COMP1521 25T2

Week 10 Lecture 1

Concurrency, Parallelism and Threads and Virtual Memory

Adapted from Angela Finlayson, Xavier Cooney, Andrew Taylor and John Shepherd's slides

Announcements

Assignment 2: Due this Friday 18:00!

Optional practice exams:

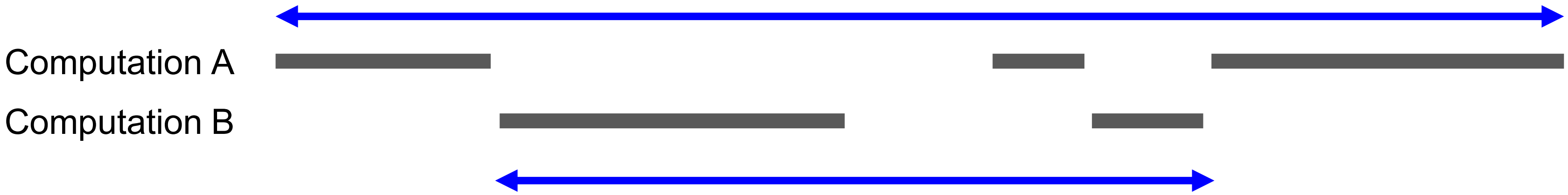
- Held during lab time this week
- Virtual exam environment
  - See what is available and what is not.
  - Become familiar with the environment before the exam.
  - Answer practice questions.
- Week 10 lab work must still be submitted.

Today's Lecture

- Concurrency and threads
  - Recap pthreads and mutexes
  - Deadlock
  - Atomics
- Virtual Memory

Concurrency & Parallelism

Concurrency: Multiple computations with *overlapping* time periods. Does not *have* to be simultaneous.



Parallelism: Multiple computations executing *simultaneously*.

Threads: parallelism *within* a process

- Threads allows us to create concurrency *within* a process
- Threads within a process *share* the address space:
  - Threads share code
  - Threads share global variables
  - Threads share the heap (`malloc`)
- Some other process state is shared
  - environment variables, file descriptors, current working directory, …

Threads: parallelism *within* a process

- Each thread has a separate execution state
    - Often called the Thread Control Block (TCB)
    - Includes CPU register values (including the program counter)
- Each thread has it's own stack
    - But a thread can still read/write to another thread's stack
- Each thread gets its own copy of errno!

Creating threads with `pthread_create`

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

- Starts a new thread running `start_routine(arg)`
- Information about the new thread stored in `thread`
- Thread has attributes specified in `attr` (NULL if you don't want special attributes)
- Returns 0 if OK, otherwise an error number (**does not set errno!**)
- Analogous to ***posix_spawn***.

Data Lifetime Issues

- When sharing data with a thread, we pass in the addresses of data
  - What if by the time the thread reads the data, that data no longer exists or has changed?

- The return value of a thread is also an address.
  - Is the memory allocated and it's content at that address still valid once the thread returns?

