

OS for Database systems

Concurrency and IPC

1

Seehwan Yoo
Dankook University

Disclaimer: Some slides are borrowed from UC. Berkeley's 2014 OS and system programming

- 강의 개요
- OS 이론 강의 (8/29~9/2)
 - OS 개론, 프로세스. + mini 실습
 - 병렬성. + mini 실습
 - 메모리, IPC. + mini 실습
 - 파일시스템, 네트워킹. + mini 실습
 - 클라우드 컴퓨팅 및 시스템 관리
- OS 실습 강의 (9/19~9/23)
 - 리눅스 개발 환경 구축 및 셸 프로그래밍 (1d)
 - 다중쓰레드 자료구조 실습 프로젝트 (2d)
 - AWS 실습 및 WordPress 기반 DB 실습 프로젝트 (2d)

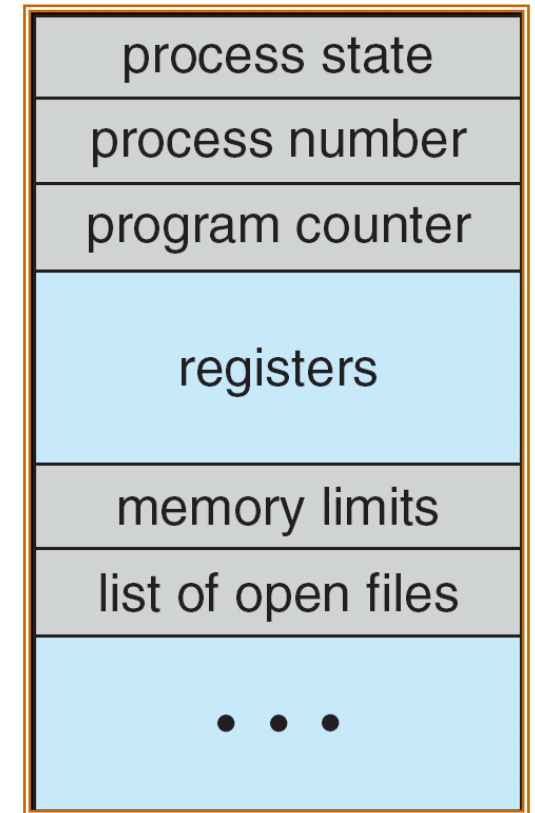
- Threads
- Concurrency
- SW synchronization
- HW synchronization
- synchronization primitives
- spinlock, mutex
- Some more practical problems and solutions

- Process: *Operating system abstraction to represent what is needed to run a single program*
 - Often called a “HeavyWeight Process”
- Two parts:
 - Sequential program execution stream
 - Code executed as a *sequential* stream of execution (i.e., thread)
 - Includes State of CPU registers
 - Protected resources:
 - Main memory state (contents of Address Space)
 - I/O state (i.e. file descriptors)

How do we Multiplex Processes?

5

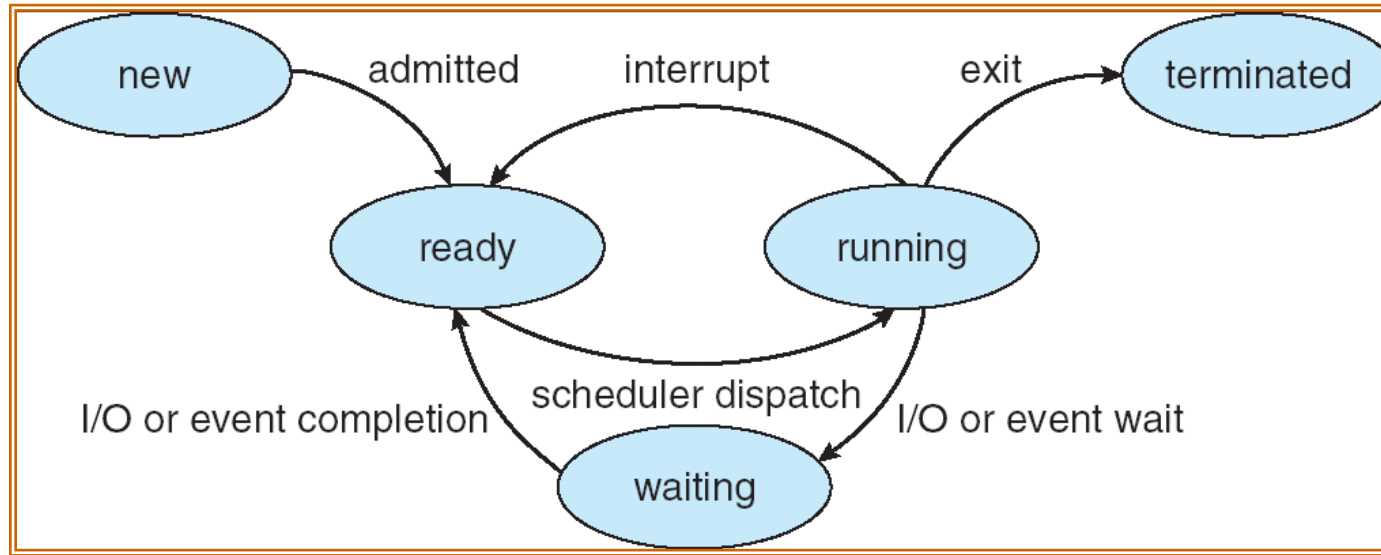
- The current state of process held in a process control block (PCB):
 - This is a “snapshot” of the execution and protection environment
 - Only one PCB active at a time
- Give out CPU time to different processes ([Scheduling](#)):
 - Only one process “running” at a time
 - Give more time to important processes
- Give pieces of resources to different processes ([Protection](#)):
 - Controlled access to non-CPU resources
 - Example mechanisms:
 - Memory Mapping: Give each process their own address space
 - Kernel/User duality: Arbitrary multiplexing of I/O through system calls



**Process
Control
Block**

Lifecycle of a Process

6

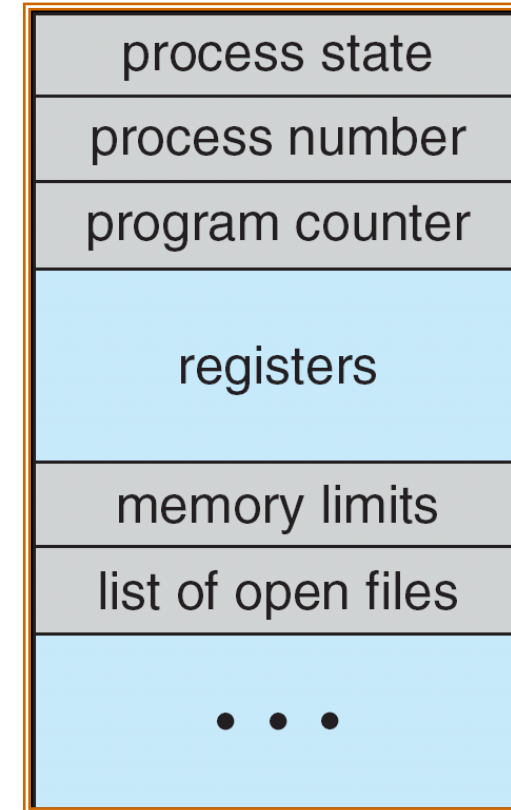


- As a process executes, it changes state:
 - **new**: The process is being created
 - **ready**: The process is waiting to run
 - **running**: Instructions are being executed
 - **waiting**: Process waiting for some event to occur
 - **terminated**: The process has finished execution

Process Control Block

7

- The current state of process held in a process control block (PCB):
(for a single-threaded process)



Process Control Block

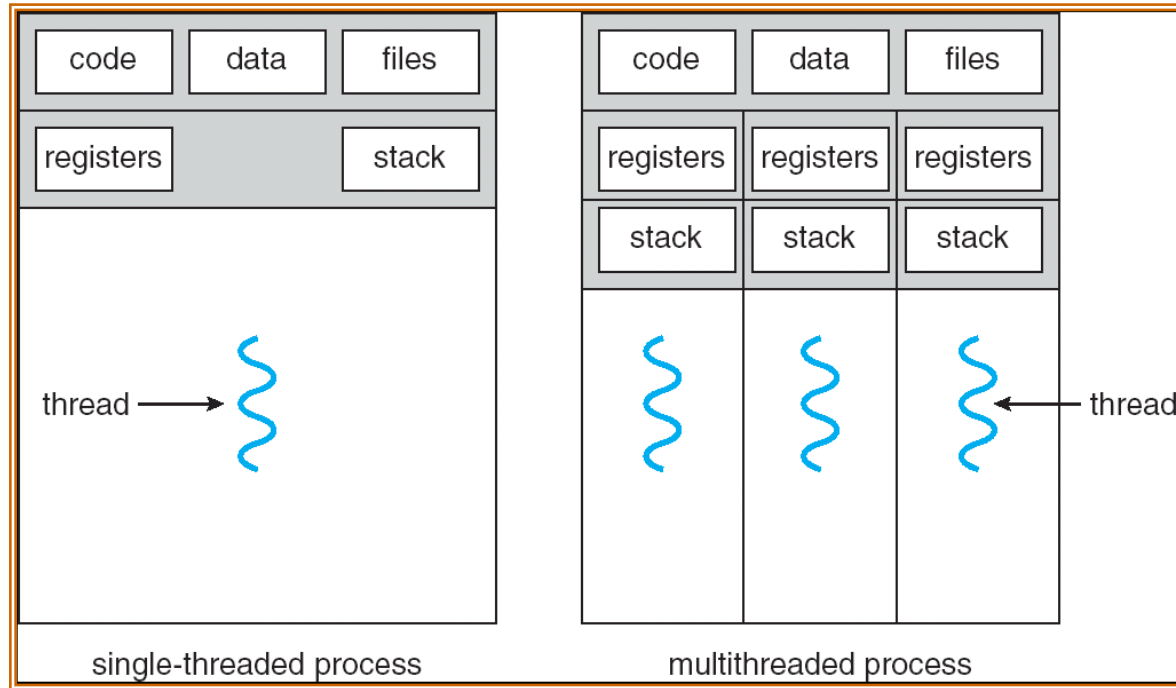
- Thread: *a sequential execution stream within process* (Sometimes called a “Lightweight process”)
 - Process still contains a single Address Space
 - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
 - Sometimes called multitasking, as in Ada ...
- Why separate the concept of a thread from that of a process?
 - Discuss the “thread” part of a process (concurrency)
 - Separate from the “address space” (protection)
 - Heavyweight Process \equiv Process with one thread

- Independently schedulable entity
- Sequential thread of execution that runs concurrently with other threads
 - It can block waiting for something while others progress
 - It can work in parallel with others
- Has local state (its stack) and shared state (static data and heap)

- State shared by all threads in process/addr space
 - Content of memory (global variables, heap)
 - I/O state (file system, network connections, etc)
- Execution Stack (logically private)
 - Parameters, temporary variables
 - Return PCs are kept while called procedures are executing
- State “private” to each thread
 - CPU registers (including, program counter)
 - Ptr to Execution stack
 - Kept in TCB \equiv Thread Control Block
 - When thread is not running
- Scheduler works on TCBs

Single and Multithreaded Processes

11



- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
 - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

Examples multithreaded programs

12

- Embedded systems
 - Elevators, Planes, Medical systems, Wristwatches
 - Single Program, concurrent operations
- Most modern OS kernels
 - Internally concurrent because have to deal with concurrent requests by multiple users
 - But no protection needed within kernel
- Database Servers
 - Access to shared data by many concurrent users
 - Also background utility processing must be done

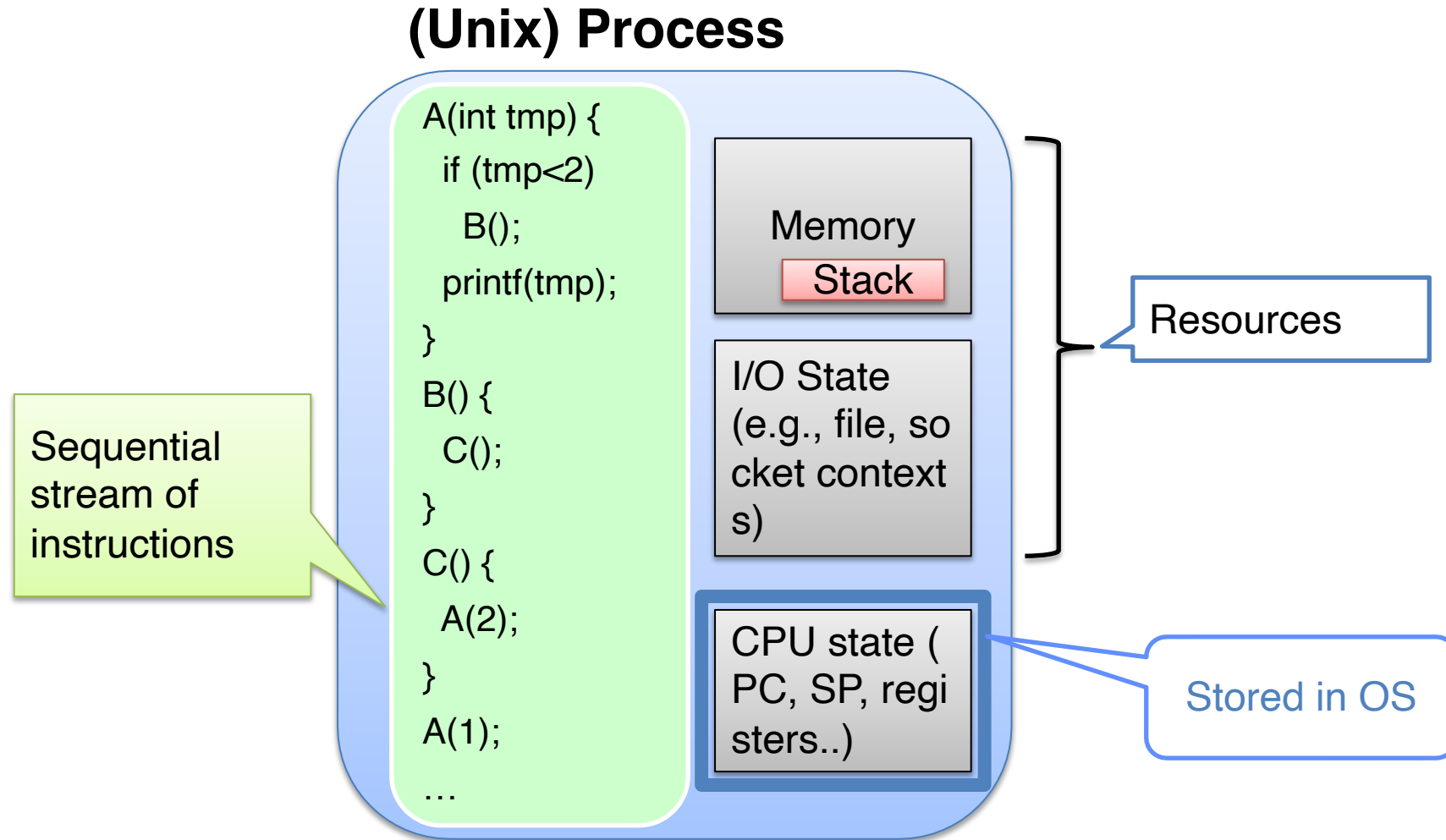
Example multithreaded programs (con't)

13

- Network Servers
 - Concurrent requests from network
 - Again, single program, multiple concurrent operations
 - File server, Web server, and airline reservation systems
- Parallel Programming (More than one physical CPU)
 - Split program into multiple threads for parallelism
 - This is called Multiprocessing
- Some multiprocessors are actually uniprogrammed:
 - Multiple threads in one address space but one program at a time

