2) **ProcessManagmnt**: **kernel vs user**: kernel: privileged instructions (set mode bit), direct access to all of mem, to dvices. **Sys calls**: only interface from program 2 os. **Traps**: generated by cpu as result of error, like involuntary sys call. **Interrupts**: generated by device that needs attention. **Process**: program in execution, execution stream in context of process state. Process state: registers, address space, heap stack, open files. **Address space**: static: code & some global vars; dynamic: stack & heap. **Why dynamic alloc**: dk memory needed at compile time, if static must be pessimistic, recursive procedures, complex data structures. **Stack**: efficient, simple: pointer separates allocated & freed space, increment pointer 2 alloc, decrement 2 free, no fragmentation, management done automatically. **Heap**: allocate from random location: has allocated areas & holes, order of alloc & free unpredictable; work for all ds, alloc slow, fragmentation. **PID=fork(**): creates identical copy of parent, in parent returns pid of child; in child returns 0. **Exec**(filename): loads executable from file w filename. **Wait**(): wait for 1 of children to terminate. **Exit**(): terminate process. Shell operation: new command line(!=logout): shell forks a new process & waits, child execs prog on command line. Linux processes tree: Boot: 1st process after boot is init; user logs in: init forks & wait, child execs shell; user runs make: shell forks & waits, child execs make; another user: init forks & wait, child execs shell; make runs gcc: make forks & wait, child execs gcc; gcc finishes: gcc exits, make returns from wait; 2nd user logs out: csh exits, init returns from wait; make runs cp: make forks & wait, child execs cp. **Process** (for os): computes (use cpu) or does IO (use devices); Issues (single): low utilization, long wait times. **Multi**: state diagram: new->ready->running->terminated/waiting[IO]->ready[IO completion]. **Process switch**: switch from 1 process running on cpu to another, st can later switch back to process currently holding cpu. Process consists of code stack heap (in process private locations) & registers, mmu info (in shared locations). Ex: P1->P2: save registers-> PCB[P1].saveArea, restore PCB[P2].saveArea->registers. **Process scheduling**: new -> ready -(scheduler)> running -> terminated/ -(IO)> waiting –(IO completion> ready. Non-preemptive: process only voluntarily relinquishes cpu, can monopolize; preemptive: may be forced off cpu at any time. How to run: At end of handlers for • System calls• Interrupts• Traps

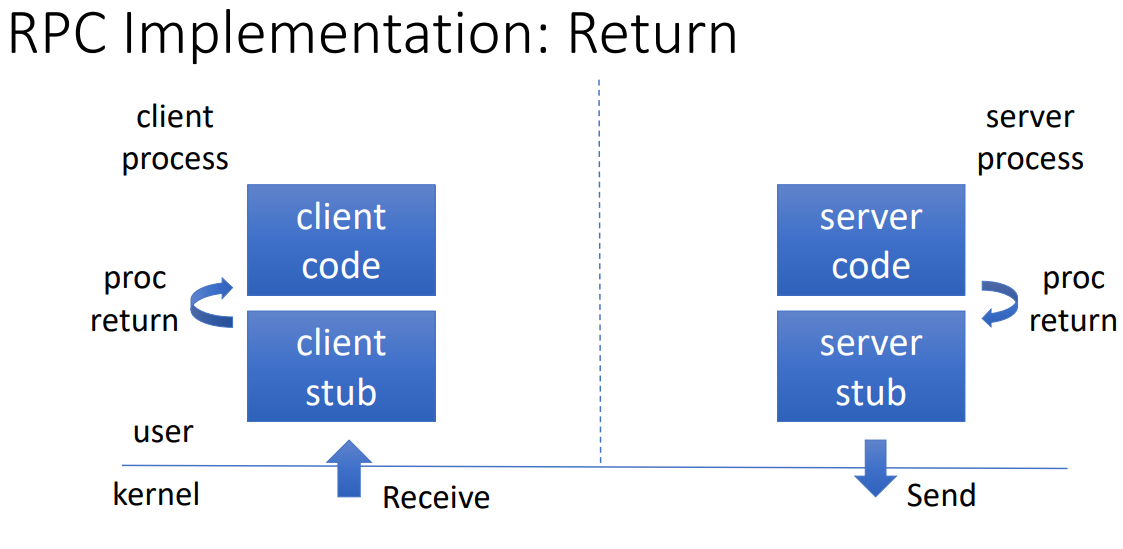
• Scheduler runs: decides on process to run• Switches to a new process• Sets another timer. Scheduling metrics: minimize turnaround time: completion time- arrival time; minimize response time: initial schedule time- arrival time; min wait time; max throughput; max resource utilization; min overhead: reduce #context switch; max fairness. **For interactive**: short resp time; for batch: high throughput. **FCFS**: low overhead (few sch events), good throughput, uneven resp time, proc can monopolize cpu; **SJF**: good resp time for short jobs, starvation; **RR**: compromise for long & short, no need to know job length, if dt too small: many sch events, good resp time, low throughput; too large: opposite. 3) sync primitives: **process**: provide separation: memory sep (no shared data), suitable for coarse-grain interaction; **threads**: share mem, suitable for tighter integration. Shared data: adv: many threads can r/w it; dis: data races: unexpected access to shared data. Nondeterminism: Concurrency leads to non-deterministic results• Different results even with same inputs• Race conditions, Whether bug manifests depends on CPU schedule! Multithreading: 1. Divide “work” among multiple threads & 2. Share data• Which data is shared?• Global variables and heap• Not local variables, not read-only variables• Where is shared data accessed?• Put shared data access in critical section; Want 3 instructions to execute as an uninterruptable group (be atomic). **Mut exc**: • Prevents simultaneous access to a shared resource (shared mem region). **Synchronization**: sw: Monitors, Semaphores, Condition Variables, locks(mutex); hw: loads, stores, test&set, disable interrupts. **POSIX**: create(), exit(), join(): • Join with a terminated thread• Waits for the thread specified by thread to exit, mutex\_lock(): • If lock is held by another thread, block• If lock is not held by another thread: • Acquire lock• Proceed; mutex\_unlock(): release lock. **Cond vars**: • Used when thread A needs to wait for an event done by thread B• A waits until a certain condition is true• First test condition, • If cond not true, call cond\_wait() • A blocks until cond true • At some point B makes the cond true • Then B cond\_signal(), which unblocks A. **Semaphores**: • Ashared, non-negative counter.

