

Final Exam Review: Part 2

COMP400727: Introduction to Computer Systems

Hao Li

Xi'an Jiaotong University

Code Optimization

Improving Performance

```
void lower(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- Move call to `strlen` outside of loop
- Legal since result does not change from one iteration to another
- Form of code motion

Optimization Blocker: Procedure Calls

■ *Why couldn't compiler move strlen out of inner loop?*

- Procedure may have side effects
 - Alters global state each time called
- Function may not return same value for given arguments
 - Depends on other parts of global state
 - Procedure `lower` could interact with `strlen`

■ **Warning:**

- Compiler may treat procedure call as a black box
- Weak optimizations near them

■ Remedies:

- Use of inline functions
 - GCC does this with `-O1`
 - Within single file
- Do your own code motion

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```

Memory Matters

```

/* Sum rows of n X n matrix a
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}

```

```

# sum_rows1 inner loop
                                .L4:
                                movsd    (%rsi,%rax,8), %xmm0      # FP load
                                addsd     (%rdi), %xmm0            # FP add
                                movsd     %xmm0, (%rsi,%rax,8)      # FP store
                                addq      $8, %rdi
                                cmpq      %rcx, %rdi
                                jne       .L4

```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
{ 0, 1, 2,
  4, 8, 16},
{ 32, 64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

```
double A[9] =
{ 0, 1, 2,
  3, 22, 224},
{ 32, 64, 128};
```

Value of B:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

Benchmark Computation

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or
product of vector
elements

■ Data Types

- Use different declarations for data_t
- int
- long
- float
- double

■ Operations

- Use different definitions of OP and IDENT
- + / 0
- * / 1

Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

- Move `vec_length` out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

2x1

2x1a

2x2

Loop Unrolling (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
<i>Latency Bound</i>	<i>1.00</i>	<i>3.00</i>	<i>3.00</i>	<i>5.00</i>

■ Helps integer add

- Achieves latency bound

```
x = (x OP d[i]) OP d[i+1];
```

■ Others don't improve. *Why?*

- Sequential dependency

Loop Unrolling: Reduce Loop Overhead

Loop Unrolling with Reassociation (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

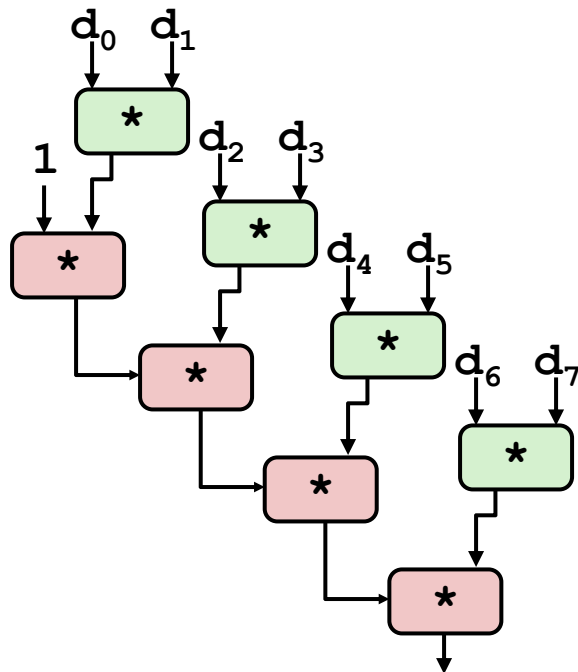
Compare to before

$x = (x \text{ OP } d[i]) \text{ OP } d[i+1];$

- Can this change the result of the computation?
- Yes, for FP. *Why?*

Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```



■ What changed:

- Ops in the next iteration can be started early (no dependency)

■ Overall Performance

- N elements, D cycles latency/op
- $(N/2+1)*D$ cycles:
 $CPE = D/2$

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
<i>Latency Bound</i>	<i>1.00</i>	<i>3.00</i>	<i>3.00</i>	<i>5.00</i>
<i>Throughput Bound</i>	<i>0.50</i>	<i>1.00</i>	<i>1.00</i>	<i>0.50</i>

Pipeline: Reduce N to 1

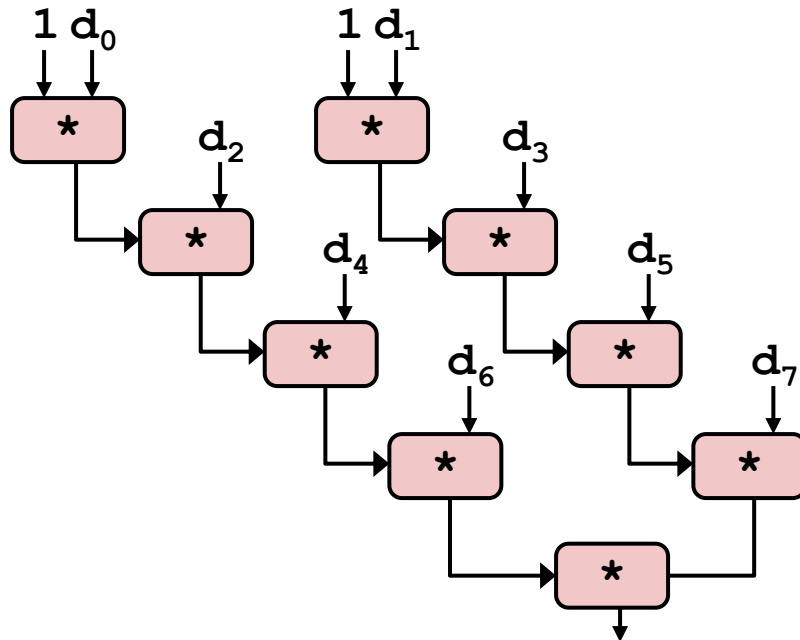
Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



■ What changed:

- Two independent “streams” of operations

■ Overall Performance

- N elements, D cycles latency/op
- Should be $(N/2+1)*D$ cycles:
CPE = D/2
- CPE matches prediction!

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
<i>Latency Bound</i>	<i>1.00</i>	<i>3.00</i>	<i>3.00</i>	<i>5.00</i>
<i>Throughput Bound</i>	<i>0.50</i>	<i>1.00</i>	<i>1.00</i>	<i>0.50</i>

- Int + makes use of two load units

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

- 2x speedup (over unroll2) for Int *, FP +, FP *

Multiple EU: Reduce N to N/M

2x1 : Reduce loop overhead

2x1a: Pipeline

2x2 : Pipeline + Multiple-EU

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
<i>Latency Bound</i>	<i>1.00</i>	<i>3.00</i>	<i>3.00</i>	<i>5.00</i>
<i>Throughput Bound</i>	<i>0.50</i>	<i>1.00</i>	<i>1.00</i>	<i>0.50</i>

4 func. units for int +,
2 func. units for load
Why Not .25?

1 func. unit for FP +
3-stage pipelined FP +

2 func. units for FP *,
2 func. units for load
5-stage pipelined FP *

Linking

Step 1: Symbol Resolution

```
int sum(int *a, int n);  
  
int array[2] = {1, 2};  
  
int main(int argc, char **argv)  
{  
    int val = sum(array, 2);  
    return val;  
} main.c
```

```
int sum(int *a, int n)  
{  
    int i, s = 0;  
  
    for (i = 0; i < n; i++) {  
        s += a[i];  
    }  
    return s;  
} sum.c
```


Step 1: Symbol Resolution

...that's defined here

Referencing
a global...

```
int sum(int *a, int n);  
  
int array[2] = {1, 2};  
  
int main(int argc, char **argv)  
{  
    int val = sum(array, 2);  
    return val;  
}  
main.c
```

Defining
a global

Linker knows
nothing of val

Referencing
a global...

...that's defined here

```
int sum(int *a, int n)  
{  
    int i, s = 0;  
    for (i = 0; i < n; i++) {  
        s += a[i];  
    }  
    return s;  
}  
sum.c
```

Linker knows
nothing of i or s

Symbol Identification

Which of the following names will be in the symbol table of `symbols.o`?

`symbols.c`:

```
int time;

int foo(int a) {
    int b = a + 1;
    return b;
}

int main(int argc,
          char* argv[]) {
    printf("%d\n", foo(5));
    return 0;
}
```

Names:

- `time`
- `foo`
- `a`
- `argc`
- `argv`
- `b`
- `main`
- `printf`
- Others?

Can find this with `readelf`:

```
linux> readelf -s symbols.o
```

Symbol Identification

Which of the following names will be in the symbol table of `symbols.o`?

`symbols.c`:

```
int time;

int foo(int a) {
    int b = a + 1;
    return b;
}

int main(int argc,
          char* argv[]) {
    printf("%d\n", foo(5));
    return 0;
}
```

Names:

- `time`
- `foo`
- `a`
- `argc`
- `argv`
- `b`
- `main`
- `printf`
 - `"%d\n"`

Can find this with `readelf`:

```
linux> readelf -s symbols.o
```

How Linker Resolves Duplicate Symbol Definitions

- Program symbols are either *strong* or *weak*
 - **Strong**: procedures and initialized globals
 - **Weak**: uninitialized globals
 - Or ones declared with specifier `extern`

p1.c

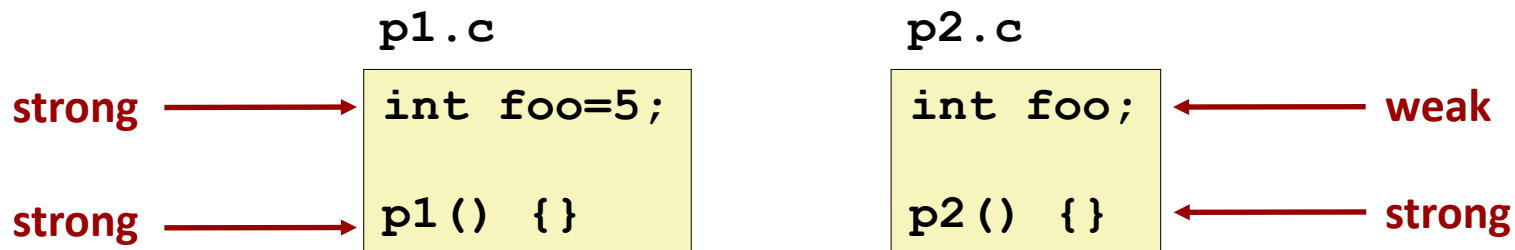
```
int foo=5;  
  
p1() {}
```

p2.c

```
int foo;  
  
p2() {}
```

How Linker Resolves Duplicate Symbol Definitions

- Program symbols are either *strong* or *weak*
 - **Strong**: procedures and initialized globals
 - **Weak**: uninitialized globals
 - Or ones declared with specifier `extern`



Linker's Symbol Rules

- **Rule 1: Multiple strong symbols are not allowed**
 - Each item can be defined only once
 - Otherwise: Linker error

- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
 - References to the weak symbol resolve to the strong symbol

- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
 - Can override this with `gcc -fno-common`

- **Puzzles on the next slide**

Linker Puzzles

