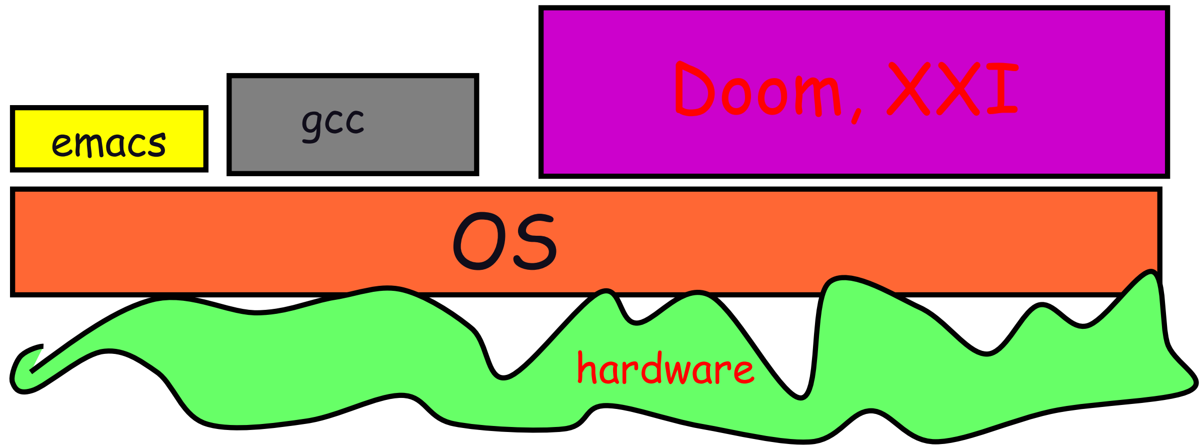
What is an operating system?

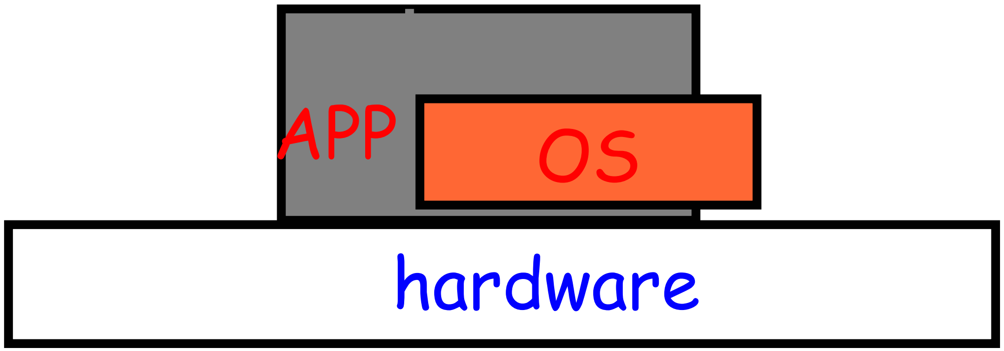
- **Layer between applications and hardware**



- **Makes hardware useful to the programmer**
- **[Usually] Provides abstractions for applications**
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
- **[Often] Provides protection**
  - Prevents one process/user from clobbering another