• Two primary operations:

•Wait -> attempts to decrement the counter; blocks when counter is 0 •Post (or Signal) -> attempts to increment the counter, changes are atomic. Uses: Mut excl •A sem w counter initialized to 1 is eq 2 a lock; Bound concurrency: •Only allow X threads out of N to proceed. Producer-consumer problem •More complex use of semaphores. Sem\_init(sem\_t \*sem, int pshared, unsigned value): •Initializes the semaphore \* sem. initial val of the sem is value. •If pshared 0, sem shared among all threads of a process. •If pshared not 0, sem shared but should be in shared memory. sem\_wait(sem\_t \*sem): •If sem has val> 0, decrement val by 1 •If sem has val 0, caller will be blocked (busy-waiting or more likely on a queue) until sem has val>0. sem\_post(sem\_t \*sem): •Inc val of sem by 1. •If threads blocked waiting for sem, 1 of them (at random) will return successfully from its call to sem\_wait(); sem val is immediately decremented. **Monitors:**•Collection of vars and fcns •Threads can only access monitor functions •Vars private to monitor •Only 1 process at a time can execute code inside monitor. Disabling interrupts: •Lock implementation code executed in kernel mode •Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable; why disable: •Avoid interruption between checking and setting lock value •Otherwise 2 threads could think they both have lock **4) Multiprocess comm**: **Process Pool**

•Create worker processes during initialization

•Hand incoming request to them. Worker = code, globals, heap, stack, registers, pc. **Changes**: Amount of work on server per request • Receive network packet •Run listener process •Send message to worker process (cheaper) •Read file from disk•Send network packet. IPC: os support to allow process 2 manage shared data thru msg passing, remote procedure calls (rpc); needed btw client/server, btw cooperating processes. **Msg passing**: by value comm, never by ref, rcver cant affect msg in sender. Blocking send: sender blocks til msg delivered; nonblock send: send returns imm after msg sent, more common. Blocking receive: receive blocks til msg present, more common. RPC Interface •List of remotely callable procedures •With their arguments and return values •Example: file system interface •Open(string filename) •returns int fd. **2 msg types**: call msg: from client 2 server, contains args; return msgs: from server to client, contains ret vals. **Client stub**: sends args in call msg, receives ret vals in ret msg; **server stub**: receives args in call msgs, invokes procedure, sends ret vals in ret msg.



**Automatic stub gen**: • rpcbind: universal addresses to RPC program number mapper •A server that converts RPC program numbers into universal addresses •It must be running on the host to be able to make RPC calls on a server on that machine. Rpcgen: tool that generates remote program interface modules.

•compiles source code written in the RPC language. •RPC language is similar in syntax and structure to C •produces one or more C language source modules, which are then compiled by a C compiler