We acknowledge and pay our respects to the Kaurna people,
the traditional custodians whose ancestral lands we gather on.

We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

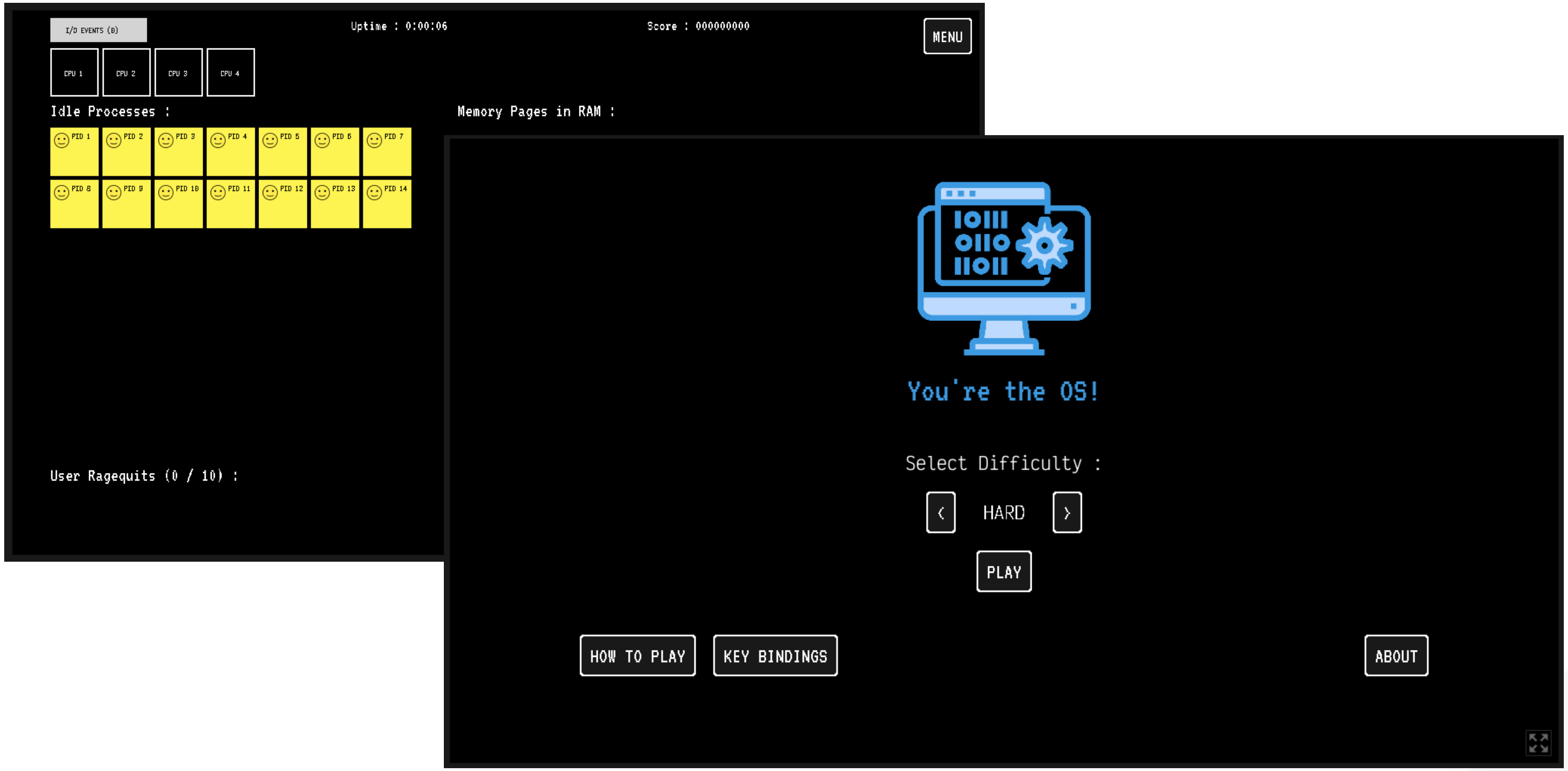
COMP SCI 3004 Operating Systems

Week 5 – Recap ”Virtualisation”

make history.

THE UNIVERSITY of ADELAIDE

Thanks to **Chloe Walsh!**



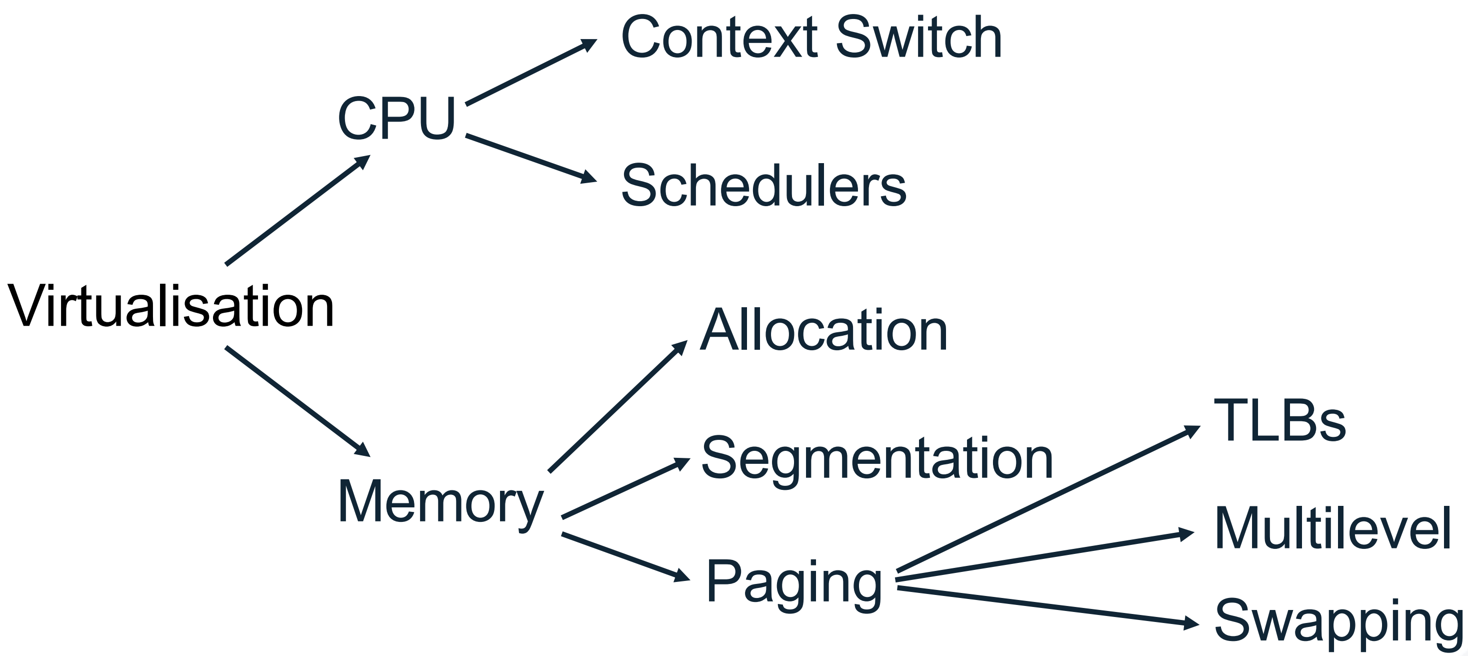
https://drfreckles42.itch.io/youre-the-os

Introduction

- Discussion of online quiz
- **Recap of ”Easy piece”: Virtualisation**
- **Start of next chapter:**
  - **Concurrency**
  - Today, Chapter 26: *Concurrency and Threads*

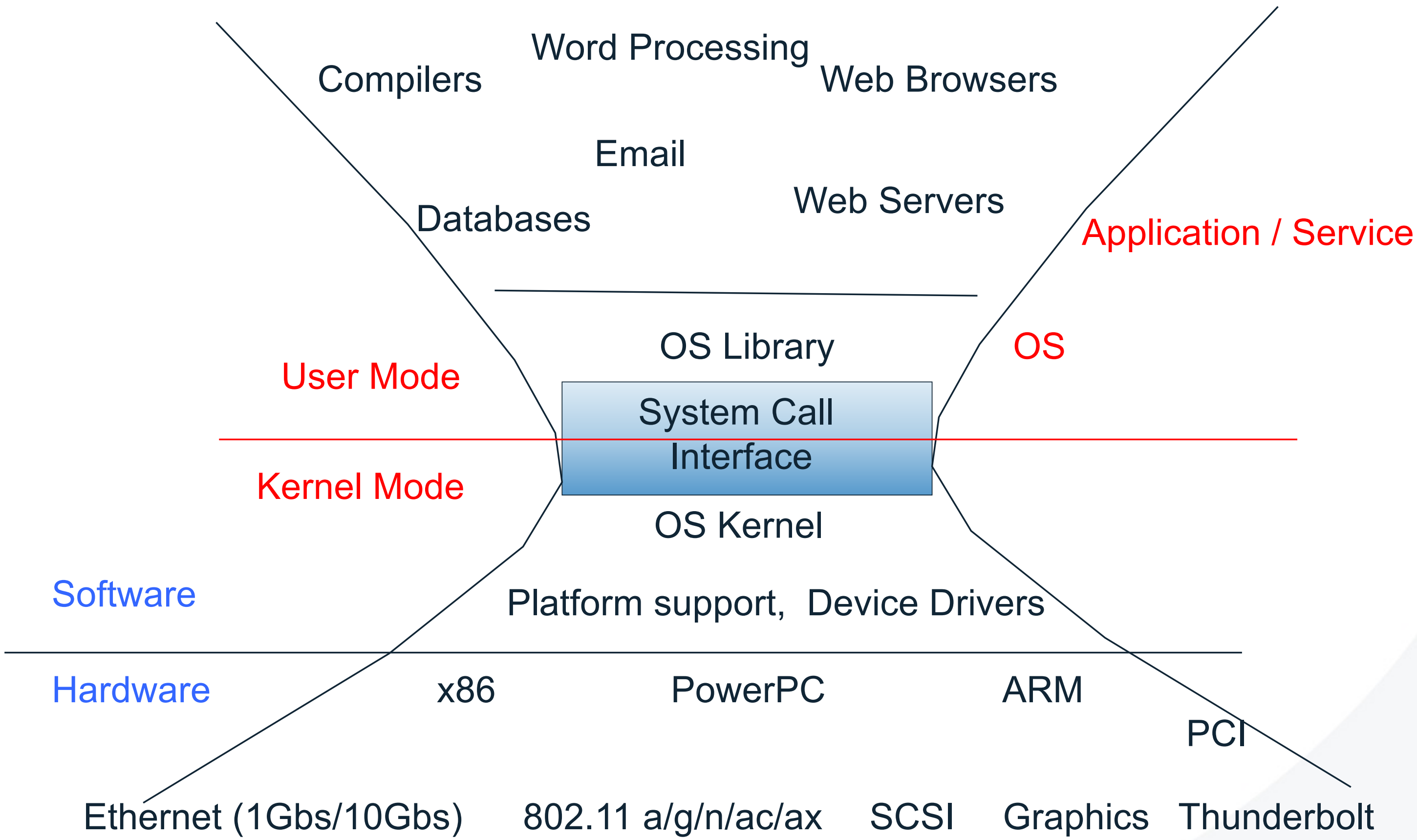
THE UNIVERSITY of ADELAIDE

Recall: Easy Piece 1 – Virtualisation



THE UNIVERSITY of ADELAIDE

Recall: A Narrow Waist



Recall: Context Switching

OS runs dispatch loop

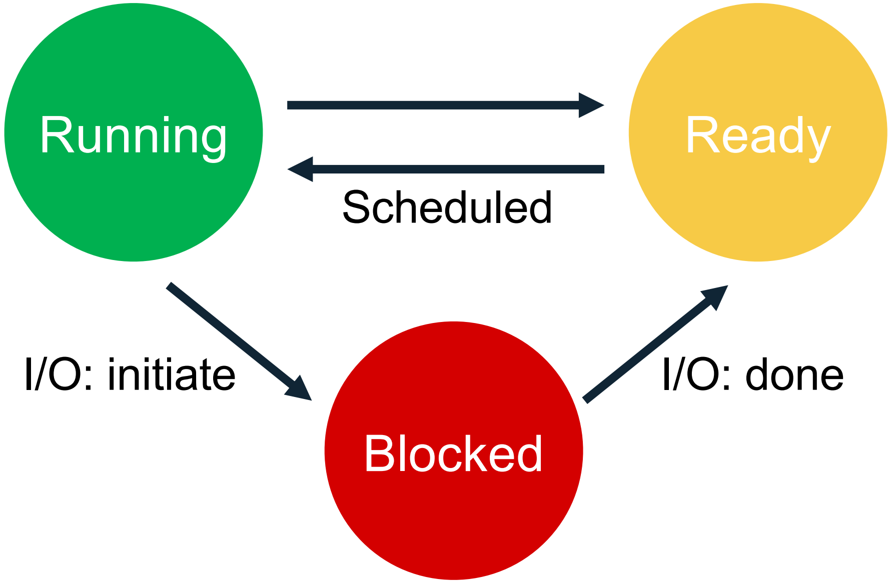
```
while (1) {
        run process A for some time-slice
        stop process A and save its context
        load context of another process B
}
```

Context-switch

Stored information in PCB (Process Control Block). E.g., PID, Process state, Execution state (all registers, PC, stack ptr), Scheduling priority, Credentials (which resources can be accessed, owner), Pointers to other allocated resources (e.g., open files).

