Dependability, reliability and security research at TalTech

TalTech School of IT (https://taltech.ee/en/school-of-information-technologies) in Estonia has five departments: Software Science, Computer Systems, Electronics, Health Technologies, and IT College.

Professor Gert Jervan is the Dean at the School of Information Technologies. In this talk he gives an overview of the research conducted in various departments of the School. Thereafter, a more detailed discussion of the research conducted in the Department of Computer Systems will follow. Various topics, such as hardware security (obfuscation, side-channel attacks, PUFs), reliability of AI accelerators, and hardware trustworthiness will be covered.

This talk is open for all. Especially if you are interested in studying abroad. The University of Adelaide and TalTech are signing an MoU regarding student mobility and study changes. This also includes research exchanges.

Next TUESDAY (8.8.2023) @ 10.00 am in IW 5.57

THE UNIVERSITY of ADELAIDE

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.

Operating Systems

Week 2:

Scheduling and Memory

make history.

THE UNIVERSITY of ADELAIDE

Tutorials – next week!

2023 Semester 2

- Home
- Attendance
- Announcements
- Modules
- Course Readings
- Piazza
- Echo 360
- Syllabus
- **Assignments**
- Assignment Help
- Grades

Tutorial 1

Published · Edit

Tutorial 1 questions are available here

Please make an attempt prior to your workshop session and bing your answers for discussion and revision.

Checking attendance and participation (grade will be entered by the tutor).

Points 1.2

Submitting Nothing

Due	For	Available from	Until
11 Aug at 12:00	Everyone	-	-

THE UNIVERSITY of ADELAIDE

Tutorial (2)

Operating Systems, Tutorial 1.

Week3

Question 1.

(a) What is the difference between kernel mode and user mode? Why is the difference important to an OS?

(b) Which of the following instructions should be allowed only in kernel mode

1- disable all interrupts
2- read the time-of-day clock
3- set the time-of-day clock
4- change the memory map

Question 2

(a) What is the main difference between a process and a thread?

(b) In a system with threads, is there normally one stack per thread or one stack per process? Explain

Question 3

Draw a diagram that illustrate the transitions of a process state for

a) a non pre-emptive scheduler
b) a pre-emptive scheduler

Give two reasons to support pre-emption.

Question 4

The table below desc…
Assume 0 i…

Quizzes

Quizzes will be released on MyUni on Thursday, 10.8.2023 in the evening.

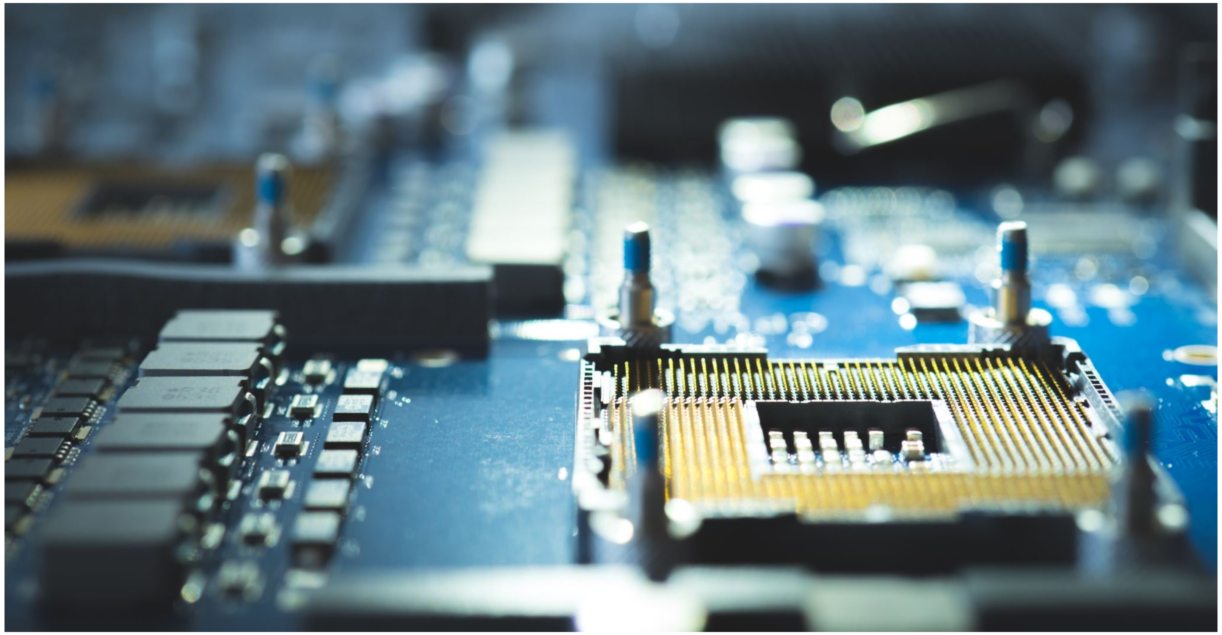
Do online first! Submit before the class starts!

Online Quiz closes at 13.00!

Then In-class session on Friday, 11.8.2023 in the 2nd half of the lecture at around 14.15-15.00!

Submit the scratch card

Recap: Last week



How to virtualise the CPU?

Programs, Processes and Threats.

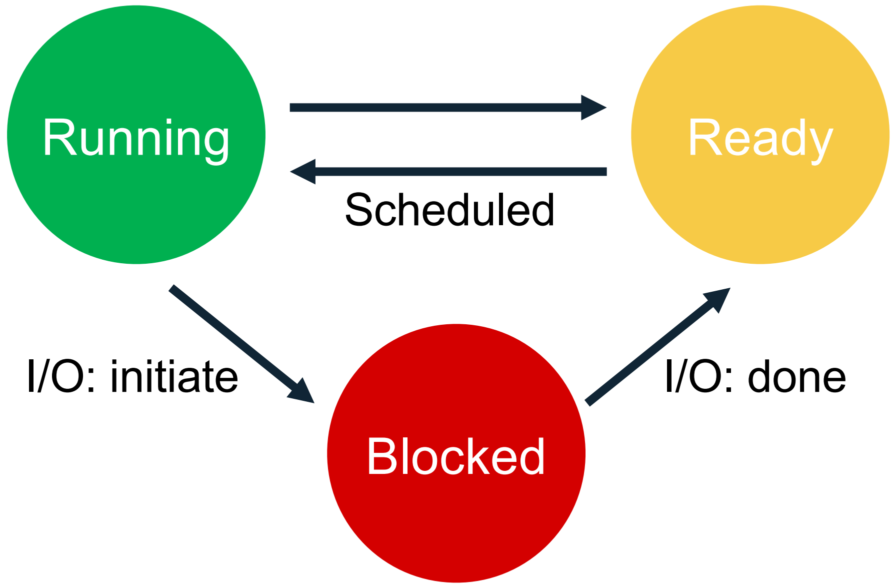
OS as resource mananger

Limited Direct Execution & the need ”to take the CPU away”.

Scheduling the workload.

THE UNIVERSITY of ADELAIDE

Recap: State Transitions



CPU Virtualization: Scheduling

Last week we started to review different scheduling policies, such as: FCFS, SJF.

What type of workload performs well with each scheduler?

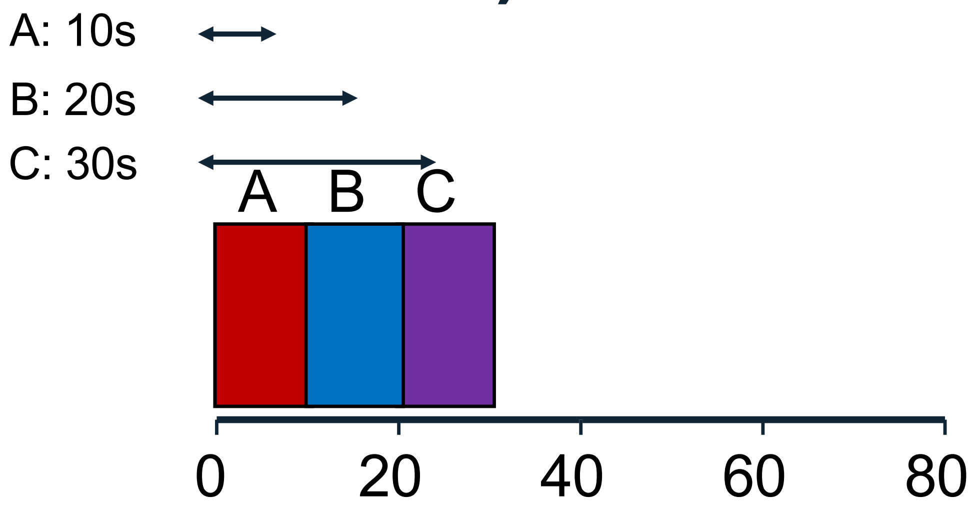
Today: STCF, RR, MLFQ, CFS.

THE UNIVERSITY of ADELAIDE

Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

FIFO: (Identical JOBS)



- **What is the average turnaround time?**
  - Def: turnaround_time = completion_time - arrival_time
  - (10 + 20 + 30) / 3 = **20s**

THE UNIVERSITY of ADELAIDE

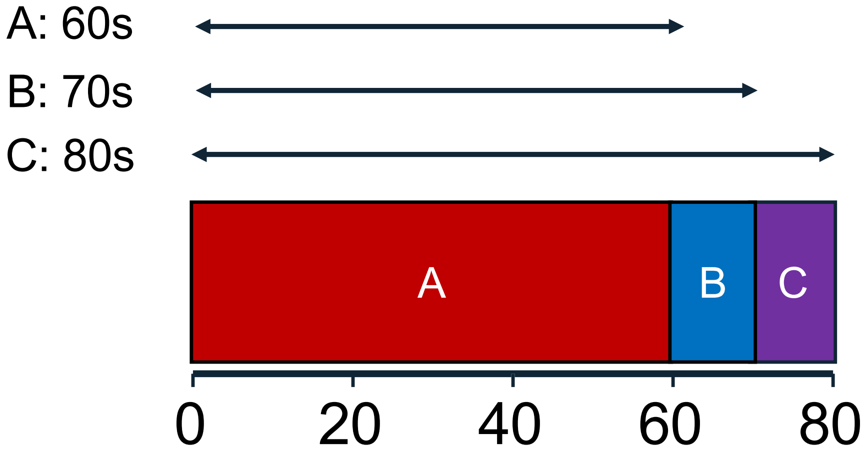
Workload Assumptions

1. ~~Each job runs for the same amount of time~~
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

THE UNIVERSITY of ADELAIDE

Example: Big First Job

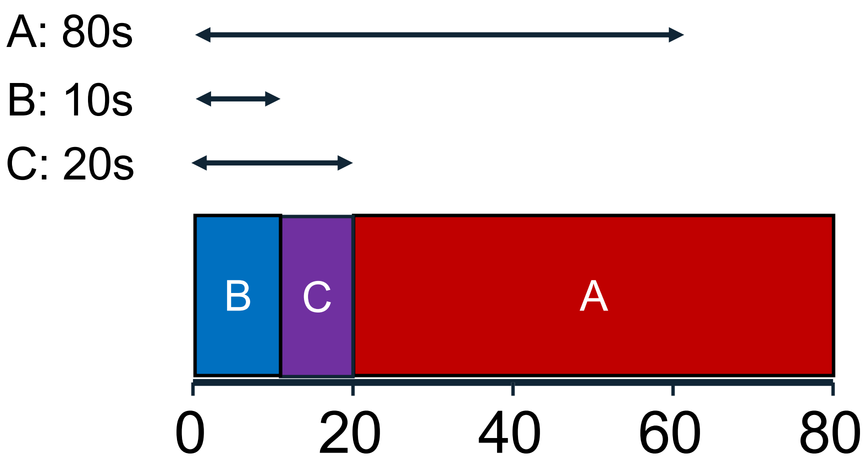
JOB	arrival_time (s)	run_time (s)
A	~0	60
B	~0	10
C	~0	10



Average turnaround time: **70s**

THE UNIVERSITY of ADELAIDE

SJF Turnaround Time



What is the average turnaround time with SJF?

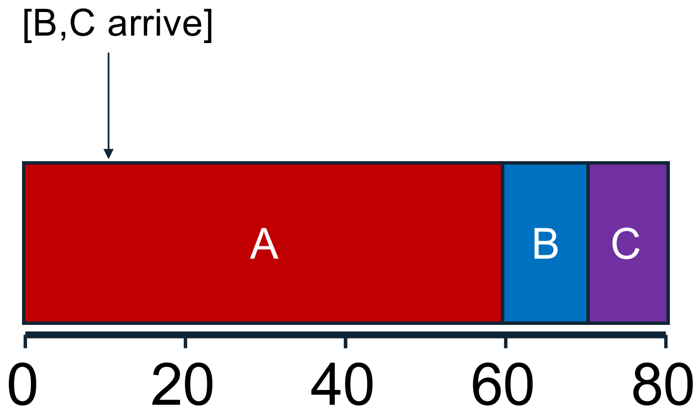
- (80 + 10 + 20) / 3 = **~36.7s**
- **Average turnaround with FIFO: 70s**

Workload Assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

THE UNIVERSITY of ADELAIDE

Stuck Again



JOB	arrival_time (s)	run_time (s)
A	~0	60
B	~10	10
C	~10	10

What is the average turnaround time with SJF?

- (60 + (70 – 10) + (80 – 10)) / 3 = **63.3s**

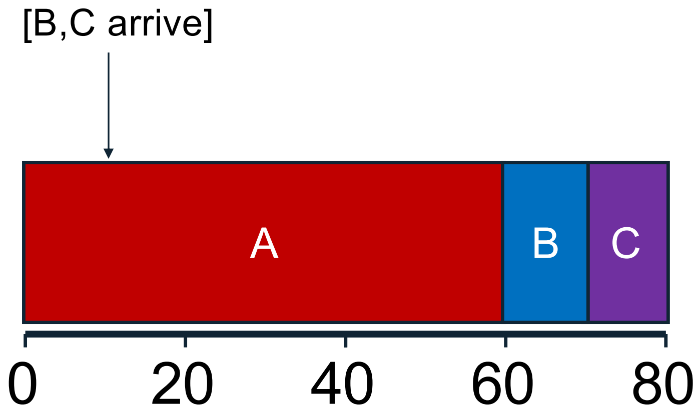
THE UNIVERSITY of ADELAIDE

Preemptive Scheduling

- **Prev schedulers:**
  - FIFO and SJF are non-preemptive
  - Only schedule new job when previous job voluntarily relinquishes CPU (performs I/O or exits)
- **New scheduler:**
  - Preemptive: Potentially schedule different job at any point by taking CPU away from running job
  - STCF (Shortest Time-to-Completion First)
  - Always run job that will complete the quickest

THE UNIVERSITY of ADELAIDE

NON-PREEMPTIVE: STCF



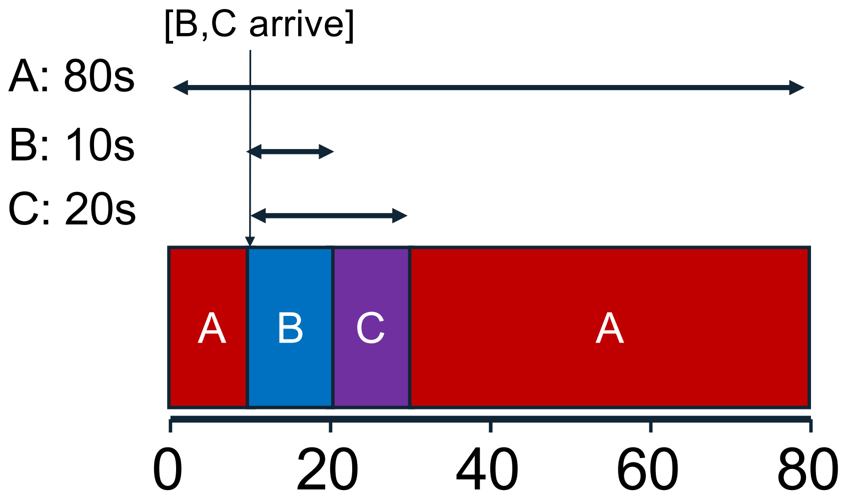
JOB	arrival_time (s)	run_time (s)
A	~0	60
B	~10	10
C	~10	10

What is the average turnaround time with SJF?

- (60 + (70 – 10) + (80 – 10)) / 3 = **63.3s**

THE UNIVERSITY of ADELAIDE

PREEMPTIVE: STCF



JOB	arrival_time (s)	run_time (s)
A	~0	60
B	~10	10
C	~10	10

- Average turnaround time with STCF? **36.6s**
- Average turnaround time with SJF: **63.3s**

Scheduling Basics

Workloads:

- arrival_time
- run_time

Schedulers:

- FIFO
- SJF
- STCF
- RR

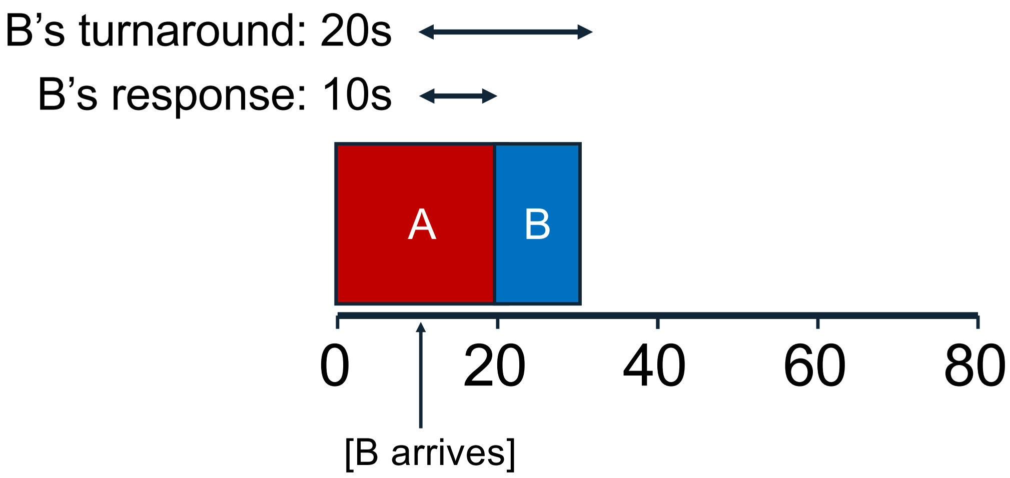
Metrics:

- turnaround_time
- response_time

Response Time

- **Sometimes care about when job starts instead of when it finishes**
- **New metric:**
  - response_time = first_run_time - arrival_time

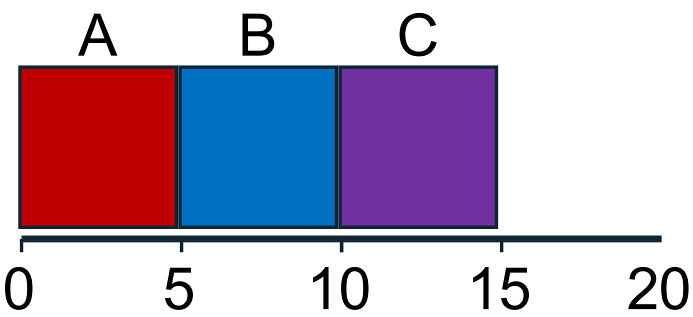
Response vs. Turnaround



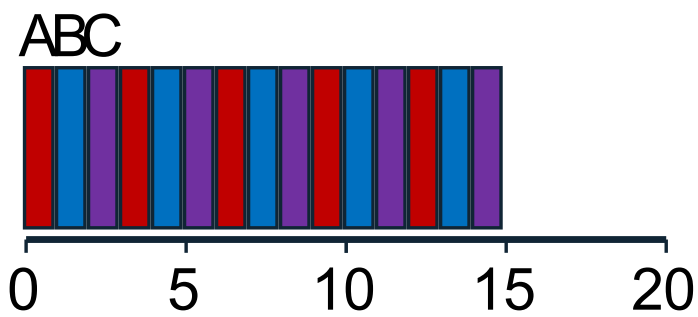
Round-Robin Scheduler

- **Prev schedulers:**
  - FIFO, SJF, and STCF can have poor response time
- **New scheduler: RR (Round Robin)**
  - Alternate ready processes every fixed-length time-slice

FIFO vs RR



Avg Response Time?
(0+5+10)/3 = **5**



Avg Response Time?
(0+1+2)/3 = **1**

In what way is RR worse?

- Ave. turn-around time with equal job lengths is horrible

Other reasons why RR could be better?

