Double free

```
char *ptr = malloc (100);
…
free (ptr);
…
free (ptr);
```

```
void* mymalloc (int size) {
  char *ptr = malloc (size + 4);
  if (ptr != NULL) {
     ptr[0] = 1;
     ptr += 4;
  }
  return (void*)ptr;
}
```

```
void* myfree (void *_ptr) {
  if (_ptr != NULL) {
     char *ptr = (char*)_ptr - 4;
     if (ptr[0] != 1) {
       printf ("double free detected\n");
       exit (0);
     }
     ptr[0] = 0;
     free (ptr);
  }
}
```

- The application code is using mymalloc and myfree APIs
- malloc and free are implementing buddy allocation
- Will myfree always terminate a program if the program has double free bug?
- Justify your answer

Hash function

- Maps data of arbitrary size onto a data of fix size

```
char buf[] = “Hello world\n”;
--------
int hash = 0;
int i;
for (i = 0; i < strlen (buf); i++)
    hash += (int)buf[i];
return hash;
```

Hash function

- A simple hash function has collisions
- If we use the hash function of the previous slide “Hello world\n” and “world Hello\n” yield the same hash value

Cryptographic hash function

- It is infeasible to find two massages whose hash values (generated by cryptographic hash function) are same
  - e.g., “Hello world\n” and “world hello\n” will have very different hash values

- SHA1 generates 20 bytes hash value for any arbitrary length of string
  - It is infeasible to find two messages whose SHA1 are same

Integrity of files

- Suppose a process wants to protect the integrity of files created by it
- By integrity, we mean that no other process is allowed to modify the file contents

Integrity

- Assuming that a process has a special file (say secure.txt) that can not be tampered

- Store SHA1 of the file contents in secure.txt

read

- read the entire file
- compute the SHA1 of the file contents
- check the SHA1 against the one stored in the secure.txt file
- report an error if the SHA1 does not match

write

- read the entire file
- check the integrity
- update the data in the read buffer
- recompute SHA1
- update SHA1 in secure.txt
- update the file

Problems with the previous approach

read / write the entire file on every read / write

- On every read/write, we need to read the entire file

Alternative approach

- Divide the files into blocks
- Compute SHA1 at the block granularity
- Store the SHA1 for each block in secure.txt
- On every read/write, read the conflicting blocks and check the integrity

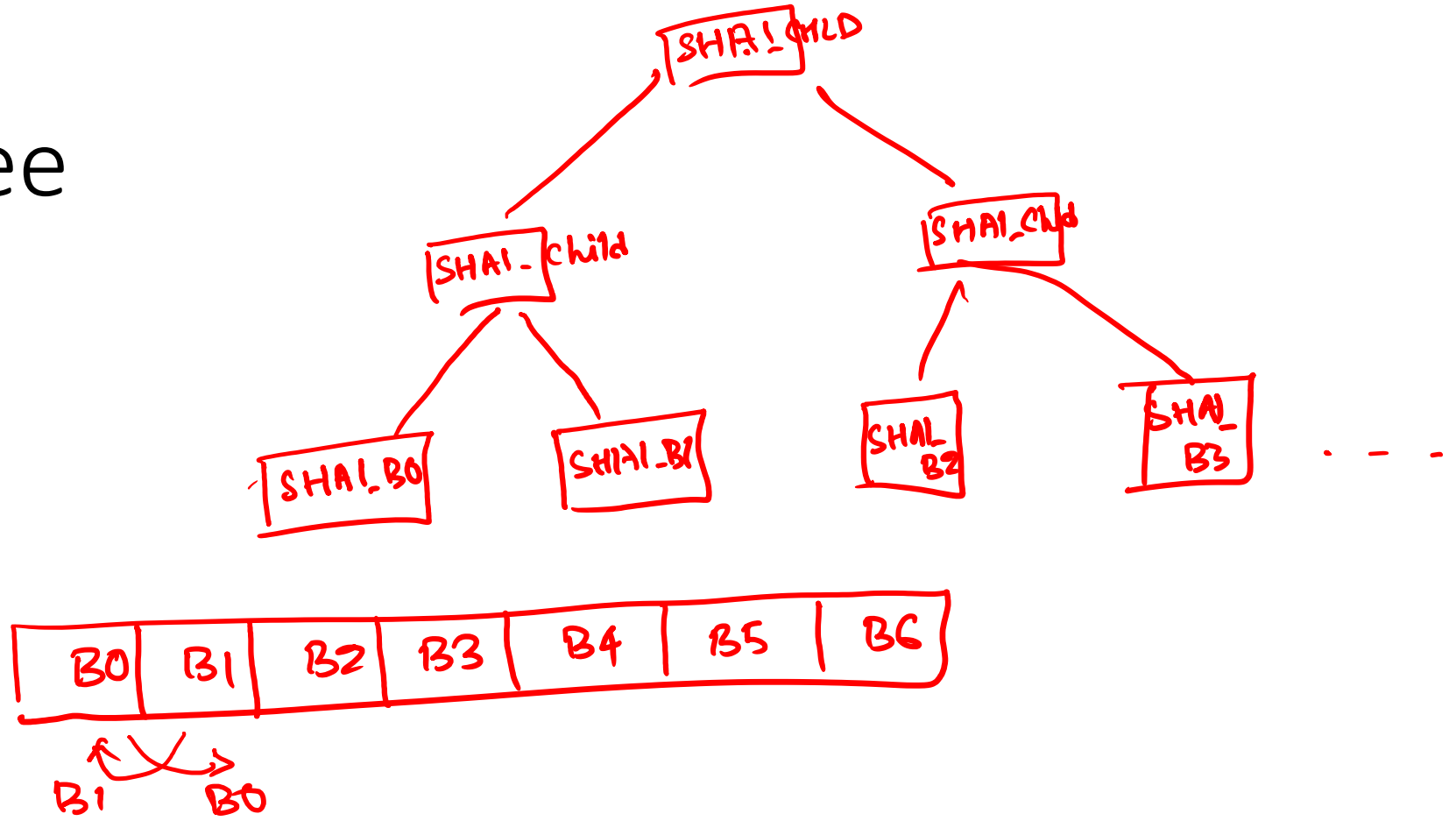
Problem with this approach

- Number of entries in the secure.txt depends on the size of the file

Merkle tree

- File is divided into blocks
- The leaf nodes of Merkle tree contain the SHA1 of the blocks
- An internal node in Merkle tree contain the SHA1 of the concatenation of its child nodes
- The root of the Merkel tree is unique for a given file

Merkle tree


SHA1_CHILD
SHA1_Child
SHA1_B0
SHA1_B1
SHA1_B2
SHA1_B3
B0
B1
B2
B3
B4
B5
B6
B1
B0

Merkle tree

- What if we reorder two blocks?

Concatenation will ensure the integrity

Next assignment

- Protect the integrity of files created by a library that implements Merkle tree