Chapter 8. Type Confusion and Race Condition

Prof. Jaeseung Choi

Dept. of Computer Science and Engineering

Sogang University

Topics

- **Type Confusion**
  - General concept of type confusion
  - Background on type casting in C/C++
  - Common patterns of type confusion
- **Race condition**
  - Review of process and thread
  - Time-of-check to Time-of-use (TOCTOU)
  - Common patterns of race condition

Type Confusion

- Mistaking a memory location for certain type as a memory for different type
- Consider the example below with two structure types
  - Pointer `p1` for `struct S` and pointer `p2` for struct `T`

```
struct S *p1 =
malloc(sizeof(S))
```

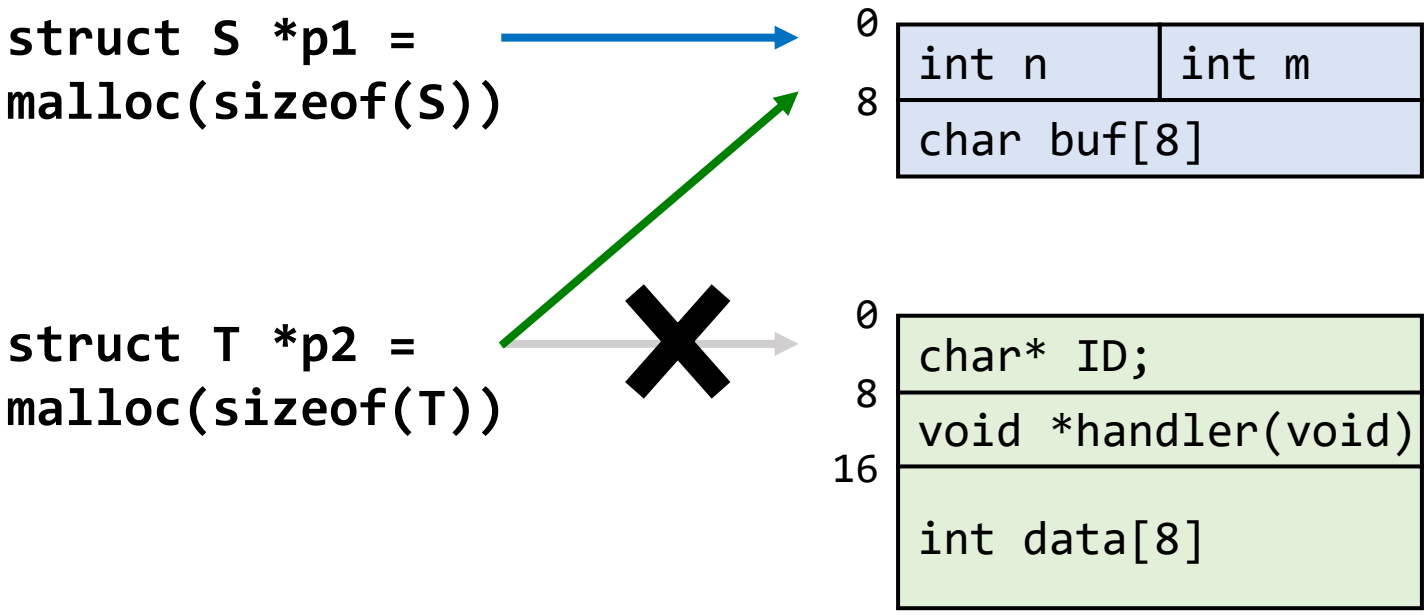
int n	int m
char buf[8]	

```
struct T *p2 =
malloc(sizeof(T))
```

char* ID;
void *handler(void)
int data[8]