```
int x;  
p1() {}
```

```
p1() {}
```

```
int x;  
p1() {}
```

```
int x;  
p2() {}
```

```
int x;  
int y;  
p1() {}
```

```
double x;  
p2() {}
```

```
int x=7;  
int y=5;  
p1() {}
```

```
double x;  
p2() {}
```

```
int x=7;  
p1() {}
```

```
int x;  
p2() {}
```

Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (p1)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` might overwrite `y`!
Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` might overwrite `y`!
Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same initialized variable.

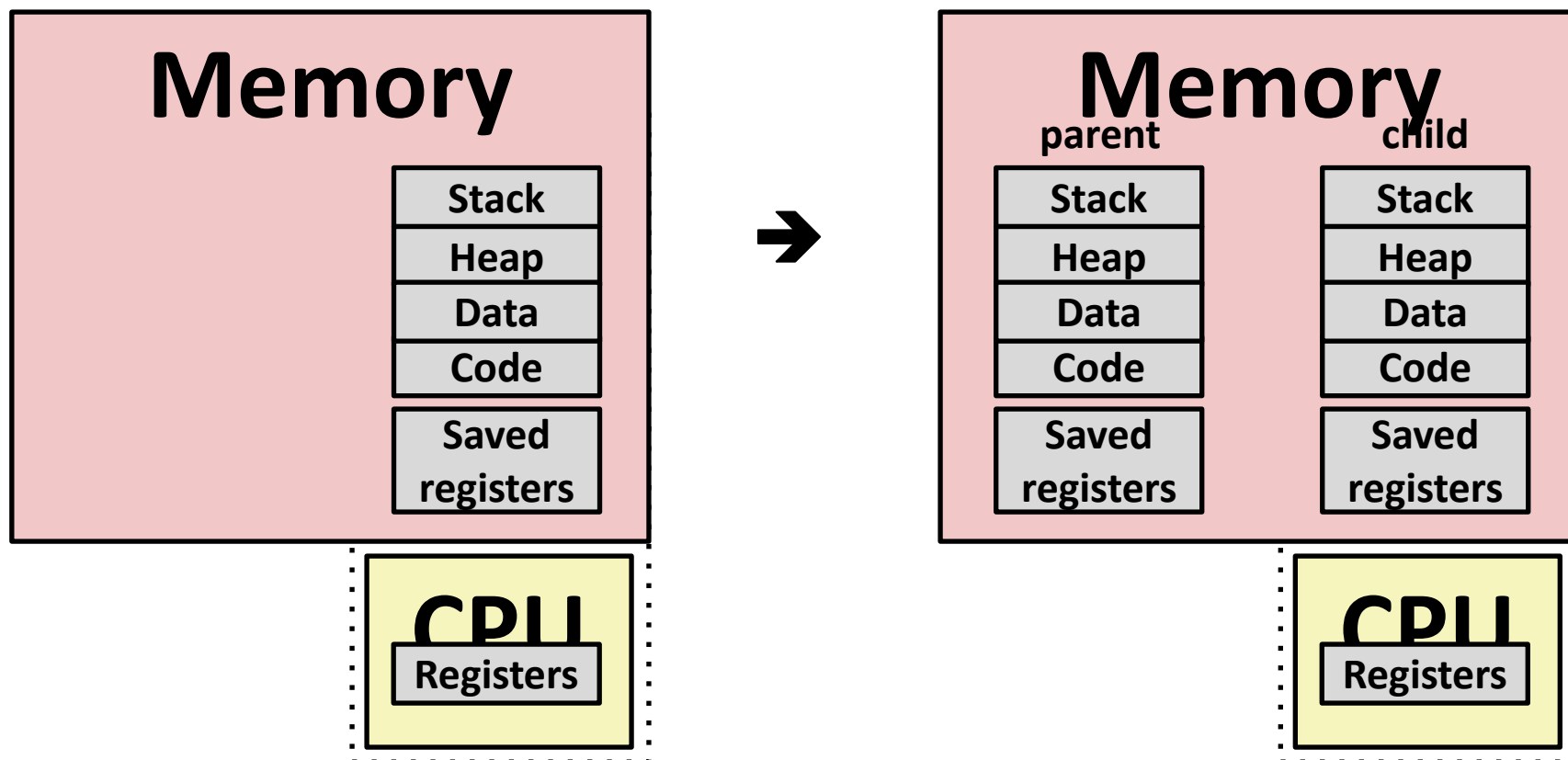
Important: Linker does not do type checking.

Processes and Multitasking

Creating Processes

- *Parent process* creates a new running *child process* by calling `fork`
- `int fork(void)`
 - Returns 0 to the child process, child's PID to parent process
 - Child is *almost* identical to parent:
 - Child get an identical (but separate) copy of the parent's virtual address space.
 - Child gets identical copies of the parent's open file descriptors
 - Child has a different PID than the parent
- `fork` is interesting (and often confusing) because it is called *once* but returns *twice*

Conceptual View of fork



■ Make complete copy of execution state

- Designate one as parent and one as child
- Resume execution of parent or child

forkx2 Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    printf("parent: x=%d\n", --x);
    return 0;
}
```

```
linux> ./fork2
parent: x=0
parent: x=-1
child : x=2
child : x=3
```

- Call once, return twice
- Concurrent execution
 - Can't predict execution order of parent and child
- Duplicate but separate address space
 - `x` has a value of 1 when fork returns in parent and child
 - Subsequent changes to `x` are independent
- Shared open files
 - `stdout` is the same in both parent and child

Modeling fork with Process Graphs

- A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:
 - Each vertex is the execution of a statement
 - $a \rightarrow b$ means a happens before b
 - Edges can be labeled with current value of variables
 - `printf` vertices can be labeled with output
 - Each graph begins with a vertex with no inedges
- Any *topological sort* of the graph corresponds to a feasible total ordering.
 - Total ordering of vertices where all edges point from left to right

Process Graph Example

```

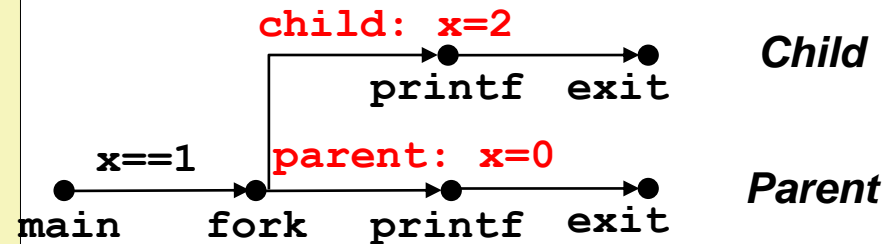
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {    /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}

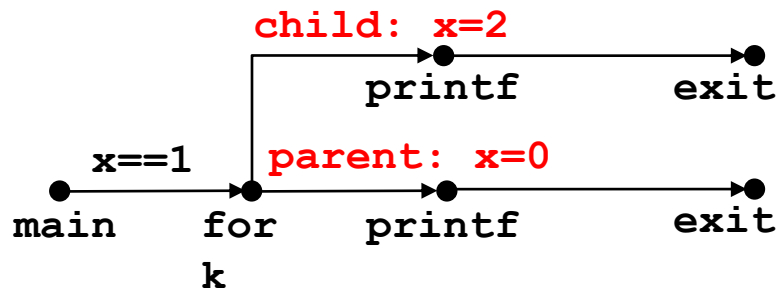
```

fork.c

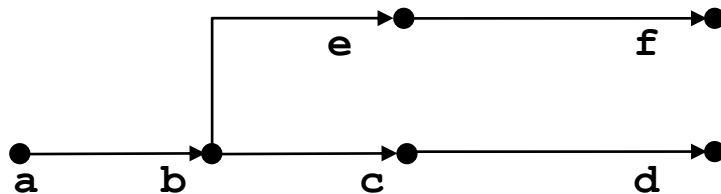


Interpreting Process Graphs

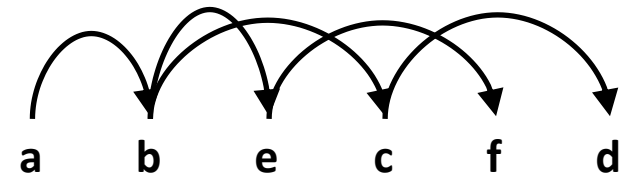
■ Original graph:



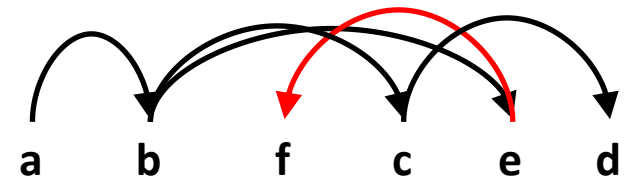
■ Relabelled graph:



Feasible total ordering:



Infeasible total ordering:

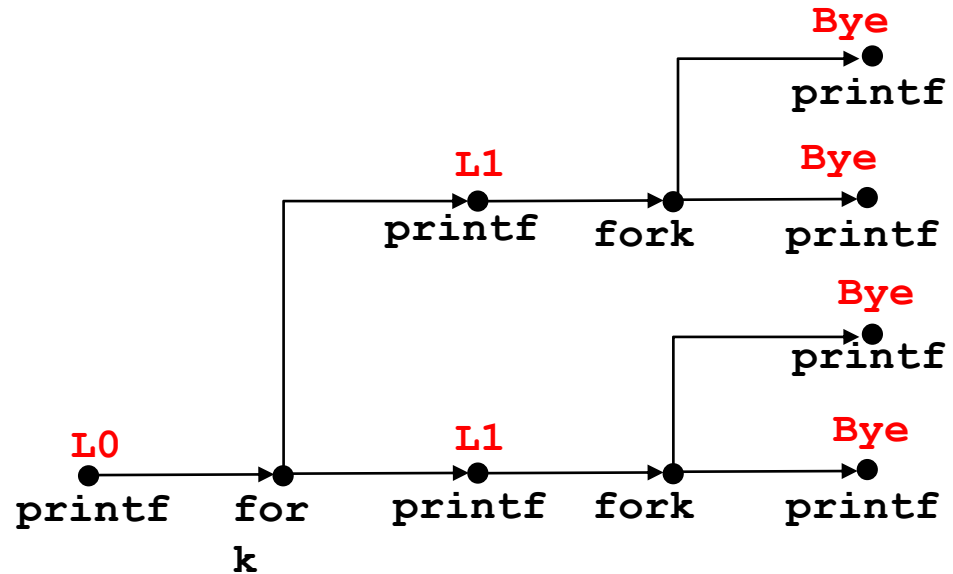


fork Example: Two consecutive forks

```

void fork2 ()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
forks.c

