CSE 120
PA4 Discussion

PA4

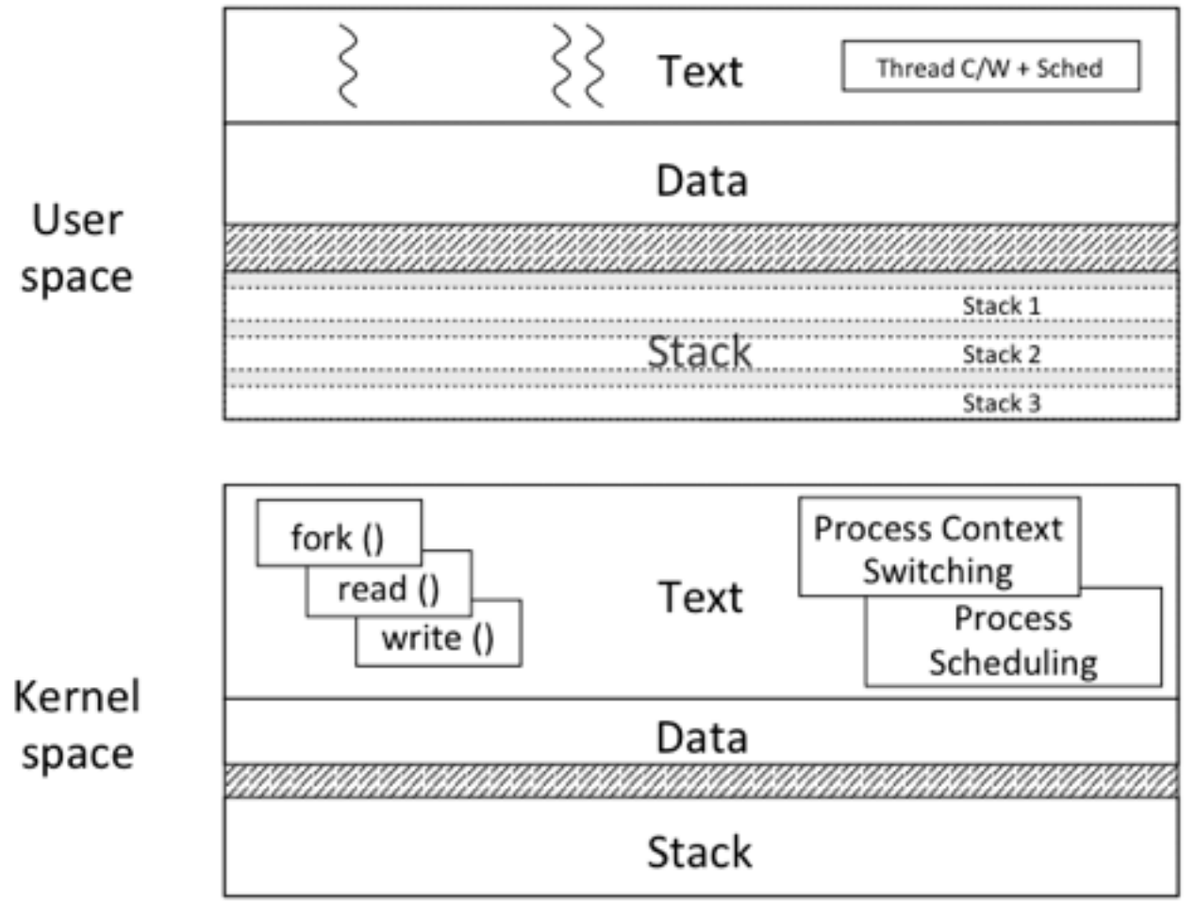
- Implement a user-level thread package
- Entirely on user-level, no kernel modification
- You will be only turning in mythreads.c, which includes following functions:
  - MyInitThreads()
  - MySpawnThread(func, param)
  - MyYieldThread(t)
  - MyGetThread()
  - MySchedThread()
  - MyExitThread()

Building Block

- setjmp(env) and longjmp(env, t) are the two utility function used to build thread packages

- setjmp and longjmp are very similar to SaveContext and RestoreContext in PA1.