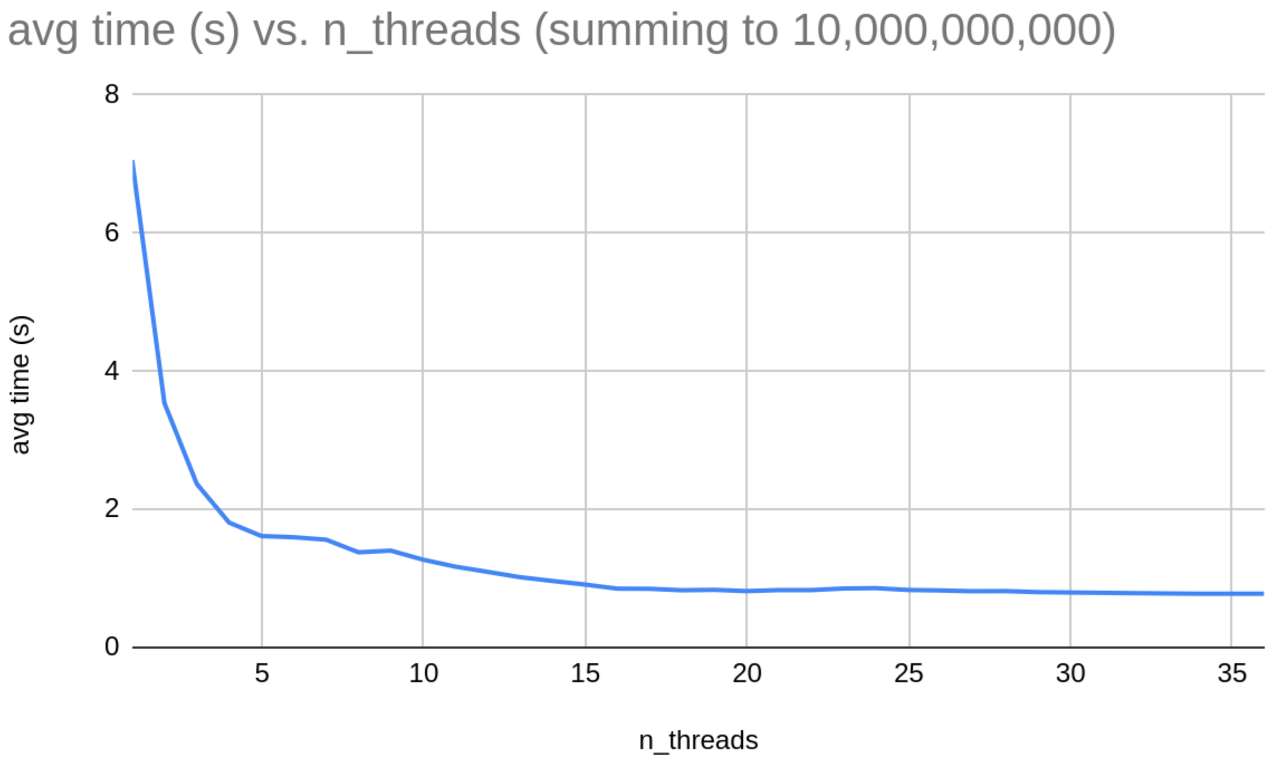
Waiting for threads with `pthread_join`

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

- Waits for `thread` to terminate, if it hasn't already terminated
- Return/exit value of thread placed in `*retval`
- Analogous to `waitpid`

- When **main** returns, *all* threads terminate

A graph of the performance of thread_sum.c



Some other concurrency benefits

- One thread can wait for I/O (block) while others make progress or wait for other I/O
- Useful for user interface programming

Global Variables and Race Condition

If bank_account = 100 and two threads execute concurrently

```
la      $t0, bank_account
# {| bank_account = 100 |}
lw      $t1, ($t0)
# {| $t1 = 100 |}
addi    $t1, $t1, 100
# {| $t1 = 200 |}
sw      $t1, ($t0)
# {| bank_account = ...? |}
```

```
la      $t0, bank_account
# {| bank_account = 100 |}
lw      $t1, ($t0)
# {| $t1 = 100 |}
addi    $t1, $t1, -50
# {| $t1 = 50 |}
sw      $t1, ($t0)
# {| bank_account = 50 or 200 |}
```

- This is a critical section.
- We want only one thread in the critical section at a time
  - We must establish mutual exclusion.

A solution: mutexes

- We need a way of guaranteeing *mutual exclusion* for certain shared resources (such as ***bank_account***)
- We associate each of those resources with a ***mutex***
- Only one thread can hold a mutex, any other threads which attempt to lock the mutex must wait until the mutex is unlocked
- So only one thread will be executing the section between the mutex lock and the mutex unlock