```



Feasible output:

L0
 L1
 Bye
 Bye
 L1
 Bye
 Bye

Infeasible output:

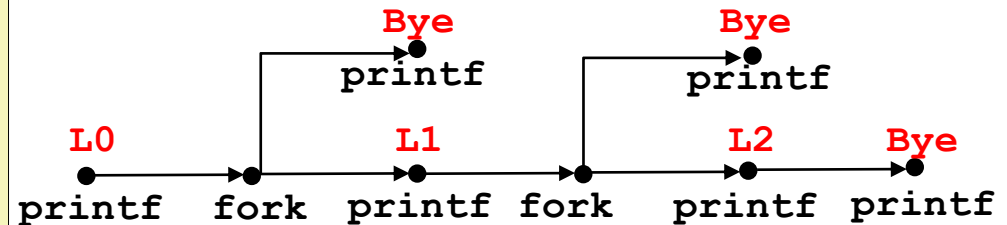
L0
 Bye
 L1
 Bye
 L1
 Bye
 Bye

fork Example: Nested forks in parent

```

void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
forks.c

```



Feasible output:

L0
 L1
 Bye
 Bye
 L2
 Bye

Infeasible output:

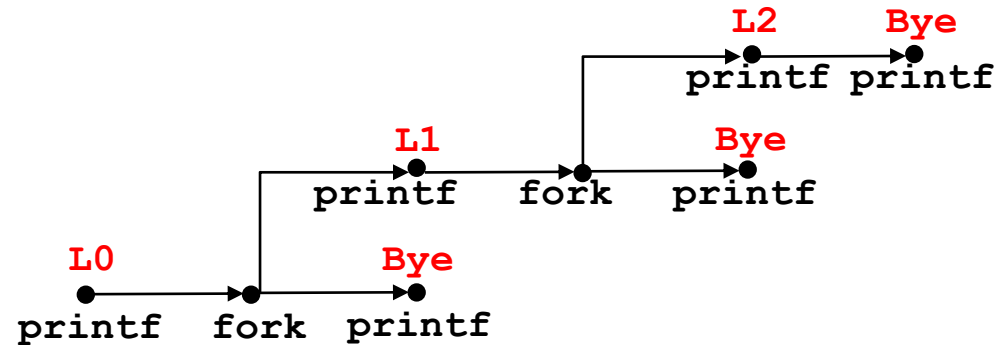
L0
 Bye
 L1
 Bye
 Bye
 L2

fork Example: Nested forks in children

```

void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
forks.c

```



Feasible output:

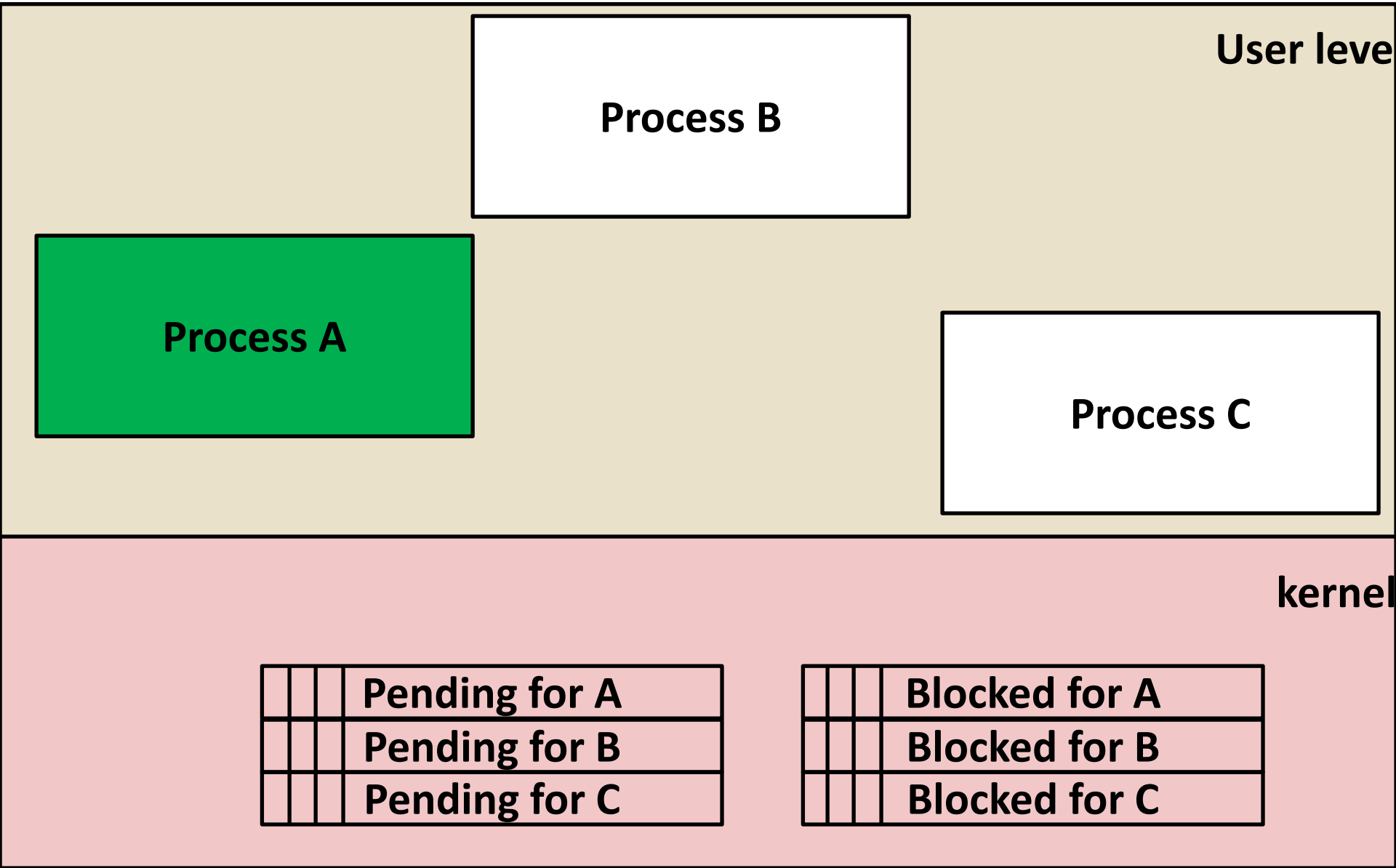
L0
 Bye
 L1
 L2
 Bye
 Bye

Infeasible output:

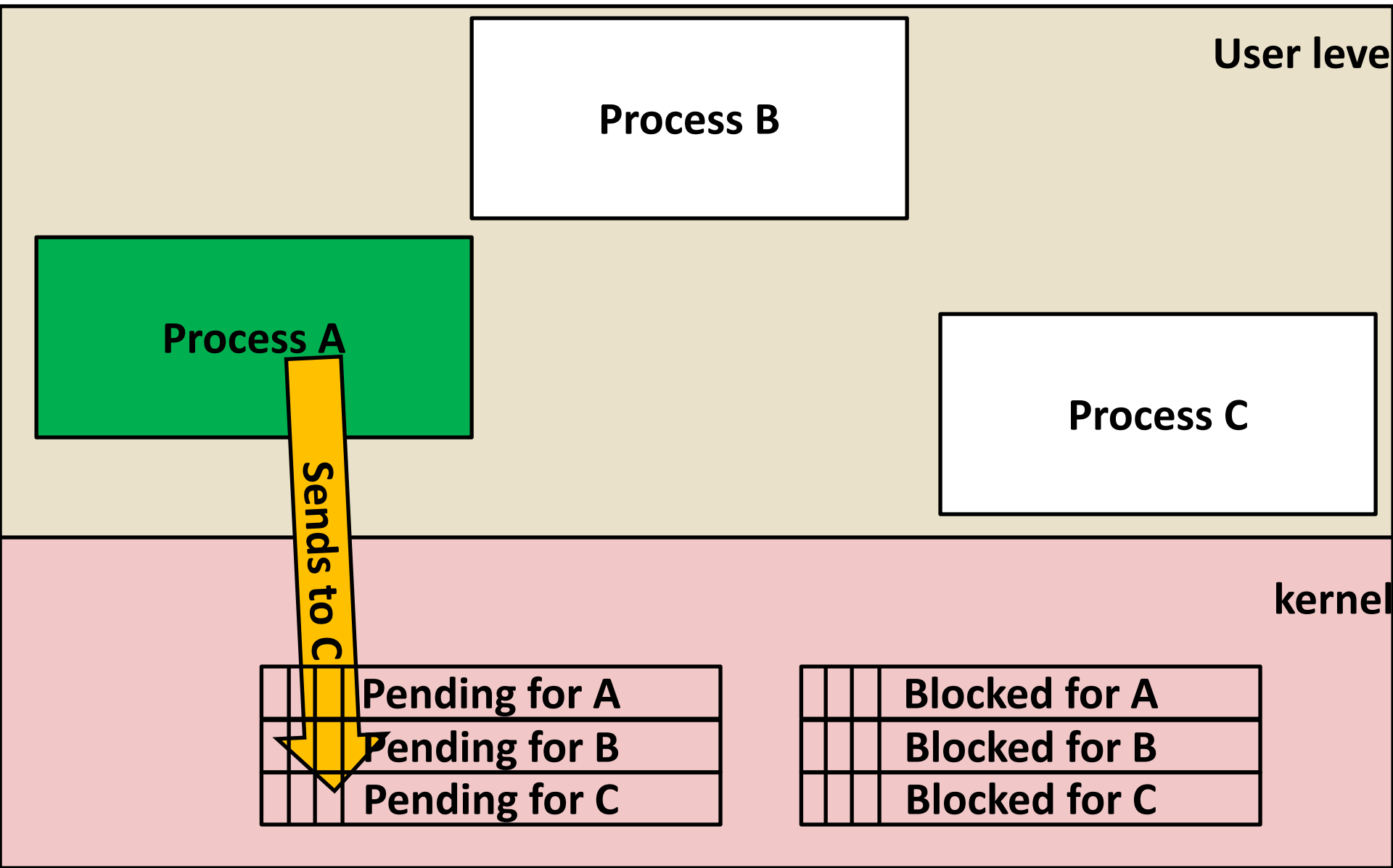
L0
 Bye
 L1
 Bye
 Bye
 L2

Exceptional Control Flow

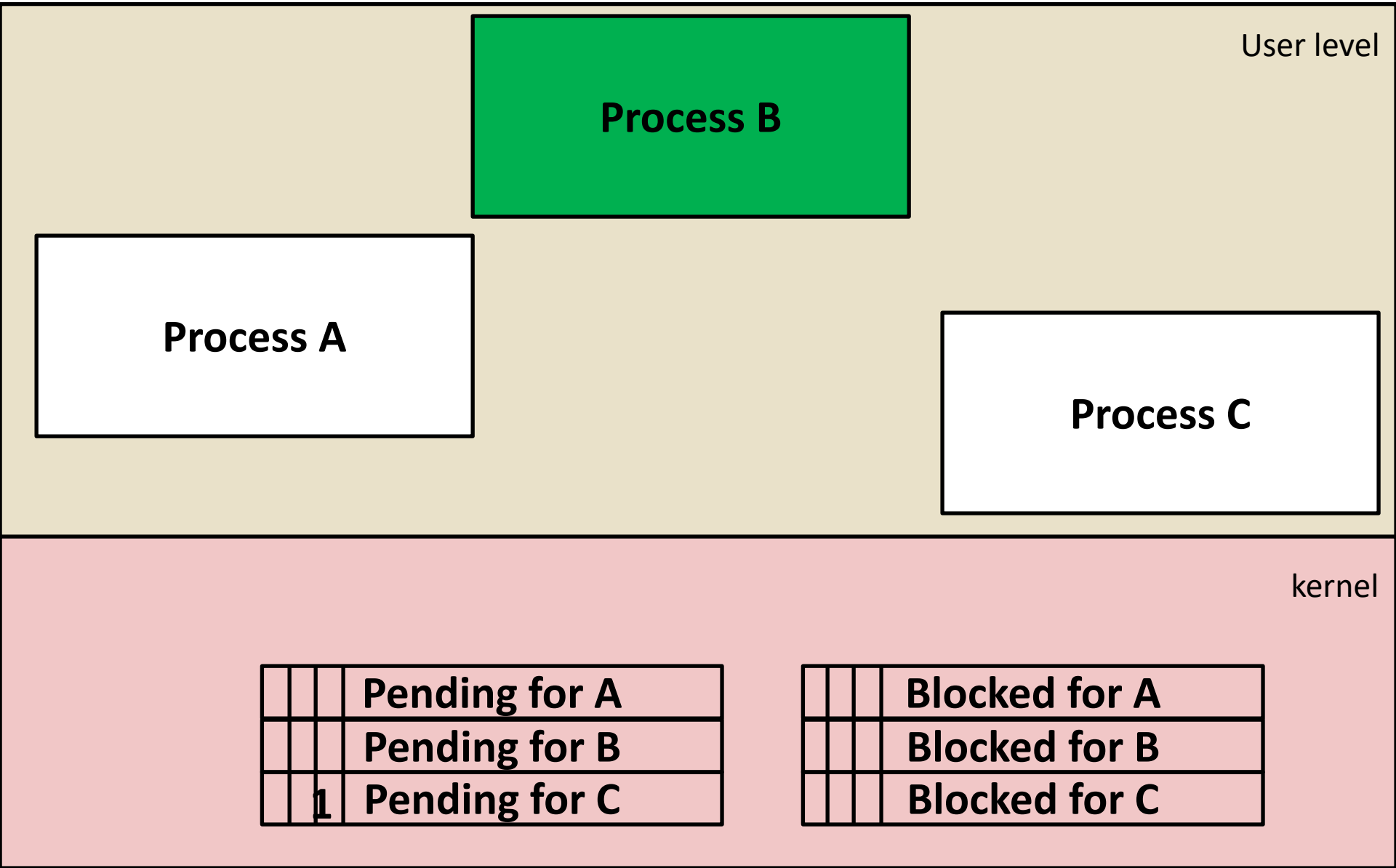
Signal Concepts: Sending a Signal



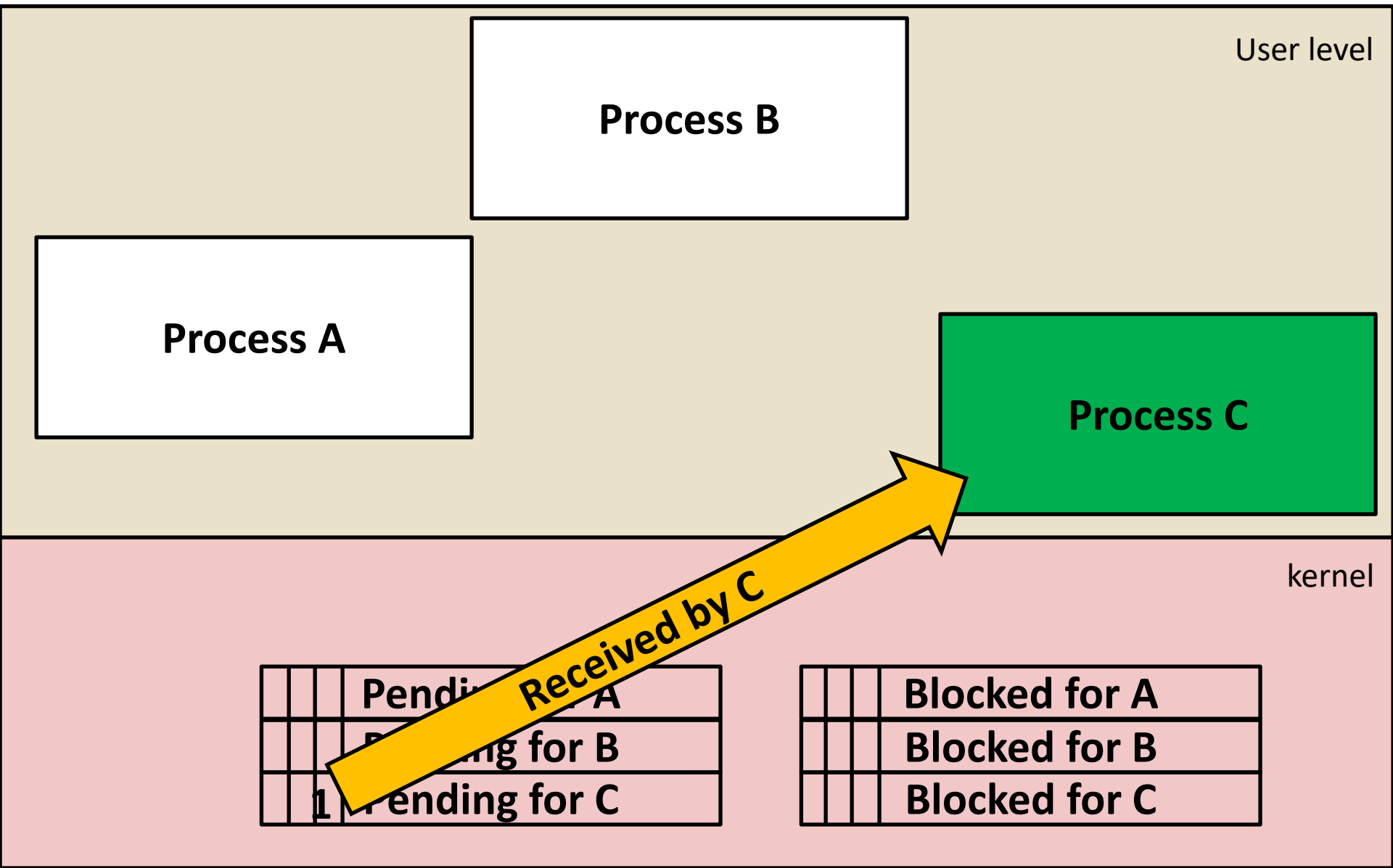
Signal Concepts: Sending a Signal



Signal Concepts: Sending a Signal



Signal Concepts: Sending a Signal



Signal Concepts: Sending a Signal

User level

Process B

Process A

Process C

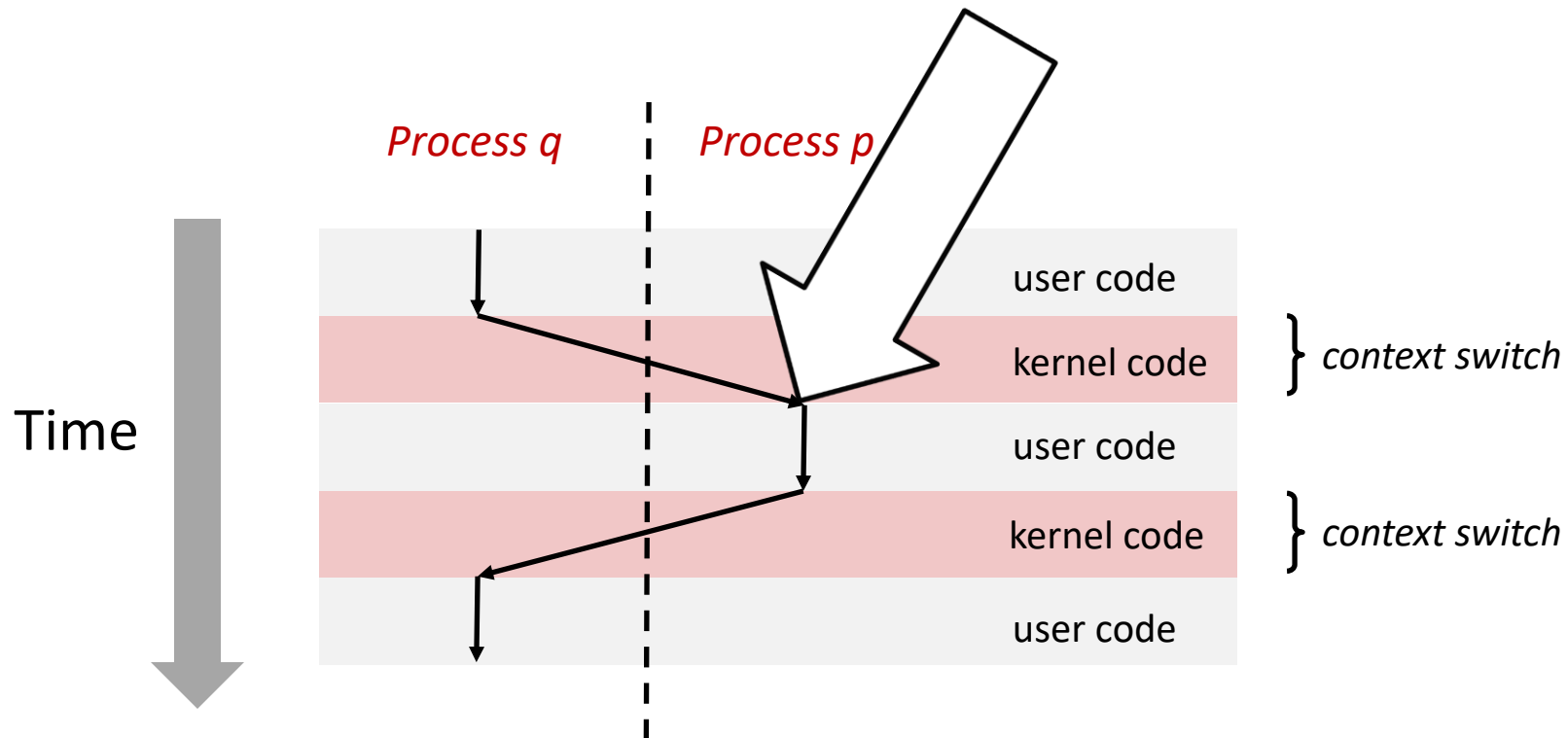
kernel

			Pending for A
			Pending for B
	0		Pending for C

			Blocked for A
			Blocked for B
			Blocked for C

Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process p



Correct Signal Handling

```
volatile int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}