User-level Threads



Stack is partitioned

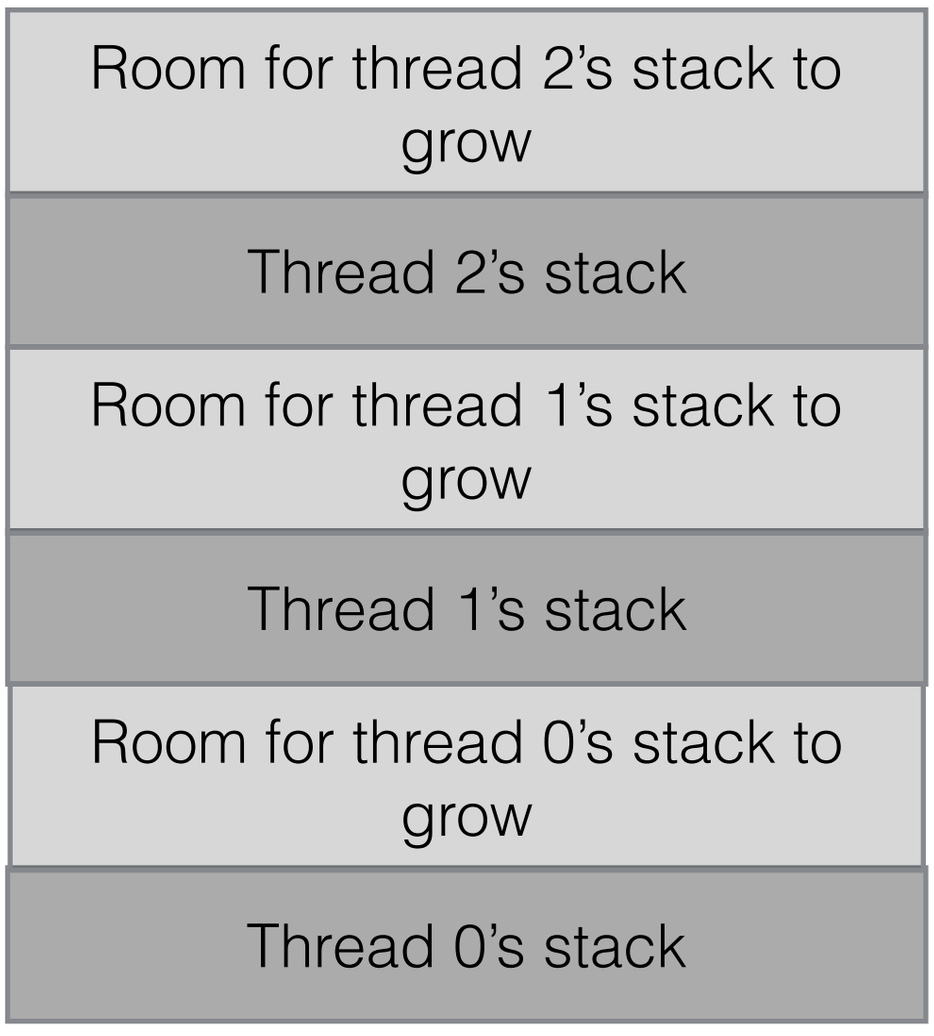
Stack Partition

- What is given

…

- What we desire

…



InitThreads

- Partition the stack into MAXTHREADS parts
- Make use of dummy array to separate stacks of different threads
- Needs to generalize the code in MySpawnThread(func, param) to support partition into MAXTHREADS parts

