`setjmp`, `longjmp` and User-Level Threads

Failures are much more fun to hear about afterward,
they are not so funny at the time.

Charles Anthony Richard Hoare

Spring 2019

`setjmp` and `longjmp`

- You can jump between functions, conditionally!
- Use a jump buffer to save the return point.
- Use a long-jump to jump to a return point.
- Header file `setjmp.h` is required.
- A jump buffer is of type `jmp_buf`.
- Set up a jump buffer with function `setjmp()`.
- Execute a long jump with function `longjmp()`.

Here is the concept

- Declare a variable of type `jmp_buf`:

```
jmp_buf   JumpBuffer;
```

- Call function `setjmp()` to mark a return point:

```
setjmp(JumpBuffer);
```

- Later on, use function `longjmp()` to jump back:

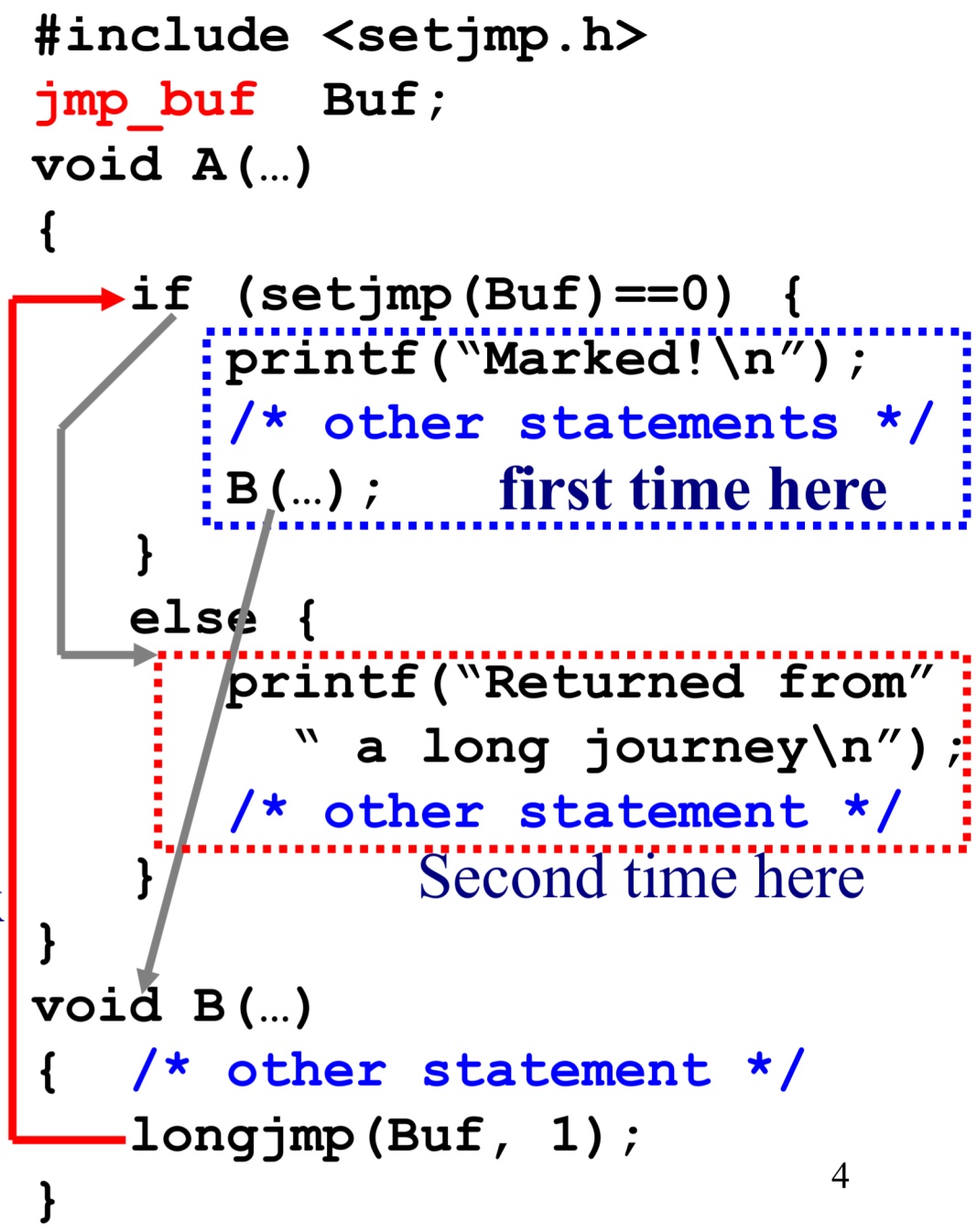
```
longjmp(JumpBuffer, 1);
```

The jump buffer is used in both calls

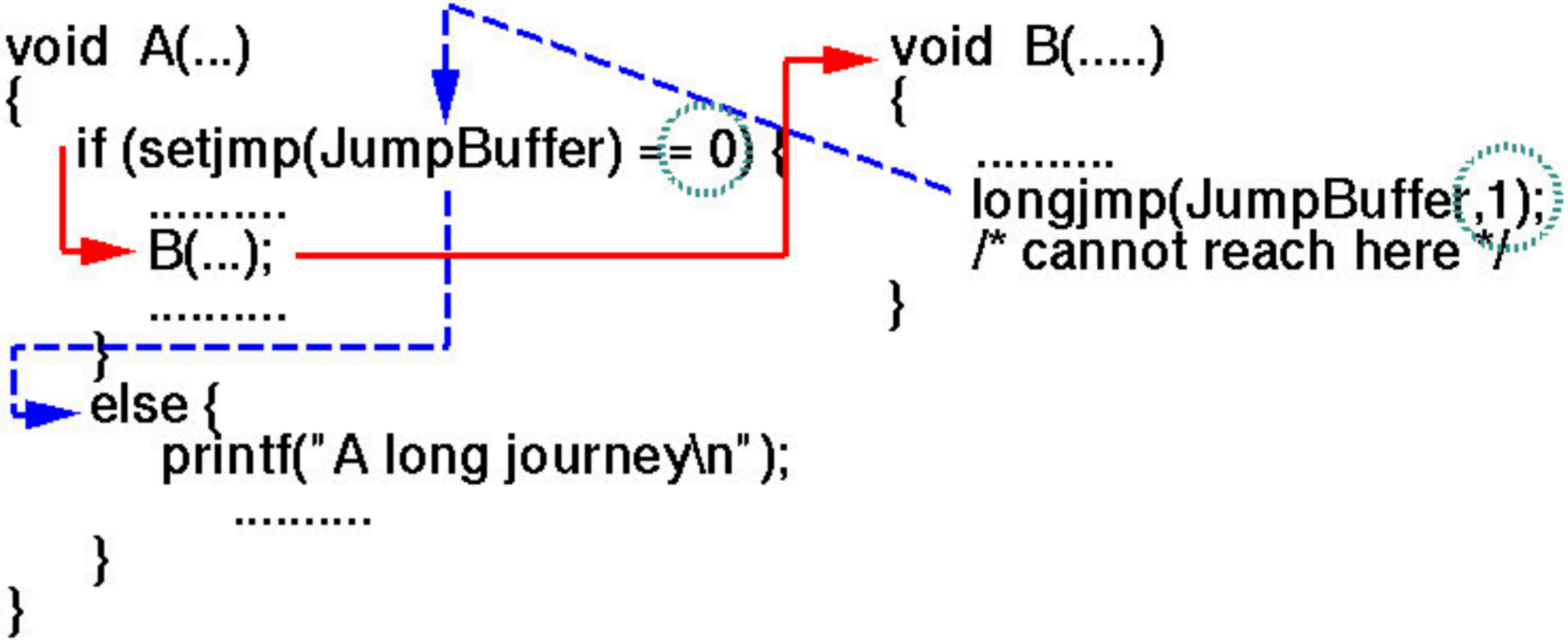
The meaning of this argument will be clear later