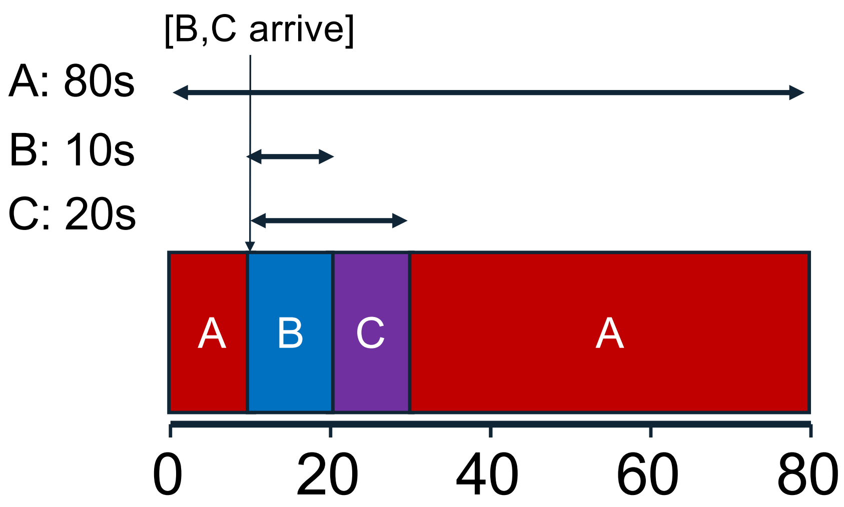
THE UNIVERSITY of ADELAIDE

Recall: State Transitions



How to schedule jobs?

Recall: PREEMPTIVE Scheduling



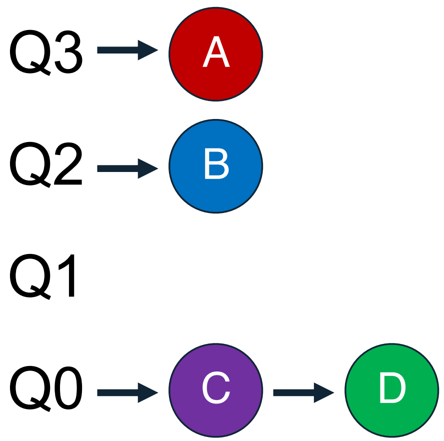
Schedulers:

- FIFO
- SJF
- STCF
- RR
- MLFQ
- Lottery
- CFS

Metrics:

- turnaround_time
- response_time

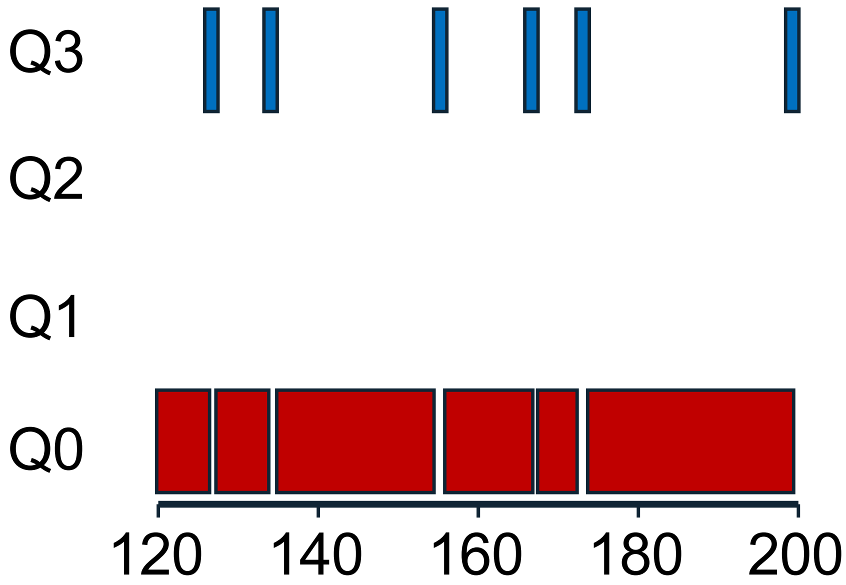
Recall: MLFQ (Multi-Level Feedback Queue)



- **Rule 1**: If priority(A) > Priority(B), A runs
- **Rule 2**: If priority(A) == Priority(B), A & B run in RR using the time slice (quantum length) of the given queue.
- **Rule 3**: Processes start at top priority (highest queue).
- **Rule 4**: If job uses whole slice, demote process (longer time slices at lower priorities)
- **Rule 5**: After some time period S, move all the jobs in the system to the topmost queue.

THE UNIVERSITY of ADELAIDE

Recall: MLFQ: Prevent Starvation



- **Problem: Low priority job may never get scheduled**
  - Periodically boost priority of all jobs (or all jobs that haven’t been scheduled)

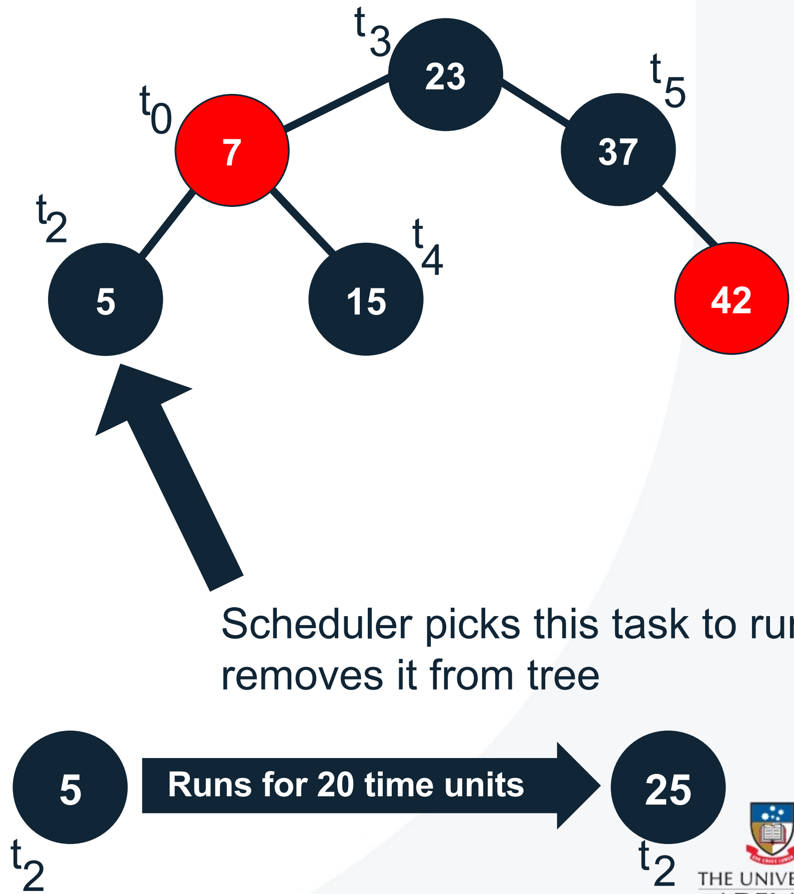
Recall: Linux's "Completely Fair Scheduler"

For now, make these simplifying assumptions:

- All tasks have the same priority
- There are always T tasks ready to run at any moment

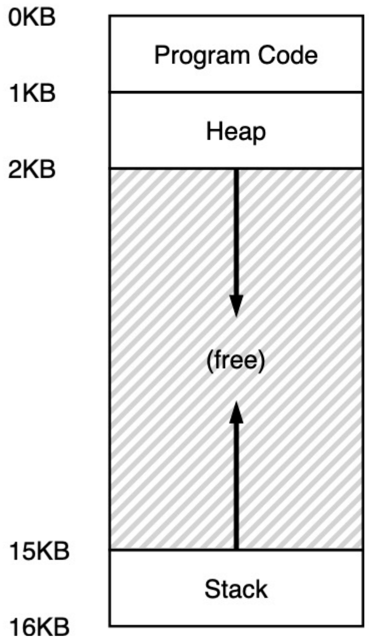
Basic idea: each task gets 1/T of the CPU's resources

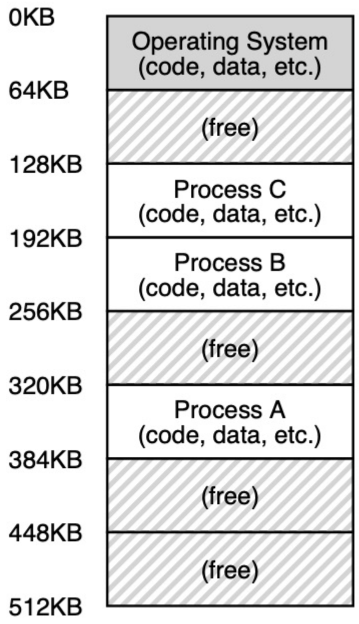
- CFS tries to model an "ideal CPU" that runs each task simultaneously, but at 1/T the CPU's clock speed
- A real CPU can only run a single task at once, so a task will get "ahead" or "behind" of its 1/T allotment
- CFS tracks how long each task has actually run; during a scheduling decision (e.g., timer interrupt), picks the task with lowest runtime so far.



Recall: Abstraction: Address Space

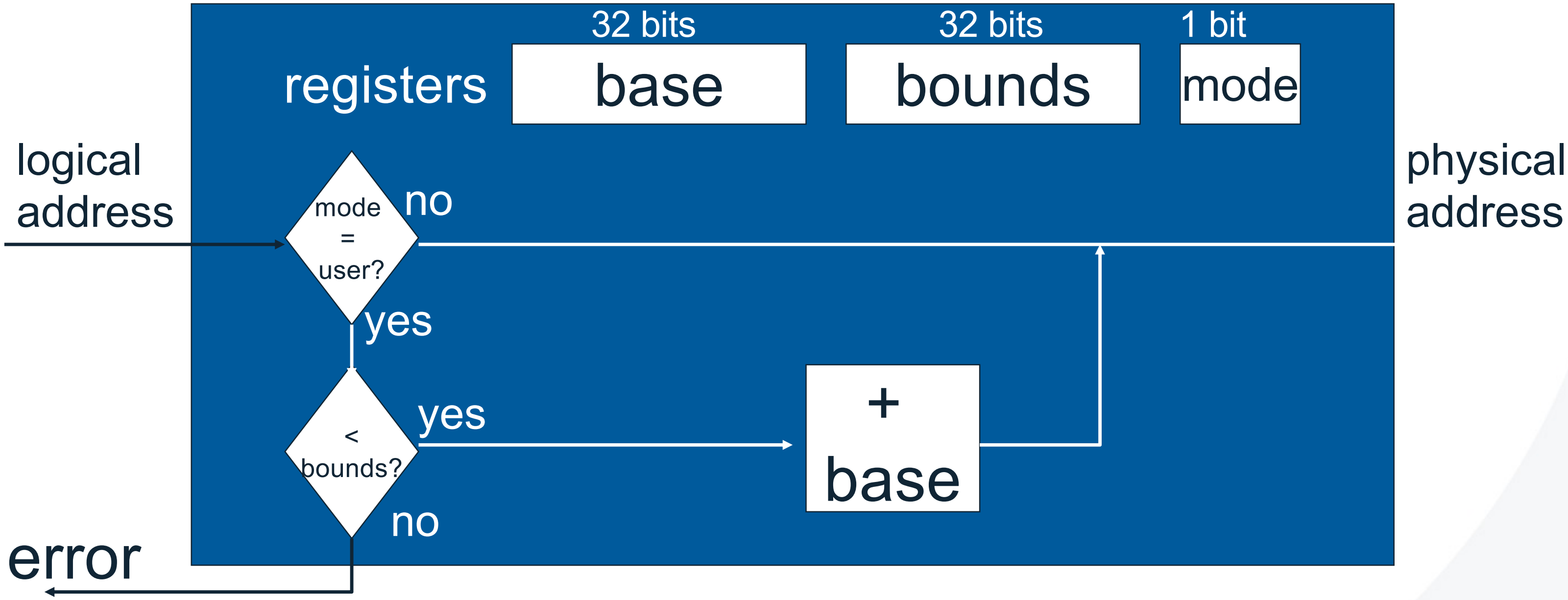
- Address space: Each process has set of addresses that map to bytes
- Problem: Address space has static and dynamic components





