Engineering & Computer Science
Syracuse University

CIS 657 – Principles of Operating Systems

Topic: Process – Mechanism
(Limited Direct Execution)

Endadul Hoque

Acknowledgement

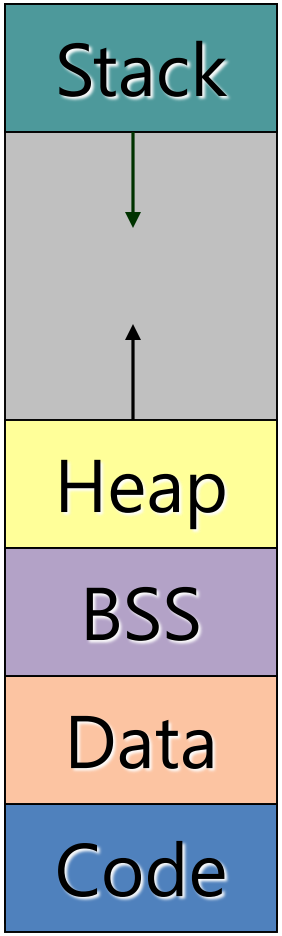
- Youjip Won (Hanyang University)
- OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)

Background: memory layout + function calling

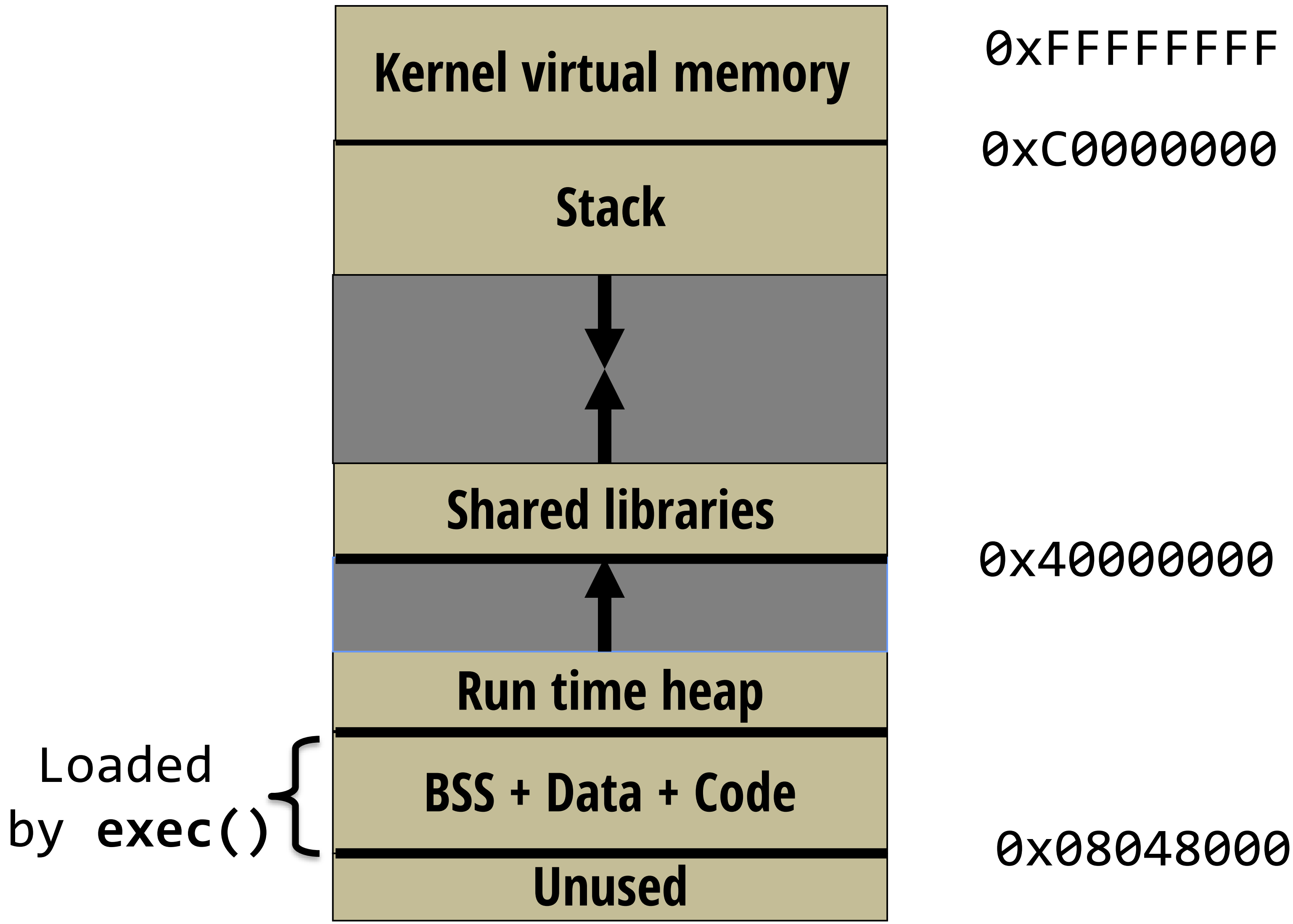
Revisit: Memory Layout of a Process

- The operating system creates a **process** by assigning memory and other resources
- **Code:** The program instructions to be executed
- **Data:** Initialized global and static variables
- **BSS:** un-initialized global and static variables (filled with 0 by OS)
- **Heap:** Dynamic memory for variables that are created with **malloc**, **calloc**, **realloc** and disposed of with **free**