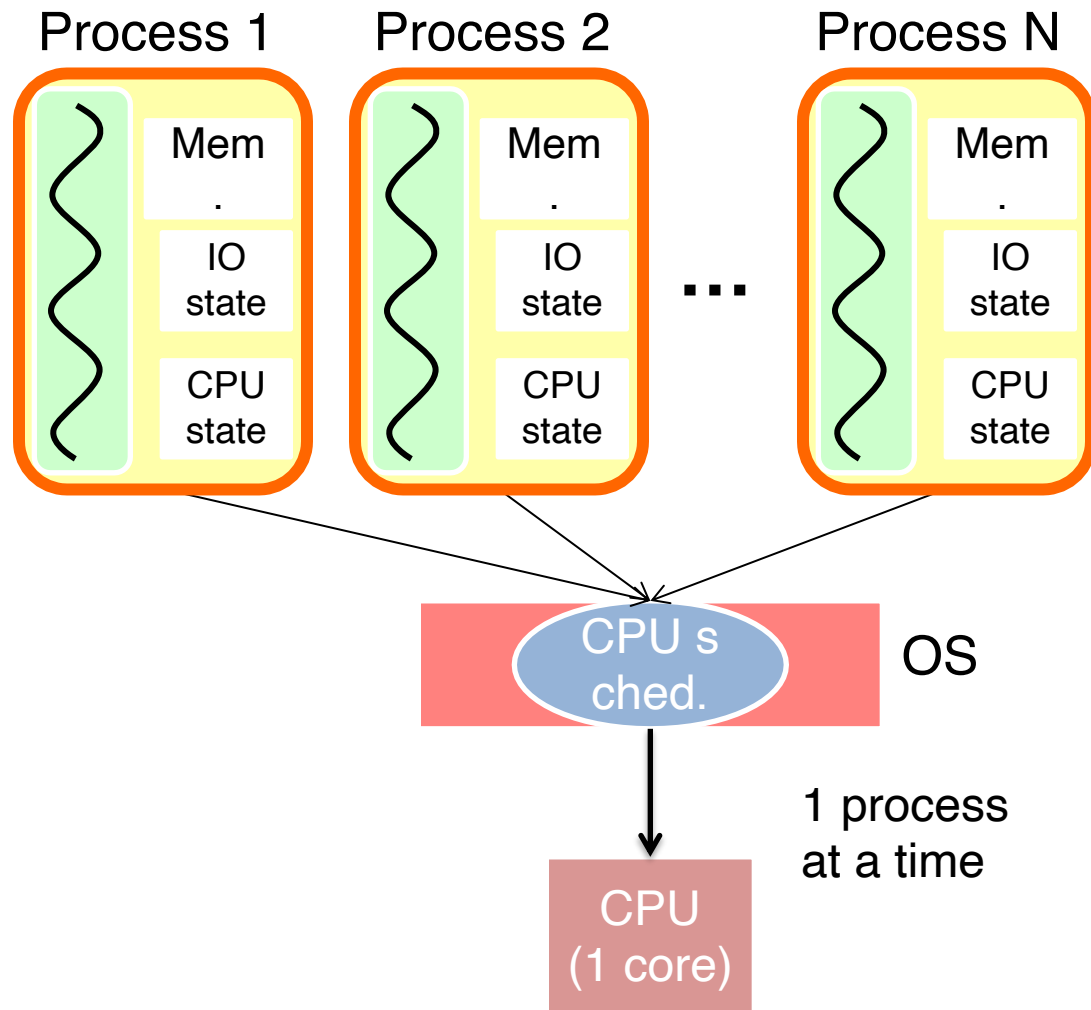
Putting it together: Process

14



Putting it together: Processes

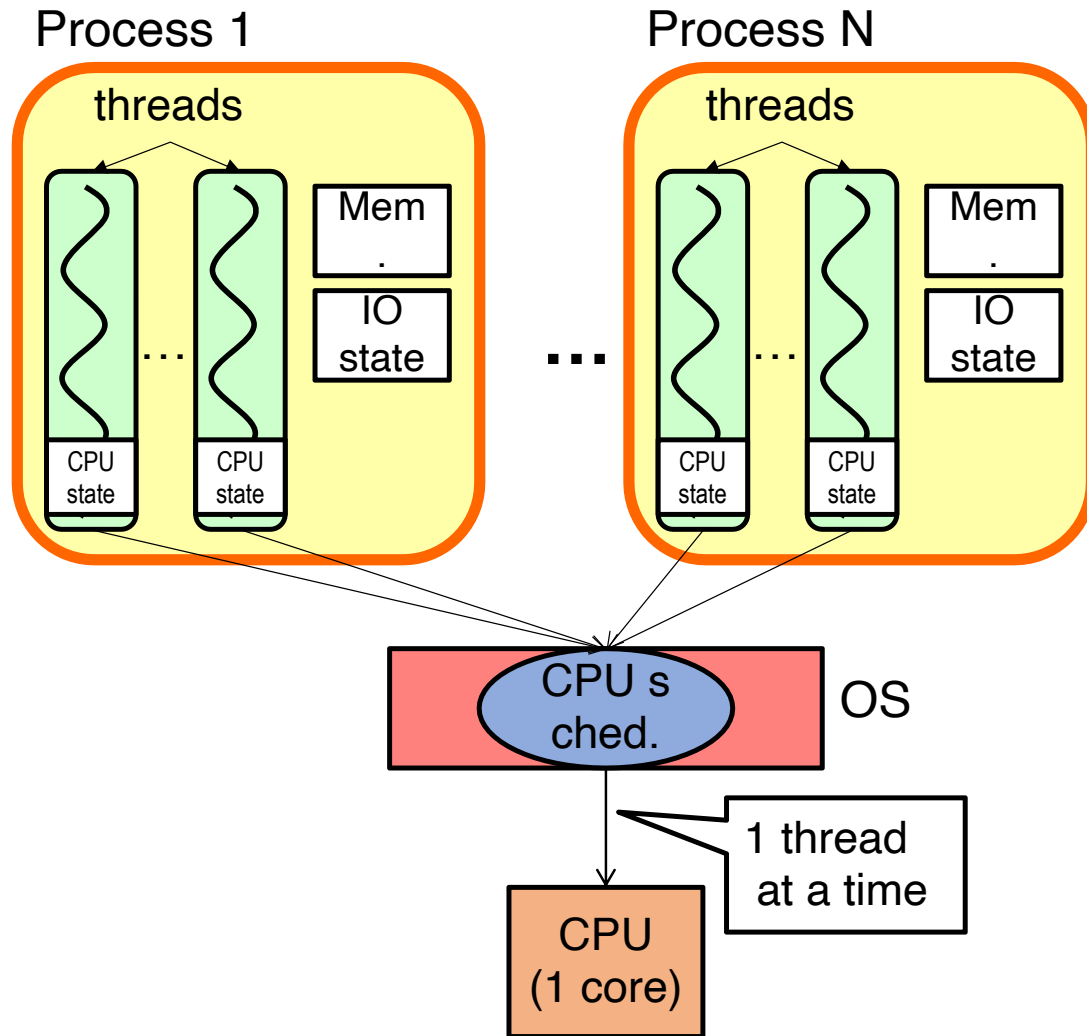
15



- Switch overhead: **high**
 - CPU state: **low**
 - Memory/IO state: **high**
- Process creation: **high**
- Protection
 - CPU: **yes**
 - Memory/IO: **yes**
- Sharing overhead: **high** (involves at least a context switch)

Putting it together: Threads

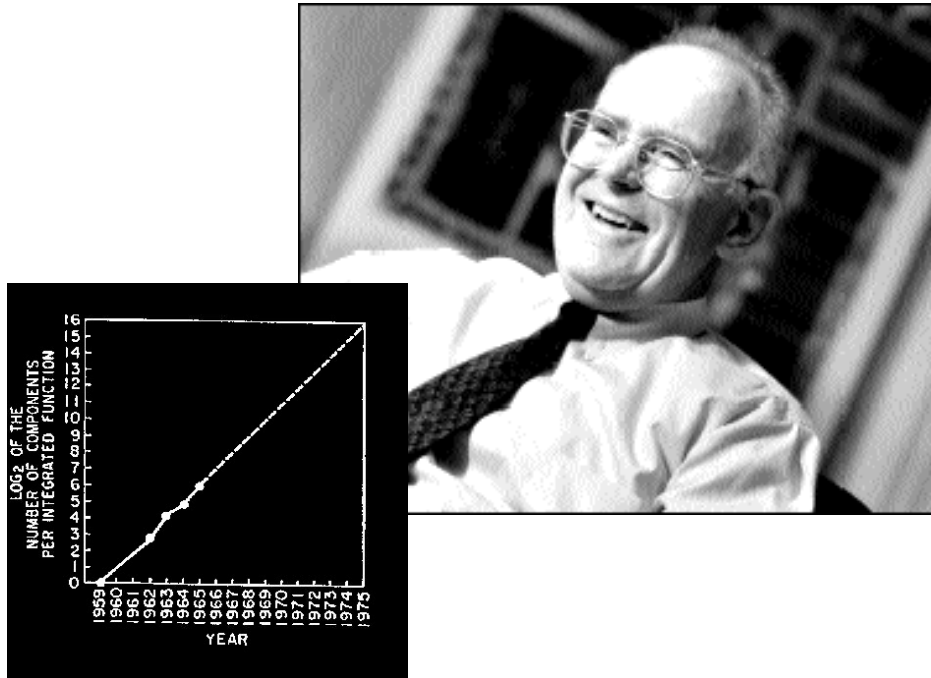
16



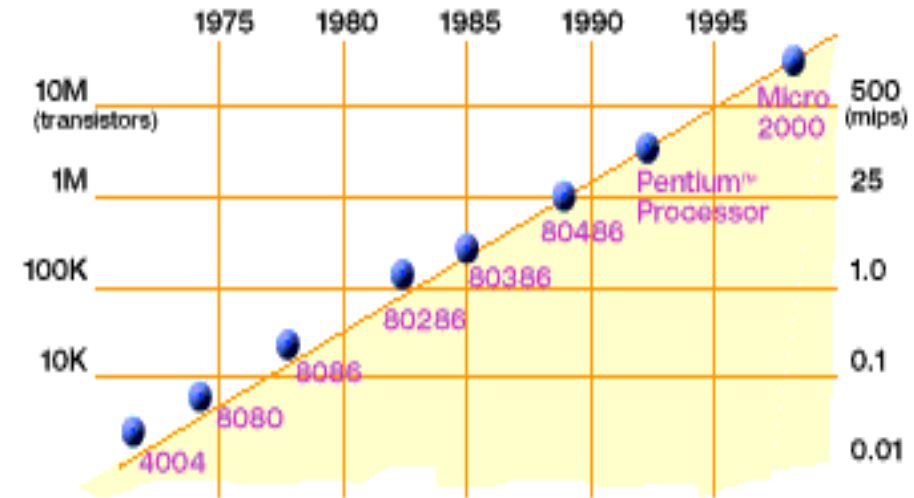
- Switch overhead: **low** (only CPU state)
- Thread creation: **low**
- Protection
 - CPU: **yes**
 - Memory/IO: **No**
- Sharing overhead: **low** (thread s witch overhead low)

Technology Trends: Moore's Law

17



Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

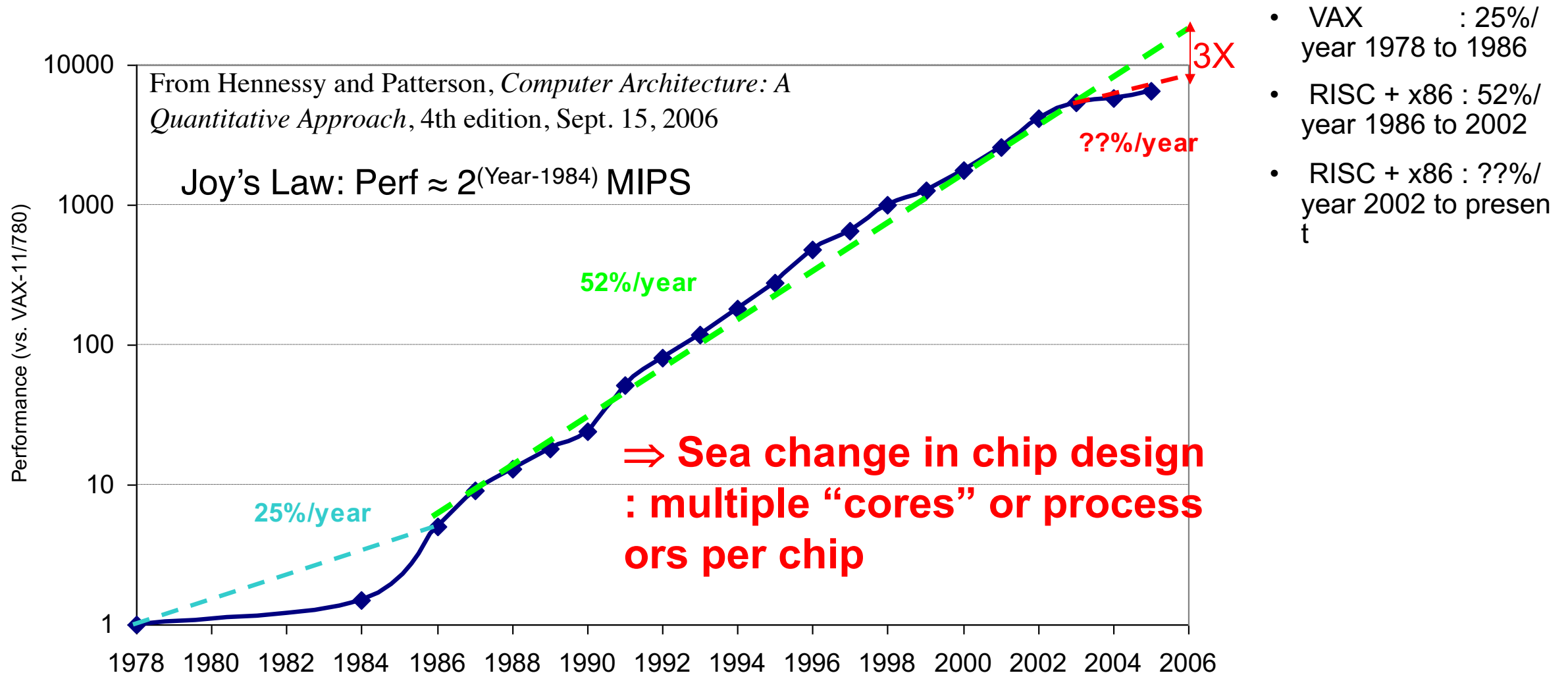


2X transistors/Chip Every 1.5 years
Called "Moore's Law"

Microprocessors have become smaller, denser, and more powerful.