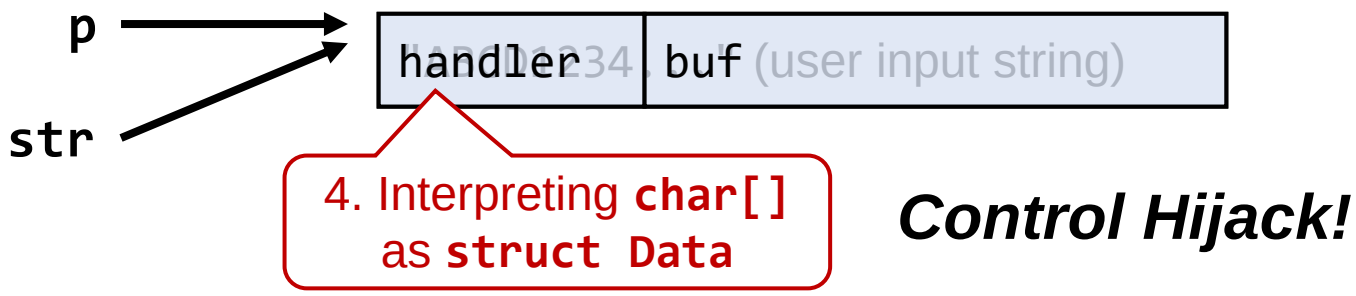
Type Confusion

- **What if we `p2` was actually pointing at `struct S`?**
  - Accessing `ID` will try to interpret **two integers** as a pointer
  - Using `handler` will interpret **characters** as a function pointer
  - Printing `data[]` will disclose the **memory beyond `struct S`**



Review: Use-After-Free

- **Recall the exploitation of UAF in the previous chapter**
- **We interpreted `char[]` as a `struct` with function pointer**
  - This allowed the attacker to perform code execution
- **As this example shows, use-after-free is a common source for type confusion**
  - But there are also other causes for type confusion



Background: Type Casting in C

- **C programs often *simulate* inheritance by declaring a `struct` that contains common fields shared by structs**
  - **`packet*`** pointer can point at both **`packet_A`** and **`packet_B`**
  - Based on **`kind`** field, cast it to either **`packet_A*`** or **`packet_B*`**

```
struct packet {
  int kind;
};
```

This field records whether the packet is A type or B type

```
struct packet_A {
  int kind;
  char header[4];
  int data[8];
};
```

```
struct packet_B {
  int kind;
  char header[8];
  uint32_t len;
  ...
};
```