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

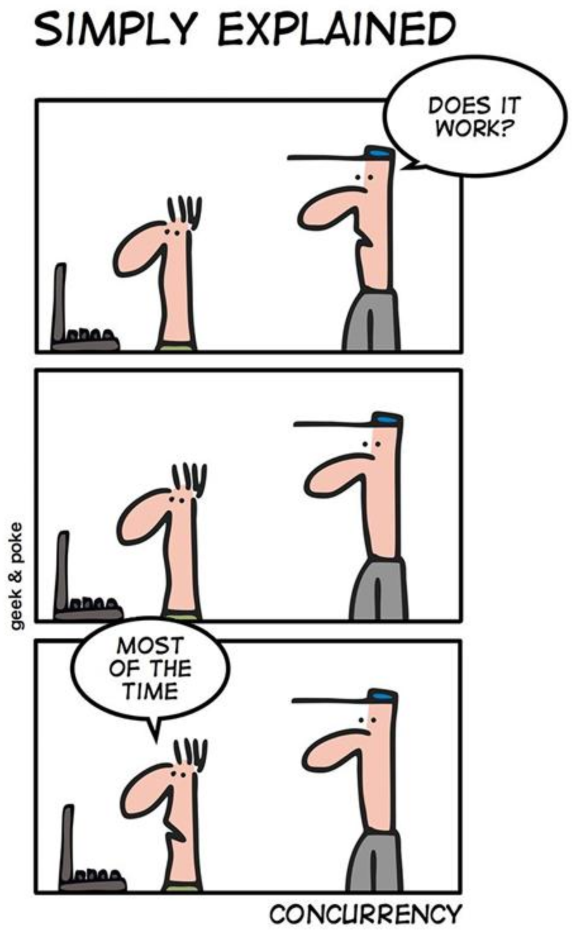
bank_account_mutex.c

```
int bank_account=0;
pthread_mutex_t bank_account_lock=PTHREAD_MUTEX_INITIALIZER;

void *add_100000(void*argument){
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&bank_account_lock);
        // only one thread can execute this
        // section of code at any time
        bank_account = bank_account + 1;
        pthread_mutex_unlock(&bank_account_lock);
    }
```

Code Demo: Deadlock

bank_account_deadlock.c



Deadlocks

THREAD 1

1. acquire lock_A
2. acquire lock_B
3. do_somthing(A, B)
4. release lock_B
5. release lock_A

THREAD 2

1. acquire lock_B
2. acquire lock_A
3. do_somthing(A, B)
4. release lock_A
5. release lock_B

Solving deadlocks

- A simple rule to avoid deadlocks:
  - All thread *must* acquire locks in the same order
  - (also good if locks are released in reverse order, if possible)
- e.g., always acquire lock_A before lock_B

THREAD 1

1. acquire lock_A
2. acquire lock_B
3. do_somthing(A, B)
4. release lock_B
5. release lock_A

THREAD 2

1. acquire lock_A
2. acquire lock_B
3. do_somthing(A, B)
4. release lock_B
5. release lock_A

Atomics

- With hardware support, we can avoid data races without needing to use locks!
- In C, we can use ‘atomic types’, which guarantee that certain operations using them will be performed *atomically* (indivisibly) ⇒ no data race!
- Also avoids overhead of mutexes
- And since no locks are involved, we can’t introduce deadlock
- Atomics don’t solve all concurrency problems
- There are still some subtle problems (which we don’t cover in COMP1521)

Atomics

- Declaring an atomic variable
  - `atomic_int x = 10;`
  - `x += 1;`   // Will be done atomically
  - `x = x + 1;`   //Will NOT be done atomically!!!!
- A subset of functions in **stdatomic.h**:
  - atomic_fetch_add
    - `atomic_int x = 10;`
    - `int old = atomic_fetch_add(&x, 1);`
  - atomic_fetch_sub
  - atomic_fetch_or, atomic_fetch_xor, atomic_fetch_and

Add code with atomic in it

```
atomic_int bank_account = 0;

void *add_100000(void *argument) {
    for (int i = 0; i < 100000; i++) {
        // NOTE: This *cannot* be `bank_account = bank_account + 1`,
        // as that will not be atomic!
        // However, `bank_account++` would be okay
        // `atomic_fetch_add(&bank_account, 1)` would also be okay
        bank_account += 1;
    }
```

Concurrency is really complex!

- This is just a taste of concurrency!
- Other fun concurrency problems/concepts: livelock, starvation, thundering herd, memory ordering, semaphores, software transactional memory, user threads, fibers, etc.

- A number of courses at UNSW offer more:
    - COMP3231/COMP3891: [Extended] operating systems
    - COMP3151: Foundations of Concurrency
    - COMP6991: Solving Modern Programming Problems with Rust
    - … and more!

