

Chapter 4

Process Abstraction

Dong Dai

CIS/UD

dai@udel.edu

Crux of the Problem

Although there is **ONLY ONE** physical CPU available, how can OS provide the **illusion** of a nearly-endless supply of said CPUs?

Process

- One fundamental **abstraction** that OS provides to users
- Process is a **running program**
 - a program (**executable**, a.out) itself is a "lifeless" thing sitting on disk with instructions (and static data)
 - OS **transforms** a program into a running process
- Need to run many processes at once
- Problem: **how to provide the illusion of many CPUs?**
 - virtualize the CPU via "time" sharing
 - what is its "cost?" (there is no free lunch!)
 - performance
 - scheduling policy (high-level intelligence) & context switching mechanism (low-level machinery)

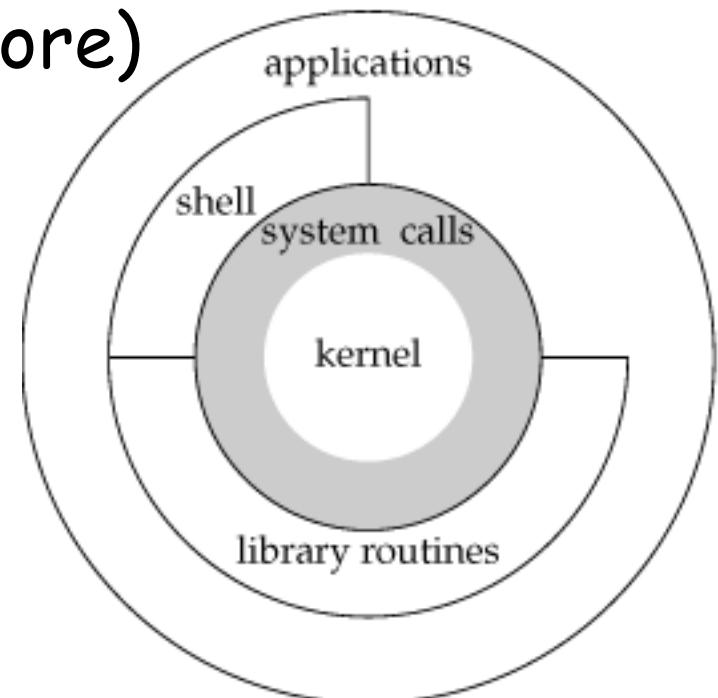
Process

- Process: **abstraction** [provided by OS] of a running program
- What constitute a **process**?
 - what pieces of a computer system will a process access or affect during the course of its execution?
 - summarized by an “inventory” of these pieces
- **Machine state** (at any instance in time) per process
 - CPU registers (including PC/IP, SP, FP)
 - memory that the process can access: **address space**
 - opened files
 - etc.



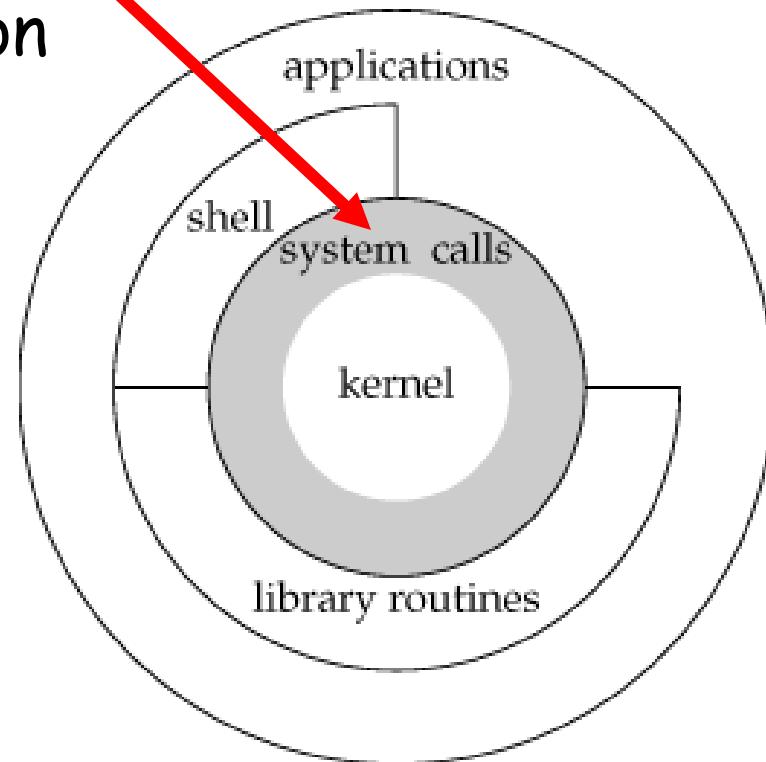
Architecture of Unix

- OS is the **software** that
 - controls the hardware resources of the computer
 - provides an **environment** under which programs can run
- **OS kernel** (residing at the core)
- **System calls**: a layer of **software** providing **interface to kernel**
- Library functions, **shells**, application programs, etc.