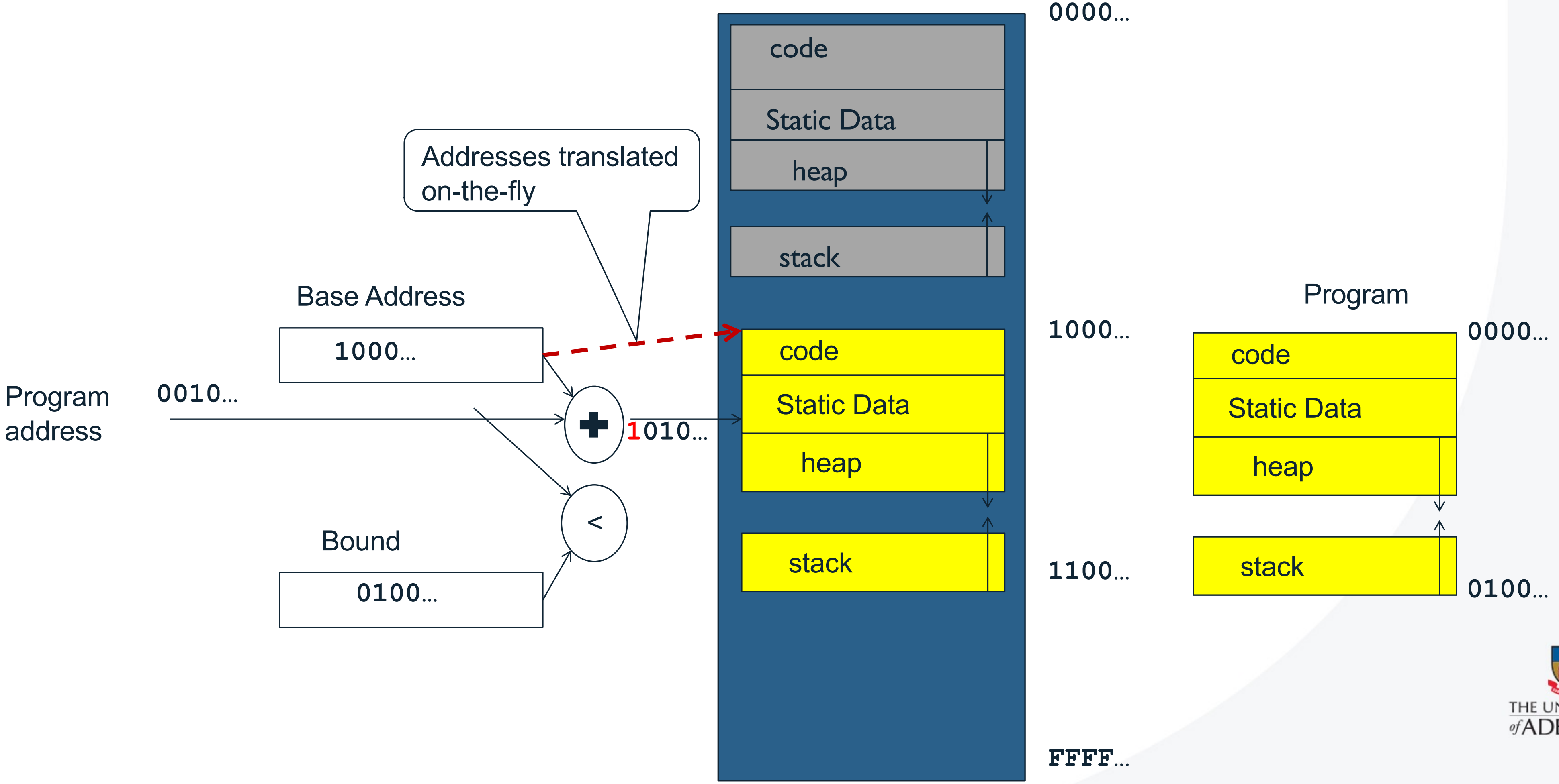
THE UNIVERSITY of ADELAIDE

Recall: Base+Bounds



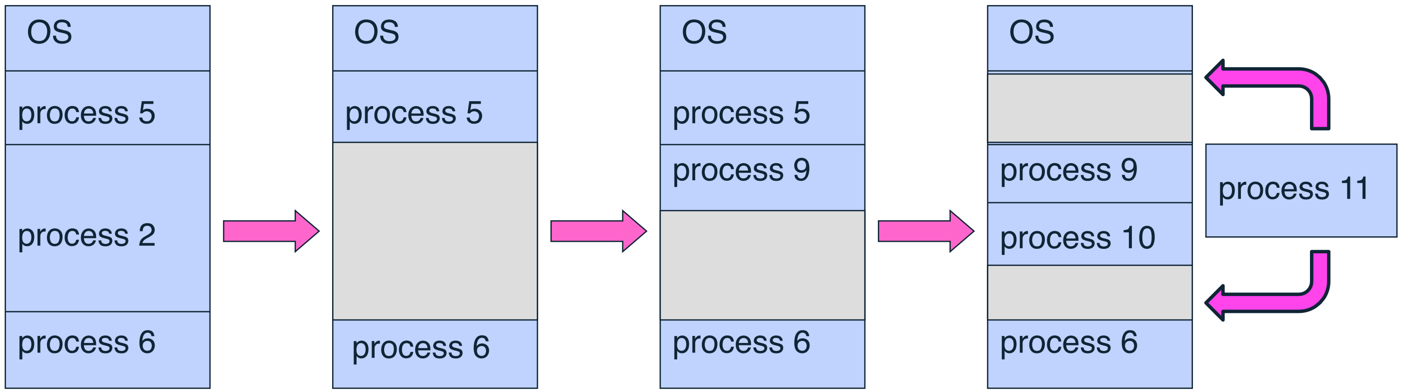
THE UNIVERSITY of ADELAIDE

Recall: Base+Bounds



THE UNIVERSITY of ADELAIDE

Issues with Simple B&B Method



Fragmentation problem over time

Not every process is same size ⇒ memory becomes fragmented over time

Missing support for sparse address space

Would like to have multiple chunks/program (Code/Data, Stack, Heap, etc)

Hard to do inter-process sharing

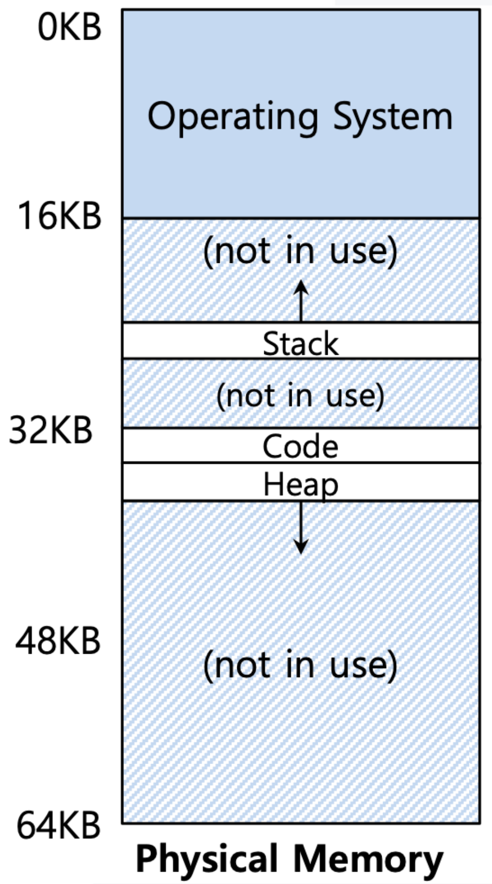
Want to share code segments when possible

Want to share memory between processes

Helped by providing multiple segments per process

Segmented Addressing

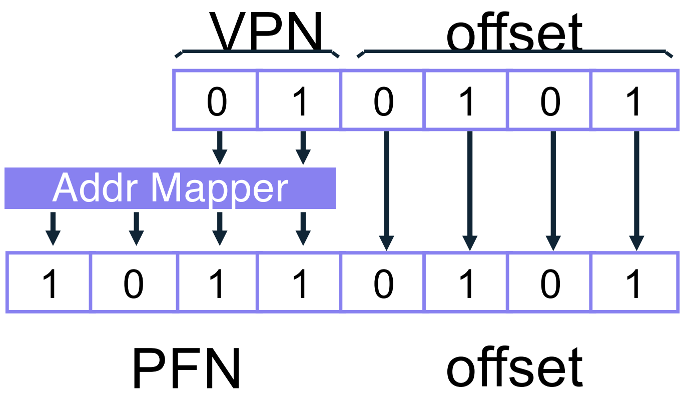
- **Process now specifies segment and offset within segment**
- **How does process designate a particular segment?**
  - Use part of logical address
    - Top bits of logical address select segment
    - Low bits of logical address select offset within segment
- **What if small address space, not enough bits?**
  - Implicitly by type of memory reference
  - Special registers

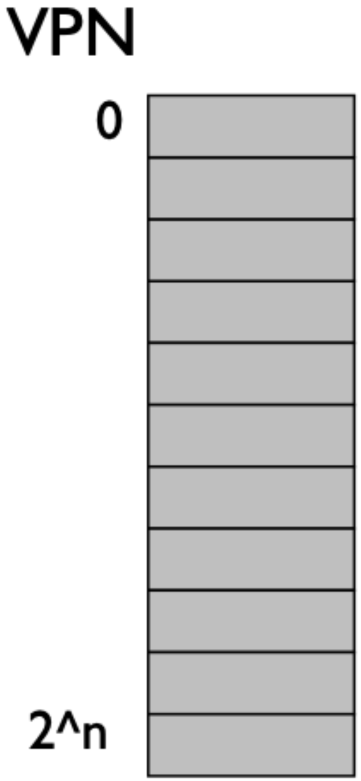


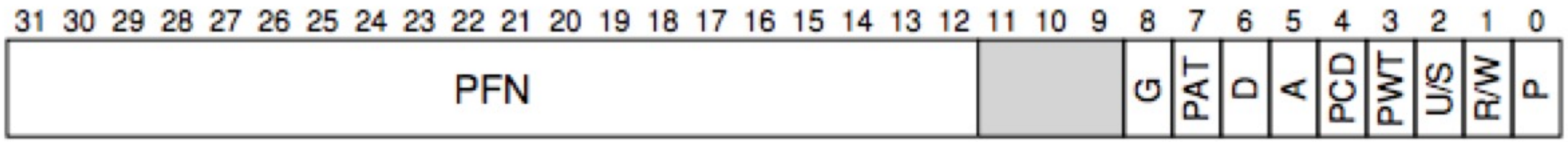
Recall: Page Tables

Big array: pagetable

Number of bits in virtual address format does not need to equal number of bits in physical address format





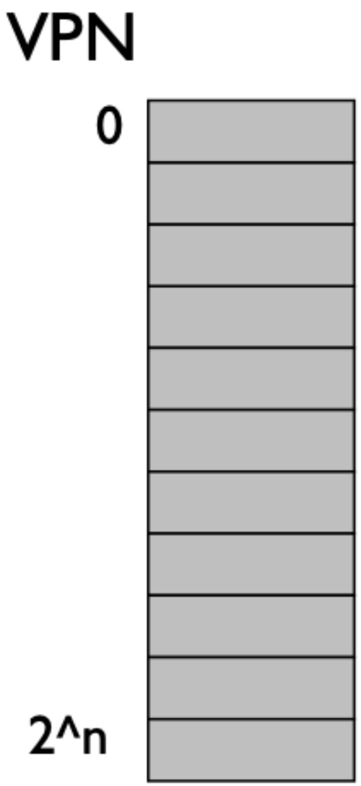


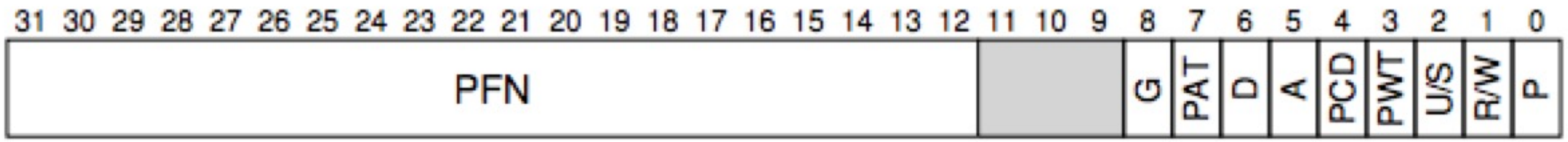
THE UNIVERSITY of ADELAIDE

Recall: Page Tables

For each mem reference:

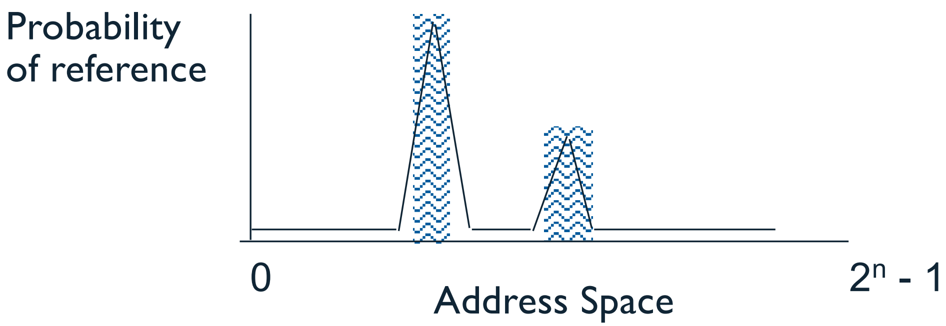
(cheap) 1. extract VPN (virt page num) from VA (virt addr)
(cheap) 2. calculate addr of PTE (page table entry)
(expensive) 3. read PTE from memory
(cheap) 4. extract PFN (page frame num)
(cheap) 5. build PA (phys addr)
(expensive) 6. read contents of PA from memory into register



