Matrix Multiplication Example

- Major Cache Effects to Consider
  - Total cache size
    - Exploit temporal locality and keep the working set small (e.g., by using blocking)
  - Block size
    - Exploit spatial locality
- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

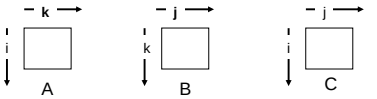
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable ***sum*** *held in register*

1

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Line size = 32B (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop



2

Layout of C Arrays in Memory (review)

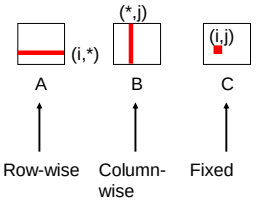
- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)

3

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:



- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

4

Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

16

Example: Matrix Multiplication

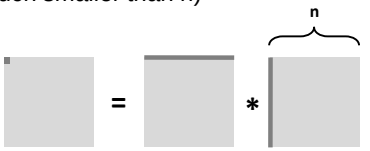
```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

17

Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
- First iteration:
  - n/8 + n = 9n/8 misses



  - Afterwards in cache: (schematic)

18

Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
- Second iteration:
  - Again: n/8 + n = 9n/8 misses

- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$

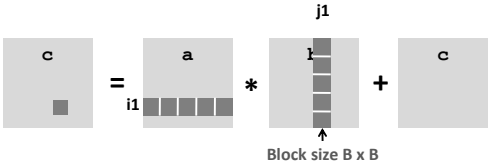
19

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

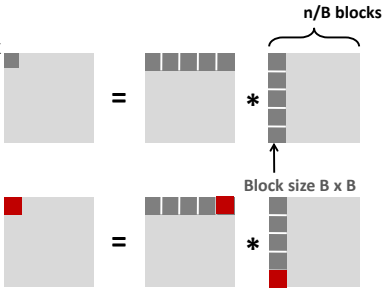
matmult/bmm.c



20

Cache Miss Analysis

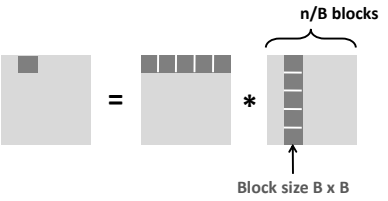
- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)



21

Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- Second (block) iteration:
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$
- Total misses:
  - $nB/4 * (n/B)^2 = n^3/(4B)$



22

Blocking Summary

- No blocking: $(9/8) * n^3$
- Blocking: $1/(4B) * n^3$
- Suggest largest possible block size B, but limit $3B^2 < C$!
- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array element used O(n) times!
  - Program has to be written properly

23

Concluding Observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)

25

Exceptions and Processes

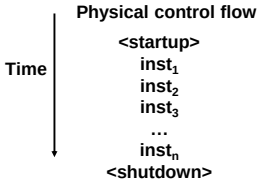
- Topics
  - Exceptions and modes
  - Processes
  - Signals

class12.ppt

26

Control Flow

- Computers do Only One Thing
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time.
  - This sequence is the system’s physical *control flow* (or *flow of control*).



27

Altering the Control Flow

- Up to Now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return using the stack discipline.
