COM SCI 111 (Operating System Principles)

October 13, 2022

1 9.22 0th

```
$ ls -l big
-rw-r--r-- 1 eggert faculty 9223382036854775000 Sep 22 11:31 big
$ grep x big
$ time grep x big
real 0m0.009s
```

- grep scans at 10^{21} bytes/s (8 Zb/s) searching for an existent line in the file
- so it has to be doing smth other than sequentially searching thru the file
- “grep cheats”
  - “i know it cheats cuz i put the code in it” - eggert 2022
- “i hope you gain the intuition to know when is it ok to cheat and when is it not”
- paul eggert
- https://web.cs.ucla.edu/classes/fall22/cs111
- prereqs: cs32 <- c++, algs, data structurues / cs118 networking 33 <- computer org, machine code, etc. |-> cs111 - . . . 35l <- shell, python, scripting, . . . / . . .
- cs131 programming languages
- cs151b architecture
- textbooks
  - ad os 3ep (2018)
  - sk systems (2009)

1.1 whats a system

- *oxford english dictionary* (1928)
  - i. an organized or connected group of objects
  - ii. a set of principles, etc.; a scheme, method
  - from ancient greek $\sigma\iota\sigma\tau\eta\mu\alpha$(roots, “set up w”)
- *principle of computer science design: an introduction* (2009)
  - smth that ops in an env and the boundary betw the 2 is called the interface
    - interface v important
    - system built from a lot of subsystems
    - standard design: decompose big system
    - often time its useful to have multiple views of the same system
      - sometimes we can look at the system from a diff viewpoint and come up w a completely different set of subsystems

1.2 operating system

- *american heritage dictionary* (2000)
  - software designed to control the hardware of a specific data processing system in order to allow users and application programs to make use of it
    - claims os is very system dependent, not true
- *encarta* (2007)
  - master control program in a computer
- *wikipdeia* (2016/8)
  - system software that manages computer, hardware, software resources, and provides common services for computer programs

1.3 goals of an operating system

- protection (of apps, data, . . .)
- performance (from users view)
- utilization (from check-writers view)
- robustness
  - does this operating system do well when given problems out of the ordinary
- flexibility
- simplicity / ease of use
- portability w diff hardware
- scalability
- safety

1.4 what are our main tools?

- abstraction + modularity
  - abstraction: look at the big pic of the system from a particular viewpoint and discard details to understand everything abt that aspect of the system
  - modularity: splitting up a big problem into little problems
    - since the diff in writing a program scales worse than linearly,
  - interface vs implementation
  - mechanisms vs policy
    - policy: high level concept in which u say what u want
      - "i want my interactive processes to have higher priority than background batches"
    - mechanism: how u actually get that stuff to work
- measurements + monitoring
  - measurements: measure how well ur system is working
    - performance + correctness

  - monitoring: monitor measurements, do smth w them
- *operating systems: three easy pieces*: main problems in os
  - virtualization: how to build efficient + effective virtual systems
  - concurrency: interacting, simultaneous tasks
  - persistence: data survive failures in hardware, software
  - more
    - security

1.5 a bad os interface

```
char* readline(int fd);
```

- assumes infinite resources

2 9.23 0f

- kernel: lowest level of the os
  - decides what resources are available to apps (thus protecting the system)
  - provides layer of abstraction so that apps dont have to deal w hardware
  - todo: see notes.md.old

2.1 kernel modules

- are pieces of code that can be loaded and unloaded into the kernel upon demand
- they extend the functionality of the kernel without the need to reboot the system
- why not just add all the new functionalities into the kernel image
  - would be bloated
  - security implications
- modules are stored in `/usr/lib/modeuls/kernel_release`
- to see what kernel modules are currently loaded use

  ```
  lsmod
  ```

  ```
  cat /proc/modules
  ```

- example

  ```
  #include <linux/module.h> /* needed by all modules */

  #include <linux/kernel.h> /* needed for KERN_INFO */
  ```

```c
int init_module(void) {
  printk(KERN_INFO "hello world 1.\n");
  return 0;
}

void cleanup_module(void) {
  printk(KERN_INFO "goodbye world 1.\n");
}
```

