CS3510
Operating Systems

Processes

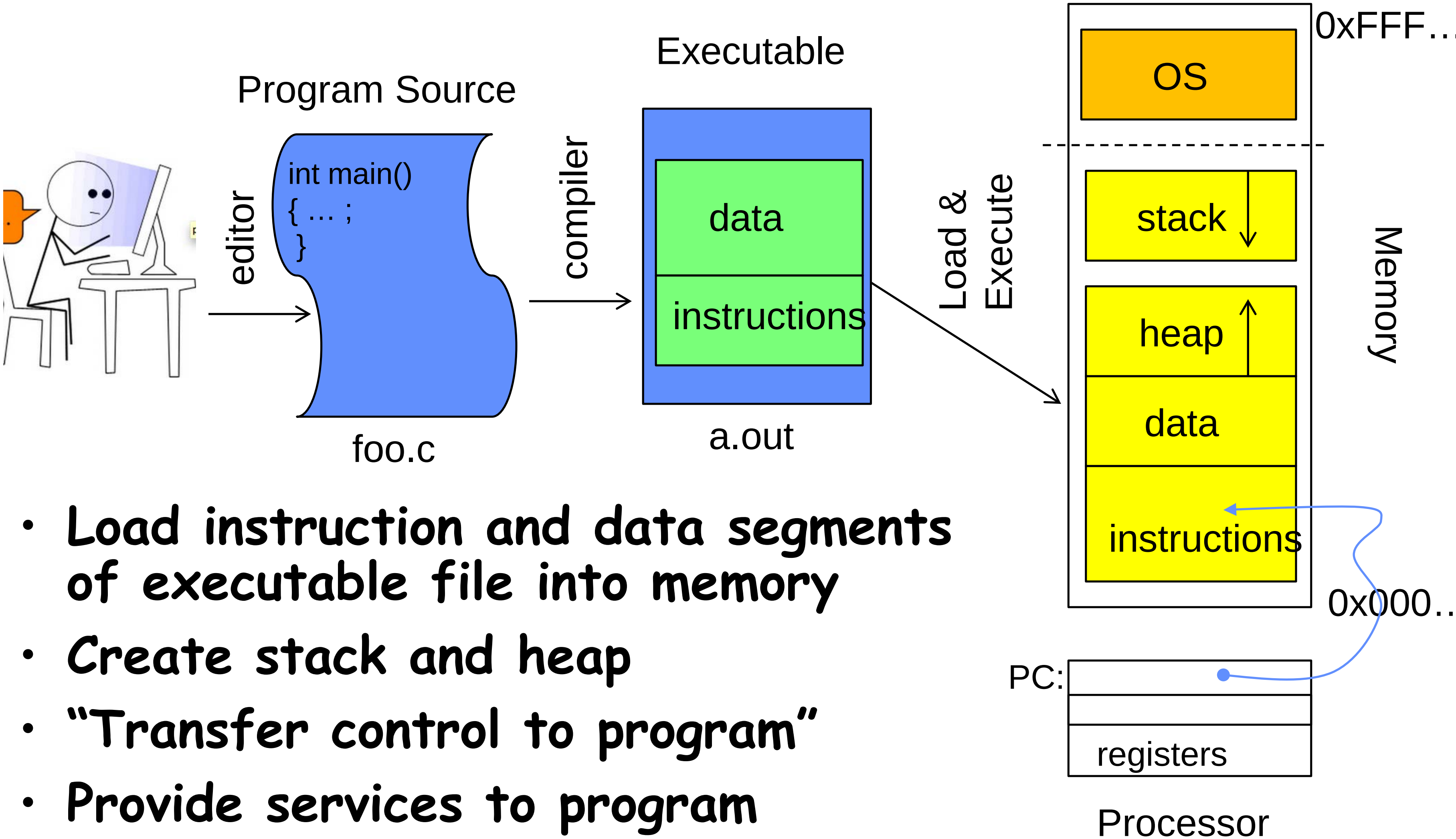
Bheemarjuna Reddy Tamma
IIT HYD

Outline

- Uni vs multiprogramming?
- What are Processes?
- How are they related to Programs and Address Spaces?

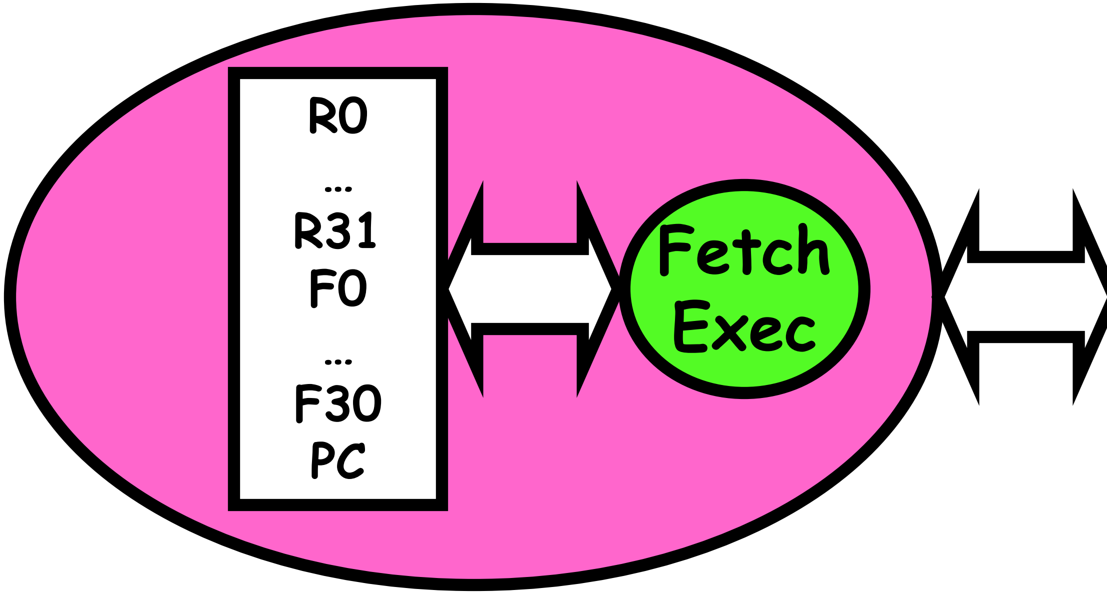
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides are from Prof John Kubi, UCB.

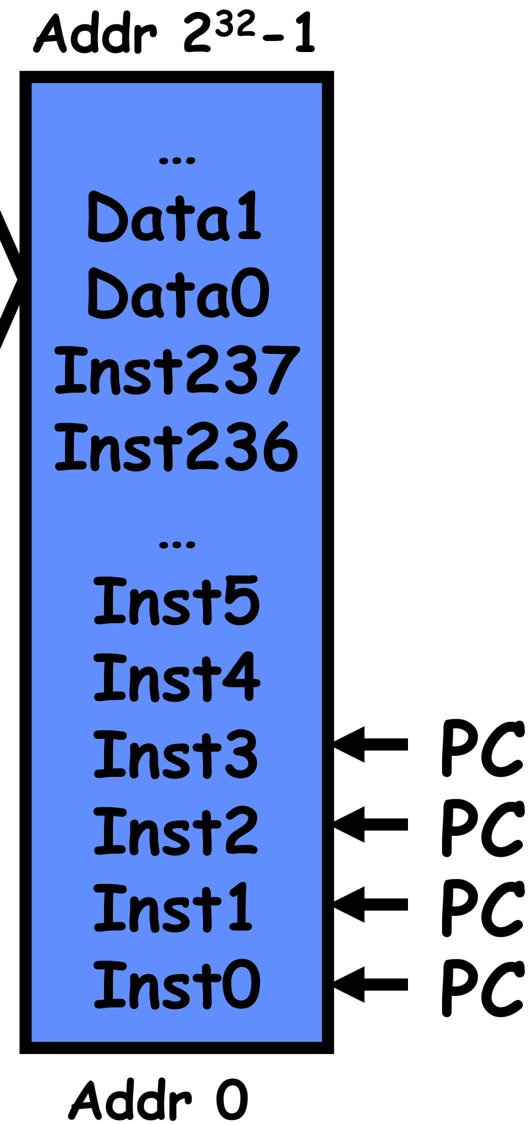
OS Bottom Line: Run Programs



- Load instruction and data segments of executable file into memory
- Create stack and heap
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

What happens during execution?





- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

Thread of Control

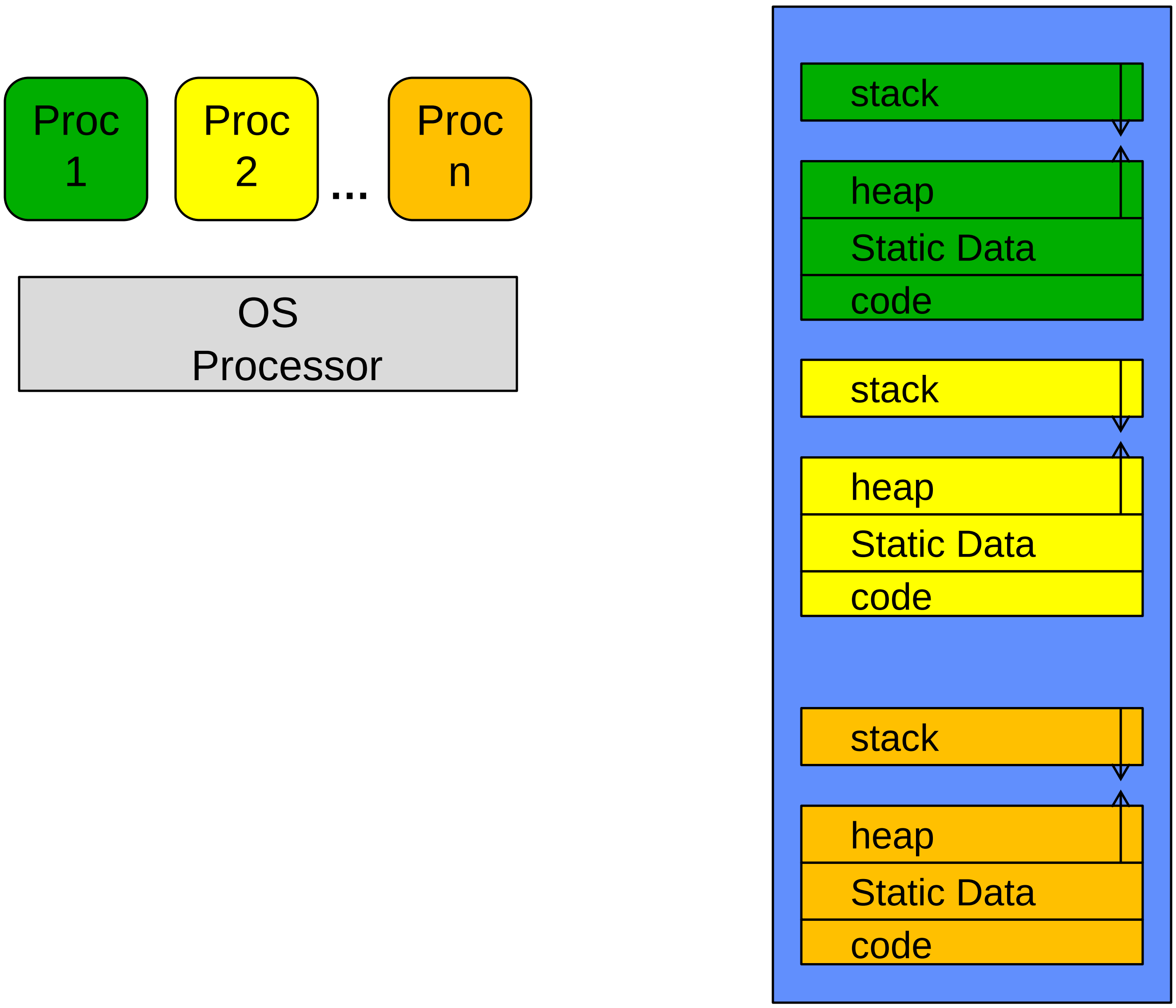
- Thread: Single unique execution context
  - Independent Fetch/Decode/Execute loop
  - Operating in some Address space
  - Program Counter, Registers, Execution Flags, Stack
- Certain registers hold the *context* of thread
  - PC register holds the address of executing instruction in the thread
  - Stack pointer holds the address of the top of stack
  - May be defined by the instruction set architecture or by compiler conventions
- A thread is executing on a processor when it is resident in the processor registers
  - Registers hold the root state of the thread
    - » The rest is "in memory"

Concurrency vs Parallelism

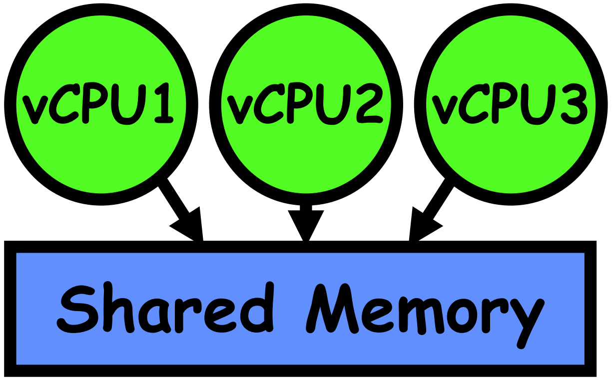
- Uniprogramming: *one thread at a time in the system*
  - MS/DOS, early Macintosh, Batch processing
  - Easier for operating system builder
  - Does this make sense for personal computers?
- Multiprogramming: *more than one thread at a time in the system*
  - Multics, UNIX/Linux, OS/2, Windows 10, Mac OS X
  - Often called "multitasking", but multitasking (aka time sharing and concurrency) is bit different from it
    - » CPU executes multiple jobs alternatively by switching from one to other, but switching is so frequent to provide interactive computing (time slice)
- Multi-processor (or ManyCore) System ⇒ Multiprogramming or Multitasking?
  - No, it's parallel programming!

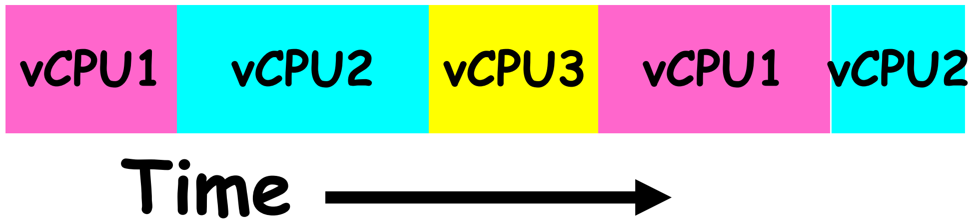
https://techdifferences.com/difference-between-concurrency-and-parallelism.html

Multiprogramming/Concurrency-Multiple Threads of Control



How can we give the illusion of multiple processors?





- Assume a single processor (CPU core). How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

The Basic Problem of Concurrency

- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multitasking API: users think they have exclusive access to shared resources
- OS Has to coordinate all activity
  - Multiple users, I/O interrupts, …
  - How can it keep all these things straight?
- Basic Idea: Use Virtual Machine abstraction
  - Abstract the notion of an executing program
  - Then, worry about multiplexing these abstract machines
- Dijkstra did this for the "THE system"
  - Few thousand lines vs 1 million lines in OS 360 (1K bugs)

Properties of this simple multiprogramming technique

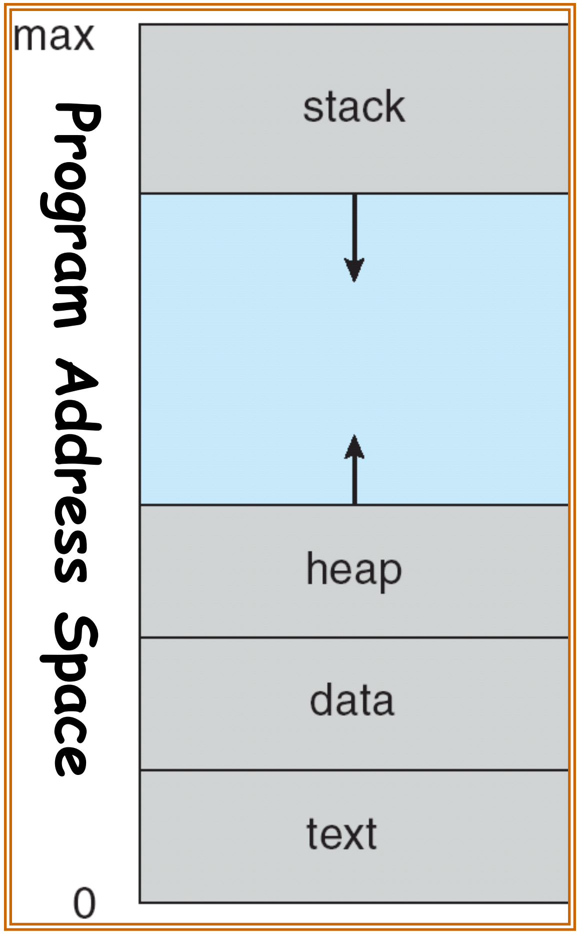
- All virtual CPUs (i.e., threads of control) share same non-CPU resources
  - I/O devices the same
  - Memory the same
- Consequence of sharing:
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
- This (unprotected) model common in:
  - Embedded applications
  - Windows 3.1/Machintosh (switch only with yield)
  - Windows 95—ME (switch with both yield and timer)

How to protect threads from one another?