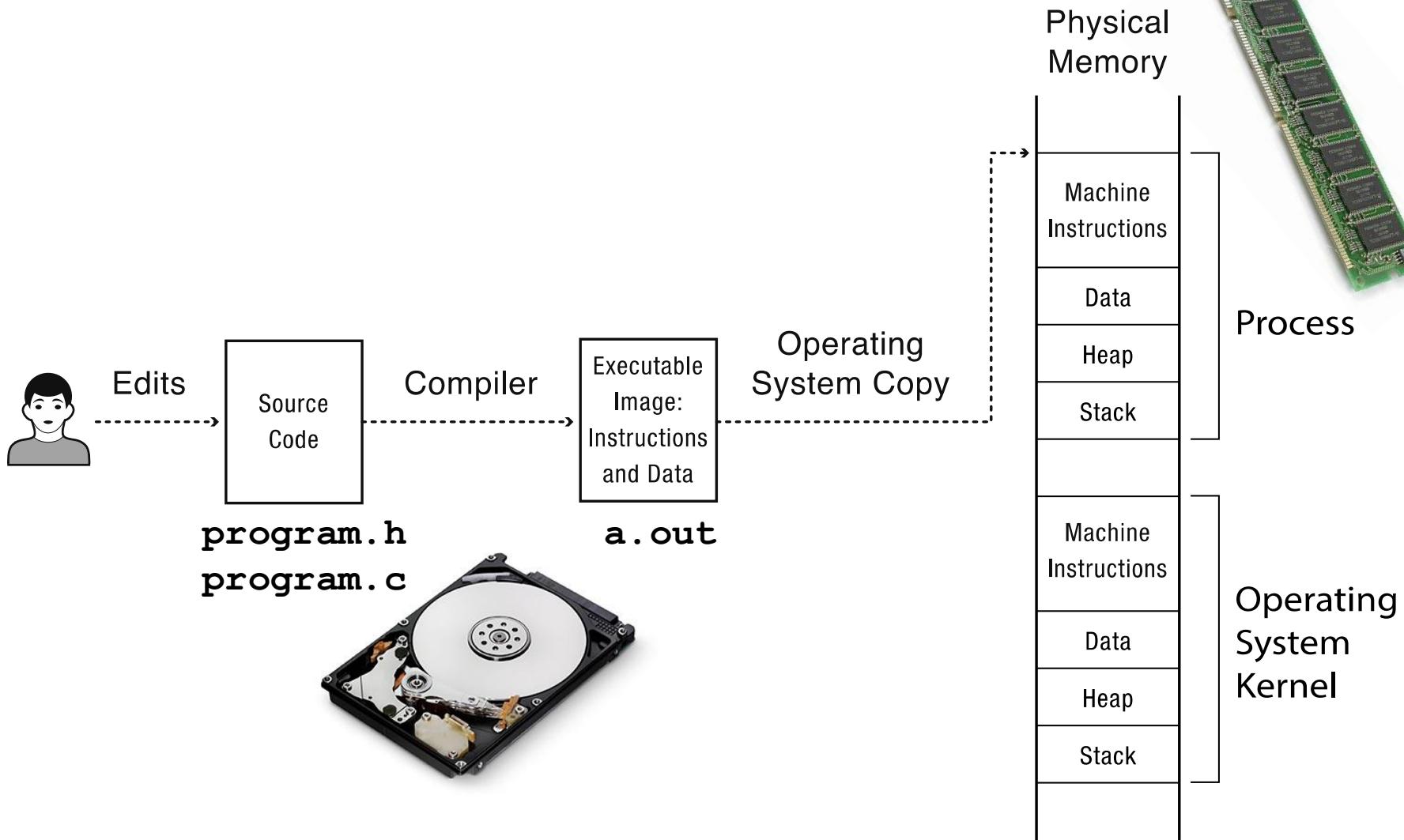


Process API

- Interface (system calls) of an OS for processes
 - create - e.g., a process is created when an application icon is double-clicked or a .out is executed
 - destroy
 - wait
 - miscellaneous control - suspend and resume
 - status