- If don’t know run-time of each job, gives short jobs a chance to run and finish fast

Scheduling Basics

Workloads:

- arrival_time
- run_time

Schedulers:

- FIFO
- SJF
- STCF
- RR

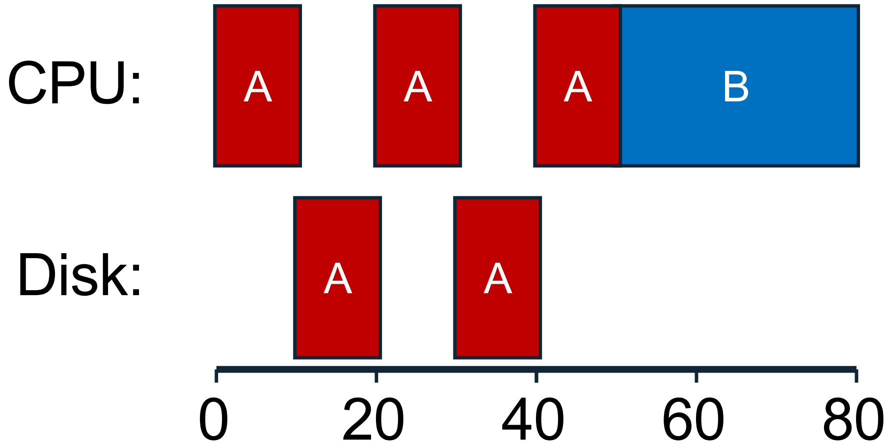
Metrics:

- turnaround_time
- response_time

Workload Assumptions

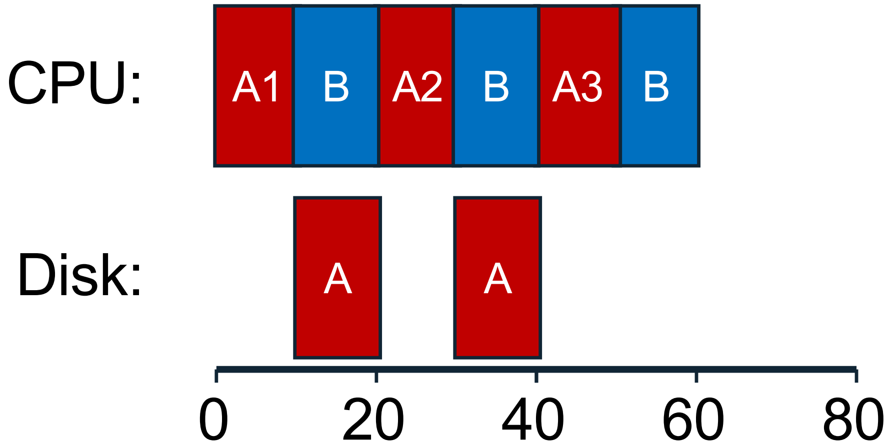
1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. Run-time of each job is known

Not I/O Aware



Don’t let Job A hold on to CPU while blocked waiting for disk

I/O Aware (Overlap)



- Treat Job A as 3 separate CPU bursts
- When Job A completes I/O, another Job A is ready
- Each CPU burst is shorter than Job B, so with SCTF, Job A pre-empts Job B

THE UNIVERSITY of ADELAIDE

Workload Assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. ~~Run-time of each job is known~~

(need smarter, fancier scheduler)

THE UNIVERSITY of ADELAIDE

MLFQ (Multi-Level Feedback Queue)

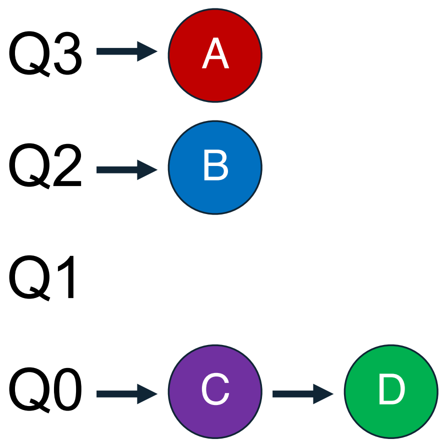
- **Goal: general-purpose scheduling**
- **Must support two job types with distinct goals**
  - "**interactive**" programs care about **response time**
  - "**batch**" programs care about **turnaround time**
- **Approach: multiple levels of round-robin:**
  - each level has higher priority than lower levels and preempts them

THE UNIVERSITY *of* ADELAIDE

Priorities

Rule 1: If priority(A) > Priority(B), A runs

Rule 2: If priority(A) == Priority(B), A & B run in RR



"Multi-level"

How to know how to set priority?

Approach 1: nice
Approach 2: history "feedback"

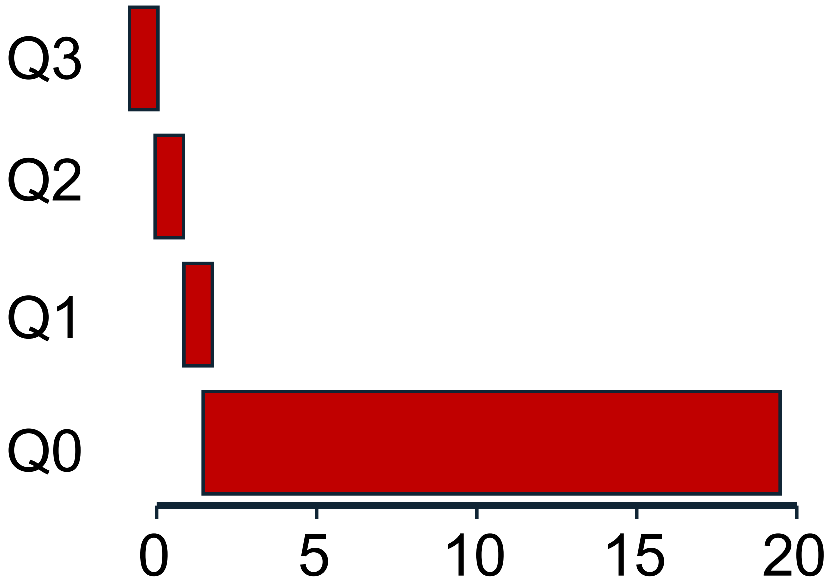
History

- **Use past behaviour of process to predict future behaviour**
  - Common technique in systems
- **Guess how CPU burst (job) will behave based on past CPU bursts (jobs) of this process**
- **Processes alternate between I/O and CPU work**

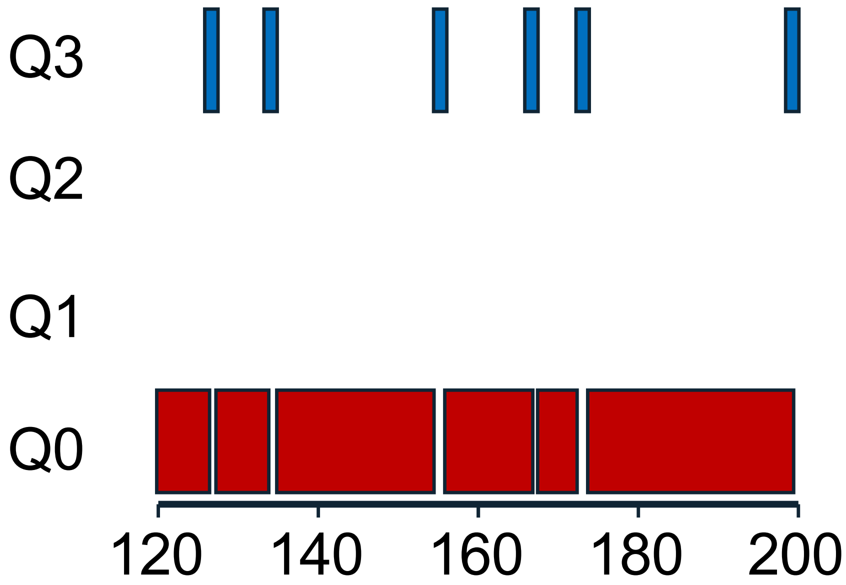
More MLFQ Rules

- **Rule 1: If priority(A) > Priority(B), A runs**
- **Rule 2: If priority(A) == Priority(B), A & B run in RR**
- **More rules:**
  - Rule 3: Processes start at top priority
  - Rule 4: If job uses whole slice, demote process (longer time slices at lower priorities)

One Long Job (Example)

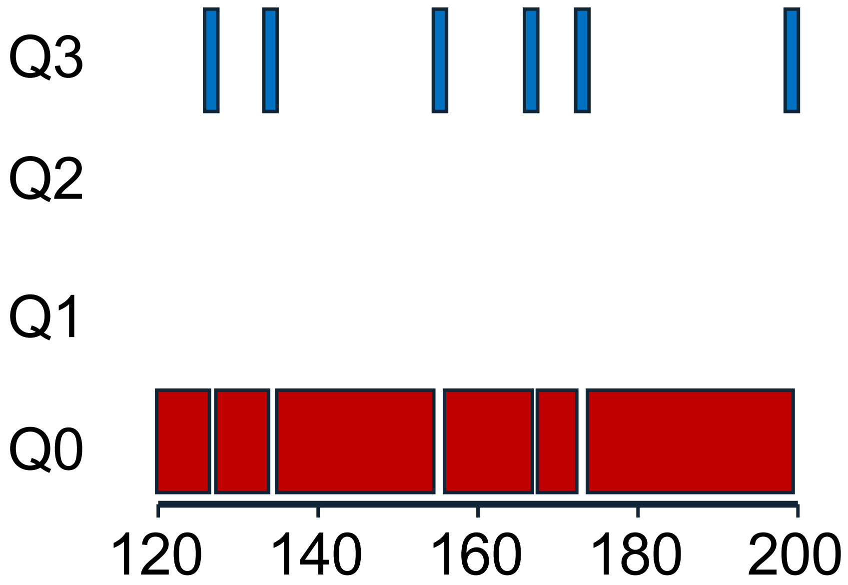


An Interactive Process Joins



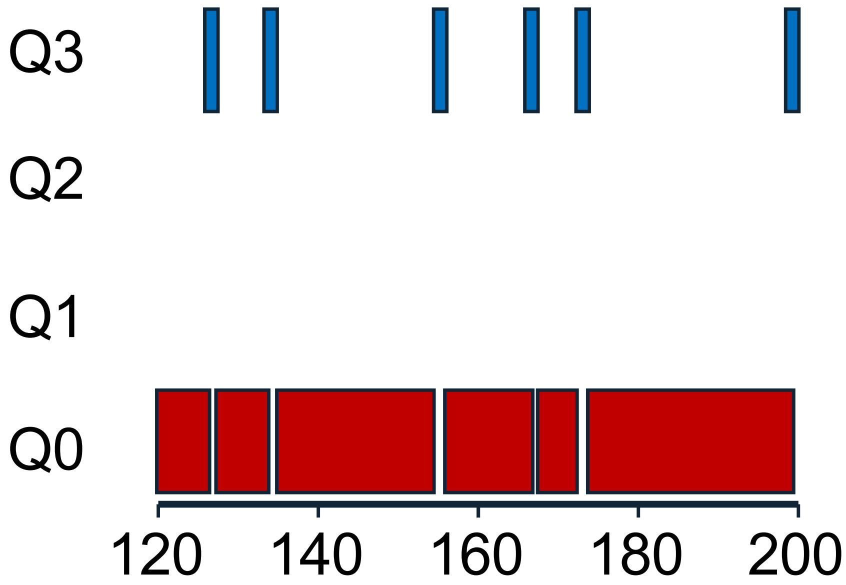
Interactive process never uses entire time slice, so never demoted

Problems with MLFQ?



- **Problems**
  - unforgiving + starvation
  - gaming the system

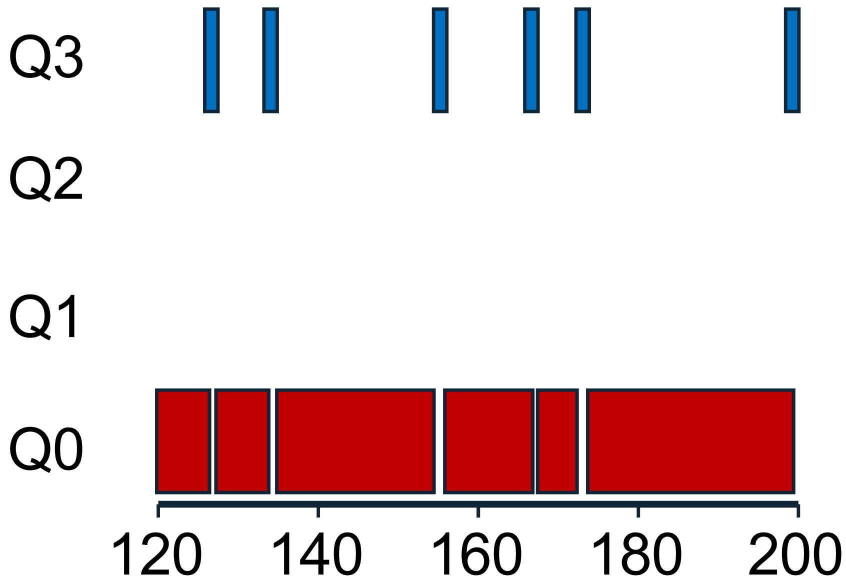
Prevent Starvation



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

THE UNIVERSITY of ADELAIDE

Prevent Gaming



- **Problem: High priority job could trick scheduler and get more CPU by performing I/O right before time-slice ends**

Summary of MLFQ Rules

- **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

Lottery Scheduling

- **Goal: proportional (fair) share**
- **Approach:**
  - give processes lottery tickets
  - whoever wins runs
  - higher priority => more tickets
- **Amazingly simple to implement**

Lottery Code

```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while (current) {
                counter += current->tickets;
                if (counter > winner) break;
                current = current->next;
}
// current is the winner
```

THE UNIVERSITY of ADELAIDE

Lottery Example

```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while (current) {
                counter += current->tickets;
                if (counter > winner) break;
                current = current->next;
}
// current is the winner
```

Who runs if **winner** is:

50

350

0

THE UNIVERSITY of ADELAIDE

Other Lottery Ideas

- Ticket Currencies
- Ticket Transfers
- Ticket Inflation

THE UNIVERSITY of ADELAIDE

Linux's "Completely Fair Scheduler" (CFS)

For now, make these simplifying assumptions:

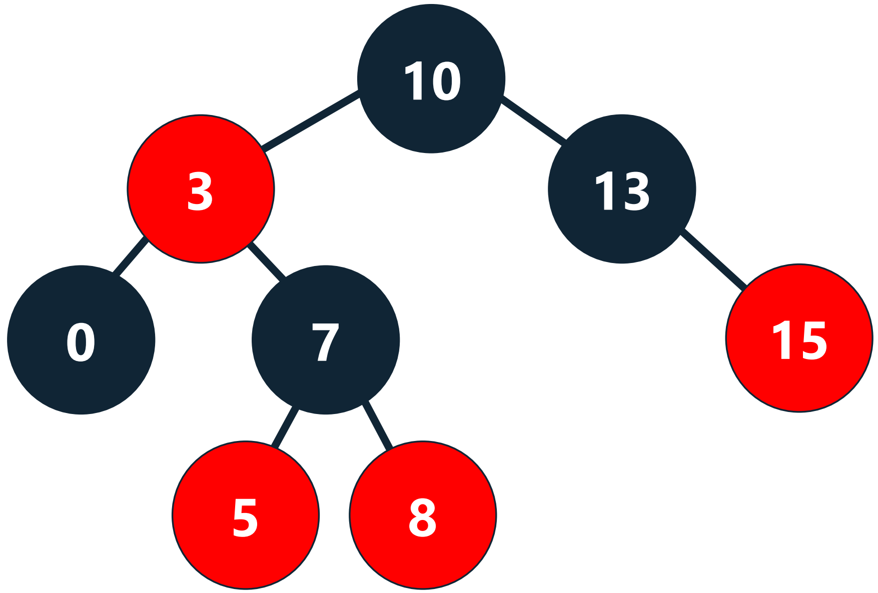
- All tasks have the same priority
- There are always T tasks ready to run at any moment

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.

THE UNIVERSITY *of* ADELAIDE

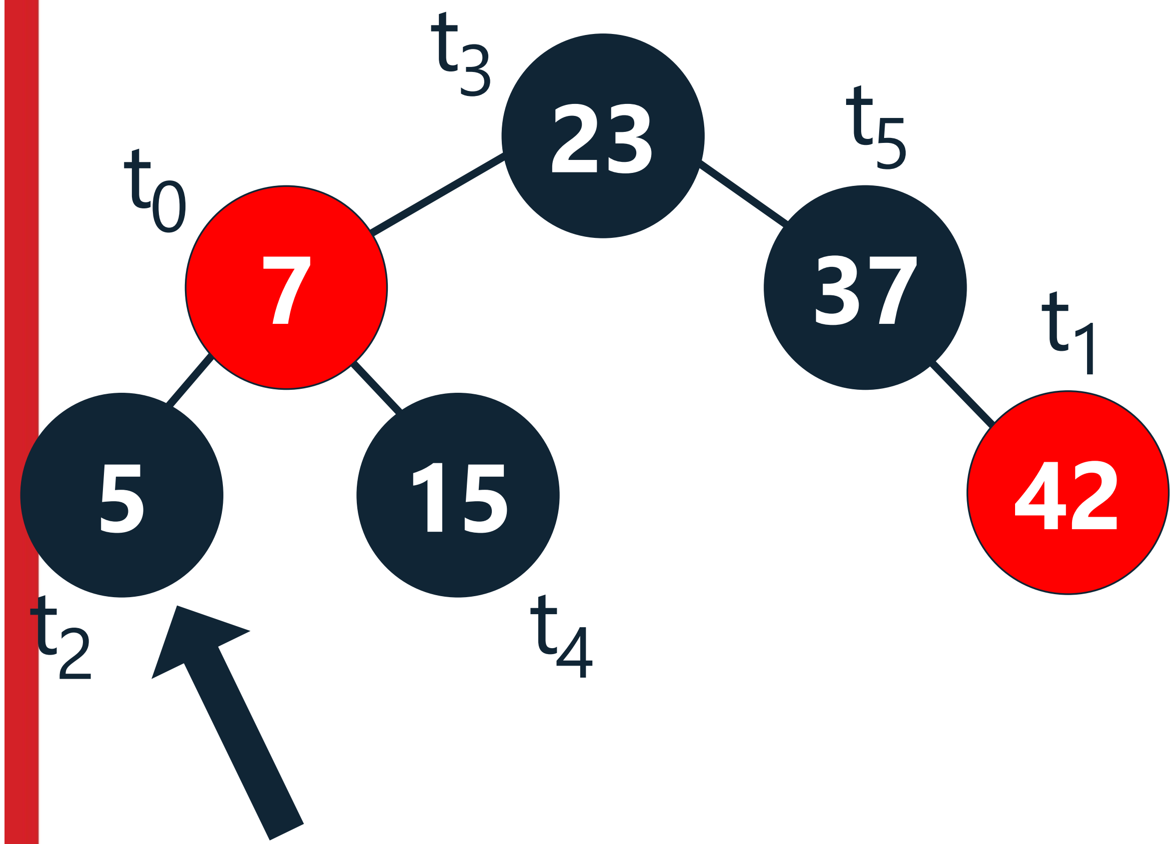
Red-black binary trees



Self-balancing: Insertions and deletions ensure that the longest tree path is at most twice the length of any other path

Guaranteed logarithmic: All insertions, deletions, and searches run in O(log N) time

THE UNIVERSITY of ADELAIDE

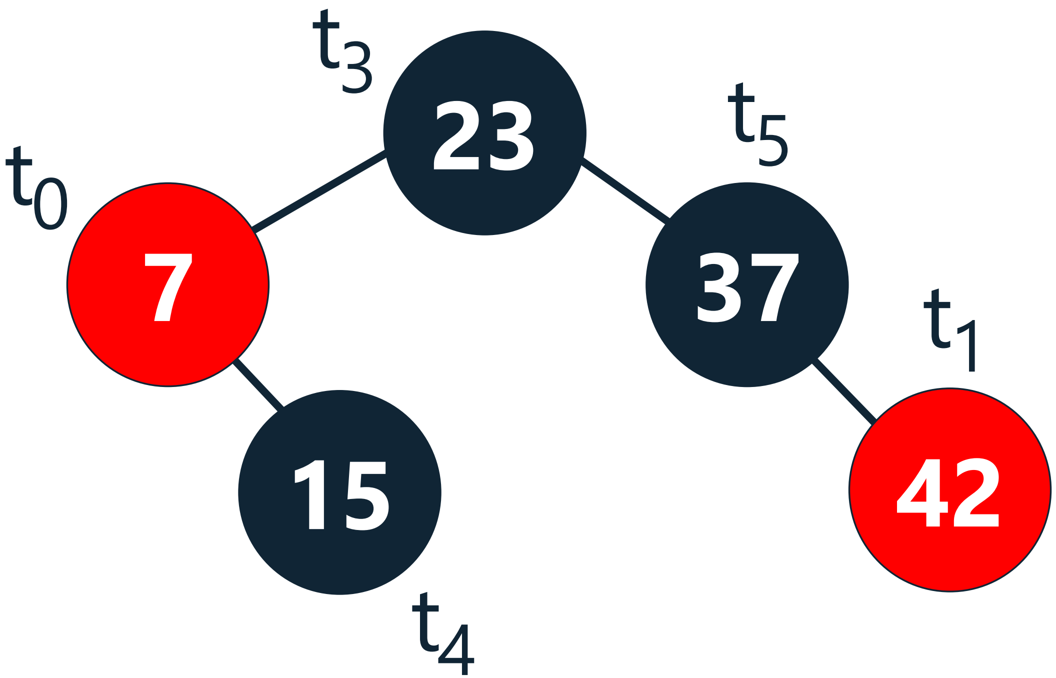


Scheduler picks this task to run, removes it from tree

CFS scheduler

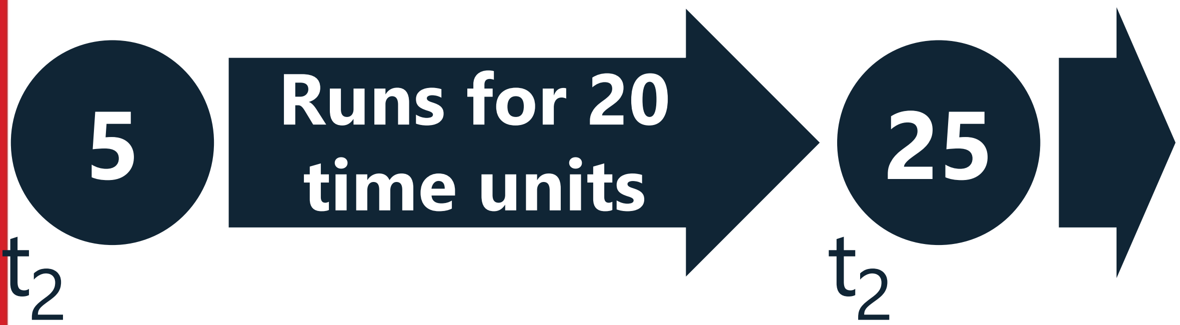
- Associate each task with its elapsed runtime (nanosecond granularity)
- For each core, keep runnable tasks in a red-black tree, with an insertion key of elapsed runtime
  - A newly-created task is tagged with the minimal runtime in the tree
- Next task to run is just the left-most task in tree!