- Need three important things:
  1. Protection of memory
     » Every thread does not have access to all memory
  2. Protection of I/O devices
     » Every thread does not have access to every device
  3. Protection of Access to Processor:
     Preemptive switching from thread to thread
     » Use of timer
     » Must not be possible to disable timer from user-code

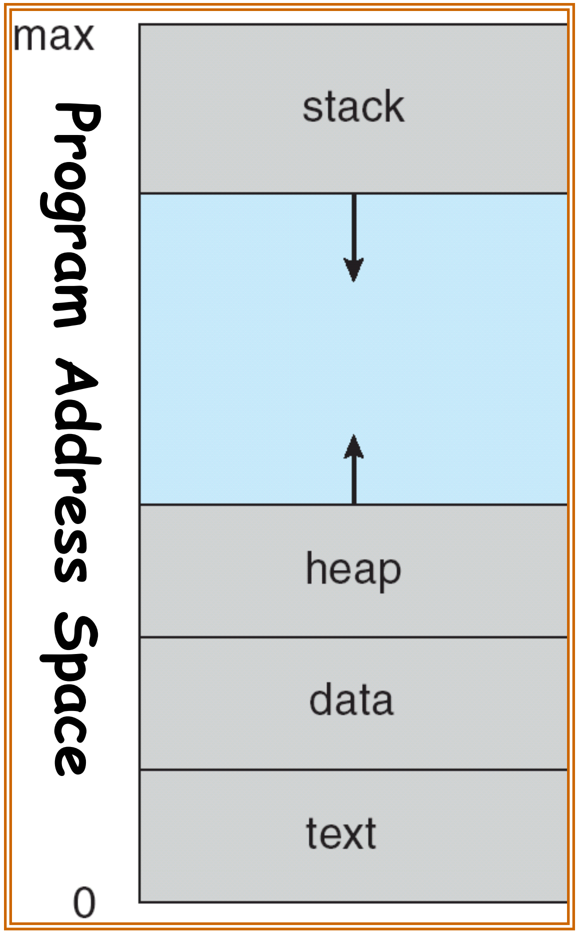
Thread's Address Space

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor, there are 2^{32} = 4 billion addresses
- Virtual address space is divided into segments
- Segment: Contiguous range of virtual address space
- Four segments of a thread:
  - Text/Code: instructions
  - Data: working storage of prog (Global, Static variables)
  - Stack: local data, return values, parameter passing
  - Heap: dynamic memory

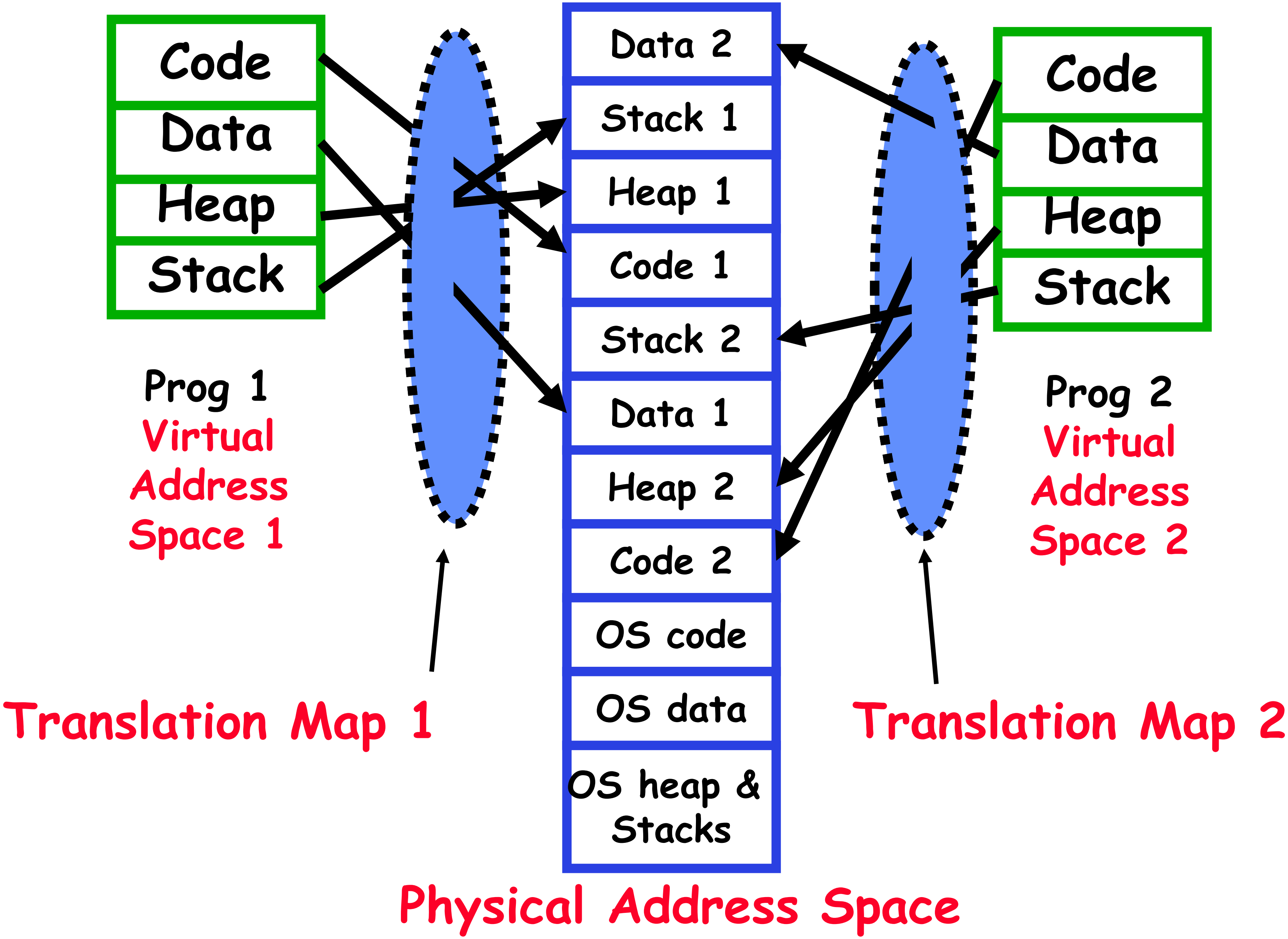


Thread's Address Space

- What happens when you read from or write to an address?
  - Perhaps Nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - » (Memory-mapped I/O)
  - Perhaps causes exception (fault)



Providing Illusion of Separate Address Space: Load new Translation Map on Context Switch



Traditional UNIX Process

- Process: *Operating system abstraction to represent what is needed to run a single program*
  - Process is an instance of a program in execution
  - Often called a "HeavyWeight Process"
  - There is no concurrency in a heavyweight process → a single thread!
  - Collection of data structures that fully describes how far the execution of the program has progressed
  - Formally: a single, sequential stream of execution in its *own* address space
- Process has two parts:
  1. Sequential Program Execution Stream
     » Code executed as a *single, sequential* stream of execution
     » Includes State of CPU registers
  2. Protected Resources:
     » Main Memory State (contents of Address Space)
     » I/O state (i.e. file descriptors)