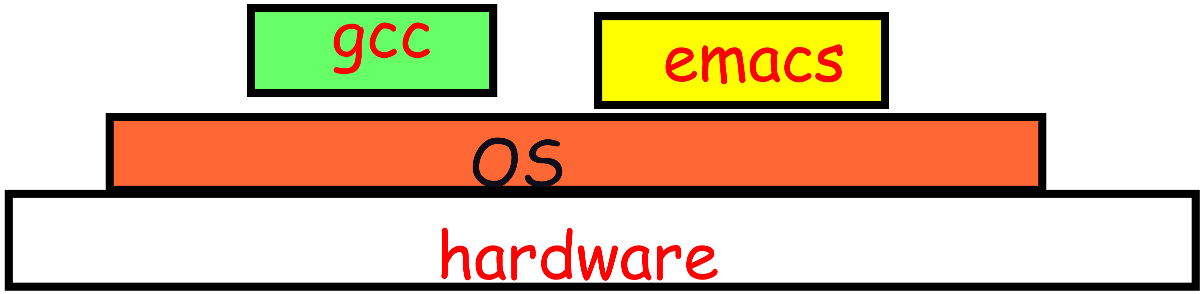
Primitive Operating Systems

- **Just a library of standard services [no protection]**



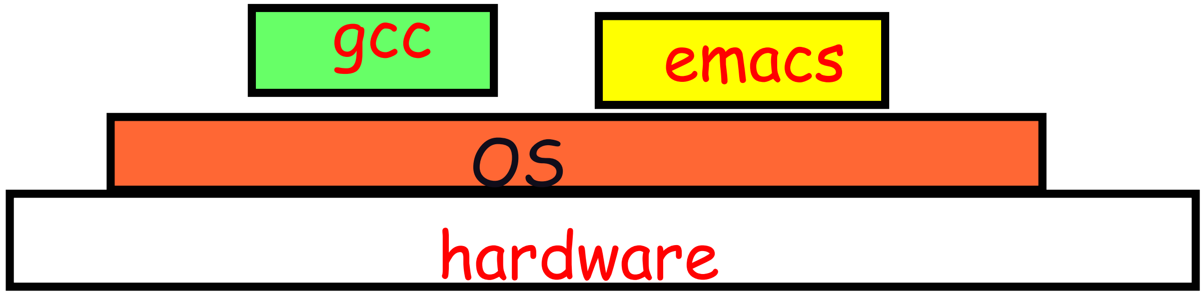
  - Standard interface above hardware-specific drivers, etc.
- **Simplifying assumptions**
  - System runs one program at a time
  - No bad users or programs (often bad assumption)
- **Problem: Poor utilization**
  - . . . of hardware (e.g., CPU idle while waiting for disk)
  - . . . of human user (must wait for each program to finish)

Multitasking



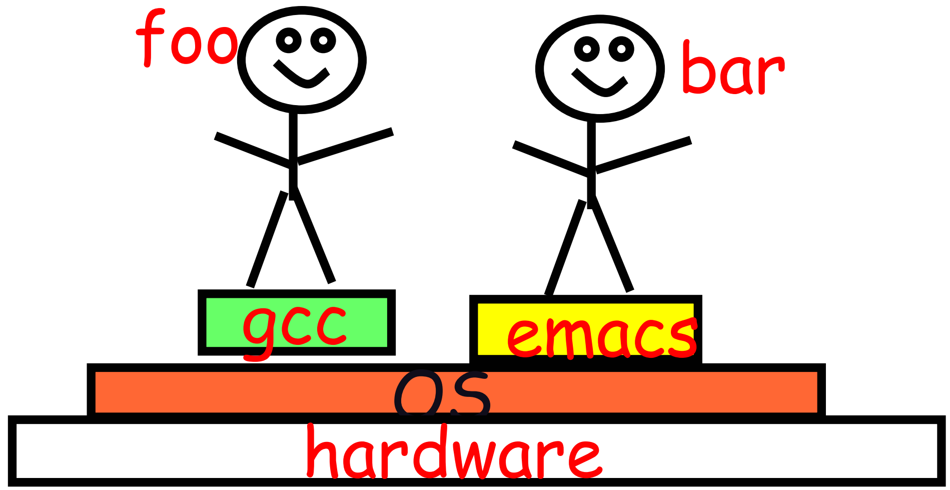
- **Idea: Run more than one process at once**
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**

Multitasking



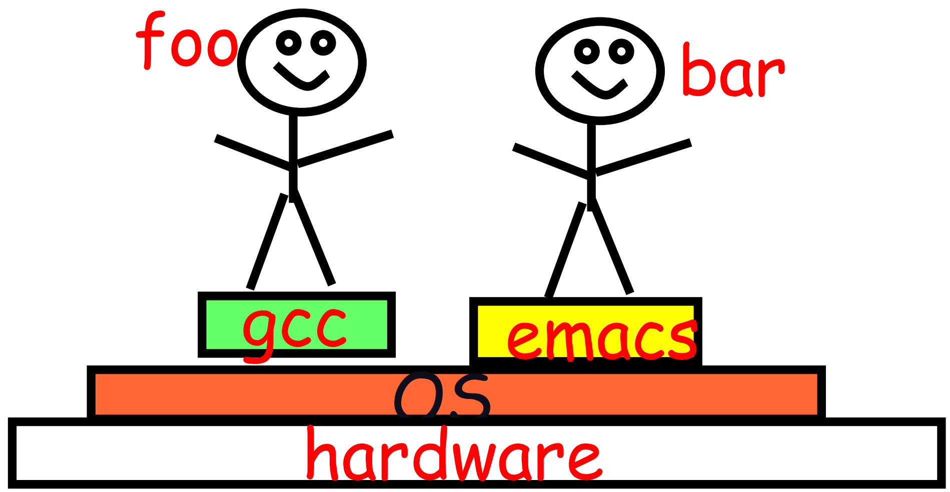
- **Idea: Run more than one process at once**
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**
  - Go into infinite loop and never relinquish CPU
  - Scribble over other processes' memory to make them fail
- **OS provides mechanisms to address these problems**
  - *Preemption* – take CPU away from looping process
  - *Memory protection* – protect process's memory from one another