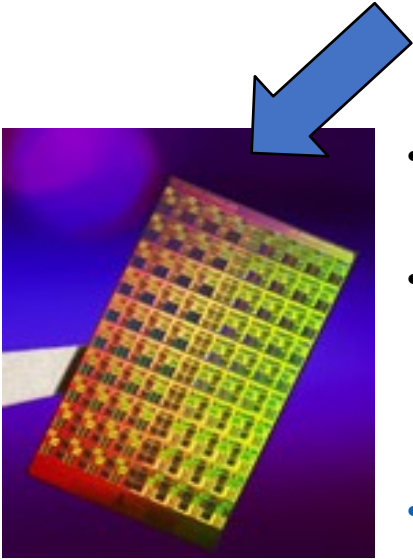
New Challenge: Slowdown in Joy's law of Performance

18



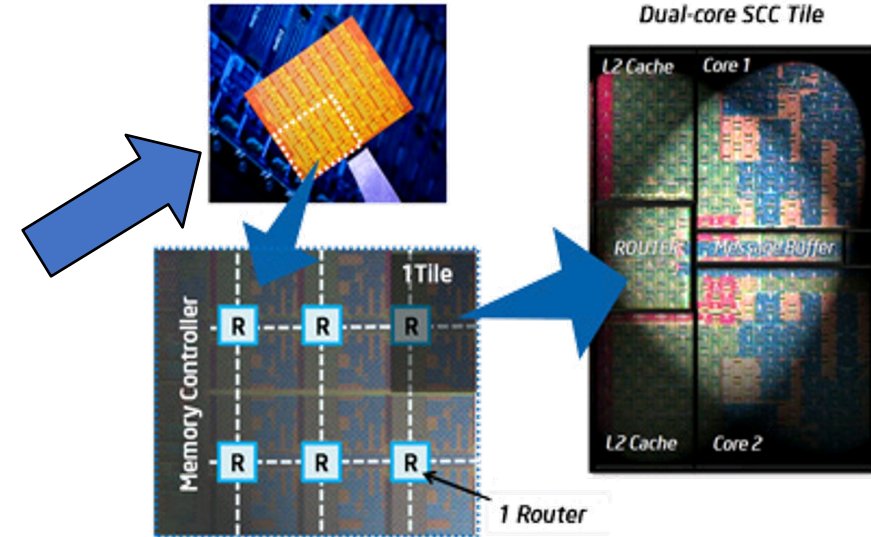
ManyCore Chips: The future is here

19



- “ManyCore” refers to many processors/chip
 - 64? 128? Hard to say exact boundary
- How to program these?
 - Use 2 CPUs for video/audio
 - Use 1 for word processor, 1 for browser
 - 76 for virus checking???
- Parallelism must be exploited at all levels

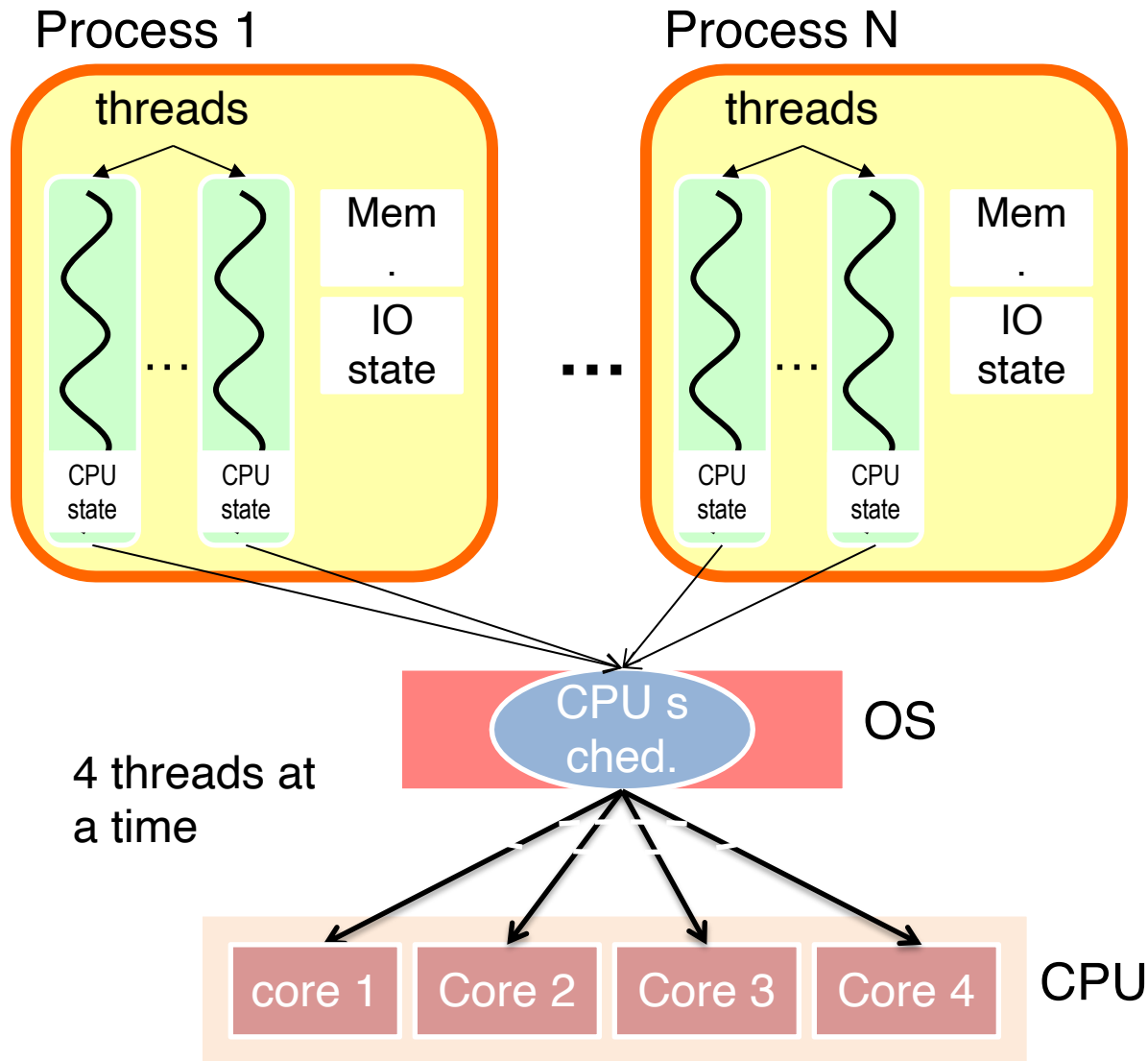
- Intel 80-core multicore chip (Feb 2007)
 - 80 simple cores
 - Two FP-engines / core
 - Mesh-like network
 - 100 million transistors



- Intel Single-Chip Cloud Computer (August 2010)
 - 24 “tiles” with two cores/tile
 - 24-router mesh network
 - 4 DDR3 memory controllers
 - Hardware support for message-passing

Putting it together: Multi-Cores

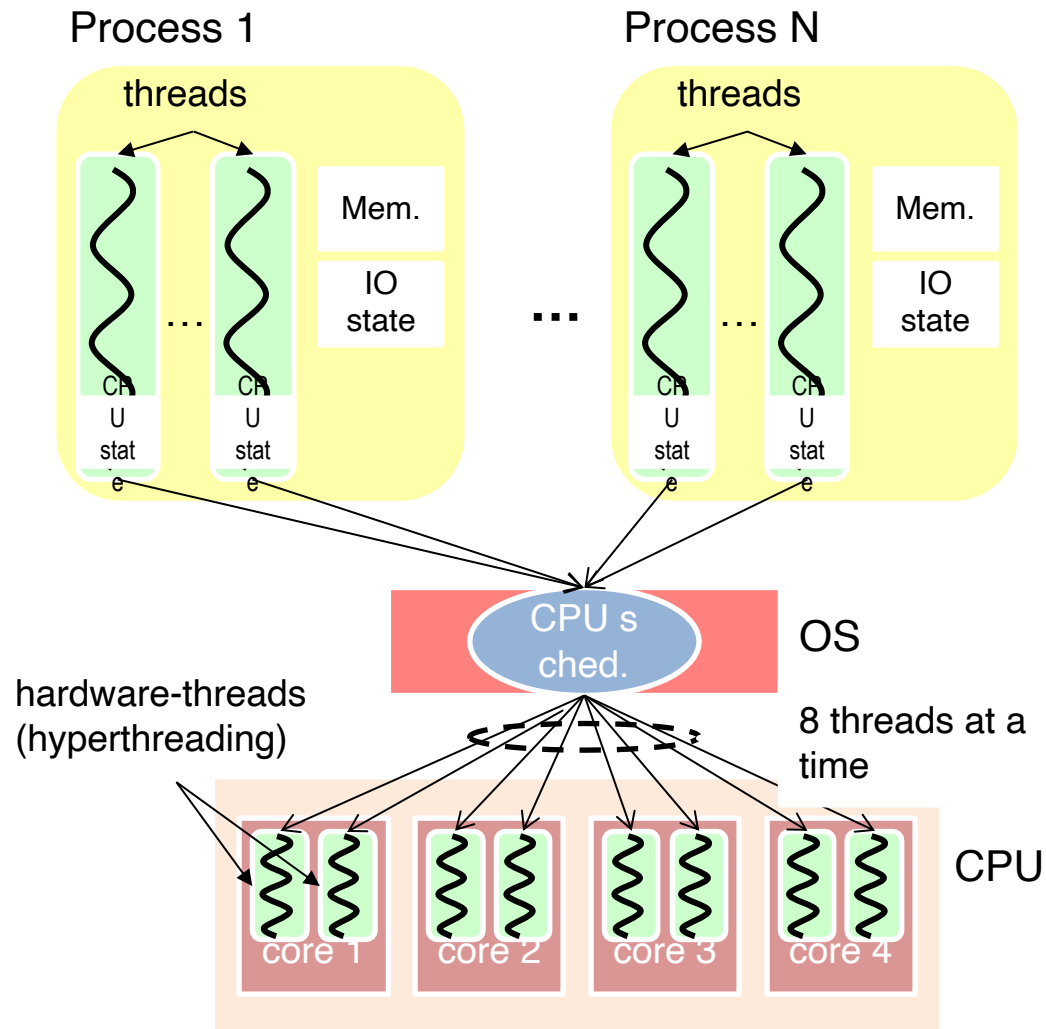
20



- Switch overhead: **low** (only CPU state)
- Thread creation: **low**
- Protection
 - CPU: **yes**
 - Memory/IO: **No**
- Sharing overhead: **low** (thread switch overhead low)

Putting it together: Hyper-Threading

21

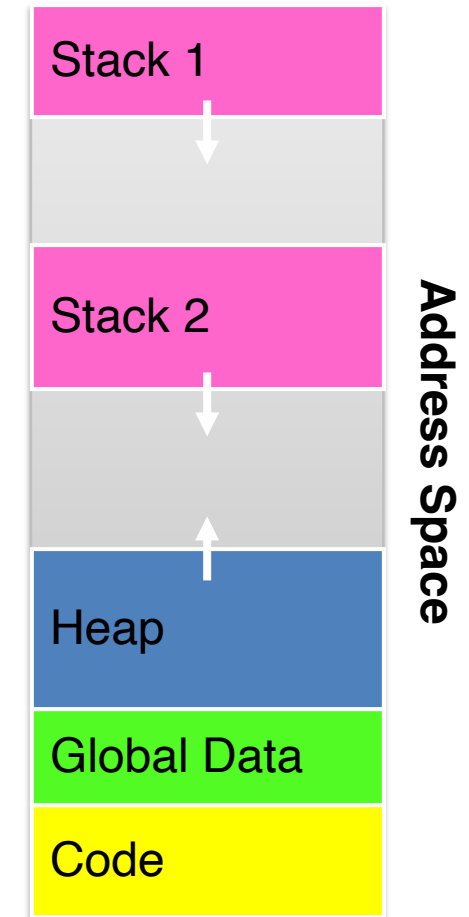


- Switch overhead between hardware-threads: **very-low** (done in hardware)
- Contention for ALUs/FPUs may hurt performance

Memory Footprint: Two-Threads

22

- If we stopped this program and examined it with a debugger, we would see
 - Two sets of CPU registers
 - Two sets of Stacks
- Questions:
 - How do we position stacks relative to each other?
 - What maximum size should we choose for the stacks?
 - What happens if threads violate this?
 - How might you catch violations?

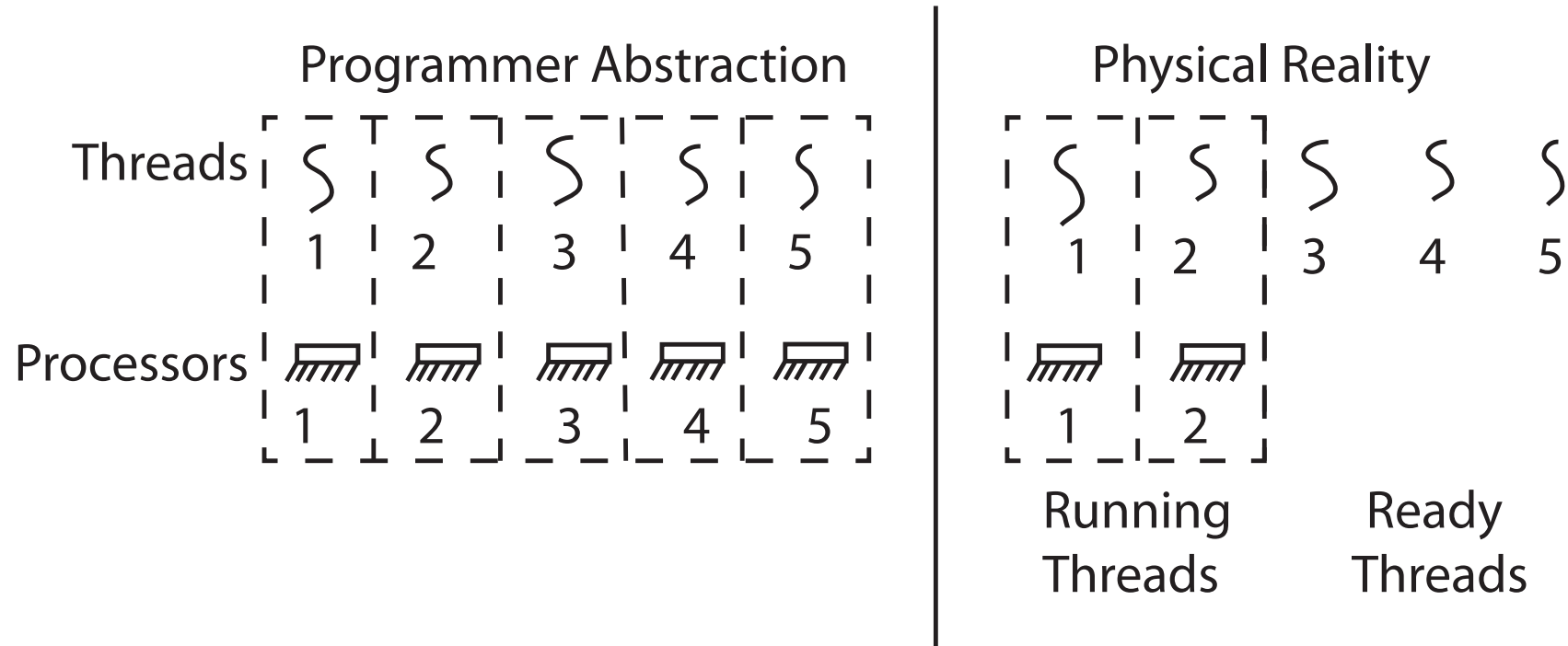


- `thread_fork(func, args)`
 - Create a new thread to run `func(args)`
 - `pthread_create`
- `thread_join(thread)`
 - In parent, wait for forked thread to exit, then return
 - `pthread_join`
- `thread_exit`
 - Quit thread and clean up, wake up joiner if any
 - `pthread_exit`

Thread Abstraction

24

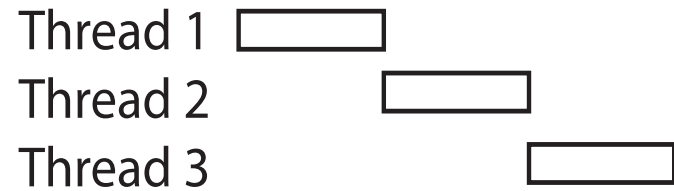
- Infinite number of processors
- Threads execute with variable speed
 - Programs must be designed to work with any schedule