But, you need to know more!

- When `setjmp()` is called, it saves the current state of execution and returns `0`.
- When `longjmp()` is called, it sends the control back to a marked return point and let `setjmp()` to return its *second* argument????

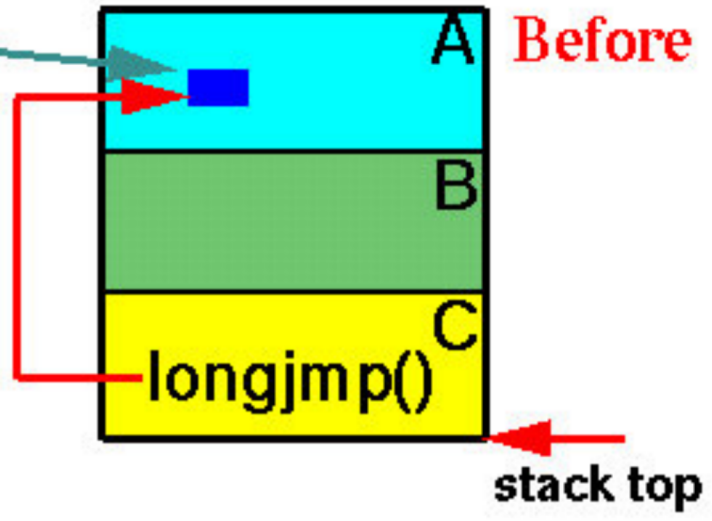


Control Flow of setjmp() and longjmp()

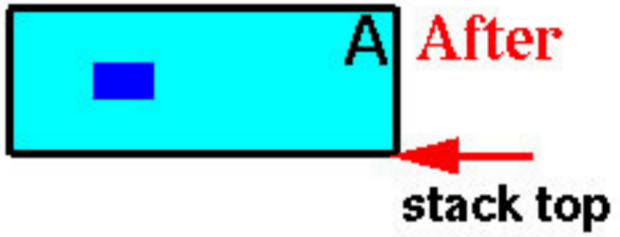


The content of a jump buffer when execute a longjmp() *must be valid*

```
jmp_buf  Buf;

void  A(...)
{
    if (setjmp(Buf) == 0)
        B(.....);
    else
        printf("I am back\n");
}

void  B(...)
{
    C(.....)
}

void  C(...)
{
    longjmp(Buf,1);
}
```



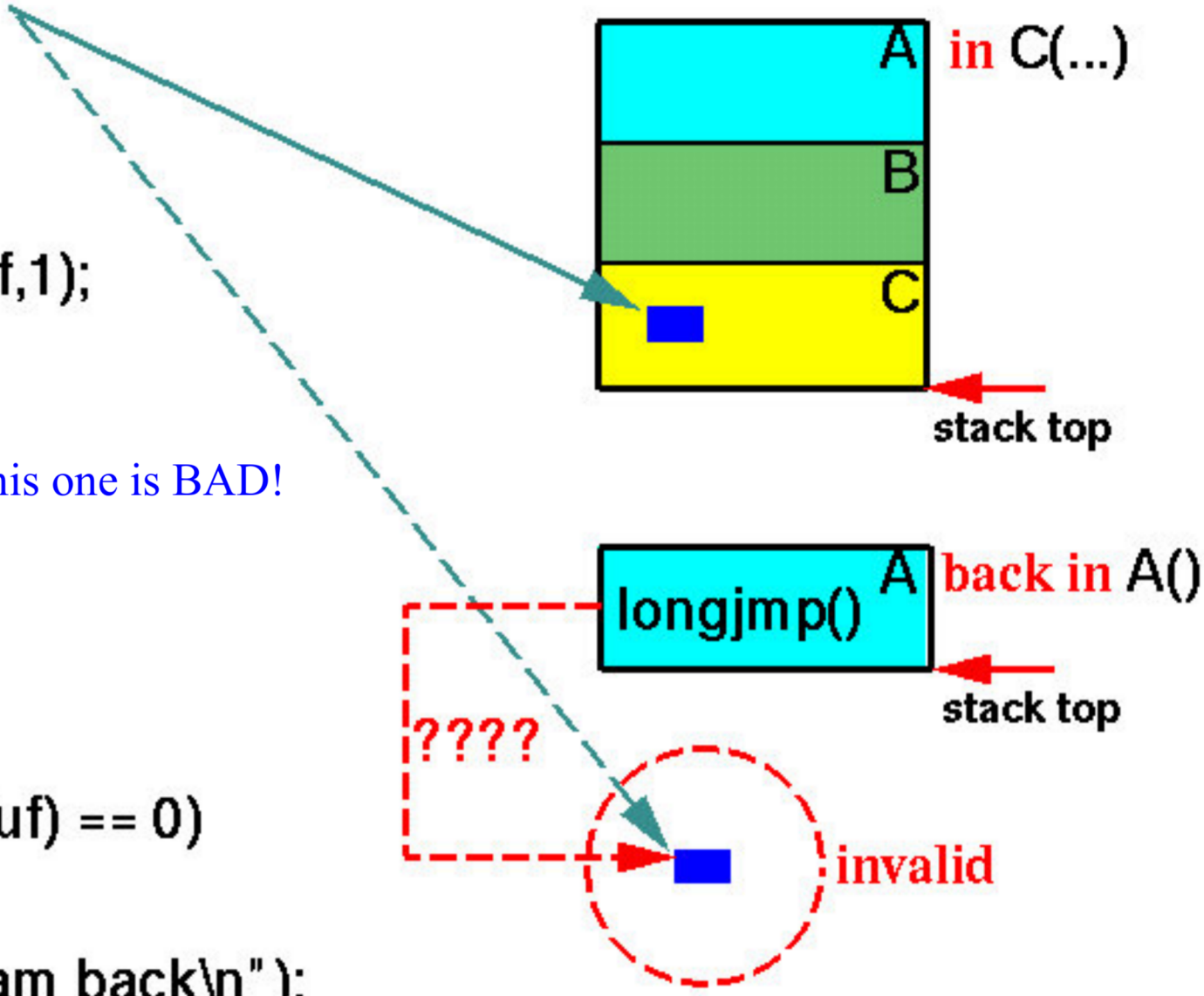
This is a good one!



