CS 3214 lecture # 13

take-home test #1 Thursday

.h → (preprocessor) → .c → (compiler) → .s → (assembler) → .o → (linker) → executable → running in memory

.a [.o .o .o] → executable

dynamic linker (ld-linux.so) → running in memory

.so → running in memory

(indirection) { PLT procedure linkage table
GOT global offset table

static vs dynamic

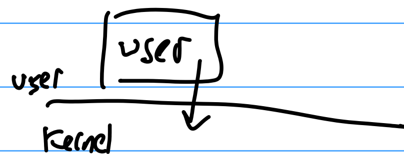
dynamic { + dynamic can save mem
+ no need to recompile to update library

static { + self-contained

[libc.so] ←

libc.a

/proc/<pid>/maps



Processes : a running program

user mode vs. kernel mode

dual-mode operation

system calls

process states

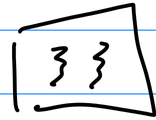
lifecycle management fork/exec/wait

- pipes & file descriptors
- signals

[Process] → [compiling / linking]

what it looks like while we're running

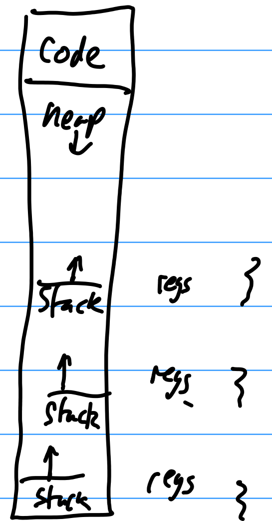
how we build what will become a process



multithreading   thread = logical unit of execution

concurrent execution in SAME address space

- I/O computation overlap
- distributing a problem
- use multiple CPUs/cores
- fault tolerance?
- multiple clients in web server
- the OS ← the first concurrent program

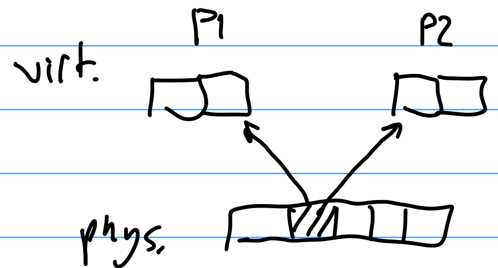


threads share address space
- code + data
- open file descriptors

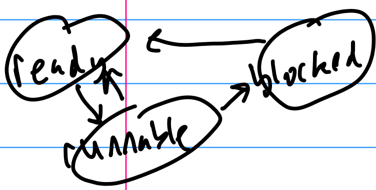
threads do NOT share:
- stack (local variables, return addresses, etc.)
- CPU registers PC, SP

processes: share
- some mem (shmem)
- inherited fds
- kernel /machine resources
- files on disk

not share
- address space
- stack, regs

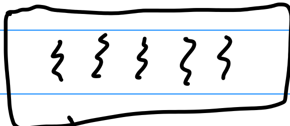
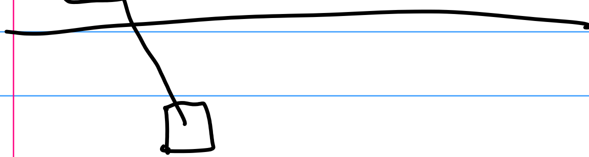


clone (..., SHARE_VM)



Does the kernel need to help implement threads? NO

user threads
1. save/restore stack + registers (setjmp/longjmp) ← save regs
2. cooperative scheduling
non-preemptive

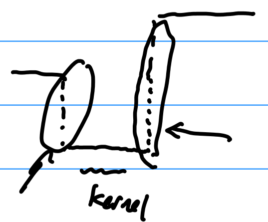



user-level threads
coroutines
green threads

yield - give up CPU to another thread
directed "yield to thread n"
undirected "yield to ?"

Higher-level languages
yield temporary result
yield "promise" obj for async I/O async/await

Pros vs Cons
+ no need for syscalls
+ cooperative means blocking/inf loops are bad
+ ctxt switch is fast
- can't run in parallel



too many threads = inefficient

Kernel-supported threads