Virtual Memory

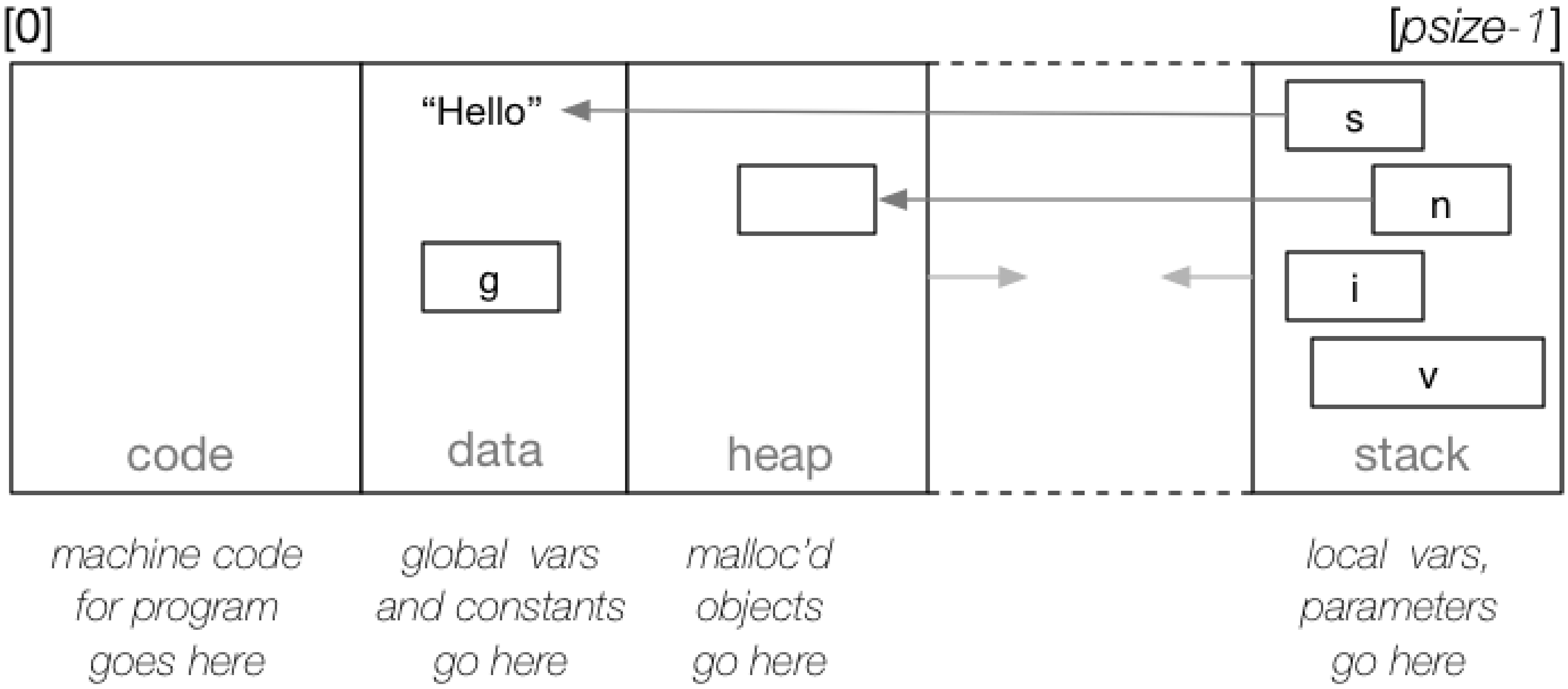
(A short intro)

Virtual memory goals

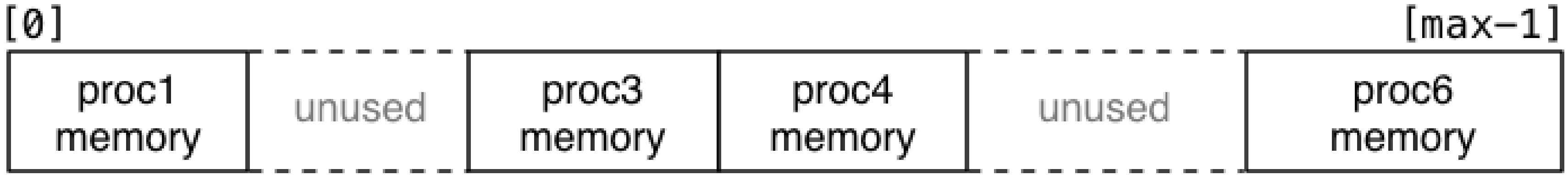
- System RAM location and size differs across machines
  - How can we provide an abstract view of memory to hide these details from applications?

