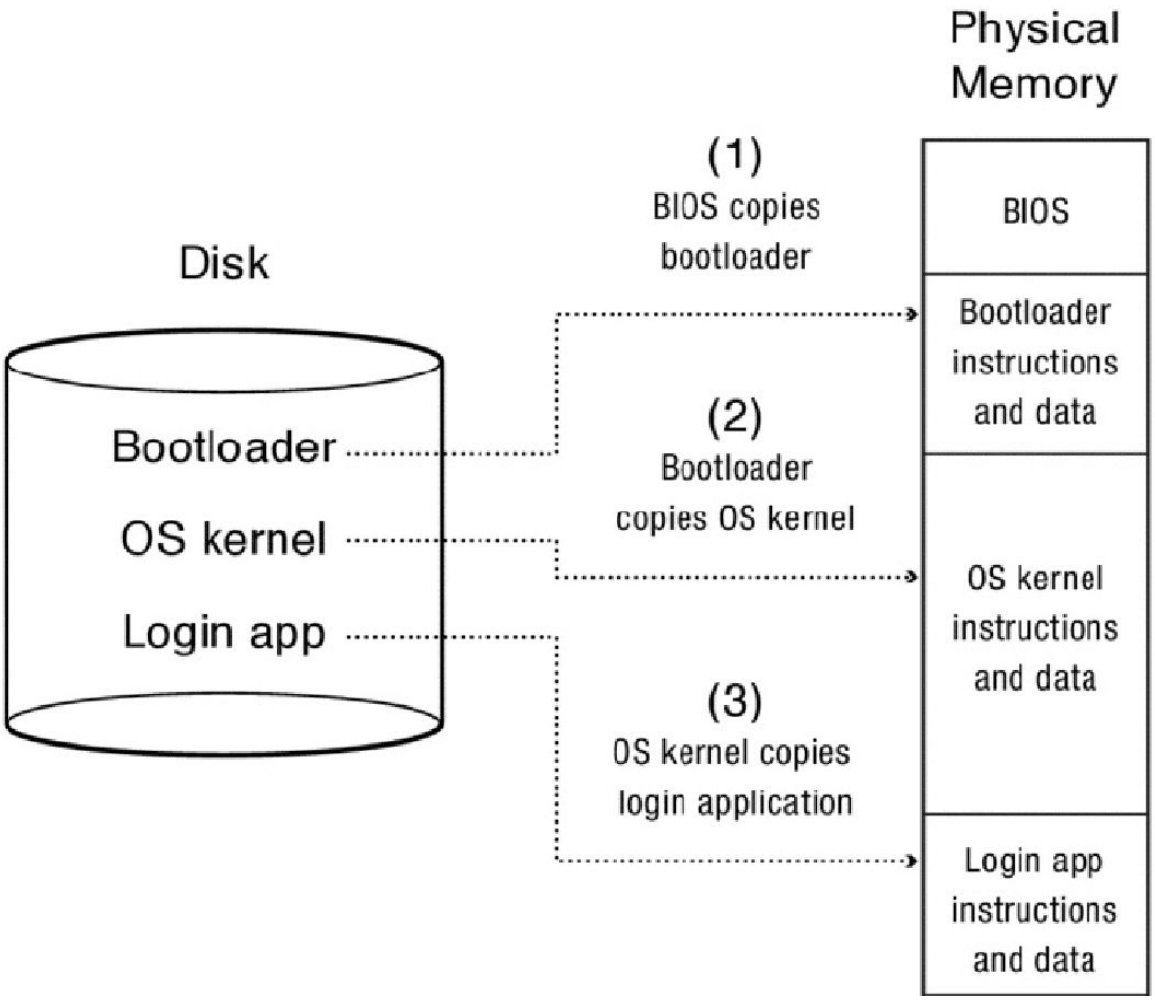
Introduction to Process

Booting an Operating System Kernel



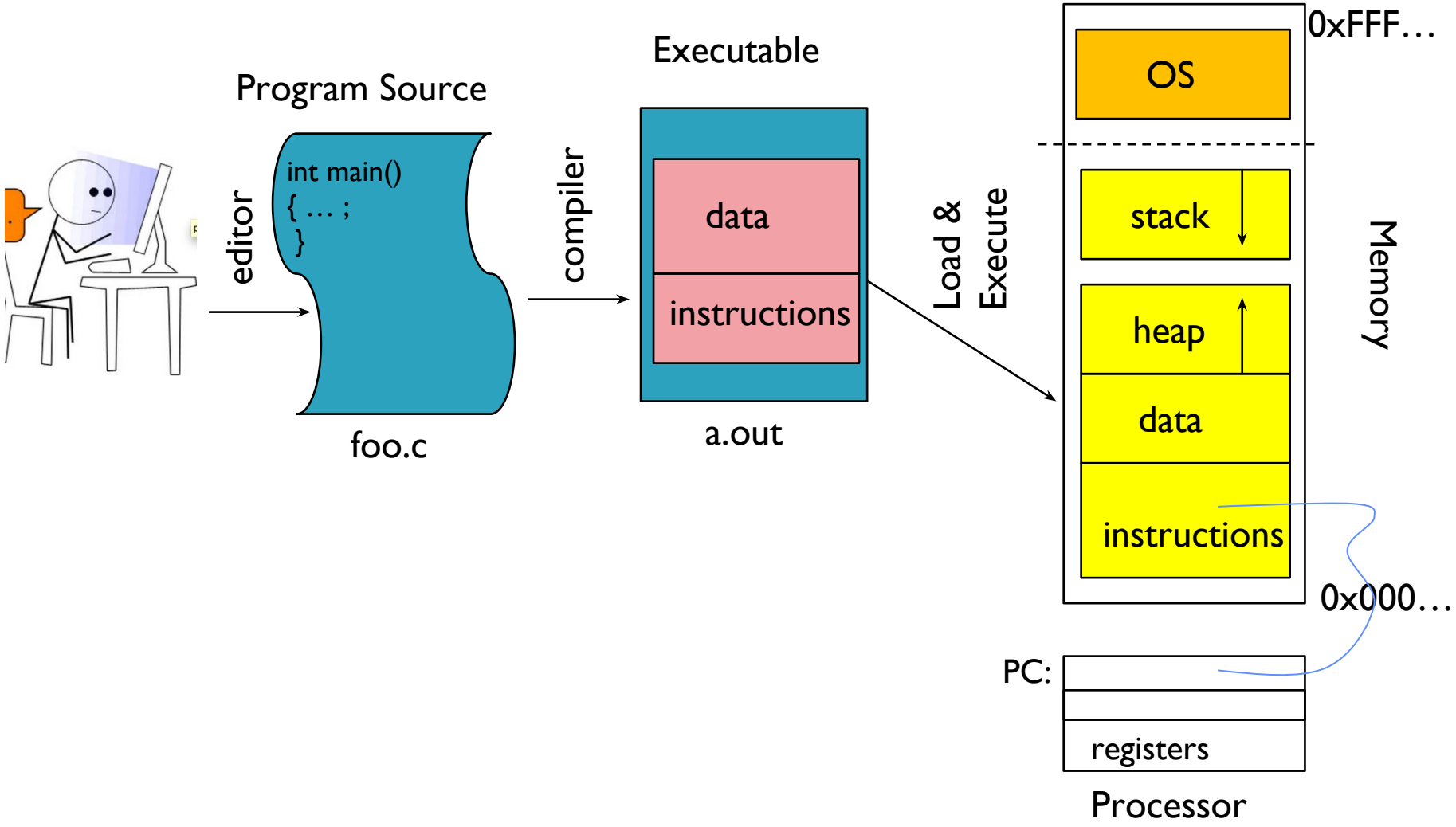
Memory Layout of C Programs

```
                $ gcc memory-layout.c -o memory-layout
                $ size memory-layout
text        data         bss         dec         hex    filename
 960         248          16        1224         4c8    memory-
```

