

# Welcome to OS @ UNSW

COMP3231/9201/3891/9283

(Extended) Operating Systems

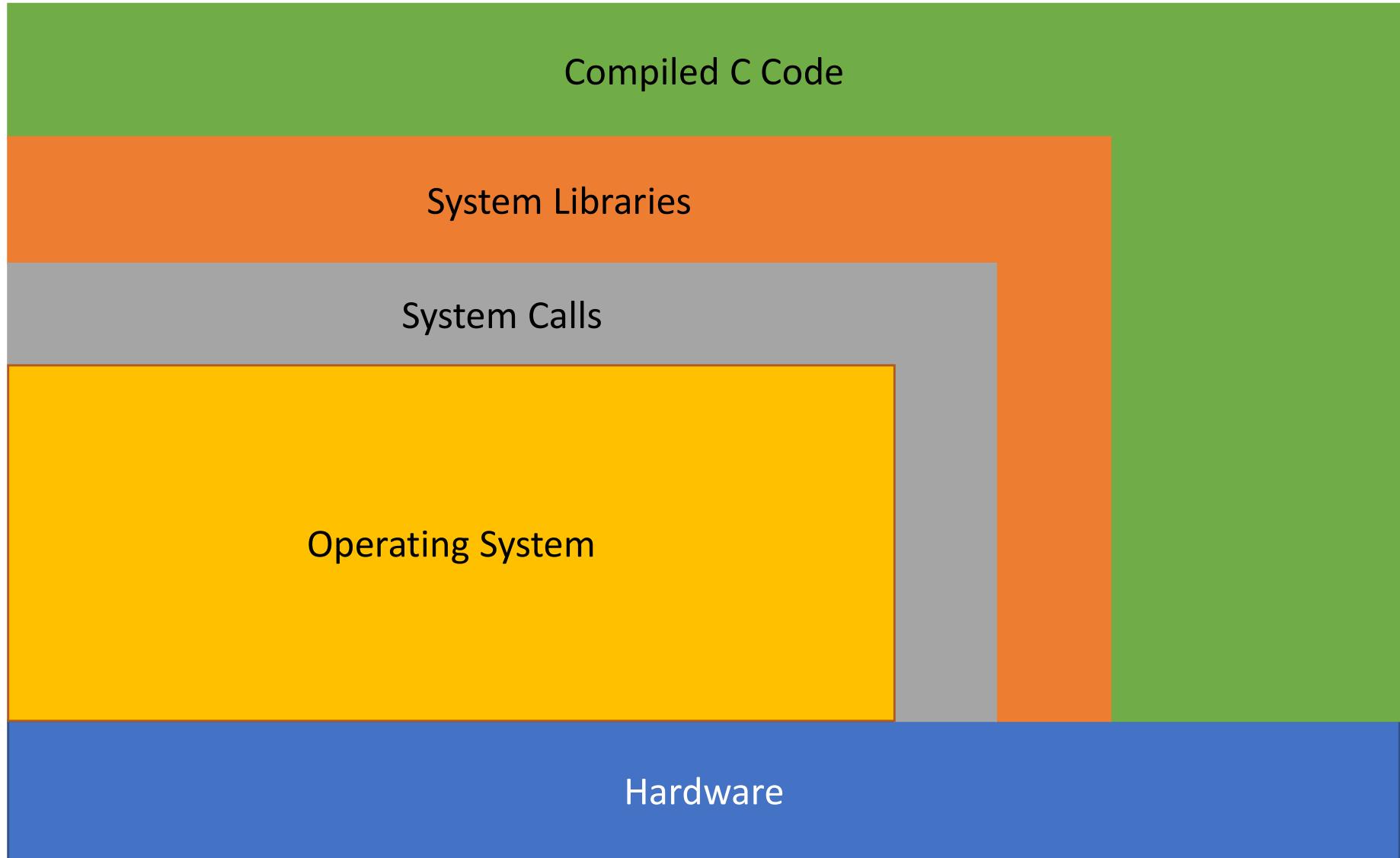
Dr. Kevin Elphinstone

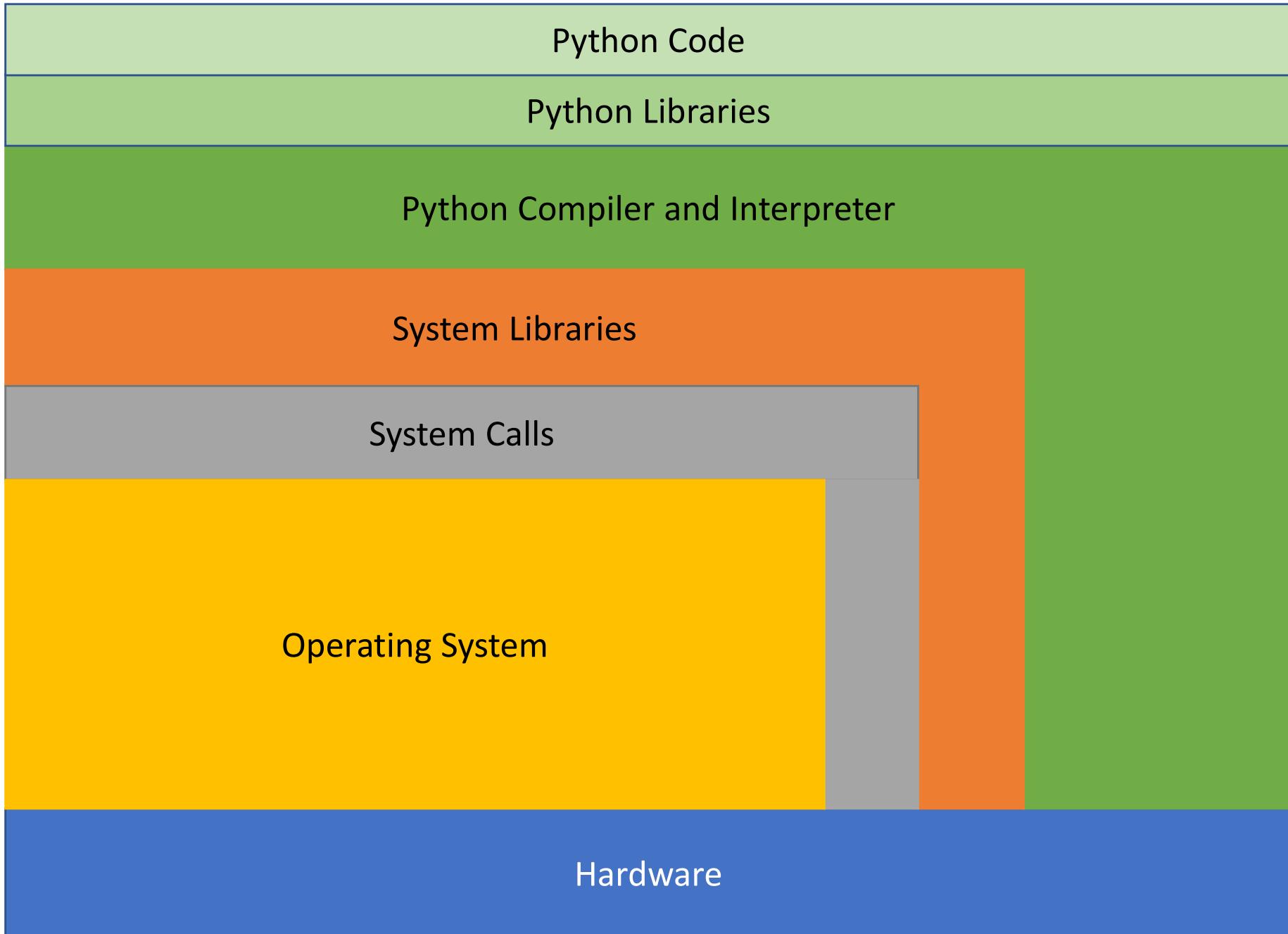


# Questions

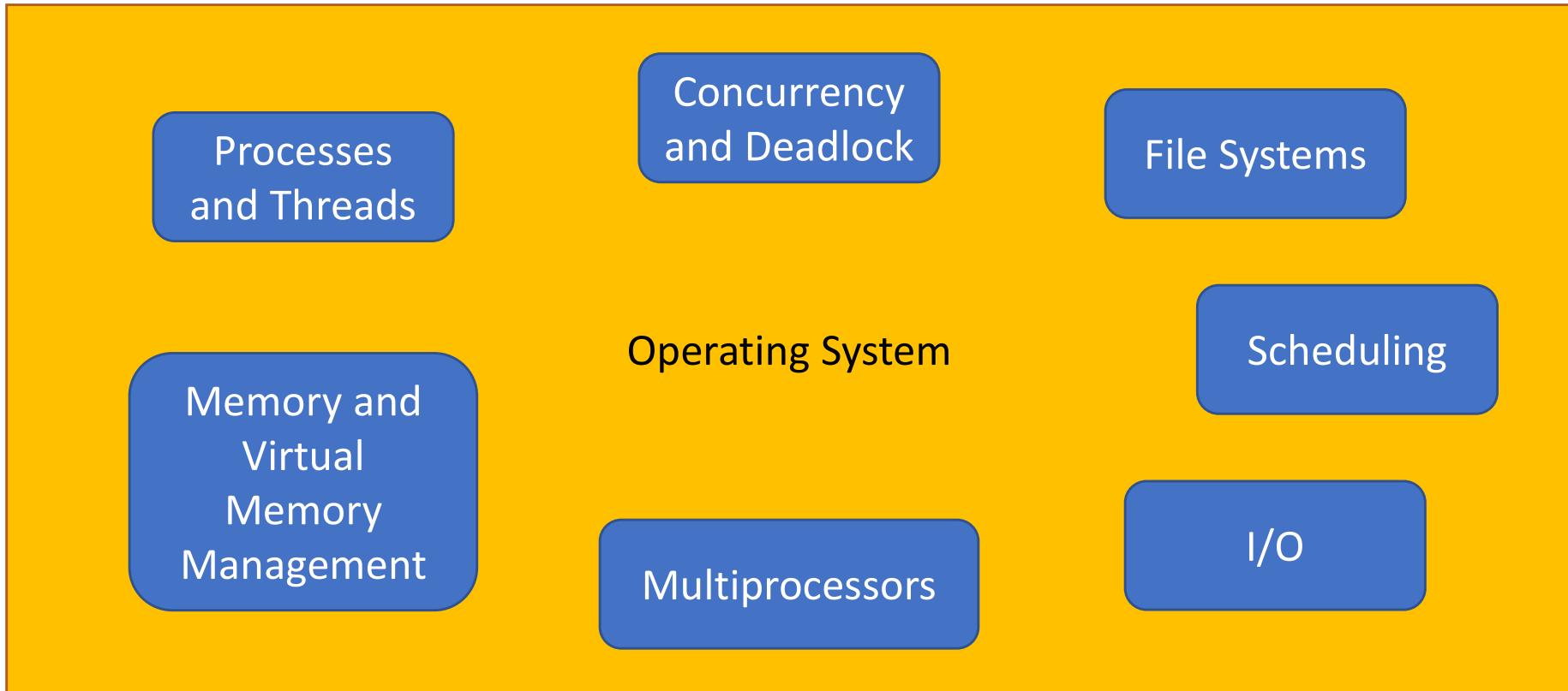
- Ask any questions you have in the course forum before the first lecture.
  - I'll answer either on the forum or in the first lecture.

# System Software Structure



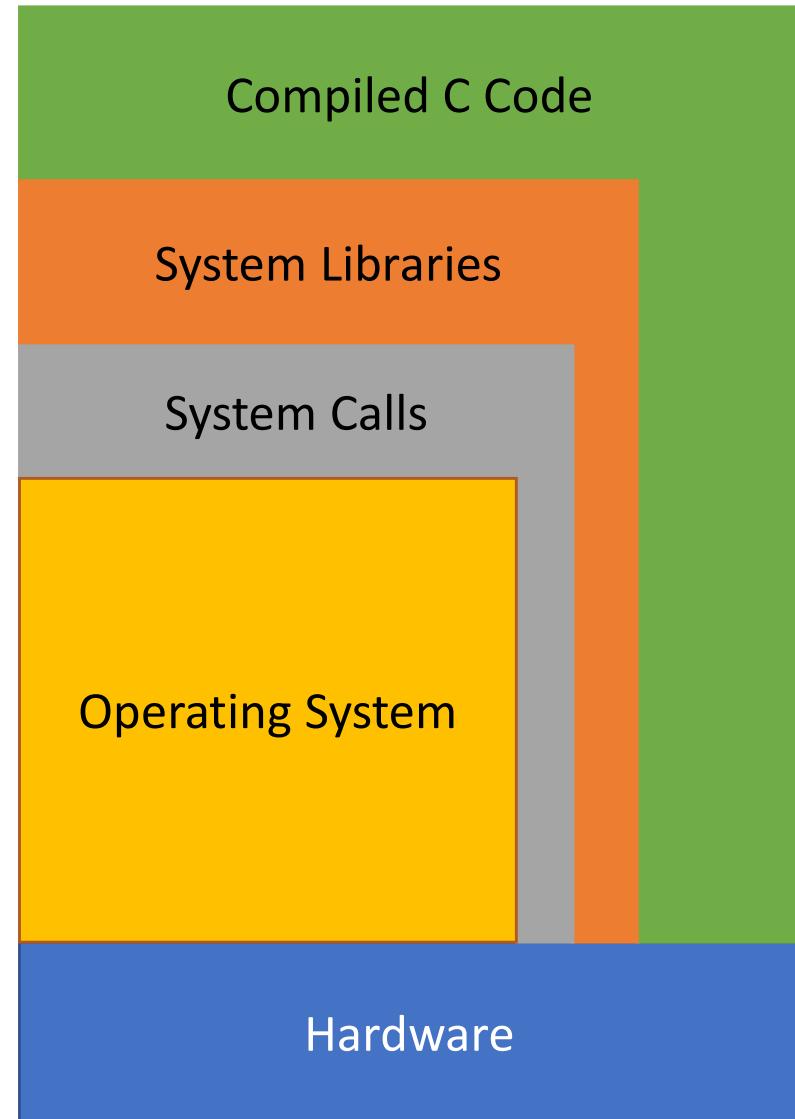


# Major OS Topics



# Why Learn Operating Systems?

- Understand the whole software stack
- Develop OS code
- Develop concurrent code
- Application performance
  - Understand operating system behaviour and how best to interface with it.
  - Diagnose system performance issues.



# How will we learn about Operating Systems?

## Lectures

- Introduce OS theory and case studies

## Tutorials

- Re-enforce theory
- Provide guidance on the assignments

## Assignments

- Opportunity to write real OS code
  - OS/161 is a simplified UNIX-clone intended for teaching
- Consist of the following
  - Warm-up exercise
  - Concurrency and synchronisation
  - OS Structure involving system calls and file system
  - Memory management



# Intended schedule\*

- Lectures
  - Weeks 1-5, 7-9
- Tutorials
  - Weeks 2-5, 7-10
- Assignments Due
  - ASST0 – Week 2
  - ASST1 – Week 4
  - ASST2 – Week 7
  - ASST3 – Week 10

\* Subject to change



# Overview of Course Outline

# Prerequisites

- Data structures and algorithms
  - COMP2521, COMP9024 or COMP1927
  - Stacks, queues, hash tables, lists, trees, heaps,....
- Computer systems
  - COMP1521, DPST1092, COMP2121, COMP9032 or ELEC2142
  - Computer systems architecture
  - Assembly programming
  - Mapping of high-level procedural language to assembly language
  - Interrupts

# Assumed Knowledge

- Computing Theory and Background
  - Basic computer architecture
    - CPUs, memory, buses, registers, machine instructions, interrupts/exceptions.
  - Common CS algorithms and data structures
    - Lists, arrays, hashing, trees, sorting, searching...
  - Ability to read assembly language
  - Exposure to programming using low-level systems calls (e.g. reading and writing files)
- Practical computing background
  - Capable UNIX command line users
  - Familiar with the git revision control system
  - Competent C programmers
    - Understand pointers, pointer arithmetic, function pointers, memory allocation (`malloc()`)
    - The dominant language for OS (and embedded systems) implementation.
  - Comfortable navigating around a large-ish existing code base.
  - Able to debug an implementation.

# Why does this fail?

Operating System Coding

```
void set(int *x)
{
    *x = 1;

}

void thingy()
{
    int *a;
    set(a);
    printf("%d\n", *a);
}
```



# Why does this fail?

```
void set(int *x)
{
    *x = 1;

}

void thingy()
{
    int a;
    set(&a);
    printf("%d\n",a);
}
```

# Lectures

- Common for all courses (3231/3891/9201/9283)
- 2 \* 2 hrs each week
- The lecture notes will be available on the course web site
  - <http://www.cse.unsw.edu.au/~cs3231>
    - Available prior to lectures, when possible.
    - Slide numbers for note taking, when not.
- Lectures will be a mix of live streaming and pre-recorded
  - Will announce in advance
  - Video will be available afterwards in both cases

**Administration**

- Course Outline
- UNSW Timetable
- Consultations
- Group Nomination
- Survey Results!!

**Work**

- Lectures
- Tutorials
- Extended Lectures

**Support**

- Ed Forums
- Wiki

**Assignments**

- Submission Guide
- Assignment 0 Warm-up
- Assignment 1
- Assignment 2
- Assignment 3

**Resources**

- OS/161
- General
- Man Pages
- Sys161 Pages

**C coding**

- Info Sheet

**Debugging**

- Learn Debugging

## Lectures

The lecturer reserves the right to make changes to this schedule, so check it occasionally to see if there have been changes. The most likely changes are extra details on lecture content and references to the text. Click on the topic name to get one slide per page, and the print version to get 6 slides per page.

Week	Topic	Book Ref	Print Format	Video
1	<a href="#">Course Introduction</a> <a href="#">Operating Systems Overview</a>	1		
2	<a href="#">Processes And Threads</a> <a href="#">Concurrency and Synchronisation</a> <a href="#">Deadlock</a>	2-2.2 2.3-2.3.7,2.5 6 - 6.7	  	  
3	<a href="#">Process and Thread Implementation</a> <a href="#">System Calls and R3000 Overview</a>	2.2 - 2.2.5 1.6	 	 
4	<a href="#">Computer Hardware, Memory Hierarchy, and Caching</a> <a href="#">File Management</a> <a href="#">File Management Part 2</a>	1.3 4 4	  	  
5	<a href="#">File Management (continued)</a> <a href="#">Case Study: Ext2</a> <a href="#">Case study: Ext3</a>		  	  
6	<a href="#">Memory Management</a> <a href="#">ASST2 Overview Video</a> Flexibility Week	3	 	 

# Extended OS Comp3891/9283

Starts in week 1

- A combination of:
  - Examination of topics in more depth
  - Looking at research in areas (past/present)
  - OS/161 internals in more depth
- Separate Assessment
  - 80%-ish of final exam common with base course
  - 20%-ish targeted to extended students
  - ~~Advanced assignment components part of the assessment~~
- Assumes the tutorials are not challenging enough
  - Effectively replaces the tutorial with extra interactive lecture.

# Tutorials

- Start in week 2
- A mix of online and f2f
  - Depends on tutorial you enrolled in
- Attendance is strongly recommended
  - but not marked.
- Tutorial questions cover a broad range of examples
  - Answers available online the week after.
  - Use the tutorial to focus where needed
    - There is intentionally more questions than can be covered
    - Review the questions beforehand

# Assignments

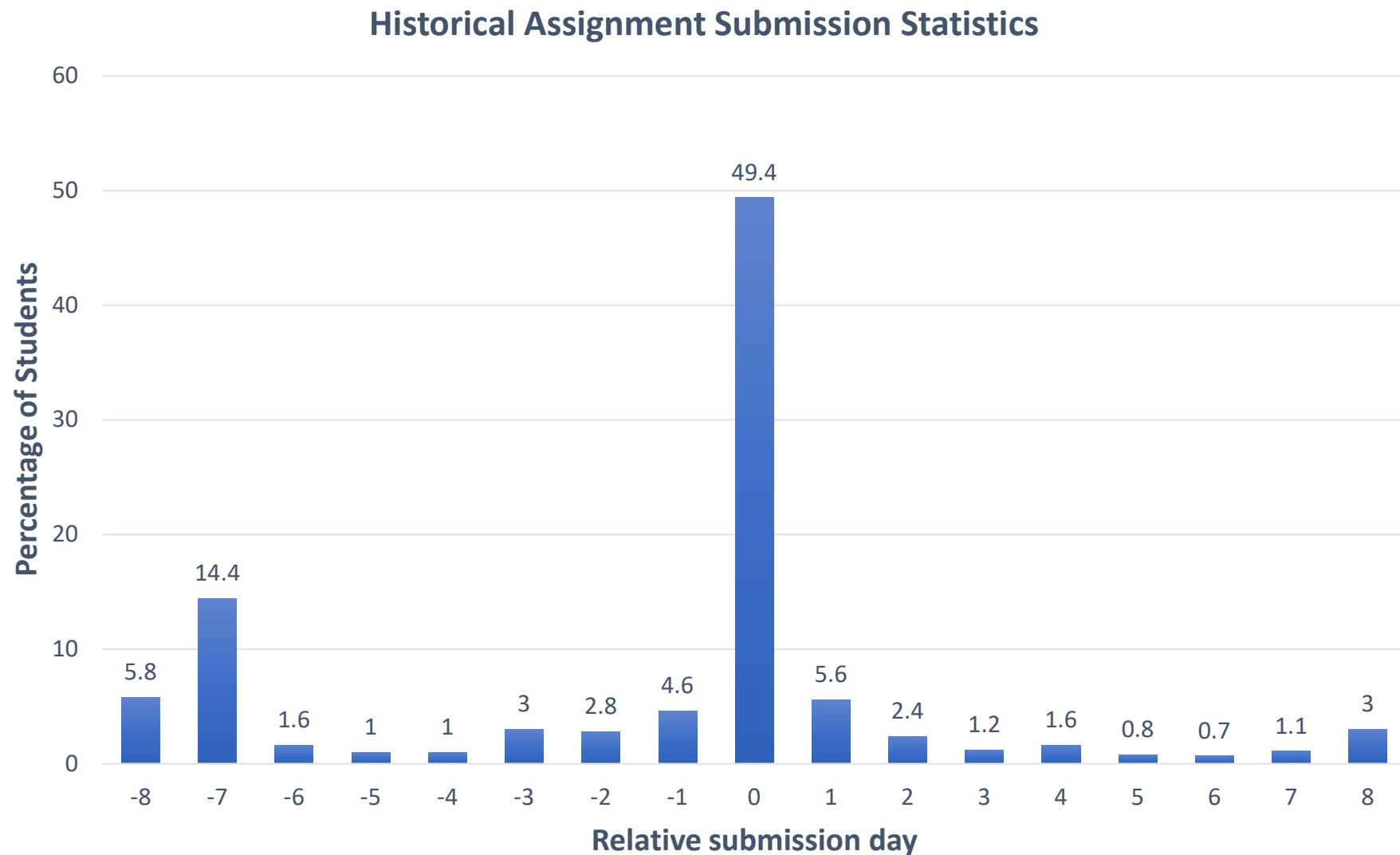
- Assignments form a substantial component of your assessment.
- They are challenging!!!!
  - Because operating systems are challenging
- We will be using OS/161,
  - an educational operating system
  - developed by the [Systems Group At Harvard](#)
    - With local changes.
  - It contains roughly 20,000 lines of code and comments
    - Comments are part of the documentation

# Assignments

- Don't underestimate the time needed to do the assignments.
  - 80% is understanding
  - 20% programming
- Avoid
  - 1% understanding
  - 9% programming
  - 90% debugging
- If you start a couple days before they are due, you will be late.
- To encourage you to start early,
  - Bonus 2% of awarded mark per day early, capped at 10%
  - See course outline for exact details
    - Read the fine print!!!!

# Assignment Submission Times

16% late



# Assignments

- Late penalty
  - 4% of total assignment value per day
    - Assignment is worth 20%
    - You get 18, and are 2 days late
    - Final mark =  $18 - (20 * 0.04 * 2) = 16$  (16.4)
  - Assignments are only accepted up to one week late. >5 days = 0

# Assignments

- Warmup assignment (ASST0)
  - Done individually
  - Available NOW!!!!
- ASST2 and ASST3 are in pairs
  - Info on how to pair up available soon
- Additionally, advanced versions of the assignment 2 & 3
  - Available bonus marks are small compared to amount of effort required.
  - Student should do it for the challenge, not the marks.
  - Attempting the advanced component is not a valid excuse for failure to complete the normal component of the assignment

Assignment	Due
ASST0	Week 2
ASST1	Week 4
ASST2	Week 7
ASST3	Week 10

# Assignment 0

- Warm-up exercise due in week 2
  - It's a warm-up to have you familiarize yourself with the environment and easy marks.
    - Practice with git revision control
    - Practice submitting a solution
    - Practice using code browser/editor
  - Do not use it as a gauge for judging the difficulty of the following assignments.

# Assignments

Submission test failed. Continue with submission (y/n)? y

- Lazy/careless submitter penalty: 15%
- Submitted the wrong assignment version penalty: 15%
  - Assuming we can validly date the intended version

# Assignments

- To help you with the assignments
  - We dedicate a tutorial per-assignment to discuss issues related to the assignment
  - Prepare for them!!!!

# Group Work Policy

- Groups of two
- Group members do not have to be in the same tutorial
- Group assignments will be marked as a group
  - Including ‘groups’ of one.
- Group members are expected to contribute equally to each assignment.
  - No “I’ll do the 2<sup>nd</sup> if you do the 3<sup>rd</sup> assignment”
  - We accept statements of unequal contributions and do adjust marks of the lesser contributor down.
- Submissions are required to have significant contributions attributable to individual group members.
  - E.g. verifiable using the git revision control system

# Plagiarism

- We take cheating seriously!!!
- We systematically check for plagiarised code
  - Penalties are generally enough to make it difficult to pass
- We can google as easy as you can
  - Some solutions are wrong
  - Some are greater scope than required at UNSW
    - You do more than required
    - Makes your assignment stick out as a potential plagiarism case
  - We do vary UNSW requirements

# Exams

- There is NO mid-session
- The final written exam is 2 hours
- Supplementary exam are available according to UNSW & school policy, not as a second chance.
  - Medical or other special consideration only

# Assessment\*

- Exam Mark Component
  - Max mark of 100
- Based solely on the final exam
- Class Mark Component
  - Max mark of 100
- 100% Assignments

\* Course outline is authoritative.

# Assessment

- The final assessment is a weighted geometric mean of 60% exam ( $E$ ) and 40% class ( $C$ ) component.

$$M = e^{\frac{60 \ln E + 40 \ln C}{100}}$$

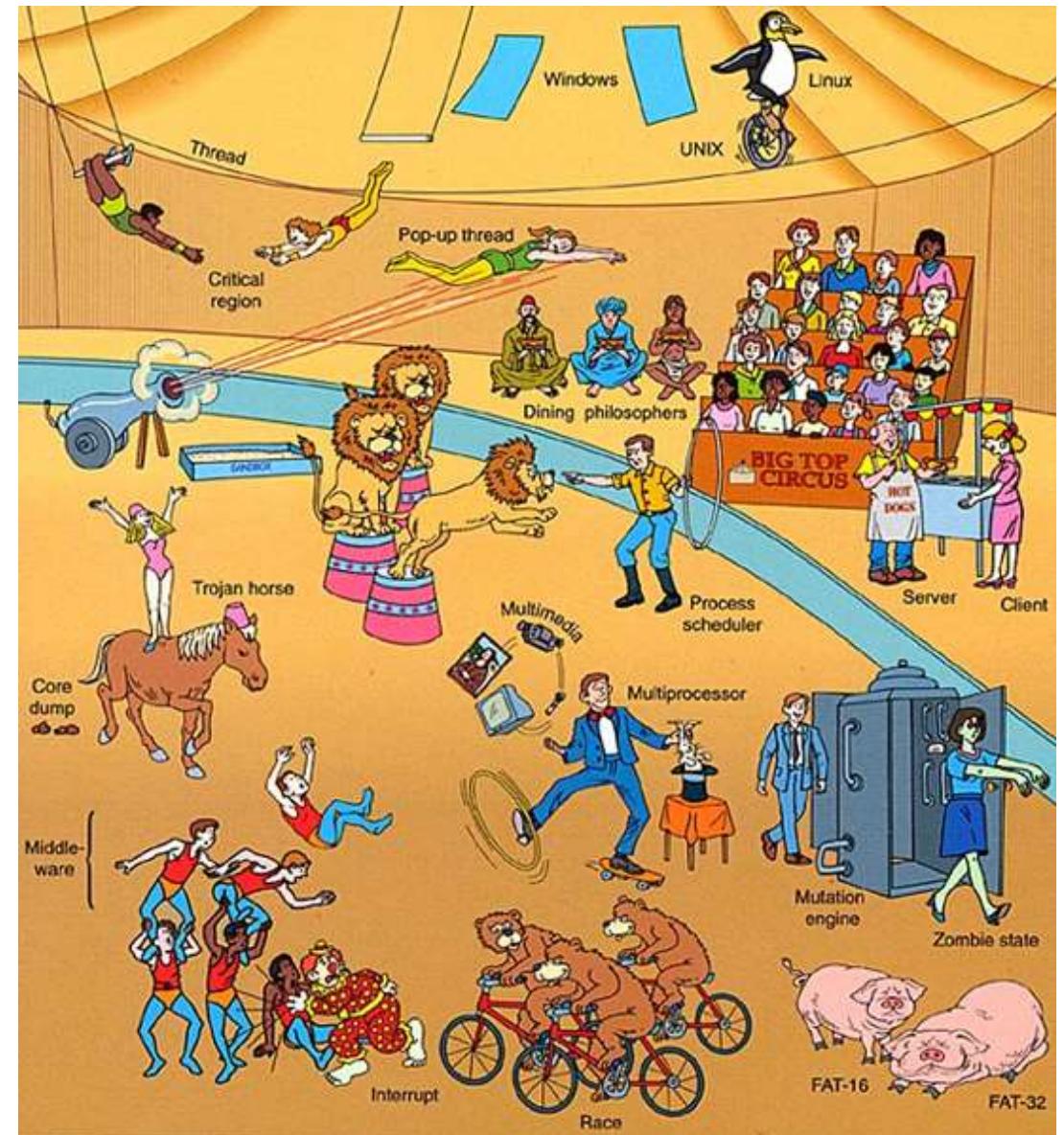
- Additionally, minimum of 40 required in exam ( $E$ ) and class ( $C$ ) components to pass.

# Assessment

- You need to perform reasonably consistently in both exam and class components.
- Geometric mean only has significant effect with significant variation.
- Reserve the right to moderate marks, and moderate courses individually if required.
  - Warning: We have moderated marks only once in the past

# Textbook

- Andrew Tanenbaum, *Modern Operating Systems*, 3<sup>rd</sup>/4<sup>th</sup> Edition, Prentice Hall



# References

- A. Silberschatz and P.B. Galvin, *Operating System Concepts*, 5<sup>th</sup>, 6<sup>th</sup>, or 7<sup>th</sup> edition, Addison Wesley
- William Stallings, *Operating Systems: Internals and Design Principles*, 4th or 5<sup>th</sup> edition, Prentice Hall.
- A. Tannenbaum, A. Woodhull, *Operating Systems--Design and Implementation*, 2<sup>nd</sup> edition Prentice Hall
- John O'Gorman, *Operating Systems*, MacMillan, 2000
- Uresh Vahalla, *UNIX Internals: The New Frontiers*, Prentice Hall, 1996
- McKusick et al., *The Design and Implementation of the 4.4 BSD Operating System*, Addison Wesley, 1996

# Ed Forums

- Where announcements are posted!!
- Forum for Q/A about assignments and course
  - Ask questions there for the benefit of everybody
  - Share your knowledge for the benefit of your peers
  - Look there before asking
- <https://edstem.org/>
  - Longer link on class web page
    - You will have received an invite from them to your UNSW email address.
      - z8888888@unsw.edu.au
    - You need to join to follow the course.

# Piazza Etiquette

## Search first!

- You are probably not the first to experience the problem, so see if the question is answered before asking again.

## Add to an existing post if directly related

- If you are experiencing a variant of the same issue, add to an existing post.

## Start a new post for a separate issue

- Try to have an accurate title
- Avoid adding an unrelated question to a hot topic because you just happen to be there when you had the thought. It makes it hard to find for others.

## Avoid bitmaps (screenshots)

- Bitmaps are not searchable so you limit the chances of fellow students finding your post, and indirectly make us less enthusiastic about providing a detailed answer to your non-searchable post.

## Provide some context

- Cut-n-paste the error if appropriate, and include the preceding output to provide a chance for others to understand what is going on. Mention the OS/machine/environment you're using if it's not clear from the cut-n-paste.

## Mark questions resolved if they are!

- Don't leave follow-ups unresolved if you have fixed your issue.

## Leave questions unresolved if they are!

- I filter using 'unresolved' to find outstanding issues, I won't find them unless they are marked unresolved.

## You're very welcome to post if you know the answer to an issue.

- The course staff do not have a monopoly on answers, nor do we monitor the forum 24hrs a day. A quick answer can make somebody's day (or at least avoid wasting it). A responsive forum can be an awesome resource for the entire course.

# Enforcing standards

- Don't be offended if we reject your post
  - Simply post again following the guidelines

## A good example

Hi, been trying to diagnose this for a while. Basically our program fails with:

```
panic: Fatal exception 2 (TLB miss on load) in kernel mode
panic: EPC 0x80020984, exception vaddr 0x0
```

Using GDB I backtraced to this call for copyout

```
#3 0x80020984 in copyout (src=0x0,
    userdest=0x0, len=0)
```

## A bad example

Unable to access the full range of the page table, when initialising all values of the page table to NULL, I am unable to access the whole page table.

Here's how i accessed it:

```
void init_pt(paddr_t ***pt) {
    for (paddr_t i = 0; i <= 255; i++) {
        for (paddr_t j = 0; j <= 63; j++) {
            kprintf("[started init_pt[%d][%d]]\n", i, j);
            pt[i][j] = NULL;
            kprintf("[finished init_pt[%d][%d]]\n", i, j);
        }
        pt[i] = NULL;
    }
}
```

# Consultations/Questions

- Questions should be directed to the forum.
- Admin and Personal queries can be directed to the class account [cs3231@cse.unsw.edu.au](mailto:cs3231@cse.unsw.edu.au)
  - Don't post private threads in Ed
- We reserve the right to ignore email sent directly to us (including tutors) if it should have been directed to the forum.
- Consultation Times
  - See course web site.
  - Must email (cs3231@cse) at least an hour in advance and show up on time.
    - If we get at least one email, we'll run the consult.

# What next?

<https://wiki.cse.unsw.edu.au/cs3231cgi/Checklist>

The screenshot shows a web browser window for the UNSW Sydney Wiki. At the top left is the UNSW crest with the motto "MANU ET MENTE". To its right is the large "UNSW SYDNEY" logo. In the top right corner, there are links for "KevinElphinstone", "Settings", and "Logout". Below these links, the word "Checklist" is displayed twice: once in blue text and once in a larger, bold black font. A horizontal menu bar below the logo includes links for "FrontPage", "RecentChanges", "FindPage", "HelpContents", and "Checklist" (which is highlighted in blue). Another menu bar at the bottom of the page includes "Edit (Text)", "Edit (GUI)", "Info", "Unsubscribe", "Add Link", "Attachments", and "More Actions: ▾".

## Startup Checklist

- Watch the online intro lecture.
  - Bring any questions to the first lecture.
- Join Piazza (you should have received an invite sent to [zID@unsw.edu.au](mailto:zID@unsw.edu.au))
- Review assignment 0
- Choose where you plan to do your assignment work (desktop, laptop, and at CSE).
  - Make sure the toolchain works on where you plan to work (see [Setup Overview](#))
- Set up git (see [Setup Overview](#))
- Choose an editor capable of code browsing (see [Setup Overview](#)).
- Complete ASST0

# Welcome to OS @ UNSW

COMP3231/9201/3891/9283

(Extended) Operating Systems

Dr. Kevin Elphinstone

# Operating Systems

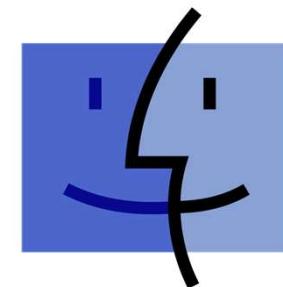
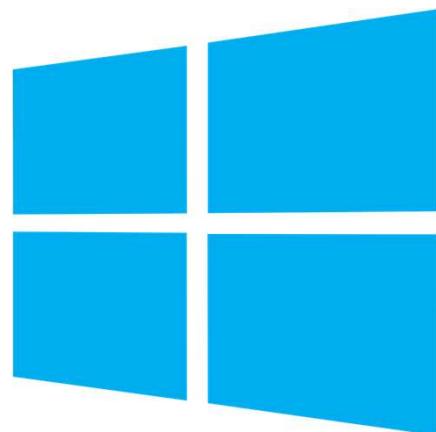
Chapter 1 – 1.3

Chapter 1.5 – 1.9

# Learning Outcomes

- High-level understand what is an operating system and the role it plays
- A high-level understanding of the structure of operating systems, applications, and the relationship between them.

# What is an Operating System?



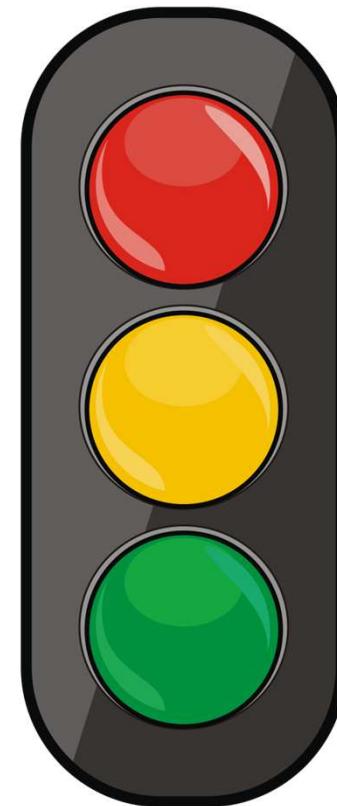
Mac OS

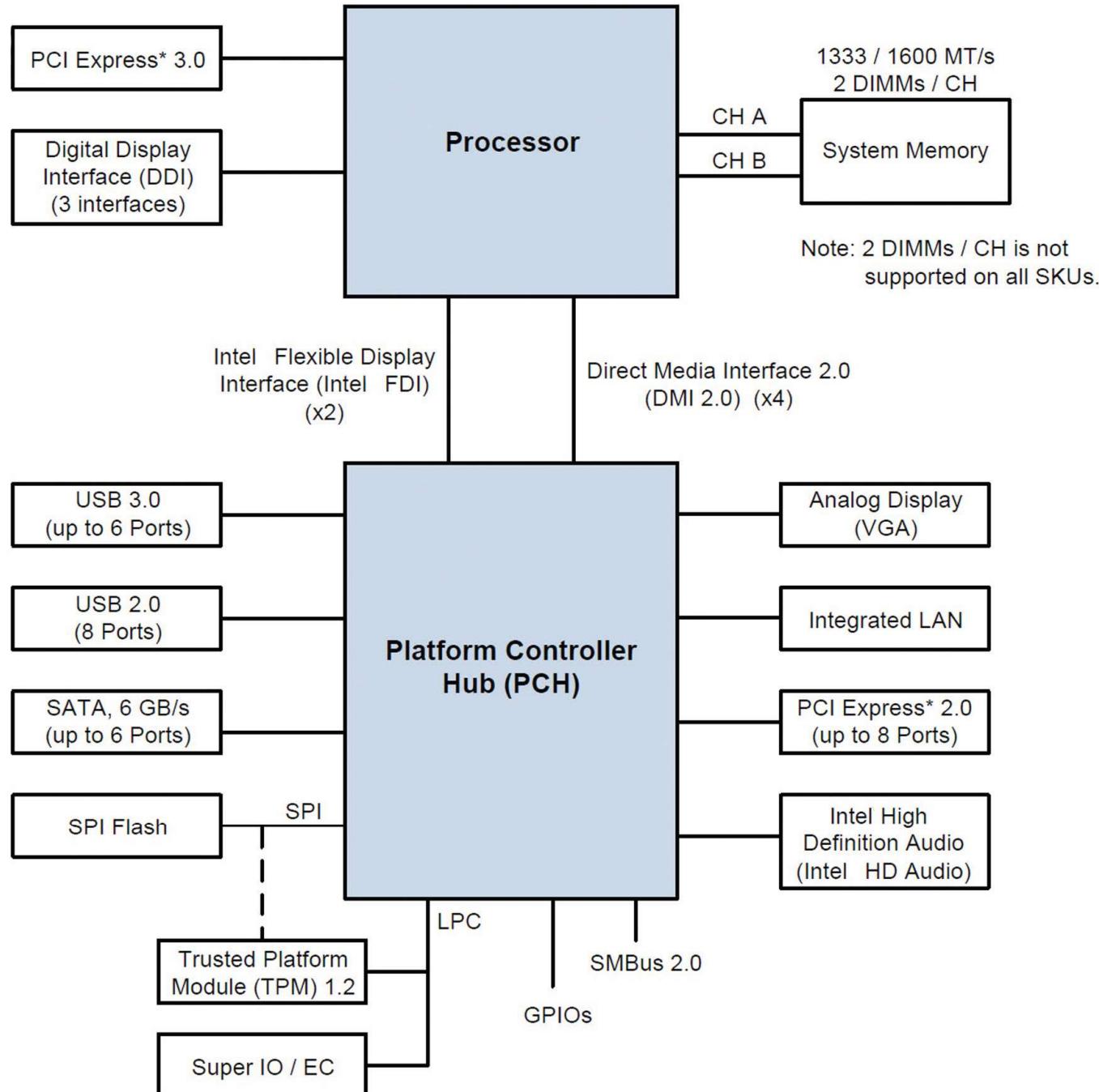


UNSW  
SYDNEY

# What is a traffic light?

- A signalling device that controls the flow of traffic
  - Defined in terms of the **role** it plays
- A signalling device consisting of three lights mounted at an intersection
  - Defined in terms of what it **is**





# *Role 1: The Operating System is an Abstract Machine*

- Extends the basic hardware with added functionality
- Provides high-level abstractions
  - More programmer friendly
  - Common core for all applications
    - E.g. Filesystem instead of just registers on a disk controller
- It hides the details of the hardware
  - Makes application code portable

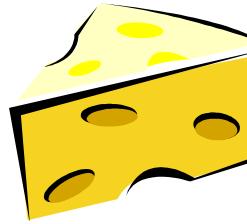
Disk



Memory

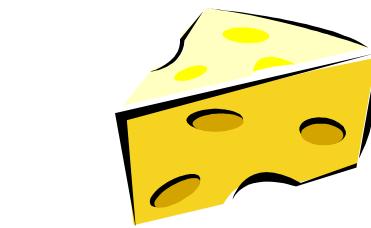


CPU



Network

Bandwidth



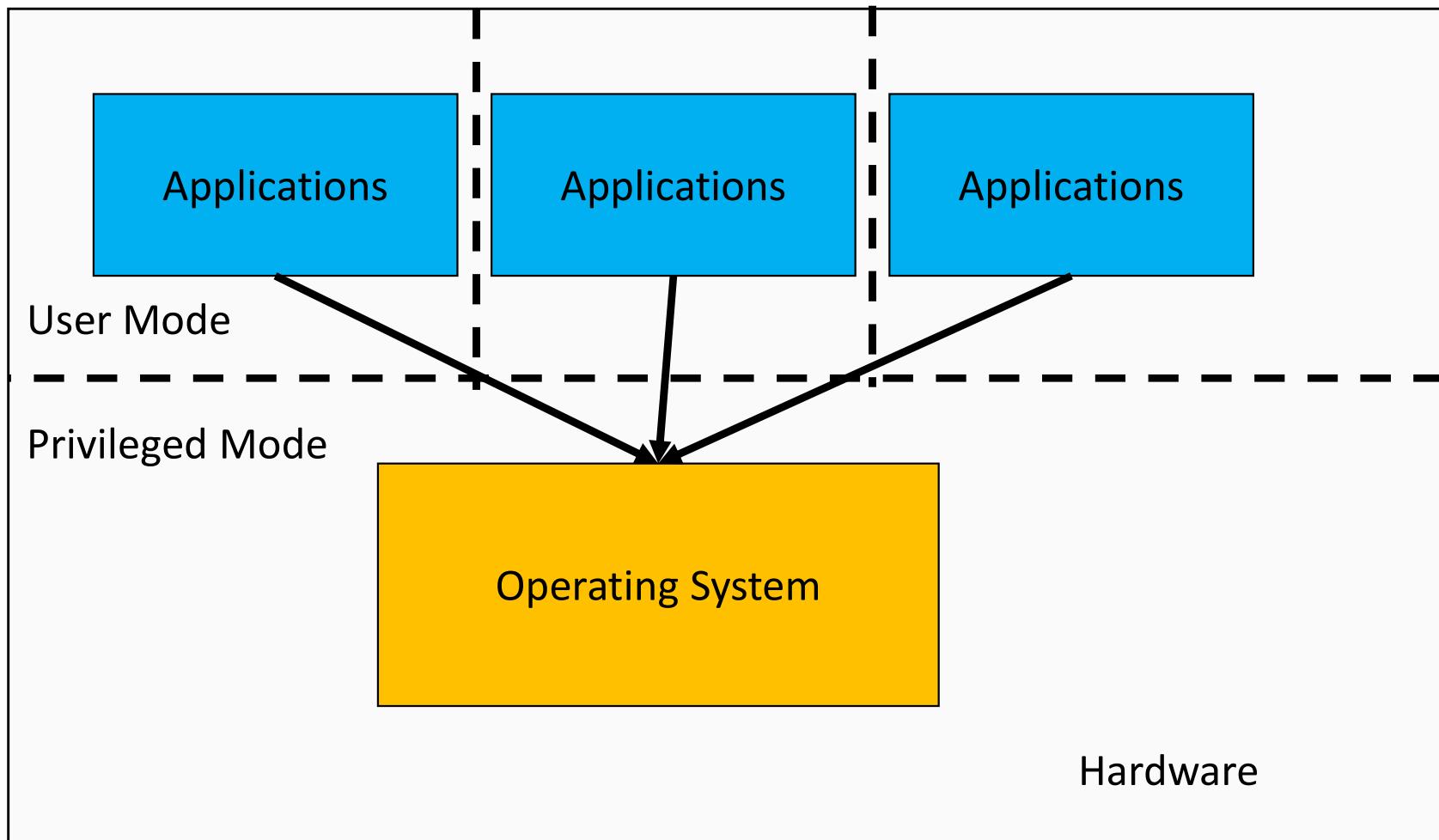
Users



# *Role 2: The Operating System is a Resource Manager*

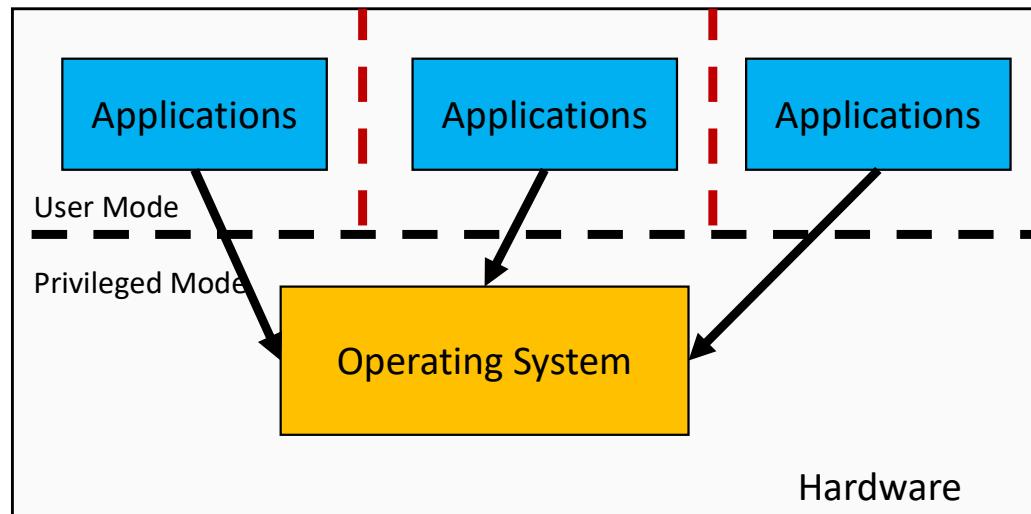
- Responsible for allocating resources to users and processes
- Must ensure
  - No Starvation
  - Progress
  - Allocation is according to some desired policy
    - First-come, first-served; Fair share; Weighted fair share; limits (quotas), etc...
  - Overall, that the system is efficiently used

Structural (Implementation) View: the Operating System *is* the software in *Privileged mode*.



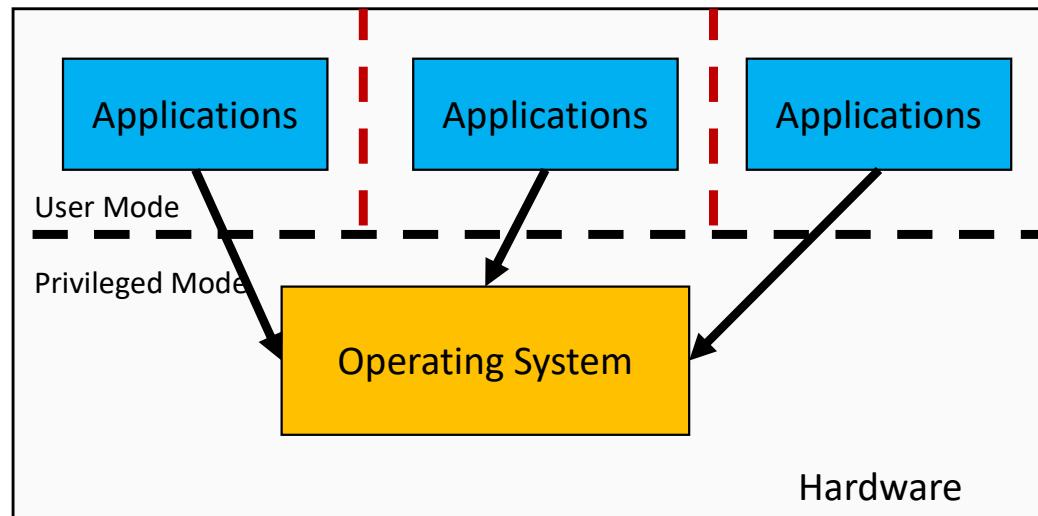
# Operating System Kernel

- Portion of the operating system that is running in *privileged mode*
- Contains fundamental functionality
  - Whatever is required to implement other services
  - Whatever is required to provide security
- Contains most-frequently used functions
- Also called the nucleus or supervisor

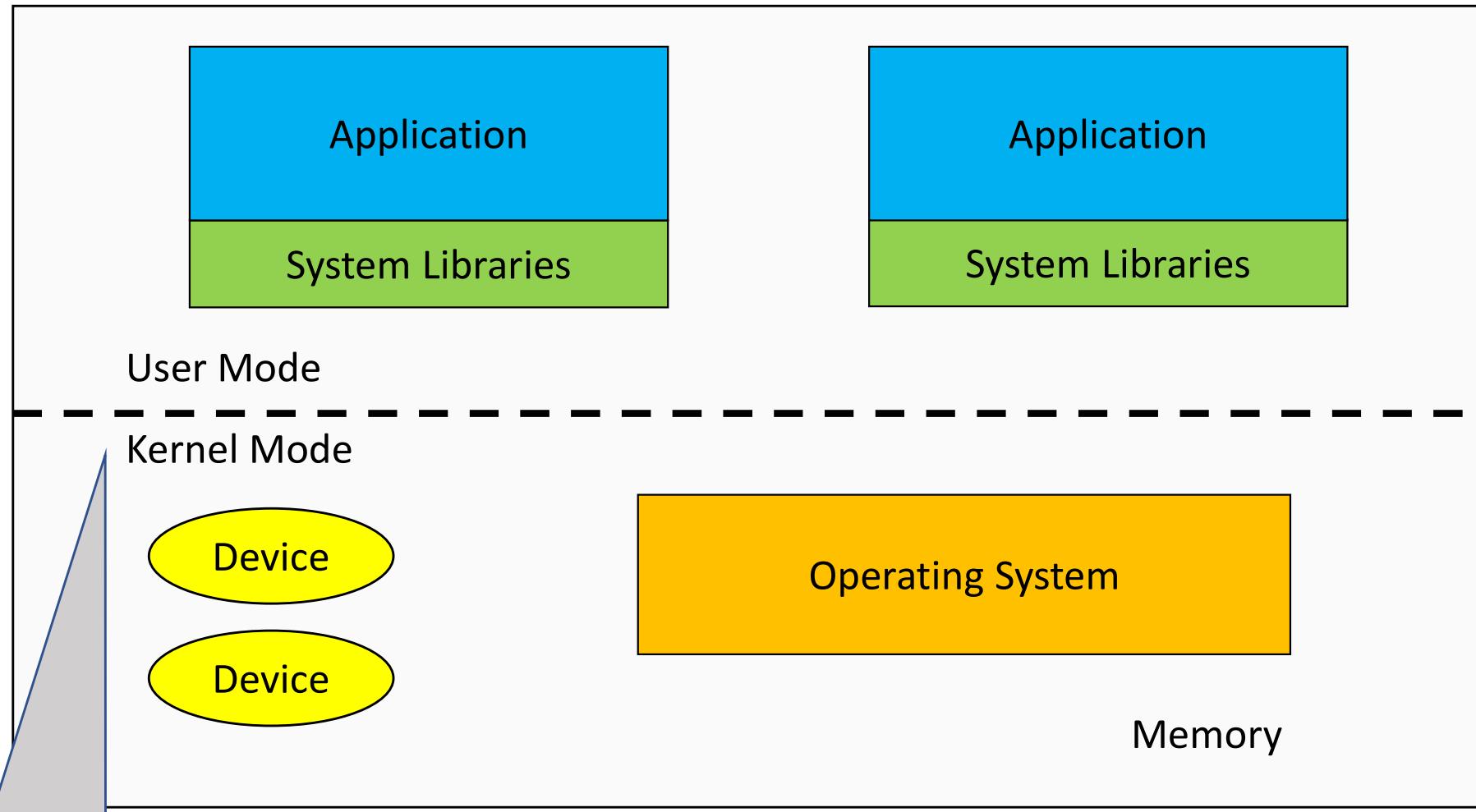


# The Operating System is Privileged

- Applications should not be able to interfere or bypass the operating system
  - OS can enforce the “extended machine”
  - OS can enforce its resource allocation policies
  - Prevent applications from interfering with each other



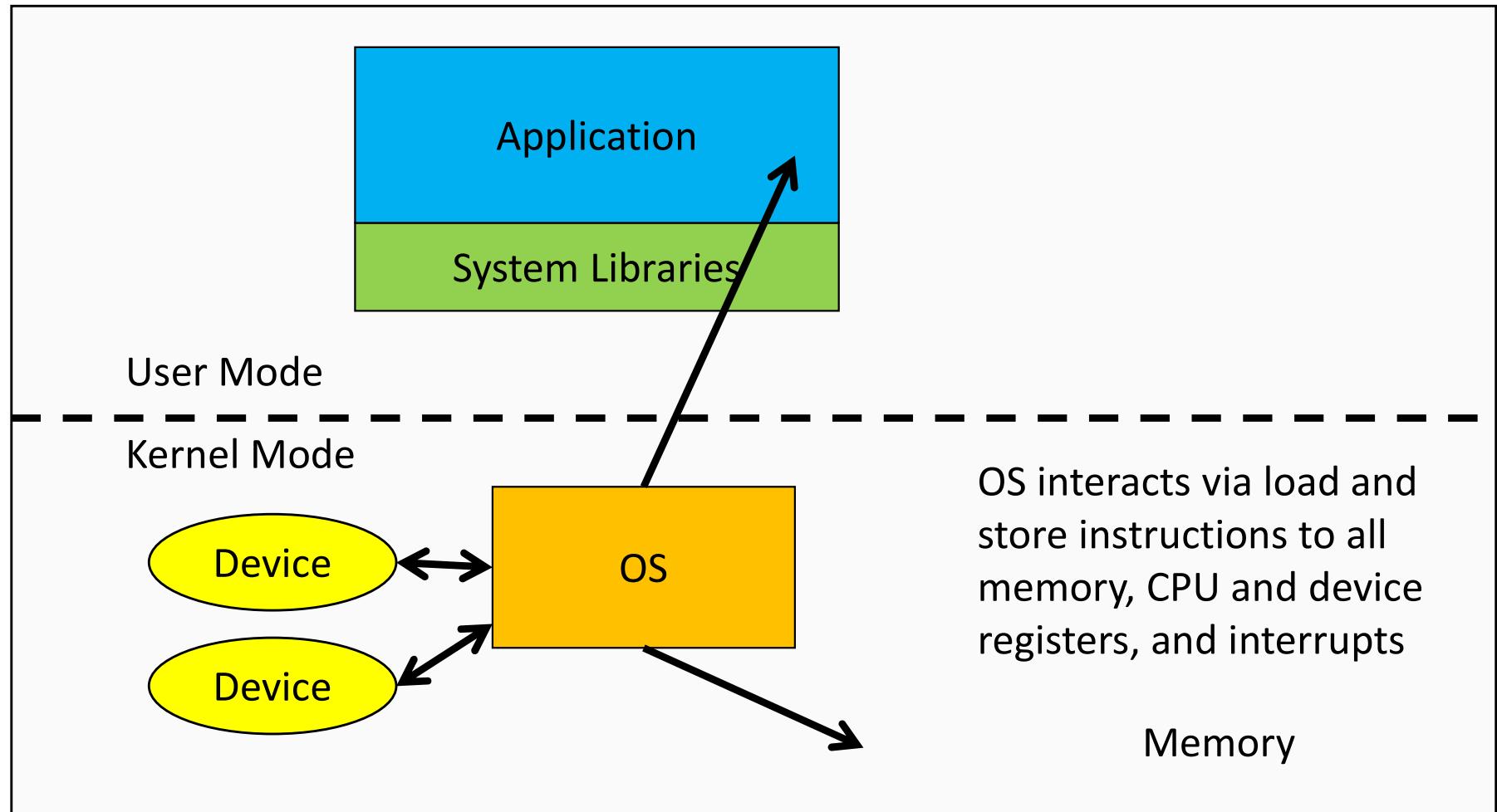
# Delving Deeper: The Structure of a Computer System



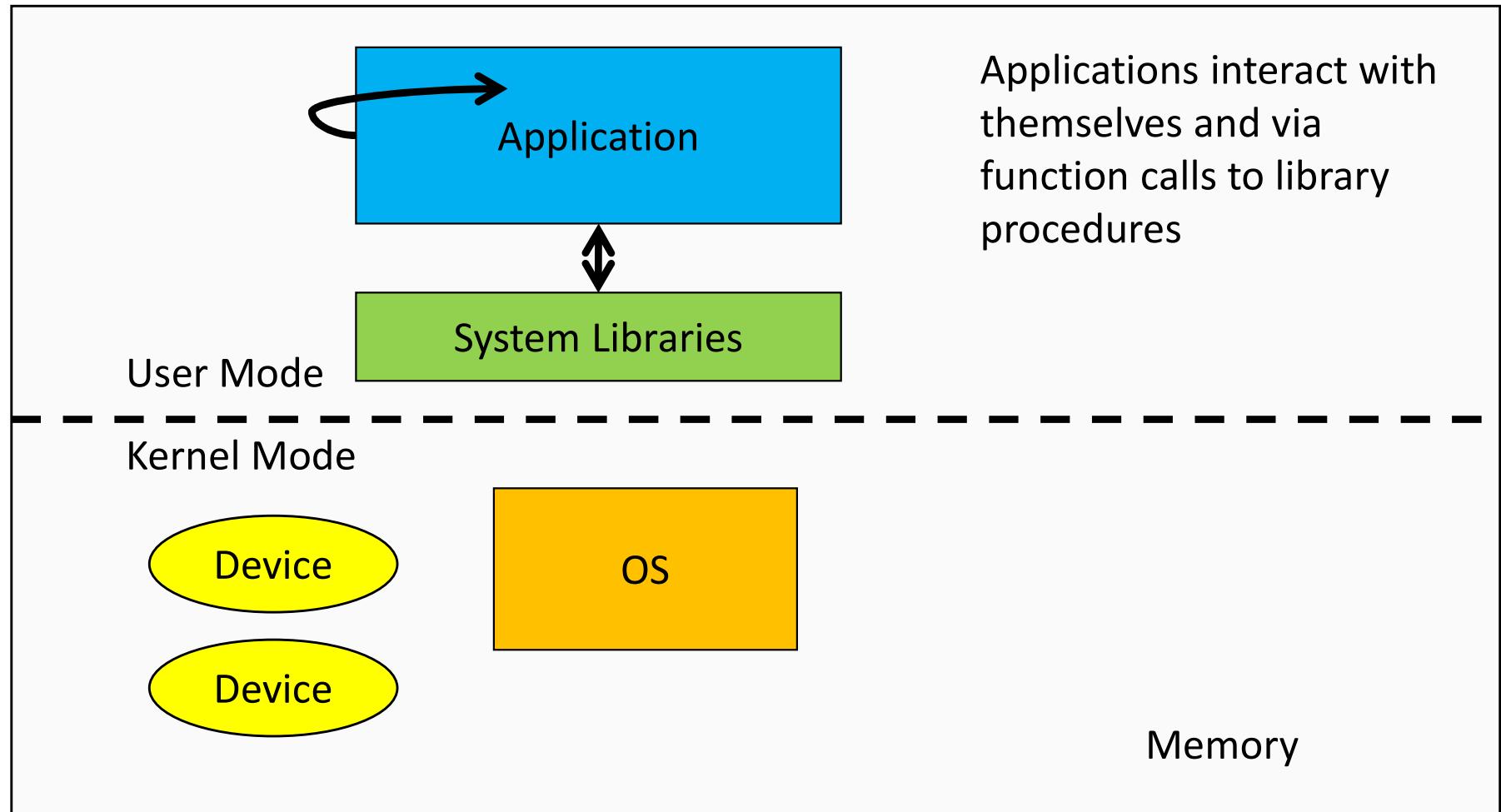
Kernel = Privileged  
Mode



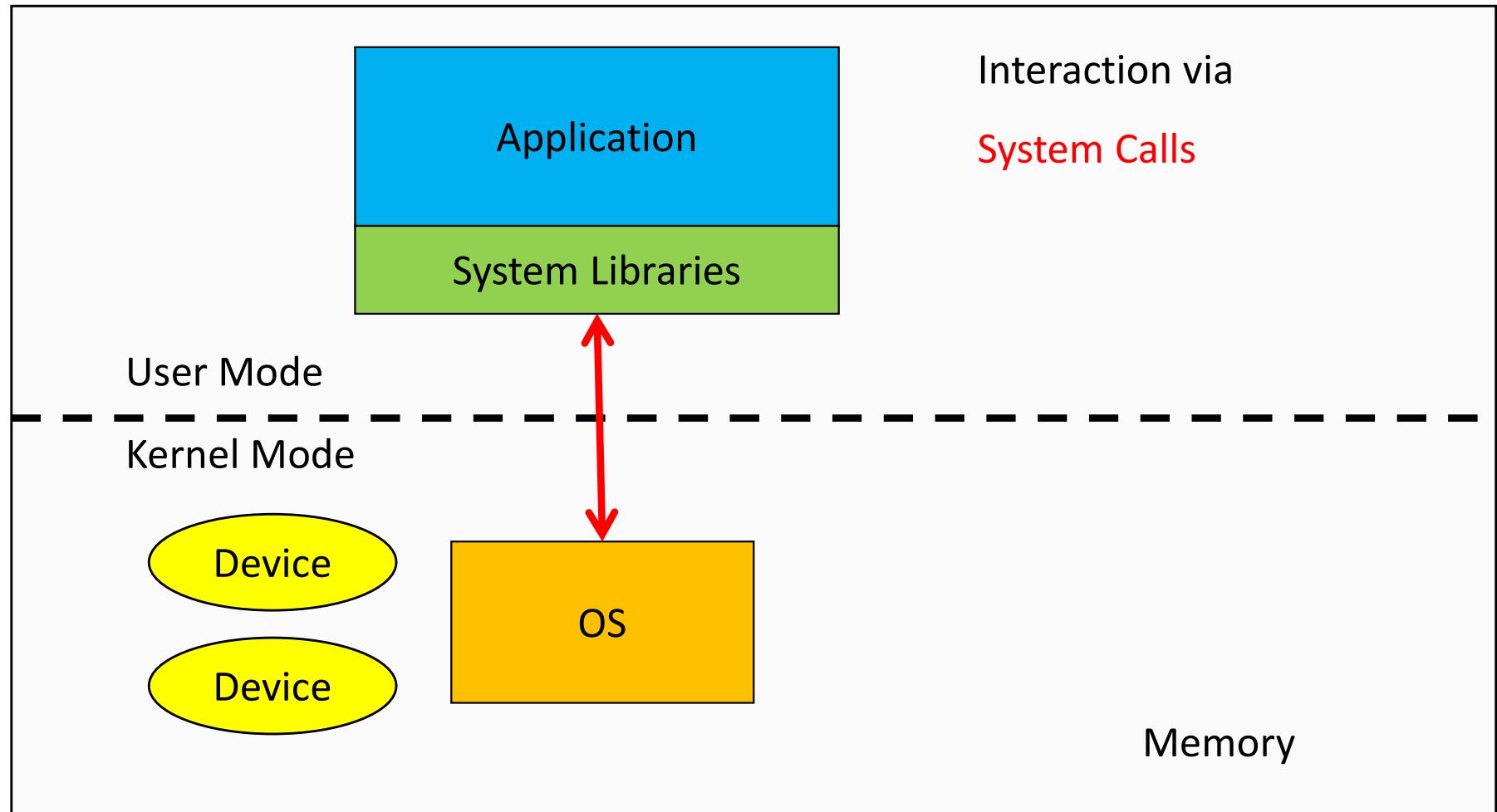
# The Structure of a Computer System



# The Structure of a Computer System

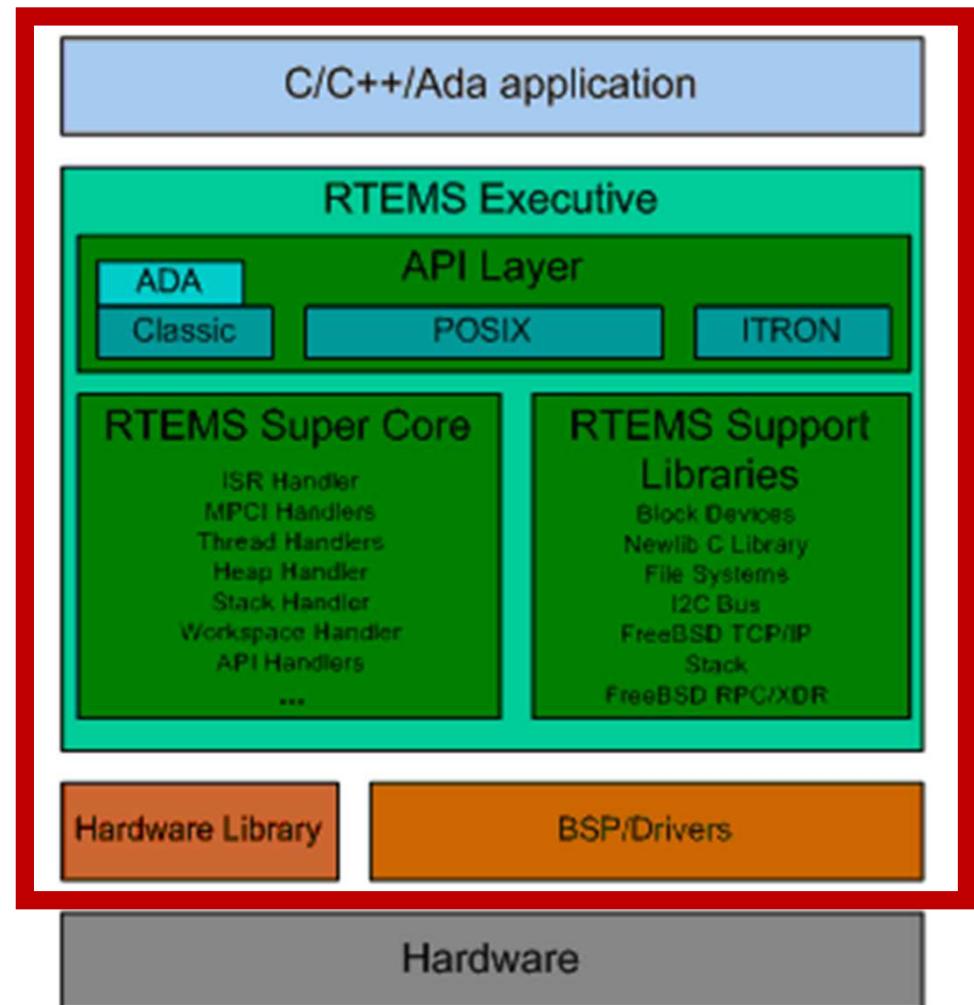


# The Structure of a Computer System



# Privilege-less OS

- Some Embedded OSs have no privileged component
  - e.g. PalmOS, Mac OS 9, RTEMS
  - Can implement OS functionality, but cannot enforce it.
    - All software runs together
    - No isolation
    - One fault potentially brings down entire system



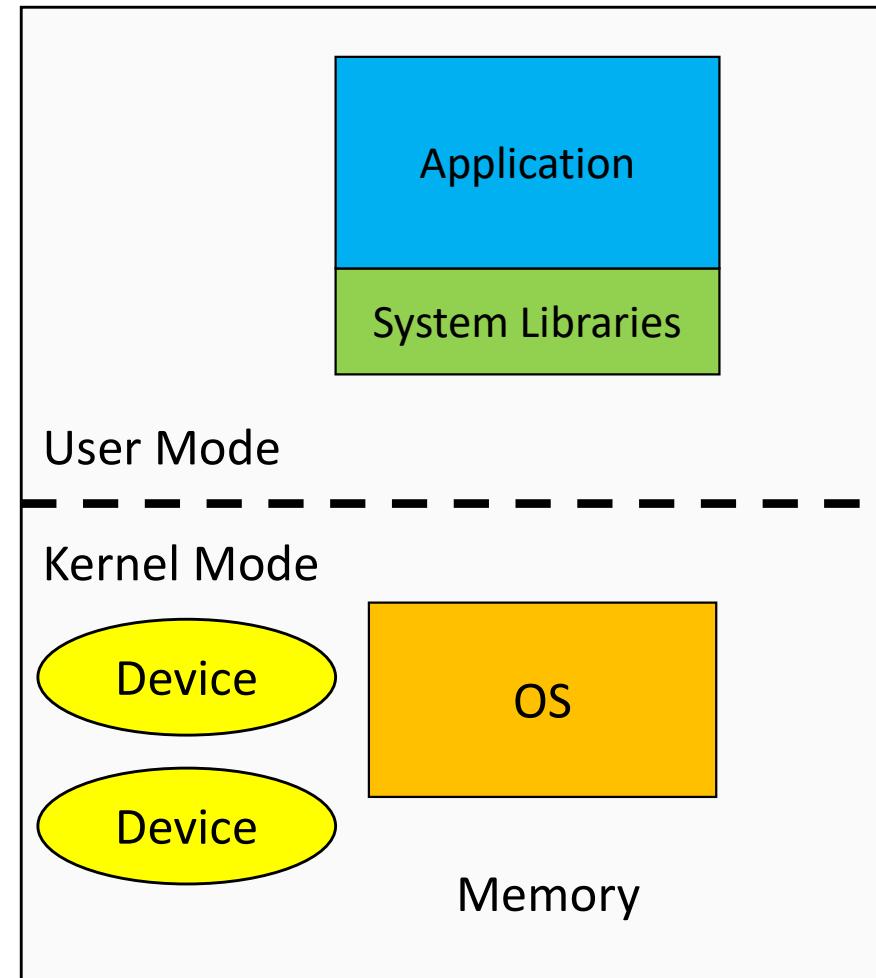
# A Note on System Libraries

System libraries are just that, libraries of support functions (procedures, subroutines)

- Only a subset of library functions are actually system calls
  - `strcmp()`, `memcpy()`, are pure library functions
    - manipulate memory within the application, or perform computation
  - `open()`, `close()`, `read()`, `write()` are system calls
    - they cross the user-kernel boundary, e.g. to read from disk device
    - Implementation mainly focused on passing request to OS and returning result to application
- System call functions are in the library for convenience
  - try `man syscalls` on Linux

# Operating System Software

- Fundamentally, OS functions the same way as ordinary computer software
  - It is machine code that is executed (same machine instructions as application)
  - It has more privileges (extra instructions and access)
- Operating system relinquishes control of the processor to execute other programs
  - Reestablishes control after
    - System calls
    - Interrupts (especially timer interrupts)

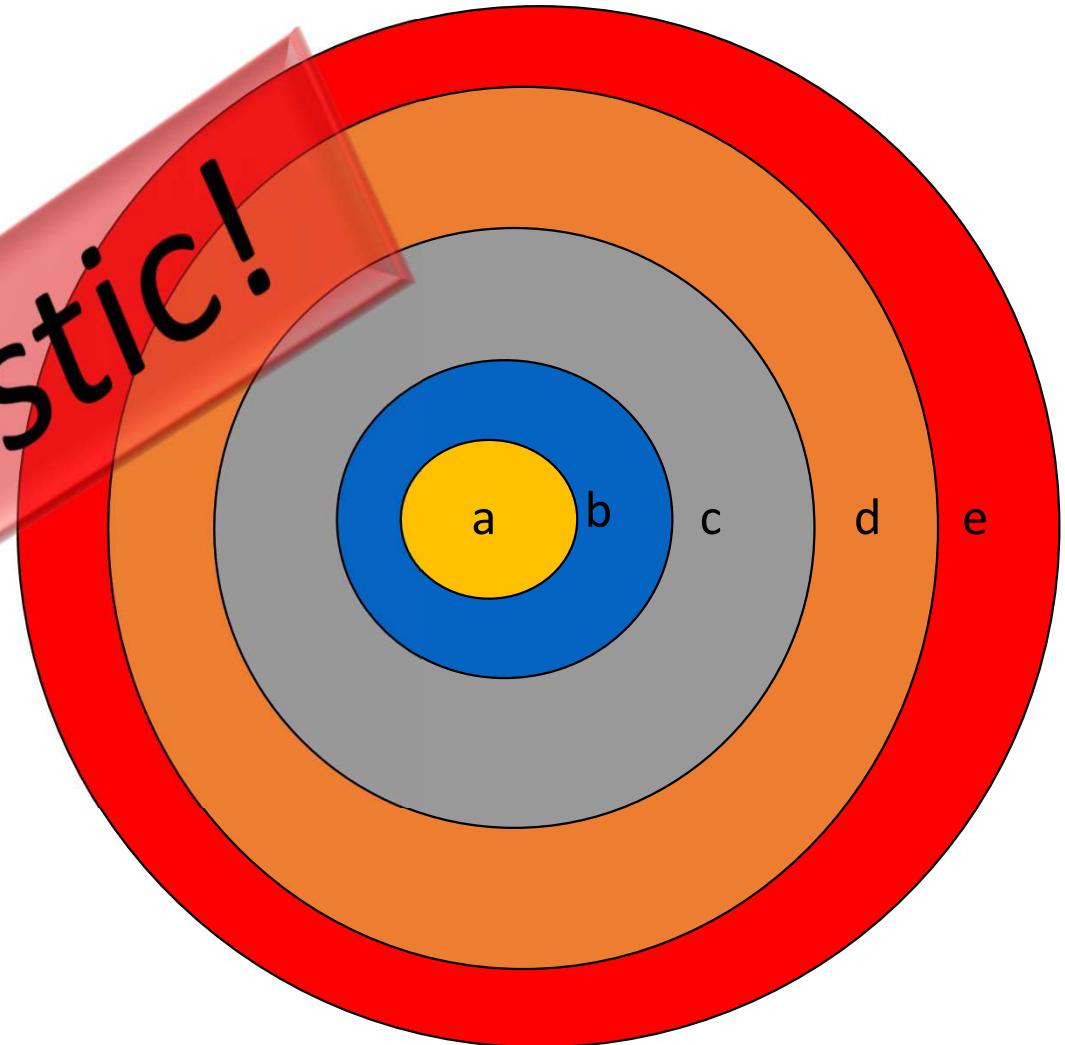


# Operating System Internal Structure?

# Classic Operating System Structure

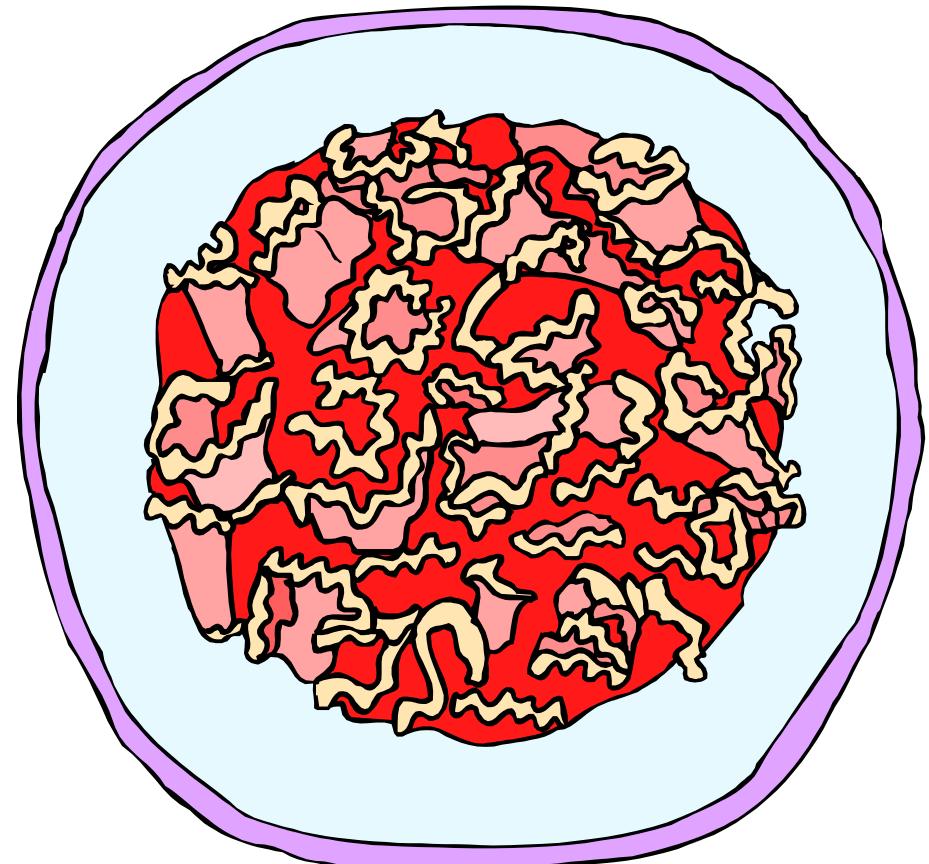
- The layered approach
  - a) Processor allocation and multiprogramming
  - b) Memory Management
  - c) Devices
  - d) File system
  - e) Users
- Each layer depends on the inner layers

Unrealistic!



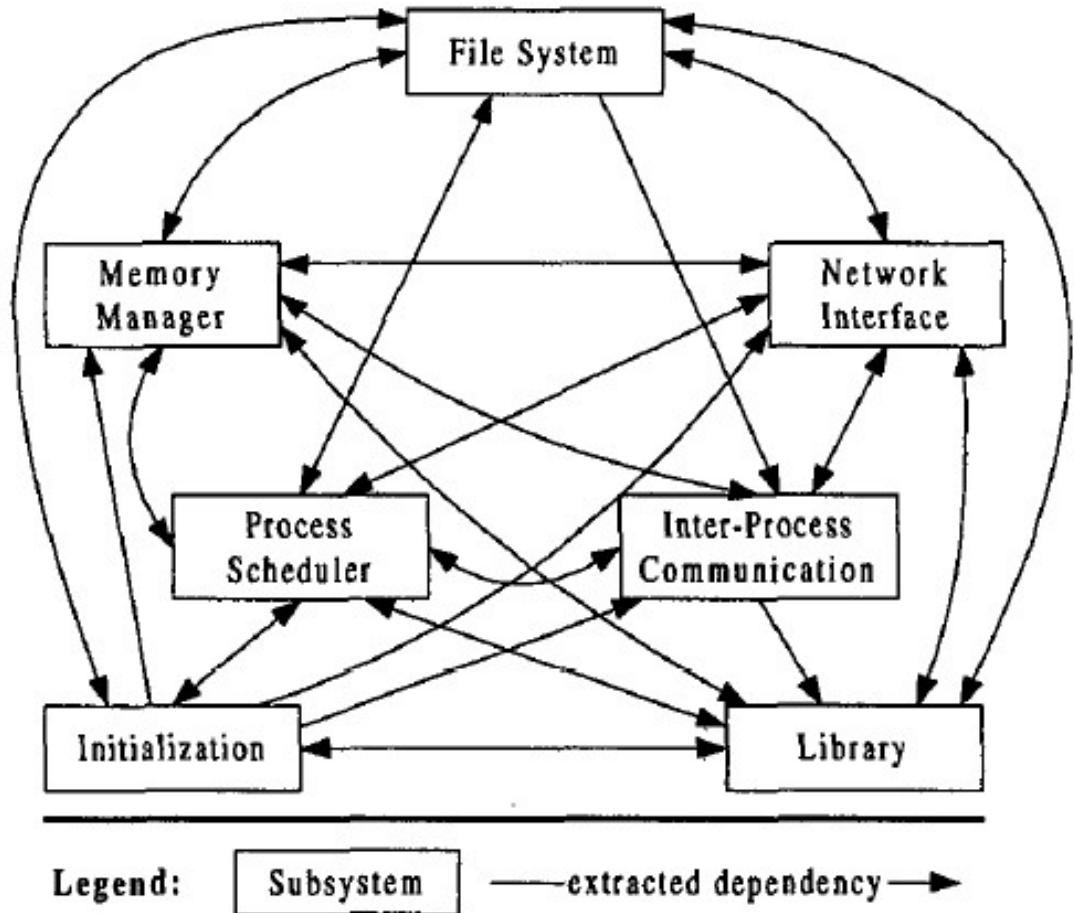
# The Monolithic Operating System Structure

- Also called the “spaghetti nest” approach
  - Everything is tangled up with everything else.
- Linux, Windows, ....

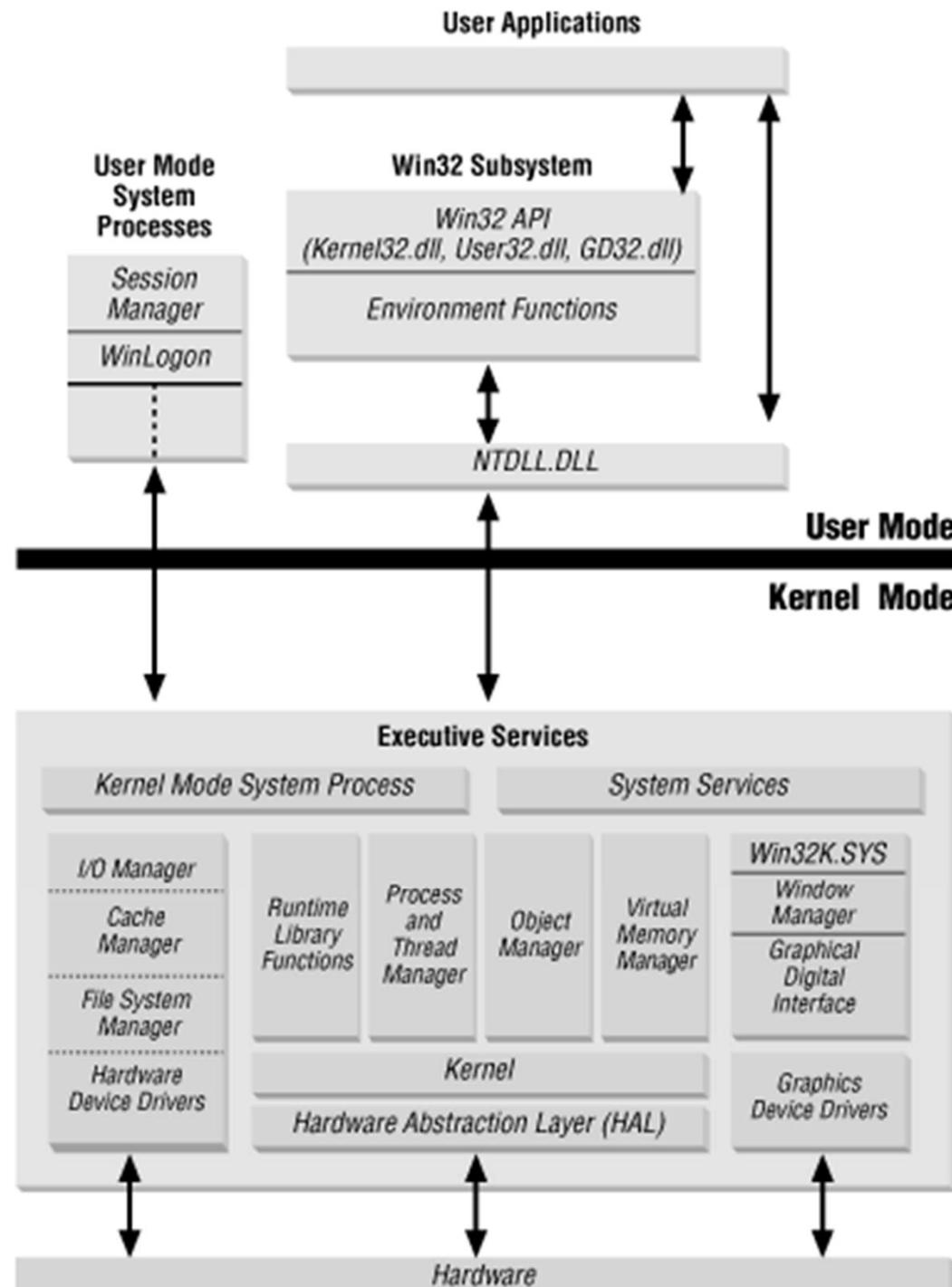


# The Monolithic Operating System Structure

- However, some reasonable structure usually prevails



Bowman, I. T., Holt, R. C., and Brewster, N. V. 1999. Linux as a case study: its extracted software architecture. In *Proceedings of the 21st international Conference on Software Engineering* (Los Angeles, California, United States, May 16 - 22, 1999). ICSE '99. ACM, New York, NY, 555-563.  
DOI= <http://doi.acm.org/10.1145/302405.302691>



# Processes and Threads

# Learning Outcomes

- An understanding of fundamental concepts of processes and threads
  - I'll cover implementation in a later lecture

# Major Requirements of an Operating System

- Interleave the execution of several processes to maximize processor utilization while providing reasonable response time
- Allocate resources to processes
- Support interprocess communication and user creation and management of processes

# Processes and Threads

- Processes:
  - Also called a task or job
  - Execution of an individual program
  - “Owner” of resources allocated for program execution
  - Encompasses one or more threads
- Threads:
  - Unit of execution
  - Can be traced
    - list the sequence of instructions that execute
  - Belongs to a process
    - Executes within it.

Execution snapshot of three single-threaded processes (No Virtual Memory)

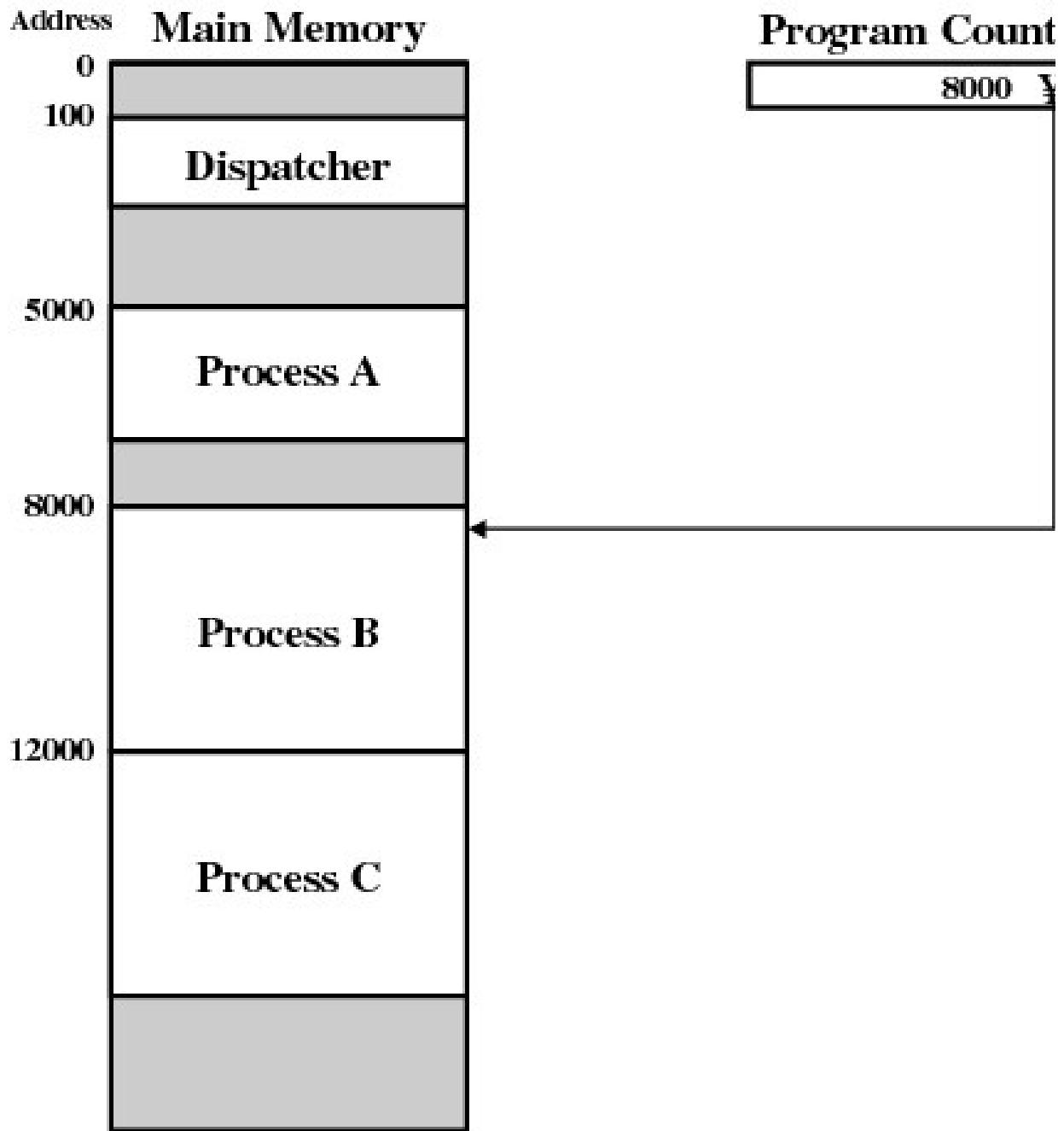


Figure 3.1 Snapshot of Example Execution (Figure 3 at Instruction Cycle 13)

## Logical Execution Trace

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A

(b) Trace of Process B

(c) Trace of Process C

5000 = Starting address of program of Process A

8000 = Starting address of program of Process B

12000 = Starting address of program of Process C

**Figure 3.2 Traces of Processes of Figure 3.1**

## Combined Traces

(Actual CPU Instructions)

What are the shaded sections?

1	5000	27	12004
2	5001	28	12005
3	5002	----- Time out	
4	5003	29	100
5	5004	30	101
6	5005	31	102
----- Time out		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	----- Time out	
16	8003	41	100
----- I/O request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
----- Time out			

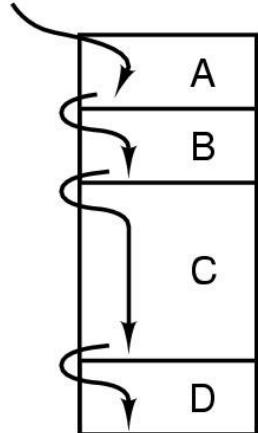
100 = Starting address of dispatcher program

shaded areas indicate execution of dispatcher process;  
first and third columns count instruction cycles;  
second and fourth columns show address of instruction being executed

Figure 3.3 Combined Trace of Processes of Figure 3.1

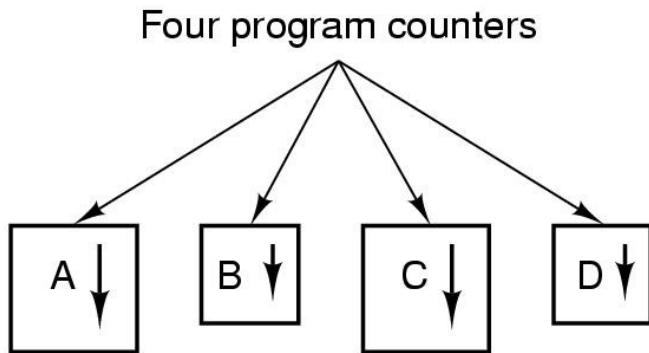
# Summary: The Process Model

One program counter

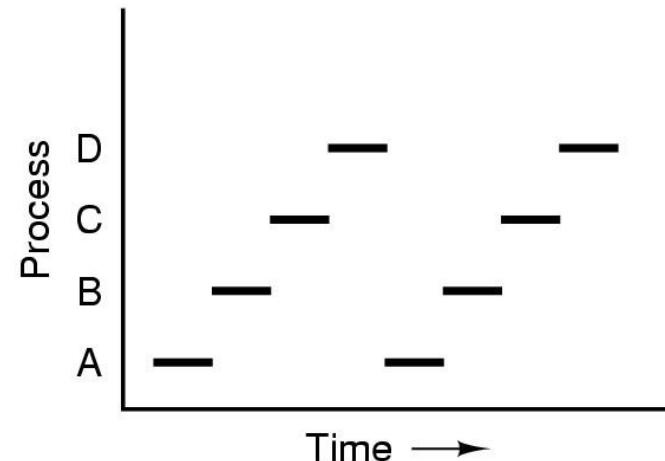


(a)

Process switch

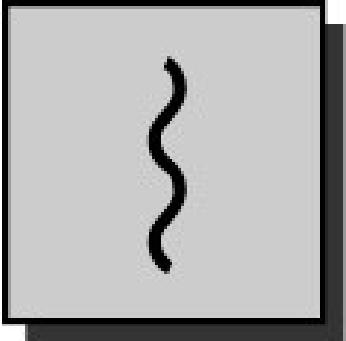


(b)

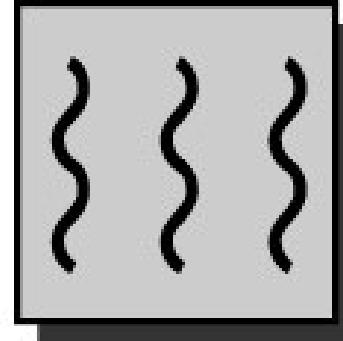


(c)

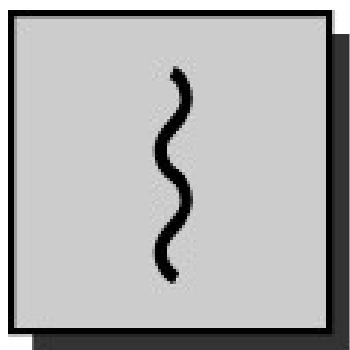
- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes (with a single thread each)
- Only one program active at any instant



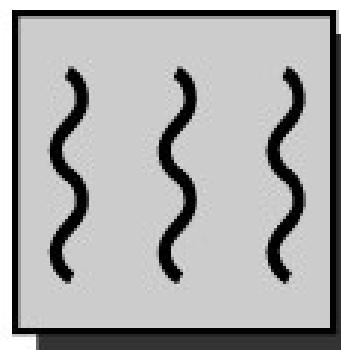
one process  
one thread



one process  
multiple threads



multiple processes  
one thread per process



multiple processes  
multiple threads per process

{ = Instruction trace

**Figure 4.1 Threads and Processes [ANDE97]**

# Process and thread models of selected OSes

- Single process, single thread
  - MSDOS
- Single process, multiple threads
  - OS/161 as distributed
- Multiple processes, single thread
  - Traditional UNIX
- Multiple processes, multiple threads
  - Modern Unix (Linux, Solaris), Windows

Note: Literature (incl. Textbooks) often do not cleanly distinguish between processes and threads (for historical reasons)

# Process Creation

## Principal events that cause process creation

### 1. System initialization

- Foreground processes (interactive programs)
- Background processes
  - Email server, web server, print server, etc.
  - Called a *daemon* (unix) or *service* (Windows)

### 2. Execution of a process creation system call by a running process

- New login shell for an incoming ssh connection

### 3. User request to create a new process

### 4. Initiation of a batch job

Note: Technically, all these cases use the same system mechanism to create new processes.

# Process Termination

Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

# Implementation of Processes

- A processes' information is stored in a *process control block* (PCB)
- The PCBs form a *process table*
  - Reality can be more complex (hashing, chaining, allocation bitmaps,...)

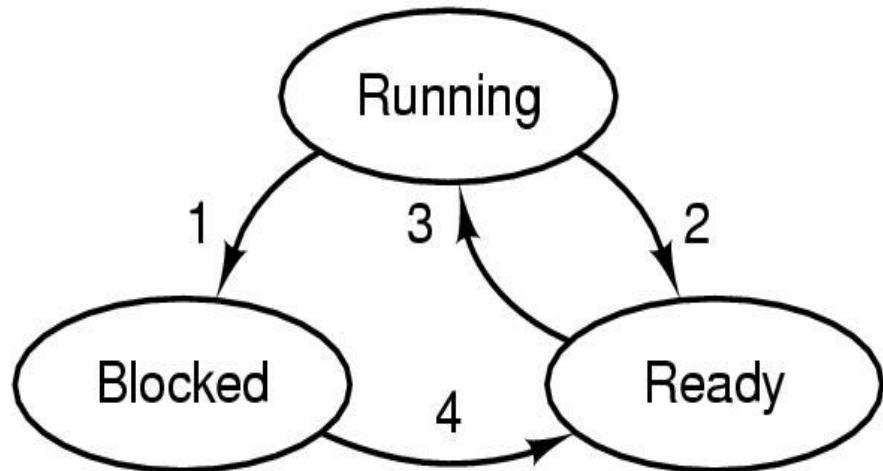
P7
P6
P5
P4
P3
P2
P1
P0

# Implementation of Processes

<b>Process management</b>	<b>Memory management</b>	<b>File management</b>
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Example fields of a process table entry

# Process/Thread States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process/thread states
  - running
  - blocked
  - ready
- Transitions between states shown

# Some Transition Causing Events

Running → Ready

- Voluntary **Yield()**
- End of timeslice

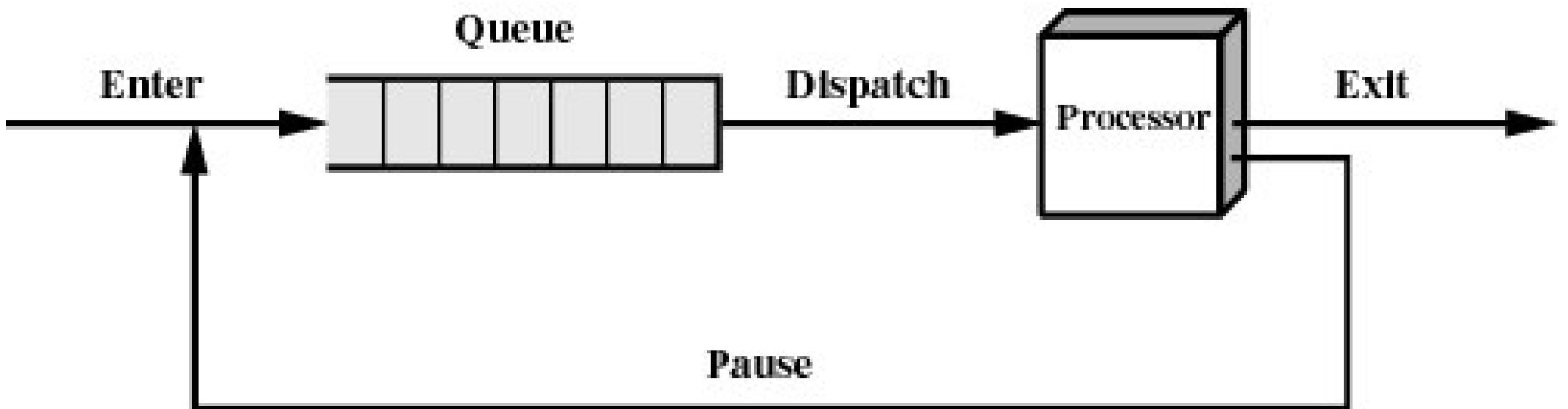
Running → Blocked

- Waiting for input
  - File, network,
- Waiting for a timer (alarm signal)
- Waiting for a resource to become available

# Scheduler

- Sometimes also called the *dispatcher*
  - The literature is also a little inconsistent on with terminology.
- Has to choose a *Ready* process to run
  - How?
  - It is inefficient to search through all processes

# The Ready Queue

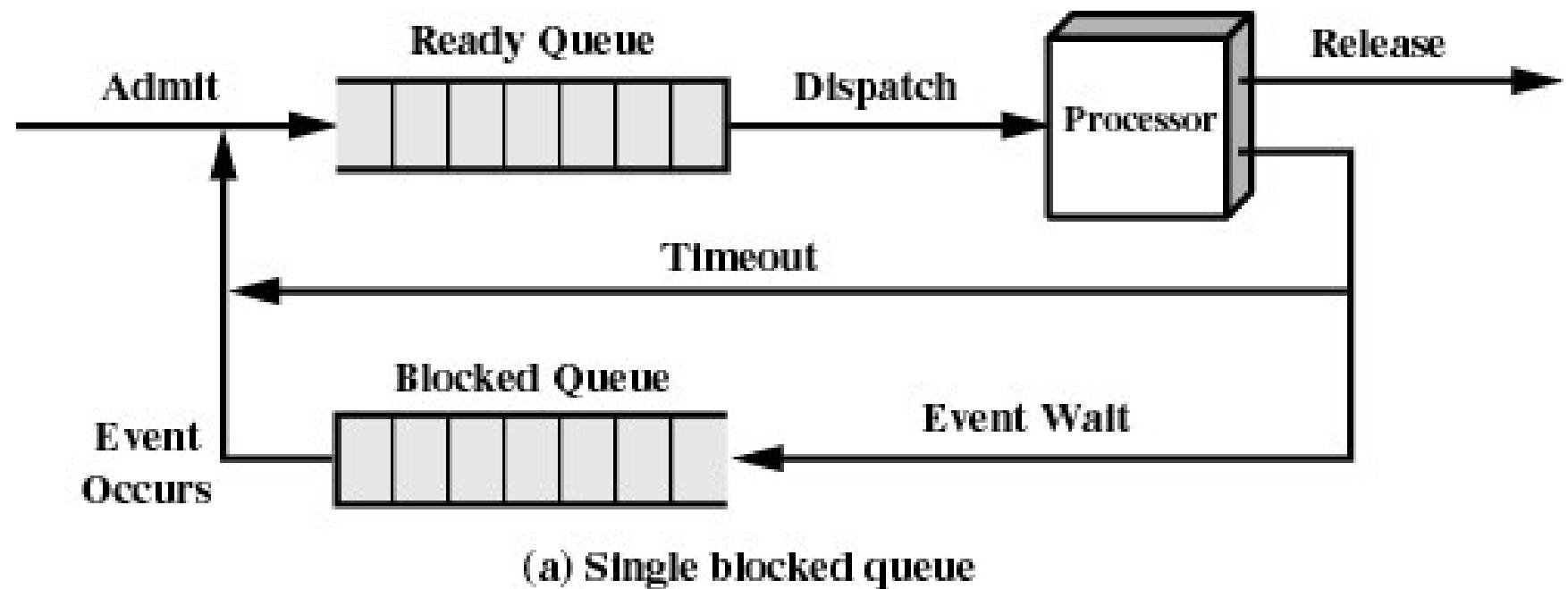


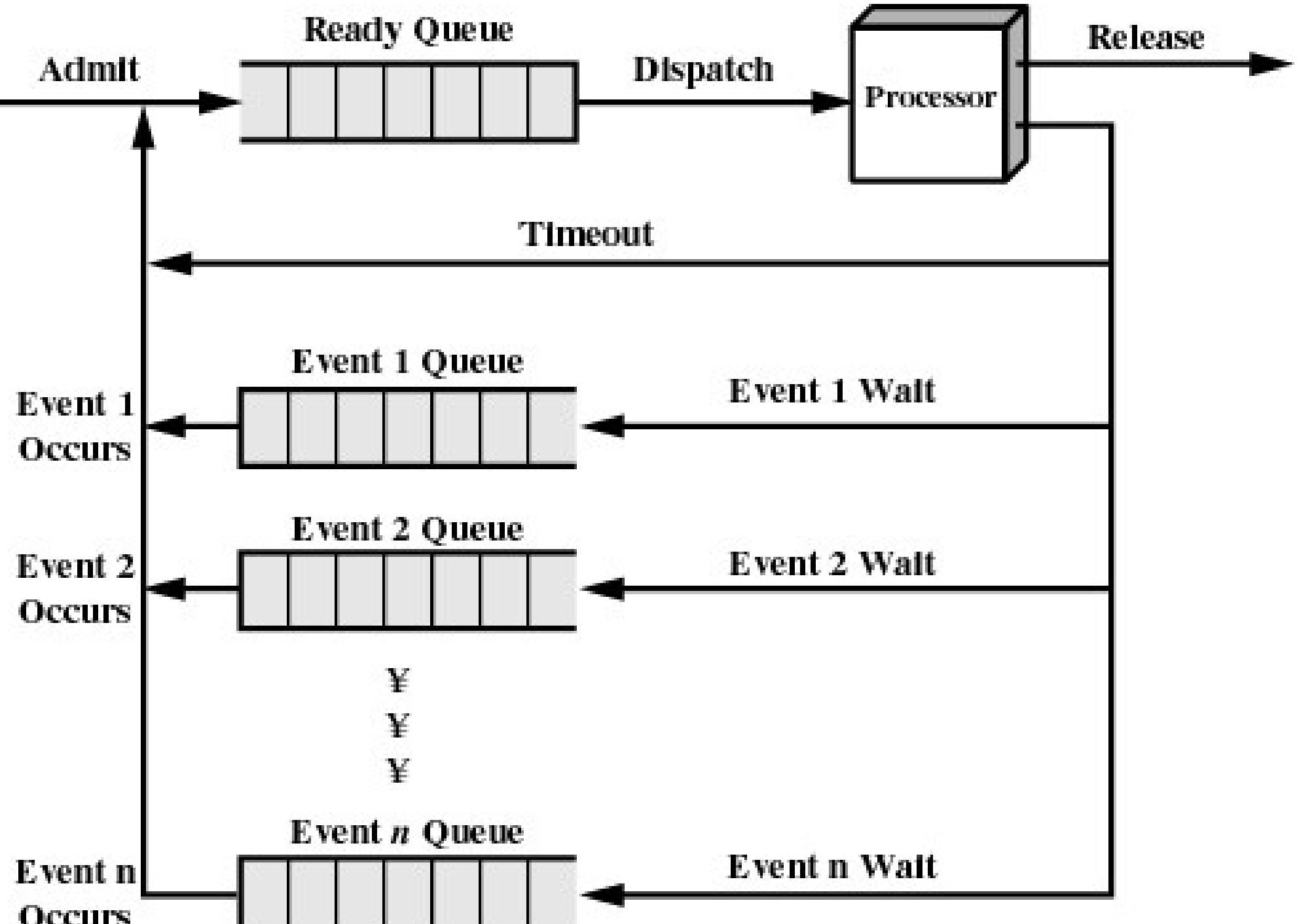
(b) Queuing diagram

# What about blocked processes?

- When an *unblocking* event occurs, we also wish to avoid scanning all processes to select one to make *Ready*

# Using Two Queues

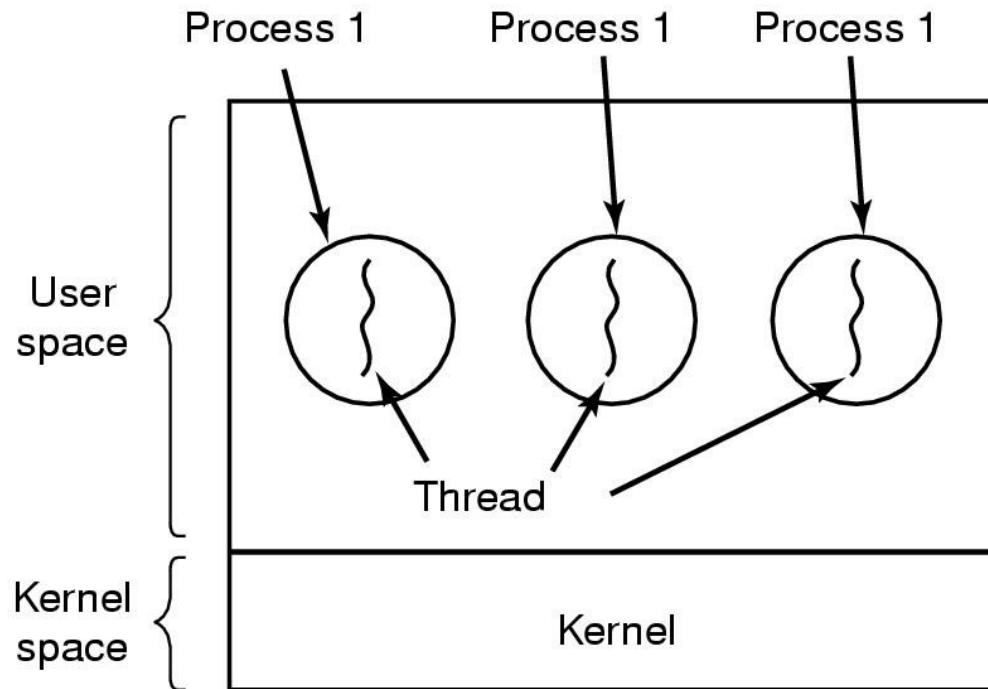




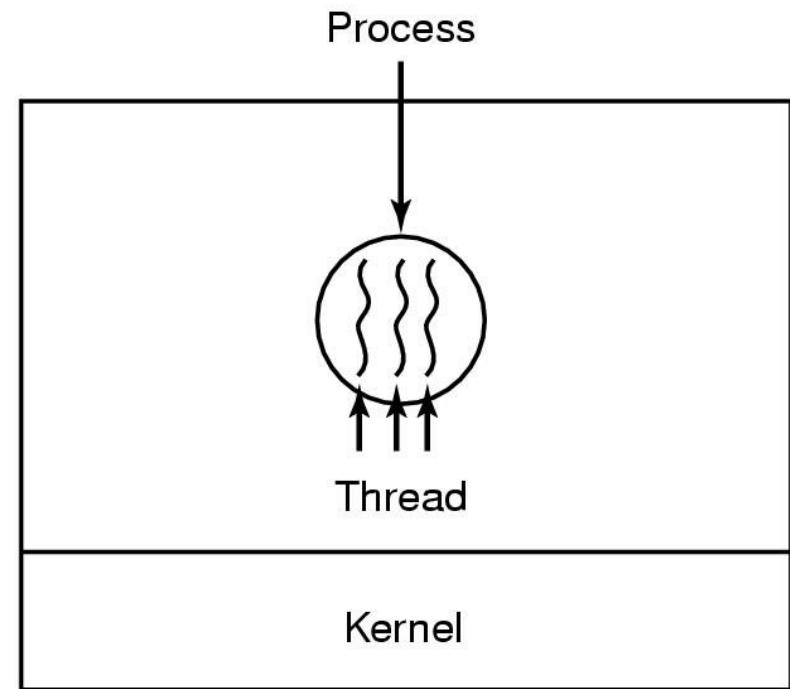
(b) Multiple blocked queues

# Threads

## The Thread Model



(a)



(b)

- (a) Three processes each with one thread
- (b) One process with three threads

# The Thread Model – Separating execution from the environment.

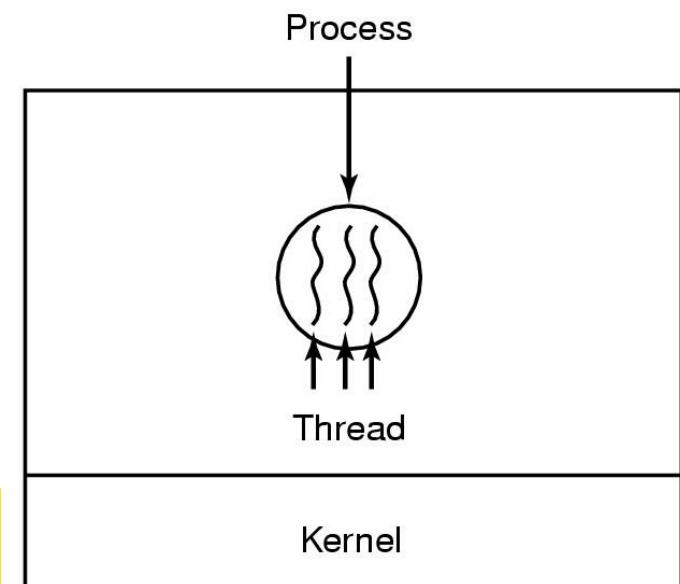
## Per process items

Address space  
Global variables  
Open files  
Child processes  
Pending alarms  
Signals and signal handlers  
Accounting information

## Per thread items

Program counter  
Registers  
Stack  
State

- Items shared by all threads in a process
- Items private to each thread

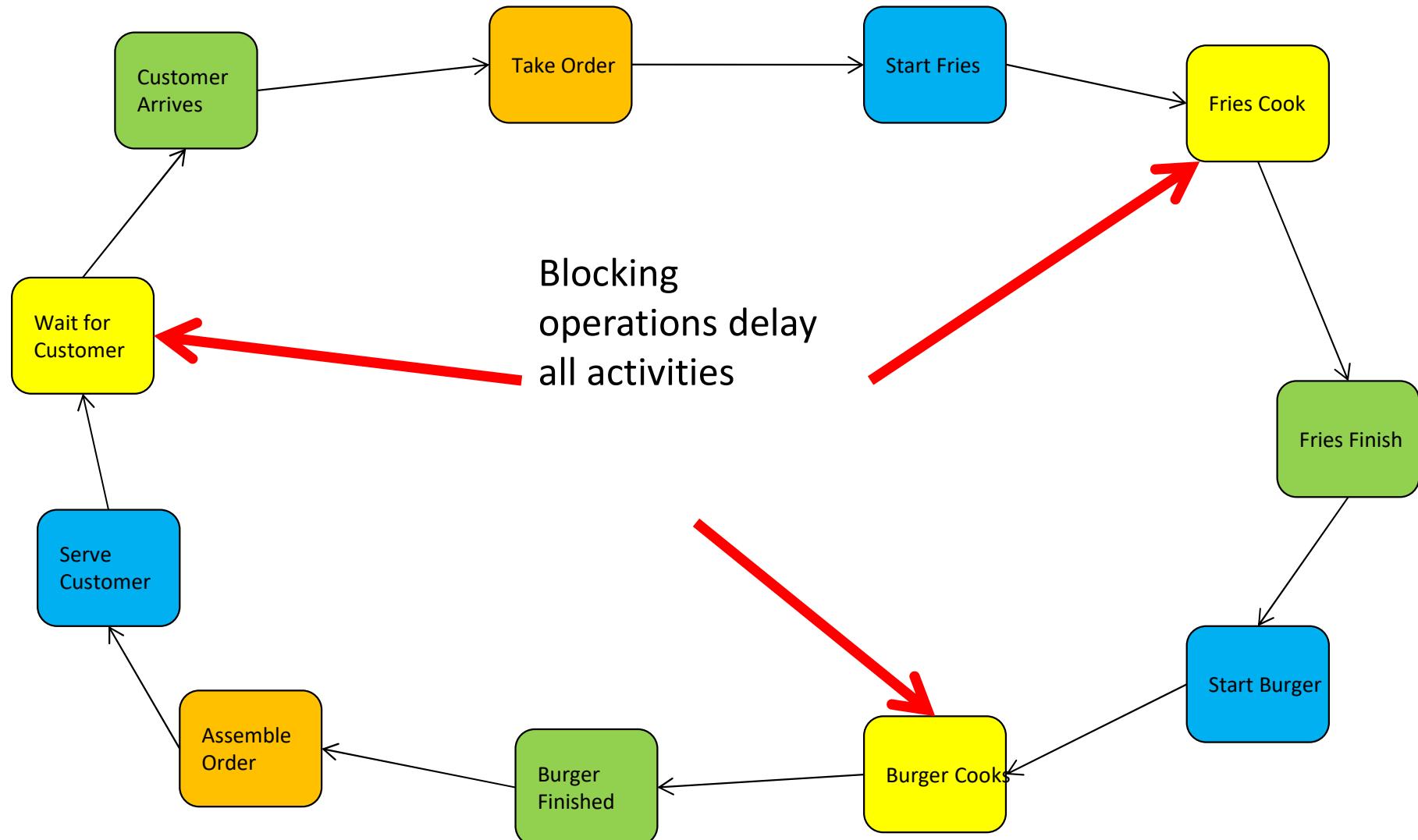


# Threads Analogy

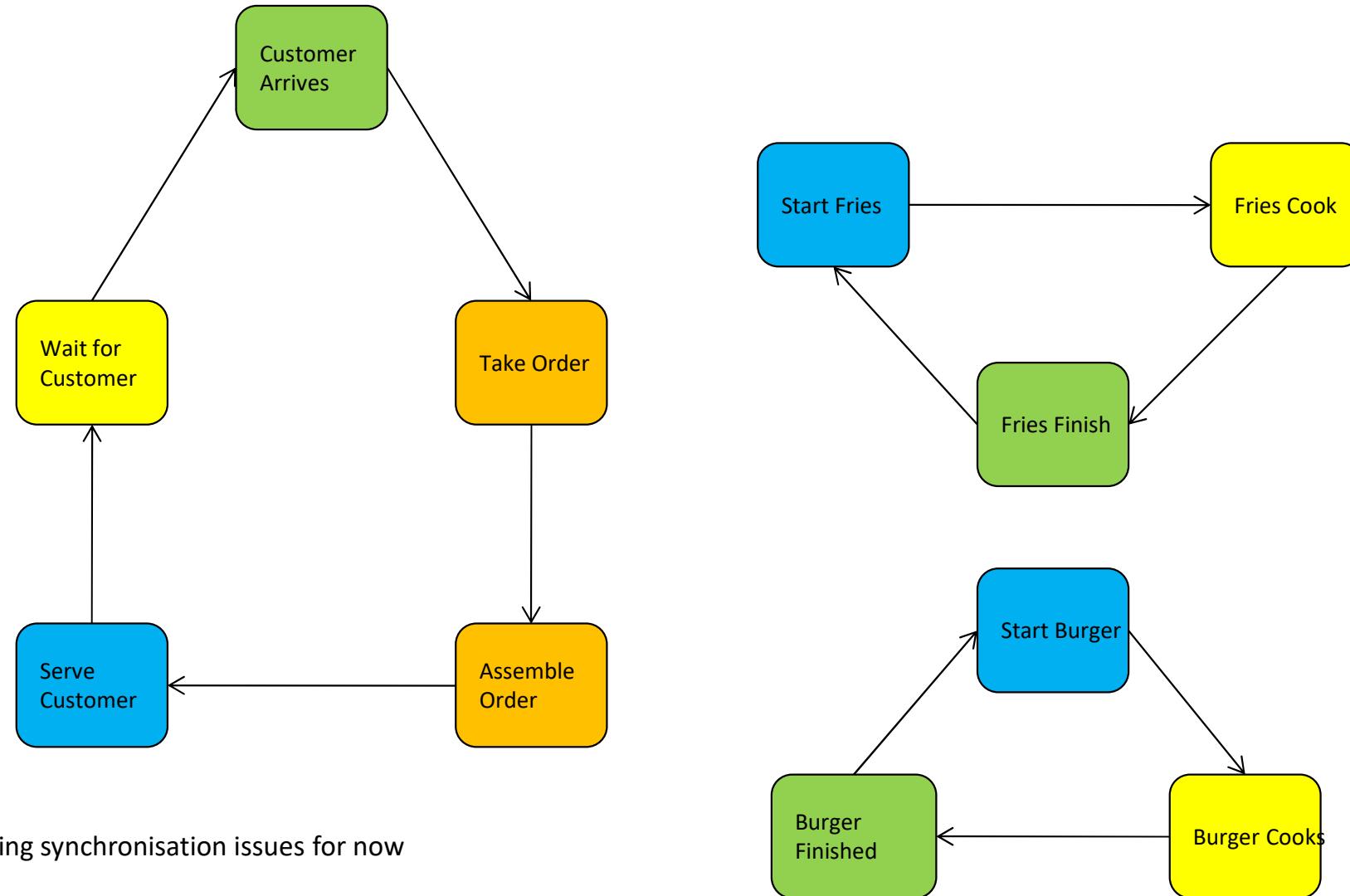


The Hamburger Restaurant

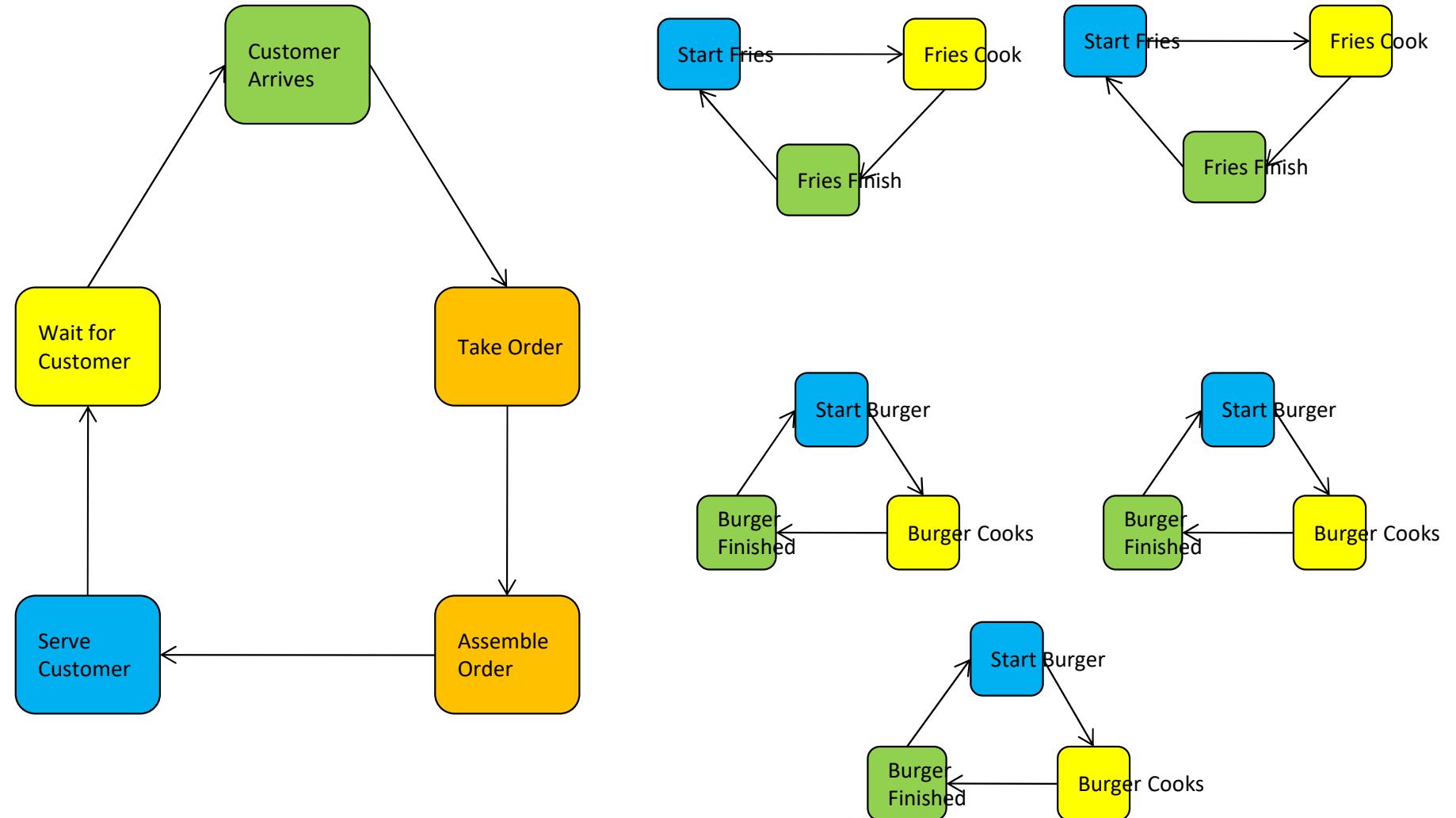
# Single-Threaded Restaurant



# Multithreaded Restaurant

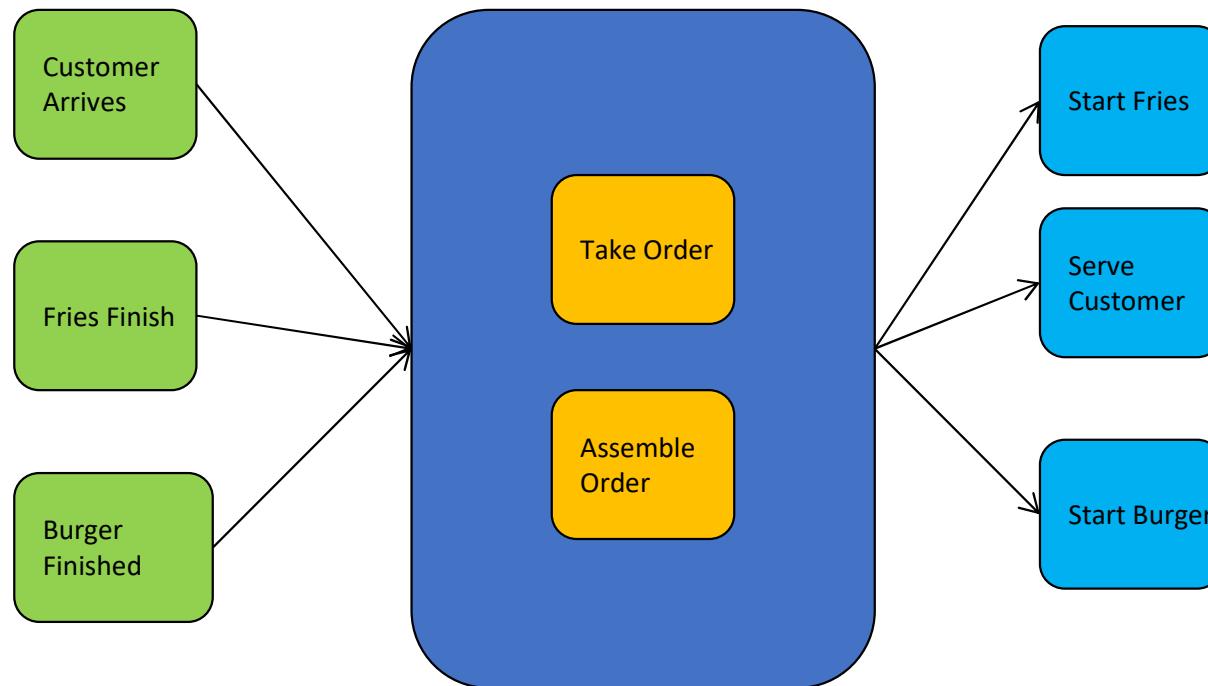


# Multithreaded Restaurant with more worker threads



# Finite-State Machine Model (Event-based model)

Input  
Events



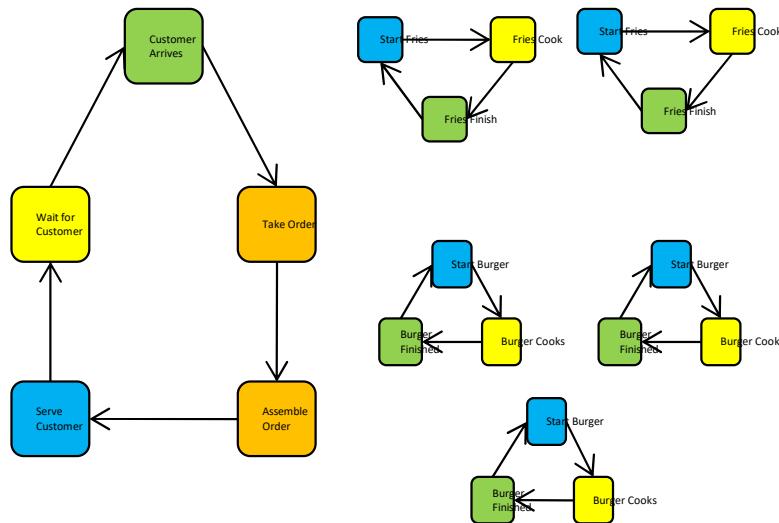
Non-Blocking  
actions



External  
activities

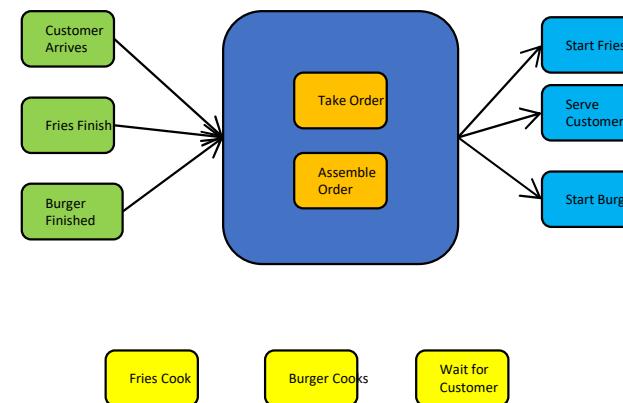
# Observation: Computation State

Thread Model



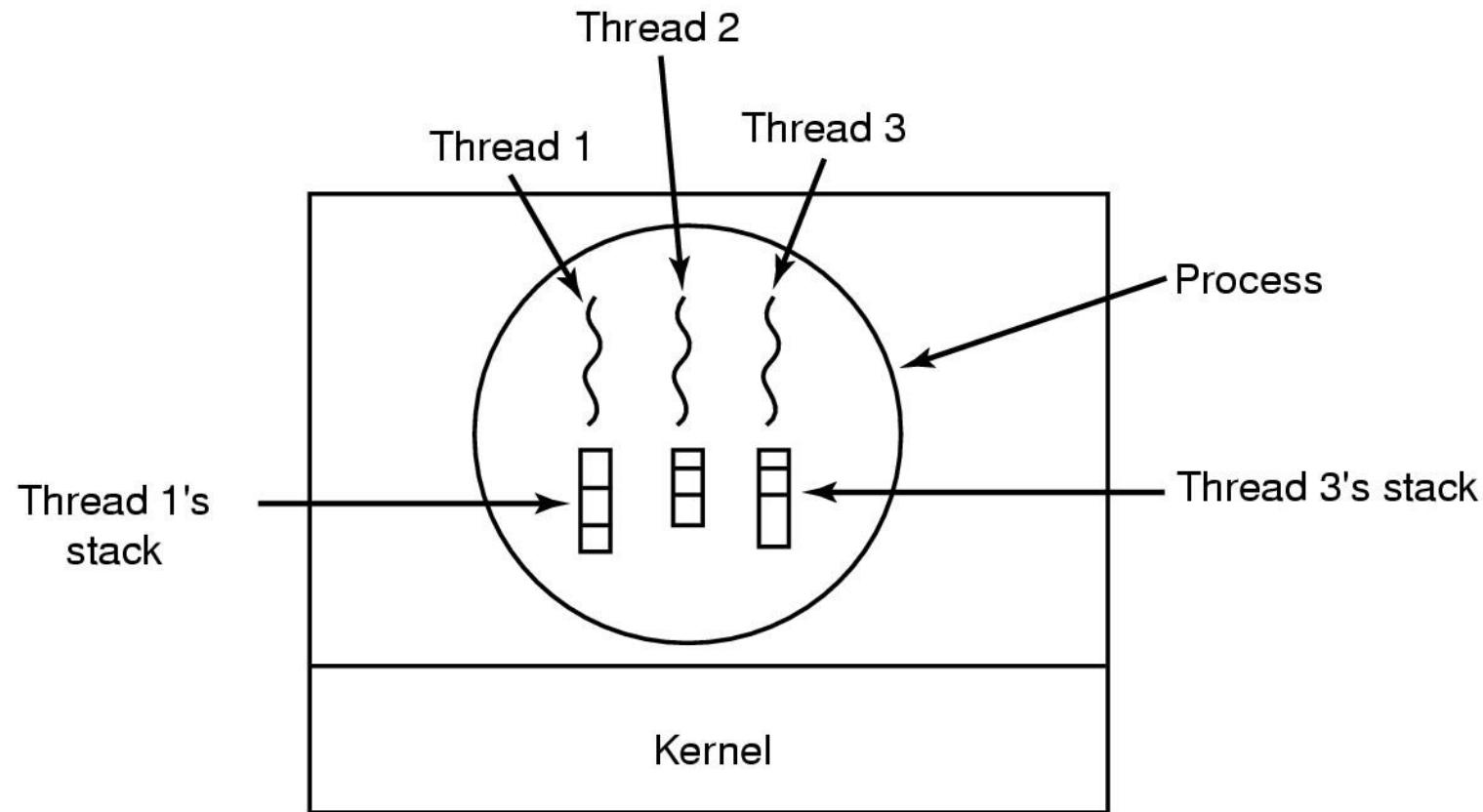
- State implicitly stored on the stack.

Finite State (Event) Model



- State explicitly managed by program

# The Thread Model

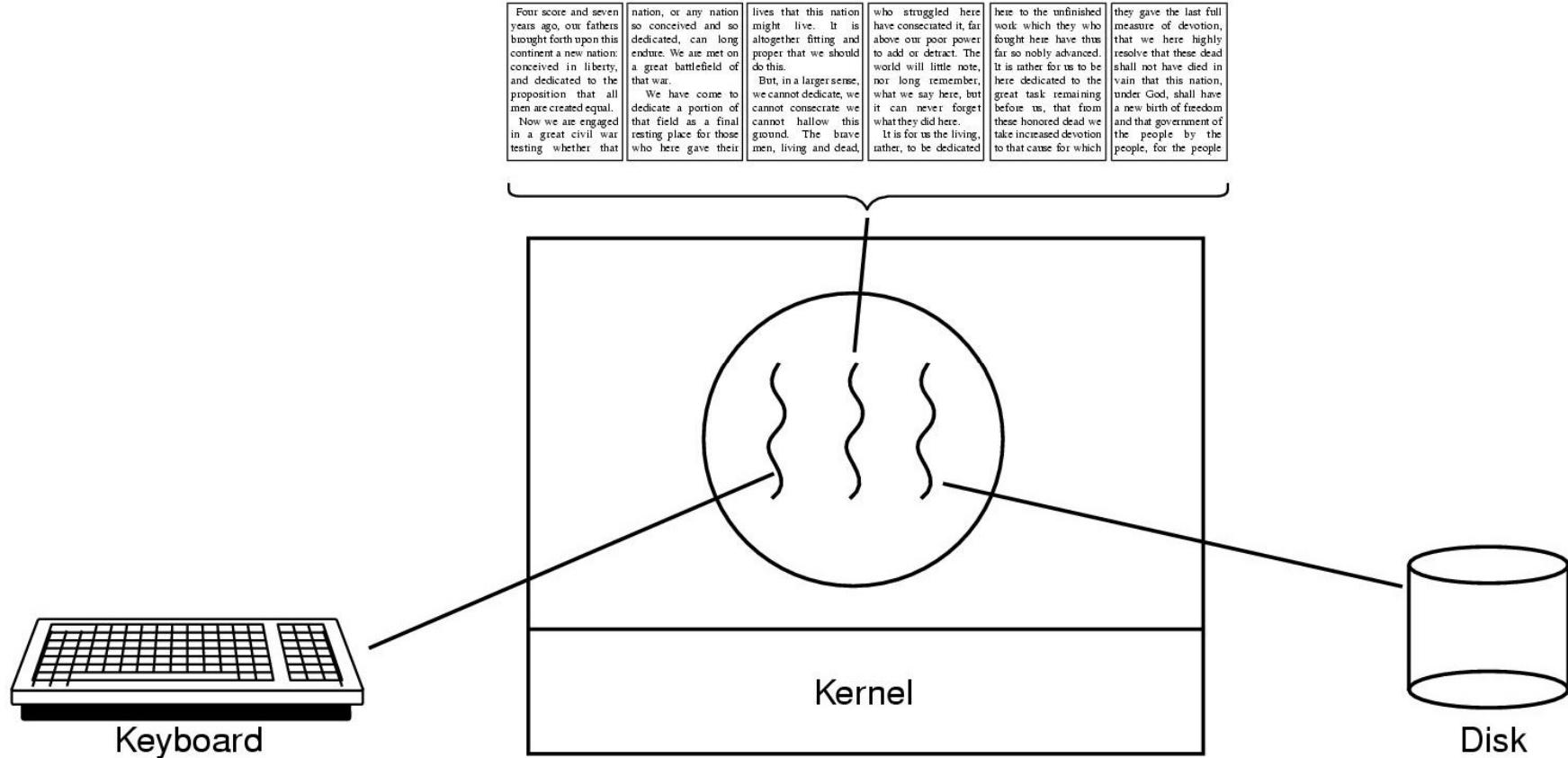


Each thread has its own stack

# Thread Model

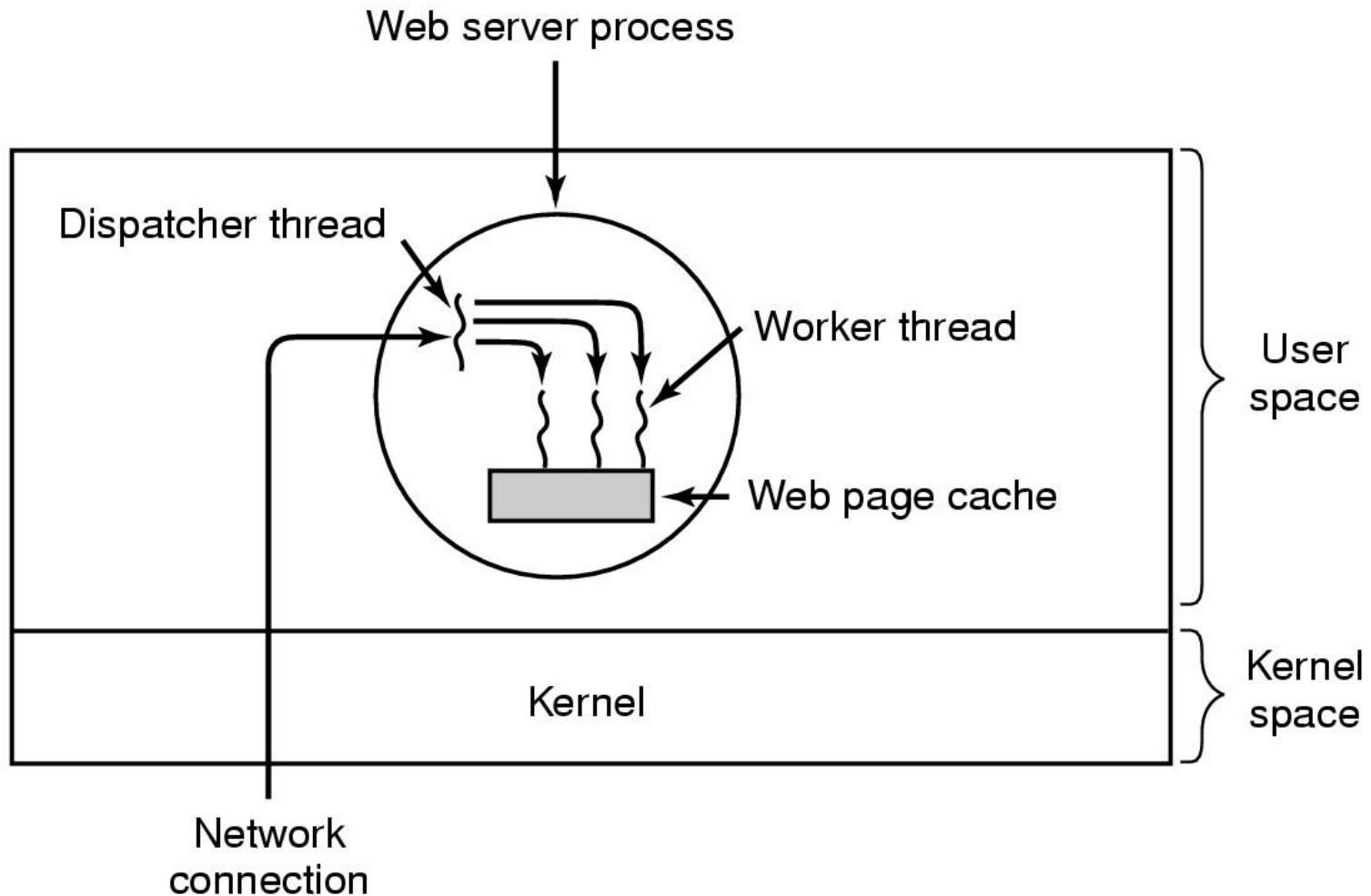
- Local variables are per thread
  - Allocated on the stack
- Global variables are shared between all threads
  - Allocated in data section
  - Concurrency control is an issue
- Dynamically allocated memory (`malloc`) can be global or local
  - Program defined (the pointer can be global or local)

# Thread Usage



A word processor with three threads

# Thread Usage



A multithreaded Web server

# Thread Usage

```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a)

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page)  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b)

- Rough outline of code for previous slide
  - (a) Dispatcher thread
  - (b) Worker thread – can overlap disk I/O with execution of other threads

# Thread Usage

<b>Model</b>	<b>Characteristics</b>
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Three ways to construct a server

# Summarising “Why Threads?”

- Simpler to program than a state machine
- Less resources are associated with them than multiple complete processes
  - Cheaper to create and destroy
  - Shares resources (especially memory) between them
- Performance: Threads waiting for I/O can be overlapped with computing threads
  - Note if all threads are *compute bound*, then there is no performance improvement (on a uniprocessor)
- Threads can take advantage of the parallelism available on machines with more than one CPU (multiprocessor)

# Concurrency and Synchronisation

# Learning Outcomes

- Understand concurrency is an issue in operating systems and multithreaded applications
- Know the concept of a *critical region*.
- Understand how mutual exclusion of critical regions can be used to solve concurrency issues
  - Including how mutual exclusion can be implemented correctly and efficiently.
- Be able to identify and solve a *producer consumer bounded buffer* problem.
- Understand and apply standard synchronisation primitives to solve synchronisation problems.

# Textbook

- Sections 2.3 - 2.3.7 & 2.5

# Concurrency Example

count is a global variable shared between two threads, t is a local variable.  
After increment and decrement complete, what is the value of count?

