UPE Tutoring:

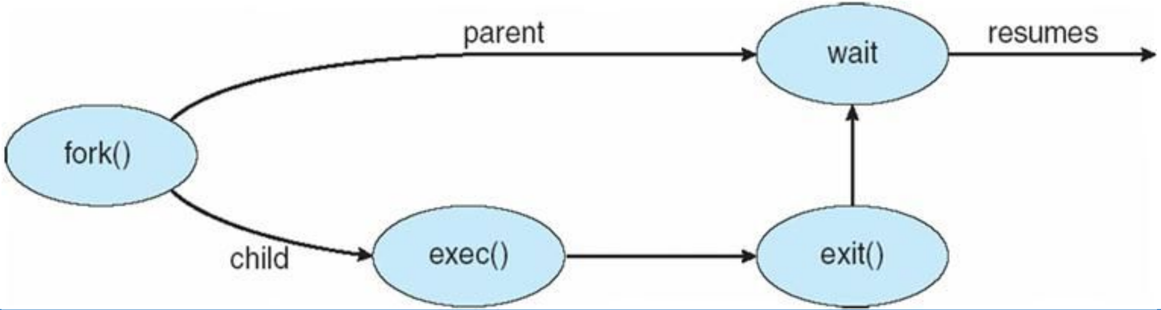
CS 111 Midterm Review

Sign-in https://goo.gl/RT38gH

Slides link available after sign-in

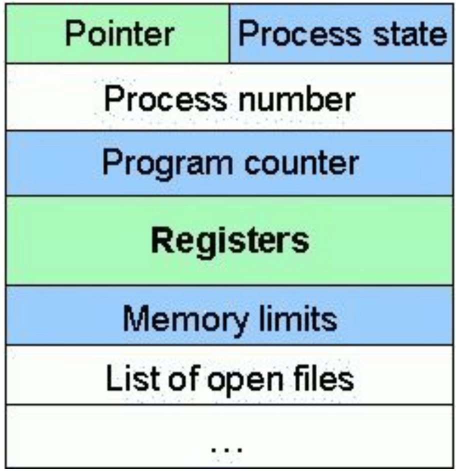
Processes - Basics

- Processes consist of code and data in a process's address space and a representation within the OS itself
- Processes run under limited direct execution where they have full control of the CPU
- Processes in UNIX created through a combination of fork and exec
  - fork copies the current process into a new one with the same data and open files
  - exec replaces the currently running code with another program's code and wipes the process's stack and heap and machine state, but keeps file table



Processes - Representation in Memory

- OS also keeps some data for each process within its own structures
  - Process control block for each process within process list keeps track of information such as PID, state, parent, address space information, etc.
  - OS keeps track of open files in FD table, locks, and current working directory



Process Control Block Example

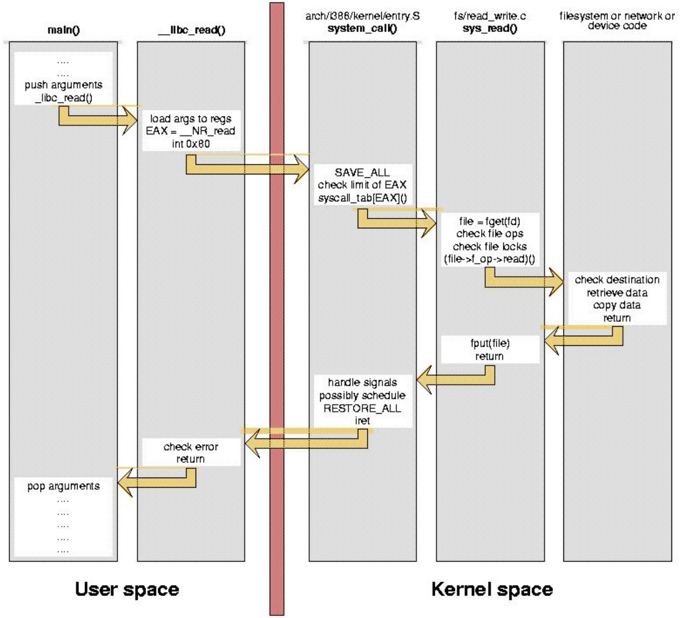
```
/* ....*/
/* memory management info */ struct mm_struct *mm;
/* open file information */ struct files_struct *files;
/* tss for this task */ struct thread_struct tss;
int pid;
volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped
*/
long priority;
unsigned short uid,euid,suid,fsuid;
long utime, stime, cutime, cstime, start_time;
/* .... */

struct x86_hw_tss {
/* .... */
       unsigned short        __ss1h;
       unsigned long         sp2;
       unsigned short        ss2, __ss2h;
       unsigned long         __cr3;
       unsigned long         ip;
       unsigned long         flags;
       unsigned long         ax;
       unsigned long         cx;
/* .... */
}
```