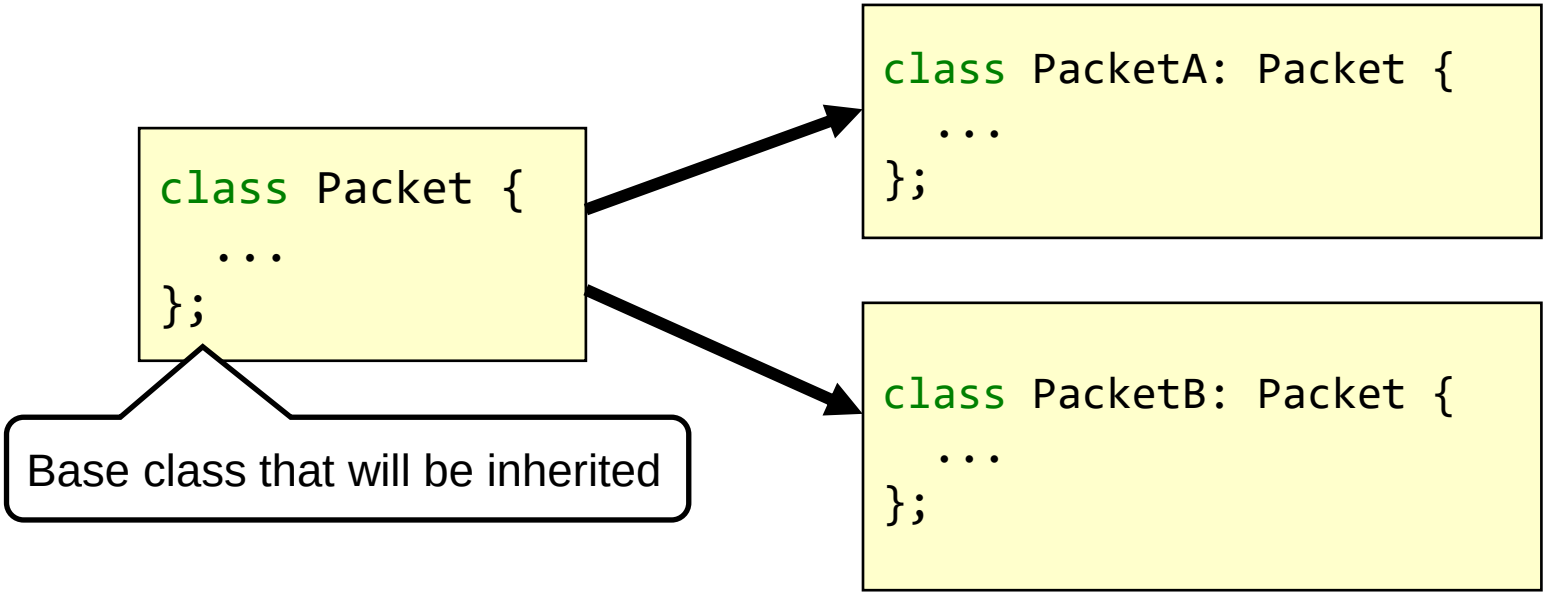
Background: Type Casting in C

- **C programs often *simulate* inheritance by declaring a `struct` that contains common fields shared by structs**
  - **`packet*`** pointer can point at both **`packet_A`** and **`packet_B`**
  - Based on **`kind`** field, cast it to either **`packet_A*`** or **`packet_B*`**

```
void f(struct packet *p) {
  if (p->kind == 1) {
    struct packet_A *pa = (struct packet_A*) p;
    // Process as packet_A type
    ...
  } else (p->kind == 2) {
     struct packet_B *pb = (struct packet_B*) p;
    // Process as packet_B type
    ...
  }
};
```