```
void increment ()  
{
```

```
    int t;  
  
    t = count;  
    t = t + 1;  
    count = t;
```

```
}
```

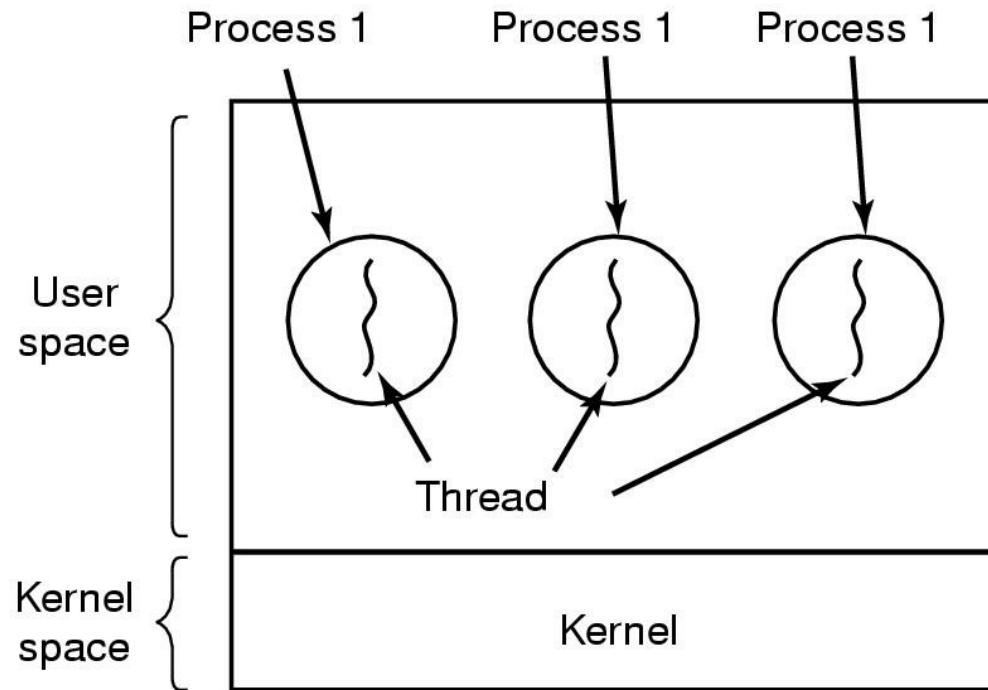
```
void decrement ()  
{
```

```
    int t;  
  
    t = count;  
    t = t - 1;  
    count = t;
```

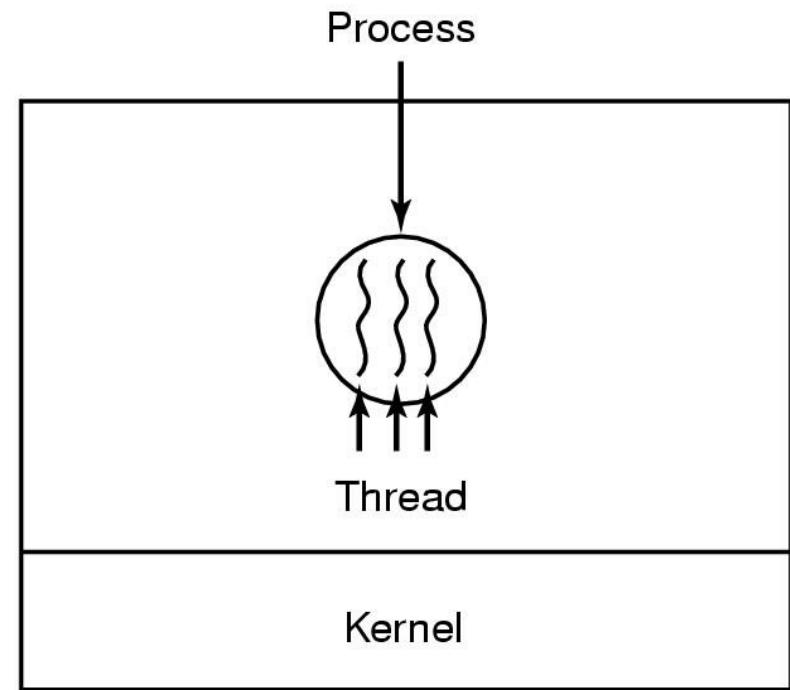
```
}
```

We have a  
*race  
condition*

# Where is the concurrency?



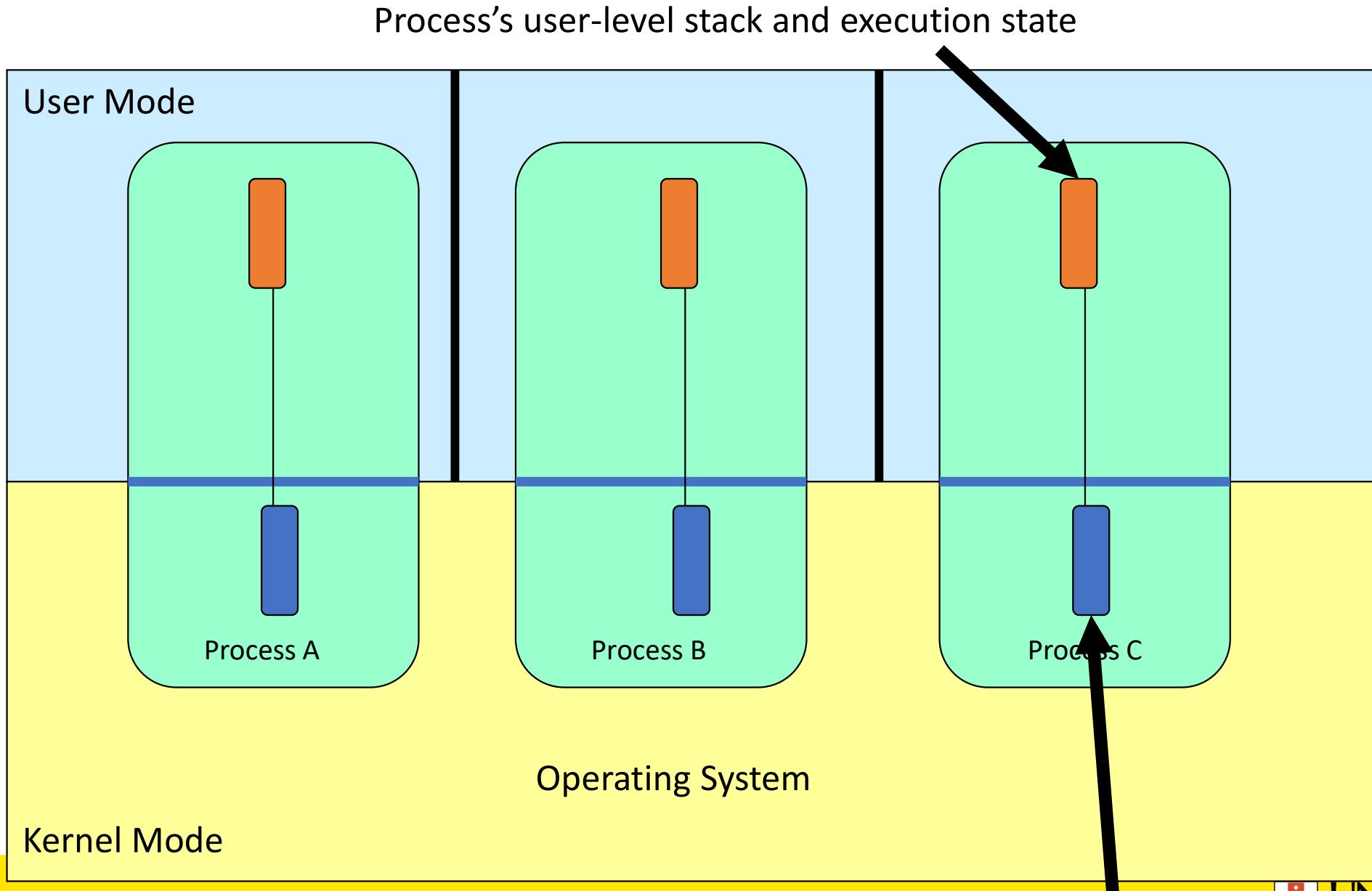
(a)



(b)

- (a) Three processes each with one thread
- (b) One process with three threads

# There is in-kernel concurrency even for single-threaded processes



# Critical Region

- We can control access to the shared resource by controlling access to the code that accesses the resource.  
⇒ A *critical region* is a region of code where shared resources are accessed.
  - Variables, memory, files, etc...
- Uncoordinated entry to the critical region results in a race condition  
⇒ Incorrect behaviour, deadlock, lost work,...

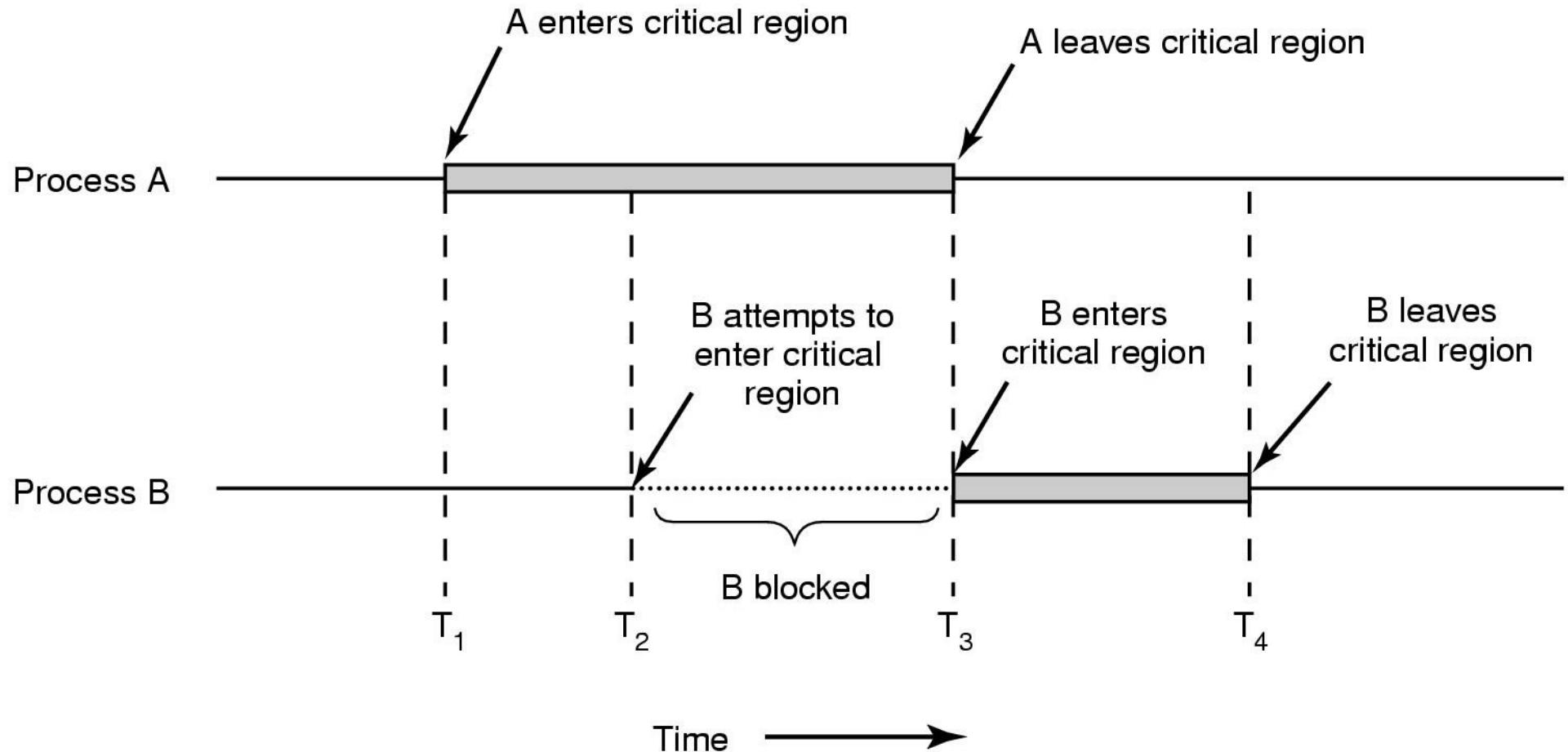
# Identifying critical regions

- Critical regions are regions of code that:
  - Access a shared resource,
  - and correctness relies on the shared resource not being concurrently modified by another thread/process/entity.

```
void increment ()  
{  
    int t;  
    t = count;  
    t = t + 1;  
    count = t;  
}
```

```
void decrement ()  
{  
    int t;  
    t = count;  
    t = t - 1;  
    count = t;  
}
```

# Accessing Critical Regions



Mutual exclusion using critical regions

# Example critical regions

```
struct node {  
    int data;  
    struct node *next;  
};  
struct node *head;
```

```
void init(void)  
{  
    head = NULL;  
}
```

- Simple last-in-first-out queue implemented as a linked list.

```
void insert(struct *item)  
{  
    item->next = head;  
    head = item;  
}
```

```
struct node *remove(void)  
{  
    struct node *t;  
    t = head;  
    if (t != NULL) {  
        head = head->next;  
    }  
    return t;  
}
```

# Example Race

```
void insert(struct *item)
{
    item->next = head;
    head = item;
}
```

```
void insert(struct *item)
{
    item->next = head;
    head = item;
}
```

# Example critical regions

```
struct node {  
    int data;  
    struct node *next;  
};  
struct node *head;
```

```
void init(void)  
{  
    head = NULL;  
}
```

- Critical sections

```
void insert(struct *item)  
{  
    item->next = head;  
    head = item;  
}
```

```
struct node *remove(void)  
{  
    struct node *t;  
    t = head;  
    if (t != NULL) {  
        head = head->next;  
    }  
    return t;  
}
```

# Critical Regions Solutions

- We seek a solution to coordinate access to critical regions.
  - Also called critical sections
- Conditions required of any solution to the critical region problem
  1. Mutual Exclusion:
    - No two processes simultaneously in critical region
  2. No assumptions made about speeds or numbers of CPUs
  3. Progress
    - No process running outside its critical region may block another process
  4. Bounded
    - No process waits forever to enter its critical region

# A solution?

- A lock variable
  - If  $\text{lock} == 1$ ,
    - somebody is in the critical section and we must wait
  - If  $\text{lock} == 0$ ,
    - nobody is in the critical section and we are free to enter

# A solution?

```
while(TRUE) {  
    while(lock == 1);  
    lock = 1;  
    critical();  
    lock = 0  
    non_critical();  
}
```

```
while(TRUE) {  
    while(lock == 1);  
    lock = 1;  
    critical();  
    lock = 0  
    non_critical();  
}
```

# A problematic execution sequence

```
while(TRUE) {  
    while(lock == 1);  
    lock = 1;  
    critical();  
    lock = 0  
    non_critical();  
}
```

```
while(TRUE) {  
    while(lock == 1);  
    lock = 1;  
    critical();  
    lock = 0  
    non_critical();  
}
```



# Observation

- Unfortunately, it is usually easier to show something does not work, than it is to prove that it does work.
  - Easier to provide a counter example
  - Ideally, we'd like to prove, or at least informally demonstrate, that our solutions work.

# Mutual Exclusion by Taking Turns

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(b)

Proposed solution to critical region problem

(a) Process 0.      (b) Process 1.

# Mutual Exclusion by Taking Turns

- Works due to *strict alternation*
  - Each process takes turns
- Cons
  - Busy waiting
  - Process must wait its turn even while the other process is doing something else.
    - With many processes, must wait for everyone to have a turn
      - Does not guarantee progress if a process no longer needs a turn.
    - Poor solution when processes require the critical section at differing rates

# Mutual Exclusion by Disabling Interrupts

- Before entering a critical region, disable interrupts
- After leaving the critical region, enable interrupts
- Pros
  - simple
- Cons
  - Only available in the kernel
  - Delays everybody else, even with no contention
    - Slows interrupt response time
  - Does not work on a multiprocessor

# Hardware Support for mutual exclusion

- Test and set instruction
  - Can be used to implement lock variables correctly
    - It loads the value of the lock
    - If lock == 0,
      - set the lock to 1
      - return the result 0 – we acquire the lock
    - If lock == 1
      - return 1 – another thread/process has the lock
  - Hardware guarantees that the instruction executes atomically.
    - Atomically: As an indivisible unit.

# Mutual Exclusion with Test-and-Set

enter\_region:

TSL REGISTER,LOCK	copy lock to register and set lock to 1
CMP REGISTER,#0	was lock zero?
JNE enter_region	if it was non zero, lock was set, so loop
RET   return to caller; critical region entered	

leave\_region:

MOVE LOCK,#0	store a 0 in lock
RET   return to caller	

Entering and leaving a critical region using the  
TSL instruction

# Test-and-Set

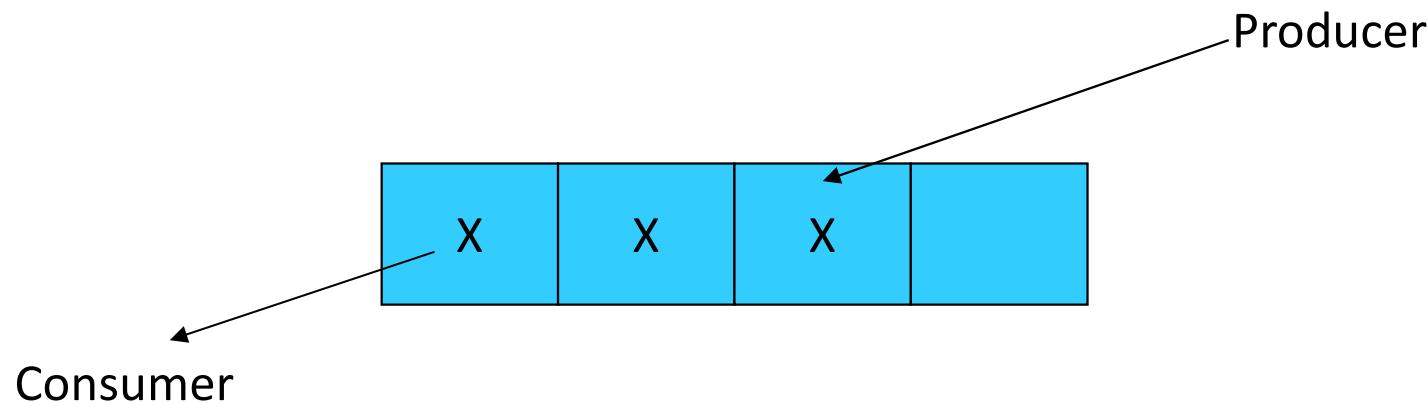
- Pros
  - Simple (easy to show it's correct)
  - Available at user-level
    - To any number of processors
    - To implement any number of lock variables
- Cons
  - Busy waits (also termed a *spin lock*)
    - Consumes CPU
    - Starvation is possible when a process leaves its critical section and more than one process is waiting.

# Tackling the Busy-Wait Problem

- Sleep / Wakeup
  - The idea
    - When process is waiting for an event, it calls sleep to block, instead of busy waiting.
    - The event happens, the event generator (another process) calls wakeup to unblock the sleeping process.
    - Waking a ready/running process has no effect.

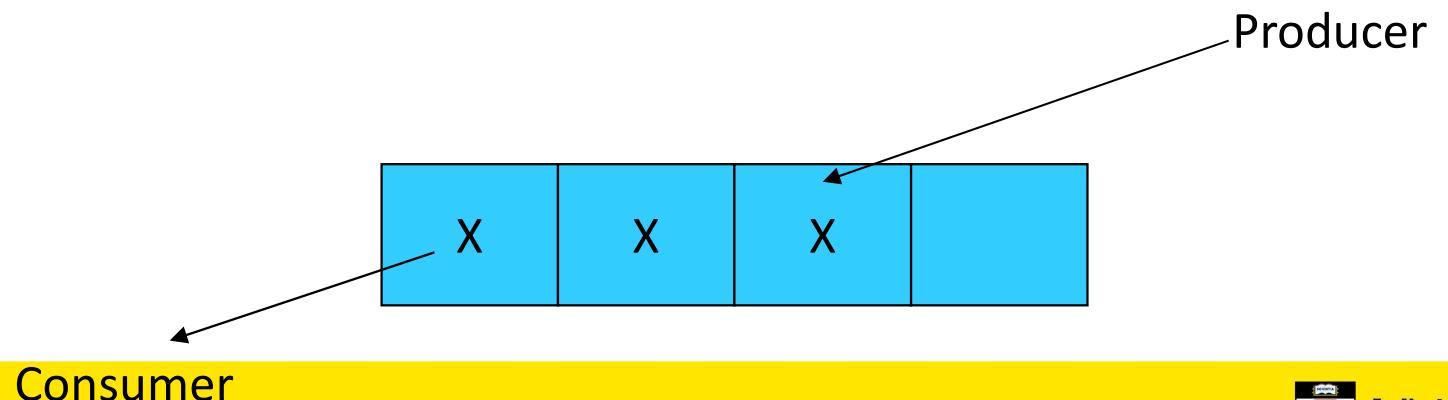
# The Producer-Consumer Problem

- Also called the *bounded buffer* problem
- A producer produces data items and stores the items in a buffer
- A consumer takes the items out of the buffer and consumes them.



# Issues

- We must keep an accurate count of items in buffer
  - Producer
    - should sleep when the buffer is full,
    - and wakeup when there is empty space in the buffer
      - The consumer can call wakeup when it consumes the first entry of the full buffer
  - Consumer
    - should sleep when the buffer is empty
    - and wake up when there are items available
      - Producer can call wakeup when it adds the first item to the buffer



# Pseudo-code for producer and consumer

```
int count = 0;                                con() {  
#define N 4 /* buf size */  
prod() {  
    while(TRUE) {  
        item = produce()  
        if (count == N)  
            sleep(prod);  
        insert_item();  
        count++;  
        if (count == 1)  
            wakeup(con);  
    }  
}  
}  
while(TRUE) {  
    if (count == 0)  
        sleep(con);  
    remove_item();  
    count--;  
    if (count == N-1)  
        wakeup(prod);  
}
```

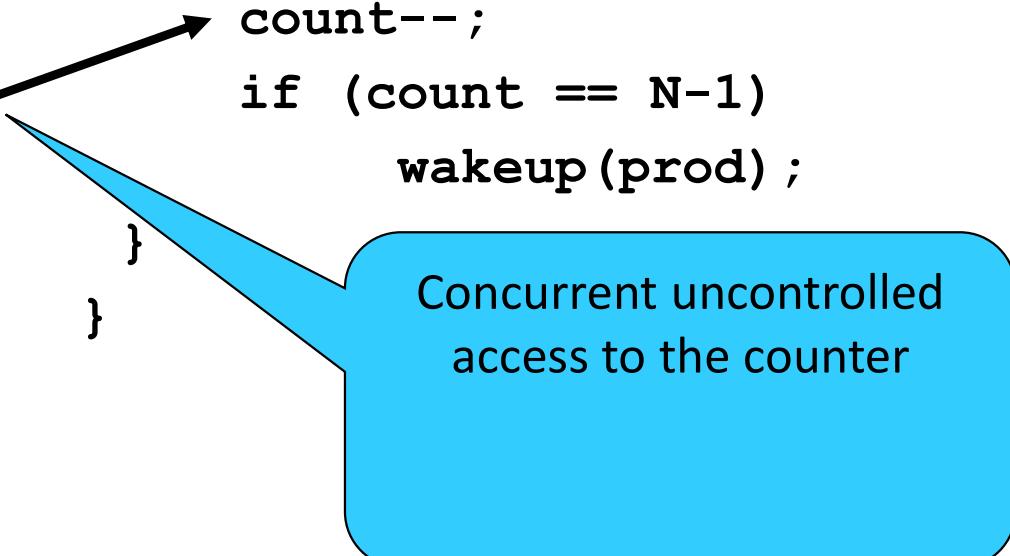
# Problems

```
int count = 0;                                con() {  
#define N 4 /* buf size */  
prod() {  
    while(TRUE) {  
        item = produce()  
        if (count == N)  
            sleep(prod);  
        insert_item();  
        count++;  
        if (count == 1)  
            wakeup(con);  
    }  
}  
while(TRUE) {  
    if (count == 0)  
        sleep(con);  
    remove_item();  
    count--;  
    if (count == N-1)  
        wakeup(prod);  
}
```

Concurrent uncontrolled access to the buffer

# Problems

```
int count = 0;                                con() {  
#define N 4 /* buf size */  
prod() {  
    while(TRUE) {  
        item = produce()  
        if (count == N)  
            sleep(prod);  
        insert_item();  
        count++;  
        if (count == 1)  
            wakeup(con);  
    }  
}  
while(TRUE) {  
    if (count == 0)  
        sleep(con);  
    remove_item();  
    count--;  
    if (count == N-1)  
        wakeup(prod);  
}
```



Concurrent uncontrolled access to the counter

# Proposed Solution

- Lets use a locking primitive based on test-and-set to protect the concurrent access

# Proposed solution?

```
int count = 0;
lock_t buf_lock;
#define N 4 /* buf size */
prod() {
    while(TRUE) {
        item = produce()
        if (count == N)
            sleep(prod);
        acquire_lock(buf_lock)
        insert_item();
        count++;
        release_lock(buf_lock)
        if (count == 1)
            wakeup(con);
    }
}

con() {
    while(TRUE) {
        if (count == 0)
            sleep(con);
        acquire_lock(buf_lock)
        remove_item();
        count--;
        release_lock(buf_lock);
        if (count == N-1)
            wakeup(prod);
    }
}
```

# Problematic execution sequence

```
prod() {  
    while(TRUE) {  
        item = produce()  
        if (count == N)  
            sleep(prod);  
        acquire_lock(buf_lock)  
        insert_item();  
        count++;  
        release_lock(buf_lock)  
        if (count == 1)  
            wakeup(con);  
    }  
}
```

wakeup without a matching sleep is lost

```
    sleep(con);  
    acquire_lock(buf_lock)  
    remove_item();  
    count--;  
    release_lock(buf_lock);  
    if (count == N-1)  
        wakeup(prod);  
    }  
}
```

# Problem

- The test for *some condition* and actually going to sleep needs to be atomic
- The following does not work:

```
acquire_lock(buf_lock)
if (count == N)
    sleep();
release_lock(buf_lock)
```

The lock is held while asleep  
⇒ count will never change

```
acquire_lock(buf_lock)
if (count == 1)
    wakeup();
release_lock(buf_lock)
```

# Semaphores

- Dijkstra (1965) introduced two primitives that are more powerful than simple sleep and wakeup alone.
  - P(): *proberen*, from Dutch *to test*.
  - V(): *verhogen*, from Dutch *to increment*.
  - Also called *wait & signal, down & up*.

# How do they work

- If a resource is not available, the corresponding semaphore blocks any process **waiting** for the resource
- Blocked processes are put into a process queue maintained by the semaphore (avoids busy waiting!)
- When a process releases a resource, it **signals** this by means of the semaphore
- Signalling resumes a blocked process if there is any
- Wait (P) and signal (V) operations cannot be interrupted
- Complex coordination can be implemented by multiple semaphores

# Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {  
    int count;  
    struct process *L;  
} semaphore;
```

- Assume two simple operations:
  - **sleep** suspends the process that invokes it.
  - **wakeup(*P*)** resumes the execution of a blocked process **P**.

- Semaphore operations now defined as

*wait(S):*

```
S.count--;
if (S.count < 0) {
    add this process to S.L;
    sleep;
}
```

*signal(S):*

```
S.count++;
if (S.count <= 0) {
    remove a process P from S.L;
    wakeup(P);
}
```

- Each primitive is atomic

- E.g. interrupts are disabled for each

# Semaphore as a General Synchronization Tool

- Execute  $B$  in  $P_j$  only after  $A$  executed in  $P_i$
- Use semaphore *count* initialized to 0
- Code:

$P_i$	$P_j$
⋮	⋮
$A$	$wait(flag)$
$signal(flag)$	$B$

# Semaphore Implementation of a Mutex

- Mutex is short for Mutual Exclusion
  - Can also be called a lock

```
semaphore mutex;  
mutex.count = 1; /* initialise mutex */
```

```
wait(mutex); /* enter the critical region */
```

```
Blahblah();
```

```
signal(mutex); /* exit the critical region */
```

Notice that the initial count determines how many waits can progress before blocking and requiring a signal  $\Rightarrow$  mutex.count initialised as 1

# Solving the producer-consumer problem with semaphores

```
#define N = 4

semaphore mutex = 1;

/* count empty slots */
semaphore empty = N;

/* count full slots */
semaphore full = 0;
```

# Solving the producer-consumer problem with semaphores

```
prod() {                                con() {  
    while(TRUE) {  
        item = produce()  
        wait(empty);  
        wait(mutex);  
        insert_item();  
        signal(mutex);  
        signal(full);  
    }  
}  
  
while(TRUE) {  
    wait(full);  
    wait(mutex);  
    remove_item();  
    signal(mutex);  
    signal(empty);  
}
```

# Summarising Semaphores

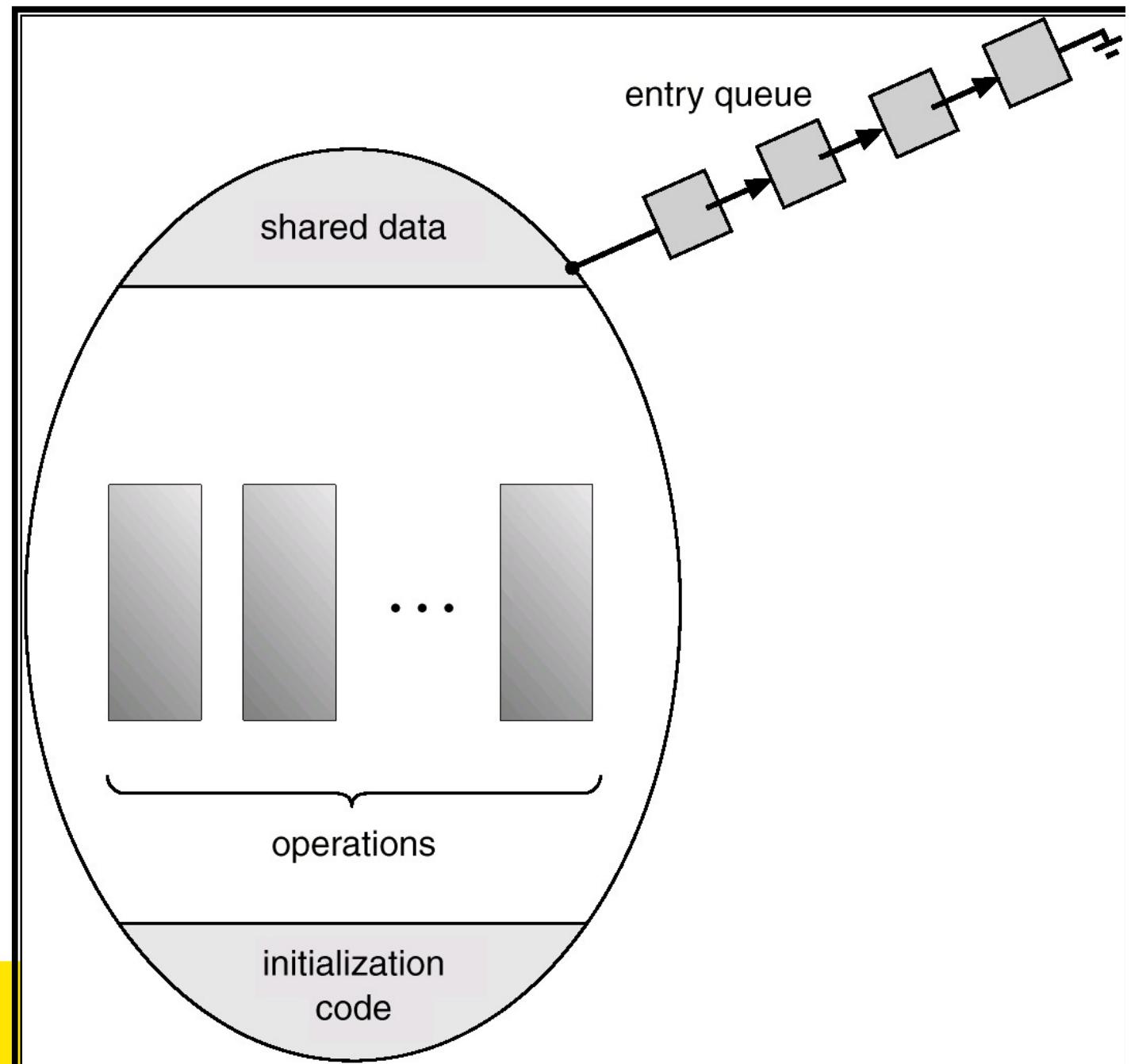
- Semaphores can be used to solve a variety of concurrency problems
- However, programming with them can be error-prone
  - E.g. must *signal* for every *wait* for mutexes
    - Too many, or too few signals or waits, or signals and waits in the wrong order, can have catastrophic results

# Monitors

- To ease concurrent programming, Hoare (1974) proposed *monitors*.
  - A higher level synchronisation primitive
  - Programming language construct
- Idea
  - A set of procedures, variables, data types are grouped in a special kind of module, a *monitor*.
    - Variables and data types only accessed from within the monitor
  - Only one process/thread can be in the monitor at any one time
    - Mutual exclusion is implemented by the compiler (which should be less error prone)

# Monitor

- When a thread calls a monitor procedure that has a thread already inside, it is queued and it sleeps until the current thread exits the monitor.



# Monitors

**monitor** *example*

**integer** *i*;

**condition** *c*;

**procedure** *producer()*;

        .

        .

        .

**end;**

**procedure** *consumer()*;

        .

        .

        .

**end;**

**end monitor;**

Example of a monitor

# Simple example

```
monitor counter {  
    int count;  
    procedure inc() {  
        count = count + 1;  
    }  
    procedure dec() {  
        count = count -1;  
    }  
}
```

Note: “paper” language

- Compiler guarantees only one thread can be active in the monitor at any one time
- Easy to see this provides mutual exclusion
  - No race condition on **count**.

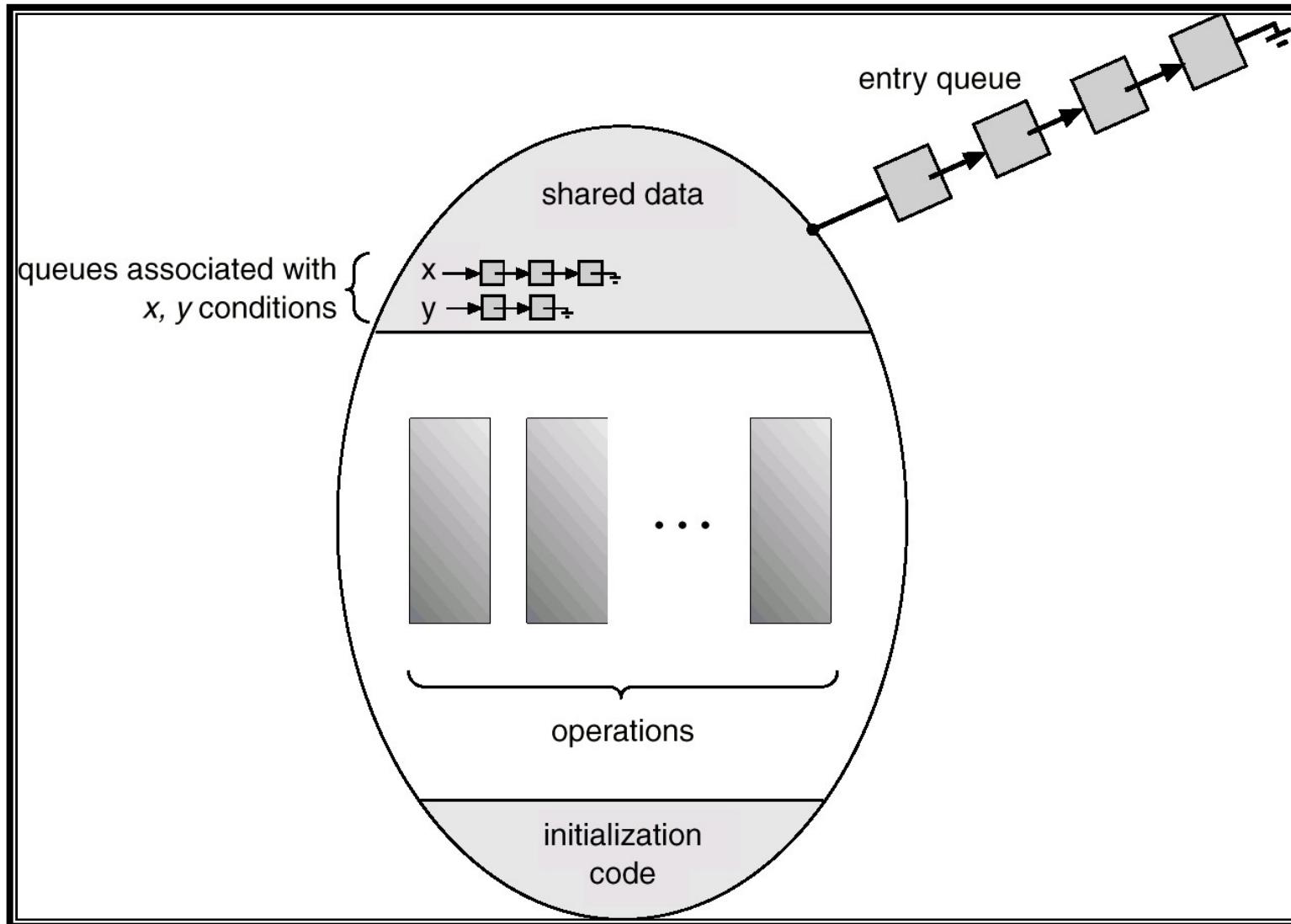
# How do we block waiting for an event?

- We need a mechanism to block waiting for an event (in addition to ensuring mutual exclusion)
  - e.g., for producer consumer problem when buffer is empty or full
- *Condition Variables*

# Condition Variable

- To allow a process to wait within the monitor, a **condition** variable must be declared, as  
**condition x, y;**
- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation  
**x.wait();**
    - means that the process invoking this operation is suspended until another process invokes
    - Another thread can enter the monitor while original is suspended
  - The **x.signal()** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

# Condition Variables



# Monitors

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has  $N$  slots

# OS/161 Provided Synchronisation

## Primitives

- Locks
- Semaphores
- Condition Variables

# Locks

- Functions to create and destroy locks

```
struct lock *lock_create(const char *name);  
void          lock_destroy(struct lock *);
```

- Functions to acquire and release them

```
void          lock_acquire(struct lock *);  
void          lock_release(struct lock *);
```

# Example use of locks

```
int count;  
  
struct lock *count_lock  
  
main() {  
    count = 0;  
    count_lock =  
        lock_create("count  
lock");  
    if (count_lock == NULL)  
        panic("I'm dead");  
    stuff();  
}
```

```
procedure inc() {  
    lock_acquire(count_lock);  
    count = count + 1;  
    lock_release(count_lock);  
}  
  
procedure dec() {  
    lock_acquire(count_lock);  
    count = count -1;  
    lock_release(count_lock);  
}
```

# Semaphores

```
struct semaphore *sem_create(const char *name, int
                             initial_count);
void               sem_destroy(struct semaphore *);

void              P(struct semaphore *);
void              V(struct semaphore *);
```

# Example use of Semaphores

```
int count;  
  
struct semaphore  
 *count_mutex;  
  
main() {  
    count = 0;  
    count_mutex =  
        sem_create("count",  
                   1);  
    if (count_mutex == NULL)  
        panic("I'm dead");  
    stuff();  
}  
  
procedure inc() {  
    P(count_mutex);  
    count = count + 1;  
    V(count_mutex);  
}  
  
procedure dec() {  
    P(count_mutex);  
    count = count -1;  
    V(count_mutex);  
}
```

# Condition Variables

```
struct cv *cv_create(const char *name);  
void cv_destroy(struct cv *);
```

```
void cv_wait(struct cv *cv, struct lock *lock);
```

- Releases the lock and blocks
- Upon resumption, it re-acquires the lock
  - Note: we must recheck the condition we slept on

```
void cv_signal(struct cv *cv, struct lock *lock);
```

```
void cv_broadcast(struct cv *cv, struct lock *lock);
```

- Wakes one/all, does not release the lock
- First “waiter” scheduled after signaller releases the lock will re-acquire the lock

Note: All three variants must hold the lock passed in.

# Condition Variables and Bounded Buffers

## Non-solution

```
lock_acquire(c_lock)
if (count == 0)
    sleep();
remove_item();
count--;
lock_release(c_lock)
;
```

## Solution

```
lock_acquire(c_lock)
while (count == 0)
    cv_wait(c_cv, c_lock);
remove_item();
count--;
lock_release(c_lock);
```

# Alternative Producer-Consumer Solution Using OS/161 CVs

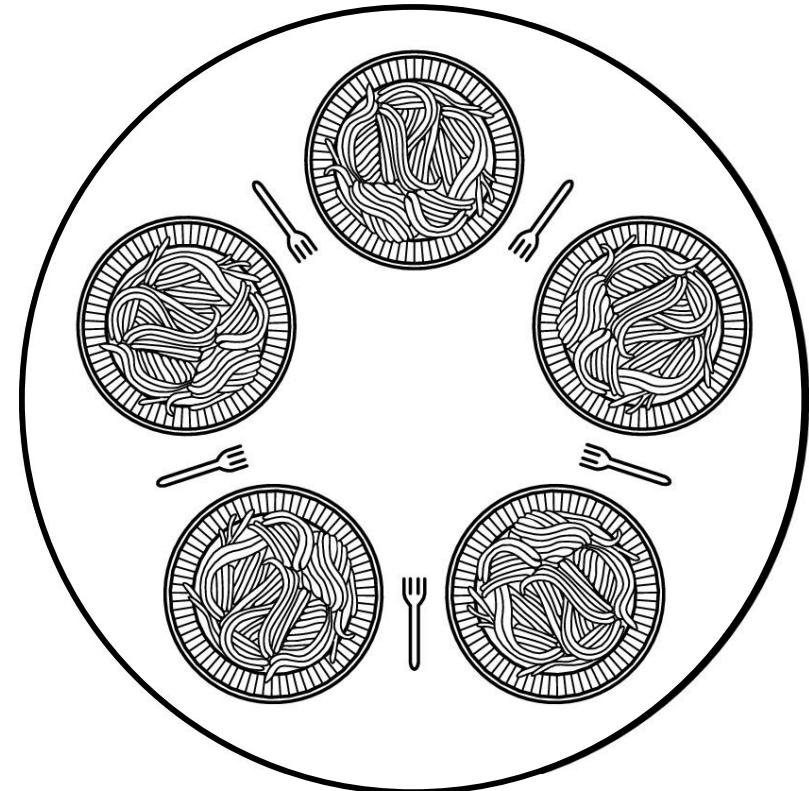
```
int count = 0;
#define N 4 /* buf size */

prod() {
    while(TRUE) {
        item = produce()
        lock_acquire(1)
        while (count == N)
            cv_wait(full,1);
        insert_item(item);
        count++;
        cv_signal(empty,1);
        lock_release(1)
    }
}

con() {
    while(TRUE) {
        lock_acquire(1)
        while (count == 0)
            cv_wait(empty,1);
        item = remove_item();
        count--;
        cv_signal(full,1);
        lock_release(1)
        consume(item);
    }
}
```

# Dining Philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



# Dining Philosophers

```
#define N          5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N    /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

/\* number of philosophers \*/  
/\* number of i's left neighbor \*/  
/\* number of i's right neighbor \*/  
/\* philosopher is thinking \*/  
/\* philosopher is trying to get forks \*/  
/\* philosopher is eating \*/  
/\* semaphores are a special kind of int \*/  
/\* array to keep track of everyone's state \*/  
/\* mutual exclusion for critical regions \*/  
/\* one semaphore per philosopher \*/  
  
/\* i: philosopher number, from 0 to N-1 \*/  
  
/\* repeat forever \*/  
/\* philosopher is thinking \*/  
/\* acquire two forks or block \*/  
/\* yum-yum, spaghetti \*/  
/\* put both forks back on table \*/

Solution to dining philosophers problem (part 1)

# Dining Philosophers

```
#define N 5                                     /* number of philosophers */  
  
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */  
{  
    while (TRUE) {  
        think();                                /* philosopher is thinking */  
        take_fork(i);                            /* take left fork */  
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */  
        eat();                                   /* yum-yum, spaghetti */  
        put_fork(i);                            /* put left fork back on the table */  
        put_fork((i+1) % N);                   /* put right fork back on the table */  
    }  
}
```

A nonsolution to the dining philosophers problem

# Dining Philosophers

```
void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */  
{  
    down(&mutex);  
    state[i] = HUNGRY;  
    test(i);  
    up(&mutex);  
    down(&s[i]);  
}  
  
void put_forks(i)                                    /* i: philosopher number, from 0 to N-1 */  
{  
    down(&mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    up(&mutex);  
}  
  
void test(i)                                         /* i: philosopher number, from 0 to N-1 */  
{  
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {  
        state[i] = EATING;  
        up(&s[i]);  
    }  
}
```

Solution to dining philosophers problem (part 2)



# The Readers and Writers Problem

- Models access to a database
  - E.g. airline reservation system
  - Can have more than one concurrent reader
    - To check schedules and reservations
  - Writers must have exclusive access
    - To book a ticket or update a schedule

# The Readers and Writers Problem

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

/* use your imagination */
/* controls access to 'rc' */
/* controls access to the database */
/* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

/* repeat forever */
/* get exclusive access to 'rc' */
/* one reader more now */
/* if this is the first reader ... */
/* release exclusive access to 'rc' */
/* access the data */
/* get exclusive access to 'rc' */
/* one reader fewer now */
/* if this is the last reader ... */
/* release exclusive access to 'rc' */
/* noncritical region */

void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}

/* repeat forever */
/* noncritical region */
/* get exclusive access */
/* update the data */
/* release exclusive access */
```

A solution to the readers and writers problem

# Chapter 6

## Deadlocks

- 6.1. Resources
- 6.2. Introduction to deadlocks
- 6.3. The ostrich algorithm
- 6.6. Deadlock prevention
- 6.4. Deadlock detection and recovery
- 6.5. Deadlock avoidance
- 6.7. Other issues

# Learning Outcomes

- Understand what deadlock is and how it can occur when giving mutually exclusive access to multiple resources.
- Understand several approaches to mitigating the issue of deadlock in operating systems.
  - Including deadlock *prevention, detection and recovery*, and deadlock *avoidance*.

# Resources

- Examples of computer resources
  - printers
  - tape drives
  - Tables in a database
- Processes need access to resources in reasonable order
- Preemptable resources
  - can be taken away from a process with no ill effects
- Nonpreemptable resources
  - will cause the process to fail if taken away

# Resources & Deadlocks

- Suppose a process holds resource A and requests resource B
  - at same time another process holds B and requests A
  - both are blocked and remain so - *Deadlocked*
- Deadlocks occur when ...
  - processes are granted exclusive access to devices, **locks**, tables, etc..
  - we refer to these entities generally as resources

# Resource Access

- Sequence of events required to use a resource
  1. request the resource
  2. use the resource
  3. release the resource
- Must wait if request is denied
  - requesting process may be blocked
  - may fail with error code

# Two example resource usage patterns

```
semaphore res_1, res_2;  
void proc_A() {  
    down(&res_1);  
    down(&res_2);  
    use_both_res();  
    up(&res_2);  
    up(&res_1);  
}  
  
void proc_B() {  
    down(&res_1);  
    down(&res_2);  
    use_both_res();  
    up(&res_2);  
    up(&res_1);  
}
```

```
semaphore res_1, res_2;  
void proc_A() {  
    down(&res_1);  
    down(&res_2);  
    use_both_res();  
    up(&res_2);  
    up(&res_1);  
}  
  
void proc_B() {  
    down(&res_2);  
    down(&res_1);  
    use_both_res();  
    up(&res_1);  
    up(&res_2);  
}
```

# Introduction to Deadlocks

- Formal definition :

*A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause*

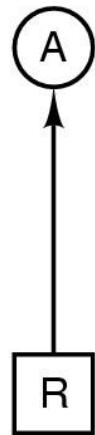
- Usually the event is release of a currently held resource
- None of the processes can ...
  - run
  - release resources
  - be awakened

# Four Conditions for Deadlock

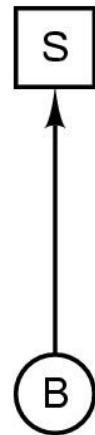
1. Mutual exclusion condition
  - each resource assigned to 1 process or is available
2. Hold and wait condition
  - process holding resources can request additional
3. No preemption condition
  - previously granted resources cannot be forcibly taken away
4. Circular wait condition
  - must be a circular chain of 2 or more processes
  - each is waiting for resource held by next member of the chain

# Deadlock Modeling

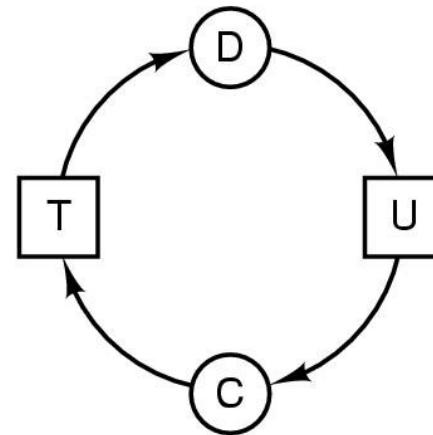
- Modeled with directed graphs



(a)



(b)



(c)

- resource R assigned to process A
- process B is requesting/waiting for resource S
- process C and D are in deadlock over resources T and U

# Deadlock Modeling

A

Request R  
Request S  
Release R  
Release S

(a)

B

Request S  
Request T  
Release S  
Release T

(b)

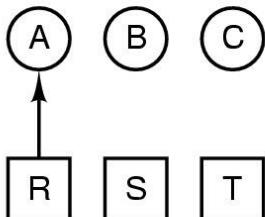
C

Request T  
Request R  
Release T  
Release R

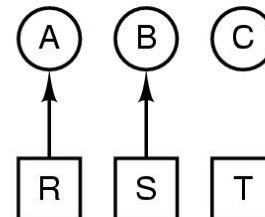
(c)

1. A requests R
  2. B requests S
  3. C requests T
  4. A requests S
  5. B requests T
  6. C requests R
- deadlock

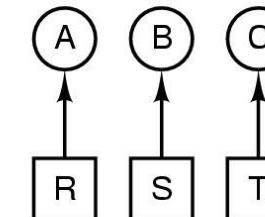
(d)



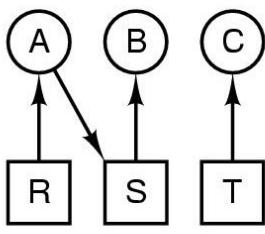
(e)



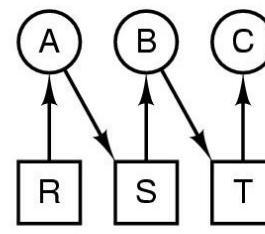
(f)



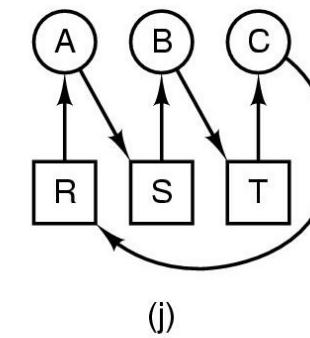
(g)



(h)



(i)



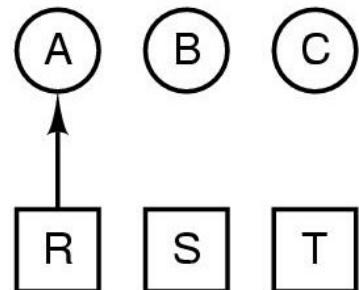
(j)

How deadlock occurs

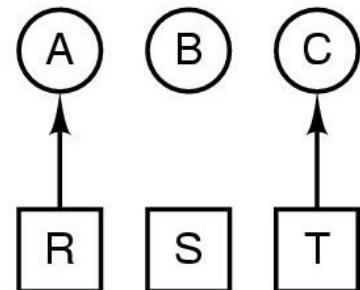
# Deadlock Modeling

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
- no deadlock

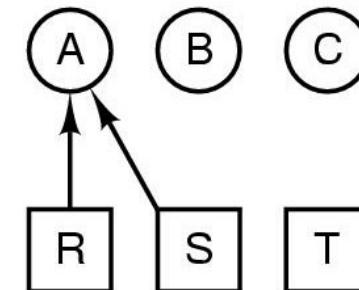
(k)



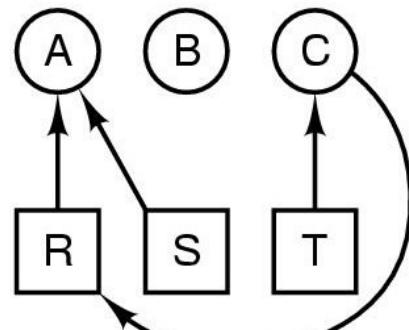
(l)



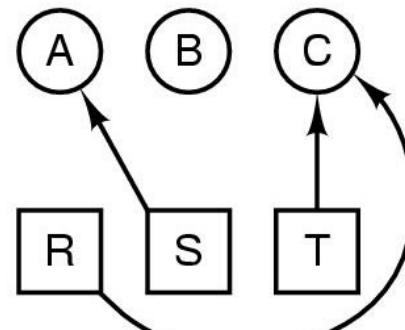
(m)



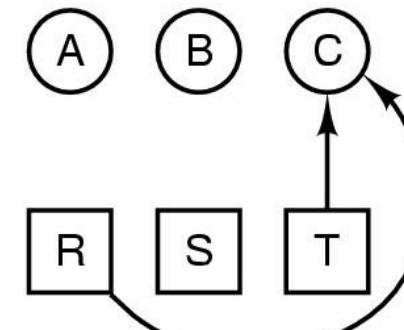
(n)



(o)



(p)



(q)

How deadlock can be avoided

# Deadlock

## Strategies for dealing with Deadlocks

1. just ignore the problem altogether
2. prevention
  - negating one of the four necessary conditions
3. detection and recovery
4. dynamic avoidance
  - careful resource allocation

# Approach 1: The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
    - Example of “cost”, only one process runs at a time
- UNIX and Windows takes this approach for some of the more complex resource relationships they manage
- It’s a trade off between
  - Convenience (engineering approach)
  - Correctness (mathematical approach)

# Approach 2: Deadlock Prevention

- Resource allocation rules prevent deadlock by prevent one of the four conditions required for deadlock from occurring
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular Wait

# Approach 2

## Deadlock Prevention

### Attacking the Mutual Exclusion Condition

- Not feasible in general
  - Some devices/resource are intrinsically not shareable.

# Attacking the Hold and Wait Condition

- Require processes to request resources before starting
  - a process never has to wait for what it needs
- Issues
  - may not know required resources at start of run
    - ⇒ not always possible
  - also ties up resources other processes could be using
- Variations:
  - process must give up all resources if it would block holding a resource
  - then request all immediately needed
  - prone to livelock

# Livelock

- Livelocked processes are not blocked, change state regularly, but never make progress.
- Example: Two people passing each other in a corridor that attempt to step out of each other's way in the same direction, indefinitely.
  - Both are actively changing state
  - Both never pass each other.

# Deadlock example

```
void proc_A() {  
    lock_acquire(&res_1);  
    lock_acquire(&res_2);  
    use_both_res();  
    lock_release(&res_2);  
    lock_release(&res_1);  
}
```

```
void proc_B() {  
    lock_acquire(&res_2);  
    lock_acquire(&res_1);  
    use_both_res();  
    lock_release(&res_1);  
    lock_release(&res_2);  
}
```

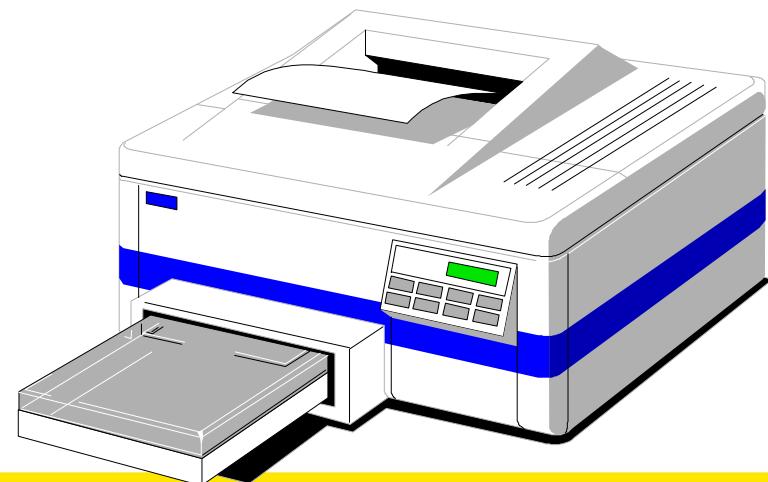
# Livelock example

```
void proc_A() {  
    lock_acquire(&res_1);  
    while(try_lock(&res_2) == FAIL) {  
        lock_release(&res_1);  
        wait_fixed_time();  
        lock_acquire(&res_1);  
    }  
    use_both_res();  
    lock_release(&res_2);  
    lock_release(&res_1);  
}
```

```
void proc_B() {  
    lock_acquire(&res_2);  
    while(try_lock(&res_1) == FAIL) {  
        lock_release(&res_2);  
        wait_fixed_time();  
        lock_acquire(&res_2);  
    }  
    use_both_res();  
    lock_release(&res_1);  
    lock_release(&res_2);  
}
```

# Attacking the No Preemption Condition

- This is not a viable option
- Consider a process given the printer
  - halfway through its job
  - now forcibly take away printer
  - !!??



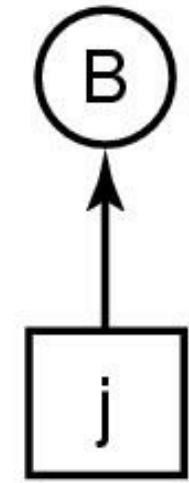
# Attacking the Circular Wait Condition

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

(a)



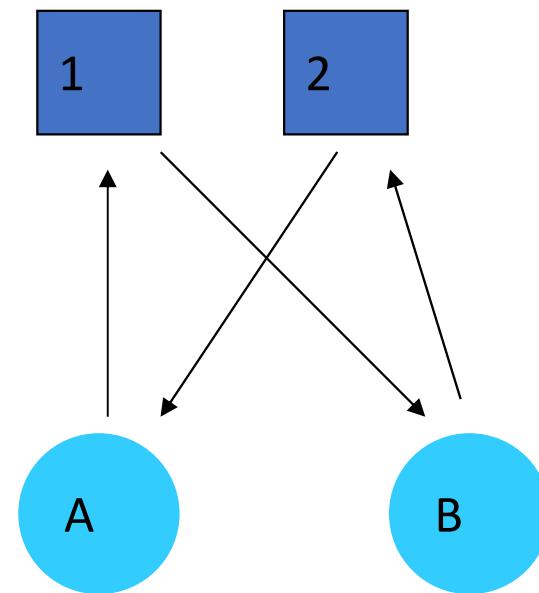
(b)



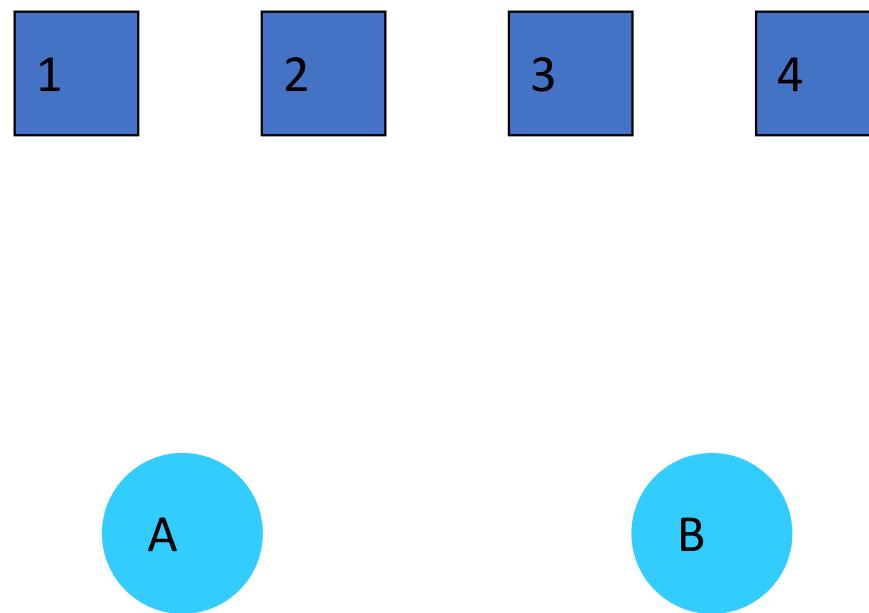
- Numerically ordered resources

# Attacking the Circular Wait Condition

- The displayed deadlock cannot happen
  - If A requires **1**, it must acquire it before acquiring **2**
  - Note: If B has **1**, all higher numbered resources must be free or held by processes who doesn't need **1**
- Resources ordering is a common technique in practice!!!!



# Example



# Summary of approaches to deadlock prevention

## Condition

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

## Approach

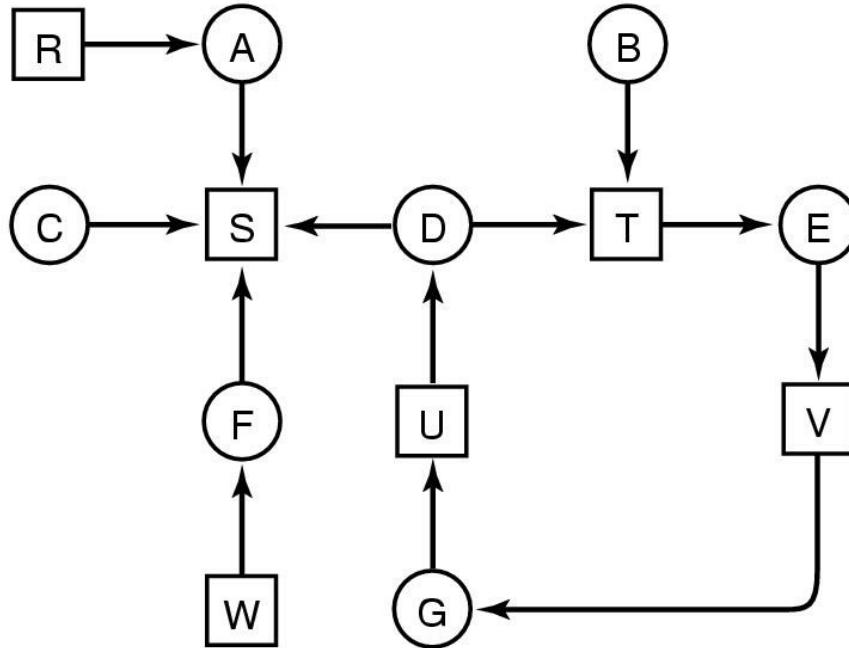
- Not feasible
- Request resources initially
- Take resources away
- Order resources

# Approach 3: Detection and Recovery

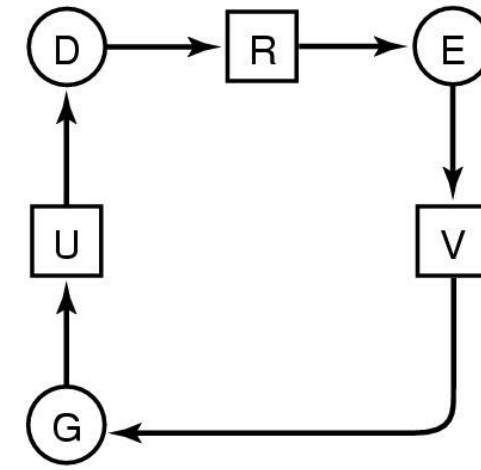
- Need a method to determine if a system is deadlocked.
- Assuming deadlocked is detected, we need a method of recovery to restore progress to the system.

# Approach 3

## Detection with One Resource of Each Type



(a)



(b)

- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock

# What about resources with multiple units?

- Some examples of multi-unit resources
  - RAM
  - Blocks on a hard disk drive
  - Slots in a buffer
- We need an approach for dealing with resources that consist of more than a single unit.

# Detection with Multiple Resources of Each Type

Resources in existence  
( $E_1, E_2, E_3, \dots, E_m$ )

Current allocation matrix

$C_{11}$	$C_{12}$	$C_{13}$	$\dots$	$C_{1m}$
$C_{21}$	$C_{22}$	$C_{23}$	$\dots$	$C_{2m}$
:	:	:		:
$C_{n1}$	$C_{n2}$	$C_{n3}$	$\dots$	$C_{nm}$

A curved arrow points from the bottom-left corner of the matrix to the text below it.

Row  $n$  is current allocation  
to process  $n$

Resources available  
( $A_1, A_2, A_3, \dots, A_m$ )

Request matrix

$R_{11}$	$R_{12}$	$R_{13}$	$\dots$	$R_{1m}$
$R_{21}$	$R_{22}$	$R_{23}$	$\dots$	$R_{2m}$
:	:	:		:
$R_{n1}$	$R_{n2}$	$R_{n3}$	$\dots$	$R_{nm}$

A curved arrow points from the top-right corner of the matrix to the text below it.

Row 2 is what process 2 needs

Data structures needed by deadlock detection algorithm

# Note the following invariant

Sum of current resource allocation + resources available =  
resources that exist

$$\sum_{i=1}^n C_{ij} + A_j = E_j$$

# Detection with Multiple Resources of Each Type

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Tape drives  
Plotters  
Scanners  
CD Roms

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Tape drives  
Plotters  
Scanners  
CD Roms

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

An example for the deadlock detection algorithm

# Detection Algorithm

1. Look for an unmarked process  $P_i$ , for which the  $i$ -th row of R is less than or equal to A
2. If found, add the  $i$ -th row of C to A, and mark  $P_i$ . Go to step 1
3. If no such process exists, terminate.

Remaining processes are deadlocked

# Example Deadlock Detection

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (2 \quad 1 \quad 0 \quad 0)$$

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

# Example Deadlock Detection

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (2 \quad 1 \quad 0 \quad 0)$$

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$


# Example Deadlock Detection

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (2 \quad 1 \quad 0 \quad 0)$$

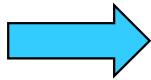
$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

# Example Deadlock Detection

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (2 \quad 2 \quad 2 \quad 0)$$

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

# Example Deadlock Detection

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (2 \quad 2 \quad 2 \quad 0)$$

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \xrightarrow{\text{blue arrow}} \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

# Example Deadlock Detection

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (4 \quad 2 \quad 2 \quad 1)$$

$$\begin{array}{c} \xrightarrow{\hspace{1cm}} \\ \xrightarrow{\hspace{1cm}} \end{array} C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \xrightarrow{\hspace{1cm}} \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

# Example Deadlock Detection

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (4 \quad 2 \quad 2 \quad 1)$$

$$\begin{array}{l} \xrightarrow{\hspace{1cm}} C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \\ \xrightarrow{\hspace{1cm}} \end{array}$$

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

# Example Deadlock Detection

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (4 \quad 2 \quad 2 \quad 1)$$

$$\begin{array}{l} \xrightarrow{\hspace{1cm}} C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \quad \xrightarrow{\hspace{1cm}} R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix} \\ \xrightarrow{\hspace{1cm}} \end{array}$$

# Example Deadlock Detection

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (4 \quad 2 \quad 3 \quad 1)$$

$$\begin{array}{l} \xrightarrow{\hspace{1cm}} \\ \xrightarrow{\hspace{1cm}} C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \\ \xrightarrow{\hspace{1cm}} \end{array}$$

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

# Example Deadlock Detection

- Algorithm terminates with no unmarked processes
  - We have no dead lock

## Example 2: Deadlock Detection

- Suppose,  $P_3$  needs a CD-ROM as well as 2 Tapes and a Plotter

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (2 \quad 1 \quad 0 \quad 0)$$

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{pmatrix}$$

# Recovery from Deadlock

- Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource
- Recovery through rollback
  - checkpoint a process periodically
  - use this saved state
  - restart the process if it is found deadlocked
    - No guarantee is won't deadlock again

# Recovery from Deadlock

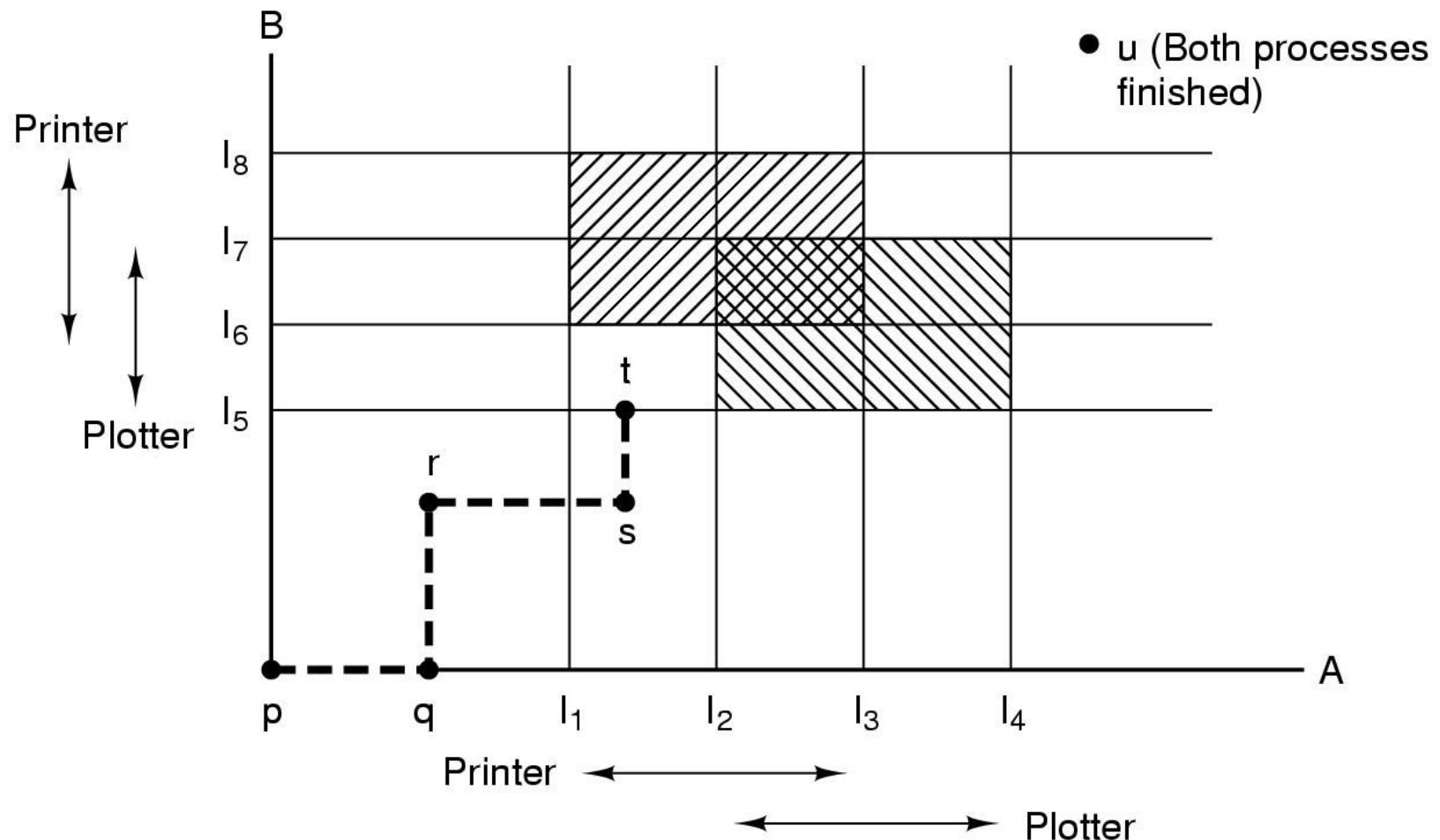
- Recovery through killing processes
  - crudest but simplest way to break a deadlock
  - kill one of the processes in the deadlock cycle
  - the other processes get its resources
  - choose process that can be rerun from the beginning

# Approach 4

## Deadlock Avoidance

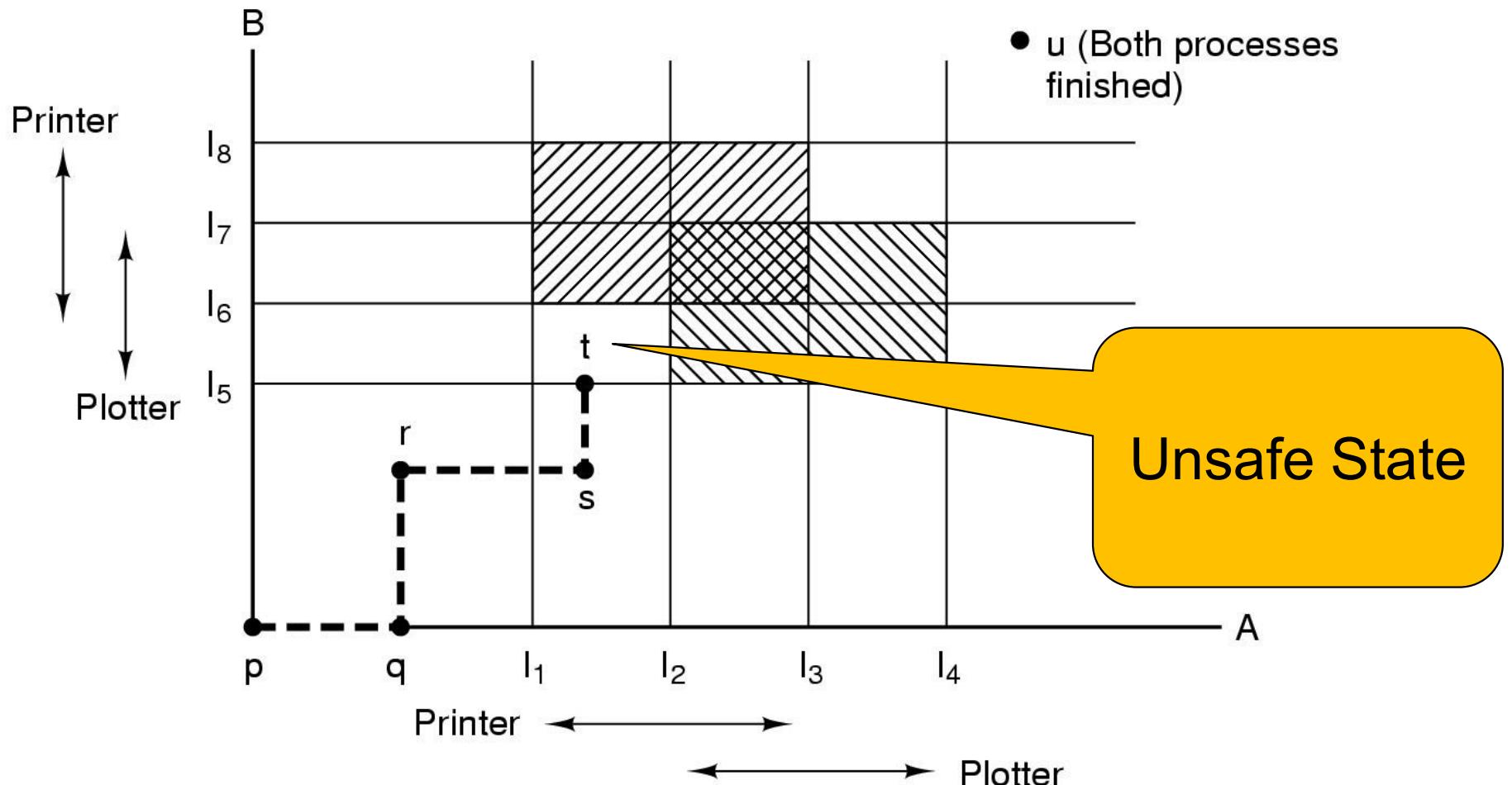
- Instead of detecting deadlock, can we simply avoid it?
  - YES, but only if enough information is available in advance.
  - Maximum number of each resource required

# Deadlock Avoidance Resource Trajectories



Two process resource trajectories

# Deadlock Avoidance Resource Trajectories



Two process resource trajectories

# Safe and Unsafe States

- A state is *safe* if
  - The system is not deadlocked
  - There exists a scheduling order that results in every process running to completion, *even if they all request their maximum resources immediately*

# Safe and Unsafe States

Note: We have 10 units of the resource

	Has	Max
A	3	9
B	2	4
C	2	7
Free: 3		

(a)

	Has	Max
A	3	9
B	4	4
C	2	7
Free: 1		

(b)

	Has	Max
A	3	9
B	0	-
C	2	7
Free: 5		

(c)

	Has	Max
A	3	9
B	0	-
C	7	7
Free: 0		

(d)

	Has	Max
A	3	9
B	0	-
C	0	-
Free: 7		

(e)

Demonstration that the state in (a) is safe

# Safe and Unsafe States

A requests one extra unit resulting in (b)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

Demonstration that the state in b is not safe

# Safe and Unsafe State

- Unsafe states are not necessarily deadlocked
  - With a lucky sequence, all processes may complete
  - However, we *cannot guarantee* that they will complete (not deadlock)
- Safe states guarantee we will eventually complete all processes
- Deadlock avoidance algorithm
  - Only grant requests that result in safe states

# Bankers Algorithm

- Modelled on a Banker with Customers
  - The banker has a limited amount of money to loan customers
    - Limited number of resources
  - Each customer can borrow money up to the customer's credit limit
    - Maximum number of resources required
- Basic Idea
  - Keep the bank in a *safe* state
    - So all customers are happy even if they all request to borrow up to their credit limit at the same time.
  - Customers wishing to borrow such that the bank would enter an unsafe state must wait until somebody else repays their loan such that the transaction becomes safe.

# The Banker's Algorithm for a Single Resource

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

- Three resource allocation states
  - safe
  - safe
  - unsafe

B requests one more, should we grant it?

# Banker's Algorithm for Multiple Resources

	Process	Tape drives	Plotters	Scanners	CD ROMS
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Resources assigned

	Process	Tape drives	Plotters	Scanners	CD ROMS
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Resources still needed

$$\begin{aligned}E &= (6342) \\P &= (5322) \\A &= (1020)\end{aligned}$$

- Example of banker's algorithm with multiple resources
- Problem is structured similar to deadlock detection with multiple resources.
- Example in tutorial

# Bankers Algorithm is not commonly used in practice

- It is difficult (sometimes impossible) to know in advance
  - the resources a process will require
  - the number of processes in a dynamic system

# Starvation

- A process never receives the resource it is waiting for, despite the resource (repeatedly) becoming free, the resource is always allocated to another waiting process.
  - Example: An algorithm to allocate a resource may be to give the resource to the shortest job first
  - Works great for multiple short jobs in a system
  - May cause a long job to wait indefinitely, even though not blocked.
- One solution:
  - First-come, first-serve policy

# Processes and Threads

## Implementation

# Learning Outcomes

- A basic understanding of the MIPS R3000 assembly and compiler generated code.
- An understanding of the typical implementation strategies of processes and threads
  - Including an appreciation of the trade-offs between the implementation approaches
    - Kernel-threads versus user-level threads
- A detailed understanding of “context switching”

# MIPS R3000

- Load/store architecture
  - No instructions that operate on memory except load and store
  - Simple load/stores to/from memory from/to registers
    - Store word: **sw r4, (r5)**
      - Store contents of r4 in memory using address contained in register r5
    - Load word: **lw r3, (r7)**
      - Load contents of memory into r3 using address contained in r7
      - Delay of one instruction after load before data available in destination register
        - Must always an instruction between a load from memory and the subsequent use of the register.
    - **lw, sw, lb, sb, lh, sh,....**

# MIPS R3000

- Arithmetic and logical operations are register to register operations
  - E.g., `add r3, r2, r1`
  - No arithmetic operations on memory
- Example
  - `add r3, r2, r1`  $\Rightarrow r3 = r2 + r1$
- Some other instructions
  - `add, sub, and, or, xor, sll, srl`
  - `move r2, r1`  $\Rightarrow r2 = r1$

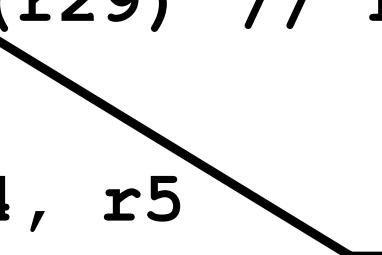
# MIPS R3000

- All instructions are encoded in 32-bit
- Some instructions have *immediate* operands
  - Immediate values are constants encoded in the instruction itself
  - Only 16-bit value
  - Examples
    - Add Immediate: **addi r2, r1, 2048**  
 $\Rightarrow r2 = r1 + 2048$
    - Load Immediate : **li r2, 1234**  
 $\Rightarrow r2 = 1234$

# Example code

Simple code example: **a = a + 1**

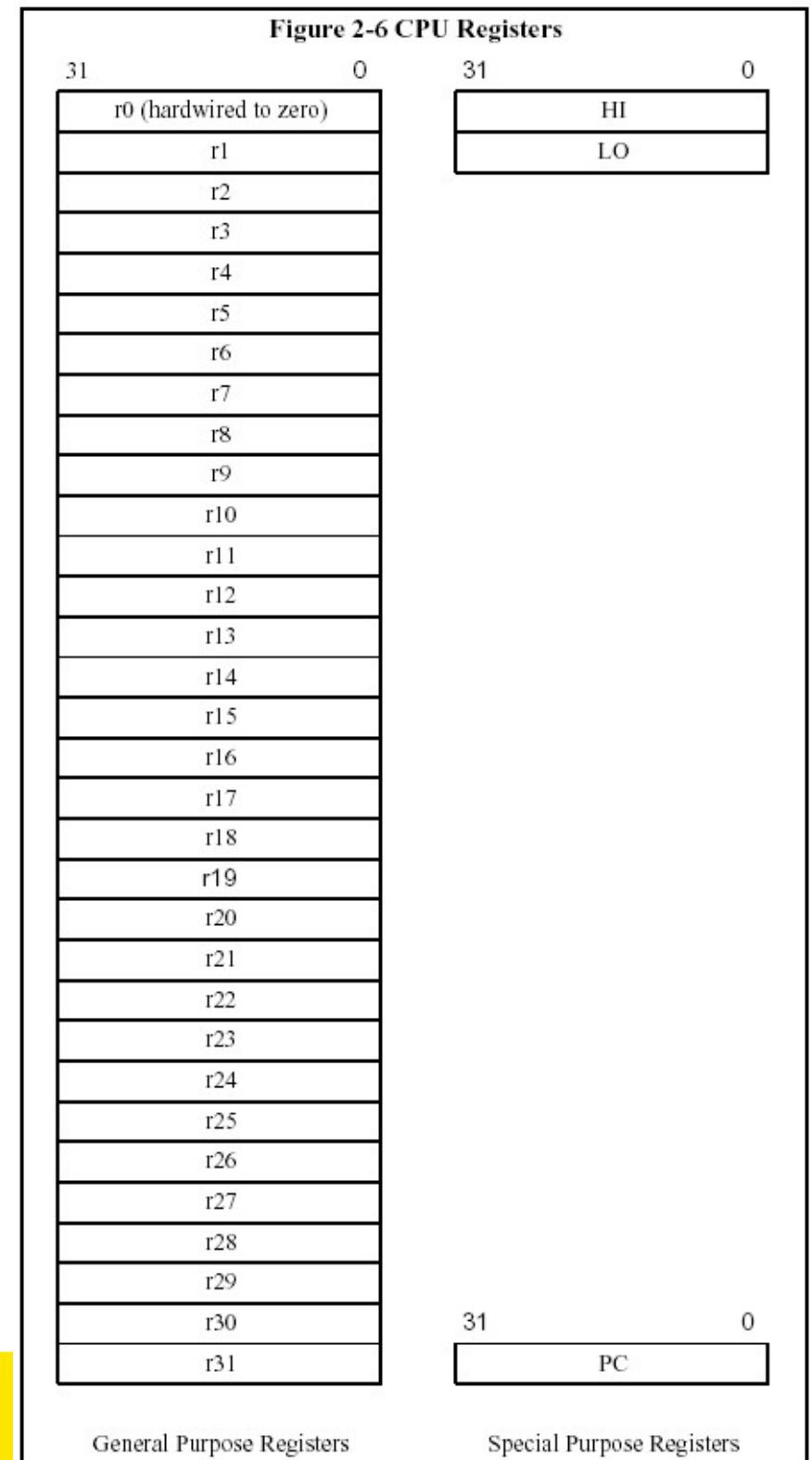
```
lw r4,32(r29) // r29 = stack pointer  
li r5, 1  
add r4, r4, r5  
sw r4,32(r29)
```



Offset(Address)

# MIPS Registers

- User-mode accessible registers
  - 32 general purpose registers
    - r0 hardwired to zero
    - r31 the *link* register for jump-and-link (JAL) instruction
  - HI/LO
    - 2 \* 32-bits for multiply and divide
  - PC
    - Not directly visible
    - Modified implicitly by jump and branch instructions



# Branching and Jumping

- Branching and jumping have a *branch delay slot*
  - The instruction following a branch or jump is always executed prior to destination of jump

```
    li      r2, 1  
    sw      r0, (r3)  
  
    j       1f  
  
    li      r2, 2  
  
    li      r2, 3  
  
1:    sw      r2, (r3)
```

# MIPS R3000

- RISC architecture – 5 stage pipeline
  - Instruction partially through pipeline prior to jmp having an effect

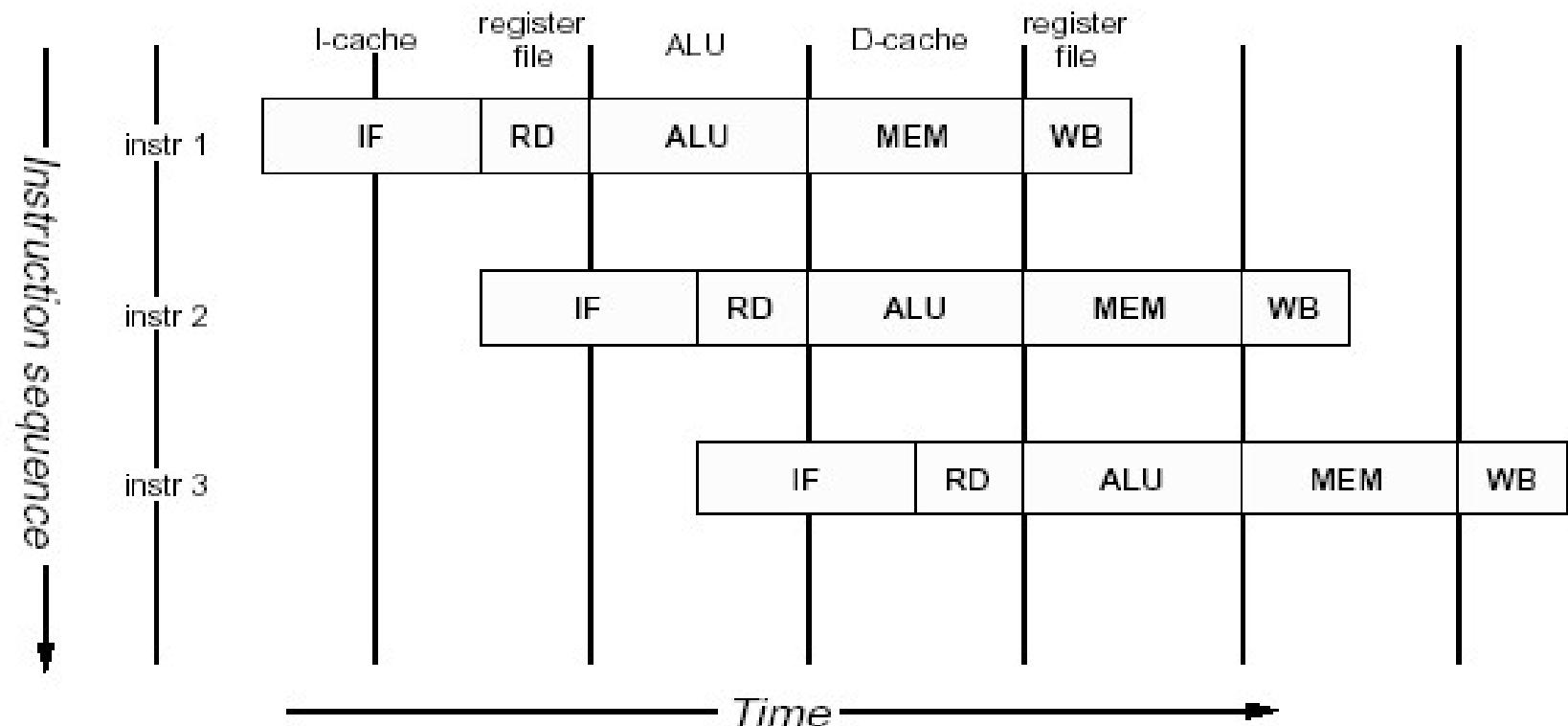
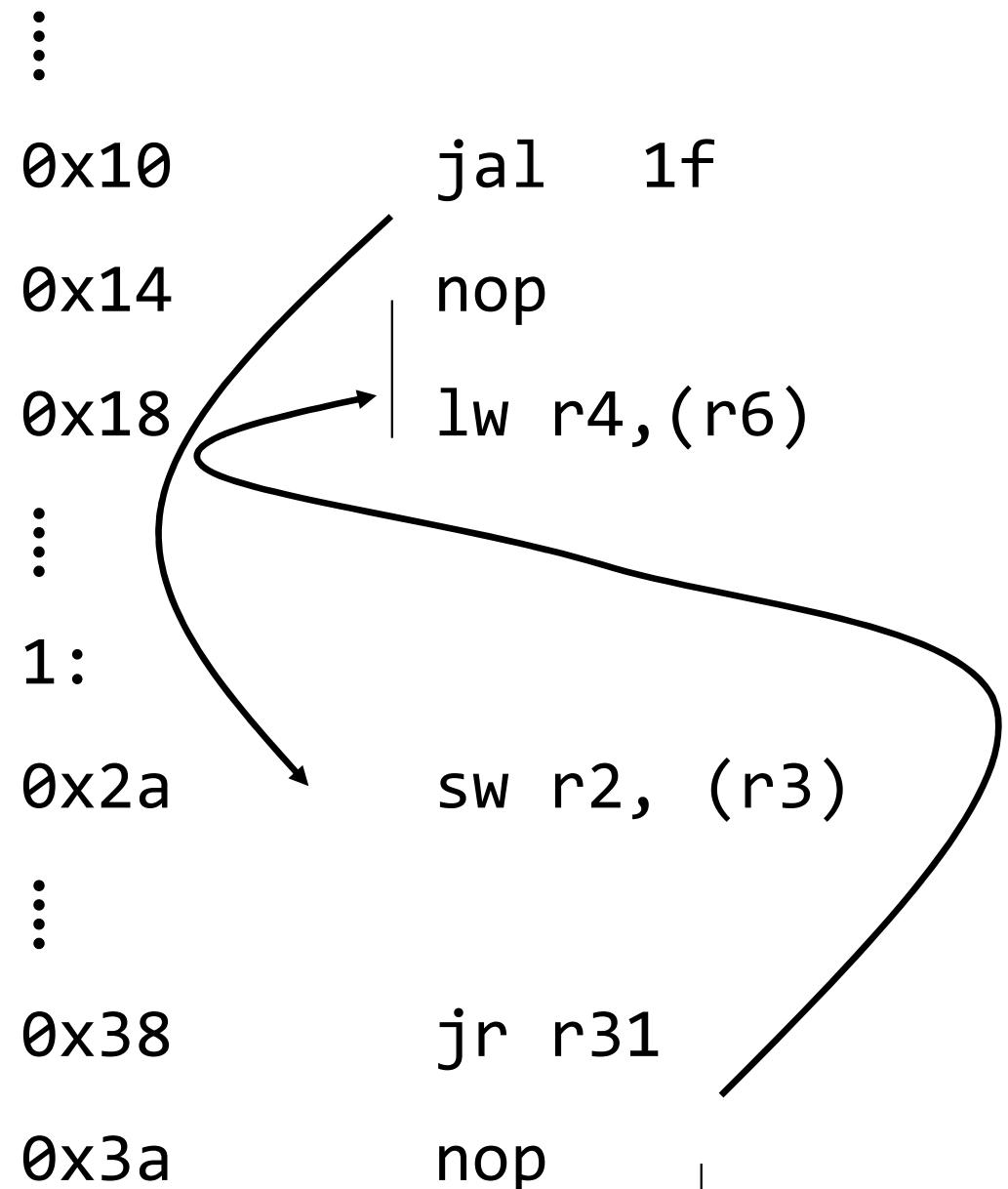


Figure 1.1. MIPS 5-stage pipeline

# Jump and Link Instruction

- JAL is used to implement function calls
  - $r31 = PC+8$
- Return Address register (RA) is used to return from function call



# Compiler Register Conventions

- Given 32 registers, which registers are used for
  - Local variables?
  - Argument passing?
  - Function call results?
  - Stack Pointer?

# Compiler Register Conventions

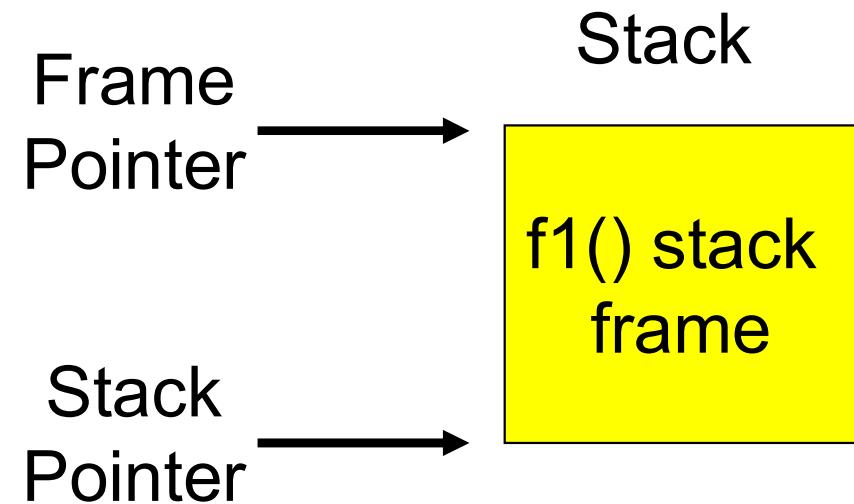
Reg No	Name	Used for
0	zero	Always returns 0
1	at	(assembler temporary) Reserved for use by assembler
2-3	v0-v1	Value (except FP) returned by subroutine
4-7	a0-a3	(arguments) First four parameters for a subroutine
8-15	t0-t7	(temporaries) subroutines may use without saving
24-25	t8-t9	
16-23	s0-s7	Subroutine “register variables”; a subroutine which will write one of these must save the old value and restore it before it exits, so the <i>calling</i> routine sees their values preserved.
26-27	k0-k1	Reserved for use by interrupt/trap handler - may change under your feet
28	gp	global pointer - some runtime systems maintain this to give easy access to (some) “static” or “extern” variables.
29	sp	stack pointer
30	s8/fp	9th register variable. Subroutines which need one can use this as a “frame pointer”.
31	ra	Return address for subroutine

# Simple factorial

int fact(int n)	0:	1880000b	blez	a0,30 <fact+0x30>
{	4:	24840001	addiu	a0,a0,1
int r = 1;	8:	24030001	li	v1,1
int i;	c:	24020001	li	v0,1
	10:	00430018	mult	v0,v1
for (i = 1; i < n+1; i++) {	14:	24630001	addiu	v1,v1,1
r = r * i;	18:	00001012	mflo	v0
}	1c:	00000000	nop	
return r;	20:	1464ffffc	bne	v1,a0,14 <fact+0x14>
}	24:	00430018	mult	v0,v1
	28:	03e00008	jr	ra
	2c:	00000000	nop	
	30:	03e00008	jr	ra
	34:	24020001	li	v0,1

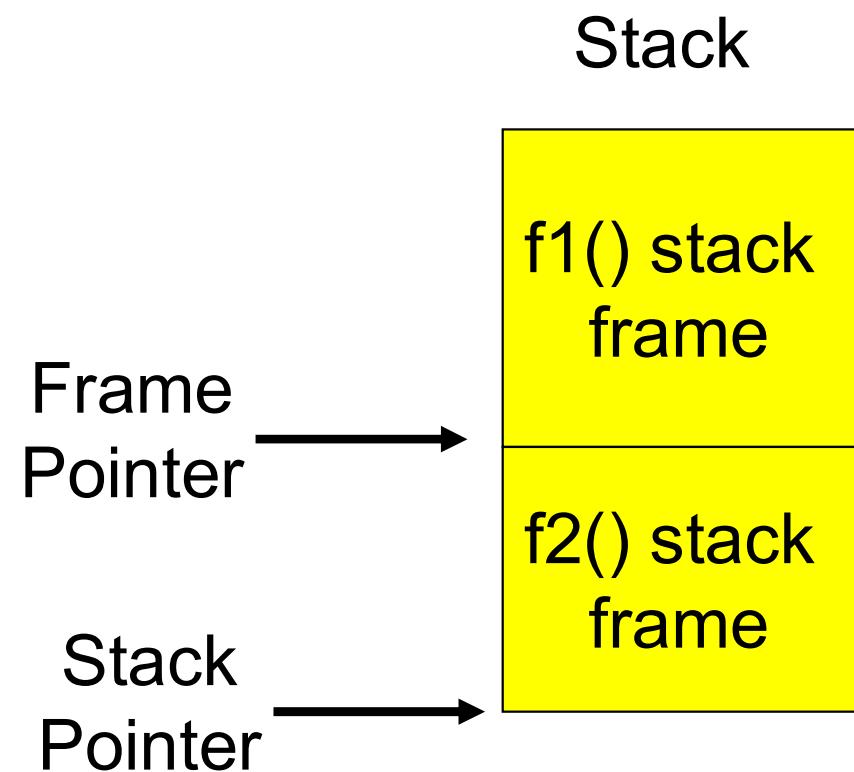
# Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame
- Example: assume f1() calls f2(), which calls f3().



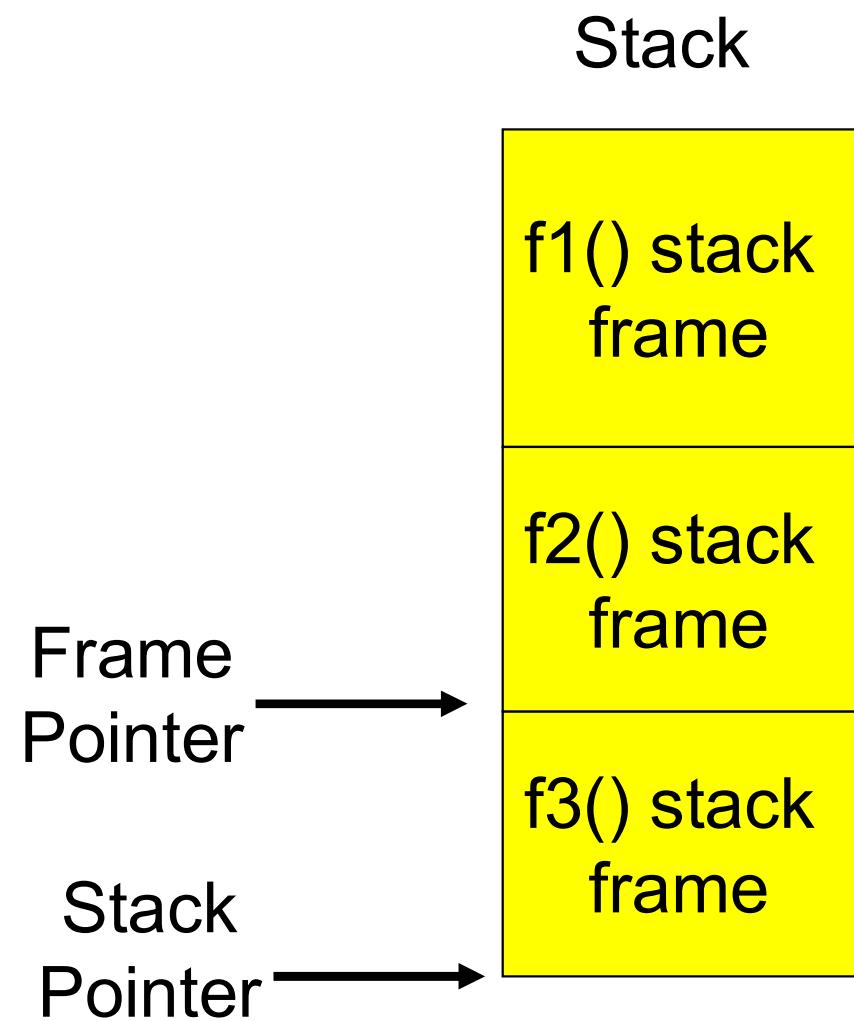
# Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame
- Example: assume f1() calls f2(), which calls f3().



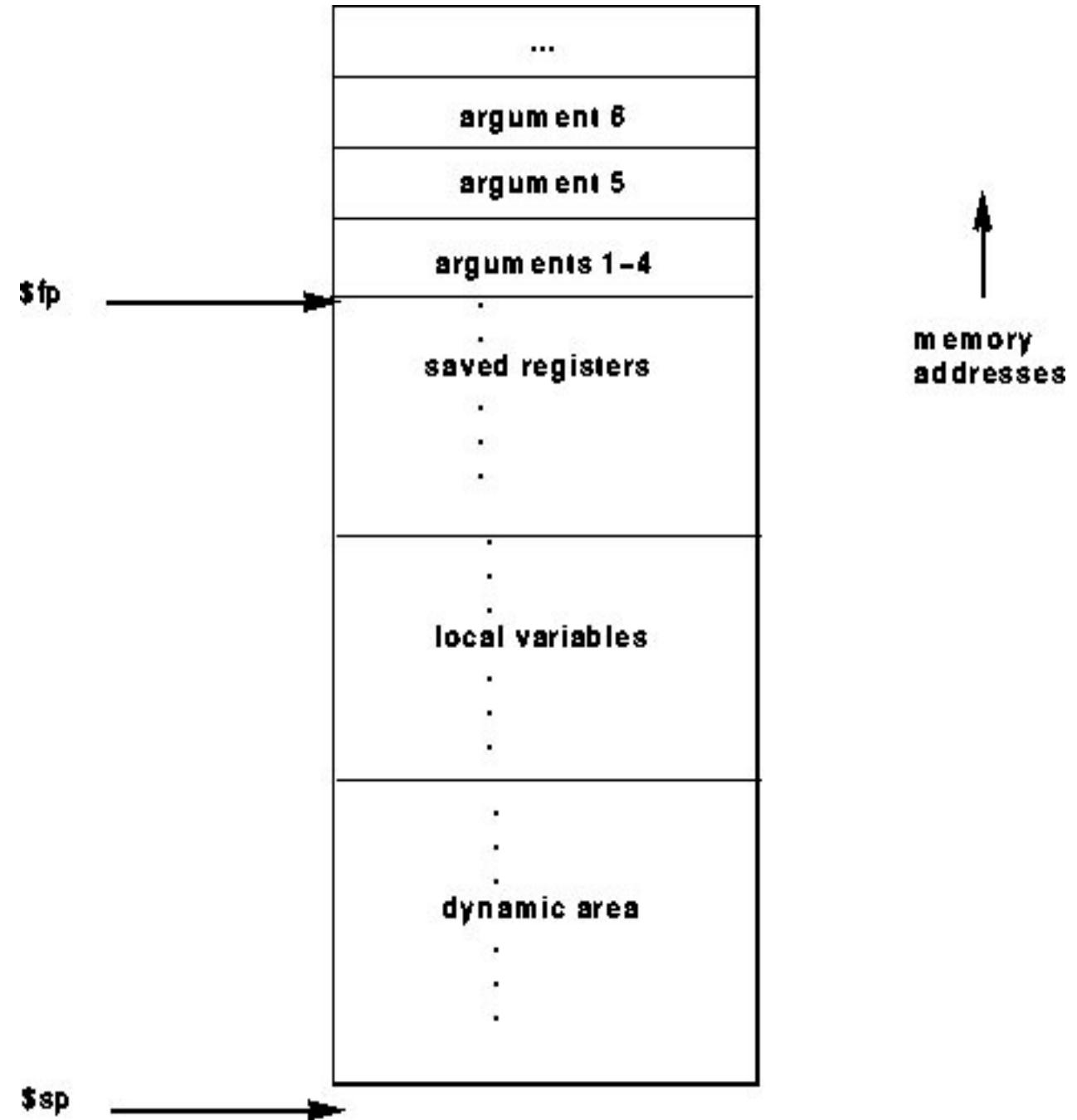
# Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame
- Example: assume f1() calls f2(), which calls f3().



# Stack Frame

- MIPS calling convention for gcc
  - Args 1-4 have space reserved for them



# Example Code

```
main ()  
{  
    int i;  
  
    i =  
    sixargs(1,2,3,4,5,6);  
}  
  
int sixargs(int a, int  
            b, int c, int d, int e,  
            int f)  
{  
    return a + b + c + d  
        + e + f;  
}
```

0040011c <main>:

40011c:	27bdffd8	addiu	sp, sp, -40
400120:	afb0024	sw	ra, 36(sp)
400124:	afbe0020	sw	s8, 32(sp)
400128:	03a0f021	move	s8, sp
40012c:	24020005	li	v0, 5
400130:	afa20010	sw	v0, 16(sp)
400134:	24020006	li	v0, 6
400138:	afa20014	sw	v0, 20(sp)
40013c:	24040001	li	a0, 1
400140:	24050002	li	a1, 2
400144:	24060003	li	a2, 3
400148:	0c10002c	jal	4000b0 <sixargs>
40014c:	24070004	li	a3, 4
400150:	afc20018	sw	v0, 24(s8)
400154:	03c0e821	move	sp, s8
400158:	8fb0024	lw	ra, 36(sp)
40015c:	8fbe0020	lw	s8, 32(sp)
400160:	03e00008	jr	ra
400164:	27bd0028	addiu	sp, sp, 40

...

004000b0 <sixargs>:

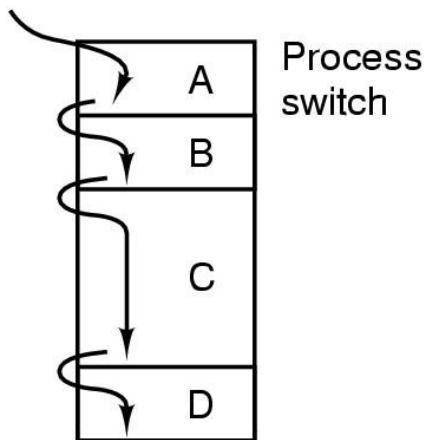
4000b0:	27bdfff8	addiu	sp,sp,-8
4000b4:	afbe0000	sw	s8,0(sp)
4000b8:	03a0f021	move	s8,sp
4000bc:	afc40008	sw	a0,8(s8)
4000c0:	afc5000c	sw	a1,12(s8)
4000c4:	afc60010	sw	a2,16(s8)
4000c8:	afc70014	sw	a3,20(s8)
4000cc:	8fc30008	lw	v1,8(s8)
4000d0:	8fc2000c	lw	v0,12(s8)
4000d4:	00000000	nop	
4000d8:	00621021	addu	v0,v1,v0
4000dc:	8fc30010	lw	v1,16(s8)
4000e0:	00000000	nop	
4000e4:	00431021	addu	v0,v0,v1
4000e8:	8fc30014	lw	v1,20(s8)
4000ec:	00000000	nop	
4000f0:	00431021	addu	v0,v0,v1
4000f4:	8fc30018	lw	v1,24(s8)
4000f8:	00000000	nop	

```
4000fc: 00431021 addu v0,v0,v1
400100: 8fc3001c lw v1,28($8)
400104: 00000000 nop
400108: 00431021 addu v0,v0,v1
40010c: 03c0e821 move sp,s8
400110: 8fbe0000 lw s8,0(sp)
400114: 03e00008 jr ra
400118: 27bd0008 addiusp,sp,8
```

# The Process Model

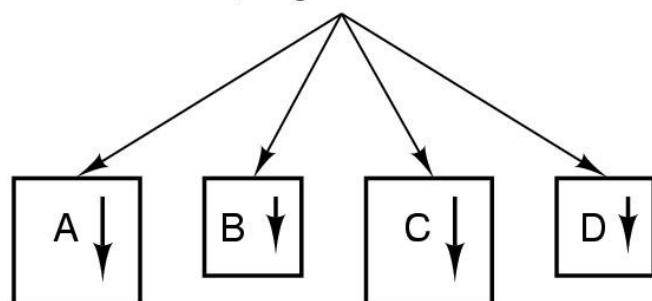
- Multiprogramming of four programs

One program counter

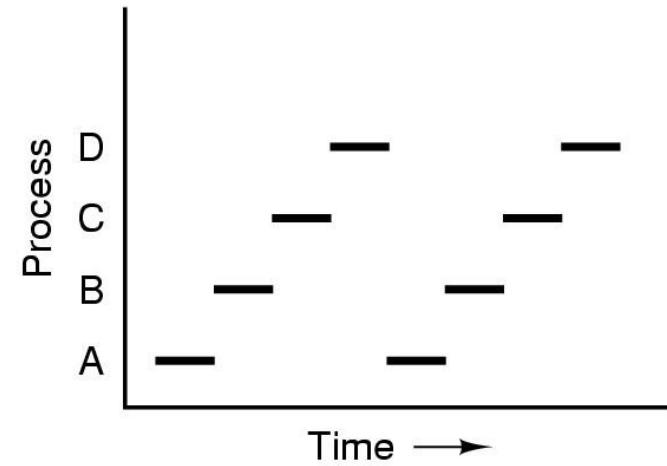


(a)

Four program counters



(b)

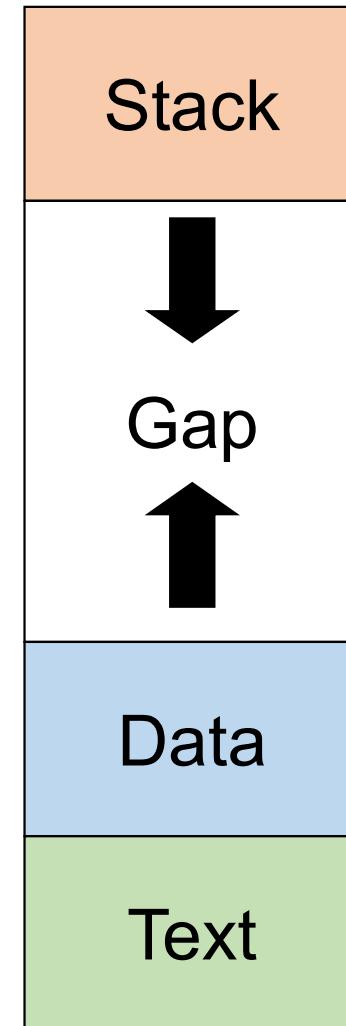


(c)

# Process

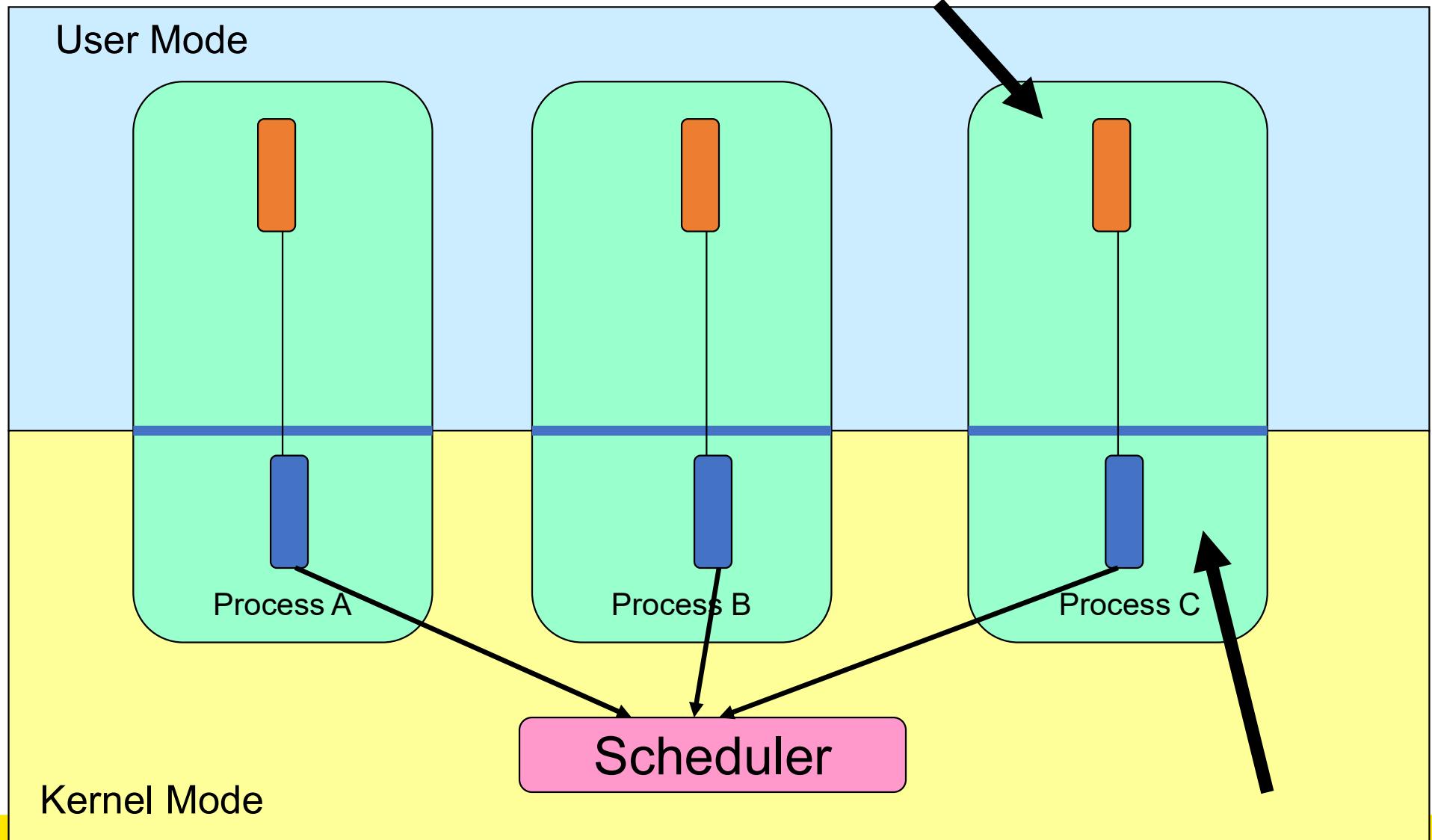
- Minimally consist of three segments
  - Text
    - contains the code (instructions)
  - Data
    - Global variables
  - Stack
    - Activation records of procedure/function/method
    - Local variables
- Note:
  - data can dynamically grow up
    - E.g., malloc()-ing
  - The stack can dynamically grow down
    - E.g., increasing function call depth or recursion

# Process Memory Layout



# Processes

Process's user-level stack and execution state



Process's in-kernel stack and execution state



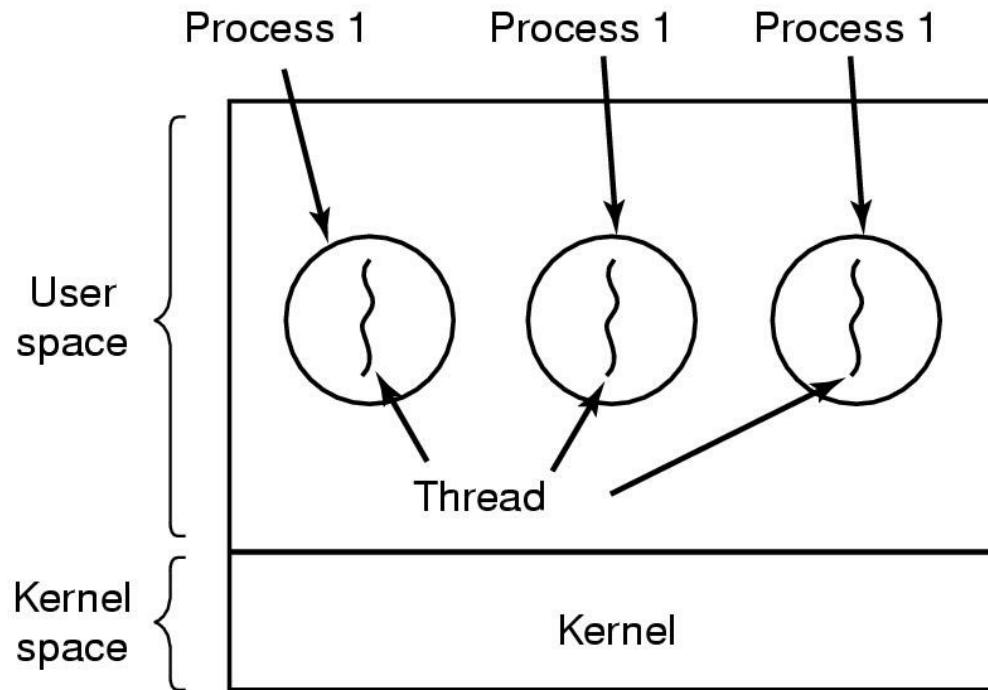
UNSW  
SYDNEY

# Processes

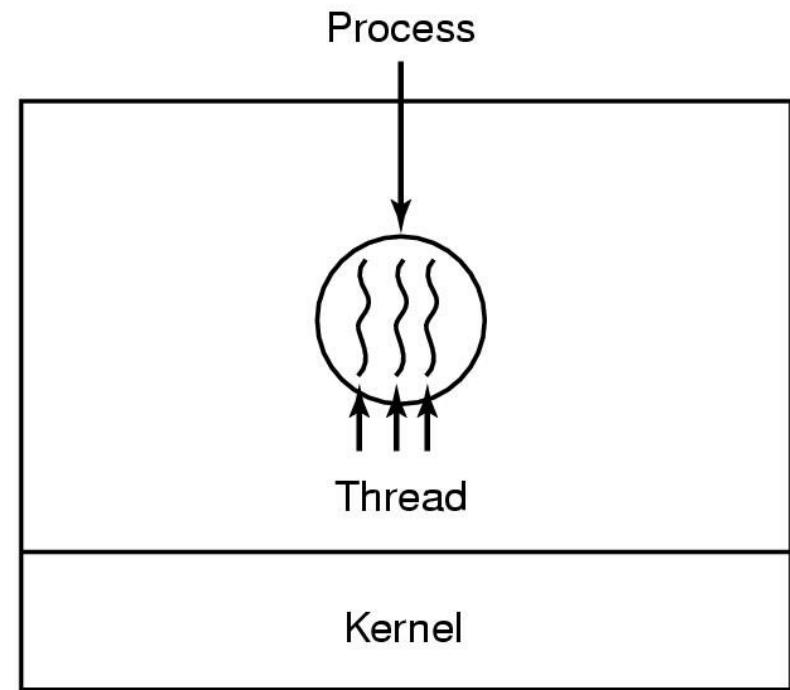
- User-mode
  - Processes (programs) scheduled by the kernel
  - Isolated from each other
  - No concurrency issues between each other
- System-calls transition into and return from the kernel
- Kernel-mode
  - Nearly all activities still associated with a process
  - Kernel memory shared between all processes
  - Concurrency issues exist between processes concurrently executing in a system call

# Threads

## The Thread Model



(a)



(b)

- (a) Three processes each with one thread
- (b) One process with three threads

# The Thread Model

## **Per process items**

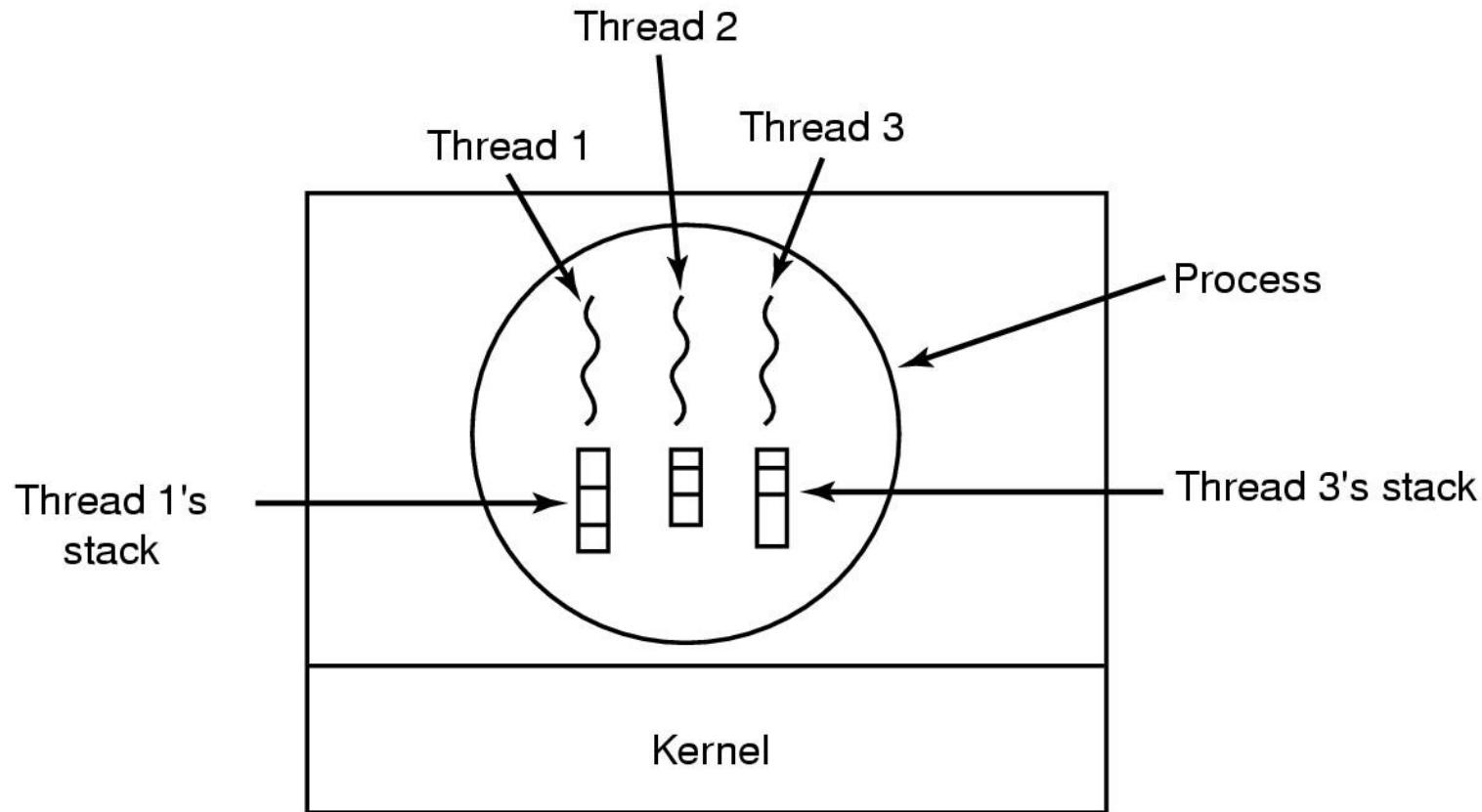
Address space  
Global variables  
Open files  
Child processes  
Pending alarms  
Signals and signal handlers  
Accounting information

## **Per thread items**

Program counter  
Registers  
Stack  
State

- Items shared by all threads in a process
- Items that exist per thread

# The Thread Model



Each thread has its own stack

# A Subset of POSIX threads API

```
int    pthread_create(pthread_t *, const pthread_attr_t *,
                     void *(*)(void *), void *);

void  pthread_exit(void *);

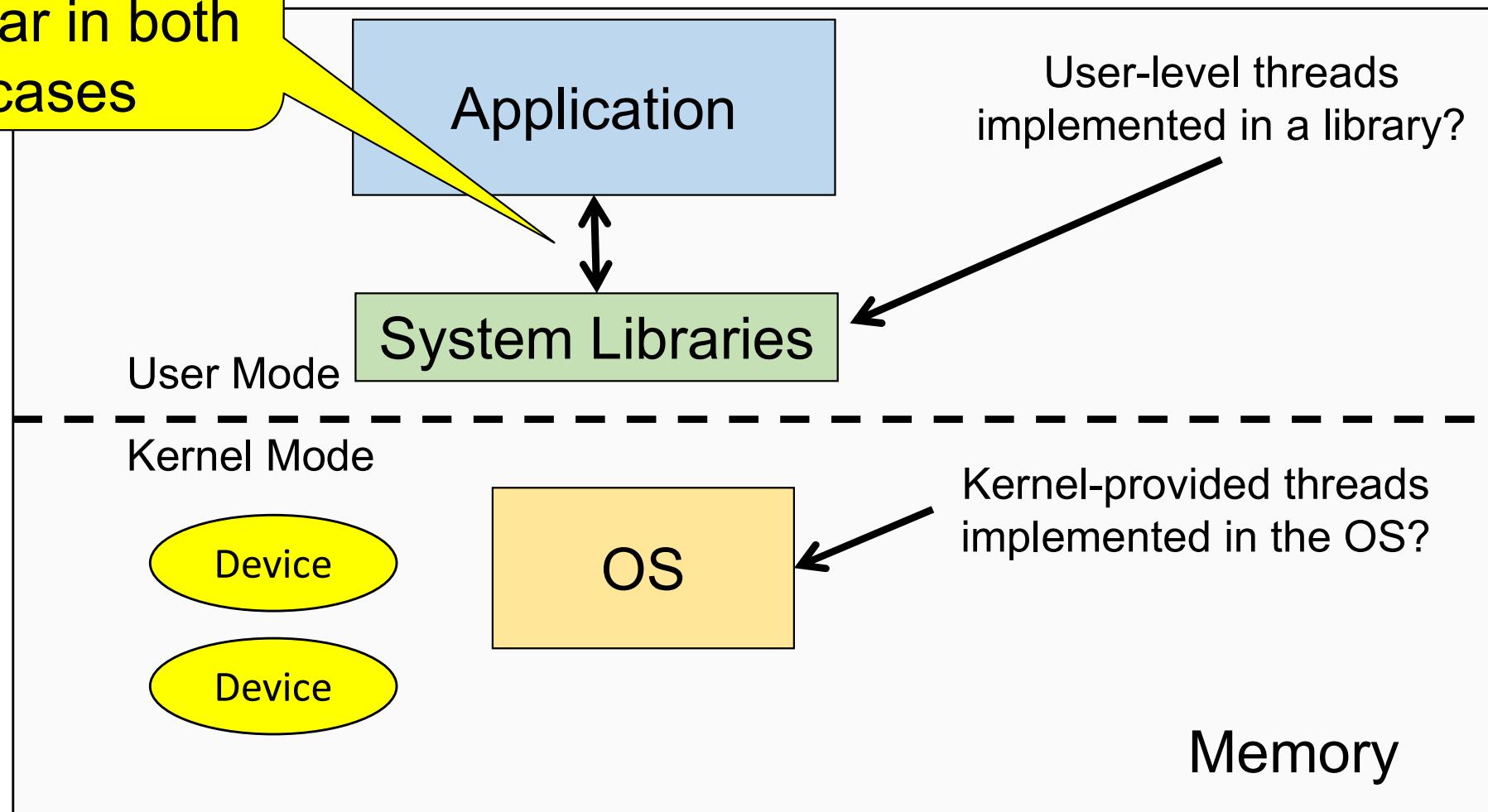
int    pthread_mutex_init(pthread_mutex_t *, const pthread_mutexattr_t *);
int    pthread_mutex_destroy(pthread_mutex_t *);
int    pthread_mutex_lock(pthread_mutex_t *);
int    pthread_mutex_unlock(pthread_mutex_t *);

int    pthread_rwlock_init(pthread_rwlock_t *,
                          const pthread_rwlockattr_t *);
int    pthread_rwlock_destroy(pthread_rwlock_t *);
int    pthread_rwlock_rdlock(pthread_rwlock_t *);
int    pthread_rwlock_wrlock(pthread_rwlock_t *);
int    pthread_rwlock_unlock(pthread_rwlock_t );
```

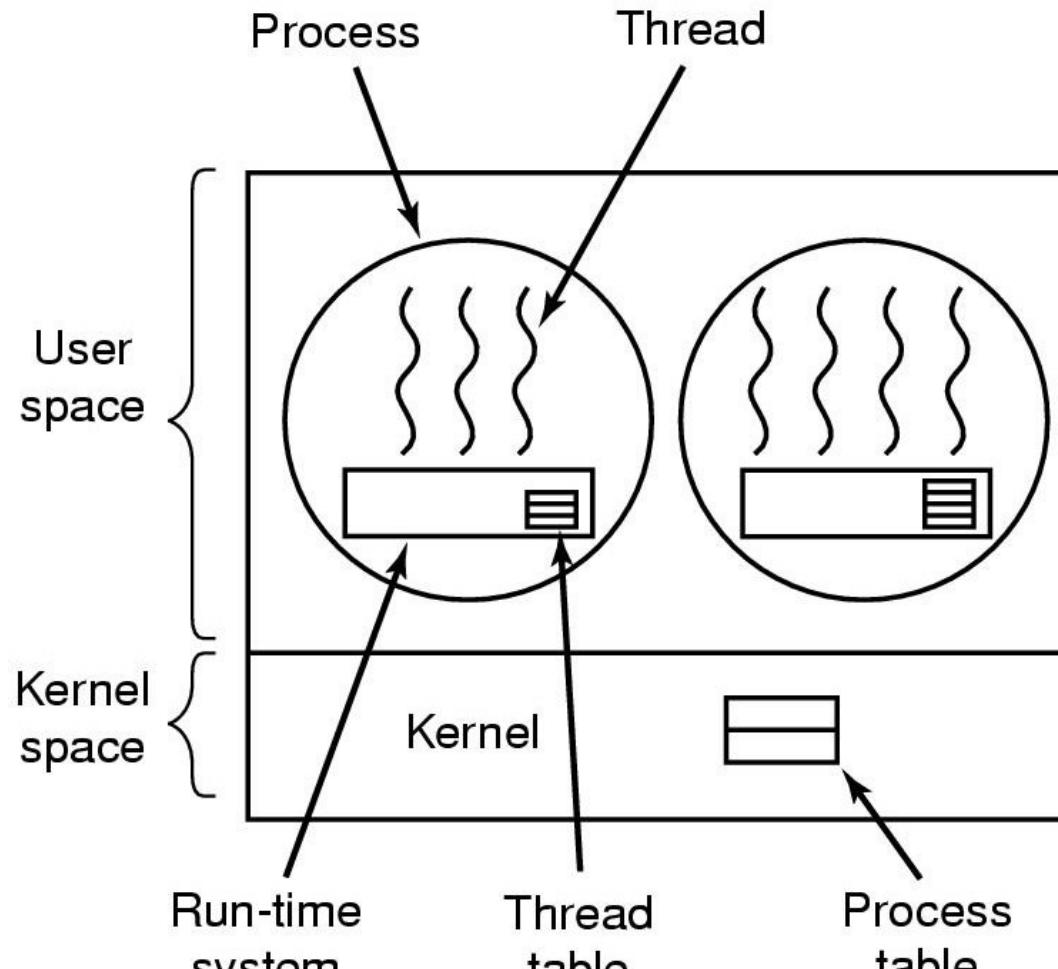
# Where to Implement Application

Threads?

Note: Thread API  
similar in both  
cases

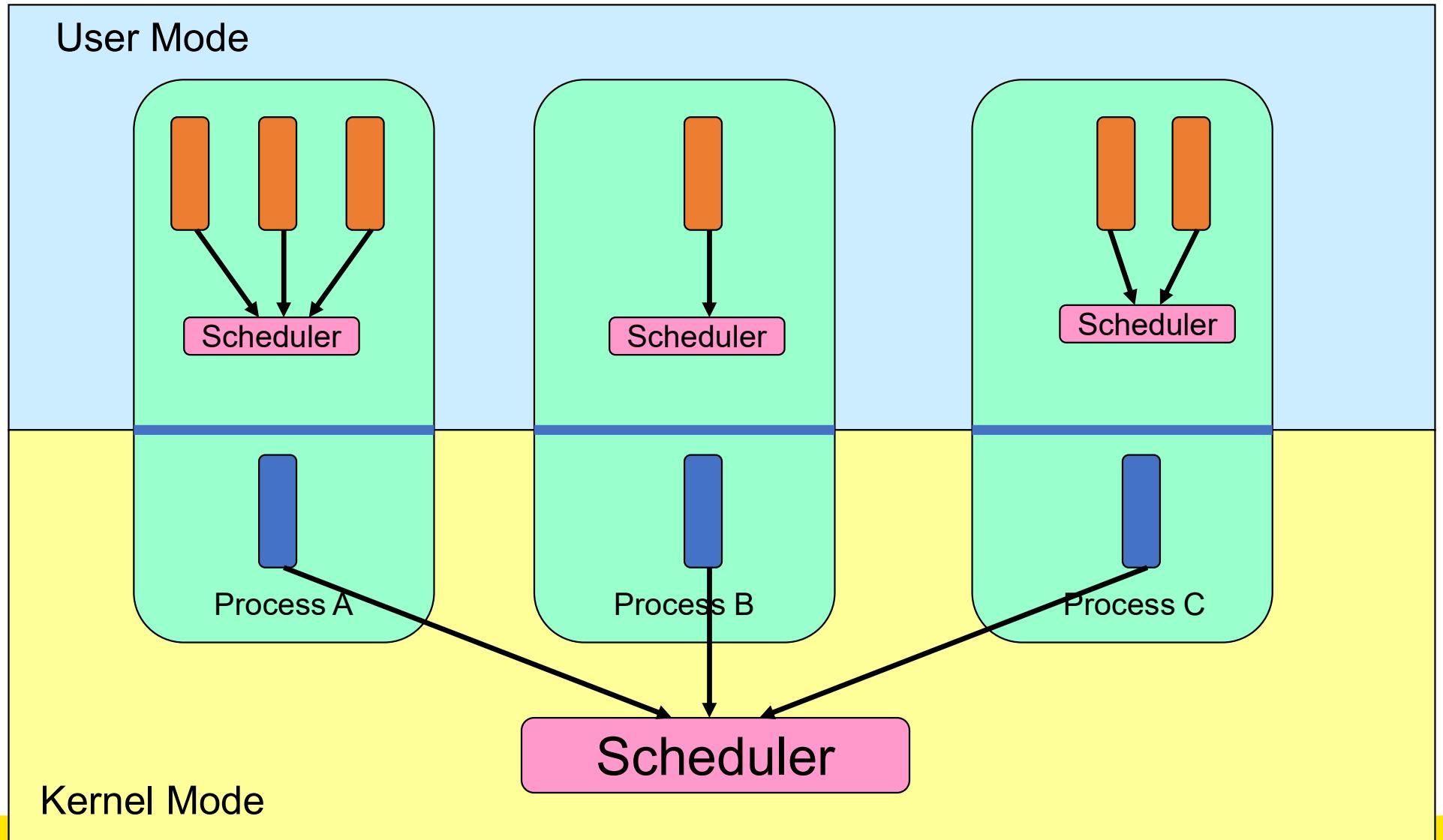


# Implementing Threads in User Space



A user-level threads library

# User-level Threads



# User-level Threads

- Implementation at user-level
  - User-level Thread Control Block (TCB), ready queue, blocked queue, and dispatcher
  - Kernel has no knowledge of the threads (it only sees a single process)
  - If a thread blocks waiting for a resource held by another thread inside the same process, its state is saved and the dispatcher switches to another ready thread
  - Thread management (create, exit, yield, wait) are implemented in a runtime support library

# User-Level Threads

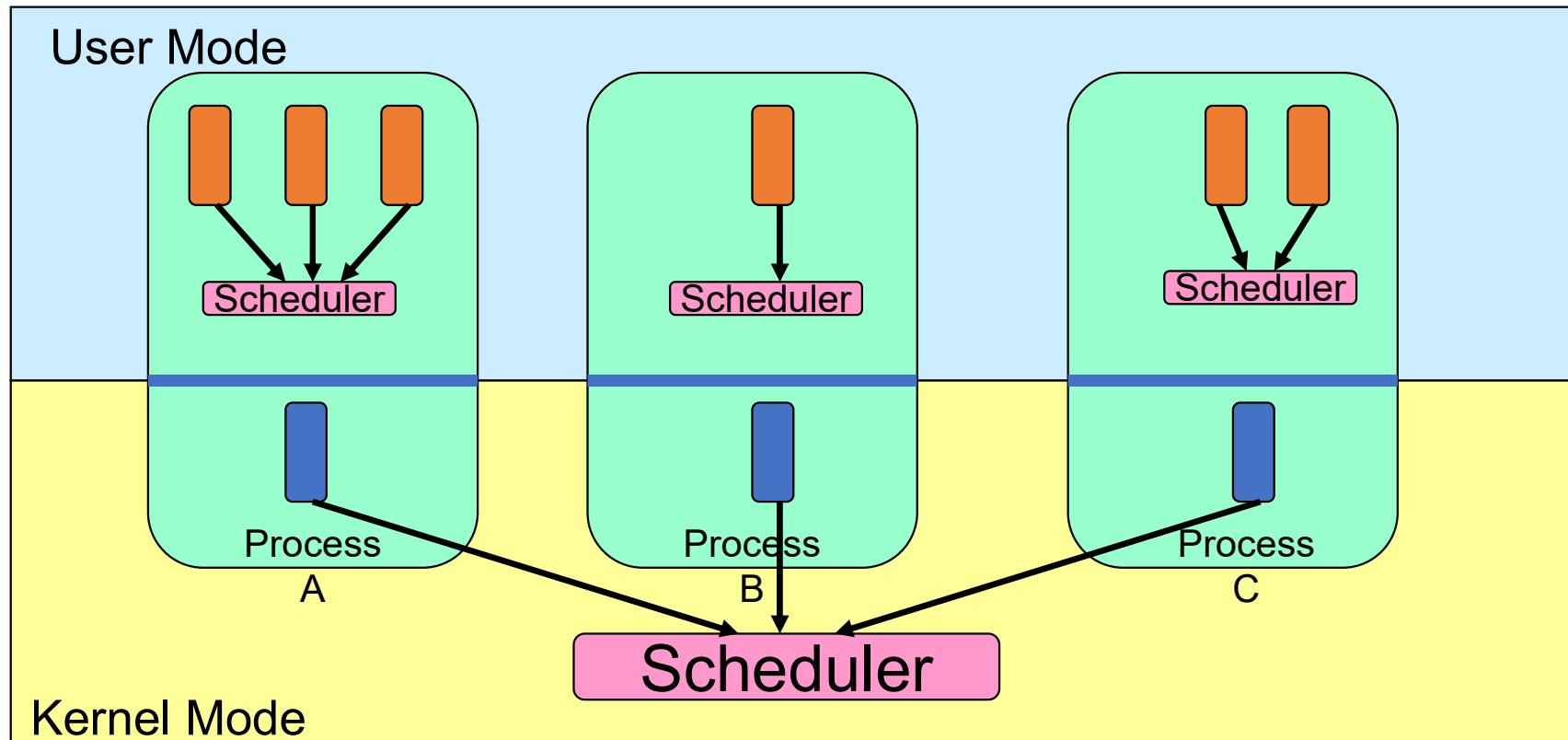
- Pros
  - Thread management and switching at user level is much faster than doing it in kernel level
    - No need to trap (take syscall exception) into kernel and back to switch
  - Dispatcher algorithm can be tuned to the application
    - E.g. use priorities
  - Can be implemented on any OS (thread or non-thread aware)
  - Can easily support massive numbers of threads on a per-application basis
    - Use normal application virtual memory
    - Kernel memory more constrained. Difficult to efficiently support wildly differing numbers of threads for different applications.