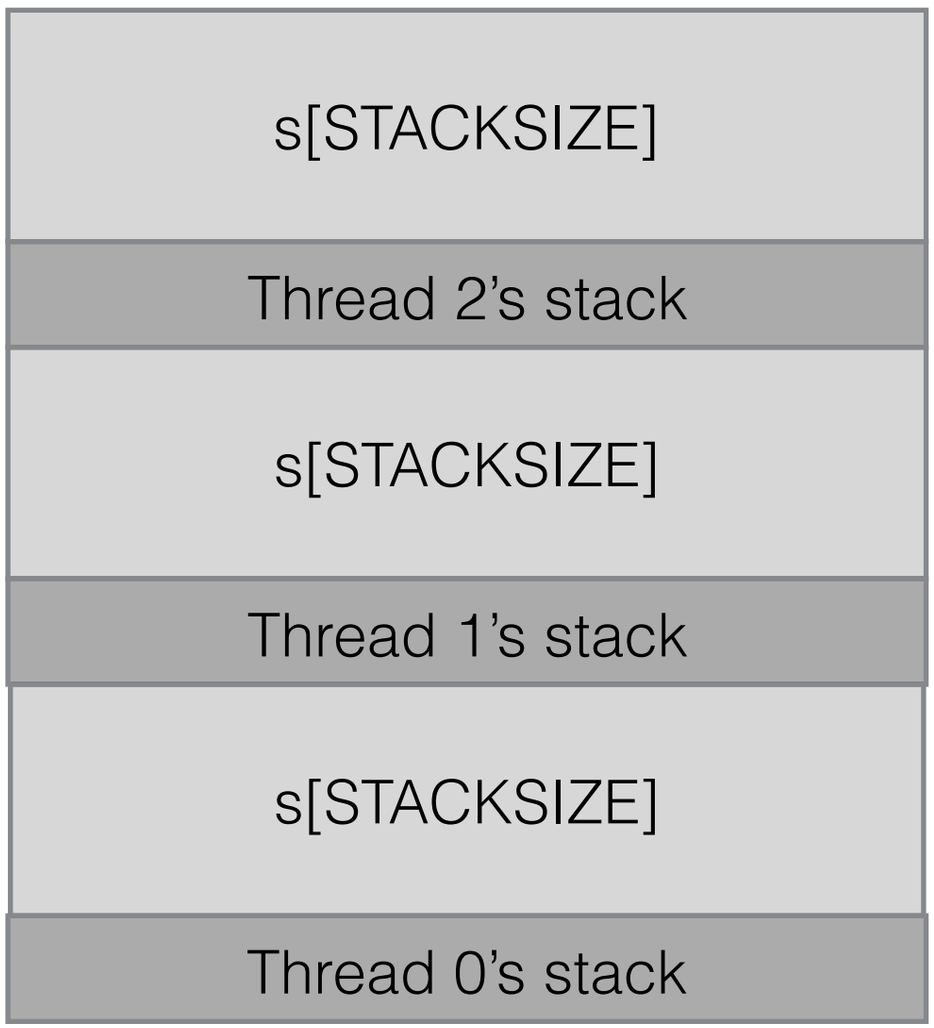
How to achieve partition

- Push the SP up by declaring a dummy array s[STACKSIZE]
- Save the execution environment (PC, SP, etc.) using setjmp(env)

Initial State

After partition

...



setjmp(thread[2].env)

setjmp(thread[1].env)

setjmp(thread[0].env)

One Solution

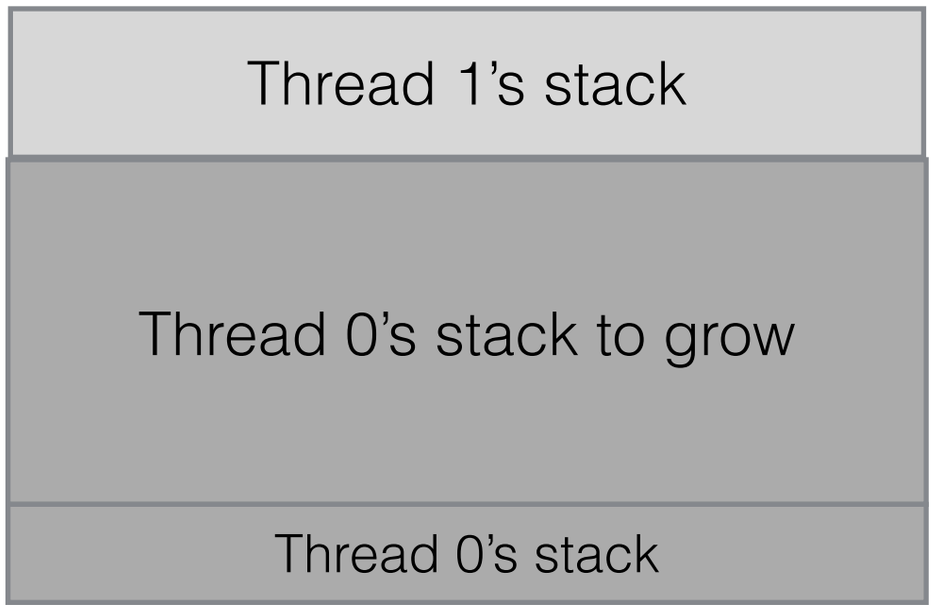
```
for (int i=1; i<= MAXTHREADS; i++)

{
    char s[STACKSIZE]; //dummy array

    Save the environment using setjmp(thread[i].env);

}
```

Not correct. In each iteration of the for loop, s[STACKSIZE-1] points to the same location

