

Operating Systems

Misc

Dr. Young-Woo Kwon

Announcements

- 기말시험
 - 6.19 (IT5 245), 10:30AM ~ 11:45AM
 - 시험범위
 - Memory paging ~ Filesystem

Approach 2: Journaling

- Problem: *fsck* is slow because it checks the entire file system after a crash
 - What if we knew where the last writes were before the crash, and just checked those?
- Key idea: make writes transactional by using a **write-ahead log**
 - Commonly referred to as a **journal**
- **Ext3 and NTFS use journaling**



Write-Ahead Log

- Key idea: **writes to disk are first written into a log**
 - After the log is written, the writes execute normally
 - In essence, **the log records transactions**
- What happens after a crash...
 - **If the writes to the log are interrupted?**
 - The transaction is incomplete
 - The user's data is lost, but the file system is consistent
 - **If the writes to the log succeed, but the normal writes are interrupted?**
 - The file system may be inconsistent, but...
 - The log has exactly **the right information to fix the problem**

Data Journaling Example

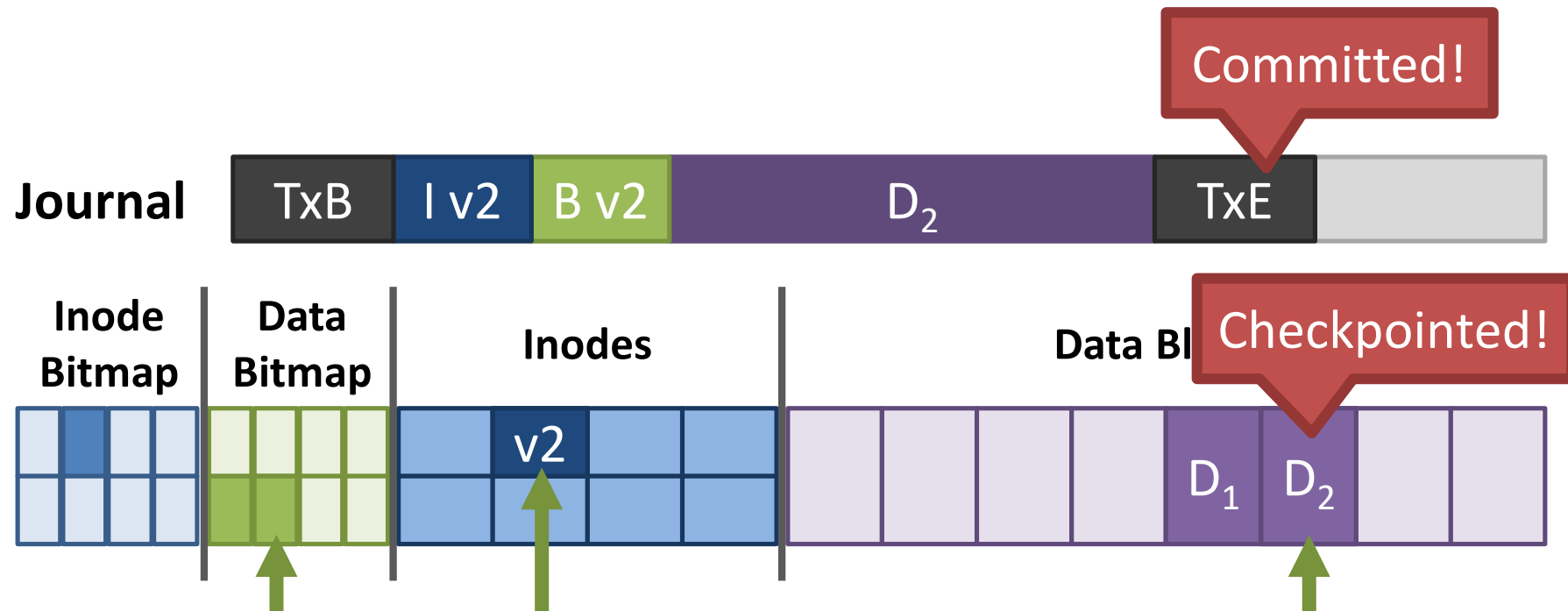
- Assume we are appending to a file
 - Three writes: inode v2, data bitmap v2, data D_2
- Before executing these writes, first log them



1. Begin a new transaction with a unique $ID=k$
2. Write the updated meta-data block(s)
3. Write the file data block(s)
4. Write an end-of-transaction with $ID=k$

Commits and Checkpoints

- A transaction is committed after all writes to the log are complete
- After a transaction is committed, the OS checkpoints the update

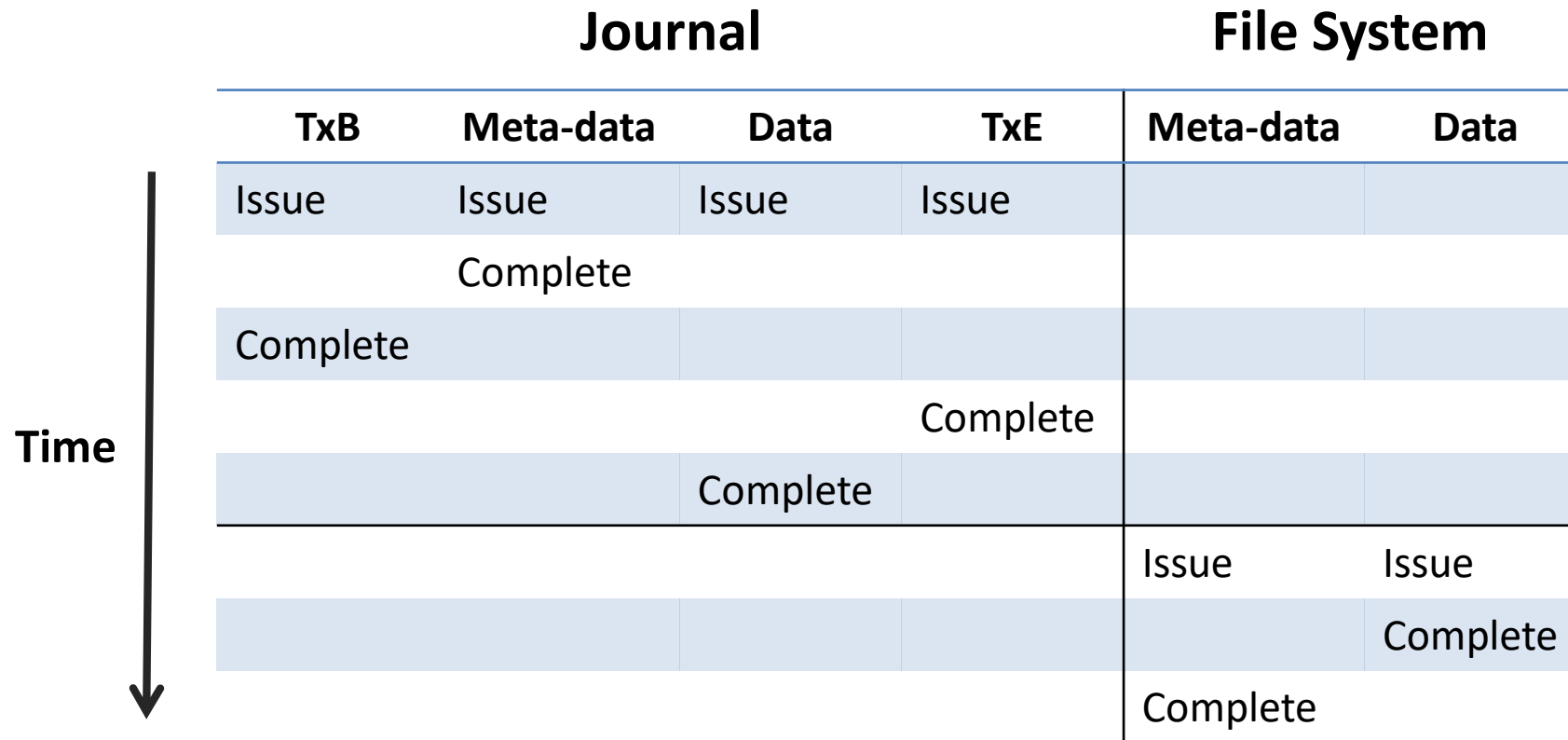


- Final step: **free** the checkpointed transaction

Journal Implementation

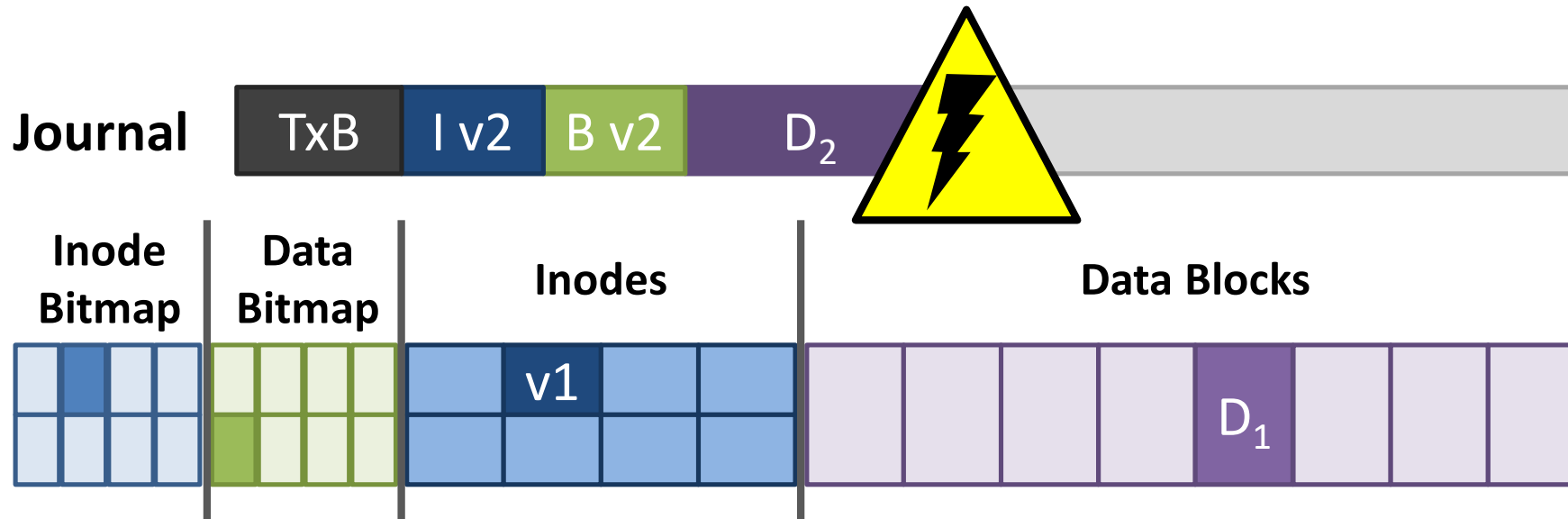
- Journals are typically implemented as a circular buffer
 - Journal is **append-only**
- OS maintains pointers to the front and back of the transactions in the buffer
 - As transactions are freed, the back is moved up
- Thus, the contents of the journal are never deleted, they are just overwritten over time

Data Journaling Timeline



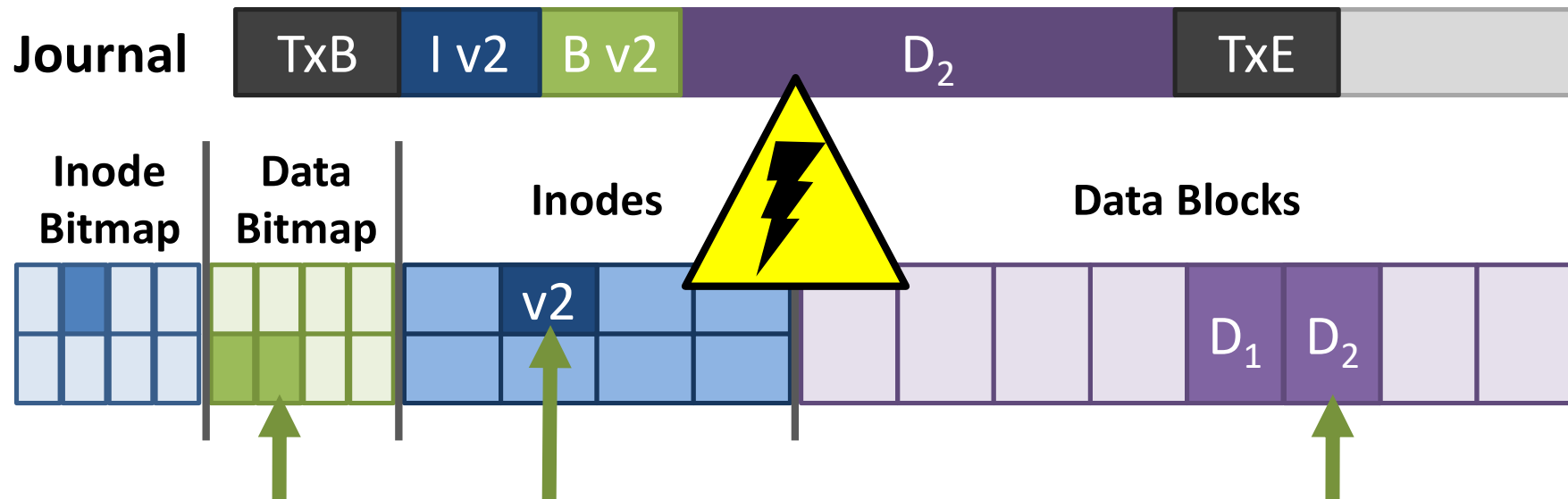
Crash Recovery (1)

- What if the system crashes during logging?
 - If the transaction is not committed, data is lost
 - But, the file system remains consistent



Crash Recovery (2)

- What if the system crashes during the checkpoint?
 - File system may be inconsistent
 - During reboot, transactions that are committed but not free are replayed in order
 - Thus, no data is lost and consistency is restored



Corrupted Transactions

- Problem: the disk scheduler may not execute writes in-order
 - Transactions in the log may appear committed, when in fact they are invalid

Journal



- Solution: add a checksum to TxB
- During recovery, reject transactions with invalid checksums
- Implemented on Linux in ext4

- Transaction looks valid, but the data is missing!
- During replay, garbage data is written to the file system

