STUDENTS SUGGESTED REVIEWING TOPICS

Topics

1. 2-level paging
2. Adding hexadecimal numbers
3. Generating PTR entry address
4. Message queue
5. Multithreading model
6. Synchronization Mechanisms

2-level Paging

Page Table **Sizes**

- Page size = 2^N **bytes**
- Number of page table entries = $2^{(\text{Address bits} - N)}$
- Page table size = **Number of page table entries** x **sizeof(1 page table entry)**

Example:

- Let the number of Address bits be **32** (virtual address bits)
- Let the Page size be **4KB** = 2^{12} **bytes.**
- Then, number of Page Table Entries (PTE) = $2^{(\text{Address bits} - N)} = 2^{32-12} = 2^{20}$
- Let one page table entry size = **4 bytes**
- **So, the Page Table Size** = 2^{20} **x 4 bytes = 4MB**

In a **1-level paging scheme**, each process needs a page table size of **4MB** in physical memory!!! Can we do better?

2-LEVEL PAGING BASIC IDEA

Original Page Table (4MB)

Need **4MB** in Main Memory per process

Example:

- Let the number of Address bits be **32** (virtual address bits)
- Let the Page size be **4KB** = 2^{12} **bytes.**
- Then, number of Page Table Entries (PTE) = $2^{(\text{Address bits} - N)} = 2^{32-12} = 2^{20}$
- Let one page table entry size = **4 bytes**
- **So, the Page Table Size** = 2^{20} **x 4 bytes = 4MB**

In a **1-level paging scheme**, each process needs a page table size of 4MB in physical memory!!! Can we do better?

We can break the Original Page Table down into **multiple** "smaller pages" !!!.

2-LEVEL PAGING BASIC IDEA

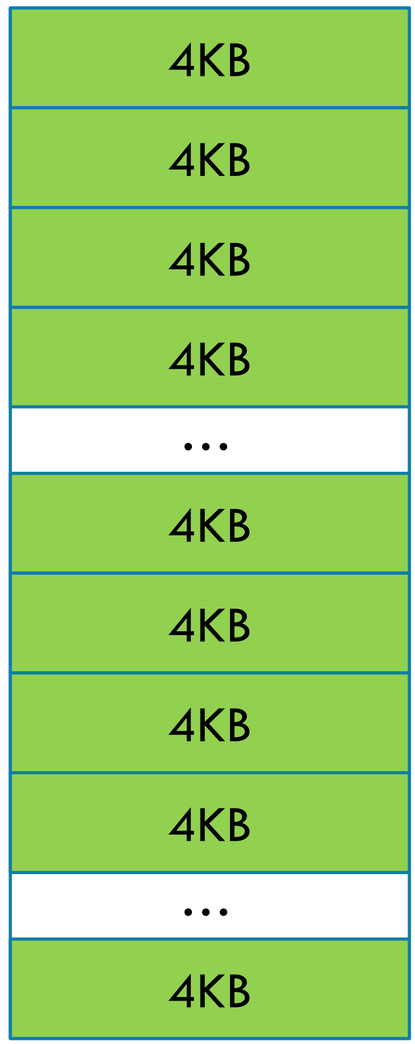
Example:

- Let the number of Address bits be **32** (virtual address bits)
- Let the Page size be **4KB** = **2^{12} bytes.**
- Then, number of Page Table Entries (PTE) = $2^{(\text{Address bits} - N)}$ = $\mathbf{2^{32-12} = 2^{20}}$
- Let one page table entry size = **4 bytes**
- **So, the Page Table Size** = **2^{20}x 4 bytes = 4MB**

Each process needs a page table size of **4MB** in physical memory!!! Can we do better?

We can break the Original Page Table down into multiple "smaller pages", each of 4KB size !!!.

Original Page Table (4MB)



Let's think about how each "smaller page" (**size of 4KB**) of original page table entries now matches the logical address space.

- **4MB** consists of **1K** of **4KB** pages i.e., 4 MB consists of 1K pages.

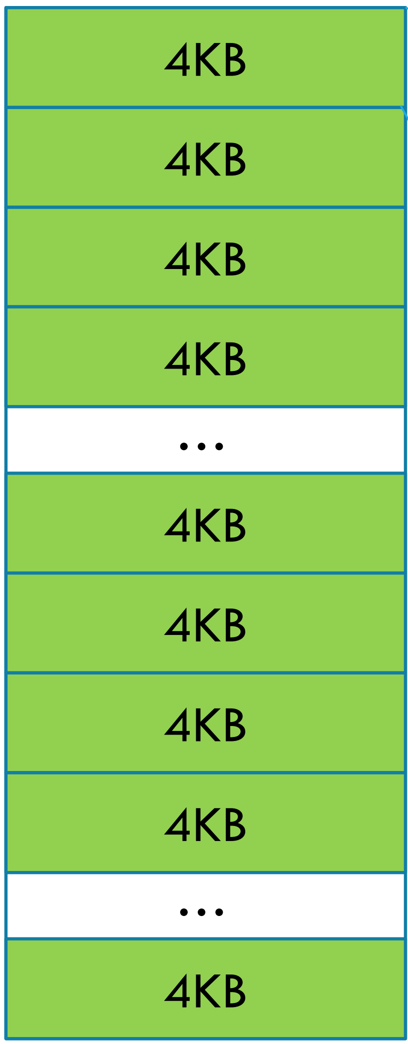
2-LEVEL PAGING BASIC IDEA

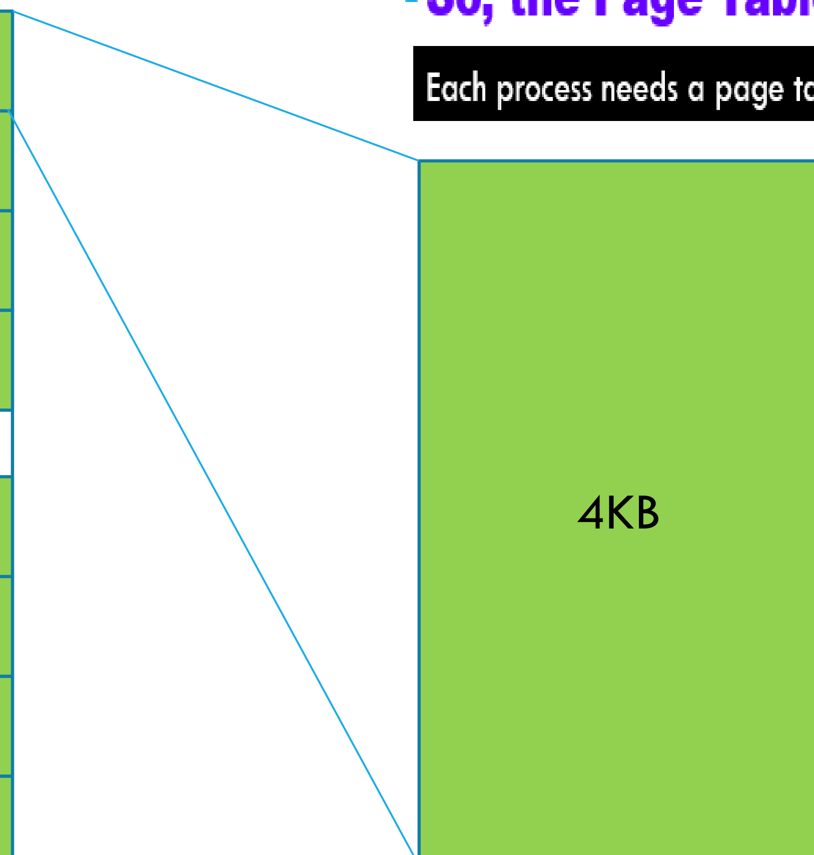
Example:

- Let the number of Address bits be **32** (virtual address bits)
- Let the Page size be **4KB = 2^{12} bytes.**
- Then, number of Page Table Entries (PTE) = $2^{(\text{Address bits} - N)} = 2^{32-12} = 2^{20}$
- Let one page table entry size = **4 bytes**
- **So, the Page Table Size** = 2^{20} **x 4 bytes = 4MB**

Each process needs a page table size of 4MB in physical memory!!! Can we do better?

Original Page Table (4MB)





One smaller page (4KB) of original page table entries

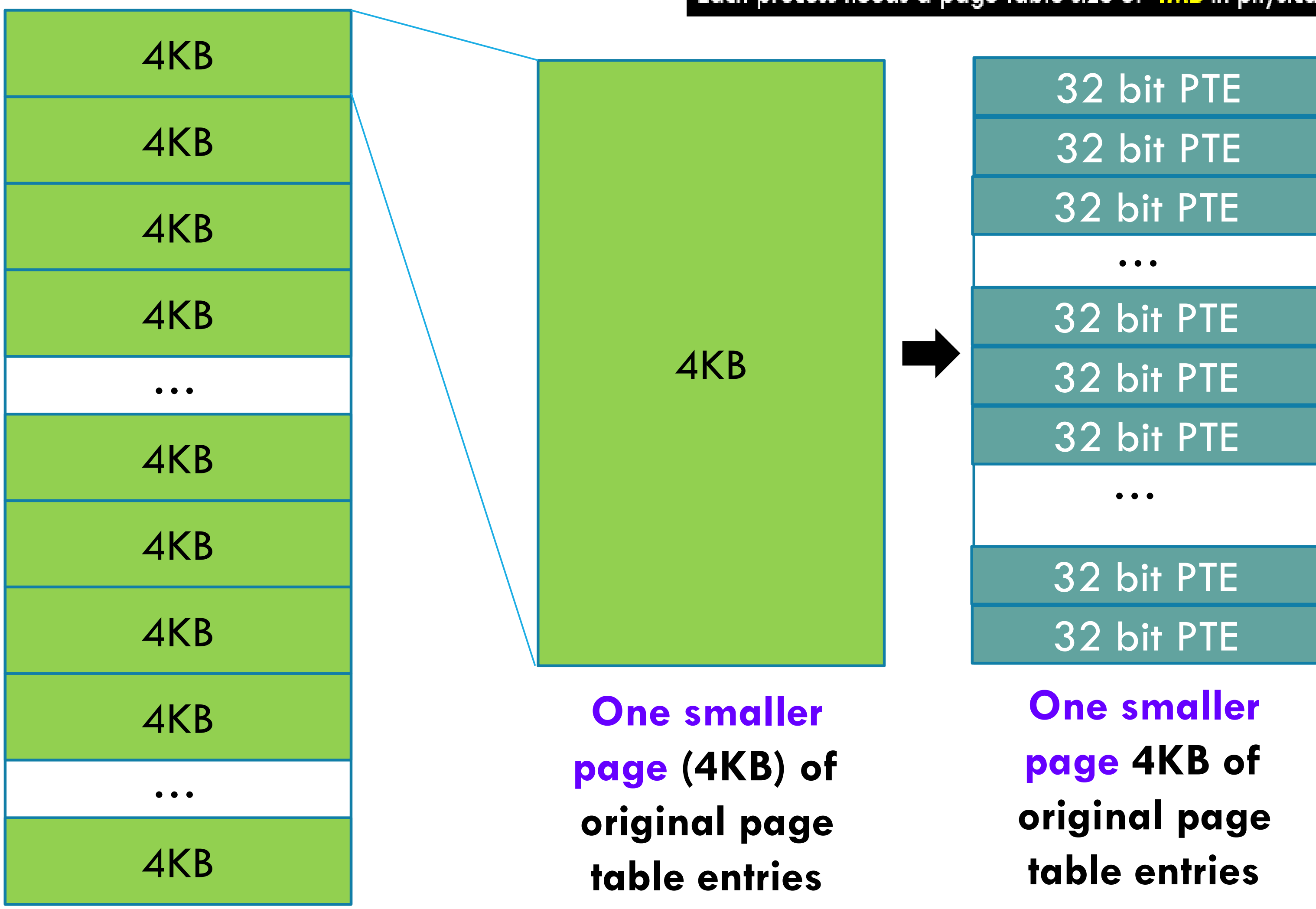
2-LEVEL PAGING BASIC IDEA

Example:

- Let the number of Address bits be **32** (virtual address bits)
- Let the Page size be **4KB** = 2^{12} **bytes.**
- Then, number of Page Table Entries (PTE) = $2^{(\text{Address bits} - N)} = 2^{32-12} = 2^{20}$
- Let one page table entry size = **4 bytes**
- **So, the Page Table Size** = 2^{20} **x 4 bytes = 4MB**

Each process needs a page table size of 4MB in physical memory!!! Can we do better?

Original Page Table (4MB)



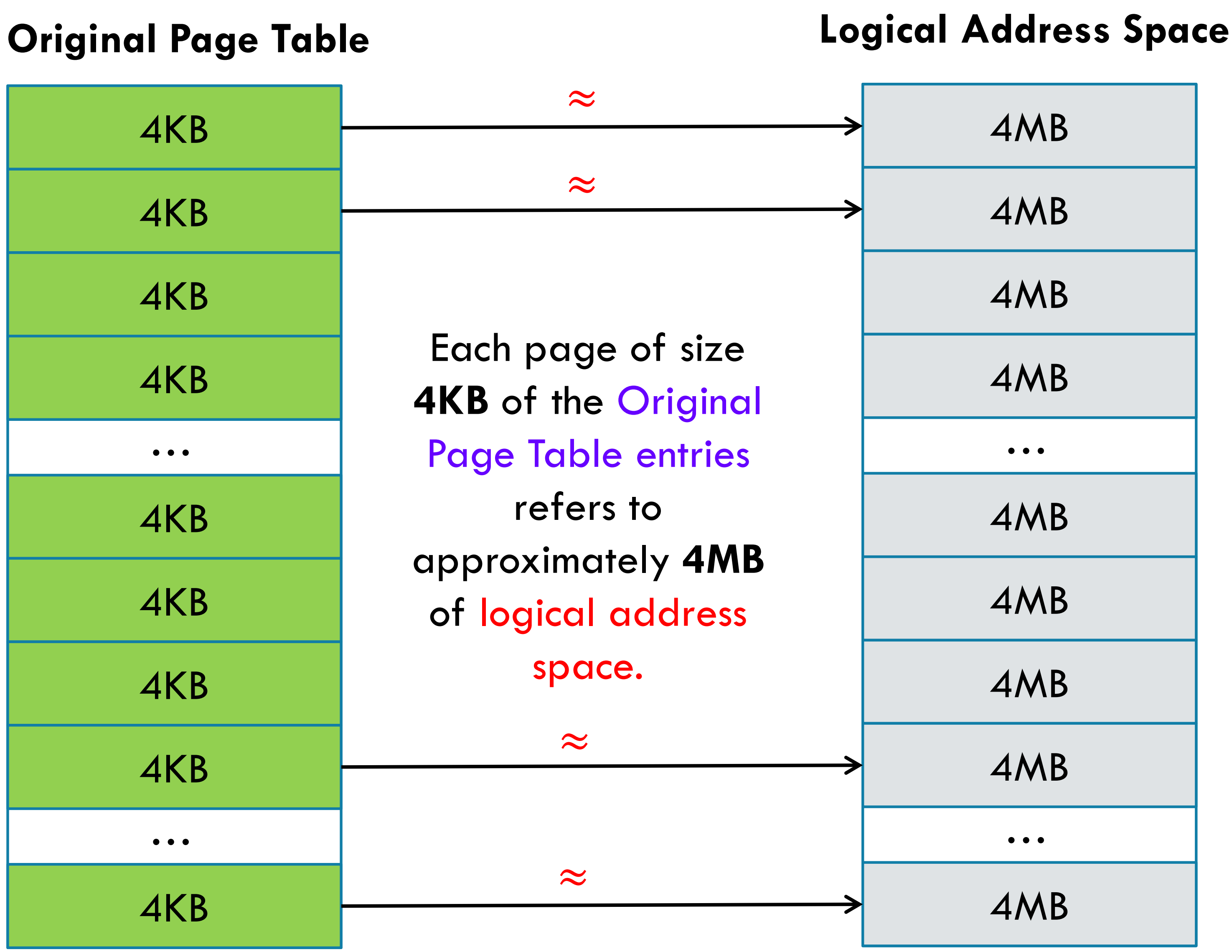
One smaller page (4KB) of original page table entries

One smaller page 4KB of original page table entries

- **One smaller page (4KB)** of Original Page Table entries is 2^{10} = **1024**, 32-bit Page Table Entries **(PTE).**
- Recall that each Original Page Table Entry refers to a page of size **4KB.**

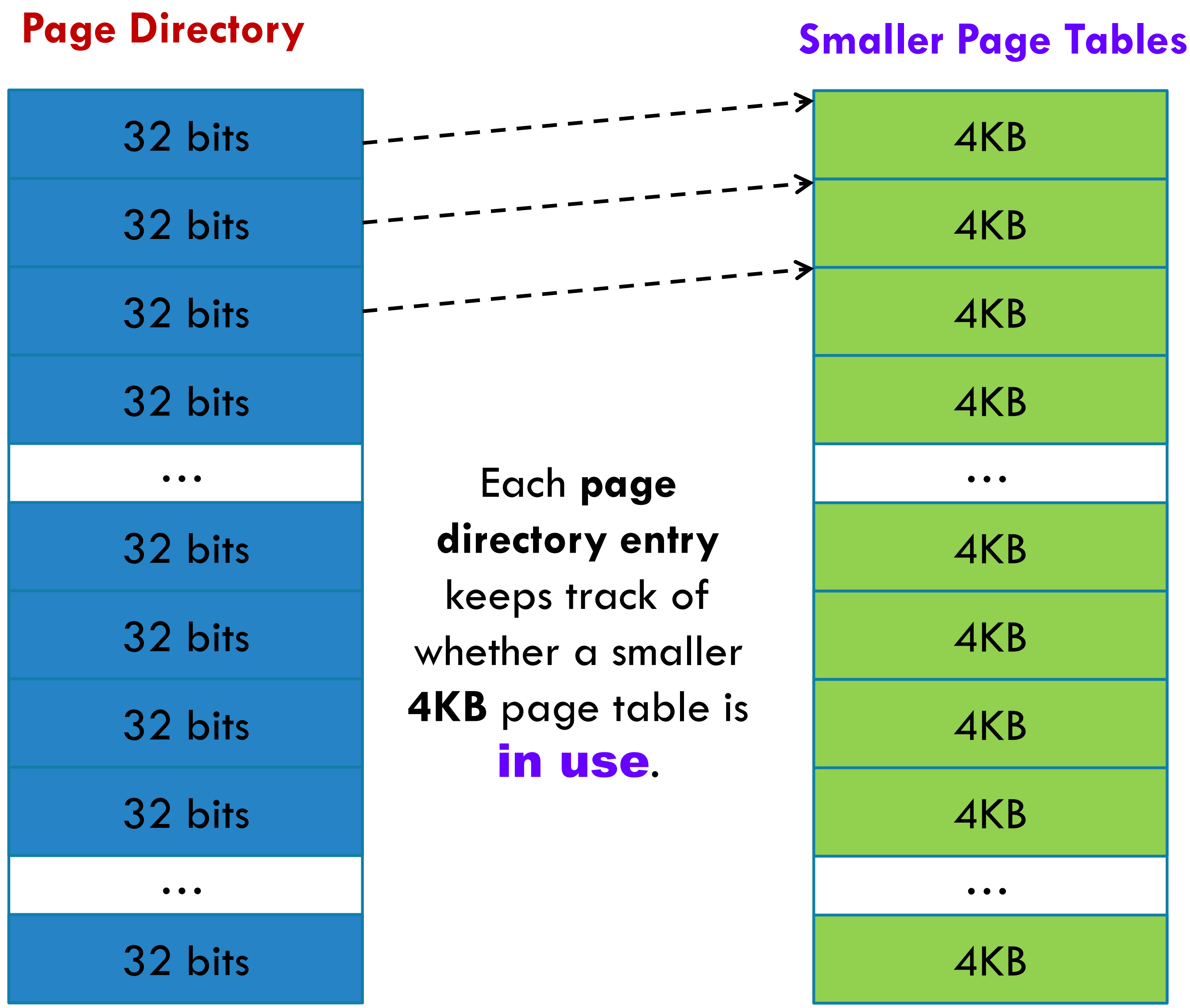
So **1024 Original Page Table entries** *each referring to a page size* of **4KB** refers to **4MB** of logical memory !!!

RELATIONSHIP BETWEEN THE PAGE TABLE AND THE LOGICAL ADDRESS SPACE

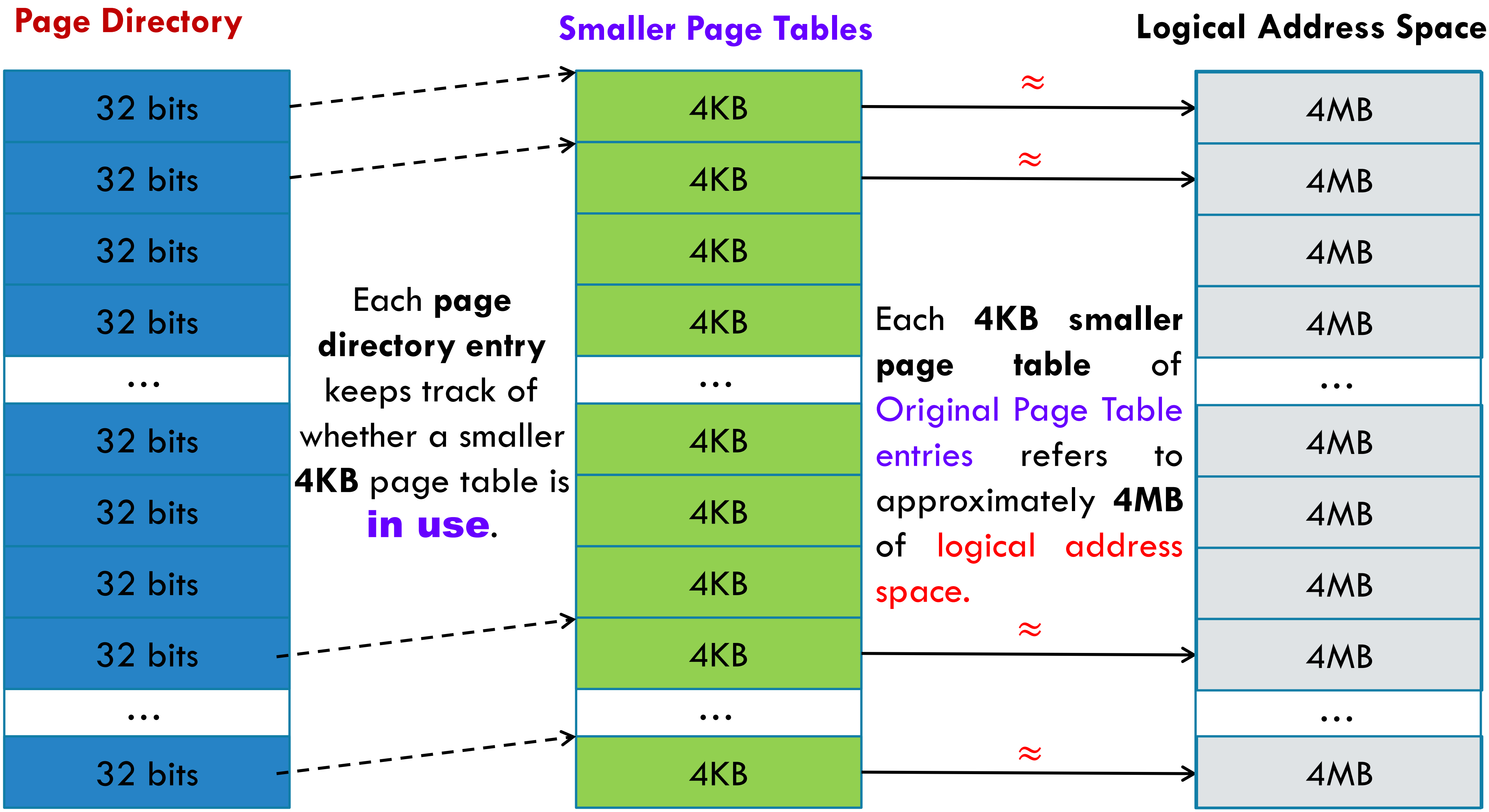


SOLUTION: 2–LEVEL PAGING (VIRTUALIZE THE PAGE TABLE)

1. Break the **Original Page Table** into smaller page tables of **4KB** size each.
2. Original Page Table size = **4MB** = **1K** smaller page tables of **4KB** each.
3. Use a **page directory** of size **1K** to keep track of the **starting address** of each smaller page table.

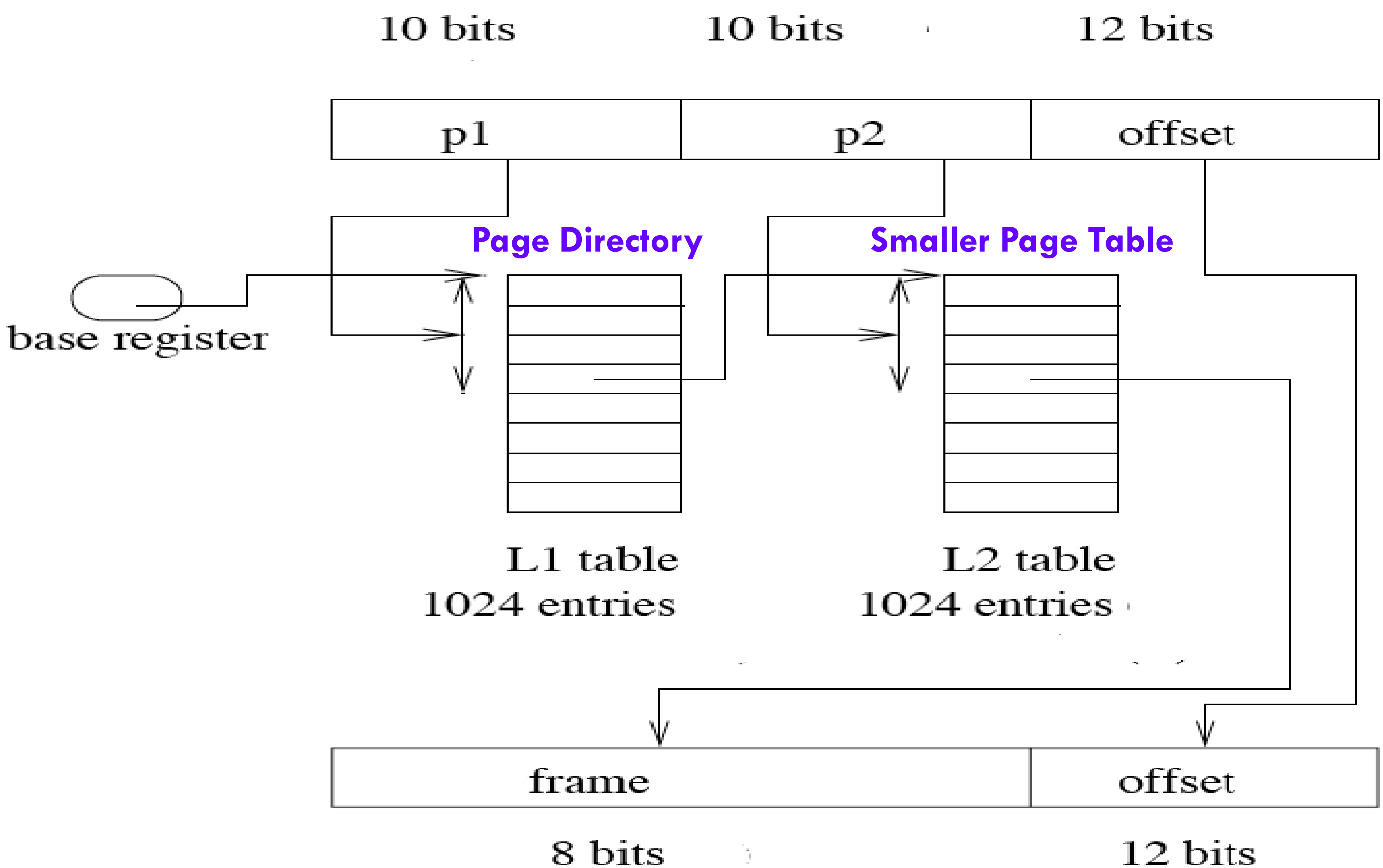


2-LEVEL PAGING: OVERVIEW



2-LEVEL PAGING VIEW: EXAMPLE

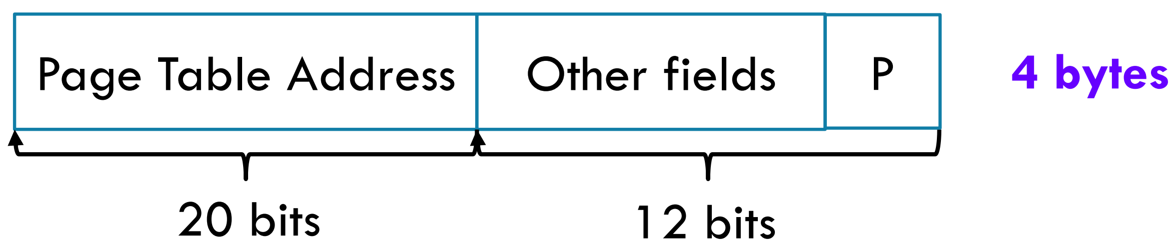
* Page directory → outer page table
* Smaller Page Table →inner page table



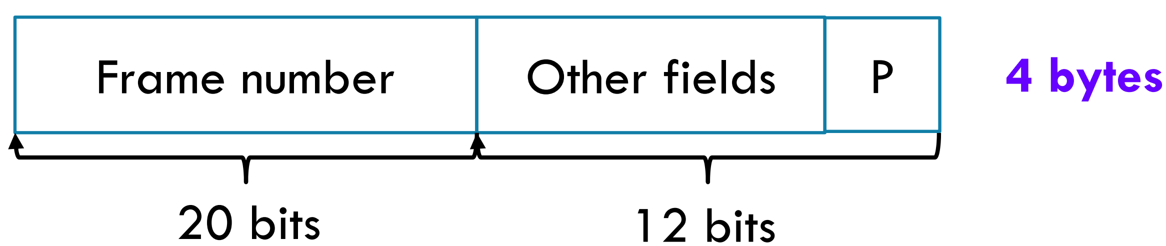
PAGE DIRECTORY ENTRY

Page Directory Entry looks the same as a **page table entry.** Instead of the frame number, we have the starting address of the page table.