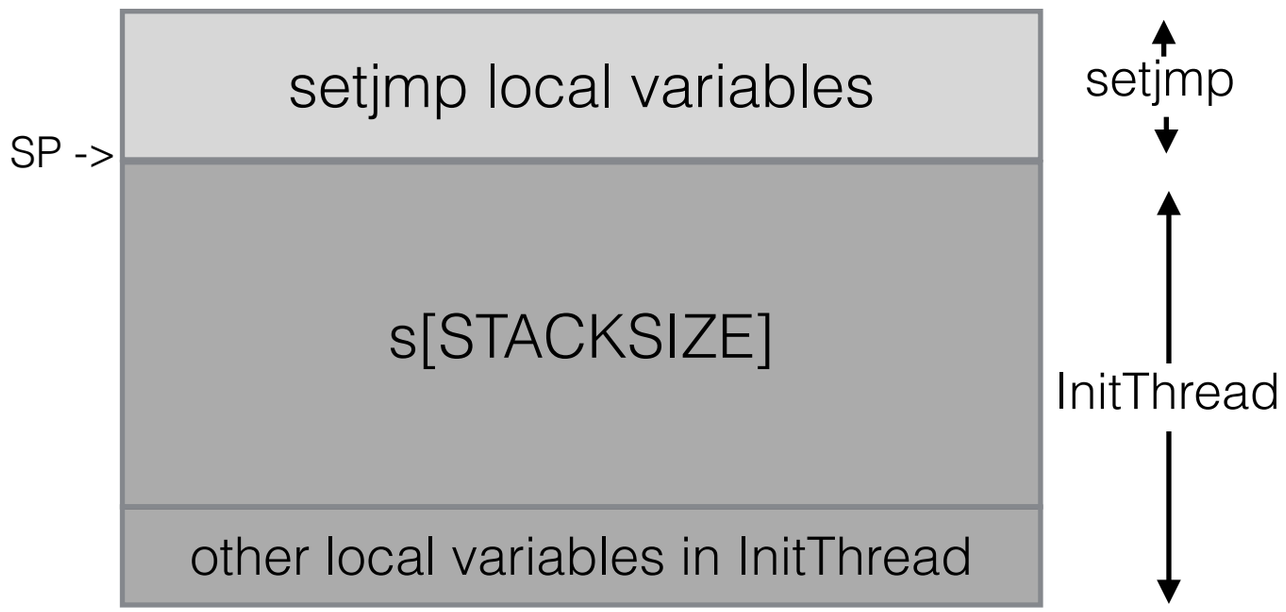
i=1

- Memory snapshot when thread context is saved inside setjmp(thread[i].env)