- `printk`
    - was not meant to communicate info to user
- to load a module: `sudo insmod <module_name> [args]`

```
sudo insmod proc_count.ko
```

- to unload a module: `sudo rmmod <module_name>`

```
sudo rmmod proc_count
```

3 9.27 1t

3.1 simple os architecture

- todo: insert pic
- user mode code: apps, libraries / c-lib
- kernel mode: you can execute instructions here
    - e.g. `inb`
    - kernel-app interface: “imaginary instructions” that are not real
        - x84-64 instructions `ret`: “`rip = *(--rsp)`” in user mode
        - `syscall 35`: “`os[35]()`” in kernel mode
            - `syscall`s cannot be called in user mode, kernel will figure it out

```c
char *readline(int fd)
```

- `fd`
    - `0` - `stdin`
    - `1` - `stdout`
    - `2` - `stderr`
    - `>2` - other files
- points to a newly allocated buffer, e.g. `"ab\0xyz\n"`
- problems
    - unbounded cost
    - allocates memory for the application
    - large potential for memory leaks & bad pointers

    - probles with giving the job to kernel
        - force same memory mgmt on all apps
        - syscall overhead

```
ssize_t read(int fd, void *buf, size_t bufsize);
```

- it is now the applications job to figure allocation out, kernel doesnt care
- there is now a limit on read size
- comes at a cost: a program that counts the number of lines in a file becomes more complicated although it has nice properties for the kernel

3.2 problems with designing operating systems

- waterbed effect (tradeoffs): general problem in systems
- incommensurate scaling
    - economies of scale (pin factory)
        - if u just want a few pins then just go to local blacksmith
        - if u want 10mil pins u should create a pin factory
    - diseconomies of scale (star network)
- emergent properties (as u scale)
    - qualitative instead of quantitative changes
    - ucla in school network used for music pirating
- propagation of effects
    - 2 features that independently work may not work when combined
    - e.g. msft invented shift-jis to encode japanese characters
        - 2-byte encoding w top bit on = 2^15 characters
        - other feature: file names `C:\abc\def\ghi.txt`
        - combination doesnt work because 2nd byte of shift-jis character can be anything (could just happen to be `'\'`)
        - fixed by moving into kernel (complicating the os)
- complexity
    - moores law

3.3 app: count words in a file

- todo: add pic
- power button = count number of words and put it on screen
- historically called a standalone program
    - operates without benefits of an os
- modern desktop
    - todo: add pic

3.4 uefi: unified extensible firmware interface

- os-independent way to boot
- efi format for **bootloaders**
  - bootloader: find a program in 2ndary storage, copy it into ram and `jmp` to it
  - can be chained, not uncommon to see 3 or 4 chained bootloaders
    - for portability and stuff
    - each bootloader can be more complicated than the next one until one that knows how to boot linux
- guid: globally unique identifier
  - for partitions
  - 128-bit integer
- guid partion table
- uefi boot manager (in firmware)
  - read only but configurable via parameters in some sort of nvram (nonvolatile ram)
- can read gpt tables
- can access files in vfat format etc
- can run code in efi format
- 6 phases

3.5 coreboot

- more hardware-specific / lower level
- no large boot manager (but a small one)
- 4 phases
  - test rom (find out where it is) + flash / disk
  - test ram, early initialization of chipset
  - ram stage: init cpu, chipset, motherboard, devices, etc
  - load payload

3.6 intel core i3-9100

- supports an older, simpler way of booting
- 6 mib l3 cache, 3.6 ghz + 4 gib ddr3 sdram + 1 tb flash sata + intel uhd graphic
- sata: serial ata
  - 7-conductor connector
  - ata (pata): advance technology attachment (16-bit connector in parallel)
    - sequentization is the bane of parallel
  - before ata = ide: integrated drive electronics (western digital, 1986)
- x86 boot procedure
  - 1 mib of physical ram cpu can access