The content of a jump buffer when execute a longjmp() *must be valid*

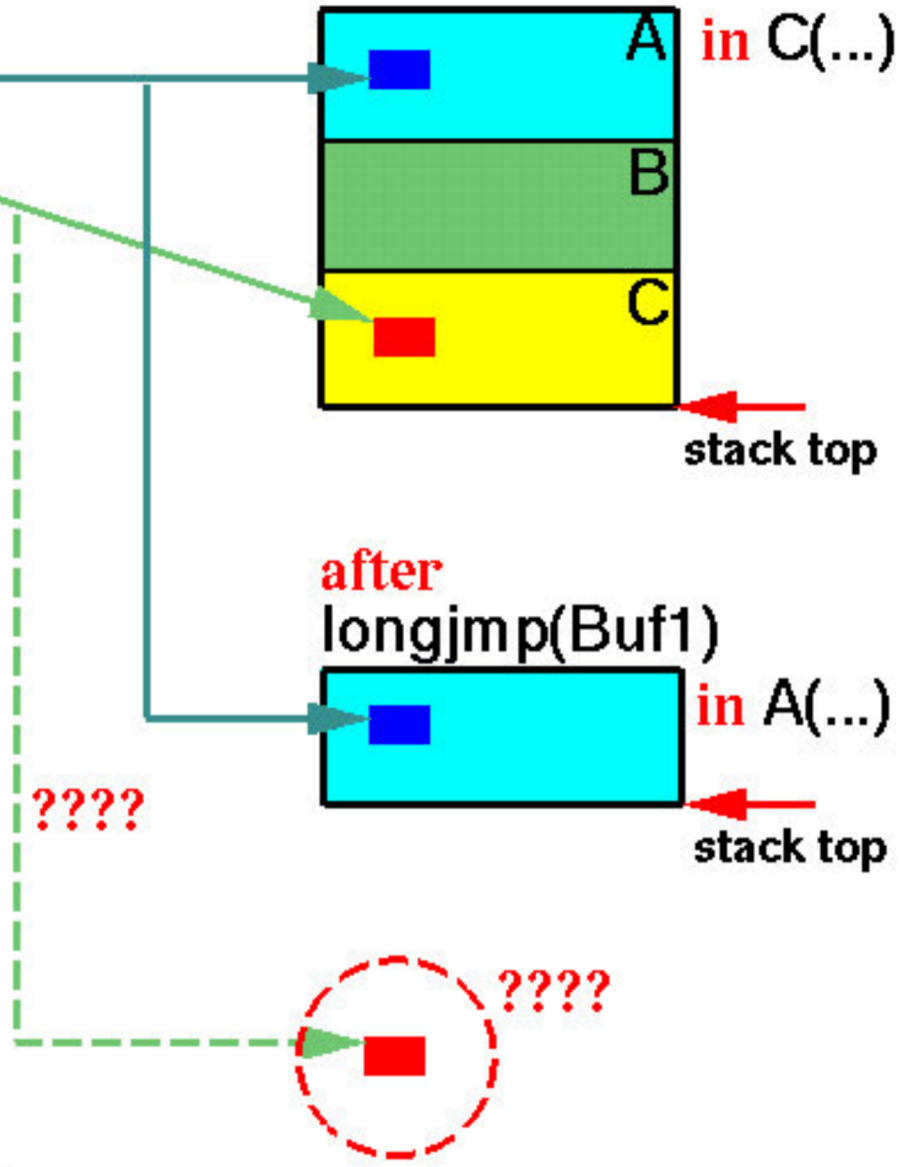
```
jmp_buf  Buf;

void  A(...)
{
    B(.....);
    longjmp(Buf,1);
}

void  B(...)
{
    C(.....)
}

void  C(...)
{
    if (setjmp(Buf) == 0)
        return;
    else
        printf("I am back\n");
}
```

This one is BAD!



The content of a jump buffer when execute a longjmp() *must be valid*

```
jmp_buf  Buf1, Buf2;

void  A(...)
{
    if (setjmp(Buf1) == 0)
        B(.....);
    else
        longjmp(Buf2, 1);
}

void  B(...)
{
    C(.....)
}

void  C(...)
{
    if (setjmp(Buf2) == 0)
        longjmp(Buf1, 1);
    else
        printf("I am back\n");
}
```



Jump Buffer Example: Factorial: 1/3

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

jmp_buf   JumpBuffer;

int  result;

void  main(int argc, char *argv[])
{
   int n;
   n = atoi(argv[1]);
   if (setjmp(JumpBuffer) == 0)
      factorial(n);
   else
      printf("%d! = %d\n", n, result);
   exit(0);
}
```

Execution will return to here!

Result is in here!

Jump Buffer Example: Factorial: 2/3