Process = Program ??

```
main () {
    …;
}
A() {
    …
}
```

Program

```
main ()
{
    …;
}
A() {
    …
}
```

Heap

Stack

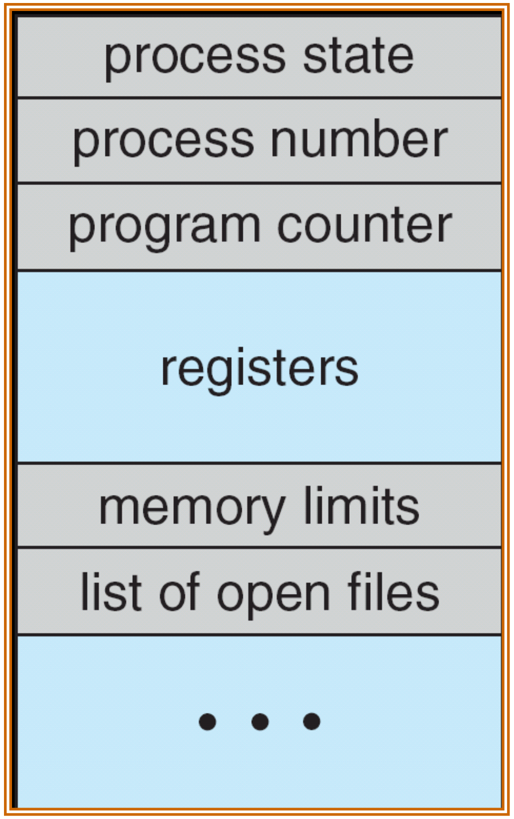
A
main

Process

- Program is passive entity, but process is active entity
  - Program becomes process when its executable file is loaded into memory → click on icon or type it on command line (shell)
- More to a process than just a program:
  - Program is just a part of the process state
  - I run emacs editor on lectures.txt, you run it on homework.c – Same program, but different processes
- Less to a process than a program:
  - A program can invoke more than one process
  - cc starts up cpp, cc1, cc2, as, and ld

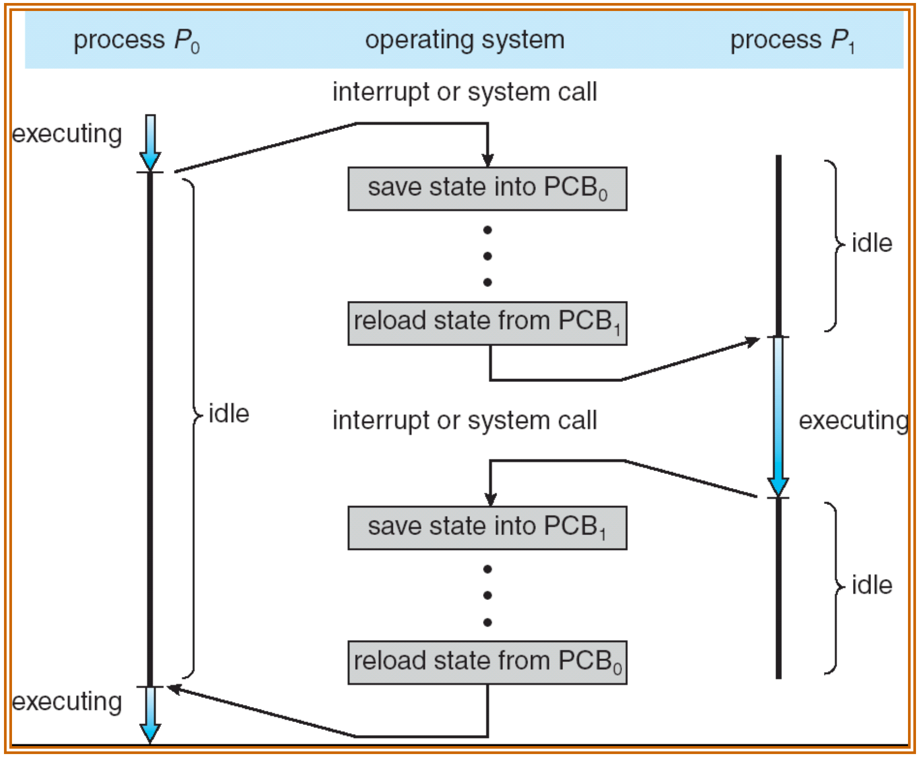
How do we multiplex processes?

- The current state of process held in a DS called process control block (PCB):
  - This is a "snapshot" of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time slice to different processes (Scheduling):
  - Only one process "running" at a time
  - Give more time to important processes e.g, I/O, foreground jobs
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    - » Memory Mapping: Give each process its own (virtual) address space
    - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls



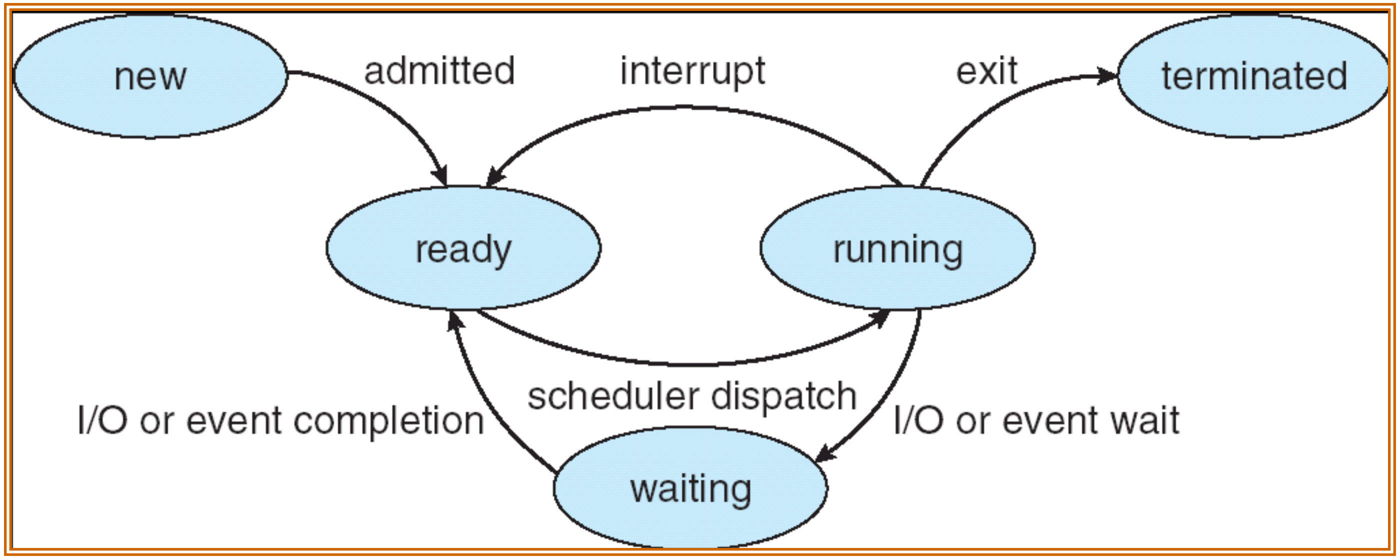
Process Control Block (PCB)