Journaling: The Good and the Bad

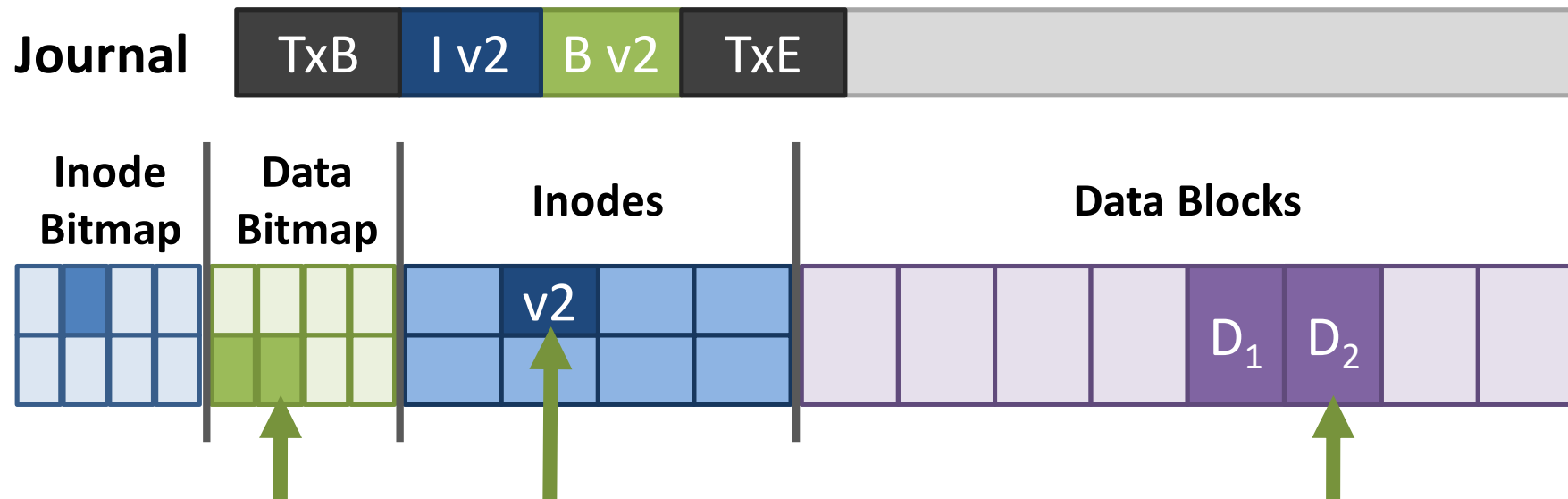
- Advantages of journaling
 - Robust, fast file system recovery
 - No need to scan the entire journal or file system
 - Relatively straight forward to implement
- Disadvantages of journaling
 - Write traffic to the disk is doubled
 - Especially the file data, which is probably large
 - Deletes are very hard to correctly log
 - Example in a few slides...

Making Journaling Faster

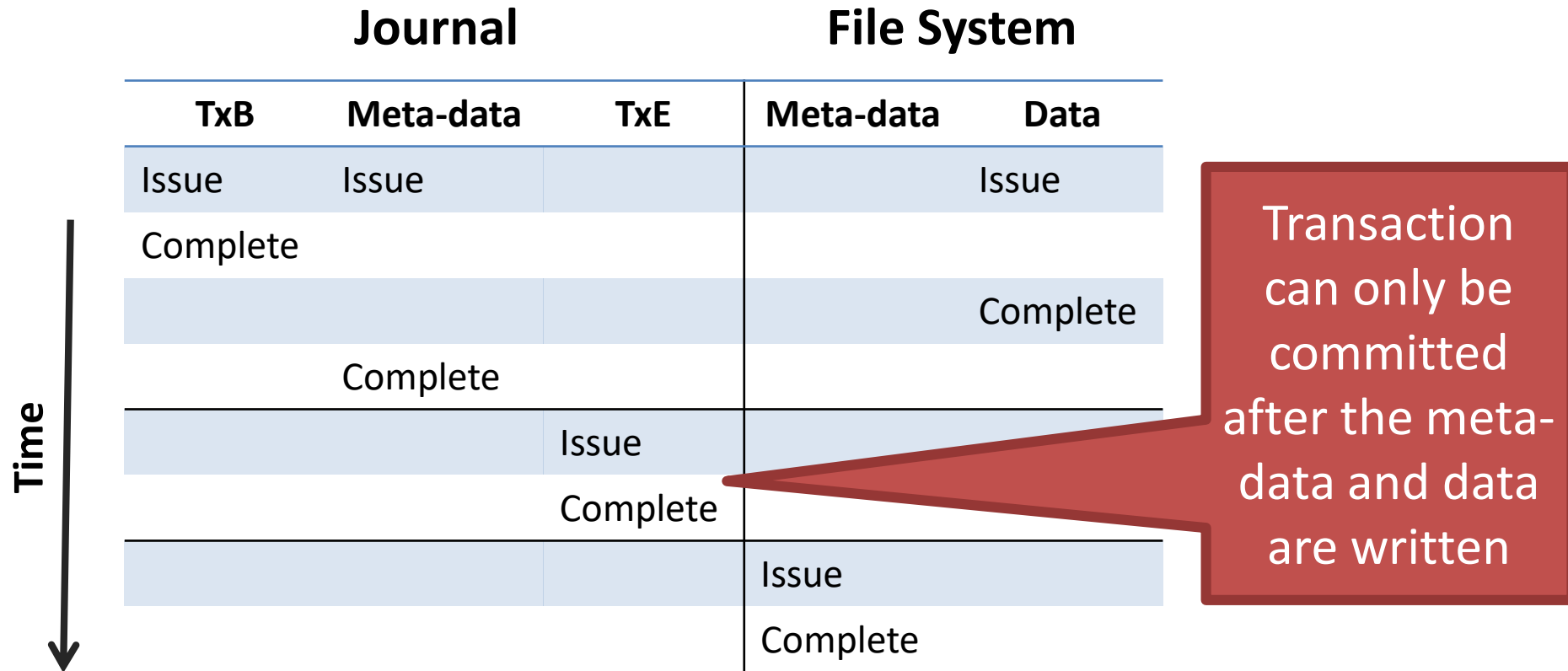
- Journaling adds a lot of write overhead
- OSes typically **batch updates to the journal**
 - Buffer sequential writes in memory, then issue one large write to the log
 - Example: `ext3` batches updates for 5 seconds
- **Tradeoff between performance and persistence**
 - Long batch interval = fewer, larger writes to the log
 - Improved performance due to large sequential writes
 - But, **if there is a crash, everything in the buffer will be lost**

Meta-Data Journaling

- The most expensive part of data journaling is writing the file data twice
 - Meta-data is small (~1 sector), file data is large
- ext3 implements meta-data journaling

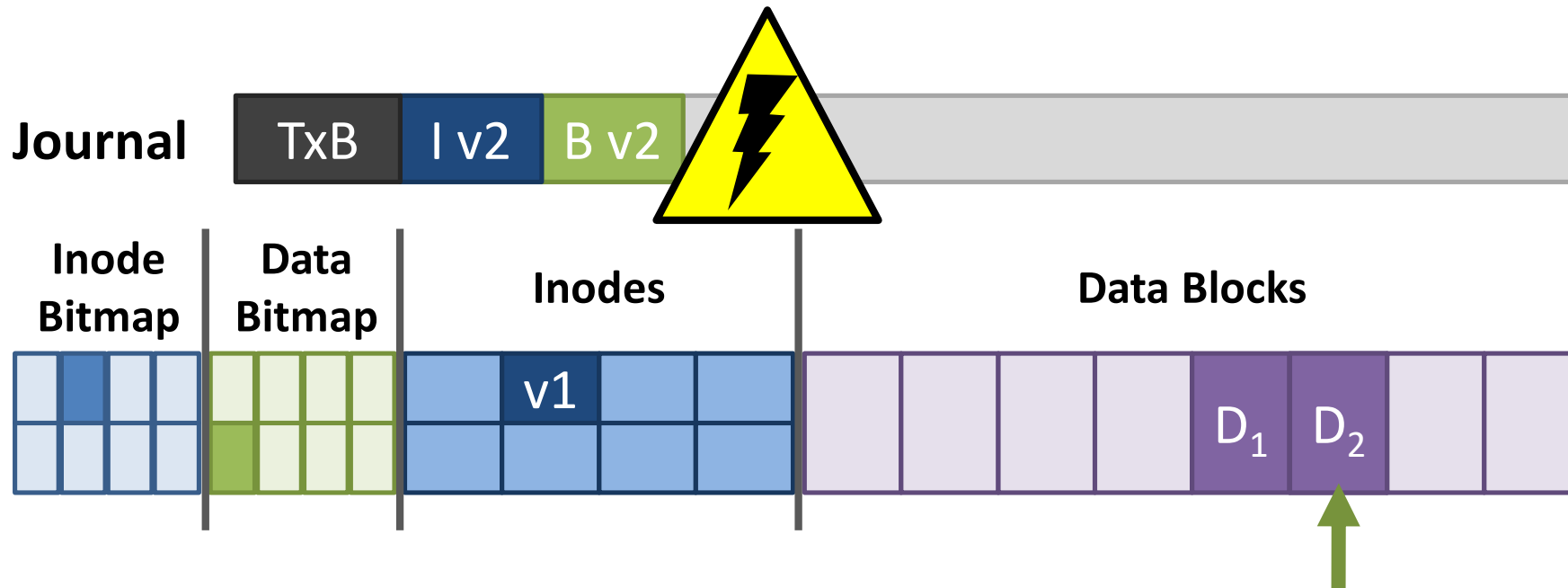


Meta-Journaling Timeline



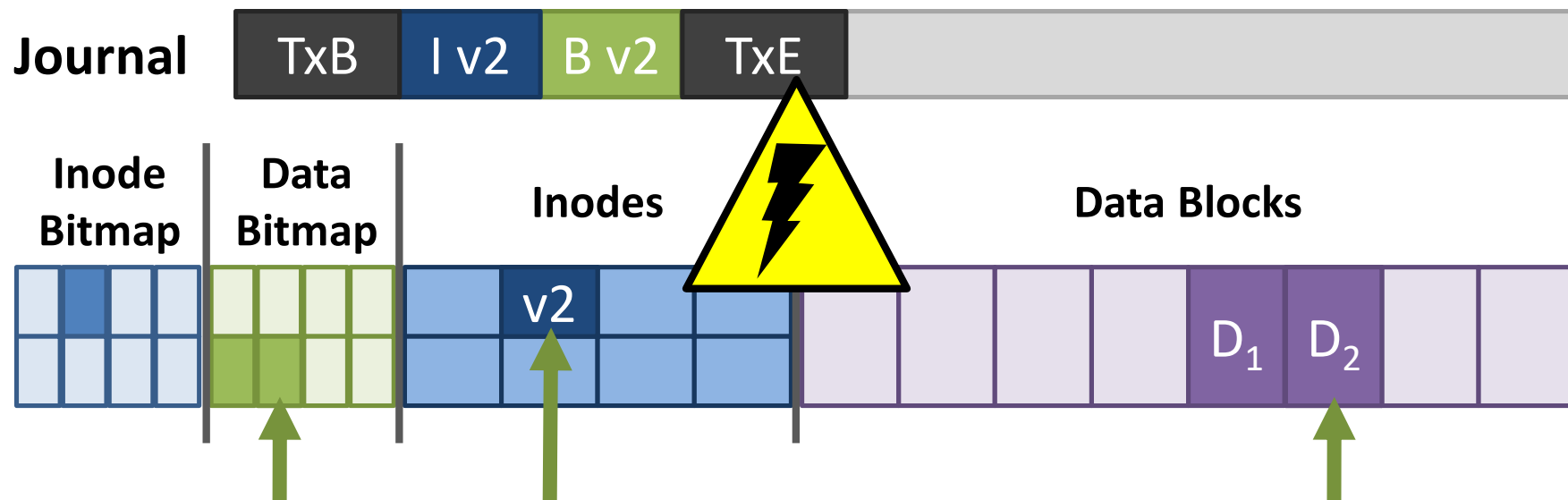
Crash Recovery (1)

- What if the system crashes during logging?
 - **If the transaction is not committed, data is lost**
 - D_2 will eventually be overwritten
 - The file system remains consistent

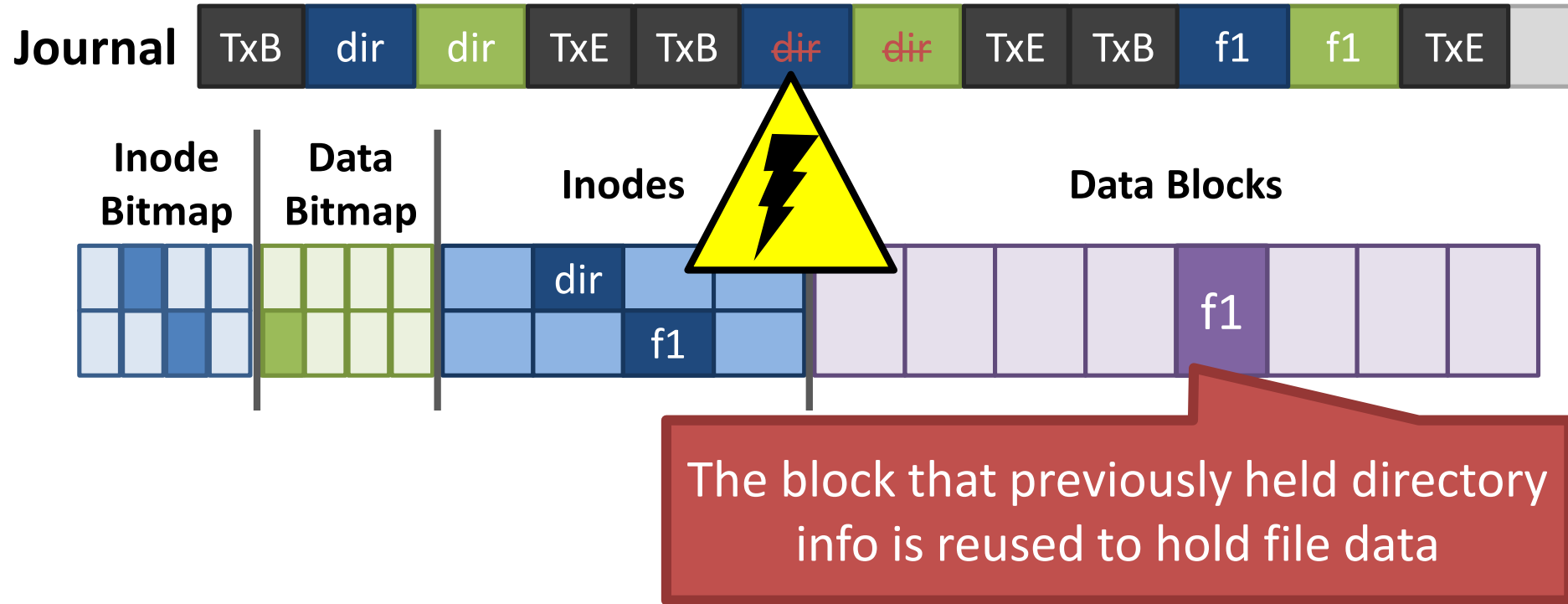


Crash Recovery (2)