```
void  factorial(int n)
{
    fact(1, 1, n);
}

void  fact(int Result, int Count, int n)
{
    if (Count <= n)
        fact(Result*Count, Count+1, n);
    else {
        result = Result;
        longjmp(JumpBuffer, 1);
    }
}
```

result

count

limit

Why not `Count++` or `++Count`?

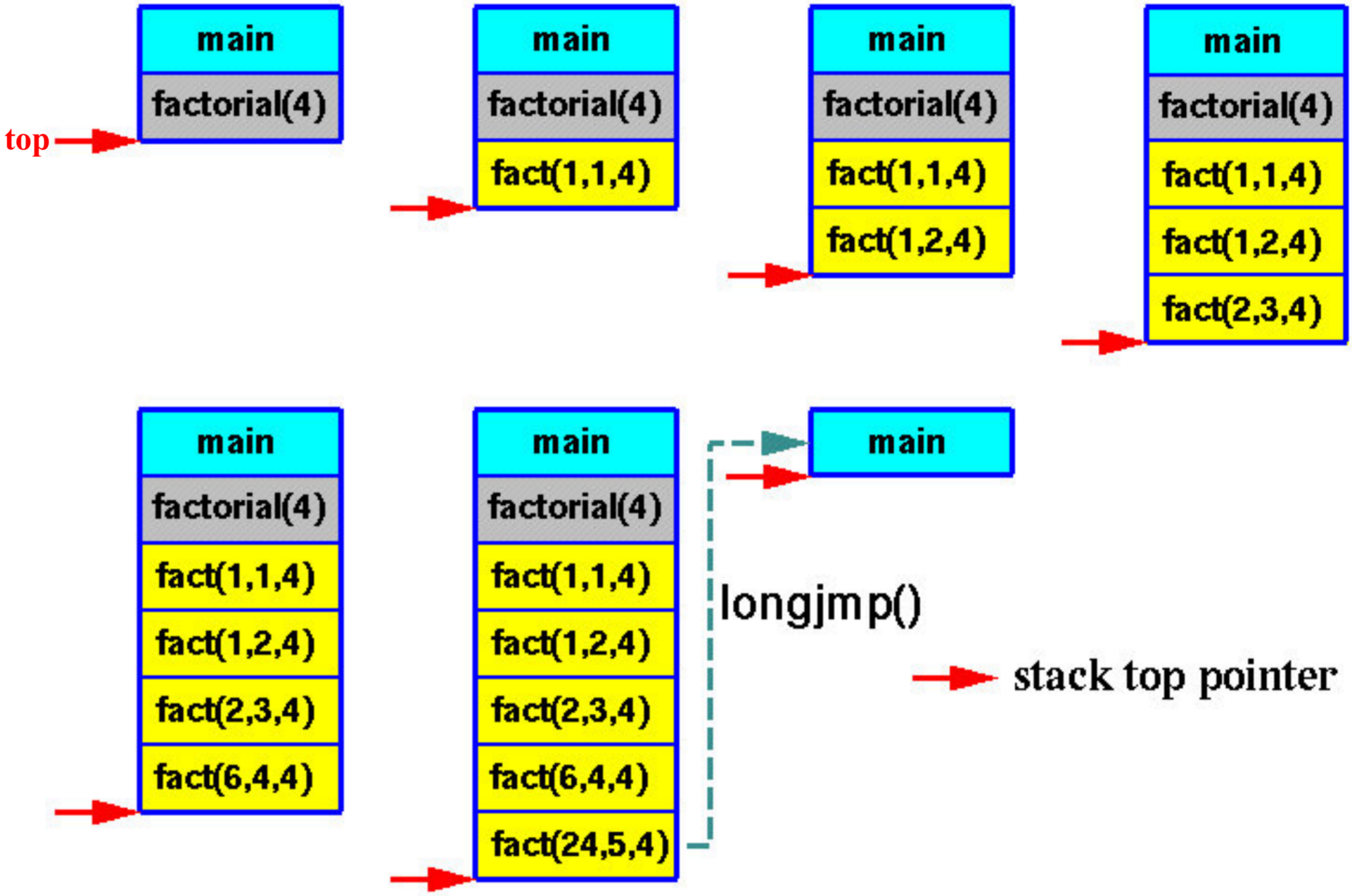
`Count++`: the value of current (rather than the next one) `Count` is passed

`++Count`: We don't know the evaluation order of the argument.

Left to Right: OK

Right to Left: Oops! The value of `Count` in `Result*Count` is wrong.

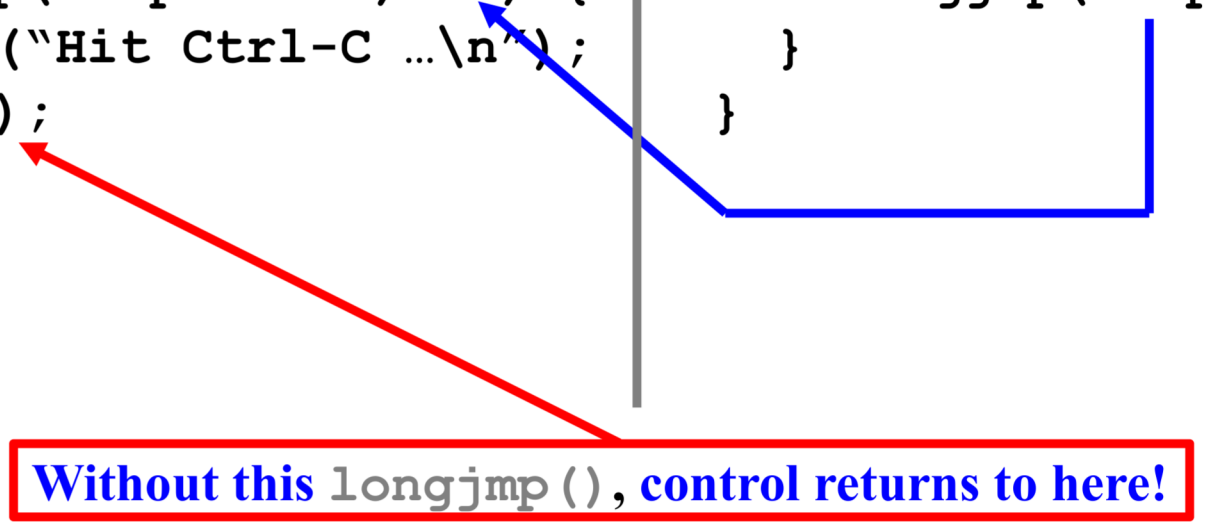
Jump Buffer Example: Factorial: 3/3



Jump Buffer Example: Signals-1

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

jmp_buf   JumpBuffer;
void      handler(int);

void main(void)
{
  signal(SIGINT, handler);
  while (1) {
    if (setjmp(JumpBuffer)==0) {
       printf("Hit Ctrl-C …\n");
       pause();
    }
  }
}
```

```
void handler(int sig)
{
  char c;
  signal(sig, SIG_IGN);
  printf("Ah, Ctrl-C?\n");
  printf("Want to quit?");
  c = getchar();
  if (c == 'y' || c == 'Y')
    exit(0);
  else {
    signal(SIGINT,handler);
    longjmp(JumpBuffer,1);
  }
}
```



Jump Buffer Example: Signals-2 (1/2)