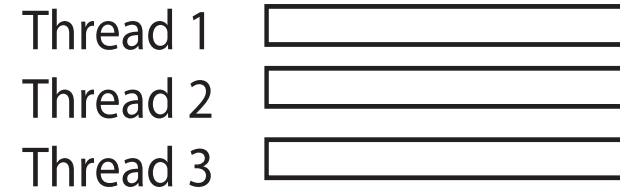
Programmer vs. Processor View

25

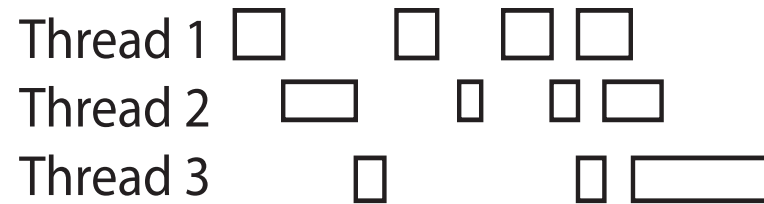
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
$x = x + 1;$	$x = x + 1;$	$x = x + 1$	$x = x + 1$
$y = y + x;$	$y = y + x;$	$y = y + x$
$z = x + 5y;$	$z = x + 5y;$	thread is suspended
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	thread is resumed
		$y = y + x$
		$z = x + 5y$	$z = x + 5y$



a) One execution



b) Another execution

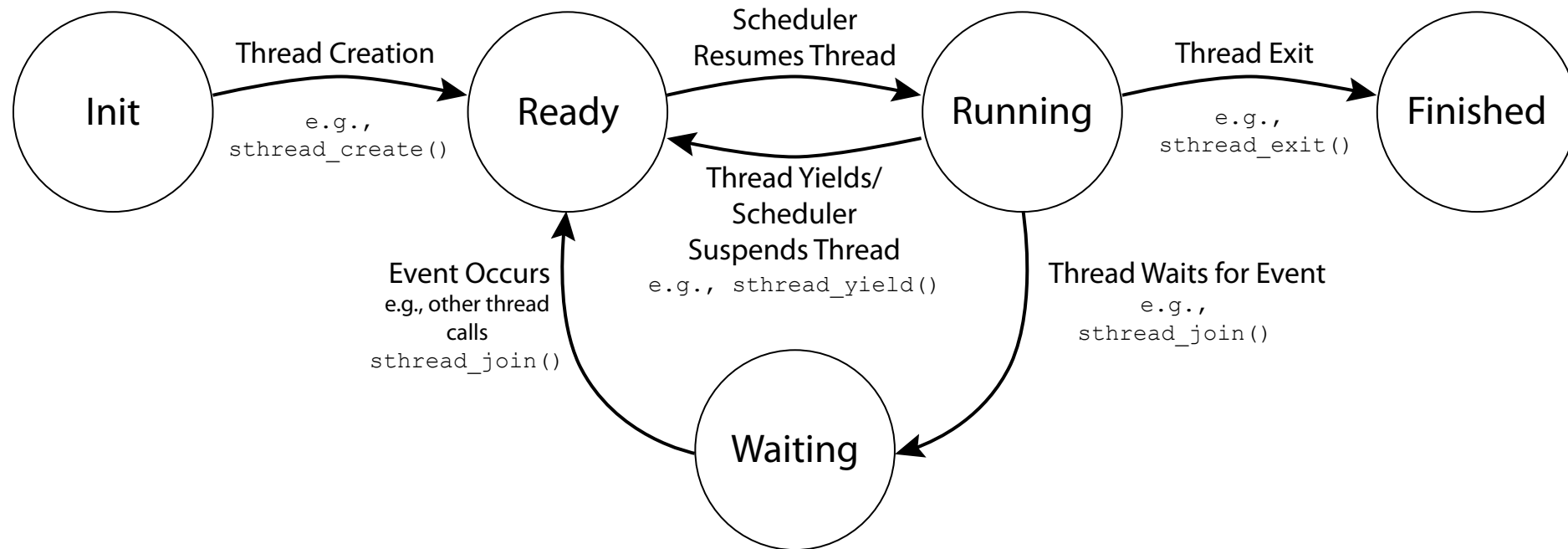


c) Another execution

- State shared by all threads in process/addr space
 - Content of memory (global variables, heap)
 - I/O state (file system, network connections, etc)
- State “private” to each thread
 - Kept in TCB \equiv Thread Control Block
 - CPU registers (including, program counter)
 - Execution stack - what is this?
- Execution Stack
 - Parameters, temporary variables
 - Return PCs are kept while called procedures are executing

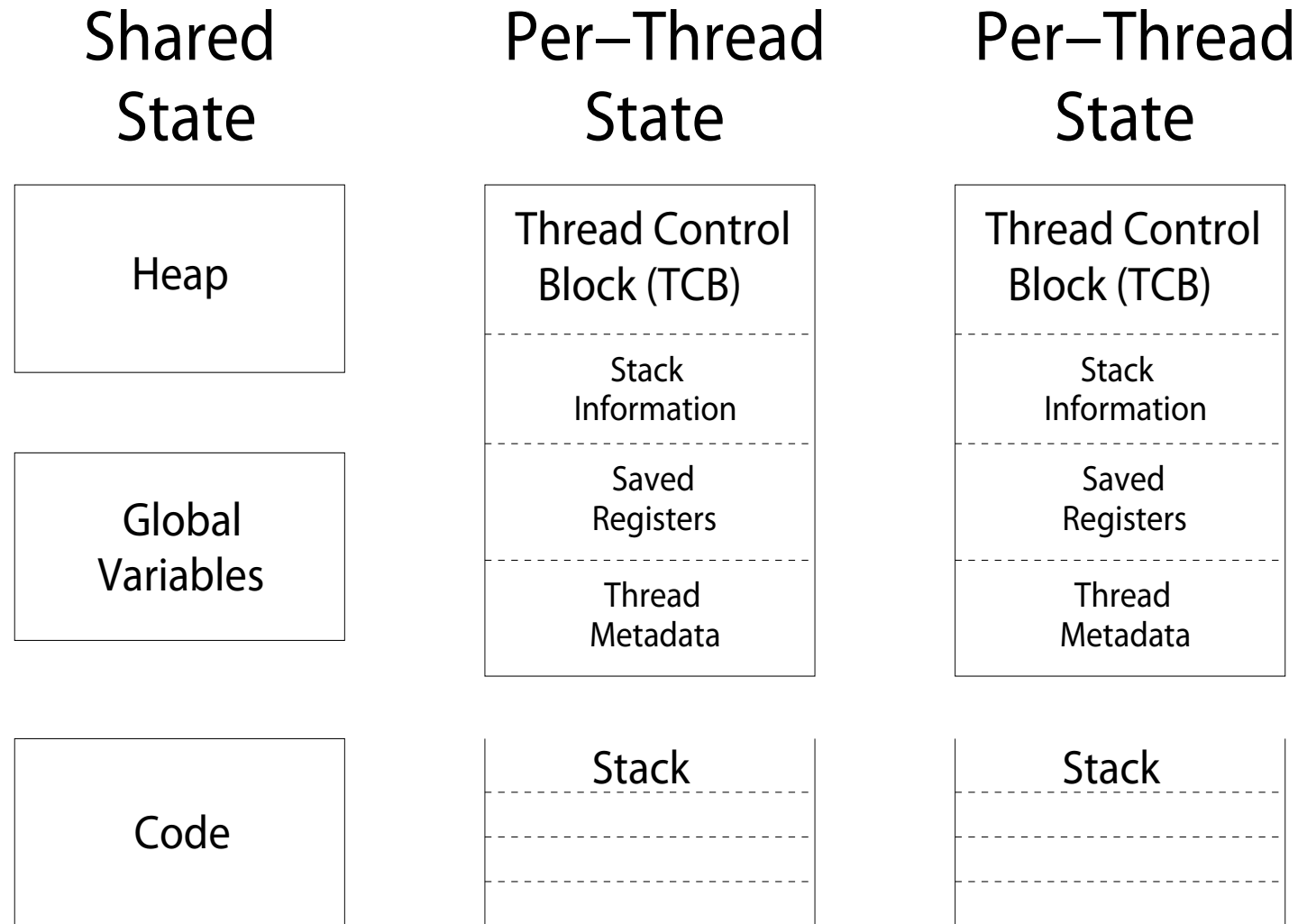
Thread Lifecycle

28



Shared vs. Per-Thread State

29

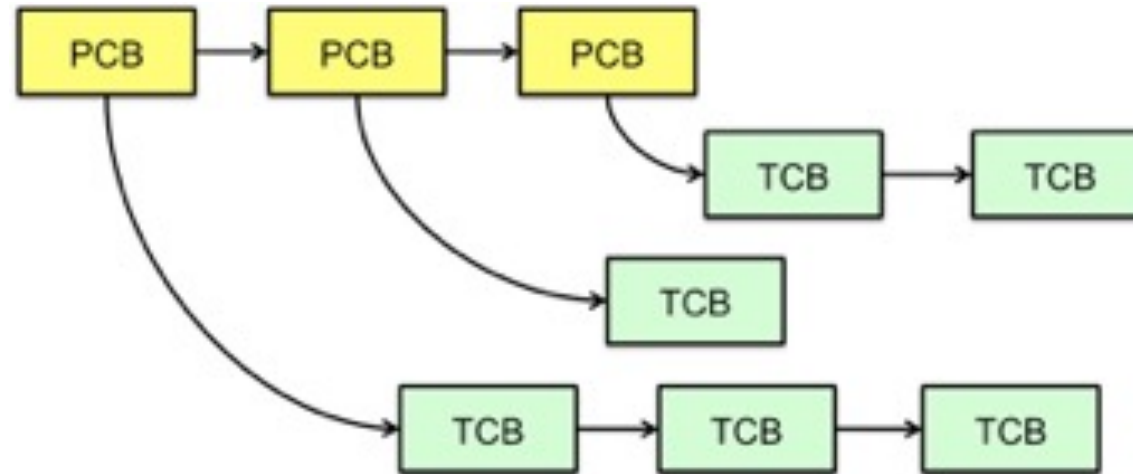


- Each Thread has a *Thread Control Block* (TCB)
 - Execution State: CPU registers, program counter (PC), pointer to stack (SP)
 - Scheduling info: state, priority, CPU time
 - Various Pointers (for implementing scheduling queues)
 - Pointer to enclosing process (PCB)
 - Etc (add stuff as you find a need)
- OS Keeps track of TCBs in protected memory
 - In Array, or Linked List, or ...

Multithreaded Processes

31

- PCB points to multiple TCBs:



- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables.

- Context switch in Linux: 3-4 microsecs (Current Intel i7 & E5).
- Thread switching faster than process switching (100 ns).
- But switching across cores about 2x more expensive than within-core switching.
- Context switch time increases sharply with the size of the working set*, and can increase 100x or more.
- * The working set is the subset of memory used by the process in a time window.
- **Moral:** Context switching depends mostly on cache limits and the process or thread's hunger for memory.

- Many processes are multi-threaded, so thread context switches may be either **within-process** or **across-processes**.

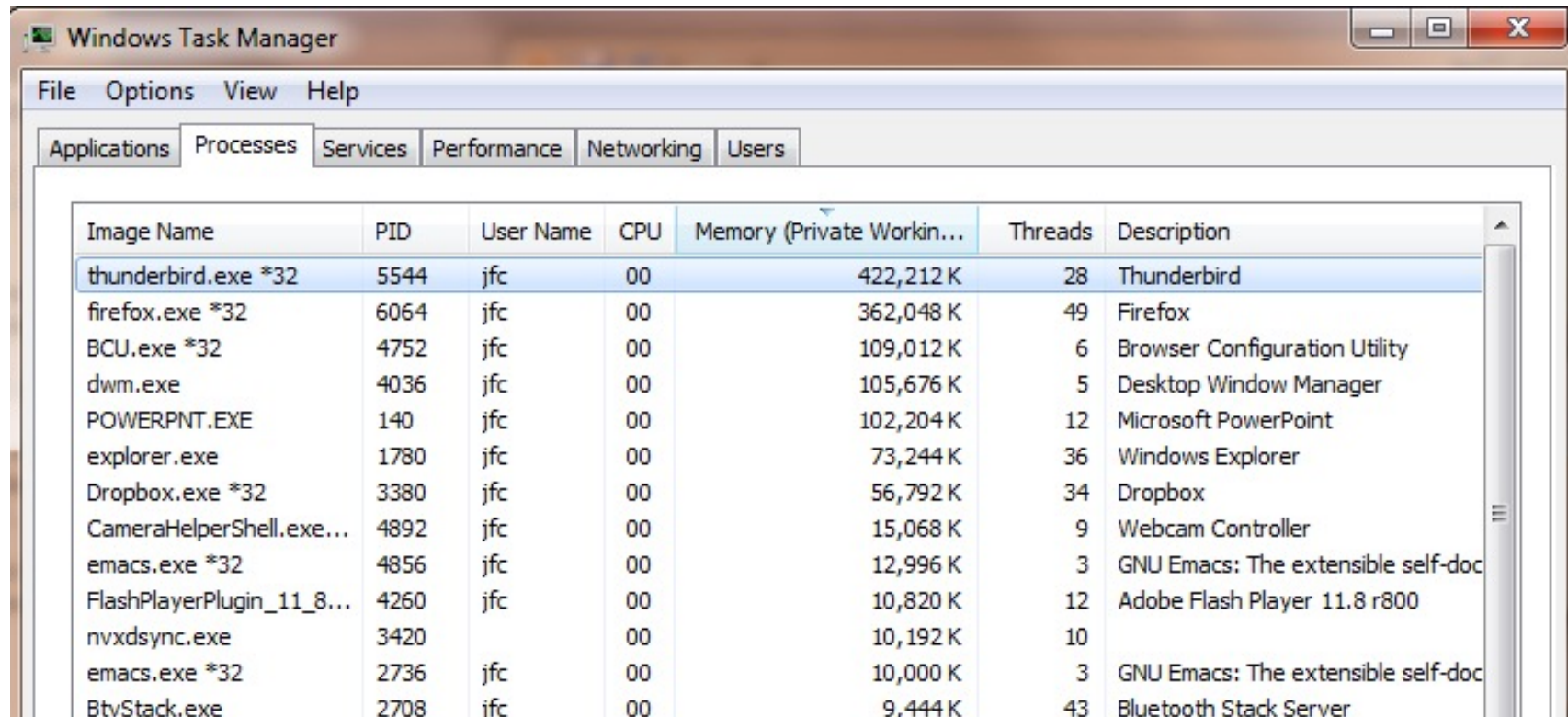


Image Name	PID	User Name	CPU	Memory (Private Workin...	Threads	Description
thunderbird.exe *32	5544	jfc	00	422,212 K	28	Thunderbird
firefox.exe *32	6064	jfc	00	362,048 K	49	Firefox
BCU.exe *32	4752	jfc	00	109,012 K	6	Browser Configuration Utility
dwm.exe	4036	jfc	00	105,676 K	5	Desktop Window Manager
POWERPNT.EXE	140	jfc	00	102,204 K	12	Microsoft PowerPoint
explorer.exe	1780	jfc	00	73,244 K	36	Windows Explorer
Dropbox.exe *32	3380	jfc	00	56,792 K	34	Dropbox
CameraHelperShell.exe...	4892	jfc	00	15,068 K	9	Webcam Controller
emacs.exe *32	4856	jfc	00	12,996 K	3	GNU Emacs: The extensible self-doc
FlashPlayerPlugin_11_8...	4260	jfc	00	10,820 K	12	Adobe Flash Player 11.8 r800
nvxdsync.exe	3420		00	10,192 K	10	
emacs.exe *32	2736	jfc	00	10,000 K	3	GNU Emacs: The extensible self-doc
BtvStack.exe	2708	ifc	00	9,444 K	43	Bluetooth Stack Server