Process perspective



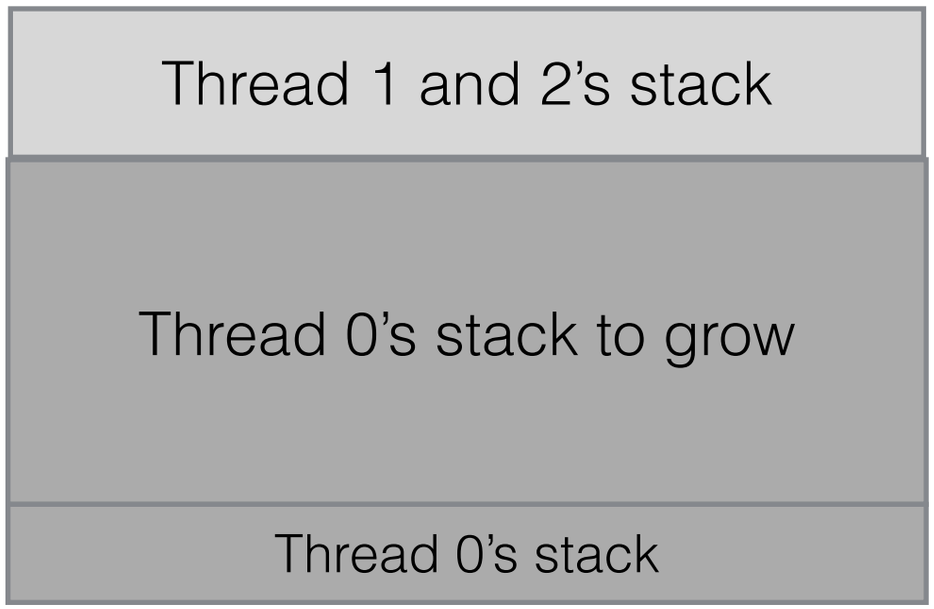
Kernel perspective



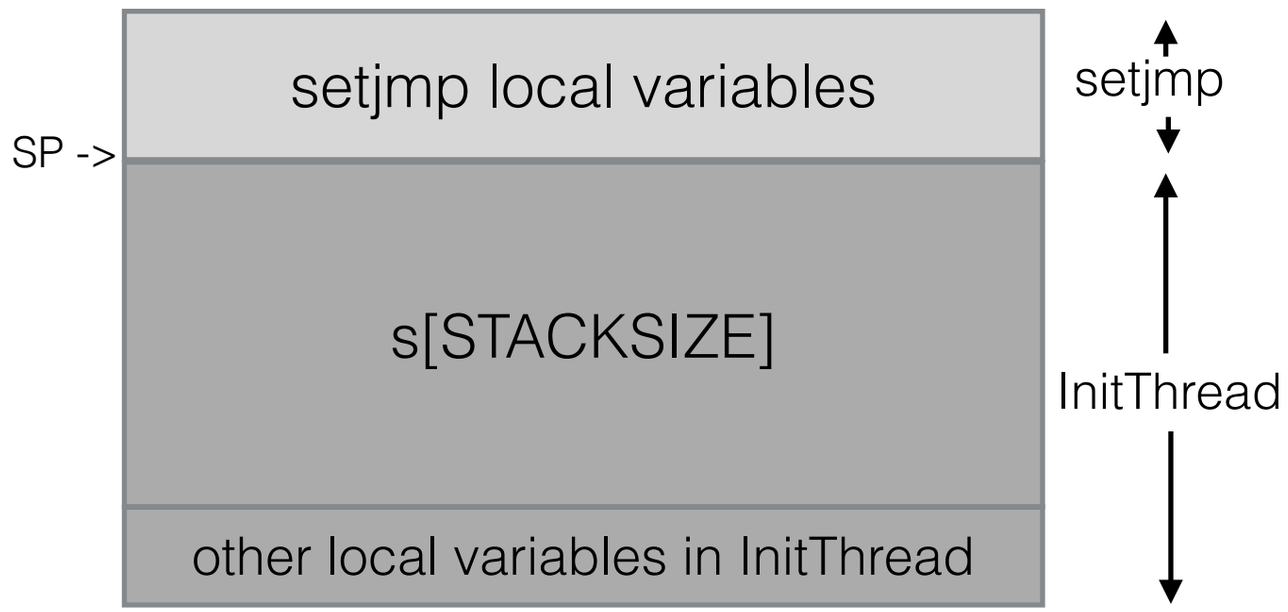
i=2

- Memory snapshot when thread context is saved inside setjmp(thread[i].env)

Process perspective



Kernel perspective



SP is not pushed up!

Thread 1 and 2's stack ended at the same memory location!

How about this

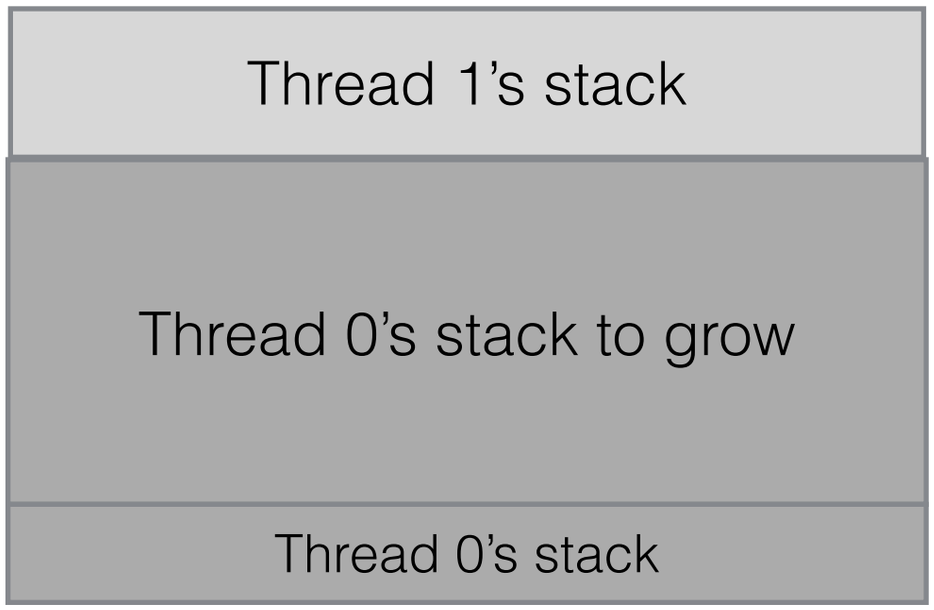
```
for (int i=1; i<= MAXTHREADS; i++)

{
    char s[i*STACKSIZE]; //dummy array

    Save the environment using setjmp(thread[i].env);

}
```

Now it works. In each iteration of the for loop, s[i*STACKSIZE-1] points to different location in different iterations