Processes - Libraries

- Processes can access libraries either statically or dynamically
  - Static linking - done at link time, libraries placed inside executable
  - Dynamic linking - done at runtime, libraries mapped into address space
    - Dynamic libraries can be shared by multiple processes
    - Can either be done at load time or runtime depending on implementation
- Processes are written to connect with API or ABI
  - API - Application Programming Interface, source level interface
    - Source level interface such as functions, macros, data types
    - API compliant program must be recompiled for each different architecture
  - ABI - Application Binary Interface, binary interface
    - Specific conventions on linkage, data formats, linkage conventions
    - ABI compliant program will run unmodified on a system with that ABI

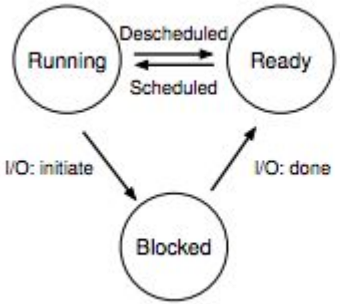
Processes - System Calls and Traps

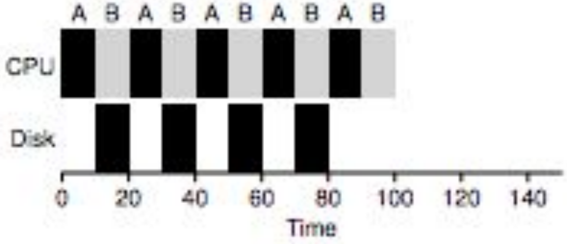
- Process is unable to execute privileged instructions, must ask kernel to do so



Scheduling - Mechanism

- Preemptive scheduling requires preemption enabled by the hardware to stop a process at a given time interval
- Non-preemptive scheduling waits for an opportunity given by the process





Scheduling Policies

- Non-preemptive
  - First In First Out
  - Shortest Job First
  - Real-time Scheduling
    - Soft versus hard real time, schedules normally created beforehand
- Preemptive
  - Shortest Time to Completion First
    - When a new job comes in, see if it will complete sooner than the current one
  - Round Robin
    - Run all jobs for a specific amount of time and then switch
  - Multilevel Feedback Queue
    - Observe a programs behavior and run it accordingly

Process/Scheduling Related Questions

1. What information must the OS save when performing a context switch?
2. Why does Shortest Time to Completion require preemption?
3. What resources are replaced when exec is called?
4. What is the relationship between a system’s ABI and its system call interface?
5. Why can’t shared libraries include global data?

Virtual Memory

What?

- Abstract way of referring to physical memory location using virtual memory
- Generally a hardware method (software doesn't know anything about what the real memory location is)

Why?

- Memory Abstraction
- Safety
- Running more than just one process

How?

- Address Translation (how hardware converts from virtual to physical memory
- Need to juggle translation for multiple programs
- MMU(Memory Management Unit)

Virtual Memory

The basic way: Base and Bounds

- Hardware keeps track of the Base and Bounds for each process
- This helps translate each processes’ virtual memory to the actual physical memory
- Process thinks it is at address 0x00000000
    - In reality it is 0x00000000 + Base
    - Max value is 0x00000000 + Bounds
- The physical base location is picked from the *free list*
- There is a Memory Management Unit that does all of this per address