- Threads are useful at user-level
 - Parallelism, hide I/O latency, interactivity
- Option A (early Java): user-level library, within a single-threaded process
 - Library does thread context switch
 - Kernel time slices between processes, e.g., on system call I/O
- Option B (Linux, MacOS, Windows): use kernel threads
 - System calls for thread fork, join, exit (and lock, unlock, ...)
 - Kernel does context switching
 - Simple, but a lot of transitions between user and kernel mode
- Option C (Windows): scheduler activations
 - Kernel allocates processors to user-level library
 - Thread library implements context switch
 - System call I/O that blocks triggers upcall
- Option D: Asynchronous I/O

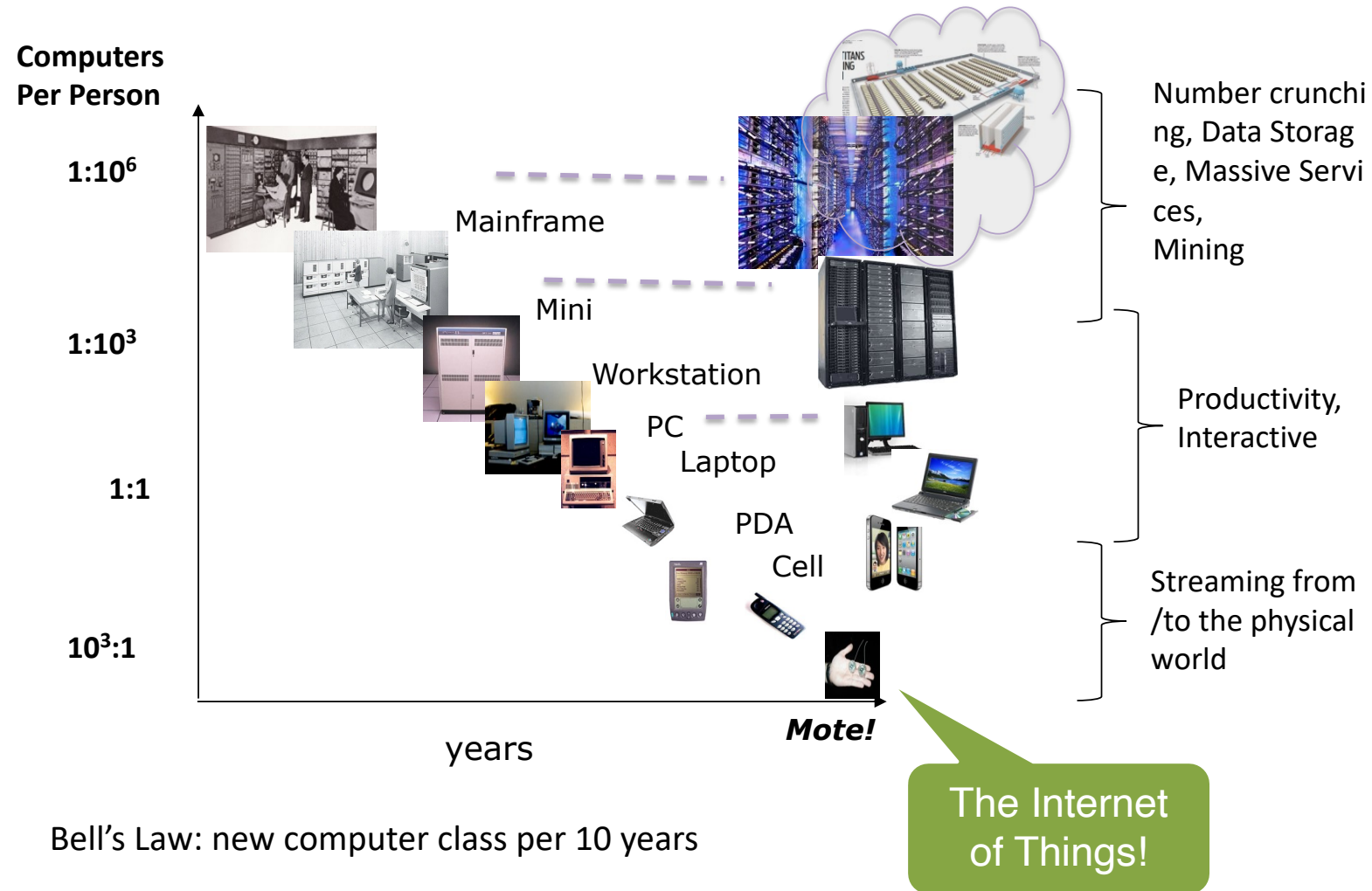
- Real operating systems have either
 - One or many address spaces
 - One or many threads per address space

# threads Per AS:	# of addr spaces:	One	Many
		MS/DOS, early Macintosh	Traditional UNIX
One			
Many		Embedded systems (Geoworks, VxWorks, JavaOS, etc) JavaOS, Pilot(PC)	Mach, OS/2, HP-UX, WinNT to 8, Solaris, OS X, Android, iOS

- Because of the cost of developing an OS from scratch, most modern OSes have a long lineage:
 - Multics → AT&T Unix → BSD Unix → Ultrix, SunOS, NetBSD, ...
 - Mach (micro-kernel) + BSD → NextStep → XNU → Apple OSX, iPhone iOS
 - Linux → Android OS
 - CP/M → QDOS → MS-DOS → Windows 3.1 → NT → 95 → 98 → 2000 → XP → Vista → 7 → 8 → phone → ...
 - Linux → RedHat, Ubuntu, Fedora, Debian, Suse, ...

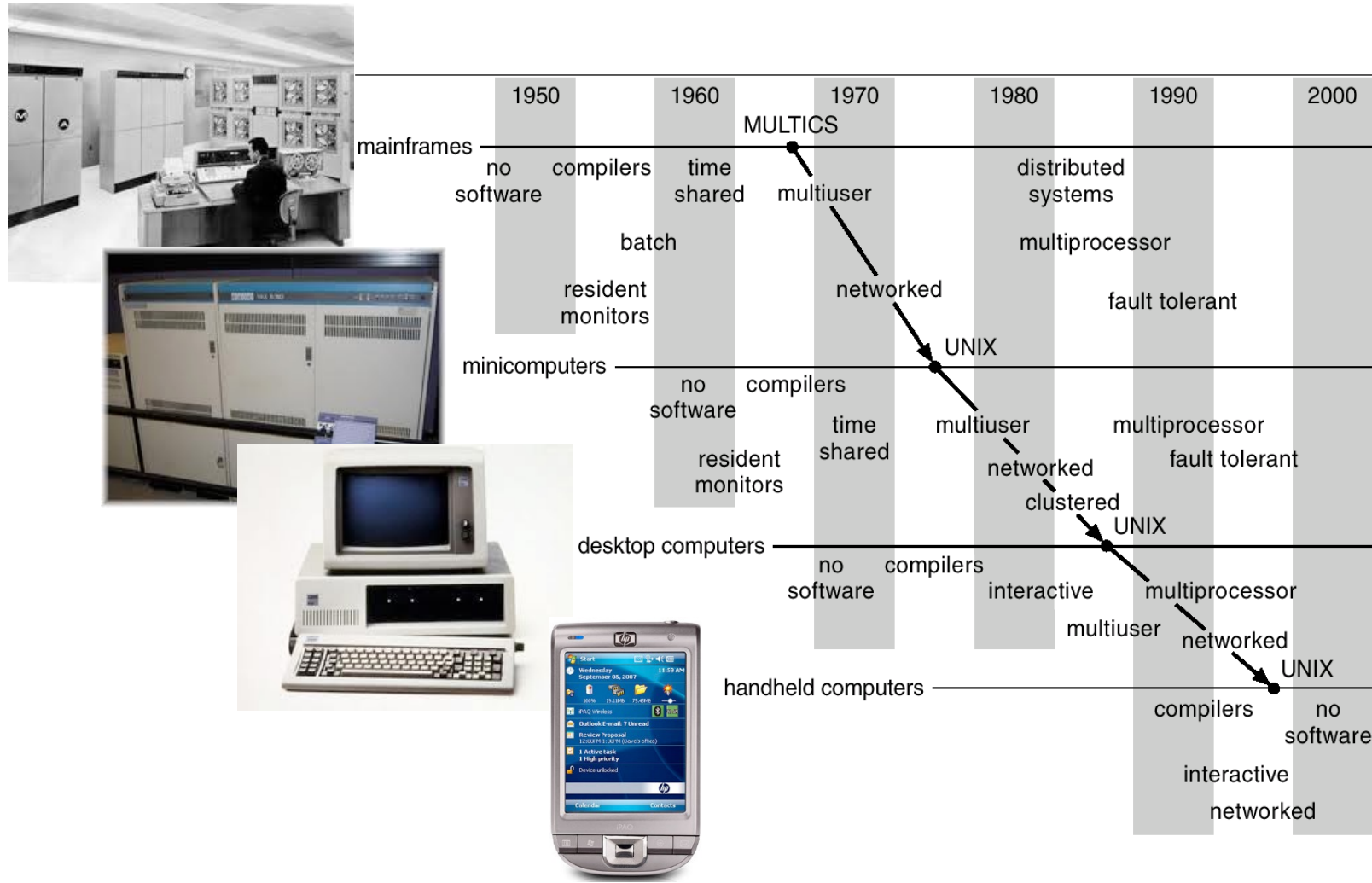
Dramatic change

37



Migration of OS Concepts and Features

38



Threads Synchronization & SW solutions

39

When multiple threads are running

40

- Access memory
- Makes progress according to the scheduler
 - However, we don't know when it will be scheduled out
- Some threads cooperate with each other
 - Withdraw
 - You claim your money from your account
 - Deposit
 - You save your money into your account
 - Both threads need to update balance

Multiple threads in cooperation

41

- Synchronization problem occurs
 - When multiple process are accessing shared variable
 - The shared variable should be mutually accessible
 - Only one process must complete operation before starting another operation

In-depth look of the synchronization problem

42

```
Withdraw(int amount)
{
    if (balance < amount)
        return -ENOMONEY;
    else balance -= amount;
    return balance;
}
```

```
Load r0, [&amount]
Load r1, [&balance]
cmp r1, r0
ble no_money // r1 < r0 —> no money
sub r1, r1, r0
Store r1, [&balance]
b func_end
no_money:
mov r0, #-enomoney
func_end:
mov pc, ra
```

```
Deposit (int amount)
{
    balance += amount;
    Return balance;
}
```

```
Load r0, [&amount]
Load r1, [&balance]
add r1, r1, r0
Store r1, [&balance]
```

There are some sensitive time instants

43

- Computer (CPU) works with registers!
 - Before updating the value to memory, value is only on the registers
 - While, others can access the memory!
 - without knowing the fact that another is accessing the memory
 - Possibly, we can have wrong value
-
- Balance = 100;
 - thread A
 - Deposit (50);
 - thread B
 - Withdraw (70);
- Expectation?
should be 80
 $100 + 50 - 70 = 80$

In-depth look of the synchronization problem

44

Deposit (int amount)

```
{  
    balance += amount;  
    Return balance;  
}
```

Load r0, [&amount]

Load r1, [&balance]

add r1, r1, r0

Store r1, [&balance]

Withdraw(int amount)

```
{  
    if (balance < amount)  
        return -ENOMONEY;  
    else balance -= amount;  
    return balance;  
}
```

Load r0, [&amount] // r0: 70

Load r1, [&balance] // r1: 100

cmp r1, r0

ble no_money // r1 < r0 —> no money

sub r1, r1, r0 // r1: 30

// scheduled out!

Store r1, [&balance]

b func_end

no_money:

mov r0, #-enomoney

func_end:

In-depth look of the synchronization problem

45

Deposit (int amount)

```
{  
    balance += amount;  
    Return balance;  
}
```

```
Load r0, [&amount] // r0: 50  
Load r1, [&balance] // r1: 100  
add r1, r1, r0 // r1: 150  
Store r1, [&balance]  
// Yeah! we have more money!  
// :>  
// scheduled out
```

Withdraw(int amount)

```
{  
    if (balance < amount)  
        return -ENOMONEY;  
    else balance -= amount;  
    return balance;  
}
```

```
Load r0, [&amount]  
Load r1, [&balance]  
cmp r1, r0  
ble no_money // r1 < r0 —> no money  
sub r1, r1, r0  
Store r1, [&balance]  
b func_end  
no_money:  
mov r0, #-enomoney  
func_end:  
mov pc, ra
```

In-depth look of the synchronization problem

46

Deposit (int amount)

```
{  
    balance += amount;  
    Return balance;  
}
```

Load r0, [&amount]

Load r1, [&balance]

add r1, r1, r0

Store r1, [&balance]

Withdraw(int amount)

```
{  
    if (balance < amount)  
        return -ENOMONEY;  
    else balance -= amount;  
    return balance;  
}
```

Load r0, [&amount]

Load r1, [&balance]

cmp r1, r0

ble no_money // r1 < r0 —> no money

sub r1, r1, r0

Store r1, [&balance] // r1: 30

// OMG! we lost money!

...

Threads synchronization problem

47

- Guarantee mutually exclusive access to the shared resource
 - usually memory access
- Race condition
 - Occurs when multiple processes or threads read and write data items
 - The final result depends on the order of execution
 - the “loser” of the race is the process that updates last and will determine the final value of the variable
- Critical section
 - code region (segment) that synchronization is required

A synchronization mechanism should provide

48

- Mutual exclusion
 - only one thread should access at a time
 - all others should not access the shared variable at the same time
 - data integrity has to be guaranteed
 - result in correct memory value
- Avoid deadlock
 - Somebody should make progress in the code
 - Deadlock: threads are alive, but nobody can make progress
 - All waits for another threads
- Avoid starvation
 - Could be serviced in bounded-waiting time

Solution to synchronization problem

49

- Read does not make problems (so far)
- Set a flag (turn) to enter critical section
 - Let turn = 0 for A to enter CS
 - Let turn = 1 for B to enter CS
 - Check the other's turn before entering the CS
 - When exit, reset turn for the other to enter

Example withdraw, deposit

50

```
A (int amount) {  
    turn = 0;  
    while (turn == 1) ;  
    Enter Critical Section  
    turn = 1;  
    return balance;  
}
```

```
mov r2, #0  
store r2, [&turn]
```

LOOP:

```
load r2, [&turn]  
cmp r2, #1  
beq LOOP
```

```
load r0, [&amount]  
load r1, [&balance]  
add r1, r1, r0  
store r1, [&balance]
```

```
mov r2, #1  
store r2, [&turn]
```

```
B (int amount) {  
    turn = 1;  
    while (turn == 0) ;  
    Enter Critical Section  
    turn = 0;  
    return balance;  
}
```

```
mov r2, #1  
store r2, [&turn]
```

LOOP:

```
load r2, [&turn]  
cmp r2, #0  
beq LOOP
```

```
load r0, [&amount]  
load r1, [&balance]
```

...

```
store r1, [&balance]
```

```
mov r2, #0  
store r2, [&turn]
```

Close, but... not an answer

51

Thread A:

`mov r2, #0 // Schedule out`

`store r2, [&turn]`

LOOP:

`load r2, [&turn]`

`cmp r2, #1`

`beq LOOP`

`load r0, [&amount]`

`load r1, [&balance]`

`add r1, r1, r0`

`store r1, [&balance]`

`mov r2, #1`

`store r2, [&turn]`

Thread B:

`mov r2, #1`

`store r2, [&turn]`

LOOP:

`load r2, [&turn]`

`cmp r2, #0`

`beq LOOP`

`load r0, [&amount]`

`load r1, [&balance]`

...