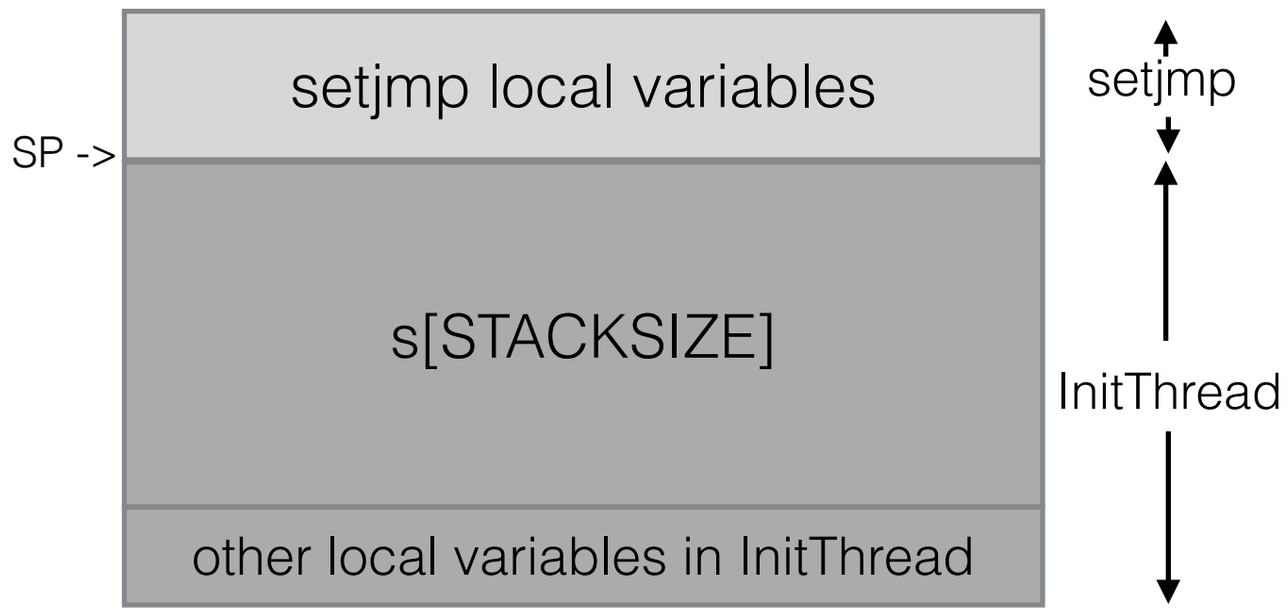
i=1

- Memory snapshot when thread context is saved inside setjmp(thread[i].env)

Process perspective

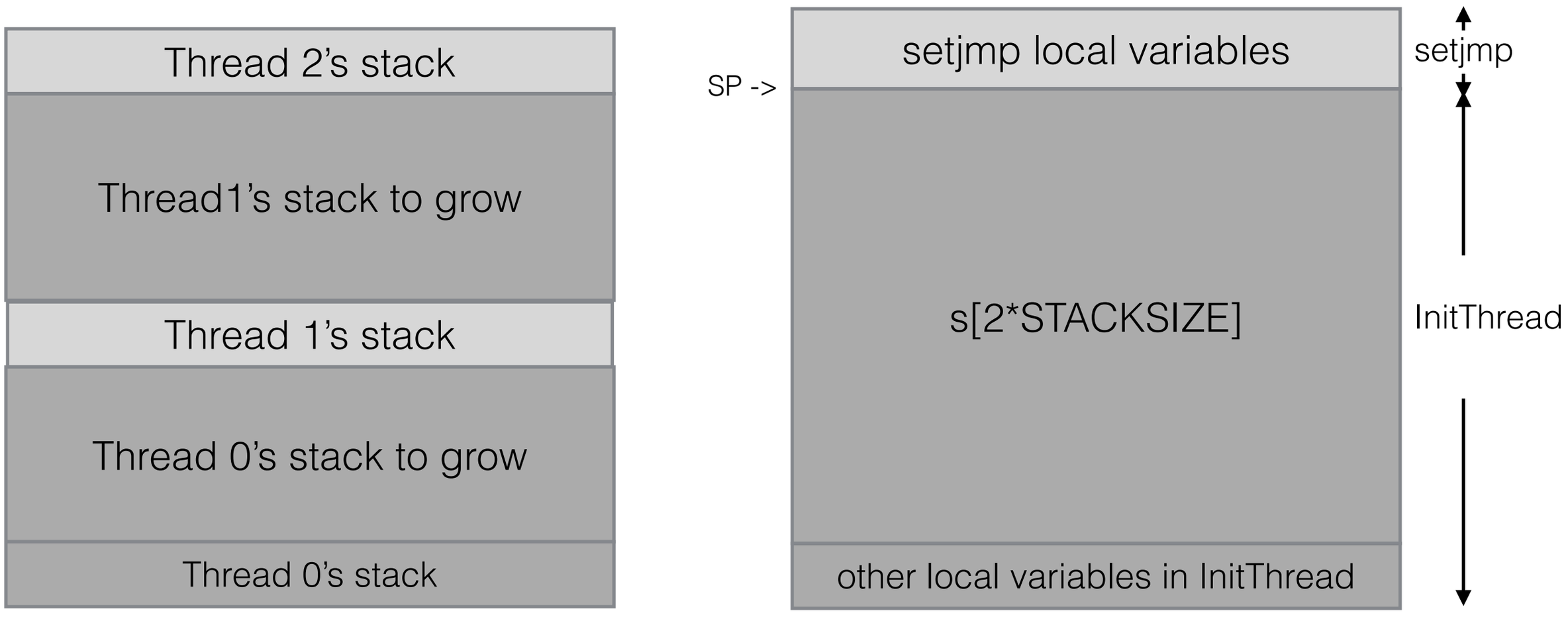


Kernel perspetive



i=2

- Memory snapshot when thread context is saved inside setjmp(thread[i].env)



SP moves to a different location this time, so it would work!