- **Stack**: Keeps track of the point to which each active subroutine should **return** control when it finishes executing; **stores variables** that are local to functions

Virtual Memory



Revisit: Linux Process Memory Layout

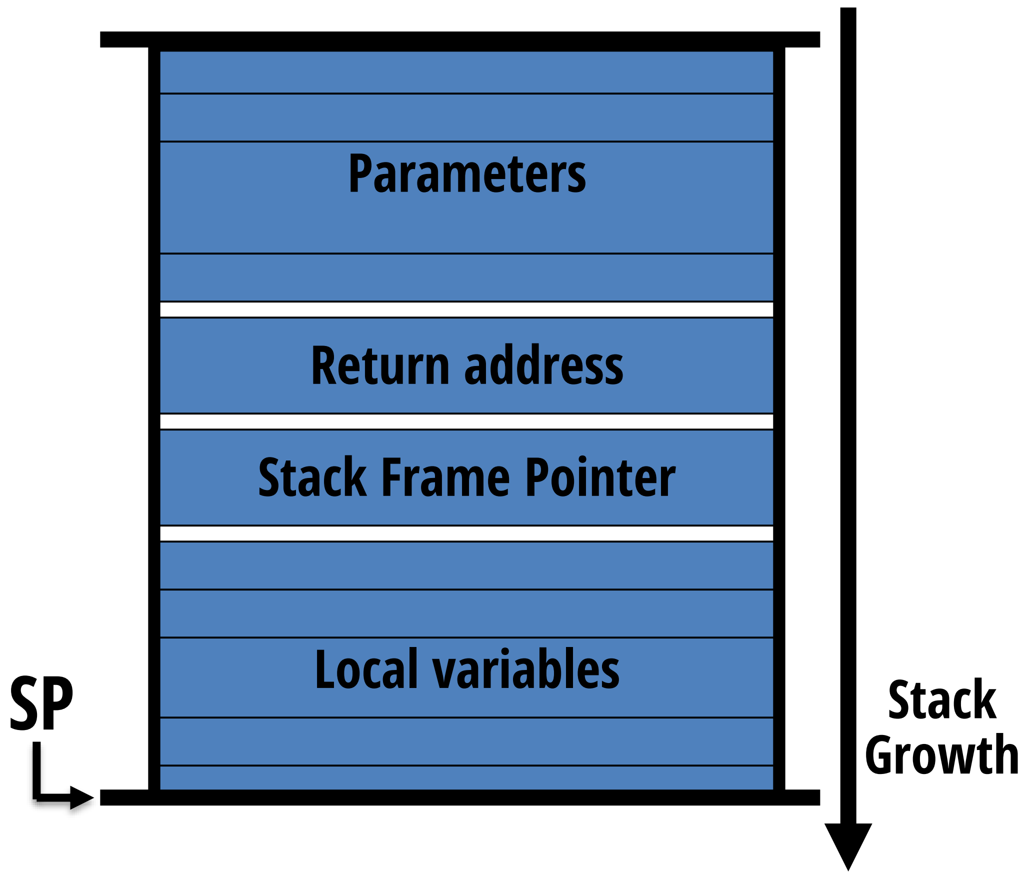


Revisit: Execution of a C Program

- **PC** (**program counter** or instruction pointer) points to next machine instruction to be executed
- **Procedure call**
  - Prepare parameters
  - Save state (**SP** (stack frame pointer) and **PC**) and allocate on stack local variables
  - Here **PC** is pointing to the **return address**
  - Jumps to the beginning of procedure being called
- **Procedure return**
  - Recover state (**SP** and **PC** (this is return address)) from stack and adjust stack
  - Execution continues from **return address**