- What if the system crashes during the checkpoint?
 - File system may be inconsistent
 - **During reboot, transactions that are committed** but not free are replayed in order
 - Thus, no data is lost and consistency is restored



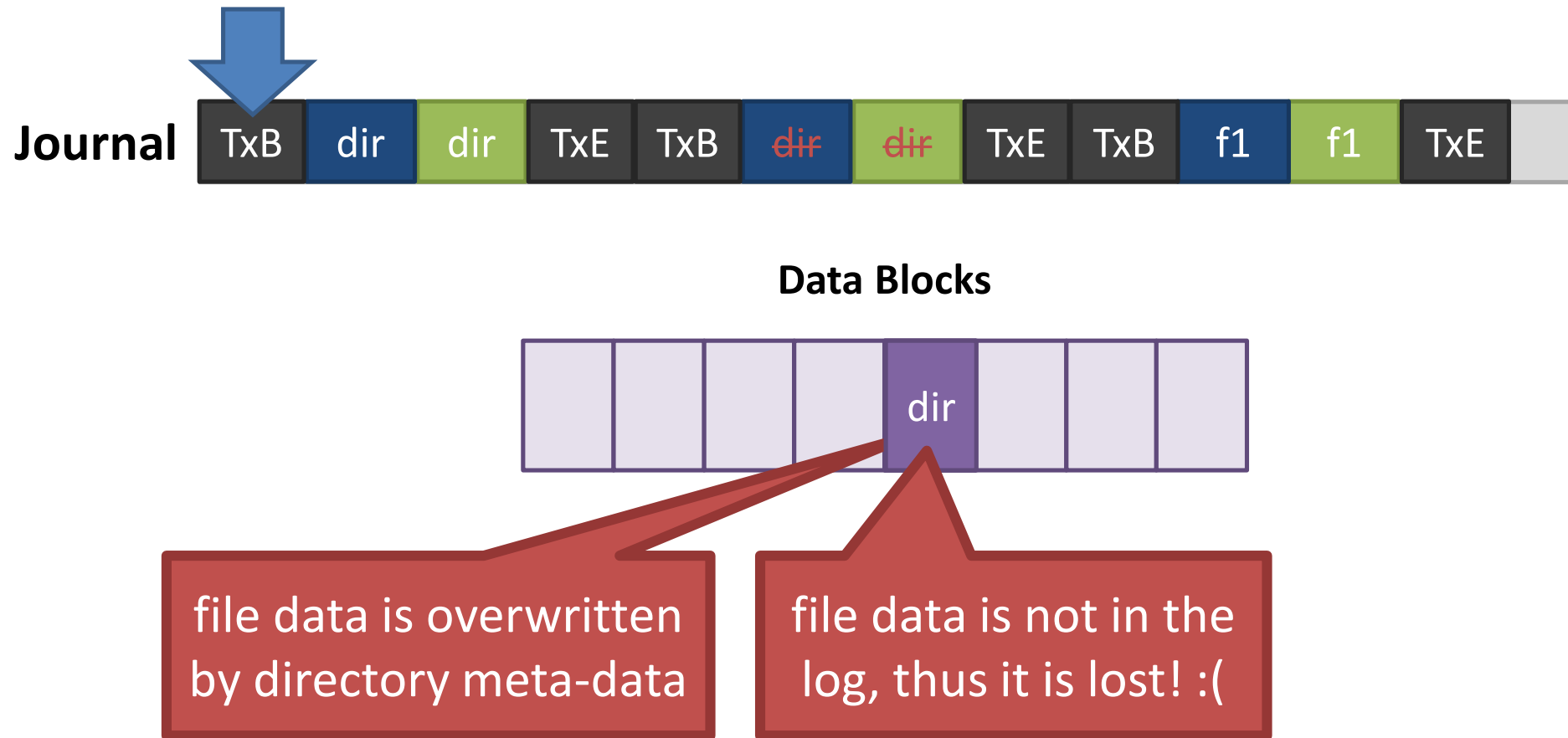
Delete and Block Reuse



1. Create a directory: inode and data are written
2. Delete the directory: inode is removed
3. Create a file: inode and data are written

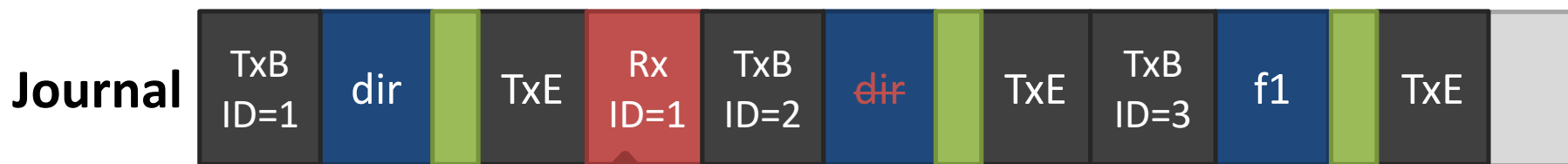
The Trouble With Delete

- What happens when the log is replayed?



Handling Delete

- Strategy 1: don't reuse blocks until the delete is checkpointed and freed
- Strategy 2: add a **revoke** record to the log
 - **ext3** used revoke records



If the log is replayed,
ignore transaction ID=1

Journaling Wrap-Up

- Today, most OSes use journaling file systems
 - ext3/ext4 on Linux
 - NTFS on Windows
- Provides excellent crash recovery with relatively low space and performance overhead
- Next-gen OSes will likely move to file systems with copy-on-write semantics
 - btrfs and zfs on Linux

THE BYZANTINE GENERALS PROBLEM




Failure Models

- **Crash failure:** ceases to execute
 - Permanent
 - Cause:, e.g., power loss
 - Variant: dead for finite period of time then resumes
- **Omission failure:** process or communication channel fails to perform actions that it is supposed to do.
 - Communication omission failure: sender sends a sequence of messages but receiver does not receive some subset of messages
 - Cause: e.g., interference in medium
 - Process omission failures: crash failure
- **Timing failures**
 - Messages do not arrive in time, computation takes longer then expected times
 - Cause: e.g., congestion, over-loading, garbage-collection

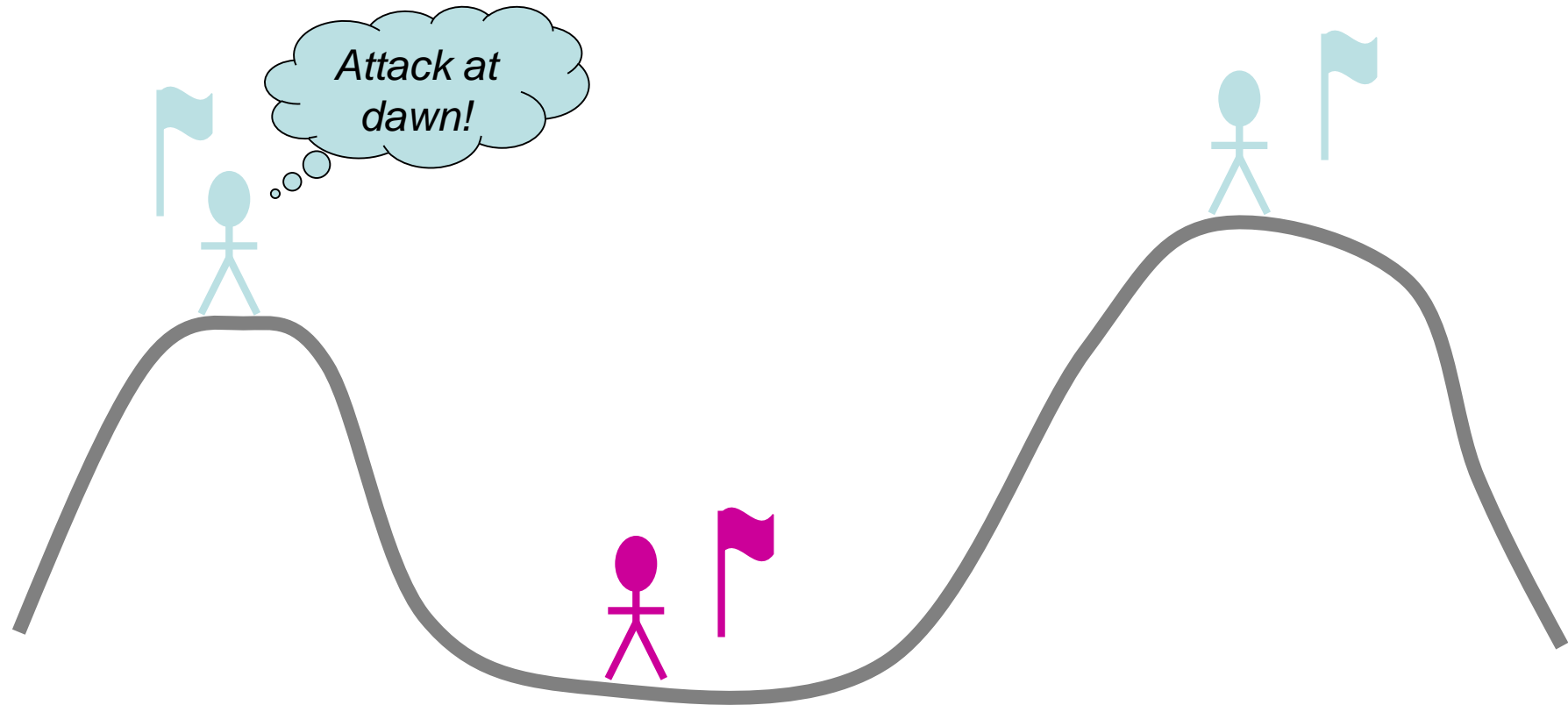
Failure Models

- **Transient failure**: process jumps to arbitrary state and resumes normal execution
 - Cause: e.g., gamma rays
- **Byzantine failure**: arbitrary messages and transitions
 - Cause: e.g., software bugs, malicious attacks

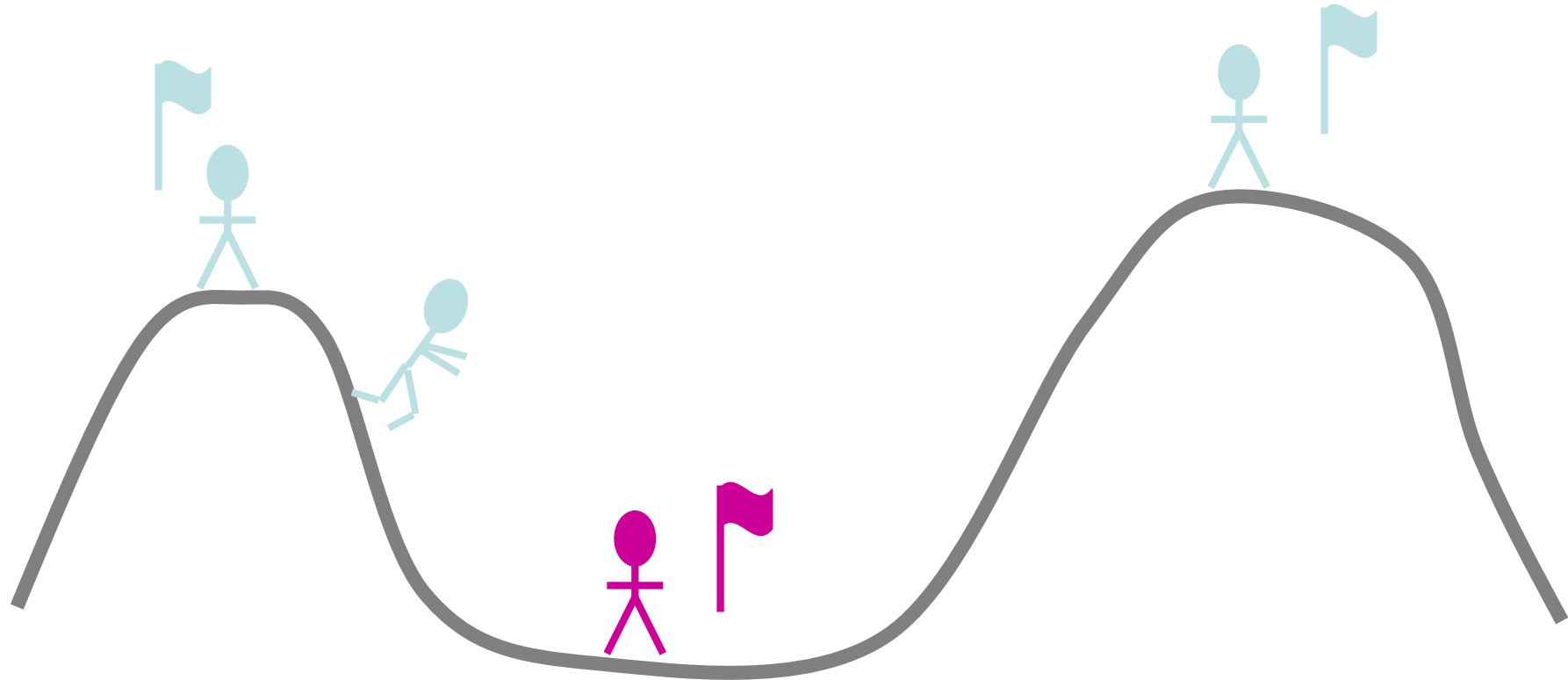
A simple distributed coordination problem

- Two  armies are camped on separate hills surrounding a valley in which the enemy army  is encamped.
- The enemy can vanquish each army separately, but
- if both  armies attack at the same time, they can vanquish the enemy.
- Two generals synchronize through messengers
- Messengers can be lost