    - cpu starts off in real mode = no virtual memory
  - initial program counter `ip` points to rom @ `0xffff0` = $2^{20} - 16$
  - program in rom = bios: basic input output system
    - "horrible misdesign"
    - originally user apps ran in real mode, calling subroutines in bios
      - only protection is rom being read only
    - library + kernel + mini os basically
    - bios tries to do the 4 steps in coreboot
      - run in cache only mode
      - step 4: looks for a device containing a particular bit pattern in its first 512 bytes (sector) (mbr: master boot record, 446 bytes of x86 machine code, 64 bytes of partition table (list of 4 pieces of drive, takes role of gpt), 2 bytes of signature `0x55 0xaa` or `0xaa55`)

4 9.29 1th

4.1 less modular os

- one happy program
- uses function calls
- count lines in a file only via function calls + machine instructions at lowest level
- last time we got 426 of bytes of machine code into `0x7c00`
- file is in flash drive
- bootloader reads word count program, say 20 sectors (of 512 bytes), into ram, say `0x1000`

```
static void read_ide_sector(long secno, char *addr) {
  while ((inb(0x1f7) & 0xc0) != 0x40) continue;
  outb(0x1f2, 1);
  outb(0x1f3, secno);
  outb(0x1f4, secno >> 8);
  outb(0x1f5, secno >> 16);
  outb(0x1f6, secno >> 24);
  outb(0x1f7, 0x20);
  while ((inb(0x1f7) & 0xc0) != 0x40) continue;
  insl(0x1f0, addr, 128);
}
```

- `inb` reads from io registers
- `0x1f7`: status + control
- first `while` loop waits for drive to be ready
- `insl`: "insert long", read 128 words (512 bytes) from `1f0` to `addr`

```
static unsigned char inb(unsigned short port) {
  unsigned char data;
  asm volatile("inb %0, %1" : "=a" (data) : "dN" (port));
  return data;
}
```

4.2 i/o

- programmed i/o (pio): special instructions for io
- memory-mapped i/o: more popular, use ordinary load / store instructions `mov_ _`
- intel: flash drive uses pio, monitor uses memory-mapped

```
void write_integer_to_console(long n) {
  unsigned short *p = (unsigned short*) 0xb8014;
  while (n) {
    *p-- = (7 << 8) | ('0' + n % 10);
    n /= 10;
  }
}
```

- at `0xb8000`, 2 bytes per char, 1st byte being format (7: gray on black), 2nd byte being ascii, 80×25 grid
- code above prints

```
static int isalpha(int x) { return 'a' <= x && x <= 'z' || 'A' <= x && x <= 'Z'; }

void main(void) {
  long words = 0;
  bool in_word = false;
  long secno = 100'000;
  for (;; ++secno) {
    char buf[512];
    read_ide_sector(secno, buf);
    for (int j = 0; j < 512; ++j) {
      if (!buf[j]) {
        write_integer_to_console(words + in_word);
        while (true);
      }
      bool this_alpha = isalpha((unsigned char) buf[j]);
      words += in_word & ~this_alpha;
      in_word = this_alpha;
    }
  }
}
```

- performance issues

  - read 1 sector at a time: change to 255 sectors
  - bus contention (controller ↔ cpu, cpu ↔ ram): dma (direct memory access), controller ↔ ram, cpu sends instruction to controller telling where to copy data to
  - cpu and controller doing work disjointly: double buffering—cpu issues command to read sector $n+1$ *then* cpu counts words in sector n = overlapping work

5 10.4 2t

5.1 how not to have an os standalone program

- standalone program
  - pro: embedded systems
  - con: double buffering api is more complex
  - con: multitasking

```
read_ide_sector(...) {
  while (inb(...) & ...) {
    --> schedule(); <--
  }
}
```

  - con: dma (direct memory access)
  - want separation of concern, guy writing word count program only focuses on counting words
  - need better modularity

5.2 what's wrong with fuction call modularity for apps

- too much pain to change implemetation
- too much pain to reuse parts of os in other apps
- too much pain to run simultaneous apps
- too much pain to recover from faults
- not enough insulation between apps

5.3 advantages of modularity

- assume bugs ∝ (k)loc ((thousand) lines of code)
- assume cost to find & fix a bug ∝ kloc
- cost to fix all bugs: $O(\text{kloc}^2)$
  - not accurate: more bugs may appear, bugs appear superlinearly to kloc, bugs get harder to fix, etc
- modular: assume k modules, bugs are isolated $\rightarrow O\left(k \cdot \left(\frac{\text{kloc}}{k}\right)^2\right) = O\left(\frac{\text{kloc}}{k}\right)$

5.4 how can we tell whether our modularization is good or bad?