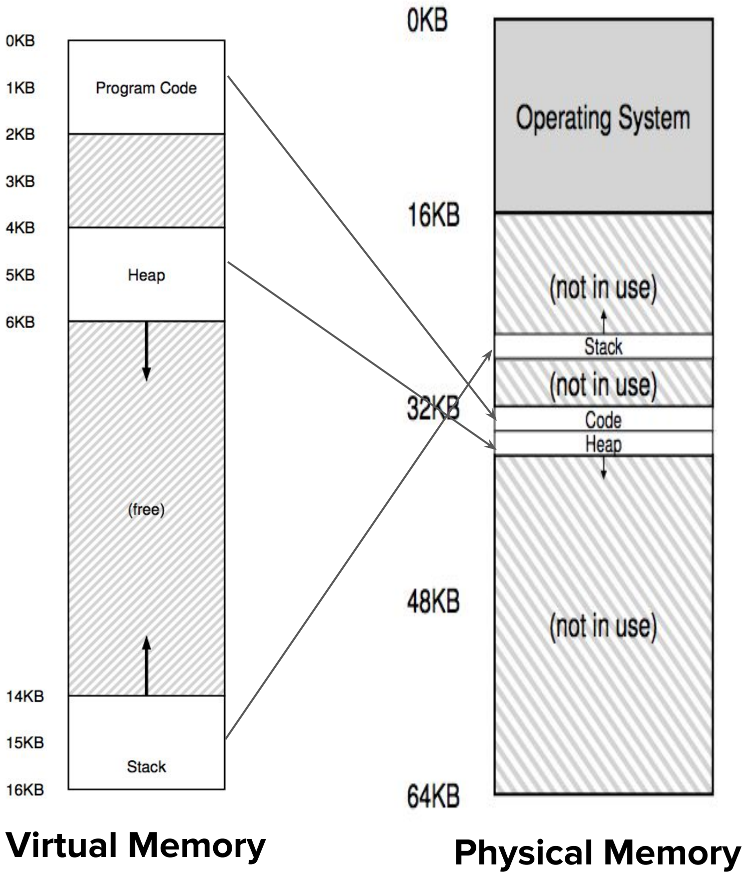
Segmentation

- Problem of fragmentation
  - Sometimes segments get placed in weird spots and even though you have enough memory, a new process might not fit in the “gaps”
- Lets use segments!
- Segmentation breaks up process memory into regularized chunks
- Essentially base/bounds for every segment of memory
- Since stack and heap grow in opposite directions, you also need another bit to specify direction of each segment