Multi-user OSes



- **Many OSes use *protection* to serve distrustful users**
- **Idea: With N users, system not N times slower**
  - Users' demands for CPU, memory, etc. are bursty
  - Win by giving resources to users who actually need them
- **What can go wrong?**

Multi-user OSes

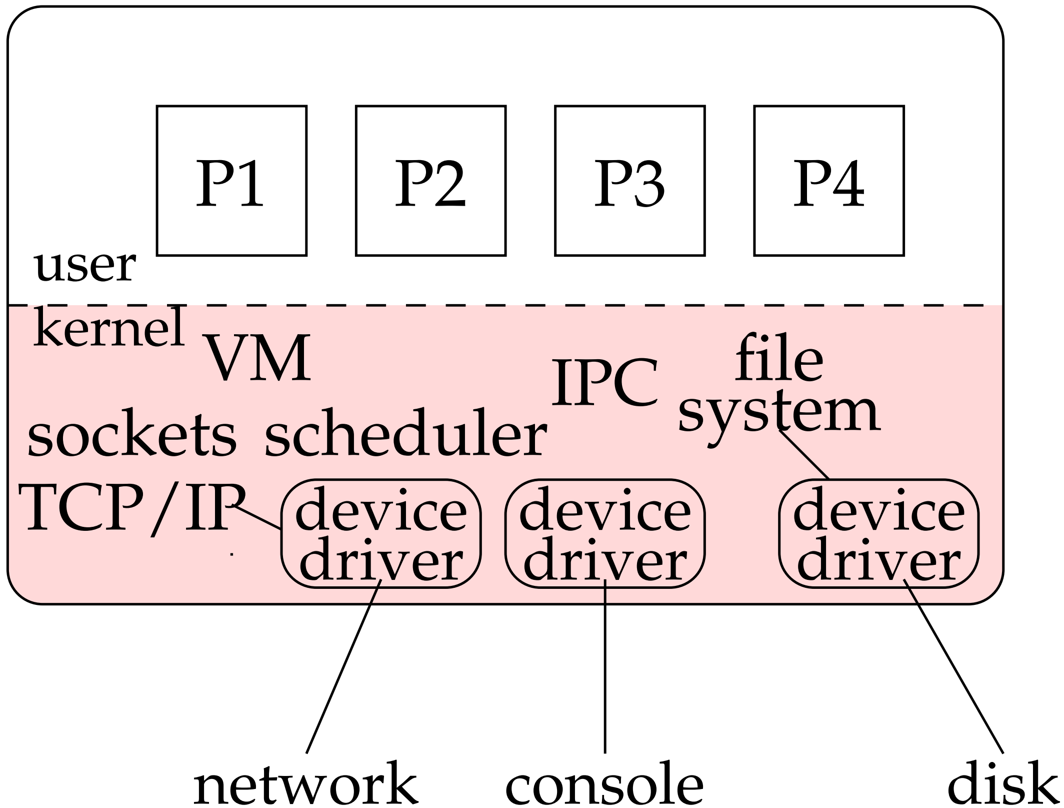


- **Many OSes use *protection* to serve distrustful users**
- **Idea: With N users, system not N times slower**
    - Users' demands for CPU, memory, etc. are bursty
    - Win by giving resources to users who actually need them
- **What can go wrong?**
    - Users are gluttons, use too much CPU, etc. (need policies)
    - Total memory usage greater than in machine (must virtualize)
    - Super-linear slowdown with increasing demand (thrashing)