- Multi-processing
  - How can we concurrently run two applications that expect to be at the same memory address?

Memory regions

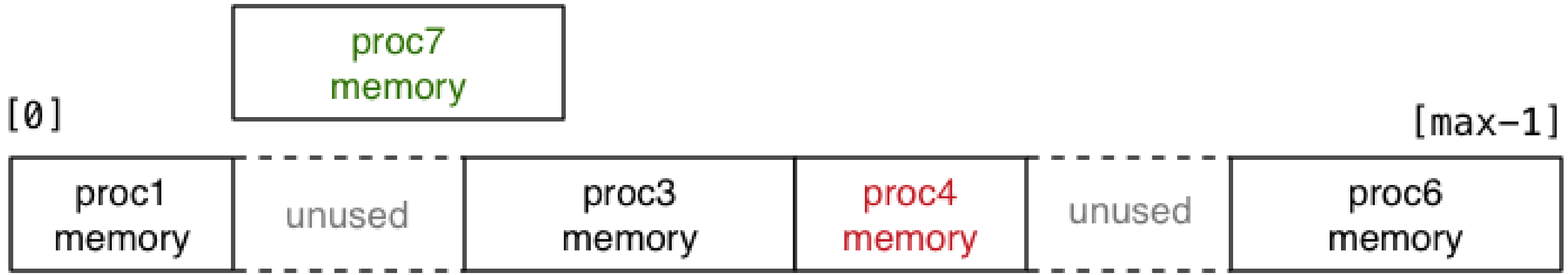


RAM partitioning

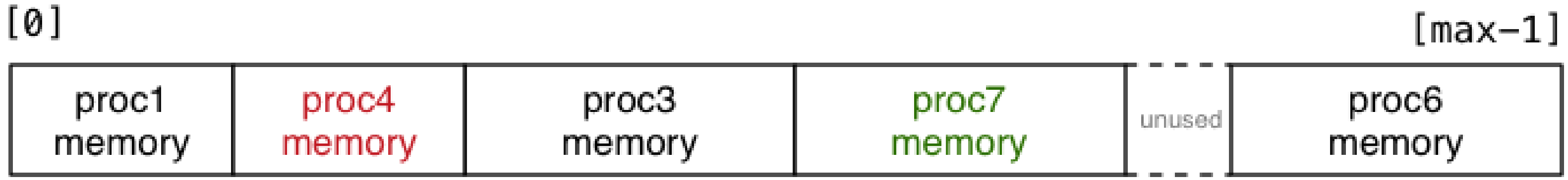


- Every process in a contiguous region of RAM, starting from address *base* and finishing at address *limit*
- Each process sees it's own address space as [0 .. psize-1]
- Process can be loaded anywhere in memory unchanged
- Address a translated to a + *base*
- Access check to ensure a + *base* < *limit*
- Easy to implement in hardware

What if we want to add a new process?



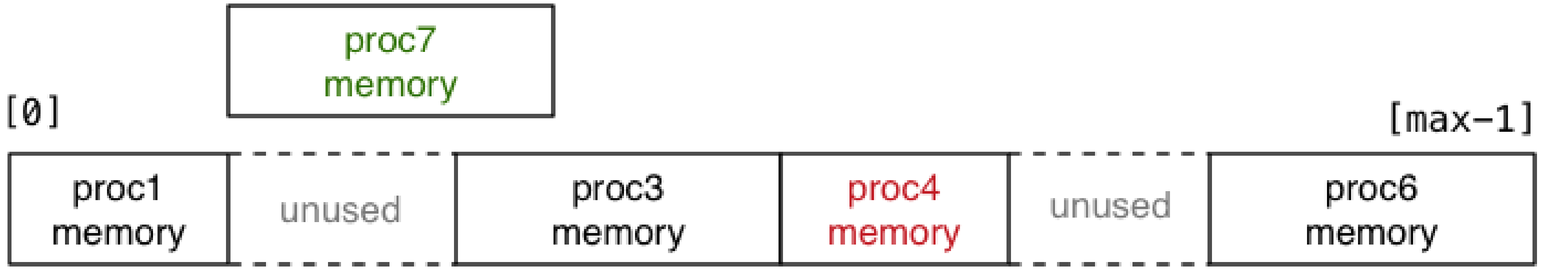
- New process doesn't fit in unused fragments
- Must move other process to defragment memory