- performance
  - usually willing to sacrifice, say, 5% to 10%
- robustness: tolerance of faults / failures / errors
  - error: mistake in ur head
  - fault: potential problem in code
  - failure: observable behavior that is wrong
    - e.g. dereferencing a null pointer
- neutrality / flexbility / lack of assumptions
- simplicity (of use / to learn)

5.5 mechanism for modularity

0. no modularity
1. function call modularity (call and return instructions)
   - callee can modify
   - callee can loop forever
   - callee can overslow the stack
   - callee can mess w wrong devices
   - callee can execute invalid instruction
   - soft modularity: if callee and caller are both well-behaved / bug-free
   - we want hard modularity: barrier

5.6 3 fundamental system abstractions

1. memory
   - `write(addr, value)`
   - `value = read(addr)`
   - ram, secondary storage
   - issues: thruput, latency, word size, volatility, coherence with caches
2. interpreter
   - `ip` instructor pointer + `ep` environment pointer + repertoire (instruction set)
   - in x86: `rip` and `rsp`
   - above for normal execution
   - interrupts: when normal execution is disturbed
3. link
   - think of 2 modules as different devices, send signals from one to another thru a link

5.7 2 major ways to get hard modularity

1. client / service
   - client and server w a link
   - if client wants work done, send signal thru link and wait
   - client

     ```
     send(factn, {"!", 5});
     receive(factn, response);
     if response.code == ok:
       print(response.val)
     else:
       print("error", response.err)
     ```

   - service

     ```
     for (;;) {
       receive(factn, req);
       if (req.op.code == "!") {
         for (int i = 1; i <= req.n; ++i) {
           a *= i;
         }
         response = { "ok", a };
       } else {
         response = { "bad", 0 };
       }
       send(factn, response);
     }
     ```

   - pro: limits error propagation
     - client / server dying doesnt affect the other
   - pro: no shared state
   - pro: even if service loops forever, client can still make progress
     - timeout on `receive`
   - con: more setup hassle (configuration overhead)
   - con: more resources (in the simplest case, need 2 cpus)
   - con: latency / thruput / reliability (thru network)
2. virtualization
   - using interpreters
   - simplest way: write an x86 emulator

     ```
     int epi, eax;

     for (;;) {
       char i = *epi++;
     ```

```
    switch (i) {
      case ...:
        add(eax, ...);
    }
  }
```

- "client" code: interpreted
- "service" code: functions called by interpreter
- pro: can put whatever checking code for safety
  - e.g. checking pointers before dereferencing
- con: performance (2x - 10x)
- for better speed
  - virtualizable hardware
    - user-mode instructions: `addq`, `call`, `ret`, `jlt`, ...
      - run at full speed in a virtuazlized (hw interpreter) program
  - kernel-mode instructions: `inb`, `outb`, `insl`, `reti`, ...
    - need them to be rare
  - we need **protected trasfer of control** for hard modularity via virtualization
  - `int`, `0x80`: interrupt, traps in user mode
    - call kernel w
      - `eax` = syscall number
      - `args 1..n`: `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`
    - hw pushes onto kernel stack
      - `ss`: stack segment
      - `esp`: (user) stack pointer
      - `eflags`
      - `cs`: code segment
      - `cip` (code) stack pointer
      - error code: type of trap
    - trap table: goes to code in kernel

6 10.6 2th

6.1 orthogonality

- processes, races
- how you handle files, stream / network io, processes, should not interfere with each other
- api / interfaces should be
  - simple
  - complete
  - composable