# User-level Threads

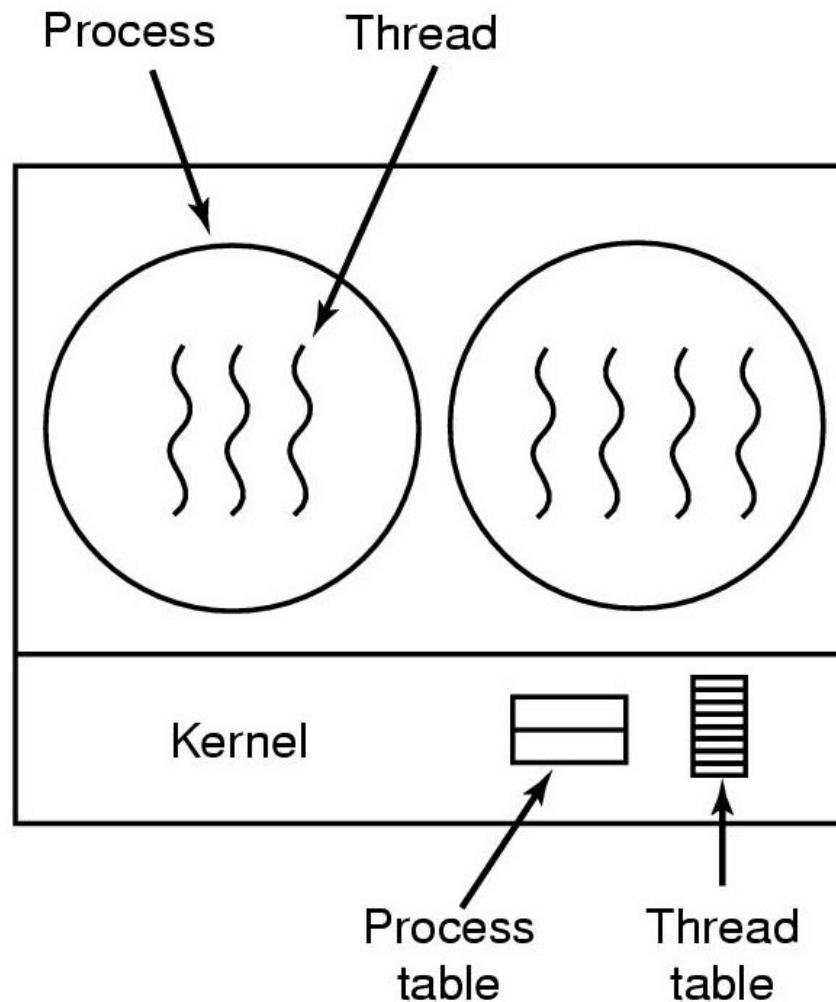
- Cons
  - Threads have to yield() manually (no timer interrupt delivery to user-level)
    - Co-operative multithreading
      - A single poorly design/implemented thread can monopolise the available CPU time
    - There are work-arounds (e.g. a timer signal per second to enable pre-emptive multithreading), they are course grain and a kludge.
  - Does not take advantage of multiple CPUs (in reality, we still have a single threaded process as far as the kernel is concerned)

# User-Level Threads

- Cons
  - If a thread makes a blocking system call (or takes a page fault), the process (and all the internal threads) blocks
    - Can't overlap I/O with computation

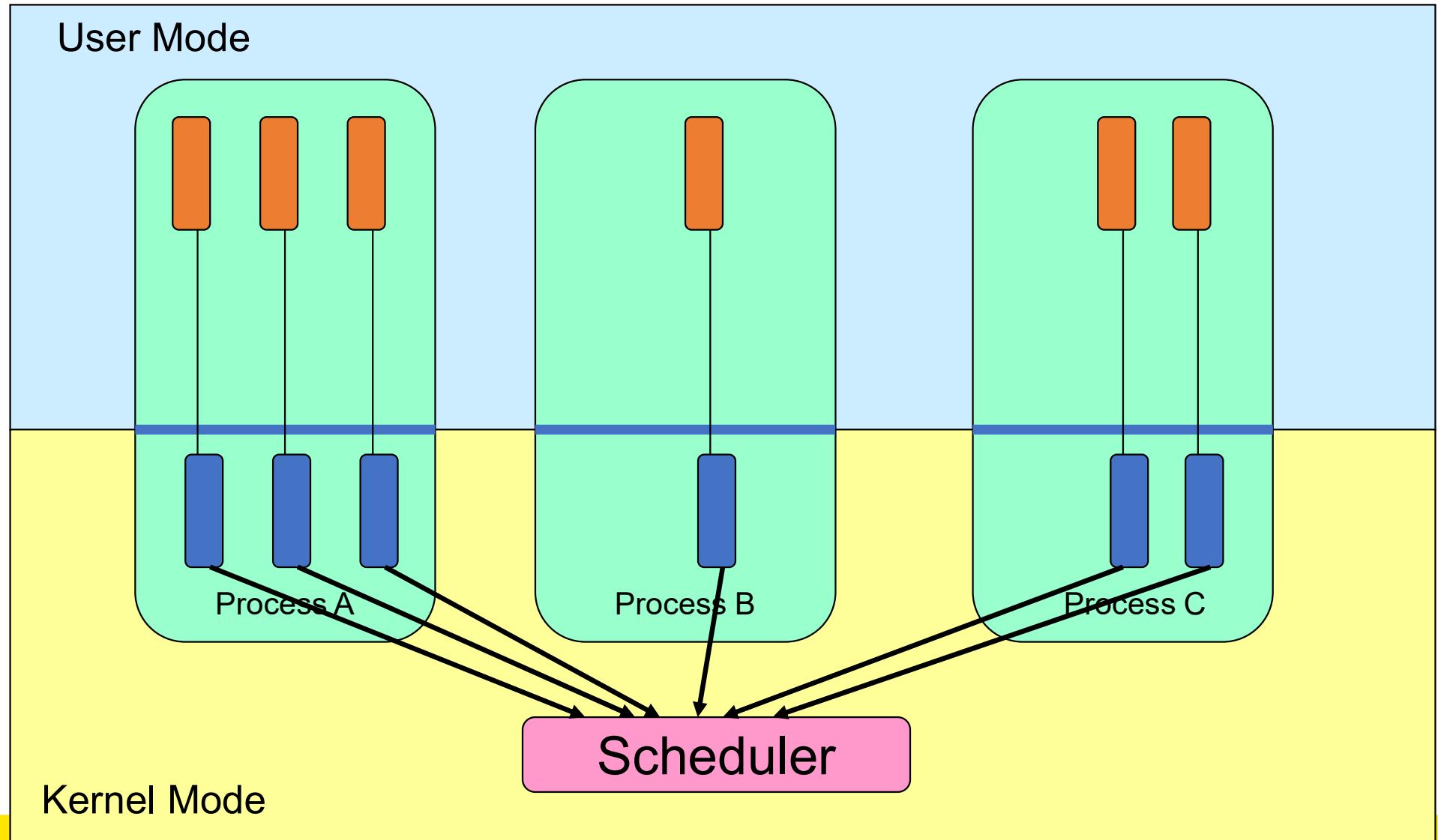


# Implementing Threads in the Kernel



A threads package managed by the kernel

# Kernel-provided Threads



# Kernel-provided Threads

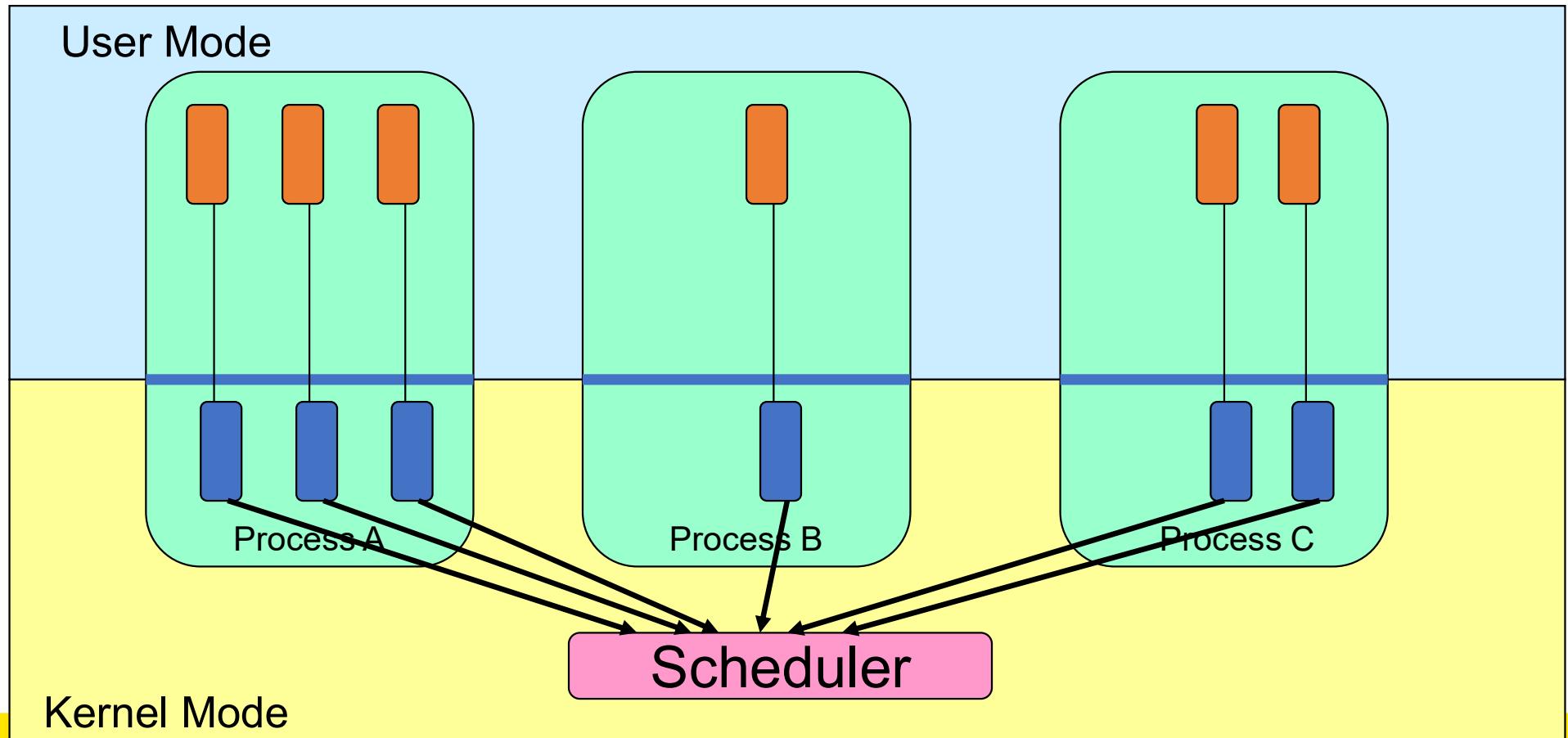
- Also called kernel-level threads
  - Even though they provide threads to applications
- Threads are implemented by the kernel
  - TCBs are stored in the kernel
    - A subset of information in a traditional PCB
      - The subset related to execution context
    - TCBs have a PCB associated with them
      - Resources associated with the group of threads (the process)
  - Thread management calls are implemented as system calls
    - E.g. create, wait, exit

# Kernel-provided Threads

- Cons
  - Thread creation and destruction, and blocking and unblocking threads requires kernel entry and exit.
  - More expensive than user-level equivalent

# Kernel-provided Threads

- Pros
  - Preemptive multithreading
  - Parallelism
    - Can overlap blocking I/O with computation
    - Can take advantage of a multiprocessor



# Multiprogramming Implementation

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Skeleton of what lowest level of OS does when an interrupt occurs – a context switch

# *Context Switch Terminology*

- A context switch can refer to
  - A switch between threads
    - Involving saving and restoring of state associated with a thread
  - A switch between processes
    - Involving the above, plus extra state associated with a process.
      - E.g. memory maps

# Context Switch Occurrence

- A switch between process/threads can happen any time the OS is invoked
  - On a system call
    - Mandatory if system call blocks or on exit();
  - On an exception
    - Mandatory if offender is killed
  - On an interrupt
    - Triggering a dispatch is the main purpose of the *timer interrupt*

A thread switch can happen between any two instructions

Note instructions do not equal program statements

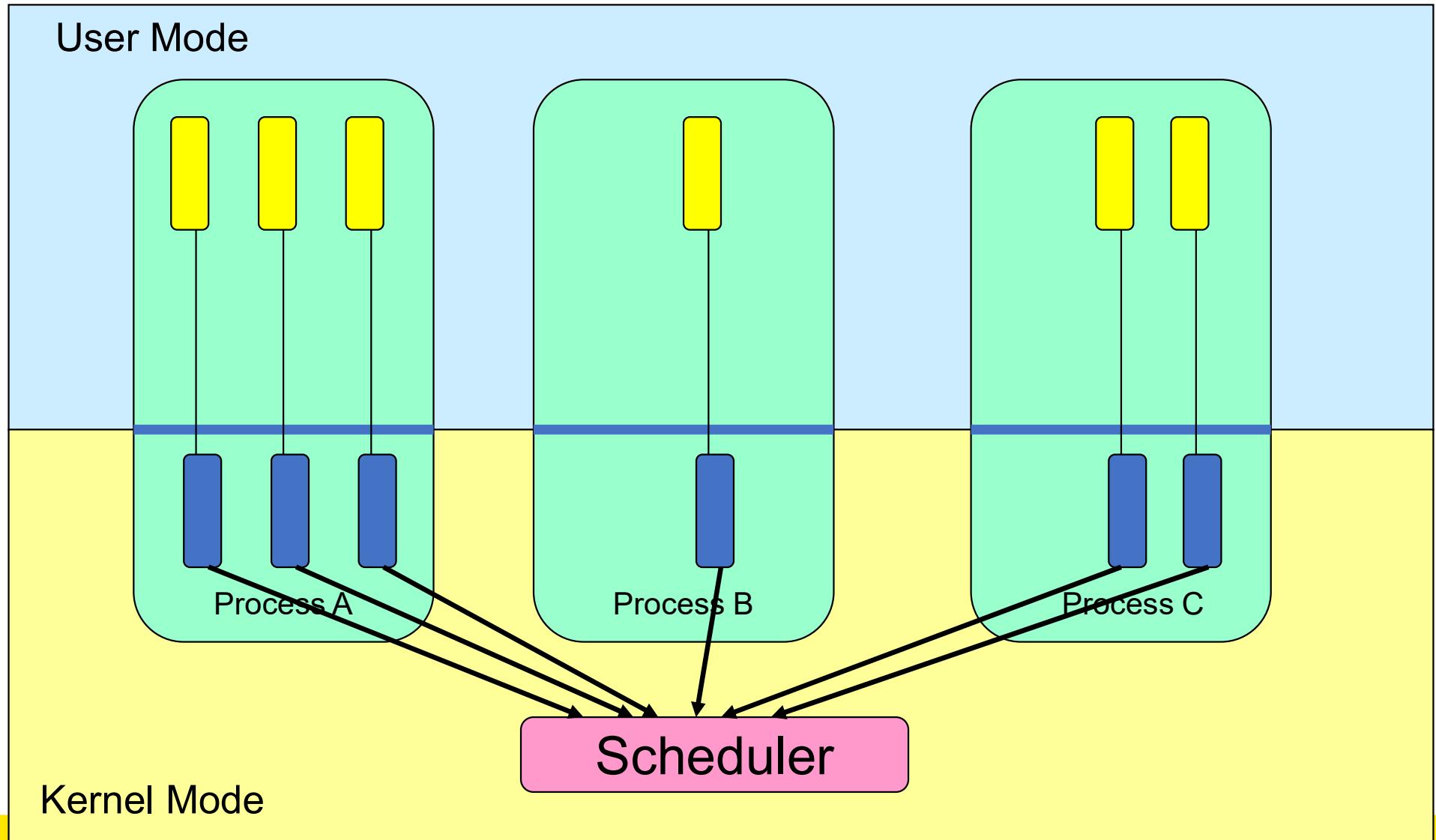
# Context Switch

- Context switch must be *transparent* for processes/threads
    - When dispatched again, process/thread should not notice that something else was running in the meantime (except for elapsed time)
- ⇒ OS must save all state that affects the thread
- This state is called the *process/thread context*
  - Switching between process/threads consequently results in a *context switch*.

# Simplified Explicit Thread Switch

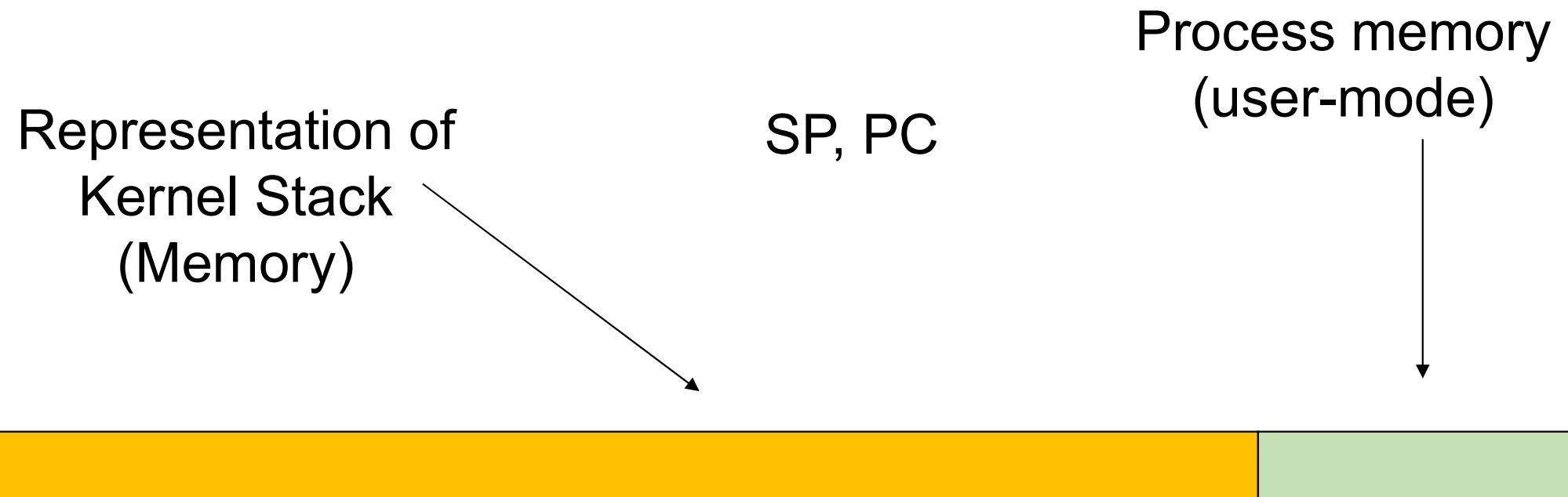
# Assume Kernel-Level Threads

Lets focus on user->kernel – switch – kernel -> user



# Example Context Switch

- Running in user mode, SP points to user-level stack (not shown on slide)



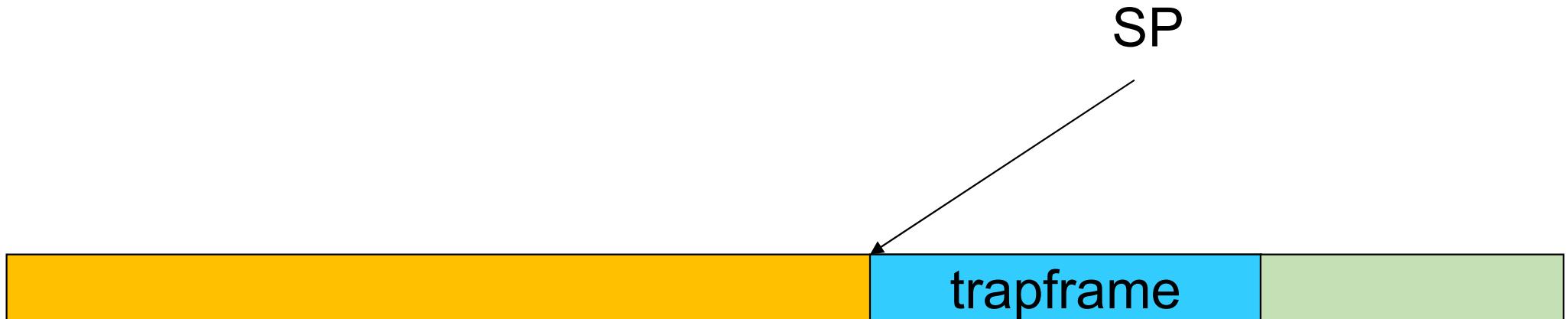
# Example Context Switch

- Take an exception, syscall, or interrupt, and we switch to the kernel stack



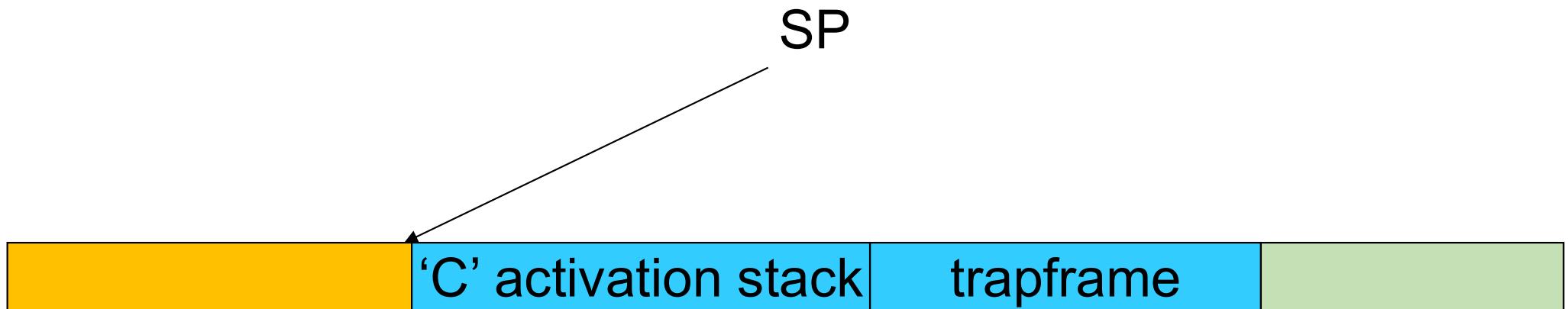
# Example Context Switch

- We push a *trapframe* on the stack
  - Also called *exception frame*, *user-level context*...
  - Includes the user-level PC and SP



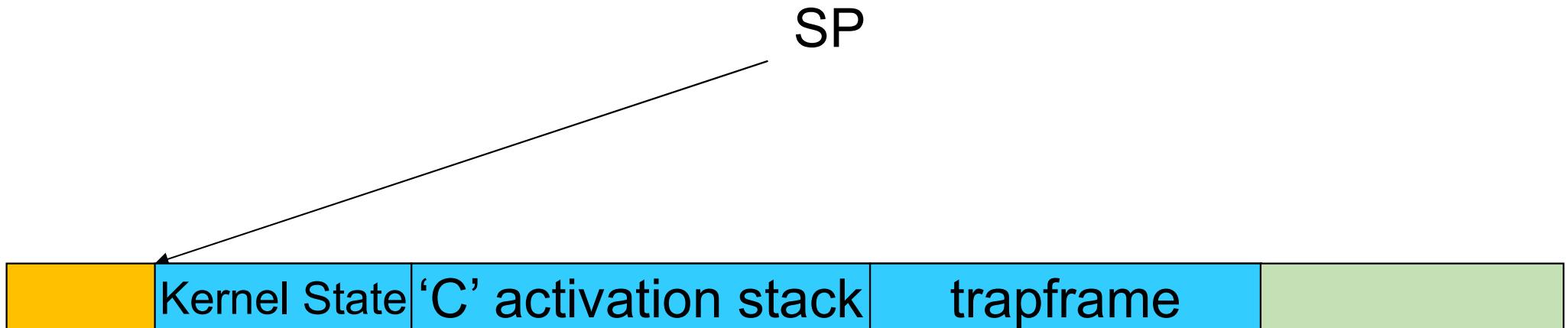
# Example Context Switch

- Call ‘C’ code to process syscall, exception, or interrupt
  - Results in a ‘C’ activation stack building up



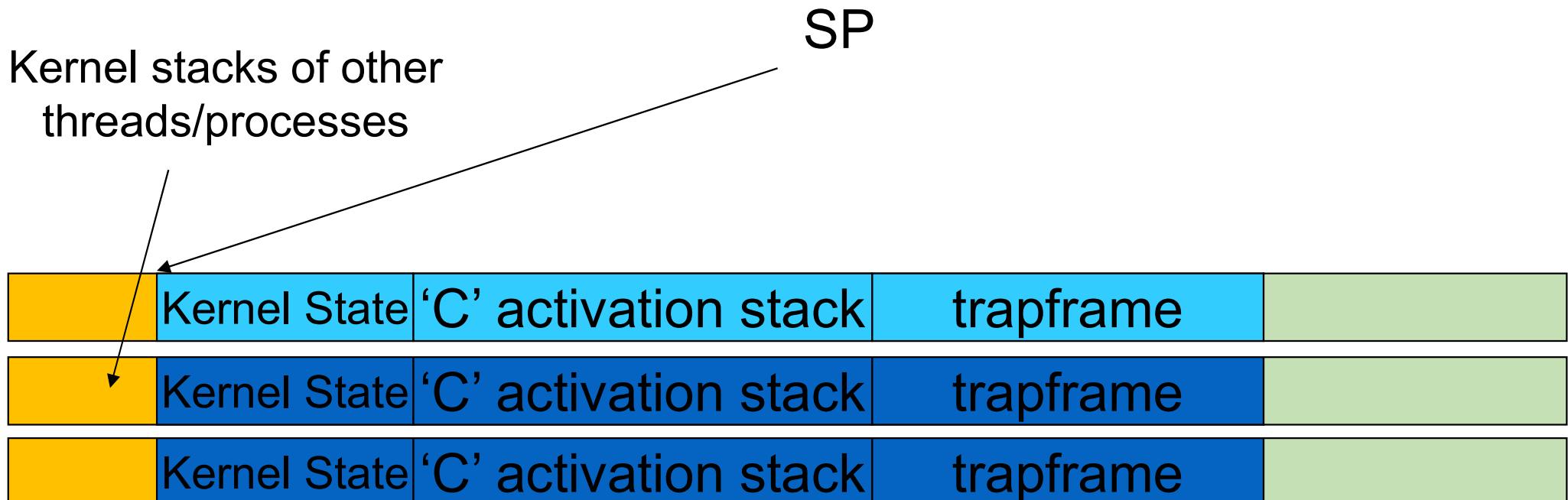
# Example Context Switch

- The kernel decides to perform a context switch
  - It chooses a target thread (or process)
  - It pushes remaining kernel context onto the stack



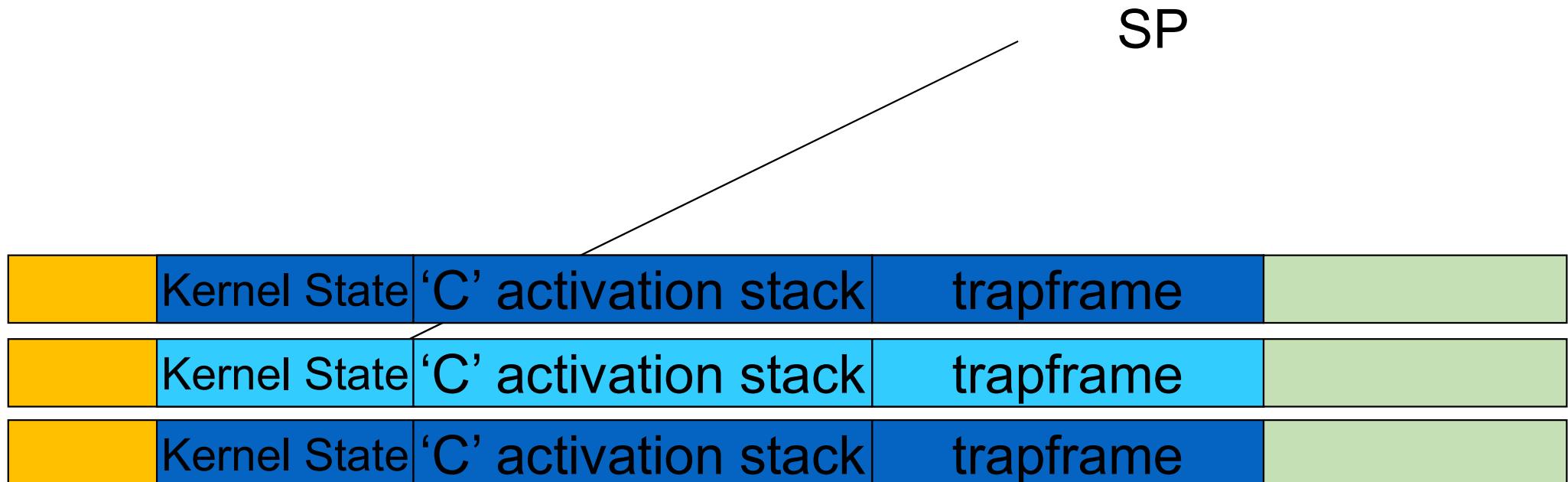
# Example Context Switch

- Any other existing thread must
  - be in kernel mode (on a uni processor),
  - and have a similar stack layout to the stack we are currently using



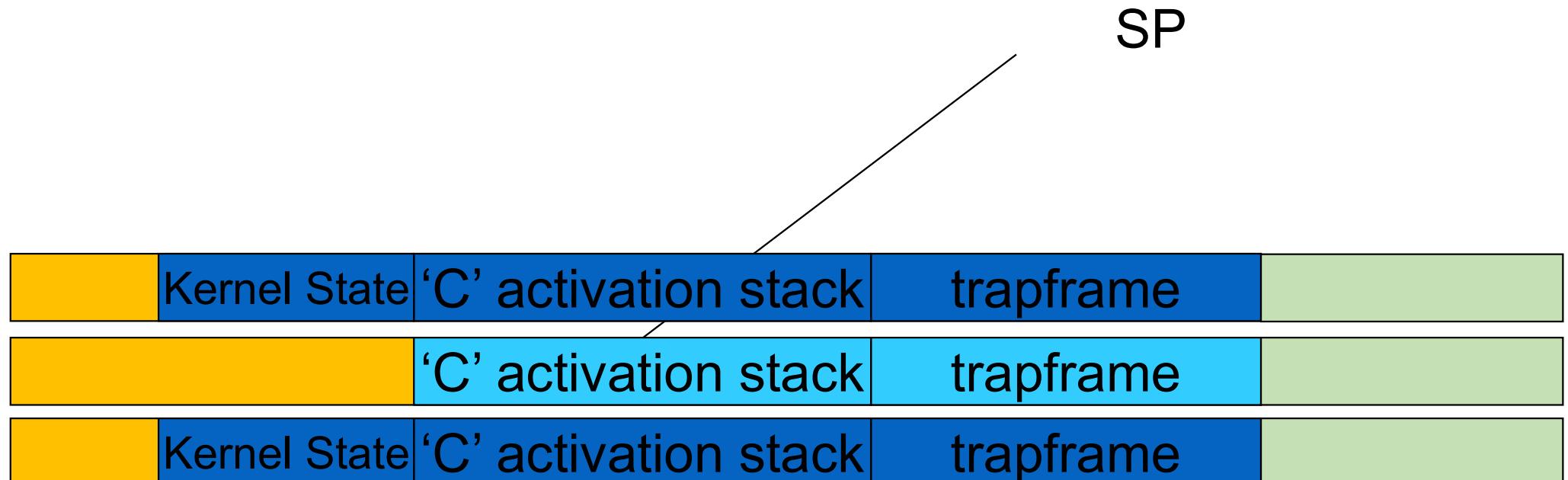
# Example Context Switch

- We save the current SP in the PCB (or TCB), and load the SP of the target thread.
  - Thus we have *switched contexts*



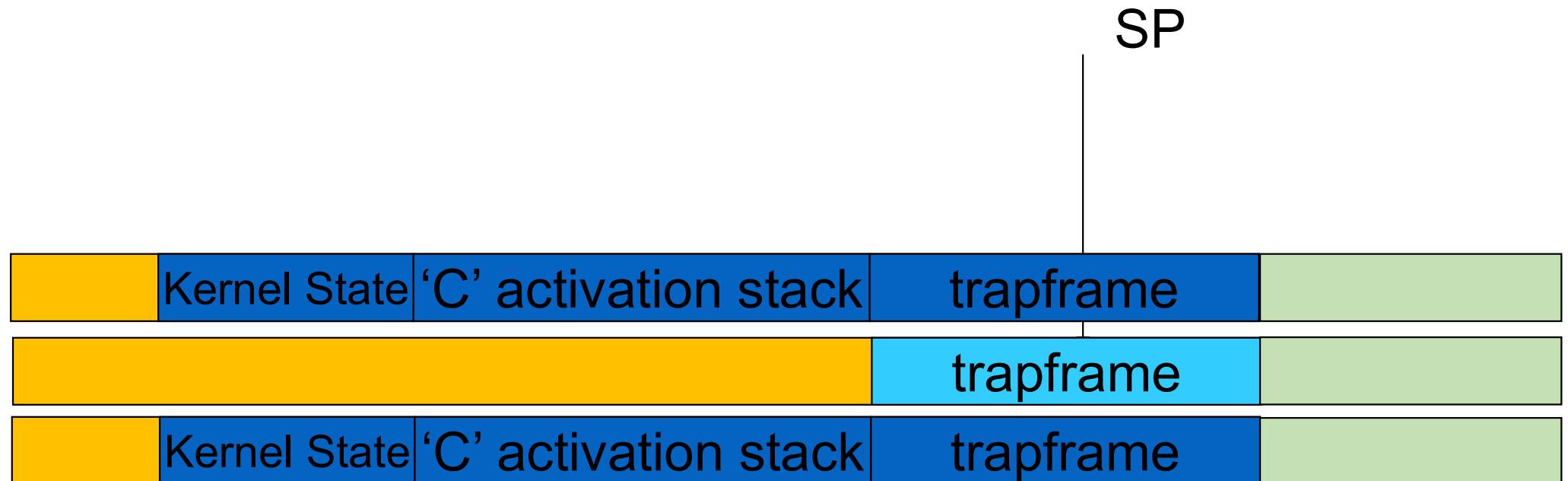
# Example Context Switch

- Load the target thread's previous context, and return to C



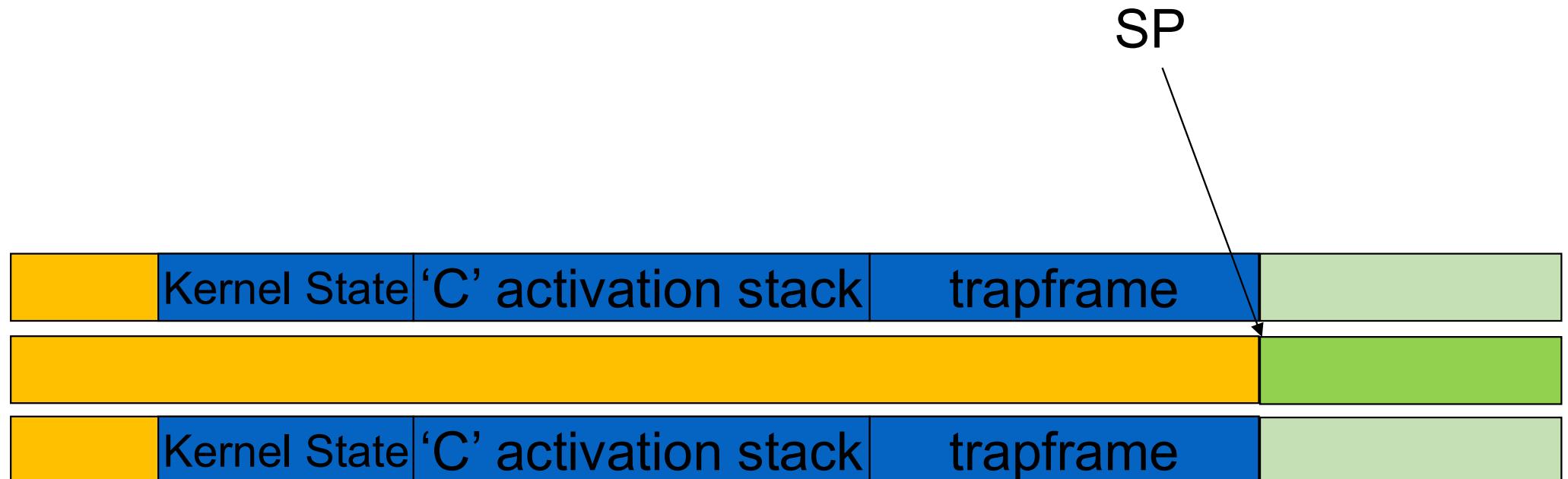
# Example Context Switch

- The C continues and (in this example) returns to user mode.



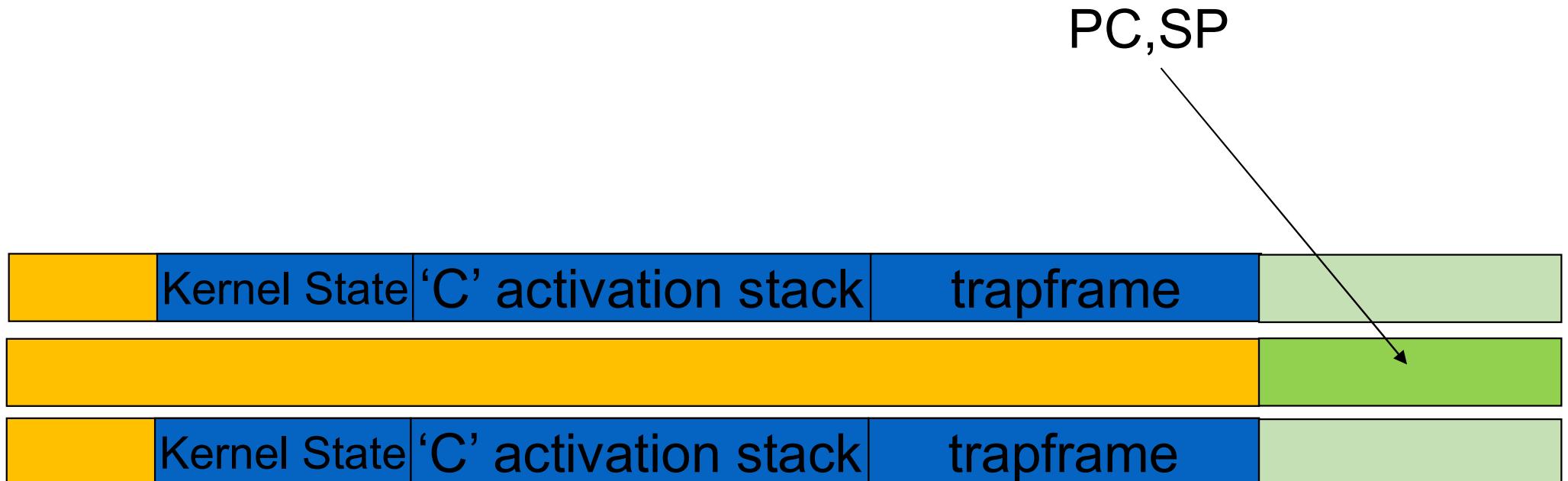
# Example Context Switch

- The user-level context is restored
  - The registers load with that process's previous content



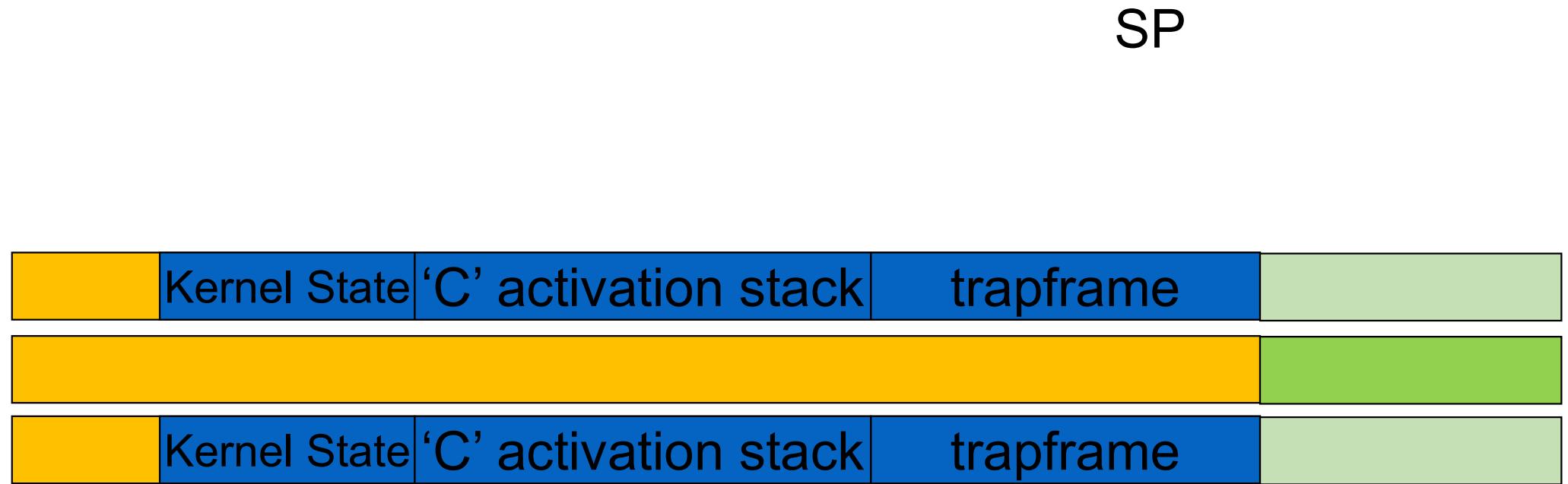
# Example Context Switch

- The user-level SP and PC is restored



# The Interesting Part of a Thread Switch

- What does the “push kernel state” part do???



# Simplified OS/161 thread\_switch

```
static
void
thread_switch(threadstate_t newstate, struct wchan *wc)
{
    struct thread *cur, *next;

    cur = curthread;
    do {
        next = threadlist_remhead(&curcpu->c_runqueue);
        if (next == NULL) {
            cpu_idle();
        }
    } while (next == NULL);

    /* do the switch (in assembler in switch.S) */
    switchframe_switch(&cur->t_context, &next->t_context);
}
```

Lots of code removed – only basics of pick next thread and switch to it remain

# OS/161 switchframe\_switch

## **switchframe\_switch:**

```
/*
 * a0 contains the address of the switchframe pointer in the old thread.
 * a1 contains the address of the switchframe pointer in the new thread.
 *
 * The switchframe pointer is really the stack pointer. The other
 * registers get saved on the stack, namely:
 *
 *      s0-s6, s8
 *      gp, ra
 *
 * The order must match <mips/switchframe.h>.
 *
 * Note that while we'd ordinarily need to save s7 too, because we
 * use it to hold curthread saving it would interfere with the way
 * curthread is managed by thread.c. So we'll just let thread.c
 * manage it.
 */
```

# OS/161 switchframe\_switch

```
/* Allocate stack space for saving 10 registers. 10*4 = 40 */  
addi sp, sp, -40
```

```
/* Save the registers */
```

```
sw ra, 36(sp)  
sw gp, 32(sp)  
sw s8, 28(sp)  
sw s6, 24(sp)  
sw s5, 20(sp)  
sw s4, 16(sp)  
sw s3, 12(sp)  
sw s2, 8(sp)  
sw s1, 4(sp)  
sw s0, 0(sp)
```

```
/* Store the old stack pointer in the old thread */  
sw sp, 0(a0)
```

Save the registers  
that the 'C'  
procedure calling  
convention  
expects  
preserved

# OS/161 switchframe\_switch

```
/* Get the new stack pointer from the new thread */
```

```
lw sp, 0(a1)
```

```
nop      /* delay slot for load */
```

```
/* Now, restore the registers */
```

```
lw s0, 0(sp)
```

```
lw s1, 4(sp)
```

```
lw s2, 8(sp)
```

```
lw s3, 12(sp)
```

```
lw s4, 16(sp)
```

```
lw s5, 20(sp)
```

```
lw s6, 24(sp)
```

```
lw s8, 28(sp)
```

```
lw gp, 32(sp)
```

```
lw ra, 36(sp)
```

```
nop      /* delay slot for load */
```

# OS/161 switchframe\_switch

```
/* and return. */  
j ra  
addi sp, sp, 40    /* in delay slot */
```

# Revisiting Thread Switch

Thread a                      Thread b

```
switchframe_switch(a,b) }  
{  
}  
← switchframe_switch(b,a)  
{
```

```
switchframe_switch(a,b) }  
{
```

# System Calls

Interface and Implementation

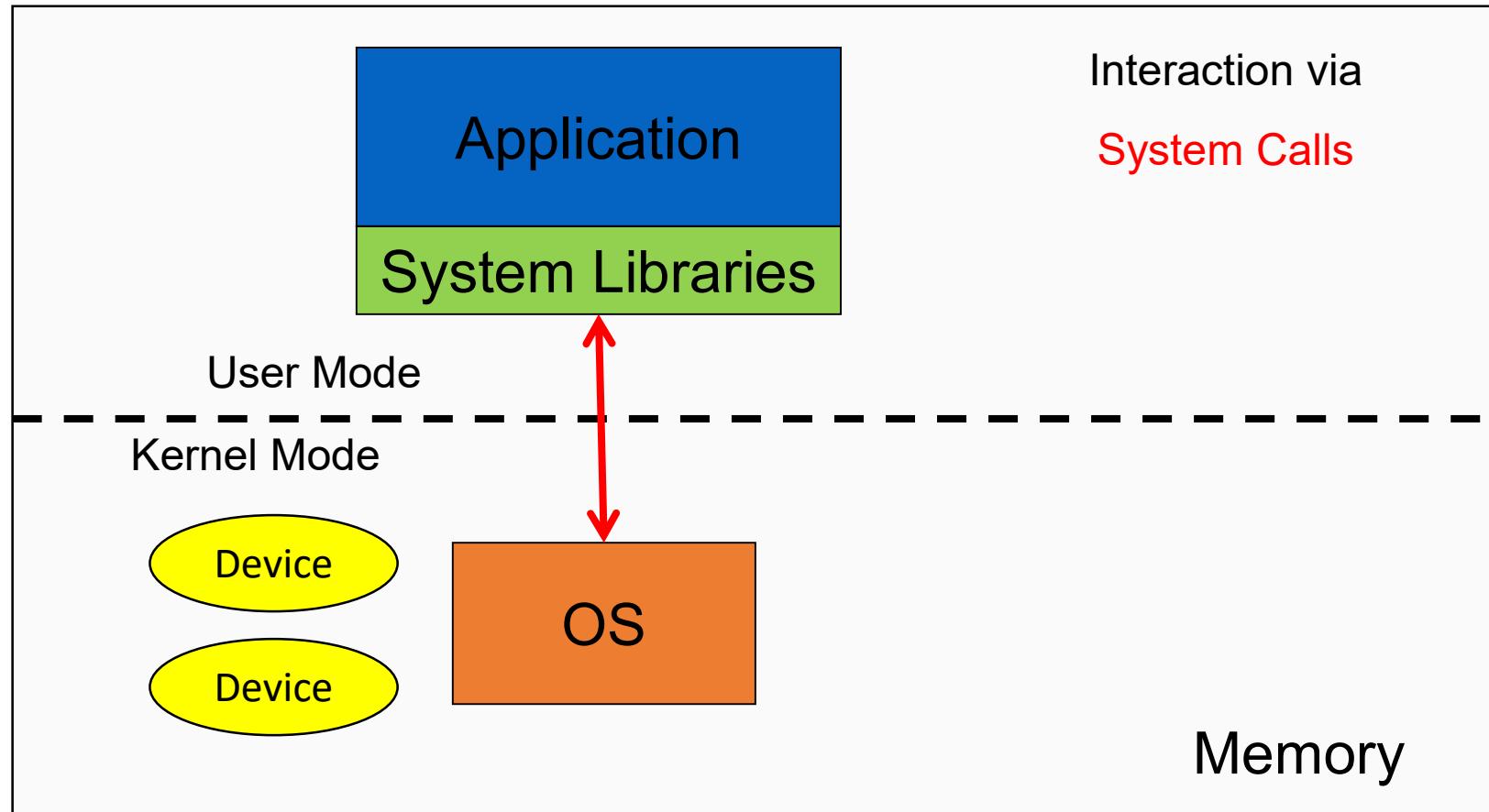
# Learning Outcomes

- A high-level understanding of *System Call* interface
  - Mostly from the user's perspective
    - From textbook (section 1.6)
- Understanding of how the application-kernel boundary is crossed with system calls in general
  - Including an appreciation of the relationship between a case study (OS/161 system call handling) and the general case.
- Exposure architectural details of the MIPS R3000
  - Detailed understanding of the exception handling mechanism
    - From "Hardware Guide" on class web site

# System Calls

Interface

# The Structure of a Computer System



# System Calls

- Can be viewed as special function calls
  - Provides for a controlled entry into the kernel
  - While in kernel, they perform a privileged operation
  - Returns to original caller with the result
- The system call interface represents the abstract machine provided by the operating system.

# The System Call Interface: A Brief Overview

- From the user's perspective
  - Process Management
  - File I/O
  - Directories management
  - Some other selected Calls
  - There are many more
    - On Linux, see **man syscalls** for a list

# Some System Calls For Process Management

## Process management

Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

# Some System Calls For File Management

**File management**

<b>Call</b>	<b>Description</b>
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &amp;buf)</code>	Get a file's status information

# System Calls

- A stripped down shell:

```
while (TRUE) {                                     /* repeat forever */  
    type_prompt( );                            /* display prompt */  
    read_command (command, parameters)        /* input from terminal */  
  
    if (fork() != 0) {                         /* fork off child process */  
        /* Parent code */  
        waitpid( -1, &status, 0);                /* wait for child to exit */  
    } else {  
        /* Child code */  
        execve (command, parameters, 0);        /* execute command */  
    }  
}
```

# System Calls

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Some Win32 API calls

# System Call Implementation

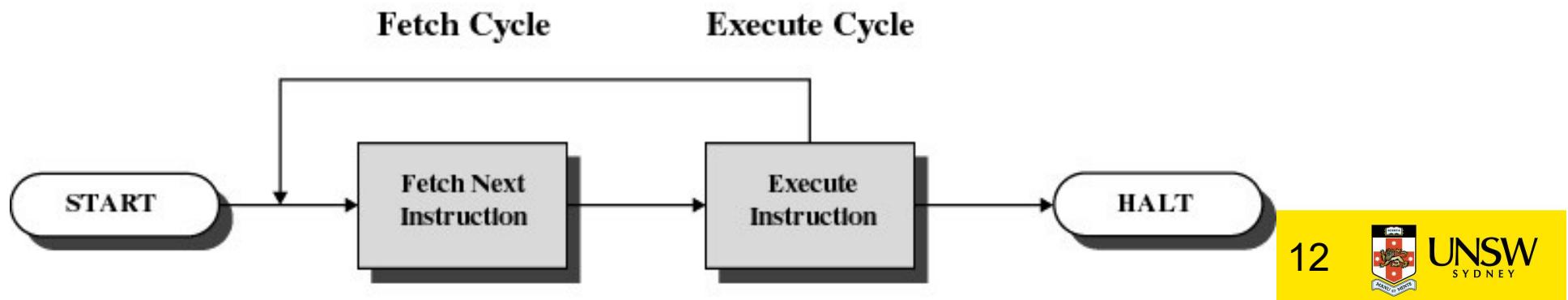
Crossing user-kernel boundary

# A Simple Model of CPU Computation

- The fetch-execute cycle
  - Load memory contents from address in program counter (PC)
    - The instruction
  - Execute the instruction
  - Increment PC
  - Repeat

CPU Registers

PC: 0x0300



# A Simple Model of CPU Computation

- Stack Pointer (SP)
- Status Register
  - Condition codes
    - Positive result
    - Zero result
    - Negative result
- General Purpose Registers
  - Holds operands of most instructions
  - Enables programmers (compiler) to minimise memory references.

CPU Registers

PC: 0x0300
SP: 0xcbf3
Status
R1
↓
Rn

# Privileged-mode Operation

- To protect operating system execution, two or more CPU modes of operation exist
  - Privileged mode (system-, kernel-mode)
    - All instructions and registers are available
  - User-mode
    - Uses ‘safe’ subset of the instruction set
      - Only affects the state of the application itself
      - They cannot be used to uncontrollably interfere with OS
    - Only ‘safe’ registers are accessible

## CPU Registers

Interrupt Mask
Exception Type
MMU regs
Others
PC: 0x0300
SP: 0xcbf3
Status
R1
↔
Rn

# Example Unsafe Instruction

- “cli” instruction on x86 architecture
  - Disables interrupts
- Example exploit

```
cli /* disable interrupts */  
while (true)  
/* loop forever */;
```

# Privileged-mode Operation

- The accessibility of addresses within an address space changes depending on operating mode
  - To protect kernel code and data
- Note: The exact memory ranges are usually configurable, and vary between CPU architectures and/or operating systems.

Memory Address Space

0xFFFFFFFF

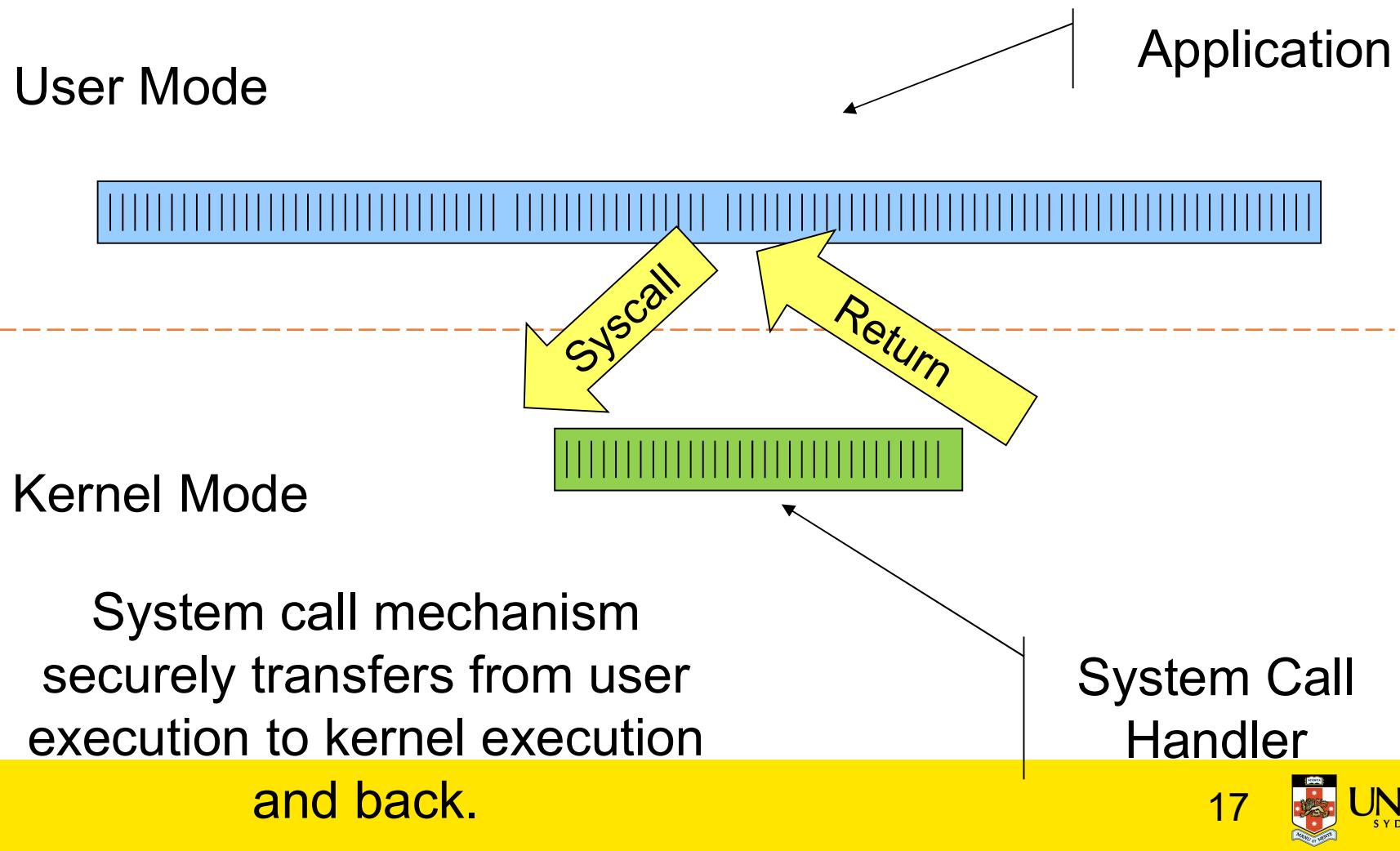
Accessible only  
to  
Kernel-mode

0x80000000

Accessible to  
User- and  
Kernel-mode

0x00000000

# System Call



# Questions we'll answer

- There is only one register set
  - How is register use managed?
  - What does an application expect a system call to look like?
- How is the transition to kernel mode triggered?
- Where is the OS entry point (system call handler)?
- How does the OS know what to do?

# System Call Mechanism Overview

- System call transitions triggered by special processor instructions
  - User to Kernel
    - System call instruction
  - Kernel to User
    - Return from privileged mode instruction

# System Call Mechanism Overview

- Processor mode
  - Switched from user-mode to kernel-mode
    - Switched back when returning to user mode
- Stack Pointer (SP)
  - User-level SP is saved and a kernel SP is initialised
    - User-level SP restored when returning to user-mode
- Program Counter (PC)
  - User-level PC is saved and PC set to kernel entry point
    - User-level PC restored when returning to user-level
  - Kernel entry via the designated entry point must be strictly enforced

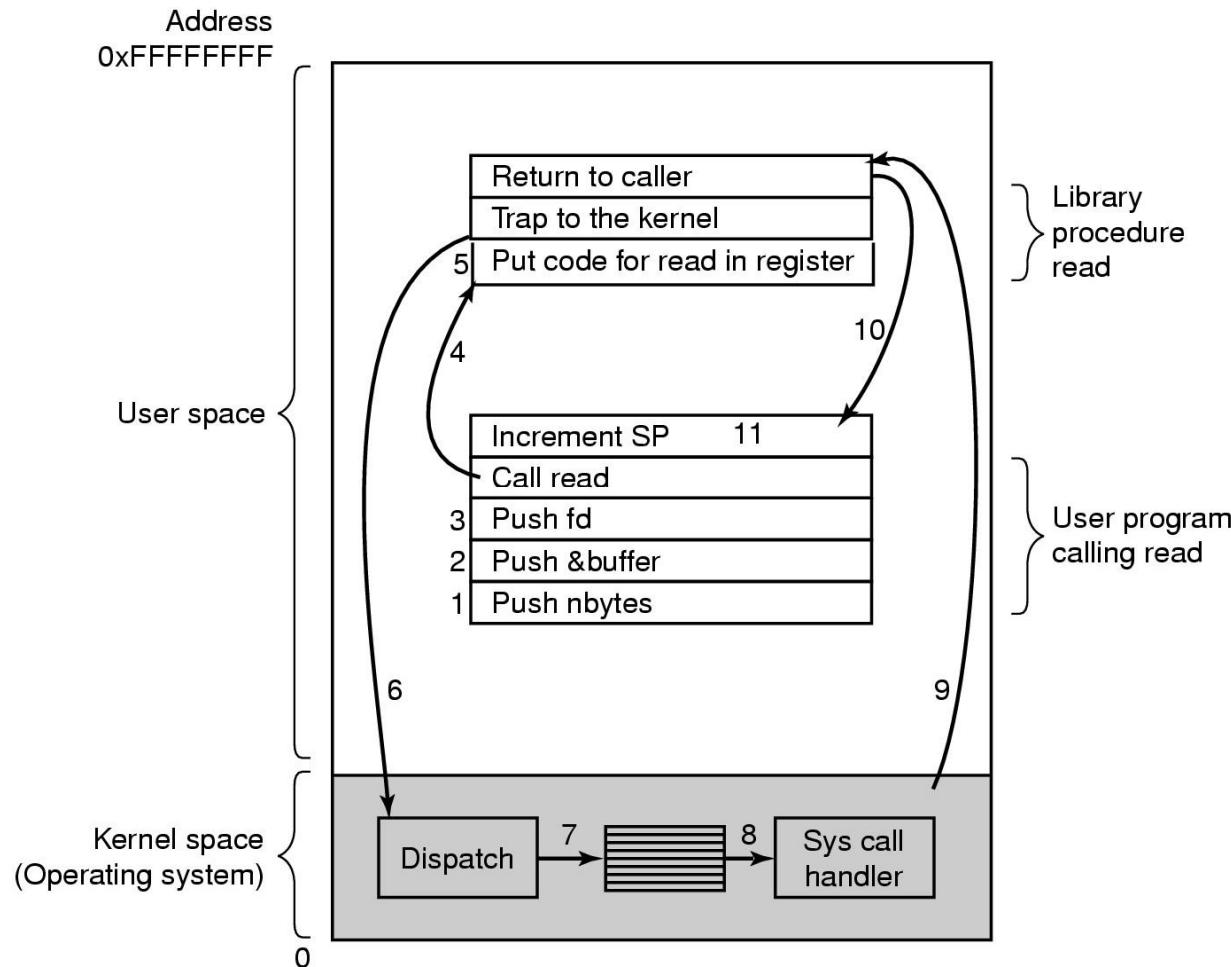
# System Call Mechanism Overview

- Registers
  - Set at user-level to indicate system call type and its arguments
    - A convention between applications and the kernel
  - Some registers are preserved at user-level or kernel-level in order to restart user-level execution
    - Depends on language calling convention etc.
  - Result of system call placed in registers when returning to user-level
    - Another convention

# Why do we need system calls?

- Why not simply jump into the kernel via a function call????
  - Function calls do not
    - Change from user to kernel mode
      - and eventually back again
    - Restrict possible entry points to secure locations
      - To prevent entering after any security checks

# Steps in Making a System Call



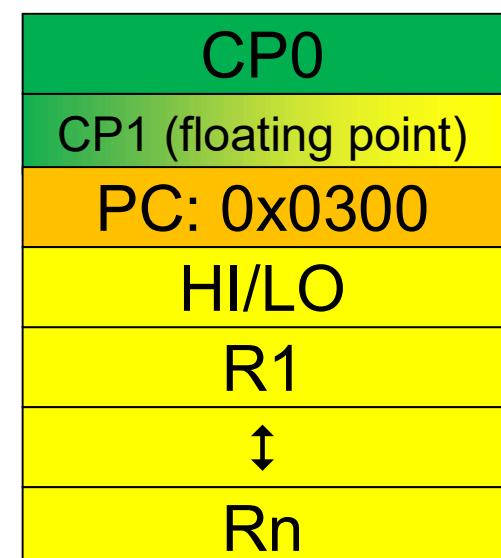
There are 11 steps in making the system call  
read (fd, buffer, nbytes)

# The MIPS R2000/R3000

- Before looking at system call mechanics in some detail, we need a basic understanding of the MIPS R3000

# Coprocessor 0

- The processor control registers are located in CP0
  - Exception/Interrupt management registers
  - Translation management registers
- CP0 is manipulated using mtc0 (move to) and mfc0 (move from) instructions
  - mtc0/mfc0 are only accessible in kernel mode.



# CPO Registers

- Exception Management
  - c0\_cause
    - Cause of the recent exception
  - c0\_status
    - Current status of the CPU
  - c0\_epc
    - Address of the instruction that caused the exception
  - c0\_badvaddr
    - Address accessed that caused the exception
- Miscellaneous
  - c0\_prid
    - Processor Identifier
- Memory Management
  - c0\_index
  - c0\_random
  - c0\_entryhi
  - c0\_entrylo
  - c0\_context
  - More about these later in course

# c0\_status

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CU3	CU2	CU1	CU0	0	RE	0	BEV	TS	PE	CM	PZ	SwC	IsC		
15					8	7	6	5	4	3	2	1	0		
	IM				0	KUo	IEo	KUp	IEp	KUc	IEc				

Figure 3.2. Fields in status register (SR)

- For practical purposes, you can ignore most bits
  - Green background is the focus

# c0\_status

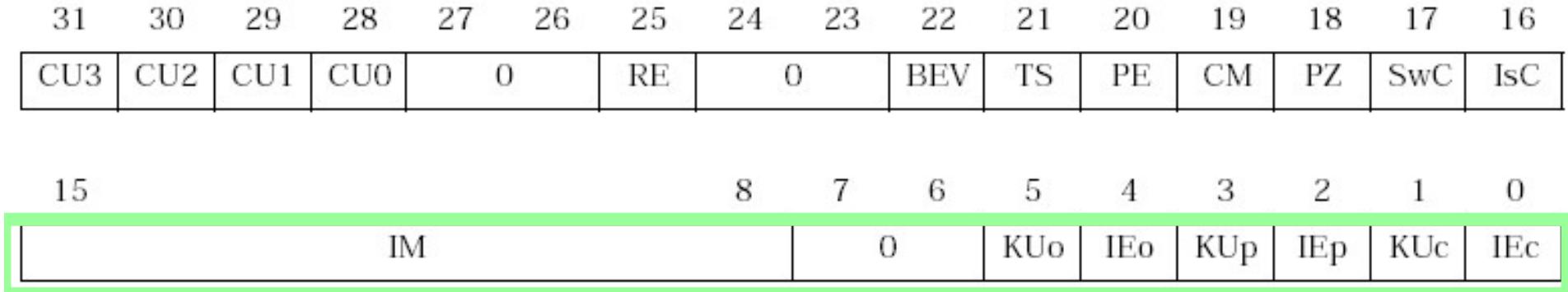


Figure 3.2. Fields in status register (SR)

- IM

- Individual interrupt mask bits
- 6 external
- 2 software

- KU

- 0 = kernel
- 1 = user mode

- IE

- 0 = all interrupts masked
- 1 = interrupts enable
- Mask determined via IM bits

- c, p, o = current, previous, old

# c0\_cause

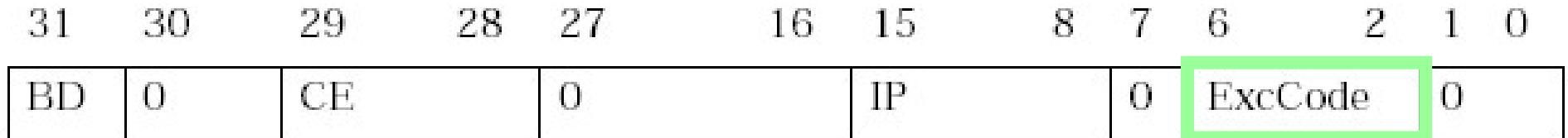


Figure 3.3. Fields in the Cause register

- IP
  - Interrupts pending
    - 8 bits indicating current state of interrupt lines
- CE
  - Coprocessor error
    - Attempt to access disabled Copro.
- BD
  - If set, the instruction that caused the exception was in a branch delay slot
- ExcCode
  - The code number of the exception taken

# Exception Codes

ExcCode Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	"TLB modification"
2	TLBL	"TLB load/TLB store"
3	TLBS	
4	AdEL	Address error (on load/I-fetch or store respectively). Either an attempt to access outside kuseg when in user mode, or an attempt to read a word or half-word at a misaligned address.
5	AdES	

Table 3.2. ExcCode values: different kinds of exceptions

# Exception Codes

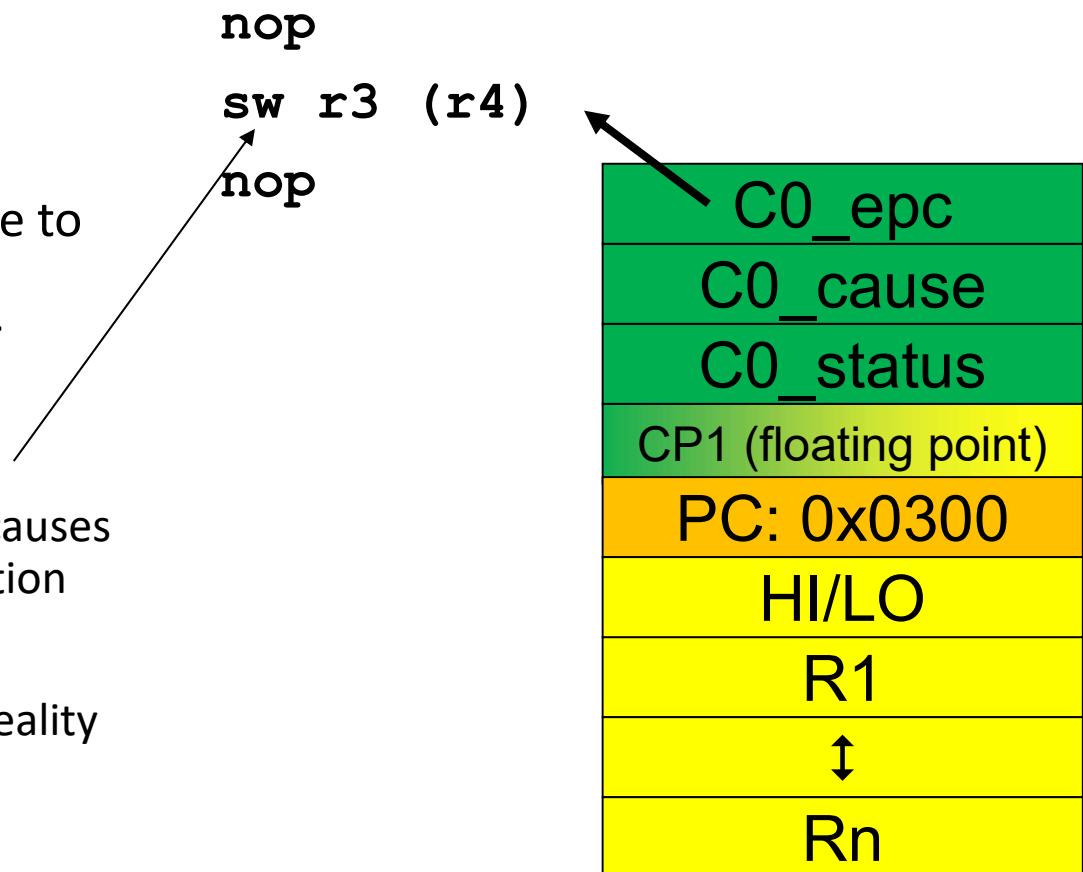
ExcCode Value	Mnemonic	Description
6	IBE	Bus error (instruction fetch or data load, respectively). External hardware has signalled an error of some kind; proper exception handling is system-dependent. The R30xx family CPUs can't take a bus error on a store; the write buffer would make such an exception "imprecise".
8	Syscall	Generated unconditionally by a <i>syscall</i> instruction.
9	Bp	Breakpoint - a <i>break</i> instruction.
10	RI	"reserved instruction"
11	CpU	"Co-Processor unusable"
12	Ov	"arithmetic overflow". Note that "unsigned" versions of instructions (e.g. <i>addu</i> ) never cause this exception.
13-31	-	reserved. Some are already defined for MIPS CPUs such as the R6000 and R4xxx

Table 3.2. ExcCode values: different kinds of exceptions

# c0\_epc

- The Exception Program Counter
  - Points to address of where to restart execution after handling the exception or interrupt
  - Example
    - Assume **sw r3, (r4)** causes a restartable fault exception

Aside: We are ignore BD-bit in c0\_cause which is also used in reality on rare occasions.

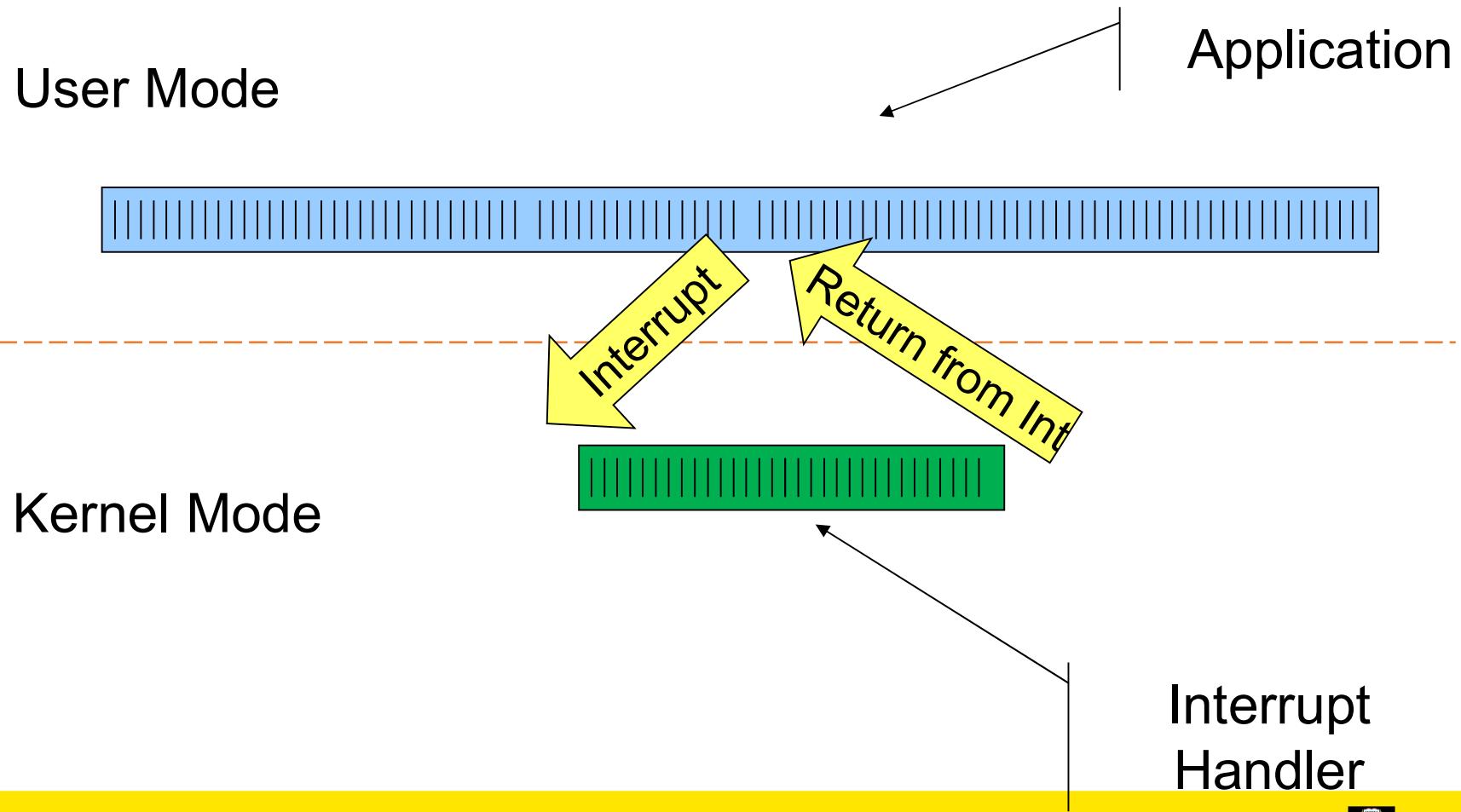


# Exception Vectors

Program address	“segment”	Physical Address	Description
0x8000 0000	kseg0	0x0000 0000	TLB miss on kuseg reference only.
0x8000 0080	kseg0	0x0000 0080	All other exceptions.
0xbfc0 0100	kseg1	0x1fc0 0100	Uncached alternative kuseg TLB miss entry point (used if SR bit BEV set).
0xbfc0 0180	kseg1	0x1fc0 0180	Uncached alternative for all other exceptions, used if SR bit BEV set).
0xbfc0 0000	kseg1	0x1fc0 0000	The “reset exception”.

Table 4.1. Reset and exception entry points (vectors) for R30xx family

# Simple Exception Walk-through



# Hardware exception handling

PC

0x12345678

EPC

?

Cause

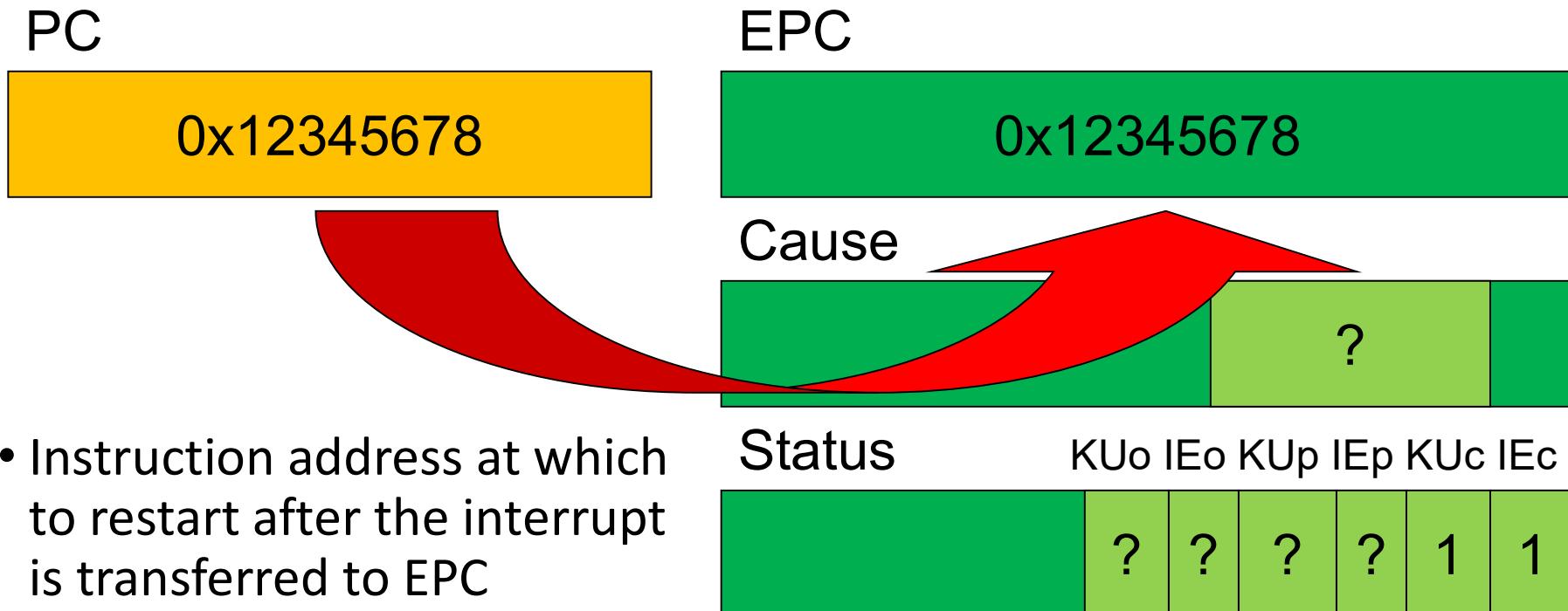
?

Status

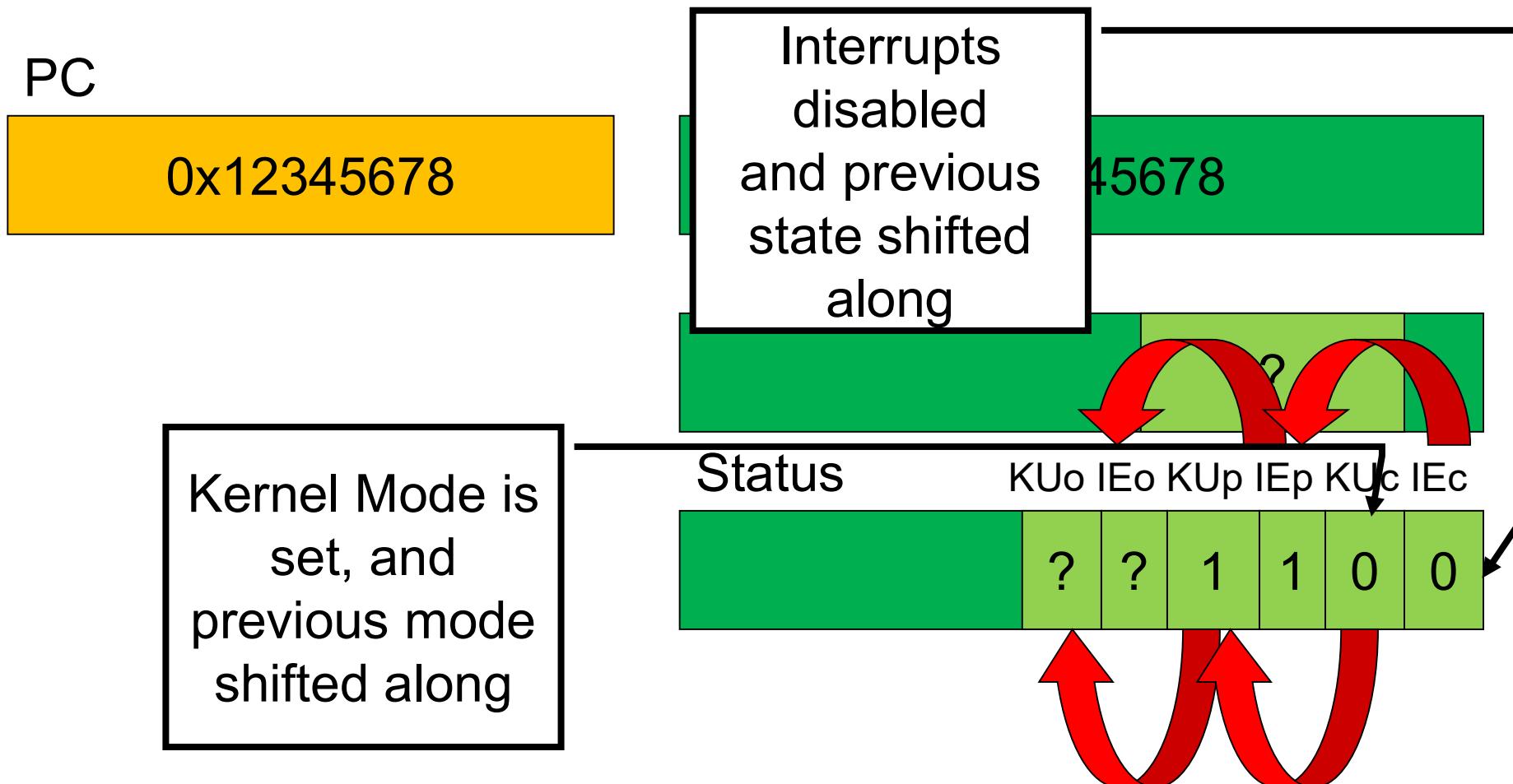
KUo IEo KUp IEp KUc IEc

?	?	?	?	1	1
---	---	---	---	---	---

# Hardware exception handling



# Hardware exception handling



# Hardware exception handling

PC



EPC



Cause

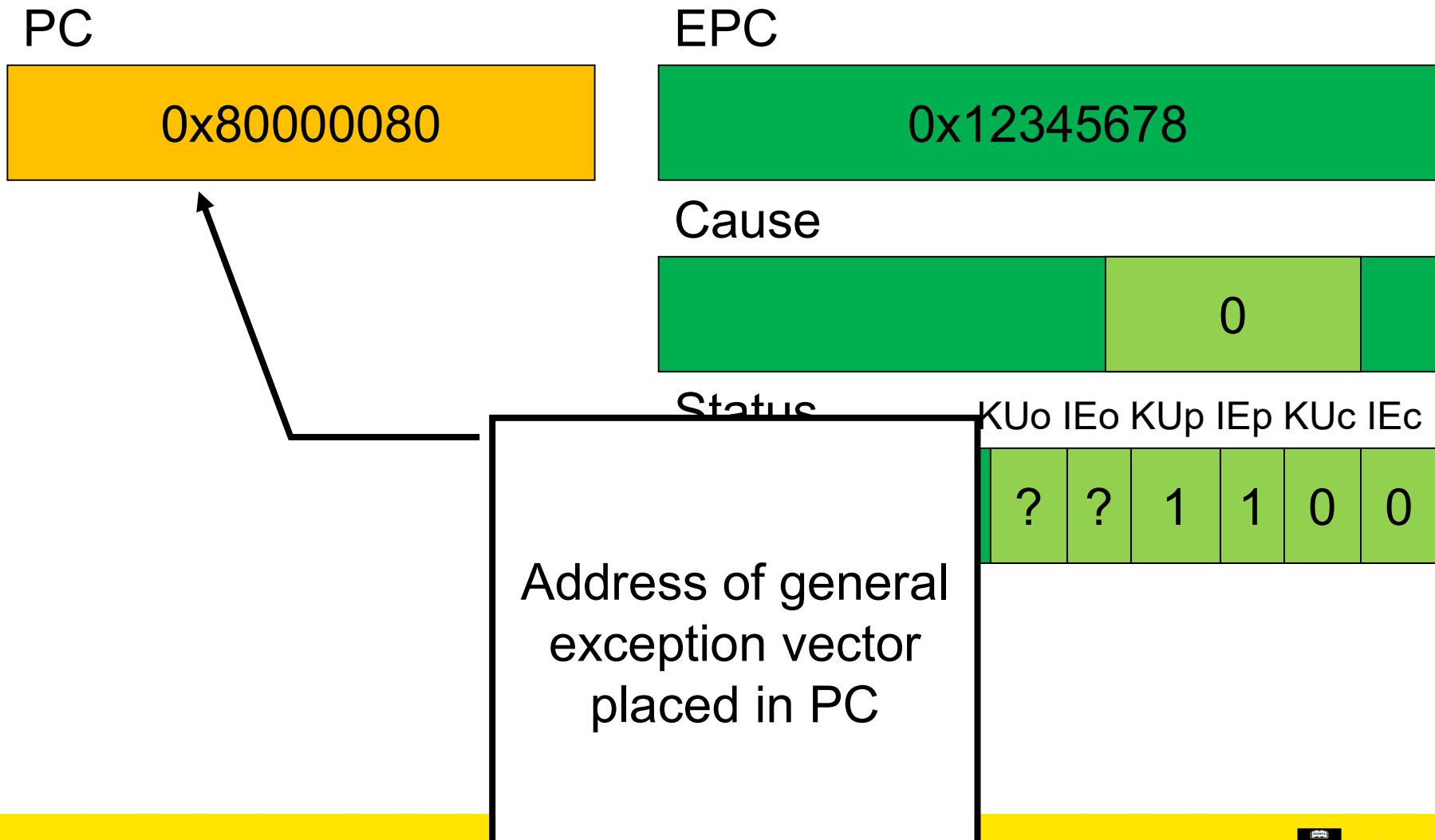


Status



Code for the  
exception placed in  
Cause. Note  
Interrupt code = 0

# Hardware exception handling



# Hardware exception handling

PC

0x80000080

- CPU is now running in kernel mode at 0x80000080, with interrupts disabled
  - All information required to
    - Find out what caused the exception
    - Restart after exception handling
- is in coprocessor registers

EPC

0x12345678

Cause

		0	
--	--	---	--

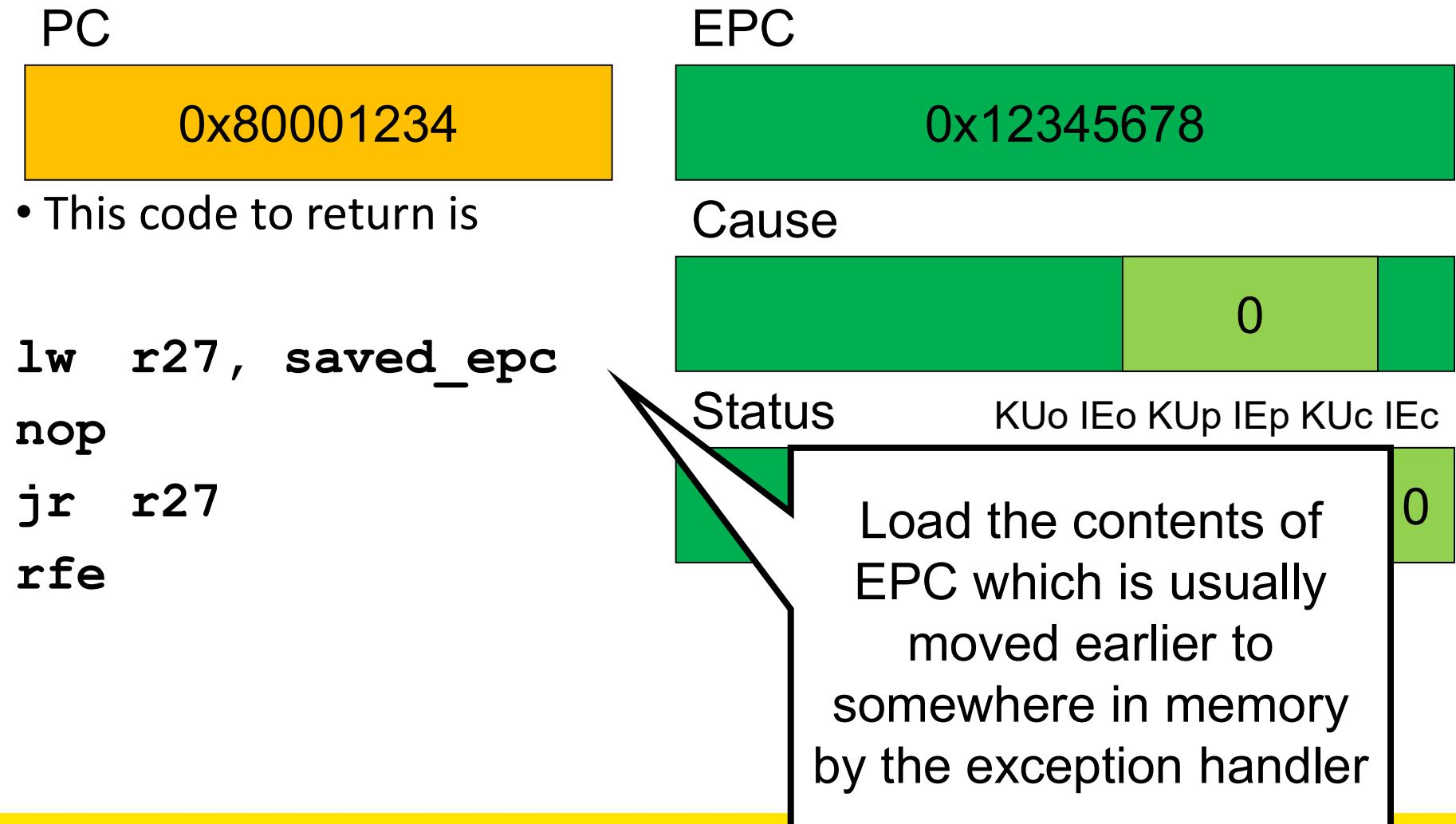
Status

KUo	IEo	KUp	IEp	KUc	IEc
	?	?	1	1	0 0

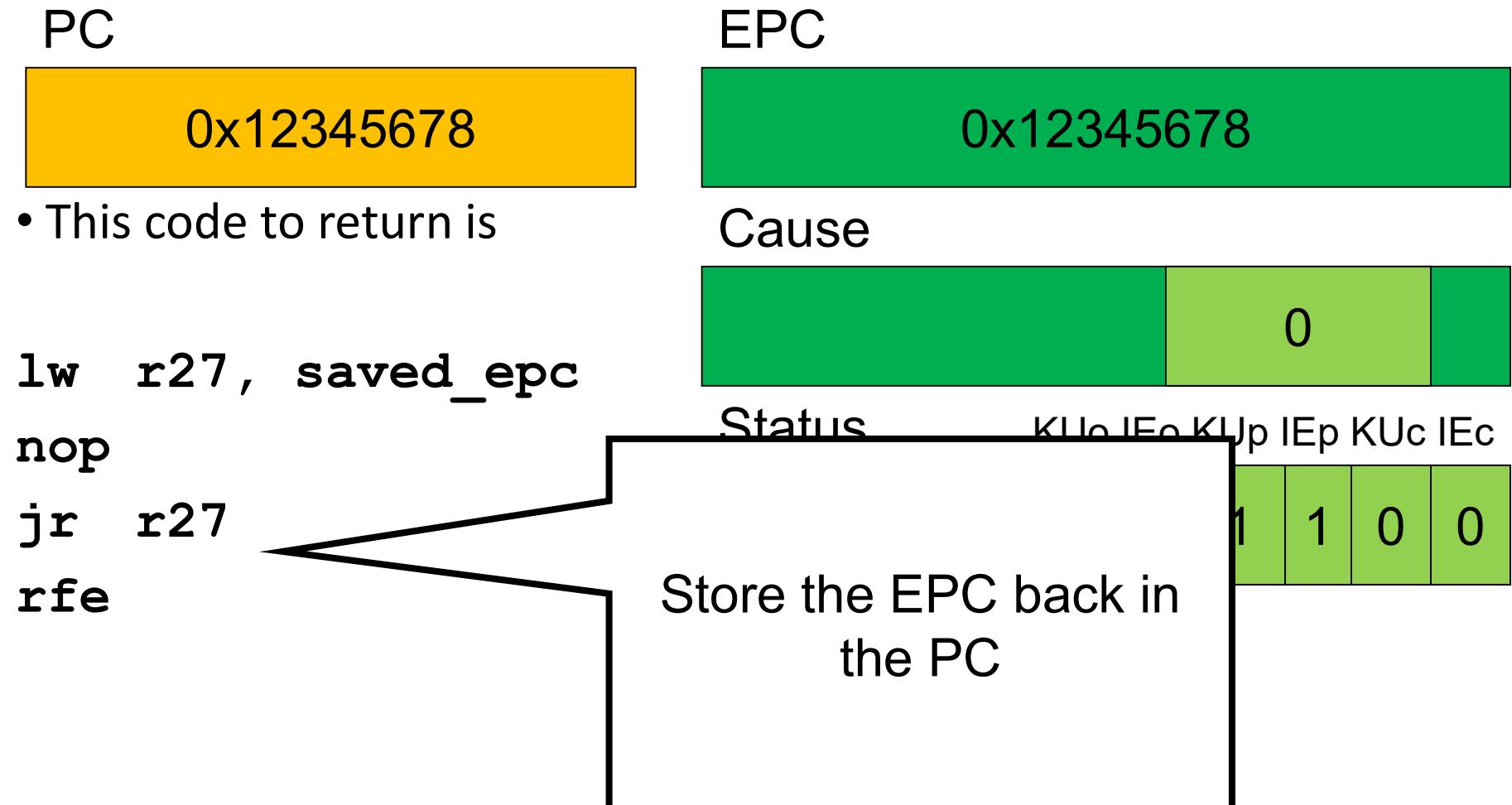
# Returning from an exception

- For now, let's ignore
  - how the exception is actually handled
  - how user-level registers are preserved
- Let's simply look at how we return from the exception

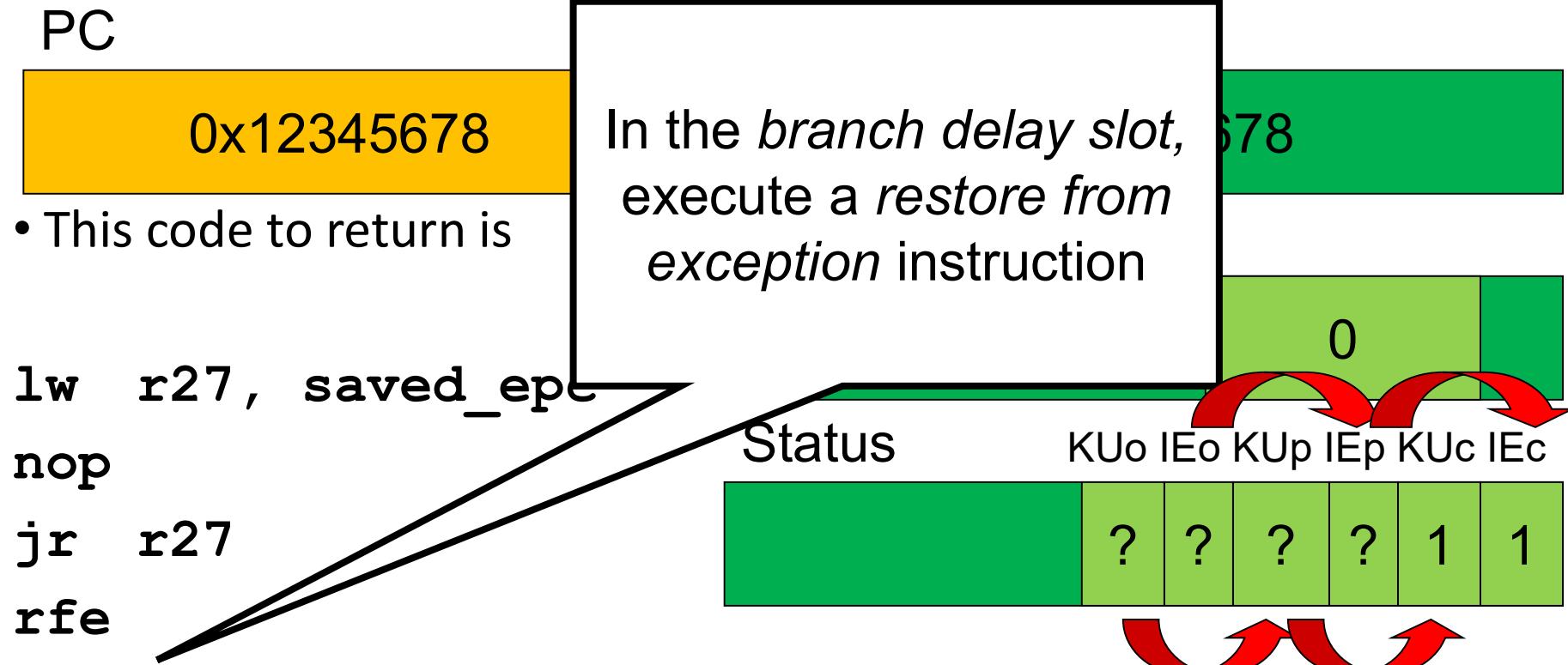
# Returning from an exception



# Returning from an exception



# Returning from an exception



# Returning from an exception

PC

0x12345678

EPC

0x12345678

- We are now back in the same state we were in when the exception happened

Cause

	0	
--	---	--

Status

KUo IEo KUp IEp KUc IEc

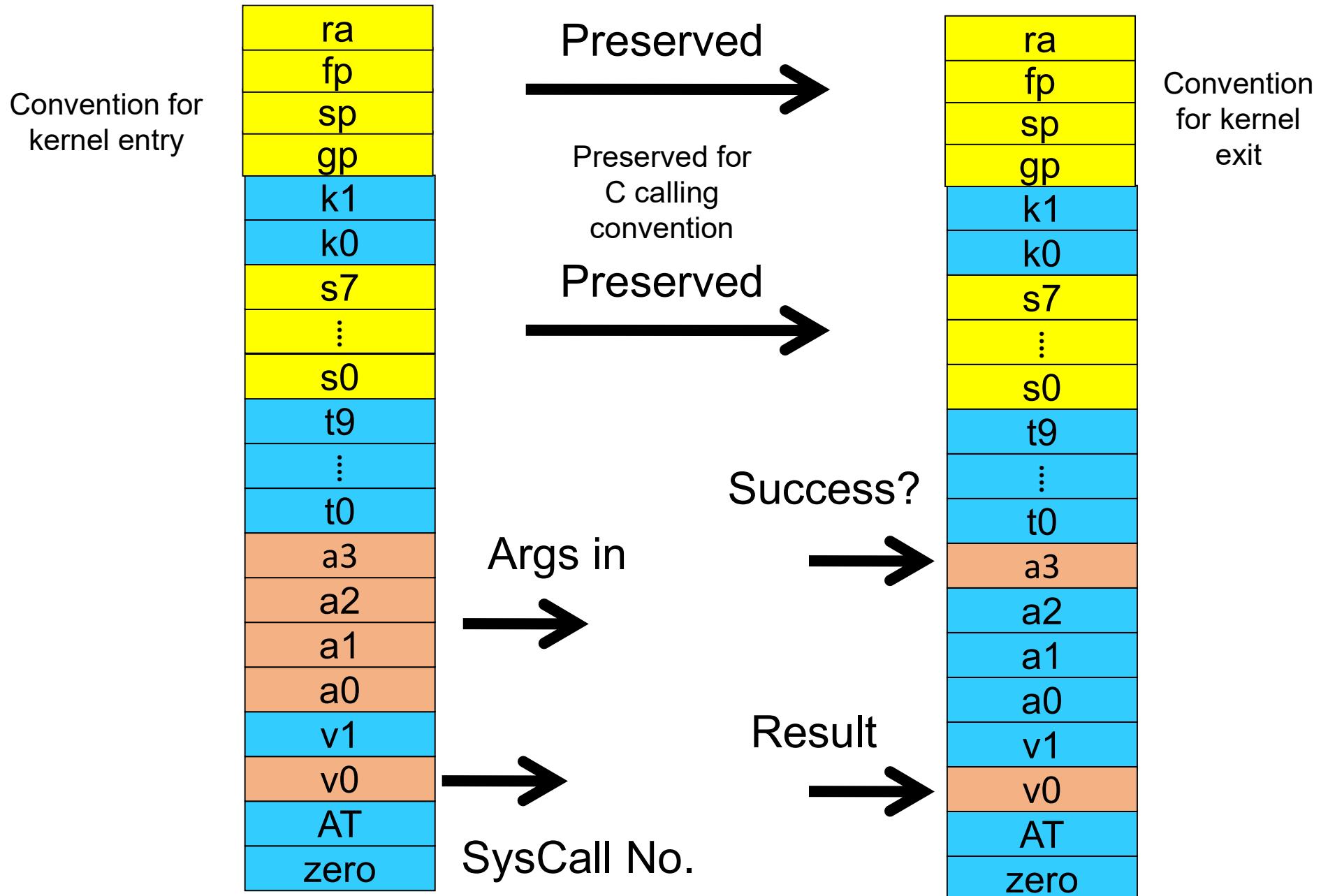
	?	?	?	?	1	1
--	---	---	---	---	---	---

# MIPS System Calls

- System calls are invoked via a *syscall* instruction.
  - The *syscall* instruction causes an exception and transfers control to the general exception handler
  - A convention (an agreement between the kernel and applications) is required as to how user-level software indicates
    - Which system call is required
    - Where its arguments are
    - Where the result should go

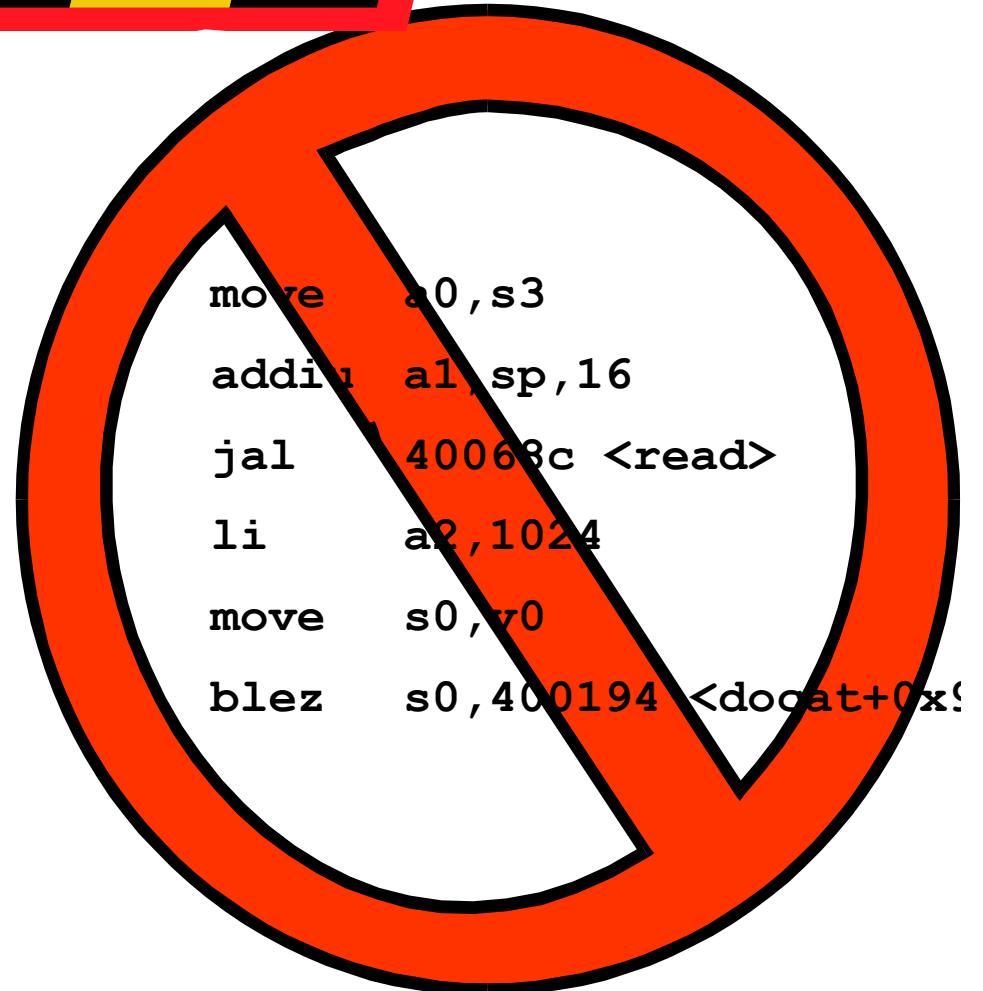
# OS/161 Systems Calls

- OS/161 uses the following conventions
  - Arguments are passed and returned via the normal C function calling convention
  - Additionally
    - Reg v0 contains the system call number
    - On return, reg a3 contains
      - 0: if success, v0 contains successful result
      - not 0: if failure, v0 has the errno.
        - v0 stored in errno
        - -1 returned in v0



# CAUTION

- Seriously low-level code follows
- This code is not for the faint hearted



# User-Level System Call Walk Through – Calling read()

```
int read(int filehandle, void *buffer, size_t size)
```

- Three arguments, one return value
- Code fragment calling the read function

400124:	02602021	move a0,s3
400128:	27a50010	addiu a1,sp,16
40012c:	0c1001a3	jal 40068c <read>
400130:	24060400	li a2,1024
400134:	00408021	move s0,v0
400138:	1a000016	blez s0,400194
<docat+0x94>		

- Args are loaded, return value is tested

# Inside the read() syscall function

## part 1

0040068c <read>:

```
40068c: 08100190    j 400640
<__syscall>
400690: 24020005    li v0,5
```

- Appropriate registers are preserved
  - Arguments (a0-a3), return address (ra), etc.
- The syscall number (5) is loaded into v0
- Jump (not jump and link) to the common syscall routine

# The read() syscall function part 2

00400640 <\_\_syscall>:

400640:	0000000c	syscall
400644:	10e00005	beqz a3,40065c <__syscall+0x1c>
400648:	00000000	nop
40064c:	3c011000	lui at,0x1000
400650:	ac220000	sw v0,0(at)
400654:	2403ffff	li v1,-1
400658:	2402ffff	li v0,-1
40065c:	03e00008	jr ra
400660:	00000000	nop

Generate a syscall exception

# The read() syscall function part 2

00400640 <\_\_syscall>:

400640:	0000000c	syscall
400644:	10e00005	beqz a3, 40065c <__syscall+0x1c>
400648:	00000000	nop
40064c:	3c011000	lui at, 0x1000
400650:	ac220000	sw v0, 0(at)
400654:	2403ffff	li v1, -1
400658:	2402ffff	li v0, -1
40065c:	03e00008	jr ra
400660:	00000000	nop

Test success, if yes,  
branch to return  
from function

# The read() syscall function part 2

00400640 <\_\_syscall>:

400640:	0000000c	syscall
400644:	10e00005	beqz a3, 400650
400648:	00000000	nop
40064c:	3c011000	lui at, 0x1000
400650:	ac220000	sw v0, 0(at)
400654:	2403ffff	li v1, -1
400658:	2402ffff	li v0, -1
40065c:	03e00008	jr ra
400660:	00000000	nop

If failure, store code  
in *errno*

# The read() syscall function part 2

```
00400640 <__syscall>:  
 400640: 0000000c      syscall  
 400644: 10e00005      beqz a3, 400650  
 400648: 00000000      nop  
 40064c: 3c011000      lui   at,0x1000  
 400650: ac220000      sw    v0,0(a)  
 400654: 2403ffff      li    v1,-1  
 400658: 2402ffff      li    v0,-1  
 40065c: 03e00008      jr    ra  
 400660: 00000000      nop
```

Set read() result to  
-1

# The read() syscall function part 2

```
00400640 <__syscall>:  
 400640: 0000000c      syscall  
 400644: 10e00005      beqz a3,400658  
 400648: 00000000      nop  
 40064c: 3c011000      lui  at,0x1000  
 400650: ac220000      sw   v0,0(at)  
 400654: 2403ffff      li   v1,-1  
 400658: 2402ffff      li   v0,1  
 40065c: 03e00008      jr   ra  
 400660: 00000000      nop
```

Return to location  
after where read()  
was called

# Summary

- From the caller's perspective, the read() system call behaves like a normal function call
  - It preserves the calling convention of the language
- However, the actual function implements its own convention by agreement with the kernel
  - Our OS/161 example assumes the kernel preserves appropriate registers(s0-s8, sp, gp, ra).
- Most languages have similar *libraries* that interface with the operating system.

# System Calls - Kernel Side

- Things left to do
  - Change to kernel stack
  - Preserve registers by saving to memory (on the kernel stack)
  - Leave saved registers somewhere accessible to
    - Read arguments
    - Store return values
  - Do the “read()”
  - Restore registers
  - Switch back to user stack
  - Return to application

# OS/161 Exception Handling

- Note: The following code is from the uniprocessor variant of OS161 (v1.x).
  - Simpler, but broadly similar to current version.

```
exception:
```

```
    move k1, sp          /* Save previous stack pointer in k1 */
    mfc0 k0, c0_status   /* Get status register */
    andi k0, k0, CST_NI  /* Check the we-were-in-user-mode bit */
    beq     k0, $0, 1f    /* If clear, from kernel, already have stack */
    nop                 /* delay slot */

    /* Coming from user mode */
    la k0, curkstack
    lw sp, 0(k0)
    nop

1:
    mfc0 k0, c0_cause   /* Note cause. */
    j common_exception
    nop
```

Note k0, k1  
registers  
available for  
kernel use

exception:

```
move k1, sp          /* Save previous stack pointer in k1 */
mfc0 k0, c0_status  /* Get status register */
andi k0, k0, CST_Kup /* Check the we-were-in-user-mode bit */
beq    k0, $0, 1f /* If clear, from kernel, already have stack */
nop                /* delay slot */

/* Coming from user mode - load kernel stack into sp */
la k0, curkstack   /* get address of "curkstack" */
lw sp, 0(k0)        /* get its value */
nop                /* delay slot for the load */
```

1:

```
mfc0 k0, c0_cause /* Now, load the exception cause. */
j common_exception /* Skip to common code */
nop                /* delay slot */
```

```
common_exception:
```

```
/*
 * At this point:
 *     Interrupts are off. (The processor did this for us.)
 *     k0 contains the exception cause value.
 *     k1 contains the old stack pointer.
 *     sp points into the kernel stack.
 *     All other registers are untouched.
 */
```

```
/*
 * Allocate stack space for 37 words to hold the trap frame,
 * plus four more words for a minimal argument block.
 */
```

```
addi sp, sp, -164
```

```
/* The order here must match mips/include/trapframe.h. */

sw ra, 160(sp) /* dummy for gdb */
sw s8, 156(sp) /* save s8 */
sw sp, 152(sp) /* dummy for gdb */
sw gp, 148(sp) /* save gp */
sw k1, 144(sp) /* dummy for gdb */
sw k0, 140(sp) /* dummy for gdb */

sw k1, 152(sp) /* real saved sp */
nop /* delay slot for store */

mfco k1, c0_epc /* Copr.0 reg 13 == PC for
sw k1, 160(sp) /* real saved PC */
```

These six stores are  
a “hack” to avoid  
confusing GDB  
You can ignore the  
details of why and  
how

```
/* The order here must match mips/include/trapframe.h. */
```

```
sw ra, 160(sp) /* dummy for gdb */
sw s8, 156(sp) /* save s8 */
sw sp, 152(sp) /* dummy for gdb */
sw gp, 148(sp) /* save gp */
sw k1, 144(sp) /* dummy for gdb */
sw k0, 140(sp) /* dummy for gdb */

sw k1, 152(sp) /* real saved sp */
nop /* delay slot for store */

mfcc0 k1, c0_epc /* Copr.0 reg 13 == PC for exception */
sw k1, 160(sp) /* real saved PC */
```

The real work starts  
here

```
sw t9, 136(sp)
sw t8, 132(sp)
sw s7, 128(sp)
sw s6, 124(sp)
sw s5, 120(sp)
sw s4, 116(sp)
sw s3, 112(sp)
sw s2, 108(sp)
sw s1, 104(sp)
sw s0, 100(sp)
sw t7, 96(sp)
sw t6, 92(sp)
sw t5, 88(sp)
sw t4, 84(sp)
sw t3, 80(sp)
sw t2, 76(sp)
sw t1, 72(sp)
sw t0, 68(sp)
sw a3, 64(sp)
sw a2, 60(sp)
sw a1, 56(sp)
sw a0, 52(sp)
sw v1, 48(sp)
sw v0, 44(sp)
sw AT, 40(sp)
sw ra, 36(sp)
```

Save all the registers  
on the kernel stack

```
/*
 * Save special registers.
 */
mfhi t0
mflo t1
sw t0, 32(sp)
sw t1, 28(sp)
```

We can now use the other registers (t0, t1) that we have preserved on the stack

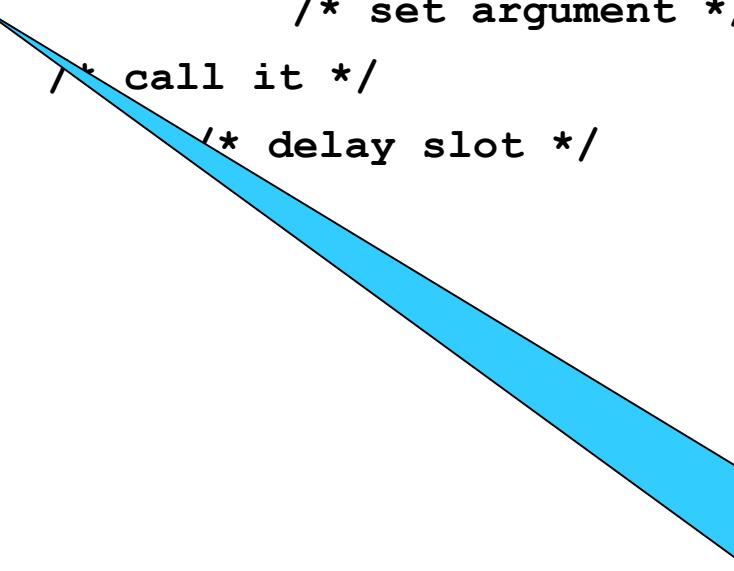
```
/*
 * Save remaining exception context information.
 */

sw k0, 24(sp)                      /* k0 was loaded with cause earlier */
mfc0 t1, c0_status                 /* Copr.0 reg 11 == status */
sw t1, 20(sp)
mfc0 t2, c0_vaddr                  /* Copr.0 reg 8 == faulting vaddr */
sw t2, 16(sp)

/*
 * Pretend to save $0 for gdb's benefit.
 */
sw $0, 12(sp)
```

```
/*
 * Prepare to call mips_trap(struct trapframe *)
 */

addiu a0, sp, 16          /* set argument */
jal mips_trap             /* call it */
nop                      /* delay slot */
```

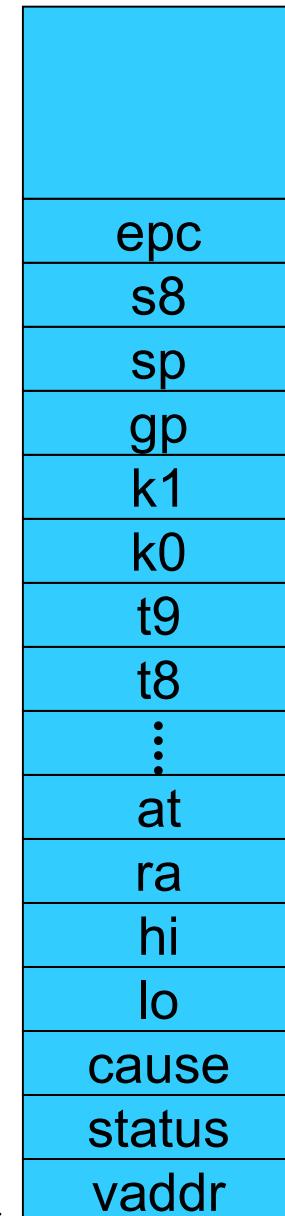


Create a pointer to the base of the saved registers and state in the first argument register

```
struct trapframe {  
    u_int32_t tf_vaddr; /* vaddr register */  
    u_int32_t tf_status; /* status register */  
    u_int32_t tf_cause; /* cause register */  
    u_int32_t tf_lo;  
    u_int32_t tf_hi;  
    u_int32_t tf_ra; /* Saved register 31 */  
    u_int32_t tf_at; /* Saved register 1 (AT) */  
    u_int32_t tf_v0; /* Saved register 2 (v0) */  
    u_int32_t tf_v1; /* etc. */  
    u_int32_t tf_a0;  
    u_int32_t tf_a1;  
    u_int32_t tf_a2;  
    u_int32_t tf_a3;  
    u_int32_t tf_t0;  
    ...  
    u_int32_t tf_t7;  
    u_int32_t tf_s0;  
    ...  
    u_int32_t tf_s7;  
    u_int32_t tf_t8;  
    u_int32_t tf_t9;  
    u_int32_t tf_k0;  
    /*  
     *  
     u_int32_t tf_k1;  
     u_int32_t tf_gp;  
     u_int32_t tf_sp;  
     u_int32_t tf_s8;  
     u_int32_t tf_epc;  
};  
/* coprocessor 0 epc register */
```

By creating a pointer to here of type struct trapframe \*, we can access the user's saved registers as normal variables within 'C'

# Kernel Stack



# Now we arrive in the ‘C’ kernel

```
/*
 * General trap (exception) handling function for mips.
 * This is called by the assembly-language exception handler once
 * the trapframe has been set up.
 */
void
mips_trap(struct trapframe *tf)
{
    u_int32_t code, isutlb, iskern;
    int savespl;

    /* The trap frame is supposed to be 37 registers long. */
    assert(sizeof(struct trapframe)==(37*4));

    /* Save the value of curspl, which belongs to the old context. */
    savespl = curspl;

    /* Right now, interrupts should be off. */
    curspl = SPL_HIGH;
```

# What happens next?

- The kernel deals with whatever caused the exception
  - Syscall
  - Interrupt
  - Page fault
  - It potentially modifies the *trapframe*, etc
    - E.g., Store return code in v0, zero in a3
- ‘mips\_trap’ eventually returns

```

exception_return:

/*      16(sp)          no need to restore tf_vaddr */
lw t0, 20(sp)           /* load status register value into t0 */
nop                   /* load delay slot */
mtc0 t0, c0_status     /* store it back to coprocessor 0 */
/*      24(sp)          no need to restore tf_cause */

/* restore special registers */
lw t1, 28(sp)
lw t0, 32(sp)
mtlo t1
mthi t0

/* load the general registers */
lw ra, 36(sp)

lw AT, 40(sp)
lw v0, 44(sp)
lw v1, 48(sp)
lw a0, 52(sp)
lw a1, 56(sp)
lw a2, 60(sp)
lw a3, 64(sp)

```

```
lw t0, 68(sp)
lw t1, 72(sp)
lw t2, 76(sp)
lw t3, 80(sp)
lw t4, 84(sp)
lw t5, 88(sp)
lw t6, 92(sp)
lw t7, 96(sp)
lw s0, 100(sp)
lw s1, 104(sp)
lw s2, 108(sp)
lw s3, 112(sp)
lw s4, 116(sp)
lw s5, 120(sp)
lw s6, 124(sp)
lw s7, 128(sp)
lw t8, 132(sp)
lw t9, 136(sp)

/*      140(sp)          "saved" k0 was dummy garbage anyway */
/*      144(sp)          "saved" k1 was dummy garbage anyway */
```

```

lw gp, 148(sp)      /* restore gp */
/*      152(sp)          stack pointer - below */
lw s8, 156(sp)      /* restore s8 */
lw k0, 160(sp)      /* fetch exception return PC into k0 */

lw sp, 152(sp)      /* fetch saved sp (must be last) */

/* done */
jr k0               /* jump back */
rfe                /* in delay slot */
.end common_exception

```

Note again that only  
k0, k1 have been  
trashed

# Computer Hardware Review (Memory Hierarchy)

Chapter 1.4

# Learning Outcomes

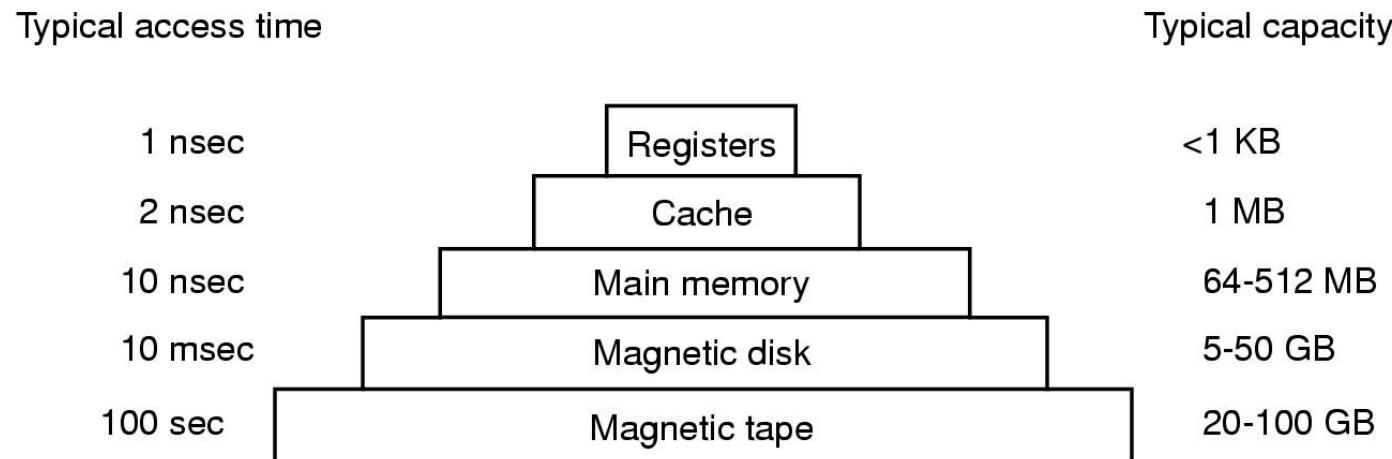
- Understand the concepts of memory hierarchy and caching, and how they affect performance.

# Operating Systems

- Exploit the hardware available
- Provide a set of high-level services that represent or are implemented by the hardware.
- Manages the hardware reliably and efficiently
- *Understanding operating systems requires a basic understanding of the underlying hardware*

# Memory Hierarchy

- Going down the hierarchy
  - Decreasing cost per bit
  - Increasing capacity
  - Increasing access time
- Decreasing frequency of access to the memory by the processor
  - Hopefully
  - Principle of locality!!!!

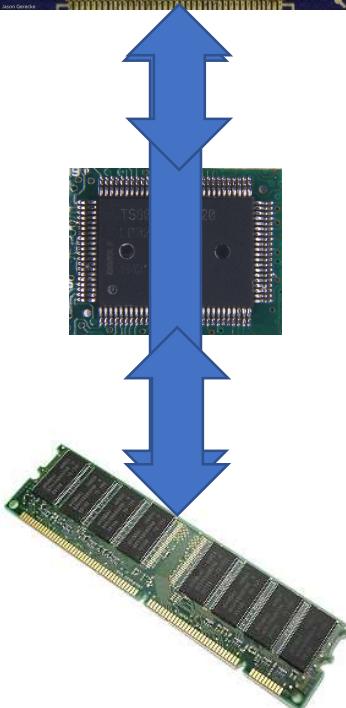
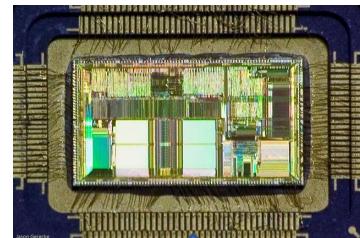


# Caching as a general technique

- Given two-levels of data storage: small and fast, versus large and slow,
- Can speed access to slower storage by using intermediate-speed storage as a cache.

# A hardware approach to improving system performance?

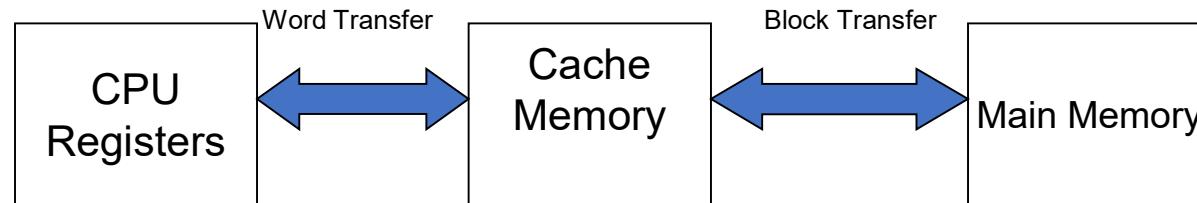
CPU Registers  
Fast



Cache Memory (SRAM)  
Fast

Main Memory (DRAM)  
Slow

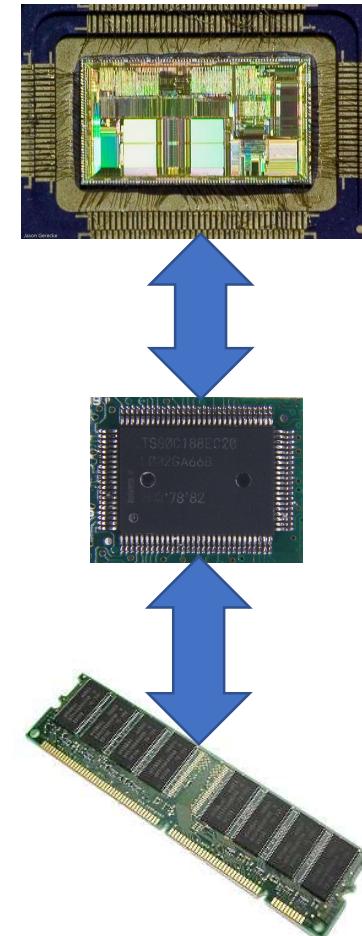
# CPU Cache



- CPU cache is fast memory placed between the CPU and main memory
  - 1 to a few cycles access time compared to RAM access time of tens – hundreds of cycles
- Holds recently used data or instructions to save memory accesses.
- Matches slow RAM access time to CPU speed if high hit rate
- Is hardware maintained and (mostly) transparent to software
- Sizes range from few kB to tens of MB.
- Usually a hierarchy of caches (2–5 levels), on- and off-chip.

# Performance

- What is the effective access time of memory subsystem?
- Answer: It depends on the hit rate in the first level.



# Effective Access Time

$$T_{eff} = H \times T_1 + (1 - H) \times T_2$$

$T_1$  = access time of memory 1

$T_2$  = access time of memory 2

$H$  = hit rate in memory 1

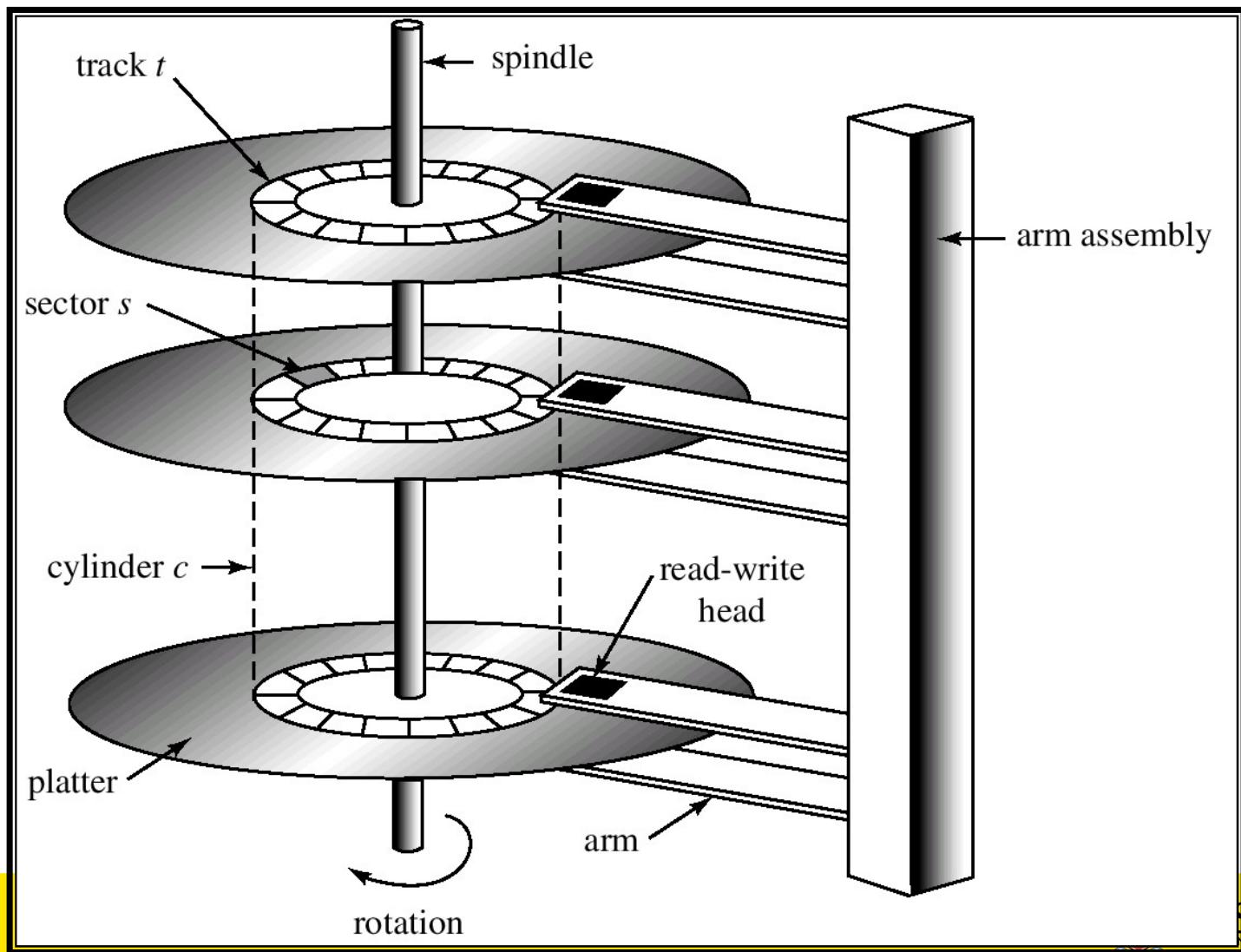
$T_{eff}$  = effective access time of system

# Example

- Cache memory access time 1ns
- Main memory access time 10ns
- Hit rate of 95%

$$\begin{aligned}T_{eff} &= 0.95 \times 10^{-9} + \\&(1 - 0.95) \times (10^{-9} + 10 \times 10^{-9}) \\&= 1.5 \times 10^{-9}\end{aligned}$$

# Moving-Head Disk Mechanism

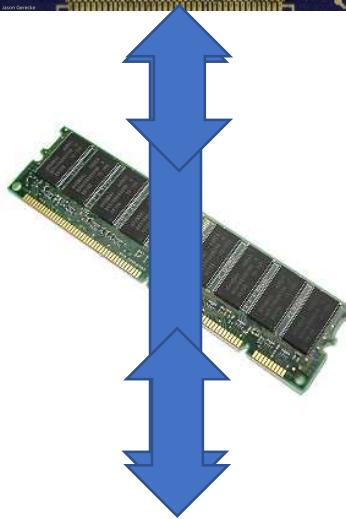
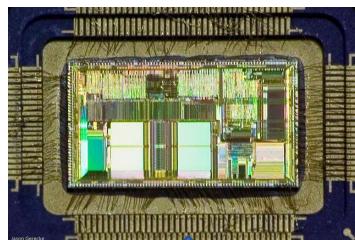


# Example Disk Access Times

- Disk can read/write data relatively fast
  - 15,000 rpm drive - 80 MB/sec
  - 1 KB block is read in 12 microseconds
- Access time dominated by time to locate the head over data
  - Rotational latency
    - Half one rotation is 2 milliseconds
  - Seek time
    - Full inside to outside is 8 milliseconds
    - Track to track .5 milliseconds
- 2 milliseconds is 164KB in “lost bandwidth”

# A OS approach to improving system performance?

CPU Registers  
Fast



Main Memory (DRAM)  
Fast

Hard disk  
Slow...

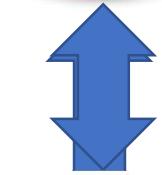


# A Strategy: Avoid Waiting for Disk Access

- Keep a subset of the disk's data in main memory  
⇒ OS uses main memory as a *cache* of disk contents

# Application approach to improving system performance

Web browser  
Fast



Hard disk  
Fast



Internet  
Slow...

# A Strategy: Avoid Waiting for Internet Access

- Keep a subset of the Internet's data on disk
- ⇒ Application uses disk as a *cache* of the Internet

# File Management

Tanenbaum, Chapter 4

COMP3231  
Operating Systems

Kevin Elphinstone

# Outline

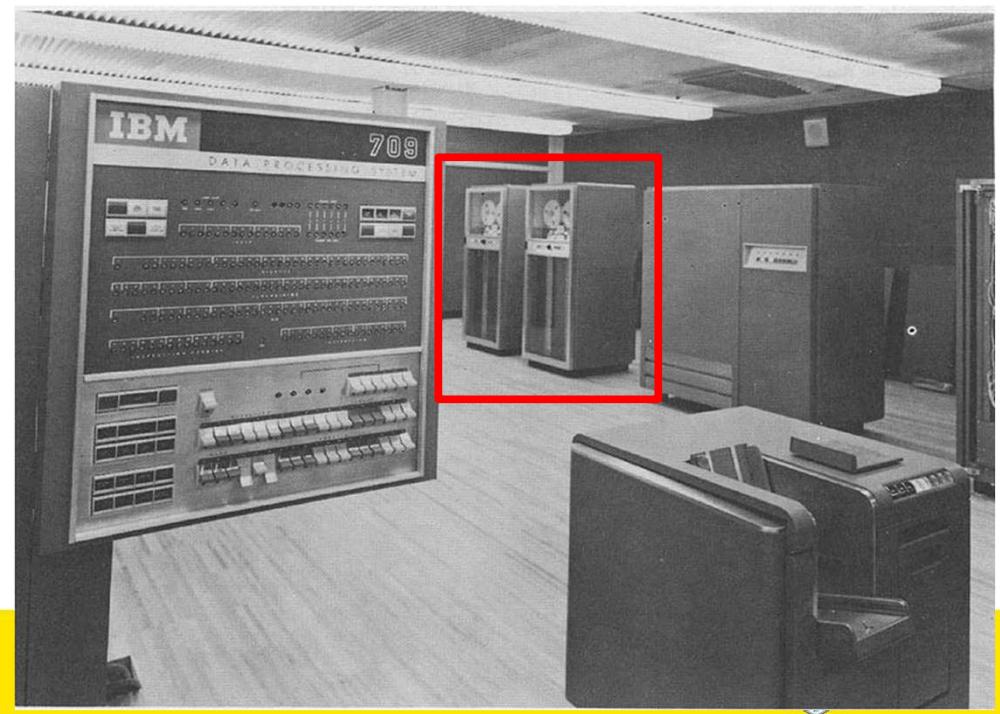
- Files and directories from the programmer (and user) perspective
- Files and directories internals – the operating system perspective

# A brief history of file systems

## Early batch processing systems

- No OS
- I/O from/to punch cards
- Tapes and drums for external storage, but no FS
- Rudimentary library support for reading/writing tapes and drums

IBM 709 [1958]



# A brief history of file systems

- The first file systems were single-level (everything in one directory)
- Files were stored in contiguous chunks
  - Maximal file size must be known in advance
- Now you can edit a program and save it in a named file on the tape!



PDP-8 with DECTape [1965]

# A brief history of file systems

- Time-sharing OSs
  - Required full-fledged file systems
- MULTICS
  - Multilevel directory structure (keep files that belong to different users separately)
  - Access control lists
  - Symbolic links

Honeywell 6180 running  
MULTICS [1976]



# A brief history of file systems

- **UNIX**
  - Based on ideas from **MULTICS**
  - Simpler access control model
  - Everything is a file!

PDP-7



# Overview of the FS abstraction

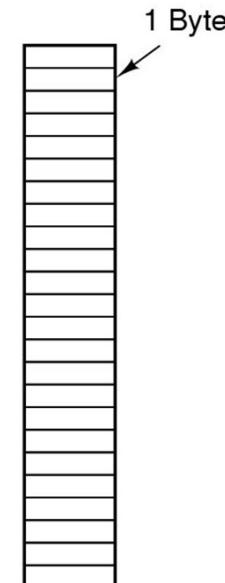
User's view	Under the hood
Uniform namespace	Heterogeneous collection of storage devices
Hierarchical structure	Flat address space (block numbers)
Arbitrarily-sized files	Fixed-size blocks
Symbolic file names	Numeric block addresses
Contiguous address space inside a file	Fragmentation
Access control	No access control
Tools for <ul style="list-style-type: none"><li>• Formatting</li><li>• Defragmentation</li><li>• Backup</li><li>• Consistency checking</li></ul>	

# File Names

- File system must provide a convenient naming scheme
  - Textual Names
  - May have restrictions
    - Only certain characters
      - E.g. no '/' characters
    - Limited length
    - Only certain format
      - E.g DOS, 8 + 3
  - Case (in)sensitive
  - Names may obey conventions (.c files for C files)
    - Interpreted by tools (e.g. UNIX)
    - Interpreted by operating system (e.g. Windows "con:")

# File Structure

- Sequence of Bytes
  - OS considers a file to be unstructured
  - Applications can impose their own structure
  - Used by UNIX, Windows, most modern OSes



(a)

# File Types

- Regular files
- Directories
- Device Files
  - May be divided into
    - Character Devices – stream of bytes
    - Block Devices
- Some systems distinguish between regular file types
  - ASCII text files, binary files

# File Access Types (Patterns)

- Sequential access

- read all bytes/records from the beginning
  - cannot jump around, could rewind or back up
  - convenient when medium was magnetic tape

- Random access

- bytes/records read in any order
  - essential for data base systems
  - read can be ...
    - move file pointer (seek), then read or
      - lseek(location,...);read(...)
    - each read specifies the file pointer
      - read(location,...)