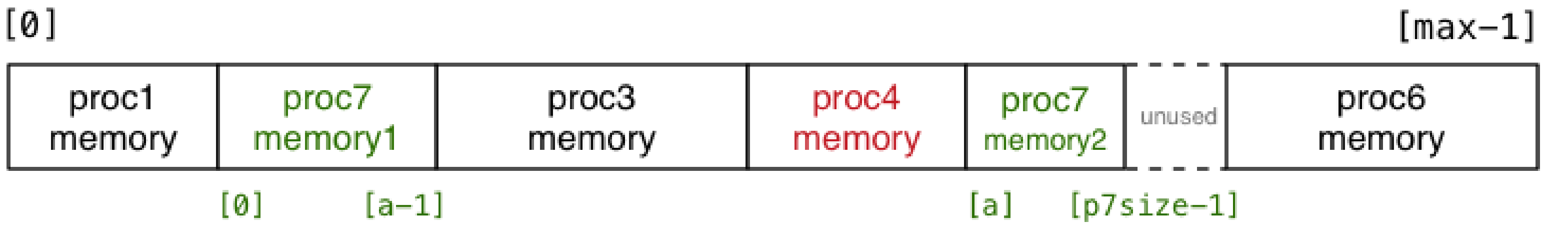


- Defragmentation reduces system performance
  - Search for free space, copy memory, etc

Split processes

- Idea: split process memory over multiple regions





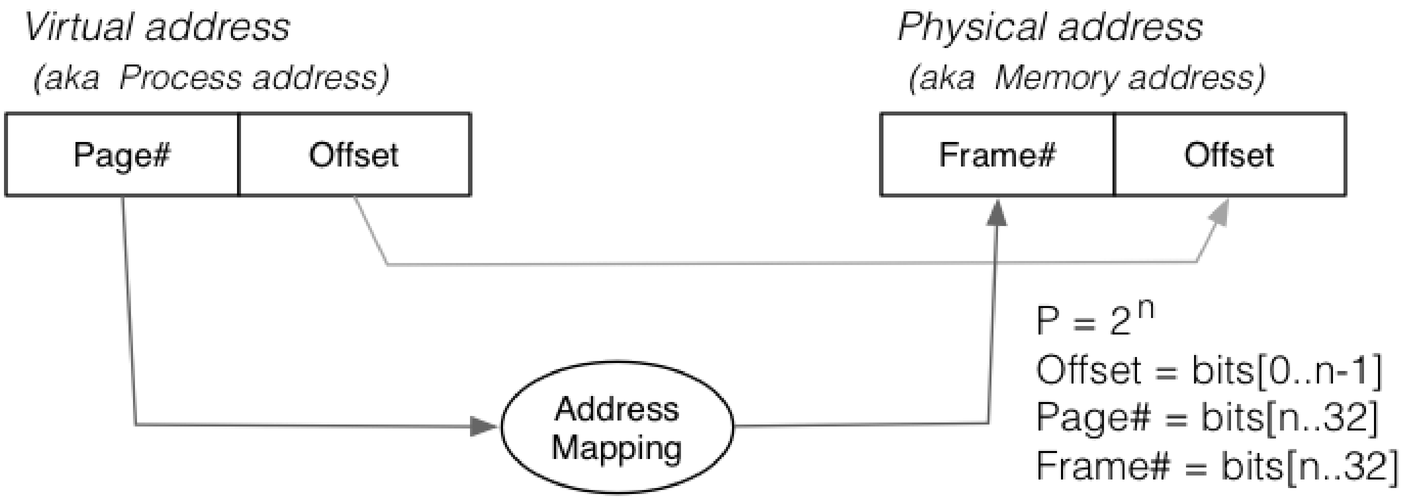
Virtual memory with pages

- Big idea: make all segments the same size (a power of 2)
    - Call each segment of address space a *page* of size *P*
    - Translation of address can be implemented as array
    - Each process has an array called it's *page table*
    - Each array element contains the physical address in RAM of the corresponding page
    - Given virtual address V and page size P:

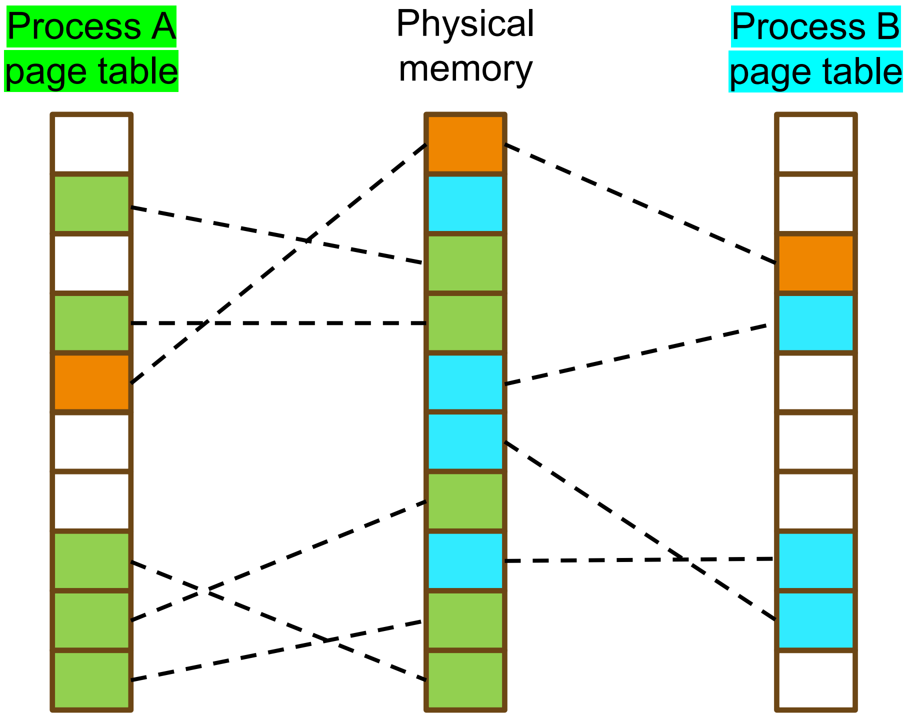
      physical_address = page_table[V/P] + V%P
    - Simple to implement in silicon using bitops with P being pow2

Page mapping

- Since P == 2^n, some bits (offset) are the same in virtual and physical address



Process page tables with memory sharing



Note: For 32 bit address space and 4096 Byte pages, page table size is ~ 1 M entries!

How many entries for a 64 bit address space?

Memory efficient page table representations out of scope of this course.

COMP3231/3891: Operating Systems covers virtual memory in more detail.

Lazy loading

- How much of our memory segments must be loaded before a program can execute?
  - .text?
  - .data?
  - or just main(...) and a stack?

Lazy loading

- How much of our memory segments must be loaded before a program can execute?
  - .text?
  - .data?
  - or just main(...) and a stack?
  - Nothing at all?

Lazy loading

- Idea:
  - Don't allocate pages
  - Link page table entries to files and offsets
  - When page is accessed, intercept SIGSEGV, load file content, then resume
  - Pages only loaded when needed
  - Some pages may never be loaded