6.2 last time

- `INT 0x80` "meta instruction" → pushes onto kernel stack `ip` and `ep`, goes into kernel mode
  - kernel code does the syscall in question
  - `reti` instruction: inverse, pops `ip` and `ep` off the stack into register, gets out of kernel mode
  - `reti` can go to anywhere or run some other process
- seasnet: x86-64
  - usermode does not use `INT` instruction
  - uses `SYSCALL`: uses `INT` but the idea is that it's faster
    - `INT` is slower than a function call
      - pushing 6 words instead of 1
      - have to access ram (kernel stack)
      - make sure cache is right (context switch)
    - attempts to streamline, transfer to kernel mode and left kernel figure out what to use and what to restore
    - `rax` = syscall #
    - args: `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`
    - sets `rip`, etc. to model specific registers
  - one way to implement "syscalls"
- `getpid()`
- `vdso`: virtual dynamicallly-linked shared obejct
  - `ldd /bin/sh`
  - `libc.so.6` ⇒ `/lib64/libc.so.6` shared object code
    - more complcated things such as `open()`
  - `linux-vdso.1` ⇒ kernel memory, readonly for users
    - `getpid()` in here

6.3 processes

- built from virtualizable processor + os = program in execution in an isolated domain
  - safety
  - simplicity
- processes need to access
  - registers: give cpu 1 process at a time
- `cr0` points at page table, maps virtual to physical addresses
- access registers (fast)
- access primary memory:l each process has its own page table
- access io devices (less common): syscall, devices do differ
  - storage: flash, hard drive
    - request / response

    - random access
    - finite
  - stream: keyboard, network
    - spontaneous data generation
    - infinite
  - graphics: high performance
    - kind of a mix
  - need a set of syscall primitives
    - `fd = open("/usr/lib/libc.so", O_WRONLY | O_NOFOLLOW | ..., 0644)`
      - puts a pointer (file descriptor = index) to file in file descriptor table in process table
      - "opaque handle"
    - `close(12)`
    - `read(12, buf, 512)`
      - will return -1, `errno = EBADF` bad file descriptor
      - has an implicit offset, needs to be stored somewhere
        - stored in file description pointed to by file descriptor
        - which then points to actual file
      - `(sed 1q; sed 's/a/b/;) < file`
        - second `sed` should start where first stopped
        - so offset shouldnt live in the `sed` process itself and hence the file description layer between file descriptor and actual file
    - `write(12, buf, 1024)`
    - `n = dup(12)` → `15`
      - same file descriptor
      - by convention, file descriptors 0, 1, 2 are `stdin`, `stdout`, `stderr`
      - `int fd0 = dup(0); close(0); int fd1 = open(...);`
  - limitations
    - access is sequential
      - potential fix: have a different flavor of `read` for storage devices taking an extra argument
      - orthogonality → `lseek(12, 192308, 0)` position read / write pointer at 192308 from file start (`lseek(fd, offset, whence)`), whence: 1 = from file start `SEEK_SET`, 2 = from current location `SEEK_CUR`, 3 = from file end `SEEK_END`
      - `lseek read` / `lseek write` hurts performance → `pread` / `pwrite`: positioning read / write

6.4 next lecture

- process primitives
  - `fork()`

7 10.11 3t

7.1 file descriptor trouble

- use a fd thats closed (or never opened)

  ```
  #include <errno>

  write(47, "xy", 2)
  ```

  - returns -1, sets `errno == EBADF`

- open, but resource not available

  ```
  int f = open(...);
  if (f < 0) { error; return; }
  read(f);
  somefun(f);
  write(1, ...);
  ```

- io error
- end of file (`read` returns `0`)
- errno: lots of `if`s

7.2 process api in posix/linux

- `pid_t fork(void);`
  - `#include <sys/wait.h>`: `typedef int pid_t`
  - returns an integer containing a process id
  - returns `-1` on failure
  - otherwise returns new process id
    - positive integer
  - returns `0` if in child process
  - `int execvp(char const *file, char * const *argv);`
  - `pid_t waitpid(pid_t p, int *status, int options);`: wait for child process to die and store exit status into an integer in parent, returns `p` if successfully `p` to die, otherwise returns `-1` (e.g. wrong `p`, `p` already dead)