`store r1, [&balance]`

`mov r2, #0`

`store r2, [&turn]`

Close, but... not an answer

52

Thread A:

mov r2, #0 // stopped here

store r2, [&turn]

LOOP:

load r2, [&turn]

cmp r2, #1

beq LOOP

load r0, [&amount]

load r1, [&balance]

add r1, r1, r0

store r1, [&balance]

mov r2, #1

store r2, [&turn]

Thread B:

mov r2, #1

store r2, [&turn] // Now turn= 1

// pass the condition

LOOP:

load r2, [&turn]

cmp r2, #0

beq LOOP

// We entered the CS!

load r0, [&amount]

load r1, [&balance]

// Scheduled out at somewhere here!

...

store r1, [&balance]

mov r2, #0

store r2, [&turn]

Close, but... not an answer

53

Thread A:

mov r2, #0 // resume execution from here

store r2, [&turn] // OMG! turn= 0

// pass the condition

LOOP:

load r2, [&turn]

cmp r2, #1

beq LOOP

// We also entered the CS!

load r0, [&amount]

load r1, [&balance]

add r1, r1, r0

store r1, [&balance]

// mutual exclusion is broken

mov r2, #1

store r2, [&turn]

Thread B:

mov r2, #1

store r2, [&turn] // Now turn= 1

// passed the condition

LOOP:

load r2, [&turn]

cmp r2, #0

beq LOOP

// We entered the CS!

load r0, [&amount]

load r1, [&balance]

// Scheduled out at somewhere here!

...

store r1, [&balance]

mov r2, #0

store r2, [&turn]

Use additional variables, flags[2]

54

```
Deposit (int amount) {  
    flag[0] = true;  
    turn = 0;  
    while (turn == 1 && flag[1] == true) ;  
    Enter Critical Section  
    flag[0] = false;  
    turn = 1;  
    return balance;  
}
```

```
    mov r3, #1  
    store r3, [&flag]  
    mov r2, #0  
    store r2, [&turn]
```

```
LOOP:  
    load r2, [&turn]  
    cmp r2, #1  
    bne CS  
    load r3, [&flag+4]  
    cmp r3, #1  
    beq LOOP
```

```
CS:  
    mov r3, #0  
    store r3, [&flag]  
    mov r2, #1  
    store r2, [&turn]
```

```
Withdraw (int amount) {  
    flag[1] = true;  
    turn = 1;  
    while (turn == 0 && flag[0] == true) ;  
    Enter Critical Section  
    flag[1] = false;  
    turn = 0;  
    return balance;  
}
```

```
    mov r3, #1  
    store r3, [&flag+4]  
    mov r2, #1  
    store r2, [&turn]
```

```
LOOP:  
    load r2, [&turn]  
    cmp r2, #0  
    bne CS  
    load r3, [&flag]  
    cmp r3, #1  
    beq LOOP
```

```
CS:  
    mov r2, #0  
    store r2, [&flag+4]  
    mov r2, #0  
    store r2, [&turn]
```

Does that guarantee mutual exclusion?

55

Thread A:

```
mov r3, #1
store r3, [&flag] // it's safe cuz flag is not shared
mov r2, #0 // schedule out!
```

```
store r2, [&turn]
```

LOOP:

```
load r2, [&turn]
cmp r2, #1
bne CS
load r3, [&flag+4]
cmp r3, #1
beq LOOP
```

CS:

```
mov r3, #0
store r3, [&flag]
mov r2, #1
store r2, [&turn]
```

Thread B:

```
mov r3, #1
store r3, [&flag+4]
mov r2, #1
store r2, [&turn]
```

LOOP:

```
load r2, [&turn]
cmp r2, #0
bne CS
load r3, [&flag]
cmp r3, #1
beq LOOP
```

CS:

```
mov r2, #0
store r2, [&flag+4]
mov r2, #0
store r2, [&turn]
```

Does that guarantee mutual exclusion?

56

Thread A:

```
mov r3, #1
store r3, [&flag]
mov r2, #0 // stopped here
store r2, [&turn]
```

LOOP:

```
load r2, [&turn]
cmp r2, #1
bne CS
load r3, [&flag+4]
cmp r3, #1
beq LOOP
```

CS:

```
mov r3, #0
store r3, [&flag]
mov r2, #1
store r2, [&turn]
```

Thread B:

```
mov r3, #1
store r3, [&flag+4] // it's safe cuz flag+4 is not
shared
```

```
mov r2, #1
store r2, [&turn] // Now, turn=1
```

// pass 1st condition

LOOP:

```
load r2, [&turn]
cmp r2, #0
bne CS
```

// does not check 2nd condition

```
load r3, [&flag]
cmp r3, #1
beq LOOP
```

CS:

// Entered CS

// schedule out during condition check

```
mov r2, #0
store r2, [&flag+4]
mov r2, #0
store r2, [&turn]
```


Does that guarantee mutual exclusion?

57

Thread A:

```
    mov r3, #1
    store r3, [&flag]
    mov r2, #0 // stopped here
    store r2, [&turn] // Now turn = 0
// Pass 1st condition
LOOP:
    load r2, [&turn]
    cmp r2, #1
    bne CS
// does not check 2nd condition
    load r3, [&flag+4]
    cmp r3, #1
    beq LOOP
CS:
// We also entered CS
// Mutual exclusion broken!
    mov r3, #0
    store r3, [&flag]
    mov r2, #1
    store r2, [&turn]
```

Thread B:

```
    mov r3, #1
    store r3, [&flag+4] // it's safe cuz flag+4 is not
shared
    mov r2, #1
    store r2, [&turn]
LOOP:
    load r2, [&turn]
    cmp r2, #0
    bne CS
    load r3, [&flag]
    cmp r3, #1
    beq LOOP
CS:
// scheduled out
// We are in CS
    mov r2, #0
    store r2, [&flag+4]
    mov r2, #0
    store r2, [&turn]
```

Twisting the algorithm, yield first!

58

```
Deposit (int amount) {  
    flag[0] = true;  
    turn = 1; // NOTE HERE!  
    while (flag[1] == true && turn == 1) ;  
    // NOTE the checking logic!  
    Enter Critical Section  
    flag[0] = false;  
    return balance;  
}  
  
    mov r3, #1  
    store r3, [&flag]  
    mov r2, #1  
    store r2, [&turn]  
LOOP:  
    load r3, [&flag+4]  
    cmp r3, #1  
    bne CS  
    load r2, [&turn]  
    cmp r2, #1  
    beq LOOP  
CS:  
    mov r3, #0  
    store r3, [&flag]  
    // mov r2, #1  
    // store r2, [&turn]
```

```
Withdraw (int amount) {  
    flag[1] = true;  
    turn = 0; // NOTE HERE!  
    while (flag[0] == true && turn == 0) ;  
    // NOTE the checking logic!  
    Enter Critical Section  
    flag[1] = false;  
    return balance;  
}  
  
    mov r3, #1  
    store r3, [&flag+4]  
    mov r2, #0  
    store r2, [&turn]  
LOOP:  
    load r3, [&flag]  
    cmp r3, #1  
    bne CS  
    load r2, [&turn]  
    cmp r2, #0  
    beq LOOP  
CS:  
    mov r2, #0  
    store r2, [&flag+4]  
    // mov r2, #0  
    // store r2, [&turn]
```

Does that guarantee mutual exclusion?

59

Thread A:

```
mov r3, #1
store r3, [&flag] // it's safe cuz flag is not shared
mov r2, #1 // schedule out!
store r2, [&turn]
```

LOOP:

```
load r3, [&flag+4]
cmp r3, #1
bne CS
load r2, [&turn]
cmp r2, #1
beq LOOP
```

CS:

```
mov r3, #0
store r3, [&flag]
// mov r2, #1
// store r2, [&turn]
```

Thread B:

```
mov r3, #1
store r3, [&flag+4]
mov r2, #0
store r2, [&turn]
```

LOOP:

```
load r3, [&flag]
cmp r3, #1
bne CS
load r2, [&turn]
cmp r2, #0
beq LOOP
```

CS:

```
mov r2, #0
store r2, [&flag+4]
// mov r2, #0
// store r2, [&turn]
```

Does that guarantee mutual exclusion?

60

Thread A:

```
    mov r3, #1
    store r3, [&flag] // it's safe cuz flag is not shared
    mov r2, #1 // stopped here
    store r2, [&turn]
LOOP:
    load r3, [&flag+4]
    cmp r3, #1
    bne CS
    load r2, [&turn]
    cmp r2, #1
    beq LOOP
CS:
    mov r3, #0
    store r3, [&flag]
    // mov r2, #1
    // store r2, [&turn]
```

Thread B:

```
    mov r3, #1
    store r3, [&flag+4] // it's safe cuz flag is not shared
    mov r2, #0
    store r2, [&turn] // Now, turn = 0
    // Can we pass branch condition?
    // Process A set flag[0] = 1, turn = 0
LOOP:
    load r3, [&flag]
    cmp r3, #1
    bne CS
    // Check 2nd condition
    load r2, [&turn]
    cmp r2, #0
    beq LOOP
    // Schedule out inside LOOP, checking condition
CS:
    mov r2, #0
    store r2, [&flag+4]
    // mov r2, #0
    // store r2, [&turn]
```

Does that guarantee mutual exclusion?

61

Thread A:

```
mov r3, #1
store r3, [&flag] // it's safe cuz flag is not shared
mov r2, #1 // stopped here
store r2, [&turn] // Now, turn = 1
// Can we pass branch condition?
// Process B set flag[1] = 1, turn = 1
LOOP:
load r3, [&flag+4]
cmp r3, #1
bne CS
load r2, [&turn]
cmp r2, #1
beq LOOP
// Schedule out inside LOOP, checking condition
CS:
mov r3, #0
store r3, [&flag]
// mov r2, #1
// store r2, [&turn]
```

Thread B:

```
mov r3, #1
store r3, [&flag+4] // it's safe cuz flag is not shared
mov r2, #0
store r2, [&turn] // Now, turn = 0
LOOP:
load r3, [&flag]
cmp r3, #1
bne CS
load r2, [&turn]
cmp r2, #0
beq LOOP
// stopped inside LOOP, checking condition
cmp r2, #0
beq LOOP
CS:
mov r2, #0
store r2, [&flag+4]
// mov r2, #0
// store r2, [&turn]
```

Does that guarantee mutual exclusion?

62

Thread A:

```
    mov r3, #1
    store r3, [&flag] // it's safe cuz flag is not shared
    mov r2, #1 // stopped here
    store r2, [&turn] // Now, turn = 1
LOOP:
    load r3, [&flag+4]
    cmp r3, #1
    bne CS
    load r2, [&turn]
    cmp r2, #1
    beq LOOP
// stopped inside LOOP, checking condition
CS:
    mov r2, #0
    store r2, [&flag]
    // mov r2, #1
    // store r2, [&turn]
```

Thread B:

```
    mov r3, #1
    store r3, [&flag+4] // it's safe cuz flag is not shared
    mov r2, #0
    store r2, [&turn] // Now, turn = 0
// stopped inside LOOP, checking condition
// Process A changed turn = 1, at somewhere
LOOP:
    load r3, [&flag]
    cmp r3, #1
    bne CS
    load r2, [&turn]
    cmp r2, #0
    beq LOOP
// Now we enter CS due to turn flag
CS:
// Schedule out
    mov r2, #0
    store r2, [&flag+4]
    // mov r2, #0
    // store r2, [&turn]
```

Does that guarantee mutual exclusion?

63

Thread A:

```
    mov r3, #1
    store r3, [&flag] // it's safe cuz flag is not shared
    mov r2, #1 // stopped here
    store r2, [&turn] // Now, turn = 1
LOOP:
    load r3, [&flag+4]
    cmp r3, #1
    bne CS
    load r2, [&turn]
    cmp r2, #1
    beq LOOP
// stopped inside LOOP, checking condition
// No check condition changed
// scheduled out inside LOOP, checking condition
CS:
    mov r2, #0
    store r2, [&flag]
    // mov r2, #1
    // store r2, [&turn]
```

Thread B:

```
    mov r3, #1
    store r3, [&flag+4] // it's safe cuz flag is not shared
    mov r2, #0
    store r2, [&turn] // Now, turn = 0
LOOP:
    load r3, [&flag]
    cmp r3, #1
    bne CS
    load r2, [&turn]
    cmp r2, #0
    beq LOOP
CS:
// Now we are in the CS
// stopped here
    mov r2, #0
    store r2, [&flag+4]
    // mov r2, #0
    // store r2, [&turn]
```

Does that guarantee mutual exclusion?

64

Thread A:

```
    mov r3, #1
    store r3, [&flag] // it's safe cuz flag is not shared
    mov r2, #1 // stopped here
    store r2, [&turn] // Now, turn = 1
LOOP:
    load r3, [&flag+4]
    cmp r3, #1
    bne CS
    load r2, [&turn]
    cmp r2, #1
    beq LOOP
// stopped inside LOOP, checking condition
CS:
    mov r2, #0
    store r2, [&flag]
    // mov r2, #1
    // store r2, [&turn]
```

Thread B:

```
    mov r3, #1
    store r3, [&flag+4] // it's safe cuz flag is not shared
    mov r2, #0
    store r2, [&turn] // Now, turn = 0
LOOP:
    load r3, [&flag]
    cmp r3, #1
    bne CS
    load r2, [&turn]
    cmp r2, #0
    beq LOOP
CS:
    // Now we are in the CS
    // stopped here
    // resume execution
    // exit CS
    mov r2, #0
    store r2, [&flag+4]
    // mov r2, #0
    // store r2, [&turn]
    // Now flag[1] changed
    // scheduled out
```


Does that guarantee mutual exclusion?

65

Thread A:

```
    mov r3, #1
    store r3, [&flag] // it's safe cuz flag is not shared
    mov r2, #1 // stopped here
    store r2, [&turn] // Now, turn = 1
// stopped inside LOOP, checking condition
// enter CS due to flag[1]
LOOP:
    load r3, [&flag+4]
    cmp r3, #1
    bne CS
    load r2, [&turn]
    cmp r2, #1
    beq LOOP
CS:
// Now, we enter CS
    mov r2, #0
    store r2, [&flag]
    // mov r2, #0
    // store r2, [&turn]
```

Thread B:

```
    mov r3, #1
    store r3, [&flag+4] // it's safe cuz flag is not shared
    mov r2, #0
    store r2, [&turn] // Now, turn = 0
LOOP:
    load r3, [&flag]
    cmp r3, #1
    bne CS
    load r2, [&turn]
    cmp r2, #0
    beq LOOP
CS:
    // exit CS
    // turn changed
    mov r2, #0
    store r2, [&flag+4]
    // mov r2, #0
    // store r2, [&turn]
```

Mutual exclusion algorithm

66

- Peterson's algorithm
 - work with two threads
 - variables: turn, flag[2] = {false, false}

P0:

flag[0] = true;

turn = 1;

while (flag[1] && turn == 1) ;

ENTER_CS

flag[0] = false;

P1:

flag[1] = true;

turn = 0;

while (flag[0] && turn == 0) ;

ENTER_CS

flag[1] = false;

Dekker's algorithm

67

- A more polite version
- If you cannot enter the CS, then put your flag down
 - then compete again!

```
P0:
flag[0] = true;
while (flag[1] == true) {
    if (turn != 0) {
        flag[0] = false;
        while (turn != 0) ;
        // wait until P1 completes job
        flag[0] = true;
        // set flag to show willingness
    }
}
ENTER_CS
turn = 1;
flag[0] = false;
```

```
P1:
flag[1] = true;
while (flag[0] == true) {
    if (turn != 1) {
        flag[1] = false;
        while (turn != 1) ;
        // wait until P0 completes job
        flag[1] = true;
        // set flag to show willingness
    }
}
ENTER_CS
turn = 0;
flag[1] = false;
```

For more threads, rather than two

68

- It is hard to design algorithms;
- What makes synchronization so difficult?
 - Update the variable; live within only register
- General solution for multi-process synchronization
 - Set the flag to enter CS
- Test flag should also be protected!
 - Test flag
 - If we can protect test flag, we may do this
 - Lock (test_flag)
 - Enter CS
 - Unlock (test_flag)

lock(test_flag)/unlock(test_flag)

69

- To ensure mutual exclusion, we must have some different approach
- The following code will not work!

```
while (test_flag) ;  
test_flag = 1;  
Enter CS  
test_flag = 0;
```

- LOOP:
 - Load r0, [test_flag]
 - Cmp r0, #0
 - Bne LOOP
 - Mov r1, #1
 - Store r1, [&test_flag]
- CS:
 - Mov r1, #0
 - Store r1, [&test_flag]

Does that guarantee mutual exclusion?