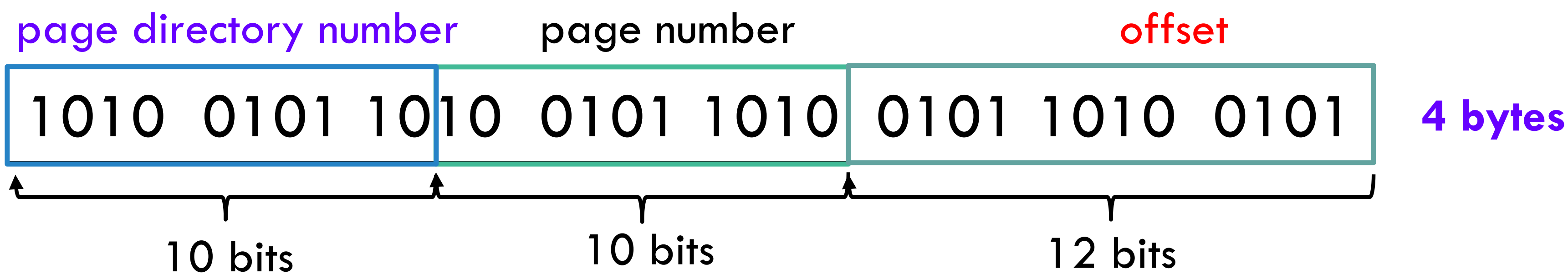
Page Directory Entry



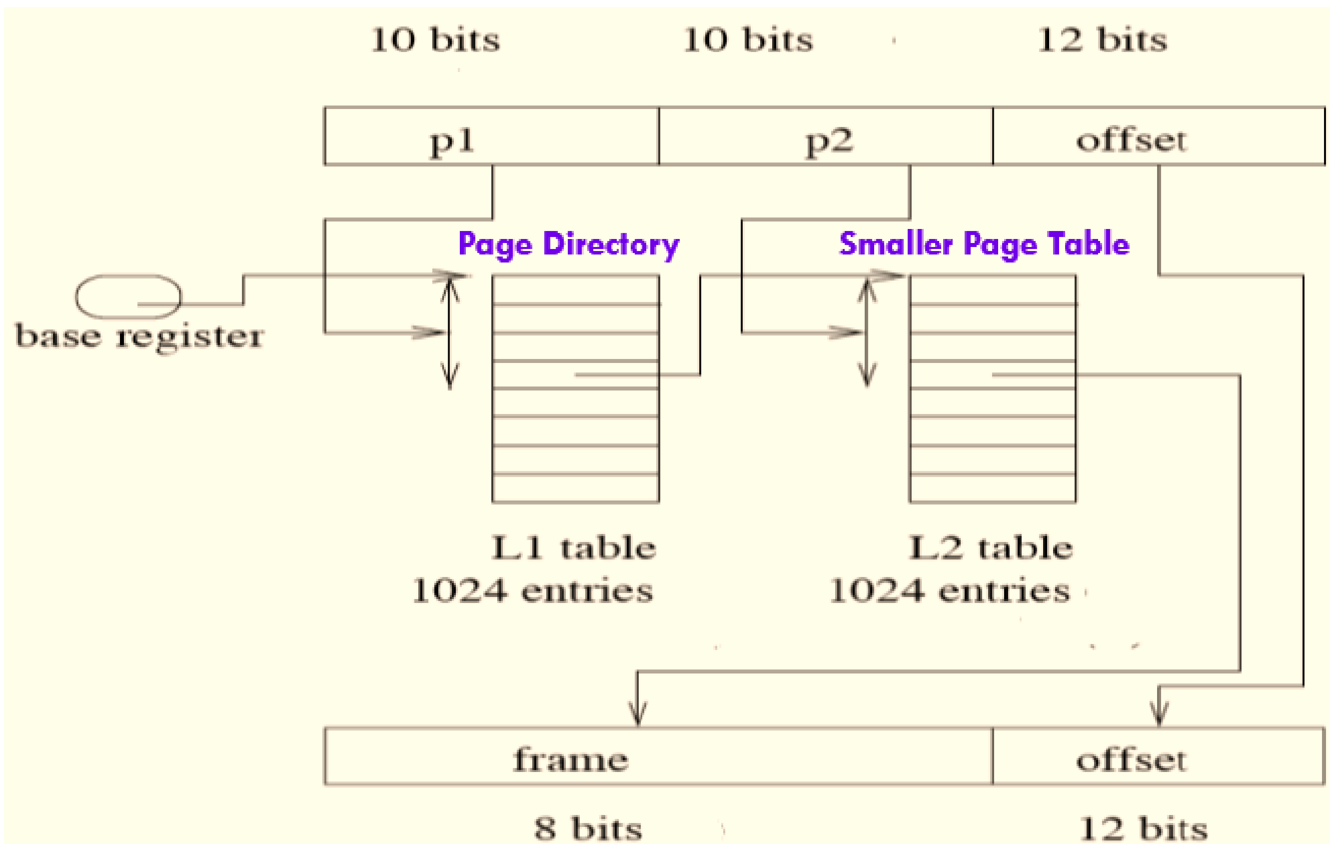
Page Table Entry



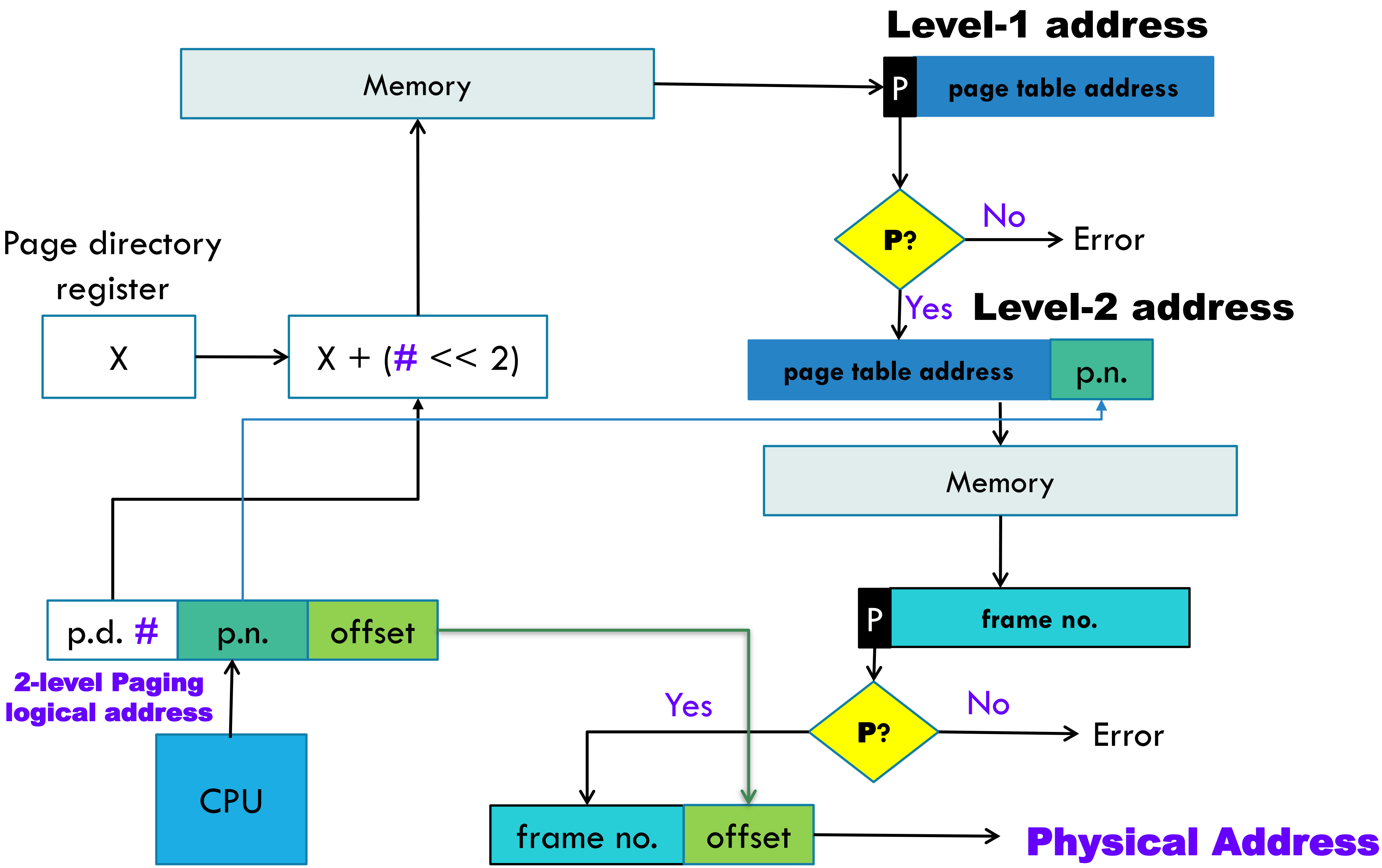
2-LEVEL PAGING LOGICAL ADDRESS



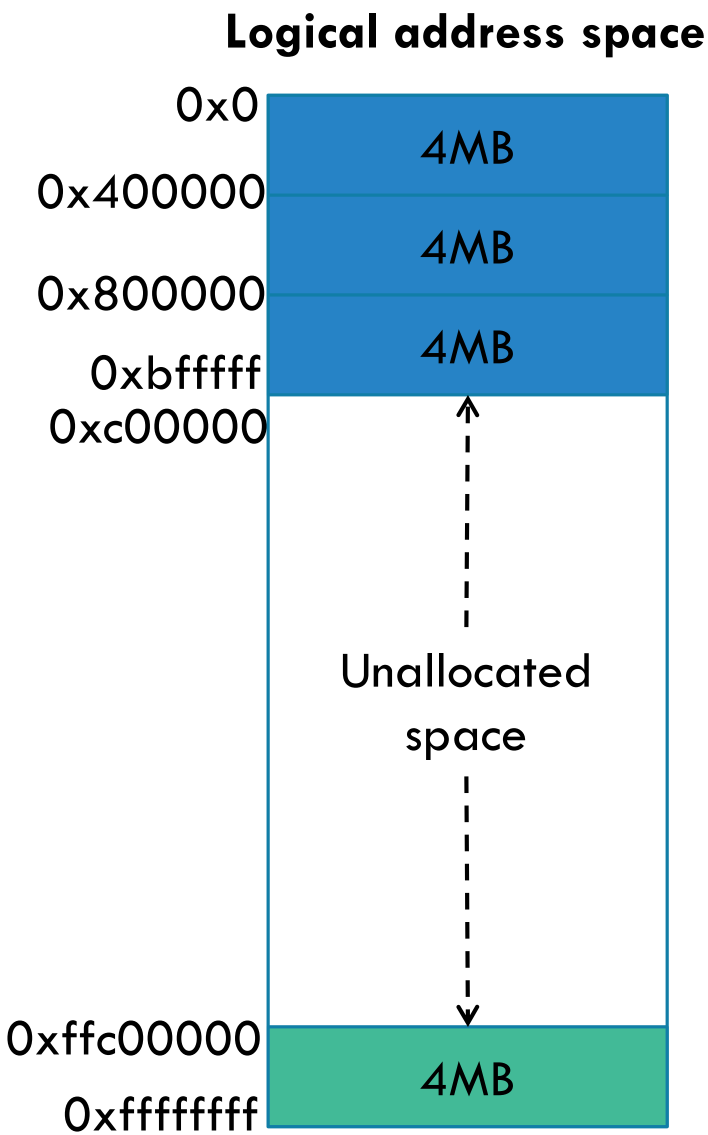
- Now instead of just **2 parts.** The logical address consists of **3 parts**.



2-LEVEL PAGING LOGICAL ADDRESS TRANSLATION



AN EXAMPLE FOR 2-LEVEL PAGING TO SHOW THE SIZE OF PAGE TABLES IN RAM

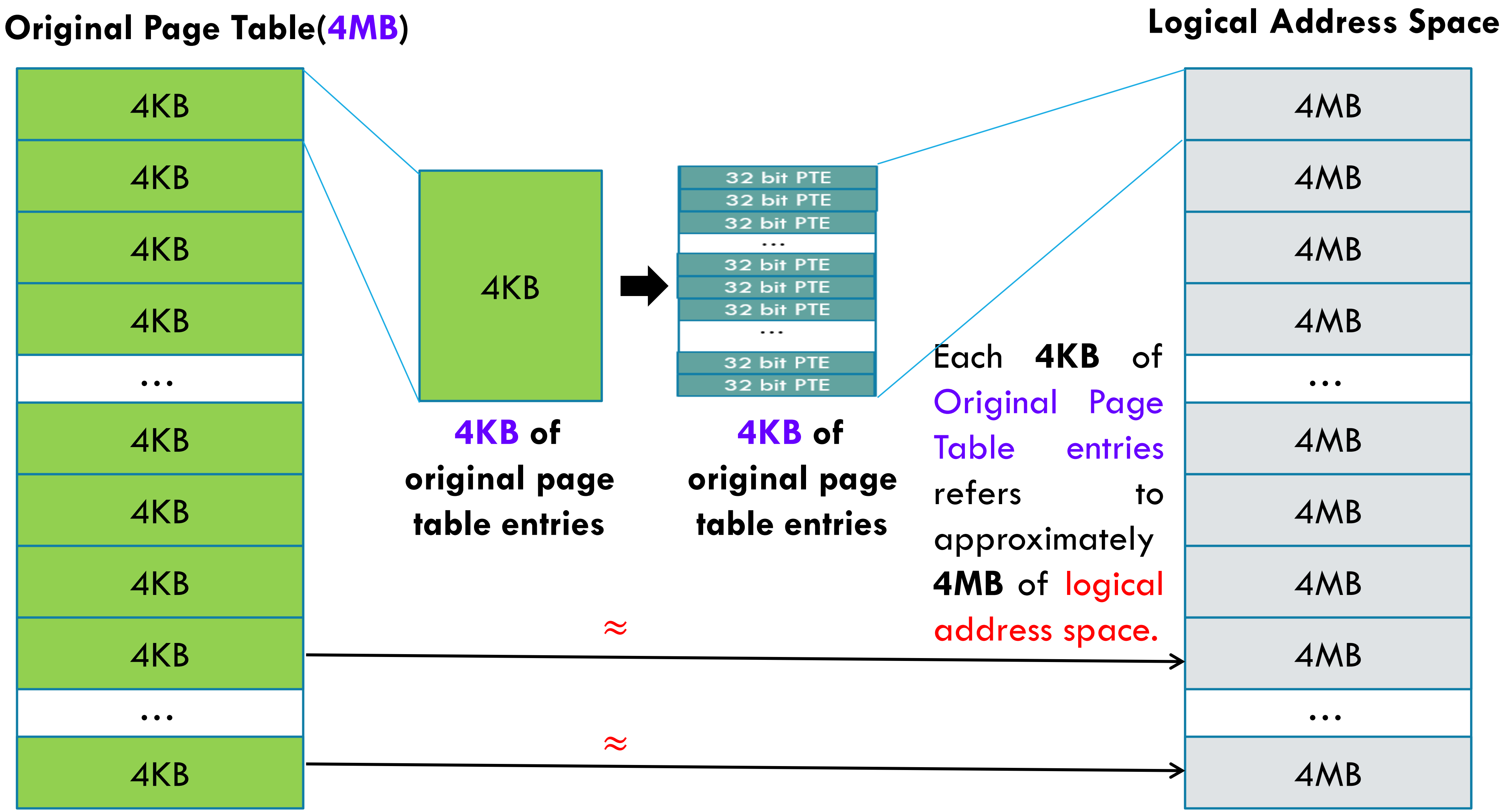


Suppose that this is a the logical address space of a process currently running.

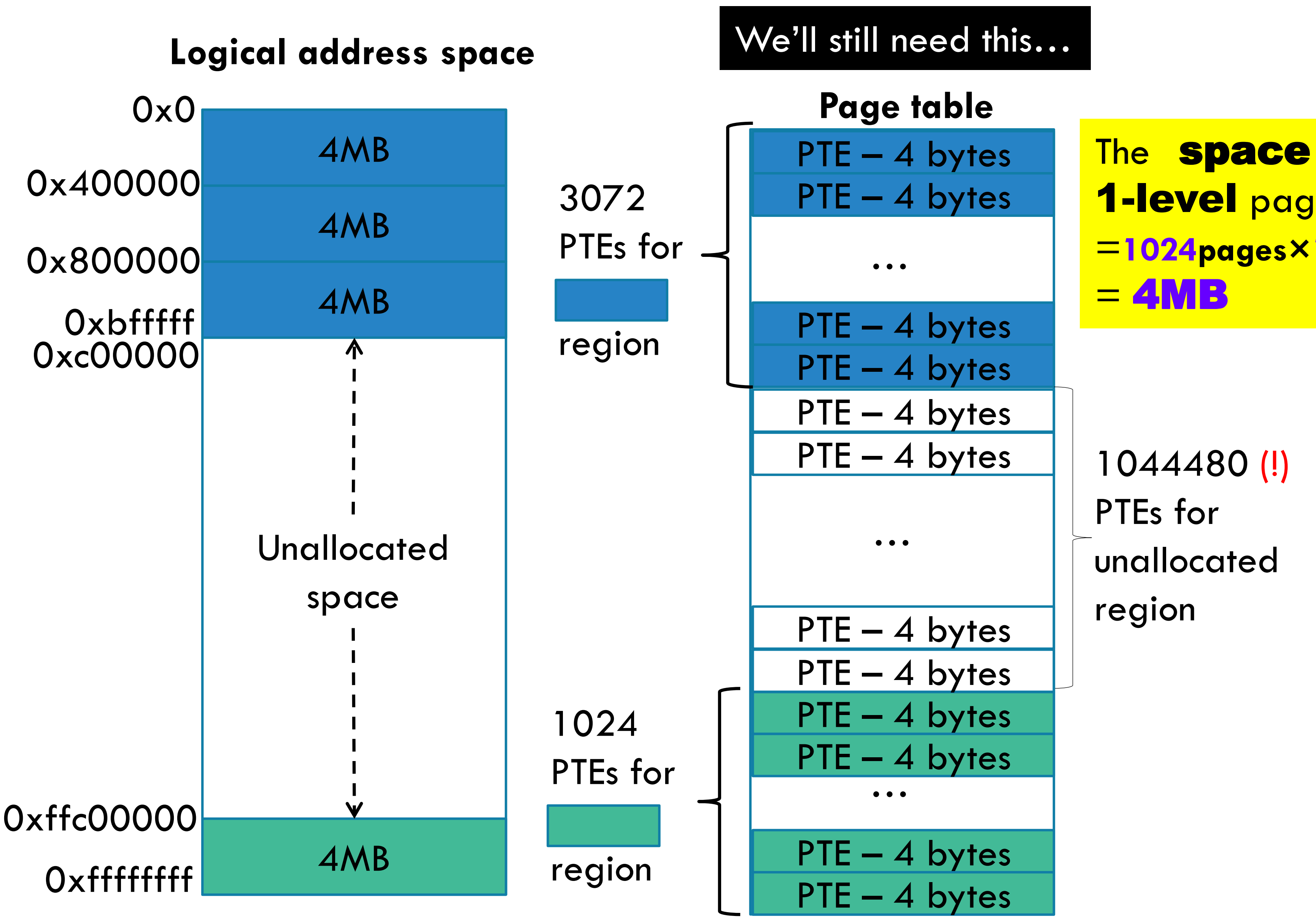
The first **12MB** are **allocated**. The last **4MB** is allocated. The rest of the space is **un-allocated**.

WHAT IF WE HAVE 1-LEVEL PAGING?

- There is a **one-to-one mapping** between a PTE and a Page. **The size of the page table is proportional to the number of virtual pages.**



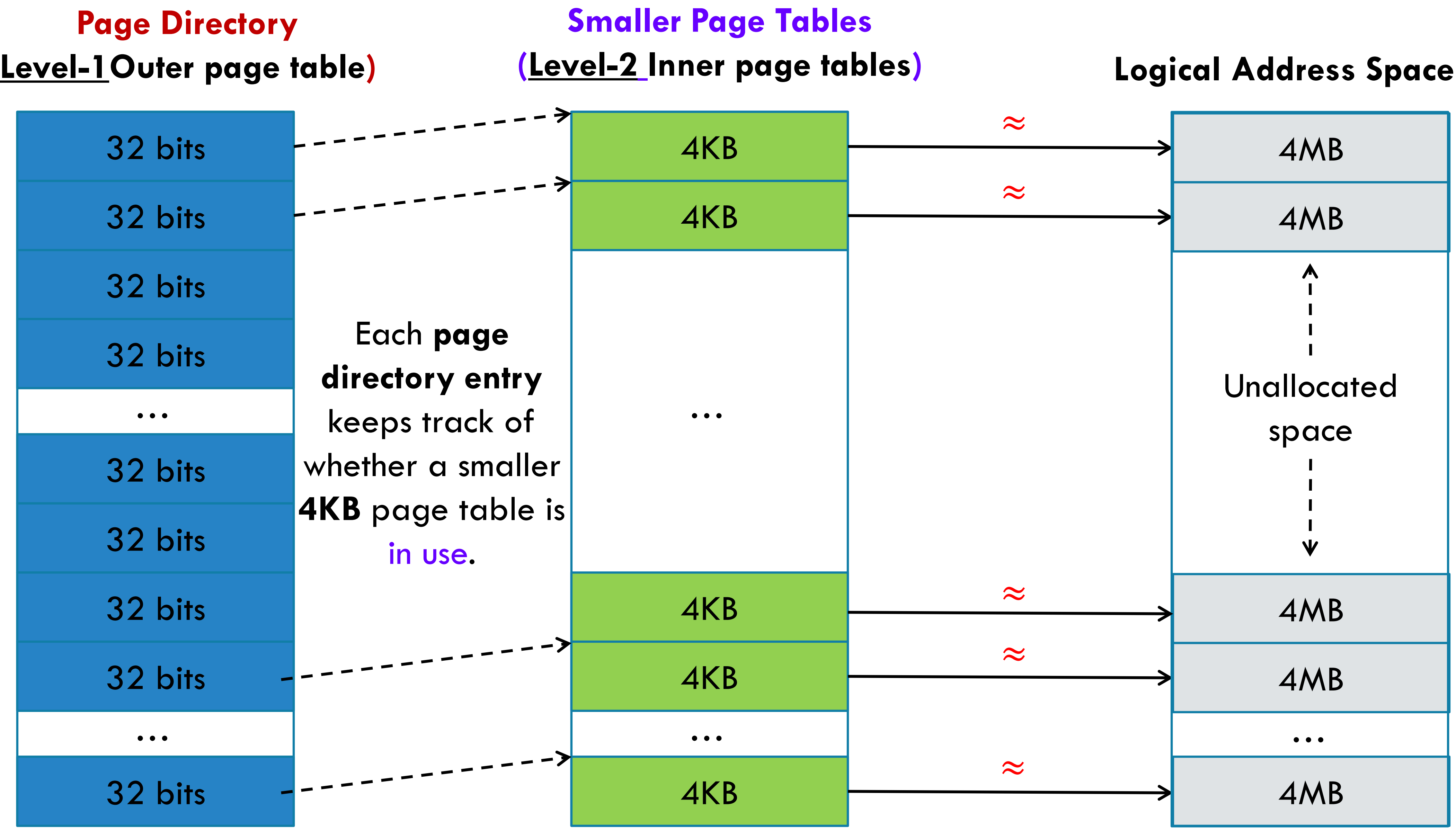
WHAT IF WE HAVE 1-LEVEL PAGING?



The **space** needed for a **1-level** page table?
$= 1024 \text{ pages} \times 1024 \text{ PTE} \times 4 \text{ bytes}$
= **4MB**

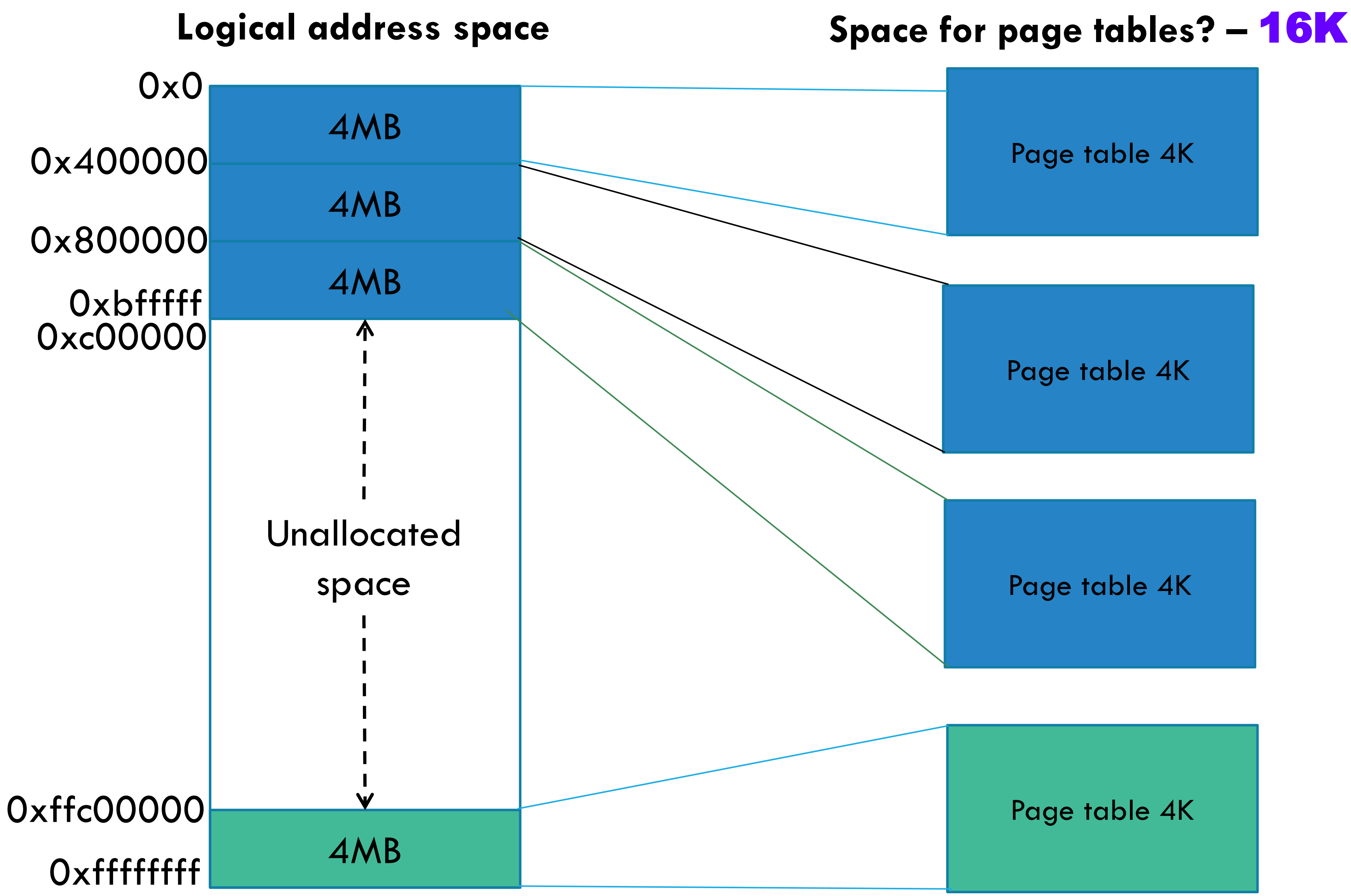
WHAT IF WE HAVE 2-LEVEL PAGING?

- **The inner page table is only created for the portions of the address space that are actually in use.**



WHAT IF WE HAVE 2-LEVEL PAGING?

The inner page table is only created for the portions of the address space that are actually in use.



Adding Hexadecimal Numbers

Adding Hexadecimal Numbers

0x1234

+

0x5678

0x 68AC

0x2222

+

0xf f f f

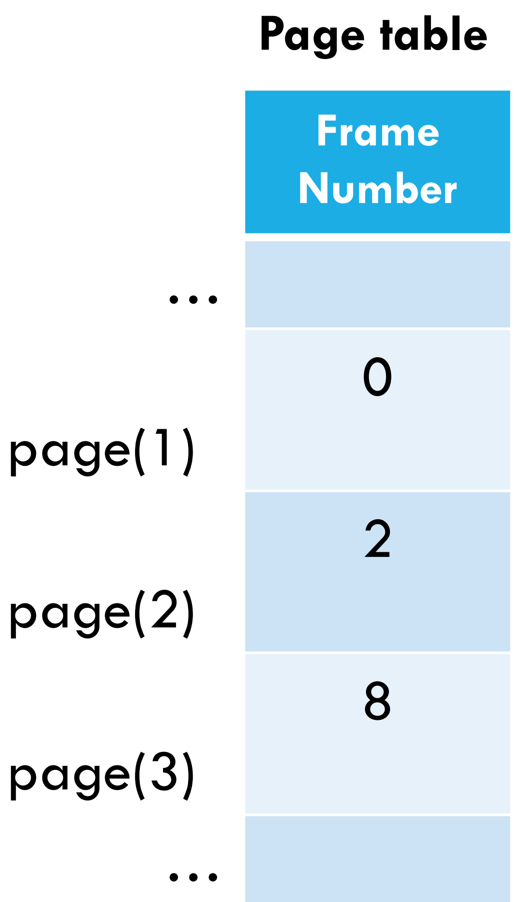
0x12221

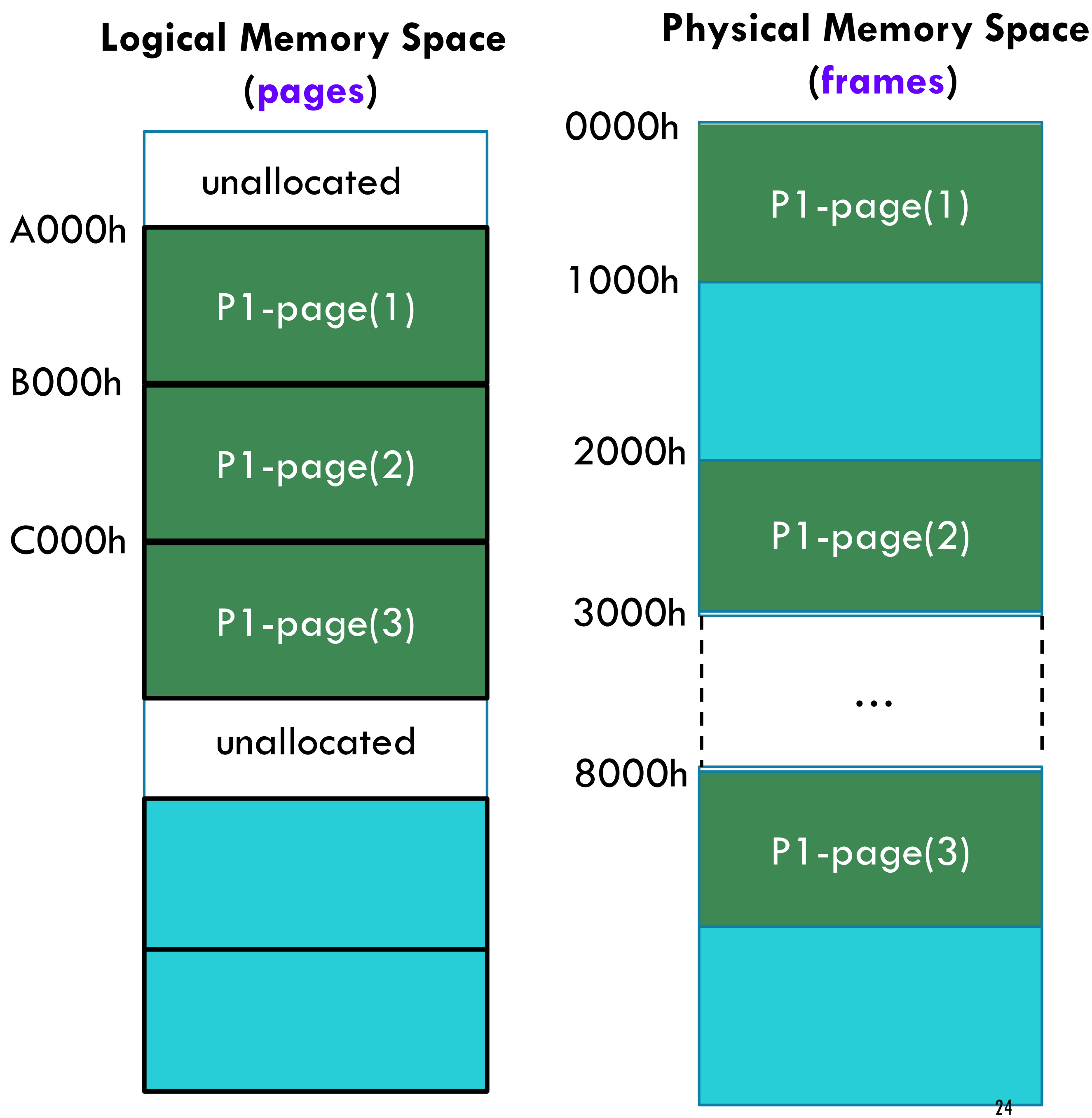
Decimal	Hexadecimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11
18	12
19	13
20	14

- **Adding two hexadecimal addresses is not a meaningful operation** in the context of memory addresses !!!.
- Adding memory addresses might not result in a valid memory location.