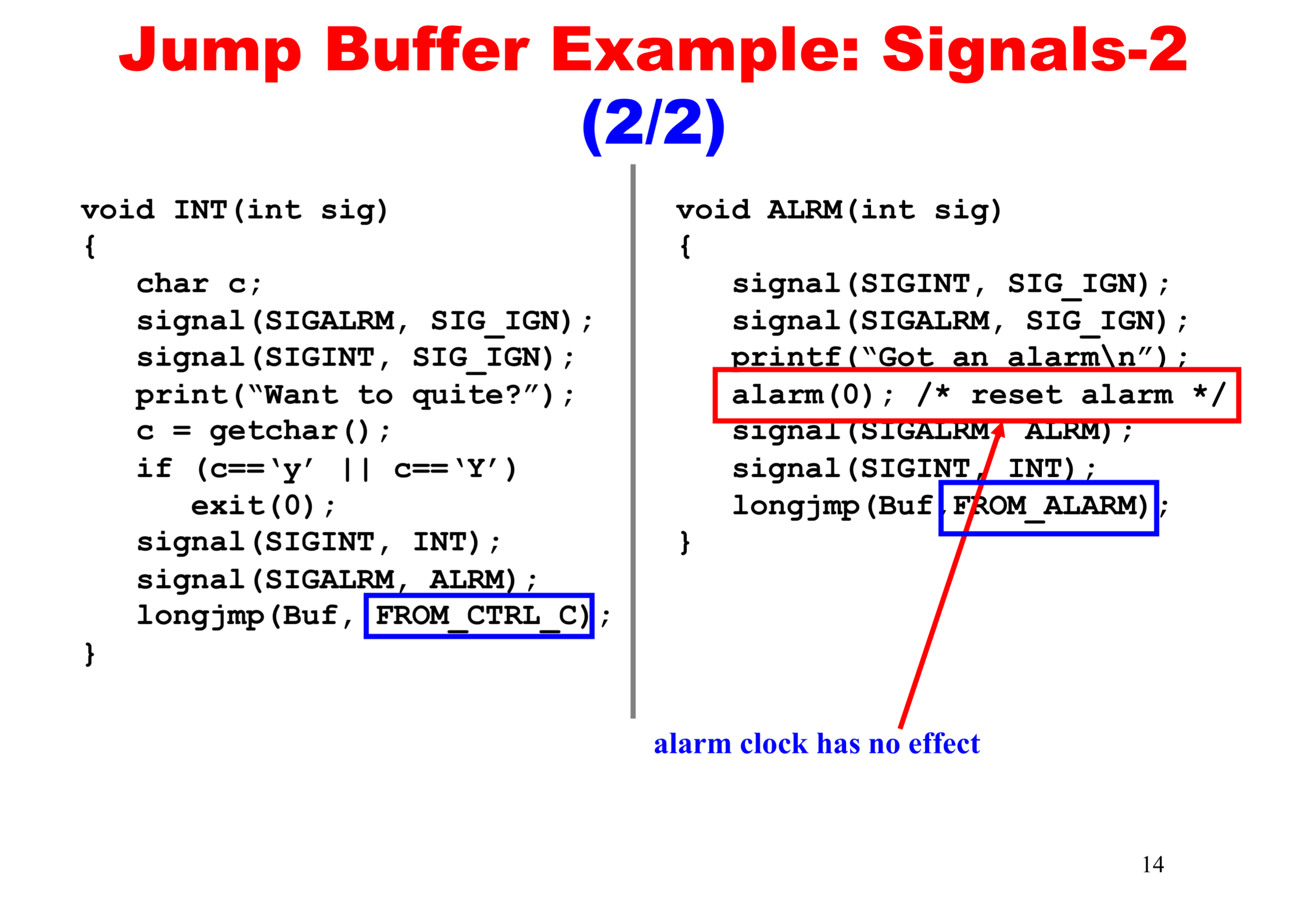
```
#define   START          0
#define   FROM_CTRL_C    1
#define   FROM_ALARM     2
#define   ALARM          5

jmp_buf   Buf;
void      INT(int);
void      ALRM(int);

Void main(void)
{
   int  Return;
   signal(SIGINT, INT);
   signal(SIGALRM, ALRM);
```

```
  while (1) {
    if ((Return=setjmp(Buf))==START) {
      alarm(ALARM);
      pause();
    }
    else if (Return == FROM_CTRL_C) {
    }
    else if (Return == FROM_ALARM) {
      print("Alarm reset to %d sec.\n",
              ALARM);
      alarm(ALARM);
    }
  }
}
```

Jump Buffer Example: Signals-2 (2/2)

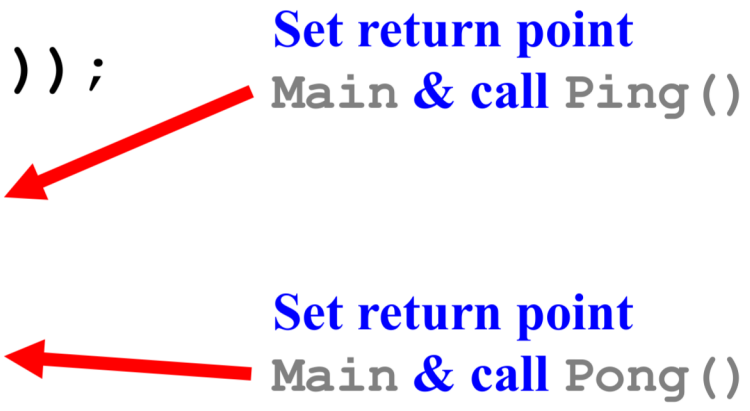
```
void INT(int sig)
{
    char c;
    signal(SIGALRM, SIG_IGN);
    signal(SIGINT, SIG_IGN);
    print("Want to quite?");
    c = getchar();
    if (c=='y' || c=='Y')
        exit(0);
    signal(SIGINT, INT);
    signal(SIGALRM, ALRM);
    longjmp(Buf, FROM_CTRL_C);
}
```

```
void ALRM(int sig)
{
    signal(SIGINT, SIG_IGN);
    signal(SIGALRM, SIG_IGN);
    printf("Got an alarm\n");
    alarm(0); /* reset alarm */
    signal(SIGALRM, ALRM);
    signal(SIGINT, INT);
    longjmp(Buf, FROM_ALARM);
}
```



A Strange Use: 1/2

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

int         max, iter;
jmp_buf     Main, PointA, PointB;
void        Ping(void), Pong(void);

void main(int argc, char *argv[])
{
    max = abs(atoi(argv[1]));
    iter = 1;
    if (setjmp(Main) == 0)
        Ping();
    if (setjmp(Main) == 0)
        Pong();
    longjmp(PointA, 1);
}
```



A Strange Use: 2/2

```
void Ping(void)
{
  if (setjmp(PointA)==0)
    longjmp(Main,1);
  while (1) {
    printf("Ping-");
    if (setjmp(PointA)==0)
      longjmp(PointB,1);
  }
}
```

```
void Pong(void)
{
  if (setjmp(PointB)==0)
    longjmp(Main,1);
  while (1) {
    printf("Pong-");
    iter++;
    if (iter > max)
      exit(0);
    if (setjmp(PointB)==0)
      longjmp(PointA,1);
  }
}
```

This program does not work if there are local variables. Why?

Output:

Ping-Pong-Ping-Pong-.....

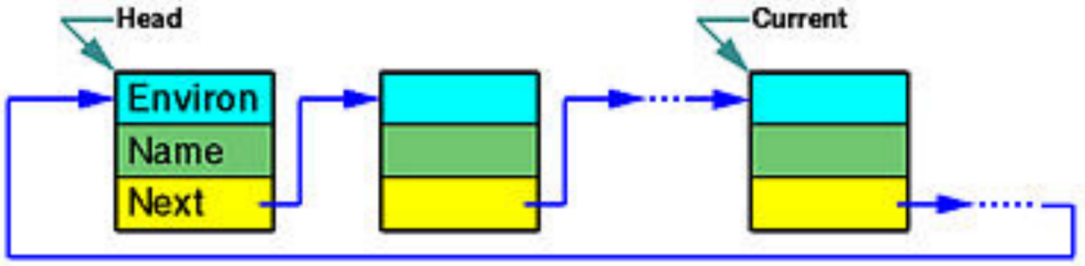
A Not-So-Correct Multithread System: 1/10

- Before going into more details, we examine a not-so-correct way to build a user-level thread system.
- First, we need a (simplified) TCB data structure.