Protection

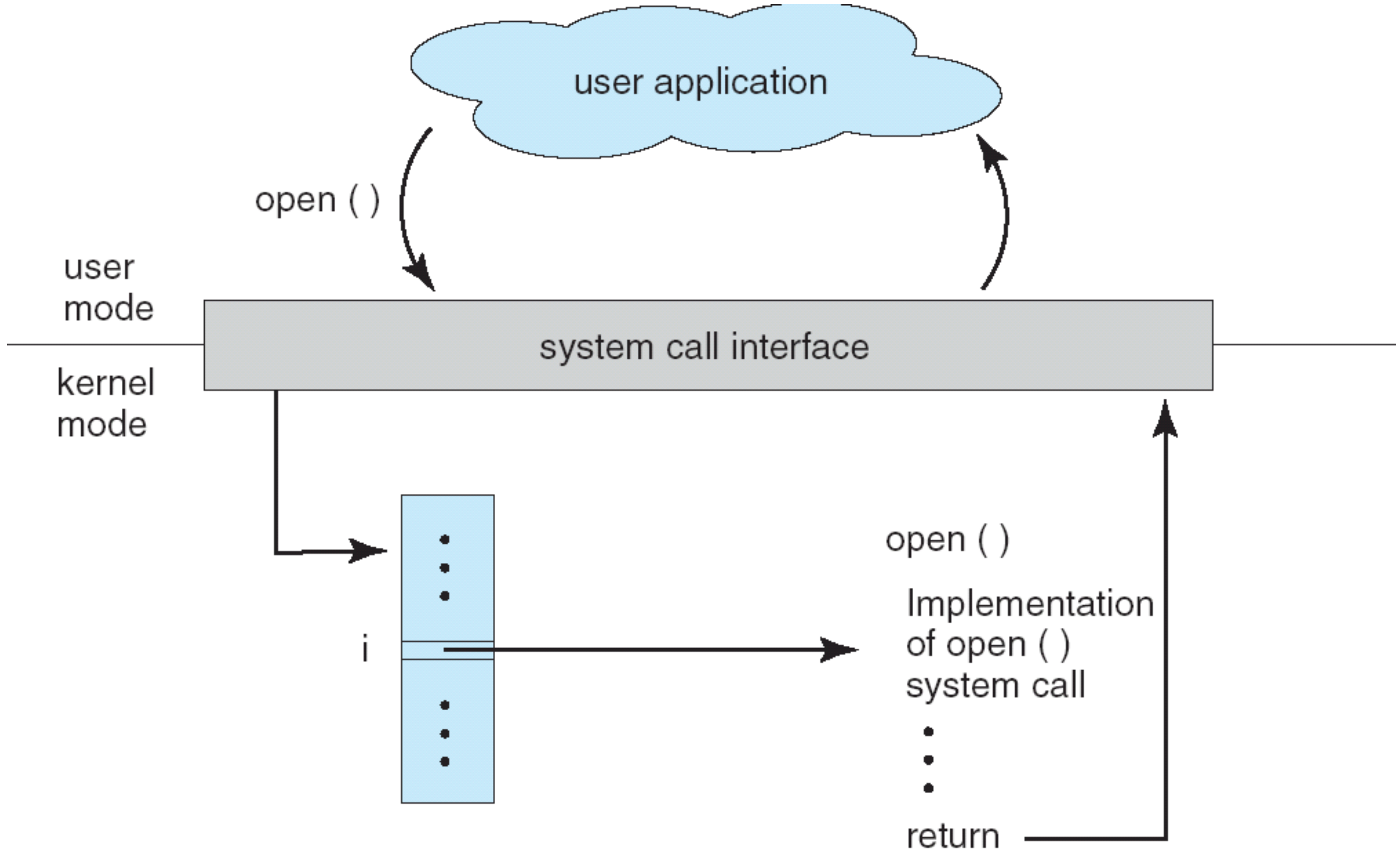
- **Mechanisms that isolate bad programs and people**
- **Pre-emption:**
  - Give application a resource, take it away if needed elsewhere
- **Interposition/mediation:**
  - Place OS between application and "stuff"
  - Track all pieces that application allowed to use (e.g., in table)
  - On every access, look in table to check that access legal
- **Privileged & unprivileged modes in CPUs :**
  - Applications unprivileged (user/unprivileged mode)
  - OS privileged (privileged/supervisor mode)
  - Protection operations can only be done in privileged mode

