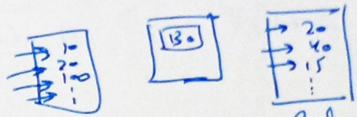
Class-5

Class-2

Concurrency ↔ Parallelisms



Balance Sheet 1 Balance Sheet 2

(130,3)
↑ Sheet context 1

Concurrency
S_1 — — —
S_2 — —
S_3 —

→ Time

Parallelism → multiple context in parallel

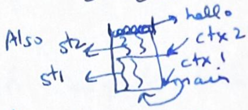
EIP → Instruction Pointer (CPU Register)/Program counter
Its a

Stack Pointer (SP) ↑ points to where stack is in memory ↓ A CPU register

Contexts are basically CPU registers. Changing Context make different programs start run. # Saving Context saves SP, EIP, General Purpose Registers.

In Class code,

ctx[2].uc-link = &ctx[1]; # when ctx[2] finishes, fall back to ctx[1].



In the code &ctx[0] refers to main context.

int *p1; → pointer to a int address , p1 = &x (address of x) , int **p = &p2; (Address of
,*p2

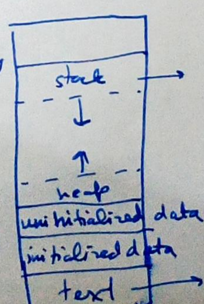
→ Now *p1 = 5; #change x to 5

```
# void fun1(int x){
        ____
  }
  int main(    ){
     void (*f_ptr)(int)= &fun1;
     *f_ptr(10);
  }
```

[function Pointer]

Stack

0x FFFFFFF — High address



→ stores local variables

0x 0000000 — Low address

→ stores the code being executed

0x → Base 16

Memory layout for C++ program

Class - 3

int foo_ref (int *p) {
    =
}

int main {
    int x = 2;
    foo_ref (&x);
}

Context switching makes more sense in parallelism

char x[sizee]; # char array

size of address > size of char.
Hence call by ref is slower in a 'char' w.r.t. call by value.

Functions can't return pointers of stack variables.

Heap memory → explicitly allocated → using new → returns address of creation.
→ explicitly deallocated

#
```
int sz;
scanf ("%d", &sz);
char *c = (char*) malloc (sz * sizeof(char));
```

Heaps can have memory leaks. → Keep allocating without free → program crashes

In stacks, you don't need to worry about re-cleaning the memory no longer in use. Stacks have Automatic Memory Management
eg (when func. returns)

Memory can get → fragmentation while malloc

→ C may not give stack overflow exception, unlike other high-level languages.

\# Stack memory is not happy with GBs of memory allocation, but small memory.

\# Look malloc.c file.

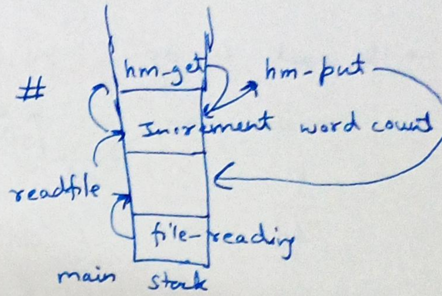
In Part 3,

```
char * foo_stack(){
    char A[Size]
    .
    return A ;}

char * foo_heap(){
    char *A = (char *) malloc(size * sizeof(char))

    return A;}
```

→ print return value of the two functions,
stack returns null;
heap returns data;

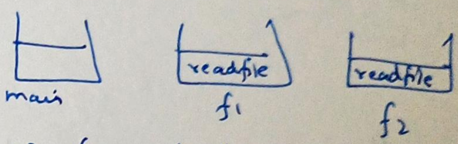
→ On return, all variable in stack memory are cleaned when function ends. Whereas heap memory stays alive.

\#



[Normal Execution without using thread]

\#

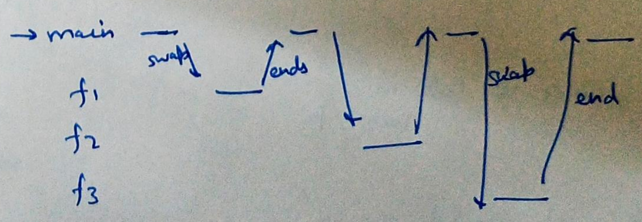


→ for each file, we want to create a new stack/ thread/context by mythread_create(readfile*, f1)
↓
sets up stack by make context API,

\# But won't run it.

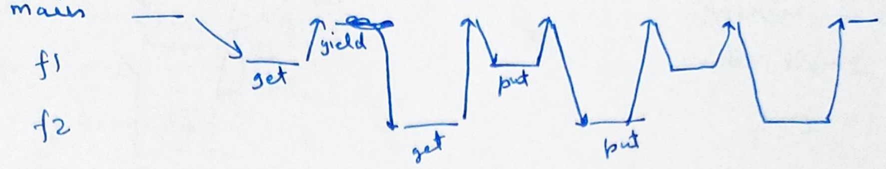
→ At mythread.join(), main will swap context with f1 thread, then uclink will come back to main, then join function, then swap context with f2, then uc link to main i.e. join.

\# for N files, we will have total N+1 stacks



\# From Comparison to simple stack, instead of 1 main stack, we have N+1 stacks now

→ We want to do



⇒ Join will start f1, yield ⟶ swap to f2.
↳ walk around the list of contexts

\# f1

- increase word_count
- c=get('foo')
- yield
- put ('foo', c+1)

f2

- increase word_count
- get('foo')
- yield
- put() 'foo', c+1

(race condition)
↓
[cause error in output] like less count

\# We add acquire_bucket(), release bucket
↓
will take 'foo' & only 1 context will be able to acquire bucket & come out.

f2 is stuck in acquire_bucket until f1 release the bucket.
↳ acquire a lock in hashmap

\# acquire_bucket → hashes key → lock_acquire()

\# release_bucket → hashes key → lock_realease()

\# Lock → Only 1 context can enter the lock.
↳ keeps * to context, that context is holding the lock

\# lock → (context pointer).

acquire → while ctx != NULL:
    yield()
  ctx = your context

release →
  ctx = NULL

\# yield should give us cyclic behaviour.
↓
Comment acquire/release & get race condition output.

\# Part 3 ⟶ Compare performance
↳ main.c is same, .h files are same