```
typedef struct PCB_NODE *PCB_ptr; /* pointer to a PCB    */

typedef struct PCB_NODE {         /* a PCB:              */
     jmp_buf Environment;         /* jump buffer         */
     int Name;                    /* thread name: unused*/
     PCB_ptr Next;                /* next PCB.           */
} PCB;
```

A Not-So-Correct Multithread System: 2/10

- We need two more jump buffers `MAIN` and `SCHEDULER`.
- The former is used to save the main program's environment, and the latter is for the scheduler.
- Because the main program and the scheduler are not scheduled by the scheduler, they are not in the PCB list.
- There are two pointers: `Head` pointing to the head of the PCB list and `Current` to the running thread.



A Not-So-Correct Multithread System: 3/10

- The scheduler is simple.
- Initially the scheduler `Scheduler()` is called by the main program to set an entry in jump buffer `SCHEDULER` and jump back to the main program using jump buffer `MAIN` that was setup *before* the call to `Scheduler`.
- After this, we use a long jump to `SCHEDULER` rather than via function call.

```
void Scheduler(void)
{
     if (setjmp(SCHEDULER) == 0)          /* setup return point  */
          longjmp(MAIN, 1);               /* get back to main.   */
     Current = Current->Next;             /* reenter. Get next   */
     longjmp(Current->Environment, 1); /* jump to its environ */
}
```

A Not-So-Correct Multithread System: 4/10

- `THREAD_YIELD()` is very simple.
- Release CPU voluntarily.
- What we need is saving the current environment to this thread's environment (actually a jump buffer) and transferring the control to the scheduler via a `longjump`.
- Because this is so simple, we use `#define`

```
#define THREAD_YIELD(name) {                          \
      if (setjmp(Current->Environment) == 0)  \
      longjmp(SCHEDULER, 1);                          \
}
```

A Not-So-Correct Multithread System: 5/10

- `THREAD_INIT()` can be part of `THREAD_CREATE()`.
- We create and initialize a PCB, set its return point, and long jump back to the main program.

```
#define  THREAD_INIT(name) {                                  \
           work = (PCB_ptr) malloc(sizeof(PCB));              \
           work->Name = name;                                 \
           if (Head == NULL)                                  \
              Head = work;                                    \
           else                                               \
              Current->Next = work;                           \
           work->Next = Head;                                 \
           Current = work;                                    \
           if (setjmp(work->Environment) == 0)                \
              longjmp(MAIN, 1);                               \
         }
```

A Not-So-Correct Multithread System: 6/10

- `THREAD_CREATE()` is simple.
- We just set the return point of `MAIN` and call the function.

```
#define THREAD_CREATE(function, name) { \
            if (setjmp(MAIN) == 0)        \
               (function)(name);          \
            }
```

```
void main(void)
{
   Head = Current = NULL;      /* initialize pointers */
   THREAD_CREATE(funct_1, 1); /* initialize threads  */
   THREAD_CREATE(funct_2, 2);
   THREAD_CREATE(funct_3, 3);
   THREAD_CREATE(funct_4, 4);
   if (setjmp(MAIN) == 0)     /* initialize scheduler */
      Scheduler();
   longjmp(SCHEDULER,1);      /* start scheduler      */
}
```

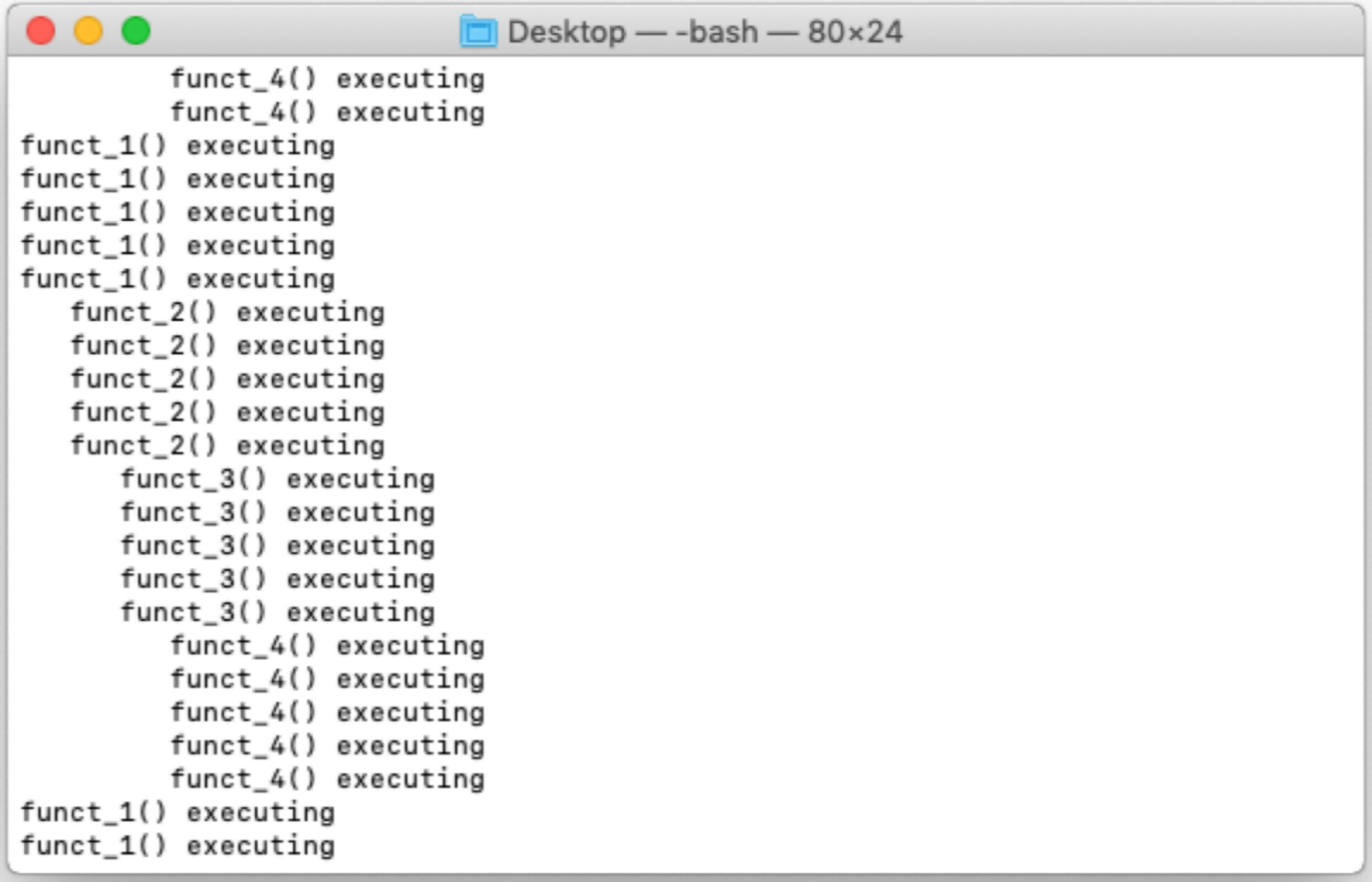
A Not-So-Correct Multithread System: 7/10

- Each function to be run as a thread must call `THREAD_INIT()`.