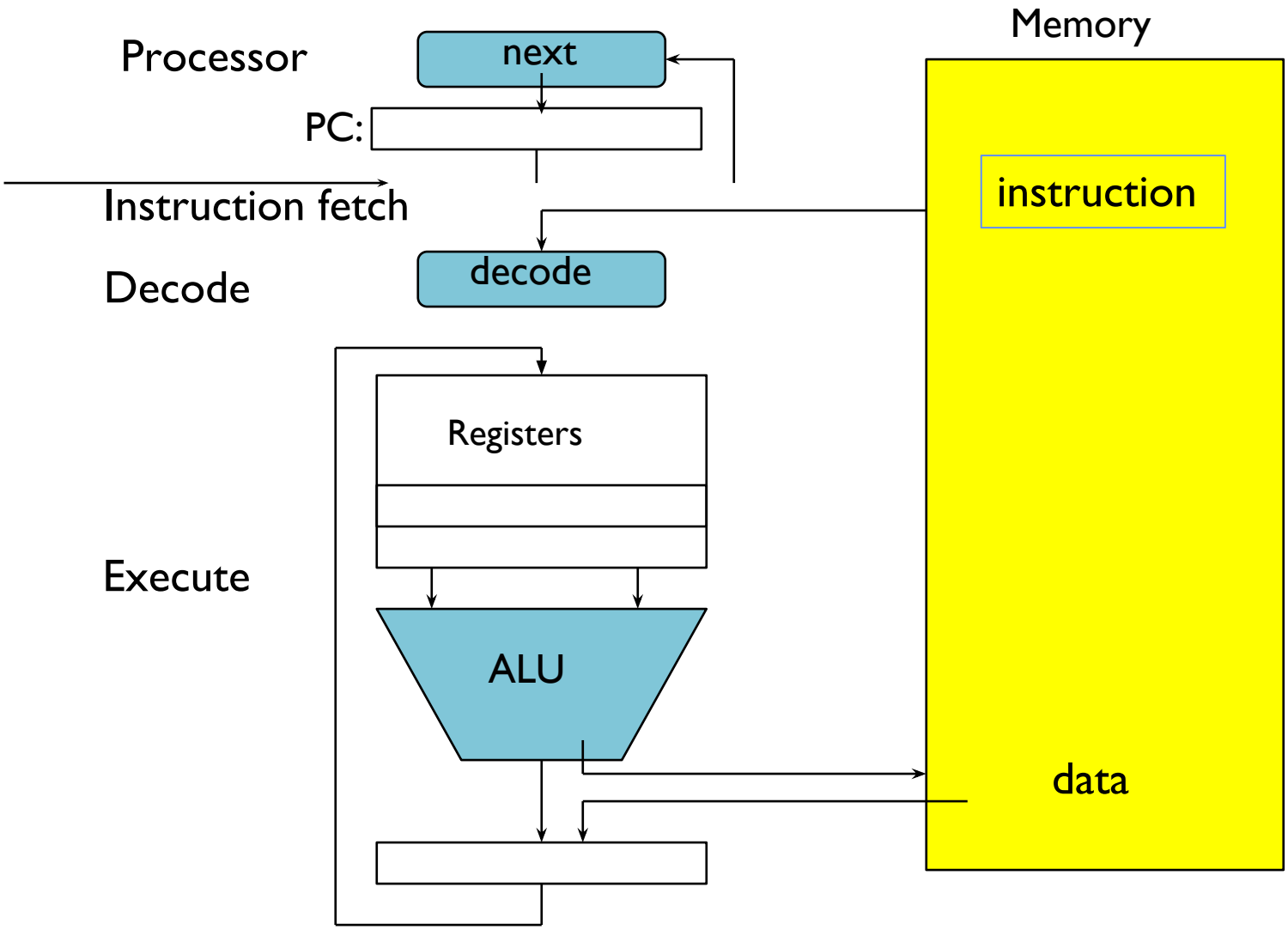
OS Bottom Line: Run Programs

- 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

OS Bottom Line: Run Programs



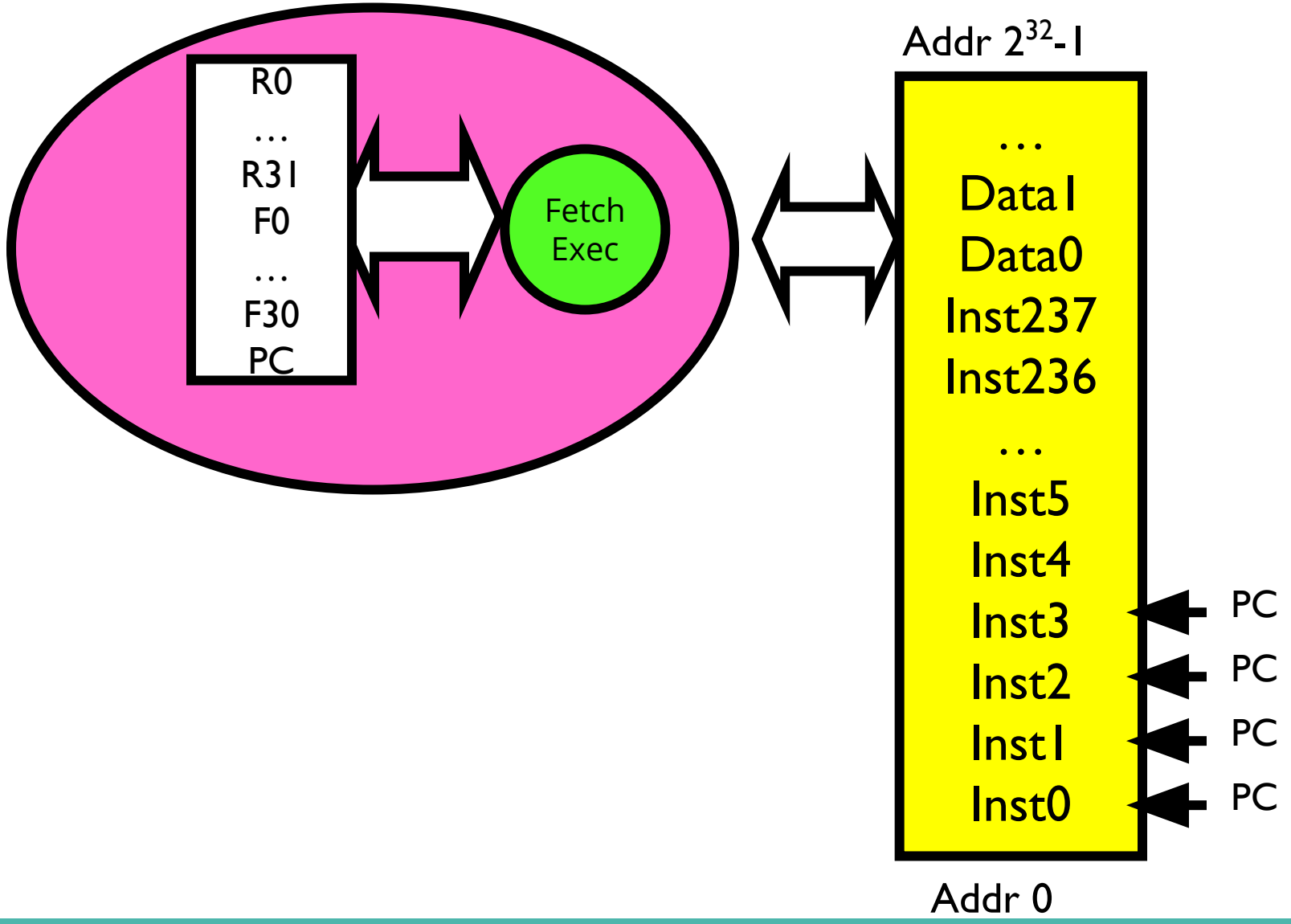
Fetch/Decode/Execute The Instruction Cycle