Generating Page Table Entry Address

PAGE TABLE

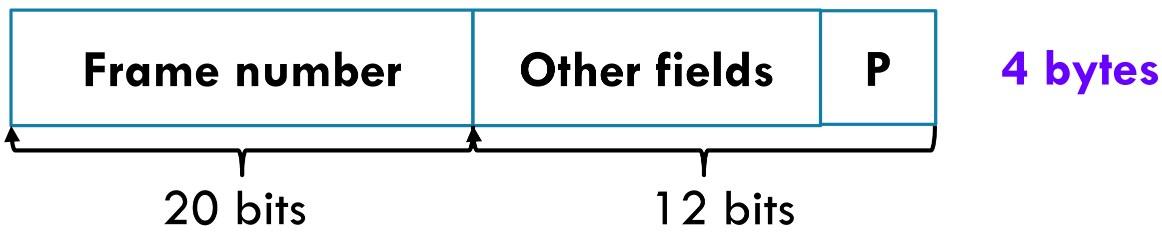


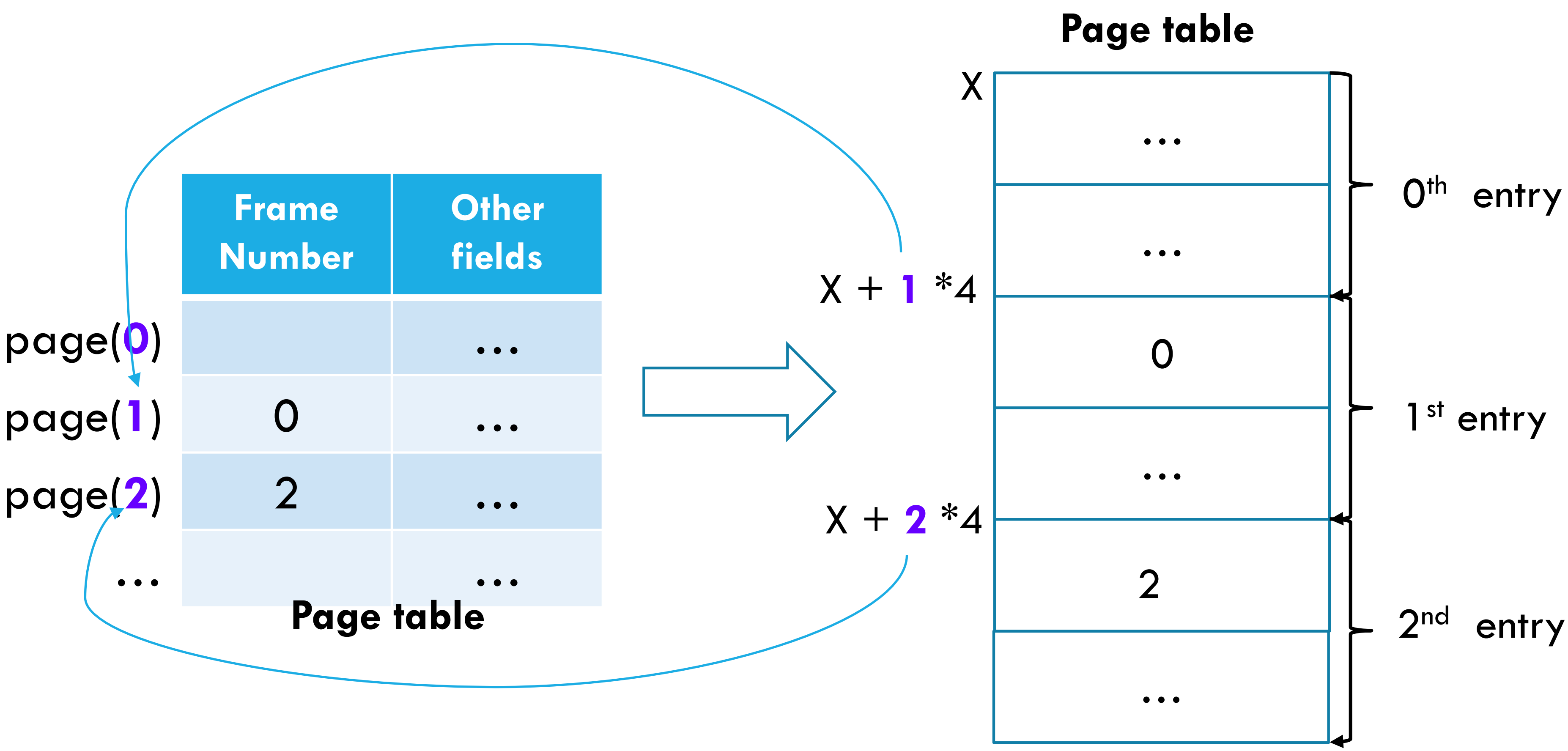


- **Per-process page table**
- **Page-to-Frame mapping**
- **Usually Page Size = 4K**

PAGE TABLE ENTRY

A realistic page table entry contains more than just a frame number





PAGE TABLE REGISTER

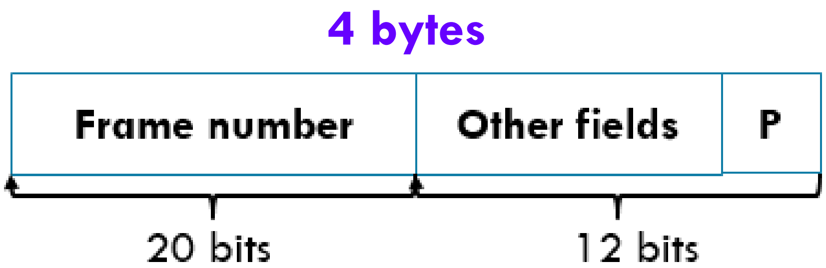
page table register

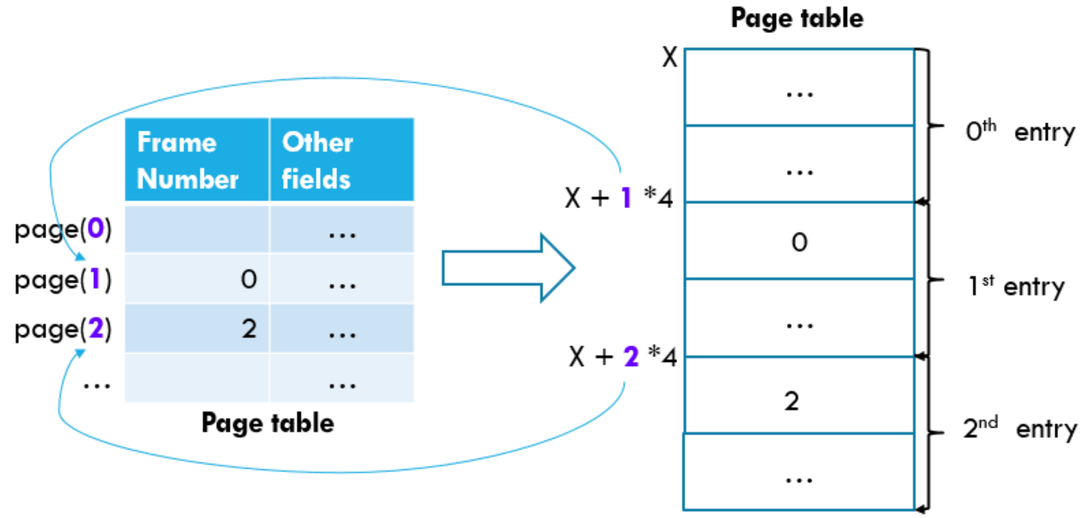
Points to the **first address** of the page table.

Page table

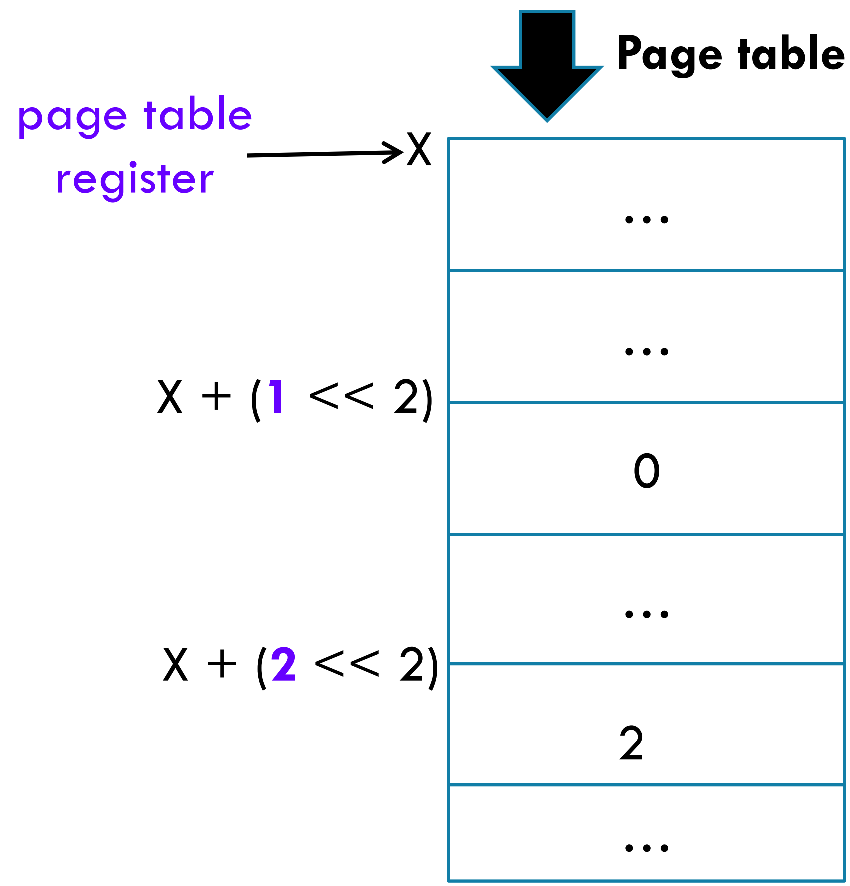
	Frame Number	Other fields	P
…	…	…	
page(1)	0	…	1
page(2)	2	…	1
…	….	…	

USING PAGE TABLE REGISTER + PAGE NO.

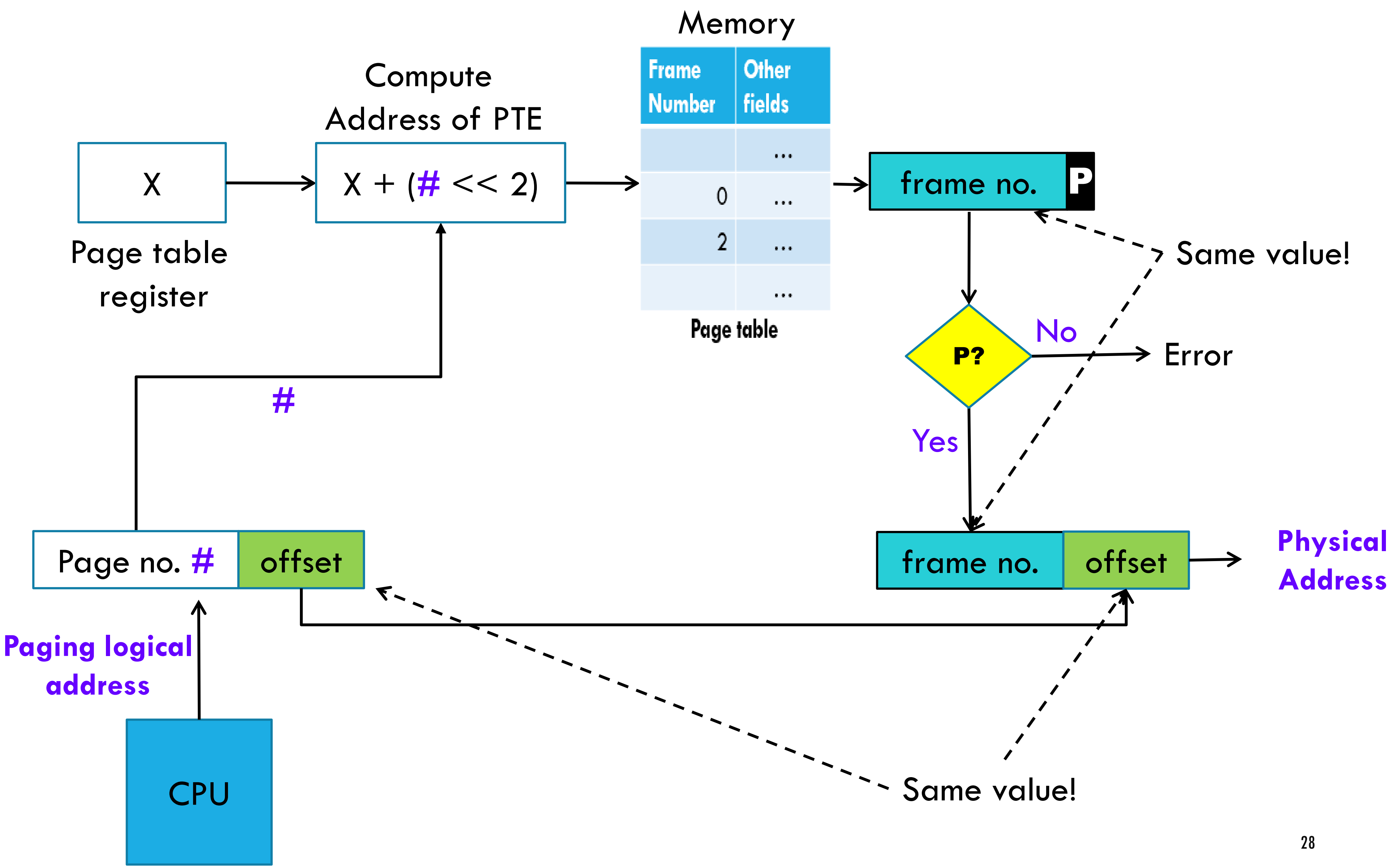




Page table register	X
Page No.	#
Address of Page table entry matching #	$X + (\# << 2)$



1-LEVEL PAGING LOGICAL ADDRESS TRANSLATION



Generating Page Table Entry Address

- **4KB** pages and **32 bit** (4bytes) logical and physical addresses.
- Given, that the logical address is: ***0x00003408***
  - ❑ Page size = 4KB = 2^{12} bytes, which requires 12 bits for the offset.
  - ❑ The remaining bits (32 - 12 = 20 bits) will be used for the page number.
- Given, that the PTR value is: ***0x64000***
- Given, that each page table entry size is **32 bits** (4bytes)

Logical address 0x00003408 to binary:

0000 0000 0000 0000 0011 0100 0000 1000

page number (the leftmost 20 bits):

0000 0000 0000 0000 0011

PTR value 0x64000 to binary:

0110 0100 0000 0000 0000

Left shift page number by 2 bits since each Page Table Entry is 4bytes:

0000 0000 0000 0000 1100 → C

➢ **Add the PTR value with the modified page number (**the result of left-shift by 2 bits**) to get the physical address of the page table entry:**

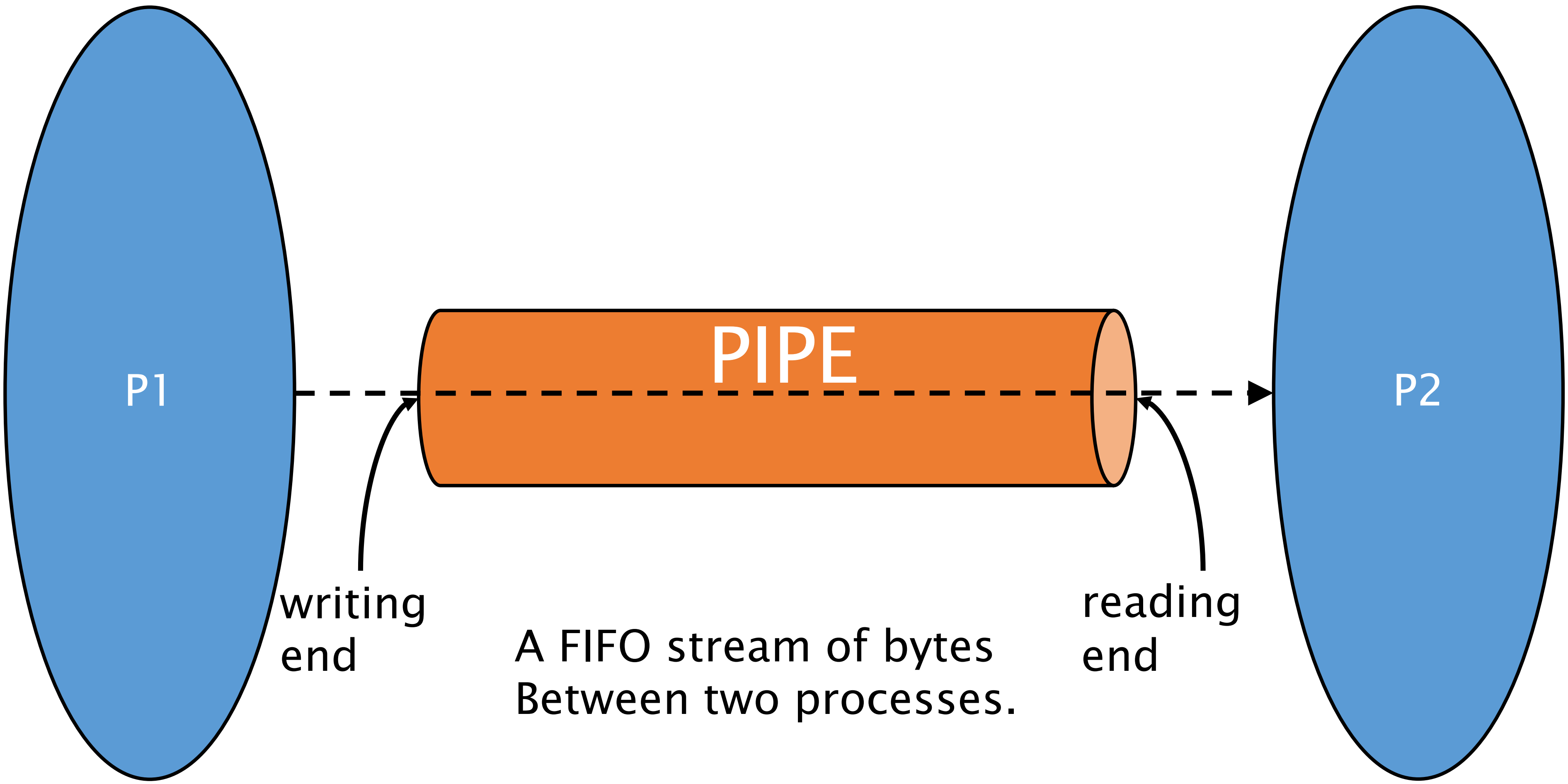
0x64000 + C → 0x6400C

Message Queue

3 IPC mechanisms

- Pipes
- Shared memory
- Message Passing

Pipes - the basic idea

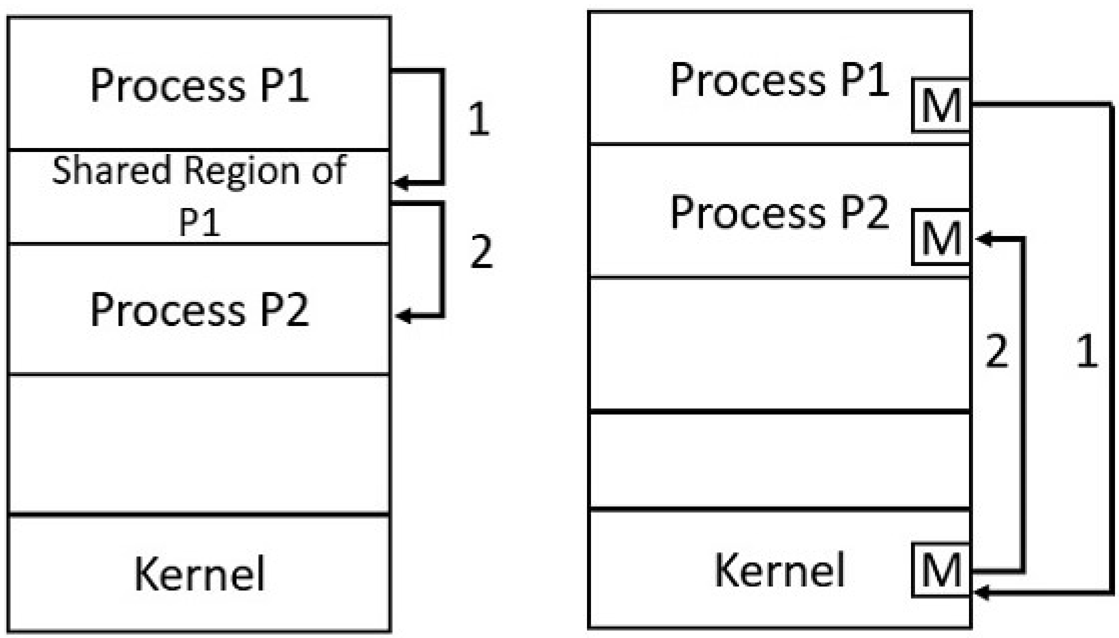


- **Full-Duplex:** Bidirectional communication with simultaneous transmission and reception.
- **Half-Duplex:** Bidirectional communication with non-simultaneous transmission and reception.
- **Unidirectional:** One-way communication with data flowing in only one direction.

Shared Memory - the basic idea

- It's a memory that may be **simultaneously accessed** by multiple programs with an intent to provide communication among them or avoid redundant copies.

- In a shared memory system, the processes share data using a shared region established in their own address space (no kernel intervention). It is the **fastest IPC mechanism.**



Shared Memory System　　Message Passing System

POSIX Message queue example

```
typedef struct
 {
          long int type;
          char buffer[1024];
} msg_struct;

#define MSG_STRUCT 1
```

- Allows for passing messages between processes.
- Messages can have different structures.
- A message structure must have a long int as its **first field** (used for the message type identifier).
- Message size (specified in msgsnd() and msgrcv()) is the message structure size ***without*** the long int identifier

POSIX message queues

- **int msgget(key_t key, int msgflg);**
  - creates (or retrieves an existing) message queue.
  - returns a non-negative integer, namely a message queue identifier upon successful completion

```
int pid, queue_id;
msg_struct msg;
queue_id = msgget((key_t)100,0666|IPC_CREAT);
```

POSIX message queues

- **int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);**
  - sends a message to the message queue.
  - returns zero upon successful completion

```
msg.type = MSG_STRUCT;
strcpy(msg.buffer,"Was it a rat I saw?");
msgsnd(queue_id,&msg,1024,0);
strcpy(msg.buffer,"No lemons, no melon.");
msgsnd(queue_id,&msg,1024,0);
```

POSIX message queues

- **ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);**
  - retrieves (and removes) a message from the message queue.
  - if **msgtyp** is **0**, the first message of any type on the queue shall be received
  - has two modes of receiving:
    - IPC_NOWAIT: returns -1 if there are no messages on the queue (nonblocking)
    - not IPC_NOWAIT: blocks until there is a message on the queue
  - return a value equal to the number of bytes placed into the buffer upon successful completion

```
while (msgrcv(queue_id,&msg,1024,0,IPC_NOWAIT) != -1)
{
      if (msg.type == MSG_STRUCT)
         fprintf(stdout,"message: %s\n",msg.buffer);
      else
         fprintf(stdout,"unknown message\n");
}
```

POSIX message queues

- **int msgctl(int msqid, int cmd, struct msqid_ds *buf);**
  - deletes the message queue
  - returns zero upon successful completion

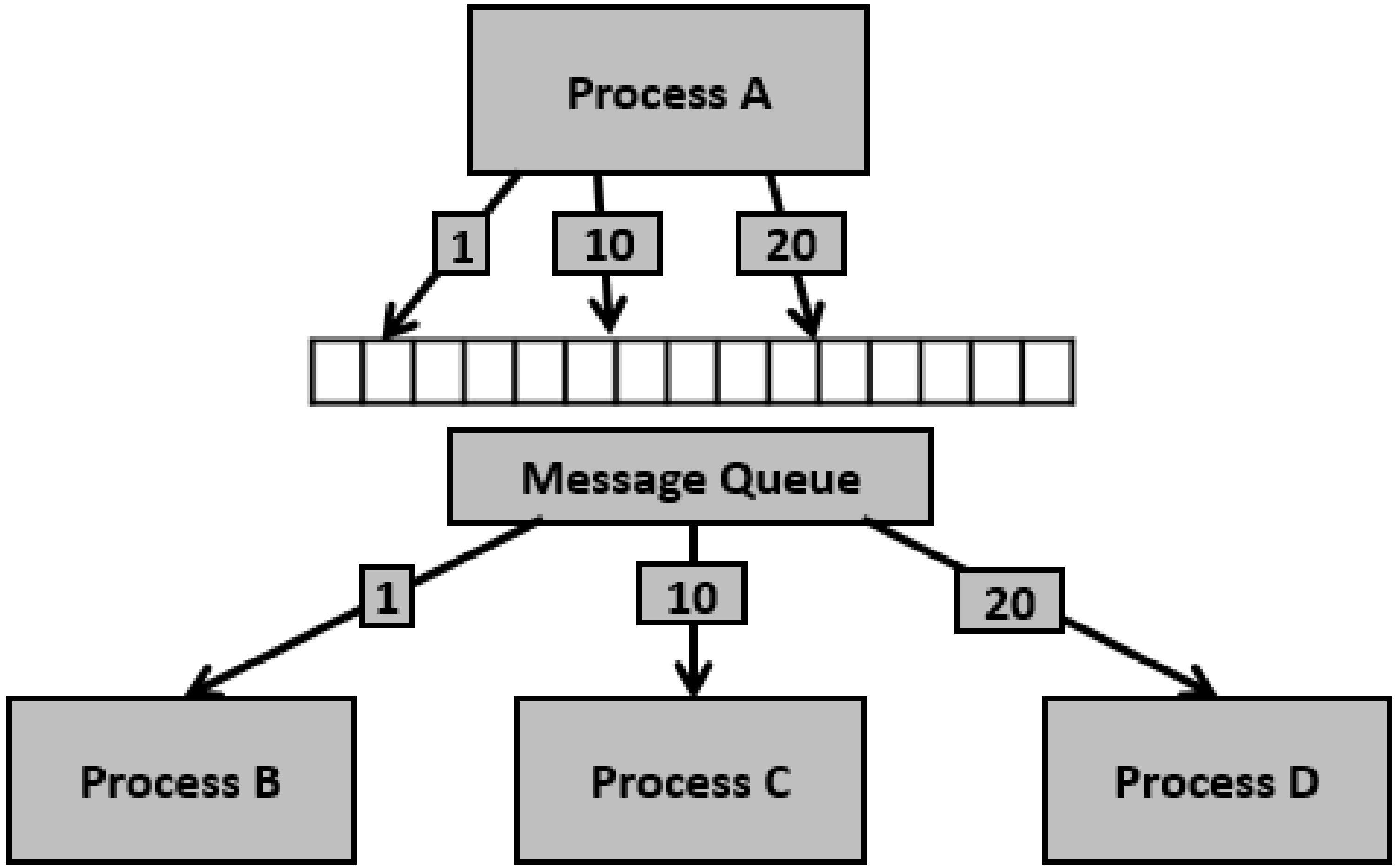
```
msgctl(queue_id,IPC_RMID,NULL);
```

POSIX Message queue example

```
int main(void) {
        int pid, queue_id;
        msg_struct msg;
        queue_id = msgget((key_t)100,0666|IPC_CREAT);
        pid = fork();

        if (pid == 0) {
                msg.type = MSG_STRUCT;
                strcpy(msg.buffer,"Was it a rat I saw?");
                msgsnd(queue_id,&msg,1024,0);
                strcpy(msg.buffer,"No lemons, no melon.");
                msgsnd(queue_id,&msg,1024,0);
                exit(0); }
        else {
                wait(NULL);
                while (msgrcv(queue_id,&msg,1024,0,IPC_NOWAIT) != -1) {
                        if (msg.type == MSG_STRUCT)
                                fprintf(stdout,"message: %s\n",msg.buffer);
                        else
                                fprintf(stdout,"unknown message\n"); }

        msgctl(queue_id,IPC_RMID,NULL); }
        return 0; }
```

- Header file **sys/msg.h** must be included
- Each process must know the **queue_id** of the queue
- Queue has **file-style** permissions

POSIX Message queue example

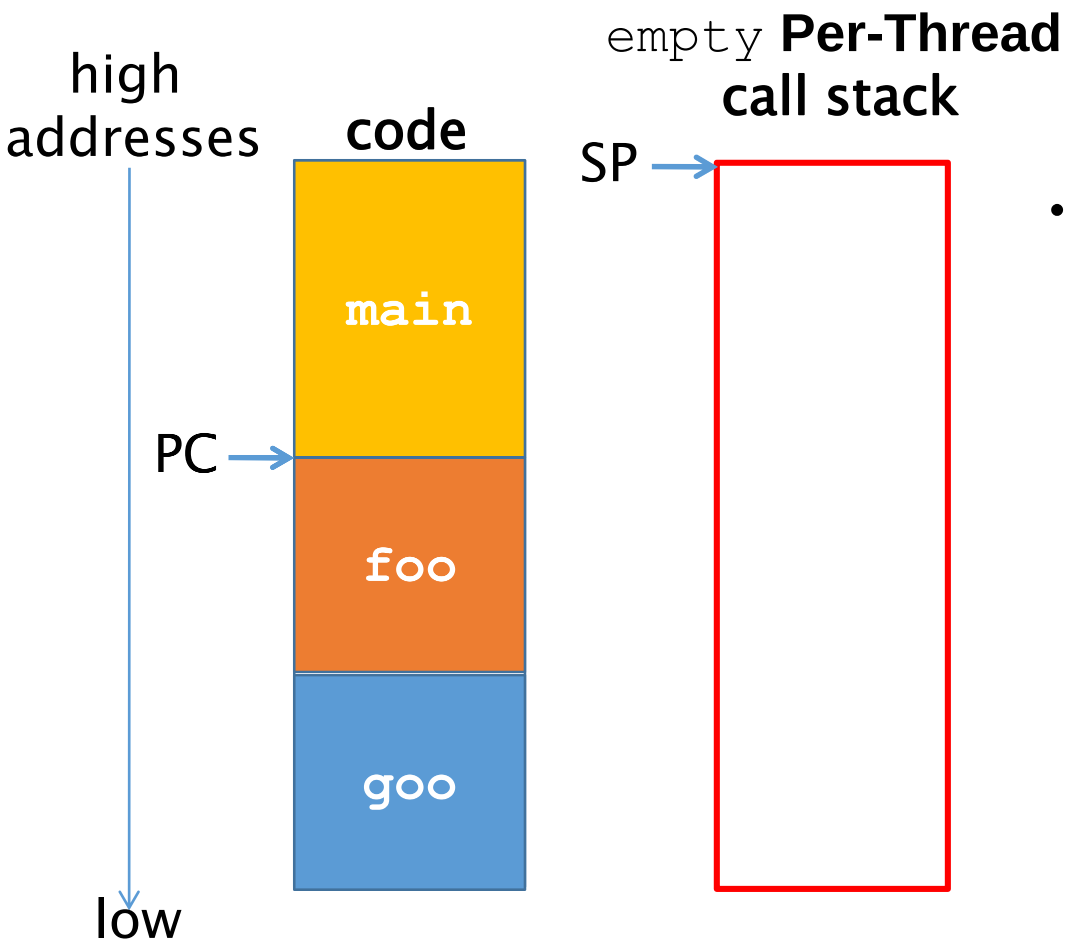


- POSIX message queues are typically **unidirectional,** supporting communication in one direction between processes.