```

This code is incorrect!

```
void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    Signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) {
            Sleep(1);
            exit(0); /* Child exits */
        }
    }
    while (ccount > 0) /* Parent spins */
        ;
}
```

`N == 5`

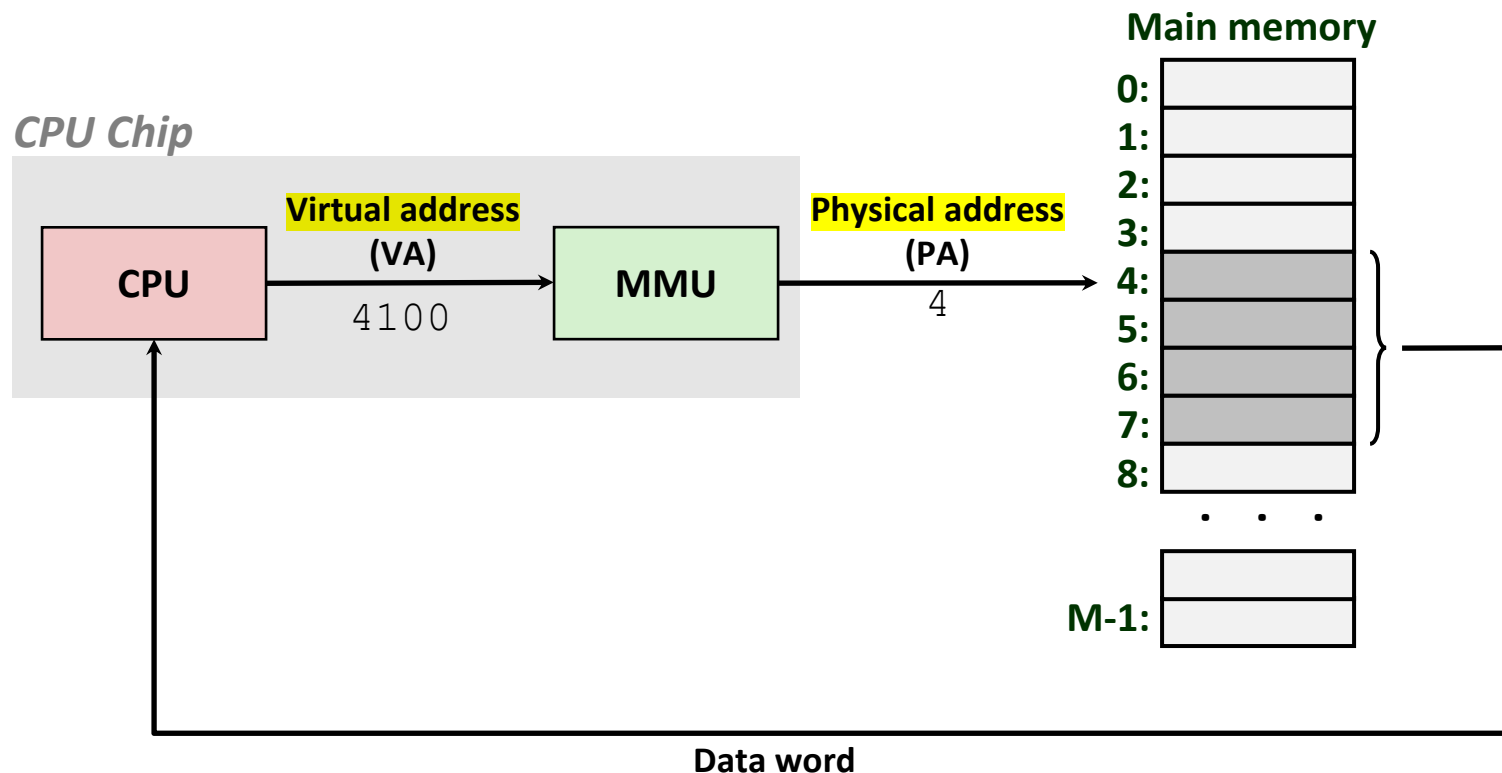
```
whaleshark> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
...(hangs)
```

forks.c

- Pending signals are not queued
 - For each signal type, one bit indicates whether or not signal is pending...
 - ...thus at most one pending signal of any particular type.
- You can't use signals to count events, such as children terminating.