Typical OS structure



- **Most software runs as user-level processes (P[1-4])**
- **OS *kernel* runs in *privileged* mode [shaded]**
  - Creates/deletes processes
  - Provides access to hardware

System calls

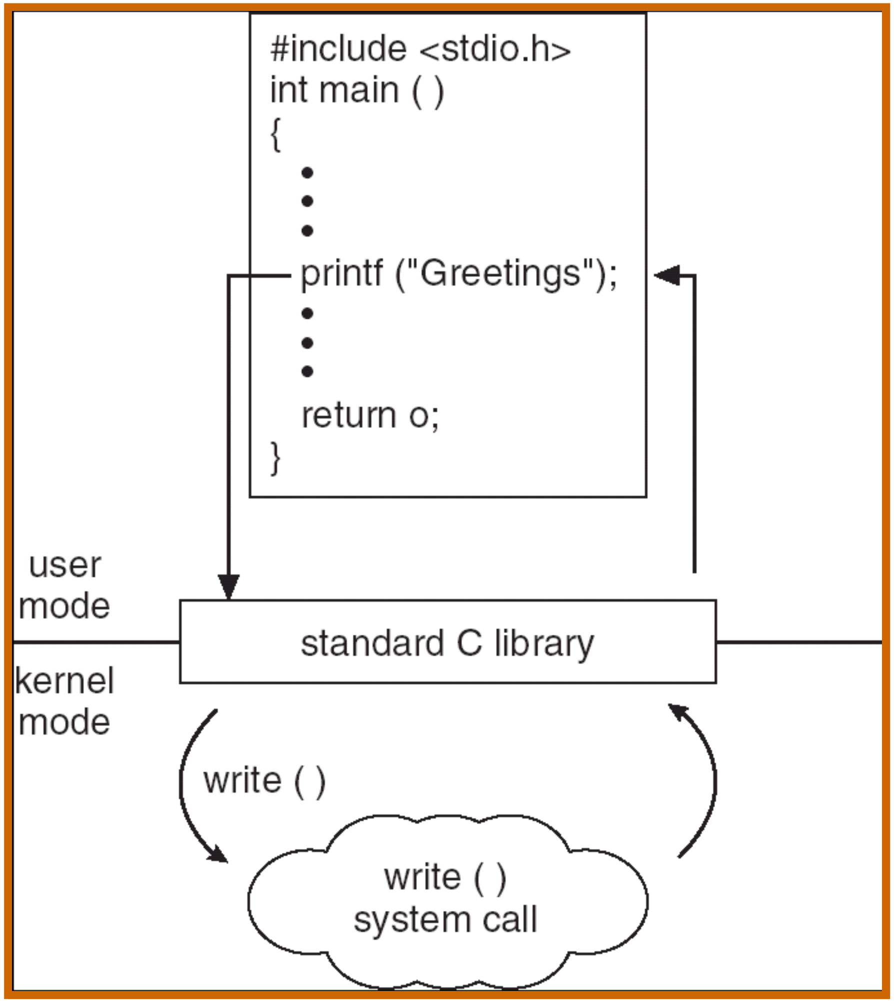


- **Applications can invoke kernel through *system calls***
  - Special instruction transfers control to kernel
  - …which dispatches to one of few hundred syscall handlers

System calls (continued)

- **Goal: Do things app. can't do in unprivileged mode**
  - Like a library call, but into more privileged kernel code
- **Kernel supplies well-defined *system call* interface**
  - Applications set up syscall arguments and *trap* to kernel
  - Kernel performs operation and returns result
- **Higher-level functions built on syscall interface**
  - `printf`, `scanf`, `gets`, etc. all user-level code
- **Example: POSIX/UNIX interface**
  - `open, close, read, write, ...`

System call example



- **Standard library implemented in terms of syscalls**
  - *printf* – in libc, has same privileges as application
  - calls *write* – in kernel, which can send bits out serial port

UNIX file system calls

- **Applications “open” files (or devices) by name**
  - I/O happens through open files
- `int open(char *path, int flags, /*mode*/...);`
  - `flags`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - `O_CREAT`: create the file if non-existent
  - `O_EXCL`: (w. `O_CREAT`) create if file exists already
  - `O_TRUNC`: Truncate the file
  - `O_APPEND`: Start writing from end of file
  - `mode`: final argument with `O_CREAT`
- **Returns file descriptor—used for all I/O to file**