    ```
    pid_t p = fork();
    switch (p) {
      case -1: return error();
      case 0:
        execvp("/bin/date", (char*[]) {"date", "-u", NULL});
        return error(); // <-- executed in child process
    ```

```
    default:
      int status;
      if (waitpid(p, &status, 0) < 0) return error(...);
      if (!WIFEXITED(status) || WEXITSTATUS(status) != 0)
      return error(...);
  }
```

  – date.c:

```
int main(int argc, char **argv) {...}
```

– aside on `restrict`

  – caller not allowed to pass pointers pointing to the same place (todo)

```
int posix_spawnvp(
  pid_t *restrict pid,
  char const *restrict file,
  posix_spawn_file_actions_t const *restrict acts,
  posix_spawn_attr_t const *restrict attrp,
  char *const *restrict argv,
  char *const *restrict envp
);
```

– `_Noreturn void _exit(int status);`

  – send message to parent process

```
while (fork()) continue;
```

7.3 todo: ??

– todo: hr 2
– copy entry in process table

  – except `rax` which stores result of system call
  – pid in `rax` of parent
  – `0` in `rax` of child

– `fork`: child = parent *except* for

  – return value of `fork`
  – pid
  – ppid (parent pid)
  – accumulated execution times
  – file descriptors (file descriptions are shared)
  – file locks (child does not have the lock)
  – pending signals

– `exec` is opposite: it destroys / replaces program data (stack, heap), registers, signal handlers reset to default

7.4 processes do need to affect each other

- need controlled isolation
- files: simple, straightforward, slow, space, names, hassle
  - race conditions
    - occurs when behavior depends on timing
    - `(cat a & cat b) >outfile`: nondeterministic
    - `(cat >a & cat >b) <infile`
    - sorting a big file and using a temp file to store intermediate results: `int tmpfd = open("/tmp/sorttmp", O_WRONLY | O_CREAT | O_TRUNC, 0600);`
      - somone else might have it
      - wait for other `sort` instance to finish using it
        - still has space where race condition can happen
        - toctou race

      ```
      acquire_lock("/tmp");
      // begin critical sector
      if (access(...))
        open(...);
      // end critical sector
      release_lock("/tmp");
      ```

      - `fcntl(fd, F_SETLK / F_SETLKW / F_GETLK / F_UNLOC)` system call
        - *voluntary* locks
        - performance problem:
      - `O_EXCL` exclusive flag: fail if file already exists
      - include pid in filename
        - todo: why bad?
        - use random number + `O_EXCL` flag
          - work but somewhat unsatisfactory
      - `O_TMPFILE` flag: create a file somewhere in directory but dont give it a name
        - `int fd = open("/tmp", O_TMPFILE | O_CREAT | ...);`
        - essentially do `open` and `unlink` atomically
- message-passing
- shared memory: 2 processes not isolated completely, fastest, most dangerous - race conditions
- exit status, signals `kill -HUP 1923`
- covert channels: e.g. cpu load

8 10.13 3th

8.1 breaking application down

- find data → $\boxed{\text{read input}}$ → $\boxed{\text{find words}}$ → $\boxed{\text{count words}}$ → output
- whos in charge

- lazy evaluation
  - minimize total amt of work done
  - `count_words` in charge
- eager evaluation
  - `read_input` in charge
- mixed evaluation
  - all 3 run simultaneously
  - allows for more parallelism
- pipes
  - control execution of multiple processes without any of the processes being in charge

8.2 what a pipe looks like inside the kernel

- buffer of certain size, depending on the kernel, say 4 KiB
- w/in the buffer there are values r / w: read / write pointer
- problems with pipes
  - write to a full pipe
    - consider as an error, return `-1`, set `errno = EAGAIN`
    - process blocks (default)
    - process gets `SIGPIPE` signal
      - if no reader
  - read from an empty pipe
    - rare (todo:??)
    - normal
    - if no writers: `read` returns `0 == EOF`
- pipes are nameless
  - unless named pipe