Virtual Memory

A System Using Virtual Addressing

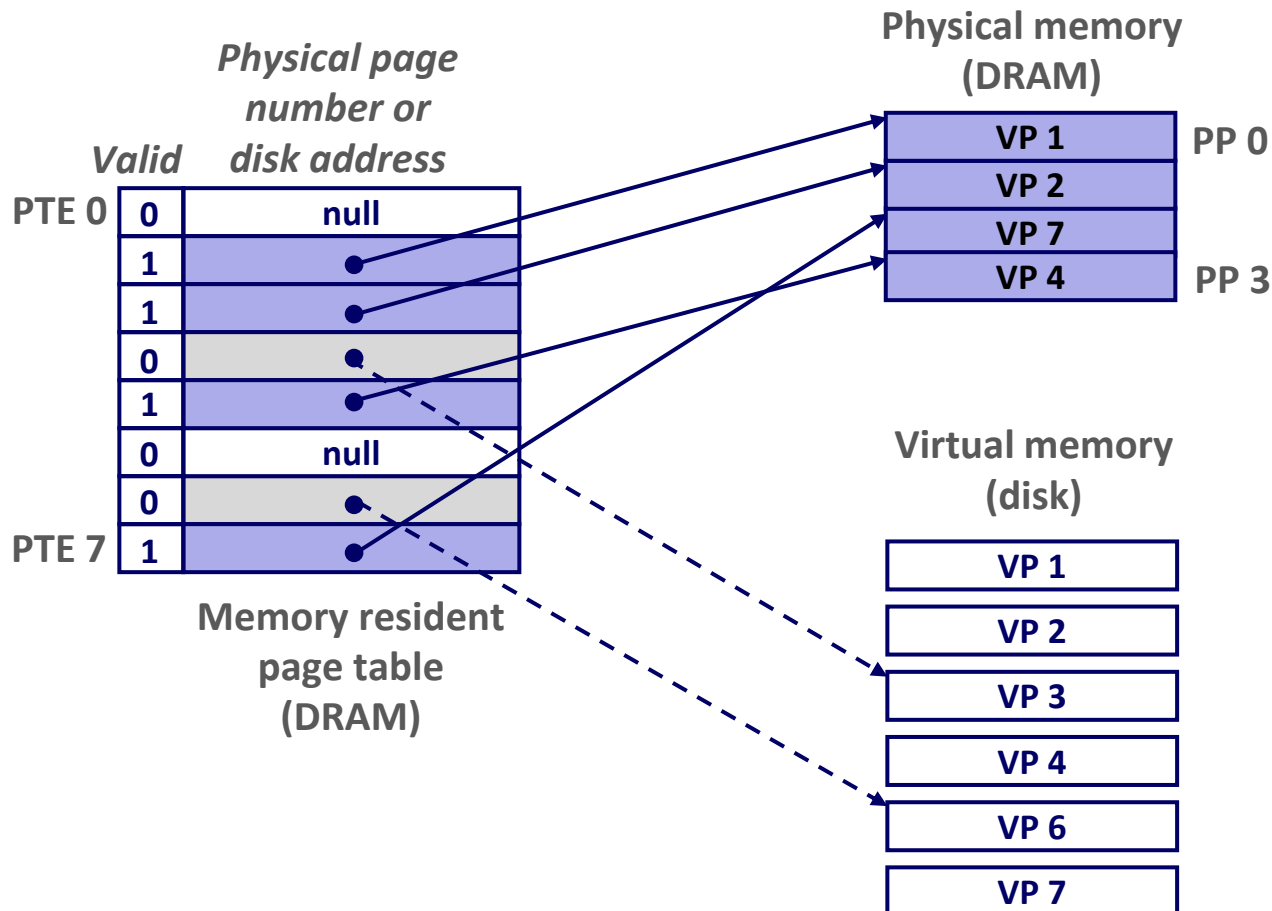


- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

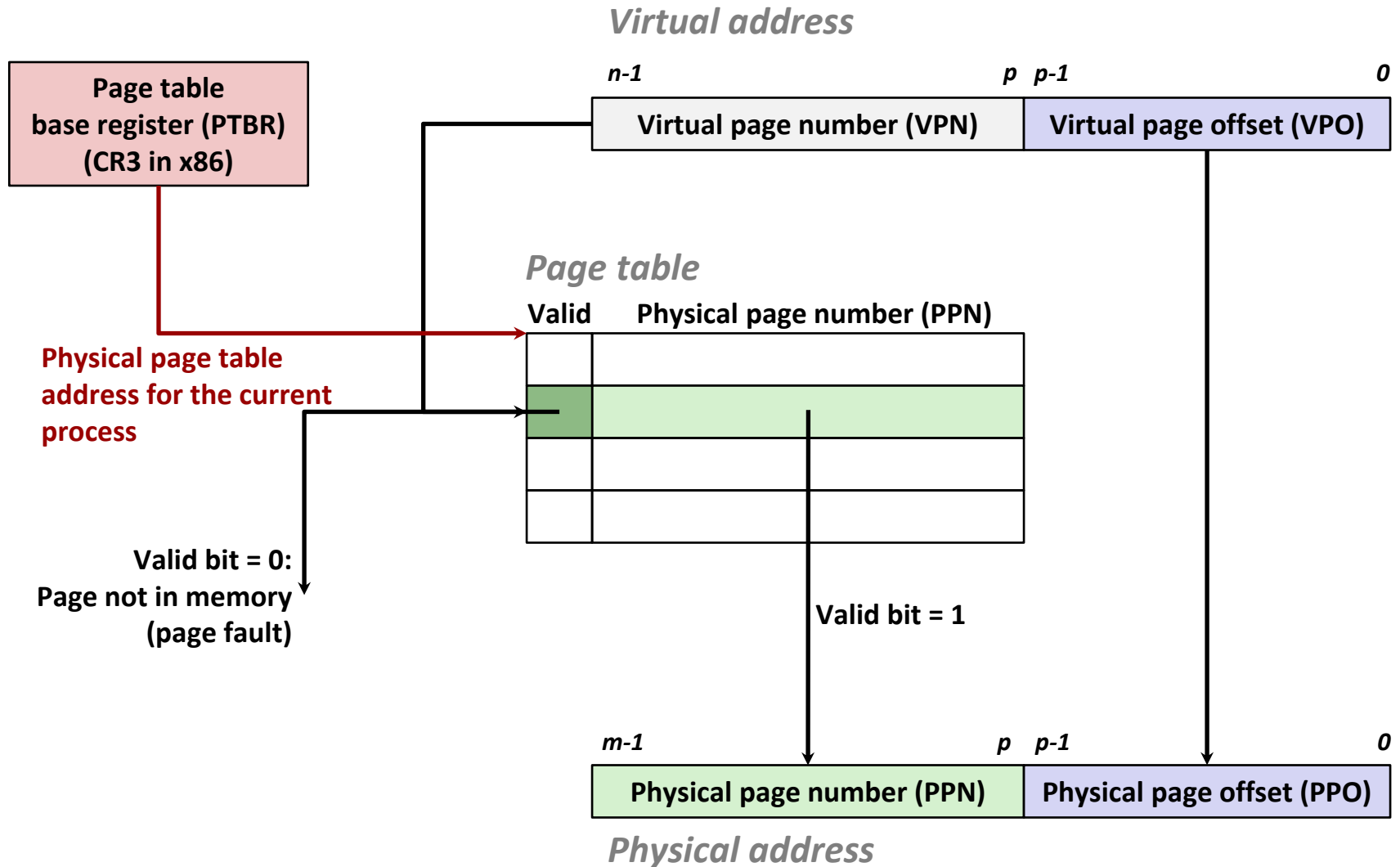
Enabling Data Structure: Page Table

■ A **page table** is an array of page table entries (PTEs) that maps virtual pages to physical pages.

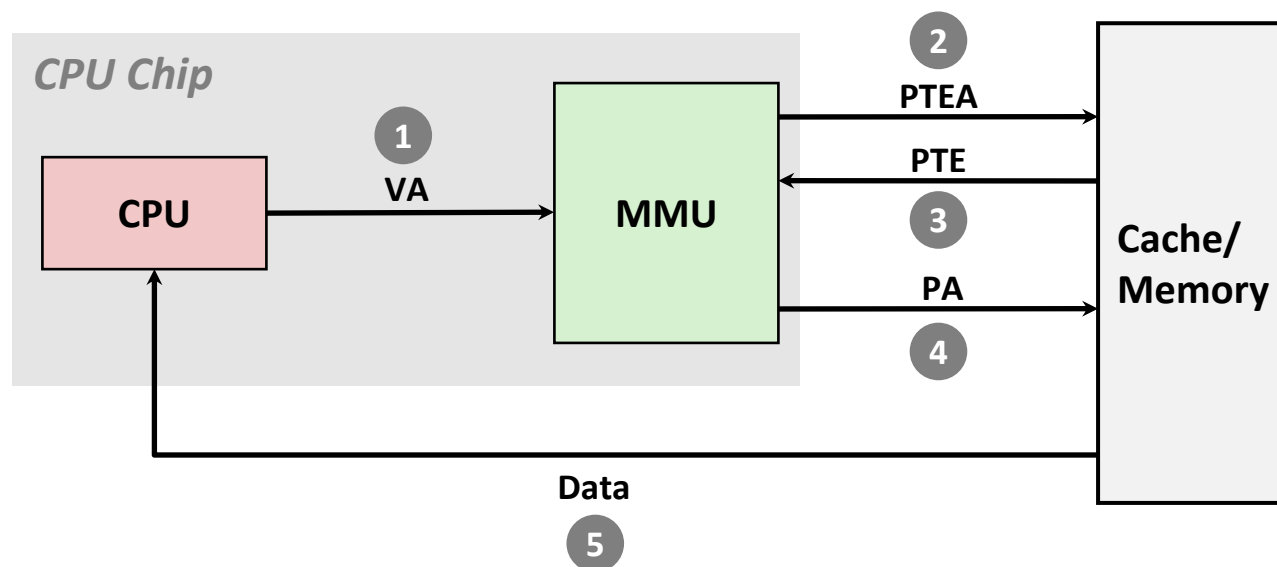
- Per-process kernel data structure in DRAM



Address Translation With a Page Table

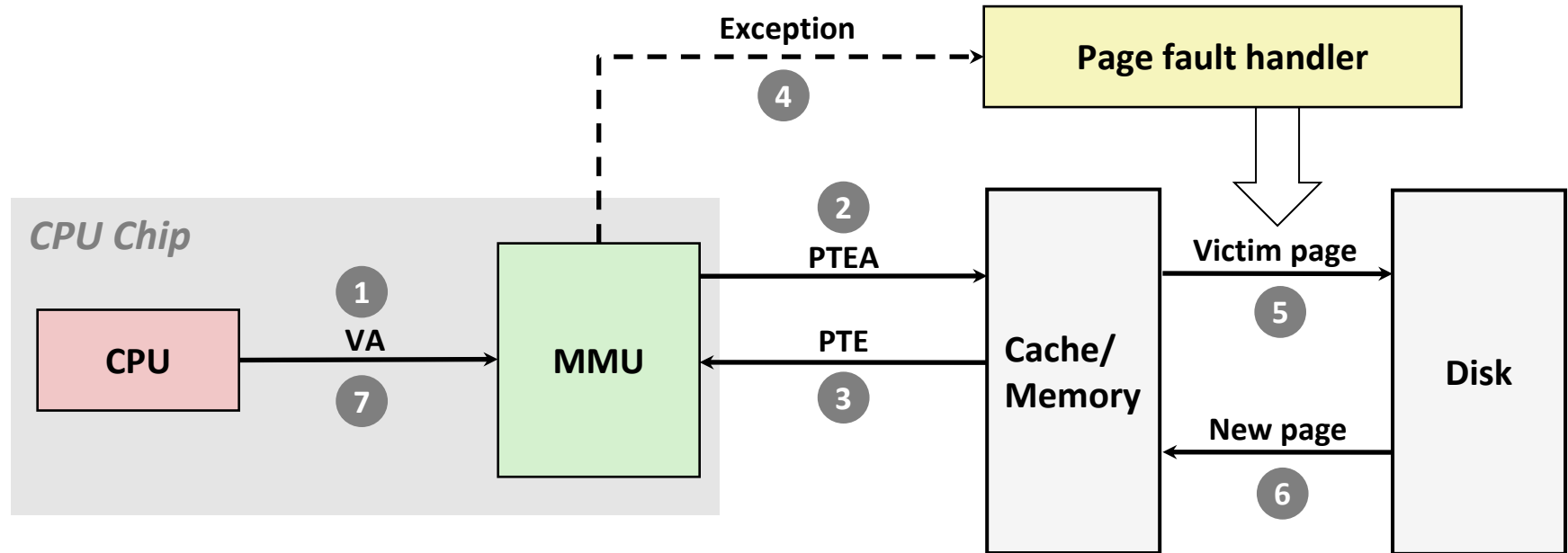


Address Translation: Page Hit



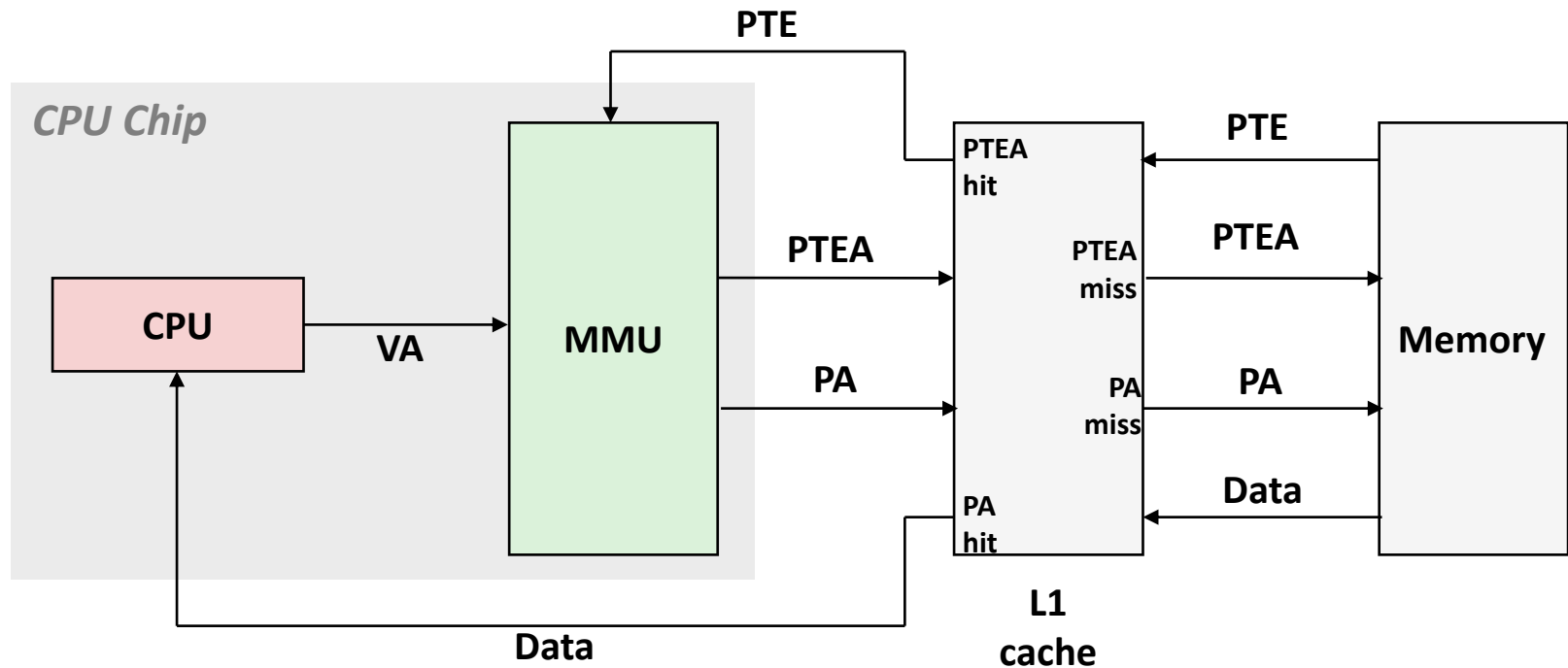
- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