70

Thread A:

LOOP:

```
load r0, [&test_flag]
cmp r0, #0
bne LOOP
mov r1, #1
// schedule out!
store r1, [&test_flag]
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Thread B:

LOOP:

```
load r0, [&test_flag]
cmp r0, #0
bne LOOP
mov r1, #1
store r1, [&test_flag]
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

71

Thread A:

LOOP:

```
load r0, [&test_flag]
cmp r0, #0
bne LOOP
mov r1, #1
// stopped here
store r1, [&test_flag]
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Thread B:

LOOP:

```
load r0, [&test_flag]
cmp r0, #0
bne LOOP
mov r1, #1
store r1, [&test_flag]
```

CS:

```
// enter CS
// and scheduled out
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

72

Thread A:

LOOP:

load r0, [&test_flag]

cmp r0, #0

bne LOOP

mov r1, #1

// stopped here

store r1, [&test_flag]

CS:

// we also enter CS

// mutual exclusion is broken

mov r1, #0

store r1, [&test_flag]

Thread B:

LOOP:

load r0, [&test_flag]

cmp r0, #0

bne LOOP

mov r1, #1

store r1, [&test_flag]

CS:

// entered CS

// and stopped here

mov r1, #0

store r1, [&test_flag]

Architectural support for synchronization ⁷³

- Concurrent execution of multiple threads
- New CPUs have atomic instructions
 - test-and-set instruction
 - does all the following things at once (with a single instruction)
 - load value in the memory to register (test)
 - store value in the memory (set)
 - Test-and-set example
 - test-and-set reg, [&flag]
- Previous code would be like
 - LOOP:
 - test-and-set r0, [&test_flag]
 - cmp r0, #0
 - bne LOOP
 - CS:
 - Mov r1, #0
 - Store r1, [&test_flag]

Does that guarantee mutual exclusion?

74

Thread A:

LOOP:

```
test-and-set r0, [&test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
// enter CS
// scheduled out!
mov r1, #0
store r1, [&test_flag]
```

Thread B:

LOOP:

```
test-and-set r0, [&test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

75

Thread A:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
// entered CS
// stopped here
mov r1, #0
store r1, [&test_flag]
```

Thread B:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
// cannot enter CS
// because test_flag has been set by Process A
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

76

Thread A:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
// entered CS
// stopped here
// resume execution
mov r1, #0
// scheduled out!
store r1, [&test_flag]
```

Thread B:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
// cannot enter CS
// because test_flag has been set by Process A
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

77

Thread A:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
// entered CS
mov r1, #0
// stopped here
store r1, [&test_flag]
```

Thread B:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
// Still, cannot enter CS
// because test_flag has been changed, yet
// scheduled out!
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

78

Thread A:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
// entered CS
mov r1, #0
// stopped here
// resume execution
store r1, [&test_flag]
// now, we exit cs
```

...

```
// schedule out!
```

Thread B:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

// Still, cannot enter CS

// because test_flag has been changed, yet

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

79

Thread A:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
mov r1, #0
store r1, [&test_flag]
// now, we exit cs
```

...

```
// schedule out!
```

Thread B:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
// Still, cannot enter CS
// because test_flag has been changed, yet
// resume execution
```

CS:

```
// now, we can enter CS
// and exit CS
mov r1, #0
store r1, [&test_flag]
```

That is what we call spinlock

80

- Lock variable (flag)
 - usually, shared structure has lock member variable
- spinning for checking lock
- consumes CPU cycles!
 - *busy-waiting*

- Usage

```
lock(&shared_data.lock);  
CS  
unlock(&shared_data.lock);
```

- C implementation

```
void lock(lock_t * lock_var) {  
    while (test_and_set(lock_var)) ;  
}
```

```
void unlock(lock_t * lock_var) {  
    *lock_var = 0;  
}
```


Synchronization primitives

- Spinlocks & Mutex

81

- It is hard to design algorithm;
- What makes synchronization so difficult?
 - Update the variable; live within only register
- General solution for multi-process synchronization
 - Set the flag to enter CS
- Test flag should also be protected!
 - Test flag
 - If we can protect test flag, we may do this
 - Lock (test_flag)
 - Enter CS
 - Unlock (test_flag)

lock(test_flag)/unlock(test_flag)

83

- To ensure mutual exclusion, we must have some different approach
- The following code will not work!

```
while (test_flag) ;  
test_flag = 1;  
Enter CS  
test_flag = 0;
```

- LOOP:
 - Load r0, [test_flag]
 - Cmp r0, #0
 - Bne LOOP
 - Mov r1, #1
 - Store r1, [&test_flag]
- CS:
 - Mov r1, #0
 - Store r1, [&test_flag]

Does that guarantee mutual exclusion?

84

Process A:

LOOP:

load r0, [&test_flag]

cmp r0, #0

bne LOOP

mov r1, #1

// schedule out!

store r1, [&test_flag]

CS:

mov r1, #0

store r1, [&test_flag]

Process B:

LOOP:

load r0, [&test_flag]

cmp r0, #0

bne LOOP

mov r1, #1

store r1, [&test_flag]

CS:

mov r1, #0

store r1, [&test_flag]

Does that guarantee mutual exclusion?

85

Process A:

LOOP:

```
load r0, [&test_flag]
cmp r0, #0
bne LOOP
mov r1, #1
// stopped here
store r1, [&test_flag]
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Process B:

LOOP:

```
load r0, [&test_flag]
cmp r0, #0
bne LOOP
mov r1, #1
store r1, [&test_flag]
```

CS:

```
// enter CS
// and scheduled out
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

86

Process A:

LOOP:

```
load r0, [&test_flag]
cmp r0, #0
bne LOOP
mov r1, #1
// stopped here
store r1, [&test_flag]
```

CS:

```
// we also enter CS
// mutual exclusion is broken
mov r1, #0
store r1, [&test_flag]
```

Process B:

LOOP:

```
load r0, [&test_flag]
cmp r0, #0
bne LOOP
mov r1, #1
store r1, [&test_flag]
```

CS:

```
// entered CS
// and stopped here
mov r1, #0
store r1, [&test_flag]
```

Architectural support for synchronization ⁸⁷

- Concurrent execution of multiple threads
- New CPUs have atomic instructions
 - test-and-set instruction
 - does all the following things at once (with a single instruction)
 - load value in the memory to register (test)
 - store value in the memory (set)
 - Test-and-set example
 - test-and-set reg, [&flag]
- Previous code would be like
 - LOOP:
 - test-and-set r0, [&test_flag]
 - cmp r0, #0
 - bne LOOP
 - CS:
 - Mov r1, #0
 - Store r1, [&test_flag]

Does that guarantee mutual exclusion?

88

Process A:

LOOP:

test-and-set r0, [&test_flag]

cmp r0, #0

bne LOOP

CS:

// enter CS

// scheduled out!

mov r1, #0

store r1, [&test_flag]

Process B:

LOOP:

test-and-set r0, [&test_flag]

cmp r0, #0

bne LOOP

CS:

mov r1, #0

store r1, [&test_flag]

Does that guarantee mutual exclusion?

89

Process A:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
// entered CS
// stopped here
mov r1, #0
store r1, [&test_flag]
```

Process B:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
// cannot enter CS
// because test_flag has been set by Process A
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

90

Process A:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
// entered CS
// stopped here
// resume execution
mov r1, #0
// scheduled out!
store r1, [&test_flag]
```

Process B:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
// cannot enter CS
// because test_flag has been set by Process A
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

91

Process A:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
// entered CS
mov r1, #0
// stopped here
store r1, [&test_flag]
```

Process B:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
// Still, cannot enter CS
// because test_flag has been changed, yet
// scheduled out!
```

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

92

Process A:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
// entered CS
mov r1, #0
// stopped here
// resume execution
store r1, [&test_flag]
// now, we exit cs
```

...

```
// schedule out!
```

Process B:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

// Still, cannot enter CS

// because test_flag has been changed, yet

CS:

```
mov r1, #0
store r1, [&test_flag]
```

Does that guarantee mutual exclusion?

93

Process A:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
```

CS:

```
mov r1, #0
store r1, [&test_flag]
// now, we exit cs
```

...

```
// schedule out!
```

Process B:

LOOP:

```
load r0, [test_flag]
cmp r0, #0
bne LOOP
// Still, cannot enter CS
// because test_flag has been changed, yet
// resume execution
```

CS:

```
// now, we can enter CS
// and exit CS
mov r1, #0
store r1, [&test_flag]
```

That is what we call spinlock

94

- Lock variable (flag)
 - usually, shared structure has lock member variable
- spinning for checking lock
- consumes CPU cycles!
 - busy-waiting
- Usage

```
lock(&shared_data.lock);
CS
unlock(&shared_data.lock);
```

- C implementation

```
void lock(lock_t * lock_var) {
    while (test_and_set(lock_var)) ;
}

void unlock(lock_t * lock_var) {
    *lock_var = 0;
}
```

- Test-and-set requires some complex hardware operation
 - bus locking
 - esp. when multiple CPUs
 - synchronize updating memory from one CPU
- Code would be
 - OUT_LOOP:
 - test-and-set r0, [&test_flag]
 - cmp r0, #0
 - beq CS
 - IN_LOOP:
 - load r0, [&test_flag]
 - cmp r0, #0
 - beq OUT_LOOP
 - b IN_LOOP
 - CS:
 - mov r1, #0
 - Store r1, [&test_flag]

- C implementation

```
void lock(lock_t * lock_var) {  
    while (test_and_set(lock_var)) {  
        while (lock_var) ;  
    }  
}  
  
void unlock(lock_t * lock_var) {  
    *lock_var = 0;  
}
```

- Less use of test-and-set
 - esp. when Spinning inside loop
 - just looking at variable is enough

- In uniprocessor system, what if
 - you own a lock, then asleep?
 - No body who waits for the lock cannot make progress!
- What is good strategy?
 - When a process enters the critical section
 - or hold the lock
 - make the process run as quick as possible, so that
 - the process completes the job (or exit the critical section)
 - or release the lock
- the lock holder would disable the interrupts
 - when it holds the lock
 - No timer interrupts,
 - result in no context switch
 - the lock holder 'only' can make progress!
 - works only with uniprocessor system, search for why (homework)

A better idea! than spinlock

97

- why not sleep?
 - rather than checking the condition
- A big difference between sleep and spin
 - consume (waste) CPU time for checking until the condition meet
- For single CPU system, (uni-processor system)
 - If a process in spinlock loop, it will not be unlocked until
 - context switch happens, and lock holder process releases the lock
- So, make the process sleep!
 - and awake it when the lock is released!
 - OS knows all!

- OS services for process synchronization
 - guarantees mutual exclusive access to a resource
 - try lock,
 - if you get the lock, enter the critical section
 - if you cannot get the lock, sleep until another releases the lock
- Semaphore
 - P() / V() operations
 - P() for entering the critical section
 - V() for exiting the critical section
- Binary semaphore (mutex)
 - initial semaphore value = 1
 - P()
 - decrease semaphore value
 - enter CS when semaphore value > 0
 - V()
 - increase semaphore value

- POSIX is a standard programming interface in UNIX OS
 - pthread, posix thread library is one of the most popular thread library
 - When you compile the code, you need to link pthread library
- Use `-lpthread` or `-pthread` option
- `pthread_create` / `pthread_join`
 - Create a new thread / wait until the thread finishes
 - `pthread_mutex_lock(&lock)` / `pthread_mutex_unlock(&lock)`
 - Acquire / release the lock variable

- Wait on the condition
 - Wait until the condition becomes false
 - Enter the critical section, holding the lock
 - Exit the critical section, awakening another waiting thread
- `pthread_cond_wait(cv, lock) / pthread_cond_signal(cv)`
- `pthread_mutex_lock` should be called before wait
 - `pthread_cond_wait()` should be called in a loop