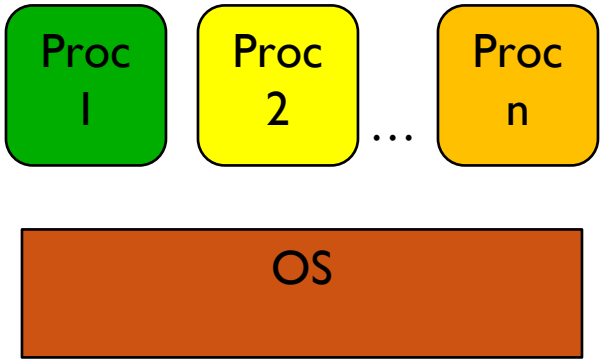
What happens during program execution?

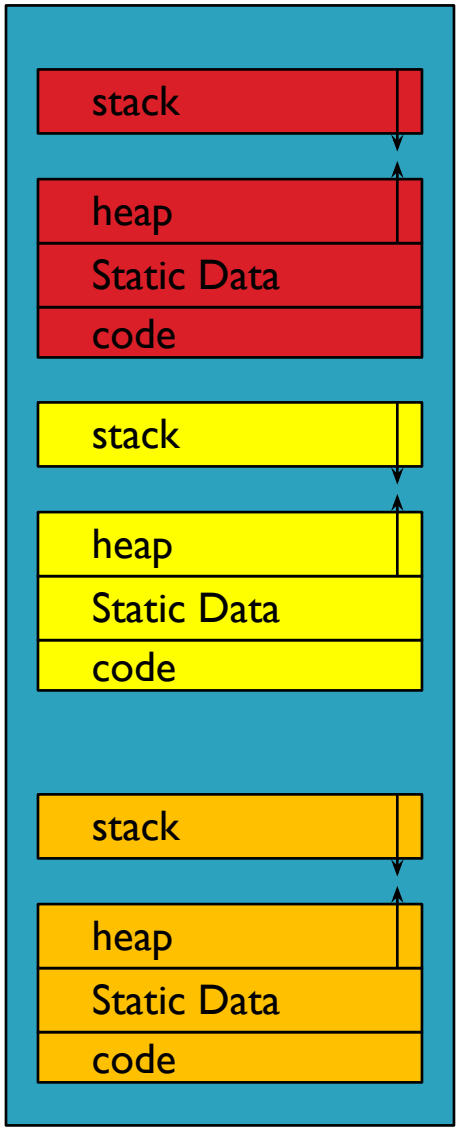
- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

What happens during program execution?