- To enable **bidirectional communication**, you would generally need to set up two message queues—one for sending messages from process A to process B and another for sending messages from process B to process A.

Multi-threading Model

Stack Pointer and Program Counter



- Consider a code with the following functions:

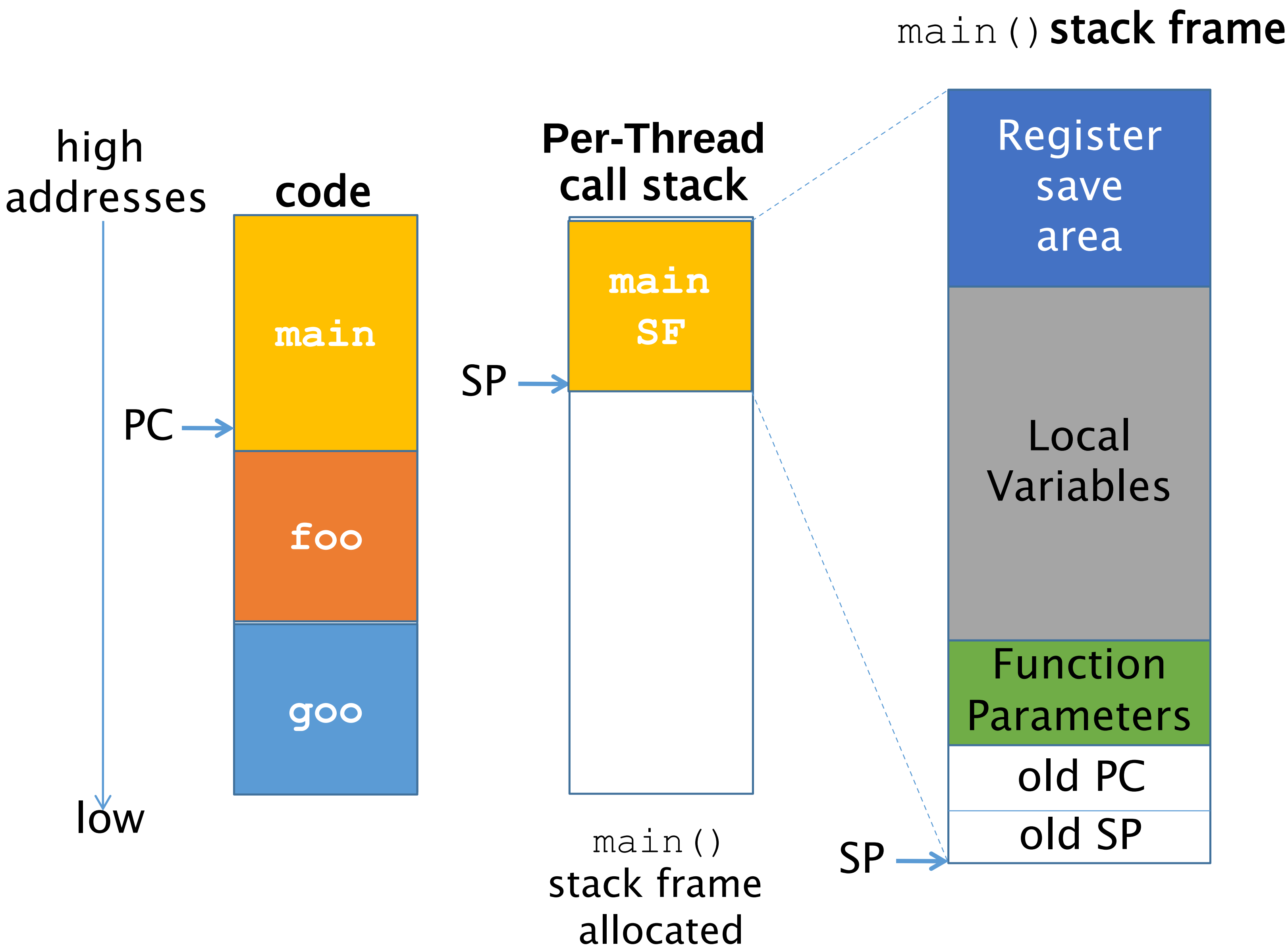
```
int main(int argc, char**argv)
{
    char *foo(int, int, int);
    int goo(double, int);
}
```

- The functions are called:
  `main->foo->goo`

- PC currently pointing to `main()`.

- The Stack is empty. No stack frame.

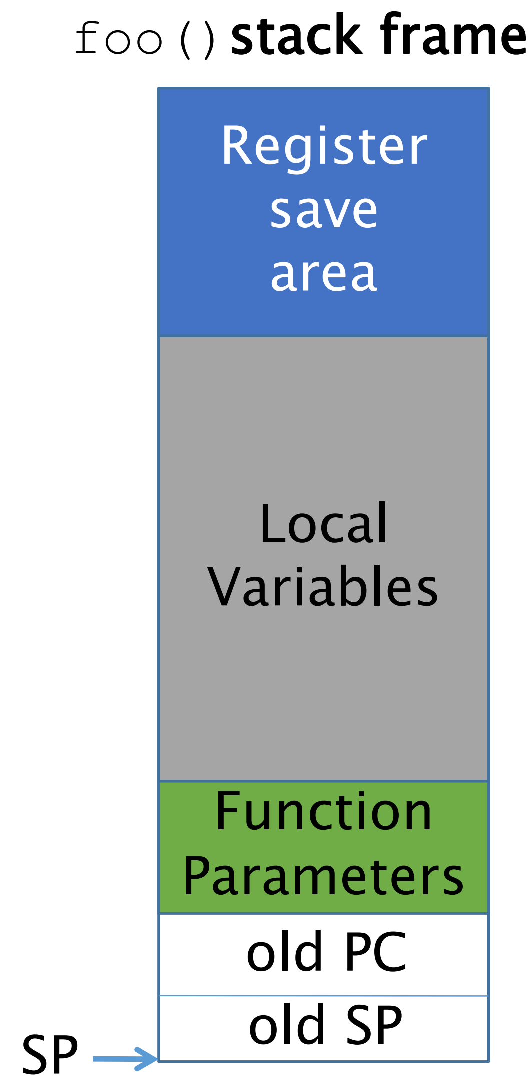
Stack Frame



main() stack frame
high addresses
low
code
main
foo
goo
PC
Per-Thread call stack
main SF
SP
main() stack frame allocated
Register save area
Local Variables
Function Parameters
old PC
old SP
SP

Stack Frame Organization - I

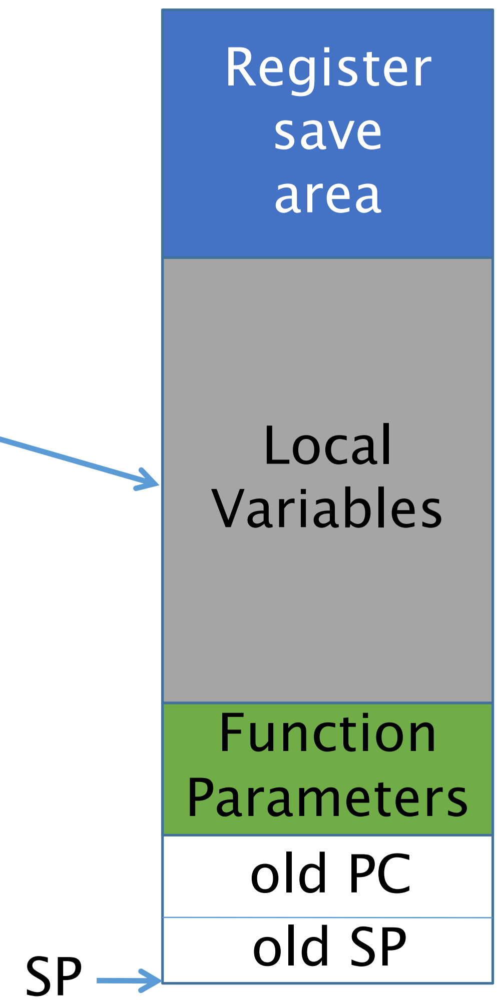
```
char *foo(int x, int y , int z)
{
     int a;
     char array[500];
     double d;
     …

     a = x+y+goo(d,z);
     …
}
```



Stack Frame Organization - II

```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```

`foo()` **stack frame**

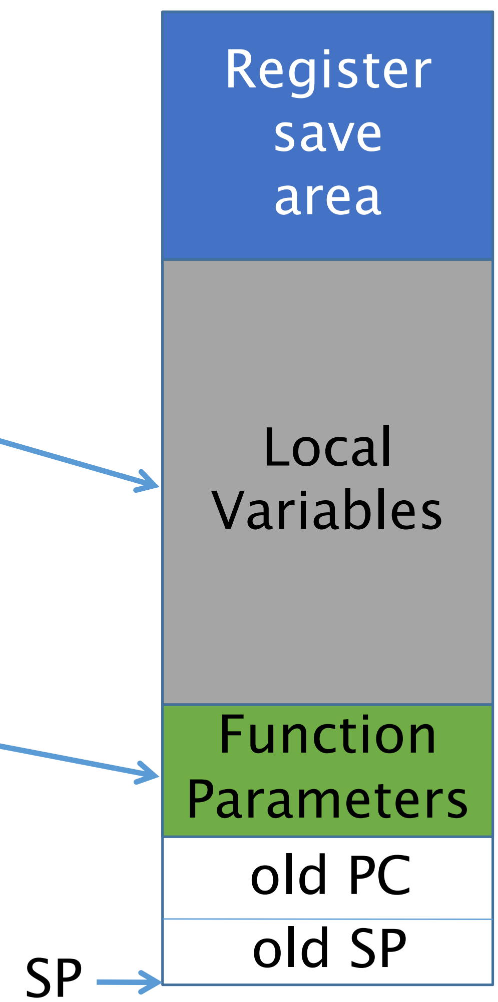


a and d could be in registers

Stack Frame Organization - III

```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```

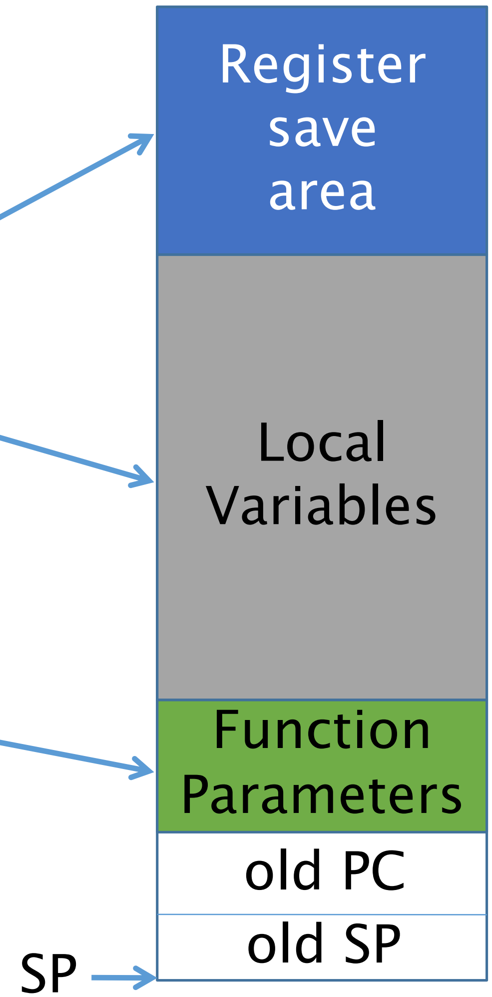
`foo()` **stack frame**



Stack Frame Organization - IV

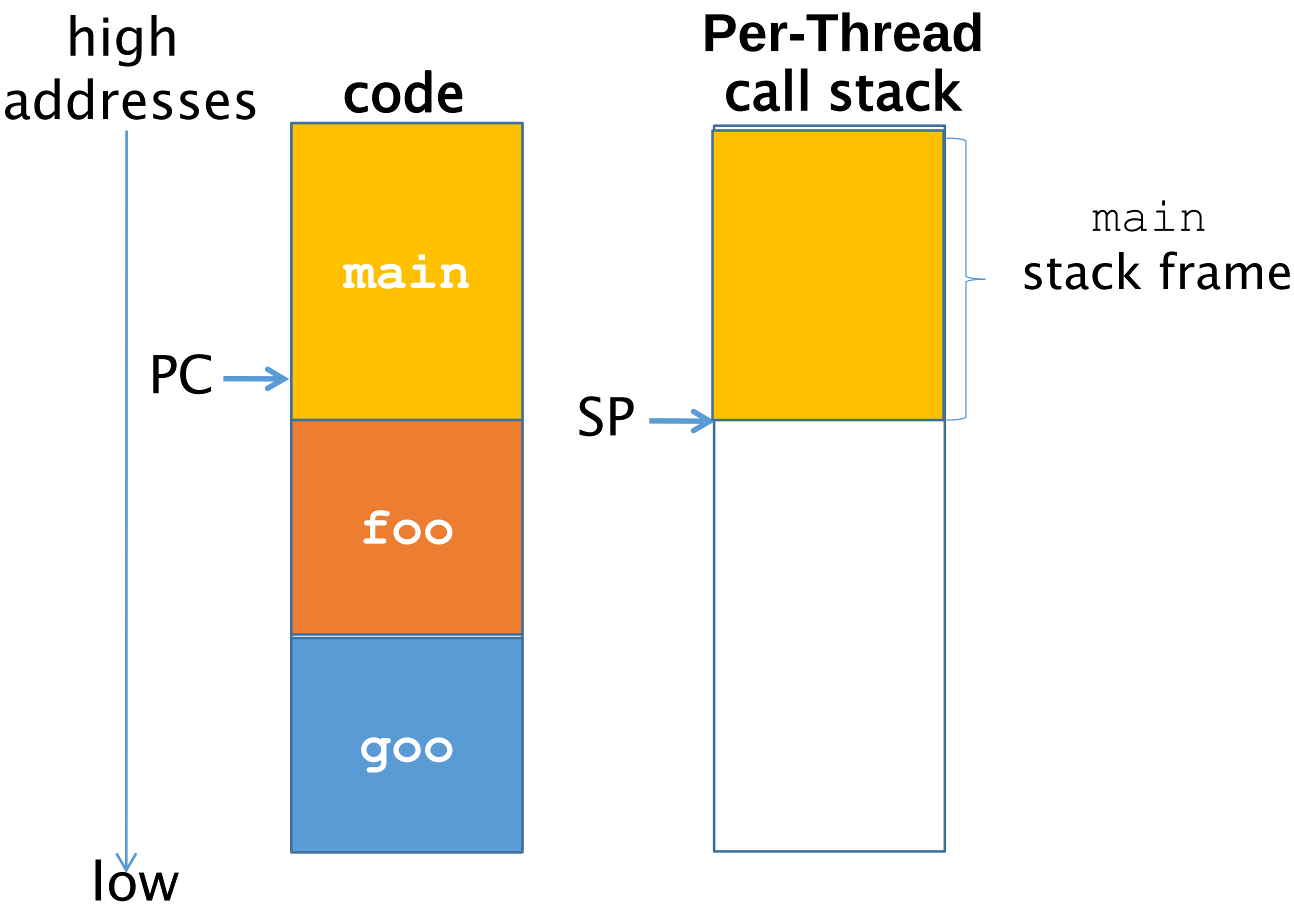
```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```

`foo()` **stack frame**



Function Call and SF Creation - I

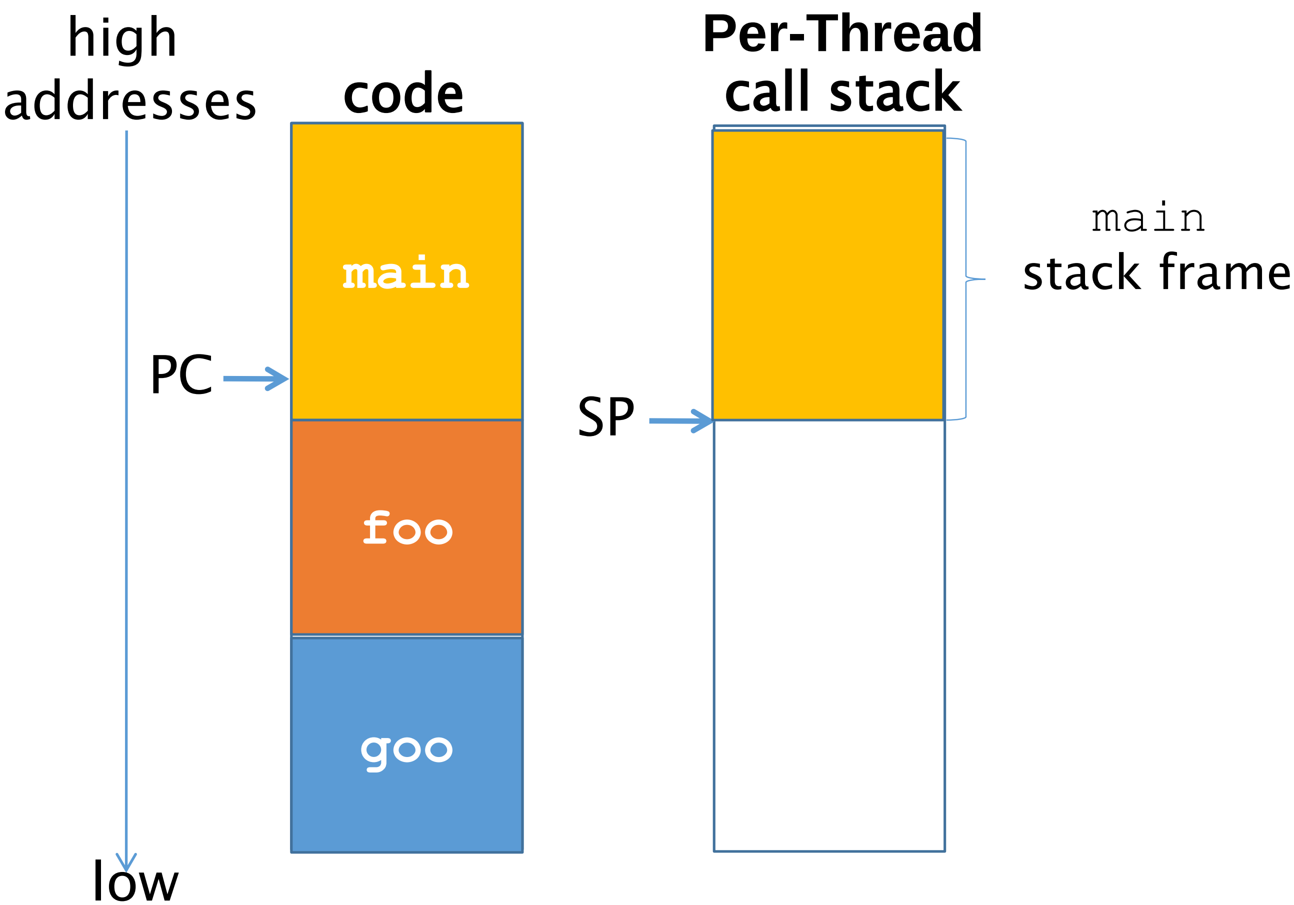
```
int main(int argc, char**argv)
  {
      char *foo(int, int, int);
      int goo(double, int);
  }
```



Function Call and SF Creation - II

```
int main(int argc, char**argv)
  {
      char *foo(int, int, int);
      int goo(double, int);
  }
```

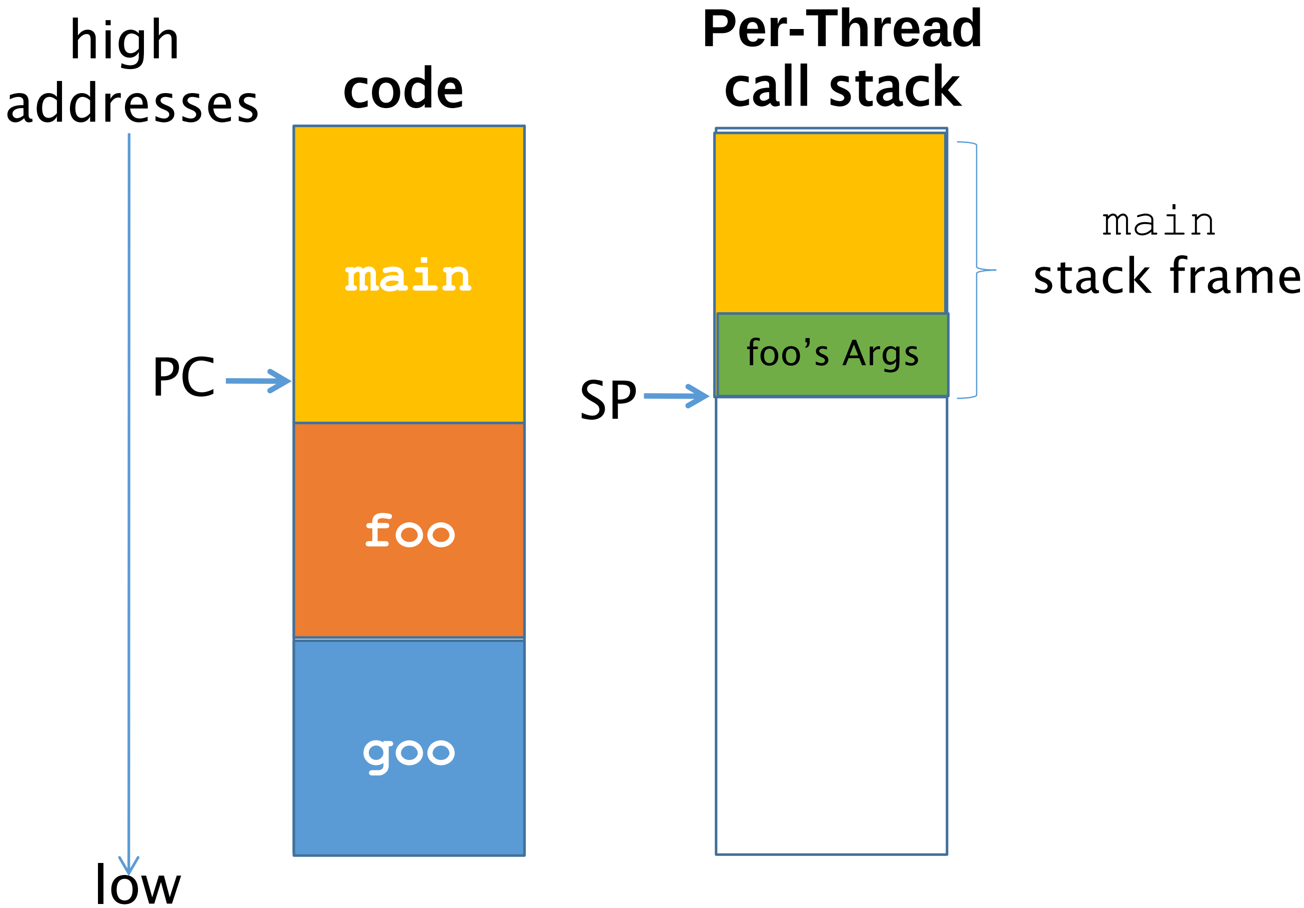
1. Call to `foo`



Function Call and SF Creation - III

```
int main(int argc, char**argv)
  {
      char *foo(int, int, int);
      int goo(double, int);
  }
```

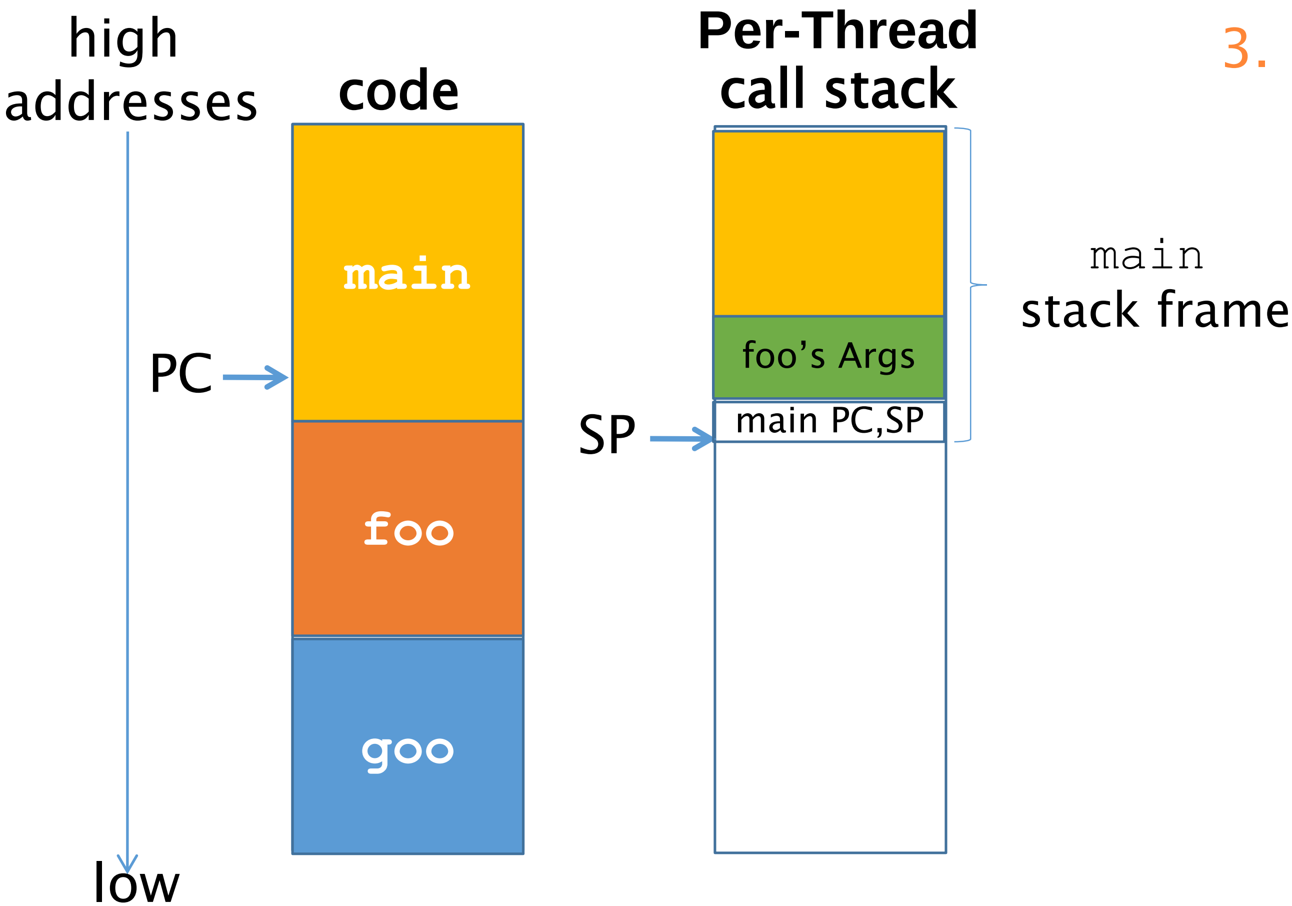
1. Call to `foo`
2. Push `foo`'s Args



Function Call and SF Creation - IV

```
int main(int argc, char**argv)
  {
      char *foo(int, int, int);
      int goo(double, int);
  }
```

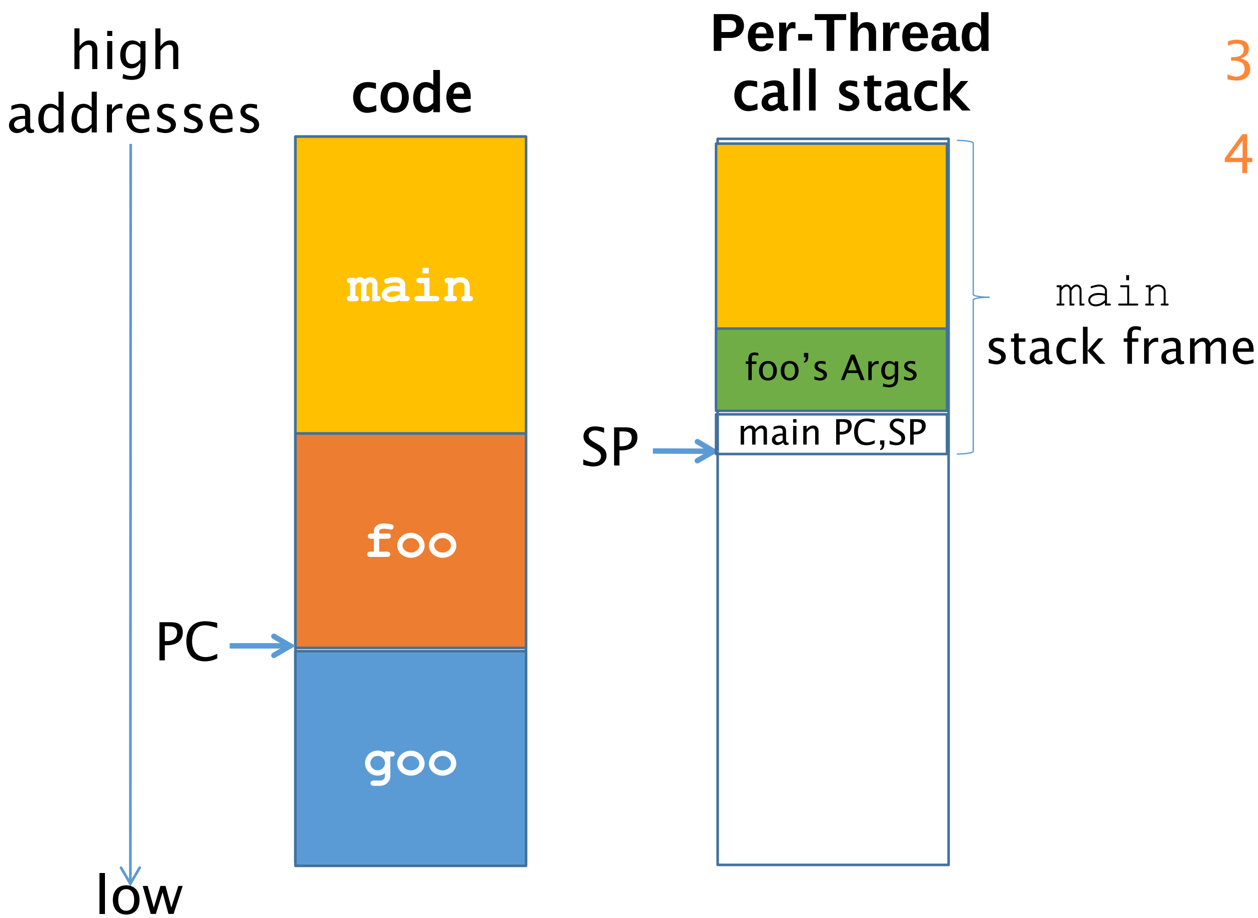
1. Call to `foo`
2. Push `foo`'s Args
3. Save `main` PC & SP



Function Call and SF Creation - V