```
void funct_1(int name)
{
    int i;
    THREAD_INIT(name); /* initialize as thread */
    while (1) {        /* running the thread   */
        for (i = 1; i <= MAX_COUNT; i++)
            printf("funct_1() executing\n");
        THREAD_YIELD(name); /* yield control    */
    }
}
```

A Not-So-Correct Multithread System: 8/10

- This implementation appears to be correct. The following is a screenshot.



A Not-So-Correct Multithread System: 9/10

- It is **not**! ***Why?*** But you have all the ideas!
- We do not use many local variables, in fact only one variable `i`. In a function, say `funct_1()`, `i` is used before `THREAD_YIELD()`.
- Once `THREAD_YIELD()` is called, the stack frame of `funct_1()` becomes invalid. However, this is fine, because after returning from `THREAD_YIELD()` this variable is reinitialized.
- Is the jump buffer `Environment` of each thread correct? In general it is not. However, the PC is correct because it is not stored there.

A Not-So-Correct Multithread System: 10/10

- The key issue making this system not-so-correct is that each thread does not have its stack frame.
- As a result, once it long jumps out of its environment the stack frame allocated by the system becomes invalid.
- The solution is simple: allocating a separate stack frame for each “created” thread so that it won’t go away.
- This is what we intend to do.

Let Us Solve the Problem: 1/10

- We need a better TCB. `Env` is a redefined `jmpbuf`.

```
typedef struct TCB_NODE *TCB_ptr;
typedef TCB_ptr          THREAD_t;

typedef struct TCB_NODE {    /* thread control block              */
     int      Name;          /* thread ID                         */
     int      Status;        /* thread state                      */
     Env      Environment;   /* processor context area            */
     void     *StackBottom;  /* bottom of stack attached          */
     int      Size;          /* stack size                        */
     void     (*Entry)(int, char**); /* entry point (function)    */
     int      Argc;          /* # of arguments                    */
     char     **Argv;        /* argument list                     */
     Queue    JoinList;      /* joining list of threads           */
} TCB;
```

Initialize the Coroutines: 2/10

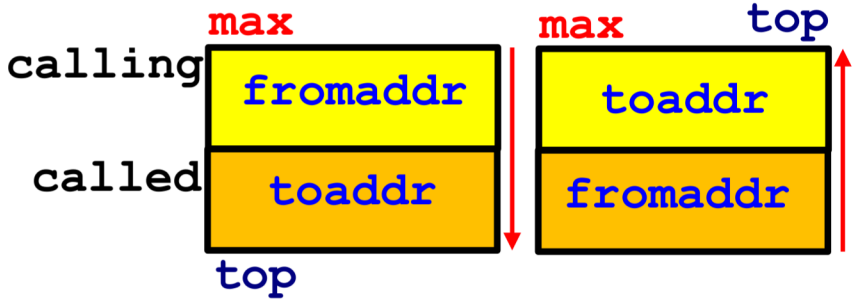
- Initialize the coroutines structure. Refer to slides used in CS3331 Concurrent Computing for the concept of coroutines.

```
static int THREAD_SYS_INIT (void)
{
   int *stack;

   Running = (THREAD_t) malloc(sizeof(TCB)); /* dummy running thread */
   stack = (int *) malloc(64);               /* 64 bytes for stack        */
   Running->Name = mtu_MAIN;                 /* dummy = main()            */
   Running->StackBottom = stack;             /* StackBottom=dummy stack   */
   ReadyQ.Head = ReadyQ.Tail = NULL;         /* initialize ready and      */
   SuspendQ.Head = SuspendQ.Tail = NULL;     /*   suspend queues          */
   SysTable.Head = SysTable.Tail = NULL;     /* initialize sys table.     */
   SYSTEM_INITIALIZE = TRUE;
   return mtu_NORMAL;
}
```

Stack Growing Direction: 3/10

- Because the user stack can grow up (stack at the highest address) or grow down (stack at the end of code and data sections), we need to know which way it goes.
- `fromaddr` is a variable in the calling function, and `toaddr` is a variable in the called function.

```
static int growsdown (void *fromaddr)
{
      int   toaddr;

      return fromaddr > (void *) &toaddr;
}
```



Wrap the Created Thread: 4/10

- `THREAD_WRAP()` wraps up the created thread and runs it as a function. The function to be run is indicated by `Running`.

```
static volatile void THREAD_WRAP (void)
{
     (*Running->Entry) (Running->Argc, Running->Argv); /* run thread*/
     THREAD_EXIT();                                    /* if user did not call     */
                                                       /* THREAD_EXIT, do it here  */
}
```

THREAD_INIT: System Dependent

- Initialize a new thread's environment.
- Newer version of gcc may not allow you to modify the jump buffer.

```
void THREAD_INIT (volatile TCB *volatile NewThread, void *StackPointer)
{
/* In Linux 1.0 the code maybe like following three lines
 *     NewThread->Environment->__sp = StackPointer;
 *     NewThread->Environment->__bp = StackPointer;
 *     NewThread->Environment->__pc = (void *)THREAD_WRAP;
 * Here is THREAD_INIT for Linux 2.0
 */
    NewThread->Environment[0].__jmpbuf[JB_SP] = (int)StackPointer;
    NewThread->Environment[0].__jmpbuf[JB_BP] = (int)StackPointer;
    NewThread->Environment[0].__jmpbuf[JB_PC] = (int)THREAD_WRAP;
}
```

THREAD_CREATE(): 1/2

- THREAD_CREATE() allocates a TCB and a stack for the thread being created and initialize the TCB.

```
THREAD_t THREAD_CREATE (void (*Entry)(), int Size, int Flag,
            int Argc, char **Argv)
{
   THREAD_t NewThread;
   int      *StackBottom, FromAddr;
   void     *StackPointer;

   NewThread = (THREAD_t )malloc (sizeof(TCB)); /* new thread TCB */
   if (NewThread == NULL)
      return (THREAD_t) mtu_ERROR;
   Size += sizeof(StackAlign);          /* get new stack size       */
   StackBottom = (int *) malloc (Size);
   StackPointer =
      (void *)(growsdown (&FromAddr)?
       (Size+(int)StackBottom)&-sizeof(StackAlign):(int)StackBottom);
   THREAD_INIT (NewThread, StackPointer);  /* initialize thread
                                              /* architecture-dependent!  */
```

-- Next Page --

THREAD_CREATE(): 2/2

- Continue from previous page.