CPU Switch From Process to Process



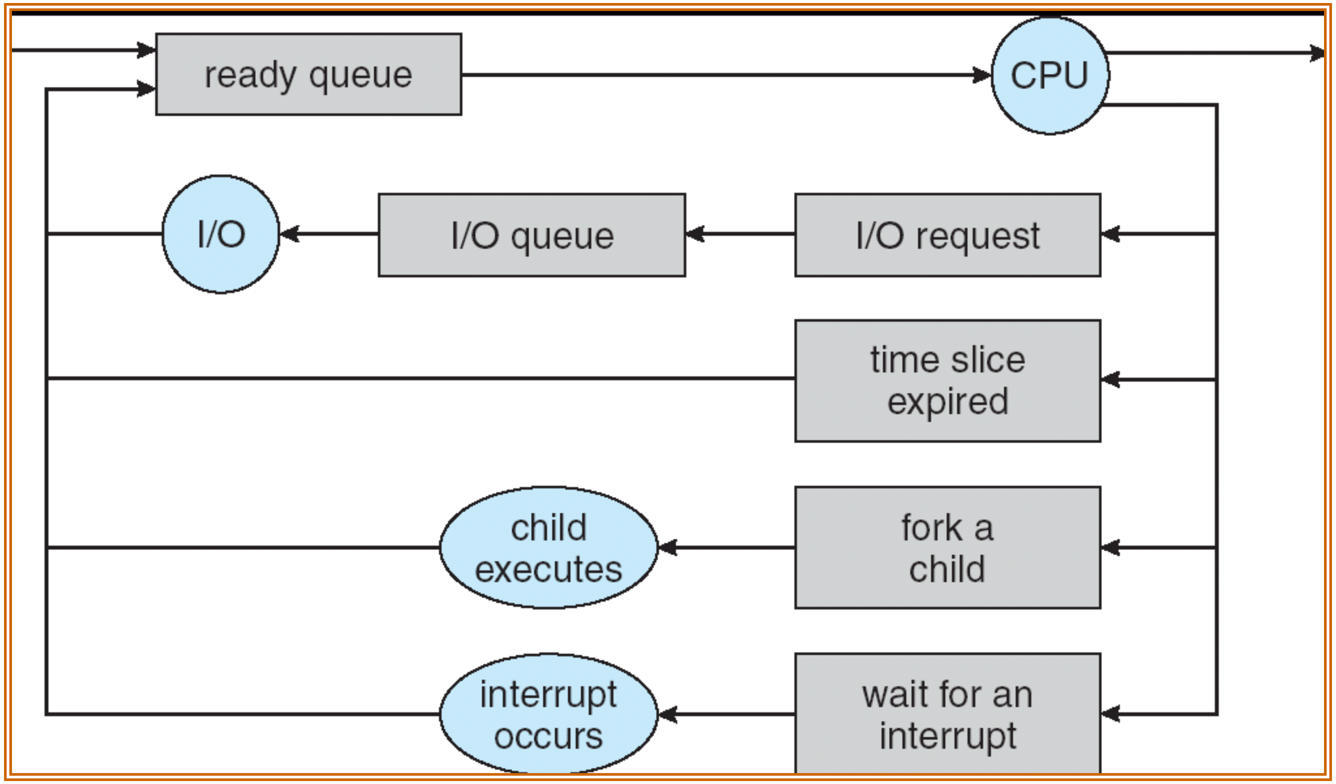
- This is also called a "context switch"
- Code executed in kernel above is overhead as the system does no useful work while switching
  - Overhead (~ few millisecs) sets minimum practical switching time
  - The more complex OS and PCB ➔ the longer context switch time
  - Some hardware provides multiple sets of registers ➔ multiple contexts loaded at once → no saving/reloading overhead

Diagram of Process State



- As a process executes, it changes *state*
  - new: The process is being created
  - ready: The process is waiting to run by CPU
  - running: Instructions are being executed by CPU
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

Process Scheduling (Queue representation)

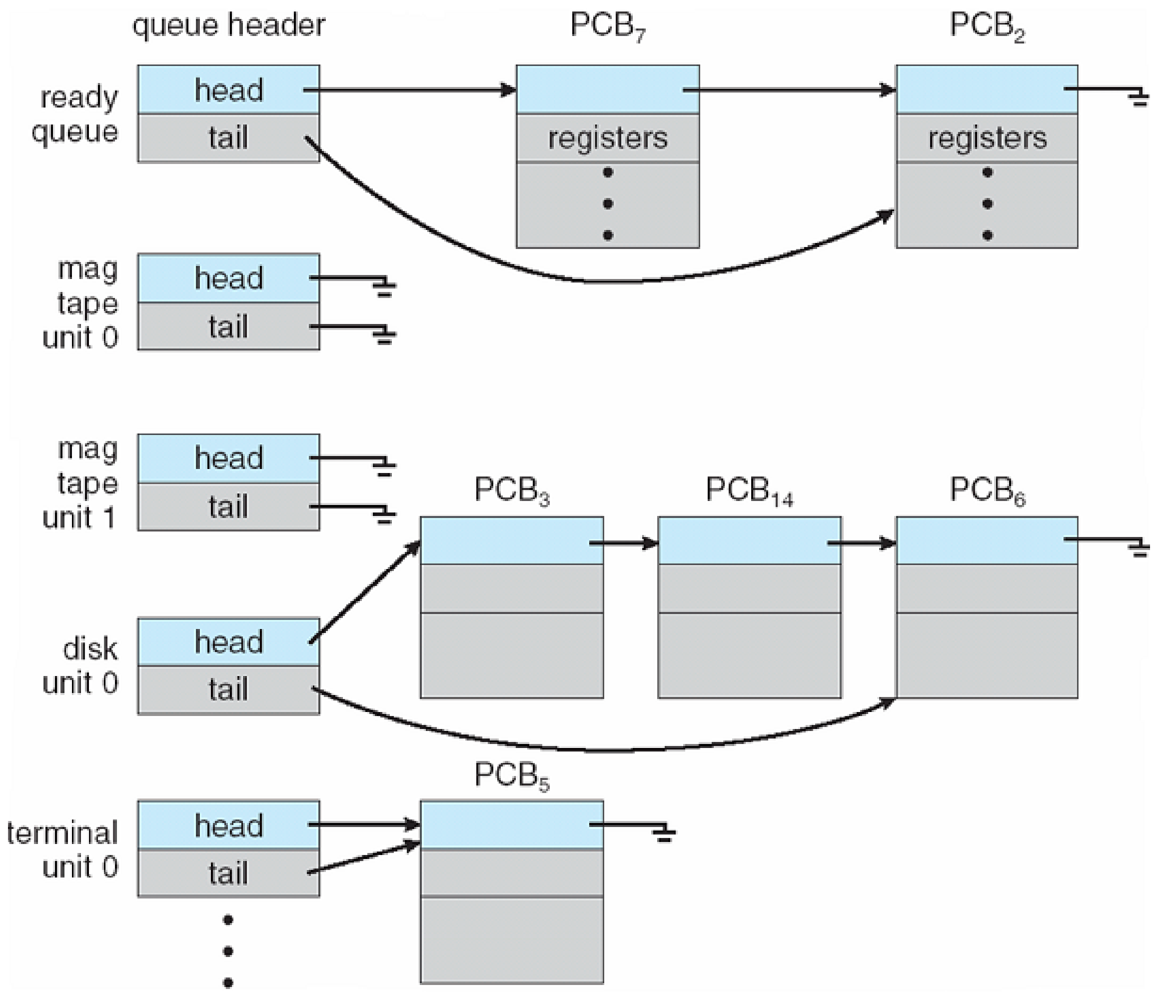


- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible for CPU ands other queues (discussed later)

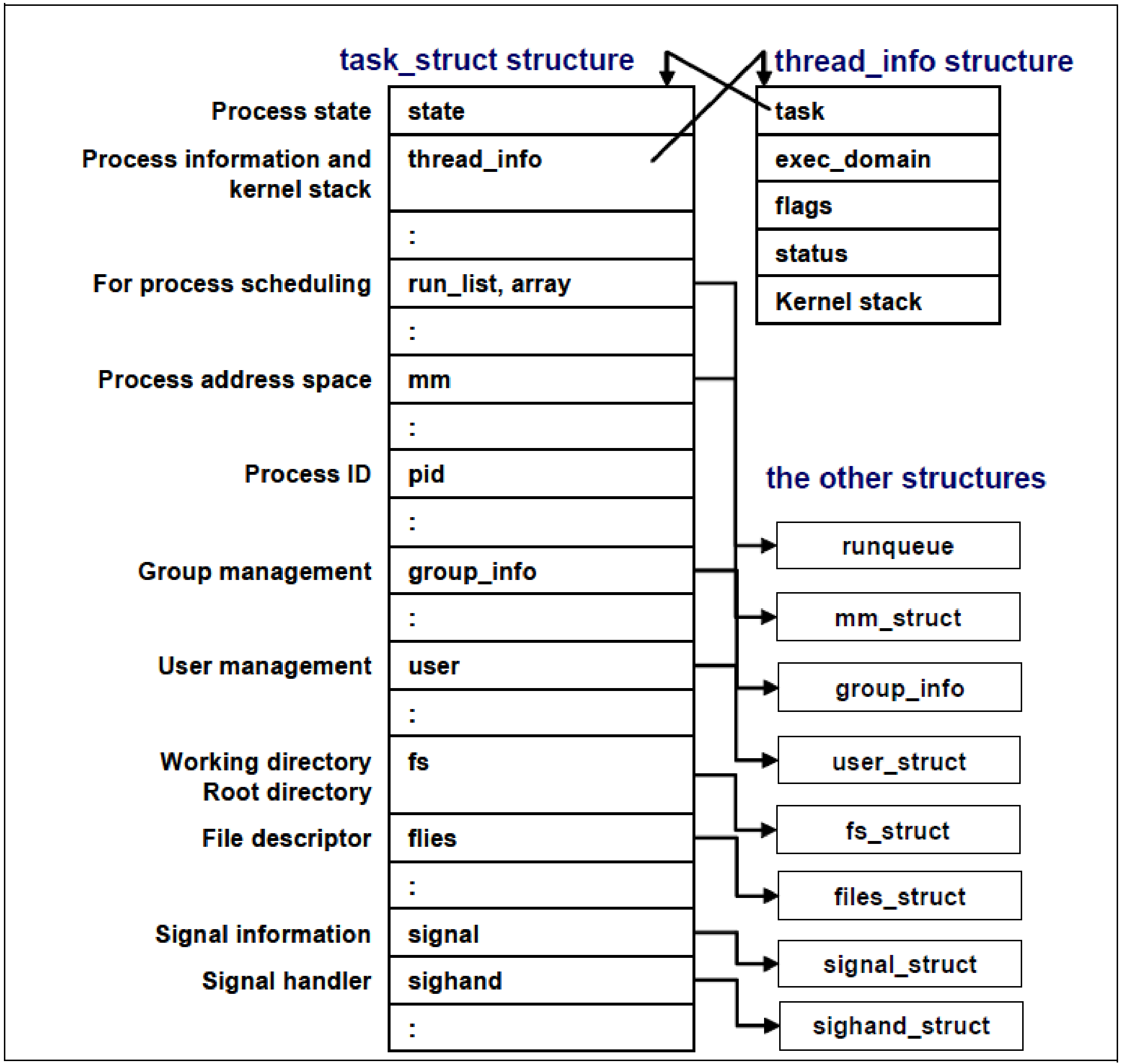
CPU Scheduler

```
if ( readyProcesses(PCBs) ) {
        nextPCB = selectProcess(PCBs);
        run( nextPCB );
} else {
        run_idle_process();
}
```