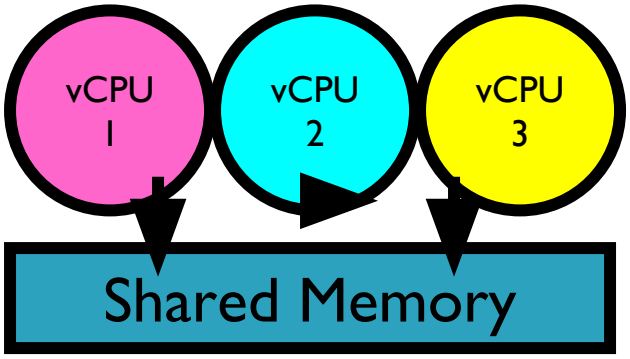
Multiprogramming

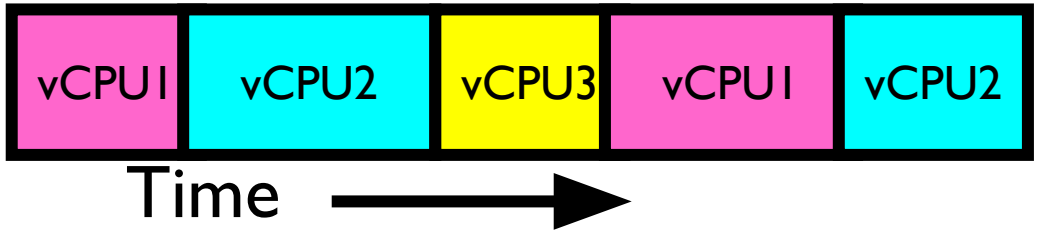




How can we give the illusion of multiple processors?

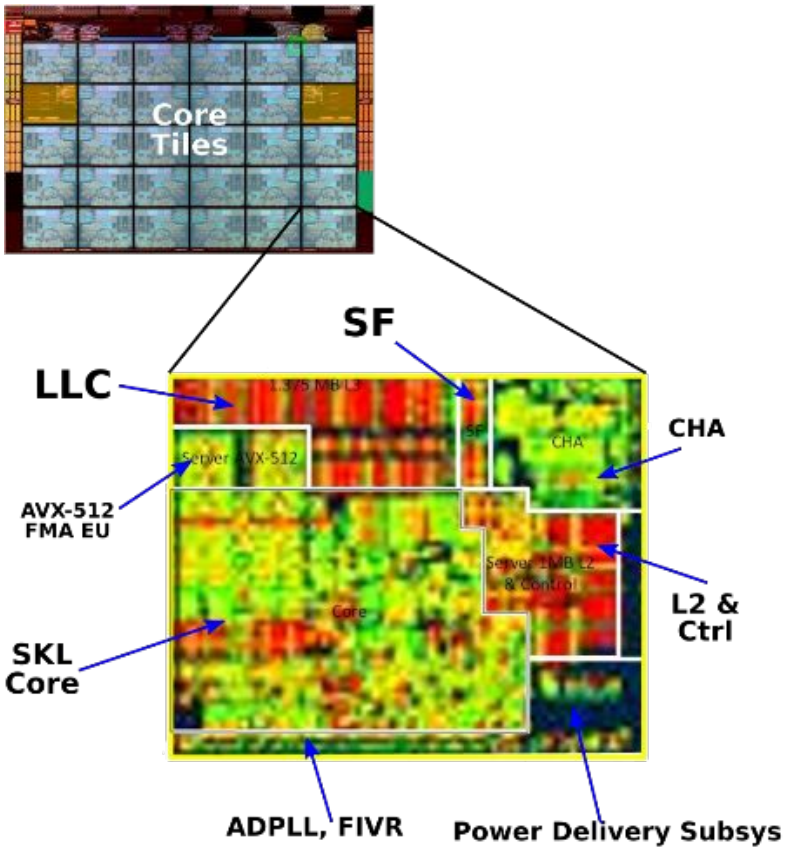
- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual “CPU” needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one virtual CPU to the next?
  - 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 World Is Parallel

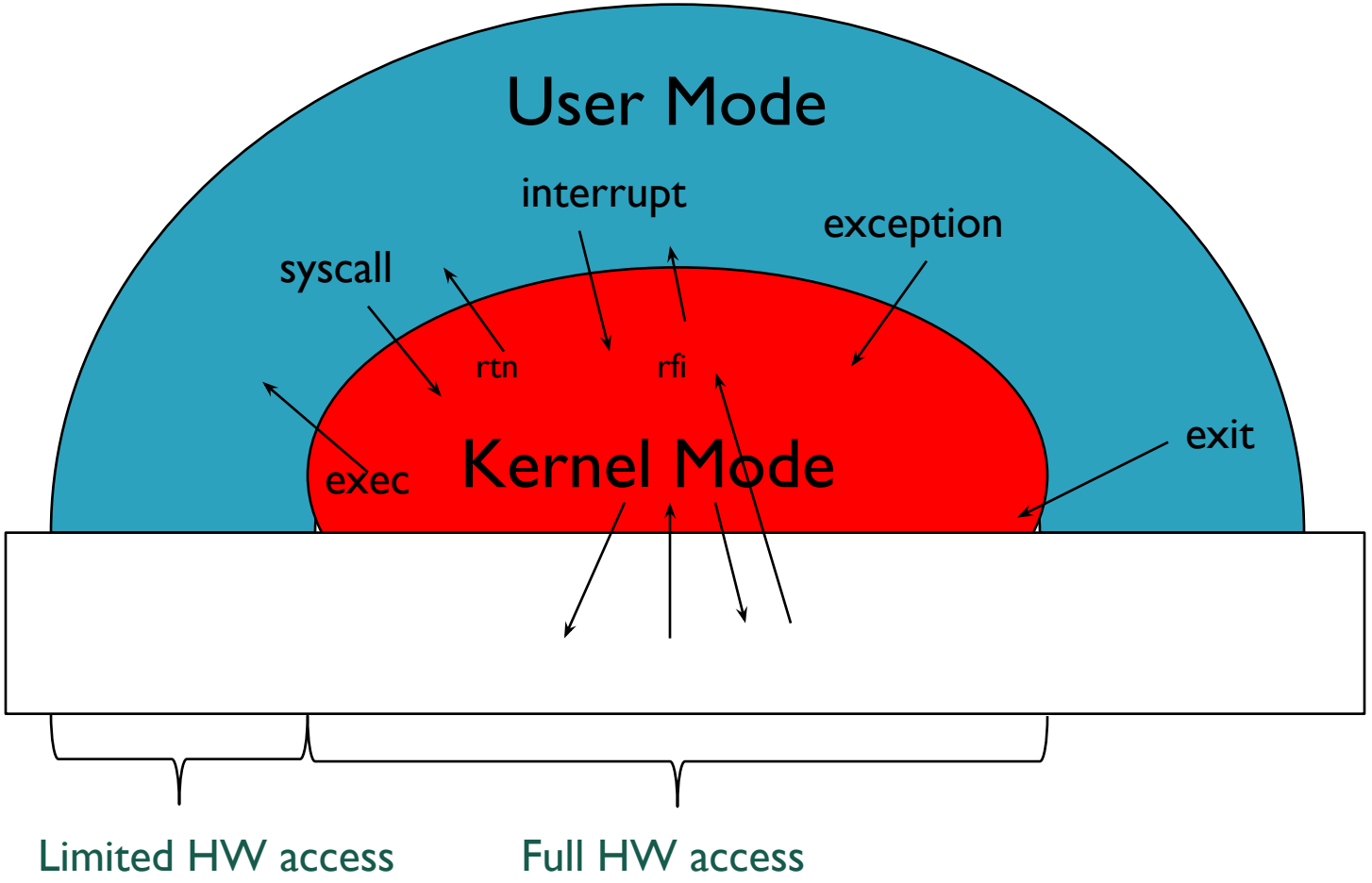
- Intel Skylake (2017)
  - 28 Cores
  - Each core has two hyperthreads!
  - So: 54 Program Counters(PCs)
- Scheduling here means:
  - Pick which core
  - Pick which thread