THE UNIVERSITY of ADELAIDE



Timer interrupt fires, scheduler runs

- Now, t_2 no longer has the smallest elapsed runtime
- So, scheduler reinserts t_2 into the tree and runs t_0!



THE UNIVERSITY of ADELAIDE

Classic CFS Example

Suppose there are two tasks:

- Video rendering application (CPU-intensive, long-running, non-interactive)
- Word processor (interactive, only uses CPU for bursts)

Both tasks start with an elapsed runtime of 0

- Video rendering task quickly accumulates runtime . . .
- . . . but word processor's runtime stays low (task is mainly blocked on IO)

So, whenever word processor receives keyboard/mouse input and wakes up, it will be the left-most task, and immediately get scheduled

It’s Nice To Be Nice

nice(1) - Linux man page

Name

nice - run a program w

Synopsis

nice [*OPTION*] [*COMM*

Description

Run COMMAND with ar
current niceness. Nicer

-n, **--adjustment**=*N*
add integer N to t
--help
display this help a
--version
output version inf

NOTE: your shell may
Please refer to your she

nice(2) - Linux man page

Name

nice - change process priority

Synopsis

#include <unistd.h>

int nice(int *inc***);**

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

nice(): _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE

Description

nice() adds *inc* to the nice value for the calling process. (A higher nice value means a low priority.) Only the superuser may specify a negative increment, or priority increase. The range for nice values is described in **getpriority**(2).

Return Value

On success, the new nice value is returned (but see NOTES below). On error, -1 is returned, and *errno* is set appropriately.

Task Priorities in CFS

```
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

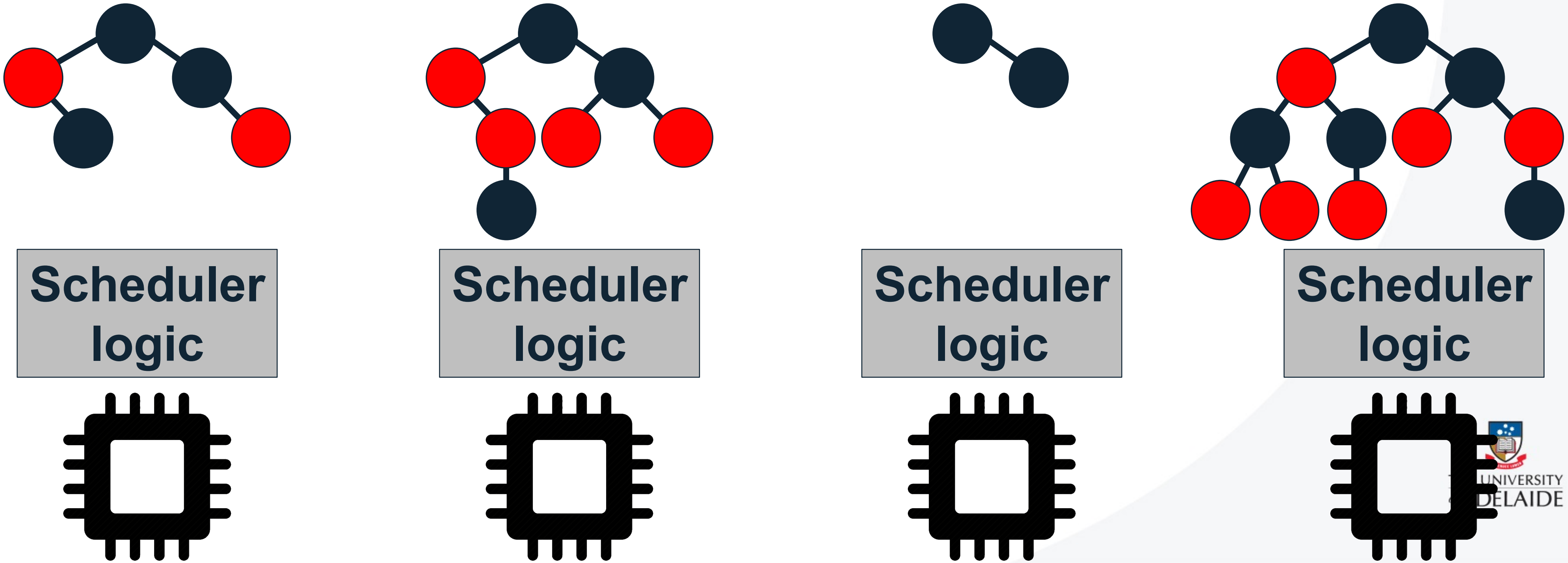
Linux scheduler now: CFS

Uses a red-black tree per core, to avoid cross-core lock contention

However, cross-core load balancing is needed to ensure:

All cores are highly-utilized

All of the high-priority tasks don't end up on a small number of cores



Summary

- Understand goals (metrics) and workload, then design scheduler around that
- General purpose schedulers need to support processes with different goals
- Past behaviour is good predictor of future behaviour
- Random algorithms (lottery scheduling) can be simple to implement, and avoid corner cases.

THE UNIVERSITY *of* ADELAIDE

Address space and translation

make history.

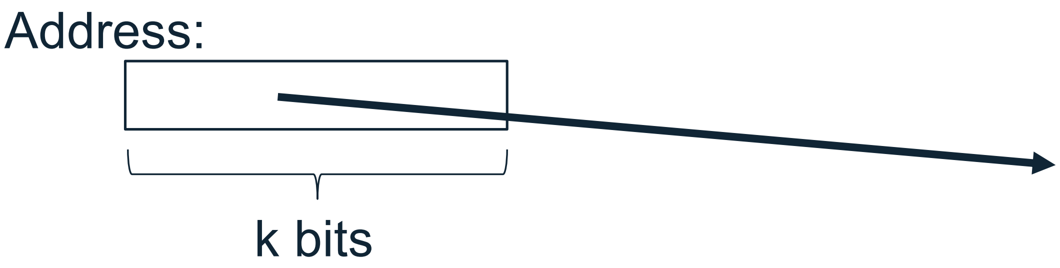
THE UNIVERSITY of ADELAIDE

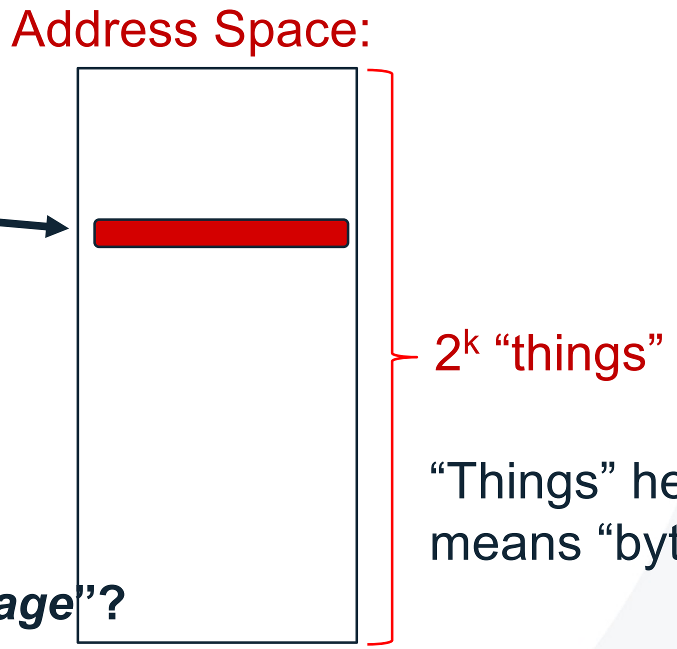
Introduction

Questions answered in this lecture:

- **What is in the address space of a process (review)?**
- **What are the different ways that that OS can virtualize memory?**
  - Time sharing, static relocation, dynamic relocation
  - (base, base + bounds, segmentation)
- **What hardware support is needed for dynamic relocation?**

THE BASICS: Address/Address Space





“Things” here usually means “bytes” (8 bits)

What is 2^{10} bytes (where a byte is “B”)?

- 2^{10} B = 1024B = 1 KiB (1Ki = 1024, *not* 1000)

How many bits to address each byte of 4KiB “*page*”?

- 4KiB = 4×1KiB = $4 \times 2^{10} = 2^{12} \Rightarrow$ 12 bits

How much memory can be addressed with 20 bits? 32 bits? 64 bits?

- Use 2^k

THE UNIVERSITY of ADELAIDE

Address Space, Process Virtual Address Space

Definition: Set of accessible addresses and the state associated with them

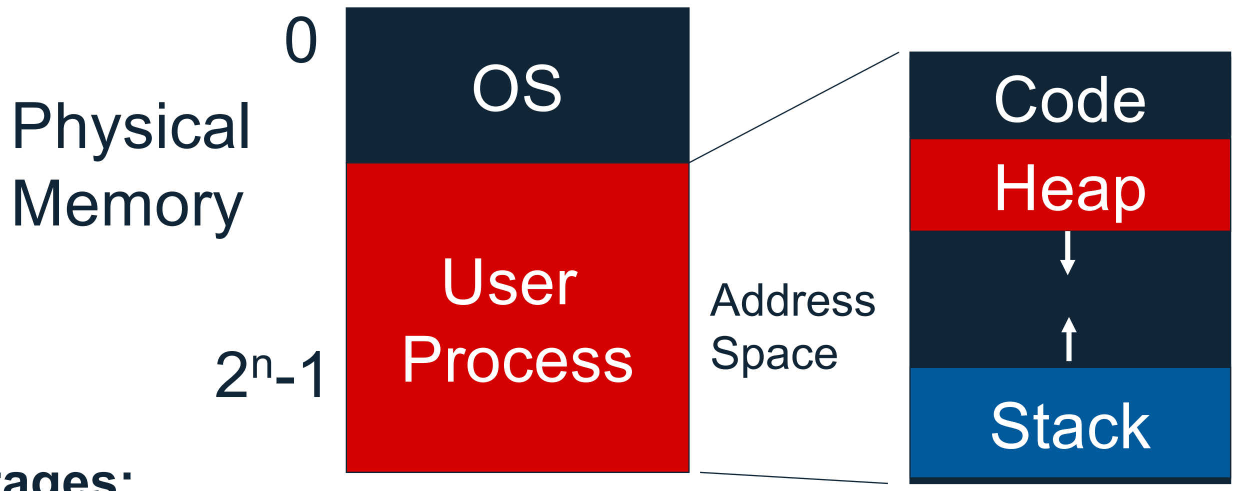
- 2^{32} = ~4 billion ***bytes*** on a 32-bit machine

How many 32-bit numbers fit in this address space?

- A 32-bit register can store 2^{32} different values.
- 32-bits can hold unsigned values: 0 through 4,294,967,295
- ~4 GiB of byte-addressable memory.

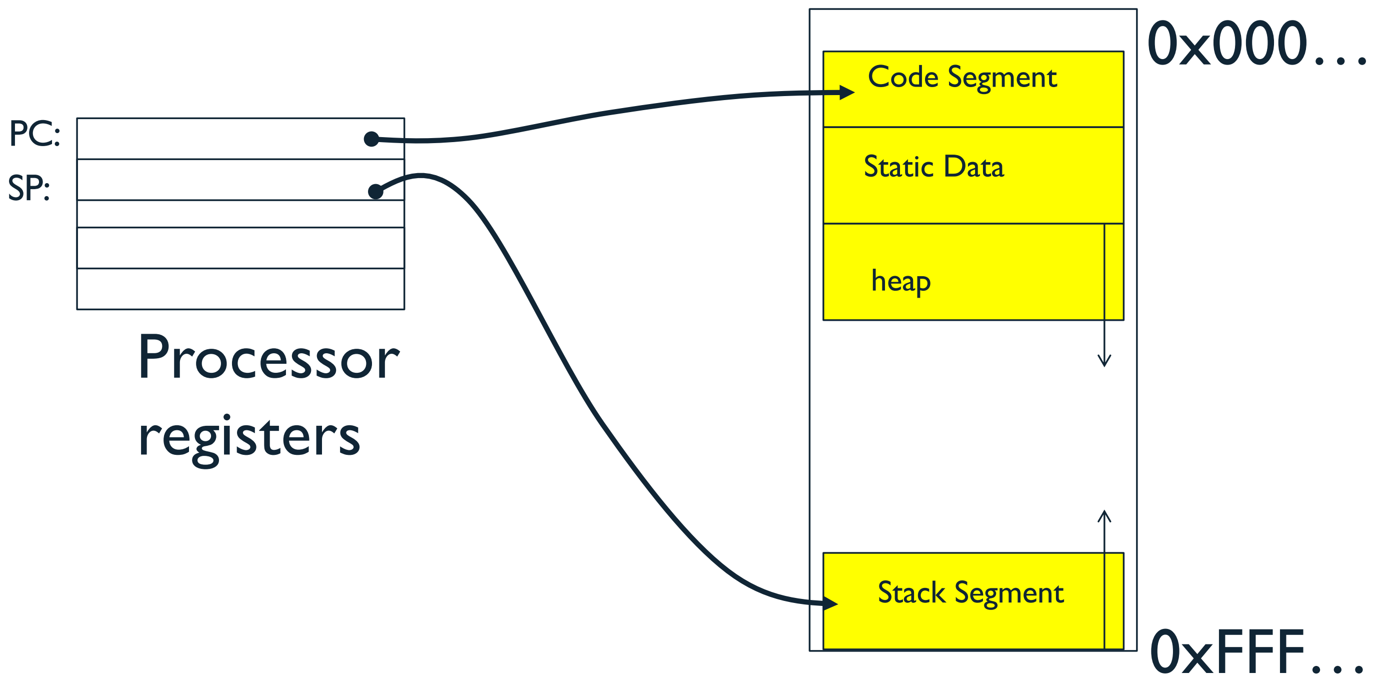
Motivation for Virtualization

- **Uniprogramming: One process runs at a time**



- **Disadvantages:**
  - Only one process runs at a time
  - Process can destroy OS

Process Address Space: typical structure



THE UNIVERSITY of ADELAIDE

Motivation for Dynamic Memory

- **Why do processes need dynamic allocation of memory?**
  - Do not know amount of memory needed at compile time
  - Must be pessimistic when allocate memory statically
  - Allocate enough for worst possible case; Storage is used inefficiently
- **Recursive procedures**
  - Do not know how many times procedure will be nested

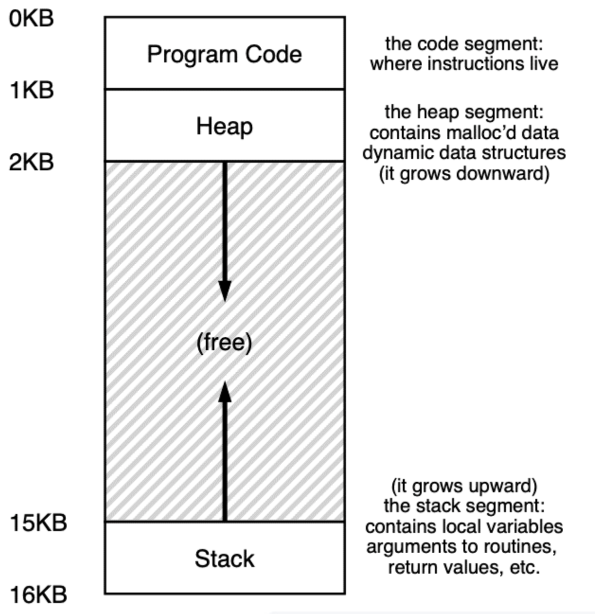
THE UNIVERSITY of ADELAIDE

Motivation for Dynamic Memory

- **Complex data structures: lists and trees**

```
struct my_t *p = (struct my_t *)malloc(sizeof(struct my_t));
```

- **Two types of dynamic allocation**
  - Stack
  - Heap



Stack Organization

- Simple and efficient implementation:
- Pointer separates allocated and freed space
- Allocate: Increment pointer
- Free: Decrement pointer
- No fragmentation

Where are stacks Used?

OS uses stack for procedure call frames (local variables and parameters)

```
main () {
      int A = 0;
      foo (A);
      printf("A: %d\n", A);
}
void foo (int Z) {
      int A = 2;
      Z = 5;
      printf("A: %d Z: %d\n", A, Z);
}
```

THE UNIVERSITY of ADELAIDE

Heap Organization

- **Definition**
  - Allocate from any random location: malloc(), new()
  - Heap memory consists of allocated areas and free areas (holes)
  - Order of allocation and free is unpredictable

Heap Organization

- **Advantage**
  - Works for all data structures
- **Disadvantages**
  - Allocation can be slow
  - End up with small chunks of free space - fragmentation
  - Where allocate 12 bytes? 16 bytes? 24 bytes??
- **What is OS's role in managing heap?**
  - OS gives big chunk of free memory to process; library manages individual allocations

THE UNIVERSITY of ADELAIDE

Match that Address Location

```
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof(int));); 
}
```

Possible segments: static data, code, stack, heap

What if no static data segment?

Address	Location
main	
y	
z	
*z	

THE UNIVERSITY of ADELAIDE

Match that Address Location