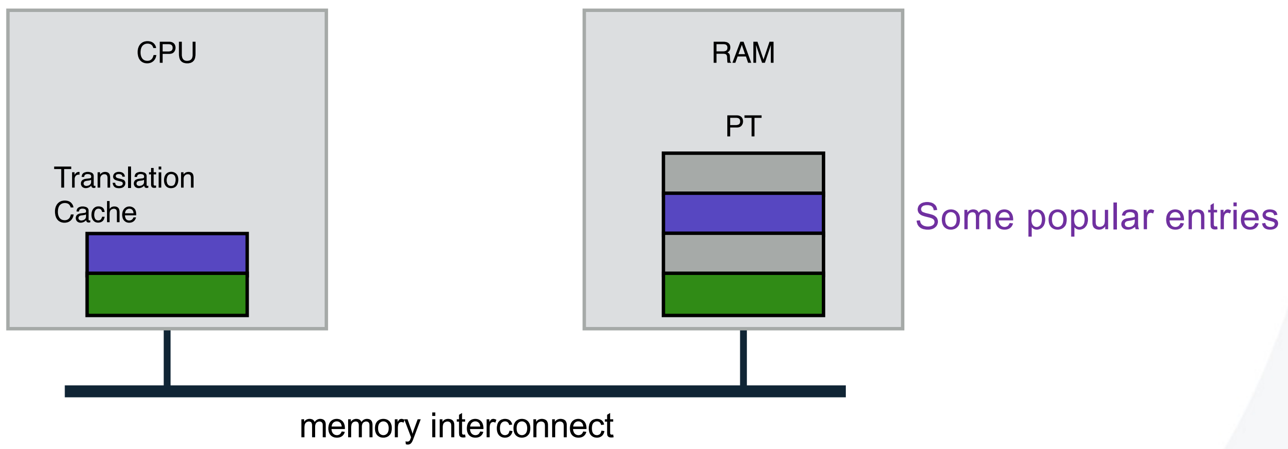


Recall: Locality of Reference

- **Leverage locality of reference within processes**
  - Spatial: reference memory addresses near previously referenced addresses
  - Temporal: reference memory addresses that have referenced in the past
  - Processes spend majority of time in small portion of code



Recall: Cache Page Translations



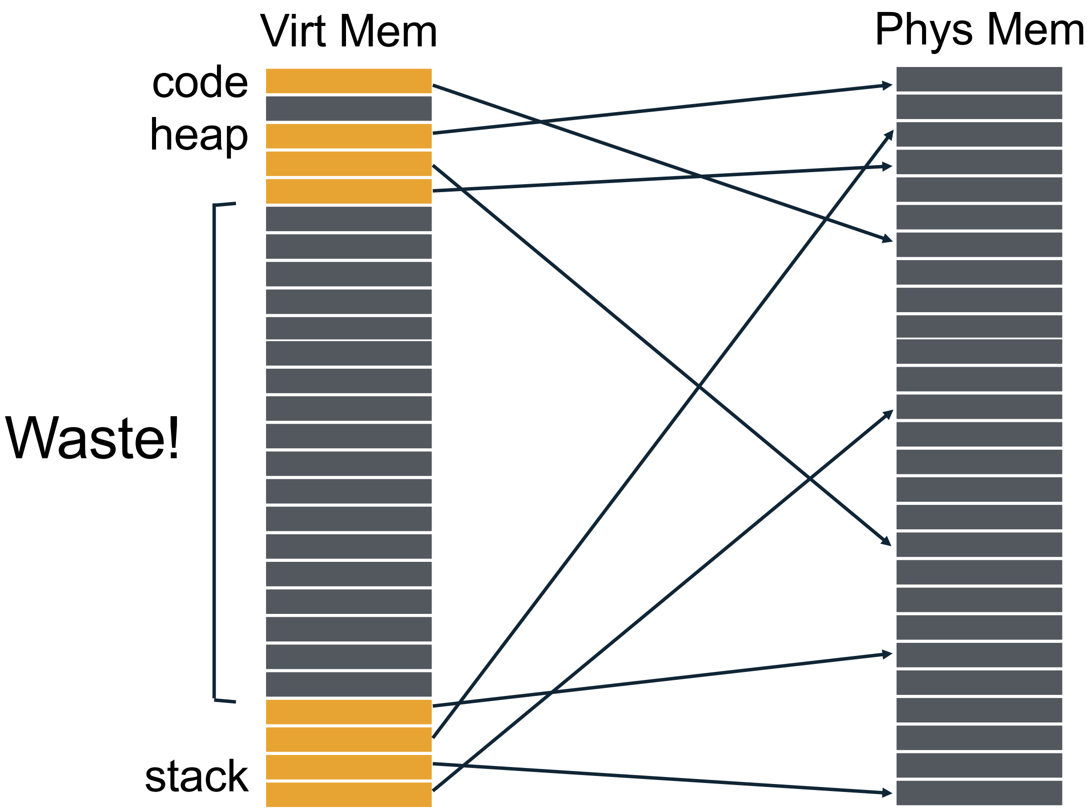
TLB: Translation Lookaside Buffer

Recall: Context Switches

- **What happens if a process uses cached TLB entries from another process?**
- **Solutions?**

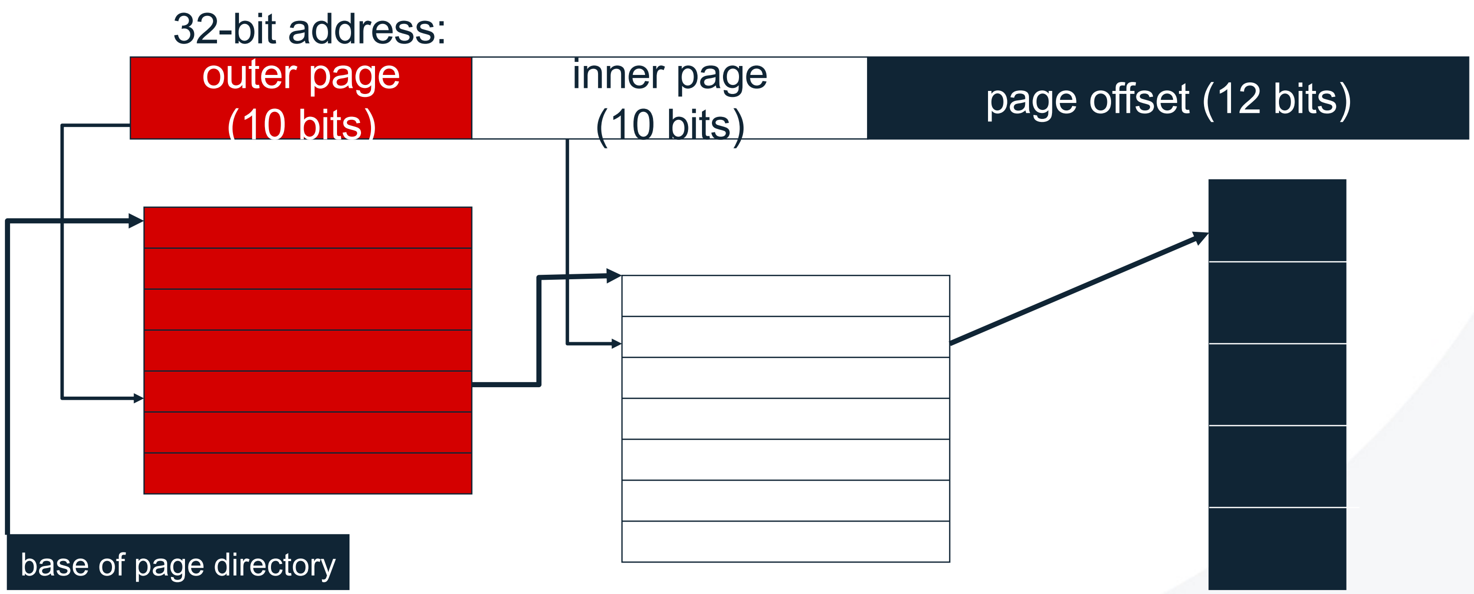
1. Flush TLB on each switch
   - **Costly** → lose all recently cached translations
2. Track which entries are for which process
   - Address Space Identifier
   - Tag each TLB entry with an 8-bit ASID

Recall: Why are page tables so large?



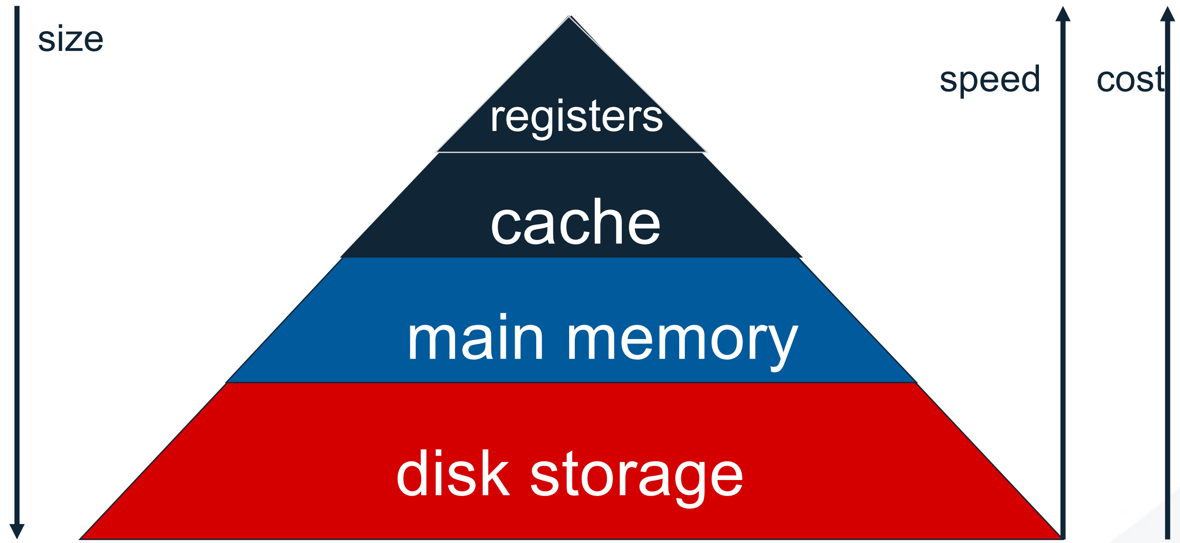
Recall: Multilevel Page Tables

- **Goal: Allow each page tables to be allocated non-contiguously**
- **Idea: Page the page tables**



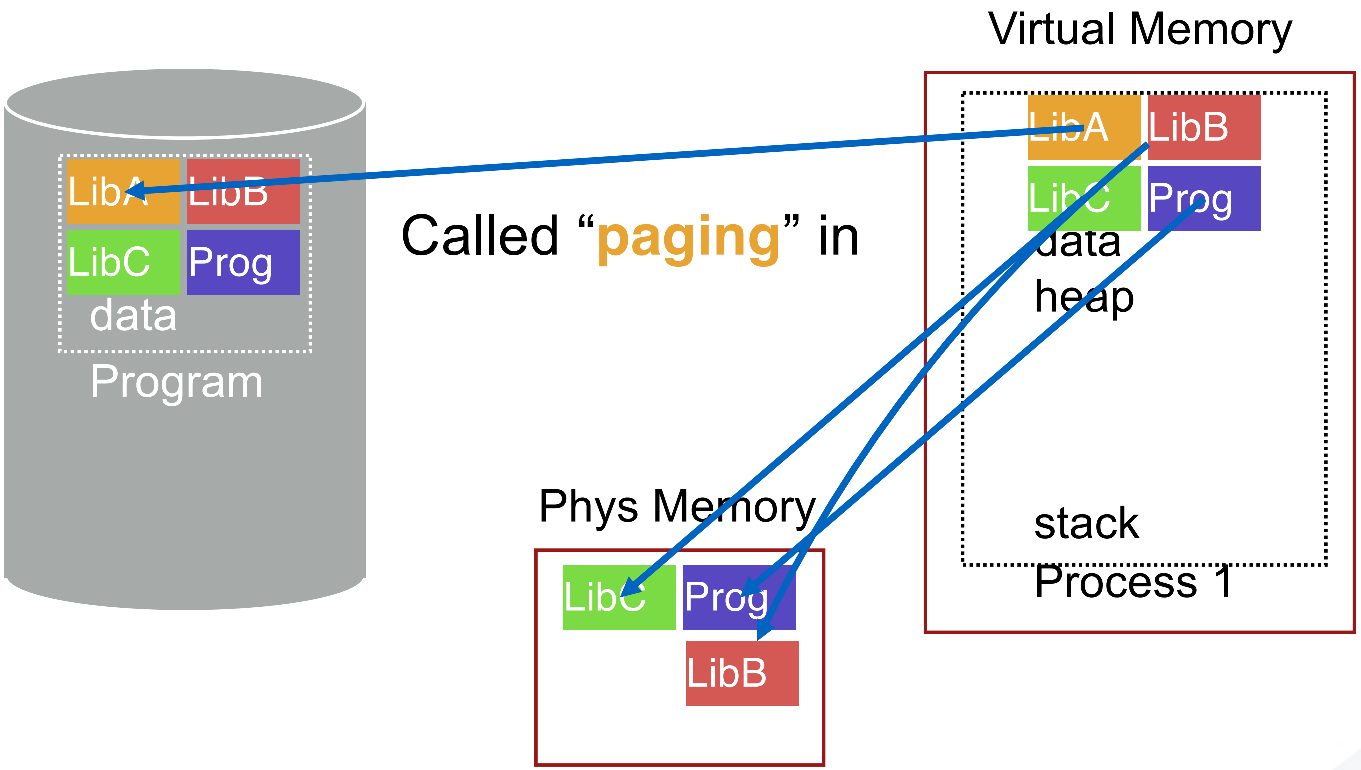
Recall: Memory Hierarchy

- Leverage memory hierarchy of machine architecture
- Each layer acts as “backing store” for layer above