3 types of Kernel Mode Transfer

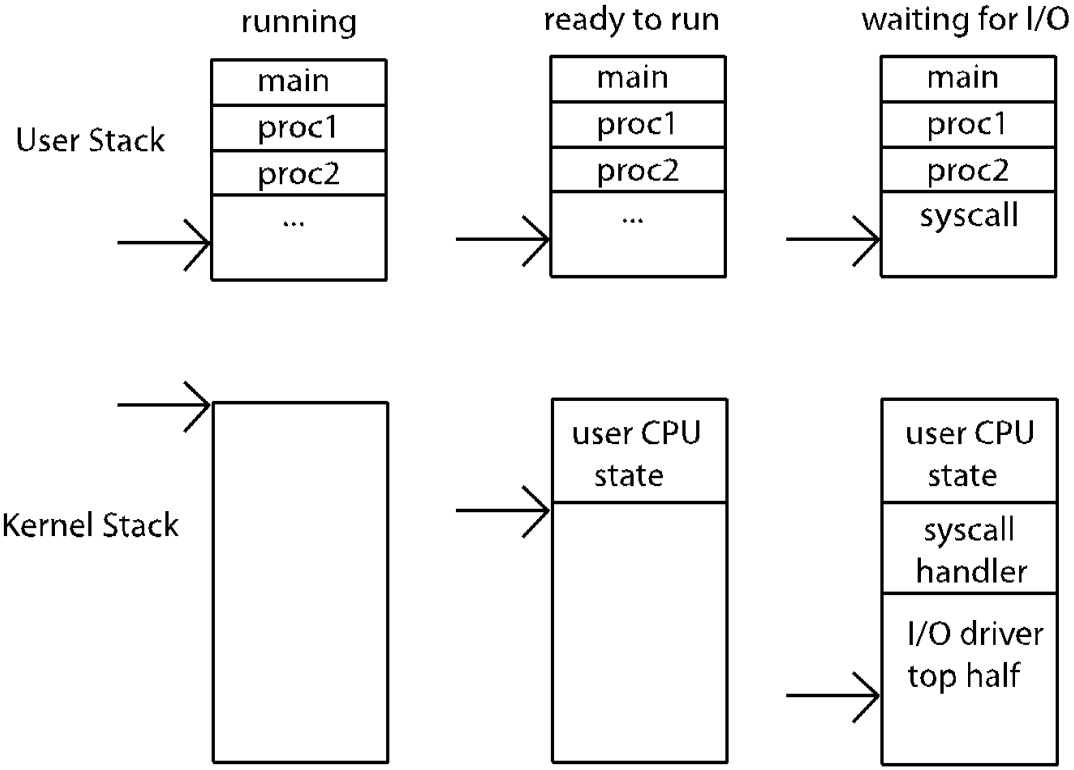
- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but “outside” the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt
  - External asynchronous event triggers context switch
  - eg. Timer, I/O device
  - Independent of user process
- Trap or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …

User/Kernel (Privileged) Mode

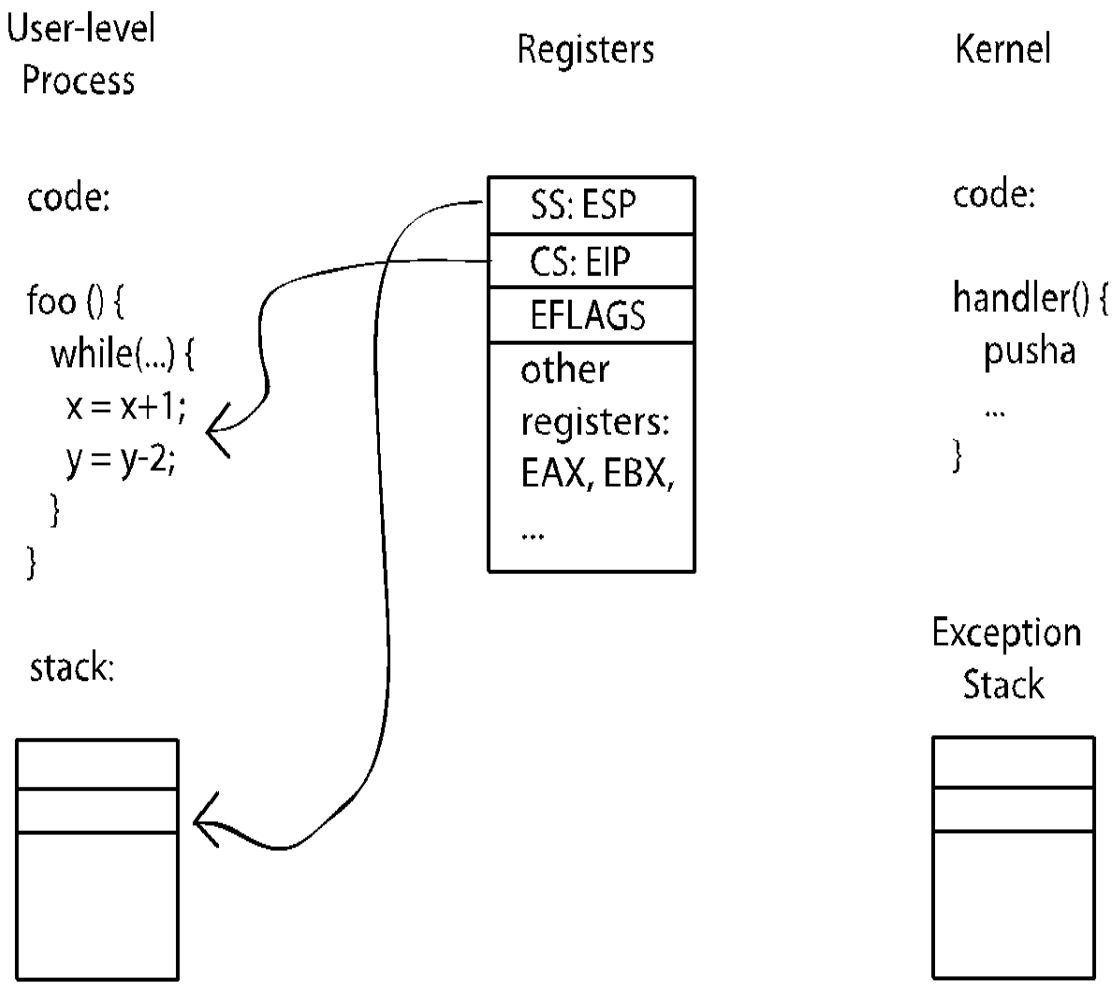


Need for Separate Kernel Stacks

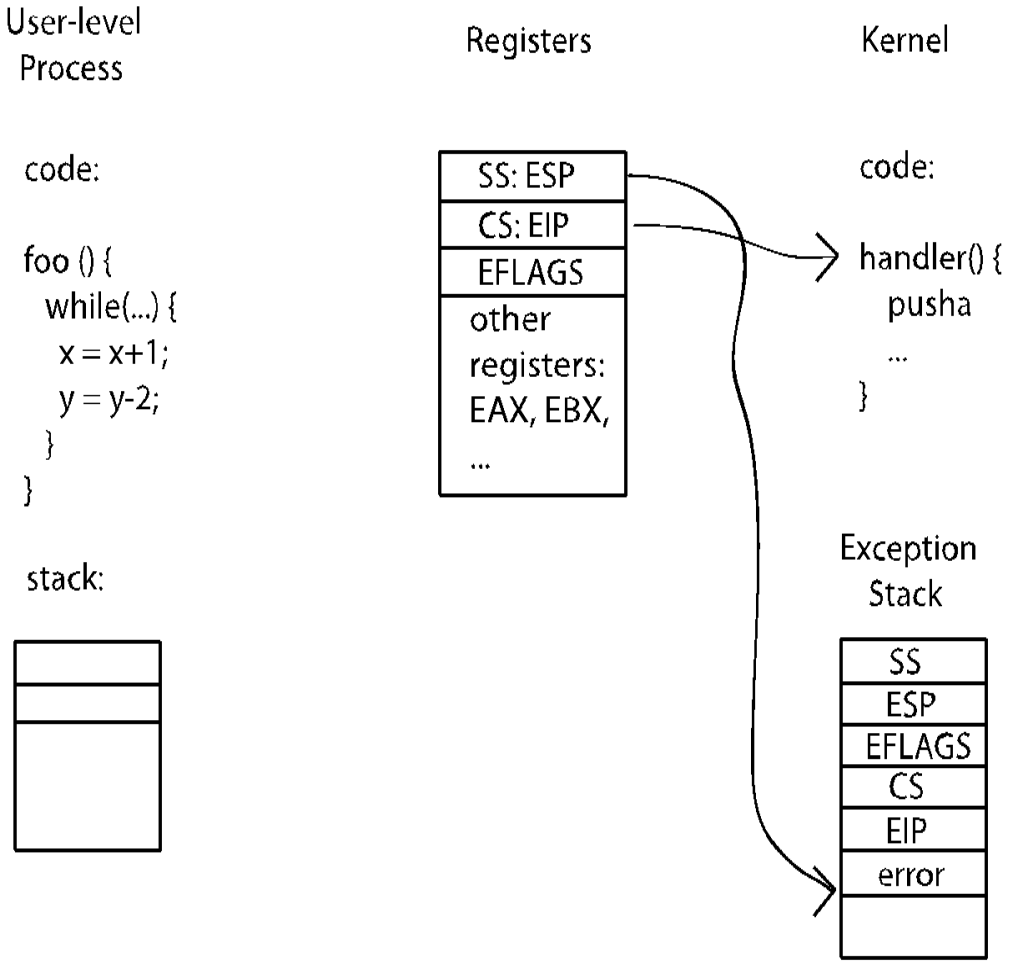
- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model



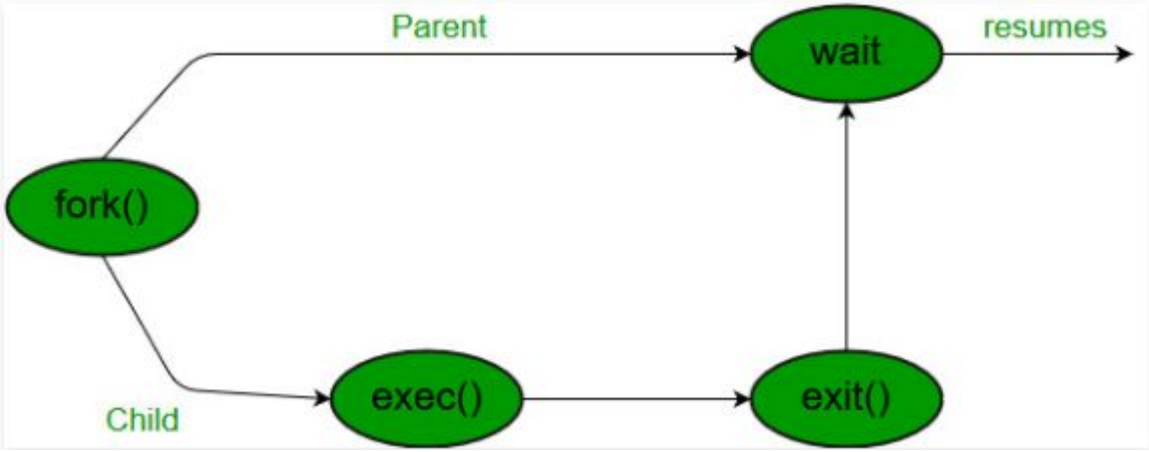
Before


User-level
Process
code:
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
stack:
Registers
SS: ESP
CS: EIP
EFLAGS
other
registers:
EAX, EBX,
...
Kernel
code:
handler() {
pusha
...
}
Exception
Stack

During


User-level Process
code:
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
stack:
Registers
SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...
Kernel
code:
handler() {
pusha
...
}
Exception Stack
SS
ESP
EFLAGS
CS
EIP
error

Can a process create a process ?