# File Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

# Typical File Operations

- Create
- Delete
- Open
- Close
- Read
- Write
- Append
- Seek
- Get attributes
- Set Attributes
- Rename

# An Example Program Using File System Calls (1/2)

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                      /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);             /* ANSI prototype */

#define BUF_SIZE 4096                         /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                        /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);                    /* syntax error if argc is not 3 */
```

# An Example Program Using File System Calls (2/2)

```
/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY); /* open the source file */
if (in_fd < 0) exit(2); /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
if (out_fd < 0) exit(3); /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
    rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break; /* if end of file or error, exit loop */
    wt_count = write(out_fd, buffer, rd_count); /* write data */
    if (wt_count <= 0) exit(4); /* wt_count <= 0 is an error */
}

/* Close the files */
close(in_fd);
close(out_fd);
if (rd_count == 0) /* no error on last read */
    exit(0);
else
    exit(5); /* error on last read */
}
```

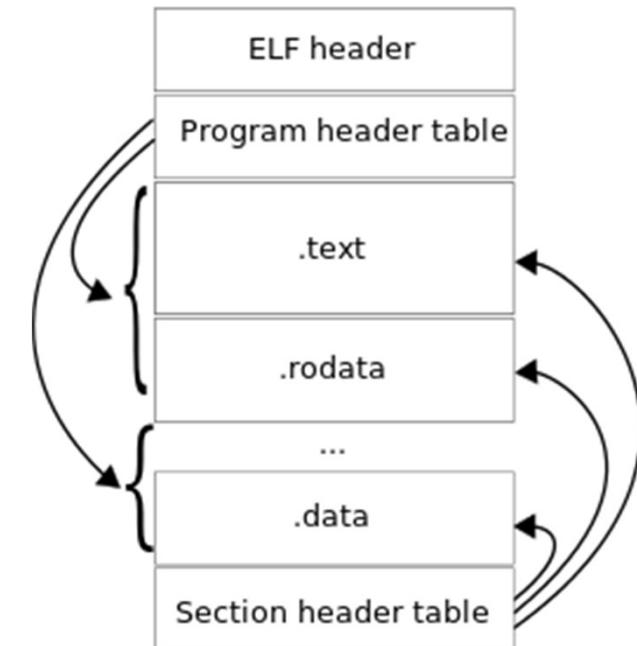
# File Organisation and Access

## Programmer's Perspective

- Given an operating system supporting unstructured files that are a *stream-of-bytes*,

how can one organise the contents of the files?

E.g. Executable Linkable Format (ELF)



# File Organisation and Access

## Programmer's Perspective

- Some possible access patterns:
  - Read the whole file
  - Read individual records from a file
    - record = sequence of bytes containing the record
  - Read records preceding or following the current one
  - Retrieve a set of records
  - Write a whole file sequentially
  - Insert/delete/update records in a file

Programmers are free to structure the file to suit the application.

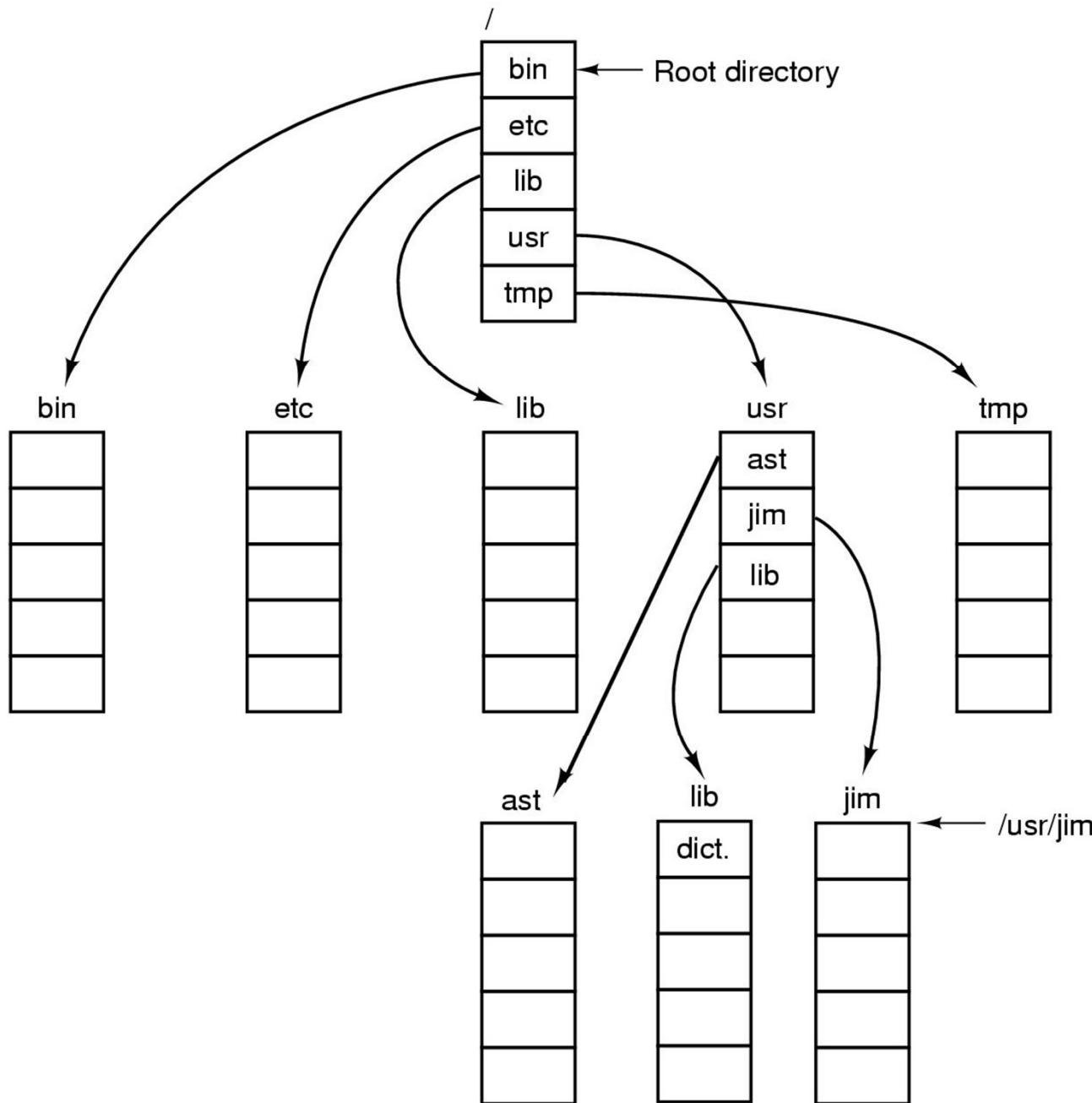
# Criteria for File Organization

Things to consider when designing file layout

- Rapid access
  - Needed when accessing a single record
  - Not needed for batch mode
    - read from start to finish
- Ease of update
  - File on CD-ROM will not be updated, so this is not a concern
- Economy of storage
  - Should be minimum redundancy in the data
  - Redundancy can be used to speed access such as an index

# File Directories

- Provide mapping between file names and the files themselves
- Contain information about files
  - Attributes
  - Location
  - Ownership
- Directory itself is a file owned by the operating system



# Hierarchical (Tree-Structured) Directory

- Files can be located by following a path from the root, or master, directory down various branches
  - This is the *absolute* pathname for the file
- Can have several files with the same file name as long as they have unique path names

# *Current Working Directory*

- Always specifying the absolute pathname for a file is tedious!
- Introduce the idea of a *working directory*
  - Files are referenced relative to the working directory
- Example: cwd = /home/kevine  
.profile = /home/kevine/.profile

# Relative and Absolute Pathnames

- Absolute pathname
  - A path specified from the root of the file system to the file
- A *Relative pathname*
  - A pathname specified from the cwd
- Note: ‘.’ (dot) and ‘..’ (dotdot) refer to current and parent directory

Example: cwd = /home/kevine

**.../.../etc/passwd**

**/etc/passwd**

**.../kevine/.../.../etc/passwd**

Are all the same file

# Typical Directory Operations

- Create
- Delete
- Opendir
- Closedir
- Readdir
- Rename
- Link
- Unlink

# Nice properties of UNIX naming

- Simple, regular format
  - Names referring to different servers, objects, etc., have the same syntax.
    - Regular tools can be used where specialised tools would be otherwise be needed.
- Location independent
  - Objects can be distributed or migrated, and continue with the same names.

Where is /home/kevine/.profile?

You only need to know the name!

# An example of a bad naming convention

- From, Rob Pike and Peter Weinberger,  
“The Hideous Name”, Bell Labs TR

UCBVAX::SYS\$DISK:[ROB.BIN]CAT\_V.EXE;13

# File Sharing

- In multiuser system, allow files to be shared among users
- Two issues
  - Access rights
  - Management of simultaneous access

# Access Rights

- **None**
  - User may not know of the existence of the file
  - User is not allowed to read the directory that includes the file
- **Knowledge**
  - User can only determine that the file exists and who its owner is

# Access Rights

- Execution
  - The user can load and execute a program but cannot copy it
- Reading
  - The user can read the file for any purpose, including copying and execution
- Appending
  - The user can add data to the file but cannot modify or delete any of the file's contents

# Access Rights

- Updating
  - The user can modify, delete, and add to the file's data. This includes creating the file, rewriting it, and removing all or part of the data
- Changing protection
  - User can change access rights granted to other users
- Deletion
  - User can delete the file

# Access Rights

- Owners
  - Has all rights previously listed
  - May grant rights to others using the following classes of users
    - Specific user
    - User groups
    - All for public files

# Simultaneous Access

- Most OSes provide mechanisms for users to manage concurrent access to files
  - Example: flock(), lockf(), system calls
- Typically
  - User may lock entire file when it is to be updated
  - User may lock the individual records (i.e. ranges) during the update
- Mutual exclusion and deadlock are issues for shared access

# File system internals

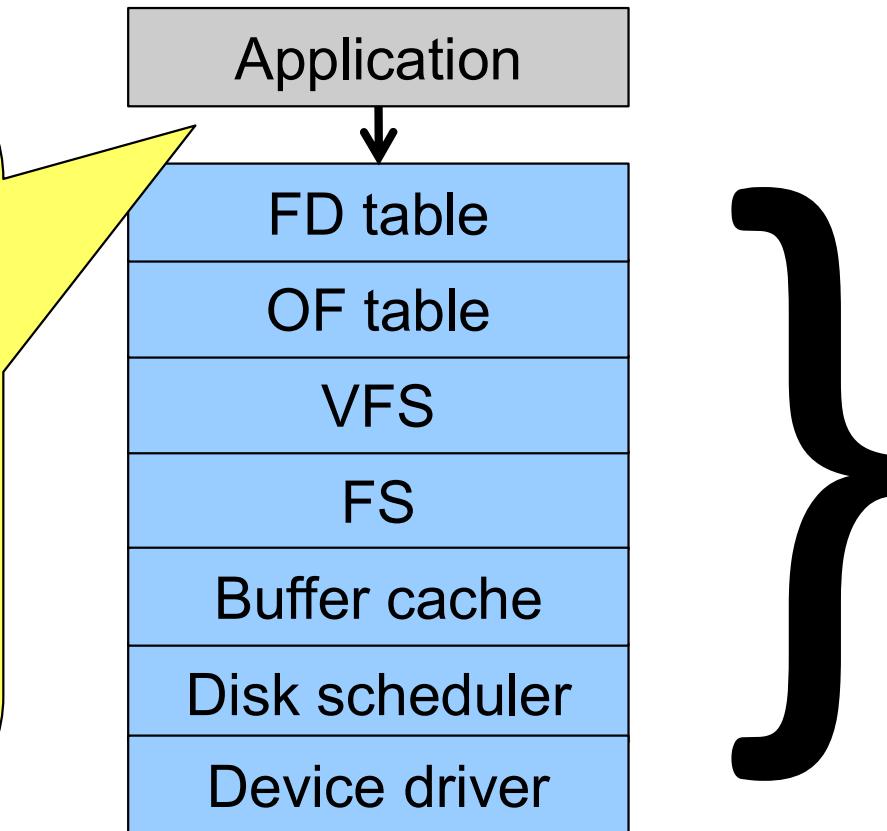
Tanenbaum, Chapter 4

COMP3231  
Operating Systems

# UNIX storage stack

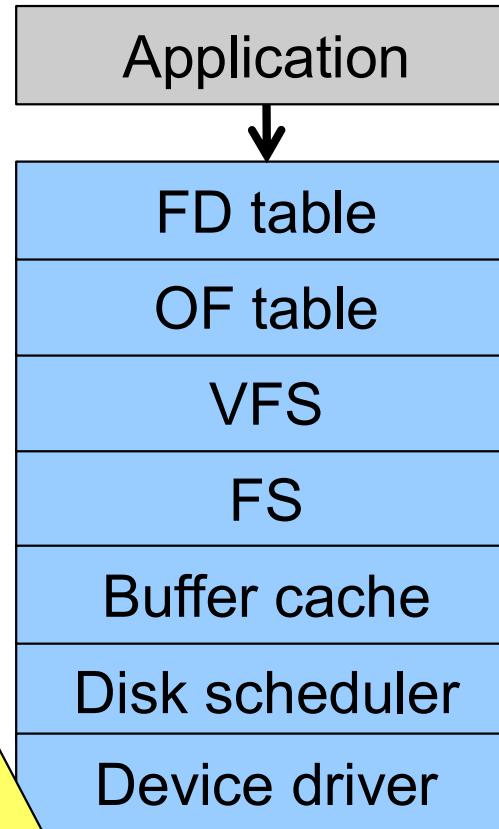
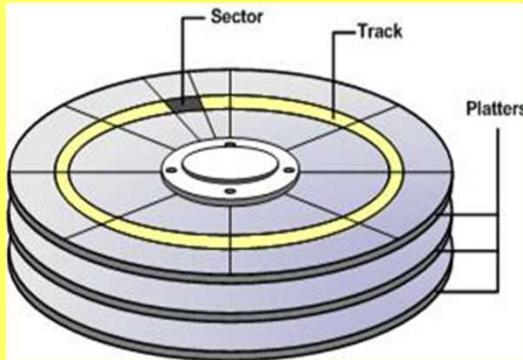
Syscall interface:

- creat
- open
- read
- write
- ...



# UNIX storage stack

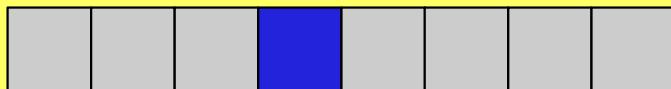
Hard disk platters:  
tracks  
sectors



# UNIX storage stack

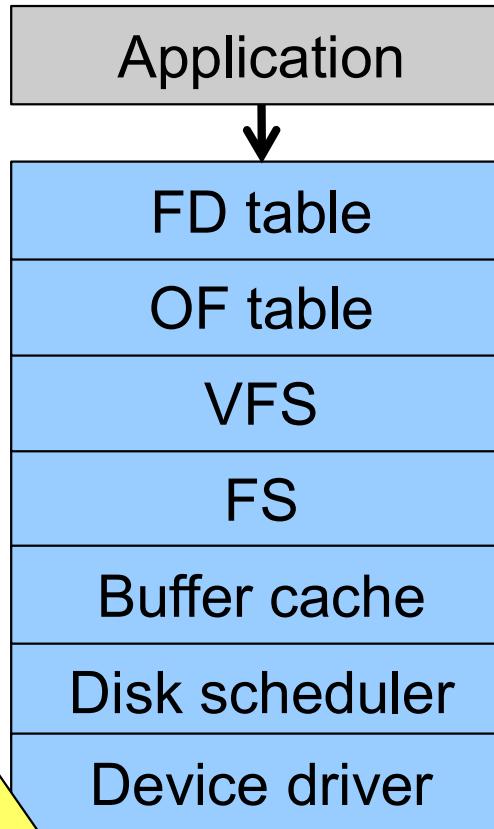
Disk controller:

Hides disk geometry,  
bad sectors  
Exposes linear  
sequence of blocks



0

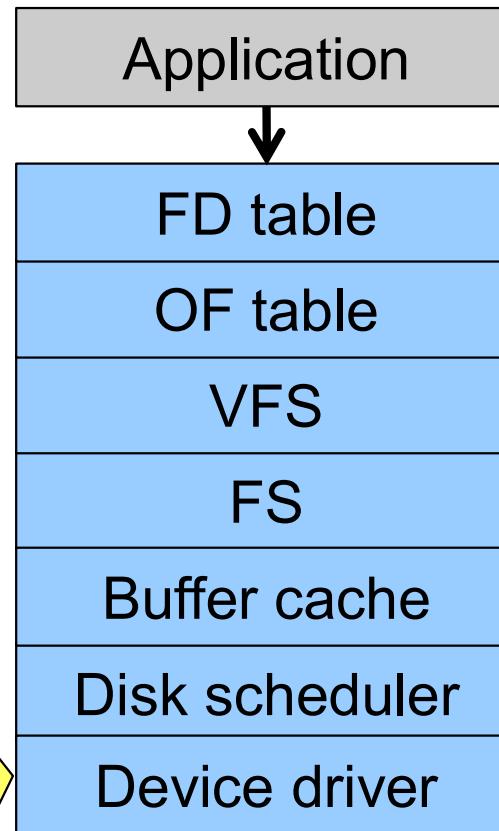
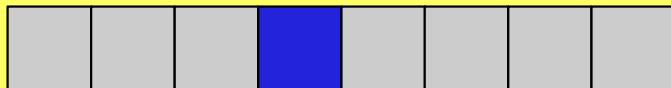
N



# UNIX storage stack

Device driver:

Hides device-specific protocol  
Exposes block-device Interface (linear sequence of blocks)

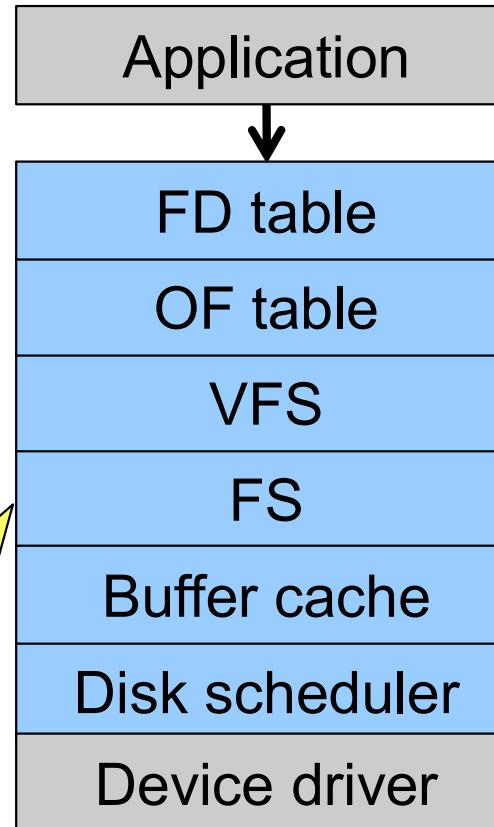


# UNIX storage stack

File system:

Hides physical location  
of data on the disk

Exposes: directory  
hierarchy, symbolic file  
names, random-access  
files, protection

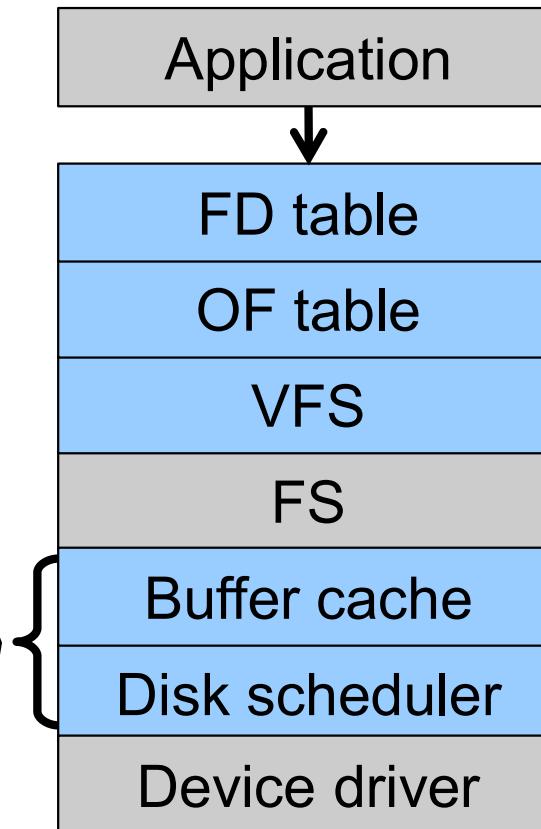


# UNIX storage stack

Optimisations:

Keep recently accessed disk blocks in memory

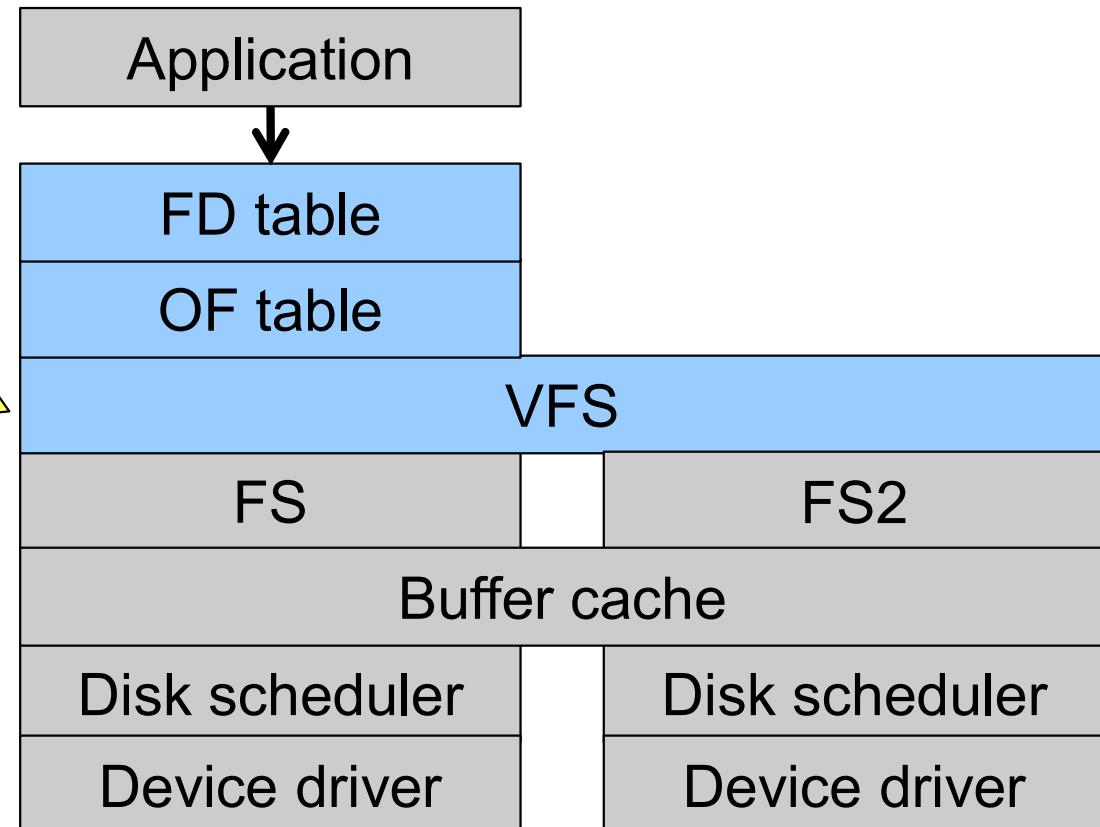
Schedule disk accesses from multiple processes for performance and fairness



# UNIX storage stack

Virtual FS:

Unified interface to  
multiple FSs

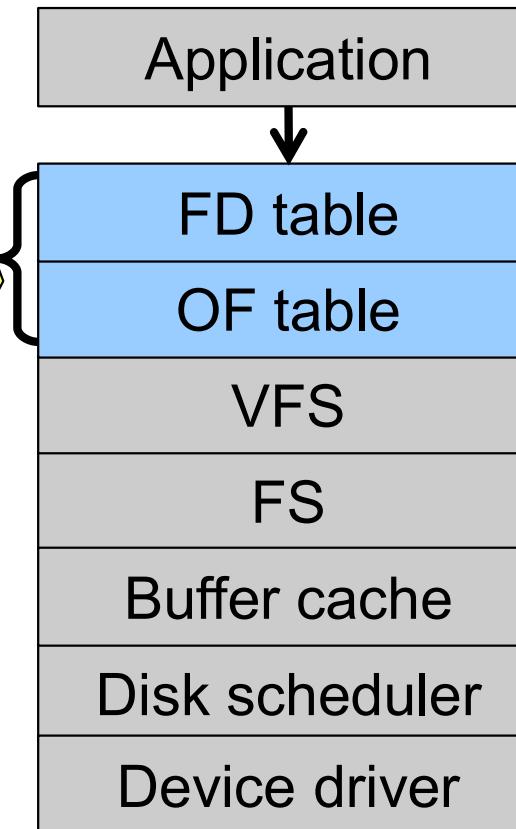


# UNIX storage stack

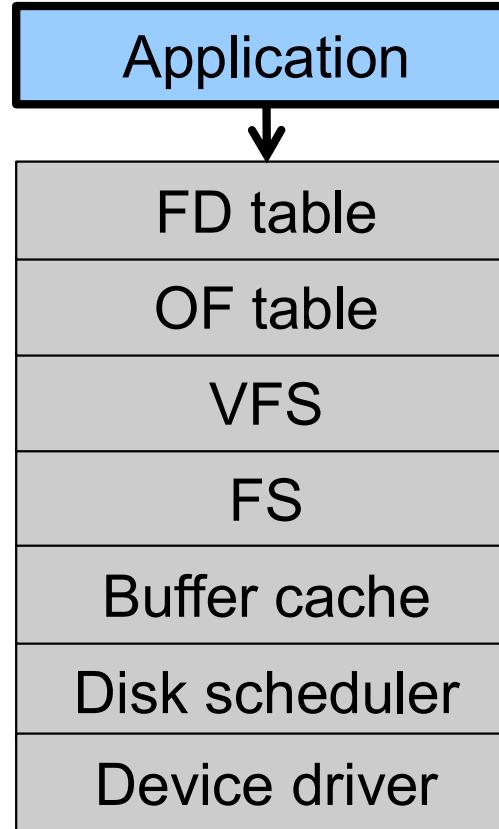
File descriptor and  
Open file tables:

Keep track of files  
opened by user-level  
processes

Matches syscall interface  
to VFS Interface



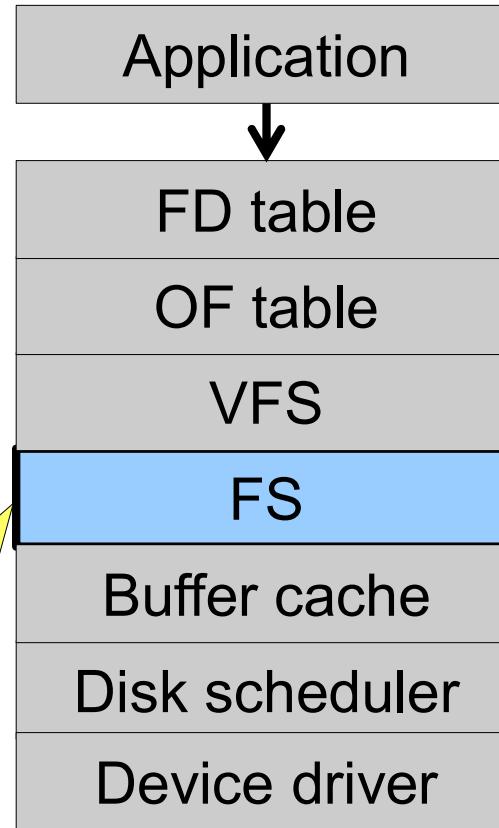
# UNIX storage stack



# Architecture of the OS storage stack

## File system:

- Hides physical location of data on the disk
- Exposes: directory hierarchy, symbolic file names, random-access files, protection



# Some popular file systems

- FAT16
- FAT32
- NTFS
- Ext2
- Ext3
- Ext4
- ReiserFS
- XFS
- ISO9660
- HFS+
- UFS2
- ZFS
- JFS
- OCFS
- Btrfs
- JFFS2
- ExFAT
- UBIFS

Question: why are there so many?

# Why are there so many?

- Different physical nature of storage devices
  - Ext3 is optimised for magnetic disks
  - JFFS2 is optimised for flash memory devices
  - ISO9660 is optimised for CDROM
- Different storage capacities
  - FAT16 does not support drives >2GB
  - FAT32 becomes inefficient on drives >32GB
  - ZFS, Btrfs is designed to scale to multi-TB disk arrays
- Different CPU and memory requirements
  - FAT16 is not suitable for modern PCs but is a good fit for many embedded devices
- Proprietary standards
  - NTFS may be a nice FS, but its specification is closed

# Outline

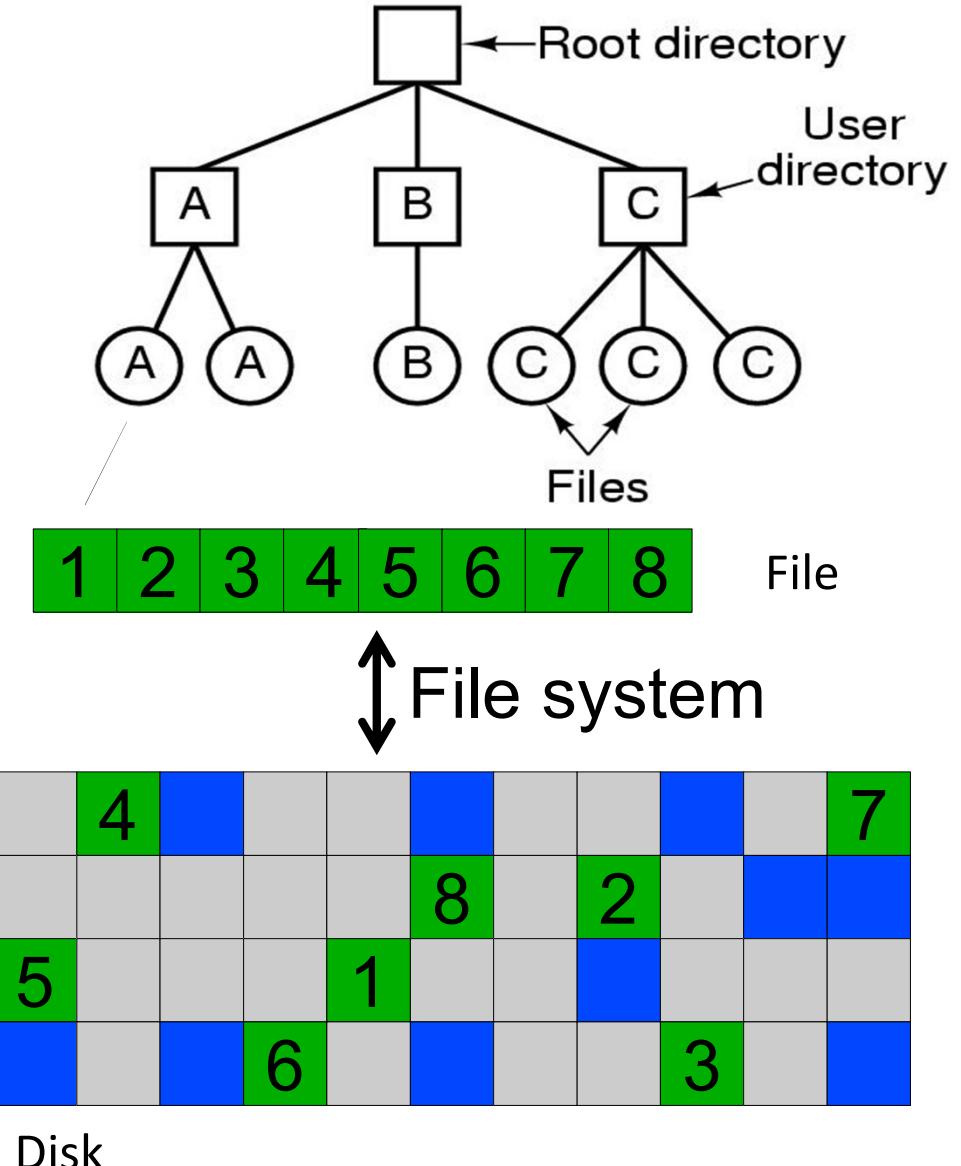
- File allocation methods
  - How files are stored in disk blocks, and what book keeping is required.
- Layout on disk
- Managing free space
- Directories
- Block size trade off

# Assumptions

- In this lecture we focus on file systems for magnetic disks
  - Seek time
    - ~15ms worst case
  - Rotational delay
    - 8ms worst case for 7200rpm drive
  - For comparison, disk-to-buffer transfer speed of a modern drive is  $\sim 10\mu\text{s}$  per 4K block.
- Conclusion: keep blocks that are likely to be accessed together close to each other

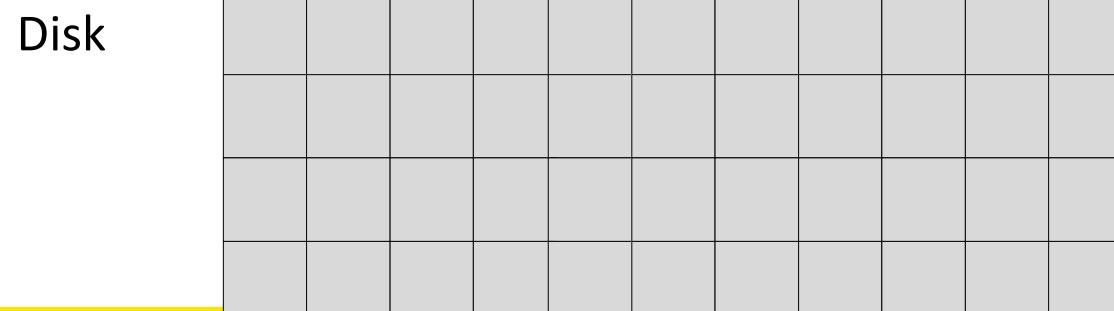
# Implementing a file system

- The FS must map symbolic file names into a collection of block addresses
- The FS must keep track of
  - which blocks belong to which files.
  - in what order the blocks form the file
  - which blocks are free for allocation
- Given a logical region of a file, the FS must track the corresponding block(s) on disk.
  - Stored in file system metadata



# File Allocation Methods

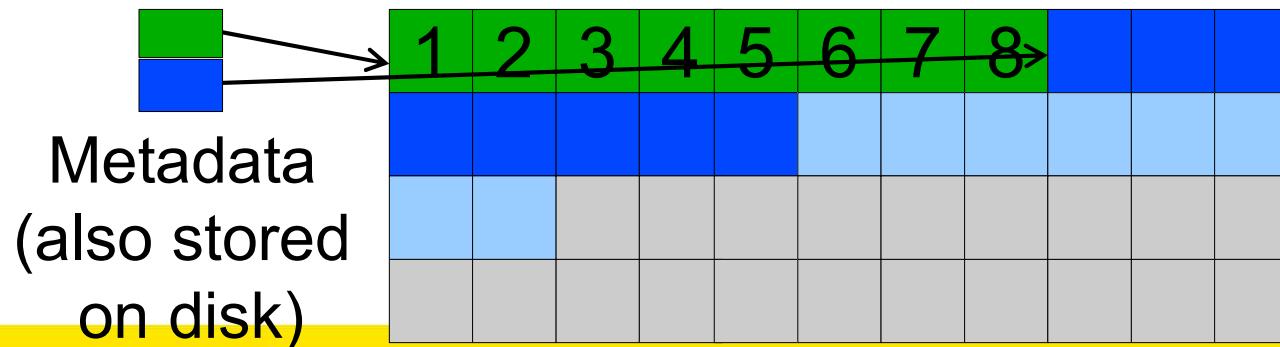
- A file is divided into “blocks”
  - the unit of transfer to storage
- Given the logical blocks of a file, what method is used to choose where to put the blocks on disk?



# Contiguous Allocation

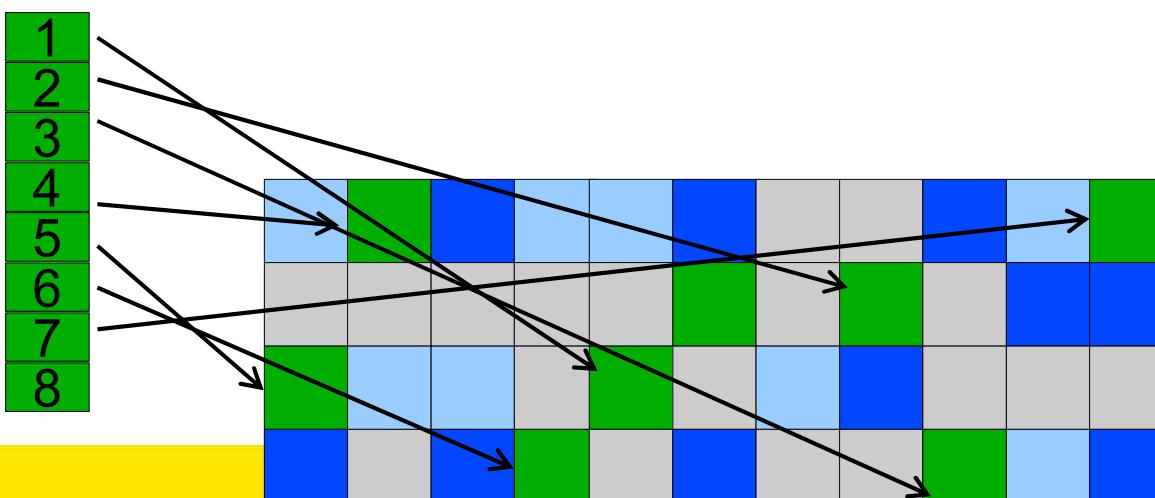
- ✓ Easy bookkeeping (need to keep track of the starting block and length of the file)
- ✓ Increases performance for sequential operations
- ✗ Need the maximum size for the file at the time of creation
- ✗ As files are deleted, free space becomes divided into many small chunks (external fragmentation)

Example: ISO 9660 (CDROM FS)



# Dynamic Allocation Strategies

- Disk space allocated in portions as needed
- Allocation occurs in fixed-size blocks
  - ✓ No external fragmentation
  - ✓ Does not require pre-allocating disk space
  - ✗ Partially filled blocks (internal fragmentation)
  - ✗ File blocks are scattered across the disk
  - ✗ Complex metadata management (maintain the collection of blocks for each file)

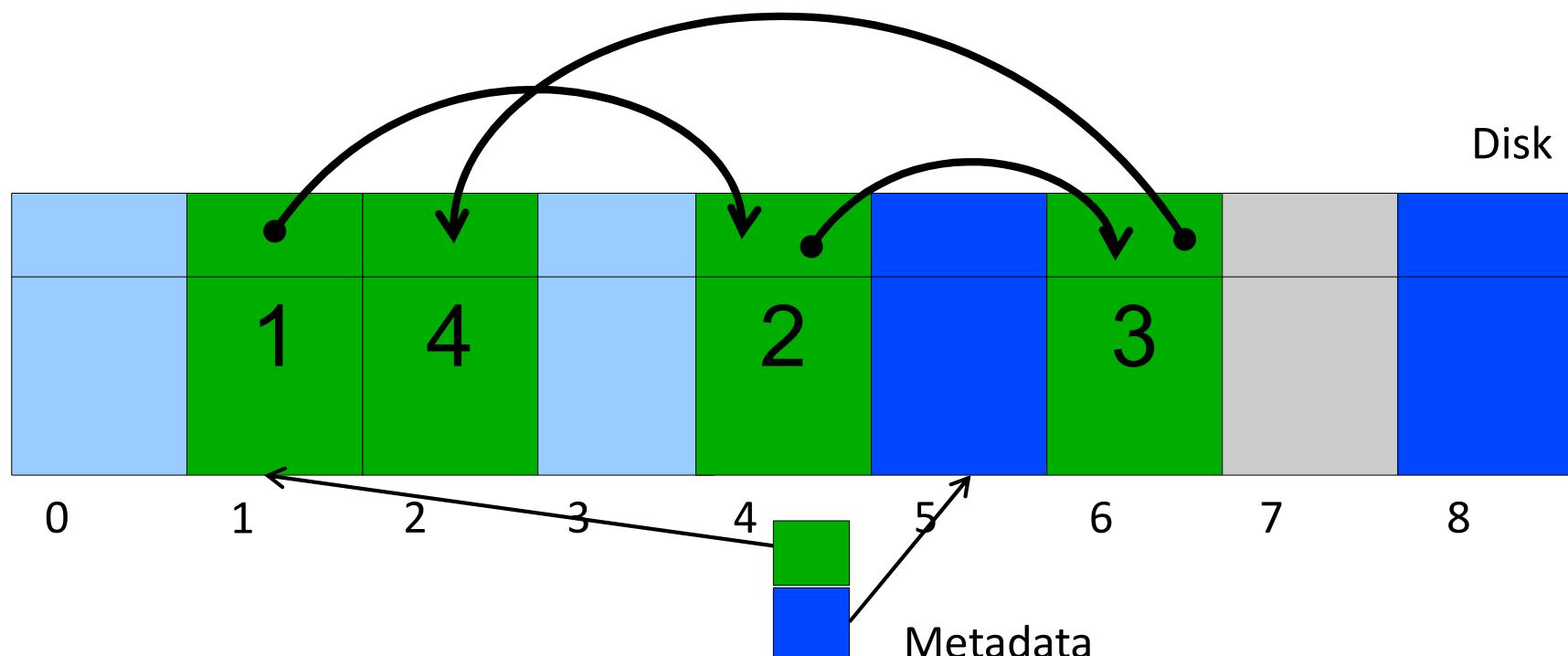


# External and internal fragmentation

- External fragmentation
  - The space wasted external to the allocated memory regions
  - Memory space exists to satisfy a request but it is unusable as it is not contiguous
- Internal fragmentation
  - The space wasted internal to the allocated memory regions
  - Allocated memory may be slightly larger than requested memory; this size difference is wasted memory internal to a partition

# Dynamic allocation: Linked list allocation

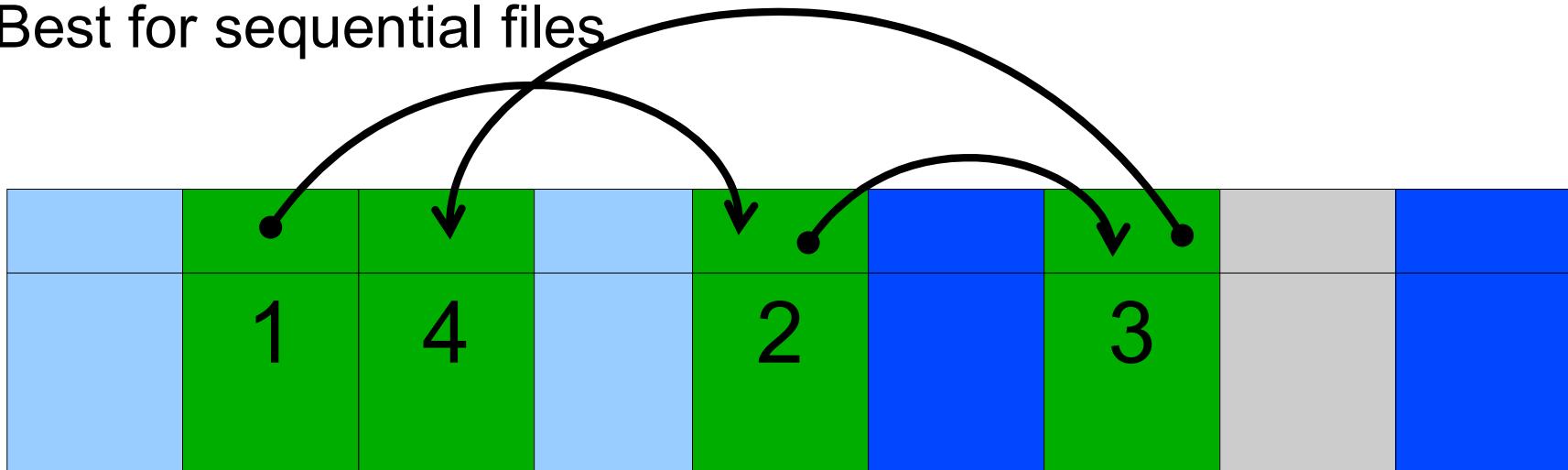
- Each block contains a pointer to the next block in the chain. Free blocks are also linked in a chain.
  - ✓ Only single metadata entry per file
  - ✓ Best for sequentially accessed files



Question: What are the downsides?

# Linked list allocation

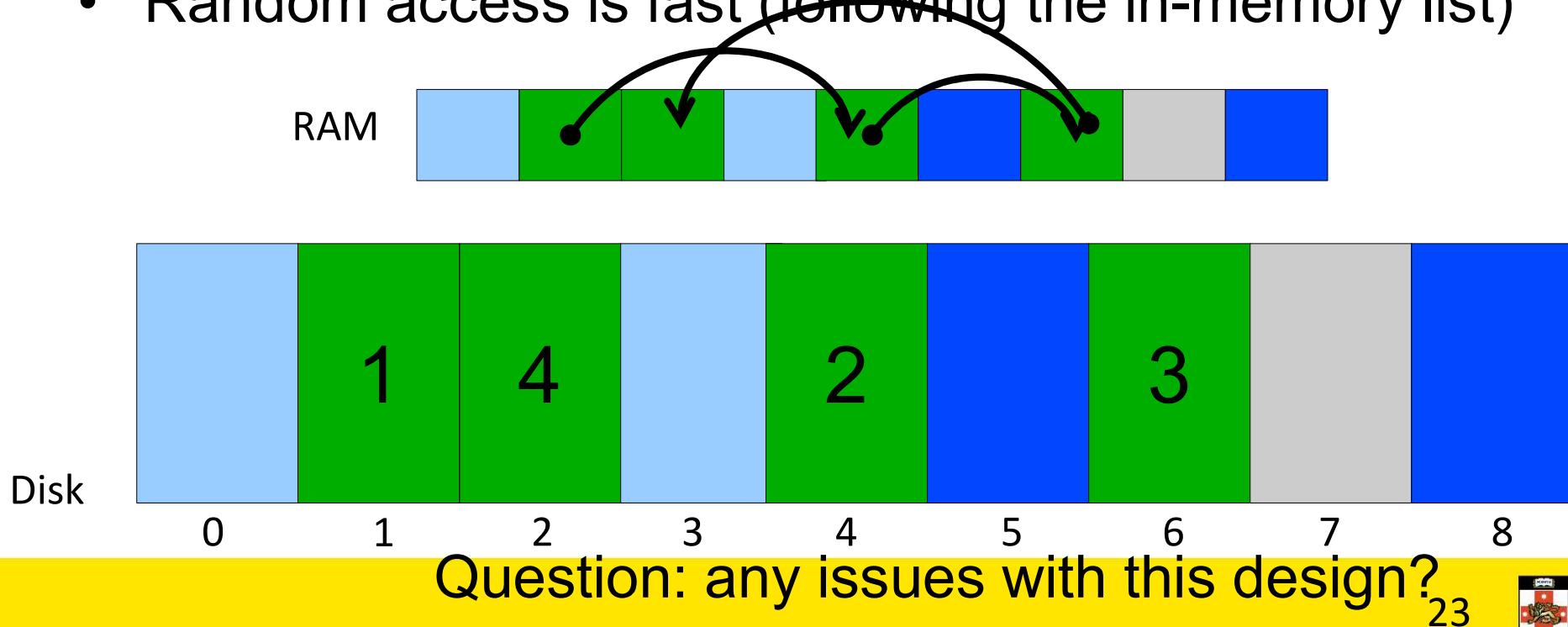
- Each block contains a pointer to the next block in the chain. Free blocks are also linked in a chain.
  - ✓ Only single metadata entry per file
  - ✓ Best for sequential files



- ✗ Poor for random access
- ✗ Blocks end up scattered across the disk due to free list eventually being randomised

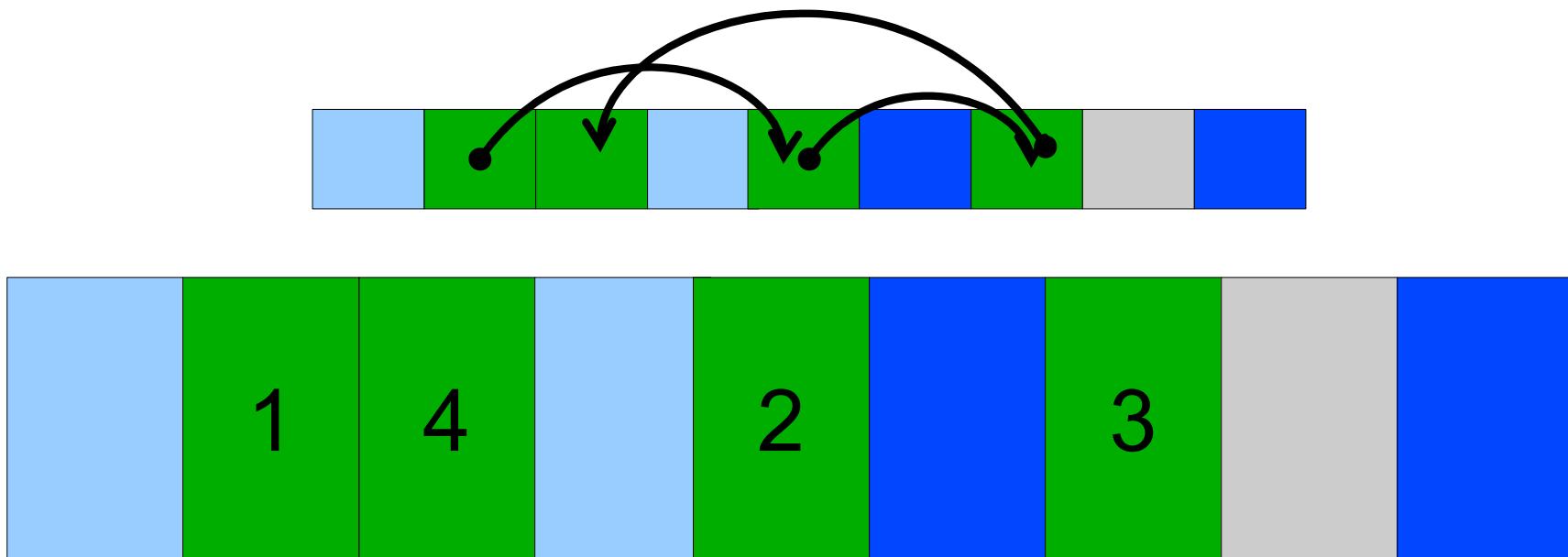
# Dynamic Allocation: File Allocation Table (FAT)

- Keep a map of the entire FS in a separate table
  - A table entry contains the number of the next block of the file
  - The last block in a file and empty blocks are marked using reserved values
- The table is stored on the disk and is replicated in memory
- Random access is fast (following the in-memory list)



# File allocation table

- Issues
  - Requires a lot of memory for large disks
    - $200\text{GB} = 200 * 10^6 * 1\text{K-blocks} ==>$   
 $200 * 10^6 \text{ FAT entries} = 800\text{MB}$
  - Free block lookup is slow
    - searches for a free entry in table



# File allocation table disk layout

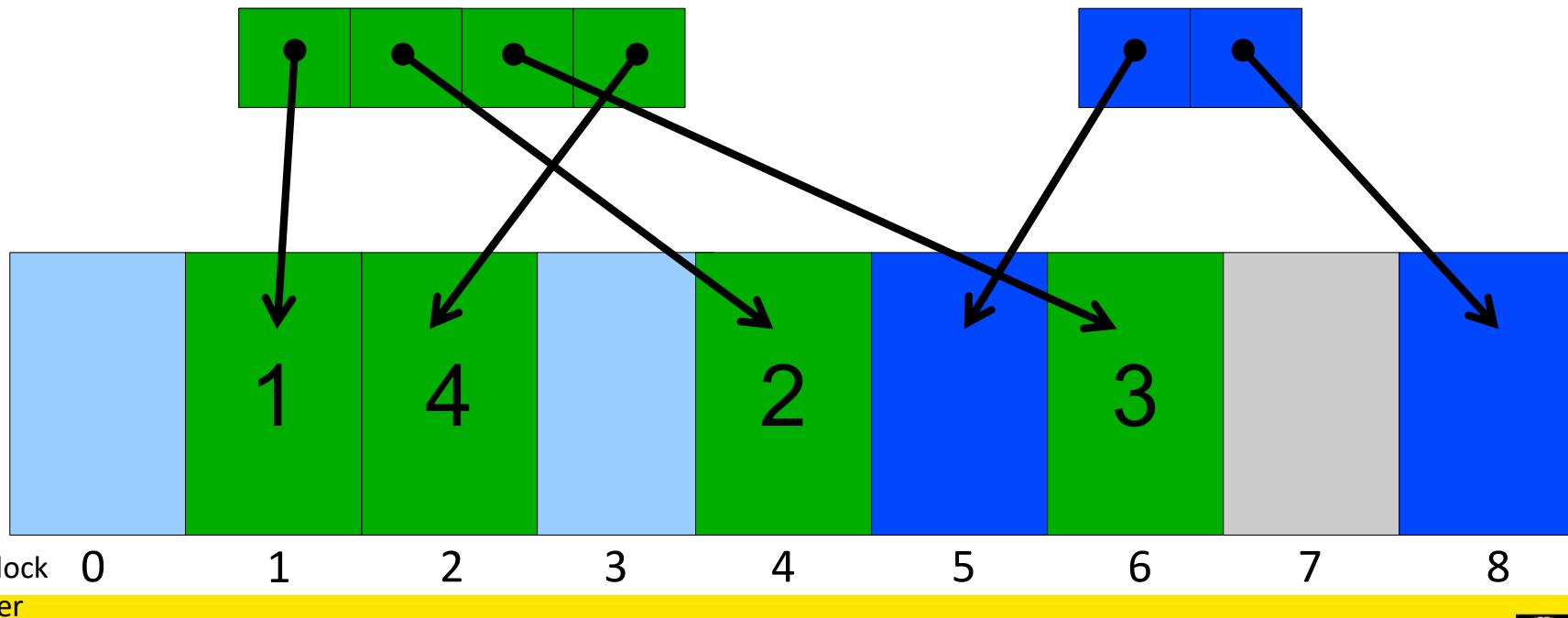
- Examples
  - FAT12, FAT16, FAT32



Two copies of FAT for  
redundancy

# Dynamical Allocation: inode-based FS structure

- Idea: separate table (index-node or i-node) for each file.
  - Only keep table for open files in memory
  - Fast random access
- The most popular FS structure today



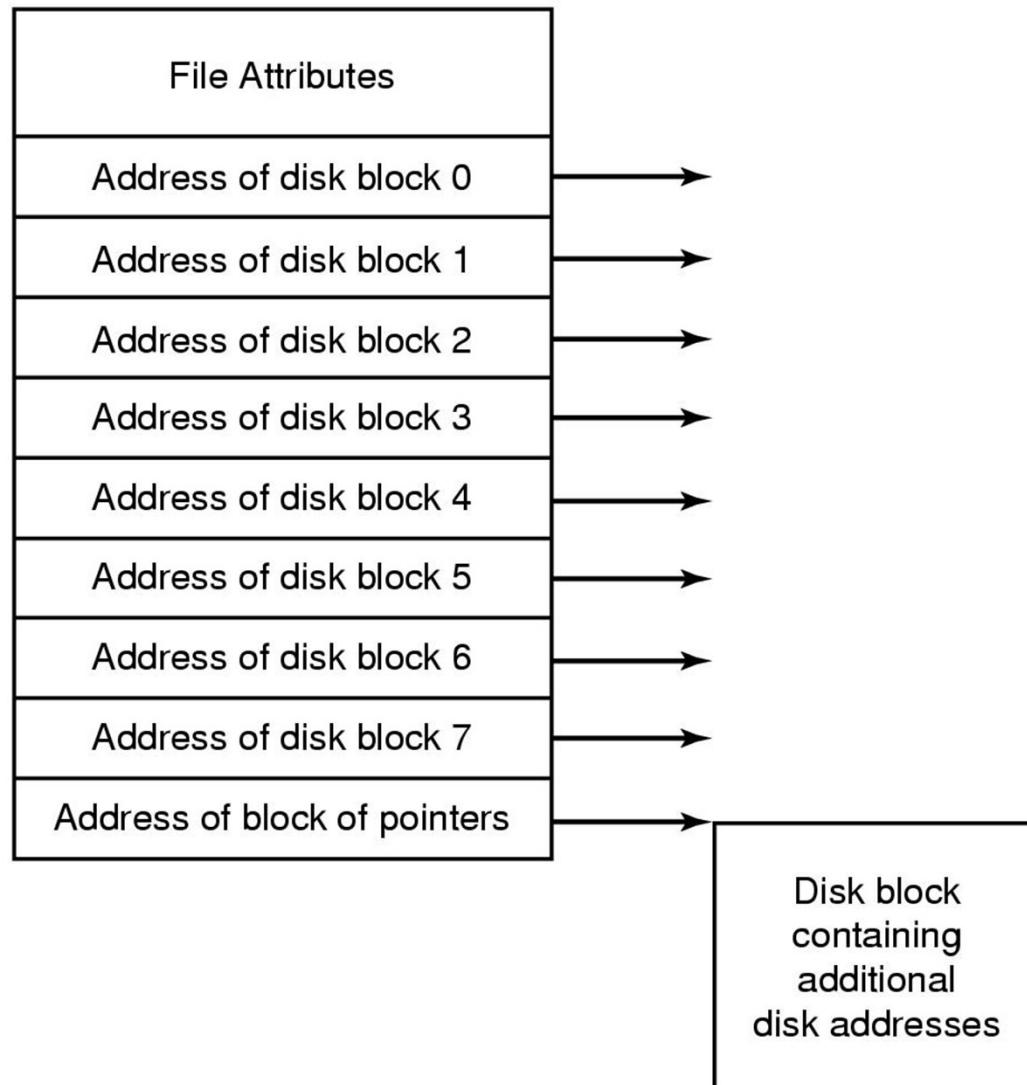
# i-node implementation issues

- i-nodes occupy one or several disk areas



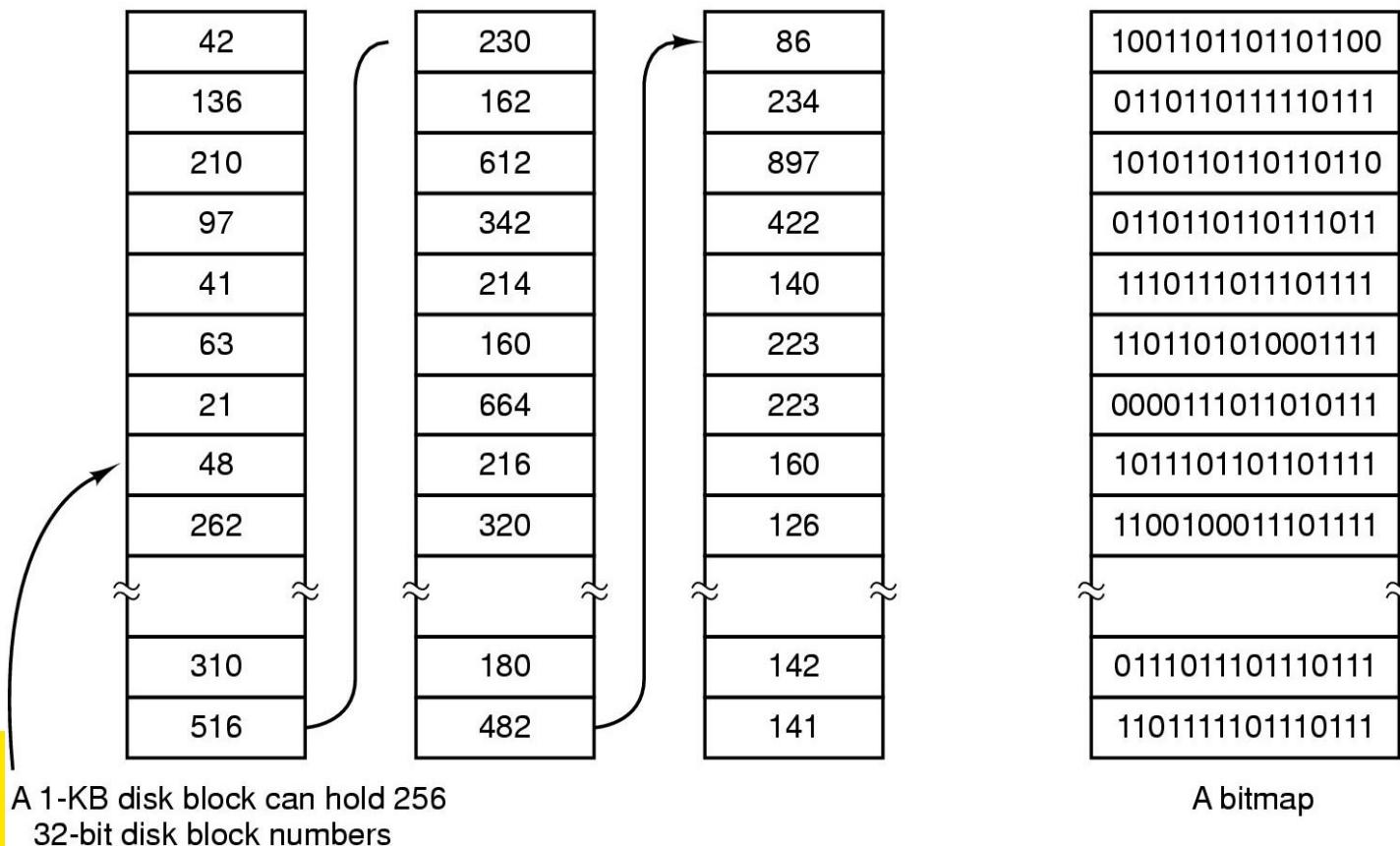
- i-nodes are allocated dynamically, hence free-space management is required for i-nodes
    - Use fixed-size i-nodes to simplify dynamic allocation
    - Reserve the last i-node entry for a pointer (a block number) to an extension i-node.

# i-node implementation issues



# i-node implementation issues

- Free-space management
  - Approach 1: linked list of free blocks in free blocks on disk
  - Approach 2: keep bitmaps of free blocks and free i-nodes on disk



# Free block list

- List of all unallocated blocks
- Background jobs can re-order list for better contiguity
- Store in free blocks themselves
  - Does not reduce disk capacity
- Only one block of pointers need be kept in the main memory

# Bit tables

- Individual bits in a bit vector flags used/free blocks
- 16GB disk with 512-byte blocks --> 4MB table
- May be too large to hold in main memory
- Expensive to search
  - Optimisations possible, e.g. a two level table
- Concentrating (de)allocations in a portion of the bitmap has desirable effect of concentrating access
- Simple to find contiguous free space

# Implementing directories

- Directories are stored like normal files
  - directory entries are contained inside data blocks
- The FS assigns special meaning to the content of these files
  - a directory file is a list of directory entries
  - a directory entry contains file name, attributes, and the file i-node number
    - maps human-oriented file name to a system-oriented name

# Fixed-size vs variable-size directory entries

- Fixed-size directory entries
  - Either too small
    - Example: DOS 8+3 characters
  - Or waste too much space
    - Example: 255 characters per file name
- Variable-size directory entries
  - Freeing variable length entries can create external fragmentation in directory blocks
    - Can compact when block is in RAM

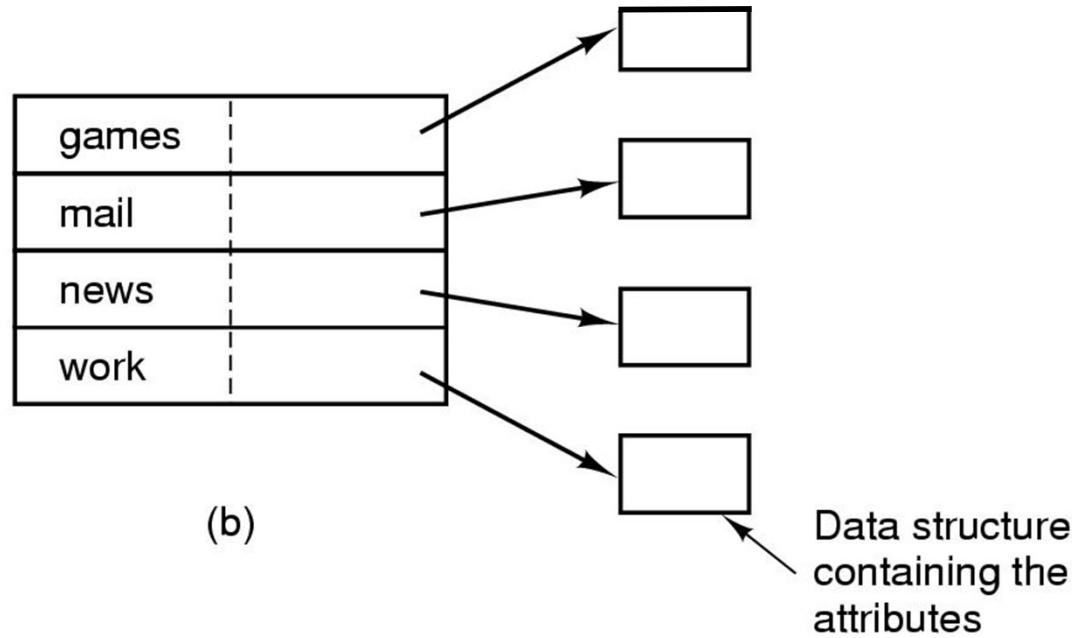
# Searching Directory Listings

- Locating a file in a directory
  - Linear scan
    - Implement a directory cache in software to speed-up search
  - Hash lookup
  - B-tree (100's of thousands entries)

# Storing file attributes

games	attributes
mail	attributes
news	attributes
work	attributes

(a)



(a) disk addresses and attributes in directory entry

-FAT

(b) directory in which each entry just refers to an i-node

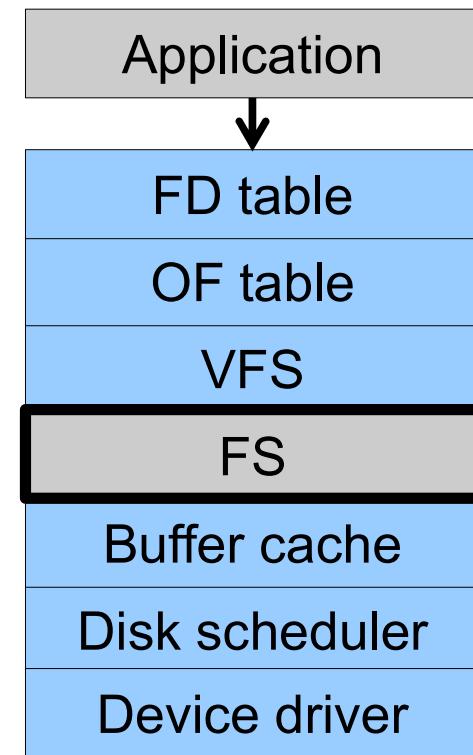
-UNIX

# Trade-off in FS block size

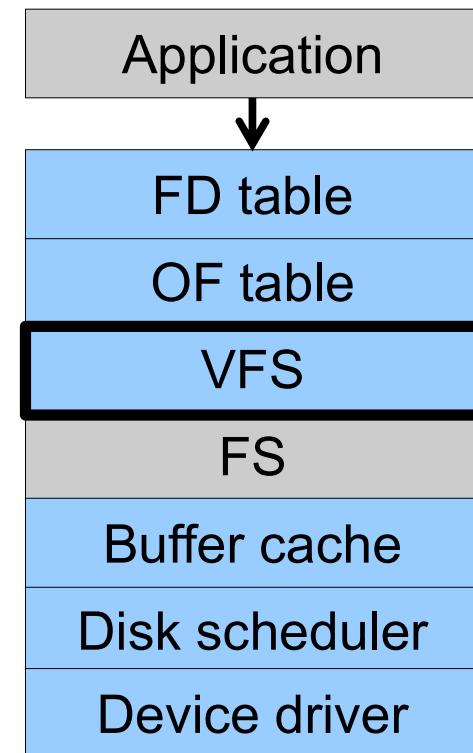
- File systems deal with 2 types of blocks
  - Disk blocks or sectors (usually 512 bytes)
  - File system blocks  $512 * 2^N$  bytes
  - What is the optimal N?
- Larger blocks require less FS metadata
- Smaller blocks waste less disk space (less internal fragmentation)
- Sequential Access
  - The larger the block size, the fewer I/O operations required
- Random Access
  - The larger the block size, the more unrelated data loaded.
  - Spatial locality of access improves the situation
- Choosing an appropriate block size is a compromise

# UNIX File Management (continued)

# OS storage stack (recap)



# Virtual File System (VFS)



# Older Systems only had a single file system

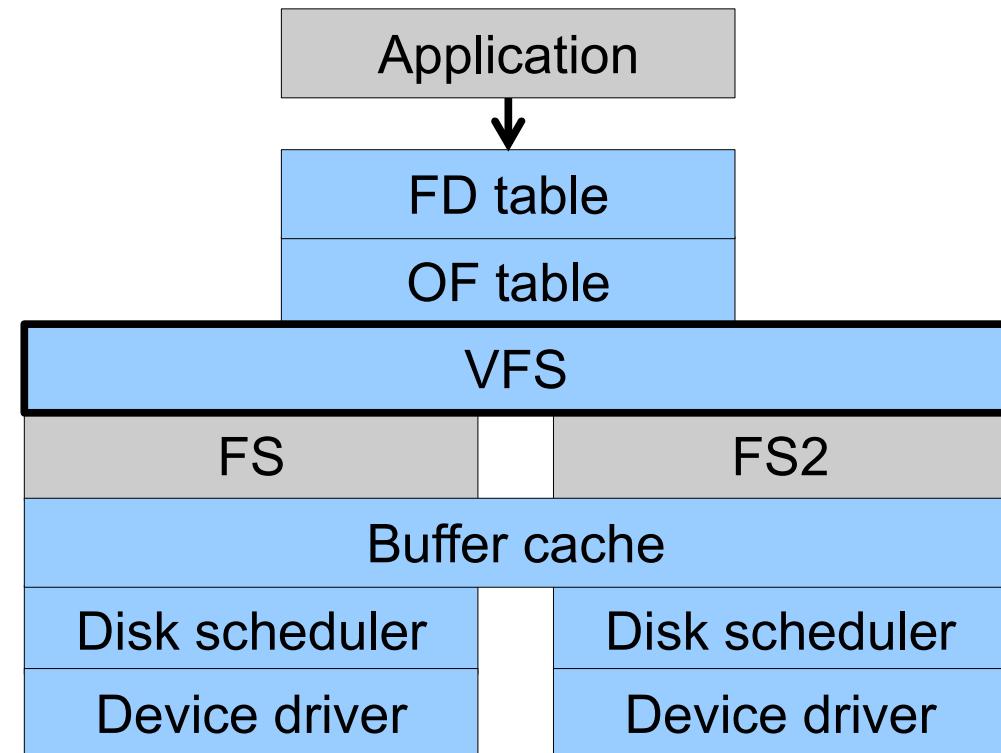
- They had file system specific open, close, read, write, ... calls.
- However, modern systems need to support many file system types
  - ISO9660 (CDROM), MSDOS (floppy), ext2fs, tmpfs

# Supporting Multiple File Systems

## Alternatives

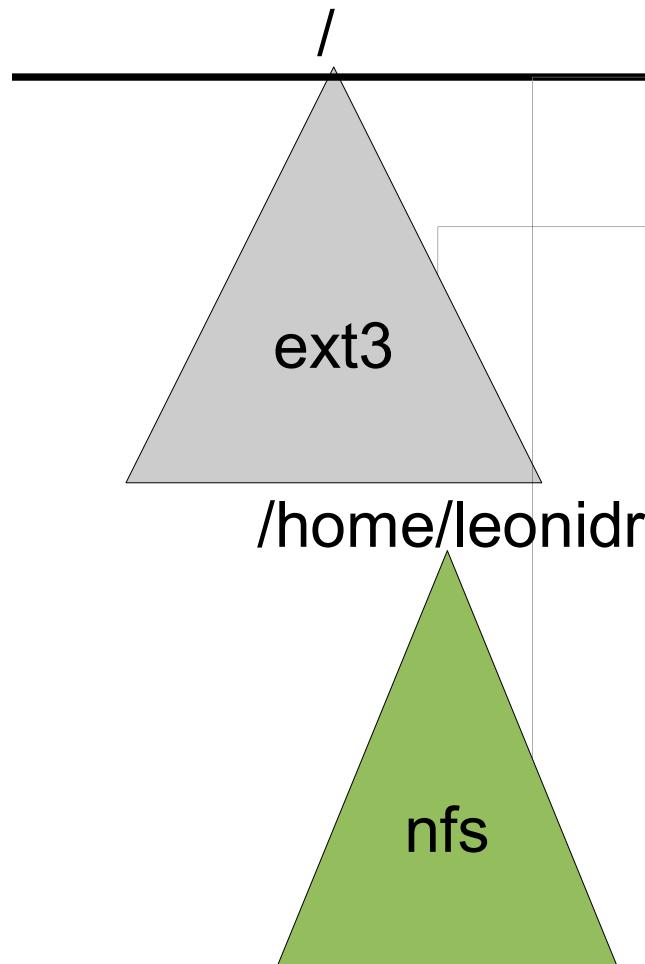
- Change the file system code to understand different file system types
  - Prone to code bloat, complex, non-solution
- Provide a framework that separates file system independent and file system dependent code.
  - Allows different file systems to be “plugged in”

# Virtual File System (VFS)



NSW  
SYDNEY

# Virtual file system (VFS)

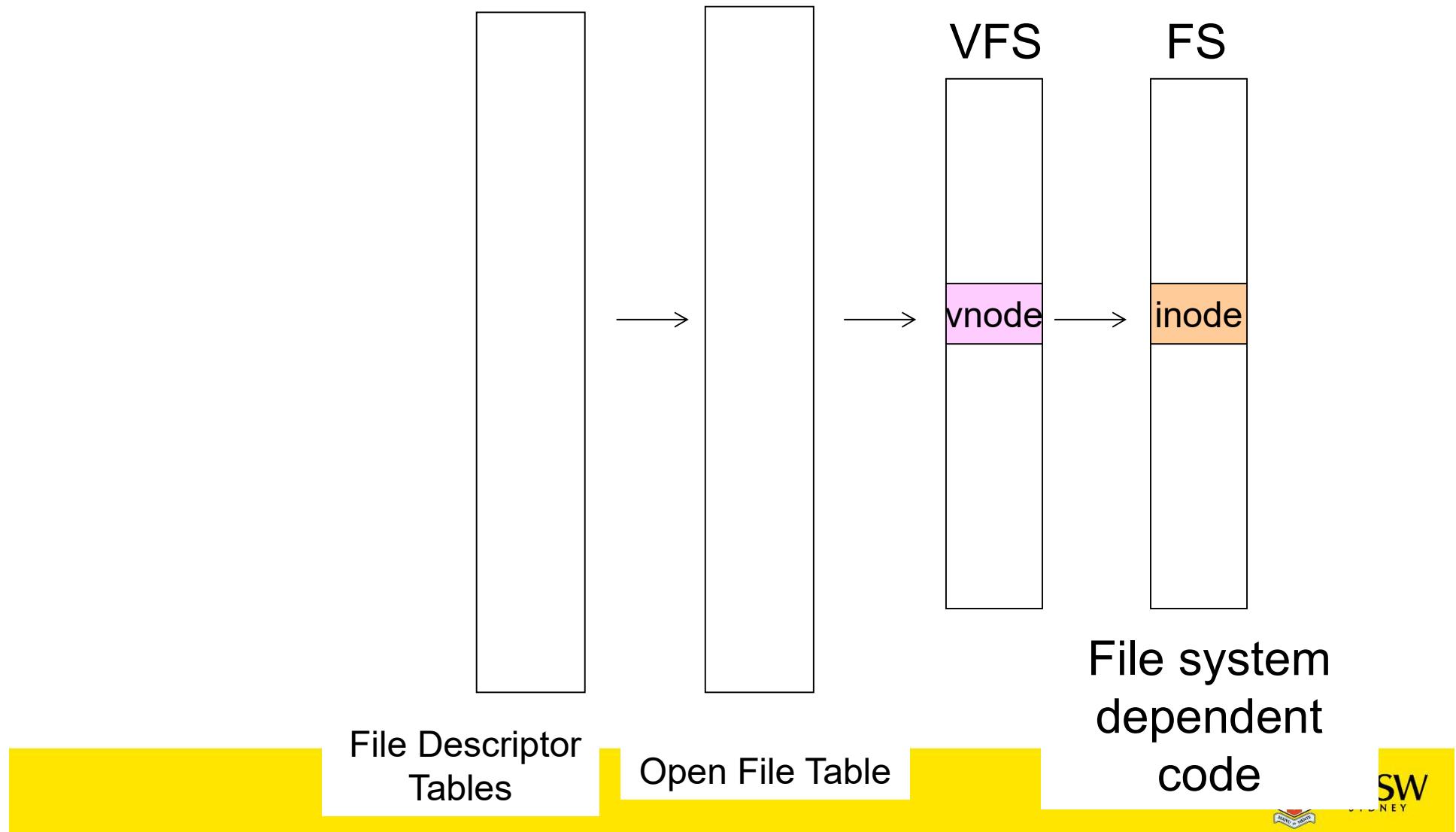


Traversing the directory hierarchy  
may require VFS to issue requests  
to several underlying file systems

# Virtual File System (VFS)

- Provides single system call interface for many file systems
  - E.g., UFS, Ext2, XFS, DOS, ISO9660,...
- Transparent handling of network file systems
  - E.g., NFS, AFS, CODA
- File-based interface to arbitrary device drivers (`/dev`)
- File-based interface to kernel data structures (`/proc`)
- Provides an indirection layer for system calls
  - File operation table set up at file open time
  - Points to actual handling code for particular type
  - Further file operations redirected to those functions

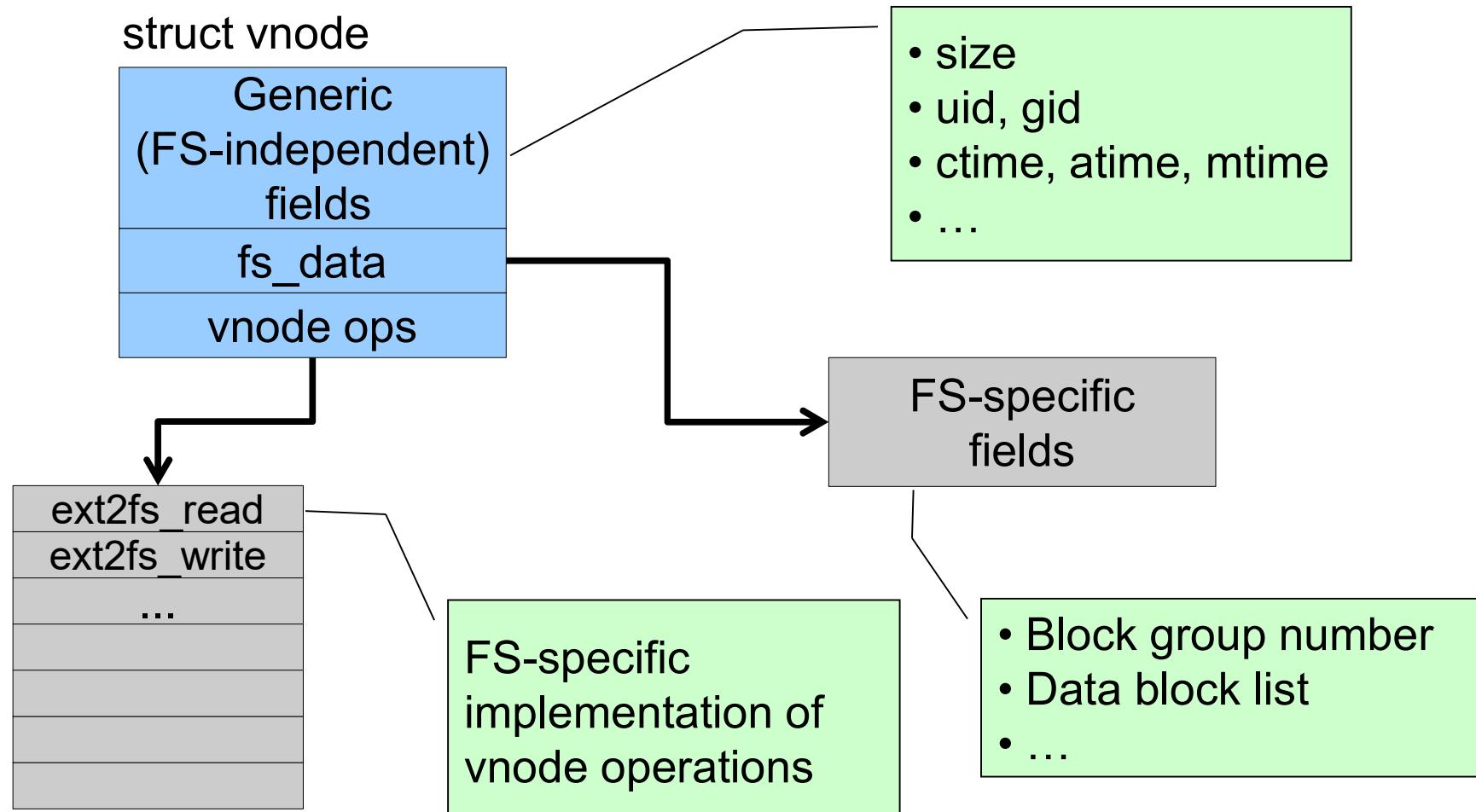
# The file system independent code deals with vfs and vnodes



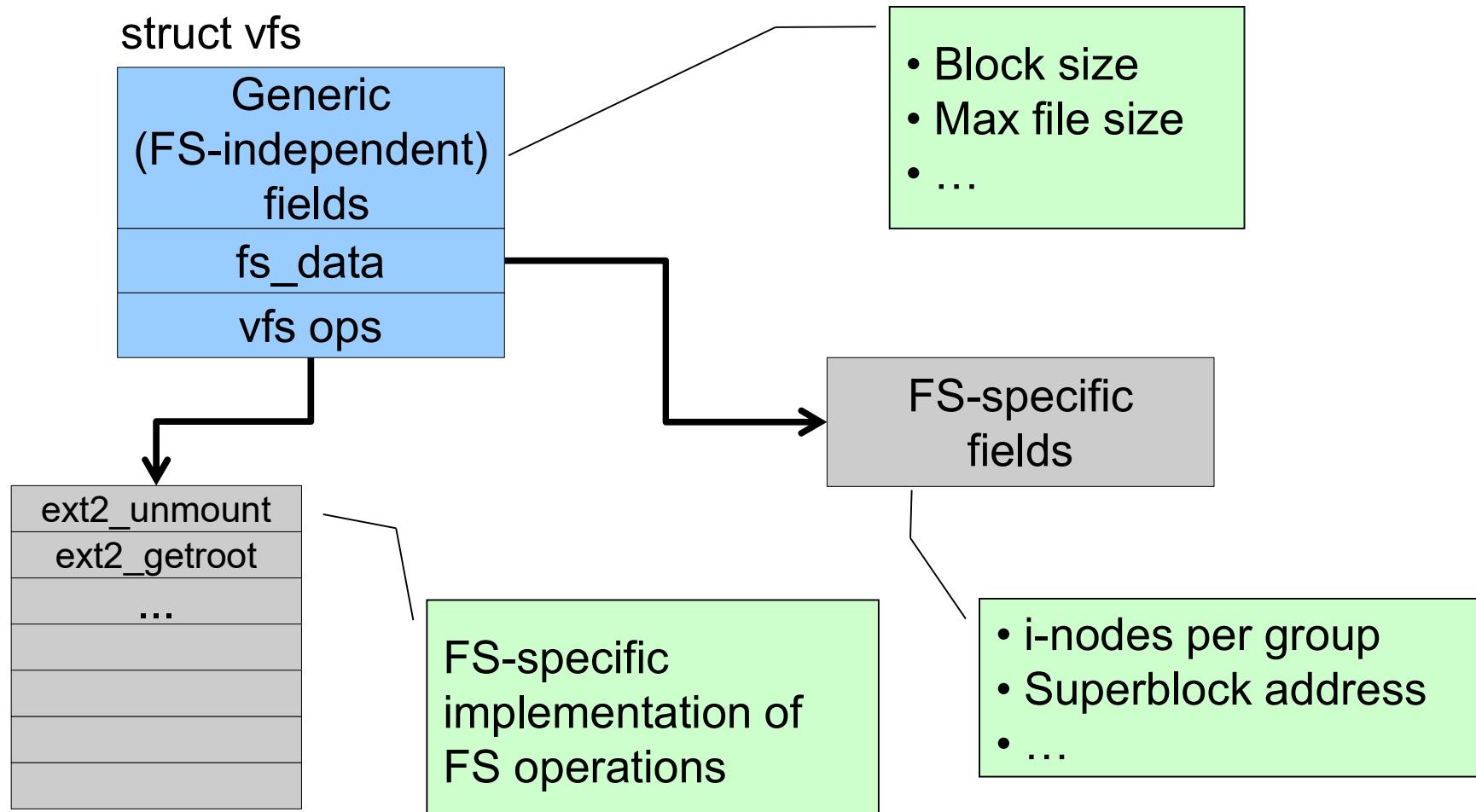
# VFS Interface

- Reference
  - S.R. Kleiman., *"Vnodes: An Architecture for Multiple File System Types in Sun Unix,"* USENIX Association: Summer Conference Proceedings, Atlanta, 1986
  - Linux and OS/161 differ slightly, but the principles are the same
- Two major data types
  - VFS
    - Represents all file system types
    - Contains pointers to functions to manipulate each file system as a whole (e.g. mount, umount)
      - Form a standard interface to the file system
  - Vnode
    - Represents a file (inode) in the underlying filesystem
    - Points to the real inode
    - Contains pointers to functions to manipulate files/inodes (e.g. open, close, read, write,...)

# Vfs and Vnode Structures



# Vfs and Vnode Structures



# A look at OS/161's VFS

The OS161's file system type

Represents interface to a mounted filesystem

```
struct fs {  
    int          (*fs_sync) (struct fs *);  
    const char   *(*fs_getvolname) (struct fs *);  
    struct vnode *(*fs_getroot) (struct fs *);  
    int          (*fs_unmount) (struct fs *);  
  
    void *fs_data;  
};
```

Force the  
filesystem to  
flush its content  
to disk

Retrieve the  
volume name

Retrieve the vnode  
associated with the  
root of the  
filesystem

Unmount the filesystem  
Note: mount called via  
function ptr passed to  
**vfs\_mount**

Private file system  
specific data

```
Count the  
number of  
“references”  
to this vnode
```

Vnode

```
Lock for mutual  
exclusive  
access to  
counts
```

```
struct vnode {  
    int vn_refcount;  
    struct spinlock vn_countlock;k  
    struct fs *vn_fs;  
    void *vn_data;  
    const struct vnode_ops *vn_ops;  
};
```

Pointer to FS specific  
vnode data (e.g. in-  
memory copy of  
inode)

Pointer to FS  
containing  
the vnode

Array of pointers  
to functions  
operating on  
vnodes

# Vnode Ops

```
struct vnode_ops {
    unsigned long vop_magic;           /* should always be VOP_MAGIC */

    int (*vop_eachopen) (struct vnode *object, int flags_from_open);
    int (*vop_reclaim) (struct vnode *vnode);

    int (*vop_read) (struct vnode *file, struct uio *uio);
    int (*vop_readlink) (struct vnode *link, struct uio *uio);
    int (*vop_getdirentry) (struct vnode *dir, struct uio *uio);
    int (*vop_write) (struct vnode *file, struct uio *uio);
    int (*vop_ioctl) (struct vnode *object, int op, userptr_t data);
    int (*vop_stat) (struct vnode *object, struct stat *statbuf);
    int (*vop_gettime) (struct vnode *object, int *result);
    int (*vop_isseekable) (struct vnode *object, off_t pos);
    int (*vop_fsync) (struct vnode *object);
    int (*vop_mmap) (struct vnode *file /* add stuff */);
    int (*vop_truncate) (struct vnode *file, off_t len);
    int (*vop_namefile) (struct vnode *file, struct uio *uio);
```

# Vnode Ops

```
int (*vop_creat) (struct vnode *dir,
                  const char *name, int excl,
                  struct vnode **result);
int (*vop_symlink) (struct vnode *dir,
                     const char *contents, const char *name);
int (*vop_mkdir) (struct vnode *parentdir,
                  const char *name);
int (*vop_link) (struct vnode *dir,
                 const char *name, struct vnode *file);
int (*vop_remove) (struct vnode *dir,
                   const char *name);
int (*vop_rmdir) (struct vnode *dir,
                  const char *name);

int (*vop_rename) (struct vnode *vn1, const char *name1,
                   struct vnode *vn2, const char *name2);

int (*vop_lookup) (struct vnode *dir,
                  char *pathname, struct vnode **result);
int (*vop_lookparent) (struct vnode *dir,
                      char *pathname, struct vnode **result,
                      char *buf, size_t len);
};
```

# Vnode Ops

- Note that most operations are on vnodes. How do we operate on file names?
  - Higher level API on names that uses the internal VOP\_\* functions

```
int vfs_open(char *path, int openflags, mode_t mode, struct vnode **ret);
void vfs_close(struct vnode *vn);
int vfs_readlink(char *path, struct uio *data);
int vfs_symlink(const char *contents, char *path);
int vfs_mkdir(char *path);
int vfs_link(char *oldpath, char *newpath);
int vfs_remove(char *path);
int vfs_rmdir(char *path);
int vfs_rename(char *oldpath, char *newpath);

int vfs_chdir(char *path);
int vfs_getcwd(struct uio *buf);
```

# Example: OS/161 emufs vnode ops

```
/*
 * Function table for emufs
 * files.
 */
static const struct vnode_ops
emufs_fileops = {
    VOP_MAGIC, /* mark this a
    valid vnode ops table */

    emufs_eachopen,
    emufs_reclaim,

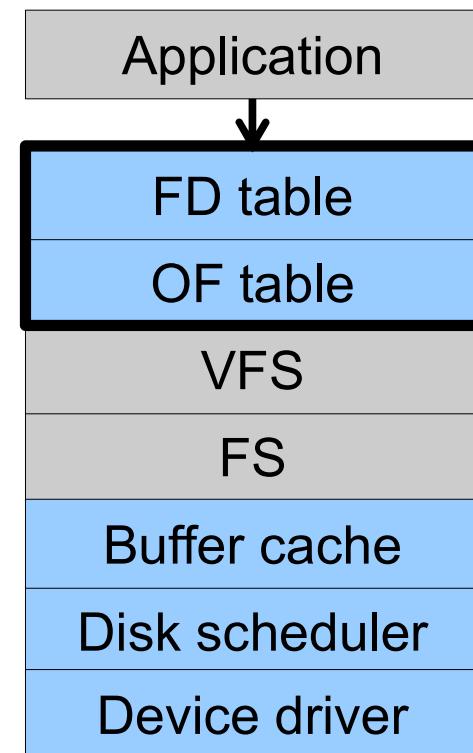
    emufs_read,
    NOTDIR, /* readlink */
    NOTDIR, /* getdirentry */
    emufs_write,
    emufs_ioctl,
    emufs_stat,
    emufs_file_gettype,
    emufs_tryseek,
    emufs_fsync,
    UNIMP, /* mmap */
    emufs_truncate,
    NOTDIR, /* namefile */

    NOTDIR, /* creat */
    NOTDIR, /* symlink */
    NOTDIR, /* mkdir */
    NOTDIR, /* link */
    NOTDIR, /* remove */
    NOTDIR, /* rmdir */
    NOTDIR, /* rename */

    NOTDIR, /* lookup */
    NOTDIR, /* lookparent */
};

}
```

# File Descriptor & Open File Tables



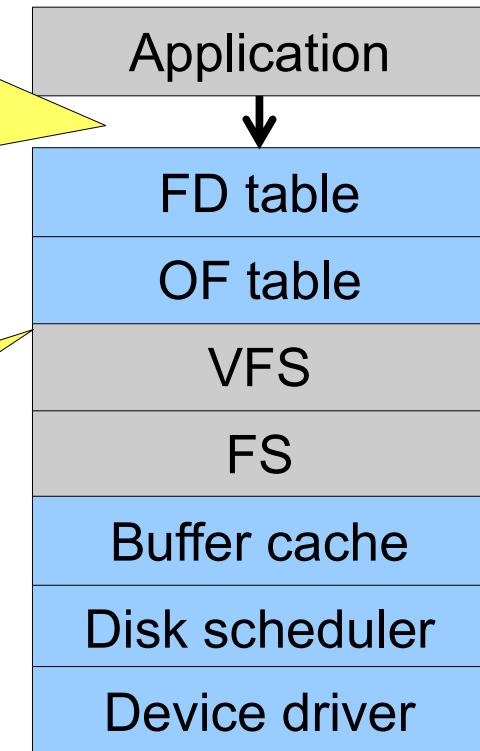
# Motivation

**System call interface:**

```
fd = open("file", ...);  
read(fd, ...); write(fd, ...); lseek(fd, ...);  
close(fd);
```

**VFS interface:**