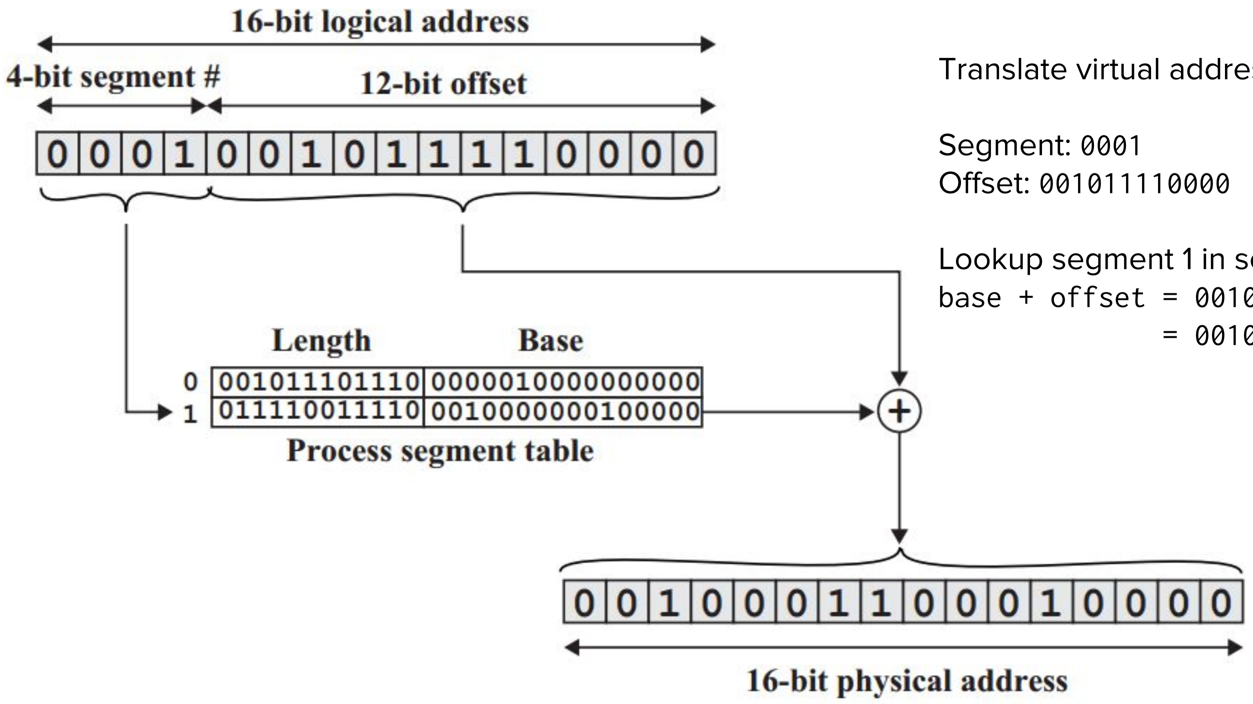
Segmentation

Some Process

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



Address Translation with Segmentation

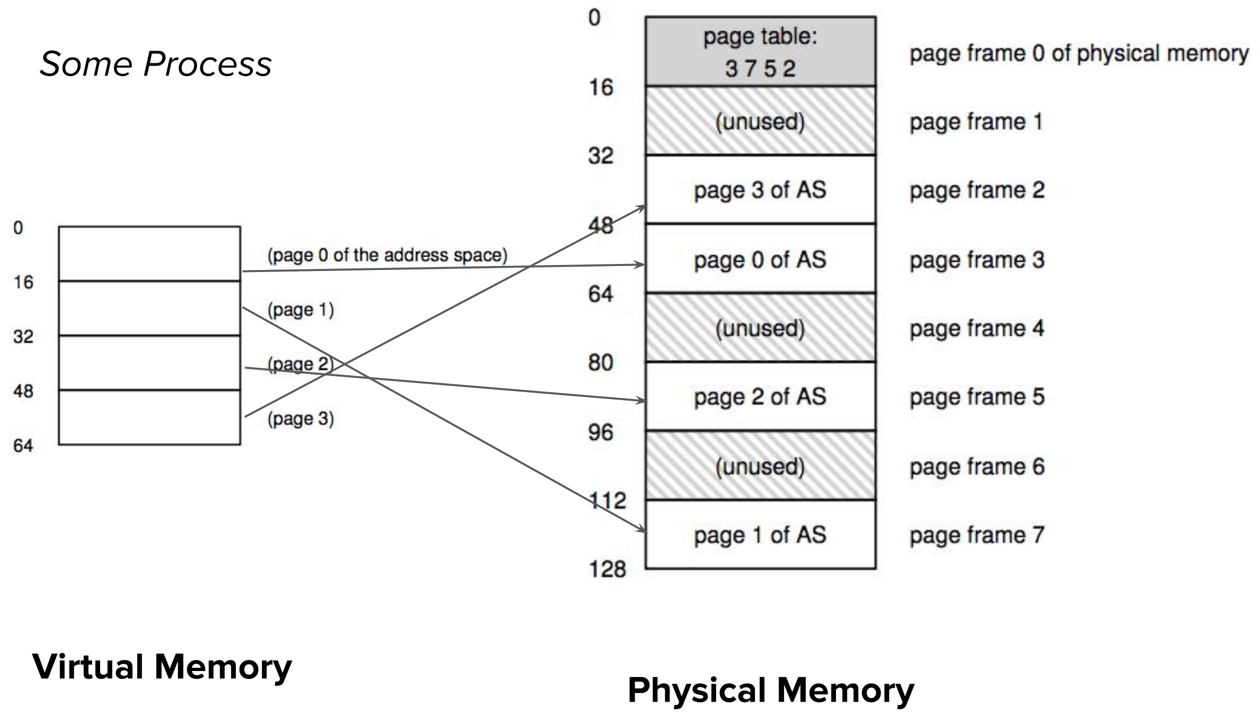


Translate virtual address `0001001011110000` to physical address

Segment: `0001`
Offset: `001011110000`

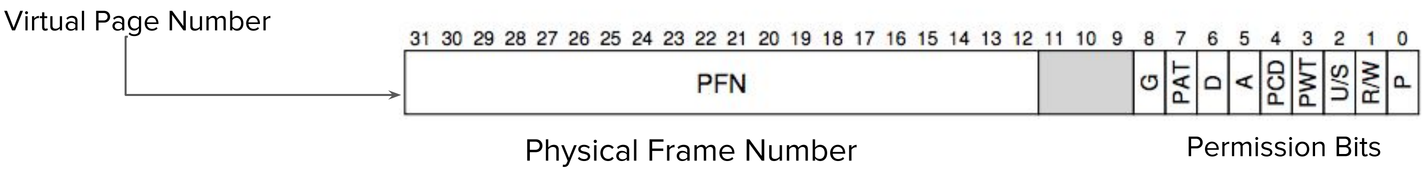
Lookup segment 1 in segment table, physical address is
```
base + offset = 0010000000100000 + 001011110000
              = 0010001100010000
```

Paging

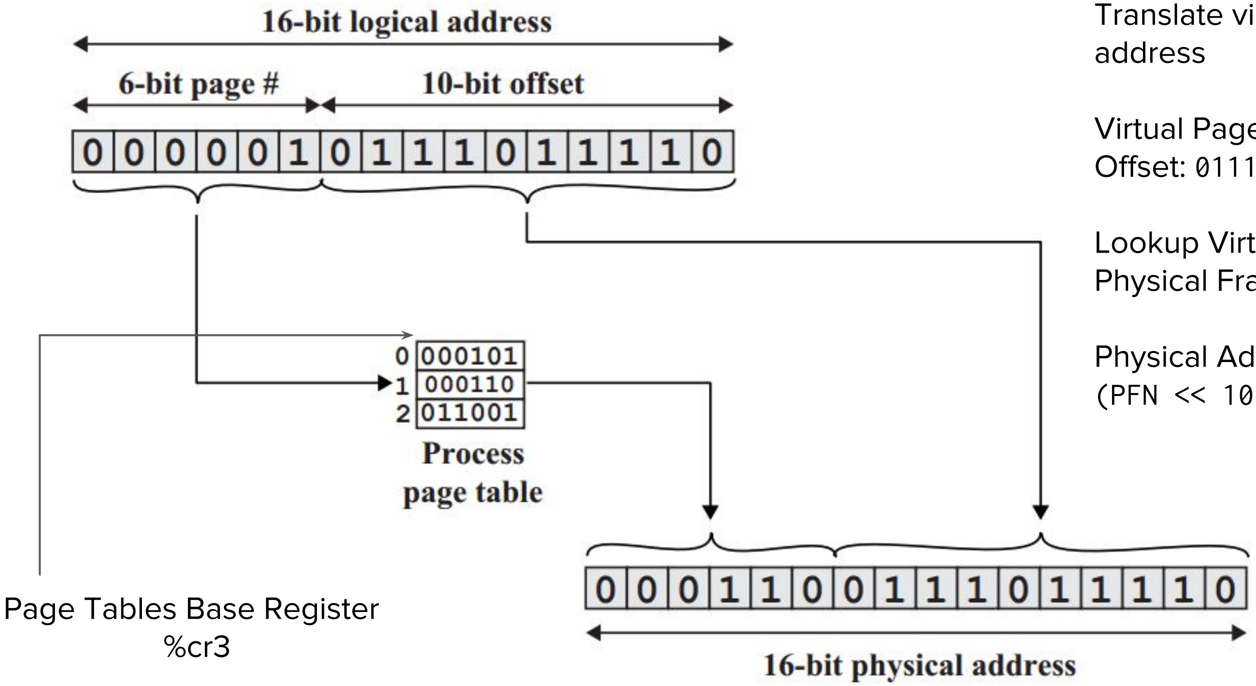

Some Process
0
16
32
48
64
(page 0 of the address space)
(page 1)
(page 2)
(page 3)
0
16
32
48
64
80
96
112
128
page table:
3 7 5 2
(unused)
page 3 of AS
page 0 of AS
(unused)
page 2 of AS
(unused)
page 1 of AS
page frame 0 of physical memory
page frame 1
page frame 2
page frame 3
page frame 4
page frame 5
page frame 6
page frame 7
Virtual Memory
Physical Memory

Paging

- Divide physical memory and virtual address space into units of single **fixed size**
  - On seasnet, page size is 4K (see it for yourself! `getconf PAGE_SIZE`)
  - Typically called page frame
- Treat the virtual address space in the same way
- Store data in each ***page*** in virtual address space to ***page frame*** in physical address
- How to translate from page to page frame
  - Page Table: per process data structure that stores address translations of virtual pages to physical page frames
  - Page Table Entry