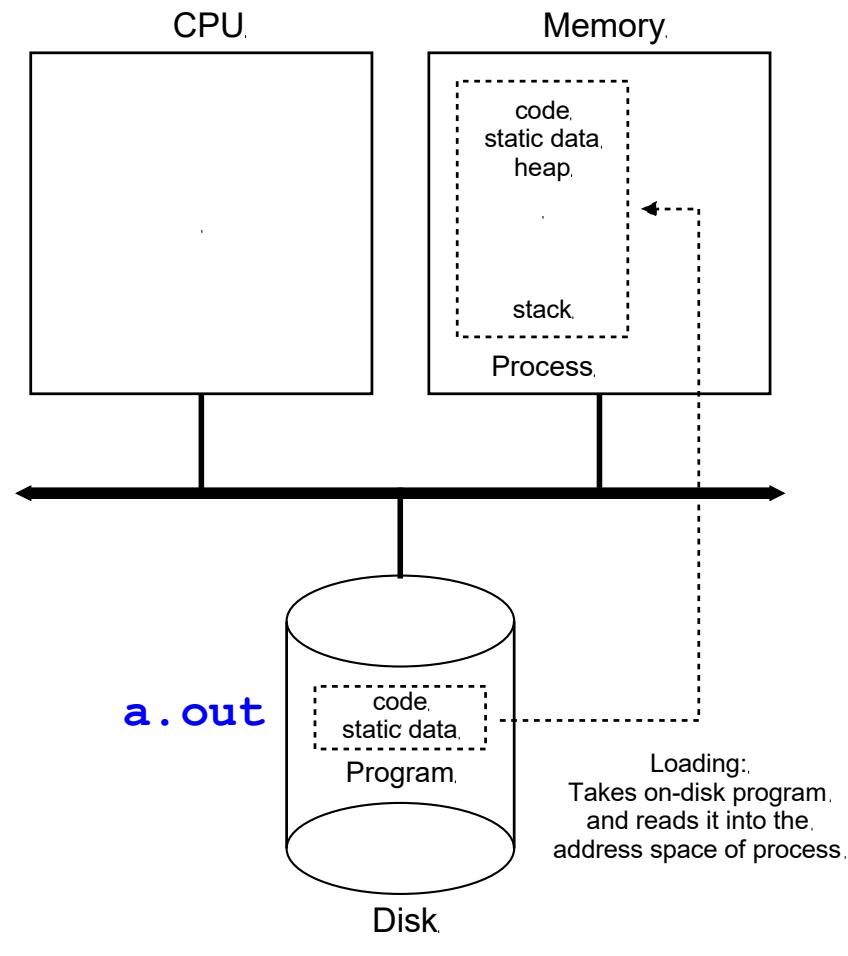


From Program to Process



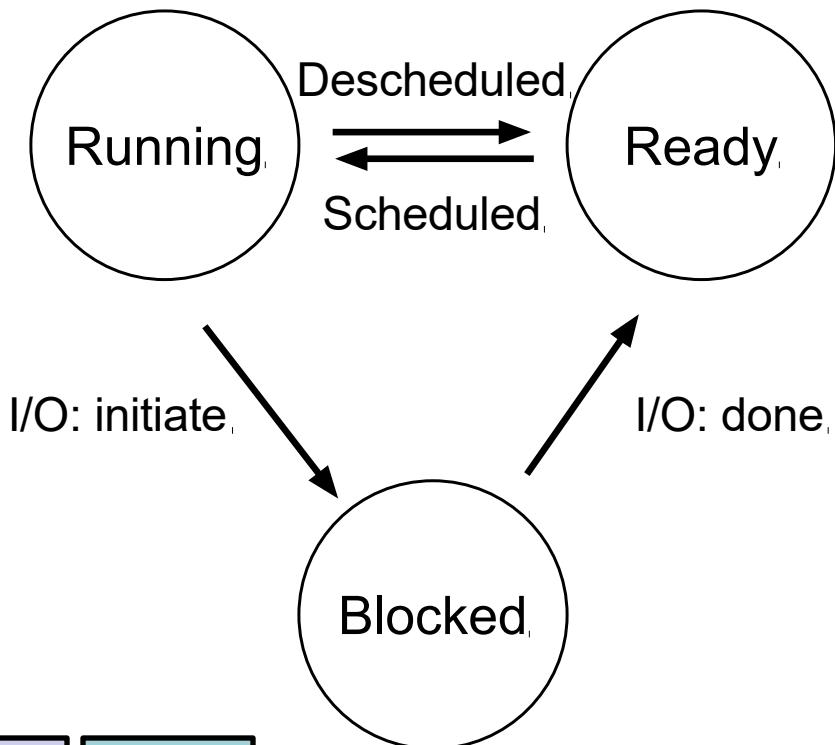
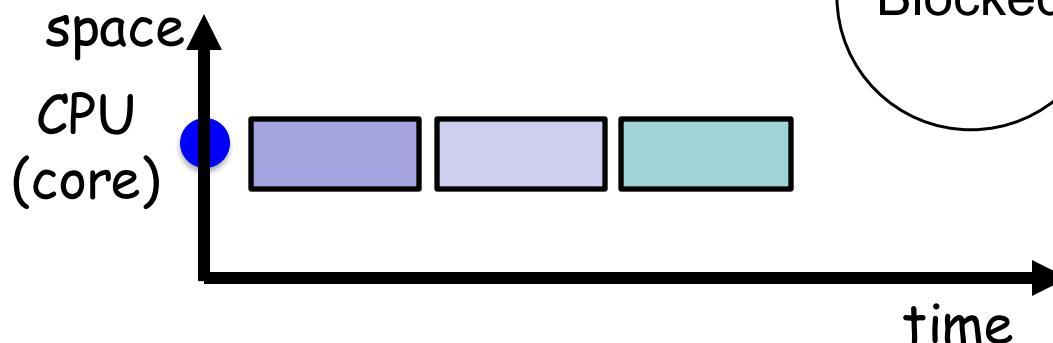
Process Creation