Can a process create a process ?

- Yes! Unique identity of process is the “process ID” (or PID)
- **fork()** system call creates a *copy* of current process with a new PID
- Return value from **fork()**: integer
  - When > 0:
  - Running in (original) Parent process
  - return value is pid of new child
  - When = 0:
  - Running in new Child process
  - When < 0:
  - Error! Must handle somehow
  - Running in original process
- State of original process duplicated in *both* Parent and Child!
  - Memory, File Descriptors (next topic), etc…

Create Process: fork1.c

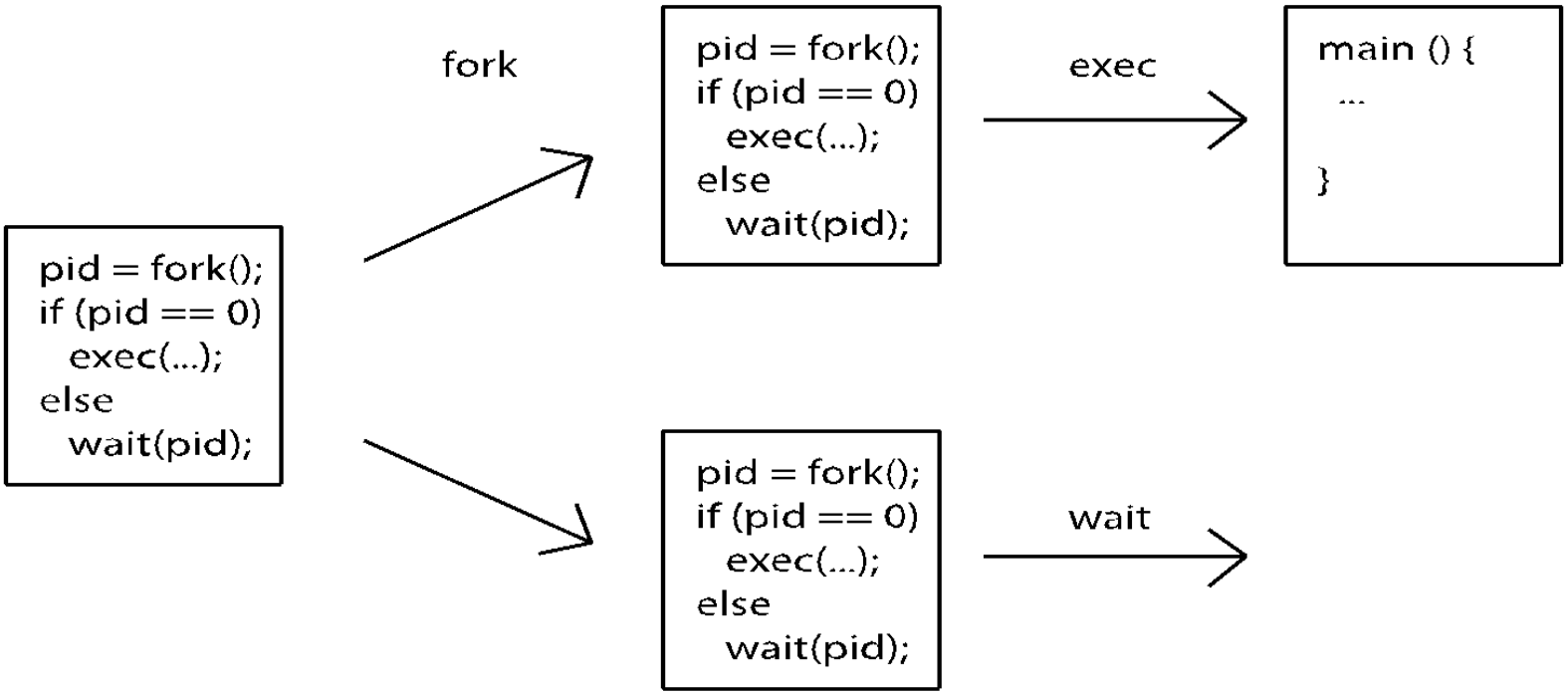
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
  pid_t cpid, mypid;

  pid_t pid = getpid();             /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {            /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  }  else if (cpid == 0) {      /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
    exit(1);
  }
  exit(0);
}
```

Exec Process

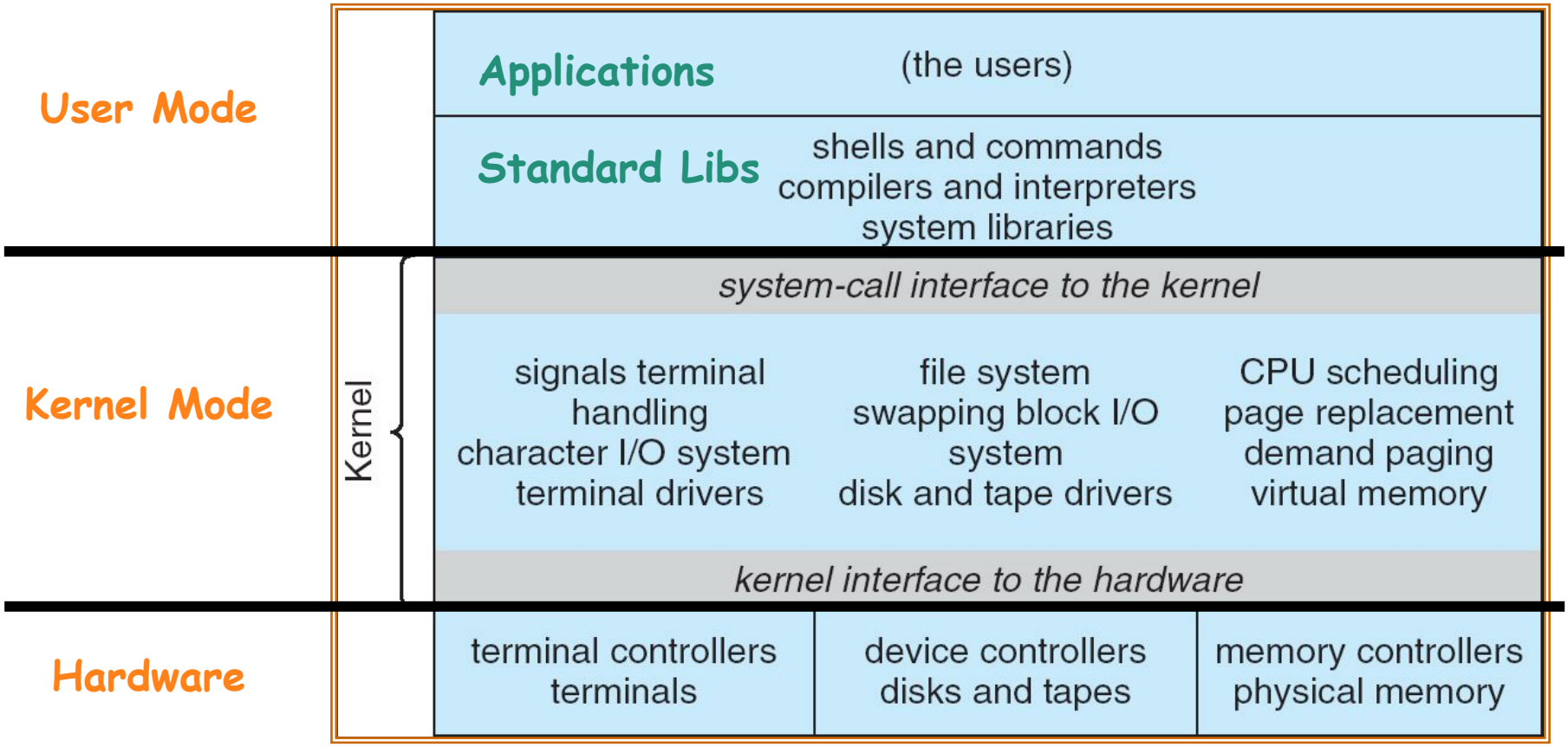
```
int execv(const char *path, char *const argv[]);
```