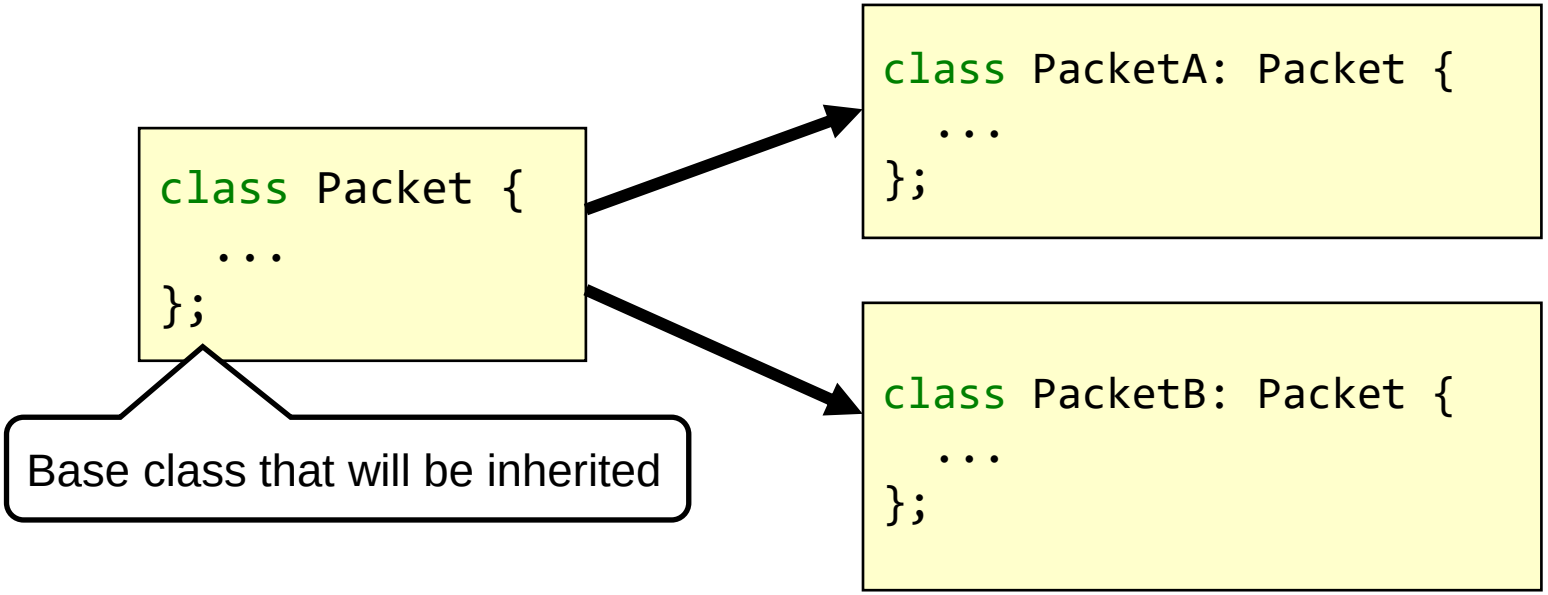
Background: Type Casting in C++

- **C++ supports inheritance, but casting is still possible**
  - ***Downcasting:*** casting a pointer for base class object (**`Packet`**) into a pointer for derived class object
  - Such downcasting can be dangerous, just as in the C language



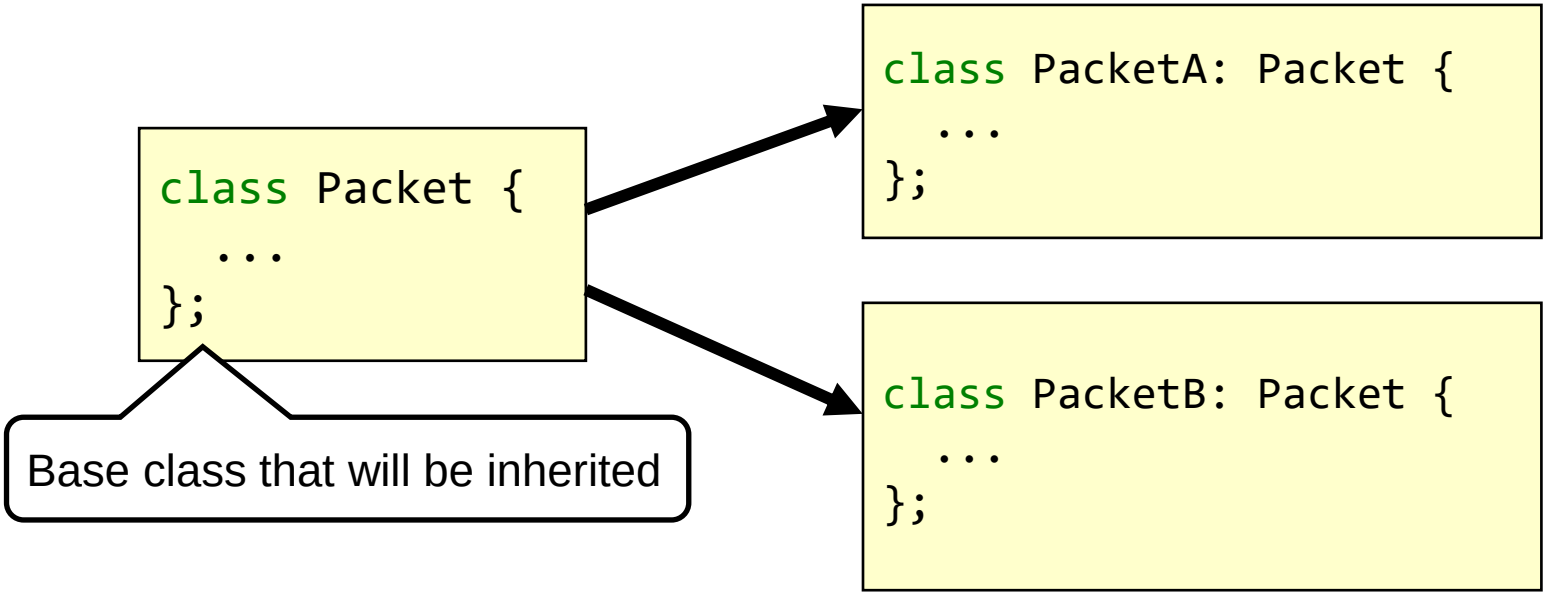
Background: Type Casting in C++

- **Downcasting with static casting**
  - Ex) `PacketA *pa = static_cast<PacketA*>(p);`
  - This converts `p` into `PacketA*` pointer without any check
  - So use this only when you are perfectly sure about `p`'s type