```
int main(int argc, char**argv)
  {
      char *foo(int, int, int);
      int goo(double, int);
  }
```

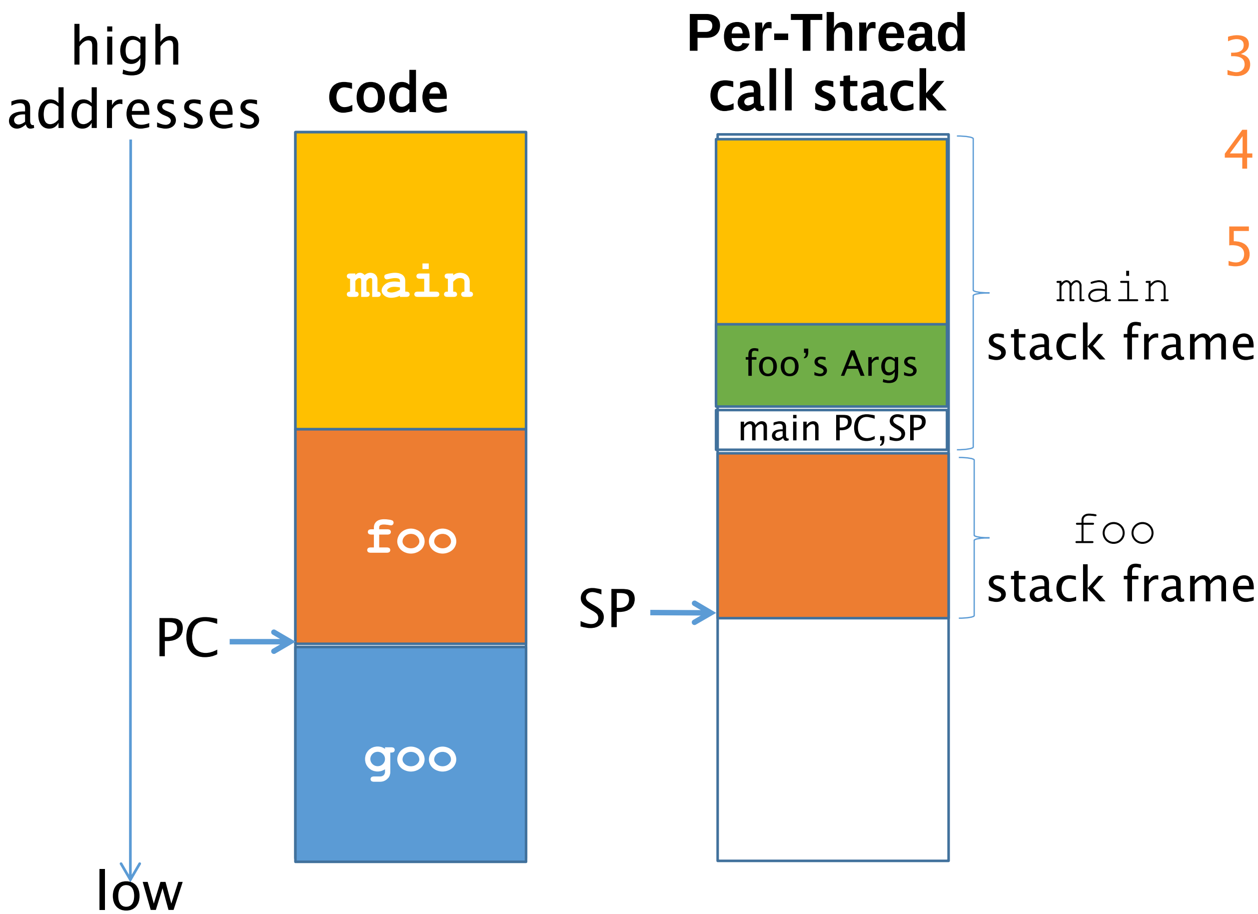
1. Call to `foo`
2. Push `foo`'s Args
3. Save `main` PC & SP
4. Call `foo`



Function Call and SF Creation - VI

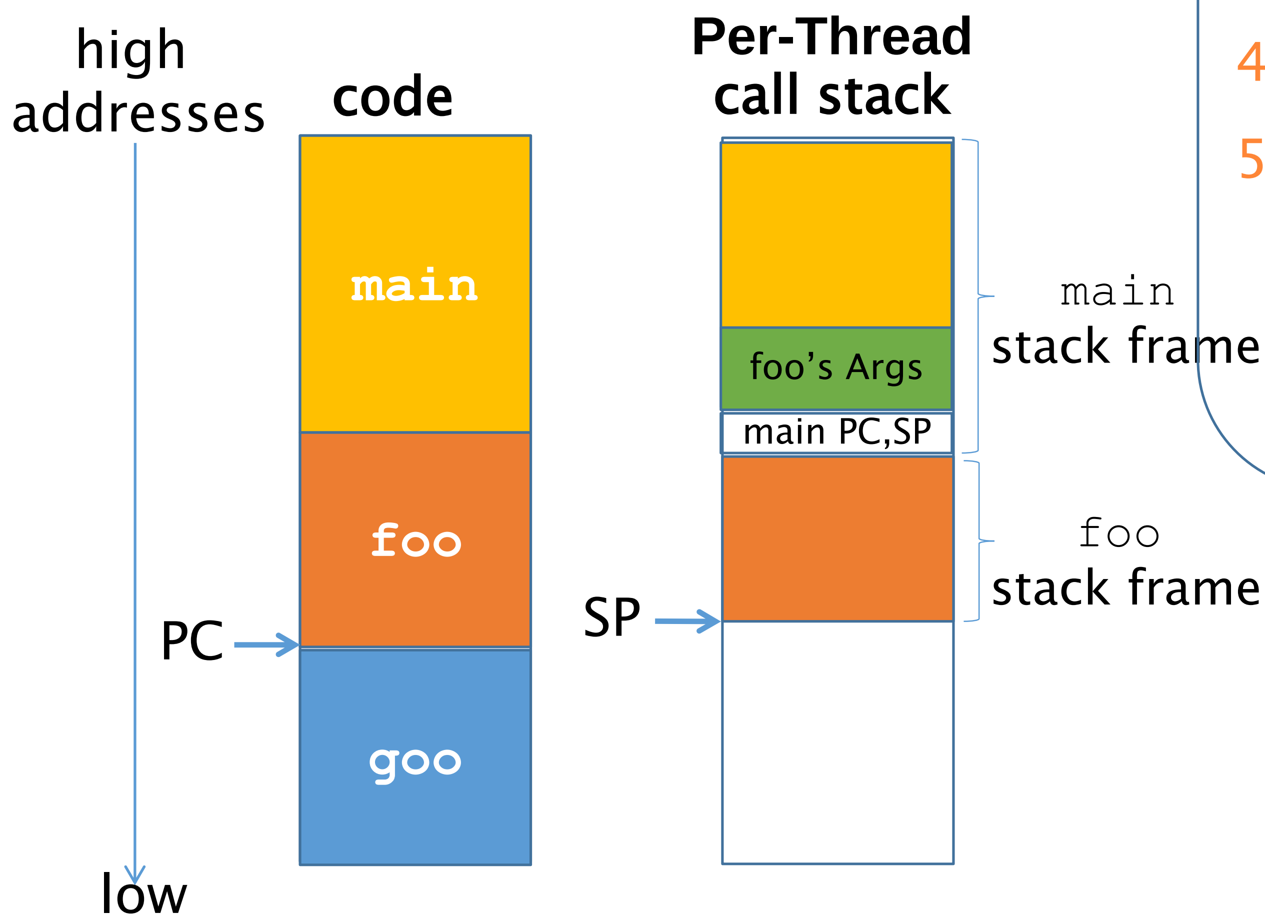
```
int main(int argc, char**argv)
  {
      char *foo(int, int, int);
      int goo(double, int);
  }
```

1. Call to `foo`
2. Push `foo`'s Args
3. Save `main` PC & SP
4. Call `foo`
5. Allocate `foo` stack frame and adjust SP to point this stack frame



Function Call and SF Creation - VII

```
int main(int argc, char**argv)
  {
       char *foo(int, int, int);
       int goo(double, int);
  }
```



1. Call to `foo`
2. Push `foo`'s Args
3. Save `main` PC & SP
4. Call `foo`
5. Allocate `foo` stack frame and adjust SP to point this stack frame

Done in software
i.e., done by instructions
generated by the compiler!

Q & A

- Does each function use the same stack frame size?
  - No. Depends on the size of local variables
- How and when is the size of stack frame determined?
  - The compiler determines by looking at the code. Compile-time.
- How is the stack frame allocated during run-time?
  - By decrementing the stack pointer

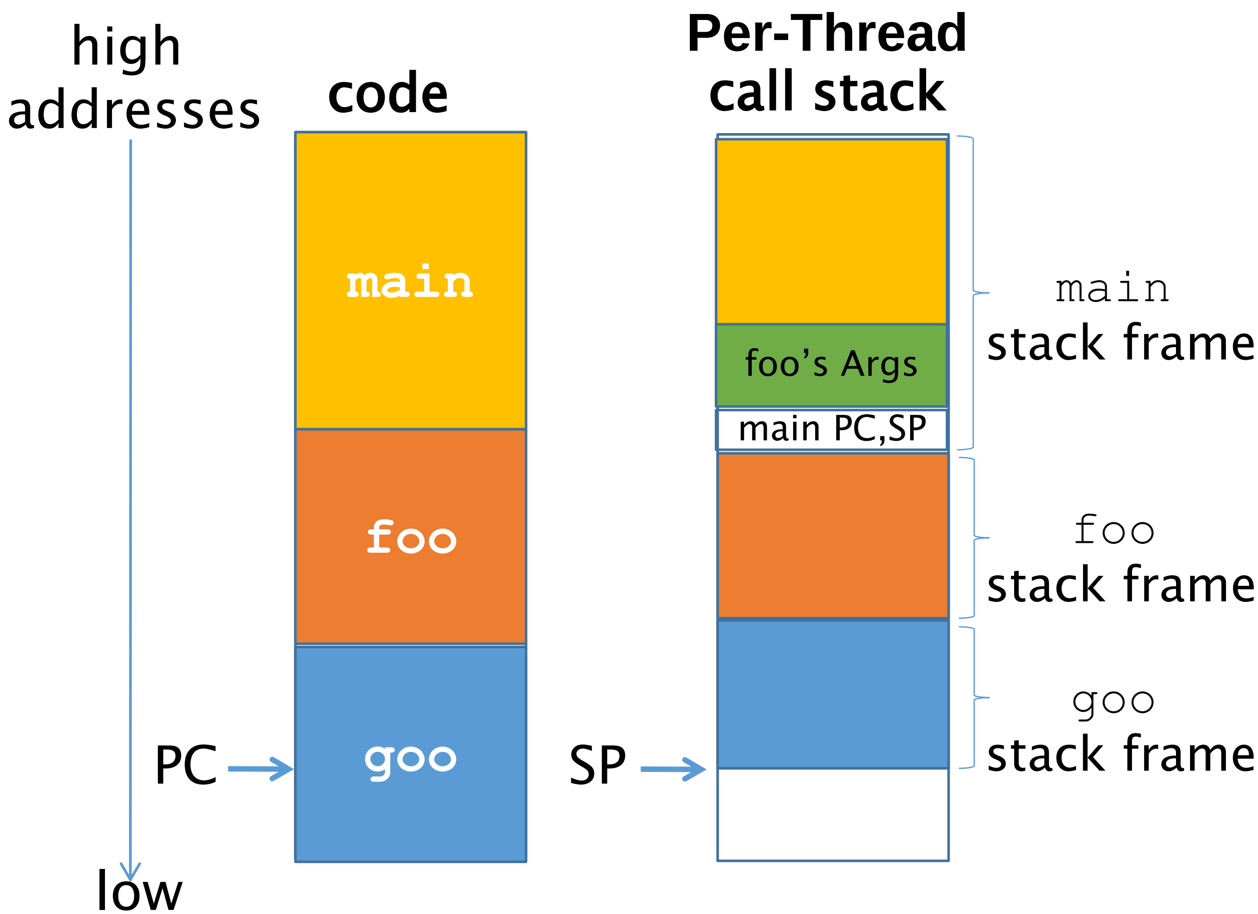
Q & A

- What is the stack pointer?
  - A value stored in stack pointer register (%esp(32bits), %rsp(64bits)) pointing to the beginning of the stack frame
- What is a program counter?
  - A value stored in program counter register (%eip(32bits), %rip(64bits)) pointing to address of the next instruction to be executed.

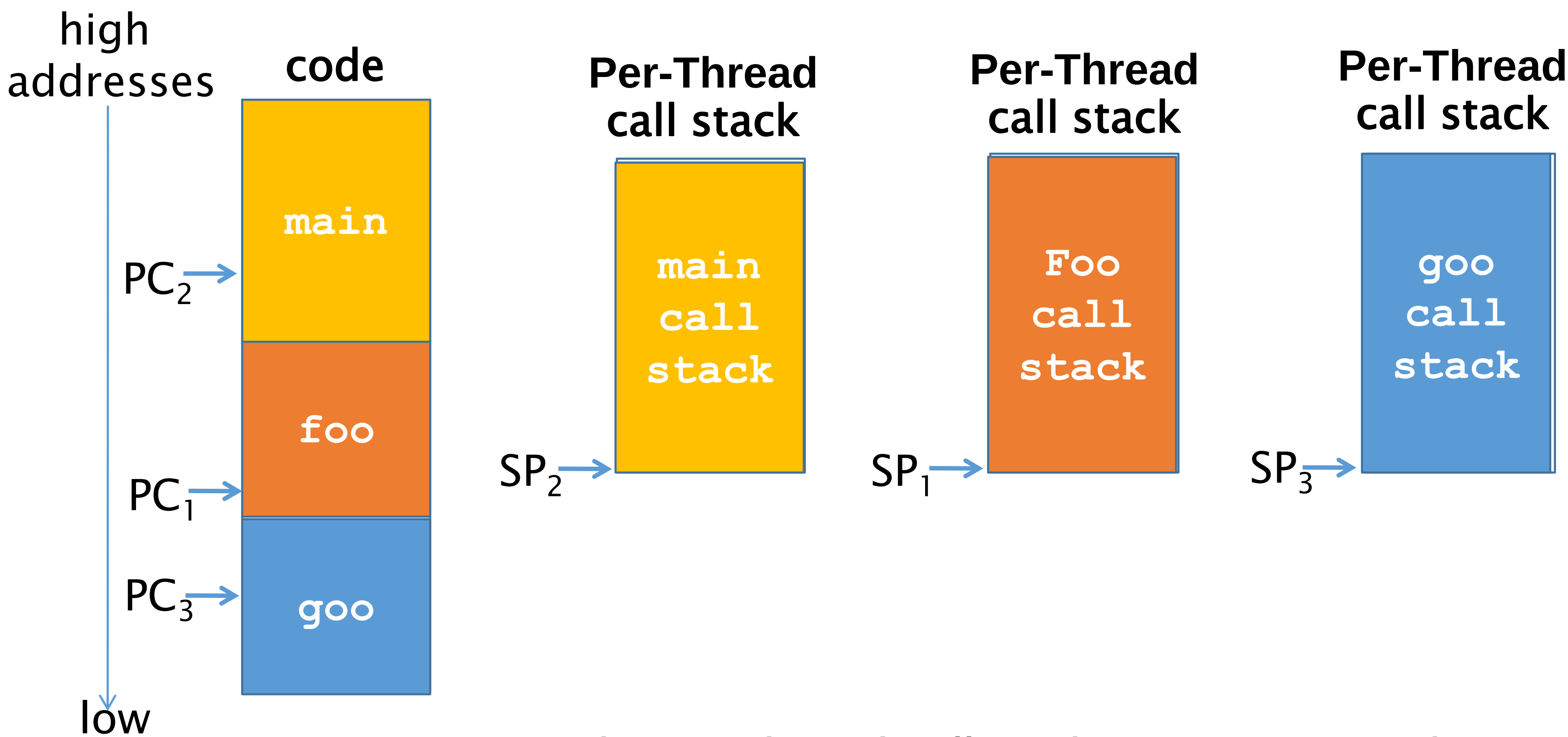
Single-thread process

```
int main(int argc, char**argv)
  {
      char *foo(int, int, int);
      int goo(double, int);
  }
```

- Thread of control
  - PC
  - SP

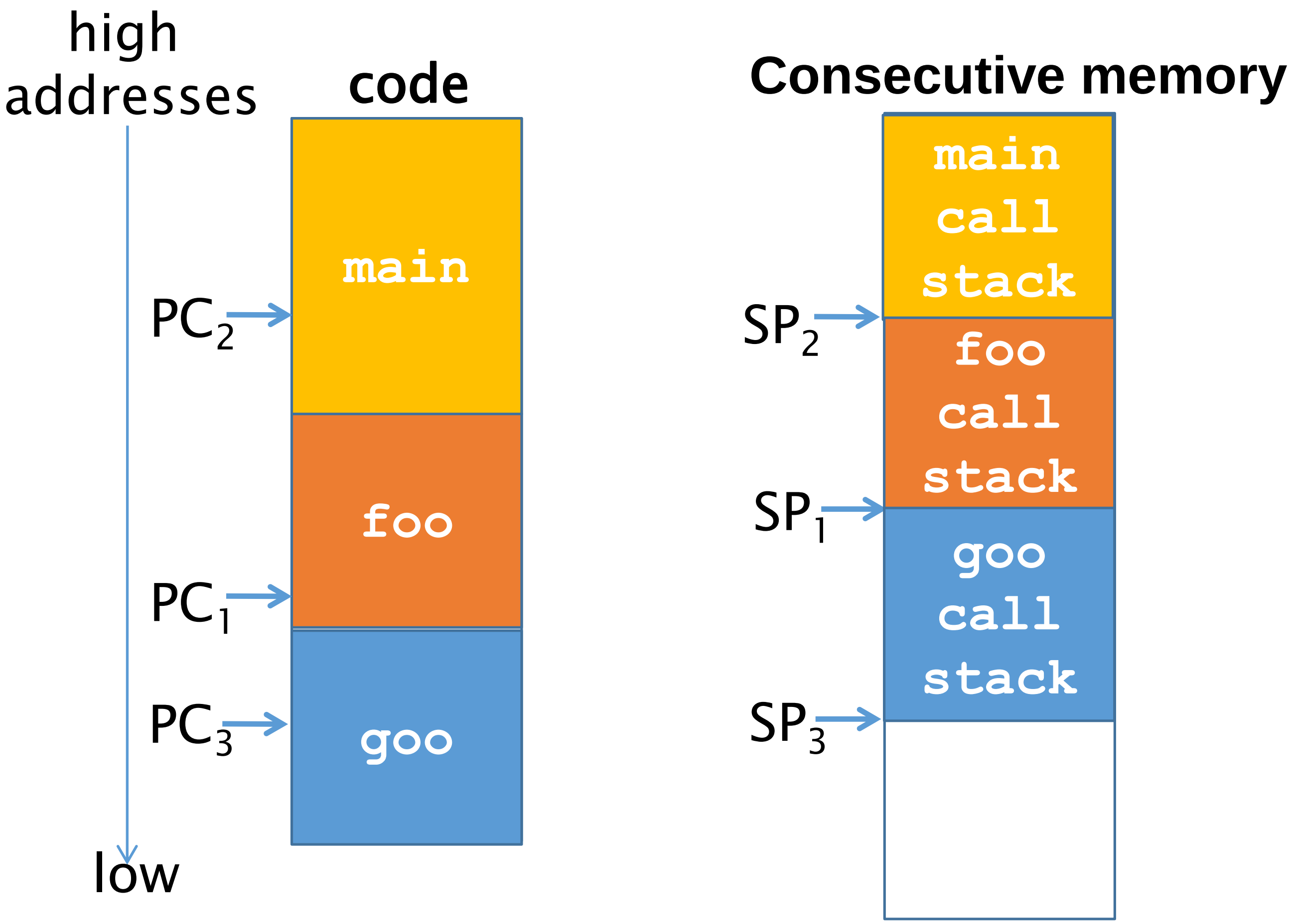


Multi-threaded process



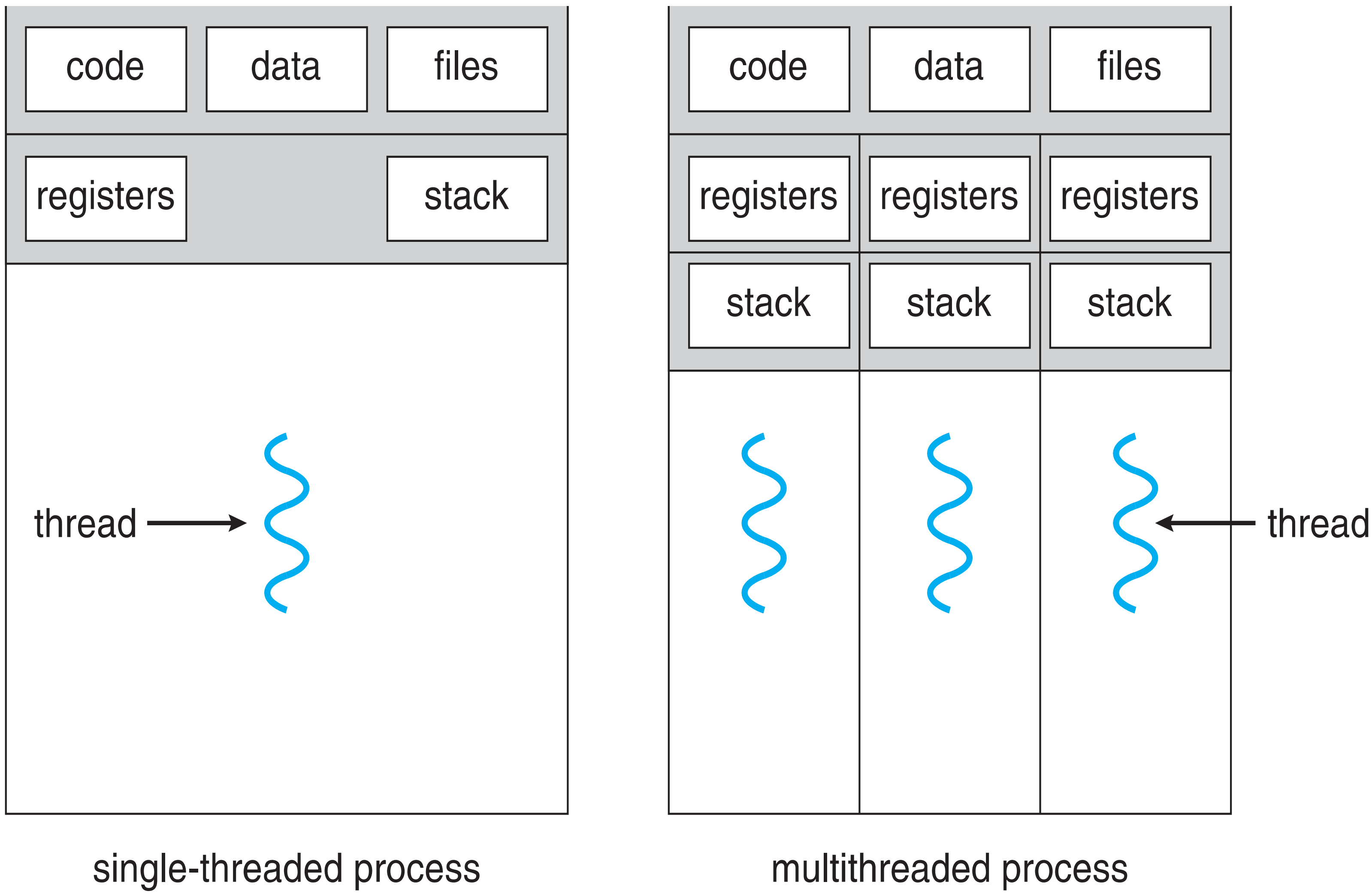
The per-thread call stacks are maintained in **different memory locations** where individual SP points to the respective thread call stack

Multithreaded process



The per-thread call stacks are maintained in **consecutive memory locations** where individual SP points to the respective thread call stack

Multi-threaded versus single threaded



Why Multithreading?

- Responsiveness
- Resource Sharing
- Economy
- Scalability

Matrix multiplication

$$c_{ij} = \sum_{r=1}^{n} a_{ir} \times b_{rj}$$

$$\begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \ldots & \ldots & \ldots \\ a_{m1} & \ldots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & \ldots & b_{1k} \\ \ldots & \ldots & \ldots \\ b_{n1} & \ldots & b_{nk} \end{pmatrix} = \begin{pmatrix} c_{11} & \ldots & c_{1k} \\ \ldots & \ldots & \ldots \\ c_{m1} & \ldots & c_{mk} \end{pmatrix}$$

Slow Matrix multiplication

$$\begin{vmatrix} 2 & 1 & 3 \\ 0 & -1 & 4 \\ 5 & 2 & -2 \end{vmatrix} \times \begin{vmatrix} 3 & 0 & 2 \\ 1 & 4 & -1 \\ 2 & -2 & 5 \end{vmatrix} = \begin{vmatrix} 13 & -8 & 21 \\ 7 & -12 & 21 \\ 17 & 6 & -7 \end{vmatrix}$$

$$\begin{vmatrix} (2*3 + 1*1 + 3*2) & (2*0 + 1*4 + 3*-2) & (2*2 + 1*-1 + 3*5) \\ (0*3 + -1*1 + 4*2) & (0*0 + -1*4 + 4*-2) & (0*2 + -1*-1 + 4*5) \\ (5*3 + 2*1 + -2*2) & (5*0 + 2*4 + -2*-2) & (5*2 + 2*-1 + -2*5) \end{vmatrix}$$

Number of arithmetic operations:

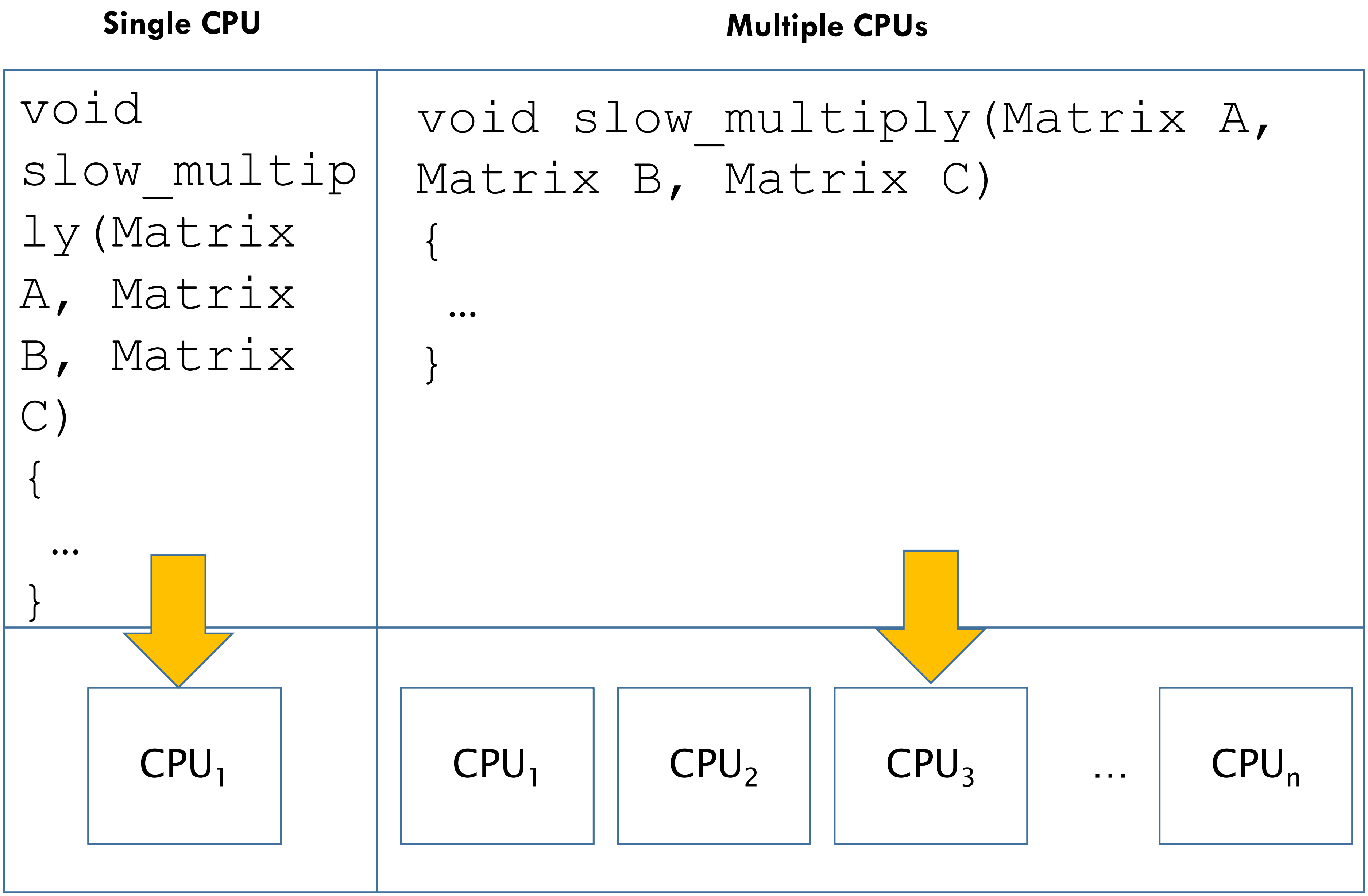
- 27 multiplications and 18 additions = **45**

Slow Matrix multiplication

if input matrices **A = m × n** and **B = p × q** then resultant matrix **C = m × q**