Ready Queue and I/O Device Queues



Linux Task Struct



Linux Task Struct (also called Process Descriptor):
https://github.com/torvalds/linux/blob/master/include/linux/sched.h
https://github.com/hungys/tech-note/blob/master/linux_kernel/process.md

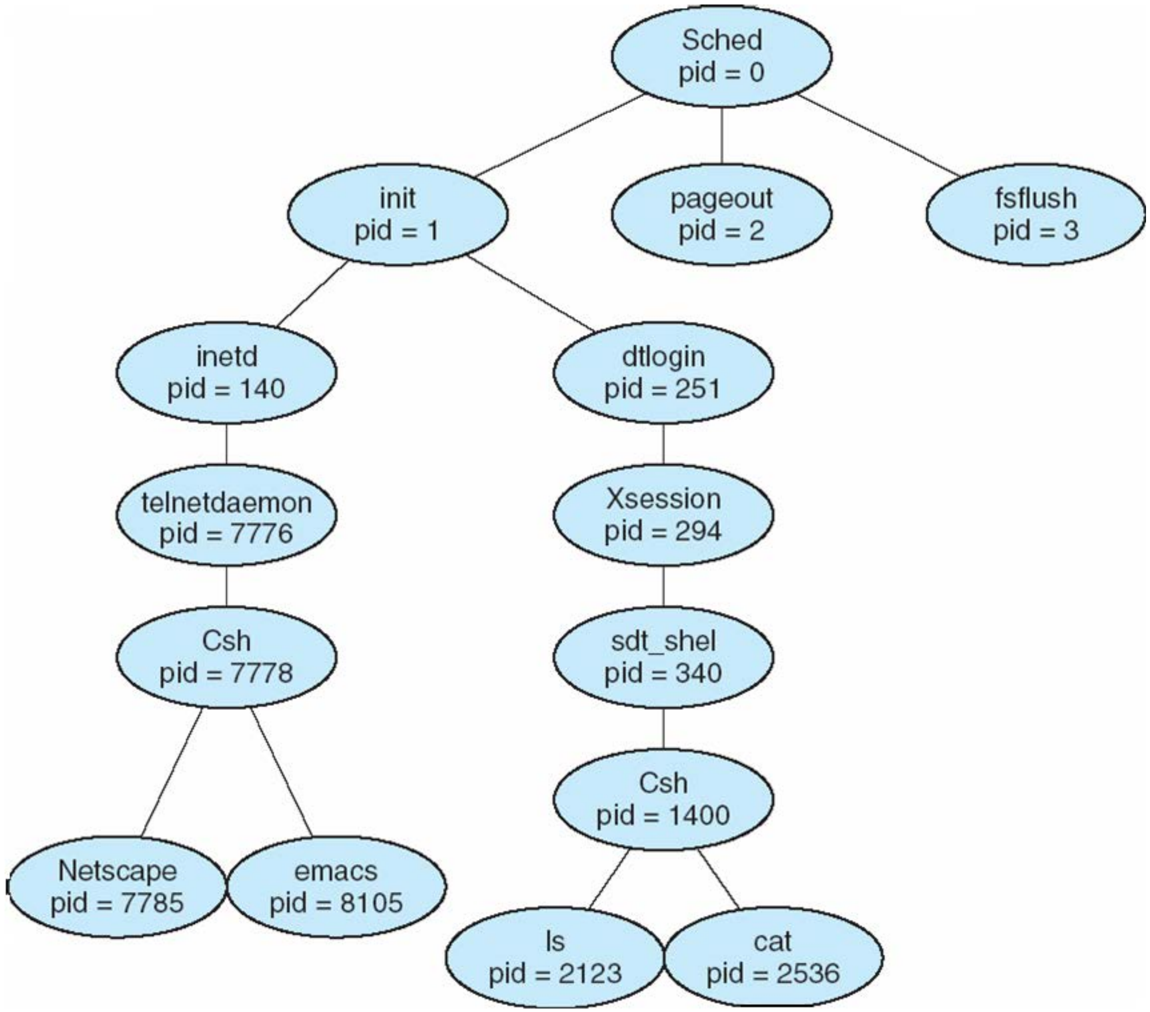
What does it take to create a process?

- Must construct new PCB
  - Inexpensive
- Must set up new page table for address space
  - More expensive (discussed later)
- Copy data from parent process? (Unix `fork()`)
  - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
  - Originally *very* expensive
  - Much less expensive with "copy-on-write"
- Copy I/O state (file handles, etc)
  - Medium expense

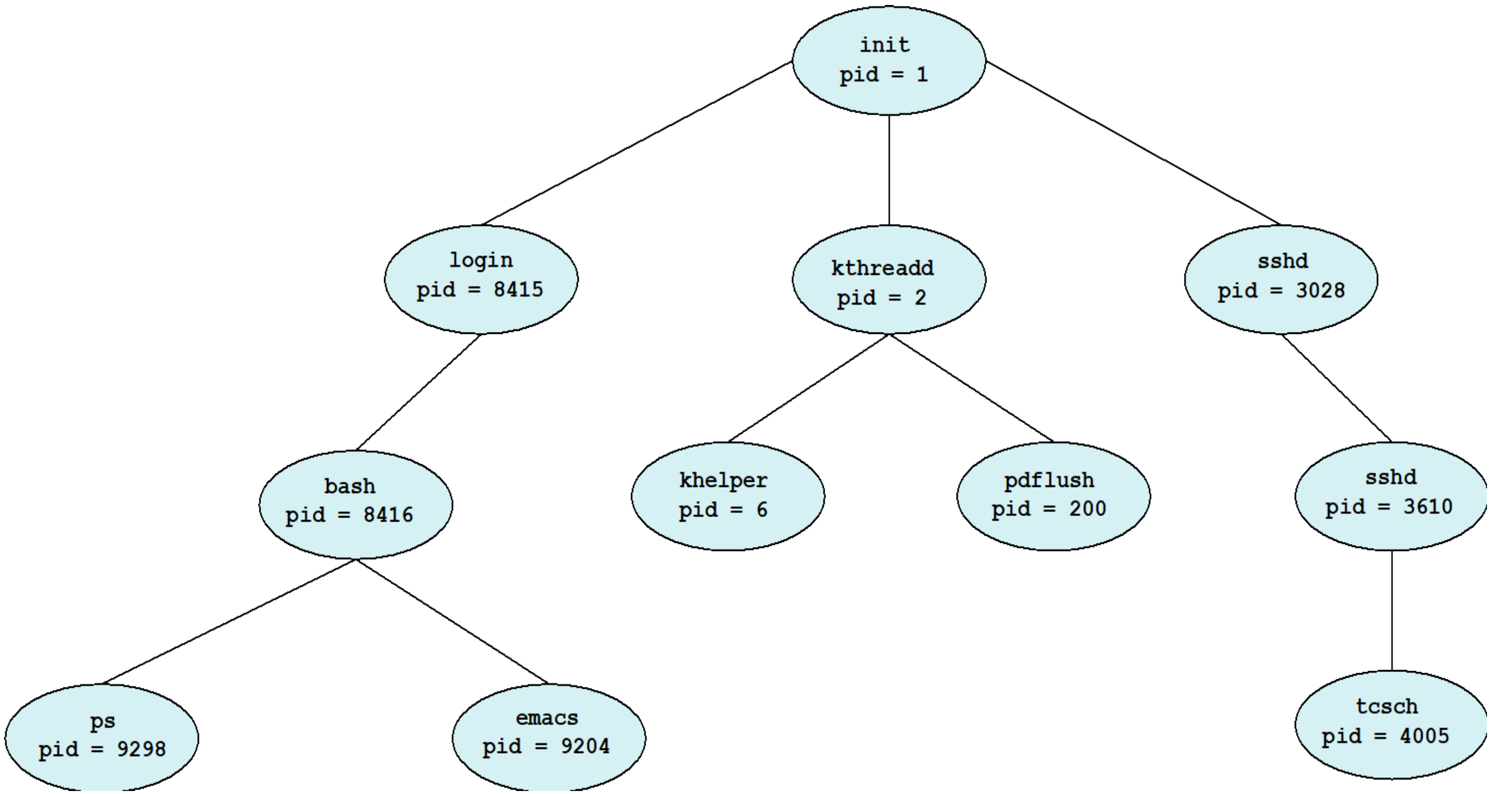
How to create a process?