```
vnode = vfs_open("file", ...);  
vop_read(vnode, uio);  
vop_write(vnode, uio);  
vop_close(vnode);
```



# File Descriptors

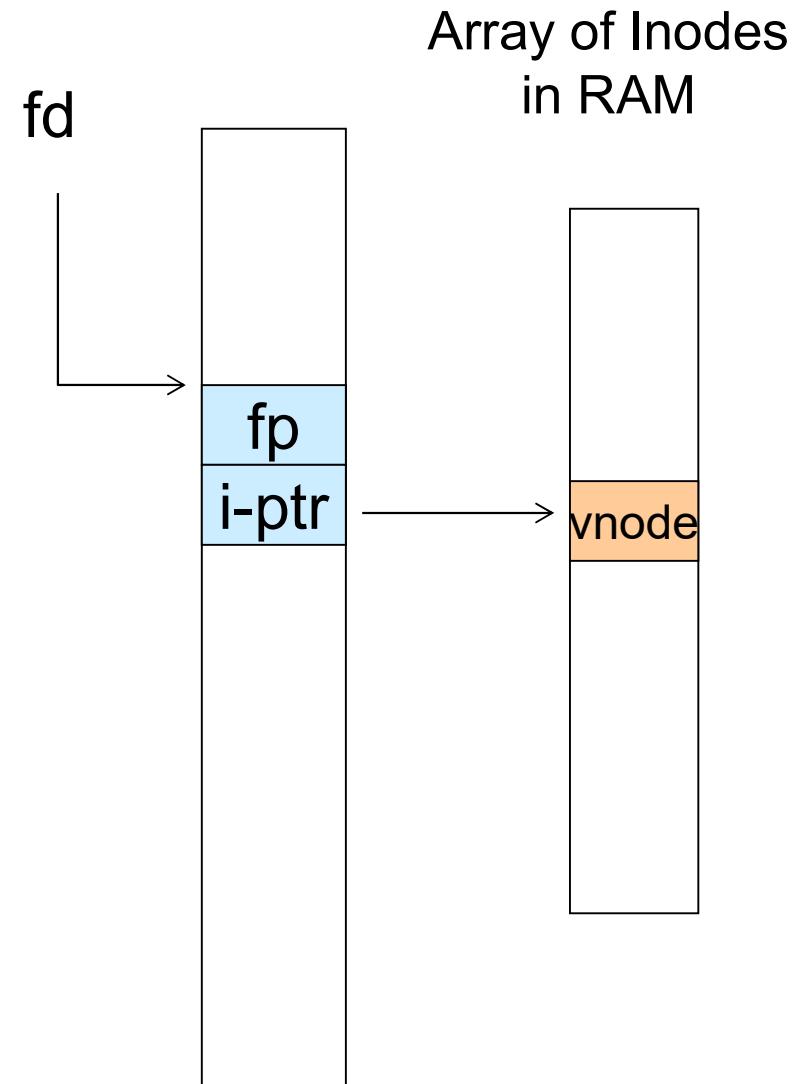
- File descriptors
  - Each open file has a file descriptor
  - Read/Write/Iseek/.... use them to specify which file to operate on.
- State associated with a file descriptor
  - File pointer
    - Determines where in the file the next read or write is performed
  - Mode
    - Was the file opened read-only, etc....

# An Option?

- Use vnode numbers as file descriptors and add a file pointer to the vnode
- Problems
  - What happens when we concurrently open the same file twice?
  - We should get two separate file descriptors and file pointers....

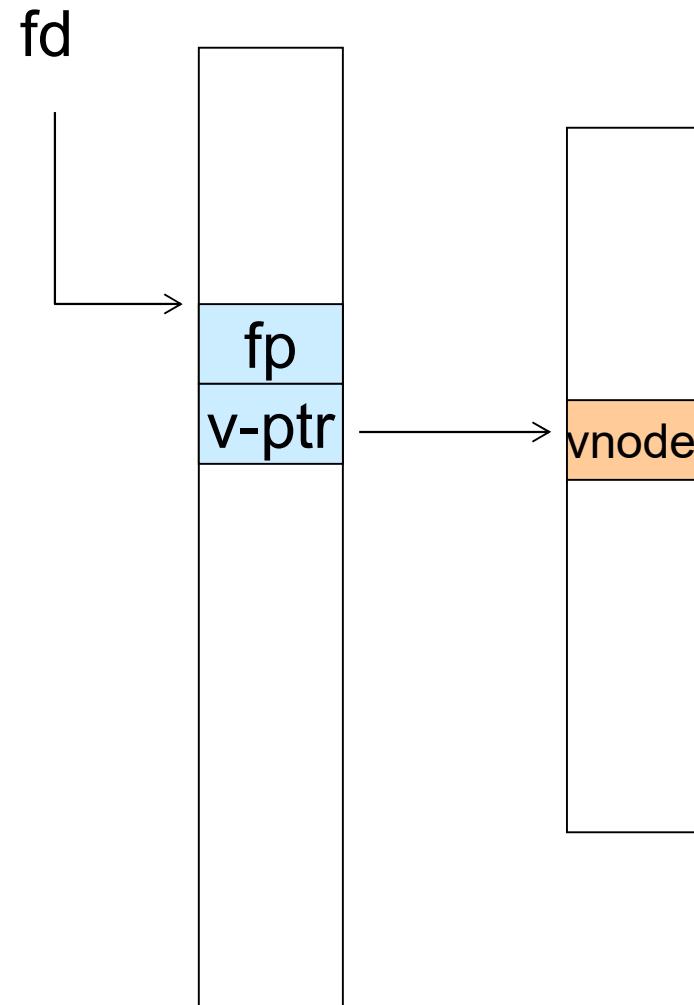
# An Option?

- Single global open file array
  - *fd* is an index into the array
  - Entries contain file pointer and pointer to a vnode



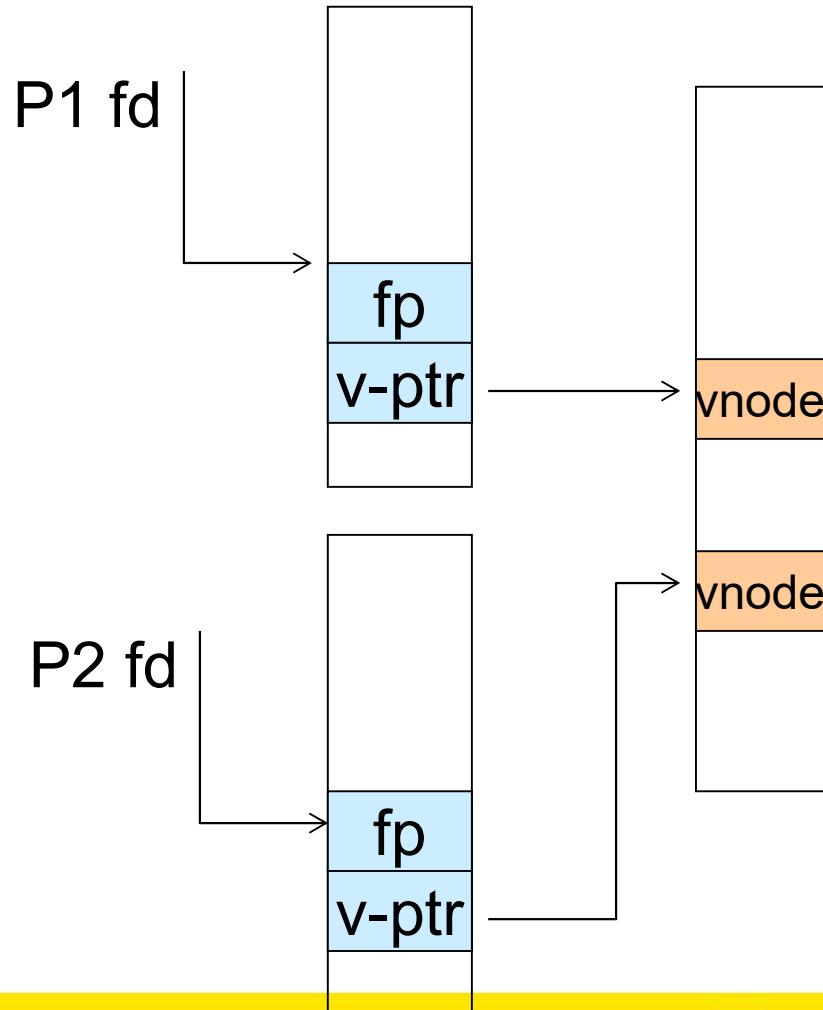
# Issues

- File descriptor 1 is stdout
  - Stdout is console for some processes
  - A file for others
  - Entry 1 needs to be different per process!



# Per-process File Descriptor Array

- Each process has its own open file array
  - Contains fp, v-ptr etc.
  - *Fd* 1 can point to any vnode for each process (console, log file).



# Issue

- Fork

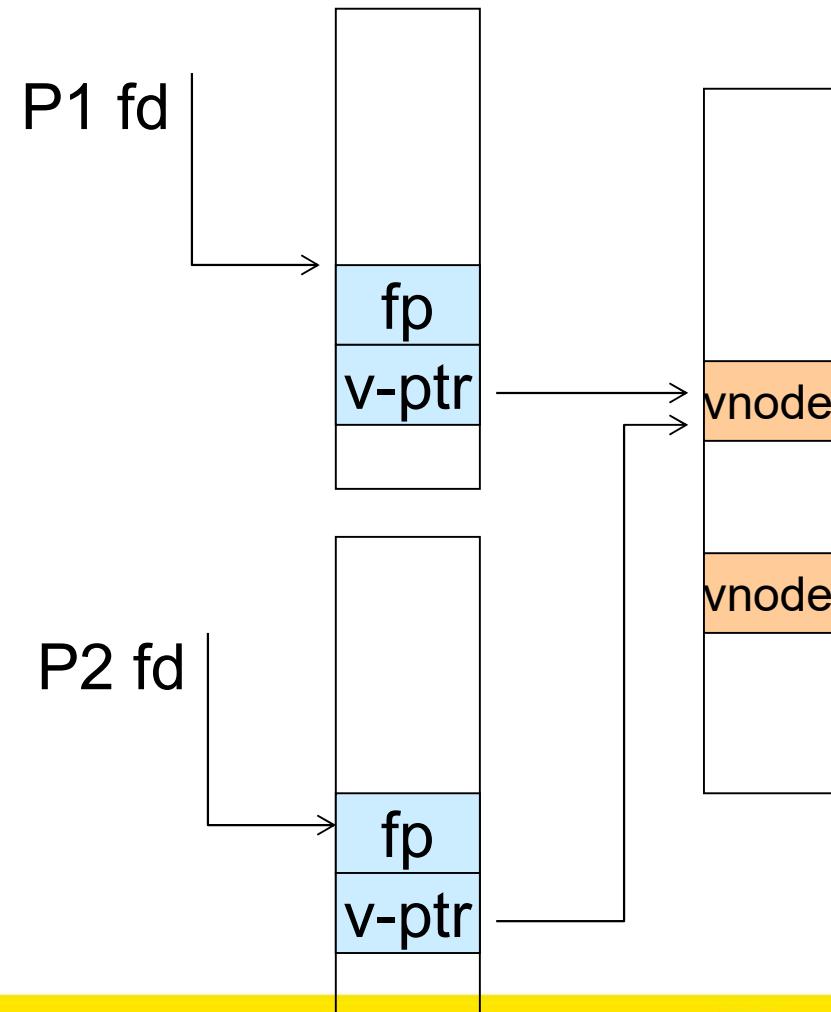
- Fork defines that the child shares the file pointer with the parent

- Dup2

- Also defines the file descriptors share the file pointer

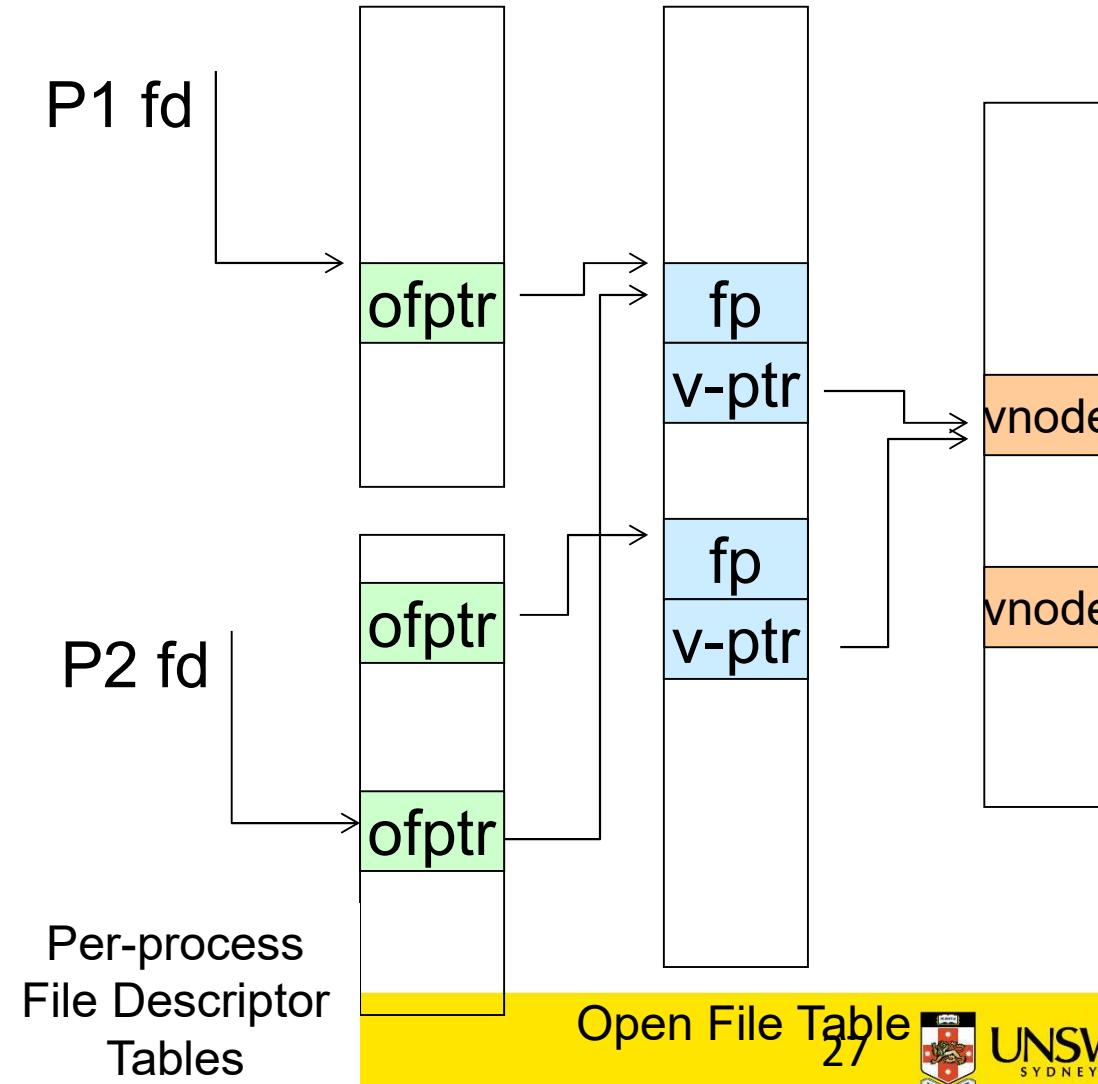
- With per-process table, we can only have independent file pointers

- Even when accessing the same file



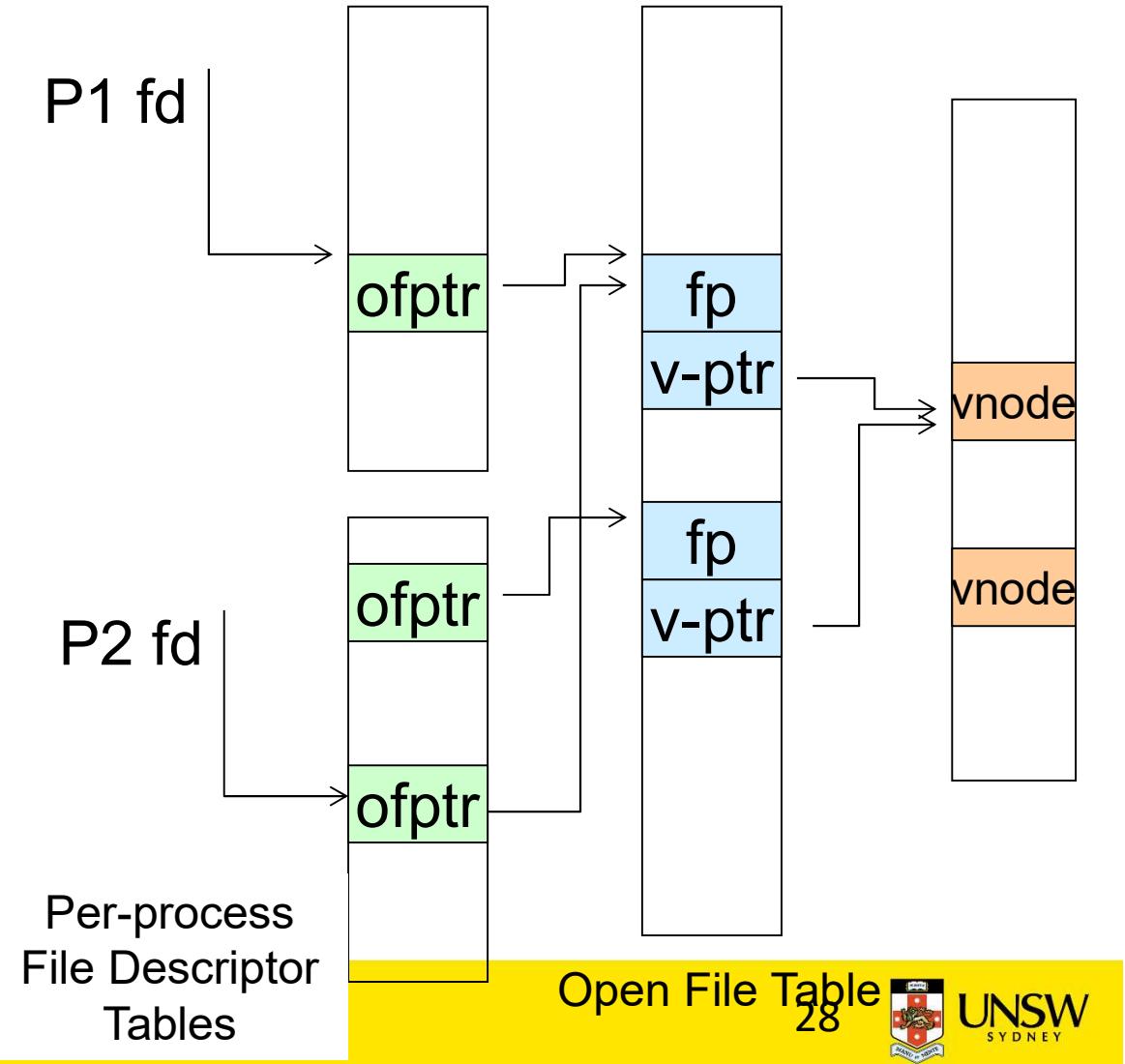
# Per-Process *fd* table with global open file table

- Per-process file descriptor array
  - Contains pointers to *open file table entry*
- Open file table array
  - Contain entries with a fp and pointer to an vnode.
- Provides
  - Shared file pointers if required
  - Independent file pointers if required
- Example:
  - All three *fds* refer to the same file, two share a file pointer, one has an independent file pointer

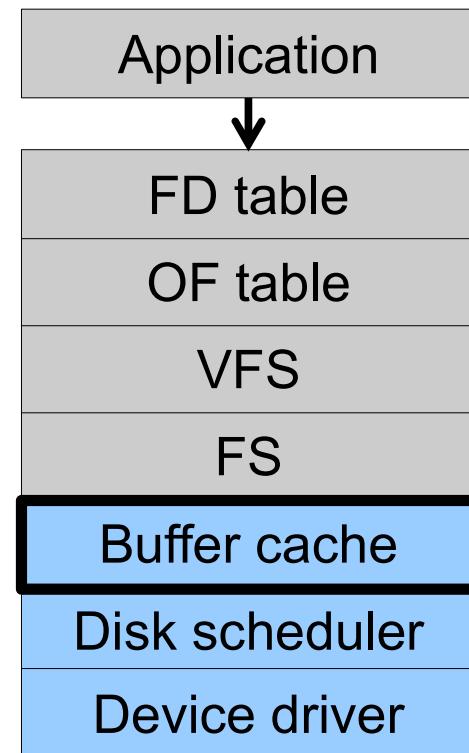


# Per-Process *fd* table with global open file table

- Used by Linux and most other Unix operating systems



# Buffer Cache

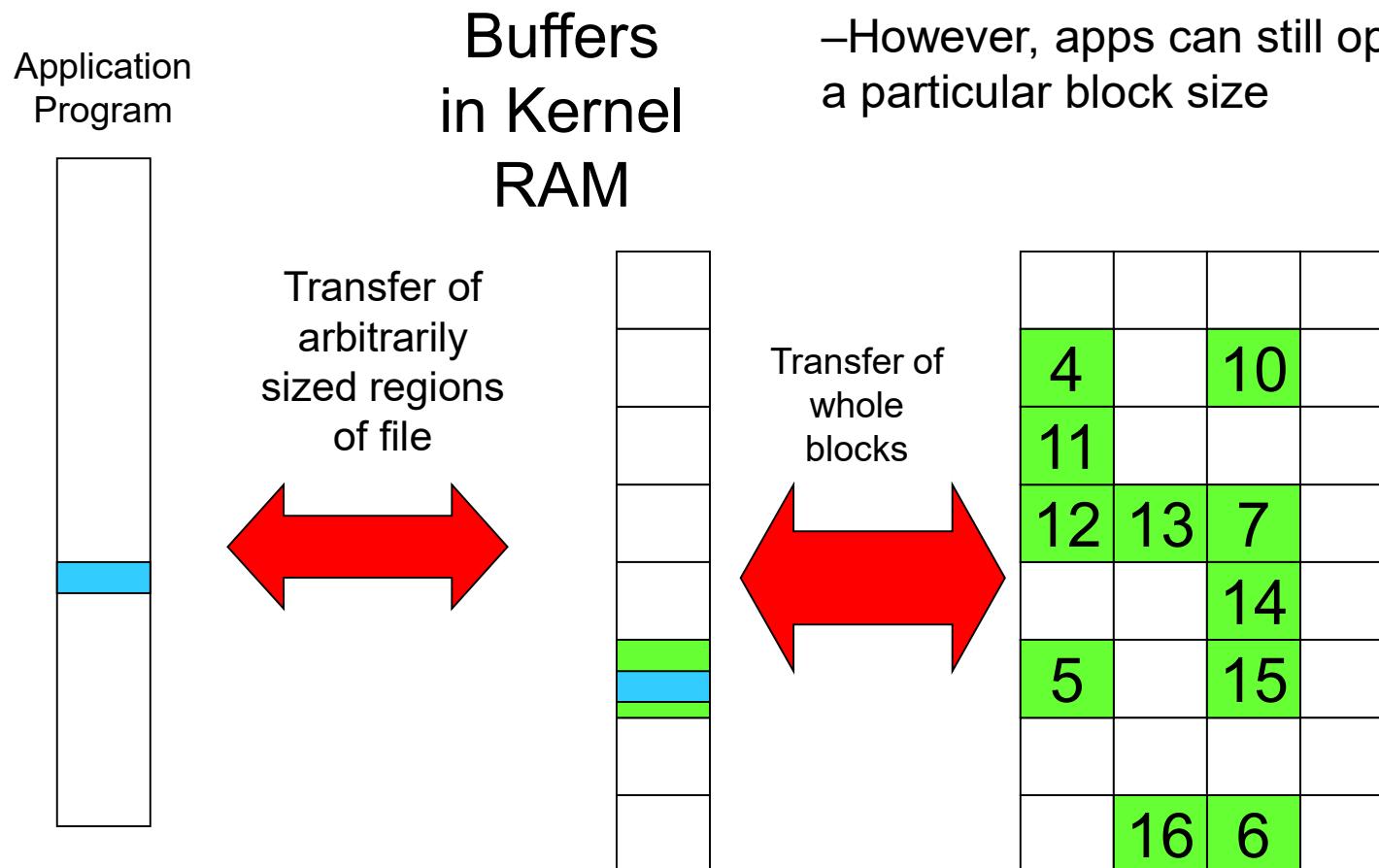


NSW  
SYDNEY

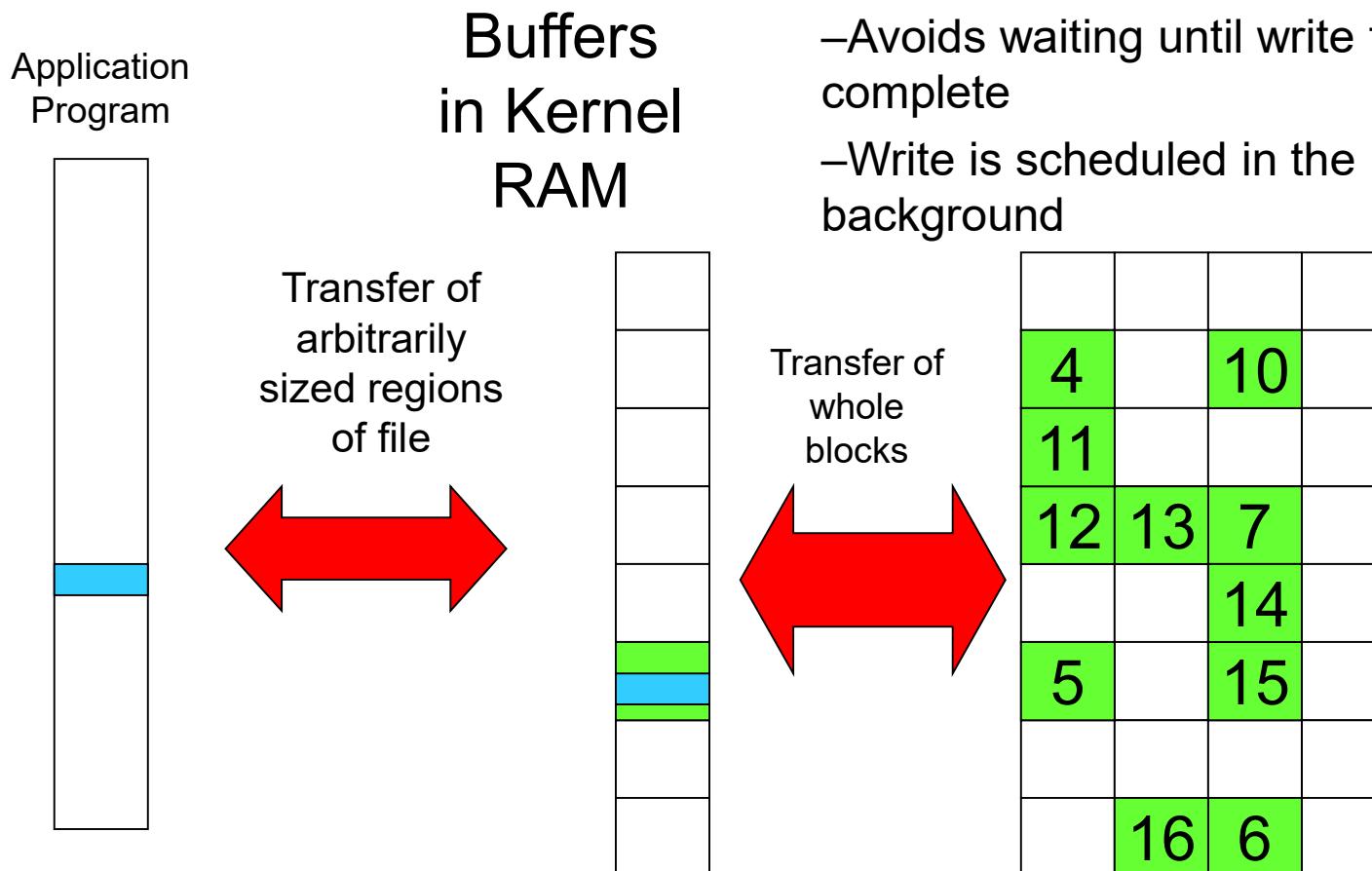
# Buffer

- **Buffer:**
  - Temporary storage used when transferring data between two entities
  - Especially when the entities work at different rates
  - Or when the unit of transfer is incompatible
  - Example: between application program and disk

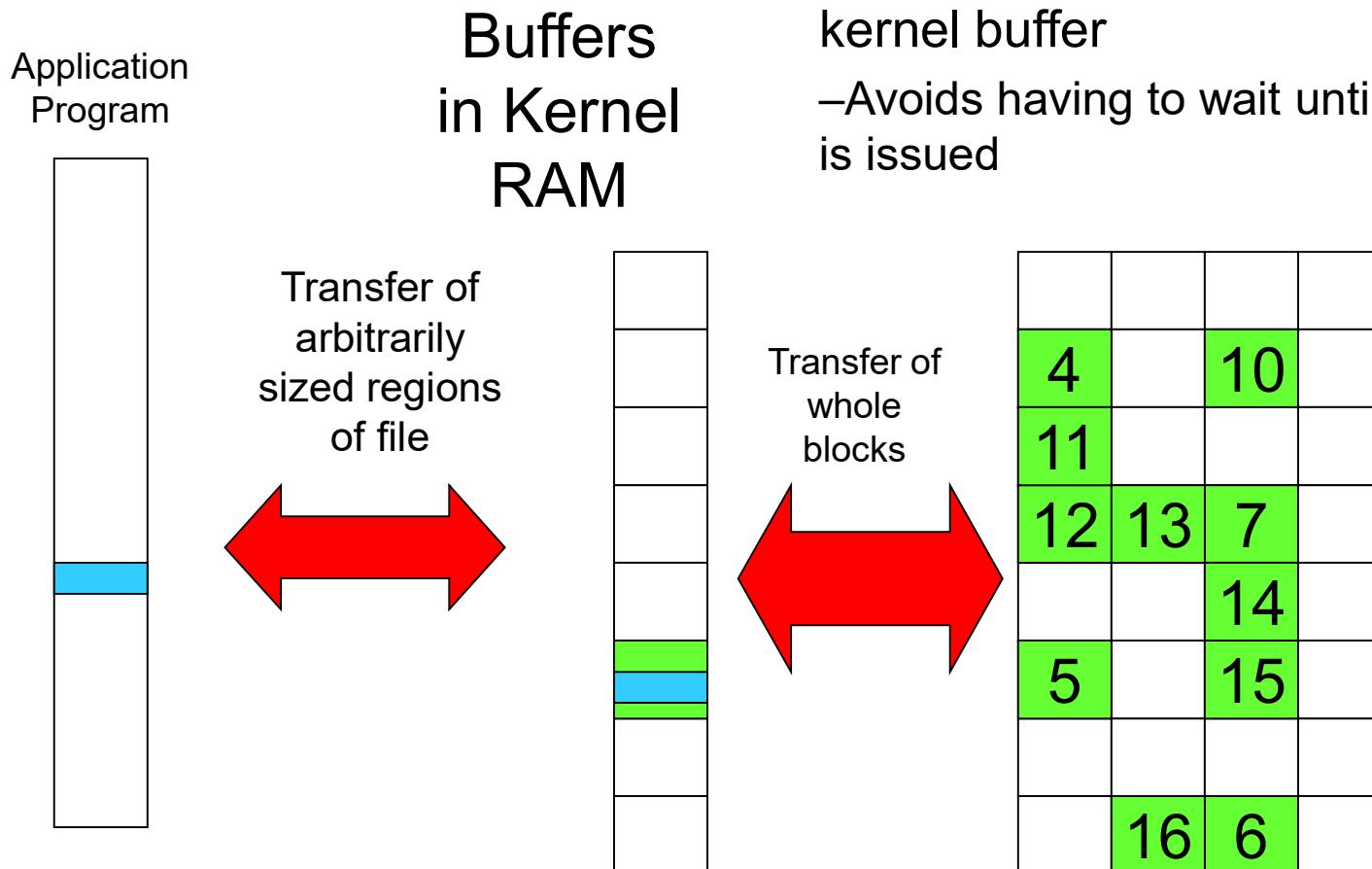
# Buffering Disk Blocks



# Buffering Disk Blocks



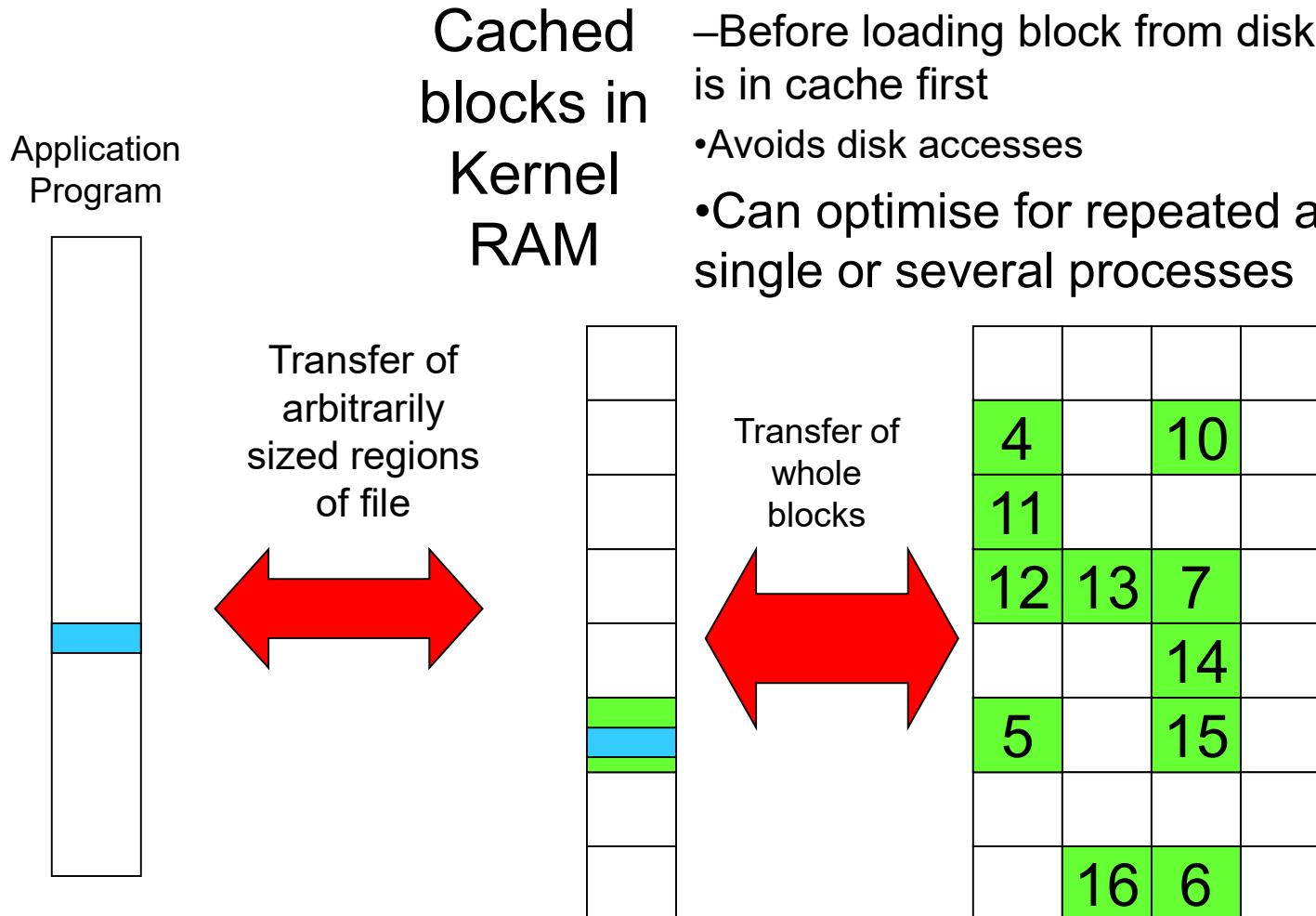
# Buffering Disk Blocks



# Cache

- Cache:
  - Fast storage used to temporarily hold data to speed up repeated access to the data
- Example: Main memory can cache disk blocks

# Caching Disk Blocks



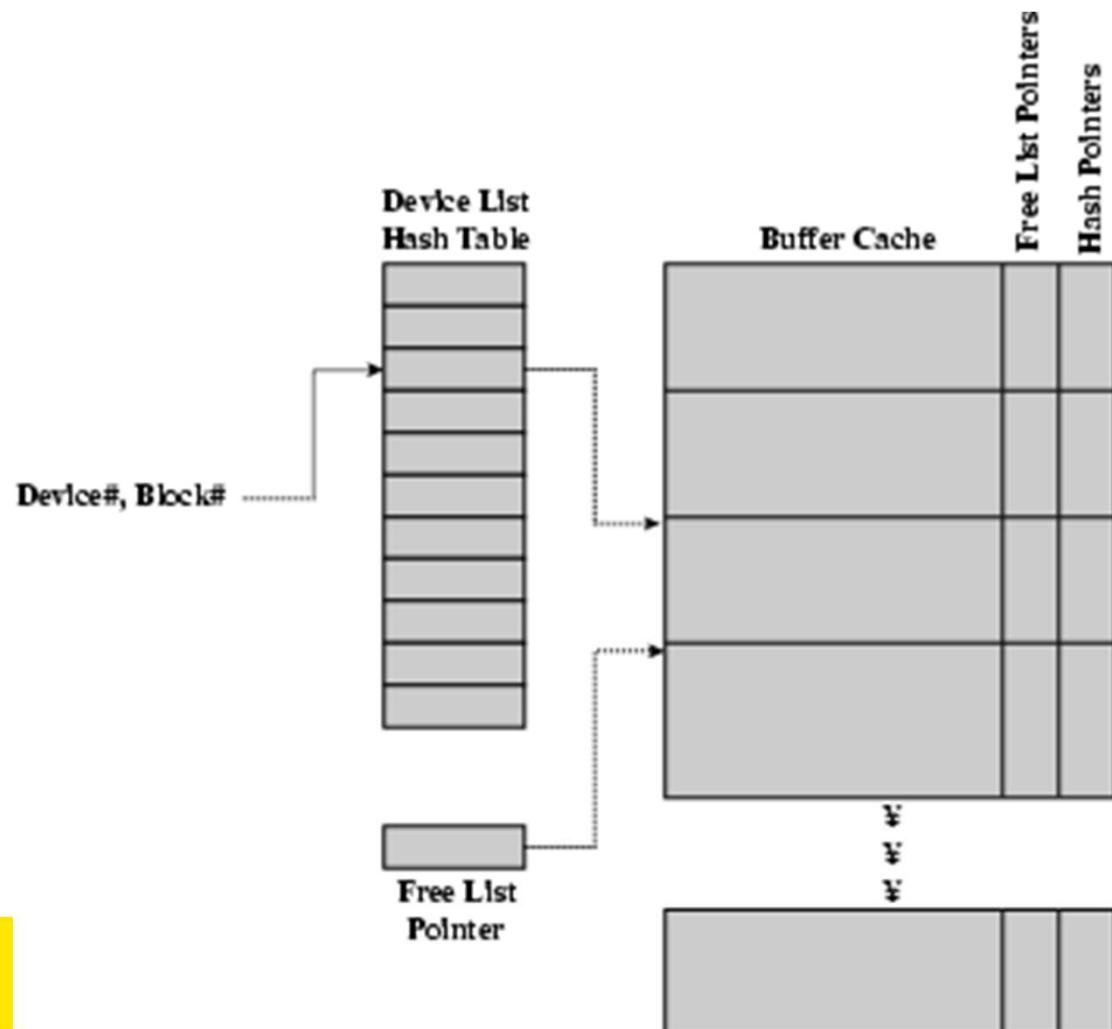
# Buffering and caching are related

- Data is read into buffer; an extra independent cache copy would be wasteful
- After use, block should be cached
- Future access may hit cached copy
- Cache utilises unused kernel memory space;
  - may have to shrink, depending on memory demand

# Unix Buffer Cache

## On read

- Hash the device#, block#
- Check if match in buffer cache
- Yes, simply use in-memory copy
- No, follow the collision chain
- If not found, we load block from disk into buffer cache



# Replacement

- What happens when the buffer cache is full and we need to read another block into memory?
  - We must choose an existing entry to replace
  - Need a policy to choose a victim
    - Can use First-in First-out
    - Least Recently Used, or others.
      - Timestamps required for LRU implementation
    - However, is strict LRU what we want?

# File System Consistency

- File data is expected to survive
- Strict LRU could keep modified critical data in memory forever if it is frequently used.

# File System Consistency

- Generally, cached disk blocks are prioritised in terms of how critical they are to file system consistency
  - Directory blocks, inode blocks if lost can corrupt entire filesystem
    - E.g. imagine losing the root directory
    - These blocks are usually scheduled for immediate write to disk
    - Data blocks if lost corrupt only the file that they are associated with
    - These blocks are only scheduled for write back to disk periodically
    - In UNIX, flushd (*flush daemon*) flushes all modified blocks to disk every 30 seconds

# File System Consistency

- Alternatively, use a write-through cache
  - All modified blocks are written immediately to disk
  - Generates much more disk traffic
    - Temporary files written back
    - Multiple updates not combined
  - Used by DOS
- Gave okay consistency when
  - » Floppies were removed from drives
  - » Users were constantly resetting (or crashing) their machines
- Still used, e.g. USB storage devices

# Case study: ext2 FS



# The ext2 file system

- Second Extended Filesystem
  - The main Linux FS before ext3
  - Evolved from Minix filesystem (via “Extended Filesystem”)
- Features
  - Block size (1024, 2048, and 4096) configured at FS creation
  - inode-based FS
  - Performance optimisations to improve locality (from BSD FFS)
- Main Problem: unclean unmount → **e2fsck**
  - Ext3fs keeps a journal of (meta-data) updates
  - Journal is a file where updates are logged
  - Compatible with ext2fs

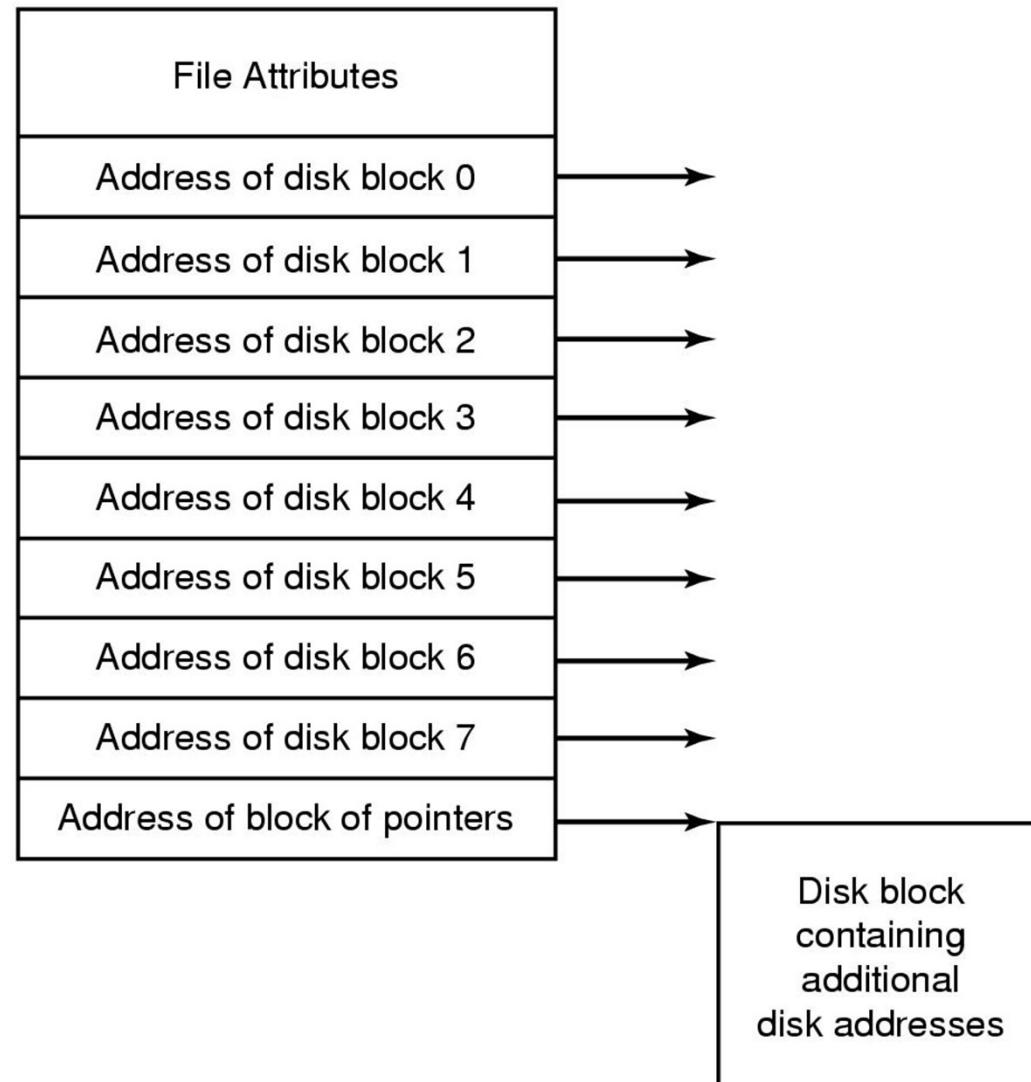


# Recap: i-nodes

- Each file is represented by an inode on disk
- Inode contains the fundamental file metadata
  - Access rights, owner, accounting info
  - (partial) block index table of a file
- Each inode has a unique number
  - System oriented name
  - Try ‘ls -i’ on Unix (Linux)
- Directories map file names to inode numbers
  - Map human-oriented to system-oriented names



# Recap: i-nodes



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
single indirect
double indirect
triple indirect

# Ext2 i-nodes

- Mode
  - Type
    - Regular file or directory
  - Access mode
    - rwxrwxrwx
- Uid
  - User ID
- Gid
  - Group ID



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
single indirect
double indirect
triple indirect

# Inode Contents

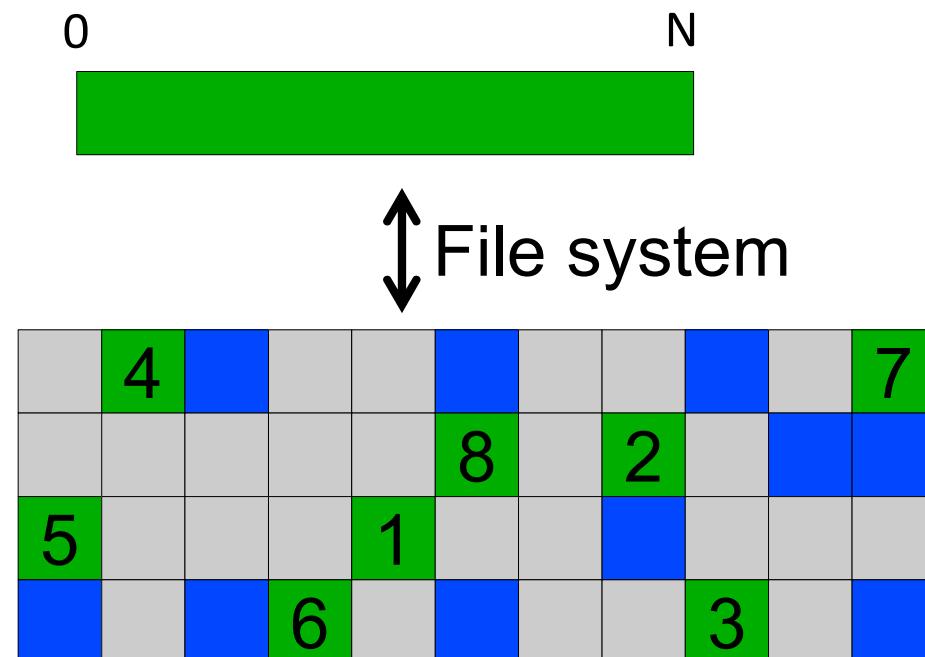
- atime
  - Time of last access
- ctime
  - Time when file was created
- mtime
  - Time when file was last modified



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
single indirect
double indirect
triple indirect

# Inode Contents - Size

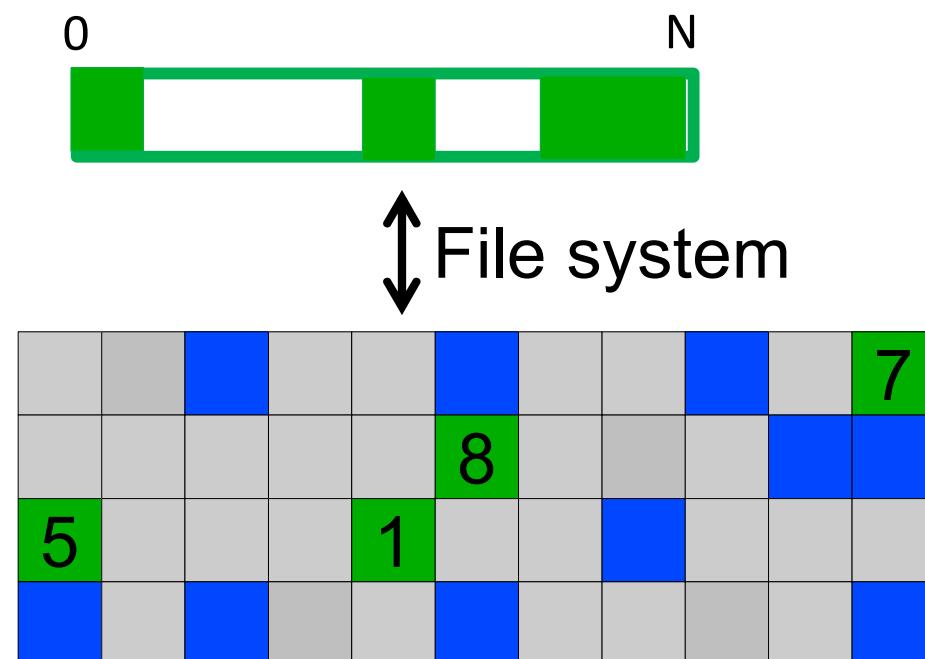
- What does ‘size of a file’ really mean?
  - The space consumed on disk?
    - With or without the metadata?
  - The number of bytes written to the file?
  - The highest byte written to the file?



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
single indirect
double indirect
triple indirect

# Inode Contents - Size

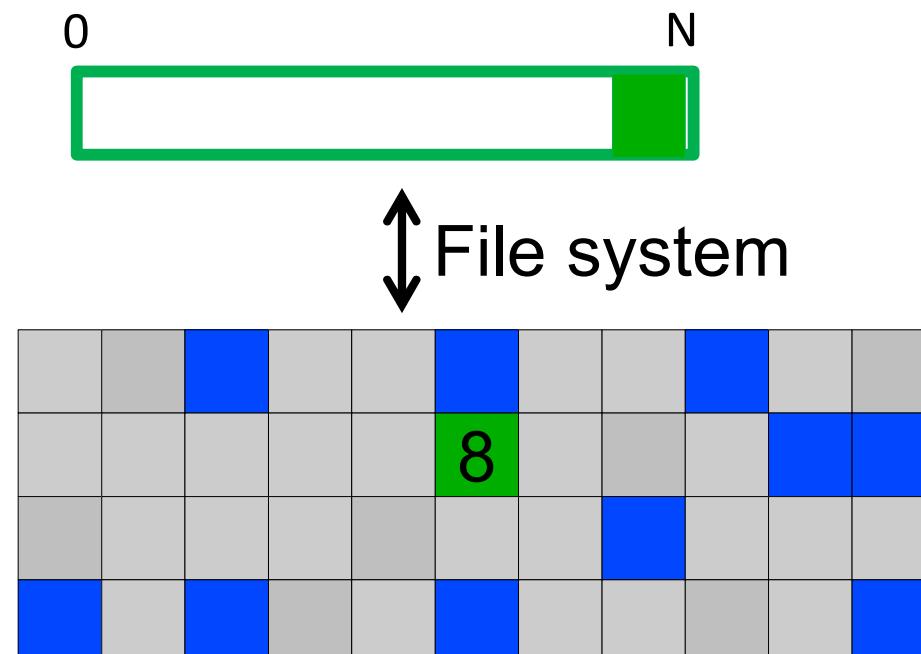
- What does ‘size of a file’ really mean?
  - The space consumed on disk?
    - With or without the metadata?
  - The number of bytes written to the file?
  - The highest byte written to the file?



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
single indirect
double indirect
triple indirect

# Inode Contents - Size

- What does ‘size of a file’ really mean?
  - The space consumed on disk?
    - With or without the metadata?
  - The number of bytes written to the file?
  - The highest byte written to the file?



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
single indirect
double indirect
triple indirect

# Inode Contents

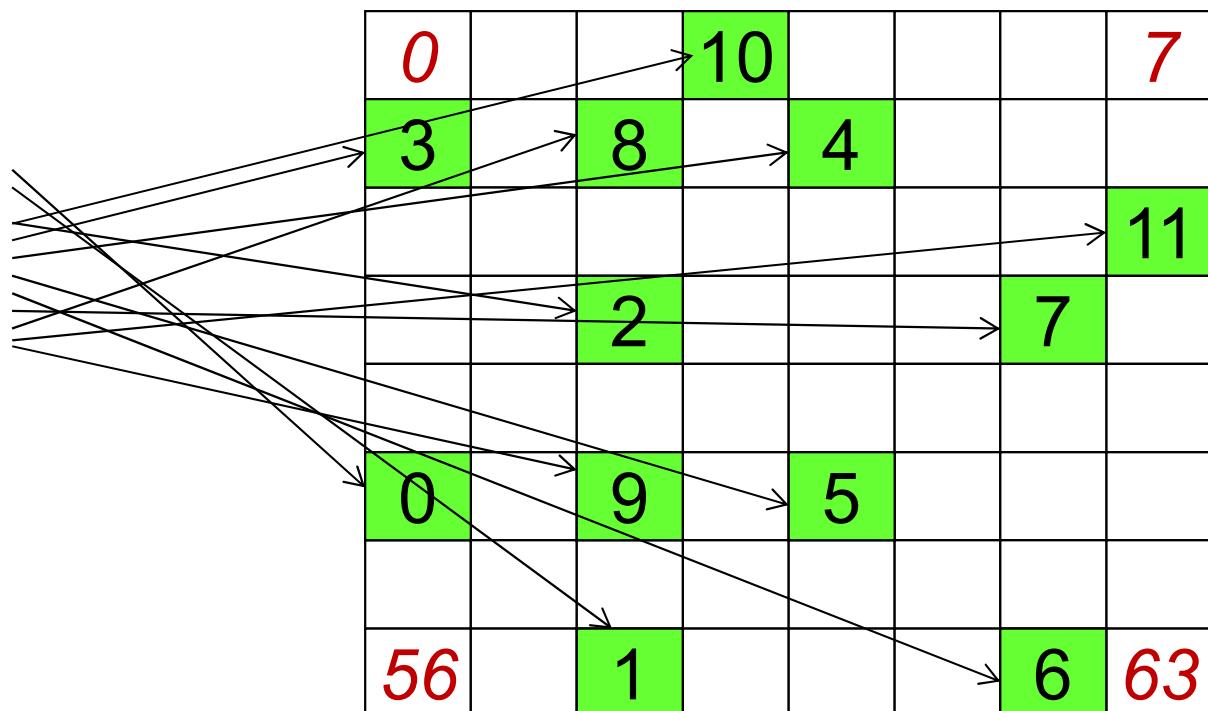
- Size
  - Offset of the highest byte written
- Block count
  - Number of disk blocks used by the file.
  - Note that number of blocks can be much less than expected given the file size
- Files can be sparsely populated
  - E.g. `write(f, "hello"); lseek(f, 1000000); write(f, "world");`
  - Only needs to store the start and end of file, not all the empty blocks in between.
  - Size = 1000005
  - Blocks = 2 + any indirect blocks



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12) 40,58,26,8,12, 44,62,30,10,42,3,21
single indirect
double indirect
triple indirect

# Inode Contents

- Direct Blocks
  - Block numbers of first 12 blocks in the file
  - Most files are small
    - We can find blocks of file *directly* from the inode



File

11
10
9
8
7
6
5
4
3
2
1
0

Disk

11



# Problem

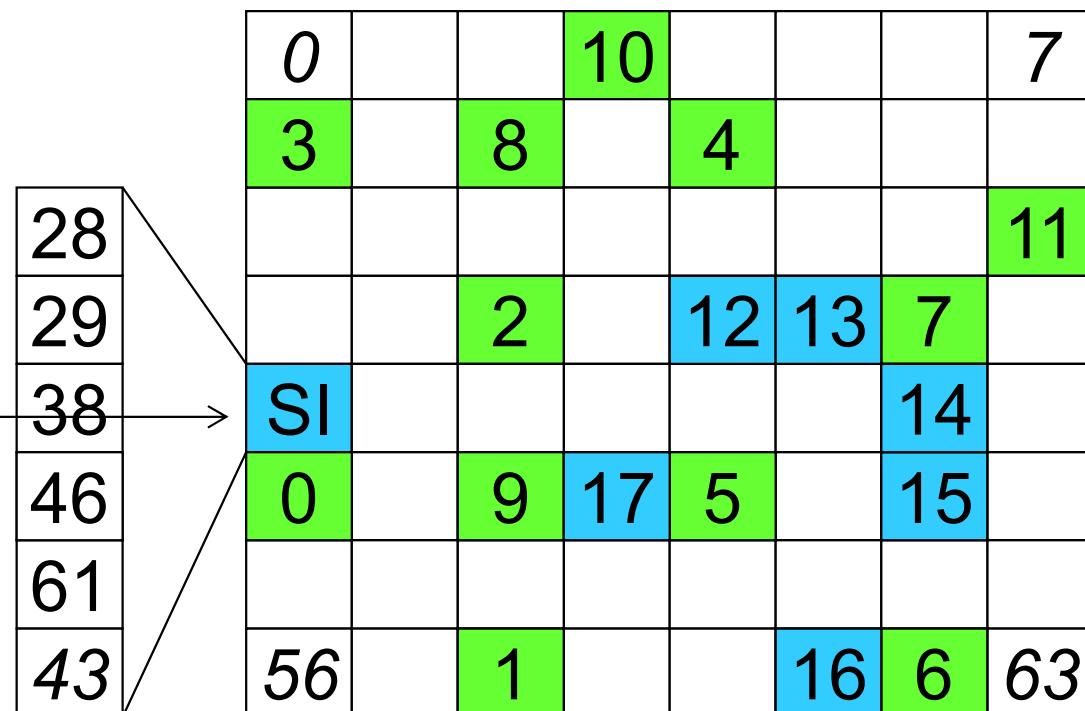
- How do we store files with data at offsets greater than 12 blocks?
  - Adding significantly more direct entries in the inode results in many unused entries most of the time.



mode	17
uid	16
gid	15
atime	14
ctime	13
mtime	12
size	11
block count	10
reference count	9
direct blocks (12)	8
40,58,26,8,12,	7
44,62,30,10,42,3,21	6
single indirect: 32	5
double indirect	4
triple indirect	3

# Inode Contents

- Single Indirect Block
  - Block number of a block containing block numbers



Disk

13



# Single Indirection

- Requires two disk access to read
  - One for the indirect block; one for the target block
- Max File Size
  - Assume 1Kbyte block size, 4 byte block numbers
$$12 * 1K + 1K/4 * 1K = 268 \text{ KiB}$$
- For large majority of files (< 268 KiB), given the inode, only one or two further accesses required to read any block in file.



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12) 40,58,26,8,12, 44,62,30,10,42,3,21
single indirect: 32
double indirect
triple indirect

# Inode Contents

- Double Indirect Block
  - Block number of a block containing block numbers of blocks containing block numbers



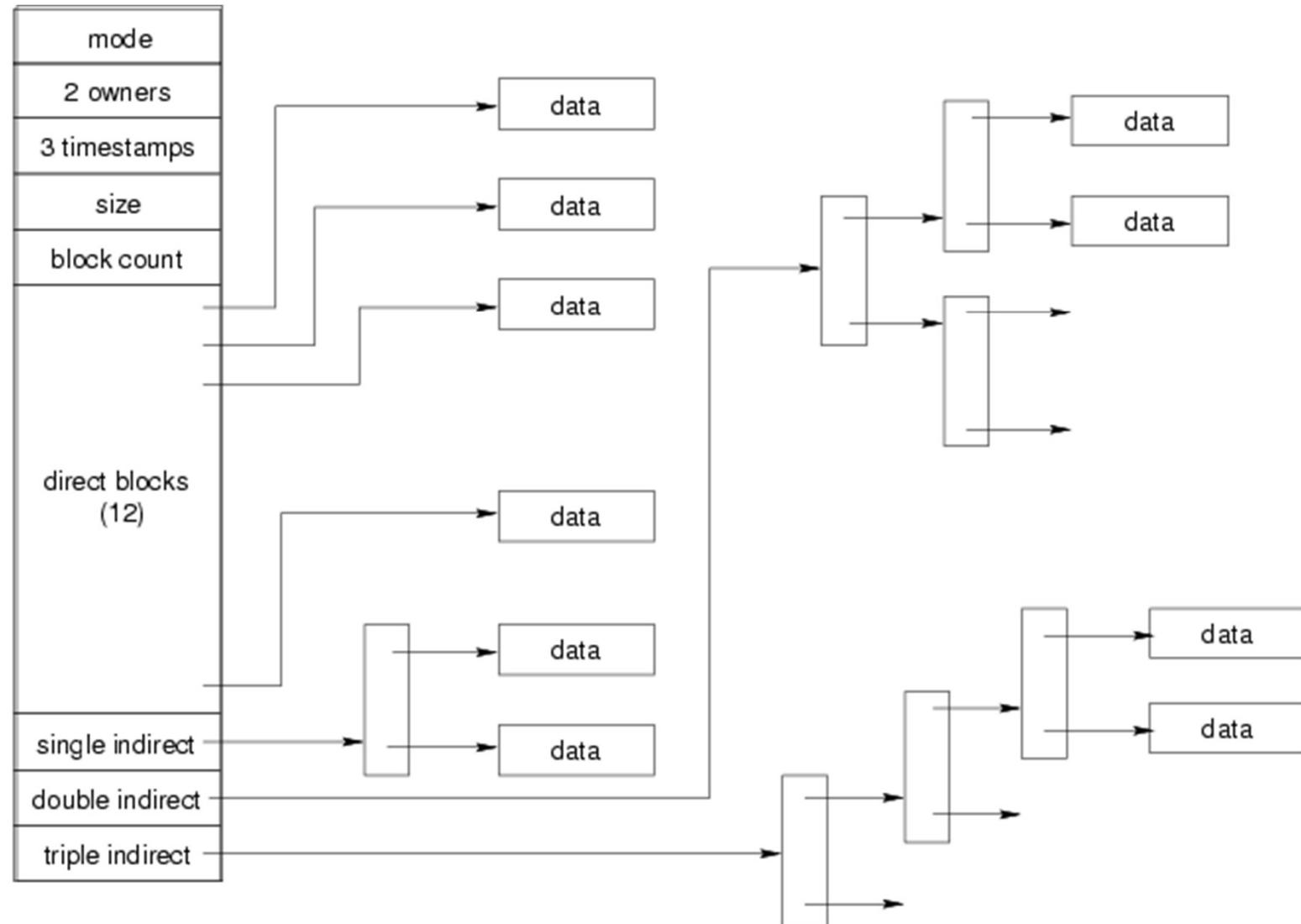
mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12) 40,58,26,8,12, 44,62,30,10,42,3,21
single indirect: 32
double indirect
triple indirect

# Inode Contents

- Double Indirect Block
  - Block number of a block containing block numbers of blocks containing block numbers
- Triple Indirect
  - Block number of a block containing block numbers of blocks containing block numbers of blocks containing block numbers ☺



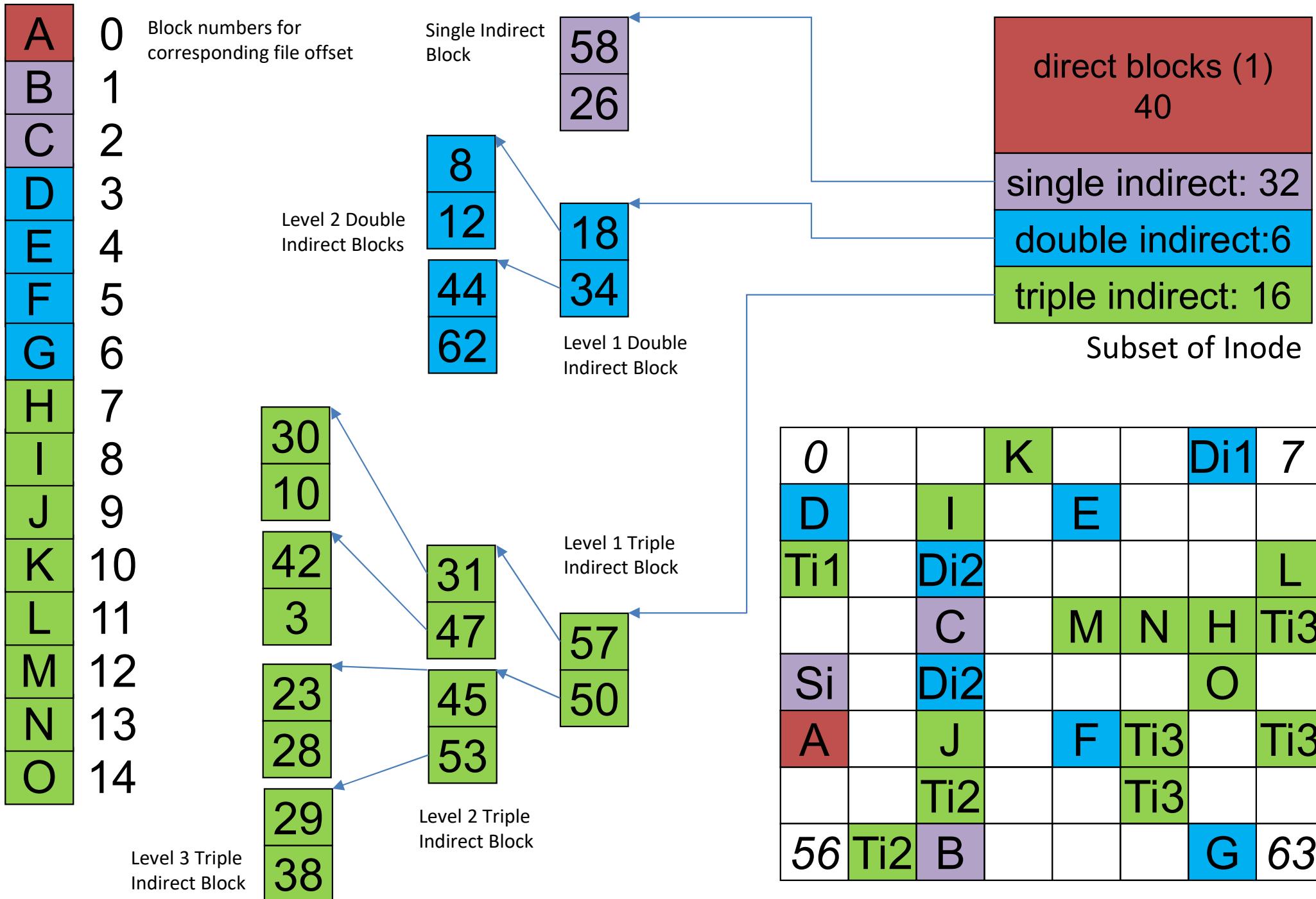
# UNIX Inode Block Addressing Scheme



# UNIX Inode Block Addressing Scheme

- Assume 8 byte blocks, containing 4 byte block numbers
- => each block can contain 2 block numbers (1-bit index)
- Assume a single direct block number in inode





# Max File Size

- Assume 4 bytes block numbers and 1K blocks
- The number of addressable blocks
  - Direct Blocks = 12
  - Single Indirect Blocks = 256
  - Double Indirect Blocks =  $256 * 256 = 65536$
  - Triple Indirect Blocks =  $256 * 256 * 256 = 16777216$
- Max File Size

$$12 + 256 + 65536 + 16777216 = 16843020 \text{ blocks} \approx 16 \text{ GB}$$

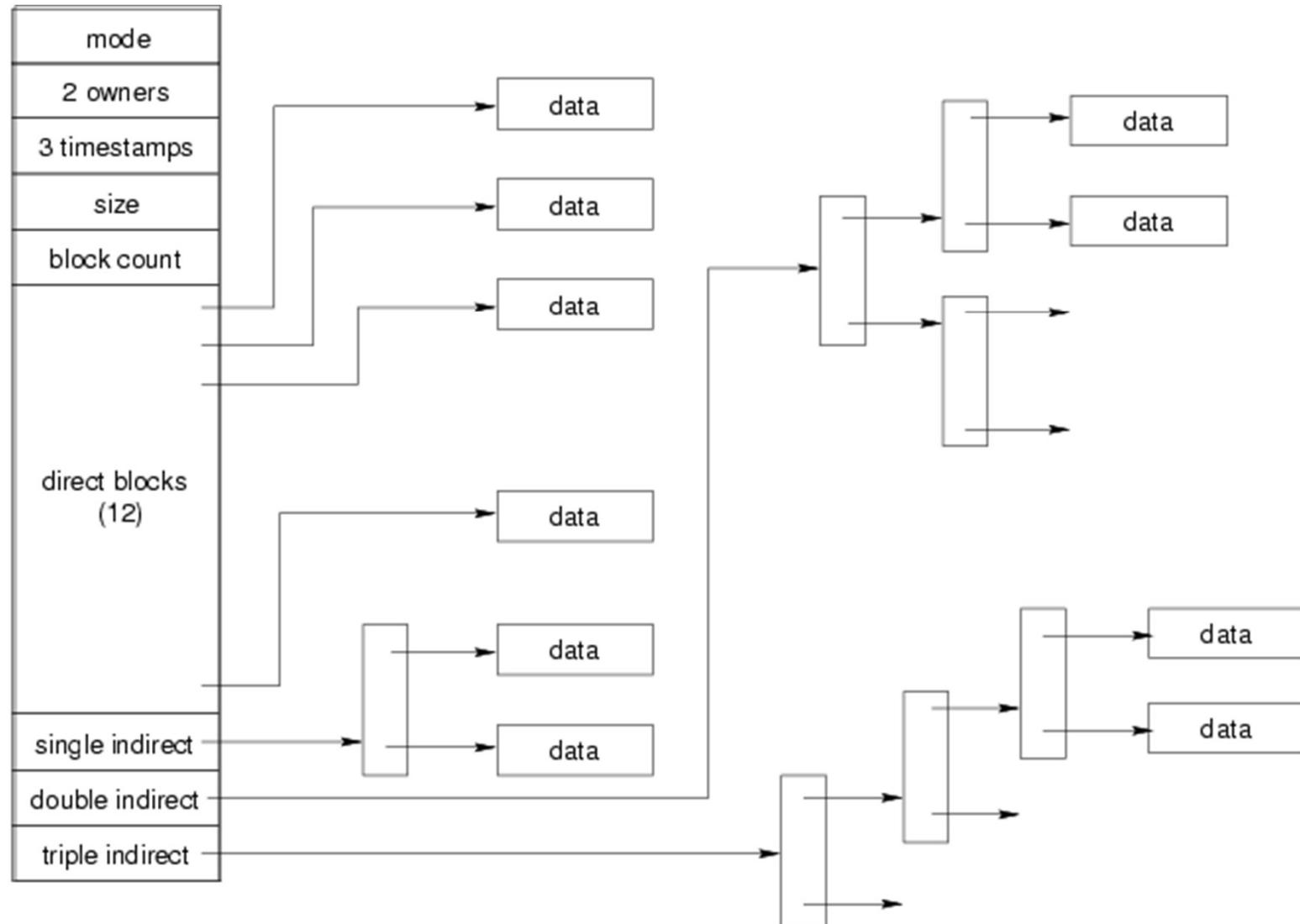


# Where is the data block number stored?

- Assume 4K blocks, 4 byte block numbers, 12 direct blocks
- A 1 byte file produced by
  - lseek(fd, 1048576, SEEK\_SET) /\* 1 megabyte \*/
  - write(fd, "x", 1)
- What if we add
  - lseek(fd, 5242880, SEEK\_SET) /\* 5 megabytes \*/
  - write(fd, "x", 1)



# Where is the block number in this tree?



# Solution?

4K blocks, 4 byte block numbers => 1024 block numbers in indirect blocks (10 bit index)

Block # range	location
0 --- 11	Direct blocks
12 --- 1035 (11 + 1024)	Single-indirect blocks
1036 --- 1049611 (1035 + 1024 * 1024)	Double-indirect blocks
1049612 --- ????	Triple-indirect blocks

File (not to scale)



# Solution

Address = 1048576 ==>

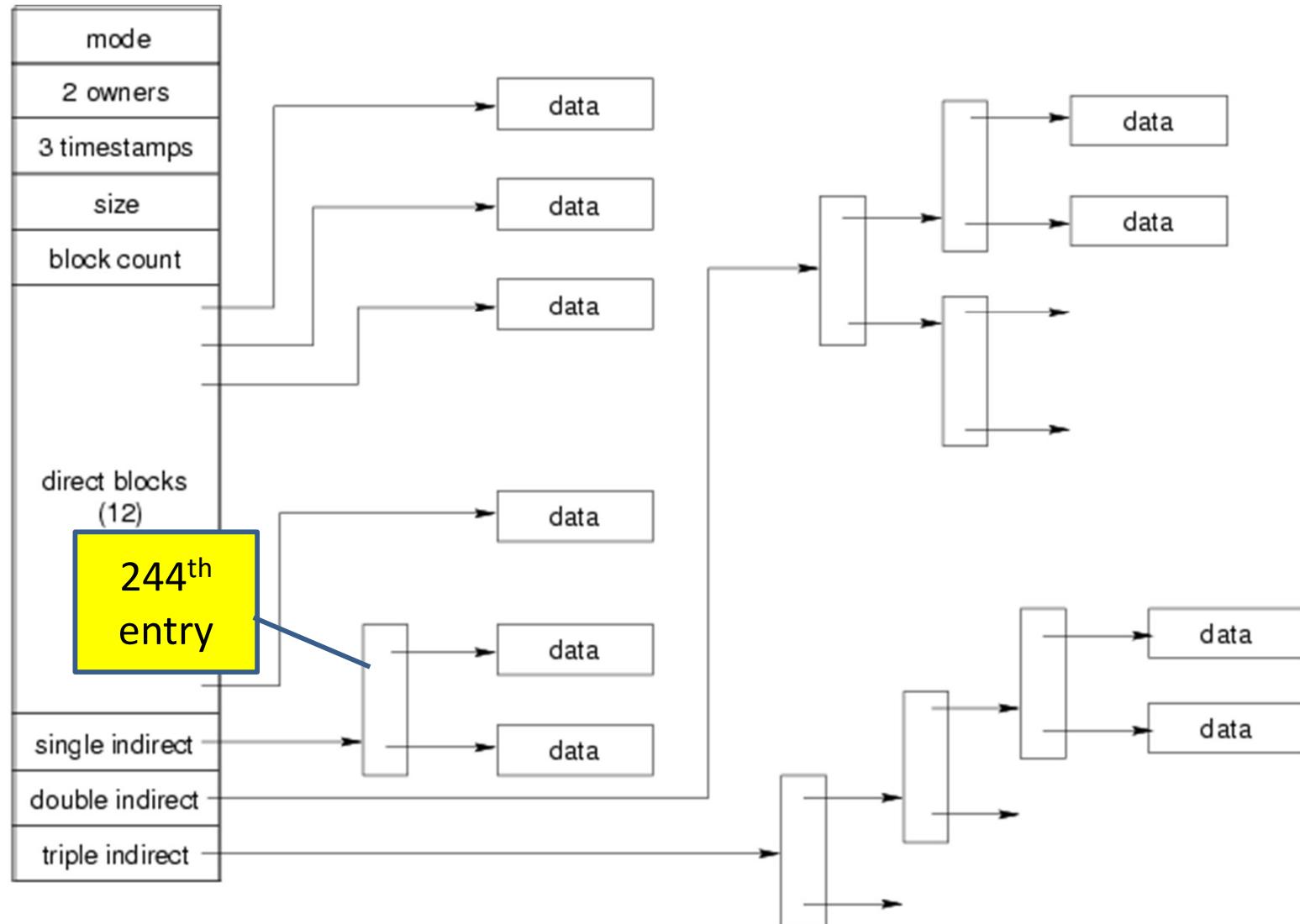
block number=1048576/4096=256

Single indirect offset = 256 – 12  
= 244

Block # range	location
0 ---11	Direct blocks
12 --- 1035	Single-indirect blocks
1036 --- 1049611	Double-indirect blocks
1049612 --- ????	Triple-indirect blocks



# Where is the block number in this tree?



# Solution

Address = 5242880 ==>

Block number =  $5242880/4096$   
= 1280

Double indirect offset (20-bit)  
=  $1280 - 1036$   
= 244

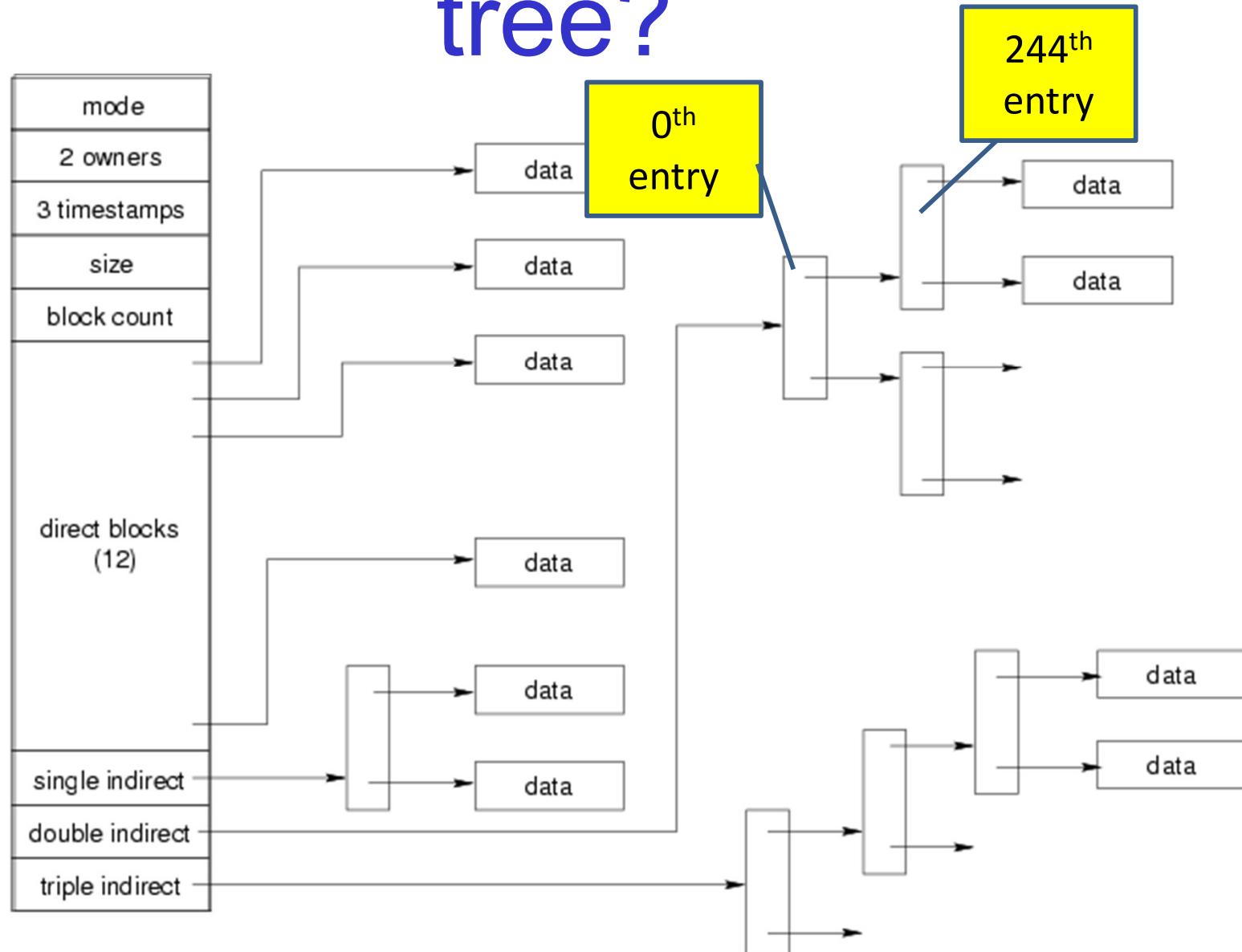
Top 10 bits = 0

Lower 10 bits = 244

Block # range	location
0 --- 11	Direct blocks
12 --- 1035	Single-indirect blocks
1036 --- 1049611	Double-indirect blocks
1049612 --- ????	Triple-indirect blocks



# Where is the block number in this tree?



# Some Best and Worst Case Access Patterns

Assume Inode already in memory

- To read 1 byte
  - Best:
    - 1 access via direct block
  - Worst:
    - 4 accesses via the triple indirect block
- To write 1 byte
  - Best:
    - 1 write via direct block (with no previous content)
  - Worst:
    - 4 reads (to get previous contents of block via triple indirect) + 1 write (to write modified block back)



# Worst Case Access Patterns with Unallocated Indirect Blocks

- Worst to write 1 byte
  - 4 writes (3 indirect blocks; 1 data)
  - 1 read, 4 writes (read-write 1 indirect, write 2; write 1 data)
  - 2 reads, 3 writes (read 1 indirect, read-write 1 indirect, write 1; write 1 data)
  - 3 reads, 2 writes (read 2, read-write 1; write 1 data)
- Worst to read 1 byte
  - If reading writes a zero-filled block on disk
  - Worst case is same as write 1 byte
  - If not, worst-case depends on how deep is the current indirect block tree.



# Inode Summary

- The inode (and indirect blocks) contains the on-disk metadata associated with a file
  - Contains mode, owner, and other bookkeeping
  - Efficient random and sequential access via *indexed allocation*
  - Small files (the majority of files) require only a single access
  - Larger files require progressively more disk accesses for random access
    - Sequential access is still efficient
  - Can support really large files via increasing levels of indirection



# Where/How are Inodes Stored



- System V Disk Layout (s5fs)
  - Boot Block
    - contain code to bootstrap the OS
  - Super Block
    - Contains attributes of the file system itself
    - e.g. size, number of inodes, start block of inode array, start of data block area, free inode list, free data block list
  - Inode Array
  - Data blocks



# Some problems with s5fs

- Inodes at start of disk; data blocks end
  - Long seek times
    - Must read inode before reading data blocks
- Only one superblock
  - Corrupt the superblock and entire file system is lost
- Block allocation was suboptimal
  - Consecutive free block list created at FS format time
    - Allocation and de-allocation eventually randomises the list resulting in random allocation
- Inode free list also randomised over time
  - Directory listing resulted in random inode access patterns

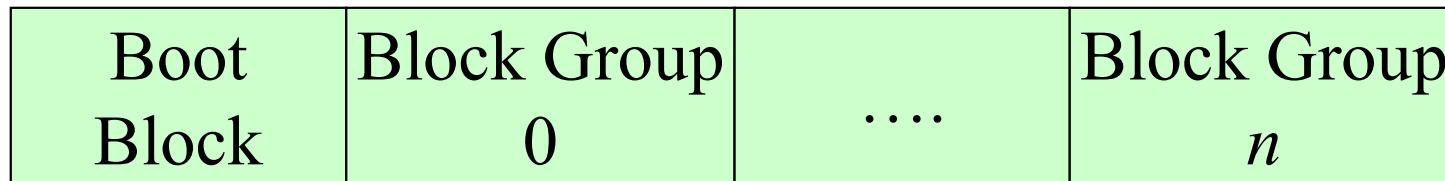


# Berkeley Fast Filesystem (FFS)

- Historically followed s5fs
  - Addressed many limitations with s5fs
  - ext2fs mostly similar



# Layout of an Ext2 FS



- Partition:
  - Reserved boot block,
  - Collection of equally sized *block groups*
  - All block groups have the same structure



# Layout of a Block Group

Super Block	Group Descriptors	Data Block Bitmap	Inode Bitmap	Inode Table	Data blocks
1 blk	$n$ blks	1 blk	1 blk	$m$ blks	$k$ blks

- *Replicated* super block
  - For e2fsck
- *Replicated* Group descriptors
- Bitmaps identify used inodes/blocks
- All block groups have the same number of data blocks
- Advantages of this structure:
  - Replication simplifies recovery
  - Proximity of inode tables and data blocks (reduces seek time)



# Superblocks

- Size of the file system, block size and similar parameters
- Overall free inode and block counters
- Data indicating whether file system check is needed:
  - Uncleanly unmounted
  - Inconsistency
  - Certain number of mounts since last check
  - Certain time expired since last check
- Replicated to provide redundancy to aid recoverability



# Group Descriptors

- Location of the bitmaps
- Counter for free blocks and inodes in this group
- Number of directories in the group
- Replicated to provide redundancy to aid recoverability



# Performance considerations

- EXT2 optimisations
  - Block groups cluster related inodes and data blocks
  - Pre-allocation of blocks on write (up to 8 blocks)
  - 8 bits in bit tables
  - Better contiguity when there are concurrent writes
  - Aim to store files within a directory in the same group



# Thus far...

- Inodes representing files laid out on disk.
- Inodes are referred to by number!!!
  - How do users name files? By number?



# Ext2fs Directories

inode	rec_len	name_len	type	name...
-------	---------	----------	------	---------

- Directories are files of a special type
  - Consider it a file of special format, managed by the kernel, that uses most of the same machinery to implement it
    - Inodes, etc...
- Directories translate names to inode numbers
- Directory entries are of variable length
- Entries can be deleted in place
  - inode = 0
  - Add to length of previous entry



# Ext2fs Directories

- “f1” = inode 7
- “file2” = inode 43
- “f3” = inode 85

7	Inode No
12	Rec Length
2	Name Length
‘f’ ‘1’ 0 0	Name
43	
16	
5	
‘f’ ‘i’ ‘l’ ‘e’	
‘2’ 0 0 0	
85	
12	
2	
‘f’ ‘3’ 0 0	
0	



# Hard links

- Note that inodes can have more than one name
  - Called a *Hard Link*
  - Inode (file) 7 has three names
    - “f1” = inode 7
    - “file2” = inode 7
    - “f3” = inode 7

7	Inode No
12	Rec Length
2	Name Length
‘f’ ‘1’ 0 0	Name
7	
16	
5	
‘f’ ‘i’ ‘l’ ‘e’	
‘2’ 0 0 0	
7	
12	
2	
‘f’ ‘3’ 0 0	
0	



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12) 40,58,26,8,12, 44,62,30,10,42,3,21
single indirect: 32
double indirect
triple indirect

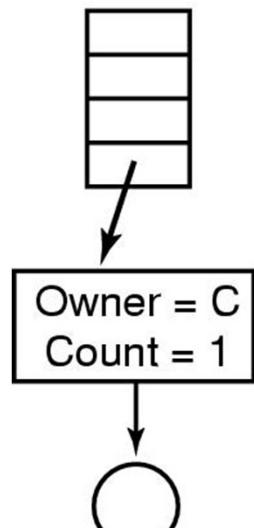
# Inode Contents

- We can have many names for the same inode.
- When we delete a file by name, i.e. remove the directory entry (link), how does the file system know when to delete the underlying inode?
  - Keep a *reference count* in the inode
- Adding a name (directory entry) increments the count
- Removing a name decrements the count
- If the reference count == 0, then we have no names for the inode (it is unreachable), we can delete the inode (underlying file or directory)



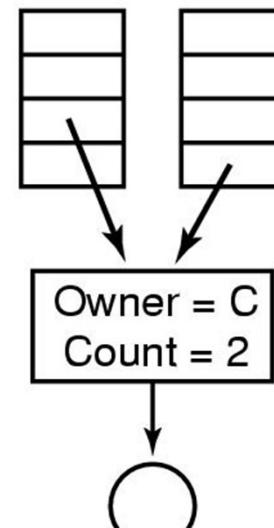
# Hard links

C's directory



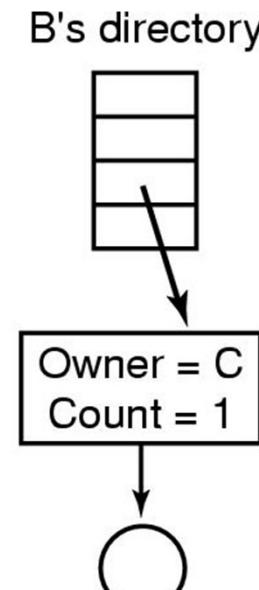
(a)

B's directory



(b)

C's directory



(c)

- (a) Situation prior to linking
- (b) After the link is created
- (c) After the original owner removes the file



# Symbolic links

- A symbolic link is a file that contains a reference to another file or directory
  - Has its own inode and data block, which contains a path to the target file
  - Marked by a special file attribute
  - Transparent for some operations
  - Can point across FS boundaries



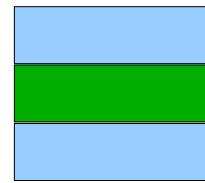
# FS reliability

- Disk writes are buffered in RAM
  - OS crash or power outage ==> lost data
  - Commit writes to disk periodically (e.g., every 30 sec)
  - Use the sync command to force a FS flush
- FS operations are non-atomic
  - Incomplete transaction can leave the FS in an inconsistent state

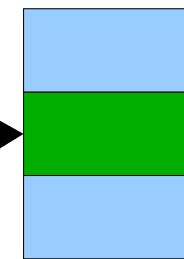


# FS reliability

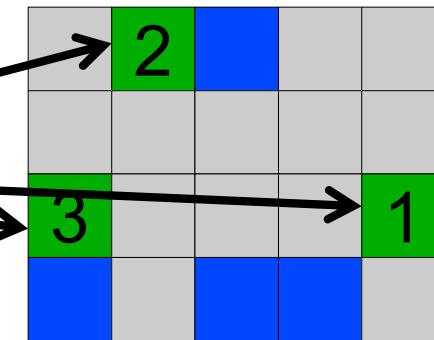
dir entries



i-nodes



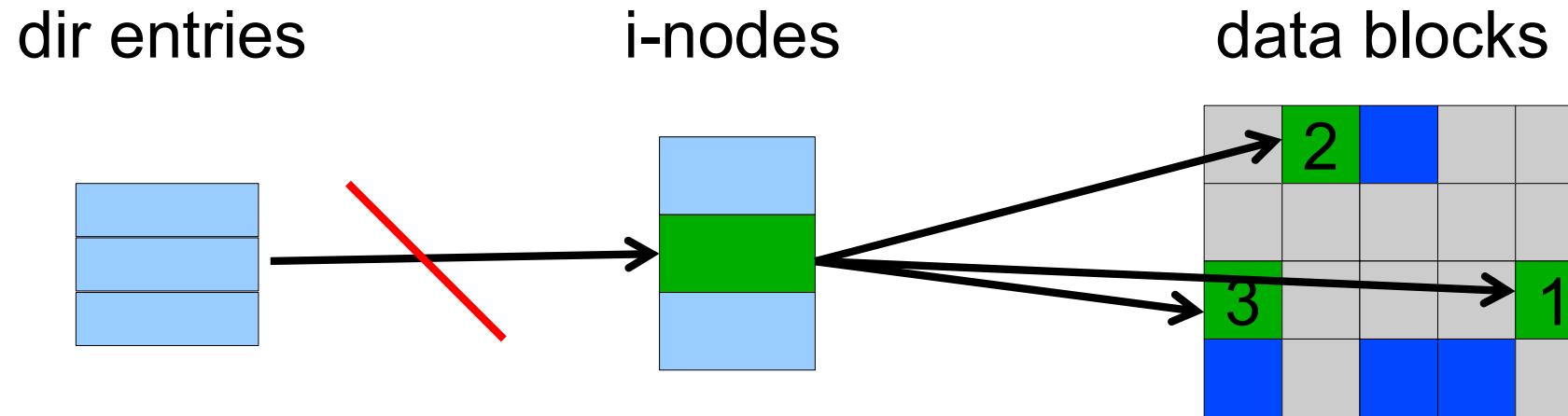
data blocks



- Example: deleting a file
  - 1. Remove the directory entry
  - 2. Mark the i-node as free
  - 3. Mark disk blocks as free



# FS reliability



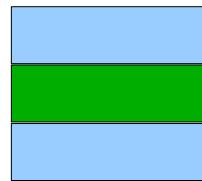
- Example: deleting a file
  - 1. Remove the directory entry **→ crash**
  - 2. Mark the i-node as free
  - 3. Mark disk blocks as free

The i-node and data blocks are lost

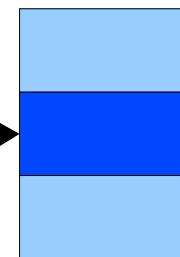


# FS reliability

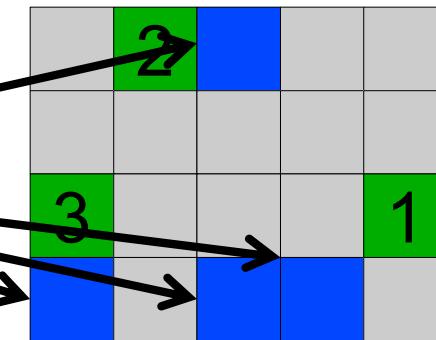
dir entries



i-nodes



data blocks



- Example: deleting a file
  1. Mark the i-node as free **--> crash**
  2. Remove the directory entry
  3. Mark disk blocks as free

The dir entry points to the wrong file

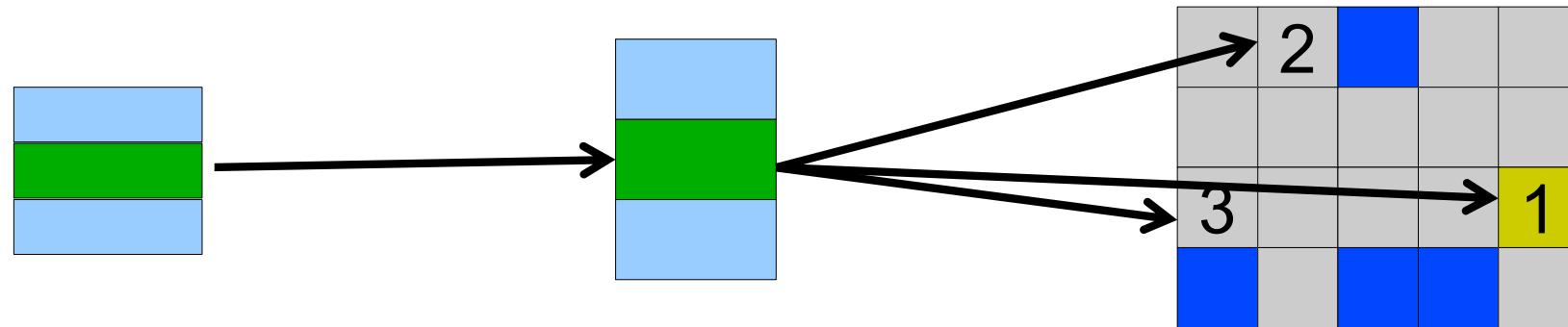


# FS reliability

dir entries

i-nodes

data blocks



- Example: deleting a file
  1. Mark disk blocks as free **--> crash**
  2. Remove the directory entry
  3. Mark the i-node as free

The file randomly shares disk blocks with other files



THE UNIVERSITY OF  
NEW SOUTH WALES

# FS reliability

- e2fsck
  - Scans the disk after an unclean shutdown and attempts to restore FS invariants
- Journaling file systems
  - Keep a journal of FS updates
  - Before performing an atomic update sequence,
  - write it to the journal
  - Replay the last journal entries upon an unclean shutdown
  - Example: ext3fs





THE UNIVERSITY OF  
NEW SOUTH WALES

# Case study: ext3 FS

# A brief intro to Journaling

# The ext3 file system

- Design goals
  - Add journaling capability to the ext2 FS
  - Backward and forward compatibility with ext2
    - Existing ext2 partitions can be mounted as ext3
  - Leverage the proven ext2 performance
  - Reuse most of the ext2 code base
  - Reuse ext2 tools, including e2fsck

# The ext3 journal

Option1: Journal FS data structure updates

- Example:
  - **Start transaction**
  - Delete dir entry
  - Delete i-node
  - Release blocks 32, 17, 60
  - **End transaction**

Option2: Journal disk block updates

- Example:
  - **Start transaction**
  - Update block #n1 (*contains the dir entry*)
  - Update block #n2 (*i-node allocation bitmap*)
  - Update block #n3 (*data block allocation bitmap*)
  - **Add transaction**

Question: which approach is better?

# The ext3 journal

## Option1: Journal FS data structure updates

- ✓ Efficient use of journal space; hence faster journaling
- ✗ Individual updates are applied separately
- ✗ The journaling layer must understand FS semantics

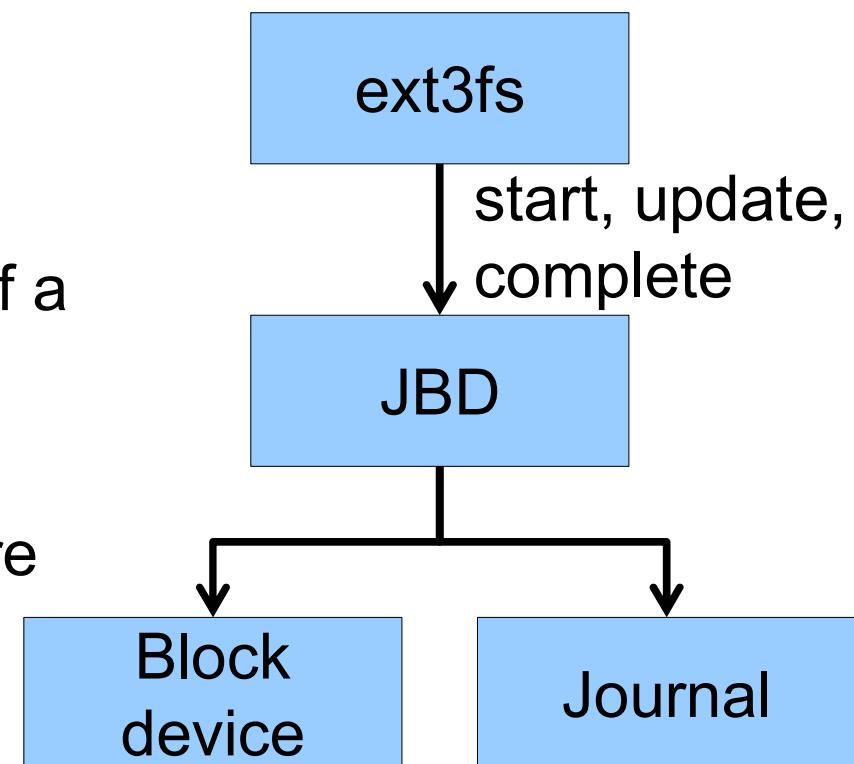
## Option2: Journal disk block updates

- ✗ Even a small update adds a whole block to the journal
- ✓ Multiple updates to the same block can be aggregated into a single update
- ✓ The journaling layer is FS-independent (easier to implement)

Ext3 implements Option 2

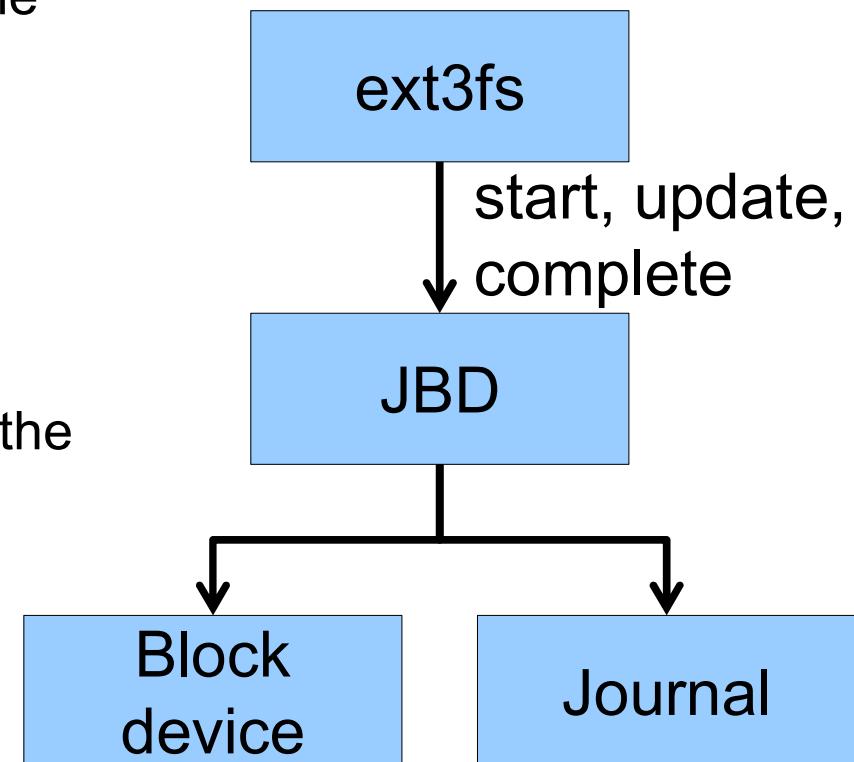
# Journaling Block Device (JBD)

- The ext3 journaling layer is called Journaling Block Device (JBD)
- JBD interface
  - Start a new transaction
  - Update a disk block as part of a transaction
  - Complete a transaction
    - Completed transactions are buffered in RAM

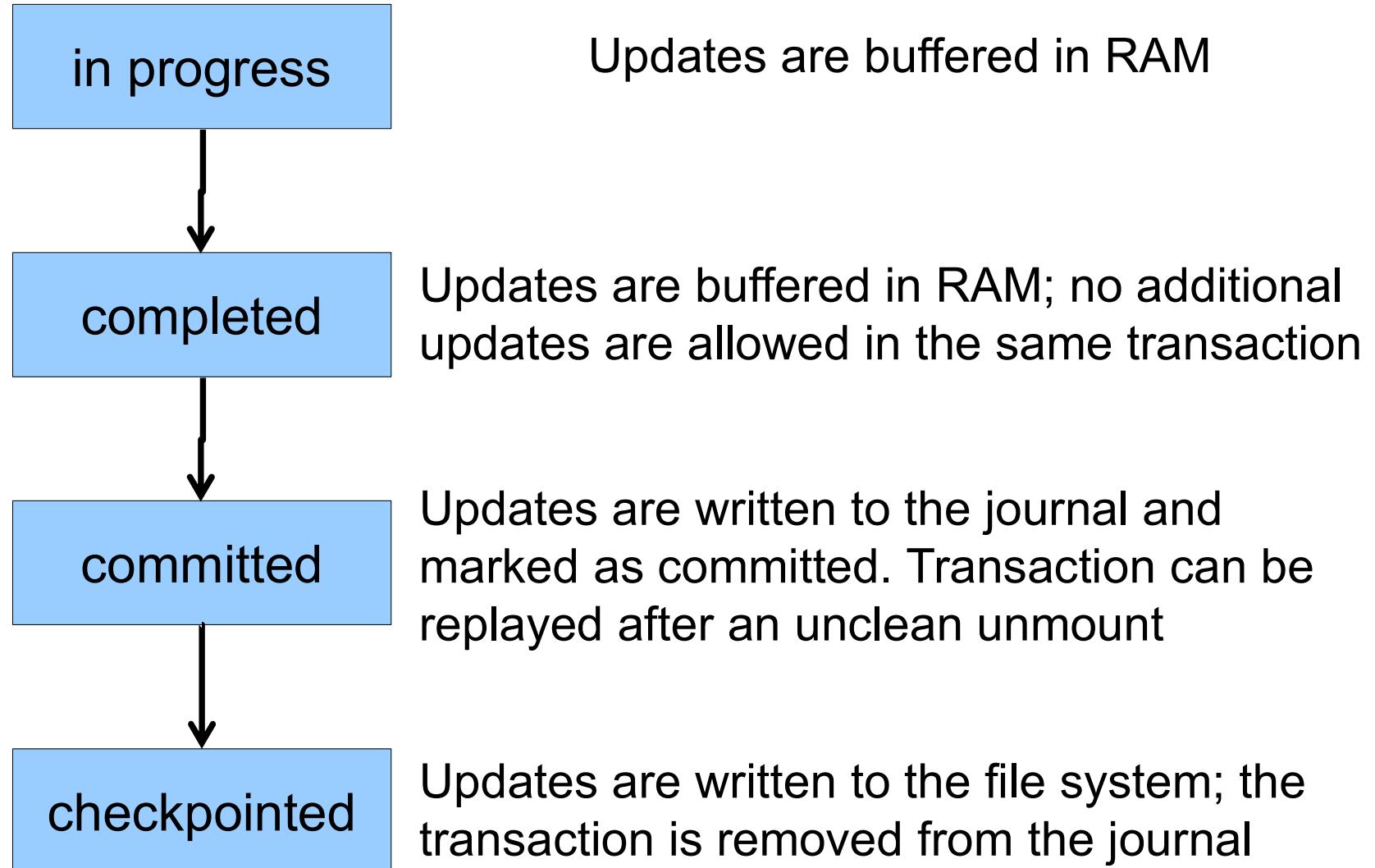


# Journaling Block Device (JBD)

- JBD interface (continued)
  - Commit: write transaction data to the journal (persistent storage)
    - Multiple FS transactions are committed in one go
  - Checkpoint: flush the journal to the disk
    - Used when the journal is full or the FS is being unmounted



# Transaction lifecycle



# Journaling modes

- Ext3 supports two journaling modes
  - Metadata+data
    - Enforces atomicity of all FS operations
  - Metadata journaling
    - Metadata is journalled
    - Data blocks are written directly to the disk
    - Improves performance
    - Enforces file system integrity
    - Does not enforce atomicity of `write's`
      - New file content can be stale blocks

# JBD

- JBD can keep the journal on a block device or in a file
  - Enables compatibility with ext2 (the journal is just a normal file)
- JBD is independent of ext3-specific data structures
  - Separation of concerns
    - The FS maintains on-disk data and metadata
    - JBD takes care of journaling
  - Code reuse
    - JBD can be used by any other FS that requires journaling

# Memory Management

# Learning Outcomes

- Appreciate the need for memory management in operating systems, understand the limits of fixed memory allocation schemes.
- Understand fragmentation in dynamic memory allocation, and understand basic dynamic allocation approaches.
- Understand how program memory addresses relate to physical memory addresses, memory management in base-limit machines, and swapping
- An overview of virtual memory management.

# Process

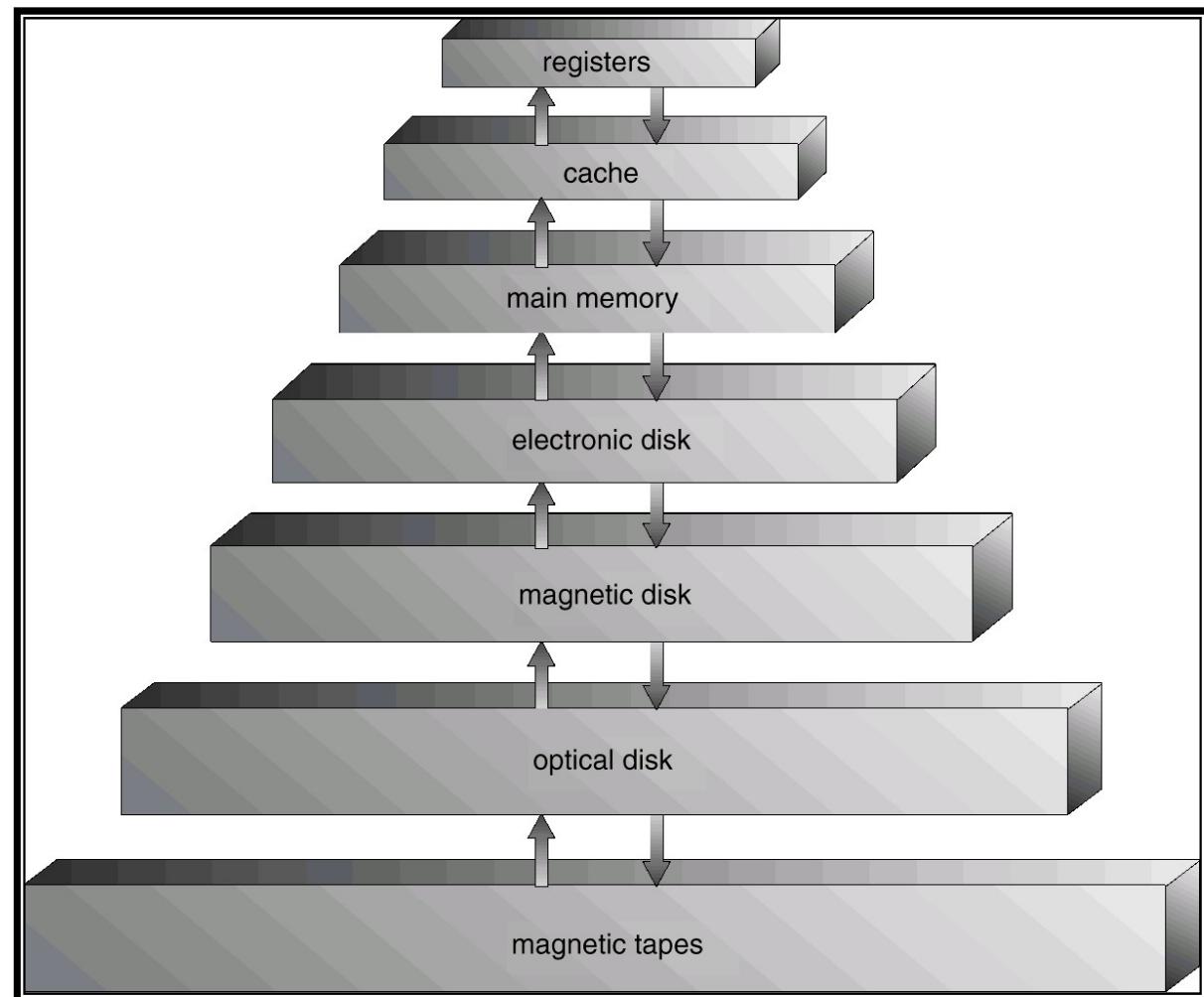
- One or more threads of execution
- Resources required for execution
  - Memory (RAM)
    - Program code (“text”)
    - Data (initialised, uninitialised, stack)
    - Buffers held in the kernel on behalf of the process
  - Others
    - CPU time
    - Files, disk space, printers, etc.

# OS Memory Management

- Keeps track of what memory is in use and what memory is free
- Allocates free memory to process when needed
  - And deallocates it when they don't
- Manages the transfer of memory between RAM and disk.

# Memory Hierarchy

- Ideally, programmers want memory that is
  - Fast
  - Large
  - Nonvolatile
- Not possible
- Memory management coordinates how memory hierarchy is used.
  - Focus usually on RAM ⇔ Disk

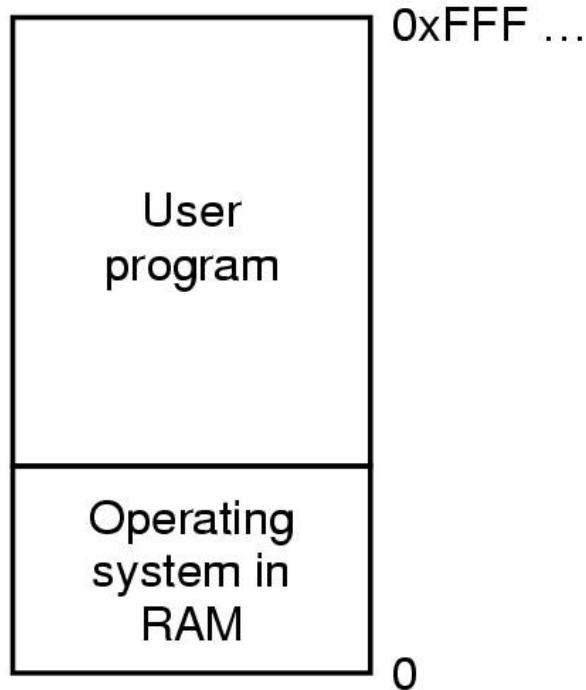


# OS Memory Management

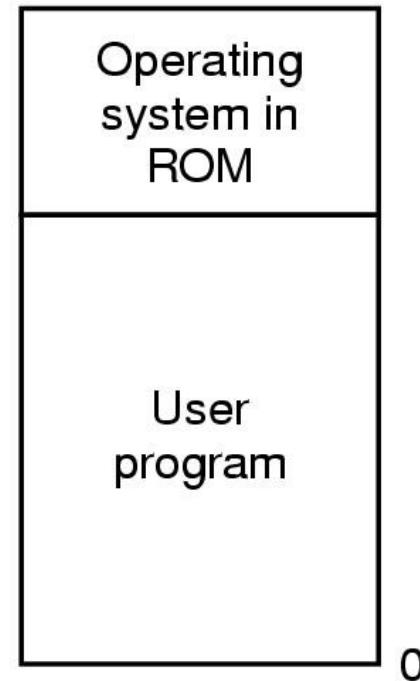
- Two broad classes of memory management systems
  - Those that transfer processes to and from external storage during execution.
    - Called swapping or paging
  - Those that don't
    - Simple
    - Might find this scheme in an embedded device, dumb phone, or smartcard.

# Basic Memory Management

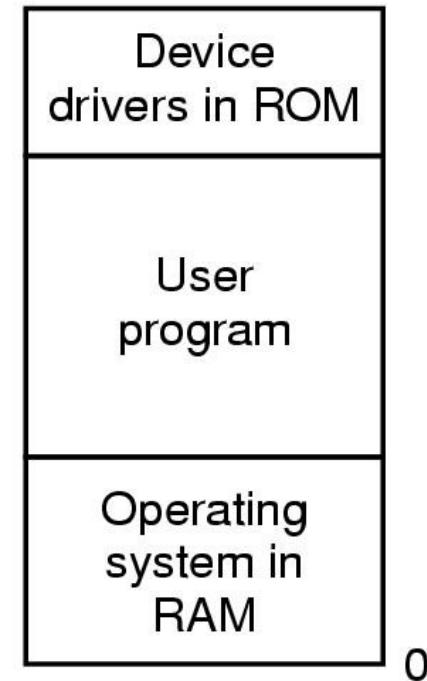
## Monoprogramming without Swapping or Paging



(a)



(b)



(c)

Three simple ways of organizing memory

- an operating system with one user process

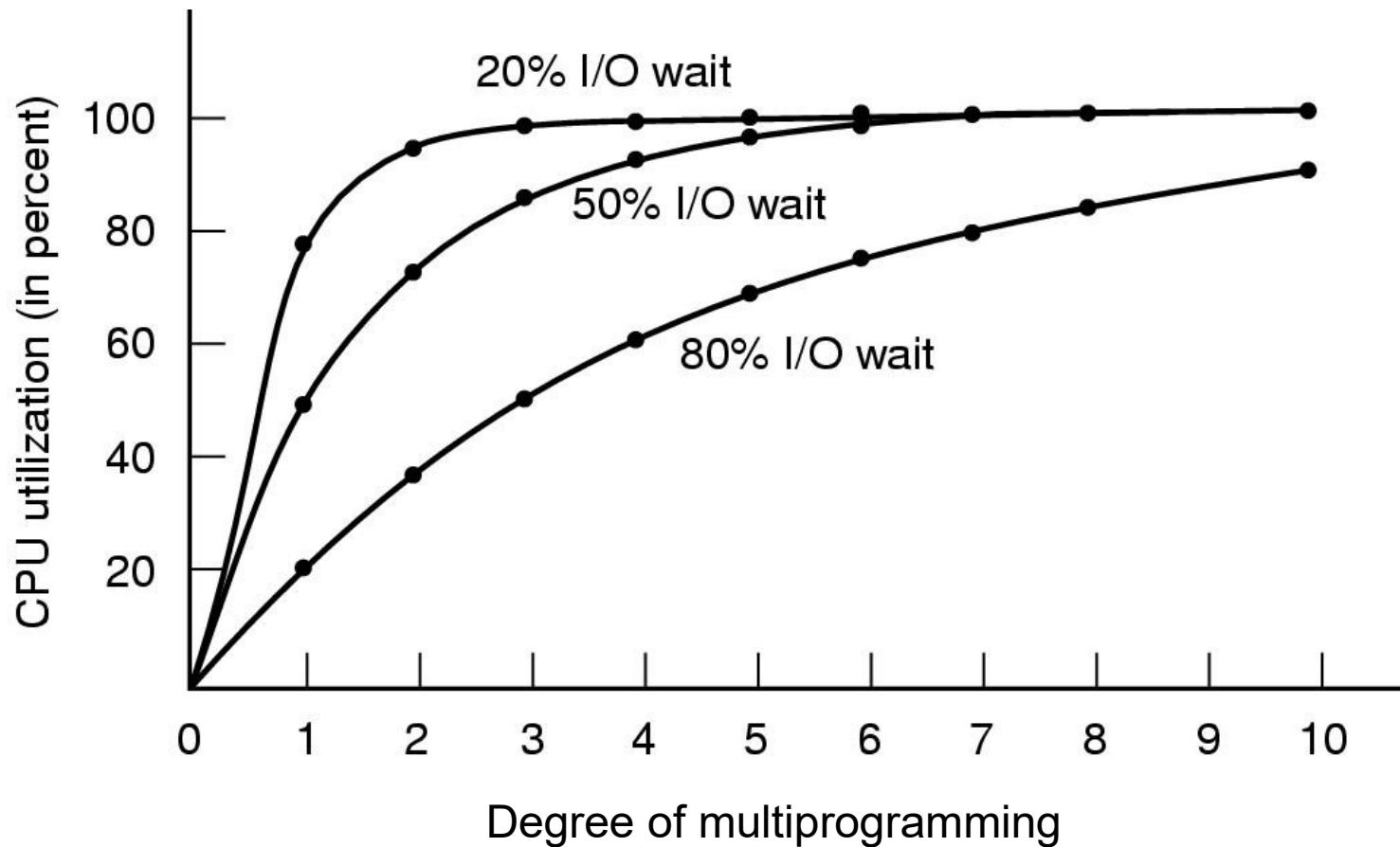
# Monoprogramming

- Okay if
  - Only have one thing to do
  - Memory available approximately equates to memory required
- Otherwise,
  - Poor CPU utilisation in the presence of I/O waiting
  - Poor memory utilisation with a varied job mix

# Idea

- Recall, an OS aims to
  - Maximise memory utilisation
  - Maximise CPU utilization
  - (ignore battery/power-management issues)
- Subdivide memory and run more than one process at once!!!!
  - Multiprogramming, Multitasking

# Modeling Multiprogramming



CPU utilization as a function of number of processes in memory

## Slide 10

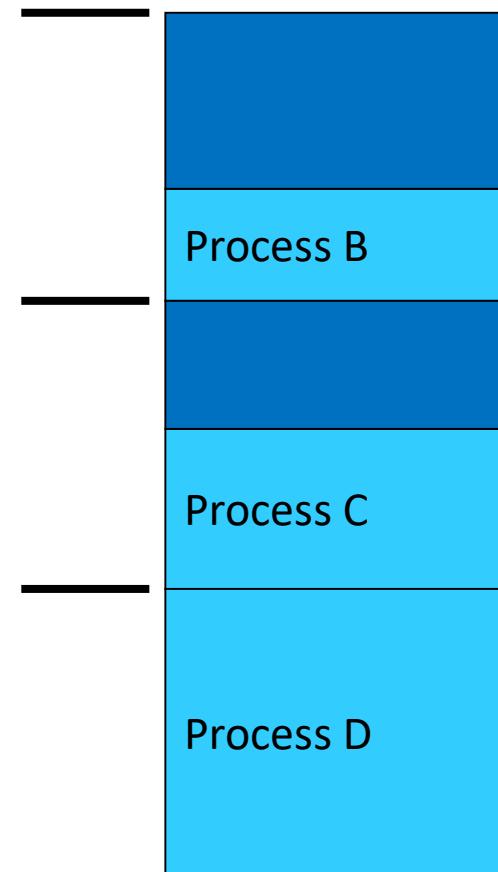
---

**KE1**

Kevin Elphinstone, 30/03/2020

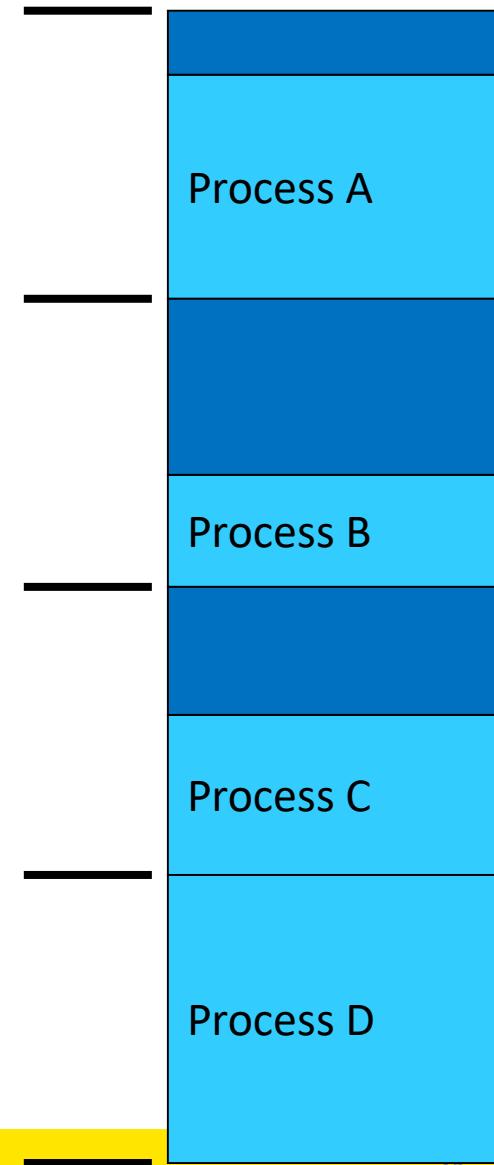
# General problem: How to divide memory between processes?

- Given a workload, how to we
  - Keep track of free memory?
  - Locate free memory for a new process?
- Overview of evolution of simple memory management
  - Static (fixed partitioning) approaches
    - Simple, predictable workloads of early computing
  - Dynamic (partitioning) approaches
    - More flexible computing as compute power and complexity increased.
- Introduce virtual memory
  - Segmentation and paging



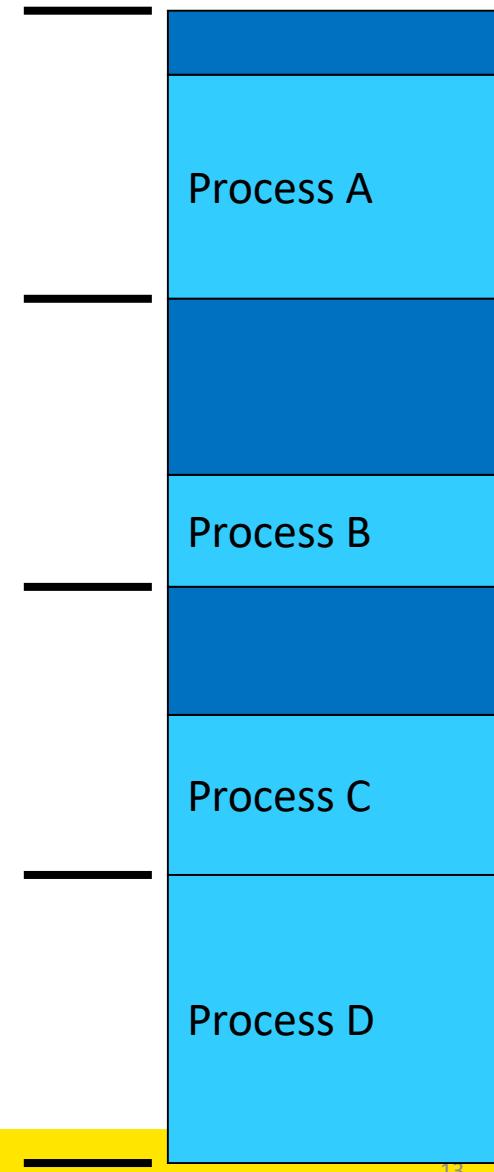
# Problem: How to divide memory

- One approach
  - divide memory into fixed equal-sized partitions
  - Any process  $\leq$  partition size can be loaded into any partition
  - Partitions are free or busy



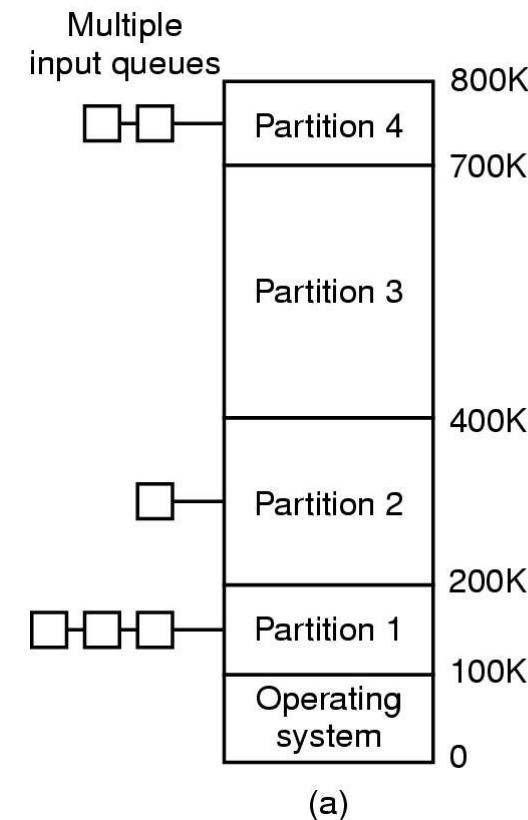
# Simple MM: Fixed, equal-sized partitions

- Any unused space in the partition is wasted
  - Called internal fragmentation
- Processes smaller than main memory, but larger than a partition cannot run.



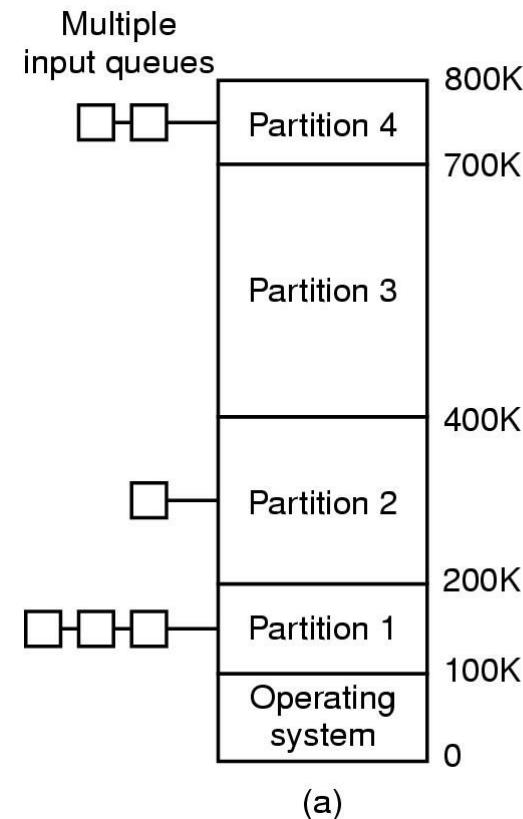
# Simple MM: Fixed, variable-sized partitions

- Divide memory at boot time into a selection of different sized partitions
  - Can base sizes on expected workload
- Each partition has queue:
  - Place process in queue for smallest partition that it fits in.
  - Processes wait for when assigned partition is empty to start



- Issue

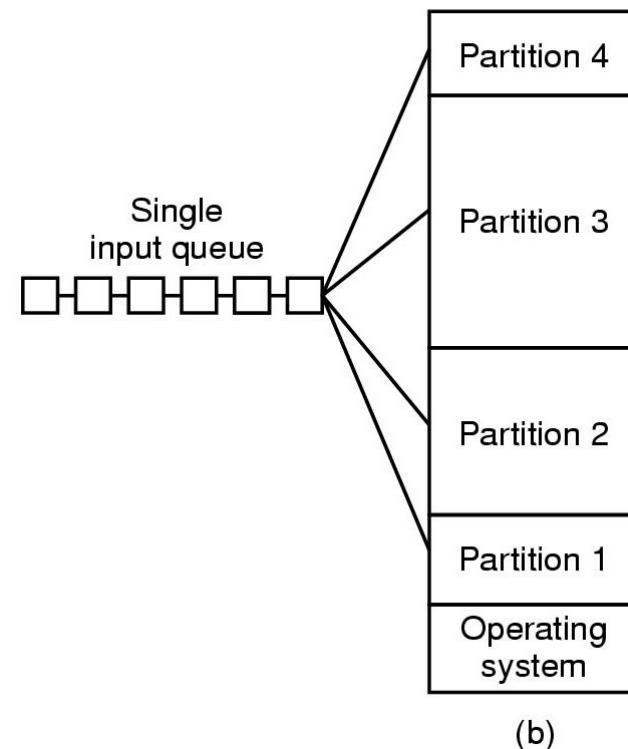
- Some partitions may be idle
  - Small jobs available, but only large partition free
  - Workload could be unpredictable



(a)

# Alternative queue strategy

- Single queue, search for any jobs that fit
  - Small jobs in large partition if necessary
  - Increases internal memory fragmentation

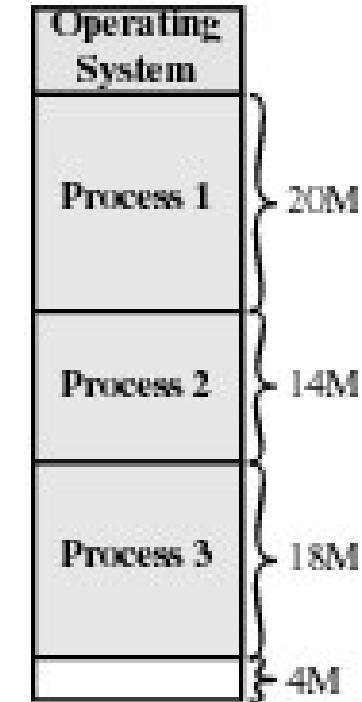
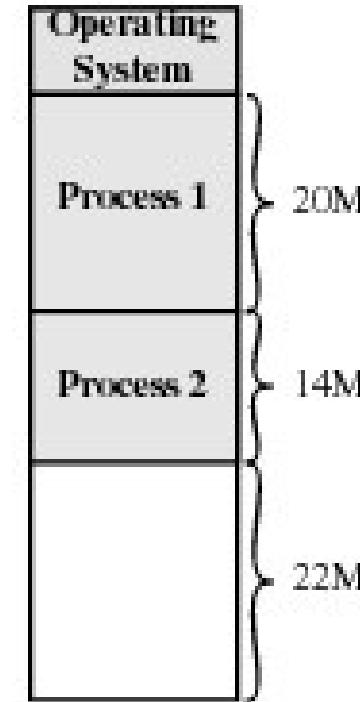
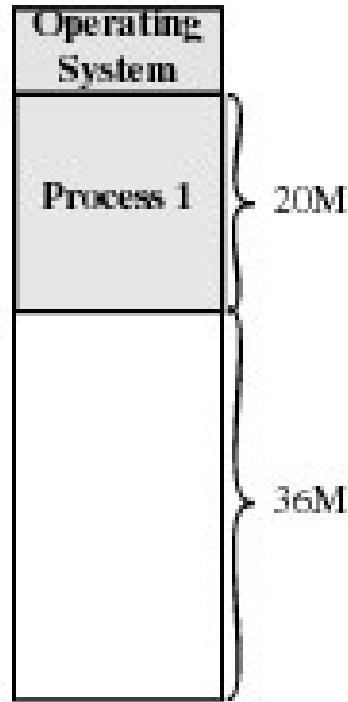
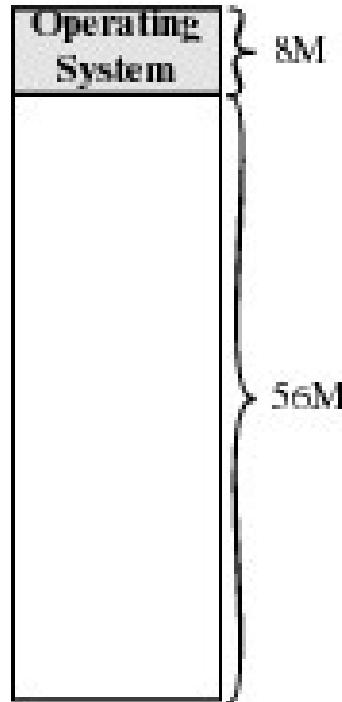


# Fixed Partition Summary

- Simple
- Easy to implement
- Can result in poor memory utilisation
  - Due to internal fragmentation
- Used on IBM System 360 operating system (OS/MFT)
  - Announced 6 April, 1964
- Still applicable for simple embedded systems
  - Static workload known in advance

# Dynamic Partitioning

- Partitions are of variable length
  - Allocated on-demand from ranges of free memory
- Process is allocated exactly what it needs
  - Assumes a process knows what it needs



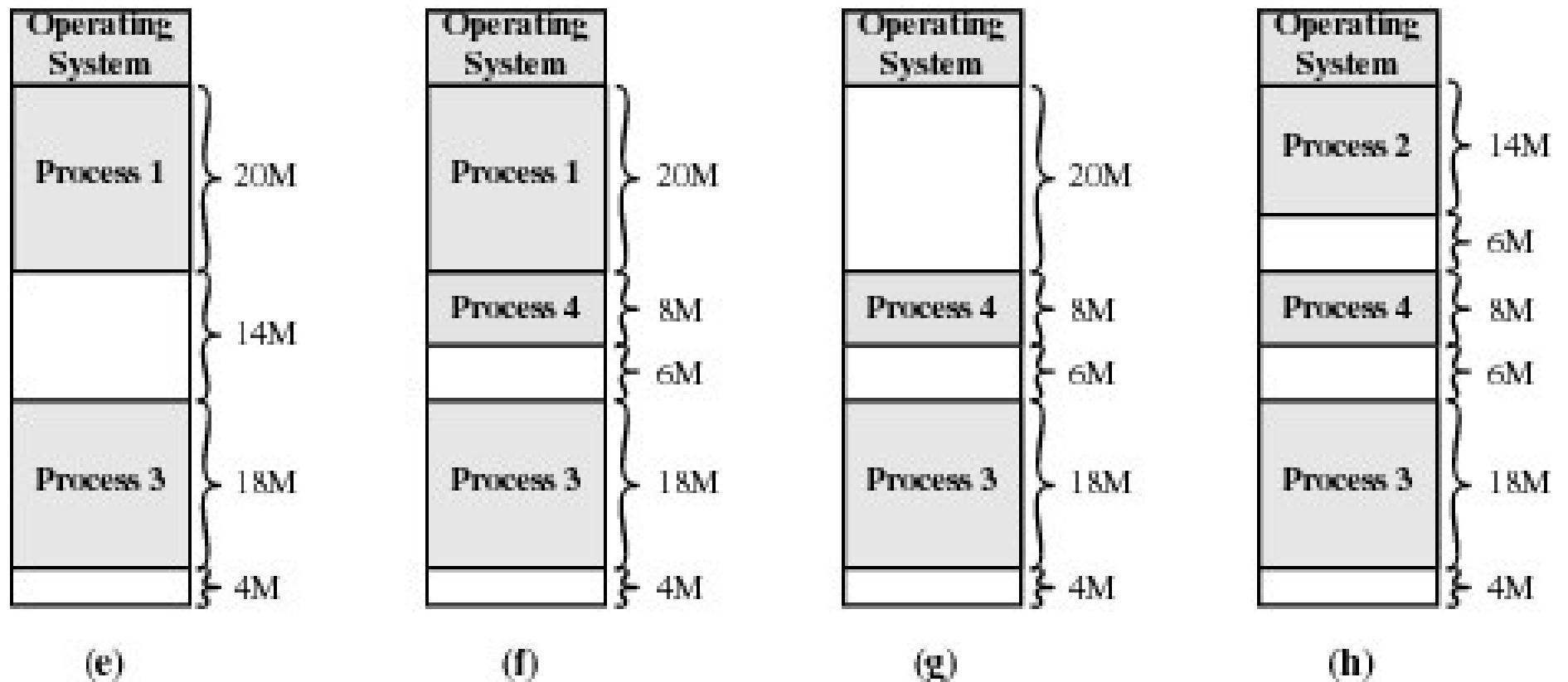
(a)

(b)

(c)

(d)

**Figure 7.4 The Effect of Dynamic Partitioning**



**Figure 7.4 The Effect of Dynamic Partitioning**

# Dynamic Partitioning

- In previous diagram
  - We have 16 meg free in total, but it can't be used to run any more processes requiring > 6 meg as it is fragmented
    - Called *external fragmentation*
  - We end up with unusable holes

# Recap: Fragmentation

- **External Fragmentation:**

- The space wasted external to the allocated memory regions.
- Memory space exists to satisfy a request, but it is unusable as it is not contiguous.

- **Internal Fragmentation:**

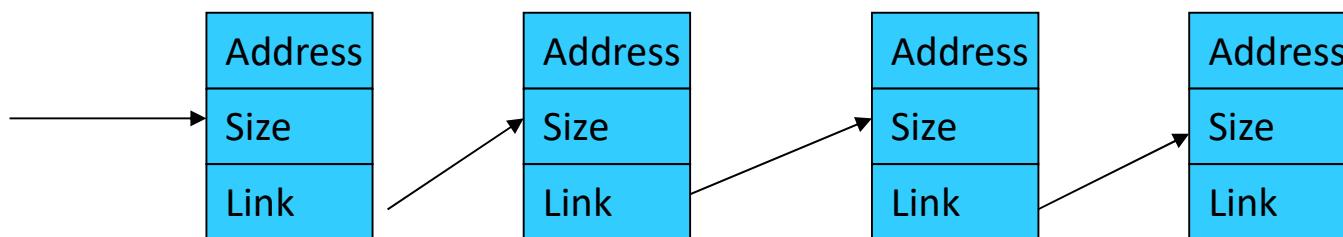
- The space wasted internal to the allocated memory regions.
- allocated memory may be slightly larger than requested memory; this size difference is wasted memory internal to a partition.

# Dynamic Partition Allocation Algorithms

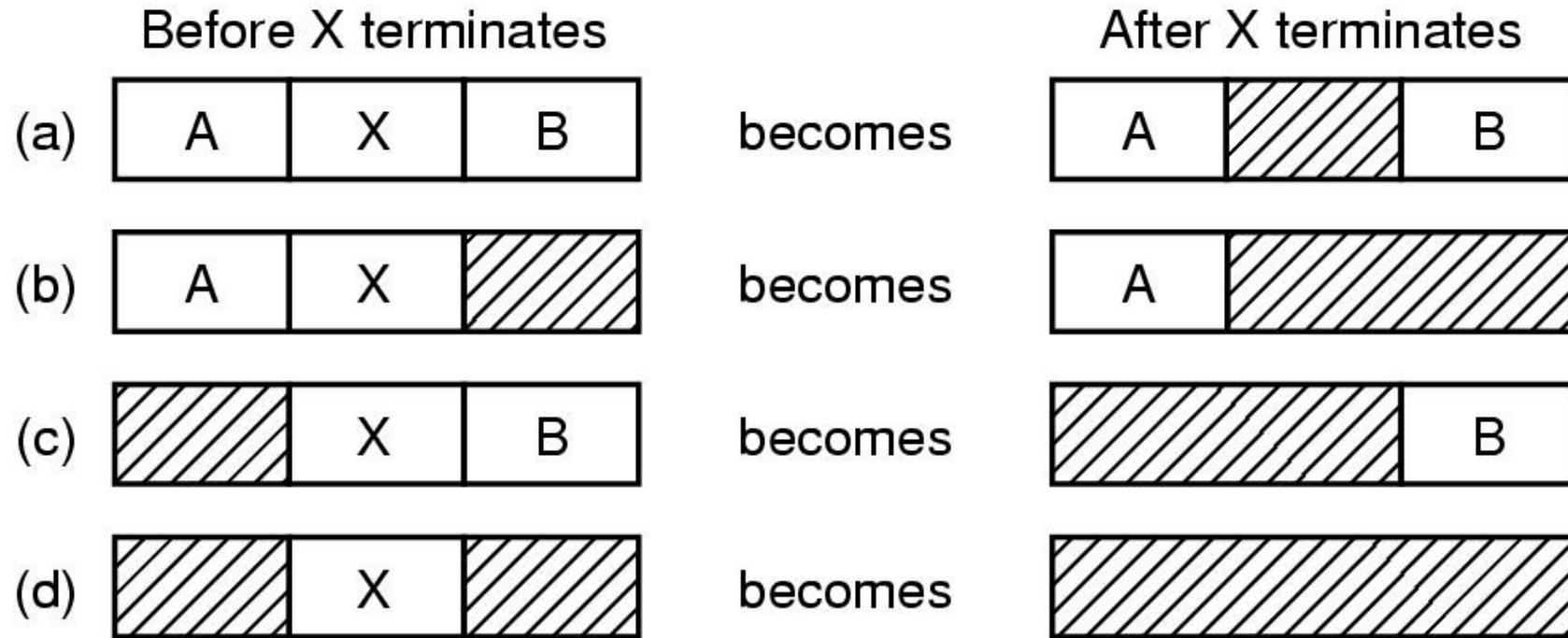
- Also applicable to `malloc()`-like in-application allocators
- Given a region of memory, basic requirements are:
  - Quickly locate a free partition satisfying the request
    - Minimise CPU time search
  - Minimise external fragmentation
  - Minimise memory overhead of bookkeeping
  - Efficiently support merging two adjacent free partitions into a larger partition

# Classic Approach

- Represent available memory as a linked list of available “holes” (free memory ranges).
  - Base, size
  - Kept in order of increasing address
    - Simplifies merging of adjacent holes into larger holes.
  - List nodes be stored in the “holes” themselves



# Coalescing Free Partitions with Linked Lists

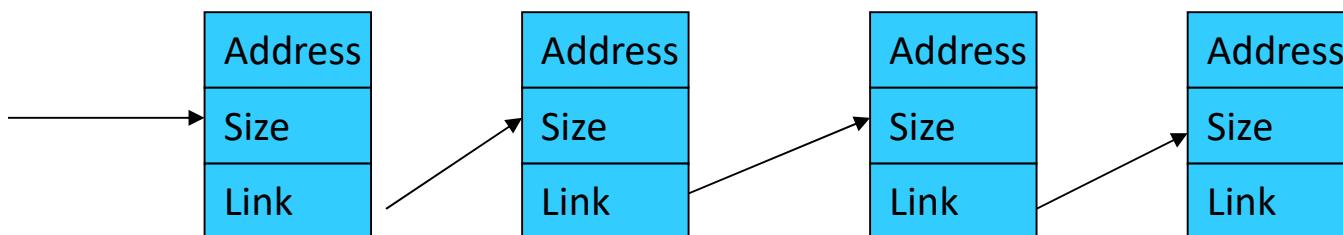


Four neighbor combinations for the terminating process X

# Dynamic Partitioning Placement Algorithm

- First-fit algorithm

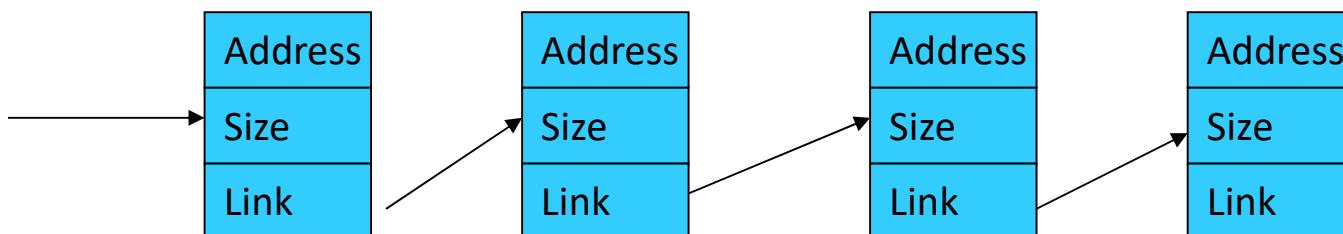
- Scan the list for the first entry that fits
  - If greater in size, break it into an allocated and free part
  - Intent: Minimise amount of searching performed
- Aims to find a match quickly
- Biases allocation to one end of memory
- Tends to preserve larger blocks at the end of memory



# Dynamic Partitioning Placement Algorithm

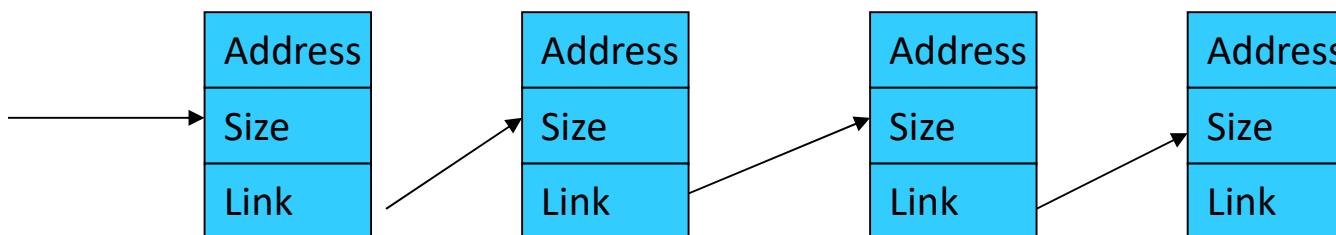
- Next-fit

- Like first-fit, except it begins its search from the point in list where the last request succeeded instead of at the beginning.
  - (Flawed) Intuition: spread allocation more uniformly over entire memory to avoid skipping over small holes at start of memory
  - Performs worse than first-fit as it breaks up the large free space at end of memory.



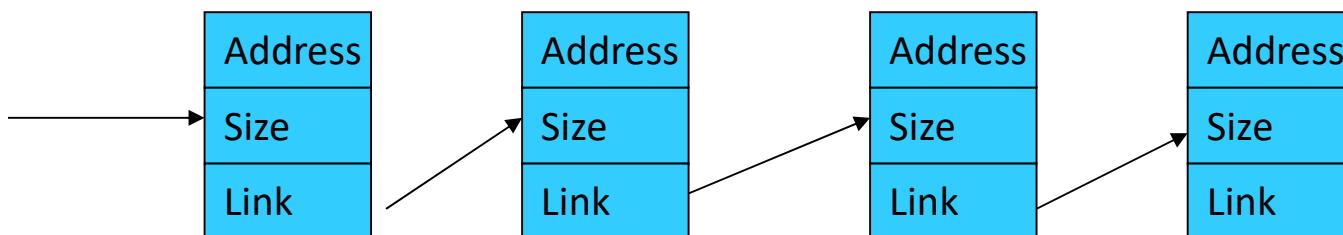
# Dynamic Partitioning Placement Algorithm

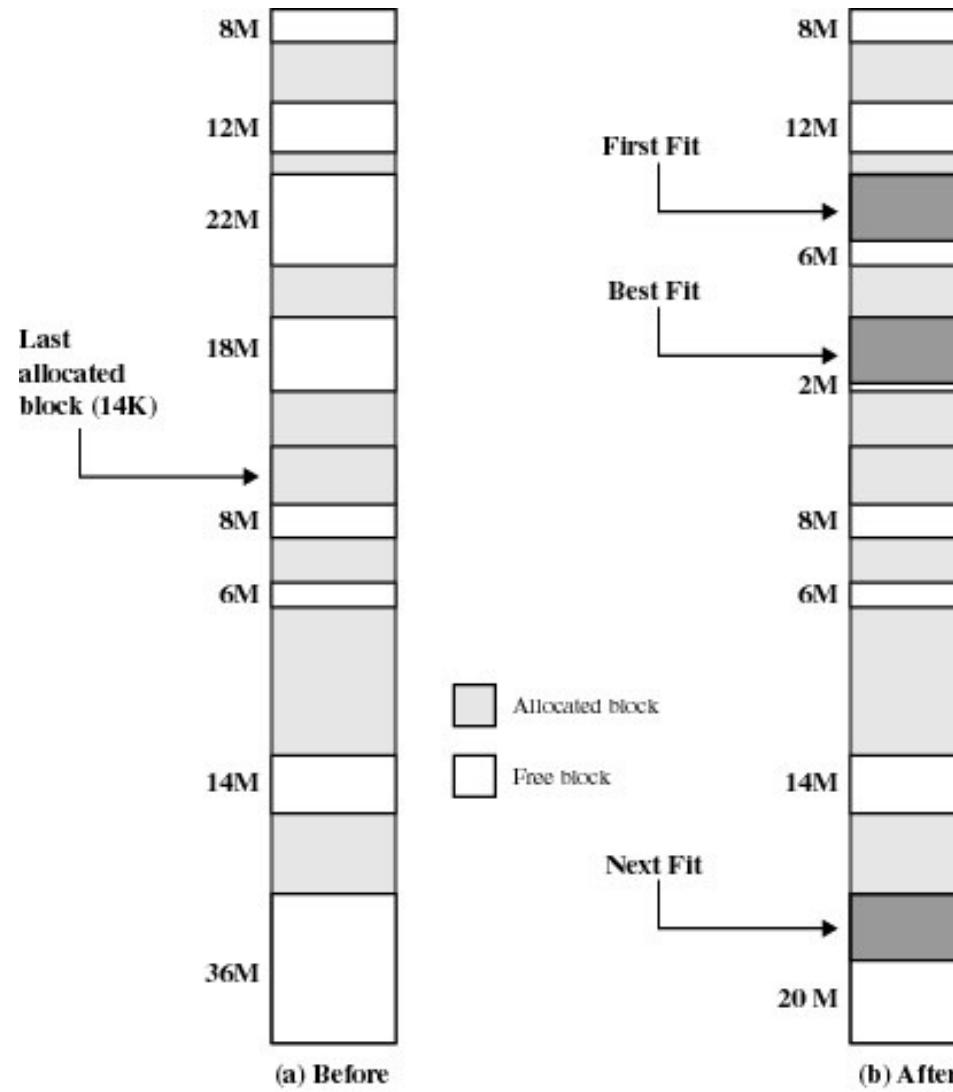
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Performs worse than first-fit
    - Has to search complete list
      - does more work than first-fit
      - Since smallest block is chosen for a process, the smallest amount of external fragmentation is left
      - Create lots of unusable holes



# Dynamic Partitioning Placement Algorithm

- Worst-fit algorithm
  - Chooses the block that is largest in size (worst-fit)
    - (whimsical) idea is to leave a usable fragment left over
  - Poor performer
    - Has to do more work (like best fit) to search complete list
    - Does not result in significantly less fragmentation





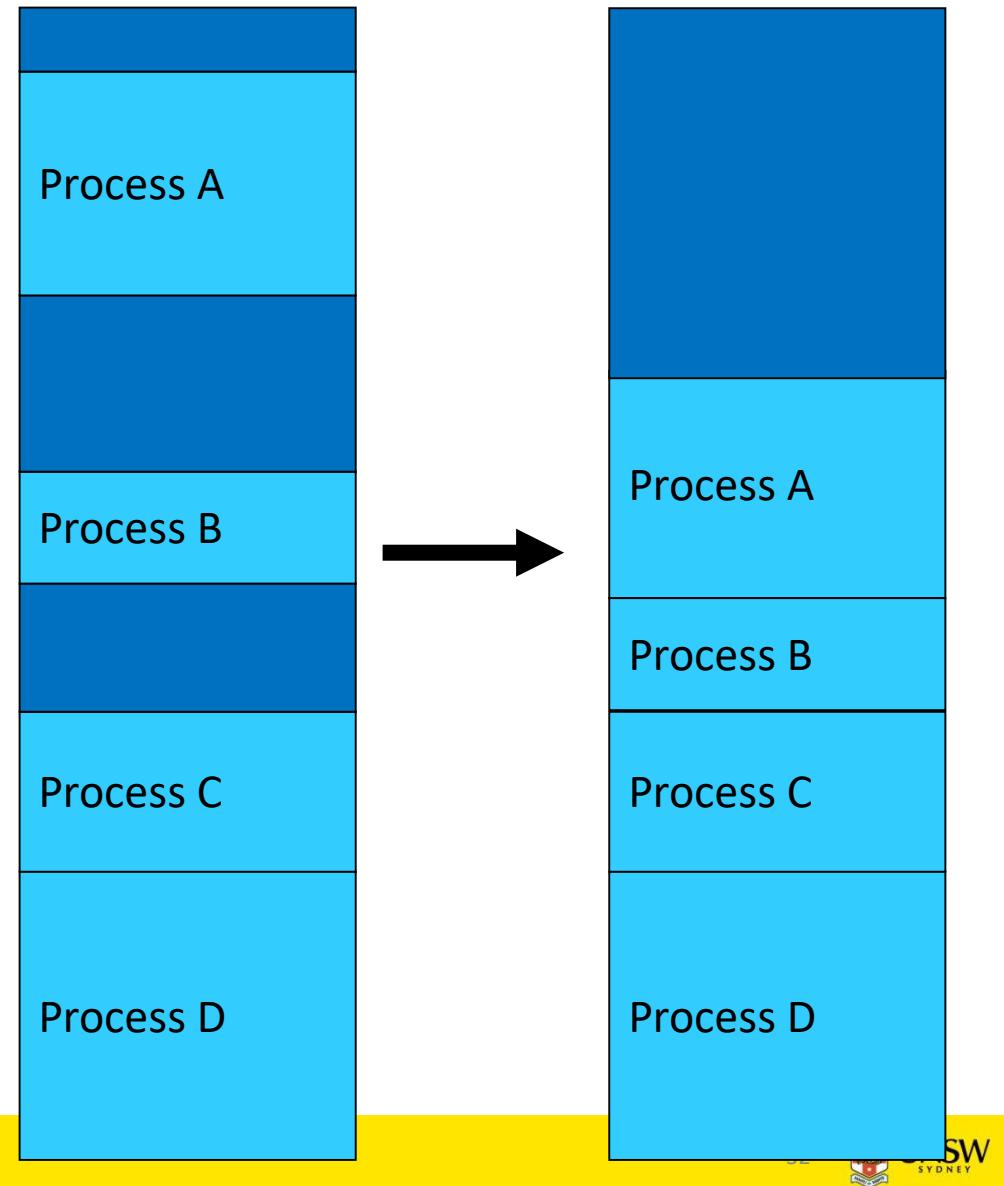
**Figure 7.5 Example Memory Configuration Before and After Allocation of 16 Mbyte Block**

# Dynamic Partition Allocation Algorithm

- Summary
  - First-fit generally better than the others and easiest to implement
- You should be aware of them
  - They are simple solutions to a still-existing OS or application service/function – memory allocation.
- Note: Largely have been superseded by more complex and specific allocation strategies
  - Typical in-kernel allocators used are *lazy buddy*, and *slab* allocators

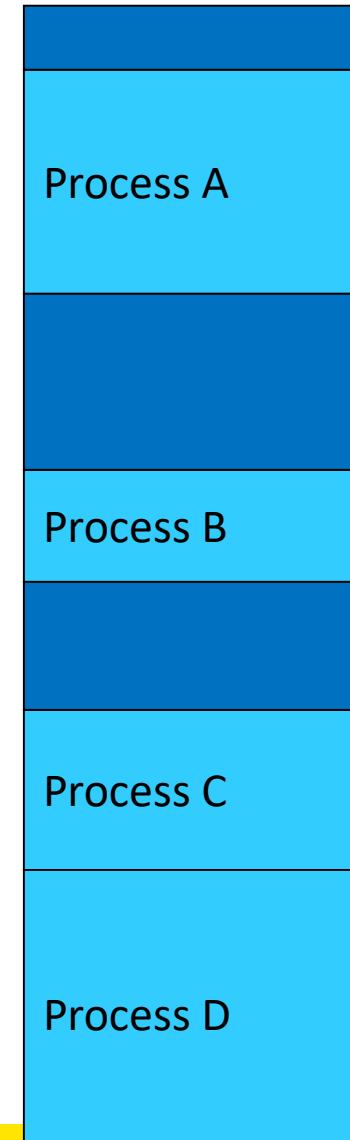
# Compaction

- We can reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block.
  - Only if we can relocate running programs?
    - Pointers?
  - Generally requires hardware support



# Some Remaining Issues with Dynamic Partitioning

- We have ignored
  - Relocation
    - How does a process run in different locations in memory?
  - Protection
    - How do we prevent processes interfering with each other



# Example Logical Address-Space Layout

- Logical addresses refer to specific locations within the program
- Once running, these address must refer to real physical memory
- When are logical addresses bound to physical?

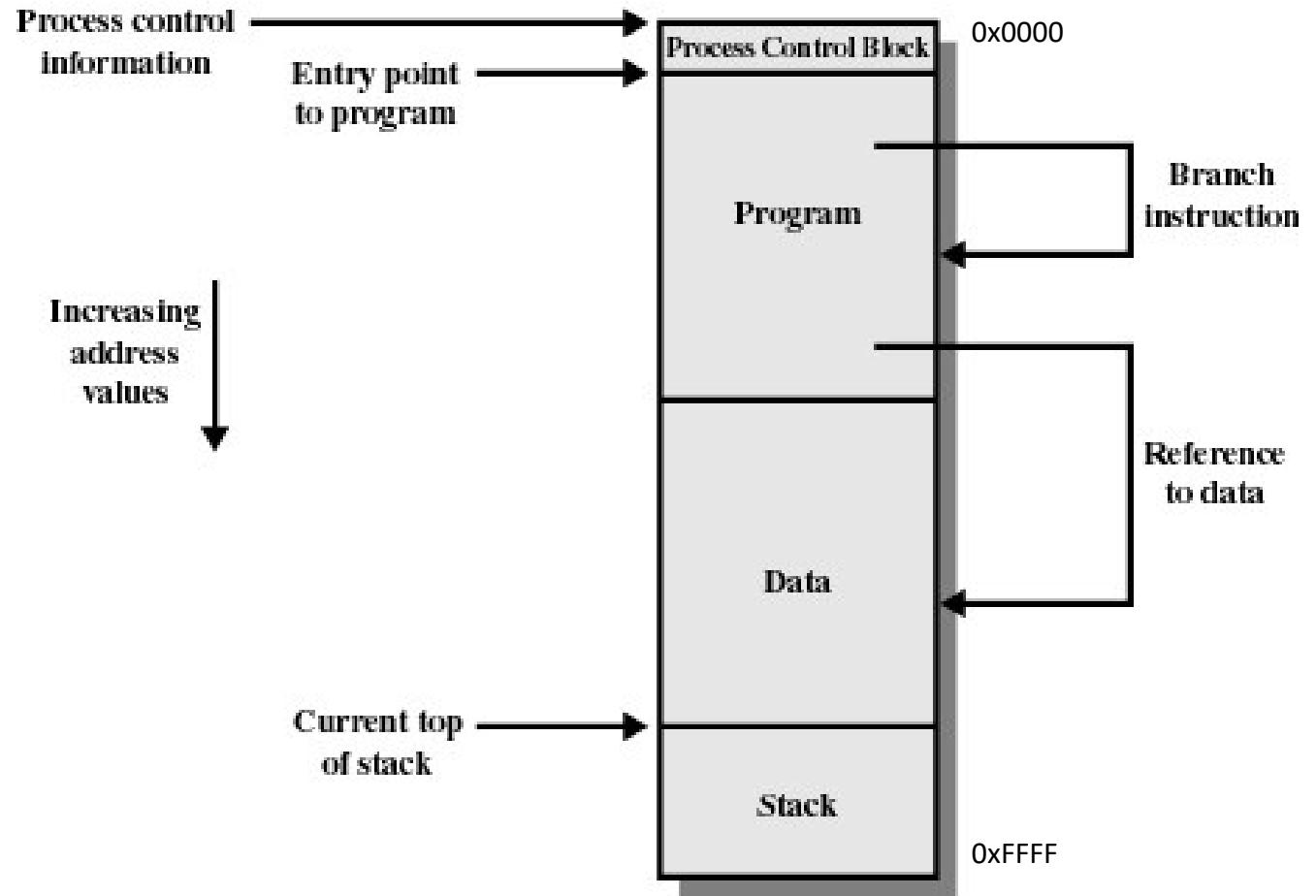
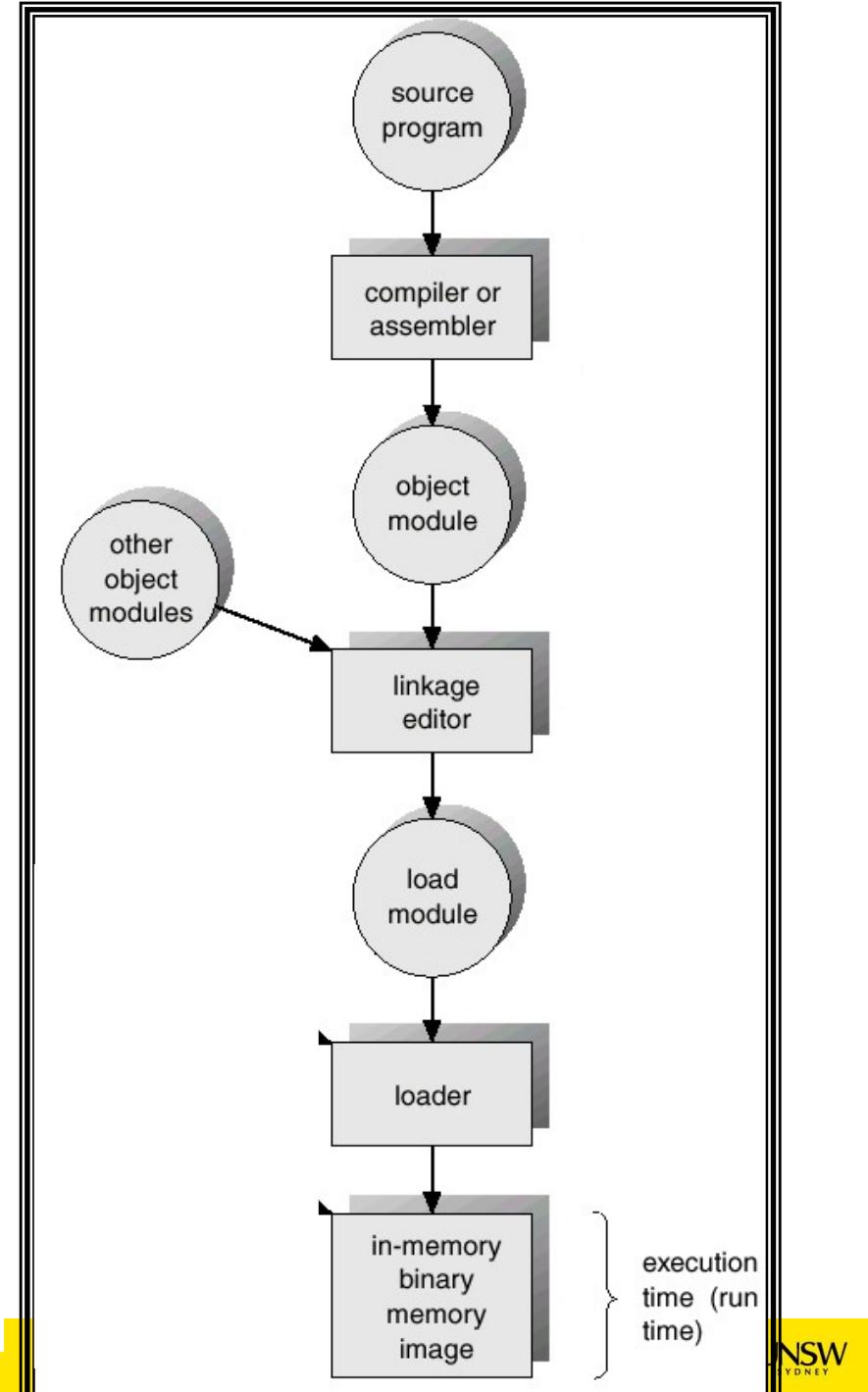


Figure 7.1 Addressing Requirements for a Process

# When are memory addresses bound?

- Compile/link time
  - Compiler/Linker binds the addresses
  - Must know “run” location at compile time
  - Recompile if location changes
- Load time
  - Compiler generates *relocatable* code
  - Loader binds the addresses at load time
- Run time
  - Logical compile-time addresses translated to physical addresses by *special hardware*.



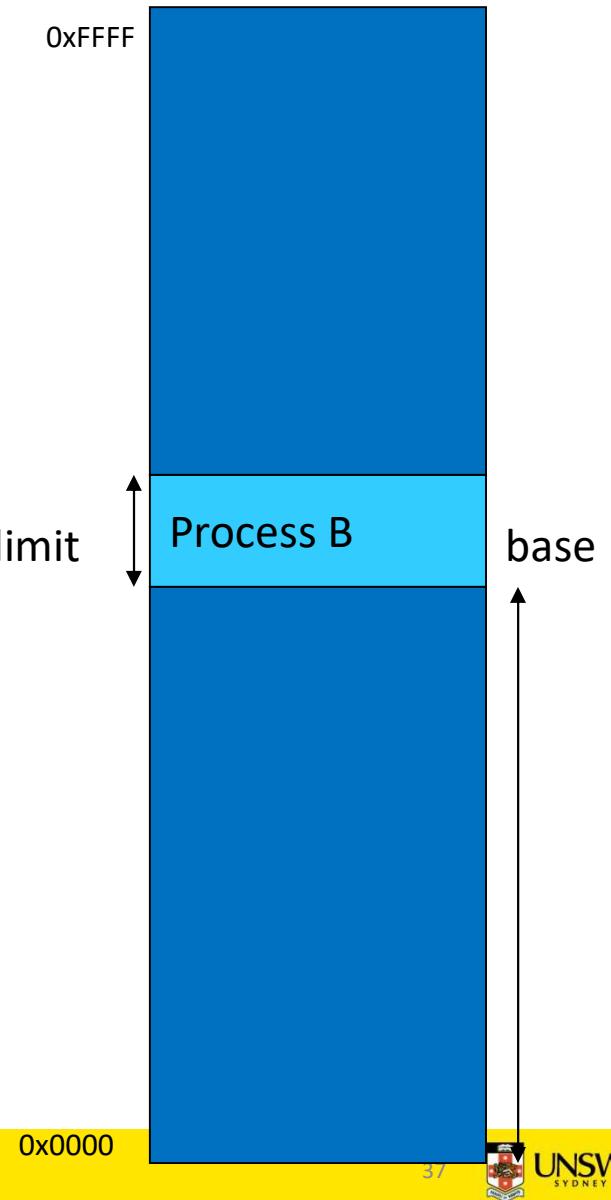
# Hardware Support for Runtime Binding and Protection

- For process B to run using logical addresses
  - Process B expects to access addresses from zero to some limit of memory size

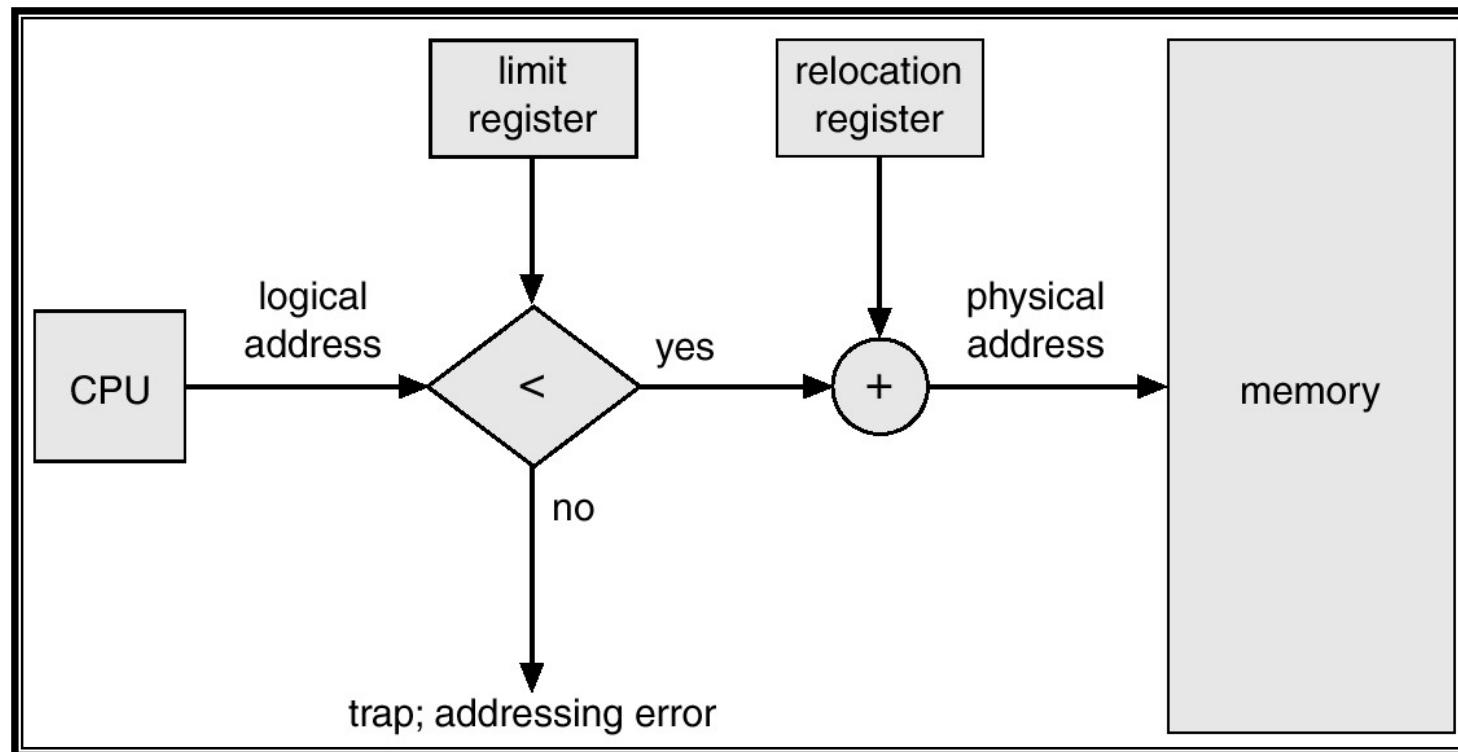


# Hardware Support for Runtime Binding and Protection

- For process B to run using logical addresses
  - Need to add an appropriate offset to its logical addresses
    - Achieve relocation
    - Protect memory “lower” than B
  - Must limit the maximum logical address B can generate
    - Protect memory “higher” than B

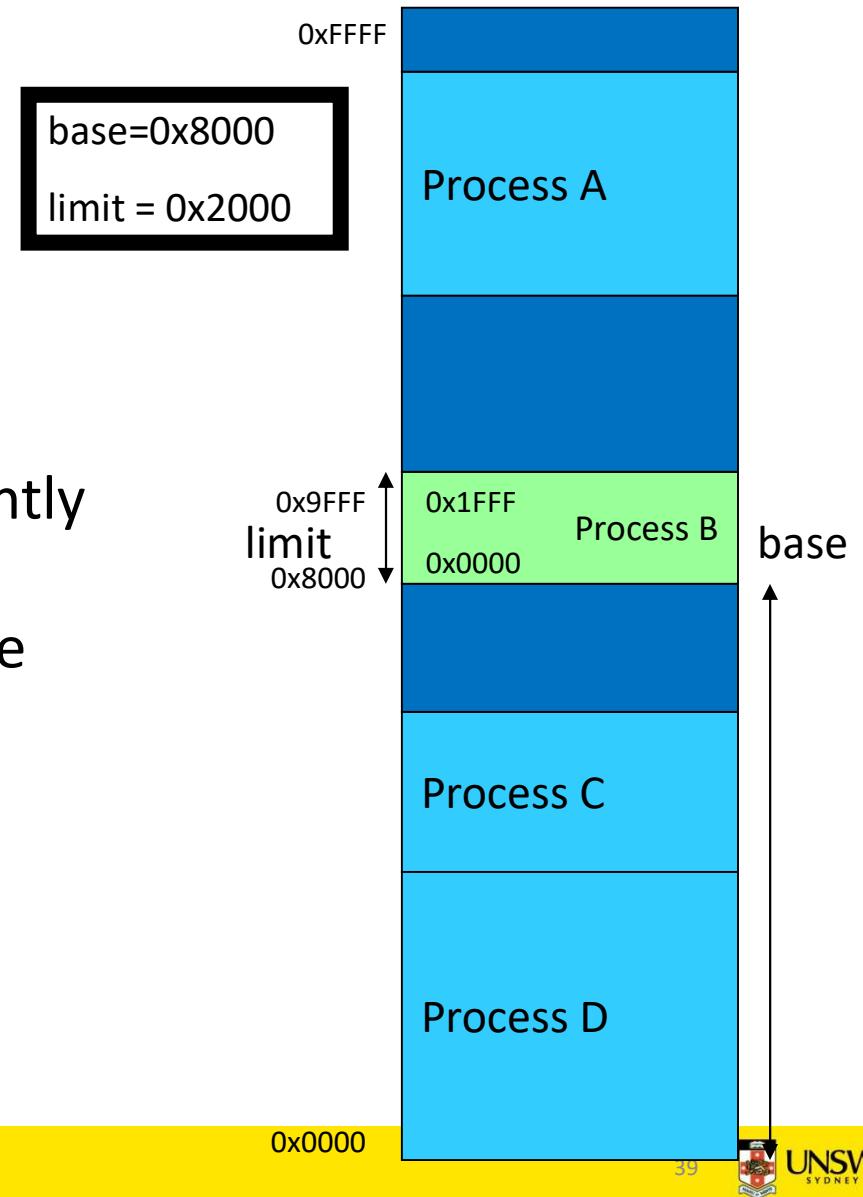


# Hardware Support for Relocation and Limit Registers



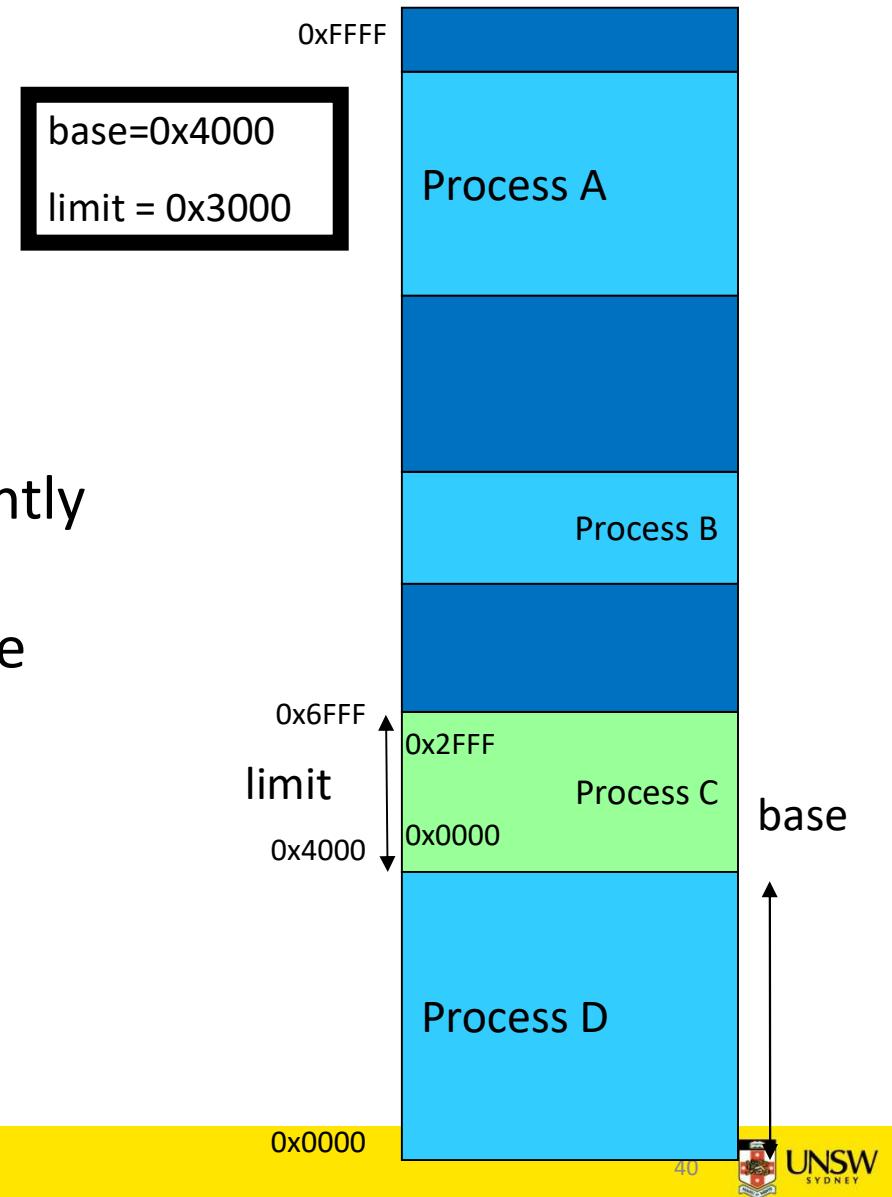
# Base and Limit Registers

- Also called
  - Base and bound registers
  - Relocation and limit registers
- Base and limit registers
  - Restrict and relocate the currently active process
  - Base and limit registers must be changed at
    - Load time
    - Relocation (compaction time)
    - On a context switch



# Base and Limit Registers

- Also called
  - Base and bound registers
  - Relocation and limit registers
- Base and limit registers
  - Restrict and relocate the currently active process
  - Base and limit registers must be changed at
    - Load time
    - Relocation (compaction time)
    - On a context switch

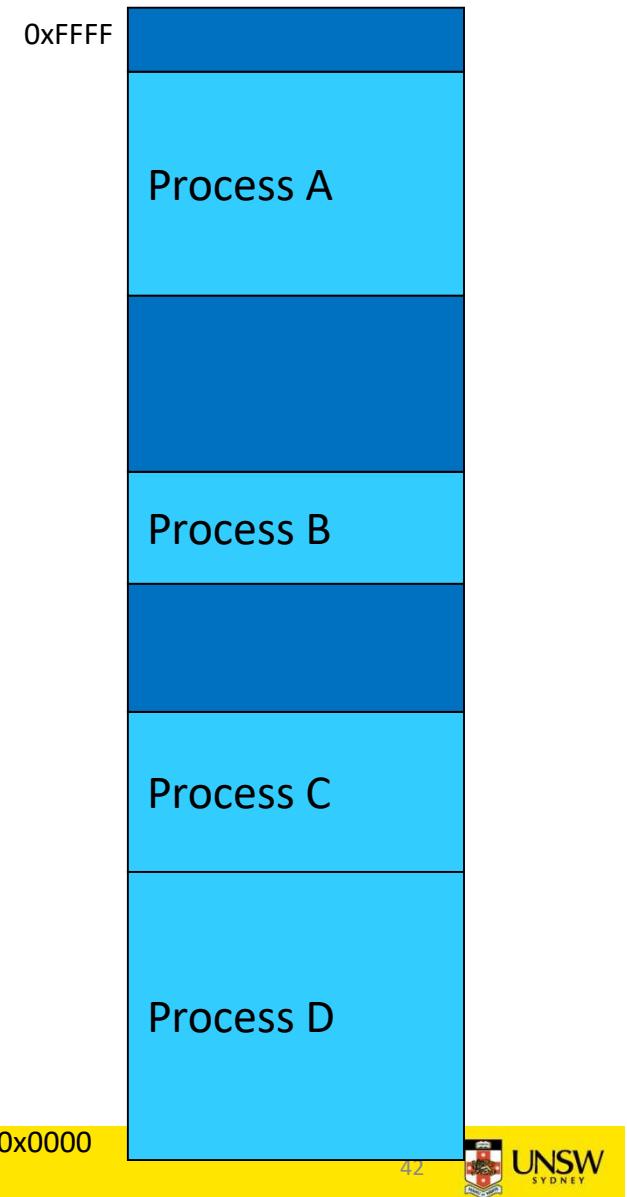


# Base and Limit Registers

- Pro
  - Supports protected multi-processing (-tasking)
- Cons
  - Physical memory allocation must still be contiguous
  - The entire process must be in memory
  - Do not support partial sharing of address spaces
    - No shared code, libraries, or data structures between processes

# Timesharing

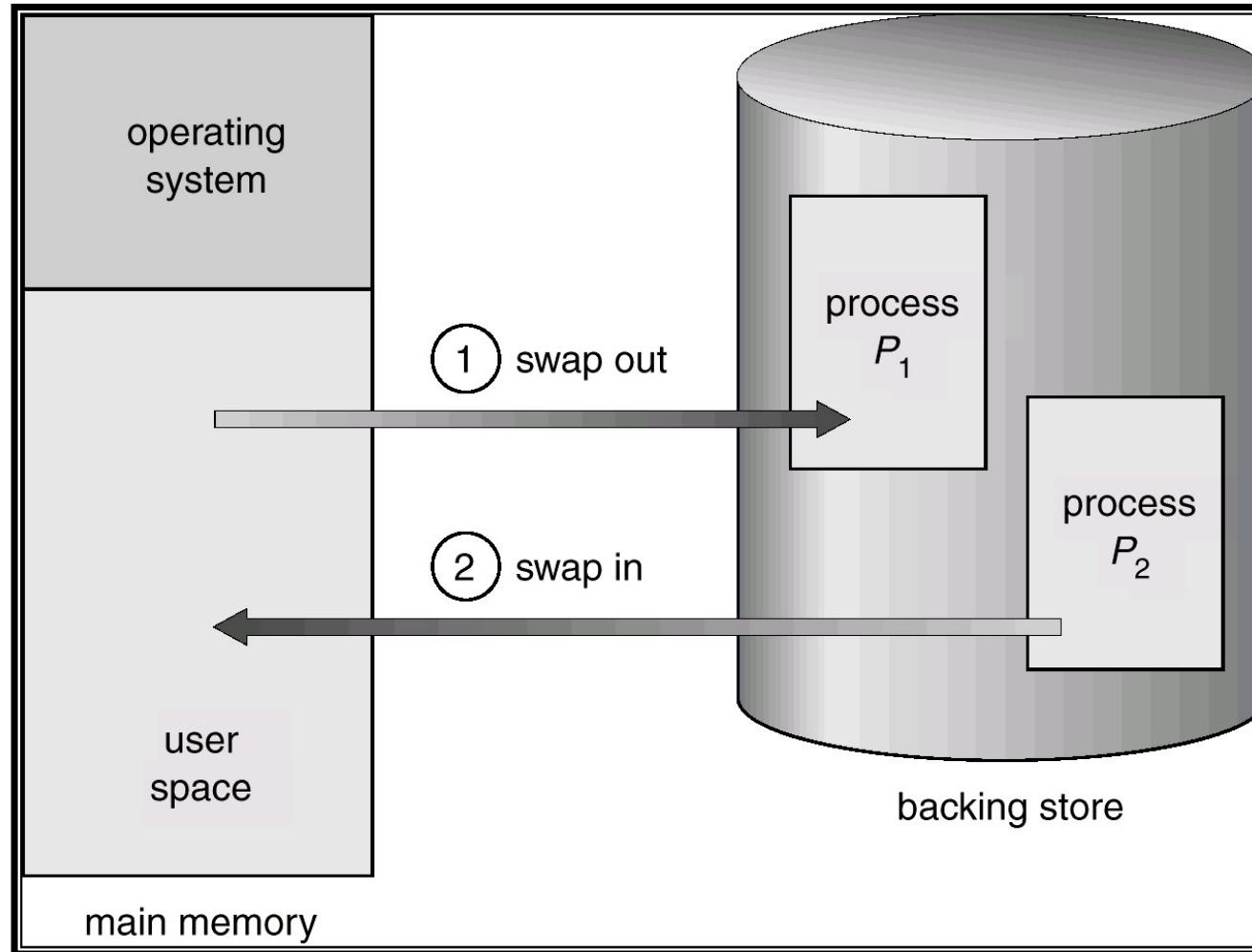
- Thus far, we have a system suitable for a batch system
  - Limited number of dynamically allocated processes
    - Enough to keep CPU utilised
  - Relocated at runtime
  - Protected from each other
- But what about timesharing?
  - We need more than just a small number of processes running at once
  - Need to support a mix of active and inactive processes, of varying longevity



# Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- Can prioritize – lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.
  - slow

# Schematic View of Swapping



So far we have assumed a process is smaller than memory

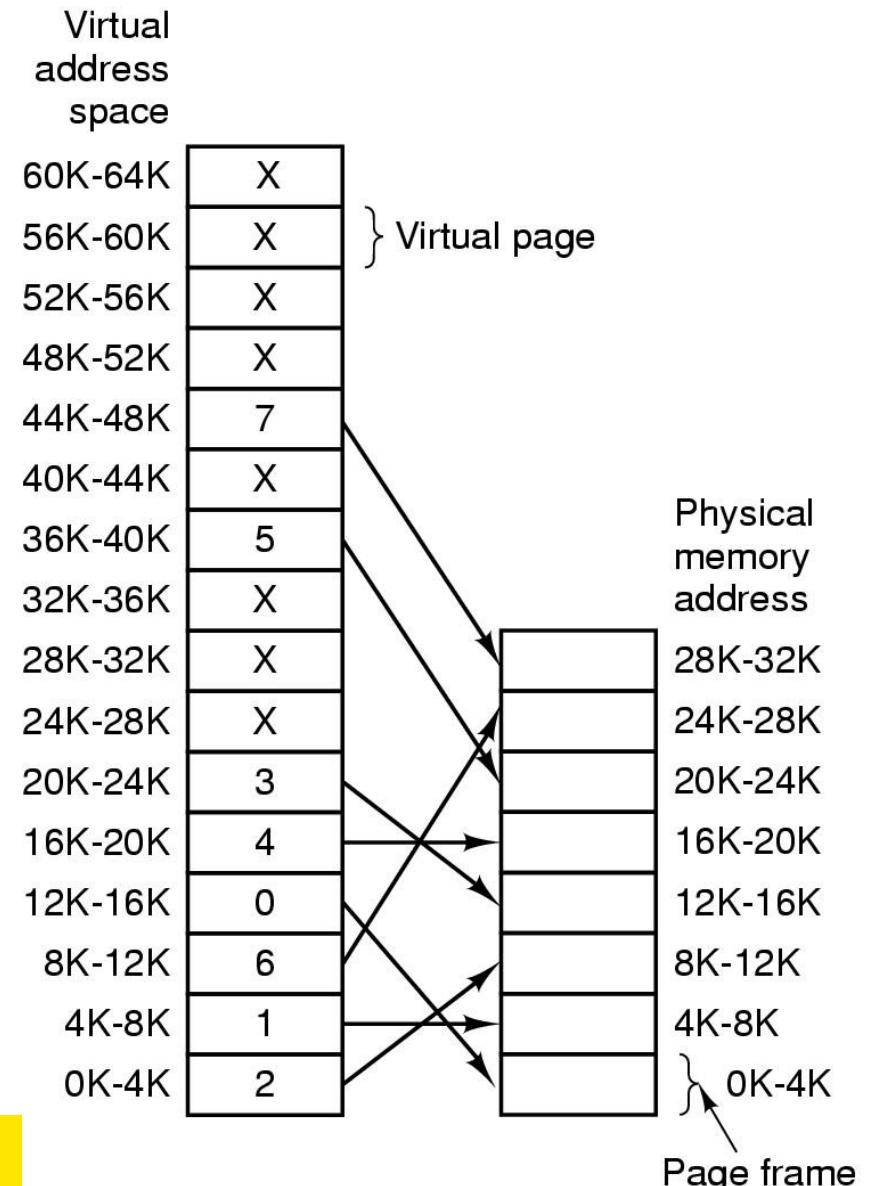
- What can we do if a process is larger than main memory?

# Virtual Memory

- Developed to address the issues identified with the simple schemes covered thus far.
- Two classic variants
  - Paging
  - Segmentation
    - (no longer covered in course, see textbook if interested)
- Paging is now the dominant one of the two
  - We'll focus on it
- Some architectures support hybrids of the two schemes
  - E.g. Intel IA-32 (32-bit x86)
    - Becoming less relevant

# Virtual Memory – Paging Overview

- Partition physical memory into small equal sized chunks
  - Called *frames*
- Divide each process's virtual (logical) address space into same size chunks
  - Called *pages*
  - Virtual memory addresses consist of a *page number* and *offset* within the page
- OS maintains a *page table*
  - contains the frame location for each page
  - Used by *hardware* to translate each virtual address to physical address
  - The relation between virtual addresses and physical memory addresses is given by page table
- Process's physical memory does **not** have to be contiguous



Frame number	Main memory
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(a) Fifteen Available Frames

Main memory
0 A.0
1 A.1
2 A.2
3 A.3
4
5
6
7
8
9
10
11
12
13
14

(b) Load Process A

Main memory
0 A.0
1 A.1
2 A.2
3 A.3
4 B.0
5 B.1
6 B.2
7
8
9
10
11
12
13
14

(b) Load Process B

**Figure 7.9 Assignment of Process Pages to Free Frames**

Main memory

0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

(d) Load Process C

0	0
1	1
2	2
3	3

Process A  
page table

0	—
1	—
2	—

Process B  
page table

Main memory

0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

(e) Swap out B

0	7
1	8
2	9
3	10

Process C  
page table

Main memory

0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

(f) Load Process D

0	4
1	5
2	6
3	11
4	12

Process D  
page table

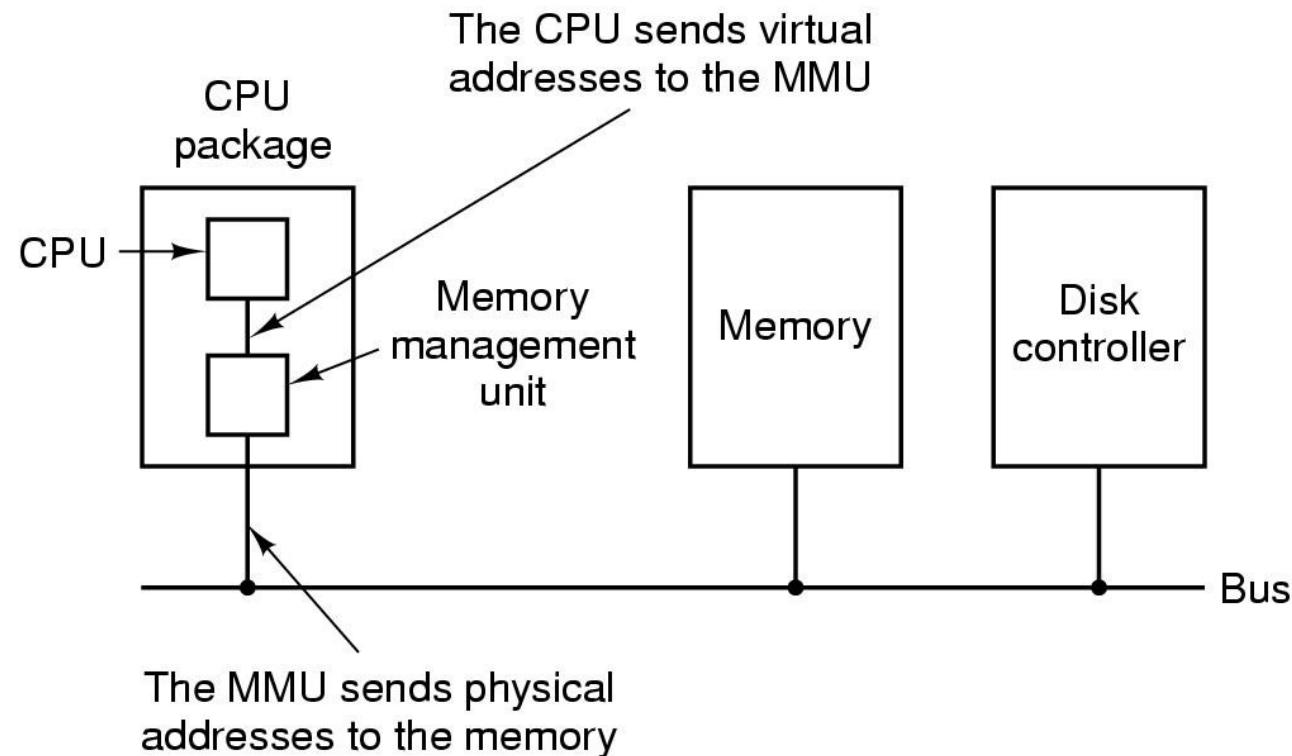
13
14

Free frame  
list

# Paging

- No external fragmentation
- Small internal fragmentation (in last page)
- Allows sharing by *mapping* several pages to the same frame
- Abstracts physical organisation
  - Programmer only deal with virtual addresses
- Minimal support for logical organisation
  - Each unit is one or more pages

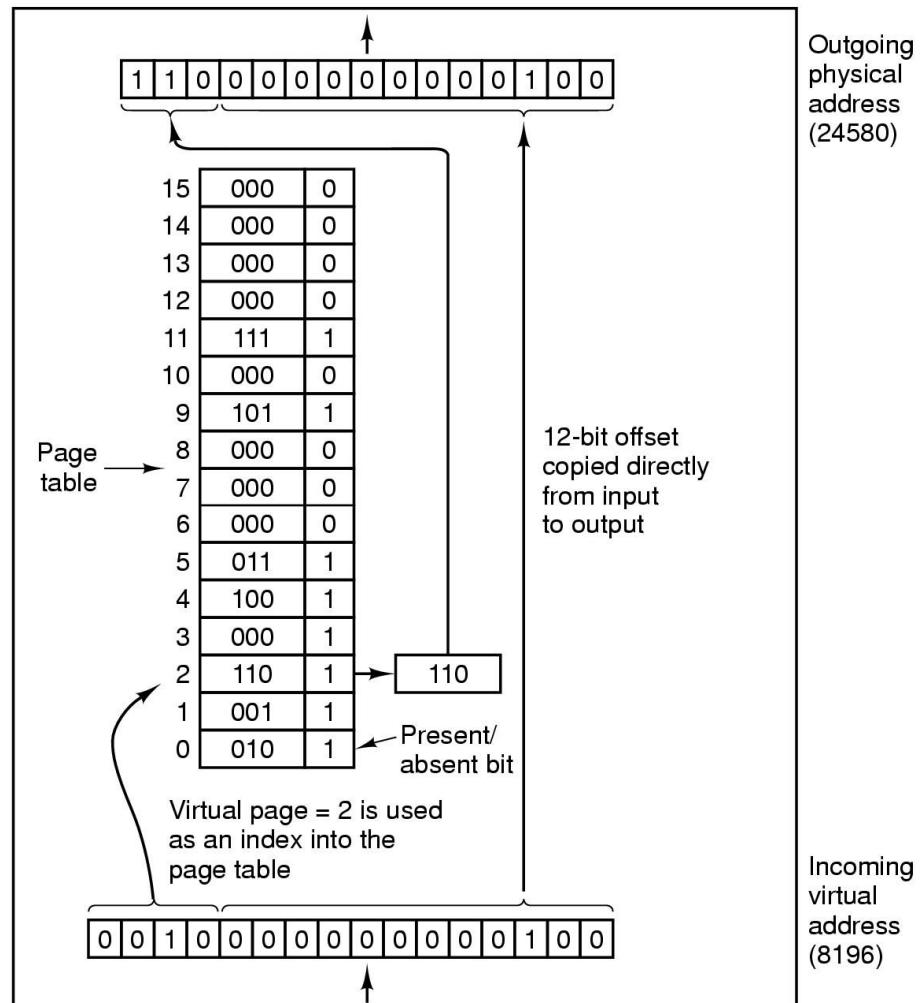
# Memory Management Unit (also called Translation Look-aside Buffer – TLB)



The position and function of the MMU

# MMU Operation

Assume for now that the page table is contained wholly in registers within the MMU – in practice it is not



Internal operation of simplified MMU with 16 4 KB pages

# Virtual Memory

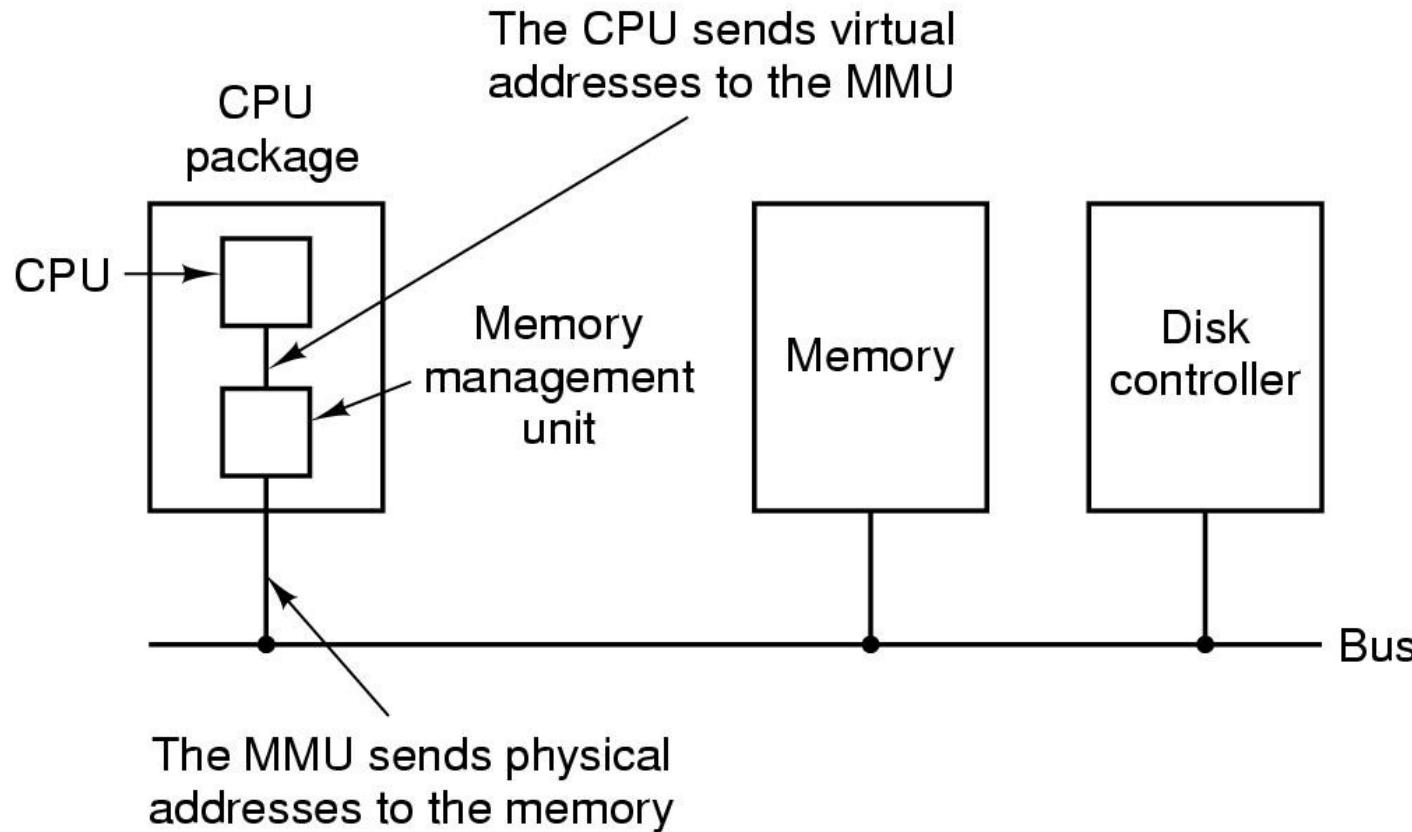


# Learning Outcomes

- An understanding of page-based virtual memory in depth.
  - Including the R3000's support for virtual memory.



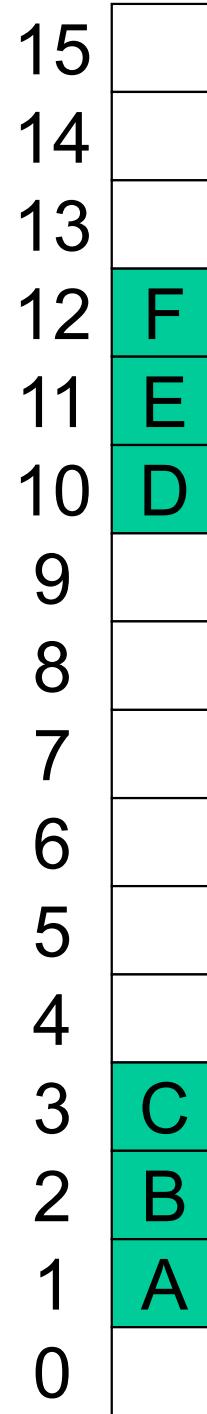
# Memory Management Unit (or TLB)



The position and function of the MMU



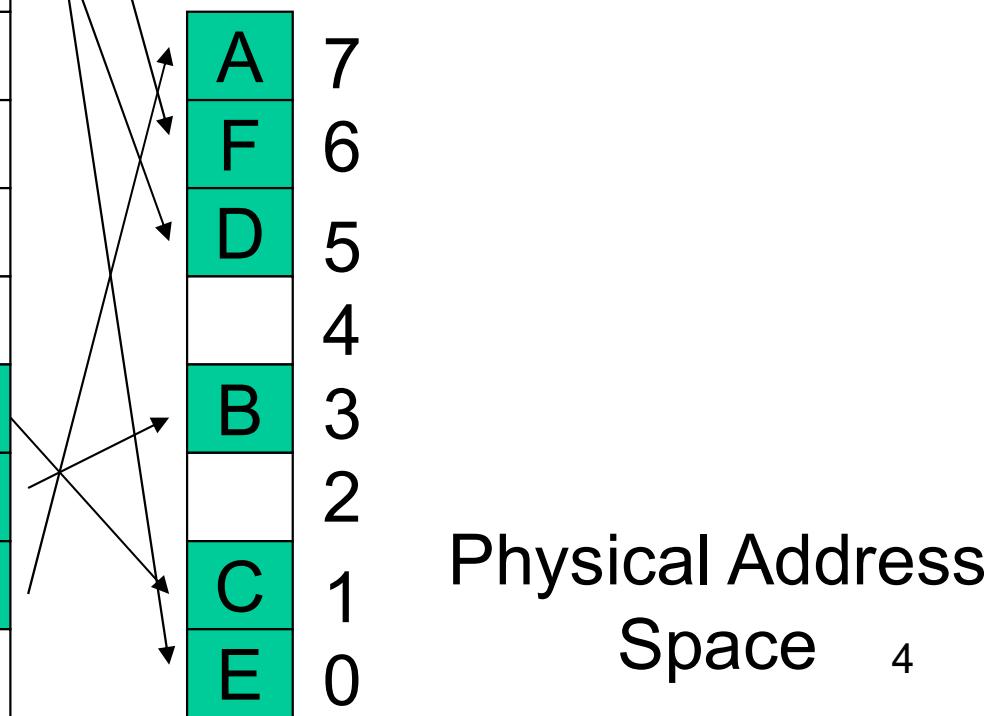
## Virtual Address Space



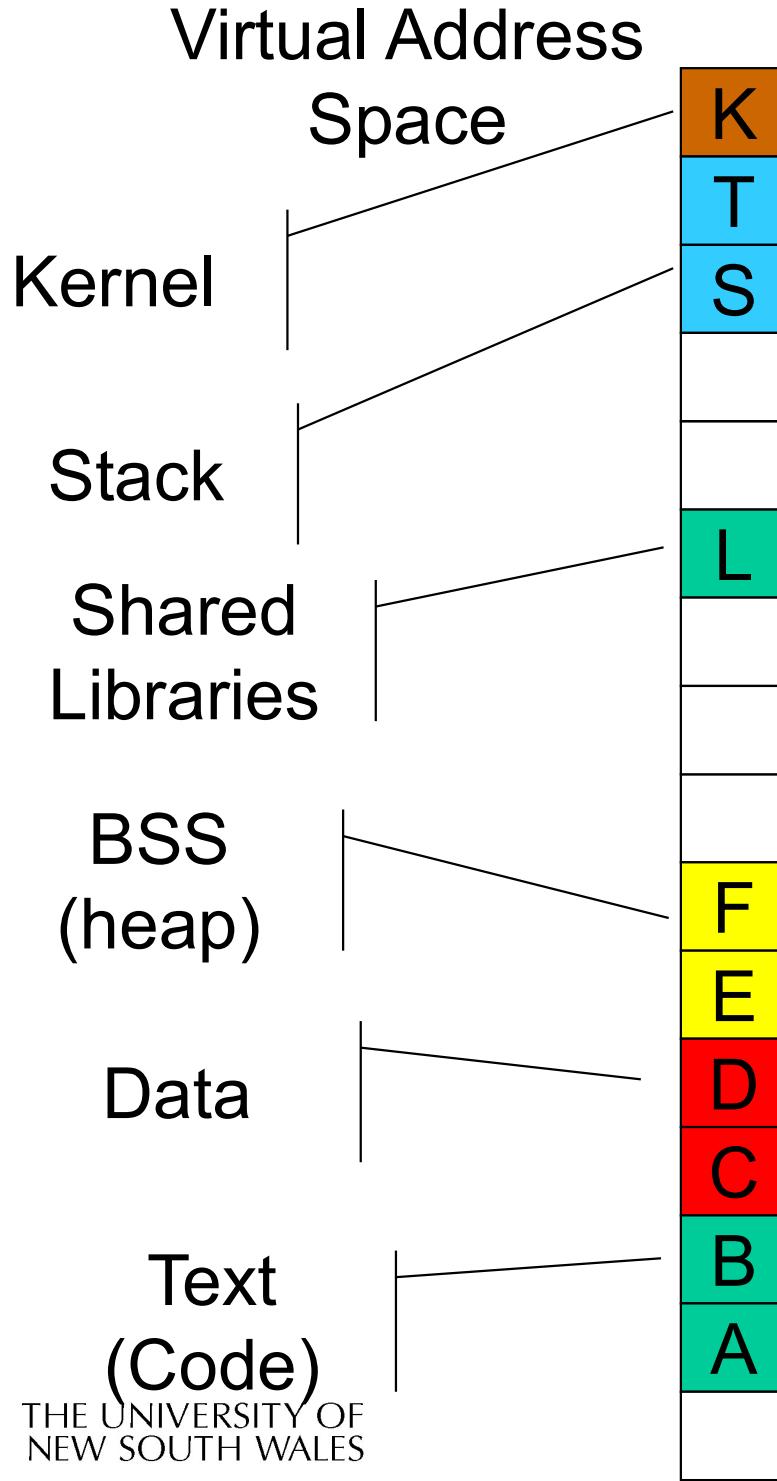
- Virtual Memory
  - Divided into equal-sized *pages*
  - A *mapping* is a translation between
    - A page and a frame
    - A page and *invalid*
  - Mappings defined at runtime
    - They can change
  - Address space can have holes
  - Process does not have to be contiguous in physical memory

## Page-based VM

- Physical Memory
  - Divided into equal-sized *frames*



Physical Address Space

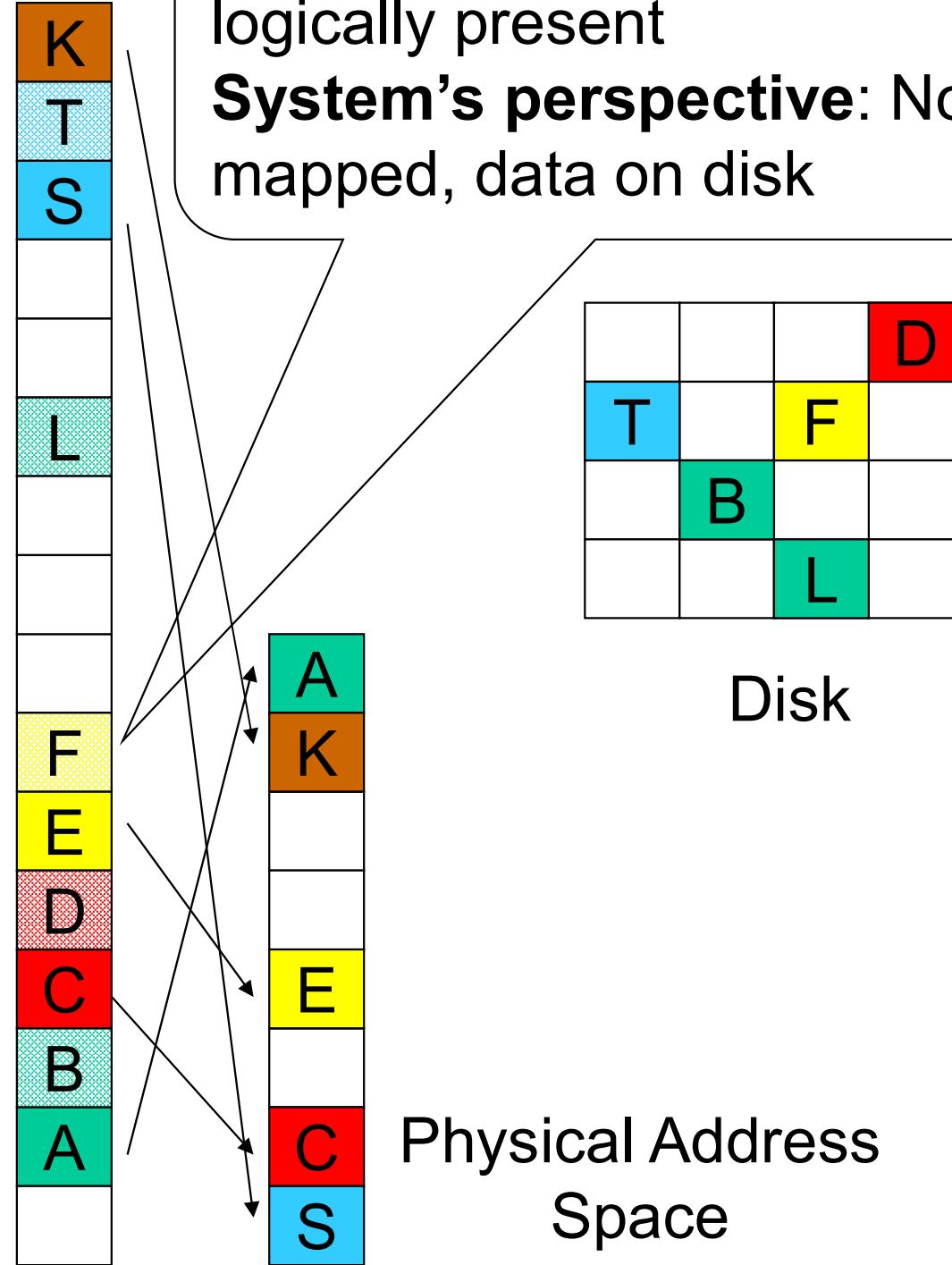


# Typical Address Space Layout

- Stack region is at top, and can grow down
- Heap has free space to grow up
- Text is typically read-only
- Kernel is in a reserved, protected, shared region
- 0-th page typically not used, why?

## Virtual Address Space

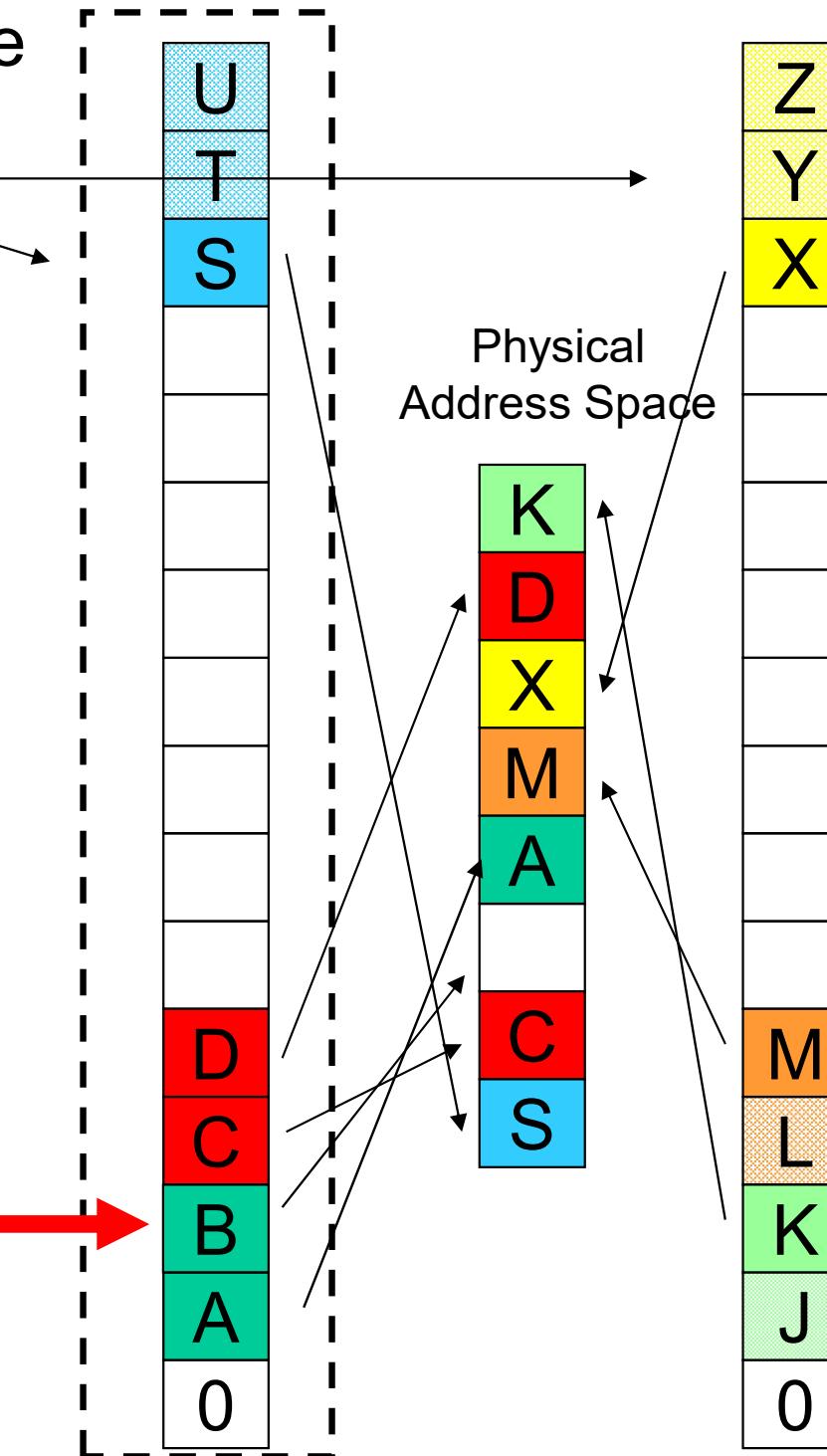
- A process may be only partially resident
  - Allows OS to store individual pages on disk
  - Saves memory for infrequently used data & code
- What happens if we access non-resident memory?



# Proc 1 Address Space

Currently running

Memory Access



# Proc 2 Address Space

Z  
Y  
X

U			
T			
B		Y	L
Z		J	
M			
L			
K			
J			
0			

Disk



# Page Faults

- Referencing an invalid page triggers a page fault
  - An exception handled by the OS
- Broadly, two standard page fault types
  - Illegal Address (protection error)
    - Signal or kill the process
  - Page not resident
    - Get an empty frame
    - Load page from disk
    - Update page (translation) table (enter frame #, set valid bit, etc.)
    - Restart the faulting instruction



## Virtual Address Space

Space

15  
14  
13  
12  
11  
10  
9  
8  
7  
6  
5  
4  
3  
2  
1  
0

K  
T  
S  
  
L  
  
F  
E  
D  
C  
B  
A

## Page Table

6

15  
14  
13  
12  
11  
10  
9  
8  
7  
6  
5  
4  
3  
1  
2  
7  
1  
0

9  
0

Physical  
Address Space

7  
6  
5  
4  
3  
2  
1  
0

A  
K  
  
E  
  
C  
S

- Page table for resident part of address space

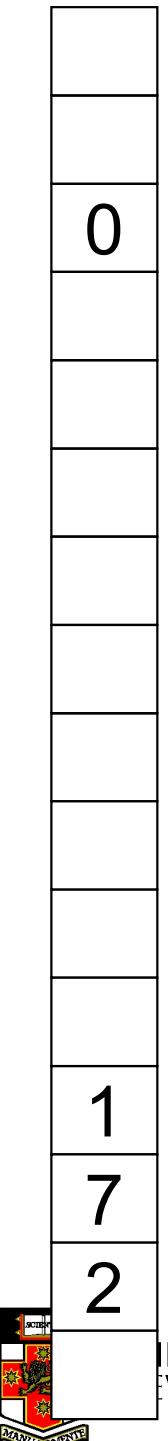


# Shared Pages

- Private code and data
  - Each process has own copy of code and data
  - Code and data can appear anywhere in the address space
- Shared code
  - Single copy of code shared between all processes executing it
  - Code must not be self modifying
  - Code must appear at same address in all processes

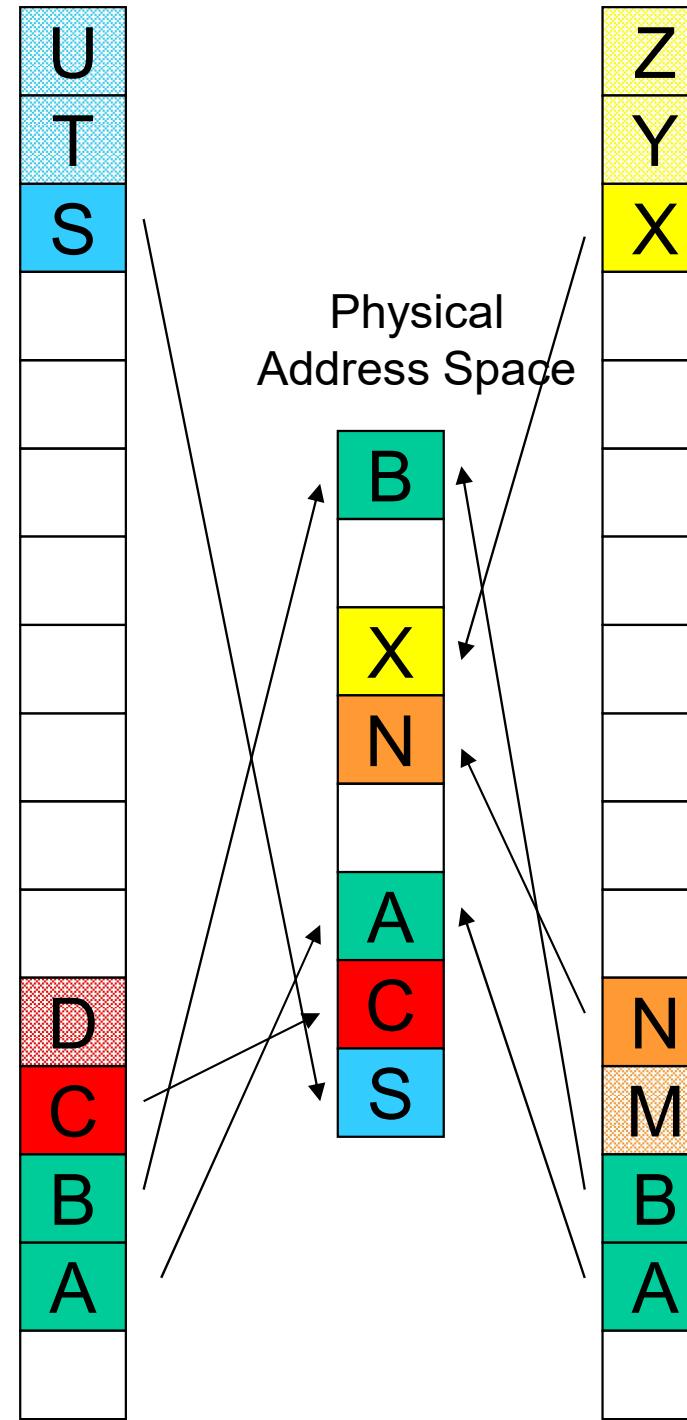


## Proc 1 Address Space

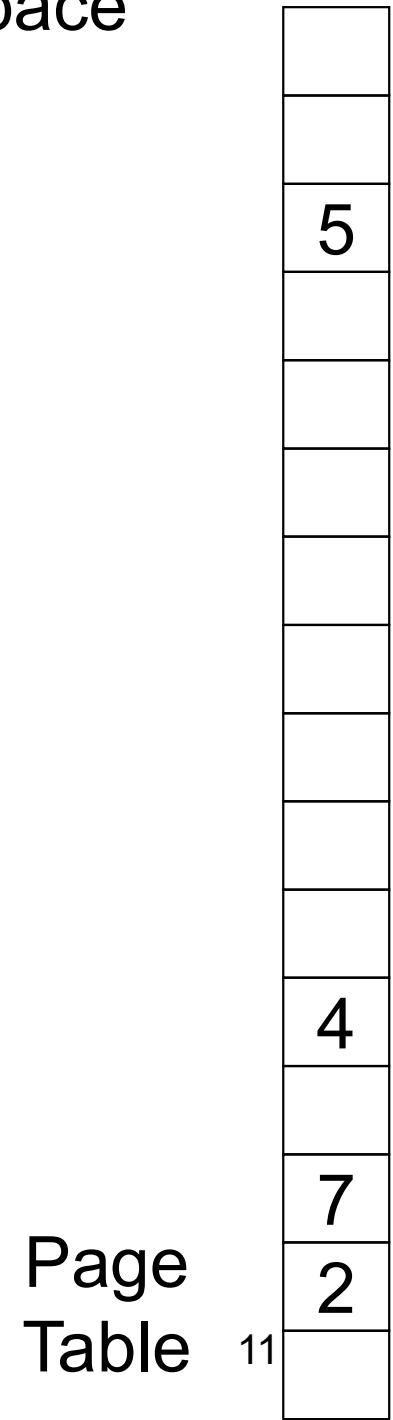


Two (or more) processes running the same program and sharing the text section

Page Table

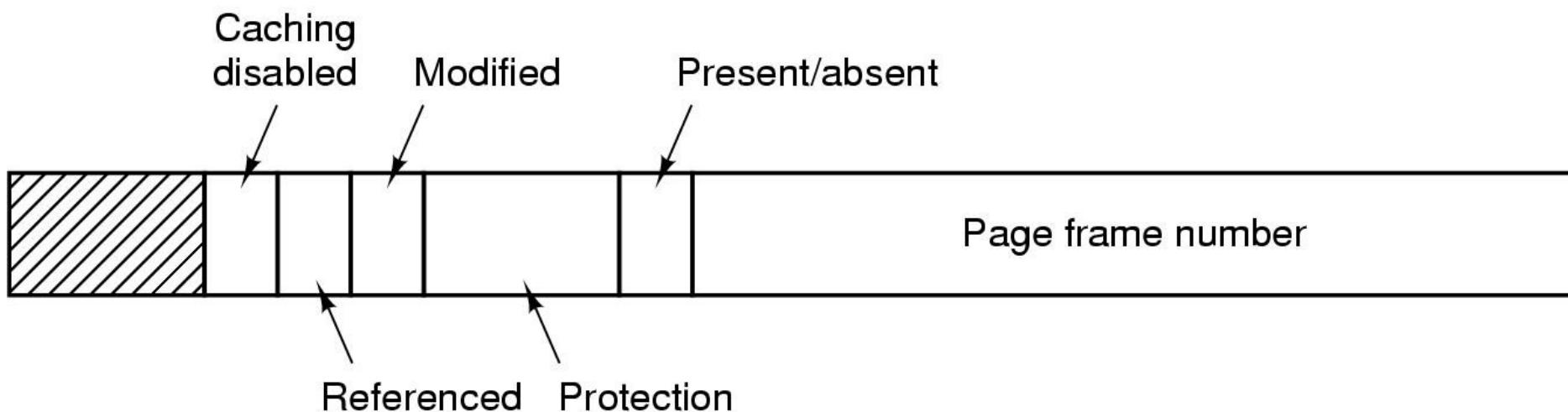


## Proc 2 Address Space



# Page Table Structure

- Page table is (logically) an array of frame numbers
  - Index by page number
- Each page-table entry (PTE) also has other bits



Page  
Table

12

5
4
7
2



# PTE Attributes (bits)

- Present/Absent bit
  - Also called *valid bit*, it indicates a valid mapping for the page
- Modified bit
  - Also called *dirty bit*, it indicates the page may have been modified in memory
- Reference bit
  - Indicates the page has been accessed
- Protection bits
  - Read permission, Write permission, Execute permission
  - Or combinations of the above
- Caching bit
  - Use to indicate processor should bypass the cache when accessing memory
    - Example: to access device registers or memory



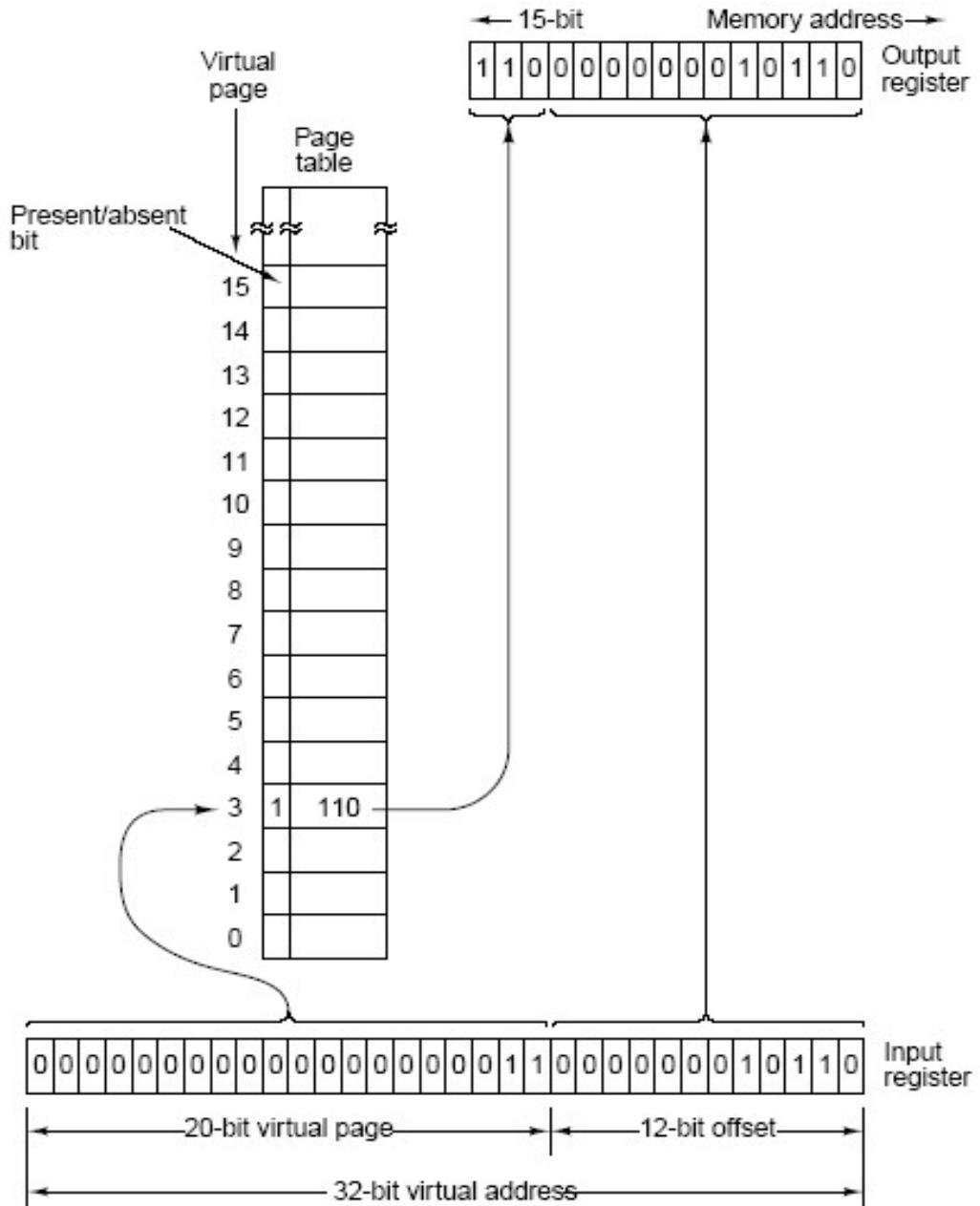
# Address Translation

- Every (virtual) memory address issued by the CPU must be translated to physical memory
  - Every *load* and every *store* instruction
  - Every instruction fetch
- Need Translation Hardware
- In paging system, translation involves replace page number with a frame number

# Virtual Memory Summary

virtual and physical mem chopped up in pages/frames

- programs **use virtual addresses**
- virtual to physical mapping by **MMU**
  - first check if page present (**present/absent bit**)
  - if yes: address in page table form MSBs in physical address
  - if no: bring in the page from disk  
→ **page fault**



# Page Tables

- Assume we have
  - 32-bit virtual address (4 Gbyte address space)
  - 4 KByte page size
  - How many page table entries do we need for one process?



# Page Tables

- Assume we have
  - 64-bit virtual address (**humungous** address space)
  - 4 KByte page size
  - How many page table entries do we need for one process?
- Problem:
  - Page table is very large
  - Access has to be fast, lookup for every memory reference
  - Where do we store the page table?
    - Registers?
    - Main memory?



# Page Tables

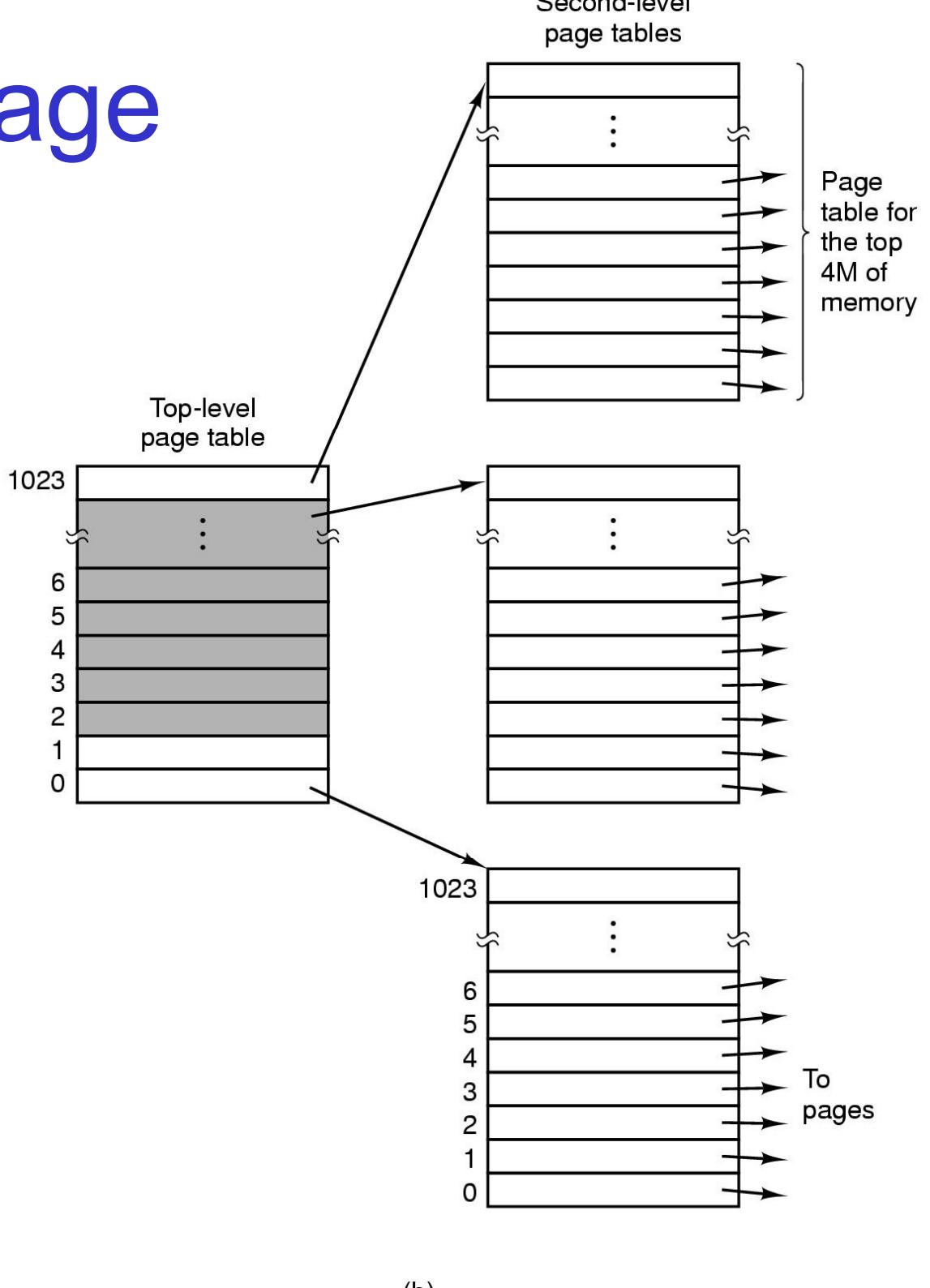
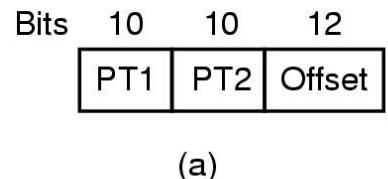
- Page tables are implemented as data structures in main memory
- Most processes do not use the full 4GB address space
  - e.g., 0.1 – 1 MB text, 0.1 – 10 MB data, 0.1 MB stack
- We need a compact representation that does not waste space
  - But is still very fast to search
- Three basic schemes
  - Use data structures that adapt to sparsity
  - Use data structures which only represent resident pages
  - Use VM techniques for page tables (details left to extended OS)



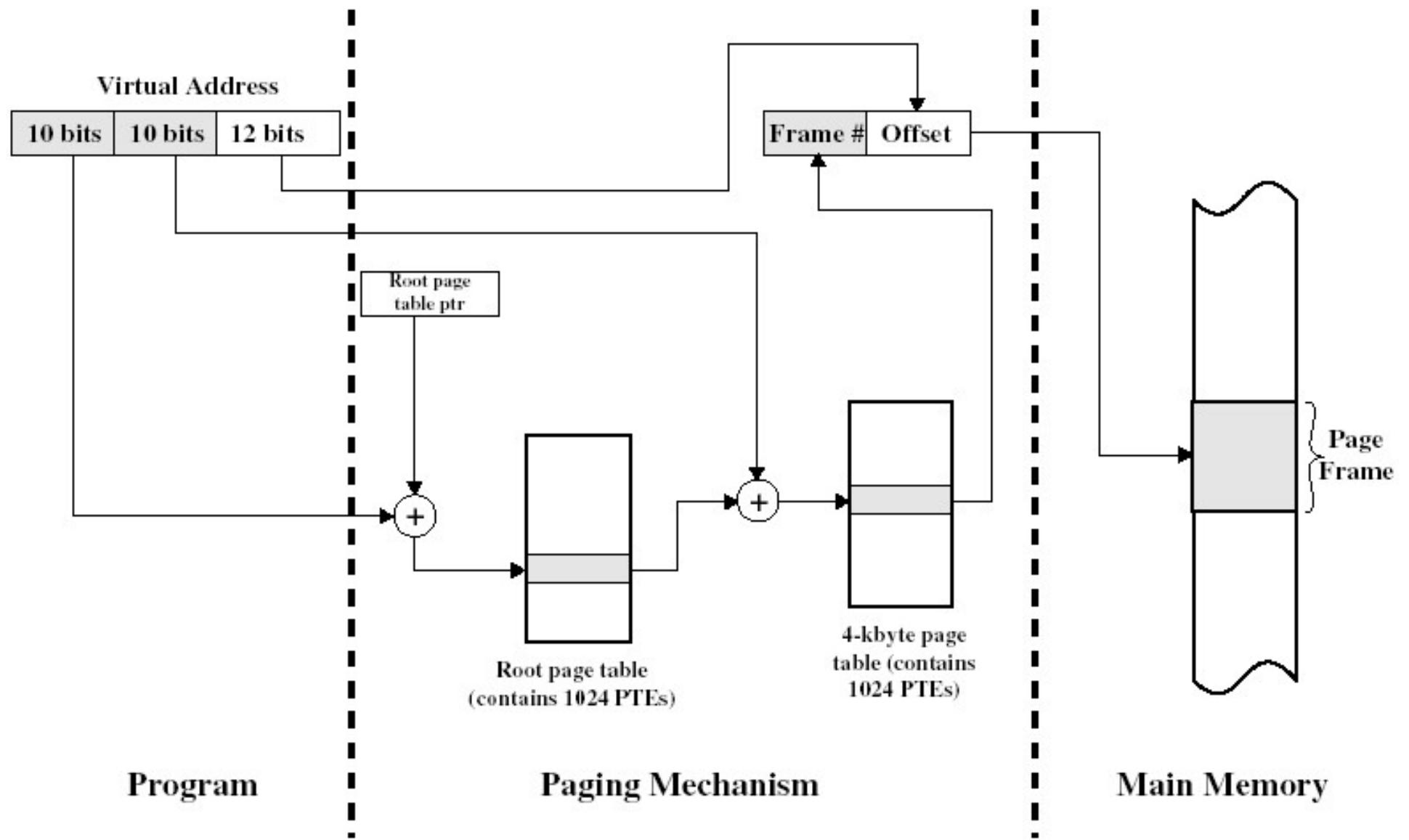
# Two-level Page Table

- 2<sup>nd</sup> –level page tables representing unmapped pages are not allocated

- Null in the top-level page table



# Two-level Translation



# Example Translations

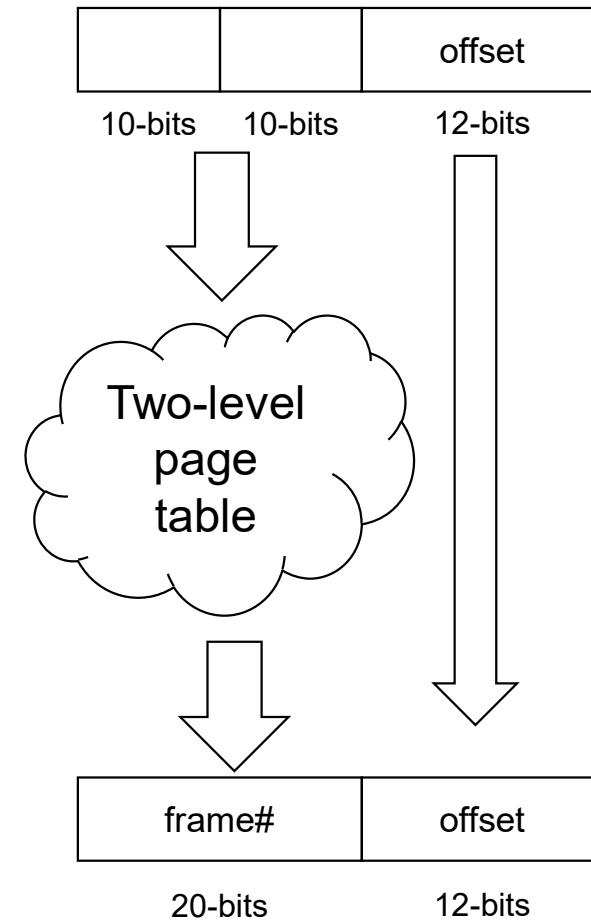




THE UNIVERSITY OF  
NEW SOUTH WALES

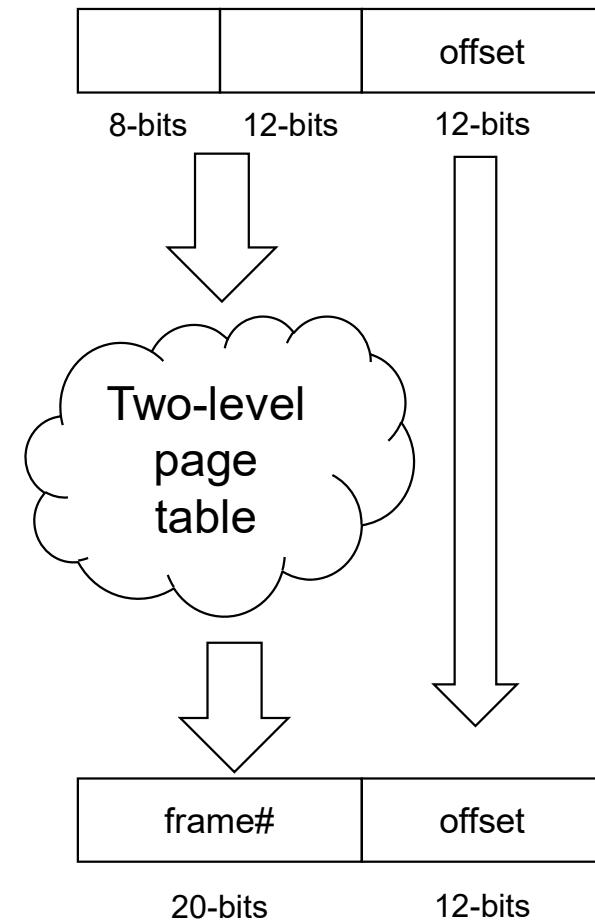
# Summarising Two-level Page Tables

- Translating a 32-bit virtual address into a 32-bit physical
- Recall:
  - the level 1 page table node has  $2^{10}$  entries
    - $2^{10} * 4 = 4 \text{ KiB}$  node
  - the level 2 page table node have  $2^{10}$  entries
    - $2^{10} * 4 = 4 \text{ KiB}$  node



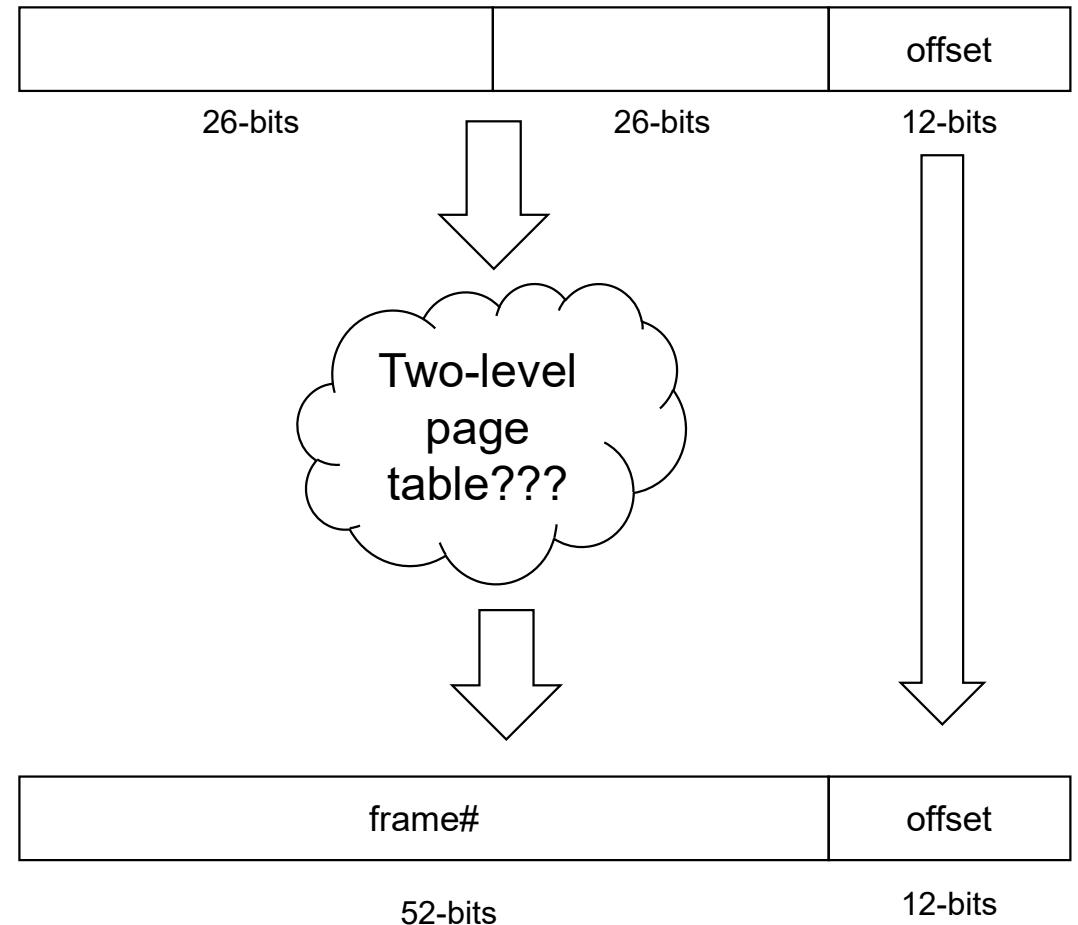
# Index bits determine node sizes

- Translating a 32-bit virtual address into a 32-bit physical
- Changing the indexing:
  - the level 1 page table node has  $2^8$  entries
    - $2^8 * 4 = 1 \text{ KiB}$  node
  - the level 2 page table node have  $2^{12}$  entries
    - $2^{12} * 4 = 16 \text{ KiB}$  node



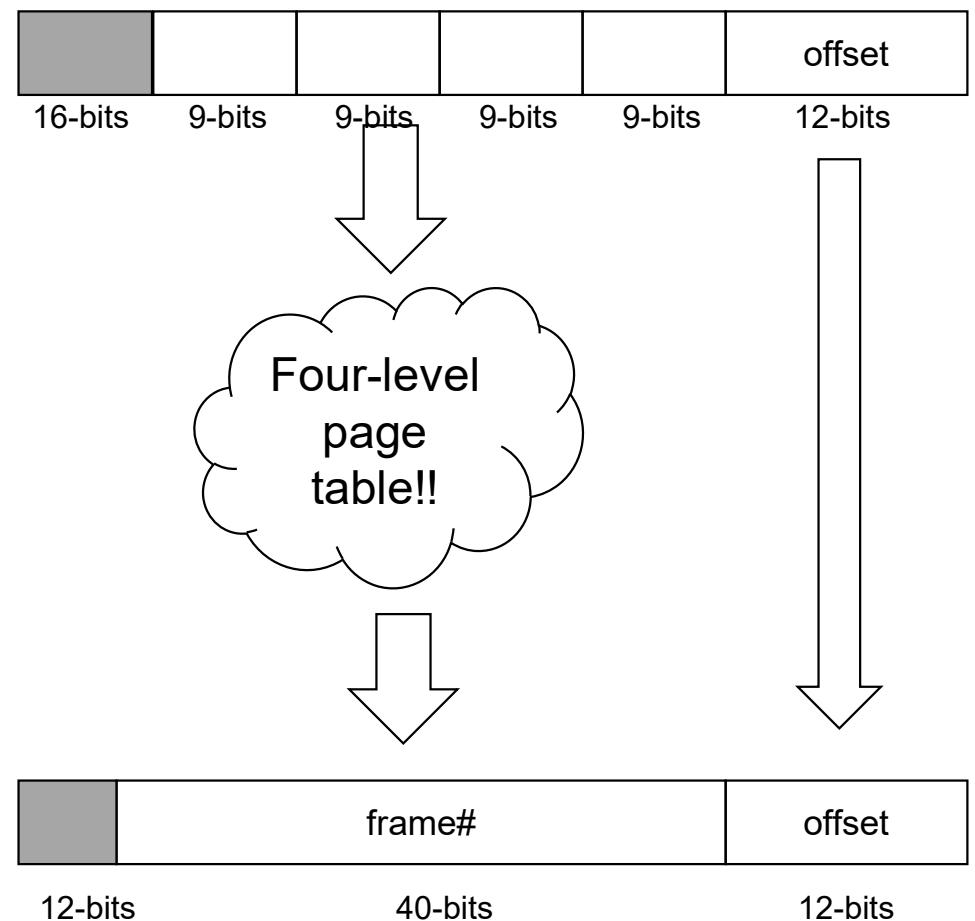
# Supporting 64-bit Virtual to Physical Translation

- Translating a 64-bit virtual address into a 64-bit physical???
- Support 64-bits?:
  - the level 1 page table node has  $2^{26}$  entries
    - $2^{26} * 8 = 512$  MiB node
  - the level 2 page table node have  $2^{12}$  entries
    - $2^{26} * 8 = 512$  MiB node



# Multi-level Page Tables

- Translating a 64-bit virtual address into a 64-bit physical (Intel/AMD pre-Ice Lake)
  - Only support 48-bit addresses
    - Top 16-bits unused
  - the level 1 page table node has  $2^9$  entries
    - $2^9 * 8 = 4 \text{ KiB node}$
  - the level 2 page table node have  $2^9$  entries
    - $2^9 * 8 = 4 \text{ KiB node}$
  - the level 3 page table node have  $2^9$  entries
    - $2^9 * 8 = 4 \text{ KiB node}$
  - the level 4 page table node have  $2^9$  entries
    - $2^9 * 8 = 4 \text{ KiB node}$



# Intel 4-Level Page Tables

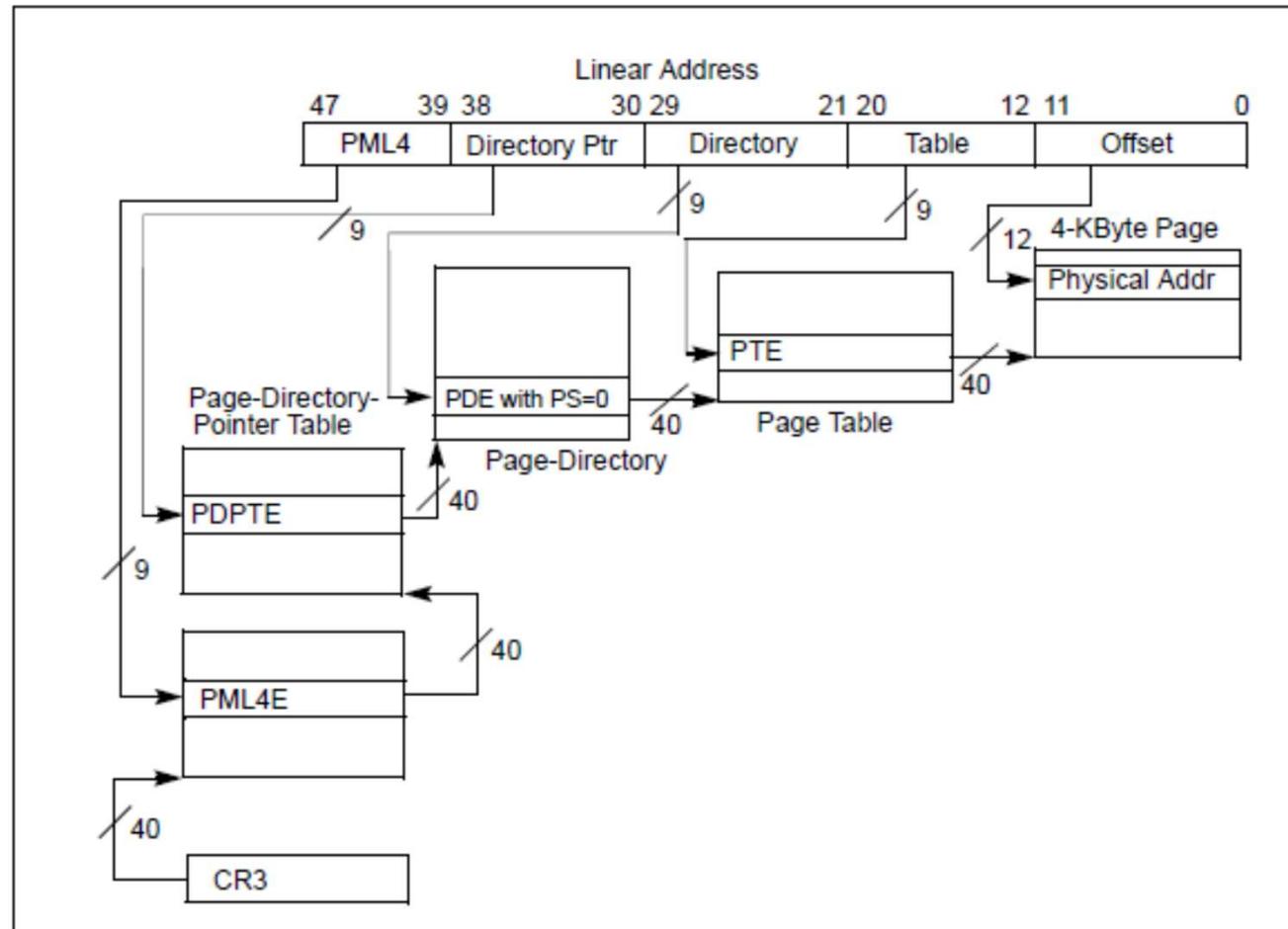
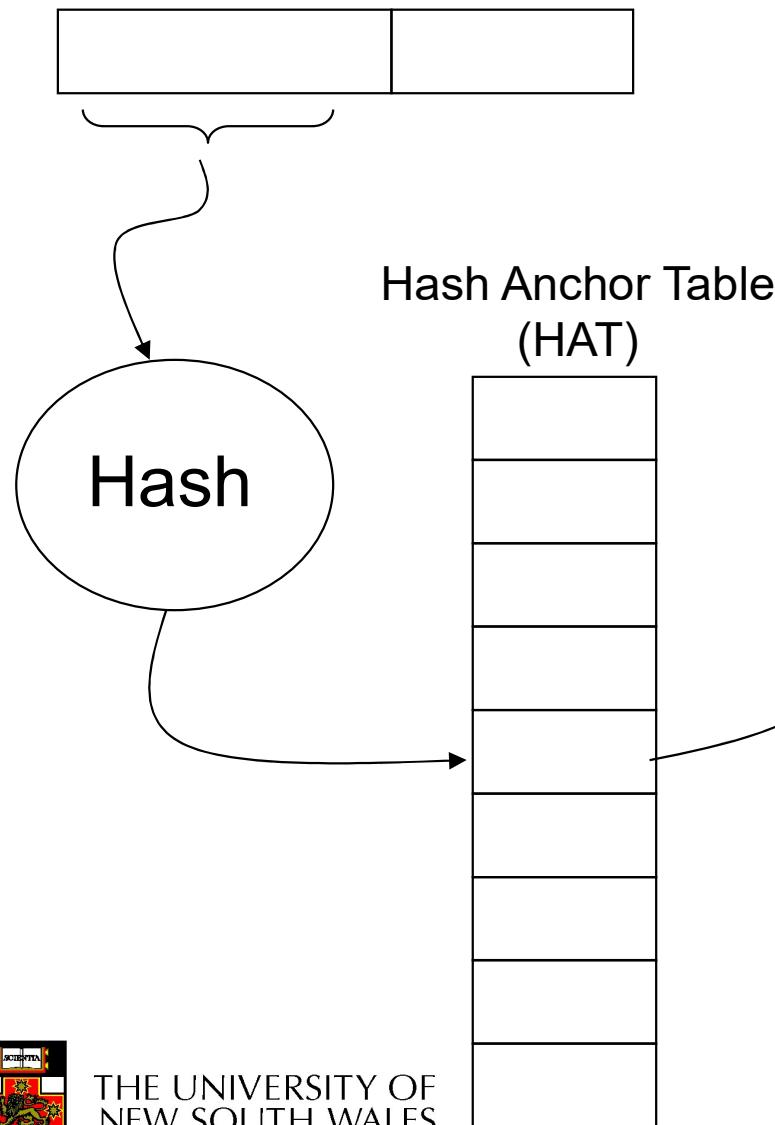


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging



# Alternative: Inverted Page Table

PID    VPN    offset



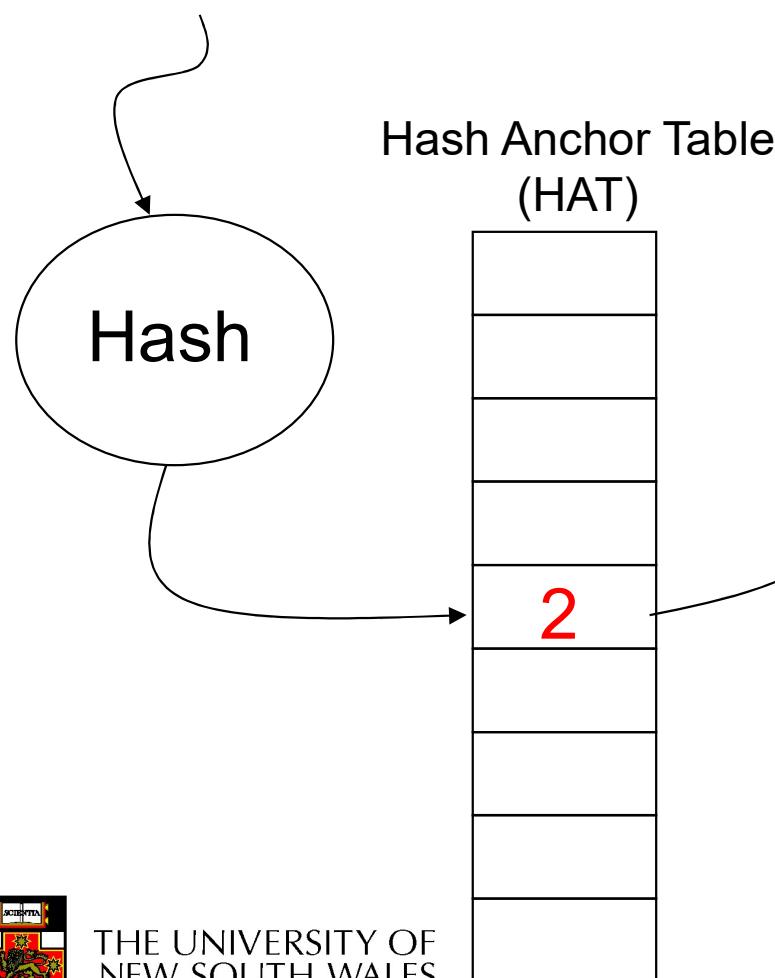
Index	PID	VPN	ctrl	next
0				
1				
2				
3				
4				
5				
6				
...				

IPT: entry for each *physical* frame



# Alternative: Inverted Page Table

PID	VPN	offset
0	0x5	0x123



Index	PID	VPN	ctrl	next
0				
1				
2	1	0x1A		0x40C
...				
0x40C	0	0x5		0x0
0x40D				
...				
...				

ppn	offset
0x40C	0x123



# Inverted Page Table (IPT)

- “Inverted page table” is an array of page numbers sorted (indexed) by frame number (it’s a frame table).
- Algorithm
  - Compute hash of page number
  - Extract index from hash table
  - Use this to index into inverted page table
  - Match the PID and page number in the IPT entry
  - If match, use the index value as frame # for translation
  - If no match, get next candidate IPT entry from chain field
  - If NULL chain entry  $\Rightarrow$  page fault



# Properties of IPTs

- IPT grows with size of RAM, NOT virtual address space
- Frame table is needed anyway (for page replacement, more later)
- Need a separate data structure for non-resident pages
- Saves a vast amount of space (especially on 64-bit systems)
- Used in some IBM and HP workstations



# Given $n$ processes

- how many page tables will the system have for
  - ‘normal’ page tables
  - inverted page tables?



# Another look at sharing...



THE UNIVERSITY OF  
NEW SOUTH WALES

Proc 1 Address Space

Proc 2 Address Space

'easy' for normal page tables

Two (or more) processes running the same program and sharing the text section

Physical Address Space

B

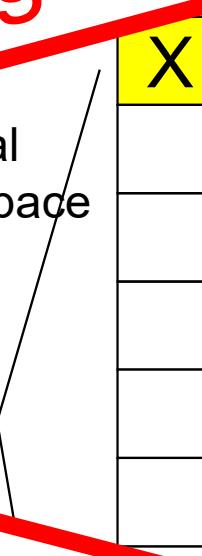
X

D  
C  
B  
A

A  
C  
S

X

N  
M  
B  
A



bit of a nightmare for IPT

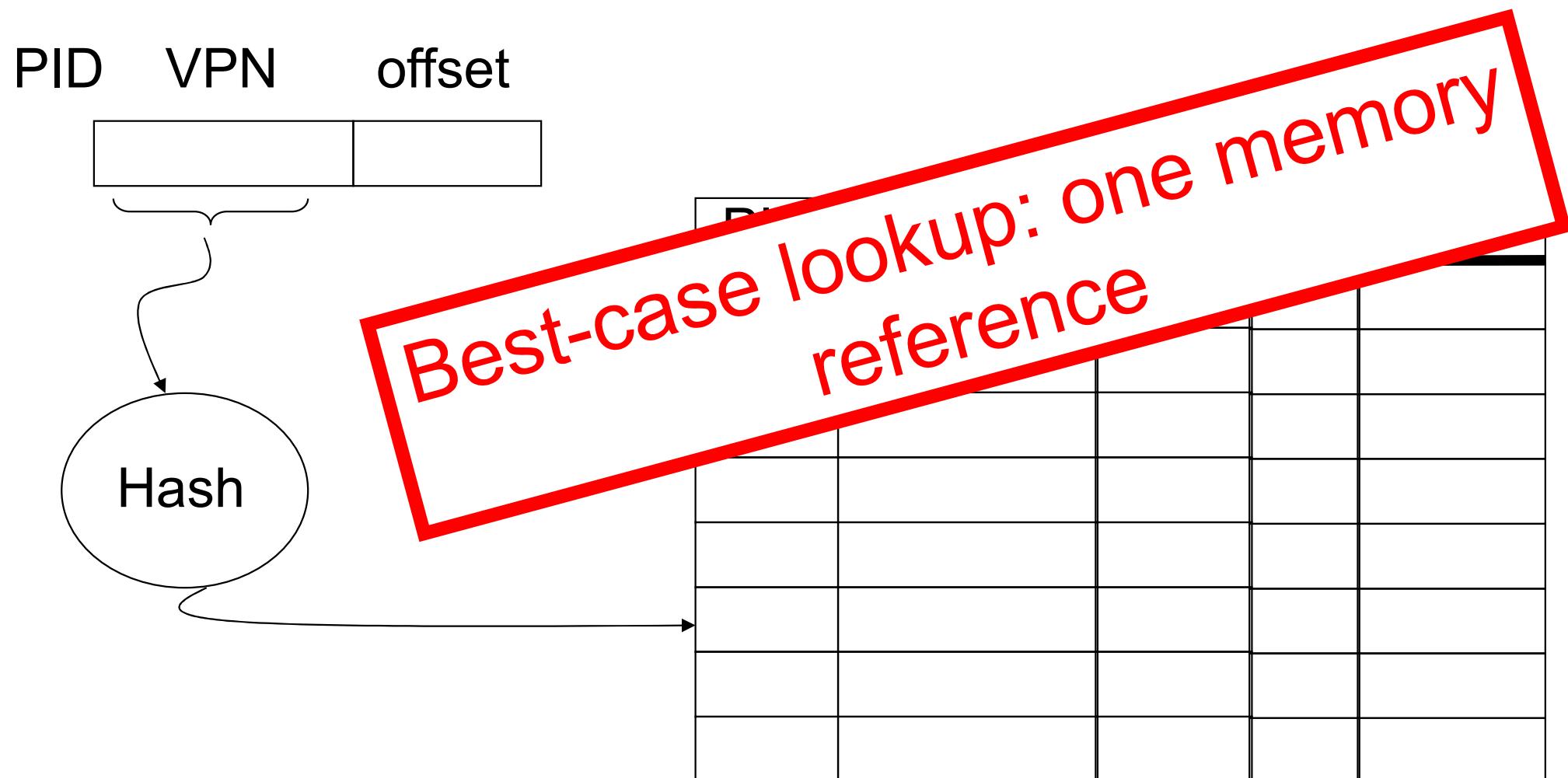


# Improving the IPT: Hashed Page Table

- Retain fast lookup of IPT
  - A single memory reference in best case
- Retain page table sized based on physical memory size (not virtual)
  - Enable efficient frame sharing
  - Support more than one mapping for same frame



# Hashed Page Table



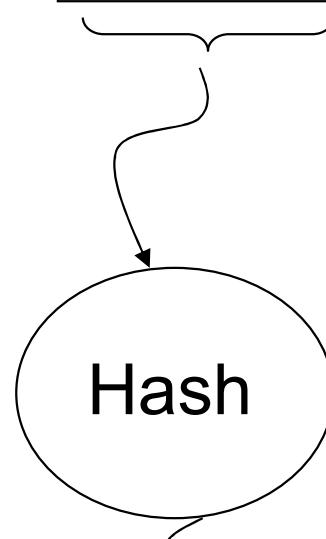
HPT: Frame number stored in table



# Hashed Page Table

PID    VPN    offset

0	0x5	0x123
---	-----	-------



	PID	VPN	PFN	ctrl	next
1					
2					
3	0	0x5	0x42		0x0
4					
5					
6	1	0x1A	0x13		0x3
...					

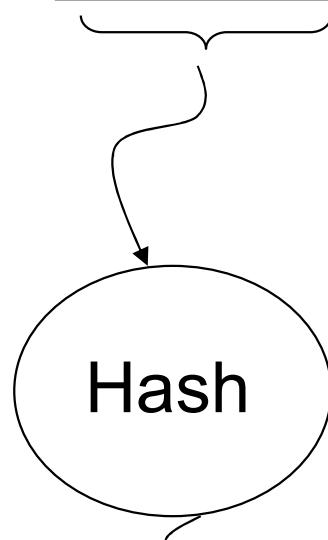
ppn	offset
0x42	0x123



# Sharing Example

PID    VPN    offset

0	0x5	0x123
---	-----	-------



	PID	VPN	PFN	ctrl	next
1					
2					
3	1	0x5	0x42		0x0
4					
5					
6	0	0x5	0x42		0x3
...					

ppn	offset
0x42	0x123



# Sizing the Hashed Page Table

- HPT sized based on physical memory size
- With sharing
  - Each frame can have more than one PTE
  - More sharing increases number of slots used
    - Increases collision likelihood
- However, we can tune HPT size based on:
  - Physical memory size
  - Expected sharing
  - Hash collision avoidance.
  - HPT a power of 2 multiple of number of physical memory frame



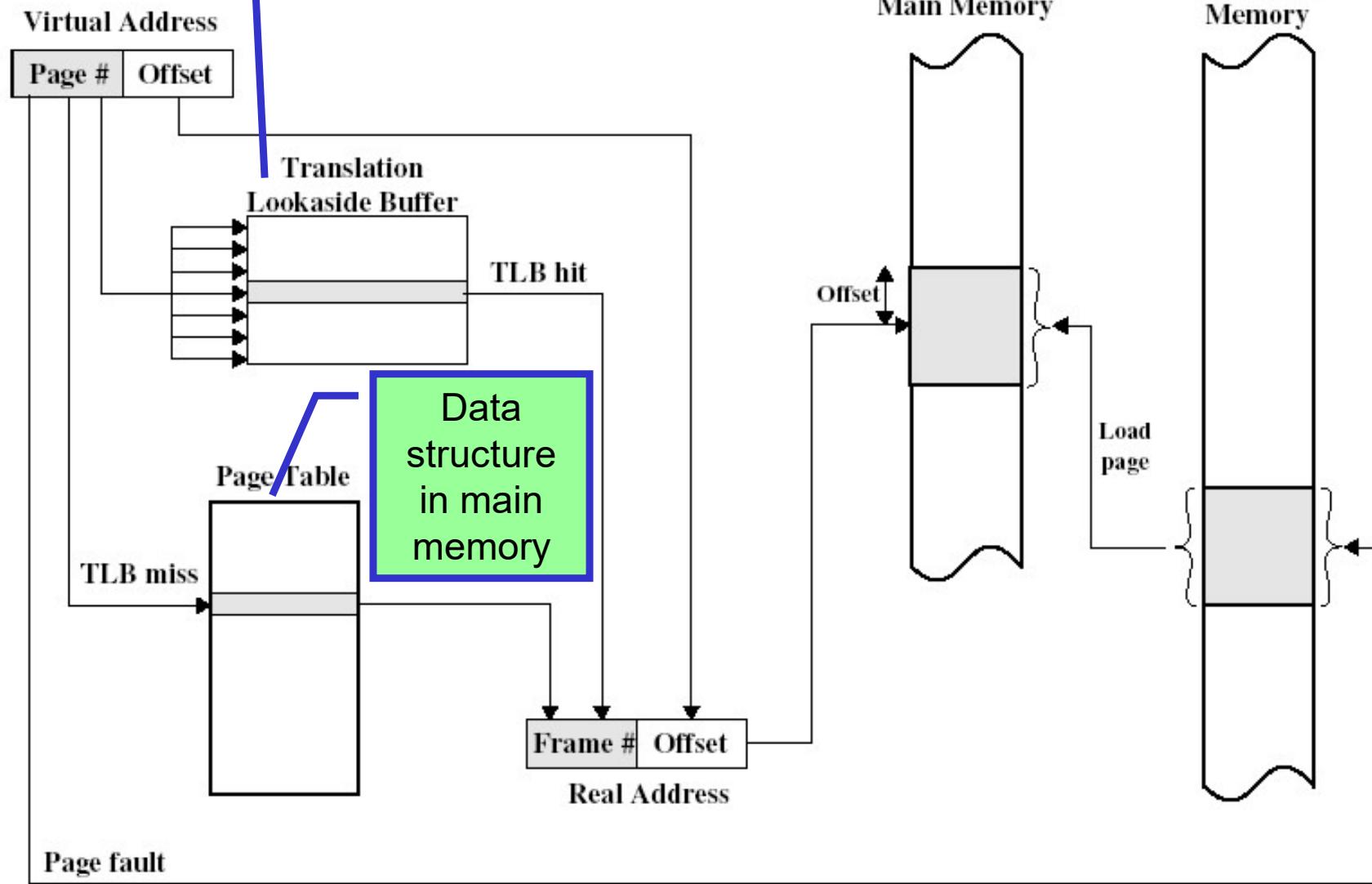
# VM Implementation Issue

- Performance?
  - Each virtual memory reference can cause two physical memory accesses
    - One to fetch the page table entry
    - One to fetch/store the data⇒Intolerable performance impact!!
- Solution:
  - High-speed cache for page table entries (PTEs)
    - Called a *translation look-aside buffer* (TLB)
    - Contains recently used page table entries
    - Associative, high-speed memory, similar to cache memory
    - May be under OS control (unlike memory cache)



On-CPU  
hardware  
device!!!

# TLB operation



# Translation Lookaside Buffer

- Given a virtual address, processor examines the TLB
- If matching PTE found (*TLB hit*), the address is translated
- Otherwise (*TLB miss*), the page number is used to index the process's page table
  - If PT contains a valid entry, reload TLB and restart
  - Otherwise, (page fault) check if page is on disk
    - If on disk, swap it in
    - Otherwise, allocate a new page or raise an exception



# TLB properties

- Page table is (logically) an array of frame numbers
- TLB holds a (recently used) subset of PT entries
  - Each TLB entry must be identified (tagged) with the page # it translates
  - Access is by associative lookup:
    - All TLB entries' tags are concurrently compared to the page #
    - TLB is associative (or content-addressable) memory

<i>page #</i>	<i>frame #</i>	<i>V</i>	<i>W</i>
...	...	.	.
...	...	.	.



# TLB properties

- TLB may or may not be under direct OS control
  - Hardware-loaded TLB
    - On miss, hardware performs PT lookup and reloads TLB
    - Example: x86, ARM
  - Software-loaded TLB
    - On miss, hardware generates a TLB miss exception, and exception handler reloads TLB
    - Example: MIPS, Itanium (optionally)
- TLB size: typically 64-128 entries
- Can have separate TLBs for instruction fetch and data access
- TLBs can also be used with inverted page tables (and others)



# TLB and context switching

- TLB is a shared piece of hardware
- Normal page tables are per-process (address space)
- TLB entries are *process-specific*
  - On context switch need to *flush* the TLB (invalidate all entries)
    - high context-switching overhead (Intel x86)
  - **or** tag entries with *address-space ID* (ASID)
    - called a *tagged TLB*
    - used (in some form) on all modern architectures
    - TLB entry: ASID, page #, frame #, valid and write-protect bits



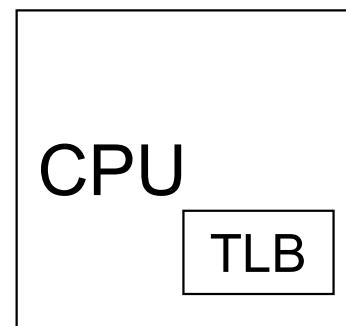
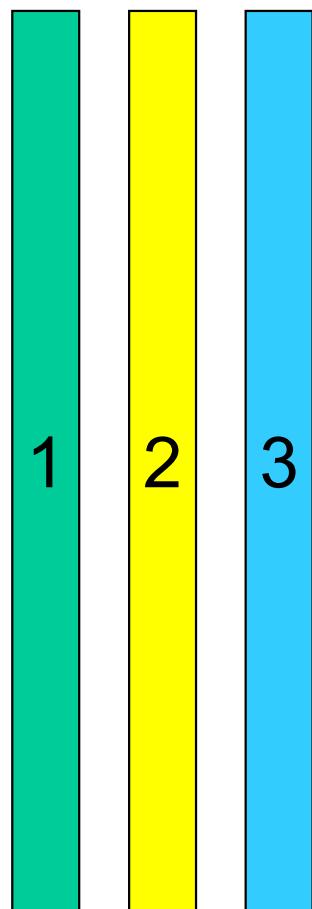
# TLB effect

- Without TLB
  - Average number of physical memory references per virtual reference  
= 2
- With TLB (assume 99% hit ratio)
  - Average number of physical memory references per virtual reference  
 $= .99 * 1 + 0.01 * 2$   
= 1.01



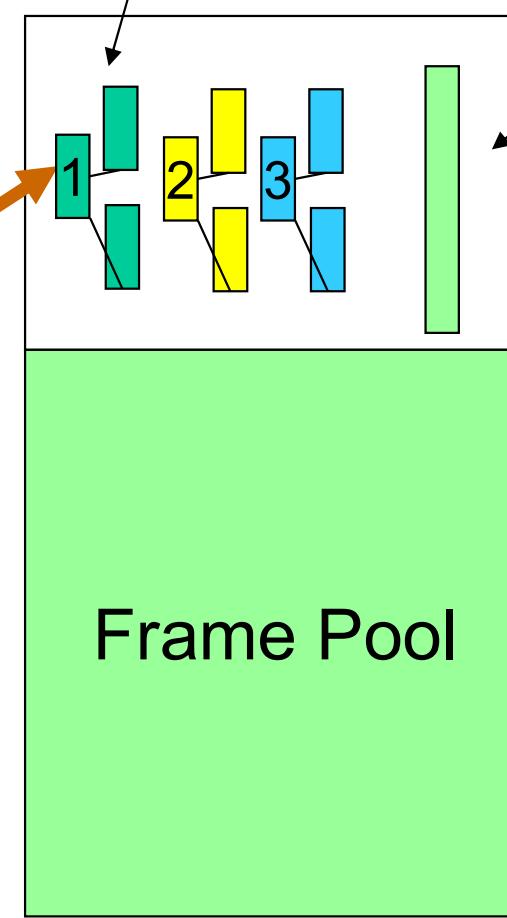
# Recap - Simplified Components of VM System

Virtual Address Spaces  
(3 processes)



TLB Refill Mechanism

Page Tables for 3 processes



Frame Table

Frame Pool

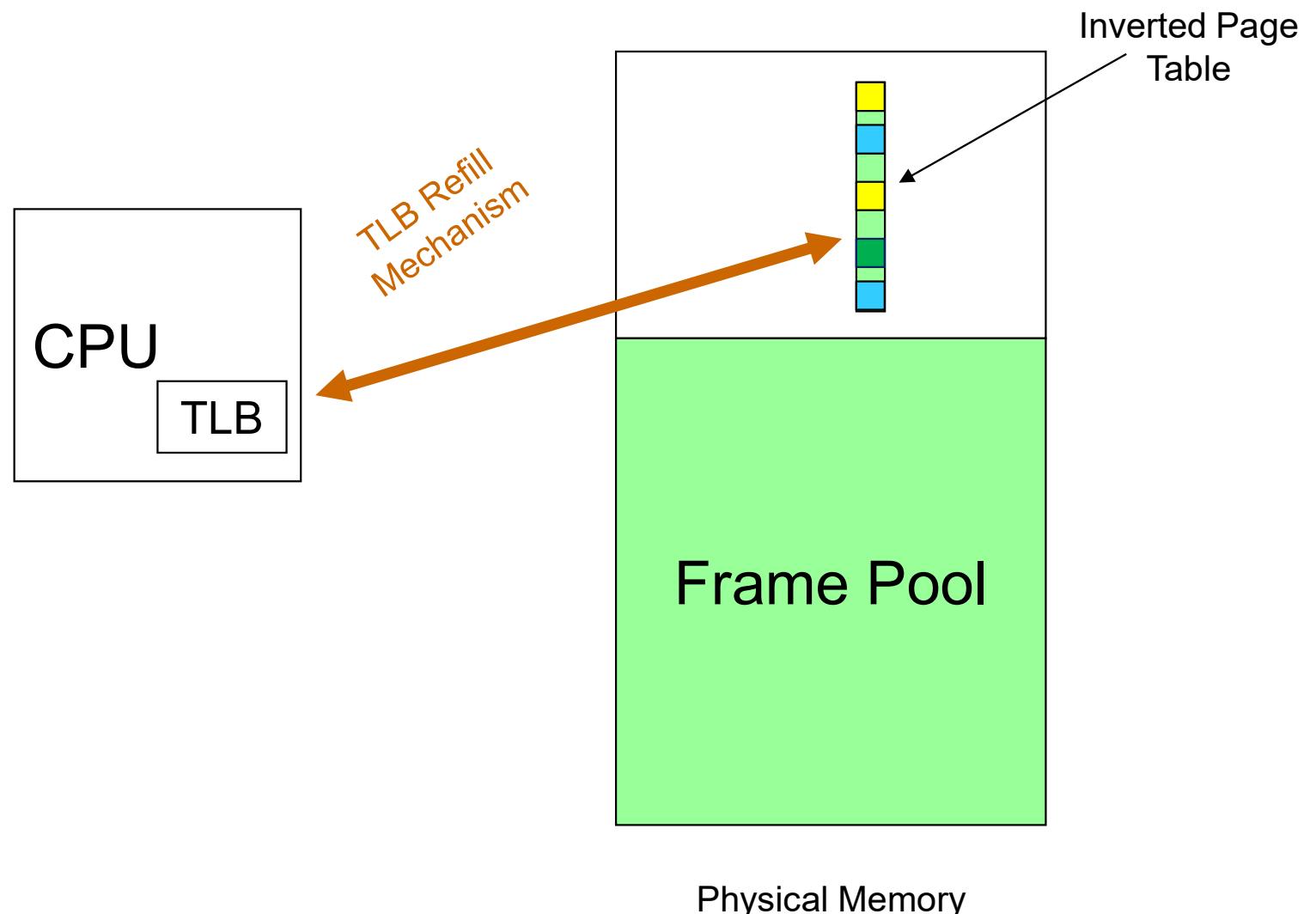
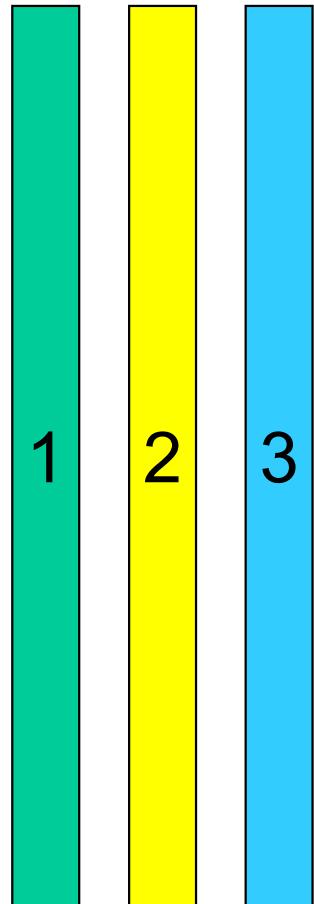
Physical Memory



THE UNIVERSITY OF  
NEW SOUTH WALES

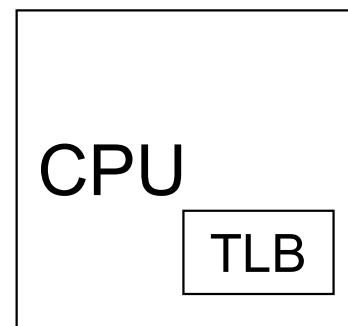
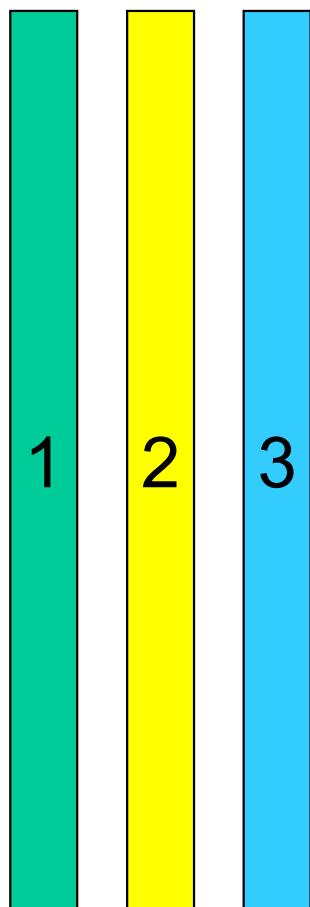
# Recap - Simplified Components of VM System

Virtual Address Spaces  
(3 processes)

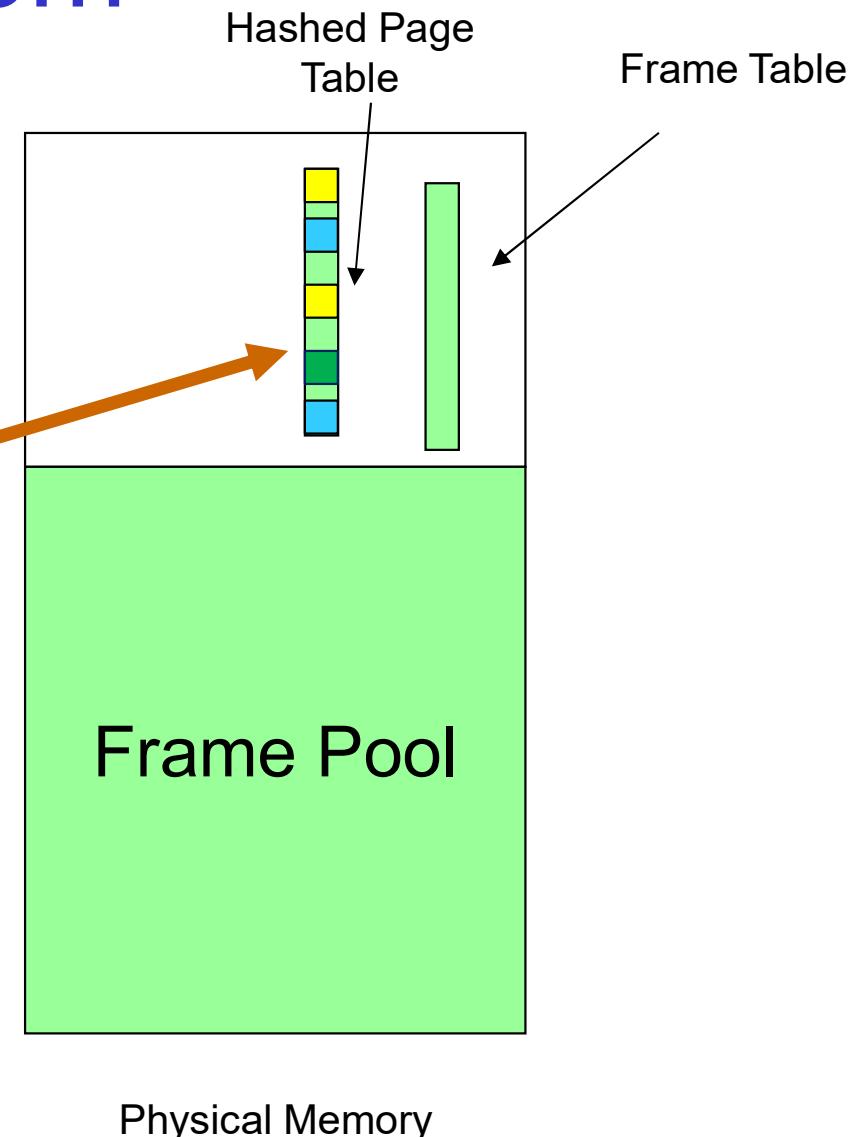


# Recap - Simplified Components of VM System

Virtual Address Spaces  
(3 processes)



TLB Refill  
Mechanism



# MIPS R3000 TLB

31                    12            11                    6            5                    0

VPN	ASID	0
-----	------	---

EntryHi Register (TLB key fields)

31                    12            11            10            9            8            7                    0

PFN	N	D	V	G	0
-----	---	---	---	---	---

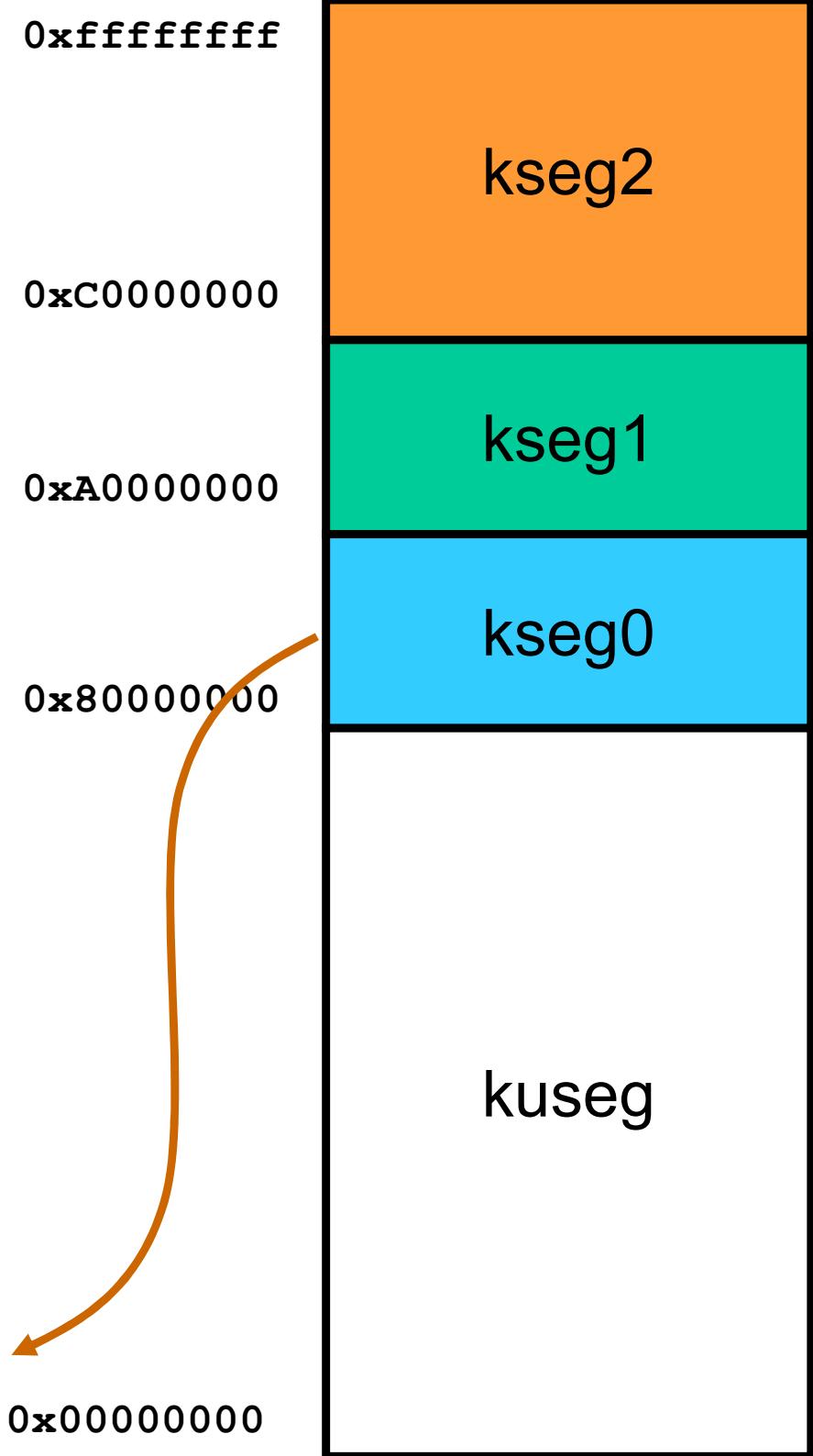
EntryLo Register (TLB data fields)

- N = Not cacheable
- D = Dirty = Write protect
- G = Global (ignore ASID in lookup)
- V = valid bit
- 64 TLB entries
- Accessed via software through Cooprocessor 0 registers
  - EntryHi and EntryLo



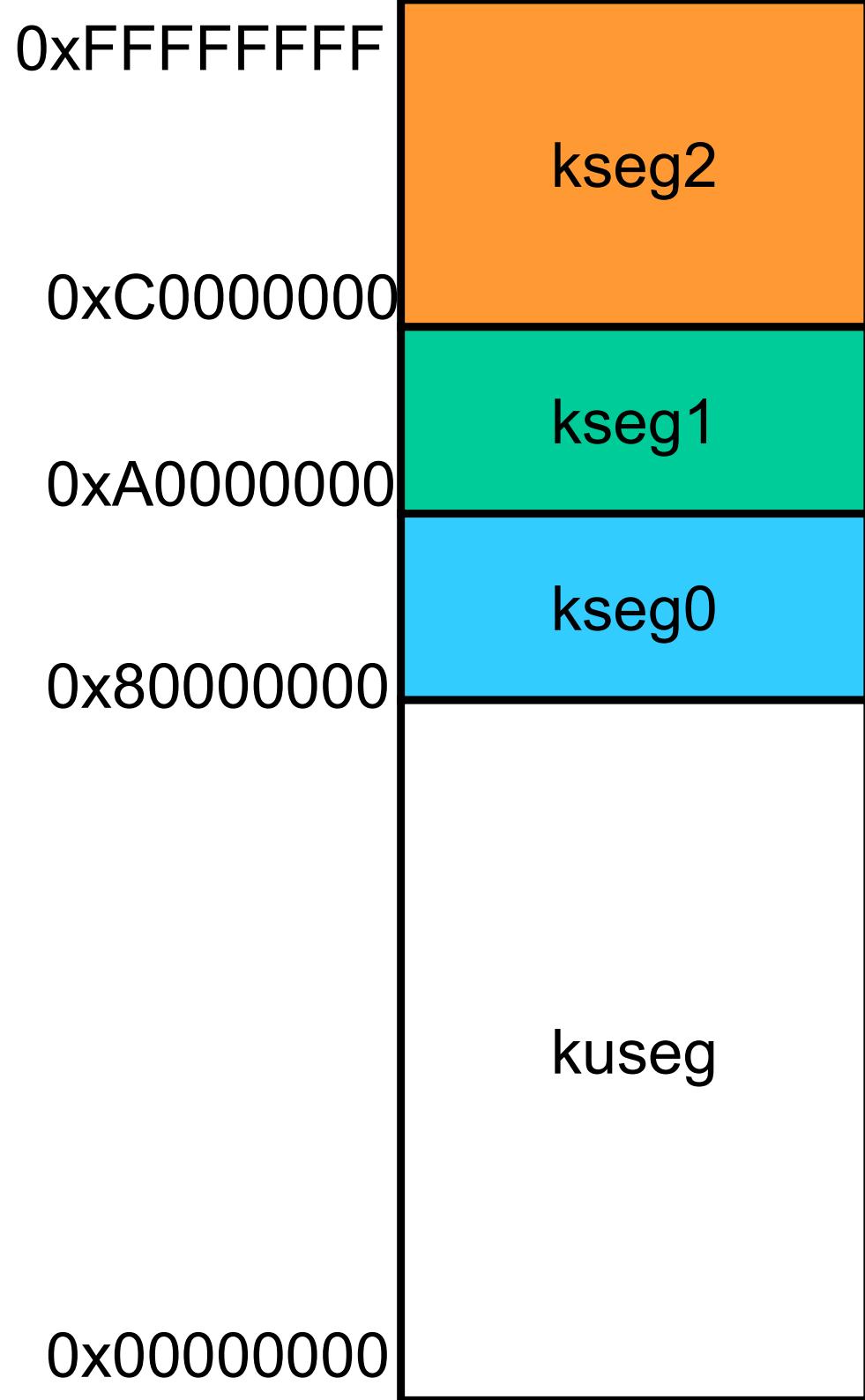
# R3000 Address Space Layout

- kseg0:
  - 512 megabytes
  - Fixed translation window to physical memory
    - $0x80000000 - 0xfffffff virtual = 0x00000000 - 0x1fffffff physical$
    - TLB not used
  - Cacheable
  - Only kernel-mode accessible
  - Usually where the kernel code and data is placed



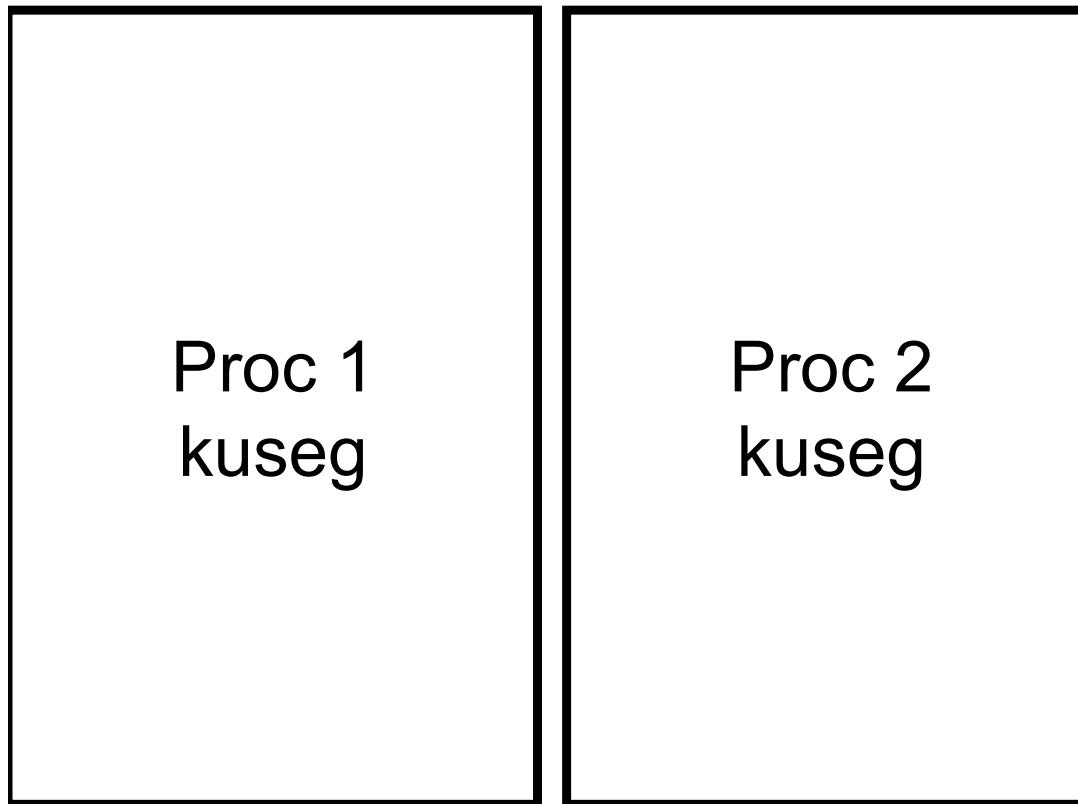
# R3000 Address Space Layout

- kuseg:
  - 2 gigabytes
  - TLB translated (mapped)
  - Cacheable (depending on ‘N’ bit)
  - user-mode and kernel mode accessible
  - Page size is 4K



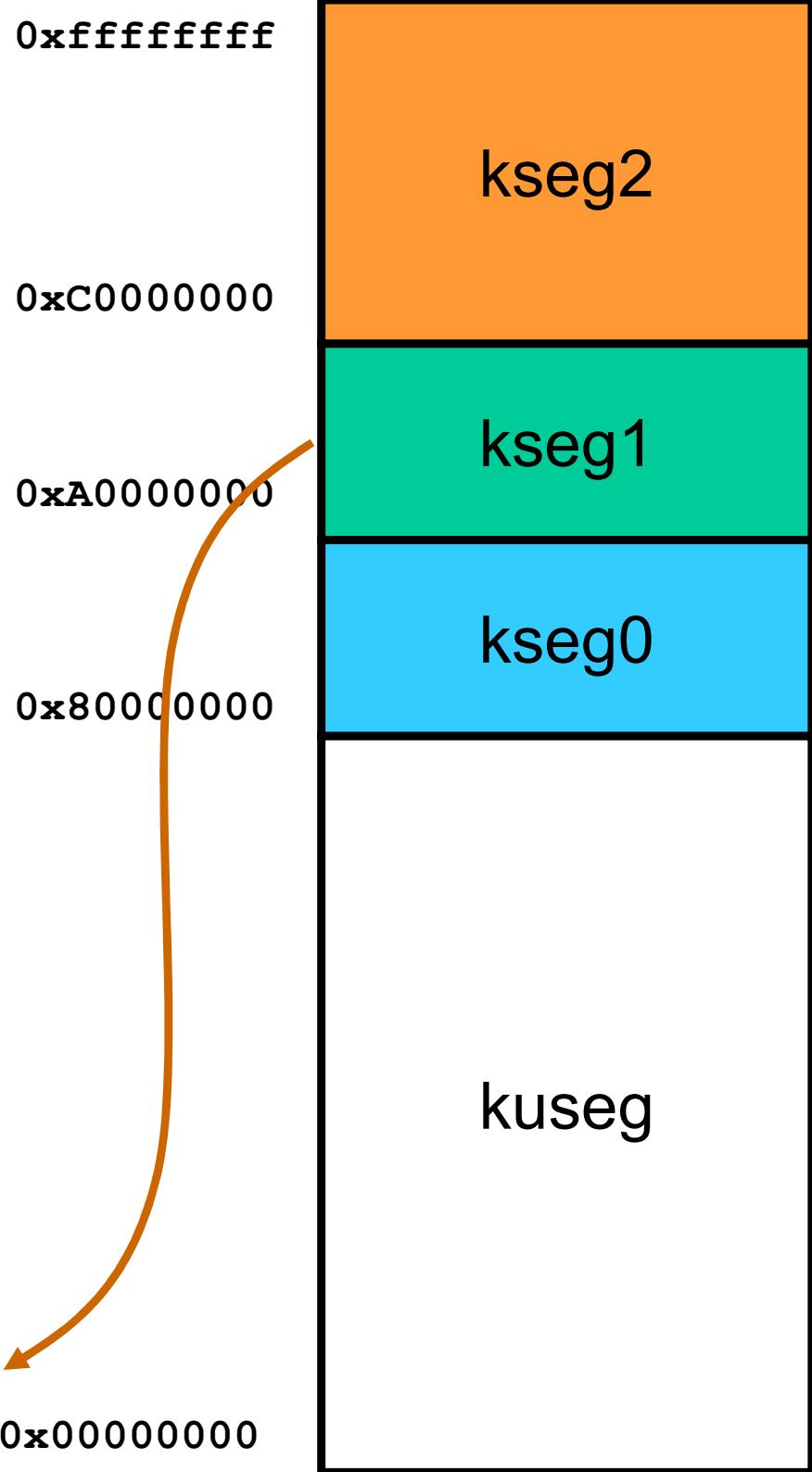
# R3000 Address Space Layout

- Switching processes switches the translation (page table) for kuseg



# R3000 Address Space Layout

- kseg1:
  - 512 megabytes
  - Fixed translation window to physical memory
    - $0xa0000000 - 0xbfffffff$  virtual =  $0x00000000 - 0x1fffffff$  physical
    - TLB not used
  - **NOT** cacheable
  - Only kernel-mode accessible
  - Where devices are accessed (and boot ROM)



# Virtual Memory II



# Learning Outcomes

- An understanding of TLB refill:
  - in general,
  - and as implemented on the R3000
- An understanding of demand-paged virtual memory in depth, including:
  - Locality and working sets
  - Page replacement algorithms
  - Thrashing



# TLB Recap

- Fast associative cache of page table entries
  - Contains a subset of the page table
  - What happens if required entry for translation is not present (a *TLB miss*)?



# TLB Recap

- TLB may or may not be under OS control
  - Hardware-loaded TLB
    - On miss, hardware performs PT lookup and reloads TLB
    - Example: Pentium
  - Software-loaded TLB
    - On miss, hardware generates a TLB miss exception, and exception handler reloads TLB
    - Example: MIPS



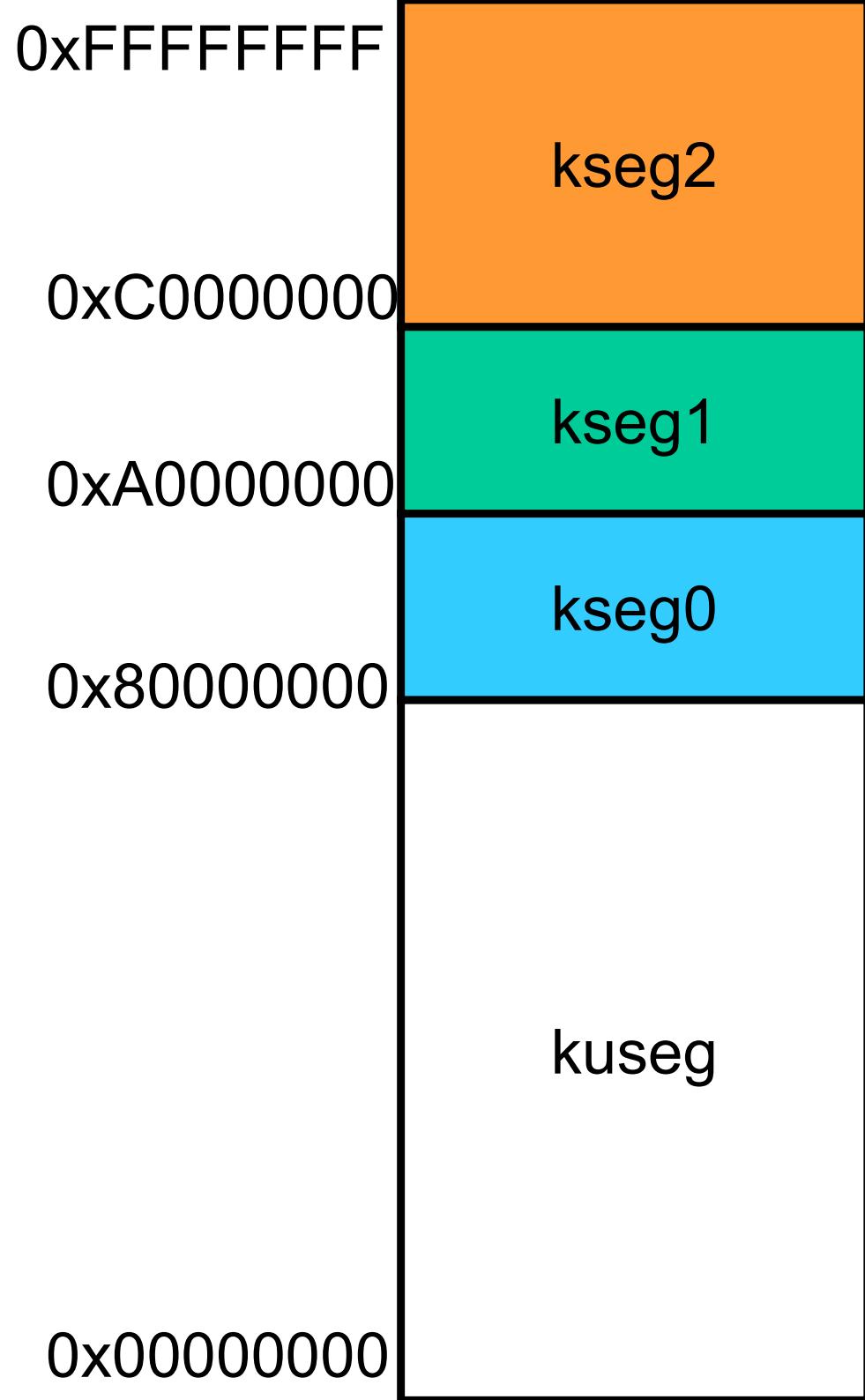
# Aside: even if filled by software

- TLB still a hardware-based translator



# R3000 TLB Handling

- TLB refill is handled by software
  - An exception handler
- TLB refill exceptions accessing kuseg are expected to be frequent
  - CPU optimised for handling kuseg TLB refills by having a special exception handler just for TLB refills



# Exception Vectors

Program address	"segment"	Physical Address	Description
0x8000 0000	kseg0	0x0000 0000	TLB miss on <i>kuseg</i> reference only.
0x8000 0080	kseg0	0x0000 0080	All other exceptions.
0xbfc0 0100	kseg1	0x1fc0 0100	Uncached alternative <i>kuseg</i> TLB miss entry point (used if SR bit BEV set).
0xbfc0 0180	kseg1		Alternative for all other exceptions, used if SR bit BEV set).
0xbfc0 0000	kseg1		Special exception vector for kuseg TLB refills

Table 4.1. Reset and exception vectors for R30xx family



# Special Exception Vector

- Can be optimised for TLB refill only
  - Does not need to check the exception type
  - Does not need to save any registers
    - It uses a specialised assembly routine that only uses k0 and k1.
  - Does not check if PTE exists
    - Assumes virtual linear array – see extended OS notes (if interested)
- With careful data structure choice, exception handler can be made very fast

- An example routine