Alternative

- Using Recursion

stackPartition(int number_partitions) {

if (number_of_partitions <=1) then return;

else {

char s[STACKSIZE];

Save the environment using setjmp(thread[i].env);

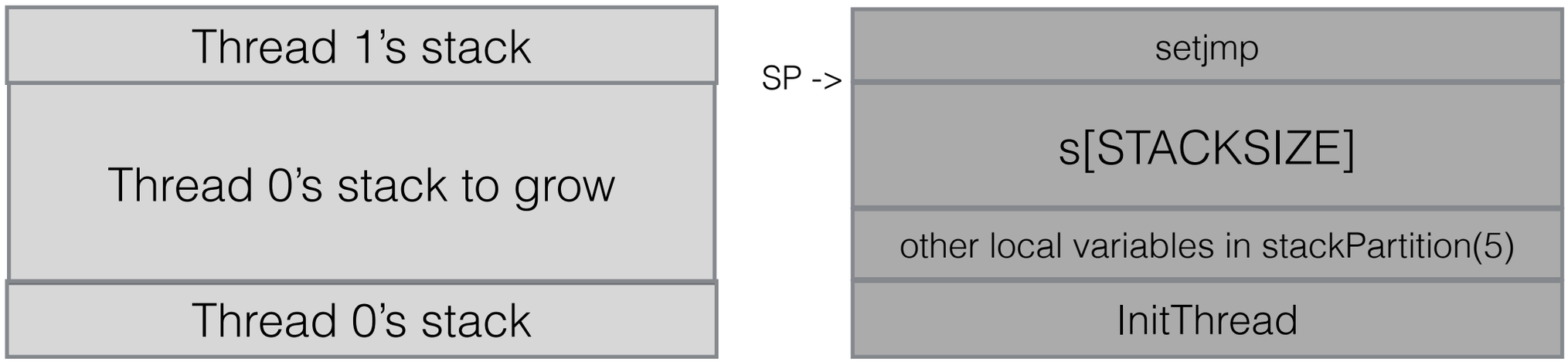
stackPartition(number_partitions-1);

}

}

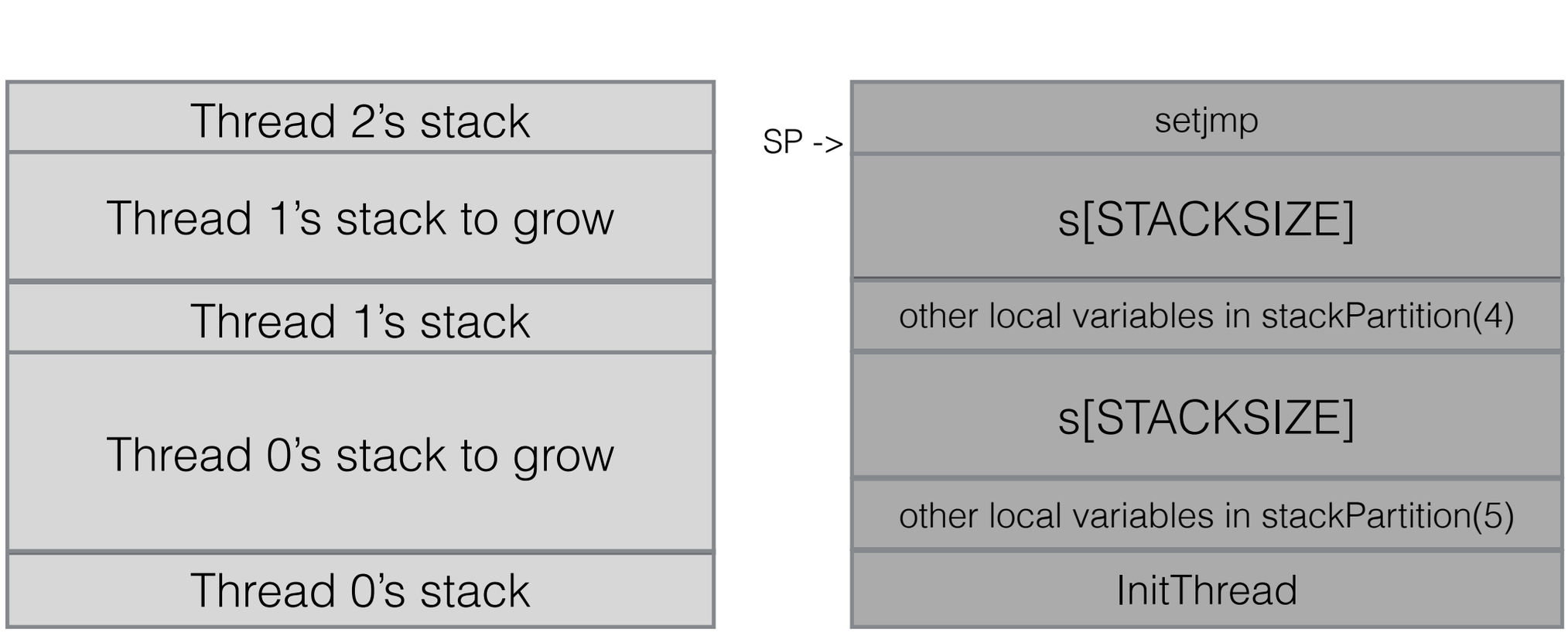
Example

- Partition the stack into 5 parts, initially calls stackPartition(5).



Example

- Inside stackPartition(5), we call stackPartition(4)



MySpawnThread

- MySpawnThread(func, param) spawns a new thread which executes func(param)

- Returns thread-id of the thread spawned.