```
void slow_multiply(Matrix A, Matrix B, Matrix C)
{
    for(int i=0;i<m;i++)
    {
        for(int j=0; j<p; j++)
        {
            C[i][j] = 0;

            for(int k=0; k<n; k++)
            {
                C[i][j] = C[i][j]+A[i][k]*B[k][j];
            }
        }
    }
}
```

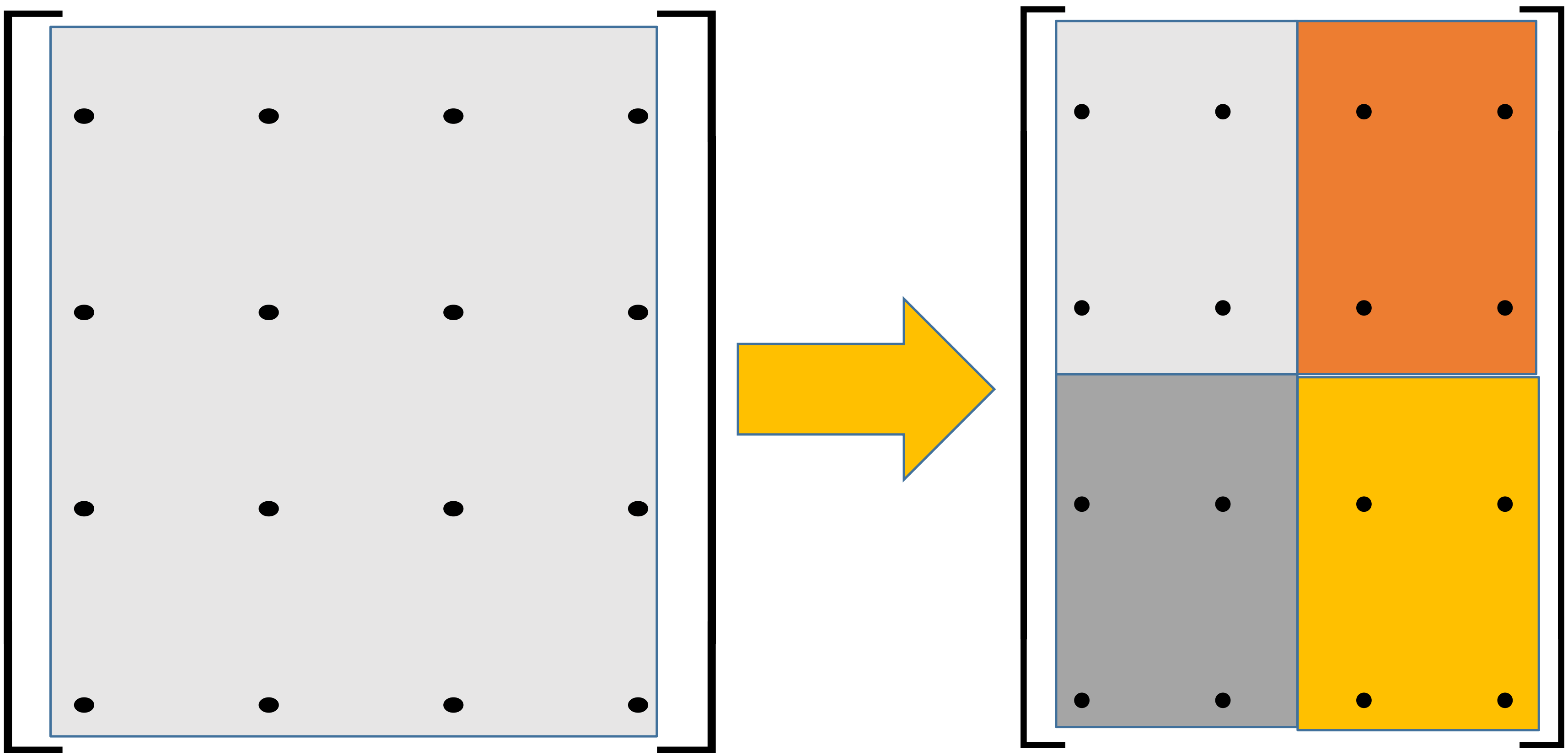
Resources Usage



Doing better

Instead of computing $C_{00}, C_{01}, C_{02}, C_{03}, \ldots.$

Why don't we …. split up the computation?



Faster Multiplication with Multithreading

$$\begin{vmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{vmatrix} \times \begin{vmatrix} b11 & b12 & b13 & b14 \\ b21 & b22 & b23 & b24 \\ b31 & b32 & b33 & b34 \\ b41 & b42 & b43 & b44 \end{vmatrix}$$

$A11 = \begin{vmatrix} a11 & a12 \\ a21 & a22 \end{vmatrix}$ $A12 = \begin{vmatrix} a13 & a14 \\ a23 & a24 \end{vmatrix}$

$A21 = \begin{vmatrix} a31 & a32 \\ a41 & a42 \end{vmatrix}$ $A22 = \begin{vmatrix} a33 & a34 \\ a43 & a44 \end{vmatrix}$

$B11 = \begin{vmatrix} b11 & b12 \\ b21 & b22 \end{vmatrix}$ $B12 = \begin{vmatrix} b13 & b14 \\ b23 & b24 \end{vmatrix}$

$B21 = \begin{vmatrix} b31 & b32 \\ b41 & b42 \end{vmatrix}$ $B22 = \begin{vmatrix} b33 & b34 \\ b43 & b44 \end{vmatrix}$

Thread 1 computes C11 = A11 * B11 + A12 * B21
Thread 2 computes C12 = A11 * B12 + A12 * B22
Thread 3 computes C21 = A21 * B11 + A22 * B21
Thread 4 computes C22 = A21 * B12 + A22 * B22

$C11 = \begin{vmatrix} c11 & c12 \\ c21 & c22 \end{vmatrix}$ $C12 = \begin{vmatrix} c13 & c14 \\ c23 & c24 \end{vmatrix}$

$C21 = \begin{vmatrix} c31 & c32 \\ c41 & c42 \end{vmatrix}$ $C22 = \begin{vmatrix} c33 & c34 \\ c43 & c44 \end{vmatrix}$

The individual elements of C11, C12, C21, and C22 are computed by summing the products of corresponding elements from the multiplication of their respective A and B blocks.

final matrix C:

$$\begin{vmatrix} c11 & c12 & c13 & c14 \\ c21 & c22 & c23 & c24 \\ c31 & c32 & c33 & c34 \\ c41 & c42 & c43 & c44 \end{vmatrix}$$

Faster Multiplication with Multithreading

Additional matrices or data structures may be required for efficient parallelization, depending on your specific implementation strategy.

Procedure multiply(C, A, B):

- Base case: if $n = 1$, set $c_{11} \leftarrow a_{11} \times b_{11}$ (or multiply a small block matrix).
- Otherwise, allocate space for a new matrix T of shape $n \times n$, then:
  - Partition A into A_{11}, A_{12}, A_{21}, A_{22}.
  - Partition B into B_{11}, B_{12}, B_{21}, B_{22}.
  - Partition C into C_{11}, C_{12}, C_{21}, C_{22}.
  - Partition T into T_{11}, T_{12}, T_{21}, T_{22}.
  - Parallel execution:
    - *Fork* multiply(C_{11}, A_{11}, B_{11}).
    - *Fork* multiply(C_{12}, A_{11}, B_{12}).
    - *Fork* multiply(C_{21}, A_{21}, B_{11}).
    - *Fork* multiply(C_{22}, A_{21}, B_{12}).
    - *Fork* multiply(T_{11}, A_{12}, B_{21}).
    - *Fork* multiply(T_{12}, A_{12}, B_{22}).
    - *Fork* multiply(T_{21}, A_{22}, B_{21}).
    - *Fork* multiply(T_{22}, A_{22}, B_{22}).
  - *Join* (wait for parallel forks to complete).
  - add(C, T).
  - Deallocate T.

Two kinds of stack

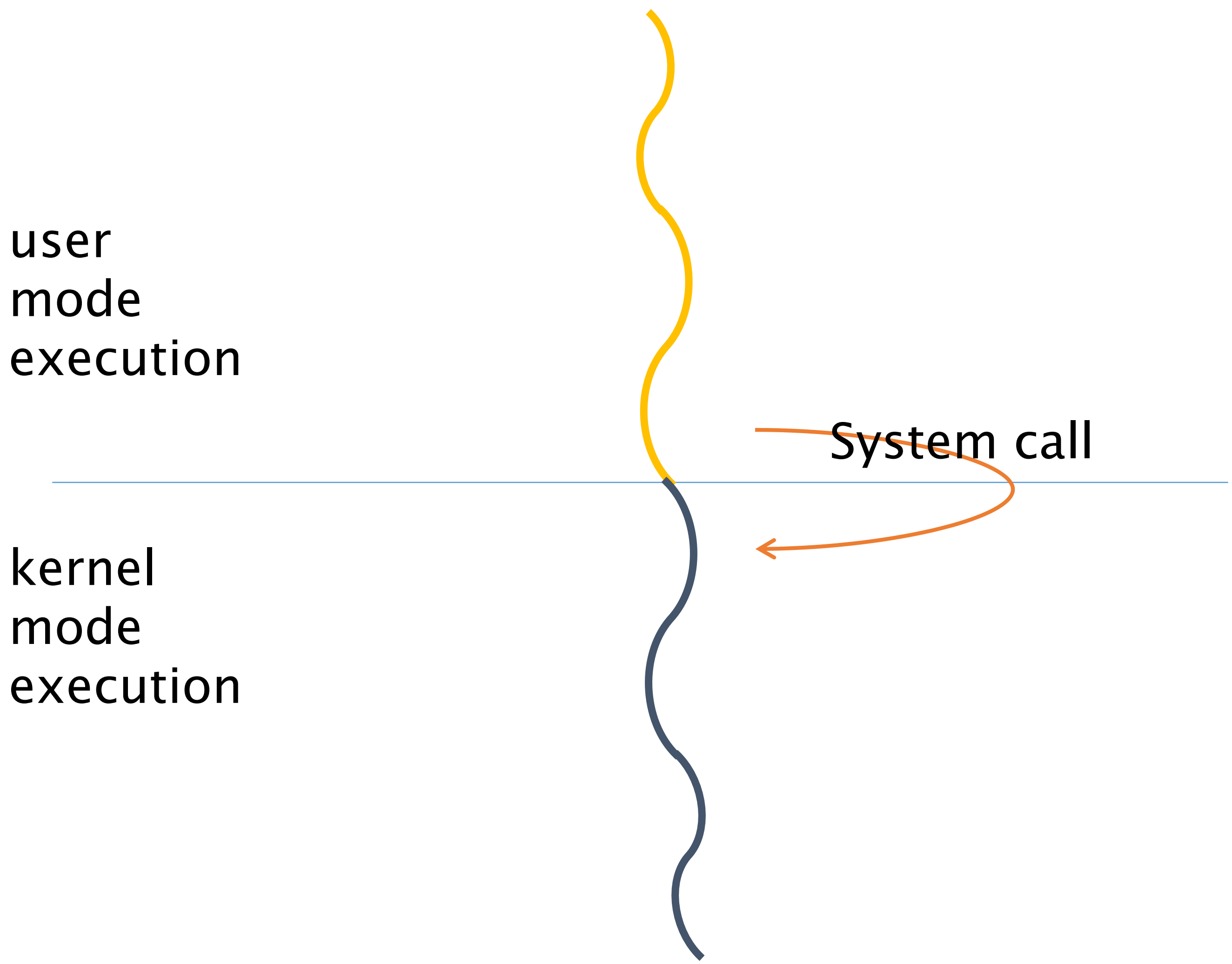
- **User Stack**
  - Used for user-level programs
- **Kernel Stack**
  - Used by system-calls

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
  - ❑ Three primary thread libraries:
    - POSIX **Pthreads**
    - Windows threads
    - Java threads

- **Kernel threads** - Supported by the Kernel
  - ❑ Examples – virtually all general purpose operating systems, including:
    - Windows
    - Linux
    - Mac OS X

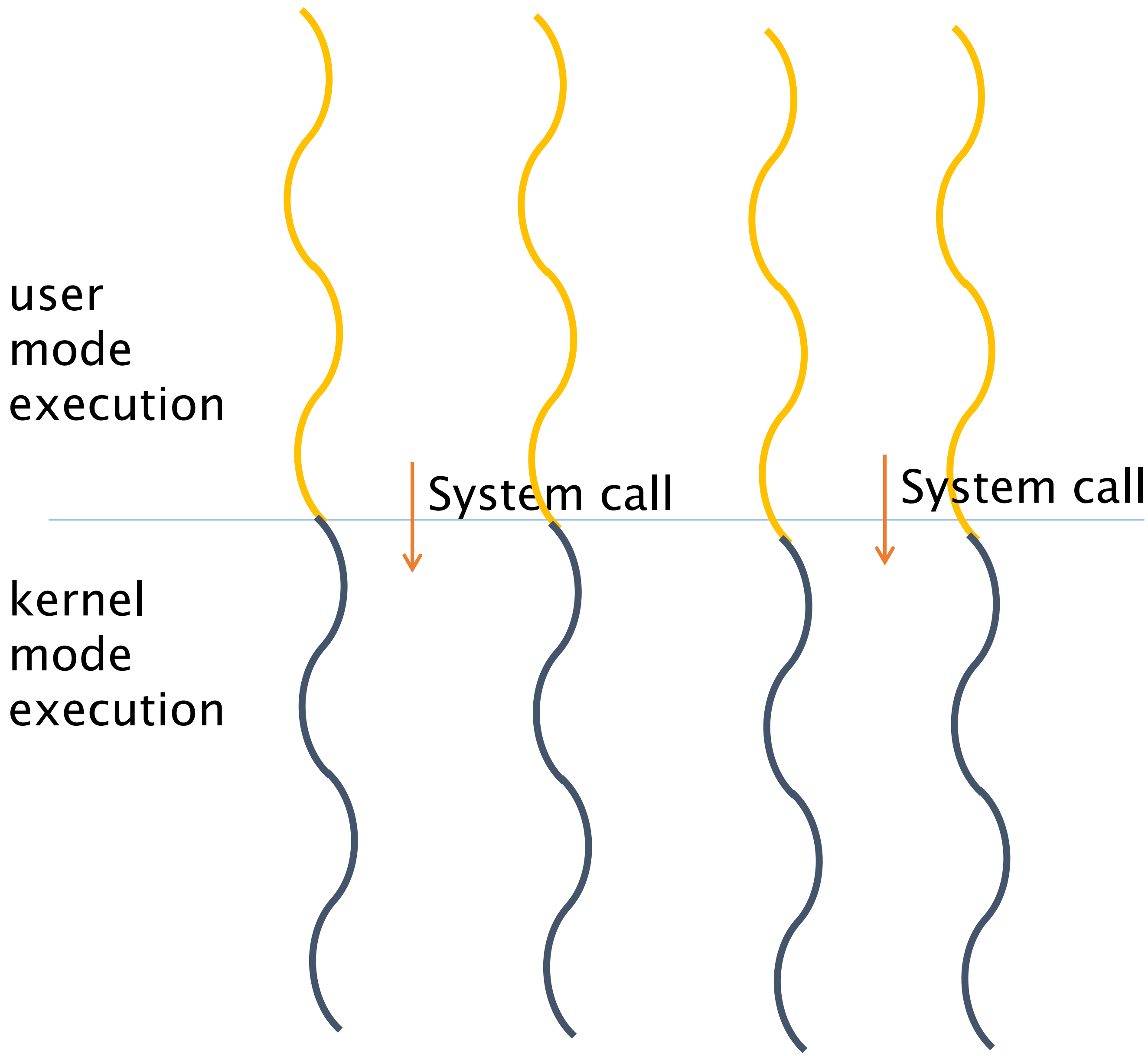
Single thread execution



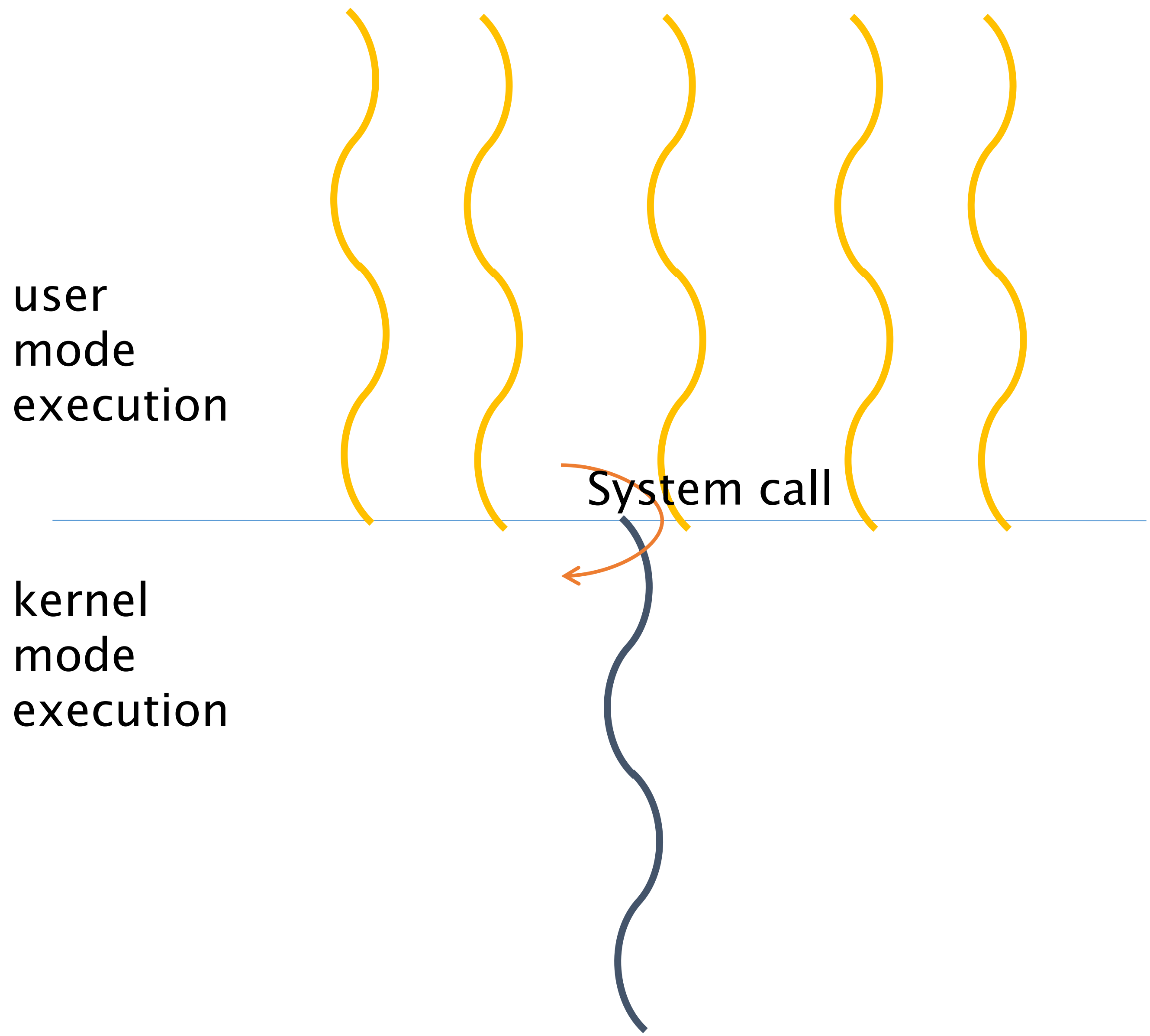
Multithreading Model

- Ratio of User Level Threads to Kernel Level Threads in a Process:
  - 1 : 1
  - M : 1
  - M : N

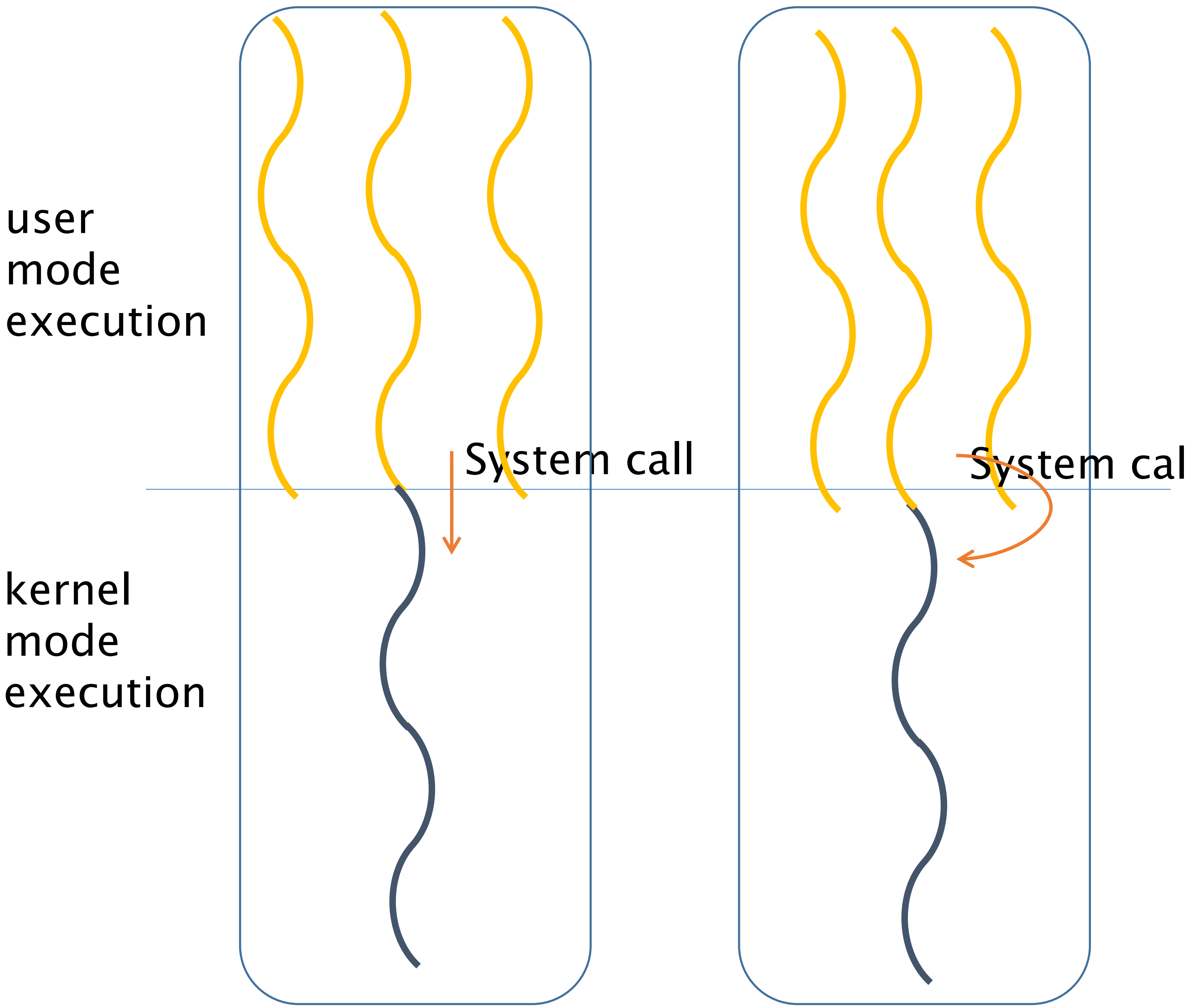
1:1 Thread Execution



M : 1 Model



M:N Thread Execution



Threads API

- **Basic**
  - Thread Creation
    - creating and initializing a new thread of execution within a program.
  - Thread Joining
    - one thread waiting for another thread to finish its execution before proceeding further.
  - Thread Exiting
    - allowing the OS to clean up resources associated with the thread and potentially freeing up system resources.

- **Advanced**
  - Processor Affinity
    - concept of binding a thread or a process to a specific processor or a subset of processors.
  - Yield CPU
    - thread voluntarily gives up its current time slice or quantum on the CPU to allow other threads to execute.

Thread Creating, Joining and Exiting

- **Linux**
  - pthread_creat( ), pthread_join( ), pthread_exit( )
- **Win32**
  - CreateThread( )
  - WaitForSingleObject( )
  - ExitThread( )

pthread Creating, Joining and Exiting

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

// Function to be executed by each thread
void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    printf("Thread %d says hello!\n", thread_id);

    // Exit the thread
    pthread_exit(NULL);
}

int main() {
    pthread_t thread1, thread2;
    int id1 = 1, id2 = 2;

    // Create thread 1
    if (pthread_create(&thread1, NULL, thread_function,
&id1) != 0) {
        fprintf(stderr, "Error creating thread 1\n");
        return 1;
    }

// Create thread 2
    if (pthread_create(&thread2, NULL,
thread_function, &id2) != 0) {
        fprintf(stderr, "Error creating thread 2\n");
        return 1;
    }

    // Wait for threads to finish
    if (pthread_join(thread1, NULL) != 0) {
        fprintf(stderr, "Error joining thread 1\n");
        return 1;
    }

    if (pthread_join(thread2, NULL) != 0) {
        fprintf(stderr, "Error joining thread 2\n");
        return 1;
    }

    printf("Both threads have finished.\n");

    return 0;
}
```

Synchronization

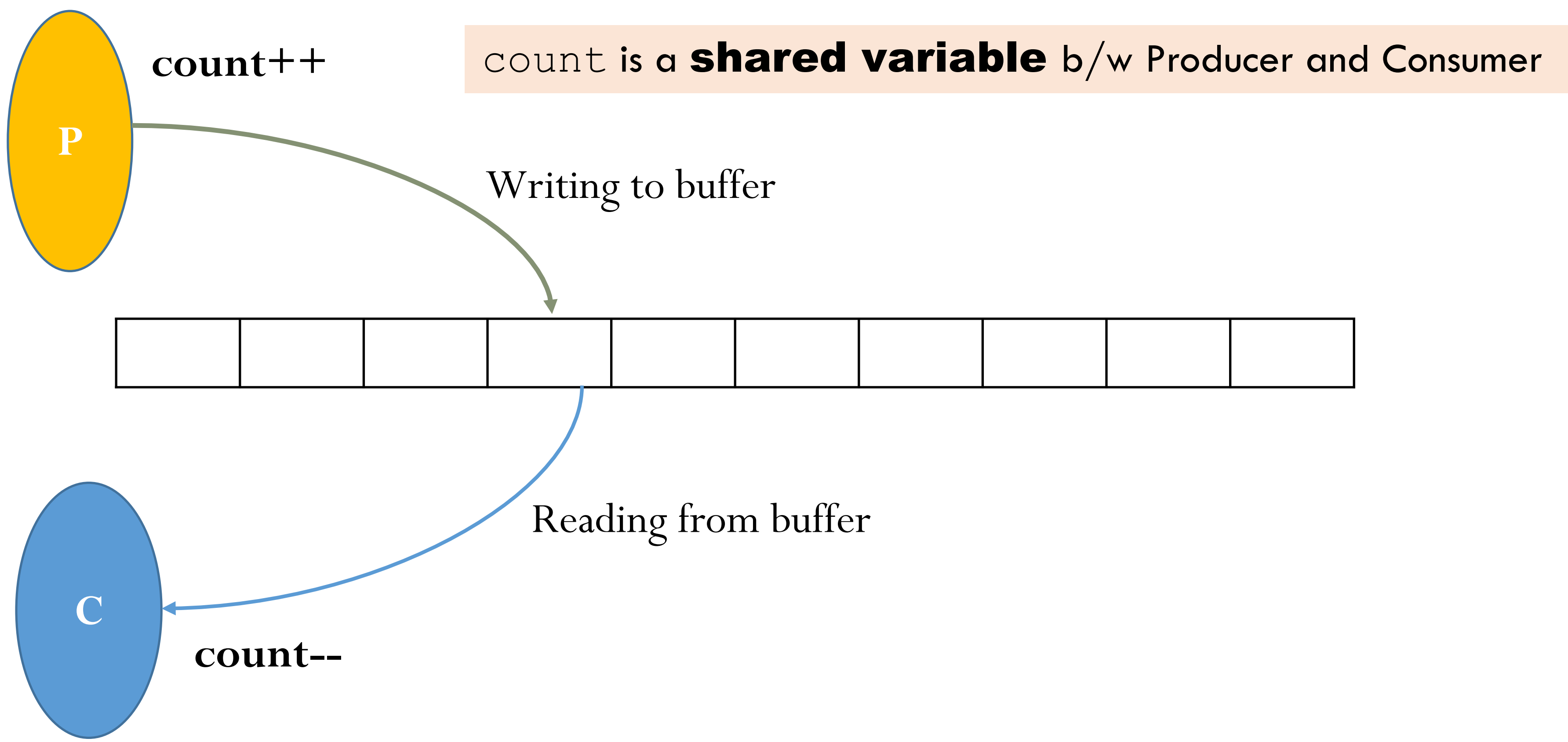
Race Condition

Multiple threads or processes **access shared data concurrently**, and the final outcome depends on the **specific order in which their operations are interleaved.**

- Involves Shared Data
- Produces Unpredictable Outcome
- Result of Non-atomic Operations on Shared Data
- Result of Lack of Synchronization

Producer – Consumer Problem

- A scenario where a producer checks if the buffer is not full and decides to insert an item. A consumer checks if the buffer is not empty and decides to remove an item.