```
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof(int));) ;
}
```

Possible segments: static data, code, stack, heap

What if no static data segment?

Address	Location
main	**Code**
y	**Stack**
z	**Stack**
*z	**Heap**

THE UNIVERSITY of ADELAIDE

How to Virtualize Memory?

- **Problem: How to run multiple processes simultaneously?**
- **Addresses are “hardcoded” into process binaries**
- **How to avoid collisions?**
- **Possible Solutions for Mechanisms:**

1. **Time Sharing**
2. Static Relocation
3. Base
4. Base+Bounds
5. Segmentation

THE UNIVERSITY of ADELAIDE

Multiprogramming Goals

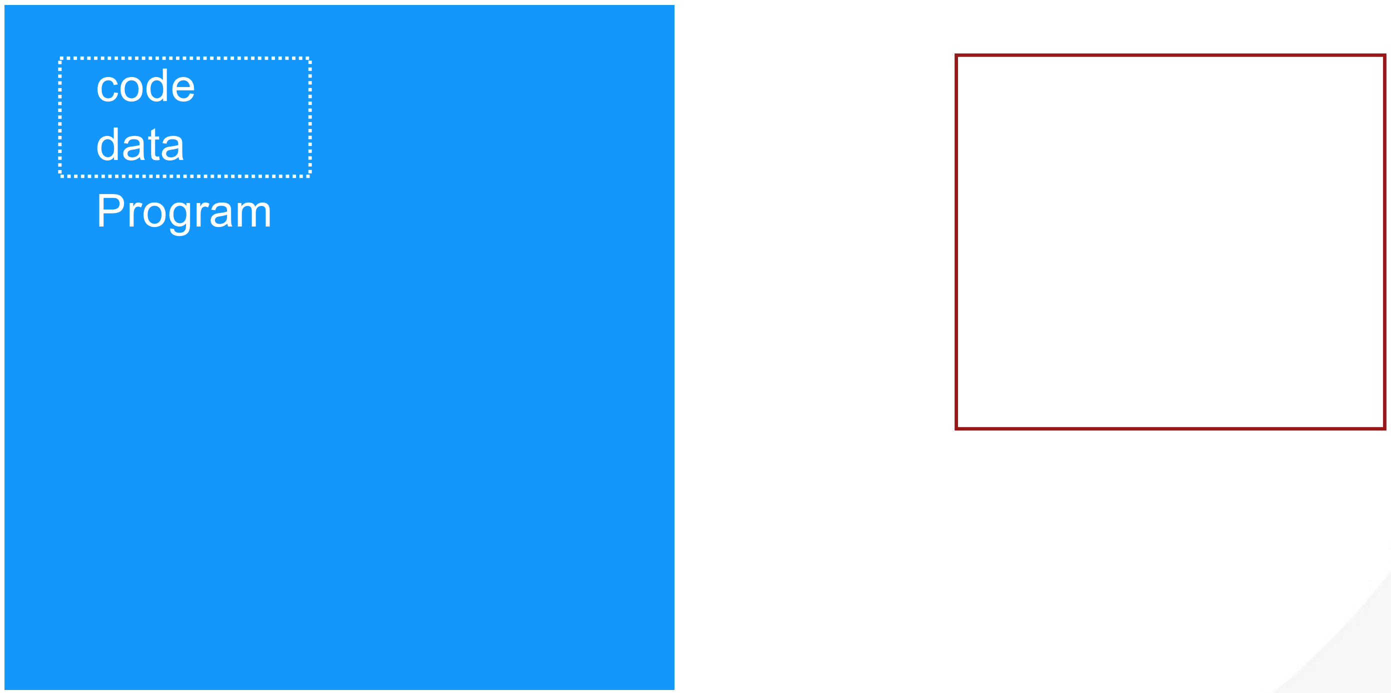
- **Transparency:** Processes are not aware that memory is shared
- **Protection:** Cannot corrupt OS or other processes
- **Privacy:** Cannot read data of other processes
- **Efficiency:** Do not waste memory resources (minimize fragmentation)
- **Sharing:** Cooperating processes can share portions of address space

1) Time Sharing of Memory

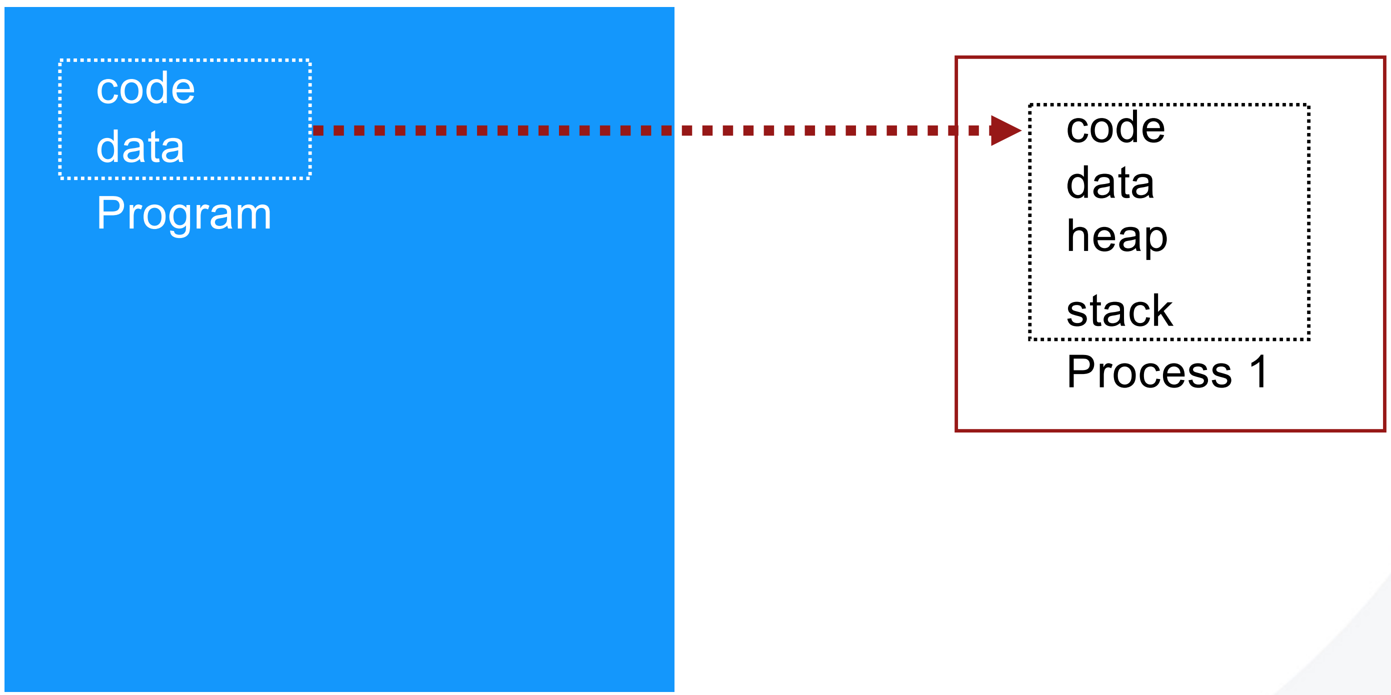
- **Try similar approach to how OS virtualizes CPU**
- **Observation:**
  - OS gives illusion of many virtual CPUs by saving CPU registers to memory when a process isn't running
  - Could give illusion of many virtual memories by saving memory to disk when process isn't running

THE UNIVERSITY of ADELAIDE

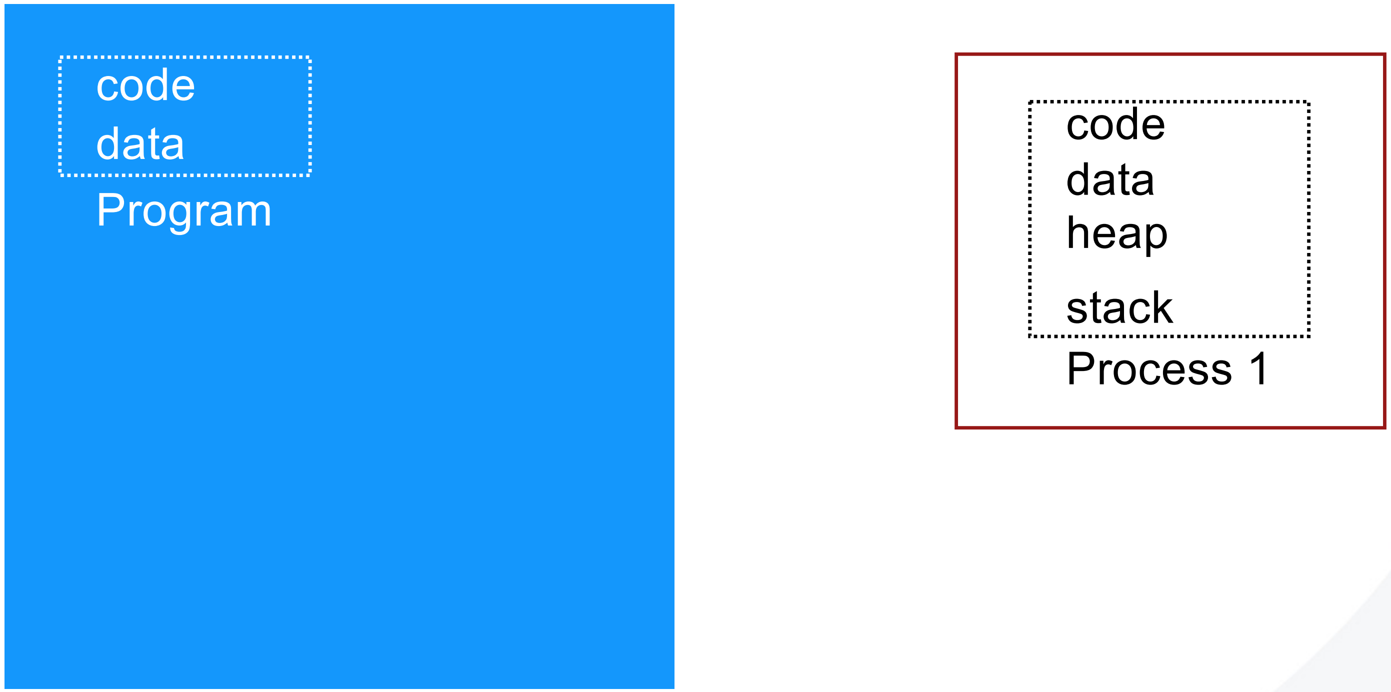
Time Share Memory: Example



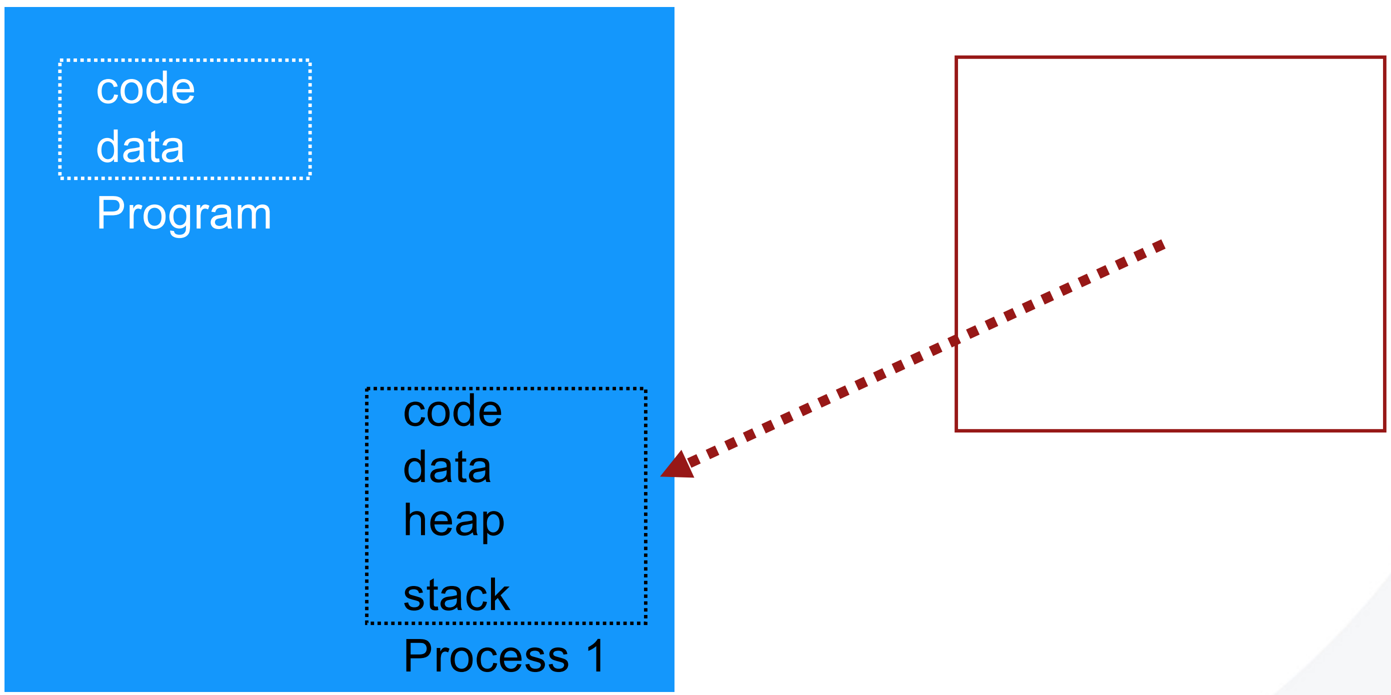
Time Share Memory: Example



Time Share Memory: Example

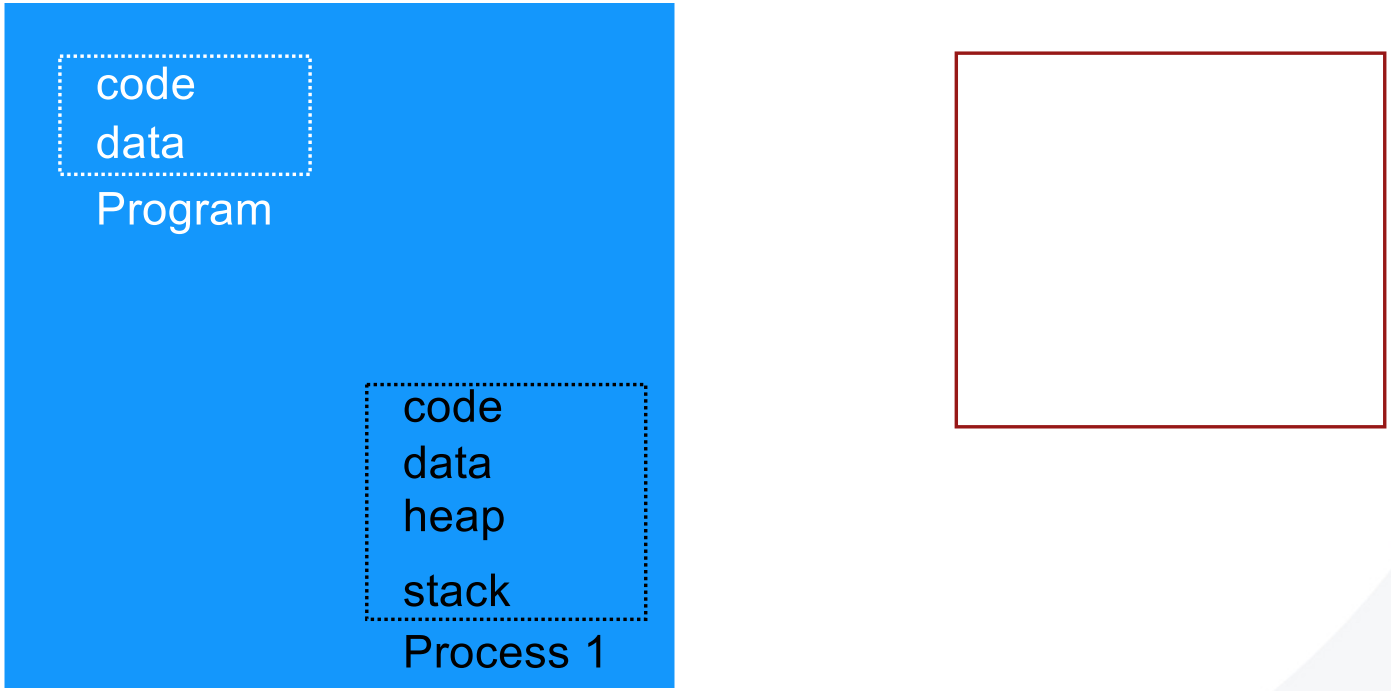


Time Share Memory: Example



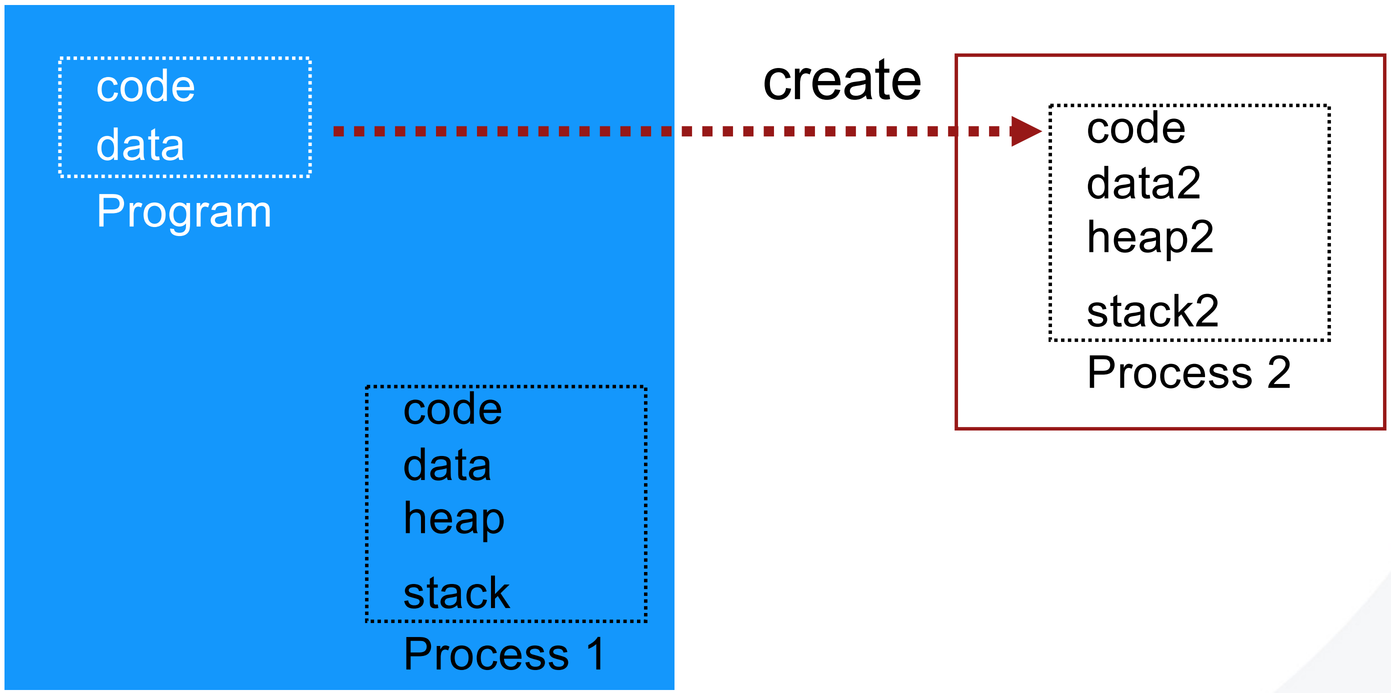
THE UNIVERSITY of ADELAIDE

Time Share Memory: Example

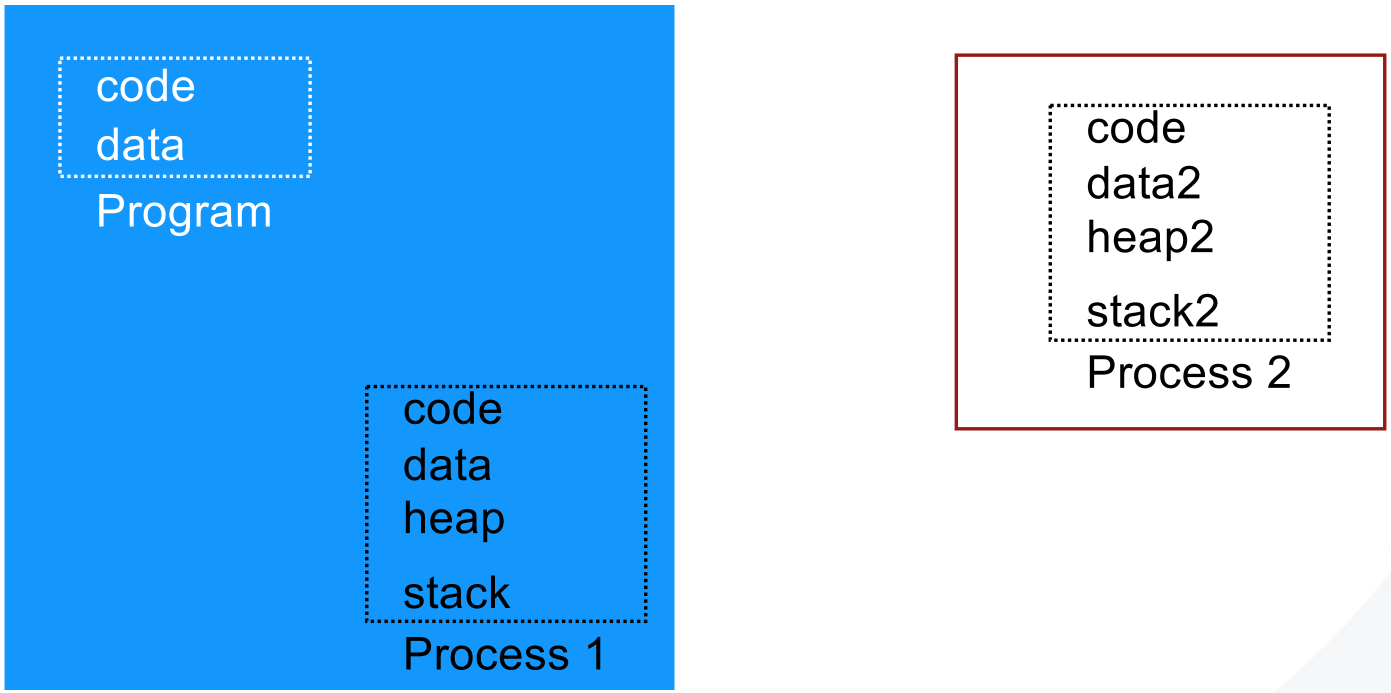


THE UNIVERSITY of ADELAIDE

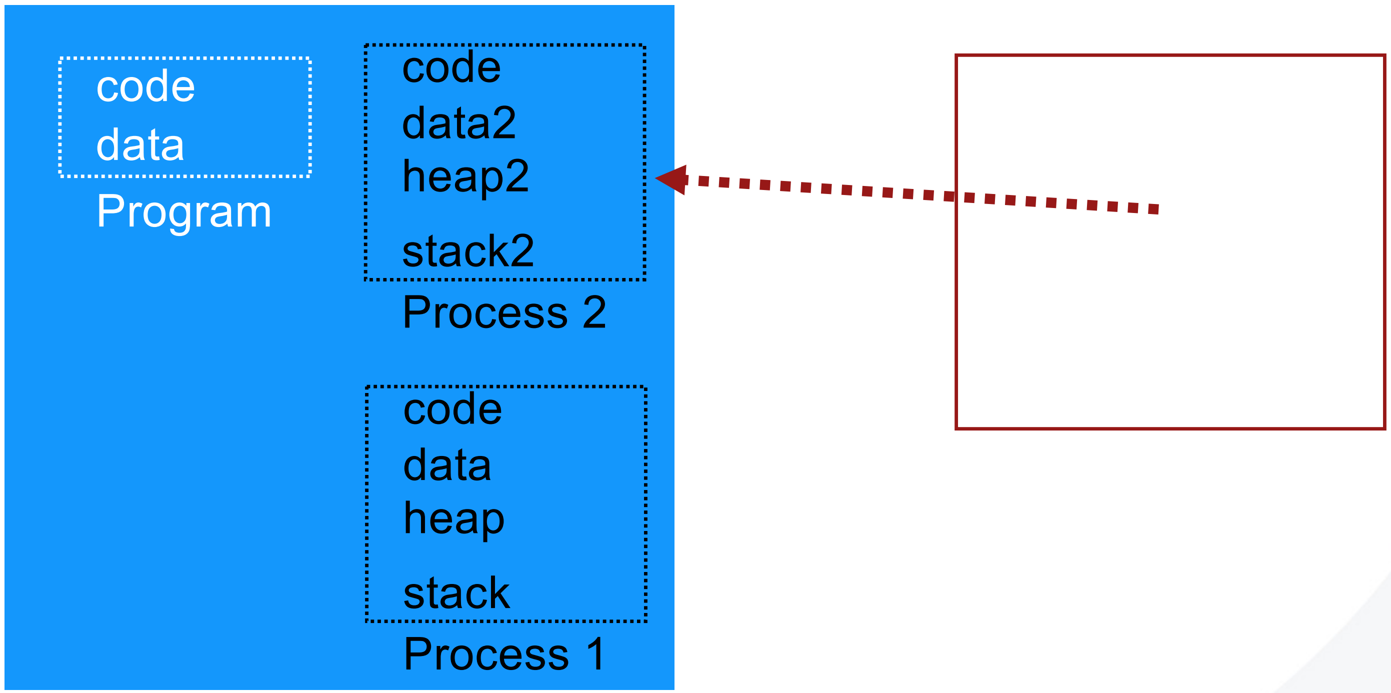
Time Share Memory: Example



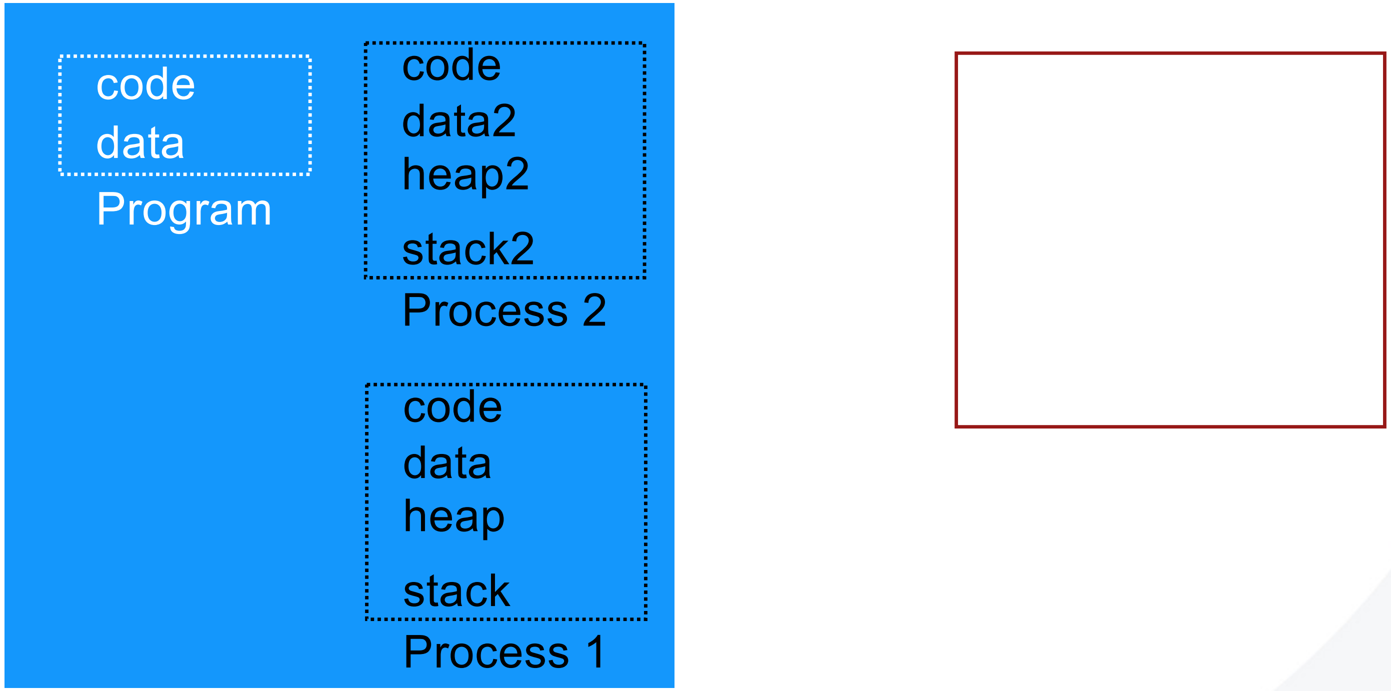
Time Share Memory: Example



Time Share Memory: Example

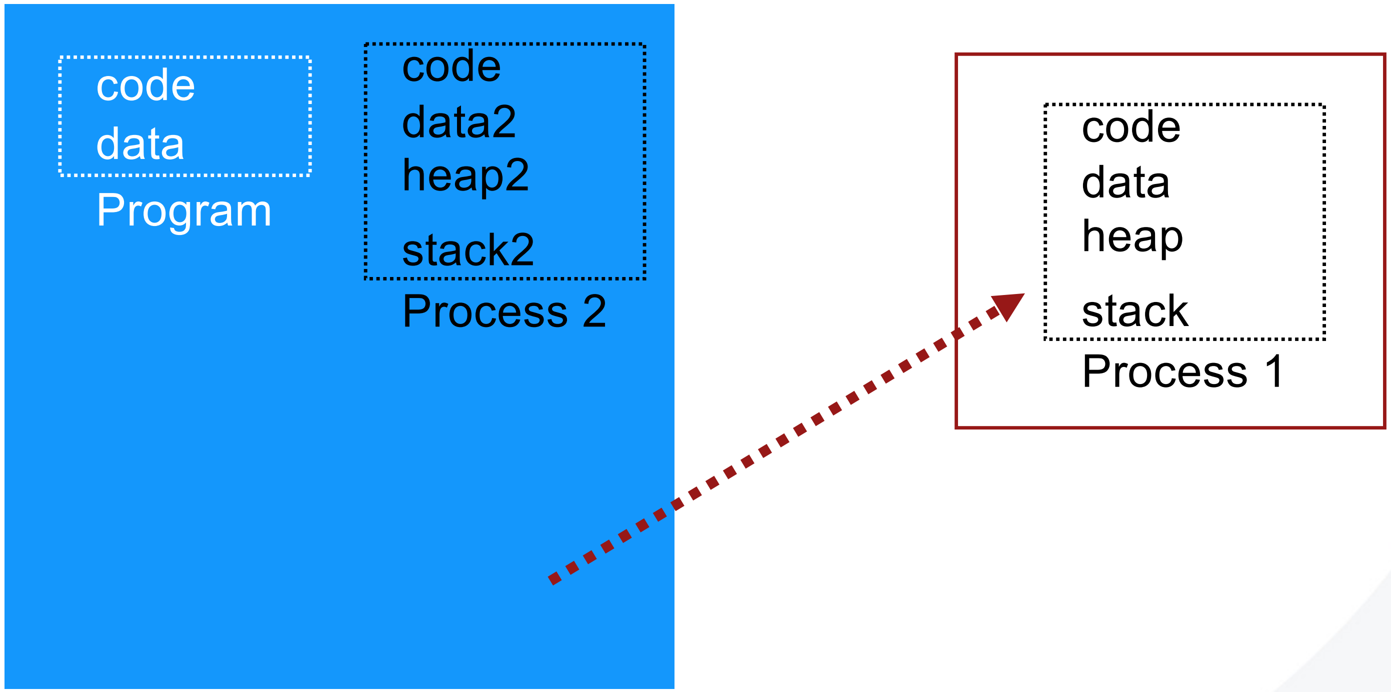


Time Share Memory: Example

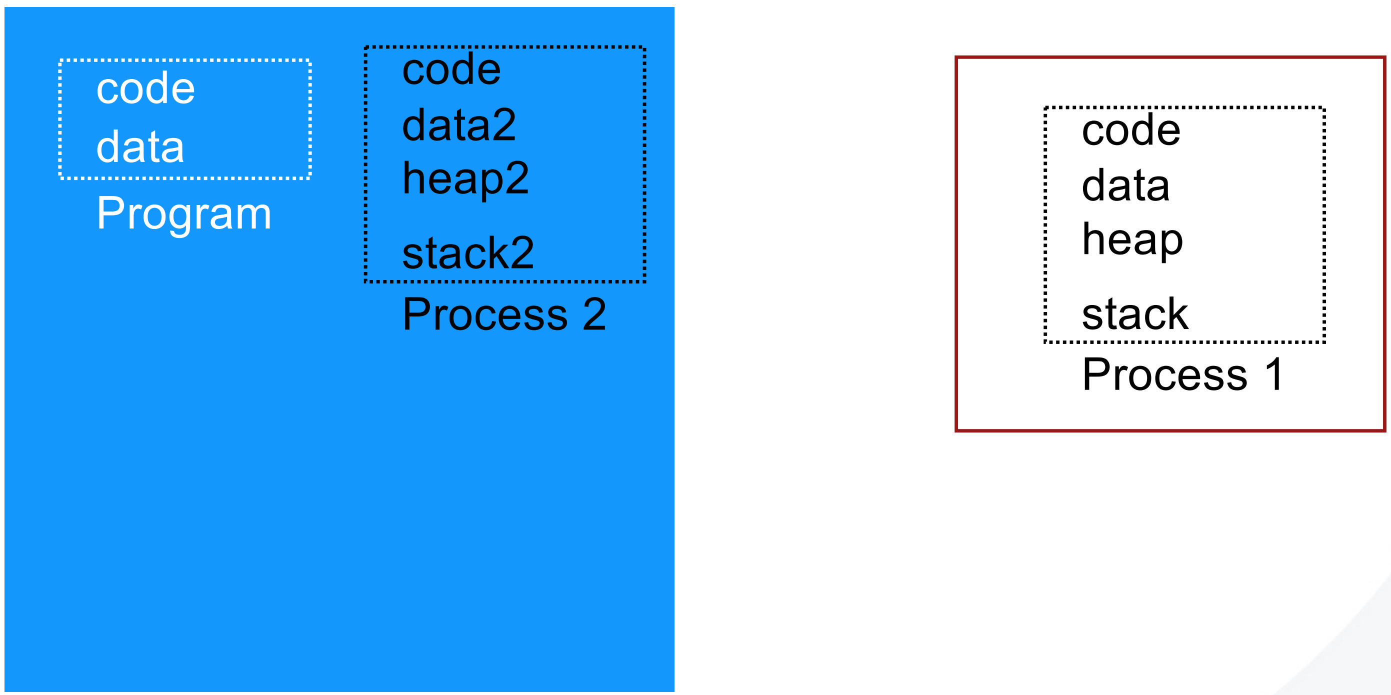


THE UNIVERSITY of ADELAIDE

Time Share Memory: Example



Time Share Memory: Example



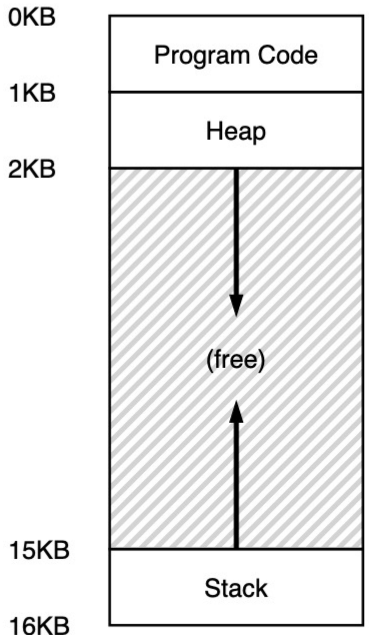
THE UNIVERSITY of ADELAIDE

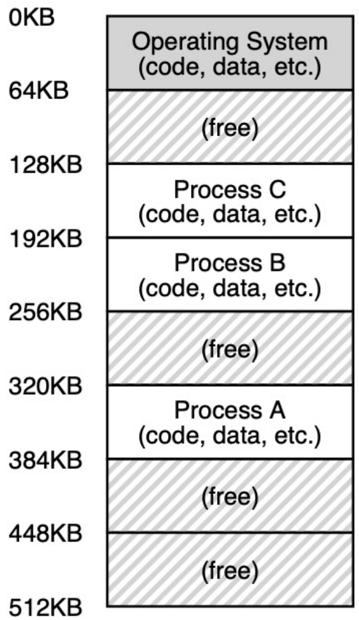
Problems with Time Sharing Memory

- **Problem: Ridiculously poor performance**
- **Better Alternative: space sharing**
  - At same time, space of memory is divided across processes
- **Remainder of solutions all use space sharing**

Abstraction: Address Space

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





Memory Accesses

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int x;
  x = x + 3;
}
```

```
0x10:  movl   0x8(%rbp), %edi
0x13:  addl   $0x3, %edi
0x19:  movl   %edi, 0x8(%rbp)
```

