# Section 8: Lab 4 Intro

CSE 451 20wi

section 8: 2/27/2020

# Announcements

- Lab 4 proposal due tomorrow
- Lab 4 due on the Friday before final week
  - No late days!

# Potential Projects Walk Through

# Adding a system call

- Take a look at how osv-public commit adding halt system call
- Files changed:
  - include/lib/syscall-num.h: define a new system call number for the new system call
  - include/lib/usyscall.h: declare function signature of the new system call for user processes
  - arch/x86_64/user/usyscall.S: define a new system syscall stub for user using macro
  - kernel/syscall.c: declare and define a new system call handler. add the handler into the array of system call handlers.

# User Threading Library

- User library that schedules user threads and manages user thread lifecycle
  - start/create, exit, yield, join
- Why is this useful?
  - very lightweight, process has control over scheduling decisions
  - great for applications that handles many small tasks
- Getting started
  - Simple version: allowing each user level thread run to completion (no context switching)
  - look at how kernel threads and its scheduling is done
  - define a user level thread abstraction (thread control block) and a scheduler
  - then support thread_yield, which means you have to allocate user stack for each user thread (can use malloc), and support context switching
  - all user level threads still run on top of a single kernel thread
  - Advanced version: use signals to allow preemption

# Kernel Threading Library

- Allow user processes to have more than one kernel thread
- Extend kernel/thread.c to allow joining kernel threads
- Why is this useful?
  - allow a process to have more than one execution unit in the kernel scheduler
  - greater share of CPU time
- Need to review previous codes to properly handle concurrency: we previously assumed that there's only one thread per process, but it's not true now.
  - E.g: two threads can access the file descriptor table at the same time!