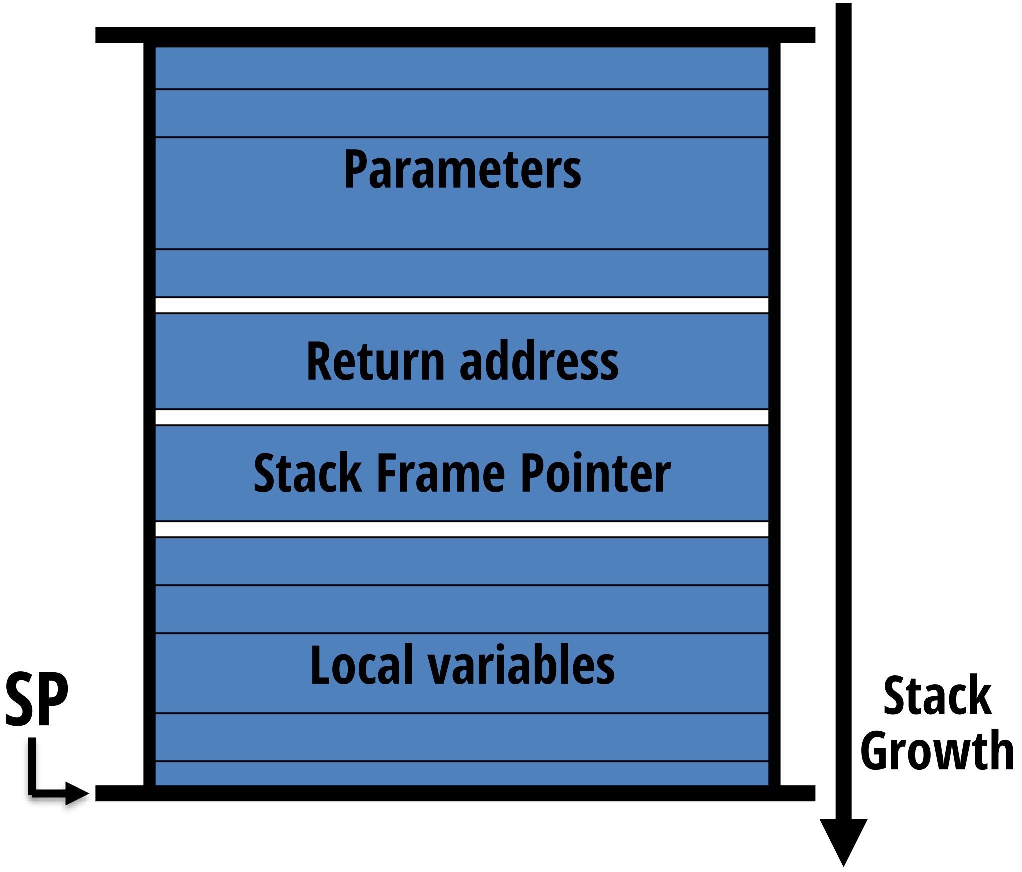
Revisit: Stack Frame

- **Parameters** for the procedure
- Save current PC onto stack (**return address**)
- Save current **SP (Stack Frame Pointer)** value onto stack
- Allocates stack space for **local variables** by decrementing SP by appropriate amount



Revisit: Stack Frame

- **Caller** : push parameter(s), push ret addr, push stack frame pointer, allocate space for locals, jmp to start addr of the callee
- **Callee**: execute the instructions, restore stack frame pointer
- **Return**: pop ret addr, jmp to addr



LIMITED DIRECT EXECUTION

How to efficiently virtualize the CPU?

- The OS needs to share the physical CPU by **time sharing**
  - Running one process, then stopping it and running another
- Issues
  - **Performance**: How can we implement virtualization without adding excessive overhead to the system?
  - **Control**: How can we run processes efficiently while retaining control over the CPU?

Direct Execution

- Just run the program directly on the CPU

OS	**Program**
1. Create entry for process list	
2. Allocate memory for program	
3. Load program into memory	
4. Set up stack with argc / argv	
5. Clear registers	
6. Execute **call main**()	
	7. Run main()
	8. Execute return from main()
9. Free memory of process	
10. Remove from process list	

Direct Execution

- There are a few problems to *direct execution* approach
  - How can OS ensure that the program does not do anything unexpected?
  - How can OS take control of the CPU from the running process?