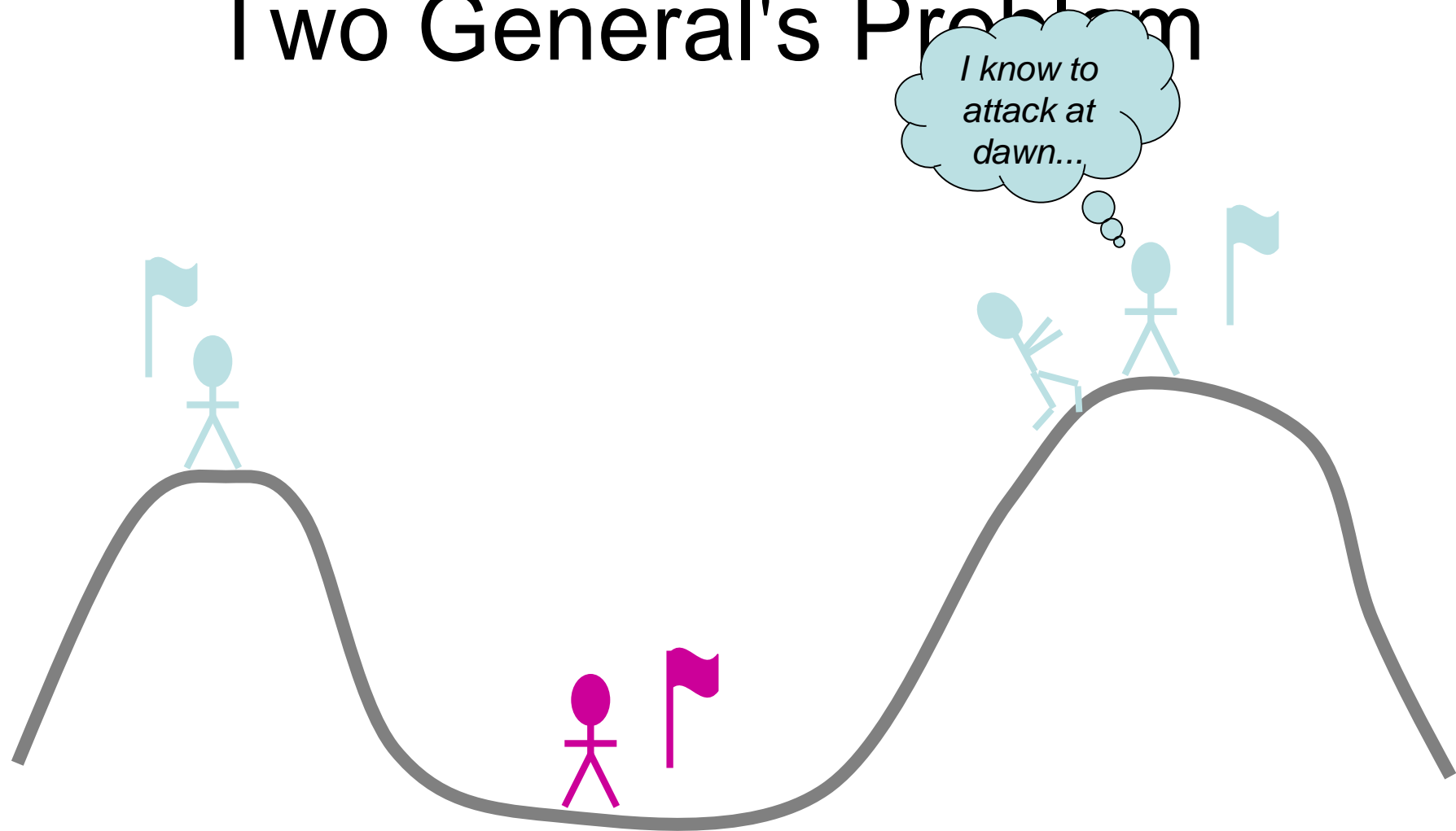
Two General's Problem



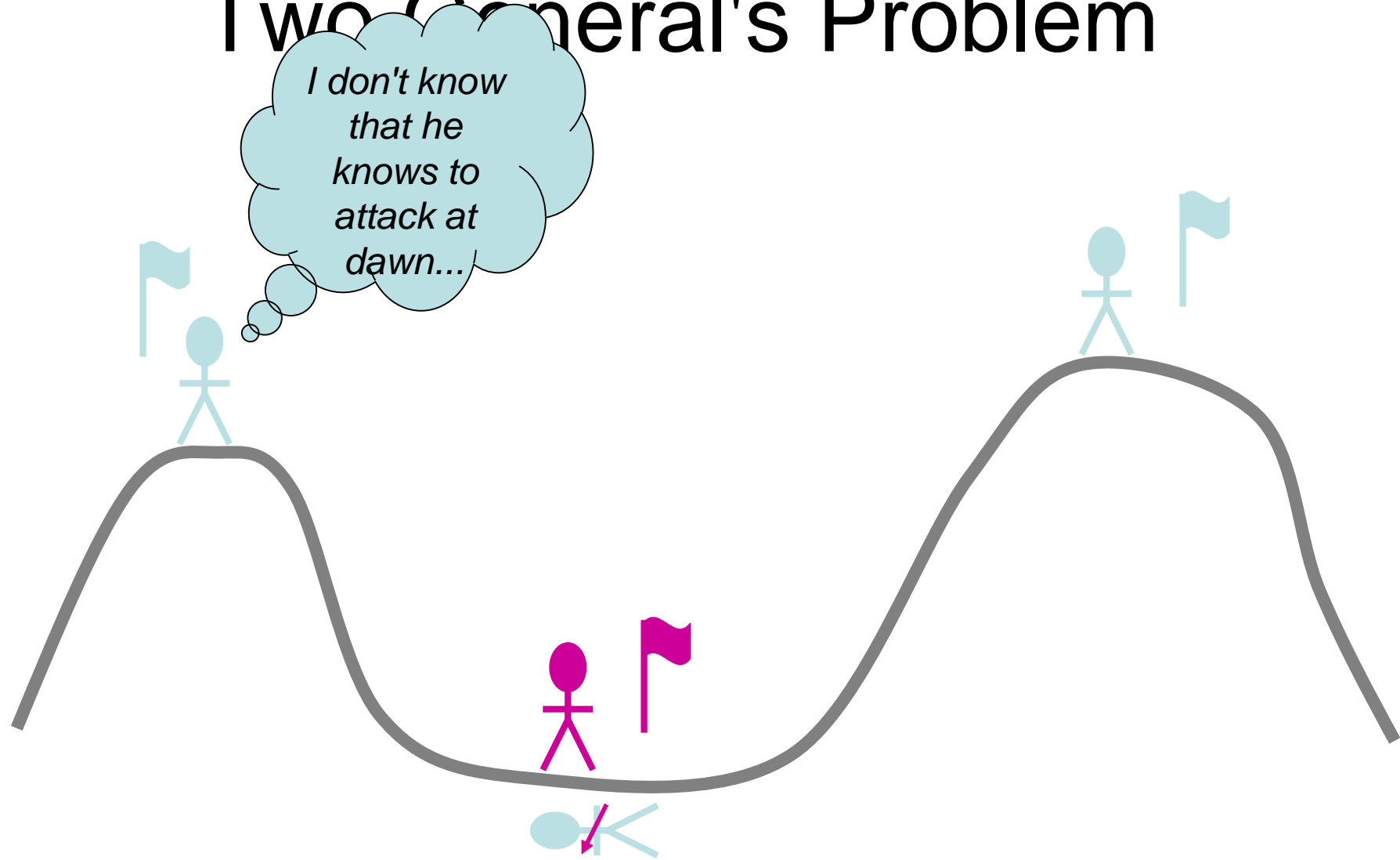
Two General's Problem



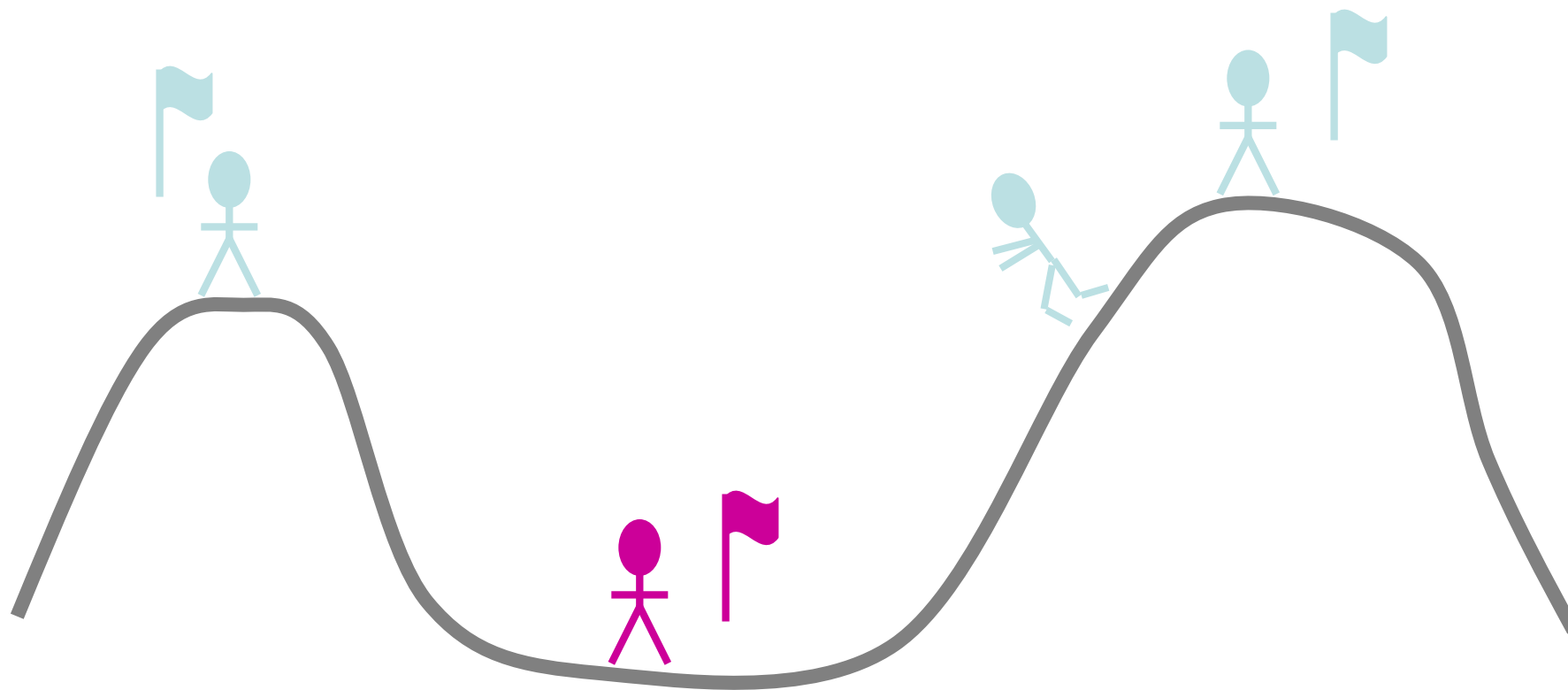
Two General's Problem



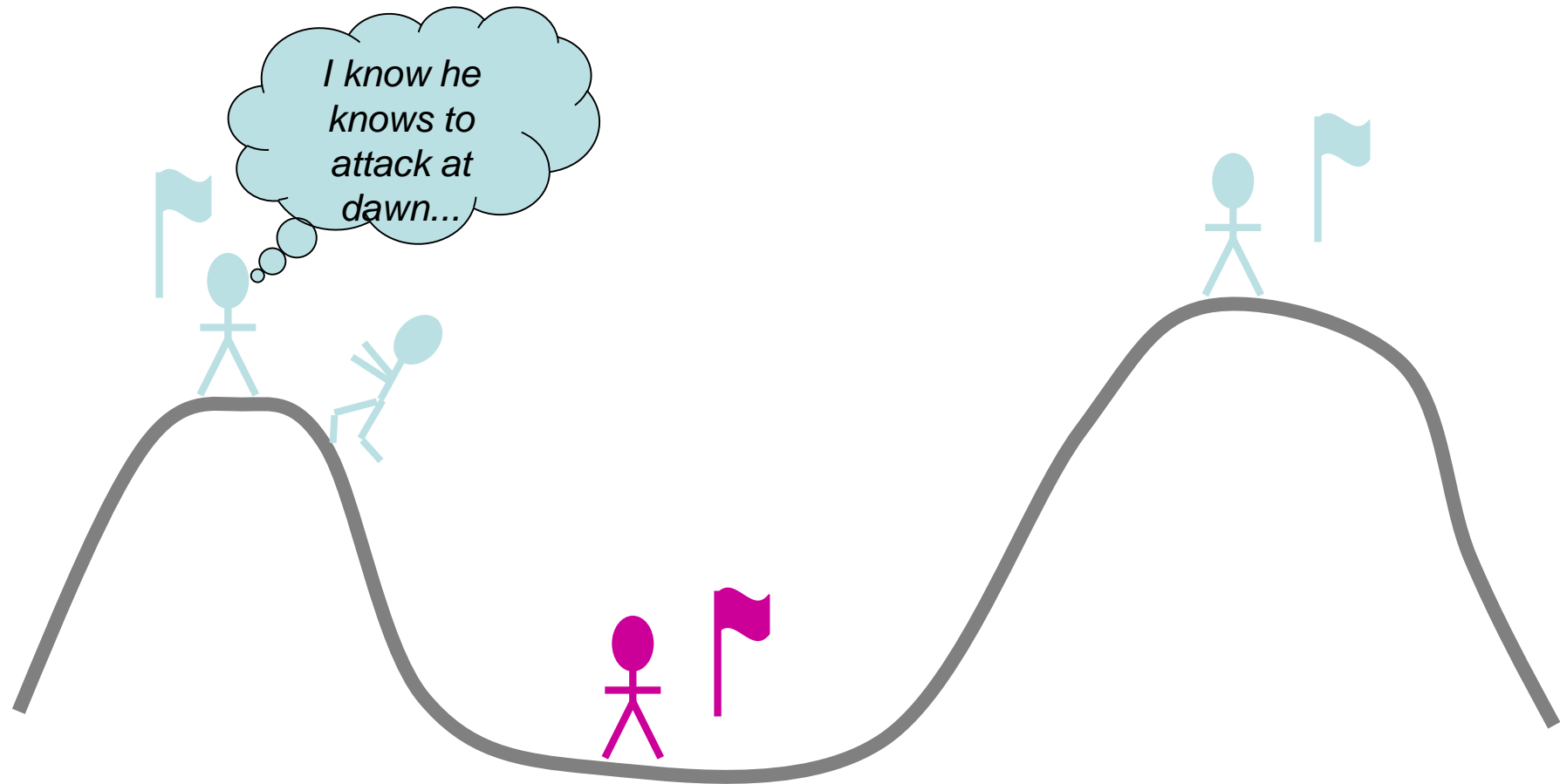
Two General's Problem



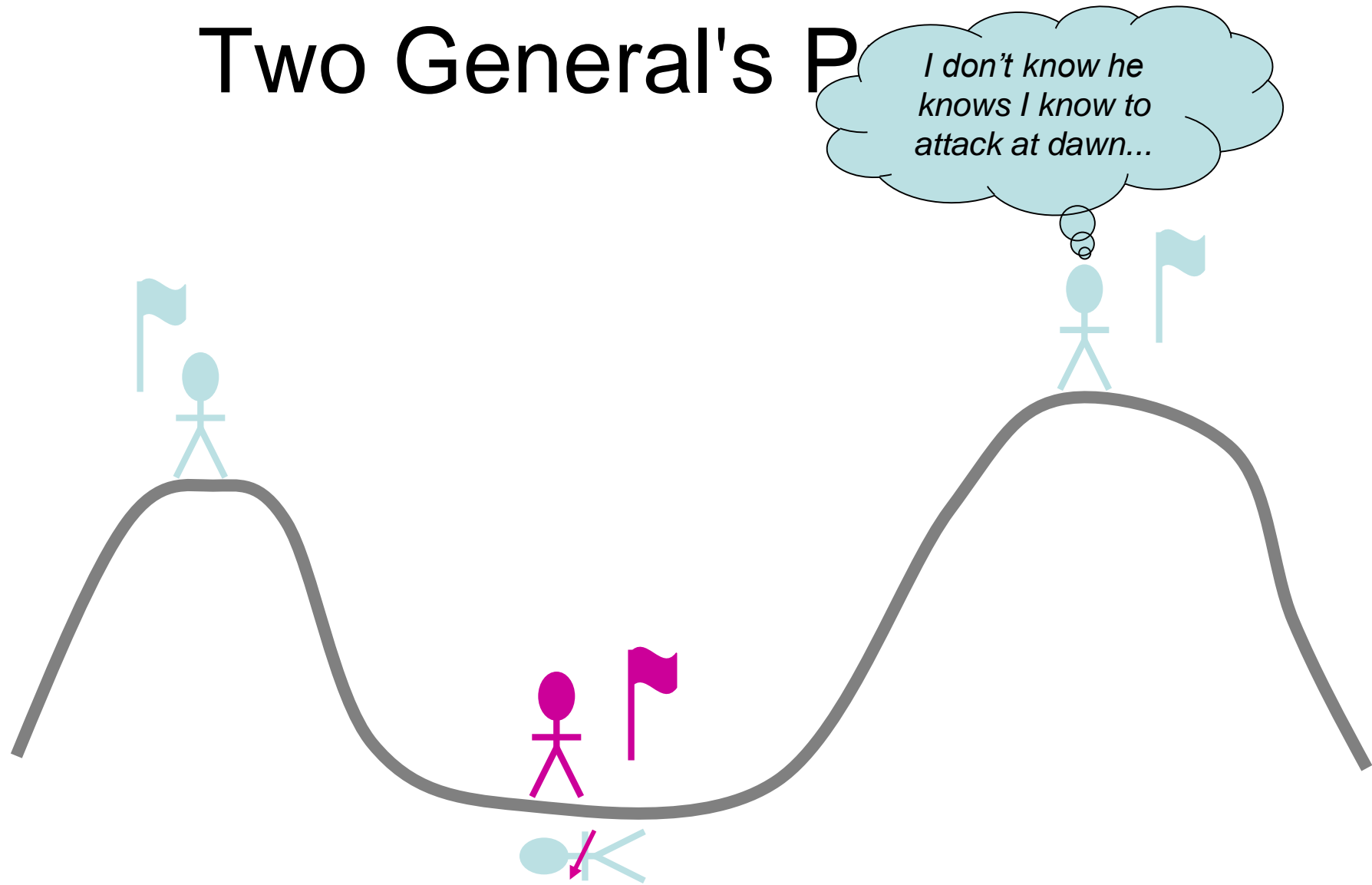