  - Both react to changes in program state.
- Insufficient for a useful system
  - Difficult for the CPU to react to changes in system state
    - data arrives from a disk or a network adapter
    - Instruction divides by zero
    - User hits ctl-c at the keyboard
    - System timer expires
- System needs mechanisms for “exceptional control flow”

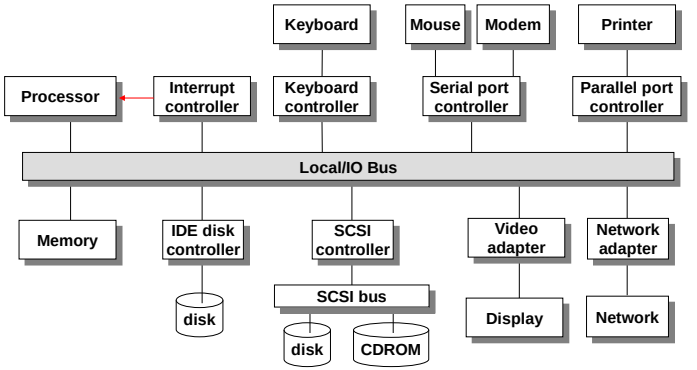
28

Exceptional Control Flow

- Mechanisms for exceptional control flow exists at all levels of a computer system
- Low-level mechanism
  - Exceptions and interrupts
    - change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- Higher-level mechanisms
  - Process context switch
  - Signals
  - Nonlocal jumps (setjmp/longjmp)
  - Implemented by either:
    - OS software (context switch and signals) with hardware support (e.g., timer)
    - C language runtime library: nonlocal jumps
    - Language level (Java and C++): try/throw/catch

29

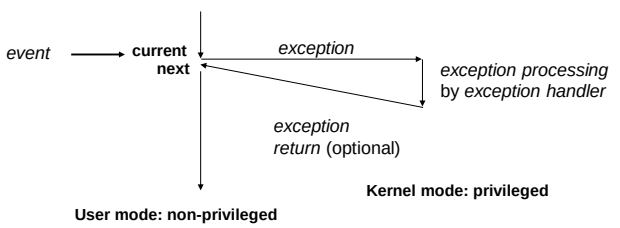
System context for exceptions



30

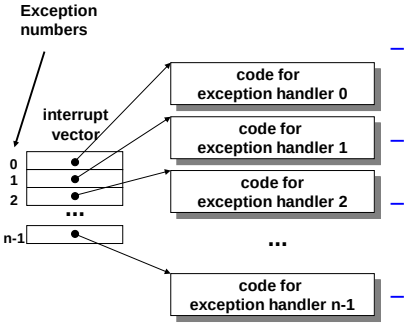
Exceptions

- An *exception* is a transfer of control to the OS in response to some *event* (i.e., change in processor state)



31

Interrupt Vectors



- Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry k points to a function (exception handler).
- Handler k is called each time exception k occurs.

32

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - handler returns to "next" instruction.
- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupts
    - hitting ctl-c at the keyboard
    - arrival of a packet from a network
    - arrival of a data sector from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting ctl-alt-delete on a PC

33

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - Traps
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - Faults
    - Unintentional but possibly recoverable
    - Examples: page faults, protection faults, floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts.
  - Aborts
    - unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program or halts machine

34

Exceptions

  - Conditions under which pipeline cannot continue normal operation
- Causes
  - Halt instruction (Current)
  - Bad address for instruction or data (Previous)
  - Invalid instruction (Previous)
- Desired Action
  - Complete some instructions
    - Either current or previous (depends on exception type)
  - Discard others
  - Call exception handler
    - Like an unexpected procedure call

35

Exception Examples

- Detect in Fetch Stage