Without **limits** on running programs,
the OS wouldn't be in control of anything and
thus would be "**just a library**"

Problem 1: Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as
  - Issuing an I/O request to a disk
  - Gaining access to more system resources such as CPU or memory

- **Solution**: Using protected control transfer
  - User mode: Applications do not have full access to hardware resources.
  - Kernel mode: The OS has access to the full resources of the machine

System Calls

- Allow the kernel to carefully expose certain key pieces of functionality to user program, e.g.,
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory

System Calls

- To execute a system call, program executes
  - **Trap** instruction
    - Jump into the kernel
    - Raise the privilege level to kernel mode
- When finished, the OS executes
  - **Return-from-trap** instruction
    - Return into the calling user program
    - Reduce the privilege level back to user mode

System Calls

- Hardware involvement
  - Trap
    - The processor pushes the process's register context onto a per-process **kernel stack**
  - Return-from-trap
    - Pop the values off the stack and resume execution

System Calls

How does the trap know which code to run inside the OS?

- Does the calling process specify where to jump?
  - No, but **why not?**
- The kernel must carefully control what code executes upon a trap
- Solution: **Trap table**
  - Specify the **location** of the **<u>trap handlers</u>**
    - what the code to jump to and execute when a *system call* or other *exceptional events* occur (e.g., interrupt)
  - At <u>boot time</u>, OS initializes the trap table and <u>informs</u> the hardware (also a *privileged* operation)

System Calls

- To specify the exact system call
  - User code cannot specify the exact address to jump to
  - Instead, it uses a **system-call number** ; that is a unique number for each system call

- Execution
  - User code places the desired syscall number to a specific *register* or memory *location*
  - OS reads the number inside trap handler
  - If valid, executes the corresponding code

- This level of indirection provides a **protection**

Limited Direct Execution Protocol (1/2)

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of … syscall handler

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from -trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() … Call system call **trap** into OS

Limited Direct Execution Protocol (2/2)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		… return from main trap (via `exit`())
Free memory of process Remove from process list		

Problem 2: Switching Between Processes

- How can the OS regain control of the CPU so that it can switch between *processes*?
  - A cooperative Approach: **Wait for system calls**
  - A Non-Cooperative Approach: **The OS takes control**

A cooperative Approach: Wait for system calls

- Processes periodically give up the CPU by making **system calls** such as `yield`.
  - The OS decides to run some other task.
  - Application also transfer control to the OS when they do something illegal.
    - Divide by zero
    - Try to access memory that it shouldn't be able to access
  - E.g., Early versions of the Macintosh OS, The old Xerox Alto system

A process gets stuck in an infinite loop.
→ **Reboot the machine**

A Non-Cooperative Approach: OS Takes Control

- **A timer interrupt**
  - During the boot sequence, the OS start the timer.
  - The timer raise an interrupt every so many milliseconds.
  - When the interrupt is raised :
    - The currently running process is halted.
    - Save enough of the state of the program
    - A pre-configured interrupt handler in the OS runs.

A **timer interrupt** gives OS the ability to run again on a CPU.

Saving and Restoring Context

- Scheduler makes a decision:
  - Whether to continue running the **current process**, or switch to a **different one**.
  - If the decision is made to switch, the OS executes context switch.

Context Switch

- A low-level piece of assembly code
  - **Save a few register values** for the current process onto its kernel stack
    - General purpose registers
    - PC
    - kernel stack pointer
  - **Restore registers** for the soon-to-be-executing process from its kernel stack
  - **Switch to the kernel stack** for the soon-to-be-executing process

Limited Direction Execution Protocol (with Timer interrupt)

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of … syscall handler timer handler	
start interrupt timer		
	start timer interrupt CPU in X ms	

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Limited Direction Execution Protocol (with Timer interrupt)

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
	(Cont.)	
Handle the trap Call switch() routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B …

The xv6 Context Switch Code

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
    # Save old registers
    movl 4(%esp), %eax          # put old ptr into eax
    popl 0(%eax)                # save the old IP
    movl %esp, 4(%eax)          # and stack
    movl %ebx, 8(%eax)          # and other registers
    movl %ecx, 12(%eax)
    movl %edx, 16(%eax)
    movl %esi, 20(%eax)
    movl %edi, 24(%eax)
    movl %ebp, 28(%eax)

```

The xv6 Context Switch Code

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
```

```
    # Load new registers
    movl 4(%esp), %eax   # put new ptr into eax
    movl 28(%eax), %ebp  # restore other registers
    movl 24(%eax), %edi
    movl 20(%eax), %esi
    movl 16(%eax), %edx
    movl 12(%eax), %ecx
    movl 8(%eax), %ebx
    movl 4(%eax), %esp   # stack is switched here
    pushl 0(%eax)        # return addr put in place
    ret                  # finally return into new ctxt
```

Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations:
  - **Disable interrupts** during interrupt processing
  - Use a number of sophisticate **locking** schemes to protect concurrent access to internal data structures.

Reading Material

- **Chapter 6** of OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin) http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf

Questions?