Background: Type Casting in C++

- **Downcasting with dynamic casting (relatively slow)**
  - Ex) `PacketA *pa = dynamic_cast<PacketA*>(p);`
  - This converts **p** into **PacketA*** only after checking **p** indeed points to an object of **PacketA** class (if not, returns **NULL**)
  - Compiler automatically inserts information to enable such check



Type Confusion from Logic Error

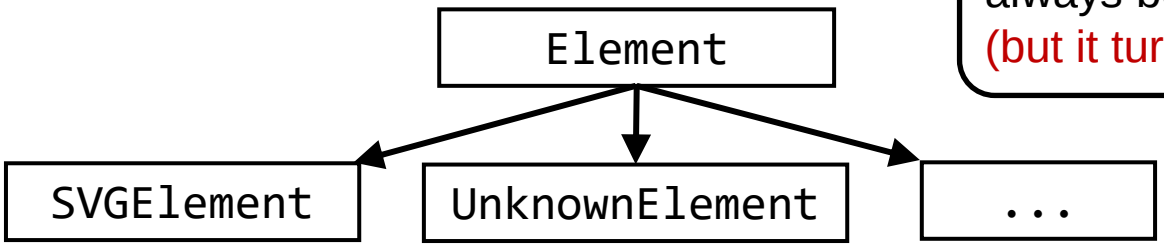
- **Type confusion can occurs from various reasons, but one notable cause is *logical error***
- **Recall that `static_cast` must be used only if the programmer is perfectly sure about the actual type**
  - Ex) `PacketA *pa = static_cast<PacketA*>(p);`
- **But the programmers sometimes make a mistake**
  - Ex) The programmer thinks "If this global variable is set to **`true`** when my function is executed, **`p`** must be pointing at **`PacketA`**", and uses **`static_cast`**
  - But what if it turns out that **`p`** could point at **`PacketB`** in certain corner cases?

Real-world Example

- **Type Confusion* in *WebKit* (Web browser engine)**

```
SVGElement* SVGViewSpec::viewTarget() {
  if (!m_contextElement)
    return 0;
  return static_cast<SVGElement*>(
    m_contextElement->treeScope()->getElementById(
      m_viewTargetString
    )
  );
}
```

Developer thought this must always be `SVGElement` type (but it turned out to be not!)