```
    /* from previous page */
    NewThread->Name    = NextThreadName++;/* initial TCB values     */
    NewThread->Status  = mtu_READY;
    NewThread->Entry   = (void(*)(int, char**))Entry;
    NewThread->Argc    = Argc; NewThread->Argv    = Argv;
    NewThread->StackBottom = StackBottom;
    NewThread->Size    = Size;
    NewThread->JoinList.Head = NULL; NewThread->JoinList.Tail = NULL;
    THREAD_READY(NewThread);                  /* thread into Ready Q */
    THREAD_READY(Running);                    /* add new and running */
    if (Flag == THREAD_SUSPENDED)
      THREAD_SUSPEND(NewThread);
    THREAD_SCHEDULER();                       /* reschedule threads  */
    return NewThread;
}
```

THREAD_EXIT(): 1/2

- Continue from previous page.

```
int THREAD_EXIT (void)
{
     THREAD_t temp;

     if (Running->Name == mtu_MAIN) {    /* if main, exit there       */
                                         /* have no thread remain     */
          if (ReadyQ.Head != NULL)       /* in the ready Q            */
               return mtu_ERROR;
          if (SuspendQ.Head != NULL)     /* and in suspend queue      */
               return mtu_ERROR;
          return mtu_NORMAL;
     }
     while (Running->JoinList.Head != NULL) { /* check for joining   */
          temp = (THREAD_t)THREAD_Remove(&(Running->JoinList));
          temp->Status = mtu_READY;           /* make them ready      */
          THREAD_Append(&ReadyQ, (void *)temp);
     }         /* continue to next page */
```

THREAD_EXIT(): 2/2

- Continue from previous page.

```
    Running->Name = mtu_INVALID;          /* set current thread's TCB */
    Running->Status = mtu_TERMINATED;     /* and status = terminated  */
    Running = NULL;
    THREAD_SCHEDULER();                   /* run next thread          */
    return mtu_ERROR;                     /* shouldn't reach here     */
}
```

THREAD_YIELD()

- THREAD_YIELD() **puts the running thread back to** READY **and calls** THREAD_SCHEDULER() **to reschedule.**

```
void THREAD_YIELD (void)
{
    THREAD_READY(Running);   /* put the running one to Ready */
    THREAD_SCHEDULER();      /* ask scheduler to reschedule. */
}
```

THREAD_SCHEDULE(): 1/2

- THREAD_SCHEDULE() **finds and runs the next ready thread.**

```
static int THREAD_SCHEDULER (void)
{
     THREAD_t volatile Nextp;
     Nextp = (THREAD_t) THREAD_Remove (&ReadyQ); /* find a thread   */
     if (Nextp == NULL) {                        /* if ready queue is empty  */
          mtuMTP_errno=mtuMTP_DEADLOCK; /* a deadlock may occur     */
          ShowDeadlock();
          exit(0);
          return mtu_ERROR;
     }
     if (Running==NULL) {                        /* if running thread exited */
          Running = Nextp;                       /* let the next run         */
          Nextp->Status = mtu_RUNNING;  /* its status = RUNNING     */
          RestoreEnvironment (Running->Environment);  /* restore env*/
     }
     /* continue to next page */
```

THREAD_SCHEDULE(): 2/2

- THREAD_SCHEDULE() **finds and runs the next ready thread.**

```
    /* continue from previous page */
    if ((Running != Nextp) &&
        (SaveEnvironment(Running->Environment) == 0)) {
                                        /* else save running's env  */
         Running = Nextp;               /* let next thread run      */
         Nextp->Status = mtu_RUNNING;   /* its status = RUNNING     */
         RestoreEnvironment (Running->Environment); /* restore env */
    }
    return mtu_NORMAL;
}
```

Dining Philosophers: 1/2

```
void Philosopher(int No)
{
     int   Left = No;                    /* left/right fork numbers  */
     int   Right = (No + 1) % PHILOSOPHERS;
     int   RandomTimes, i, j;
     char  spaces[PHILOSOPHERS*2+1];

     for (i = 0; i < 2*No; i++)          /* build leading spaces     */
          spaces[i] = ' ';
     spaces[i] = '\0';
     printf("%sPhilosopher %d starts\n", spaces, No);
     for (i = 0; i < Iteration; i++) {
          printf("%sPhilosopher %d is thinking\n", spaces, No);
          SimulatedDelay();               /* think for a while        */
          SEMAPHORE_WAIT(Seats);          /* wait for a seat          */
               printf("%sPhilosopher %d has a seat\n", spaces, No);
               MUTEX_LOCK(Chopstick[Left]);       /* wait left chop */
               MUTEX_LOCK(Chopstick[Right]);      /* wait right one */
               printf("%sPhilosopher %d gets chopsticks and eats\n", spaces, No);
               SimulatedDelay();          /* eat for a while          */
               printf("%sPhilosopher %d finishes eating\n", spaces, No);
               MUTEX_UNLOCK(Chopstick[Left]);      /* release chops */
               MUTEX_UNLOCK(Chopstick[Right]);
          SEMAPHORE_SIGNAL(Seats);       /* release chair            */
     }
     THREAD_EXIT();
}
```

Dining Philosophers: 2/2

```
int main(int argc, char *argv[])
{
     THREAD_t Philosophers[PHILOSOPHERS];
     int      SeatNo[PHILOSOPHERS];
     int      i;

     Iteration = abs(atoi(argv[1]));
     srand((unsigned int)time(NULL));   /* initialize random number */
     for (i = 0; i < PHILOSOPHERS; i++) /* create mutex locks       */
          Chopstick[i]= MUTEX_INIT();
     Seats =  SEMAPHORE_INIT(PHILOSOPHERS-1);     /* seat semaphore */
     for (i = 0; i < PHILOSOPHERS; i++) {    /* create philosophers */
          SeatNo[i] = i;                     /* philosopher number  */
          Philosophers[i] =                  /* create a thread     */
               THREAD_CREATE(Philosopher,    /* the thread function */
               THREAD_SIZE,                  /* stack size          */
               THREAD_NORMAL,                /* the thread flag     */
               SeatNo[i],                    /* play a trick here   */
               (char **)0);                  /* no argument list    */
          if (Philosophers[i] == (THREAD_t)mtu_ERROR) { /* if failed*/
               printf("Thread creation failed.\n");    /* exit     */
          }
     }
     for (i=0; i<PHILOSOPHERS; i++)     /* wait until all done      */
          THREAD_JOIN(Philosophers[i]);

    return 0;
}
```

Conclusions

- In a kernel, the kernel has to allocate a stack differently.
- Context switch has to be done differently and directly rather than using a jump buffers.
- The remaining should be very similar and could be copied easily.
- The not-so-correct system is discussed here: `http://www.csl.mtu.edu/cs4411.ck/www/NOTES/non-local-goto/index.html`
- A simple user-level thread system is in the `common` directory `mtuThread.tar.gz`. Also refer to this page: `http://www.csl.mtu.edu/cs4411.ck/www/PROG/PJ/proj.html`

The End