Two General's Problem



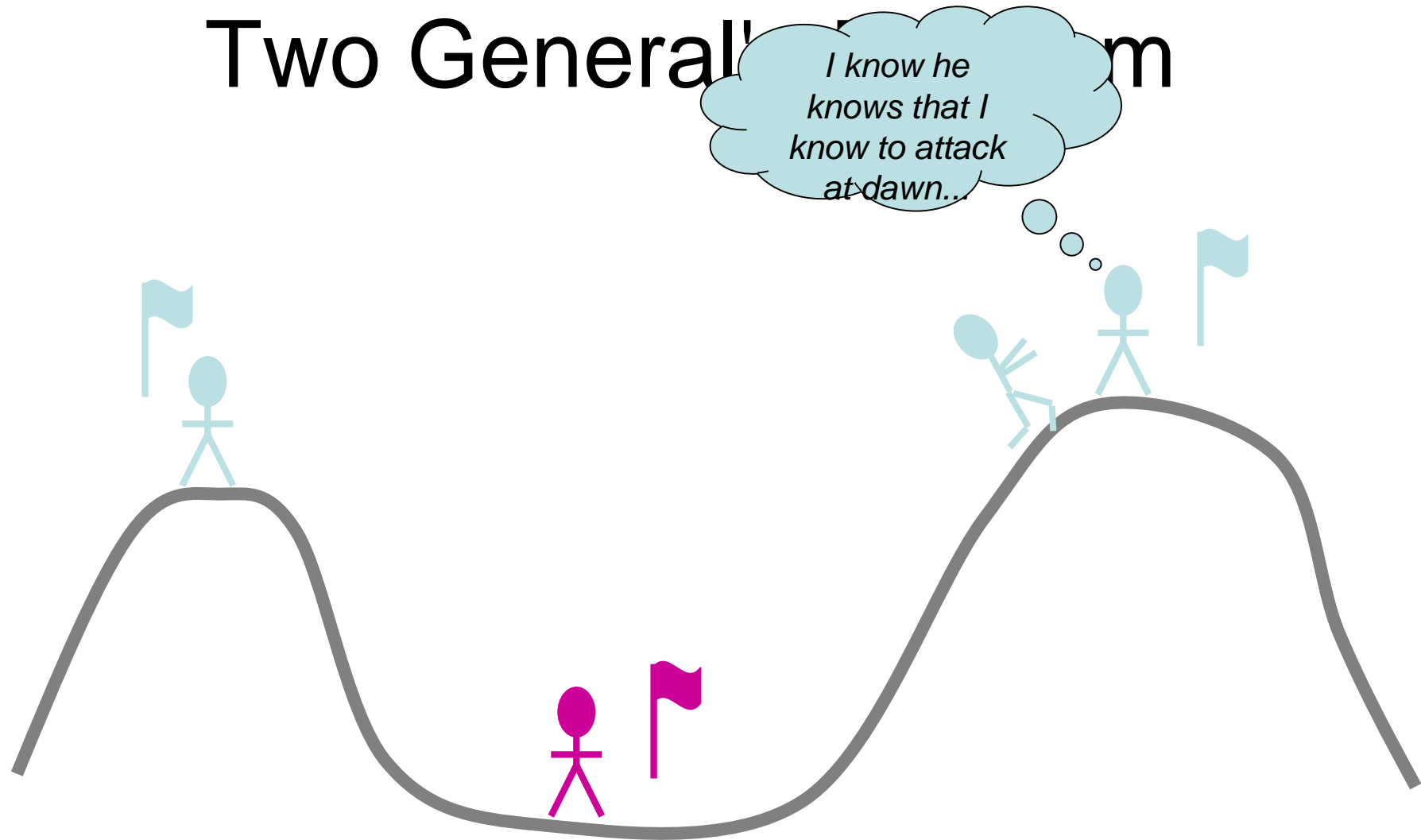
Two General's Problem



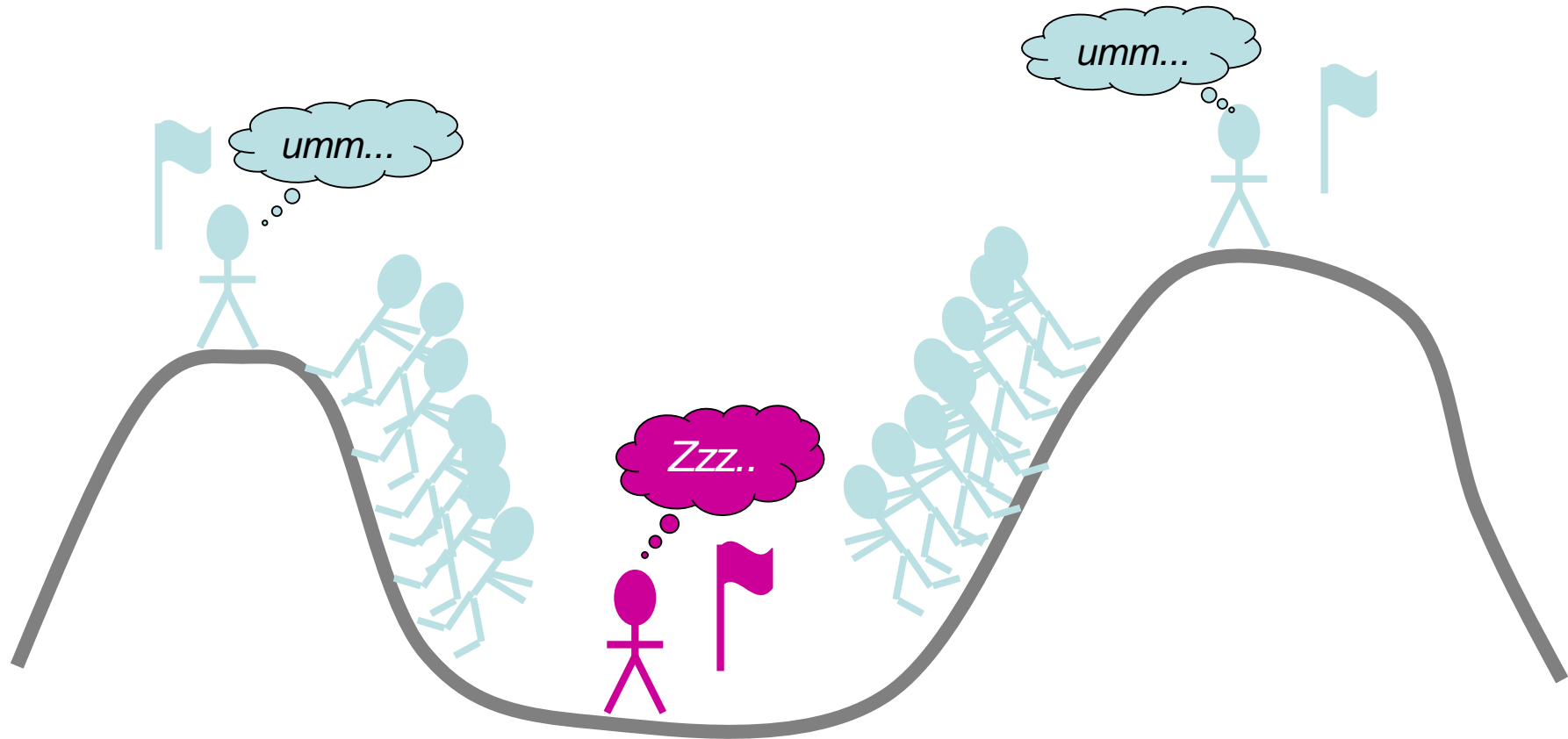
Two General's P



Two General's Problem



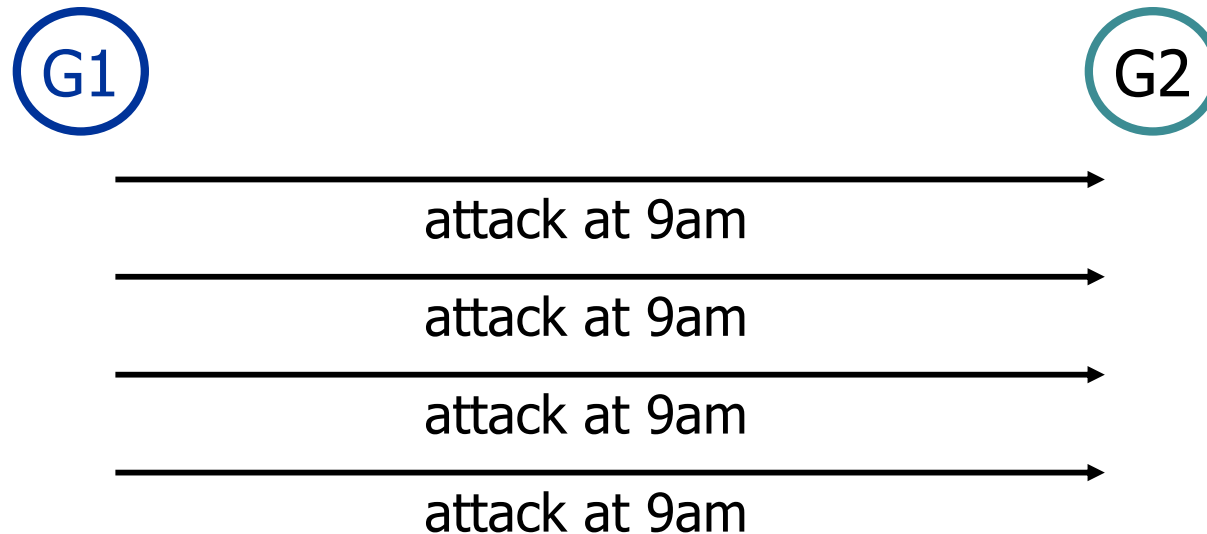
Two General's Problem



Probabilistic Approach?