Acknowledgement: Example from KAIST **IS561** lecture note

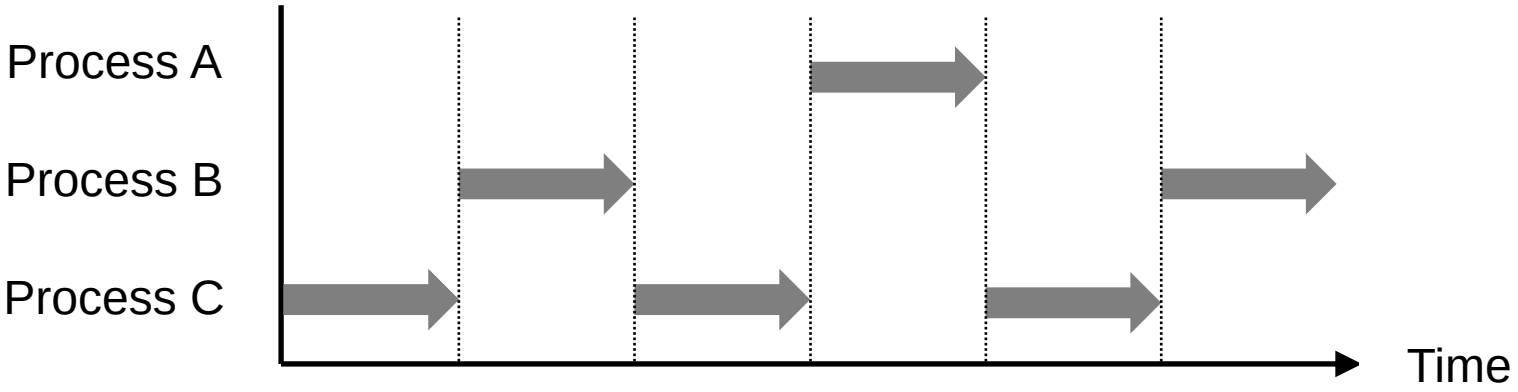
*CVE-2013-0912

Topics

- **Type Confusion**
  - General concept of type confusion
  - Background on type casting in C/C++
  - Common patterns of type confusion
- **Race condition**
  - Review of process and thread
  - Time-of-check to Time-of-use (TOCTOU)
  - Common patterns of race condition

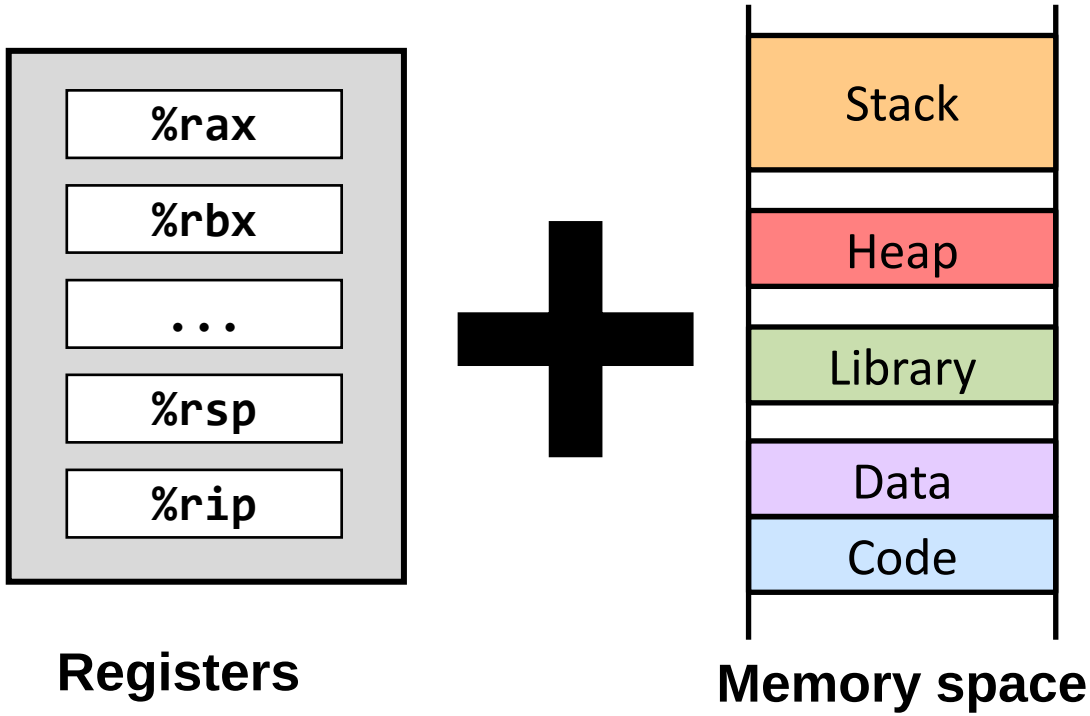
Multiple Execution Flow

- **So far, we have only considered linear flow of execution**
- **However, usually there are multiple processes running simultaneously in a system**
  - Process: an instance of program that is executing
- **Moreover, there can be multiple threads running simultaneously in one program (process)**
  - Thread: a smaller unit of execution within process



