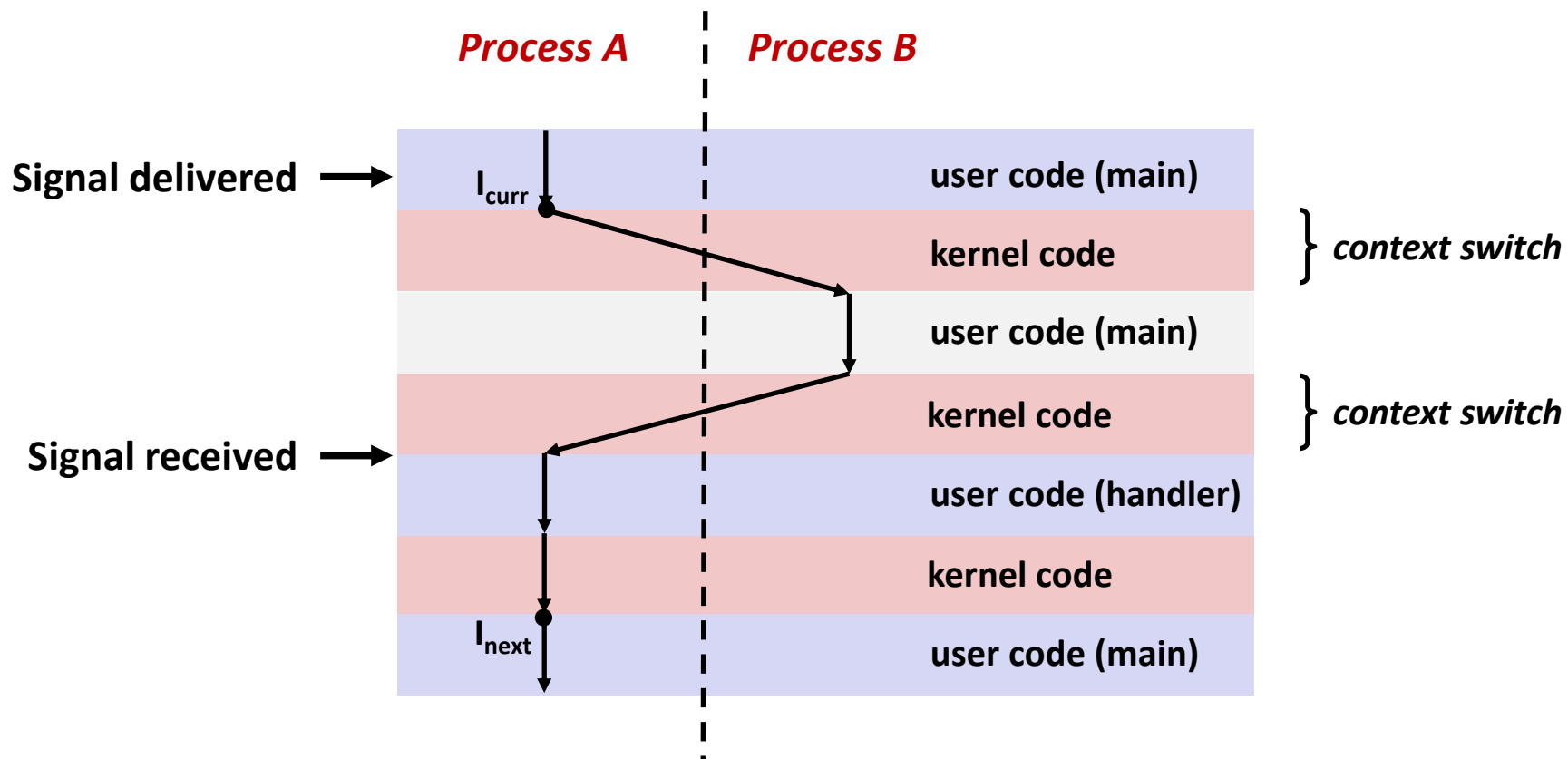
# Sending a Signal

- **Kernel** *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process
- **Kernel sends a signal for one of the following reasons:**
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - Another process has invoked the **kill** system call to explicitly request the kernel to send a signal to the destination process

# Receiving a Signal

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal
- Three possible ways to react:
  - *Ignore* the signal (do nothing)
  - *Terminate* the process (with optional core dump)
  - *Catch* the signal by executing a user-level function called *signal handler*
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt

# Another View of Signal Handlers as Concurrent Flows





# Exceptional Control Flow

- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - Exceptions
    - change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- **Higher level mechanisms**
  - Process context switch
  - Signals **Implemented by OS software**
  - Nonlocal jumps: `setjmp()/longjmp()`
  - Implemented by either:
    - OS software (context switch and signals)
    - C language runtime library (nonlocal jumps)

# Questions

3. 04-signals 31p에서 교수님 강의에서는 read 실행으로 인해 flow가 kernel로 넘어갔다가 다시 돌아올 때 signal이 pending되어있어 이를 처리하고 왔더니 read call을 했다는 사실을 까먹어서 다시 read를 실행해야 된다고 설명하셨는데, ppt에 적혀있는 대로는 read call이 실행되는 도중에 signal handler가 interrupt해서 read가 도중에 끊기게되어 결국 처음부터 다시 시작해야 한다 이렇게 이해했습니다. 어떤 설명이 맞는것인지, 또 만약 ppt의 설명이 옳다면 signal은 context switching이 일어나면서 돌아올 때 확인하는 것인데 어떻게 read 도중에 interrupt가 가능한지 궁금합니다.

이는 34p에서도 나오는데, Async-signal-safe한 함수들은 signal에 의해 interrupt가 불가능하다 되어있는데 어떻게 signal이 interrupt라는 행동을 할 수 있는지 모르겠습니다.

# Questions

4. 04-signals 36p에서 첫번째 단락 마지막 문장을 보면 "Useful for error recovery and signal handing"이라고 되어 있는데, 에러가 났을 때 longjmp를 통해 돌아가버리면 에러를 없던 것처럼 할 수 있다는 점에서 error recovery라는 부분은 이해했지만 signal handing에서는 어떤 식으로 사용되는지 궁금합니다.