Address Translation with Paging



Translate virtual address `000001 0111011110` to physical address

Virtual Page Number: `000001`
Offset: `0111011110`

Lookup Virtual Page Number (VPN) in page table, get Physical Frame Number (PFN) `000110`

Physical Address
`(PFN << 10) | offset = 000110 0111011110`

Paging v.s. Segmentation

- Segmentation
  - Specify arbitrarily sized ranges of the address space
  - Address space ranges can be used for a particular purpose, such as code segment or stack
- Paging
  - Divide allocated memory space into smaller pieces of the same size
  - Allow the virtual memory management system to load, relocate, and otherwise manage the space more flexibly
- Segmentation and paging can exist in the same system
  - Segmentation specifies address that are valid/legal
  - Paging allows OS to map small sections of virtual address ranges in physical memory

Translation Lookaside Buffer (TLB)

- TLB caches recently used pages
- TLB miss (accesses to virtual addresses not listed in the TLB) trigger a page table lookup
  - The cache entry is then added to TLB for future access
  - Huge performance penalty
- TLB gets flushed whenever a context switch happens
  - Why? Different processes have different address space

Page Fault

- When a process tries to access a page of virtual address space that is **not mapped** onto a page frame of physical memory
  - E.g. access an address that is invalid (valid bit in Page Table Entry)
- Disambiguate with *segmentation fault*: when a process tries to access an **invalid or illegal** memory address
  - E.g. writing to an address that is read only (R/W bit in Page Table Entry)
- What could happen to a process experiencing page fault?
  - Nothing: e.g. on demand paging, transparent to process
  - Receive signal `SIGSEGV`: e.g. access unmapped memory

On Demand Paging

- Why on demand paging?
  - Kernel doesn’t have to load all pages into physical memory
  - Load pages when a process actually uses it
  - Improve memory locality
- How to implement it?
  - Mark the virtual pages not present in physical memory (**present bit** in Page Table Entry)

On Demand Paging, cont’d

- How to load page on demand?
  - Hardware finds out that the page is not present in physical memory, generates interrupt
  - Traps to kernel, and control is transferred to **page fault handler**
  - Checks permission and determines which pages are needed
  - Allocates physical page frames
    - Load from disk (file or swap)
  - Update Page Table Entries with allocated physical page frames and setting appropriate permission bits
  - Resume execution