Recall: Swapping

Called "**paging**" in



Recall: Page Selection

- **When should a page be brought from disk into memory?**
  - Demand paging: Load page only when page fault occurs
    - Intuition: Wait until page must absolutely be in memory
    - When process starts: No pages are loaded in memory
    - Problems: Pay cost of page fault for every newly accessed page
  - Prepaging (anticipatory, prefetching): Load page before referenced
    - OS predicts future accesses (oracle) and brings pages into memory early
    - Works well for some access patterns (e.g., sequential)

Recall: Page Replacement – Opt, FIFO, LRU

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

Compulsory

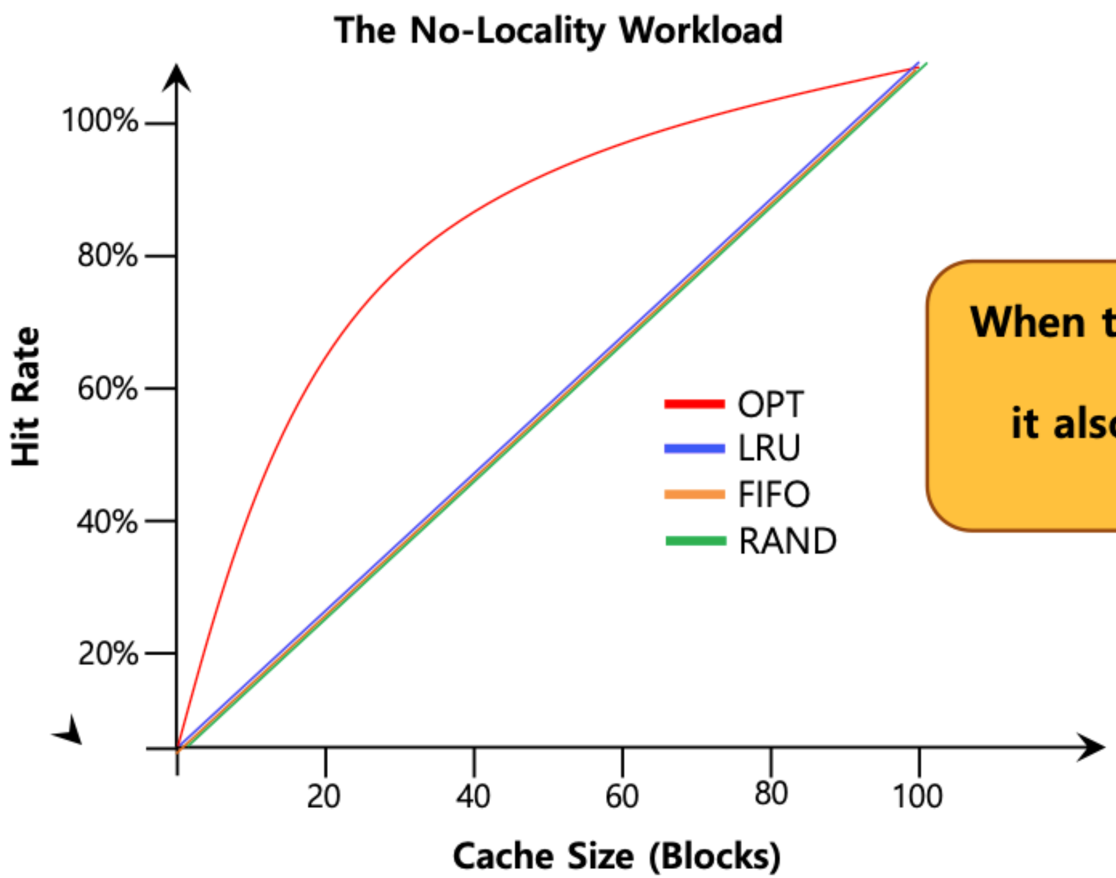
Capacity

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{54.6\%}$

Future is not known.

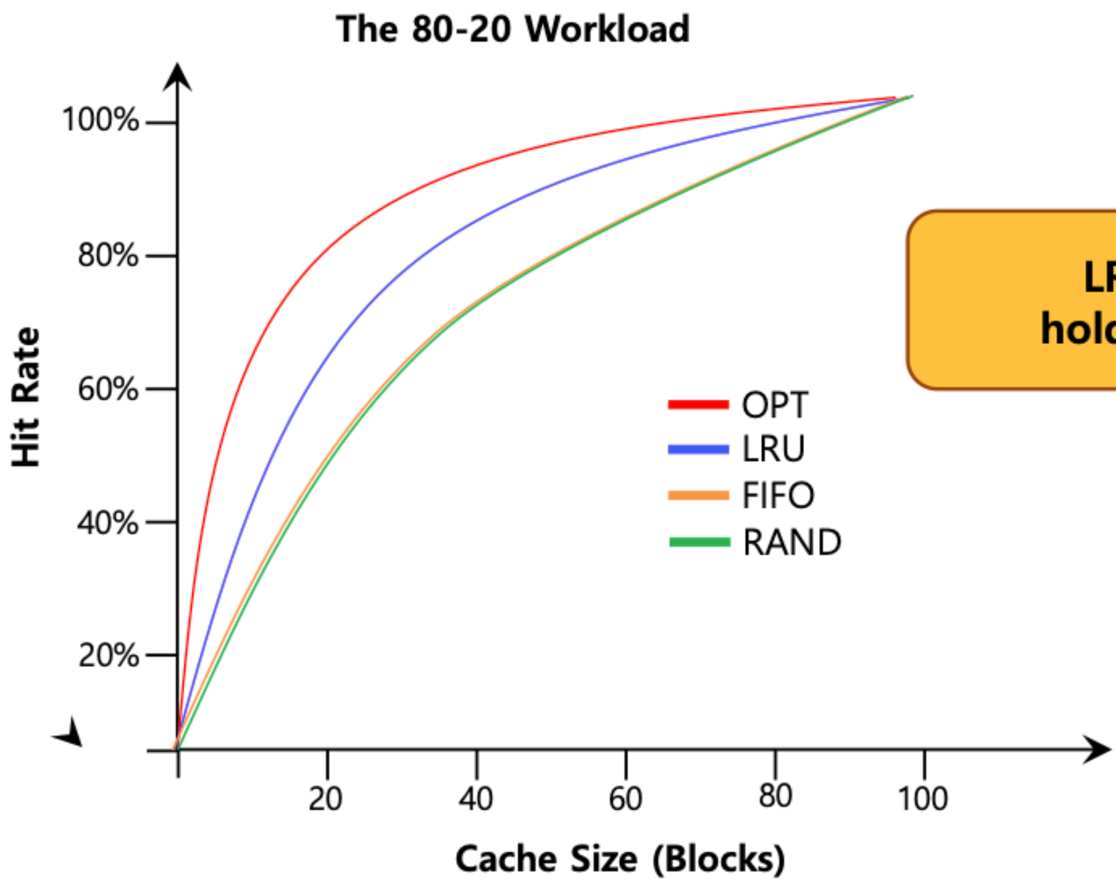
THE UNIVERSITY of ADELAIDE

Recall: The No-Locality Workload Example



When the cache is large enough to fit the entire workload, it also doesn't matter which policy you use.

Recall: The 80-20 Workload Example



LRU is more likely to hold onto the hot pages.

THE UNIVERSITY of ADELAIDE

Recall: Is LRU guaranteed to perform well?

Consider the following: A B C D A B C D A B C D

LRU Performance with capacity of 3:

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

Every reference is a page fault!

Fairly contrived example of working set of N+1 on N frames

THE UNIVERSITY of ADELAIDE

Recall: Clock Algorithm (1)

Clock Algorithm: Arrange physical pages in circle with single clock hand

Approximate LRU (*approximation to approximation to MIN*)

Replace an old page, not the oldest page

Details:

Hardware "use" bit per physical page (called "accessed" in Intel architecture):

- Hardware sets use bit on each reference
- If use bit is not set, means not referenced in a long time
- Some hardware sets use bit in the TLB; must be copied back to PTE when TLB entry gets replaced

On page fault:

- Advance clock hand (not real time)
- Check use bit: 1→ used recently; clear and leave alone
  0→ selected candidate for replacement

Single Clock Hand:

Advances on page fault!
Check for pages not used recently
Mark pages as not used recently

Recall: Clock Algorithm (2)

Will always find a page or loop forever?

Even if all use bits set, will eventually loop all the way around ⇒ FIFO

What if hand moving slowly?

Good sign or bad sign?

- Not many page faults
- or find page quickly

What if hand is moving quickly?

Lots of page faults and/or lots of reference bits set

What about “modified” (or “dirty”) pages?

Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?

Single Clock Hand:

Advances on page fault!

Check for pages not used recently

Mark pages as not used recently

THE UNIVERSITY *of* ADELAIDE

Recall: PTE bits

Which bits of a PTE entry are useful to us for the Clock Algorithm?

The “Present” bit (called “Valid” elsewhere):

- P==0: Page is invalid and a reference will cause page fault
- P==1: Page frame number is valid and MMU is allowed to proceed with translation

The “Writable” bit (could have opposite sense and be called “Read-only”):

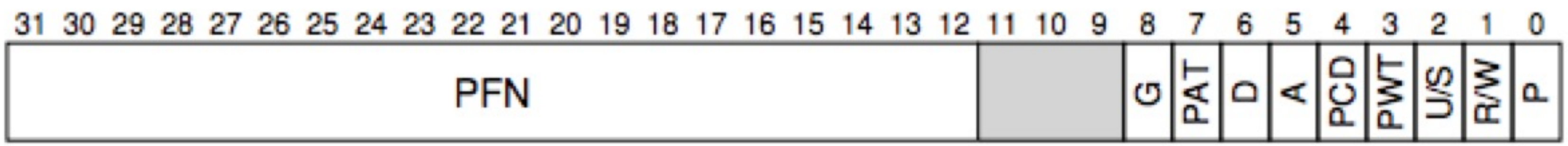
- W==0: Page is read-only and cannot be written.
- W==1: Page can be written

The “Accessed” bit (called “Use” elsewhere):

- A==0: Page has not been accessed (or used) since last time software set A→0
- A==1: Page has been accessed (or used) since last time software set A→0

The “Dirty” bit (called “Modified” elsewhere):

- D==0: Page has not been modified (written) since PTE was loaded
- D==1: Page has changed since PTE was loaded



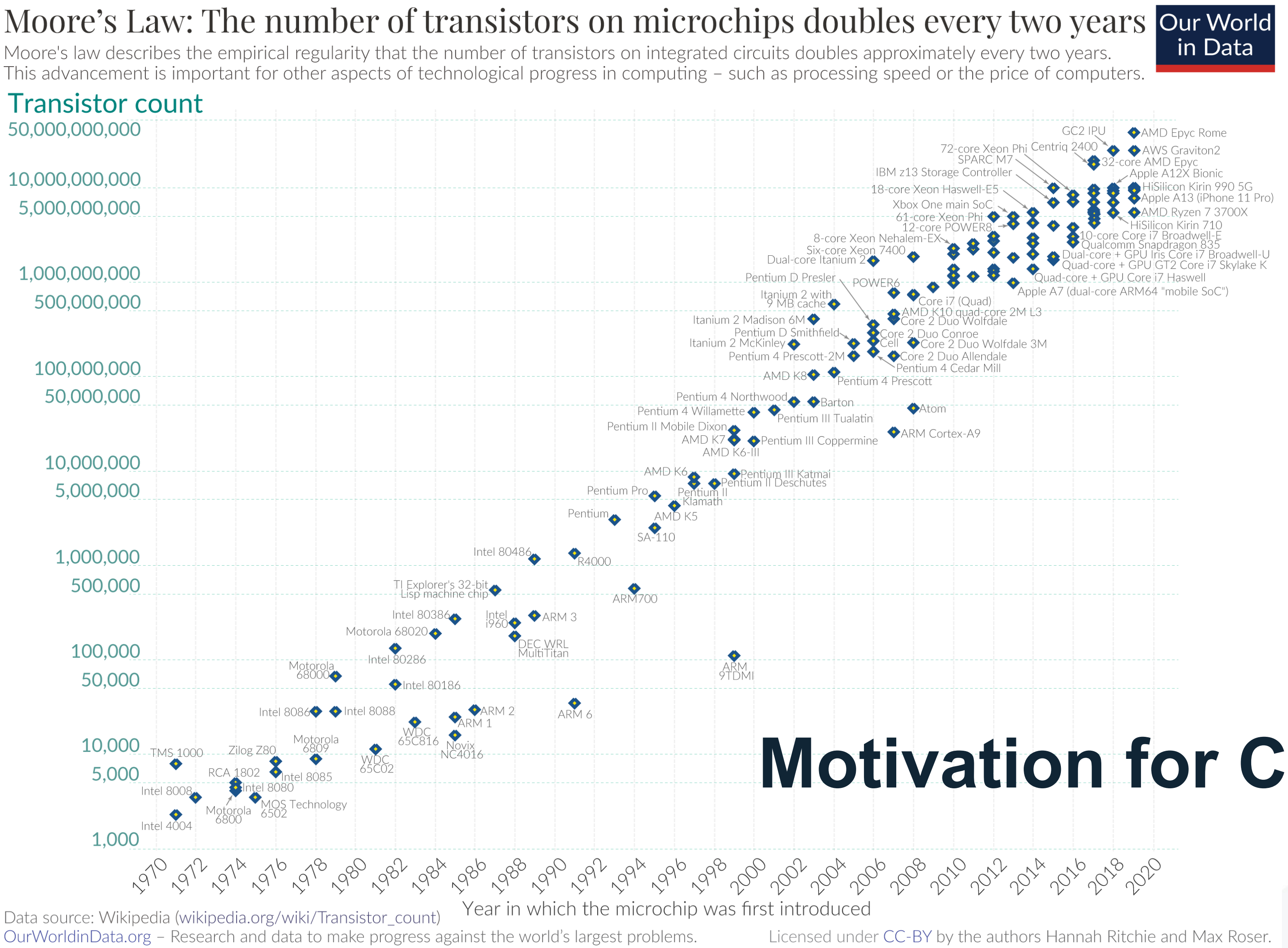
make history.