- Double click on an icon or type program name on command-line (shell)
- After boot, OS starts the first process
  - E.g. sched for Solaris, ntoskrnel.exe for Windows
- The first process creates other processes:
  - the creator is called the parent process
  - the created is called the child process
  - parent/child relationships is expressed by a process tree
- Eg, in UNIX the second process is called *init*
  - it creates all the gettys (login processes) and daemons
  - it should never die
  - it controls the system configuration (#processes, priorities…)
- Explorer.exe in Windows for graphical interface

An Example Process Tree in Solaris



An Example Process Tree in Linux



An Example Process Tree in Solaris

http://man7.org/linux/man-pages/man1/pstree.1.html

Processes Under UNIX

- Fork() system call is only way to create a new process
- int fork() does many things at once:
  - creates a new address space (for the child)
  - copies the parent's address space into the child's
  - starts a new thread of control in the child's address space
  - parent and child are equivalent -- almost
    - in parent, fork() returns a non-zero integer
    - in child, fork() returns a zero.
    - difference allows parent and child to distinguish
- int fork() returns TWICE!

Example Program using fork()

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Example

```
#include <stdio.h>
#include <sys/types.h>  /* Primitive System Data Types */


int main(int argc, char **argv)
{
   char *myName = argv[0];
   int cpid = fork();
   if (cpid == 0) { //Child
      sleep(5); //sleeps for 5 secs
      printf("My pid: %d I am a child of %s My parent 
pid: %d\n", getpid(), myName, getppid());
      exit(0);
   } else { //Parent
      printf("My pid: %d My child's pid: %d\n", 
getpid(),cpid);
      exit(0);
   }
}
```

What does this program print?

```c
#include <stdio.h>
#include <sys/types.h>  /* Primitive System Data Types */
#include <unistd.h>
#include <sys/wait.h>
int main(void)
{
  int i, pid, status;
  pid = getpid();
  fprintf(stdout, "parent pid = %d\n", pid);

  pid = fork();
  if (pid == 0) /* child process is always 0 */
  {
    for (i= 0; i < 10; ++i)
    {
      fprintf(stdout,"In child: Iteration: %d\n",i);
      sleep(0.1);
    }
    fprintf(stdout, "In child: child exiting\n");
  }
  else /* parent process is non-zero (child's pid) */
  {
    //sleep(2); //to force child to run first
    fprintf(stdout, "In Parent: child pid = %d\n", pid);
    fflush(stdout);
    fprintf(stdout, "In Parent: waiting for child\n");
    //wait(NULL); //wait for any child to change state
    //wait(&status); //status is stored here
    //waitpid(-1,&status,0); //wait for any child to change state
    waitpid(pid,&status,0); //wait for pid child to change state
    fprintf(stdout, "In Parent: Child exit status:%d\n",WEXITSTATUS(status));
    if(WIFEXITED(status))
      fprintf(stdout, "In Parent: Child exited normally\n");
    else if(WIFSIGNALED(status))
      fprintf(stdout, "In Parent: Child was killed by a signal!!\n");
    else
      fprintf(stdout, "In Parent: Child exited for other reasons\n");
    fprintf(stdout, "In Parent: child terminated\n");
    fprintf(stdout, "In Parent: parent exiting\n");
  }

  return 0;
}
```

Output of one of Runs: 1

```
parent pid = 26892
In Parent: child pid = 26893
In Parent: waiting for child
In child: Iteration: 0
In child: Iteration: 1
In child: Iteration: 2
In child: Iteration: 3
In child: Iteration: 4
In child: Iteration: 5
In child: Iteration: 6
In child: Iteration: 7
In child: Iteration: 8
In child: Iteration: 9
In child: child exiting
In Parent: Child exit status:0
In Parent: Child exited normally
In Parent: child terminated
In Parent: parent exiting
```

Output of one of Runs: 2

```
parent pid = 26898
In Parent: child pid = 26899
In child: Iteration: 0
In Parent: waiting for child
In child: Iteration: 1
In child: Iteration: 2
In child: Iteration: 3
In child: Iteration: 4
In child: Iteration: 5
In child: Iteration: 6
In child: Iteration: 7
In child: Iteration: 8
In child: Iteration: 9
In child: child exiting
In Parent: Child exit status:0
In Parent: Child exited normally
In Parent: child terminated
In Parent: parent exiting
```

Example

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
fprintf(stdout,"Parent PID:%d\n",getpid());
fflush(stdout);

while(1) {
fork();
fprintf(stdout,"My PID:%d and My Parent
                                    PID:%d\n",getpid(),getppid());
}
return 0; }
```

Example

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
fprintf(stdout,"Parent PID:%d\n",getpid());
fflush(stdout);

while(fork()) wait(NULL);
fprintf(stdout,"My PID:%d and My Parent
                                        PID:%d\n",getpid(),getppid());
return 0;
}
```

More Examples on fork( )

- http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html

- http://www.amparo.net/ce155/fork-ex.html

- http://home.adelphi.edu/sbloch/class/archive/271/fall2005/examples/c/fork_examples/

- http://man7.org/linux/man-pages/man2/vfork.2.html
  - Creates a child process and block parent till child finishes
  - Till finishes the child shares all memory with its parent, including the stack!

Fork is half the story

- Fork() gets us a new address space,
  - but parent and child share EVERYTHING
    - » memory, operating system state
- int exec(char *prgName) completes the picture
  - throws away the contents of the calling address space
  - replaces it with the program named by prgName
  - starts executing at header.startPC
  - Does not return on successful load of new prog!
  - Returns -1 if it fails to load new prog
- Pros: Clean, simple
- Con: duplicate operations

exec family

- execl("/bin/vi","vi","/home/user/ex1.txt",NULL); //PATH
- execlp("vi","vi","/home/user/ex1.txt",NULL); //File
- execle(path, arg1,…, envp[])
- execv(path, argv[])
- execvp(progName, argv[])
- execvpe(progName, argv[], envp[])
- l: list of Args, v: vector of Args
- p: Searches folders listed in PATH envr variable
- e: environment of progName to be specified
- The first argument should point to the PrgName associated with the file being executed.
- The list of arguments *must* be terminated by a null pointer
- execlp(), execvp(), and execvpe() duplicate actions of the shell in searching for an executable file if the specified PrgName does not contain / character