%**rbp** is the base pointer:
points to base of current stack frame

THE UNIVERSITY of ADELAIDE

Static Relocation

- **Idea: OS rewrites each program before loading it as a process in memory**
- **Each rewrite for different process uses different addresses and pointers**
- **Change jumps, loads of static data**

```
0x1010: movl  0x8(%rbp), %edi
0x1013: addl  $0x3, %edi
0x1019: movl  %edi, 0x8(%rbp)
```

rewrite

- `0x10: movl  0x8(%rbp), %edi`
- `0x13: addl  $0x3, %edi`
- `0x19: movl  %edi, 0x8(%rbp)`

rewrite

```
0x3010: movl  0x8(%rbp), %edi
0x3013: addl  $0x3, %edi
0x3019: movl  %edi, 0x8(%rbp)
```

THE UNIVERSITY of ADELAIDE

Static Relocation: Disadvantages

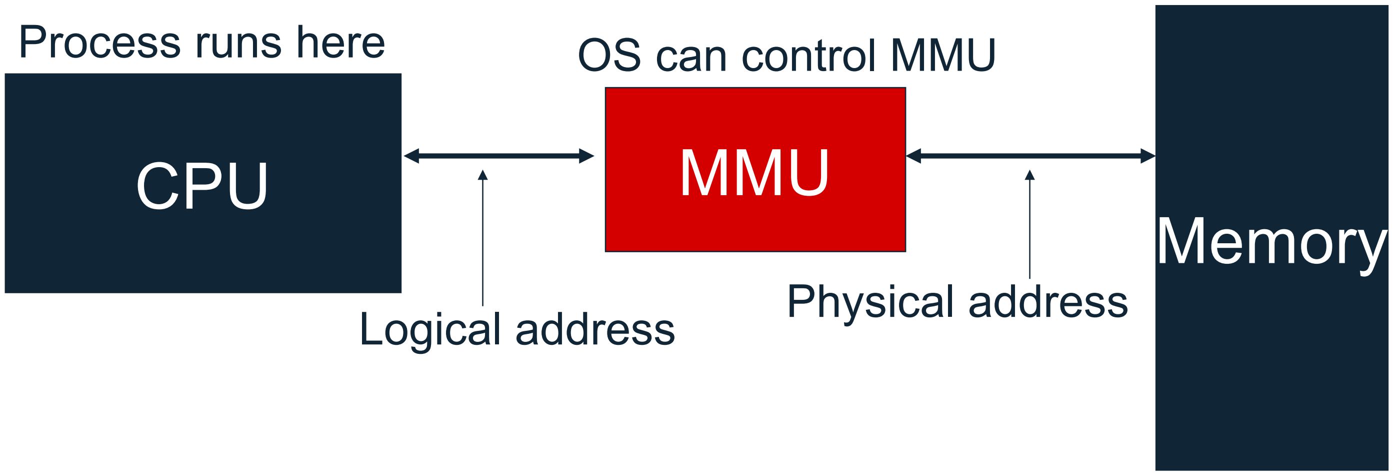
- **No protection**
  - Process can destroy OS or other processes
  - No privacy
- **Cannot move address space after it has been placed**
  - May not be able to allocate new process

THE UNIVERSITY of ADELAIDE

3) Dynamic Relocation

- **Goal: Protect processes from one another**
- **Requires hardware support**
  - Memory Management Unit (MMU)
- **MMU dynamically changes process address at every memory reference**
  - Process generates logical or virtual addresses (in their address space)
  - Memory hardware uses physical or real addresses

3) Dynamic Relocation



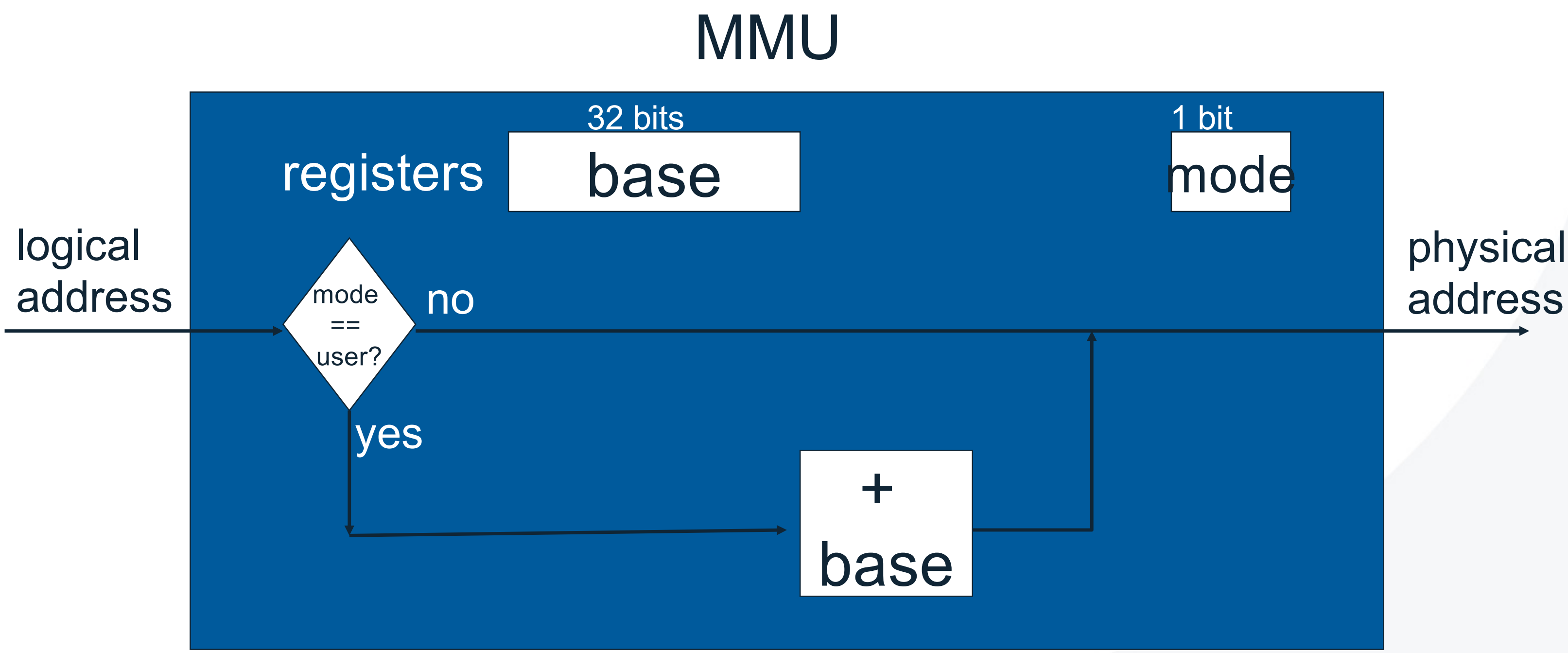
Hardware Support for Dynamic Relocation

Two operating modes

- **Privileged (protected, kernel) mode: OS runs**
  - When enter OS (trap, system calls, interrupts, exceptions)
  - Allows certain instructions to be executed
  - Allows OS to access all of physical memory
- **User mode: User processes run**
  - Perform translation of logical address to physical address
- **Minimal MMU contains base register for translation**
  - base: start location for address space

Implementation of Dynamic Relocation: BASE REG

- **Translation on every memory access of user process**
  - MMU adds base register to logical address to form physical address