```
mfc0 k1,C0_CONTEXT  
mfc0 k0,C0_EPC # mfc0 delay  
                      # slot  
lw k1,0(k1) # may double  
              # fault (k0 = orig EPC)  
nop  
mtc0 k1,C0_ENTRYLO  
nop  
tlbwr  
jr k0  
rfe
```



# MIPS VM Related Exceptions

- TLB refill
  - Handled via special exception vector
  - Needs to be very fast
- Others handled by the general exception vector
  - **TLB Mod**
    - TLB modify exception, attempt to write to a read-only page
  - **TLB Load**
    - Attempt to load from a page with an invalid translation
  - **TLB Store**
    - Attempt to store to a page with an invalid translation
  - Note: these can be slower as they are mostly either caused by an error, or non-resident page.
    - We never optimise for errors, and page-loads from disk dominate the fault resolution cost.



# <Intermezzo>



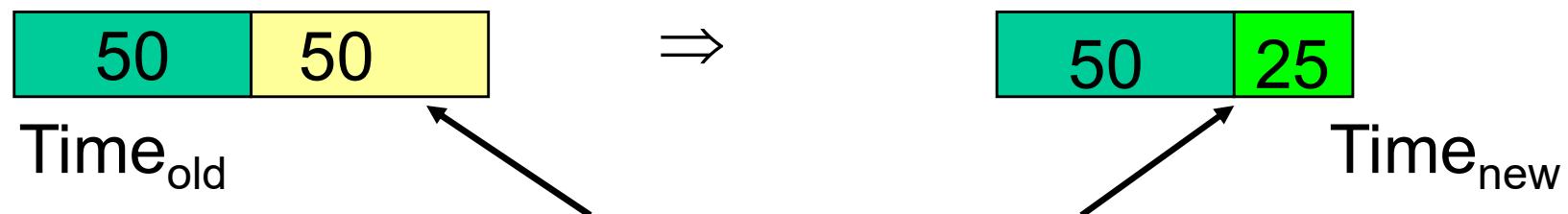
THE UNIVERSITY OF  
NEW SOUTH WALES

# Amdahl's law



- States that overall performance improvement is limited by the fraction of time an enhancement can be used

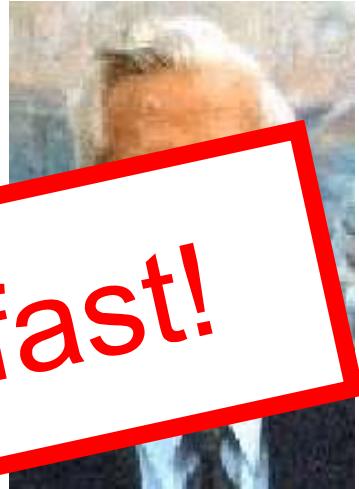
Law of diminishing returns



fraction in enhanced mode = 0.5 (based on old system)  
Speedup of enhanced mode = 2



# Amdahl's law



- States:

Make the common case fast!  
Speedup is limited by the fraction of enhancement can be used

$$\text{Speedup} = \frac{\text{ExecutionTimeWithoutEnhancement}}{\text{ExecutionTimeWithEnhancement}}$$



</Intermezzo>



THE UNIVERSITY OF  
NEW SOUTH WALES

# c0 Registers

- **c0\_EPC**
  - The address of where to restart after the exception
- **c0\_status**
  - Kernel/User Mode bits, Interrupt control
- **c0\_cause**
  - What caused the exception
- **c0\_badvaddr**
  - The address of the fault



# The TLB and EntryHi,EntryLo

## c0 Registers

c0_EntryHi
c0_EntryLo
c0_Index

Used to read  
and write  
individual TLB  
entries

- Each TLB entry contains
- EntryHi to match page# and ASID
  - EntryLo which contains frame# and protection

## TLB

EntryHi	EntryLo



# c0 Registers

31                    12            11                    6            5                    0

VPN	ASID	0
-----	------	---

EntryHi Register (TLB key fields)

31                    12            11            10            9            8            7                    0

PFN	N	D	V	G	0
-----	---	---	---	---	---

EntryLo Register (TLB data fields)

- N = Not cacheable
- D = Dirty = Write protect
- G = Global (ignore ASID in lookup)
- V = valid bit
- 64 TLB entries
- Accessed via software through Coprocessor 0 registers
  - EntryHi and EntryLo

# c0 Index Register

- Used as an index to TLB entries
  - Single TLB entries are manipulated/viewed through EntryHi and EntryLo0 registers
  - Index register specifies which TLB entry to change/view



# Special TLB management Instructions

- ***TLBR***
  - TLB read
    - EntryHi and EntryLo are loaded from the entry pointer to by the index register.
- ***TLBP***
  - TLB probe
  - Set EntryHi to the entry you wish to match, index register is loaded with the index to the matching entry
- ***TLBWR***
  - Write EntryHi and EntryLo to a psuedo-random location in the TLB
- ***TLBWI***
  - Write EntryHi and EntryLo to the location in the TLB pointed to by the Index register.



# Coprocessor 0 registers on a refill exception

c0.EPC  $\leftarrow$  PC

c0.cause.ExcCode  $\leftarrow$  TLBL ; if read fault

c0.cause.ExcCode  $\leftarrow$  TLBS ; if write fault

c0.BadVaddr  $\leftarrow$  faulting address

c0.EntryHi.VPN  $\leftarrow$  page number of faulting address

c0.status  $\leftarrow$  kernel mode, interrupts disabled.

c0.PC  $\leftarrow$  0x8000 0000



# Outline of TLB miss handling

- Software does:
  - Look up PTE corresponding to the faulting address
  - If found:
    - load c0\_EntryLo with translation
    - load TLB using TLBWR instruction
    - return from exception
  - Else, page fault
- The TLB entry (i.e. c0\_EntryLo) can be:
  - (theoretically) created on the fly, or
  - stored completely in the right format in page table
    - more efficient



# OS/161 Refill Handler

- After switch to kernel stack, it simply calls the common exception handler
  - Stacks all registers
  - Can (and does) call ‘C’ code
  - Unoptimised
  - Goal is ease of kernel programming, not efficiency
- Does not have a page table
  - It uses the 64 TLB entries and then panics when it runs out.
    - Only support 256K user-level address space



# Demand Paging



# Demand Paging

- With VM, only parts of the program image need to be resident in memory for execution.
- Can transfer presently unused pages/segments to disk
- Reload non-resident pages/segment *on demand*.
  - Reload is triggered by a page or segment fault
  - Faulting process is blocked and another scheduled
  - When page/segment is resident, faulting process is restarted
  - May require freeing up memory first
    - Replace current resident page/segment
    - How determine replacement “victim”?
  - If victim is unmodified (“clean”) can simply discard it
    - This is reason for maintaining a “dirty” bit in the PT



- Why does demand paging work?
  - Program executes at full speed only when accessing the resident set.
  - TLB misses introduce delays of several microseconds
  - Page/segment faults introduce delays of several milliseconds
  - Why do it?
- Answer
  - Less physical memory required per process
    - Can fit more processes in memory
    - Improved chance of finding a runnable one
  - Principle of locality



# Principle of Locality

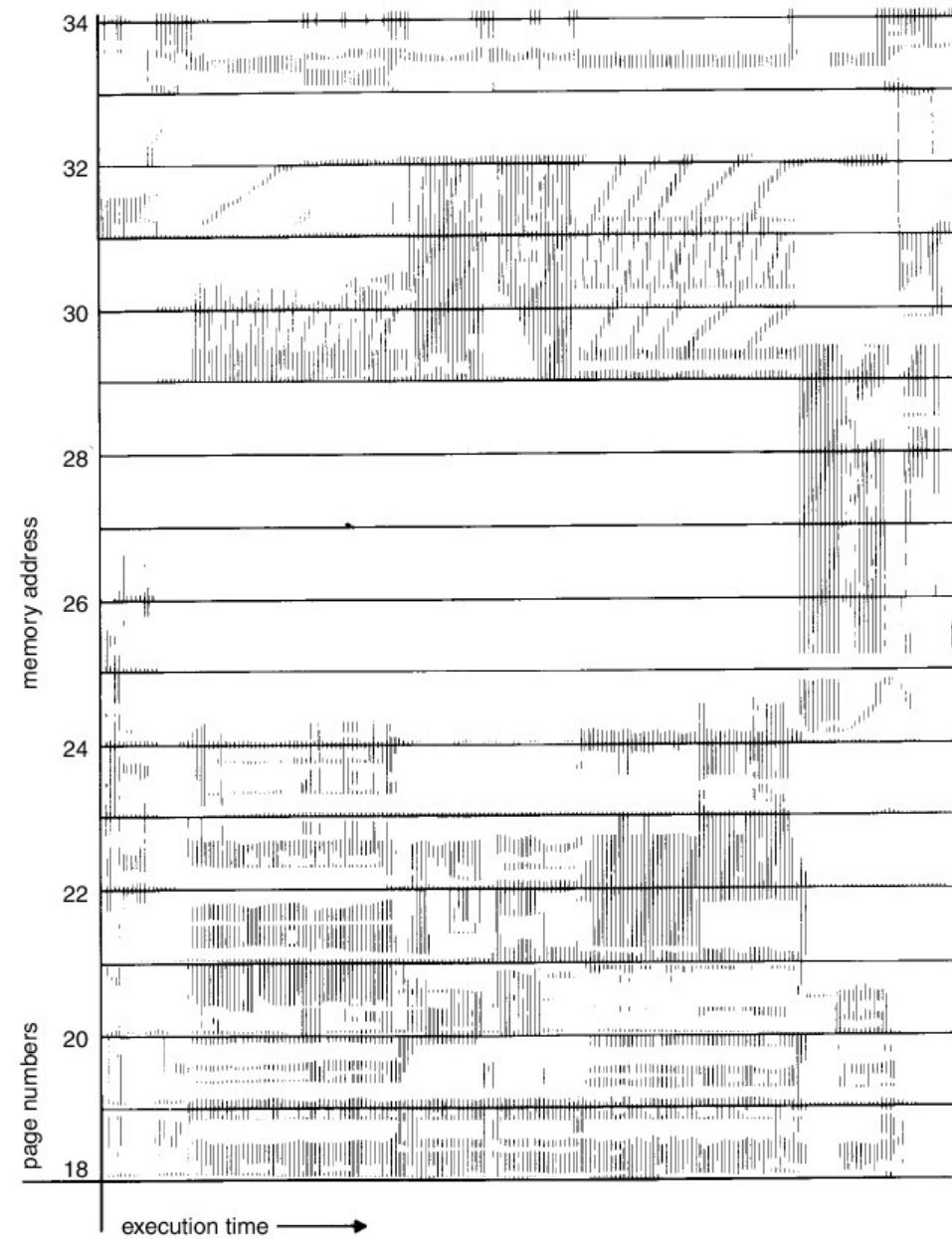
- An important observation comes from empirical studies of the properties of programs.
  - Programs tend to reuse data and instructions they have used recently.
  - **90/10 rule**  
*"A program spends 90% of its time in 10% of its code"*
- We can exploit this locality of references
- An implication of locality is that we can reasonably predict what instructions and data a program will use in the near future based on its accesses in the recent past.



- **Two different types** of locality have been observed:
  - ***Temporal*** locality: states that recently accessed items are likely to be accessed in the near future.
  - ***Spatial*** locality: says that items whose addresses are near one another tend to be referenced close together in time.



# Locality In A Memory-Reference Pattern

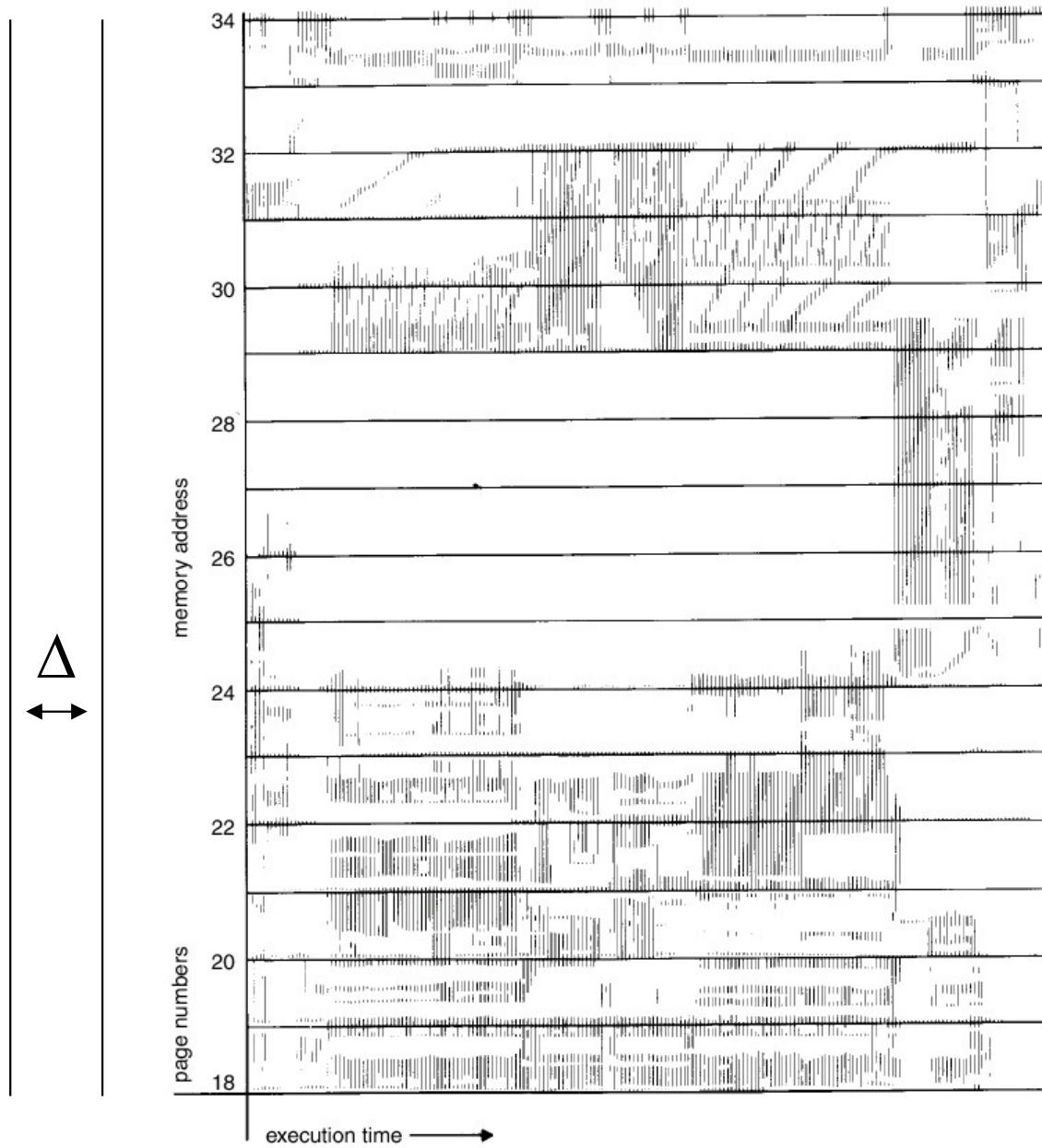


# Working Set

- The pages/segments required by an application in a time window ( $\Delta$ ) is called its memory ***working set***.
- Working set is an approximation of a programs' locality
  - if  $\Delta$  too small will not encompass entire locality.
  - if  $\Delta$  too large will encompass several localities.
  - if  $\Delta = \infty \Rightarrow$  will encompass entire program.
  - $\Delta$ 's size is an application specific tradeoff
- System should keep resident at least a process's working set
  - Process executes while it remains in its working set
- Working set tends to change gradually
  - Get only a few page/segment faults during a time window
  - Possible (but hard) to make intelligent guesses about which pieces will be needed in the future
    - May be able to pre-fetch page/segments



# Working Set Example

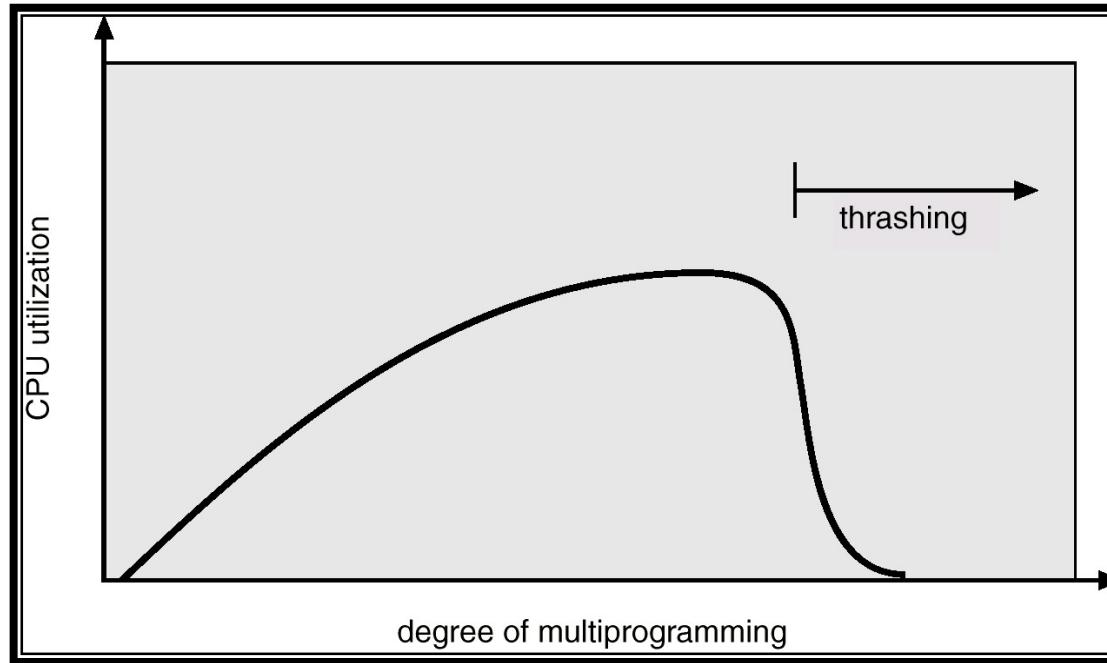


# Thrashing

- CPU utilisation tends to increase with the degree of multiprogramming
  - number of processes in system
- Higher degrees of multiprogramming – less memory available per process
- Some process's working sets may no longer fit in RAM
  - Implies an increasing page fault rate
- Eventually many processes have insufficient memory
  - Can't always find a runnable process
  - Decreasing CPU utilisation
  - System become I/O limited
- This is called ***thrashing***.



# Thrashing



- Why does thrashing occur?  
 $\Sigma$  working set sizes > total physical memory size



# Recovery From Thrashing

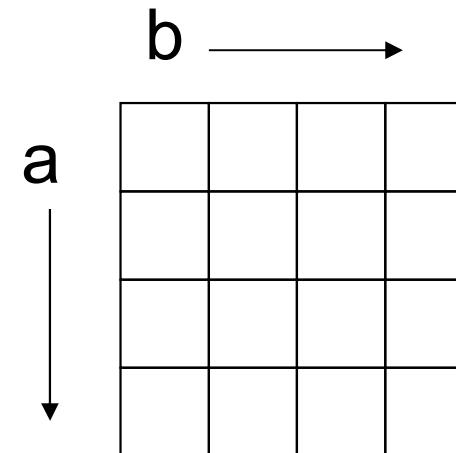
- In the presence of increasing page fault frequency and decreasing CPU utilisation
  - Suspend a few processes to reduce degree of multiprogramming
  - Resident pages of suspended processes will migrate to backing store
  - More physical memory becomes available
    - Less faults, faster progress for runnable processes
  - Resume suspended processes later when memory pressure eases



# What is the difference?