- How programs are transformed into processes?
 - **load** code and static data [executable (`a.out`)] into memory (address space)
 - **allocate** and **initialize** (run-time) stack with `argc` & `argv`
 - **initialize** 3 default file descriptors (0, 1, and 2)
 - **jump to** `main()` - now OS transfers control of CPU to the newly-created process and program starts running



Process States (Life Cycle)

- **Running** - using the CPU
- **Ready** - someone else is using the CPU
- **Blocked** - not ready to run until some other event takes place (e.g., initiated an I/O request to disk)



Tracing of Process States

Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process ₀ now done
5	—	Running	
6	—	Running	
7	—	Running	
8	—	Running	Process ₁ now done

computation only

Scheduler (within kernel)
makes scheduling
decisions

computation
and I/O

Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Blocked	Running	
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process ₁ now done
9	Running	—	
10	Running	—	Process ₀ now done

Data Structures of OS

- Program = Algorithms + Data structures
- OS is a program itself, containing data structures to **keep track of processes**
- **PCB** (Process Control Block) of a process
 - register context
 - process state
 - where it resides in memory
 - where its executable image resides on disk
 - etc.
- Lists (or queues) of PCBs maintained by kernel
 - running queue (single CPU)
 - ready queue
 - I/O queues

Context & PCB in xv6 (proc.h)

```
2326 struct context {  
2327     uint edi;  
2328     uint esi;  
2329     uint ebx;  
2330     uint ebp;  
2331     uint eip;  
2332 };  
2333 };
```

register context (for context switching)

```
2334 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };  
2335  
2336 // Per-process state  
2337 struct proc {  
2338     uint sz;                                // Size of process memory (bytes)  
2339     pde_t* pgdir;                            // Page table  
2340     char *kstack;                            // Bottom of kernel stack for this process  
2341     enum procstate state;                  // Process state  
2342     int pid;                                // Process ID  
2343     struct proc *parent;                   // Parent process  
2344     struct trapframe *tf;                   // Trap frame for current system call  
2345     struct context *context;                // Context switcher (here to run process switch)  
2346     void *chan;                             // If non-zero, sleeping on channel  
2347     int killed;                            // If non-zero, have been killed  
2348     struct file *ofile[NFILE];              // Open files  
2349     struct inode *cmd;                     // Current directory
```

(Stevens & Rago's Section 4.23)

Process Control Block (PCB)