- Incremental assignment of thread id.

Thread Id Example

Thread Id

0	T0
1	T1
2	T2
3	T3
4	T4
5	T5
6	T6
7	
8	
9	

Thread 0 ~ 6
spawned

Thread Id Example

0	T0
1	T1
2	
3	T3
4	T4
5	
6	T6
7	
8	
9	

Thread 2, 5 exits

Thread Id Example

0	T0
1	T1
2	
3	T3
4	T4
5	
6	T6
7	T7
8	
9	

Thread 7 spawned
Where should be put thread 7?

Thread Id Example

0	T0
1	T1
2	
3	T3
4	T4
5	
6	T6
7	T7
8	T8
9	T9

Thread 8, 9 spawned

Thread Id Example

0	T0
1	T1
2	T10
3	T3
4	T4
5	
6	T6
7	T7
8	T8
9	T9

Thread 10 spawned

Thread Id Example

0	
1	T1
2	T10
3	
4	T4
5	
6	T6
7	T7
8	T8
9	T9

Thread 0, 3 exits

Thread Id Example

Thread 11 spawned
Where should we put thread 11?

0	
1	T1
2	T10
3	T11
4	T4
5	
6	T6
7	T7
8	T8
9	T9

<— Last created

Thread Id Example

0	T13
1	T1
2	T10
3	T11
4	T4
5	T12
6	T6
7	T7
8	T8
9	T9

Thread 12, 13 spawned

MySpawnThread

- Keep track of the last assigned thread id
- When need to spawn a thread, find the next available id after the last assigned thread id.
- Also need to store the function pointer void (*f)(), int p, which the spawned thread would need in the future
- Needs to add some variables to the thread table

MyYieldThread

- MyYieldThread(t) causing the currently running thread to yield to thread t
- Returns the id of the thread which called MyYieldThread
- Very similar to MyContextSwitch in PA1

Example

Thread 1 starts first

Thread 1:

```
do some work
x = MyYieldThread(2)
// x = 2
continue working
```

Thread 2:

```
do some work
y = MyYieldThread(1)
// y = 1
continue working
```

MyYieldThread

- Use setjmp to save the context of current thread
- Use longjmp to give up CPU to the specified thread.
- We can set the second parameter of longjmp(env, t) to be the currently running thread id. So after longjmp returns, the thread that gets control of CPU will know what the previous running thread is (the id to return in MyYieldThread).

Edge Cases

- Yielding to self
- Yielding to an invalid thread id
  - MyYieldThread(-3)
  - MyYieldThread(MAXTHREAD + 10)

MySchedThread

- Pick one thread to run according to the scheduling policy
- In PA4, we are implementing a FIFO policy
- Once you determined which thread to run next, you can just make use of MyYieldThread

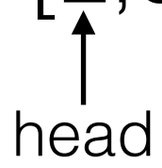
FIFO queue

- First in first out, last in last out
- Things become tricky when MyYieldThread comes into play and change the location of interior elements
- MySpawnThread would add new elements into the queue

Scheduling Example

Running Thread: 1

Queue: [2,0,3,4]



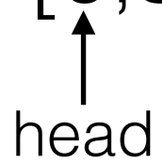
Thread 1 calls MySchedThread(), which thread should run next?

Where should we put thread 1 in the queue?

Scheduling Example

Running Thread: 2

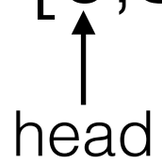
Queue: [0,3,4,1]



Scheduling Example

Running Thread: 2

Queue: [0,3,4,1]

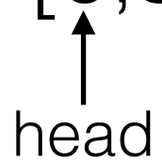


Thread 2 calls MySpawnThread(), creates thread 5, where should we put thread 5?

Scheduling Example

Running Thread: 2

Queue: [0,3,4,1]



Thread 2 calls MySpawnThread(), creates thread 5, where should we put thread 5?

Scheduling Example

Running Thread: 2

Queue: [0,3,4,1,5]

Scheduling Example

Running Thread: 2

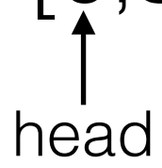
Queue: [0,3,4,1,5]

Thread 2 calls MyYieldThread(4), what should the queue look like?

Scheduling Example

Running Thread: 4

Queue: [0,3,1,5,2]



MyExitThread

- The function to call when thread is done with its job
- Always call MyExitThread() after the function execution line in mythreads.c
- It releases the slot in thread table so threads spawned later on can make use of that.
- Needs to reset the **env** to initial state (you may need to add some variables inside thread table).

PA4 Checklist

- Pre-partition stack space in MyInitThread
- MySpawnThread chooses the right thread Id
- MySchedThread picks the right thread to run
- MyYieldThread should switch to specified thread and return the id of the thread that gave up CPU.
- MyExitThread reset env so thread slot can be reused.