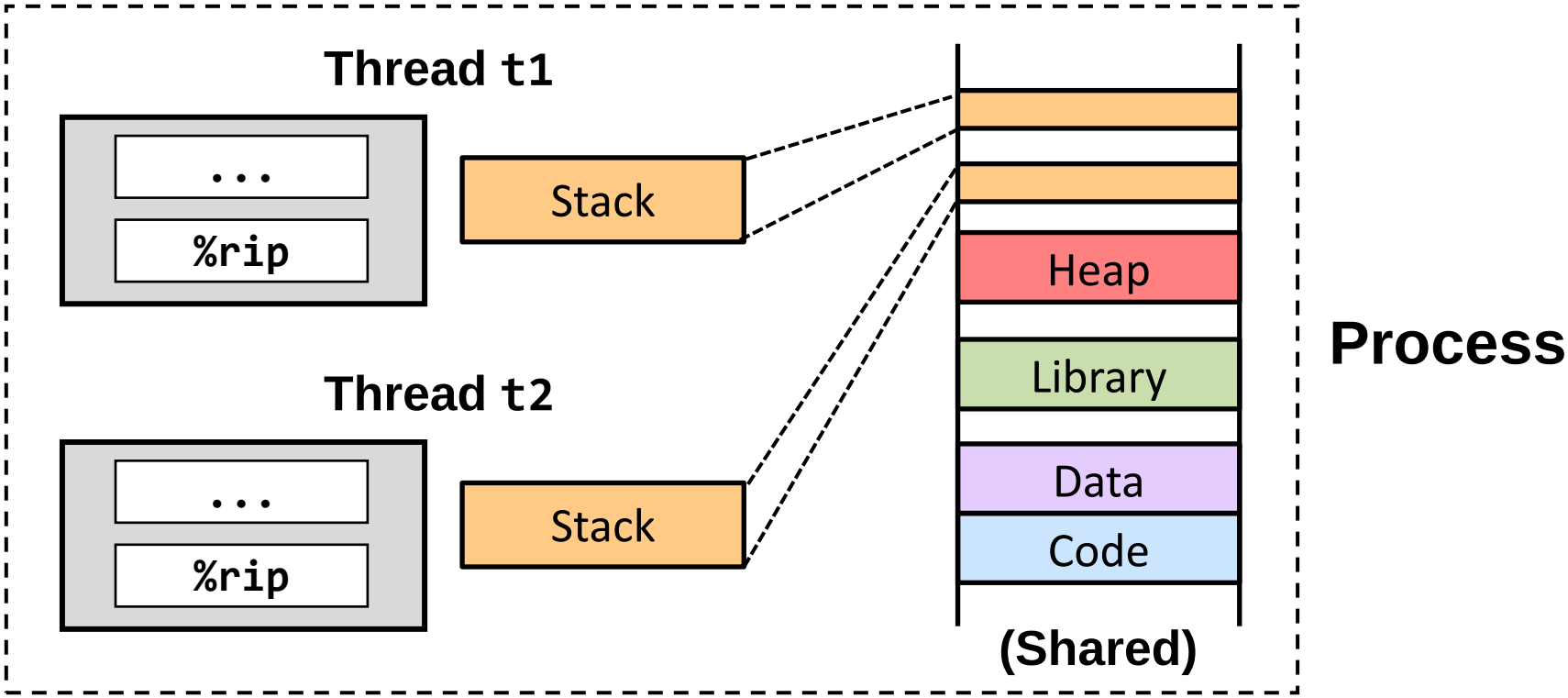
Review: Process

- **How can we model a process (running program)?**
- **Process state = registers + memory + α**
  - Note that each process has isolated memory space



Review: Thread

- Multiple threads in one process share the memory space
- But each thread has its own logical control flow and dedicated stack area