Examples

- Print Spooling in Operating Systems
- Message Passing in Communication Systems
- Traffic Management in Networking
- Event queues
- Streaming (e.g., video)

Producer – Consumer Problem

Bounded Buffer:

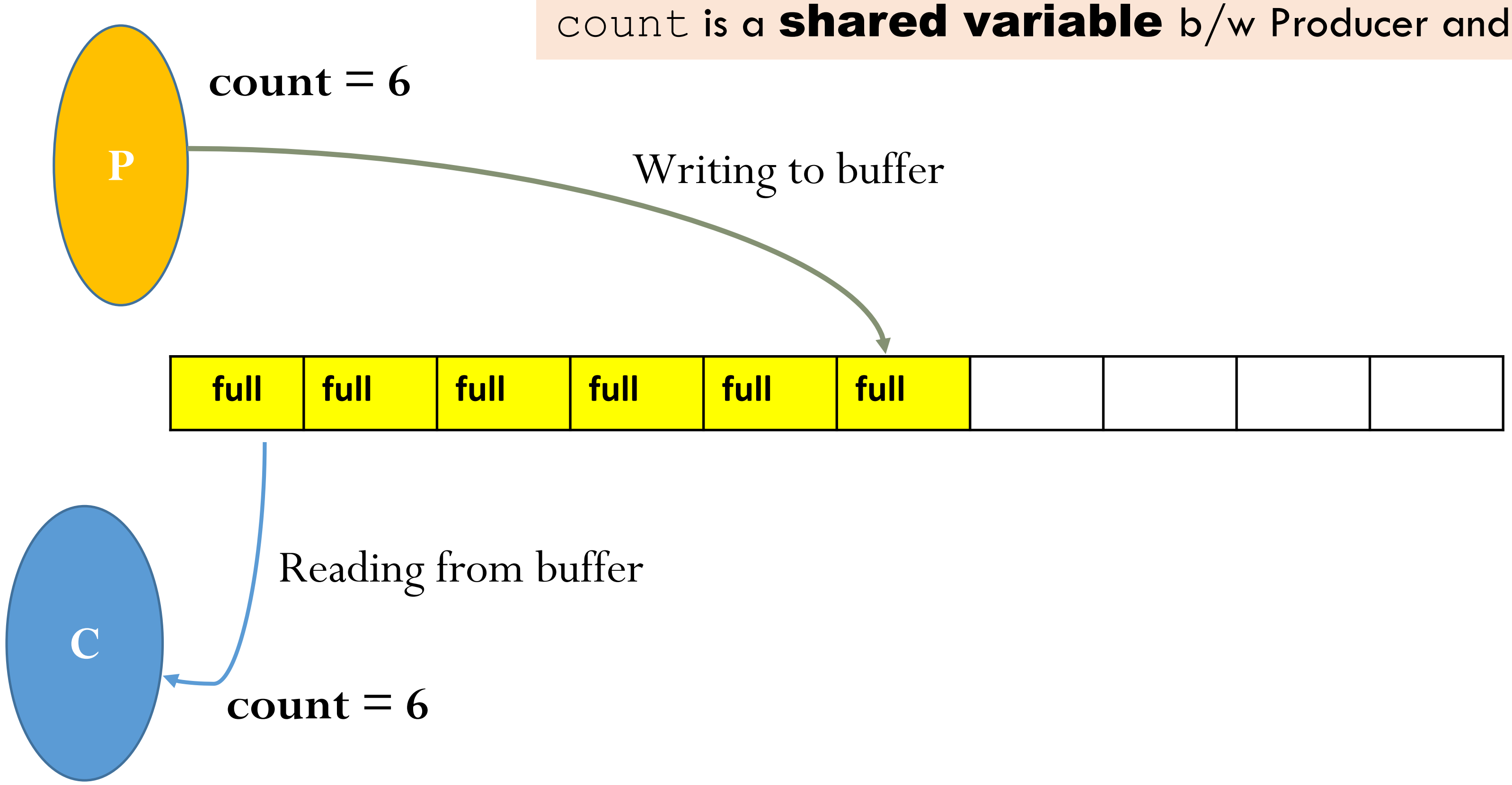
- *Fixed Size:* A bounded buffer has a fixed, limited capacity. It can hold a specific number of items at a time.
- *Blocking on Overflow:* If the buffer is full, a producer must wait until there is space available (blocking on overflow). Similarly, if the buffer is empty, a consumer must wait until there is data available.
- *Synchronization:* Bounded buffers typically require synchronization mechanisms, such as semaphores or mutexes, to coordinate access and prevent **race conditions** between producers and consumers.

Producer – Consumer Problem: no-race condition

Suppose the Producer **completes** writing an item to the buffer, followed by the Consumer retrieving an item from the buffer.

1. The Producer has written 6 items to the buffer

`count` is a **shared variable** b/w Producer and Consumer

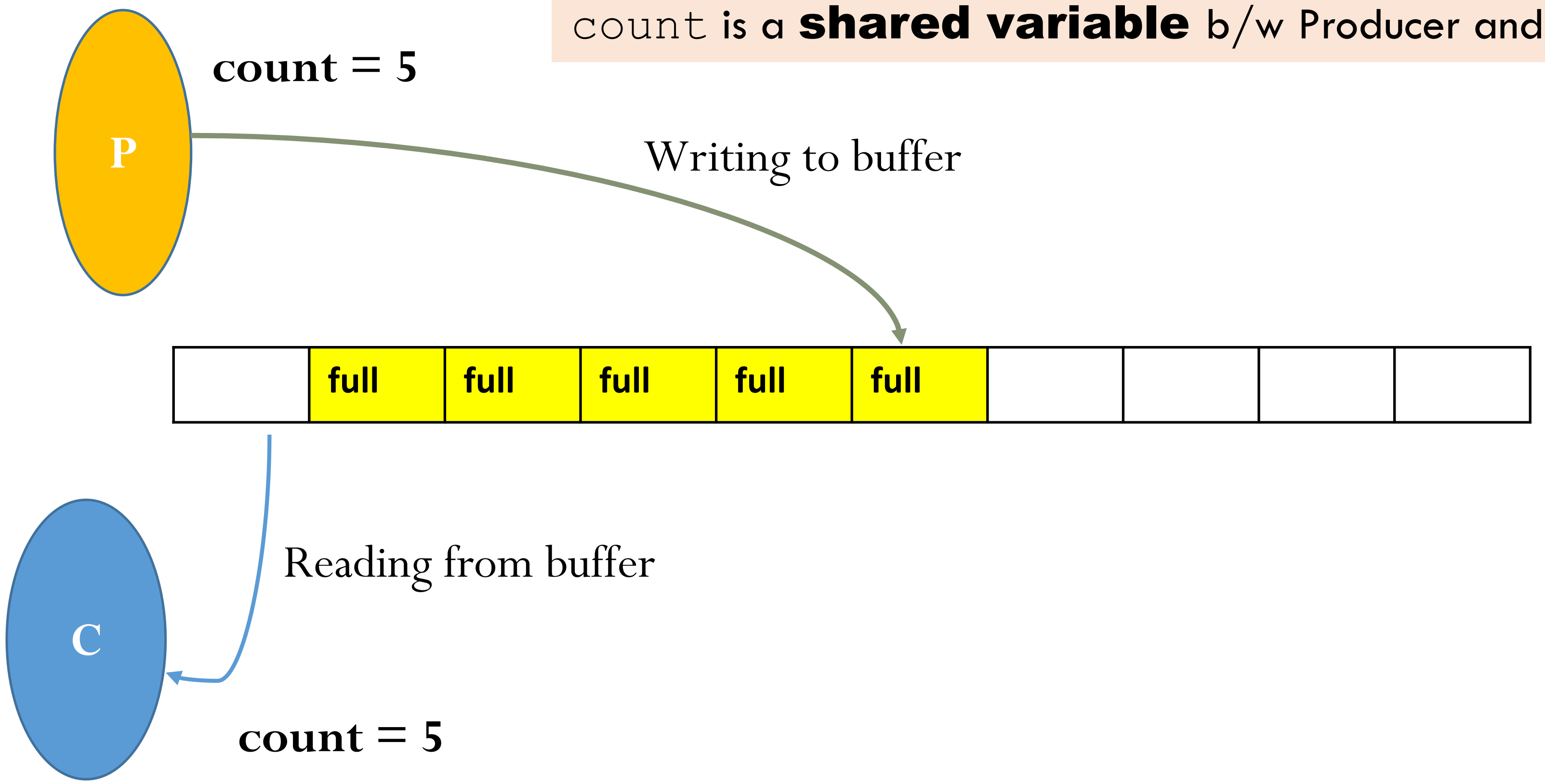


Producer – Consumer Problem:
no-race condition

Suppose the Producer **completes** writing an item to the buffer, followed by the Consumer retrieving an item from the buffer.

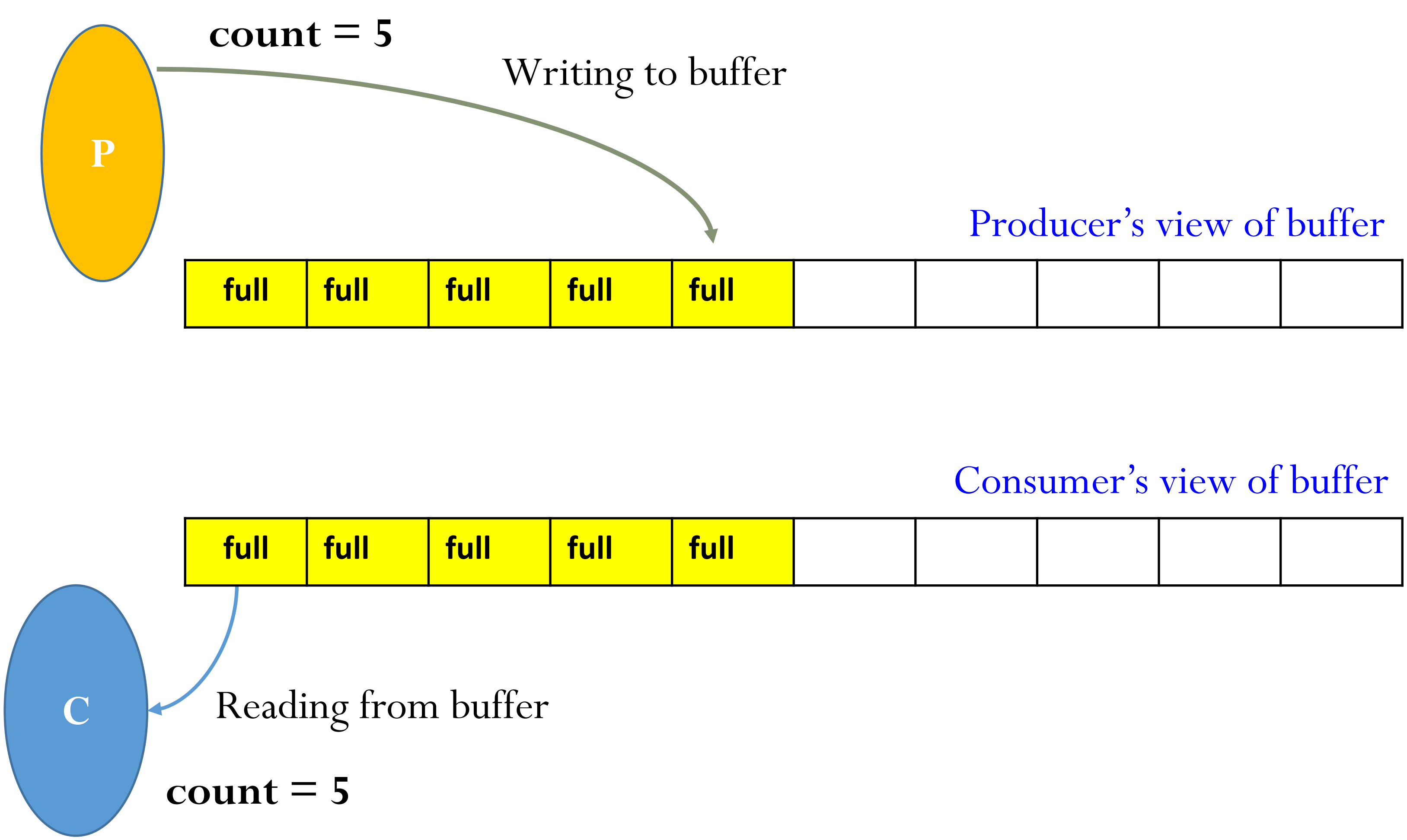
1. The Producer has written 6 items to the buffer
2. The Consumer has retrieved 1 item from the buffer

`count` is a **shared variable** b/w Producer and Consumer



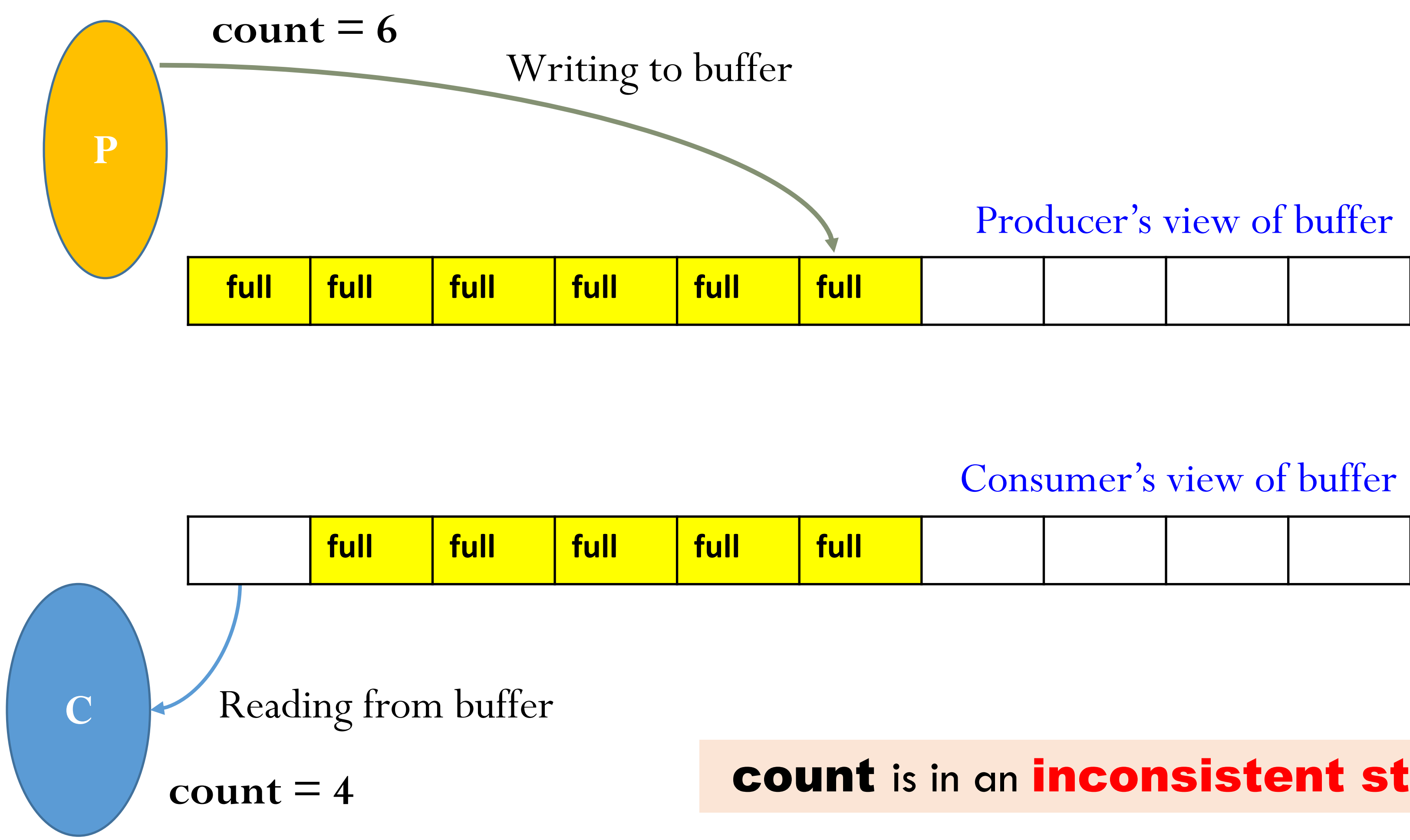
Producer – Consumer Problem: race condition

1. The Producer has written 5 items to the buffer so far.



Producer – Consumer Problem: race condition

1. The current value of **`count=5`** after the Producer has written 5 items to the buffer so far.
2. The Producer is **now writing** 6th item to the buffer (`count++`).
3. The Consumer **at the same time** is retrieving an item from the buffer (`count--`).



count is in an **inconsistent state !!!**

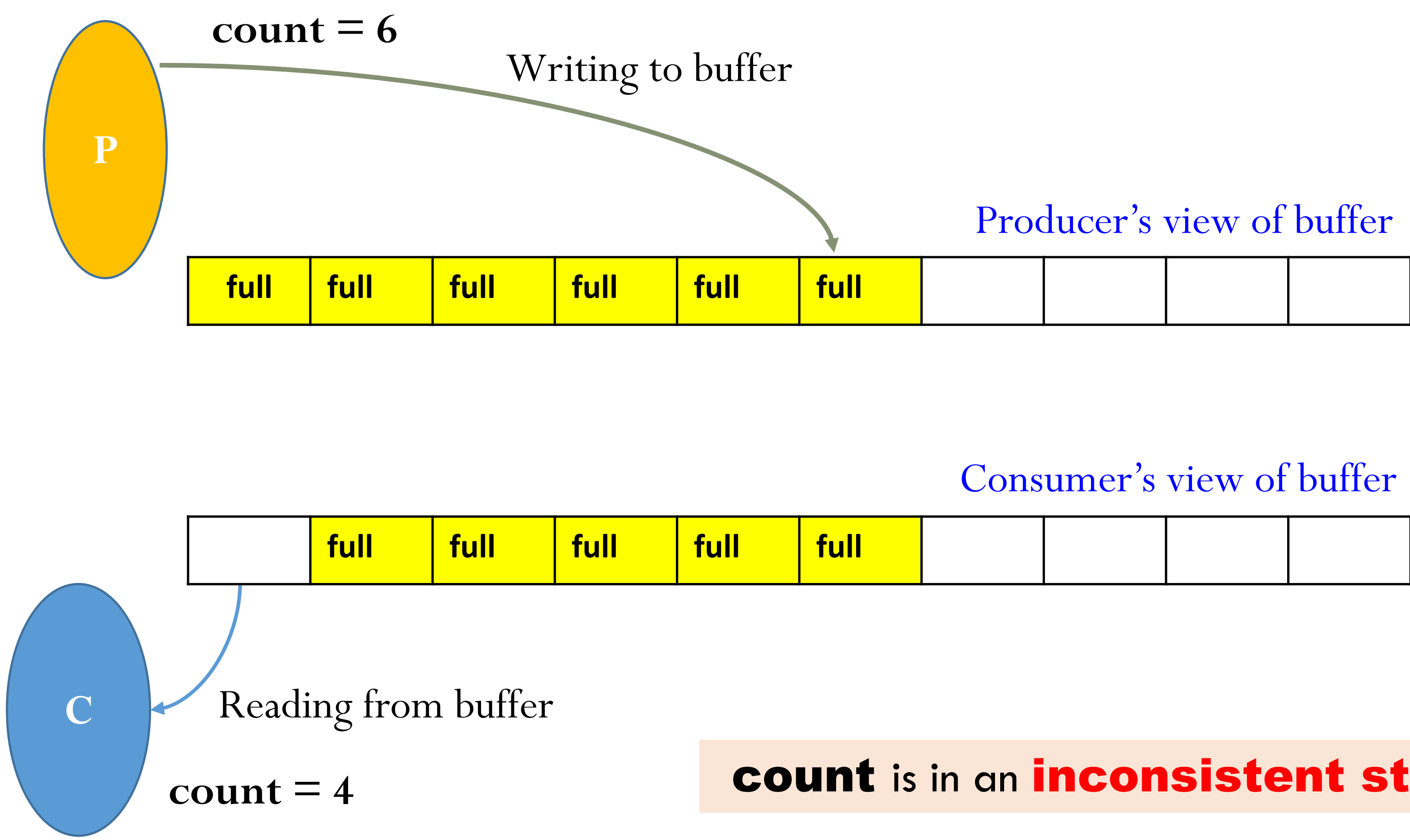
Producer – Consumer Problem: race condition

Compound operation `count ++`

```
load count, r0
add r0, r0, 1
store r0, count
```

Compound operation `count --`

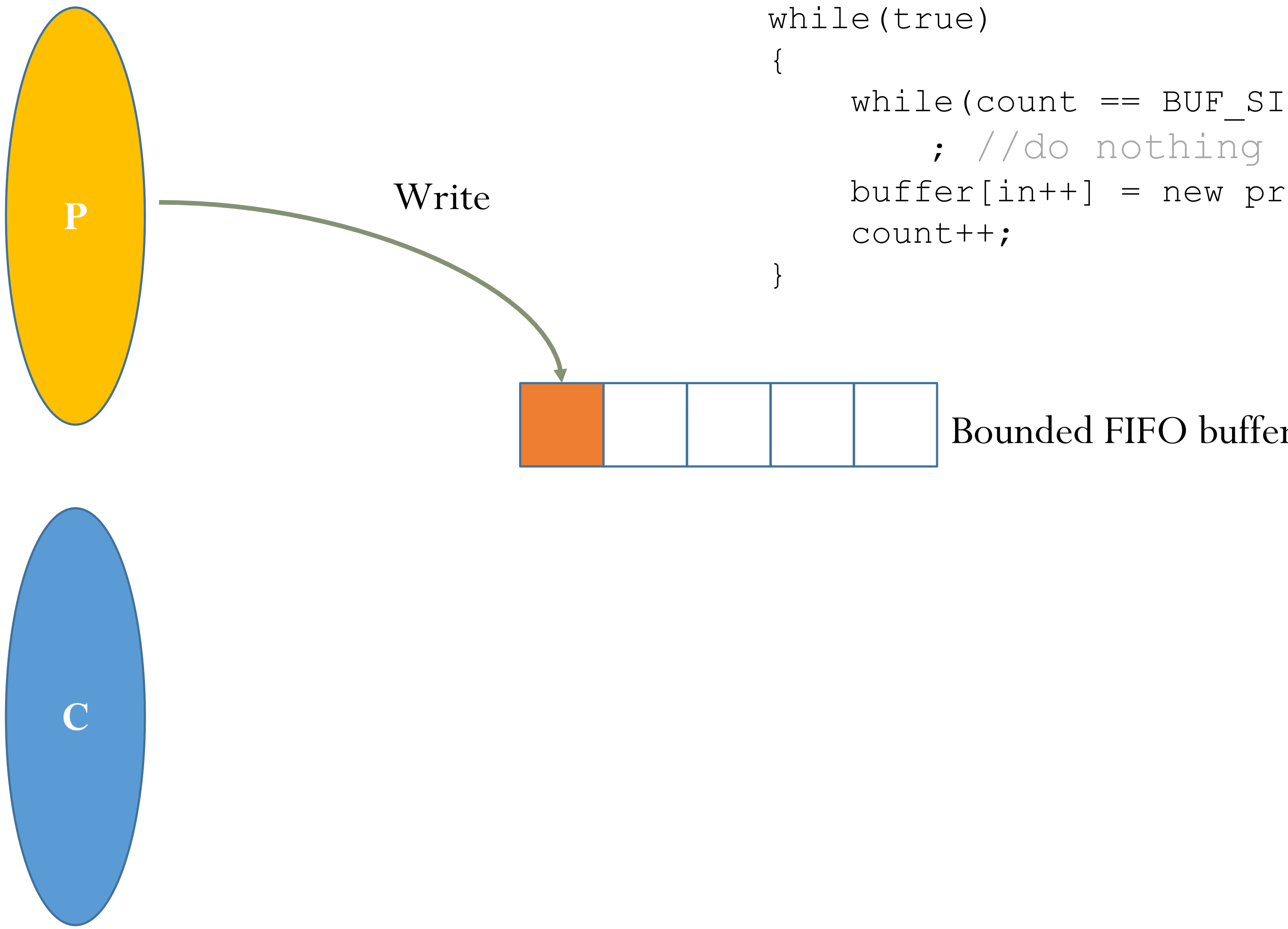
```
load count, r0
sub r0, r0, 1
store r0, count
```



count is in an **inconsistent state !!!**

Implementation:

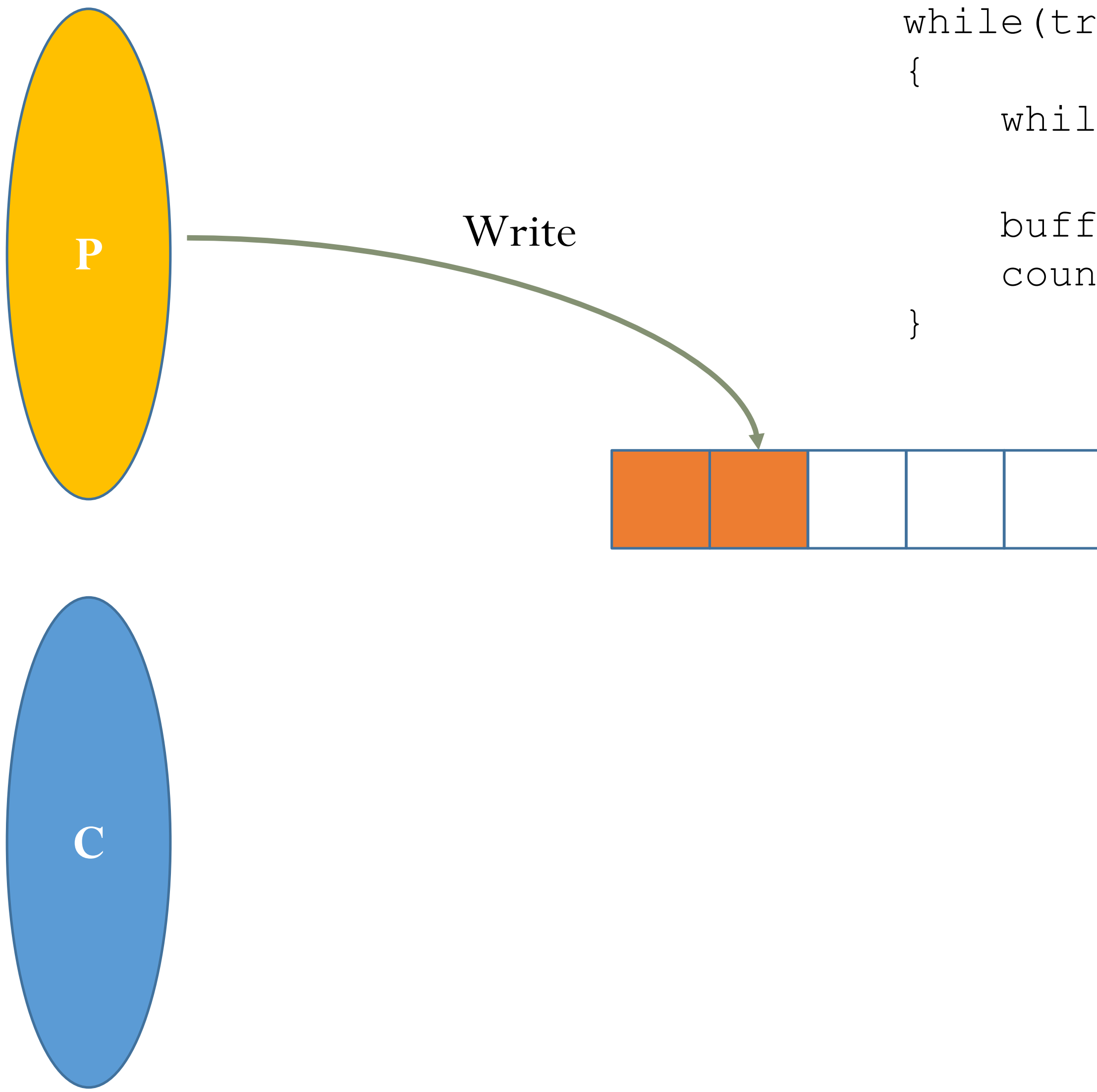
Producer writes to buffer




```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    buffer[in++] = new product;
    count++;
}
```

Implementation:
Buffer filling

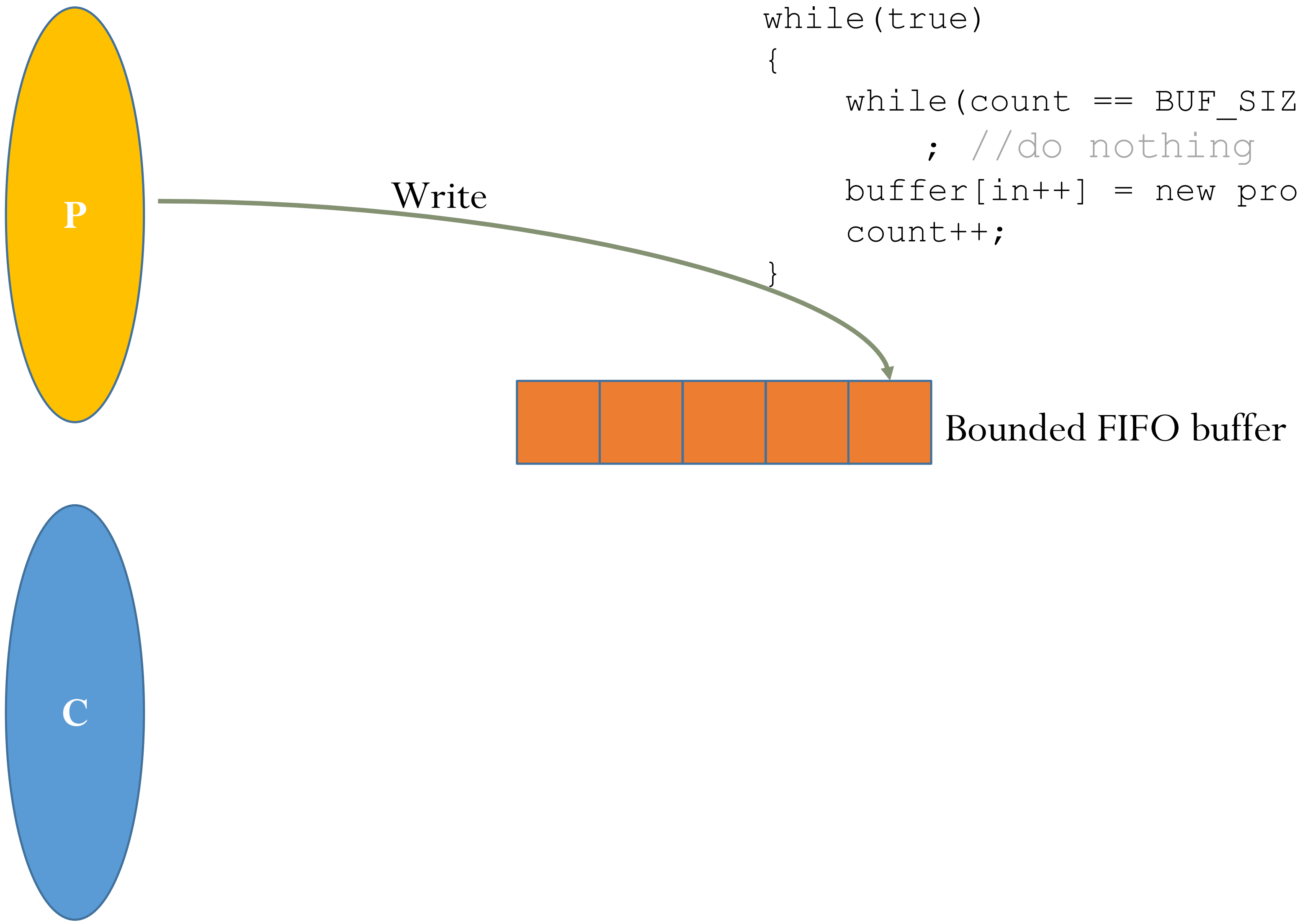
```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    buffer[in++] = new product;
    count++;
}
```