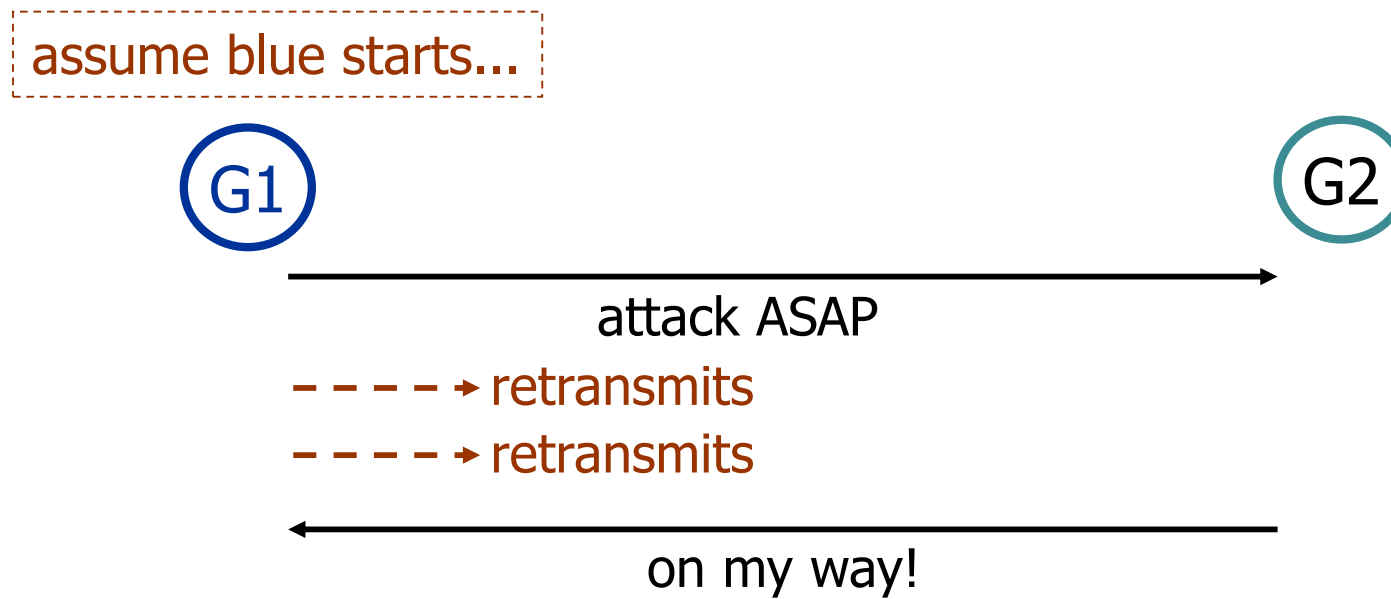
- Send as many messages as possible, hope one gets through...

assume blue starts...



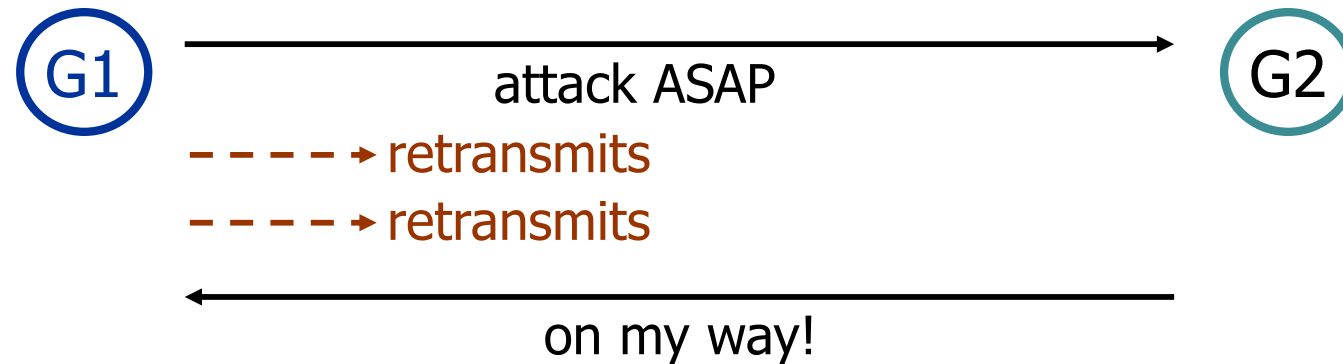
Eventual Commit

- Eventually both sides attack...

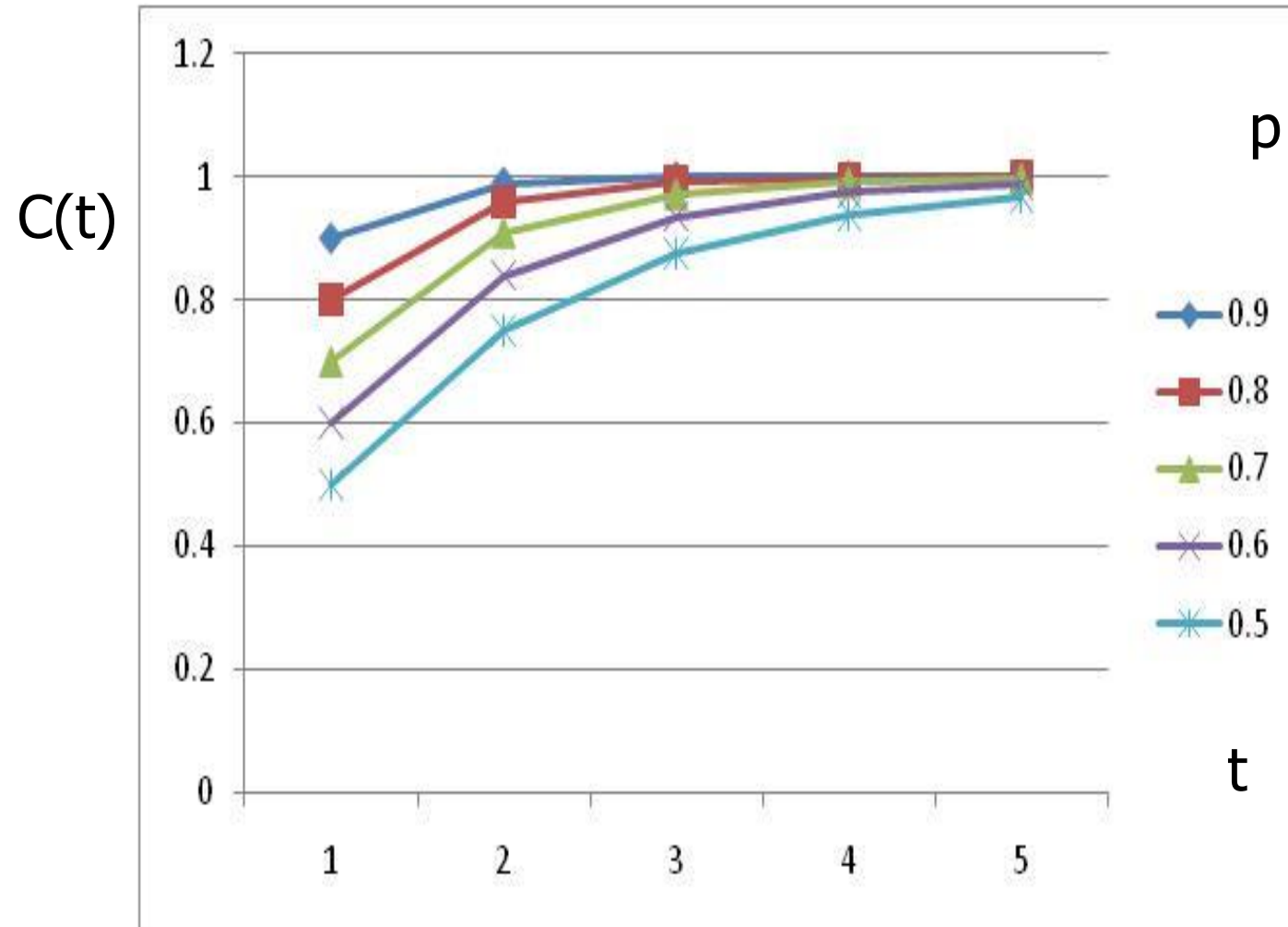


Eventual Commit

- One message sent every time unit
- Probability of success one message is p
- What is probability that red commits by time t ?

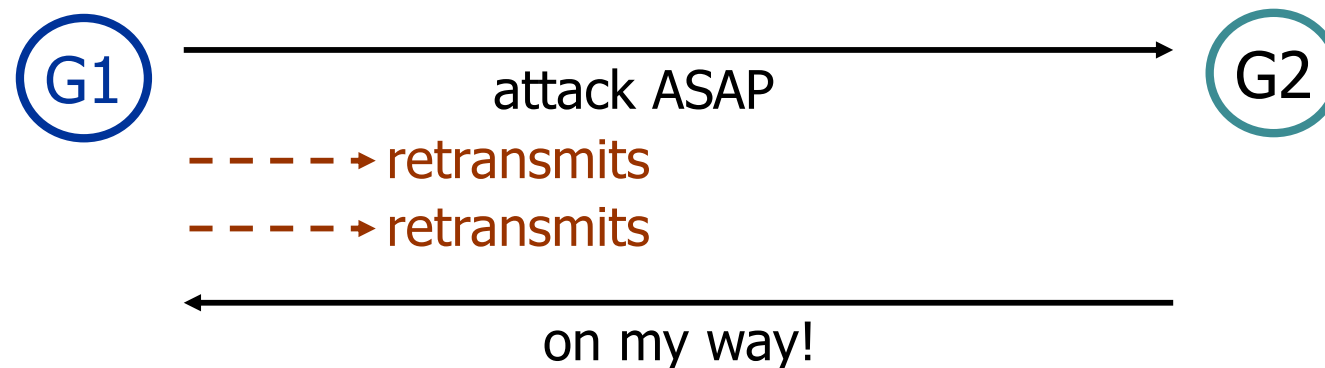


Eventual Commit



Eventual Commit

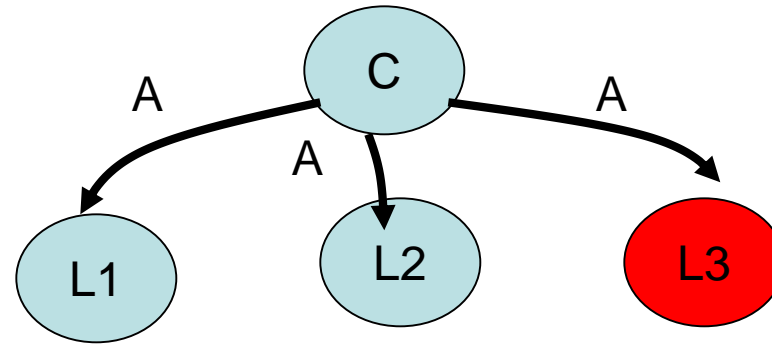
- How expensive is protocol?
- E = expected number of messages



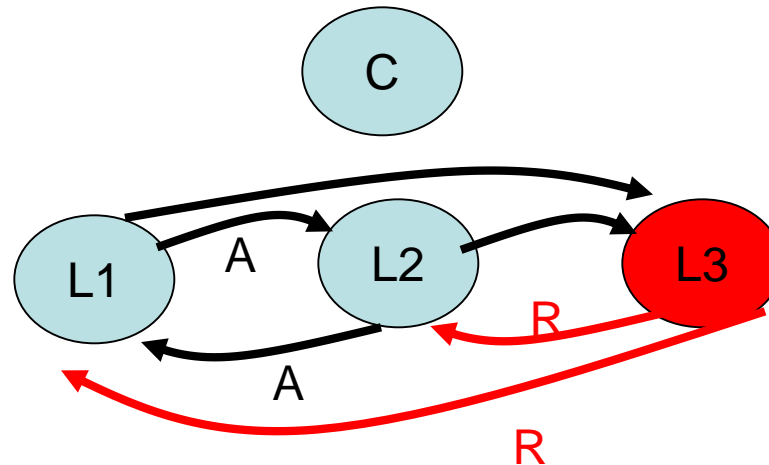
Another Problem: Bad Lieutenant

- Scenario: $m=1$, $n=4$, traitor = L3

OM(1):



OM(0):???



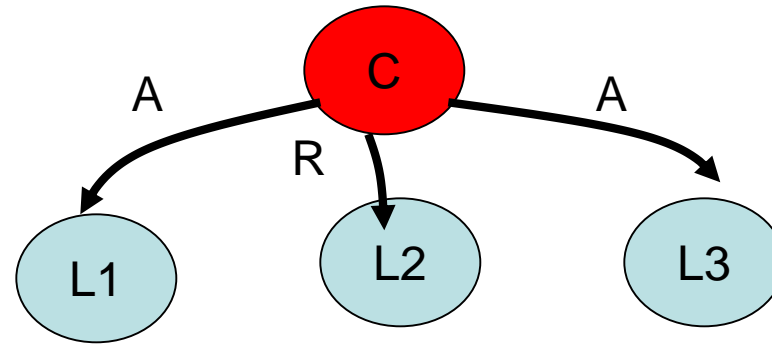
Decision??

$L1 = m(A, A, R)$; $L2 = m(A, A, R)$; Both attack!

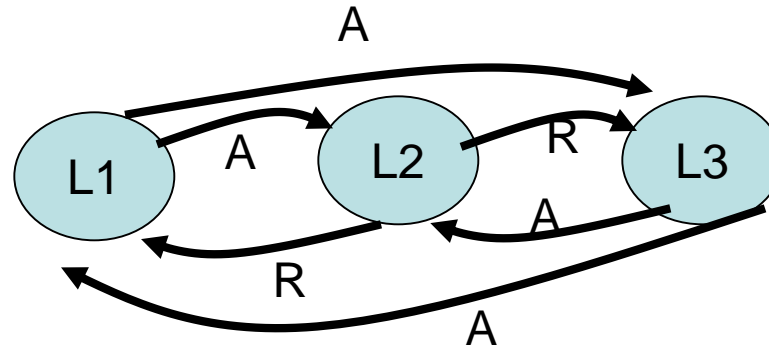
Another Problem: Bad Lieutenant

- Scenario: $m=1$, $n=4$, traitor = C

OM(1):



OM(0):???