THE UNIVERSITY of ADELAIDE

COMP SCI 3004 Operating Systems

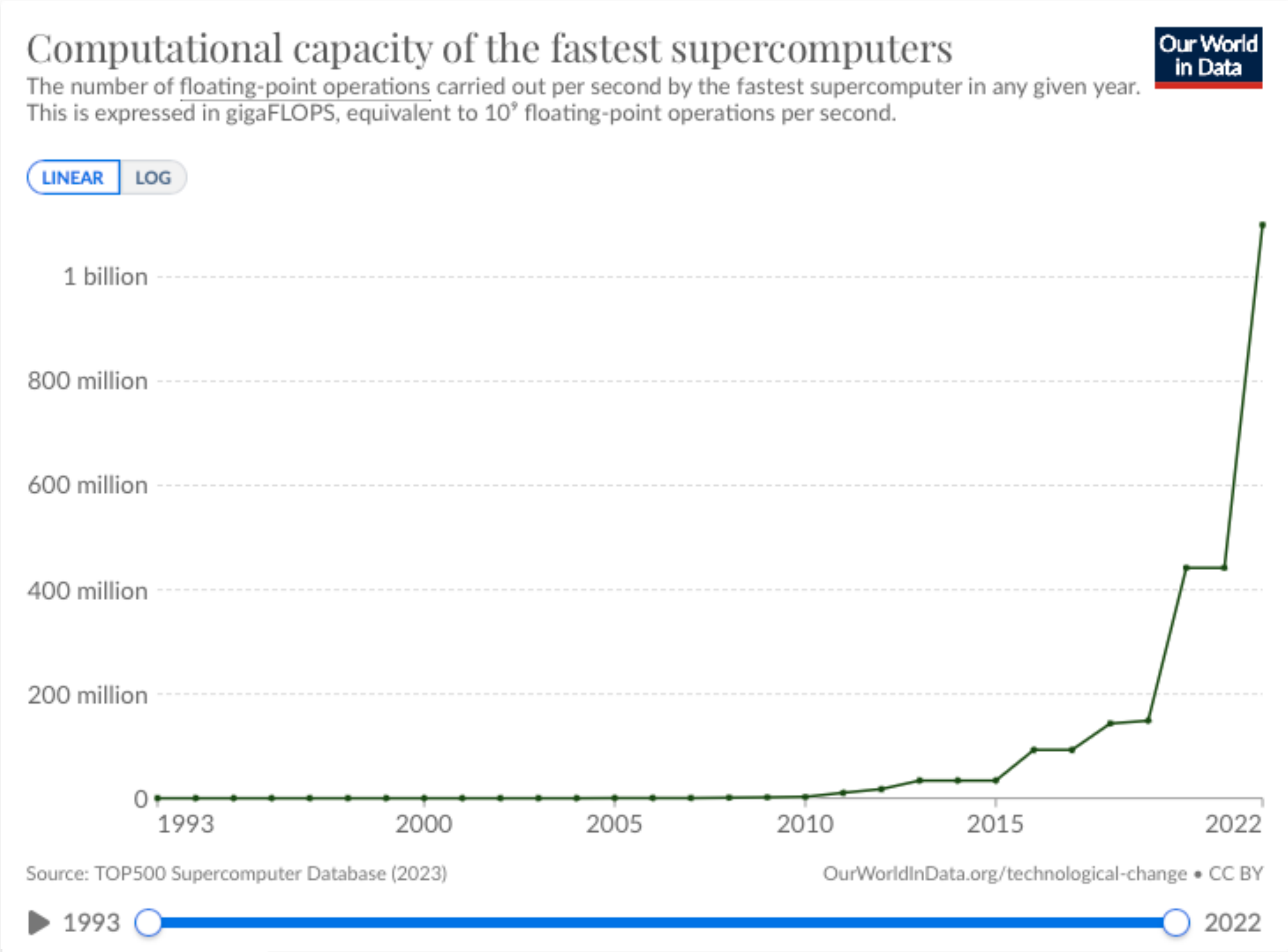
Concurrency and Threads

Motivation for Concurrency



https://en.wikipedia.org/wiki/Moore%27s_law

THE UNIVERSITY of ADELAIDE



https://ourworldindata.org/moores-law

THE UNIVERSITY of ADELAIDE

Motivation

- **CPU Trend: Same speed, but multiple cores**
- **Goal: Write applications that fully utilise many cores**
  - Option 1: Build apps from many communicating processes
- **Pros?**
  - Don’t need new abstractions; good for security
- **Cons?**
  - Cumbersome programming
  - High communication overheads
  - More expensive context switching

THE UNIVERSITY *of* ADELAIDE

Why threads?

- **Performance**
  - Parallelism is the only way to translate multiple cores into performance.
  - Parallelisation: from single-threaded programs to multi-threaded
- **Convenience**
  - Way to overlap I/O with useful work: approach of server-based applications such as web servers, DBMS, etc..
- **Why threads and not processes?**
  - In threads, it is much easier to share data
  - Less pressure over the memory
  - Processes when the tasks are separated with little (to none) sharing

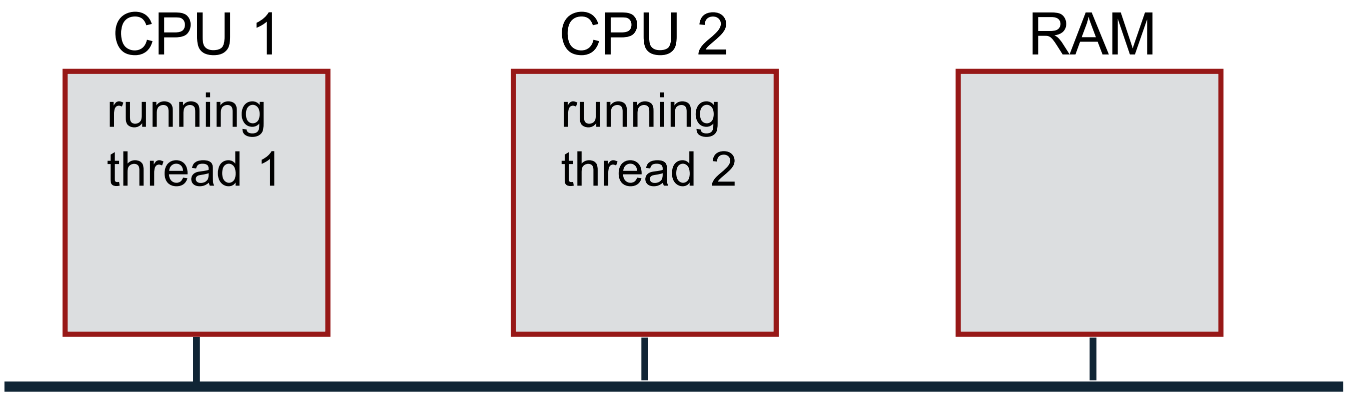
THE UNIVERSITY of ADELAIDE

Concurrency: Option 2

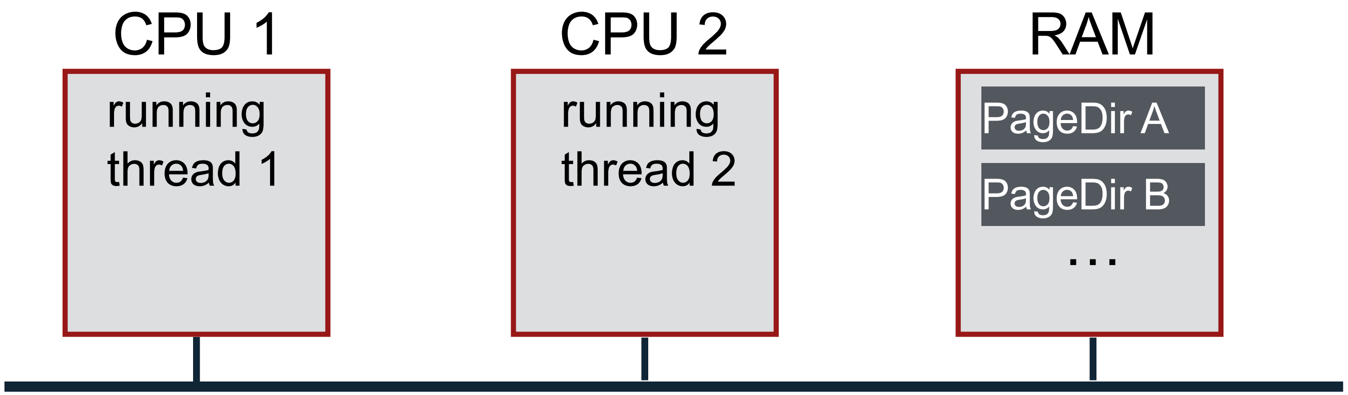
- New abstraction for a single running process: **thread**
- Threads are like processes, except:
  - multiple threads of same process **share** the same **address space**.
  - Multiple PCs (Program Counter)
- Divide large task across several cooperative threads
- Communicate through shared address space

Common Programming Models

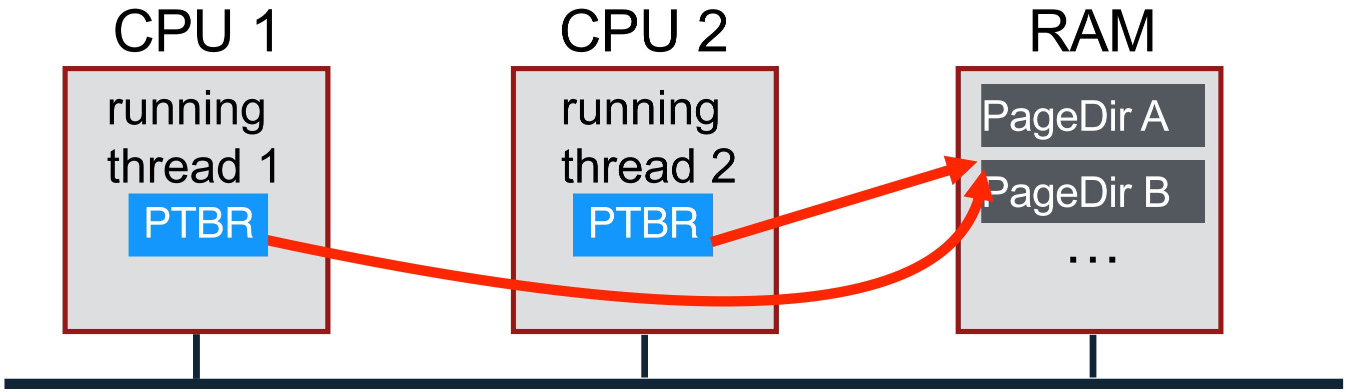
- **Multi-threaded programs tend to be structured as:**
  - Producer/consumer
    Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
  - Pipeline
    Task is divided into series of subtasks, each of which is handled in series by a different thread
  - Defer work with background thread
    One thread performs non-critical work in the background (when CPU idle)

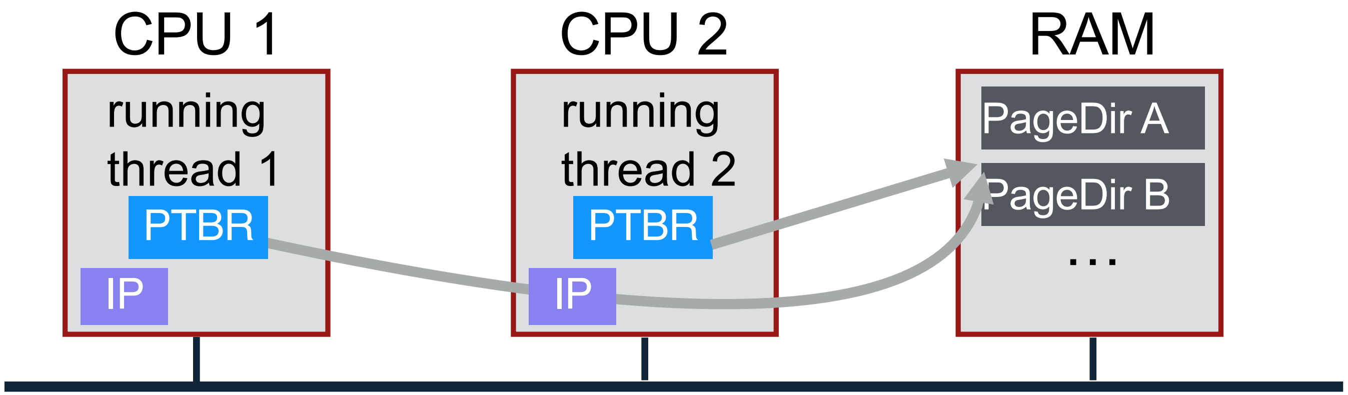


What state do threads share?