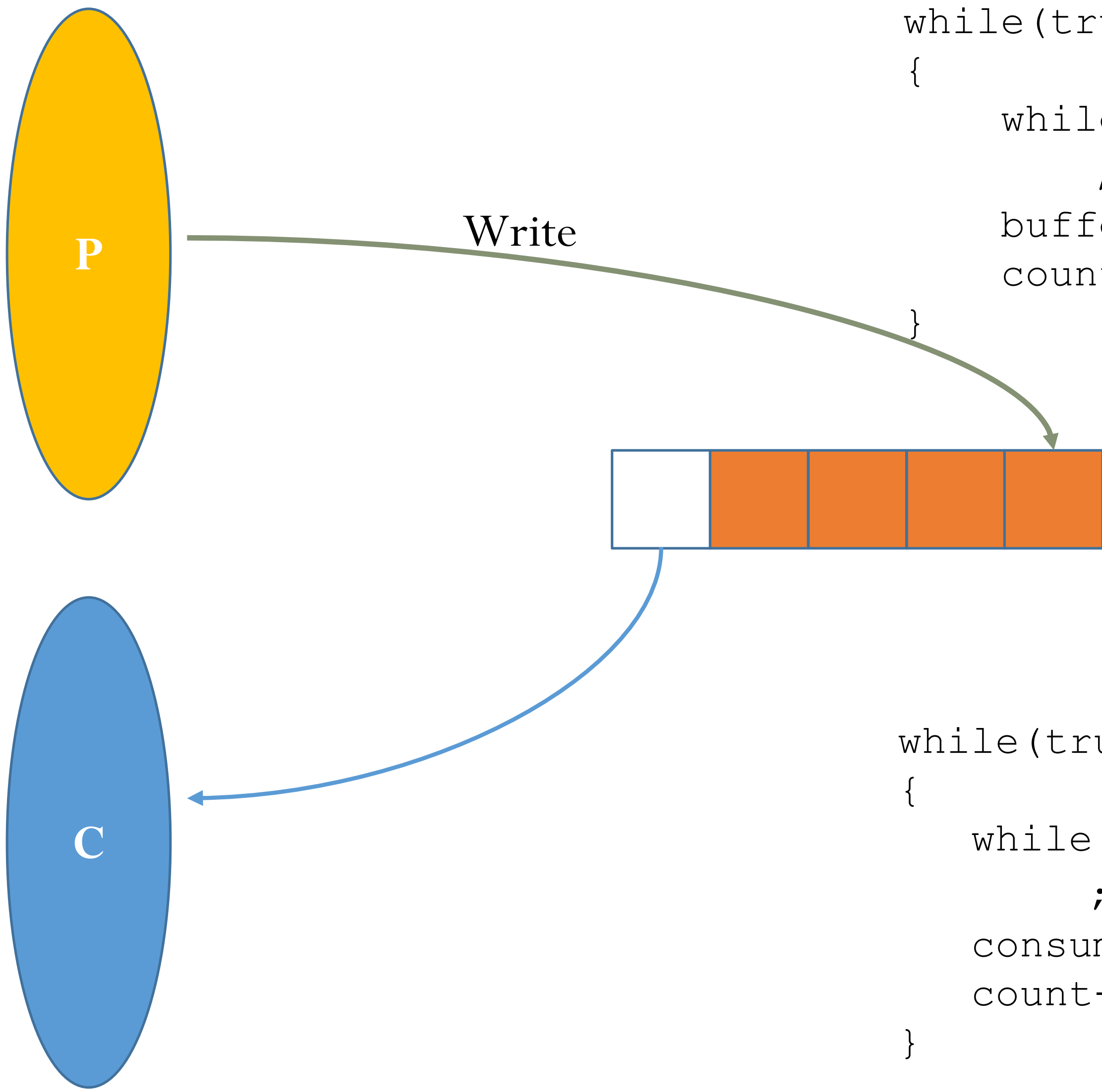
Implementation:

Stop when the buffer is full!

```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    buffer[in++] = new product;
    count++;
}
```



Implementation:
Consumer reads buffer



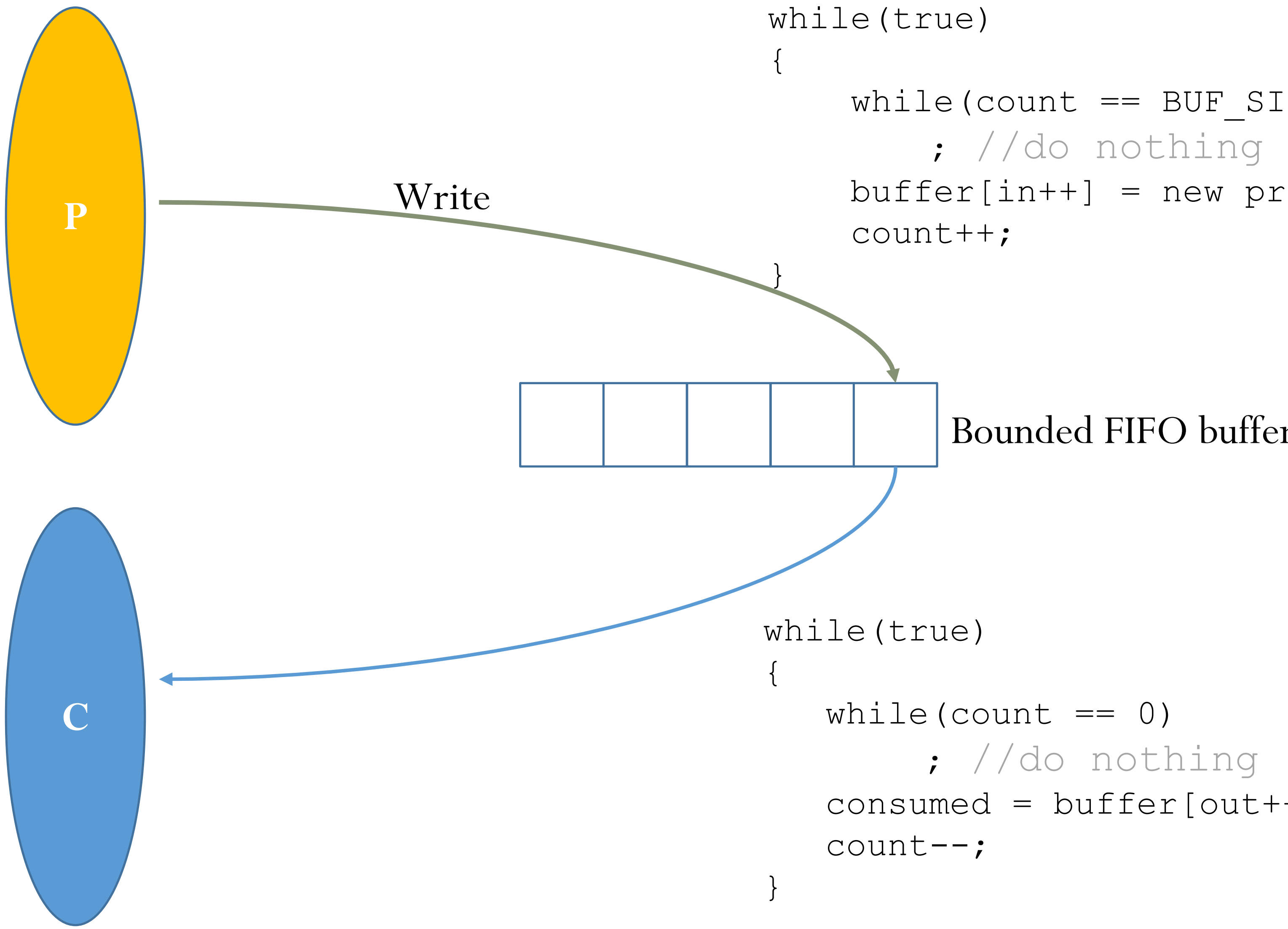

```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    buffer[in++] = new product;
    count++;
}
```

```
while(true)
{
   while(count == 0)
        ; //do nothing
   consumed = buffer[out++];
   count--;
}
```

Implementation:

Stop if buffer is empty!

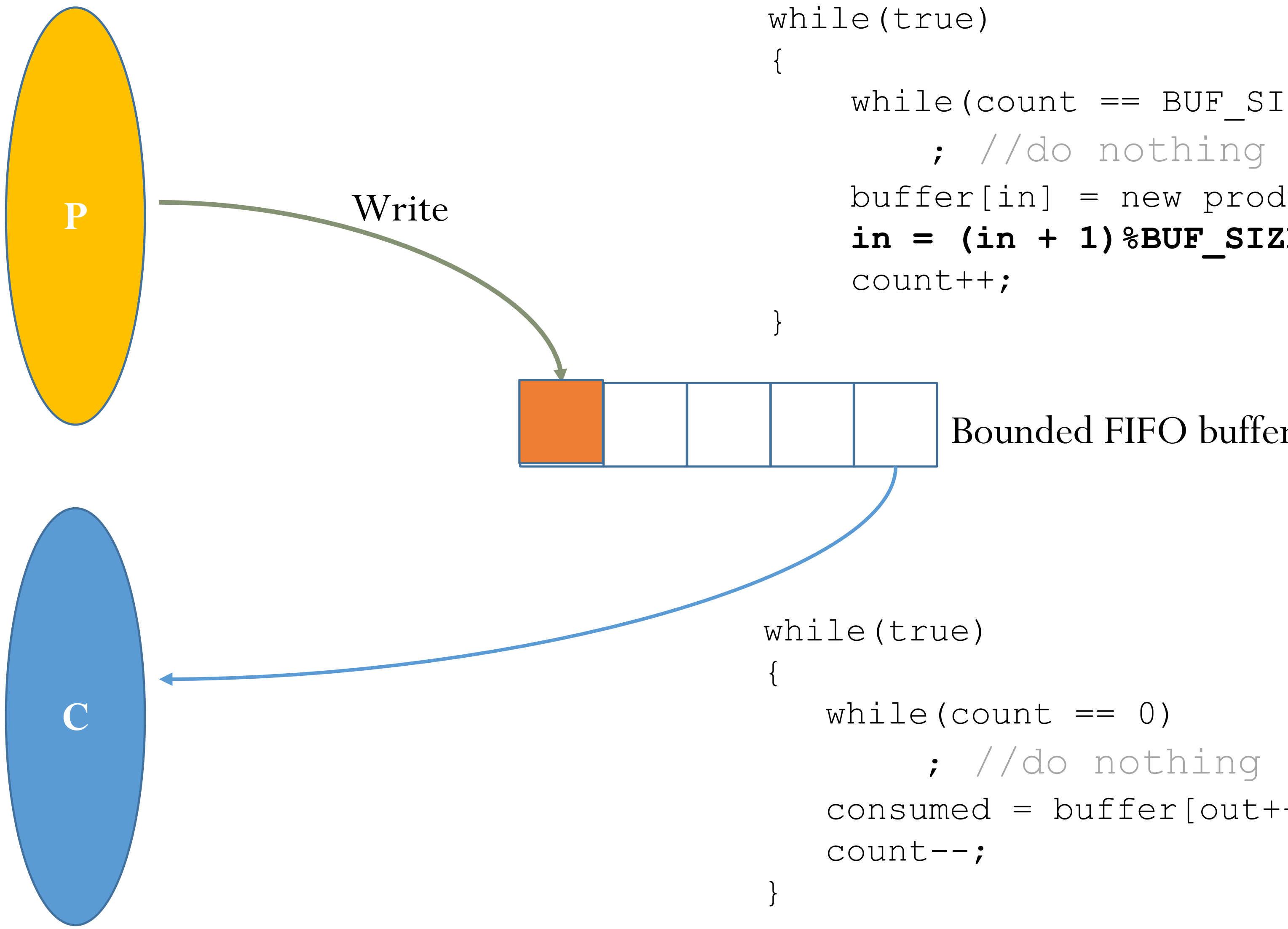
```
while(true)
{
     while(count == BUF_SIZE)
          ; //do nothing
     buffer[in++] = new product;
     count++;
}
```




```
while(true)
{
    while(count == 0)
          ; //do nothing
    consumed = buffer[out++];
    count--;
}
```

Implementation:

Wrap around - writing

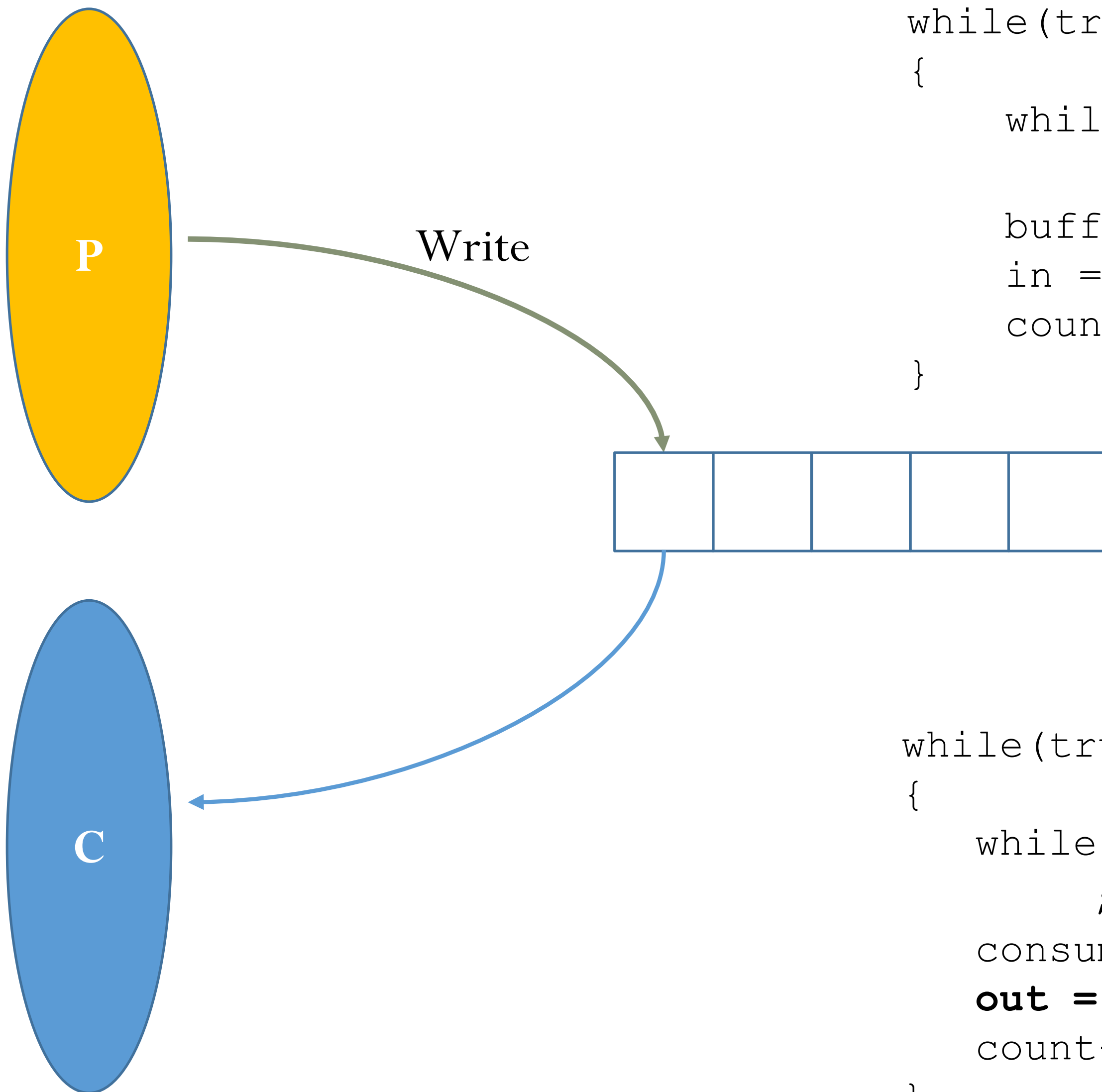



```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    count++;
}
```

```
while(true)
{
   while(count == 0)
        ; //do nothing
   consumed = buffer[out++];
   count--;
}
```

Implementation:

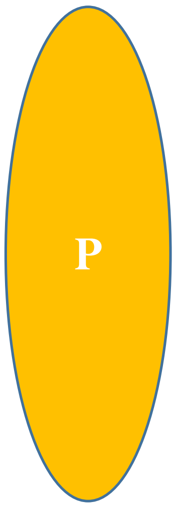
Wrap around - reading

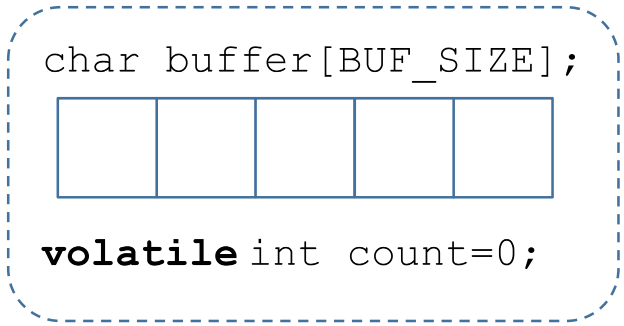



```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    count++;
}
```

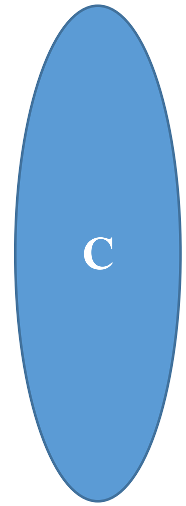
```
while(true)
{
   while(count == 0)
        ; //do nothing
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   count--;
}
```

Volatile keyword





Shared Memory



Producer:
1. Insert item (wrap around)
2. Should stop when buffer is full

Consumer:
1. Read item (wrap around)
2. Should stop when buffer is empty

The **volatile** keyword ***prevents the compiler*** from making certain optimizations such as caching the value of count in a **register** and always requires it to fetch the latest value from **memory**.

* note that the volatile keyword ***doesn't provide atomicity for compound operations*** (e.g., incrementing count).

Volatile keyword

```
volatile int count = 0;

// Thread 1
count++;

// Thread 2
count++;
```

- **Volatile** ensures that the compiler fetches the latest value of `count` from memory each time it is accessed, preventing the use of cached values.
  - However, if two threads execute `count++` **concurrently**, there's a risk of a race condition.

The sequence of operations for the race condition might look like this:

1. Thread-1 reads `count` (let's say, it's **0**).
2. Thread-2 reads `count` (still **0**).
3. Thread-1 increments its *local copy* (now **1**).
4. Thread-2 increments its *local copy* (also **1**).
5. Thread-1 writes the updated value back to `count` (now **1**).
6. Thread-2 writes the updated value back to `count` (still **1**).

* In this case, both threads read the same initial value, increment their local copies independently, and write the same updated value back to `count`, effectively **losing one of the increments**. In this case, we **need a synchronization mechanism.**

What's the problem?

Consumer

```
while(true)
{
   while(count == 0)
        ; //do nothing
   consumed = buffer[out++];
   out = (out + 1) %BUF_SIZE;
   count--;
}
```

```
load count, r0
sub r0, r0, 1
store r0, count
```

Producer

```
while(true)
{
    while(count == BUF_SIZE)
        ;
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    count++;
}
```

```
load count, r0
add r0, r0, 1
store r0, count
```

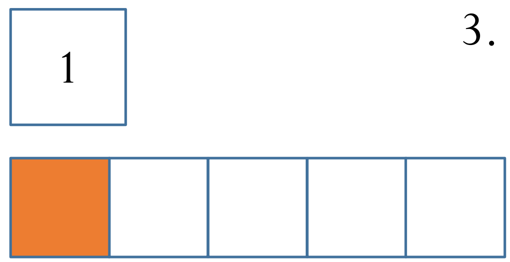
Main problem: updating of shared memory "`count`" is non-atomic

Illustration of problem - I

Scenario:

1. The Producer has just put one item in the buffer
2. Now the `count` is **1**
3. The control goes to Consumer now

count



Consumer

```
load count, r0
sub r0, r0, 1
store r0, count
```

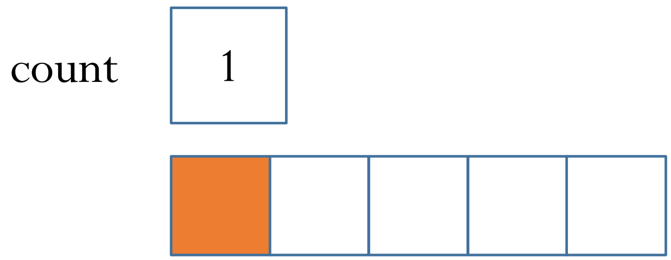
Producer

```
load count, r0
add r0, r0, 1
store r0, count
```

Illustration of problem - II

Scenario continued:

1. The Consumer has executed 2 instructions
2. Before storing back, an interrupt occures and the control switches to the Producer



r0 0

Consumer

```
load count, r0
sub r0, r0, 1
store r0, count
```

Producer

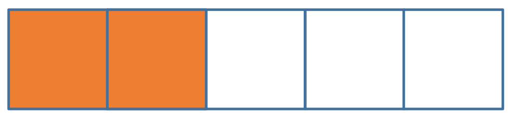
```
load count, r0
add r0, r0, 1
store r0, count
```

Illustration of problem - III

Scenario continued:

1. Producer successfully incremented `count` to **2**
2. An interrupt occurs and the control switches back to the Consumer

count 2



r0 0

Consumer

```
load count, r0
sub r0, r0, 1
store r0, count
```

Producer

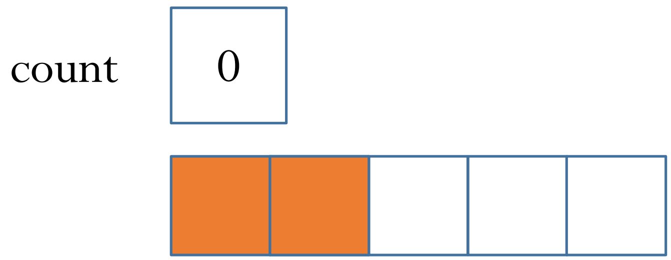
```
load count, r0
add r0, r0, 1
store r0, count
```

Illustration of problem - IV

Scenario continued:

1. Consumer writes **0** into the `count`

   Wrong value!



Consumer

r0 0

```
load count, r0
sub r0, r0, 1
store r0, count
```

Producer

```
load count, r0
add r0, r0, 1
store r0, count
```

The problem

- Non- atomicity
  - Divisible operation
  - Interruptible
- Instruction interleaving

Critical Section (CS)

Critical Section (CS)

Consumer

```
while(true)
{
   while(count == 0)
        ; //do nothing
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   count--;
}
```

Producer

```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    count++;
}
```

critical section:

Segments of code (in diff threads or processes) sharing access to shared objects (e.g., files, variables, arrays etc)

There's more than 1 CS here! How?

General structure for CS solutions

```
while (true) {
        entry section e.g, entryCS()

        critical section

        exit section e.g., leaveCS()

        remainder section
}
```

Code for CS entry request to lock the shared object

Code for CS exit request so that someone can now enter CS to lock the shared object

Solution criteria

- Criteria
  - Mutual exclusion
  - Progress
  - Bounded Waiting
- Assumptions
  - Speed independence
  - Progress assumed in CS

Mechanisms for CS solution

- No enhanced support
  - Peterson's algorithm
- Hardware mechanisms
  - TestAndSet and Swap
- Operating system support
  - Semaphores

No Enhanced Support

SW Solution 1: Peterson's algorithm

```
int turn;
int interested[2];
void EnterCS(int proc)
{
  int other;
  other = proc ^ 0x1; //toggle
  interested[proc] = true;
  turn = proc;
  while((turn==proc) && interested[other]);
}

void LeaveCS(int proc)
{
  interested[proc]= false ;
}
```

Peterson's algorithm

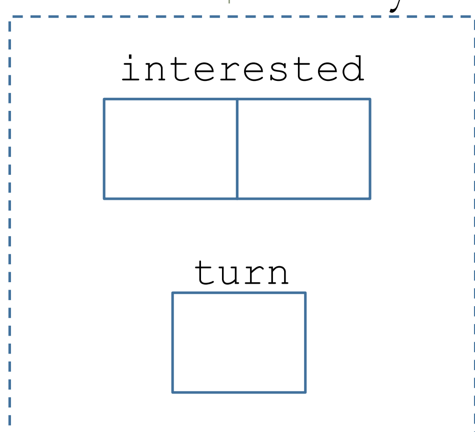
P_0 `proc=0`

```
void EnterCS(…)
{
  int other;
  other = 1; //toggle
  interested[0] = true;
  turn = 0;
  while((turn==0) &&
          interested[1]);
}

void LeaveCS(…)
{
  interested[0]= false;
}
```

P_1 `proc=1`

```
void EnterCS(…)
{
  int other;
  other = 0; //toggle
  interested[1] = true;
  turn = 1;
  while((turn==1) &&
          interested[0]);
}

void LeaveCS(…)
{
  interested[1]= false ;
}
```

shared memory



Can both go through?

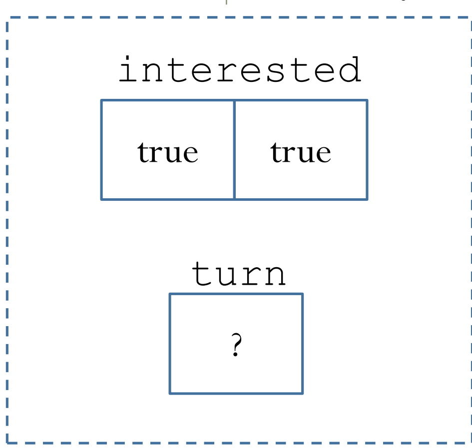
P_0 proc=0

```
void EnterCS(…)
{
  int other;
  other = 1;
  interested[0] = true;
  turn = 0;
  while((turn==0) &&
          interested[1]);
}

void LeaveCS(…)
{
   interested[0]= false ;
}
```

P_1 proc=1

```
void EnterCS(…)
{
  int other;
  other = 0; //toggle
  interested[1] = true;
  turn = 1;
  while((turn==1) &&
         interested[0]);
}

void LeaveCS(…)
{
   interested[1]= false ;
}
```

shared memory



Memory order

P_0 proc=0

Expected ordering

```
store 1, interested[0];
store 0, turn;
load r0, interested[1];
Load r1, turn;
```

Hardware messes it up during run-time!

```
load r0, interested[1];
store 1, interested[0];
store 0, turn;
load r1, turn;
```

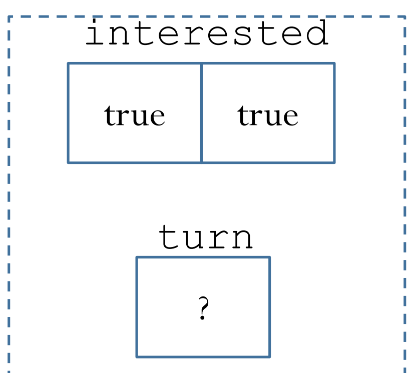
P_1 proc=1

Expected ordering

```
store 1, interested[1];
store 1, turn;
load r0, interested[0];
Load r1, turn;
```

```
load r0, interested[0];
store 1, interested[1];
store 0, turn;
load r1, turn;
```

shared memory



Peterson's algorithm (Modern version)

```
int turn;
Int interested[2];
void EnterCS(int proc)
{
  int other;
  other = proc ^ 0x1; //toggle
  interested[proc] = true;
  turn = proc;
  __asm__ ("mfence");
  while((turn==proc) && interested[other]);
}

void LeaveCS(int proc)
{
  interested[proc]= false ;
}
```

Memory barrier:
No memory reordering before this instruction

H/W Solutions

HW Solution 1: Disable Interrupts

```
while(true)
{
   Disable interrupts;
   //Critical Section
   …
   Enable interrupts;
   //Remainder Section

}
```

- Requirements
  - Computer Instruction
  - Usually kernel-mode
- Caveat
  - Uniprocessor only
  - Scalability issues

HW Solution 2: Test And Set

```
bool TestAndSet(bool *tgt)
{
  bool rv = *tgt;
  *tgt = true;
  return rv;
}
```

```
do
{
   while(TestAndSet(&lock));

   // critical section

   lock = false;

   // remainder section
} while (true);
```

HW Solution 3: Compare and Swap

```
int CompareAndSwap(
  int *value,
  int expected
  int new_value)
{
  int temp = *value;


  if(*value == expected)
      *value = new_value;


  return temp;
}
```

```
do {
  while(CompareAndSwap(&lock,
  0, 1) !=0);
  // critical section

  lock = 0;

  // remainder section
} while (true);
```

OS Solution

Semaphores – initial definition

```
class Semaphore
{
  void wait ()
  {
      while(s <=0) ;
      s--;
  }
  void signal()
  {
      s++;
  }
  private:
      int s;
}
```

```
class Semaphore
{
  void wait ()
  {
      if(s <= 0)
         block process
      s--;
  }
  void signal()
  {
      s++;
      wake up one of blocking
      process
  }
  private:
      int s;
}
```

Types of semaphore

- Counting/General semaphores
  - value of semaphore initialized to N
- Binary semaphore
  - value semaphore initialized to 1

Using semaphores

```
Semaphore s(1);

while(true)
{
   s.wait();
   //critical section
   s.signal();
   //remainder section;
}
```

Only 1 in CS each time

```
Semaphore s(N);

while(true)
{
   s.wait();
   //critical section
   s.signal();
   //remainder section;
}
```

Up to N in CS each time

Producer-Consumer Problem Revisited

Consumer

```
while(true)
{
   while(count == 0)
        ; //do nothing
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   count--;
}
```

Producer

```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    count++;
}
```

Producer-Consumer Problem Revisited

Consumer

```
while(true)
{
   while(count == 0)
        ; //do nothing
   mutex.wait();
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   mutex.signal();
   count--;

}
```

Producer

```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    mutex.wait();
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    mutex.signal();
    count++;
}
```

Not in the same process!

Process 1

```
while(true)
{
    S1;
    synch.signal();
}
```

Process 2

```
while(true)
{
    synch.wait();
    S2;
}
```

Shared memory

```
Semaphore synch;
```

The consumer should block when empty!

Consumer

```
while(true)
{
   while(count == 0)
        ; //do nothing
   mutex.wait();
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   mutex.signal();
   count--;
}
```

Producer

```
while(true)
{
    while(count == BUF_SIZE)
        ; //do nothing
    mutex.wait();
    buffer[in] = new product;
    in = (in + 1)%BUF_SIZE;
    mutex.signal();
    count++;
}
```

The consumer should block when empty!

Consumer

should block when buffer is empty!

```
while(true)
{
   empty.wait();
   mutex.wait();
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   mutex.signal();
   count--;
}
```

Producer

```
while(true)
{
     while(count == BUF_SIZE)
         ;
     mutex.wait();
     buffer[in] = new product;
     in = (in + 1)%BUF_SIZE;
     mutex.signal();
     empty.signal();
}
```

```
class Semaphore
{
  void wait ()
  {
      while(s <=0) ;
      s--;
  }
  void signal()
  {
      s++;
  }
  private:
      int s;
}
```

increment available buffer count

Semaphore is binary or counting (i.e. the value of **S**) semaphore based on the **count** of buffer size (1 or more)

Producer should block when buffer is full!

Consumer

```
while(true)
{
   empty.wait();
   mutex.wait();
   consumed = buffer[out];
   out = (out + 1) %BUF_SIZE;
   mutex.signal();
   full.signal();
}
```

increment available buffer count

Producer

should block when buffer is full!

```
while(true)
{
     full.wait();
     mutex.wait();
     buffer[in] = new product;
     in = (in + 1)%BUF_SIZE;
     mutex.signal();
     empty.signal();
}
```

```
class Semaphore
{
  void wait ()
  {
      while(s <=0) ;
      s--;
  }
  void signal()
  {
      s++;
  }
  private:
      int s;
}
```

Semaphore is binary or counting (i.e. the value of **S**) semaphore based on the **count** of buffer size (1 or more)