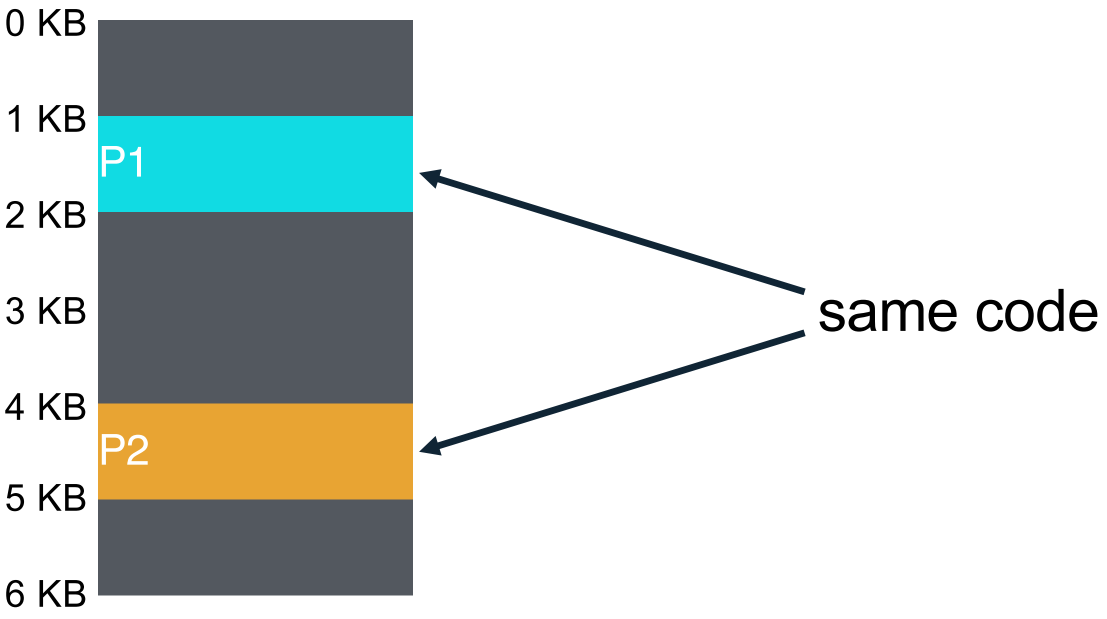


Dynamic Relocation with Base Register

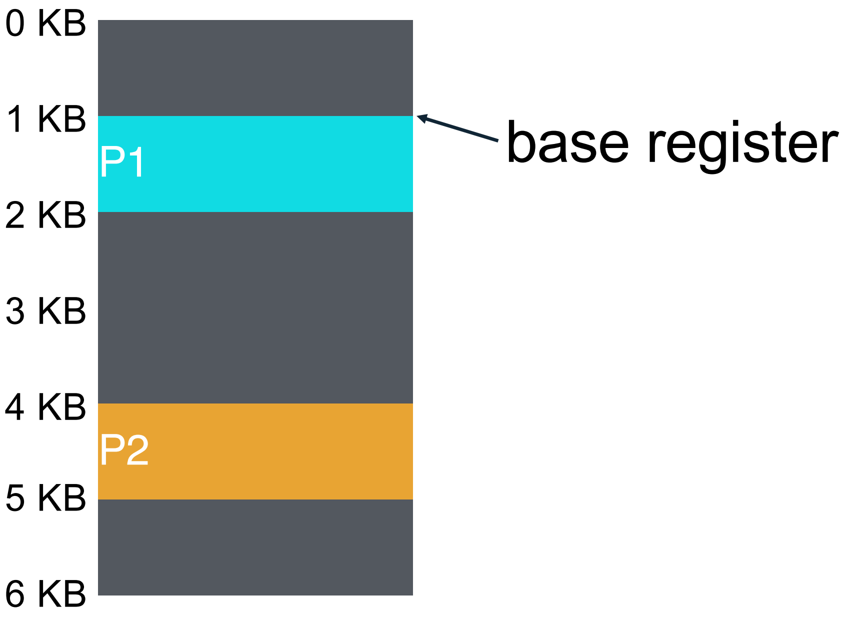
- Idea: translate virtual addresses to physical by adding a fixed offset each time
- Store offset in base register
- Each process has different value in base register
- Dynamic relocation by changing the value of the base register!

THE UNIVERSITY *of* ADELAIDE

Visual example of Dynamic Relocation: BASE REGISTER



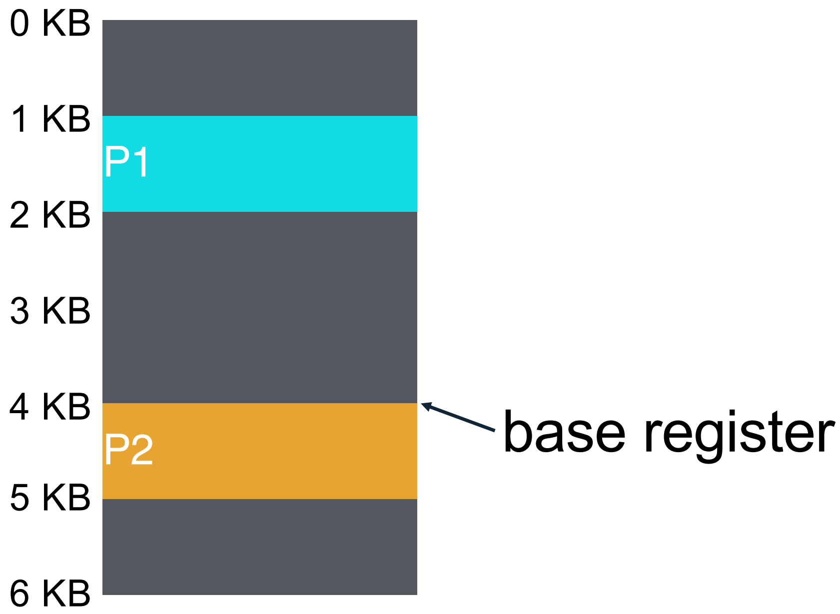
Visual example of Dynamic Relocation: BASE REGISTER



P1 is running

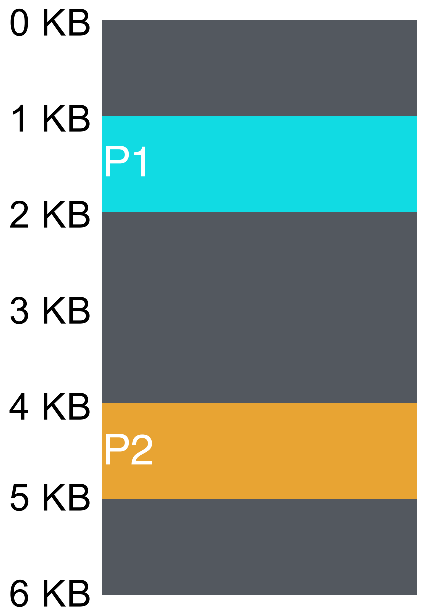
THE UNIVERSITY of ADELAIDE

Visual example of Dynamic Relocation: BASE REGISTER



P2 is running

Visual example of Dynamic Relocation: BASE REGISTER



(Decimal notation)

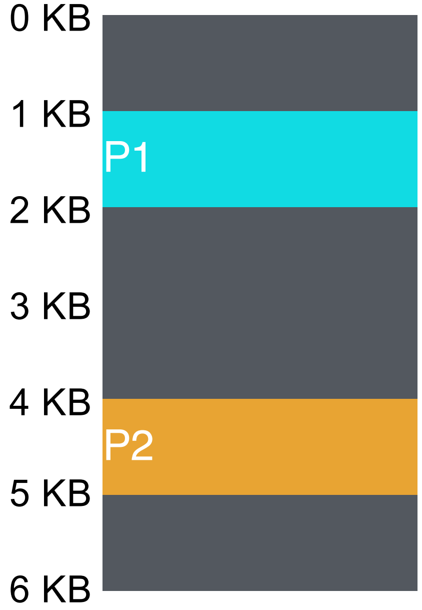
Virtual	Physical
P1: load 100, R1	

Visual example of Dynamic Relocation: BASE REGISTER

(Decimal notation)

Virtual	Physical	
P1: load 100, R1	load 1124, R1	(1024 + 100)

Visual example of Dynamic Relocation: BASE REGISTER

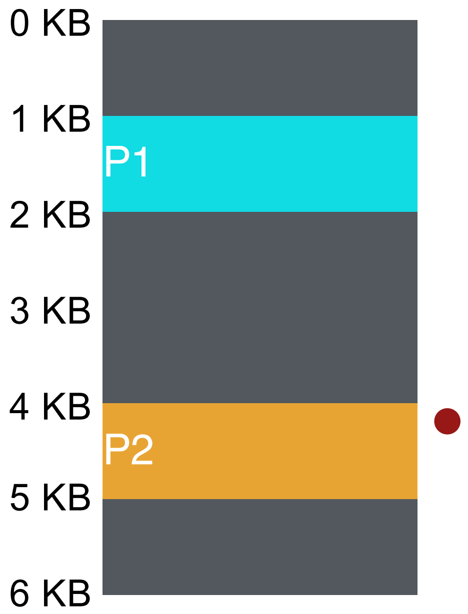


(Decimal notation)

Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	

THE UNIVERSITY of ADELAIDE

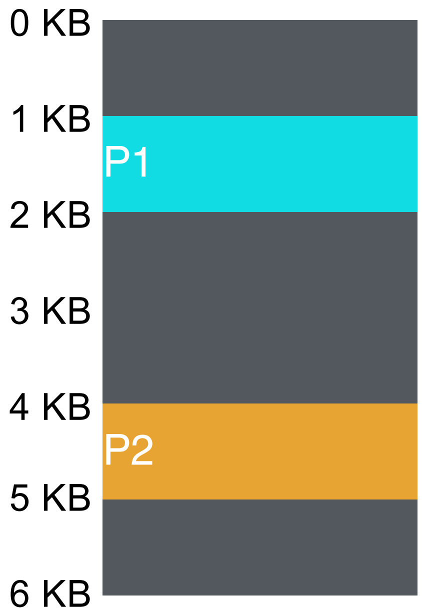
Visual example of Dynamic Relocation: BASE REGISTER



(Decimal notation)

Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 100, R1	load 4196, R1	(4096 + 100)

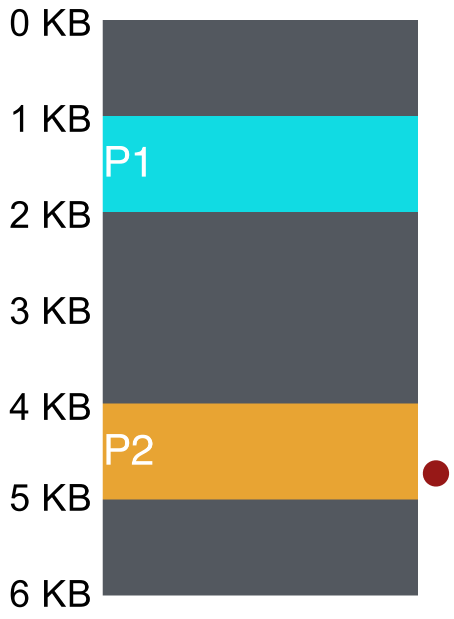
Visual example of Dynamic Relocation: BASE REGISTER



(Decimal notation)

Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	

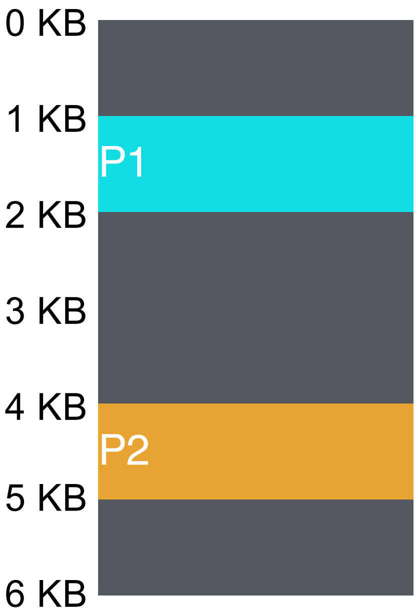
Visual example of Dynamic Relocation: BASE REGISTER



(Decimal notation)

Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1

Visual example of Dynamic Relocation: BASE REGISTER

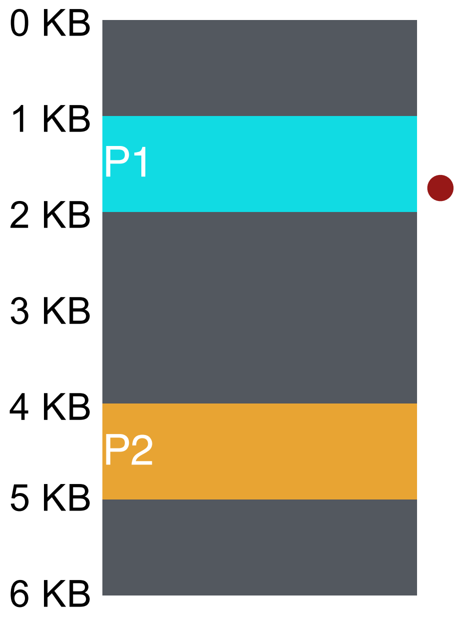


(Decimal notation)

Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 1000, R1	

THE UNIVERSITY of ADELAIDE

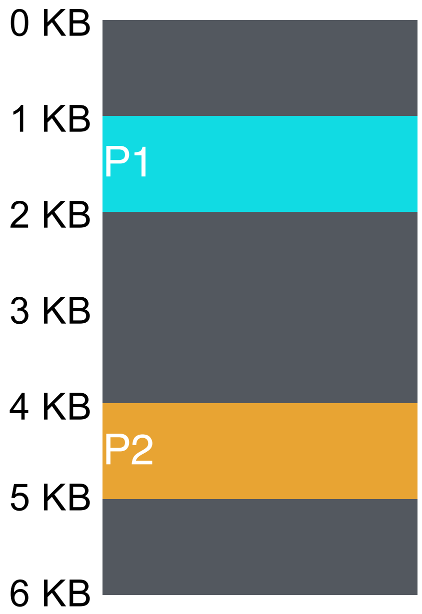
Visual example of Dynamic Relocation: BASE REGISTER



(Decimal notation)

Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 1000, R1	load 2024, R1

Dynamic Relocation: BASE REGISTER



(Decimal notation)

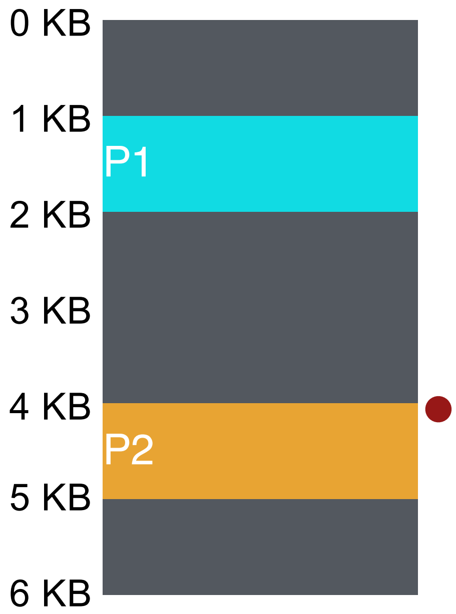
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 1000, R1	load 2024, R1

Can P2 hurt P1?
Can P1 hurt P2?

How well does dynamic relocation do with base register for protection?

THE UNIVERSITY *of* ADELAIDE

Dynamic Relocation: BASE REGISTER



(Decimal notation)

Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 100, R1	load 4196, R1	
P2: load 1000, R1	load 5096, R1	
P1: load 1000, R1	load 2024, R1	
P1: store 3072, R1	store 4096, R1	(3072 + 1024)

Can P2 hurt P1?

Can P1 hurt P2?

How well does dynamic relocation do with base register for protection?

THE UNIVERSITY of ADELAIDE

4) Dynamic with Base+Bounds

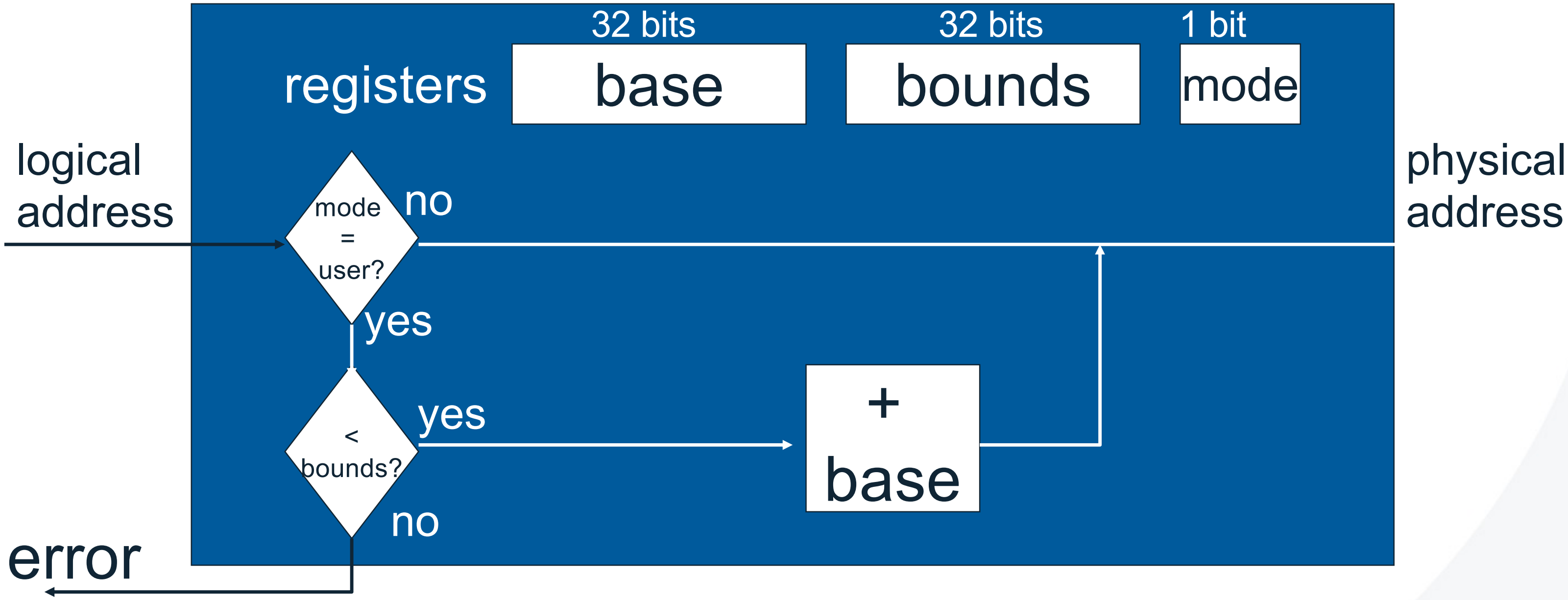
- **Idea**
  - limit the address space with a bounds register
- **Base register**
  - smallest physical addr (or starting location)
- **Bounds register**
  - size of this process's virtual address space
  - Sometimes defined as largest physical address (base + size)
- **OS kills process if process loads/stores beyond bounds**

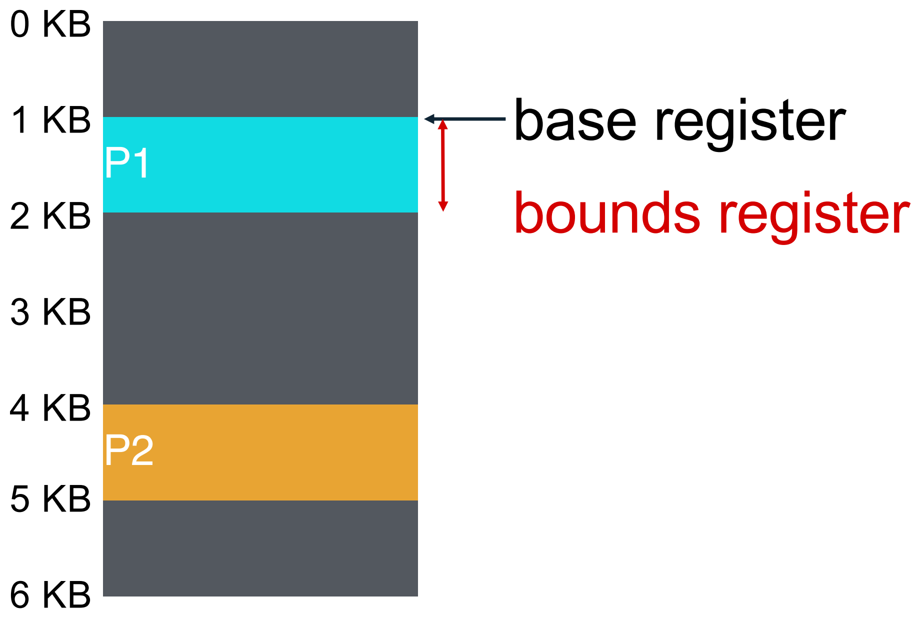
THE UNIVERSITY of ADELAIDE

Implementation of BASE+BOUNDS

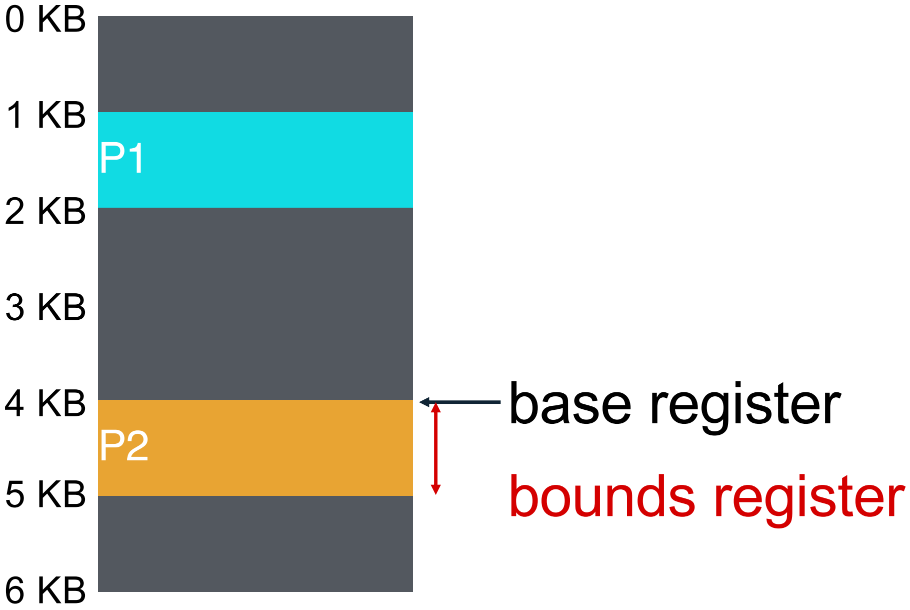
- **Translation on every memory access of user process**
  - MMU compares logical address to bounds register
    - if logical address is greater, then generate error
  - MMU adds base register to logical address to form physical address