```
#include <unistd.h>

int main(void) {
  char *binaryPath = "/bin/ls";
  char *args[] = {binaryPath, "-lh", "/home", NULL};

  execv(binaryPath, args);

  return 0;
}
```

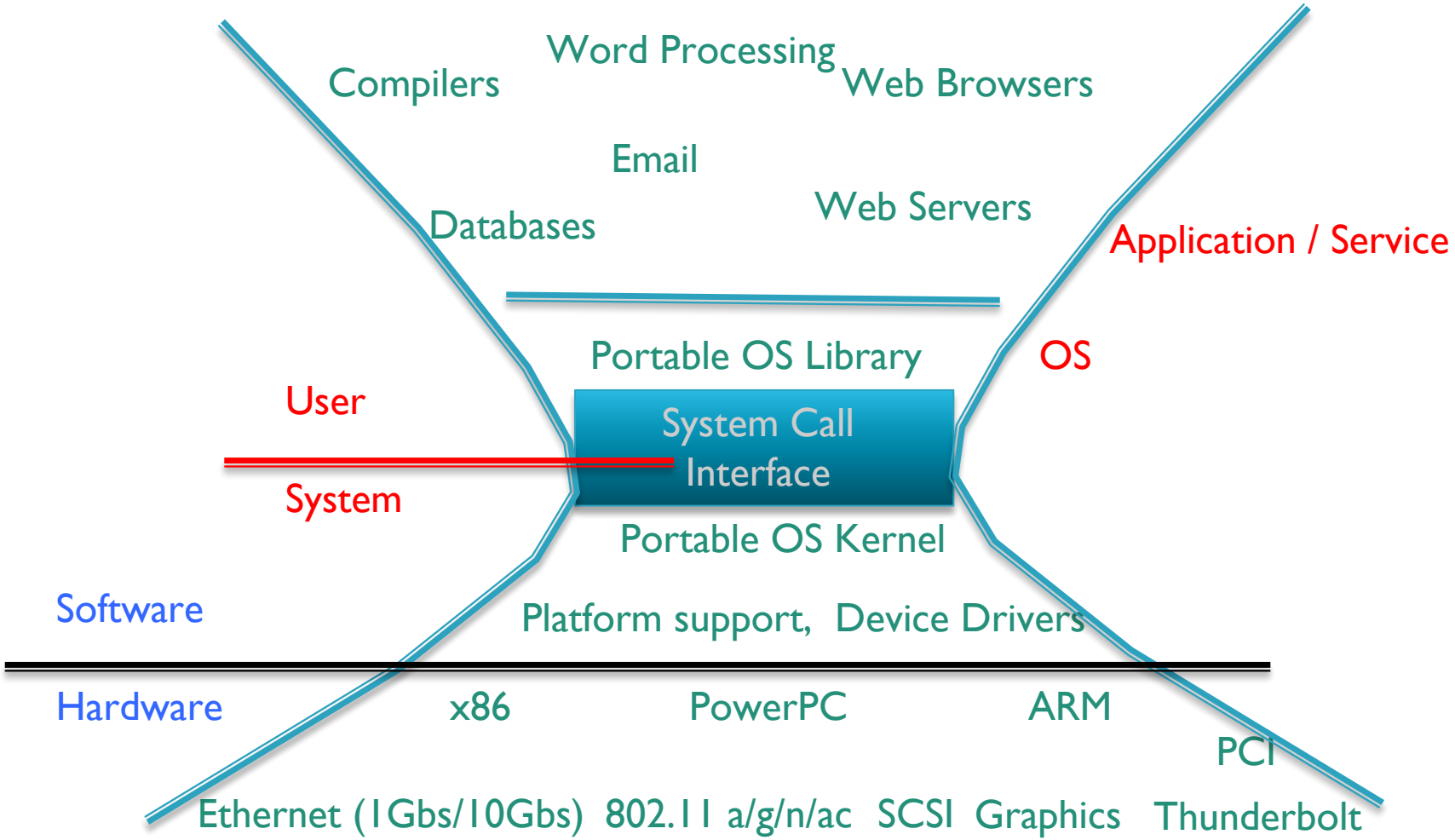
UNIX Process Management



UNIX System Structure



A Kind of Narrow Waist



UNIX Process Management

- `ps aux | grep  process_name`
- `ps -p process_id`
- `ps j          // list all process_parent_id`
- `Pstree       // list tree view of processes`
- `ls -la /proc/3956/`

UNIX Process Management

```
#!/bin/sh
P=$1
if [ -z "$P" ]; then
    read P
fi
cat /proc/"$P"/status | grep PPid
```