```
pthread_mutex_lock(lock);  
while (!cond) {  
    pthread_cond_wait(cv, lock);  
}  
// Enter CS  
pthread_mutex_unlock(lock);
```

- What if
 - There is no mutex lock variable in the condition variable?
 - There is no loop in the condition variable?
 - There is no mutex unlock?
- Who can wake up the sleeping thread?
 - When and how?

Some practical problems: producer-consumer with bounded buffer

102

Simple producer-consumer problem

103

- Producer - consumer threads
 - Shared buffer with lock
 - Producer thread
 - Read in data from input to local buffer
 - Put the data on to the shared buffer
 - Consumer thread
 - Get data from the shared buffer to local buffer
 - Write the data to output
- Mutex with shared buffer
 - Allows mutual exclusive access from producer or consumer
- What if?
 - The shared buffer is full, and the producer gets the lock?
 - The shared buffer is empty, and the consumer gets the lock?

- Producer - consumer threads
 - Shared buffer with lock, condition variable
 - Producer thread
 - Read in data from input to local buffer
 - Put the data on to the shared buffer
 - Consumer thread
 - Get data from the shared buffer to local buffer
 - Write the data to output
- Mutex with shared buffer
 - Allows mutual exclusive access from producer or consumer
- Check condition before enter the critical section
 - Check condition in a loop
 - Wake up the waiting threads, at completion time

What if?

The shared buffer is full, and the producer gets the lock?

The shared buffer is empty, and the consumer gets the lock?

- Produce an element & ship it in the buffer
- If the buffer is full, we cannot ship it
 - so, wait until the buffer is empty
 - wait until the condition is met! - condition variable
- mutex_lock (lock)
- while (!buffer_empty) {
 - cond_wait (cv, lock);
- }
- // in the CS
- mutex_unlock (lock);

- Put out an element from the buffer
- If the buffer is empty, we cannot get it
 - so, wait until the buffer is full
 - wait until the condition is met! - condition variable
- mutex_lock (c_lock)
- while (!buffer_full) {
 - cond_wait (cv2, c_lock);
- }
- // in the CS
- mutex_unlock(c_lock);
- cond_signal(cv);

single buffer case: single producer/consumer

107

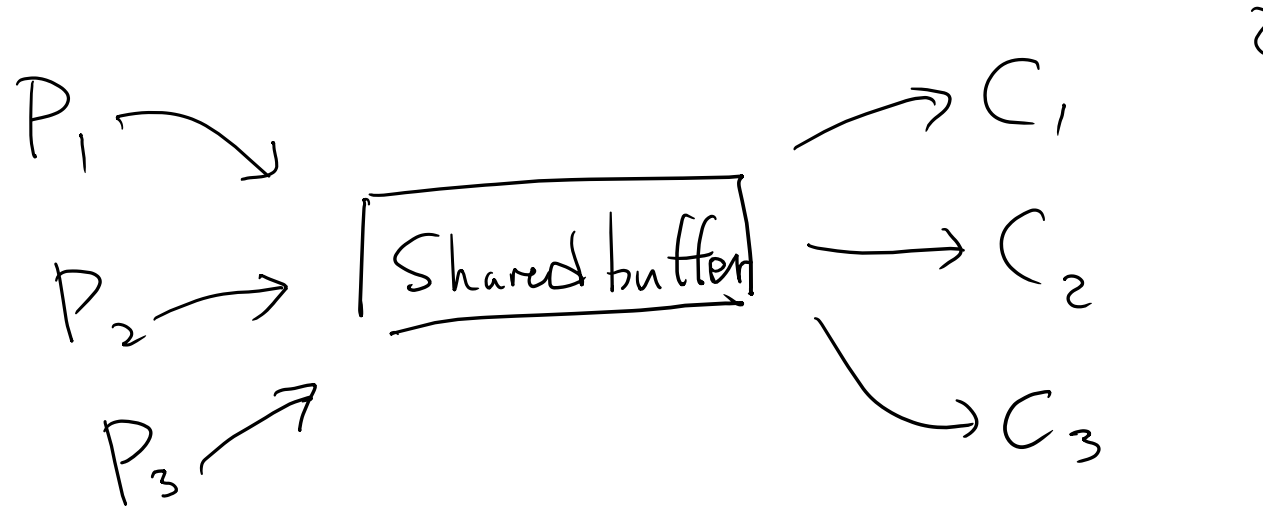
- single buffer, synchronized with mutex and condition variable
- two threads (P_1 , C_1)
 - one reads a line from a file, the other puts it in the buffer
- Design the (shared) buffer structure
 - pointer (and memory) to store the string (line)
 - buffer state (empty or full)
 - mutex lock for the buffer state change
 - condition variable for producer, consumer



What if multiple producer/consumer?

108

- If there's one shared buffer, one producer and one consumer can work
 - Even though multiple consumers exists, only one consumer can access the shared buffer

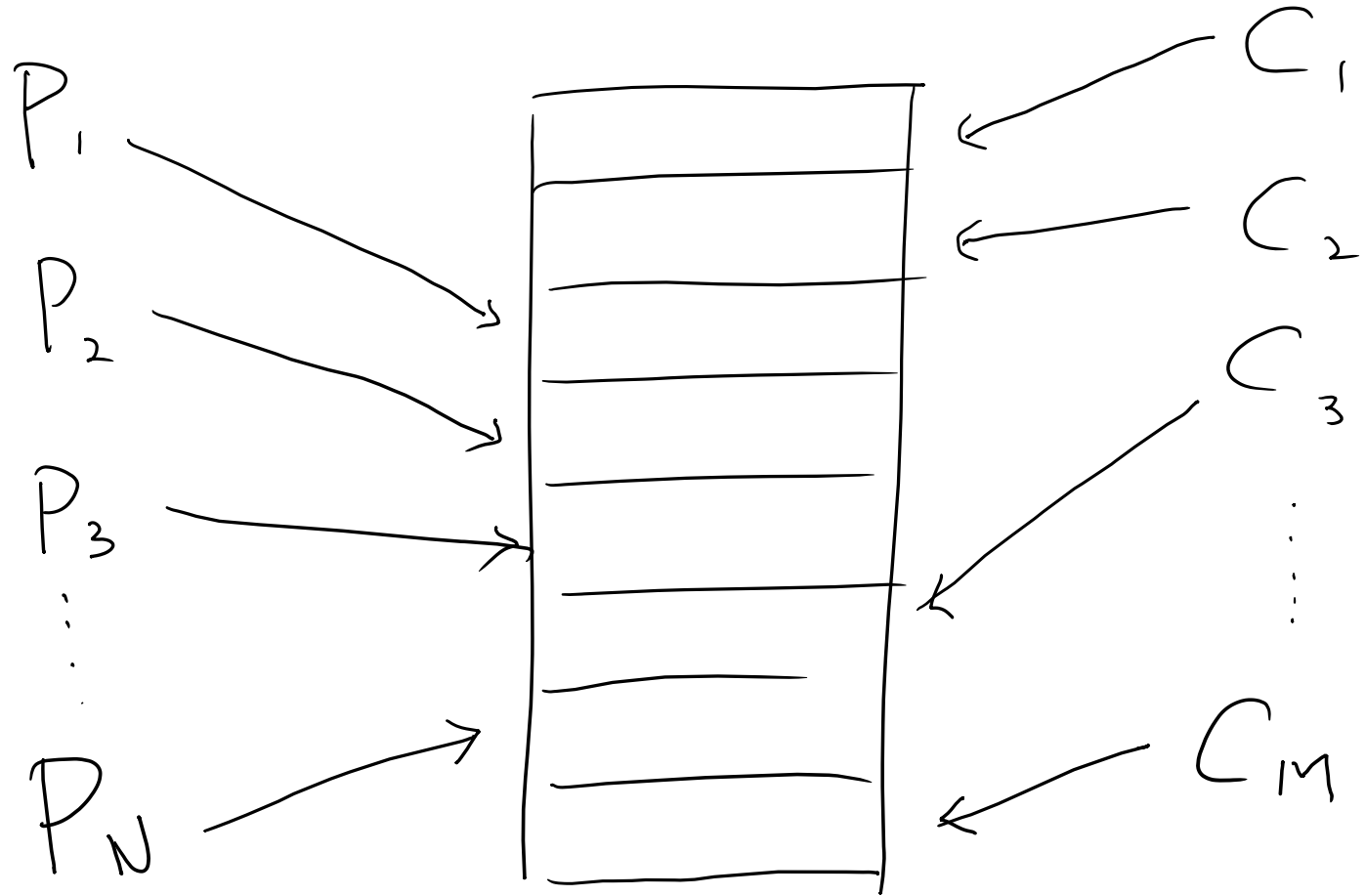


- Instead of using single shared buffer, use multiple buffers!

Extending shared buffer

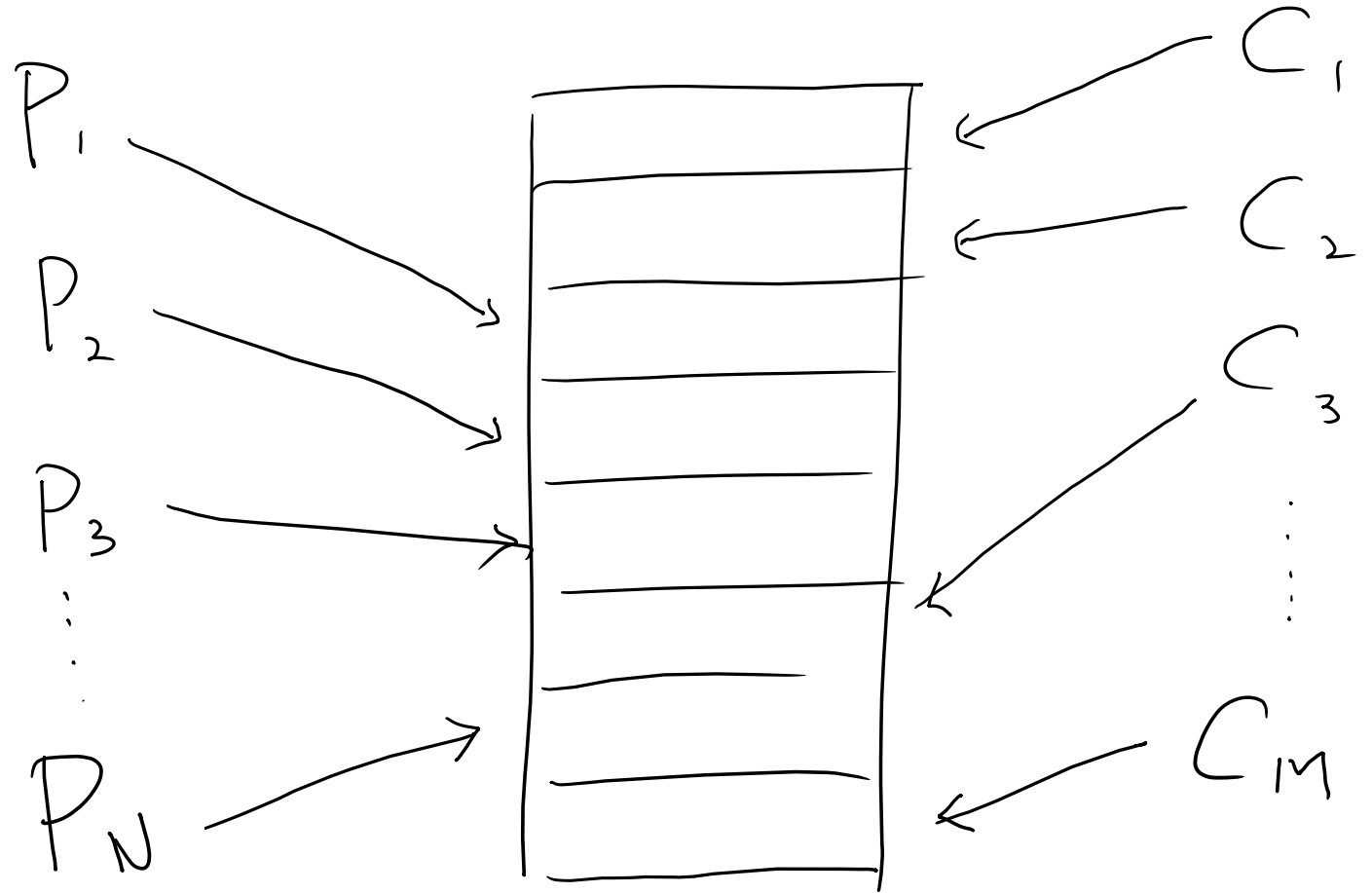
109

- Different threads can access to different buffers



- Requires synchronization for
 - Determine who will access which buffer
 - Check whether the buffer is empty, or full
- Goal of the problem
 - Process all the data, maximum throughput by concurrency
- Make some proper assumptions
 - Producer should be fast as much as consumer
(having similar number of producers and consumers)
- If the number of buffers is limited, it is called as bounded-buffer

- Producer thread
 - Get the buffer index to use
 - Check the buffer state
 - Put data on the buffer
 - Mark the buffer full
- Consumer thread
 - Get the buffer index to use
 - Check the buffer state
 - Get data from the buffer
 - Mark the buffer empty



- Multi-core concurrency strategy
 - To fully utilize the cpu cores
- To make N producers, N consumers work
 - proper data structure to work with more threads
 - if you have single big lock, you can work with single thread (no more than 2 threads)
 - fine-grained lock vs. coarse-grained lock
 - producers-consumers balance
 - if producers take too long and much time, and consumer takes too short-
 - N producers - N consumers would not make sense;
 - More producers, less consumers ? / less producers, more consumers?
 - more concurrency, arbitrary orders of execution
 - consumers execution may not be the same with index
 - per-byte overhead vs. per-transaction overhead

- producers: reading files (10us~1ms)
- consumers: printing out data / analysis for the buffer (0.x us)
- reuse buffer (to avoid allocation)
- increase the buffer size (to minimize per-transaction overhead)
- introduce buffer structure

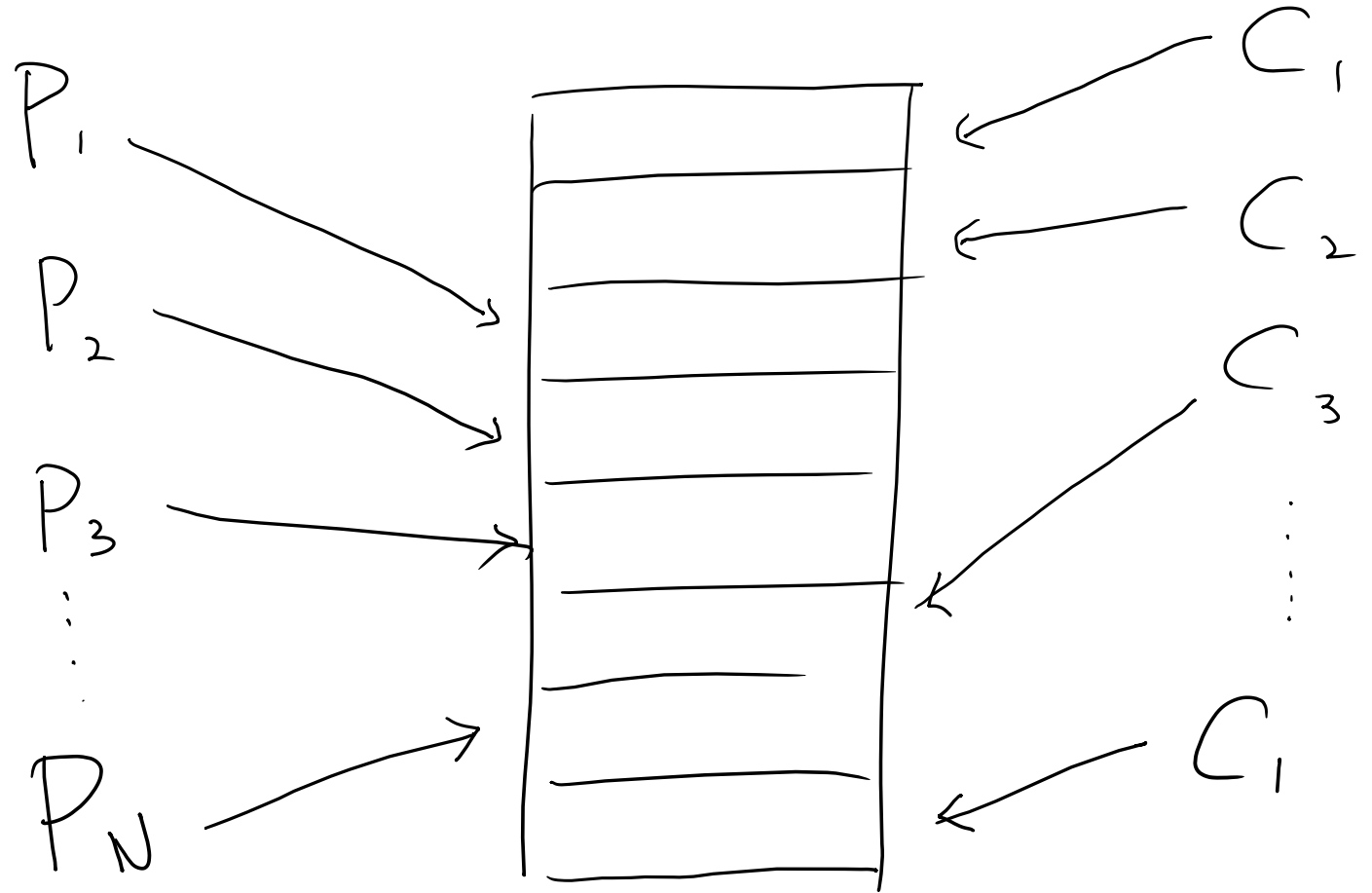
```
struct element {  
    byte buffer[4K];  
    int buffer_state; // fill or empty  
    mutex_lock lock;  
    cond_var cv;  
}
```

```
struct buffer {  
    struct element e[1024];  
    int p_index, c_index;  
    mutex_lock p_lock, c_lock;  
}
```

Again, N producers – N consumers

114

- Producer thread
 - Get the buffer index to use
 - Check the buffer state
 - Put data on the buffer
 - Mark the buffer full
- Consumer thread
 - Get the buffer index to use
 - Check the buffer state
 - Get data from the buffer
 - Mark the buffer empty



Mini-lab.

115