Address Translation: Page Fault



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

Integrating VM and Cache



VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Speeding up Translation with a TLB

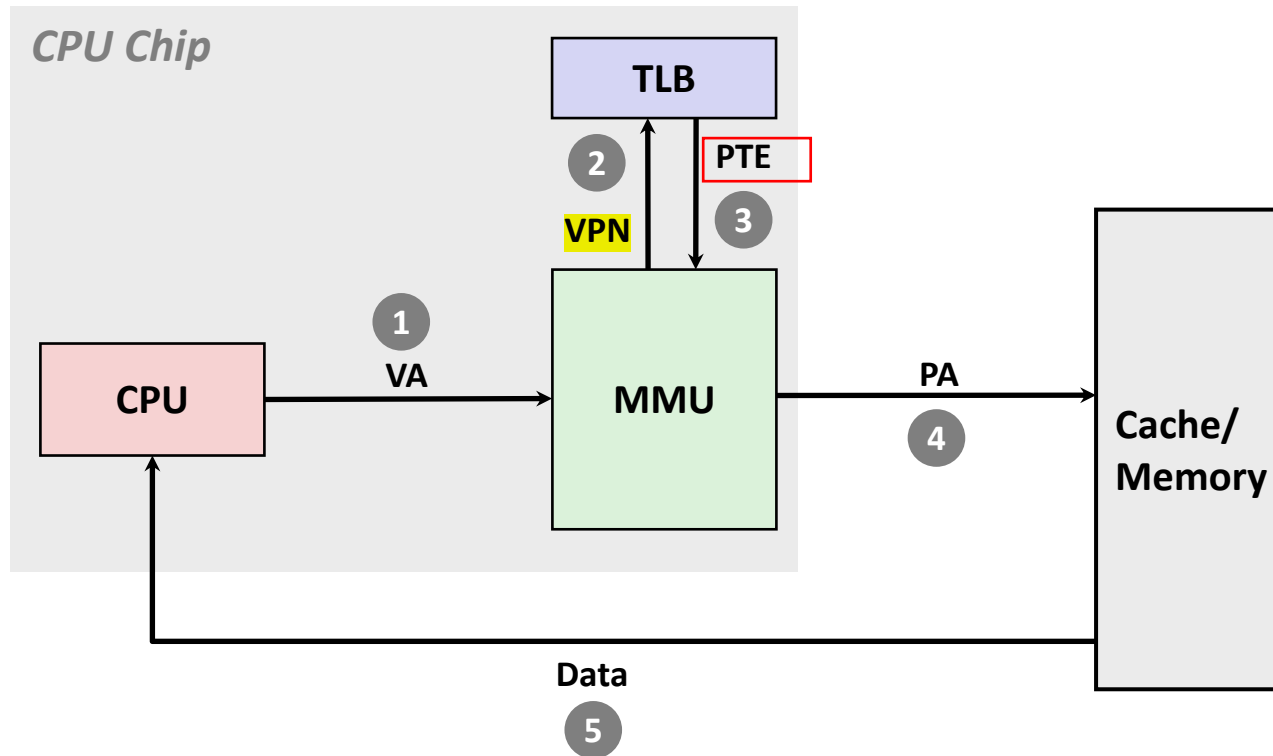
■ Page table entries (PTEs) are cached in L1 like any other memory word

- PTEs may be evicted by other data references
- PTE hit still requires a small L1 delay

■ Solution: *Translation Lookaside Buffer* (TLB)

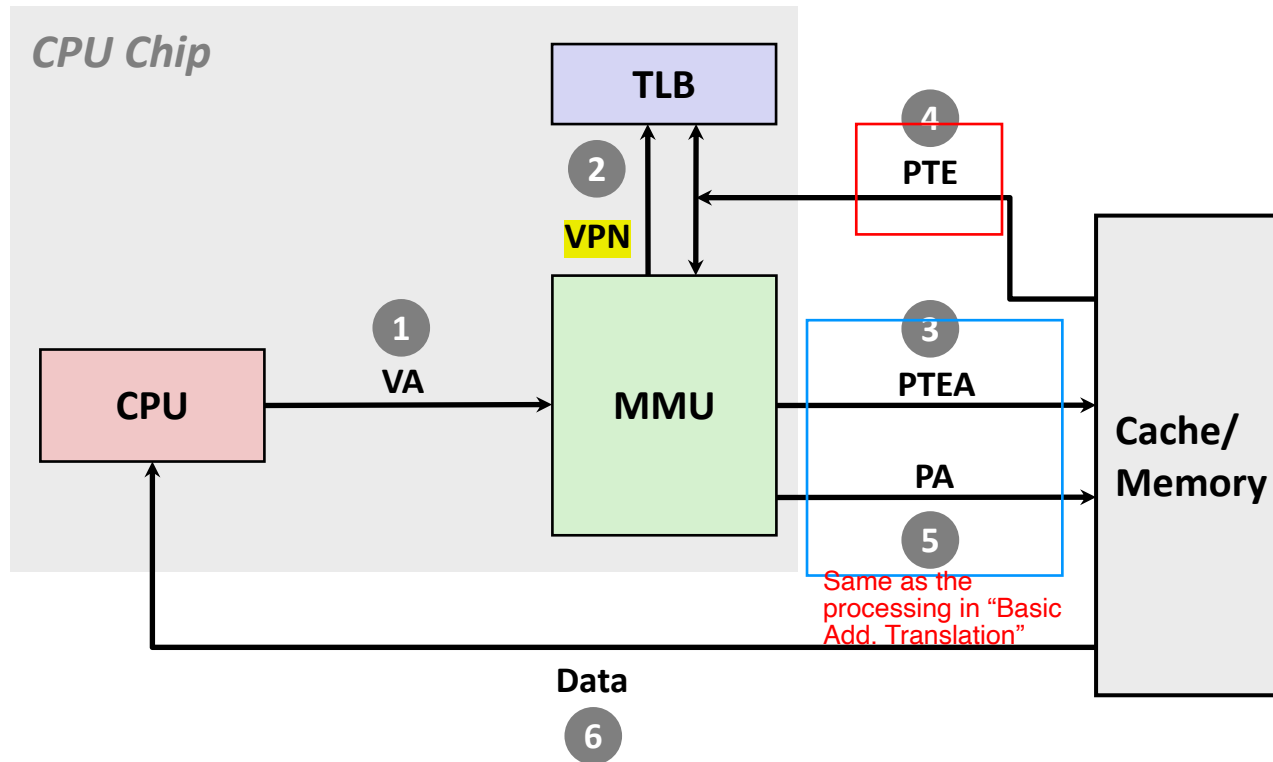
- Small set-associative hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages

TLB Hit



A TLB hit eliminates a memory access

TLB Miss



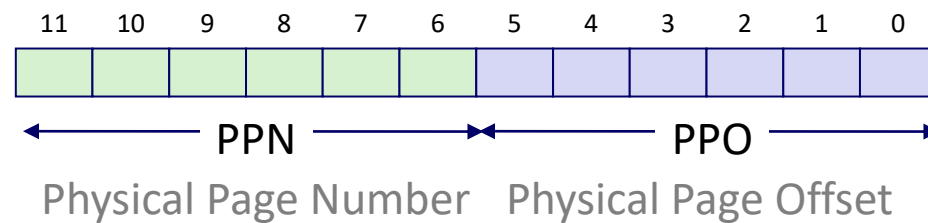
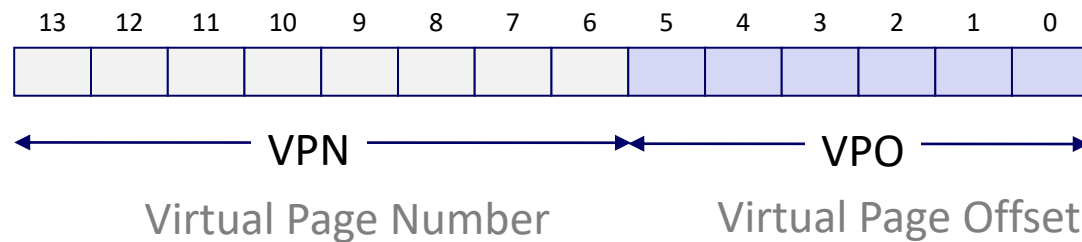
A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare. Why?