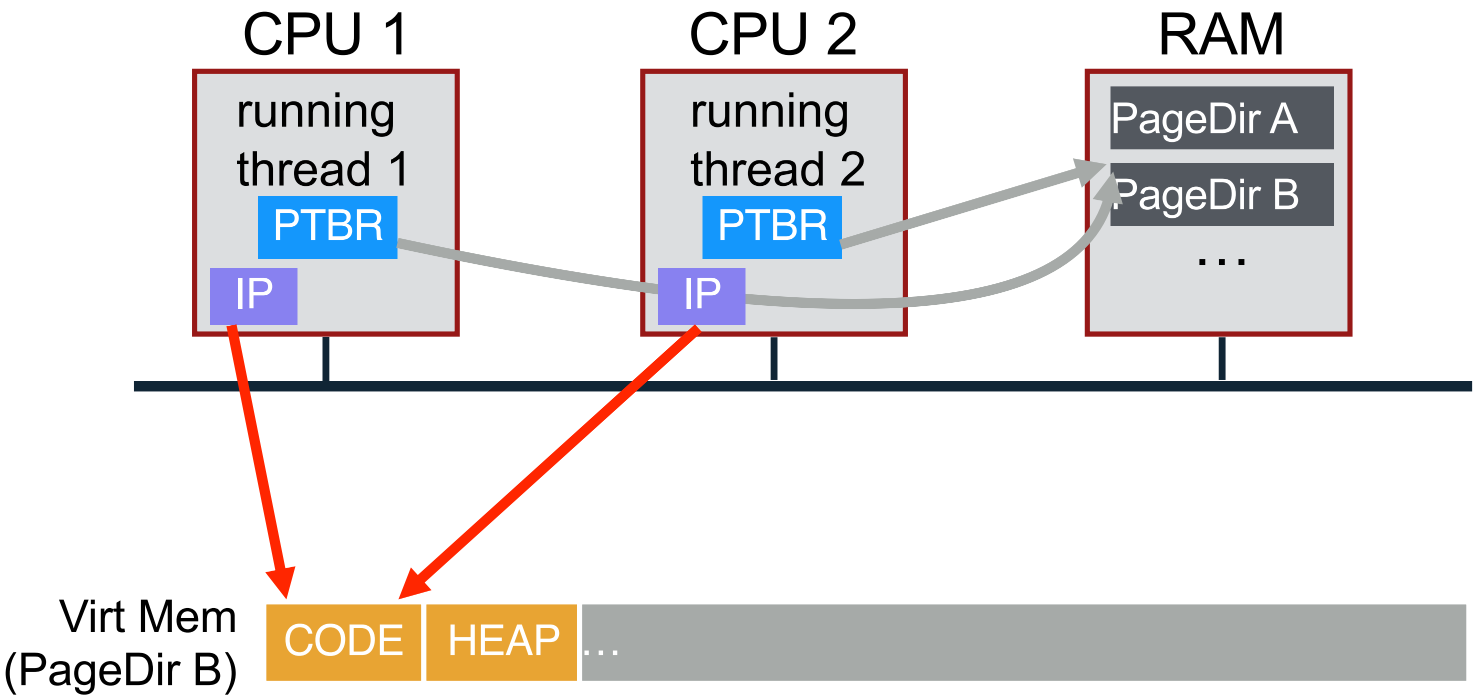


What threads share page directories?

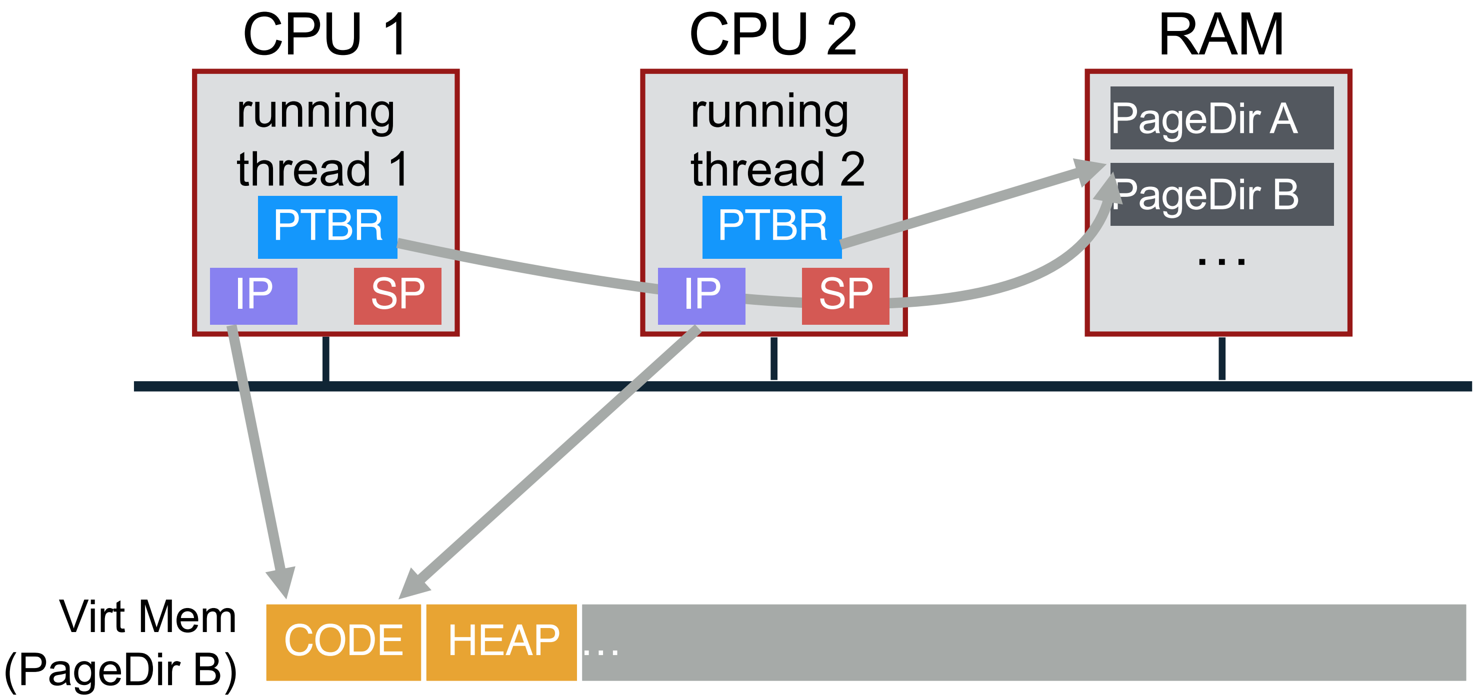

CPU 1
running
thread 1
PTBR
CPU 2
running
thread 2
PTBR
RAM
PageDir A
PageDir B
...



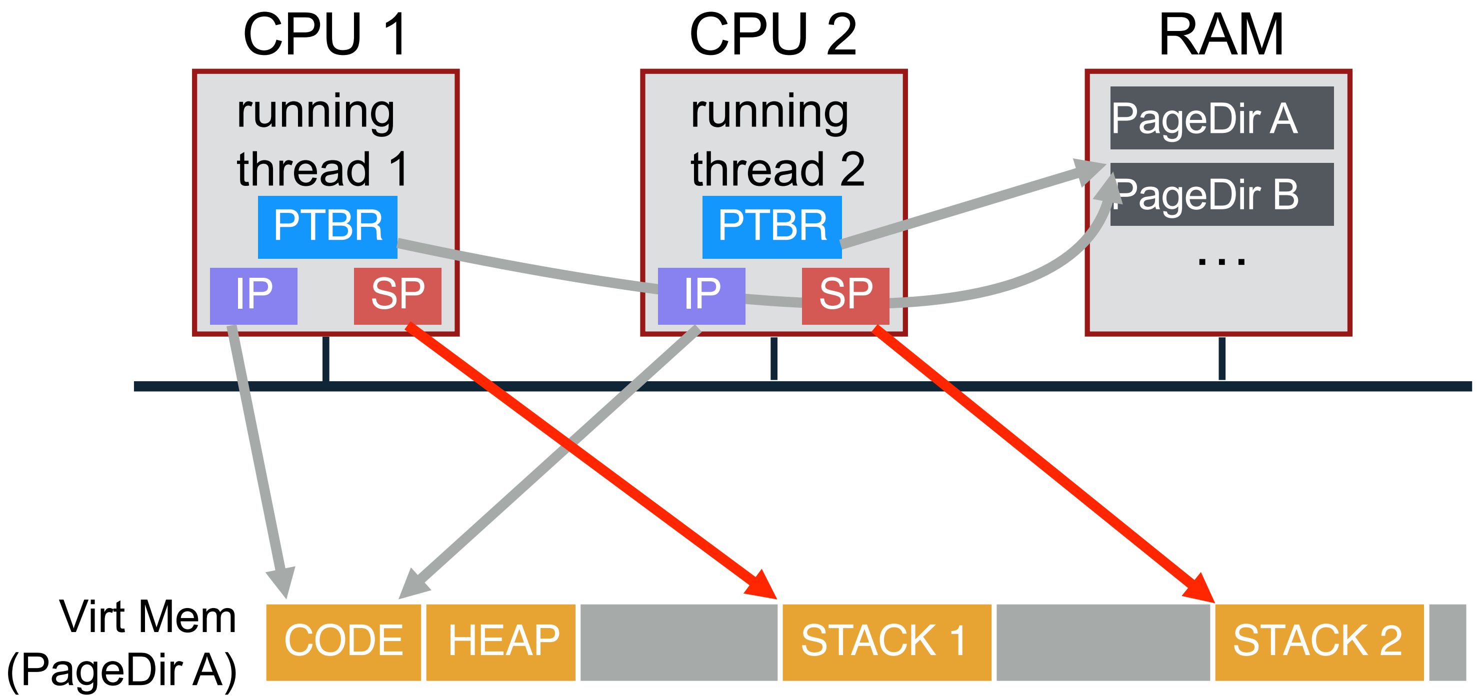
Do threads share Instruction Pointer?



Share code, but each thread may be executing different code at the same time → Different Instruction Pointers



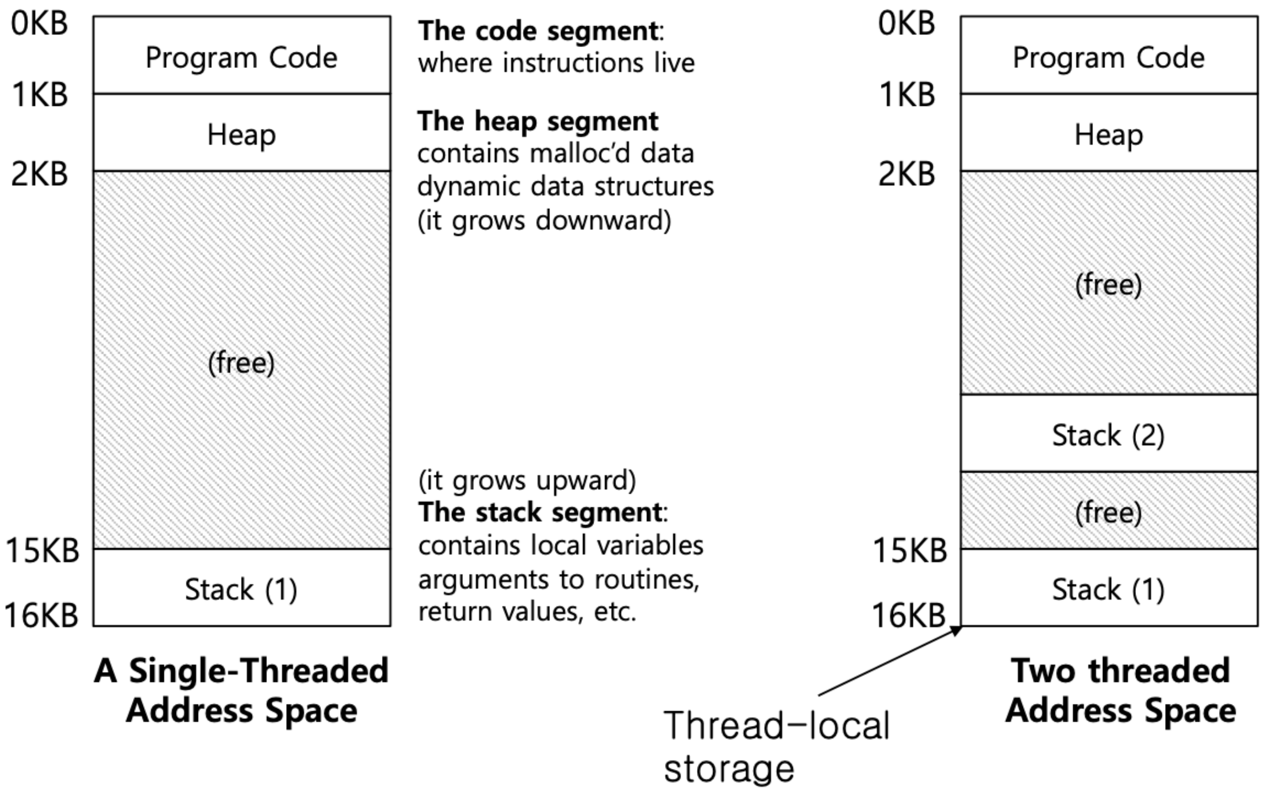
Do threads share stack pointer?