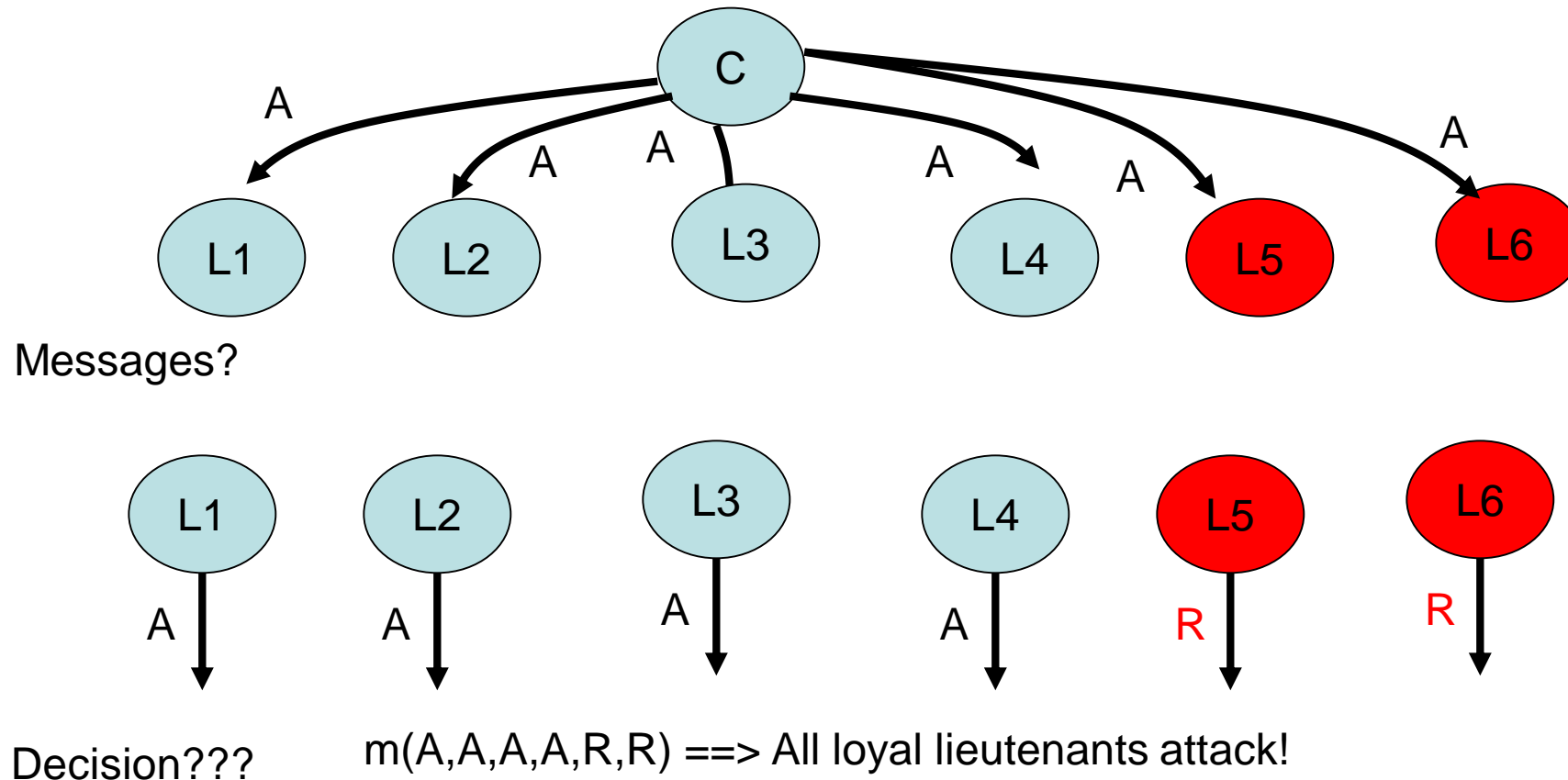


Decision??

$L1=m(A, R, A)$; $L2=m(A, R, A)$; $L3=m(A, R, A)$; Attack!

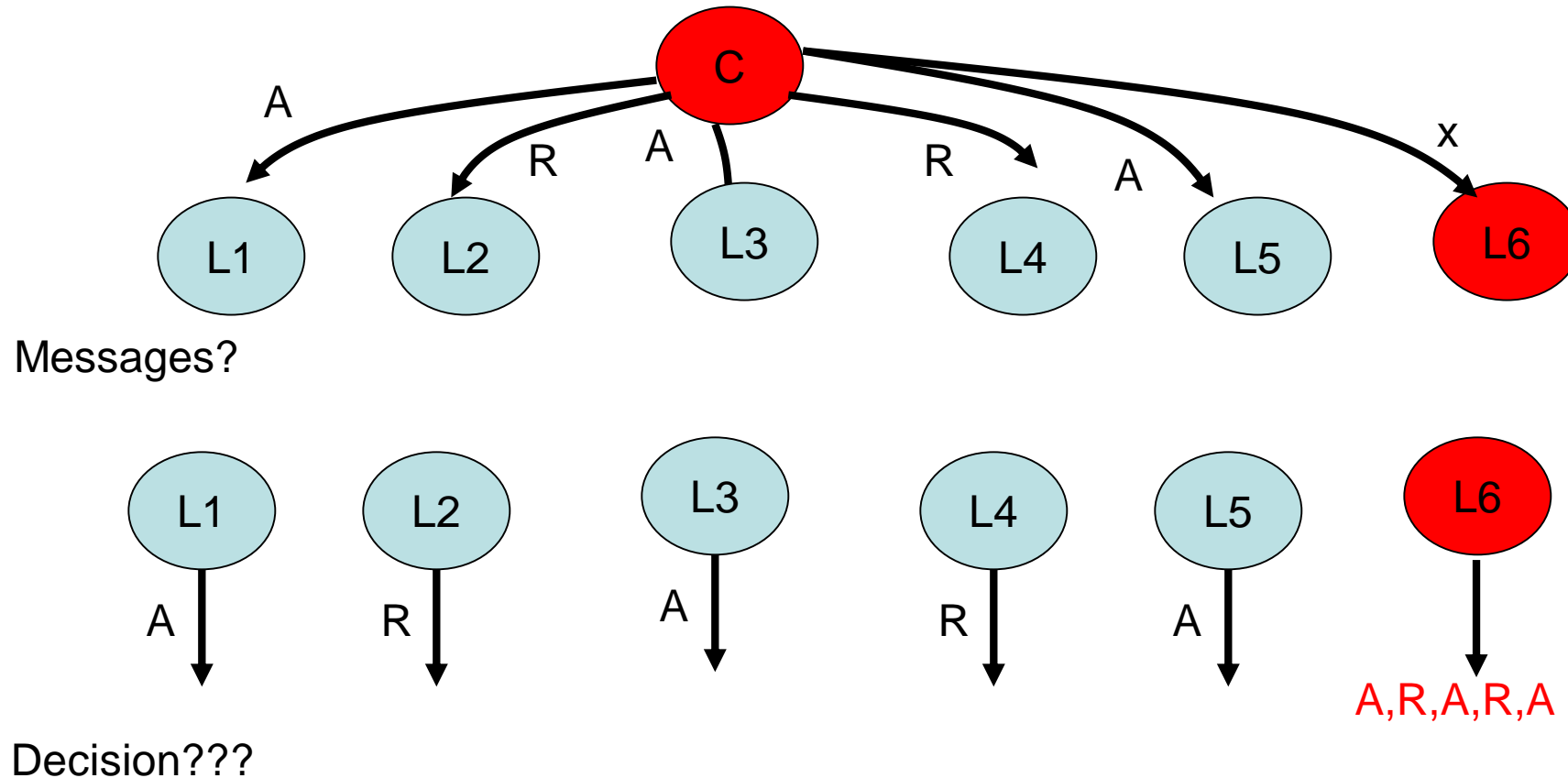
Another Problem: Bad Lieutenant

- Scenario: $m=2$, $n=3m+1=7$, traitors=L5, L6



Another Problem: Bad Lieutenant

- Scenario: $m=2$, $n=7$, traitors= C , $L6$



Decision with Bad Commander+

- L1: $m(A, R, A, R, A, A) \implies \text{Attack}$
- L2: $m(A, R, A, R, A, R) \implies \text{Retreat}$
- L3: $m(A, R, A, R, A, A) \implies \text{Attack}$
- L4: $m(A, R, A, R, A, R) \implies \text{Retreat}$
- L5: $m(A, R, A, R, A, A) \implies \text{Attack}$
- Problem: All loyal lieutenants do NOT choose same action

Next Step of Algorithm

- Verify that lieutenants tell each other the same thing
 - Requires rounds = $m+1$
 - OM(0): Msg from Lieut i of form: “L0 said v_0 , L1 said v_1 , etc...”
- What messages does L1 receive in this example?
 - OM(2): A
 - OM(1): (A,R,A,R,A,A) 2R, 3A, 4R, 5A, 6A (doesn't know 6 is traitor)
 - OM(0):
 - 2{ 3A, 4R, 5A, 6R}
 - 3{2R, 4R, 5A, 6A}
 - 4{2R, 3A, 5A, 6R}
 - 5{2R, 3A, 4R, 6A}
 - 6{ total confusion }
- All see same messages in OM(0) from L1,2,3,4, and 5
- $m(A,R,A,R,A,-) \implies$ All attack

BUFFER OVERFLOW ATTACK

The Problem

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer1[20] = {0,};
    char buffer2[20] = {0,};

    if(argc > 1) {
        strcpy(buffer2, argv[1]);
    }

    printf("buffer1: %s\n", buffer1);
    printf("buffer2: %s\n", buffer2);

    return 0;
}
```

The Problem

```
ywkwon@Kwon-Office-Windows ~/OS_lab  
$ ./a.exe ASDFWQAER  
buffer1:  
buffer2: ASDFWQAER
```

```
ywkwon@Kwon-Office-Windows ~/OS_lab  
$ ./a.exe ASDFWQAERADASFADFWQERASDFAD  
buffer1:  
buffer2: ASDFWQAERADASFADFWQERASDFAD
```