fork() optimisations

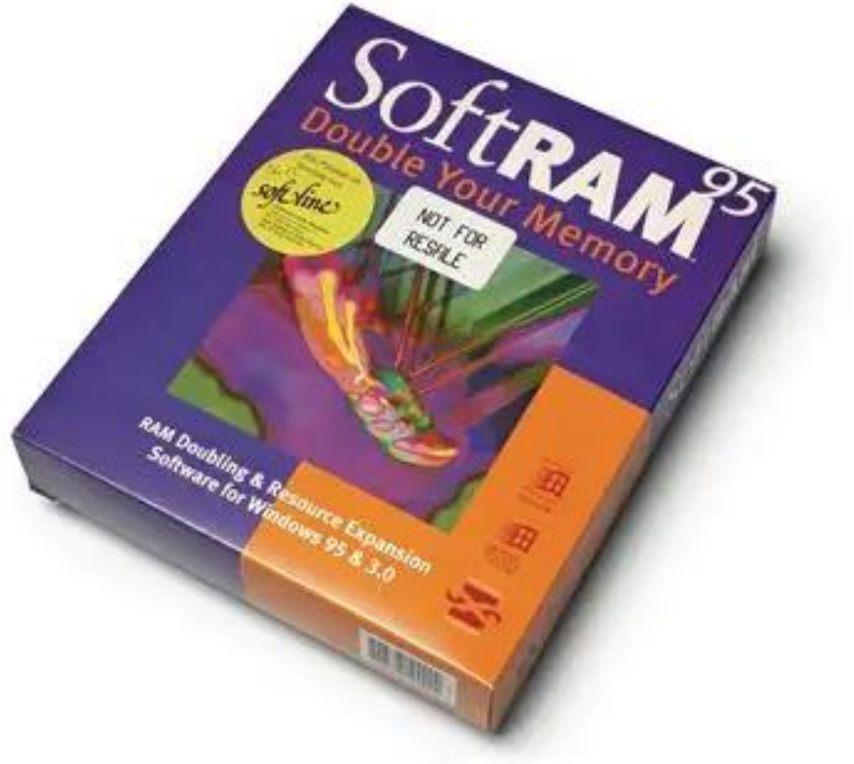
- Mark all pages read only
- Copy the address space -- the page table -- not memory
- Both parent and child share physical memory
- When page is written, intercept SIGSEGV, copy the page, update the page table, add write permissions, resume.

Software RAM

~~Software RAM~~ Swapping

- Three options when system is out of RAM:
  - Pause a process until memory is available
  - Kill a process (we can't pause the OS!)
  - Swapping: Temporarily move memory to disk to free that memory for other more immediate uses
    - Mac/Linux uses swap files/partitions
    - Windows uses a pagefile

Software RAM



For $80, increases windows pagefile size setting on your behalf, thereby increasing memory limits.

Also provides a fancy dashboard to show how much "RAM" you got for your money.

Swapping

- Similar operation to lazy loading, but page data contained in swap file
    - Link page table entry to file and file offset.
    - Intercept SIGSEGV to load the page back in on demand

- How to choose which page to move to disk?
    - Best page is one that won't be used again by its process
    - Prefer pages that are read-only and already on disk
    - Prefer pages that are unmodified and already on disk
    - Prefer pages that are used by only one process

Swapping

- OS can’t predict whether a page will be required again by its process
- But we do know whether it has been used recently (if we track this)
- One good heuristic - replace Least Recently Used (LRU) page.
    - Page not used recently probably not needed again soon

What we learnt Today

- Concurrency and threads
  - Recap pthreads and mutexes
  - deadlock
  - atomics

- Virtual memory

Next Lecture

Find out about the Final Exam!

Reach Out

Content Related Questions:
[Forum]

Admin related Questions email:
cs1521@cse.unsw.edu.au

Student Support | I Need Help With…

My Feelings and Mental Health Managing Low Mood, Unusual Feelings & Depression	**Mental Health Connect**	student.unsw.edu.au/**counselling** Telehealth
	Mind HUB	student.unsw.edu.au/**mind-hub** Online Self-Help Resources
	In Australia Call Afterhours UNSW Mental Health Support Line	1300 787 026 5pm-9am
	Outside Australia Afterhours 24-hour Medibank Hotline	+61 (2) 8905 0307
Uni and Life Pressures Stress, Financial, Visas, Accommodation & More	**Student Support Indigenous Student Support**	— student.unsw.edu.au/**advisors**
Reporting Sexual Assault/Harassment	**Equity Diversity and Inclusion (EDI)**	— edi.unsw.edu.au/**sexual-misconduct**
Educational Adjustments To Manage my Studies and Disability / Health Condition	**Equitable Learning Service (ELS)**	— student.unsw.edu.au/**els**
Academic and Study Skills	**Academic Language Skills**	— student.unsw.edu.au/**skills**
Special Consideration Because Life Impacts our Studies and Exams	Special Consideration	— student.unsw.edu.au/**special-consideration**