    ```
    $ mkfifo p
    $ ls -l p
    prw-rw-rw- ... p
    ```

    ```
    int pipe(int ds[2]);
    ```

  - to use the pipe, call `fork`
  - `$ A | B` want another shell prompt after this
  - need 3 processes
  - sh → `A`, `B`
    - `cat bigfile | less` → q
    - `less` is gone
    - shell only cares about `B` fishing

- sh ⟶ `A`, `B`
- sh ⟶ `B`, `A`
  - conventionally, this process tree is used
  - performance advantage: more parallelism, child process sets up pipeline
- shell.c

```
pid_t b = fork();
if (b < 0) error();
if (b == 0)
  setup_a_b(...);
else if (waitpid(b, &status, 0) < 0)
  error();
return status;

...

void setup_a_b(void) {
  int fd[2];
  if (pipe(fd) != 0) error();
  pid_t a = fork();
  if (a < 0) error();
  if (a == 0) {
    dup2(fd[1], STDOUT_FILENO);
    close(fd[0]);
    close(fd[1]);
    execlp("A", "A", NULL);
    error();
  }
  dup2(fd[0], STDIN_FILENO);
  close(fd[0]);
  close(fd[1]);
  execlp("B", "B", NULL);
  error();
}
```

  - no exit status for `A`

8.3 losing power

- small backup power (battery)
- copy cache / ram into secondary storage fast
- how to support this
  - simplify applications by saying kernel will handle everything
    - snapshots each process
    - as if nothing happened in the processes' pov

- relatively expensive (efficiency problem)
- assumes you can snapshot everything
  - often not the case when process is talking to the outside world (e.g. cloud)
  - cannot snapshot everything
  - clock will change (time dependent processes)

- `/dev/power`
  - tells you the power status
  - make processes inspect `/dev/power` and do appropriate things
  - each program must be modified to read `/dev/power` (polling)
- `blocking read c` `char c; fd = open("/dev/power", ...); read(fd, &c, 1);`
  - cant do anything while blocked
  - unless assuming multithreading, but need to notify other threads
- signals
  - rare events
  - localize handling
    - break out of an expensive / complicated loop
  - kill a process
  - timeouts

8.4 signals on linux

- `kill -HUP 19362`
  - send hangup signal to process 19362
  - `if (kill(SIGHUP, 19362) < 0)`
- ordinarily user code can only send signals to processes with the same user id
- user
  - `C-c`: send `SIGINT` to every process in the foreground
  - `SIGKILL`: i want you to die and i dont want you to be able to do anything abt it, cannot be ignored
    - `kill -KILL $$`
  - `SIGSTOP`
  - `SIGUSR1`...
  - `SIGHUP`
- internal errors
  - invalid instruction: `SIGILL`, illegal instruction signal
  - floating point exception: `SIGFPE`, not sent anymore, only sent when `INT_MIN / -1` or `0 / 0`
  - invalid address (bad ptr etc): `SIGSEGV`, `SIGBUS`
- io errors
  - `SIGIO`
  - `SIGPIPE`: writing to pipe w no readers
- death of a child process

  - `SIGCHLD`: rare, one of subsidiaries has died, prob want to clal `waitpid`
- alarm clock
  - `SIGALRM`
- `SIGXCPU`: sent to process when (just abt to) out of cpu time
- `SIGXFSZ`: file size
- `#include <signal.h>`
  - `typedef void (*sig_handler_t)(int);` function type, takes `int` and returns `void`
  - `sighandler_t signal(int signo, sighandler_t handler)` set signal handler and return previous

```
void handle_control_C(int sig) {
  write(1, "?", 1);
}

int main(void) {
  signal(SIGINT, handle_control_C);
  ...
}
```

  - signal handling dispatch table in process table
- signal handlers can be called (by default) between any pair of machine instructions
  - instructions are atomic
  - syscalls are atomic
  - others are not: library calls
    - have to be prepared for the handler function to be called anywhere within a library call
    - e.g. `malloc(1000)`
      - `malloc` in handler = disaster
      - posix rule: dont do that, never call `malloc` from signal handler
    - e.g. `printf("?")`
      - may mess up internal data structure
  - reentrant functions: functions that you can call in the middle of their call
    - e.g. `sqrt`
    - !e.g. `malloc`, `printf`