Simple Memory System Example

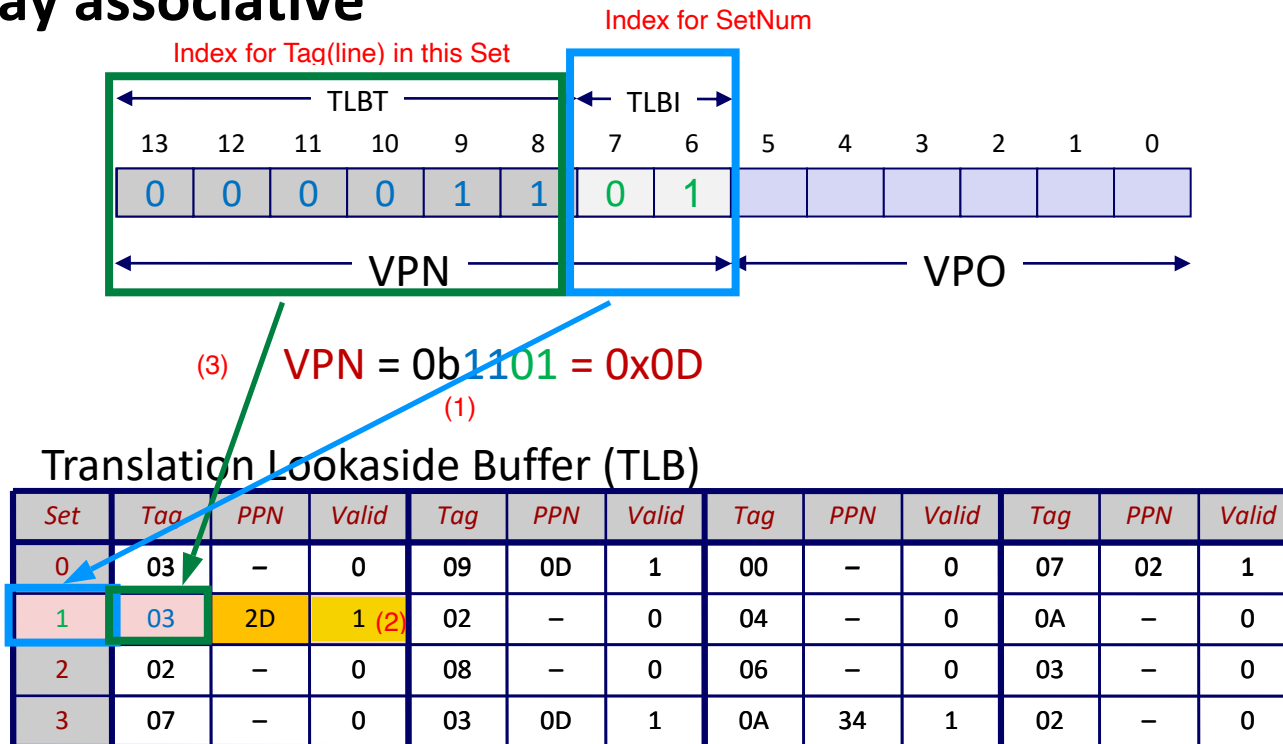
■ Addressing

- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bytes



Simple Memory System TLB

- 16 entries
- 4-way associative



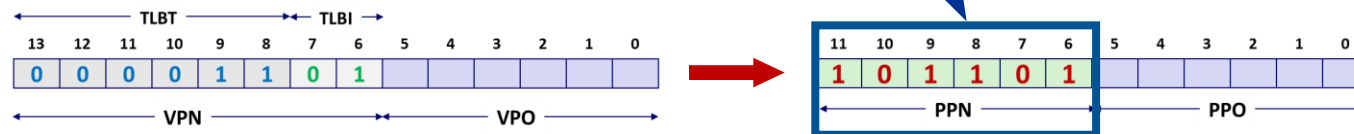
Simple Memory System Page Table

- Only showing the first 16 entries (out of 256)

VPN	PPN	Valid
00	28	1
01	—	0
02	33	1
03	02	1
04	—	0
05	16	1
06	—	0
07	—	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	—	0
0C	—	0
0D	2D	1
0E	11	1
0F	0D	1

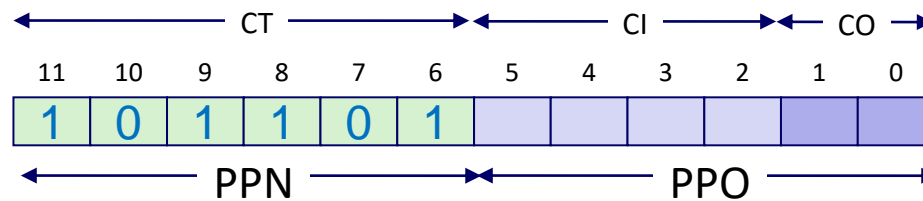
VPN PPN
0x0D → 0x2D



Simple Memory System Cache

- 16 lines, 4-byte cache line size
- Physically addressed
- Direct mapped**

直接相联 => 每一组里面E=1, 只有一行

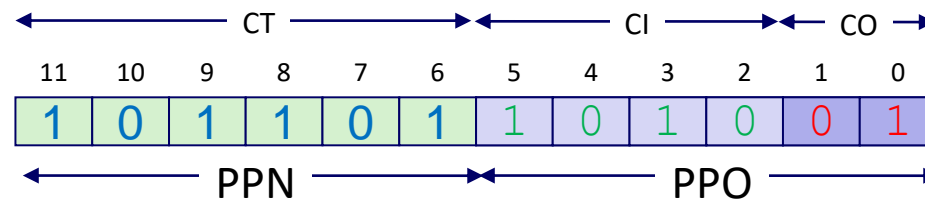


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	—	—	—	—
2	1B	1	00	02	04	08
3	36	0	—	—	—	—
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	—	—	—	—
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	—	—	—	—
A	2D	1	93	15	DA	3B
B	0B	0	—	—	—	—
C	12	0	—	—	—	—
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	—	—	—	—

Simple Memory System Cache

- 16 lines, 4-byte cache line size
- Physically addressed
- Direct mapped

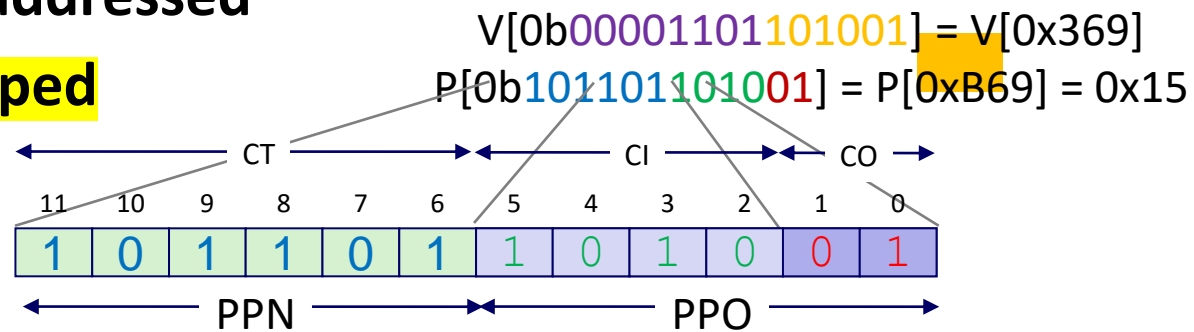


<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Simple Memory System Cache

- 16 lines, 4-byte cache line size
- Physically addressed
- Direct mapped**

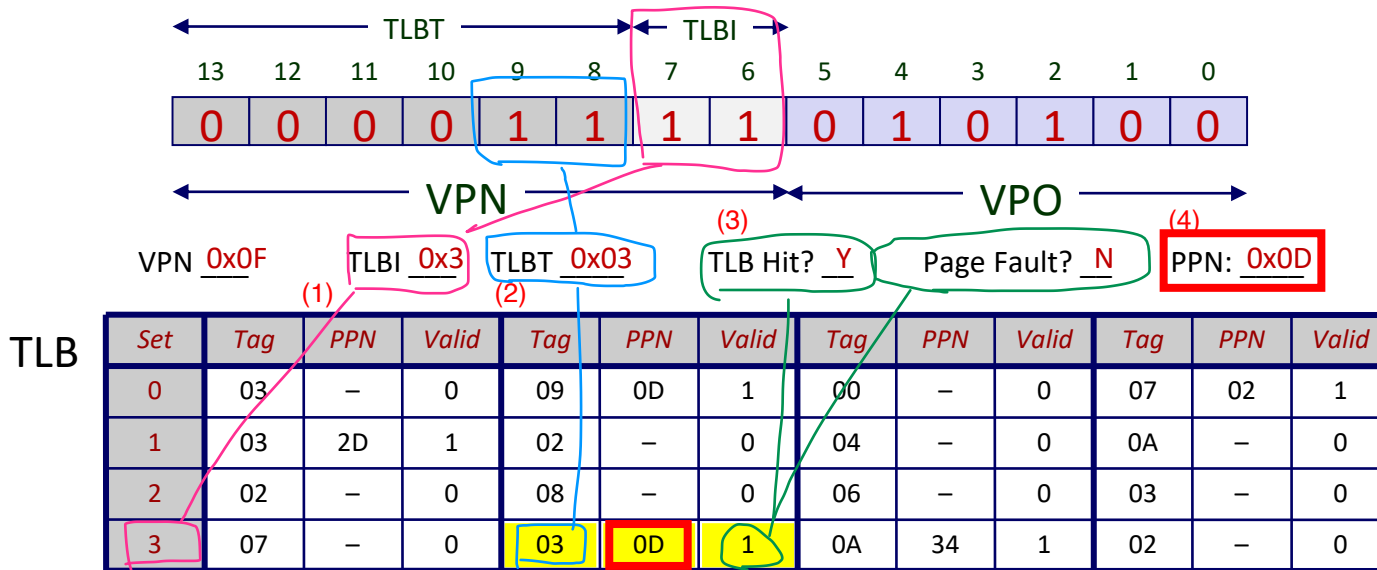


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	—	—	—	—
2	1B	1	00	02	04	08
3	36	0	—	—	—	—
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	—	—	—	—
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	—	—	—	—
A	2D	1	93	15	DA	3B
B	0B	0	—	—	—	—
C	12	0	—	—	—	—
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	—	—	—	—

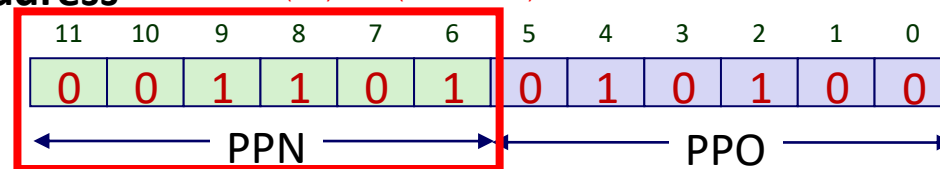
Address Translation Example

Virtual Address: 0x03D4



Physical Address

(0D)16 = (0000 1101)2



Thanks And Hope You Enjoy!