```
/* reset array */  
int array[10000][10000];  
int i,j;  
for (i = 0; i < 10000; i++) {  
    for (j = 0; j < 10000;j++) {  
        array[i][j] = 0;  
        /* array[j][i] = 0 */  
    }  
}
```

Array[a][b]



# VM Management Policies



# VM Management Policies

- Operation and performance of VM system is dependent on a number of policies:
  - Page table format (may be dictated by hardware)
    - Multi-level
    - Inverted/Hashed
  - Page size (may be dictated by hardware)
  - Fetch Policy
  - Replacement policy
  - Resident set size
    - Minimum allocation
    - Local versus global allocation
  - Page cleaning policy



# Page Size

## Increasing page size

- ✗ Increases internal fragmentation
  - reduces adaptability to working set size
- ✓ Decreases number of pages
  - Reduces size of page tables
- ✓ Increases TLB coverage
  - Reduces number of TLB misses
- ✗ Increases page fault latency
  - Need to read more from disk before restarting process
- ✓ Increases swapping I/O throughput
  - Small I/O are dominated by seek/rotation delays
- Optimal page size is a (work-load dependent) trade-off.



# Working Set Size Generally Increases with Increasing Page Size: True/False?



Atlas	512 words (48-bit)
Honeywell/Multics	1K words (36-bit)
IBM 370/XA	4K bytes
DEC VAX	512 bytes
IBM AS/400	512 bytes
Intel Pentium	4K and 4M bytes
ARM	4K and 64K bytes
MIPS R4000	4k – 16M bytes in powers of 4
DEC Alpha	8K - 4M bytes in powers of 8
UltraSPARC	8K – 4M bytes in powers of 8
PowerPC	4K bytes + “blocks”
Intel IA-64	4K – 256M bytes in powers of 4



# Page Size

- Multiple page sizes provide flexibility to optimise the use of the TLB
- Example:
  - Large page sizes can be used for code
  - Small page size for thread stacks
- Most operating systems have limited support for only a single page size
  - Dealing with multiple page sizes is hard!



# Fetch Policy

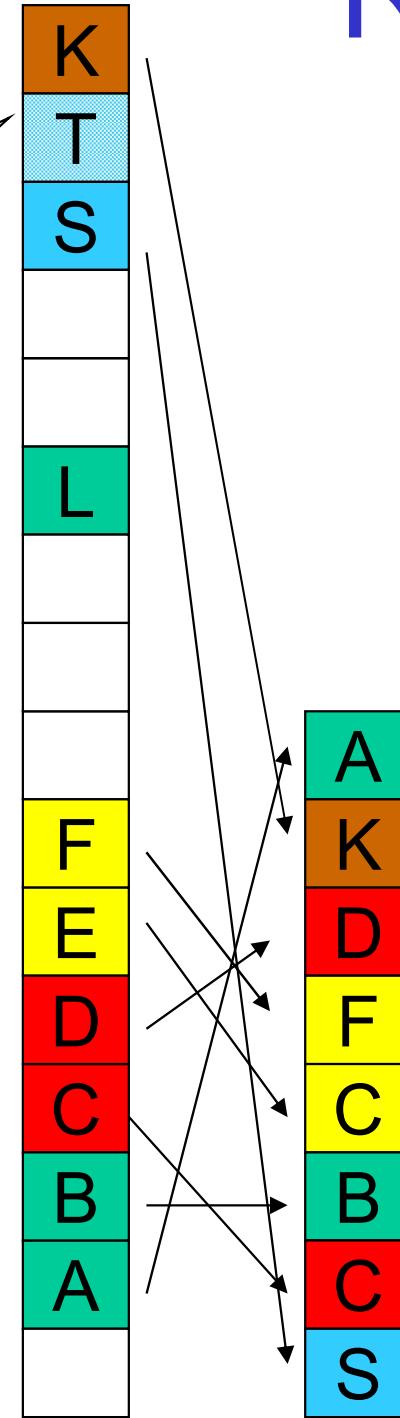
- Determines *when* a page should be brought into memory
  - *Demand paging* only loads pages in response to page faults
    - Many page faults when a process first starts
  - *Pre-paging* brings in more pages than needed at the moment
    - Pre-fetch when disk is idle
    - Wastes I/O bandwidth if pre-fetched pages aren't used
    - Especially bad if we eject pages in working set in order to pre-fetch unused pages.
    - Hard to get right in practice.



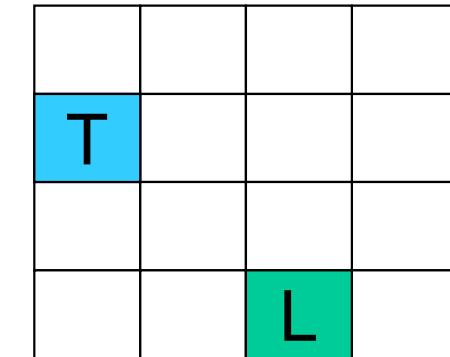
# Replacement Policy

Page fault on page 14, physical memory full, which page should we evict?

Virtual  
Memory



Physical Address  
Space



Disk



THE UNIVERSITY OF  
NEW SOUTH WALES

# Replacement Policy

- Which page is chosen to be tossed out?
  - Page removed should be the page least likely to be references in the near future
  - Most policies attempt to predict the future behaviour on the basis of past behaviour
- Constraint: locked frames
  - Kernel code
  - Main kernel data structure
  - I/O buffers
  - Performance-critical user-pages (e.g. for DBMS)
- Frame table has a *lock* (or *pinned*) bit



# *Optimal* Replacement policy

- Toss the page that won't be used for the longest time
- Impossible to implement
- Only good as a theoretic reference point:
  - The closer a practical algorithm gets to *optimal*, the better
- Example:
  - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - Four frames
  - How many page faults?



# FIFO Replacement Policy

- First-in, first-out: Toss the oldest page
  - Easy to implement
  - Age of a page is isn't necessarily related to usage
- Example:
  - Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
  - Four frames



# *Least Recently Used (LRU)*

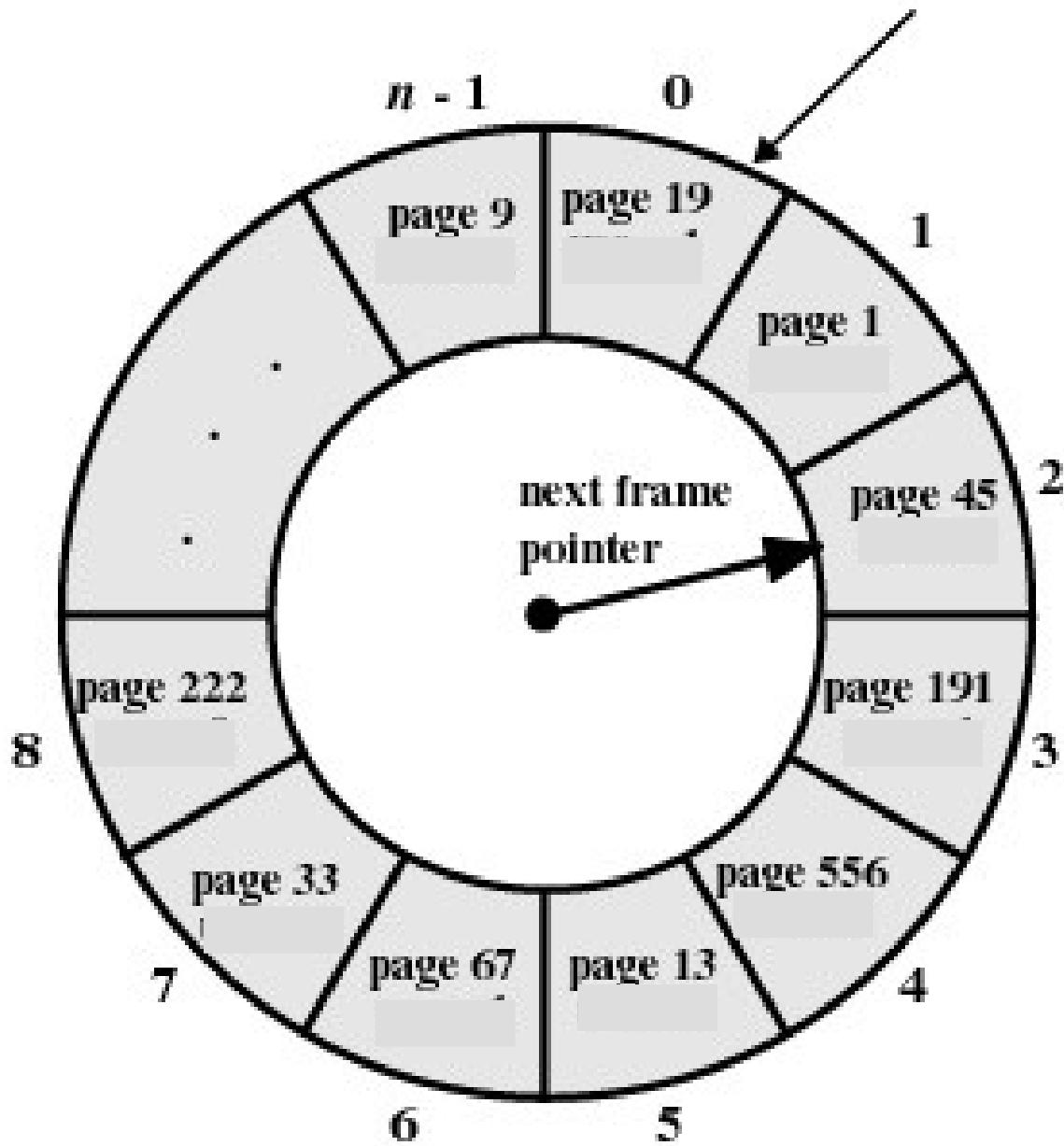
- Toss the least recently used page
  - Assumes that page that has not been referenced for a long time is unlikely to be referenced in the near future
  - Will work if locality holds
  - Implementation requires a time stamp to be kept for each page, updated **on every reference**
  - Impossible to implement efficiently
  - Most practical algorithms are approximations of LRU



# Clock Page Replacement

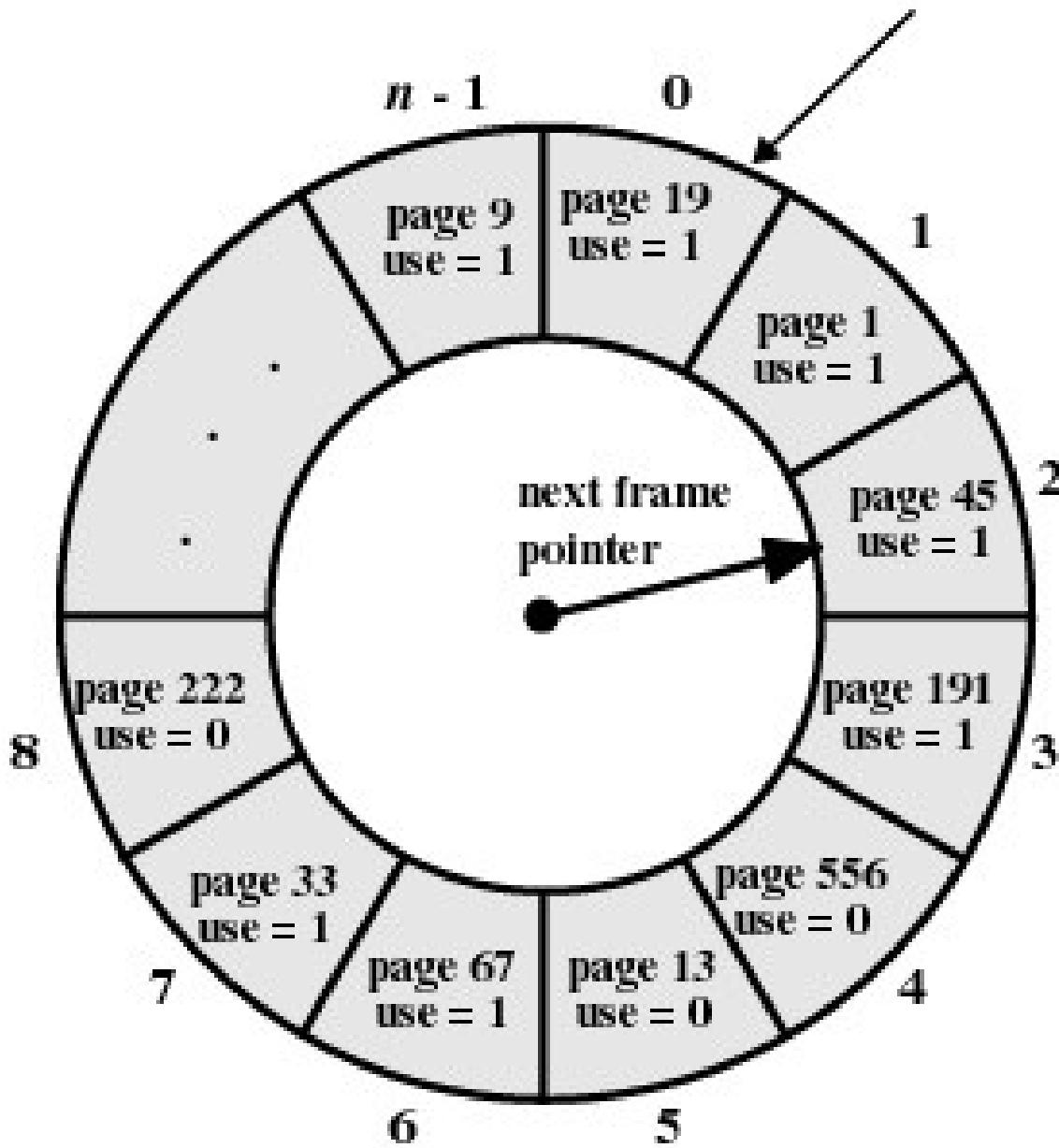
- Clock policy, also called *second chance*
  - Employs a *usage* or *reference* bit in the frame table.
  - Set to *one* when page is used
  - While scanning for a victim, reset all the reference bits
  - Toss the first page with a zero reference bit.





(a) State of buffer just prior to a page replacement

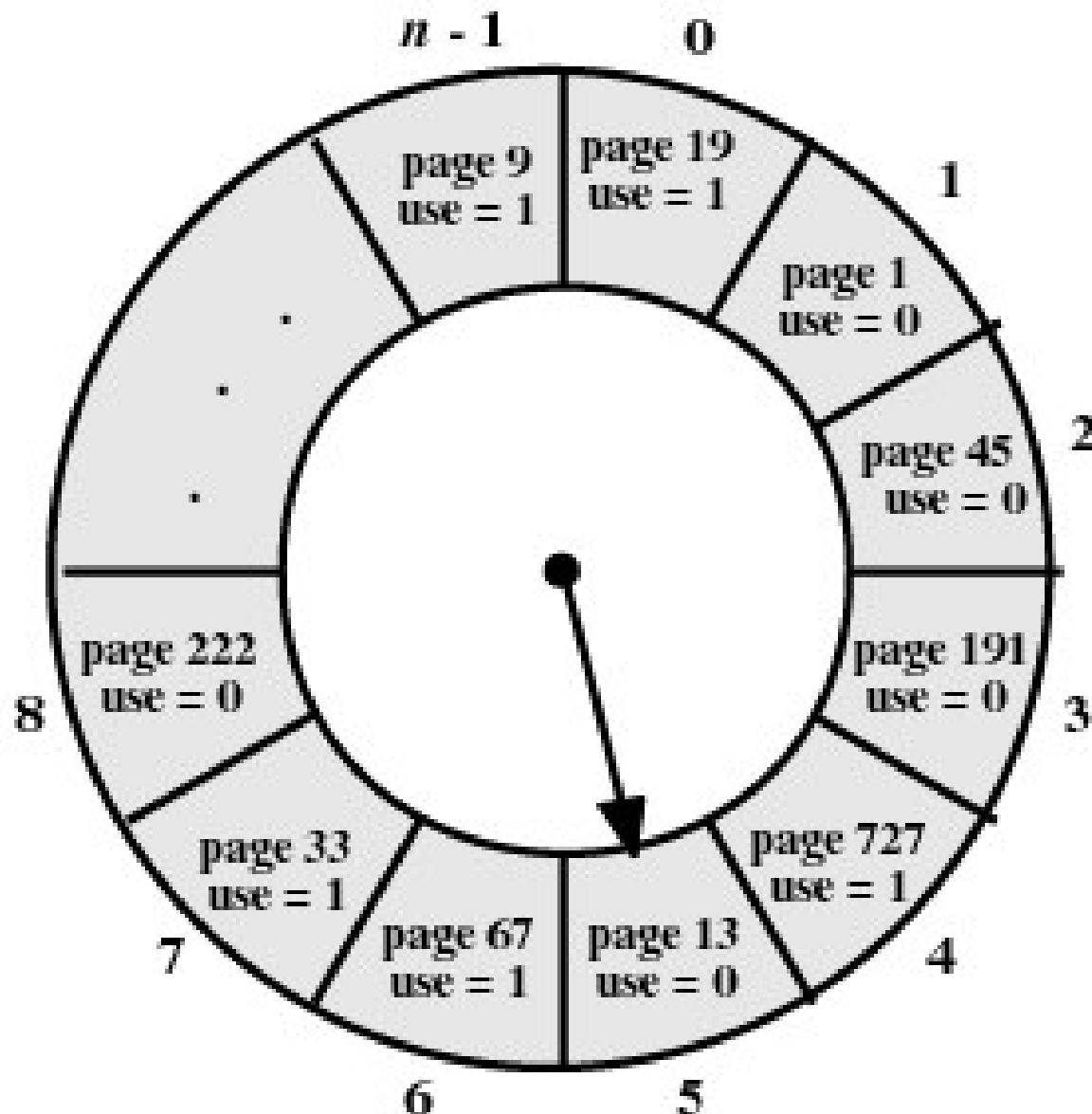
**Figure 8.16 Example of Clock Policy Operation**



Assume a page fault on page 727

(a) State of buffer just prior to a page replacement

**Figure 8.16 Example of Clock Policy Operation**



(b) State of buffer just after the next page replacement

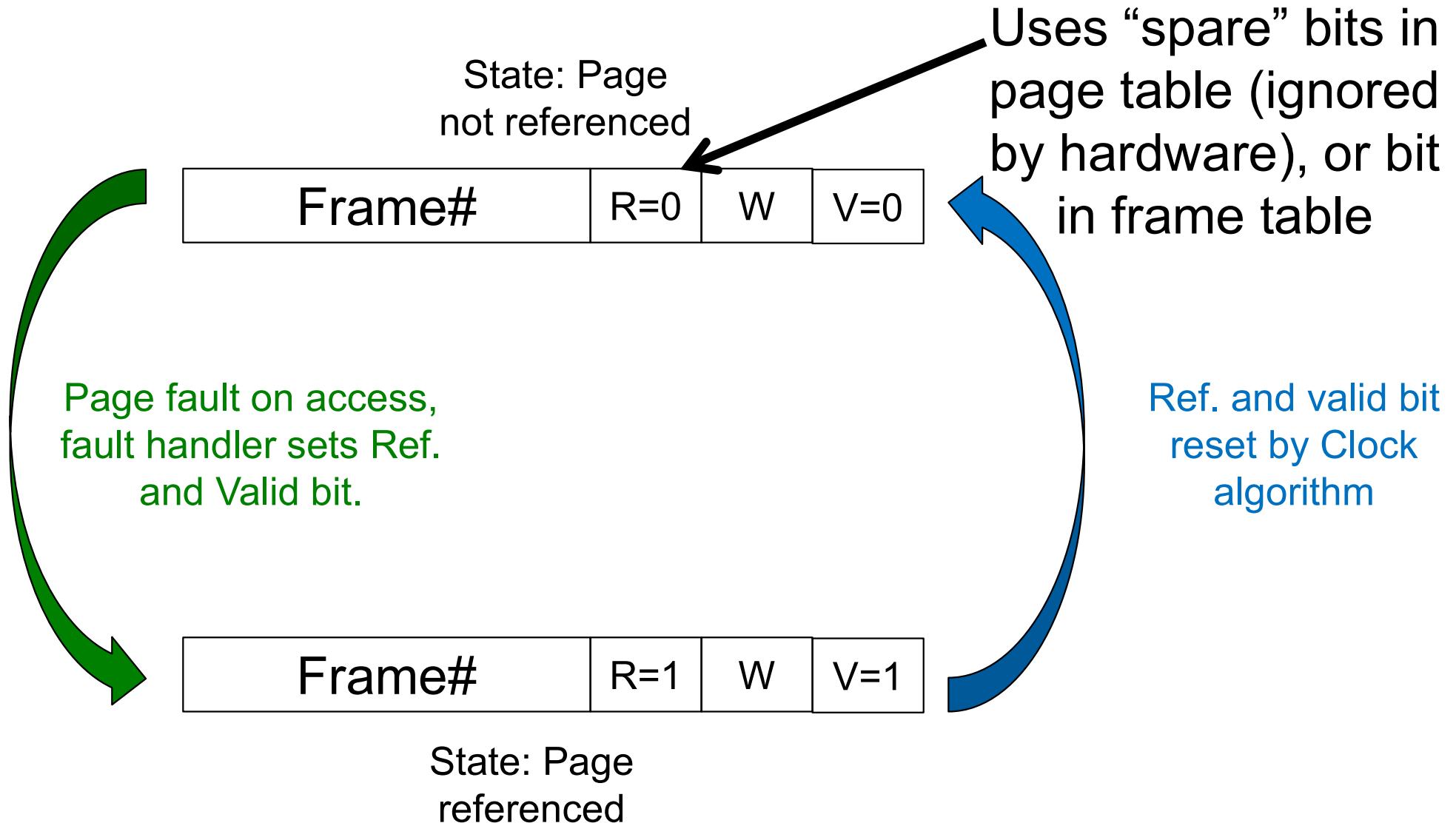
**Figure 8.16 Example of Clock Policy Operation**

# Issue

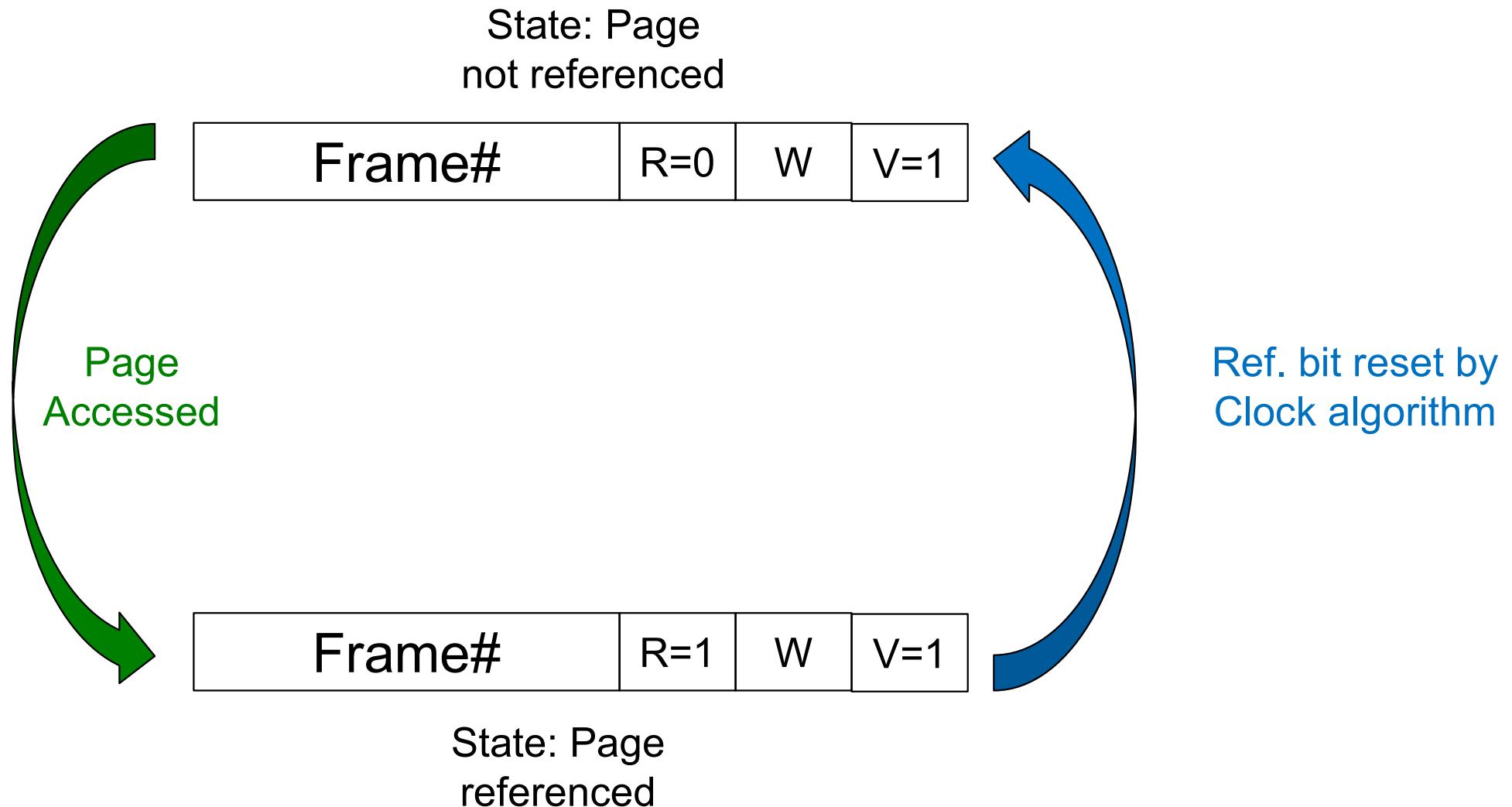
- How do we know when a page is referenced?
- Use the valid bit in the PTE:
  - When a page is mapped (valid bit set), set the reference bit
  - When resetting the reference bit, invalidate the PTE entry
  - On page fault
    - Turn on valid bit in PTE
    - Turn on reference bit
- We thus simulate a reference bit in software



# Simulated Reference Bit



# Hardware Reference Bit



# Performance

- In terms of selecting the most appropriate replacement, they rank as follows
  1. Optimal
  2. LRU
  3. Clock
  4. FIFO
- Note there are other algorithms (Working Set, WSclock, Ageing, NFU, NRU)
  - We don't expect you to know them in this course



# Resident Set Size

- How many frames should each process have?
  - *Fixed Allocation*
    - Gives a process a fixed number of pages within which to execute.
    - Isolates process memory usage from each other
    - When a page fault occurs, one of the pages of that process must be replaced.
    - Achieving high utilisation is an issue.
      - Some processes have high fault rate while others don't use their allocation.
  - *Variable Allocation*
    - Number of pages allocated to a process varies over the lifetime of the process



# Variable Allocation, Global Scope

- Easiest to implement
- Adopted by many operating systems
- Operating system keeps global list of free frames
- Free frame is added to resident set of process when a page fault occurs
- If no free frame, replaces one from any process
- Pro/Cons
  - Automatic balancing across system
  - Does not provide guarantees for important activities

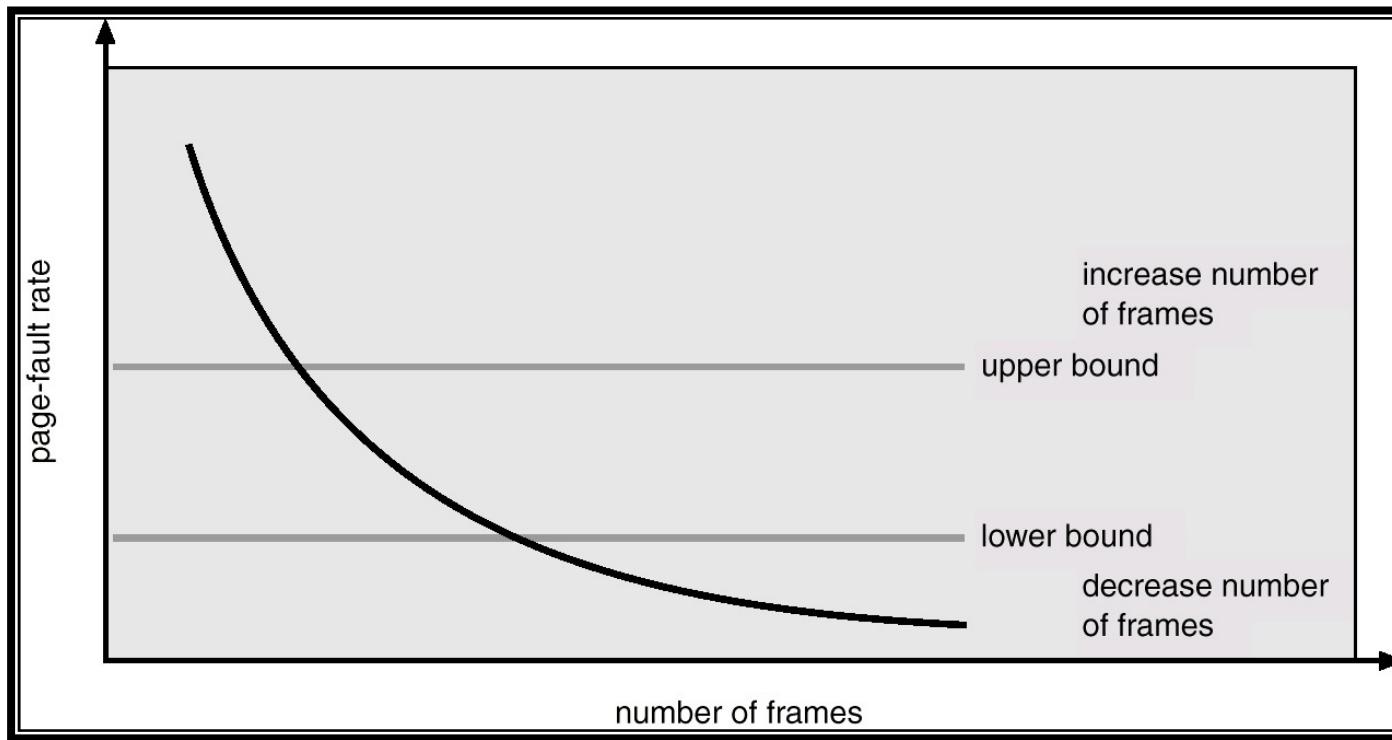


# Variable Allocation, Local Scope

- Allocate number of page frames to a new process based on
  - Application type
  - Program request
  - Other criteria (priority)
- When a page fault occurs, select a page from among the resident set of the process that suffers the page fault
- *Re-evaluate allocation from time to time!*



# Page-Fault Frequency Scheme



- Establish “acceptable” page-fault rate.
  - If actual rate too low, process loses frame.
  - If actual rate too high, process gains frame.



# Cleaning Policy

- Observation
  - Clean pages are much cheaper to replace than dirty pages
- Demand cleaning
  - A page is written out only when it has been selected for replacement
  - High latency between the decision to replace and availability of free frame.
- Precleaning
  - Pages are written out in batches (in the background, the *pagedaemon*)
  - Increases likelihood of replacing clean frames
  - Overlap I/O with current activity



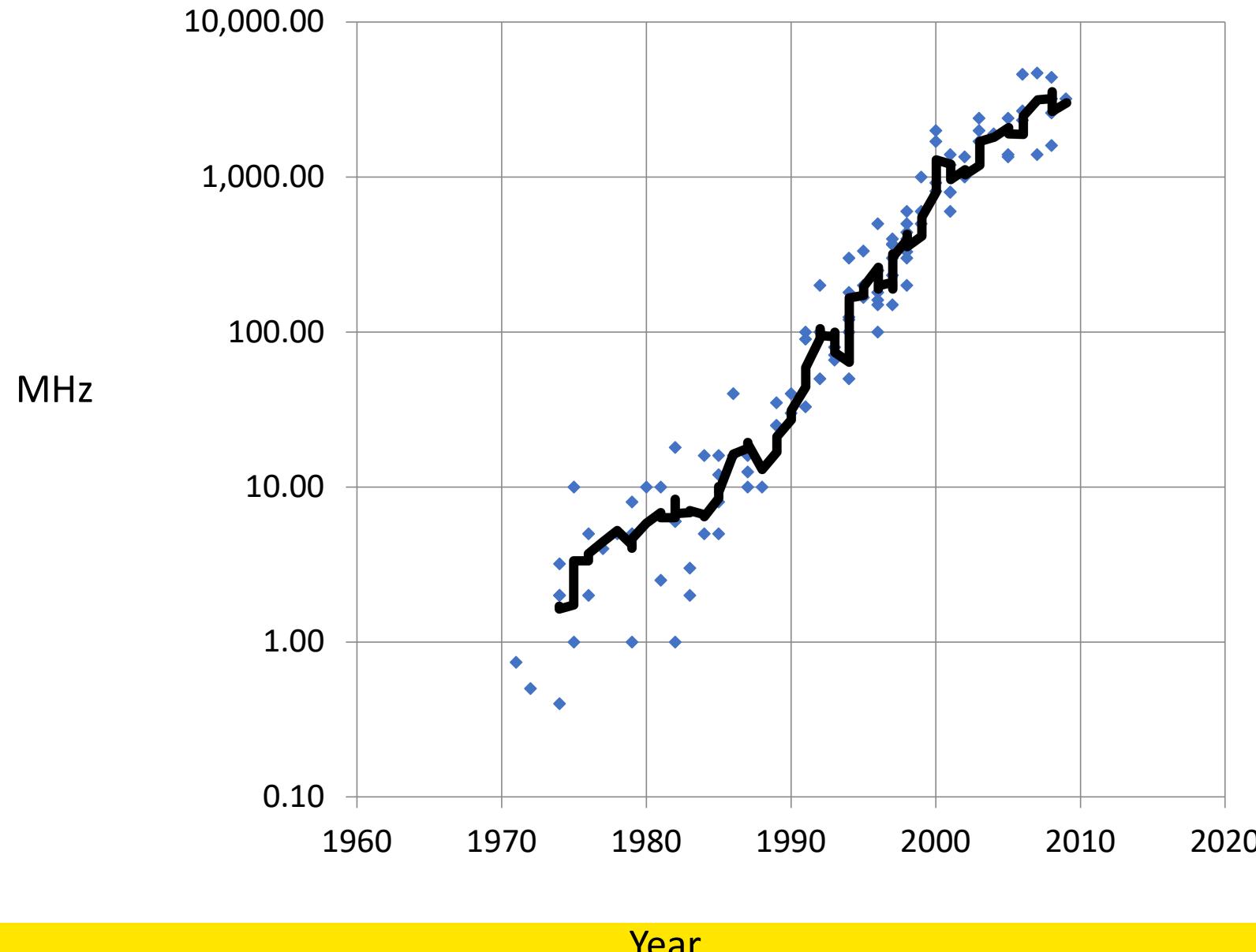
# Multiprocessor Systems

Chapter 8, 8.1

# Learning Outcomes

- An understanding of the structure and limits of multiprocessor hardware.
- An appreciation of approaches to operating system support for multiprocessor machines.
- An understanding of issues surrounding and approaches to construction of multiprocessor synchronisation primitives.

# CPU clock-rate increase slowing



# Multiprocessor System

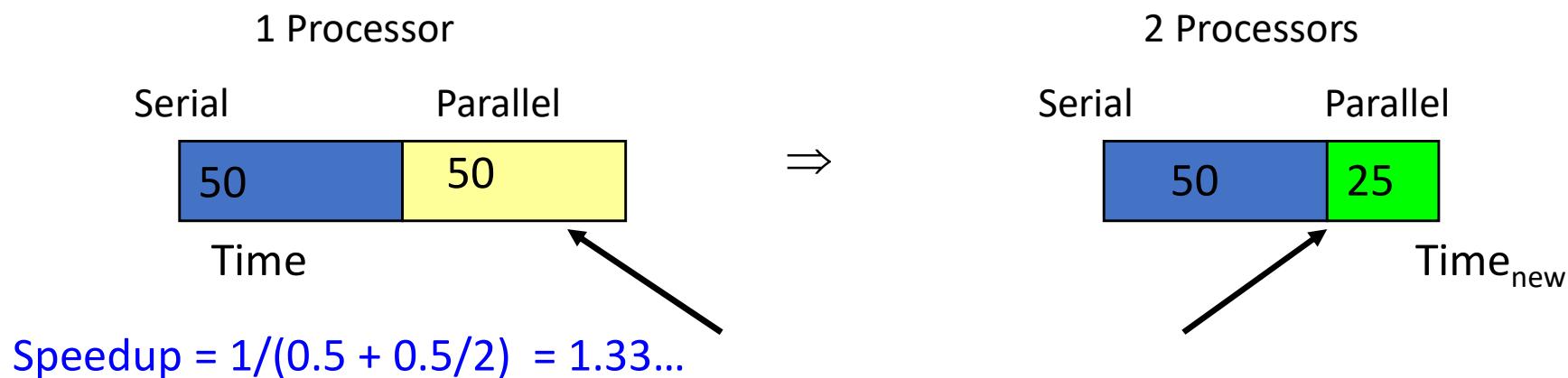
- We will look at *shared-memory multiprocessors*
  - More than one processor sharing the same memory
- A single CPU can only go so fast
  - Use more than one CPU to improve performance
  - Assumes
    - Workload can be parallelised
    - Workload is not I/O-bound or memory-bound
- Disks and other hardware can be expensive
  - Can share hardware between CPUs

# Amdahl's law



- Given a proportion  $P$  of a program that can be made parallel, and the remaining serial portion  $(1-P)$ , speedup by using  $N$  processors

$$\frac{1}{(1-P) + \frac{P}{N}}$$

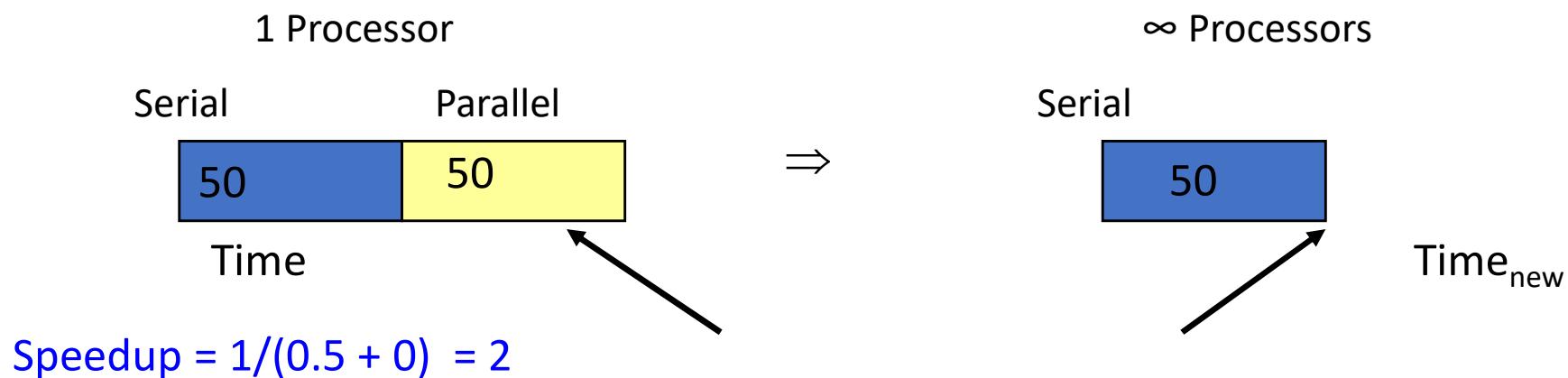


# Amdahl's law



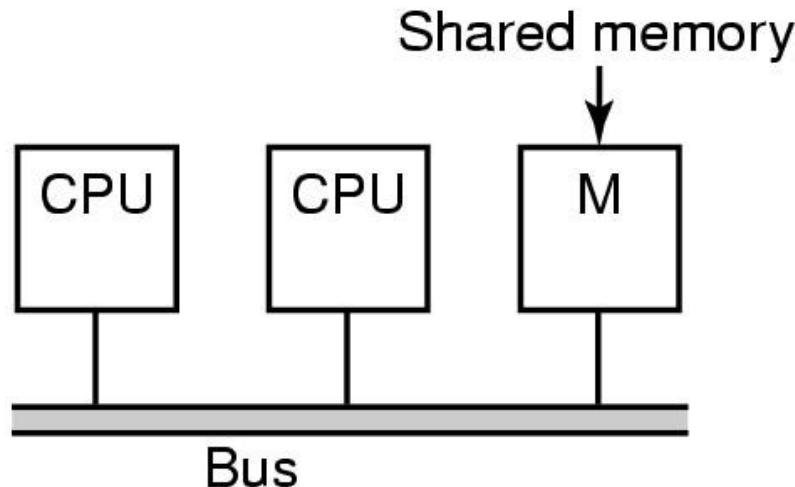
- Given a proportion  $P$  of a program that can be made parallel, and the remaining serial portion ( $1-P$ ), speedup by using  $N$  processors

$$\frac{1}{(1-P) + \frac{P}{N}}$$



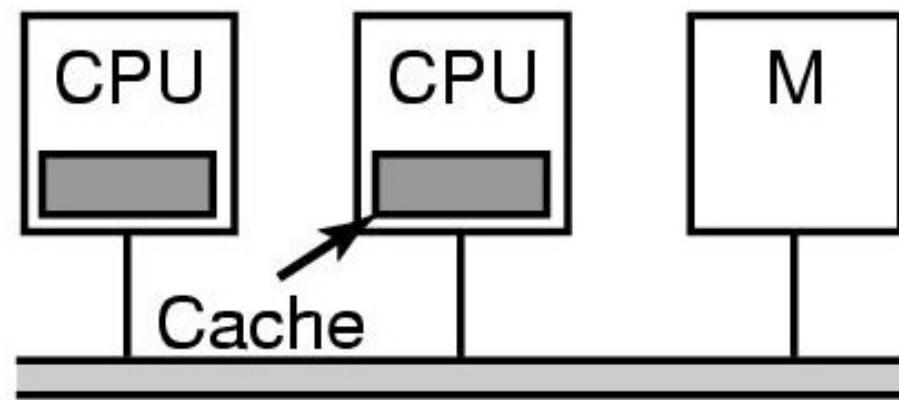
# Bus-Based Uniform Memory Access Multiprocessors

- Simplest MP is more than one processor on a single bus connect to memory
  - Access to all memory occurs at the same speed for all processors.
  - Bus bandwidth becomes a bottleneck with more than just a few CPUs



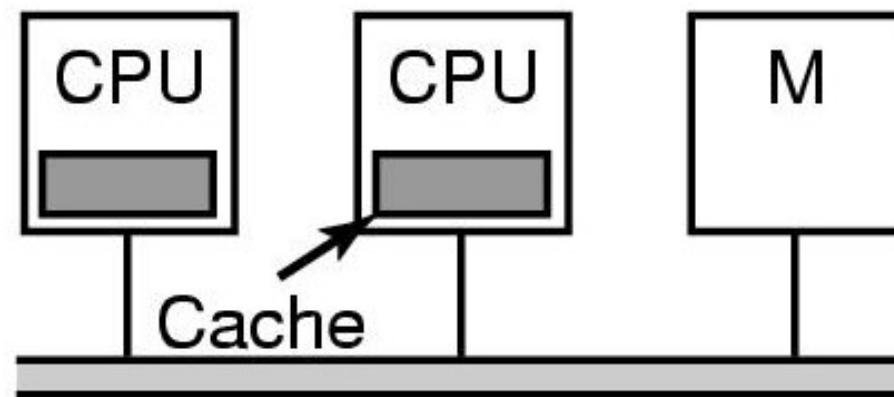
# Multiprocessor caches

- Each processor has a cache to reduce its need for access to memory
  - Hope is most accesses are to the local cache
  - Bus bandwidth still becomes a bottleneck with many CPUs



# Cache Consistency

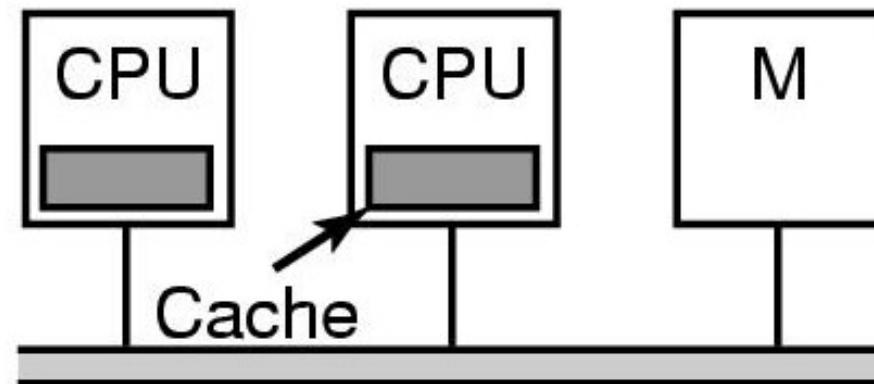
- What happens if one CPU writes to address 0x1234 (and it is stored in its cache) and another CPU reads from the same address (and gets what is in its cache)?



(b)

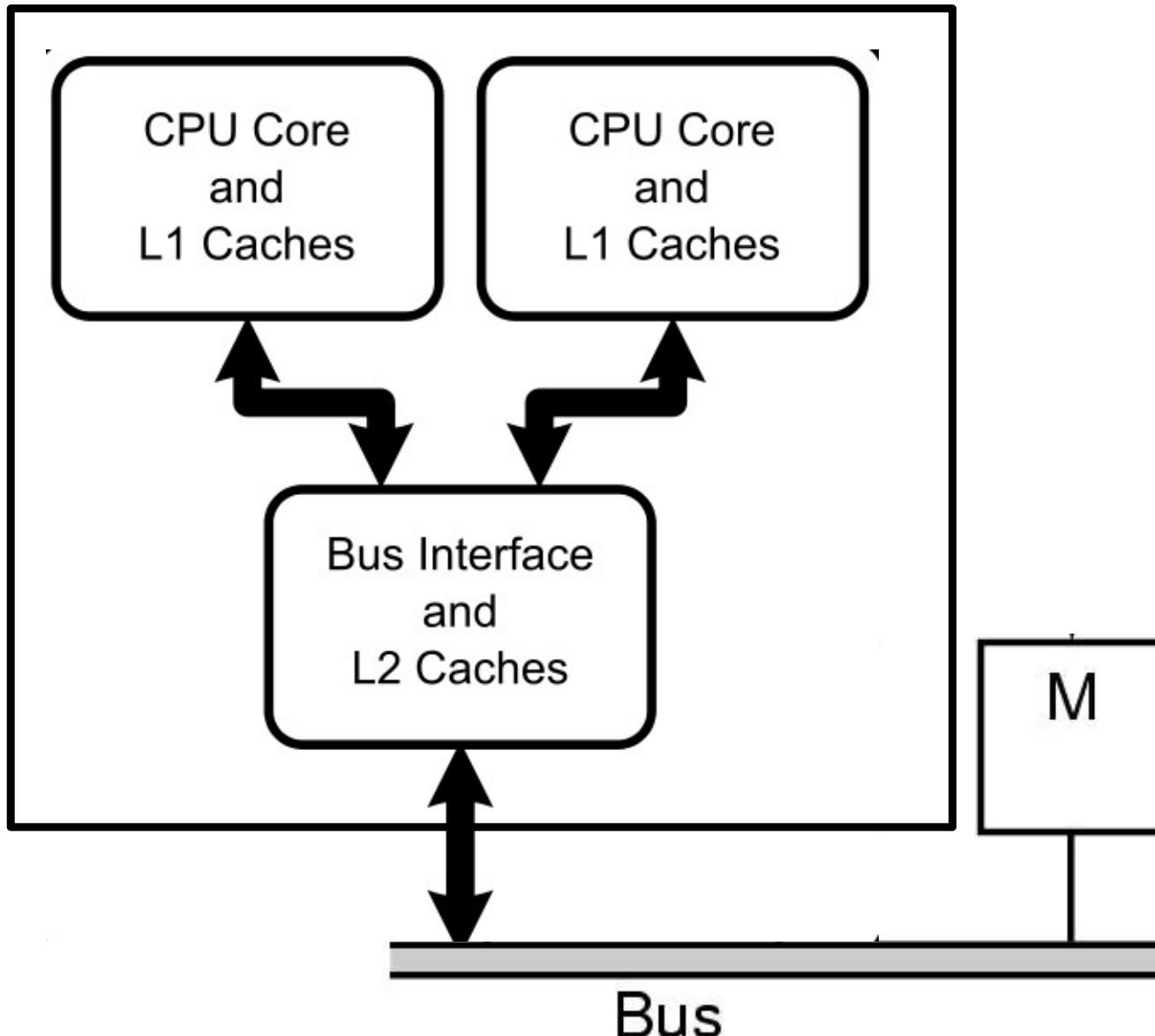
# Cache Consistency

- Cache consistency is usually handled by the hardware.
  - Writes to one cache propagate to, or invalidate appropriate entries on other caches
  - Cache transactions also consume bus bandwidth



(b)

# Multi-core Processor



# Bus-Based UMA Multiprocessors

- With only a single shared bus, scalability can be limited by the bus bandwidth of the single bus
  - Caching only helps so much
- Alternative bus architectures do exist.
  - They improve bandwidth available
  - Don't eliminate constraint that bandwidth is limited

# Summary

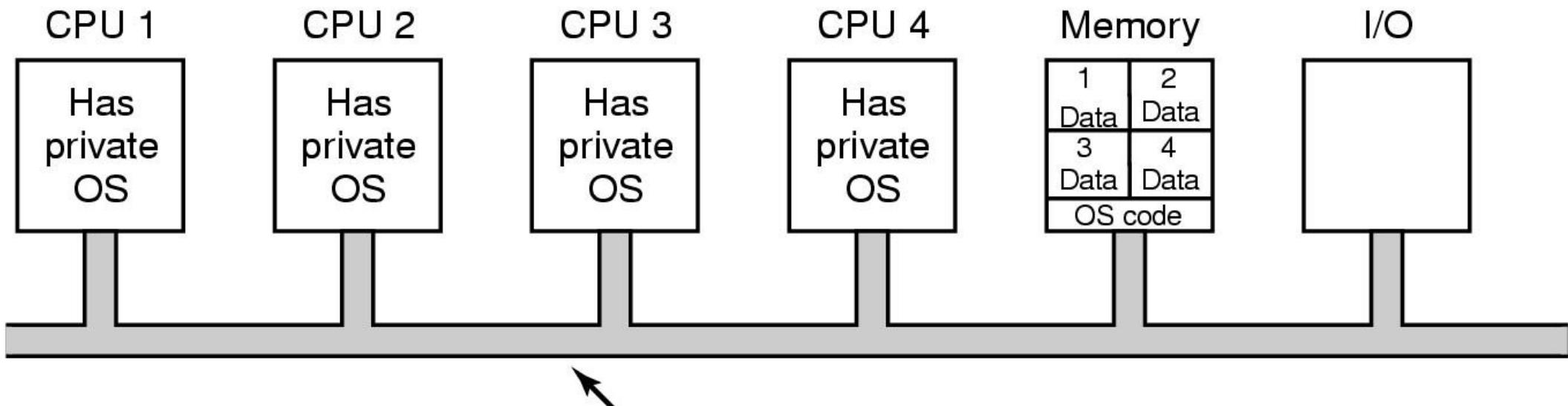
- Multiprocessors can
  - Increase computation power beyond that available from a single CPU
  - Share resources such as disk and memory
- However
  - Assumes parallelizable workload to be effective
  - Assumes not I/O bound
  - Shared buses (bus bandwidth) limits scalability
    - Can be reduced via hardware design
    - Can be reduced by carefully crafted software behaviour
      - Good cache locality together with limited data sharing where possible

# Question

- How do we construct an OS for a multiprocessor?
  - What are some of the issues?

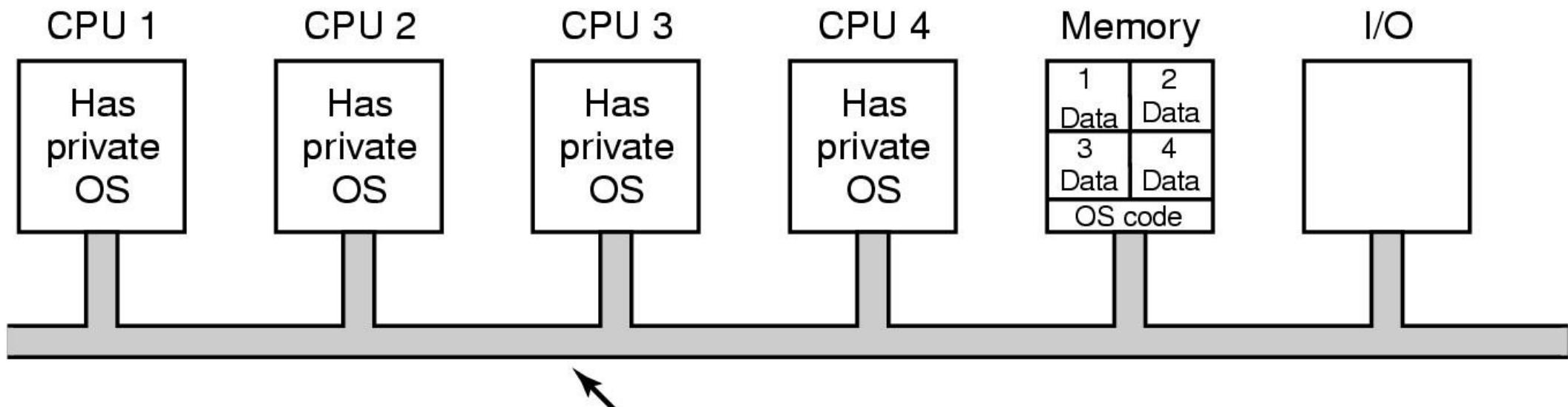
# Each CPU has its own OS?

- Statically allocate physical memory to each CPU
- Each CPU runs its own independent OS
- Share peripherals
- Each CPU (OS) handles its processes system calls



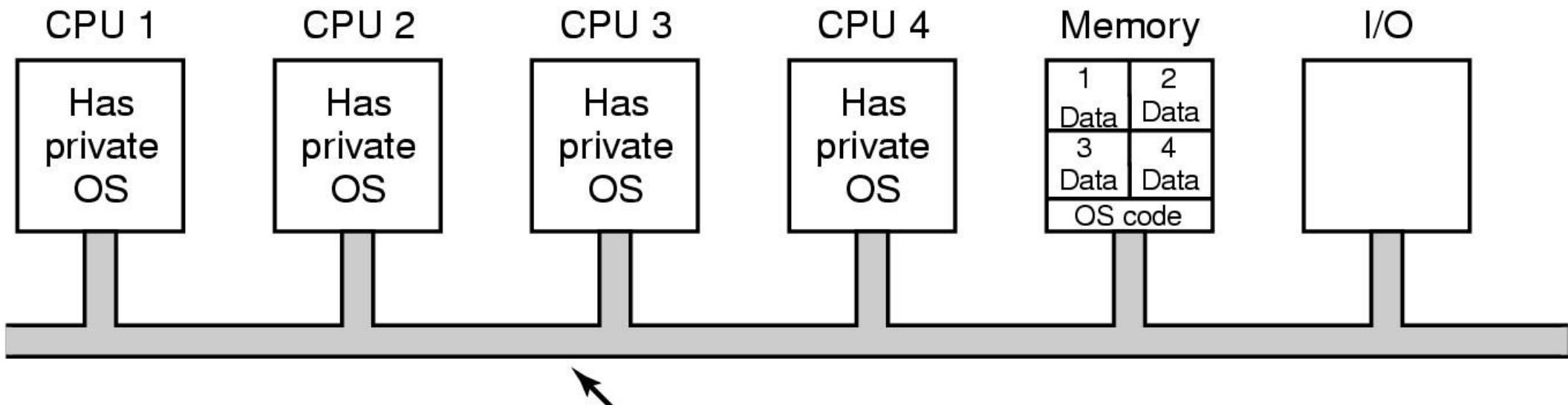
# Each CPU has its own OS

- Used in early multiprocessor systems to ‘get them going’
  - Simpler to implement
  - Avoids CPU-based concurrency issues by not sharing
  - Scales – no shared serial sections
  - Modern analogy, virtualisation in the cloud.



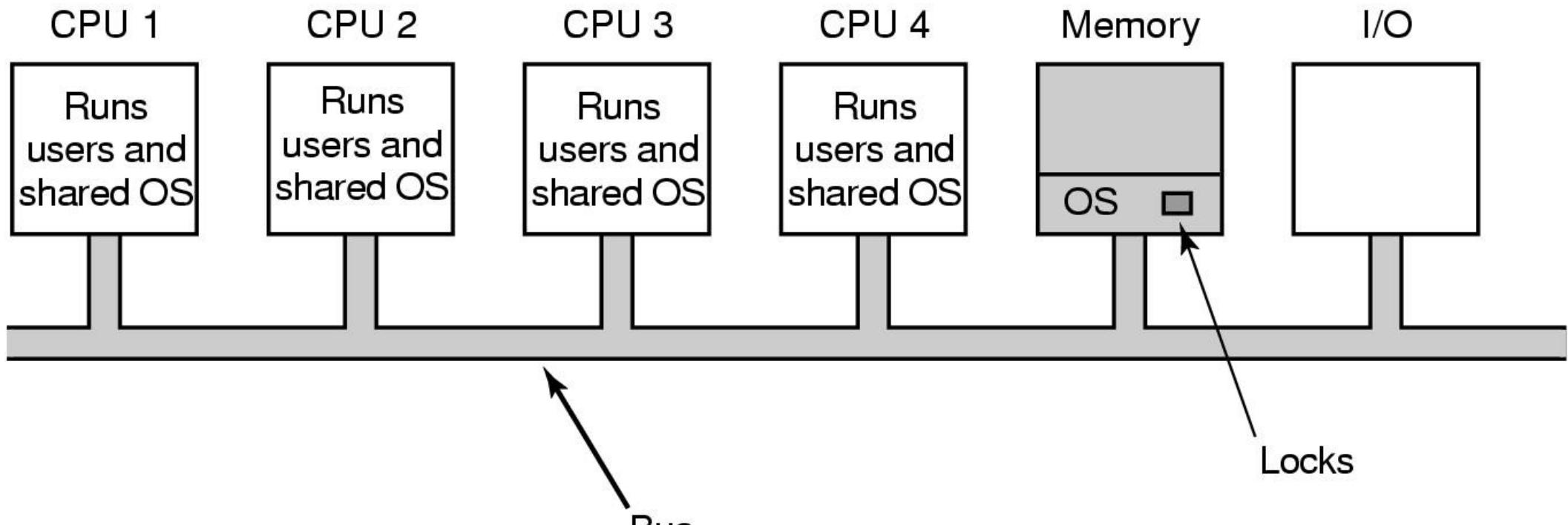
# Issues

- Each processor has its own scheduling queue
  - We can have one processor overloaded, and the rest idle
- Each processor has its own memory partition
  - We can have one processor thrashing, and the others with free memory
    - No way to move free memory from one OS to another



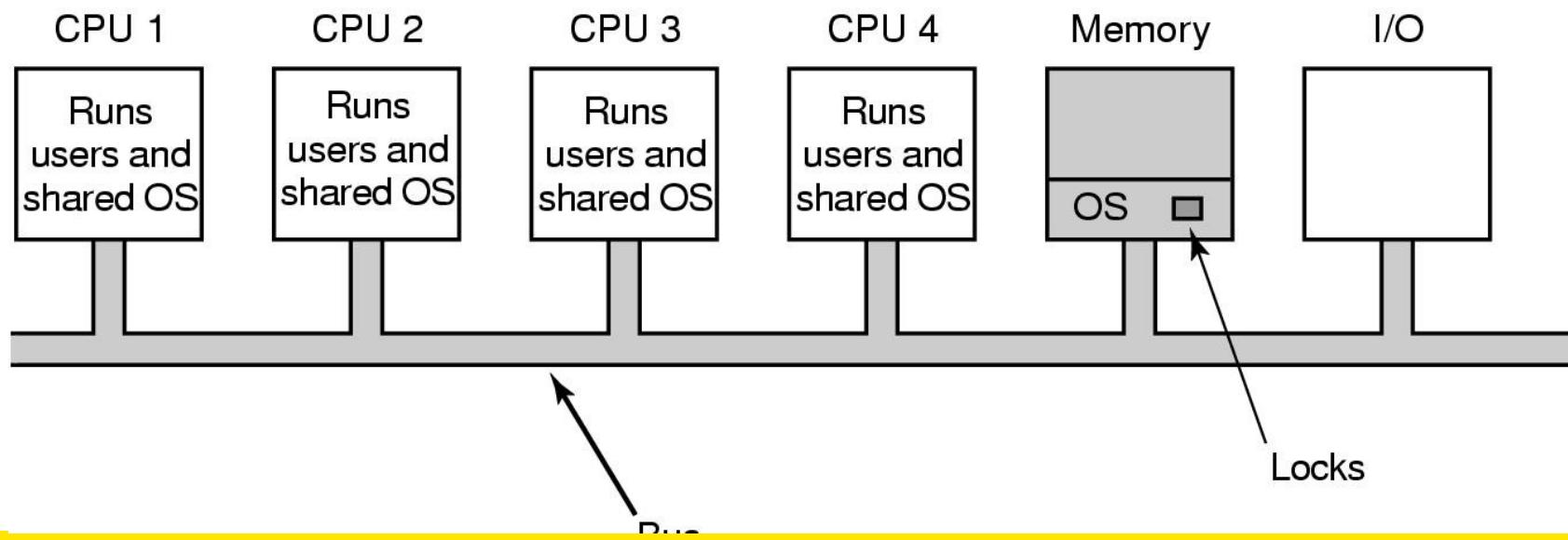
# Symmetric Multiprocessors (SMP)

- OS kernel run on all processors
  - Load and resource are balance between all processors
    - Including kernel execution
  - Issue: *Real* concurrency in the kernel
    - Need carefully applied synchronisation primitives to avoid disaster



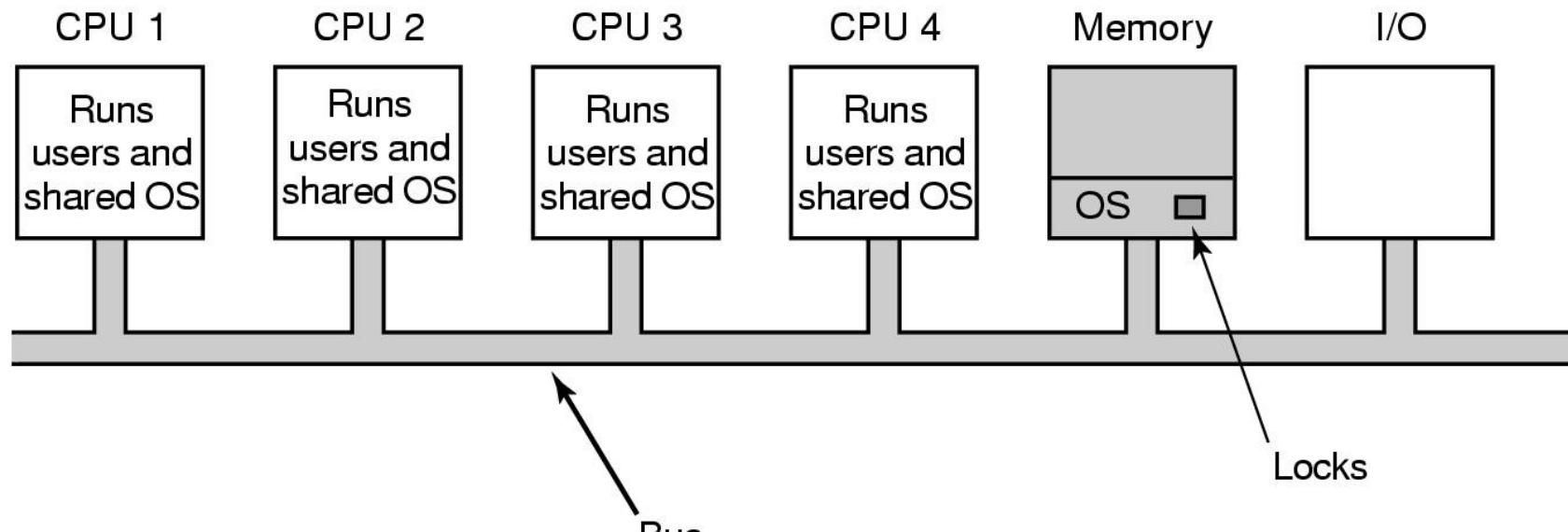
# Symmetric Multiprocessors (SMP)

- One alternative: A single mutex that make the entire kernel a large critical section
  - Only one CPU can be in the kernel at a time
  - The “big lock” becomes a bottleneck when in-kernel processing exceeds what can be done on a single CPU



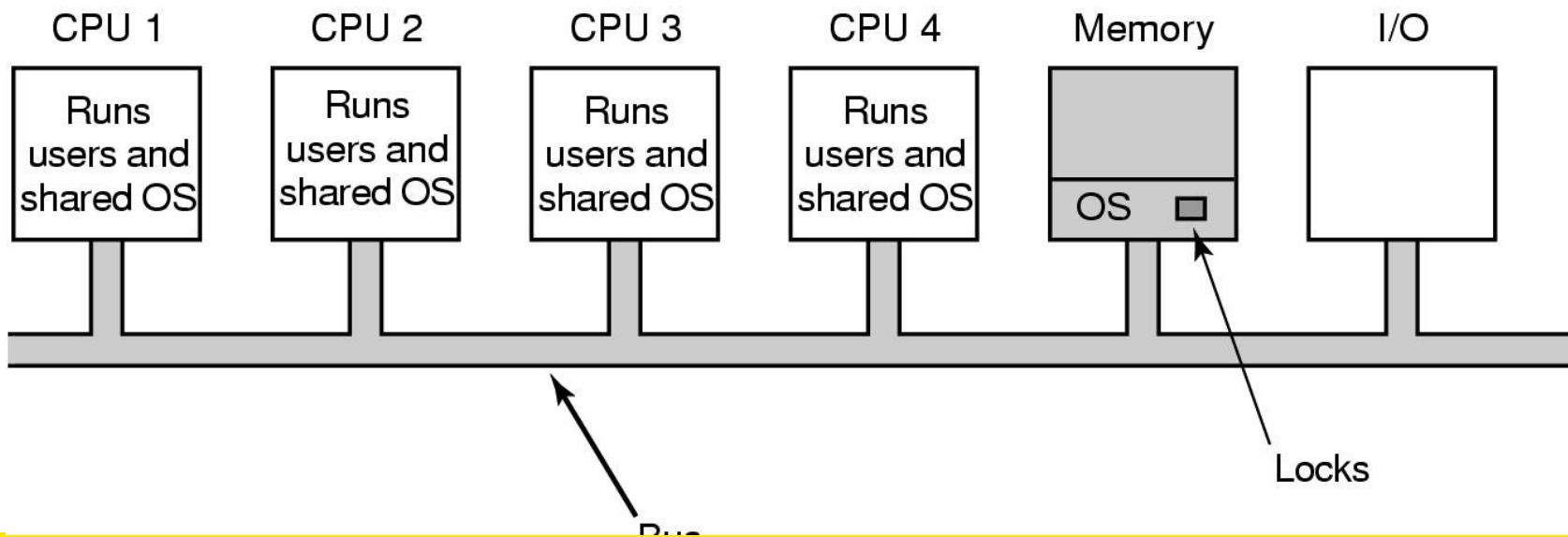
# Symmetric Multiprocessors (SMP)

- Better alternative: identify largely independent parts of the kernel and make each of them their own critical section
  - Allows more parallelism in the kernel
- Issue: Difficult task
  - Code is mostly similar to uniprocessor code
  - Hard part is identifying independent parts that don't interfere with each other
    - Remember all the inter-dependencies between OS subsystems.



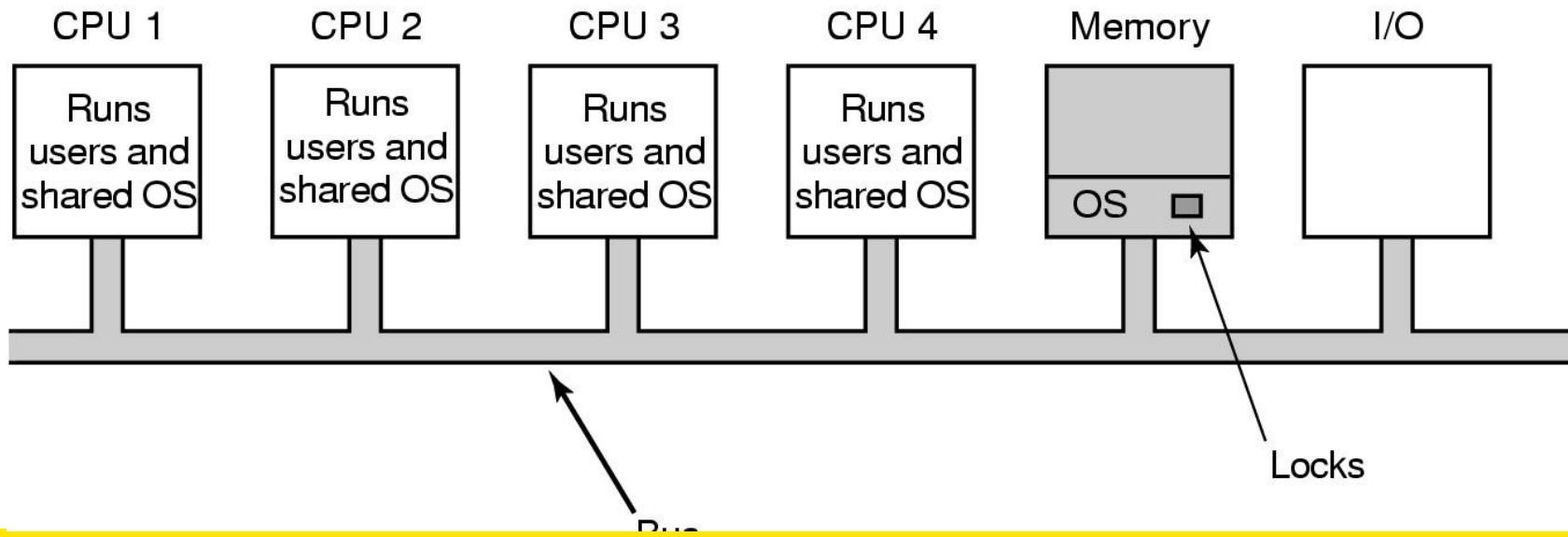
# Symmetric Multiprocessors (SMP)

- Example:
  - Associate a mutex with independent parts of the kernel
  - Some kernel activities require more than one part of the kernel
    - Need to acquire more than one mutex
    - Great opportunity to deadlock!!!!
  - Results in potentially complex lock ordering schemes that must be adhered to



# Symmetric Multiprocessors (SMP)

- Example:
  - Given a “big lock” kernel, we divide the kernel into two independent parts with a lock each
    - Good chance that one of those locks will become the next bottleneck
    - Leads to more subdivision, more locks, more complex lock acquisition rules
      - Subdivision in practice is (in reality) making more code multithreaded (parallelised)



# Real life Scalability Example

- Early 1990's, CSE wanted to run 80 X-Terminals off one or more server machines
- Winning tender was a 4-CPU bar-fridge-sized machine with 256M of RAM
  - Eventual config 6-CPU and 512M of RAM
  - Machine ran fine in all pre-session testing

# Real life Scalability Example

- Students + assignment deadline = machine unusable

# Real life Scalability Example

- To fix the problem, the tenderer supplied more CPUs to improve performance (number increased to 8)
  - No change????
- Eventually, machine was replaced with
  - Three 2-CPU pizza-box-sized machines, each with 256M RAM
  - Cheaper overall
  - Performance was dramatically improved!!!!!
  - Why?

# Real life Scalability Example

- Paper:
  - Ramesh Balan and Kurt Gollhardt, “A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machines”, Proc. 1992 Summer USENIX conference
- The 4-8 CPU machine hit a bottleneck in the single threaded VM code
  - Adding more CPUs simply added them to the wait queue for the VM locks, and made others wait longer
- The 2 CPU machines did not generate that much lock contention and performed proportionally better.

# Lesson Learned

- Building scalable multiprocessor kernels is hard
- Lock contention can limit overall system performance

# SMP Linux similar evolution

- Linux 2.0 Single kernel big lock (1996)
- Linux 2.2 Big lock with interrupt handling locks
- Linux 2.4 Big lock plus some subsystem locks
- Linux 2.6 most code now outside the big lock, data-based locking, lots of scalability tuning, etc, etc..
- Big lock removed in 2011 in kernel version 2.6.39

# Multiprocessor Synchronisation

- Given we need synchronisation, how can we achieve it on a multiprocessor machine?
  - Unlike a uniprocessor, disabling interrupts does not work.
    - **It does not prevent other CPUs from running in parallel**
  - Need special hardware support

# Recall Mutual Exclusion with Test-and-Set

enter\_region:

```
TSL REGISTER,LOCK          | copy lock to register and set lock to 1
CMP REGISTER,#0            | was lock zero?
JNE enter_region           | if it was non zero, lock was set, so loop
RET | return to caller; critical region entered
```

leave\_region:

```
MOVE LOCK,#0               | store a 0 in lock
RET | return to caller
```

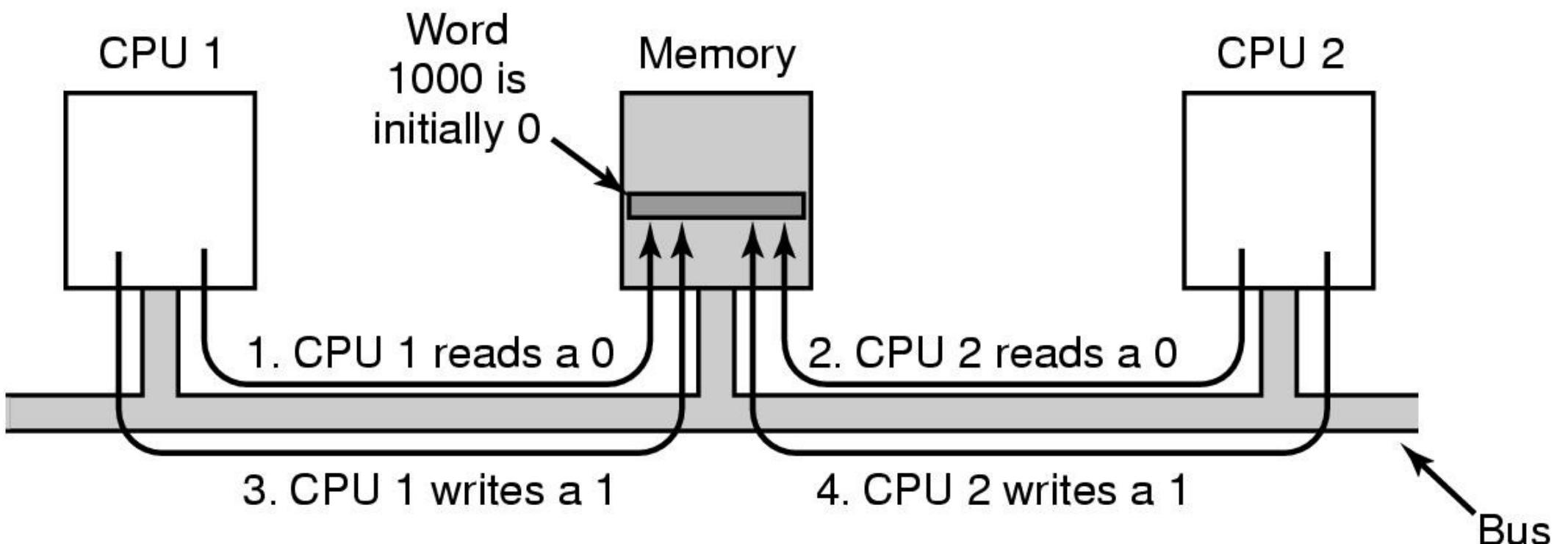
Entering and leaving a critical region using the  
TSL instruction

# Test-and-Set

- Hardware guarantees that the instruction executes atomically on a CPU.
  - Atomically: As an indivisible unit.
  - The instruction can not stop half way through

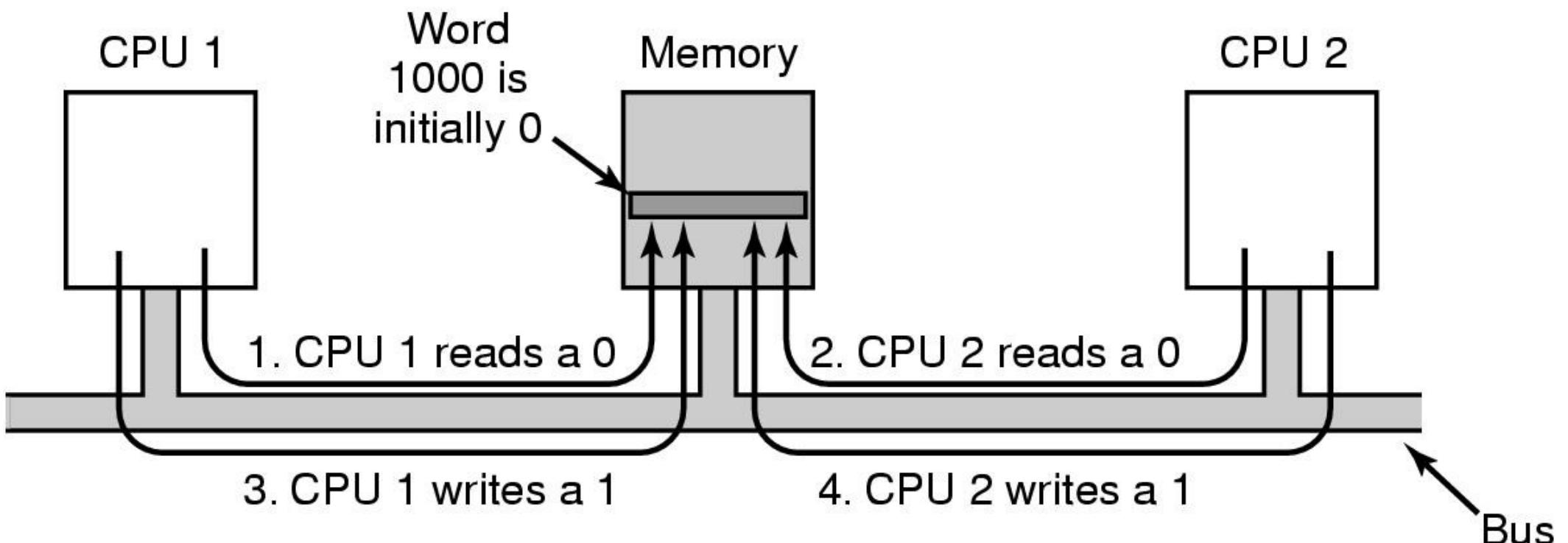
# Test-and-Set on SMP

- It does not work without some extra hardware support



# Test-and-Set on SMP

- A solution:
  - Hardware blocks all other CPUs from accessing the bus during the TSL instruction to prevent memory accesses by any other CPU.
    - TSL has mutually exclusive access to memory for duration of instruction.

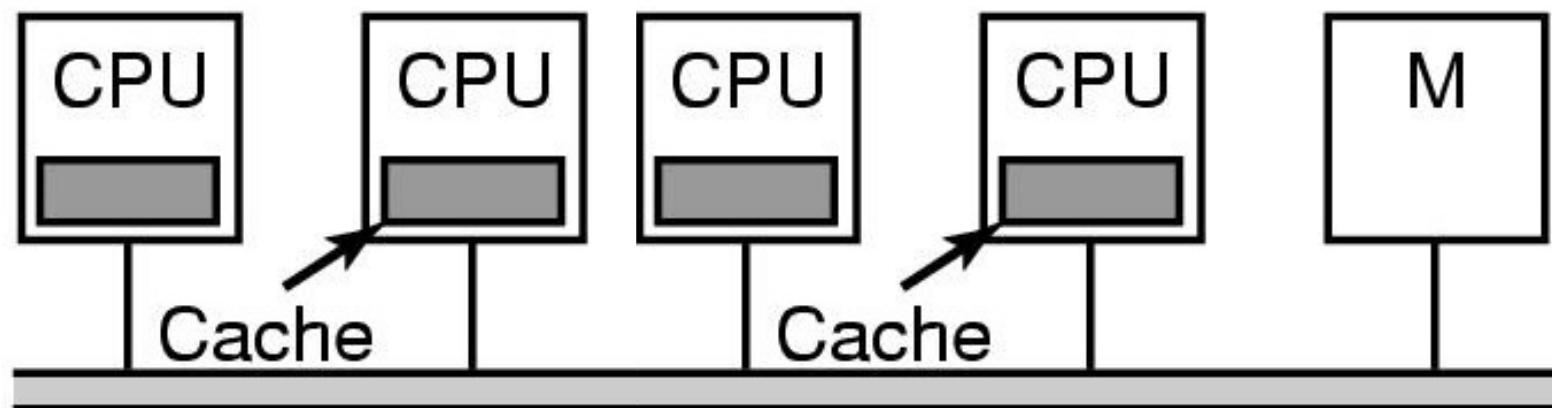


# Test-and-Set on SMP

- Test-and Set is a busy-wait synchronisation primitive
  - Called a *spinlock*
- Issue:
  - Lock contention leads to spinning on the lock
    - Spinning on a lock requires blocking the bus which slows all other CPUs down
      - Independent of whether other CPUs need a lock or not
      - Causes bus contention

# Test-and-Set on SMP

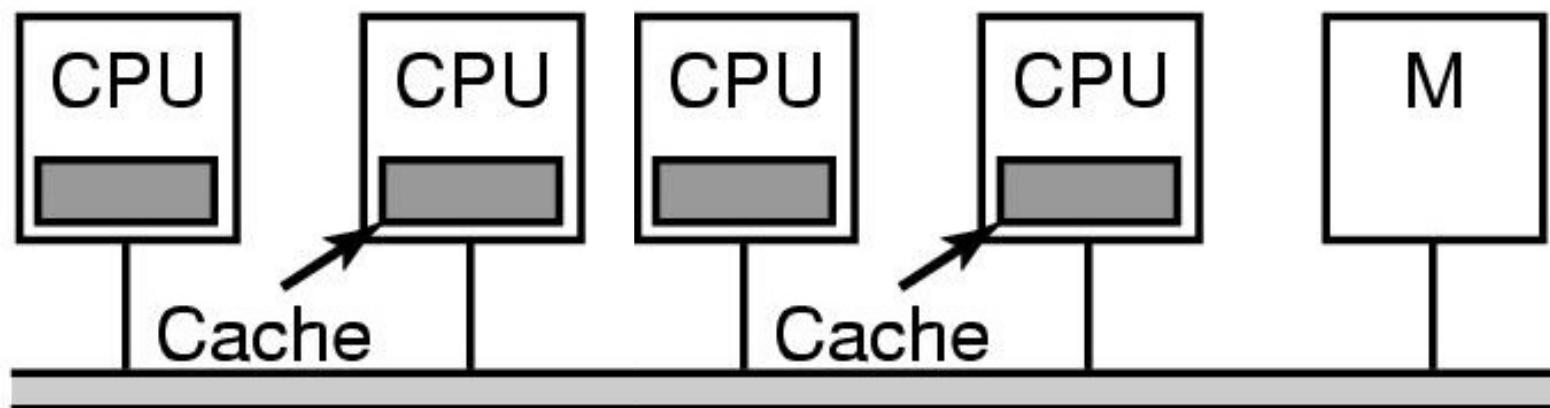
- Caching does not help reduce bus contention
  - Either TSL still blocks the bus
  - Or TSL requires exclusive access to an entry in the local cache
    - Requires invalidation of same entry in other caches, and loading entry into local cache
    - Many CPUs performing TSL simply bounce a single exclusive entry between all caches using the bus



# Reducing Bus Contention

- Read before TSL
  - Spin reading the lock variable waiting for it to change
  - When it does, use TSL to acquire the lock
- Allows lock to be shared read-only in all caches until its released
  - no bus traffic until actual release
- No race conditions, as acquisition is still with TSL.

```
start:  
while (lock == 1);  
r = TSL(lock)  
if (r == 1)  
    goto start;
```



Thomas Anderson, “The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors”, *IEEE Transactions on Parallel and Distributed Systems*, Vol 1, No. 1, 1990

# Compares Simple Spinlocks

- Test and Set

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

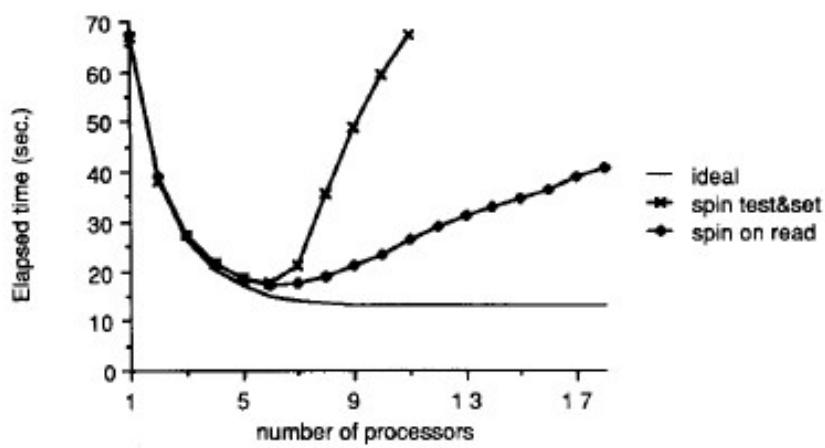
- Read before Test and Set

```
void lock (volatile lock_t *l) {  
    while (*l == BUSY || test_and_set(l)) ;  
}
```

# Benchmark