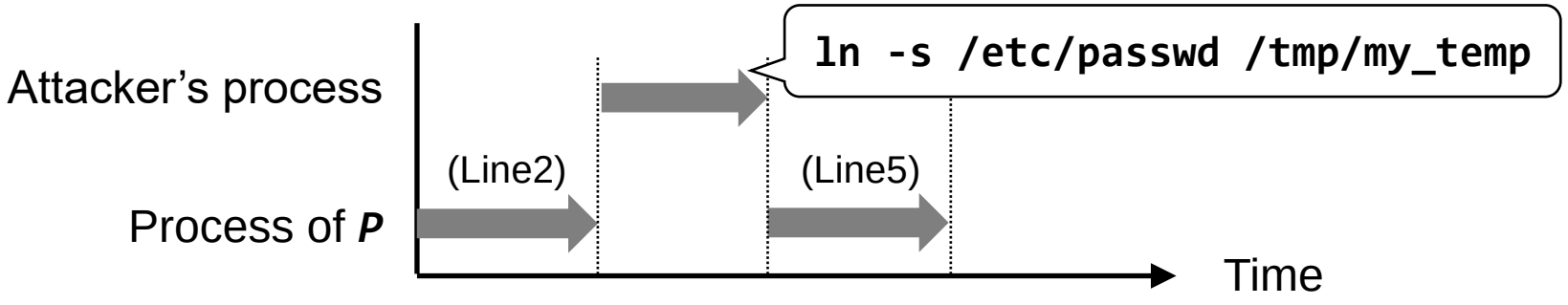
Race Condition

- **Broad meaning: Situation where the execution result depends on the timing and order of threads/processes**
- **In security: Critical misbehavior of a program caused by unexpected *interleaving* of threads/processes**
- **One popular type of race condition vulnerability is *time-of-check to time-of-use (TOCTOU)***
  - **Time-of-check:** program checks if it is safe to use/do something
  - **Time-of-use:** program actually performs that action
  - Between these two, another process or thread may kick in and change the status - then it is not safe anymore!

Race Condition Example (1)

- **Assume a program executed with `root` privilege: it will write a file in the temporary directory (`/tmp`) as below**
  - What if an attacker creates a ***symbolic link*** between the **time-of-check (Line 2)** and **time-of-use (Line 5)**?

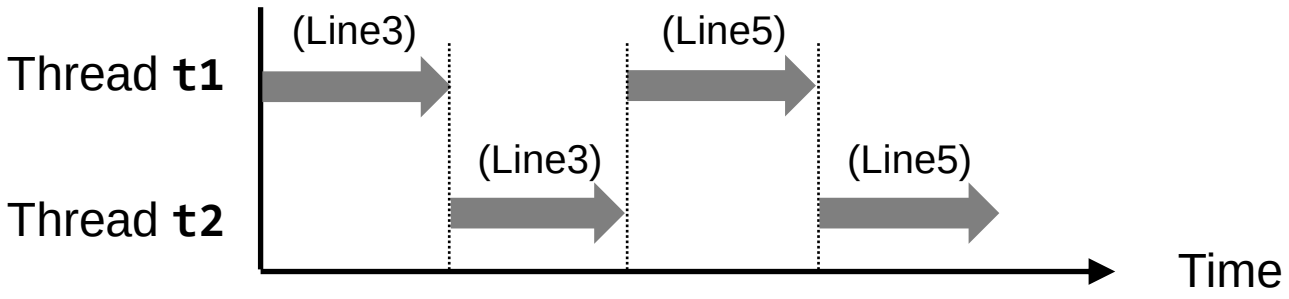
```
1  // Pseudo-code of program P
2  if ("/tmp/my_temp" already exists) {
3    exit(1);
4  } else {
5    Open "/tmp/my_temp" and write some data there
6  }
```



Race Condition Example (2)

- **Assume a multi-threaded program for banking**
  - **`transfer()`** checks if the balance is enough before sending
  - Assume there are multiple threads that can execute **`transfer()`**
  - What if two threads are interleaved in the following way?

```
1  void transfer(int sender_id, int receiver_id, int amount) {
2    if (amount < 0) return;
3    if (balance[sender_id] < amount) return;
4    ... // Update receiver's balance accordingly
5    balance[sender_id] -= amount;
6  }
```



Mutex: Not a Silver Bullet

- **Certain race conditions can be prevented with mutex**
  - For example, the race condition in the previous page
- **But misuse of mutex raises another problem: *deadlock***
- **For example, locking a mutex from within a signal handler may result in a deadlock**
  - Signal handler stops the original (main) execution flow until the handler code is finished