Implementation of BASE+BOUNDS

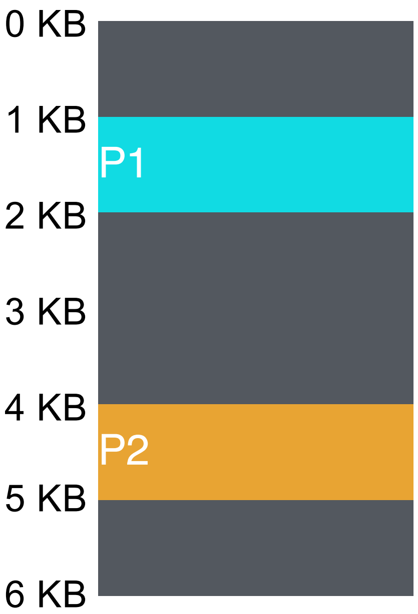




P1 is running

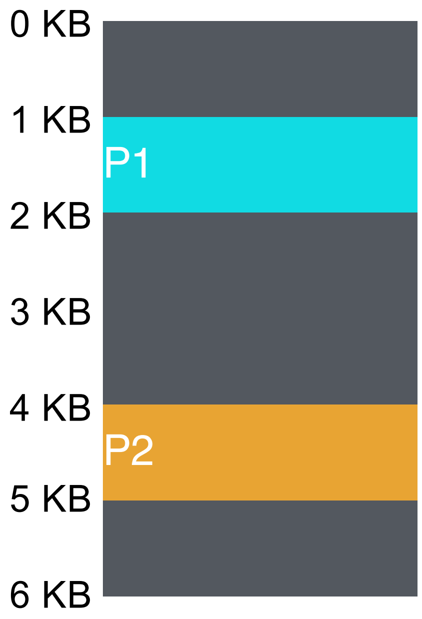


P2 is running



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 1000, R1	load 2024, R1
P1: store 3072, R1	

Can P1 hurt P2?

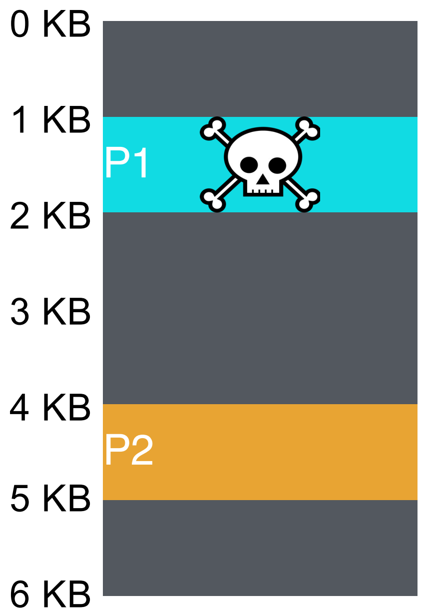


Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 1000, R1	load 2024, R1
P1: store 3072, R1	interrupt OS!

3072 > 1024

Can P1 hurt P2?

THE UNIVERSITY of ADELAIDE



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	interrupt OS!

Can P1 hurt P2?

Managing Processes with Base and Bounds

- **Context-switch - Add base and bounds registers to PCB steps**
  - Change to privileged mode
  - Save base and bounds registers of old process
  - Load base and bounds registers
  - Change to user mode and jump to new process
- **What if don't change base and bounds registers when switch?**
- **Protection requirement**
  - User process cannot change base and bounds registers
  - User process cannot change to privileged mode

THE UNIVERSITY *of* ADELAIDE

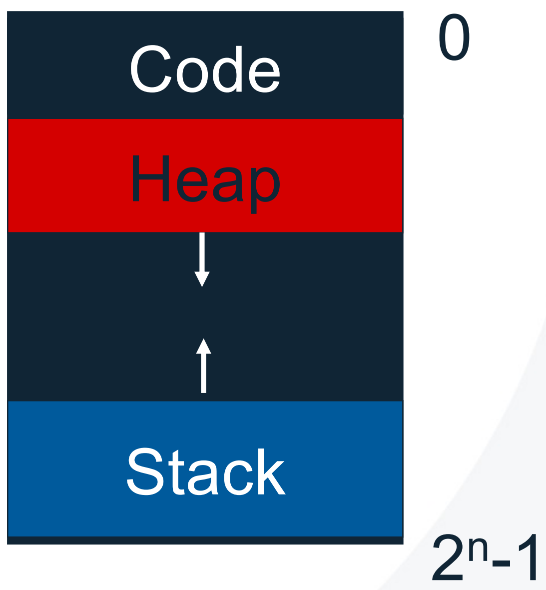
Base and Bounds Advantages

- **Advantages**
  - Provides protection (both read and write) across address spaces
  - Supports dynamic relocation
    - Can place process at different locations initially and also move address spaces
- **Simple, inexpensive implementation**
  - Few registers, little logic in MMU
- **Fast**
  - Add and compare in parallel

THE UNIVERSITY of ADELAIDE

Base and Bounds DISADVANTAGES

- **Disadvantages**
  - Each process must be allocated contiguously in physical memory
  - Must allocate memory that may not be used by process
  - No partial sharing: Cannot share limited parts of address space



C Strings

C strings terminated with \0 character.

Many operating systems and software components are written in C

- Interfaces inherit semantic “strings end with \0”.
- Some components don’t handle \0 embedded in string gracefully, even if programming language can.
- Note that UTF-16/UTF-32 include many byte 0s.

Note that \0 takes space – account for it!

- Overwriting can create a string that doesn’t end.

Formal name is NUL character

H	e	l	l	o	\0

THE UNIVERSITY of ADELAIDE

An Example Buffer Overflow

```
char A[8];
short B=3;
```

A	A	A	A	A	A	A	A	B	B
0	0	0	0	0	0	0	0	0	3

```
gets(A);
```

A	A	A	A	A	A	A	A	B	B
o	v	e	r	f	l	o	w	s	0

THE UNIVERSITY of ADELAIDE

Exercise: Out of Bounds Write

Writing beyond the buffer:

```
int main() {
    int array[5] = {1, 2, 3, 4, 5};
    int i;

    for( i=0; i <= 255; ++i )
        array[i] = 41;
}
```

What is the program output?

```
> gcc -o bufferw bufferw.c
> ./bufferw
```

THE UNIVERSITY of ADELAIDE

Stack Smashing

1. Calling a function
2. Function prologue
3. Overflowing a buffer
4. Shellcode

Calling a function

Given this C function:

```
void main() {
  f(1,2,3);
}
```

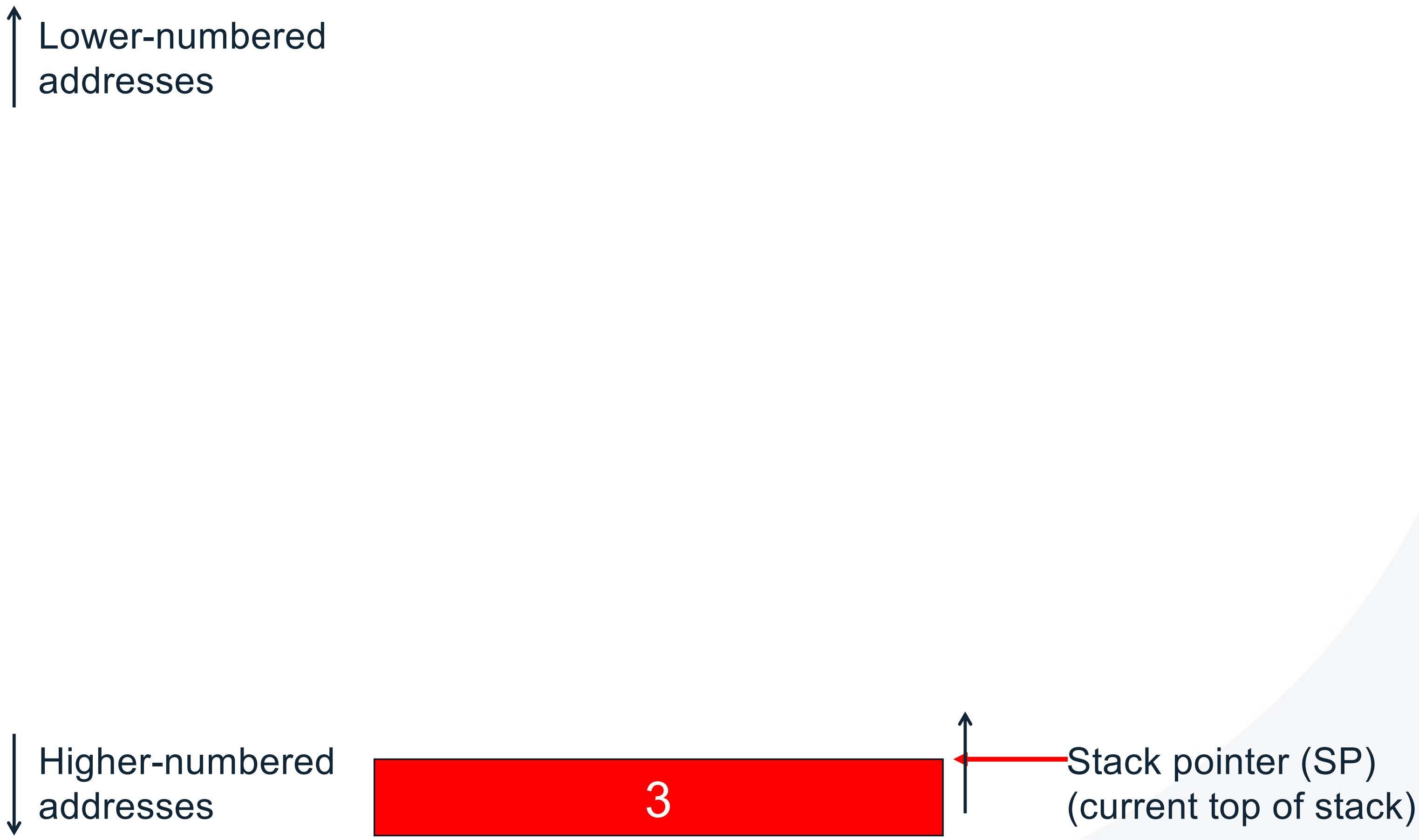
The invocation of f() might generate assembly:

```
pushl $3 ; constant 3
pushl $2 ; Most C compilers push in reverse order
pushl $1
call f
```

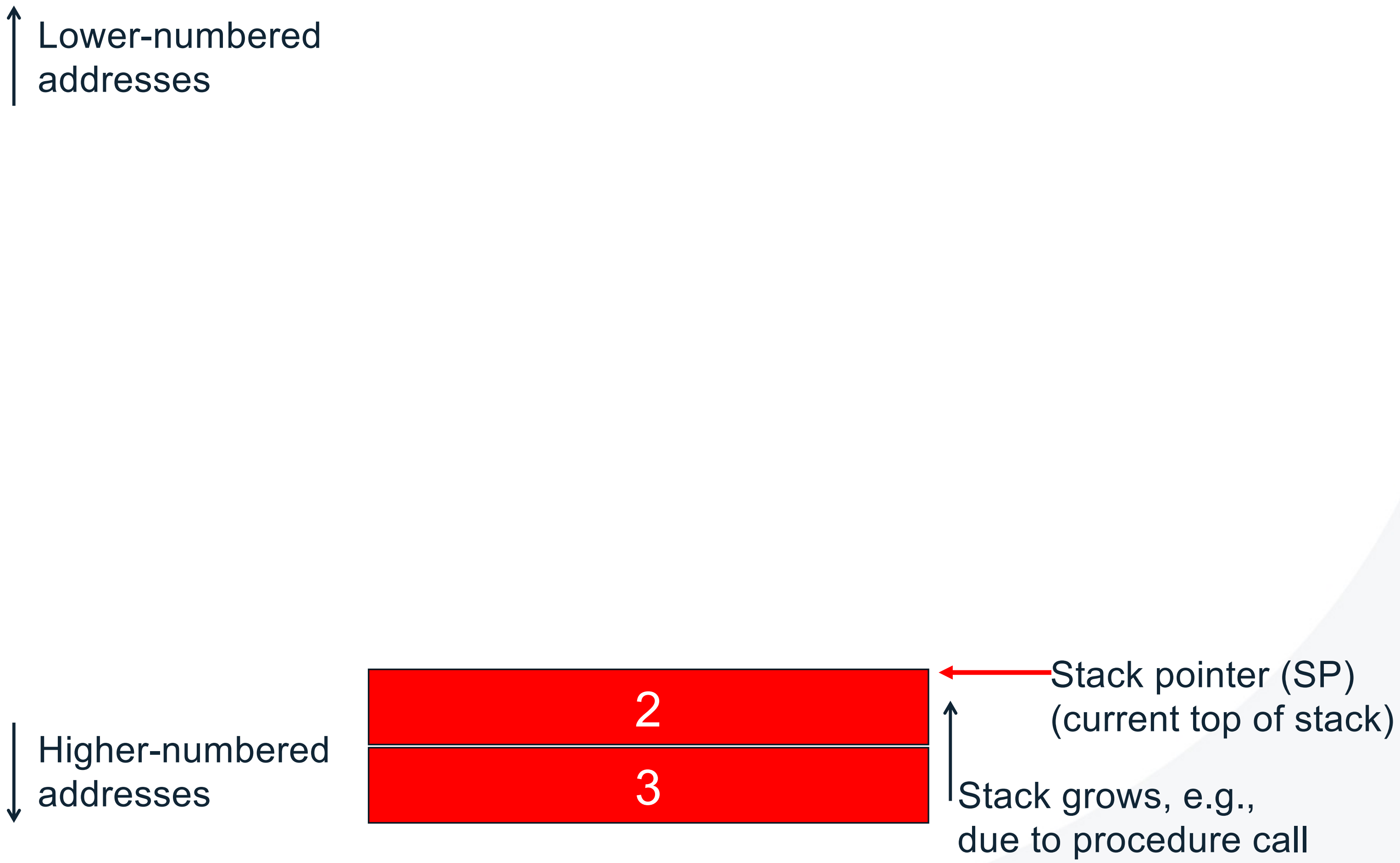
“call” instruction pushes instruction pointer (IP) on stack

- In this case, the position in “main()” just after f(…)
- Saved IP named the return address (RET)
- CPU then jumps to start of “function”

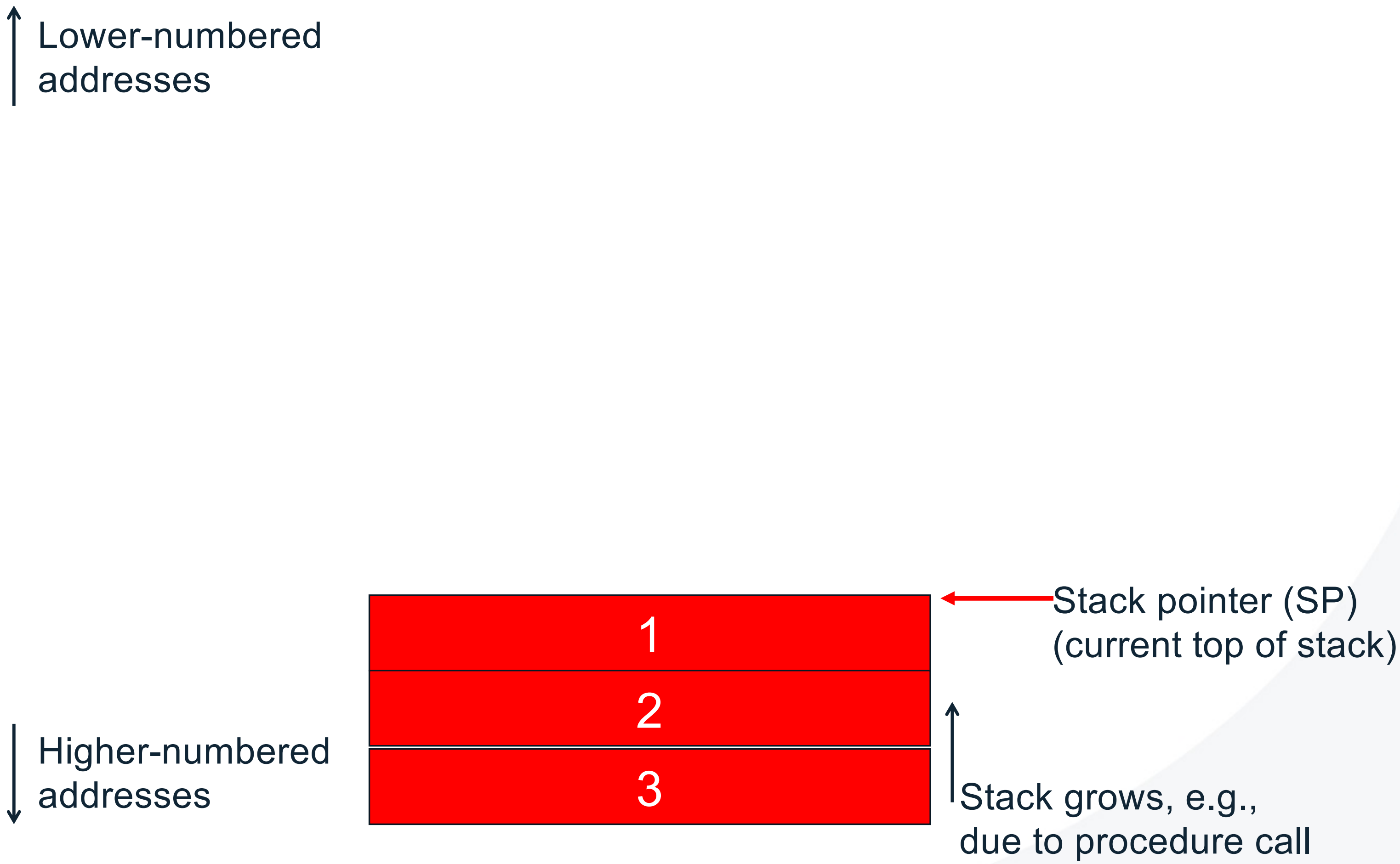
Stack: After push of value 3



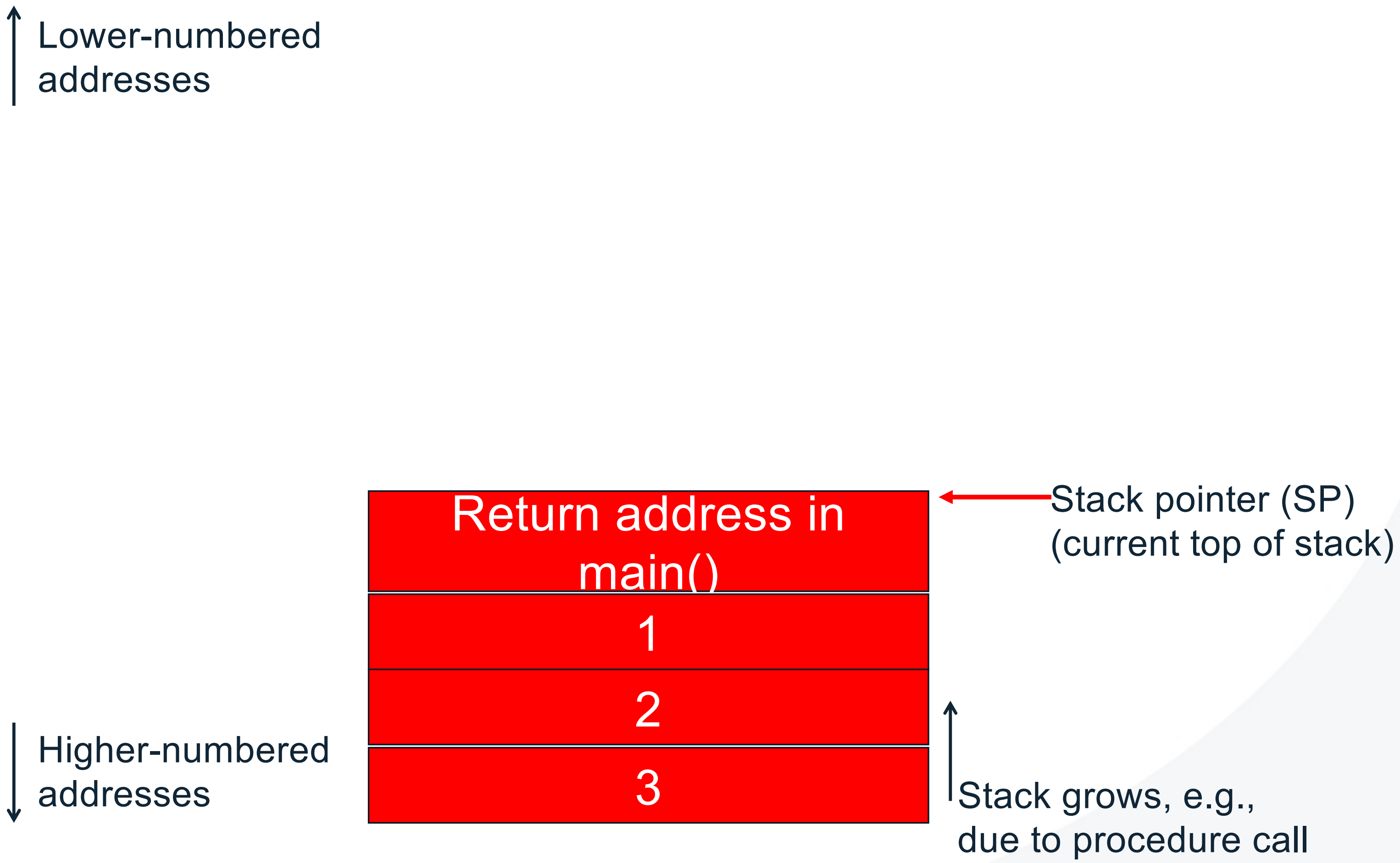
Stack: After push of value 2



Stack: After push of value 1



Stack: Immediately after call



Function prologue

Imagine `f()` has local variables, e.g. in C:

```
void f(int a, int b, int c) {
  char buffer1[4];
  char buffer2[12];
  strcpy(buffer2, "This is a very long string!!!!!!!");
}
```