Background X86

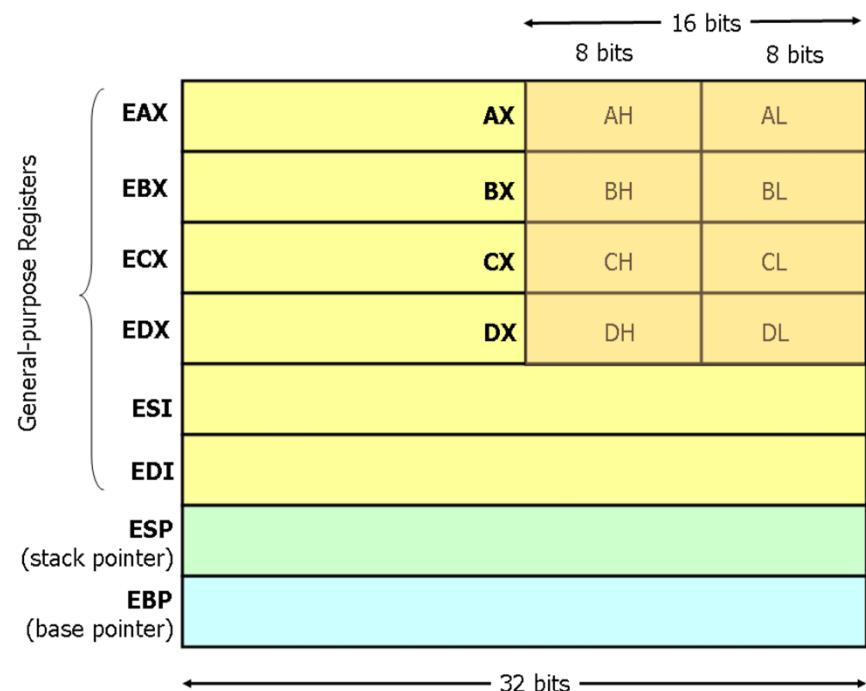
- General registers
 - EAX EBX ECX EDX
- Segment registers
 - CS DS ES FS GS SS
- Index and pointers
 - ESI EDI EBP EIP ESP
- Indicator
 - EFLAGS

Background X86

- Several Key Registers:
 - "cs" has the address of code segment of current app
 - "eip" is the instruction pointer inside code segment
 - "ss" is the address of the stack segment of current app
 - "ebp" is the stack base pointer within the stack segment
 - "esp" is the stack top pointer within the stack segment

X86 ASM

- Examples (AT&T, used in Linux)
 - `movl %eax, %edx : edx = eax`
 - `movl %0x123, %edx: edx = 0x123`
 - `movl 0x123, %edx: edx = *(int *)0x123`
 - `movl (%ebx), %edx: edx = *(int *)ebx`
 - `movl 4(%ebx), %edx: edx = *(int *)(ebx + 4)`
- Ops
 - `pushl %eax: subl %4, %esp; movl %eax, (%esp)`
 - `popl %eax: movl (%esp) %eax; subl %4, %esp`
 - `call 0x12345: pushl %eip; movl $0x12345, %e`
 - `ret: popl %eip`
 - `leave: movl %ebp, %esp; popl %ebp`
 - `enter: push %ebp; movl %esp, %ebp`



An Example: Application Runs

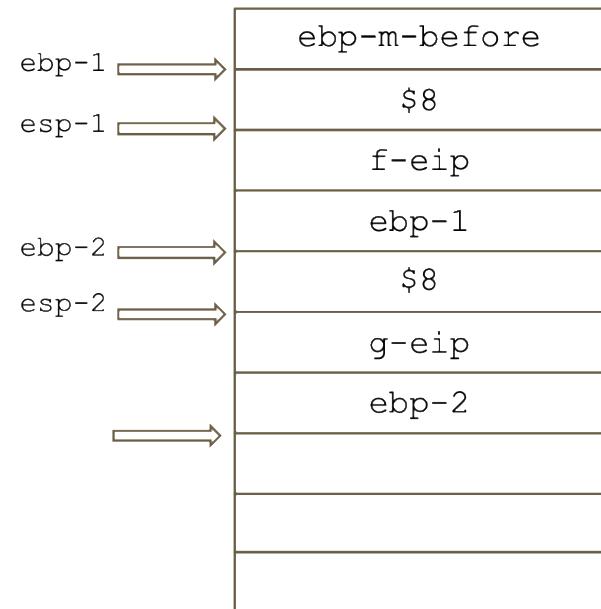
```
int g(int x)
{
    return x + 3;
}
int f(int x)
{
    return g(x);
}
int main(void)
{
    return f(8) + 1;
}
```

g:

```
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
addl $3 %eax
popl %ebp
Ret
```

f:

```
pushl %ebp
movl %esp, %ebp
subl $4, %esp
movl 8(%ebp), %eax
movl %eax, (%esp)
call g
leave
Ret
```



Summary

- **Process:** the most basic abstraction of OS
 - a running program
- Low-level mechanisms needed to implement processes
- Higher-level policies required to schedule processes in an intelligent way
- Together, OS could **virtualize** the CPU