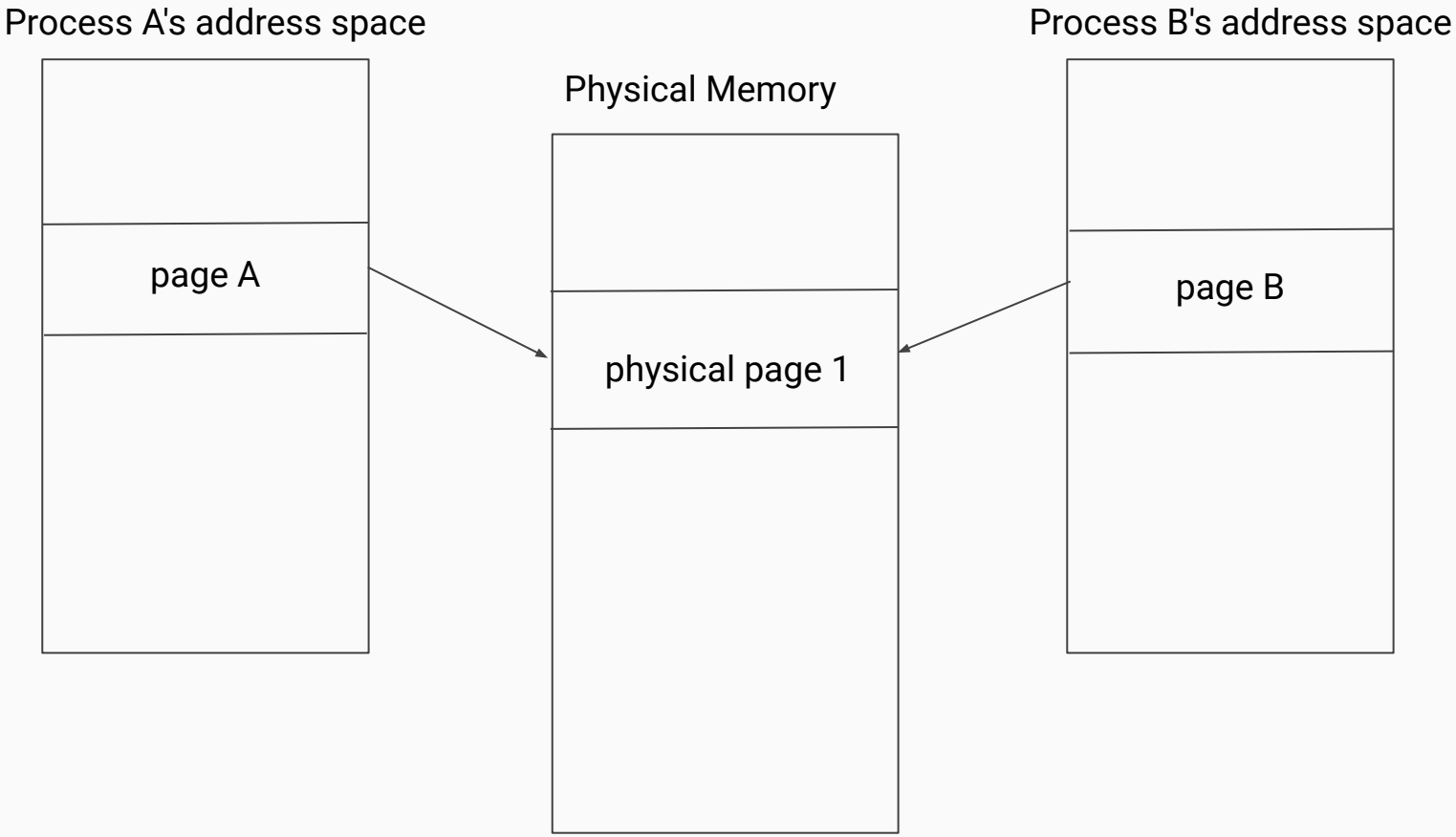
# Kernel Threading Library

- Getting started
  - first study kernel/thread.c implementation
  - add system calls to allow process to interact with kernel threads
    - new system calls that would call into kernel/thread.c (just like sys_wait => proc_wait)
  - user process should be able to pass a pointer of the starting function and its argument (similar to pthread_create)
  - each kernel thread needs to have its own kernel stack and user stack
    - can either assume each kernel thread uses no more than 2 pages of stack and give new threads different pages of the stack memory region
    - Or just create a new memregion for the new stack region, allocate it right below the previous stack limit.
  - a kernel thread should be able to wait for another kernel thread in the same process

# Named Shared Memory

- Pipe allows processes to communicate via a kernel buffer, but that means all communication needs to go through kernel overhead
- Instead, we can let process to communicate through shared memory directly
- Shared memory meaning multiple processes have the same physical page(s) mapped into their address space, and can read/write to that shared memory
- Named shared memory = giving a shared memory region a name so other processes can refer to it

# *Shared memory*



# Named Shared Memory

- Getting started
  - define a set of shared memory system calls: allow processes to create a shared memory region of size X (should be a multiple of page sizes) with a given name, map a shared memory region into its address space, unmap it, delete a shared memory region
  - you get to define the behaviors of these system calls (for example, you can prevent someone from deleting a shared memory region if other processes still have it mapped in their address space)
  - need to track names with shared memory regions (global data structure)
  - to start with, you can restrict the size of shared memory and allocate a physical page when it's created, later you can allow for larger shared memory and allocate the pages on demand
  - Advanced version: implement a user library to avoid race conditions in shared memory

# Signals

- Enable software interrupts, allow process to communicate via a concise signal (just a number). More info here.
- Each signal you support should have a default signal handler - code that gets run when a process receive this signal
- Signal is non-blocking
    - The process sending signal returns as soon as the signal is queued in the receiver process, the receiver process might not have executed the signal handler yet

# Signals

- Getting started
  - define a set of signals to support and the corresponding handlers
  - add a system call to let a process send signal to another: signal(signal number, pid)
  - figure out how to make signaled process execute a handler
- Project progression
  - Simple version: the kernel handles signals
  - Advanced version: allow user process to provide a signal handler
    - The handler should run in user mode to avoid security issues
    - hint: at the end of sched_sched, scheduler makes a thread execute an additional task before resuming its previous execution
  - Allow process to mask certain signals (not process certain signal numbers)

# Priority Scheduling (or other scheduling algorithms)

- Scheduling decisions are purely based on priority
  - schedules the thread with highest priority in the ready queue.
  - when a new thread is added to the ready queue, if it has a higher priority than the current running thread, then current thread should yield and let this new thread run.
  - when a sleeplock or condition variable wakes up its waiters, it needs to wake up the waiter with the highest priority.
- Need to allow threads/processes to change their priorities
  - A thread may raise or lower its own priority at any time, but lowering its priority such that it no longer has the highest priority must cause it to immediately yield the CPU
- Additional problems: priority inversion (solve using priority donation) - see 2.2.3 priority scheduling

# Priority Scheduling (or other scheduling algorithms)

- Getting started
  - extend thread interface and system calls to allow changing priority of a thread/process
  - yield when the running thread is no longer the highest priority thread (see previous slide)
  - test and make sure priority scheduling is respected
- next step (at least need to attempt)
  - implement advanced scheduling algorithm (ex. Multi-level feedback queue)
  - implement priority donation for sleeplock to fix priority inversion problem
    - when a high priority thread wants to grab a lock, but the lock is held by a low priority thread, the waiting thread can donate its priority temporarily to the low priority thread so it can get a chance to run and release the lock.
    - thread with donated priority must restore to its previous priority when it releases the lock. priority donation can be nested.

# Q&A

Man pages are helpful!
Looking at what linux does also helps a lot!