```
for i = 1 .. 1,000,000 {  
    lock(l)  
    crit_section()  
    unlock()  
    compute()  
}
```

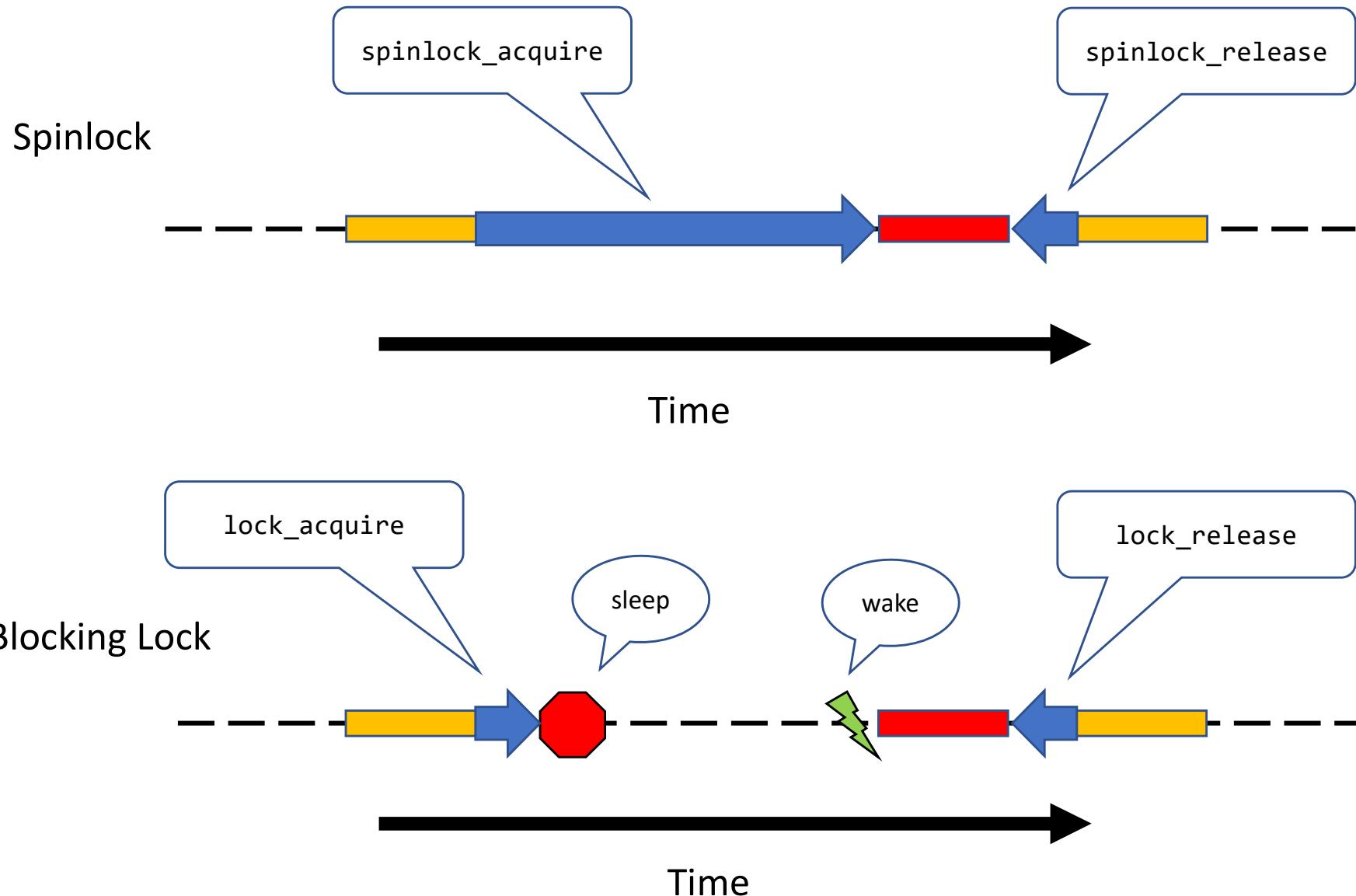
- Compute chosen from uniform random distribution of mean 5 times critical section
- Measure elapsed time on Sequent Symmetry (20 CPU 30386, coherent write-back invalidate caches)



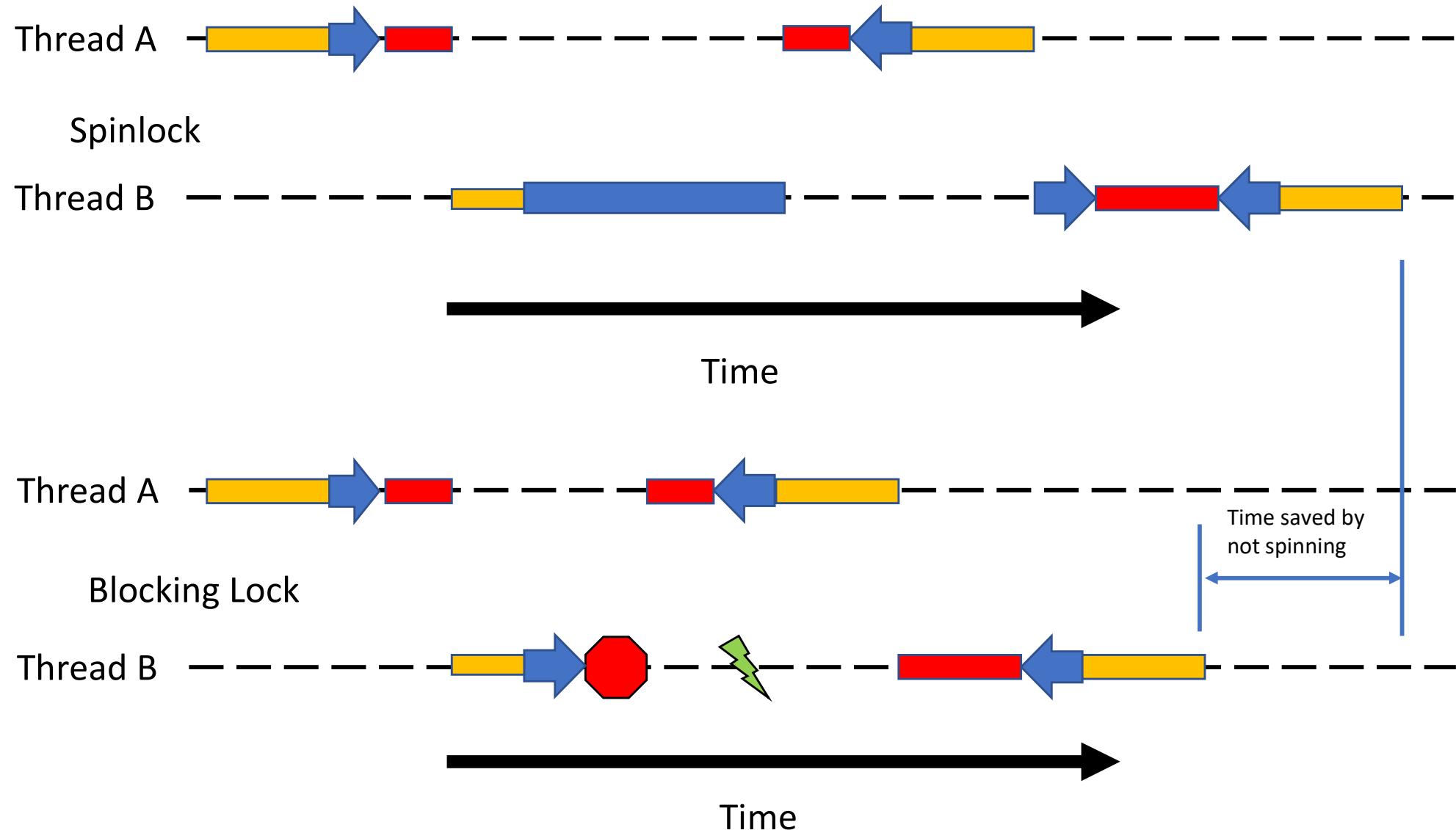
# Results

- Test and set performs poorly once there is enough CPUs to cause contention for lock
  - Expected
- Read before Test and Set performs better
  - Performance less than expected
  - Still significant contention on lock when CPUs notice release and all attempt acquisition
- Critical section performance degenerates
  - Critical section requires bus traffic to modify shared structure
  - Lock holder competes with CPU that's waiting as they test and set, so the lock holder is slower
  - Slower lock holder results in more contention

# Spinning Locks versus Blocking Locks



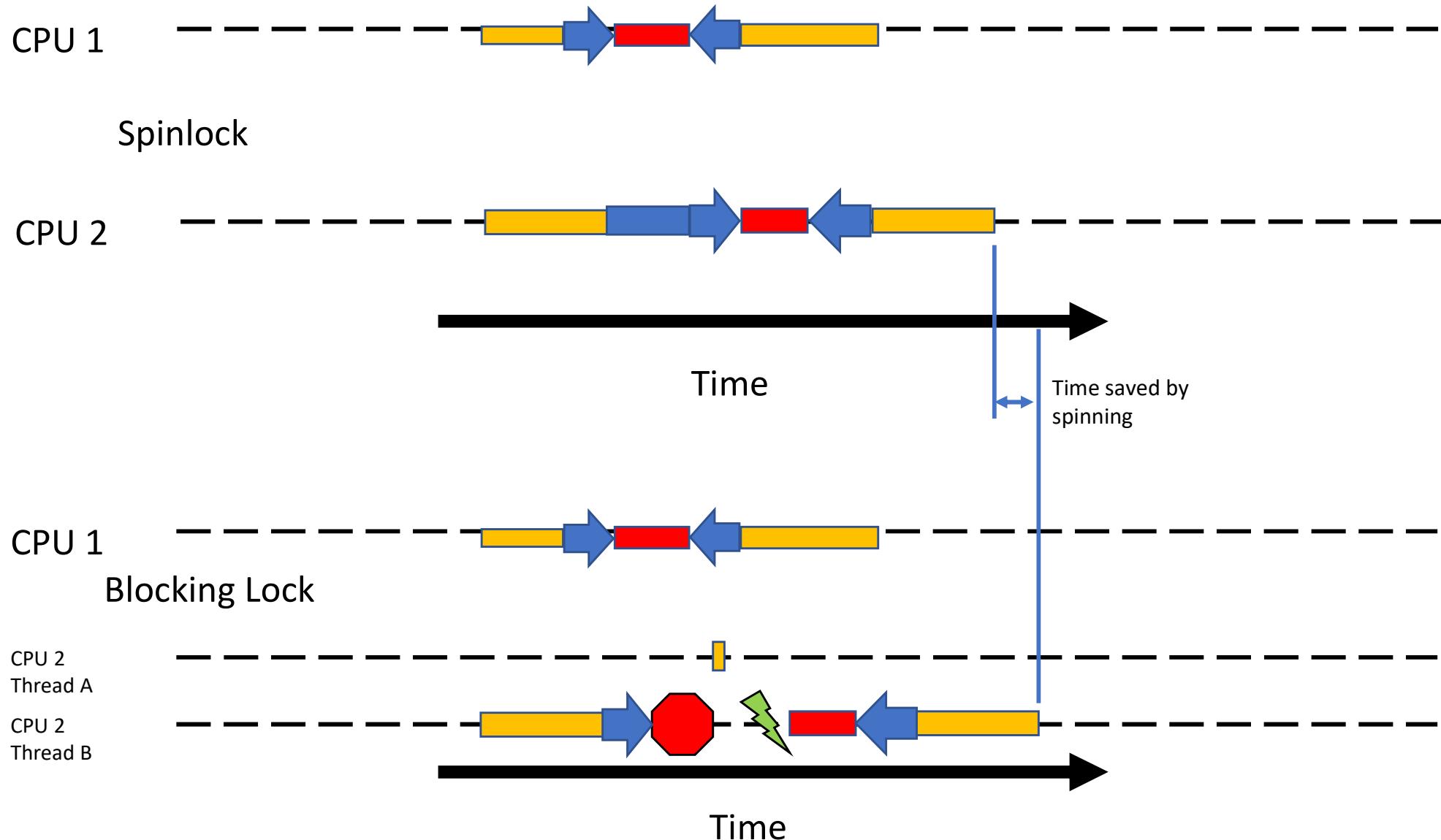
# Uniprocessor: Spinning versus Blocking



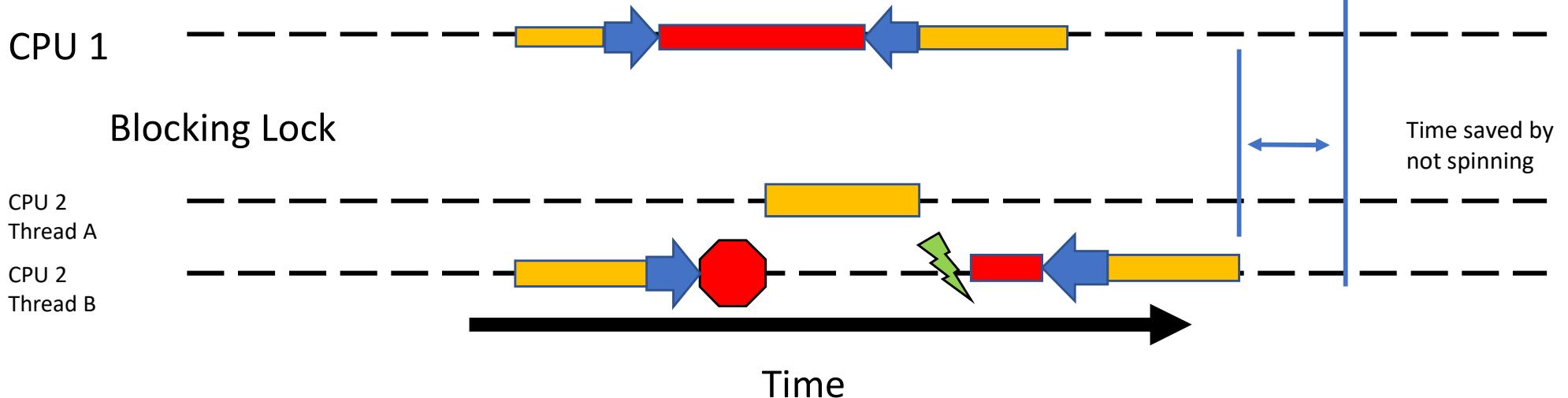
# Spinning versus Blocking and Switching

- Spinning (busy-waiting) on a lock makes no sense on a uniprocessor
  - There was no other running process to release the lock
  - Blocking and (eventually) switching to the lock holder is the only sensible option.
- On SMP systems, the decision to spin or block is not as clear.
  - The lock is held by another running CPU and will be freed without necessarily switching away from the requestor

# Multiprocessor: Spinning versus Blocking



# Multiprocessor: Spinning versus Blocking



# Spinning versus Switching

- Blocking and switching
  - to another process takes time
    - Save context and restore another
    - Cache contains current process not new process
      - Adjusting the cache working set also takes time
    - TLB is similar to cache
  - Switching back when the lock is free encounters the same again
  - Spinning wastes CPU time directly
- Trade off
  - If lock is held for less time than the overhead of switching to and back
    - ⇒ It's more efficient to spin
    - ⇒ Spinlocks expect critical sections to be short
    - ⇒ No waiting for I/O within a spinlock
    - ⇒ No nesting locks within a spinlock

# Preemption and Spinlocks

- Critical sections synchronised via spinlocks are expected to be short
    - Avoid other CPUs wasting cycles spinning
  - What happens if the spinlock holder is preempted at end of holder's timeslice
    - Mutual exclusion is still guaranteed
    - Other CPUs will spin until the holder is scheduled again!!!!
- ⇒ Spinlock implementations disable interrupts in addition to acquiring locks
- avoids lock-holder preemption
  - avoids spinning on a uniprocessor

# Scheduling



THE UNIVERSITY OF  
NEW SOUTH WALES

# Learning Outcomes

- Understand the role of the scheduler, and how its behaviour influences the performance of the system.
- Know the difference between I/O-bound and CPU-bound tasks, and how they relate to scheduling.



# What is Scheduling?

- On a multi-programmed system
  - We may have more than one *Ready* process
- On a batch system
  - We may have many jobs waiting to be run
- On a multi-user system
  - We may have many users concurrently using the system
- The ***scheduler*** decides who to run next.
  - The process of choosing is called *scheduling*.



# Is scheduling important?

- It is not in certain scenarios
  - If you have no choice
    - Early systems
      - Usually batching
      - Scheduling algorithm simple
        - » Run next on tape or next on punch tape
    - Only one thing to run
      - Simple PCs
        - Only ran a word processor, etc....
      - Simple Embedded Systems
        - TV remote control, washing machine, etc....

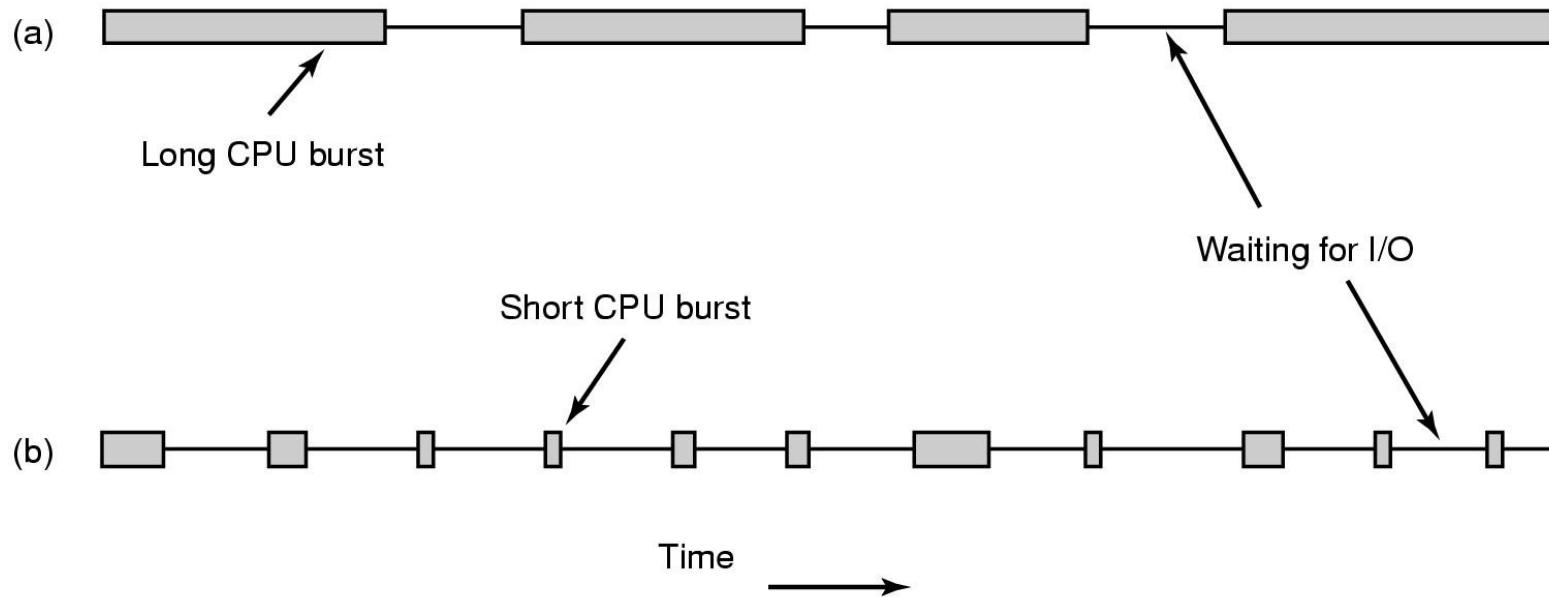


# Is scheduling important?

- It is in most realistic scenarios
  - Multitasking/Multi-user System
    - Example
      - Email daemon takes 2 seconds to process an email
      - User clicks button on application.
    - Scenario 1
      - Run daemon, then application
        - » System appears really sluggish to the user
    - Scenario 2
      - Run application, then daemon
        - » Application appears really responsive, small email delay is unnoticed
  - Scheduling decisions can have a dramatic effect on the perceived performance of the system
    - Can also affect correctness of a system with deadlines



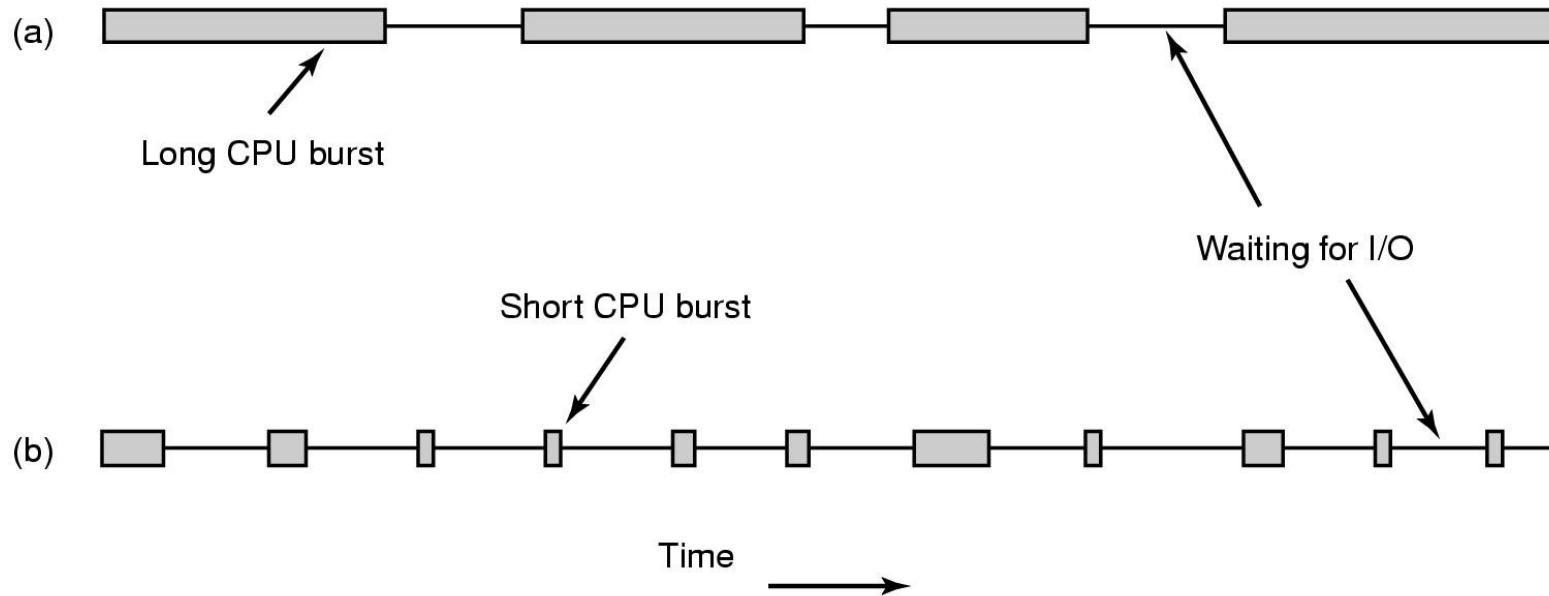
# Application Behaviour



- Bursts of CPU usage alternate with periods of I/O wait



# Application Behaviour

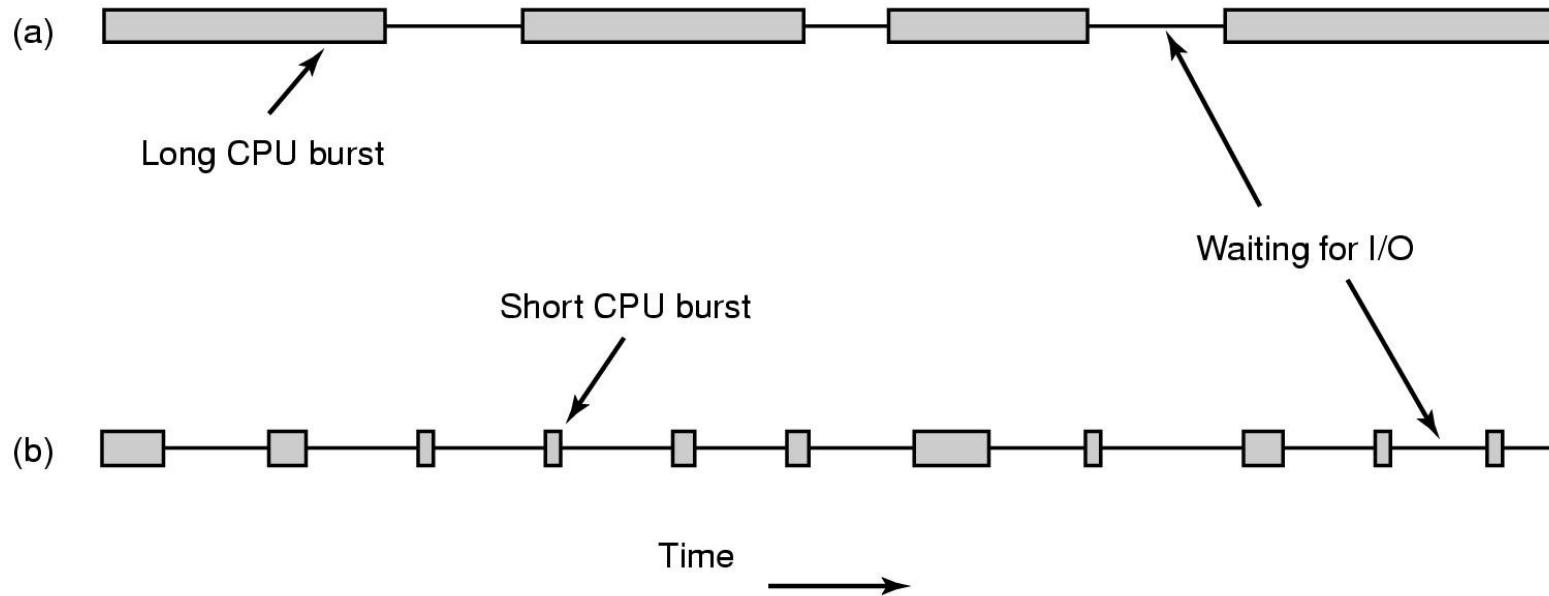


## a) CPU-Bound process

- Spends most of its computing
- Time to completion largely determined by received CPU time



# Application Behaviour

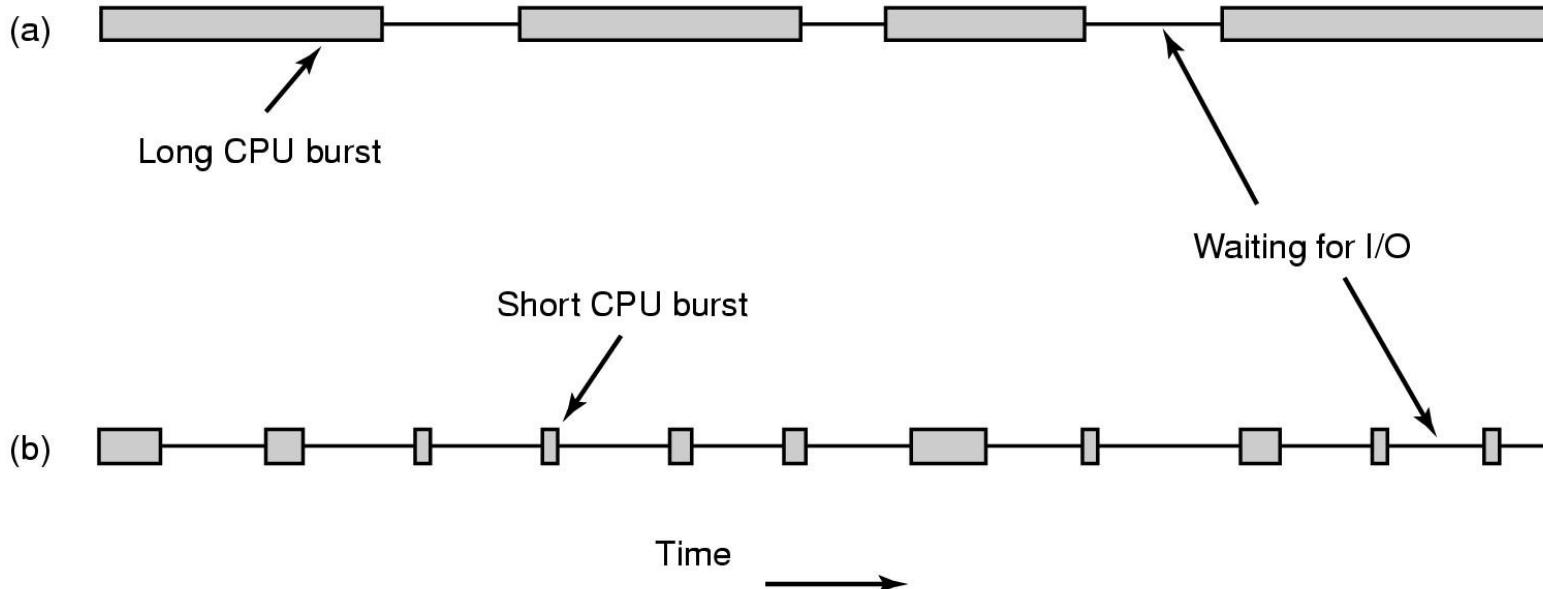


## b) I/O-Bound process

- Spend most of its time waiting for I/O to complete
  - Small bursts of CPU to process I/O and request next I/O
- Time to completion largely determined by I/O request time



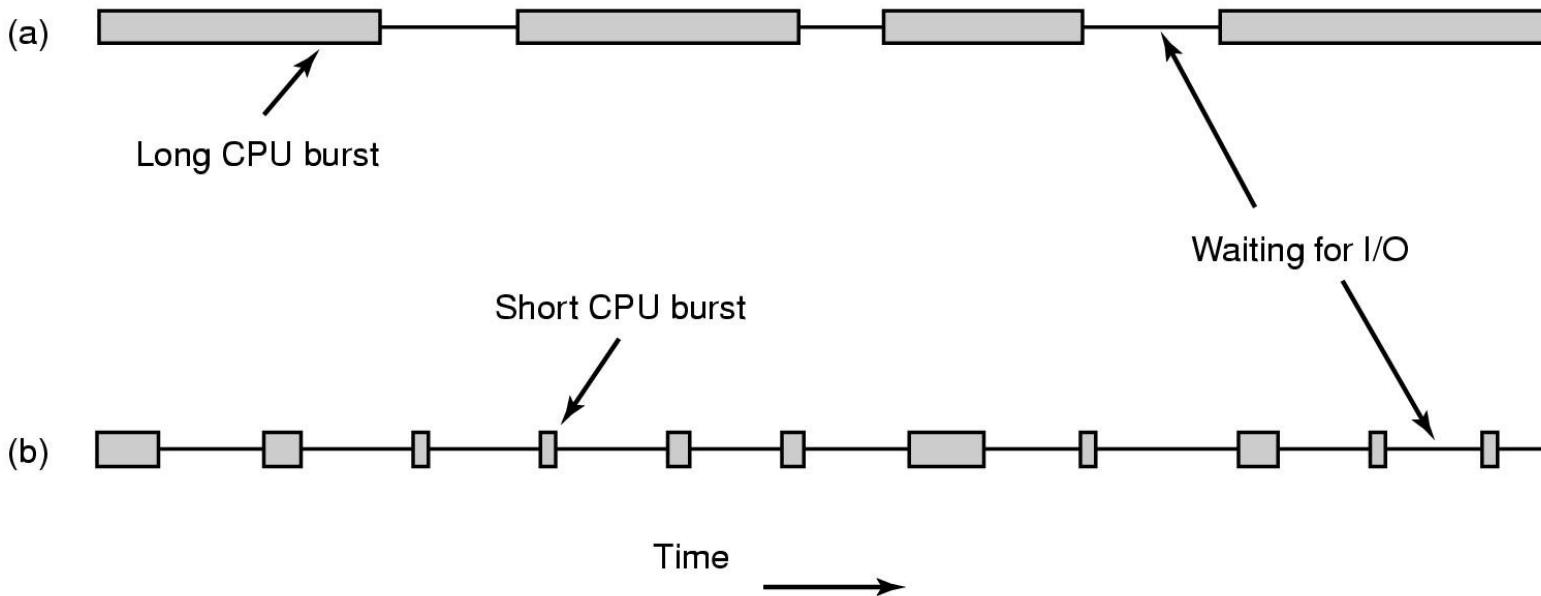
# Observation



- We need a mix of CPU-bound and I/O-bound processes to keep both CPU and I/O systems busy
- Process can go from CPU- to I/O-bound (or vice versa) in different phases of execution



# Key Insight



- Choosing to run an I/O-bound process delays a CPU-bound process by very little
  - Choosing to run a CPU-bound process prior to an I/O-bound process delays the next I/O request significantly
    - No overlap of I/O waiting with computation
    - Results in device (disk) not as busy as possible
- ⇒ Generally, favour I/O-bound processes over CPU-bound processes



# When is scheduling performed?

- A new process
  - Run the parent or the child?
- A process exits
  - Who runs next?
- A process waits for I/O
  - Who runs next?
- A process blocks on a lock
  - Who runs next? The lock holder?
- An I/O interrupt occurs
  - Who do we resume, the interrupted process or the process that was waiting?
- On a timer interrupt? (See next slide)
- Generally, a scheduling decision is required when a process (or thread) can no longer continue, or when an activity results in more than one ready process.



# Preemptive versus Non-preemptive Scheduling

- Non-preemptive
  - Once a thread is in the *running* state, it continues until it completes, blocks on I/O, or voluntarily yields the CPU
  - A single process can monopolised the entire system
- Preemptive Scheduling
  - Current thread can be interrupted by OS and moved to *ready* state.
  - Usually after a timer interrupt and process has exceeded its maximum run time
    - Can also be as a result of higher priority process that has become *ready* (after I/O interrupt).
  - Ensures fairer service as single thread can't monopolise the system
    - Requires a timer interrupt



# Categories of Scheduling Algorithms

- The choice of scheduling algorithm depends on the goals of the application (or the operating system)
  - No one algorithm suits all environments
- We can roughly categorise scheduling algorithms as follows
  - Batch Systems
    - No users directly waiting, can optimise for overall machine performance
  - Interactive Systems
    - Users directly waiting for their results, can optimise for users perceived performance
  - Realtime Systems
    - Jobs have deadlines, must schedule such that all jobs (predictably) meet their deadlines.



# Goals of Scheduling Algorithms

- All Algorithms
  - Fairness
    - Give each process a *fair* share of the CPU
  - Policy Enforcement
    - Whatever policy chosen, the scheduler should ensure it is carried out
  - Balance/Efficiency
    - Try to keep all parts of the system busy



# Goals of Scheduling Algorithms

- Interactive Algorithms
  - Minimise *response time*
    - Response time is the time difference between issuing a command and getting the result
      - E.g selecting a menu, and getting the result of that selection
    - Response time is important to the user's perception of the performance of the system.
  - Provide *Proportionality*
    - Proportionality is the user expectation that short jobs will have a short response time, and long jobs can have a long response time.
    - Generally, favour short jobs



# Goals of Scheduling Algorithms

- Real-time Algorithms
  - Must meet deadlines
    - Each job/task has a deadline.
    - A missed deadline can result in data loss or catastrophic failure
      - Aircraft control system missed deadline to apply brakes
  - Provide Predictability
    - For some apps, an occasional missed deadline is okay
      - E.g. DVD decoder
    - Predictable behaviour allows smooth DVD decoding with only rare skips



# Interactive Scheduling



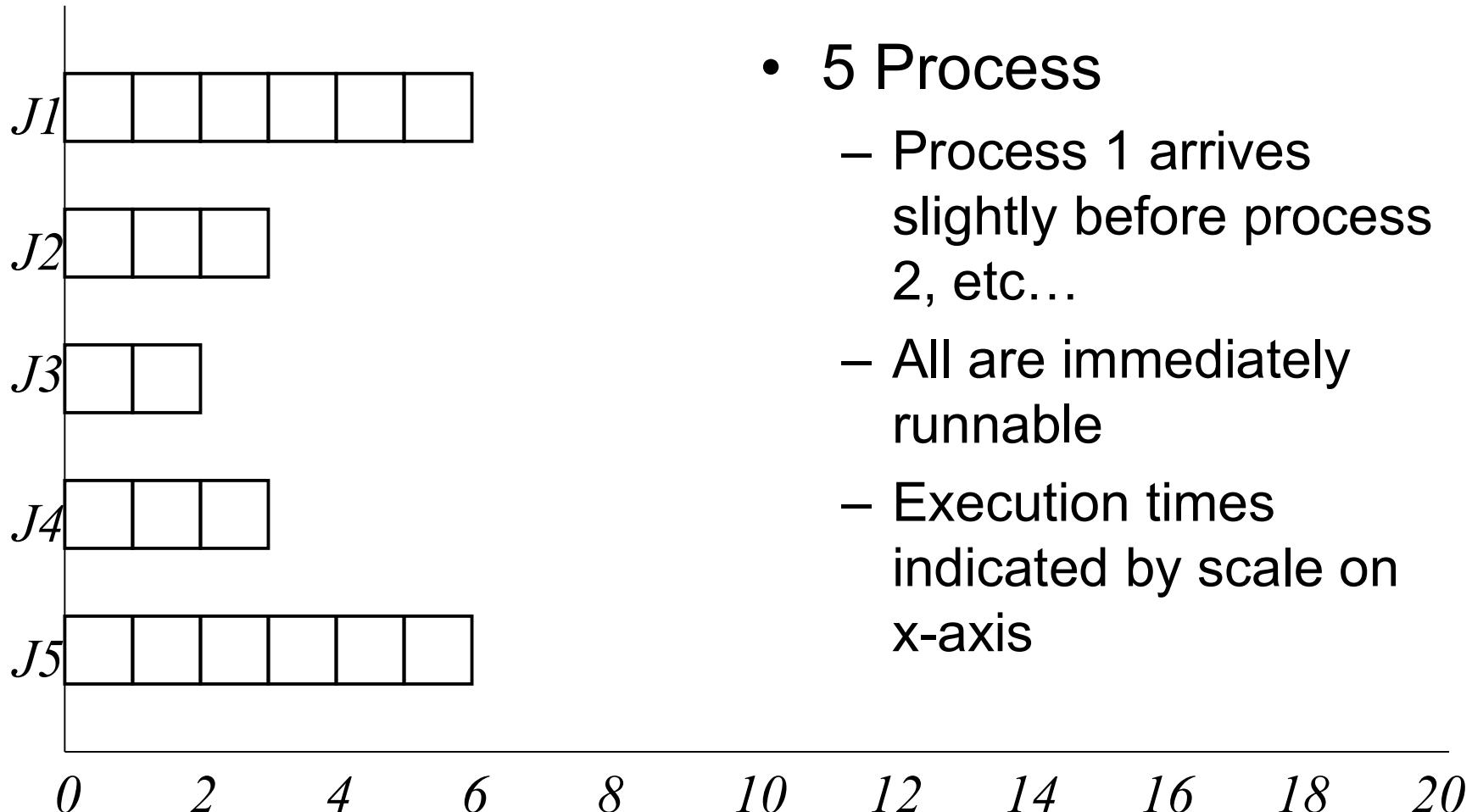
THE UNIVERSITY OF  
NEW SOUTH WALES

# Round Robin Scheduling

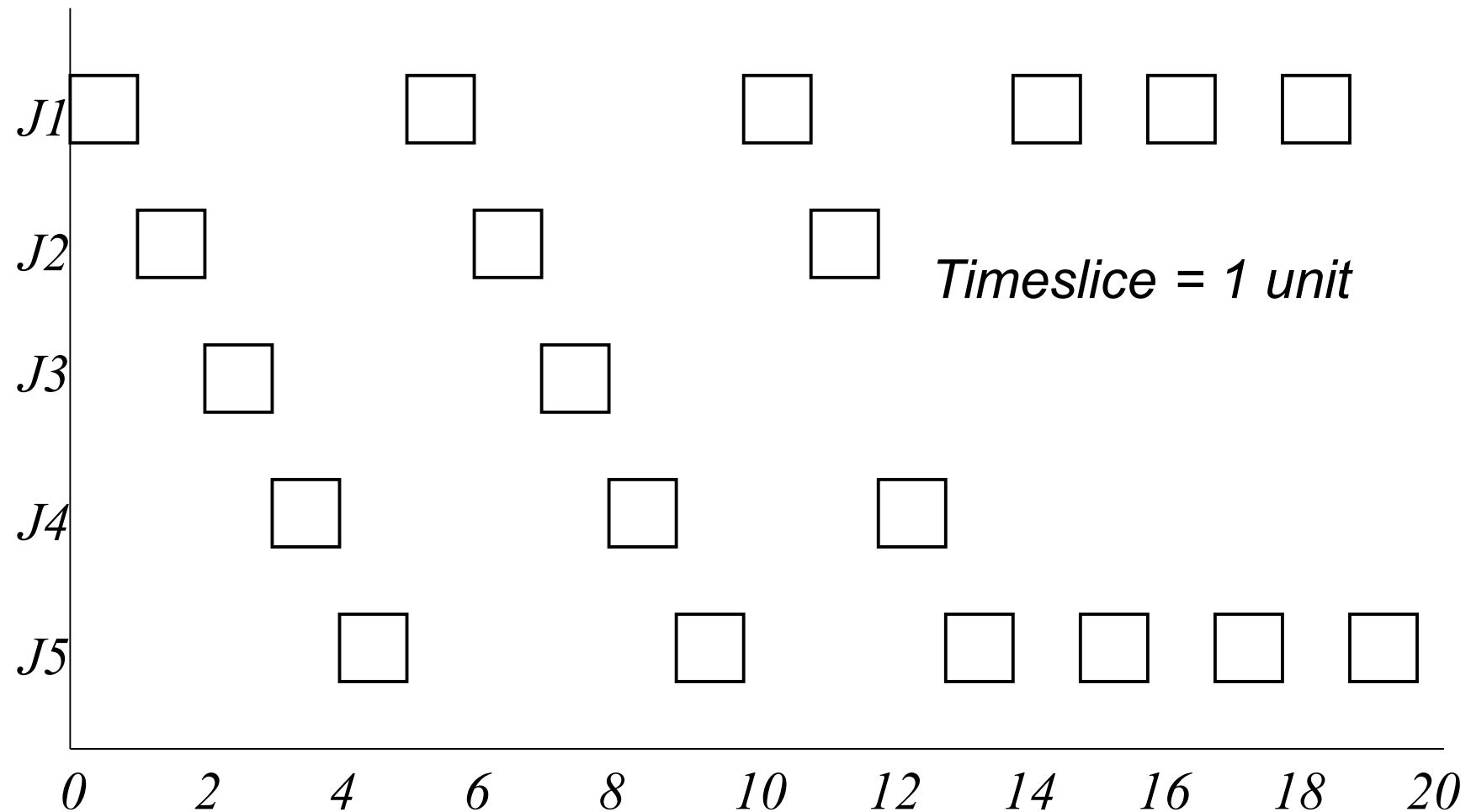
- Each process is given a *timeslice* to run in
- When the timeslice expires, the next process preempts the current process, and runs for its timeslice, and so on
  - The preempted process is placed at the end of the queue
- Implemented with
  - A ready queue
  - A regular timer interrupt



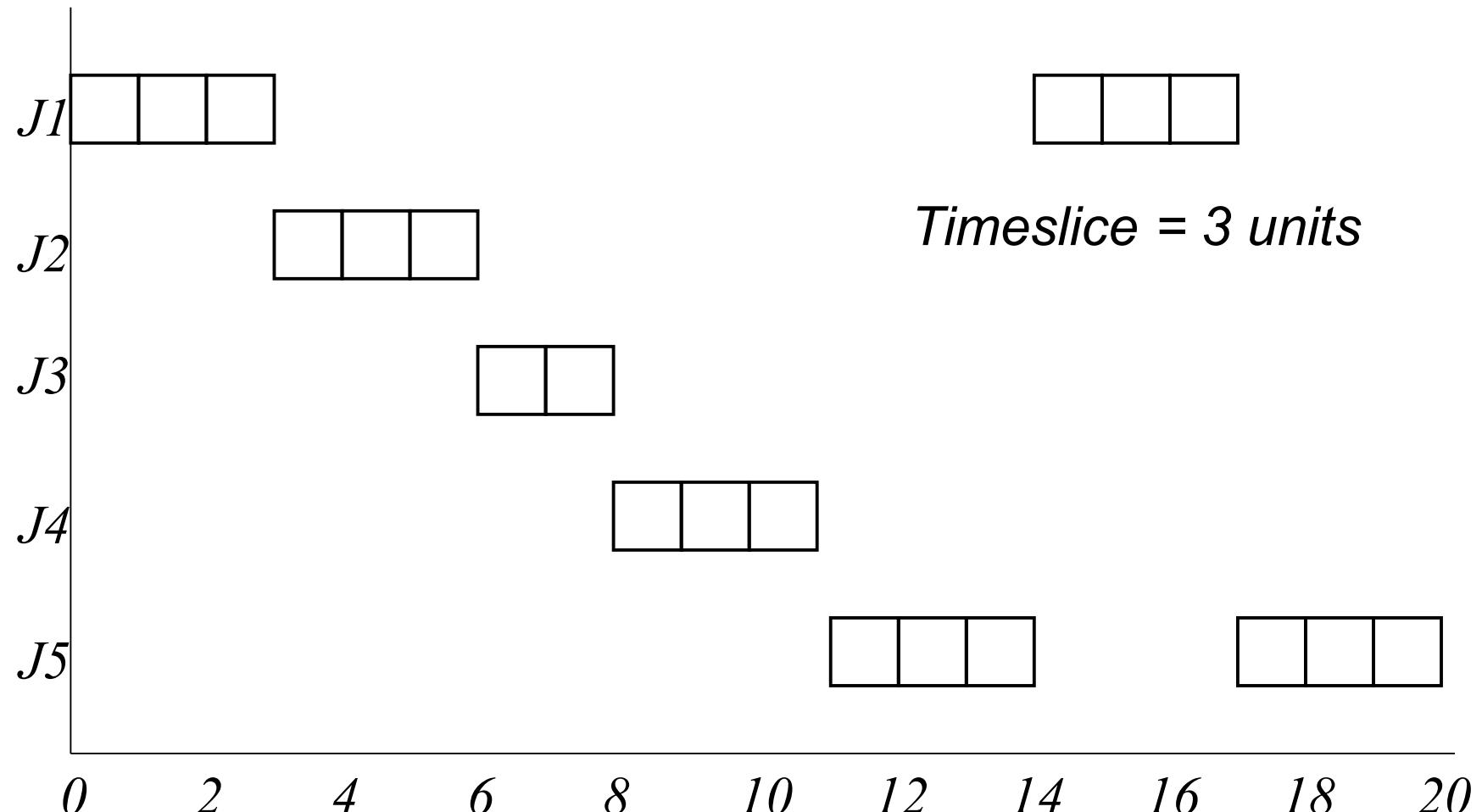
# Example



# Round Robin Schedule



# Round Robin Schedule



# Round Robin

- Pros
  - Fair, easy to implement
- Con
  - Assumes everybody is equal
- Issue: What should the timeslice be?
  - Too short
    - Waste a lot of time switching between processes
    - Example: timeslice of 4ms with 1 ms context switch = 20% round robin overhead
  - Too long
    - System is not responsive
    - Example: timeslice of 100ms
      - If 10 people hit “enter” key simultaneously, the last guy to run will only see progress after 1 second.
    - Degenerates into FCFS if timeslice longer than burst length

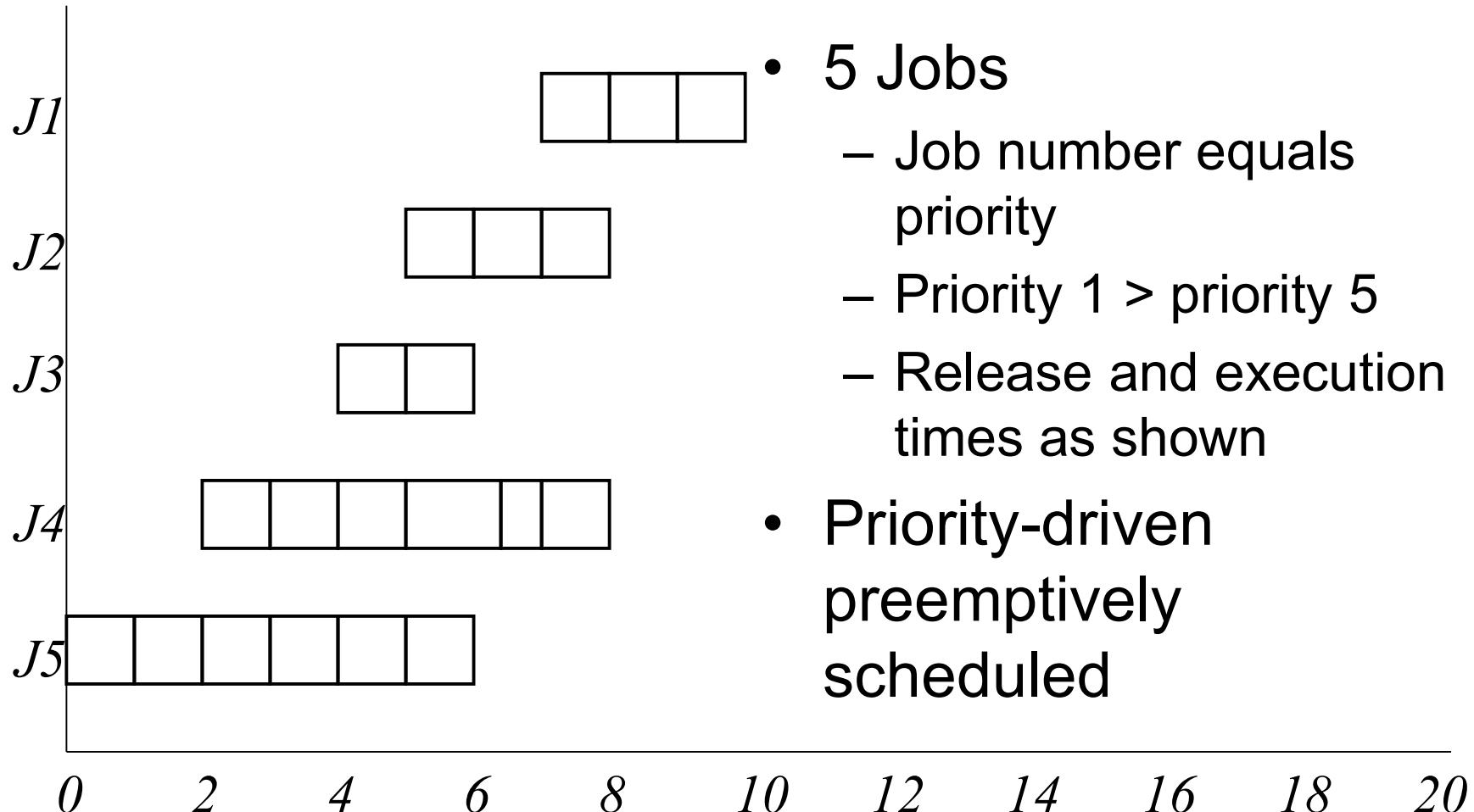


# Priorities

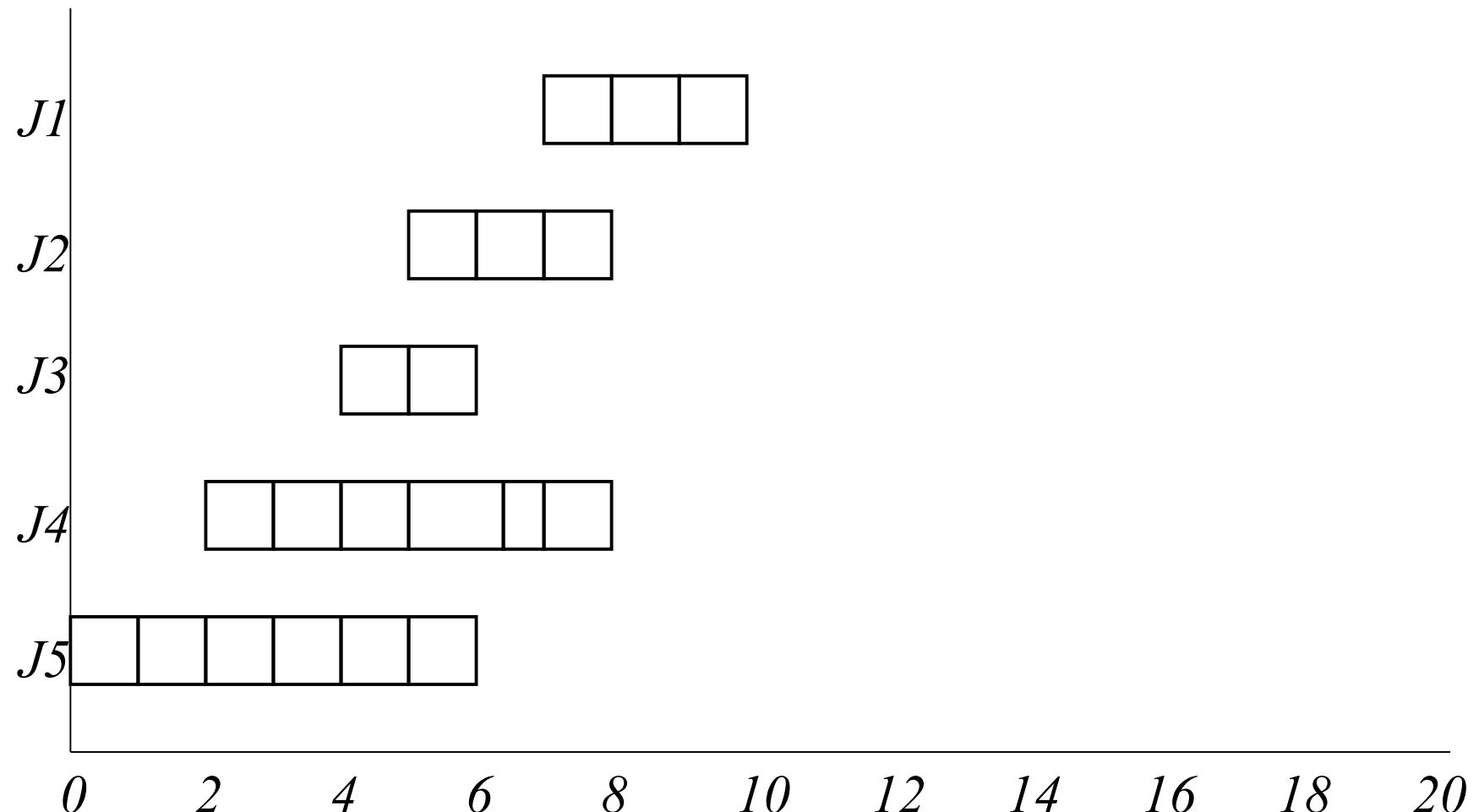
- Each Process (or thread) is associated with a priority
- Provides basic mechanism to influence a scheduler decision:
  - Scheduler will always choose a thread of higher priority over lower priority
- Priorities can be defined internally or externally
  - Internal: e.g. I/O bound or CPU bound
  - External: e.g. based on importance to the user



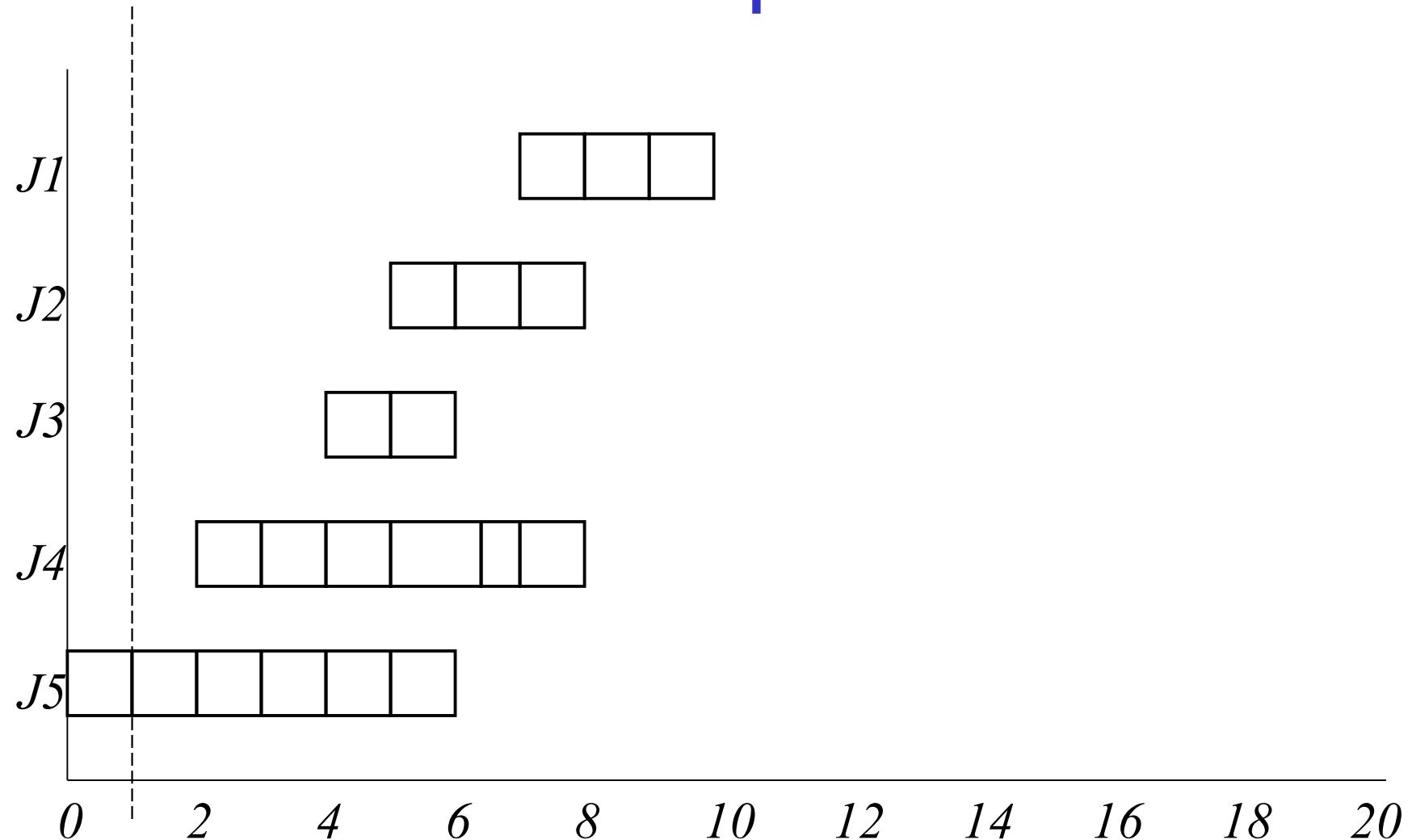
# Example



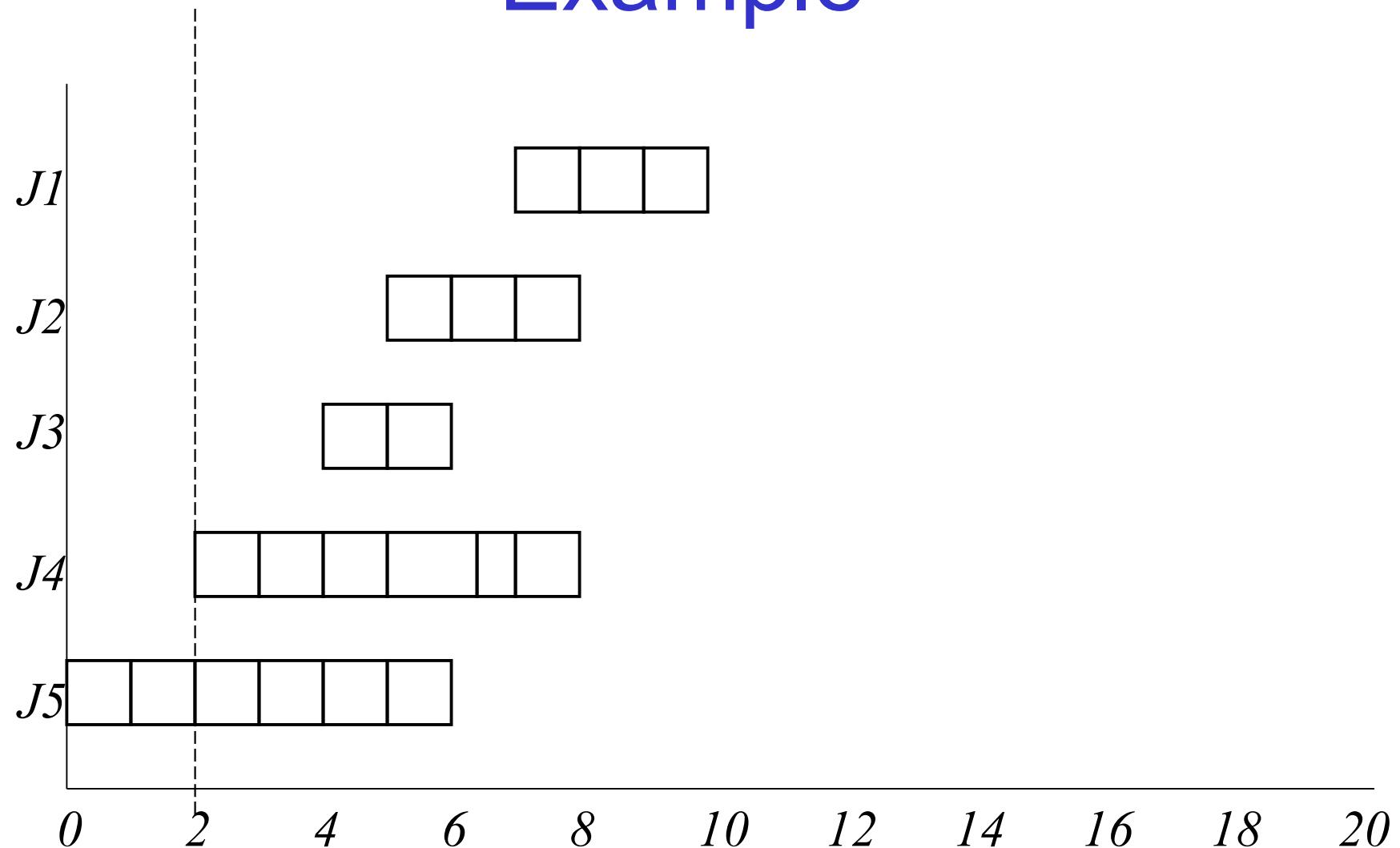
# Example



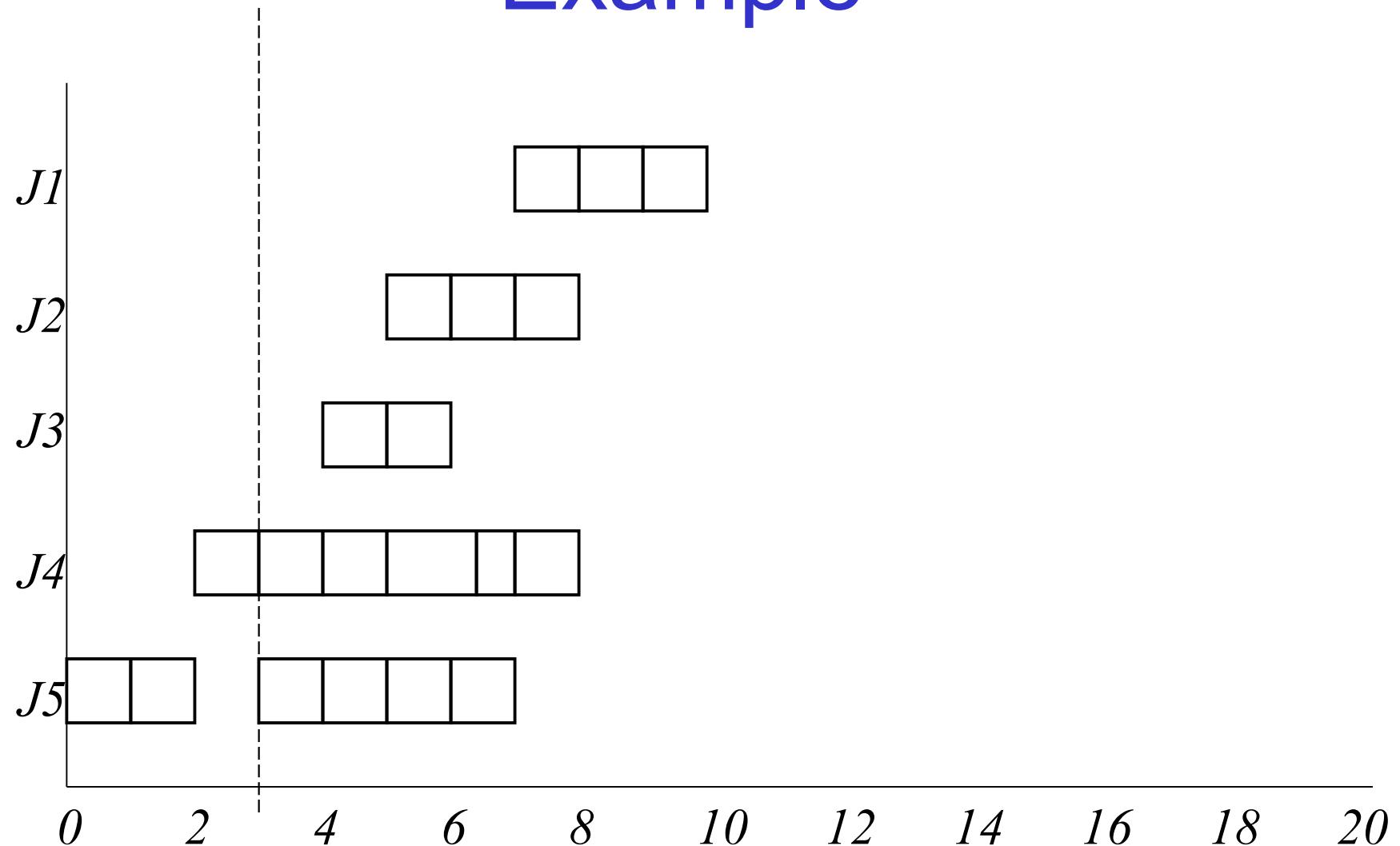
# Example



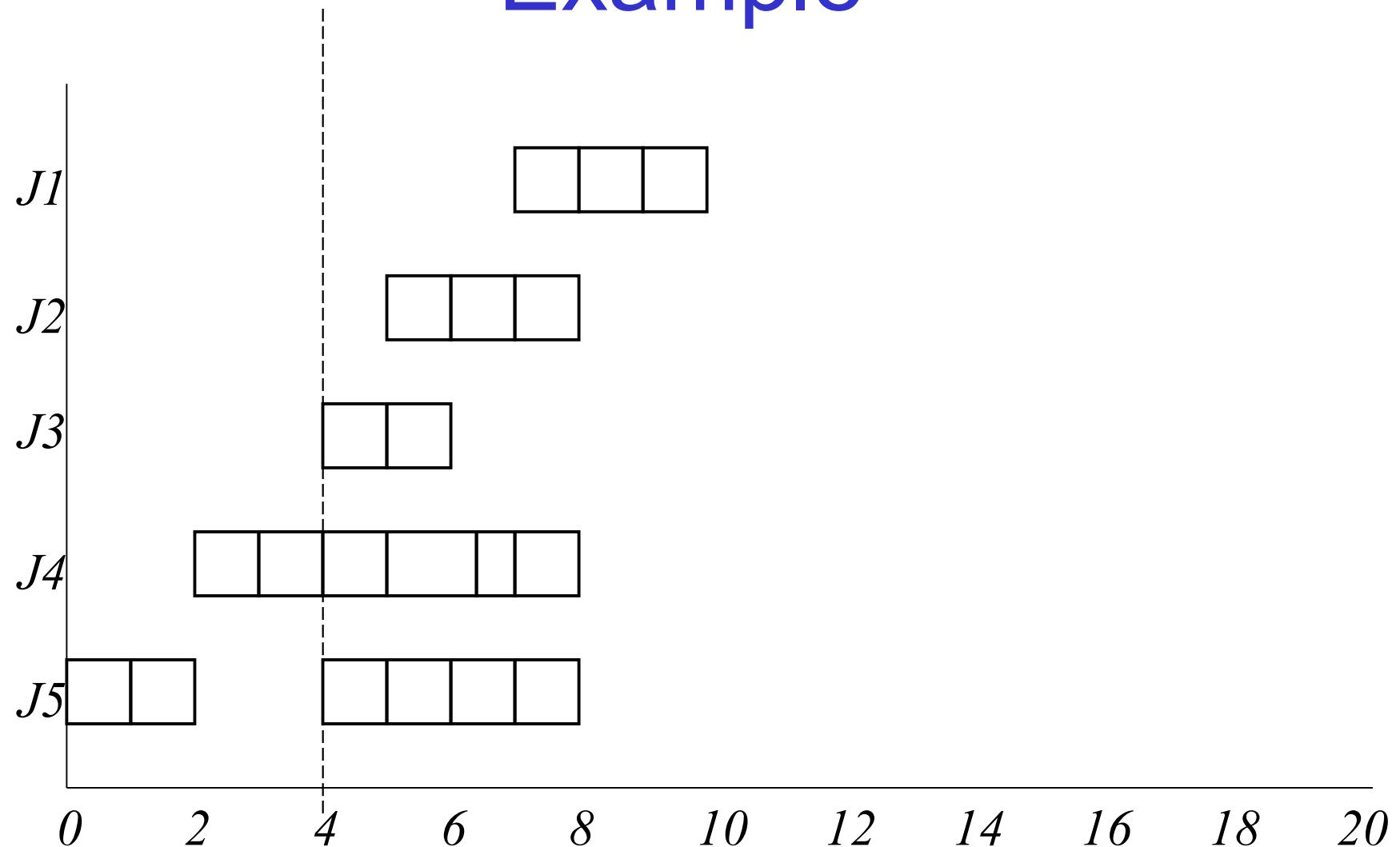
# Example



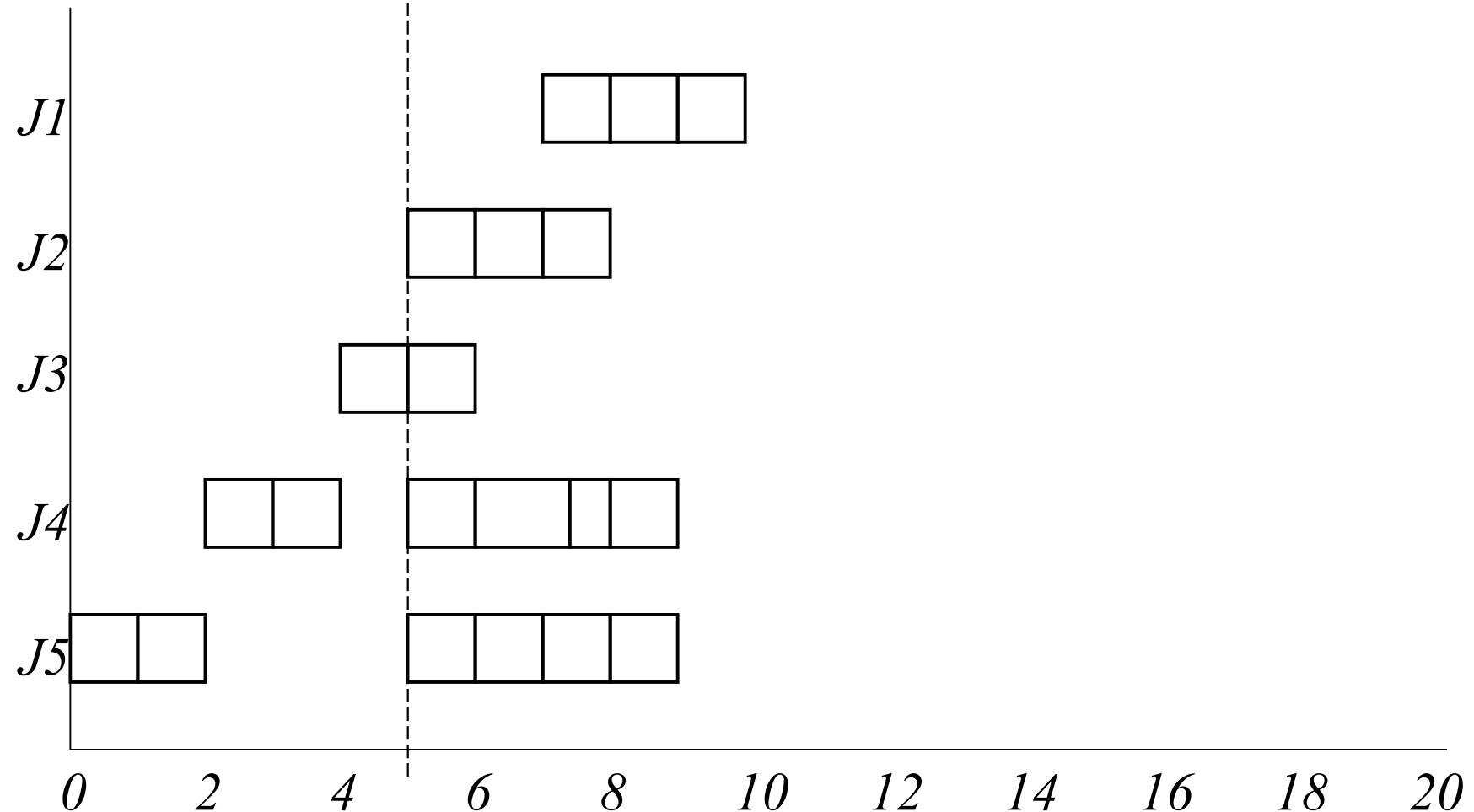
# Example



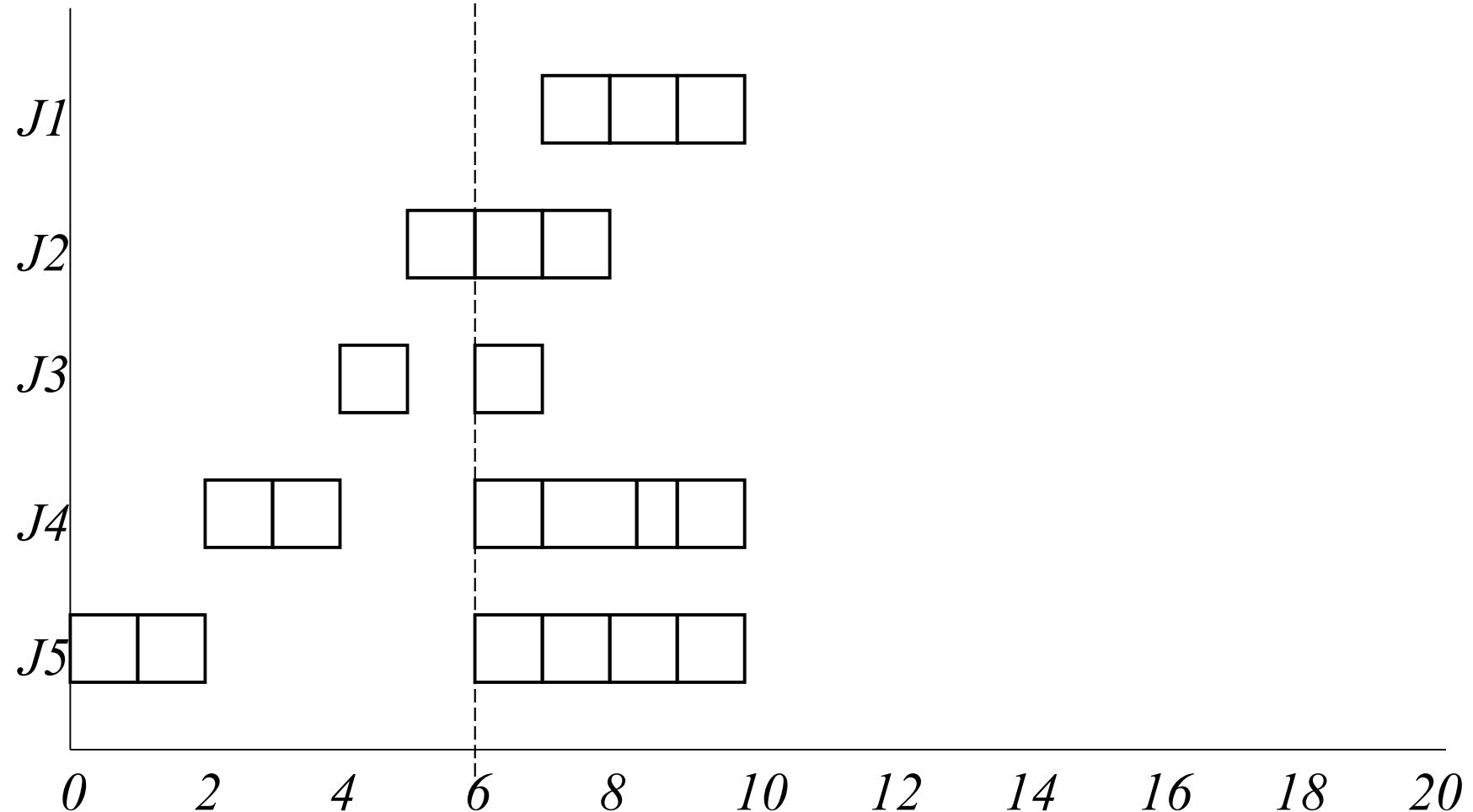
# Example



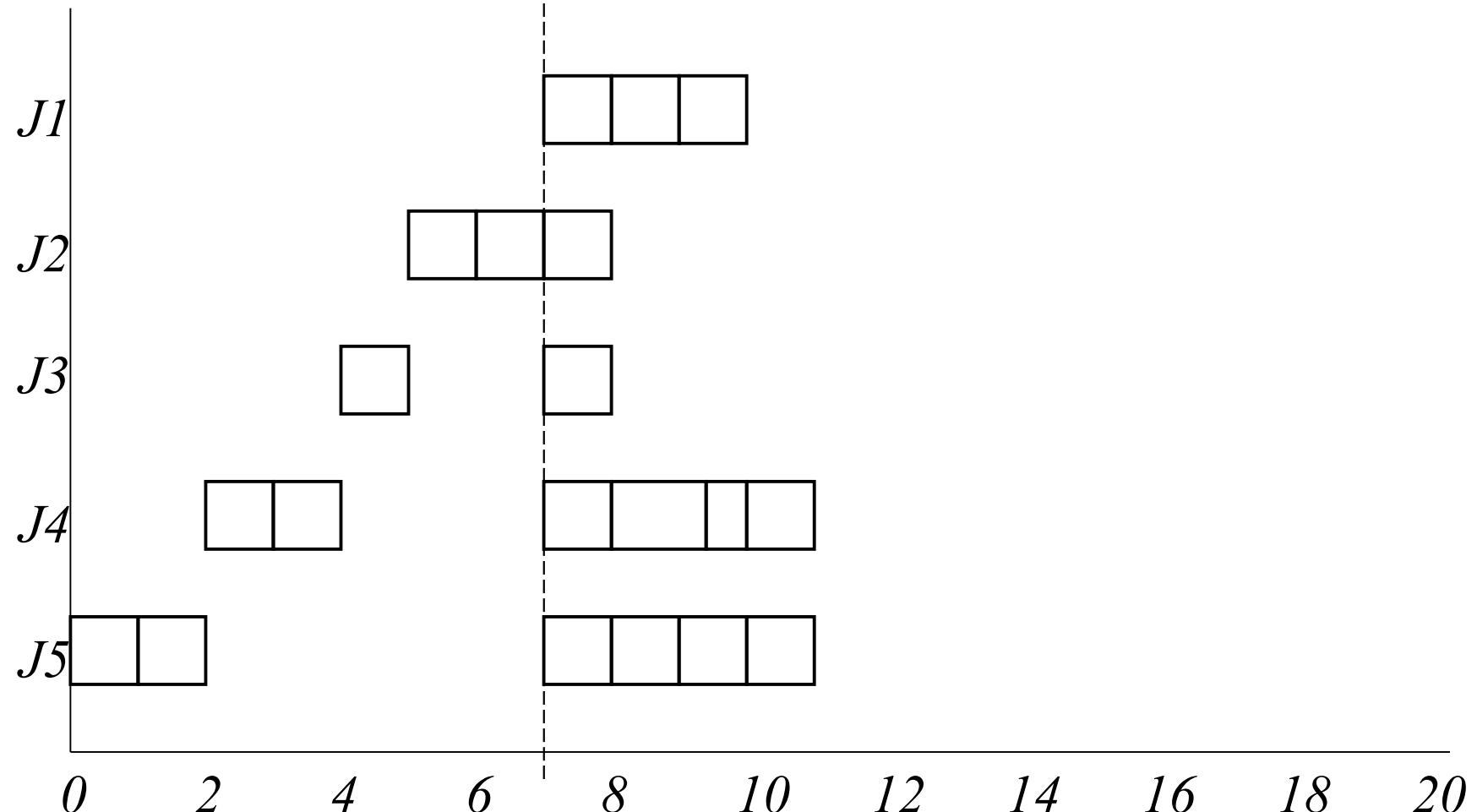
# Example



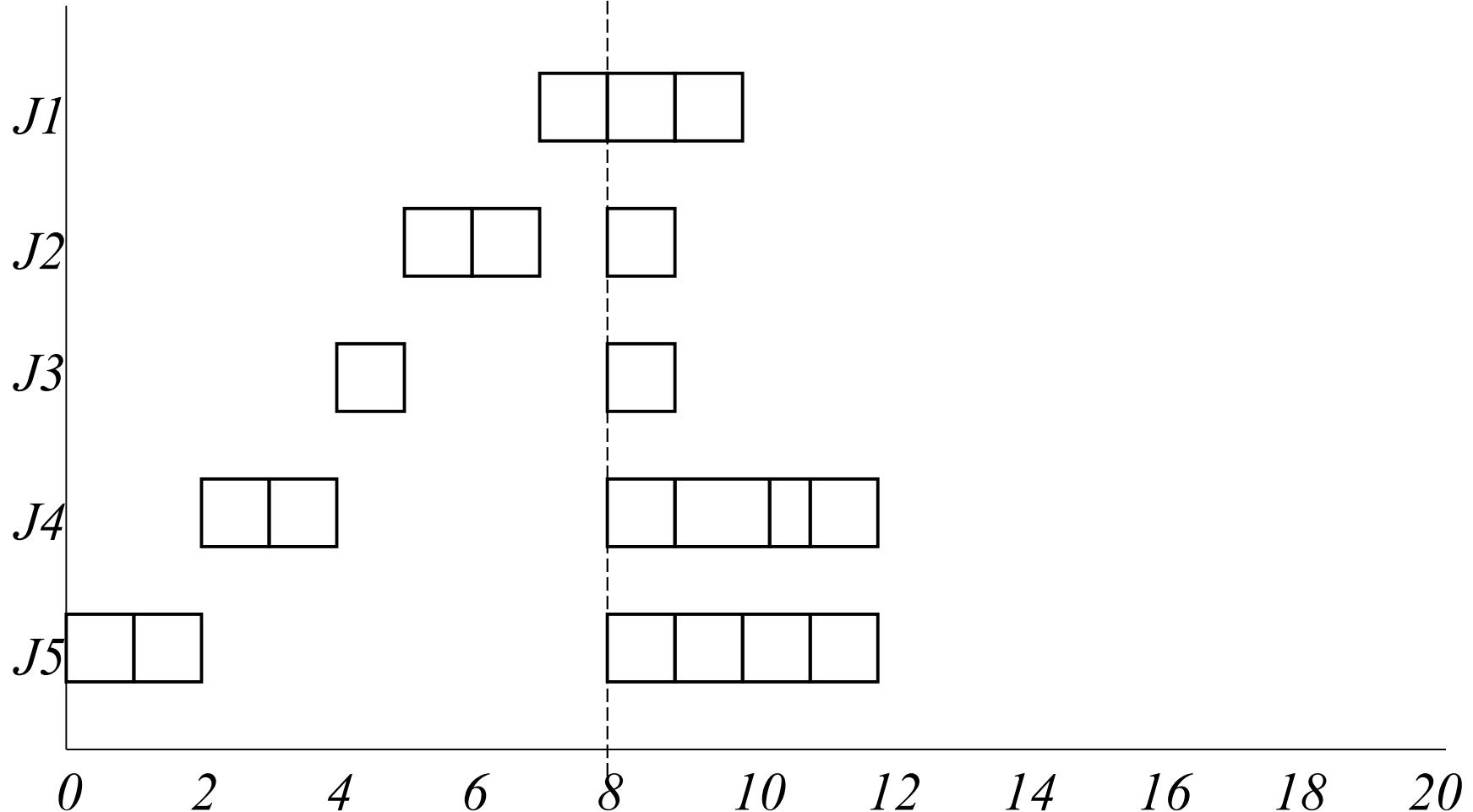
# Example



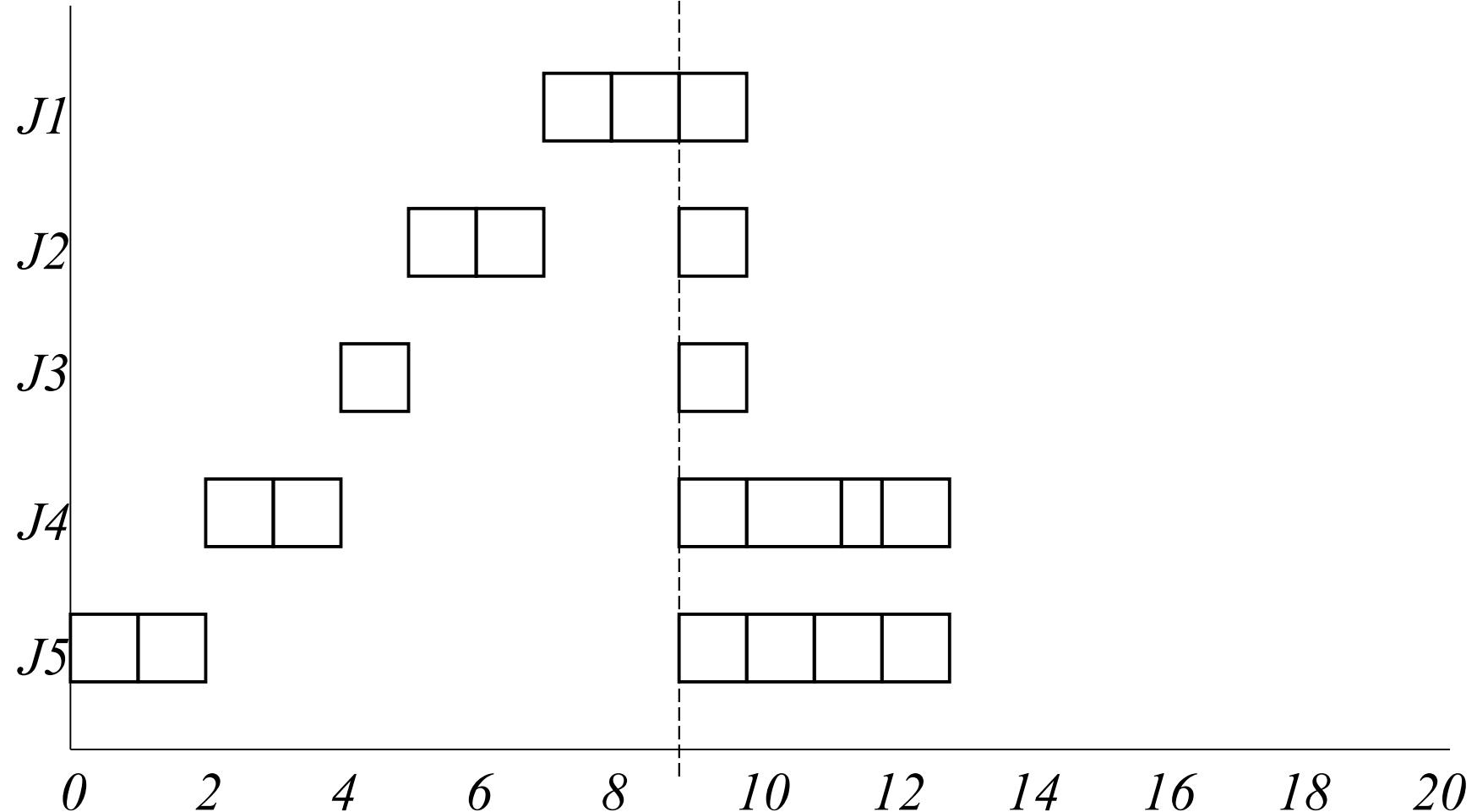
# Example



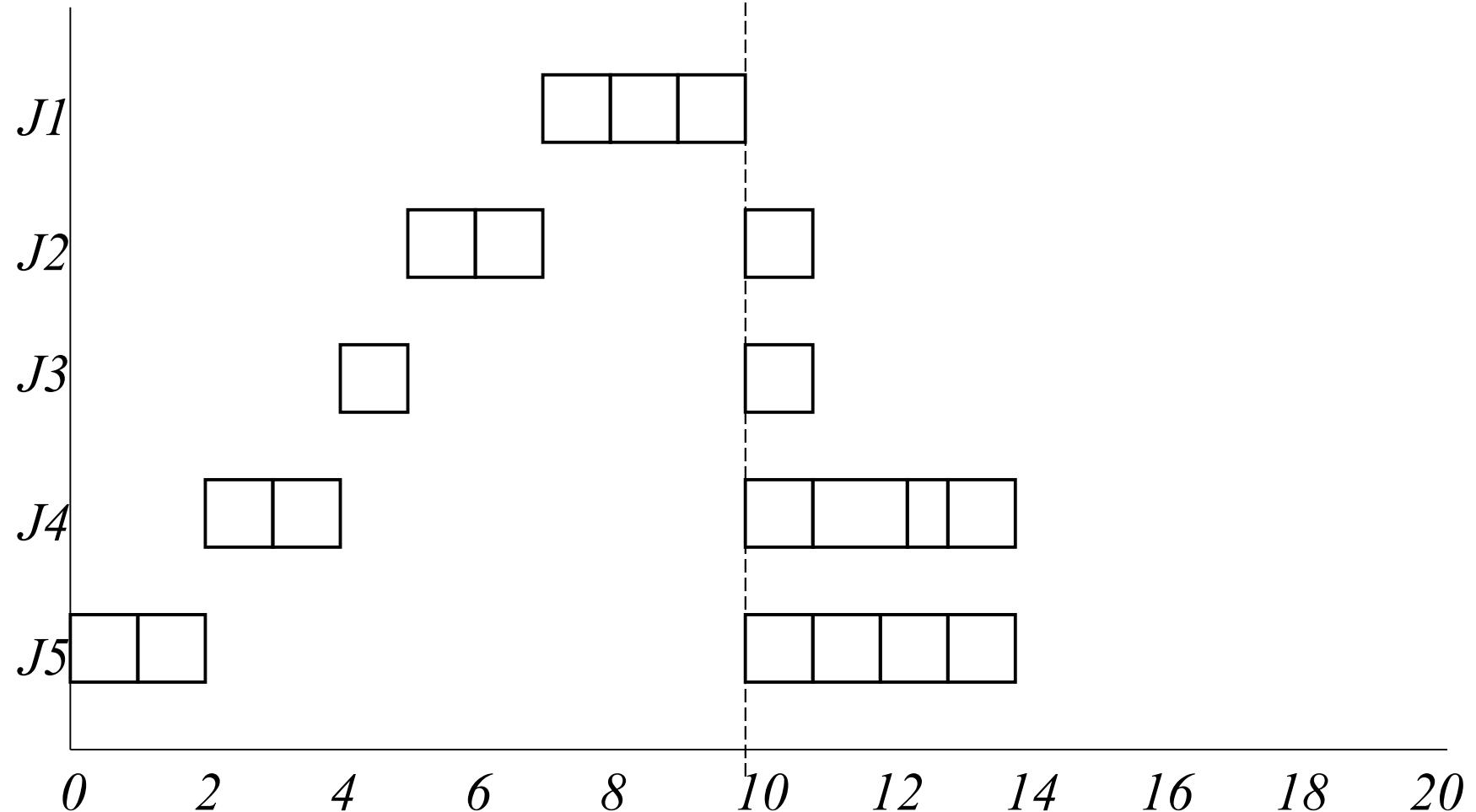
# Example



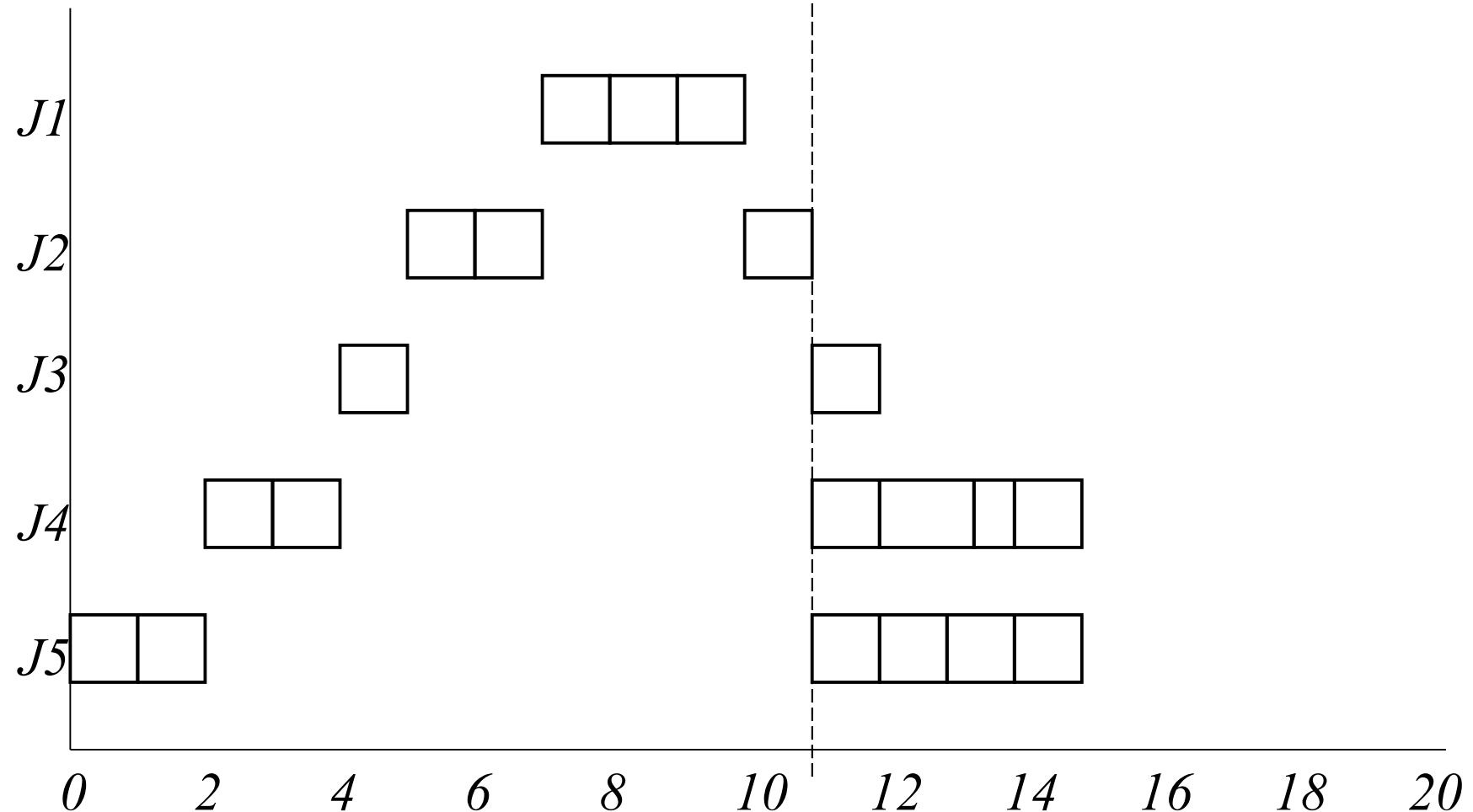
# Example



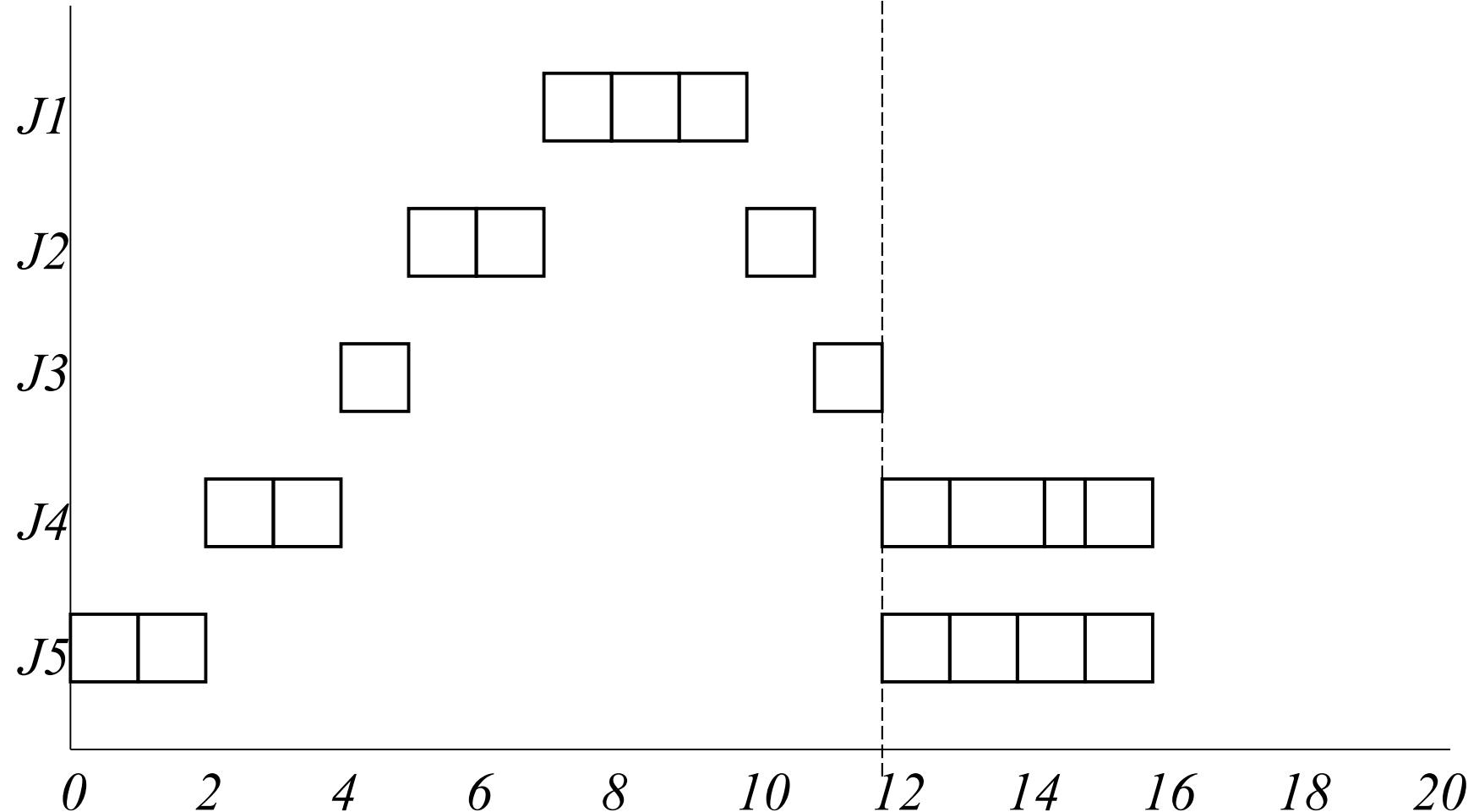
# Example



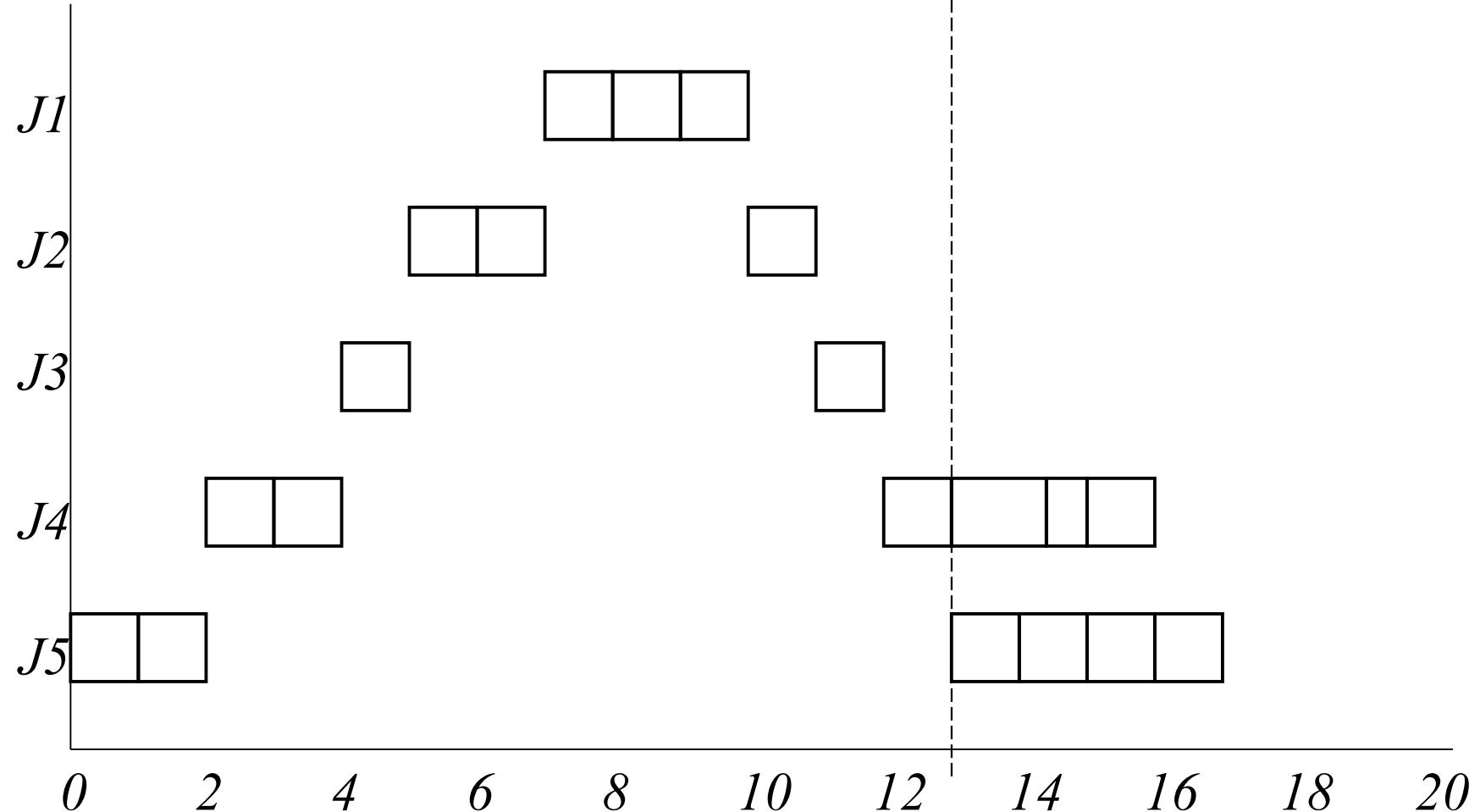
# Example



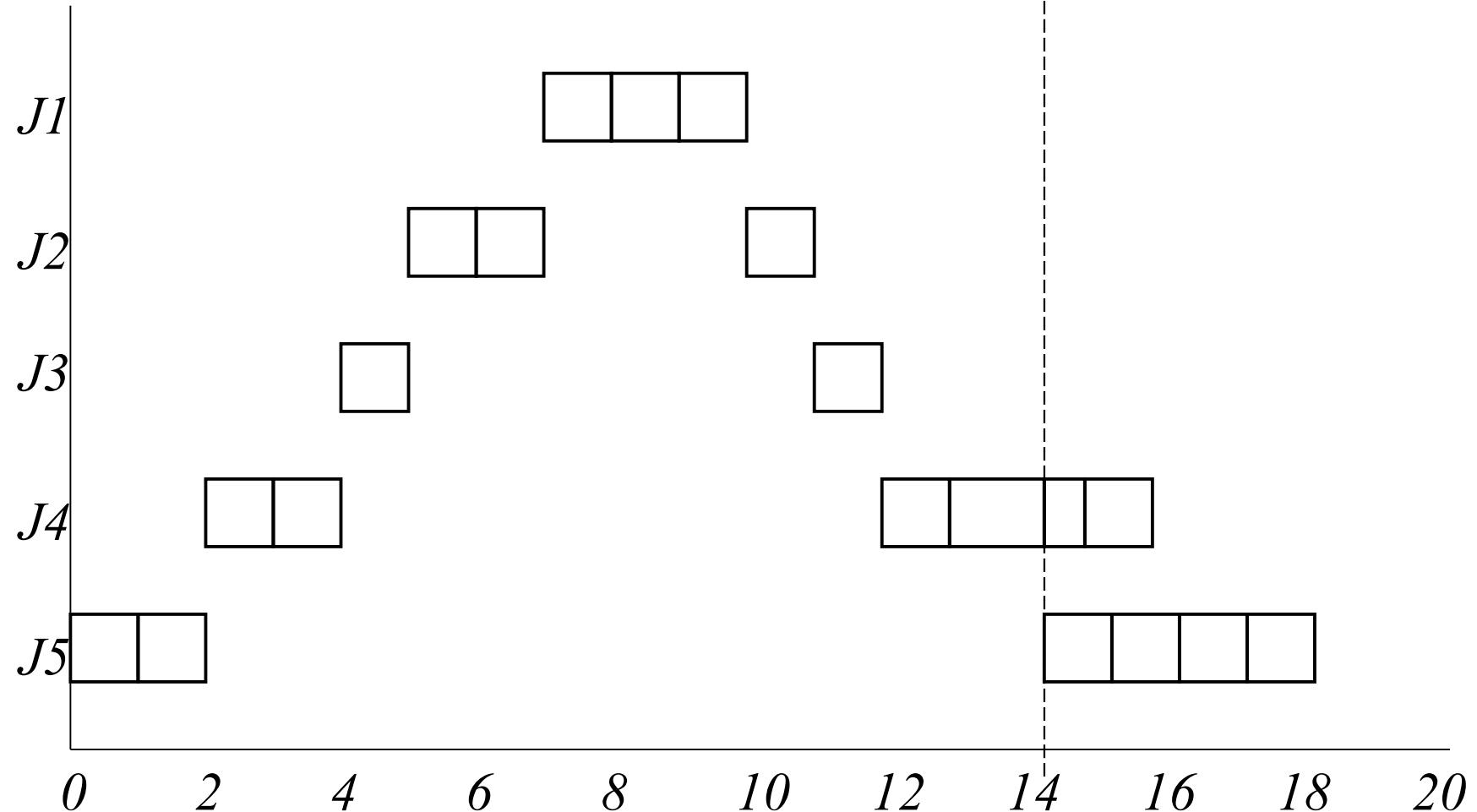
# Example



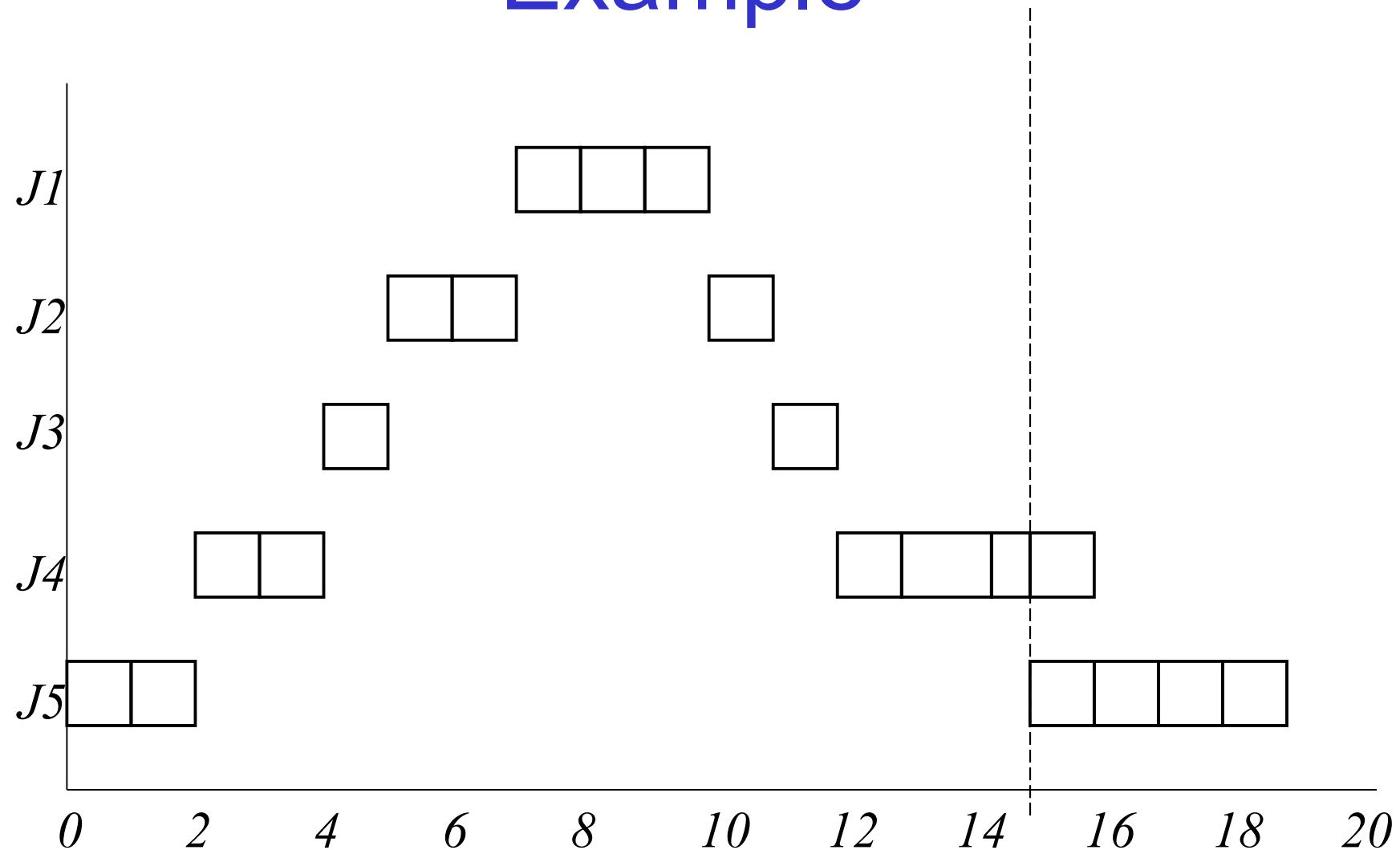
# Example



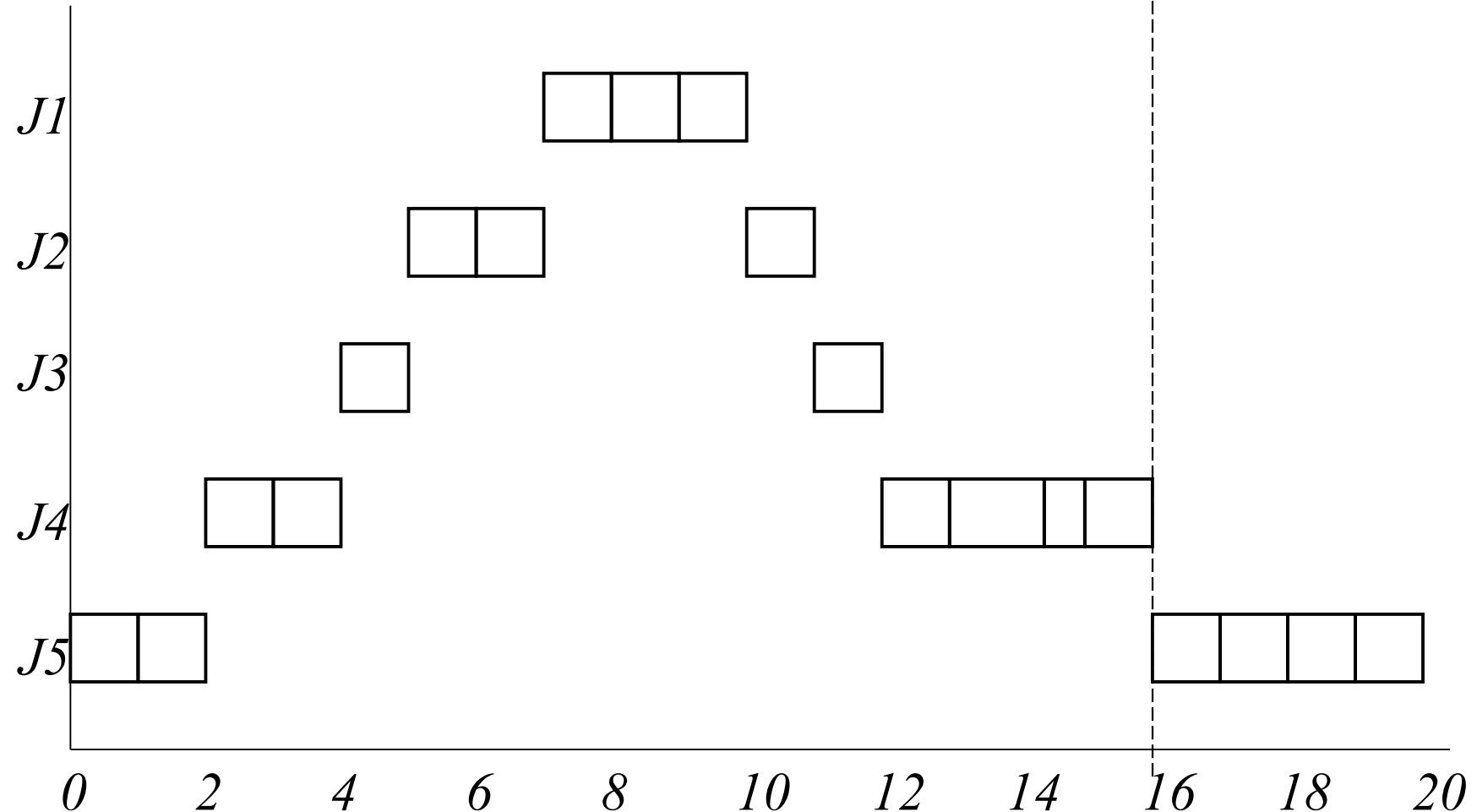
# Example



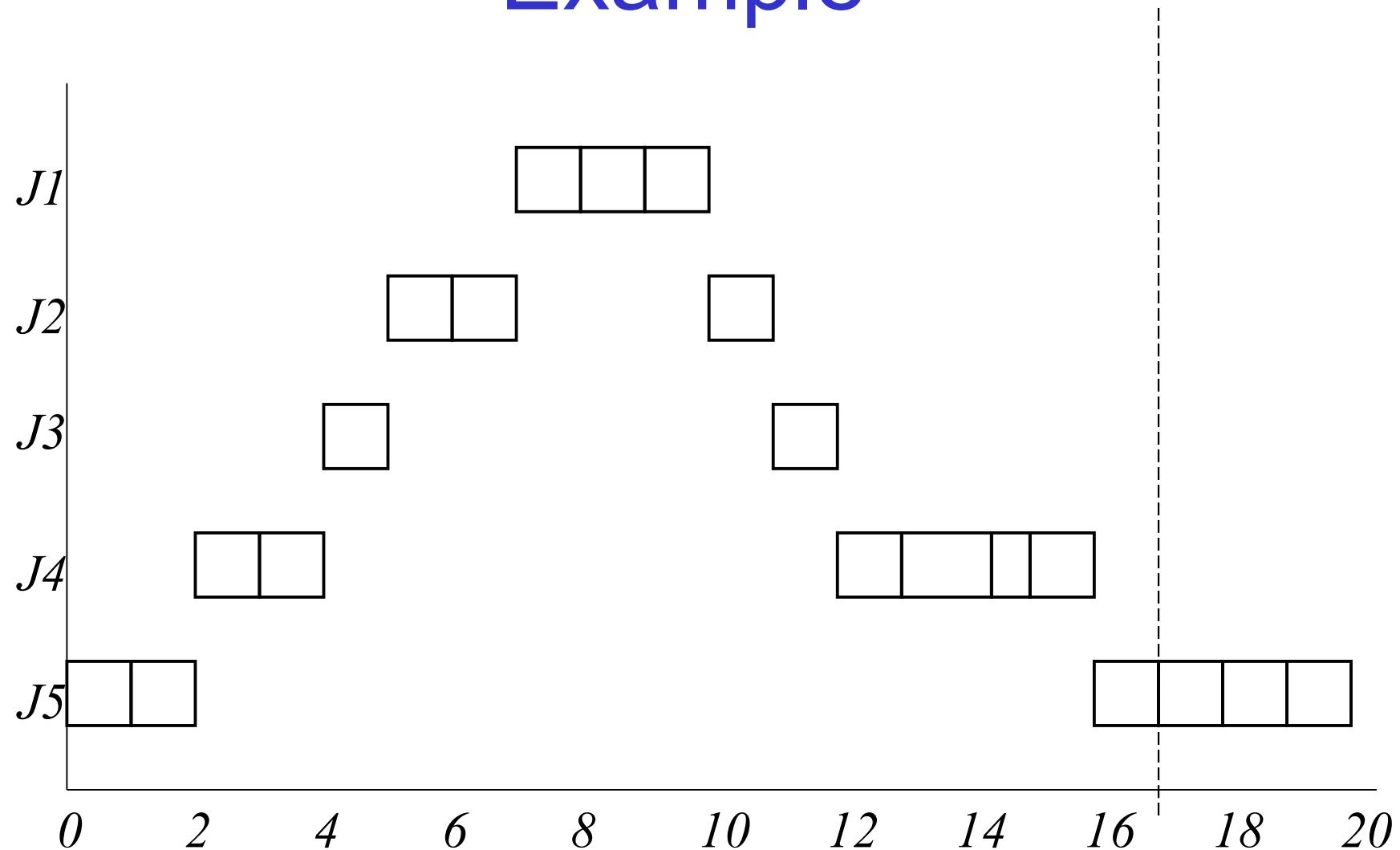
# Example



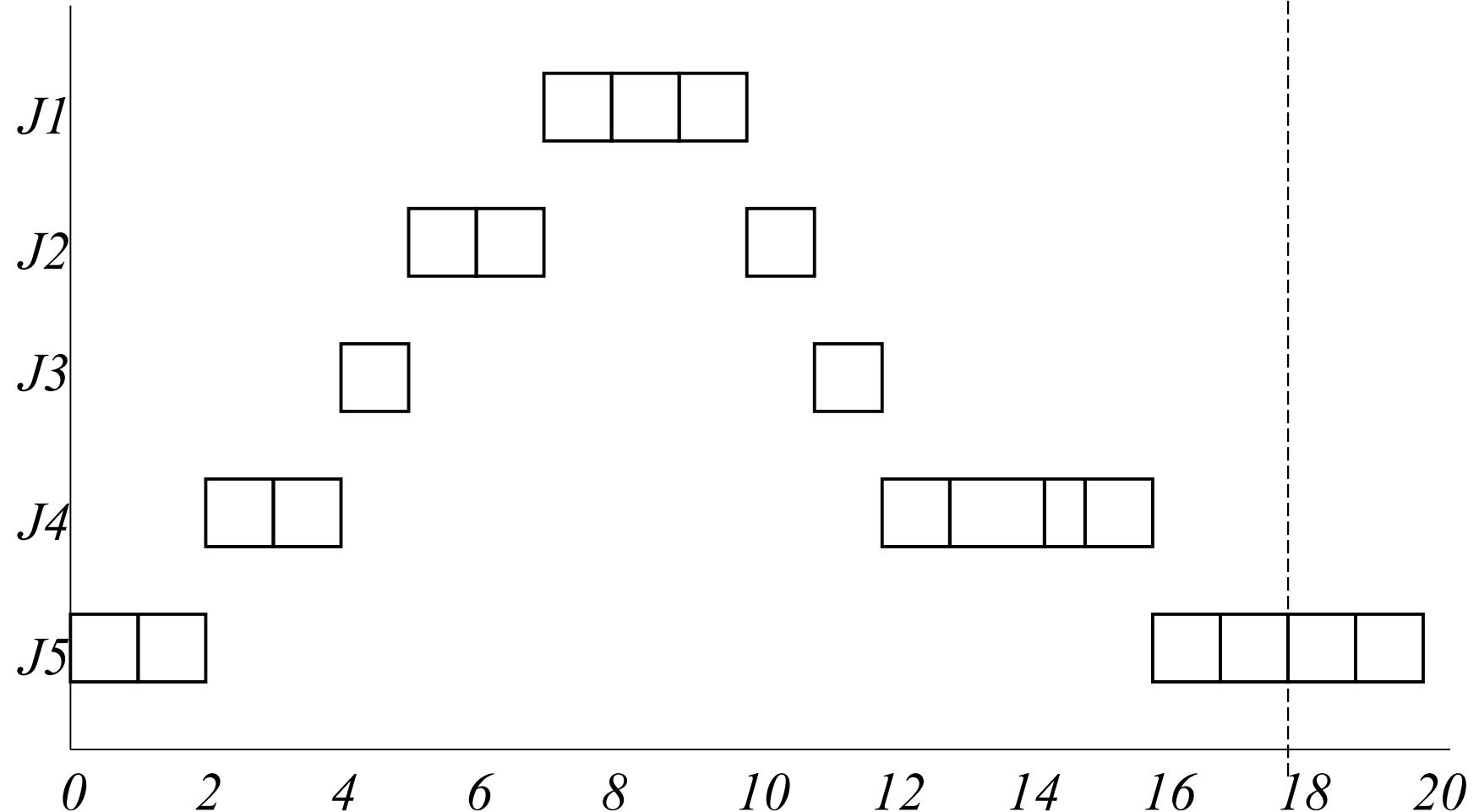
# Example



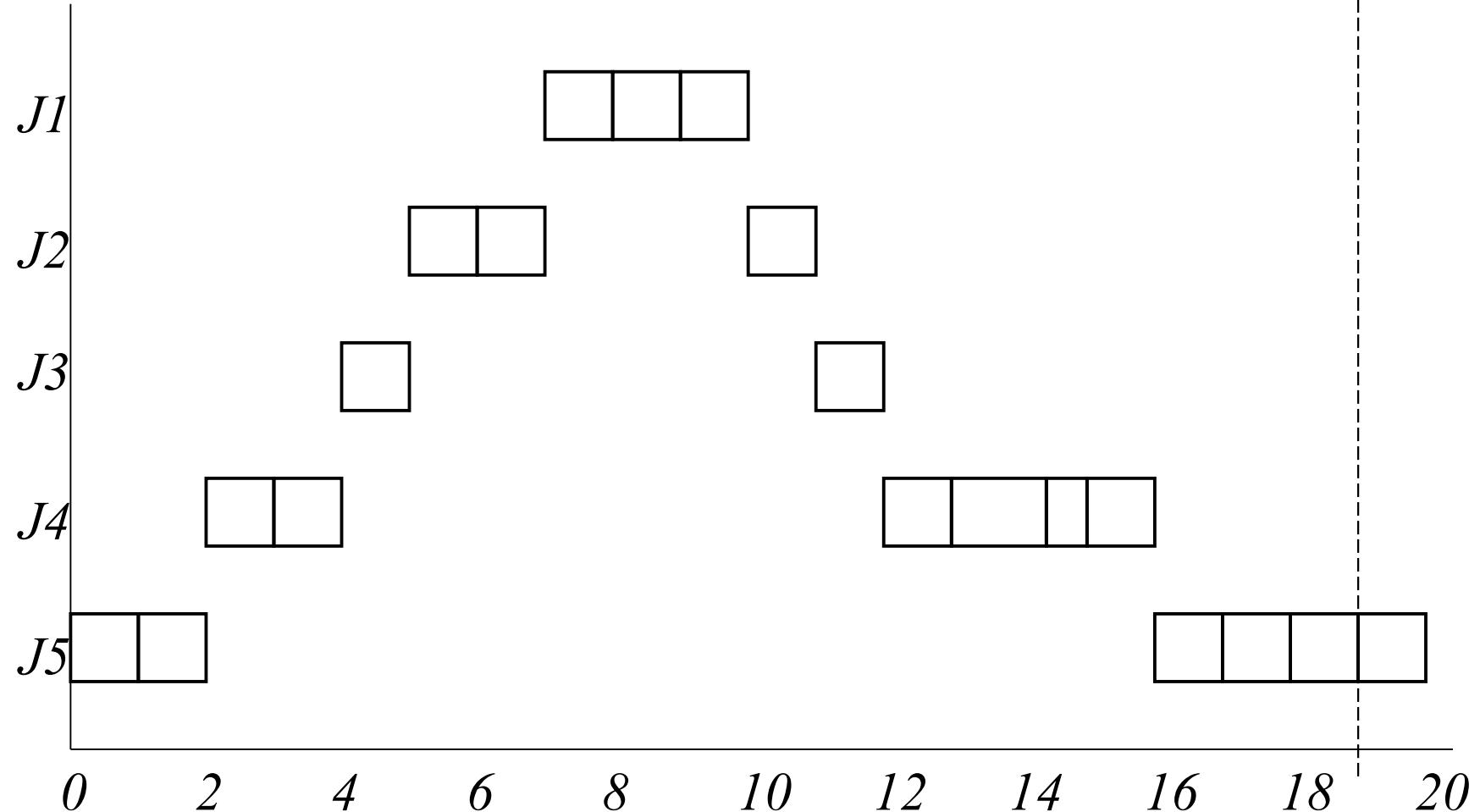
# Example



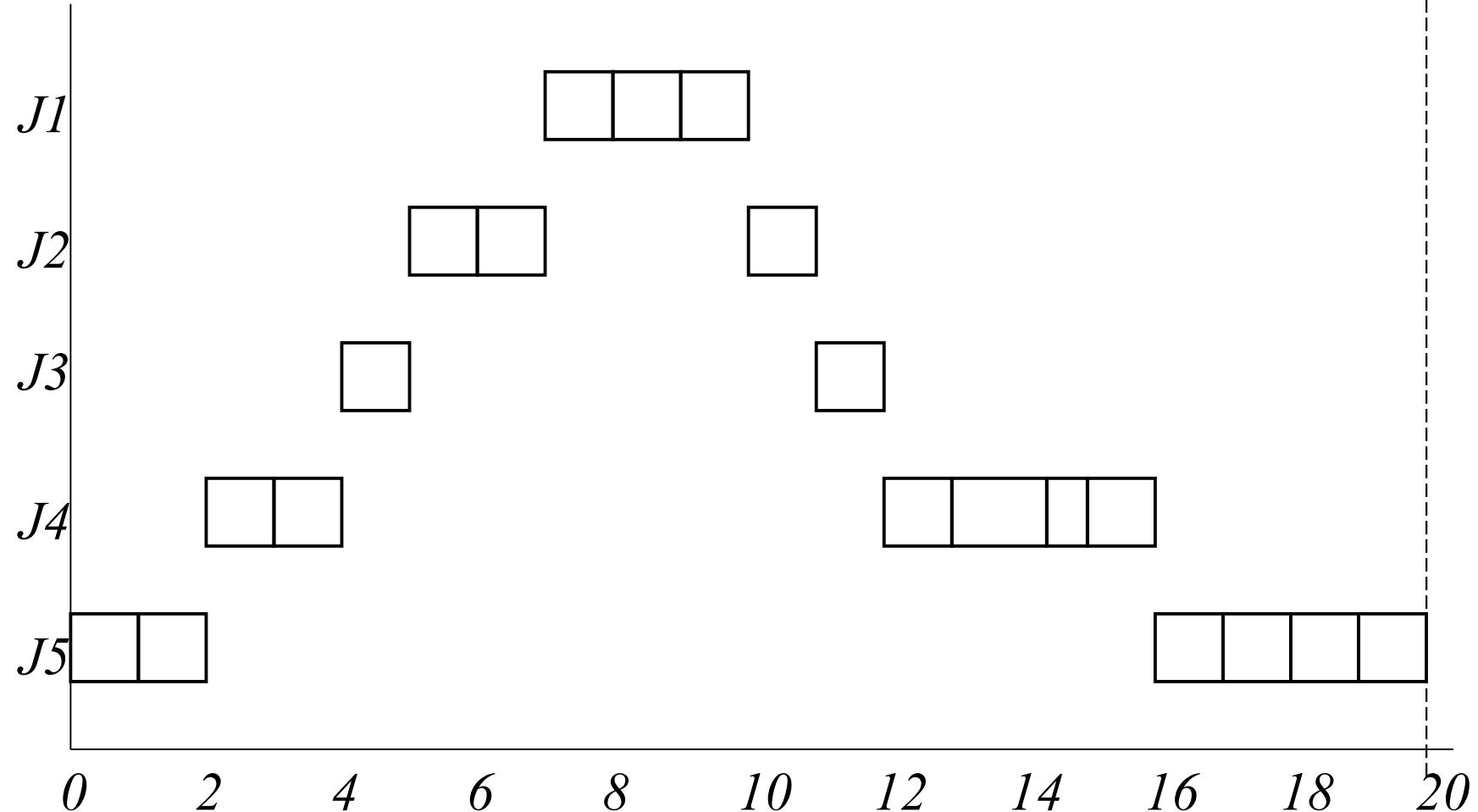
# Example



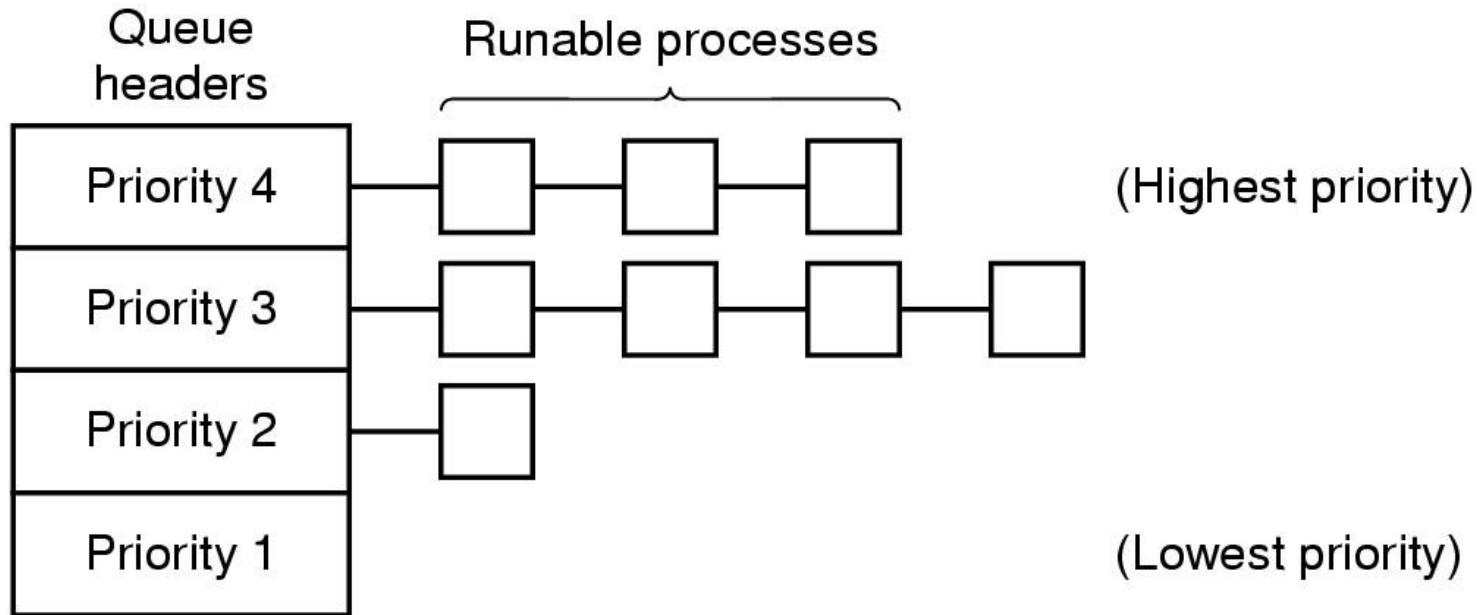
# Example



# Example



# Priorities

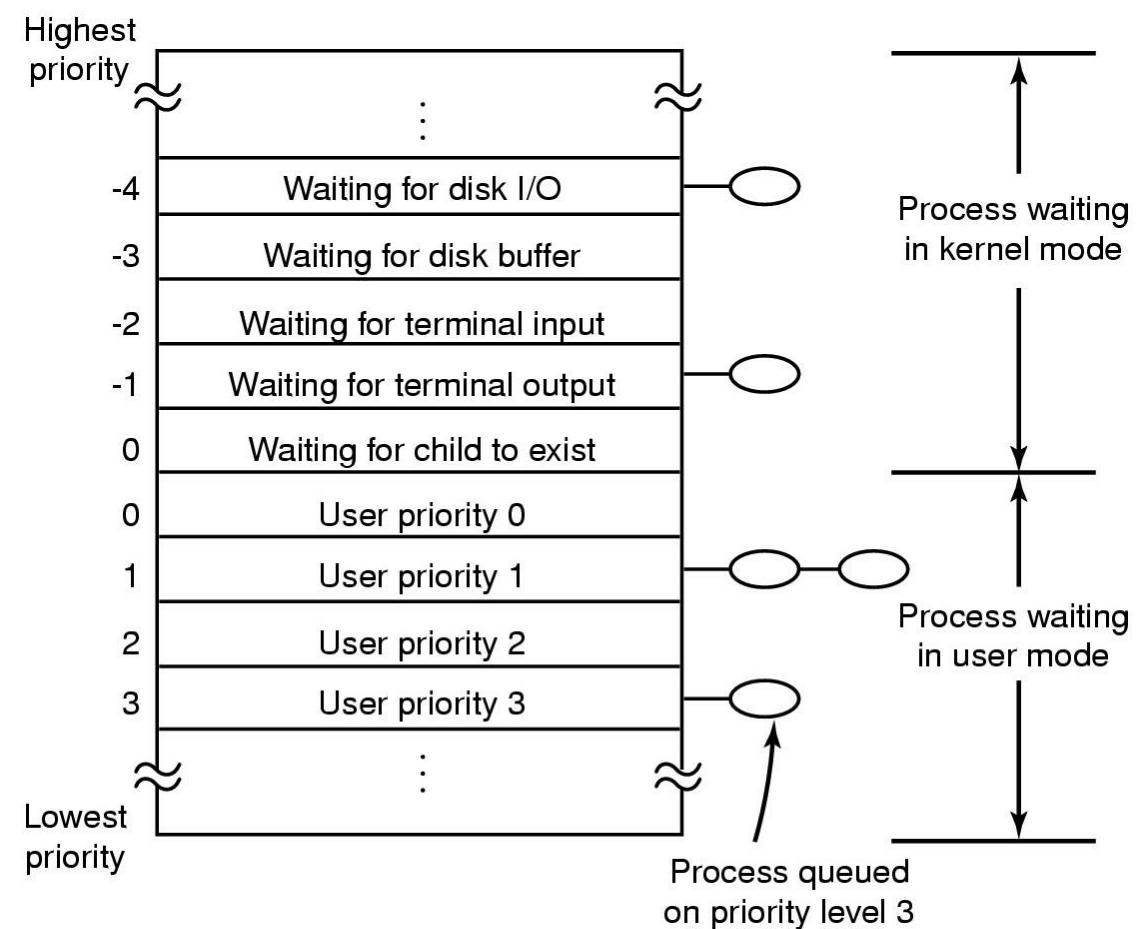


- Usually implemented by multiple priority queues, with round robin on each queue
- Con
  - Low priorities can starve
    - Need to adapt priorities periodically
      - Based on ageing or execution history



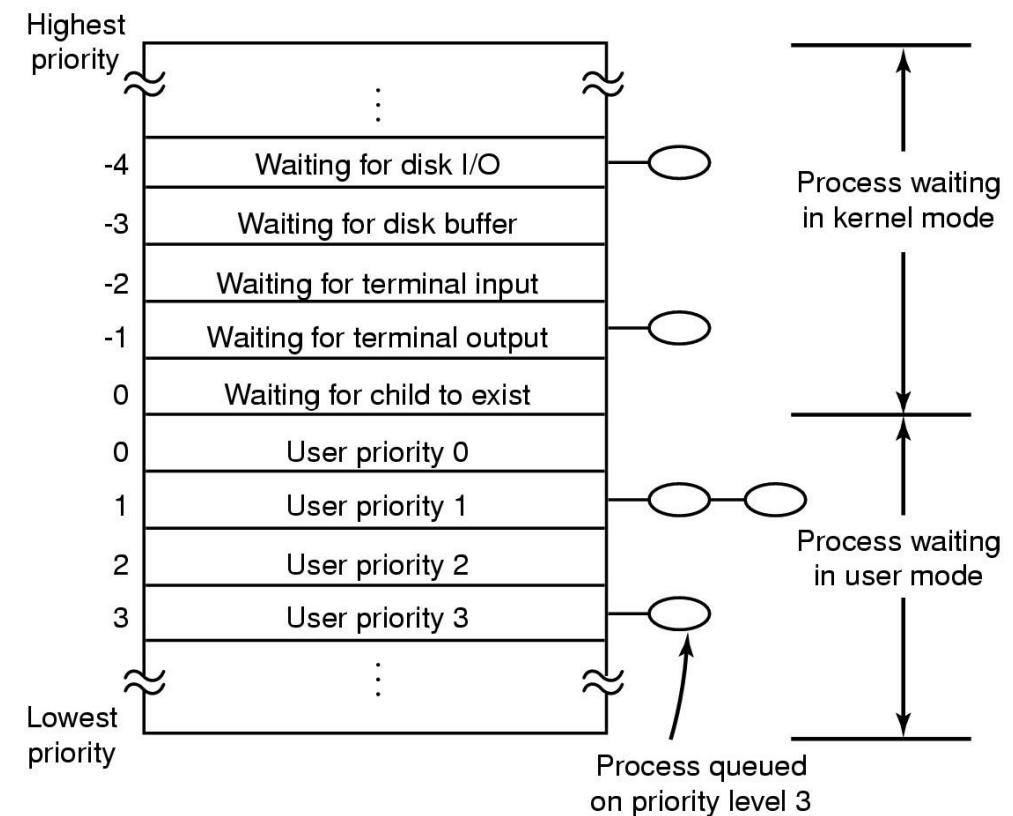
# Traditional UNIX Scheduler

- Two-level scheduler
  - High-level scheduler schedules processes between memory and disk
  - Low-level scheduler is CPU scheduler
    - Based on a multi-level queue structure with round robin at each level



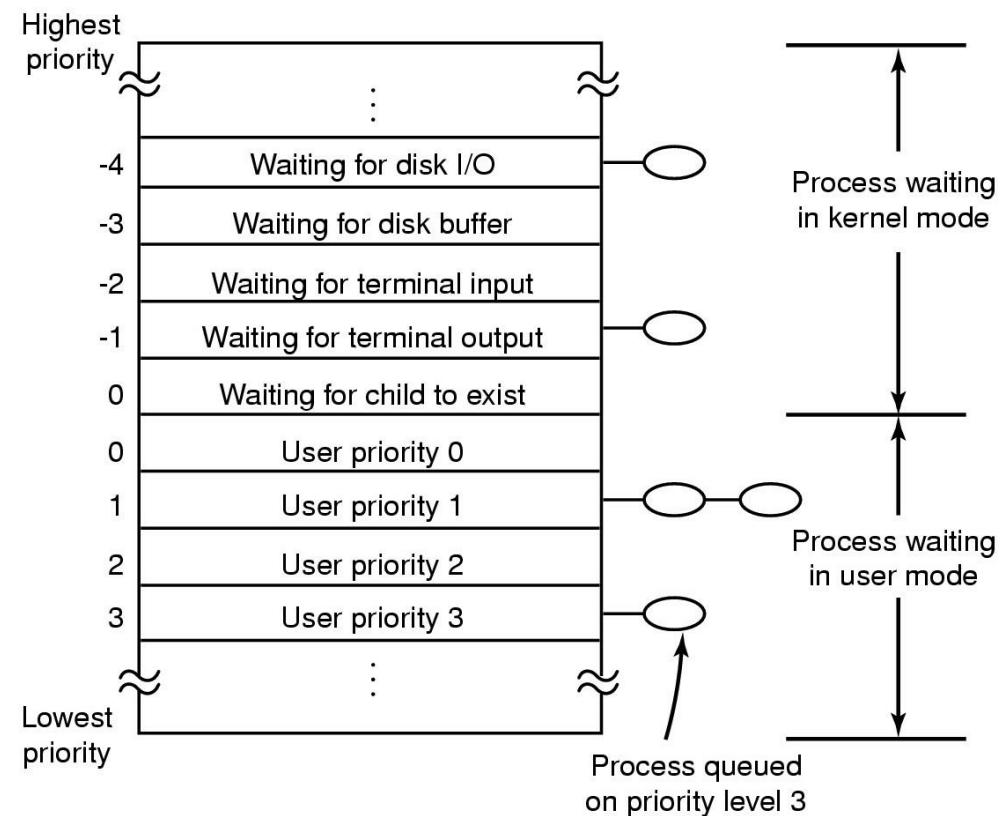
# Traditional UNIX Scheduler

- The highest priority (lower number) is scheduled
- Priorities are re-calculated once per second, and re-inserted in appropriate queue
  - Avoid starvation of low priority threads
  - Penalise CPU-bound threads



# Traditional UNIX Scheduler

- $\text{Priority} = \text{CPU\_usage} + \text{nice} + \text{base}$ 
  - $\text{CPU\_usage}$  = number of clock ticks
    - Decays over time to avoid permanently penalising the process
  - $\text{Nice}$  is a value given to the process by a user to permanently boost or reduce its priority
    - Reduce priority of background jobs
  - $\text{Base}$  is a set of hardwired, negative values used to boost priority of I/O bound system activities
    - Swapper, disk I/O, Character I/O



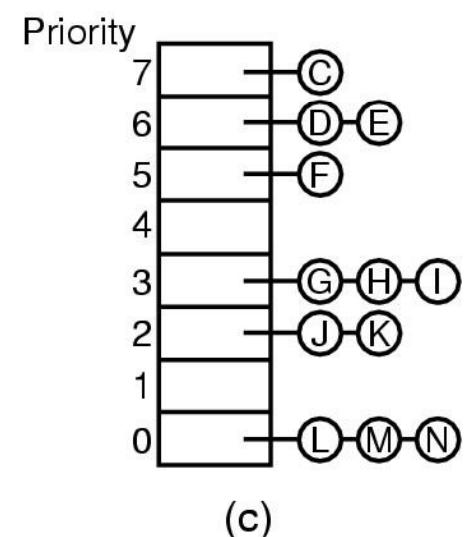
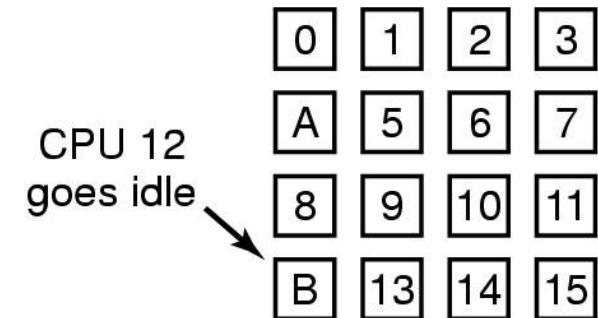
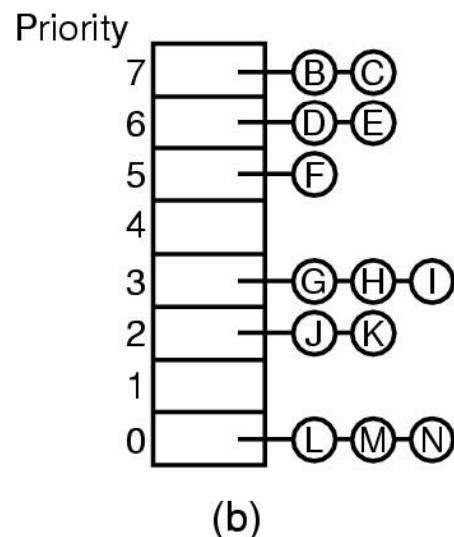
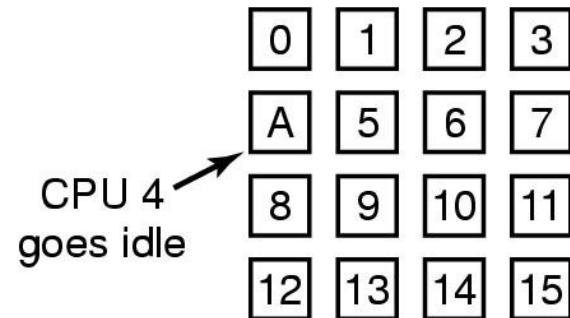
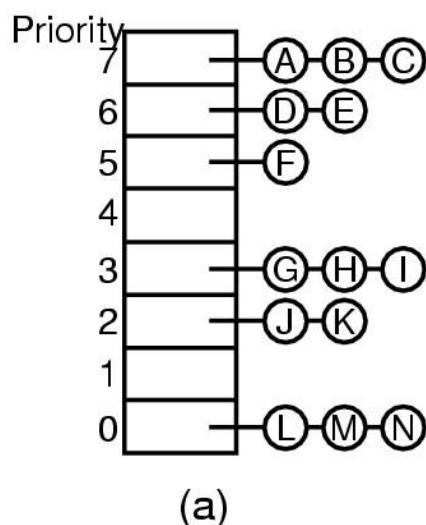
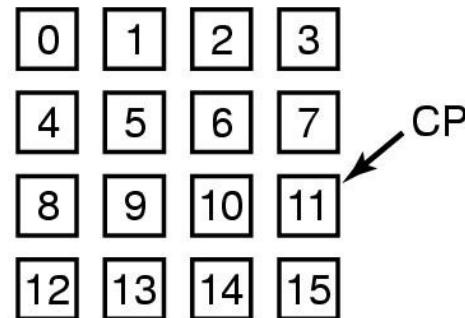
# Multiprocessor Scheduling

- Given  $X$  processes (or threads) and  $Y$  CPUs,
  - how do we allocate them to the CPUs



# A Single Shared Ready Queue

- When a CPU goes idle, it takes the highest priority process from the shared ready queue



# Single Shared Ready Queue

- Pros
  - Simple
  - Automatic load balancing
- Cons
  - Lock contention on the ready queue can be a major bottleneck
    - Due to frequent scheduling or many CPUs or both
  - Not all CPUs are equal
    - The last CPU a process ran on is likely to have more related entries in the cache.



# Affinity Scheduling

- Basic Idea
  - Try hard to run a process on the CPU it ran on last time
- One approach: *Multiple Queue Multiprocessor Scheduling*



# Multiple Queue SMP Scheduling

- Each CPU has its own ready queue
- Coarse-grained algorithm assigns processes to CPUs
  - Defines their affinity, and roughly balances the load
- The bottom-level fine-grained scheduler:
  - Is the frequently invoked scheduler (e.g. on blocking on I/O, a lock, or exhausting a timeslice)
  - Runs on each CPU and selects from its own ready queue
    - Ensures affinity
  - If nothing is available from the local ready queue, it runs a process from another CPUs ready queue rather than go idle
    - Termed “Work stealing”



# Multiple Queue SMP Scheduling

- Pros
  - No lock contention on per-CPU ready queues in the (hopefully) common case
  - Load balancing to avoid idle queues
  - Automatic affinity to a single CPU for more cache friendly behaviour



# I/O Management Intro

Chapter 5 - 5.3

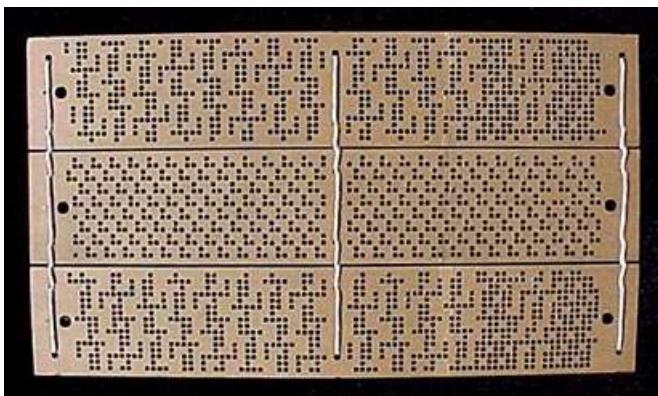


THE UNIVERSITY OF  
NEW SOUTH WALES

# Learning Outcomes

- A high-level understanding of the properties of a variety of I/O devices.
- An understanding of methods of interacting with I/O devices.





THE UNIVERSITY OF  
NEW SOUTH WALES

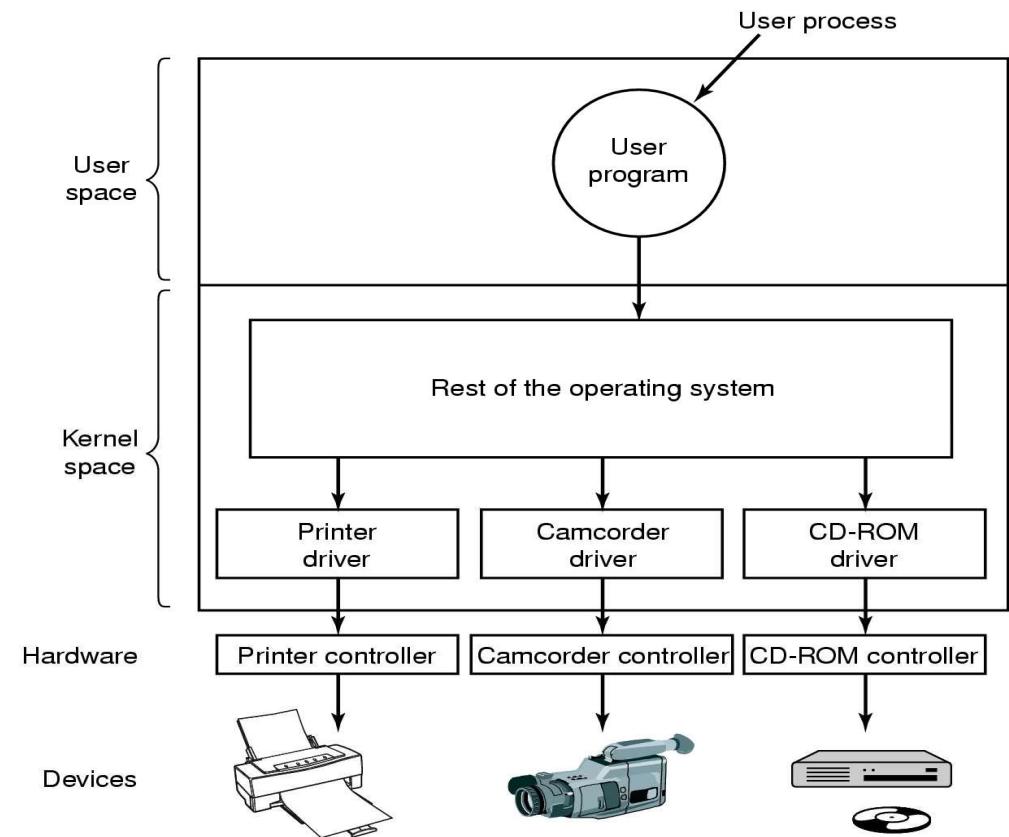
# I/O Devices

- There exists a large variety of I/O devices:
  - Many of them with different properties
  - They seem to require different interfaces to manipulate and manage them
    - We don't want a new interface for every device
    - Diverse, but similar interfaces leads to code duplication
- Challenge:
  - Uniform and efficient approach to I/O



- Logical position of device drivers is shown here
- Drivers (originally) compiled into the kernel
  - Including OS/161
  - Device installers were technicians
  - Number and types of devices rarely changed
- Nowadays they are dynamically loaded when needed
  - Linux modules
  - Typical users (device installers) can't build kernels
  - Number and types vary greatly
    - Even while OS is running (e.g hot-plug USB devices)

# Device Drivers



# Device Drivers

- **Drivers classified into similar categories**
  - Block devices and character (stream of data) device
- **OS defines a standard (internal) interface to the different classes of devices**
  - Example: USB *Human Input Device* (HID) class specifications
    - human input devices follow a set of rules making it easier to design a standard interface.



# USB Device Classes

Base Class	Descriptor Usage	Description
00h	Device	Use class information in the Interface Descriptors
01h	Interface	Audio
02h	Both	Communications and CDC Control
03h	Interface	HID (Human Interface Device)
05h	Interface	Physical
06h	Interface	Image
07h	Interface	Printer
08h	Interface	Mass Storage
09h	Device	Hub
0Ah	Interface	CDC-Data
0Bh	Interface	Smart Card
0Dh	Interface	Content Security
0Eh	Interface	Video
0Fh	Interface	Personal Healthcare
10h	Interface	Audio/Video Devices
DCh	Both	Diagnostic Device
E0h	Interface	Wireless Controller
EFh	Both	Miscellaneous
FFh	Interface	Application Specific
FFh	Both	Vendor Specific



# I/O Device Handling

- Data rate
  - May be differences of several orders of magnitude between the data transfer rates
  - Example: Assume 1000 cycles/byte I/O
    - Keyboard needs 10 KHz processor to keep up
    - Gigabit Ethernet needs 100 GHz processor.....



# Sample Data Rates

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

USB 3.0 625 MB/s (5 Gb/s)  
Thunderbolt 2.5GB/sec (20 Gb/s)  
PCIe v3.0 x16 16GB/s



# Device Drivers

- **Device drivers job**
  - translate request through the device-independent standard interface (open, close, read, write) into appropriate sequence of commands (register manipulations) for the particular hardware
  - Initialise the hardware at boot time, and shut it down cleanly at shutdown



# Device Driver

- **After issuing the command to the device, the device either**
  - Completes immediately and the driver simply returns to the caller
  - Or, device must process the request and the driver usually blocks waiting for an I/O complete interrupt.
- **Drivers are thread-safe** as they can be called by another process while a process is already blocked in the driver.
  - Thread-safe: Synchronised...



# Device-Independent I/O Software

- There is commonality between drivers of similar classes
- Divide I/O software into device-dependent and device-independent I/O software
- Device independent software includes
  - Buffer or Buffer-cache management
  - TCP/IP stack
  - Managing access to dedicated devices
  - Error reporting

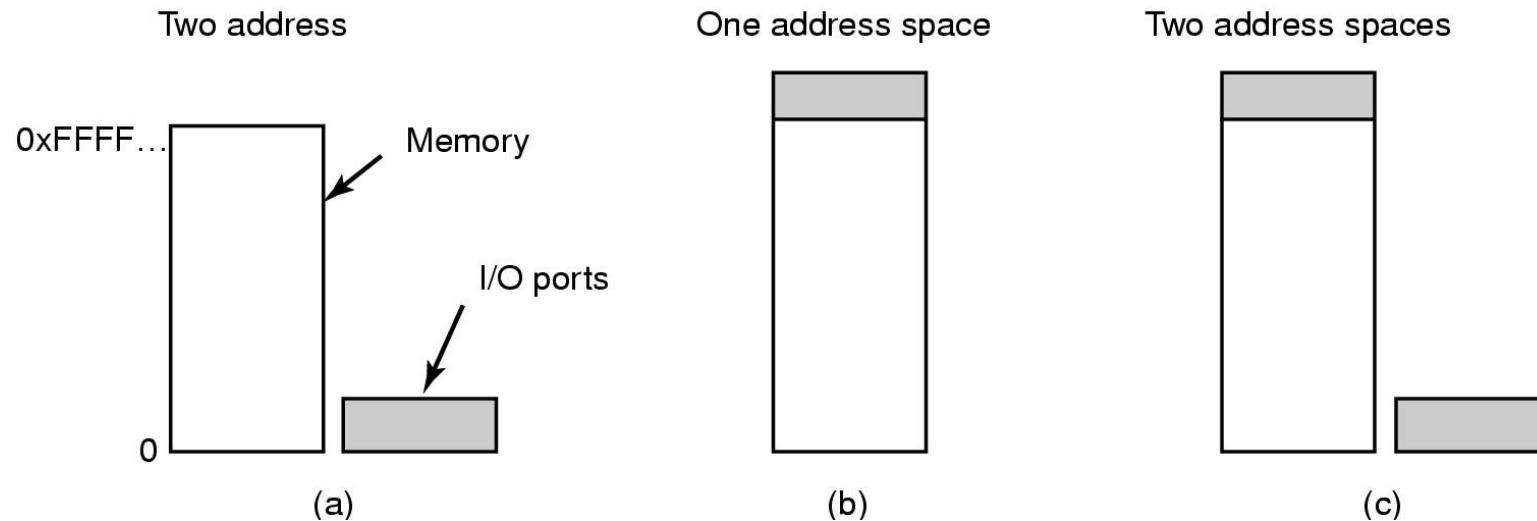


# Driver $\leftrightarrow$ Kernel Interface

- Major Issue is uniform interfaces to devices and kernel
  - Uniform device interface for kernel code
    - Allows different devices to be used the same way
      - No need to rewrite file-system to switch between SCSI, IDE or RAM disk
    - Allows internal changes to device driver with fear of breaking kernel code
  - Uniform kernel interface for device code
    - Drivers use a defined interface to kernel services (e.g. kmalloc, install IRQ handler, etc.)
    - Allows kernel to evolve without breaking existing drivers
  - Together both uniform interfaces avoid a lot of programming implementing new interfaces
    - Retains compatibility as drivers and kernels change over time.



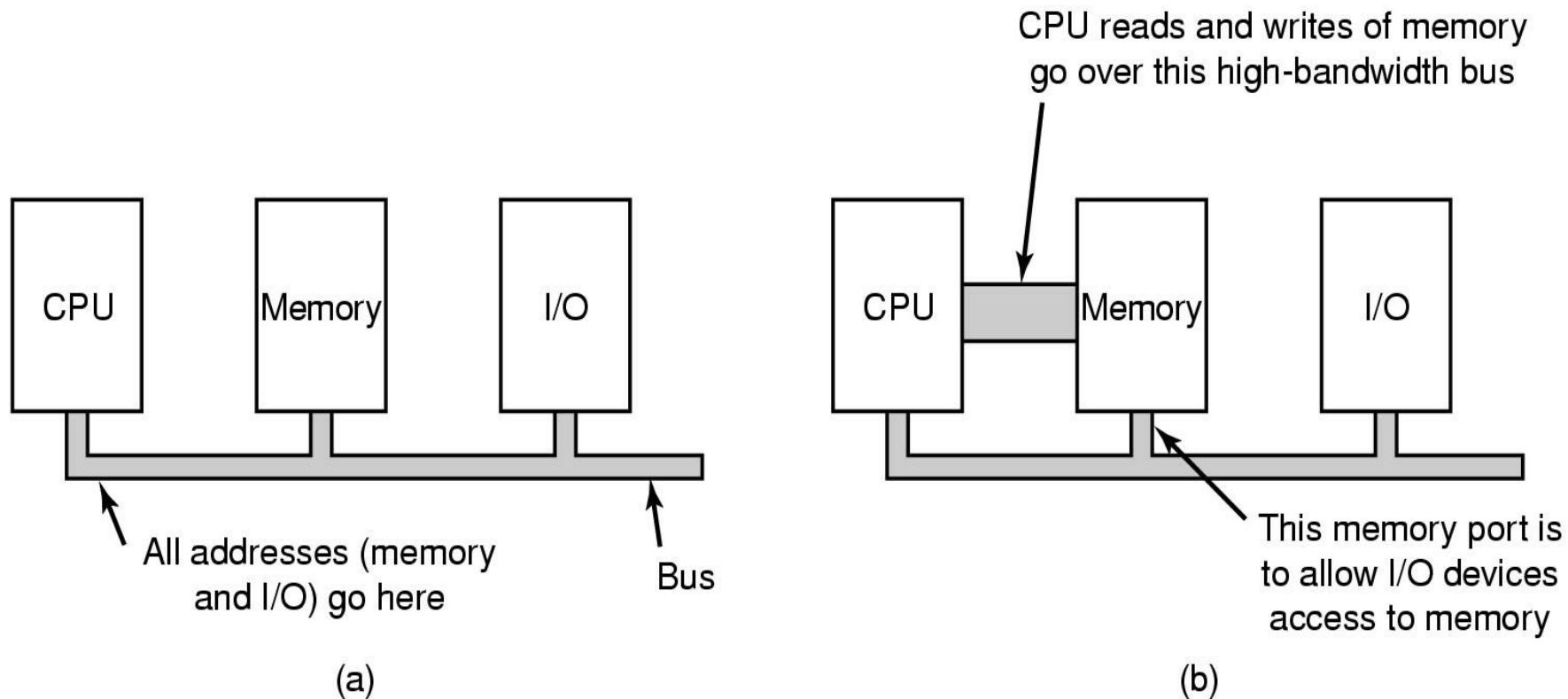
# Accessing I/O Controllers



- a) Separate I/O and memory space**
  - I/O controller registers appear as I/O ports
  - Accessed with special I/O instructions
- b) Memory-mapped I/O**
  - Controller registers appear as memory
  - Use normal load/store instructions to access
- c) Hybrid**
  - x86 has both ports and memory mapped I/O



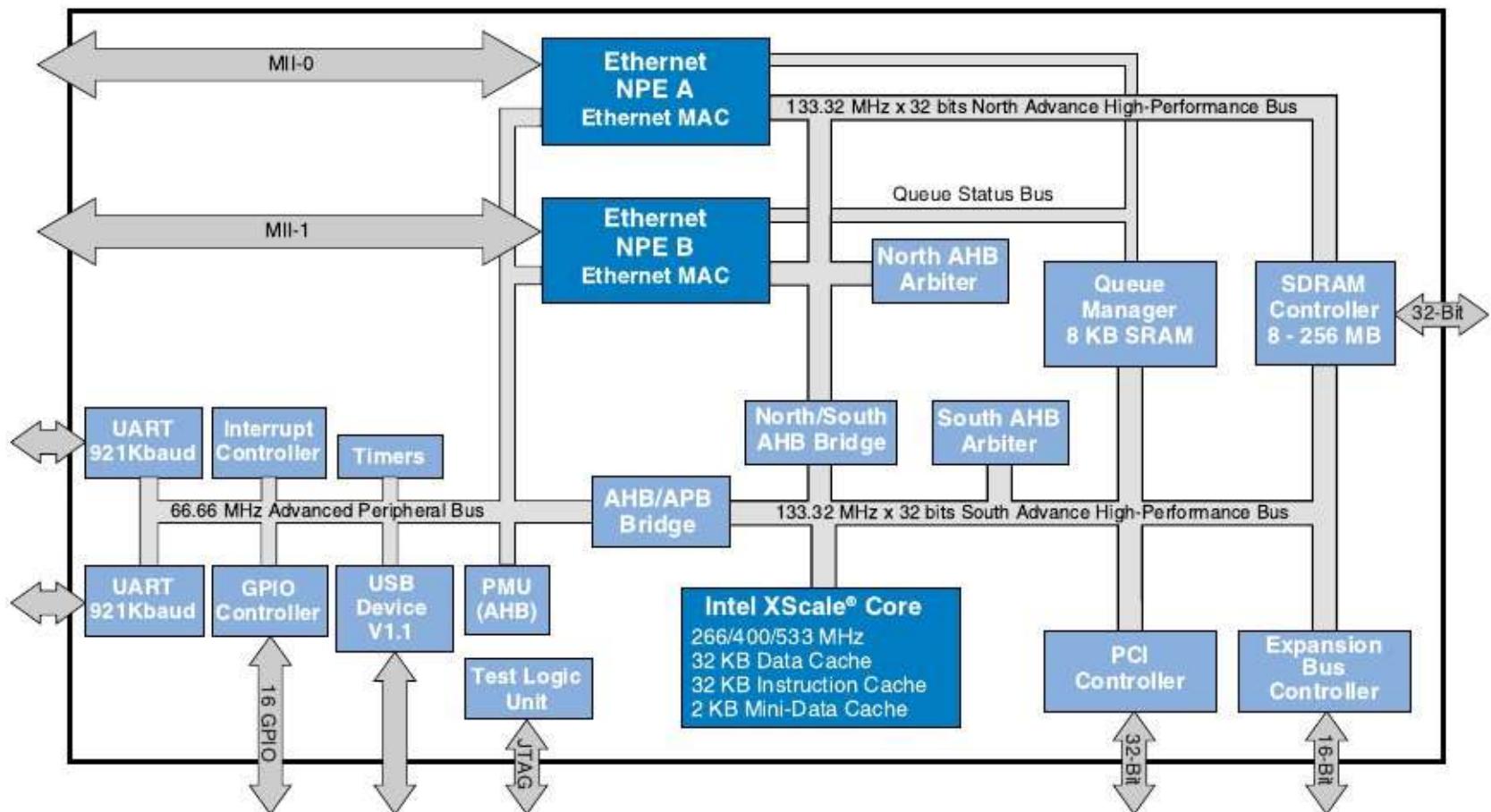
# Bus Architectures



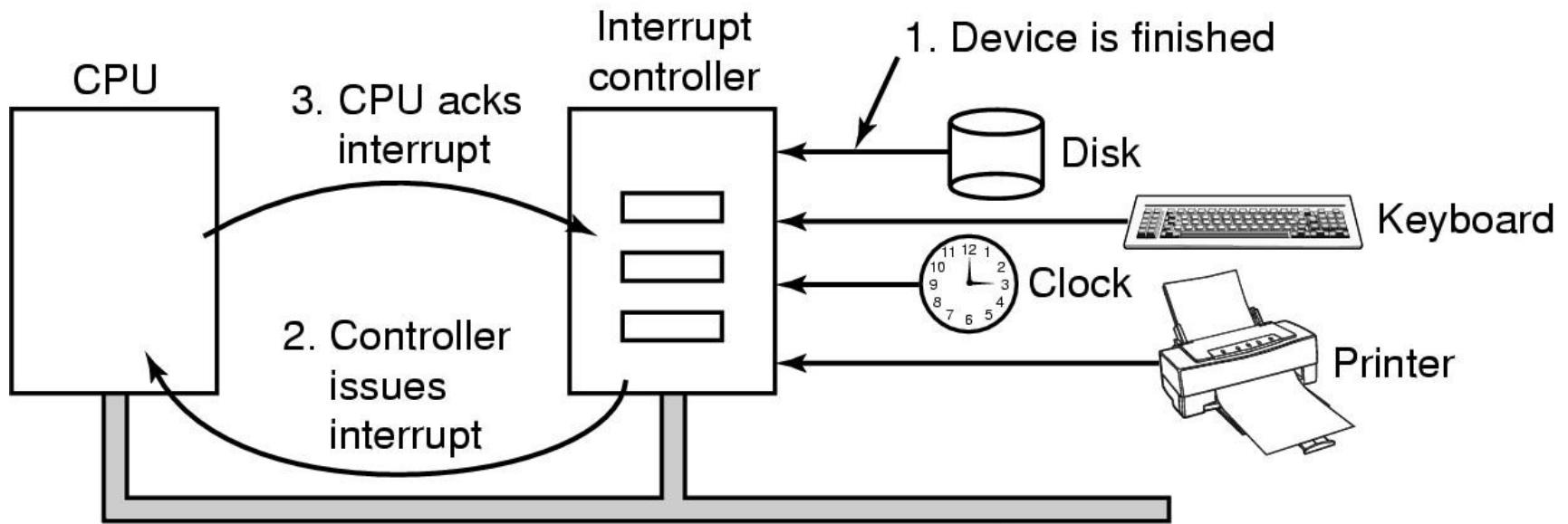
- (a) A single-bus architecture**
- (b) A dual-bus memory architecture**



# Intel IXP420



# Interrupts



- Devices connected to an *Interrupt Controller* via lines on an I/O bus (e.g. PCI)
- Interrupt Controller signals interrupt to CPU and is eventually acknowledged.
- Exact details are architecture specific.



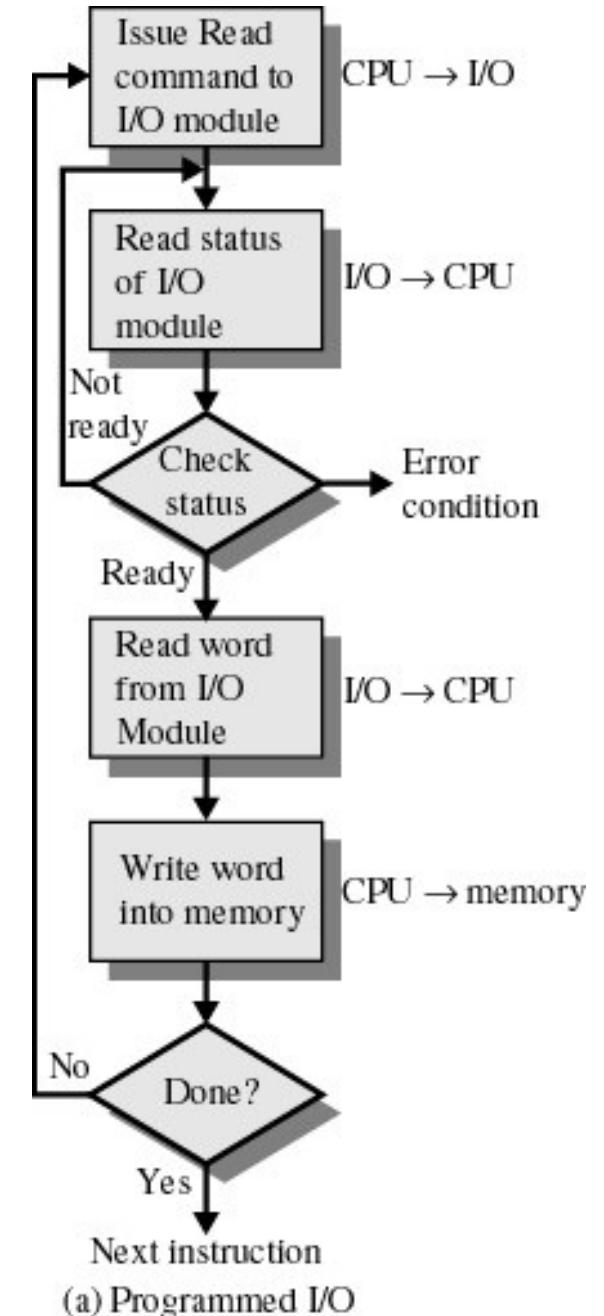
# I/O Interaction



THE UNIVERSITY OF  
NEW SOUTH WALES

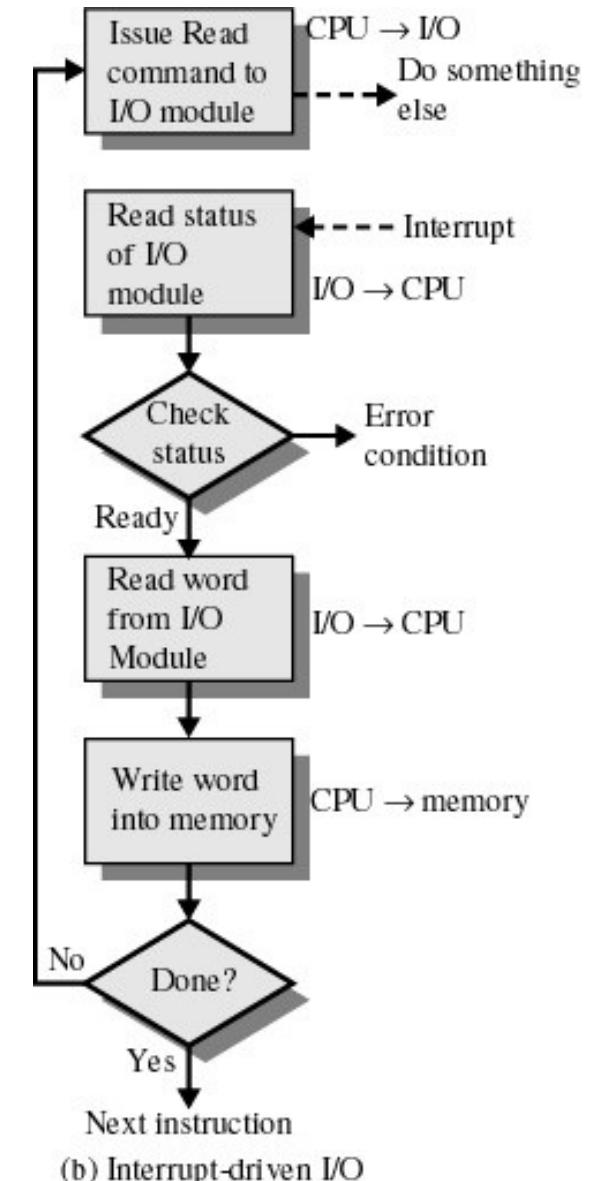
# Programmed I/O

- Also called *polling*, or *busy waiting*
- I/O module (controller) performs the action, not the processor
- Sets appropriate bits in the I/O status register
- No interrupts occur
- Processor checks status until operation is complete
  - Wastes CPU cycles



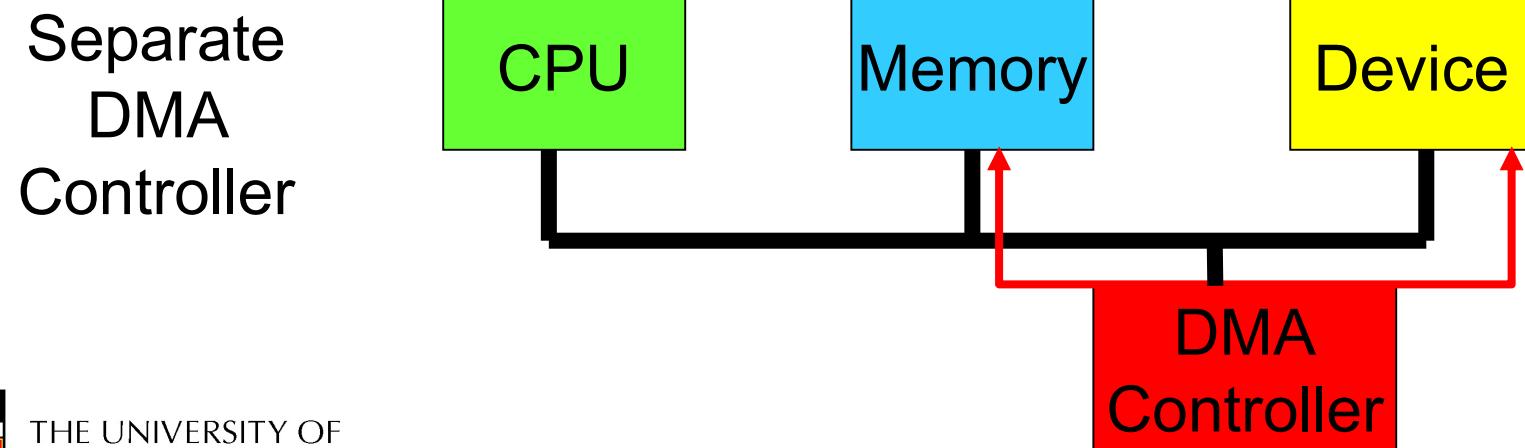
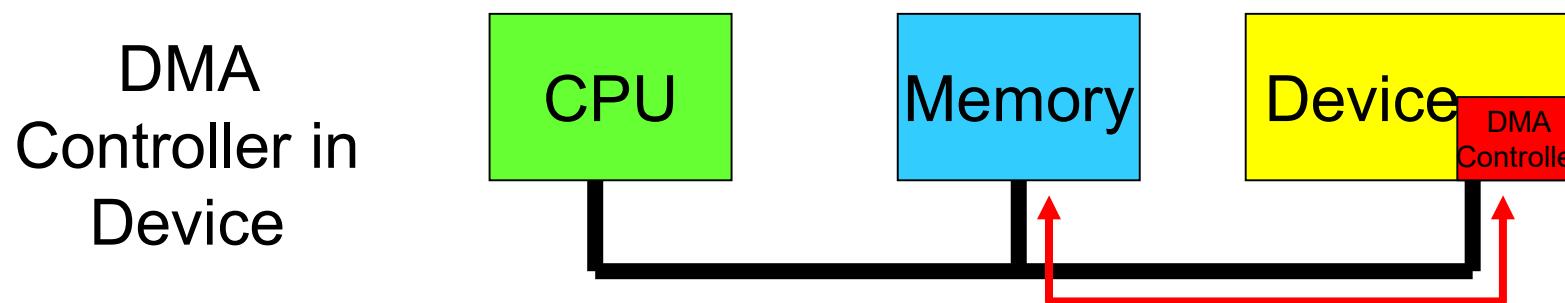
# Interrupt-Driven I/O

- Processor is interrupted when I/O module (controller) ready to exchange data
  - Processor is free to do other work
  - No needless waiting
  - Consumes a lot of processor time because every word read or written passes through the processor



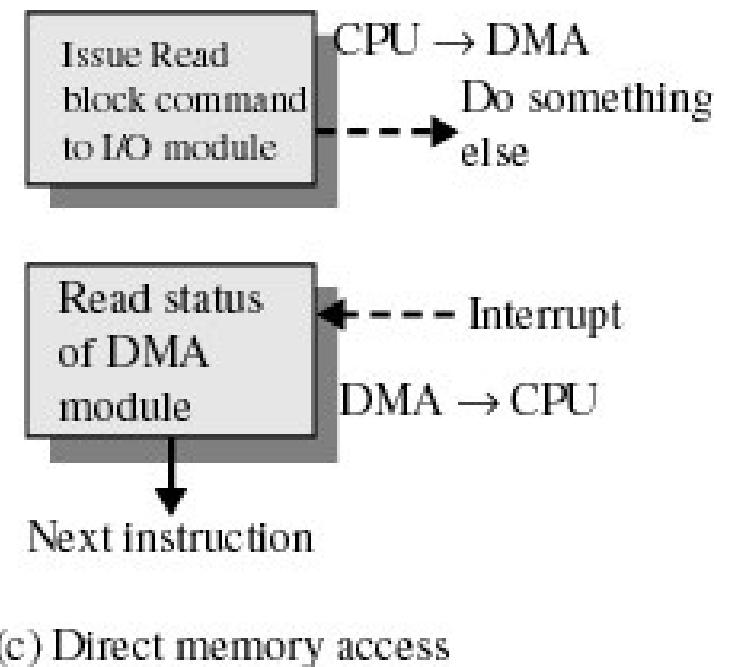
# Direct Memory Access

- Transfers data directly between Memory and Device
- CPU not needed for copying



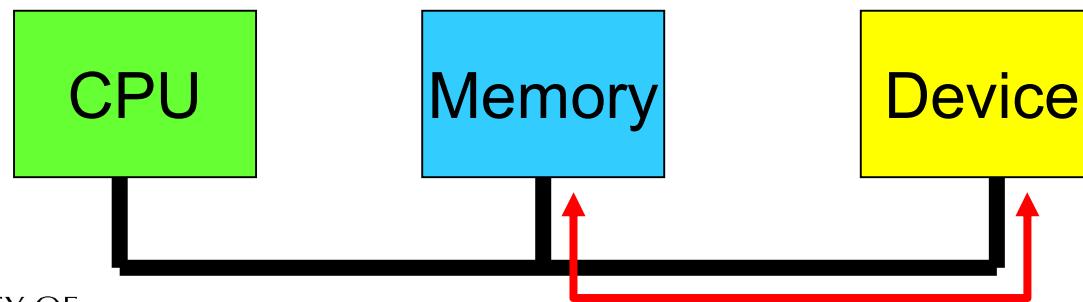
# Direct Memory Access

- Transfers a block of data directly to or from memory
- An interrupt is sent when the task is complete
- The processor is only involved at the beginning and end of the transfer

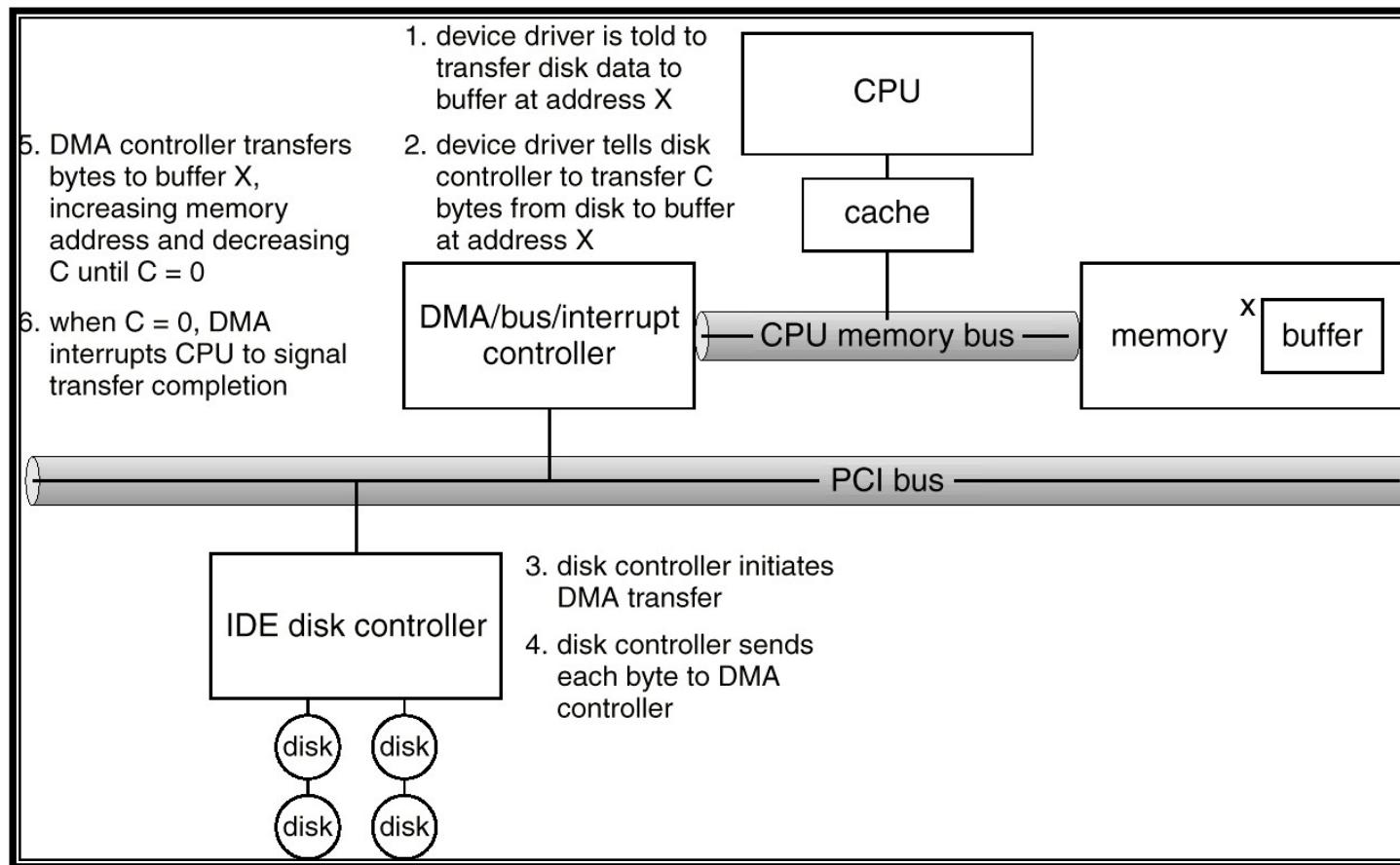


# DMA Considerations

- ✓ Reduces number of interrupts
  - Less (expensive) context switches or kernel entry-exits
- ✗ Requires contiguous regions (buffers)
  - Copying
  - Some hardware supports “Scatter-gather”
- Synchronous/Asynchronous
- Shared bus must be arbitrated (hardware)
  - CPU cache reduces (but not eliminates) CPU need for bus



# The Process to Perform DMA Transfer



# I/O Management Software

Chapter 5 – 5.3



THE UNIVERSITY OF  
NEW SOUTH WALES

# Learning Outcomes

- An understanding of the structure of I/O related software, including interrupt handlers.
- An appreciation of the issues surrounding long running interrupt handlers, blocking, and deferred interrupt handling.
- An understanding of I/O buffering and buffering's relationship to a producer-consumer problem.



# Operating System Design Issues

- **Efficiency**
  - Most I/O devices slow compared to main memory (and the CPU)
    - Use of multiprogramming allows for some processes to be waiting on I/O while another process executes
    - Often I/O still cannot keep up with processor speed
    - Swapping may be used to bring in additional Ready processes
      - More I/O operations
- **Optimise I/O efficiency – especially Disk & Network I/O**

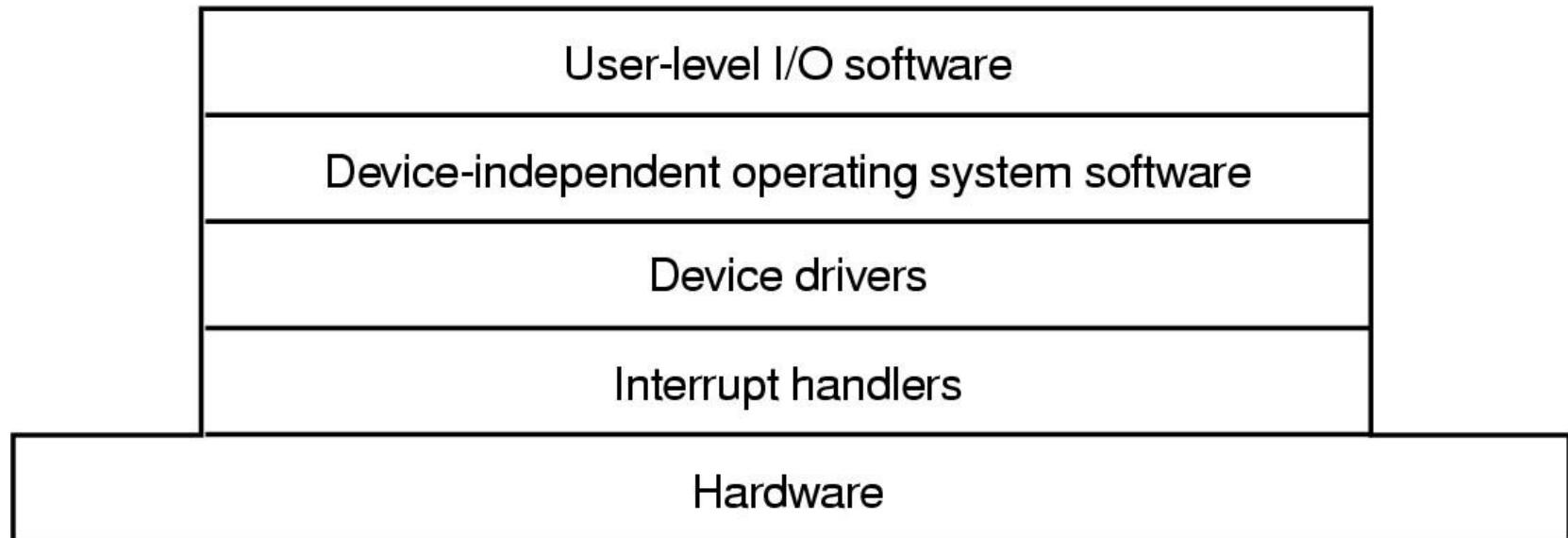


# Operating System Design Issues

- **The quest for generality/uniformity:**
  - Ideally, handle all I/O devices in the same way
    - Both in the OS and in user applications
  - Problem:
    - Diversity of I/O devices
    - Especially, different access methods (random access versus stream based) as well as vastly different data rates.
    - Generality often compromises efficiency!
  - Hide most of the details of device I/O in lower-level routines so that processes and upper levels see devices in general terms such as read, write, open, close.



# I/O Software Layers



## Layers of the I/O Software System



# Interrupt Handlers

- **Interrupt handlers**
  - Can execute at (almost) any time
    - Raise (complex) concurrency issues in the kernel
    - Can propagate to userspace (signals, upcalls), causing similar issues
    - Generally structured so I/O operations block until interrupts notify them of completion
      - kern/dev/lamebus/lhd.c



# Interrupt Handler Example

```
static int
lhd_io(struct device *d,
       struct uio *uio)
{
...
/* Loop over all the sectors
 * we were asked to do. */
for (i=0; i<len; i++) {
    /* Wait until nobody else
     * is using the device. */
    P(lh->lhd_clear);
    ...
    /* Tell it what sector we want... */
    lhd_wreg(lh, LHD_REG_SECT, sector+i);
    /* and start the operation. */
    lhd_wreg(lh, LHD_REG_STAT, statval);
    /* Now wait until the interrupt
     * handler tells us we're done. */
    P(lh->lhd_done);

    /* Get the result value
     * saved by the interrupt handler. */
    result = lh->lhd_result;
}
```

INT

SLEEP

```
lhd_iodone(struct lhd_softc *lh, int err)
{
    lh->lhd_result = err;
    V(lh->lhd_done);
}

void
lhd_irq(void *vlh)
{
...
    val = lhd_rdreg(lh, LHD_REG_STAT);

    switch (val & LHD_STATEMASK) {
        case LHD_IDLE:
        case LHD_WORKING:
            break;
        case LHD_OK:
        case LHD_INVSECT:
        case LHD_MEDIA:
            lhd_wreg(lh, LHD_REG_STAT, 0);
            lhd_iodone(lh,
                       lhd_code_to_errno(lh, val));
            break;
    }
}
```



# Interrupt Handler Steps

- **Save Registers** not already saved by hardware interrupt mechanism
- (Optionally) **set up context** for interrupt service procedure
  - Typically, handler runs in the context of the currently running process
    - No expensive context switch
- **Set up stack** for interrupt service procedure
  - Handler usually runs on the kernel stack of current process
  - Or “nests” if already in kernel mode running on kernel stack
- **Ack/Mask interrupt controller**, re-enable other interrupts
  - Implies potential for interrupt nesting.



# Interrupt Handler Steps

- **Run interrupt service procedure**
  - Acknowledges interrupt at device level
  - Figures out what caused the interrupt
    - Received a network packet, disk read finished, UART transmit queue empty
  - If needed, it signals blocked device driver
- **In some cases, will have woken up a higher priority blocked thread**
  - Choose newly woken thread to schedule next.
  - Set up MMU context for process to run next
  - What if we are nested?
- **Load new/original process' registers**
- **Re-enable interrupt;** Start running the new process

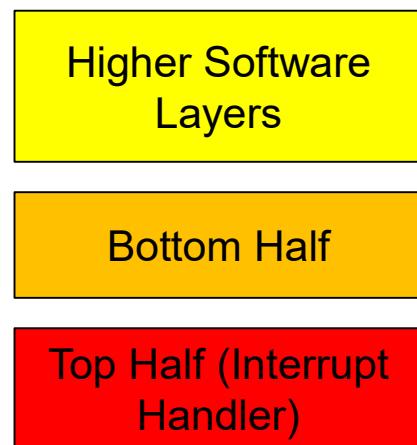


# Sleeping in Interrupts

- An interrupt generally has no **context** (runs on current kernel stack)
  - Unfair to sleep on interrupted process (deadlock possible)
  - Where to get context for long running operation?
  - What goes into the ready queue?
- What to do?
  - Top and Bottom Half
  - Linux implements with tasklets and workqueues
  - Generically, in-kernel thread(s) handle long running kernel operations.



# Top/Half Bottom Half

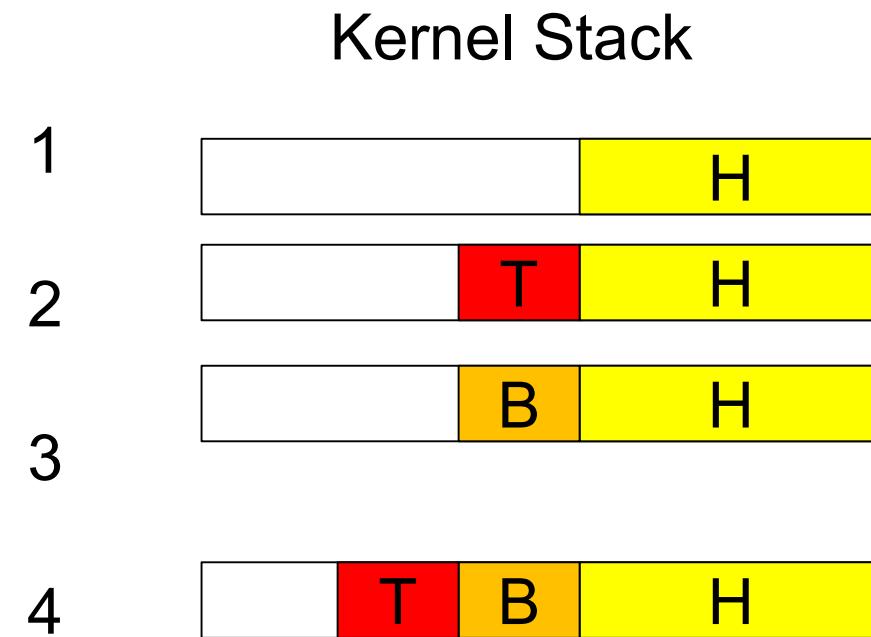


- Top Half
  - Interrupt handler
  - remains short
- Bottom half
  - Is preemptable by top half (interrupts)
  - performs deferred work (e.g. IP stack processing)
  - Is checked prior to every kernel exit
  - signals blocked processes/threads to continue
- Enables low interrupt latency
- Bottom half can't block



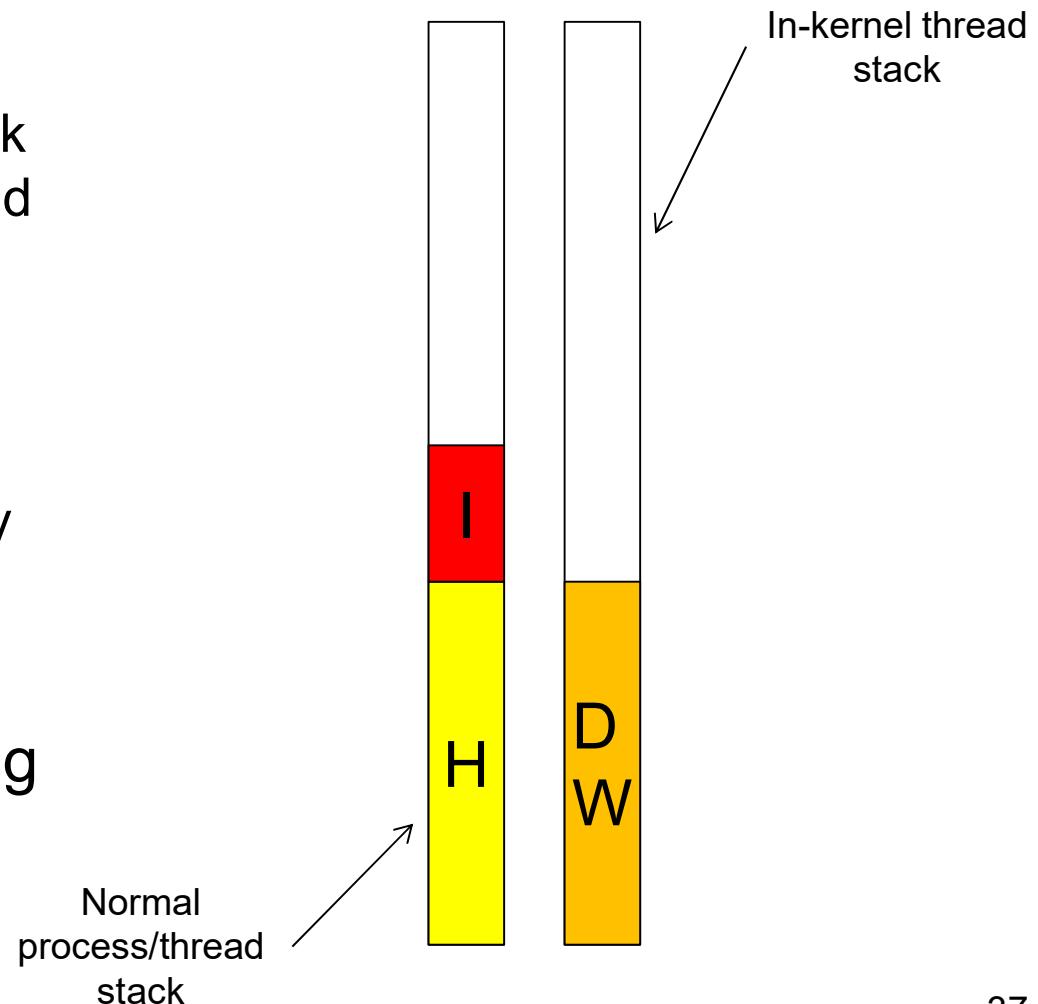
# Stack Usage

1. Higher-level software
2. Interrupt processing (interrupts disabled)
3. Deferred processing (interrupt re-enabled)
4. Interrupt while in bottom half



# Deferring Work on In-kernel Threads

- Interrupt
  - handler defers work onto in-kernel thread
- In-kernel thread handles deferred work (DW)
  - Scheduled normally
  - Can block
- Both low interrupt latency and blocking operations

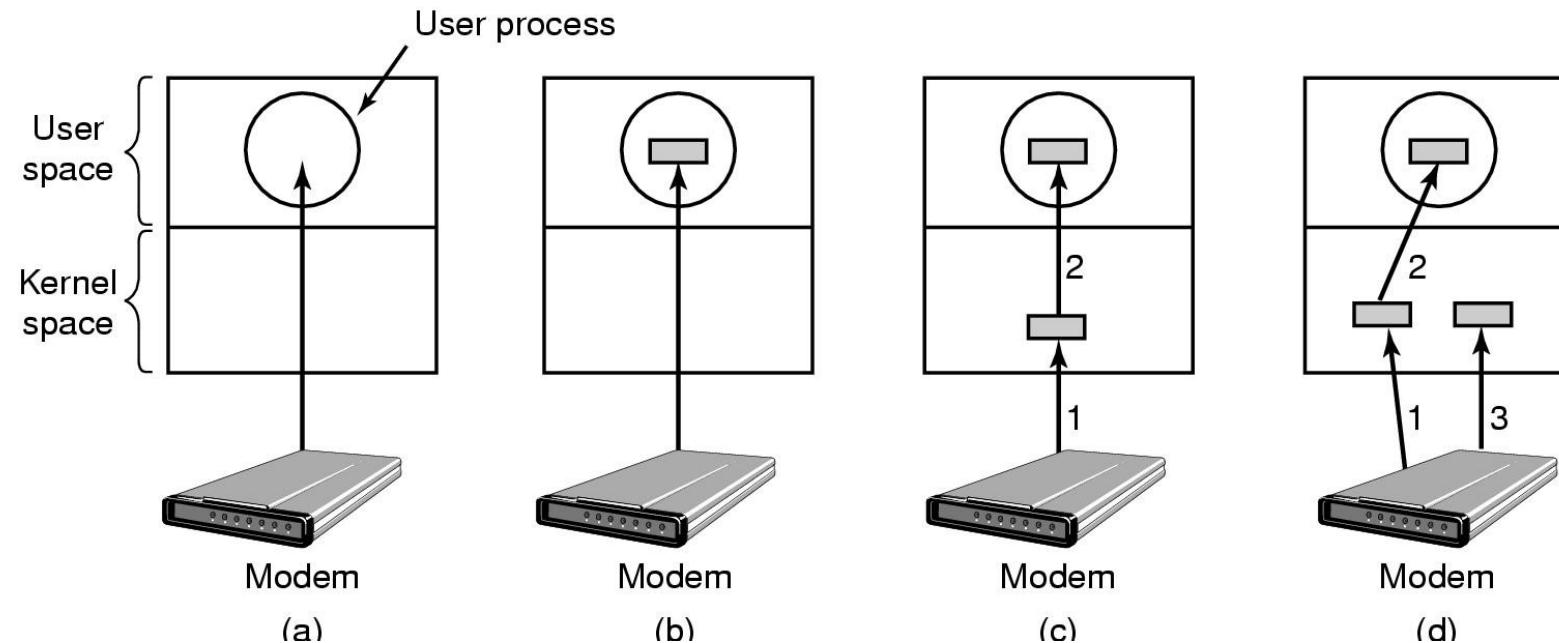


# Buffering



THE UNIVERSITY OF  
NEW SOUTH WALES

# Device-Independent I/O Software



- (a) Unbuffered input
- (b) Buffering in user space
- (c) *Single buffering* in the kernel followed by copying to user space
- (d) Double buffering in the kernel



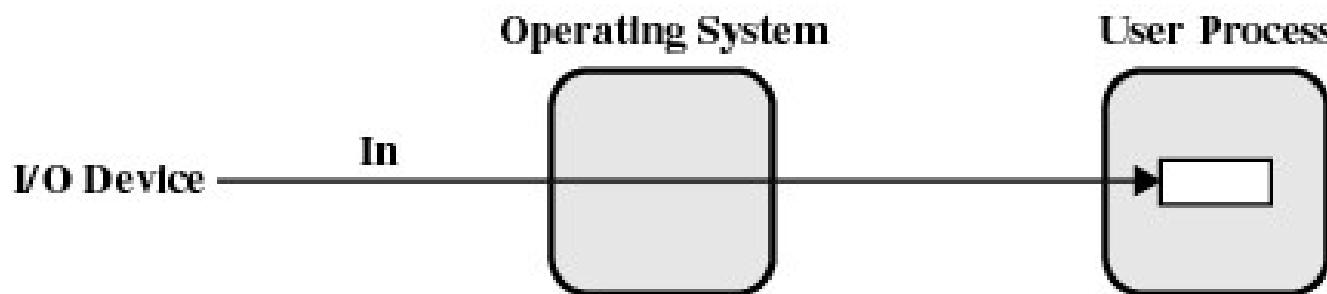
# No Buffering

- Process must read/write a device a byte/word at a time
  - Each individual system call adds significant overhead
  - Process must wait until each I/O is complete
    - Blocking/interrupt/waking adds to overhead.
    - Many short runs of a process is inefficient (poor CPU cache temporal locality)



# User-level Buffering

- Process specifies a memory *buffer* that incoming data is placed in until it fills
  - Filling can be done by interrupt service routine
  - Only a single system call, and block/wakeup per data buffer
    - Much more efficient



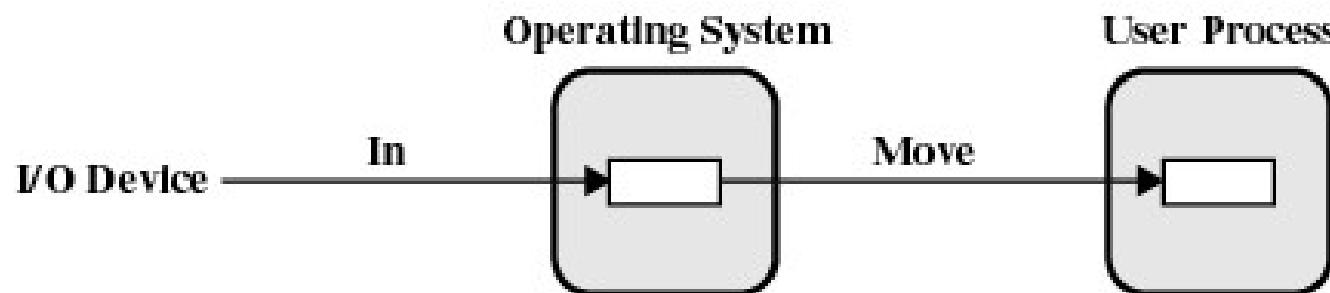
# User-level Buffering

- Issues
  - What happens if buffer is paged out to disk
    - Could lose data while unavailable buffer is paged in
    - Could lock buffer in memory (needed for DMA), however many processes doing I/O reduce RAM available for paging. Can cause deadlock as RAM is limited resource
  - Consider write case
    - When is buffer available for re-use?
      - Either process must block until potential slow device drains buffer
      - or deal with asynchronous signals indicating buffer drained



# Single Buffer

- Operating system assigns a buffer in kernel's memory for an I/O request
- In a stream-oriented scenario
  - Used a line at time
  - User input from a terminal is one line at a time with carriage return signaling the end of the line
  - Output to the terminal is one line at a time

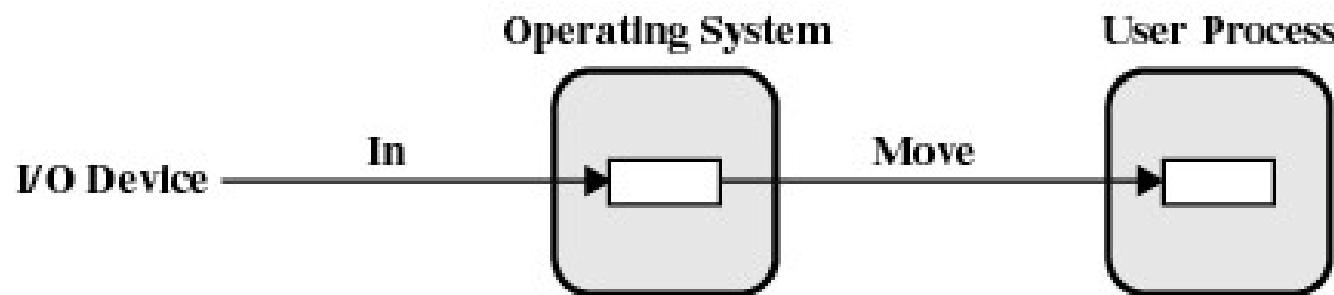


(b) Single buffering



# Single Buffer

- Block-oriented
  - Input transfers made to buffer
  - Block copied to user space when needed
  - Another block is written into the buffer
    - Read ahead

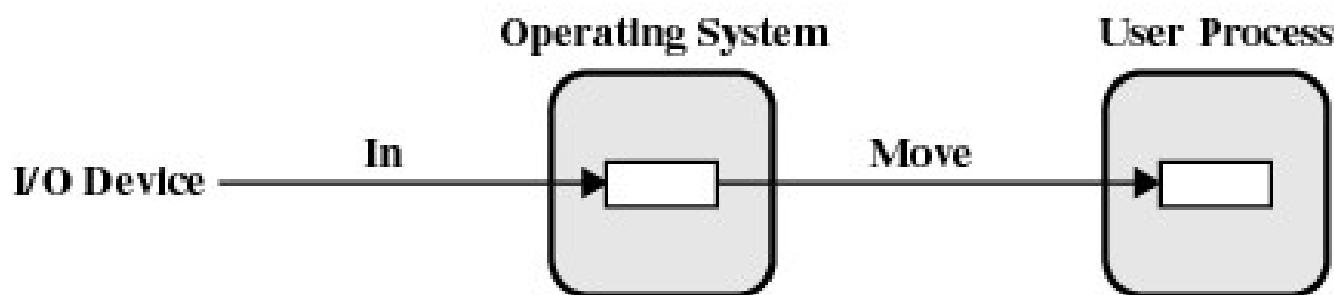


(b) Single buffering



# Single Buffer

- User process can process one block of data while next block is read in
- Swapping can occur since input is taking place in system memory, not user memory
- Operating system keeps track of assignment of system buffers to user processes



(b) Single buffering



# Single Buffer Speed Up

- Assume
  - $T$  is transfer time for a block from device
  - $C$  is computation time to process incoming block
  - $M$  is time to copy kernel buffer to user buffer
- Computation and transfer can be done in parallel
- Speed up with buffering

$$\frac{T + C}{\max(T, C) + M}$$

No Buffering Cost

Single Buffering Cost



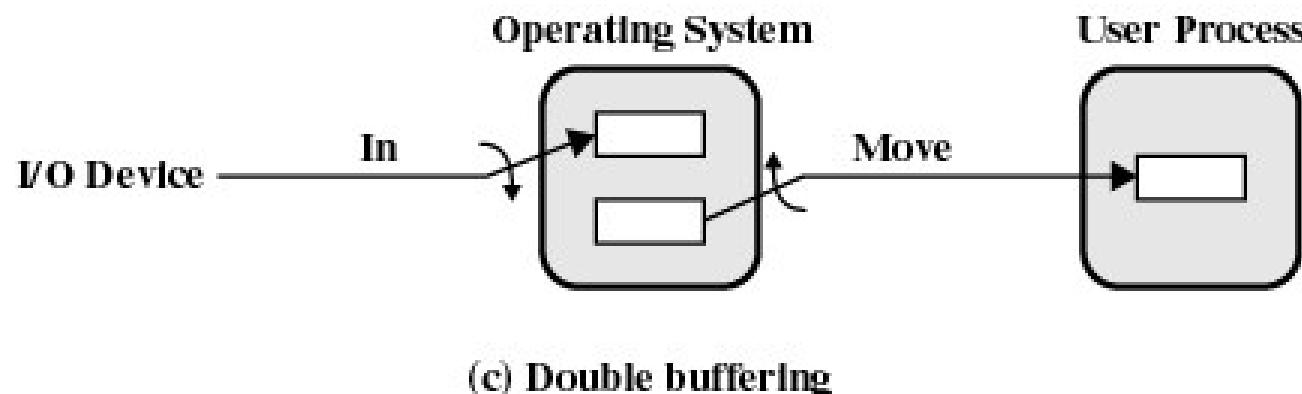
# Single Buffer

- What happens if kernel buffer is full
    - the user buffer is swapped out, or
    - The application is slow to process previous buffer
- and more data is received???
- => We start to lose characters or drop network packets



# Double Buffer

- Use two system buffers instead of one
- A process can transfer data to or from one buffer while the operating system empties or fills the other buffer



# Double Buffer Speed Up

- Computation and Memory copy can be done in parallel with transfer
- Speed up with double buffering

$$\frac{T + C}{\max(T, C + M)}$$

- Usually  $M$  is much less than  $T$  giving a favourable result



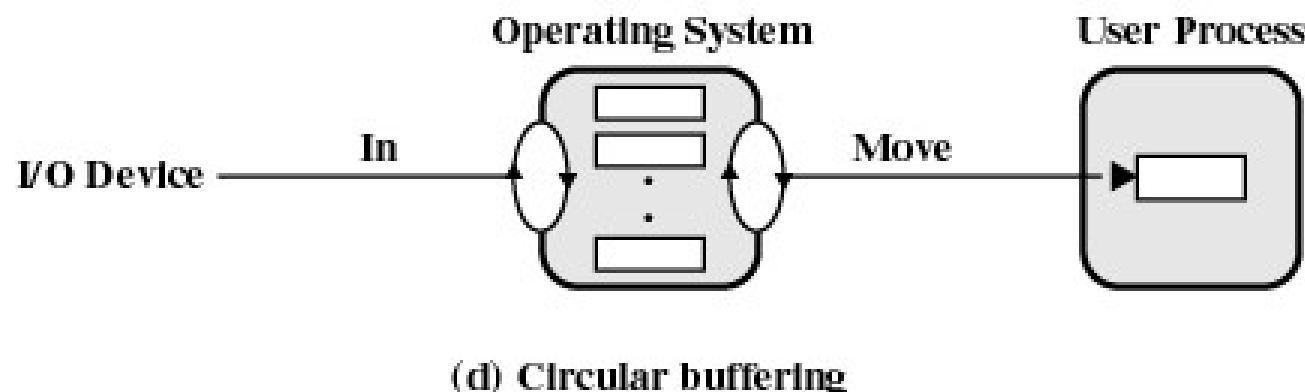
# Double Buffer

- May be insufficient for really bursty traffic
  - Lots of application writes between long periods of computation
  - Long periods of application computation while receiving data
  - Might want to read-ahead more than a single block for disk



# Circular Buffer

- More than two buffers are used
- Each individual buffer is one unit in a circular buffer
- Used when I/O operation must keep up with process



# Important Note

- Notice that buffering, double buffering, and circular buffering are all

# Bounded-Buffer Producer-Consumer Problems



# Is Buffering Always Good?

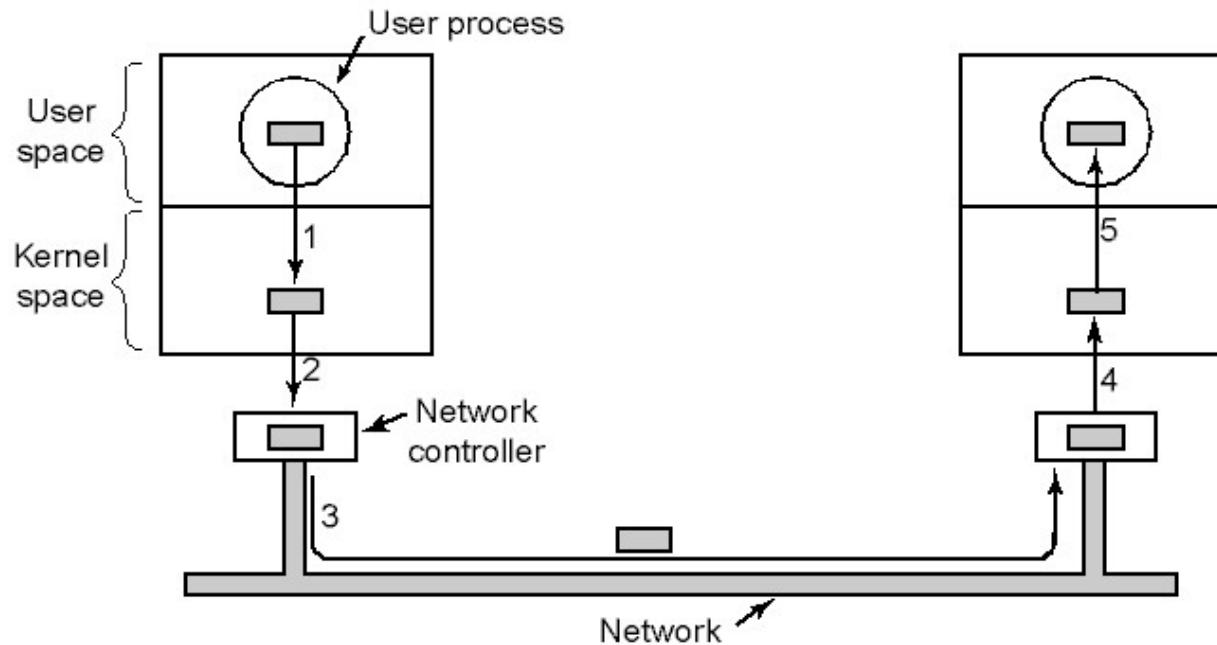
$$\frac{T + C}{\max(T, C) + M} \quad \frac{T + C}{\max(T, C + M)}$$

Single    Double

- Can  $M$  be similar or greater than  $C$  or  $T$ ?



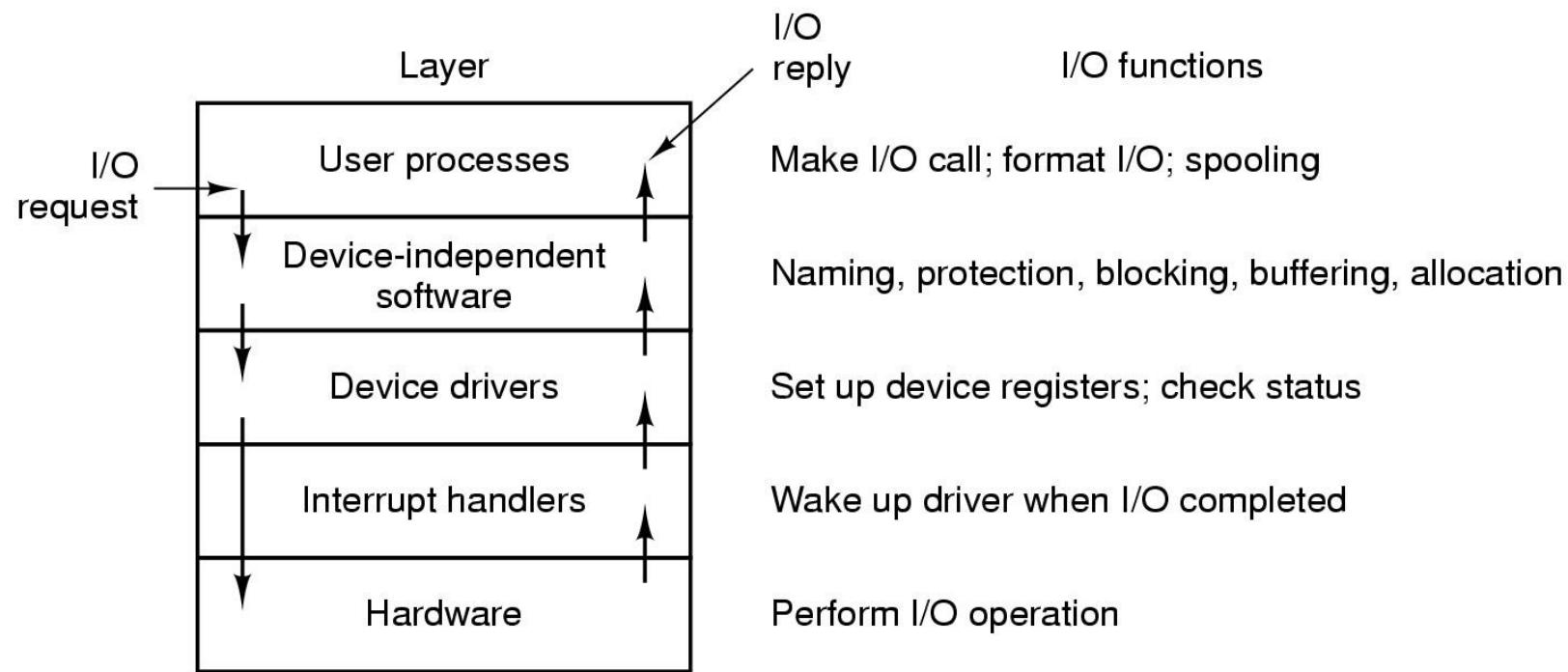
# Buffering in Fast Networks



- Networking may involve many copies
- Copying reduces performance
  - Especially if copy costs are similar to or greater than computation or transfer costs
- Super-fast networks put significant effort into achieving zero-copy
- Buffering also increases latency



# I/O Software Summary



Layers of the I/O system and the main functions of each layer