```
jmp $-1                    # Invalid jump target

.byte 0xFF                 # Invalid instruction code

halt                       # Halt instruction
```

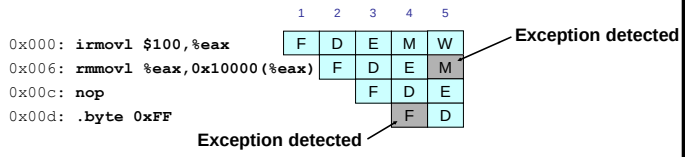
- Detect in Memory Stage

```
irmovl $100,%eax
rmmovl %eax,0x10000(%eax) # invalid address
```

36

Exceptions in Pipeline Processor #1

```
# demo-exc1.ys
irmovl $100,%eax
rmmovl %eax,0x10000(%eax) # Invalid address
nop
.byte 0xFF                # Invalid instruction code
```

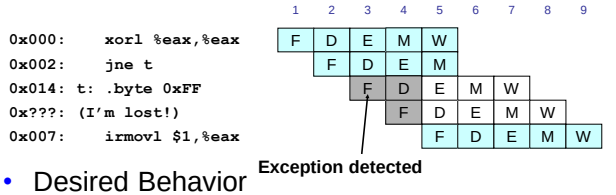


- Desired Behavior
  - `rmmovl` should cause exception

37

Exceptions in Pipeline Processor #2

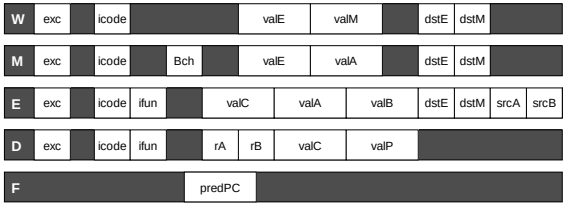
```
# demo-exc2.ys
0x000:    xorl %eax,%eax   # Set condition codes
0x002:    jne t            # Not taken
0x007:    irmovl $1,%eax
0x00d:    irmovl $2,%edx
0x013:    halt
0x014: t: .byte 0xFF       # Target
```



- Desired Behavior
  - No exception should occur

38

Maintaining Exception Ordering

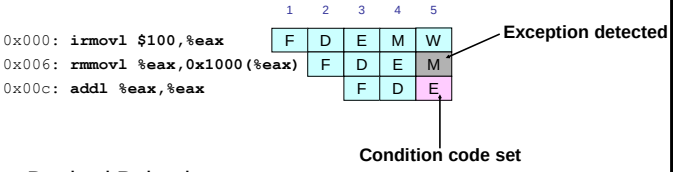


- Add exception status field to pipeline registers
- Fetch stage sets to either "AOK," "ADR" (when bad fetch address), or "INS" (illegal instruction)
- Decode & execute pass values through
- Memory either passes through or sets to "ADR"
- Exception triggered only when instruction hits write back

39

Side Effects in Pipeline Processor

```
# demo-exc3.ys
irmovl $100,%eax
rmmovl %eax,0x10000(%eax) # invalid address
addl %eax,%eax            # Sets condition codes
```



- Desired Behavior
  - `rmmovl` should cause exception
  - No following instruction should have any effect

40

Avoiding Side Effects

- Presence of Exception Should Disable State Update
  - When detect exception in memory stage
    - Disable condition code setting in execute
    - Must happen in same clock cycle
  - When exception passes to write-back stage
    - Disable memory write in memory stage
    - Disable condition code setting in execute stage

41

Rest of Exception Handling

- Calling Exception Handler
  - Push PC onto stack
    - Either PC of faulting instruction or of next instruction
    - Usually pass through pipeline along with exception status
  - Jump to handler address
    - Usually fixed address
    - Defined as part of ISA

42

More on: What does the hardware do?

- Precise exceptions: every instruction before the exception has completed and no instruction after the exception has had a noticeable effect
- Restartable exception: HW provides the OS with enough information to tell which instructions have completed, to complete those that have not completed (if any), and to get the pipeline going again in user mode
  - Squash necessary instructions
  - Disable further exceptions
  - Switch to kernel mode
  - Push resume PC (onto kernel stack)
  - Push other processor state (including condition codes)
  - Jump to a predefined address

43

System Calls

- **Each x86-64 system call has a unique ID number**
- **Examples:**

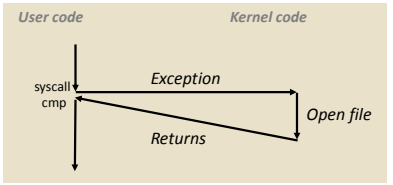
Number	*Name*	*Description*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

44

System Call Example: Opening File

- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
00000000000e5d70 <__open>:
...
e5d79:  b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:  0f 05               syscall         # Return value in %rax
e5d80:  48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:  c3                  retq
```



- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

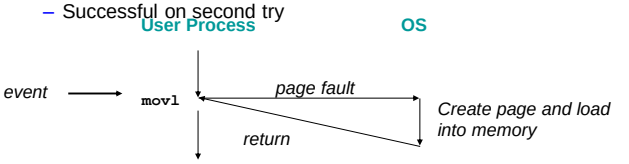
Fault Example: Page Fault

- Memory Reference
  - User writes to memory location
  - That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:    c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```

  - Page handler must load page into physical memory
  - Returns to faulting instruction
  - Successful on second try



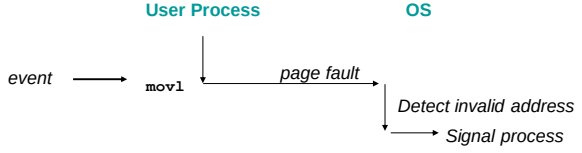
Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

- Memory Reference
  - User writes to memory location
  - Address is not valid

```
80483b7:    c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```

  - Page handler detects invalid address
  - Sends `SIGSEG` signal to user process
  - User process exits with "segmentation fault"

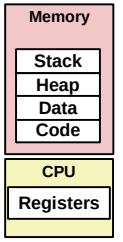


Today

- Exceptional Control Flow
- Exceptions
- Processes
- Process Control

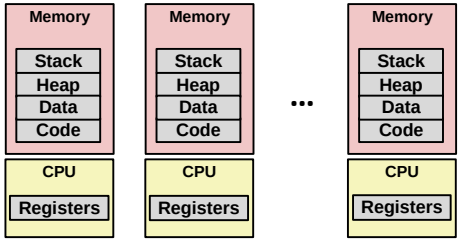
Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as "program" or "processor"



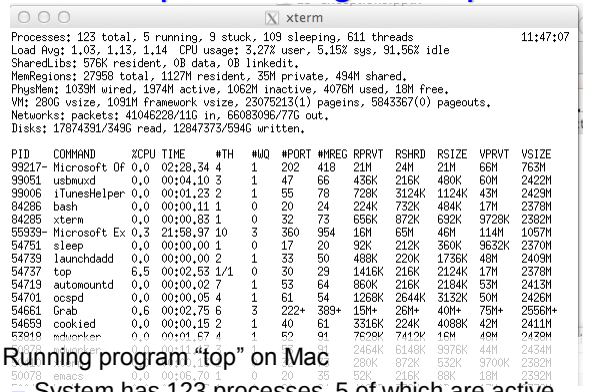
- Process provides each program with two key abstractions:
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*

Multiprocessing: The Illusion



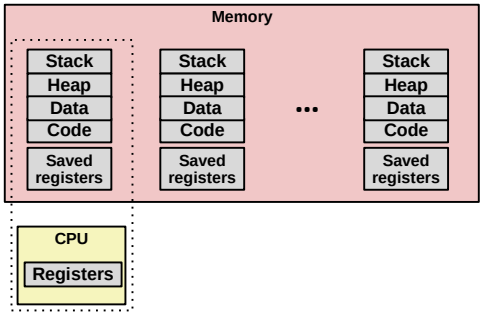
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example



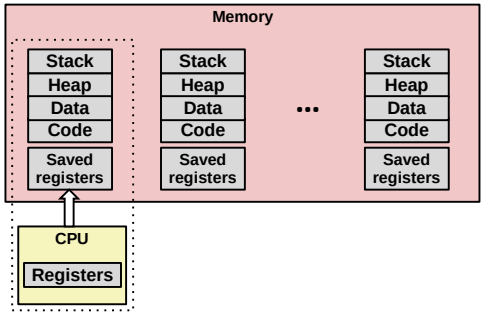
- Running program "top" on Mac
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Multiprocessing: The (Traditional) Reality



- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

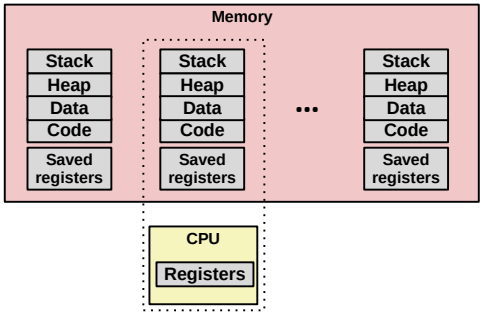
Multiprocessing: The (Traditional) Reality



- Save current registers in memory

53

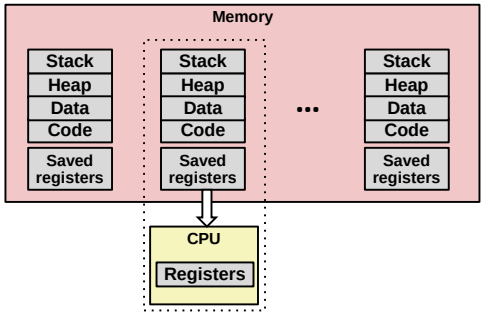
Multiprocessing: The (Traditional) Reality



- Schedule next process for execution

54

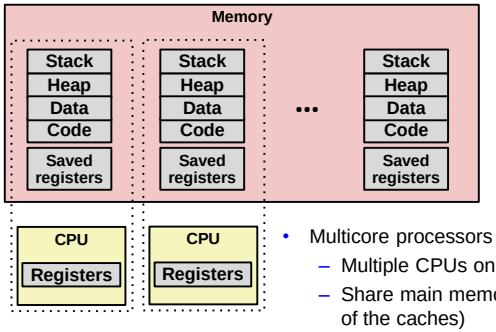
Multiprocessing: The (Traditional) Reality



- Load saved registers and switch address space (context switch)

55

Multiprocessing: The (Modern) Reality

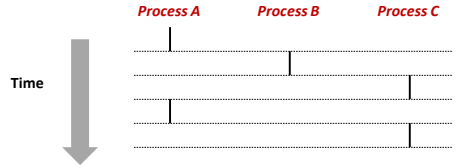


- Multicore processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

56

Concurrent Processes

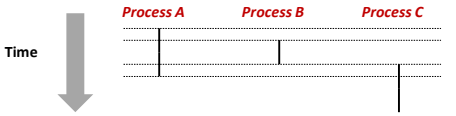
- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



57

User View of Concurrent Processes

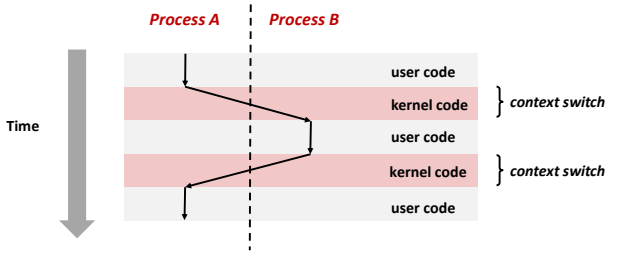
- Control flows for concurrent processes are physically disjoint in time
- However, we can think of concurrent processes as running in parallel with each other



58

Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*

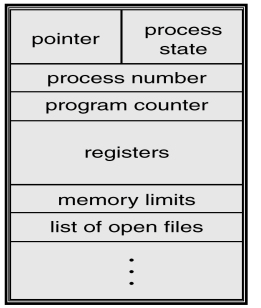


59

Process Control Block (PCB)

OS data structure (in kernel memory) maintaining information associated with each process.

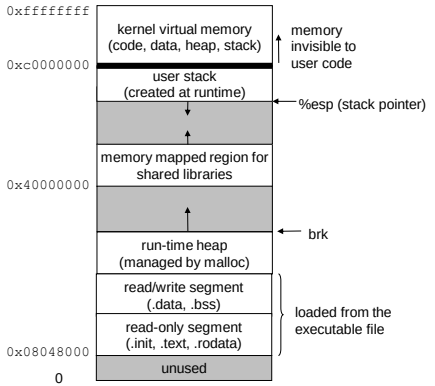
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- Information about open files
- maybe kernel stack?



60

Private Address Spaces

- Each process has its own private address space.



61