Typical x86-32 assembly on entry of `f()` ("prologue"):

```
        pushl %ebp          ; Push old frame pointer (FP)
        movl %esp,%ebp      ; New FP is old SP
        subl $10,%esp       ; New SP is after local vars
              ; "$10" is calculated to be >= local var space
```

In the assembly above, ";" introduces a comment to end of line

THE UNIVERSITY of ADELAIDE

Stack: Immediately after call

↑ Lower-numbered addresses

Return address in main()
1
2
3

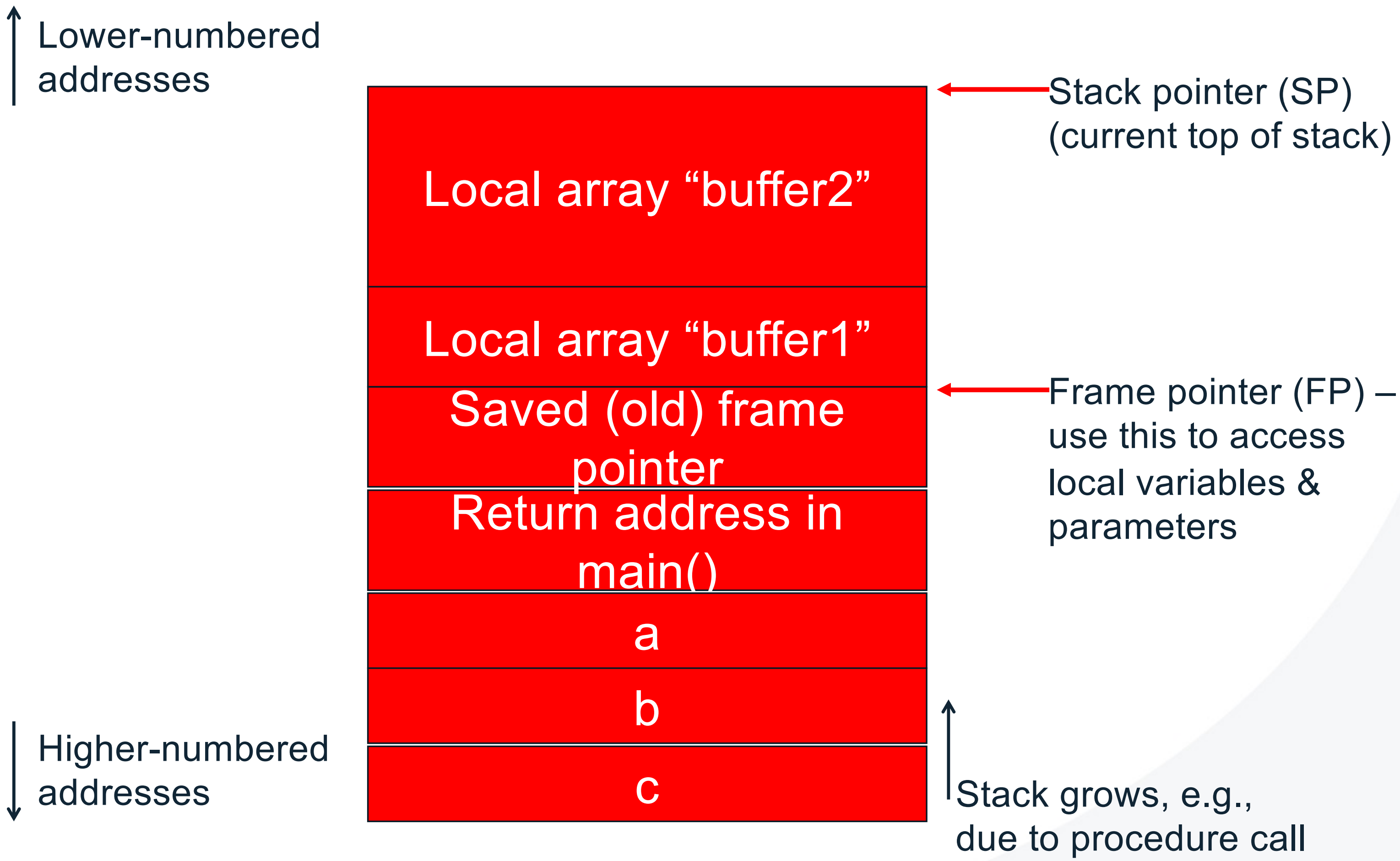
← Stack pointer (SP) (current top of stack)

↑ Stack grows, e.g., due to procedure call

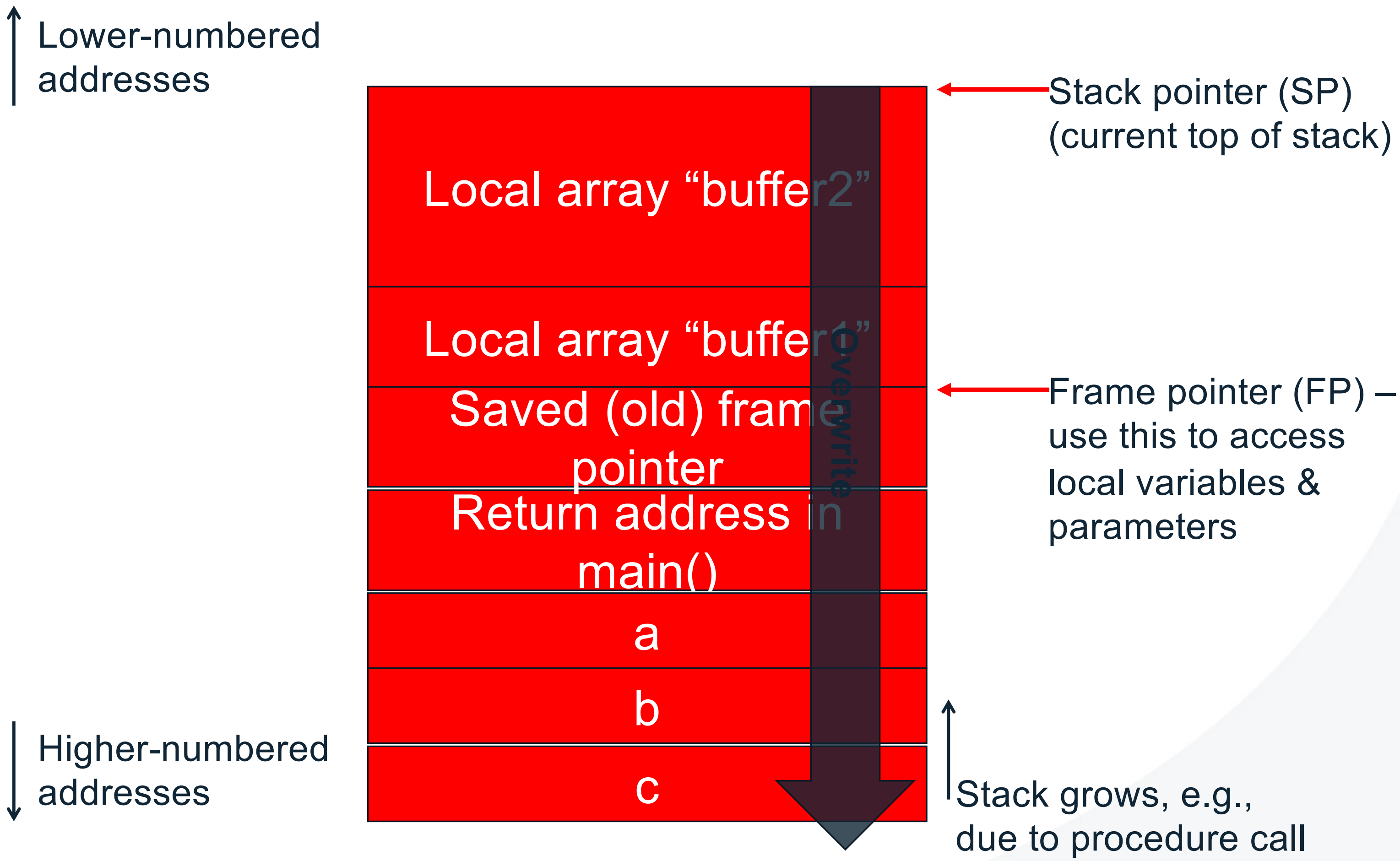
↓ Higher-numbered addresses

THE UNIVERSITY of ADELAIDE

Stack: After prologue

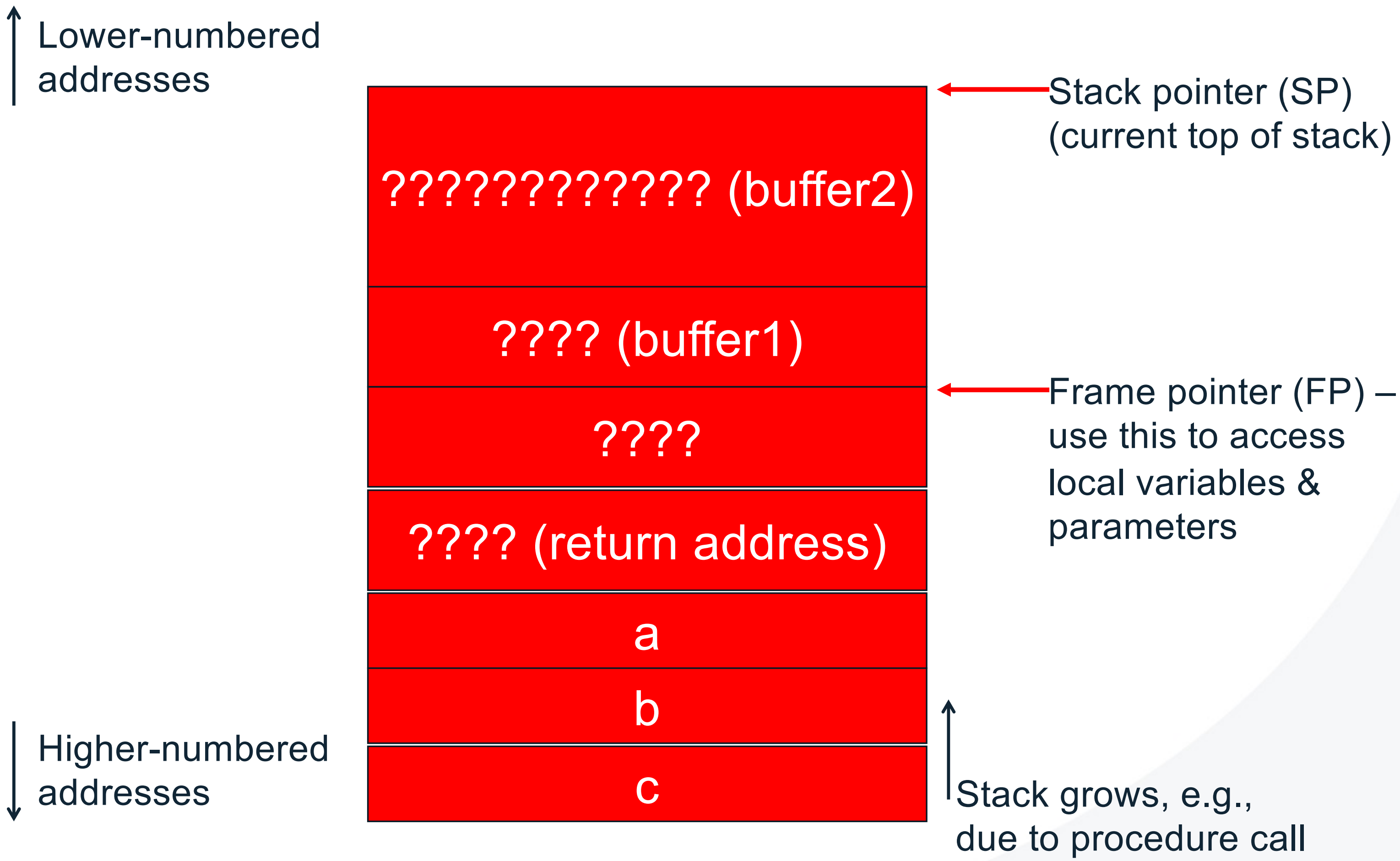


Stack: Overflowing buffer2



TPS: Replace ? By Overflow Values

buffer2 =
AAAAAAAAAAAABBBBCCCCDDDD

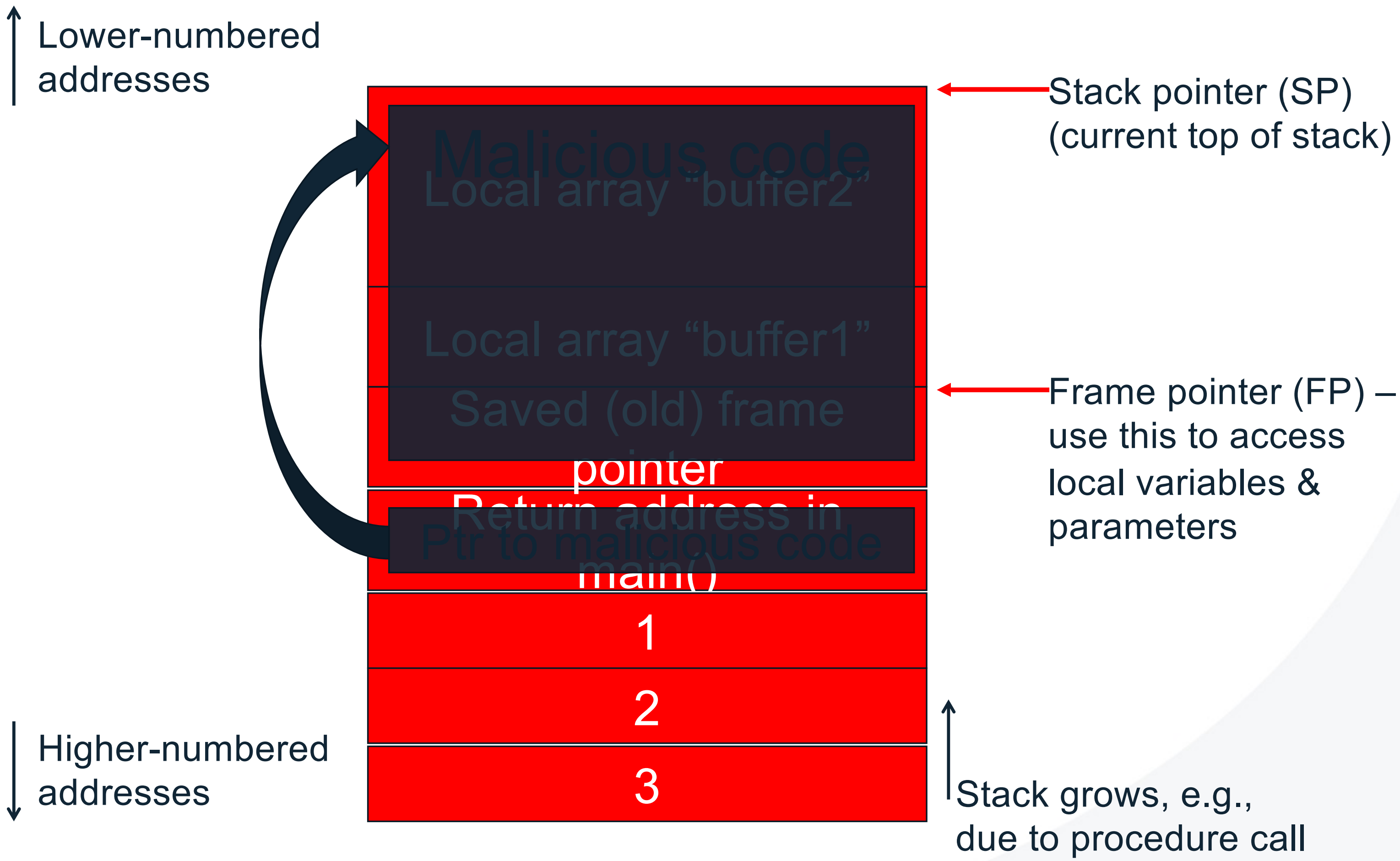


THE UNIVERSITY of ADELAIDE

Shellcode Injection

- Attacker can use buffer overflow to write machine code to the stack.
- If they can set the return value to point to this malicious code, on return from the attacked function, the victim program will run that code.

Stack: Overflow with Shellcode



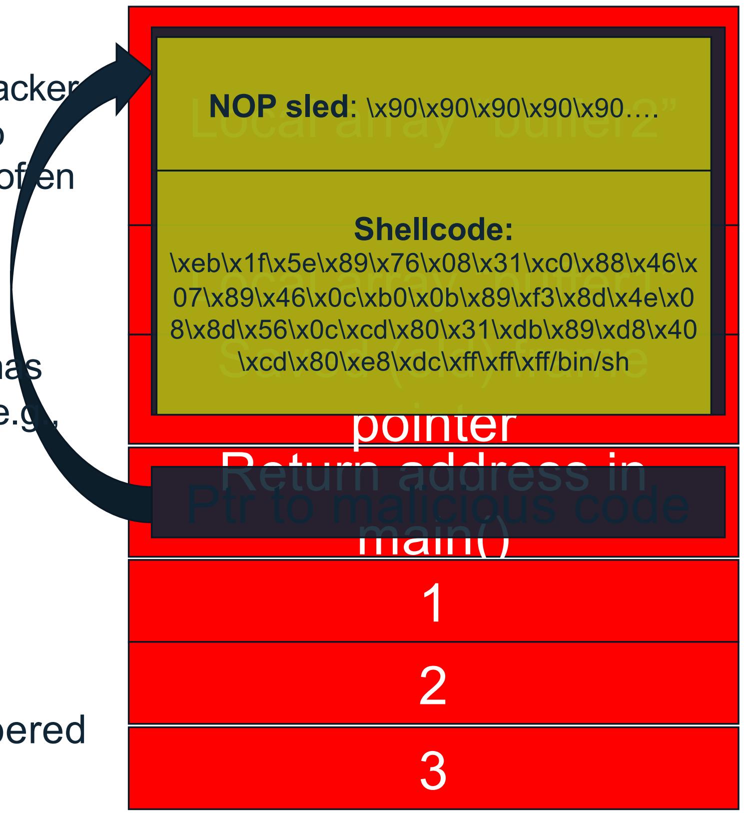
Stack: Shellcode + NOP Sled

↑ Lower-numbered addresses

NOP sleds let attacker jump anywhere to attack; real ones often more complex (to evade detection)

Shellcode often has odd constraints, e.g., no byte 0

↓ Higher-numbered addresses



Stack pointer (SP) (current top of stack)

Frame pointer (FP) – use this to access local variables & parameters

Stack grows, e.g., due to procedure call

THE UNIVERSITY of ADELAIDE

Conclusion

Virtualising the CPU: Scheduling

- Multi-Level Feedback Queue (MLFQ)
- Linux Completely Fair Scheduler (CFS)

Virtualising the Memory: Basics

- Address Spaces
- Address Translation Mechanism
- A bit on Stack and Heap