Threads executing different functions need different stacks

The stack of the relevant thread

- There will be **one stack per thread**.



Thread vs. Process

- **Multiple threads within a single process share:**
  - Process ID (PID)
  - Address space
    - Code (instructions)
    - Most data (heap)
  - Open file descriptors
  - Current working directory
  - User and group ID

Thread vs. Process

- **Each thread has its own**
  - Thread ID (TID)
  - Set of registers, including Program counter and Stack pointer
  - Stack for local variables and return addresses
    (in same address space)

Context switch between threads

- Each thread has its own program counter and set of registers.
  - One or more thread control blocks (TCBs) are needed to store the state of each thread.
  - All of them within a common PCB
- When switching from running one (T1) to running the other (T2),
  - The register state of T1 be saved.
  - The register state of T2 restored.
  - The **address space remains** the same.

THE UNIVERSITY of ADELAIDE

Example: Simple Threat Creation Code

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

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Figure 26.2: **Simple Thread Creation Code (t0.c)**

THE UNIVERSITY of ADELAIDE

OS Library API for Threads: *pthreads*

pThreads: POSIX standard for thread programming [POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)]

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

thread is created executing *start_routine* with *arg* as its sole argument.

return is implicit call to pthread_exit

```
void pthread_exit(void *value_ptr);
```

terminates the thread and makes *value_ptr* available to any successful join

```
int pthread_yield();
```

causes the calling thread to yield the CPU to other threads

```
int pthread_join(pthread_t thread, void **value_ptr);
```

suspends execution of the calling thread until the target *thread* terminates.

On return with a non-NULL *value_ptr* the value passed to *pthread_exit()* by the terminating thread is made available in the location referenced by *value_ptr*.

man pthread

THE UNIVERSITY of ADELAIDE

Possible outcomes

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

THE UNIVERSITY of ADELAIDE

Uh Oh

Figure 26.6:
Sharing Data (t1.c)

```
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Possible outcomes

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

THE UNIVERSITY
of ADELAIDE

Thread Schedule #1

counter = counter + 1; counter at 0x9cd4

State:

```
0x9cd4:  100
%eax:     ?
%rip:   0x195
```

process control blocks:

	Thread 1	Thread 2
	%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

T1 →

```
0x195   mov 0x9cd4, %eax
0x19a   add $0x1, %eax
0x19d   mov %eax, 0x9cd4
```

Thread Schedule #1

State:
```
0x9cd4:  100
%eax:    100
%rip:    0x19a
```

process control blocks:

	Thread 1	Thread 2
%eax	?	?
%rip	0x195	0x195

```
     0x195   mov 0x9cd4, %eax
T1 → 0x19a   add $0x1, %eax
     0x19d   mov %eax, 0x9cd4
```

Thread Schedule #1

State:
`0x9cd4: 100`
`%eax:   101`
`%rip:   0x19d`

process control blocks:

	Thread 1	Thread 2
%eax	?	?
%rip	0x195	0x195

```
        0x195   mov 0x9cd4, %eax
        0x19a   add $0x1, %eax
T1 →    0x19d   mov %eax, 0x9cd4
```

Thread Schedule #1

State:
`0x9cd4: 101`
`%eax: 101`
`%rip: 0x1a2`

process control blocks:

	Thread 1	Thread 2
	%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

T1 →

Thread Context Switch

THE UNIVERSITY of ADELAIDE

Thread Schedule #1

State:
`0x9cd4: 101`
`%eax:   ?`
`%rip:  0x195`

process control blocks:

	Thread 1	Thread 2
	%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

```
T2 ➡ 0x195  mov 0x9cd4, %eax
     0x19a  add $0x1, %eax
     0x19d  mov %eax, 0x9cd4
```

THE UNIVERSITY of ADELAIDE

Thread Schedule #1

State:

```
0x9cd4:  101
%eax:    101
%rip:    0x19a
```

process control blocks:

	Thread 1	Thread 2
	%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

```
      0x195  mov 0x9cd4, %eax
T2 →  0x19a  add $0x1, %eax
      0x19d  mov %eax, 0x9cd4
```

THE UNIVERSITY of ADELAIDE

Thread Schedule #1

State:

```
0x9cd4:  101
%eax:    102
%rip:   0x19d
```

process control blocks:

	Thread 1	Thread 2
	%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

```
     0x195  mov 0x9cd4, %eax
     0x19a  add $0x1, %eax
T2 → 0x19d  mov %eax, 0x9cd4
```

THE UNIVERSITY of ADELAIDE

Thread Schedule #1

State:

```
0x9cd4:  102
%eax:    102
%rip:   0x1a2
```

process control blocks:

	Thread 1	Thread 2
	%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

T2

Desired Result!

THE UNIVERSITY of ADELAIDE

Thread Schedule #2

State:
```
0x9cd4: 100
%eax:   ?
%rip:   0x195
```

process control blocks:

Thread 1	Thread 2
%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

T1 →
```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

State:
```
0x9cd4:  100
%eax:    100
%rip:    0x19a
```

process control blocks:

	Thread 1	Thread 2
	%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

```
      0x195  mov 0x9cd4, %eax
T1 →  0x19a  add $0x1, %eax
      0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

State:
`0x9cd4: 100`
`%eax:   101`
`%rip:   0x19d`

process control blocks:

	Thread 1	Thread 2
%eax:	?	?
%rip:	0x195	0x195

```
      0x195  mov 0x9cd4, %eax
      0x19a  add $0x1, %eax
T1 →  0x19d  mov %eax, 0x9cd4
```

Thread Context Switch

Thread Schedule #2

State:

```
0x9cd4: 100
%eax:   ?
%rip:   0x195
```

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

```
T2 → 0x195  mov 0x9cd4, %eax
     0x19a  add $0x1, %eax
     0x19d  mov %eax, 0x9cd4
```

THE UNIVERSITY of ADELAIDE

Thread Schedule #2

State:
`0x9cd4: 100`
`%eax:   100`
`%rip:   0x19a`

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

```
      0x195  mov 0x9cd4, %eax
T2 →  0x19a  add $0x1, %eax
      0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

State:

```
0x9cd4:  100
%eax:    101
%rip:   0x19d
```

process control blocks:

	Thread 1	Thread 2
	%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

```
     0x195  mov 0x9cd4, %eax
     0x19a  add $0x1, %eax
T2 → 0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

State:
`0x9cd4: 101`
`%eax:   101`
`%rip:   0x19d`

process control blocks:

	Thread 1	Thread 2
	%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

T2 →

Thread Context Switch

THE UNIVERSITY of ADELAIDE

Thread Schedule #2

State:

```
0x9cd4:  101
%eax:    101
%rip:   0x19d
```

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: 101 %rip: 0x1a2

```
       0x195  mov 0x9cd4, %eax
       0x19a  add $0x1, %eax
T1 →   0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

State:

```
0x9cd4:  101
%eax:    101
%rip:    0x1a2
```

process control blocks:

	Thread 1	Thread 2
	%eax: 101 %rip: 0x19d	%eax: 101 %rip: 0x1a2

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

T1

THE UNIVERSITY of ADELAIDE

Thread Schedule #2

State:
```
0x9cd4:  101
%eax:    101
%rip:   0x1a2
```

process control blocks:	Thread 1	Thread 2
	%eax: 101 %rip: 0x1a2	%eax: 101 %rip: 0x1a2

```
0x195   mov 0x9cd4, %eax
0x19a   add $0x1, %eax
0x19d   mov %eax, 0x9cd4
```

T1

WRONG Result! Final value is 101

THE UNIVERSITY of ADELAIDE

Timeline View

Thread 1

Thread 2

```
mov 0x9cd4, %eax
add %0x1, %eax
mov %eax, 0x9cd4
```

```
mov 0x9cd4 , %eax
add %0x2, %eax
mov %eax, 0x9cd4
```

How much is added to shared variable? **3: correct!**

Timeline View

Thread 1

Thread 2

```
Thread 1                      Thread 2

mov 0x9cd4, %eax
add %0x1, %eax
                              mov 0x9cd4 , %eax
mov %eax, 0x9cd4

                              add %0x2, %eax
                              mov %eax, 0x9cd4
```

How much is added to shared variable? **2: incorrect!**

THE UNIVERSITY of ADELAIDE

Timeline View

Thread 1

Thread 2

```
                      mov 0x9cd4 , %eax
mov 0x9cd4, %eax
                      add %0x2, %eax
add %0x1, %eax
                      mov %eax, 0x9cd4
mov %eax, 0x9cd4
```

How much is added to shared variable? **1: incorrect!**

THE UNIVERSITY of ADELAIDE

Timeline View

Thread 1	Thread 2
	`mov 0x9cd4 , %eax`
	`add %0x2, %eax`
`mov 0x9cd4, %eax`	
`add %0x1, %eax`	
`mov %eax, 0x9cd4`	
	`mov %eax, 0x9cd4`

How much is added to shared variable? **2: incorrect!**

Timeline View

Thread 1

Thread 2

```
mov 0x9cd4 , %eax
add %0x2, %eax
mov %eax, 0x9cd4
```

```
mov 0x9cd4, %eax
add %0x1, %eax
mov %eax, 0x9cd4
```

How much is added to shared variable? **3: correct!**

THE UNIVERSITY of ADELAIDE

What do we want?

We want 3 instructions to execute as an uninterruptable group

That is, we want them to be atomic

```
mov 0x9cd4, %eax
add %0x1, %eax
mov %eax, 0x123
```

critical section

More general:

Need mutual exclusion for critical sections

- if process A is in critical section C, process B can't
- (okay if other processes do unrelated work)

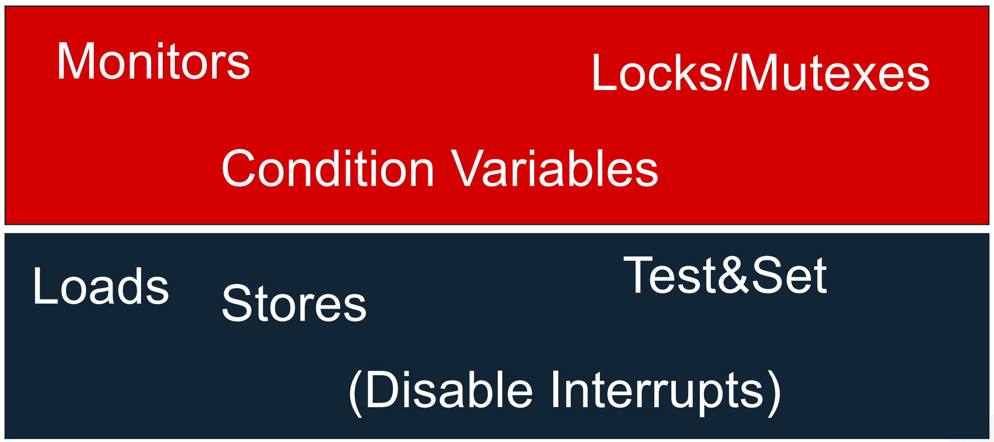
THE UNIVERSITY of ADELAIDE

Non-Determinism

- **Concurrency leads to non-deterministic results**
  - Not deterministic result: different results even with same inputs
  - race conditions
- **Whether bug manifests depends on CPU schedule!**
- **Passing tests means little**
- **How to program: imagine scheduler is malicious**
- **Assume scheduler will pick bad ordering at some point…**

Synchronisation

- **Build higher-level synchronization primitives in OS**
  - Operations that ensure correct ordering of instructions across threads
- **Motivation: Build them once and get them right**



Wishing for atomicity

- Do the read and modification of the memory in a single step
  - i.e. “all or nothing”!
- How ho handle complex data?
  - Use some atomic hardware support (called synchronization primitives) to construct OS support
- A piece of code that accesses a shared variable and must not be concurrently executed by more than one thread (mixing R and W).
  - Multiple threads executing critical section can result in a race condition.
  - Need to support atomicity for critical sections (mutual exclusion)

THE UNIVERSITY *of* ADELAIDE

Locks

- **Goal: Provide mutual exclusion (mutex)**
- **Three common operations:**
  - Allocate and Initialize
    - `Pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;`
  - Acquire
    - Acquire exclusion access to lock;
    - Wait if lock is not available (some other process in critical section)
    - Spin or block (relinquish CPU) while waiting
    - `Pthread_mutex_lock(&mylock);`

THE UNIVERSITY of ADELAIDE

Locks

- Release
  - Release exclusive access to lock; let another process enter critical section
  - `Pthread_mutex_unlock(&mylock);`

Locks

Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

Waiting for another/s

Sometimes the thread interaction is wait for another thread

- When a thread should wait to another that had issued a I/O
- Need to be slept until the other thread receives the I/O end

Sometimes the action of multiple threads should be synchronous

- Many threads are performing in parallel an iteration in a numerical problem
- All threads should start the next iteration at once (barrier)

This sleeping/waking cycle will be controlled by condition variables

Conclusions

- Concurrency is needed to obtain high performance by utilizing multiple cores
- Threads are multiple execution streams within a single process or address space (share PID and address space, own registers and stack)
- Context switches within a critical section can lead to non-deterministic bugs (race conditions)
- Use locks to provide mutual exclusion