Error returns

- **What if** `open` **fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error in global int `errno`
- `#include <sys/errno.h>` **for possible values**
  - 2 = `ENOENT` "No such file or directory"
  - 13 = `EACCES` "Permission Denied"
- `perror` **function prints human-readable message**
  - `perror ("initfile");`

    → "`initfile: No such file or directory`"

Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- `int write (int fd, void *buf, int nbytes);`
  - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
  - `whence`: 0 – start, 1 – current, 2 – end
    - ▷ Returns previous file offset, or -1 on error
- `int close (int fd);`

File descriptor numbers

- **File descriptors are inherited by processes**
  - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
  - 0 – "standard input" (`stdin` in ANSI C)
  - 1 – "standard output" (`stdout`, `printf` in ANSI C)
  - 2 – "standard error" (`stderr`, `perror` in ANSI C)
  - Normally all three attached to terminal
- **Example:** `type.c`
  - Prints the contents of a file to `stdout`

type.c

```
void
typefile (char *filename)
{
  int fd, nread;
  char buf[1024];

  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    perror (filename);
    return;
  }

  while ((nread = read (fd, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  close (fd);
}
```

Different system contexts

- **A system can typically be in one of several contexts**
- ***User-level* – running an application**
- **Kernel process context**
  - Running kernel code on behalf of a particular process
  - E.g., performing system call
  - Also exception (mem. fault, numeric exception, etc.)
  - Or executing a kernel-only process (e.g., network file server)
- **Kernel code not associated w. a process**
  - Timer interrupt (hardclock)
  - Device interrupt
  - "Softirqs", "Tasklets" (Linux-specific terms)
- **Context switch code – changing address spaces**

CPU preemption

- **Protection mechanism to prevent monopolizing CPU**
- **E.g., kernel programs timer to interrupt every 10 ms**
  - Must be in supervisor mode to write appropriate I/O registers
  - User code cannot re-program interval timer
- **Kernel sets interrupt to vector back to kernel**
  - Regains control whenever interval timer fires
  - Gives CPU to another process if someone else needs it
  - Note: must be in supervisor mode to set interrupt entry points
  - No way for user code to hijack interrupt handler
- **Result: Cannot monopolize CPU with infinite loop**
  - At worst get $1/N$ of CPU with N CPU-hungry processes

Protection is not security

- **How *can* you monopolize CPU?**

Protection is not security

- **How *can* you monopolize CPU?**
- **Use multiple processes**
- **For many years, could wedge most OSes with**

```
int main() { while(1) fork(); }
```

  - Keeps creating more processes until system out of proc. slots

- **Other techniques: use all memory (`chill` program)**
- **Typically solved with technical/social combination**
  - Technical solution: Limit processes per user
  - Social: Reboot and yell at annoying users
  - Social: Pass laws (often debatable whether a good idea)

Address translation

- **Protect mem. of one program from actions of another**
- **Definitions**
  - *Address space*: all memory locations a program can name
  - *Virtual address*: addresses in process' address space
  - *Physical address*: address of real memory
  - *Translation*: map virtual to physical addresses
- **Translation done on every load and store**
  - Modern CPUs do this in hardware for speed
- **Idea: If you can't name it, you can't touch it**
  - Ensure one process's translations don't include any other process's memory

Resource allocation & performance

- **Multitasking permits higher resource utilization**
- **Simple example:**
  - Process downloading large file mostly waits for network
  - You play a game while downloading the file
  - Higher CPU utilization than if just downloading
- **Complexity arises with cost of switching**
- **Example: Say disk 1,000 times slower than memory**
  - 1 GB memory in machine
  - 2 Processes want to run, each use 1 GB
  - Can switch processes by swapping them out to disk
  - Faster to run one at a time than keep context switching

Useful properties to exploit

- **Skew**
  - 80% of time taken by 20% of code
  - 10% of memory absorbs 90% of references
  - Basis behind cache: place 10% in fast memory, 90% in slow, usually looks like one big fast memory
- **Past predicts future (a.k.a. temporal locality)**
  - What's the best cache entry to replace?
  - If past = future, then least-recently-used entry
- **Note conflict between fairness & throughput**
  - Higher throughput (fewer cache misses, etc.) to keep running same process
  - But fairness says should periodically preempt CPU and give it to next process