Process Memory Layout, revisited

```
$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:07 131133                     /usr/bin/cat
0060b000-0060c000 r--p 0000b000 08:07 131133                     /usr/bin/cat
0060c000-0060d000 rw-p 0000c000 08:07 131133                     /usr/bin/cat
01577000-01598000 rw-p 00000000 00:00 0                          [heap]
7f62049a6000-7f620aecf000 r--p 00000000 08:07 661772             /usr/lib/locale/locale-archive
7f620aecf000-7f620b087000 r-xp 00000000 08:07 261432             /usr/lib64/libc-2.17.so
7f620b087000-7f620b287000 ---p 001b8000 08:07 261432             /usr/lib64/libc-2.17.so
7f620b287000-7f620b28b000 r--p 001b8000 08:07 261432             /usr/lib64/libc-2.17.so
7f620b28b000-7f620b28d000 rw-p 001bc000 08:07 261432             /usr/lib64/libc-2.17.so
7f620b28d000-7f620b292000 rw-p 00000000 00:00 0
7f620b292000-7f620b2b3000 r-xp 00000000 08:07 261434             /usr/lib64/ld-2.17.so
7f620b487000-7f620b48a000 rw-p 00000000 00:00 0
7f620b4b2000-7f620b4b3000 rw-p 00000000 00:00 0
7f620b4b3000-7f620b4b4000 r--p 00021000 08:07 261434             /usr/lib64/ld-2.17.so
7f620b4b4000-7f620b4b5000 rw-p 00022000 08:07 261434             /usr/lib64/ld-2.17.so
7f620b4b5000-7f620b4b6000 rw-p 00000000 00:00 0
7ffda2887000-7ffda28a8000 rw-p 00000000 00:00 0                  [stack]
7ffda29ad000-7ffda29af000 r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

Concurrency

- Parallel computation ➔ potentially dramatic increases in throughput
  - I.e. mergesort by dividing to 10 arrays, **sort each array in parallel**, then merge pairwise
- But parallel computation w/o proper synchronization ➔ useless
  - I.e. one thread blindly starting to merge arrays before waiting for sorting to complete

Concurrency - Toy Example

- Two threads both decrementing bank account balance by 1
- *a = *a - 1
  - After examining hypothetical execution order below, what's actual final value of ****a****?

Thread 1	Thread 2
movq 0xdeadbeef, %rax	
	movq 0xdeadbeef, %rax
	addq $-0x1, %rax
addq $-0x1, %rax	
movq %rax, 0xdeadbeef	
	movq %rax, 0xdeadbeef

Concurrency

- Classic examples of operations with potential synchronization problems (race-conditions):
  - Different **threads** accessing the same **data structure** (ie. linked list)
  - Different **processes** writing to the same **file**
  - Different **clients** modifying a single **table** on a database server

- **True/False**: race-conditions cannot arise on a single-core CPU because instructions can only be executed sequentially

Concurrency

Follow-up: assuming the same single-core architecture, propose one change at the kernel level such that inter-thread race conditions will never occur.

Concurrency

- Goal of synchronization: protect access to resources so that data always appears to be in consistent state
- To do this we need **critical sections**, code segments where only one thread may be running at a time
- Critical sections ensure two forms of **atomicity**
  - Before/after atomicity - operations do not step on one another, no mingling of intermediate steps
  - All/nothing atomicity - from the perspective of other threads, an operation (no matter how complex) appears to be either done or not done, <u>never half finished</u>

Concurrency: Locks

- Critical sections can be enforced using **locks**
  - i.e. pthread_mutex_lock() in C
- Entrance to critical section only after thread acquires lock
- Other threads cannot enter until lock released

Concurrency: Lock Implementation

- 4 evaluation measures of locks
  - Correctness
  - Progress - potential for deadlock?
  - Fairness - can some thread never acquire lock?
  - Performance - is CPU overhead of using the lock minimized?

Concurrency: Lock Implementation

- Disable interrupts
  - Usually used in interrupt handlers themselves to enforce **re-entrancy**
  - Correctness: works unless on multi-core architecture, impossible in user-mode code
  - Progress: can deadlock if some resource takes forever (lock never released)
  - Fairness: long disables lead to potential monopolization of CPU, short ones OK
  - Performance: overhead for the disabling instruction itself is small, but long lock-could degrade system-wide performance

Concurrency: Lock Implementation

- A simple variable
  - I.e. 1 for locked, 0 for unlocked
- But how to ensure setting of variable itself is atomic?

Lock Implementation: Hardware instructions

- Test and set: one way of performing atomic variable updates
- Atomically do 3 steps:
    - Record old value
    - Set new value
    - Return old value
- We call testAndSet(1) ← try to lock the lock to 1
- If it returns 0 → old value was 0 → we have the lock and it's set to 1
- If it returns 1 → old value was 1 → value unchanged: we don't have the lock

Lock Implementation: Hardware instructions

- Compare and swap
- Atomically:
    - If value is what's expected, set to new value
    - Return old value
- We call compareAndSwap(0, 1) ← try to lock the lock to 1, expecting it to be 0
- If it returns 0 → old value was 0 → we have the lock and it's set to 1
- If it returns 1 → old value was 1 → value unchanged: we don't have the lock

Lock Implementation

- The aforementioned hardware tools allow us to implement different kinds of locks
- Spinlock
  - while (locked) { try lock }
  - Simple, desirable if contention low, so overhead of sleeping/waking from blocking locks is avoided
  - Waste of CPU clock cycles “spinning” waiting for lock to be freed
- Spin & yield
  - Spin only few times, then yield (context switch)
  - Potential context switch overhead

Lock Implementation

- Condition variable
  - When a thread must wait for an event to occur before proceeding
  - Instead of spinning, use condition variable
  - **Blocks** (sleeps) until variable changes, resulting in a signal being sent to OS

Good luck!

Sign-in https://goo.gl/RT38gH
Slides https://goo.gl/fuQVir

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
    - Location: ACM/UPE Clubhouse (Boelter 2763)
    - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.