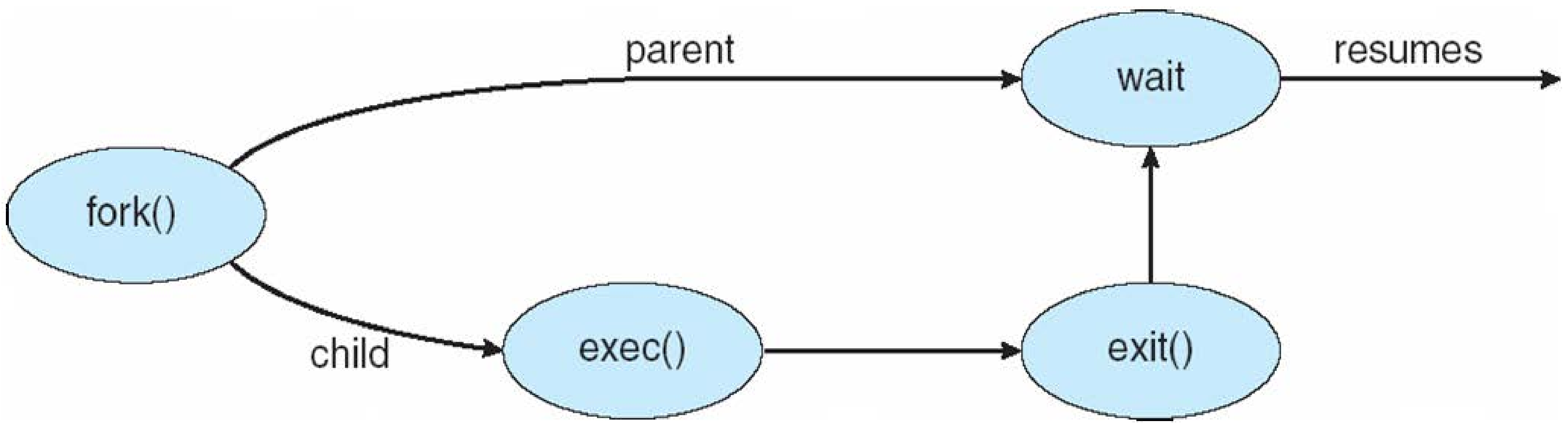
http://man7.org/linux/man-pages/man2/execve.2.html

https://www.gnu.org/software/libc/manual/html_node/Environment-Variables.html#Environment-Variables

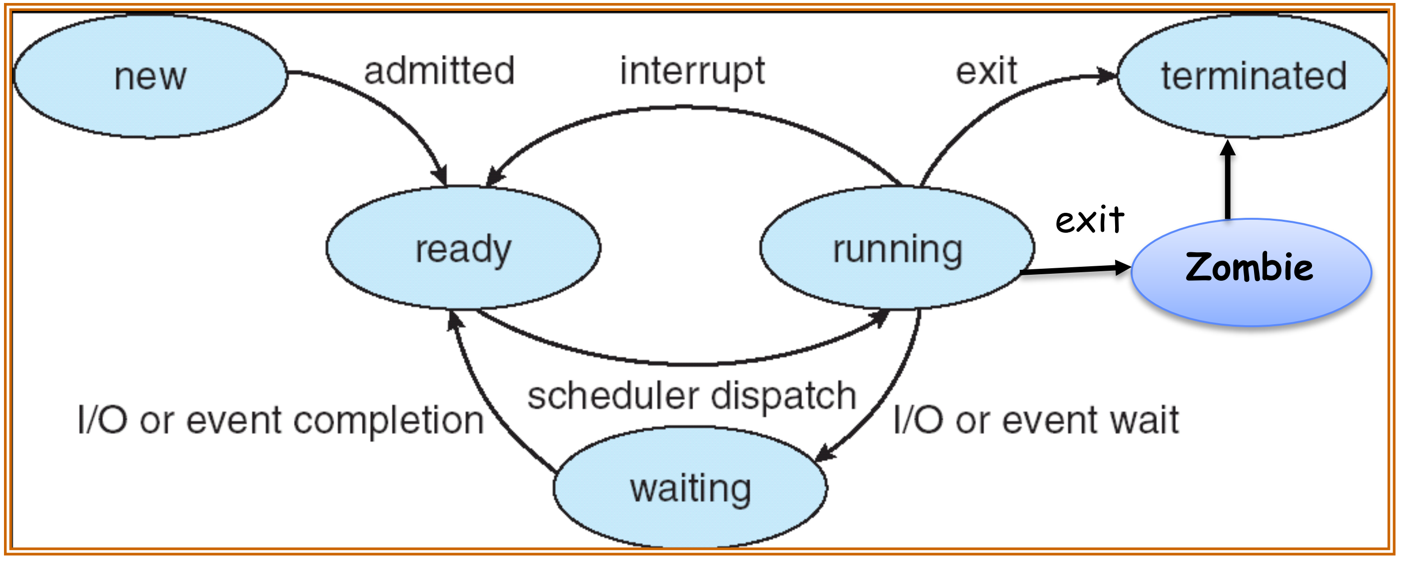
Process Termination

- Process executes last statement and OS decides(exit)
  - Output data from child to parent (via wait)
  - Process' resources are deallocated by operating system
- Child exits before parent calls up waitpid()
  - Child is in Zombie state amd has entry still maintained in PCB
  - It is removed later when parent calls waitpid( )
- Parent may terminate execution of child process (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting, orphan child
    - » Some OSes don't allow child to continue if parent terminates
      - All children terminated - *cascading termination*
      - *Linux: init becomes parent of these orphan childs*

Process Life Cycle



Process State Diagram



Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - Browser process manages UI, disk and network I/O
  - Renderer process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened on a tab
    - » Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits
  - Plug-in process for each type of plug-in

Multitasking in Mobile Systems

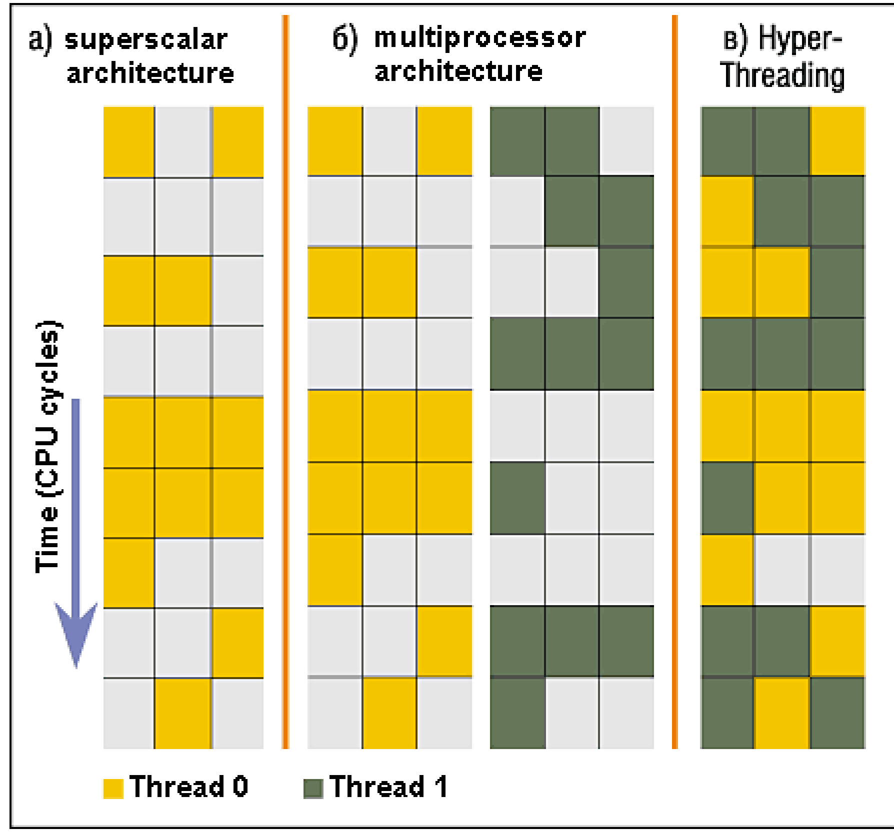
- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single foreground process- controlled via user interface
  - Multiple background processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a service to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Reading Assignment

- Chapter 3 from OSC by Galvin et al
- Chapter 2 from OSD&I by Tanenbaum et al
- Chapter 3. Processes and Chapter 19. Process Communication from Understanding the Linux Kernel by Daniel P. Bovet and Marco Cesati (available on Intranet)
- http://man7.org/linux/man-pages/man2/kill.2.html
- https://www.gnu.org/software/libc/manual/html_node/index.html
- https://www.chromium.org/developers/design-documents/process-models
- http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html
  - time cmd
  - /usr/bin/procinfo

Modern Technique: SMT/Hyperthreading

- Hardware technique
  - Exploit natural properties of superscalar processors to provide illusion of multiple (logical) processors on each CPU/core (socket)
  - Higher utilization of CPU resources
- Can schedule each thread as if were separate CPUs
  - However, not linear speedup!
  - If have multiprocessor, should schedule each processor first



- Original technique called "Simultaneous Multithreading"
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 ("Hyperthreading"), Xeon servers, Intel Core i & Skylake server series