```
ywkwon@Kwon-Office-Windows ~/OS_lab  
$ ./a.exe ASDFWQAERADASFADFWQERASDFADWRQ#SAFASDF@#RADFASDDFASDFA  
buffer1: AFASDF@#RADFASDDFASDFA  
buffer2: ASDFWQAERADASFADFWQERASDFADWRQ#SAFASDF@#RADFASDDFASDFA
```

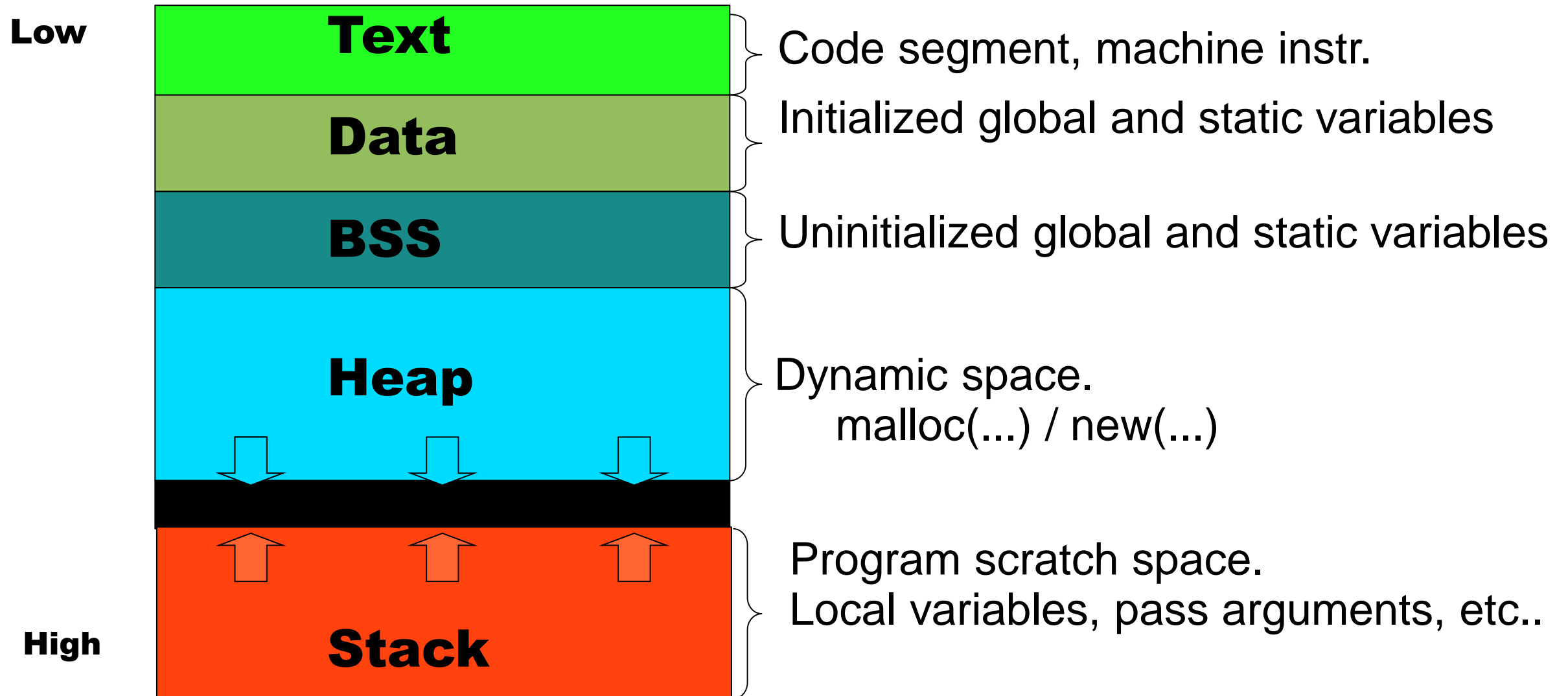

Idea

- The general idea is to pass very large strings to a buffer, which will overflow the buffer.
- For a program with sloppy code – it's easy to crash the server by overflowing a buffer
- It's sometimes possible to actually make the program do whatever you want (instead of crashing)

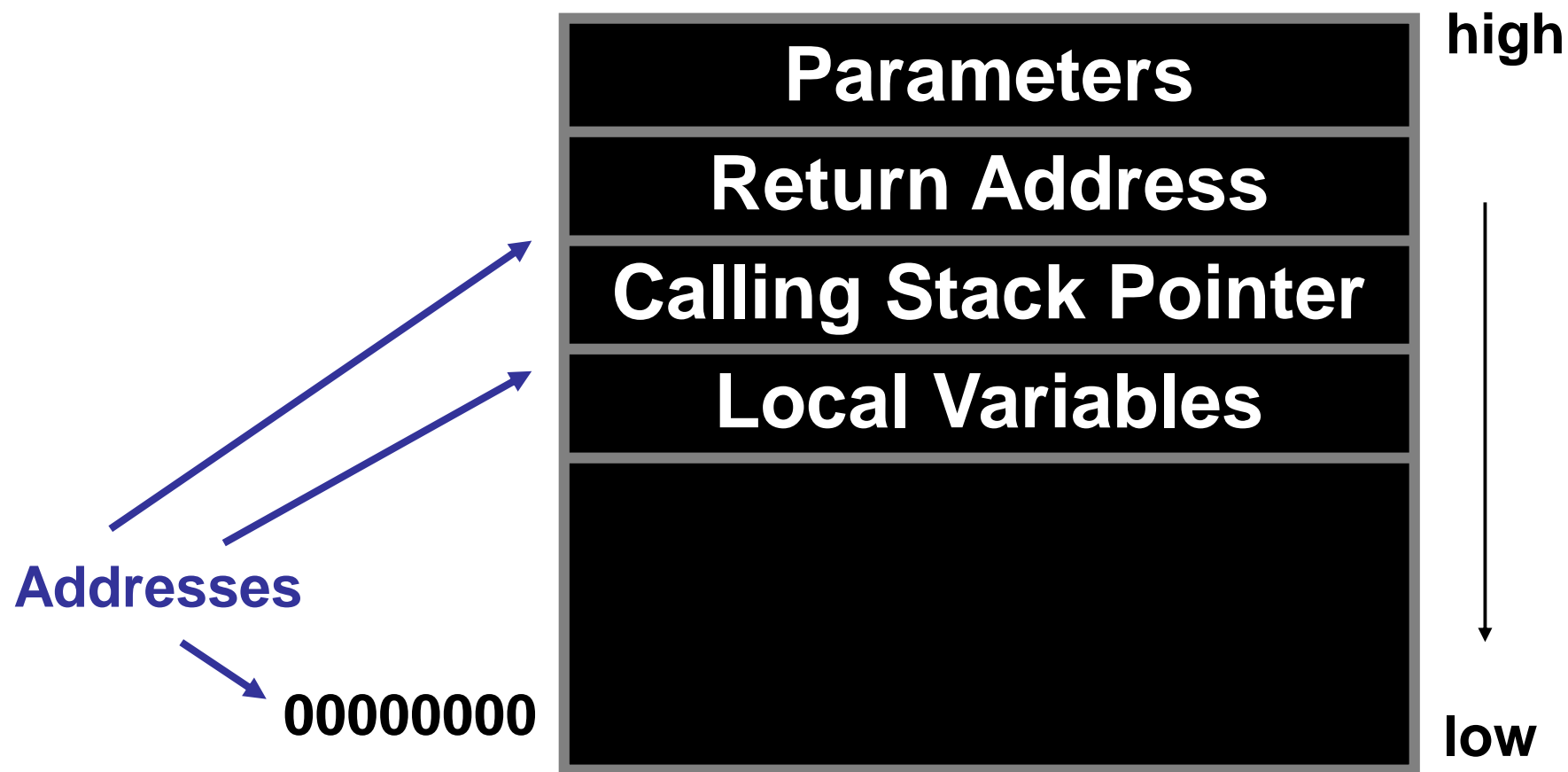
Background Necessary

- C functions and the stack.
- A little knowledge of assembly/machine language.
- How system calls are made (at the machine code level).
- **exec ()** system calls
- How to “guess” some key parameters.

Memory Structure



Stack



Sam

18
addressof(y=3) *return*
address
saved stack pointer
y
x
buf

Parameters

Return Address

Calling Stack Pointer

Local Variables

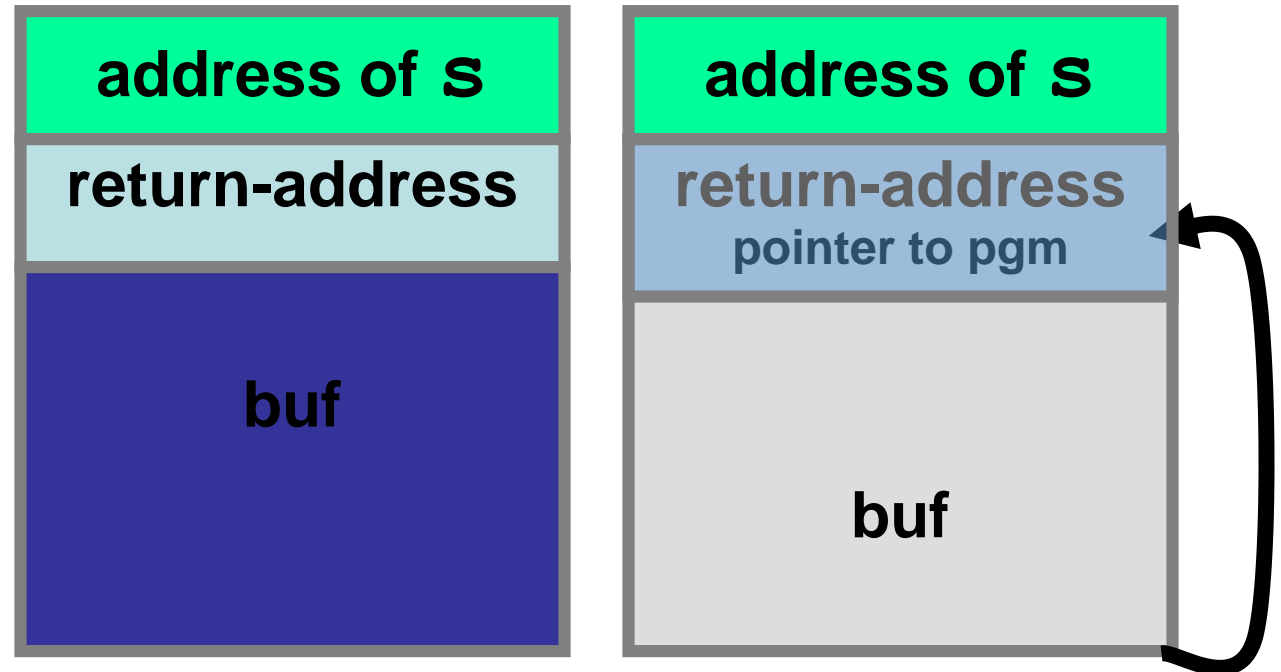
```
x=2;  
foo(18);  
y=3;
```

```
void foo(int j) {  
    int x, y;  
    char buf[100];  
    x = j;  
    ...  
}
```

How it works?

```
void foo(char *s) {  
    char buf[100];  
    strcpy(buf, s);  
    ...  
}
```

- The general idea is to overflow a buffer so that it overwrites the return address.
- When the function is done it will jump to whatever address is on the stack.
- We put some code in the buffer and set the return address to point to it!



Dump of assembler code for function `main`:

```
0x00000000100401080 <+0>:    push    %rbp
0x00000000100401081 <+1>:    mov     %rsp,%rbp
0x00000000100401084 <+4>:    sub     $0x60,%rsp
0x00000000100401088 <+8>:    mov     %ecx,0x10(%rbp)
0x0000000010040108b <+11>:   mov     %rdx,0x18(%rbp)
0x0000000010040108f <+15>:   call    0x100401130 <__main>
0x00000000100401094 <+20>:   movq    $0x0,-0x20(%rbp)
0x0000000010040109c <+28>:   movq    $0x0,-0x18(%rbp)
0x000000001004010a4 <+36>:   movl    $0x0,-0x10(%rbp)
0x000000001004010ab <+43>:   movq    $0x0,-0x40(%rbp)
0x000000001004010b3 <+51>:   movq    $0x0,-0x38(%rbp)
0x000000001004010bb <+59>:   movl    $0x0,-0x30(%rbp)
0x000000001004010c2 <+66>:   cmpl    $0x1,0x10(%rbp)
0x000000001004010c6 <+70>:   jle     0x1004010df <main+95>
0x000000001004010c8 <+72>:   mov     0x18(%rbp),%rax
0x000000001004010cc <+76>:   add     $0x8,%rax
0x000000001004010d0 <+80>:   mov     (%rax),%rdx
0x000000001004010d3 <+83>:   lea     -0x40(%rbp),%rax
0x000000001004010d7 <+87>:   mov     %rax,%rcx
0x000000001004010da <+90>:   call    0x100401150 <strcpy>
0x000000001004010df <+95>:   lea     -0x20(%rbp),%rax
0x000000001004010e3 <+99>:   mov     %rax,%rdx
0x000000001004010e6 <+102>:  lea     0x1f13(%rip),%rax      # 0x100403000
0x000000001004010ed <+109>:   mov     %rax,%rcx
0x000000001004010f0 <+112>:  call    0x100401140 <printf>
0x000000001004010f5 <+117>:  lea     -0x40(%rbp),%rax
0x000000001004010f9 <+121>:  mov     %rax,%rdx
0x000000001004010fc <+124>:  lea     0x1f0a(%rip),%rax      # 0x10040300d
0x00000000100401103 <+131>:  mov     %rax,%rcx
0x00000000100401106 <+134>:  call    0x100401140 <printf>
0x0000000010040110b <+139>:  mov     $0x0,%eax
0x00000000100401110 <+144>:  add     $0x60,%rsp
0x00000000100401114 <+148>:  pop     %rbp
0x00000000100401115 <+149>:  ret
0x00000000100401116 <+150>:  nop
0x00000000100401117 <+151>:  nop
0x00000000100401118 <+152>:  nop
0x00000000100401119 <+153>:  nop
0x0000000010040111a <+154>:  nop
```

Type `q` for more, `c` to quit, `s` to continue without paging.

[illegible]

Building a small (exploit) program

- Typically, the small program stuffed in to the buffer does an `exec()`.
- Sometimes it changes the password db or other files...



An **exploit** (from the English verb *to exploit*, meaning "using something to one's own advantage") is a piece of [software](#), a chunk of data, or a sequence of commands that takes advantage of a [bug](#), [glitch](#) or [vulnerability](#) in order to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic (usually computerized).

exec () example

```
#include <stdio.h>
```

```
void main() {
```

```
    char *name[2];
```

```
    name[0] = "sh";
```

```
    name[1] = NULL;
```

```
    execve("/bin/sh", name, NULL);
```

```
}
```

A Sample Program

```
unsigned char cde[] =
```

```
.....
```

```
"\xff\xff\xff\xff\xff\xff\xff\xff"  
"\xff\xff\xff\xff\xff\xff\xff\xff"  
"\xff\xff\xff\xff\xff\xff\xff\xff"  
"\xff\xff\xff\xff\xff\xff\xff\xff"  
"\xff\xff\xff\xff\xef\xbe\xad\xde";
```

```
void foo(char *s) {  
    char buf[100];  
    strcpy(buf,s);  
    ...  
}
```

} 108 bytes



Exploit code's location

Vulnerable C Code

- strcpy(), strncpy()
- strcat(), strncat()
- sprintf(), snprintf()
- gets()
- sscanf()
- Many others...

Vulnerable C Code

- strcpy() doesn't check size
- If we have
 - char buf[128];
 - strcpy(buf, userSuppliedString);
- This makes it too easy...

Vulnerable C Code

- `char *strncpy(char *dest, const char *src, size_t n);`
- We have a size, but what if..
 - `strncpy(somebuffer, str, strlen(str));`
- or..
 - `strncpy(somebuffer, str, sizeof(somebuffer));`
- Where `str` is supplied by user

Vulnerable C Code

- `char *strncat(char *dest, const char *src, size_t n);`
- Ex:

```
int vulnerable(char *str1, char *str2)
{
    char buf[256];
    strncpy(buf, str1, 100);
    strncat(buf, str2, sizeof(buf)-1);
    return;
}
```

OS FINGER PRINTING

What is OS Fingerprinting

- **OS Fingerprinting** is a method of detecting the remote host's operating system using information leaked by that host's TCP stack. To do this, we use:
 - the responses it gives to carefully crafted packets (active mode)
 - or by observing captured network traffic (passive mode).
- These methods are possible because each OS implements their TCP stack differently.
- OS Fingerprinting (ab)uses these differences.

Why is this useful?

- **Learning remote OS versions** can be an extremely valuable network reconnaissance tool, since many security holes are dependent on OS version.

Why is this useful – an example

- Lets say you are doing a penetration test and find **port 53 open**. If this is a vulnerable version of Bind, you only have one shot to exploit it since a failed attempt is likely to crash the daemon.
- An attack usually **involves a buffer overflow**, followed by the execution of **OS/architecture dependent shellcode**.
- With a good TCP/IP fingerprinter, you will quickly ***find that this machine is running 'Solaris 2.51' or 'Linux 2.0.35'*** and can adjust your shellcode accordingly.

Uses – another example

- Another possible use is for **social engineering**. Lets say that you are scanning your target company and the scanner reports a 'Datavoice TxPORT PRISM 3000 T1 CSU/DSU 6.22/2.06'. The hacker might now call up the target pretending to be 'Datavoice support' and discuss some issues about their PRISM 3000. "We are going to announce a security hole soon, but first we want all our current customers to install the patch -- I just mailed it to you ..."

Uses – last example

- Evil Cable/DSL companies like to charge customers based on the number of hosts in their home, not based on the bandwidth given.
- Using passive OS fingerprinting, if they observe two different operating systems, or two different versions of windows sending traffic from the same IP, they can assume you are running NAT, and fine you for violating their Terms of Service.

Active OS Fingerprinting

```
babylon~> telnet hpux.u-aizu.ac.jp
```

```
Trying 163.143.103.12 ...
```

```
Connected to hpux.u-aizu.ac.jp.
```

```
Escape character is '^]'.
```

```
HP-UX hpux B.10.01 A 9000/715 (ttyp2)
```

```
login:
```

Active OS Fingerprinting

```
babylon> telnet ftp.netscape.com 21
```

```
Trying 207.200.74.26 ...
```

```
Connected to ftp.netscape.com.
```

```
Escape character is '^['.
```

```
220 ftp29 FTP server ready.
```

```
SYST
```

```
215 UNIX Type: L8 Version: SUNOS
```

Active OS Fingerprinting

```
babylon> echo 'GET / HTTP/1.0\n' | nc hotbot.com 80 |  
egrep '^Server:'
```

```
Server: Microsoft-IIS/4.0
```

```
babylon>
```

Here, we send a **HTTP GET request** to a remote server, and observe the webserver software which is identified in the response.

The most dangerous/effective attack?

